汇编语言程序设计

新型计算机研究所

张兴军

西一楼B段811室

Email: xjzhang@xjtu.edu.cn

计算机语言的发展

- **机器语言** *（二进制编码）*
- **汇编语言** *（符号式）*
- **高级语言**
  - **算法语言** *（接近自然语言及面向过程）*
  - **非过程化语言** *（面向对象）*
  - **智能性语言** *（具有一定智能，抽象问题求解）*

什么是汇编语言?

- **机器语言：机器指令的集合。**以二进制形式的指令组成的指令集合，它**是计算机惟一能够直接识别和处理的语言**

  例如：1000100111011000；将寄存器BX的内容送到寄存器AX。

  缺点：编写、阅读、改错很不方便

  机器语言描述的程序称为目标程序（可执行程序），只有目标程序CPU才能直接执行

- **汇编语言：机器语言的符号化表示。**面向机器的语言，用简单且容易记忆的符号（助记符号）来代替机器语言中“0”、“1”的一种程序设计语言

  例如：　机器语言指令　　汇编语言指令

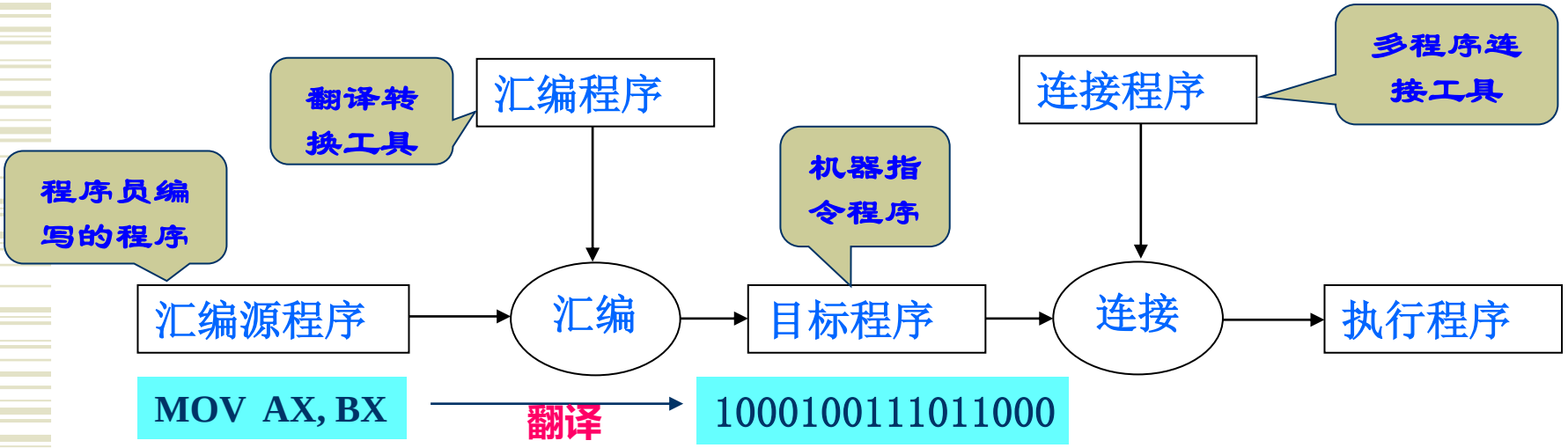
  1000100111011000　　MOV　AX,BX

  汇编语言的执行语句与机器语言的指令是一对一的关系

  现代计算机中使用汇编语言主要是为了某些情况下直接控制计算机硬件以获得高级语言无法实现的功能和性能

- 用汇编语言编写程序执行过程

汇编语言建立在机器语言之上，由于计算机不能够直接识别这种符号语言，所以汇编语言编写的源程序必须用汇编程序直接翻译成机器语言后才能执行



为什么要学习汇编语言?

- 理解计算机的基本系统结构
- 能够学习到处理器是如何工作的
- 探究数据和指令的内部表述
- 能够创建小巧有效的程序
- 允许程序员绕过高层语言的限制编程
- 有些工作必须用汇编语言完成

编写嵌入式程序、实时应用、编写驱动程序等

汇编语言程序设计与后续课程的关系

- 硬件课程：
  - 微机原理与接口技术：使用汇编语言编写硬件驱动控制程序等
  - 计算机组成原理：如何用硬件实现机器指令，即汇编语言指令功能
  - 计算机系统结构：CPU内部结构、机器指令、存储系统和IO系统优化设计、评价方法
- 软件课程：
  - 编译原理：如何将 高级语言程序模块 通过优化编译转换成 机器指令程序模块
  - 操作系统：基于汇编语言设计实现对计算机系统资源管理、任务管理
  - 高级语言程序设计如何使用汇编语言程序对低层的直接控制

使用教材及其他

- **教材、主要参考书籍**
  - 教材：《80X86汇编语言程序设计》，沈美明，温冬婵 编著，清华大学出版社
  - 主要参考书籍：
    - 《汇编语言》（第2版），王爽著，清华大学出版社
    - 《Intel汇编语言程序设计》，Kip R.Irvine 著，温玉杰 等译，电子工业出版社
    - 《IBM PC Assembly Language and Programming》, Peter Abel, (影印版，清华大学出版社)
- **FTP课件下载**
  - 公共邮箱：huibianyy@163.com； 密码：huibian
- **答疑**
  - 线上班级QQ群，线下办公室（上班时间)
  - 地点： 西一楼B段811房间 （张兴军）
    西一楼A段413房间 （董小社）
    西一楼A段415房间 （陈　衡）

课程计划安排

- **课内基础知识讲授：32学时**
  - 第一章：基础知识 …… 2学时
  - 第二章：80x86计算机组织结构 … 4学时
  - 第三章：指令系统 …… 8学时
  - 第四章：汇编语言程序格式 …… 4学时
  - 第五章：循环与分支程序设计 …… 3学时
  - 第六章：子程序结构 …… 3学时
  - 第七章：高级汇编语言技术 …… 4学时
  - 第八章：输入输出程序设计 …… 2学时
  - 第九章： BIOS和DOS中断 …… 2学时
- **课内上机实验：16学时**
  - 第1次: 待定
  - 第2次: 待定
  - 第3次: 待定
  - 第4次: 待定

考核方式：

考试成绩 60%

平时作业 10%

平时表现 15%

上机成绩 15%

本课程学习方法

1. 熟练掌握计算机硬件组织结构、功能和工作原理
   - 80X86处理器：寄存器、运算器、PSW 等工作原理
   - 存储器（内存）：数据存储和访问方式
   - 输入/输出：工作原理和控制方式
2. 熟记常用指令、DOS功能中断调用、数据表示
3. 掌握非常用指令、数据表示方式
4. 学习汇编语言程序设计方法
5. 在实践中总结、创新编程优化技巧

注：汇编语言程序设计课程要求掌握如何使用硬件资源；

计算机组成原理课程要求掌握如何设计实现硬件；

计算机系统结构课程要求掌握硬件优化设计和评价方法

第1章　基础知识

1.1 进位计数制与不同基数的数之间的转换

1.2 二进制数和十六进制数运算

1.3 计算机中数和字符的表示

1.4 几种基本的逻辑运算

本章目标

- **掌握**
  - 常用的各种进制数的表示和运算
  - 常用的各种进制数的数制之间的转换规则
  - 带符号数的**补码**表示方法和补码的运算
- **熟悉**
  - 符号扩展的概念
  - 数据的表数范围
- **了解**
  - 了解计算机存取信息的基本数据类型
  - 了解计算机中字符的表示

1.1 计算机中的数

1. 汇编程序中常用的数制表示

二进制数：计算机硬件唯一识别和使用的数制

以2为基的数制表示法，数由2个数字构成（0、1），二进制数后缀为B，如10110111B。

十进制数：人类自然语言中常用的数制

以10为基的数制表示法，数由10个数字构成（0~9），十进制数后缀为D，如1945D。

十六进制数：程序设计中方便使用和转换的数制

以16为基的数制表示法，数由16个数字构成[0~9、A（10）、B（11）、C（12）、D（13）、E（14）、F（15）]，十六进制数后缀为H，如18ADH。

十进制数的特点

(1) 基数为10：有10个不同的数字符号

0、1、2、3、4、5、6、7、8、9

(2) 逢10进位/借1当10：由于十进制数是逢10进位的，因此同一个数字符号在不同的位置（数的排列先后）代表的数值是不同的

十进制数的特点

(3) 一般表达式

$$D = D_{n-1} \cdot 10^{n-1} + D_{n-2} \cdot 10^{n-2} + \cdots + D_1 \cdot 10^1 + D_0 \cdot 10^0 + D_{-1} \cdot 10^{-1}$$
$$+ D_{-2} \cdot 10^{-2} + \cdots D_{-m} \cdot 10^{-m}$$
$$= \sum_{i=-m}^{n-1} D_i \cdot 10^i$$

在此处键入公式。例如: **666.66D**

- 小数点左边第1位数“6”位于个位，它的值就是6本身
- 小数点左边第2位数“6”位于十位，它的值就是6×10=60
- 小数点左边第3位数“6”位于百位，它的值就是6×100=600
- 小数点右面第1位数“6”位于十分位，它的值是$6\times10^{-1}=0.6$，小数点右面第2位数“6”位于百分位，它的值是$6\times10^{-2}=0.06$，……

二进制数的特点

特点：

(1)基数是2：具有两个不同的基本符号0、1

(2)逢2进1，借1当2

二进制一般表达式：

$$B_{n-1}\cdot 2^{n-1}+B_{n-2}\cdot 2^{n-2}+\cdots+B_1\cdot 2^1+B_0\cdot 2^0+B_{-1}\cdot 2^{-1}$$
$$+B_{-2}\cdot 2^{-2}+\cdots B_{-m}\cdot 2^{-m}$$

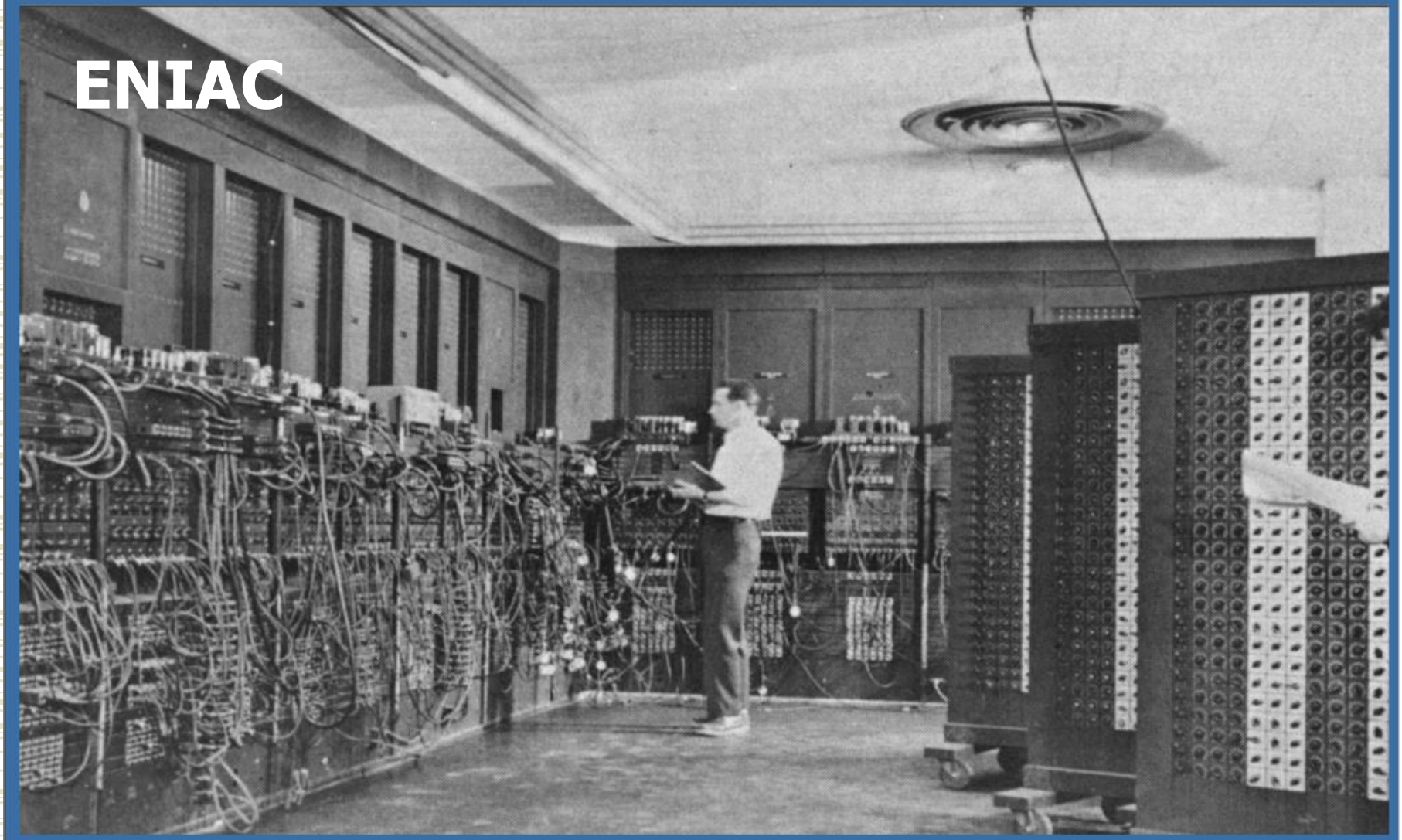
二进制与十进制关系

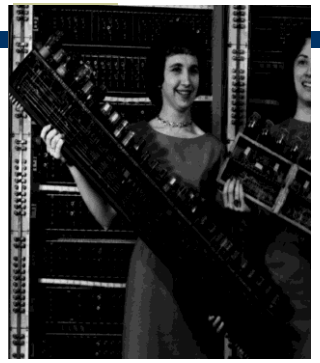
$$D=\sum_{i=-m}^{n-1}B_i\cdot 2^i$$

为什么计算机中采用二进制数

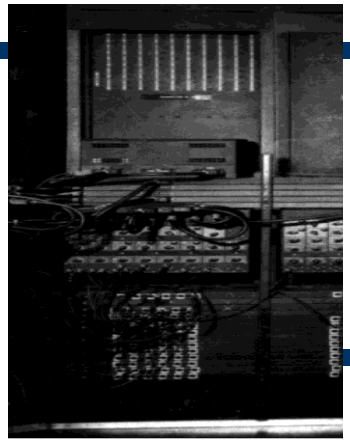
- **便于存储及计算的物理实现：**二进制数各位上的数码只有0和1两种取值，用计算机内部的电路实现时（高电压、低电压），比十进制数要**方便**的多
- **工作稳定可靠：**抗干扰能力强
- 实际计算机电子线路中
  - 例如：用+5V或+3.3V表示1，用0V表示0
- 若干位排列起来就可表示一个二进制数

the Modern Computer Age
1946

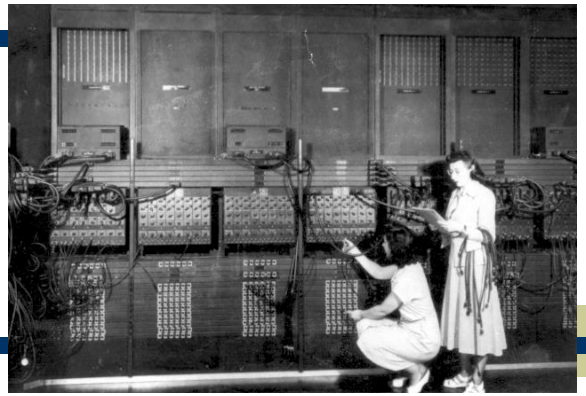




Decade Counter



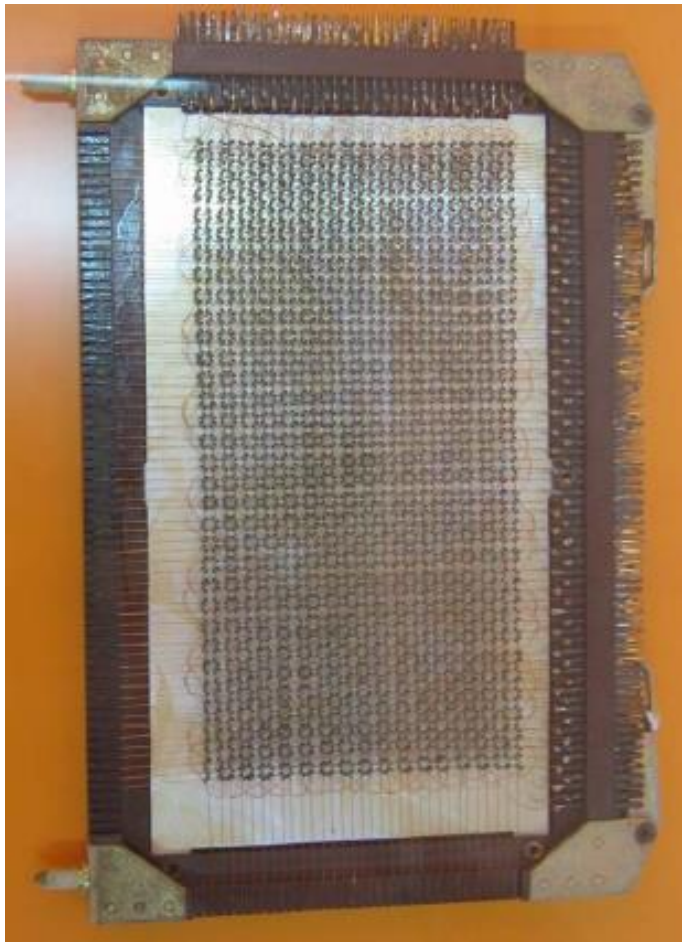
Accumulator



“Wiring” a Program

- 18000 tubes
- 1500 relays
- 174 KW
- 30 tons
- 1800 sq. ft. footprint
- Clock: 100 kHz
- IO: punched cards

早期计算机部件



早期计算机IO

二进制计量单位

- ***比特***：bit，或称位元，简称位，0或1；以“b”表示；最小单位
- ***字节***：byte，位组，8个bit；以“B”表示，一个字符用一个字节表示
- ***字***：word，一般情况下指2个字节
- $2^8=256$, $2^{10}=1024=1K$, $2^{16}=64K$, $2^{20}=1M$, $2^{24}=16M$, $2^{30}=1G$, $2^{32}=4G$, …
- ***1KB=1024B***，***1MB=1024KB***，***1GB=1024MB***；***1TB=1024GB***

几种常用进位记数制的基数和数码

数　　制	基 数	各位数码表示
二进制 Binary	2	0,1
八进制 Octal	8	0,1,2,3,4,5,6,7
十进制 Decimal	10	0,1,2,3,4,5,6,7,8,9
十六进制 Hexadecimal	16	0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F

十进制数 、二进制数、八进制数和十六进制数之间的对应关系

十进制(D)	二进制(B)	八进制(O)	十六进制(H)
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

为什么需要十六进制数

- 便于编程与阅读：二进制的阅读、书写和记忆不方便
  - 用十进制数?
    - 十进制与二进制无直接对应关系，转换困难
    - 如果不转换，用BCD码(Binary Code Decimal，二进制编码的十进制数）表数效率太低，处理困难
      - 0000～1001分别表示8421 BCD码的0～9
      - 1010～1111没有用，浪费6个编码，即37.5%
  - 2^n作为基数的数制转换方便、处理方便，表数效率最高
    - 八进制(000～111)、十六进制(0000～1111)
- 十六进制数与计算机中数据存储、处理的单位长度适用
  - 基本单位：一个二进制位（bit)
  - 常用字符单位：8位二进制数组成的一个字节（byte)
  - 计算机字长：字节的整数倍，8位、16位 (字，Word) 、32位、64位二进制数
  - 1位十六进制数对应4位二进制数
  - 1位八进制数对应3位二进制数

2.不同进位计数制之间的数据转换

- **CPU处理的是二进制数**
  - 写程序时，需要把十进制、二进制数转换为十六进制数
  - 阅读程序时，需要将二进制、十六进制转换为十进制数

- **数据转换：** 把一种进制数转换为另一种进制的数，其实质是进行基数的转换。基数转换是依据两个有理数相等，其整数部分与小数部分分别相等的原则。

- **转换方法：** 转换时，其整数部分与小数部分应分别进行转换，将转换后的结果合并，整数部分与小数部分之间用小数点隔开，就得到相应的转换结果。

(1) 二进制数转换为十进制数的转换规则是“按权值相加”。也就是说，只要把二进制数中数位是“1”的那些位的权值相加，其和就是等效的十进制数。

$$D = \sum_{i=-m}^{n-1} B_i \cdot 2^i$$

$$1101B = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 13D$$

(2) 十六进制转换为十进制数的转换规则

$$D = \sum_{i=-m}^{n-1} h_i \cdot 16^i$$

(2) 十进制数转换为二进制数

整数和小数部分分别进行转换，转换结束后将整数转换结果写在左边，小数转换结果写在右边，中间点上小数点。

- 两种基本方法：降幂法、除法

十进制数转换为二进制数

方法1：降幂法

$$D = B_{n-1} \cdot 2^{n-1} + B_{n-2} \cdot 2^{n-2} + \cdots + B_1 \cdot 2^1 + B_0 \cdot 2^0 + B_{-1} \cdot 2^{-1} + B_{-2} \cdot 2^{-2} + \cdots B_{-m} \cdot 2^{-m}$$

整数部分转换规则：

- 写出要转换的十进制数
- 写出所有小于此数的二进制各位权值
- 十进制数减去二进制权值，权值由大到小
  - 如够减，相应二进制位记1；如不够减，相应二进制位记0
- 不断反复，直到该数为0

小数部分的转换规则：

同整数部分

例： 27D = ? B

27 11 3 3 1

-↓ -↓ -↓ -↓ -↓

16 8 4 2 1

1 1 0 1 1

∴ 27D = 11011B

十进制数转换为二进制数

方法2：除法

$$D = B_{n-1} \cdot 2^{n-1} + B_{n-2} \cdot 2^{n-2} + \cdots + B_1 \cdot 2^1 + B_0 \cdot 2^0 + B_{-1} \cdot 2^{-1} + B_{-2} \cdot 2^{-2} + \cdots B_{-m} \cdot 2^{-m}$$

整数部分转换规则：

- 将十进制整数用基数2连续去除，直到商为0为止；
- 将每次除得的余数反向排列，就可得到十进制数整数部分的转换结果。
- 反向排列是指最后得到的余数排在前边，作为结果的最高位，最先得到的余数排在后边，作为结果的最低位

小数部分的转换规则：

- 将十进制数的小数部分用基数2连续去乘，直到小数部分为0或达到精度为止；
- 将每次所得的乘积的整数部分正向排列，就可得到十进制小数的转换结果。
- 正向排列是指最先得到的整数为结果的最高位，最后得到的整数为结果的最低位

例 N = 117.8125D

$$D = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + b_0 \cdot 2^{-1} + b_1 \cdot 2^{-2} + \cdots b_m \cdot 2^{-m}$$

- 整数部分：117D

117/2=58　　a_0=1

58/2=29　　a_1=0

29/2=14　　a_2=1

14/2=7　　a_3=0

7/2=3　　a_4=1

3/2=1　　a_5=1

1/2=0　　a_6=1

117D = 1110101B

- 小数部分：0.8125D

0.8125×2=1.625　　b_0=1

0.625×2=1.25　　b_1=1

0.25×2=0.5　　b_2=0

0.5×2=1.0　　b_3=1

0.8125D = 0.1101B

N = 117.8125D = 1110101.1101B

(3) 二进制数转换为(八进制)十六进制数

- 将二进制数以小数点为界，向左、向右分别按（3位）4位一组划分，不足（3位）4位的部分用“0”补足（整数部分左补0，小数部分右补0），将每一组数写成一位对应的（八进制）十六进制数，就可得到转换结果。

例如：1110101.11B=？H

0111　0101 . 1100
7　　5 . C

∴ 1110101.11B=75.CH

(4) （八进制）十六进制数转换为二进制数

- 将十六进制数以小数点为界，向左、向右分别展开为（3位）4位二进制数，就可得到转换结果。

例如：EA.11H=？B

E	A	.	1	1
1110	1010		0001	0001

∴ EA.11H=11101010.00010001B

1.2 二进制数和十六进制数运算

- 二进制数加法/减法规则

  二进制加法规则：

  0 + 0 = 0

  1 + 0 = 1

  0 + 1 = 1

  1 + 1 = 0——向高位进位为1（逢2进1）

  二进制减法规则：

  0 – 0 = 0

  1 – 0 = 1

  0 – 1 = 1 ——向高位借1（借1当2）

  1 – 1 = 0

- 十六进制数
  - 自学

```
  1 0 1 1 0 1 1 1.1 1
+ 0 0 0 0 0 0 0 1.0 0
-------------------
  1 0 1 1 1 0 0 0.1 1
```

课内测试使用说明（1/6）

1. 登录思源学堂 http://syxt.xjtu.edu.cn

课内测试使用说明（2/6）

2. 点击左侧栏目的 **课内测试**

3. 点击 **课内测试01-1-1**

课内测试使用说明（3/6）

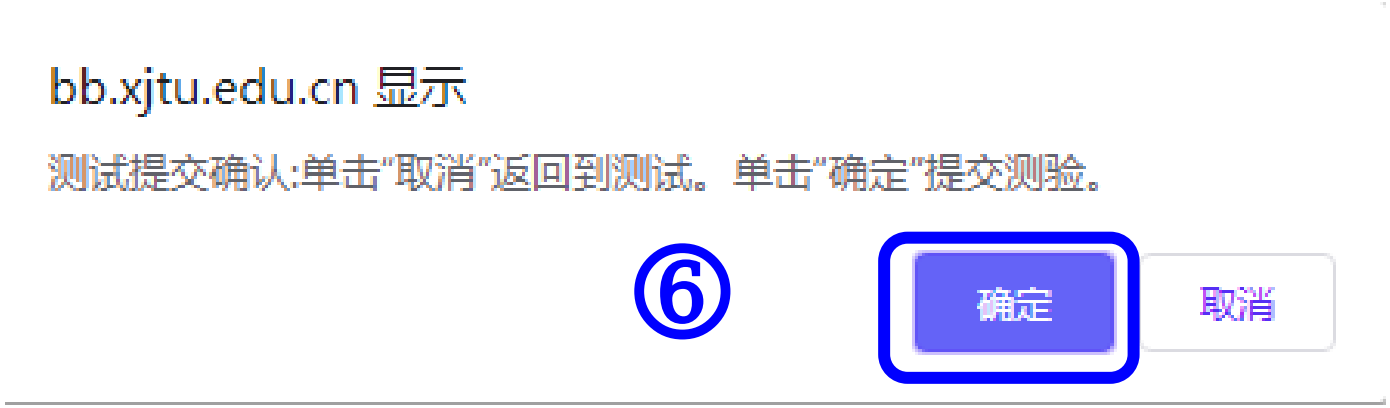
4. 点击 开始，启动**倒计时5分钟**答题

课内测试使用说明（4/6）

5. 完成后，点击 保存并提交

课内测试使用说明（5/6）

6. 提交前最后确认，点击 **确定**



课内测试使用说明（6/6）

7. 提交后信息显示，点击 **确定**可查询得分情况及正确答案

课内测试 01-1-1

- 【多项填空题】将十进制数转换为二进制数常用的方法是_____和_____。

1.3 计算机中数和字符的表示

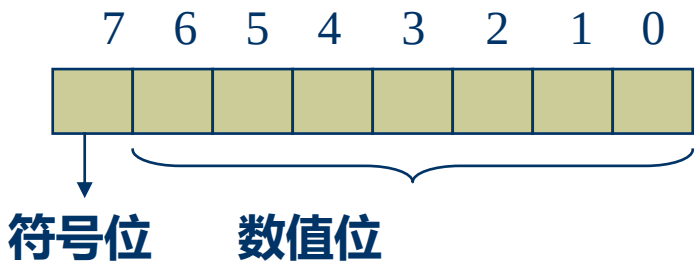
1.3.1 数的补码表示

1.3.2 补码的加法和减法

1.3.3 无符号整数

1.3.4 字符表示法

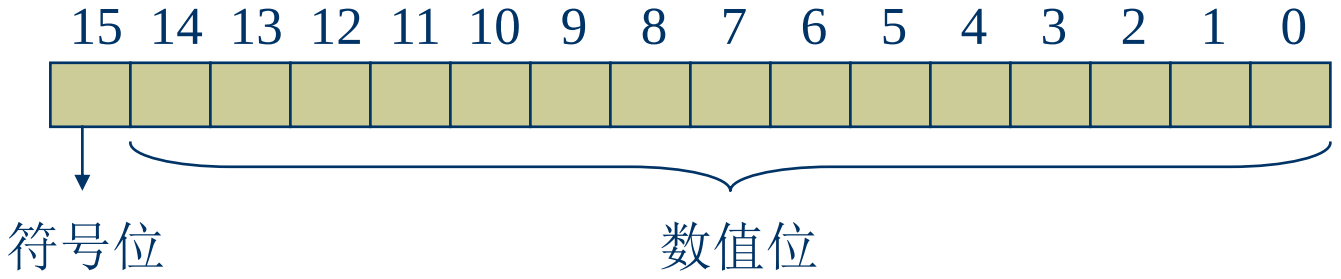
- **数（机器数）的表示：**
  - **计算机中的数用二进制表示，数的符号也用二进制表示。**
  - **机器数：数连同其符号在内数值化表示。**
  - **机器字长：指参与运算的数的基本位数，标志着计算精度，一般是字节的整数倍（8位、16位、32位、64位等）。**
  - **假设机器字长n为8位**



符号位=0 表示正数

符号位=1 表示负数

- **假设机器字长n为16位**



- **机器数常用表示法 — 原码, 反码, 补码**

(Sign-Magnitude, Ones‘ complement, Two's complement)

1.3.1 数的补码表示法

1.原码表示法

（1）原码表示法： 将数的真值形式中的正（负）号，用代码0(1)来表示，数值部分用二进制来表示。符号 + 绝对值

正数：符号位为0，后面的n-1位为数值部分

负数：符号位为1，后面的n-1位为数值部分

（2）原码的特点

- “0” 的原码有两种表示法

  $[+0]_{原}$ = 00000000B， $[-0]_{原}$ = 10000000B

- n位二进制原码所能表示的数值范围为： $-(2^{n-1}-1) \sim (2^{n-1}-1)$
- 原码表示一个数时，最高位为符号位

例：n=8bit

$[+3]_{原码}$ = 0 000,0011 = 03H

$[-3]_{原码}$ = 1 000,0011 = 83H

$[+0]_{原码}$ = 0 000,0000 = 00H

$[-0]_{原码}$ = 1 000,0000 = 80H

0的表示不唯一

2. 反码表示法

- 正数的反码同原码，负数的反码数值位与原码相反
- 例：n=8bit

$[+5]_{反码}$ = 0 000,0101 = 05H

$[-5]_{反码}$ = 1 111,1010 = FAH

$[+0]_{反码}$ = 0 000,0000 = 00H

$[-0]_{反码}$ = 1 111,1111 = FFH

0的表示不唯一

3. 补码表示法

（1）补码表示规则：

- **正数的补码: 符号 - 绝对值（与正数的原码相同）**

  $[+1]_{补码}$ = 0000 0001 = 01H

  $[+127]_{补码}$ = 0111 1111 = 7FH

  $[+0]_{补码}$ = 0000 0000 = 00H

- **负数的补码: 负数 X 用 2^n-|X| 表示**

  $[-1]_{补码}$ = 2^8-1 = 1111 1111 = FFH

  $[-127]_{补码}$ = 2^8-127 = 1000 0001 = 81H

  一种简单方法：

  （1）写出与该负数相对应的正数的补码

  （2）按位求反

  （3）末位加1

[-1]补=?

[+1]补= 0000 0001

1111 1110

[-1]补= 1111 1111

例： 机器字长8位，$[-46]_{补码}$ = ？

$[+46]_{补码}$ = 0010 1110

按位求反

1101 0001

末位加1

$[-46]_{补码}$ = 1101 0010 = D2H

补码的符号扩展问题

- 指一个数从位数较少扩展到位数较多时应该注意的问题
  - 8位扩展到16位，16位扩展到32位，等
- 补码表示的扩展规则
  - 正数：前边补0
  - 负数：前边补1

机器字长8位，$[-46]_{补码}$ = 1101 0010B = D2H

机器字长16位，$[-46]_{补码}$ = 1111 1111 1101 00010B = FFD2H

(2) 补码的特点

- “0” 的补码表示是唯一的

  $[+0]_{补}=[-0]_{补}=00000000B$

- 补码运算时符号位无需单独处理
- 采用补码运算时，减法可用加法来实现

$[13-10]_{补}=[13]_{补}+[-10]_{补}=00001101+11110110=00000011B=[+3]_{补}$

(有进位，自动丢失，符号位为0结果为正)

$[10-13]_{补}=[10]_{补}+[-13]_{补}=00001010+11110011=11111101B=[-3]_{补}$

(无进位，符号位为1，结果为负)

- 符号扩展问题简单

补码的表数范围

n位补码的表数范围： $-2^{n-1} \le N \le 2^{n-1}-1$

n=8　　$-128 \le N \le 127$　　2^8个数

n=16　$-32768 \le N \le 32767$　　2^{16}个数

比原码和反码多表示一个数，表数效率也高一点

1.3.2 补码的加法和减法

求补运算：对一个二进制数按位求反后在末位加1的运算

补码表示的数具有以下特点：

$$[X]_{补} \xrightarrow{求补} [-X]_{补} \xrightarrow{求补} [X]_{补}$$

$$[117]_{补} \xrightarrow{求补} [-117]_{补} \xrightarrow{求补} [117]_{补}$$

补码加法和减法的规则

$$[x + y]_{补} = [x]_{补} + [y]_{补}$$

$$[x - y]_{补} = [x]_{补} - [y]_{补}$$

$$= [x]_{补} + [-y]_{补}$$

- 补码减法可转换为补码加法
- 不必判断数的正负，符号位一起参加运算，能自动得到正确结果

举例说明补码运算

已知： $X = -27D$

$Y = +29D$

求： $[X + Y]_{补} = ?$

$[X - Y]_{补} = ?$

解：运算步骤

求$[X + Y]_{补}$:

1）求出$[X]_{补}$，$[Y]_{补}$

$[X]_{补}$ = 11100101B ， $[Y]_{补}$ = 00011101B

2）求出 $[X + Y]_{补}$

$[X + Y]_{补}$= $[X]_{补}$ + $[Y]_{补}$

= 11100101 + 00011101

= 00000010B

求$[X - Y]_{补}$：

$[X]_{补}$ = 11100101B， $[-Y]_{补}$ = 11100011B

1）$[X - Y]_{补} = [X]_{补} - [Y]_{补}$

= 11100101 - 00011101

= 11001000B

2）$[X - Y]_{补} = [x]_{补} + [-y]_{补}$

= 11100101 + 11100011

= 11001000B

课内测试 01-1-2

- 【填空题】78H的二进制表示是____。

1.3.3 无符号整数

- 当程序处理的数全是正数时,保留符号位就没有意义了,此时应该采用无符号数表示
- 表数范围

  16位无符号数的表示范围

  $0<=N<=2^{16}-1=65535$

  8位无符号数的表示范围

  $0<=N<=2^{8}-1=255$

- 一般用途：表示地址的数，双精度数的低位字

表数范围

表数范围：给出n位二进制整数的表数范围

（1）无符号整数的范围

8位二进制数所能表示的无符号整数的范围是

0 ~ 255；　2^8=256个数

16位二进制数所能表示的无符号整数的范围是

0 ~ 65535；　2^{16}=65536个数

n位二进制数N能够表示的无符号数的范围

0 ~ (2^n-1)；　2^n个数

表数范围

（2）采用补码表示的带符号整数的表数范围

8位二进制数所能表示的带符号整数的范围是

-128 ~ 127；

16位二进制数所能表示的带符号整数的范围是

-32768 ~ +32767；

n位二进制数N能够表示的带符号数的范围

$-2^{n-1} \sim 2^{n-1}-1$

n位二进制补码的表数范围

十进制	二进制	十六进制	十进制	十六进制
	n=8		n=16	
+127	0111 1111	7F	+32767	7FFF
+126	0111 1110	7E	+32766	7FFE
…	…	…	…	…
+2	0000 0010	02	+2	0002
+1	0000 0001	01	+1	0001
0	0000 0000	00	0	0000
-1	1111 1111	FF	-1	FFFF
-2	1111 1110	FE	-2	FFFE
…	…	…	…	…
-126	1000 0010	82	-32766	8002
-127	1000 0001	81	-32767	8001
-128	1000 0000	80	-32768	8000

BCD 码

- 十进制数在机器中通常采用BCD码表示，而字符及字符串通常用ASCII码表示
- BCD码是一种用二进制编码的十进制数，即用4位二进制形式（0000B-1001B）来表示一位十进制数（0-9），但每4位二进制数之间的进位又是十进制的形式

579 D = 010101111001 BCD

BCD码
- 压缩的BCD码（用4位二进制表示）
- 非压缩的BCD码（用8位二进制表示，前面4位为0000）

1.3.4 字符表示法

- 计算机处理的信息并不全是数,有时需要处理字符或字符串。又因为机器中只能存储二进制数，所以字符在机器里必须用二进制数来表示。一般采用目前最常用的**美国信息交换标准代码**（ASCII：American Standard Code for Information Interchange）来表示
- **ASCII码：**用一个字节来表示一个字符，其中7位为字符的ASCII码值，最高位一般用作校验位。

常用字符的ASCII码

字符	ASCII码
0～9	30H~39H
A～Z	41H~5AH
a～z	61H~7AH
回车CR	0DH
换行LF	0AH
$	24H
空格(SPACE)	20H

目前常用输入输出设备显示器、打印机、键盘等均采用ASCII码

1.4 几种基本的逻辑运算

1.4.1 AND “与”运算

1.4.2 OR “或”运算

1.4.3 NOT “非”运算

1.4.4 XOR “异或”运算

逻辑变量: 只能有0或1两种取值

1.4.1 AND “与”运算

A	B	F=A∧B
0	0	0
0	1	0
1	0	0
1	1	1

1.4.2 OR “或”运算

A	B	F=A ∨B
0	0	0
0	1	1
1	0	1
1	1	1

1.4.2 NOT “非”运算

A	$F=\bar{A}$
0	1
1	0

1.4.4 XOR “异或”运算

A	B	F=A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

作 业

思源学堂→课后作业→第1次课后作业

要求：

（1）独立完成，**严禁抄袭**

（2）以word文档在思源学堂提交

（3）截至日期：4月20日23:59

汇编语言的特点

1.汇编语言相对于机器指令易于编程、阅读等

2.汇编语言与机器指令一一对应，可充分理解计算机的操作过程

有助于理解指令在机器中的执行过程

3.汇编语言是靠近机器的语言

可以充分利用机器硬件的全部功能，发挥机器的特点

4.汇编语言效率高于高级语言

目标程序的高效率反映在运行速度（时间）、目标代码短（空间）

高级语言

- 高级语言包括面向过程的语言和面向对象的语言，是更接近人类语言（英语）和数学语言的计算机语言。
- 面向过程语言的出现为程序设计带来了方便，要求写出每个任务完成的一系列明确过程。 （C、 BASIC、 Pascal、FORTRAN）
  - 一般按指令顺序执行
- 面向对象语言是从面向过程语言发展而来的，它改变了编程者的思维方式，采用与人们认识世界相同的方法，将问题分解成若干对象和对象间的相互作用。 （c++、java）
  - 可乱序执行，只要资源或条件满足就可执行，提高资源利用率和程序效率
  - 更适合现代计算机中的多CPU结构和多线程技术等
  - 使程序设计更接近于自然语言，设计出来的软件质量更高

汇编语言与高级语言的比较

1.程序特点

高级语言 - 面向问题：：汇编语言 - 面向机器

2.软件开发

高级语言：节省开发时间，但不允许程序员直接使用微处理器的许多特性（寄存器、标志、内存等）

汇编语言：编程比较难，但可充分发挥机器特性的作用

3.代码生成

高级语言：编译后目标代码程序比较长，效率低

汇编语言：目标代码程序短，执行速度快，占用内存少

第二章　80X86计算机组织

2.1 80X86微处理器概况

2.2 基于微处理器的计算机系统构成

2.3 中央处理机

2.4 存储器

2.5 外部设备

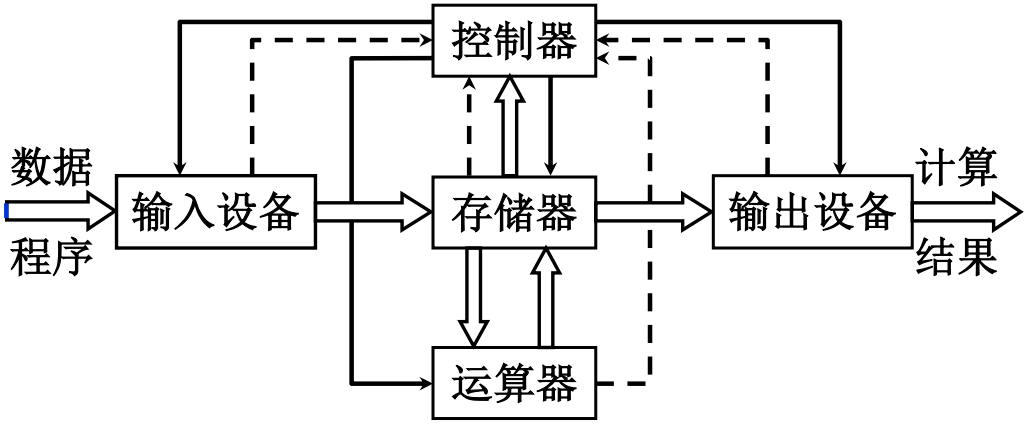
本 章 目 标

- 了解80X86微处理器
- 熟悉基于微处理器的计算机系统构成
- 熟练掌握80X86CPU的寄存器组功能和作用
- 掌握存储器地址的分段表示及其物理地址的计算
- 熟悉段应用规定

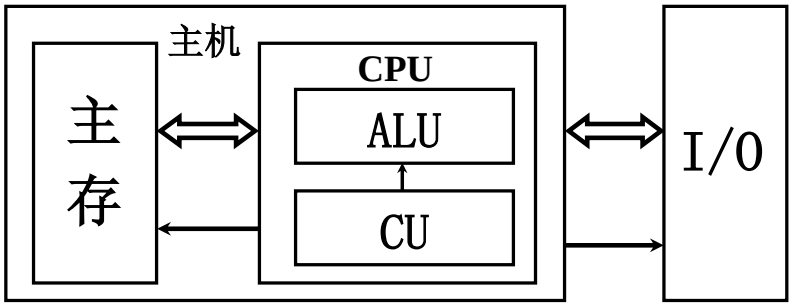
计算机结构

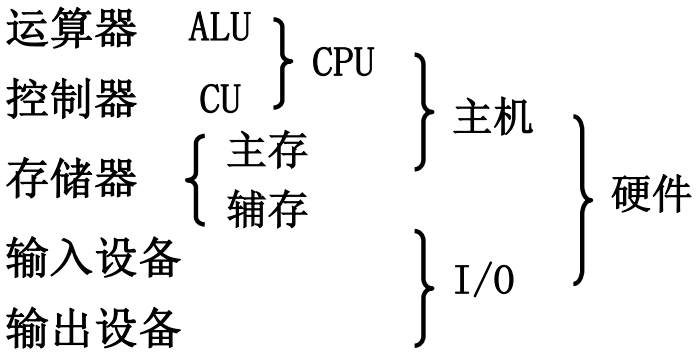
- **以存储器为中心的计算机结构**

计算机系统主要由存储器、运算器+控制器、输入设备和输出设备构成

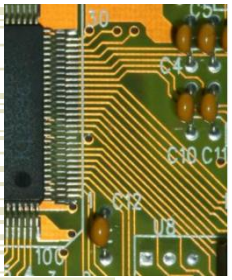


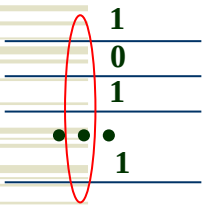
- **现代计算机硬件组成**



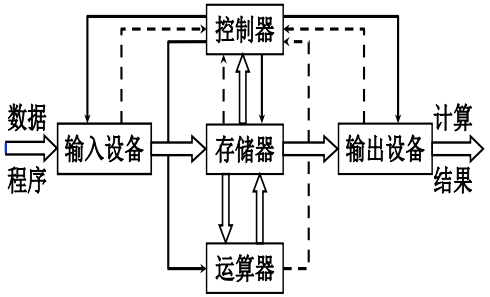


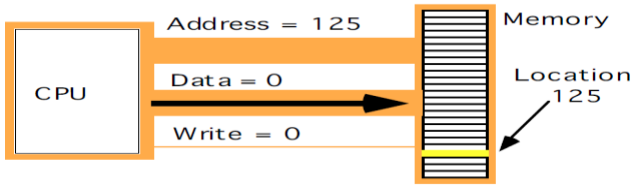
面向总线的体系结构



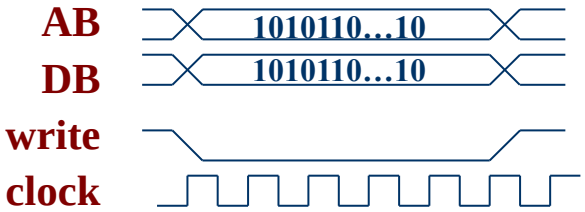


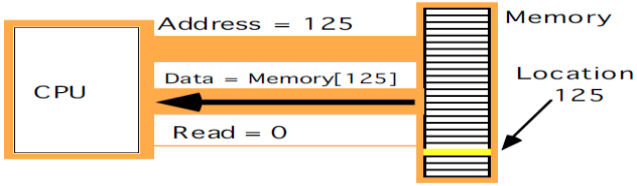
- 计算机五大部件互连方式
  - 分散连接：各部件之间使用单独的连线
  - 总线连接：各部件连接到一组公共信息传输线
  - 从分散连接→总线连接 （I/O设备与主机之间灵活连接）
- 系统总线
  - 数据总线（Data Bus）
    - 各部件之间数据传输
    - 内部数据总线宽度：CPU芯片内部数据传送的宽度（位数）
    - 外部数据总线宽度：CPU与外部交换数据时的数据宽度
  - 地址总线（Address Bus）
    - 传输地址 （Which memory location or I/O devices?）
  - 控制总线（Control Bus）
    - 控制CPU和其他部件的通信方式等 （Is sending or receiving?）



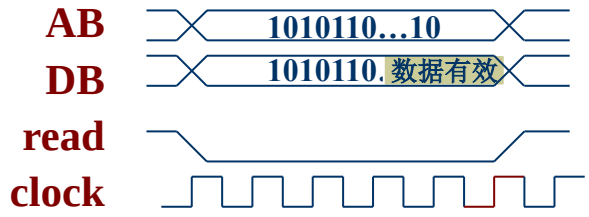


内存写操作　MOV [125], AX





内存读操作　MOV AX, [125]



2.1 80X86微处理器

- 20世纪70年代初期，由于大规模集成电路技术的发展 ，已经开始把运算器和控制器集成在一块芯片上，构成中央处理器CPU（Central Processing Unit），80X86是由Intel公司开发的微处理器系列
  - 由80X86微处理器芯片构成的微机称为X86微机
  - 另外还有AMD公司微处理器系列、IBM公司POWERPC、SUN公司SPARC等
  - 各种CPU系列有自己的机器指令集，互不兼容
    - 高级语言：C, PASCAL, FORTRAN, JAVA等高级语言，与机器指令不对应，但对各CPU兼容，因为通过编译转换为对应的机器指令
    - 汇编语言：与机器指令一一对应，对各CPU不兼容，但汇编语言程序设计方法通用，助记符、程序结构大体相同
- 本课程以80X86微处理器为例讲解汇编语言程序设计基本概念、原理、方法

Intel公司处理器系列

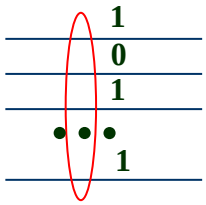
- 1971年，设计了第一片4位微处理器Intel 4004, 随之又设计生产了8位微处理器8008
- 1981年推出了8080；1974年基于8080的个人计算机（PC）问世
- 1977年Intel推出了8085
- 自此之后，又陆续推出了8086、80286、80386、80486、Pentium、XEON、EMT’64、Itanium2、多核等80X86系列微处理器


1.E+9
1.E+8
1.E+7
1.E+6
1.E +5
1.E+4
1.E+3
'70
'74
'78
'82
'86
'90
'94
'98
'2002
"Itanium":15,950,000
Pentium II: 7,500,000
PowerPC620:6,900,000
PentiumPro: 5,500,000
Pentium:3,300,000
PowerPC604:3,600,000
PowerPC601:2,800,000
i80486DX:1,200,000
m68040:1,170,000
i80386DX:275,000
m68030:273,000
i80286:134,000
m68020:190,000
m68000:68,000
i8086:28,000
M6800:
4,000
i8080:6,000
i4004:2,300

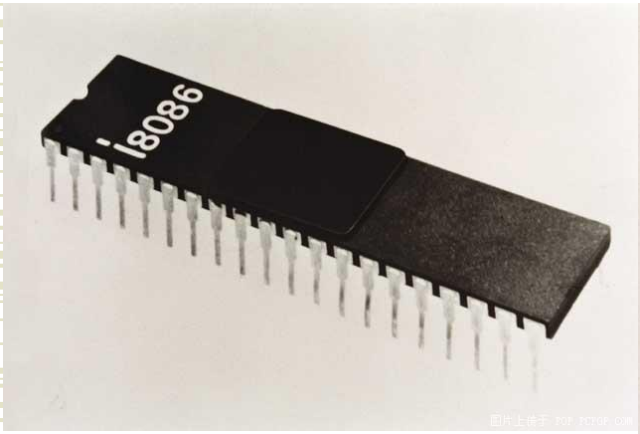
80X86微处理器概况

这里仅列出发布年份、字长、主频、地址总线宽度、寻址空间和高速缓冲

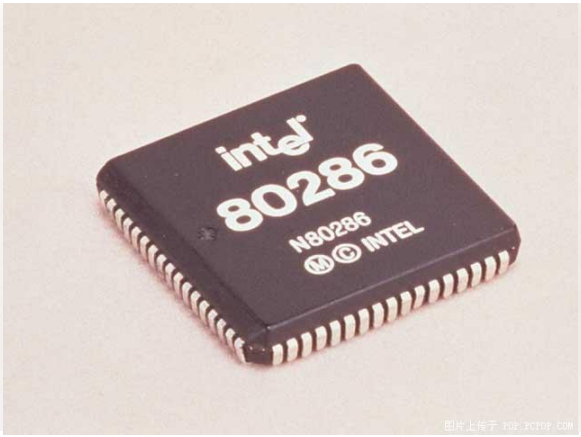
型号	发布年份	字长	主频(MHz)	地址总线宽度	寻址空间	高速缓存
8086	1978	16	4.77	20	1M	无
8088	1979	16	4.77	20	1M	无
80286	1982	16	6~20	24	16M	无
80386	1986	32	12.5~33	32	4G	有
80486	1989	32	25~100	32	4G	8KB
Pentium (586)	1993	32	60~166	32	4G	8KB数 8KB指令
PRO(P6)	1995	32	150~200	36	64G	8KB数据 8KB指令 256KB二级Cache
PII	1997	32	233~333	36	64G	32KB 512JB二级Cache, 有独立封装和独立总线



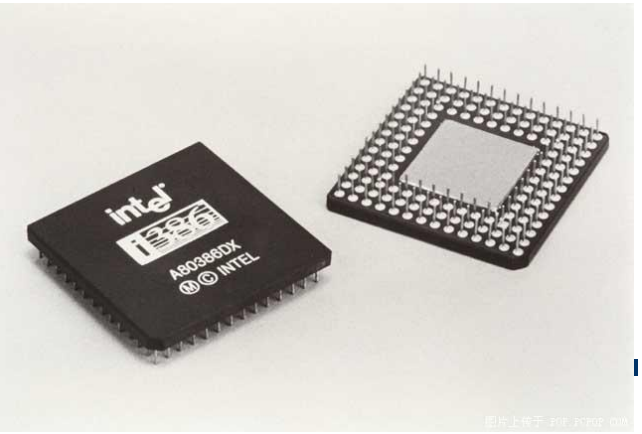
CPU和内存之间有多少根地址信号线，2^{20}=1M



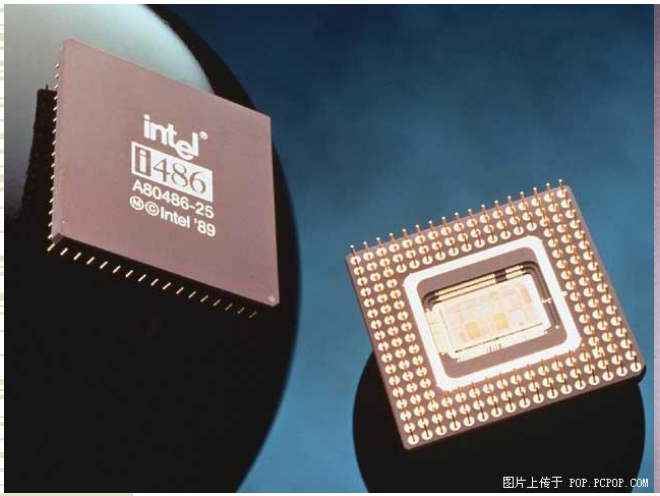
8086 (1978)



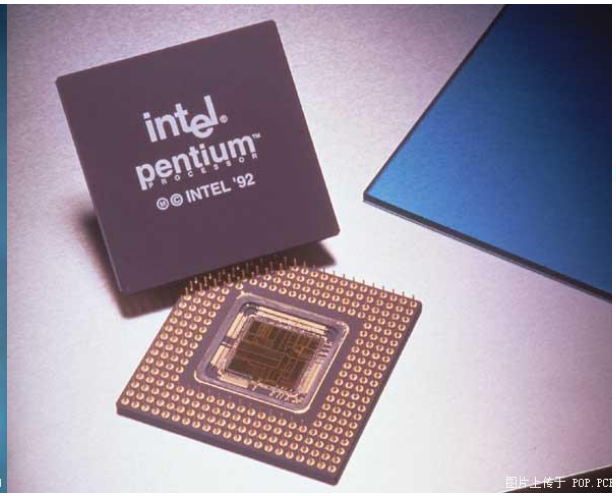
80286 (1982)



80386 (1985)



80486 (1989)



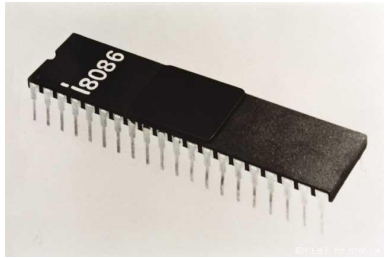
Pentium (1993)

Pentium IV (2000)

- 80286开始：增加了保护模式，可提供虚拟存储管理和多任务管理
  - 虚拟存储管理：提供更大的存储空间，允许用户运行程序空间大于实际主存储器空间
  - 多任务管理：允许多个用户或多个任务同时使用计算机
- 80386开始：又增加了虚拟8086工作模式
  - 虚拟86工作模式：一台机器可同时模拟多个8086处理器的工作，多用户可以完全安全隔离等
  - 硬件支持多任务转换，提高效率

CPU+浮点运算协处理器
如8086、8087独立芯片

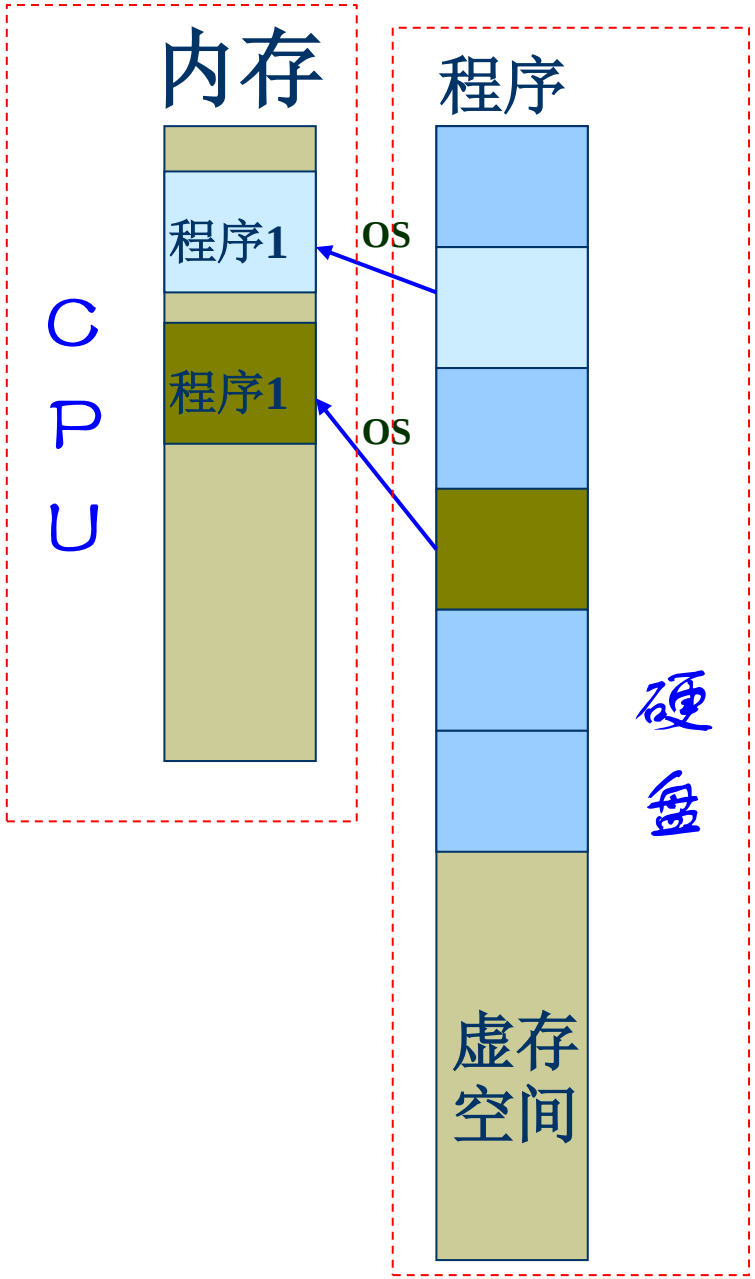
- 80486开始：把协处理器集成到CPU芯片中，提高浮点处理速度
- 字长增加有利于提高计算精度



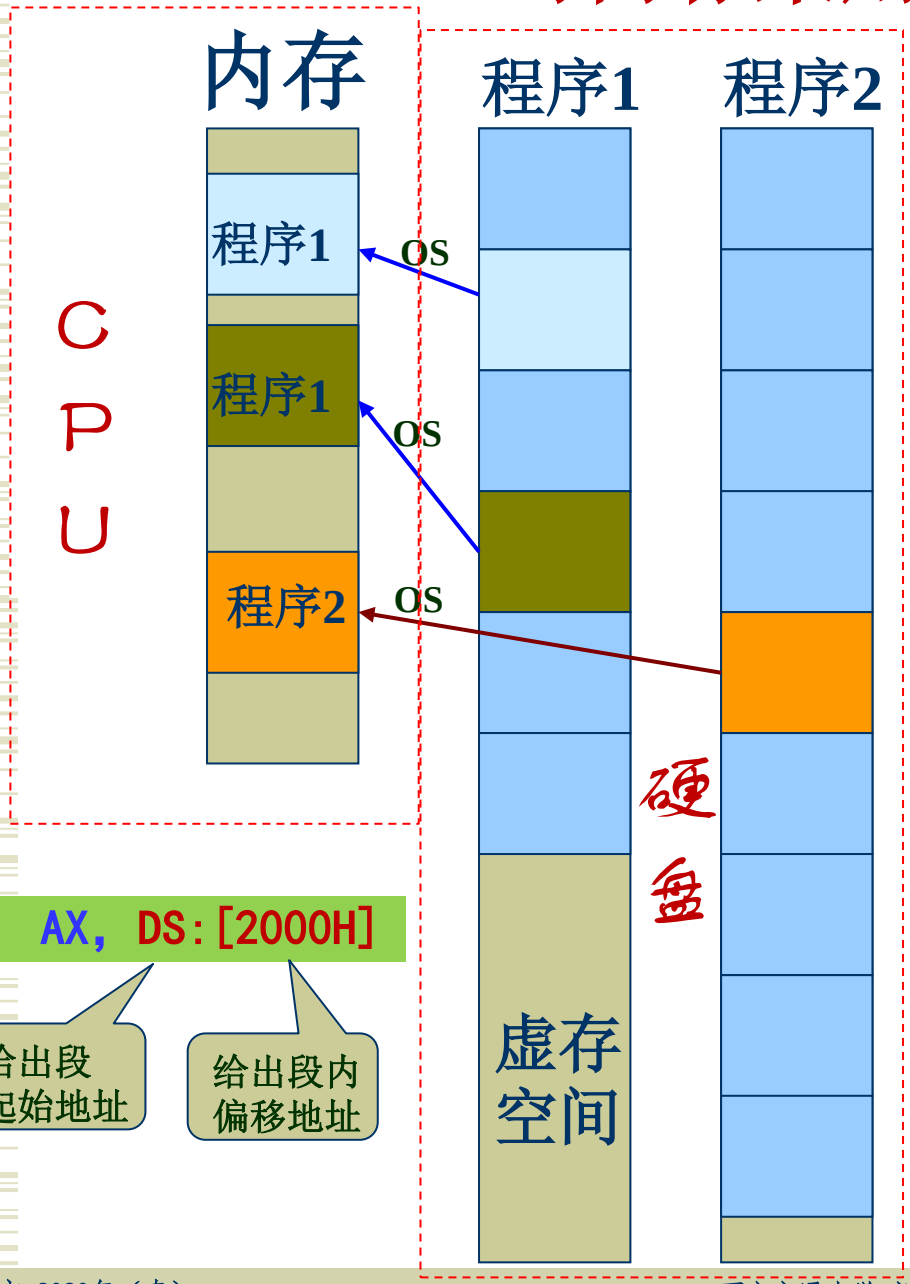
对内存管理由OS完成

- 虚拟存储空间设在硬盘上
- 内存物理空间可比虚拟存储空间小
- 程序执行时
  - 操作系统先分配一个内存段
  - 将程序从虚拟存储空间装入内存执行
  - 内存中程序/数据只是虚拟存储中程序/数据一小段副本
  - 管理过程对应用程序员透明

因此，对内存进行分段管理使用，这样计算机可支持多用户多任务，并且可提高内存的利用率。



对内存采用分段管理优点

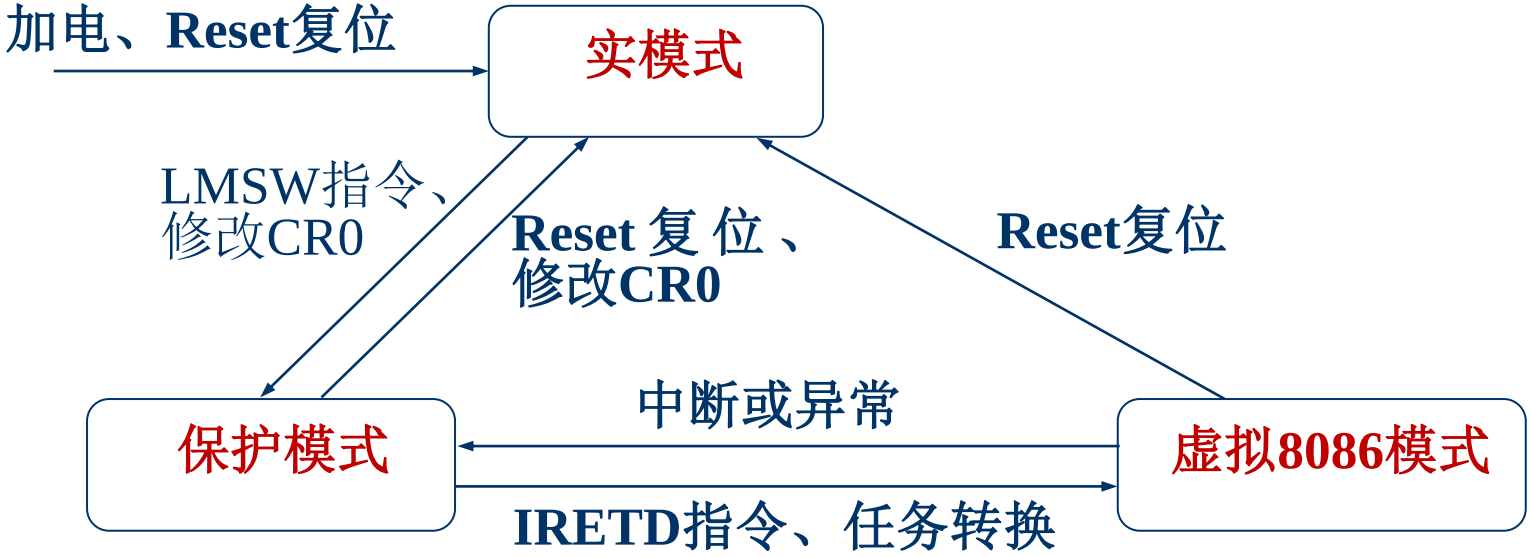


- 内存空间可以比虚存空间小
  - 虚存空间=寻址空间(2^n), 奔腾64G
  - 内存空间可以很小
- 程序可以按虚存空间编写
- 分段管理
  - 正在执行的代码段和数据段装入内存即可
  - 多个程序可以同时使用内存
  - 提高内存利用率
  - 可以运行比内存大的程序
  - 操作系统装入/替换, 对用户透明

①资源利用率，②多任务，③保护隔离

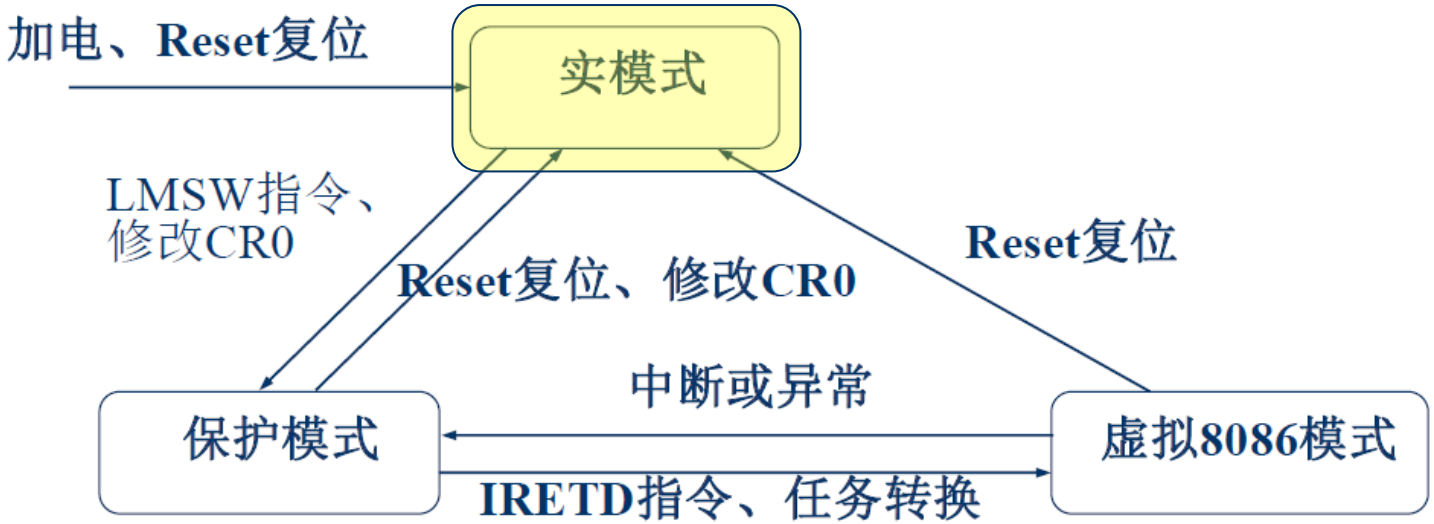
3种运行模式关系

- 从80386开始，Intel的CPU具有3种运行模式：实模式、保护模式和虚拟8086模式。



(1) 实模式

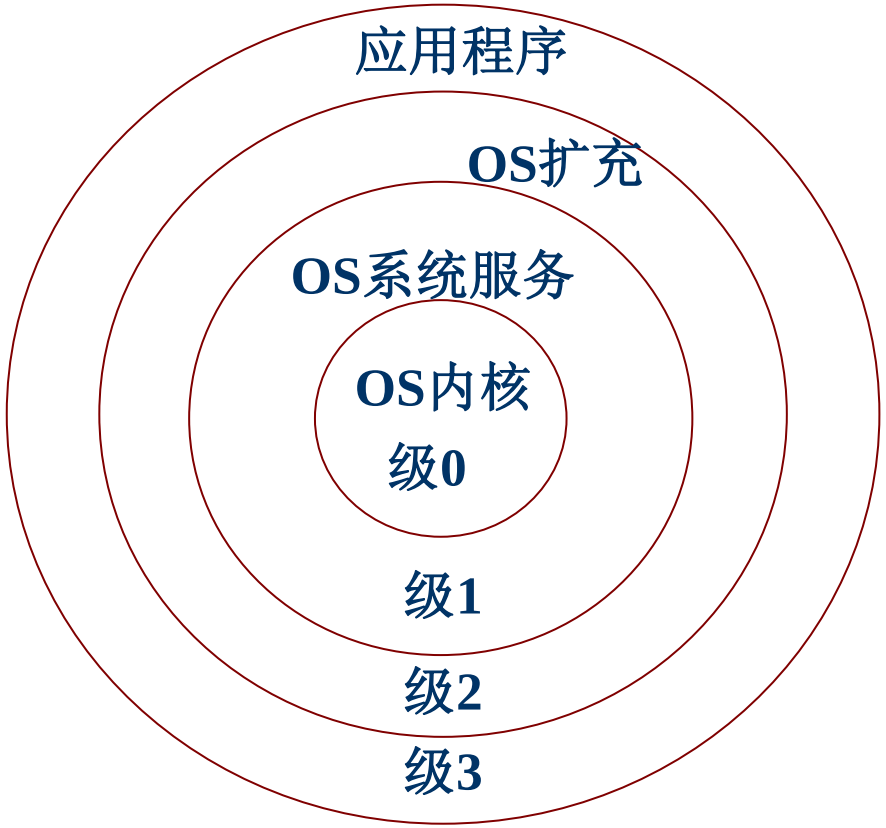
- CPU复位（Reset）或加电（Power On）时，处理器以实模式工作
- 在实模式下，80X86内存寻址方式和8086相同，由16位段寄存器和16位偏移地址形成20位的内存物理地址
- **在实模式下，所有的段都是可以读、写和可执行的**



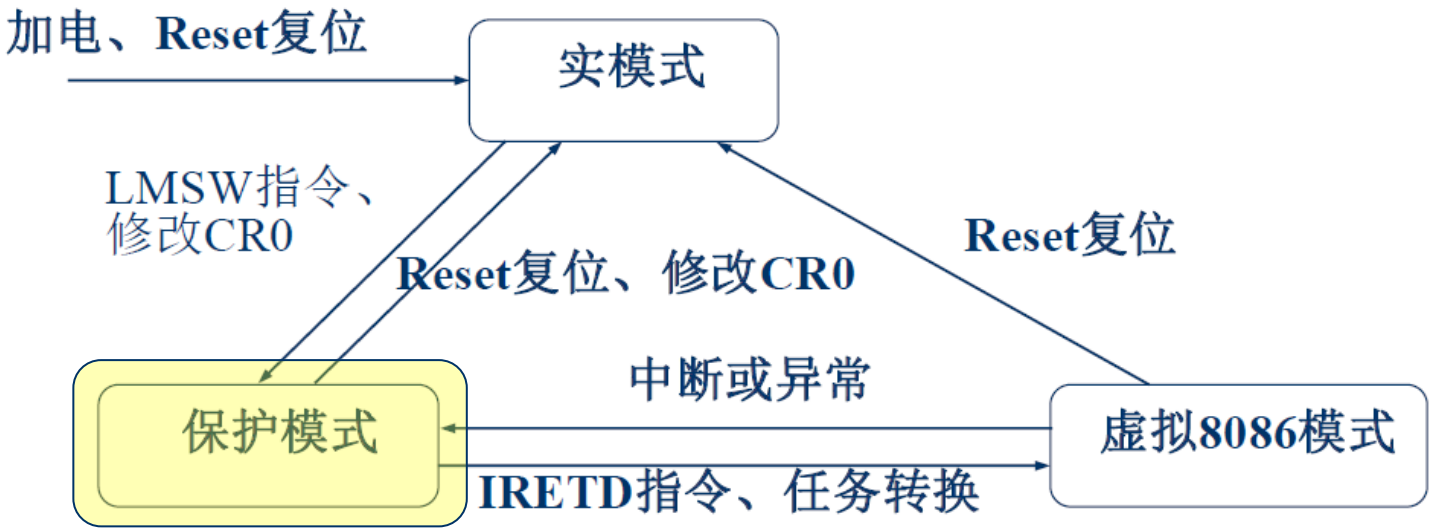
(2) 保护模式

- 在保护模式下，CPU提供了多任务、内存分段分页管理和特权级保护等功能
  - 这些功能是Windows/Linux等现代操作系统的基石
  - 如果没有CPU的支持，操作系统的许多功能根本无法实现
    - 例如，在实模式下，应用程序可以执行任何的CPU指令，读写所有的内存，DOS操作系统就不能控制应用程序的行为，应用程序可以做任何事情，没有任何限制。而在保护模式下，通过设置特权级和内存的分段分页，应用程序只能读写属于它自己的内存空间，而不能破坏其他应用程序和操作系统

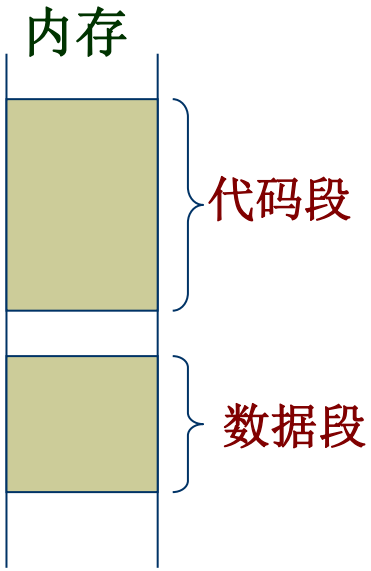
存储保护包括特权级保护和存储区域保护。

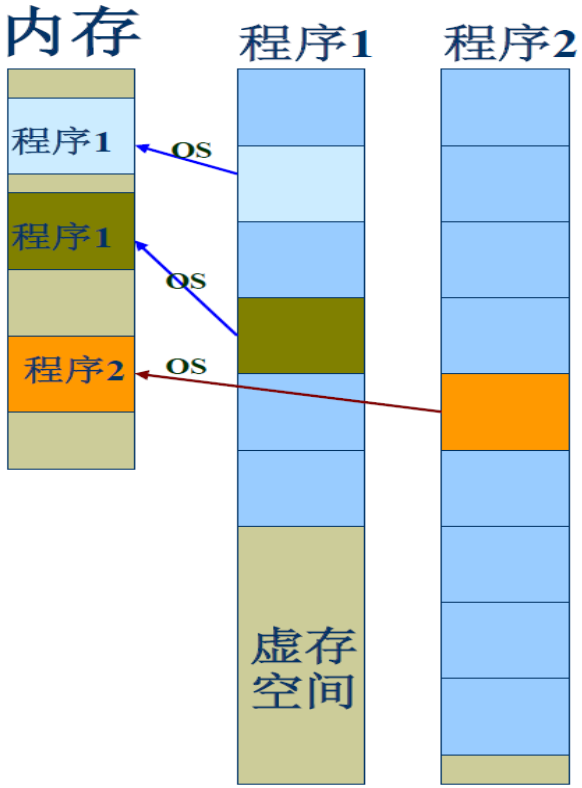


- 实模式下没有特权级的概念，相当于所有的指令都工作在操作系统的特权级0，即最高的特权级。
  - 可以执行所有特权指令，包括读写控制寄存器CR0等。Windows/Linux操作系统是通过在实模式下初始化控制寄存器、GDTR、LDTR、IDTR、TR等寄存器以及页表，然后再通过置CR0的保护模式位（PE位）为1而进入保护模式的。
- 实模式下硬件不支持多任务切换，所有的指令都在同一个环境下执行

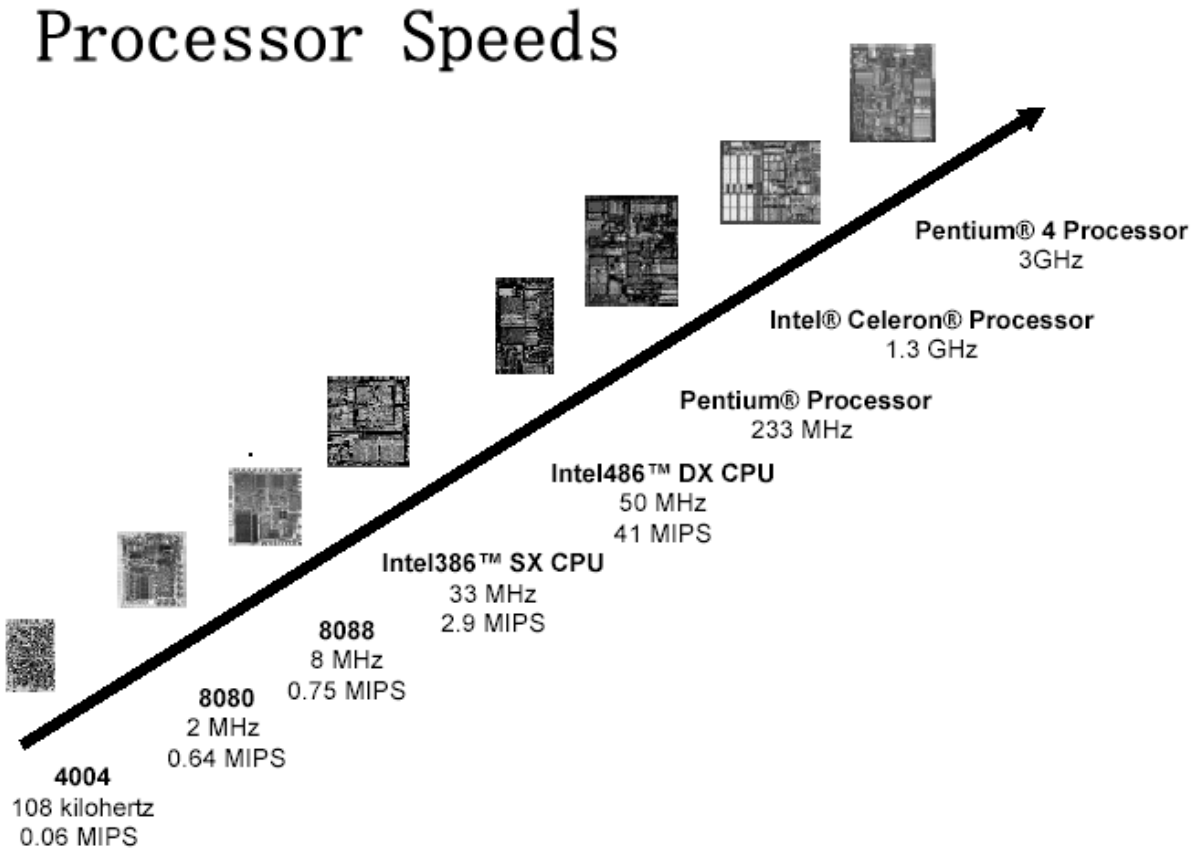


- 保护模式下提供的主要功能有：
  - 段的大小可以设置为2^{32}=4GB，段内的偏移量为32位
  - 特权级保护
  - 支持内存分页机制，支持虚拟内存
  - 支持多任务



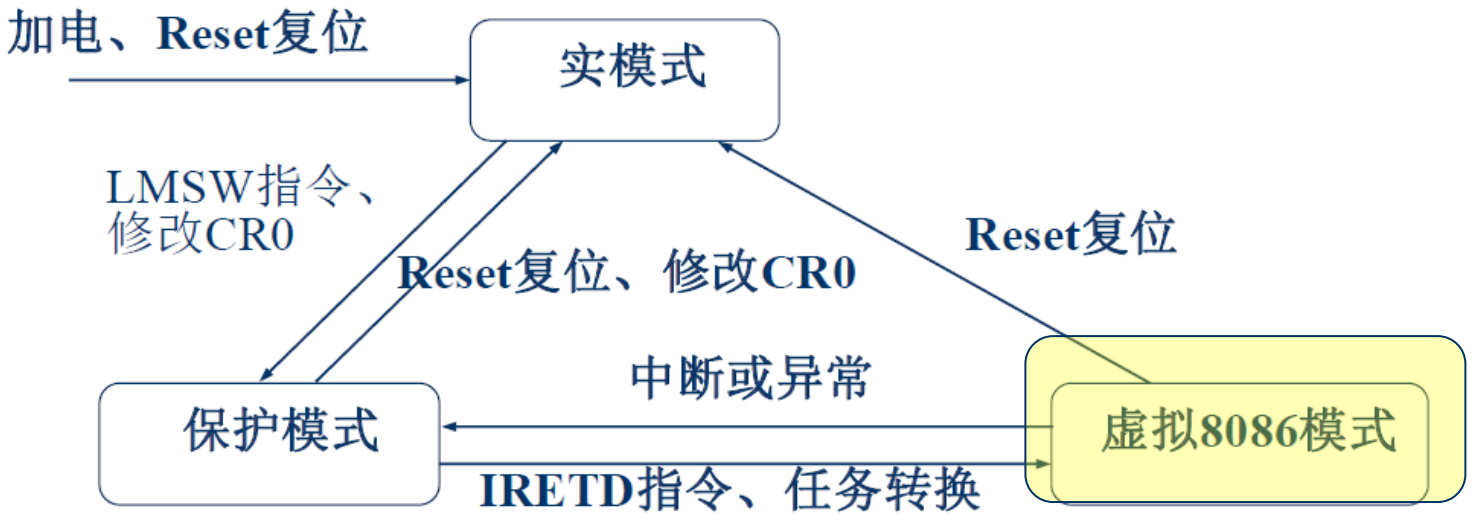


- Intel 推出的80x86系列处理器的性能和功能越来越强。但是，从汇编语言程序设计人员面对这些CPU的体系结构角度来看，8086的实模式和80386的保护模式到目前为止一直适用。
- **本课程介绍的实模式编程以8086为例说明，保护模式编程以80386为例说明**



(3) 虚拟8086模式

- CPU可以同时支持多个真正的CPU任务和多个虚拟86任务
- 以任务形式在保护模式下执行
- CPU支持任务切换和内存分页

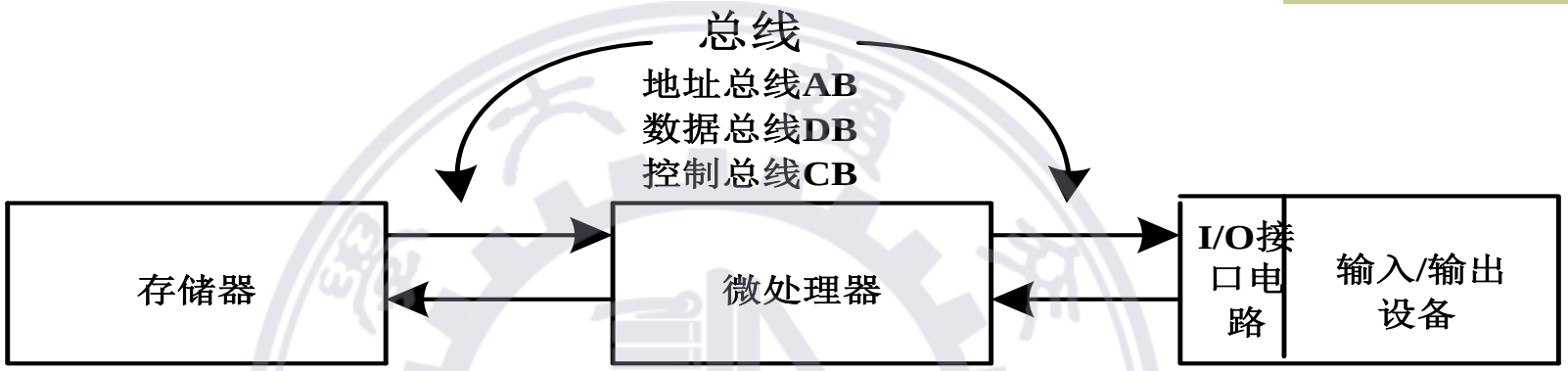


课内测试 01-2-1

1. 请填写正确答案“1”（9分）；

2. 目前80x86处理器有三种工作模式，分别是[填空]、[填空]和[填空]（9分）

2.2 基于微处理器的计算机系统构成



动态RAM
静态RAM
高速缓存
只读存储器ROM
等

8086
8088
80186
80286
80386
80486
Pentium
Pentium Pro
Pentium Pro mmx

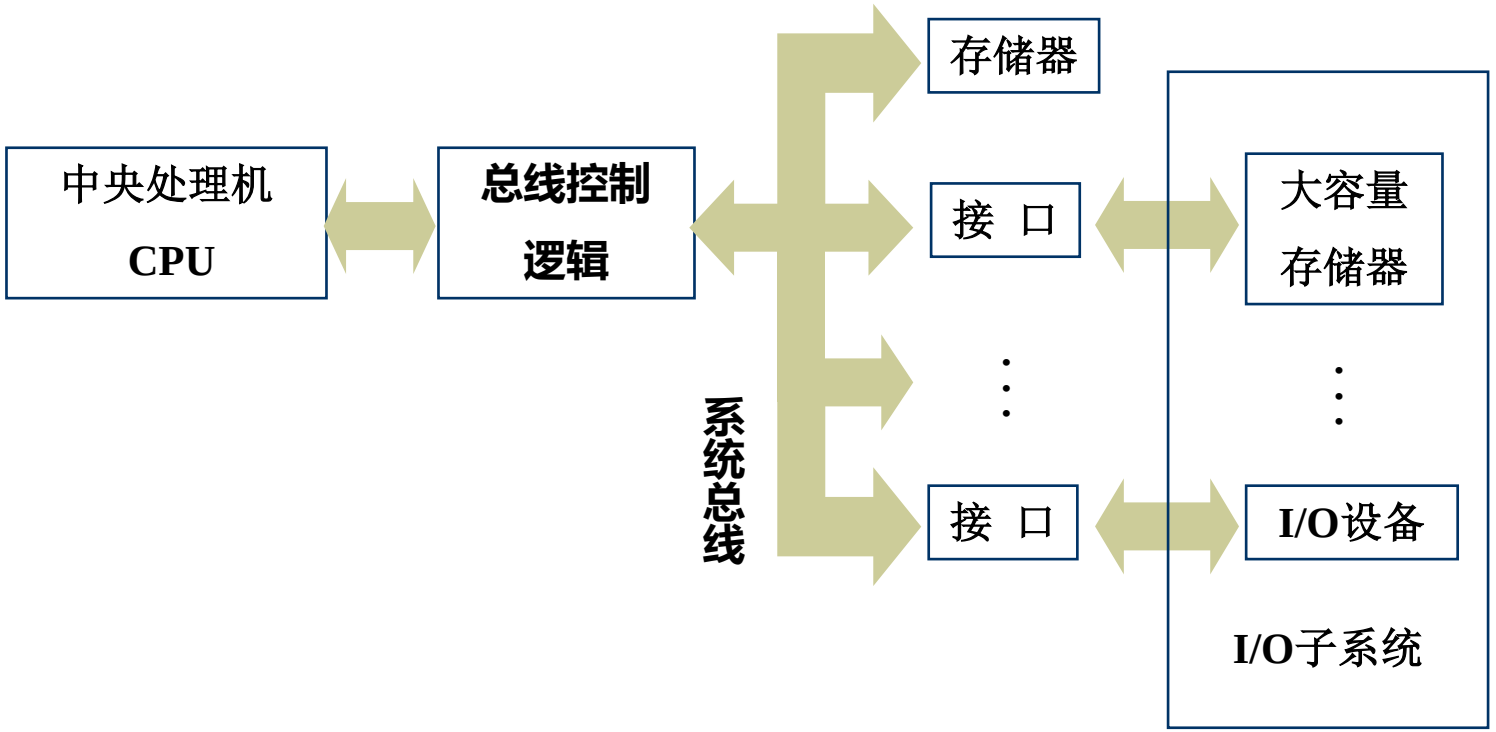
键盘
显示器
打印机
串行通信设备
软盘驱动器
硬盘驱动器
鼠标
CD-ROM驱动器
绘图仪
磁带机

基于微处理器的个人计算机方框图

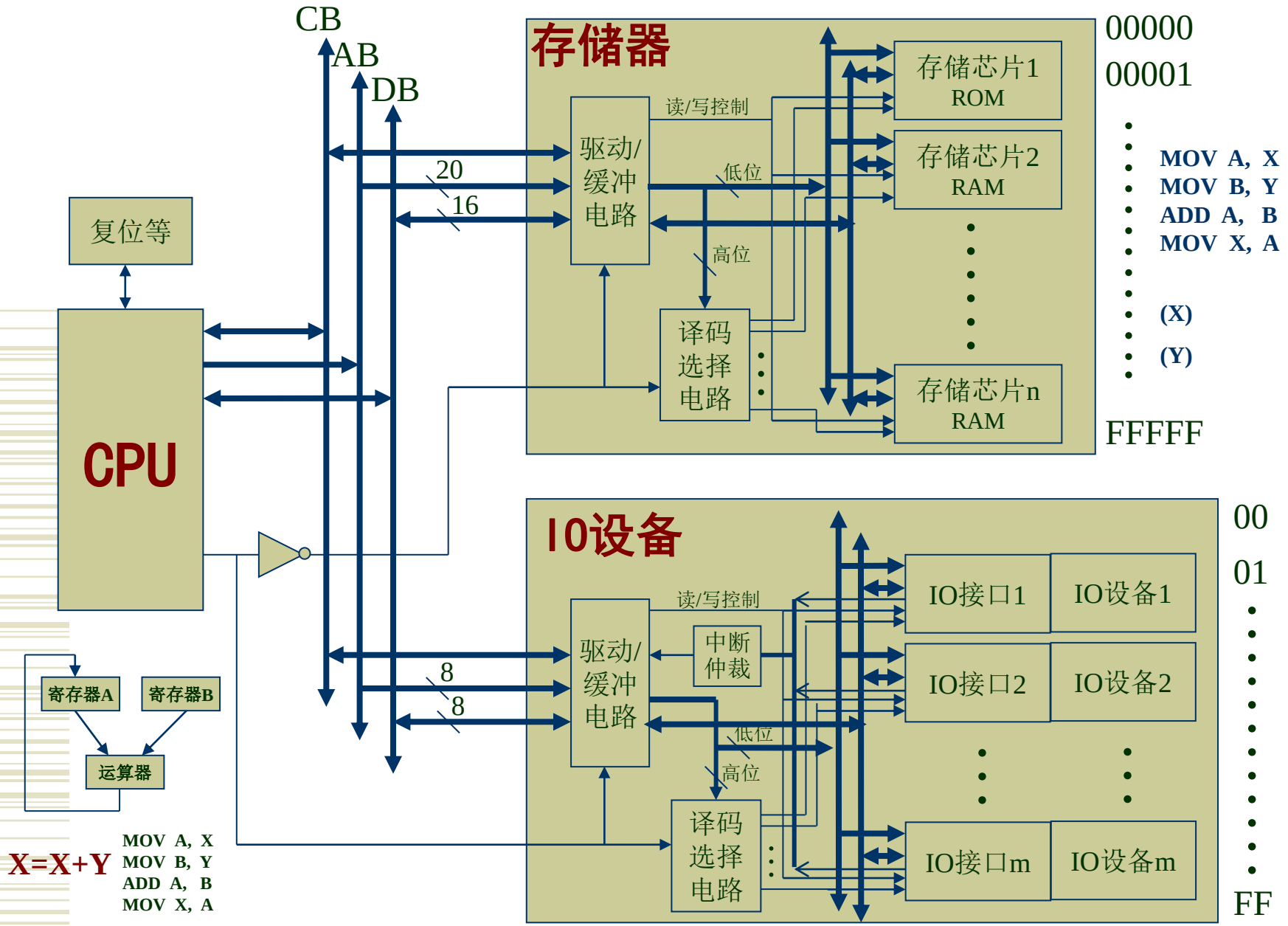
2.2.1 硬件

硬件包括：电路、插件板、机柜等。

典型的计算机结构硬件包括：微处理器、存储器、I/O接口电路及输入输出设备。



邮政编号：710049 → 71：西安市，0049：西安交通大学



2.2.2 软件

软件是为了运行、管理和维护计算机而编制的各种程序的总和。软件可分为系统软件和用户软件两大类

- **系统软件**：由计算机生产厂家提供给用户的一组程序，包括：操作系统、系统程序（编译、汇编、连接等）
  - 核心是操作系统OS
  - C等编译器
  - 汇编语言工具软件
    - MASM.EXE　　TASM.EXE
    - LINK.EXE　　TLINK.EXE
    - DEBUG.EXE
- **用户软件**：用户自行编制的各种程序，包括：用户程序、用户程序库

2.3 中央处理机

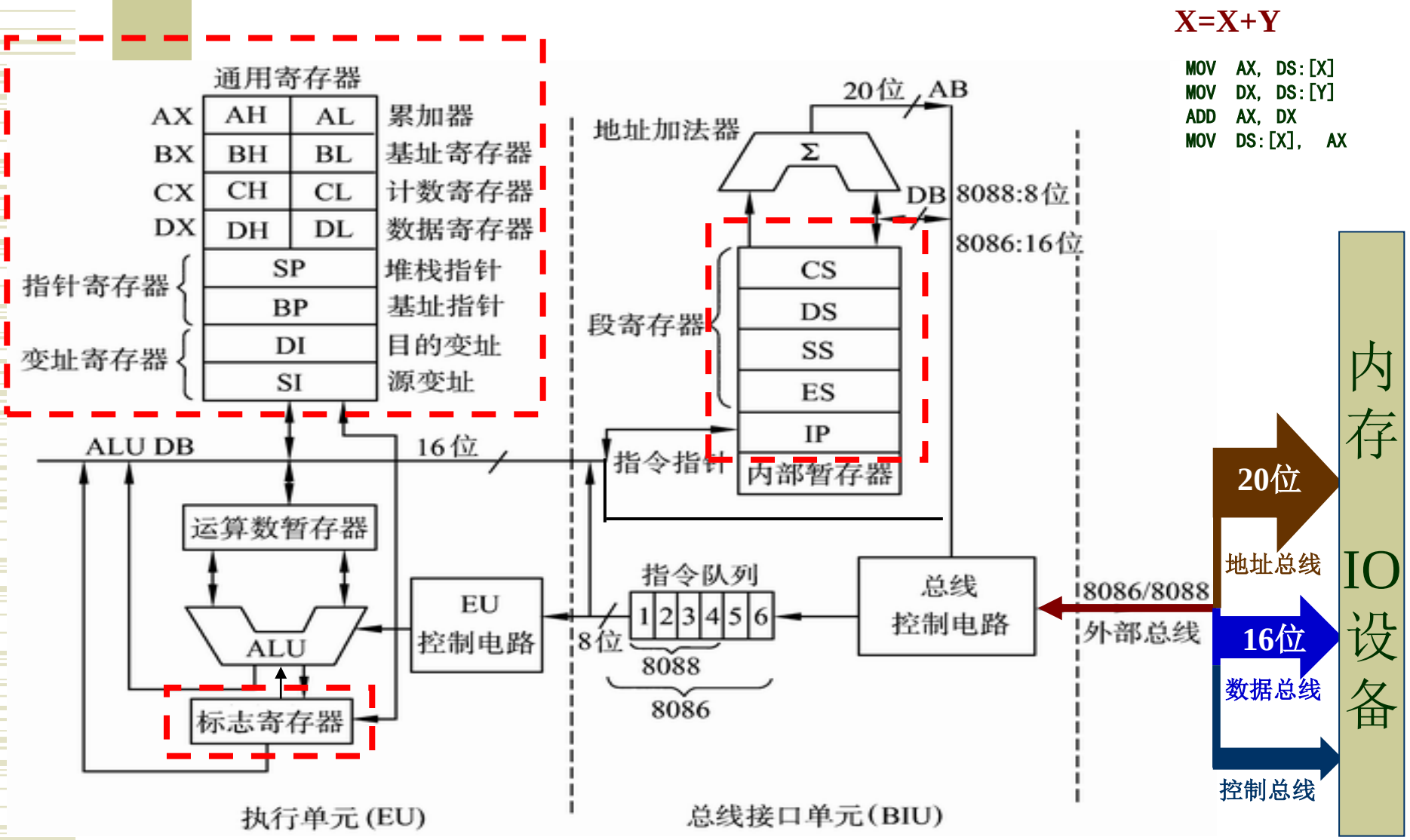
2.3.1 中央处理机（CPU）的组成

CPU的任务是执行存放在存储器里的指令序列，完成用户指定的功能

CPU组成：

1. 算术逻辑部件ALU
2. 控制逻辑EU
3. 工作寄存器（☆必须熟记）

8088/8086CPU的内部结构框图

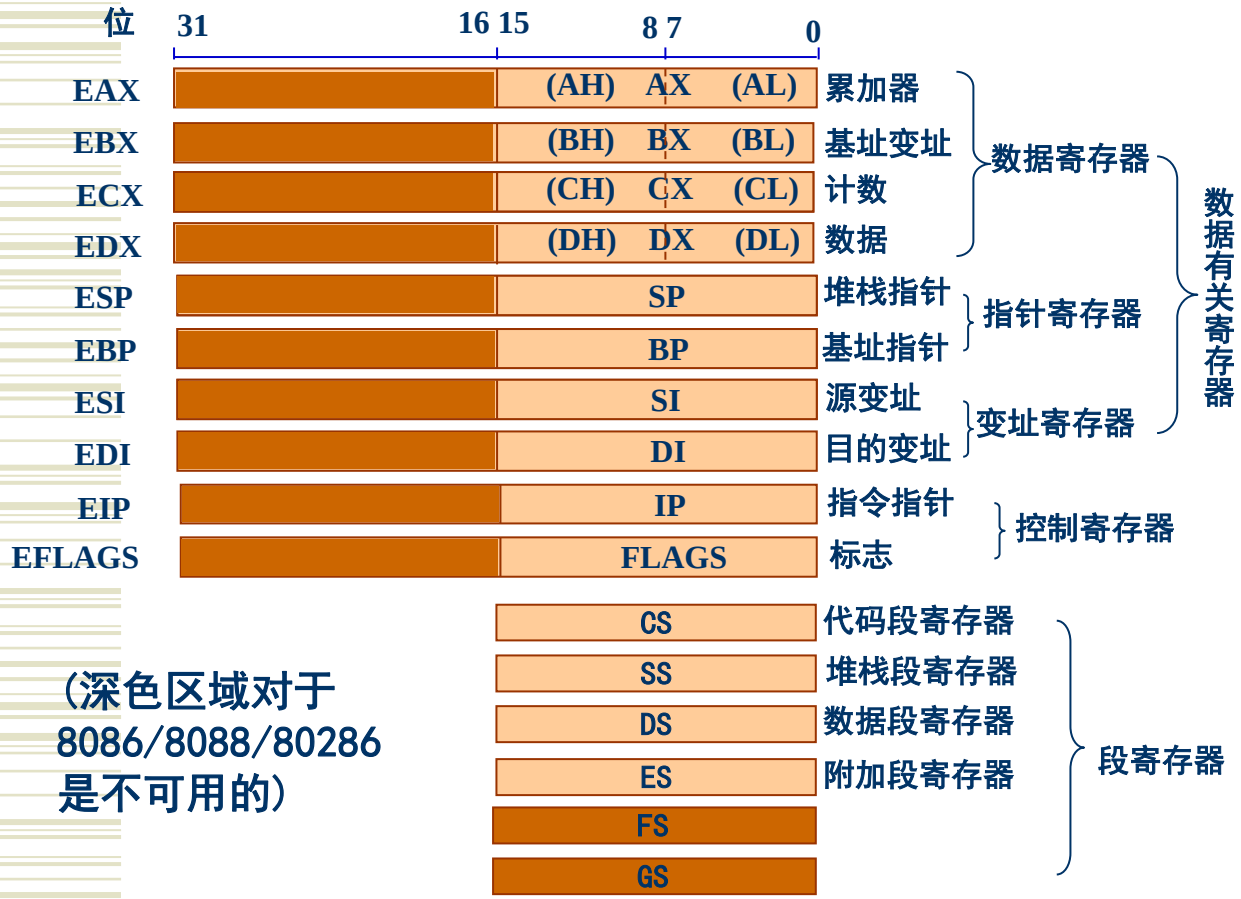


基本概念--寄存器 (Register)

- 处理器中临时数据的存储单元
- 存放运算过程中所需要的或所得到的信息（地址、数据、中间结果）
- 访问速度（ns级）比内存快（μs级）

2.3.2 80X86的寄存器组

80X86 程序可见寄存器组：



(深色区域对于8086/8088/80286是不可用的)

- 程序可见寄存器组包括多个8位、16位和32位寄存器，如图所示。深色部分只对80386（含80386）以上CPU有效。

1. 通用寄存器

AX、BX、CX、DX、SP、BP、SI、DI

80386以上CPU

EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI

2. 专用寄存器

SP、IP、FLAGS

80386以上CPU

ESP、EIP

3. 段寄存器

CS、DS、ES、SS、FS、 GS

以8086/8088的标志为主介绍

- 对于汇编程序员，CPU中的主要部件是寄存器
- 寄存器是CPU中程序员可以用指令读写的部件

8086/8088的寄存器组

- **数据寄存器**（4个16位）

	高8位	低8位
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

- 暂存计算过程中所用到的：操作数、结果或其他信息
- 4个16位寄存器：
  AX、BX、CX、DX
- 8个8位寄存器：
  AH、AL、BH、BL、CH、CL、DH、DL

专用目的

AX：累加器，乘除指令中存放操作数，I/O指令使用其与外设传送信息

BX：基址寄存器

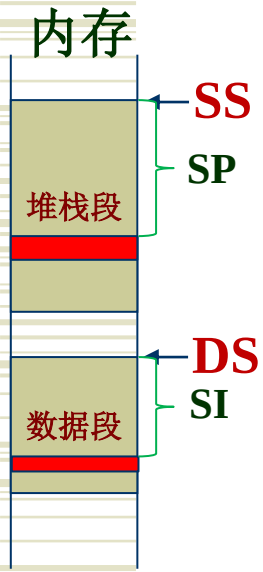
CX：计数器（移位指令、循环指令、串处理指令）

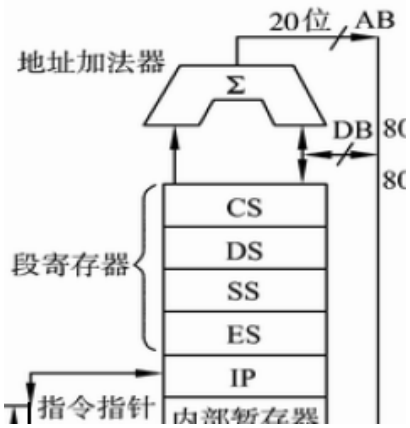
DX：双字长运算（和AX组合）存放高位字，I/O操作存放I/O端口地址

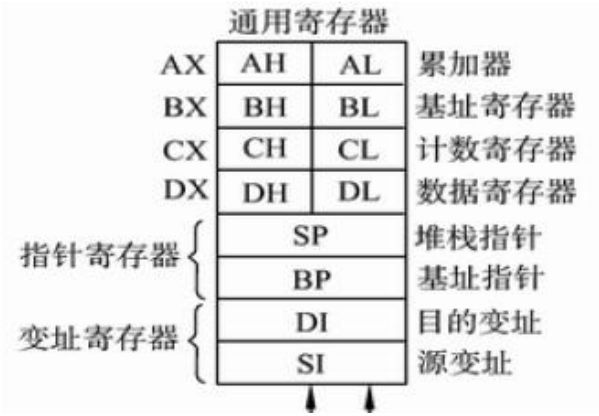
注：386以上增加四个32位寄存器：EAX、EBX、ECX、EDX。

指针及变址寄存器（4个，16位）

- **堆栈指针寄存器：SP**
  - 存放当前堆栈段栈顶偏移量
  - 总是与SS堆栈段寄存器配合存取堆栈中的数据
- **基址指针寄存器：BP**
  - 存放地址的偏移量（或数据）
  - 若存放偏移量时，缺省情况与SS配合
- **变址寄存器：SI**
  - 存放串数据的源地址偏移量（或数据）
  - 若存放偏移量时，缺省情况与DS配合
- **变址寄存器：DI**
  - 存放串数据的目的地址偏移量（或数据）
  - 若存放偏移量时，缺省情况与DS配合

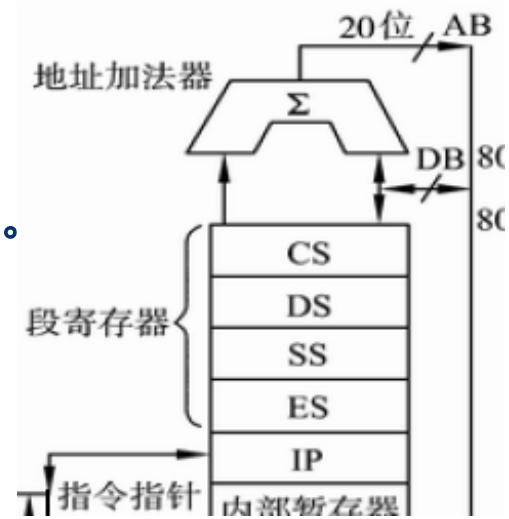




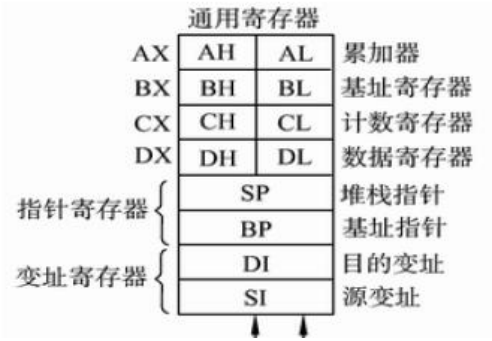


- 注： ESP、EBP、ESI、EDI（32位）只有386以上使用
  实模式使用SP、PB、SI、DI，保护模式使用ESP、EBP、ESI、EDI

段寄存器（4个，16位）



- 存储器采用分段管理方法组织；存储单元物理地址可以用段基址和偏移量计算获得；一个程序可以由多个段组成。
- **功能**：段寄存器存放段基址。在实模式下存放段基地址，在保护模式下存放段选择子
- **代码段寄存器：CS**
  - 存放当前正在运行的程序代码段基地址（开始地址）
- **堆栈段寄存器：SS**
  - 指定堆栈段位置，存放堆栈段的基地址
  - 堆栈段是在内存开辟的一块特殊区域，其中的数据访问原则是后进先出（LIFO），SP指向栈顶，SS指向堆栈段基地址
- **数据段寄存器：DS**
  - 指定当前运行程序所使用的数据段基地址
- **附加数据段寄存器：ES**
  - 指定当前运行程序所使用的附加数据段基地址

