# 汇编语言程序设计

电子科技大学·计算机学院

邢建川

E-mail: xingjianchuan@sina.com

# 教学计划

1. 总学时数为32。

2. 教学方式：大型开放式网络授课（MOOC）

3. 最后成绩评定办法：平时作业占65%，期末考试占35%。

4. 教材：《汇编语言程序设计》廖建明主编　清华大学出版社 2009.10

## 5. 参考资料：

《8086/8088宏汇编语言程序设计教程》（第二版）王正智 编著电子工业出版社 2002.3

《微机原理与接口技术》陆鑫等编著 机械工业出版社2005.9

《IBM-PC汇编语言程序设计》（第2版）沈美明等编著 清华大学出版社2001.8

《Win32汇编语言程序设计教程》 严义等编著 机械工业出版社 2005.8

《汇编语言程序设计》 殷肖川 主编 清华大学出版社 2005.1

# 第一章　基础知识

本章主要学习内容：

1.汇编语言的一般概念

2.学习和使用汇编语言的目的

3.进位计数制及其相互转换

4.带符号数的表示

5.字符的表示

6.基本逻辑运算

# §1.1 汇编语言的一般概念

计算机程序设计语言可分为机器语言、高级语言和汇编语言三类。

## 1. 机器语言

机器语言就是把控制计算机的命令和各种数据直接用二进制数码表示的一种程序设计语言。

例如，要实现将寄存器AH的内容与数10相加，结果再送回到寄存器AH中。

用机器语言实现上述操作的代码：

**1011 0100 0000 1010**

为了书写和记忆方便可用十六进制数表示：**B40A**

又如，要让计算机完成4×6+40的算式运算。假设参加运算的数据事先分别存放在寄存器AL、BL和CL中，并要求将运算结果存放到寄存器AL中。

用机器指令来实现的代码为：

| | |
|---|---|
| 1111 0110 1110 0011 | 十六进制数：F6E3 |
| 0000 0000 1100 1000 | 十六进制数：00C8 |

在32位二进制数表示的机器语言程序代码中，包含了乘法和加法运算操作，其中前16位代码表示了乘法运算，后16位代码表示了加法运算。

机器指令中既包含了指示运算功能的代码，又给出了参加运算的操作数据，表示非常详细。

优点：机器语言最直接地表示了计算机内部的基本操作，用它编制的程序在计算机中运行的效率最高。即运行速度最快，程序长度最短。

缺点：用二进制数表示的内容既不便于记忆又难于阅读。

## 2. 高级语言

高级语言将计算机内部的操作细节屏蔽起来，用户不需要知道计算机内部数据的传送和处理的细节，使用类似于自然语言的一些语句来编制程序，完成指定的任务。

特点：程序设计简单，但程序效率较机器语言低。

## 3. 汇编语言

### （1）定义

虽然高级语言方便了人对计算机的使用，但其运行效率较低。在一些应用场合，如系统管理,实时控制等，难于满足要求。因此又希望使用机器语言。

为了便于记忆和阅读，使用字母和符号来表示机器语言的命令，用十进制数或十六进制数来表示数据，这样的计算机程序设计语言就称为汇编语言。

## （2）汇编语言程序与机器语言程序的关系

一条汇编语言的语句与一条机器语言指令对应，汇编语言程序与机器语言程序效率相同.

例如,对于前述的4×6+40算式运算，如果把机器语言程序改写为汇编语言程序，则为以下两条汇编指令：

```
MUL  BL
ADD  AL, CL
```

## （3）不同类型计算机有不同的机器指令系统和汇编语言描述

为了学习和使用某种计算机的汇编语言，必须熟悉计算机的内部组成结构。但并非要掌握计算机系统的全部硬件组成，只需掌握用汇编语言编制程序时所涉及到的那些硬件的结构和功能。

对一台计算机来说，机器语言的执行主要取决于该计算机的中央处理器CPU。因此熟悉计算机内部结构主要是指CPU的功能结构。它包括：

- CPU中有多少个寄存器及其作用？
- CPU是如何访问存储器？
- 输入输出操作的方式有哪些？

在本课程中，将以IBM-PC系列微型计算机及其汇编语言为例，学习汇编语言程序设计的基本原理、方法和技巧。

# § 1.2 学习和使用汇编语言的目的

**1.学习和使用汇编语言可以从根本上认识、理解计算机的工作过程。**

通过用汇编语言编制程序可以更清楚地了解计算机是如何完成各种复杂的工作。在此基础上，程序设计人员能更充分地利用机器硬件的全部功能，发挥机器的长处。

**2. 在计算机系统中，某些功能必须用汇编语言程序来实现。**

如：机器自检、系统初始化、实际的输入输出设备的操作等。

## 3. 汇编语言程序的效率高于高级语言程序

“效率”有两个方面的含义：程序的目标代码长度和运行的速度。

在某些要求节省内存空间和提高程序运行速度的应用场合，如实时过程控制、智能仪器仪表等，常常用汇编语言来编制程序。

# § 1.3 进位计数制及其相互转换

## 一. 进位计数制

使用一定个数的数码的组合来表示数字，这种表示方法称为进位计数制。根据所使用的数码的个数，就产生了不同的进位计数制。

如十进制数，用0、1～9十个数码的组合来表示数字。每个数码排在不同位置，所表示的数值大小不相同。

例如：222从右边开始,第一个2表示2个1，第二个2表示有2个10，第三个2表示有2个100.

将各个位置上所表示的基本数值称为位权，简称权。

不同的进位制和不同的位置其位权是不同的。位权乘以对应位置上的数码就等于该数位上数值的大小。

每个数位上能使用不同数码的个数称为基数。

例如十进制有十个数码0～9，基数为10，二进制基数为2。

每个数位能取的最大数码值=基数-1。如十进制为10-1=9

在计算机中数据表示一般采用二进制数，因为它在计算机中最容易表示和存储，且适合于逻辑值的表达与运算。

对人来说二进制不便于书写和阅读，因此书写时常使用8进制和16进制。

二进制与8进制、16进制之间有非常简单的转换关系：3位二进制数与一位8进制数对应，4位二进制数与一位16进制数对应。

在书写不同进位计数制数时，为了区别，常在数的尾部用一个字母来表示。

**B（Binary）——二进制数**
**O (Octal)或Q——八进制数**
**D (Decimal)——十进制数**
**H (Hexadecimal)——十六进制数。**
**如未使用任何字母，则默认表示是十进制数。**

**例如，10B, 10Q, 10D,10H**

## 二. 各种数制间的相互转换

由于二进制与八进制和十六进制间的转换很简单，下面主要讨论二进制与十进制之间的相互转换。

### 1. 十进制整数转换为二进制数

有两种转换方法：

#### （1）减权定位法

- 从二进制数高位起，依次用待转换的十进制数与各位权值进行比较；
- 如够减，则该数位系数**Ki=1**，同时减去该位权值，余数作为下一次比较的值；
- 如不够减，则**Ki=0** 。

例：将325转换为二进制数，直到余数为0。
首先确定二进制数的最高位
因为$2^9$(512)>325>$2^8$(256)。因此从$K_8$位开始比较。

| 减数比较 | Ki | 对应二进制数 |
|---|---|---|
| 325-256=69 | K8 | 1 |
| 69<128 | K7 | 0 |
| 69-64=5 | K6 | 1 |
| 5<32 | K5 | 0 |
| 5<16 | K4 | 0 |
| 5<8 | K3 | 0 |
| 5-4=1 | K2 | 1 |
| 1<2 | K1 | 0 |
| 1-1=0 | K0 | 1 |

所以　325D=101000101B

## (2) 除基取余法

将十进制数除以基数2，其余数为二进制数的最低位，再用其商除2，其余数为次低位，反复做下去，直到商0.

| 除基 | 余数 | Ki |
|---|---|---|
| 2 \| 325 | | |
| 2 \| 162 | 1 | **k0** |
| 2 \| 81 | 0 | **k1** |
| 2 \| 40 | 1 | **k2** |
| 2 \| 20 | 0 | **k3** |
| 2 \| 10 | 0 | **k4** |
| 2 \| 5 | 0 | **k5** |
| 2 \| 2 | 1 | **k6** |
| 2 \| 1 | 0 | **k7** |
| 0 | 1 | **k8** |

这种转换方法同样适合于其它进制数之间的转换。

# 2.十进制小数转换为二进制数

## (1) 减权定位法

例 将十进制数0.645转换为二进制数

| 减权比较 | Ki | 对应二进制数 |
|---|---|---|
| 0.645-0.5 =0.145 | k-1 | 1 |
| 0.145<0.25 | k-2 | 0 |
| 0.145-0.125=0.02 | k-3 | 1 |
| 0.02<0.0625 | k-4 | 0 |
| 0.02<0.03125 | k-5 | 0 |
| 0.02-0.015625 | k-6 | 1 |

所以 0.645D=0.101001B

转换时应根据程序要求的精度或计算机的字长来确定二进制的位数.

## （2）乘基取整法

例 将**0.8125D**转换为二进制数

| 乘以基数 | **Ki** | 整数部分 |
|---|---|---|
| 0.8125×2＝1.625 | K-1 | 1 |
| 0.625×2＝1.25 | K-2 | 1 |
| 0.25×2＝0.5 | K-3 | 0 |
| 0.5×2＝1. | K-4 | 1 |

所以 **0.8125D＝0.1101B**

## 3. 二进制整数转换为十进制数

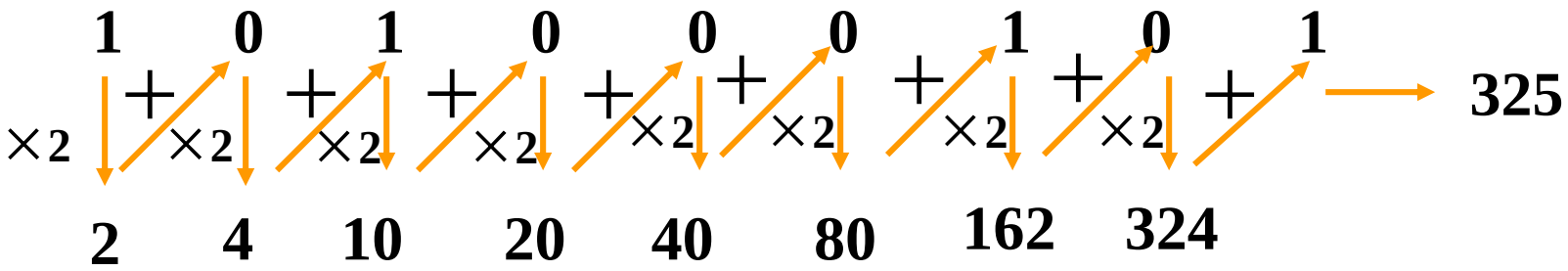
### （1）按权相加法

例 $101000101B = 1\times2^8+1\times2^6+1\times2^2+1\times2^0$

$= 256+64+4+1$

$= 325$

### （2）逐次乘基相加法



# 4. 二进制小数转换为十进制数

## （1）按权相加法

例　$0.101001B = 1\times2^{-1}+1\times2^{-3}+1\times2^{-6}$

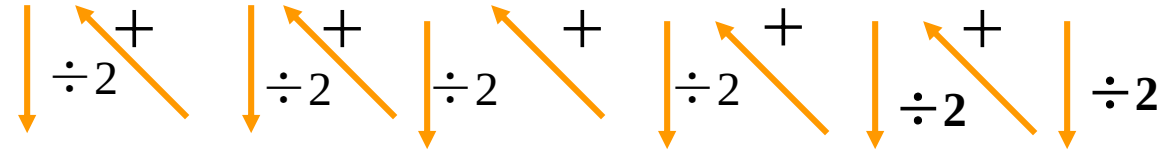
$=0.5+0.125+0.0156$

$=0.640625D$

## （2）逐次除基相加法

转换从最低位开始

例　0.　1　0　1　0　0　1




0.640625　0.28125　0.5625　0.125　0.25　0.5

0.101001B＝0.640625D

# 5. 二进制与八进制和十六进制间的转换

二进制与八进制和十六进制之间的对应关系很简单：

三位二进制数对应一位八进制数，四位二进制数对应一位十六进制数。

例如：10100010B =10 100 010

2 4 2

所以 10100010B=242Q

10100010B = 1010 0010

A 2

所以 10100010B = A2 H

# § 1.4 带符号数的表示

在一般算术表示中使用”“+”和“-”来表示正数与负数，而在计算机中使用“0”和“1”来表示正数和负数。

用“＋”或“－”表示正负的数叫真值
用“0”或“1”表示正负的数叫机器数

带符号的机器数可以用原码、反码和补码三种不同码制来表示。一般计算机中大多采用补码表示。

## 一、原码表示

二进制数的最高位表示符号，0表示正，1表示负。数值 部分用二进制数绝对值表示

8位二进制数原码的最大数为01111111（+127）

最小数为11111111 （-127）

8位二进制数表示范围：－127≤X≤+127

0的原码有两种表示形式：00000000和10000000（+0和-0）

## 二、补码的表示

### 1.补码的定义

带符号数X的补码表示$[X]_{补}$定义为：

$$[X]_{补}=M+X \quad (\text{Mod } M)$$

其中模数M根据机器数的位数而定，如位数为8则$M=2^8$

用补码表示的机器数，符号位仍然表示数的符号：0为正,1为负。对于正数，补码与原码相同，对于负数需要进行变换。

## 2.由真值、原码变换为补码

由于正数的原码与补码相同，下面讨论负数的变换方法。

负数的真值变换为补码的方法：将各位变反（0变1，1变0）然后在最低位加1.

负数的原码变换为补码：保持符号位不变,其余各位变反，最低位加1。

例 将－59变换为补码

| | | | |
|---|---|---|---|
| 真值 | －00111011 | 原码 | 10111011 |
| 变反 | 11000100 | 变反 | 11000100 |
| 加1 | 11000101 | 加1 | 11000101 |

所以 [-59]补＝11000101

## 3.补码数的表示范围

当位数为8时，最大补码为01111111=$[+127]_{补}$

最小补码为10000000=$[-128]_{补}$

0的补码只有一个，$[0]_{补}$=00000000，而10000000是$[-128]_{补}$

11111111=$[-1]_{补}$

对于16位数，则补码表示范围为-32768~+32767

## 4. 补码的加减运算

规则：$[X+Y]_{补}=[X]_{补}+[Y]_{补}$

$[X-Y]_{补}=[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}$

其中$[-Y]_{补}$是对$[Y]_{补}$执行一次求补运算

求补运算是将原数连同符号位一起（不管是正还是负）按位求反，再在最低位加1。

## （1）加法运算：X+Y

例1 X=74D　Y=41D

$[X]_{补}$＝01001010　$[Y]_{补}$＝00101001

$$\begin{array}{r} 01001010 \\ + \quad 00101001 \\ \hline 01110011 \end{array}$$

所以 $[X]_{补}+[Y]_{补}=01110011=[115]_{补}$

例2 X=74D　Y=-41D

$[X]_{补}$＝01001010　$[Y]_{补}$＝11010111

$$\begin{array}{r} 01001010 \\ + \quad 11010111 \\ \hline 1 \quad 00100001 \end{array}$$

自动舍去

所以 $[X]_{补}+[Y]_{补}=00100001=[33]_{补}$

例 3　X= -74D　　Y=41D

$[X]_{补}$＝10110110　$[Y]_{补}$＝00101001

```
  10110110
+ 00101001
----------
  11011111
```

所以 $[X]_{补}+[Y]_{补}=11011111=[-33]_{补}$

例4　X= -74D　Y= -41D

$[X]_{补}$＝10110110　$[Y]_{补}$＝11010111

```
  10110110
+ 11010111
----------
1 10001101
```

自动舍去

所以 $[X]_{补}+[Y]_{补}=10001101=[-115]_{补}$

## (2) 减法运算

例5　X=74D　　Y=41D

$[X]_{补}$＝01001010　　$[Y]_{补}$＝00101001　$[-Y]_{补}$=11010111

```
   01001010
 + 11010111
-----------
 1 00100001
```

自动舍去

所以 $[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}=00100001=[33]_{补}$

例6　X=74D　Y=－41D

$[X]_{补}$＝01001010　　$[Y]_{补}$＝11010111　$[-Y]_{补}$=00101001

```
   01001010
 + 00101001
-----------
 1 01110011
```

自动舍去

所以 $[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}=01110011=[115]_{补}$

例7　X= -74D　　Y=41D

$[X]_{补}$＝10110110　　$[Y]_{补}$＝00101001　$[-Y]_{补}$=11010111

自动舍去

```
  10110110
+ 11010111
-----------
1 10001101
```

所以 $[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}$＝10001101＝$[-115]_{补}$

例8　X= -74D　　Y= -41D

$[X]_{补}$＝10110110　　$[Y]_{补}$＝11010111　$[-Y]_{补}$=00101001

自动舍去

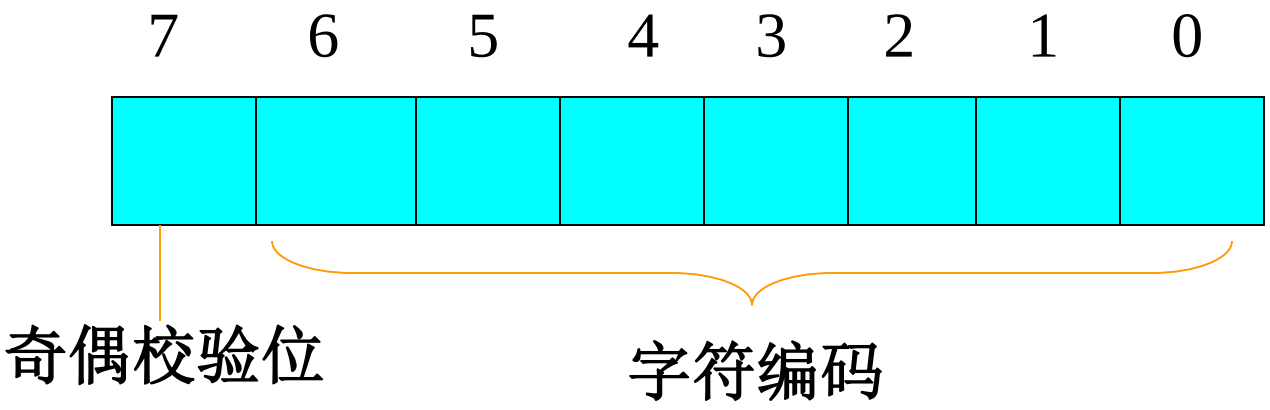
```
  10110110
+ 00101001
-----------
1 11011111
```

所以 $[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}$＝11011111＝$[-33]_{补}$

# § 1.5 字符的表示

在计算机内部，各种字符（字母、符号、数字码）都是按一定的方式编写成二进制信息。不同的计算机以及不同的场合所采用的编码形式可能不同。目前最广泛采用的是ASCII码（American Standard Code for Information Interchange）

标准ASCII码为一字节，其中用低七位表示字符编码(见附录A),用最高位表示奇偶数验位。



**标准ASCII码共有128个，可分为两类：**

**非打印ASCII码,共33个,用于控制操作, 如BEL(响铃07H), DEL(删除7FH),CR(回车,0DH), LF(换行,0AH).**

**可打印ASCII码共有95个，如数字符0~9，大小写字母等。**

# § 1.6 基本逻辑运算

计算机内部采用二进制数表示信息，具有物理实现容易、可靠性高的优点，且由于状态“0”和“1”正好与逻辑运算中的逻辑“真”和“假”对应，因此可以用“0”和“1”来表示逻辑变量的取值，很容易地实现各种复杂的逻辑运算。

在计算机的指令系统中，一般都有逻辑运算指令。下面介绍几种常见的基本逻辑运算。

## 1.“与”运算（AND）

“与”运算也叫逻辑乘，常用 ∧ 或 · 表示。

设有逻辑变量A和B，则“与”运算为：

**F=A ∧ B 或 F=A · B**

“与”运算是指：仅当逻辑变量A与B都是1时，运算结果F才为1。其它情况F为0，

即有：

**0 ∧ 0 =0　0 ∧ 1 = 0　1 ∧ 0 = 0　1 ∧ 1 =1**

## 2. “或”运算（OR）

“或”运算也叫逻辑加，用∨或＋表示。即有：

**F=A ∨B　　F=A +B**

**“或”运算是指当逻辑变量A与B中，至少有一个为1时，结果F为1，其他情况F为0。**

**即有：0 ∨0＝0　　0 ∨1＝1　　1 ∨0＝1　　1 ∨1＝1**

## 3. “非”运算

“非”运算是指对逻辑变量取相反的一个逻辑值。

逻辑“非”运算通常是在逻辑变量上方加一横线表示。

如A为1，则 $\overline{A}=0$，若A为0，则 $\overline{A}=1$

“非”运算规则为：

$\overline{1} = 0 \qquad \overline{0} = 1$

## 4. “异或”运算（XOR）

通常用 ⊕表示，即 F =A ⊕ B

“异或”运算是指：当A和B相同时（同时为1或同时为0），运算结果F为0，而不同时，F为1。

运算规则为：

1 ⊕ 1=0　　1 ⊕ 0=1　　0 ⊕ 1=1　　0 ⊕ 0=0

上述四种基本逻辑运算规则用真值表表示为：

| A | B | A ∧ B | A ∨ B | $\overline{A}$ | $\overline{B}$ | A ⊕ B |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

# 第二章　IBM-PC微机的功能结构

本章主要内容：

- IBM-PC微机基本结构
- 8086/8088寄存器结构及其用途
- 8086/8088系统的存储器组织结构
- 8086/8088系统的堆栈及其操作方法

# 2.1 IBM-PC微机基本结构

## 一. 微机的一般构成

一般计算机应包括五大部件：

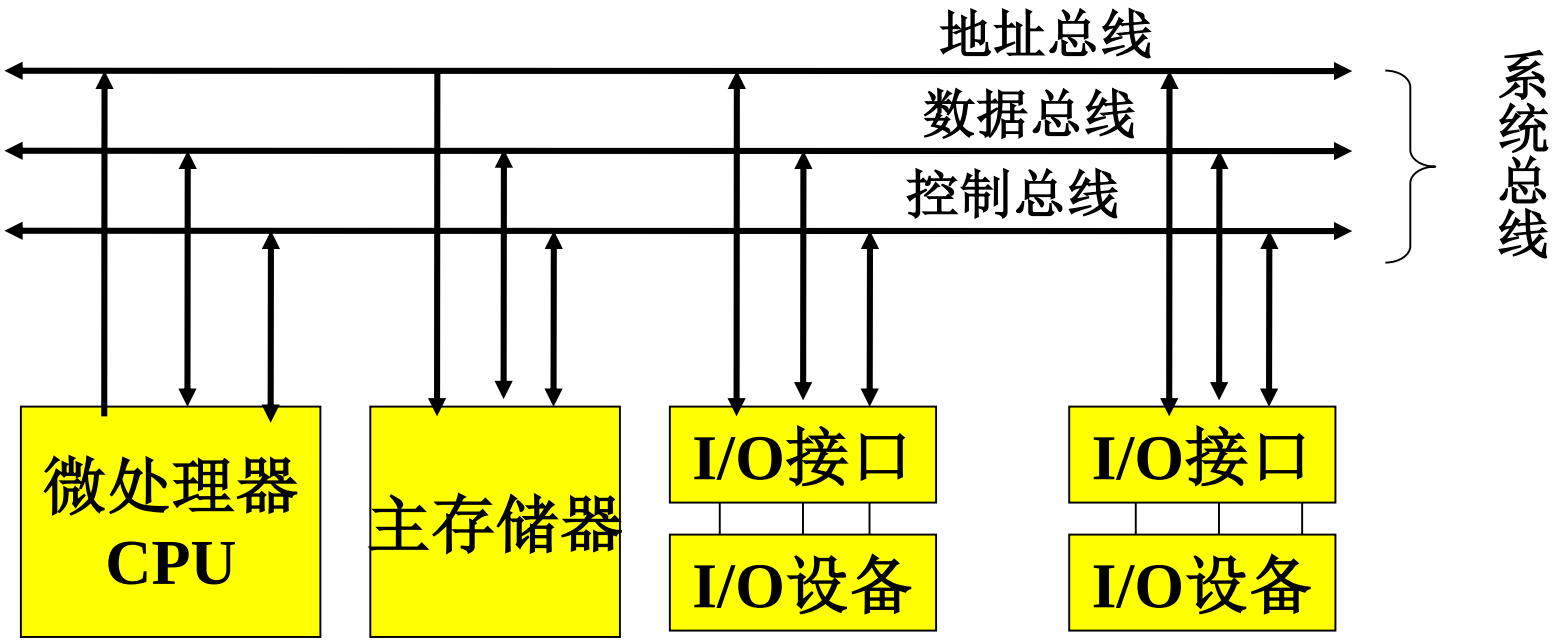
运算器、控制器、存储器、输入设备和输出设备。

由于微机的主要特点是其体积很小，因此在系统设计上就有一些特殊考虑。

➢将运算器和控制器两大部件集成在一个集成电路芯片上,称为中央处理器 ,简称CPU,也叫微处理器.

- 系统采用总线结构，具有较大的灵活性和扩展性。

## 微机硬件系统基本组成框图



## 1、中央处理器CPU

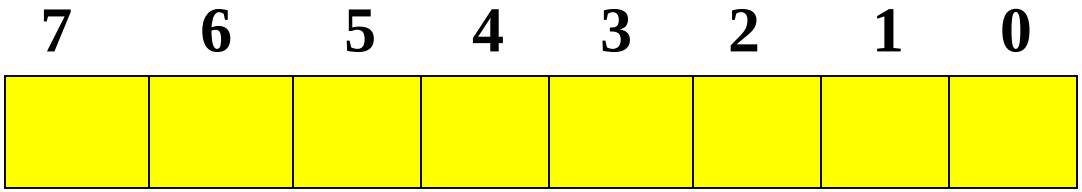
微型计算机中的中央处理器也叫微处理器。它包括运算器和控制器。

**功能：** 分析从主存储器取来的各条指令的功能，控制计算机各部件完成指定功能的各项操作。

## 2、主存储器

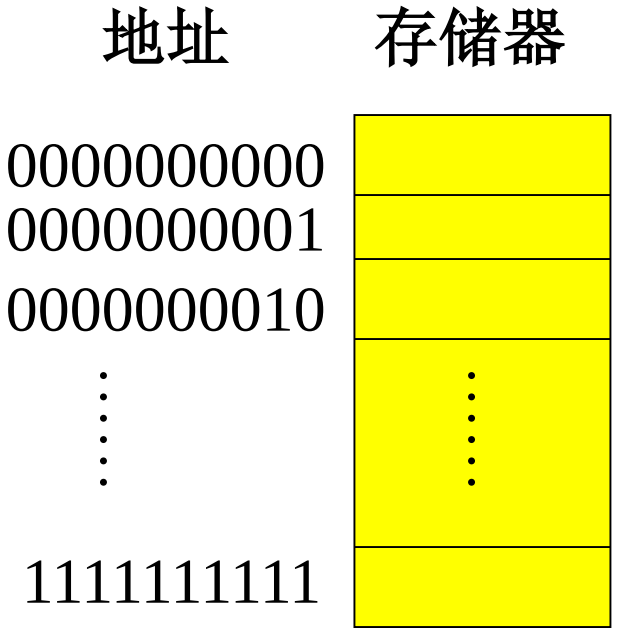
- 主存储器是用来存放程序和数据的部件。它由若干个存储单元构成。
- 存储单元的多少表示存储器的容量。每个存储单元使用一个唯一的编号来标识，称为存储单元的地址。
- 对每个存储单元内容的存和取是按照地址进行访问的。

计算机存储信息的基本单位是一个二进制位，一位可存储一个二进制数0或1。每8位组成一个字节（BYTE)。




在大多数计算机中，存储器的组织都是以字节为基本单位。每一个基本单位称为一个存储单元。

指示存储单元编号的地址长度决定了存储器的最大容量，例如一个10位二进制数表示的地址，可以用来区分$2^{10}$＝1024＝1K个单元。



习惯上将CPU与主存储器合称为主机

在计算机中，除了主存储器之外，一般还配置有辅助存储器，简称辅存。由于它的位置是在主机之外，因此也叫做外存。

## 3、输入输出设备及接口

- 输入设备将外部信息（程序、数据和命令）送入计算机。包括键盘、鼠标等。

- 输出设备将计算机处理后的结果转换为人或其它系统能识别的信息形式向外输出。如显示器、打印机等。

- 有的设备既具有输入功能又具有输出功能。如磁盘、磁带、触摸显示屏等。

- 由于I/O设备的工作速度、工作原理以及所处理的信息格式等与主机相差很大，因此I/O设备要通过I/O接口才能与系统总线连接。
- I/O接口是主机与I/O设备之间设置的逻辑控制部件。通过它实现主机与I/O设备间的信息传送。

## 4、系统总线

- 系统总线将CPU、存储器和I/O设备连接起来，实现各大部件之间的各种信息传送。
- 系统总线包括地址总线、数据总线和控制总线三组。它们分别用于传送不同的信息。

# 二、 Intel 8086/8088 CPU的功能结构

汇编语言程序是由一系列的指令(指令序列)构成。

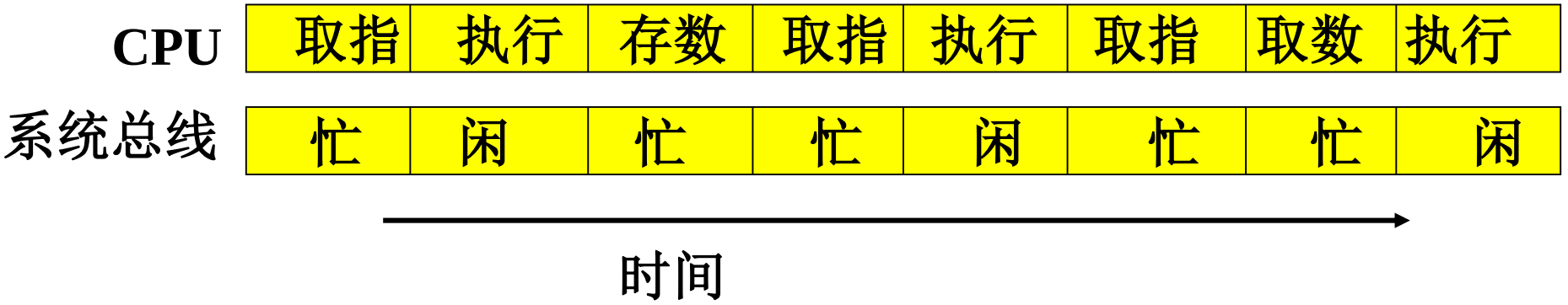
指令是构成汇编语言程序的最基本单位，就象高级语言中的语句。

CPU执行指令序列就是重复执行以下两个步骤：

- 从存储器中取指令
- 执行指令规定的操作

这两个步骤的执行又分为串行方式和指令流水线方式。

# 1. 串行方式




特点：

（1）当CPU在指令执行阶段，不需要占用系统总线，但此时总线也不工作，因此系统总线的空闲时间比较多 。

（2）在从存储器取指令、取数据或存数据时，总线处于忙状态，其所占用的时间也较长。而CPU却只需要花很短的时间去处理，因此大部分时间处于闲置状态。

采用串行工作方式的计算机其运行速度较慢

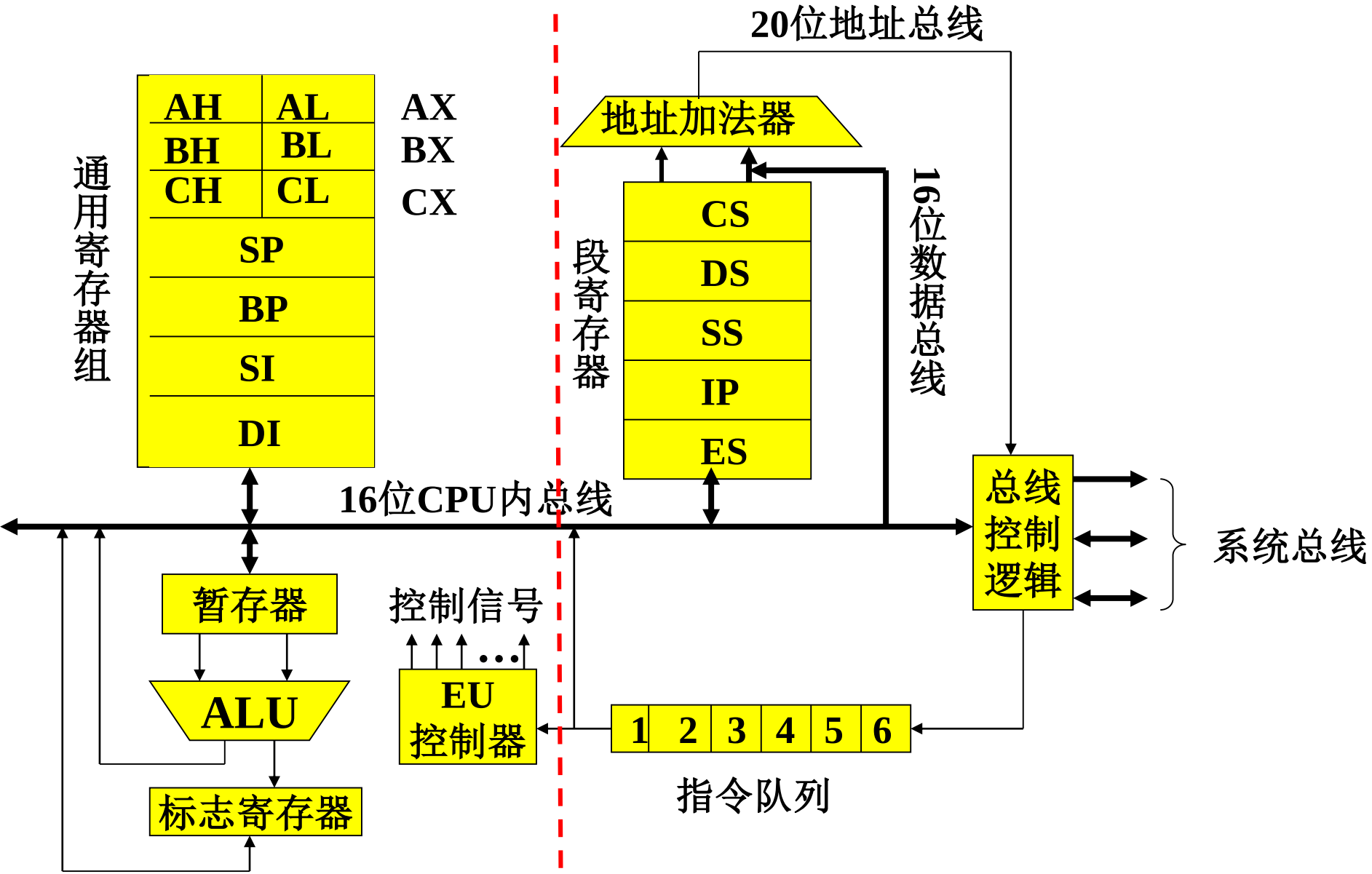
## 2.指令流水线方式

采用指令流水线工作方式的计算机具有较高的工作效率。CPU内部采用了一种先进的指令流水线结构，这种结构可以有效而充分地利用各主要硬件资源。

指令流水线结构最先出现在Intel公司的8086/8088 CPU中

要实现指令流水线方式，从CPU组成结构上要划分成多个单元。8086/8088CPU被划分成两个单元。

# 8086CPU结构


20位地址总线
AH
AL
AX
BH
BL
BX
CH
CL
CX
SP
BP
SI
DI
通用寄存器组
地址加法器
CS
DS
SS
IP
ES
段寄存器
16位数据总线
16位CPU内总线
总线控制逻辑
系统总线
暂存器
ALU
标志寄存器
控制信号
EU控制器
1
2
3
4
5
6
指令队列
执行单元（EU）
总线接口单元（BIU）