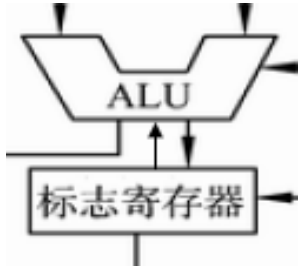


- **注：**
  - 段寄存器FS和GS指定当前运行程序的另外两个存放数据的存储段，只对80386以上；
  - 在默认情况下使用DS所指向段的数据，若要引用其他段中的数据，通常需要显式说明

控制寄存器（2个）

- **指令指针寄存器：IP**
  - 存放代码段中的指令地址偏移量，始终指向下一条即将执行的指令的首地址，控制器根据指令字长自动增加
  - 总是与CS段寄存器配合指出下一条要执行指令的地址
  - 实模式使用IP，保护模式使用EIP （386以上）
- **标志寄存器：FLAGS**

（PSW　程序状态字寄存器）



31…22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0	
										OF	DF	IF	TF	SF	ZF		AF		PF		CF	8086/8088
								NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80286
·····					RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80386
·····				AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80486
·····	ID	VIP	VIF	AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80586

课内测试 02-1-1

1. 请填写正确答案“6”（10分）；

2. 存储器的存储空间分段管理时，实模式情况下，段的起始地址由哪类寄存器给出：[填空]（10分）。

标志寄存器FLAGS

记录正在运行程序的当前状态、控制和访问权限等

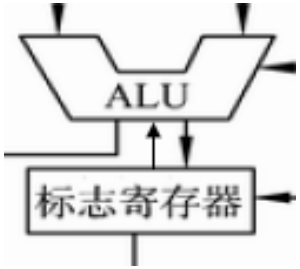
31…22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0	
										OF	DF	IF	TF	SF	ZF		AF		PF		CF	8086/8088
								NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80286
·····					RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80386
·····				AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80486
·····	ID	VIP	VIF	AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80586

PSW (Program Status Word):

条件码标志 – 记录程序运行结果的状态信息，用作后续转移控制条件

控制标志 – 用以控制程序的执行

系统标志 – 用于I/O、可屏蔽中断、程序调试、任务切换和系统工作方式等的控制



8088/8086标志寄存器：FLAGS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

条件码（状态）标志

（记录程序中运行结果的状态信息）

- OF 溢出标志
- SF 符号标志
- ZF 零标志
- CF 进位标志
- AF 辅助进位标志
- PF 奇偶标志

规则：事件成立置1，事件不成立清0

控制标志

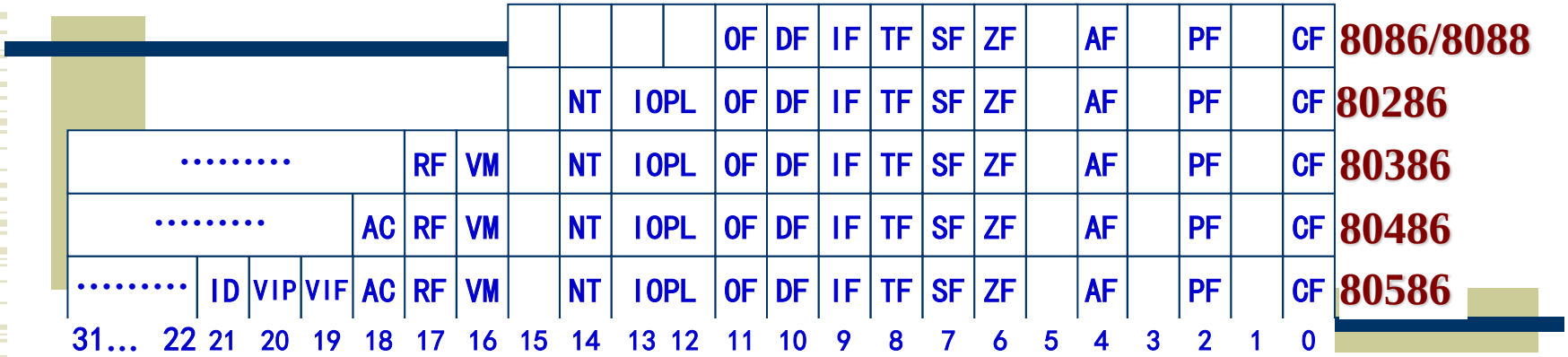
控制标志控制处理器的操作，要通过专门的指令才能使控制标志发生变化

DF 方向标志

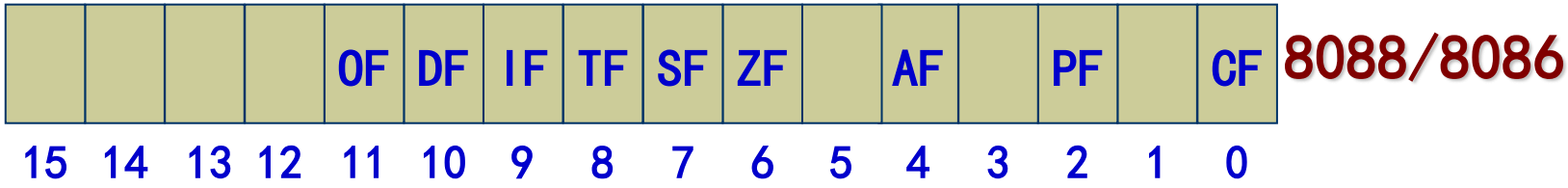
系统标志

用于I/O、可屏蔽中断、程序调试、任务切换和系统工作方式等的控制

- IF 中断标志
- TF 陷阱标志



FLAGS寄存器



- OF(Overflow Flag)溢出标志：由运算结果自动设置

  所谓溢出是指字节（字）运算结果超过了所能表数的范围

  字节运算带符号数范围：-128 ~ +127

  字 运 算带符号数范围：-32768 ~ +32767

  溢出时，标志OF=1，否则OF=0

FLAGS寄存器

8088/8086

- SF(Sign Flag)符号标志：由运算结果自动设置

  SF的值与运算结果的最高位相同

  运算结果为负，SF=1；运算结果为正，SF=0

- ZF(Zero Flag)零标志：由运算结果自动设置

  运算结果为零时，ZF=1；否则，ZF=0。

FLAGS寄存器

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

8088/8086

- CF(Carry Flag)进位标志：由运算结果自动设置

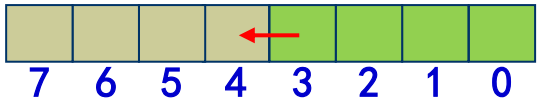
  在运算结果中，若最高有效位产生进位或借位，则CF=1；否则，CF=0

- AF(Auxiliary Carry Flag)辅助进位标志：由运算结果自动设置

  记录运算结果中，低半字节（最低4位）向高半字节（即D3向D4）的进位情况。

  若D3向D4有进位或借位，AF=1；否则，AF=0

  只有在执行十进制运算指令时才关心此位



FLAGS寄存器

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

8088/8086

- PF(Parity Flag) 奇偶标志：由运算结果自动设置

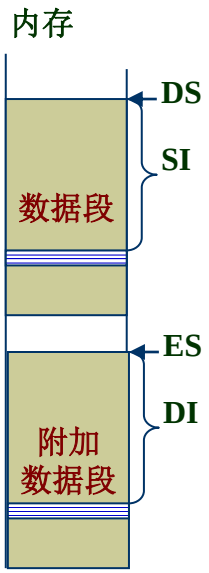
  若运算结果的低8位中，“1”的个数为偶数，则PF=1；否则，PF=0

0	1	1	0	1	0	1	0
7	6	5	4	3	2	1	0

- DF(Direction Flag)方向标志：由指令设置

  用于控制串操作，指示串操作时操作数地址的增减方向

  - DF为1时，串操作后使变址寄存器SI、DI自动减量
  - DF为0时，串操作后，使SI、DI自动增量



FLAGS寄存器

8088/8086

- IF(Interrupt Flag)中断标志：由指令设置

  IF只对外部可屏蔽中断请求（INTR）起作用

  若IF＝1，允许CPU响应INTR

  若IF＝0， 禁止响应INTR

- TF(Trap Flag)陷阱标志：由指令设置

  用于程序调试

  若TF=1，CPU处于单步运行方式

  若TF=0，CPU处于正常工作方式

标志位分类

- **条件（状态）标志**

OF、SF、ZF、AF、CF和PF，其值取决于一个操作完成后，算逻部件ALU所处的状态

- **控制标志和系统标志**

DF、IF和TF，其值是通过指令人为设置的，用以控制程序的执行

例： MOV AX, 1

MOV BX, 2

ADD AX, BX

指令执行后，AX=3, OF=0, CF=0, ZF=0, SF=0

例： MOV AX, FFFFH

MOV BX, 1

ADD AX, BX

指令执行后，AX=0, OF=0, CF=1, ZF=1, SF=0

80286以上机增加的标志（简介）

- 286以上CPU
  - IOPL（I/O Privilege Level）：I/O特权级(保护模式下用,第八章详细内容)
    - 当在保护模式工作时，IOPL指定要求执行I/O指令的特权级。若当前任务的特权级比IOPL高则执行I/O指令；否则发生一个保护异常，导致执行程序被挂起
  - NT（Nested Task）：嵌套任务标志
    - 指示当前任务是否嵌套于另一任务之内
    - 保护模式在执行中断返回指令IRET时要测试NT值
      - 当NT＝1时，表示当前执行的任务嵌套于另一任务之中，执行完该任务后要返回到另一任务，IRET指令的执行是通过任务切换实现的
      - 当NT＝0时，用堆栈中保存的值恢复FLAGS、CS及IP寄存器的内容，以执行常规的IRET中断返回操作

80286以上机增加的标志（简介）

- 386以上CPU
  - *RF (Resume Flag)*：重启动标志。该标志控制是否接受调试故障

    RF＝0　接受

    RF＝1　关闭
  - *VM (Virtual-8086 Mode)*：虚拟方式标志。
    - 当CPU处于保护模式时，若VM＝1则切换到虚拟模式；否则CPU工作在一般的保护模式

80286以上机增加的标志（简介）

- 486以上CPU
  - AC (Alignment Check Mode)：地址对齐检查标志。
    - 对存储单元地址要求：字节从任意地址访问，字从偶地址开始，双字从4的倍数地址开始
    - AC＝1，进行地址对齐检查，当出现地址不对齐时会引起地址对齐异常，只有在特权级3运行的应用程序才检查引起地址对齐故障
    - AC＝0，不进行地址对齐检查

80286以上机增加的标志（简介）

- Pentium
  - ID (Identification Flag)：标识标志
    - 若ID＝1，则表示Pentium支持CPUID指令，CPUID指令提供Pentium微处理器有关版本号及制造商等信息
  - VIF (Virtual Interrupt Flag)：虚拟中断标志
    - 是虚拟方式下中断标志位的映像
  - VIP (Virtual Interrupt Pending Flag)：虚拟中断挂起标志
    - 与VIF配合，用于多任务环境给操作系统提供虚拟中断挂起标志

Debug下演示标志位符号

标志位的符号表示		
标志名	标志= 1	标志=O
OF 溢出（是/否） DF 方向(减量/增量) IF 中断（允许/关闭） SF 符号（负/正） ZF 零（是/否） AF 辅助进位（是/否） PF 奇偶（偶/奇） CF 进位（是/否）	OV DN EI NG ZR AC PE CY	NV UP DI PL NZ NA PO NC

程序调试时，将标志位1或0用符号表示，便于阅读、理解

```
C:\>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0100   NV UP EI PL NZ NA PO NC
0B39:0100 40              INC     AX
-r ax
AX 0000
:1111
-r
AX=1111  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0100   NV UP EI PL NZ NA PO NC
0B39:0100 40              INC     AX
```

2.4 存储器

➢ 存储器是用来存放程序、数据、中间结果和最终结果的记忆装置

2.4.1 存储单元的地址和内容（☆）

- 计算机存储信息的基本单位是一个二进制位
- 8086字长为16位，地址长度20位
- 80386以上机的字长为32位，地址长度32位以上

➢16位二进制数能表示的地址？
➢20位地址如何表示？
➢存储单元中的地址和内容？

存储器地址和内容示意图

地址	内容
00000H	
00001H	
00002H	
00003H	
…	…
03080H	78H
03081H	56H
03082H	34H
03083H	12H
…	…
FFFFFH	

- 存储器以字节(8bit)为单位存储信息(数据等)
- 每个字节单元有一个地址

  从0编号，顺序加1
- 地址也用二进制数表示

  无符号整数，写成十六进制
- 16位二进制数可表示2^{16}=64K个地址

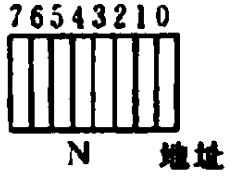
  0000H ～ FFFFH,
- 字长16位，一个字要占用相继的两个字节
  - 低位字节存入低地址，高位字节存入高地址
  - 以偶地址访问（读/写）存储器
  - 字单元地址用它的低地址来表示

    (03080H)=5678H

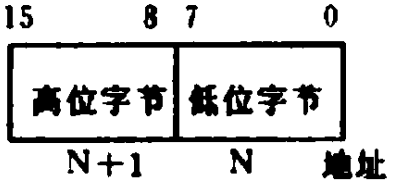
    ((0004H))=2F1EH （书P25）
- 双字长32位时？

  (03080H)=12345678H

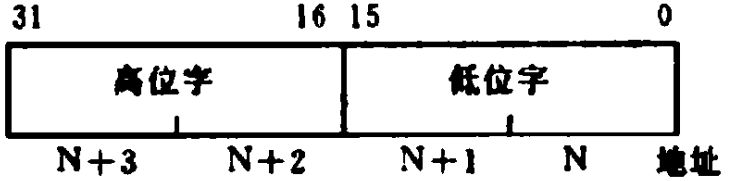
存储器仅是按地址以字节为单位存储二进制编码的器件。



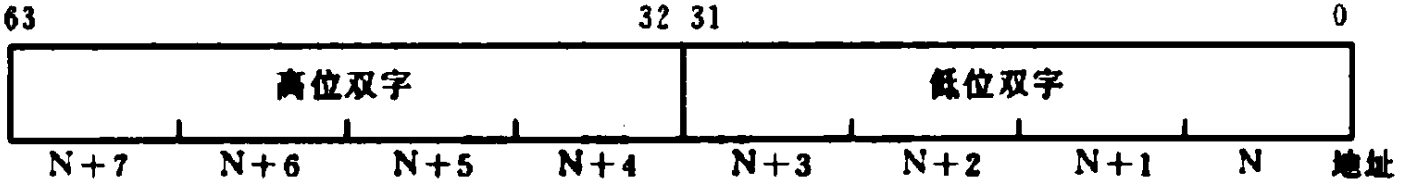
(1) 字节



(2) 字

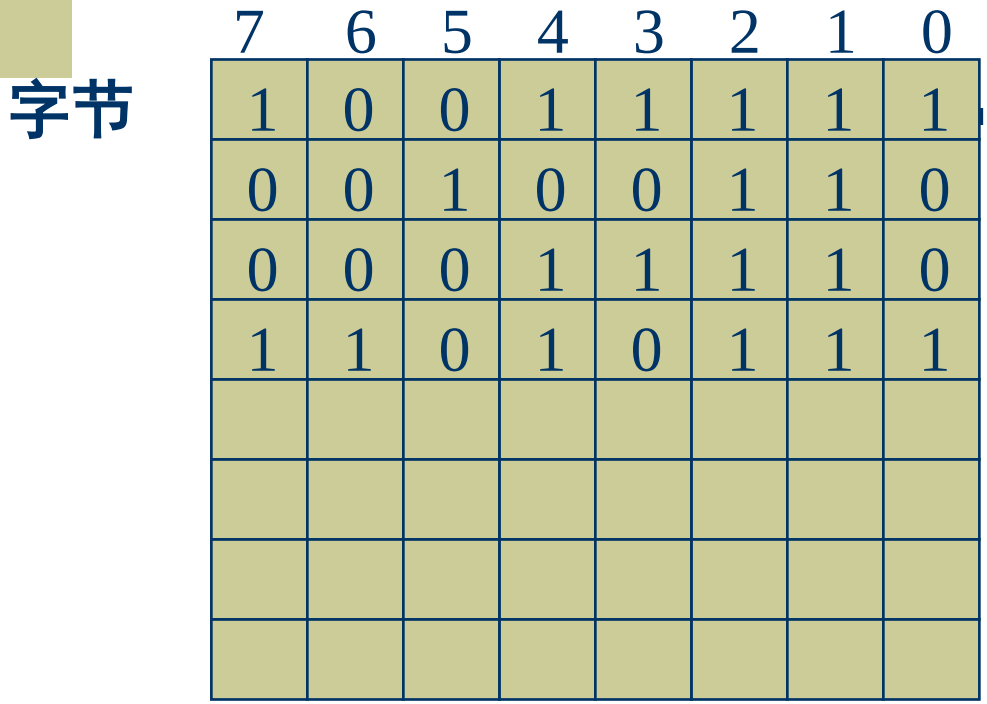


(3) 双字



(4) 4字

例：存储器单元地址和内容



00000H (00000H)=9FH

00001H (00001H)=26H

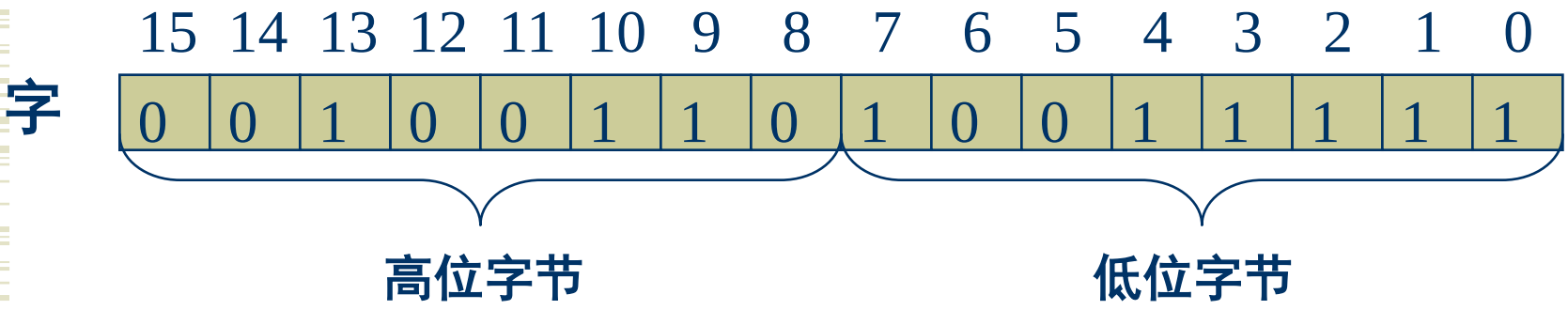
00002H (00002H)=1EH

00003H (00003H)=D7H

(00000H)=269FH

(00002H)=D71EH

读写1个字，需要访问两次存储器



存储系统中的存储部件层次与关系

每级存储器的性能参数可以表示为T_i，S_i，C_i。存储系统的性能可表示为：$T_i<T_{i+1}$；$S_i<S_{i+1}$；$C_i>C_{i+1}$。

X86机存储器管理方式

- X86机的存储器逻辑上采用分段管理的方法
  - X86CPU的相关地址寄存器16位，可管理访问2^{16}=64K空间
  - X86机的存储器物理地址20位，可使用空间2^{20}=1024K=1M

“...程序员在编制程序时要把存储器划分成段...”

- 存储器采用分段管理后，一个内存单元地址要用段基地址和偏移量两个逻辑地址来描述，表示为:

段基址:偏移量

段基址和偏移量的限定、物理地址的形成视CPU工作模式决定

2.4.2 实模式存储器寻址（☆）

- **存储器地址的分段**
  - 存储器有20根地址线　2^{20}=1024K=1M=1048576

    地址范围　00000H ～ FFFFFH
  - 每段最大2^{16}=64K空间
  - 小段：每16个字节为一小段，共有64K个小段

小段的首地址

⇩

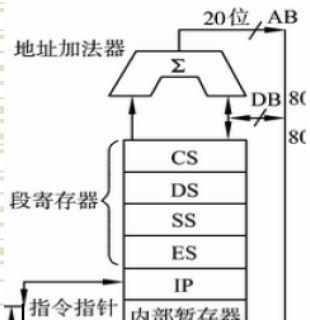
00000H ～ 0000FH
00010H ～ 0001FH
00020H ～ 0002FH
…
FFFF0H ～ FFFFFH

- 存储器分段：段起始地址必须是某一小段的首地址，
  段的大小可以是64K范围内的任意字节

物理地址：每个存储单元的唯一的20位地址

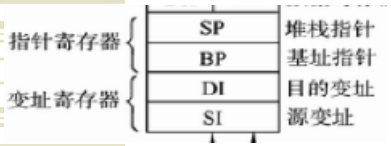
段基地址：段起始地址(20位)=10H × 段寄存器(16位)

偏移地址：段内相对于段起始地址的偏移量（16位），
偏移量又称为**有效地址（EA）**



物理地址 = 16d × 段寄存器 + 偏移地址

= 10H × 段寄存器 + 偏移地址



16 位 段 地 址 0000

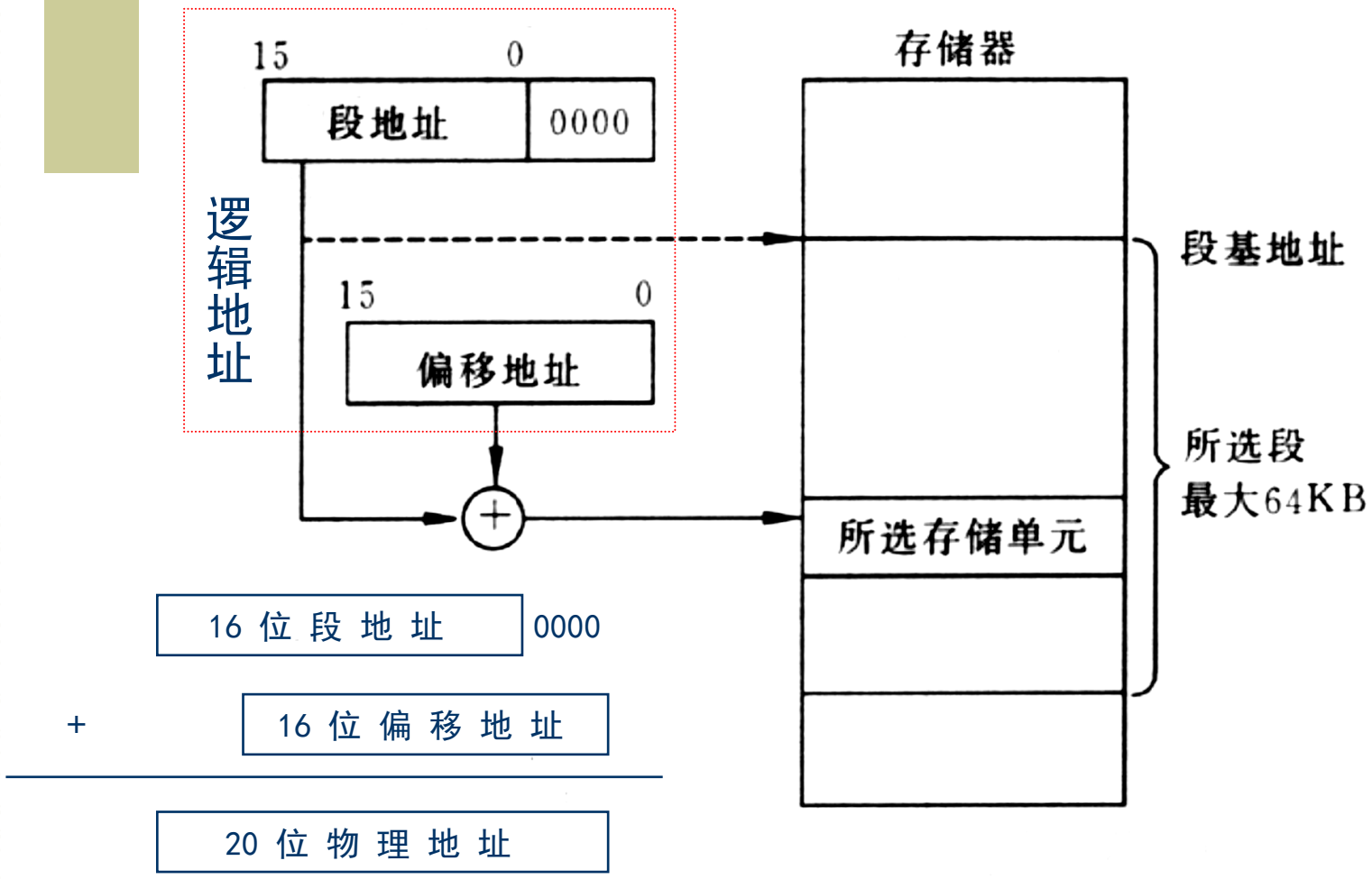
\+

16 位 偏 移 地 址

20 位 物 理 地 址

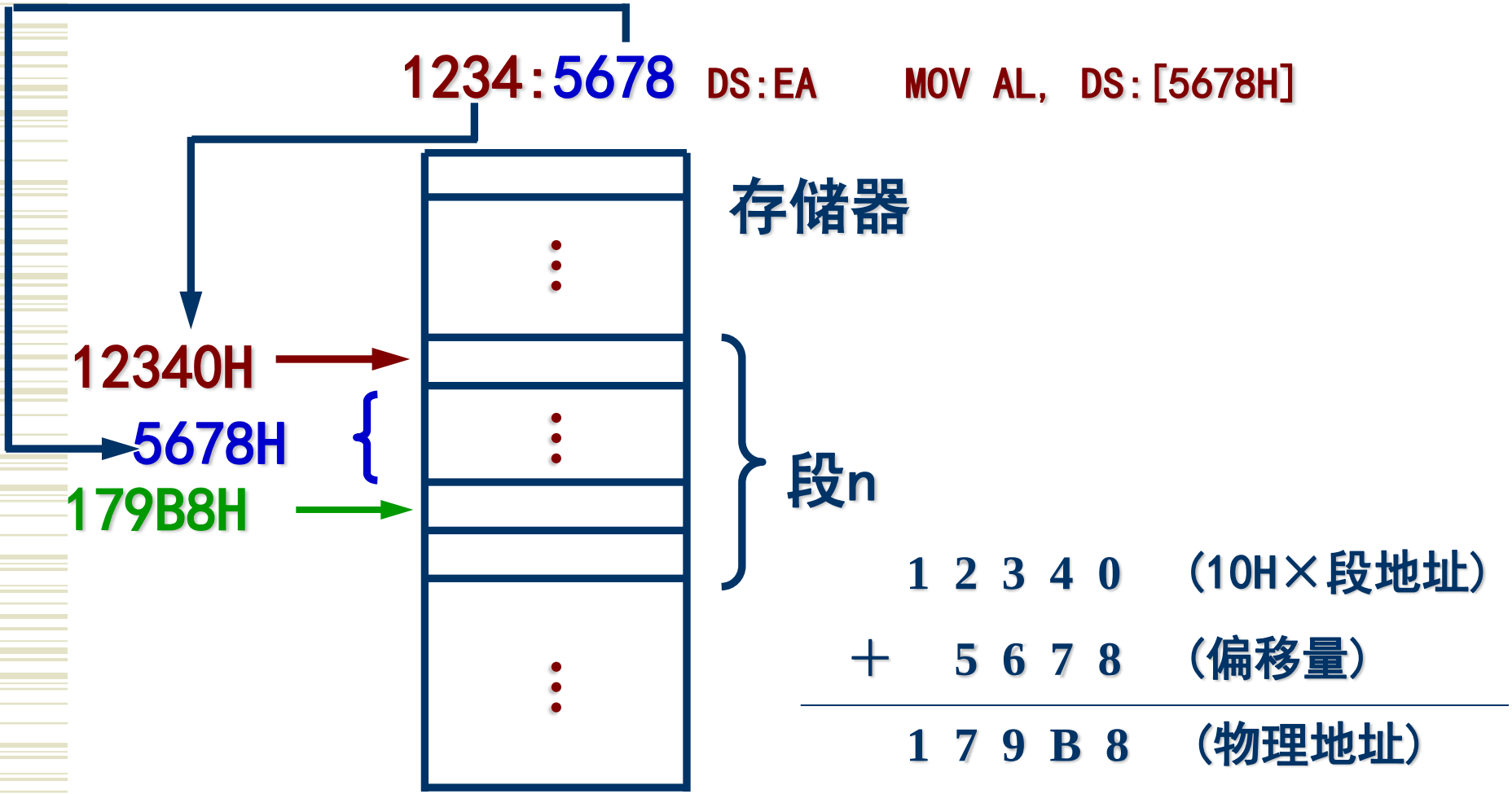
再看看小段、段寄存器、与地址有关寄存器长度想想有什么关系？

为什么段起始地址必须是某一小段的首地址？



存储单元中的物理地址是唯一的
指令中给出的逻辑地址各种各样

例：某内存单元的地址用十六进制数表示为1234:5678，则其物理地址为12340H＋5678H＝179B8H。



课内测试 02-1-2

1. 请在填空中［填空1］填写“3”（10分）；

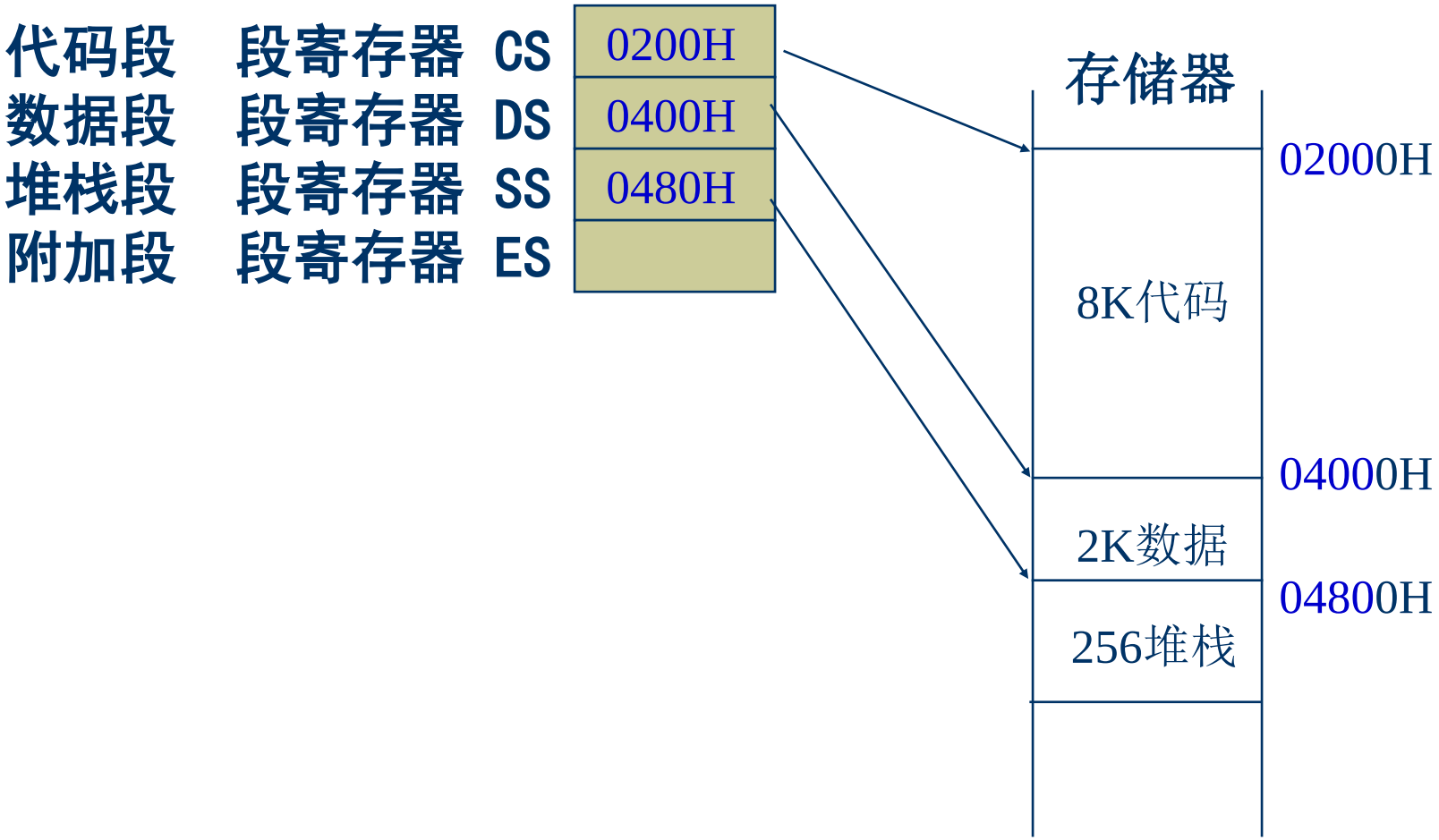
2. 假设：指令中给出某内存单元的逻辑地是DS:[SI]

DS=2345H，SI=0001H

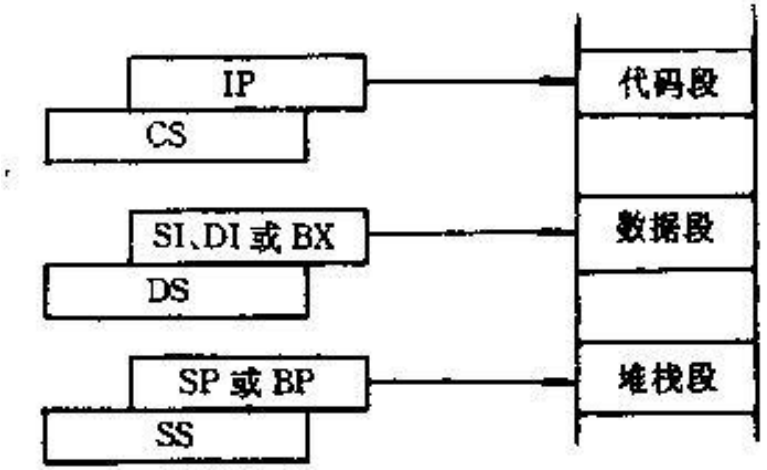
则：实模式下，访问的内存单元物理地址是：

［填空2］H （10分）

- X86处理器中有4个专门存放段地址的段寄存器（16位）



一定要注意段寄存器与偏移地址的缺省组合(P28)及段地址的用途



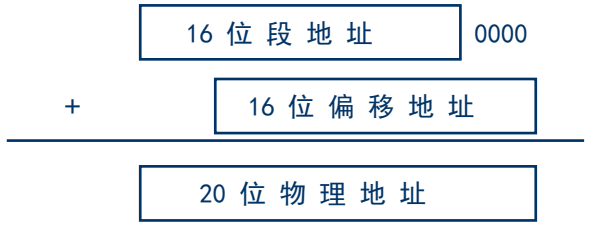
- 汇编指令中给出逻辑地址

  段地址：偏移地址

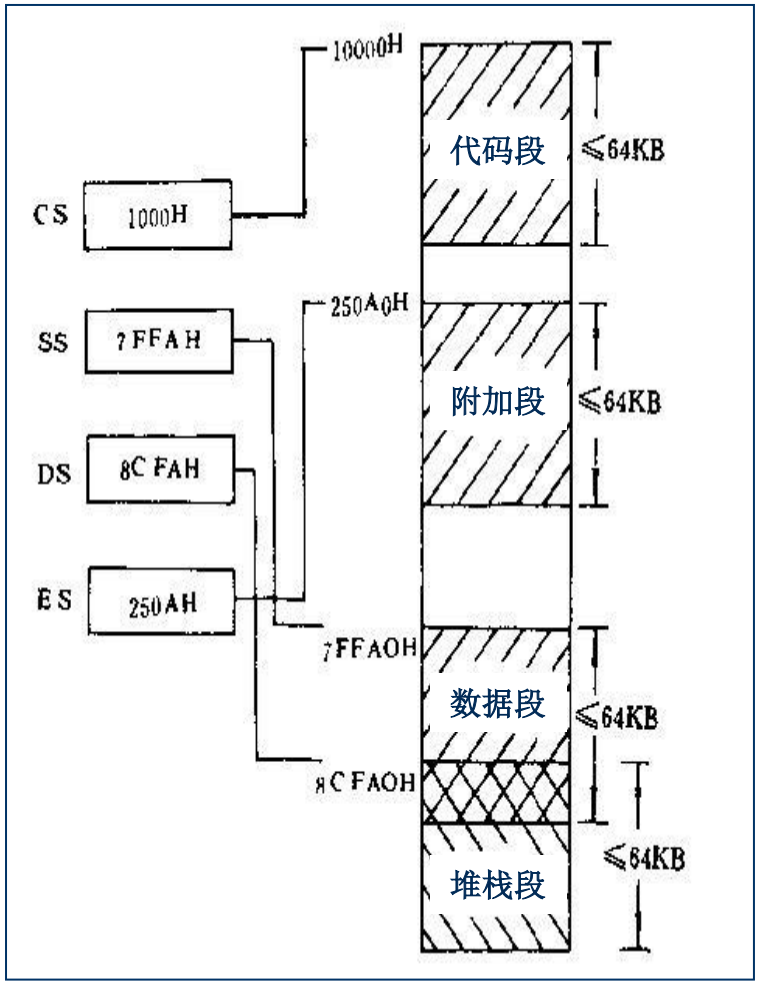
- 机器自动计算物理地址

  实模式与保护模式计算方法不一样

- 逻辑地址：机器指令给出的地址



- 各段之间关系：
  - 相邻
  - 相离
  - 重叠



地址	内容
00000H	
00001H	
00002H	
00003H	
…	…
03080H	78H
03081H	56H
03082H	34H
03083H	12H
…	…
…	以字节为单位编址
FFFFFH	

2.4.3 保护模式

- 80386以后的微机（80386、80486和Pentium）在性能上比8086、80286有了质的飞跃。它们不仅支持实模式，而且支持保护模式。
- 在实模式下，80386只相当于一台高速8086，编程方法与8086相似
- 只有在保护模式下，才能发挥80386的真正作用。在保护模式下，80386可寻址物理地址空间高达2^{32}=4G，支持2^{48}=64T的虚拟存储器，支持多任务和保护机制。

引出保护模式的存储器寻址，主要原因如下：

1. 解决如何寻址的问题（4GB或更多的地址空间）
2. 多用户共享cpu和mm，使微机系统能支持多任务

微机广泛使用要求系统能提供多任务处理功能，即多个应用程序能在同一台计算机上同时运行，而且它们之间必须相互隔离，使一个应用程序中的缺陷和故障不会破坏系统，也不会影响其他应用程序的运行。防止一个进程以非法方式侵权访问存储区域

3. 在系统支持多任务功能的同时，系统也支持了虚拟存储器特性。虚拟存储器可支持程序员编写的程序具有比主存储器所能提供的更大的空间

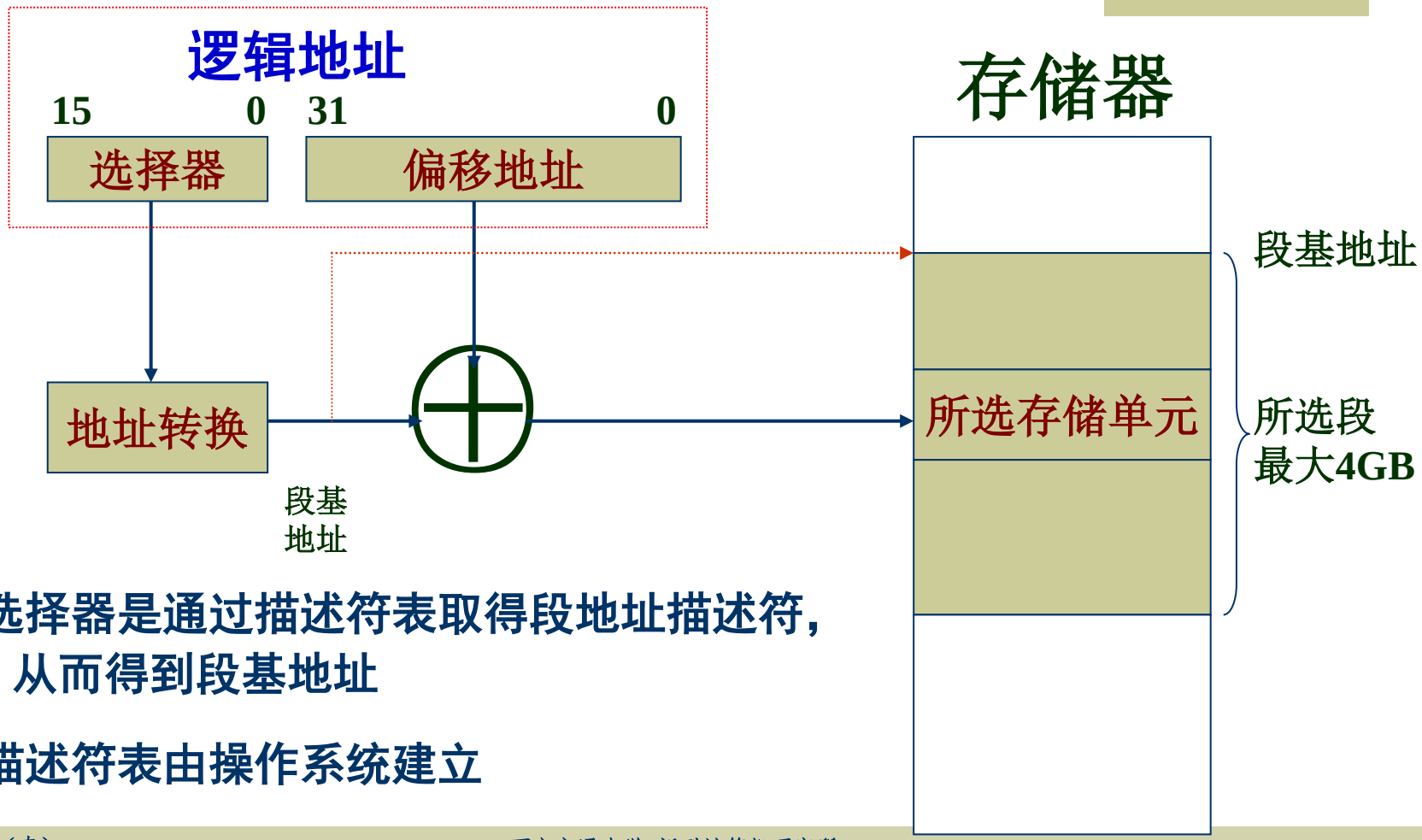
保护模式下存储器寻址

- 讨论保护模式下如何实现存储器寻址，主要内容如下：
  1. 逻辑地址
  2. 描述符
  3. 选择器和描述符表
  4. 程序不可见寄存器

1. 逻辑地址

- 在保护模式存储器寻址中，仍然要求程序员在程序中指定逻辑地址，只是机器采用另一种比较复杂的或者说间接的方法来求得相应的物理地址
- 在保护模式下，逻辑地址由选择器和偏移地址两部分组成
  - 选择器存放在段寄存器中，但它不能直接表示段基地址，而由控制器通过一定的方法取得段基地址，再和偏移量相加，从而求得所选存储单元的物理地址
  - 段基地址由操作系统从磁盘装入指令代码和数据段时自动分配设置，存放在存储器内的描述符表中

保护模式存储器寻址过程



- ✓ 选择器是通过描述符表取得段地址描述符，从而得到段基地址
- ✓ 描述符表由操作系统建立

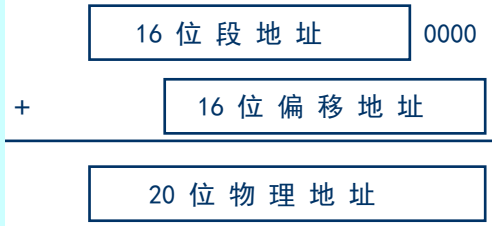
实模式下：

指令给出的逻辑地址=段寄存器：偏移地址

段寄存器左移4位，获得20位段起始地址

加上段内偏移地址

获得20位存储单元物理地址



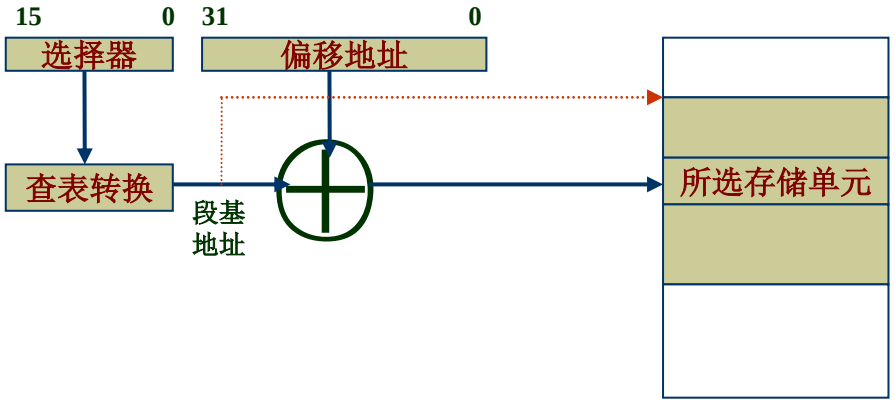
保护模式下：

指令给出的逻辑地址=段寄存器：偏移地址

查段描述符表，获得24、32位段起始地址

加上段内偏移地址

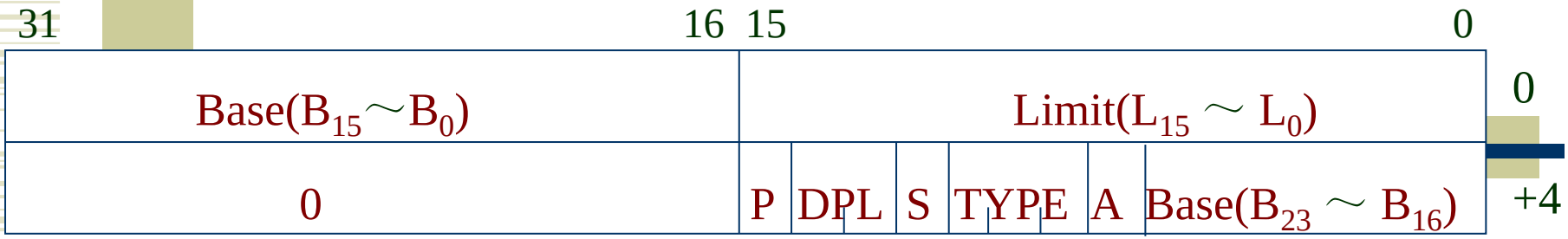
获得24、32位存储单元物理地址



2. 描述符（段的描述符）

- 描述符长度和组成
  - 8个字节
  - 由段基地址、界限、访问权限、附加字段4部分组成
- 用途：用来说明段在存储器中的位置、段的大小、控制和状态信息
- 描述符存放在存储器中，由操作系统从磁盘装入指令代码和数据段时自动分配设置，另外操作系统还要修改选择器内容

80286描述符



Base：段基地址，24位；Limit：段长度，2^{16}=64KB；
P：存在位；DPL：特权级（0-3）；　　S：1 系统段，0 应用程序段；
TYPE（E,ED/C,W/R）：E 可执行位，E=0不可执行段（数据段），E=1可执行代码段；ED/C 地址扩展方向位，ED=0，向上扩展，ED=1，向下扩展，W/R 可读/写位；
A：已访问位

80386/80486/Pentium描述符

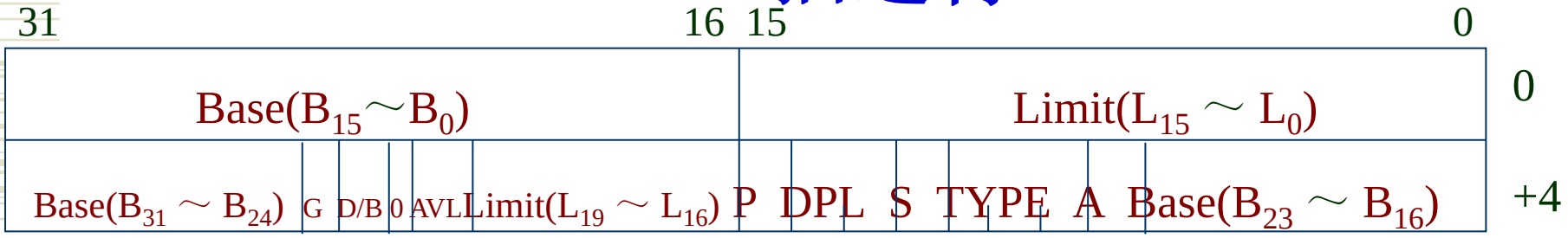
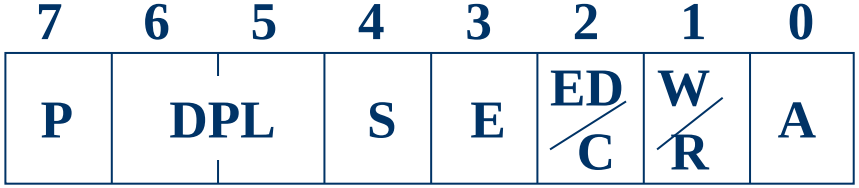


图2.11　描述符格式

访问权限字节



P位：存在位

P=0，段不在内存

P=1，段在内存中

DPL：段特权级

取值0~3，允许访问该段的最低特权级

S位：段描述符

S=1 系统段

S=0 应用程序段

A位：已访问位

A=0，段尚未被访问

A=1，段已被访问

E位：可执行位

E=0，不可执行代码段(数据段)

ED=0，段向上扩展为数据段

ED=1，段向下扩展为堆栈段

W=0，数据段只读

W=1，数据段可写

E=1，可执行代码段(代码段)

C=0，忽略描述符特权级

C=1，遵循描述符特权级

R=0，代码段不可读，即只执行

R=1，代码段可读

G位(粒度位)：

G=0, 段的长度以字节为单位

段长最大1M字节

G=1,段的长度以页(4K字节)为长度单位

段长最大1M×4K=4G字节

D位：D=0，16位指令方式

D=1，32位指令方式

AVL位：AVL=0，程序不可使用本段

AVL=1，程序可以使用本段

3. 选择器和描述符表

选择器存放在段寄存器中，16位长，格式位：

INDEX：所选描述符在描述符表中的偏移地址；

TI：指定描述符表

TI=0 从全局描述符表 GDT 中取描述符

TI=1 从局部描述符表 LDT 中取描述符

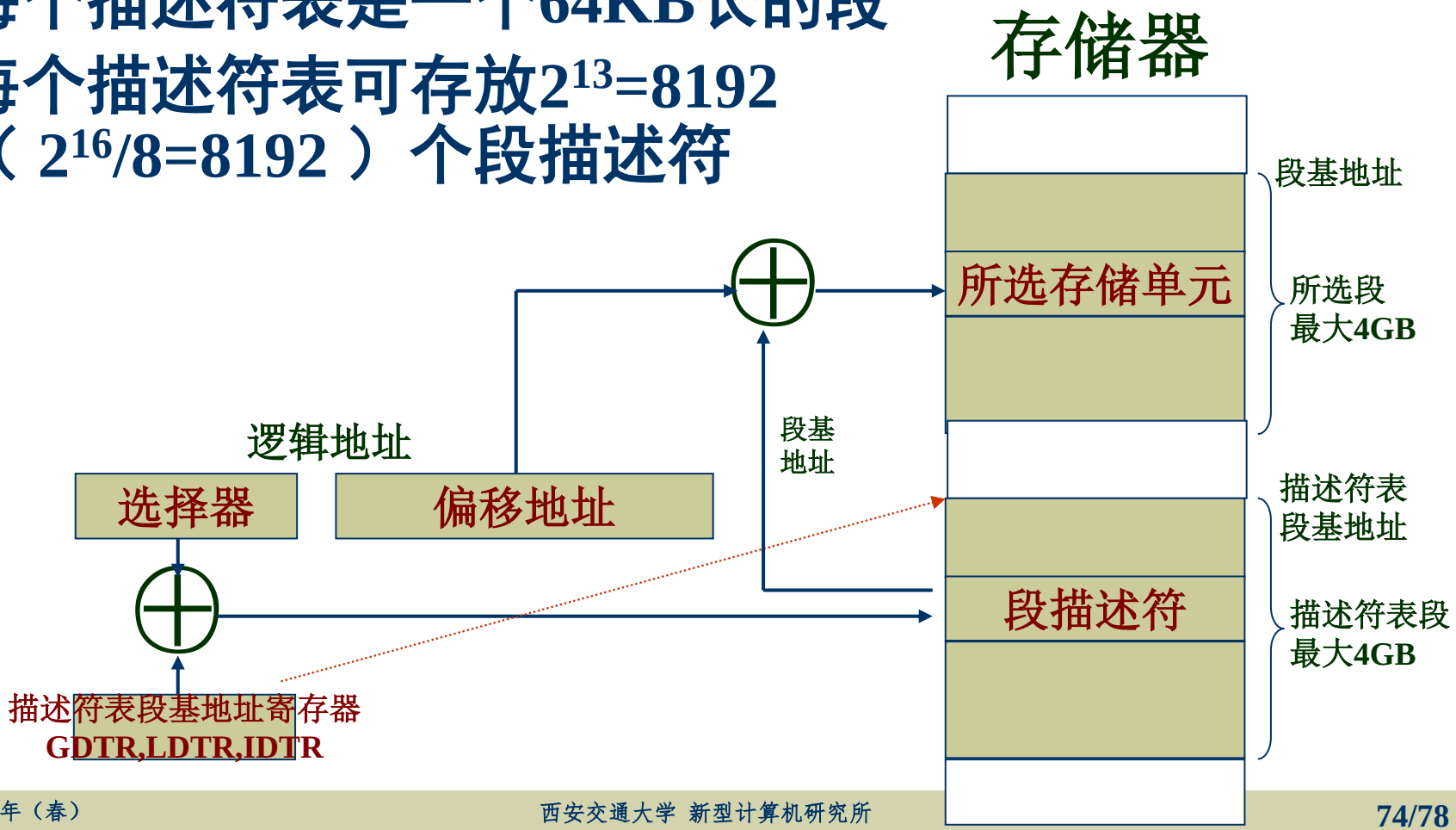
RPL：请求访问特权级，RPL≤DPL 才可以访问该段，取到描述符

描述符在描述符表中，一共有3类描述符表：

- 全局描述符表（GDT）【整个系统只有一个】
  - 其中的描述符指定的段可以用于所有的程序（如操作系统所用段）
- 局部描述符表（LDT） 【可以有多个】
  - 所指定的段通常只用于一个用户程序
- 中断描述符表（IDT） 【整个系统只有一个】

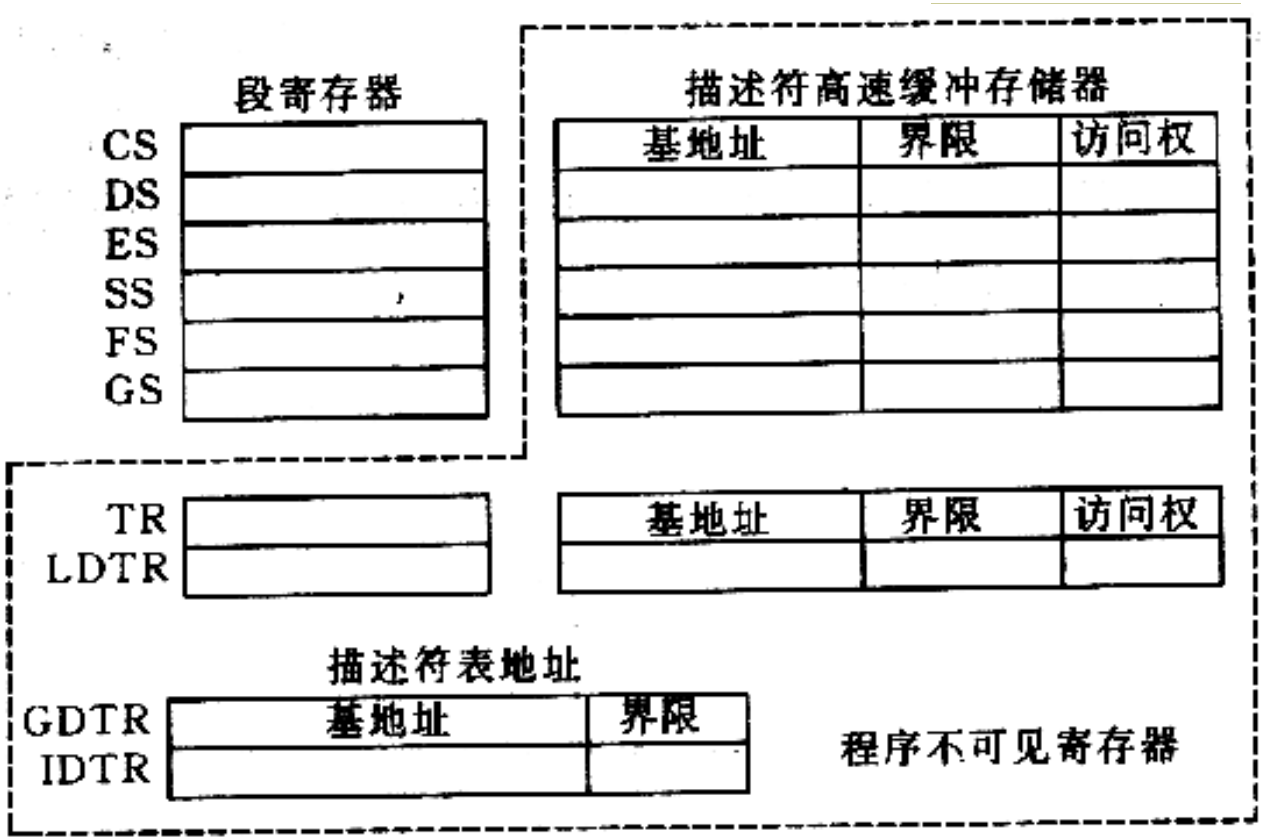
保护模式存储器寻址过程

- 描述符表存放在存储器中
- 每个描述符表是一个64KB长的段
- 每个描述符表可存放2^{13}=8192（$2^{16}/8$=8192）个段描述符



4. 程序不可见寄存器

- 指不能由用户程序访问而是只能由操作系统或硬件管理的寄存器
  - 如 GDTR, LDTR, IDTR 等



5. 描述符高速缓冲存储器

- 保护模式存储器寻址过程中多次访问主存，效率很低
  - 解决办法，采用cache原理，将常用（正在使用段）的描述符放在描述符高速缓冲存储器中
  - 描述符高速缓冲存储器在CPU中，访问速度与寄存器一样
  - 描述符高速缓冲存储器容量很小

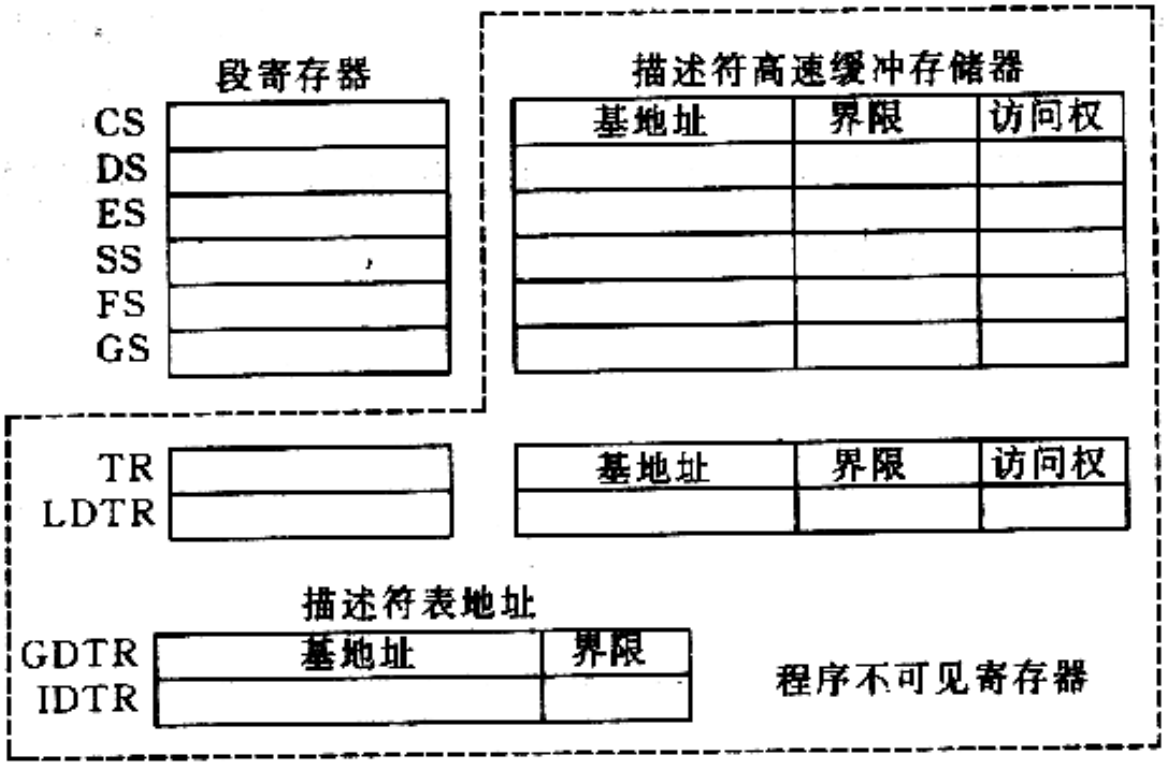
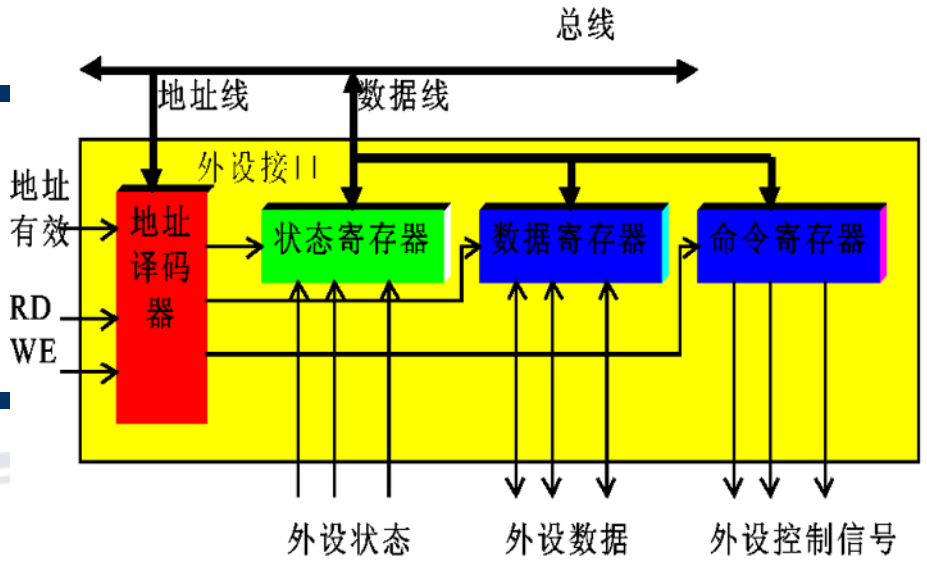


图 2.13　80x86 的程序不可见寄存器

2.5 外部设备



外部设备

2.5.1 外部设备简介

输入、输出设备，大容量的外存储器。

2.5.2 外设接口寄存器

- CPU对外部设备控制通过外设接口寄存器
  1. 数据寄存器：数据传送
  2. 状态寄存器：查看外部设备状态
  3. 命令寄存器：控制外部设备工作
- 外设接口寄存器访问（编址方式）

  单独编址：用专用指令访问，如output, input

  统一编址：用访存指令，如mov等

习　　题

P35

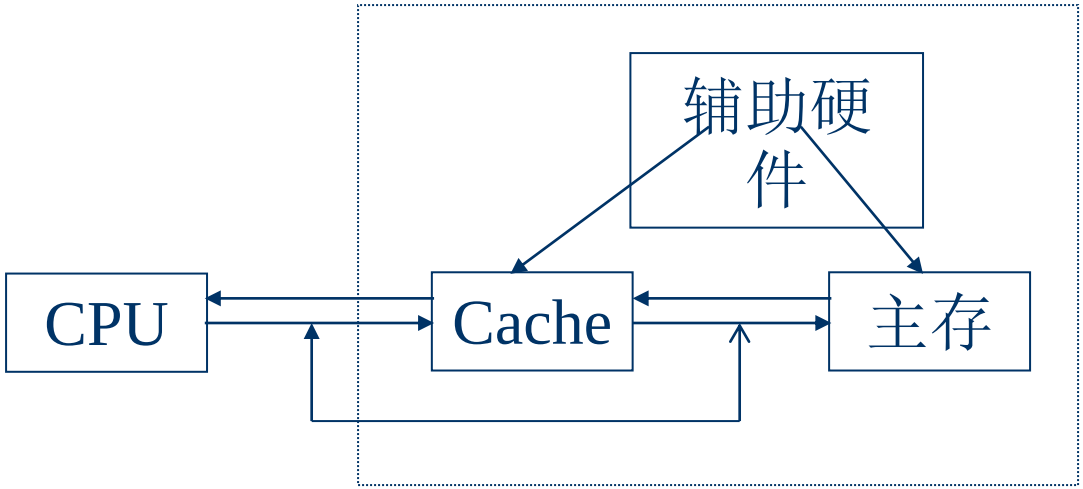
2. 1　　2. 2　　2. 3　　2. 4　　2. 5

2. 6　　2. 8　　2. 9　　2. 15

谢谢！

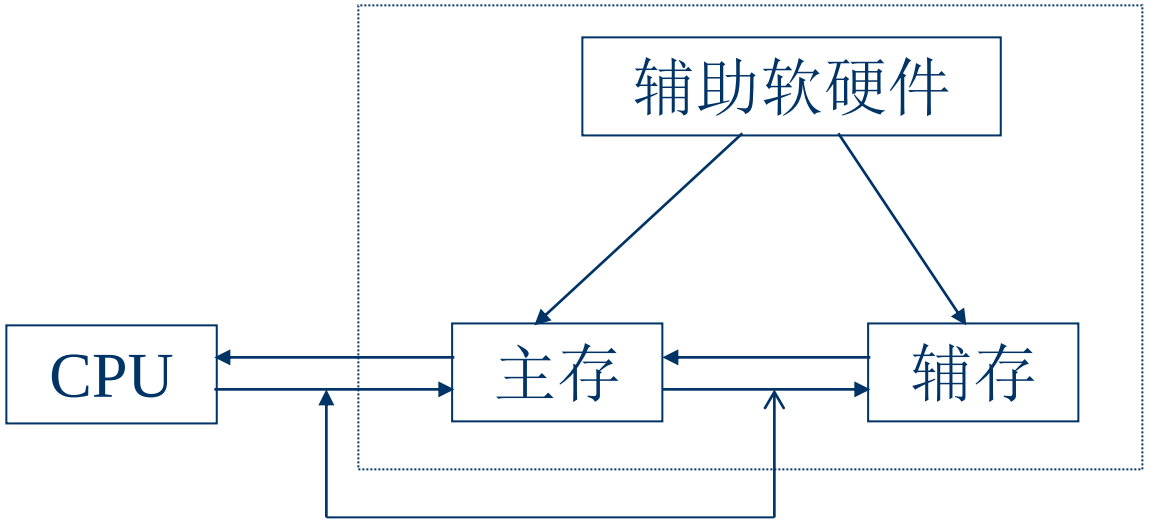
存储系统的一般组成

- 存储系统：Cache-RAM、 RAM-Disk
- Cache存储系统：(Cache-RAM)的设置目标是提高速度，由**硬件实现管理**，对操作系统设计者透明；



- Cache的速度：T_{Cache} =x ns~xxns，SRAM
- RAM的速度：T_{RAM} =xx ns ~xxx ns，DRAM
- T_{RAM}≈5~10 T_{Cache}

• **虚拟存储系统：(RAM-Disk)的目标是扩充RAM容量，由操作系统实现管理，对操作系统设计者不透明,对应用程序设计者透明。**



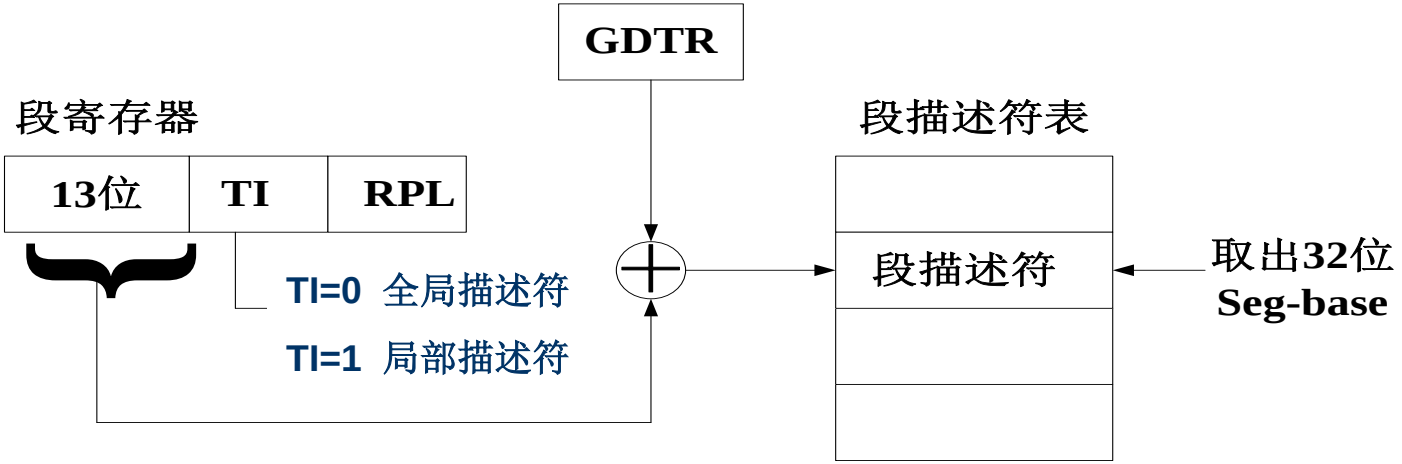
- **RAM的速度：T_{RAM} =xx ns ~xxx ns**
- **Disk的存取速度:T_{DISK}=x ms ~xx ms**

386/486/586中的虚拟地址转换成实地址详细过程

举例：Mov Ax，Es：[EBX+100]

（1）虚地址=（16位段选择字，32位偏移地址值）

（2）段描述符变址值=段选择字的高13位；TI值确定对应的段描述符表基地址值在GDTR或LDTR内；两者相加得到段描述符在描述符表中的位置；从段描述符中取出段基地址。



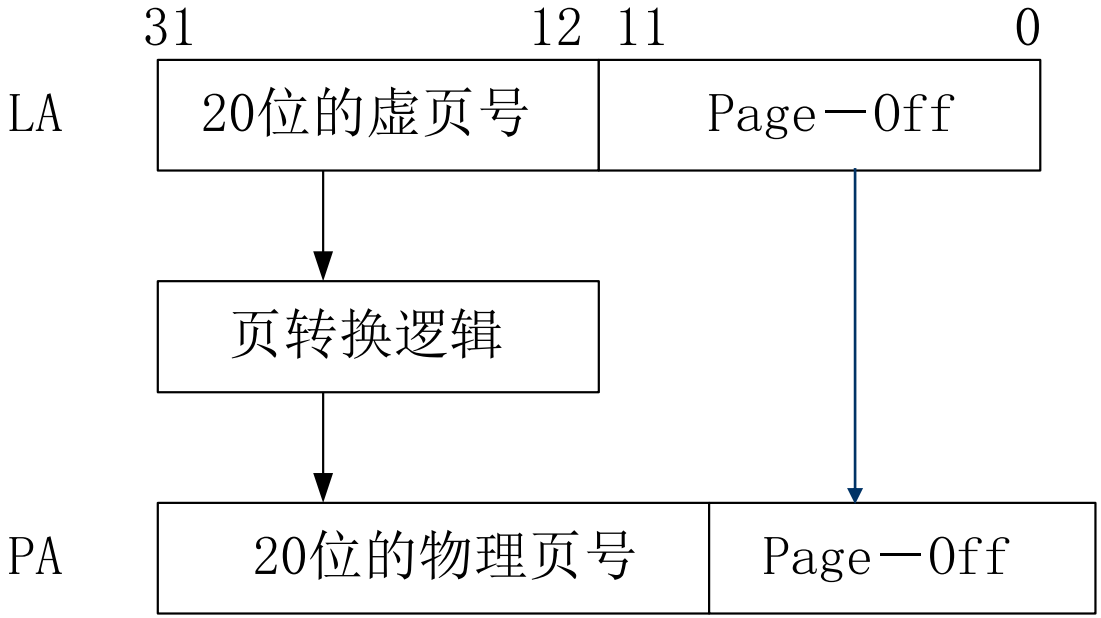
（ 3）线性地址=Seg-Base+32位的段内偏移地址值

Offset LA=Seg-Base +EBX+100

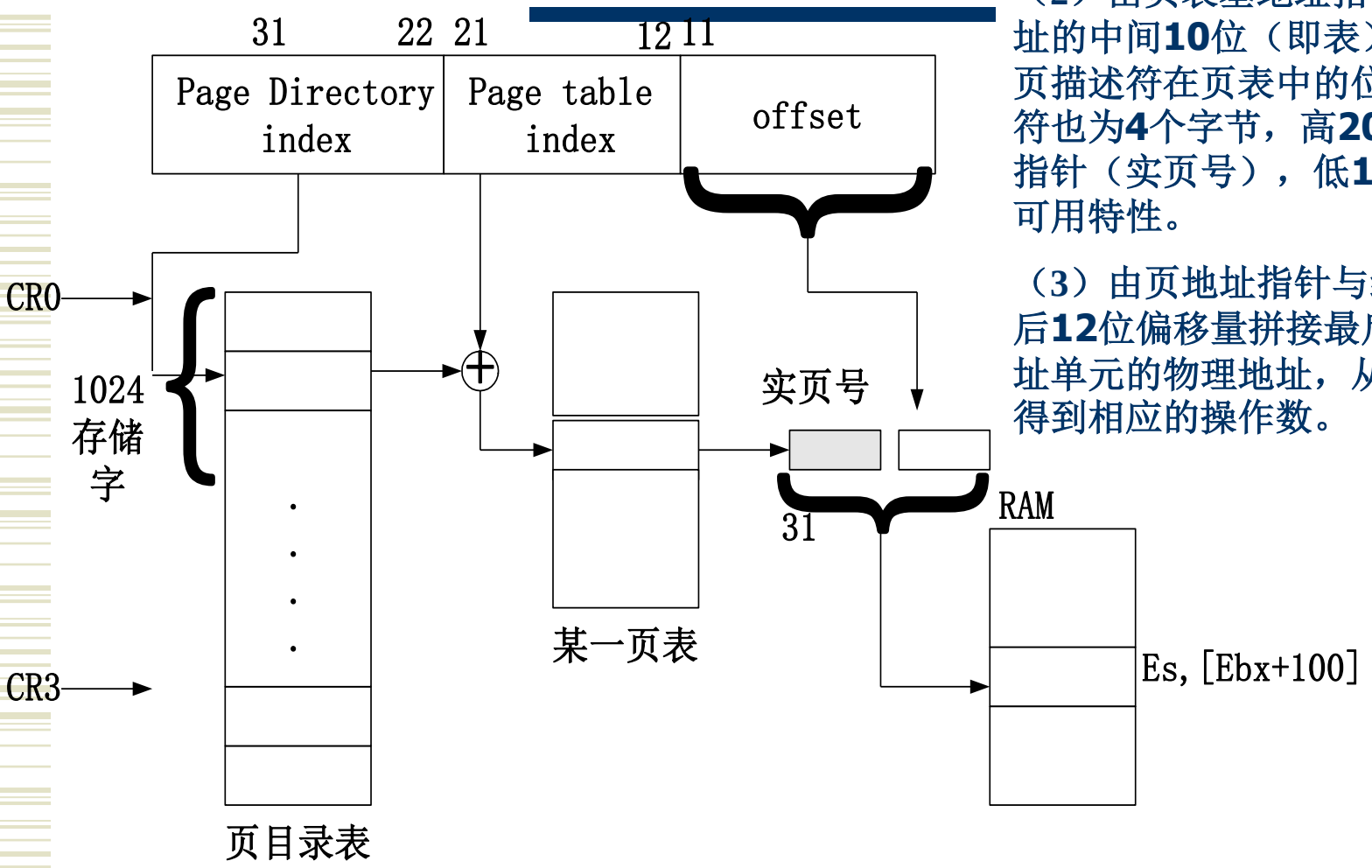
至此完成段式管理。

页式管理通过：

- 线性地址内容分成三部分：高**10**位指向页面目录表的偏移量；中间**10**位指向页表的偏移量；低**12**位是所寻址的操作数所在页的偏移地址。
- 根据分页机制将LA转换成物理地址PA



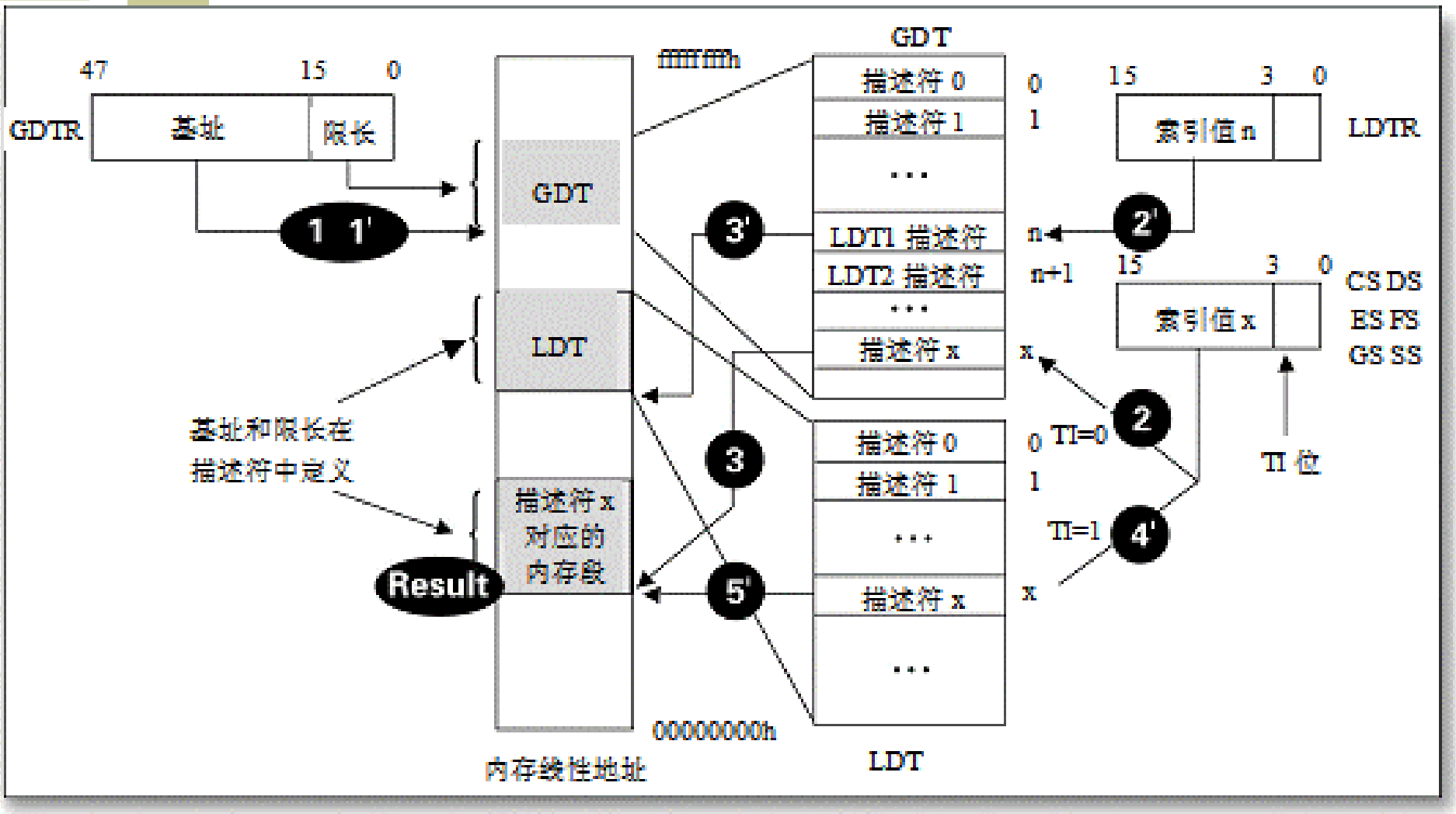
页转换：二级查表完成



- （1）由目录基地址（在**CR0-3**内）与线性地址高**10**位（即目录）相加得到页目录描述符在页目录表中的位置，页目录描述符为**4**个字节，高**20**位是页表地址指针，低**12**位表示页表可用特性。

（2）由页表基地址指针与线性地址的中间**10**位（即表）相加得到页描述符在页表中的位置。页描述符也为**4**个字节，高**20**位是页地址指针（实页号），低**12**位表示页可用特性。

（3）由页地址指针与线性地址的后**12**位偏移量拼接最后得到该寻址单元的物理地址，从物理地址可得到相应的操作数。


GDT
47
15
0
GDTR
基址
限长
1
1'
GDT
LDT
描述符 x
对应的
内存段
Result
基址和限长在
描述符中定义
ffffffffh
00000000h
内存线性地址
3'
3
5'
描述符 0
描述符 1
...
LDT1 描述符
LDT2 描述符
...
描述符 x
0
1
n
n+1
x
描述符 0
描述符 1
...
描述符 x
...
0
1
x
LDT
15
3
0
索引值 n
LDTR
2'
15
3
0
索引值 x
CS DS
ES FS
GS SS
TI 位
2
TI=0
TI=1
4'

第三章　指令系统和寻址方式

3.1　80X86寻址方式

3.2　80X86机器语言指令概况

3.3　80X86指令系统

本 章 目 标

了解计算机的一般指令格式

掌握80X86CPU寻址方式

掌握80X86CPU指令系统

熟练掌握80X86CPU常用指令功能

计算机的一般指令格式

- **指令系统**：一组指令集。计算机所能执行的所有指令的集合就是指令系统
  - CPU依靠机器指令来计算和控制系统
  - 每款CPU在设计时就规定了一系列与其硬件电路相配合的指令集
- 指令系统决定了计算机能执行的全部基本操作。
- 要使计算机完成一个特定的任务，就要告诉计算机按照怎样的　序执行一个个基本操作（指令），这个具有约定　序的一条条指令构成程序

常见的指令格式

指令：

操作码	操作数1	…	操作数n

零地址指令	RET IRET
一地址指令	INC AX DEC CX
二地址指令	MOV AX, [2000H] ADD AH, BL
三地址指令	很少使用

80x86最常用

学习指令系统关键点：
①要熟记指令操作码助记符；
②正确给出操作数

80X86寻址方式

寻址方式：指令指定操作数地址的方式

1、与操作数据有关的寻址方式

2、与转移地址有关的寻址方式

操作数通常保存在

（1） 指令中　　MOV AX, 2000H

（2） CPU的寄存器中　　MOV AX, BX

（3） 内存单元中　　MOV AX, [2000H]

（4） I/O接口寄存器中　　IN AH, 20H

3.1 80X86寻址方式

机器指令格式： OP dst, src

OP 操作码

src 源操作数

dst 目的操作数

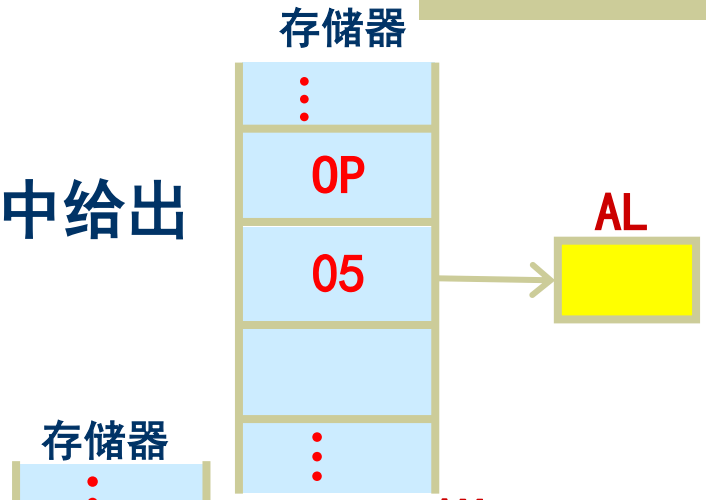
例： MOV dst, src 把src送到dst

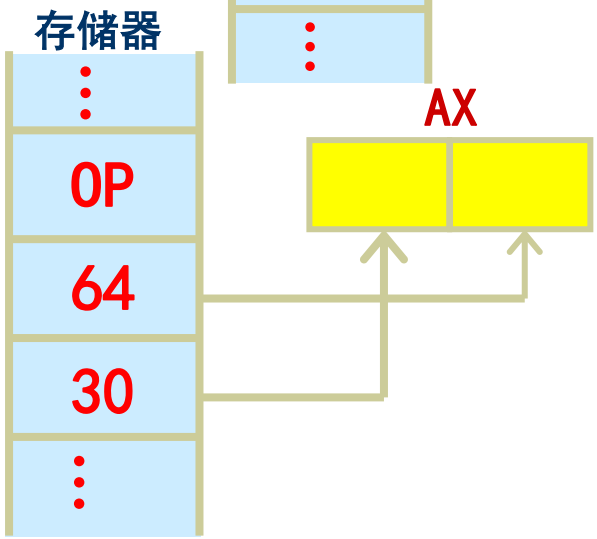
3.1.1 与数据有关的寻址方式

1. 立即寻址方式

立即寻址方式 —— 操作数在指令中给出

```
MOV   AL, 5
MOV   AX, 3064H
MOV   EAX, 88663064H
```





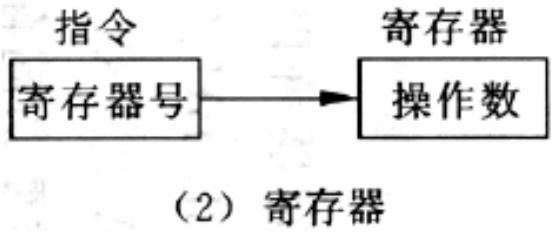
* 经常用于给寄存器赋初值
* 只能用于SRC字段
* SRC 和 DST的字长一致

```
MOV   AH, 3064H
```

2. 寄存器寻址方式

寄存器寻址方式* —— 操作数在指定的寄存器中

```
MOV   AL, BH
MOV   AX, BX
MOV   EAX, EBX
```



(2) 寄存器

* 字节寄存器只有 AH AL BH BL CH CL DH DL
* SRC 和 DST的字长一致

```
MOV   AH, BX
```

* CS不能用MOV指令改变

```
MOV   CS, AX
```

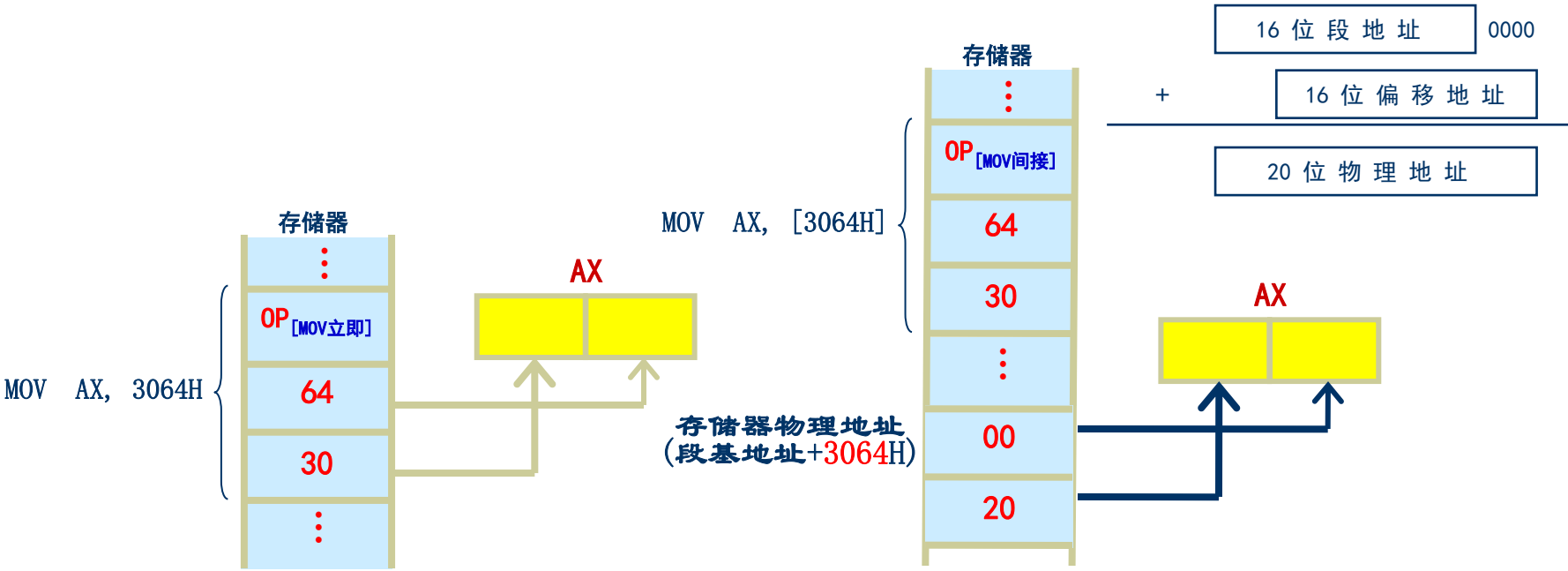
```
例1 MOV   BX, AX          ;BX ← AX
例2 MOV   DI, 5678H       ;DI ← 5678H
例3 MOV   AL, 78H         ;AL ← 78H
例4 MOV   ECX, 7890ABCDH  ;ECX ← 7890ABCDH
```

BX、AX、DI、AL、ECX均为寄存器寻址方式

说明： 除立即寻址和寄存器寻址外，以下各种寻址方式的操作数都在除代码段以外的存储区中。

- **如何得到操作数呢？！**
  - 通过不同寻址方式求得操作数在存储器中的地址，从而得到操作数

    物理地址=段基地址+偏移地址
  - 如何指定操作数的段基地址和偏移地址？



课内测试 02-2-1

1. 请在填空中[填空1] 填写“5”；（10分）

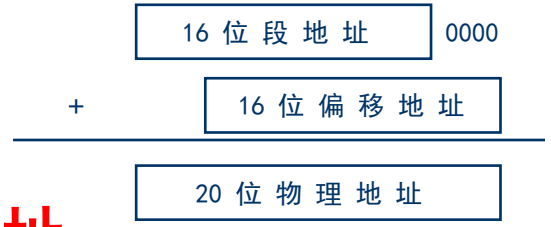
2. 指令MOV AX,3064H中，目的操作数“AX”是 [填空2] 寻址方式，源操作数“3064H”是 [填空3] 寻址方式。（ 10分）

操作数的段基地址表示和生成

- 操作数的段基地址用段寄存器获得
  - 若工作在实模式：段基址为段寄存器中的内容乘以16的值

    段基地址= 段寄存器 × 16D

    段基地址= 段寄存器 × 10H
  - 若工作在保护模式：段基址通过段寄存器中的段选择子从描述符中得到



- 指令中给出的操作数逻辑地址格式为

  段寄存器名：偏移地址

  - 段寄存器名如何指定
    - 段寄存器名可以缺省，这时使用　认约定的段寄存器
    - 段寄存器名可以特别指定，用于跨越段访问，但也要遵循一定的规则
  - 偏移地址如何指定

段基址和偏移量的约定情况

段寄存器名：偏移地址

操作类型	约定段寄存器	允许指定的段寄存器	偏移量
1. 指令	CS	无	IP
2. 堆栈操作	SS	无	SP
3. 普通变量	DS	ES、SS、CS	EA
4. 字符串指令的源串地址	DS	ES、SS、CS	SI
5. 字符串指令的目标串地址	ES	无	DI
6. BP用作基址寄存器	SS	DS、ES、CS	EA

数据操作

操作数的偏移地址表示和生成

- 操作数的偏移地址又称为**有效地址**EA，所以下述各种寻址方式即为求得有效地址EA的不同途径以及在指令中如何表示
- 有效地址EA的四种成分：

  基址、变址、比例因子、位移量

  (1) **位移量**（displacement）是指令中指定的一个8位、16位或32位的数，它不是立即数而是一个地址的位移量

(2) **基址**（base）是存放在基址寄存器中的内容

- 它是有效地址中的基址部分，通常用来指向数据段中数组或字符串的首地址

(3) **变址**（index）存放在变址寄存器中的内容

- 它通常用来指定数组中的某个元素或字符串中的某个字符

(4) **比例因子**（scale factor）是386及其后继机型新增加的寻址方式中的一个术语

- 其值可为1，2，4或8
- 在寻址中，可用变址寄存器的内容乘以比例因子来取得变址值
  - 这类寻址方式对访问元素长度1、2、4、8字节的数组特别有用

有效地址的计算

- 有效地址的用下式计算

  EA=基址+（变址*比例因子）+位移量

  在这4个成分中，除比例因子是固定值外，其他3个成分的数值都可正可负，以保证指针移动的灵活性。

- 注意：
  - 8086/80286只能使用16位有效地址寻址
  - 80386及其后继机型既可用32位有效地址寻址，也可用16位有效地址寻址

表3.1：16/32位有效地址寻址时有4种成分组成

	16位寻址	32位寻址
位移量	0，8位，16位	0，8位，32位
基址寄存器	BX，BP	任何32位通用寄存器（包括ESP）
变址寄存器	SI，DI	除ESP以外的32位通用寄存器
比例因子	无	1，2，4，8

EA=基址+（变址*比例因子）+位移量

指令中操作数的偏移地址（有效地址）表示

- 指令中操作数的有效地址表示形式：

  [基址][变址*比例因子][位移量]

  或者

  [基址+变址*比例因子+位移量]

  - 比例因子必须与变址一起组合使用
  - 基址、变址、比例因子、位移量 4个成分有8种组合，即8种寻址方式

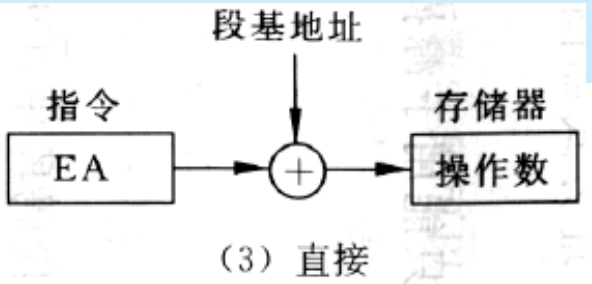
- 指令中操作数的地址表示形式：

  段寄存器名：[基址+变址*比例因子+位移量]

[基址+变址*比例因子+位移量]，只有位移量情况

EA=[位移量]

3. 直接寻址方式



- 操作数存放在存储器的存储单元中，有效地址EA由指令直接给出

  例：MOV BX, [2100H]　　；DS：[2100H] →BX

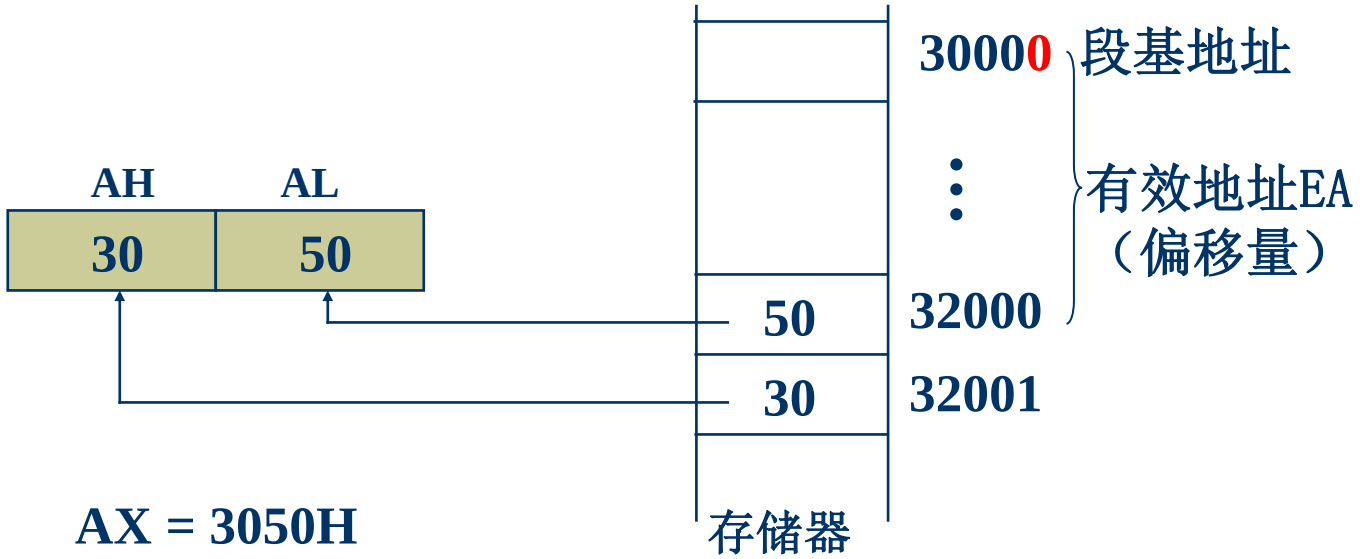
  - [ ]内为操作数在段内的偏移地址
  - 如果指令中缺省段寄存器名，直接寻址的段寄存器约定为数据段DS

    物理地址计算公式：　PA = DS × 16 + EA
  - 如果数据存放在其他段如附加数据段ES(代码段CS，堆栈段SS)，偏移量XXXX处
    - 操作数地址表示为ES：[XXXX]
      - 对应的指令，如 MOV　BX, ES：[100H]
    - 操作数物理地址计算时也要采用ES段

      物理地址计算公式如下：　PA = ES × 16 + EA

EA=2000H，假设DS=3000H

那么，PA=DS×10H+EA=30000+2000=32000H



- 操作数地址可由符号地址表示

```
MOV   AH, VALUE
MOV   AH, [VALUE]
```

VALUE的属性应该为地址

  - 两者等效，这时VALUE是操作数的符号地址，但必须在程序中提前定义属性和数值

- 使用变量时，要注意变量的属性

```
VALUE   DB   10
×       MOV   AX, VALUE
√       MOV   AX, WORD PTR VALUE
```

- 实际上在汇编语言源程序中一般所看到的直接寻址方式都是用符号表示的，只有在DEBUG环境下，才有[2000H]这样的表示

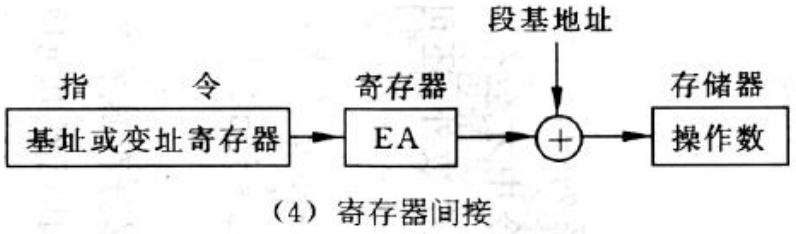
直接寻址方式举例

```
data segment
value db 66H
other db 88H
arr   db 12h, 34h, 56h, 78h
data ends

code segment
mov ax, data
mov ds, ax
mov al, value
mov cl, [value]
mov ax, word ptr value
mov bx, word ptr [value]
…
code ends
```

```
C:\LEARN>debug ch03_dma.exe
-u 0 f
077F:0000 B87E07        MOV     AX,077E
077F:0003 8ED8          MOV     DS,AX
077F:0005 A00000        MOV     AL,[0000]
077F:0008 8A0E0000      MOV     CL,[0000]
077F:000C A10000        MOV     AX,[0000]
077F:000F 8B1E0000      MOV     BX,[0000]
-r
AX=FFFF  BX=0000  CX=00E6  DX=0000  SP=0000
DS=076E  ES=076E  SS=077D  CS=077F  IP=0000
077F:0000 B87E07        MOV     AX,077E
-d 0 5
076E:0000  CD 20 FF 9F 00 EA                              . ....
-g5

AX=077E  BX=0000  CX=00E6  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=077E  ES=076E  SS=077D  CS=077F  IP=0005   NV UP EI PL NZ NA PO NC
077F:0005 A00000        MOV     AL,[0000]                      DS:0000=66
-d 0 5
077E:0000  66 88 12 34 56 78                            f..4Vx
```

4. 寄存器间接寻址方式

- 操作数存放在存储器单元中
- 16位的操作数偏移地址EA在指令指定的基址寄存器BX、BP或变址寄存器SI、DI中
  - EA=指定寄存器内容



(4) 寄存器间接

  - 物理地址的计算如下：

PA = DS × 16 + BX （或者SI、 DI）
PA = SS × 16 + BP

```
例: MOV   CX,   [SI]    ; (DS×16+SI) →CX, 源操作数用SI
                        ; 寄存器间接寻址方式
    MOV   CX,   [BP]    ; (SS×16+BP) →CX, 源操作数用BP
                        ; 寄存器间接寻址方式
```

- 32位的操作数偏移地址由指令指定的8个扩展寄存器指定。 EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

例. MOV AX,[BP] ；(SS:[BP]) →AX

- 若SS = 2000H，BP = 0080H

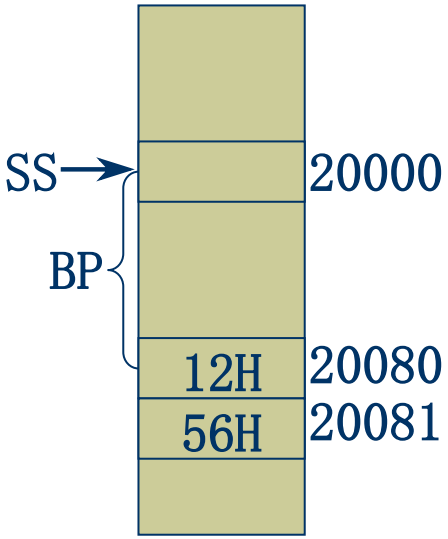
  存储单元（20080H） = 12H

  存储单元（20081H） = 56H

- 则物理地址 = 10H ×SS + BP

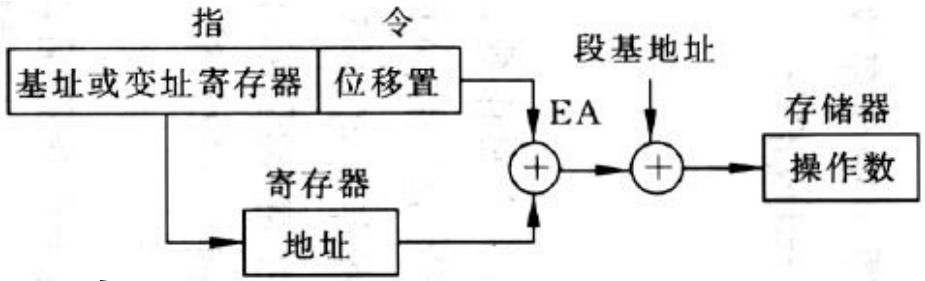
  = 20080H

- 该指令的执行结果是 AX = 5612H



5. 寄存器相对寻址方式

- 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的寄存器BX、BP、SI、DI和指令中给定的位移量相加得到



(5) 寄存器相对

- 物理地址计算：

$$PA = DS \times 16 + \begin{Bmatrix} BX \\ SI \\ DI \end{Bmatrix} + 位移量$$

$$PA = SS \times 16 + BP + 位移量$$

- 例：
  - MOV AX, NUM [BX] ; (DS×16+BX+NUM) →AX
  - MOV BX, CCC [BP] ; (SS×16+BP+CCC) →BX

例： MOV AX, COUNT[SI]

或 MOV AX, [COUNT+SI]

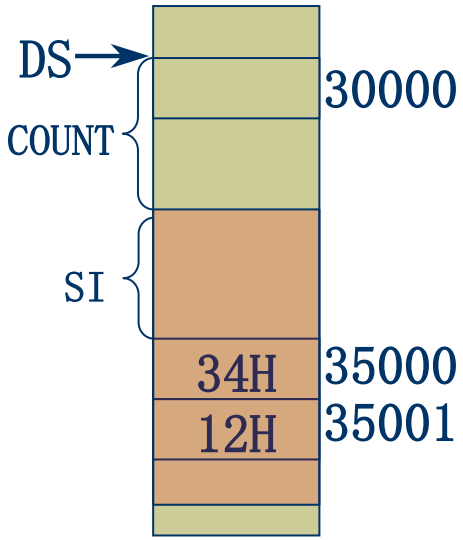
- 假设：DS=3000H, SI=2000H, COUNT=3000H

  (35000H)=1234H

- 那么： EA = SI+COUNT=2000+3000 = 5000H

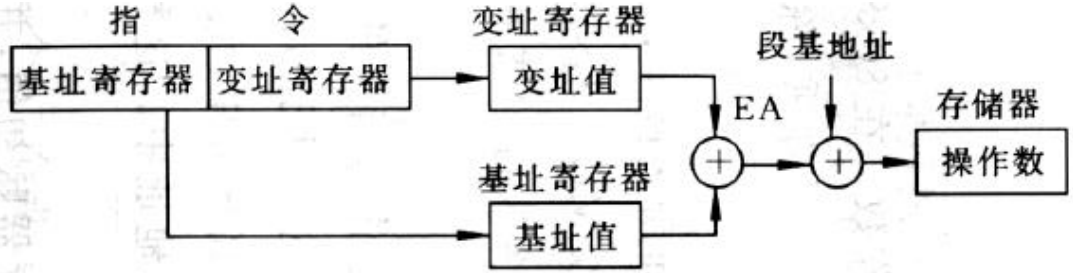
  PA = 30000+5000 = 35000H

- 则 AX=1234H



6. 基址变址寻址方式

- 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的基址寄存器（BX,BP）和变址寄存器（SI、DI）内容相加



（6）基址变址

- 物理地址计算：

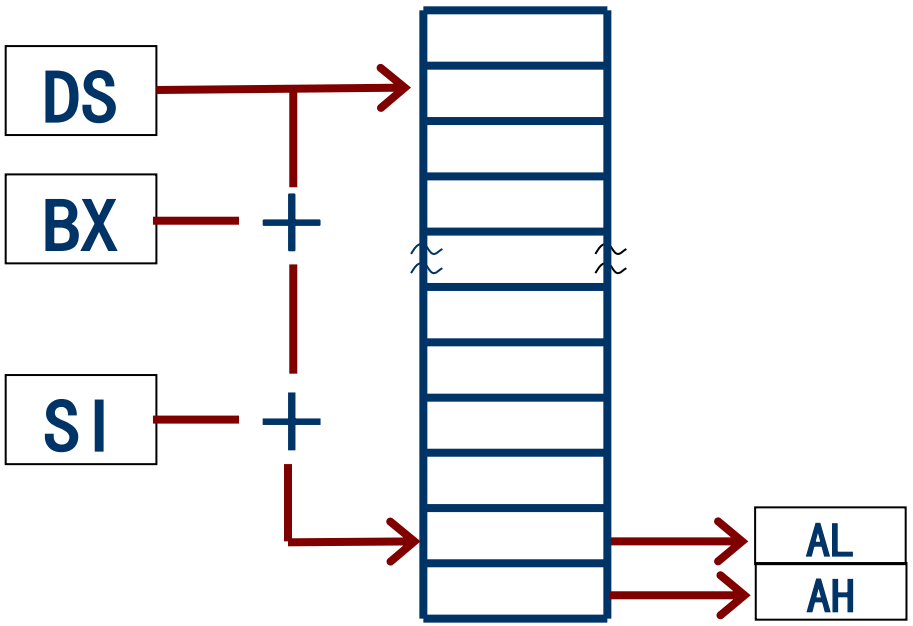
$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix}$$

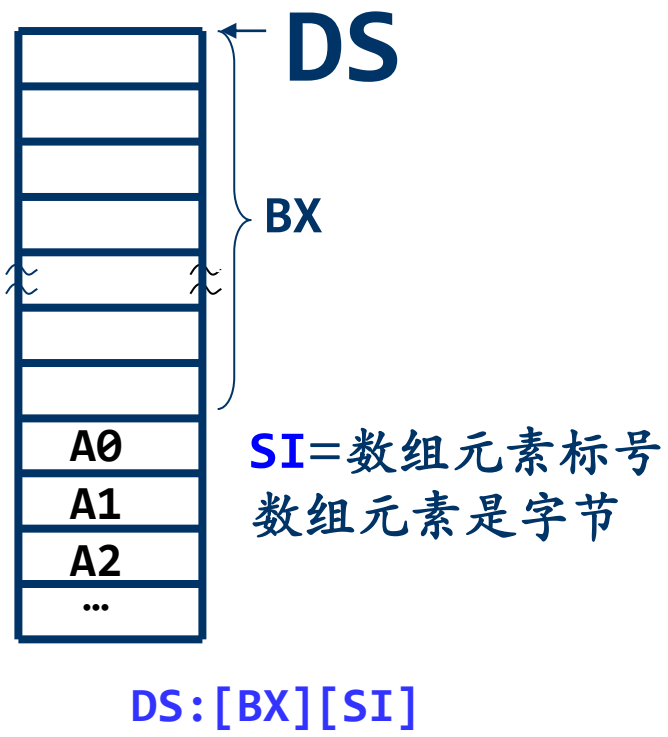
$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix}$$

- 例：MOV AX, [BX][SI] 或MOV AX, [BX+SI]
  (DS×16+BX+SI) →AX

MOV AX, [BX][SI]

基址变址寻址方式执行情况



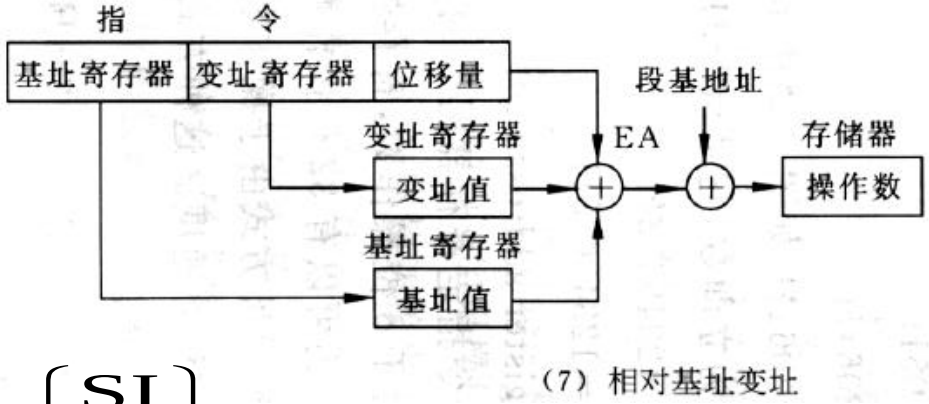


$$PA = DS \times 16 + BX + SI$$

[基址+变址*比例因子+位移量]，同时有基址、变址、位移量情况　　EA=[基址+变址+位移量]

7. 相对基址变址寻址方式

- 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的基址寄存器（BX、BP）和变址寄存器（SI、DI）以及位移量相加



（7）相对基址变址

- 物理地址计算：

$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + 位移量$$

$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + 位移量$$

MOV AX, MASK[BX][SI],
MOV AX, MASK[BX+SI],
MOV AX, [MASK+BX+SI] 等同

机器指令	汇编指令
B4 32	mov ah, 50
B8 50 00	mov ax, 50h ; mov ax, 0050h
A1 50 00	mov ax, ds:[50h] ; mov ax, [0050h]
8B C3	mov ax, bx
8B 07	mov ax, [bx]
8B 00	mov ax, [bx][si]
8B 40 50	mov ax, 50h[bx][si]
8B 80 50 07	mov ax, 0750h[bx][si]

使用寄存器的指令效率高（字节少、速度快）

课内测试 02-2-2

1. 请在填空中[填空1] 填写“7” ；（15分）
2. 指令MOV AX,[BX][SI]中，源操作数将从 [填空2]中获得，“[BX][SI]”是 [填空3] 寻址方式，认约定段寄存器是 [填空4] 。（15分）

寻址方式总结

(1) 3、4、5、6、7这五种寻址方式

- 所指定的操作数都存在内存单元中
- 处理器根据指令中给出的地址信息求出存放操作数的内存单元偏移地址(或有效地址)
  - 计算机根据指令给出的寻址方式求出操作数的偏移地址就是求出了操作数地址的段内偏移地址
- 再加上段基地址，得到操作数在内存中的物理地址
- 然后对存放在内存的操作数进行存取操作

寻址方式总结

(2) 4、5、6、7这四种寻址方式

- 使用了BX、BP、SI、DI寄存器
- 只要指令寻址时使用了BP，计算物理地址时约定段是SS段
- 指令寻址时使用了除BP以外的其它寄存器，计算物理地址时约定段为DS段

(3) 若操作数部分使用了段超越（即段更换前缀），则计算物理地址时使用超越段

- 如 MOV AX, ES:[100H]，计算物理地址时段地址为ES，这条指令的源操作数的物理地址=ES×16+100H

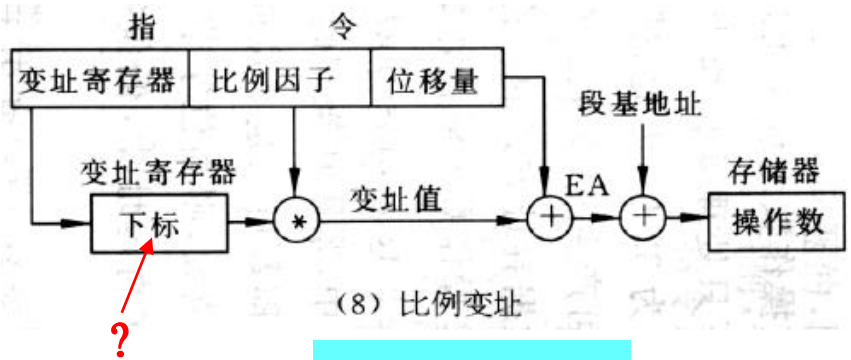
8. 比例变址寻址方式

- 相对比例变址寻址方式更好理解
- 32位地址寻址方式，80386以上微处理器
- 操作数的有效地址是变址寄存器的内容乘以指令中指定的比例因子，再加上位移量

EA=[变址*比例因子+位移量]

$$PA = DS \times 16 + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S + 位移量$$

$$PA = SS \times 16 + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S + 位移量$$



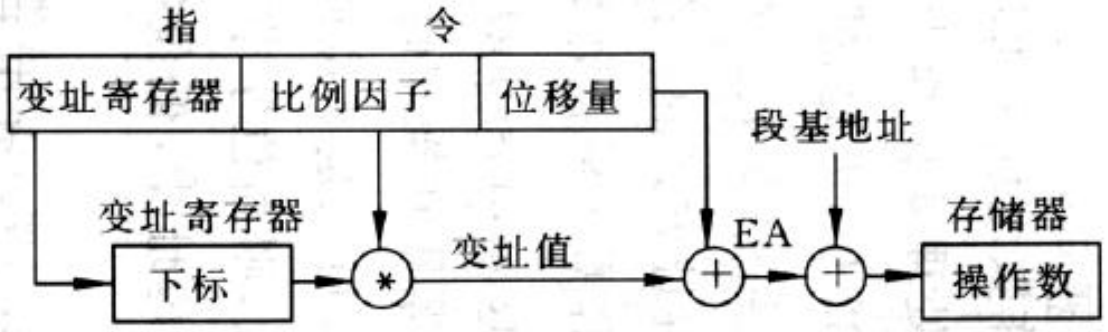
（8）比例变址

例如：MOV　EAX，NUM[ESI×8]

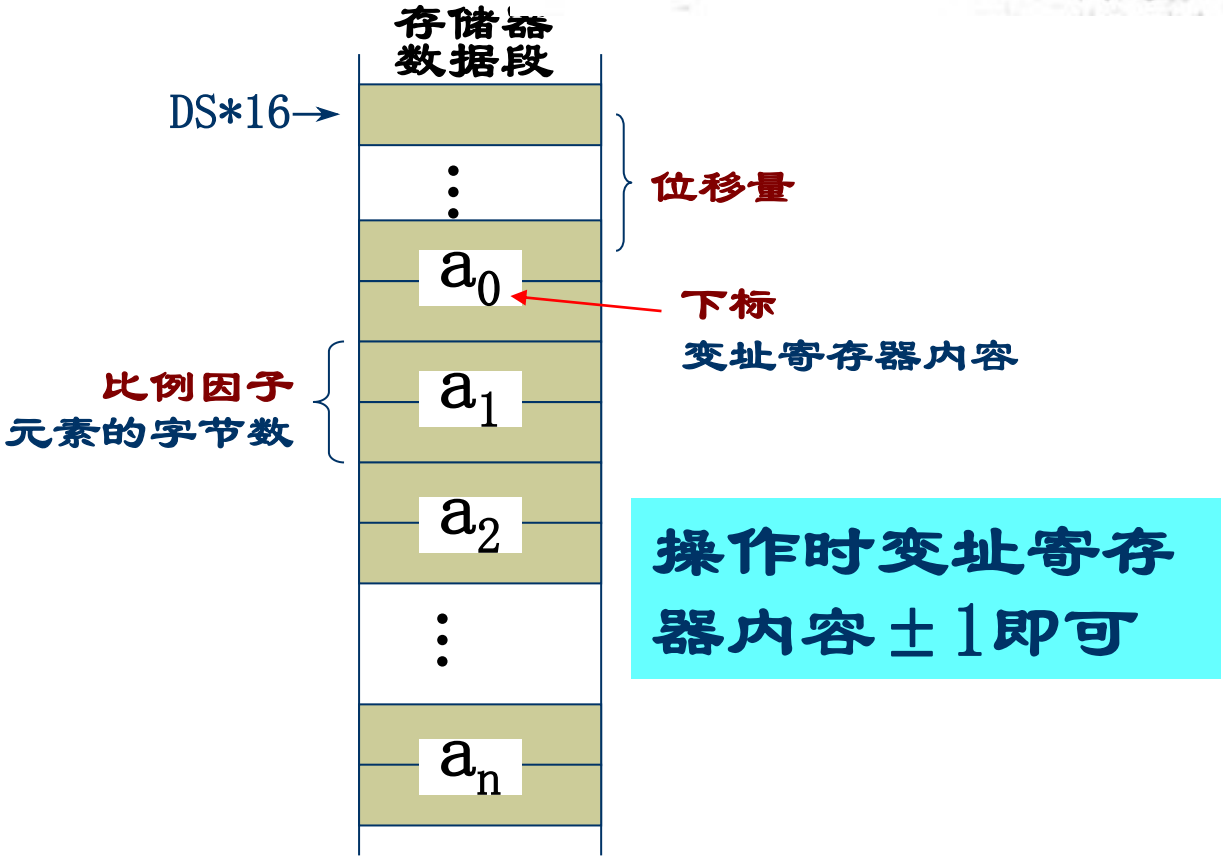
PA=DS×10H + ESI×8 + NUM

下标一定从0开始

字符串和数组处理使用



（8）比例变址



操作时变址寄存器内容±1即可

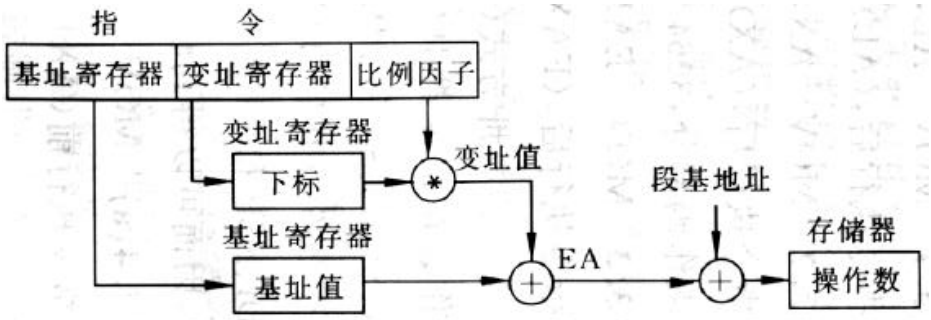
9. 基址比例变址寻址方式

- 32位地址寻址方式，80386以上微处理器
- 操作数的有效地址是变址寄存器的内容乘以比例因子再加上基址寄存器的内容

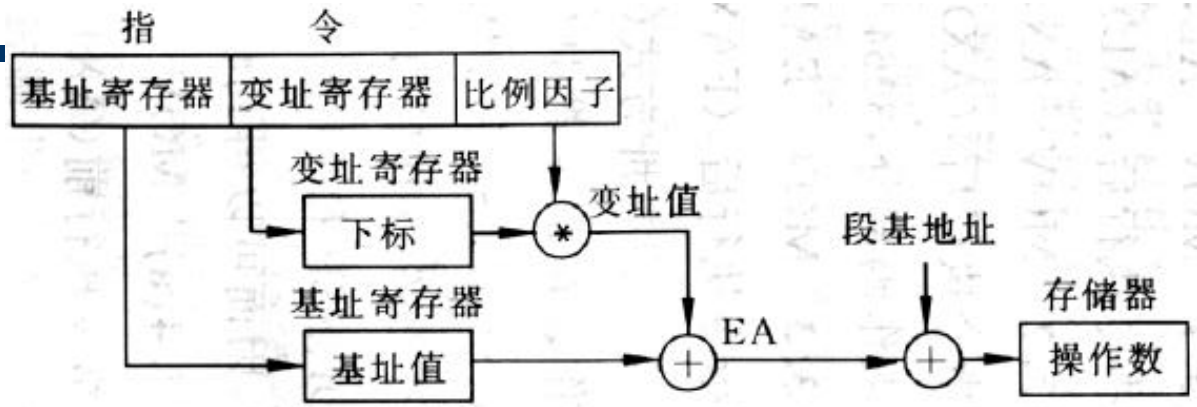
EA= [基址+变址*比例因子]

$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S$$

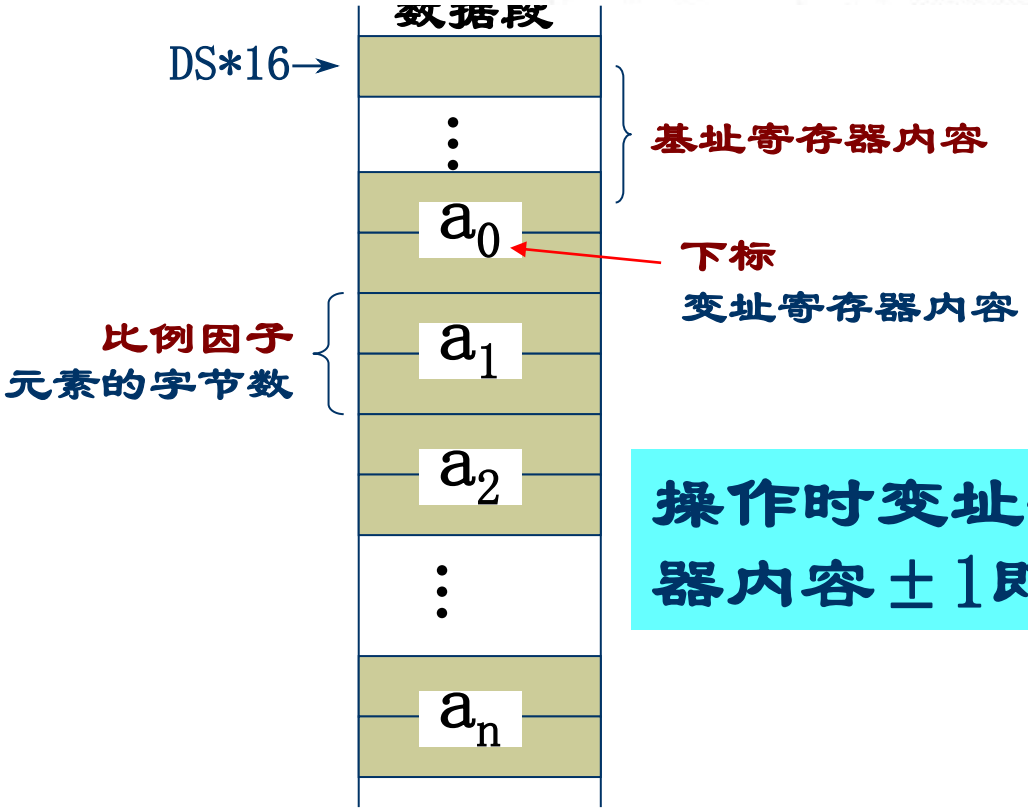
$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S$$



（9）基址比例变址



（9）基址比例变址



基址寄存器可根据数组起始地址在程序中设置，编程灵活方便

操作时变址寄存器内容±1即可

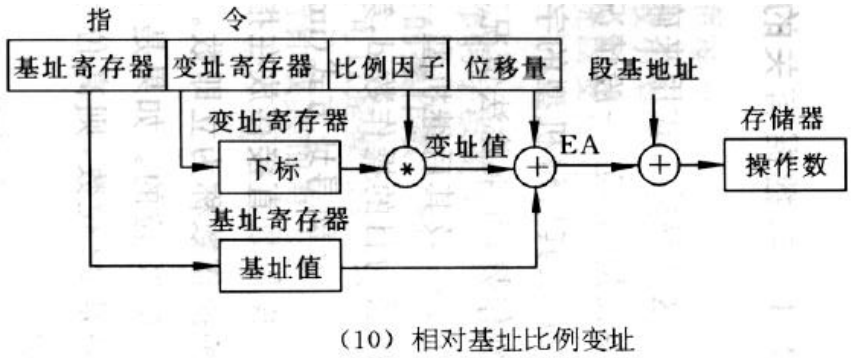
10. 相对基址比例变址寻址方式

- 32位地址寻址方式，80386以上微处理器
- 操作数的有效地址是变址寄存器的内容乘以比例因子，加上基址寄存器的内容再加上位移量

EA=[基址+变址*比例因子+位移量]

$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S + 位移量$$

$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix} * S + 位移量$$



（10）相对基址比例变址

例：MOV AX, NUM [BX][SI*2]

完成 (DS×16 + BX +SI*2+ OFFSET NUM) →AX

也可写成：MOV AX, [BX+SI*2+NUM]

多个字符串和数组处理使用

8086/8088寻址方式

与数据有关的寻址方式：以 MOV 指令为例

- 立即寻址　　MOV AX , 3069H
- 寄存器寻址　　MOV AL , BH
- 直接寻址　　MOV AX , [2000H]
- 寄存器间接寻址　　MOV AX , [BX]
- 寄存器相对寻址　　MOV AX , COUNT [SI]
- 基址变址寻址　　MOV AX , [BP] [DI]
- 相对基址变址寻址　　MOV AX , MASK [BX] [SI]

存储器寻址

3.1.2 与转移地址有关的寻址方式

```
again: cmp [si],0ffffh
       jz exit
          …
exit:  mov ah,4ch
```

- 改变程序执行顺序的指令
  - 若需要改变程序的正常执行顺序，转移到所要求的指令处执行程序时，可以安排一条转移指令或转子指令(CALL指令)
- 这种寻址方式主要是如何确定转移指令及CALL指令的转向地址
  - 程序指令地址是由代码段寄存器CS和指令指针IP决定

    CS*16d+IP
  - 功能就是修改CS和IP的值，确定目的地址
- 转移的目的地址位置
  - 可以段内转移（转移指令和转移的目的地址在同一代码段内）
  - 也可以段间转移（转移指令和转移的目的地址不在同一代码段）
  - 与转移地址有关的有四种寻址方式

3.1.2 与转移地址有关的寻址方式

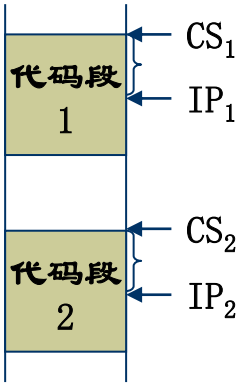
- 段内寻址
  - 段内直接寻址
  - 段内间接寻址

  如何求得目标指令的新IP
- 段间寻址
  - 段间直接寻址
  - 段间间接寻址

  如何求得目标指令的新CS、IP

段内寻址：转移指令与转向的目标指令在同一代码段中
CS内容不变，IP内容修改

段间寻址：转移指令与转向的目标指令在两个代码段中
CS和IP内容修改



直接寻址：转向的目标指令地址由转移指令直接指明

段间寻址：转向的目标指令地址由转移指令中的寄存器或存储单元内容给出

以无条件转移为例介绍

1. 段内直接寻址

- 转向的有效地址EA是当前IP寄存器的内容和指令中指定的8位或16位 位移量之和。如

```
code1  segment
       ...
       jmp    skip      ; skip 为转向的目标地址
       ...
       ...
skip:  nop
       ...
code1  ends
```

IP当前 → 指向 jmp skip 之后的下一条指令

skip数值=skip符号地址的有效地址-当前IP寄存器内容

JMP指令执行后，$IP_{新} = IP_{当前} + skip$

当程序执行完jmp skip指令后，就跳转到skip所指的nop指令执行。jmp和nop指令之间的指令不被执行。

转向的有效地址EA = $IP_{当前}$ + 位移量(8bit / 16bit) ⇒ $IP_{新}$

新指令物理地址 = CS × 16 + $IP_{新}$

例：

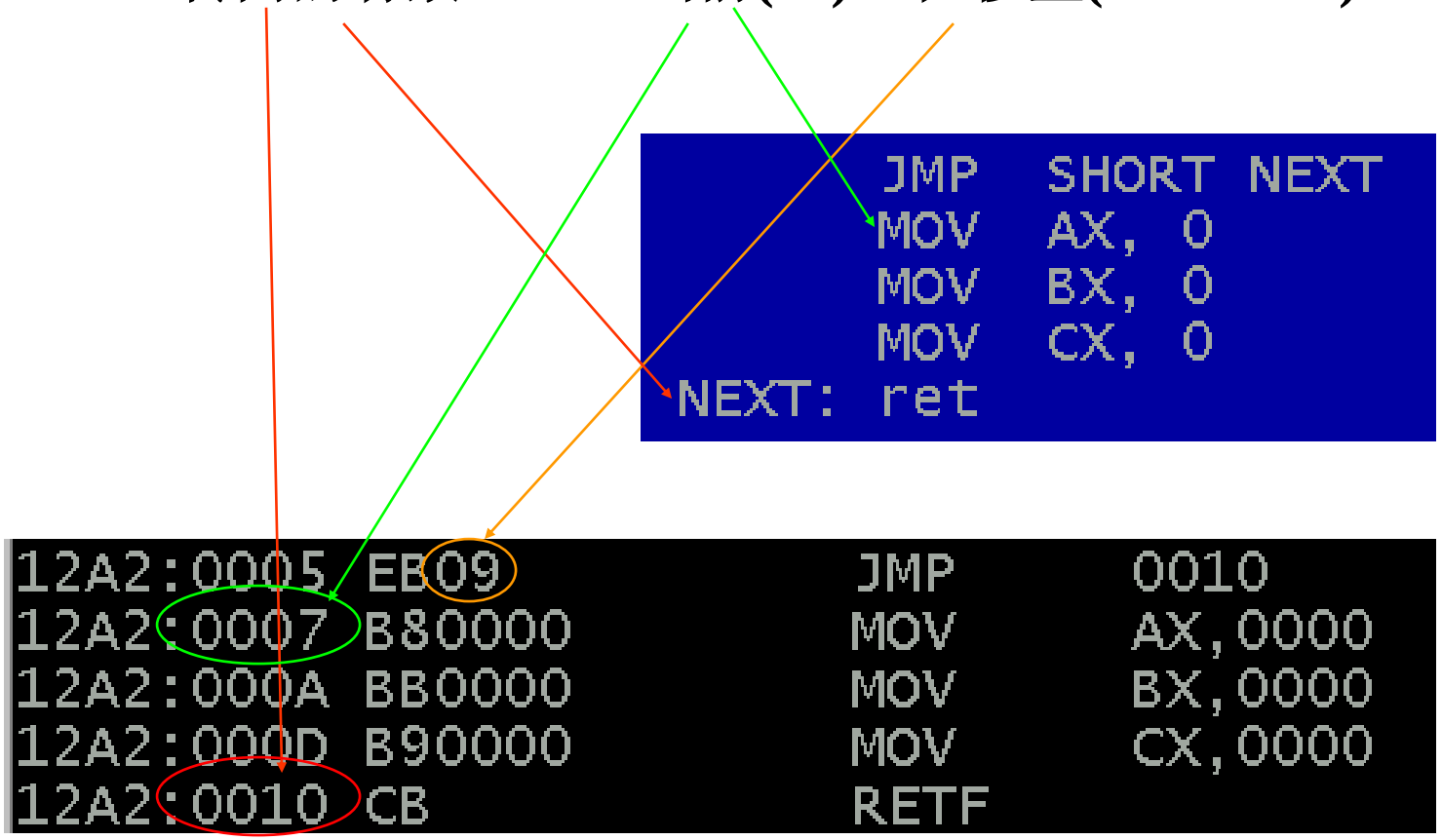
```
        ……
        JMP  AGAIN
(IP)当前 → MOV  BX, AX
        ……
(IP)新 → AGAIN: ……
```

位移量 (8bit / 16bit)

- 位移量为16bit时，近转移：
  JMP *NEAR PTR* NEXT ， 位移量范围：-32768 ~ +32767
- 位移量为8bit时，短转移：
  JMP *SHORT* NEXT ，位移量范围： -128 ~ +127

段内直接寻址示例

转向的有效地址 = 当前(IP) + 位移量(8bit/16bit)



2. 段内间接寻址

- 转向的目标指令的有效地址EA，即新IP内容是在寄存器或存储单元的内容
  - 跳转指令中给出寄存器名或存储单元的有效地址
    - 用数据寻址方式中除立即数以外的任何一种寻址方式
  - 取得转向的目标指令的有效地址
  - 用所得的目标指令的有效地址EA取代IP寄存器的内容即可实现段间跳转
- 新指令物理地址 = CS × 16 + $IP_{新}$
- 例如 jmp si

Offset 告诉汇编程序取相对于段基地址的偏移量

```
code1    segment
         assume  cs:code1
start:   mov   si, offset next   ; next位置的偏移地址送SI
         jmp   si                ; 转向next，寄存器间接寻址
         mov   ax, 1000h
         mov   bx, 2001h
next:    nop                     ; next是符号地址（或称标号）
          …
code1    ends
```

要想转到next处，只要将IP内容设置为next处的偏移量
CPU就会从next处读取指令并执行

- 当程序执行完mov si,offset next指令后，si中得到了next标号在代码段中的偏移量
- 执行jmp si指令后，SI内容送IP，程序就跳转到next所指的nop指令执行
- jmp si和next: nop指令之间的两条mov指令不被执行

段内间接寻址得到的有效地址是16位，属于近转移

例： BX=1256H, SI=528EH, TABLE=20A2H
DS=2000H, (232F8H)=3280H, (264E4H)=2450H

JMP BX ; BX →IP, $IP_{新}$=1256H

实模式下，寄存器寻址只能使用16位寄存器

JMP TABLE[BX]
JMP *WORD PTR* TABLE[BX] ; $IP_{新}$=3280H

DS*10H+BX+TABLE=20000+1256+20A2=232F8H (232F8H) →IP

JMP [BX][SI]
JMP *WORD PTR* [BX][SI] ; $IP_{新}$=2450H

DS*10H+BX+SI=20000+1256+528E=264E4H (264E4H) →IP

3. 段间直接寻址

- 指令中直接提供了转向的段地址和偏移地址
  - 用指令中提供的转向段地址和偏移地址取代CS和IP
  - 新指令物理地址 = $CS_{新} \times 16d + IP_{新}$
- 段间直接寻址转移指令

```
jmp    far ptr seg2
```

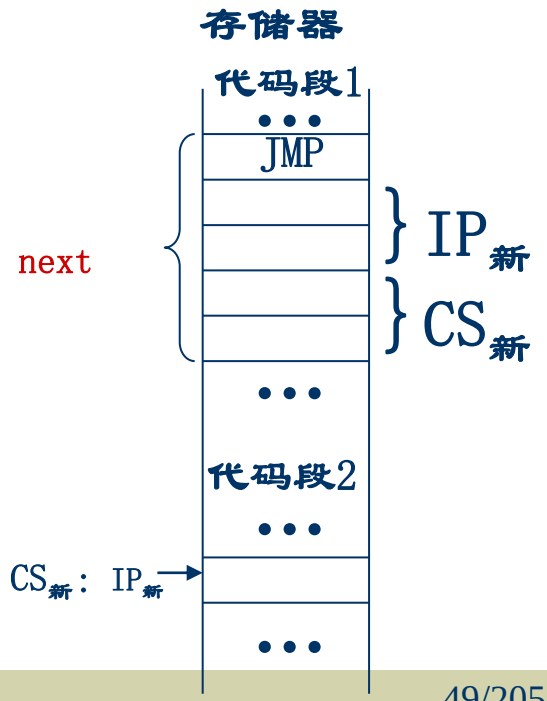
  - 其中seg2为转向的目标指令的符号地址
  - far ptr为段间转移的伪操作符
  - 汇编语言程序编写时用符号地址（标号）
  - 如果不用符号地址时，指令中直接给出**段基地址：偏移地址**

汇编程序会生成机器指令

JMP CS':IP'

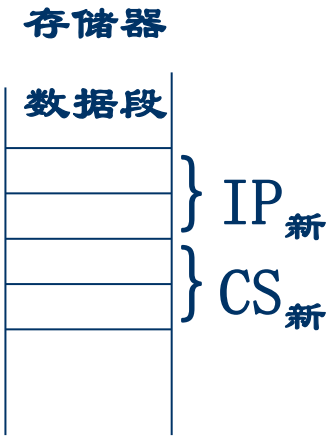
```
;  定义第一个代码段
code1    segment
begin:    ...
          jmp far ptr   next    ;转向next去执行,next为另一个代段的符号地址
          ...
code1    ends                   ;第一个代码段结束
```

```
;  定义第二个代码段
code2    segment
          ...
next:     ...                   jmp far ptr  next
          ...
code2    ends                   ;第二个代码段结束
```

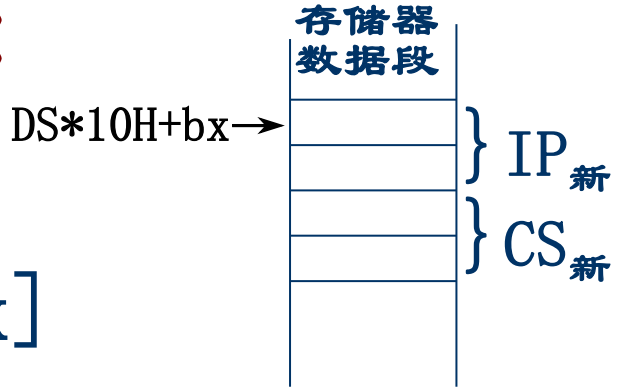


4. 段间间接寻址

- 用存储器中的两个相继字内容来取代IP和CS寄存器中的当前内容
  - 指令中给出这两个相继字的存储器寻址的有效地址
  - 存储单元中的内容可以用数据寻址方式中除立即数和寄存器以外的任何一种寻址方式取得
  - 设置新的CS、IP内容
- 新指令物理地址 = $CS_{新} \times 16 + IP_{新}$



段间间接寻址指令举例：



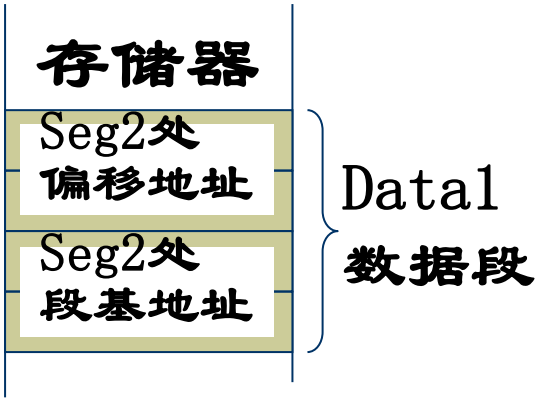

```
jmp   dword   ptr   [bx]
```

- 其中[bx]说明获取目标指令地址的寻址方式为寄存器间接寻址方式
- dword ptr为双字操作符，说明转向地址需取双字，为段间转移指令
- 执行结果：

  (DS*10H+BX+1, DS*10H+BX) → IP

  (DS*10H+BX+3, DS*10H+BX+2) → CS

```
data1   segment
addrtab dw   offset seg2
        dw   seg    seg2
data1   ends

code1   segment
begin:   ...
        mov   bx, offset  addrtab      ; 转移目标地址的偏移地址的地址送
        jmp   dword  ptr  [bx]         ; [bx]→IP, [bx+2]→CS, 间接寻址到
                                       ; 另一个代码段
          ...
code1   ends
```




```
; 转移的目标代码段
code2   segment
          ...
   seg2:  ...
          ...
code2   ends
```

注意：

- 条件转移指令只能使用段内直接寻址方式
- 无条件转移（JMP）和转子指令（CALL）可用四种方式的任何一种

课内测试 03-1-1

1．请在填空中[填空1]填写“32”；（15分）

2．假设：DS=13E4H，BX=0000H，有存储单元内容如下图。（15分）

- 指令jmp dword ptr [bx]是[填空2]（段内/段间）转移指令；
- CPU执行该指令后，CS=[填空3]H，IP=[填空4]H。

存储器	
12H	13E4:0000
34H	13E4:0001
56H	13E4:0002
78H	13E4:0003
90H	13E4:0004

3.2 80X86机器语言指令概况

Intel 80x86指令集的发展

- 如果不基于汇编语言编程，一定要仔细研读
- 如果想设计优化的汇编语言，也一定要仔细研读

3.2.1 操作码的机器语言表示

- 8086机器语言指令操作码
  - 多字节指令
  - 一条指令1-7个字节

操作码	mod-reg-r/m	位移量	立即数
1-2字节	0-1字节	0-2字节	0-2字节

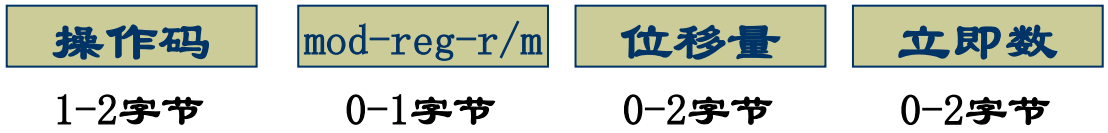
  - 操作码长度8-11位，一般情况下8位
  - 多数指令操作码格式：

    - w=1，字操作；w=0，字节操作
    - d双操作数有效；操作数最多只能有一个放在存储器中；

      d=1，寄存器用于目的操作数；d=0，寄存器用于源操作数
  - 立即数寻址方式时，操作码格式：

    - s=1，按符号扩展规则将立即数从8位扩展为16位
    - sw=00，字节操作；sw=01，有16位立即数且字操作
    - sw=10，无意义； sw=11， 8位立即数，但需字操作

3.2.2 寻址方式的机器语言表示



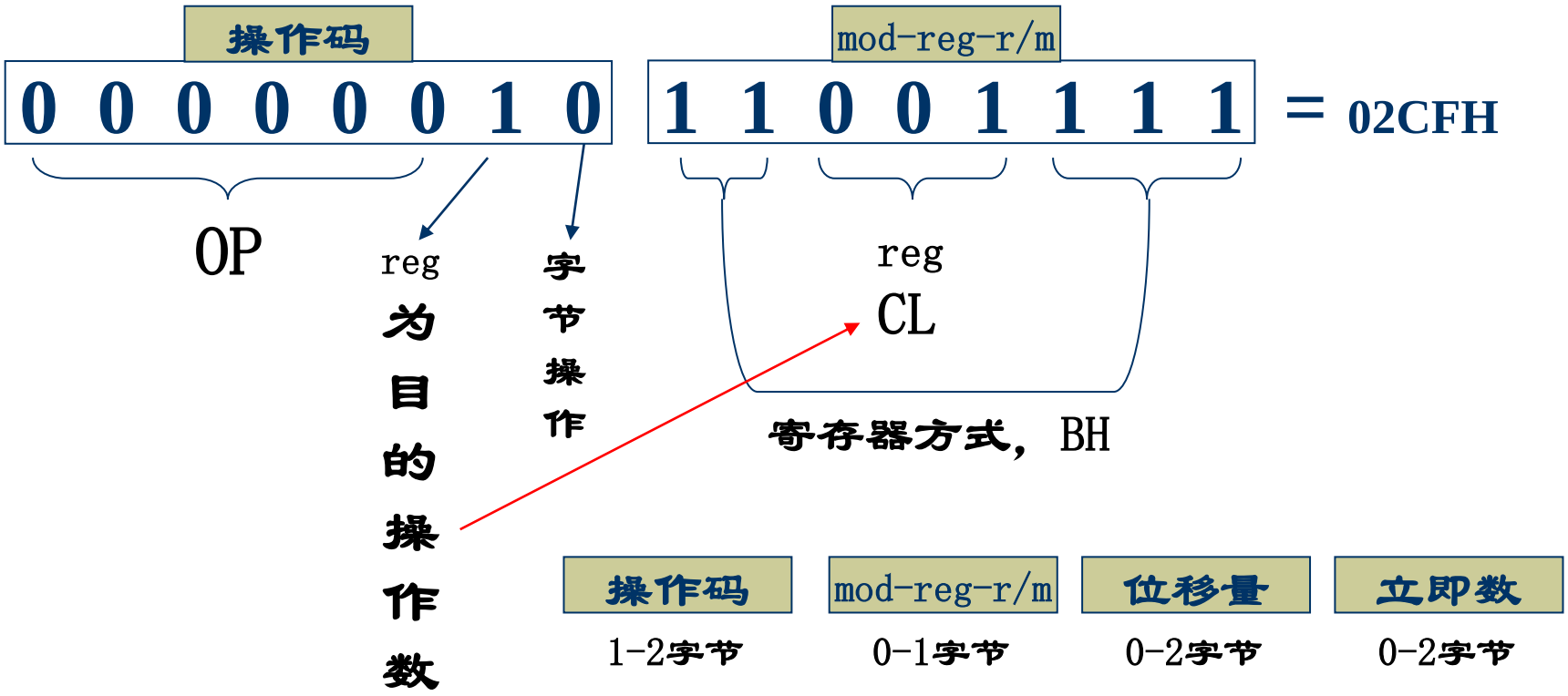
- 8086机器语言指令中寻址方式表示
  - 一般是指令的第二个字节
  - 寻址方式字节表示：

    - reg与操作码中w结合使用指定寄存器
      - P50，见3.3表

    - mod，r/m结合一起确定寻址方式
      - P51，见3.4表
  - 指定段跨越前缀表示：

    - 放在指令之前的一个字节
    - 001，110段前缀标志
    - SEG指定4个段寄存器中的一个，见表3.5

例如： ADD CL, BH



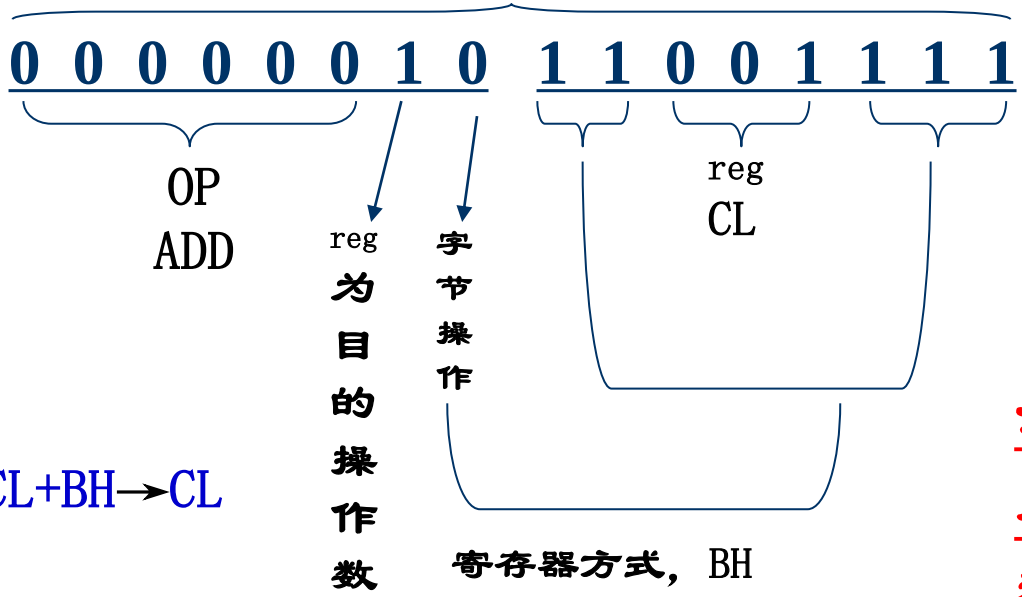
结果： CL+BH → CL

汇编指令： ADD CL, BH

由汇编程序
将汇编指令
翻译成
二进制代码的机器指令

- 如果没有汇编程序就需要人工差错、查表翻译等
- 将人工查表翻译的处理过程编成程序，这就是汇编程序

机器指令： 0 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 = 02CFH

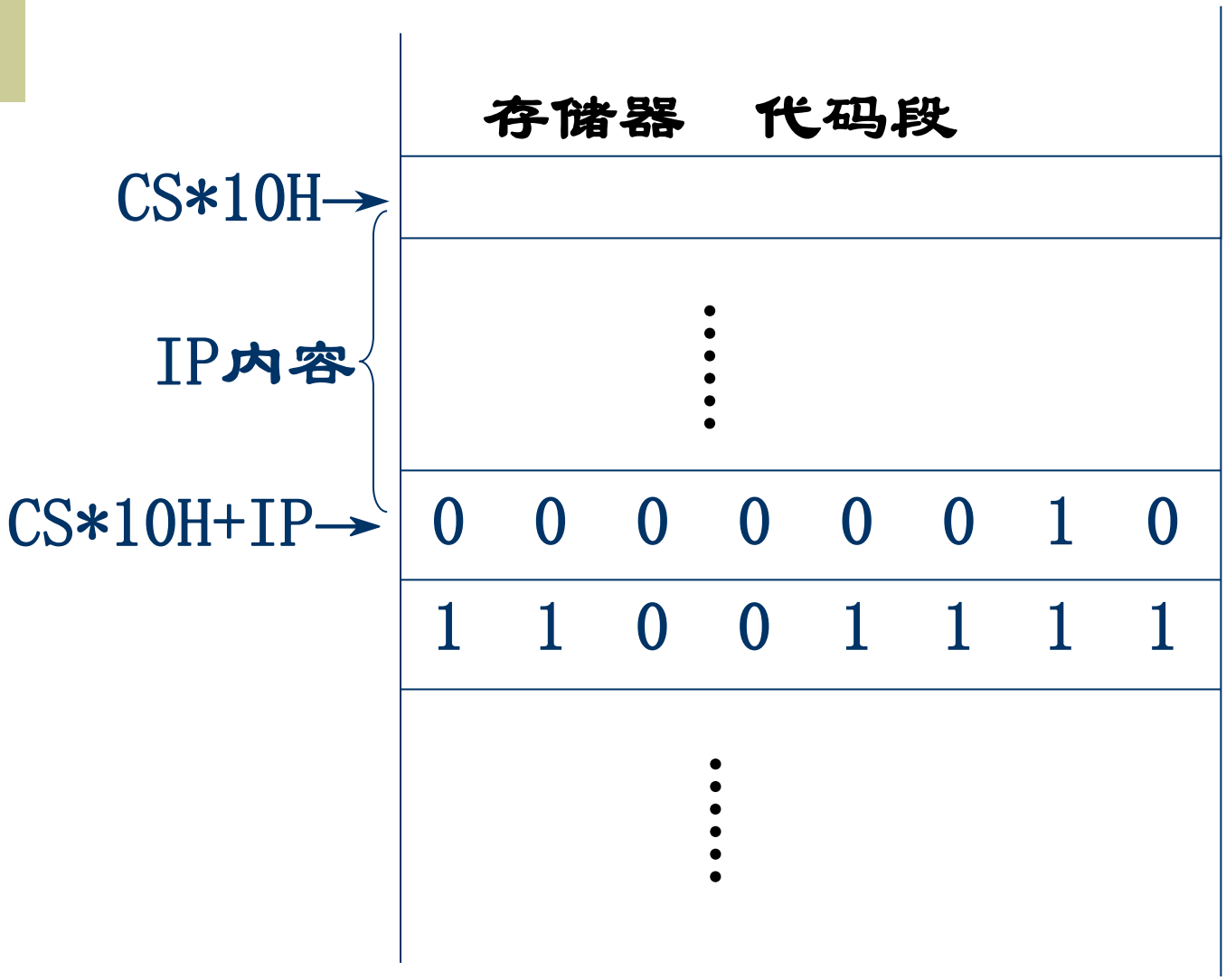


结果：CL+BH→CL

注意汇编程序和汇编源程序概念

ADD CL, BH_

0 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 = 02CFH

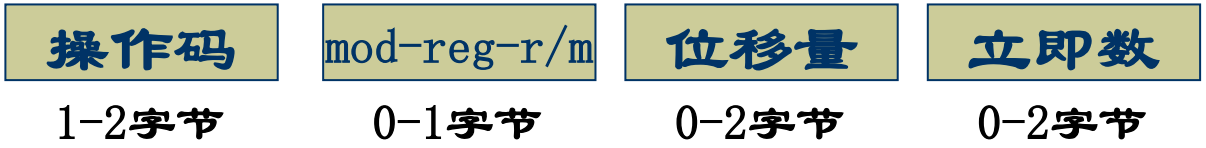


3.2.3 指令的执行时间

- 指令执行时间
  - 指令执行时间=取指令+取操作数+执行操作+传送结果
  - 每一步至少一个时钟周期
- 指令执行时间用时钟周期数表示
- 请见P56，表3.6、3.7、3.8
  - 表中为单条指令执行时间
  - 现代处理器中由于流水线、超标量等，一组指令（程序）的总执行时间会大大缩短，每条指令的平均执行时间<1个时钟周期
  - 编程时尽量使用时钟周期数少、指令字节数少的指令实现相同的功能

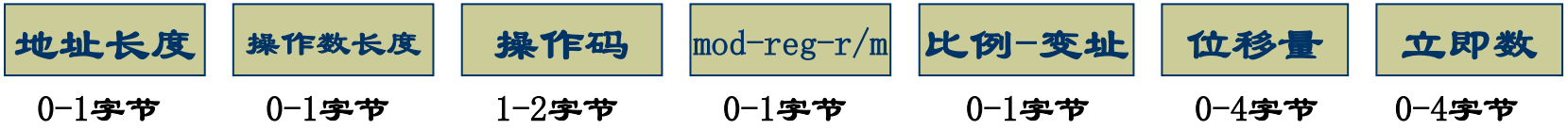
3.2.4 机器指令格式

- 8086/80286 指令格式



  - 16位地址，16位偏移量；8、16位数据

- 80386以上 指令格式



  - 32位地址/ 32位地址，16、32位偏移量；
  - 段前缀；8、16、32位数据

3.3 80X86的指令系统

3.3.1 数据传送指令

3.3.2 算术指令

3.3.3 逻辑指令

3.3.4 串处理指令

3.3.5 控制转移指令

3.3.6 处理机控制指令

学习时注意：

1. 指令的基本功能
2. 指令的执行对标志位的影响
3. 对寄存器和存储单元内容的影响
4. 对寻址方式或寄存器使用的限制和隐含使用的情况

3.3.1 数据传送指令

- 数据传送指令负责把数据、地址或立即数传送到寄存器或存储单元中
- 这类指令可实现：
  1. 寄存器之间 的数据交换
  2. 寄存器和存储器之间 的数据交换
  3. 立即数送 寄存器/存储器 单元中
  4. 把地址送 指定的寄存器
  5. 寄存器/存储器单元的内容 压入堆栈，或弹出堆栈
  6. 累加器和I/O端口之间 的数据传送
  7. 标志寄存器和AH之间 的数据交换

3.3.1 数据传送指令

这类指令分为如下五类：

1、通用数据传送指令

2、累加器专用传送指令

3、地址传送指令

4、标志寄存器传送指令

5、类型转换指令

1、通用数据传送指令

① MOV (move)

② MOVSX (move with sign-extend) (386及后继机型)

③ MOVZX (move with zero-extend) (386及后继机型)

④ PUSH (push onto the stack)

⑤ POP (pop from the stack)

⑥ PUSHA/PUSHAD (push all(16/32) registers)

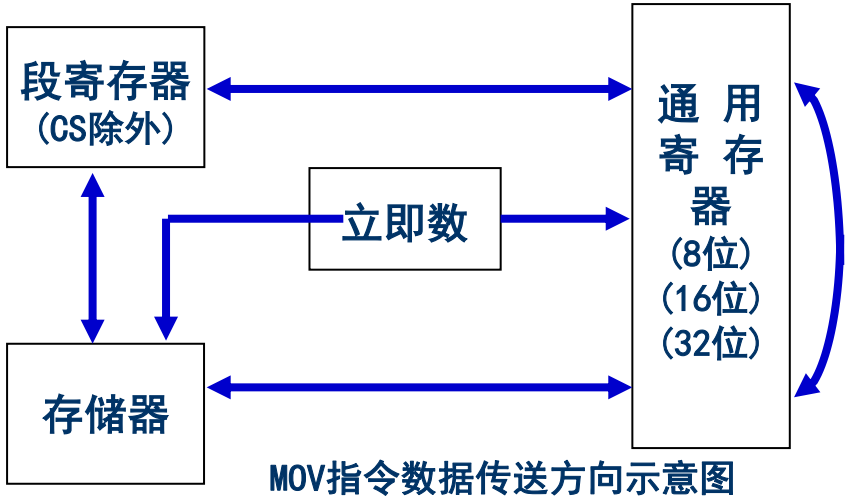
⑦ POPA/POPAD (pop all(16/32) registers)

⑧ XCHG (exchange)

① MOV指令

传送指令： MOV DST, SRC

执行操作： (DST) ← (SRC)



MOV指令数据传送方向示意图

注意:

- 立即数不能作为目标操作数
- 立即数不能直接送段寄存器 × MOV DS, 2000H
- DST不能是CS，因为随意修改CS会引起不可预料的结果
- DST、SRC不同时为段寄存器 × MOV DS, ES
- 两个存储单元之间不能直接传送
- 不影响标志位

例： MOV AX, DATA_SEG
MOV DS, AX

段名
这里按段基址处理

例： MOV AL, ‘E’ ; MOV AL, 45H

例： MOV BX, OFFSET TABLE

例： MOV AX, Y[BP][SI]

② MOVSX带符号扩展传送指令

- **386及后继机型可用**
- **格式：MOVSX DST,SRC**
- **功能：(DST) ← 符号扩展(SRC),DST空出的位用SRC的符号位填充**
  - **该指令的源操作数SRC可以是8位(16位)的寄存器或存储单元的内容,不能是立即数**
  - **目的操作数DST则必须是16位(32位)的寄存器,传送时把源操作数符号扩展送入目的寄存器**
  - **MOVSX不影响标志位**

```
MOV     DL,80H
MOVSX   AX,DL
```

AX中得到80H的带符号扩展值FF80H

```
MOV     VAR,56H
MOVSX   AX,VAR
```

AX中得到56H的带符号扩展值0056H

③ MOVZX 带零扩展传送指令

- **386及后继机型可用**
- 格式:MOVZX DST,SRC
- 功能:(DST)←零扩展(SRC),DST空出的位用0填充
  - 该指令的源操作数和目的操作数以及对标志位的影响均与MOVSX指令相同

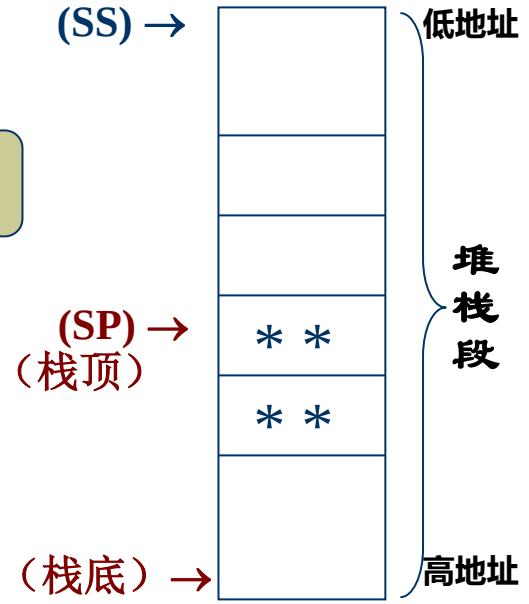
- 区别: MOVSX的源操作数为**带符号数**扩展

  MOVZX的源操作数为**无符号数**扩展

```
MOV    DL,80H
MOVZX  AX,DL          ;AX= 0080H
MOVZX  EAX,DL         ;EAX= 00000080H

MOV    CX,1234H
MOVZX  EAX,CX         ;EAX= 00001234H

MOV    VAR,56H
MOVZX  AX,VAR         ;AX=0056H
```

- 使用MOVSX指令可以方便地实现对带符号数的扩展
- 使用MOVZX指令可以方便地实现对无符号数的扩展

④ PUSH指令
⑤ POP指令



目标地址默认为堆栈栈顶

进栈指令： PUSH SRC

执行操作： (SP) ← (SP) – 2

((SP)+1, (SP)) ← (SRC)

源操作数地址默认为堆栈栈顶

出栈指令： POP DST

执行操作： (DST) ← ((SP)+1, (SP))

(SP) ← (SP) + 2

8086 PUSH/POP指令格式

PUSH/POP REG

PUSH/POP MEM

PUSH/POP SEGREG

不允许立即数操作

80286以上PUSH/POP指令格式

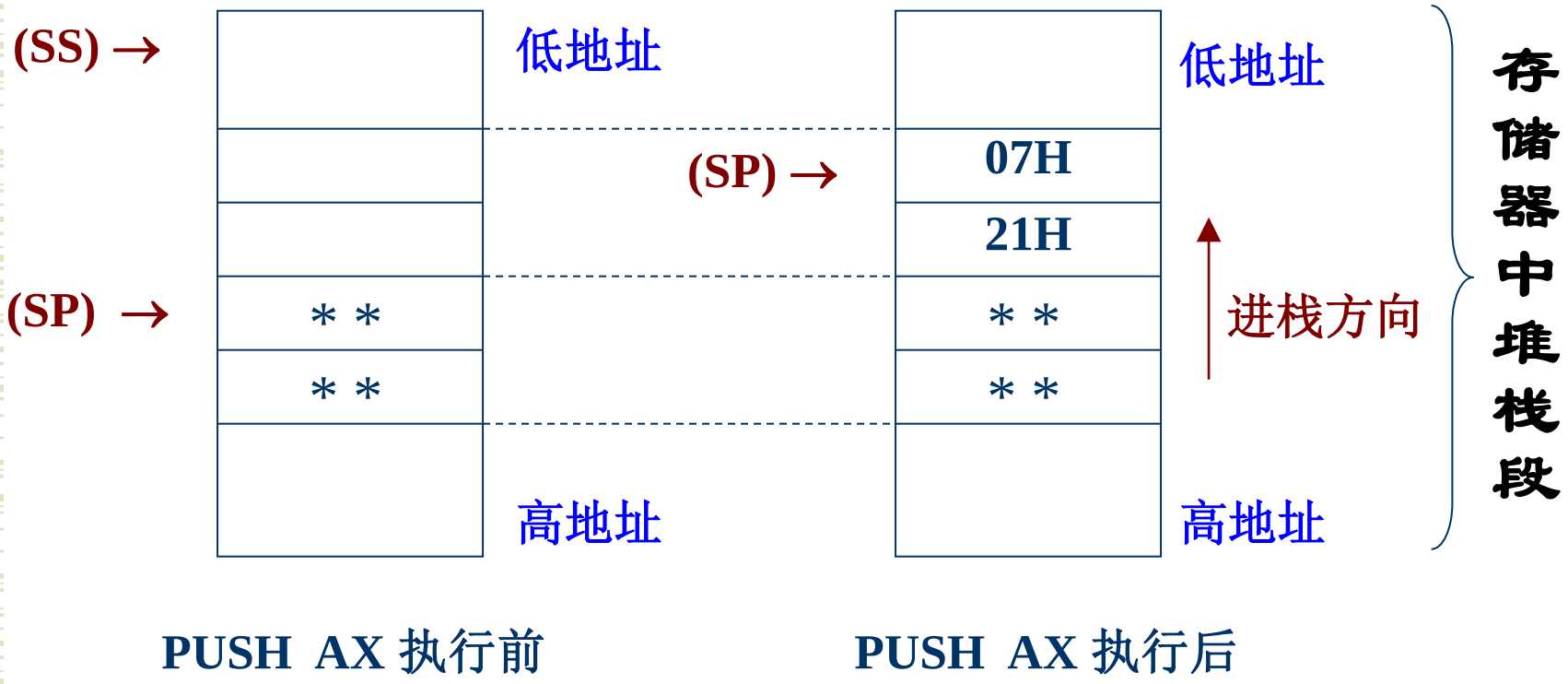
PUSH/POP REG

PUSH/POP MEM

PUSH/POP DATA

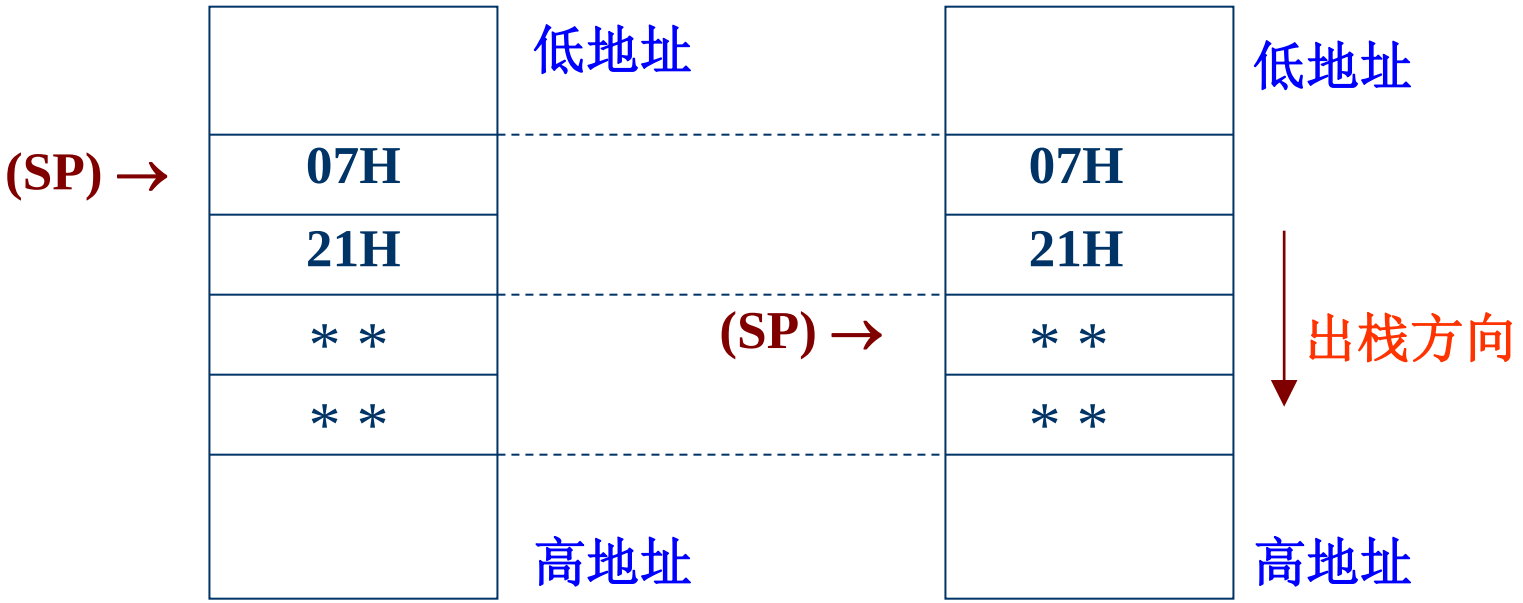
PUSH/POP SEGREG

例：　假设 AX = 2107 H，　执行　PUSH　AX



堆栈指针SP内容自动修改，并且是地址减量

例： POP BX



POP BX 执行前

POP BX 执行后

BX = 2107H

堆栈指针SP内容自动修改，并且是地址增量

课内测试 03-1-2

1．请在填空中[填空1] 填写“16”；　　（10分）

2．假设：SS=13E4H，SP=1234H。

CPU执行PUSH AX指令后，SP=[填空2]H；

CPU执行POP AX指令后，SP=[填空3]H。（10分）

- 堆栈：“后进先出”的存储区，存在于堆栈段中，SP 在任何时候都指向栈顶
- 若SRC是16位操作数，则堆栈指针寄存器减2；若SRC是32位操作数，则堆栈指针寄存器减4
- 若DST是16位，则堆栈指针寄存器加2；若DST是32位，则堆栈指针寄存器加4

注意：

* 堆栈操作必须以字为单位
* 不影响标志位
* 8086不能用立即寻址方式 × PUSH 1234H
* DST不能是CS × POP CS

  CS不能人为改动，需要时可以借助RET指令让CPU自己改动
* PUSH、POP指令一般配对使用

例：　PUSH AX

PUSH BX

…

…

POP BX

POP AX

- 堆栈主要用于对现场数据的保护与恢复、子程序与中断服务返回地址的保护与恢复、参数传递等

⑥ PUSHA/PUSHAD所有(16/32位 P62)寄存器进栈指令

PUSHA:8个16位通用寄存器依次进栈 (AX、BX、CX、DX,指令执行前的SP 、 BP 、 SI 、 DI)

PUSHAD:8个32位通用寄存器依次进栈

⑦ POPA/POPAD 所有 (16/32位 P63) 寄存器出栈指令

- PUSHA、POPA 80286及以上机器
- PUSHAD、POPAD 80386及以上机器

⑧ XCHG交换指令

格式：XCHG OPR1,OPR2

执行的操作（ OPR1 ）↔（ OPR2 ）

- 操作数可以是8位、16位、32位
- 该指令可能的组合是：

XCHG	寄存器操作数,寄存器操作数	XCHG AL, BH
XCHG	寄存器操作数,存储器操作数	XCHG BX, [BP+SI]
XCHG	存储器操作数,寄存器操作数	XCHG [BP+SI], BX

注意：

* 不影响标志位
* 不允许使用段寄存器

3.3.1.2 累加器专用传送指令

1. IN (input) I/O输入
2. OUT (output) I/O输出

只限于使用EAX、AX或AL

3. XLAT (translate)

只限于使用AL和BX

1、 I/O输入指令

- 格式： IN ACR, PORT
- 功能：把外设端口（PORT）的内容传送给累加器（ACR）
  - 可以传送8位、16位、32位,相应的累加器选择AL、AX、EAX
- 端口号在0～255之间，则端口号直接写在指令中

长格式：	IN AL, PORT	(字节)
	IN AX, PORT	(字)
执行操作：	(AL) ← (PORT)	(字节)
	(AX) ← (PORT+1,PORT)	(字)

- 端口号大于255，则端口号通过DX寄存器间接寻址，即端口号应先放入DX中

短格式：	IN AL, DX	(字节)
	IN AX, DX	(字)
执行操作：	(AL) ← (DX)	(字节)
	(AX) ← (DX+1, DX)	(字)

2. I/O输出指令 OUT

- **格式：** OUT PORT,ACR
- **功能：** 把累加器的内容传送给外设端口,对累加器和端口号的选择限制同IN指令

长格式：	OUT PORT, AL	(字节)
	OUT PORT, AX	(字)
执行操作：	(PORT) ← (AL	(字节)
	(PORT+1, PORT) ← (AX)	(字)
短格式：	OUT DX, AL	(字节)
	OUT DX, AX	(字)
执行操作：	((DX)) ← (AL)	(字节)
	((DX+1, DX)) ← AX	(字)

注意：

* 不影响标志位
* 前256个端口号00H-FFH可直接在指令中指定(长格式)
* 如果端口号≥ 256，端口号 → DX（短格式）

```
例：    IN    AX, 28H              ;    MOV   DX, 28H
                                   ;    IN    AX, DX
        MOV   DATA_WORD, AX
```

```
例：    MOV   DX, 3FCH          例：OUT   5, AL
        IN    AX, DX
```

例：测试某状态寄存器（端口号27H）的第2位是否为1

```
        IN      AL, 27H
        TEST    AL, 00000100B
        JNZ     ERROR              ;若第2位为1，转到ERROR处理
```

3. 换码指令（查表转换指令）

- 换码指令：XLAT　或　XLAT　OPR
- 执行操作(AL)←((BX)+(AL))
  - OPR一般是表格的起始符号基址，只是为了程序可读性，执行时只能使用BX、AL
  - 这条指令前，一定要使BX指向表格的起始符号基址
  - 只限于使用AL和BX
- 换码指令常用于把一种代码转换为另一种代码

字符	ASCII码
0～9	30H~39H
A～Z	41H~5AH
a～z	61H~7AH

注意：

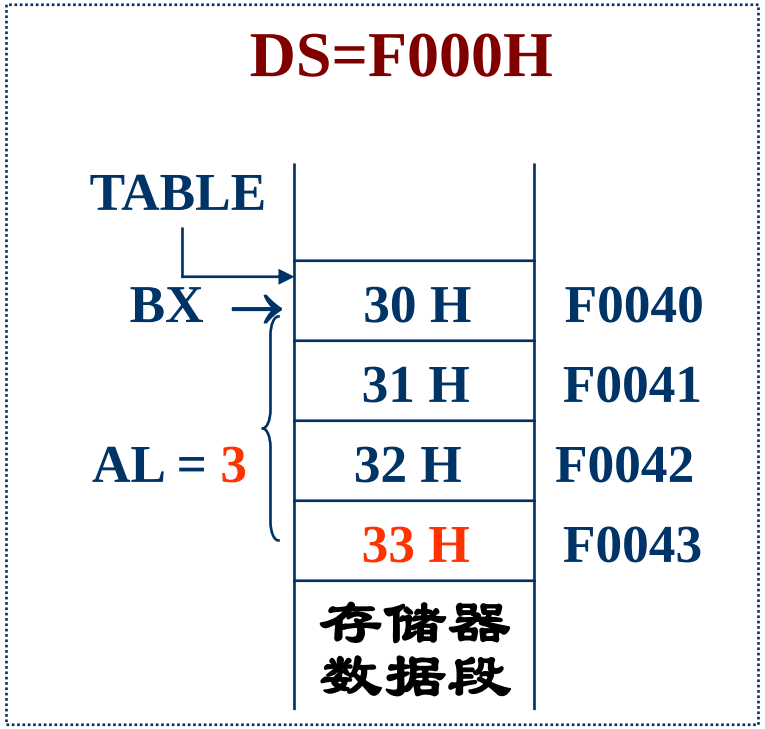
* 不影响标志位
* 字节表格(长度不超过256)首地址的偏移地址 → BX
* 需转换代码 → AL

例：十进制3转换成它的ASCII码

假设DS=F000H

```
MOV  BX, OFFSET TABLE      ; BX=0040H
MOV  AL, 3
XLAT  TABLE                ; (AL) ← ((BX) + (AL))
```

- DS*16 + BX + AL=F0043H
- 指令执行后 **AL=33H**



课内测试 04-1-1

1．请在填空中[填空1] 填写“13”。（10分）

2．I/O输入输出指令中 （10分）

- 端口号小于等于 [填空2]，则端口号可以直接写在指令中
- 当端口号大于 [填空3]，则端口号必须通过DX寄存器间接寻址，即端口号应先放入DX中

3.3.1.3 地址传送指令

1. LEA(load effective address) 有效地址送寄存器
2. LDS(load DS with pointer) 指针送寄存器和DS
3. LES(load ES with pointer) 指针送寄存器和ES
4. LSS(load SS with pointer) 指针送寄存器和SS
5. LFS(load FS with pointer) 指针送寄存器和FS
6. LGS(load GS with pointer) 指针送寄存器和GS

- 这组指令完成把操作数地址送到指定的寄存器
- LEA是把源操作数的有效地址EA送到指定的寄存器中
- 其余5条指令源操作数只能用存储器寻址方式，对于16位指针寄存器而言，低地址中的16位数据装入指针寄存器，高地址中的16位数据装入段寄存器

- 地址传送指令

有效地址送寄存器指令： LEA REG, SRC
执行操作： (REG) ← SRC

把源操作数有效地址送到指定的寄存器中

指针送寄存器和DS指令： LDS REG, SRC
执行操作： (REG) ← (SRC)
(DS) ← (SRC+2)

相继二字 → 寄存器、DS

指针送寄存器和ES指令： LES REG, SRC
执行操作： (REG) ← (SRC)
(ES) ← (SRC+2)

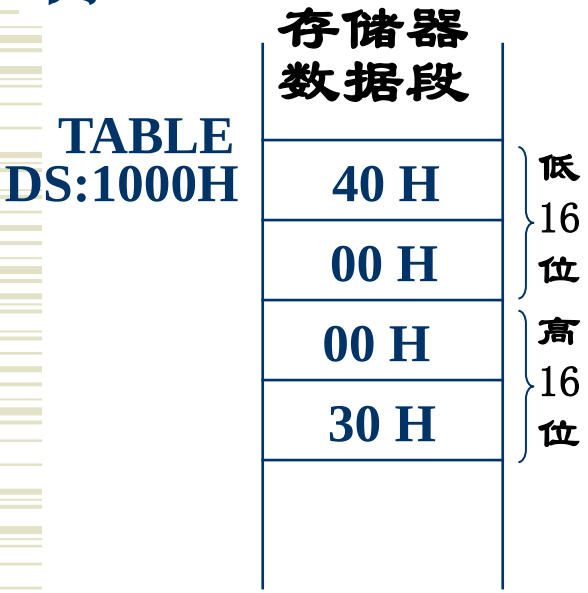
相继二字 → 寄存器、ES

例1：LEA BX, [BX+SI+0F62H] ；操作数的有效地址BX+SI+0F62H送BX

例2：LDS SI, [10H] ;操作数低16位送SI，高16位送DS

例3：LES DI, [BX] ;操作数低16位送DI，高16位送ES

例4：




```
MOV  BX, TABLE              ; BX=0040H

MOV  BX, OFFSET TABLE       ; BX=1000H

LEA  BX, TABLE              ; BX=1000H

LDS  BX, TABLE              ; BX=0040H
                            ; DS=3000H

LES  BX, TABLE              ; BX=0040H
                            ; ES=3000H

LSS  BX, TABLE              ; BX=0040H
                            ; SS=3000H
```

注意:

* 不影响标志位
* REG不能是段寄存器
* SRC必须为存储器寻址方式

MOV指令就可以实现这些功能

为什么使用地址传送指令

- **简单**：多个数据块操作时，指针改变非常方便
- **安全**：可以在一个不被中断的指令操作中同时改变标识地址的两个寄存器，确保在一条指令中使段寄存器和指针寄存器都被重置

  例如 SS、SP改变时特别重要

3.3.1.4 标志寄存器传送指令

LAHF (load AH with flags)

标志送AH

SAHF (store AH into flags)

AH送标志寄存器

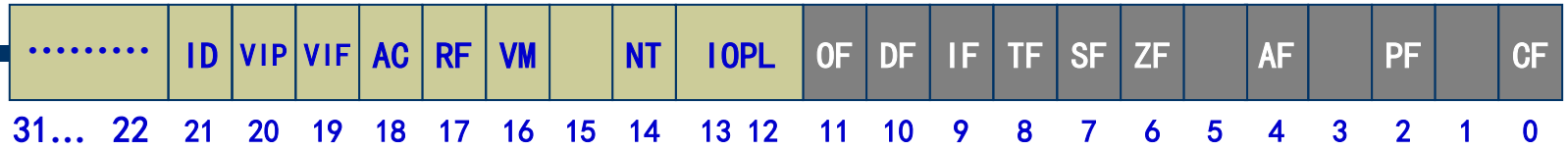
PUSHF/PUSHFD(push the flags or eflags)

标志进栈

POPF/POPFD (pop the flags or eflags)

标志出栈

其中指令中的POPF、POPFD、SAHF指令影响标志位，其他不影响



标志送AH指令： LAHF
执行操作： (AH) ← (PSW的低字节)

AH送标志寄存器指令： SAHF
执行操作： (PSW的低字节) ← (AH)

标志进栈指令： PUSHF
执行操作： (SP)←(SP) - 2
((SP+1, SP)) ← (PSW)

标志出栈指令： POPF
执行操作： (PSW) ← ((SP+1, SP))
(SP) ← (SP) + 2

PUSHFD、POPFD对应32位EFLAGS

3.3.1.5 类型转换指令

1. CBW(convert byte to word)

字节转换为字，AL中的内容符号扩展到AH

2. CWD/CWDE

字转换为双字

- CWD：AX的内容符号扩展到DX,形成DX:AX中的双字
- CWDE：AX的内容符号扩展到EAX,形成EAX中的双字

3. CDQ

- 双字转换为4字指令
- EAX的内容符号扩展到EDX,形成EDX:EAX中的4字

4. BSWAP(486及后继机型使用)

- 字节交换指令,字节次序变反
- 例如 EAX=11 22 33 44H
  - 执行 BSWAP EAX 后 EAX=44 33 22 11H

这些指令只对累加器内容进行带符号扩展和处理

3.3.2 算术指令

这类指令包括二进制运算指令和十进制运算指令：

1. **加法指令**（加法指令ADD、带进位加法指令ADC、加1指令INC等）
2. **减法指令**（减法指令SUB、带借位减法指令SBB、减1指令DEC、求补指令NEG、比较指令CMP等）
3. **乘法指令**（无符号乘法指令MUL、带符号数乘法指令IMUL等）
4. **除法指令**（无符号除法指令DIV、带符号除法指令IDIV）
5. **十进制调整指令**（DAA、DAS等）

注意：

- 目标操作数不允许是立即数和CS段寄存器
- 两个操作数不能同时为存储器操作数
- 单操作数指令不允许使用立即数方式
- 除十进制调整指令外，其他指令均影响某些标志位

3.3.2.1 加法指令

① ADD (add) 加法指令

② ADC (add with carry) 带进位加法指令

③ INC (increment) 加1指令

④ XADD(exchange and add) 交换并相加指令

- 该指令只能用于486及后继机型,它把目的操作数装入源,并把源和目的操作数之和送目的地址

看 p70~71加法运算后对OF和CF的影响情况

(1)加法指令 ADD

格式：ADD　DST,SRC

功能：（DST）＋（SRC）➞（DST）

说明：对操作数的限定同MOV指令

(2)带进位加法指令 ADC

格式：ADC　DST,SRC

功能:（DST）＋（SRC）＋CF➞(DST)

说明:对操作数的限定同MOV指令,该指令适用于多字节或多字的加法运算

(3) 加1指令 INC

格式：INC OPR

功能：（OPR）＋1 → (OPR)

说明：很方便地实现地址指针或循环次数的加1修改

(4) 互换并加法指令 XADD(486以上)

格式：XADD DST, SRC

功能：(SRC) + (DST) → 暂存器

(DST) → (SRC)

暂存器→(DST)

说明：该指令执行后,原DST的内容在SRC中,和在DST中

加法指令对条件标志位的影响

* 条件标志位最主要的4位：CF, ZF, SF, OF
* 除INC指令不影响CF标志外，均对条件标志位有影响

SF= { 1 结果为负；0 否则 }　　ZF= { 1 结果为0；0 否则 }

CF= { 1 和的最高有效位 有 向高位的进位；0 否则 }

OF= { 1 两个操作数符号相同，而结果符号与之相反；0 否则 }

CF 位表示 无符号数 相加的溢出

OF 位表示 带符号数 相加的溢出

举例：n=8 bit 带符号数(-128~127)，无符号数(0~255)

```
  0 0 0 0   0 1 0 0
+ 0 0 0 0   1 0 1 1
-------------------
  0 0 0 0   1 1 1 1
```

带：(+4)+(+11)=+15　OF=0

无：4+11=15　CF=0

带符号数和无符号数都不溢出

```
  1 0 0 0   0 1 1 1
+ 1 1 1 1   0 1 0 1
-------------------
1 0 1 1 1   1 1 0 0
```

带：(-121)+(-11)=+124　OF=1

无：135+245=124　CF=1

带符号数和无符号数都溢出

```
  0 0 0 0   0 1 1 1
+ 1 1 1 1   1 0 1 1
-------------------
1 0 0 0 0   0 0 1 0
```

带：(+7)+(-5)=+2　OF=0

无：7+251=2　CF=1

无符号数溢出

```
  0 0 0 0   1 0 0 1
+ 0 1 1 1   1 1 0 0
-------------------
  1 0 0 0   0 1 0 1
```

带：(+9)+(+124)=-123　OF=1

无：9+124=133　CF=0

带符号数溢出

例：双精度数的加法

(DX) = 0002H　(AX) = 0F365H
(BX) = 0005H　(CX) = 8100H

指令序列　ADD AX, CX　　；(1)
　　ADC DX, BX　　；(2)

(1) 执行后，(AX) = 7465H
　　CF=1　OF=1　SF=0　ZF=0

(2) 执行后，(DX) = 0008H
　　CF=0　OF=0　SF=0　ZF=0

3.3.2.1 减法指令

① SUB 减法
② SBB 带借位减法
③ DEC 减1
④ NEG 求补
⑤ CMP 比较

⑥ *CMPXCHG* 比较并交换（486以后）
⑦ *CMPXCHG8B* 比较并交换8字节（Pentium 以后）

(1) 减法指令 SUB

格式：SUB　DST,SRC

功能：（DST）－（SRC）➞(DST)

说明：除是实现减法功能外，其他要求同ADD

(2) 带借位减法指令 SBB

格式:SBB　DST,SRC

功能:(DST)－(SRC)－CF➞(DST)

说明:除了操作为减外,其他要求同ADC,该指令适用于多字节或多字的减法运算

(3) 减1指令 DEC

格式：DEC　OPR

功能：(OPR)－1➞(OPR)

说明：可以很方便地实现地址指针或循环次数的减1修改

(4) 求补指令 NEG

格式：NEG　OPR

功能：0FFFFH －(OPR)+1 → (OPR)

对目标操作数（含符号位）求反加1，并且把结果送回目标

说明：利用NEG指令可实现求一个数的补码

(5) 比较指令 CMP

格式:CMP　OPR1,OPR2

功能:(OPR1)－(OPR2)，只影响标志位，不影响源和目的操作数

说明:这条指令执行相减操作后只根据结果设置标志位，并不改变两个操作数的原值，其他要求同SUB。CMP指令常用于比较两个数的大小。

减法指令对条件标志位（CF/OF/ZF/SF）的影响

* 除DEC指令不影响CF标志外，均对条件标志位有影响。

$$CF=\begin{cases}1 & \text{被减数的最高有效位 有 向高位的借位}\\ 0 & \text{否则}\end{cases}$$

或

$$CF=\begin{cases}1 & \text{减法转换为加法运算时 无 进位}\\ 0 & \text{否则}\end{cases}$$

$$OF=\begin{cases}1 & \text{两个操作数符号相反，而结果的符号与减数相同}\\ 0 & \text{否则}\end{cases}$$

CF 位表示 无符号数 减法的溢出。

OF 位表示 带符号数 减法的溢出。

例 3.54, 3.55 (P73)

NEG 指令对 CF/OF 的影响

$$CF = \begin{cases} 0 & \text{操作数为0} \\ 1 & \text{否则} \end{cases}$$

$$OF = \begin{cases} 1 & \text{操作数为 -128 （字节运算）或} \\ & \text{操作数为 -32768 （字运算）} \\ 0 & \text{否则} \end{cases}$$

(6)比较并交换指令 CMPXCHG

80486及其后续机型

格式:CMPXCHG DST,SRC

SRC：AL、AH、AX寄存器； DST：寄存器或存储器寻址方式

功能:(DST)－(SRC)，影响标志位；如果相等，(SRC)→(DST)；不相等，(DST)→(SRC)

说明:这条指令执行相减操作后根据结果设置标志位

(7)比较并交换８字节指令 CMPXCHG8B

Pentium及其后续机型

格式:CMPXCHG8B DST,SRC

SRC：EDX:EAX构成的64位； DST：存储器寻址方式确定的64位

功能:(DST)－(SRC)，影响标志位；如果相等，(SRC)→(DST)；不相等，(DST)→(SRC)

例：x、y、z 均为双精度数，分别存放在地址为X, X+2；Y, Y+2；Z, Z+2的存储单元中，用指令序列实现 w ← x + y + 24 - z ，并用W, W+2单元存放w (P74)

```
MOV   AX,   X
MOV   DX,   X+2
ADD   AX,   Y
ADC   DX,   Y+2        ;   x+y
ADD   AX,   24
ADC   DX,   0          ;   x+y+24
SUB   AX,   Z
SBB   DX,   Z+2        ;   x+y+24-z
MOV   W,    AX
MOV   W+2,  DX         ;   结果存入W，W+2单元
```

3.3.2.3 乘法指令

① MUL 无符号数乘法

② IMUL 带符号数乘法

- 对除CF和OF以外的条件码无定义，即不确定
- 在乘法指令里，目的操作数必须是累加器（字运算为AX，字节运算为AL）
- 两个 8 位数相乘得到的是16位乘积，两个16位数相乘得到的是32位乘积，存放在DX:AX中

(1) 无符号数乘法 MUL

格式: MUL SRC　　; SRC：除立即数以外的寻址方式

功能:　字节操作：　(AL)*(SRC) → (AX)

　　字操作：　(AX)*(SRC) → (DX:AX)

　　双字操作：　(EAX)*(SRC) → (EDX:EAX)

- 乘积的　一半为0，则CF、OF均为0，否则CF、OF均为1

  这样可以检查结果是字节、字或双字

(2) 带符号数乘法 IMUL

格式: IMUL SRC　　; SRC：除立即数以外的寻址方式

功能:　字节操作：　(AL)*(SRC) → (AX)

　　字操作：　(AX)*(SRC) → (DX:AX)

　　双字操作：　(EAX)*(SRC) → (EDX:EAX)

- 乘积的　一半是低一半的符号扩展，则CF、OF均为0，否则CF、OF均为1。 其实质和MUL情况下一样，主要用于判断结果是字节、字或双字

乘法指令对 CF/OF 的影响

MUL指令: CF,OF = { 00 乘积的高一半为零
11 否则

IMUL指令: CF,OF = { 00 乘积的高一半是低一半的符号扩展
11 否则

(3) 多操作数带符号数乘法 IMUL

80286及其后续机型，增加了双操作数和三操作数指令

- 格式：IMUL REG, SRC

  SRC:任一种寻址方式，立即数可以8位（自动符号扩展）、16位

  功能：字操作：(REG16)*(SRC) → (REG16)

  双字操作：(REG32)*(SRC) → (REG32)

- 格式：IMUL REG, SRC, IMM

  SRC：除立即数外的寻址方式

  IMM：立即数，8位（自动符号扩展）、16位、32位，与目的操作数长度一致

  功能：字操作：IMM*(SRC) → (REG16)

  双字操作：IMM*(SRC) → (REG32)

课内测试 04-2-1

1．请在填空中[填空1] 填写“36”；（15分）

2．无符号数乘法指令 MUL SRC，字操作时：（15分）

- 16位被乘数在[填空2]寄存器中
- 32位乘积的高16位在[填空3]，低16位在[填空4]

3.3.2.4 除法指令

① DIV 无符号数除法

② IDIV 带符号数除法

- 对所有条件码无定义，即不确定
- 源操作数可以用除立即数以外的任一种寻址方式

(1)无符号除法指令 DIV

格式：DIV SRC （DIV reg DIV mem）

- 源操作数不能是立即数；被除数必 事前放在隐含的寄存器中；可以实现8位、16位、32位无符号数除
- **具体操作为：**

字节型除法：(AX) ÷ (SRC)$_8$ → { 商:(AL) 余数:(AH)

字型除法：(DX:AX) ÷ (SRC)$_{16}$ → { 商:(AX) 余数:(DX)

双字型除法:(EDX:EAX) ÷ (SRC)$_{32}$ → { 商:(EAX) 余数:(EDX)

(2) 带符号除法指令IDIV

格式: IDIV SRC （IDIV reg IDIV mem）

- 实现两个带符号数相除，商和余数均也为带符号数，余数符号与被除数相同
- 源操作数不能是立即数；被除数必 事前放在隐含的寄存器中；可以实现8位、16位、32位无符号数除
- 具体操作为：

字节型除法：(AX) ÷ (SRC)$_8$ → { 商: (AL)；余数: (AH)

字型除法：(DX:AX) ÷ (SRC)$_{16}$ → { 商: (AX)；余数：(DX)

双字型除法: (EDX:EAX) ÷ (SRC)$_{32}$ → { 商: (EAX)；余数: (EDX)

注意：

（1）除法指令： **除数、商和余数的位数是被除数的一半**

- 字节操作时，要求被除数为16位，商为8位
- 字操作时，要求被除数为32位，商为16位
- 双字操作时，要求被除数为64位，商为32位

（2）若除法结果产生溢出（如：字节操作被除数的 8位绝对值≥除数的绝对值，这时商会超过8位，则商就会产生溢出），80X86是由系统直接转入0号中断处理

例：x，y，z，v均为16位带符号数，计算 (v-(x*y+z-540))/x (P78)

```
data    segment
    X   dw ?
    Y   dw ?
    Z   dw ?
    V   dw ?
data    ends
```

```
MOV     AX, X
IMUL    Y               ;   x*y →（DX,AX）
MOV     CX, AX
MOV     BX, DX          ;   结果暂存（BX,CX）
MOV     AX, Z
CWD                     ;   Z →（DX，AX）
ADD     CX, AX
ADC     BX, DX          ;   x*y+z →（BX，CX）
SUB     CX, 540
SBB     BX, 0           ;   x*y+z-540
MOV     AX, V
CWD                     ;   V →（DX，AX）
SUB     AX, CX
SBB     DX, BX          ;   v-(x*y+z-540)
IDIV    X               ;   (v-(x*y+z-540))/x→（AX）
                                余数→（DX）
```

3.3.2.5 十进制调整指令

为便于十进制计算，80X86还提供了一组十进制调整指令，这组指令在二进制计算的基础上，给予十进制调整，可以直接得到十进制的结果

(1) 压缩的BCD码调整指令

DAA DAS

(2) 非压缩的 BCD码调整指令

AAA AAS AAM AAD

- AAA(ASCII adjust after addition)

- BCD码

BCD码：用二进制编码的十进制数，又称二--十进制数

压缩的BCD码：用 4 位二进制数表示 1 位十进制数

例：$(59)_{10}=(0101\ 1001)_{BCD}$

非压缩的BCD码：用 8 位二进制数表示 1 位十进制数

例：$(59)_{10}=(uuuu\ 0101\quad uuuu\ 1001)_{BCD}$

数字的 ASCII 码是一种 非压缩的 BCD 码

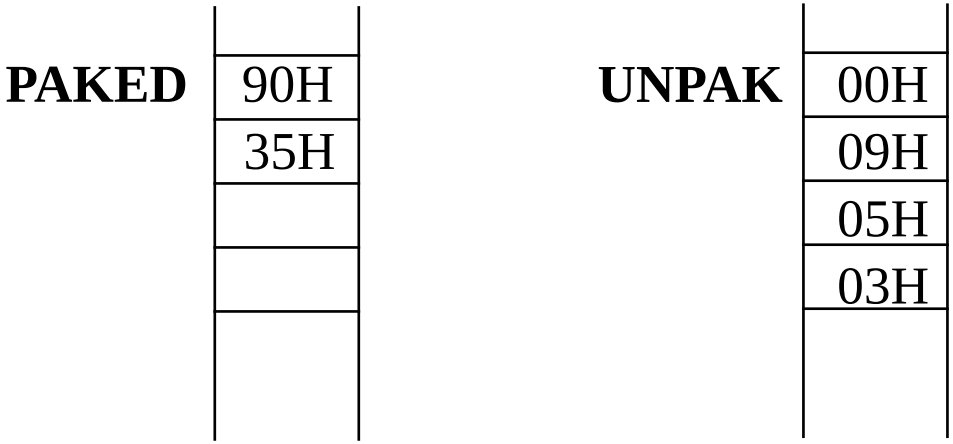
DIGIT	ASCII	BCD
0	30H	0011 0000
1	31H	0011 0001
2	32H	0011 0010
…	…	…
9	39H	0011 1001

例：写出$(3590)_{10}$的压缩 BCD 码和非压缩BCD码，并分别 把它们存入数据区 PAKED 和 UNPAK

压缩BCD：　　$(3590)_{10}=(0011\ 0101\ 1001\ 0000)_{BCD}$

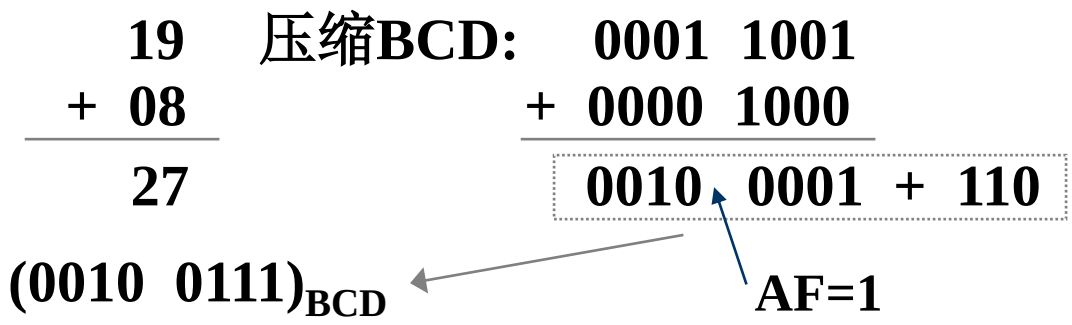
非压缩BCD：

$(3590)_{10}=(00000011\ 00000101\ 00001001\ 00000000)_{BCD}$



十进制调整指令

问题的提出：



P79

注意：

- 由于仅仅是调整而不是真正意义上的十进制运算，所以这组指令都需要与相应的二进制运算指令配合才可得到正确的十进制结果
- 使用这组指令时，要注意参与运算的操作数必是十进制数的BCD编码形式

（1） 压缩的BCD码调整指令

- DAA 加法的压缩BCD码调整指令

  格式：DAA

  功能：AL中的和调整为压缩BCD码 → AL

  - 必 跟在两个压缩BCD码的二进制加法指令ADD/ADC之后
  - 结果：AL中有2位压缩的BCD码
  - OF位无定义，但影响其他标志位

调整规则：

- 如和的数值在1010-1111之间或者AF标志为1，则加6（0110）就可得到正确结果

调整指令操作（按调整规则系统自动执行）：

第一步：调整低位

```
IF((AL AND 0FH)＞09H) OR (AF＝1)
THEN
    AL←AL＋06H;
    /*十进制个位数加6调整成压缩BCD码*/
    AF←1;
```

第二步：调整高位

```
IF((AL AND 0F0H)＞90H) OR (CF＝1)
THEN
    AL←AL＋ 60H;
    /*十进制十位数加6调整成压缩BCD码*/
    CF←1;
```

例3.64（p80）

- DAS　减法的压缩BCD码调整指令

  格式:DAS

  功能：AL中的差调整为压缩BCD码 → AL

  - 必　跟在两个压缩BCD码的二进制减法指令SUB/SBB之后
  - 结果：AL中有2位压缩的BCD码
  - OF位无定义，但影响其他标志位

调整指令操作（按调整规则系统自动执行）：

第一步：调整低位

```
IF((AL AND 0FH)＞09H) OR (AF＝1)
  THEN
   AL ← AL－ 06H;
   /*十进制个位数减6调整成压缩BCD码*/
   AF←1;
```

第二步：调整　位

```
IF((AL AND 0F0H)＞90H) OR (CF＝1)
THEN
   AL ← AL－ 60H;
   /*十进制十位数减6调整成压缩BCD码*/
   CF←1;
```

为什么减6？ 因为十进制借位应该当10用，而SUB把借位当16用少减了6

(2)非压缩的BCD码调整指令

- AAA　加法的ASCII码调整指令
  - AL中的和调整为非压缩BCD码 → AL
  - AH + 调整产生的进位值 → AH
  - 影响AF、CF，其他标志位无定义
- AAS　减法的ASCII码调整指令
  - AL中的差调整为非压缩BCD码 → AL
  - AH - 调整产生的借位值 → AH
  - 影响AF、CF，其他标志位无定义

- **AAM　乘法的ASCII码调整指令**
  - AL中的积调整为非压缩BCD码 → AX
  - AL/0AH，商→ AH，余数→ AL
  - AF、CF和OF位无定义，影响SF、ZF和PF

- **AAD　除法的ASCII码调整指令**
  - 用在除法指令前使用，为除法指令作好准备
  - 10*(AH)+AL(变成二进制数) → AL，0 → AH
  - AF、CF和OF位无定义，影响SF、ZF和PF

压缩BCD运算举例：

```
（1）MOV     AL, BCD1      ; BCD1=34H
     ADD     AL, BCD2      ; BCD2=59H,  (AL)=8DH
     DAA                   ;  8DH+06H=93H
     MOV     BCD3, AL      ; BCD3=93H

（2）MOV     AL, BCD1      ; BCD1=34H
     SUB     AL, BCD2      ; BCD2=59H ,  (AL)=0DBH
     DAS                   ;  0DBH－60H－06H=75H
     MOV     BCD3, AL      ; BCD3= 75 = - 25  (10n’补码）
```

P81

非压缩BCD运算举例：

```
（1） MUL     BL      ; (AX)=(AL)×(BL)=08 × 09
     AAM              ; (AL)/0AH= 48H /0AH→ 0702

（2）AAD              ; (AX) →(AH) ×0AH+(AL)=48H
     DIV     BL       ; (AL) = (AX)/(BL)=48H/4=12H
     AAM              ; (AL)/0AH=12H/0AH=0108
```

3.3.3 逻辑指令

逻辑指令包括：

1．逻辑运算指令

2．位测试并修改指令

3．位扫描指令

4．移位指令

1. 逻辑运算指令

逻辑运算指令可以对字或字节执行逻辑运算，386及其后继机型还可以执行双字操作

基本指令如下：

AND(and)	逻辑与
OR(or)	逻辑或
NOT(not)	逻辑非
XOR(exclusive or)	异或
TEST(test)	测试

名称	格 式	功 能	标 志
逻辑非	NOT DST	(DST)按位变反 送DST	不影响
逻辑与	AND DST, SRC	(DST)∧(SRC) →(DST)	CF和OF清0，影响SF、ZF及PF， AF无定义
测 试	TEST OPR1,OPR2	(OPR1)∧(OPR2)	同AND指令
逻辑或	OR DST, SRC	(DST)∨(SRC) →(DST)	同AND指令
逻辑 异或	XOR DST, SRC	(DST) ∀(SRC) →(DST)	同AND指令

- NOT指令不允许立即数
- 其他指令操作数寻址方式与MOV指令的限制相同
  - 一个操作数放在寄存器中，一个数据用任意寻址方式得到
- 逻辑与和逻辑测试的区别在于后者执行后只影响标志位而不改变任何操作数本身

逻辑运算指令用途

- **逻辑非指令**：可用于把操作数的每一位变反
- **逻辑与指令**：用于把某位清0（和0相与，也可称为屏蔽某位）、某位保持不变（和1相与）

```
X X X X X X X X
0 1 1 0 0 1 1 1
---------------
0 X X 0 0 X X X
```

- **逻辑或指令**：用于把某位置1(与1相或)、某位保持不变(与0相或)

```
X X X X X X X X
0 1 1 0 0 1 1 1
---------------
X 1 1 X X 1 1 1
```

- **逻辑异或指令**：用于把某位变反（与1相异或）、某位保持不变（与0相异或）

```
X X X X X X X X
0 1 1 0 0 1 1 1
---------------
X Y Y X X Y Y Y
```

- **逻辑测试指令**：可用于只测试某位值而不改变操作数

```
MOV   AL,00001111B  ;AL=00001111B

NOT   AL            ;AL=11110000B

AND   AL,0FH        ;清0  4位,低4位不变

OR    AL,30H        ;D5、D4位置1(与1相或)、其他位不变
```

```
X X X X X X X X
0 0 0 0 1 1 1 1
---------------
0 0 0 0 X X X X
```

```
IN    AL,61H
XOR   AL,2    ; 00000010
OUT   61H,AL  ; 使61H端口的D1位变反
```

```
X X X X X X X X
0 0 0 0 0 0 1 0
---------------
X X X X X X Y X
```

```
XOR   AX,AX   ; AX寄存器清0
```

例：屏蔽AL的第0、1两位

```
AND  AL, 0FCH
```

```
        * * * * * * * *
AND     1 1 1 1 1 1 0 0
        * * * * * * 0 0
```

例：置AL的第5位为1

```
OR  AL, 20H
```

```
        * * * * * * * *
OR      0 0 1 0 0 0 0 0
        * * 1 * * * * *
```

例：使AL的第0、1位变反

```
XOR  AL, 3
```

```
        * * * * * * 0 1
XOR     0 0 0 0 0 0 1 1
        * * * * * * 1 0
```

例：测试某些位是0是1

```
TEST  AL, 1
JZ    EVEN
```

```
        * * * * * * * *
AND     0 0 0 0 0 0 0 1
        0 0 0 0 0 0 0 *
```

2. 位测试并修改指令

- 386及其后继机型增加了本组指令

BT	位测试
BTS	位测试并置 1
BTR	位测试并置 0
BTC	位测试并变反

名称	格 式	功 能	标 志
位测试	BT DST, SRC	测试由SRC指定的 DST中的位	所选位值送CF, 其他标志无定义
位测试 并置位	BTS DST, SRC	测试并置1由SRC 指定的DST中的位	同上
位测试 并复位	BTR DST, SRC	测试并清0由SRC 指定的DST中的位	同上
位测试 并取反	BTC DST, SRC	测试并取反由SRC 指定的DST中的位	同上

- 首先把指定位的值送给CF标志，然后对该位按照指令的要求操作
- 目标可以是除立即数以外的任一种寻址方式
- 目标操作数的位置从最右边位开始、从0开始计数
- 源可以是8位的**立即数**、**寄存器**
  - 若源操作数是立即数，则其值不应超过目标操作数的长度
  - 也可用任一字寄存器或双字寄存器给出同一个值

```
MOV    AX,2357H
;AX=0010 0011 0101 0111B

BT       AX,0              ;CF=1, AX=2357H
;AX=0010 0011 0101 0111B

;AX=0010 0011 0101 0111B
BTC      AX,3              ;CF=0, AX= 235FH
;AX=0010 0011 0101 1111B
```

3. 位扫描指令

- 386及其后继机型增加了本组指令

BSF(bit scan forward)

正向位扫描（从最低位开始）

BSR(bit scan reverse)

反向位扫描（从最高位开始）

记录正向/反向检索到的第一个“1”的位置，记录在目标寄存器中

0010101001011000

(1) 正向位扫描指令 BSF

格式：BSF　REG, SRC

功能：从位0开始，从右向左扫描SRC操作数中第一个是1的位

- 若遇到第一个为1的位，则ZF置0，并把该位的位号送REG
- 若SRC＝0，则ZF置1,REG值不确定

- REG只能是字或双字通用寄存器；SRC不能是立即数，其他寻址方式均可
- 其他标志位无定义

ZF=0，扫描到1,位置记录在REG中
ZF=1，没有扫描到1，即SRC=0

(2) 反向位扫描指令 BSR

格式：BSR REG,RSC

功能：该指令从最　位开始，从左向右扫描，其他同BSF

4. 移位指令

(1) 移位指令（基本移位指令）

SHL　　SAL　　SHR　　SAR

(2) 循环移位指令

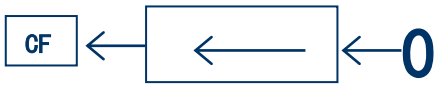
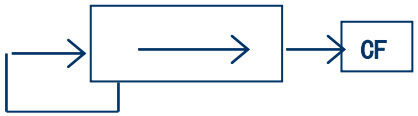
ROL　　ROR　　RCL　　RCR

(3) 双精度移位指令（386及后继机型使用本组指令）

SHLD　双精度左移指令

SHRD　双精度右移指令

(1) 移位指令

名 称	格 式	功 能	标 志
逻辑 左移	SHL DST, CNT		CF中总是最后移出的一位； ZF、SF、PF按结果设置；当CNT＝ 1时, 移位使DST最 位变化OF置1, 否则清0
算术 左移	SAL DST, CNT		同上
逻辑 右移	SHR DST, CNT		同上
算术 右移	SAR DST, CNT		同上

说明：**逻辑移位看作无符号数移位， 算术移位看作带符号数移位**

DST可以是8位、16位或32位的寄存器或存储器操作数

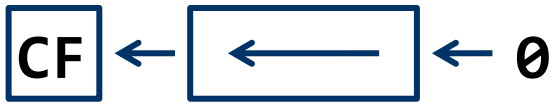
CNT是移位位数计数器

8086 / 8088对CNT的限定：

当CNT＝1时，直接写在指令中； 当CNT＞1时，由CL寄存器给出

其他机型对CNT的限定：

当CNT=1，或8位立即数（1-31）时，直接写在指令中； 也可由CL寄存器给出

SHL DST, CNT 

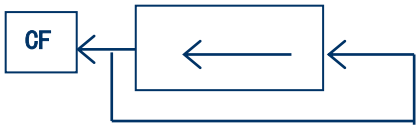
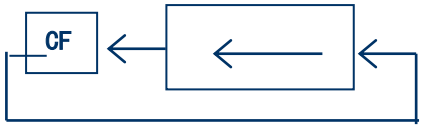
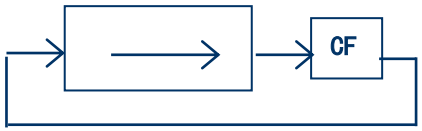
假设：CF=0，AL=10011100，CL=3

那么：CPU执行 SHL AL,1后， CF=1，AL=00111000

CPU执行 SHL AL,CL后，CF=0，AL=11100000

- SHL和SAL指令的功能相同，在机器中实际上它们对应的是同一种操作
- 使用这组指令
  - 可以实现基本的移位操作
  - 可以用于对一个数进行2^n的倍增或倍减运算，比直接使用乘除法效率要 得多

(2) 循环移位指令

名 称	格 式	功 能	标 志
循环 左移	ROL DST, CNT		不影响CF、OF以外的其它条件标志；CF中总是最后移进的位；当CNT＝1时, 移位使DST符号位改变则OF置1, 否则清0
循环 右移	ROR DST, CNT		同上
带进位 循环左移	RCL DST, CNT		同上
带进位 循环右移	RCR DST, CNT		同上

对DST和CNT的限定同基本移位指令

注意:

* OPR可用除立即数以外的任何寻址方式
* CNT=1，SHL　OPR, 1

  CNT>1，MOV　CL,　CNT

  　　SHL　OPR, CL　　; 以SHL为例
* 条件标志位:

  CF = 移入的数值

  $$OF = \begin{cases} 1 & CNT=1时，最高有效位的值发生变化 \\ 0 & CNT=1时，最高有效位的值不变 \end{cases}$$

  移位指令:

  　SF、ZF、PF 根据移位结果设置，AF无定义

  循环移位指令:

  　不影响 SF、ZF、PF、AF

例：(AX)= 0012H，(BX)= 0034H，把它们装配成(AX)= 1234H

```
MOV   CL，8
ROL   AX，CL
ADD   AX，BX
```

例：(BX) = 84F0H

(1) (BX) 为无符号数，求 (BX) / 2

```
SHR   BX，1          ；(BX) = 4278H
```

(2) (BX) 为带符号数，求 (BX) ×2

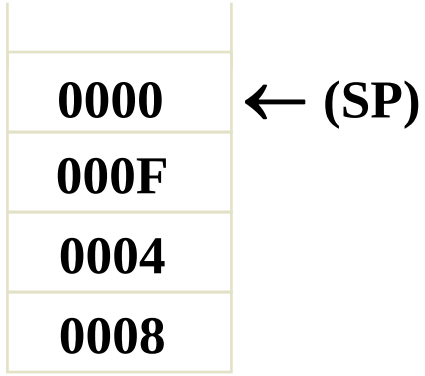
```
SAL   BX，1          ；(BX) = 09E0H，OF=1
```

(3) (BX) 为带符号数，求 (BX) / 4

```
MOV   CL，  2
SAR   BX，CL         ；(BX) = 0E13CH
```