# （1）执行单元EU

EU的主要任务是分析与执行指令，具体包括：

A、从指令队列中取出指令代码，由控制器译码后产生相应的控制信号，控制各部件完成指令规定的操作。

B、对操作数执行各种指定的算术或逻辑运算。

C、向总线接口单元BIU发送访问主存或I/0的命令，并提供相应的地址和传送的数据。

# （2）总线接口单元BIU

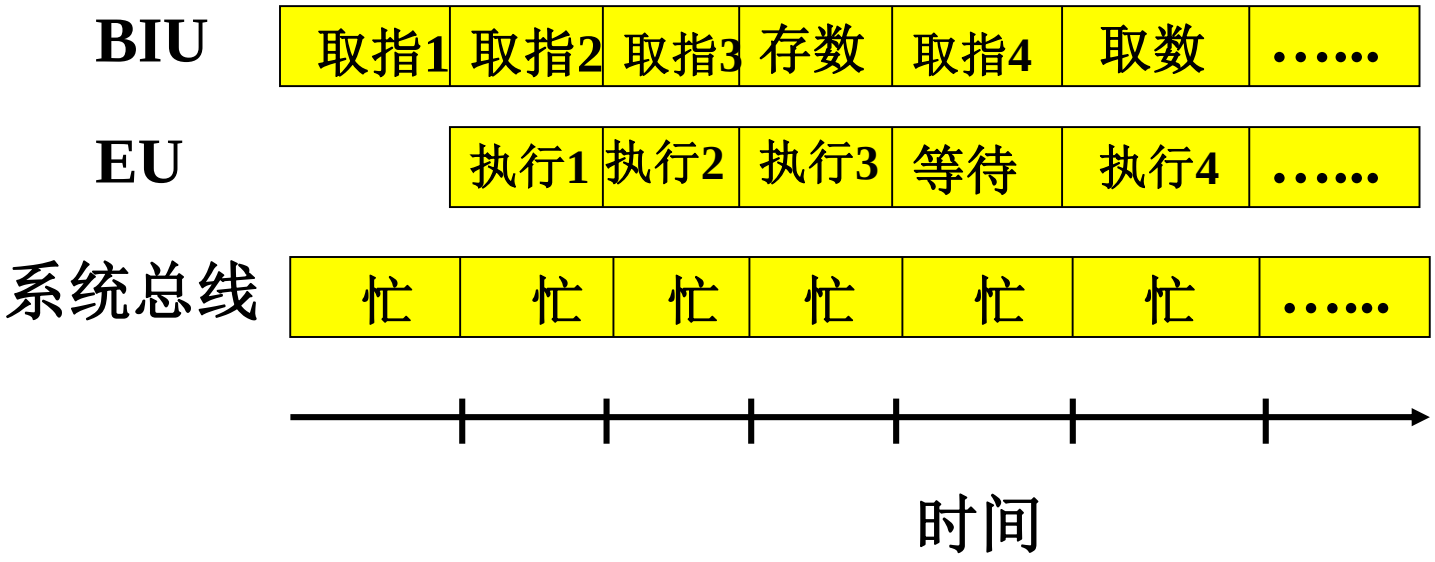
BIU负责CPU与存储器、I/O的信息传送。具体包括：

A、取指令——根据CS寄存器和指令指针IP形成20位的物理地址，从相应的存储器单元中取出指令，暂存到指令队列中，等待EU取走并执行。

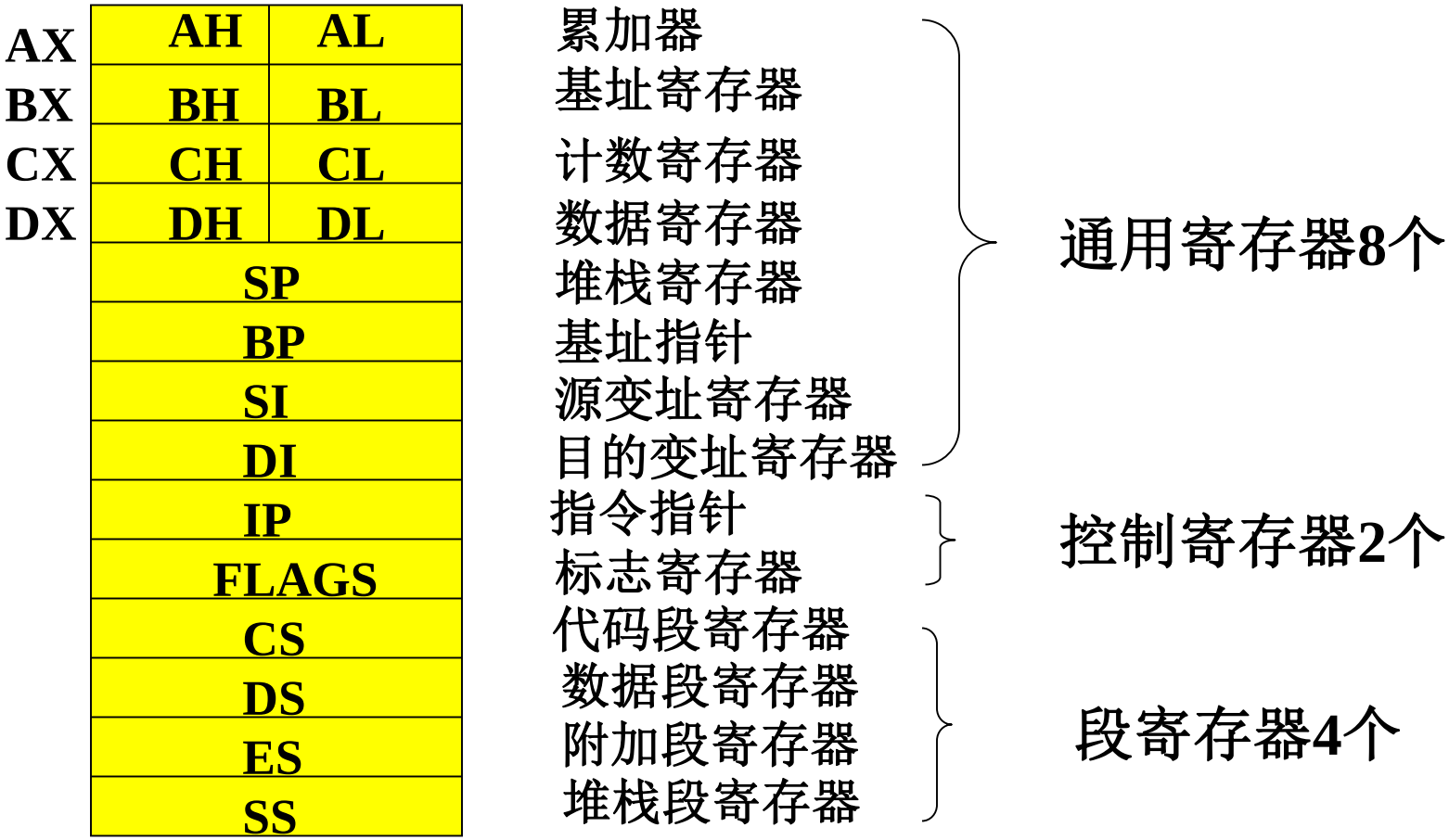
B、存取数据——在EU执行指令的过程中，需要与存储器或I/O端口传送数据时，由EU提供的数据和地址，结合段寄存器，通过外部总线与存储器或I/0进行数据的存取。

EU和BIU是既分工又合作的两个独立部分。它们的操作在一定程度上是并行工作的，分别完成不同的任务，从而大大加快了指令执行速度。

Intel 8086/8088 运行时执行过程大致如下图所示。



# 2. 2 Intel 8086/8088CPU寄存器结构及其用途



# 一、通用寄存器

Intel 8086/8088有8个16位通用寄存器，它们具有良好的通用性，并且还可以用作某个特定的功能，可以由程序设计人员进行编程访问。

## 1. 数据寄存器

它包括AX、BX、CX和DX四个寄存器。它们中的每一个既可以是16位寄存器，也可以分成两个8位寄存器使用。即可以当作8个独立的8位寄存器使用。

数据寄存器既可以用来存放参加运算的操作数，也可以存放运算的结果。在多数情况下，使用这些寄存器时必须在指令中明确指示。

例：MOV AX , BX；将BX的内容送到AX中

ADD CH , DH；将DH和CH的内容相加，结果送到CH

在有些指令中，不需要明确指出使用的寄存器名，即隐念使用了某寄存器，称为隐含使用。

例如，在循环指令 LOOP中，CX被隐含指定作循环次数计数用。

个别指令对寄存器有特定的使用，并且又必须在指令中指明它的名字，这类寄存器的使用称为特定使用。

例如，移位指令 SHL AX，CL

CL被固定用作移位次数。

## 2. 指针寄存器

指针寄存器有堆栈指针SP和基址指针BP

它们一般被用来存放16位地址，在形成20位的物理地址时常被作为偏移量使用。

SP指针——在进行堆栈操作时，被隐含使用，被用来指向堆栈顶部单元。

BP指针——被用来指向堆栈段内某一存储单元。BP除用作地址指针外也可以象数据寄存器一 样，存放参加运算的操作数和运算的结果。

### 3. 变址寄存器

有两个16位的变址寄存器SI和DI，一般被用来作地址指针。

SI——源变址寄存器　　DI——目的变址寄存器

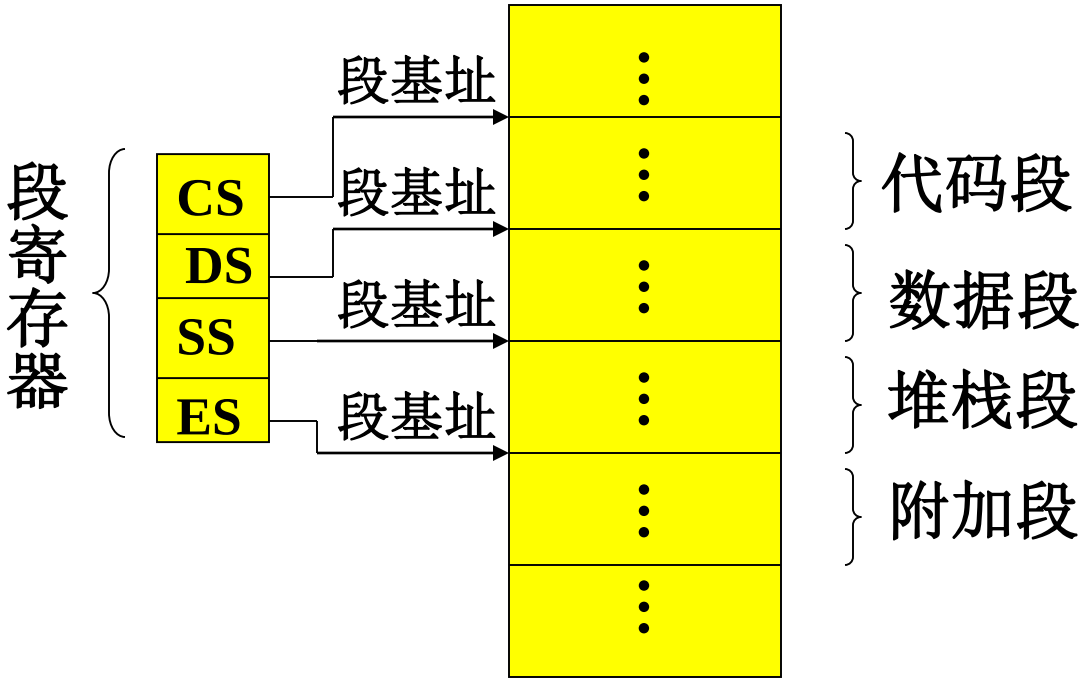
同BP寄存器一样，SI和DI也可以用作通用数据寄存器存放操作数和运算结果。

# 二、段寄存器

✓8086/8088CPU在使用存储器时，将它划分成若干个段。

✓每个段用来存放不同的内容，如程序代码、数据等等。

✓每个存储段用一个段寄存器来指明该段的起始位置（也叫段基址）。



**CPU在访问存储器时必须指明两个内容：**

**（1）所访问的存储单元属于哪个段，即指明使用的段寄存器。**

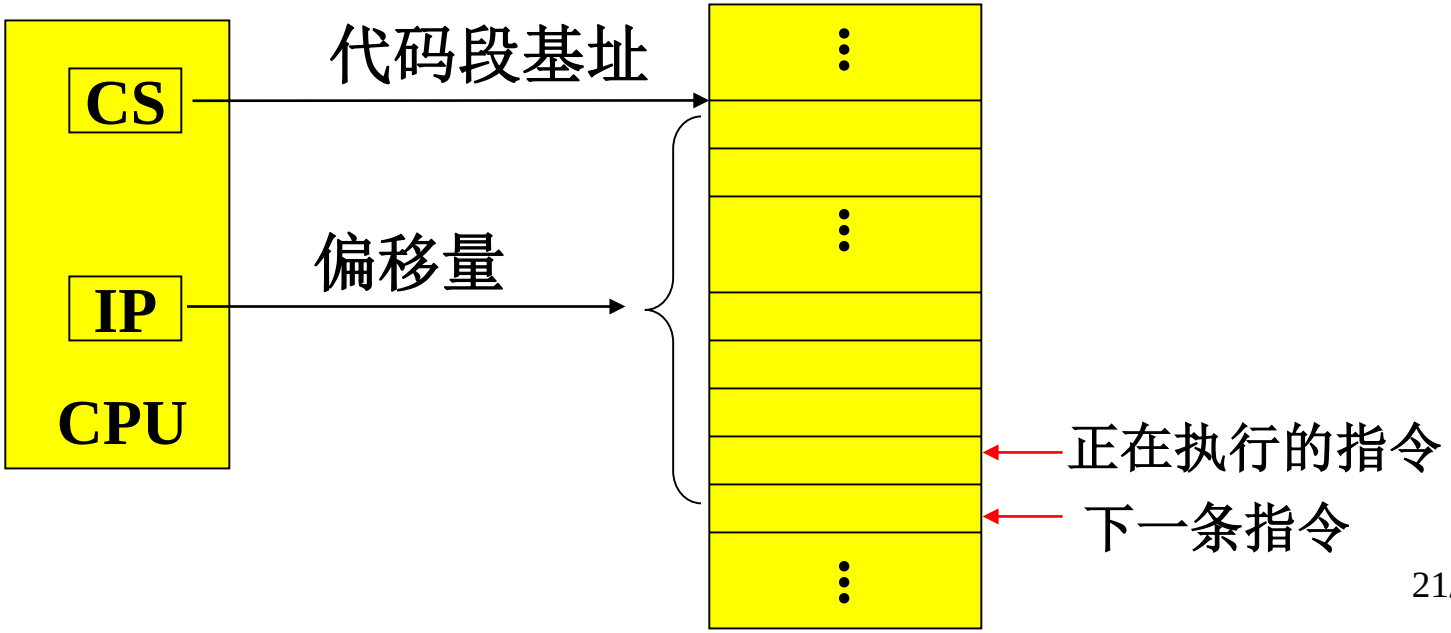
**（2）该存储单元与段起始地址（段基址）的相距多少，即偏移量。**

**在程序设计中，一个程序将存储器划分成多少个存储段是任意的。但在程序运行的任何时刻最多只有4个段，分别由CS、DS、ES和SS指定。**

# 三、指令指针IP

CPU在从存储器取指令时，以段寄存器CS作为代码段的基址指针，以IP的内容作为偏移量，共同形成一条指令的存放地址。

当CPU从内存中取出一条指令后，IP内容自动修改为指向下一条指令。



**注意：IP的内容不能被直接访问，既不能用指令去读IP的值，也不能用指令给它赋值。但是可以通过某些指令的执行而自动修改IP的内容。**

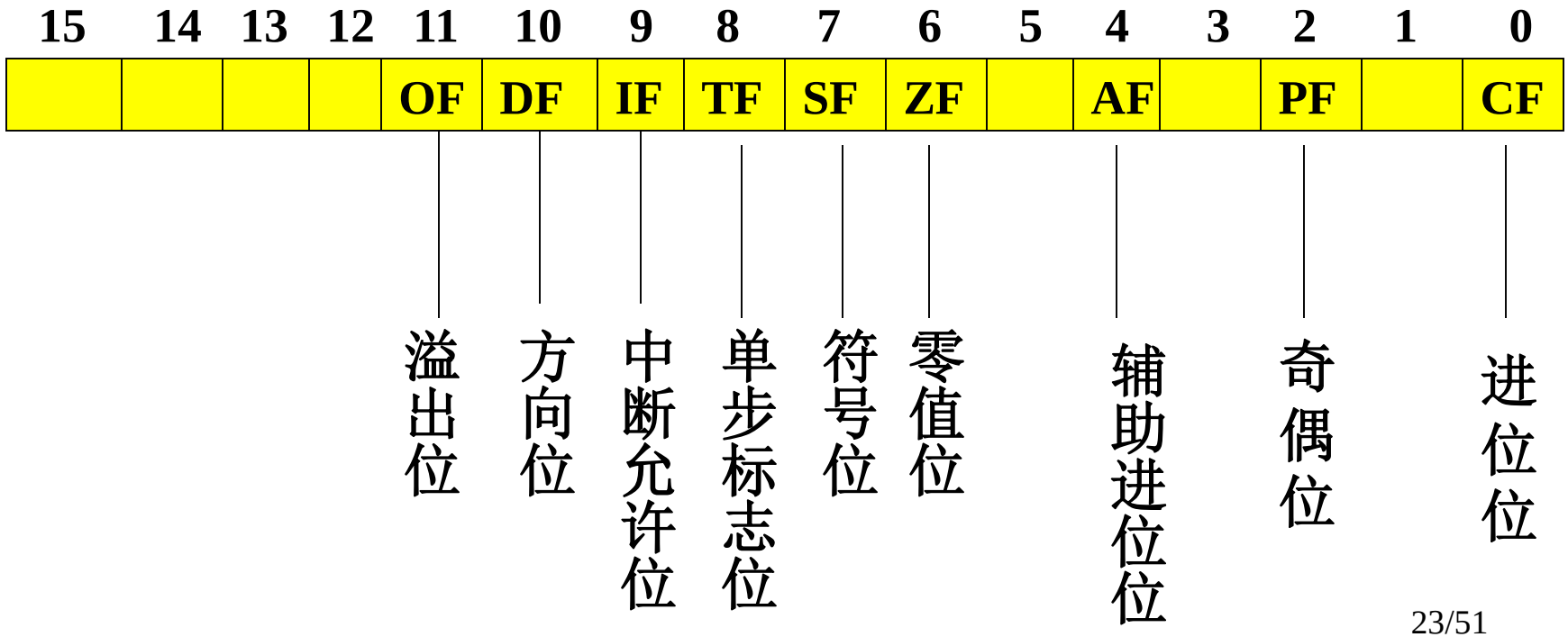
**例如，下面两种指令就可以自动改变IP寄存器的内容。**

- **转移指令将指令中的目的地址的偏移量送入IP**
- **子程序调用指令CALL，将IP原有内容自动压入堆栈，而将子程序的入口地址偏移量自动送入IP，而返回指令RET，又自动从堆栈中弹回原有IP的内容。**

# 四、标志寄存器

标志寄存器是用来反映CPU在程序运行时的某些状态，如是否有进位、奇偶性、结果的符号、结果是否为零等等。

8086/8088CPU中标志寄存器的长度为16位，但只定义了其中的9位。



标志位分为
- 状态标志：CF, PF, AF, ZF, SF, OF
- 控制标志：TF , DF, IF

## 1. 进位标志位CF

在进行算术运算时，若最高位（对字操作是第15位，字节操作是第7位）产生进位或借位时CF被自动置“1”，否则置“0”

在移位类指令中，CF也被用来存放从最高位（左移时）或最低位（右移时）移出的数值（0或1）。

## 2. 奇偶标志位PF

当指令操作结果的低8位中含有1的个数为偶数时，则PF被置1，否则PF被置0。

注意：PF只反映操作结果的低8位的奇偶性，与指令操作数的长度无关。

### 3. 辅助进位标志位AF

在进行算术运算时，若低字节的低四位向高4位产生进位或借位，即第3位产生进位或借位时，AF位被置1，否则置0。AF标志位用于十进制运算的调整。

注意：AF只反映运算结果低八位，与操作数长度无关。

### 4. 零值标志位ZF

若运算结果各位全为0，则ZF被置1，否则置0。

### 5. 符号标志位SF

将运算结果视为带符号数，当运算结果为负数时SF被置1，为正数时，则置0。

由于第7位是字节操作数的符号位，而第15位是字操作数的符号位，因此，SF位与运算结果的最高位（第7位或第15位）相同。

## 6.溢出标志位OF

当运算结果超过机器用补码所能表示数的范围时，则OF置1，否则置0.

字节数据，机器用补码所能表示的数范围为-128～+127。
字数据的表示范围为：-32768～+32767

注意：溢出与进位是两个完全不同的概念，不能相互混淆。
下面通过几个例子来说明

例如：计算 -85D+（-1D）= - 86D

进位被丢弃

```
    10101011  B
  +）11111111  B
-------------------
 1 ←10101010  B    -86D
```

CF=1,OF=0, 结果正确。

计算 100 D+100 D =200 D

```
    01100100 B
  +）01100100 B
-------------------
    11001000 B  -56 D
```

CF=0，OF=1, 结果发生溢出，即结果出错。

计算 -85 D +-117 D = -202 D

10101011 B

+）10001011 B

---

1←00110110 B 54 D

CF=1, OF=1,结果发生溢出，即结果出错。

## 7. 单步标志位TF（Trace Flag）

单步标志也叫跟踪位，该标志为控制标志位。单步标志位供调试程序使用。

当TF位被设置为1时，每执行一条指令后，CPU暂停运行，即产生单步中断。

## 8. 中断允许标志位IF

该标志位为控制标志位。当IF被设置为1时，CPU可以响应可屏蔽中断，否则不允许响应可屏蔽中断。

## 9. 方向标志位DF

DF也是控制标志位。它被用来规定串操作指令的增减方向。

当DF=0时，串操作指令自动使变址寄存器（SI和DI）的内容递增。当DF=1时，串操作指令自动使变址寄存器的内容递减。

# 2.3 存储器组织结构

## 一、存储器的组成

**1. 存储器是由若干个存储单元构成**

存储单元的多少就表示了存储器的容量。

**2. 每个存储单元存放相同长度的二进制数**

一个存储单元的长度一般为8位二进制数，即一个字节。

**3. 每个存储单元有一个唯一的地址编号——地址**

8086/8088CPU有20根地址线，即它可以产生20位的地址码，它的存储器寻址能力为$2^{20}$，即1兆字节空间。

这一兆字节存储单元的地址范围为：00……0~11……1。

20位　　20位

| 十六进制数地址 | 二进制数地址 | 存储单元（字节） |
| --- | --- | --- |
| 00000H | 00000000000000000000 | |
| 00001H | 00000000000000000001 | |
| 00002H | 00000000000000000010 | |
| ⋮ | ⋮ | ⋮ |
| FFFFEH | 11111111111111111110 | |
| FFFFFH | 11111111111111111111 | |

7　　0

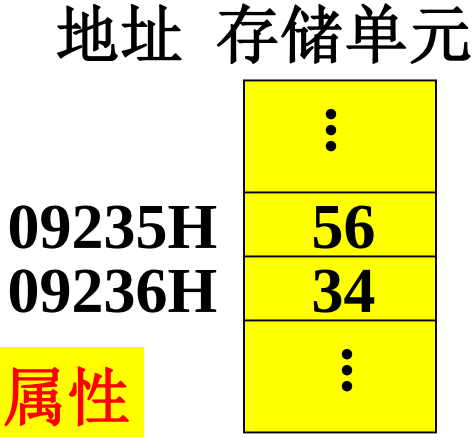
为了方便书写，在源程序中常用5位十六进制数或一个符号来表示一个存储单元的地址。

## 4. 任何两个相邻字节单元就构成一个字单元

一个字存储单元（WORD）的长度为16位二进制数，即两个字节。字单元的地址为两个字节单元中较小地址字节单元的地址。

16位长数据的存放规则是低8位放在较低地址字节单元中，高8位放在较高地址字节单元中。

例如，将数据3456H放在地址为09235H的存储单元中的存储分配。




## 5、在定义一个地址时必须指出是字节或字类型属性

由于存储单元可分为字单元和字节单元，因此8086/8088CPU访问内存的指令中，分为字节访问和字访问两种指令。

# 二、存储器的段结构

由于8086/8088可寻址的存储空间为1MB，需要提供20位长的地址码。而CPU内部的寄存器长度只有16位。能够直接访问的最大地址空间是64KB。

8086/8088系统的存储器段结构具有以下几个特点：

1. 8086/8088CPU将1MB的存储空间划分成若干个段，每个段最大长度为64K（65536）个字节单元组成。

在8086/8088的汇编语言源程序中，用户可以根据自己需要来设定段的个数、各个段长度和每个段的用途。并且代码或数据可以存放在段内任意单元中。

## 2. 每个段的基址（段基址）必须是一个小节的首址。

段基址——一个段的起始地址。

在存储器中规定从0地址开始，每16个字节单元称为一个小节（Paragraph）。因此，1MB内存就可划分为64K个小节。

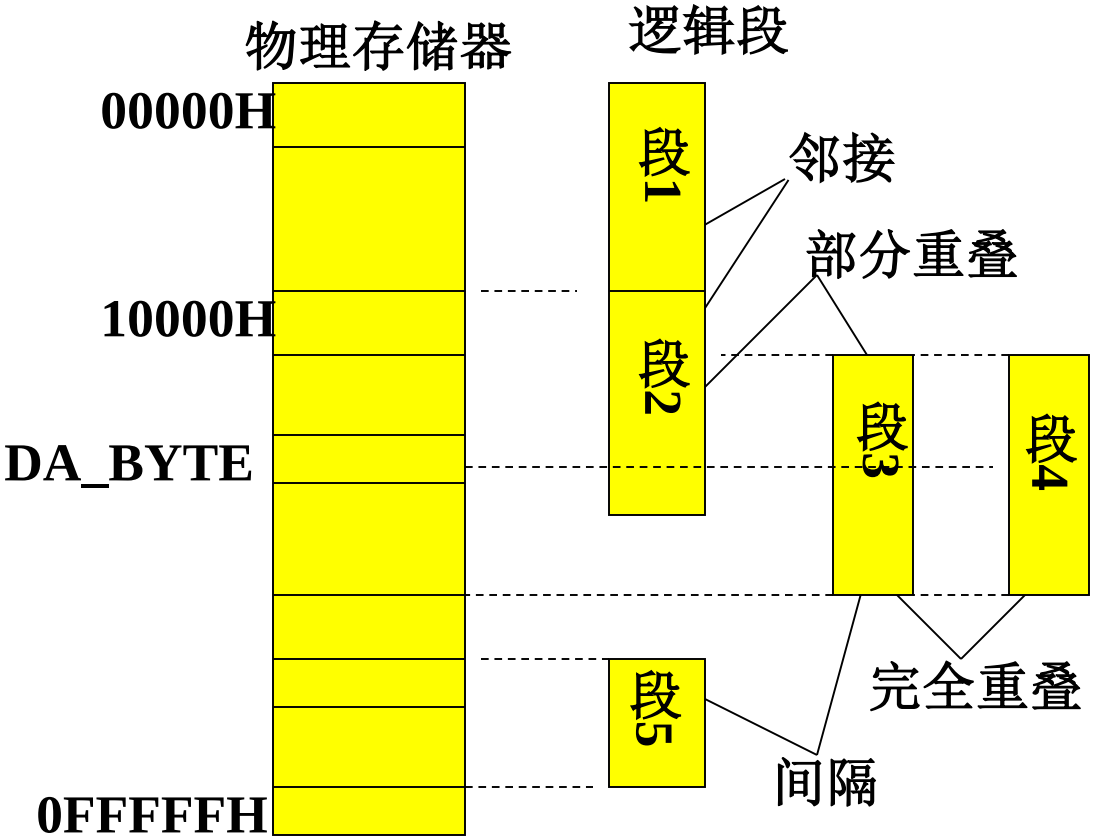
第 1 小节：00000H，00001H，00002H……0000FH
第 2 小节：00010H，00011H，00012H……0001FH
⋮ ⋮ ⋮ ⋮ ⋮
第65535小节：FFFE0H FFFE1H FFFE2H……FFFEFH
第65536小节：FFFF0H FFFF1H FFFF2H……FFFFFH

可以看出，每个小节的首地址最低位必为0（16进制数表示）。因此段基址只能是上述64K个小节首址之一。

**3. 逻辑段在物理存储器中可以是邻接的、间隔的、部分重叠的和完全重叠的等4种情况。**

逻辑段是指在汇编语言源程序中设置的段

**内存中的一个物理存储单元可以映象到一个或多个逻辑段中**




**DA_BYTE物理单元可以映象到逻辑段2、段3和段4中。**

**4. 在任一时刻，一个程序只能访问4个当前段中的内容。**

4个段分别是代码段、数据段、堆栈段和附加段，称为当前段 。4个段寄存器CS、DS、SS和ES分别保存了它们段基址的高16位地址，称为段基值。段基址的最低4位为0。（小节首址的低4位为全0 ）。

## 三、 逻辑地址与物理地址及对应关系

### 1. 物理地址

在1MB的存储空间中，每个存储单元的物理地址是唯一的，它就是该存储单元的20位地址。

**8086/8088的物理地址范围：00000H~0FFFFFH**

CPU与存储器之间的任何信息交换都使用物理地址。

## 2. 逻辑地址

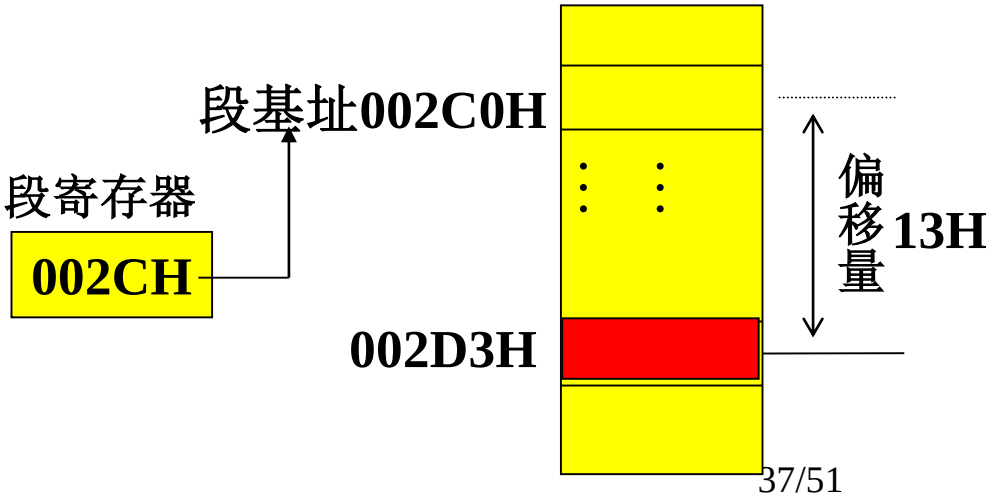
在程序设计中，为了便于程序的开发和对存储器进行动态管理，使用了逻辑地址。

一个逻辑地址包括两个部分：段基值和偏移量

段基值：存放在某一个段寄存器中，是一个逻辑段的起始单元地址（段基址）的高16位。

偏移量：表示某个存储单元与它所在段的段基址之间的字节距离。

对于一个64K的段，当偏移量为0时，就是这个段的起始单元，而偏移量为0FFFFH时，就是这个段的最后一个字节单元。



逻辑地址的表示方法　　段基值：偏移量

例如，3267H：0A0H

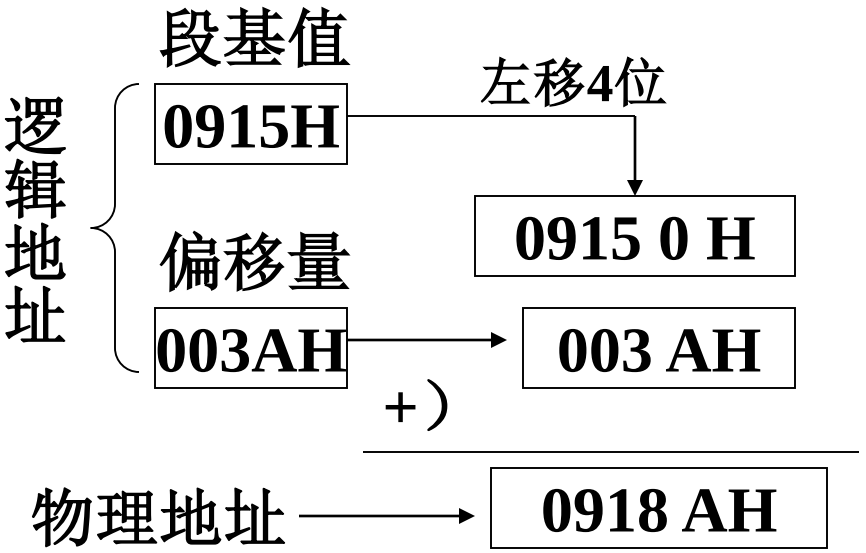
表示该逻辑单元位于段起始地址为32670H，段内偏移量为0A0H个字节。

## 3.逻辑地址转换为物理地址

当CPU要访问存储器时，需要由总线接口单元BIU将逻辑地址转换成物理地址。

转换方法：将逻辑地址的段基值左移4位，形成20位的段基址（低位为0）然后与16位的偏移量相加，结果即为20位的物理地址。

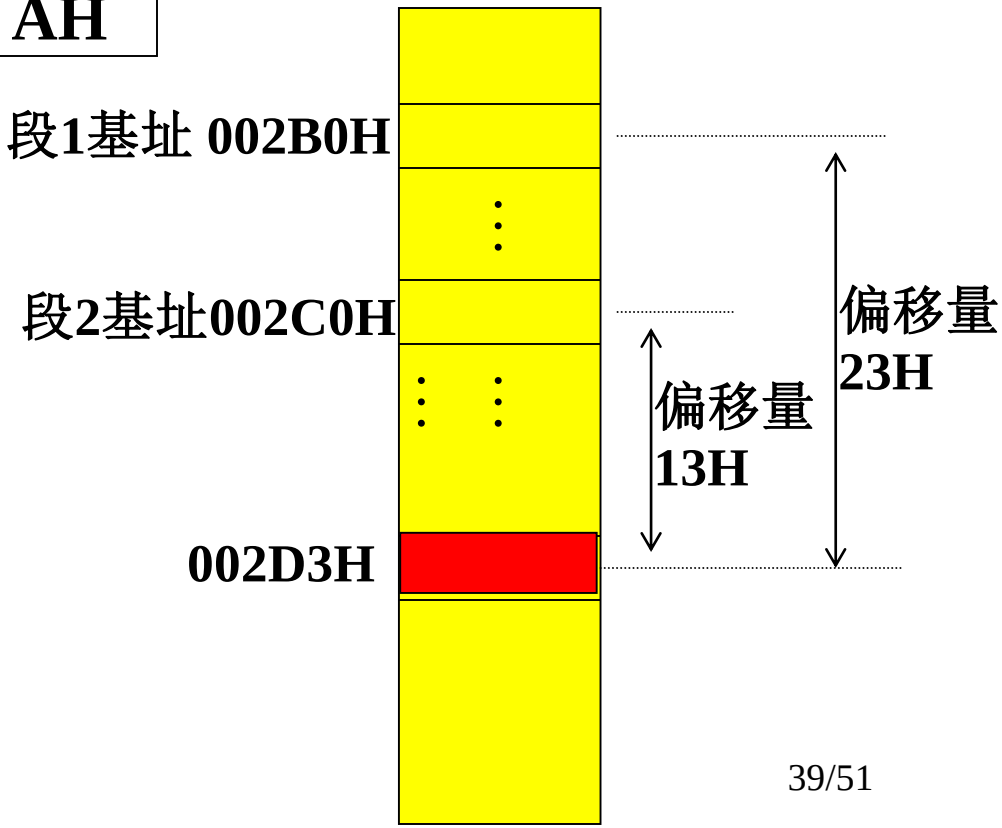
例1：




例2：同一个物理地址002D3H被两个逻辑段中的逻辑地址映射的情况。

002B0H+00023H=002D3H

002C0H+00013H=002D3H



# 4.逻辑地址的来源

在程序的执行过程中，CPU根据不同操作类型访问存储器，其逻辑地址中段基值和偏移量的来源是不一样的。下表是各种操作类型所对应的逻辑地址的来源。

| 序号 | 操作类型 | 逻辑地址 | | |
|---|---|---|---|---|
| | | 段基值 | | 偏移量(OFFSET) |
| | | 隐含来源 | 允许替代来源 | |
| 1 | 取指令 | CS | 无 | IP |
| 2 | 堆栈操作 | SS | 无 | SP |
| 3 | 取源串 | DS | CS，SS，ES | SI |
| 4 | 存目的串 | ES | 无 | DI |
| 5 | 以BP作基址 | SS | CS，DS，ES | 有效地址EA |
| 6 | 存取一般变量 | DS | CS，SS，ES | 有效地址EA |

说明：

（1）允许替代来源也叫做段超越，它表示了段基值除使用隐含的段寄存器外是否可以指定其它段寄存器来提供。

（2）有效地址EA，它表示根据指令所采用的寻址方式（下一章介绍）计算出来的段内偏移量。

# 2.4 堆栈及其操作方法

堆栈是一个特定的存储区，访问该存储区一般需要按照专门的规则进行操作。

**堆栈的用途：**主要用于暂存数据以及在过程调用或处理中断时保存断点信息。

## 一、堆栈的构造

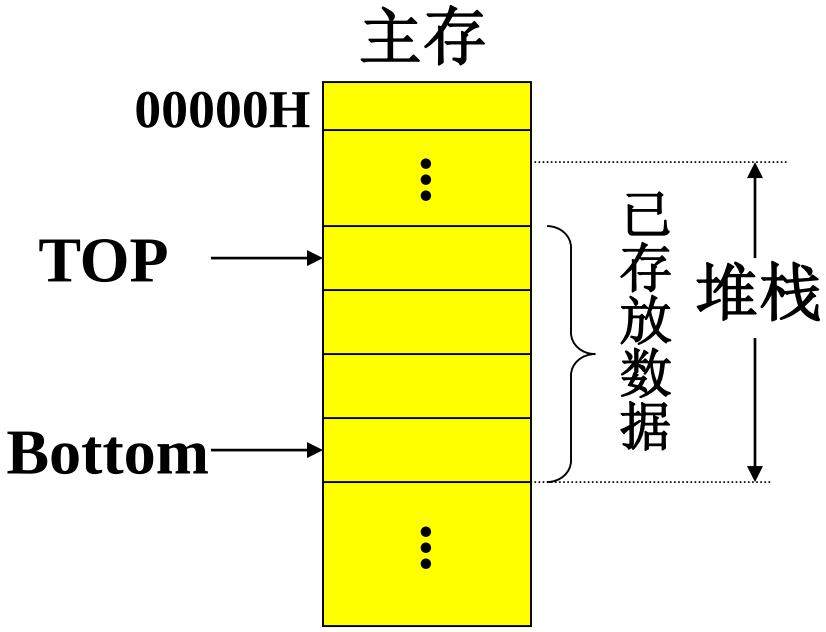
堆栈一般分为：专用堆栈存储器 和 软件堆栈

**专用堆栈存储器** 按堆栈的工作方式专门设计的存储器

**软件堆栈** 由程序设计人员用软件在内存中划出的一块存储区作为堆栈来使用。8086/8088采用这种方式。

堆栈的一端是固定的，称为栈底。栈底是堆栈存储区的最大地址单元。

另一端是浮动的，称为栈顶。在任何时刻，栈顶是最后存入信息的存储单元。栈顶是随着堆栈中存放信息的多少而改变。




为了指示现在堆栈中存放数据位置，通常设置一个寄存器来指示栈顶位置。其内容就象一个指针一样，因此被称为堆栈指针SP（Stack Pointer）。

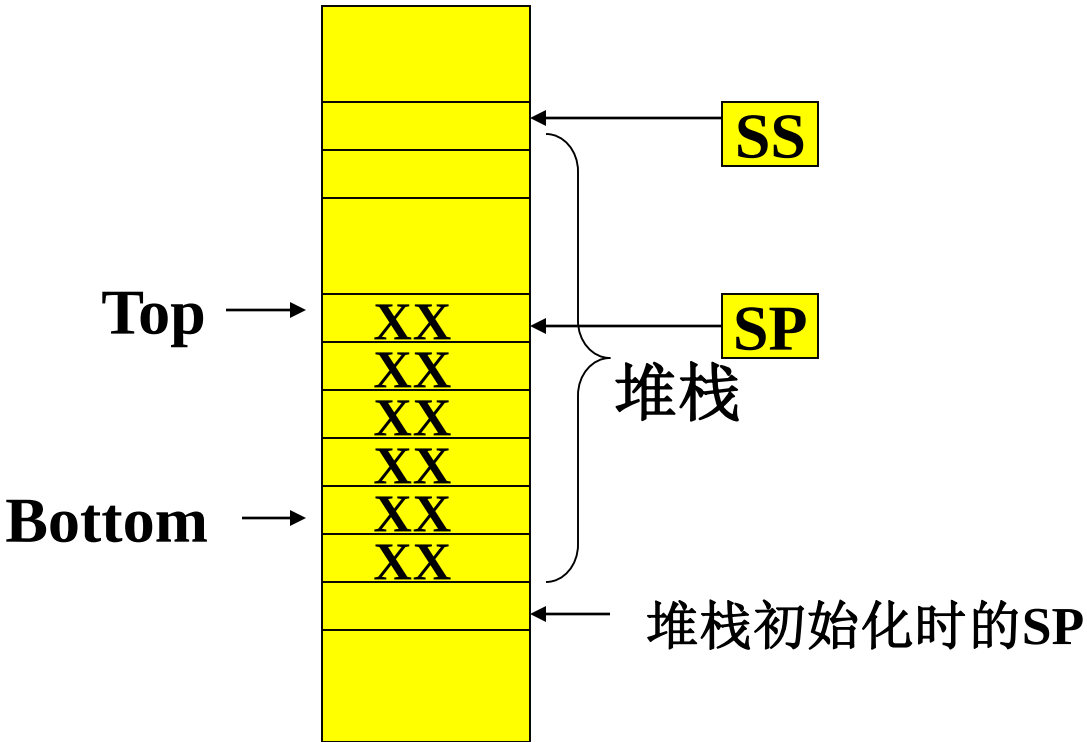
**SP的内容始终指向栈顶单元**

**堆栈中数据进出都由SP来控制**

在堆栈中存取数据的规则是：“先进后出FILO”（First-In Last-Out）。即最先送入堆栈的数据要到最后才能取出，而最后送入堆栈的数据，最先取出。

## 二、8086/8088堆栈的组织

在8086/8088微机中堆栈是由堆栈段寄存器SS指示的一段存储区。



顶由堆栈指针SP指示。SP中内容始终表示堆栈段基址与栈顶之间的距离（字节数）。当SP内容为最大(初始)值时，表示堆栈为空。而当（SP）=0时,表示堆栈全满.