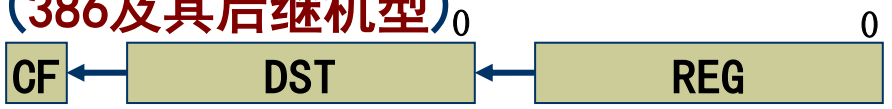
(3) (BX)=84F0H，把(BX)中的 16 位数每 4 位压入堆栈

```
        MOV     CH，4                ；循环次数
        MOV     CL，4                ；移位次数
NEXT:
        ROL     BX，CL
        MOV     AX，BX
        AND     AX，000FH
        PUSH    AX
        DEC     CH
        JNZ     NEXT
```



(3) 双精度移位指令

(1) 双精度左移指令 SHLD （386及其后继机型）



格式：SHLD　DST, REG, CNT

功能：把操作数DST左移由CNT指定的位（设为n），空出的位用REG端的n位填充，REG的内容不变，最后移出的位在进位标志CF中

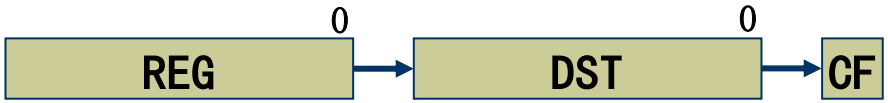
- DST：16或32位的寄存器或存储器操作数
- REG：与DST长度相同的寄存器
- CNT是移位位数：　8位立即数，或者CL内容
  - 8位立即数，提供0-31之间的值，大于31时机器自动 MOD32
  - CL内容指定
- 结果标志位：
  - CF=最后移出的一位
  - 如果CNT＝1，当移位后DST最　位发生变化时，OF置1，否则清0；当CNT＞1时OF值无定义
  - SF、ZF、PF按照DST结果设置
  - AF除移位次数为0外无定义

(2) 双精度右移指令 SHRD （386及其后继机型）

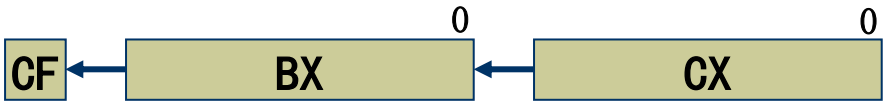
格式：SHRD　DST,REG,CNT

功能：　把操作数DST右移由CNT指定的位，空出的位用REG低端的n位填充，REG的内容不变，最后移出的位在进位标志CF中

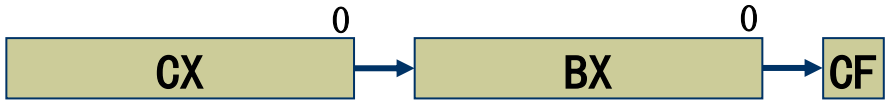
- 其他同SHLD



```
MOV     BX,8321H     ; 1000 0011 0010 0001
MOV     CX,5678H     ; 0101 0110 0111 1000

SHLD    BX,CX, 1     ; 0000 0110 0100 0010
;BX=0642H,  CX=5678H,  CF=1,  OF=1
```




```
SHRD    BX,CX,2      ; 0010 0000 1100 1000
;BX=20B8H,  CX=5678H,  CF=0,  OF=U
```



3.3.4 串处理指令

- 处理存放在存储器里的**数据串**，所有串指令都可以处理**字节**或**字**，386及后继机型还可以处理**双字**
- 利用串操作指令可以直接处理两个存储器单元的操作数，方便地处理**字符串**或**数据块**
- 串处理指令包括：

MOVS	串传送	CMPS	串比较
SCAS	串扫描	LODS	从串取
STOS	存入串		
INS	串输入（从I/O端口输入）		
OUTS	串输出（向I/O端口输出）		

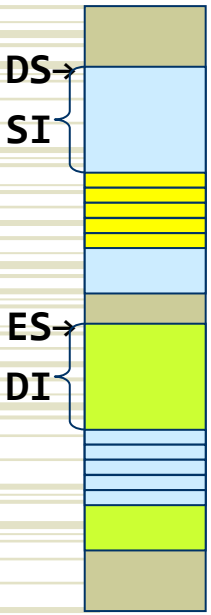
(1). MOVS 串传送

指令隐含格式：MOVS DST,SRC

显式：MOVSB DST,SRC ;字节传送

MOVSW DST,SRC ;字传送

MOVSD DST,SRC ;双字传送



- 隐含格式：应该在操作数中表明字节、字、双字传送操作

  例如：MOVS ES:BYTE PTR[DI], DS:[SI]

- 显式格式：经常使用，根据指令操作符的最后一个字母决定字节、字、双字传送操作
- 不影响标志位

① 串处理指令中操作数

- 寄存器型操作数：只能放在累加器中
  - 字节操作数应放在AL中
  - 字操作数放在AX中
  - 双字操作数放在EAX中
- 存储器型操作数：应先建立地址指针
  - 源操作数时，必　把源串　地址偏移量放入SI寄存器（若地址长度是32位的则使用ESI），段缺省寻址DS所指向的段，允许使用段超越前缀
  - 目标操作数时，必　把目标串　地址偏移量放入DI寄存器（若地址是32位的则使用EDI），段寄存器是ES

② 地址指针的修改

- 串指令执行后系统自动修改地址指针SI（ESI）、DI（EDI）
  - 字节型操作 ±1
  - 字型操作 ±2
  - 双字型操作 ±4

③ 修改方向：基于方向标志DF

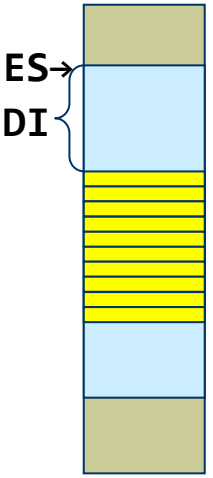
- DF＝0，则地址指针增量
- DF＝1，则地址指针减量
- 可以用CLD和STD指令对DF置0或1

◆ 串处理指令前应先建立地址指针、方向标志

(2)STOS指令

格式：STOS　DST；STOSB DST；STOSW DST；STOSD DST

- 源操作数在AL、AX或EAX；其他同MOVS
- 与REP结合使用在初始化某一缓冲区时很有用

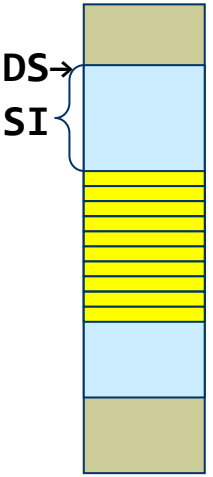



```
STOS ES:BYTE PTR[DI]
```

(3)LODS指令

格式：LODS　SRC；LODSB SRC；LODSW SRC；LODSD SRC

- 目的操作数在AL、AX或EAX；其他同MOVS
- 将某一缓冲区数据逐个取出测试时很有用




```
LODS DS:BYTE PTR[SI]
```

(4) INS指令

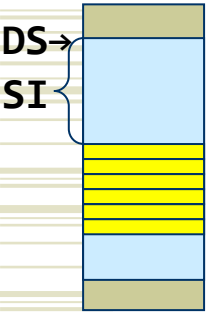
格式：INS DST,DX； INSB DST,DX； INSW DST,DX； INSD DST,DX

- 源操作数在IO寄存器中，其端口号（IO寄存器地址）在DX寄存器中
- 目标操作数、地址指针的修改、修改方向、标志位等同MOVS规定
- 与REP前缀结合使用可以实现IO寄存器中连续数据送到存储缓冲区

```
INS ES:BYTE PTR[DI], DX
```

(5) OUTS指令

格式：OUTS DX,SRC；OUTSB DX,SRC；OUTSW DX,SRC；OUTS DX,SRC

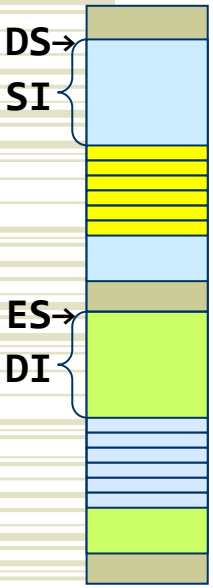


- 目标操作数地址是IO寄存器，其端口号（IO寄存器地址）在DX寄存器中
- 源操作数、地址指针的修改、修改方向、标志位等同MOVS规定
- 与REP前缀结合使用可以实现存储缓冲区的一组连续数据送到IO寄存器

```
OUTS DX, DS:BYTE PTR[SI]
```

- 操作数可以字节、字、双字
- 与REP前缀结合使用时注意执行速度和IO端口处理速度相匹配

(6) CMPS指令



格式： CMPS DST,SRC； CMPSB DST,SRC； CMPSW DST,SRC；
CMPSD DST,SRC

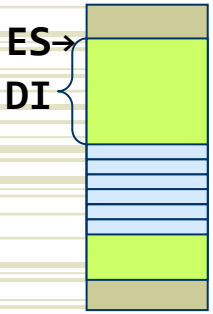
执行的操作： **两个字符串比较**

- (SRC) – (DST) ，但结果不保存，根据结果设置标志位
- 目标操作数和源操作数的地址指针（变址寄存器DI,SI）的修改、修改方向同MOVS规定
- 操作数的规定和寻址方式：同MOVS

(7) SCAS指令

格式： SCAS DST； SCASB DST； SCASW DST； SCASD DST

执行的操作： **累加器内容与字符串比较**



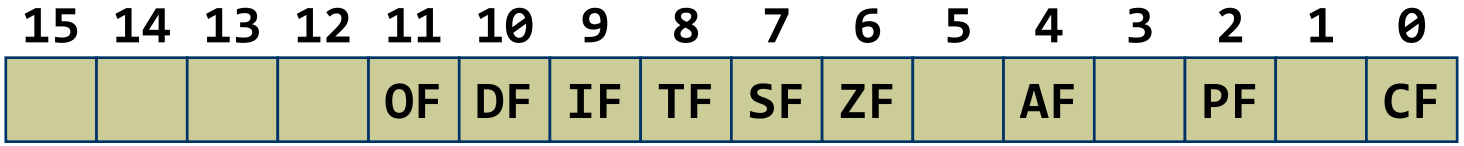
- (AL/AX/EAX)–(DST)，但结果不保存，根据结果设置标志位
- 目标操作数的地址指针（变址寄存器DI）的修改、修改方向同MOVS规定
- 目标操作数的规定和寻址方式：同MOVS

课内测试 04-2-2

1. 请在填空中[填空1] 填写“19”；（10分）

2. 字操作的串处理指令中，CPU将：（10分）

根据标志寄存器中的标志位[填空2]，自动对变址寄存器SI和DI ±[填空3]。



```
MOV   ES:BYTE PTR[DI], DS:[SI]
MOVS  ES:BYTE PTR[DI], DS:[SI]
```

```
MOV   AL, DS:BYTE PTR[SI]
LODS      DS:BYTE PTR[SI]
```

思考题：单个串处理指令与基本指令中的MOV等指令区别？

- 实现功能是相同的；
- 但单个串处理指令执行时将操作数地址的变址指针根据修改方向自动修改，且寄存器只能使用累加器；
- MOV指令2个操作数不能同为存储单元

3.3.5 重复前缀

- 单条的串指令只能处理数据串中的一个数据
- 处理整个数据串时，必　要有重复前缀才可以
- 串处理指令使用的前缀，必　与串指令一起使用

MOVS、STOS、LODS、 INS、OUTS指令前缀：

```
REP                    ; 重复
```

CMPS、SCAS指令前缀：

```
REPE/REPZ              ; 相等/为零则重复
REPNE/REPNZ            ; 不相等/不为零则重复
```

字符串比较和串中查找字符很有用

1. 重复 REP

格式：REP　串处理指令

- 串处理指令可以是MOVS、STOS、LODS、INS、OUTS
- 例如：REP　MOVS ES:BYTE PTR[DI], DS:[SI]

执行的操作：

① 如(count reg)=0,则退出REP, 否则，往下执行

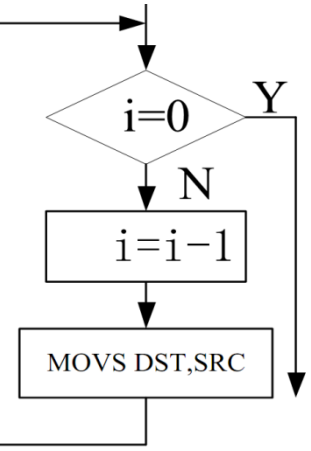
② (count reg)-1 → (count reg)

③ 执行其后的串处理指令

④ 转①，即重复①～③

- 16位地址时count reg是CX;32位地址时count reg是ECX
- 执行前必　设置好count reg

例如：REP　MOVS ES:BYTE PTR[DI], DS:[SI]



① 如CX=0,则退出REP, 该指令执行结束；否则，往下执行

② CX-1 → CX

③ 执行《MOVS ES:BYTE PTR[DI], DS:[SI]》串处理指令

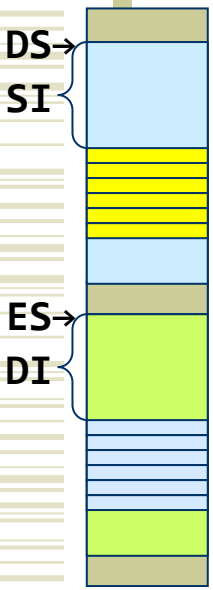
- (DS*16+SI) → (ES*16+DI)
- DF=0时, SI+1 → SI, DI+1 → DI　(字节操作，+1)
- DF=1时, SI-1 → SI, DI-1 → DI

④ 转①，即重复①～③

- REP MOVSB ES:[DI], DS:[SI] 指令相当如下一组指令：

```
REPLOOP:CMP CX,0
        JZ NEXT
        DEC CX
        MOVSB ES:[DI], DS:[SI]
        JMP REPLOOP

NEXT:   ......
```

2. 相等重复前缀 REPE / REPZ



- 格式： REPE / REPZ CMPS 或 SCAS指令
- 执行的操作为：
  ① 若 CX＝0（计数到）或 ZF＝0（不相等），则结束重复
  ② 否则，修改计数器CX－1→CX，执行后跟的串操作指令。转①，继续重复上述操作

◆ REPE CMPSB ES:[DI], DS:[SI]指令相当如下一组指令：

```
REPLOOP:CMP CX,0
        JZ NEXT
        DEC CX
        CMPSB ES:[DI], DS:[SI]   ;比较，并根据DF修改DI,SI
        JE REPLOOP

NEXT:   ……
```

REPE CMPSB ES:[DI], DS:[SI]

3. 不等重复前缀 REPNE / REPNZ

- 格式： REPNE / REPNZ　CMPS或 SCAS指令
- 执行的操作为：
  ① 若 CX＝0（计数到）或ZF＝1(相等)，则结束重复
  ② 否则，修改计数器 CX－1→CX，执行后跟的串操作指令。转①，继续重复上述操作

- REPNE CMPSB ES:[DI], DS:[SI]指令相当如下一组指令：

```
REPLOOP:CMP CX,0
        JZ NEXT
        DEC CX
        CMPSB ES:[DI], DS:[SI]   ;比较，并根据DF修改DI,SI
        JNE REPLOOP

NEXT:   ......
```

REPNE CMPSB ES:[DI], DS:[SI]

对于串处理指令需要注意的其他几个问 ：

1. 如果需要在同一段内传送或比较数据

   ① 在DS和ES中设置相同的段基地址

   ② 源操作数字段使用段跨越前缀

   如 MOVSB [DI],ES:[SI]

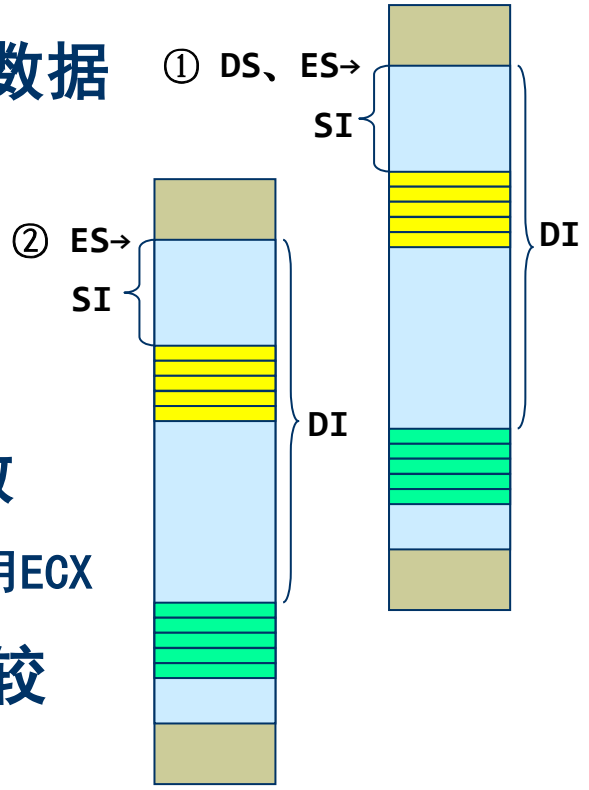
2. count reg(CX)中是重复执行次数

   - 计数器：16位地址时用CX；32位地址时用ECX

3. 根据DF进行正向、反向传送或比较

   - DF=0,操作数地址自动递增修改
   - DF=1,操作数地址自动递减修改

4. 执行前必 设置好count reg，DF，DI,SI



例3.86

3.3.6 控制转移指令

控制转移指令包括：

1．无条件转移指令

2．条件转移指令

3．条件设置指令

4．循环指令

5．子程序调用/返回

6．中断指令/返回

1、无条件转移指令JMP

(1)	段内直接短转移	JMP SHORT OPR
(2)	段内直接近转移	JMP NEAR PTR OPR
(3)	段内间接近转移	JMP WORD PTR OPR
(4)	段间直接转移	JMP FAR PTR OPR
(5)	段间间接转移	JMP DWORD PTR OPR

不影响标志位

(1) 段内直接短转移 JMP SHOT OPR

- $(IP_{新})=(IP_{原})+$ 8位位移量，CS不变
- 8位EA的位移量由目标地址OPR确定
- 80386极其后续机型：(EIP新)=(EIP原)+ 8位位移量，CS不变
- P102，图3.22

```
        JMP SHORT B1
A1:     ADD AX,BX
B1:     …
```

(2) 段内直接近转移 JMP NEAR PTR OPR

- $(IP_{新})=(IP_{原})+$ 16位位移量，CS不变
- 16/32位目标指令EA的位移量由OPR确定
- 80386极其后续机型：$(EIP_{新})=(EIP_{原})+$ 32位位移量，CS不变

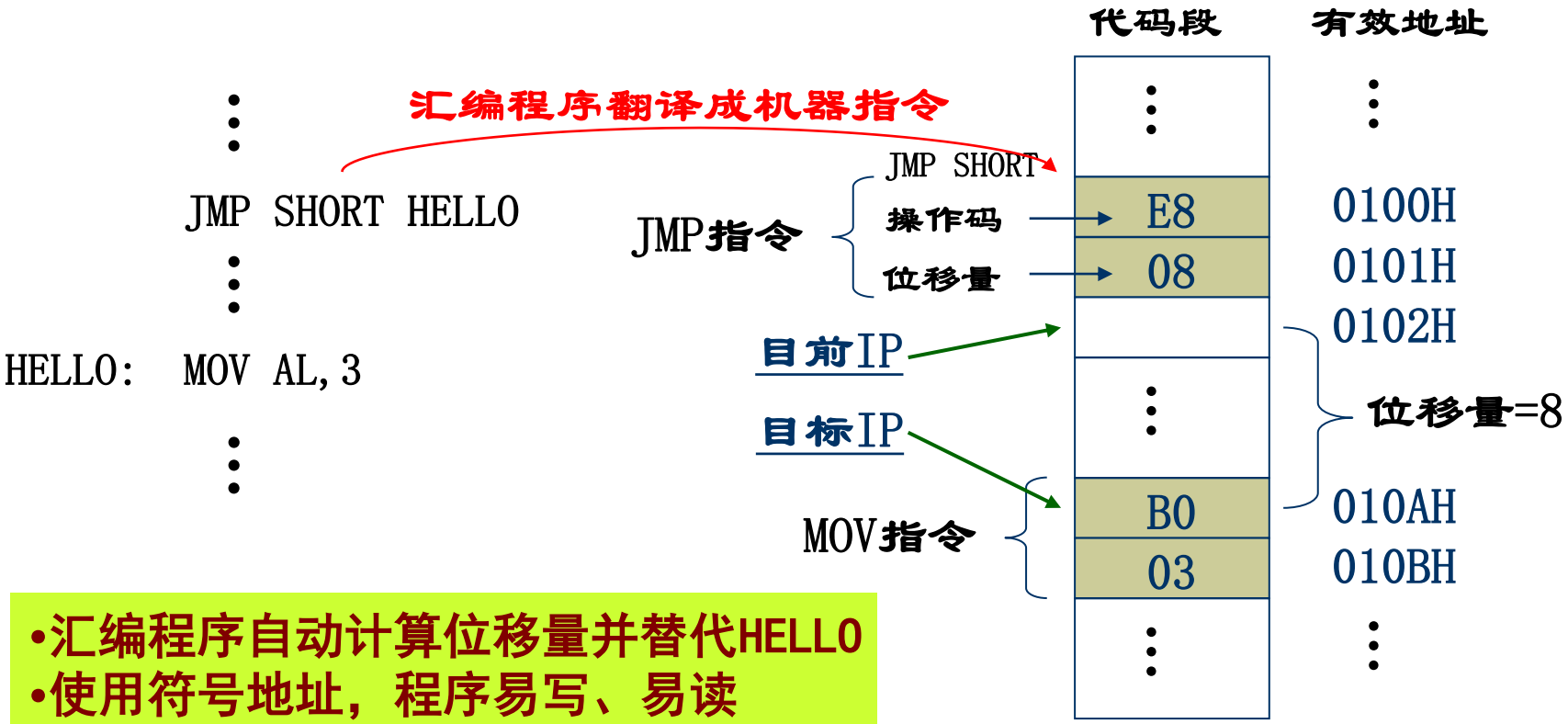
```
      JMP NEAR PTR B2
A2: ADD  AX,CX
        …
B2: SUB  AX,CX
```

(3) 段内间接近转移 JMP WORD PTR OPR

- EA值由OPR的寻址方式确定，它可以使用除立即数方式以外的任一种寻址方式
- $(IP_{新})$ = (EA)，CS不变
- 80386极其后续机型：
  $(EIP_{新})$= (EA)，CS不变

```
      LEA BX,B2 ; 装入有效地址
      JMP  WORD PTR  BX
A2: ADD AX,CX
        …
B2: SUB AX,CX
```

- **注意概念：**
  - 偏移量=有效地址EA：
    - 一般相对于段基地址（段开始地址）的位移量
  - 位移量：两个有效地址EA差，EA_2-EA_1
    - 位移量是带符号数



- 汇编程序自动计算位移量并替代HELLO
- 使用符号地址，程序易写、易读

(4) 段间直接转移　JMP FAR PTR OPR

OPR直接给出目标指令所在段的16/32位段基地址和段内有效地址EA

- $(IP_{新})$ =OPR给出的16位段内偏移量
- $(CS_{新})$=OPR给出的16位段基地址

P103，　图3.23

80386极其后续机型：

- $(EIP_{新})$=OPR给出的32位段内偏移量
- $(CS_{新})$=OPR给出的32位段基地址

(5) 段间间接转移　JMP DWOR PTR OPR

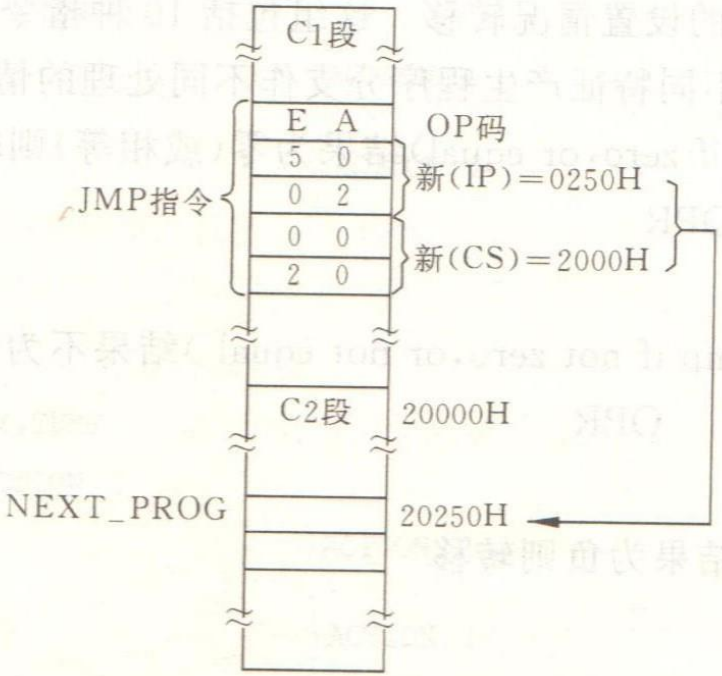
OPR存储器寻址，给出目标指令所在段的16/32位段基地址和段内有效地址EA

- $(IP_{新})$ =目标指令所在段的16位EA　;存储器间接寻址获得
- $(CS_{新})$=目标指令所在段的16位段基地址　;存储器间接寻址获得

80386极其后续机型：

- $(EIP_{新})$=目标指令所在段的32位EA　;存储器间接寻址获得
- $(CS_{新})$=目标指令所在段的32位段基地址　;存储器间接寻址获得

```
C1   SEGMENT
                ⋮
                JMP     FAR PTR NEXT_PROG
                ⋮
C1   ENDS
C2   SEGMENT
                ⋮
NEXT_PROG:
                ⋮
C2   ENDS
```



2、条件转移指令

指令格式Jcc　OPR

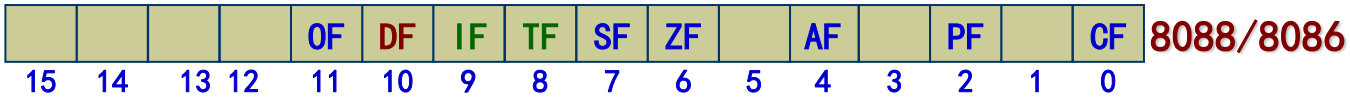
- OPR定义同无条件转移指令
- 根据上一条指令设置的条件码来判别测试条件
- 这类指令本身并不影响标志
- 分为4组来讨论

只能是段内短转移！

(1) 根据单个条件标志(状态位)的情况转移转移

JZ 结果为零转移	JNZ 结果不为零转移
JS 结果小于零转移	JNS 结果不小于零转移
JO 结果溢出转移	JNO 结果不溢出转移
JP 结果中1的个数偶数转移	JNP 结果1的个数奇数转移
JC 结果进位/借位转移	JNC 结果没有进位/借位转移

				OF	DF	IF	TF	SF	ZF		AF		PF		CF	8088/8086
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	



8088/8086

JZ (或JE) 结果为零转移　　如果ZF=1，转移
JNZ (或JNE)结果不为零转移　　如果ZF=0，转移

JS 结果小于零（为负）转移　　如果SF=1，转移
JNS 结果不小于（为正）零转移　　如果SF=0，转移

JO 结果溢出转移　　如果OF=1，转移
JNO 结果不溢出转移　　如果OF=0，转移

JP (或JPE)结果1的个数偶数转移　　如果PF=1，转移
JNP(或JPO)结果1的个数奇数转移　　如果PF=0，转移

JC(或JB,或JNAE)结果进位/借位转移　　如果CF=1，转移
JNC(或JNB,或JAE)结果没有进位/借位转移　　如果CF=0，转移

状态位由该指令前边的运算等指令决定，这组指令只判断、跳转

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

8088/8086

(2) 比较两个无符号数，并根据比较结果转移*

A（高于），B（低于），E（等于）

	格式		测试条件
<	JB (JNAE, JC)	OPR	CF = 1
⩾	JNB (JAE, JNC)	OPR	CF = 0
≤	JBE (JNA)	OPR	CF∨ZF = 1
>	JNBE (JA)	OPR	CF∨ZF = 0

* 适用于地址或双精度数低位字的比较

指令JZ/JE和 JNZ/JNE同样可以用于两个无符号数的比较转移

比较两个无符号数由该指令的前边其他指令执行并设置条件标志，这组指令只判断、跳转

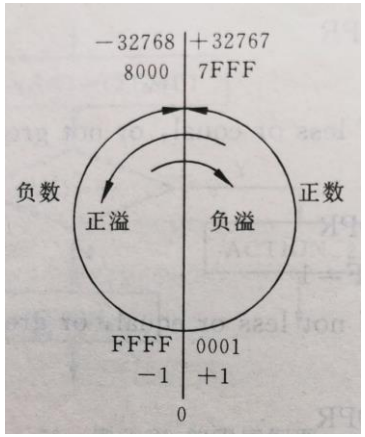
```
CMP DST, SRC
JB  NEXT
```

(3) 比较两个带符号数以后，根据比较结果 转移

G（大于），L（小于），E（等于）

<	JL (JNGE)	SF∀OF=1
≥	JNL (JGE)	SF∀OF=0
≤	JLE (JNG)	(SF∀OF)∨ZF=1
>	JNLE (JG)	(SF∀OF)∨ZF=0

SF	OF	
0	0	0
0	1	1
1	0	1
1	1	0



SF	OF	ZF	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

比较两个无符号数由该指令的前边其他指令执行并设置条件标志，这组指令只判断、跳转

例：X、Y是带符号数，编制程序实现：如果 X>50，转到TOO_HIGH；否则 |X-Y| → RESULT，如果溢出转到OVERFLOW.

```
                MOV   AX, X
                CMP   AX, 50
                JG    TOO_HIGH
                SUB   AX, Y
                JO    OVERFLOW
                JNS   NONNEG
                NEG   AX
    NONNEG:
                MOV   RESULT, AX
TOO_HIGH:
                ……
OVERFLOW:
                ……
```

(4) 测试CX或ECX的值为0则转移

- JCXZ　　CX寄存器内容为零则转移
- JECXZ　　ECX寄存器内容为零则转移

- 这组指令只能提供8位位移量，即短转移

课内测试 05-1-1

1. 请在填空中[填空1]填写“56”；(10分)
2. 条件转移指令使用相应助记符，体现了机器指令操作和上下文含义，可以使程序易编程和阅读等，但对应的机器指令实质上是根据标志位是1或0跳转，如：（10分）
   - JZ　AA1：如果[填空2]，转移到AA1
   - JNC AA1：如果[填空3]，转移到AA1

   - 说明：填空2，3的答案格式　XX=0 或 XX=1

3．条件设置指令

- 也称为条件字节设置指令，386及后继机型才提供这组指令
- 这些指令根据前边指令对状态标志位的设置，将目标字节置1或清0
  - 格式：SET_{CC}　DST
  - 功能：如果条件为真,则置DST=1,否则DST=0
- 用于保存条件码，在以后适当时机再判断转移等
  - 这时用这组指令效率　、简单
- 这组指令不会影响标志位

指令格式	功　能	测试条件
SETZ/SETE DST	等于0/相等时置DST为1；否则，DST为0	ZF = 1
SETNZ/SETNE DST	不等于0/不相等时置DST为1；否则，DST为0	ZF = 0
SETS DST	为负时置DST为1；否则，DST为0	SF = 1
SETNS DST	非负时置DST为1；否则，DST为0	SF = 0
SETO DST	溢出时置DST为1；否则，DST为0	OF = 1
SETNO DST	无溢出时置DST为1；否则，DST为0	OF = 0
SETP/SETPE DST	结果低8位有偶数个1时置DST为1；否则，DST为0	PF = 1
SETNP/SETPO DST	结果低8位有奇数个1时置DST为1；否则，DST为0	PF = 0
SETG/SETNLE DST	大于/不小于等于时置DST为1；否则，DST为0	ZF = 0 and SF = OF
SETNG/SETLE DST	不大于/小于等于时置DST为1；否则，DST为0	ZF = 1 or SF≠OF
SETL/SETNGE DST	小于/不大于等于时置DST为1；否则，DST为0	SF≠OF
SETNL/SETGE DST	不小于/大于等于时置DST为1；否则，DST为0	SF = OF
SETA/SETNBE DST	高于/不低于等于时置DST为1；否则，DST为0	CF = 0 and ZF = 0
SETNA/SETBE DST	不高于/低于等于时置DST为1；否则，DST为0	CF = 1 or ZF = 1
SETB/SETNAE/SETC DST	低于/不高于等于/有进位时DST置1；否则，DST为0	CF = 1
SETNB/SETAE/SETNC DST	不低于/高于等于/无进位时DST置1；否则，DST为0	CF = 0

4. 循环指令

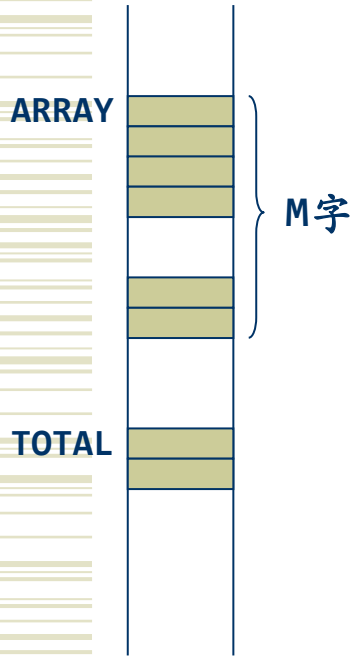
80X86为了简化循环程序的设计，设计了一组循环指令如下：

LOOP OPR

LOOPE/LOOPZ OPR

LOOPNE/LOOPNZ OPR

参看P113 例子3.96 (P113.ASM)




```
    MOV   CX, M
    MOV   AX, 0
    MOV   SI,  AX
START_LOOP:
    ADD   AX,
    ARRAY[SI]
    ADD   SI, 2
    LOOP   START_LOOP
    MOV   TOTAL,  AX
```

```
    MOV   CX, M
    XOR   AX, AX
    MOV   SI,  AX
    JCXZ   A1
START_LOOP:
    ADD   AX,  ARRAY[SI]
    ADD   SI, 2
    LOOP   START_LOOP
A1: MOV   TOTAL,  AX
```

循环指令特点：

① 循环入口地址（指令中的LABEL）只能在当前IP值的－128～＋127范围之内， 即短转移

② 用CX或ECX（计数操作数长度为32位时）作为循环次数的计数器

③ 不影响标志

思考 ：循环入口地址，为什么不能近转移或跨越段？

- 一般循环体比较小！一般情况下短转移指令就够用，指令系统越简单越好
- 循环指令在程序中出现和执行 率最 ！循环指令应该短、快速

1. 循环指令 LOOP

```
star: ...
      ...
      loop start
      ...
```

格式：LOOP　OPR

功能：

① (CX)－1→(CX)

② 若(CX)≠0，则转向标号处执行循环体，即IP+位移量 →IP

- 否则，退出循环体，　序执行下一条指令，即 IP 不变

2. 相等循环指令 LOOPE/LOOPZ

格式：LOOPE/LOOPZ　OPR

功能：

① (CX)－1→(CX) ；注意不影响ZF，这时ZF是前面指令的执行结果

② 若(CX)≠0 and ZF＝1，则转向标号处执行循环体，
即IP+位移量 →IP

- 否则，退出循环体，　序执行下一条指令，即 IP 不变

循环指令不影响标志

3. 不等循环指令 LOOPNE/LOOPNZ

格式：LOOPNE/LOOPNZ　OPR

功能：

① (CX)－1→(CX)　;注意不影响ZF，这时ZF是前面指令的执行结果

② 若(CX)≠0 and ZF＝0，则转向标号处执行循环体，即IP+位移量 →IP

- 否则，退出循环体，　序执行下一条指令，即IP 不变

参看P113 例子3.96

5. 子程序调用/返回指令

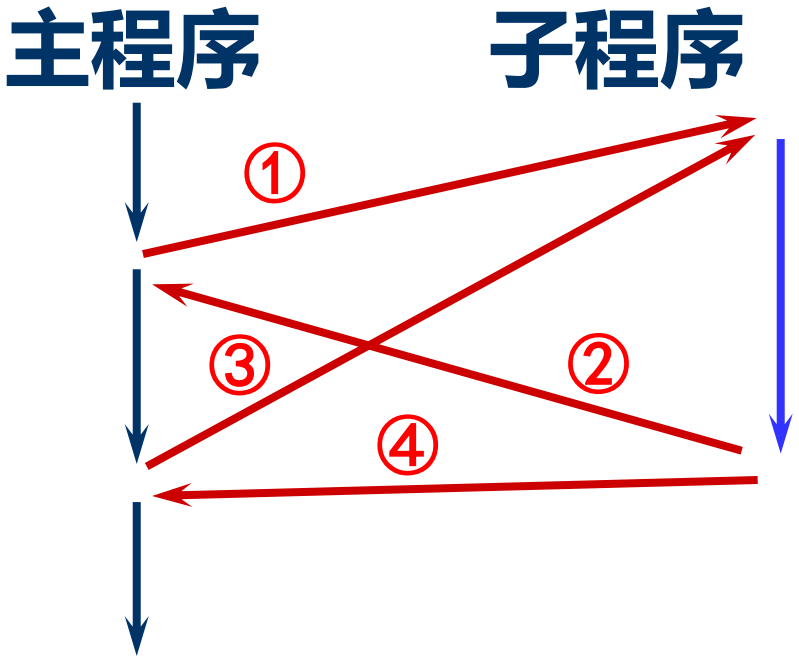
- 子程序:子程序结构相当于　级语言中的过程(PROCEDURE).为便于模块化程序设计，往往把程序中某些具有独立功能的部分编写成独立的程序模块，称之为子程序
- 程序中 由子程序调用指令调用子程序，而在子程序执行完后由返回调用指令返回调用程序继续执行
- 80X86提供了以下指令

  CALL　子程序调用

  RET　子程序返回

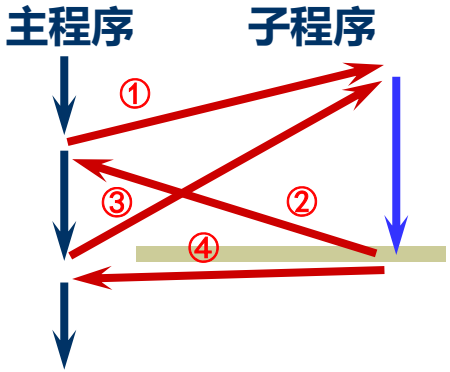
汇编程序中一定要注意正确使用堆栈及返回方式等，而高级语言不必关心这些

主程序和子程序关系



主子程序关系示意图

子程序多次被调用



主子程序关系示意图

1）. CALL指令

(1) 段内直接近调用　　CALL　DST

(2) 段内间接近调用　　CALL　DST

(3) 段间直接远调用　　CALL　DST

(4) 段间间接远调用　　CALL　DST

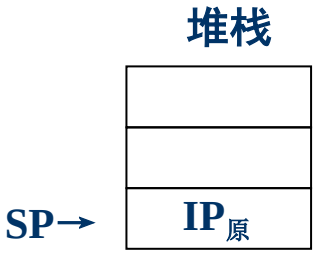
与无条件转移指令的转移方式相同，不同之处是CPU会自动将返回点(IP,CS)保存在堆栈中

① 段内直接调用：

格式：CALL　DST

功能：执行时先把返回地址（$IP_{原}$）压入堆栈，

再使（$IP_{新}$）=（$IP_{原}$）+指令中给出的16位位移量



② 段内间接调用：

格式：CALL　DST　　;REG／M寻址方式

功能：执行时先把返回地址（$IP_{原}$）压入堆栈，

再使(IP新)=指令指定的16位通用寄存器或内存单元的内容

③ 段间直接调用：

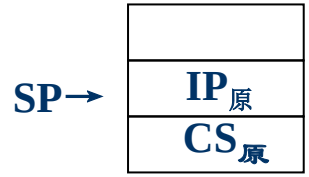
格式：CALL　DST

功能：执行时先把返回地址(当前IP值和当前CS值)压入堆栈，再把指令中给出的偏移量（EA）部分送给IP，段基址部分送给CS

④ 段间间接调用：

格式：CALL　M

功能：执行时先把返回地址(当前IP值和当前CS值)压入堆栈，再把存储器中连续4字节的低字送给IP，　字送给CS



2）RET指令

段内返回
- (1) 段内近返回　　RET
- (2) 段内带立即数近返回　　RET　EXP

段间返回
- (3) 段间远返回　　RET
- (4) 段间带立即数远返回　　RET　EXP

- 执行这组指令可以返回到被调用处
- 不影响标志
- 返回地址隐含，地址在堆栈中，隐含执行下面操作：
  ① 段内：POP IP(EIP)
  ② 段间：POP IP(EIP)， POP CS
- 带立即数的返回指令：主要是为了配合在调用子程序前通过堆栈给子程序传递参数

如何判断段内和段间返回？子程序段说明时会给出

(1) 段内回指令 RET

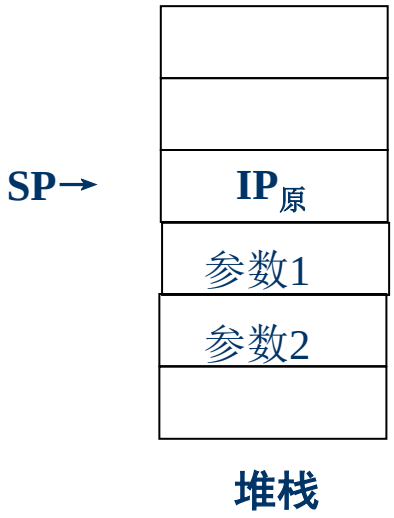
格式：RET

功能：隐含执行 POP IP(EIP)

(2) 段间远返回 RET

格式：RET

功能：隐含执行 ①POP IP(EIP)；②POP CS



(3) 段内带立即数的返回指令

格式：RET imm_{16}

功能：隐含执行 ①POP IP(EIP)；②修改栈顶指针 (SP)＋imm_{16} → (SP)

注：其中imm_{16}是16位的立即数，设通过堆栈给子程序传递了n个字型参数，则imm_{16}＝2n

(4) 段间带立即数远返回 RET EXP

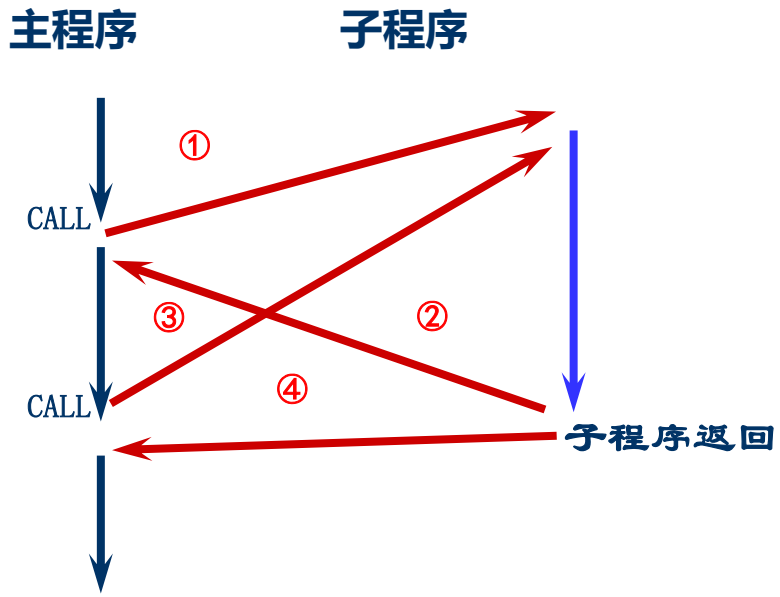
格式：RET imm_{16}

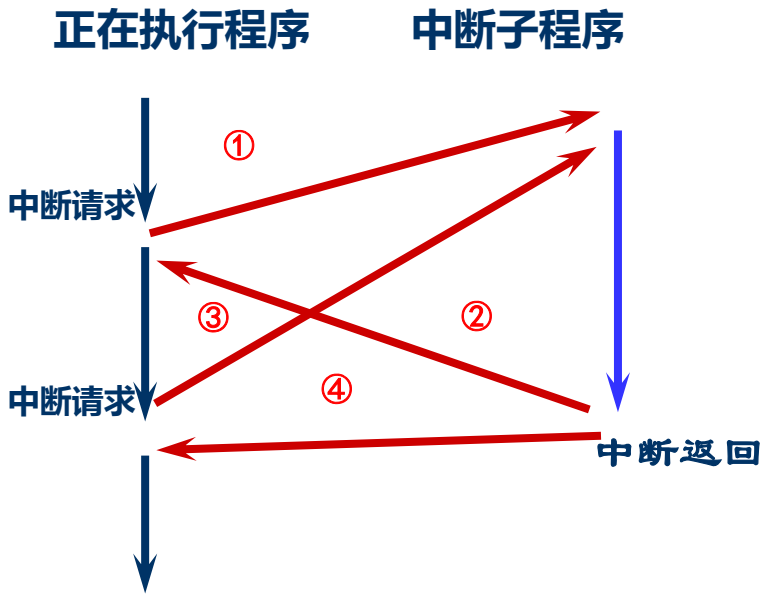
功能：隐含执行 ①POP IP(EIP)；②POP CS； ②修改栈顶指针 (SP)＋imm_{16} → (SP)

6、 中断

- 有时当系统运行或者程序运行期间在遇到某些特殊情况时，需要计算机自动执行一组专门的程序来进行处理。这种情况称为中断，所执行的这组程序称为中断例行程序或中断子程序

- 其它随机事件，如I/O控制和数据传送，不采用中断方式系统效率会很低





- 主程序调用
- 主、子程序都是程序的一部分

- 中断请求打断正在执行程序
- 执行程序和中断程序关系无关

中断过程：向CPU发中断请求；CPU响应中断，中断正在执行程序，转中断子程序；返回被中断程序。

- CPU响应一次中断时，也要和调用子程序时类似地把指令指针和CS保存到堆栈。为了能全面的保存现场信息，以便在中断处理结束时返回现场，需要把反映现场状态的FLAGS保存入栈，然后才能转到中断服务程序去执行

- 从中断返回时，需要恢复指令指针IP、CS、FLAGS

- 有关中断的指令：（软中断）

  INT　中断指令
  INTO　如溢出则中断
  IRET　从中断返回

中断请求方式：

1. **软中断：程序中安排**
2. **硬件中断：与程序无关**

1） 中断向量

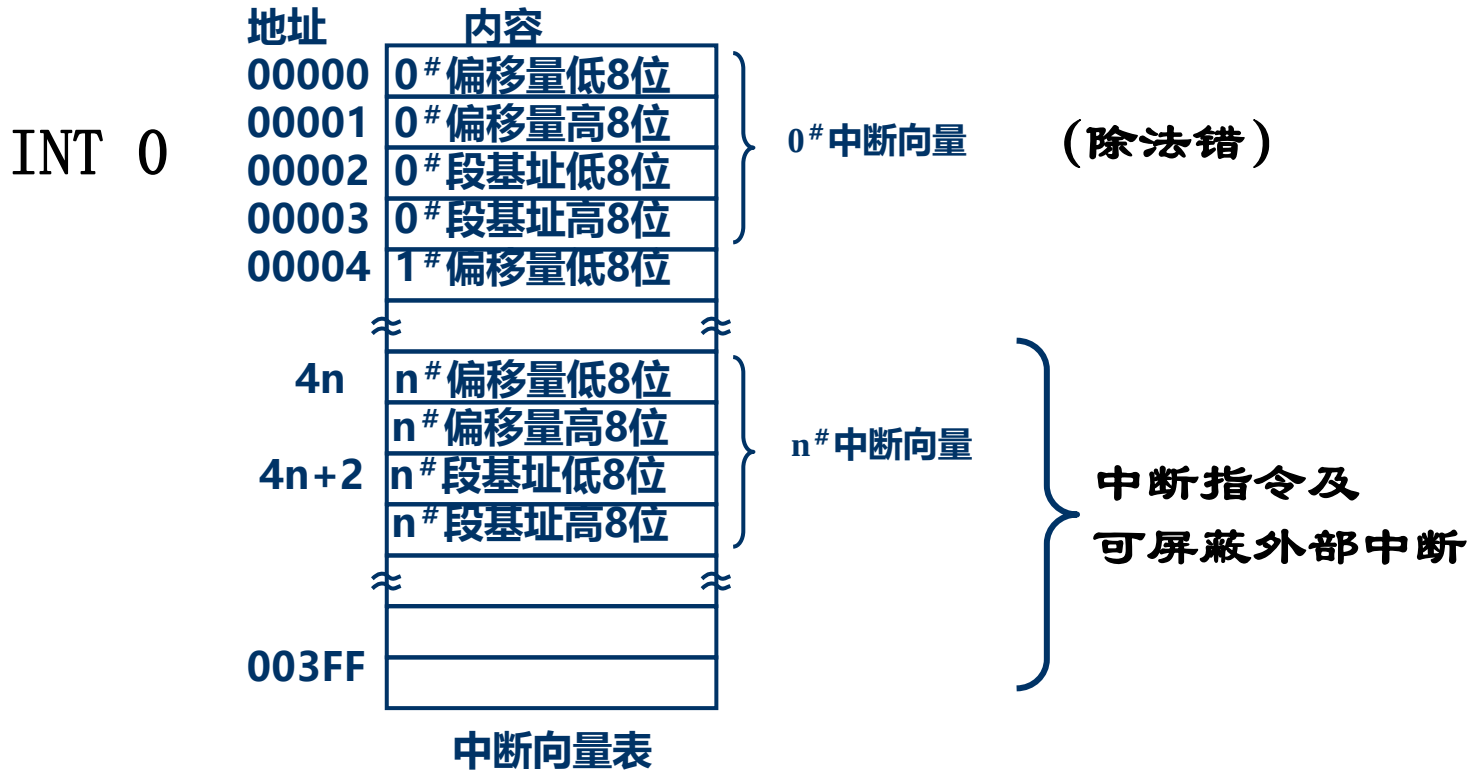
- 中断向量：中断处理子程序的入口地址
- 在PC机中规定中断处理子程序为FAR型
  - 每个中断向量占用4个字节，其中低两个字节为中断向量的偏移量部分， 两个字节为中断向量的段基址部分

2） 中断类型号

- IBM PC机共支持256种中断，相应编号为0～255，把这些编号称为中断类型号

3）中断向量表

- 256种中断有256个中断向量，把这些中断向量按照中断类型号由小到大的　序排列，形成中断向量表
- 表长为4×256＝ 1024字节，从内存的0000:0000地址开始存放，占用内存最低端的0～3FFH单元
- 请见图3.29



4) 软件中断指令 INT

- 在8086/8088中,中断分为内中断(或称软中断)和外中断(或称硬中断),这里只介绍内中断的中断调用指令
- 格式：INT n ；n为中断类型号
- 功能：中断当前正在执行的程序，把当前的FLAGS、CS、IP值依次压入堆栈(保护断点),关中断（IF=0）、陷阱(TF=0)和对 （AC=0),然后从中断向量表的4n处取出n类中断向量,其中(4n)→IP,(4n+2)→CS，转去执行中断处理子程序

5) 中断返回指令 IRET

- 格式：IRET
- 功能：从栈顶弹出三个字分别送入IP、CS、FLAGS寄存器（按中断调用时的逆序恢复断点）， 返回到原程序断点继续执行

6）子程序和中断子程序中特别注意：

- 为了保证返回地址正确，子程序中PUSH和POP必一对一出现；
- 凡子程序中使用寄存器，必　在子程序开始推入堆栈，子程序返回指令前出栈以**按序**恢复寄存器内容；
  - 有些CPU机型在中断响应/返回时另外也自动保存/恢复了常用寄存器
- 注意关中断和开中断时机！

3.3.7 处理机控制指令

1. 标志处理指令

汇编格式	功　　能	影响标志
CLC (Clear Carry)	把进位标志CF清0	CF
STC (Set Carry)	把进位标志CF置1	CF
CMC (Complement Carry)	把进位标志CF取反	CF
CLD (Clear Direction)	把方向标志DF清0	DF
STD (Set Direction)	把方向标志DF置1	DF
CLI (Clear Interrupt)	把中断允许标志IF清0	IF
STI (Set Interrupt)	把中断允许标志IF置1	IF

2. 其他处理机控制指令

不影响标志

名称	汇编格式	功 能	说 明
空操作	NOP (No Operation)	空操作	CPU不执行任何操作，其机器码占用一字节
停机	HLT (Halt)	使CPU处于停机状态	只有外中断或复位信号才能退出停机，继续执行
等待	WAIT (Wait)	使CPU处于等待状态	等待TEST信号有效后，方可退出等待状态，继续执行
锁定前缀	LOCK (Lock)	使总线锁定信号有效	LOCK是一个单字节前缀，在其后的指令执行期间，维持总线的锁存信号直至该指令执行结束

如何查阅《附录1 80x86指令系统一览表》

助记符	汇编语言格式	功 能	操作数	时钟周期数	字节数	标志位 O	D	I	T	S	Z	A	P	C	备注
AAA	AAA	AL←把 AL 中的和调整到非压缩的 BCD 格式		3	1	u	-	-	-	u	u	x	u	x	
ADD	ADD dst, src	(dst) ← (dst) + (src)	reg, reg	1	2	x	-	-	-	x	x	x	x	x	
			reg, mem	2	2～7										
			mem, reg	3	2～7										
			reg, imm	1	3～6										
			ac, imm	1	2～5										
			mem, imm	3	3～11										
JCXZ	JCXZ opr	CX=0 则转移		6/5	2	-	-	-	-	-	-	-	-	-	
RET	RET	段内：IP ← POP( )		2	1	-	-	-	-	-	-	-	-	-	
		段间：IP ← POP( )		4*	1										
		CS ← POP( )													
IRET	IRET	IP ← POP( )		7*	1	r	r	r	r	r	r	r	r	r	
		CS ← POP( )													
		FLAGS ← POP( )													

* 实模式下的时钟周期数。在保护模式下，由于情况复杂，未提供

第四章　汇编语言程序格式

4.1 汇编程序功能

4.2 伪操作

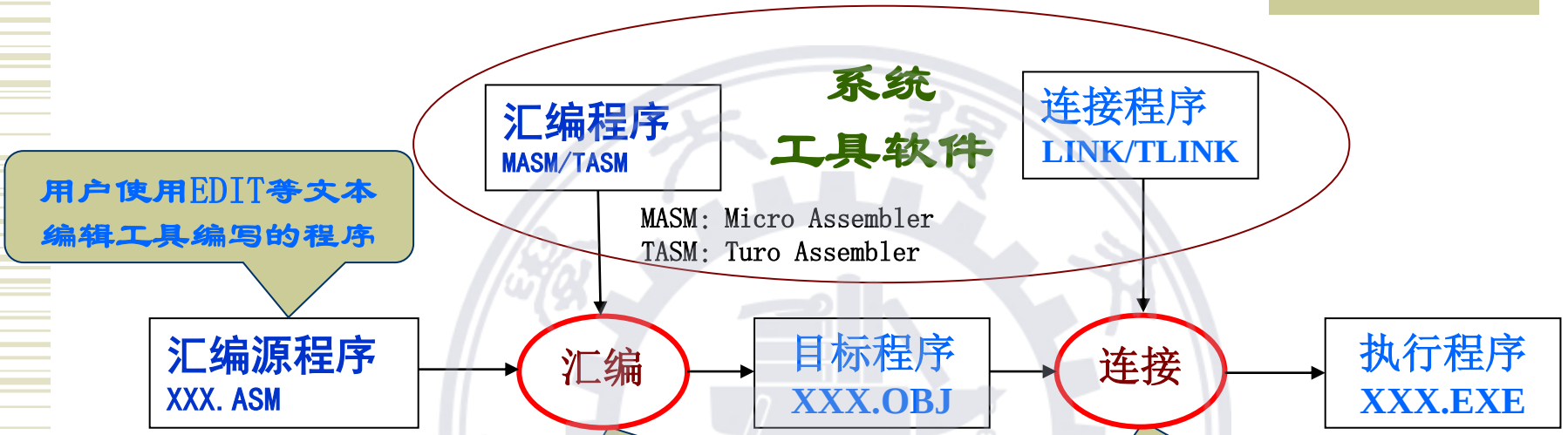
4.3 汇编语言程序格式

4.4 汇编语言程序上机过程

本章目标

- 掌握伪操作的种类、格式和应用
- 掌握汇编语言程序格式
- 熟悉汇编语言程序的上机过程

4.1 汇编语言程序的功能



汇编程序主要功能：

1、检查程序；

2、测出源程序中的语法错误，并给出错误信息；

3、展开宏指令；

4、产生源程序的机器语言目标程序，并给出列表文件，同时列出汇编语言和机器语言，XXX.LST

连接程序主要功能：

1、连接分别产生的目标模块

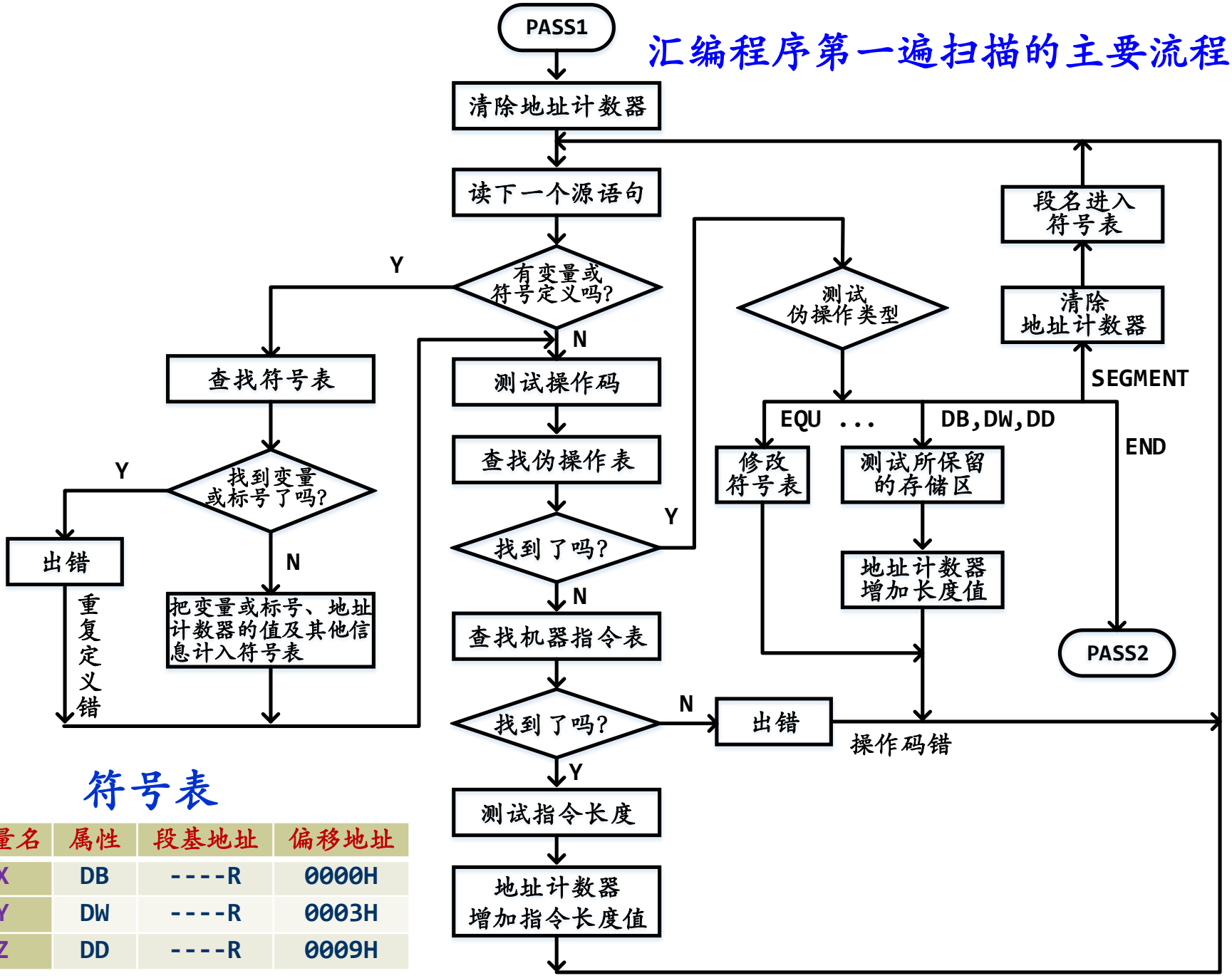
2、按要求进行段合并组合

3、解决外部交叉调用

4、产生可重定位的装入模块

5、产生可选的内存映象文件等

汇编程序第一遍扫描的主要流程



符号表

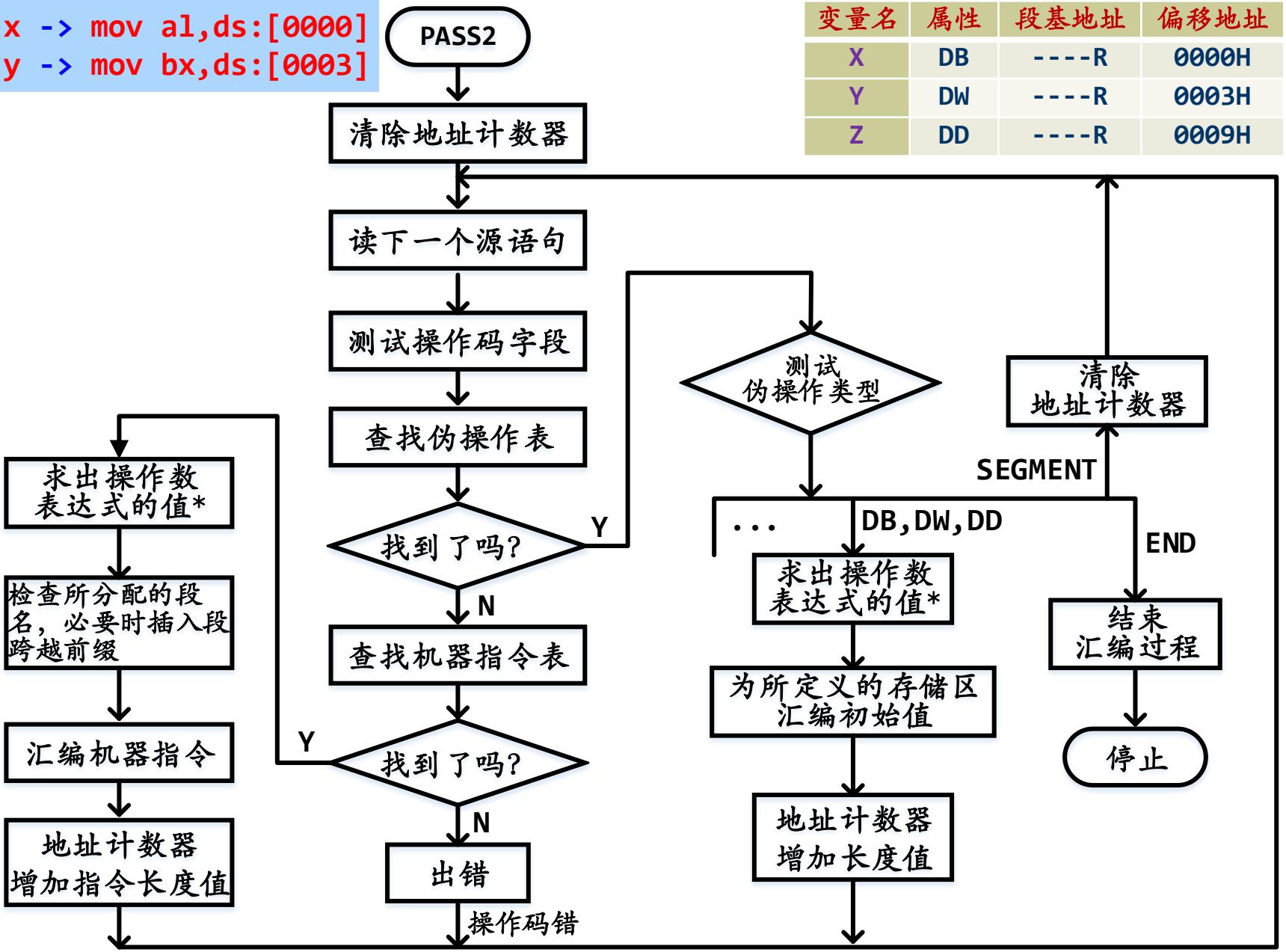
变量名	属性	段基地址	偏移地址
X	DB	----R	0000H
Y	DW	----R	0003H
Z	DD	----R	0009H

汇编程序第二遍扫描的主要流程

```
mov al,x -> mov al,ds:[0000]
mov bx,y -> mov bx,ds:[0003]
```

符号表

变量名	属性	段基地址	偏移地址
X	DB	----R	0000H
Y	DW	----R	0003H
Z	DD	----R	0009H



汇编语言指令

- 汇编语言程序的指令除机器指令以外还可以由伪指令和宏指令组成
  - 汇编指令包括：机器指令、伪指令、宏指令
- 机器指令：每条指令语句都生成机器代码，各对应一种CPU操作，在程序运行时由计算机执行
  - 第三章中汇编指令，每条对应一个机器指令
- 伪指令（又称为伪操作）：伪指令在汇编程序对源程序汇编期间仅由汇编程序按功能说明处理，以完成处理器选择、定义程序模式、定义数据、分配存储区、指示程序开始/结束等。
  - 这一章中的汇编语句
- 宏指令：自定义宏指令和标准宏指令
  - 第7章介绍，一条指令对应一组机器指令，汇编时展开

汇编语言指令分类

4.2 伪操作

4.2.1 处理器选择伪操作
4.2.2 段定义伪操作
4.2.3 程序开始和结束伪操作
4.2.4 数据定义及存储器分配伪操作
4.2.5 表达式赋值伪操作EQU
4.2.6 地址计数器对准伪操作
4.2.7 基数控制伪操作

伪操作：告诉汇编程序的某些功能说明或定义，仅在汇编时使用，不会汇编成任何机器指令

4.2.1 处理器选择伪操作

由于80X86的所有处理器都支持8086/8088系统，但每一种高档的机型又都增加了一些新的指令，因此在编写程序时要对所用指令集的处理器有一个确定的选择。也就是说，要告诉汇编程序：应该选择哪一种处理器的指令系统。

此类伪操作参看P135

缺省时默认选择8086指令系统

.8086	.486
.286	.486P
.286P	.586
.386	.586P
.386P	

4.2.2 段定义伪操作

格式：

```
segment_name    segment
                ...
segment_name    ends
```

```
data    segment
        X DB 10,4,10H
        Y DW 100,100H,-5
        Z DD 3*20,0FFFDH
data    ends
```

```
code_seg    segment
            ......
            mov   ss,  ax
            mov   sp,  offset   tos
            ......
code_seg    ends
```

- 当定义**数据段**、**附加段**和**堆栈段**时，在segment/ends伪指令中间的语句，只能包括伪指令语句
- 当定义**代码段**时，中间的语句才能为机器指令语句以及与机器指令有关的伪指令语句（宏指令）

Segment 伪操作还可以增加存储器分配组合方式类型

格式：

```
s_name   segment  [定位类型][组合类型][使用类型][类别]
           ...      [align_type] [combine_type] [use_type]  ['class']
s_name   ends
```

注意：s_name只是给段起了个有助于程序阅读的段名，不说明段的任何属性！

- **定位类型**：BYTE、WORD、DWORD、PARA、PAGE指定段在存储器分配时的对齐属性
  - PARA：段从小段地址开始（缺省默认） XX...XX0000
  - BYTE：段从任意字节开始 XX...XXXXXX
  - WORD：段从下一字地址开始，偶数地址开始 XX...XXXXX0
  - DWORD：段从下一双字地址开始，开始地址最低2位为0，4的倍数 XX...XXXX00
  - PAGE：段从下一页地址开始（256字节为一页） XX...XX00000000

Segment 伪操作还可以增加存储器分配组合方式类型

格式：

```
s_name   segment [定位类型][组合类型][使用类型][类别]
           …
s_name   ends
```

- **组合类型**： PRIVATE、PUBLIC、COMMON、AT、STACK、MEMORY，程序连接时段的合并方式
  - PRIVATE：该段为私有段，不与其他模块中的同名段合并（缺省默认）
  - PUBLIC：同名段连接成一个物理段，连接次序由连接命令指定
  - COMMON：同名段启始地址相同，重叠在一起形成一个段，可以覆盖，连接长度是各分段中的最大长度
  - AT expression：指定段地址，段地址是表达式的值，但不能指定代码段
  - STACK：该段运行时为堆栈的一部分，各堆栈段紧接，组成一个堆栈段
  - MEMORY：与PUBLIC同义，该段装入模块的最高地址

连接程序
如何对程序模块中的段合并

1. 多个模块组合时的连接情况

- 连接程序根据SEGMENT伪操作的组合类型对多个模块进行组合。例如：
  - PUBLIC：不同模块中的同名段在LINK时按指定的次序连接形成一个段。各段从小段开始定位情况下，各段之间可能有小于16字节的间隔
  - COMMON：不同模块中的同名段重叠形成一个段
  - STACK：不同模块中的同名段组合形成一个堆栈段，原有段之间无间隔，栈顶是这个大堆栈段的栈顶

例13.2：

源程序模块1

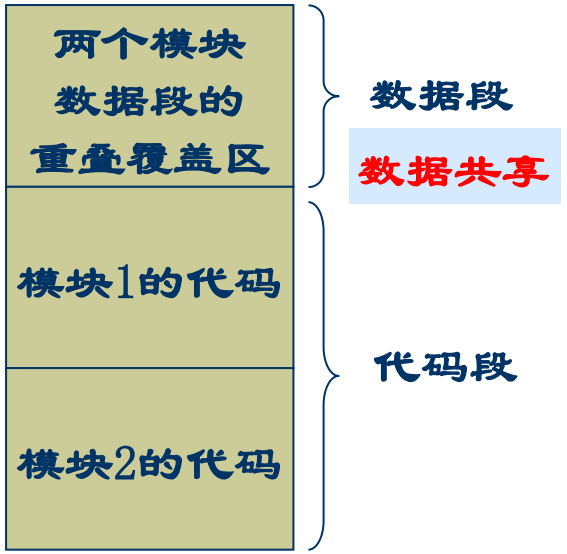
```
data   segment   common
       ...
data   ends
code   segment   public
       ...
code   ends
       end   start
```

源程序模块2

```
data   segment   common
       ...
data   ends
code   segment   public
       ...
code   ends
       end
```

注意：data、code段名助记符
只是有助于程序阅读的段名，
不说明段的任何属性！

连接后装入模块的
存储区分配情况



模块1和模块2的代码段组成一段。如果para，
模块1和模块2的代码从内存的小段位置放置

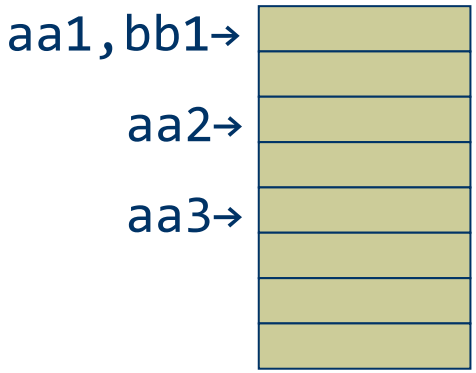
数据段覆盖组合定义

- 源程序模块1

```
data  segment  common
    aa1   dw ?
    aa2   dw ?
    aa3   dw ?
      ...
data  ends
code  segment  public
      ...
      mov aa1, ax
      ...
code  ends
      end  start
```

源程序模块2

```
data  segment  common
    bb1   dw ?
      ...
data  ends
code  segment  public
      ...
      mov ax, bb1
      ...
code  ends
      end
```



例13.3：

注意：stack_seg段名助记符
只是有助于程序阅读的段名，
不说明段的任何属性！

源程序模块1

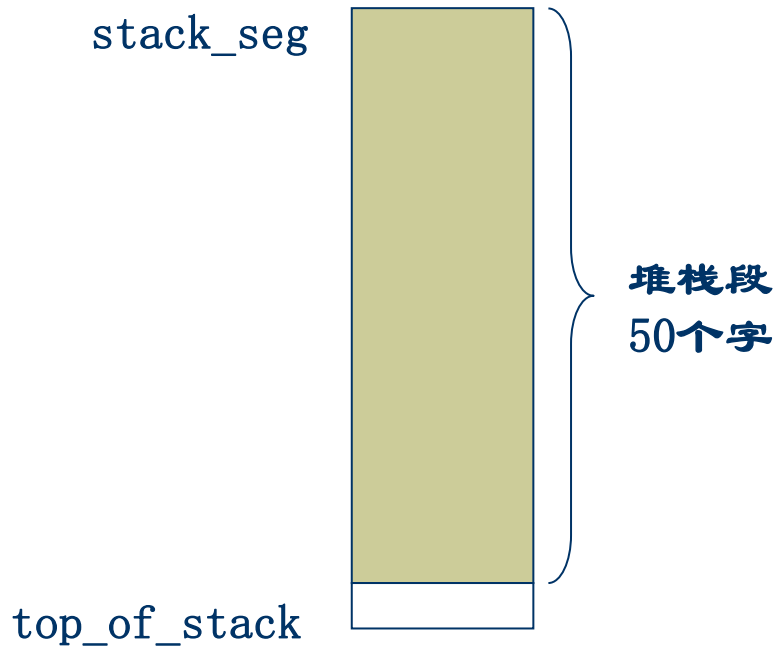
```
stack_seg   segment   stack
        dw   20 dup(?)
        top_of_stack   label   word
stack_seg   ends
```

源程序模块2

```
stack_seg   segment   stack
        dw   30 dup(?)
stack_seg   ends
```

模块1和模块2的堆栈段组成一段，之间没有留空

连接后的堆栈段
存储区分配情况



Segment 伪操作还可以增加存储器分配组合方式类型

格式：

```
s_name   segment [定位类型][组合类型][使用类型][类别]
           …
s_name   ends
```

- **使用类型**：386及后续机型，指定偏移量长度
  - USE16：16位寻址方式 （缺省默认），段长64KB
  - USE32：32位寻址方式，段长4GB
  - 实模式下，应该使用USE16

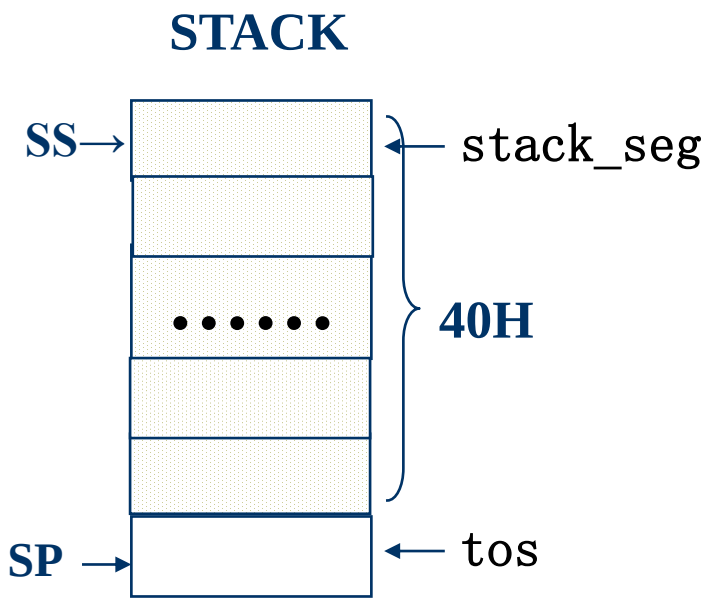
- **类别**：‘CLASS’，给出组成段组的类型名
  - 同类别的段装配在相邻的位置，组成段组
  - 如‘code’，‘data’,’bss’

例：定义用户堆栈

```
stack_seg   segment
            dw   40H   dup   (?)
    tos     label  word
stack_seg   ends

code_seg    segment
                ……
            mov  ax, stack_seg
            mov  ss, ax
            mov  sp, offset  tos
                ……
code_seg    ends
```

注意段寄存器设置方式

缺省：PARA PRIVATE USE16
分配内存时：从小段开始、私有段、16位偏移量寻址方式



例：

```
data_seg1  segment
           …
data_seg1  ends          ; 定义数据段

data_seg2  segment
           …
data_seg2  ends          ; 定义附加段

code_seg  segment
          assume  cs:code_seg, ds:data_seg1, es:data_seg2
start:
          mov  ax, data_seg1
          mov  ds, ax
          mov  ax, data_seg2
          mov  es, ax              ; 段地址→段寄存器
          …
code_seg  ends
               end start
```

ASSUME(约定)伪指令

- 明确段和段寄存器之间的关系
- 必须在代码段指明所定义的段与段寄存器的对应关系
- 格式： ASSUME 段寄存器名：段名
- 例如：

CS：**CSSEGNAME**

DS：**DSSEGNAME**

ES：**ESSEGNAME**

SS：**SSSEGNAME**

```
code_seg  segment
      assume cs:code_seg,ds:data_seg1,
             es:data_seg2
start:    ...
          mov  ax, data_seg2
          mov  es, ax
          ...
code_seg  ends
          end start
```

使用段寄存器前必须给出约定

◆应放在引用段寄存器之前，通常放在代码段或主过程的第一个语句位置

◆若一个段寄存器与NOTHING关联,则表示取消前边对该段寄存器的假设

DS:nothing ；取消原来段寄存器DS的预约分配

◆ASSUME语句并不给段寄存器赋值

课内测试 CH04-1

1. 请在填空中 [填空1] 填写“54” (10分)；

2. 汇编指令包括：机器指令、伪指令、宏指令。
   本章伪指令是：[填空2]和[填空3] (10分)

   A. 和机器指令一一对应，有CPU取指执行

   B. 汇编程序汇编时进行处理

   C. 连接程序连接时进行处理

说明：第2题从下拉列表中选择

存储模型与简化段定义伪操作

- 新版MASM5.0和MASM6.0另外提供了一种新的较简单的段定义方法
  - 不能提供较完整的表达段定义能力
  - 比较简单实用，是[定位类型][组合类型][使用类型][类别]组合的简单表示
  - 和高级语言连接也比较容易
  - 简化段定义包括：
    1. MODEL伪操作
       - 用MODEL定义存储模型时的段缺省属性（参见表4.1，P140）
    2. 简化的段定义伪操作
       - 与简化段定义有关的预定义符号

1. MODEL伪操作

格式：

.model memory-model[,model options]

- memory-model：根据代码段和数据段在存储器中的安放合并方式，可以建立7种存储模型
  - ① TINY：所有数据和代码都放在一个段内，数据和代码都是近访问
    - TINY程序可以写成 .com文件，com程序必须从0100H存储单元开始，一般用于小程序
  - ② SMALL：所有数据放在一个64KB的数据段内，所有代码放在另一个64KB的代码段内，都是近访问

```
.model  small
```

  - ③ MEDIUM：代码一个模块一个段，所有数据放在一个64KB的数据段内，数据近访问，代码远访问
  - ④ COMPACT：数据放在多个段内，所有代码放在一个64KB的代码段内
  - ⑤ LARGE：代码和数据都可以用多个段，数据和代码都可以远访问
  - ⑥ HUGE：代码和数据都可以用多个段，但允许数据段大小超过64KB
  - ⑦ FLAT：允许使用32位偏移量

格式：

```
.model  memory-model[,model options]
```

- model options：
  - ① **高级语言接口：** 该汇编语言程序作为高级语言程序的过程调用，如C, BASIC, FORTRAN, PASCAL等
  - ② **操作系统：** 说明运行于哪种操作系统环境下，如OS_DOS, OS_OS2 （缺省是OS_DOS）
  - ③ **堆栈距离：** NEARSTACK, FARSTACK
    - NEARSTACK：堆栈段和数据段组合到一个DGROUP段中，(DS)=(SS)
      - TINY、SMALL、MEDIUM、FLAT时缺省默认NEARSTACK
    - FARSTACK：堆栈段和数据段不合并
      - COMPACT、LARGE、HUGE时缺省默认FARSTACK

```
.model  large,c,os_dos,farstack
```

2. 简化的段定义伪操作

- 将数据段分得更详细
  - 常数段、初始化段、未初始化段、远初始化段、远未初始化段
  - 便于与高级语言兼容

```
.model tiny
.data
   x db 0ah,5
     …
.code
     …
```

- 格式：
  - .CODE [name] 代码段
  - .DATA 近初始化数据段
  - .DATA ? 近未初始化数据段
  - .FARDATA [name] 远初始化数据段，name缺省使用FAR_DATA
  - .FARDATA ? [name] 远初始化数据段，name缺省使用FAR_BSS
  - .CONST 常数段
  - .STACK [size] size缺省值为1KB
- 使用简化段伪操作时，必须在程序一开始先用.MODEL
- 每个段的定义开始就是上一段的结束，段结束符ENDS可以省略

与简化段定义有关的预定义符号

- @MODEL 用数值表示当前所用的存储模型

  TINY = 1
  SMALL 或 FLAT = 2
  MEDIUM = 3
  COMPACT = 4
  LARGE = 5
  HUGE = 6

```
.model small
  …
.code
  …
  mov al, @model   ;=mov al, 2
  …
```

- @code: 由.CODE伪操作定义的代码段段名或段组名

  mov ax, @code ;相当于 mov ax, 代码段名, 即 mov ax, 代码段基址

- @codesize: 以数值表示当前代码段情况
  - 只有一个代码段（TINY、SMALL、COMPACT 、FLAT），数值为0
  - 多个代码段（MEDIUM、LARGE、HUGE），数值为1

    mov ax, @codesize ;相当于 mov ax, 0或者1

- @data：由.DATA伪操作定义的数据段名，
  或由.DATA和.STACK等定义的数据段组名
- @datasize：以数值表示当前数据段情况
  - 只有一个数据段（TINY、SMALL、MEDIUM、FLAT），数值为0
  - 多个数据段（COMPACT、LARGE），数值为1
  - 多个数据段，且段大小超过64KB（COMPACT、LARGE、HUGE ），数值为2
- @fardata：由.FARDATA伪操作定义的段名
- @fardata？：由.FARDATA？伪操作定义的段名
- @curseg：当前段的段名
- @stack：堆栈段的段名或段组名
- @filecur：当前文件名（包括扩展名）
- @filename：当前文件名（不包括扩展名）
- @wordsize：表明段为16位还是32位的数值回送符
  - 16位时，回送2
  - 32位时，回送4
- 这些预定义符号可以在程序中被引用
  - 可以认为是一种段名省略定义时的默认名
- 例如：MOV AX, @DATA(未定义段名时)
  MOV DS, AX

代替段名，取段基地址

```
.data
  …
.code
  …
  mov ax, @data
  mov ds, ax
```

立即数寻址方式

```
MOV AX, data_seg1（定义了段名）
MOV DS, AX
```

- 用汇编语言编写程序可以使用两种基本格式。完整段定义、简化段定义

- 完整段定义格式虽然需要较复杂的语法，但它可以提供完整的控制，也是大多数汇编程序通用的定义格式，兼容性好

Why Even Bother With Segments?

As a beginning assembly language programmer, it's probably a good idea to ignore much of this discussion on segmentation until you are much more comfortable with 80x86 assembly language programming

Why Even Bother With Segments?

- Real-mode 64K segment limitation
- Program modularity
- Interfacing with high level languages

4.2.3 程序命名和结束伪操作

程序命名：

- NAME　　模块名　　；模块命名
- TITLE　　标题名　　；模块标题

程序结束

END　[执行的起始地址]

程序开始

- 模块命名
  - 格式：NAME　　module_name
    - module_name：模块的名字
- 模块标题
  - 格式：TITLE　text
    - 这样可在列表（LST）文件中，TITLE的标题名从第二页开始列出
    - text最多60个字符
- 如果没有NAME伪操作命令，将用text中的前6个字符作为模块名
- 程序中NAME、TITLE都没有
  - 用源文件名作为模块名
  - 但一般经常使用TITLE

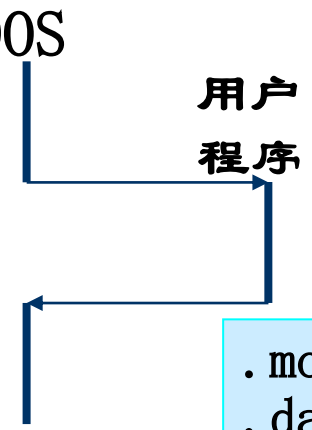
程序结束

格式：

END [label]

```
DATA    SEGMENT
        …
DATA    ENDS
CODE    SEGMENT
        ASSUME  CS:CODE , DS:DATA
START:  MOV     AX,  DATA
        MOV     DS , AX
        …
        MOV     AX, 4C00H
        INT     21H
CODE    ENDS
        END  START
```

- 标号（label）指示程序开始执行的启始地址
- [ ]中程序开始执行地址标号可选
- 只有主程序模块的END后可带label
- 若一个程序由多个模块组成，则除主程序模块外，其他模块的END语句不能带label



- MASM6.0增加了定义程序入口点和出口点的伪操作（应该归为宏指令）
  - .STARTUP
    - 定义程序的初始入口点
    - 汇编程序汇编时自动产生设置DS、SS和SP的代码
    - 这时END伪指令可以不指定程序开始执行地址标号
  - .EXIT [return_value]
    - 汇编程序汇编时自动产生退出程序并返回操作系统的代码
    - return_value返回给操作系统的值，常用 0

```
.model small
.data
  …
.code
.startup
  …
.exit 0
end
```

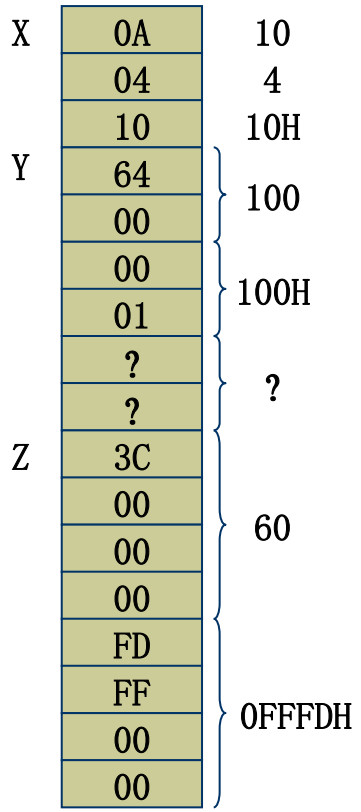
4.2.4 数据定义及存储器分配伪操作

格式： [Variable] Mnemonic Operand, ... , Operand [;Comments]

- 为变量(数据)分配存储单元，并为其初始化(赋值)或者只预留空间
- Variable:变量名，可有可无，是变量的符号地址，如果指令使用了变量名，表示(指向)第一个字节的偏移地址
- Mnemonic：伪操作助记符，是数据类型的符号表示

字节定义伪指令	DB
字定义伪指令	DW
双字定义伪指令	DD
四字定义伪指令	DQ
10个字节定义伪指令	DT

```
X DB 10,4,10H
Y DW 100,100H,?
Z DD 3*20,0FFFDH
```



- Operand：
  - 如给出具体数值常量、数值表达式、字符串常量、地址表达式，表示形成单元的初始化数据；
  - ？： 该单元内容未定，即未初始化数据
- Comments：注释，必须以“；”开始

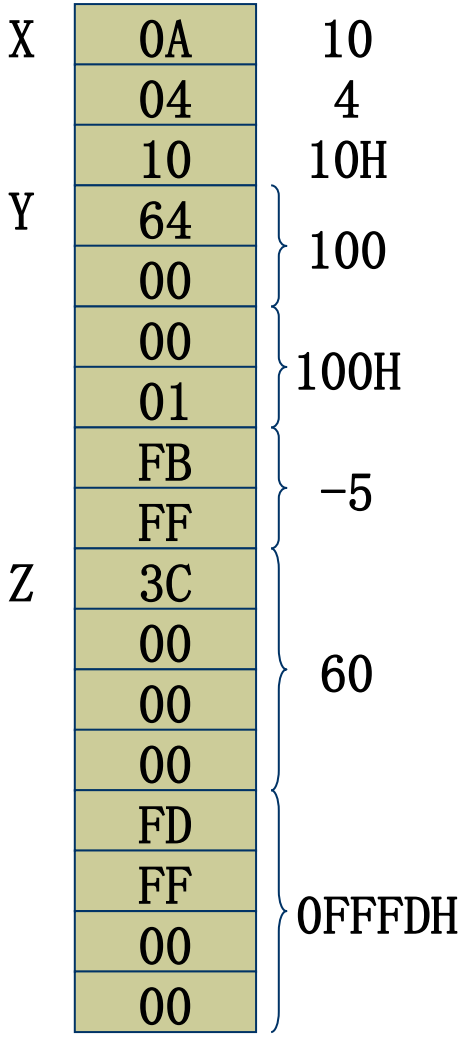
为数据分配存储单元，并初始化单元内容

完整段定义

```
data    segment
        X  DB  10,4,10H
        Y  DW  100,100H,-5
        Z  DD  3*20,0FFFDH
data    ends
```

简化段定义

```
.data
        X  DB  10,4,10H
        Y  DW  100,100H,-5
        Z  DD  3*20,0FFFDH
```



变量和标号属性：

所有的变量和标号都有三种属性

段基值(SEG)

偏移量(OFFSET)

类型(TYPE)：变量 (字节/字/双字/四字/十字节)

标号 (NEAR / FAR)

```
X DB 10,4,10H
NE: MOV AL, X
```

变量和标号作用是什么？

在汇编源程序中指明数据和指令的存储单元地址和可处理的方式

变量的类型属性总结：

(1) 指令中使用了变量名，表示的是该组数据的第一个字节的偏移地址

(2) 该操作中的每一个数据项

- DB伪操作　字节型　1字节
- DW伪操作　字型　2字节
- DD伪操作　双字型　4字节
- DF伪操作　6字节型　6字节
  - 存放远地址（16位段地址，32位偏移地址）
- DQ伪操作　四字型　8字节
- DT伪操作　10字节型　10字节

例 4.14

```
OPER1    DB    ?, ?
OPER2    DW    ?, ?
         ⋮
         MOV   OPER1, 0
         MOV   OPER2, 0
```

(3) 汇编程序可以用这种隐含的类型属性来确定某些指令是字指令还是字节指令

P148，例4.14

(4) 指令中PTR指定的变量类型属性优先于隐含的类型属性

格式：type PTR 变量名

- type可以是：BYTE WORD DWORD FWORD QWORD TBYTE
- 这样可使同一个变量具有不同的类型属性

```
Mov al, byte ptr yy
```

(5) 变量的另一种属性可以用LABEL伪操作来定义

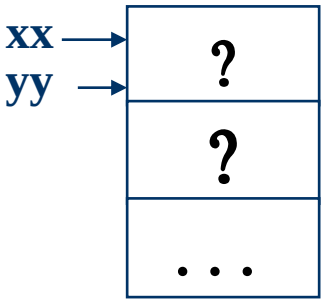
格式：name LABEL type

- 数据项类型type可以是：BYTE WORD DWORD FWORD QWORD TBYTE
- 对于可执行代码类型可以是：NEAR FAR

```
xx    LABEL   BYTE
yy    DW   50   DUP (?)
```

(6) 操作数字段可以使用复制操作符

repeat_count DUP(operand,...,operand)




```
BYTE_ARRAY   LABEL   BYTE
WORD_ARRAY   DW   25 DUP(?, 1234H)
```

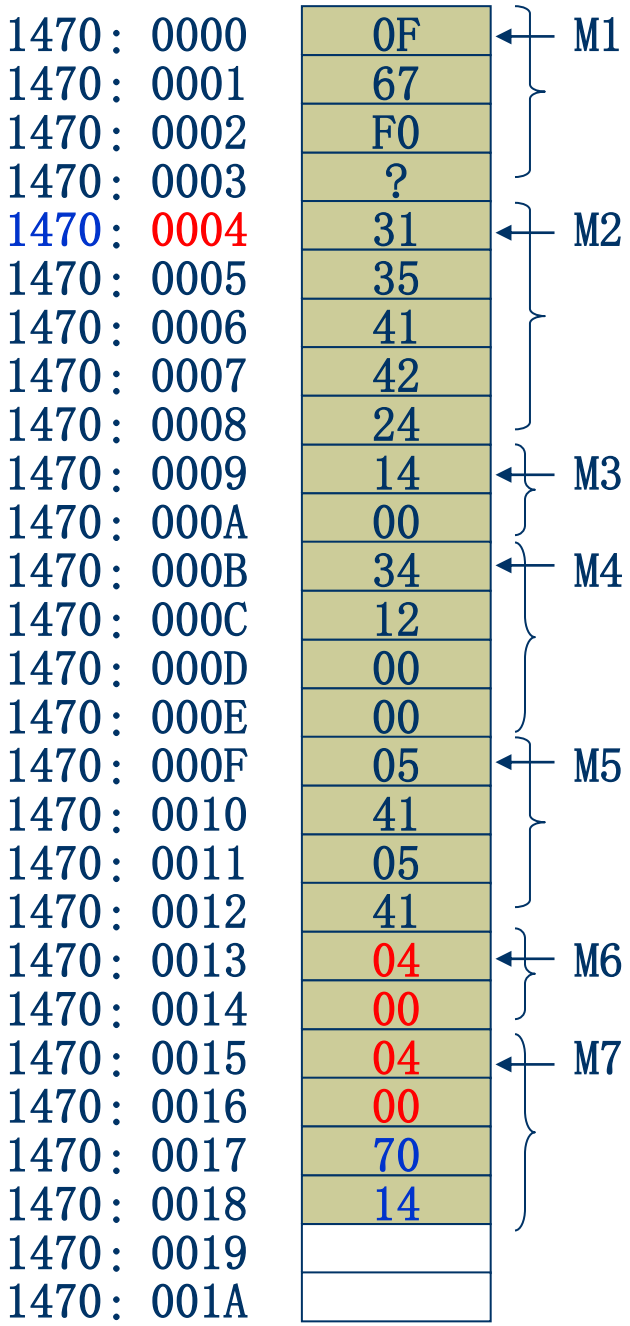
- 为数组分配100个存储单元，各单元字节内容：
  ?,?,34H,12H, ?,?,34H,12H,... ,?,?,34H,12H
- 指令中使用BYTE_ARRAY，以字节访问
- 指令中使用WORD_ARRAY，以字访问
- 有两个类型的变量名

```
MOV BYTE_ARRAY+2,0
;该数组第3个字节置0
MOV WORD_ARRAY+2,0
;该数组第3、4个字节置0
```

例.

```
data   segment
       M1  DB  15,67H,11110000B, ?
       M2  DB  '15','AB$'
       M3  DW  4*5
       M4  DD  1234H
       M5  DB  2 DUP(5,'A')
       M6  DW  M2      ;M2的偏移量
       M7  DD  M2      ;M2的偏移量、段基址
data   ends
```

注意：常数和字符串在存储器中的放置次序！



4.2.5 表达式赋值伪操作EQU

注意： 不分配占用存储单元，只是相当定义了一个常数的数值

格式：

有效的操作数格式
可求出常数值的表达式
任何有效的助记符

表达式名称 EQU 表达式

另外有一个与 equ 相类似的“=”赋值伪操作

格式：

表达式名 = 表达式

- 区别：“=” 伪操作中的表达式名允许重复定义

MOV AL, k 汇编后等同 MOV AL, 6

k=1	k EQU 1
k=k+5 允许	k EQU k+5 不允许

- 字符串、变址引用都可以赋以符号名 B EQU [BP+8]

MOV AL, B 汇编后等同 MOV AL, [BP+8]

```
DATAS SEGMENT
    var1        dw  22,33,44
    var2        db  10,22,33,44,55,10,77
    constant    equ 256
    alpha       equ 7
    beta        equ alpha-2
    adr         equ var2+5
DATAS ENDS

CODES SEGMENT
    ASSUME CS:CODES,DS:DATAS
START:
    MOV AX,DATAS
    MOV DS,AX

    mov al, adr

    MOV AH,4CH
    INT 21H
CODES ENDS
    END START
```

4.2.6 地址计数器对准伪操作

1、地址计数器

- 在汇编程序对源程序汇编过程中，使用地址计数器保存当前正在汇编的指令地址
  - 用在指令中：本条指令的第一个字节的地址
  - 用在参数中：地址计数器的当前值

- 指令中地址计数器的值用‘$’来表示，汇编语言允许用户直接用‘$’来引用地址计数器的值

```
MOV AX, $+6
```

例1： cs:0074 MOV AX, $+6

假设该指令的首地址偏移量是0074H。那么，地址计数器=0074H；汇编时，汇编程序会将$+6替换为0074+6=007AH

用在指令中

地址计数器的值

例2： ARRAY DW 1,2,$+4,3,4,$+4

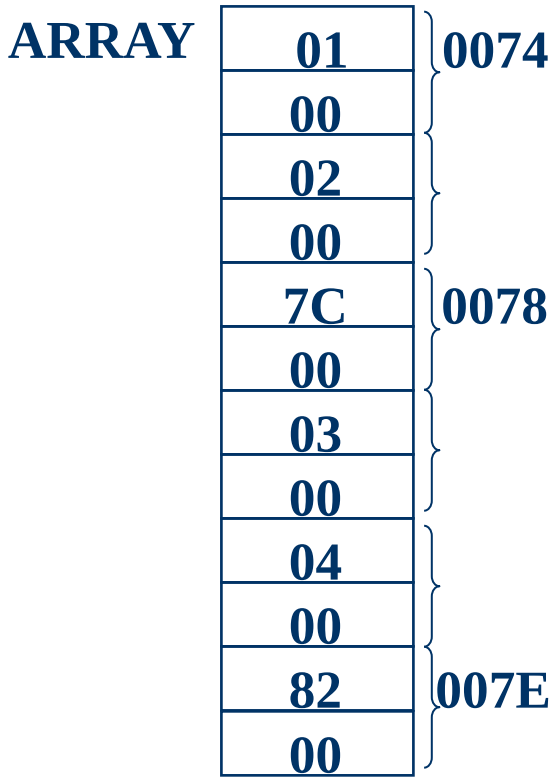
假设该数组的首地址0074H

① 汇编程序处理第一个$+4时，地址计数器=0078H

因此，$+4=0078+4=007CH

② 汇编程序处理第二个$+4时，地址计数器=007EH

因此，$+4=007E+4=0082H



用在参数中

2、ORG 伪操作

- 用来设置当前地址计数器的值，即分配后续数据、指令的存储器开始地址
  - 格式如下：

    ORG 常数表达式（n）
  - 功能：汇编时，使下一个操作数、指令等分配的存储器单元地址是常数表达式的值n
- 例如：

```
vectors     segment
            org     10          ; 10=000AH
vect1       dw      4567h       ; 偏移地址值为000AH
            org     20
vect2       dw      9876h       ; 偏移地址值为0014H
vectors     ends
```

3、EVEN 伪操作

- 格式：　EVEN
- 功能：使下一个变量或指令地址开始于偶字节地址

```
A   DB   'morning'
EVEN
B   DW   2   DUP   (?)
```

4、ALIGN 伪操作

- 格式：ALIGN　boundary
  - 其中boundary必须是2^n
- 保证双字数组边界从4的倍数地址存储单元开始
  - ALIGN 4
- 例子：

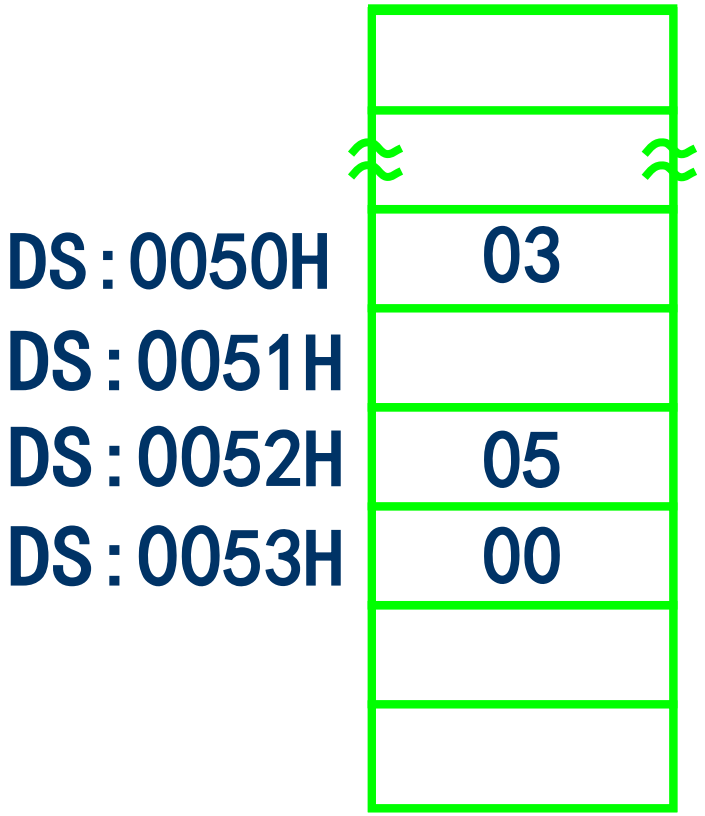
```
.DATA
   …
  ALIGN 4
ARRAY   DB   100 DUP(?)
   …
```

ARRAY的地址偏移量为4的倍数

ALIGN 2 = EVEN

例：

```
        ORG  50H
A1      DB   3
        EVEN
A2      DW   5
```



EVEN语句的效果

4.2.7 基数控制伪操作

- 汇编语言默认（不带后缀）的数为十进制数
- 在汇编语言中使用其他进制的数必须加以标记（二进制B，十六进制H等）

```
XX DB 13, 13H
```

RADIX 伪操作

格式：RADIX 表达式（n） ；表达式表示基数值，用十进制数表示

功能：用于改变汇编语言默认的基数（范围为：2~16）

例如：

```
    …
    RADIX   16
    MOV BL, 0FF
    MOV BX, 199D
```

如果199D是十六进制，后边必须带H

```
MOV BX, 199DH
```

课内测试 CH04-2

1. 请在填空中［填空1］填写“55”（15分）；

2. 根据程序中数据段定义，汇编程序汇编时为变量分配存储单元并初始为相应数值，变量对应存储单元偏移地址是：（15分）

   - M1偏移地址：［填空2］H
   - M2偏移地址：［填空3］H
   - M3偏移地址：［填空4］H

```
data   segment
       M1 DB 15,67H,“AB”
       M2 DW 3*5
       M3 LABEL BYTE
data   ends
```

变量名	段基地址	偏移地址
M1	1470	?
M2	1470	?
M3	1470	?

1470: 0000	0F
1470: 0001	67
1470: 0002	40
1470: 0003	41
1470: 0004	0F
1470: 0005	00
1470: 0006	?
1470: 0007	?