当SP被初始化时，指向栈底+2单元，其值就是堆栈的长度。由于SP是16位寄存器，因此堆栈长度≤ 64K字节。

数据在堆栈中的存放格式是：以字为单位存放，数据的低8位放在较低地址单元，高8位放在较高地址单元。

当用户程序中要求的堆栈长度超过一个堆栈段的最大长度64KB时，可以设置几个堆栈段。

通过改变堆栈段寄存器SS的内容，即可改变到另一个堆栈段，当改变了堆栈段寄存器SS的内容后，必须紧接着赋予SP新值。

# 三、 堆栈操作

## 1.设置堆栈

设置堆栈主要是对堆栈段寄存器SS和堆栈指针SP赋值。

例如：

```
STACK1  SEGMEMT  PARA   STACK
        DB  100   DUP（0）
STACK1   ENDS
```

第一行中的PARA STACK就是用来说明本段为堆栈段。

当程序经过汇编、连接并装入内存时，系统将自动为其分配一个存储区作为堆栈段，将这个段的段基址的高16位送入SS中，将程序指定的字节单元数100赋值给SP。

## 2.进栈PUSH

进栈就是把数据存入堆栈。由指令PUSH或者由机器自动实现，可以将通用寄存器、段寄存器或字存储单元的内容压入堆栈顶部。

例：PUSH AX ;将寄存器AX的内容压入堆栈
PUSH DS ;将段寄存器DS的内容压入堆栈
PUSH DATA-WORD ;将字存储单元DADA-WORD压入
; 堆栈
PUSHF ;将标志寄存器内容压入堆栈。

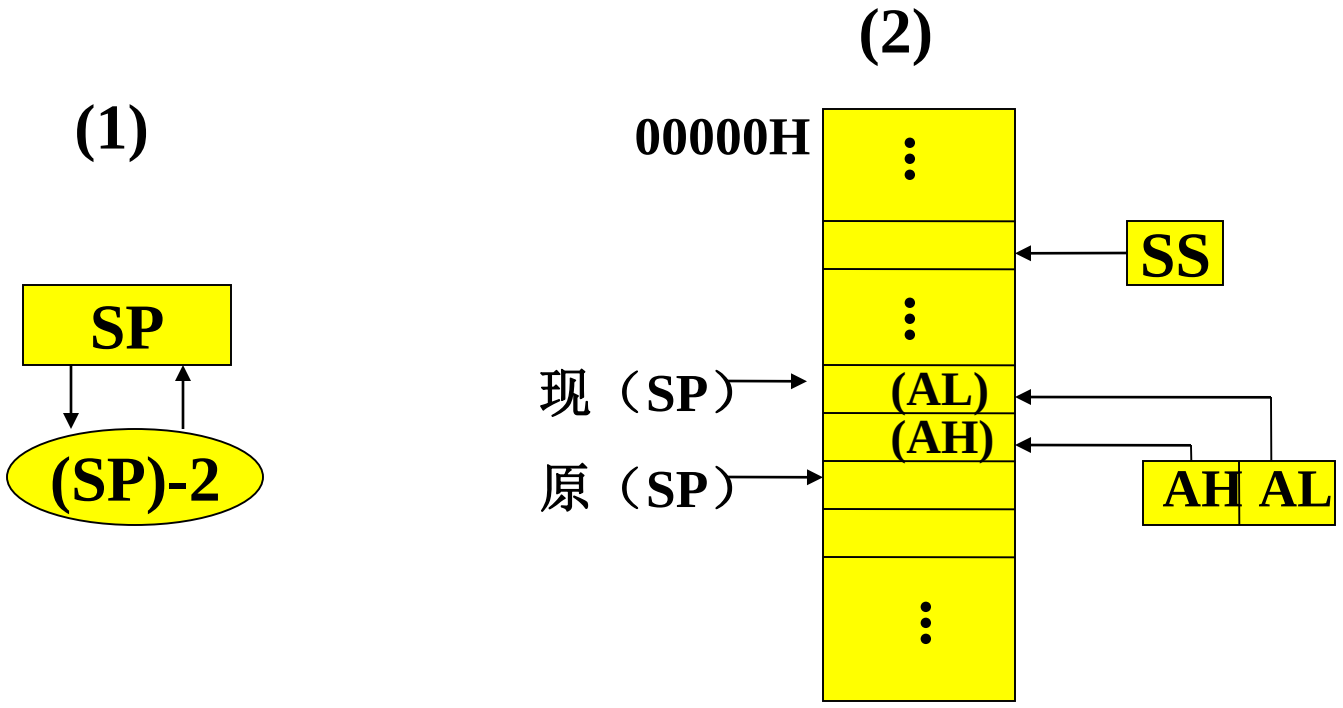
进栈的执行过程：

（1）首先将堆栈指针SP减2，即指向一个空的堆栈字单元

SP<=(SP)-2

（2）将要储存的内容（寄存器或存储单元的内容）送入SP指向的字单元中。（SP）<=数据

例如，指令PUSH AX的执行过程如下图所示：



## 3.出栈POP

出栈操作由POP指令或机器自动实现，它从堆栈顶部弹出一个字到通用寄存器、段寄存器或字存储单元。

例如：POP AX；将栈顶字单元内容弹出到AX
POP DS；将栈顶字单元内容弹出到DS
POP DATA-WORD；将栈顶字单元内容弹出到DATA-WORD存储。
POPF；将栈顶字单元内容送回标志寄存器F。

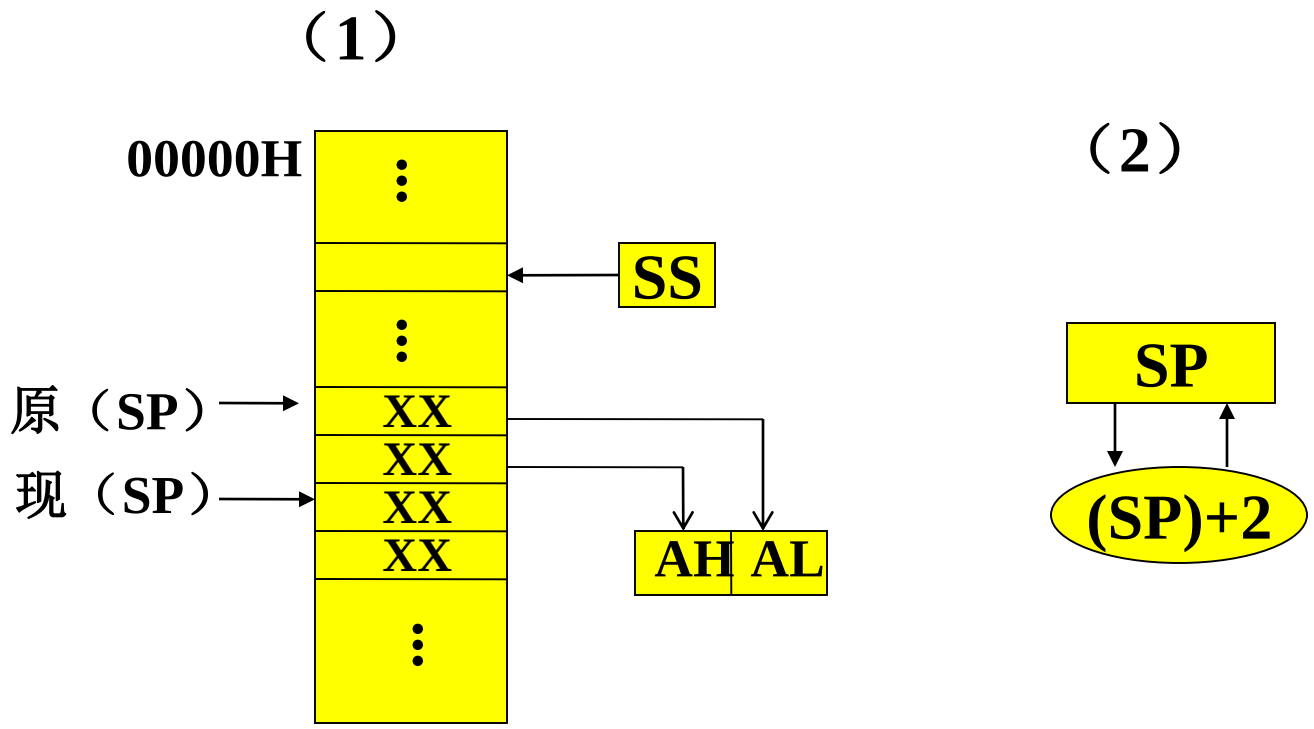
出栈的操作过程：

（1）将SP指向的字单元（即栈顶字单元）内容送往指定的寄存器或存储器。

即：寄存器/存储器<=((SP))

（2）堆栈指针SP内容加2，即：SP<=（SP）+2

例如，指令POP AX的操作过程如下图所示。



# 第三章 寻址方式与指令系统

本章主要内容：

- 8086/8088的各种寻址方式
- 8086/8088的传送类指令
- 8086/8088的基本算术类指令
- 8086/8088移位指令
- 8086/8088逻辑指令
- 8086/8088处理器控制类指令
- 8086/8088指令编码

# 3.1 寻址方式

一条指令通常由两大部分构成：

| 操作码 | 操作数 |
| --- | --- |

操作码：表示该指令应完成的具体操作，如加法、减法、乘法、移位等等。在汇编语言中使用一定的符号来表示，称为助记符。如ADD 、PUSH、POP、MOV等等。

操作数：表示该指令的操作对象。如移位操作的被移位数，加法操作的加数等等。它可以是一个操作数，也可以是多个操作数。这取决于操作码部分的具体需要。

寻址方式:寻找指令中所需操作数的各种方法，也就是提供指令中操作数的存放信息的方式。

**Intel 8086/8088 各指令中提供操作数的方法有以下四种：**

**（1）立即数操作数——操作数在指令代码中提供**

**（2）寄存器操作数——操作数在CPU的通用寄存器或段寄存器中**

**（3）存储器操作数——操作数在内存的存储单元中**

**（4）I/O端口操作数——操作数在输入/输出接口的寄存器中**

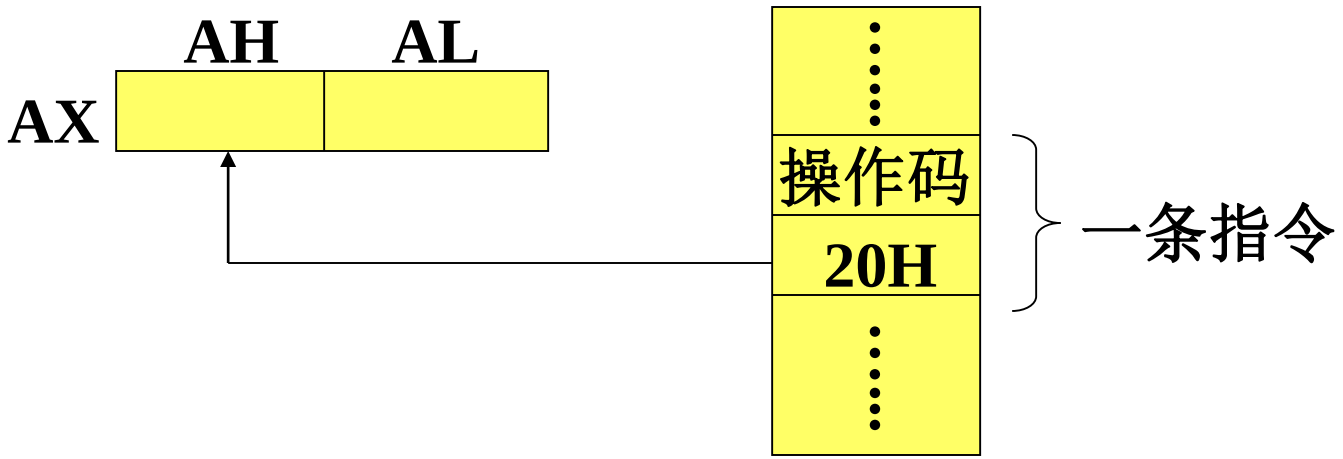
## 1.立即数寻址

立即数寻址方式的指令中，所需操作数直接包含在指令代码中，这种操作数称为立即数。

立即数可以是8位，也可以是16位。

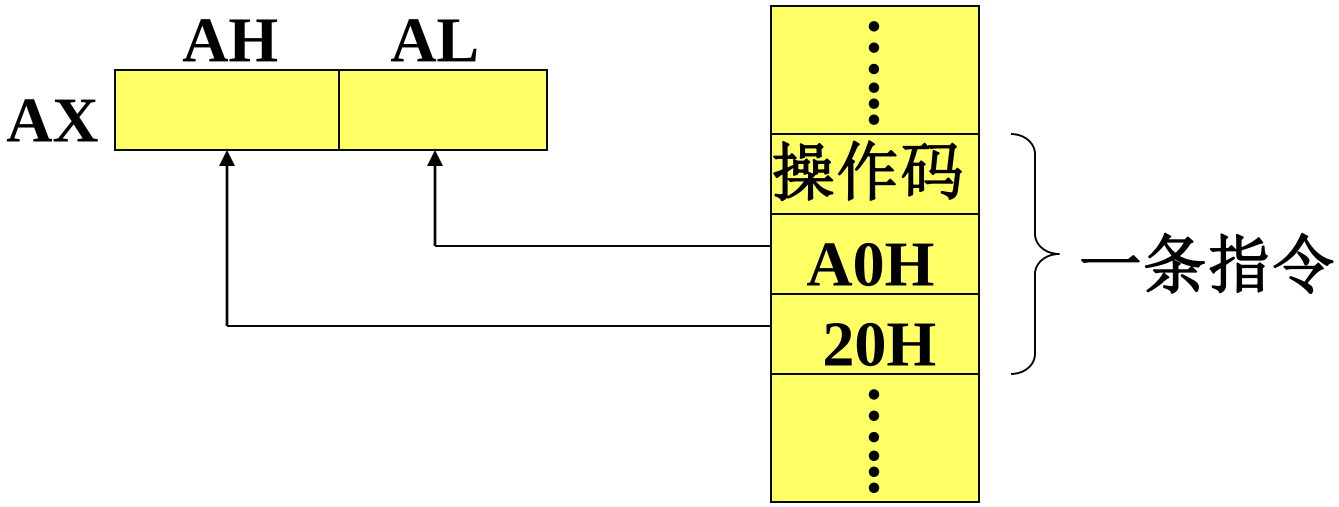
例：MOV AH，20H

它表示将8位立即数20H送入AH中



例：MOV AX，20A0H

它表示将16位立即数20A0H送入AX中




由于在指令执行过程中，立即数作为指令的一部分直接从BIU的指令队列中取出，它不需另外占用总线周期，因此这种寻址方式执行速度快。

注意：立即数只能作为源操作数，而不能作为目的操作数。

## 2.寄存器寻址

寄存器寻址方式是指指令中所需的操作数在CPU的某个寄存器中。寄存器可以是8位或16位通用寄存器，或者是段寄存器。如：AH、AL、AX、CX、DS、ES等。

例如：MOV AX，BX
MOV DS，AX

由于存取寄存器操作数完全在CPU内部进行，不需要总线周期，所以执行速度很快。

后面介绍的几种寻址方式其操作数都是在存储器中，它们的主要区别就是操作数在内存中存放地址的形成方法不同。

一个存储单元逻辑地址表示形式：段基值：偏移量

段基值由某个段寄存器提供.

偏移量表示了该存储单元与段起始地址之间的距离，也叫做有效地址EA。

有效地址EA是以下三个地址分量的几种组合，由CPU的执行单元EU计算出来的。

(1)位移量：位移量是指令中直接给出的一个8位或16位数。一般源程序中以操作数名字(变量名或标号)的形式出现。

(2)基址：由基址寄存器BX或基址指针BP提供的内容。

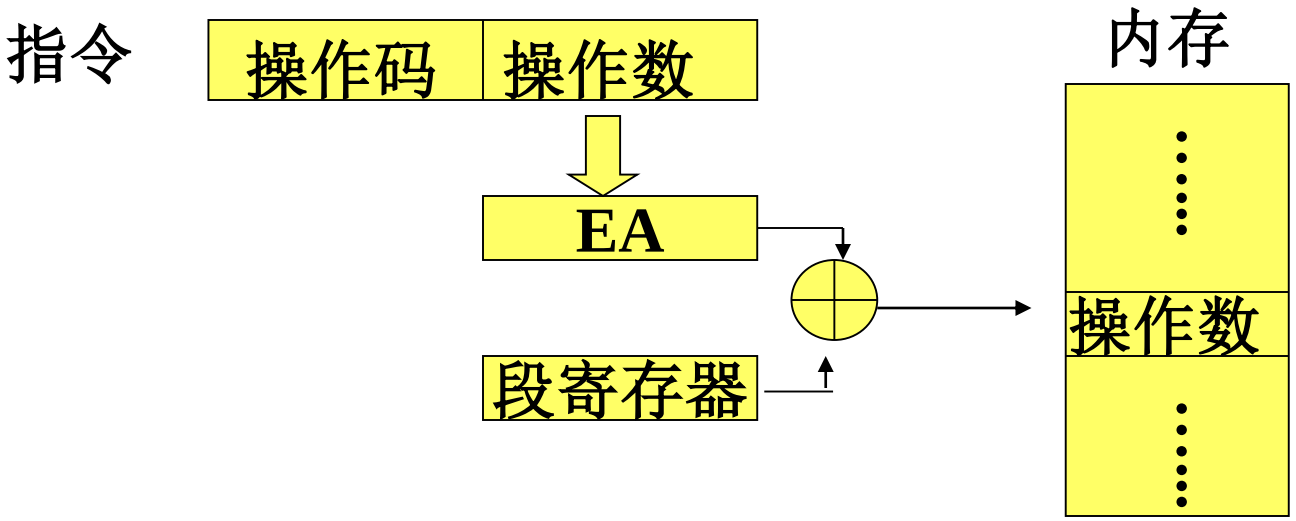
(3)变址：由源变址寄存器SI或目的变址寄存器DI提供的内容

位移量、基址和变址三个地址分量组合时，若有两个或两个以上分量时，将进行以$2^{16}$为模的十六位加法运算。

下面是由这三个地址分量的不同组合所形成的四种寻址方式。

## 3.直接寻址

在直接寻址方式的指令中，操作数的有效地址EA只有位移量地址分量。



在汇编语言源程序中，直接寻址方式用符号或常数来表示。

**（1）用符号表示**

例：MOV BX，VAR =>MOV BX，DS：VAR

它表示将数据段中，偏移了VAR个字节距离的字单元内容送到寄存器BX中。

MOV AL，DATA+2 => MOV AL, DS: DATA+2

它表示将数据段偏移了DATA+2的字节单元内容送入AL中。

**（2）用常数表示**

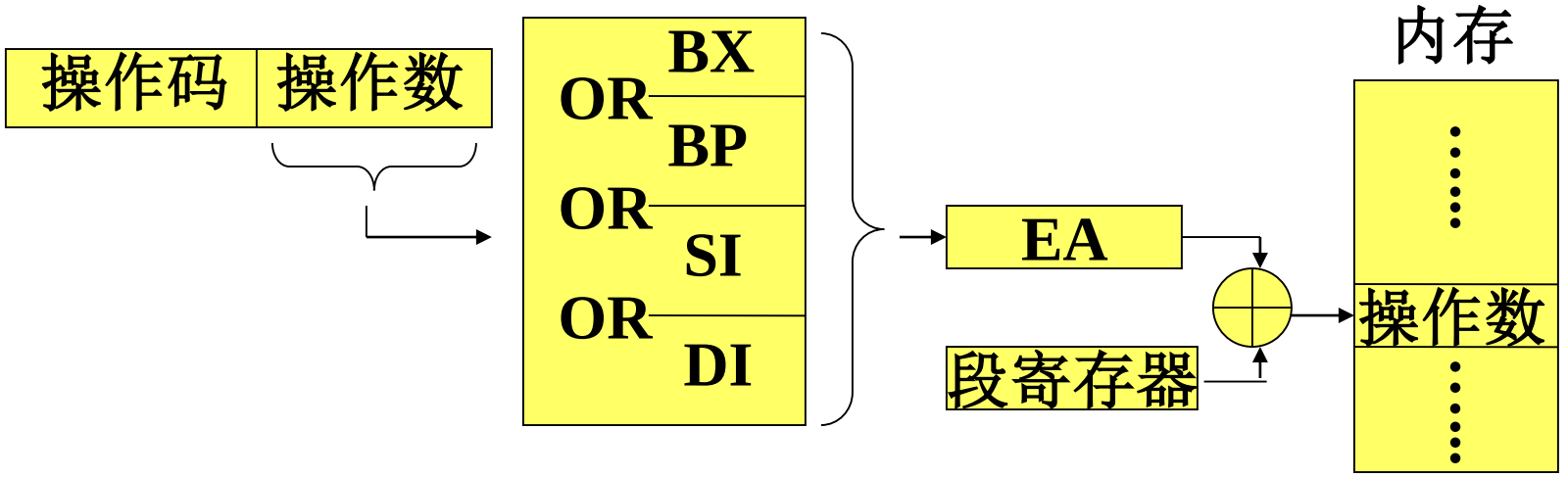
例：MOV AX，DS：[64H]

它表示从当前数据段开始，偏移100个字节的字单元内容送到AX中。不能写为：MOV AX，64H

注意：用常数表示时，必须用方括号括起来。段寄存器不能省略。

## 4.寄存器间接寻址

操作数有效地址EA直接从基址寄存器（BX或BP）或变址寄存器（SI或DI）中获得。




寄存器间接寻址就是事先将偏移量存放在某个寄存器(BX、BP、SI或DI)中，这些寄存器就如同一个地址指针。

在程序运行期间，只要对寄存器内容进行修改，就可以实现用同一条指令实现对不同存储单元进行操作。

指示存储器所在段的段寄存器可以省略，当指令中使用的是**BP寄存器**，则隐含表示使用**SS段寄存器**，其余情况则隐含使用**DS段寄存器**。

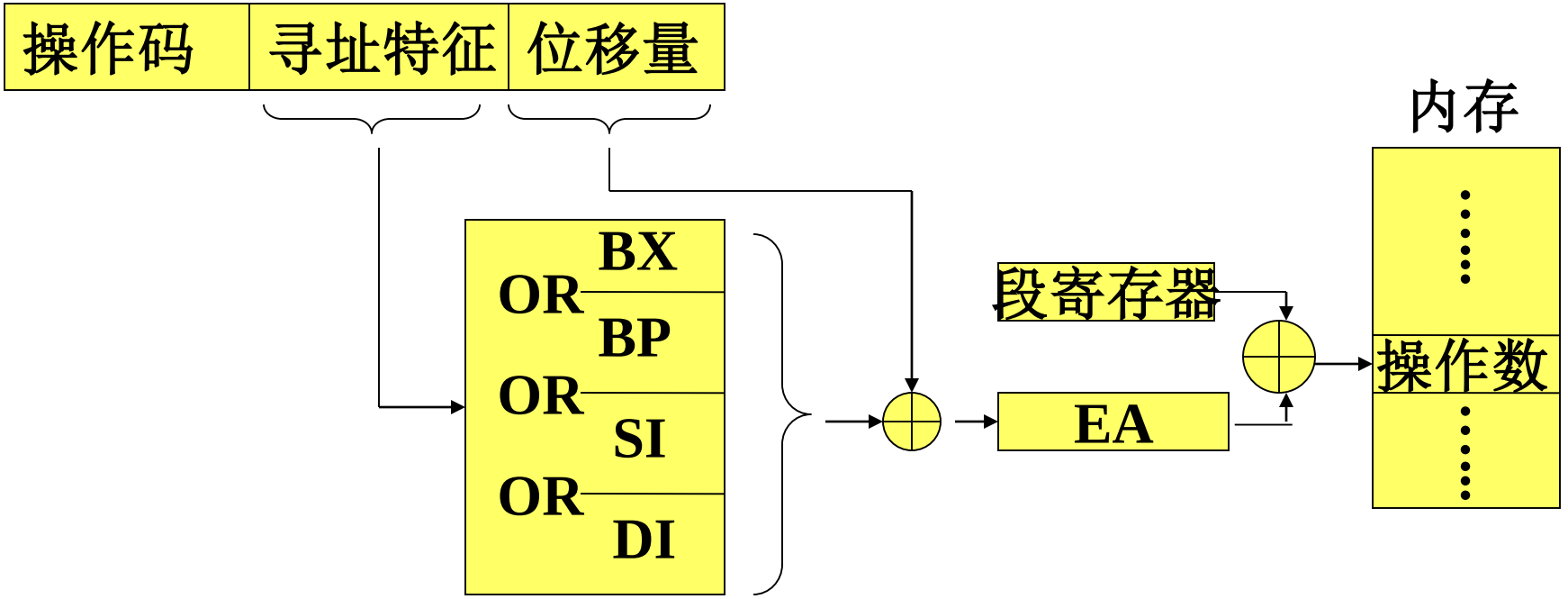
例如：MOV AX，[BX] => MOV AX，DS：[BX]

MOV BH，[BP] => MOV BH，SS：[BP]

MOV [DI]，BX => MOV DS：[DI]，BX

## 5.基址寻址/变址寻址

操作数的有效地址EA等于基址分量或变址分量加上指令中给出的位移量。




指令中使用BX或BP时为基址寻址。指令中使用SI或DI时为变址寻址。

段寄存器的隐含使用规则与寄存器间接寻址方式相同

例：MOV AX，10H [SI] => MOV AX，DS：10H [SI]

MOV TABLE [DI]，AL =>MOV DS：TABLE [DI]，AL

注意：当位移量为常数时，不能加方括号。

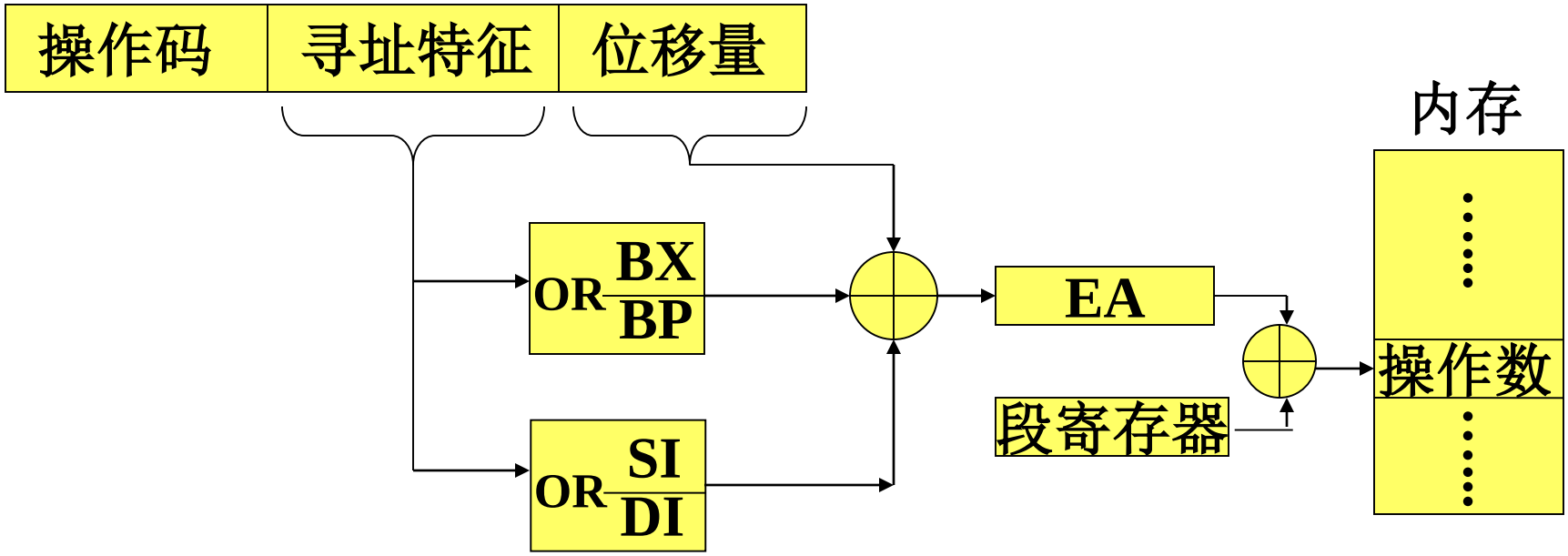
这两种寻址方式只需通过改变寄存器的内容就可用一条指令访问不同的存储单元，并且由于增加了一个位移量分量，因此它们能够很方便地访问数组和表格数据。

由于这两种寻址方式中寄存器中的内容是相对于由位移量指定的初始单元。因此也叫寄存器相对寻址。

# 6.基址变址寻址

操作数的有效地址是三个地址分量之和，即：

**EA=基址+变址+位移量**




当基址选用BX时隐含使用段寄存器DS，而选用BP时则隐含使用段寄存器SS。

例如：MOV CX，100H[BX][DI]

MOV TABLE[BX][DI]，AX

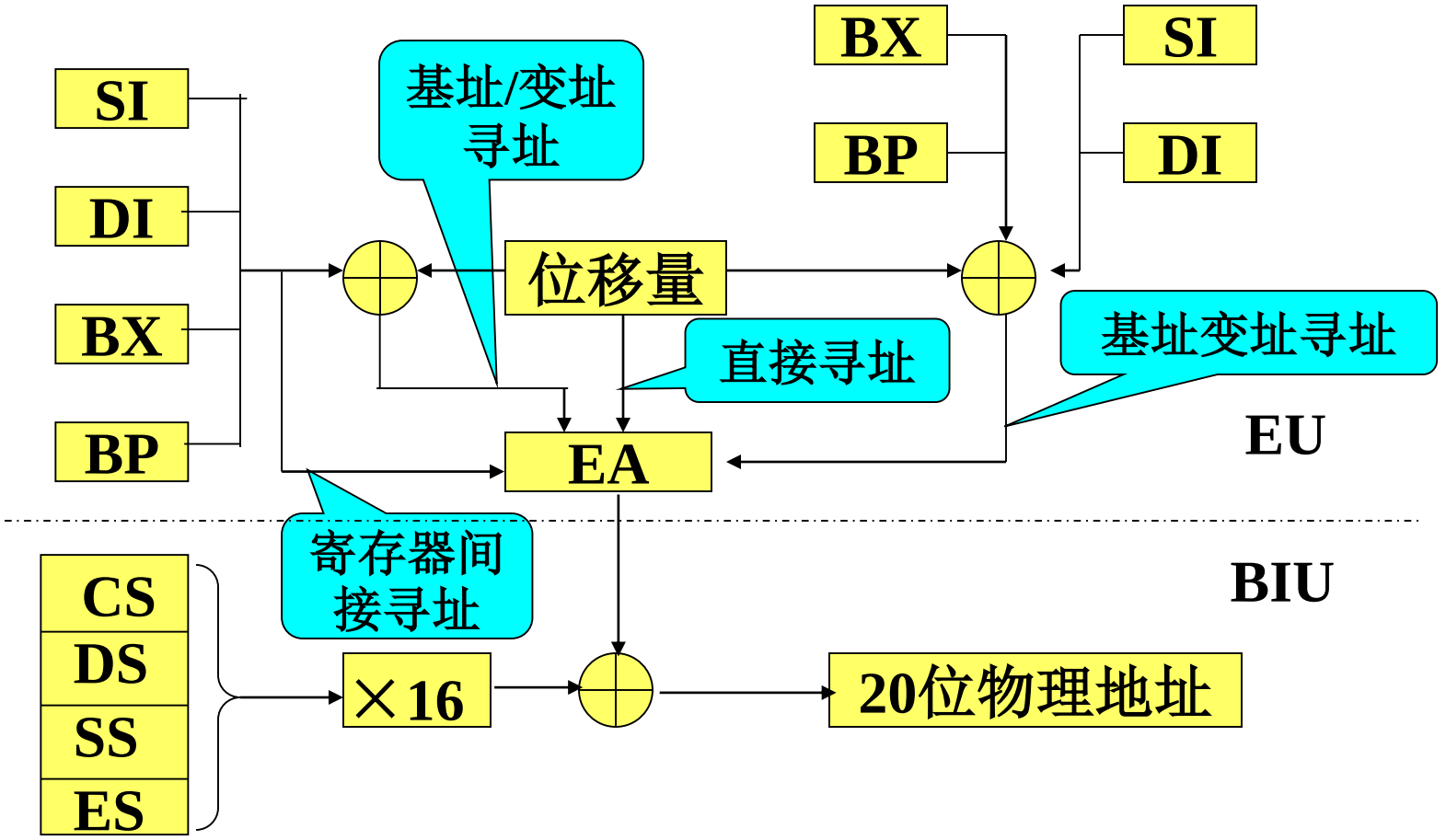
下面的用法是错误的。

MOV AX，ARRAY[BX][BP]

MOV AX，TABLE[SI][DI]

在基址变址寻址方式中，程序运行期间有两个地址分量可以修改。因此它是最灵活的一种寻址方式，可以方便地对二维数组进行访问。

# 存储器操作数寻址方式中地址形成小结



## 7.串操作寻址方式

**8086/8088**设置有专门用于串操作的指令，这些指令的操作数虽然也在存储器中，但它们不使用前面介绍的各种寻址方式，而隐含地使用变址寄存器**SI**和**DI**专门指示。

- 在寻找源操作数时，隐含使用**SI**作为地址指针。
- 在寻找目的串时，隐含使用**DI**作为地址指针。
- 在串操作完成之后，自动对**SI**和**DI**进行修改，使它们指向下一个操作数。

## 8. I/O端口寻址

在计算机系统，对I/O端口的寻址方式有以下两种方法。

### ➢ 存储器编址方法

将I/O端口视为存储器的一个单元，对端口的访问就如同访问存储单元一样。访问存储器的指令和各种寻址方式同样适用对I/O端口的访问。

特点: 程序设计灵活，但需要占用存储地址空间。

### ➢ I/O端口编址方法

I/O端口的地址与存储器地址分开，并使用专门的输入指令和输出指令。

8086/8088系统中就是采用的这种方式。可以最多访问64K个字节端口或32K个字端口，用专门的IN指令和OUT指令访问。寻址方式有如下两种。

## （1） 直接端口寻址

在指令中直接给出端口地址，端口地址一般采用2位十六进制数，也可以用符号表示。

直接端口寻址可访问的端口数为0～255个。

例如：IN AL，25H

## （2）寄存器间接端口寻址

寄存器间接端口寻址：把I/O端口的地址先送到DX中，用DX作间接寻址寄存器。

例如：MOV DX，378H
OUT DX，AL

如果访问的端口地址值大于255，则必须用I/O端口的间接寻址方式。

# 3.2 指令系统

一种计算机所能执行的各种类型的指令的集合称为该计算机的指令系统。

**Intel8086/8088CPU指令系统的指令可以分为六大类：**

1.传送类指令　　2.算术运算类指令　　3.位操作类指令

4.串操作类指令　　5.程序转移类指令　　6.处理器控制类指令

从指令的格式划分，一般可以分为三种：

1.双操作数指令：OPR DEST SRC
2.单操作数指令：OPR DEST
3.无操作数指令：OPR

对于无操作数指令，包含两种情况：

（1）指令不需要操作数，如暂停指令HLT。

（2）在指令格式中，没有显式地指明操作数，但是它隐含指明了操作数的存放地方，如指令PUSHF。

## 一、传送类指令

传送类指令的作用是将数据信息或地址信息传送到一个寄存器或存储单元中，可以分为以下四种情况。

### 1.通用数据传送指令

指令格式：MOV DEST，SRC

作用：将源操作数指定的内容传送到目的操作数，即DEST<=(SRC)。

当指令执行完后，目的操作数原有的内容被源操作数内容覆盖，即目的操作数和源操作数具有相同内容。

**MOV指令对标志寄存器的各位无影响**

MOV指令可以是*字节*数据传送也可以是*字*数据传送，但是源操作数和目的操作数的长度必须一致。

MOV指令可以分为以下几种情况：

（1）立即数传送到通用寄存器或存储单元

例：MOV AH，10H
MOV AX，2345H
MOV M-BYTE, 64H
MOV M-WORD，2364H

注意：立即数只能作为源操作数，立即数不能传送给段寄存器。

## （2）寄存器之间的传送

例：MOV AH，CH
MOV DS, AX
MOV ES, BX
MOV AX，CS
MOV CS, AX;错误

注意：段寄存器CS只能作源操作数，不能作目的操作数。

## （3）寄存器与存储单元之间传送

例：MOV AL，[SI]
MOV [DI]，AH
MOV AX, 10[BX]
MOV TABLE[BP], BX
MOV DS, [SI][BX]