4.3 汇编语言指令格式

[名字项]	操作助记符	操作数	[；注释]
↓	↓	↓	↓
变量 标号	机器指令 伪指令 宏指令	寄存器 存储器 标号 变量 常数 表达式	说明程序 或语句 的功能

带〔 〕的两项是可缺省的

表达式：数字表达式，地址表达式

〔名字项〕　操作项　操作数项　〔；注释项〕

- **机器指令语句格式：**

  [标号：] 操作项 [操作数1 [，操作数2]] [；注释]

- **伪指令语句格式：**

  [变量] 伪操作项 [操作数1 [，操作数2]] [；注释]

〔名字项〕 操作项 操作数项 〔；注释项〕

4.3.1 名字项

- 名字项可以是指令标号或伪操作的变量、过程名、段名
- 由字母A~Z、数字和专用字符(?、.、@、-、$)组成，数字不能出现在名字第一个字符位置

〔名字项〕 操作项 操作数项 〔；注释项〕

1、标号

- 是用符号表示的指令地址，也叫符号地址
- 标号有3个属性：段地址 偏移地址 类型
  - 标号的段地址和偏移地址是指标号对应的指令首字节所在的段基地址和段内的偏移地址
  - 标号的类型属性有NEAR和FAR类型
- 标号的定义：直接在指令助记符前加上标识符，必须以冒号“：”结束，如 NEXT：

```
next: mov  ax, data_seg1
```

- 标号经常在转移指令或CALL指令的操作数字段出现，用以表示转向地址

2、变量

- 是数据项的第一个字节相对应的标识符
- 变量有3个属性：段地址 偏移地址 类型
  - 变量的段地址和偏移地址：是指变量对应的数据项首字节所在的段地址和段内的偏移地址
  - 变量类型属性：定义该变量所保留的字节数，如BYTE（1个字节长），WORD（2个字节长），DWORD（4个字节长）...
- 变量的定义：

  (1) 变量在数据段或附加段中定义，后面不跟冒号

  (2) 用LABEL、DB、DW、DD伪操作定义变量属性

  A DB ‘morning’
- 变量经常在操作数字段出现

4.3.2 操作项

- 操作项：给出操作的符号表示，可以是机器指令、伪指令、宏指令的操作功能助记符
  - 对于机器指令：汇编程序将其翻译为机器语言操作码
  - 对于伪指令：汇编程序将根据其要求的功能进行处理
  - 对于宏指令：宏汇编程序将根据其定义展开

4.3.3 操作数项

- 操作数项：由一个或多个表达式组成，提供操作项操作所需要的数据信息
- 操作数项可以是常数（立即数）/寄存器/存储器地址/标号/变量/表达式
- 每条指令语句的操作数个数已由系统确定
  - 例如加法指令有两个操作数

表达式

- 表达式是常数、寄存器、标号、变量和一些**操作符**相结合的序列
- 表达式有**数字表达式**和**地址表达式**
- 在汇编时，汇编程序按一定的优先顺序计算可得到一个数值或地址，替换源程序中的表达式

操作数有关的常用操作符

1、算术操作符

- 算术操作符有：+、 −、 *、 /和MOD

  其中MOD是指除法运算后得到的余数

```
ARRAY   DW 1,2,3,4,5,6,7
ARYEND  DW ?

        MOV DX,ARRAY+(6-1)*2
        ;执行后，6→DX

        MOV CX,(ARYEND-ARRAY)/2
        ;汇编后，MOV CX,7
```

汇编时将ARRAY认为地址偏移量来计算

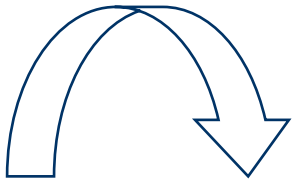
2、逻辑与移位操作符

- 逻辑操作符有：AND、 OR、 XOR、 NOT、
- 移位操作符有：SHL、 SHR

例：
```
OPR1   EQU   25    ; 00011001
OPR2   EQU   7     ; 00000111
```

```
00011001
00000111
--------
00000001=01H
```

汇编后等同



```
AND  AX,  OPR1 AND OPR2        AND  AX, 0001H
```

3、关系操作符

- 关系运算符有：EQ、NE、LT、GT、LE、GE
- 关系运算符的两个操作数是数字或同一段内的两个存储器地址
- 计算的结果应为逻辑值；结果为真，逻辑值=0ffffh；结果为假，逻辑值0000H

例：MOV FID, (OFFSET Y - OFFSET X) LE 128

```
X: ......

   ......

Y: ......
```

若≤128(真)　汇编后　MOV FID, OFFFFH

若>128(假)　汇编后　MOV FID, 0

4、数值回送操作符

- TYPE、 LENGTH、 SIZE、 OFFSET、 SEG等
- 这些操作符把一些特征或存储器地址的一部分作为数值回送

OFFSET / SEG 变量（或标号）

功能：回送变量或标号的偏址 / 段址

```
MOV   BX, OFFSET X
MOV   DX, SEG   X
```

TYPE 变量（或标号）

变量：DB DW DD DQ DT　　标号：NEAR FAR

值：1 2 4 8 10　　-1 -2

```
ARRAY   DW 1,2,3,4,5,6,7
        MOV BX,TYPE ARRAY
        ;汇编后，MOV BX,2
```

LENGTH 变量

功能：回送由DUP定义的变量的数据个数，其它情况回送1

SIZE 变量

功能：LENGTH*TYPE

```
MOV   BX, LENGTH ARRAY   ;MOV BX, 1
MOV   BX, SIZE ARRAY     ;MOV BX, 2
```

5、属性操作符

- PTR、SHORT、THIS、HIGH、LOW、HIGHWORD、LOWWORD等

  - PTR 操作符

    格式：类型 PTR 地址表达式

    功能：指定地址表达式的类型

```
MOV WORD PTR [BX], 5
;汇编后, MOV [BX],0005H
MOV BYTE PTR [BX], 5
;汇编后, MOV [BX], 05H
```

  - THIS 类型操作符

    格式：THIS 类型

    功能：为存储器操作数指定类型。该操作数地址与下一个存储单元具有相同的段基址和偏移量

```
TA  EQU  THIS  BYTE
TB  DW   100   DUP  (?)
```

```
NEXT  EQU  THIS  FAR
      MOV  CX, 100
```

  - SHORT 标号操作符

    用来修饰JMP指令中转向地址的属性，指出转向地址是在下一条指令地址的-128~+127字节范围之内

```
JMP  SHORT  NEXT
```

  - HIGH、LOW 字节分离操作符

    这两个操作符被称为字节分离操作符，它接收一个数字或地址表达式，HIGH取其高字节，LOW取其低字节

```
COUNT  EQU 1234H
       MOV AL, LOW COUNT
       ;汇编后, MOV AL, 34H
```

```
ARRAY  DW 1,2,3,4,5,6,7
       MOV AL, LOW ARRAY
       ;汇编后, MOV AL, BYTE PTR [ARRAY]
       MOV AL, HIGH ARRAY
       ;汇编后, MOV AL, BYTE PTR [ARRAY+1]
```

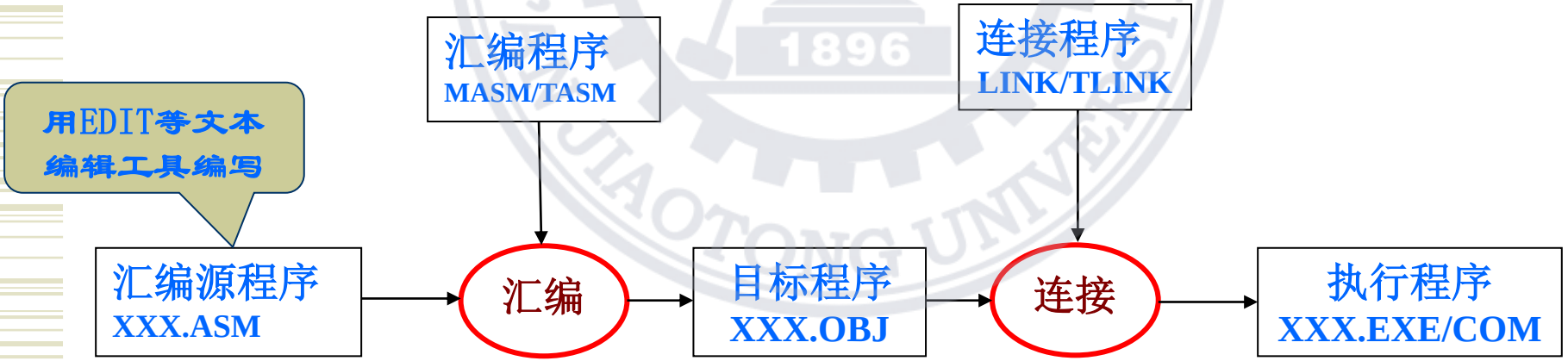
4.3.4 注释项

- 注释项用来说明一段程序、一条或几条指令在程序中的功能和作用等，尽量简单明了
- 格式：以“；”打头，回车结束
- 注释项可缺省
- 作用：使程序容易被读懂

```
MOV AL, LOW ARRAY ；将ARRAY字数组中第一个元素的低字节送累加器AL
```

4.4 汇编语言程序的上机过程

4.4.1 建立汇编语言的工作环境
4.4.2 建立ASM文件
4.4.3 用MASM(或TASM)程序产生OBJ文件
4.4.4 用LINK(或TLINK)程序产生EXE文件
4.4.5 程序的运行
4.4.6 COM文件



4.4.1 建立汇编语言的工作环境

建议新建一个文件夹设为D\ASM

- 微软公司的产品

```
EDIT.COM        ; DOS的文本编辑工具，用于编辑输入汇编语言源程序
MASM.EXE        ; 宏汇编器 (MASM---macro assembler)
LINK.EXE        ; 连接器
DEBUG.COM       ; DOS的动态调试器
```

- Borland公司的产品

```
EDIT.COM
TASM.EXE        ; 汇编器 (TASM---turbo assembler)
TLINK.EXE       ; 连接器
DEBUG.COM
```

在DOS平台上使用的较普遍的汇编器是MASM和TASM

4.4.2 建立ASM文件

第一步：编辑输入汇编语言源程序

方法： D\ASM>EDIT pname.asm↙

- 用编辑程序（如DOS6.2的EDIT）将汇编语言源程序输入计算机，经修改认为无误后，存入磁盘的文件系统
  - 例如，设源文件定名为pname.asm
  - 汇编语言源程序的后缀（扩展名）为“.asm”
  - P162 例4.30
- 若启动时带有文件名但该文件不存在，则启动后可以输入新文件，否则把已存在的文件调入编辑

例如：

D:\MASM>*EDIT* pname.*ASM*

则屏幕显示：

```
File Edit Search View Options Help
|--------- D:\masm\pname.ASM --------|
```

若EDIT是从Windows环境的MS-DOS方式进入的，则在DOS提示符后键入exit返回Windows

4.4.3 用MASM(或TASM)程序产生OBJ文件

第二步：汇编产生OBJ文件

D\ASM〉 MASM　pname ↙

调用MASM（或者TASM）汇编程序对源文件进行汇编

- MASM （TASM）也被称为汇编器，它用来把汇编语言源文件转换成目标模块pname.obj
- **如果汇编正确：** 则生成pname.obj文件，用dir命令可列出文件目录观看；反之，无法得到pname.obj文件。
- 如果在汇编过程中发现源程序有语法错误，则系统会输出“出错信息”，列出第几行有什么样的错误

- **汇编程序的基本功能**：是把用汇编语言书写的源程序**翻译**成机器语言的目标代码、**检查**用户源程序中的错误且显示出错信息、生成列表文件等
- 为了适应模块化程序要求，汇编后目标程序中的地址部分是**相对程序中各段起始位置的可浮动相对地址**，而不是可执行的存储器物理地址（绝对地址）

例如　D:\MASM>MASM pname;　↙

屏幕显示：

```
Microsoft (R) Macro Assembler Version 5.10
. . .
Source filename [pname.ASM]:
```

（汇编命令中没有输入源文件名时输入源文件名，不必输入扩展名）

```
Object filename [pname.OBJ]:
```

（要求回答目标文件名，可直接按Enter确认）

```
Source listing  [NUL.LST]: pname
```

（列表文件名，需要时输入名字部分，缺省情况不生成）

```
Cross-reference [NUL.CRF]:
```

（交叉引用文件名，需要时输入名字部分，缺省情况不生成）

```
51058 + 421678 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

- 也可以用命令行的形式按顺序对四个提示予以回答，格式是：

  MASM　源文件名,目标文件名,列表文件名,交叉引用文件名

  - 若只想对部分提示给出回答，则在相应位置用逗号隔开，若不想对剩余部分作答，则用分号结束。例如以下命令行与前边的分行回答等效：

    D:\MASM>MASM pname, ,pname;

- 另外如果需要得到列表文件pname.lst，可按如下方法进行汇编：

  D\ASM>MASM　pname.asm/l

- 可产生的3个文件：
  - 目标文件(.OBJ)
  - 列表文件(.LST)：同时列出源程序和机器语言程序清单，并给出符号表，可使程序调试更加方便
    - 源程序和机器语言程序清单

```
偏移量  目标码                    汇编格式
  ......
0000    1E                  PUSH DS      ;save old data segment
  ......
```

    - 段名表：段名、段大小、属性

```
Segments and Groups:
Name     Length   Align   Combine Class
CODE ..... 001D    PARA    NONE
DATA ..... 0028    PARA    NONE
EXTRA ... 0028     PARA    STACK
```

    - 符号表：用户定义的符号名、类型、属性
  - 交叉引用文件(.CFER)：

    给出用户定义的所有符号，每个符号列出符号所在行号和被引用行号，这样程序修改时就比较方便

    - P166

```
Microsoft Cross-Reference Version 5.00          Wed Mar 04 00:34:07 1998

  Symbol Cross-Reference      (# definition, + modification) Cref-1

CODE ....................     17#     21      50
DATA ....................      3#      8      21      31
DEST_BUFFER..............     11#     41
EXTRA....................     10#     15      21      35
MAIN ....................     19#     48
SOURCE_BUFFER............      4#     39
START....................     23#     53

7 Symbols
```

- 汇编后目标程序中的地址部分是相对段起始位置的**可浮动相对地址**，而不是可执行的物理地址（绝对地址）
- 由于程序的模块化设计、库函数调用等，一个程序可能由多个OBJ程序组成

因此，需要按各模块之间关系（各模块存储模型定义）合并，生成可重新定位的统一分配内存地址空间的可执行文件，这样要用连接程序产生.exe文件

4.4.4 用LINK(或TLINK)程序产生EXE文件

第三步：连接程序产生EXE文件

D\ASM>LINK pname；↙

- 程序被汇编通过后，需要经过连接才能生成可执行程序
  - 只有正确的得到obj文件，才能进行连接操作
- 连接程序的功能
  - 将目标程序和库函数或其它目标程序连接生成一个按存储模型要求分配内存地址的可执行的目标程序
    - 连接分别产生的目标模块
    - 解决外部交叉调用
    - 产生一个可重定位的装入模块
  - 产生可选的内存映象文件等

例如：

D:\MASM>*LINK*

Object Modules [.OBJ]: pname

（输入由汇编产生的.OBJ目标文件名）

Run File [pname.EXE]:

（直接回车确认系统给出的默认可执行文件名，也可改为其他名字）

List File [NUL.MAP]: pname

（输入内存映象文件名，可缺省不产生，直接按回车键）

Libraries [.LIB]:

（程序用到的库文件，如无特殊需求，直接按回车键）

- pname.OBJ经连接后在当前目录下产生了pname.EXE和pname.MAP文件
  - 可执行文件（.EXE）：操作系统可装入、动态重新分配内存空间的执行程序
  - 内存映象文件（.MAP）：给出每个段在存储器中的分配情况
    - pname.MAP文件的内容为：

```
 Start   Stop   Length  Name  Class
00000H  00027H  00028H  DATA
00030H  00057H  00028H  EXTRA
00060H  0007CH  0001DH  CODE
Program entry point at 0006:0000
```

      - 00060H=00060+00000
      - 从小段的开始地址分配段基地址
- 程序装入时，动态分配实际的物理段基地址，设置段寄存器内容
  - 如P162：程序段基址0006变成了08FE；根据MAP文件，数据段基址0000应该是08FE-0006=08F8

进一步说明的问题

1、“向前引用”、“向后引用”

- 向前引用：出现在操作数字段的变量或标号是未定义过的
- 向后引用：出现在操作数字段的变量或标号是已经定义过的
- 向前引用时，由于指令长度和操作数类型有关，而操作数类型和变量或标号类型有关，汇编程序第一遍扫描时就难以确定偏移地址量，因此向前引用时指令中应该明确说明操作数类型。如 JMP NEAR PTR EXIT

主要看汇编程序扫描到这里时能否确定变量或标号的属性

```
YYY   DW   1000H
            ……
            MOV   AX,   YYY
            MOV   AL, byte ptr YYY
            MOV   AX, word ptr XXX
            ……
XXX    DW 1000H
```

```
            JMP    NEAR PTR EXIT
            ……
EXIT:
```

让汇编程序能正确计算指令长度，形成正确的地址计数器值(即下条指令的偏移地址)，另符号表中符号的偏移地址才能正确形成

2、“浮动”地址概念

- 汇编时，段内所有偏移地址均为相对于本段起始地址的相对地址
- 连接时，多个段可能合并为一个段，才可确定偏移地址
- 装入时段的起始地址才可以确定
- 因此，段的起始地址和偏移地址要在0地址的基础上“浮动”一个值，在连接时才能确定
- 汇编程序确定的指令字中的变量和标号偏移地址值为浮动值（相对地址）

```
 1                      name  ch4crf
 2                      title demo how to use LST and CRF
 3 0000                     data segment common
 4 0000  11                     value db 11h
 5 0001  AABB                   aa    dw 0aabbh
 6 0003                     data ends
 7
 8 0000                     code segment public
 9                          assume cs:code, ds:data
10 0000                     start:
11 0000  B8 ---- R              mov   ax, data
12 0003  8E D8                  mov   ds, ax
13
14 0005  A0 0000 R              mov   al, value
15 0008  8B 1E 0001 R           mov   bx, aa
16
17 000C  B8 4C00                mov   ax, 4c00h
18 000F  CD 21                  int 21h
19 0011                    code ends
20                         end start
```

装入后确定

连接段组合后确定，可能段合并

浮动地址在LIST清单中标记为R，连接程序连接时再确定正确的偏移地址，或程序装入时再确定段基地址

4.4.5 程序的运行

第四步：运行程序

D:\ASM>pname ↙ 或 pname.exe ↙

- 经过前三步，在磁盘上生成了可执行文件pname.exe。在DOS状态下，直接键入文件名，DOS的装入程序就将该程序从磁盘装入内存并开始运行

DOS装入.EXE文件的过程

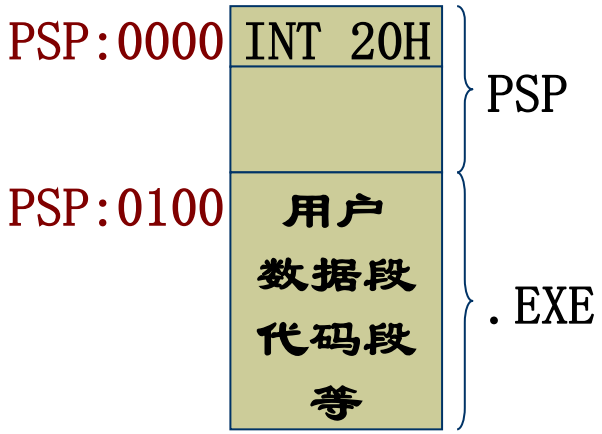
① DOS的装入程序为.EXE程序建立一个256字节的程序段前缀PSP (Program Segment Prefix)，PSP中包含可以被用户程序使用的DOS入口、DOS为自己所存储的信息、由DOS传递给用户程序的信息等。其中 PSP:0处存放一条INT 20H指令

② 把文件头读入内存

③ 计算可执行模块的大小

④ 计算装入的起始段地址

⑤ 完成重定位



⑥ 初始化段寄存器和指针寄存器。

- 装入程序对段和指针寄存器设置为：CS:IP为主程序的入口地址(程序装入后执行的第一条指令地址)
- SS：SP 设置为 PSP:0100
- 其他段寄存器全部被初始化为指向PSP的段基址，以便用户能够访问PSP中的信息

⑦ 把控制权交给.EXE程序。

- **需要使DS指向用户程序的数据段**

从装入程序对段寄存器的初始化可看到，它并没有把DS和ES（386以上还有FS、GS）指向用户自己的数据区，**而是指向了PSP的段基址**，这主要是方便用户程序通过DS等段寄存器访问PSP中信息。但在用户程序运行过程中，DS应指向程序自己的数据段以便访问其中内容，为此，应在程序中用指令为DS等段寄存器赋值。

如何返回DOS

方法1、

```
MAIN   PROC   FAR
       ASSUME  ...
       PUSH   DS                ;PSP段基址入栈
       XOR    AX,AX             ;清0
       PUSH   AX                ;数字0入栈
       ...                      ;完成程序指定功能
       RET                      ; PSP:0 送 CS:IP
```

- MAIN过程是FAR型，执行RET时从栈中弹出0给IP，再弹出一个字(PSP段基址)给CS，现在CS:IP指向PSP:0处的指令INT 20H
  - INT 20H指令功能：退出应用程序，释放所占内存并返回DOS。调用时要求CS指向PSP段基址。
- **是一种传统返回DOS的方法，对所有DOS版本均适用**

- **方法2、**
  - 在DOS较高版本中，推荐使用4CH系统功能调用返回DOS，这种方法实现起来比较简单
  - 方法：功能号4CH→AH寄存器，返回码00→AL，正常返回时返回码为0

```
CODE  SEGMENT
MAIN  PROC    FAR
              ASSUME CS:CODE , DS:DATA
              ASSUME SS:STASG
                ...
              MOV    SUM,AX
              MOV    AX , 4C00H
              INT    21H
MAIN  ENDP
CODE  ENDS
              END    MAIN
```

- 若想看内存中的结果，请借助动态调试软件DEBUG/TD

- 若执行结果有错误，请借助动态调试软件DEBUG/TD

DEBUG简介

- 每个版本的DOS都带有动态调试器DEBUG。原因是DEBUG不仅是动态调试器，也是二进制文件编辑器，还是简单的系统维护工具。DEBUG能提供一个动态调试程序的环境，程序员利用这个环境，可以方便的调试目标代码程序。

  例如：该软件提供逐条跟踪执行程序的命令，并且每一条指令执行后自动显示CPU内部所有寄存器和所有标志位的内容。逐条动态调试直到通过为止。

- DEBUG常用命令：

  u、t、p、g、d、r、q

(1)反汇编命令u

格式： u/u cs:偏移量

例如：

```
-u
1307:0000 1E              PUSH    DS
1307:0001 33C0            XOR     AX,AX
1307:0003 50              PUSH    AX
1307:0004 B80613          MOV     AX,1306
1307:0007 8ED8            MOV     DS,AX
1307:0009 A10000          MOV     AX,[0000]
1307:000C 03060200        ADD     AX,[0002]
1307:0010 A30400          MOV     [0004],AX
1307:0013 CB              RETF
```

(2)跟踪执行命令t

格式：t/t＝cs：偏移量

例如：

```
-t
AX=0243  BX=0000  CX=0064  DX=0000  SP=003C
  BP=0000  SI=0000  DI=0000  DS=1306  ES=12F2
  SS=1302  CS=1307  IP=0010  NV UP EI PL NZ AC
  PO NC
1307:0010 A30400  MOV  [0004],AX
  DS:0004=0000
```

(3)过程执行命令p

p (CALL REP LOOP INT等)

(4) 运行命令 g

格式：g程序断点

例如：

```
-g9
AX=1306  BX=0000  CX=0064  DX=0000      SP=003C  BP=0000
  SI=0000  DI=0000
DS=1306  ES=12F2  SS=1302  CS=1307      IP=0009  NV UP EI
  PL ZR NA PE NC
1307:0009 A10000 MOV AX,[0000] DS:0000=007B
```

(5) 显示存储单元内容命令 d

格式：d 段地址：偏移地址

例如：

```
-d ds:0 f
1306:0000  7B 00 C8 01 00 00 00 00-00 00 00 00 00 00 00 00
```

（6）寄存器显示/修改命令r

格式：r 寄存器名称

例如：

```
-r
AX=0000  BX=0000  CX=0064  DX=0000      SP=0040  BP=0000
  SI=0000  DI=0000        DS=12F2  ES=12F2  SS=1302  CS=1307
    IP=0000  NV UP EI PL NZ NA PO NC
1307:0000 1E              PUSH    DS
```

标志位值的符号表示：

标志位	OF	DF	IF	SF	ZF	AF	PF	CF
(=1)	OV	DN	EI	NG	ZR	AC	PE	CY
(=0)	NV	UP	DI	PL	NZ	NA	PO	NC

（7）存储单元修改命令e

格式： e 段地址：偏移地址

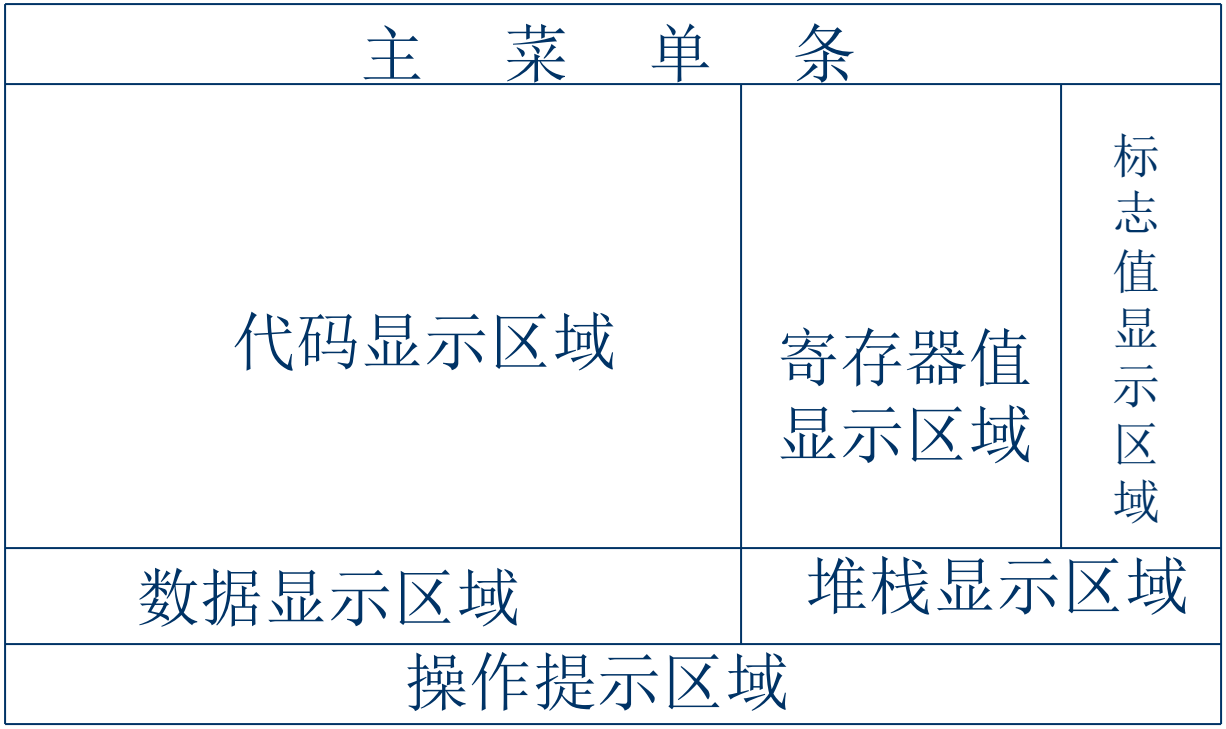
（8）debug退出命令q

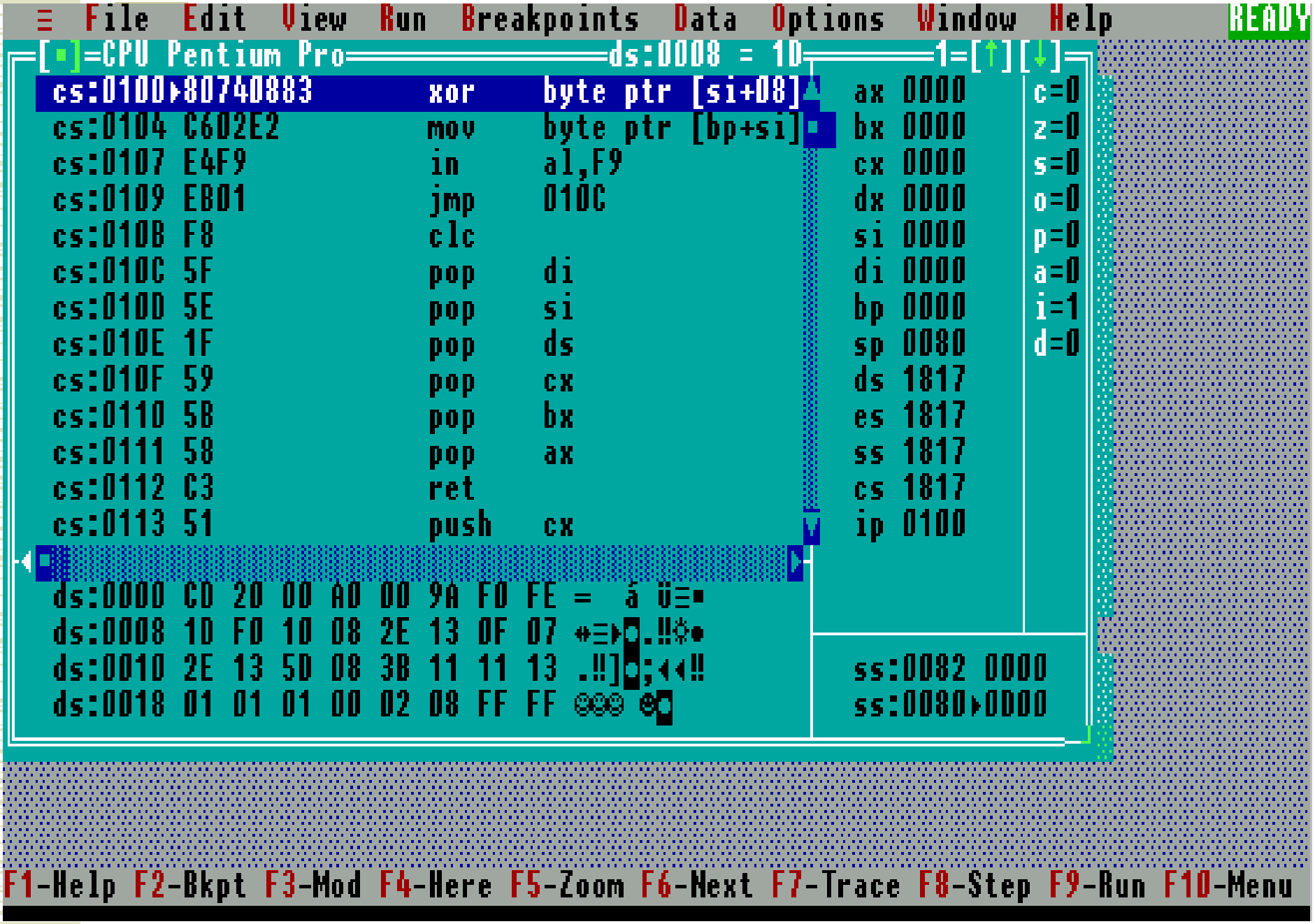
格式：q

Turbo debuger

- TD（Turbo debuger）是一个源代码调试器，它可以调试多种语言写成的程序。TD是重叠式窗口、下垃式和弹出式菜单以及鼠标器支持等，为用户提供了一个快速、方便和交互式环境。此外，联机帮助还可以在操作的的每个阶段提供相关的帮助。

TD的机器指令级调试界面



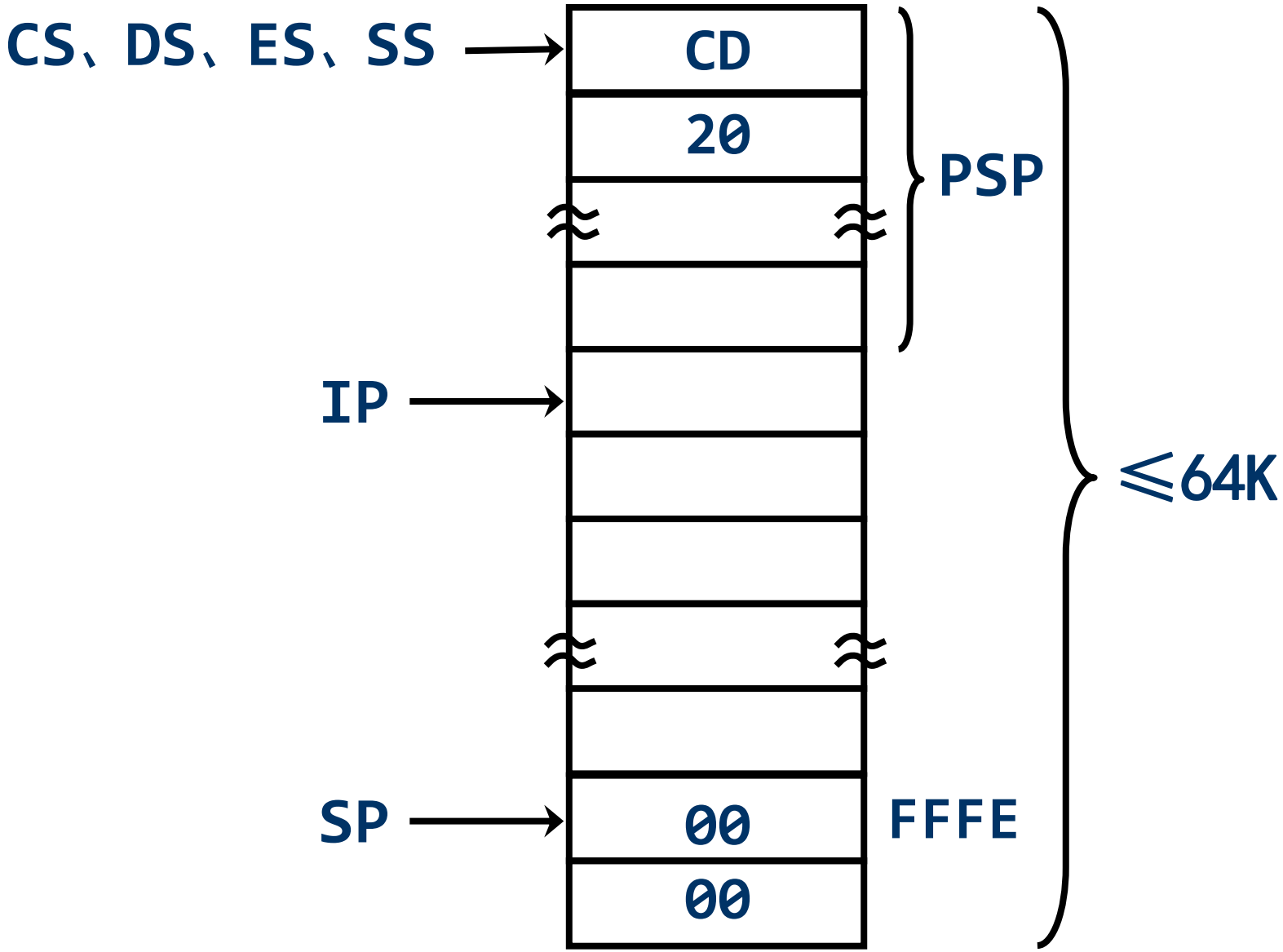

File Edit View Run Breakpoints Data Options Window Help
READY
CPU Pentium Pro
ds:0008 = 10
cs:0100 80740883 xor byte ptr [si+08]
cs:0104 C602E2 mov byte ptr [bp+si]
cs:0107 E4F9 in al,F9
cs:0109 EB01 jmp 010C
cs:010B F8 clc
cs:010C 5F pop di
cs:010D 5E pop si
cs:010E 1F pop ds
cs:010F 59 pop cx
cs:0110 5B pop bx
cs:0111 58 pop ax
cs:0112 C3 ret
cs:0113 51 push cx
ax 0000 c=0
bx 0000 z=0
cx 0000 s=0
dx 0000 o=0
si 0000 p=0
di 0000 a=0
bp 0000 i=1
sp 0080 d=0
ds 1817
es 1817
ss 1817
cs 1817
ip 0100
ds:0000 CD 20 00 A0 00 9A F0 FE
ds:0008 10 F0 10 08 2E 13 0F 07
ds:0010 2E 13 5D 08 3B 11 11 13
ds:0018 01 01 01 00 02 08 FF FF
ss:0082 0000
ss:0080 0000
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

4.4.6 .COM文件

- 是一个可执行文件
- 程序(指令代码和数据)不分段(只有一个段),它所占有的空间不允许超过64K
- 入口点必须是100H
- 程序装入时系统自动把SP建立在该段之末
- 占用内存更少，速度更快，因此适合编制较小的程序，例如DOS的外部命令SYS、FORMAT等都是.COM结构

- 程序段前缀PSP段长为100H字节，PSP:0处存放一条INT 20H指令。程序的二进制代码紧跟PSP之后装入

- .COM程序的代码、数据及堆栈数据在同一段中，所以对所有的段寄存器都初始化为指向PSP的段基址。IP＝100H,为PSP之后的下一个地址偏移量，SP指向栈顶,栈顶中存入一个字型数字0，如下图所示



.COM文件装入内存示意图

```
cseg1   segment
        org   100h
        assume cs:cseg1,ds:cseg1
        assume es:cseg1,ss:cseg1
start   proc    near
        push    cs
        pop     ds
        mov     dx,offset str1
        mov     ah,9
        int     21h
        mov     ah,4ch
        int     21h
start   endp
str1    db      0dh,0ah,'COM  file has only '
        db      'one segment',0dh,0ah,'$'
cseg1   ends
        end     start
```

COM文件的上机步骤

- 编辑输入源程序 EDIT FILEN.ASM
- 汇编源程序 MASM FILEN;
- 连接源程序 LINK/T FILEN;
- 转换: exe2bin FILEN.EXE FILEN.COM
- 删除.exe文件: del FILEN.exe
- 执行.com文件: FILEN
- 调试: DEBUG FILEN.COM

  或者TD FILEN

习　　题

4.1	4.2	4.3
4.4	4.5	4.7
4.9	4.10	4.12
4.13	4.14	4.21

第五章　循环与分支程序

5.1　循环程序设计

5.2　分支程序设计

5.3　如何在实模式下发挥 80386 及其后继机型的优势

本章目标

- 掌握汇编语言程序设计的基本步骤
- 熟练掌握顺序、分支和循环程序设计方法
- 掌握汇编语言程序常用的几种退出方法
- 掌握DOS系统功能调用

汇编语言程序设计的基本步骤

1. **分析问题** 根据实际任务（问题）确定任务的数据结构、处理的数学模型或逻辑模型；

2. **确定算法** 确定所要解决问题的适当算法，既处理步骤，如何解决问题，完成任务；

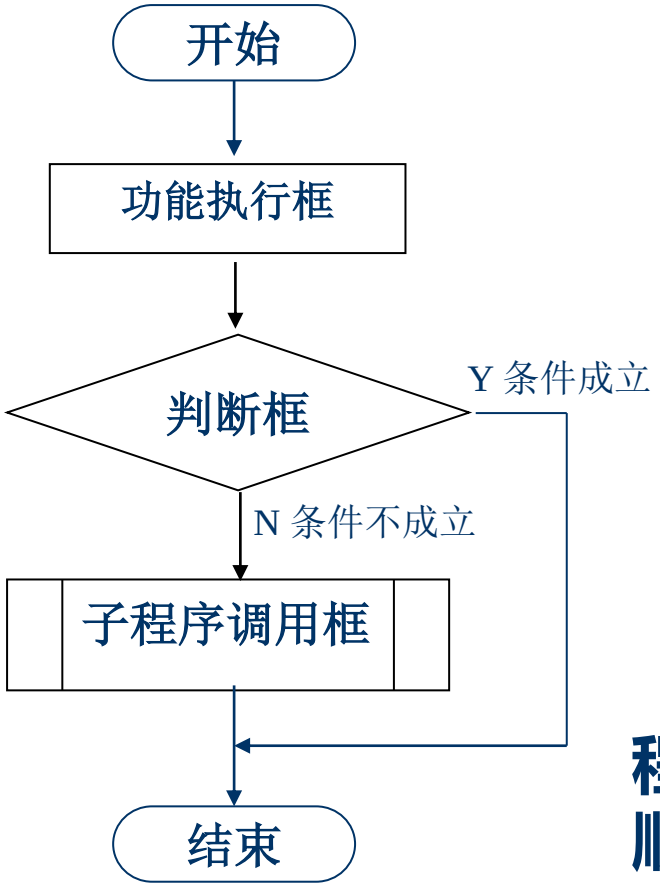
3. **绘制流程图** 设计整个程序处理的逻辑结构，从粗流程到细流程；

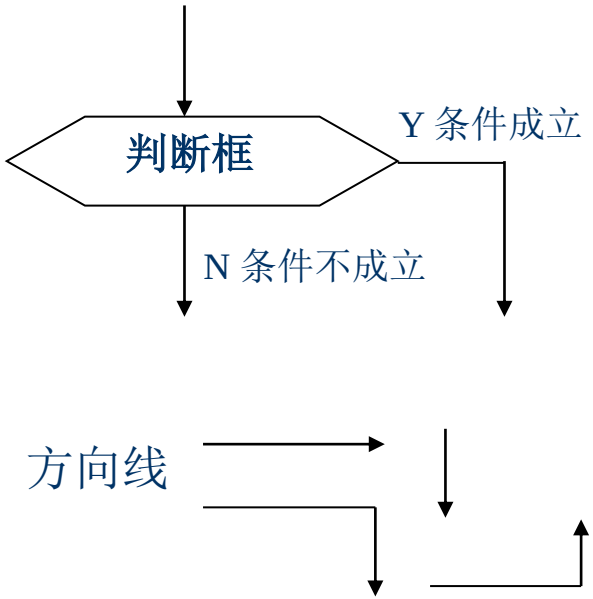
4. **存储空间分配** 分配数据段、堆栈段和代码段的存储空间，分配工作单元，借助数据段、堆栈段和代码段定义的伪操作实现；

5. **编写汇编语言源程序** 正确运用80X86CPU提供的指令、伪操作、宏指令以及DOS、BIOS功能调用，同时给出简明的注释；

6. **上机调试** 静态检查后，上机动态调试程序。

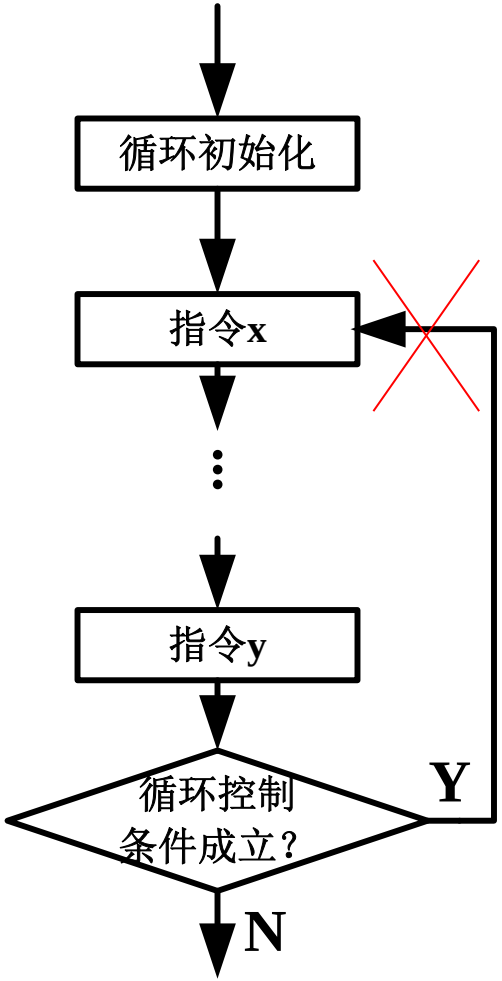
程序流程图画法规定

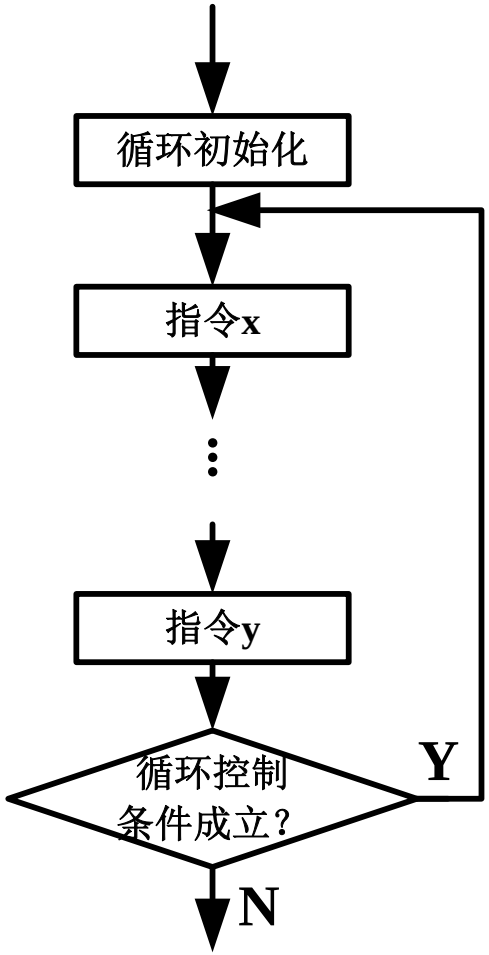




程序结构形式：
顺序、循环、分支和子程序

注意不正确画法

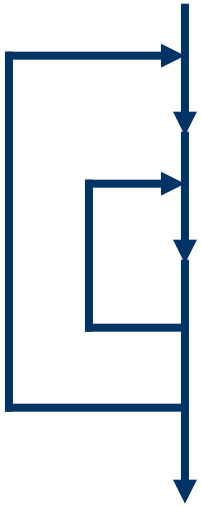




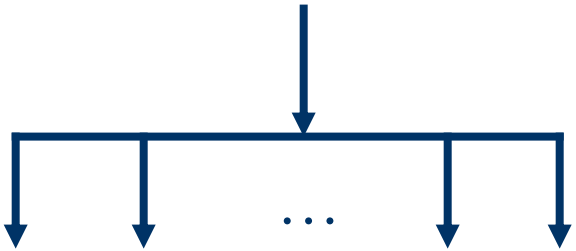
程序结构

顺序结构

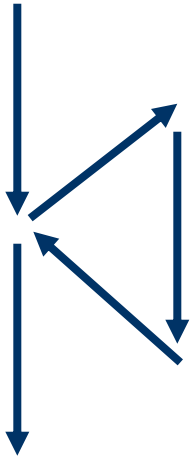
循环结构



分支结构



子程序结构



复合结构：多种程序结构的组合

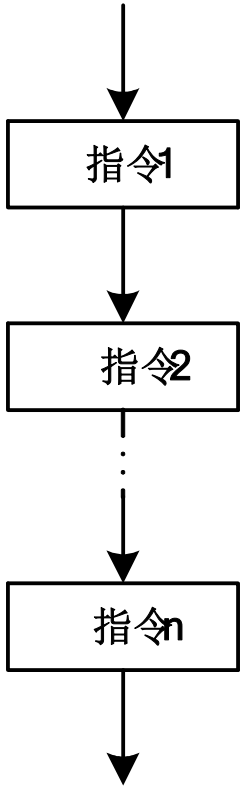
序程序设计方法

- 程序的执行顺序是从程序的第一条可执行指令开始执行，按照程序编写安排的顺序逐条执行指令直到最后一条指令为止

- 顺序程序结构所能解决的问题一般属于简单的顺序性处理问题

  如求：Y=（2*X+4*Y）*Z

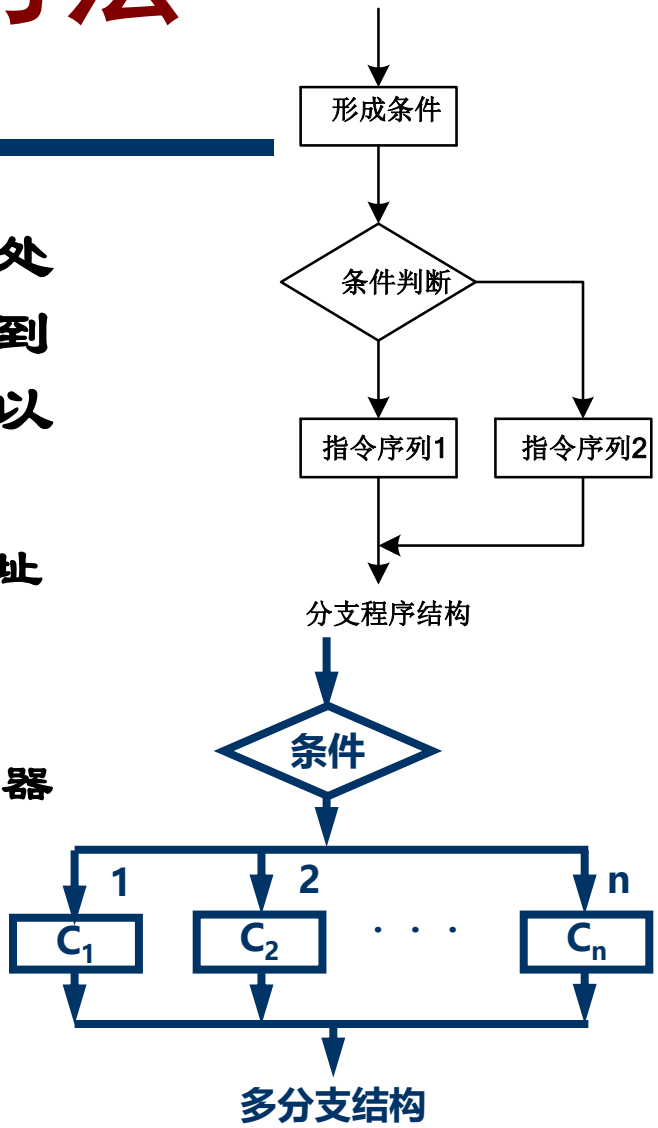
- **程序设计中最基本的结构**



顺序程序结构

分支程序设计方法

- 分支程序是根据判断条件转向不同的处理，则要采用分支程序结构。当执行到条件判断指令时，程序必定存在两个以上分支
  - **两分支**：条件转移，标志寄存器，直接寻址
    - 满足条件（条件成立）
    - 不满足条件（条件不成立）
  - **多分支**：无条件转移，变址寄存器，存储器间接寻址
  - 程序每次只能执行其中一个分支



例1. 实现符号函数Y的功能。
其中：-128≤X≤+127

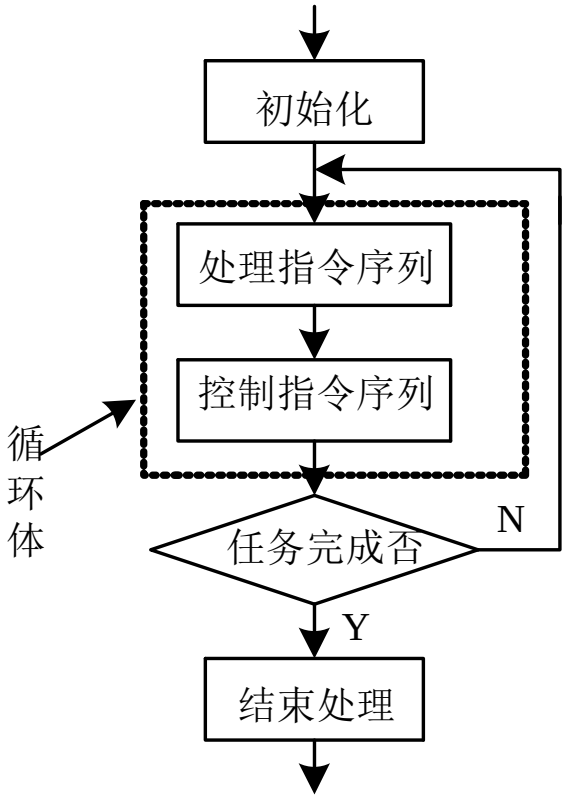
$$Y=\begin{cases}1 & \text{当}X>0\text{时}\\ 0 & \text{当}X=0\text{时}\\ -1 & \text{当}X<0\text{时}\end{cases}$$

```
X     DB    ?    ;被测数据
Y     DB    ?    ;函数值单元

      MOV   AL,0
      CMP   X,AL
      JG    BIG
      JZ    SAV
      MOV   AL,0FFH   ;小于0
      JMP   SHORT  SAV
BIG:  MOV   AL,1      ;大于0
SAV:  MOV   Y,AL      ;保存结果
```

循环程序设计方法

- 有一段指令被重复多次执行
  - 被循环多次执行的指令段称为循环体
  - 适应于处理算法相同，每次处理时需要有规律地改变数据或数据地址的问题

- 例如，求内存数据段中存放的N个字节数据（或字数据）的某种运算(加、减、乘、除，移动等等)
  - 循环体是加、减、乘、除，移动等运算指令
  - 设置一个地址指针指向这N个数据的首地址，再设置一个计数器
  - 每次运算之后，修改地址指针使其指向下一个数据，依次执行N次



循环程序结构

5.1 循环程序设计

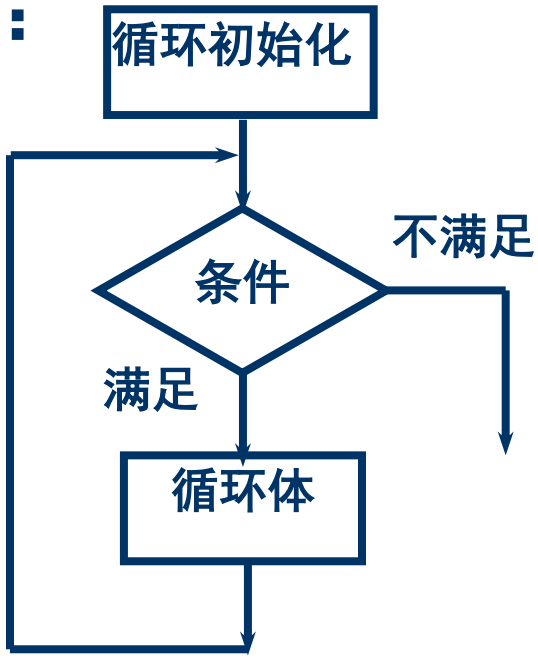
5.1.1 循环程序的结构形式

5.1.2 循环程序的设计方法

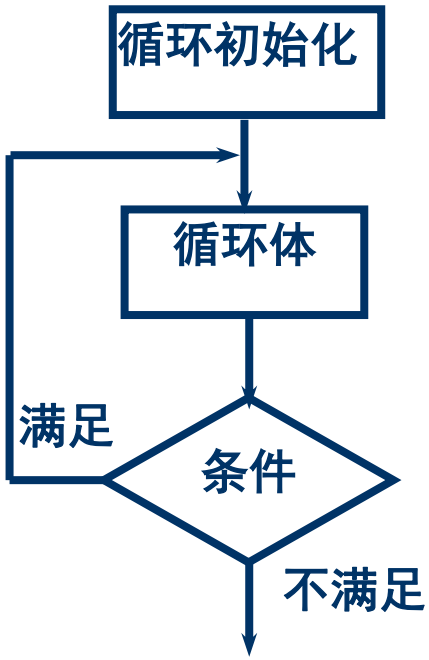
5.1.3 多重循环程序设计

5.1.1 循环程序的基本结构

- 两种基本结构
- 三个基本组成部分：
  - 1、初始设置
  - 2、循环体
  - 3、循环控制转移



DO-WHILE结构



DO-UNTIL结构

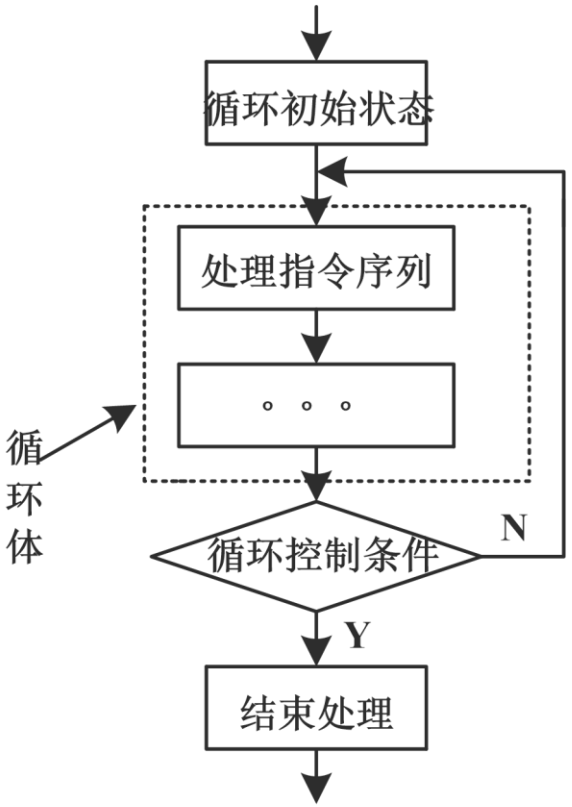
5.1.2 循环程序的设计方法

- 分析任务，实现三要素：

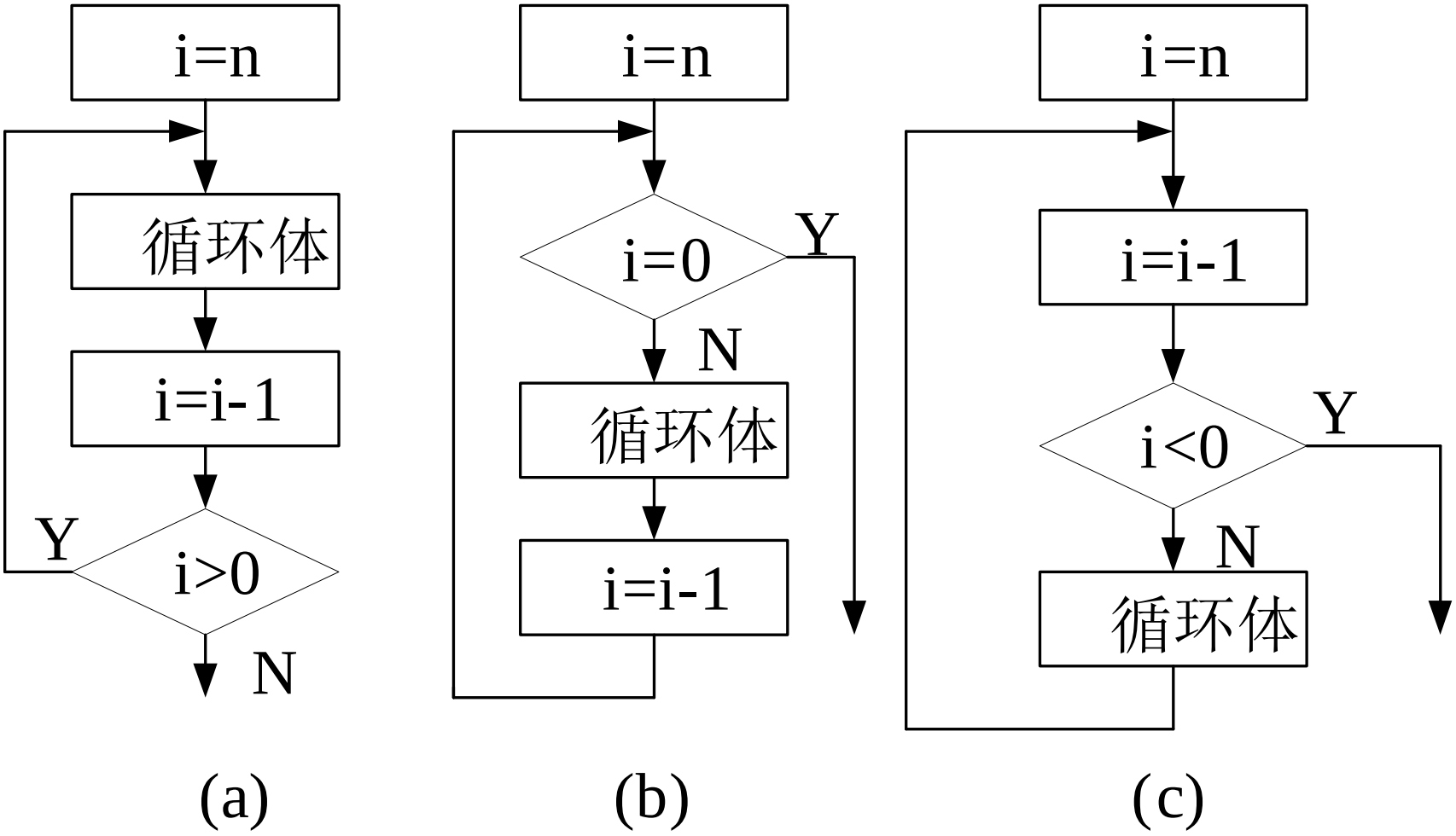
  1、初始设置
  - 循环次数，数据和变址指针初始化等

  2、循环体
  - 根据任务，选择算法
  - 修改指针等
  - 设置循环控制转移其他条件

  3、循环控制转移
  - 正确选择条件转移指令，2要素
    - 循环次数
    - 其他条件，如FLAGS
  - 可在循环体前，也可在循环体后，是程序优化设计的问



- 设计流程框图



- 不影响程序执行结果，只是性能优化的问　，有时性能可能差异较大
- 在自己的计算机上试一试(a)(b)两种循环体的运行时间

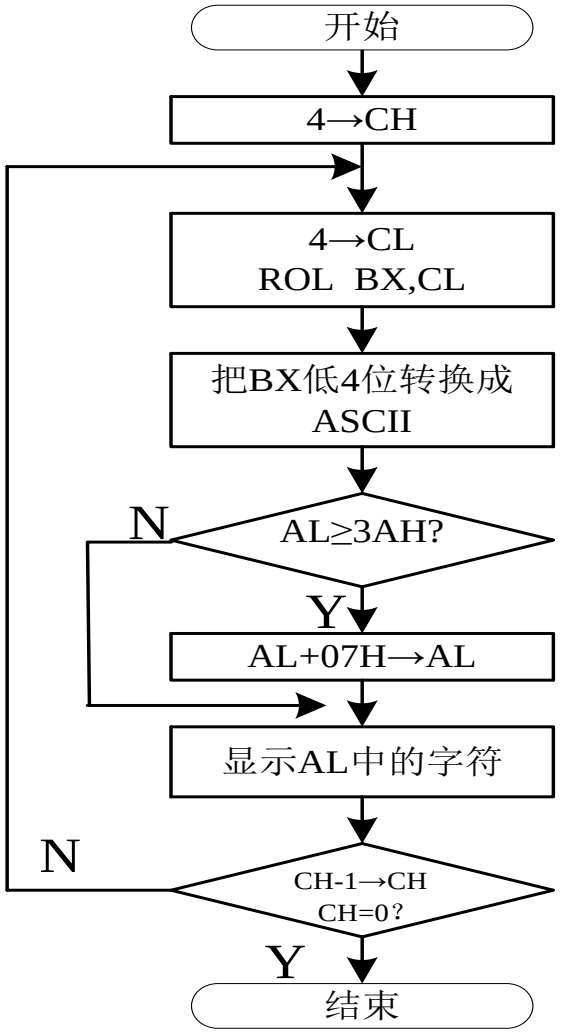
例5.1、试编制一个程序把BX寄存器内的二进制数用十六进制数的形式在屏幕上显示出来

分析问 ：把BX寄存器中16位的二进制数用4位十六进制数的形式在屏幕上显示

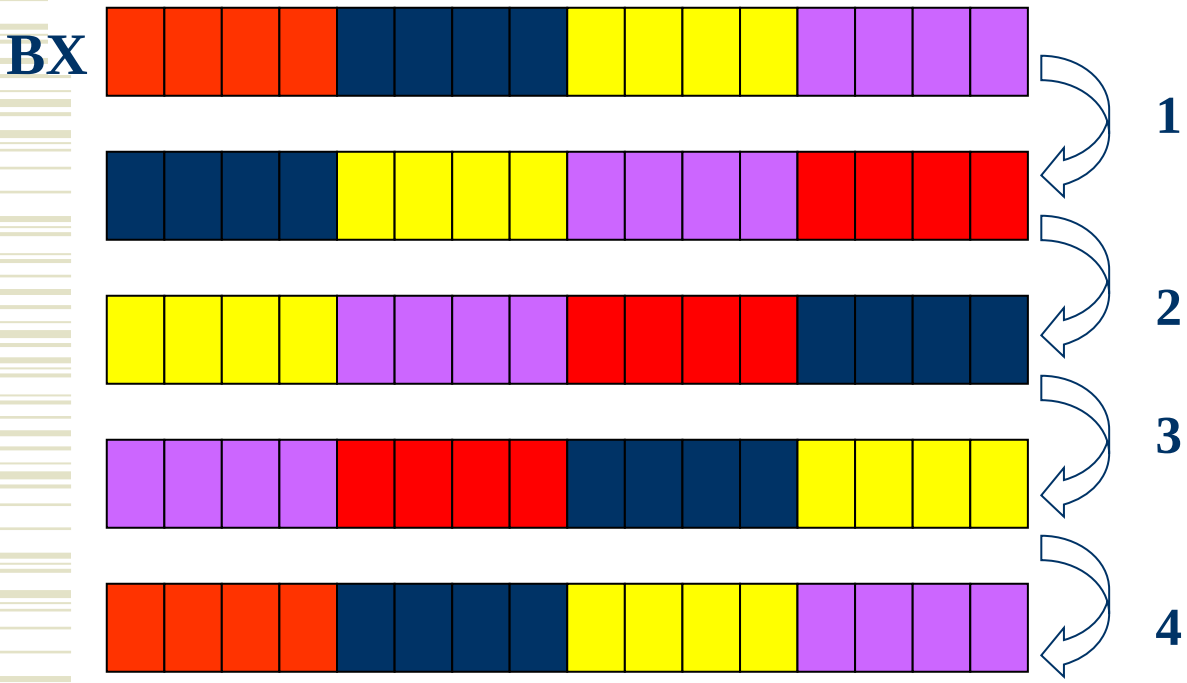
1、初始设置
- 循环次数：每次显示1个十六进制数（送显示器相应的ASCII），循环次数=4，CH=4

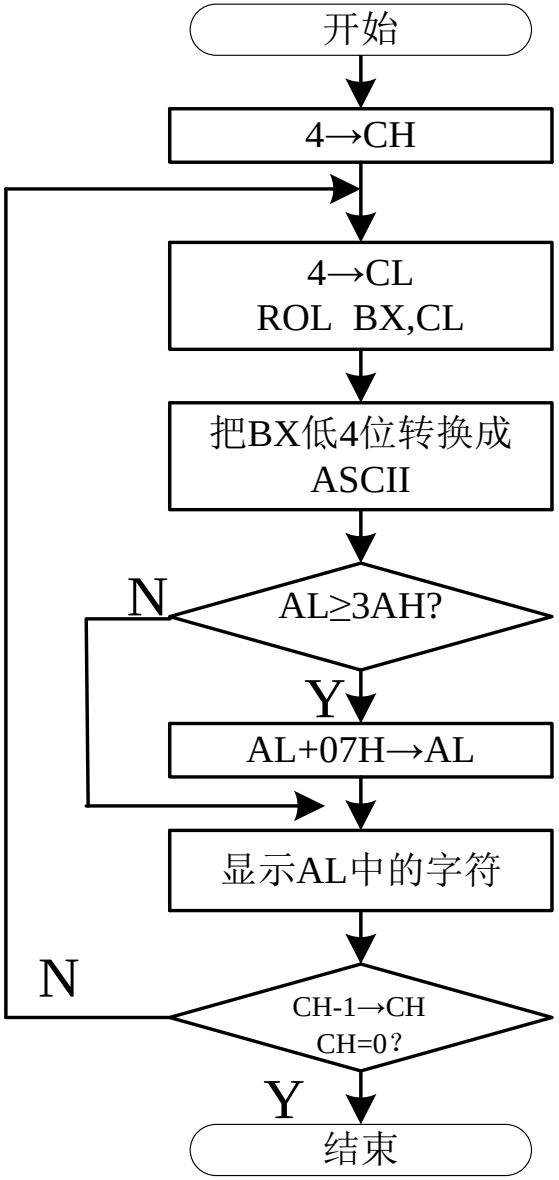
2、循环体
- 根据任务，选择算法
  - 每4位二进制数转换成1位十六进制数的ASCII
  - 调用DOS系统功能在屏幕上显示

3、循环控制转移
- 根据计数控制循环次数



流程图


BX
1
2
3
4


开始
4→CH
4→CL
ROL BX,CL
把BX低4位转换成
ASCII
AL≥3AH?
N
Y
AL+07H→AL
显示AL中的字符
CH-1→CH
CH=0?
N
Y
结束

课内测试CH05-1

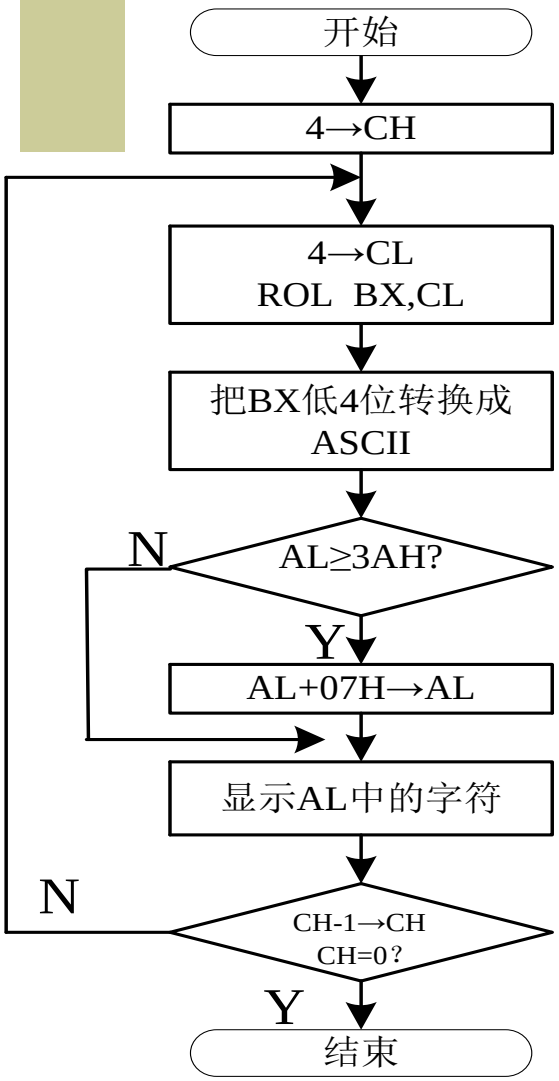
1. 请在填空中 [填空1] 填写“51”（10分）；

2. 设BX=6F02H，如下图5条指令程序（10分）

- 执行 mov al, bl 后，al=[填空2]H
- 执行 or al, 30h 后，al=[填空3]H

```
mov     cl,   4
rol     bx,   cl
mov     al,   bl
and     al,   0fh
or      al,   30h
```

没有数据分配问 ，直接编写程序代码段



流程图

```
prognam segment
main proc far
        Assume cs:prognam
start:
        push ds
        sub ax, ax
        push ax
初始化 → mov ch, 4
rotate: mov cl, 4
        …
printit:
        mov dl, al
        mov ah, 2
        int 21h
        dec ch
        jnz rotate
循环控制转移
        ret
main    endp
prognam ends
        end start
```

循环体

系统调用
显示1个字符

DOS环境下
返回DOS

熟记程序框架结构和系统调用约定

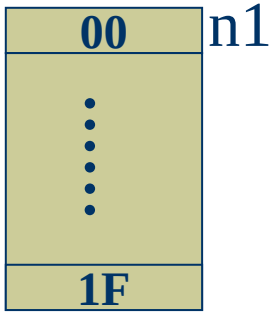
```
          mov   ch, 4
rotate:   mov   cl,  4
          rol   bx, cl
          mov   al,  bl
          and   al,  0fh          ;al的低4位0-9，A-F
          add   al,  30h          ;al的低4位30-39，3A-3F
          cmp   al,  3ah
          jl    printit                ;'0'-'9' ASCII 30H-39H
          add   al,  7h           ;'A'-'F' ASCII 41H-46H
printit:  mov   dl,  al
          mov   ah,  2
          int   21h
          dec   ch
          jnz   rotate
```

例5.1几点说明：

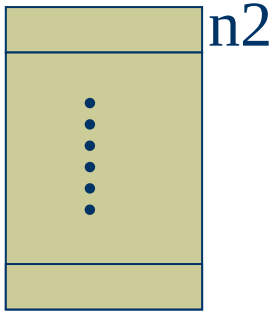
- 程序实现没有使用LOOP指令
  - 解决寄存器使用冲突
  - 用计数值控制循环结束，不是非得用LOOP指令
- 关于循环次数控制
  - 也可以把计数值初始化为0，每循环一次加1，然后比较判断
- 也可用LOOP指令
  - 通过堆栈保存信息解决CX冲突问

例：编制一个数据块移动程序 (已知循环次数)

- 1) 任务1：用程序设置数据
  - 给内存数据段（DATA）中偏移地址为n1开始的连续32个字节单元，置入数据00H，01H，02H，¨ ¨ ¨，1FH



- 2) 任务2：移动数据
  - 将内存数据段（DATA）中偏移地址为n1的数据传送到偏移地址为n2开始的连续的内存单元中去



1）任务1：用程序设置数据

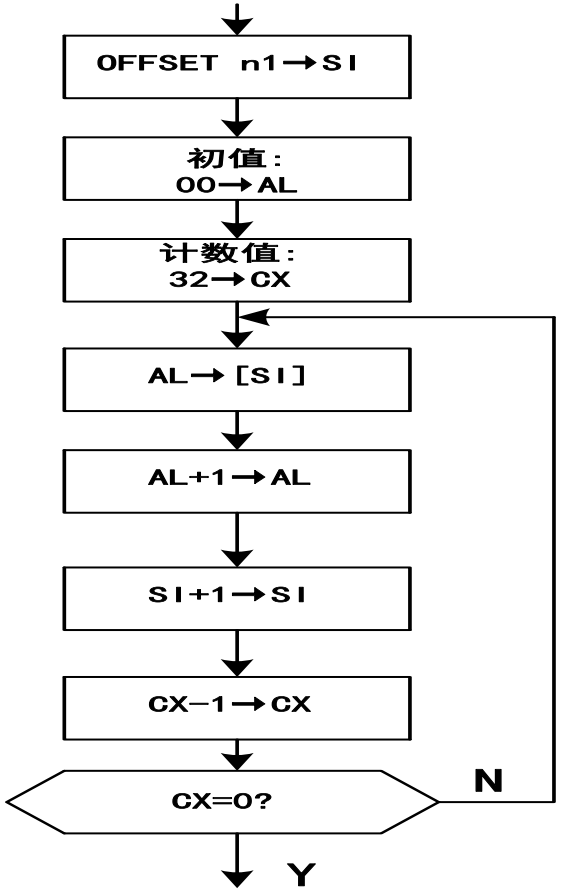
给内存数据段（DATA）中偏移地址为n1开始的连续32个字节单元，置入数据00H，01H，02H，……，1FH

- 对有规律（连续）的内存单元操作，地址有规律变化采用变址寻址方式
- 多次同样功能的处理，数据处理有规律，
- 采用循环程序结构

1、初始设置：
- 循环次数：连续32个字节单元，循环次数=32
- 数据初值：数据有规律变化，程序中可以自动生成数据，数据初值=00H
- 变址指针初始化：内存数据段中偏移地址为n1

2、循环体
- 根据任务，选择算法：一次设置一个字节
- 修改指针：变址指针修改
- 修改数据，生成新的数据
- 设置循环控制转移其他条件：无

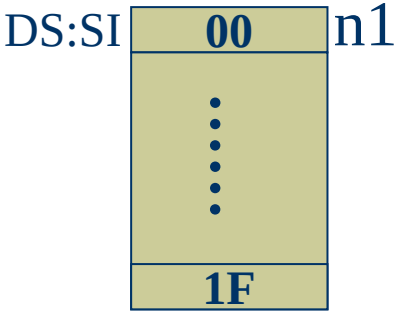
3、循环控制转移
- 根据计数控制循环次数

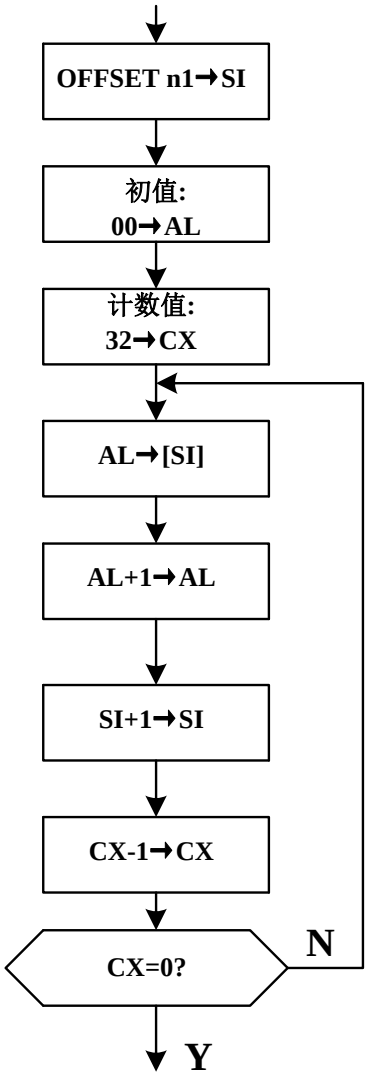


置入数据程序流程图

选择寄存器
进一步细化程序流程

- 循环的初始化部分完成
  - (1) 将n1的偏移地址置入变址寄存器SI，这里的变址寄存器作为地址指针用
  - (2) 待传送数据的起始值00H 送 AL
  - (3) 循环计数值32（或20H）送计数寄存器CX
- 循环体中完成
  - (4) AL中内容送地址指针所指存储器单元
  - (5) AL加1送AL；依次得到01H，02H，03H，…，1FH
  - (6) 地址指针SI加1，指向下一个地址单元
- 循环结束判断
  - (7) 计数器CX-1→CX
  - (8) 若CX≠0继续循环；否则结束循环
- 程序源代码请自己编写



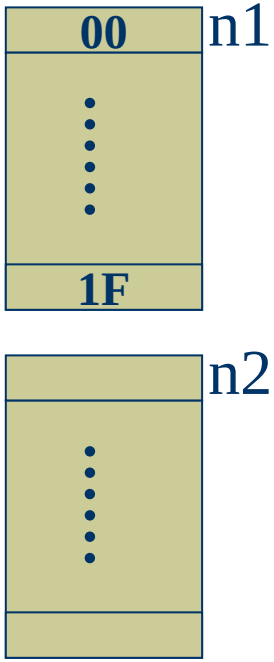


置入数据程序流程图

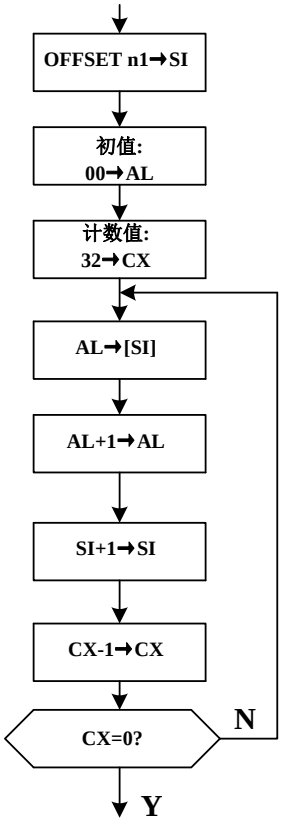
2)任务2：移动数据

将内存数据段（DATA）中偏移地址为n1的数据传送到偏移地址为n2开始的连续的内存单元中去

- 程序结构：这个问　是数据块移动问　，可选择循环结构或串传送指令来处理
  - 选择串传送指令MOVSB，或REP MOVSB
- 数据定义：要定义源串和目的串
  - 源串是任务（1）中所设置的数据
  - 目的串要保留相应长度的空间
- 处理方法：MOVSB指令是字节传送指令，它要求事先设置约定寄存器：
  - ① 将源串的　偏移地址送SI，段地址为DS
  - ② 目的串的　偏移地址送DI，段地址为ES
  - ③ 串长度送CX寄存器中
  - ④ 并设置方向标志DF

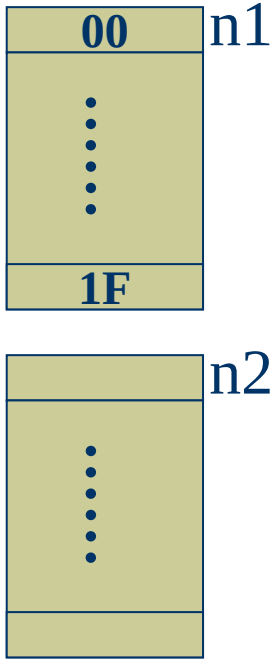


程序源代码请自己编写



置入数据程序流程图

```
data1     segment
n1        db 32 dup (?)
data1     ends

data2     segment
n2        db 32 dup (?)
data2     ends

prognam   segment
main      proc  far
          assume cs:prognam,
                 ds:data1, es:data2
start:    push  ds
          sub  ax, ax
          push  ax

          mov ax,  data1
          mov ds,  ax
          mov ax,  data2
          mov es,  ax

          mov  si, offset n1
          mov al,  0
          mov cx, 32
rotate:   mov [si],  al
          inc   si
          inc   al
          dec   cx
          jnz  rotate

          mov si,  offset n1
          mov di,  offset n2
          cld
          mov cx,  32
          rep  movsb

          ret

main      endp
prognam   ends
          end  start
```



例子5.2 数“1”的个数

在addr单元中存放着数 Y 的地址，把 Y 中1的个数存入 COUNT 单元中

参看p178

（1）数“1”的方法

a） 移位到进位，测试进位；测试符号位

b） 判最高位是否为1

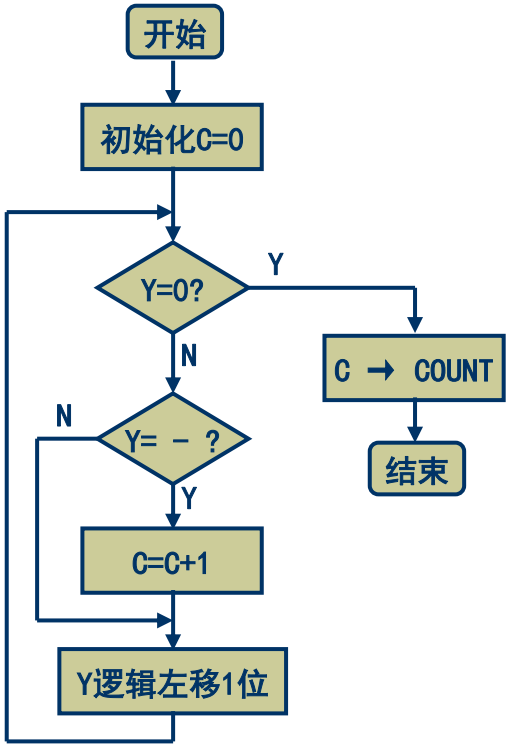
（2）循环控制条件

a） 简单思路：计数值以16控制

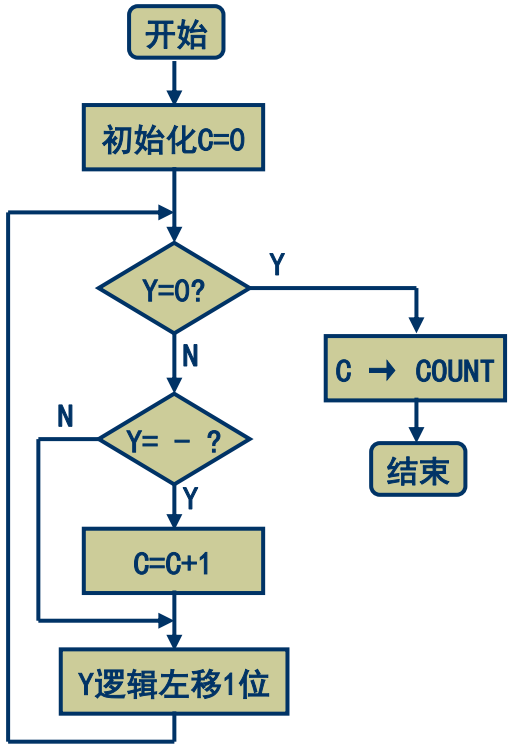
b） 比较好的方法：简单思路结合测试数是否为0

（3）DO_WHILE 结构

Y本身为0的可能性



```
datarea segment
        addr        dw  number
        number      dw  y
        count       dw  ?
datarea segment
prognam             segment
mian    proc        far
        assume      cs:prognam, ds:datarea
start:  push        ds
        sub         ax, ax
        push        ax
        mov         ax, datarea
        mov         ds, ax
        mov         cx, 0
        mov         bx, addr
        mov         ax, [bx]
repeat: test        ax, 0ffffh
        jz          exit
        jns         shift
        inc         cx
shift:  shl         ax, 1
        jmp         repeat
exit:   mov         count, cx
        ret
mian:   endp
prognam ends
        end         start
```



例子5.3 删除在未经排序的数组中找到的数（待找的数存放在AX中）P179

分析　意：

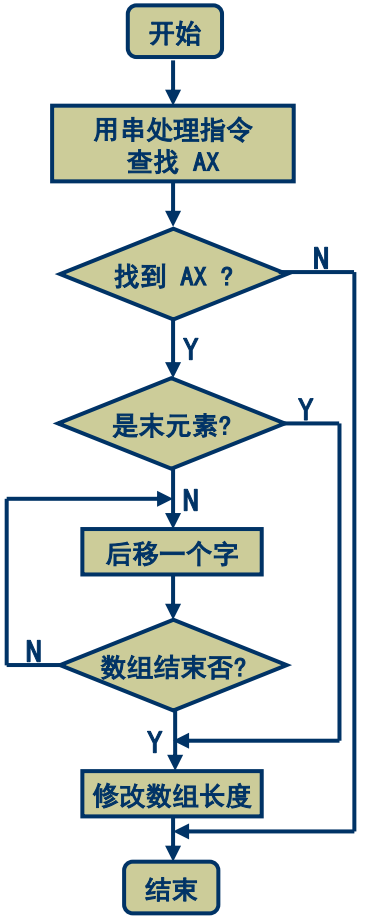
（1）如果没有找到，则不对数组作任何处理

（2）如果找到这一元素，则应把数组中位于地址中的元素向低地址移动一个字，并修改数组长度值

（3）如果找到的元素正好位于数组末尾，只要修改数组长度值

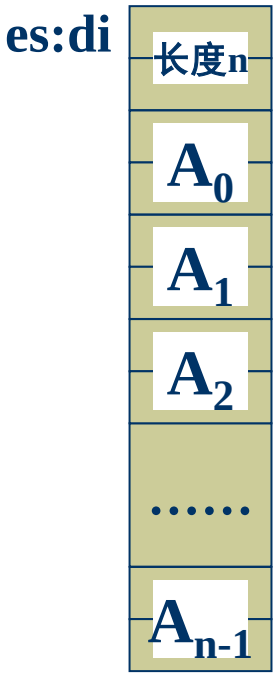
（4）查找元素用串处理指令（repnz　scasw）

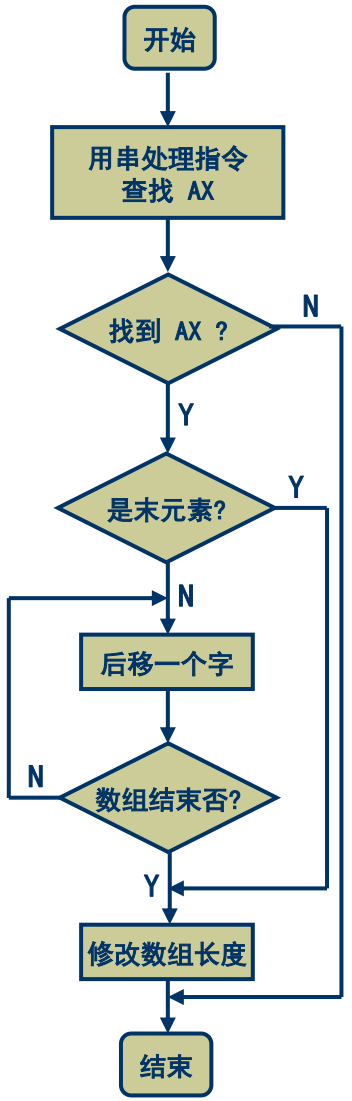
（5）删除元素用循环结构



子程序源代码

```
del_ul    proc     near
          cld
          push     di
          mov      cx, es:[di]
          add      di, 2
          repne    scasw
          je       delete
          pop      di
          jmp      short  exit
delete:   jcxz     dec_cnt     ;  如果CX=0, 转移
next_el:  mov      bx, es:[di]
          mov      es:[di-2], bx
          add      di, 2
          loop     next_el
dec_cnt:  pop      di
          dec      word ptr es:[di]
exit:     ret
del_ul    endp
```





执行了几次pop操作?

SCAS指令执行的操作：

1. DST-AL/AX/EAX，但结果不保存，根据结果设置标志位
2. 目标操作数的地址指针（变址寄存器DI）的修改

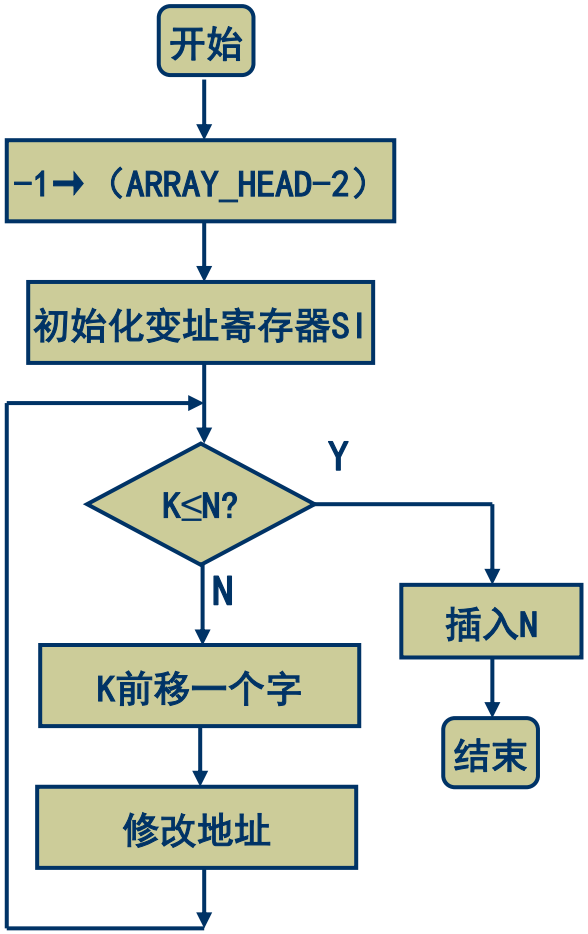
REPNE执行的操作：

1. 若 CX＝0（计数到)或ZF＝1(相等),则结束重复
2. 否则,修改计数器 CX－1→CX, 执行后跟的串操作指令。转1，继续重复上述操作

例子5.4　在已整序的正数字数组中插入正数N

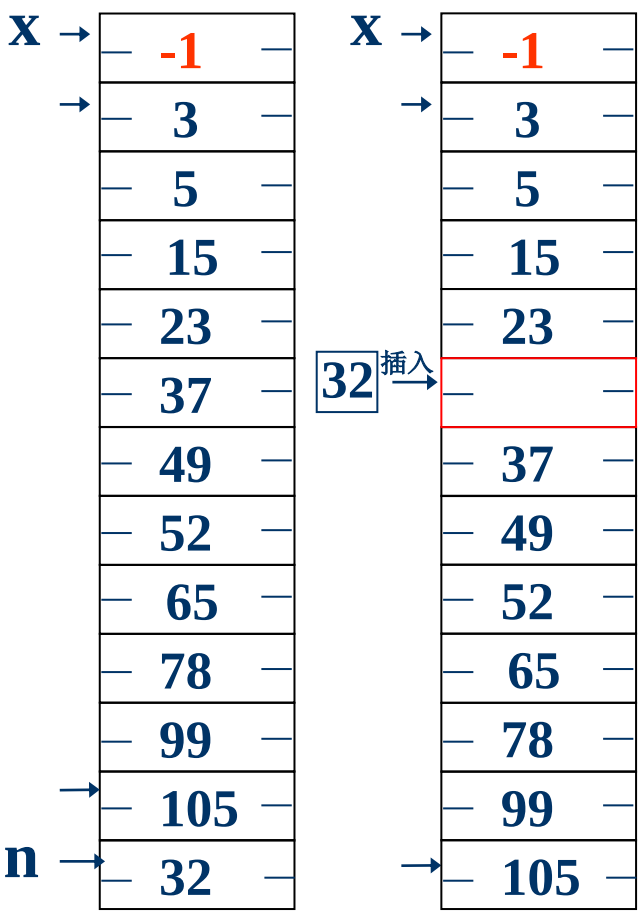
找到位置，将数据向　地址移一个字，插入N, 结束

- 循环控制条件：找到位置即可
  - 位置一定能找到，因此无　计数次数等
- 将　于N的数向　地址移一个字，边找边移，因此循环体内处理为向　地址移一个字
- 插入N在循环结构外
- 循环结构的主要任务是找位置和移字数据



```
x           dw   ?
array_head  dw   3,5,15,23,37,49,52,65,78,99
array_end   dw   105
n           dw   32
```

```
        mov   ax, n
        mov   array_head-2, 0ffffh
        mov   si, 0
compare:
        cmp   array_end[si], ax
        jle   insert
        mov   bx, array_end[si]
        mov   array_end[si+2], bx
        sub   si, 2
        jmp   short compare
insert:
        mov   array_end[si+2], ax
```



例子5.5

- 设数组X、Y中分别存有10个字型数据
  试实现以下计算并把结果存入数组Z单元

Z0＝X0+Y0　　Z1＝X1+Y1
Z2＝X2-Y2　　Z3＝X3-Y3
Z4＝X4-Y4　　Z5＝X5+Y5
Z6＝X6-Y6　　Z7＝X7-Y7
Z8＝X8+Y8　　Z9＝X9+Y9

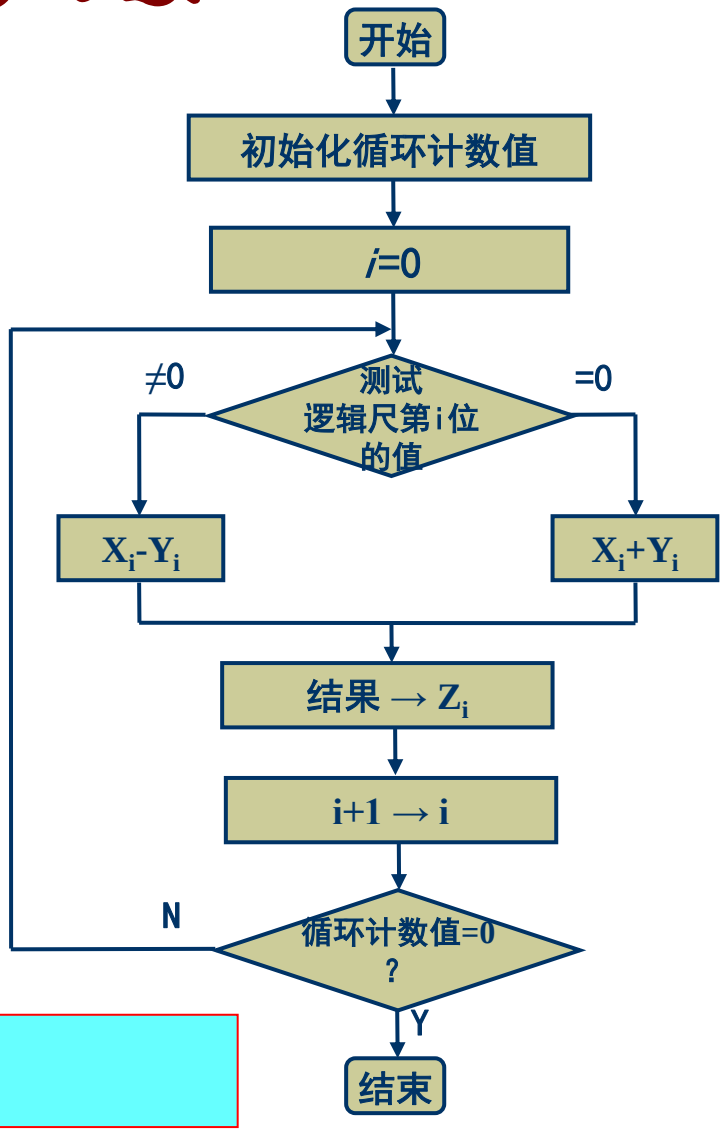
逻辑尺方法

（1）设立标志位

0000000011011100

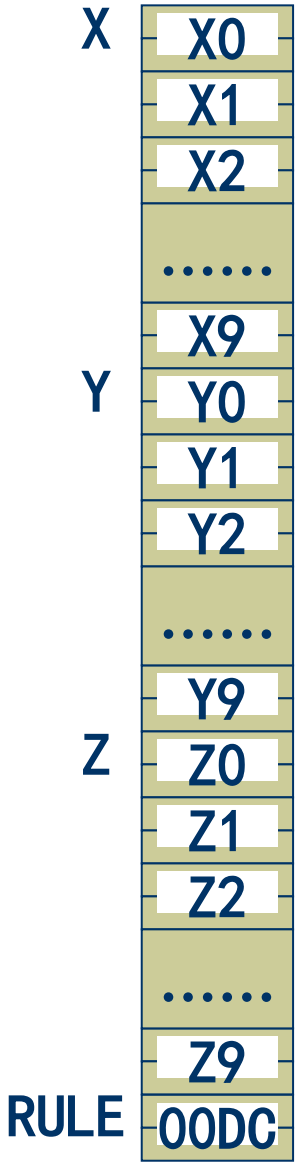
$Z_2=X_2-Y_2$　　$Z_0=X_0+Y_0$

（2）进入循环后判断标志位来确定该做的工作

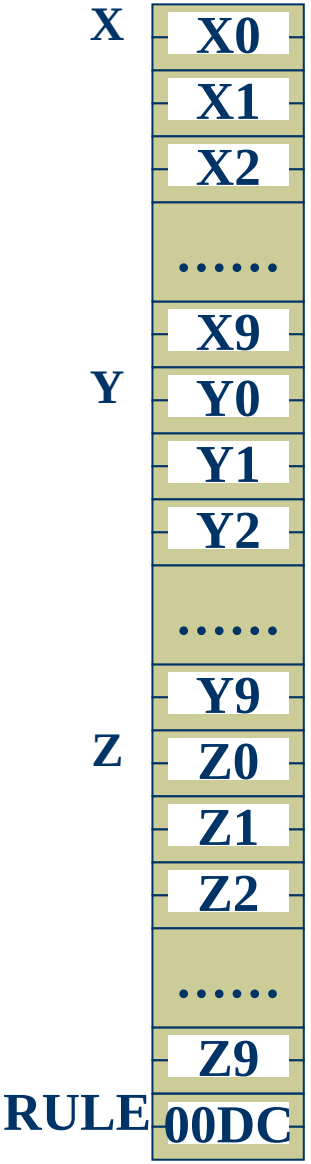


建立逻辑尺：　++−−+−−−++
0000000011011100

```
DATA    SEGMENT
   X    DW  X0,X1,X2,X3,X4
        DW  X5,X6,X7,X8,X9
   Y    DW  Y0,Y1,Y2,Y3,Y4
        DW  Y5,Y6,Y7,Y8,Y9
   Z    DW  10 DUP(?)
   RULE DW  0000000011011100B;逻辑尺
DATA    ENDS
```



```
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA
MAIN   PROC   FAR
       MOV    AX,DATA
       MOV    DS,AX
       MOV    CX,10          ;循环次数
       MOV    DX,RULE        ;逻辑尺
       MOV    BX,0           ;地址指针
NEXT:  MOV    AX,X[BX]       ;取X中的一个数
       SHR    DX,1           ;逻辑尺右移一位
       JC     SUBS           ;分支判断并实现转移
       ADD    AX,Y[BX]       ;两数加
       JMP    SHORT RESULT
SUBS:  SUB    AX,Y[BX]       ;两数减
RESULT:MOV    Z[BX],AX       ;存结果
       ADD    BX,2           ;修改地址指针
       LOOP   NEXT
       MOV    AX,4C00H       } 返回DOS
       INT    21H            }
MAIN   ENDP
CODE   ENDS
       END    MAIN
```



课内测试CH05-2

1. 请在填空中 [填空1] 填写“90”（10分）；

2. 有逻辑尺al=?101110?B，“0”表示加运算，“1”表示减运算，如右图数组运算（10分）

- 逻辑尺最高位=[填空2]
- 逻辑尺最低位=[填空3]

Z0＝X0+Y0	Z1＝X1+Y1
Z2＝X2-Y2	Z3＝X3-Y3
Z4＝X4-Y4	Z5＝X5+Y5
Z6＝X6-Y6	Z7＝X7-Y7

- 返回DOS操作系统有两种方法：

通用方法：

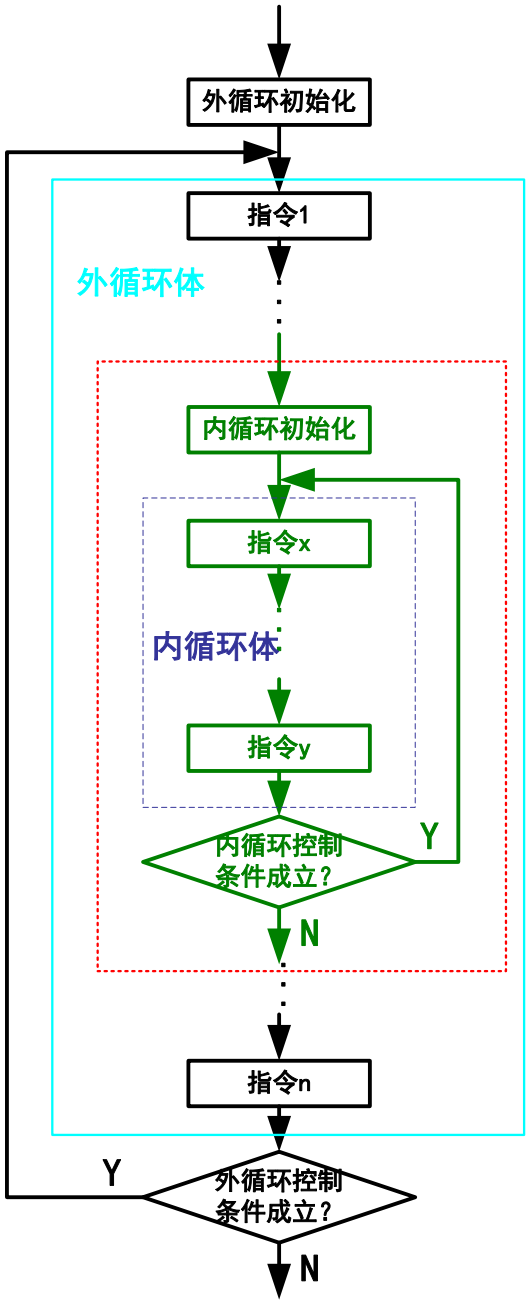
```
start:
        push  ds
        sub  ax, ax
        push  ax
        ……
        ret
main  endp
```

级DOS版本可用方法：

```
start:
        ……
        mov  ax,  4c00h
        int  21h
main  endp
```

5.1.3 多重循环程序设计

- 多重循环程序结构
  - 循环中有循环
  - 内外层循环都应该具有完整的循环程序结构



例5.7： 有一个首地址为ARRAY的N字的数组，编制程序使该数组中的数按照从大到小的顺序排序 参看p187.asm

- 算法：小数沉底法/起泡排序法
  - 结果：数据由大到小排列，或由小到大排列

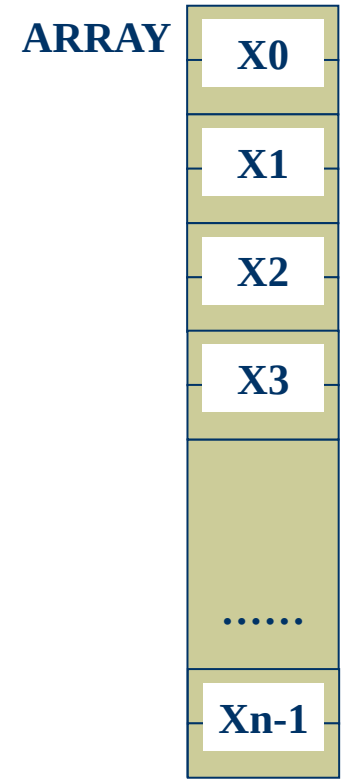
第1遍：两相邻数据比较，交换，数据按大到小排列， 比较N-1次，最后一个是最小数

第2遍：两相邻数据比较，交换，数据按大到小排列，比较N-2次，最后2个数由大到小

……

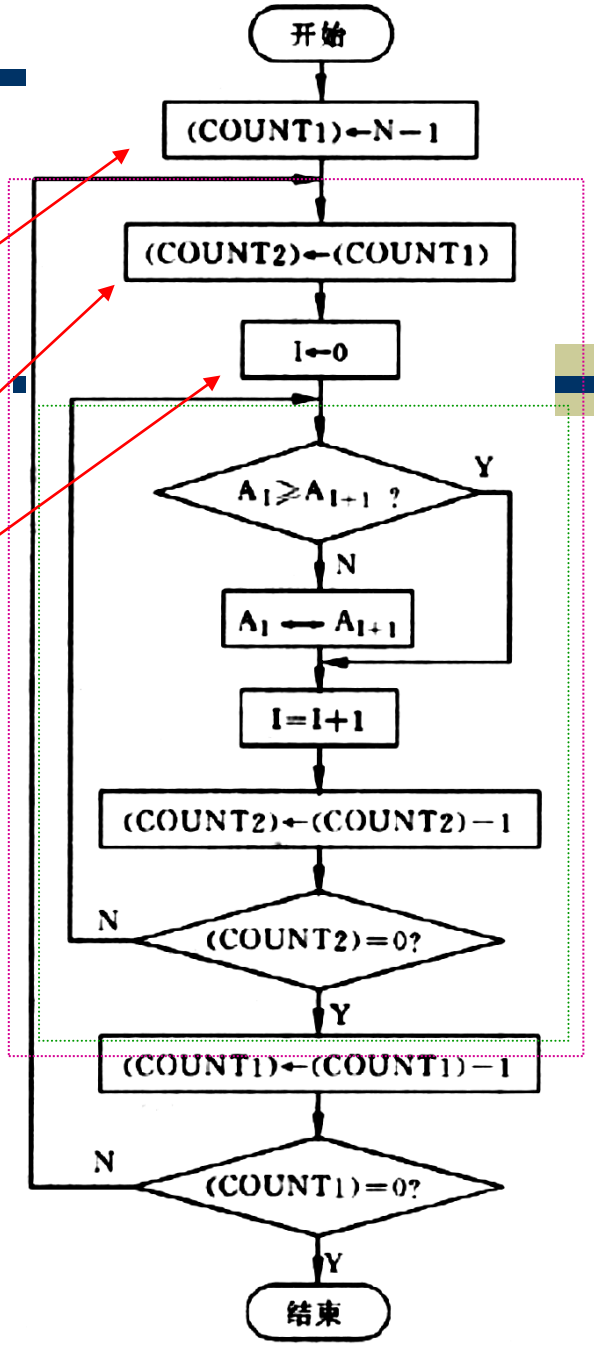
第N-1遍：两相邻数据比较，交换，数据按大到小排列，比较1次，得到结果

- 总共N-1遍：作为外循环计数，初值=N-1
- 每遍的比较次数：作为内循环计数，初值=外循环计数当前值



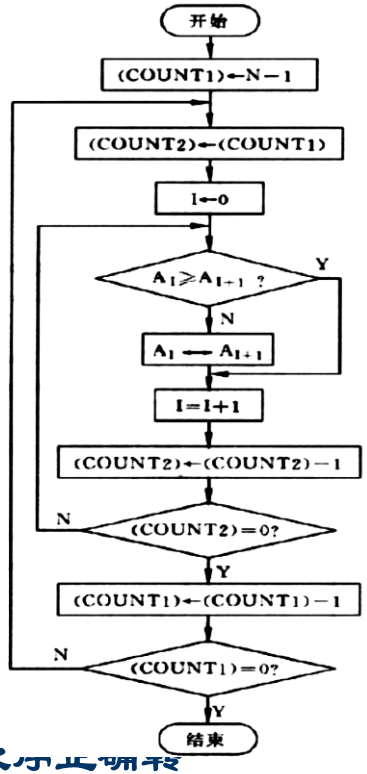
沉底由上向下比较，气泡由下向上比较

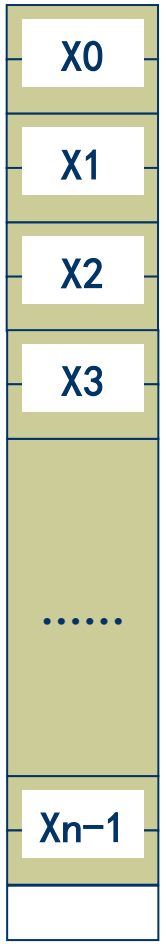
- 总共N-1遍：作为外循环计数，初值=N-1
- 每遍的比较次数：作为内循环计数，初值=外循环计数当前值
- 每次从最低地址单元开始