MOV [BX], [BP][SI];错误

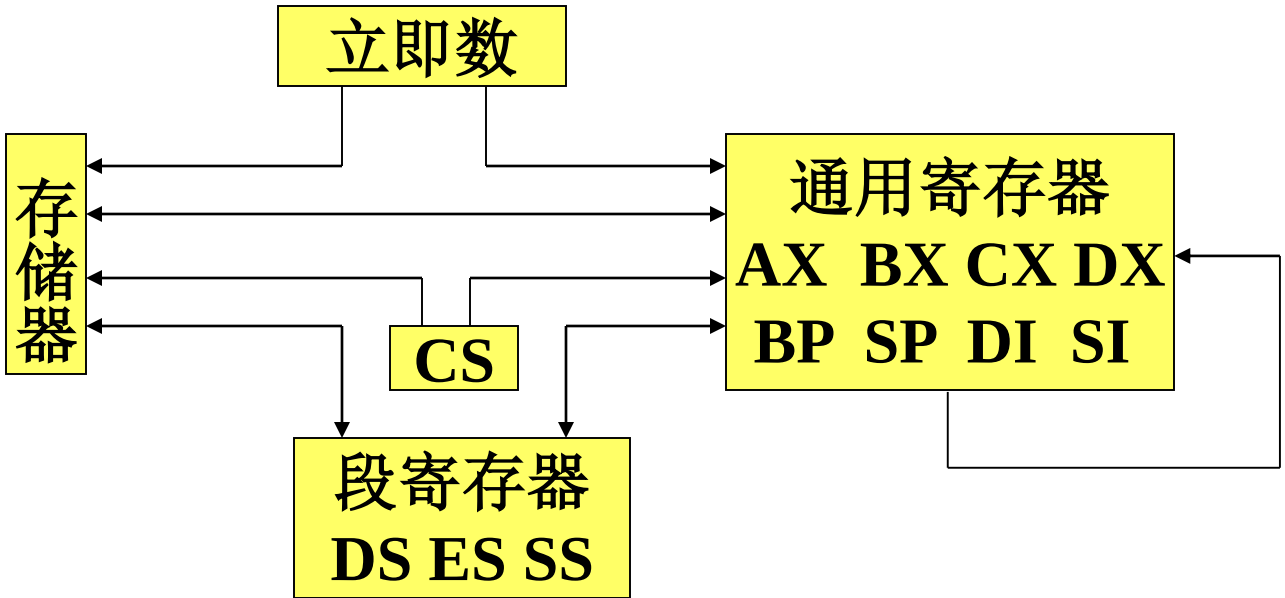
综合起来，MOV指令在使用时需注意以下几个问题：

（1）立即数只能作源操作数，且它不能传送给段寄存器。

（2）段寄存器CS只能作源操作数，段寄存器之间不能直接传送。

（3）存储单元之间不能直接传送数据

（4）MOV指令不影响标志位



# 2.交换指令

指令格式：XCHG DEST，SRC

作用：源操作数和目的操作数两者内容相互交换，即：（DEST）<=>(SRC)。

指令对标志寄存器各位无影响

数据交换可以在寄存器之间或寄存器与存储器单元之间进行。但是不能在存储单元之间直接进行数据交换。寄存器只能使用通用寄存器。

例 XCHG AX，BX
XCHG AH，CH

为了完成两个存储单元(DA_BYTE1和DA_BYTE2)之间的数据交换可以使用以下三条指令来实现。

```
MOV  AL，DA-BYTE1；     AL<=(DA_BYTE1)
XCHG  AL，DA-BYTE2 ；  (AL)<=>(DA-BYTE2
XCHG  AL，DA-BYTE1 ；  (AL)<=>(DA-BYTE1)
  或MOV DA-BYTE1，AL;(DA_BYTE1)<=(AL)
```

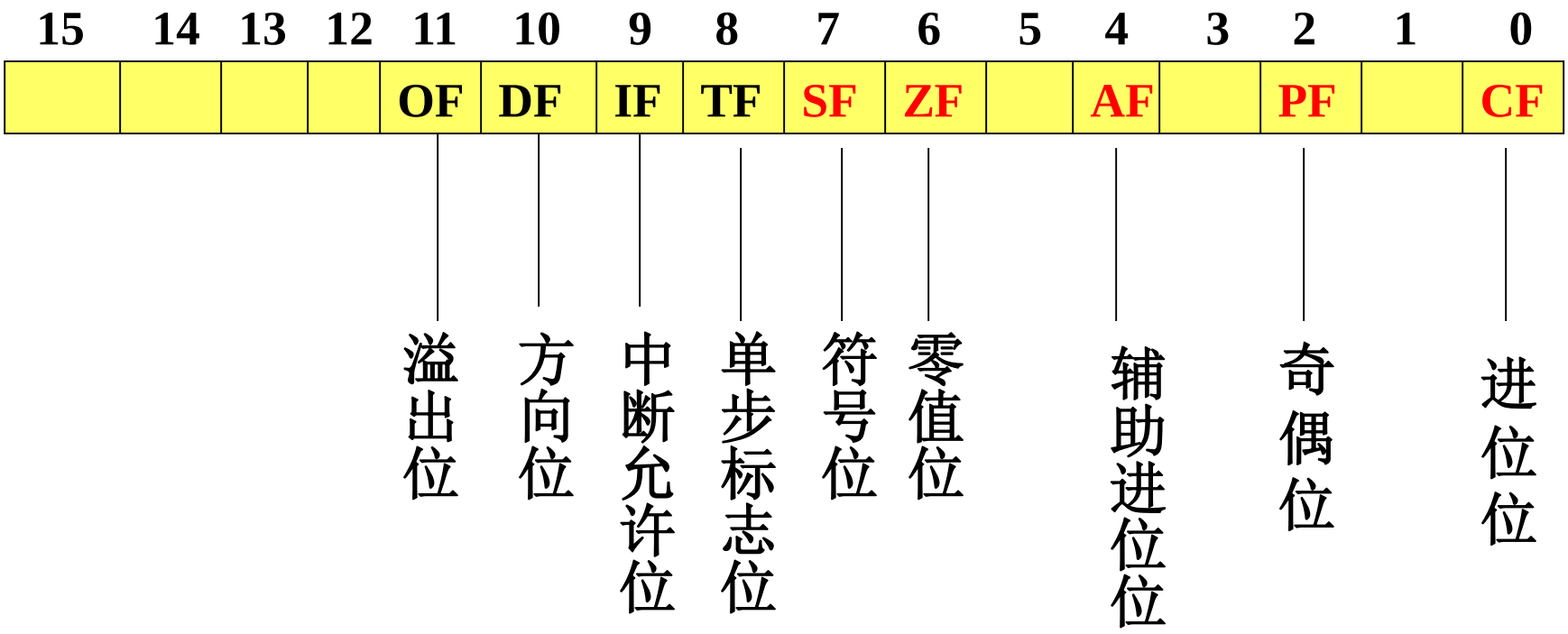
## 3.标志传送指令

对标志寄存器进行存取的指令有4条，它们都是无操作数指令，即指令隐含指定标志寄存器、AH寄存器或堆栈为操作数。

## （1）取标志寄存器指令

**指令格式：LAHF**

**作用：将标志寄存器的低8位送入AH寄存器，即将标志SF、ZF、AF、PF和CF分别送入AH的第7、6、4、2、0位，而AH的第5、3、1位不确定。**




**指令执行对标志寄存器各位无影响，即标志寄存器各位不变.**

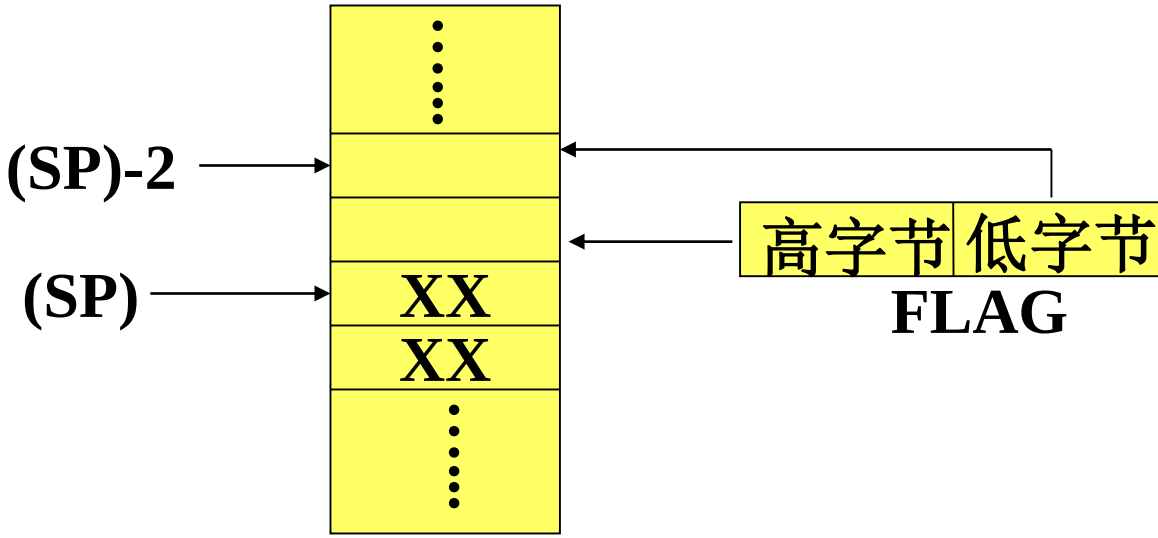
## （2）存储标志寄存器指令

指令格式：SAHF

作用：将寄存器AH中的第7、6、4、2、0位分别送入标志寄存器的SF、ZF、AF、PF和CF各标志位。而标志寄存器高8位中的各标志位不受影响。

## （3）标志进栈指令

指令格式：PUSHF




作用：先将堆栈指针SP减2，使其指向堆栈顶部的空字单元，然后将16位标志寄存器的内容送SP指向的字单元。

### （4）标志出栈指令

**指令格式：POPF**

**作用：将由SP指向的堆栈顶部的一个字单元的内容送入标志寄存器，然后SP的内容加2.**

## 4.地址传送指令

这类指令有3条，它们的作用是将存储单元的地址送寄存器。

### （1）装入有效地址

**格式：LEA DEST，SRC**

其中：源操作数SRC必须是一个字节或字存储器操作数（地址），DEST必须是一个16位通用寄存器。

作用：将SRC存储单元地址中的偏移量，即有效地址EA传送到一个16位通用寄存器中。

指令执行对标志寄存器各位无影响。

例1：LEA AX，[BX] [SI]

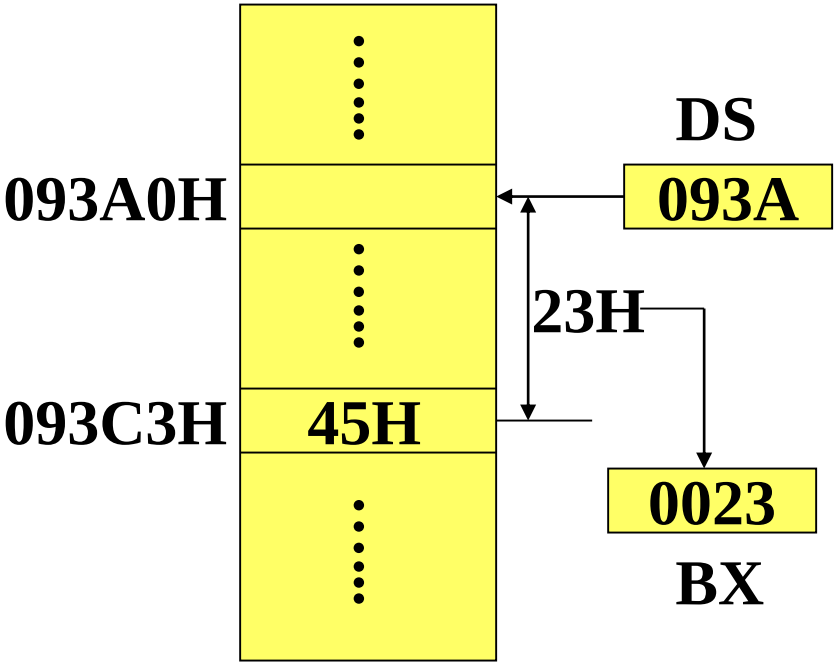
源操作数使用的是基址变址寻址方式,它所形成的有效地址就是BX的内容加上SI的内容。即

AX<=(BX)+(SI)

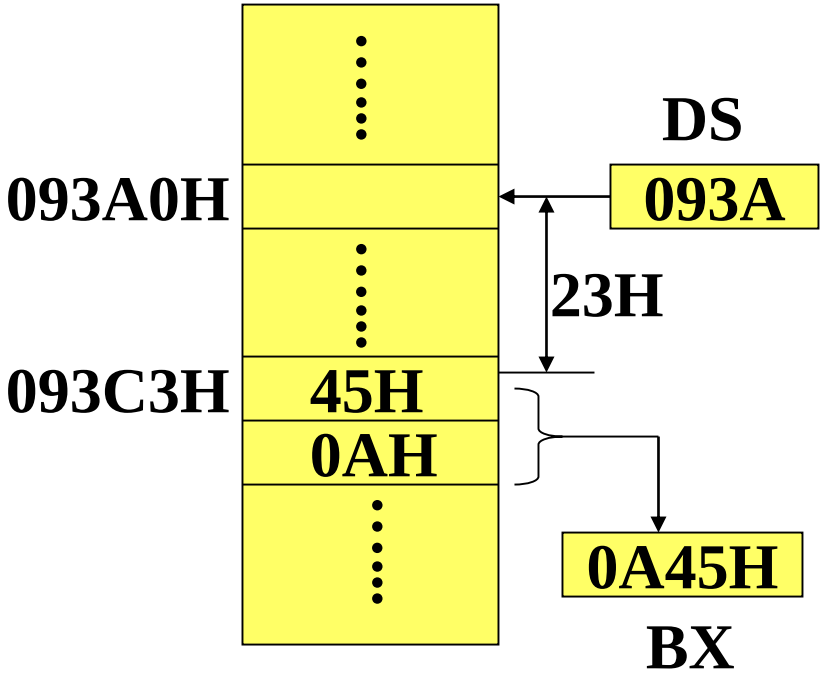
注意：它不是将BX和SI所寻址的存储单元的内容送入AX。

例2 比较指令 LEA BX，DS:[23H]与 MOV BX，DS:[23H]的功能

LEA BX，DS:[23H]




MOV BX, DS:[23H]



## （2）装入地址指针指令

格式：LDS DEST，SRC
LES DEST，SRC

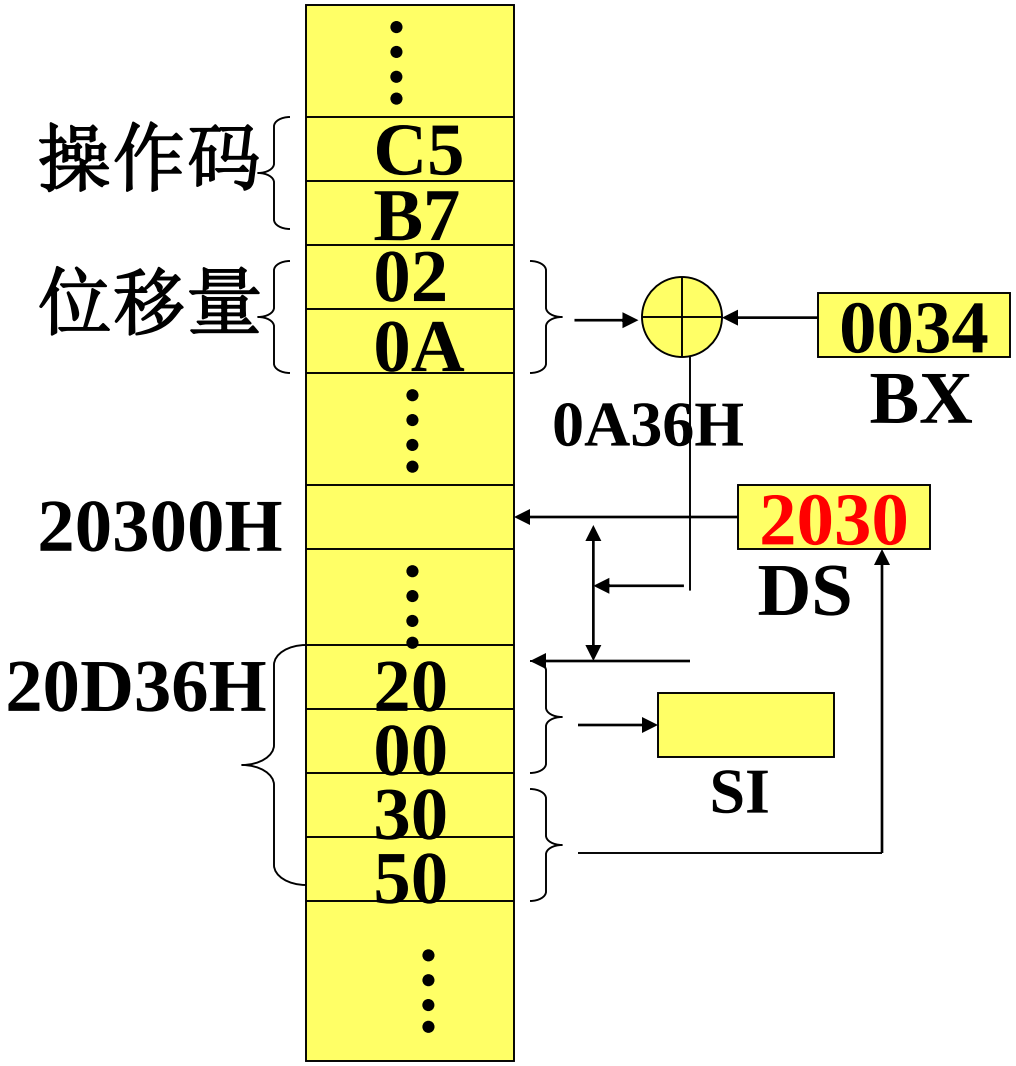
其中:DEST是任意一个16位通用寄存器。SRC必须是一个存储器操作数。

作用：把SRC存储单元开始的4个字节单元的内容(32位地址指针)送入DEST通用寄存器和段寄存器DS（LDS指令）或ES（LES指令），其中低字单元内容为偏移量送通用寄存器，高字单元内容为段基值送DS或ES。

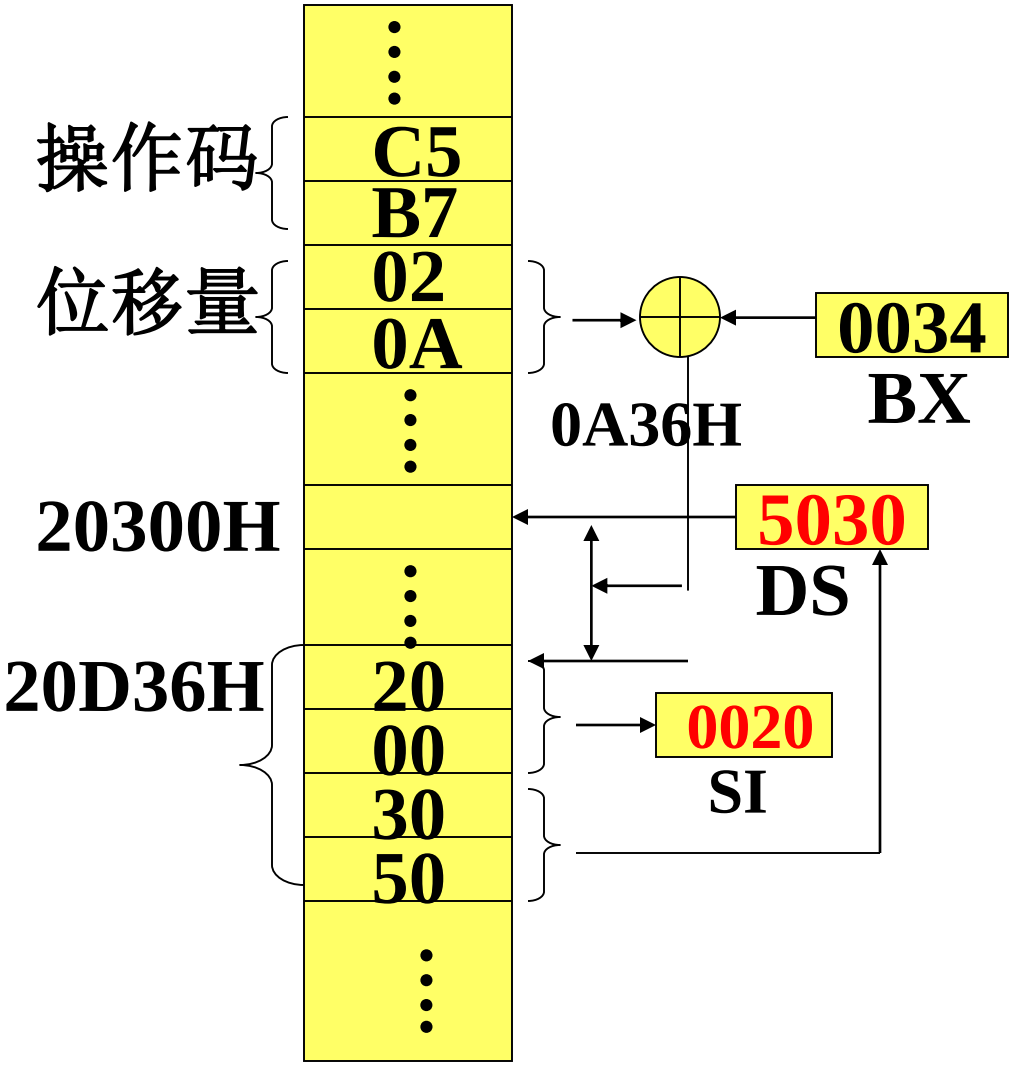
# 例：LDS SI，TABLE[BX]

**设 TABLE的值为0A02H, (BX)=34H,(DS)=2030H**

指令执行前



指令执行后


操作码
C5
B7
位移量
02
0A
0034
BX
0A36H
20300H
5030
DS
20D36H
20
00
30
50
0020
SI

# 二、算术运算类指令

**8086/8088**指令系统中有加、减、乘、除指令，这些指令可以对字节数据或字数据进行运算。

参加运算的数可以是无符号数，也可以是带符号数。带符号数用补码表示。

参加运算的数可以是二进制数，也可以是十进制数（以**BCD**码表示）。

在本节中只讨论基本的加法和减法指令，其它指令在以后的章节中介绍.

# 1.加法指令

**指令格式：ADD DEST，SRC**

**功能：目的操作数和源操作数相加，其和存放到目的操作数中，而源操作数内容保持不变，即**

**DEST<=(DEST)+(SRC)。**

**根据相加的结果将影响到标志寄存器的CF、PF、AF、ZF、SF和OF。**

**DEST只能是通用寄存器或存储器操作数。不能是立即数。**

**SRC可以是通用寄存器、存储器或立即数操作数**

**DEST和SRC不能都为存储器操作数。**

**ADD指令可以是字节操作数相加，也可以是字操作数相加。**

例 分析下列各指令功能

**(1) ADD AX,CX**

功能：将寄存器AX的内容与CX的内容相加，结果传送到AX中

**(2) ADD AH,DATA_BYTE**

功能：将由直接寻址方式所指示的存储单元的内容与AH内容相加，结果送回AH中。

**(3) ADD CX,10H**

功能：将常数10H加入到CX中。为字操作数指令.

**(4) ADD AX, [BX][SI]**

功能：将由基址变址寻址方式所指示的存储单元的内容加入到AX中。

# 2.带进位加法指令

**指令格式：ADC DEST，SRC**

该指令的功能与ADD基本相同，所不同的是其结果还要加上进位标志CF的值，即：

**DEST<=(DEST)+(SRC)+CF**

**根据相加的结果设置标志寄存器中的CF、PF、AF、ZF、SF和OF**

注意：参加运算的进位CF是本条指令执行之前的值。

用ADC指令可实现数据长度大于16位的两数相加

例：计算12349678H＋377425H

```
MOV  AX, 1234H
MOV  BX，9678H
ADD  BX，7425H
ADC  AX，37H
```

指令执行后，结果的高16位在AX，低16位在BX中。

## 3.加1指令

指令格式：INC  DEST

该指令为单操作数指令，其功能是将目的操作数加1，并送回到目的操作数，即：

DEST<=(DEST)+1

目的操作数可以是任意的8位、16位通用寄存器或存储器操作数。目的操作数被视为带符号二进制数

根据指令执行结果设置PF、AF、ZF、SF和OF标志，但不影响CF。INC指令主要用于某些计数器的计数和修改地址指针。

例：INC CL
INC SI
INC COUNT

# 4.减法指令

**指令格式：SUB DEST，SRC**

**功能:目的操作数的内容减去源操作数的内容，结果送入目的操作数，源操作数中内容保持不变。**

**即：DEST<=（DEST）－（SRC）**

**操作结果将影响标志位CF、PF、AF、ZF、SF和OF。**

**目的操作数DEST和源操作数SRC可以是8位或16位的通用寄存器、存储器操作数，但两者不能同时为存储器操作数。立即数只能作源操作数。**

例：**SUB AX，BX**

**SUB AH，10H**

**SUB DX,DA-WORD**

**SUB DA-BYTE,BL**

注意：减法指令对借位标志的影响，若采用变减为加的运算方法，则产生的进位与CF标志结果相反。

# 5.带借位减法

**指令格式：SBB DEST，SRC**

该指令的功能与SUB指令基本相同，不同的是在两个操作数相减后再减去进位标志CF的值。

即：DEST<=(DEST)－(SRC)－CF。

注意：该CF的值是本条指令执行前的结果。

SBB指令在使用上与ADC类似，主要用于长度大于16位的数相减，即将低16位相减的结果引入高位部分的减法中。

根据指令执行结果设置PF、AF、ZF、SF 、OF和CF。

# 6.减1指令

**指令格式：DEC DEST**

该指令为单操作数指令，将目的操作数的内容减1后，送回到目的操作数。即：DEST<=(DEST)－1

DEST可以是8位或16位的通用寄存器存储器操作数，该指令将DEST看作是带符号二进制数。

根据指令执行结果设置PF、AF、ZF、SF和OF，但不影响CF。

**DEC指令的使用类似INC指令。主要用于计数和修改地址指针，计数方向与INC指令相反。**

```
例          MOV  AL，10H
    LOP：  DEC   AL
           JNC    LOP
```

**上述程序段中，是一个错误应用DEC指令的例子。**

# 7.求负数指令

**指令格式：NEG DEST**

**功能：用零减去目的操作数的内容，并送回目的操作数，即：DEST<=0-(DEST)**

**DEST可以是任意一个8位或16位的通用寄存器或存储器操作数，被视为带符号的操作数。**

**由于机器中带符号数用补码表示，求操作数的负数就是求补操作。因此，NEG指令也叫取补指令。**

**NEG指令将影响标志PF、AF、ZF、SF、CF和OF。**

**对进位标志CF的影响：**

只有当操作数为零时，进位标志CF被置零，其它情况都被置1.

**对溢出标志OF的影响：**

当字节操作数为-128，或字操作数为-32768时，执行NEG指令的结果操作数将无变化，但溢出标志OF被置1.

例1　设AL中存放一个正数(AL)=25H，BL中存放一个负数：(BL)=－58H，求它们的相反数，即负数。

```
NEG   AL
NEG   BL
```

指令执行后，(AL)=-25H=11011011B
(BL)= 58H=01011000B

例2 一个32位带符号数存放在DAW开始的四个字节存储单元中，DAW字节单元存放最低字节。求该数的负数，并存入原存储单元中。

```
NEG   WORD PTR DAW
MOV  AX，0
SBB   AX，DAW+2
MOV  DAW+2，AX
```

结果的低16位由指令NEG直接得到，而高16位还要考虑低16位产生的借位，因此使用了带借位的指令SBB。

# 三、位操作类指令

## 1.逻辑运算指令

逻辑运算指令共有4条，它们的指令格式分别是：

| | |
|---|---|
| 逻辑“与”指令 | AND DEST，SRC |
| 逻辑“或”指令 | OR DEST，SRC |
| 逻辑“异或”指令 | XOR DEST，SRC |
| 逻辑“非”指令 | NOT DEST |

这4条指令都是执行按位逻辑运算，如下表所示：

| DEST | SRC | AND | OR | XOR | NOT |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

DEST和SRC可以是8位或16位的通用寄存器或存储器操作数，但两者不能同时为存储器操作数，SRC可以为立即数。

## 逻辑指令对标志位的影响：

NOT指令对标志无影响。而其余三条指令将根据结果影响SF、ZF和PF，而CF和OF总 是置0，AF为不确定。

逻辑运算指令除用来实现各种逻辑运算之外，还常用于对字节或字数据的某些位的组合、分离或位设置。

例1：AND AH，0F0H；　分离出AH中的高4位.

AND AH，0FH；　分离出AH中的低4位

OR　AH，01H；　将AH中最低位置1

AND AL，7FH；　将AL的最高位置0

XOR AX，0FFH；　将AX的低字节变反

XOR　BX，8000H；将BX的符号位变反

例2：下面的程序段将中断标志位IF清0，其它标志位保持不变。

```
PUSHF                ;将标志寄存器压栈
POP  AX              ;将栈中的标志字送AX
AND  AX，0FDFFH      ;将AX的第9位清0
PUSH  AX             ;将第9位清0后的AX内容压栈
POPF                 ;将堆栈中的值返回到标志寄存器
```

# 2.测试指令

**指令格式：TEST DEST，SRC**

该指令的功能与AND指令相似，实现源操作数与目的操作数进行按位“逻辑与”运算，对标志位的影响与AND指令相同，但运算的结果不送入目的操作数，即目的操作数内容也将保持不变。

TEST指令主要用于测试某一操作数的一位或几位的状态。

例1 TEST AL，01
JZ ZERO
…….
ZERO: ……

该程序段检查AL寄存器的最低位是否为0，如果为0，则程序转移到ZERO处执行。

例2 LAHF
TEST AH，04H
JZ TARGET

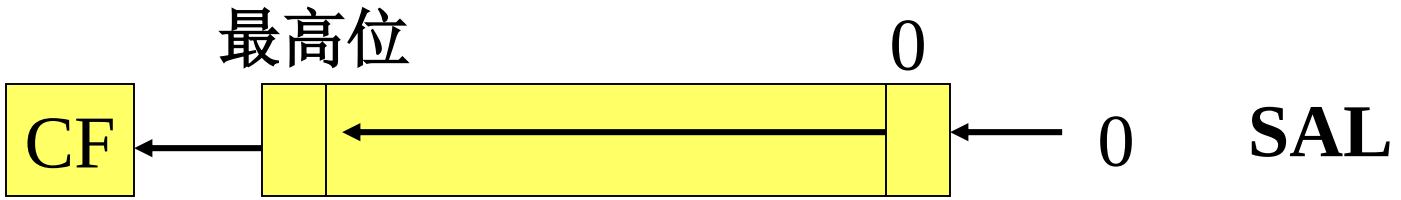
该程序段检查标志寄存器的PF位（第2位）是否为0，如果为0，则执行后标志ZF为1。因此通过测试ZF标志即可。

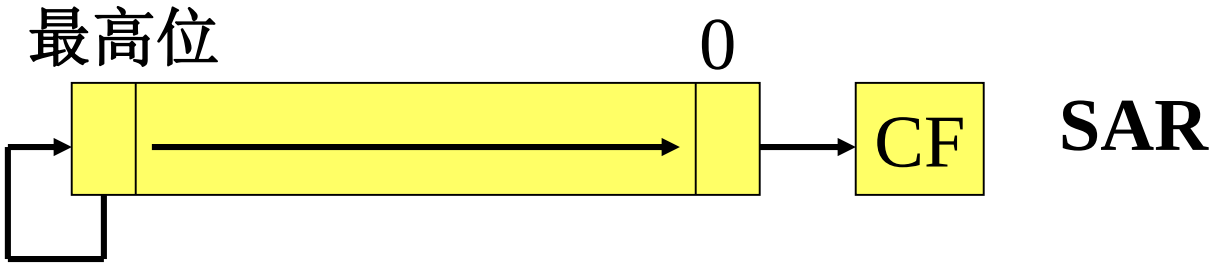
# 3.移位/循环移位指令

这一类指令共有**8条**,分为**3类**。

## （1）算术移位

算术左移 **SAL DEST，COUNT**
算术右移 **SAR DEST，COUNT**

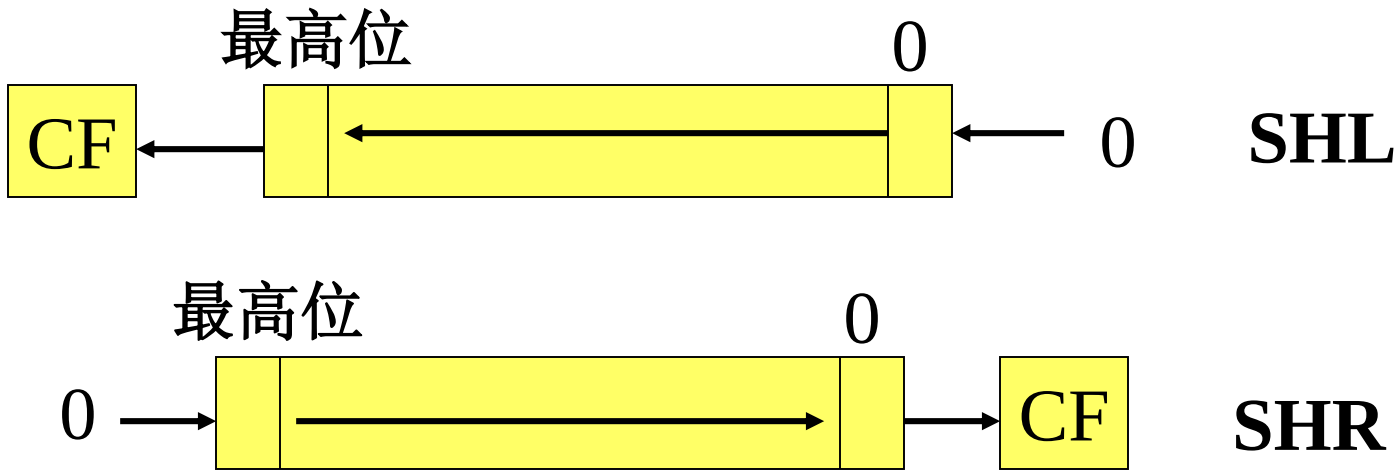


## （2）逻辑移位

逻辑左移 SHL DEST，COUNT
逻辑右移 SHR DEST，COUNT

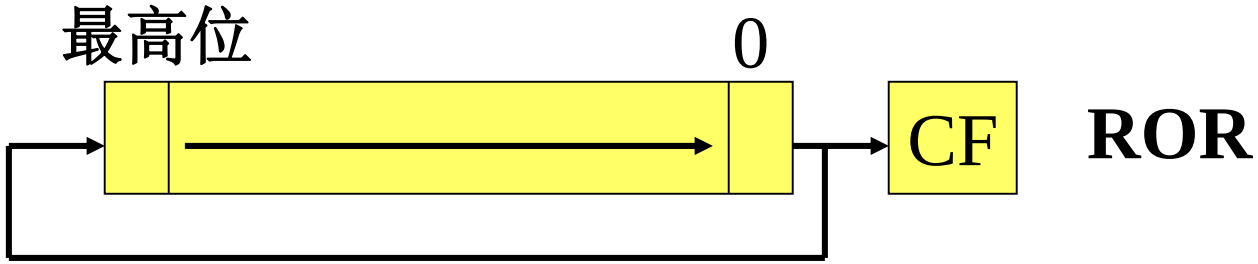
逻辑左移SHL与算术左移SAL功能相同。



## （3）循环移位

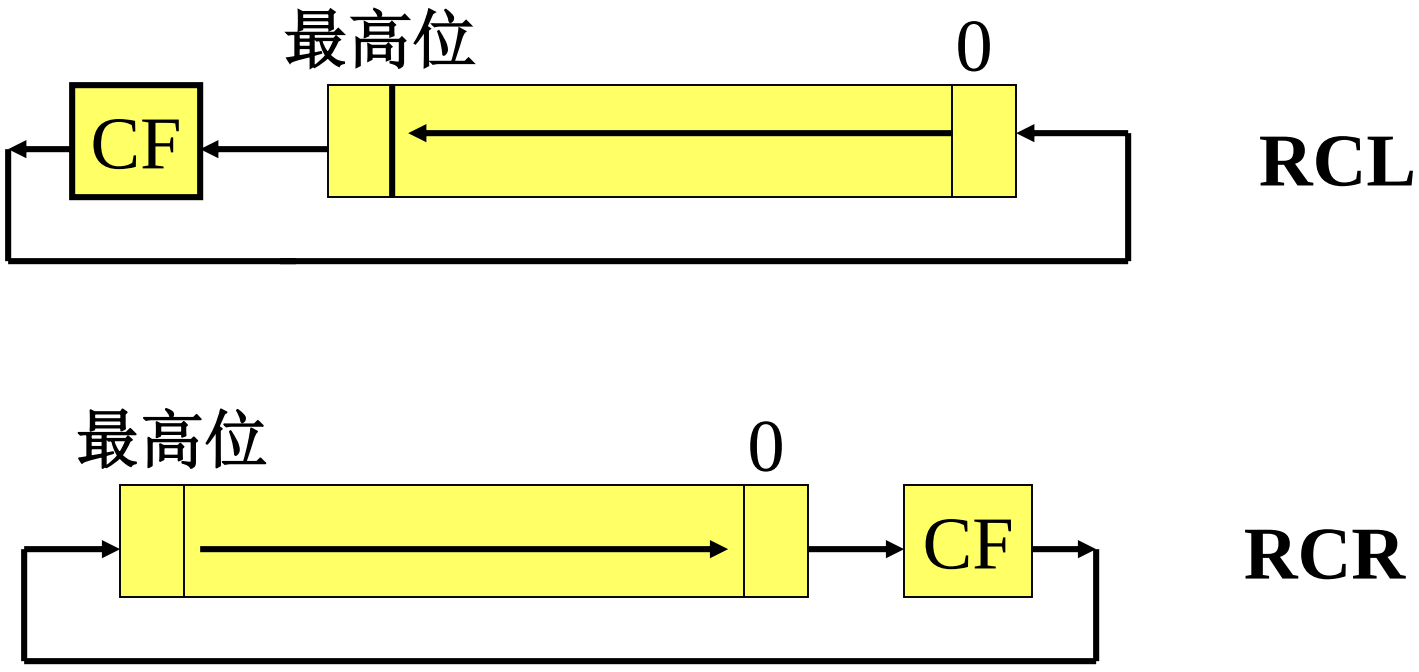
小循环：

循环左移 **ROL DEST，COUNT**
循环右移 **ROR DEST，COUNT**

# 大循环：

带进位循环左移 **RCL DEST，COUNT**
带进位循环右移 **RCR DEST，COUNT**



这8条指令具有以下几个共同点：

（1）DEST为操作对象，它可以是字节或字操作数，可以是通用寄存器或存储器操作数。

（2）COUNT用来决定移位/循环的位数，即确定移位的次数。

当移位次数为1时，使用常数1或寄存器CL。

当移位次数大于1时，必须使用寄存器CL。

例1：SAL AX，1；将AX的内容左移1位，其中最高位移入CF中，而低位补0.