```
DATA    SEGMENT
ARY     DW    n DUP(?)
CT      EQU   ($-ARY)/2      ;元素个数
DATA    ENDS

CODE    SEGMENT

        ASSUME CS:CODE,DS:DATA
MAIN    PROC    FAR
        MOV     AX,DATA
        MOV     DS,AX
        MOV     DI,CT-1       ;初始化外循环次数
LOP1:   MOV     CX,DI         ;置内循环次数
        MOV     BX,0          ;置地址指针
LOP2:   MOV     AX,ARY[BX]
        CMP     AX,ARY[BX+2]  ;两数比较
        JGE     CONT          ;Xi≥Xi+2,次序正确转
        XCHG    AX,ARY[BX+2]  ;次序不正确互换位置
        MOV     ARY[BX],AX
CONT:   ADD     BX,2          ;修改地址指针
        LOOP    LOP2          ;内循环控制
        DEC     DI            ;修改外循环次数
        JNZ     LOP1          ;外循环控制
        MOV     AX,4C00H
        INT     21H
MAIN    ENDP
CODE    ENDS
        END     MAIN
```

CT=n



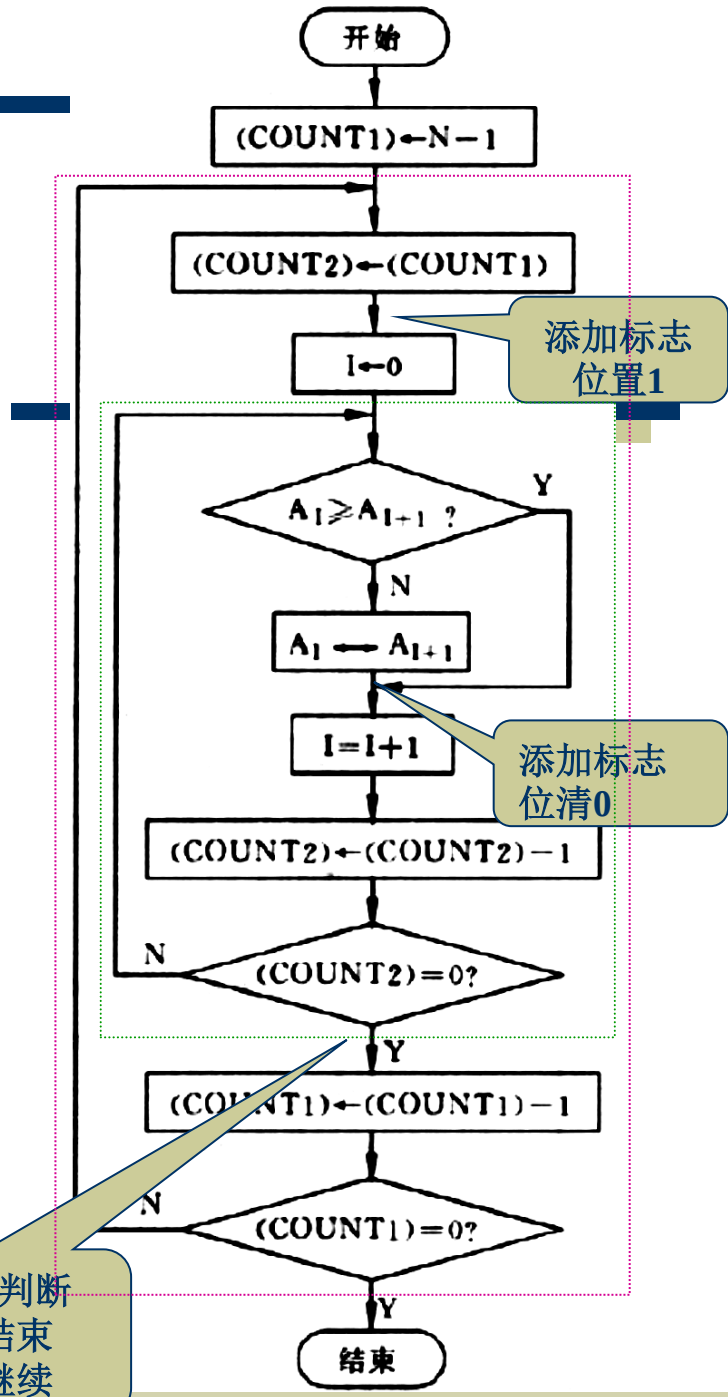


汇编程序
地址计数器值 $→

CT EQU ($-ARY)/2 什么意思？
只是定义了一个常数，不分配存储单元

CT db ($-ARY)/2 什么意思？
分配一个字节存储单元，并赋初值，变量名为CT

- 对例5.7中算法进行优化，提程序效率(参考例5.8)
  - 提前结束，设置交换操作标志位
    - 如果某遍没有交换操作，说明排序已经符合要求，可以提前结束
    - 每次进入内循环之前将交换操作标志位初始化为1，如果内循环中有数据交换则将标志位清0
    - 内循环的循环次数同例5.7
  - 外循环的结束条件：循环次数计数=0 或者 交换操作标志位=1
  - 参看p189框图和p190.asm



容易出错点

- 特别要注意进入循环体时，循环次数、各寄存器、标志位、存储单元的初始值一定要正确，必要时自己可以模拟执行一次
- 循环判定条件的确定
- 指针及循环控制条件的修改等

5.2 分支程序设计

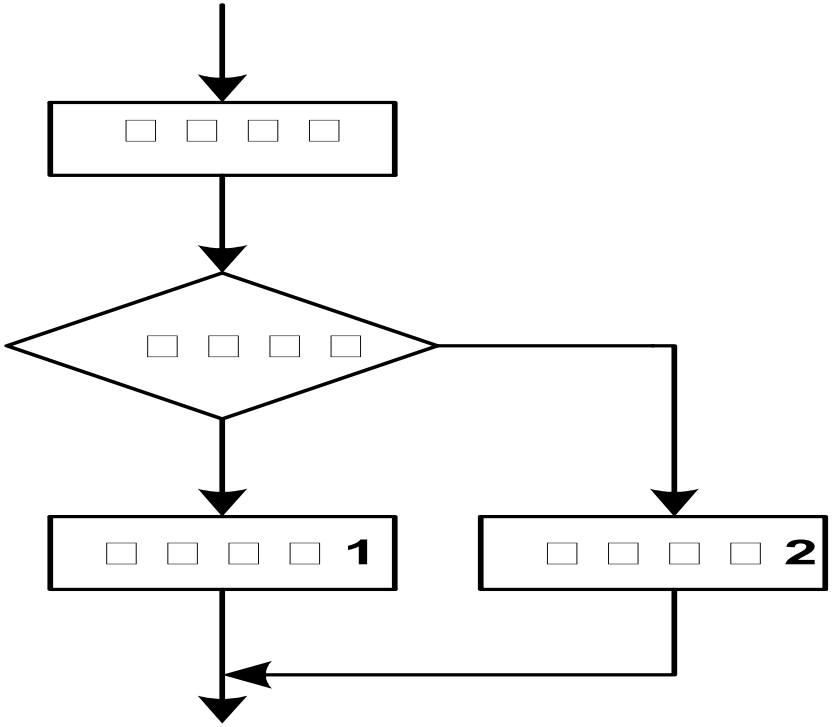
5.2.1 分支程序的结构形式

5.2.2 分支程序的设计方法

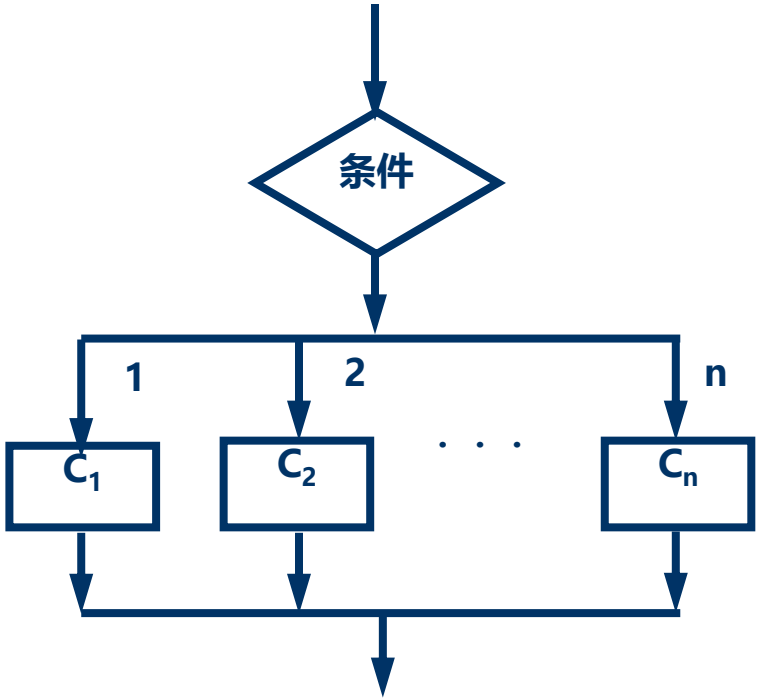
5.2.3 跳跃表法

5.2.1 分支程序的结构形式

程序有两条以上执行路径，但每次只能执行一个指令序列



IF-THEN-ELSE 结构



CASE 结构

5.2.2 分支程序的设计方法

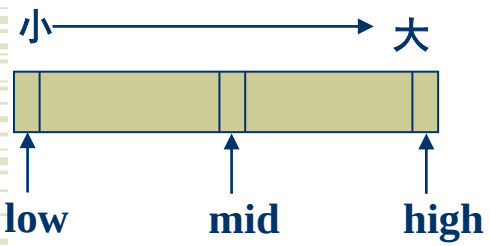
- 一般使用条件转移指令产生分支
  - 利用转移指令不影响条件码的特性，连续使用条件转移指令
- 基于逻辑尺的条件转移指令
- 跳跃表法的无条件间接寻址转移指令

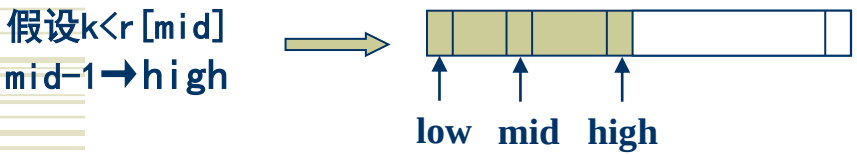
例子5.9：有一个从小到大顺序排列的无符号数数组，DI=数组首地址，数组中第一个单元存放数组长度，AX中有一个无符号数。要求在数组中查找与AX相等的数，如找到，则使CF=0，并在SI中给出该元素在数组中的相对位置；如未找到，则使CF=1，SI给出最后一个比较元素的偏移地址。

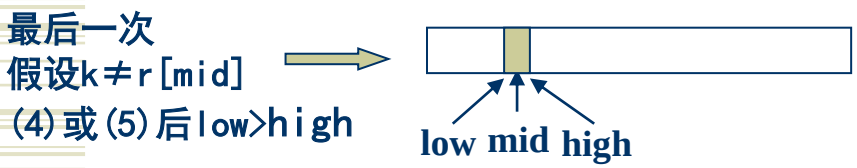
- 查找时：
  - 如果数据大小排序不定，只能用顺序查找方法；
  - 如果数据大小排序规整，也可用折半查找法，以提高查找效率

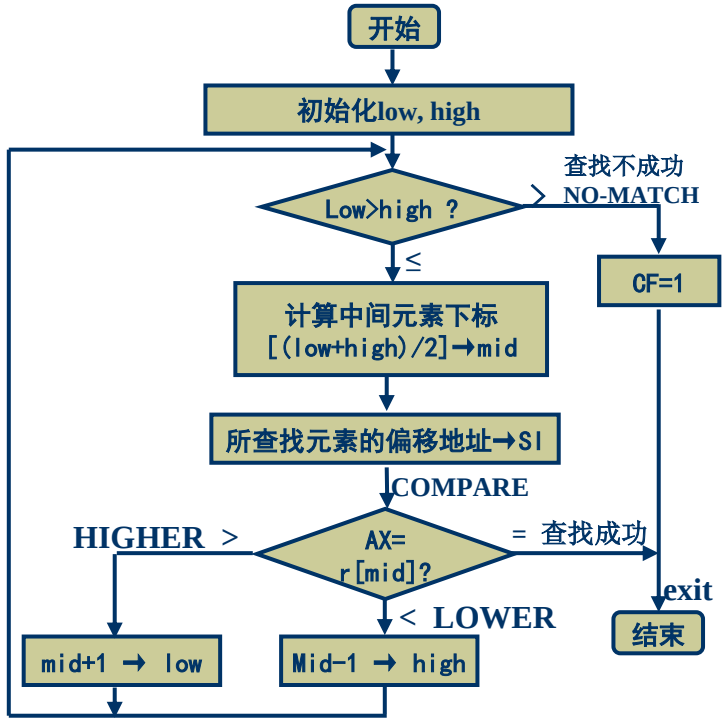
折半查找算法

在一个长度为n的有序数组r中，查找元素k的折半查找算法如下：
(1)初始化被查找数组的首尾下标：1→low, n→high;
(2)若low>high，则查找失败，置CF=1，退出程序。
　否则，计算中点：(low+high)/2 → mid;
(3)k与中点元素r[mid]比较。若k=r[mid]，则查找成功，程序结束；若k<r[mid]，
　则转步骤(4);若k>r[mid]，则转步骤(5); (假设数据由小到大排列)
(4)低半部分查找(lower)，mid-1 → high，返回步骤(2)，继续查找；
(5)低半部分查找(lower)，mid+1 → low，返回步骤(2)，继续查找；





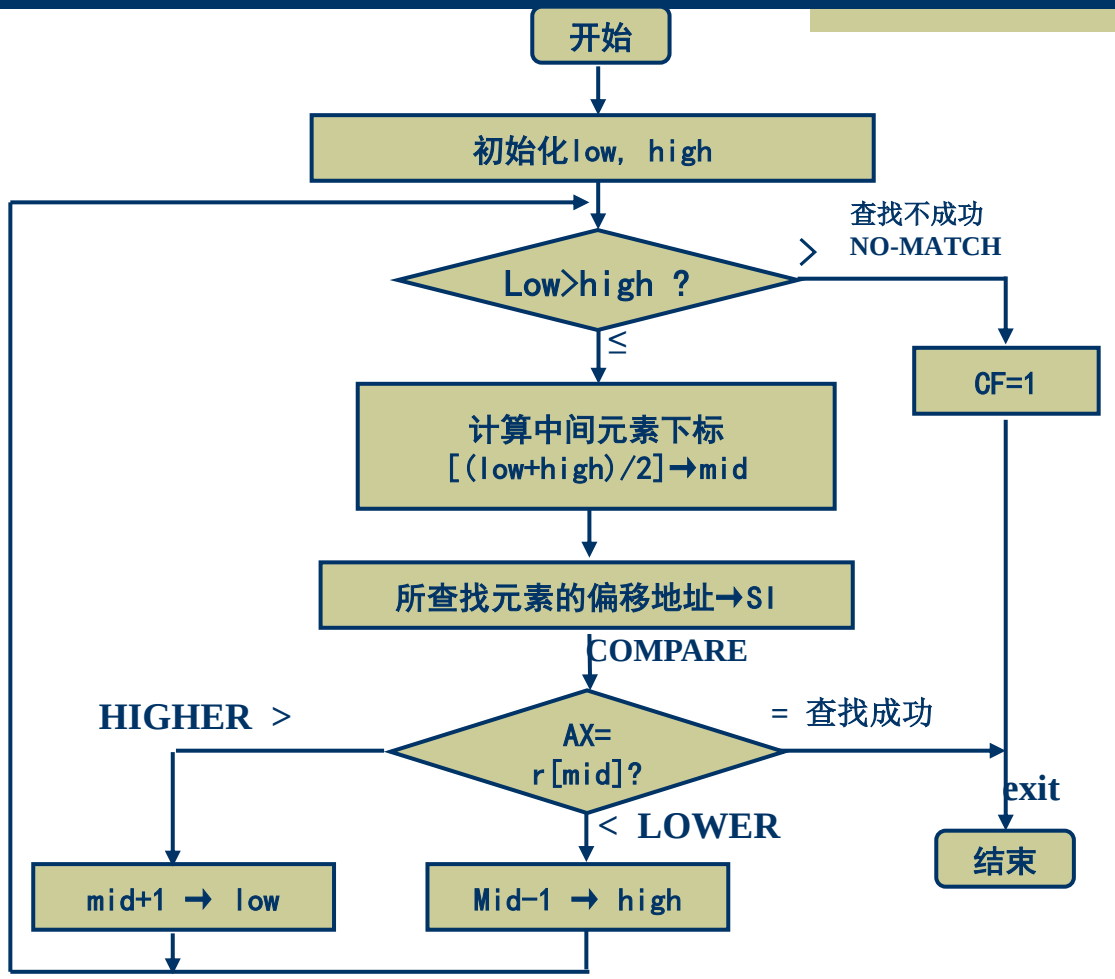




折半查找法

流程：图5.11
(参看核心程序段p194search～no_match)

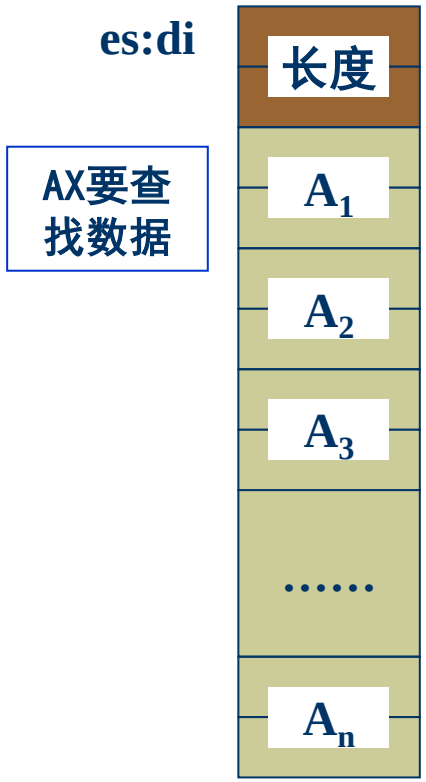
参看p192.asm



```
dseg        segment
    low_idx         dw  ?
    high_idx        dw  ?
dseg        ends
cseg        segment
b_search  proc      near
    assume  cs:cseg
    assume  ds:dseg, es:dseg
    push    ds
    push    ax
    mov     ax, dseg
    mov     ds, ax
    mov     es, ax
    pop     ax

    cmp     ax, es:[di+2]
    ja      chk_last        ;ax>A1
    mov     si, 2
    je      exit            ;ax=A1
    jmp     no_match        ;ax<A1
chk_last:
    mov     si, es:[di]
    shl     si, 1
    add     si, di
    cmp     ax, es:[si]
    jb      search          ;ax<An
    je      exit            ;ax=An
    jmp     no_match        ;ax>An
```

```
search:
    mov     low_idx, 1
    mov     bx, es:[di]
    mov     high_idx, bx
    mov     bx, di
mid:
    mov     cx, low_idx
    mov     dx, high_idx
    cmp     cx, dx
    ja      no_match
    add     cx, dx
    shr     cx, 1       ;(cx+dx)/2
    mov     si, cx
    shl     si,1        ; SI*2→SI
compare:        [mid]=[bx+si]
    add     si, bx
    cmp     ax, es:[si]
    je      exit
    ja      higher
lower:
    dec     cx
    mov     high_idx, cx
    jmp     mid
higher:
    inc     cx
    mov     low_idx, cx
    jmp     mid
```

```
no_match:
    stc         ;CF置1
exit:
    pop ds
    ret
b_search  endp
cseg        ends
    end
```



- 为了提　程序效率
  - 将最常用的数据尽量放在寄存器中
  - 尽量合理分配使用寄存器

分支程序小结

（１）形成条件：CMP指令、运算类指令、TEST指令等 （*置条件码类指令*）

（２）实现转移：条件转移指令

（３）逻辑尺方法实现2个以上分支

（４）循环结构可以认为是分支结构的一个特例，分支结构是循环结构的基本组成部分

5.2.3 跳跃表法

- 分支程序的两种结构形式都可以用上面所述的基于判断跳转方法来实现。此外，在实现CASE结构时，还可以使用跳跃表法，使程序根据不同的条件转移到多个程序分支中去

- 利用跳跃表法实现多路分支，关键是：
  - 构建跳转表（分支转移目标地址表）
  - 灵活、正确使用无条件间接转移指令实现跳转

存储器单元

目标地址1
目标地址2
……
目标地址n

例5.10：根据 AL 寄存器中哪一位为 1（从低位到高位），把程序转移到 8 个不同的程序分支

```
branch_table  dw  routine1
              dw  routine2
              dw  routine3
              dw  routine4
              dw  routine5
              dw  routine6
              dw  routine7
              dw  routine8
```

（寄存器间接寻址）

```
        ……
        cmp   al, 0                          ;AL为逻辑尺
        je    continue
        lea   bx, branch_table
L:      shr   al, 1                          ;逻辑右移
        jnc   add1
        jmp   word ptr [bx]                  ;段内间接转移
add1:   add   bx, type branch_table  ;add bx,2
        jmp   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

（寄存器相对寻址）

```
        ……
        cmp   al, 0
        je    continue
        mov   si, 0
L:      shr   al, 1                    ;逻辑右移
        jnc   add1
        jmp   branch_table[si]         ;段内间接转移
add1:
        add   si, type branch_table
        jmp   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

```
        ……
        cmp   al, 0                          （基址变址寻址）
        je    continue
        lea   bx, branch_table
        mov   si, 7 * type branch_table
        mov   cx, 8
L:      shl   al, 1                          ;逻辑左移
        jnc   sub1
        jmp   word ptr [bx][si]    ;段内间接转移
sub1:   sub   si, type branch_table ;(si)-2
        loop   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

课内测试CH05-3

1. 请在填空中 [填空1] 填写“23”（10分）；
2. 设有子程序r0、r1，其对应的段基地址和偏移地址如下左图所示，通过数据段定义，会建立对应的跳转表如下右图（10分）
   - 存储单元（b_tab+2）的十六进制字数据=[填空2]H
   - 执行指令call word ptr b_tab后IP=[填空3]H

```
(17a6:1234)   r0 proc near   ;子程序0
                  ...
                  ret
              r0 endp
;
(17a6:5678)   r1 proc near   ;子程序1
                  ...
                  ret
              r1 endp
```

```
b_addresses   segment
b_tab         dw   r0
              dw   r1
b_addresses   ends
```

存储器单元
跳转地址表

b_tab	34	17a5:0000
	12	17a5:0001
	?	17a5:0002
	?	17a5:0003

5.3 如何在实模式下发挥80386及其后继机型的优势

本书以实模式为基础来阐述程序设计方法。5.1、5.2节所给出的都是基于8086的程序，由于80X86的兼容性，这些程序都可以在任何一种80X86机型的实模式下运行，而且它们在高档机上运行可比在低档机上运行获得更高的性能。

但是，386及后继机型除提供更大容量、更高的速度和保护模式的支持外，还提供了一些良好的特性，若能在程序设计中充分利用这些优势，将有利于提高编程质量。

5.3 如何在实模式下发挥80386及其后继机型的优势

本节主要内容：

5.3.1 充分利用高档机的32位字长特性

5.3.2 通用寄存器可作为指针寄存器

5.3.3 与比例因子有关的寻址方式

5.3.4 各种机型提供的新指令

5.3.1 充分利用高档机的32位字长特性

- 字长：16位到32位
  - (1) 有利于提高运算精度
  - (2) 提高编程效率
    - 如双字长 加、减法
- 参看

p200 例子5.11 8086指令编程

p201 例子5.12 80386及其后继机型指令编程

5.3.2 通用寄存器可作为指针寄存器

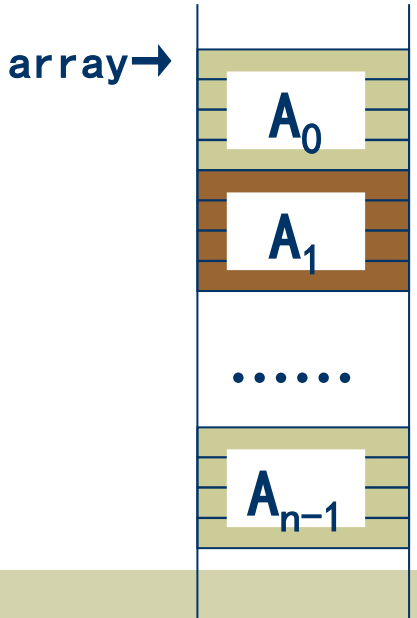
- 386及其后继机型除提供16位寻址外，还提供了32位寻址。在实模式下，这两种寻址方式可同时使用。 在使用32位寻址时，32位通用寄存器可以作为基址或者变址寄存器使用——即32位通用寄存器可以作为指针寄存器用
- 在实模式下段的大小被限制为64KB，这样，段内的偏移寻址范围为0000~ffffh，所以当32位通用寄存器用作指针寄存器时，应该注意它们的高16位应该为0

注意：32位寻址时32位通用寄存器可用作指针寄存器，但16位寻址方式中可用的16位通用寄存器中仍然只有BX、BP、SI和DI可用作指针寄存器

5.3.3 与比例因子有关的寻址方式

- 支持与比例因子有关的三种寻址方式
  - 在表格处理和多维数组处理处理时很有用
- 参看 P204 例5.13－－－比例变址寻址方式

```
        sub  ebx, ebx
        ......
Back:   add  eax, array[ebx*4]
        ......
        inc  ebx
        jnz  back
```



- 其他例子自学

5.3.4 各种机型提供的新指令

- 在80X86系列中，新机型的产生往往为用户提供了一些新指令或对原有指令的扩充，因此，在编程时可利用新机型所提供的这些有利条件。有关指令在第三章已经做了说明

- 在此，归纳便于利用

  P207~208

作　业

5.3　　5.7　　5.12　　5.13

5.14　　5.15　　5.19

5.24(假设键盘输入的歌曲编号已经在AL中)

计算N!(循环程序结构)

第六章 子程序结构

- 子程序又称为过程
  - 它相当于　级语言中的过程和函数
- 为什么需要子程序
  - 程序段共享
  - 模块化设计
  - 简化程序设计
  - 节省存储空间
- 使用子程序的主要优点
  - 节省存储空间
  - 减少程序设计时间

本章的主要内容

6.1 子程序的设计方法

6.2 嵌套与递归子程序

6.3 子程序举例

6.4 DOS系统功能调用

6.1 子程序的设计方法

6.1.1 过程（子程序）定义

- 定义语句是伪操作，只告诉汇编程序如何处理，生成合适的机器指令

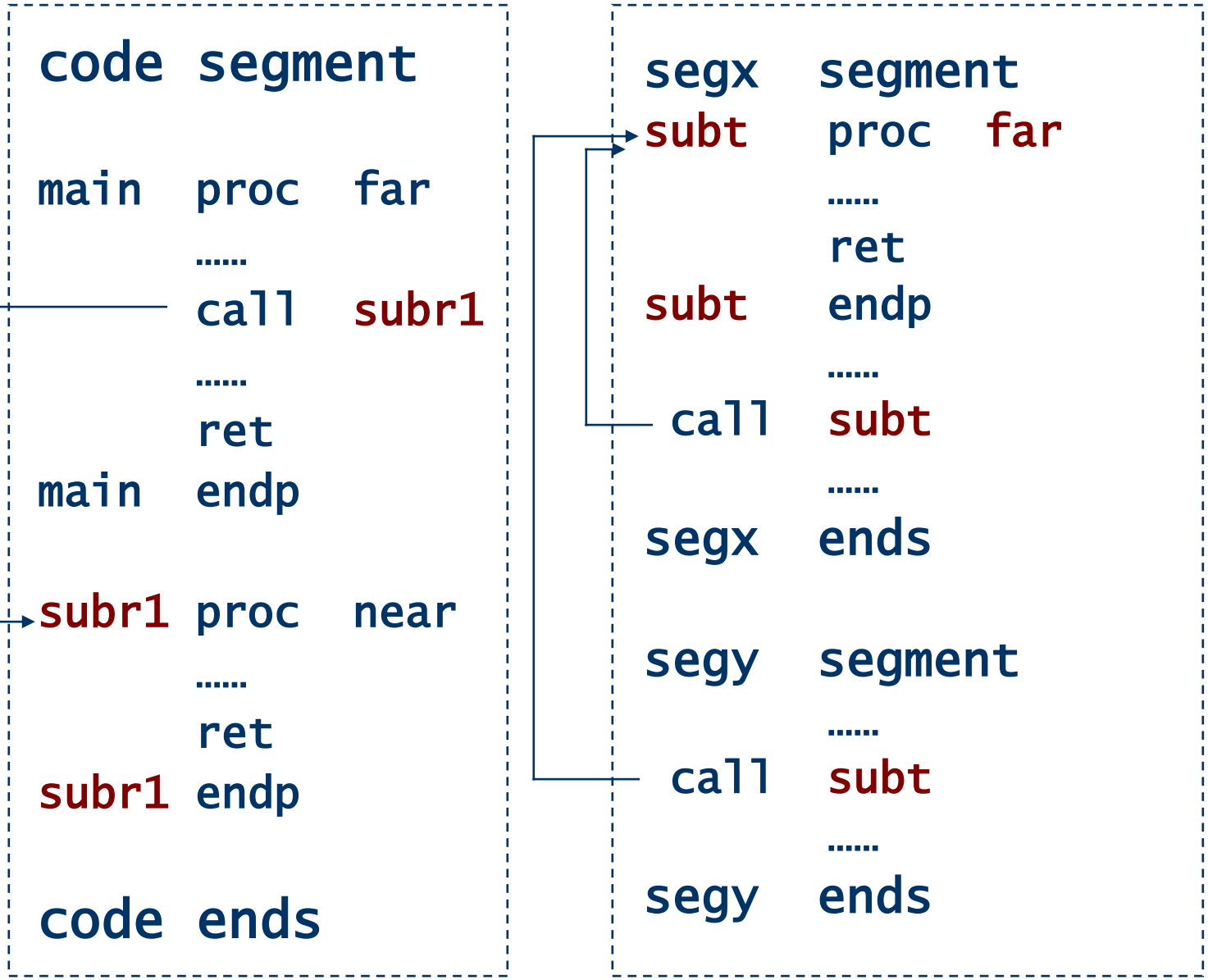
- 格式：

```
procedure name    PROC    Attribute
          ......
procedure name    ENDP
```

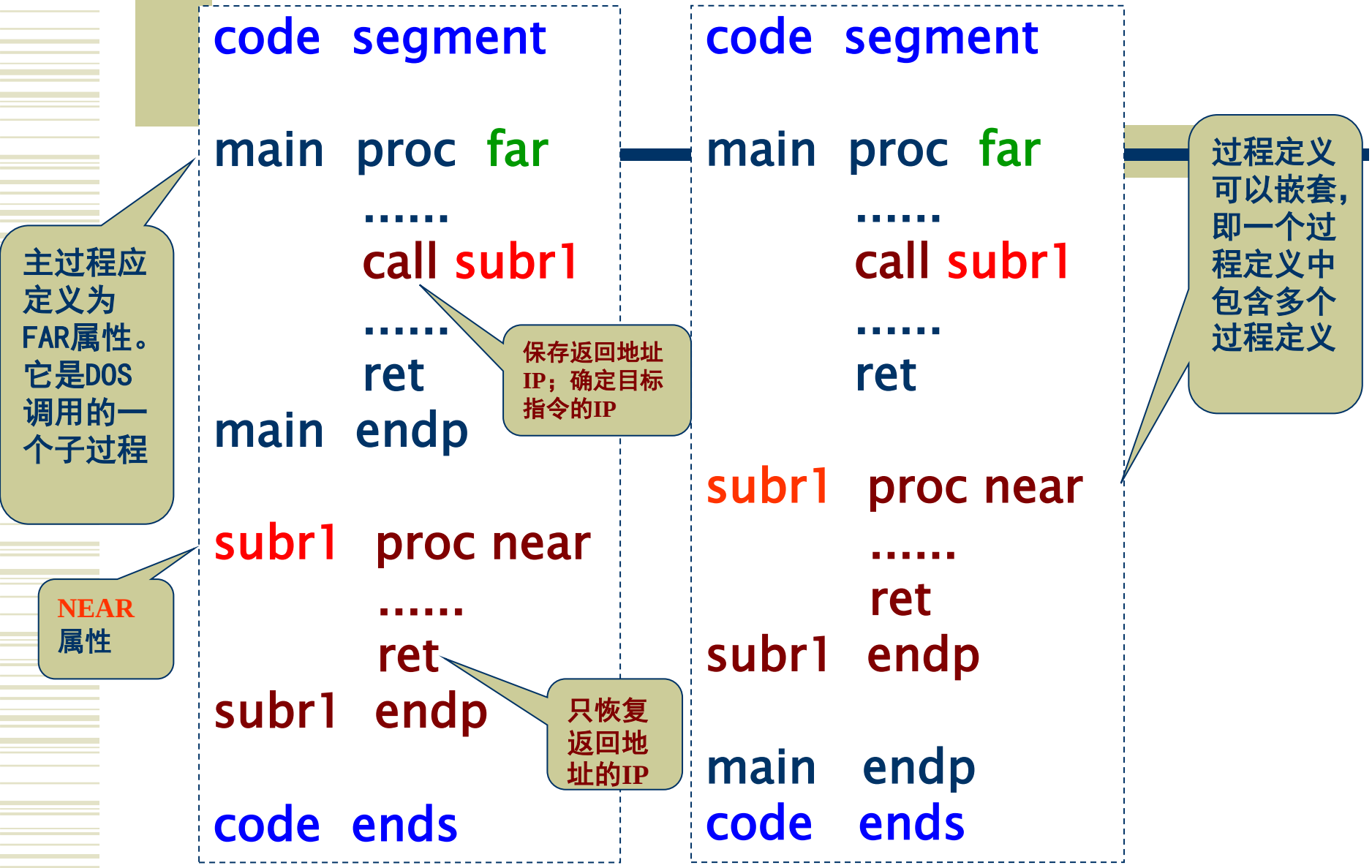
  - 过程名（procedure name）是子程序入口的符号地址
  - 类型属性（Attribute）：NEAR、 FAR

- **过程属性的确定原则：**
  - NEAR属性：调用程序和子程序在同一代码段中（段内调用）
  - FAR属性：调用程序和子程序不在同一代码段中（段间调用）

- 80X86汇编程序在汇编时**用过程属性确定CALL和RET指令属性**
  - 用户只需在定义过程时考虑属性， CALL和RET指令属性可以不考虑让汇编程序确定


code segment
main proc far
……
call subr1
……
ret
main endp
subr1 proc near
……
ret
subr1 endp
code ends
segx segment
subt proc far
……
ret
subt endp
……
call subt
……
segx ends
segy segment
……
call subt
……
segy ends

例6.1　调用程序和子程序在同一代码段中



例6.2　调用程序和子程序不在同一代码段中

```
segx   segment
subt   proc  far
       ......
       ret
subt   endp
       ......
       call  subt
       ......
segx   ends

segy   segment
       ......
       call  subt
       ......
segy   ends
```

FAR属性

恢复返回地址的IP，CS

保存返回地址CS, IP；
确定目标指令的CS, IP

FAR属性子程序可以被同一段内或不同段内程序调用，而NEAR属性子程序只能被同一段内程序调用

6.1.2 子程序调用和返回

子程序调用：隐含使用堆栈保存返回地址

call near ptr subp

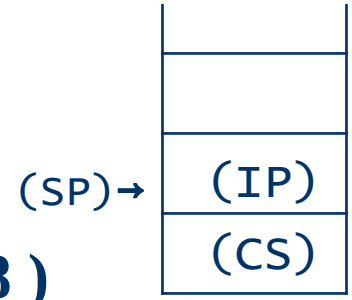
(1) 保存返回地址

(2) 转子程序

call far ptr subp

(1) 保存返回地址

(2) 转子程序



子程序返回：ret (或者 ret imm8)

6.1.3 保存和恢复寄存器

- 为什么子程序中要保存和恢复寄存器
  - 子程序是独立的共享模块，对寄存器使用具有独立性，这样会产生主程序和子程序使用寄存器冲突
  - 为了解决主程序和子程序使用寄存器冲突，保证主程序正确运行，子程序中必　保存相关使用的寄存器

- (1) 保护和恢复寄存器的方法
  - 子程序开始时，使用PUSH指令保存
  - 子程序返回前，使用POP指令恢复
  - 保存和恢复次序应该相反

子程序设计时应特别注意正确使用堆栈，及堆栈状态变化。一般情况下，子程序中PUSH和POP指令必　配对使用！

```
subt    proc    far

        push    ax
        push    bx
        push    cx
        push    dx
        ……
        ……
        pop     dx
        pop     cx
        pop     bx
        pop     ax

        ret

subt    endp
```

(2) 确定保护哪些寄存器的原则

- 保护子程序中将要使用的寄存器及标志寄存器即可
  - 子程序独立性强，不了解调用程序的寄存器使用情况
  - 如果了解调用程序的寄存器使用情况，可适量保存
- 用寄存器向主程序回送结果的寄存器不必保存
- **FLAGS寄存器保存优先，恢复时最后恢复**

课内测试CH06-1

1. 请在填空中 [填空1] 填写“68”（10分）；

2. 汇编程序汇编时，根据过程属性确定子程序返回机器指令等，过程属性的确定原则：（10分）
   - 调用程序和子程序在同一代码段中，则属性是 [填空2]
   - 调用程序和子程序不在同一代码段中，则属性是 [填空3]

6.1.4 子程序的参数传送

- 参数传送：调用程序和子程序之间的信息传送
  - 调用时，主程序传送参数给子程序
  - 返回时，子程序返回参数给主程序

- 参数传送的一般途径
  - 寄存器
  - 存储器

参数传送的具体方法：

（1）通过寄存器传送参数

（2）通过存储器传送参数

*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

*子程序和调用程序不在同一程序模块中（ *13章* ）

（3）通过地址表传送参数地址

（4）通过堆栈传送参数或参数地址

（1）通过寄存器传送参数

- 这种传递方式使用方便，适用于参数较少的情况

例6.3 十进制到十六进制的转换程序

（从键盘取得一个十进制数,然后把该数以十六进制形式在屏幕显示）

```
decihex  segment
         assume  cs: decihex

main     proc   far
         push   ds
         sub    ax, ax
         push   ax
repeat:  call   decibin   ; 从键盘取10进制数，10→2，保存在BX中
         call   crlf      ; 显示回车换行，防止屏幕显示重叠
         call   binihex   ; 2→16，并在屏幕上显示
         call   crlf
         jmp    repeat
         ret
main     endp
```

通过寄存器BX传送参数

从键盘取10进制数，10→2，保存在BX中

```
decibin  proc  near
         mov   bx, 0    ; bx初始化
newchar:
         mov   ah, 1
         int   21h      ; 从键盘取10进制数键的ASCII码
         sub   al, 30h  ; 0-9的ASCII码30-39
         jl    exit     ; <0退出
         cmp   al, 9d
         jg    exit     ; >9退出
         cbw            ; AL符号扩展到AH

         xchg  ax, bx
         mov   cx, 10d
         mul   cx
         xchg  ax, bx
         add   bx, ax
         jmp   newchar
exit:    ret            ;返回的10进制数的二进制数在BX中
decibin  endp
```

10进制数以四位2进制数形式保存在BX中

•BX中2进制数→16进制数，并在屏幕上显示

```
binihex  proc  near    ; 要显示的二进制数在BX中
         mov   ch, 4
rotate:
         mov   cl, 4
         rol   bx, cl
         mov   al, bl
         and   al, 0fh
         add   al, 30h
         cmp   al, 3ah
         jl    printit
         add   al, 7h
printit:
         mov   dl, al
         mov   ah, 2
         int   21h
         dec   ch
         jnz   rotate
         ret
binihex  endp
```

16进制数转换成ASCII码

调用DOS功能在屏幕上显示1个字符
请参看605页附录4约定

•显示回车换行

```
crlf    proc   near
        mov    dl, 0dh  ;  “回车”的ASCII码=0dH
        mov    ah, 2
        int    21h
        mov    dl, 0ah  ;  “换行”的ASCII码=0aH
        mov    ah, 2
        int    21h
        ret
crlf    endp

decihex ends          ; 程序代码在一个代码段中与前边“decihex  segment ”配对
        end   main    ; 程序从main开始执行
```

（2）通过存储器直接传送访问参数

- 子程序和调用程序在同一程序模块中，则子程序象主程序一样直接访问数据段中的变量

例6.4 累加数组中的元素

```
data   segment
       ary     dw   1,2,3,4,5,6,7,8,9,10
       count   dw   10
       sum     dw   ?
data   ends
code   segment
main   proc  far
       assume cs:code, ds:data
start:
       push   ds
       sub    ax, ax
       push   ax
       mov    ax, data
       mov    ds, ax
       call   near ptr   proadd
       ret
main   endp
```

```
proadd  proc  near
        push  ax
        push  cx
        push  si
        lea     si, ary
        mov   cx, count
        xor     ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next
        mov   sum, ax
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
code    ends
        end  start
```

问 ：假设数据段定义如下

```
data    segment

ary     dw   1,2,3,4,5,6,7,8,9,10
count   dw   10
sum     dw   ?

ary1    dw   10,20,30,40,50,60,70,80,90
count1  dw   9
sum1    dw   ?

data    ends
```

```
proadd  proc  near
        push  ax
        push  cx
        push  si
        lea   si, ary
        mov   cx, count
        xor   ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next
        mov   sum, ax
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
```

* 如果直接访问内存变量，那么累加数组ary和数组ary1中的元素，由于处理的存储单元在子程序中有固定的约定，不能用同一个子程序proadd。多编写几个子程序？

解决办法：

1、设置共享的临时参数存放区，调用时主程序先将参数放在临时存放区，子程序处理临时参数存放区中数据，主程序效率不

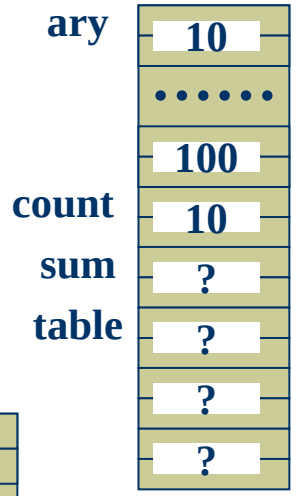
2、调用时主程序只传送变量地址表给子程序

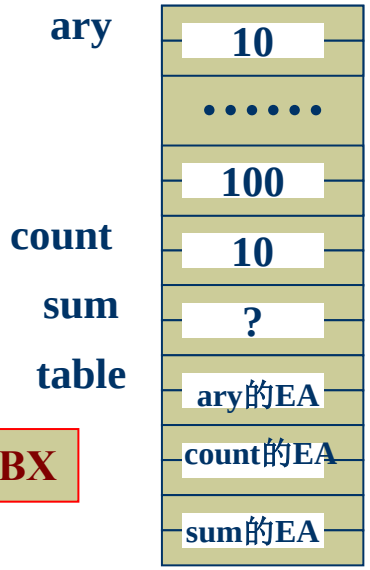
（3）通过地址表传送变量地址

- 适用于参数较多的情况。具体方法是先建立一个地址表，该表由参数地址构成。然后把表的　地址通过寄存器或堆栈传递给子程序

例6. 4 累加数组中的元素

```
data  segment
    ary     dw  10,20,30,40,50,60,70,80,90,100
    count   dw  10
    sum     dw  ?
    table   dw  3  dup  (?)        ; 地址表，存放ary,count,sum的EA
data  ends
code  segment
  main  proc  far
        assume cs:code, ds:data
  start: push  ds
        sub   ax, ax
        push  ax
        mov   ax, data
        mov   ds, ax
        mov   table,    offset  ary
        mov   table+2,  offset  count
        mov   table+4,  offset  sum
        mov   bx,       offset  table
        call  proadd
        ret
    main  endp
```





table的EA→BX

将立即数（变量count的有效地址）送到table+2给出的直接有效地址的两个存储器单元

请解释 mov table+2, offset count 的功能

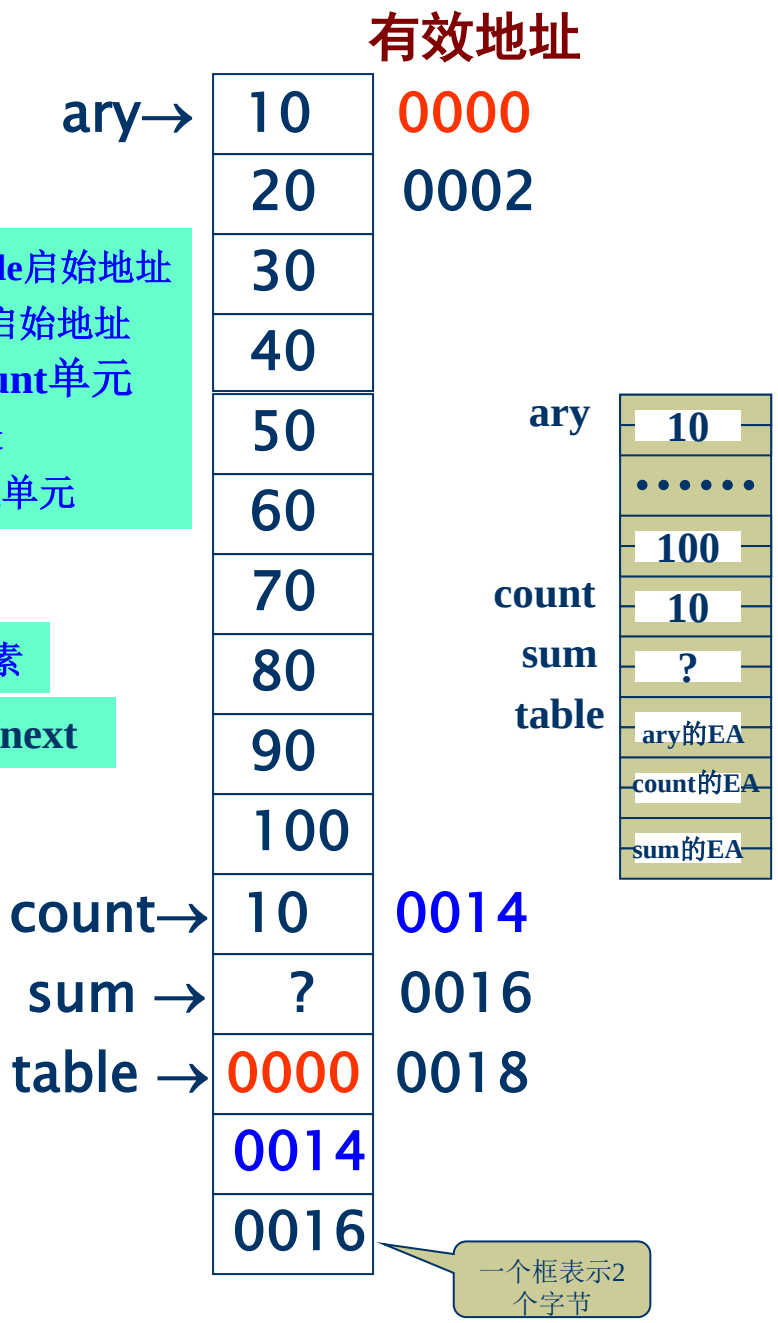
```
proadd  proc  near
        push  ax
        push  cx
        push  si
        push  di
        mov   si, [bx]
        mov   di, [bx+2]
        mov   cx, [di]
        mov   di, [bx+4]
        xor   ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next
        mov   [di], ax
        pop   di
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
code    ends
        end   start
```

table的EA在BX中

bx=0018H 指向table启始地址
si=0000H 指向ary启始地址
di=0014H 指向count单元
xx=10 计数值count
di=0016H 指向sum单元

si指向ary的下一个元素

cx=cx-1; 如cx≠0, 转next

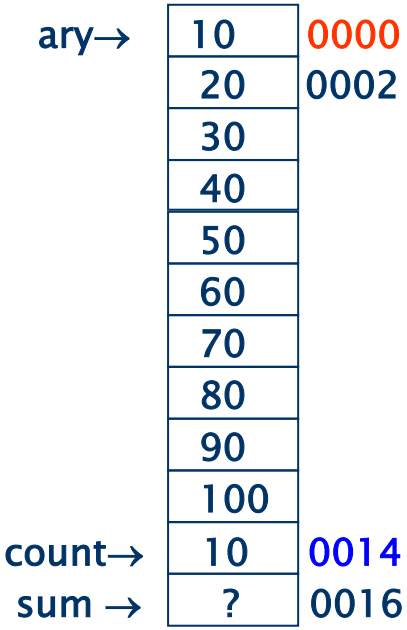


（4）通过堆栈传送变量或变量地址

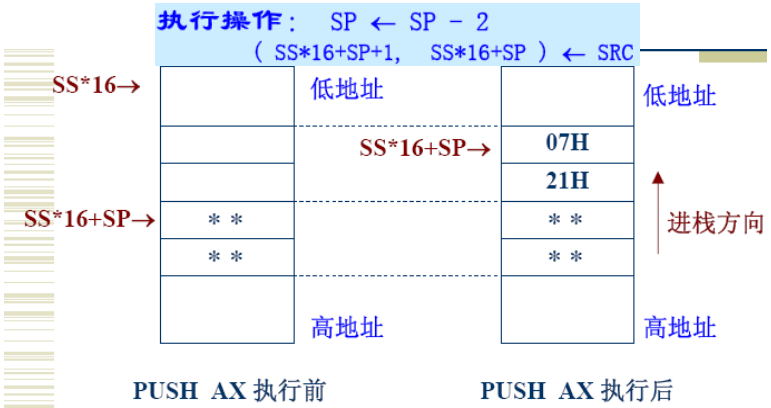
- 步　：
  1. 主程序把参数或参数地址压入堆栈；
  2. 子程序使用堆栈中的参数或通过栈中参数地址取到参数；
  3. 子程序返回时使用RET n指令调整SP指针，以便删除堆栈中已用过的参数，保持堆栈平衡，保证程序的正确返回。

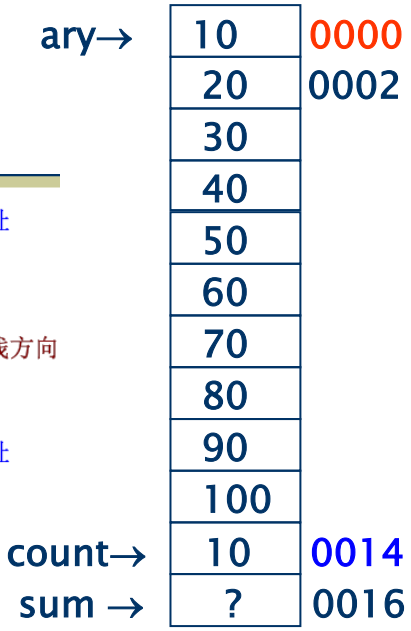
例6.4 累加数组中的元素

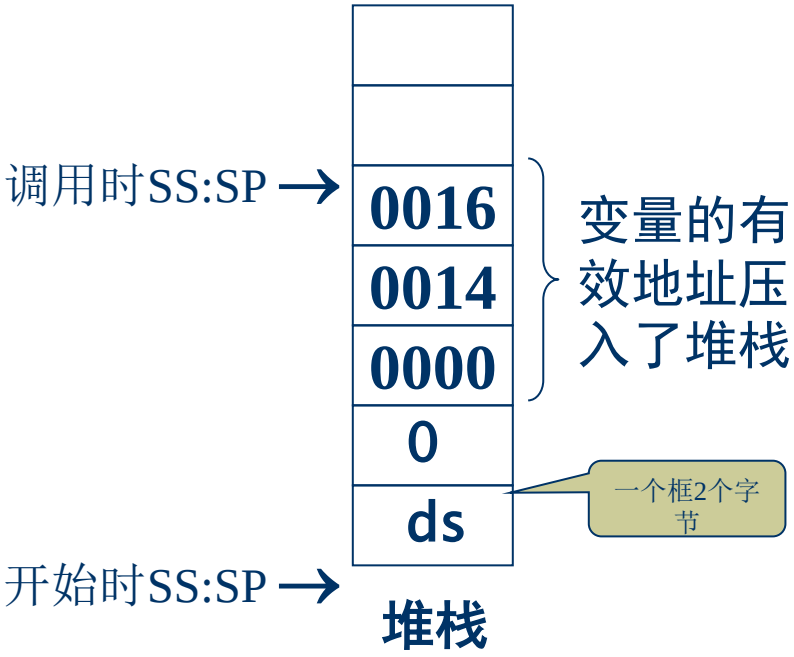
```
data  segment
      ary     dw   10,20,30,40,50,60,70,80,90,100
      count   dw   10
      sum     dw   ?
data  ends

stack  segment
         dw     100  dup  (?)
  tos   label   word
stack  ends
```



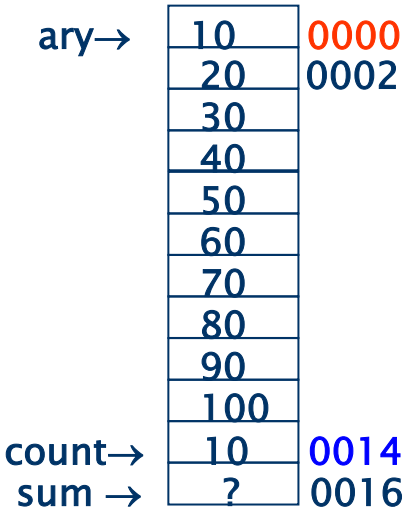
```
code1 segment
main  proc  far
      assume  cs:code1, ds:data, ss:stack
start:
      mov   ax,  stack
      mov   ss,  ax
      mov   sp,  offset  tos
      push  ds
      sub   ax,  ax
      push  ax
      mov   ax,  data
      mov   ds,  ax
      mov   bx, offset  ary
      push  bx
      mov   bx, offset  count
      push  bx
      mov   bx, offset  sum
      push  bx
       call    far ptr  proadd
       ret
main   endp
code1 ends
```

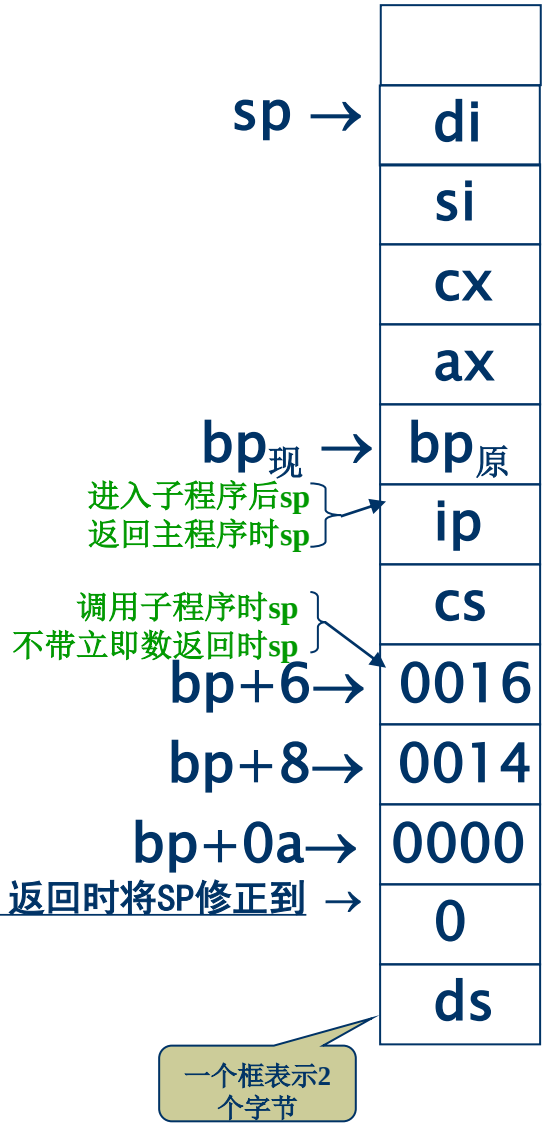






```
code2 segment
  assume  cs: code2
proadd  proc  far

    push  bp
    mov   bp, sp

    push  ax
    push  cx
    push  si
    push  di

    mov   si, [bp+0ah]
    mov   di, [bp+8]
    mov   cx, [di]
    mov   di, [bp+6]

    xor   ax, ax
next:
    add   ax, [si]
    add   si, 2
    loop  next
    mov   [di], ax

    pop   di
    pop   si
    pop   cx
    pop   ax

    pop   bp

    ret   6
proadd  endp
code2   ends
        end  start
```





课内测试CH06-2

1. 请在填空中 [填空1] 填写“69”（10分）；

2. 主程序和子程序之间参数传送，通过将参数放在它们都可访问的地方实现。因此参数传送的一般途径有：（10分）

   ① 通过 [填空2]

   ② 通过 [填空3]

■ 结构伪操作STRUC：定义一种可包含不同类型数据的结构模式，只有具体使用时才有对应存储单元的具体含义

格式：　结构名　STRUC

　　字段名1　DB　?

　　字段名2　DW　?

　　字段名3　DD　?

　　……

　　结构名　ENDS

□字段名就是变量名，可用变量名表示字段起始地址

例：学生个人信息

```
STUDENT_DATA   STRUC     ; 4个字段,18个字节的结构模式
NAME      DB    5 DUP(?)
ID        DW    0
AGE       DB    ?
DEP       DB   10 DUP(?)
STUDENT_DATA   ENDS
```

■ **结构　置语句**：为结构中各字段的数据分配存储器单元，并可为存储单元重新输入字符串和数值

格式1：　变量名　结构名　< >

•采用结构定义中的赋值

格式2：　变量名　结构名　<　赋值说明>

•重新定义结构中的值

数据段定义中使用

例：S991000　　STUDENT_DATA　　< >

S991001　　STUDENT_DATA　　< ,1001,22, >

STUDENT　　STUDENT_DATA　　100　DUP (< >)

等同于数据段中如下定义：

```
S991000.NAME    DB   5 DUP(?)
S991000.ID      DW   0
S991000.AGE     DB   ?
S991000.DEP     DB  10 DUP(?)
```

■ **访问结构数据变量方法**：

```
MOV   AL,  S991000.NAME[SI]
MOV   AL,  [BX].NAME[SI]
```

```
STUDENT_DATA  STRUC
  NAME    DB   5 DUP(?)
  ID      DW   0
  AGE     DB   ?
  DEP     DB  10 DUP(?)
STUDENT_DATA  ENDS
```

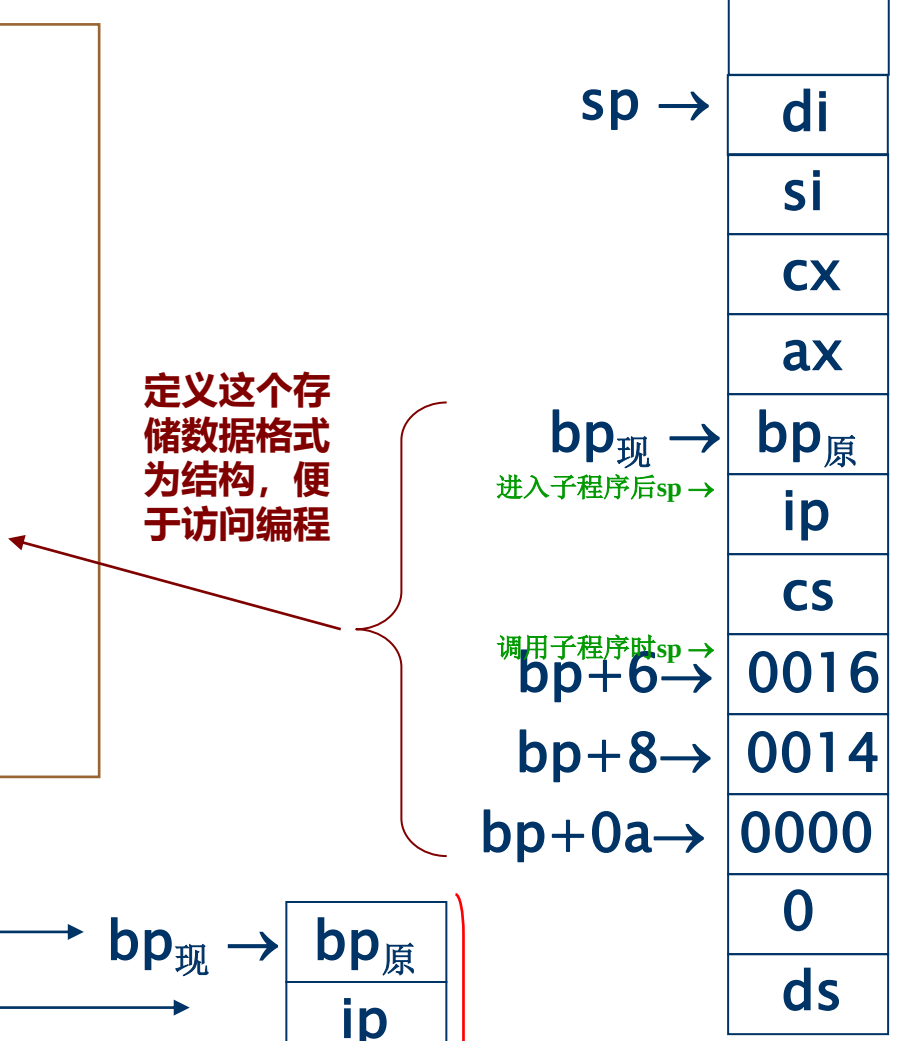
.name可以理解为相对于结构　址的位移量
bx中存的是结构　址，　si给出name字段的第几$
[BX].NAME[SI]= [BX+.NAME+SI]

结构伪操作举例：

改写例6.4 累加数组中的元素

```
stack_strc      struc

    save_bp     dw    ?
    save_cs_ip  dw    2 dup (?)
    par3_addr   dw    ?
    par2_addr   dw    ?
    par1_addr   dw    ?

stack_strc      ends
```

定义这个存储数据格式为结构，便于访问编程



让bp指向结构 址 [bp].save_bp → bp现 → bp原

[bp].save_cs_ip → ip

cs

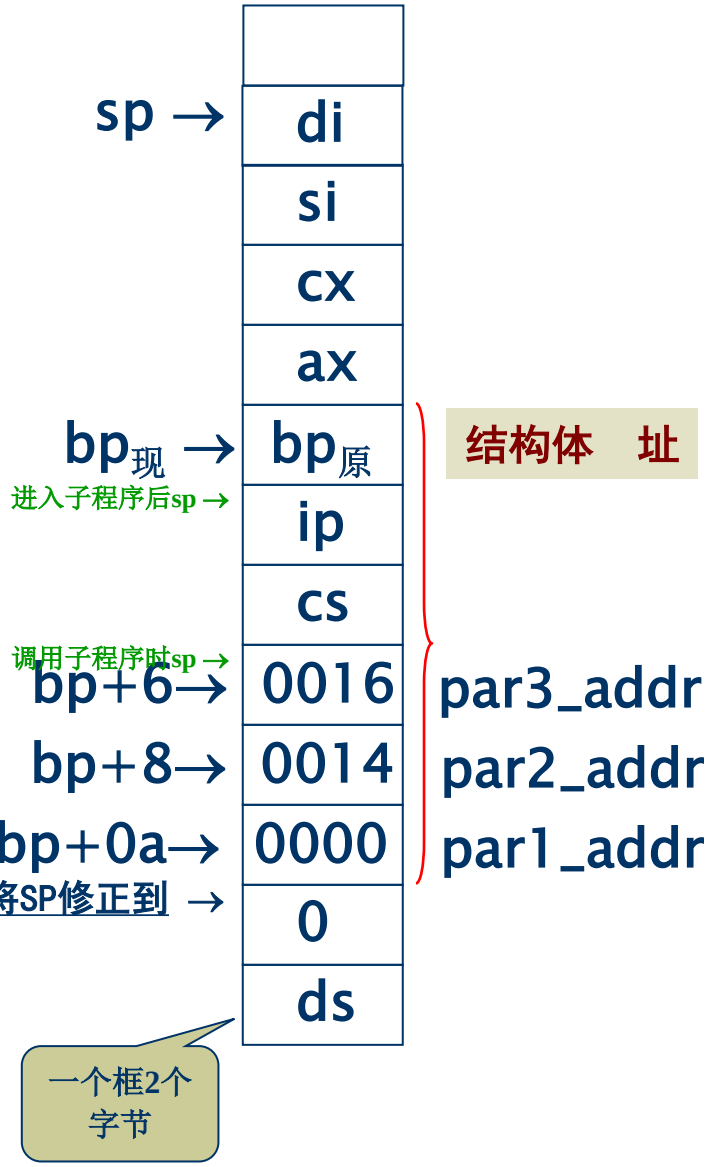
[bp].par3_addr → bp+6→ 0016 par3_addr

[bp].par2_addr → bp+8→ 0014 par2_addr

[bp].par1_addr → bp+0a→ 0000 par1_addr

```
stack_strc      struc
    save_bp     dw    ?
    save_cs_ip  dw    2 dup  (?)
    par3_addr   dw    ?
    par2_addr   dw    ?
    par1_addr   dw    ?
stack_strc      ends

proadd proc  far
    push  bp
    mov   bp, sp
    push  ax
    push  cx
    push  si
    push  di
    mov   si, [bp].par1_addr
    mov   di, [bp].par2_addr
    mov   cx, [di]
    mov   di, [bp].par3_addr
    xor   ax, ax
next:
    add   ax, [si]
    add   si, 2
    loop  next
    mov   [di], ax
    pop   di
    pop   si
    pop   cx
    pop   ax
    pop   bp
    ret   6
proadd  endp
```

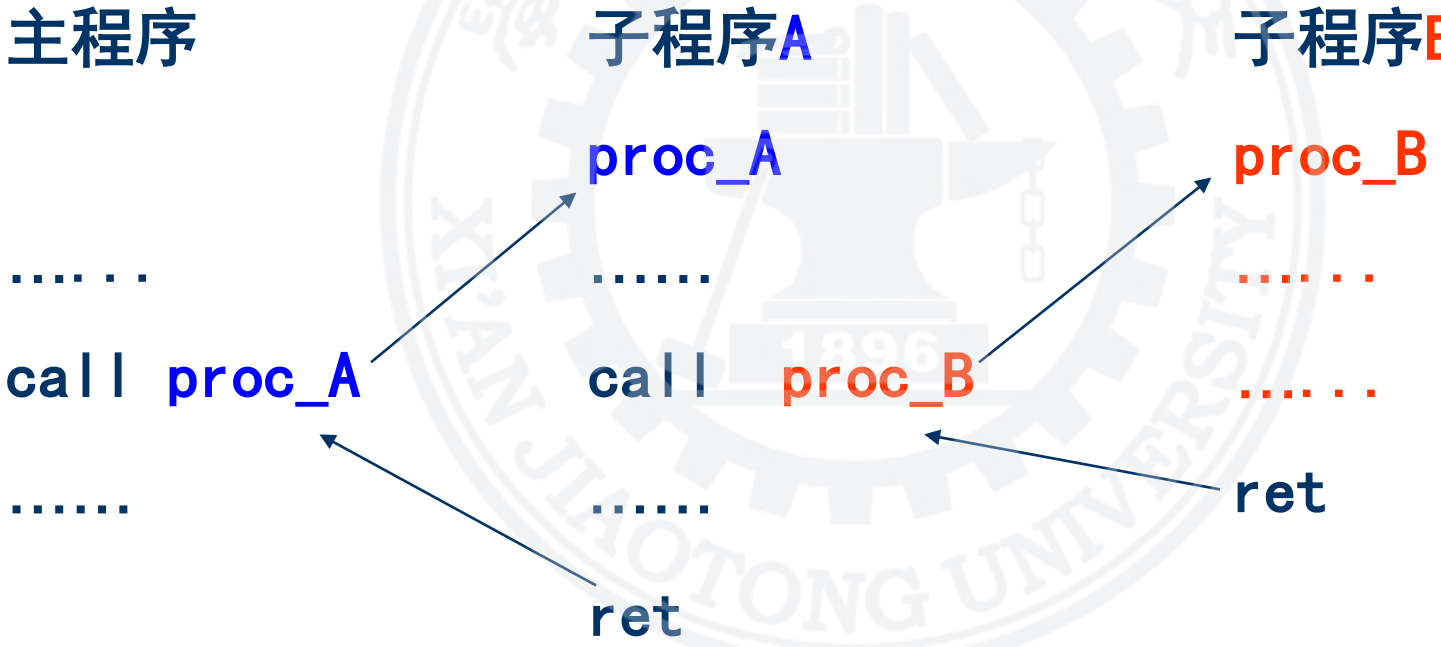
bp指向结构体　址



6.2 嵌套与递归子程序

6.2.1 子程序的嵌套

■子程序嵌套：一个子程序作为调用程序调用另一个子程序



6.2.1 递归子程序

- 递归子程序
  - 递归调用：子程序调用的子程序是它自身
  - 递归子程序：递归调用中的子程序
  - 是子程序嵌套的特殊情况

但实际中由堆栈大小和现场保护情况决定

- 嵌套深度：是嵌套的层次，层次不限
  - 堆栈大小是嵌套深度的关键因素，特别是递归调用
  - 特别注意堆栈状态和正确使用
- 注意事$ 同一般子程序调用

例6.7 计算N! (N≥0)

N!=N× (N-1) ×(N-2) × … × 1

递归定义：

$$\begin{cases} 0! = 1 \\ N! = N \times (N-1)! \quad N>0 \end{cases}$$

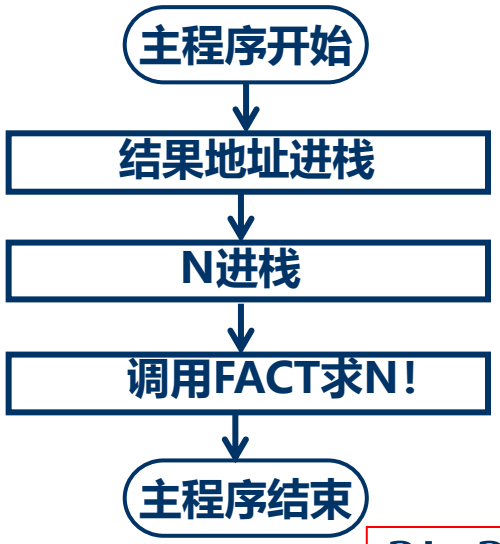
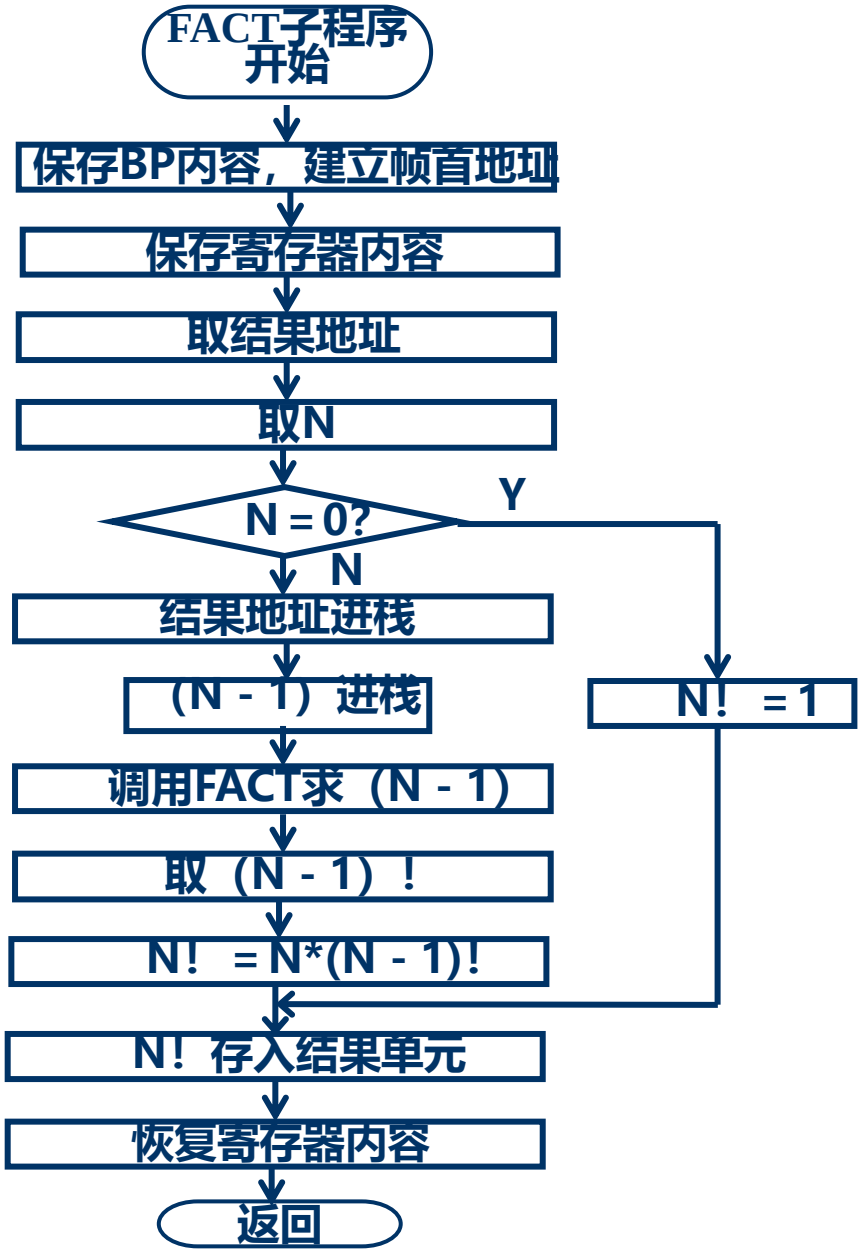


图6. 7

3!=3*2!

2!=2*1!

1!=1*0!



例6.7 计算n! 假设n=3

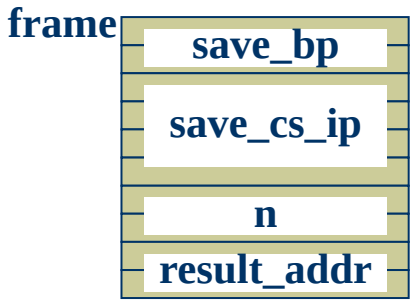
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends

data   segment
n_v    dw    3
result  dw    ?
data   ends

stack  segment
       dw  128  dup (0)
   tos label  word
stack  ends
```

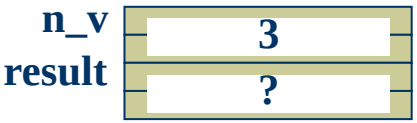
3!=3*2!

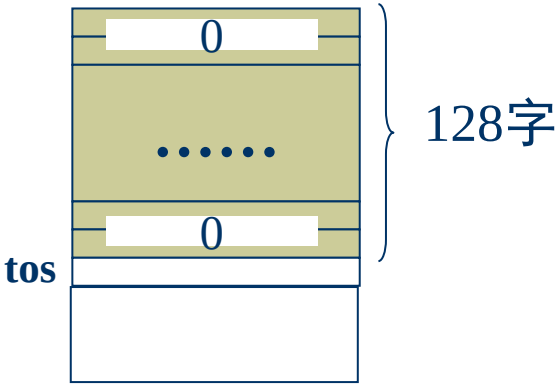
2!=2*1!

1!=1*0!

0!=1

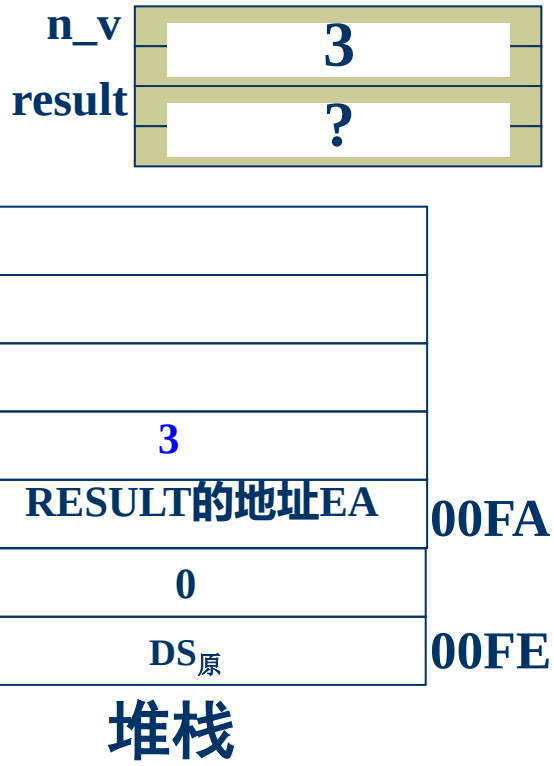






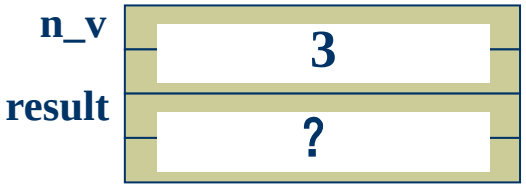
```
code segment
main  proc  far
        assume  cs:code, ds:data, ss:stack
start:
      mov  ax, stack
      mov  ss, ax
      mov  sp, offset tos
      push  ds
      sub  ax, ax
      push  ax
      mov  ax, data
      mov  ds, ax
      mov  bx, offset result
      push  bx
      mov  bx, n_v
      push  bx
      call  far  ptr  fact
      ret
main endp
code  ends
```

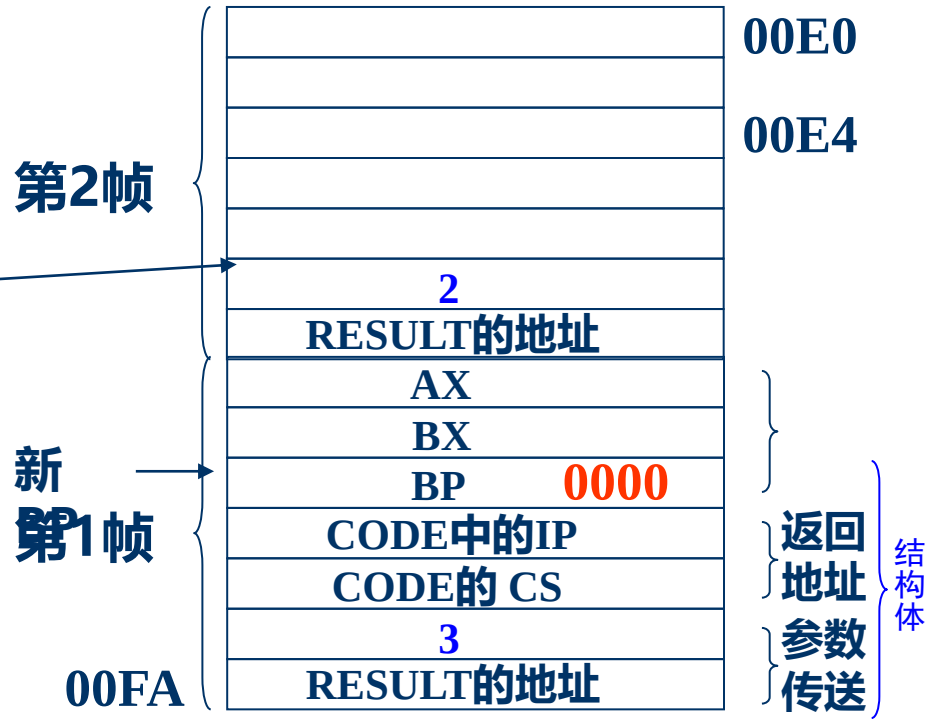
```
data    segment
n_v     dw    3
result  dw    ?
data    ends
```



```
frame  struc
       save_bp      dw    ?
       save_cs_ip   dw    2 dup (?)
       n            dw    ?
       result_addr  dw    ?
frame  ends
```

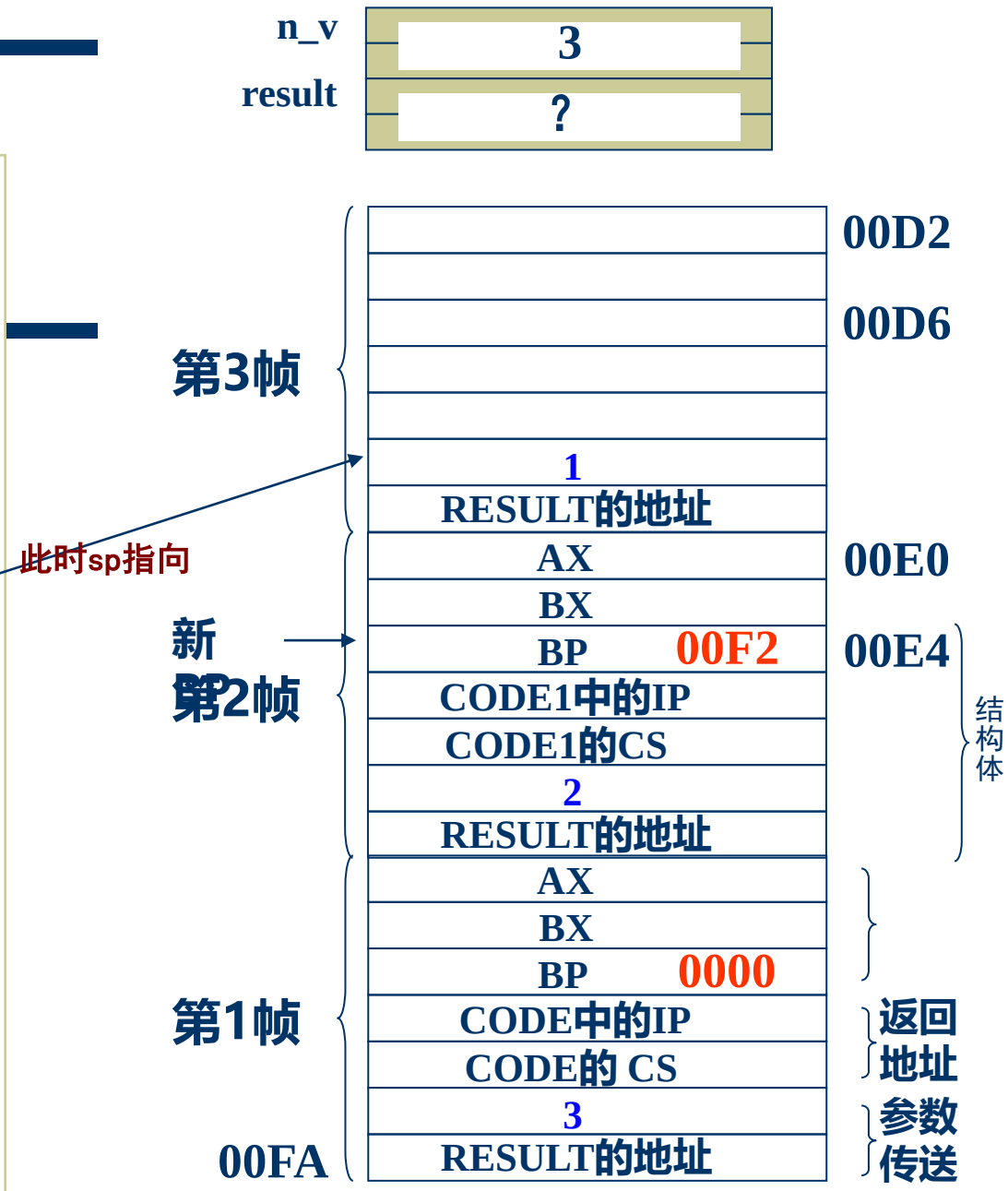
```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```





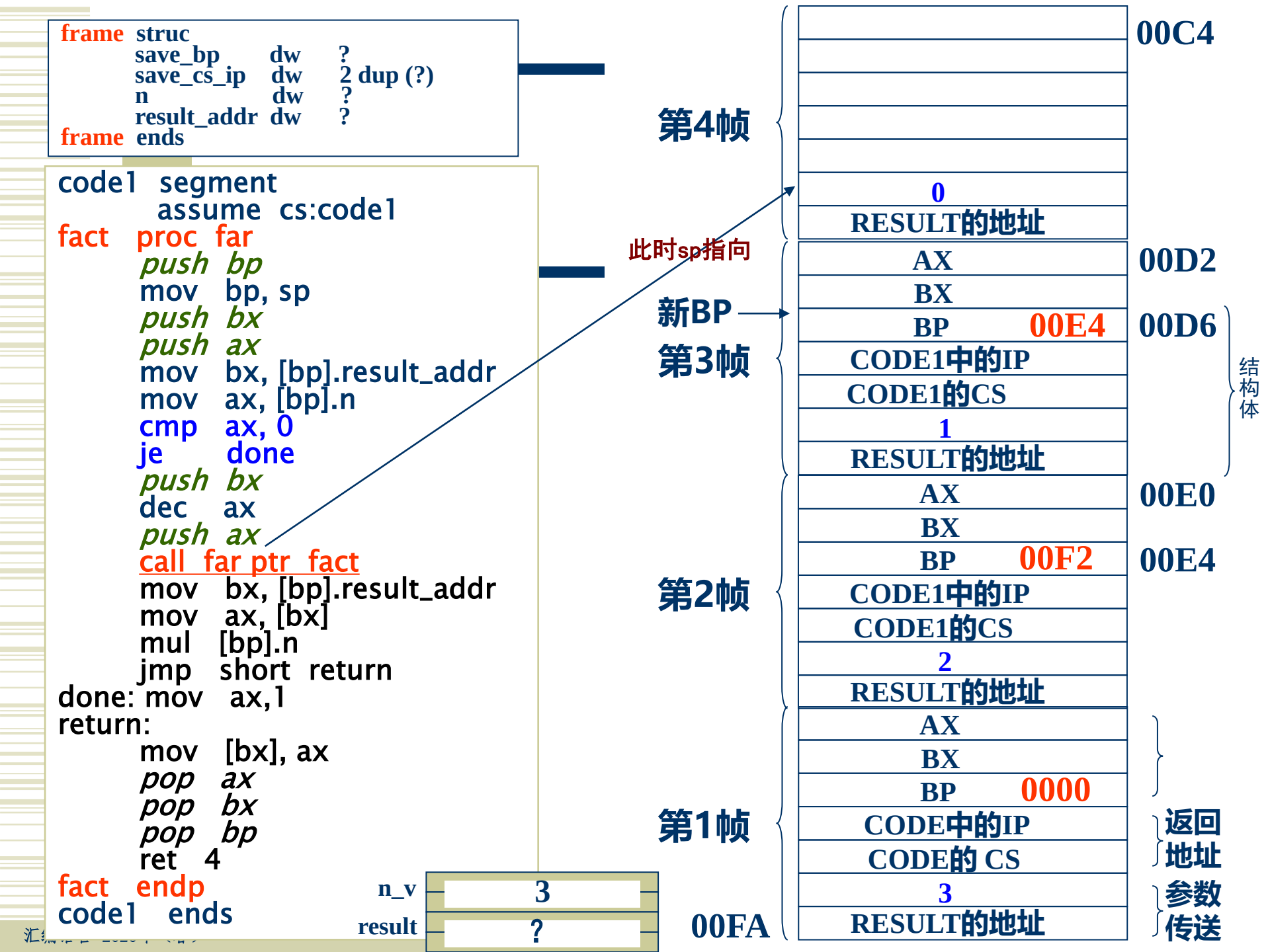
```
frame  struc
       save_bp     dw    ?
       save_cs_ip  dw    2 dup (?)
       n           dw    ?
       result_addr dw    ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je    done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done:  mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1  ends
```



```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

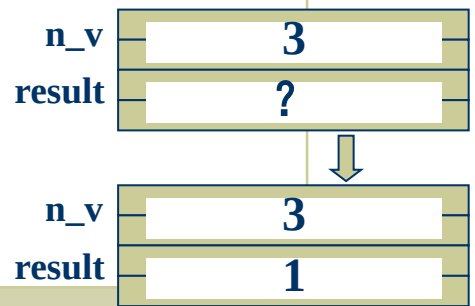


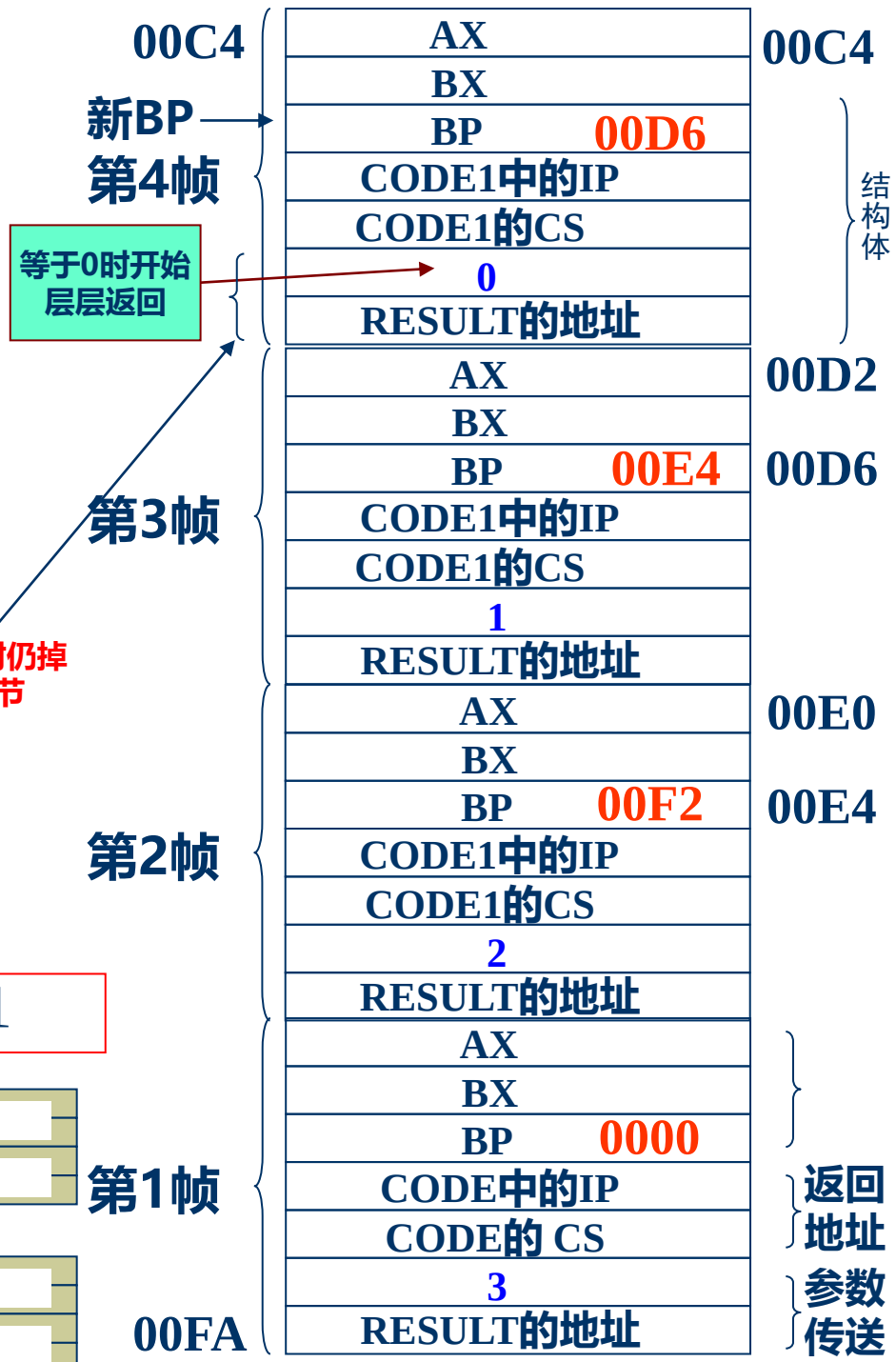
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

返回时仍掉4个字节

0!=1

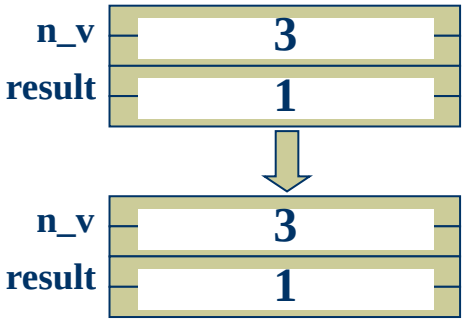




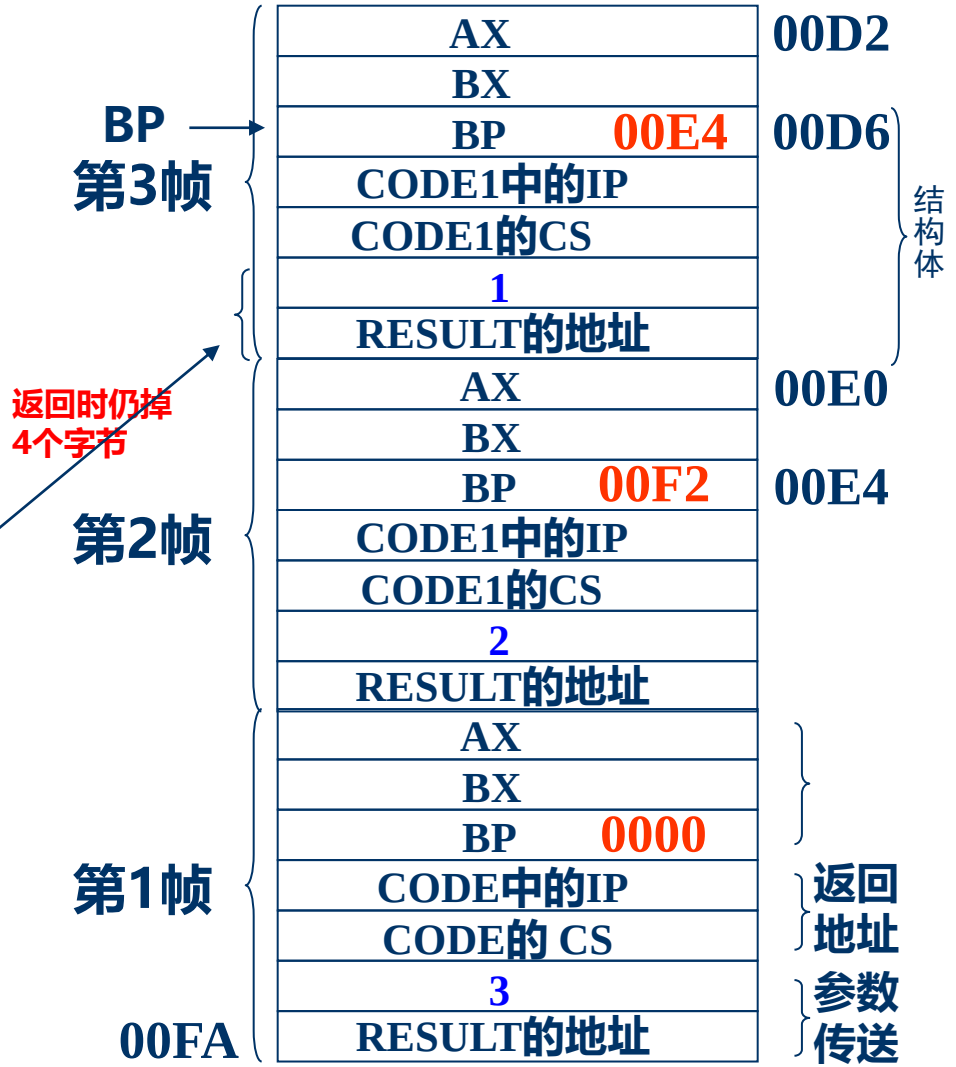
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

1!=1*0!

0!=1




```
code1  segment
       assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je    done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done:  mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1  ends
```



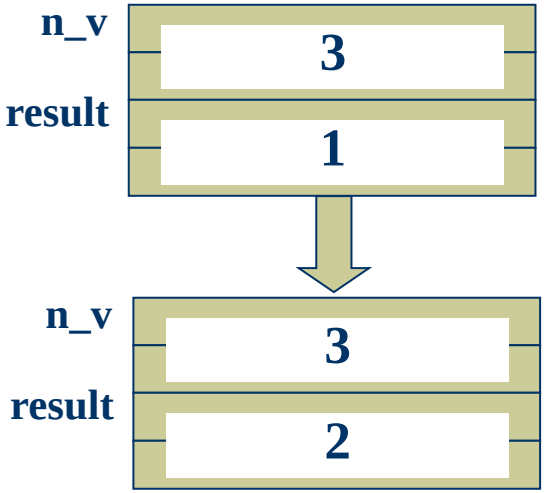
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

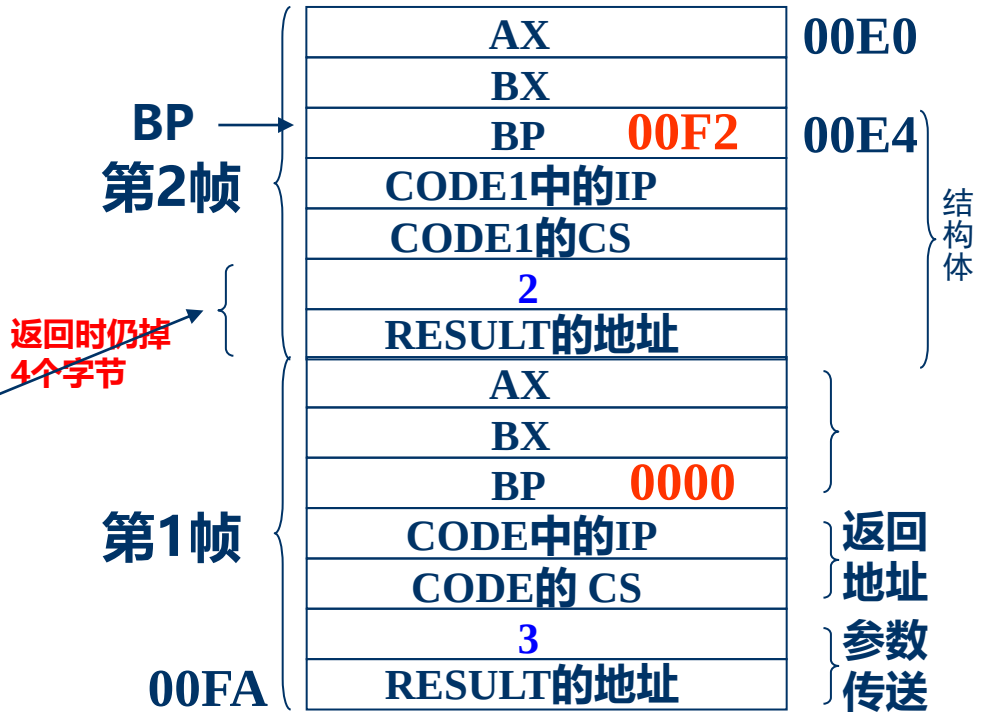
2!=2*1!

1!=1*0!

0!=1

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```





```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

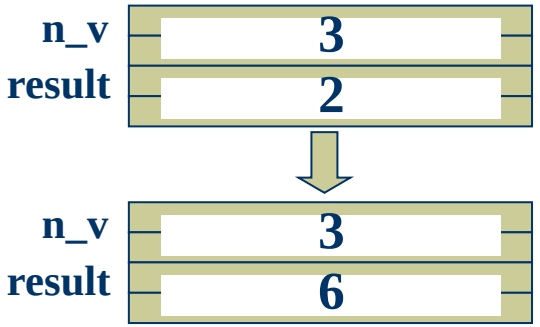
```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

3!=3*2!

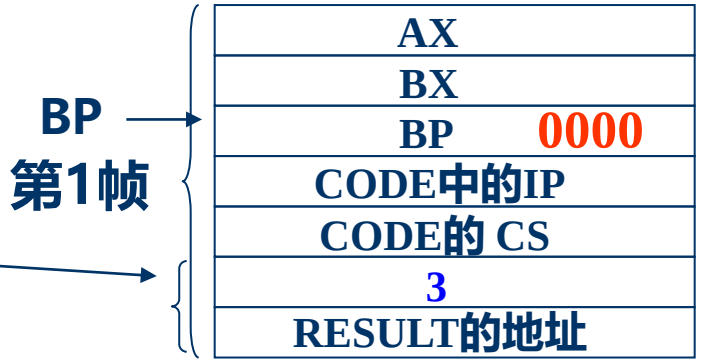
2!=2*1!

1!=1*0!

0!=1

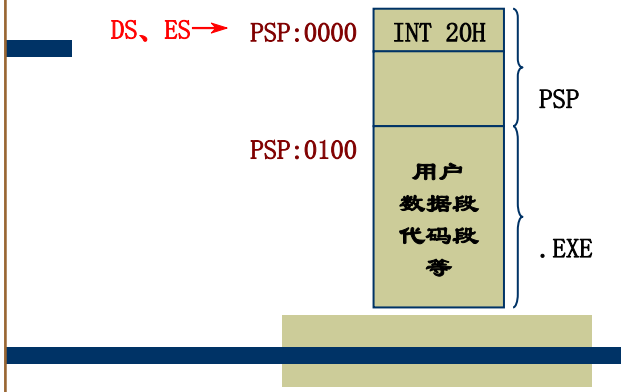


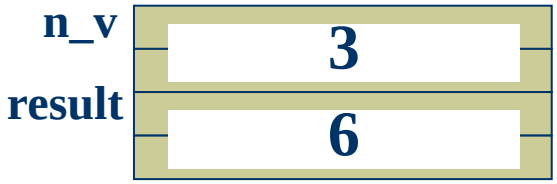
返回时仍掉
4个字节

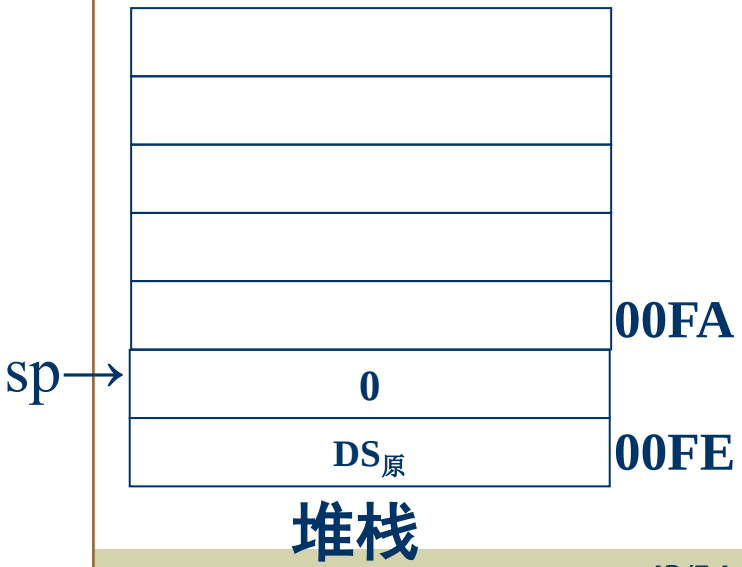


```
code segment
main  proc  far
        assume  cs:code, ds:data, ss:stack
start:
      mov  ax, stack
      mov  ss, ax
      mov  sp, offset tos
      push  ds
      sub  ax, ax
      push  ax
      mov  ax, data
      mov  ds, ax
      mov  bx, offset result
      push  bx
      mov  bx, n_v
      push  bx
      call  far  ptr  fact
      ret
main endp
code  ends
```

(源码)







课内测试CH06-3

1. 请在填空中 [填空1] 填写“36”（10分）；

2. 假设定了如下结构体，ds=17a6H，bx=00aaH，存储单元内容如下图，执行如下指令：（10分）

① mov ax,[bx].bb, ax= [填空2] H;

② mov ax,[bx].aa+2, ax= [填空3] H.