例2：MOV CL，2
SAR AX，CL；将AX的内容算术右移2位。

（3）在执行移位时，根据指令不同，每移位一次,最高位（左移）或最低位（右移）都要送到进位标志CF。

例3：MOV AL，10010011B
SHL AL，1 ；执行后CF标志为1
SAR AL，1 ；执行后CF标志为0

（4）前4条移位指令根据移位结束后修改标志位CF、PF、ZF、SF和OF，而AF不确定。而后4条循环移位指令根据移位结束后的结果仅修改CF和OF

对溢出标志位OF的影响：

移位次数为1时，移位前后操作数的符号位发生变化，则OF被置1，否则置0。移位次数大于1时，OF不确定。

```
例4：MOV  AL，11000000B；(AL)=-64
     MOV   BL，01111111B；(BL)=127
     SAL   AL，1 ；(AL)=10000000B=-128,OF=0
     SAL   BL，1 ；(BL)=11111110B=-2, OF=1
```

指令SAL和SAR当移位次为n时，其作用相当于乘以$2^n$或除以$2^n$，因此被叫做算术移位指令。

为了保持其算术运算结果的正确性，移位后的结果不能发生溢出。

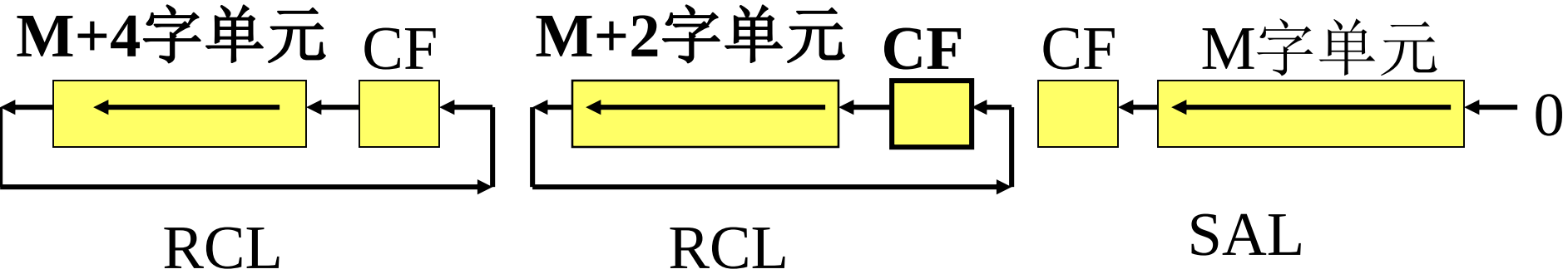
例5 设AX中存放一个带符号数，若要实现(AX) ×5÷2，可由以下几条指令完成。

```
MOV    DX，AX
SAL    AX，1
SAL    AX，1
ADD    AX，DX
SAR    AX，1
```

对于多字节或多字数据的移位，需要使用带进位循环移位指令。

**例6　下面程序段对从存储单元M开始的三字数据执行左移一位。**

```
SAL　M，1
RCL　M+2，1
RCL　M+4，1
```



下面的程序段实现将上述三字数据右移一位。

```
SAR   M+4，1
RCR   M+2，1
RCR   M，1
```

# 四、处理器控制类指令

处理器控制类指令包括以下三种情况。

## 1.标志位操作指令

它们都是无操作数指令，操作数隐含为标志寄存器的某个标志位。能直接操作的标志位有CF、IF和DF。

（1）清除进位标志　　CLC ；置CF为0

（2）置1进位标志　　STC ；置CF为1

（3）进位标志取反　　CMC ；CF的值取反

| | |
|---|---|
| （4）清除方向标志 | CLD；置DF为0 |
| （5）置1方向标志 | STD；置DF为1 |
| （6）清除中断标志 | CLI；置IF为0 |
| （7）置1中断标志 | STI；置IF为1 |

## 2、与外部事件同步的指令

**HLT ；暂停指令**

**WAIT ；等待指令**

**ESC ；外部协处理器指令前缀**

**LOCK ；总线锁定指令**

## 3、空操作指令 NOP

**执行一次NOP占用CPU三个时钟周期，它不改变任何寄存器或存储单元内容，主要用于延时。**

# 3.3 指令编码

汇编:将汇编语言程序转换为机器语言程序的过程

汇编程序:在计算机中实现汇编过程的系统程序

Intel8086/8088汇编指令的编码格式有四种基本格式。

1.双操作数指令编码格式

2.单操作数指令编码格式

3.与AX或AL有关的指令编码格式

4.其它指令编码格式

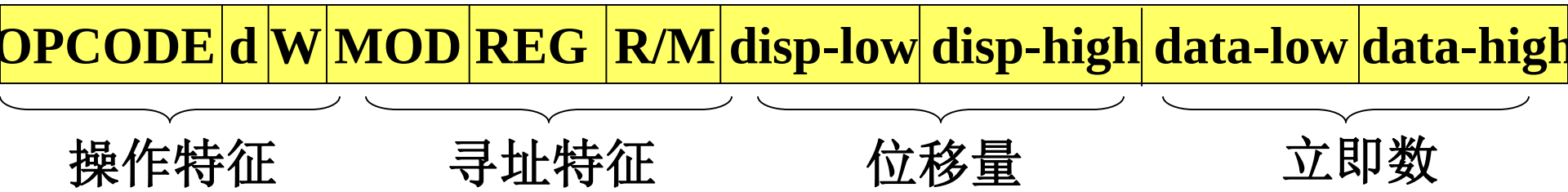
# 一、双操作数指令编码格式

对于象MOV、ADD、AND等双操作数指令，操作数可以是以下两种情形：

- 一个操作数在寄存器中，另一操作数在寄存器或存储器中。

- 目的操作数在寄存器或存储器中，源操作数是立即数。

这类指令的机器目标代码长度为2～6个字节




整个指令编码可以包含4个部分，但其中某些部分在一些指令的编码中可以没有。

## 1.操作特征部分

这部分为指令编码的首字节，它又分为以下三个段。

### （1）OPCODE:操作码字段

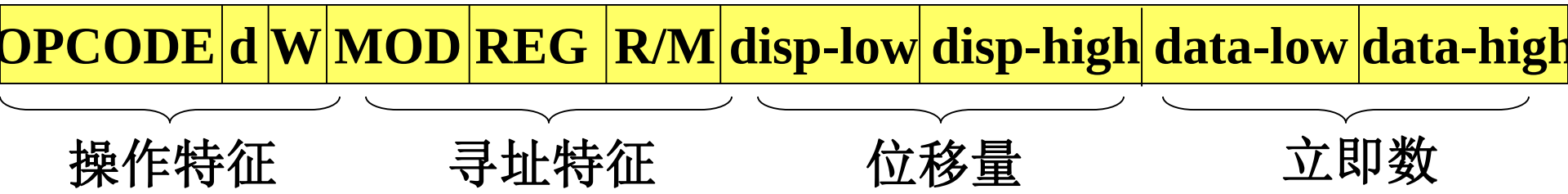
该字段长度为6bit。它表示了该指令所执行的功能和两个操作数的来源。

例如：

| 操作码 | 指令 | 目的操作数 | 源操作数 |
|---|---|---|---|
| 100010 | MOV | REG | R/M |
| 1100011 | MOV | R/M | Imm |
| 000000 | ADD | REG | R/M |
| 001000 | AND | R/M | REG |
| 1000000 | AND | R/M | Imm |
| 1000000 | OR | R/M | Imm |

如果指令的源操作数是立即数，则需要使用指令编码的第2字节中**REG**字段作辅助操作码。

前面例子中的最后两条指令，虽然其**OPCODE**字段相同，但它们的辅助操作码字段不同。



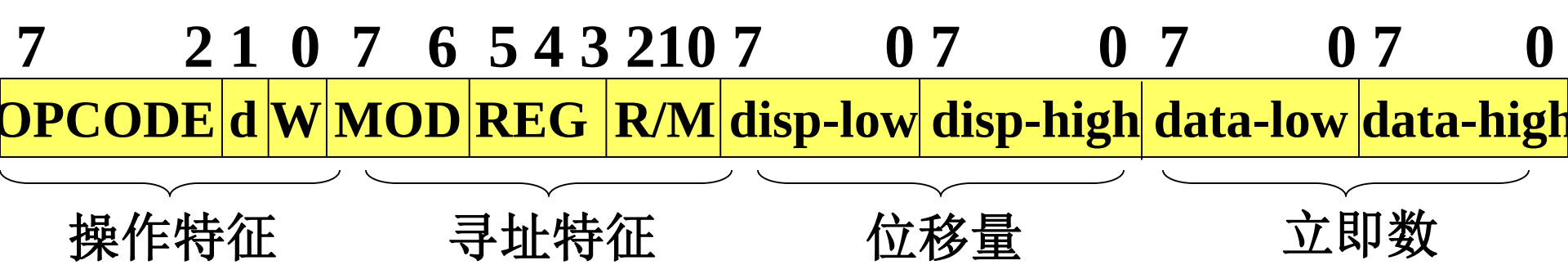

## （2）方向字段d

该字段与第2部分寻址特征一起来决定源操作数和目的操作数的来源。

注意：当源操作数为立即数Imm时，d字段无效，它被并入 操作码字段。

## （3）字/字节字段W

当W=1时，表示两操作数长度为字；当W=0时，表示两操作数长度为字节。




## 2.寻址特征部分

它与操作特征部分的方向字段d结合，指定两个操作数分别使用什么寻址方式，及使用哪个寄存器。

它包括MOD、REG和R/M三个字段，REG字段确定一个操作数,而MOD和R/M字段确定另一个操作数。

当d=1时，则目的操作数由REG字段确定，而源操作数由MOD和R/M字段确定。

当d=0时，则目的操作数由MOD和R/M 字段确定，而源操作数由REG字段确定。

## （1）REG字段

由REG字段确定的一个操作数是某一通用寄存器的内容，即使用的是寄存器寻址方式。

第一部分中的W字段决定操作数是字或是字节。

它们配合使用可以有16种组合，也即可以分别指定16个寄存器之一。如下表所示：

| REG | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| W=0 | AL | CL | DL | BL | AH | CH | DH | BH |
| W=1 | AX | CX | DX | BX | SP | BP | SI | DI |

如果REG字段被用于指定段寄存器（用于MOV指令），则它的编码与指定的段寄存器如下。

| REG | 000 | 001 | 010 | 011 |
|---|---|---|---|---|
| 段寄存器 | ES | CS | SS | DS |

## （2）寻址方式字段MOD和寄存器/存储器字段R/M

这两个字段共同确定一个操作数。该操作数可以在寄存器中，也可以在存储器中

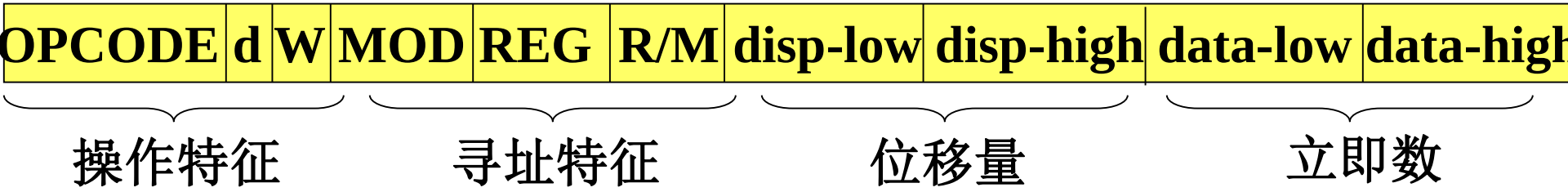
MOD、R/M和W字段共同确定操作数的寻址方式和所使用的寄存器，如下表所示。

| R/M \ MOD | 存储器有效地址计算方法 | | | 寄存器方式(11) | |
|---|---|---|---|---|---|
| | 00 | 01 | 10 | w=0 | w=1 |
| 000 | (BX)+(SI) | (BX)+(SI)+disp8 | (BX)+(SI)+disp16 | AL | AX |
| 001 | (BX)+(DI) | (BX)+(DI)+disp8 | (BX)+(DI)+disp16 | CL | CX |
| 010 | (BP)+(SI) | (BP)+(DI)+disp8 | (BP)+(DI)+disp16 | DL | DX |
| 011 | (BP)+(DI) | (BP)+(SI)+disp8 | (BP)+(SI)+disp16 | BL | BX |
| 100 | (SI) | (SI)+disp8 | (SI)+disp16 | AH | SP |
| 101 | (DI) | (DI)+disp8 | (DI)+disp16 | CH | BP |
| 110 | disp16 | (BP)+disp8 | (BP)+disp16 | DH | SI |
| 111 | (BX) | (BX)+disp8 | (BX)+disp16 | BH | DI |

注意：在表中没有使用BP作寄存器间接寻址方式，如果在指令中使用了[BP]，则将其汇编为[BP+0]，即基址寻址。

当MOD=11时，操作数为16个寄存器之一的内容。

当MOD=00,01,11时,操作数为存储器单元,可有24种有效地址EA计算方法。disp8和disp16分别为8位和16位位移量




## 3.位移量部分

根据寻址特征中MOD和R/M字段确定的有效地址计算方法，位移量可以是以下三种情况之一：

没有位移量　　1字节位移量disp8　　2字节位移量disp16

## 4.立即数部分

如果指令的源操作数为立即数，则指令编码中包含有该部分。它总是位于指令编码的最后1~2字节。

## 例1.MOV M-WORD，OAABBH

该指令功能为将16位立即数送存储单元，目的操作数为直接寻址方式。

查附录B可知：指令操作码为1100011

字操作，W=1

源操作数为立即数，REG字段为辅助操作码000

设M-WORD存储单元的偏移量为0010H

由于目的操作数为直接寻址，根据前面的MOD和R/M字段编码表可知MOD=00 R/M=110

则整个指令的编码为：

| OPCODE | W | MOD | REG | R/M | disp-low | disp-high | data-low | data-high |
|---|---|---|---|---|---|---|---|---|
| 1100011 | 1 | 00 | 000 | 110 | 10 | 00 | BB | AA |

用16进制数表示为：C7 06 10 00 BB AA共6个字节。

## 例2. MOV DS，AX

该指令将通用寄存器AX的内容送入段寄存器DS，因此REG字段必须用于指定DS，即为011。

MOD和R/M用于指定AX，即MOD=11 R/M=000

d＝1，w被作为OPCODE 查表为100011d0

整个指令编码为：

| OPCODE | MOD | REG | R/M |
|---|---|---|---|
| 10001110 | 11 | 011 | 000 |

16进制目标代码为：8ED8

## 例3 MOV AX，ES：[BX]

该指令为寄存器间址的存储单元内容送通用寄存器AX

指令中使用了段前缀ES，即由ES代替数据段DS。指令编码的第一个字节就为段前缀标记代码。

| 段前缀 | 代码 |
|---|---|
| ES | 00100110 (26) |
| CS | 00101110 (2E) |
| SS | 00110110 (36) |
| DS | 00111110 (3E) |

段前缀标记字节的前3位和后3位被固定为001和110，中间两位被用来指定不同的段寄存器。

该指令编码为：

| 段前缀码 | OPCODE | d | w | MOD | REG | R/M |
|---|---|---|---|---|---|---|
| 00100110 | 100010 | 1 | 1 | 00 | 000 | 111 |

**16进制目标代码为：26 8B 07**

# 二、单操作数指令编码格式

这种编码格式适用于只有一个操作数的指令，如INC、DEC、移位/循环等指令。指令编码为2~3字节。


操作特征部分：

包括OPCODE 、V和W三个字段，其中V字段只有移位/循环指令中才有该字段。其它指令中没有该字段。

V=0时，指令中使用常数1作为移位或循环次数。
V=1时，指令中使用寄存器CL作移位次数。

由于单操作数指令中只有一个操作数，因此寻址特征部分就不需要REG字段，而该字段被用作辅助操作码。

例1　INC　AL

该指令为将寄存器AL内容加1，查表可知其操作码和辅助操作码分别为1111111和000 。

该指令编码为

| OPCODE | W | MOD | OPCODE | R/M |
|---|---|---|---|---|
| 1111111 | 0 | 11 | 000 | 000 |

16进制目标代码为：FE　C0

## 例2　SHR AL，CL

该指令为对寄存器AL内容执行逻辑右移，移位次数由CL给出,即V字段为1。由MOD和R/M确定AL，即MOD=11 R/M=000

查附表可知：操作码和辅助操作码分别为110100和101

指令编码为：

| OPCODE | V | W | MOD | OPCODE | R/M |
|---|---|---|---|---|---|
| 110100 | 1 | 0 | 11 | 101 | 000 |

16进制目标代码为：D2 E8

# 三、与AX或AL有关的指令编码格式

这种编码格式用于隐含指定AX/AL作为一个操作数的双操作数指令，其编码格式为：

| OPCODE | W | disp-low | disp-high | data-low | data-high |
|---|---|---|---|---|---|
| 7 1 | 0 | 7 0 | 7 0 | 7 0 | 7 0 |

采用这种编码格式的指令，除一个操作数隐含指定为AX/AL外，另一个操作数可以是立即数或存储单元。

立即数：则编码中应有1~2字节的立即数

存储单元：只能使用直接寻址方式，位移量由disp字段给出。

**例1 AND AL，0FH**

**该指令功能是将寄存器AL的内容与立即数0FH进行逻辑“与”。因此指令编码中包含了立即数（8位）部分，而不包含位移量。**

**查附录二可知其操作码部分为0010010，指令编码如下：**

| OPCODE | W | data |
|---|---|---|
| 0010010 | 0 | 00001111 |

**16进制目标代码为：24 0F**

# 四、其它指令编码格式

除上述三种编码格式外，还有一些指令的编码格式更简单。如标志位操作指令、堆栈操作指令等。这些指令的编码格式一般只有一个字节。

例如 CLC清进位标志，该指令的编码只有一个字节的操作码。即：

| 7 | | | | | | | 0 |
|---|---|---|---|---|---|---|---|
| OPCODE | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

在有些单字节指令的编码中，将该字节划出部分位作为REG字段。例如PUSH指令，若压入堆栈的是通用寄存器，则编码格式为：

| 7 ~ 3 | 2 1 0 |
|---|---|
| OPCODE | REG |
| 01010 | |

若压栈的是段寄存器则编码格式为：

| 7 6 5 | 4 3 | 2 1 0 |
|---|---|---|
| OPCODE | REG | OPCODE |
| 0 0 0 | | 1 1 0 |

# 第四章　汇编语言程序格式

本章主要内容：

- 汇编语言语句种类及其格式
- 汇编语言数据
- 符号定义语句
- 表达式与运算符
- 程序的段结构
- 过程定义伪指令
- 当前位置计数器$与定位伪指令
- 从程序返回操作系统的方法

不同的汇编程序有不同的汇编语言编程规定。目前支持**Intel 8086/8088**系列微机,常用的汇编程序有ASM、MASM、TASM、OPTASM等 。

本章主要介绍汇编语言程序设计中的一些基本书写格式与语法规则。

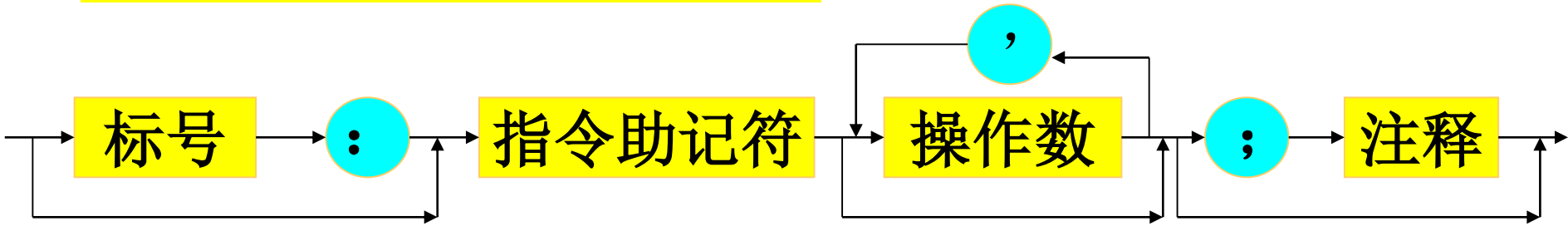
# 4.1 汇编语言语句种类及其格式

汇编语言的语句可以分为指令语句和伪指令语句

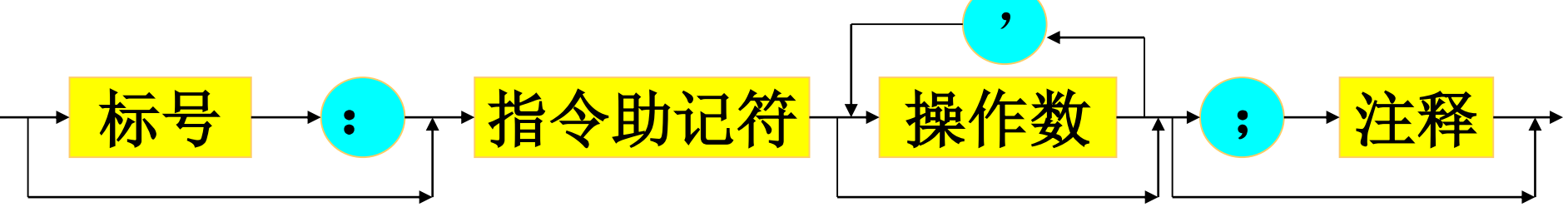
## 一、指令语句

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

指令语句的一般格式为：




一条指令语句最多可以包含4个字段



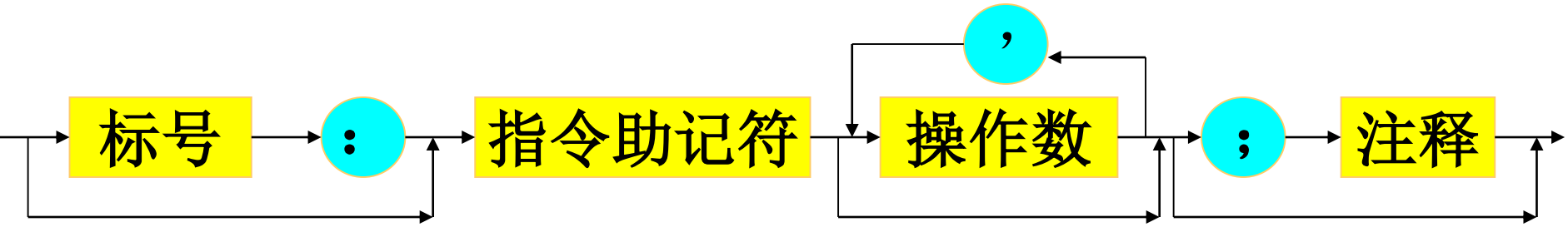

## 1.标号字段

标号是可选字段，它后面必须有“：”。标号是一条指令的符号地址，代表了该指令的第一个字节存放地址。

标号一般放在一个程序段或子程序的入口处，控制程序的执行转到该程序位置。

在转移指令或子程序调用指令中，可直接引用这个标号。

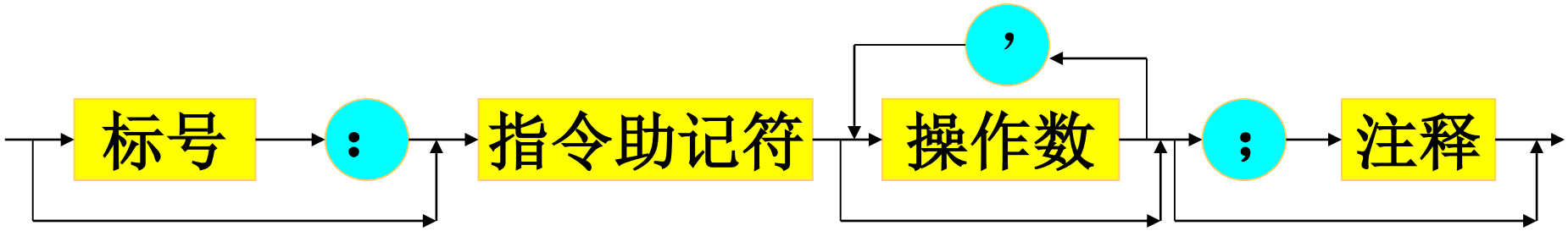
例　ADDR1：MOV　AL，100




## 2.指令助记符字段

该字段是一条指令的必选项，它表示这条语句要求CPU完成什么具体操作，如MOV、ADD、SHL等。

有些指令还可以在指令助记符的前面加上前缀，实现一定的附加操作。如串操作指令前所加的重复前缀REP（见第7章介绍）等。




## 3.操作数字段

一条指令可以有一个操作数、两个操作数或者无操作数。

如ADD、MOV指令需要两个操作数，INC、NOT指令只需一个操作数，而CLC指令不需要操作数。

# 4.注释字段

注释字段为可选项，该字段以分号“;”开始。

它的作用是为阅读程序的人加上一些说明性内容

注释字段不会产生机器目标代码，它不会影响程序和指令的功能。

注释字段可以是一条指令的后面部分，也可以是整个语句行。

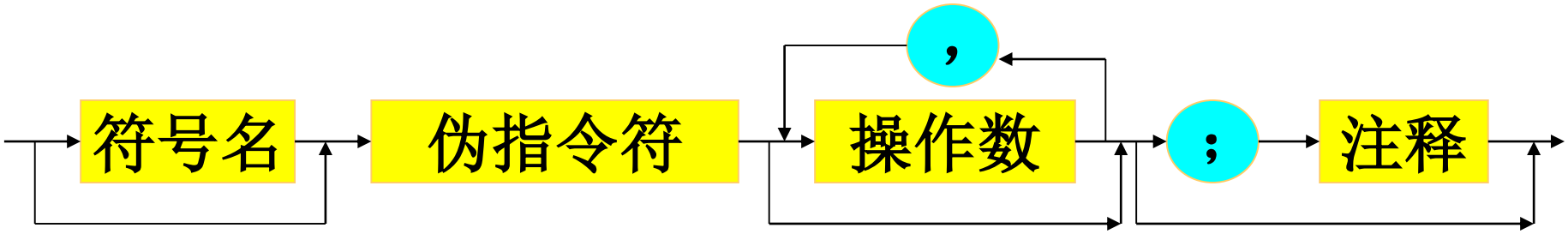
例：LABEL1：ADD AX，BX；功能为AX<=(AX)+(BX)
;后面的程序段将完成一次对存储器的访问

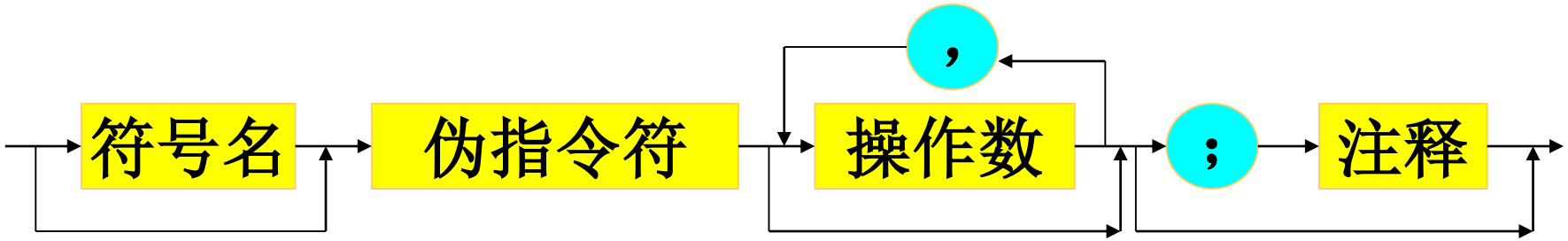
# 二、伪指令语句

伪指令语句又叫命令语句。

伪指令本身并不产生对应的机器目标代码。它仅仅是告诉汇编程序对其后面的指令语句和伪指令语句的操作数应该如何处理。

一条伪指令语句可以包含四个字段。如下所示：



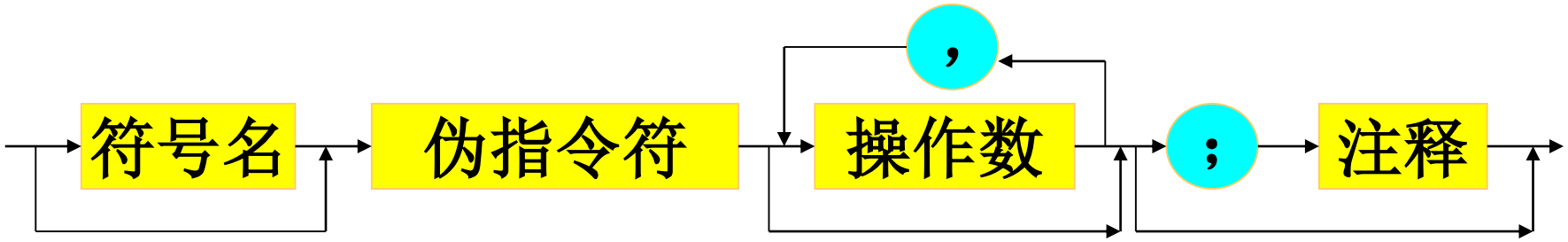



## 1.符号名字段

该字段为可选项。根据伪指令的不同，符号名可以是常量名、变量名、过程名、结构名和记录名等等。

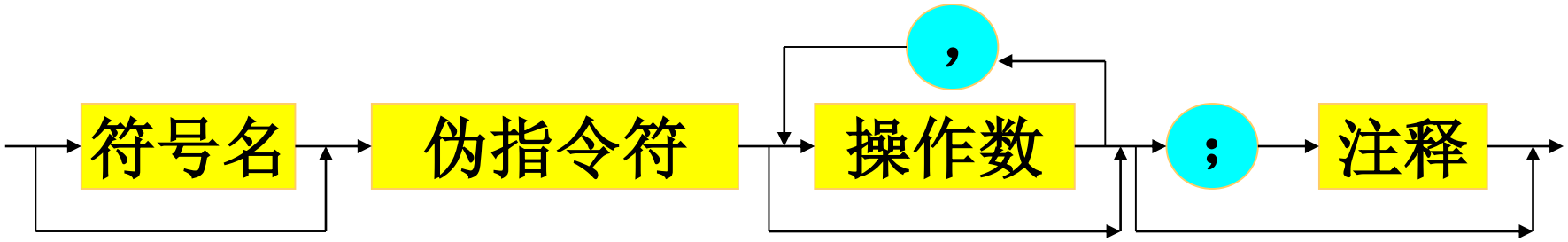
一条伪指令语句的符号名可以作其它伪指令语句或指令语句的操作数，这时它表示一个常 量或存储器地址

注意：符号名后面没有冒号“：”，这是与指令语句的重要区别。




## 2.伪指令符字段

该字段是伪指令语句的必选项，它规定了汇编程序所要完成的具体操作。本章后面的章节将对各种伪指令作详细介绍。




## 3.操作数字段

该字段是否需要，以及需要几个是由伪指令符字段来决定。

操作数可以是一个常数（二进制、十进制、十六进制等）、字符串、常量名、变量名、标号和一些专用符号（如**BYTE**、**FAR**、**PARA**等）。

## 4.注释字段

注释字段为可选项，该字段必须以分号开始。其作用与指令语句的注释字段相同。

# 三、标识符

指令语句中的标号和伪指令语句中符号名统称为标识符。标识符是由若干个字符构成的。

标识符构成规则：

1.字符的个数为1~31个；

2.第一个字符必须是字母、问号、@或下划线“_”这4种字符之一；

3.从第二个字符开始，可以是字母、数字、@ 、“_”或问号“？”；

4.不能使用属于系统专用的保留字。

**保留字: CPU中各寄存器名（如AX、CS等），指令助记符（如MOV、ADD），伪指令符（如SEGMENT、DB）、表达式中的运算符（如GE、EQ）以及属性操作符（如PTR、OFFSET等）**

# 4.2 汇编语言数据

数据是指令和伪指令语句中操作数的基本组成部分。一个数据由数值和属性两部分构成。

在说明数据时不仅要指定其数值，还需说明它的属性，比如是字节数据还是字数据。

在汇编语言中常用的数据形式有：常数、变量和标号。

# 一、常数

常数在汇编期间其值已完全确定，并且在程序运行过程中，其值不会发生变化。

常数有以下几种形式：

**1.二进制数：以字母B结尾，如01001001B**

**2.八进制数：以字母O或Q结尾，如631Q 254O**

**3. 十进制数：以字母D结尾，或者没有结尾字母。如2007D、2007。**

**4. 十六进制数：以字母H结尾，如3FEH，如果常数的第一个数字为字母，为了与标识符加以区别，必须在其前面冠以数字“0”。**

**5.实数。一般格式为：**

**±整数部分• 小数部分E ±指数部分**

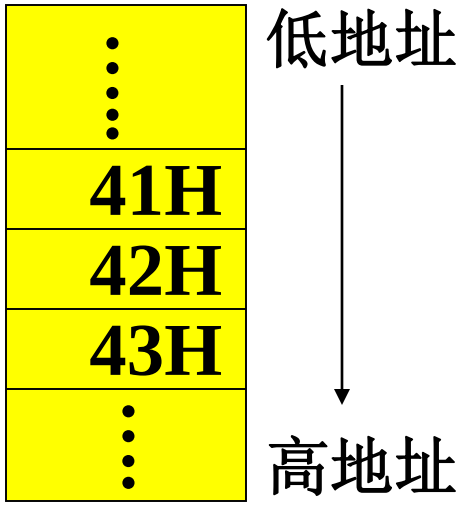
**尾数**

**例　2.134 E +10**

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

**6．字符串常数：用引号（单引号或双引号）括起来的一个或多个字符，这些字符以它的ASCII码值存储在内存。**

例如`B`在内存中为42H，`ABC`为41H 42H 43H。在内存中的存储如图所示。



常数在程序中可以用在以下几种情况：

（1）作指令语句的源操作数

```
MOV   AX，0B2F0H
ADD   AH，64H
```

（2）在指令语句的直接寻址方式、变址（基址）寻址方式或基址变址寻址方式中作位移量。

```
MOV   BX，32H [SI]
MOV   0ABH [BX]，CX
ADC   DX，1234H [BP][DI]
```

（3）在数据定义伪指令中使用

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

# 二、变量

变量用来表示存放数据的存储单元，这些数据在程序运行期间可以被改变 。

程序中以变量名的形式来访问变量，因此，可以认为变量名就是存放数据的存储单元地址。

## 1.变量的定义与预置

定义变量就是给变量在内存中分配一定的存储单元。也就是给这个存储单元赋与一个符号名，即变量名，同时还要将这些存储单元预置初值。

定义变量使用数据定义伪指令 **DB、DW、DD、DQ和DT等。**

## 变量定义的一般格式：

| | | | |
|---|---|---|---|
| 变量名 | DB | 表达式1，表达式2……； | 定义字节变量 |
| | DW | | 定义字变量 |
| | DD | | 定义4字节变量 |
| | DQ | | 定义8字节变量 |
| | DT | | 定义10字节变量 |

其中表达式1、表达式2是给存储单元赋的初值。

例如：

```
VAR_DATA    SEGMENT
DATA1    DB     12H
DATA2    DB      20H,30H
DATA3    DW     5678H
VAR_DATA     ENDS
```

当变量被定义后，就具有了以下三个属性：

## （1）段属性

它表示变量存放在哪一个逻辑段中。

例如上面例子中的变量DATA1、DATA2和DATA3三个变量都存放在VAR-DATA逻辑段中。

## （2）偏移量属性（OFFSET）

它表示变量所在位置与段起始点之间的字节数。

如上述例子中，变量DATA1的偏移量为0，DATA2为1，DATA3为3。

**段属性和偏移量属性就构造了变量的逻辑地址**

## （3）类型属性

它表示变量占用存储单元的字节数。其中DB伪指令定义的变量为字节，DW定义的变量为字，DD定义的为双字（4字节），DQ定义的为4字，DT定义的为5字。

在变量的定义语句中，给变量赋初值的表达式可以使用下面4种形式：

## （1）数值表达式

例如：DATA1 DB 32，30H
DATA1的内容为32（20H），DATA1+1单元内容为30H.

## （2）？表达式

不带引号的问号“？”表示可以预置任意内容。

例如：**DA-BYTE DB ？，？，？**

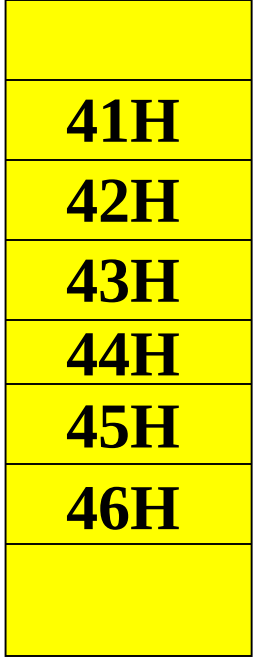
表示让汇编程序分配三个字节存储单元。这些存储单元的内容的值为任意值。

## （3）字符串表达式

对于DB伪指令，字符串为用引号括起来的不超过255个字符。给每一个字符分配一个字节单元。字符串按从左到右，将字符的ASCII编码值以地址递增的排列顺序依次存放。

例如：STRING1 DB ‘ABCDEF’

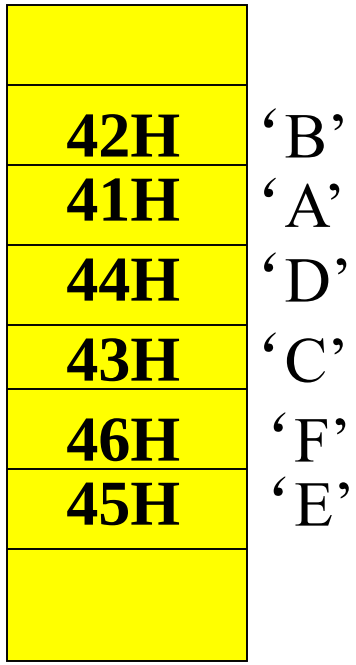
STRING1




对于DW伪指令可以给两个字符组成的字符串分配两个字字节存储单元。

注意：两个字符的存放顺序是前一个字符放在高地址，后一字符放低地址单元。

STRING2

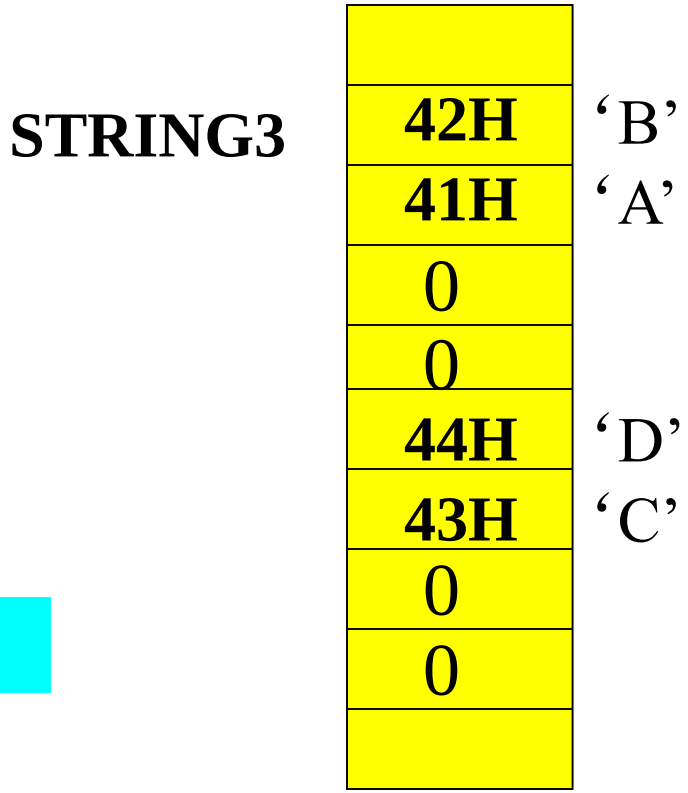



例如：STRING2 DW ‘AB’, ‘CD’, ’EF’

对于DD伪指令，只能给两个字符组成的字符串分配4个字节单元。

两个字符存放在较低地址的两个字节单元中。存放顺序与DW伪指令相同，而较高地址的两个字节单元存放0。

例如：STRING3 DD ‘AB’, ‘CD’




注意：DW和DD伪指令不能用两个以上字符构成的字符串赋初值，否则将出错。

## （4）DUP表达式

DUP称为重复数据操作符。

使用DUP表达式的一般格式为：

变量名 { DB / DW / DD } 表达式1 DUP (表达式2)

其中：表达式1是重复的次数，表达式2是重复的内容。

例如：DATA_A DB 10H DUP(?)

DATA_B DB 20H DUP('AB')

分配16个字节单元

分配20H*2=40H个 字节，其内容为重复字符串 'AB'。

DUP还可以嵌套使用，即表达式2又可以是一个带DUP的表达式。

例如：DATA_C DB 10H DUP(4 DUP(2),7)
重复10H个数字序列“2，2，2，2，7”，共占用10H*5=50H个字节。

## 2.变量的使用

### （1）在指令语句中引用

在指令语句中直接引用变量名就是对其存储单元的内容进行存取

```
例如：DA1  DB  0FEH
      DA2  DW  52ACH
      ……
      MOV  AL,DA1   ;将0FEH传送到AL中
      MOV  BX,DA2   ;将52ACH传送到BX中
```

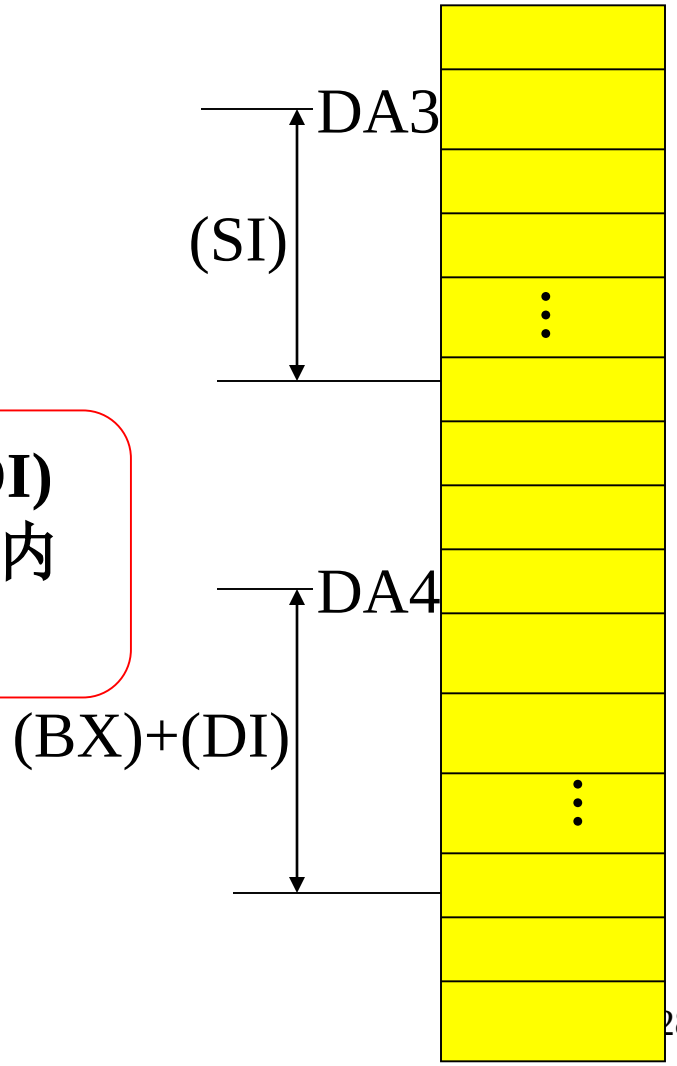
当变量出现在变址（基址）寻址或基址变址寻址的操作数中时表示取用该变量的偏移量。

例如：

```
DA3  DB  10H  DUP(?)
DA4  DW  10H  DUP（1）
MOV  DA3[SI]，AL
ADD  DX，DA4[BX][DI]
```

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

将AL的内容送入从DA3 开始再偏移(SI)的存储单元中



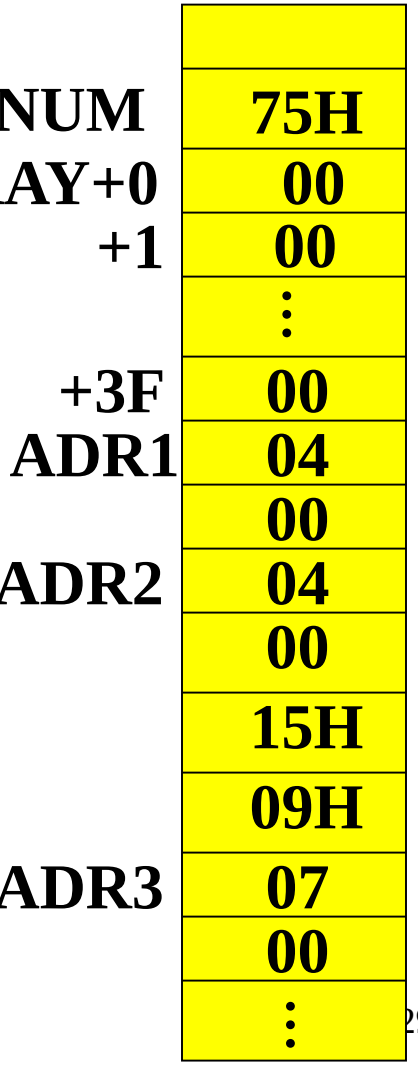
## （2）在伪指令语句中引用

```
NUM     DB   75H
ARRAY   DW   20H DUP(0)
ADR1    DW   NUM
ADR2    DD   NUM
ADR3    DW   ARRAY[2]
```

它表示取变 量地址的偏移量

后面三条伪指令的操作数中都包含了前面定义的两个变量

取变量段基值和偏移量。前两个字节存偏移量，后两个字节存段基值




设上述语句所在段的段基值为0915H，NUM的偏移量为0004H，则存储单元的分配情况如图所示。

# 三、标号

标号写在一条指令的前面，它就是该指令在内存的存放地址的符号表示，也就是指令地址的别名。

标号主要用在程序中需要改变程序的执行顺序时，用来标记转移的目的地，即作转移指令的操作数。

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

每个标号具有三属性

**（1）段属性（SEG）**

它表示该标号所代表的地址在哪个逻辑段中，即段基值。

**（2）偏移量属性（OFFSET）**

它表示该标号所代表的地址在段内与段起点间的字节数，即地址的偏移量。

**（3）距离属性（也叫类型属性）**

它表示该标号可以被段内还是段间的指令调用。

**NEAR（近）：** 该标号只能作段内转移，也就是说只能是与该标号所指指令同在一个逻辑段的转移指令和调用指令才能使用它。

**FAR（远）：** 该标号可以被非本段的转移和调用指令使用。

标号的距离属性可以有两种方法来指定：

**a.隐含方式**

当标号加在指令语句前面时，它隐含为NEAR属性。

**例 SUB1：MOV AX，30H**

SUB1的距离属性为NEAR也就是它只能被本段的转移指令和调用指令访问。

**b.用LABEL伪指令给标号指定距离属性**

格式： 标号名 LABEL 类型

类型为NEAR或FAR。该语句应与指令语句连用。

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

SUB1_FAR与SUB1两个标号具有相同的段属性和偏移量属性，即相同的逻辑地址。被转移指令或调用指令访问时，是指同一个入口地址，但SUB1-FAR可以被其它段的指令调用。

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

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

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

# 4.3 符号定义语句

在源程序设计中，使用符号定义语句可以将常数或表达式等内容用某个指定的符号来表示。在8086/8088汇编语言中有两种符号定义语句。

## 一、等值语句

语句格式：符号名　EQU　表达式

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

表达式可以是任何形式，常见的有以下几种情况。

### 1.常数或数值表达式

```
COUNT  EQU  5
NUM    EQU  COUNT+5
```

## 2.地址表达式

**ADR1 EQU DS：[BP+14]**

**ADR1被定义为在DS数据段中以BP作基址寻址的一个存储单元。**

## 3.变量、寄存器名或指令助记符

**例如：CREG EQU CX；在后面的程序使用CREG就是使用CX**
**CBD EQU DAA；DAA为十进制调整指令。**

**注意：在同一源程序中，同一符号不能用EQU定义多次。**

**例：CBD EQU DAA**
**CBD EQU ADD**

} 错误用法

# 二、等号语句

格式：符号名=表达式

等号语句与等值语句具有相同的作用。但等号语句可以对一个符号进行多次定义。

例如：

**CONT=5**

**NUM=14H**

**NUM=NUM+10H**

下面是错误用法：

**CBD=DAA**

**……**

**CBD=ADD**

等号语句不能为助记符定义别名

注意：等值语句与等号语句都不会为符号分配存储单元。因此所定义的符号没有段、偏移量和类型等属性。

# 4.4 表达式与运算符

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

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

8086/8088宏汇编语言中的操作运算符非常丰富，可以分为以下五类。

## 一、算术运算符

+、—、＊、 / 、MOD、SHL、SHR、[ ]

1.运算符“+”和“-”也可作单目运算符，表示数的正负。

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

**3.“/”运算为取商的整数部分，而“MOD”运算取除法运算的余数。**

例如：

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

## 4. “SHR ”和“ SHL ”为逻辑移位运算符

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

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

例如

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

不能改成：
SHL NUM,1

上面的指令序列等效下面三条指令。

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

**5.下标运算符“[ ]”具有相加的作用**

一般使用格式：　表达式1 [表达式2]

作用：将表达式1与表达式2的值相加后形成一个存储器操作数的地址。

下面两个语句是等效的。

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

可以用寄存器来存放下标变量

例：下面几个语句是等价的

```
MOV  AX , ARRAY[BX][SI]; 基址变址寻址
MOV   AX, ARRAY[BX+SI]
MOV   AX , [ARRAY+BX][SI]
MOV   AX , [ARRAY+SI][BX]
MOV   AX , [ARRAY+BX+SI]
```

下面是几个错误的语句。

```
MOV   AX，ARRAY+BX+SI
MOV   AX，ARRAY+BX[SI]
MOV   AX，ARRAY+DA_WORD
```

## 二、逻辑运算符

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

例如

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

## 三、关系运算符

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

关系运算符用来比较两个表达式的大小。关系运算符比较的两个表达式必须同为常数或同一逻辑段中的变量。

如果是常量的比较，则按无符号数进行比较；如果是变量的比较，则比较它们的偏移量的大小。

关系运算的结果只能是“真”（全1）或“假”（全0）

例1：

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

例2

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

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

# 四、数值返回运算符

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

## 1.SEG运算符

作用：取变量或标号所在段的段基值。

例如：

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

设DATA逻辑段的段基值为1FFEH，则两条传送指令将被汇编为：

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

## 2.OFFSET运算符

该运算符的作用是取变量或标号在段内的偏移量。

例如：

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

## 3.TYPE运算符

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

变量
- BYTE 1
- WORD 2
- DWORD 4
- QWORD 8
- TWORD 10

标号
- NEAR -1
- FAR -2

例如：

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

经汇编后的等效指令序列如下：

```
MOV  AL，01H
MOV  CL，02H
MOV  CH，04H
```

# 4.LENGTH运算符

该运算符用于取变量的长度。

- 如果变量是用重复数据操作符DUP说明的,则LENGTH运算取外层DUP给定的值。
- 如果没有用DUP说明，则LENGTH运算返回值总是1。

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

```
MOV   AL,  LENGTH   K1; (AL)=10H
MOV   BL,  LENGTH   K2 ; (BL)=1
MOV   CX,  LENGTH   K3 ;  (CX)=20H
MOV   DX,  LENGTH   K4 ;  (DX)=1
```

## 5.SIZE运算符

该运算符只能作用于变量，SIZE取值等于LENGTH和TYPE两个运算符返回值的乘积。

例如，对于上面例子，加上以下指令：

```
MOV  AL，SIZE  K1；  （AL）=10H
MOV   BL，SIZE  K2；  （BL）=1
MOV   CL，SIZE  K3；  （CL）=20H*2=40H
MOV   DL，SIZE  K4；  （DL）=1
```

# 五、属性修改运算符

这一类运算符用来对变量、标号或存储器操作数的类型属性进行修改或指定。

## 1.PTR运算符

使用格式：　类型　PTR　地址表达式

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

类型可以是BYTE、WORD、DWORD、NEAR和FAR。
这种修改是临时的，只在含有该运算符的语句内有效。

例如：

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

## 2.HIGH/LOW运算符

使用格式：

```
HIGH 表达式
LOW  表达式
```

这两个运算符用来将表达式的值分离出高字节和低字节。

如果表达式为一个常量，则将其分离成高8位和低8位；
如果表达式是一个地址（段基值或偏移量）时，则分离出它的高字节和低字节。

例如：

```
DATA   SEGMENT
CONST  EQU   0ABCDH
DA1    DB    10H DUP （0）
DA2    DW    20H DUP （0）
DATA   ENDS
       …….
MOV    AH ,HIGH  CONST
MOV    AL, LOW   CONST
MOV    BH, HIGH  (OFFSET DA1)
MOV    BL,  LOW  (OFFSET  DA2)
MOV    CH,  HIGH (SEG  DA1)
MOV    CL,  LOW  (SEG   DA2)
```

设DATA段的段基值是0926H，则上述指令序列汇编后的等效指令为:

```
MOV   AH ，  0ABH
MOV   AL ，  0CDH
MOV   BH ，  00H
MOV   BL ，  10H
MOV   CH ，  09H
MOV   CL ，  26H
```

注意：HIGH/LOW运算符不能用来分离一个变量、寄存器或存储器单元的高字节与低字节。

下面语句使用是错误的用法。

```
DA1   DW  1234H
……
MOV   AH，HIGH  DA1
MOV   BH，  LOW  AX
MOV   CH，HIGH [SI]
```

## 3、THIS运算符

THIS运算符一般与等值运算符EQU连用，用来定义一个变量或标号的类型属性。所定义的变量或标号的*段基值*和*偏移量*与紧跟其后的变量或标号相同。

例如：

```
DATA_BYTE  EQU   THIS  BYTE
DATA_WORD  DW    10 DUP (0)

           ……
           MOV   AX, DATA_WORD
           MOV   BL, DATA_BYTE
           ……
```

又如：

```
LFAR     EQU   THIS  FAR
LNEAR：  MOV   AX，  B
```

标号LFAR与LNEAR具有相同的逻辑地址值，但类型不同。LNEAR只能被本段中的指令调用，而LFAR可以被其它段的指令调用。

# 六、运算符的优先级

在一个表达式中如果存在多个运算符时，在计算时就有先后顺序问题。不同的运算符具有不同的运算优先级别。

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

汇编程序在计算表达式时，按以下规则进行运算。

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

例如：

**K1= 10 OR 5 AND 1 ；结果为K1=11**
**K2=（10 OR 5） AND 1；结果为K2=1**

# 4.5 程序的段结构

**8086/8088**在管理内存时，按照逻辑段进行划分，不同的逻辑段可以用来存放不同目的的数据。在程序中使用四个段寄存器**CS,DS,ES**和**SS**来访问它们。

在源程序设计时，使用伪指令来定义和使用这些逻辑段。

# 一、段定义伪指令

伪指令SEGMENT和ENDS用于定义一个逻辑段。使用时必须配对，分别表示定义的开始与结束。

一般格式：

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

段定义伪指令语句各部分的作用如下：

## 1、段名

段名是由用户自己任意选定的，符合标识符定义规则的一个名称。

最好选用与该逻辑段用途相关的名称。如第一个数据段为DATA1,第二个数据为DATA2等。

一个段的开始与结尾用的段名必须一致。

## 2、定位类型

定位类型用于决定段的起始边界，即第一个可存放数据的位置（不是段基址）。它可以有4种取值。

（1）**PAGE**：表示该段从一个页面的边界开始

由于一个页面为256个字节，并且页面编号从0开始，因此，PAGE定位类型的段起始地址的最后8位二进制数一定为0，即以00H结尾的地址。

（2）**PARA**:表示该段从一个小节的边界开始

如果用户未选定位类型，则缺省为PARA。

（3）**WORD**:表示该段从一个偶数字节地址开始，即段起始单元地址的最后一位二进制数一定是**0**。

（4）**BYTE**:表示该段起始单元地址可以是任一地址值。

注意：定位类型为**PAGE**和**PARA**时，段起始地址与段基址相同。定位类型为**WORD**和**BYTE**时，段起始地址与段基址可能不同。

## 3、组合类型

组合类型说明符用来指定段与段之间的连接关系和定位。它有六种取值选择。

（1）若未指定组合类型，表示本段与其它段无连接关系。在装入内存时，本段有自己的物理段，因此有自己的段基址

（2）**PUBLIC**:在满足定位类型的前提下，将与该段同名的段邻接在一起，形成一个新的逻辑段，共用一个段基址。段内的所有偏移量调整为相对于新逻辑段的段基址。

（3）**COMMON**: 产生一个覆盖段。在多个模块连接时，把该段与其它也用COMMON说明的同名段置成相同的段基址，这样可达到共享同一存储区。共享存储区的长度由同名段中最大的段确定。

（4）**STACK**:把所有同名段连接成一个连续段，且系统自动对SS段寄存器初始化为该连续段的段基址。并初始化堆栈指针**SP**。

用户程序中应至少有一个段用STACK说明，否则需要用户程序自己初始化SS和SP。

（5）**AT表达式**：表示本段可定位在表达式所指示的小节边界上。表达式的值也就是段基值。

（6）**MEMORY**:表示本段在存储器中应定位在所有其它段之后的最高地址上。如果有多个用MEMORY说明的段，则只处理第一个用MEMORY说明的段。其余的被视为COMMON

## 4.类别名

类别名为某一个段或几个相同类型段设定的类型名称。系统在进行连接处理时，把类别名相同的段存放在相邻的存储区，但段的划分与使用仍按原来的设定。

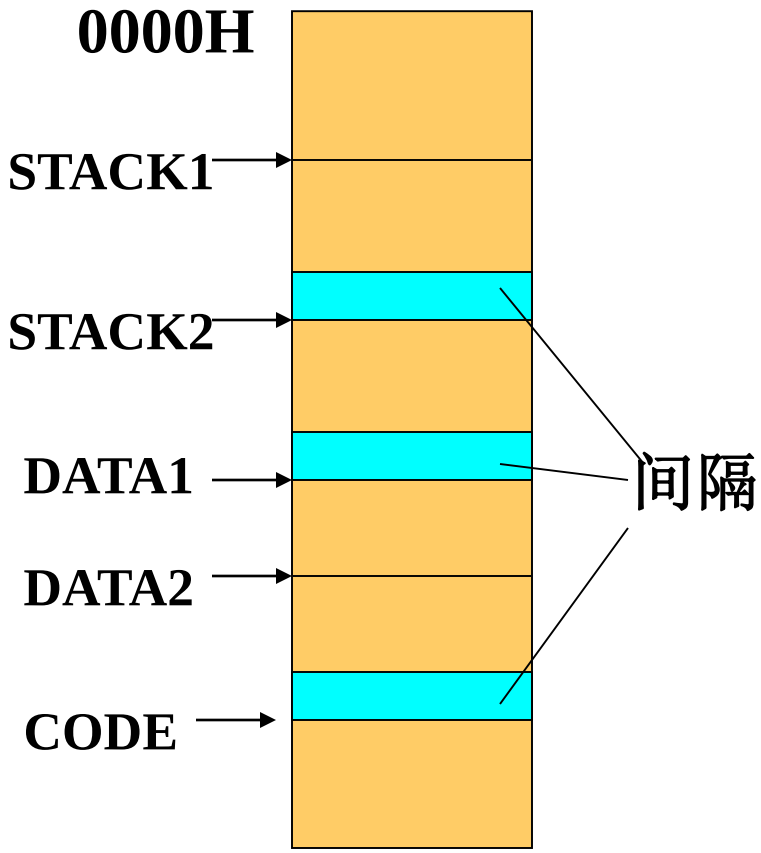
类别名必须用单引号引起来。所用字符串可任意选定，但它不能使用程序中的标号、变量名或其它定义的符号。

在定义一个段时，段名是必须有的项，而定位类型、组合类型和类别名三个参数是可选项。各个参数之间用空格分隔。各参数之间的顺序不能改变。

下面是一个分段结构的源程序框架。

```
STACK1   SEGMENT  PARA  STACK  'STACK0'
         .....
STACK1   ENDS
DATA1    SEGMENT  PARA  'DATA'
         ......
DATA1    ENDS
STACK2   SEGMENT  PARA  'STACK0'
         ......
STACK2   ENDS
CODE     SEGMENT  PARA  MEMORY
       ASSUME  CS:CODE,DS:DATA1,SS:STACK1
MAIN:    ......
         ......
CODE     ENDS
DATA2    SEGMENT  BYTE  'DATA'
         ......
DATA2    ENDS
              END  MAIN
```

上述源程序经LINK程序进行连接处理后，程序被装入内存的情况如图所示。




如果在段定义中选用了PARA定位类型说明，则在一个段的结尾与另一个段的开始之间可能存在一些空白，图中以兰色框表示。CODE段的组合类型为MEMORY,因此被装入在其它段之后。

在进行程序设计时，如果程序不大，一般只需要定义三个段就可以了。

# 二、段寻址伪指令

段寻址伪指令**ASSUME**的作用是告诉汇编程序,在处理源程序时，定义的段与哪个寄存器关联。

**ASSUME**并不设置各个段寄存器的具体内容，段寄存器的值是在程序运行时设定的。

一般格式：

**ASSUME 段寄存器名：段名，段寄存器名：段名，.....**

其中段寄存器名为**CS,DS,ES**和**SS**四个之一，段名是用**SEGMENT/ENDS**伪指令定义的段名。

例如：

```
DATA1  SEGMENT
VAR1    DB  12H
DATA1  ENDS
DATA2  SEGMENT
 VAR2    DB  34H
 DATA2  ENDS
 CODE   SEGMENT
 VAR3    DB  56H
      ASSUME  CS:CODE,DS:DATA1,ES:DATA2
START: .....
        .....
        INC  VAR1
        INC  VAR2
        INC  VAR3
        ......
CODE   ENDS
        END START
```

该指令汇编时，VAR1使用的是DS

该指令被汇编时，VAR2使用的是ES,即指令编码中有段前缀ES

该指令汇编时，VAR3使用的是CS，即指令编码中有段前缀CS

- 在一个代码段中可以有几条ASSUME伪指令，对于前面的设置，可以用ASSUME改变原来的设置。

- 一条ASSUME语句不一定设置全部段寄存器，可以选择其中一个或几个段寄存器。

- 可以使用关键字NOTHING将前面的设置删除。

例如：

```
ASSUME  ES:NOTHING ;删除前面对ES与某个定义段的关联
ASSUME  NOTHING    ;删除全部4个段寄存器的设置
```

# 三、段寄存器的装入

段寄存器的初值（段基值）装入需要用程序的方法来实现。四个段寄存器的装入方法略有不同。

## 1、DS和ES的装入

在程序中，使用数据传送语句来实现对DS和ES的装入。

例如：

```
DATA1    SEGMENT
DBYTE1  DB  12H
DATA1    ENDS
DATA2    SEGMENT
DBYTE2  DB 14H DUP(?)
DATA2    ENDS
CODE      SEGMENT
            ASSUME  CS:CODE,DS:DATA1
START: MOV  AX,DATA1
            MOV  DS,AX
            MOV  AX,DATA2
            MOV  ES,AX
            MOV  AL,DBYTE1
            MOV  DBYTE2[2],AL
            ……
CODE   ENDS
```

该指令在汇编时出错，因为在ASSUME指令中未指定ES与DATA2的联系。

为了改正上述程序中的错误，可以在变量DBYTE2前加一个段前缀说明即可。即：

MOV ES:DBYTE2[2], AL

## 2、SS的装入

SS的装入有两种方法

（1）在段定义伪指令的组合类型项中，使用STACK参数，并在段寻址伪指令ASSUME语句中把该段与SS段寄存器关联。

例如：

```
STACK1  SEGMENT  PARA  STACK
        DB  40H DUP(?)
STACK1  ENDS
    ……
CODE    SEGMENT
        ASSUME  CS:CODE,SS:STACK1
    ……
```

SS将被自动装入STACK1段的段基值，堆栈指针SP也将指向堆栈底部+2的存储单元。上例中（SP）=40H。

（2）如果在段定义伪指令的组合类型中，未使用STACK参数，或者是在程序中要调换到另一个堆栈，这时，可以使用类似于DS和ES的装入方法。

例如：

TOP变量的偏移量为40H

```
DATA_STACK   SEGMENT
             DB  40H DUP(?)
       TOP   LABEL  WORD
DATA_STACK   ENDS
          ……
CODE         SEGMENT
          ……
             MOV  AX,DATA_STACK
             MOV   SS,AX
             MOV   SP,OFFSET TOP
          ……
```

## 3、CS的装入

CPU在执行指令之前根据CS和IP的内容来从内存中提取指令,即必须在程序执行之前装入CS和IP的值。因此,CS和IP的初始值就不能用可执行语句来装入。

装入CS和IP一般有下面两种情况。

**(1)由系统软件按照结束伪指令指定的地址装入初始的CS和IP**

任何一个源程序都必须以END伪指令来结束。

其格式为：　END　起始地址

起始地址可以是一个标号或表达式，它与程序中第一条指令语句前所加的标号必须一致。

END伪指令的作用是标识源程序结束和指定程序运行时的起始地址。当程序被装入内存时，系统软件根据起始地址的段基值和偏移量分别被装入CS和IP中。

例如：

```
    ......
CODE   SEGMENT
       ASSUME  CS:CODE,.....
START: ......
       ......
CODE   ENDS
       END START
```

**(2)在程序运行期间，当执行某些指令时，CPU自动修改CS和IP，使它们指向新的代码段。**

例如：

**执行段间过程调用CALL和段间返回指令RET;**
**执行段间无条件转移指令JMP;**
**响应中断及中断返回指令;**
**执行硬件复位操作。**

# 4.6 过程定义伪指令（PROC/ENDP)

在程序设计过程中，常常将具有一定功能的程序段设计成一个子程序。在MASM宏汇编程序中，用过程(PROCEDURE)来构造子程序。

过程定义伪指令格式如下：

```
过程名   PROC  [NEAR/FAR]
          ......
          RET
          ......
过程名   ENDP
```

过程名是子程序的名称，它被用作过程调用指令CALL的目的操作数。它类同一个标号的作用。具有段、偏移量和距离三个属性。而距离属性使用NEAR和FAR来指定，若没有指定，则隐含为NEAR。

NEAR过程只能被本段指令调用，而FAR过程可以供其它段的指令调用。

每一个过程中必须包含有返回指令RET,其作用是控制CPU从子程序中返回到调用该过程的主程序。

# 4.7 当前位置计数器$与定位伪指令ORG(Origin)

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

在源程序中，使用符号$来表示位置计数器的当前值。因此，$被称为当前计数器。它位于不同的位置具有不同的值。

位置计数器$在使用上完全类似变量的使用.

定位伪指令**ORG**——用来改变位置计数器的值。

格式：　**ORG　数值表达式**

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

表达式的值必须为正值。表达式中也可以包含有当前位置计数器的现行值**$**。

```
DATA1    SEGMENT
            ORG  30H
    DB1   DB  12H,34H  ;DB1在DATA1段内的偏移量为30H
          ORG  $+20H;保留20H个字节单元，其后再存放'ABCD....
  STRING  DB  'ABCDEFGHI'
  COUNT   EQU   $-STRING;计算STRING的长度
  DB2  DW  $;  取$的偏移量,类似变量的用法
  DB3  DB   $     ;此语句错误!
DATA1    ENDS
  CODE      SEGMENT
            ASSUME  CS:CODE.....
      ORG  10H
  START:  MOV  AX,DATA
             MOV  DS,AX
             ......
  CODE    ENDS
            END START
```

# 4.8 标题伪指令TITLE

语句格式：**TITLE 标题名**

作用：给所在程序指定一个标题。以便在列表文件的每一页的第一行都显示这个标题。其中标题是用户任意选用的字符串，字符个数不能超过60。

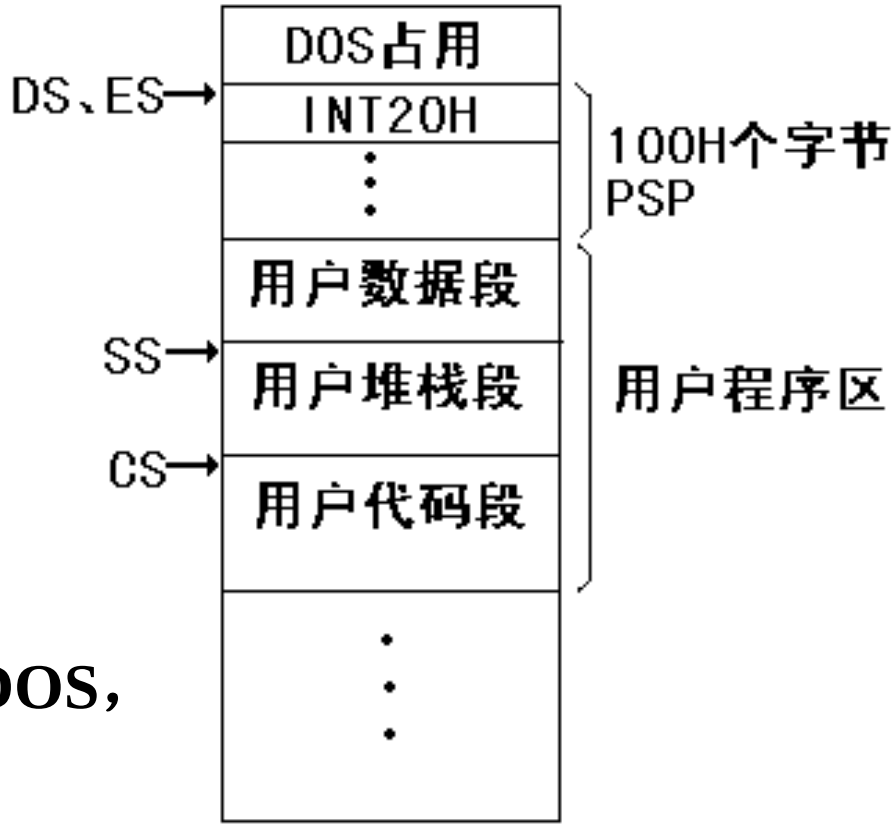
# 4.9 从程序返回操作系统的方法

为了使程序运行结束后，能够正确地返回到操作系统，需要在程序中加上一些必要的语句。一般有以下两种方法。

## 一、使用程序段前缀PSP（Program Segment Prefix)实现返回

DOS系统将一个.EXE文件（可执行文件）装入内存时，在该文件的前面生成一个程序段前缀PSP，其长度为100H字节。同时让DS和ES都指向PSP的开始，而CS指向该程序的代码段，即第一条可执行指令。

如图所示。PSP中一开始就是一条中断指令 INT 20H，执行该指令将终止用户程序，返回DOS系统。




为了使程序执行完后，正确返回DOS，需要做以下三个操作：

1. 将用户程序编制成一个过程,类型为FAR;

2. 将PSP的起始逻辑地址压栈,即将INT 20H指令的地址压栈;

3. 在用户程序结尾处,使用一条RET指令。执行该指令将使保存在堆栈中的PSP的起始地址弹出到CS和IP中。

程序结构：

```
DATA    SEGMENT
          …
DATA    ENDS
STACK1  SEGMENT STACK
          …
STACK1  ENDS
CODE    SEGMENT
BEGIN   PROC FAR
        ASSUME CS:CODE,DS:DATA,SS:STACK1
        PUSH DS
        MOV AX,0
        PUSH AX
        MOV AX,DATA
        MOV DS,AX
          …
        RET
BEGIN   ENDP
CODE    ENDS
        END BEGIN
```

# 二、使用DOS系统功能调用实现返回

执行DOS功能调用4CH，也可以控制用户程序结束，并返回DOS操作系统。

在程序结束时，使用两条指令：

```
MOV AH，4CH
INT  21H
```

代码段的结构为：

```
CODE   SEGMENT
       ASSUME  CS:CODE.....
BEGIN:MOV  AX,DATA
       MOV  DS,AX
       …
       MOV  AH,4CH
       INT  21H
CODE   ENDS
       END  BEGIN
```