```
frame   struc
    aa      dw      ?
    bb      dw      ?
    cc      dw      ?
frame   ends
```

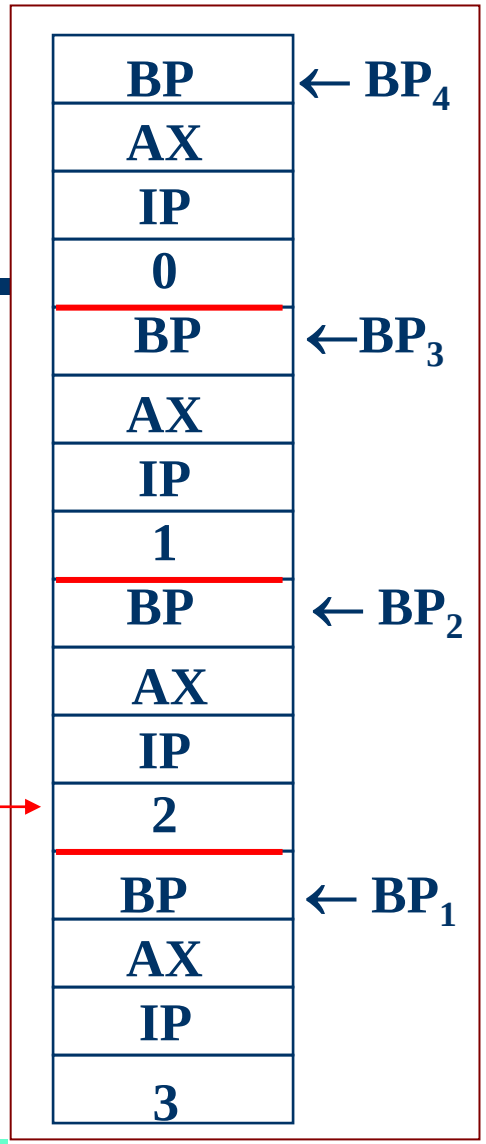
17a6:00aa	01
17a6:00ab	02
17a6:00ac	03
17a6:00ad	04
17a6:00ae	05
17a6:00af	06

例6.7 计算n! 不使用STRUC定义

```
mov  bx, n
push  bx
call  fact
pop   result
```

主程序部分

```
fact   proc  near
       push  ax
       push  bp
       mov   bp, sp
       mov   ax, [bp+6]
       cmp   ax, 0
       jne   fact1
       inc   ax
       jmp   exit
fact1: dec   ax
       push  ax
       call  fact
       pop   ax
       mul   word ptr[bp+6]
exit:  mov   [bp+6], ax
       pop   bp
       pop   ax
       ret
fact   endp
```



传递参数与
结果共用该单元

使用STRUC定义，结构清晰，不易出错，修改方便!

6.4 DOS系统功能调用

- 系统功能调用是DOS为系统程序员及用户提供的一组常用子程序
  - 用户可在程序中调用DOS提供的功能

- DOS规定用INT 21H中断指令作为进入各功能调用子程序的总入口，再为每个功能调用规定一个功能号，以便进入相应各个子程序的入口。

- DOS系统功能调用的分类：

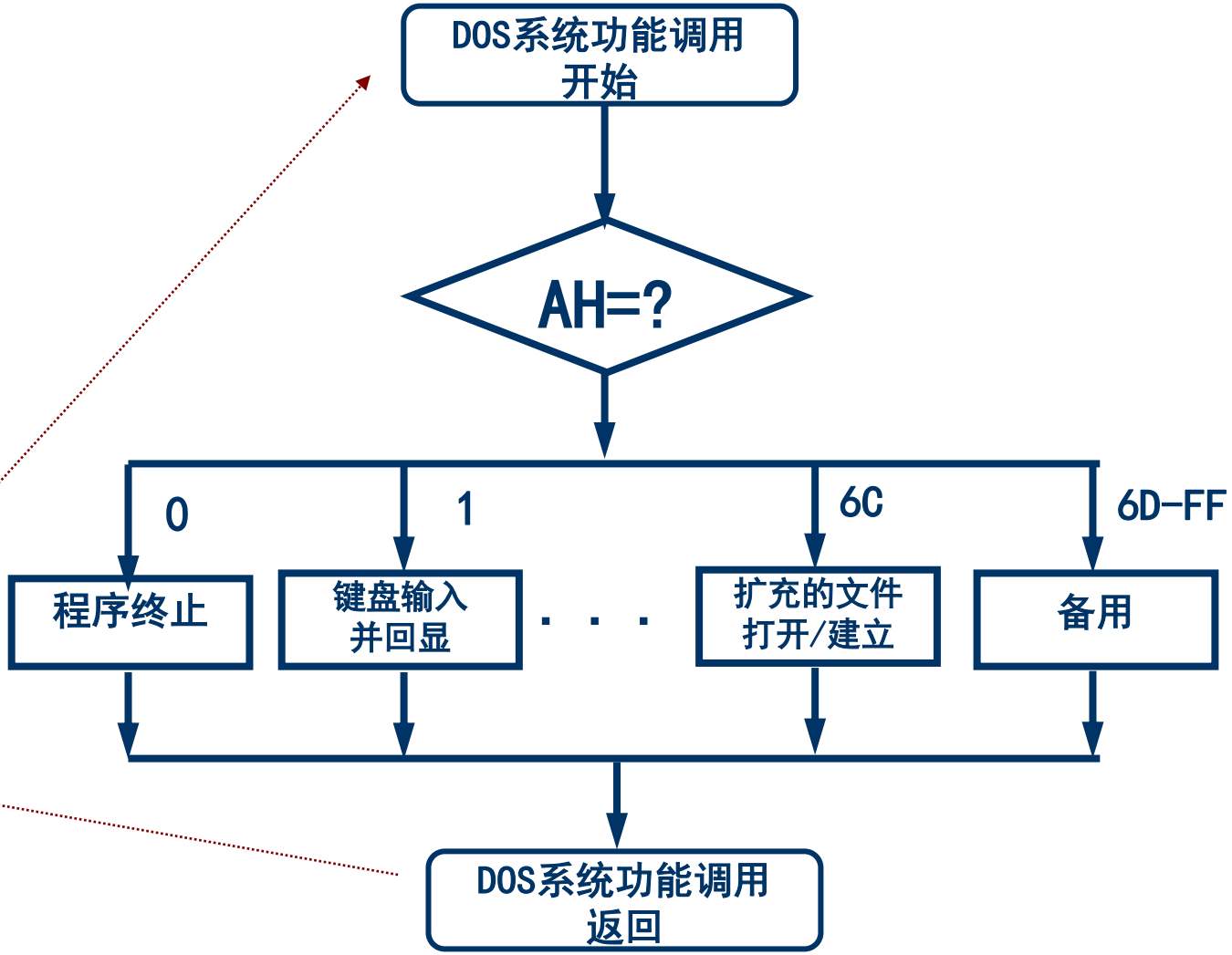
  设备管理、文件管理、目录管理

- **DOS系统功能调用的使用方法（约定）：**
  - 在AH寄存器中存入所要调用功能的功能号；
  - 根据所调用功能的规定设置入口参数；
  - 用INT 21H指令转入DOS系统功能子程序入口；
  - 相应的子程序运行完后,可以按规定取得出口参数

- **一般调用格式：**
  - 设置调用参数
  - MOV AH, 功能号
  - INT 21H
  - 取返回参数

```
MOV AH, 1    ；键盘输入并回显
INT  21H
```

- **简单举例：参看P605 附录四——键盘输入单个字符，显示器输出单个字符等**



用户程序中：

设置调用参数指令

MOV AH, 功能号

INT 21H

取返回参数指令

多分支结构

（1）DOS键盘功能调用 (AH=1, 6, 7, 8, A, B, C)

例：单字符输入 (AH=1 ： 键盘输入并回显)

```
get-key:  mov   ah, 1    ；键盘输入并回显
          int   21h
          cmp   al, 'Y'  ；键盘输入字符在AL中
          je    yes
          cmp   al, 'N'
          je    no
          jne   get_key
yes:
          ……
no:
          ……
```

例：输入字符串 (AH=0ah)

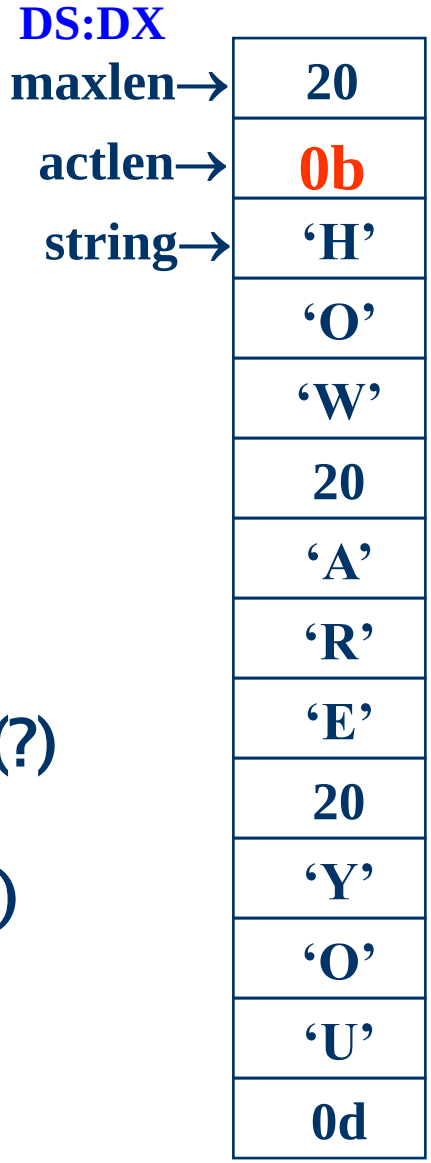
定义缓冲区：

方法1
```
maxlen  db  32
actlen  db  ?
string  db  32  dup  (?)
```

方法2
```
maxlen  db  32, 0, 32 dup (?)
```

方法3
```
maxlen  db  32, 33  dup  (?)
```

输入字符串
```
lea  dx, maxlen
mov  ah, 0ah
int  21h
```



（2） DOS显示功能调用 (AH=2, 6, 9)

例：显示单个字符 （ AH=2 ）

```
mov  ah, 2
mov  dl, ‘A’
int  21h
```

例：显示字符串 （ AH=9 ）

```
string  db  ‘HELLO’, 0dh, 0ah, ‘$’
mov  dx, offset  string
mov  ah, 9
int  21h
```

（3） DOS打印功能 (AH=5)

例：输出单个字符到打印机 (AH=5)

```
mov  ah, 5
mov  dl, ‘A’
int  21h
```

- 设计子程序时应注意的问

1. 子程序功能定义与说明
2. 参数传递方法
3. 寄存器的保存与恢复
4. 密切注意堆栈状态

初学者如何编写汇编程序？

正确理解　意

↓

设计程序流程图

↓

查指令表、DOS和BIOS调用规定，堆积指令，实现基本功能
注意正确使用寻址方式，灵活使用所学基本范例

↓

按汇编程序结构设计要求，正确划分定义数据段、代码段等，
正确使用转移指令（段内、段间）

↓

优化设计

↓

结合变量存储内容、寄存器内容变化及转移情况，
自己模拟执行一遍，静态查错

谢谢！

第七章　高级汇编语言技术

7.1　宏汇编

7.2　重复汇编

7.3　条件汇编

7.4　高级语言结构

本章目标

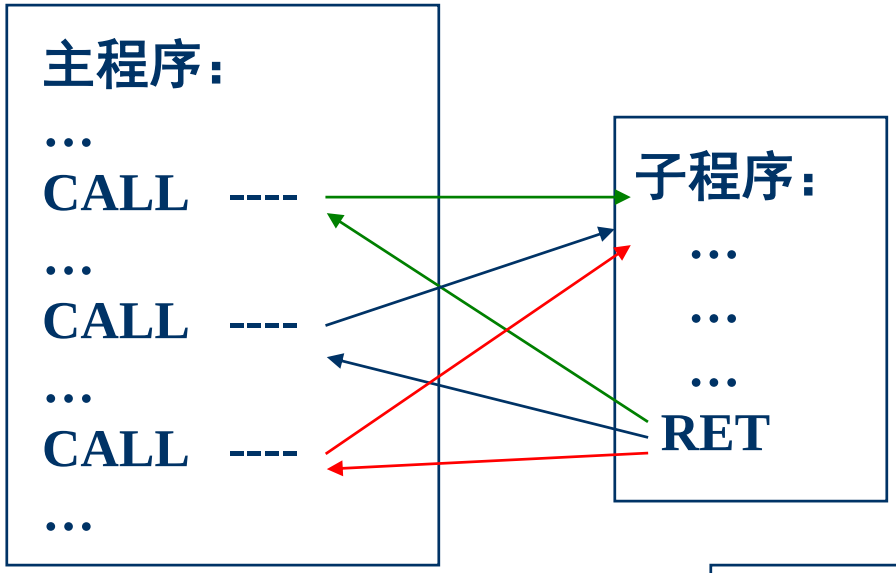
1. 掌握宏汇编
   - 定义、调用、展开
2. 掌握重复汇编
   - 读程序、写结果
3. 了解条件汇编
   - 调用、展开

7.1 宏汇编

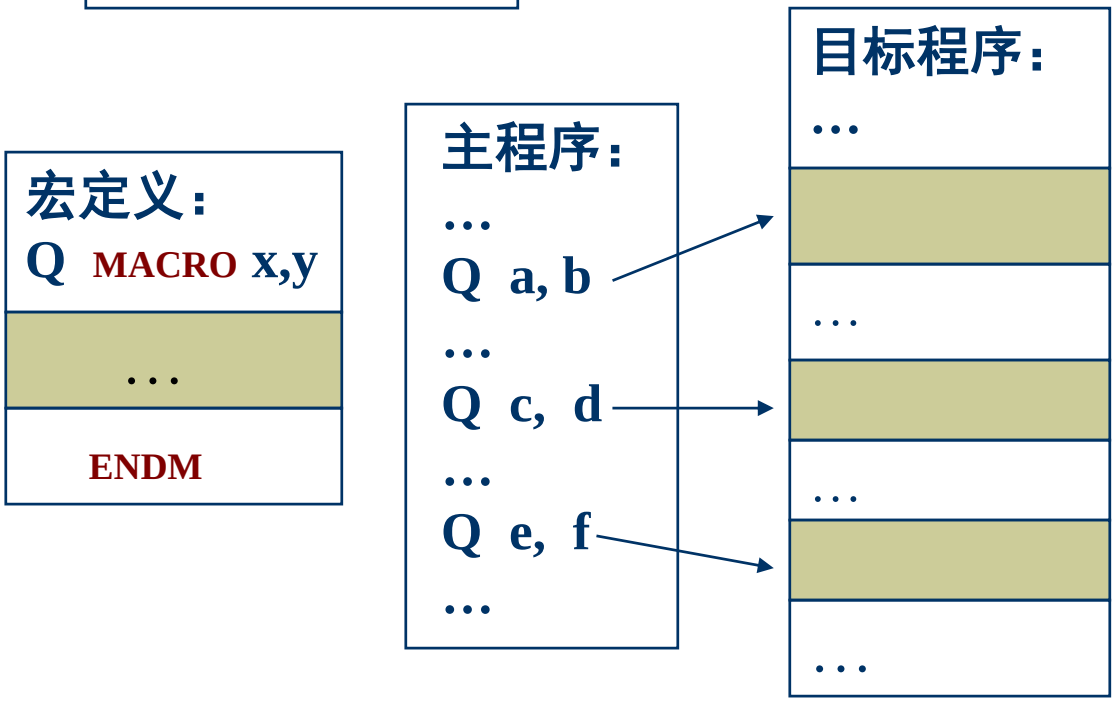
宏：源程序中一段有独立功能的程序代码。

宏指令：用户自定义的指令。在编程时，将多次使用的功能用一条宏指令来代替。

汇编语言程序
- 指令
- 伪指令（伪操作）
- **宏指令**



优：模块化
省内存
缺：开销大



优：

参数传送简单，执行效率高

缺：

占用内存空间大

- 子程序
  - 优点
    - 节省存储空间及程序设计所花的时间
    - 提供模块化程序设计的条件
    - 便于程序的调试及修改
  - 缺点
    - 转子、返回，保存、恢复寄存器，参数的传送等，增加了程序的额外开销（操作所消耗的时间、占用的存储空间）

- 宏汇编的用途
  - 当子程序本身较短或者需要传送的参数较多的情况下，使用宏汇编更加便利
  - 为用户提供更加容易、更加灵活、更加向高级语言靠拢的汇编工具

```
multiply macro opr1,opr2,result
        push  dx
        push  ax
        mov  ax, opr1
        imul  opr2
        mov  result, ax
        mov  result+2, dx
        pop  ax
        pop  dx
        endm
;
data    segment
var     dw ?
xyz     dw ?,?
data    ends
;
cseg            segment
                assume cs:cseg,ds:data
start           proc far
;
                mov ax, data
                mov ds, ax
                multiply  cx, var, xyz[bx]
;
exit:           mov ax,4c00h
                int 21h
start           endp
cseg            ends
        end start
```

源程序.ASM文件

```
-u
1425:0000 B82414        MOV     AX,1424
1425:0003 8ED8          MOV     DS,AX
1425:0005 52            PUSH    DX
1425:0006 50            PUSH    AX
1425:0007 8BC1          MOV     AX,CX
1425:0009 F72E0000      IMUL    WORD PTR [0000]
1425:000D 89870200      MOV     [BX+0002],AX
1425:0011 89970400      MOV     [BX+0004],DX
1425:0015 58            POP     AX
1425:0016 5A            POP     DX
1425:0017 B8004C        MOV     AX,4C00
1425:001A CD21          INT     21
```

宏展开后机器指令

汇编时展开用一组指令替代宏指令

```
Microsoft (R) Macro Assembler Version 5.00      4/18/9      Page     1-1

multiply    macro  opr1, opr2, result
            push  dx
            push  ax
            mov  ax, opr1
            imul  opr2
            mov  result, ax
            mov  result+2, dx
            pop  ax
            pop  dx
            endm
;
 0000                           data    segment
 0000  ????                     var     dw ?
 0002  ???? ????                xyz     dw ?,?
 0006                           data    ends
;
 0000                           cseg    segment
                                assume  cs:cseg, ds:data
 0000                           start    proc far
;
 0000  B8 ---- R                        mov ax, data
 0003  8E D8                            mov ds, ax
                                        multiply  cx, var, xyz[bx]
 0005  52                       1       push  dx
 0006  50                       1       push  ax
 0007  8B C1                    1       mov  ax, cx
 0009  F7 2E 0000 R             1       imul  var
 000D  89 87 0002 R             1       mov  xyz[bx], ax
 0011  89 97 0004 R             1       mov  xyz[bx]+2, dx
 0015  58                       1       pop  ax
 0016  5A                       1       pop  dx
;
 0017  B8 4C00                  exit:   mov ax,4c00h
 001A  CD 21                            int 21h
 001C                           start   endp
001C                            cseg    ends
                                        end   start
```

汇编后的.LIST文件

7.1.1 宏定义、宏调用和宏展开

宏定义：

macro_name MACRO [哑元表] ; 形参/虚参

[LOCAL 标号表]

……

…… ; 宏定义体

ENDM

宏调用：（必须先定义后调用）

macro_name [实元表] ; 实参

宏展开：汇编程序把宏调用展开

宏定义体 ⟹ 复制到宏指令位置,实参代虚参

LOCAL中的标号 ⟹ ??0000～??ffff

- 宏定义
  - 格式：

```
macro_name   MACRO   [哑元表]
                [ LOCAL    标号表 ]
                ……      （宏定义体）
                ENDM
```

```
multiply MACRO opr1,opr2,result
        mov  ax, opr1
        imul  opr2
        mov  result, ax
        mov  result+2, dx
      ENDM
```

* MACRO、 ENDM是一对宏定义伪操作
* 哑元表给出形式参数（虚参）
* 宏定义体：一组有独立功能的程序段
* 如果宏定义体有一个或多个标号，则必须用LOCAL伪操作列出所有的标号

宏定义

macro_name MACRO [哑元表]
[LOCAL 标号表]
…… （宏定义体）
ENDM

- **宏调用：定义了宏指令，就可以在程序中多次调用它**
  - 格式：

    macro_name ［实元表］ ；实参

    * 实元表中的实元与哑元表中的哑元在位置上一一对应
    * 若实元数>哑元数，则多余的实元无效
    * 若实元数<哑元数，则多余的哑元作“空(NUL)”处理
    * 对宏指令的调用：必须先定义后调用
    * 实元：可以是常数、寄存器、存储单元名、地址、表达式；也可以是操作码或操作码的一部分

汇编程序主要功能：

1、检查程序

2、测出源程序中的语法错误，并给出错误信息

3、展开宏指令

4、产生源程序的机器语言目标程序，并给出列表文件，同时列出汇编语言和机器语言，XXX.LST

宏定义

```
macro_name  MACRO  [哑元表]
        [ LOCAL   标号表 ]
        ……   （宏定义体）
        ENDM
```

宏调用

```
macro_name  [实元表]
```

- **宏展开：**
  - 源程序被汇编时，汇编程序将对每个宏调用作宏展开
    - 用宏定义体替换宏指令名，把宏定义体复制到调用宏指令的位置上，同时用实元取代哑元
  - 由LOCAL定义的标号也由 ??0000～??FFFF 偏移量替代 （其实质是自动给了一个新标号）
    - 多次宏调用展开时，解决标号冲突问题

课内测试CH07-1

1. 请在填空中 [填空1] 填写“78”（10分）；

2. 汇编源程序中定义宏和使用宏调用，请问：（10分）

① 宏调用是在 [填空2]时展开

A. 程序员编写程序； B. 汇编； C. 连接； D. 程序执行

② 宏展开是[填空3]

A. 伪操作替代宏调用指令；

B. 宏定义体的机器指令替代宏调用指令；

C. 根据宏的属性声明

例7.1　两个16位的字操作数相乘

宏定义：（编程时）

```
multiply MACRO opr1,opr2,result
         push  dx
         push  ax
         mov   ax, opr1
         imul  opr2
         mov   result, ax
         mov   result+2, dx
         pop   ax
         pop   dx
        ENDM
```

宏调用：（编程时）

```
multiply  cx, var, xyz[bx]
```

替代

宏展开：（汇编时）

```
1   push   dx
1   push   ax
1   mov    ax, cx
1   imul   var
1   mov    xyz[bx], ax
1   mov    xyz [bx]+2, dx
1   pop    ax
1   pop    dx
```

1 表示这些指令由宏展开，同时也表示第一层展开结果，较早版本用 + 表示

7.1.2 宏定义中的参数

- **哑元**：实质上只是一个字符串构成的符号
- **实元**：可以是常数、寄存器、存储单元、地址、表达式；也可以是操作码或操作码的一部分
- 哑元和实元统称变元

例7.2 宏定义无变元

例7.3 变元是操作码

例7.4 变元是操作码的一部分

例7.6 变元是字符串

例7.7 变元是表达式

例7.2　保存寄存器

宏定义可以无变元

宏定义：

```
savereg   MACRO
          push   ax
          push   bx
          push   cx
          push   dx
          push   si
          push   di
          ENDM
```

宏调用：

```
savereg
```

宏展开：

```
1    push   ax
1    push   bx
1    push   cx
1    push   dx
1    push   si
1    push   di
```

例7.3　变元可以是操作码

宏定义：

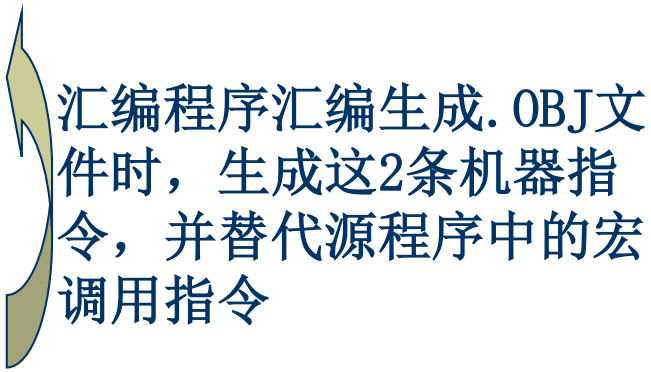
```
FOO MACRO  P1, P2, P3
      MOV  AX, P1
      P2   P3
    ENDM
```

宏调用：

```
FOO  WORD_VAR, INC, AX
```

宏展开：

```
1   MOV  AX, WORD_VAR
1   INC  AX
```



宏汇编操作符 &　;;　%　: REQ　:=

- 符号1&符号2
  - 文本替换操作符。宏展开时，合并前后两个符号形成一个符号
  - 符号可以是操作码、操作数或是一个字符串

例7.4　变元是操作码的一部分

宏定义：（源程序中定义）

```
leap  macro  cond, lab
      j&cond   lab
      endm
```

宏调用：（源程序中调用）

```
leap  z, there
……
leap  nz, here
```

宏展开：（汇编时展开）

```
1   jz   there
    ……
1   jnz  here
```

- **;;注释**
  - 宏注释。宏展开时，若注释以一个分号开始，则该注释在宏扩展时出现。若注释以两个分号开始，则;;后面的注释不予展开

例：
```
Q   MACRO   m
            ;  display a message
            ;; m is a string
            ……
    ENDM
```
每次展开保留此注释

每次展开不保留此注释

% 表达式

- 表达式操作符。汇编程序将%后面的表达式立即求值转换为数字，并在展开时用这个数取代哑元，宏调用时使用

例7.7

宏定义：

```
MSG     MACRO   COUNT, STRING
        MSG&COUNT  DB  STRING
        ENDM
ERRMSG  MACRO  TEXT
        CNTR=CNTR+1
        MSG    %CNTR, TEXT
        ENDM
```

宏调用：

```
        ……
        CNTR=0
        ERRMSG   ‘SYNTAX ERROR’
        ……
        ERRMSG   ‘INVALID OPPERAND’
```

宏展开：

```
        ……
        CNTR=0
2       MSG1   DB    ‘SYNTAX ERROR’
        ……
2       MSG2   DB    ‘INVALID OPPERAND’
        ……
```

◆ : REQ

- 指定某个变元必须有。调用时必须有对应的实元，否则汇编时出错

例7.8：宏定义

```
DIF     MACRO  A, B
        DB   B-A
        ENDM
DIF1    MACRO  A:REQ, B:REQ
        DB   B-A
        ENDM
```

```
DIF2    MACRO  A:REQ, B
        DB   B-A
        ENDM
```

P267

◆ :=

- 为宏变元提供缺省值。

例7.9：

宏定义：

```
DIF3   MACRO  A:=<10>,B:=<12>
       DB  B-A
       ENDM
```

宏调用：

```
DIF3
```

宏展开：

```
1   DB  12-10
```

宏调用：

```
DIF3   5, 8
```

宏展开：

```
1   DB  8-5
```

注意：宏指令名与指令助记符或伪操作名相同时，宏指令定义的优先级最高，即同名的助记符或伪操作名被汇编程序认为是宏指令。

例：

宏定义：

```
add    MACRO   opr1, opr2, result
         ......
       ENDM
```

宏调用：

```
       ......
       add     xx, yy, zz
       purge   add                ; 取消宏定义
       ......
```

建议宏指令名与指令助记符或伪操作名尽量不要相同

7.1.3 LOCAL伪操作

宏定义
macro_name MACRO [哑元表]
[LOCAL 标号表]
..... （宏定义体）
ENDM

- 当在宏定义体中使用了标号，多次调用该宏定义时，则展开后会出现标号的多重定义，这是不允许的。
  - LOCAL伪操作可以解决这个问题
- LOCAL伪操作格式： LOCAL 局部标号表
  - 局部标号表中的每个符号，在汇编时每扩展一次便建立一个惟一的标号，形如??xxxx（xxxx的值在0000~FFFF之间），以保证汇编时生成符号名字的惟一性
  - LOCAL伪操作只能用在宏定义体内，必须是MACRO伪操作后的第一个语句，在MACRO和 LOCAL伪操作之间，不允许有注释和分号标志

例7.10　求绝对值(使用LOCAL伪操作)

宏定义：

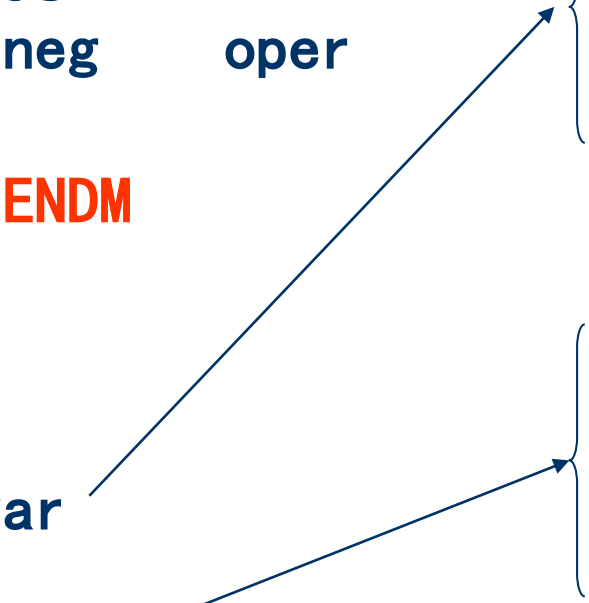
```
absol   MACRO   oper
        LOCAL   next
        cmp     oper, 0
        jge     next
        neg     oper
next:
        ENDM
```

宏调用：

```
        ......
        absol   var
        ......
        absol   bx
        ......
```

宏展开：

```
                ......
1               cmp     var, 0
1               jge     ??0000
1               neg     var
1       ??0000:
                ......
1               cmp     bx, 0
1               jge     ??0001
1               neg     bx
1       ??0001:
                ......
```



例：定义延时程序的宏指令，在同一个程序中两次被调用的扩展情况

宏定义：

```
DELAY   MACRO
        LOCAL   LOP
        MOV   CX, 2801
LOP:    LOOP   LOP
        ENDM
```

宏调用：

```
        DELAY
        DELAY
```

汇编时宏扩展如下：

```
                MOV  CX, 2801
1  ??0000:  LOOP  ??0000
                MOV  CX, 2801
1  ??0001:  LOOP  ??0001
```

7.1.4 在宏定义内使用宏

- 宏指令嵌套有两种情况：
  - 宏定义体中含有宏调用
    - 必须先定义后调用
  - 宏定义体中含有宏定义

例7.12

宏定义：

```
INT21   MACRO  FUNCTION
        MOV  AH,  FUNCTION
        INT   21H
        ENDM
DISP    MACRO  CHAR
        MOV  DL,  CHAR
        INT21  02H
        ENDM
```

宏调用：

```
        DISP        '? '
```

宏展开：

```
1       MOV         DL, '?'
2       MOV         AH, 02H
2       INT         21H
```

例7.13

宏定义：

```
DEFMAC   MACRO  MACNAM, OPERATOR
         MACNAM  MACRO  X, Y, Z
                 PUSH  AX
                 MOV  AX, X
                 OPERATOR  AX, Y
                 MOV  Z, AX
                 POP  AX
                 ENDM
         ENDM
```

用宏调用形成加法宏定义：

```
    DEFMAC     ADDITION, ADD
```

形成加法宏定义：

```
ADDITION    MACRO  X, Y, Z
         PUSH  AX
         MOV  AX, X
         ADD  AX, Y
         MOV  Z, AX
         POP  AX
         ENDM
```

宏调用：

```
    ADDITION    VAR1, VAR2, VAR3
```

宏展开：

```
1    PUSH  AX
1    MOV   AX, VAR1
1    ADD   AX, VAR2
1    MOV   VAR3, AX
1    POP   AX
```

课内测试CH07-2

1. 请在填空中 [填空1] 填写“76”（10分）；

2. 汇编源程序中，通过定义宏和使用宏调用，

优点是 [填空2]， 缺点是 [填空3] （10分）

A. 简化程序编程和修改，增强可读性

B. 使得程序编写和修改更复杂，降低了可读性

C. 节省存储空间和执行时间

D. 不节省存储空间和执行时间

7.1.5 列表伪操作
(自学 p271)

列表伪操作：

.LALL：在LST清单中列出宏展开后的全部语句(包 括注释)。

.SALL：在LST清单中不列出任何宏展开后的语句。

.XALL：缺省的列表方式，只列出宏体中产生目标代码的语句。

列表伪指令只影响列表文件，并不影响目标码的生成

7.1.6 宏库的建立与调用

(自学 P274)

建立宏库：

>EDIT MACRO . MAC

```
macro1 MACRO [哑元表]
       ……
       ENDM

macro2 MACRO [哑元表]
       ……
       ENDM
       ……

macroN MACRO [哑元表]
       ……
       ENDM
```

调用宏库：

>EDIT EXP.ASM

```
include MACRO.MAC
        ……
        macro1 [实元表]
        ……
        macro2 [实元表]
        ……
        macroN [实元表]
        ……
```

7.1.7 删除宏定义

- 格式：

```
PURGE    macro_name[, macro_name, ...]
```

  - 删除不再使用的宏定义，使该宏定义为空
  - 删除后，汇编程序再遇到该宏调用指令将忽略，也不会指示出错

7.2 重复汇编

用于连续产生完全相同或基本相同的一组代码。

重复伪操作 REPT

REPT 表达式

[redacted];重复块

ENDM

不定重复伪操作 IRP/IRPC

IRP 哑元，<自变量表>

[redacted];重复块

ENDM

IRPC 哑元，字符串

[redacted];重复块

ENDM

7.2.1 重复伪操作

- **重复伪操作 REPT**

  格式： REPT 表达式

  ..... ；重复块

  ENDM

  - 表达式：重复次数，结果应该是无符号常数
  - 不一定要用在宏定义中，程序其他段中也可以用

例7.15

```
X=0
REPT  10
X=X+1
DB     X
ENDM
```

汇编展开后：

```
1    DB    1
1    DB    2
1    DB    3
     ......
1    DB    10
```

例7.16　把字符‘A’到‘Z’的ASCII码填入数组TABLE

```
        CHAR='A'
TABLE   LABEL   BYTE
        REPT    26
        DB      CHAR
        CHAR=CHAR+1
        ENDM
```

汇编展开后：

```
TABLE   LABEL   BYTE
    1     DB    41H
    1     DB    42H
    1     DB    43H
          ……
    1     DB    5AH
```

7.2.2 不定重复伪操作

不定重复伪操作 IRP / IRPC

1. IRP伪操作：用实际参量替换哑元

格式： IRP 哑元，<自变量表>
……　　；重复块
ENDM

展开时，重复块中变量不定，无规律

- 每次重复用自变量表中的一项取代哑元，直到用完为止
- 重复次数由自变量的个数决定

例7. 20

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

汇编展开后：

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

例：在数据段产生字符区array，包括5个字符串‘NO.K’

```
data   segment
array  label  byte
       IRP  K, <1,2,3,4,5>
       db  'NO.&K'
       ENDM
data   ends
```

汇编展开后

```
data   segment
       array  label  byte
       1  db  'NO.1'
       1  db  'NO.2'
       1  db  'NO.3'
       1  db  'NO.4'
       1  db  'NO.5'
data   ends
```

课内测试CH07-3

1. 请在填空中 [填空1] 填写“73”（10分）；

2. 不定重复汇编伪操作如右图，

```
IRP REG, <AX, BP>
    MOVSW ES:[DI], REG
    ADD   DI, 2
ENDM
```

汇编时展开后生成如下机器指令：

MOVSW ES:[DI]，[填空2]

ADD DI, 2

MOVSW ES:[DI]，[填空3]

ADD DI, 2

7.2.2 不定重复伪操作

2. IRPC伪操作： 用字符串替换哑元

格式： IRPC 哑元，字符串
…… ；重复块
ENDM

- 每次重复用字符串中的一个字符取代哑，直到用完为止
- 重复次数等于字符串中的字符数

例： 在数据段产生字符区array，包括5个字符串 ‘NO.K’

```
data   segment
       array  label  byte
       IRPC  K, 12345
       db  ‘NO.&K’
       ENDM
data   ends
```

汇编展开后：

```
data   segment
       array  label  byte
       1  db  ‘NO.1’
       1  db  ‘NO.2’
       1  db  ‘NO.3’
       1  db  ‘NO.4’
       1  db  ‘NO.5’
data   ends
```

7.3 条件汇编

根据条件把一段源程序包括在汇编语言程序内或者排除在外。

一般格式：

```
IF××   自变量          ; ××为条件
   ; 自变量满足条件则汇编此块
[ELSE]
   ; 自变量不满足条件则汇编此块
ENDIF
```

条件伪操作：

```
IF      表达式          ;表达式≠0，则汇编
IFE     表达式          ;表达式＝0，则汇编

IF1                     ;在第一遍扫视期间满足条件
IF2                     ;在第二遍扫视期间满足条件

IFDEF   符号            ;符号已定义，则汇编
IFNDEF  符号            ;符号未定义，则汇编

IFB     <自变量>        ;自变量为空，则汇编
IFNB    <自变量>        ;自变量不为空，则汇编

IFIDN   <字符串1>,<字符串2>     ;串1与串2相同
IFDIF   <字符串1>,<字符串2>     ;串1与串2不同
```

例7.24　求3个变元中最大值放入AX，且变元数不同时产生不同的程序段

宏定义：

```
MAX   MACRO   K, A, B, C
      LOCAL  NEXT, OUT
      MOV  AX, A
      IF     K-1      ；如果k-1≠0，条件为真
      IF     K-2
      CMP  C, AX
      JLE   NEXT
      MOV  AX, C
      ENDIF
NEXT: CMP   B, AX
      JLE    OUT
      MOV   AX, B
      ENDIF
OUT:
      ENDM
```

宏调用：

```
MAX  1, P
MAX  2, P, Q
MAX  3, P, Q, R
```

宏展开：

```
1          MOV  AX, P
1 ??0001:
```

```
1          MOV  AX, P
1 ??0002: CMP  Q, AX
1          JLE   ??0003
1          MOV  AX, Q
1 ??0003:
```

```
1          MOV  AX, P
1          CMP  R, AX
1          JLE   ??0004
1          MOV  AX, R
1 ??0004: CMP   Q, AX
1          JLE   ??0005
1          MOV  AX, Q
1 ??0005:
```

```
MAX   MACRO   K, A, B, C
              LOCAL  NEXT, OUT
              MOV  AX, A
              IF   K-1    ;如果k-1≠0，条件为真
                IF   K-2
                   CMP  C, AX
                   JLE  NEXT
                   MOV  AX, C
                ENDIF
NEXT:           CMP   B, AX
                JLE  OUT
                MOV  AX, B
              ENDIF
OUT:
        ENDM
;
data    segment
P               dw ?
Q               dw ?
R               dw ?
data    ends
;
cseg            segment
                assume cs:cseg,ds:data
start           proc far
;
                mov ax, data
                mov ds, ax
;
                MAX  1, P
                MAX  2, P, Q
                MAX  3, P, Q, R
;
exit:           mov ax,4c00h
                int 21h
start           endp
cseg            ends
               end start
```

```
MAX   MACRO   K, A, B, C
                  LOCAL  NEXT, OUT
                  MOV  AX, A
                  IF   K-1    ;如果k-1≠0，条件为真
                    IF   K-2
                      CMP  C, AX
                      JLE  NEXT
                      MOV  AX, C
                    ENDIF
NEXT:               CMP   B, AX
                    JLE  OUT
                    MOV  AX, B
                  ENDIF
OUT:
        ENDM
;
0000            data    segment
0000 ????       P       dw ?
0002 ????       Q       dw ?
0004 ????       R       dw ?
0006            data    ends
;
0000                            cseg            segment
                                                assume cs:cseg,ds:data
0000                            start           proc far
;
0000 B8 ---- R                                  mov ax, data
0003 8E D8                                      mov ds, ax
;
                                                MAX  1, P
0005 A1 0000 R                  1               MOV  AX, P
0008                            1       ??0001:
                                                MAX  2, P, Q
0008 A1 0000 R                  1               MOV  AX, P
000B 39 06 0002 R               1       ??0002: CMP   Q, AX
000F 7E 03                      1               JLE  ??0003
0011 A1 0002 R                  1               MOV  AX, Q
0014                            1       ??0003:
                                                MAX  3, P, Q, R
0014 A1 0000 R                  1               MOV  AX, P
0017 39 06 0004 R               1               CMP  R, AX
001B 7E 03                      1               JLE  ??0004
001D A1 0004 R                  1               MOV  AX, R
0020 39 06 0002 R               1       ??0004: CMP   Q, AX
0024 7E 03                      1               JLE  ??0005
0026 A1 0002 R                  1               MOV  AX, Q
0029                            1       ??0005:
;
0029 B8 4C00                            exit:   mov ax,4c00h
002C CD 21                                      int 21h
002E                            start           endp
002E                            cseg            ends
                                                end     start
```

例7.25 根据跳转距离生成不同跳转指令

宏定义：

```
BRANCH    MACRO   X
          IF      ($-X) LT 128
                  JMP  SHORT X
          ELSE
                  JMP  NEAR PTR X
          ENDIF
          ENDM
```

宏调用：

```
BRANCH  AA
```

宏展开：

如果相对于AA距离小于128，宏展开

```
1    JMP  SHORT AA
```

否则产生

```
1   JMP  NEAR PTR AA
```

例7.26　在宏定义的递归调用中，使用条件伪操作结束宏递归

X和2^N相乘，即X左移N次

宏定义：

```
POWER  MACRO   X, N
          SAL  X,  1
          COUNT=COUNT+1
          IF   COUNT-N
          POWER  X,  N
          ENDIF
       ENDM
```

宏调用：

```
    COUNT=0
     POWER  AX, 3
```

宏展开：

```
1    SAL  AX,  1
2    SAL  AX,  1
3    SAL  AX,  1
```

课内测试CH07-4

1. 请在填空中 [填空1] 填写“79”（10分）；

2. 源程序中有条件汇编的宏定义和宏调用如右图，汇编时宏展开后生成如下机器指令：

[填空2] AX，1

[填空3] AX，1

宏定义：

```
SH   MACRO   K
     IF 'L' EQ K
          SHL AX, 1
     ELSE
          SHR AX, 1
     ENDIF
     ENDM
```

宏调用：

```
SH   'R'
SH   'L'
```

例 7.28 (*p285*):

```
divide  macro  dividend,divisor,quotient
        local  comp, out
        cnt=0
        ifndef dividend
        cnt=1
        endif
        ifndef divisor
        cnt=1
        endif
        ifndef quotient
        cnt=1
        endif
        if     cnt
        exitm
        endif
        mov ax, dividend
        mov bx, divisor
        sub cx, cx
comp:
        cmp ax, bx
        jb  out
        sub ax, bx
        inc cx
        jmp comp
out:
        mov quotient, cx
        endm
```

7.4 高级语言结构
（自学 *p293*）

- MASM 6.0引入了几种更接近高级语言编程的高级语言结构，如以下标准宏指令
  - .IF/.ELSEIF/.ELSE/.ENDIF
  - .WHILE/.ENDW
  - .REPEAT/.UNTIL
  - .REPEAT/.UNTILECXZ
  - .BREAK
  - .COUNTINUE
- 汇编程序将按标准指令段展开，形成一串指令完成特定操作

.IF/.ELSEIF/.ELSE/.ENDIF

格式：

```
.IF   expression1
(汇编语言语句组1)
.ELSEIF   expression2
(汇编语言语句组2)
.ELSEIF   expression3
(汇编语言语句组3)
   :
.ELSE
(汇编语言语句组n)
.ENDIF
```

.IF宏指令在汇编时会产生比较（CMP）和条件跳转两条指令

汇编时这里产生一条JMP指令

```
.IF  AL= ="A"
CALL  DISP
.ENDIF
```

→

```
CMP   AL, "A"
JNZ   NOTA
CALL  DISP
```

高级语言结构中使用的表达式

- 表达式中的操作符

机器指令中有对应的操作

= =	相等
! =	不等
>	大于
>=	大于或等于
<	小于
<=	小于或等于
&	位测试
!	逻辑非
&&	逻辑与
‖	逻辑或

- 表达式格式

1） 测试条件码的值(=1时表达式值为真)

- ZERO?
- CARRY?
- OVERFLOW?
- SIGN?
- PARITY?

} 机器指令中有对应的条件转移

2） 操作数构成的表达式

- reg op reg
- reg op memory
- reg op constant
- memory op constant

} 机器指令中有对应的操作数表示形式

```
.IF  reg  ==  constant   ⟹
CMP  reg, constant
JNZ  xxxxx
```

高级语言语句在汇编时汇编程序将生成标准的机器指令串替代高级语言语句

更加易于编程和可读性

```
.IF  AL==“A”
     CALL  DISP
.ELSEIF  AL==“B”
     CALL  DISP
.ELSE
     MOV  AL, “N”
     CALL  DISP
.ENDIF
```

汇编展开

```
          CMP   AL, “A”
          JNZ  NOTA
          CALL  DISP
          JMP  DONE
NOTA: CMP   AL, “B”
          JNZ  NOTB
          CALL  DISP
          JMP  DONE
NOTB: MOV  AL, “N”
          CALL  DISP
DONE:
```

例 7.32 (p294)

练习举例

P301 7.1

7.1 定义宏指令CLRB，完成用空格符将一字符区中的字符取代的工作。字符区的首地址及其长度为变元。（两种实现方案）

```
CRLB1   MACRO   ADDR, CUNT
        LOCAL   NEXT
        MOV     SI, OFFSET  ADDR
        MOV     AL, 20H   ;
    ‘ ’ =20H
        MOV     CX, CUNT
NEXT:   MOV     [SI], AL
        INC     SI
        LOOP    NEXT
        ENDM
```

```
CRLB2   MACRO   ADDR, CUNT
        LEA     DI, ADDR
        MOV     AL, ’ ’
        CLD                 ; DF=0
        MOV     CX, CUNT
        REP     STOSB
        ENDM
```

第7次课后作业

思源学堂→课后作业→第7次课后作业

要求：

（1）独立完成，严禁抄袭

（2）以word文档在思源学堂提交

（3）截至日期：6月1日23:59

第八章　输入输出程序设计

8.1　I/O设备的数据传送方式

8.2　I/O程序举例

8.3　中断传送方式

8.4　80386输入输出

8.5　80386的中断处理

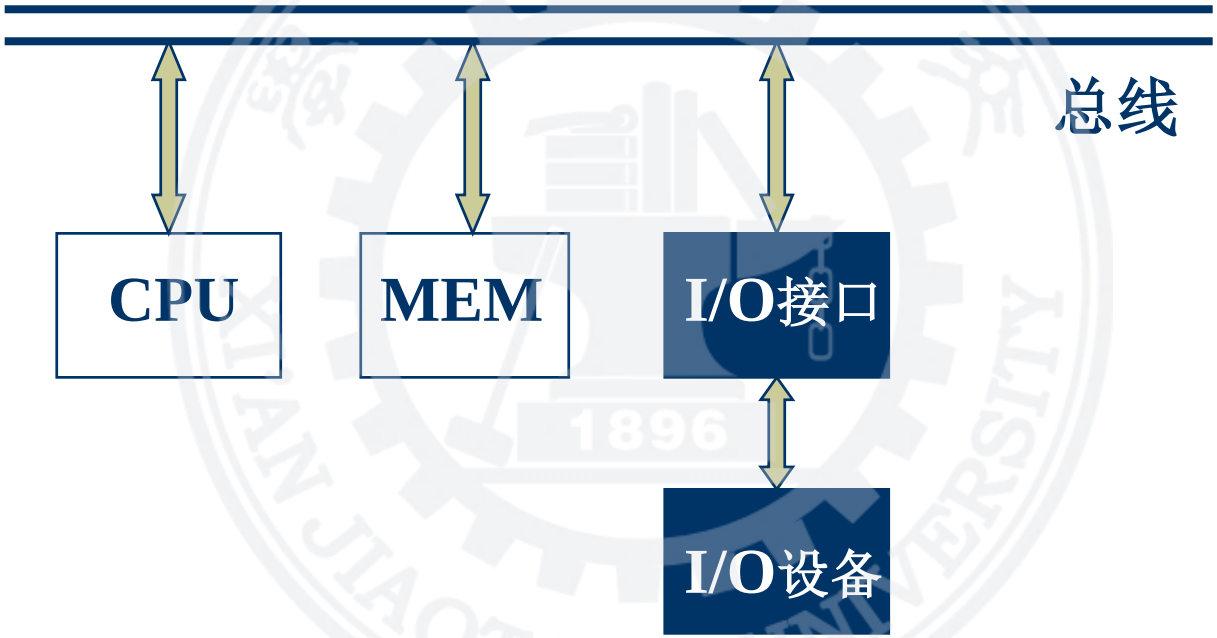
本 章 目 标

1. 了解输入输出程序设计的基本概念

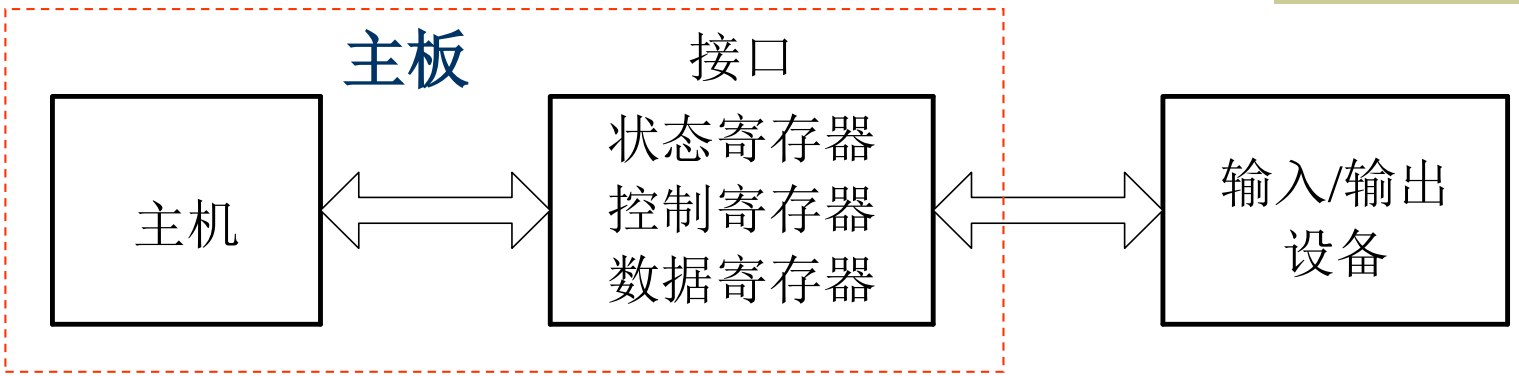
2. 掌握IN/OUT指令的用法

3. 掌握中断传送方式工作机制与编程方法

8.1 I/O设备的数据传送方式



8.1.1 主机、外设与接口



- 每种输入输出设备都要通过一个硬件接口或控制器和CPU相连
  - 如，打印机接口，显示器接口等
- 从程序设计的角度看，接口由一组寄存器组成，是完成输入输出的桥梁

8.1.2 I/O端口
(IO接口上的寄存器)

- I/O端口地址：为了访问接口上的寄存器，系统给这些寄存器分配专门的存取访问地址，这样的地址称为I/O端口地址
- 8086/8088CPU系统中，I/O端口地址和存储单元的地址是各自独立的，分占两个不同的地址空间
- 8086/8088CPU提供的I/O端口地址空间达64KB
  - 可接64K个8位端口(字节)，或可接32K个16位端口(字)
  - PC及其兼容机实际只使用0~3FFH之间的I/O端口地址
- 存取接口寄存器中的数据是依靠I/O指令完成的

PC只用了10位地址线(A0-A9)进行译码，其寻址的范围为0H-3FFH，共有1024个I/O地址。

X86机器I/O端口地址分配表

I/O 地址	功　能	I/O 地址	功　能
00～0F	DMA 控制器 8237A	2F8～2FE	2 号串行口(COM2)
20～3F	可编程中断控制器 8259A	320～324	硬盘适配器
40～5F	可编程中断计时器	366～36F	PC 网络
60～63	8255A PPI	372～377	软盘适配器
70～7F	实时钟	378～37A	2 号并行口(LPT1 打印机)
80～8F	DMA 页表地址寄存器	380～38F	SDLC 及 BSC 通讯
93～9F	DMA 控制器	390～393	Cluster 适配器
A0～BF	可编程中断控制器 2	3A0～3AF	BSC 通讯
C0～0E	DMA 接口专用	3B0～3BF	MDA 视频寄存器
F0～FF	协处理器	3BC～3BE	1 号并行口
170～1F7	硬盘控制器	3C0～3CF	EGA/VGA 视频寄存器
200～20F	游戏控制端口	3D0～3D7	CGA 视频寄存器
278～27A	3 号并行口(LPT2 打印机)	3F0～3F7	软盘控制寄存器
2E0～ 2E3	EGA/VGA 使用	3F8～3FE	1 号串行口(COM1)

主机板I/O

扩展插槽上I/O

8.1.3 I/O指令

1. **输入指令：IN　　累加器，端口地址**
   - IN AL, PORT　　;AL←(PORT)
   - IN AX, PORT　　;AL←(PORT), AH←(PORT+1)
   - IN AL, DX　　;AL←(DX)
   - IN AX, DX　　;AL←(DX), AH←(DX+1)
2. **输出指令：OUT　　端口地址，累加器**
   - OUT PORT, AL　　;(PORT)←AL
   - OUT PORT, AX　　;(PORT)←AL, (PORT+1)←AH
   - OUT DX, AL　　;(DX)←AL
   - OUT DX, AX　　;(DX)←AL, (DX+1)←AH

➢ **累加器：只能使用累加器**

- AX（16位字操作）；　AL（8位字节操作）

➢ **端口地址：只有两种方式给出**

- **立即数**：PORT（00H～FFH）；**寄存器间接**：DX（0000H～FFFFH）

8.1.4 I/O设备的数据传送方式

1. 程序直接控制I/O方式
   1). 无条件传送方式
   2). 查询方式
   - 主要缺点：CPU和I/O设备不能并行工作，CPU资源浪费十分严重
2. 直接存储器存取（DMA）方式
3. 中断方式

- 主要考虑因素：
  - ✓ CPU与IO设备速度匹配问题
  - ✓ 减轻CPU负担
  - ✓ 外设请求服务处理的时机

程序直接控制I/O方式

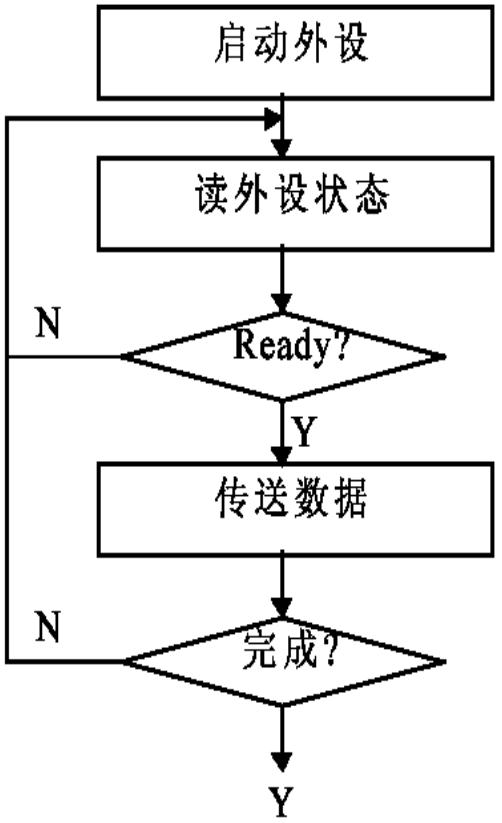
1). 无条件传送方式

- 默认外设总是处在“准备好” 或者“不忙” 状态
- 直接使用IN、OUT指令实现数据传送
- 例：外设输出数据端口地址70H，输入数据端口地址60H，则
  - 字节数据输入：IN AL，60H
  - 字数据输入：IN AX，60H ;(60H)→AL，(61H)→AH
  - 字节数据输出：OUT 70H，AL
  - 字数据输出：OUT 70H，AX
- 无条件传送方式要求（适合情况）
  - 外设的工作速度与CPU同步，否则就会出错
    - 如果CPU与外设的工作速度不同步应采用查询等其它方式
  - CPU负载轻，其它工作少时

程序直接控制I/O方式

2). 查询方式

- 在输入输出之前首先查询外设的状态
  - 当外设状态处于“准备好”（输入方式下）或者“不忙”（输出状态下），才可以进行输入/输出；
  - 反之，要一直等待到外设“准备好”或“不忙”
- 外设接口正在准备数据或者输出任务没有完成之前，其状态处于“没准备好”或者“忙”



- 例如：外设输出数据端口（输出数据寄存器）地址70H，输出状态端口地址72H，输出状态端口最高位=1表示输出设备不忙。则：
  - 查询方式字节数据输出：

```
WAIT: IN   AL, 72H     ; 读取输入口的状态
      TEST AL, 80H     ; 测试状态寄存器的最高位
      JZ   WAIT        ; 若状态位=0，则继续测试等待
      MOV  AL, VAR
      OUT  70H,  AL    ; 输出数据
```

  - 查询方式字输出：

```
WAIT: IN   AL, 72H     ; 读取输出口的状态
      TEST AL, 80H     ; 测试状态寄存器的最高位
      JZ   WAIT        ; 若状态位=0，则继续测试等待
      MOV  AX, VAR
      OUT  70H, AX     ; 输出数据
```

- 例如：外设输入数据端口地址60H，输入状态端口地址62H，输入状态端口62H的最高位=1表示输入设备准备好。则：
  - 查询方式字节数据输入：

```
WAIT:  IN    AL, 62H      ; 读取输出口的状态
       TEST  AL, 80H      ; 测试状态寄存器的最高位
       JZ    WAIT         ; 若状态位=0，则继续测试等待
       IN    AL, 60H      ; 输入数据
         ⋮
```

  - 查询方式字输入：

```
WAIT:  IN    AL, 62H      ; 读取输出口的状态
       TEST  AL, 80H      ; 测试状态寄存器的最高位
       JZ    WAIT         ; 若状态位=0，则继续测试等待
       IN    AX, 60H      ; 输入数据
         ⋮
```

课内测试CH08-1

1. 请在填空中 [填空1] 填写“83”（10分）；

2. 假设：外设输入数据端口地址81H，输入状态端口地址82H，状态端口最低位=1表示输入数据准备好。查询方式字节数据输入：（10分）

```
WAIT:  IN   AL，[填空2]
       TEST  AL，[填空3]
       JZ    WAIT
       IN    AL，81H
```

查询方式的问题

- 查询方式问题：虽然通过CPU重复查询外设状态，直到外设准备好再进行数据传送，解决了CPU与外设不同步的问题。可是存在以下问题：
  - 查询等待中的“踏步”问题
  - CPU和I/O处于串行工作方式，CPU效率不高

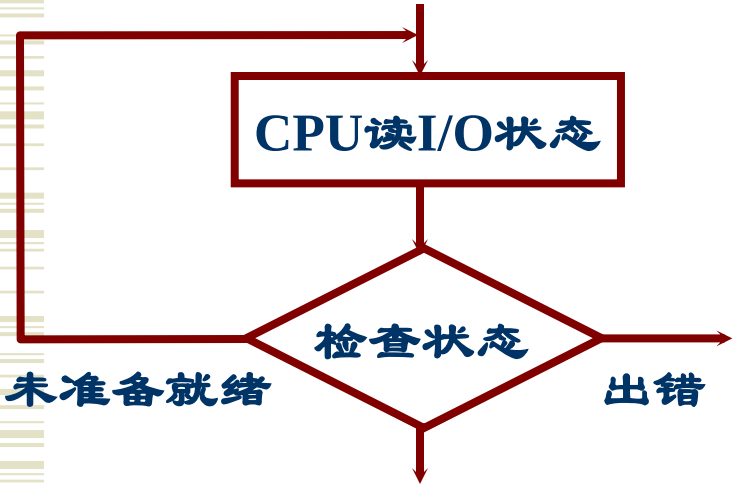
中断方式

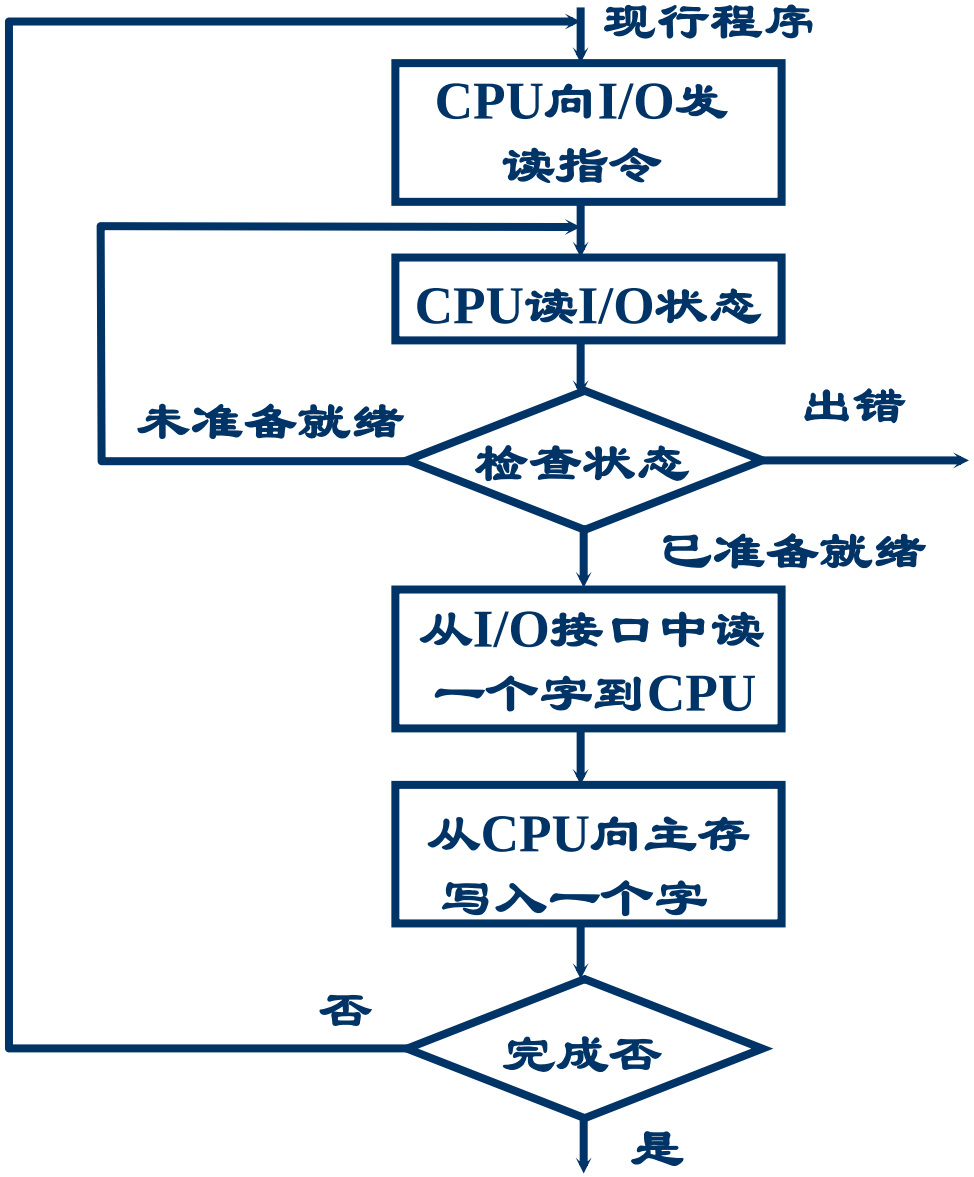
- 中断方式：
  - 当外设准备好时，外设主动向CPU发出中断服务请求，CPU暂时中止现行程序的执行，转入中断服务处理程序完成输入/输出工作，之后返回被中断的程序继续执行
- 中断方式特点：
  - CPU和I/O设备能够并行运行
  - 具有及时处理响应意外事件或异常的能力

I/O 与主机信息传送--程序查询方式

CPU 和 I/O 串行工作

踏步等待

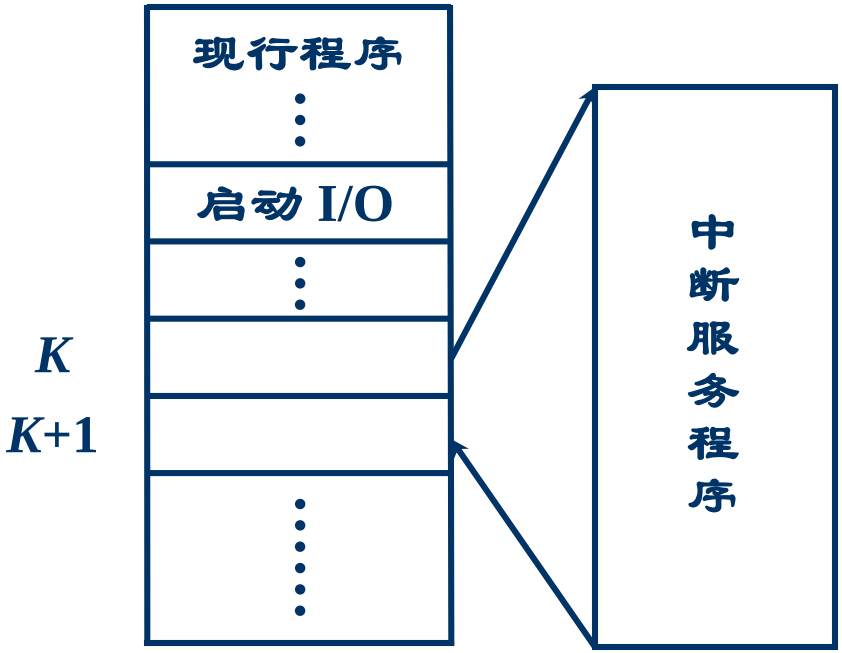




I/O 与主机信息传送--程序中断方式

I/O 工作
- 自身准备　　CPU 不查询
- 与主机交换信息　CPU 暂停现行程序

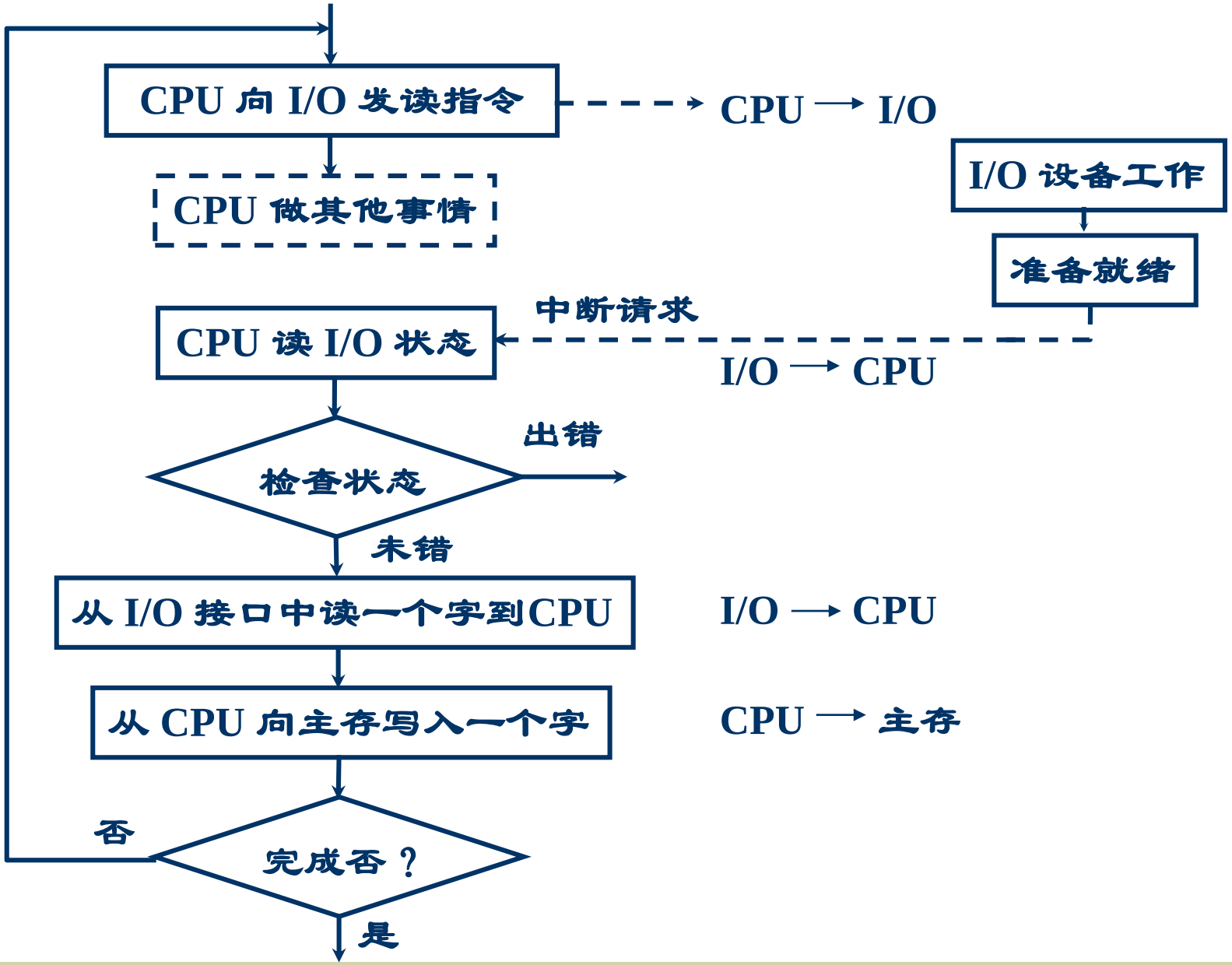
CPU 和 I/O 并行工作



没有踏步等待现象

中断现行程序

程序中断方式流程


CPU 向 I/O 发读指令
CPU → I/O
CPU 做其他事情
I/O 设备工作
准备就绪
中断请求
CPU 读 I/O 状态
I/O → CPU
出错
检查状态
未错
从 I/O 接口中读一个字到CPU
I/O → CPU
从 CPU 向主存写入一个字
CPU → 主存
否
完成否？
是

直接存储器存取（DMA）方式

- 直接存储器存取方式也称为成组传送方式
- 为什么使用DMA方式？
  - 减少大批量数据传输时CPU的开销
    - 可大大减轻CPU负担，CPU只需对DMA控制器进行初期化处理和后处理
  - 解决高速IO设备可能丢失数据问题，满足IO数据交换速度要求
    - 高速IO设备（磁盘等）数据传输速度已经接近于主存储器（DRAM）的工作速度，程序查询和中断方式不能满足要求；
  - 因此，从性能和成本方面综合考虑，必须在IO设备与RAM之间建立直接的数据传送通道，即DMA方式

- DMA方式数据传输特点：
  - 以数据块为单位
  - 主要用于高速的I/O设备，如网卡、磁带、磁盘、模数转换器等设备
  - CPU和外围设备并行工作，且整个数据传送过程不需要CPU的干预
  - I/O和CPU竞争使用存储器
- DMA方式传送数据方法：
  - 采用专用部件（DMA控制器）生成访存地址并控制访存过程，使I/O设备直接和存储器进行成批数据的快速传送
    - DMA控制器将一组数据(块)直接从IO设备送到存储器
    - DMA控制器直接从存储器取出一组数据送到I/O设备

DMA数据传送控制过程

step1、数据传送前，CPU对DMA控制器中控制寄存器初始化

① IO设备准备就绪时，向CPU申请中断服务

② CPU设置DMA控制器的**主存地址寄存器**（主存缓冲区首址）、**设备地址**（如磁盘存储器的物理地址）以及**数据块的长度计数器**

③ 启动DMA设备

step2、 DMA控制器控制设备与存储器直接进行块数据交换

- I/O设备和主存之间在DMA控制器的控制下进行直接数据传输
- 主存地址寄存器随着数据传输的进行而递减/加改变
- 直到主存地址寄存器和计数器到规定值为止

step3、数据传送结束后，CPU进行后处理

- 当计数器的值减到0时，DMA控制器撤销总线请求信号HOLD，整个DMA数据传输过程全部结束。

DMA控制器 （Intel 8237A）的基本结构

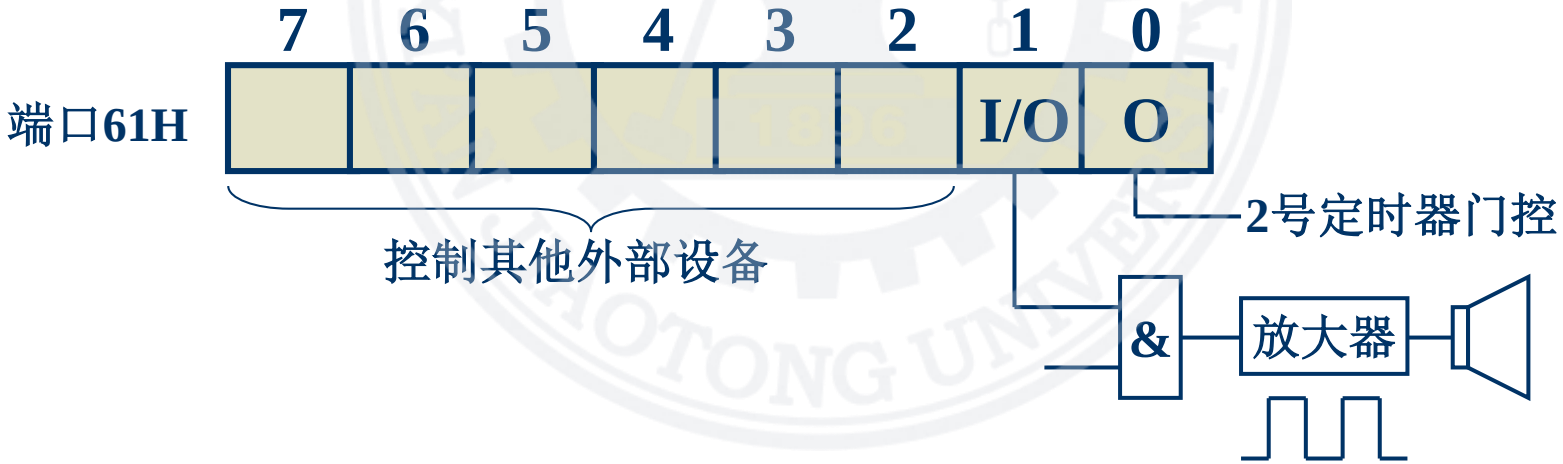
包括4个寄存器：

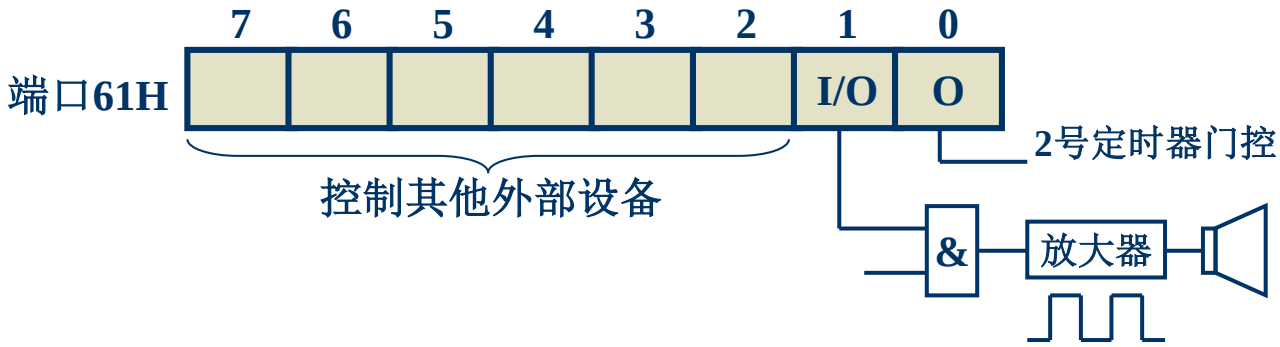
- 控制寄存器
- 状态寄存器
- 地址寄存器
  - 数据块在存储器中启始地址
- 字节计数器
  - 传送数据块的字节数

8.2 I/O程序举例

例8.1 P308： SOUND子程序

- 设备控制寄存器（I/O端口地址为61H）
- 第1位和扬声器的脉冲门相连
- 第1位交替为0和1，就产生了脉冲，根据0和1的延迟控制脉冲频率

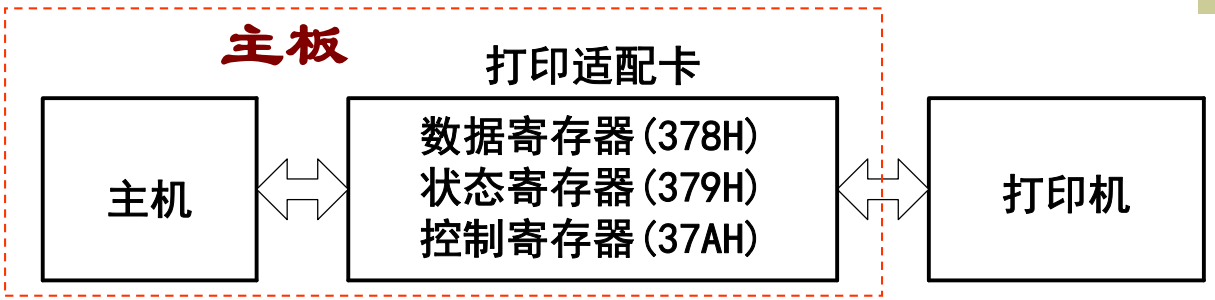





```
; BX: 声音频率（0和1的延迟），  CX：发声时间
;SOUND  PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     DX, CX
        IN      AL, 61H
        AND     AL, 11111100B
TRIG:   XOR     AL, 00000010B   ; D1求反，其他位不变
        OUT     61H, AL
        MOV     CX, BX
DELAY:  LOOP    DELAY           } 0和1的延迟，输出电平宽度
        DEC     DX
        JNE     TRIG
        POP     DX
        POP     AX
        RET
SOUND   ENDP
```

例8.2　p310：PRT_CHAR程序

- 查询方式打印输出



主机、打印机与接口

状态寄存器(379H)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

D7　忙位（0=忙）
D6　应答（0=可接受）
D5　纸出界（1=纸尽）
D4　联机状态（1=联机）
D3　打印错误（0=出错）
D2、D1、D0　保留未用

控制寄存器(37AH)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

D7、D6、D5保留
D4　控制方式（1=允许中断）
D3　选择位（1=接通打印机）
D2　初始化（0=初始化打印机）
D1　自动换行（1=换行）
D0　数据选通（1=输出数据）

打印机状态寄存器和控制寄存器各位的定义

状态寄存器(379H)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

D7　忙位（0=忙）
D6　应答（0=可接受）
D5　纸出界（1=纸尽）
D4　联机状态（1=联机）
D3　打印错误（0=出错）
D2、D1、D0　保留未用

控制寄存器(37AH)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

D7、D6、D5保留
D4　控制方式（1=允许中断）
D3　选择位（1=接通打印机）
D2　初始化（0=初始化打印机）
D1　自动换行（1=换行）
D0　数据选通（1=输出数据）

打印机状态寄存器和控制寄存器各位的定义

0dh=00001101b
0ch=00001100b

```
data    segment
        mess    db 'Printer is normal', 0dh, 0ah
        count   equ  $-mess
data    ends
cseg    segment
main    proc    far
        assume cs:cseg, ds:data
start:  mov     si, offset mess
        mov     cx, count
next:   mov     dx, 379h
wait:   in      al, dx
        test    al, 80h
        je      wait
        mov     al, [si]
        mov     dx, 378h
        out     dx, al
```

判断打印机是否忙

输出字符到接口的数据寄存器

```
        mov     dx, 37ah
        mov     al, 0dh
        out     dx, al
        mov     al, 0ch
        out     dx, al
        inc     si
        loop    next
        mov     ah, 4ch
        int     21h
main    endp
cseg    ends
        end     start
```

产生数据选通等信号

返回DOS

8.3 中断传送方式

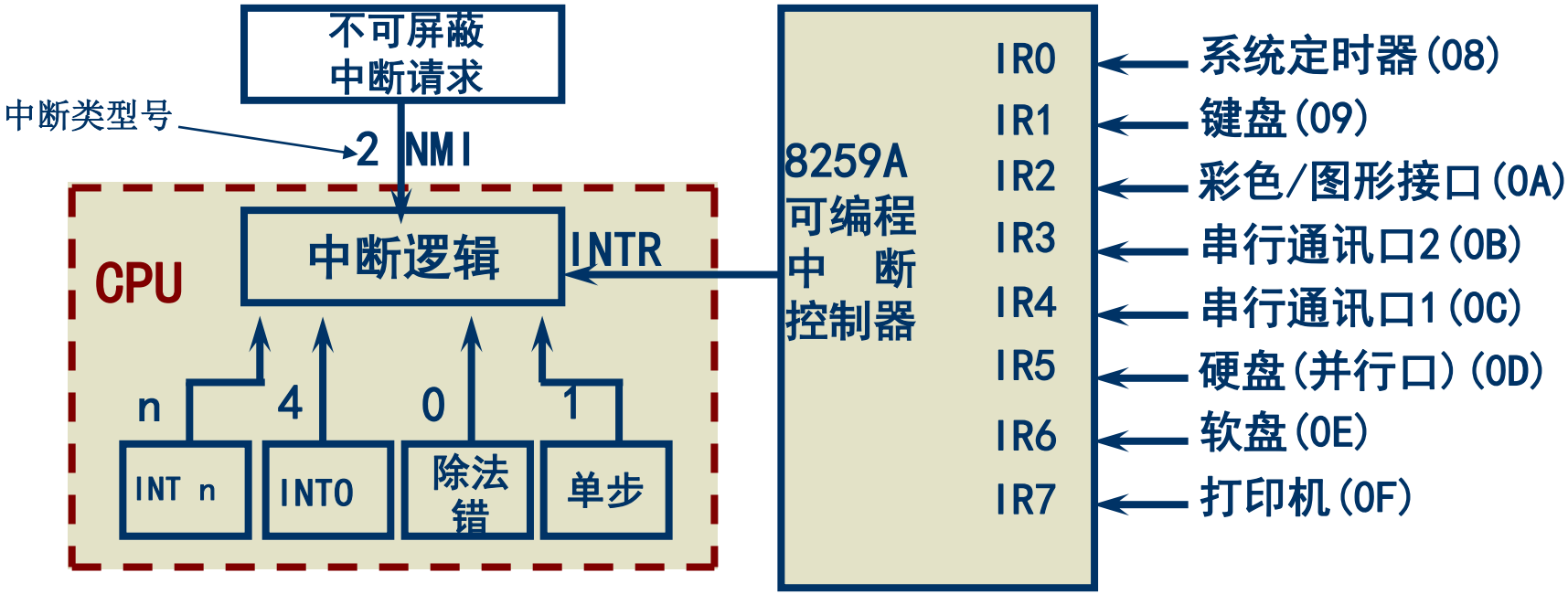
本节主要介绍如下内容：

1. 中断的概念
2. 中断的分类
3. 中断向量表
4. 中断过程
5. 用户自定义中断
6. 中断优先级
7. 中断嵌套

1、中断的概念

- **中断的概念：** 计算机在执行正常程序的过程中，当出现异常事件或事先安排好的事件，迫使CPU暂时中止现行程序的执行，转去执行另一处理程序；当处理完后，CPU再返回到被暂时中止的程序，接着执行被暂时中止的程序。这个过程称为中断
- **中断源：** 引起中断的事件。
  - 如外设输入/输出请求，计算机异常事故或其他内部原因

80X86中断源



- 不受中断标志位IF屏蔽：
  - 非屏蔽中断(NMI)：电源错、内存和总线奇偶等异常，中断类型号=2
  - 程序中INT指令、运算结果异常等
- 受中断标志位IF屏蔽：
  - 可屏蔽的外部设备中断请求(INTR)

2、中断的分类

- 8086/8088可处理256种中断，对应的中断类型码为0～255
- 按照引起中断的方式，中断可分为：
  ① 硬件中断（或外中断）：外设控制器或协处理器引起的中断
  ② 软件中断或内中断：程序中的中断指令INT或CPU错误结果产生的中断

课内测试CH08-2

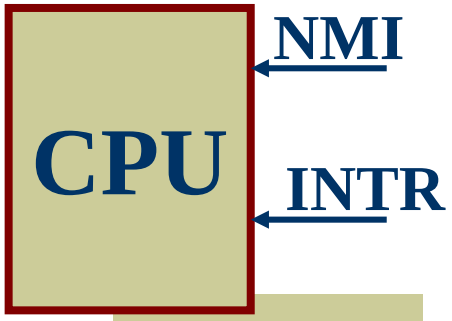
1. 请在填空中 [填空1] 填写“86”（10分）；

2. 根据中断源特点，外部硬中断分类有:（10分）

① [填空2]中断

② [填空3]中断

① 硬件中断

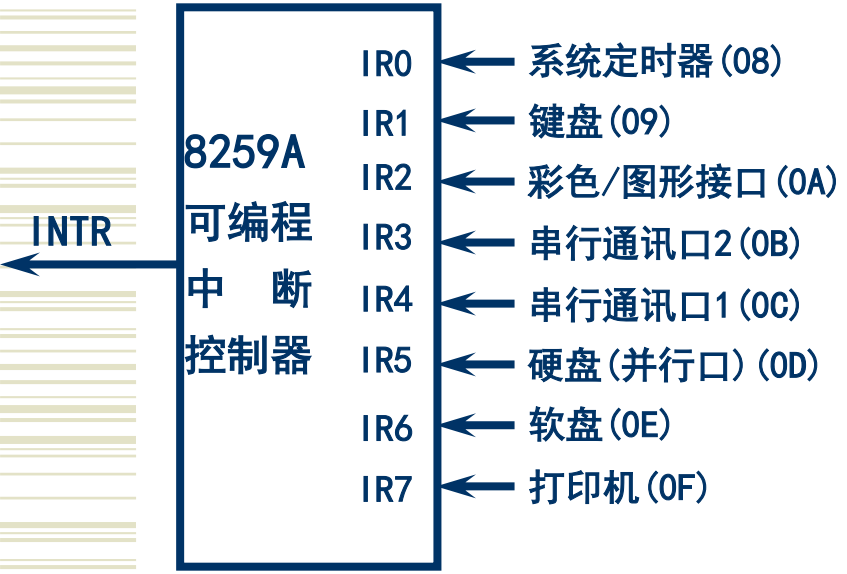


- 硬件中断由外部硬件产生，也称外部中断
- 硬件中断分两类：
  - 非屏蔽中断 (NMI: Non Maskable Interrupt)
    - 非屏蔽中断，通过8086/8088的NMI脚引入，它不受中断允许标志IF的屏蔽；中断类型号=2
  - 可屏蔽中断 (Maskable Interrupt)
    - 可屏蔽中断通过CPU的INTR脚引入，只有当标志寄存器的中断屏蔽标志IF为1时，才能引起中断
    - 开中断指令： STI ，设置中断允许位 (IF=1)
    - 关中断指令： CLI ，清除中断允许位 (IF=0)

■ 系统中，通过中断控制器（如8259A）的配合工作，并8259A 的树型连接8086/8088可处理两百多个可屏蔽中断

■ 8259A中与中断相关寄存器

- 中断屏蔽寄存器(IMR) ： 屏蔽单个中断请求
- 中断命令寄存器



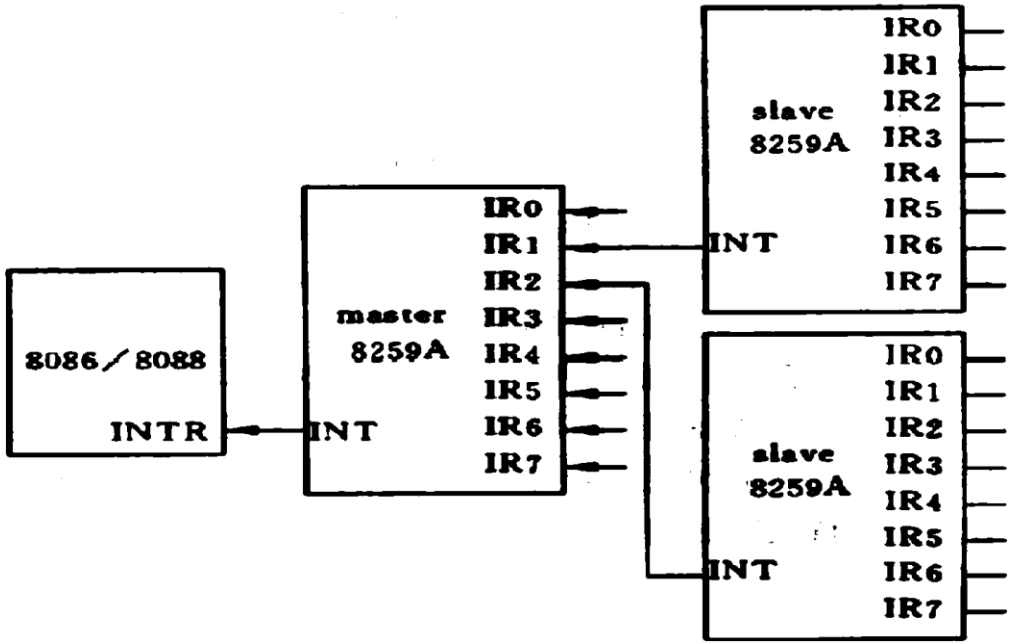
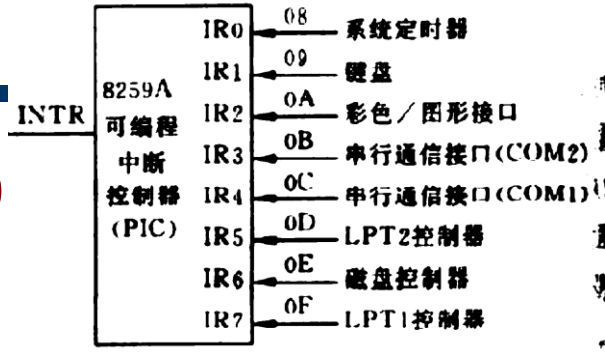


图 8.9 多级 8259A 中断系统



8259A的中断屏蔽寄存器(IMR)

- IMR的I/O端口地址 = 21H
- 8位对应8个外部设备，允许/禁止某设备产生中断

7	6	5	4	3	2	1	0
打印机	软盘	硬盘	COM1	COM2	彩显	键盘	定时器

- =0时，允许中断请求
- =1时，禁止中断请求

例：只允许键盘中断，设置中断屏蔽字

```
MOV   AL,   11111101B
OUT   21H,   AL
```

例：新增设允许键盘中断，设置中断屏蔽字

```
IN   AL,   21H
AND   AL,   11111101B
OUT   21H,   AL
```

CPU可以响应某设备的中断服务请求的条件

- 中断屏蔽寄存器中对应位=0，同时IF=1

屏蔽中断方式

1. 屏蔽所有可屏蔽的外部设备中断请求

- 用标志寄存器中的中断屏蔽标志位IF
  - 开中断指令：STI，设置中断允许位（IF=1）
  - 关中断指令：CLI，清除中断允许位（IF=0）

2. 屏蔽单个可屏蔽的外部设备中断请求

- 用8259A中的中断屏蔽寄存器对应屏蔽位
  - OUT指令

3. 禁止单个中断源的中断请求

- 用设备接口中控制寄存器对应位
  - OUT指令

状态寄存器(379H)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

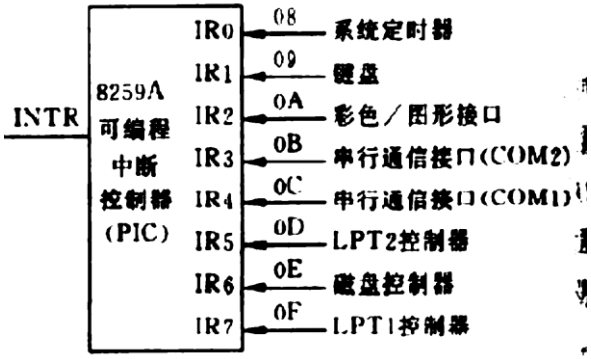
D7 忙位（0=忙）
D6 应答（0=可接受）
D5 纸出界（1=纸尽）
D4 联机状态（1=联机）
D3 打印错误（0=出错）
D2、D1、D0 保留未用

控制寄存器(37AH)各位的定义

D7	D6	D5	D4	D3	D2	D1	D0

D7、D6、D5保留
D4 控制方式（1=允许中断请求）
D3 选择位（1=接通打印机）
D2 初始化（0=初始化打印机）
D1 自动换行（1=换行）
D0 数据选通（1=输出数据）

打印机状态寄存器和控制寄存器各位的定义



中断命令寄存器

- I/O端口地址 = 20H
- 8位含义

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

  - L2-L0: 指定IR0-IR7中哪个中断优先级最低
  - R(rotate), SL(set level)控制IR0-IR7中断优先顺序
  - EOI: 中断结束，当EOI=1时，将当前中断请求清除

- 中断服务程序中，中断处理结束前，应将EOI置1
  - 结束硬件中断指令

```
IN  AL,   20H
OR  AL,   20H
OUT 20H,  AL
```

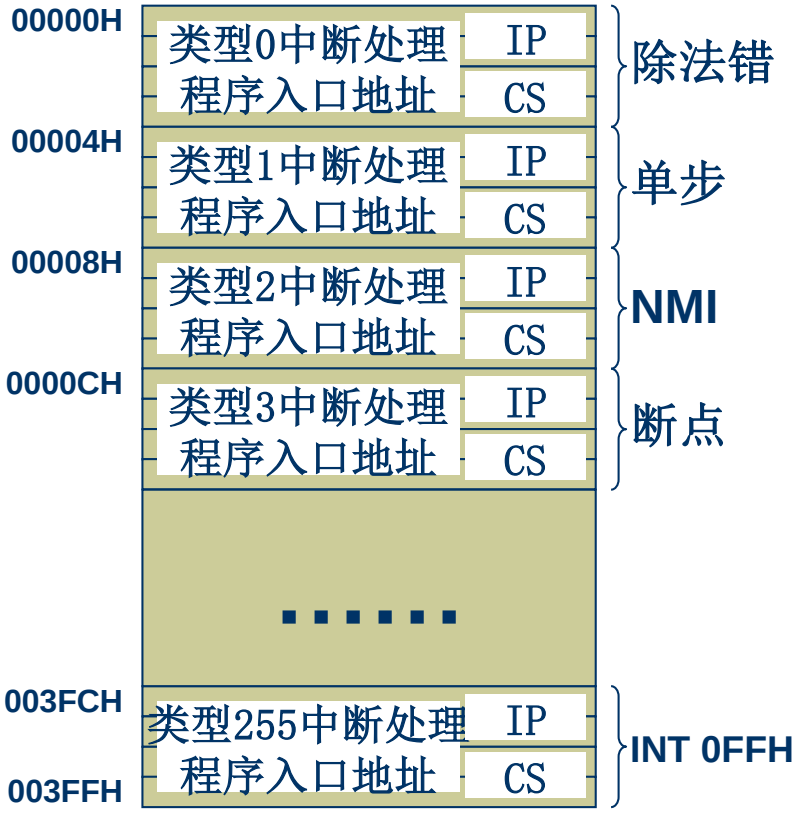
  - 硬件中断服务程序结束前要有这几条指令

② 软件中断

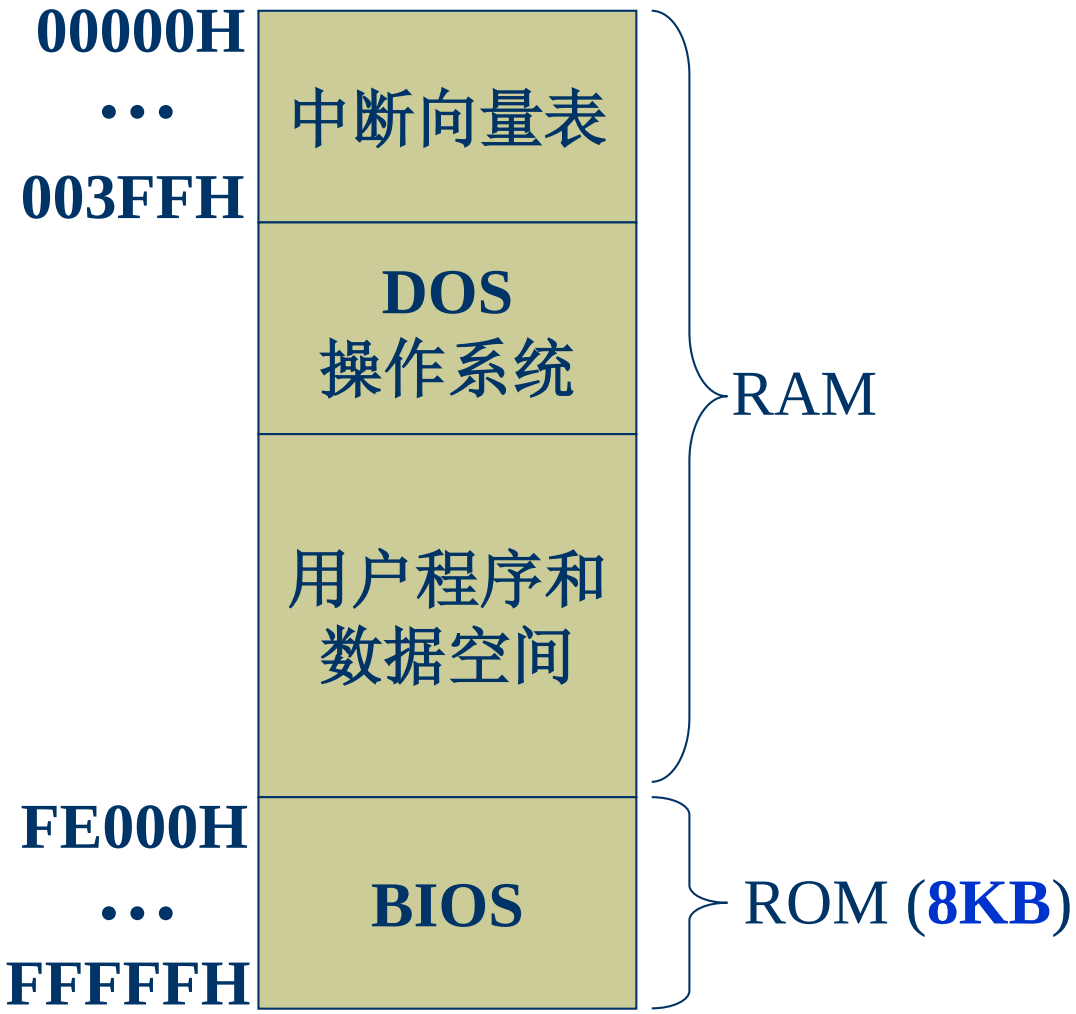
- 软件中断也称内部中断，由3种情况引起
  1. 程序中的中断指令 INT n
     - 操作数n指出中断类型号，0—FFH
     - 如 INT 12H ；存储器容量测试
  2. CPU的某些运行结果
     - 除法错中断：除数为零/商超出表数范围，中断类型号为0的内部中断
     - 溢出中断：运算结果溢出，OF=1，INTO指令将引起类型为4的内部中断
  3. 调试程序（DEBUG）设置的中断
     - 单步中断：标志位TF=1时，中断类型号=1
     - 断点中断：将程序分段，每段设置一个断点（INT 3），中断类型号=3
- 软件中断不受中断屏蔽标志IF影响，不可屏蔽中断

3、中断向量表

- 80X86有256种类型的中断
  - 每种中断有一个中断类型号，类型号 0-FFH
  - 每种类型中断都由相应的中断处理程序处理
- 中断向量表
  - 各类型中断处理程序的入口地址表
  - 保存在存储器中， 1KB (00000H-003FFH)
    - 每类型中断向量占4字节，对应中断处理程序入口CS、IP值
    - 每类中断向量表地址=4×中断类型号n



X86的空间内存分配：


00000H
…
003FFH
中断向量表
DOS
操作系统
用户程序和
数据空间
RAM
FE000H
…
FFFFFH
BIOS
ROM (8KB)

以软中断为例说明中断过程

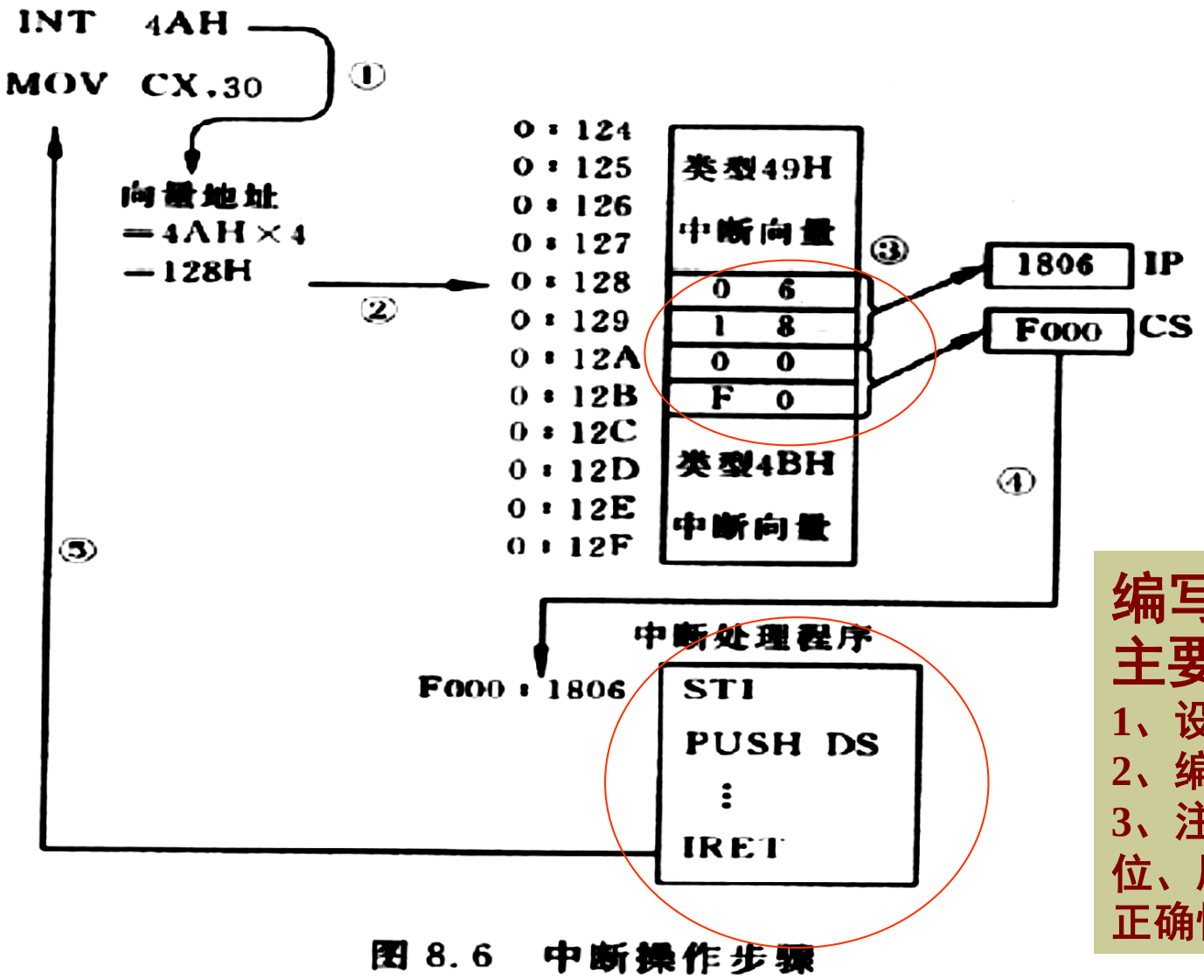


图 8.6 中断操作步骤

编写中断程序时主要工作有：

1、设置中断向量
2、编写中断处理程序
3、注意相关中断允许位、屏蔽位、优先级正确性

地 址	中断类型号	
0~7F	0~1F	BIOS 中断向量
80~FF	20~3F	DOS 中断向量
100~17F	40~5F	扩充 BIOS 中断向量
180~19F	60~67	用户中断向量
1A0~1BF	68~6F	保留
1C0~1DF	70~77	I / O 设备中断向量
1E0~1FF	78~7F	保留
200~3C3	80~FD	BASIC
3C4~3FF	F1~FF	保留

详见附录3，p603

BIOS中断：BIOS中提供的各中断程序
DOS中断：DOS中提供的各中断程序
IO设备中断： 各种IO设备的中断程序

中断类型号的分配

分类	中断类型码	地址（0000H段）	功能
系统内中断（BIOS）	0	0000H~0003H	被零除
	1	0004H~0007H	单步
	2	0008H~000BH	不可屏蔽
	3	000CH~000FH	断点
	4	0010H~0013H	溢出
	5	0014H~0017H	打印屏幕
	6	0018H~001BH	保留
	7	001CH~001FH	保留
分类	中断类型码	地址（0000H段）	功能
系统8级外中断（BIOS）	8	0020H~0023H	日时钟
	9	0024H~0027H	键盘
	A	0028H~002BH	保留
	B	002CH~002FH	异步通信（2）
	C	0030H~0033H	异步通信（1）
	D	0034H~0037H	硬盘
	E	0038H~003BH	软盘
	F	003CH~003FH	打印机

这些功能可以用中断方式调用INT n

分类	中断类型码	地址（0000H段）	功能
设备驱动（BIOS）	10	0040H~0043H	显示
	11	0044H~0047H	设备配制
	12	0048H~004BH	存储容量
	13	004CH~004FH	硬盘I/O
	14	0050H~0053H	通信I/O
	15	0054H~0057H	盒式磁带I/O
	16	0058H~005BH	键盘I/O
	17	005CH~005FH	打印机I/O
	18	0060H~0063H	ROM BASIC
	19	0064H~0067H	系统自举
	1A	0068H~006BH	日时钟I/O
	1B	006CH~006FH	键盘中断地址
	1C	0070H~0073H	定时器报时
	1D	0074H~0077H	显示器参数
	1E	0078H~007BH	软盘参数
	1F	007CH~007FH	图形字符扩展
分类	中断类型码	地址（0000H段）	功能
DOS	20~2F	0080H~00BFH	DOS调用
	30~3F	00C0H~00FFH	为DOS保留

◆ 存取中断向量的DOS功能调用 (21H)

■ 设置中断向量

预置：AH=25H

AL=中断类型号

DS:DX=中断向量（中断程序入口地址）

执行：INT 21H

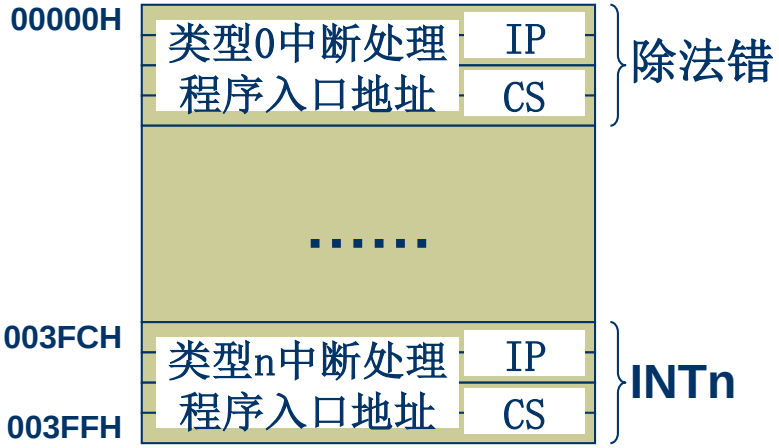
■ 取中断向量

预置：AH=35H

AL=中断类型号

执行：INT 21H

返回：ES:BX=中断向量（中断程序入口地址）



例8.4　使用DOS功能调用存取中断向量

保存原中断向量

```
…
MOV     AL,     N
MOV     AH,     35H
INT     21H
PUSH    ES
PUSH    BX
PUSH    DS
```

设置新中断向量

```
MOV     AX,     SEG   INTHAND
MOV     DS,     AX
MOV     DX,     OFFSET   INTHAND
MOV     AL,     N
MOV     AH,     25H
INT     21H
POP     DS
…
```

恢复原中断向量

```
…
POP     DX
POP     DS
MOV     AL,     N
MOV     AH,     25H
INT     21H
RET
```

拥护自定义中断程序

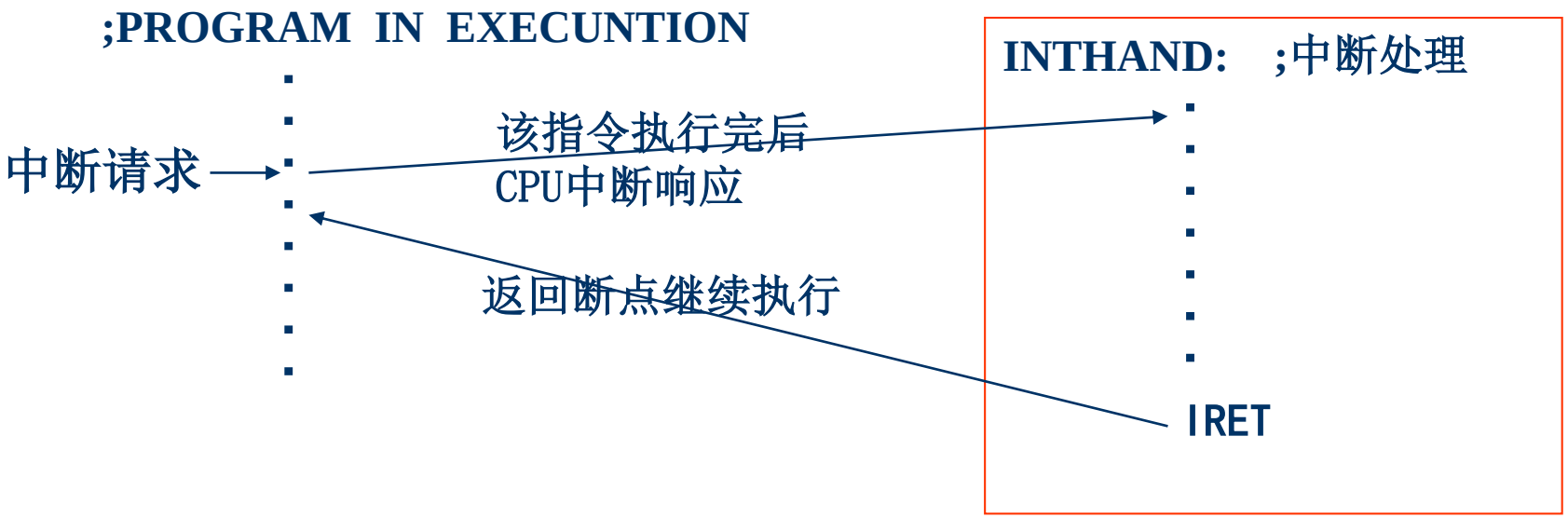
```
INHAND      PROC  FAR
            …
            …
            IRET
```

4、中断过程

① 中断响应

② 中断处理

③ 中断返回



课内测试CH08-3

1. 请在填空中 [填空1] 填写“82”（15分）；

2. 中断过程有3个步骤：（15分）

① 中断[填空2]

② 中断[填空3]

③ 中断[填空4]

中断响应过程

- 中断响应时，由CPU自动完成如下操作

① 取中断类型号N

② 标志寄存器(FLAGS)内容入栈

③ 保存被中断程序断点
  - 当前代码段寄存器(CS)内容入栈
  - 当前指令指针寄存器(IP)内容入栈

④ 禁止硬件中断和单步中断
  - IF 清 0
  - TF 清 0

⑤ 转中断服务处理程序入口地址
  - 从中断向量表中取4*N单元的字内容 送 IP
  - 从中断向量表中取4*N+2单元的字内容 送 CS

注意：如果在中断处理程序中允许处理可屏蔽中断，要重新设置IF为1

中断处理过程
（中断处理程序）

- 中断功能处理与子程序功能处理类似，但注意几点
  - 保存现场：中断发生是不可预知的，除CS、IP、SS、SP外，一般凡用到的寄存器都应保存入堆栈
  - 如果中断处理程序中允许外部中断，要重新设置IF为1
- 一般中断处理程序设计格式
  1. 保存现场
  2. 开中断(STI)　（根据需要确定）
  3. 中断处理程序主体部分
  4. 中断结束（EOI）（硬件中断时）
  5. 关中断(CLI)
  6. 恢复现场
  7. 中断返回(IRET)

中断返回过程

- 中断返回时（IRET），由CPU自动完成如下操作

① 恢复被中断程序断点程序指针

- 被中断程序指令指针寄存器(IP)内容恢复
- 被中断程序代码段寄存器(CS)内容恢复

② 标志寄存器(FLAGS)内容从栈恢复

中断与子程序调用

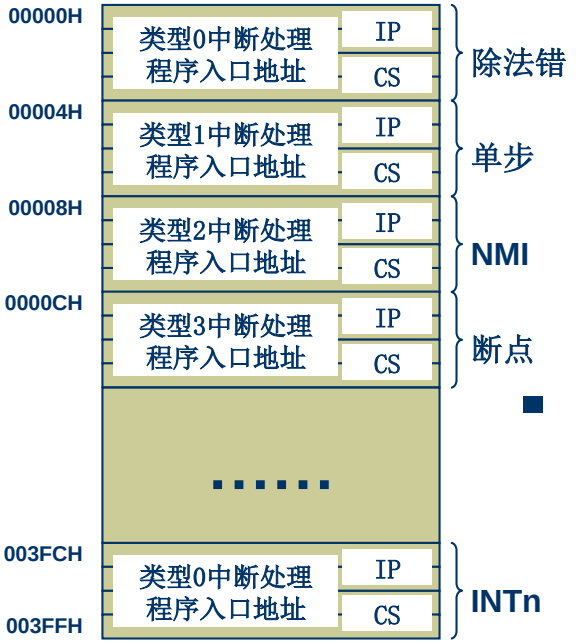
- 中断与子程序调用处理过程相似
- 差别主要在于进入和返回时的处理不同
  - 进入中断服务处理程序时
    - 子程序调用：只把CS和IP压入堆栈
    - 中断：除把CS和IP压入堆栈外，还把标志寄存器的内容压入堆栈，并且关掉了中断和单步运行方式
  - 返回时
    - 子程序返回：只把断点地址从堆栈弹出送CS和IP
    - 中断返回：除恢复断点地址CS和IP外，还要恢复标志寄存器的内容
- 时机不同：
  - 中断：一般随机发生；软中断在程序中预先安排
  - 子程序调用：程序中预先安排调用

5、用户自定义中断

- 用户可以用保留的中断类型，扩充自定义中断功能
- 新增加中断时
  - 编写中断服务处理程序
  - 主程序中：
    - 设置中断向量表中相应的中断向量
    - 正确设置中断允许、屏蔽位和优先级

地 址	中断类型号	
0～7F	0～1F	BIOS 中断向量
80～FF	20～3F	DOS 中断向量
100～17F	40～5F	扩充 BIOS 中断向量
180～19F	60～67	用户中断向量
1A0～1BF	68～6F	保留

地 址	中断类型号	
1C0～1DF	70～77	I / O 设备中断向量
1E0～1FF	78～7F	保留
200～3C3	80～FD	BASIC
3C4～3FF	F1～FF	保留



- 例如：用户自定义中断类型n