# 第五章　程序控制结构及其设计技术

本章主要内容：

- 顺序程序设计
- 无条件转移指令
- 条件转移指令
- 分支程序设计
- 循环控制指令
- 循环程序设计

# §5.1　顺序程序设计

顺序程序是指程序的结构从开始到结尾一直是顺序执行，中途没有分支。

**例 5.2.1 试编写程序计算以下表达式:**

**Z=(3X+Y-5)/2**

**设X、Y的值放在字变量VARX、VARY中，结果存放在VARZ中。**

**算法分析:**

**1、乘$2^n$和除$2^n$可以使用算术左移和右移实现**

**2、其它非$2^n$的乘除运算可以用移位和加减组合运算来实现。如 3X可以分解成2X+X**

```
TITLE  EQUATION COMPUTE
DATA   SEGMENT
VARX   DW  15
VARY   DW  10
VARZ    DW  ?
DATA    ENDS
STACK1 SEGMENT PARA STACK
                DW 20H DUP (0)
STACK1   ENDS
CODE   SEGMENT
          ASSUME CS:CODE,DS:DATA,SS:STACK1
```

```
COMP  PROC  FAR
      PUSH  DS
      MOV  AX,0
      PUSH AX
      MOV AX,DATA
      MOV DS,AX
      MOV AX,VARX
      SHL AX,1              ;  2*X
      ADD AX，VARX    ;  3*X
      ADD AX，VARY    ;  3X+Y
      SUB  AX，5            ;  3*X+Y-5
      SAR  AX，1            ;  （3*X+Y-5）/2
      MOV VARZ，AX   ;  存结果
      RET
COMP  ENDP
CODE   ENDS
       END COMP
```

**例5.2.2 利用学号查学生的数学成绩表。**

**算法分析：首先在数据段中建立一个成绩表TABLE，在表中各学生的成绩按照学号从小到大的顺序存放。要查的学号存放在变量NUM中，查表的结果放在变量MATH中。**

```
    TITLE   TABLE      LOOKUP
DATA     SEGMENT
TABLE    DB   81,78,90,64,85,76,93,82,57,80
         DB   73,62,87,77,74,86,95,91,82,71
NUM      DB    8
MATH     DB    ?
DATA     ENDS
STACK1   SEGMENT  PARA  STACK
         DW   20H    DUP(0)
STACK1   ENDS
COSEG    SEGMENT
         ASSUME  CS:COSEG,DS:DATA,SS:STACK1
```

```
START:  MOV   AX,DATA
        MOV   DS,AX
        MOV   BX,OFFSET  TABLE    ;BX指向表首址
        XOR   AH,AH
        MOV   AL,NUM
        DEC   AL                  ；实际学号是从1开始的
        ADD   BX,AX               ；BX加上学号指向要查的成绩
        MOV   AL,[BX]             ；查到成绩送AL
        MOV   MATH,AL             ；存结果
        MOV   AH,4CH              ；返回DOS
        INT    21H
COSEG   ENDS
        END   START
```

在上述程序中，如果使用换码指令XLAT，可以简化程序。

换码指令格式为：

XLAT 表首址 ；功能为：AL<=((BX)+(AL))

其中表首址可以省略。

在XLAT指令执行前，要求将表首址的偏移量送入BX中，待查项与表首址之间的字节距离 (0~255)送入AL中。

# §5.2分支程序设计

分支程序结构是指程序的执行顺序将根据某些指令的执行结果，选择某些指令执行或不执行。

分支程序的实现主要是由转移指令完成。

## 一.转移指令

### 1. 无条件转移指令

格式：JMP 目标

目标是程序中的一个标号，表示转移指令所转移的目的地指令的地址。

程序结构：

```
        :
    JMP   TARGET
        :
TARGET:  .....
        :
```

根据目标所在的位置，JMP指令分为段内转移和段间转移。

## （1） 段内转移

JMP指令与转移目标位于同一个代码段内。转移时IP寄存器内容被改变，而CS保持不变。

目标地址可以有两种提供方法：

**A. 段内转移直接寻址---- 指令中直接给出转移目的地标号**

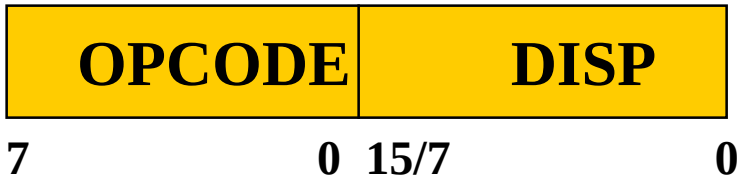
例如

```
        ……
        JMP  LABEL1
        ……
LABEL1: ……
```

或者

```
        ……
LABEL2: ……
        ……
        JMP  LABEL2
```

指令编码:




OPCODE字段长度为一个字节, DISP字段根据OPCODE字段为不同编码时分别为1个或2个字节。 DISP 为相对位移量，用补码表示。

OPCODE=
- 0EBH时， 为短转移， DISP为8位， 转移偏移量：-128—+127 字节
- 0E9H时， 为长转移， DISP为16位， 转移偏移量：-32768—+32767

指令的功能为： IP <=（ IP ）+ DISP

注意：转移偏移量是相对转移指令的下一条指令的起始地址

如果是相对于该转移指令的地址而言，则相对偏移量的值为：

-126~+129字节 或 -32765~+32770

**B. 段内转移间接寻址——指令中指定一个*16位*通用寄存器*或*字存储单元，其内容为转移目标地址。**

**例如：JMP CX**
**JMP WORD PTR [BX][SI]**

**指令编码格式：**


**指令执行时，由MOD、R/M以及位移量确定一个寄存器或有效地址EA，将所指寄存器或存储单元内容送入IP中。**

**IP<=(通用寄存器）** **或** **IP<=(EA)**

## （2）段间转移——JMP指令与目标地址不在同一个段内

执行该转移指令，将同时改变CS和IP的内容。

### A、段间转移直接寻址

在JMP指令中，目标地址符前面加属性说明符FAR。

例如：

```
COSEG1 SEGMENT                    COSEG2  SEGMENT
          :
        JMP  FAR PTR TARGET       TARGET：  .........
          :                             :
COSEG1  ENDS                      COSEG2  ENDS
```

指令编码格式：

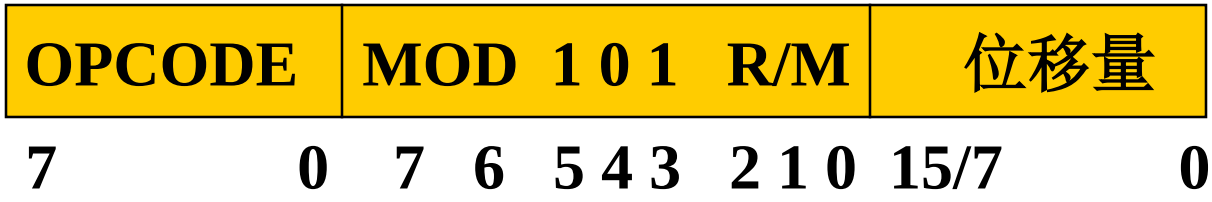

指令执行时，将有：

IP<= 目标地址偏移量

CS<=目标地址段基值

**B. 段间间接寻址——目标地址存放在一个双字存储单元中。低地址字单元内容为偏移量，高地址字单元内容为段基值。**

指令编码格式：




指令执行时，将有：

IP<= (EA)　　EA字单元内容

CS<= (EA+2)　　EA+2字单元内容

例：JMP DWORD PTR ADDR1；双字单元ADDR1的内容为转移目的地的偏移量和段基值。

JMP DWORD PTR [BX] ；由BX所指向的一个双字存储单元内容为转移目的地的偏移量和段基值。

## 2. 条件转移指令

8086/8088指令系统有18条条件转移指令

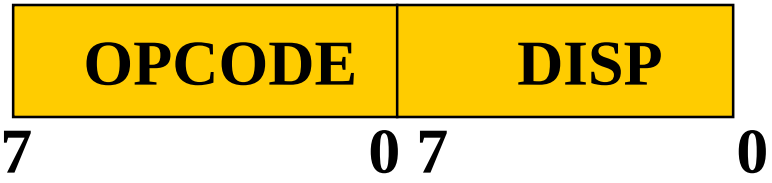
一般格式为：JXX 目标

其中：XX为1—2个字母组合，用来表示各种条件。

执行该指令时，若指定的条件成立，则转移至目标处。否则顺序执行。

条件用标志寄存器中的一个或几个标志位的状态来表示。

指令编码格式：




指令格式与段内无条件转移直接寻址指令的情况相似.但是，该指令中的DISP的长度为一个字节。 因此转移范围为 -128—+127字节。

条件转移指令分为三大类：

## （1）简单条件转移指令——条件为单个标志位的状态

| 标志位 | 指　　令 | 转移条件 | 含　　义 |
|---|---|---|---|
| CF | JC | CF=1 | 有进位/借位转移 |
| | JNC | CF=0 | 无进位/借位转移 |
| ZF | JE/JZ | ZF=1 | 相等/等于0转移 |
| | JNE/JNZ | ZF=0 | 不相等/不等于0转移 |
| SF | JS | SF=1 | 是负数转移 |
| | JNS | SF=0 | 是正数转移 |
| OF | JO | OF=1 | 有溢出转移 |
| | JNO | OF=0 | 无溢出转移 |
| PF | JP/JPE | PF=1 | 有偶数个1转移 |
| | JNP/JPO | PF=0 | 有奇数个1转移 |

（2）无符号数条件转移——在转移指令前执行了两个无符号数A和B相减的指令（A—B）

| 指　令 | 转移条件 | 含　义 |
|---|---|---|
| JA/JNBE | CF=0 且 ZF=0 | A > B 转移 |
| JAE / JNB | CF=0 或 ZF=1 | A ≥B 转移 |
| JB /JNAE | CF=1 且 ZF=0 | A < B 转移 |
| JBE / JNA | CF=1 或 ZF=1 | A ≤ B 转移 |

**(3) 带符号数条件转移指令——在转移指令之前执行了两个带符号数相减（A—B）的指令**

| 指令 | 转移条件 | 含　义 |
|---|---|---|
| JG / JNLE | SF=OF 且 ZF=0 | A > B 转移 |
| JGE / JNL | SF=OF 或 ZF=1 | A ≥B 转移 |
| JL / JNGE | SF≠OF 且 ZF=0 | A < B 转移 |
| JLE / JNG | SF≠OF 或 ZF=1 | A ≤ B 转移 |

对于带符号数的比较，需要使用符号标志位SF、溢出标志位OF和零标志位ZF来判断。下面以A>B的情况为例进行分析。

A>B可以分为以下几种情况：

1.A和B都为负数

若要A>B，则A-B的结果一定是正数（SF=0),也不会发生溢出（OF=0），且结果不为零（ZF=0）。

2. A和B都为正数

若要A>B，则A-B的结果一定是正数（SF=0），也不会发生溢出（OF=0），并且结果不为零（ZF=0）。

3.A为正数，B为负数

执行A-B的结果可能有两种情况：

（1）不发生溢出。这时结果为正数（SF=0），即有SF=OF。

（2）发生溢出。这时结果变为负数（SF=1），即有SF=OF。

# 二、分支程序设计

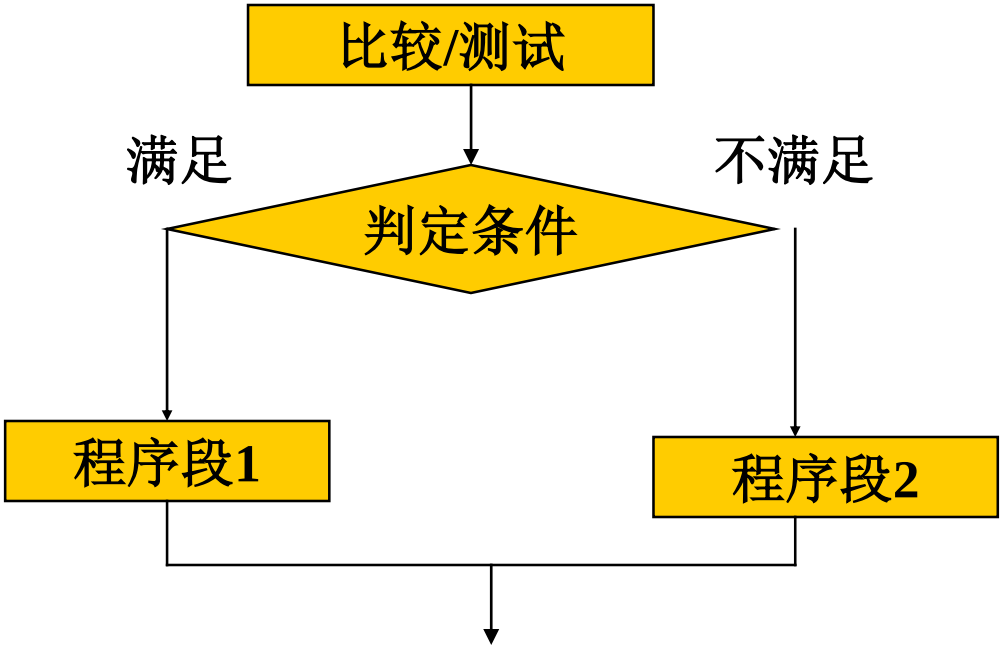
分支程序的结构有两种常见结构：

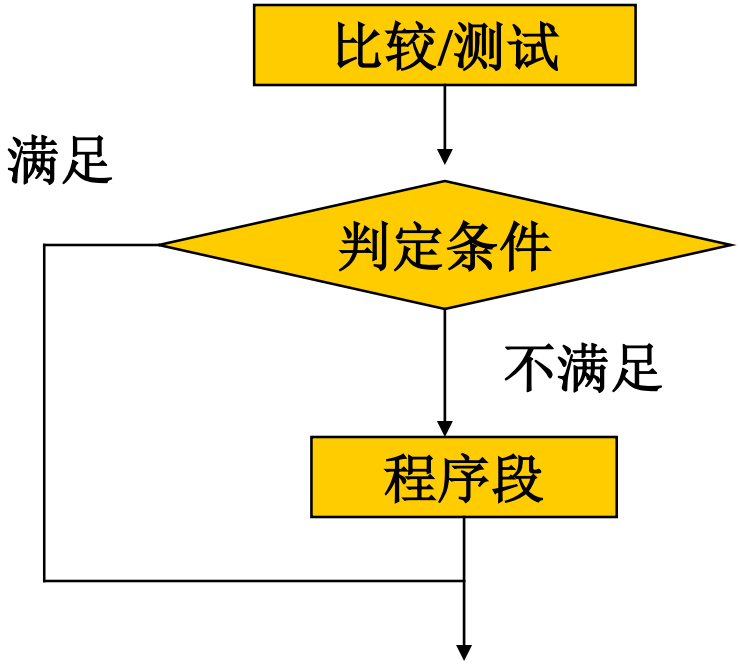
## 1、用比较/测试指令+条件转移指令实现分支

比较指令：**CMP DEST，SRC**

该指令的功能与减法指令SUB相似，区别是(DEST)-(SRC)的差值不送入DEST。而其结果影响标志位。

# 这种类型的分支程序有两种结构




一条条件转移指令只能实现两条分支程序的设计。要实现更多条分支的程序，需使用多条条件转移指令。

例5.3.2 数据段的ARY数组中存放有10个无符号数，试找出其中最大者送入MAX单元。

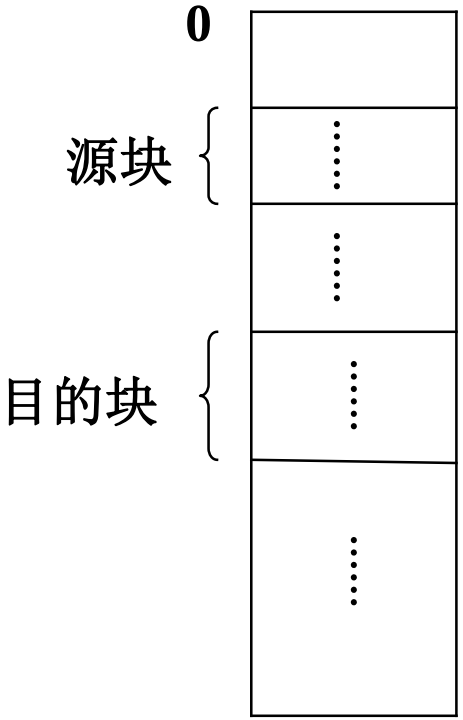
算法分析：

- 依次比较相邻两数的大小，将较大的送入AL中。
- 每次比较后，较大数存放在AL中，相当于较大的数往下传。
- 比较一共要做9次。
- 比较结束后，AL中存放的就是最大数。

```
DATA  SEGMENT
ARY   DB   17，5，40，0，67，12，34，78，32，10
MAX   DB   ？
DATA  ENDS
        ……
        MOV   SI，OFFSET ARY ；SI指向ARY的第一个元素
        MOV   CX，9          ；CX作次数计数器
        MOV   AL，[SI]       ；取第一个元素到AL
LOP：   INC   SI             ；SI指向后一个元素
        CMP   AL，[SI]       ；比较两个数
        JAE   BIGER          ；前元素≥后元素转移
        MOV   AL，[SI]       ；取较大数到AL
BIGER： DEC   CX             ；减1计数
        JNZ   LOP            ；未比较完转回去，否则顺序执行
        MOV   MAX，AL        ；存最大数
        ……
```

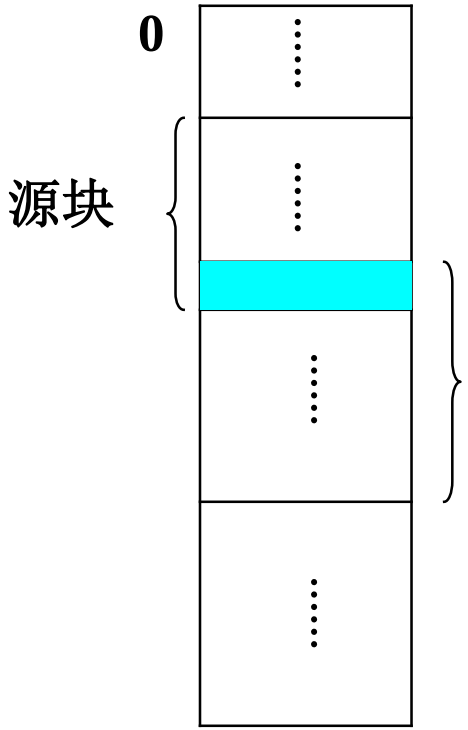
例5.3.4　编写一程序，实现将存储器中的源数据块传送到目的数据块。

在存储器中两个数据块的存放有三种情况：两个数据块分离和有部分重叠。



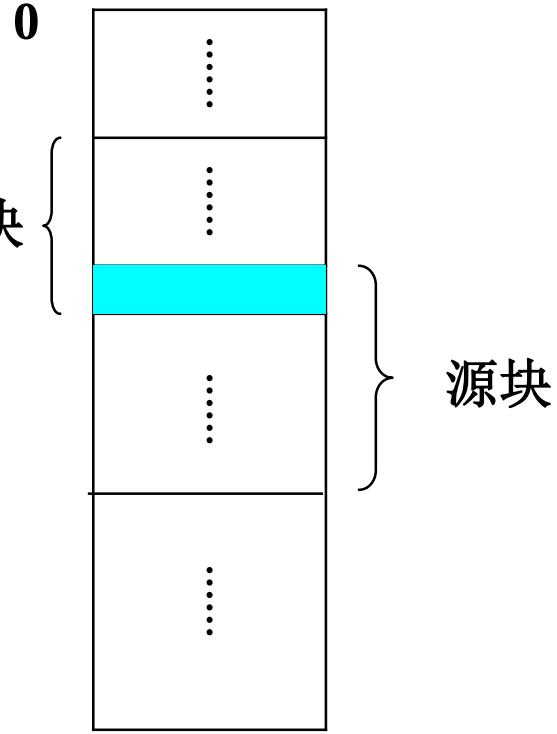

两个数据块分离

可以从首址或末址开始传送




源块首址 < 目的块首址

必须从数据块末址开始传送




源块首址 > 目的块首址

必须从数据块首址开始传送

三种相对位置情况的传送方法：

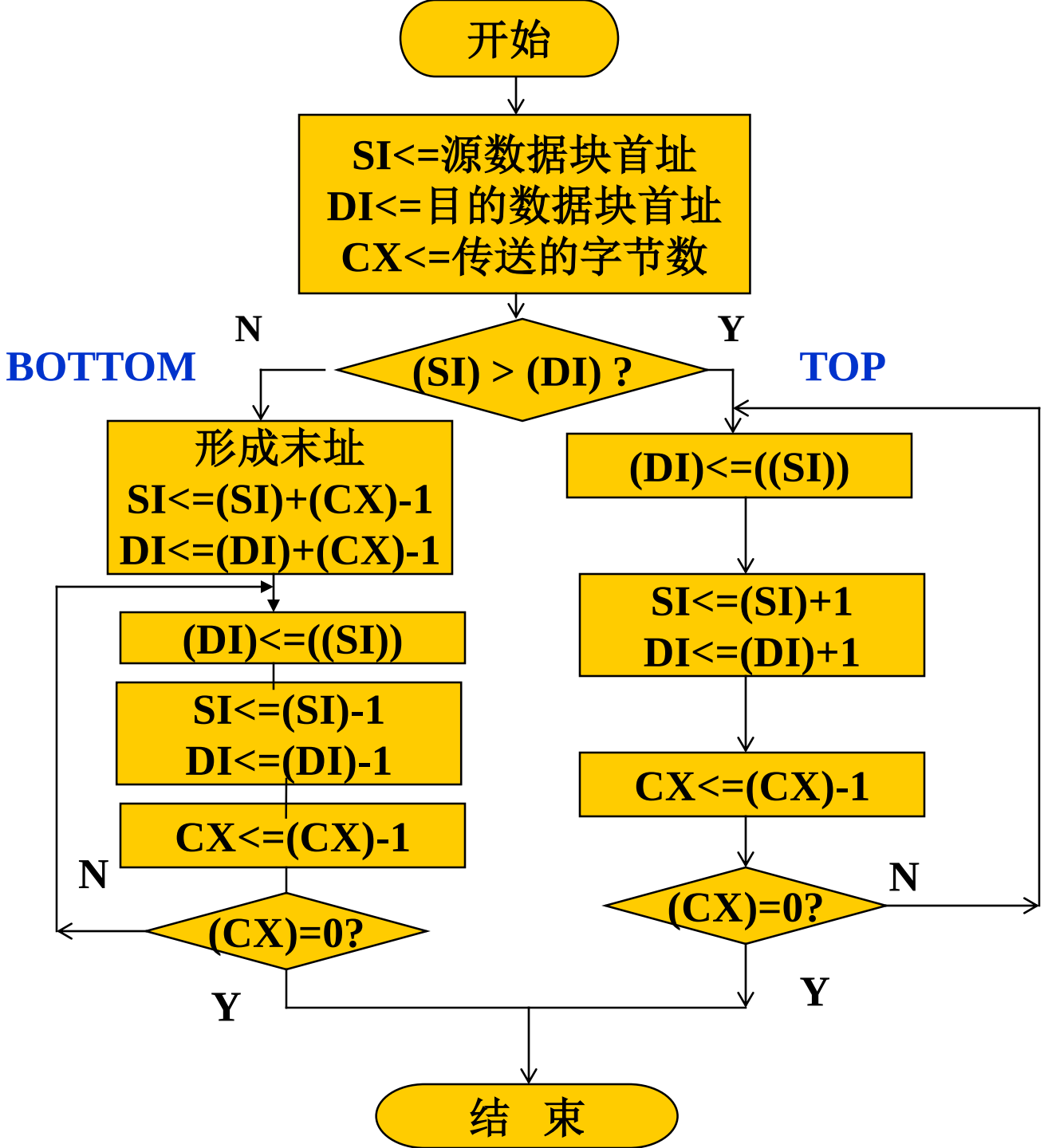
对于源块和目的块分离的情况，不论是从数据块的首址还是末址开始传送都可以。

对于源块与目的块有重叠且源块首址<目的块首址的情况，必须从数据块末址开始传送。

对于源块与目的块有重叠且源块首址>目的块首址的情况，必须从数据块首址开始传送。

将上述三种情况综合起来，只考虑源块和目的块的地址相对大小，传送方法如下：

当源块首地址<目的块首地址时，从数据块末地址开始传送。反之，则从首地址开始传送。


开始
SI<=源数据块首址
DI<=目的数据块首址
CX<=传送的字节数
N
Y
BOTTOM
(SI) > (DI) ?
TOP
形成末址
SI<=(SI)+(CX)-1
DI<=(DI)+(CX)-1
(DI)<=((SI))
SI<=(SI)-1
DI<=(DI)-1
CX<=(CX)-1
N
(CX)=0?
Y
(DI)<=((SI))
SI<=(SI)+1
DI<=(DI)+1
CX<=(CX)-1
N
(CX)=0?
Y
结 束

```
 TITLE  DATA BLOCK MOVE
DATA   SEGMENT
            ORG  $+20H
STRG      DB   'ABCDEFGHIJ' ;数据块
LENG    EQU  $-STRG            ;数据块字节长度
BLOCK1 DW    STRG              ;源块首址
BLOCK2 DW   STRG-5          ;目的块首址
DATA   ENDS
STACK1 SEGMENT STACK
              DW   20H DUP(0)
STACK1 ENDS
```

```
COSEG  SEGMENT
        ASSUME CS:COSEG,DS:DATA,SS:STACK1
BEGIN:  MOV AX, DATA
        MOV DS,AX
        MOV CX,LENG        ;设置计数器初值
        MOV SI,BLOCK1      ;SI指向源块首址
        MOV DI,BLOCK2      ;DI指向目的块首址
        CMP  SI,DI         ;源块首址>目的块首址吗?
        JA  TOP            ;大于则转到TOP处，否则顺序执行
        ADD SI,LENG-1      ;SI指向源块末址
        ADD DI,LENG-1      ;DI指向目的块末址
```

```
BOTTOM: MOV AL，[SI]        ;从末址开始传送
        MOV [DI], AL
        DEC SI
        DEC DI
        DEC CX
        JNE   BOTTOM
        JMP   END1
  TOP: MOV AL,[SI]          ;从首址开始传送
        MOV [DI],AL
        INC SI
        INC DI
        DEC CX
        JNE TOP
END1:  MOV AH,4CH
        INT 21H
COSEG  ENDS
        END BEGIN
```

## 2、用跳转表形成多路分支

当程序的分支数量较多时，采用跳转表的方法可以使程序长度变短， 跳转表有两种构成方法：

### （1）跳转表用入口地址构成

在程序中将各分支的入口地址组织成一个表放在数据段中，在程序中通过查表的方法获得各分支的入口地址。

**例5.3.5 设某程序有10路分支，试根据变量N的值（1~10），将程序转移到其中的一路分支去。**

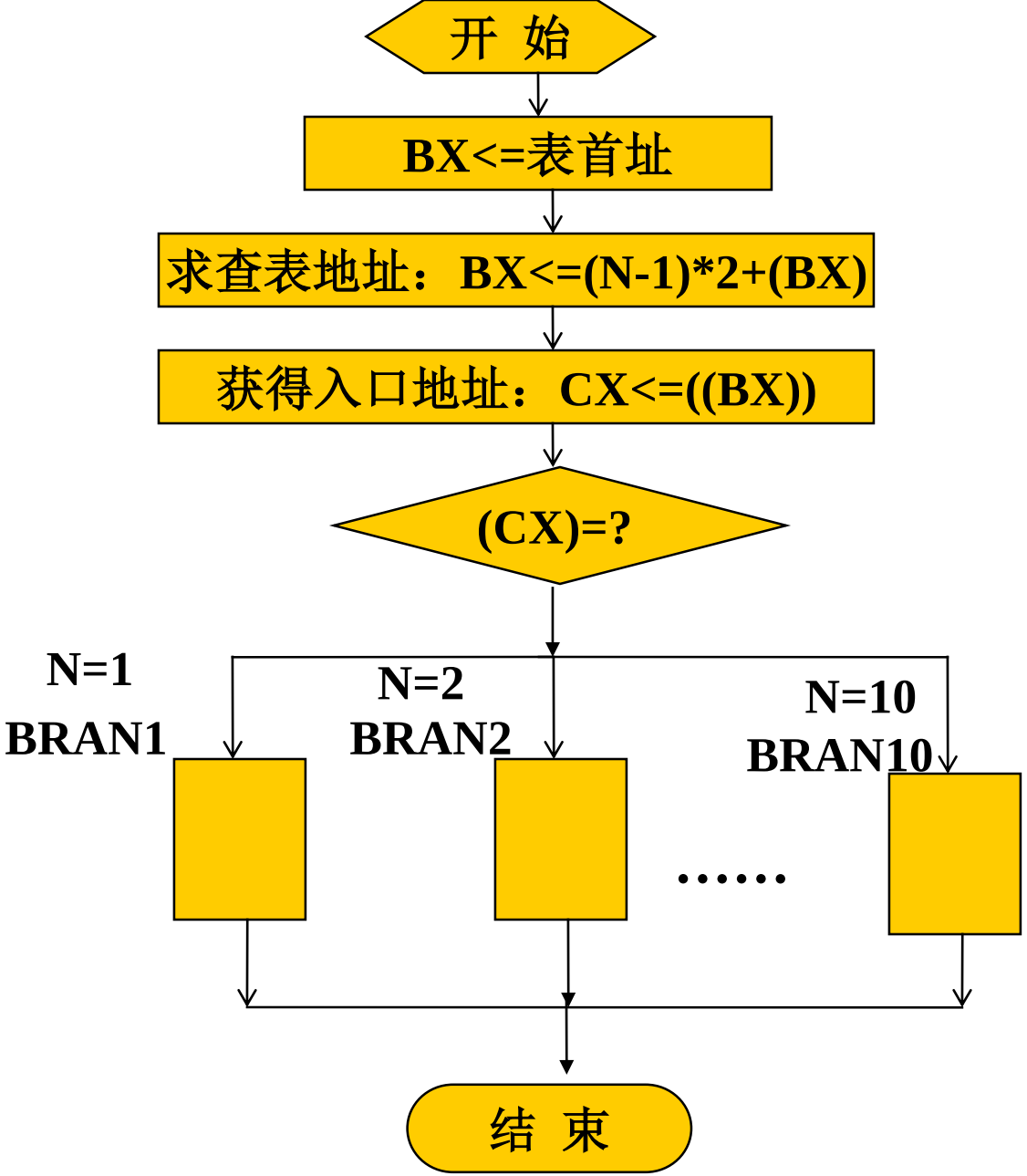
跳转表

**设10路分支程序段的入口地址分别为：BRAN1、BRAN2.....BRAN10。**

**当变量N为1时，转移到BRAN1；N为2时，转移到BRAN2，依次类推。**

**在跳转表中每两个字节存放一个入口地址的偏移量，如右图所示。**

**程序中，先根据N的值形成查表地址：（N-1）×2+表首址。**


开 始
BX<=表首址
求查表地址：BX<=(N-1)*2+(BX)
获得入口地址：CX<=((BX))
(CX)=?
N=1
BRAN1
N=2
BRAN2
N=10
BRAN10
……
结 束

多路分支结构流程图

```
        TITLE   JUMP TABLE OF ADDRESS
DATA    SEGMENT
ATABLE  DW      BRAN1，BRAN2，BRAN3，...，BRAN10
N       DB      3
DATA    ENDS
STACK1  SEGMENT PARA STACK
        DW      20H DUP（0）
STACK1  ENDS
CODE    SEGMENT
        ASSUME  CS：CODE，DS：DATA，SS：STACK1
START：  MOV     AX，DATA
        MOV     DS，AX
          ...
```

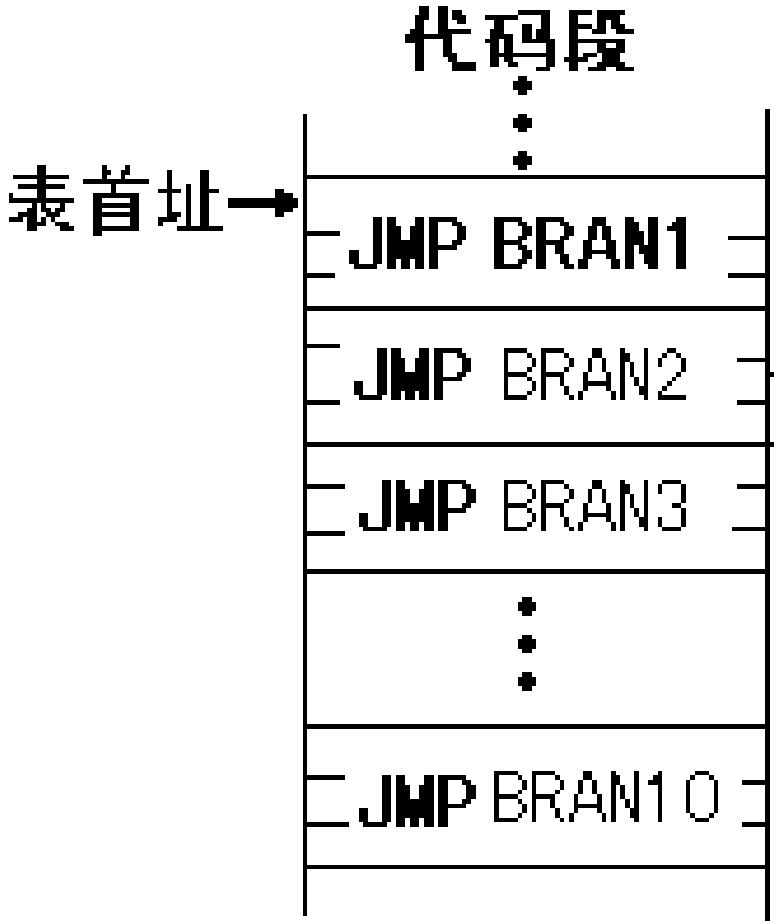
```
XOR    AH，AH
MOV    AL，N
DEC     AL
SHL     AL，1
MOV     BX，OFFSET ATABLE ；BX指向表首址
ADD     BX，AX          ；BX指向查表地址
MOV     CX，[BX]        ；将N对应的分支入口地址送到CX中
JMP     CX              ；转移到N对应的分支入口地址
```

```
BRAN1：  ⋮
         JMP    END1
BRAN2：  ⋮
         JMP    END1
BRAN3：  ⋮
         JMP    END1
           ⋮
BRAN10：⋮

END1：  MOV    AH，4CH
         INT    21H
CODE     ENDS
         END    START
```

## （2）跳转表用无条件转移指令构成

跳转表的每一个项目就是一条无条件转移指令。这时跳转表是代码段中的一段程序。



例 5.3.5的源程序可修改为如下程序：

```
TITLE  JUMP TABLE OF INSTRUCTION
DATA     SEGMENT
N          DB  3
DATA     ENDS
STACK1  SEGMENT PARA STACK
           DW   20H DUP(0)
STACK1  ENDS
CODE     SEGMENT
          ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:  MOV  AX,DATA
           MOV  DS,AX
             ...
           MOV  BH,0
           MOV  BL,N
```

```
        DEC   BL              ;四条指令实现(N-1)*3
        MOV  AL,BL
         SHL  BL,1
        ADD  BL,AL
        ADD  BX,OFFSET  ITABLE ;BX指向查表地址
        JMP  BX               ;转移到N对应的JMP指令
ITABLE: JMP  BRAN1            ;JMP指令构成的跳转表
        JMP  BRAN2
        JMP  BRAN3
               :
               :
        JMP  BRAN10
```

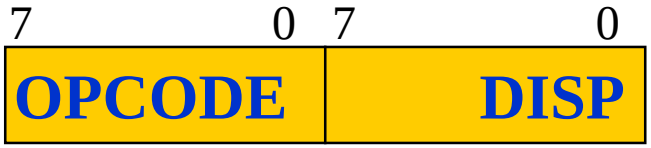
```
BRAN1: ...
          :
          JMP  END1
BRAN2: ...
          :
          JMP  END1
          :
          :
BRAN10: ...
          :
          :
END1:  MOV  AH,4CH
       INT  21H
CODE  ENDS
       END  START
```

# 5.3 循环程序设计

## 一、循环控制指令

**8086/8088指令系统中有4条循环控制指令，长度都是2字节。**

**指令编码格式为：**




**DISP：8位补码表示本指令的下一条指令的首址与目标单元之间的字节距离。**

**指令中指定一定的条件，若条件满足，则将DISP加入到IP中，即IP<=(IP)+DISP使程序转移到目的指令执行。**

**指令使用CX寄存器做循环计数。循环控制指令的执行对标志位没有影响。**

## 1、LOOP指令

格式： LOOP 目标

其中目标是程序中的一个标号。

执行一次LOOP指令将使：CX <= (CX)—1

若（CX）≠0，则转到目标处执行，否则顺序执行。

例 5.4.1 在例5.3.2中， 数据段的ARY数组中存放有10个无符号数，试找出其中最大者送入MAX单元。若使用循环指令，则程序可修改如下：

源程序结构如下：

```
DATA    SEGMENT
ARY     DB  17,5,40,0,67,12,34,78,32,10
LEN     EQU  $-ARY
MAX     DB  ?
DATA    ENDS
          :
          :
        MOV  SI,OFFSET  ARY ;SI指向ARY的第一个元素
        MOV  CX,LEN-1       ;  CX作循环(比较)次数计数
        MOV  AL,[SI]
LOP:    INC  SI
        CMP  AL,[SI]
        JAE  BIGER
        MOV  AL,[SI]
BIGER:LOOP LOP
        MOV  MAX,AL
          ……
```

## 2、LOOPE / LOOPZ指令

格式：LOOPE 目标 或 LOOPZ 目标

指令执行：CX <= (CX) —1，若(CX) ≠0且ZF=1，则转到目标处执行，否则顺序执行。

例 5.4.2 编写一程序，在一字符串中查找第一个非空格字符，并将其在字符串中的序号（1～n)送入INDEX单元中。若未找到，则将INDEX单元置为全1。

```
DATA     SEGMENT
STRG      DB ‘      CHECK NO_SPACE’
LENG      EQU $－STRG
INDEX     DB ?
DATA      ENDS
STACK1   SEGMENT  PARA STACK
          DW  20H DUP(0)
STACK1   ENDS
```