```
; 设置中断向量
   MOV   AX,  0
   MOV   ES,  AX          ; 设置中断向量基地址
   MOV   BX,  N*4         ; 设置中断向量n偏移地址
   MOV   AX,  OFFSET  INTHAND
   MOV   ES:WORD PTR[BX],  AX    ; 设置中断处理程序入口地址偏移量
   MOV   AX,  SEG  INTHAND
   MOV   ES:WORD PTR[BX+2],  AX  ; 设置中断处理程序入口段基地址
   ..... ; 等待中断请求，  或者使用软中断指令INT n进入中断服务子程序
```

也可以用DOS调用设置新中断向量

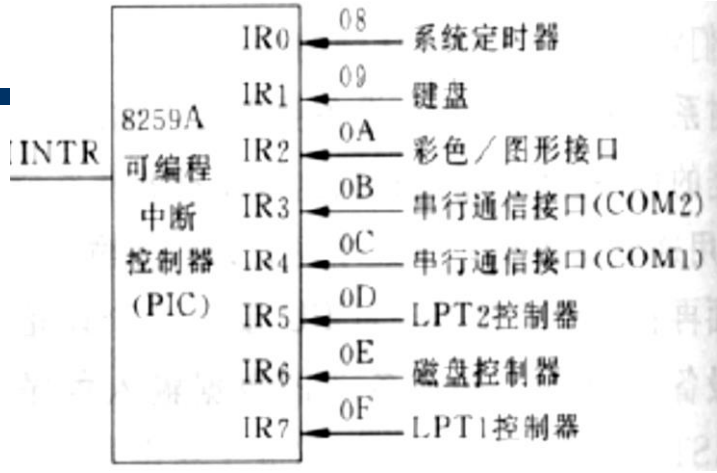
```
; 中断处理程序
INTHAND   PROC   FAR
   .....      ; 中断服务功能处理
   IRET
```

6、中断优先级

- 当多个中断源同时向CPU请求中断时，CPU应如何处理？
  - 制定优先级别，先为优先级别高的中断服务
- **中断优先级(Priority)**：当多个中断源同时向CPU请求中断时，CPU确定优先处理的次序
- 80X86中规定的中断优先级次序

优先级高 ↓ 低

  - 软件中断（除法错，INTO，INT）
  - 非屏蔽中断（NMI）
  - 可屏蔽中断（INTR）
  - 单步中断



- **可屏蔽中断优先级分8级**
  - 由8259A中断控制方式确定
  - 正常默认次序由高到低为：

IR0, IR1, IR2, IR3, IR4, IR5, IR6, IR7

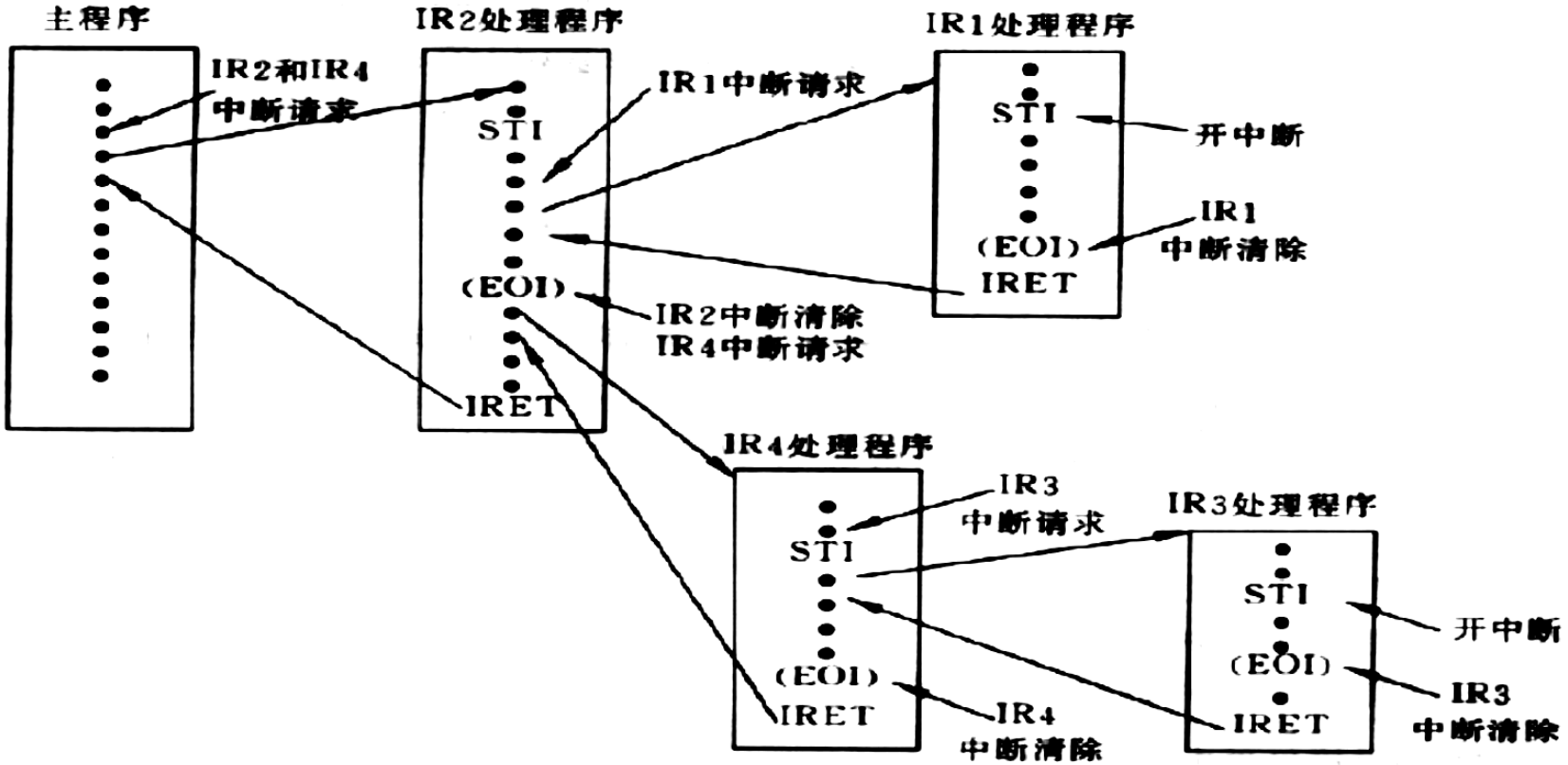
- 8259A的中断命令寄存器的6、7位可控制优先级次序

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

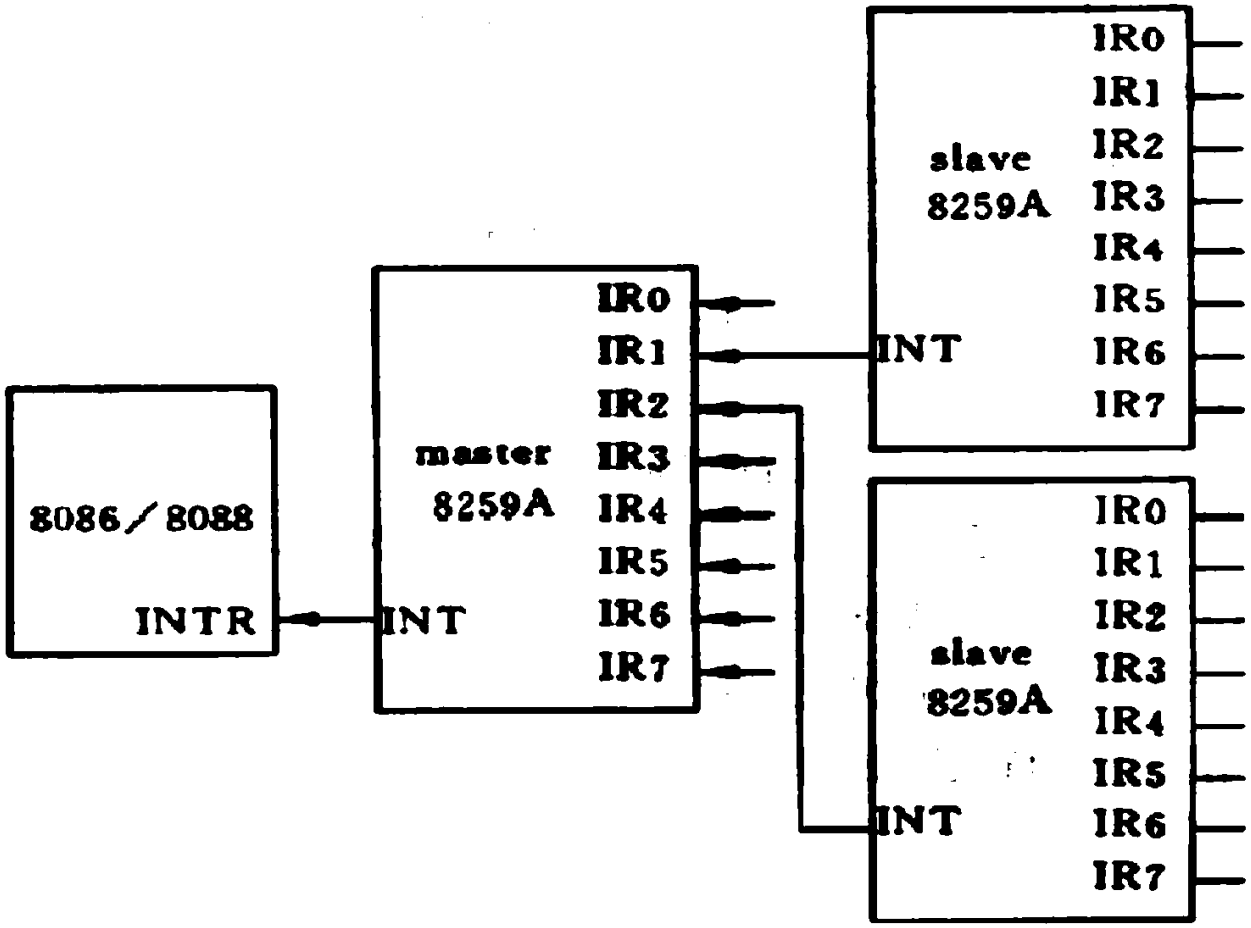
  - R, SL=00时， 正常优先级方式
  - R, SL=01时， 清除由L2-L0指定的中断请求
  - R, SL=10时， 各中断优先级依次左循环一个位置
  - R, SL=11时， 各中断优先级依次左循环，直到由L2-L0指定的中断源的优先级最低
- 根据需要给端口20H送命令， 改变优先级

7、中断嵌套

- **中断嵌套：正在运行的中断处理程序，又被其他中断源的中断请求中断**



正常优先级方式下的典型中断序列



多级 8259A 中断系统

最高优先级

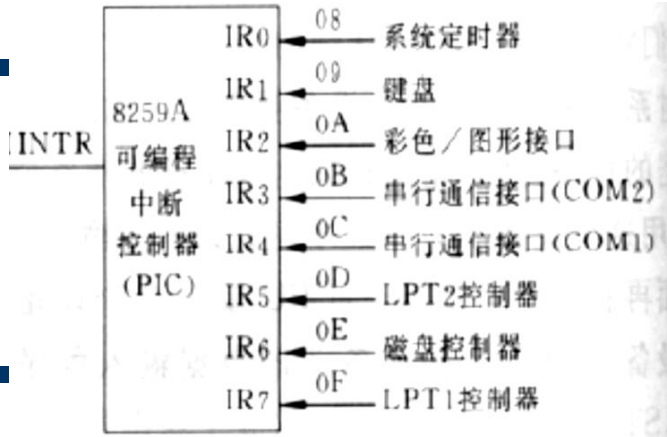
最低优先级

主8259A:IR0，

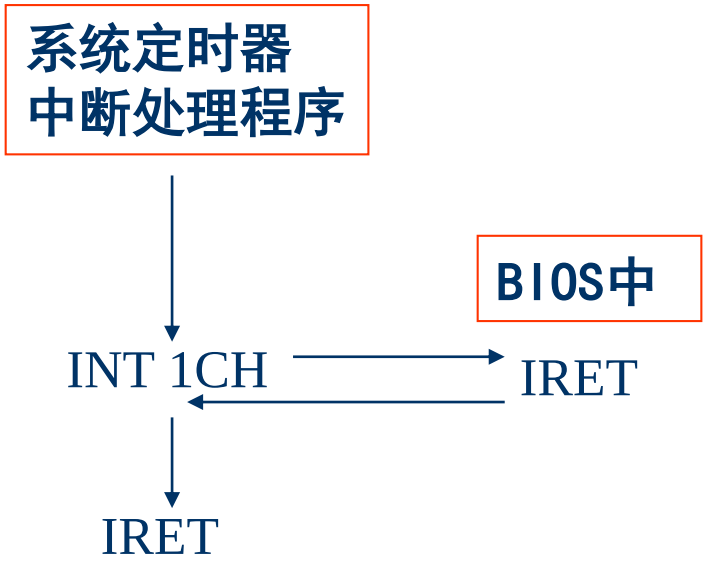
从8259A: IR0, IR1, IR2, IR3, IR4, IR5, IR6, IR7， IR0, IR1, IR2, IR3, IR4, IR5, IR6, IR7，

主8259A: IR3, IR4, IR5， IR5， IR7

中断程序举例



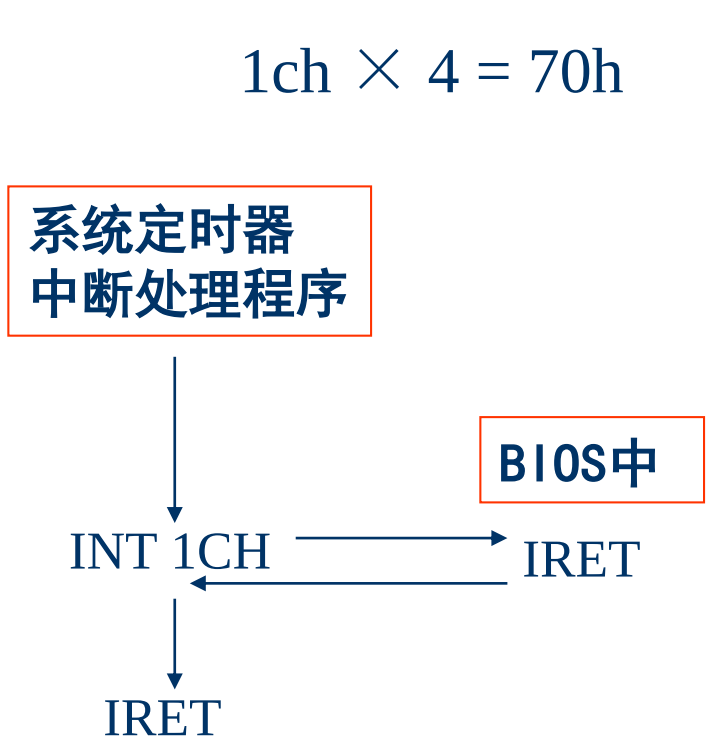
- 例8.5： p323
  - 要求每10秒响铃一次，并显示“The bell is ring!”
  - 要点：如何设计中断处理程序；如何进入中断处理程序
  - 可用资源：系统定时器（中断类型8，每秒中断18.2次）
    - 系统定时器的中断处理程序中，有一条指令INT 1CH，但嵌套调用后BIOS中只有IRET指令。用户可实现完成某些周期性工作，但影响系统时钟

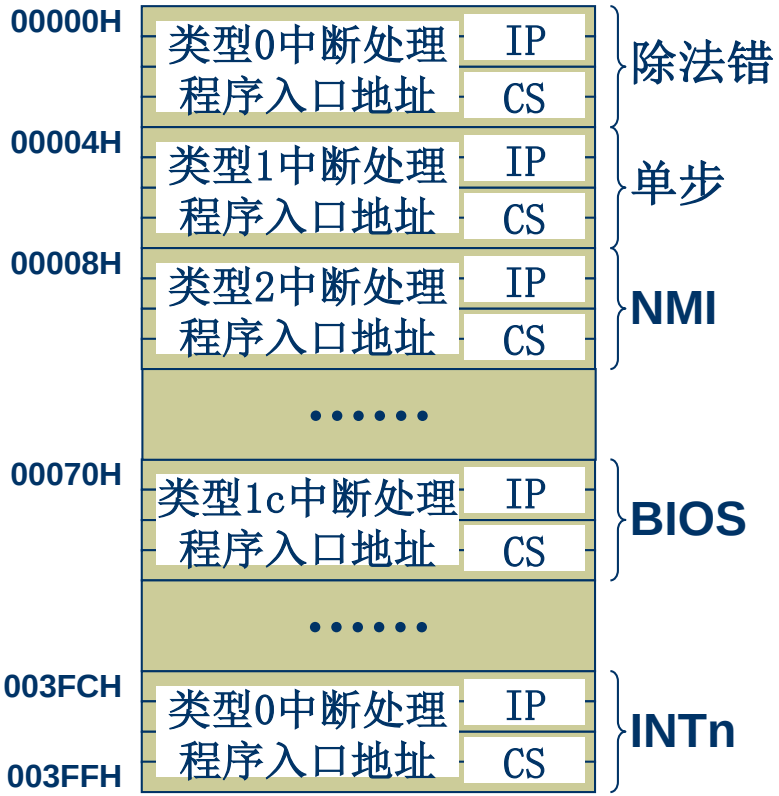


- 中断类型1CH作为用户使用的中断类型，可能已被其他功能的程序使用，所以在编写新的临时中断程序时，应做如下工作：

  1、在主程序的初始化部分，先保存当前1CH的中断向量，再设置新的中断向量

  2、在主程序结束部分恢复保存的1CH中断向量



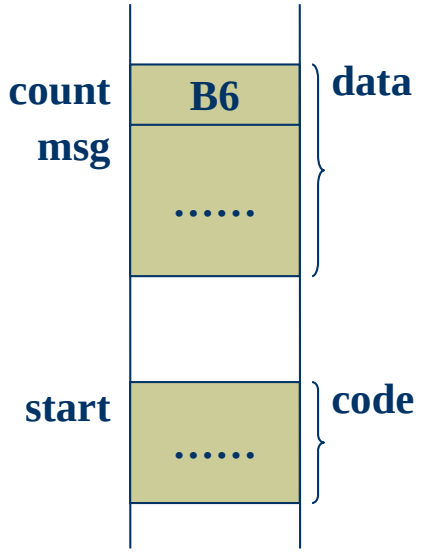


```
;****************************
;eg8-5.asm
;purpose:ring ang display a message every 10 seconds.
;****************************
          .model saml
;-----------------------------------------
          .stack
;-----------------------------------------
          .code
; main program
main      proc      far
start:    move      ax, @data
          mov       ds, ax

;save old interrupt vector
          mov       al, 1ch
          mov       ah, 35h
          int       21h
          push      es
          push      bx

;set new interrupt vector
          push      ds
          mov       dx, offset ring
          mov       ax, seg ring
          mov       ds, ax
          mov       al, 1ch
          mov       ah, 25h
          int       21h
          pop       ds
```

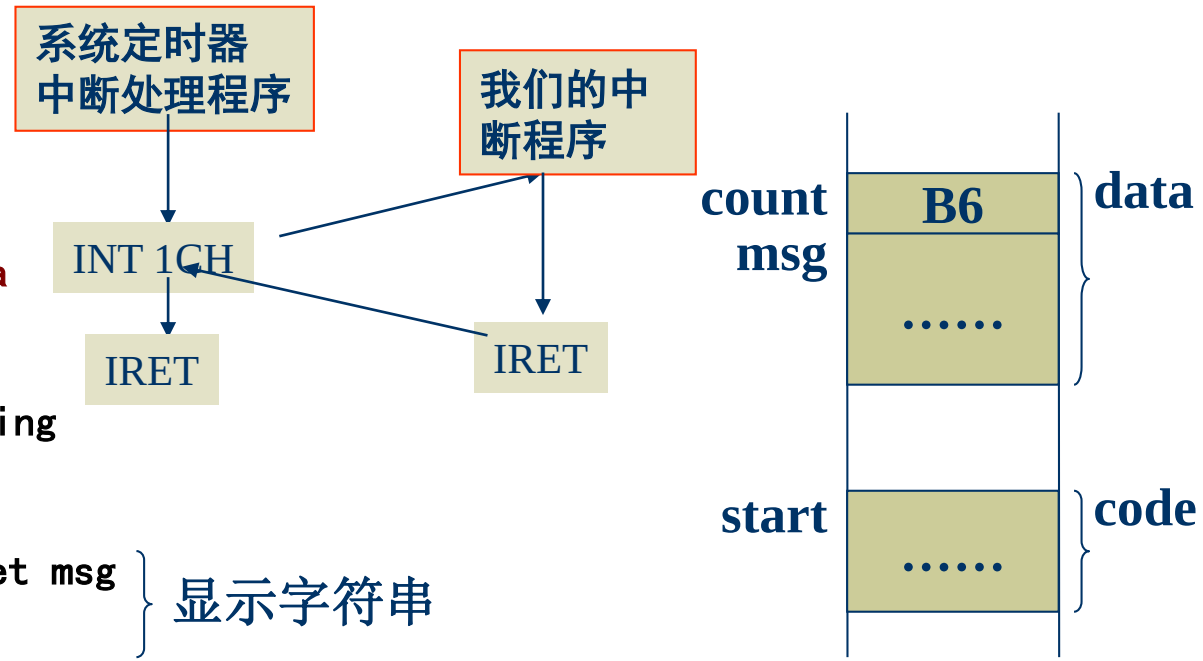
```
          in        al, 21h
          and       al, 11111110b
          out       21h, al
          sti

          mov       di, 20000
deplay:   mov       si, 30000
deplay1:  dec       si
          jnz       deplay1
          dec       di
          jnz       deplay

;restore old interrupt vector
          pop       dx
          pop       ds
          mov       al, 1ch
          mov       ah, 25h
          int       21h

          mov       ax, 4c00h
          int       21h

main      endp
```

- **设置中断向量**
  - 预置：AH=25H
    AL=中断类型号
    DS:DX=中断向量(中断程序入口地址)
  - 执行：INT 21H
- **取中断向量**
  - 预置：AH=35H
    AL=中断类型号
  - 执行：INT 21H
  - 返回：ES:BX=中断向量(中断程序入口地址)

7	6	5	4	3	2	1	0
打印机	软盘	硬盘	COM1	COM2	彩显	键盘	定时器



```
;-----------------------------------------
;Procedure ring
;Purpose:ring every 10 seconds when substitued for interrupt 1ch
ring    proc    near
        push    ds
        push    ax
        push    cx
        push    dx

        mov     ax, @data
        mov     ds, ax
        sti

;Siren if it is time for ring
        dec     count
        jnz     exit

        mov     dx, offset msg
        mov     ah, 09h
        int     21h

        mov     dx, 100
        in      al, 61h
        and     al, 0fch
sound:  xor     al, 02
        out     61h, al

        mov     cx, 1400h
wait1:  loop    wait1
        dec     dx
        jne     sound
        mov     count, 182

exit:   cli
        pop     dx
        pop     cx
        pop     ax
        pop     ds
        iret
ring    endp
;-----------------------------------------
        end     start
```



显示字符串

响铃

课内测试CH08-4

1. 请在填空中 [填空1] 填写“56”（10分）；

2. 在这个例子中，“我们的中断程序”：（10分）

① “我们的中断程序”每次执行的时机是[填空2]

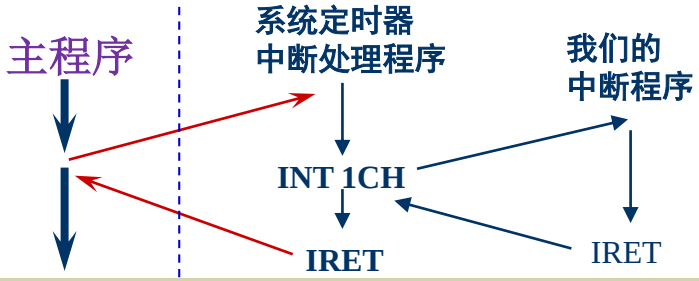
A. 和主程序有关，在主程序中进行中断调用

B. 和主程序无关，在系统定时器中断处理中进行中断调用

② 下面哪个说法对：[填空3]

A. 主程序设置好中断向量，且周期性处理响铃、显示

B. 主程序只设置好中断向量，响铃、显示和主程序无关



8.4 80386输入输出

- 支持65536个I/O端口
- 同样IN、OUT指令只能使用累加器(Al, AX, EAX)与外设通信
- 在保护模式下，I/O指令带有特权级，满足特权级才可执行I/O操作
  - 对IO操作的特权级由EFLAGS中的IOPL指定
  - 过程的CPL高于或等于IOPL（ CPL≤IOPL ）才能执行I/O指令
  - 低特权级过程执行I/O指令，会产生保护故障（中断13）
  - 只有0特权级过程才能修改IOPL
- V86方式下，IOPL仅保护中断标志IF，I/O端口通过TSS中的I/O允许位保护

.....	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF

标志寄存器FLAGS

					OF	DF	IF	TF	SF	ZF		AF		PF		CF	8086/8088
				NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80286
·····				RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80386
·····			AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80486
·····	ID	VIP	VIF	AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	80586
31…	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

80286描述符

31	16	15	0	
Base(B_{15}~B_0)		Limit(L_{15} ~ L_0)		0
0		P **DPL** S TYPE A	Base(B_{23} ~ B_{16})	+4

DPL：特权级（0-3）

80386/80486/Pentium描述符

31	16	15	0	
Base(B_{15}~B_0)		Limit(L_{15} ~ L_0)		0
Base(B_{31} ~ B_{24}) G D/B 0 AVL Limit(L_{19} ~ L_{16})		P **DPL** S TYPE A	Base(B_{23} ~ B_{16})	+4

特权级

1. RPL（Requested PL）请求特权级。保存在段选择符的b_1b_0位。表示本次访问所要求的特权级。
2. DPL（Discriptor PL），描述符特权级。它表示该段或任务门的特权级，分别被保存在段描述符和门描述符的DPL域中，段描述符的DPL具体在访问权字节中的b_6b_5位。
3. CPL（Current PL），当前特权级。CPL是当前运行程序或任务的特权级，分别被保存在段寄存器CS即段选择符的b_1b_0位和当前任务寄存器的b_1b_0位。
4. IOPL（IO PL），对IO操作的特权级。保存在EFLAGS中

- 80386还支持字符串I/O指令
  - INSB/OUTSB ；输入输出字符串
  - INSW/OUTSW ；输入输出字串
  - INSD/OUTSD ；输入输出双字串
- INS指令： DX指定端口地址，结果存于ES:EDI存储单元，按DF指定方向修改EDI
- OUTS指令： DX指定端口地址，源数据存于ES:ESI存储单元，按DF指定方向修改ESI
- 可加重复前缀REP，ECX中为重复计数值

I/O允许位图

- 要完成I/O操作，必须满足两个条件之一：
  - I/O特权级（CPL≤IOPL）
  - 任务的I/O允许位图中的端口访问许可位
    - I/O允许位图保存在任务的状态段TSS
      - 对应端口的位=0，可以访问该端口
      - 对应端口的位=1，不可以访问该端口

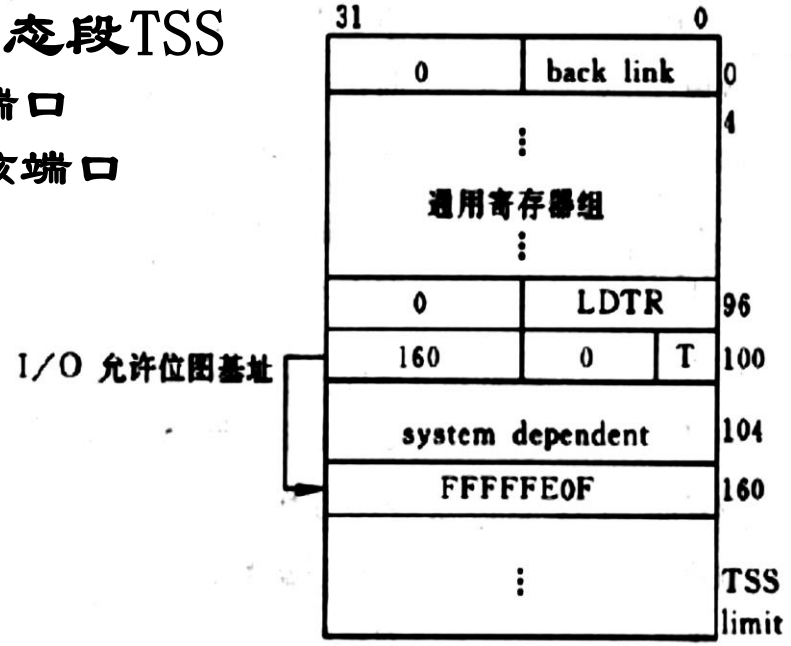


图8.10 TSS中I/O允许位图实例

8.5 80386的中断处理

- 支持3种模式下的中断处理
  - 实模式下与8086中断操作相同
  - 保护模式下，中断寻址根据中断描述符表IDT，每类中断对应IDT中8字节的门描述符
  - 虚拟8086模式下，能模拟虚拟8086的操作方式，但对应保护模式，不对应实模式

8.5.1 80386的中断与异常

中断号	类 型	描 述
0	故障	除错误(仅 DIV、IDIV)
1	故障或陷阱	调试程序中断
2	中断	非屏蔽中断(NMI)
3	陷阱	断点中断
4	陷阱	溢出中断(INTO)
5	故障	超出数组边界(BOUND)
6	故障	无效操作码
7	故障	协处理器无效
8	失败	双故障
9	失败	协处理器段溢出运行(在 80486 上保留)
10	故障	无效 TSS
11	故障	段不存在
12	故障	堆栈段溢出
13	故障	通用保护故障
14	故障	负故障
15	保留	
16	故障	协处理器错误
17	故障	调准检查(仅用于 486,386 保留)
18～30	保留	
31～255	中断或陷阱	系统相关

8.5.2 实地址下的中断处理

- 与8086相同
- INT指令或其他中断请求发生时，自动判断段的寻址方式： 16位、32位
- 中断返回时：
  - IRET指令： 恢复IP、FLAGS
  - IRETD指令： 恢复EIP、EFLAGS

8.5.3 保护方式下的中断

- 中断和异常根据处理方法分为：
  - 中断门(interrupt gate)、陷阱门(trap gate)、任务门(task gate)
- 门描述符
  - 8个字节
  - 寻址相应中断处理程序入口
  - 访问权保护检查

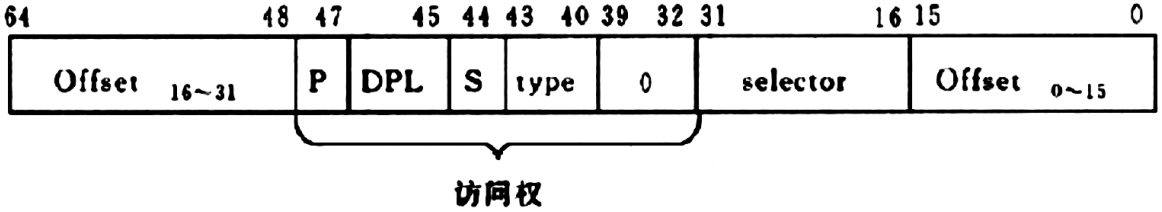
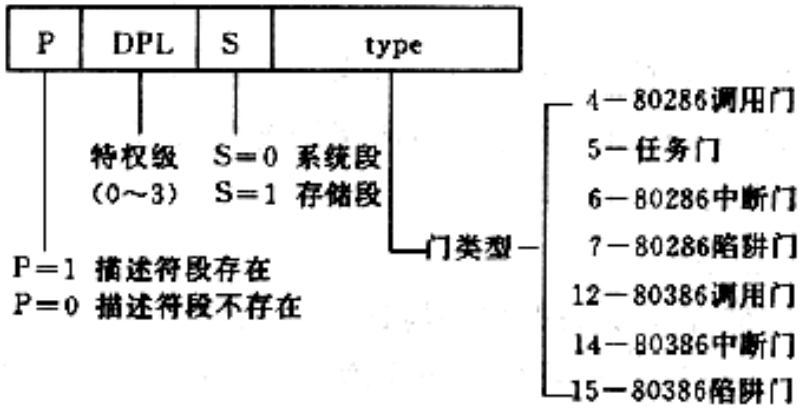
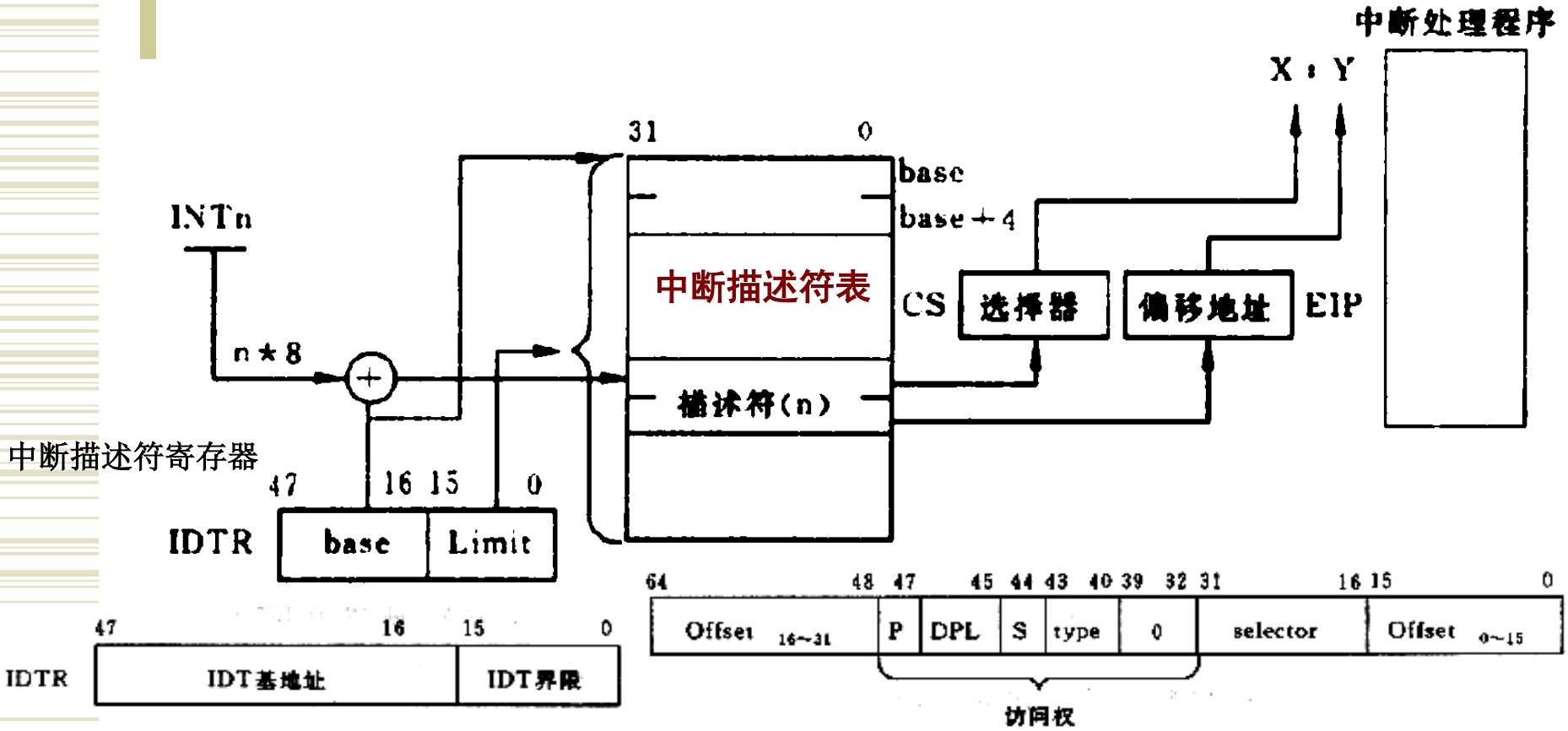


图 8.12 中断门和陷阱门描述符



- 对应任务门时，任务切换
- Selector：16位代码段选择器，直接解释为段基地址
- Offset：32位段内偏移地址

保护模式下中断过程示意



- 中断描述符表相当于实模式下的中断向量表
- 根据中断门描述符获得中断处理程序入口

作　业

8.1　　8.6

8.8　　8.11

第九章　BIOS和DOS中断

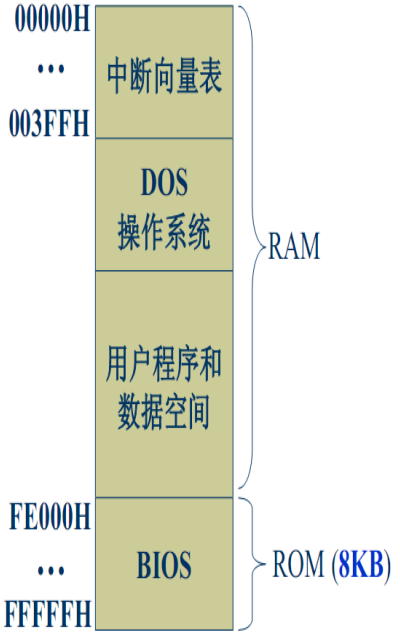
- DOS 和 BIOS 功能调用
- 键盘I/O
- 显示器I/O
- 打印机I/O

本章目标

- 理解DOS/BIOS功能调用概念
- 掌握DOS/BIOS功能调用程序设计方法

BIOS简介

- 固化在ROM中的基本输入输出系统BIOS (Basic Input / Output system)
  - 地址空间：FE000H-FFFFFH，8KB
  - 包含
    - 主要的I/O设备处理和接口控制程序
      - I/O设备硬件中断处理程序
      - I/O设备软件中断调用处理程序
    - 许多常用的系统例行程序
      - 系统加电自检、引导装入等功能模块
  - 一般以中断处理程序的形式存在、被调用
- 例如：软件中断调用
  - 显示输出：10H号中断处理程序
  - 打印输出：17H号中断处理程序
  - 键盘输入：16H号中断处理程序



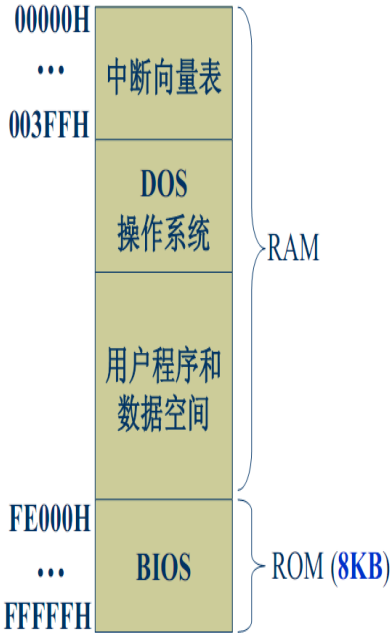
BIOS功能调用

附录5 P611：BIOS软件中断功能调用

- 显示器：INT 10H
- 磁　盘：INT 13H
- 串行口：INT 14H
- 键　盘：INT 16H
- 打印机：INT 17H
- 引导装入：INT 19H
- 鼠　标：INT 33H

DOS简介

- 磁盘操作系统DOS (Disk Operating System)
  建立在BIOS基础上的PC机操作系统
- 组成：
  - IBMBIO.COM：I/O设备处理程序，完成设备到内存或内存到外设数据传送
    - 例如：DOS调用BIOS显示输出程序完成显示输出，调用BIOS打印输出程序完成打印输出，调用BIOS键盘输入程序完成键盘输入等
  - IBMDOS.COM：作业管理与监控、文件管理程序、设备处理程序
    - 设备管理与监控：通过IBMBIO.COM 形成一个或多个BIOS调用



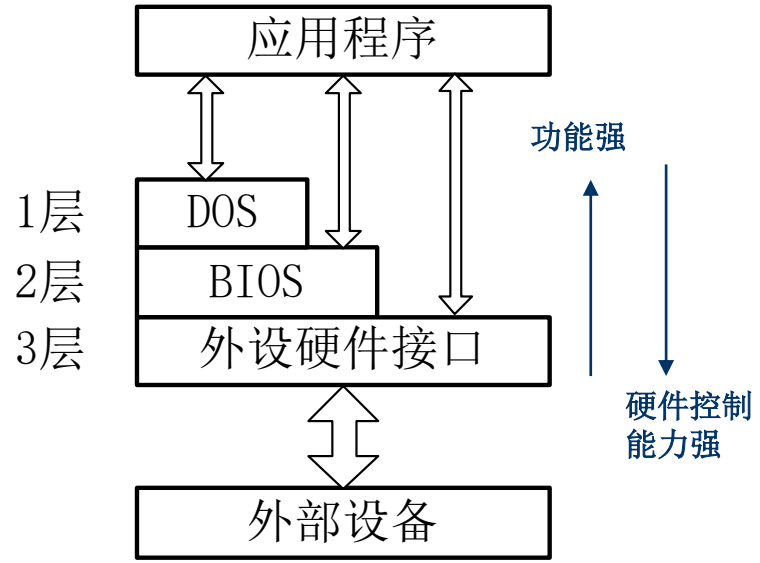
DOS系统功能调用

附录4 P605：DOS系统功能调用 INT 21H

- AH=1 键盘输入并回显
- AH=2 显示字符输出
- AH=9 显示字符串输出
- AH=0AH 键盘输入到缓冲区
- AH=4CH 带返回码终止

应用程序、DOS、BIOS和外设接口之间的关系

- DOS和BIOS都提供某些相同的功能，但它们之间的层次关系是不同的
  - BIOS直接建立在硬件的基础上
  - DOS建立在BIOS的基础上，通过BIOS控制硬件



应用程序、DOS、BIOS和硬件接口间的关系

应用程序、DOS、BIOS和外设接口之间的关系

- DOS、BIOS和硬件接口都为应用程序提供完成输入输出的功能，而且随着层次的加深访问外部设备的能力越强
- 从应用程序的编写角度出发，随着层次的加深，应用程序的编写难度和复杂度大大增加

应用程序、DOS、BIOS和外设接口之间的关系

- 编写应用程序推荐调用I/O的顺序如下：

  DOS--> BIOS--> 直接外设接口访问

  - 通常I/O操作应该首先选择调用DOS提供的系统功能，完成输入输出，这样实现容易，而且对硬件的依赖性最少，程序可移植性好
    - 通过BIOS,DOS屏蔽了底层硬件的差异
  - 如果DOS不提供某种服务或者不能使用DOS的场合可考虑BIOS调用
  - 应用程序也可以直接操纵外设接口来控制外设，从而获得最高效率，但编程复杂性最高

DOS和BIOS功能调用方法

- 采用软件中断的方式实现功能调用
- 应用程序在进行中断调用时应明确
  1. 中断类型号
  2. 功能号：功能号→AH；子功能号→AL
  3. 入口参数：通过寄存器提供专门的调用参数
- 应用程序调用时步骤如下：

  1、将调用所需的入口参数装入指定的寄存器

  2、如需功能号：AH←功能号

  3、如需子功能号：AL←子功能号

  4、按中断号调用DOS或BIOS中断：INT n

  5、检查返回参数是否正确等

- 为什么DOS和BIOS功能调用使用中断方式实现调用？（站在系统设计员的角度）
  - **系统设计简单、高效：** BIOS的功能调用也是调用BIOS中的基本硬件和异常等中断处理程序。中断调用方式与硬件中断处理程序转入方式统一，系统设计方便
    - 不必设置中断入口和子程序调用两种程序等
  - **用户编程方便、可移植性好：** DOS的功能调用实质上是调用DOS中的扩展硬件等中断处理程序，中断调用方式与中断处理程序转入方式统一，使用透明，应用程序编写独立、方便
    - 应用程序不必关心低层功能处理程序入口，现场保存情况等
    - 程序编写独立，可移植性好

    BIOS：机器不同，硬件配置不同，BIOS具体程序实现等不同
    DOS：功能子程序每次装入位置不同；类型/版本不同实现不同
  - 最主要的原因是低层BIOS和DOS的修改（处理程序的入口地址改变）对上层应用透明，应用程序不变，可移植性好

课内测试CH09-1

1. 请在填空中 [填空1] 填写“161”（10分）；

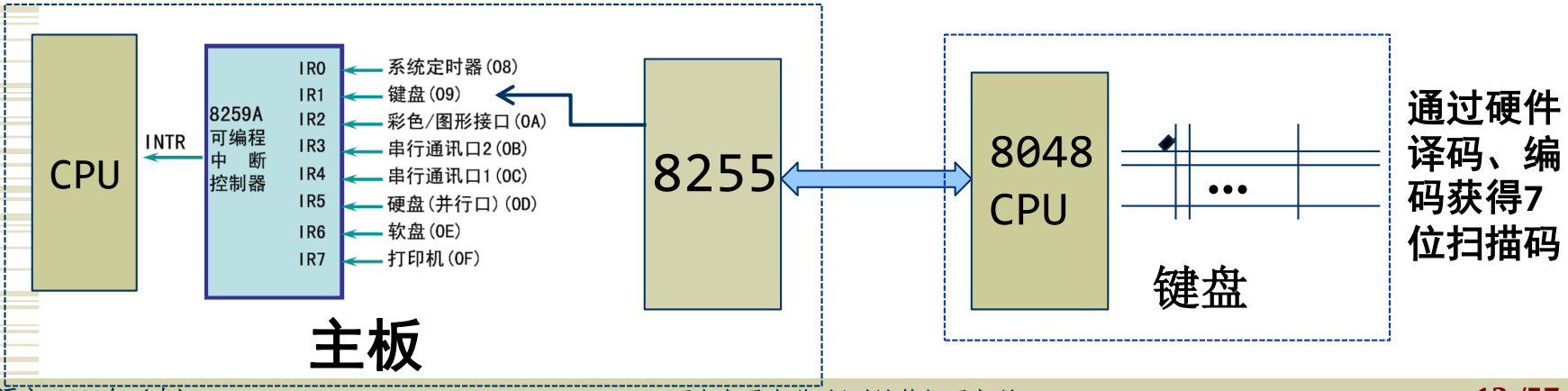
2. 请在填空中 [填空2] 填写：（10分）

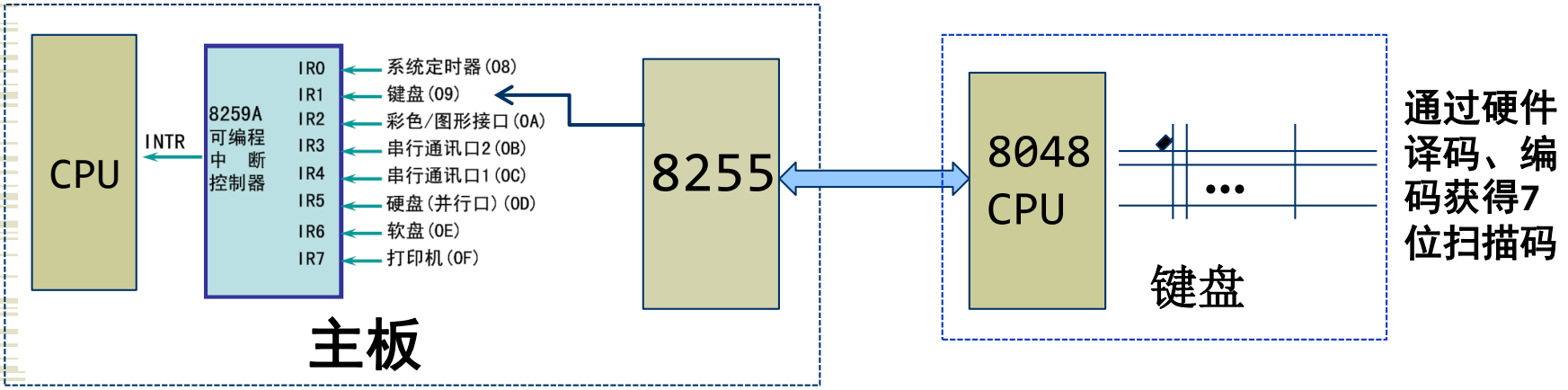
   BIOS

9.1 键盘I/O

- **主板上：**
  - **键盘接口芯片：** 8255可编程序外围接口芯片
  - **键盘中断允许：** 8259中屏蔽寄存器（21H端口）第1位=0
- **键盘上：**
  - **按 键**16*8矩阵排列
  - **由**Intel 8048**单片机控制对键盘扫描**

7	6	5	4	3	2	1	0
打印机	软盘	硬盘	COM1	COM2	彩显	键盘	定时器





- 键盘**硬件中断(09H)**：如果有键按下，且中断也允许
  - 向CPU发中断，由BIOS中的键盘外设中断程序处理，转换成字符码，并将字符码和扫描码存储在**内存缓冲区中**
- 键盘**软件中断**：DOS(21H)和BIOS(16H)软件中断调用
  - INT 21H, INT 16H
  - 应用程序可以使用DOS和BIOS软件中断调用获得存储在**内存缓冲区中**的字符码和扫描码
- 键盘**软件中断**和键盘**硬件中断**在BIOS中由不同的中断处理程序完成，功能不同

9.1.1 字符码与扫描码

- 从键盘输入端口60H读取一个字节，其低7位是扫描码（01H-53H，见表9.3）
- 每个键的扫描码有两个：

  通码（键按下，0）、断码（键放开，1）
- BIOS键盘中断处理程序将扫描码转换为字符码
  - ASCII码，或0（非ASCII码键，如功能键）
- 转换成的字符码和扫描码存储在内存的键盘缓冲区KB_BUFFER中

```
0040:0001A   BUFF_HEAD DW ?
0040:0001C   BUFF_TAIL DW ?
0040:0001E   KEY_BUFFER DW 16 DUP(?)
0040:0003E   KEY_BUFFER_END LABEL WORD
```

缓冲区是先进先出的循环队列

9.1.2 BIOS键盘中断

表9.4 BIOS键盘中断（INT 16H）

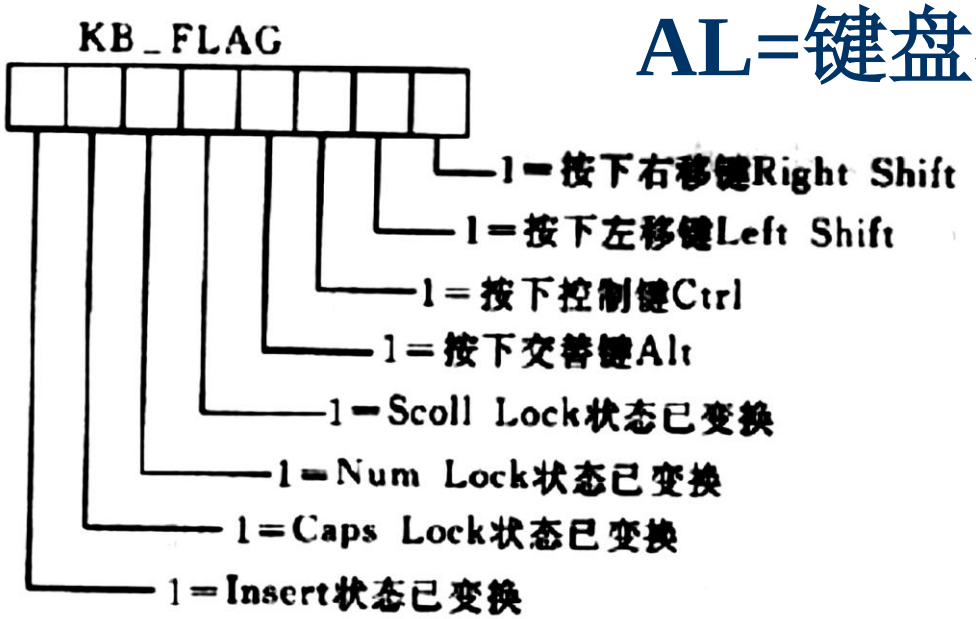
AH	功能	返回参数
0	从键盘读一字符	AL=字符码 AH=扫描码
1	检测键盘缓冲区	如ZF=0 AL=字符码 AH=扫描码 如ZF=1，缓冲区空
2	取键盘状态字	AL=键盘状态字

这里读键盘字符是从内存的键盘缓冲区中读取

如何判断不具有ASCII码的功能键动与否？

- 用INT 16H, AH=2 读键盘状态字

AL=键盘状态字



这里读键盘字符是从内存的键盘缓冲区中读取

9.1.3 DOS键盘功能调用

功能更强大
使用更方便

使用功能1时，如果按下Ctrl_C或Ctrl_Break，DOS在返回前直接结束程序

AH	功 能	调用参数	返回参数
1	从键盘输入一个字符并回显在屏幕上		AL=字符
6	读键盘字符	DL=0FFH	若有字符可取， AL=字符 ZF=0 若无字符可取， AL=0Z F=1
7	从键盘输入一个字符，不回显		AL=字
8	从键盘输入一个字符，不回显，检测Ctrl_Break		AL=字符
A	输入字符到缓冲区		
B	读键盘状态		
C	清除键盘缓冲区，并调用一种键盘功能	DS：DX=缓冲区首址 AL=键盘功能号 （1，6，7，8或A）	AL=0FFH有键入 AL=00无键入

- **注意：** 调用DOS或BIOS从内存的键盘缓冲区中读取键盘按键字符后，相应的键盘字符就会从内存的键盘缓冲区中清除掉。

例1 从键盘读一个字符

```
; BIOS调用完成从键盘读一个字符
    MOV   AH, 0
    INT   16H          ; 字符码 →AL, 扫描码 →AH
```

表9.4 BIOS键盘中断（INT 16H）

AH	功能	返回参数
0	从键盘读一字符	AL=字符码 AH=扫描码

```
; DOS调用完成从键盘读一个字符
    MOV   AH, 7
    INT   21H          ; 字符码 →AL
```

表 9.5 DOS 键盘操作（INT 21H）

AH	功能	调用参数	返回参数
7	从键盘输入一个字符，不回显		AL=字符码

例2．先清除键盘缓冲区，然后再从键盘读一个字符

表 9.4　BIOS 键盘中断（INT 16H）

AH	功能	返回参数
0	从键盘读一字符	AL=字符码 AH=扫描码
1	读键盘缓冲区的字符	如ZF=0 AL=字符码 AH=扫描码 如 ZF=1，缓冲区空
2	取键盘状态字	AL=键盘状态字

```
；BIOS调用
        ┇
REPT：  MOV    AH，1
        INT    16H          ；判缓冲区空？
        JZ     SKIP         ；为空，转移
        MOV    AH，0
        INT    16H          ；从键盘缓冲区取一个字符码
        JMP    REPT         ；继续判断
SKIP：  MOV    AH，0        ；等待键盘输入新的字符码
        INT    16H
```

；DOS调用

```
        ┆
REPT:
        MOV     AH，0BH         ；参看P605
        INT     21H             ；判有无输入？
        CMP     AL，0
        JZ      SKIP            ；为空，转移
        MOV     AH，7
        INT     21H             ；从键盘缓冲区取走一个字符
        JMP     REPT            ；继续判断
SKIP:
        MOV     AH，7           ；等待键盘输入有效字符
        INT     21H
        ┆
```

或者如下调用：

```
        MOV     AL，7
        MOV     AH，0CH
        INT     21H
```

表 9.5　DOS 键盘操作（INT 21H）

AH	功能	调用参数	返回参数
1	从键盘输入一个字符，并回显		AL=字符码
6	读键盘字符	DL=0FFH	若有字符可取： AL=字符码 ZF=0 若有字符可取： AL=0 ZF=1
7	从键盘输入一个字符，不回显		AL=字符码
8	从键盘输入一个字符，不回显，检测 Ctrl_Break		AL=字符码
A	输入字符到缓冲区	DS:DX=缓冲区首址	参见图 9.3
B	读键盘状态		AL=0FFH 有键入 AL=00 无键入
C	清除键盘缓冲区，并调用一种键盘功能	AL=键盘功能号（1，6，7，8，A）	

例3．显示按键，在检测到所按下的CTRL键后结束运行

- 调用16H号中断的2号功能
  - 取得键盘状态字节
  - 判断是否按下了CTRL键
    - 若按下CTRL键则结束运行；
    - 否则显示按键
- 汇编语言源程序如下：

```
sseg    segment stack           ; 建立堆栈段
        dw      200 dup(?)
tos     label   word
sseg    ends
ctrl=00000100b          ; 定义变换键ctrl判断字
cseg    segment                 ; 建立代码段
        assume cs:cseg, ss:sseg
begin   proc    far
        mov     sp, offset  tos  ; 初始化堆栈指针
        mov     ax, sseg
        mov     ss, ax
        push    ds   ; 压入返回DOS地址
        mov     ax, 0
        push    ax
start:  mov     ah, 2           ; 取 KB_FLAG
        int     16h
        test    al, ctrl        ; 判断是ctrl键吗?
        jnz     finish          ; 是ctrl键，结束运行
         mov    ah, 1
        int     16h
        jz      start           ; 缓冲区空，无键可读，转移到start
        mov     dl, al          ; 显示
        mov     ah, 2
        int     21h
       mov      ah,0
        int     16h
        jmp     start           ; 继续下一轮
finish: ret
begin   endp
cseg    ends
        end     begin
```

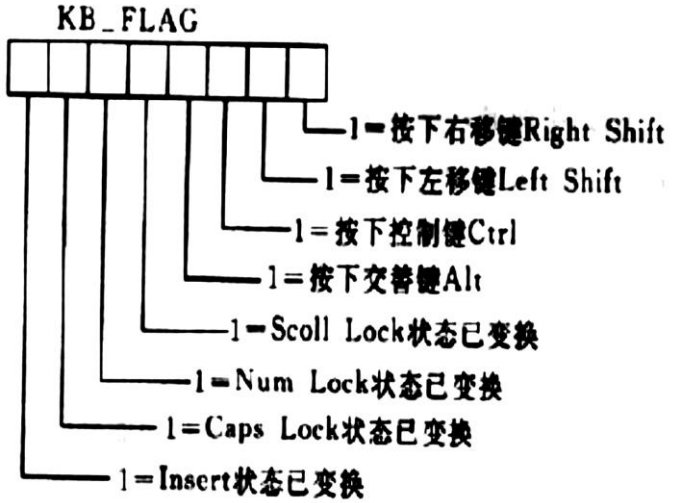


图 9.2 键盘状态字节

表 9.4 BIOS 键盘中断（INT 16H）

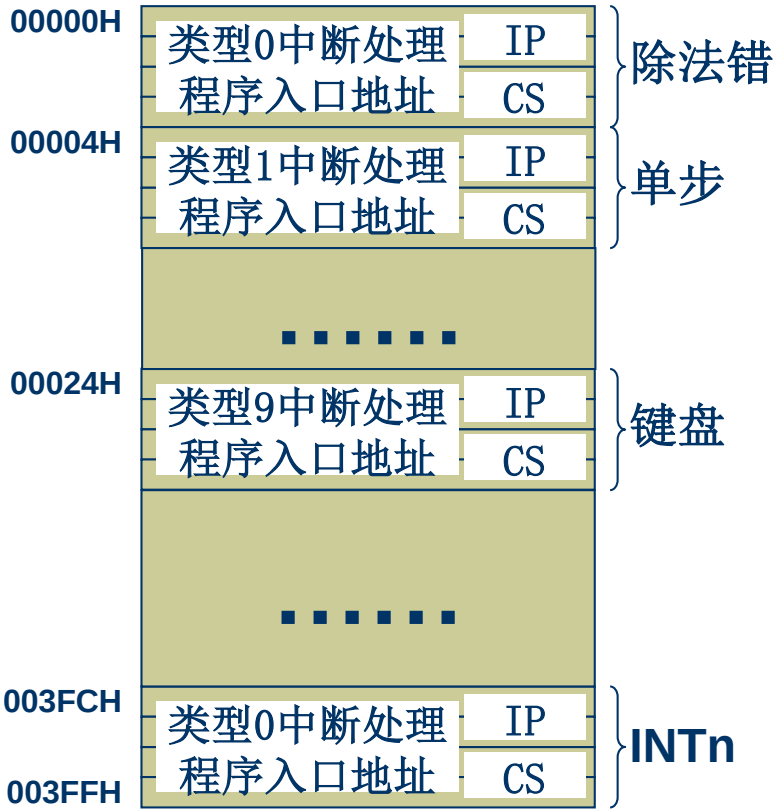
AH	功能	返回参数
0	从键盘读一字符	AL=字符码 AH=扫描码
1	读键盘缓冲区的字符	如ZF=0 AL=字符码 AH=扫描码 如 ZF=1，缓冲区空
2	取键盘状态字	AL=键盘状态字

BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

如何从键盘直接读扫描码？

1. 查询方式：
   - 给8259的21H端口送控制命令，关掉键盘中断允许位
   - 通过键盘接口，编写查询程序
2. 中断方式：
   - 主程序中，修改向量表中键盘硬件中断向量指向自己的中断处理程序
   - 编写中断程序，读键盘扫描码，按自己需要进行处理后放到一个自定义内存缓冲区
   - 主程序中，从自定义内存缓冲区读键盘相关数据进行处理



注意关/开中断、中断向量保存/恢复等要适时、正确

想读读自己机器的BIOS键盘软中断程序吗？（BIOS键盘中断INT 16H）

1、计算BIOS键盘中断向量地址

16H*4=0058H

2、看中断向量表，获得中断程序入口

```
-d 0000:0058
0000:0050                               C4 09 10 02 8B 05 10 02
```

IP　　CS

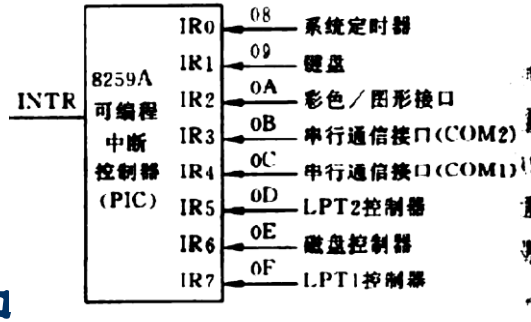
3、反汇编，读程序

```
-u 0210:09c4
0210:09C4 1E              PUSH    DS
0210:09C5 53              PUSH    BX
0210:09C6 BB4000          MOV     BX,0040
0210:09C9 8EDB            MOV     DS,BX
0210:09CB 80FC10          CMP     AH,10
0210:09CE E8E8FD          CALL    07B9
0210:09D1 7203            JB      09D6
0210:09D3 E9E000          JMP     0AB6
0210:09D6 0AE4            OR      AH,AH
0210:09D8 743E            JZ      0A18
0210:09DA FECC            DEC     AH
0210:09DC 7474            JZ      0A52
0210:09DE FECC            DEC     AH
0210:09E0 7411            JZ      09F3
0210:09E2 FECC            DEC     AH
```

内存的键盘缓冲区KB_BUFFER：

```
0040:001A  BUFF_HEAD DW ?
0040:001C  BUFF_TAIL DW ?
0040:001E  KEY_BUFFER DW 16 DUP(?)
0040:003E  KEY_BUFFER_END LABEL WORD
```

如何读DOS或BIOS中的中断程序？
（进一步学习汇编和微机接口原理的方法）



找相关文档，或者按如下方式：

1. Debug中寻找中断程序入口地址
   - 如BIOS中的键盘硬件中断程序入口
     - Debug中显示中断向量（键盘中断向量地址09H*4=0024H）

```
-d 0000:0024
0000:0020                   0A 04 0E 02-3A 00 98 03 54 00 98 03     ....:...T...
0000:0030  6E 00 98 03 8A 00 98 03-A2 00 98 03 FF 03 0E 02   n...............
0000:0040  A9 00 0E 02 A4 07 0E 02-AA 09 0E 02 5D 04 0E 02   ............]...
0000:0050  B0 07 0E 02 0D 02 DD 02-C4 09 0E 02 0B 05 0E 02   ................
0000:0060  0E 0C 0E 02 14 0C 0E 02-1F 0C 0E 02 AD 06 0E 02   ................
0000:0070  AD 06 0E 02 A4 F0 00 F0-37 05 0E 02 F1 0F 00 C0   ........7.......
0000:0080  72 10 A7 00 7C 10 A7 00-4F 03 FF 0D 8A 03 FF 0D   r...|...O.......
0000:0090  17 03 FF 0D 86 10 A7 00-90 10 A7 00 9A 10 A7 00   ................
0000:00A0  B8 10 A7 00                                       ....
-
```

2. Debug中反汇编中断程序

```
-u 020e:040a
020E:040A 50              PUSH    AX
020E:040B 33C0            XOR     AX,AX
020E:040D C4C4            LES     AX,SP
020E:040F 0958E9          OR      [BX+SI-17],BX
020E:0412 3802            CMP     [BP+SI],AL
020E:0414 90              NOP
020E:0415 90              NOP
020E:0416 E460            IN      AL,60
020E:0418 90              NOP
020E:0419 90              NOP
020E:041A C4C4            LES     AX,SP
```

9.2 显示I/O

9.2.1 显示器简介

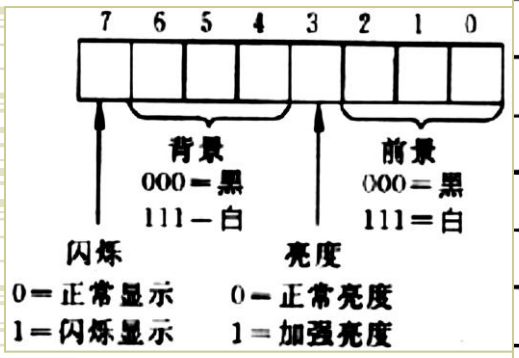
(1) 显示属性

- 屏幕上显示的字符取决于字符码及字符显示属性

(2) 显示缓冲区

- 显示适配卡带有显示存储器，用于存放屏幕上显示文本的代码及属性或图形信息
- 显示存储器作为系统存储器的一部分，可用访问普通内存的方法访问显示存储器
  - 显示屏幕是“存储器映像”
- 直接存取显示存储器内容进行显示的方法称为直接写屏

- 在单色显示时，字符显示属性定义了闪烁、反向和高亮度等显示特性



属性值（二进制）	属性值（十六进制）	显示效果
00000000	00	无显示
00000001	01	黑底白字，下划线
00000111	07	黑底白字，正常显示
00001111	0F	黑底白字，高亮度
01110000	70	白底黑字，反相显示
10000111	87	黑底白字，闪烁
11110000	F0	白底黑字，反相闪烁

- 在彩色显示时，字符显示属性定义了前景色和背景色

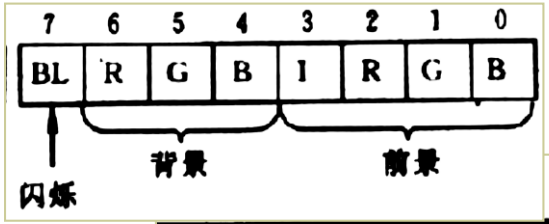


表 9.7 16种颜色的组合

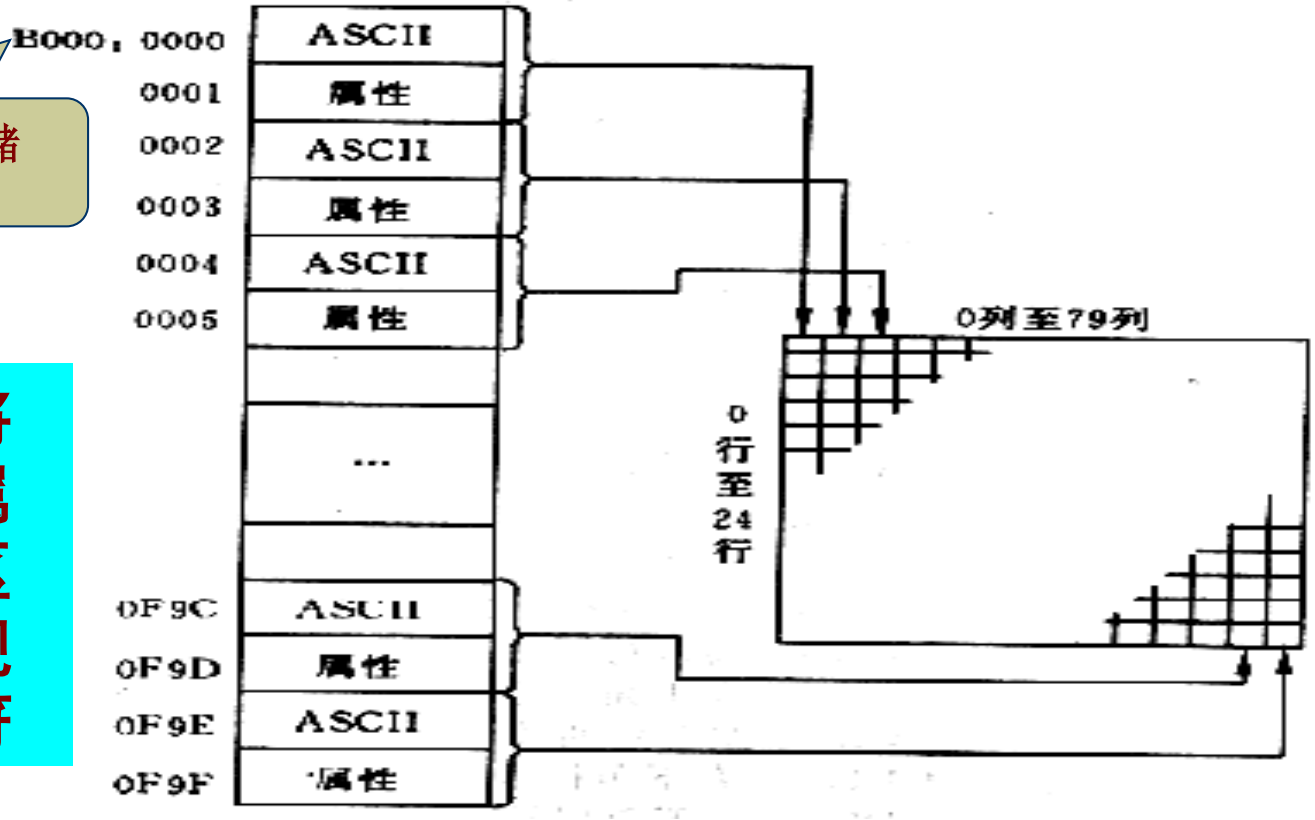
颜色	IRGB	颜色	IRGB	颜色	IRGB	颜色	IRGB
黑	0000	灰	1000	红	0100	浅红	1100
蓝	0001	浅蓝	1001	品红	0101	浅品红	1101
绿	0010	浅绿	1010	棕	0110	黄	1110
青	0011	浅青	1011	灰白	0111	白	1111

每个显示字符占2个存储单元

- 25*80的单色显示文本方式下
  - 屏幕可有2000个字符位置，显存容量需要4000B≈4KB
  - 如果显存有16KB容量，可保存4屏幕的字符数据，即4页数据
- 可根据显示位置的行列值算出显示存储区的地址
- 屏幕上某一字符在显存中的偏移地址

Char_offset=page_offset+((row*width)+column)*byte

注意：各种适配器的显示存储器（显存）的启始地址不同



可以用指令直接将要显示的字符和属性写入显示存储区的相应单元，实现在屏幕上显示字符

9.2.2 BIOS显示中断

- 显示器BIOS中断调用(INT 10H)

  显示器BIOS程序提供功能如下：

  设置显示模式

  设置光标类型、设置光标位置、读光标位置

  读取光标位置处的字符和属性

  将字符和属性写到光标位置处

  选择当前显示页

  向上滚屏、向下滚屏

AH	功 能	调用参数	返回参数 / 注释
1	置光标类型	(CH)$_{3\sim5}$ = 光标开始行 (CL)$_{0\sim3}$ = 光标结束行	
2	置光标位置	BH = 页号 DH = 行 DL = 列	
3	读光标位置	BH = 页号	CH = 光标开始行 CL = 光标结束行 DH = 行 DL = 列
4	置显示页	AL = 显示页号	
5	选择活动显示页		
6	屏幕初始化或上卷	AL = 上卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号	
7	屏幕初始化或下卷	AL = 下卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号	
8	读光标位置的属性和字符	BH = 显示页	AH = 属性 AL = 字符
9	在光标位置显示字符及其属性	BH = 显示页 AL = 字符 BL = 属性 CX = 字符重复次数	
A	在光标位置只显示字符	BH = 显示页 AL = 字符 CX = 字符重复次数	
E	显示字符(光标前移)	AL = 字符 BL = 前景色	光标跟随字符移动
13	显示字符串	ES:BP = 串地址 CX = 串长度 DH, DL = 起始行列 BH = 页号 AL = 0, BL = 属性 串:Char,char,.....,char	光标返回起始位置
		AL = 1, BL = 属性 串:Char,char,.....,char	光标跟随移动
		AL = 2 串:Char,attr,.....,char,attr	光标返回起始位置
		AL = 3 串:Char,attr,.....,char,attr	光标跟随串移动

对某个窗口操作

INT 10H

实质是对显示存储区的操作

INT 10H (ah=01h)

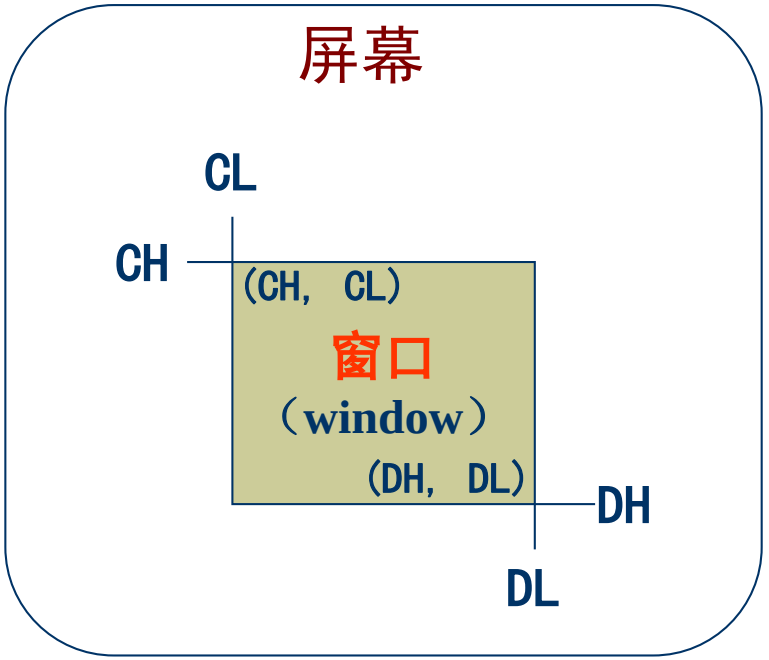
- AH = 01h CH = Scan Row Start, CL = Scan Row End
- Normally a character cell has 8 scan lines, 0-7. So, CX=0607h is a normal underline cursor, CX=0007h is a full-block cursor. If bit 5 of CH is set, that often means "Hide cursor". So CX=2607h is an invisible cursor.
- Some video cards have 16 scan lines, 00h-0Fh.
- Some video cards don't use bit 5 of CH. With these, make Start>End (e.g. CX=0706h)

例如： 窗口初始化或字符上卷

6	屏幕初始化或上卷	AL = 上卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号
7	屏幕初始化或下卷	AL = 下卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号

行向上移位

- AL>0时，要上卷的行数
  - 窗口上卷时，低端的行由空行代替，属性由BH决定
  - 上卷时移出窗口的行不能恢复
- AL=0时， 全窗口为空白
- BH=空白区域的视频属性
- CH, CL=窗口左上角的行列位置
- DH, DL=窗口右下角的行列位置



显示器BIOS中断应用举例

例1、 在当前光标位置处显示5个字符U (UUUUU)，但不移动光标

- 9号子功能调用

9	在光标位置显示 字符及其属性	BH = 显示页 AL = 字符 BL = 属性 CX = 字符重复次数

- 指令序列如下：

```
MOV     BH, 0          ; 显示页号，第0页
MOV     AL, 'U'        ; 显示字符的代码
MOV     BL, 4EH        ; 显示字符的属性，红底黄字
MOV     CX, 5          ; 字符重复次数
MOV     AH, 9          ; 显示I/O中断程序的功能号
INT     10H            ; 中断调用指令
```

例2、 清屏并把光标设置在左上角

- 在窗口滚屏时，如果滚屏行数为0，就表示清除整个窗口。设屏幕为25*80，先清除屏幕，然后把光标设定到左上角

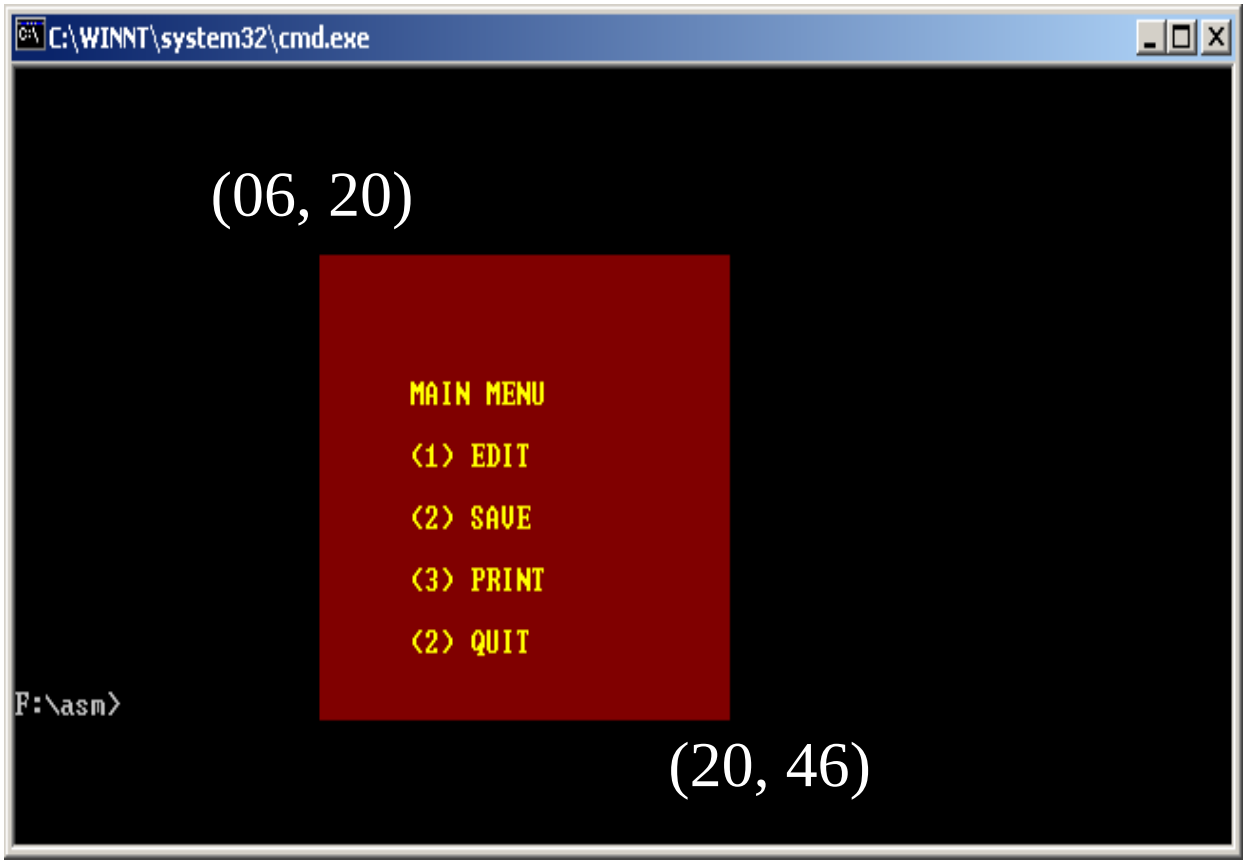
```
;清屏
        mov ah, 6
        mov al, 0
        mov bh, 7 ; 黑低白字
        mov cx, 0
        mov dh, 24
        mov dl, 79
        int 10h
;光标设置在左上角
        mov   dx,   0
        mov   ah,   2
        int 10h
```

6	屏幕初始化或上卷	AL = 上卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号

2	置光标位置	BH = 页号 DH = 行 DL = 列

例3、 编制一程序。要求：在干净的屏幕上开一窗口（红底黄字），左上角坐标（06H，14H），右下角坐标（14H，2EH）；在窗口内显示：

MAIN MENU
Edit
Save
Print
Quit

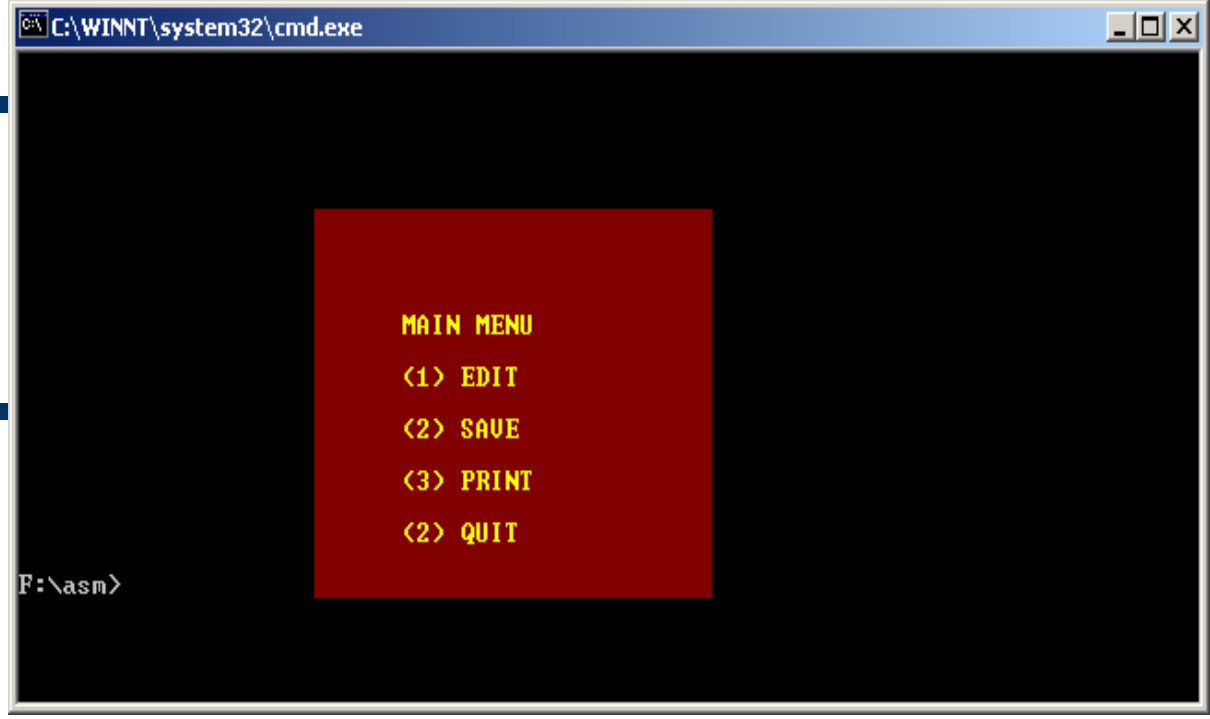


程序清单

```
; 开设空白窗口的宏定义
clrscr  macro  lu, rd, fb    ; lu=左上角坐标，rd=右下角坐标
                             ; fb=前景色和背景色
        mov    ax, 0600h
        mov    cx, lu
        mov    dx, rd
        mov    bh, fb
        int    10h
        endm

; 设置光标的宏定义
cursor  macro  row, col      ; row=行号，col=列号
        mov    bh, 0
        mov    dh, row
        mov    dl, col
        mov    ah, 2
        int    10h
        endm
```

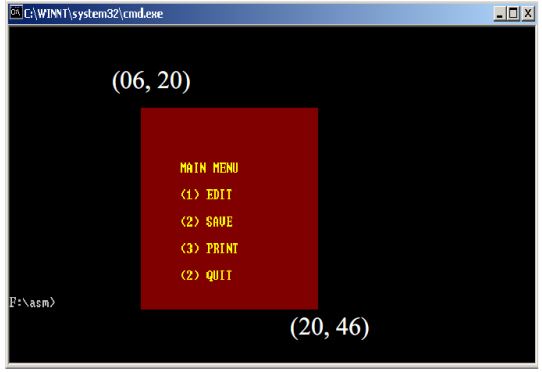
6	屏幕初始化或上卷	AL = 上卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号

2	置光标位置	BH = 页号 DH = 行 DL = 列




```
; 定义数据段
dseg    segment
; 每个字符串以’$’结束利于判断
d1      db      'MAIN MENU', 0dh, 0ah, '$'
        db      '(1) Edit',0dh,0ah,'$'
        db      '(2) Save',0dh,0ah,'$'
        db      '(3) Print',0dh,0ah,'$'
        db      '(4) Quit',0dh,0ah,'$'
dseg    ends
```

```
;定义代码段
cseg    segment
        assume  cs:cseg,ds:dseg
start   proc    far
        mov     ax, dseg
        mov     ds, ax
        clrscr  0, 184fh, 07h          ; 清屏，黑底白字
        clrscr  0614h, 142eh, 4Eh      ; 开窗口，红底黄字
        cursor  10,26                  ; 菜单首行光标
        mov     si, offset d1
        call    dischs                 ; 显示菜单首行
        cursor  12, 26                 ; 菜单第二行光标
        call    dischs                 ; 显示菜单第二行
        cursor  14, 26
        call    dischs
        cursor  16, 26
        call    dischs
        cursor  18, 26
        call    dischs
        mov     ah, 4ch
        int     21h
start   endp
; 以‘$’结束的字符串显示的子程序
```




```
d1      db      'MAIN MENU', 0dh, 0ah, '$'
        db      '(1) Edit',0dh,0ah,'$'
        db      '(2) Save',0dh,0ah,'$'
        db      '(3) Print',0dh,0ah,'$'
        db      '(4) Quit',0dh,0ah,'$'
dseg    ends
```

```
dischs  proc    near
disch:  mov     al, [si]
        inc     si
        cmp     al, '$'
        je      exit
        mov     bx, 13h
        mov     ah, 0eh
        int     10h
        jmp     disch
exit:   ret
dischs  endp

cseg    ends
        end     start
```

E	显示字符（光标前移）	AL = 字符 BL = 前景色	光标跟随字符移动

9.2.3 DOS显示功能调用

- DOS显示功能调用中断为 INT21H
- INT 21H显示操作

AH	功 能	调 用 参 数
2	显示一个字符（检查Ctrl-Break）	DL=字符， 光标跟随字符移动
6	显示一个字符（不检查Ctrl-Break）	DL=字符， 光标跟随字符移动
9	显示字符串	DS:DX=串地址， 串必须以 $ 结束，光标跟随串移动

- 不能有控制码的ASCII码
- 检查Ctrl-Break:
- 显示字符串中：串必须以$结束；如果希望光标能自动换行，在字符串结束符$前 加回车和换行的ASCII码

AH	功 能	调 用 参 数
9	显示字符串	DS:DX=串地址 串必须以$结束，光标跟随串移动

例1、显示一串字符

```
; 字符串的数据定义
    CR        EQU  0DH
    LF        EQU  0AH
    TAB       EQU  09H
    MESSAGE   DB   TAB, 'The sort operation is finished.'
              DB   CR, LF, '$'

; 显示字符串的指令
    MOV AH, 09H
    MOV DX, SEG MESSAGE
    MOV DS, DX
    MOV DX, OFFSET MESSAGE
    INT 21H
```

9.3 打印机

- 打印机I/O中断
  - DOS中断调用　INT 21H
  - BIOS中断调用　INT 17H

表 9.11　打印机 I/O 中断

INT	AH	功　能	调用参数	返回参数
21H	5	打印一个字符	DL = 字符	
17H	0	打印一个字符	AL = 字符	AH = 状态字节
		并回送状态字节	DX = 打印机号	
17H	1	初始化打印机	DX = 打印机号	AH = 状态字节
		回送状态字节		
17H	2	回送状态字节	DX = 打印机号	AH = 状态字节

- **主机输出给打印机的信息包括两类：**
  - **字符码（ASCII）**
  - **完成一定特定动作的控制码或功能码**

表9.12 打印机常用的标准控制字符

功能码	十进制	十六进制（ASCII 码）	功能含义
SP	08	08	空格
HT	09	09	水平制表（Tab）
LF	10	0A	换行
VT	11	0B	垂直制表（Tab）
FF	12	0C	换页
CR	13	0D	回车

- **水平制表：仅当打印机有此功能，并被置成打印机Tab状态时才能实现**
  否则，不执行此命令，或打印多个空格代替Tab
  Tab相当于8个字符宽度

许多打印机不认识TAB字符(09H)，这时程序就要检查TAB字符，若输出的字符是TAB，就要插入空格，把当前光标位置移到8，16，24，...字符位置上

INT	AH	功　能	调用参数	返回参数
21H	5	打印一个字符	DL = 字符	

9.3.1 dos打印功能

- 打印一个字符：INT 21H，AH=5，DL=字符
- 如果需要回车、换行等打印机控制功能，必须由汇编程序送出回车、换行等控制码给DOS

```
TEXT    DB      0CH, 'Hello, everybody!', 0DH, 0AH, 0AH
COUNT   EQU     $-TEXT
......
        MOV     CX,   COUNT
        MOV     BX,   0
NEXT:   MOV     AH,   5
        MOV     DL,   TEXT[BX]
        INT     21H
        INC     BX
        LOOP    NEXT
```

0CH：换页
0DH：回车
0AH：换行

DOS打印机操作自动测试打印机状态

9.3.2 BIOS打印功能

- BIOS提供的打印I/O程序中断号为17H
- 系统可连接多台打印机，用打印机号选择打印机，打印机号为0，1，2

INT	AH	功 能	调用参数	返回参数
17H	0	打印一个字符	AL = 字符	AH = 状态字节
		并回送状态字节	DX = 打印机号	
17H	1	初始化打印机	DX = 打印机号	AH = 状态字节
		回送状态字节		
17H	2	回送状态字节	DX = 打印机号	AH = 状态字节

INT	AH	功　能	调用参数	返回参数
17H	0	打印一个字符 并回送状态字节	AL = 字符 DX = 打印机号	AH = 状态字节

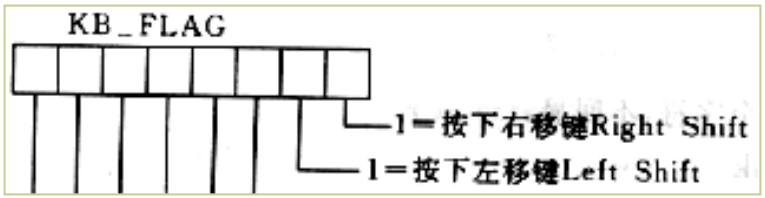
例1、将AL中的字符输出到打印机

```
MOV    AH，0
MOV    AL，CHAR
MOV    DX，0
INT    17H
```

例2、将键盘接收到的字符显示在显示器上，并由打印机输出，当键盘上按下shift键即退出程序。

```
; 定义堆栈段
sseg        segment stack
            dw  100 dup(?)
tos         label   word
sseg        ends

; 定义代码段
cseg        segment
            assume cs:cseg, ss:sseg
key_prt     proc        far
            mov         ax, sseg
            mov         ss, ax
            mov         sp, offset  tos   ; 设置堆栈指针
            call        cls               ; 调用清屏子程序
            mov         ah, 2
            mov         dl, 0ah
            int         21h               ; 光标指向下一行
key_ch:
            mov         ah,2
            int         16h               ; 取变换键状态
            test        al,03h            ; 有shift按下吗？字节的低两位为 l_shift + r_shift的状态
            jnz     endprog               ; 有shift键按下则转移到程序结束
```

AH	功 能	调 用 参 数
2	显示一个字符（检查 Ctrl-Break）	DL=字符， 光标跟随字符移动



2	取键盘状态字节	AL＝键盘状态字节

```
        mov    ah,1
        int    16h          ；判断键盘有键可读吗？
        jz     key_ch       ；无键可读，转去读键盘状态
        mov    ah,0
        int    16h          ；读取键盘字符
        push   ax
        mov    dl,al
        mov    ah,2         ；显示所读的字符
        int    21h
        pop    ax
        push   ax
        cmp    al,0dh       ；是回车，加上换行符
        jne    nnn
        mov    dl,0ah       ；换行符
        mov    ah,2
        int    21h
nnn:                        ；打印键盘键入的字符
        mov    ah,0
        mov    dx,0
        int    17h          ；打印al中的字符
        pop    ax
        cmp    al,0dh       ；是回车，加上换行符
        jne    key_ch
        mov    al,0ah
        mov    ah,0
        mov    dx,0
        int    17h
        jmp    key_ch
endprog:                    ；返回dos
        mov    ah,4ch
        int    21h
key_prt endp
```

AH	功 能	返回参数
0	从键盘读一字符	AL＝字符码 AH＝扫描码
1	读键盘缓冲区的字符	如 ZF＝0 AL＝字符码 AH＝扫描码 如 ZF＝1，缓冲区空

AH	功 能	调 用 参 数
2	显示一个字符（检查 Ctrl-Break）	DL=字符， 光标跟随字符移动

17H	0	打印一个字符 并回送状态字节	AL = 字符 DX = 打印机号	AH = 状态字节

```
;清屏子程序
cls     proc          near
            mov     ax,0600h
            mov     cx,0
            mov     dx,184fh
            mov     bh,36h
            int     10h
            ret
cls         endp

cseg        ends
            end     key_prt
```

6	屏幕初始化或上卷	AL = 上卷行数 AL = 0 全屏幕为空白 BH = 卷入行属性 CH = 左上角行号 CL = 左上角列号 DH = 右下角行号 DL = 右下角列号

课内测试CH09-2

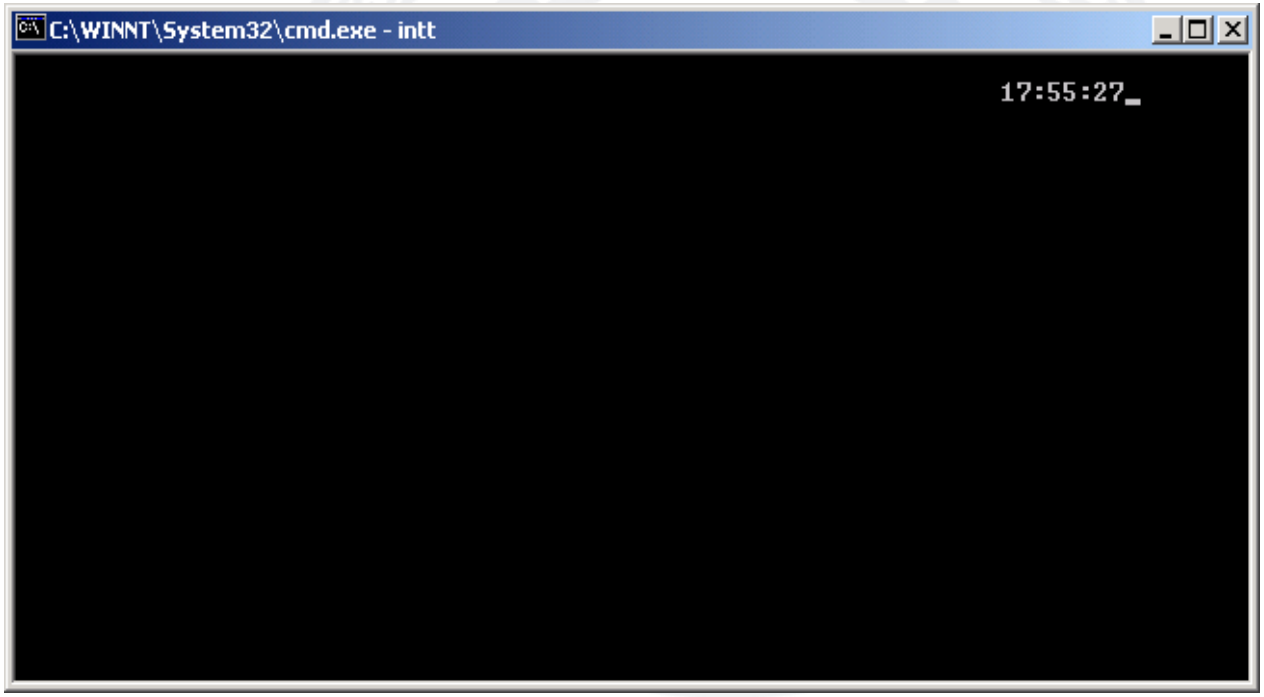
1. 请在填空中 [填空1] 填写“162”（10分）；

2. 请在填空中 [填空2] 填写：（10分）

   DOS

9.4 BIOS中断举例

例1、 显示实时钟，遇到按键时退出



- 系统加电期间，系统定时器初始化为每隔55毫秒发一次中断请求，每秒要调用约18.2次系统时钟中断处理程序
- 参看“表9.1BIOS中断类型---8259中断类型”
  - CPU在响应定时中断请求后转入08H号中断处理程序
  - BIOS提供的08H号中断处理程序中，有一条软中断指令“INT 1CH”，而BIOS的1CH号中断处理程序处只有一条中断返回指令（IRET），实际上并没有作任何工作
  - 这样安排的目的是为应用程序留下一个软接口，应用程序只要修改中断向量表中1CH号中断向量，转向新的中断处理程序，就能实现某些周期性的工作

- 新的1CH号中断处理程序功能如下：
  - 清屏，利用1AH号中断处理程序的2号功能获取当前时间；
  - 在屏幕的右上角显示当前时间；
  - 记录调用该中断处理程序的次数，当计数满18次后(1秒钟到)，更新显示时间
- 主程序的功能：
  - 保存原1CH号的中断向量；
  - 设置新的1CH号的中断向量；
  - 在主程序完成其他工作后，恢复原1CH号的中断向量

```
; 定义代码段
cseg    segment
        assume cs:cseg, ds:cseg
start   proc    far
        push    cs
        pop     ds
        mov     ax, 351ch
        int     21h                     ; 获取原1CH号的中断向量→ES:BX
        mov     ds:word ptr old1c, bx
        mov     ds:word ptr old1c+2, es         ; 保存原1CH号中断向量
        mov     dx, offset int1c        ; 设置新1CH号中断向量
        mov     ax, 251ch
        int     21h
; 此后，每55毫秒就进入一次新的1CH号的中断处理程序
waitn:  mov     ah, 1
        int     16h                     ; 查有无键按下
        jz      waitn                   ; 转等待键按下
        mov     ah, 0
        int     16h                     ; 读键盘
        lds     dx, ds:old1c
        mov     ax, 251ch
        int     21h                     ; 恢复原1ch中断向量
        mov     ah, 4ch
        int     21h                     ; 返回dos
start   endp
```

```
;定义数据空间
old1c   dd      ?               ;  保存原中断向量
count   dw      0               ;  调用1ch中断程序的次数
hhh     db      ?,?,':'         ;  保存:  “时”
mmm     db      ?,?,':'         ;  保存:  “分”
sss     db      ?,?,‘$’         ;  保存:  “秒”
```

数据是代码段的一部分，不提倡这种编程方法！！！

```
;定义新的1CH号的中断处理程序
int1c   proc   far
        cmp     count, 0
; 调用次数为“0”时（1秒钟到），显示系统时间
        jz      next
        dec     count   ; 显示次数递减（第一次18-1）
        iret
next:
        mov     count, 18   ; 置计数次数初值
        sti
        push    ds                  ; 保护现场
        push    es
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        mov     ah, 2
        int     1ah                 ; 读实时时钟
        mov     al, ch              ; 时送al
        call    ttasc               ; 转换成ascii码
        mov     word    ptr hhh, ax     ; 保存时
        mov     al, cl              ; 分转换
        call    ttasc
        mov     word    ptr mmm, ax
        mov     al, dh              ; 秒转换
        call    ttasc
        mov     word    ptr sss, ax
        call    cls         ; 清屏
        mov     bh, 0
        mov     dx, 0140h
        mov     ah, 2
        int     10h     ; 设置光标（2，65）
        push    cs
        pop     ds
        mov     dx, offset hhh
        mov     ah, 9
        int     21h         ; 显示实时时钟
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        pop     es
        pop     ds
        iret                ; 中断返回
int1c   endp
```

```
;将AL中的BCD数据转换成ASCII码存入AX中
ttasc   proc
        mov     ah, al
        and     al, 0fh
        shr     ah, 4
        add     ax, 3030h
        xchg    ah, al
        ret
ttasc   endp

;清屏子程序
cls     proc
        mov     ax, 0600h
        mov     cx, 0
        mov     dx, 184fh
        mov     bh, 7
        int     10h
        ret
cls     endp
cseg    ends
        end     start
```

变量定义在程序代码段中

```
; 定义代码段
cseg   segment
       assume cs:cseg
start  proc   far
       push   cs
       pop    ds
       mov    ax, 351ch
       int    21h                    ; 获取原1CH号的中断向量
       mov    cs:word ptr old1c, bx
       mov    cs:word ptr old1c+2, es        ; 保存原1CH号中断向量
       mov    dx, offset int1c       ; 设置新1CH号中断向量
       mov    ax, 251ch
       int    21h
; 此后，每55毫秒就进入一次新的1CH号的中断处理程序
waitn: mov    ah, 1
       int    16h                    ; 查有无键按下
       jz     waitn                  ; 转等待键按下
       mov    ah, 0
       int    16h                    ; 读键盘
       lds    dx, cs:old1c
       mov    ax, 251ch
       int    21h                    ; 恢复原1ch中断向量
       mov    ah, 4ch
       int    21h                    ; 返回dos
start  endp

;定义数据空间
old1c  dd     ?            ; 保存原中断向量
count  dw     0            ; 调用1ch中断程序的次数
hhh    db     ?,?,':'      ; 保存：“时”
mmm    db     ?,?,':'      ; 保存：“分”
sss    db     ?,?,'$'      ; 保存：“秒”
```

```
;定义新的1CH号的中断处理程序
int1c  proc   far
       cmp    count, 0
; 调用次数为“0”时（1秒钟到），显示系统时间
       jz     next
       dec    count   ; 显示次数递减（第一次18-1）
       iret
next:
       mov    count, 18   ; 置显示次数初值
       sti
       push   ds                 ; 保护现场
       push   es
       push   ax
       push   bx
       push   cx
       push   dx
       push   si
       push   di
       mov    ah, 2
       int    1ah                ; 读实时时钟
       mov    al, ch             ; 时送al
       call   ttasc              ; 转换成ascii码
       mov    word   ptr hhh, ax    ; 保存时
       mov    al, cl             ; 分转换
       call   ttasc
       mov    word   ptr mmm, ax
       mov    al, dh             ; 秒转换
       call   ttasc

       mov    word   ptr sss, ax
       call   cls       ; 清屏
       mov    bh, 0
       mov    dx, 0140h
       mov    ah, 2
       int    10h    ; 设置光标（2，65）
       push   cs
       pop    ds
       mov    dx, offset hhh
       mov    ah, 9
       int    21h      ; 显示实时时钟
       pop    di
       pop    si
       pop    dx
       pop    cx
       pop    bx
       pop    ax
       pop    es
       pop    ds
       iret            ; 中断返回
int1c  endp
```

```
;将AL中的BCD数据转换成ASCII码存入AX中
ttasc  proc
       mov    ah, al
       and    al, 0fh
       shr    ah, 4
       add    ax, 3030h
       xchg   ah, al
       ret
ttasc  endp

;清屏子程序
cls    proc
       mov    ax, 0600h
       mov    cx, 0
       mov    dx, 184fh
       mov    bh, 7
       int    10h
       ret
cls    endp
cseg   ends
       end    start
```

谢谢！