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:  MOV  AX,DATA
        MOV  DS,AX
        MOV  CX,LENG    ;字符串长度送入CX
        MOV  BX,－1     ;设地址指针初值
NEXT:   INC  BX
        CMP  STRG[BX],' '
        LOOPE NEXT   ;是空格字符且计数不为0，继续查找
        JNZ  FOUND     ;找到非空格字符，转FOUND
        MOV  BL，0FEH   ;未找到非空格字符
FOUND:  INC  BL              ;使位置序号从1开始
        MOV  INDEX,BL   ;存结果
        MOV  AH,4CH
        INT  21H
CODE    ENDS
        END  START
```

## 3、LOOPNE / LOOPNZ指令

使用格式：LOOPNE 目标　或 LOOPNZ 目标

指令执行：CX <= (CX) —1，若(CX) ≠0且ZF=0，则转到目标处执行，否则顺序执行。

例 5.4.3 编写程序，计算两个字节数组ARY1和ARY2对应元素之和，一直计算到两数之和为0或数组结束为止。并将和存入数组SUM中，将该数组的长度存放在NUM单元中。

源程序如下：

```
DATA      SEGMENT
ARY1      DB 12,10,3,5,-1,7,34,8,9,10
ARY2      DB 14,23,6,-2,1,9,45,21,8,24
LENG      EQU  ARY2-ARY1
SUM       DB LENG DUP(?)
NUM       DB ?
DATA      ENDS
STACK1  SEGMENT PARA STACK
          DW 20H DUP(0)
STACK1  ENDS
CODE      SEGMENT
          ASSUME CS:CODE,DS:DATA,SS:STACK1
BEGIN:   MOV AX,DATA
          MOV DS,AX
```

```
        MOV CX,LENG
        MOV BX,－1   ;设置指针初值
NZERO:  INC BX
        MOV AL,ARY1[BX]  ;取被加数
        ADD AL,ARY2[BX]
        MOV SUM[BX]，AL
        LOOPNE NZERO  ；和不为0转到NZERO处
        JZ  ZERO      ；和为0转到ZERO处
        INC BL
ZERO：  MOV NUM，BL  ；存结果数组长度
        MOV AH，4CH
        INT 21H
CODE    ENDS
        END BEGIN
```

## 4、JCXZ指令

指令格式：JCXZ　目标

该指令测试CX的内容是否为0，如果（CX）=0，则转移到目标处指令，否则顺序执行。

该指令相当于条件转移指令。它一般用在一个循环的开始，当循环次数计数寄存器CX为0时，就不执行该循环。如果没有这个控制，将使得循环次数变得非常大（0-1=0FFFFH），从而产生错误结果。

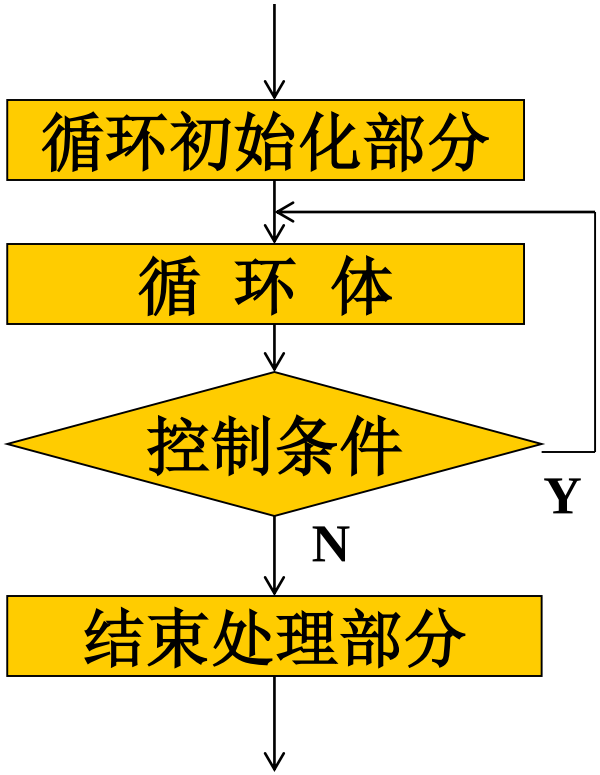
程序结构为：

```
        ..........
        MOV  CX,COUNT
        JCXZ   NEXT
LOP:    ..........
        LOOP   LOP
NEXT:  ..............
```

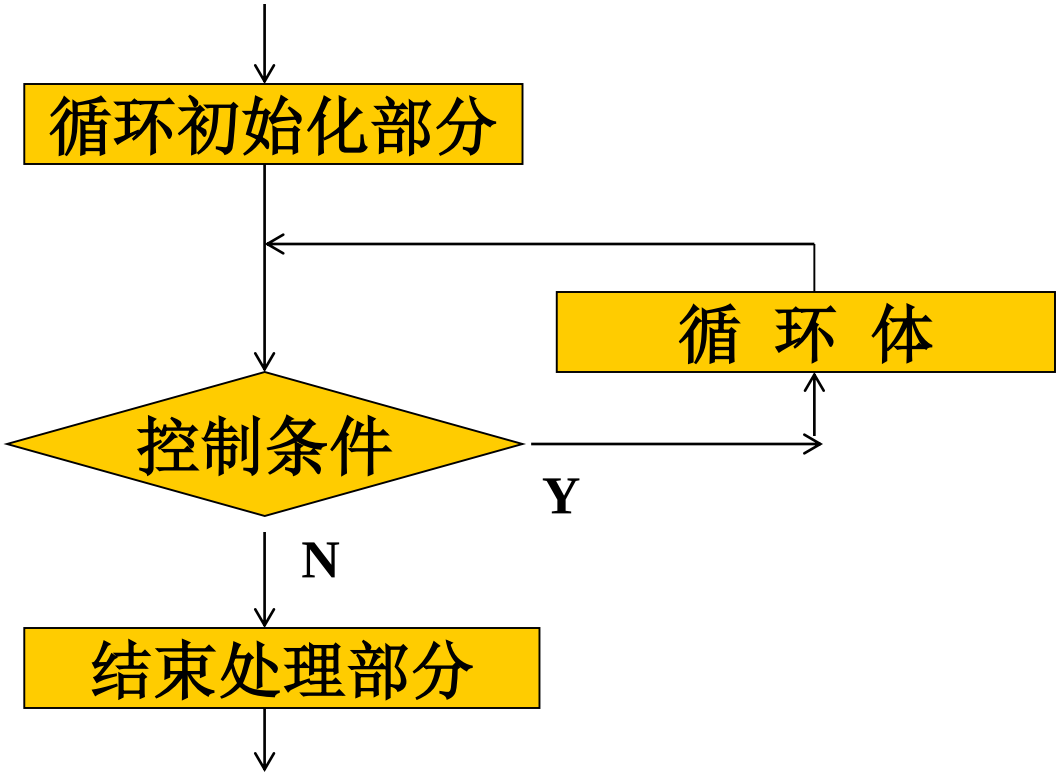
# 二、循环程序的结构

循环程序有两种结构形式

## 1、先执行后判断结构




## 2、先判断后执行结构



在循环程序中主要包括以下四个部分：

## 1、初始化部分

用于建立循环的初始状态。包括：循环次数计数器、地址指针以及其它循环参数的初始设定。

## 2、循环体

循环程序完成的主要任务。包括工作部分和修改部分。

工作部分：是完成循环程序任务的主要程序段。

修改部分：为循环的重复执行，完成某些参数的修改。

## 3、循环控制部分

判断循环条件是否成立。可以有以下两种判断方法：

（1）用计数控制循环——循环次数已知

（2）用条件控制循环——循环次数未知

## 4、结束处理部分

处理循环结束后的结果。如存储结果等。

# 三、单循环程序设计

单循环程序的循环体由顺序结构或分支结构组成

## 1、计数控制循环

常选用CX作计数器，可选用LOOP、LOOPE或LOOPNE等循环控制指令。

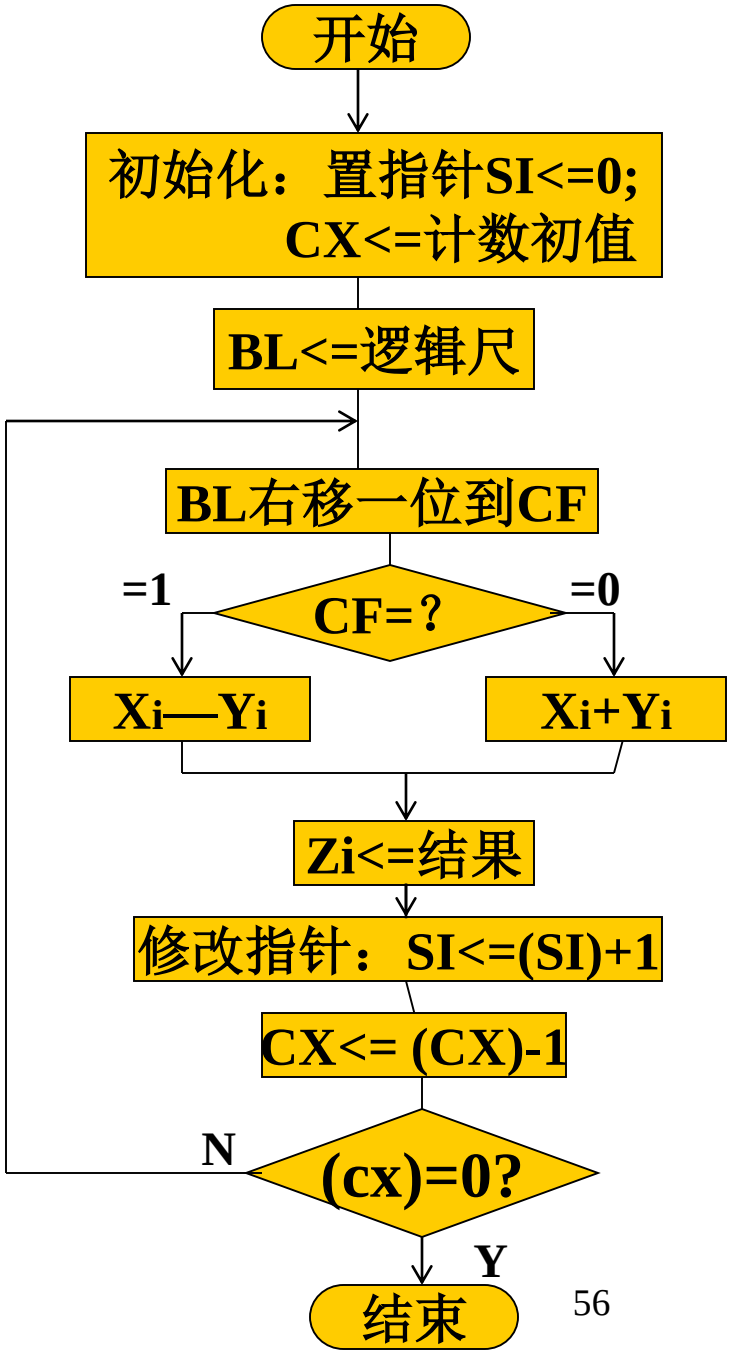
例 5.4.4 设有两个数组X和Y，它们都有8个元素，其元素按下标从小到大的顺序存放在数据段中。试编写程序完成下列计算：

Z1=X1+Y1　Z2=X2-Y2　Z3=X3+Y3

Z4=X4-Y4　Z5=X5-Y5　Z6=X6+Y6

Z7=X7+Y7　Z8=X8-Y8

由于循环体中有“+”和“-”两种可能的运算，通过设置标志0和1来判断。八个运算表达式由8位逻辑尺：10011010B来识别。



```
DATA    SEGMENT
X       DB  0A2H,7CH,34H,9FH,0F4H,10H,39H,5BH
Y       DB  14H,05BH,28H,7AH,0EH,13H,46H,2CH
LEN     EQU $ —Y
Z       DB  LEN DUP(?)
LOGR    DB  10011010B
DATA    ENDS
STACK0 SEGMENT PARA STACK
       DW  20H DUP(0)
STACK0 ENDS
COSEG   SEGMENT
        ASSUME CS:COSEG,DS:DATA,SS:STACK0
BEGIN:   MOV AX,DATA
         MOV DS,AX
         MOV CX,LEN  ;  初始化计数器
         MOV SI,0    ;  初始化指针
         MOV BL,LOGR ;初始化逻辑尺
```

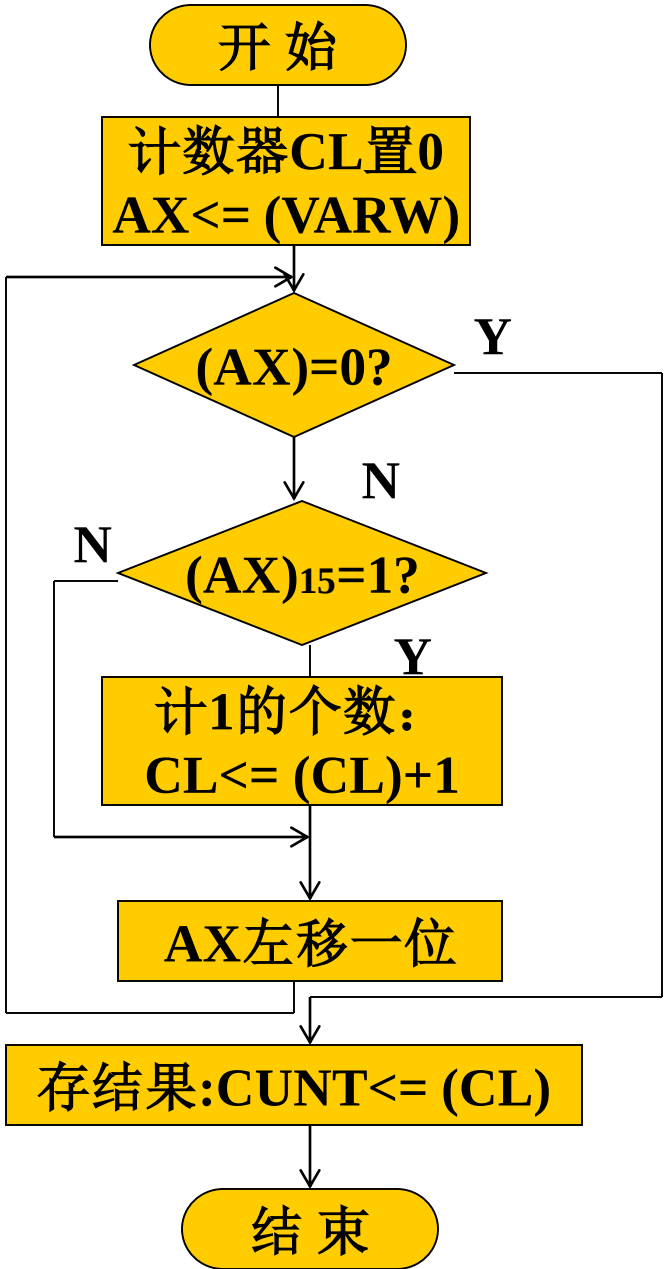
```
LOP:   MOV  AL,X[SI]
       SHR   BL,1          ;标志位送CF
       JC      SUB1        ；为1，转做减法
       ADD   AL,Y[SI] ；为0，做加法
       JMP     RES
SUB1:  SUB    AL,Y[SI]
RES:    MOV   Z[SI],AL ；存结果
        INC     SI          ；修改指针
        LOOP   LOP
        MOV   AH,4CH
        INT      21H
COSEG  ENDS
          END BEGIN
```

# 2、条件控制循环

例 5.4.5 编写一程序，将用二进制数表示的字单元VARW 中“1”的个数统计出来，存入CONT单元中。

本例中通过将字单元各位逐位移入最高位来判断，而最高位即符号位。为了减少循环次数，循环中加上了判断各位是否全为0，这样可使低位为全0时的循环次数减少。（优化性能，效率高）

其它算法？



```
DATA    SEGMENT
VARW    DW   1101010010001000B
CONT     DB   ?
DATA    ENDS
STACK1  SEGMENT PARA STACK
            DW  20H DUP(0)
STACK1  ENDS
CODE     SEGMENT
           ASSUME  CS:CODE,DS:DATA,SS:STACK1
BEGIN:   MOV  AX,DATA
           MOV  DS,AX
           MOV  CL,0
           MOV  AX,VARW
LOP:      TEST AX,0FFFFH  ；测试（AX）是否为0
           JZ   END0                ；为0，循环结束
           JNS  SHIFT               ；判最高位，为0则转SHIFT
           INC  CL                  ；最高位为1，计数
```

```
SHIFT:   SHL AX,1
         JMP LOP
END0:    MOV  CONT,CL ；存结果
         MOV  AH,4CH
         INT  21H
CODE     ENDS
         END  BEGIN
```

# 四、多重循环程序设计

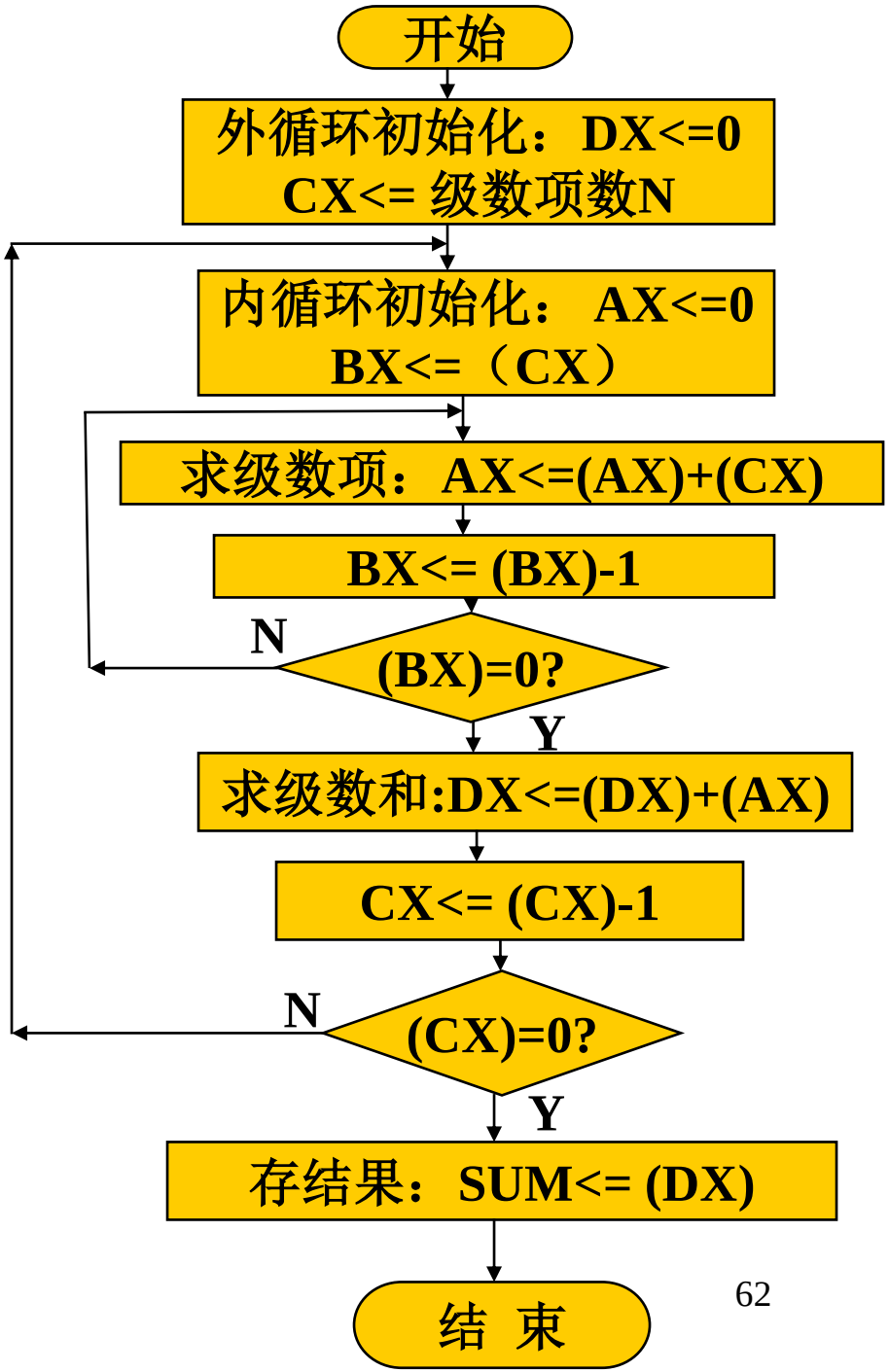
多重循环结构是指循环程序的循环体中又包含了另一个循环

例5.4.6 编写一程序，求级数$1^2+2^2+3^2+\ldots$的前N项和。

对于$N^2$的计算采用连加的方法，即：

$$N^2 = N \times N = \underbrace{N+N+\ldots+N}_{N}$$

本题程序采用双重循环（顺序:外循环倒序）。内循环计算级数各项的值，外循环计算各级数项之和。



```
DATA   SEGMENT
SUM    DW  ?
N      DB  20
DATA   ENDS
STACK1 SEGMENT PARA STACK
       DW 20H DUP(0)
STACK1 ENDS
CODE   SEGMENT
    ASSUME CS:CODE,DS:DATA,SS:STACK1
START: MOV AX,DATA
    MOV DS,AX
    MOV DX,0
    MOV CX,0
    MOV CL,N ;设置外循环次数
LOP1:  MOV AX,0
    MOV BX,CX;  设置内循环次数
```

```
LOP2: ADD AX,CX；求级数项的值
      DEC BX
      JNZ LOP2；BX计数不为0，继续内循环
      ADD DX,AX ；累加级数项
      LOOP  LOP1；CX计数不为0，继续循环
      MOV SUM,DX；存级数和
      MOV AH,4CH
      INT 21H
CODE  ENDS
       END  START
```

# 第六章 子程序设计

本章主要内容：

- 子程序调用与返回指令
- 编制子程序的基本要求
- 子程序设计举例
- DOS 功能子程序调用

子程序——在一个程序的不同的地方需要多次使用的某个程序段，将其进行独立编制。

调用与返回：在主程序中需要使用该功能时，就转移到子程序执行，执行完后又返回原程序继续执行。这样的程序结构称为子程序设计。

## §6.1 调用与返回指令

在汇编语言中，子程序是以“过程”的形式表示。根据被调用过程与调用程序是否在同一个段内，可以分为两种情况。

## ➢段内调用与返回

主程序与子程序同在一个段内。这时，子程序的调用与返回只需修改指令指针IP。

右图中指令CALL PROCA就是段内调用。

```
CODEA SEGMENT
        …
        CALL PROCA
AAA:    …
        …
PROCA   PROC
        …
        RET
        …
PROCA   ENDP
        …
CODEA   ENDS
```

## ➢段间调用与返回

调用指令与子程序分别在不同的段，这时，需要同时修改CS和IP。

下面CODEB段中的CALL FAR PTR PROCB就是段间调用。

```
CODEA SEGMENT                    CODEB  SEGMENT
      ...                              ...
PROCB   PROC ...                       CALL  FAR PTR PROCB
      ...                              BBB:      ......
          RET
PROCB   ENDP                     CODEB  ENDS
        ...
CODEA  ENDS
```

## 1、调用指令

指令格式：　CALL　过程名

执行CALL指令时，先将断点压入堆栈中保存，然后转移到目标单元。

断点是调用子程序指令CALL的下一条指令的地址。前述程序结构图中，AAA和BBB就是两条调用子程序指令的断点。

CALL指令的执行对各标志位无影响。

## （1）段内调用

### (a) 段内直接调用

汇编指令书写格式为在 CALL 之后直接书写过程名

例如：CALL SUB1

### （b）段内间接调用

子程序的起始地址（偏移量）由一个通用寄存器或一个字存储单元提供。

```
例如：CALL  BX
      CALL  CX
      CALL  WORD PTR  30H[BX][SI]
```

## （2）段间调用

### （a）段间直接调用

由于在定义过程时，对提供段间调用的过程，已经说明其属性为FAR。因此调用时，在CALL后直接书写过程名，也可以在过程名前面加FAR属性修饰。

例如： CALL PROC_NAME

CALL FAR PTR PROC_NAME

### (b) 段间间接调用

调用指令提供一个双字存储单元的地址，它所指向的双字存储单元内容为被调用过程的起始地址。其中，两个低字节存放偏移量，两个高字节存放段基值。

例如：CALL DWORD PTR DISP[BX][DI]

## （3）子程序调用指令与转移指令JMP的区别

两者都是无条件转移到目标单元，但CALL指令要保存“断点”，而JMP指令不保存断点。

## 3、返回指令

一个子程序最后执行的指令一定是返回指令，但不一定是最后一条指令。

根据子程序调用指令的使用情况，返回指令也分为段内返回和段间返回。其汇编指令书写形式都是**RET**，但它们的编码是不相同的。

（1）段内返回——指令编码为 **C3H**

执行该指令，将从堆栈顶部弹出一个字送入**IP**。

（2）段间返回——指令编码为 **CBH**

执行该指令，将从堆栈顶部弹出两个字分别送**IP**和**CS**中。

## （3）带弹出值的返回指令

汇编指令格式为：**RET n**

其中n为一个立即数，长度为2字节。并且是一个偶数。

这条指令也分为段内返回和段间返回，它们的指令编码不同，分别为C2 n和CA n。

指令执行过程：

（1）从堆栈弹出1个字送IP（段内返回）或2个字送IP和CS；

（2）执行 SP<=(SP)+n 。将堆栈中已经用过的参数(n个字节)弹出舍去.

# §6.2 编制子程序的基本要求

## 1、具有一定的通用性

选择和设计好子程序所需的各种入口参数和出口参数。

## 2、选择适当的参数传递方法

在主程序与子程序之间传递参数，可以选择的方法有：

A、使用通用寄存器

B、使用指定的存储单元

C、使用堆栈

## 3、注意信息保护

如果在子程序中需要使用某些寄存器或存储单元，为了不破坏它们原来在主程序中的值，为此需要进行信息保护。

信息的保护可以有两种方法：

A、在主程序中保存子程序中将要使用的一些寄存器的内容

```
...
PUSH  BX
PUSH  CX
CALL  SUB1
POP   CX
POP   BX
...
```

## B、在子程序中保存将要使用的一些寄存器的内容

```
SUB2  PROC
      PUSH  BX
      PUSH  CX
       ......  ;  完成子程序功能指令序列
      POP   CX
      POP   BX
      RET
SUB2  ENDP
```

## 4、正确使用堆栈

由于堆栈中保存着主程序调用子程序时的断点地址。若在子程序中也使用了堆栈，注意各个数据压栈和出栈的顺序不能错，否则将导致数据错误和子程序返回地址错误。

## 5、编制子程序文件

子程序文件应包括文字说明与子程序本身两个部分。而文字说明一般包括：

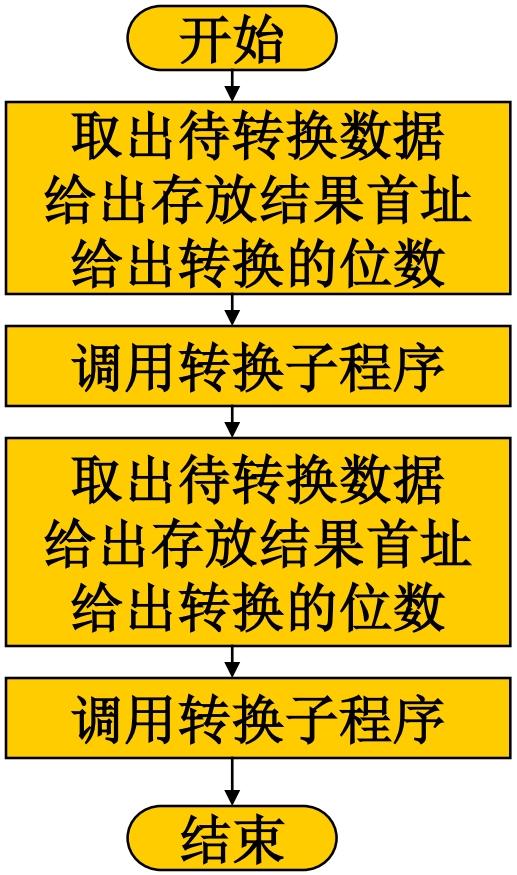
- 子程序名
- 子程序功能描述
- 子程序的入口参数与出口参数
- 使用哪些寄存器和存储单元
- 本子程序是否又调用其它子程序
- 子程序的调用形式、举例

# §6.3 子程序设计举例

例 5.5.1 将两个给定的二进制数(8位和16位)转换为ASCII码字符串。

主程序提供被转换的数据和转换后的ASCII码字符串的存储区的首地址

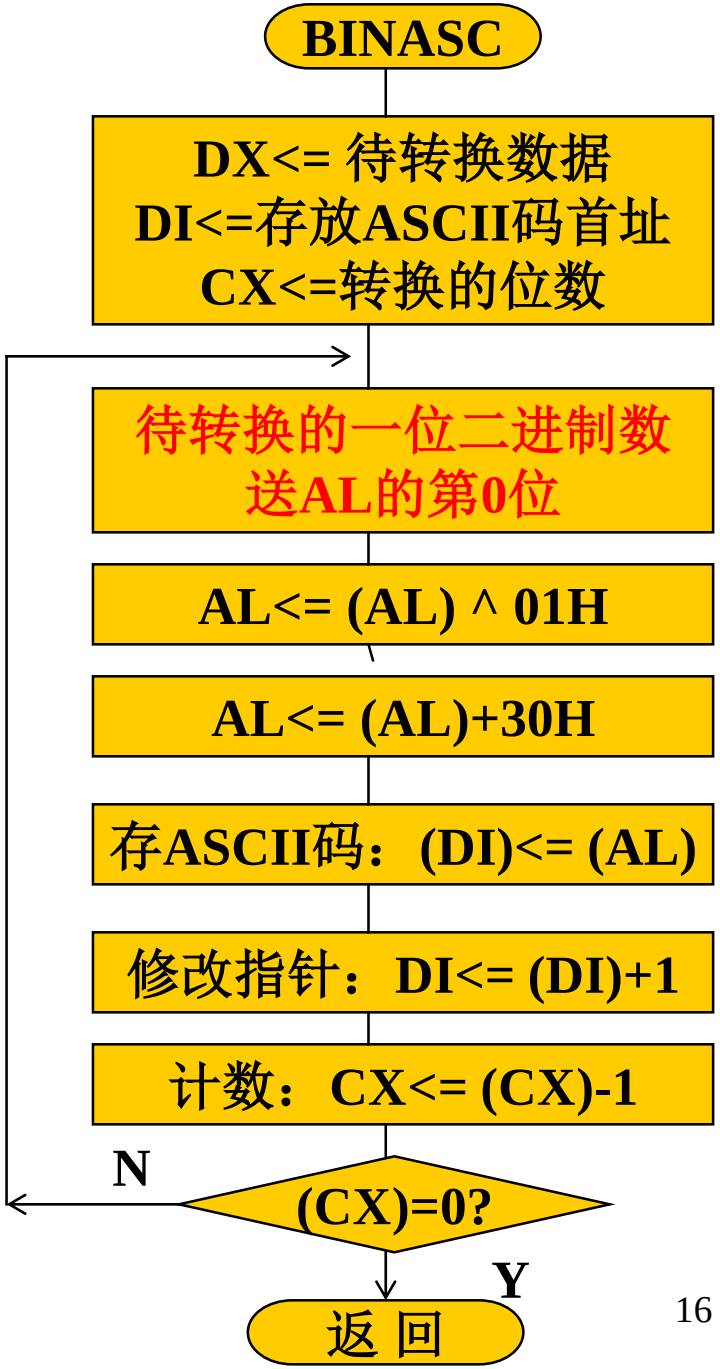
主程序框图



子程序框图：

子程序完成二进制数与ASCII码字符串的转换。子程序的入口参量有：被转换的数据、存储ASCII码字符串的首址和被转换数据的位数。无出口参量。

‘0’的ASCII码为30H，
‘1’的ASCII码为31H。



源程序的数据段和堆栈段安排如下：

```
DATA    SEGMENT
BIN1     DB  35H
BIN2     DW  0AB48H
ASCBUF   DB  20H DUP(?)
DATA    ENDS
STACK1  SEGMENT PARA STACK
         DW   20H DUP(0)
STACK1  ENDS
```

由于参量的传递方式有多种形式，其相应地在子程序中取入口参量的方法也有所不同。下面介绍三种参量的传递方法：

*用寄存器传递参量

*用堆栈传递参量

*用地址表传递参量

# 1、用寄存器传递参量

设调用子程序时，入口参量为：被转换的数在DX中，若数位<16，则从高到低地存放,转换后的ASCII码的存放首址在DI中。信息的保存由主程序完成。

主程序

```
COSEG   SEGMENT
        ASSUME CS:COSEG,DS:DATA,SS:STACK1
START:  MOV AX,DATA
        MOV DS,AX
        XOR DX,DX
        LEA  DI,ASCBUF;存放ASCII码的单元首址送DI
        MOV  DH,BIN1   ;待转换的第1个数据送DH
        MOV AX,8       ;待转换的二进制数的位数
        PUSH DI        ;保护信息
        CALL BINASC    ;调用转换子程序
        POP  DI        ;恢复信息
        MOV  DX,BIN2   ;待转换的第二个数据送DX
        MOV  AX, 16
        ADD  DI,8      ;设置下一个数的存放首址
        CALL BINASC
        MOV AH,4CH
        INT 21H
```

# 转换子程序

```
BINASC  PROC
        MOV CX，AX
LOP:    ROL DX，1   ；最高位移入最低位
        MOV AL，DL
        AND AL，1   ；保留最低位，屏蔽其它位
        ADD AL，30H ；AL中即为该数字符（0或1）的ASCII码
        MOV [DI]，AL；存结果
        INC DI          ；修改地址指针
        LOOP LOP
        RET
BINASC  ENDP
COSEG   ENDS
        END START
```

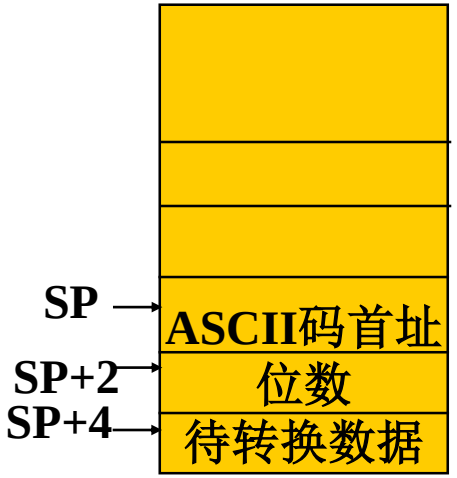
## 2、用堆栈传递参量

如果使用堆栈传递参量，一般应包括：

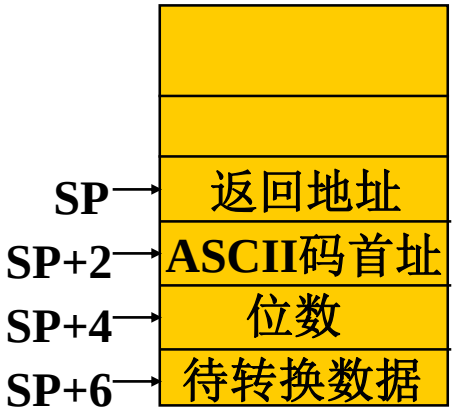
（1）在主程序中，将待转换的数据、存放ASCII码的首址和转换的位数压入堆栈；

（2）在子程序中保存信息。

```
；主程序
COSEG  SEGMENT
ASSUME CS：COSEG，DS：DATA，SS：STACK1
BEGIN：MOV AX，DATA
       MOV DS，AX
       MOV AH，BIN1
       PUSH AX          ；待转换数据压栈
       MOV AX，8
       PUSH AX          ；转换位数压栈
       LEA AX，ASCBUF
       PUSH AX     ；存放ASCII码的首址压栈
       CALL BINASC  ；调用转换子程序
       MOV AX，BIN2
      PUSH AX
      MOV AX，10H
      PUSH AX
     ADD DI，8
      PUSH DI
      CALL BINASC
      MOV  AH，4CH
       INT   21H
```

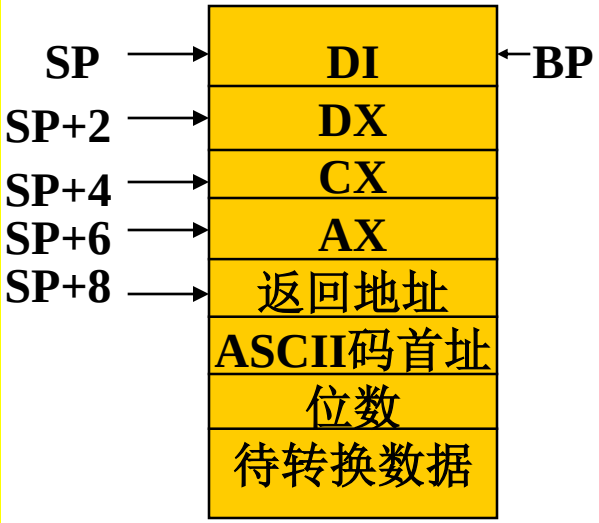



执行CALL指令前堆栈情况



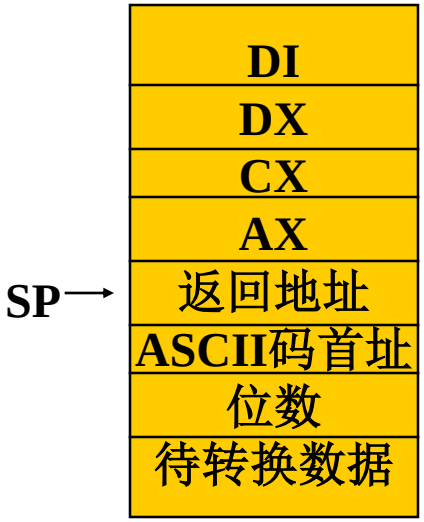

执行CALL指令后堆栈情况

```
；转换子程序
BINASC PROC
      PUSH  AX
      PUSH  CX
      PUSH  DX
      PUSH  DI
      MOV  BP，SP
      MOV DI，[BP+10] ;从堆栈取入口参数
      MOV CX，[BP+12]
     MOV DX，[BP+14];
LOP：  ROL DX，1
      MOV AL，DL
      AND AL，1
      ADD AL，30H
      MOV [DI]，AL
      INC DI
      LOOP LOP
```




子程序中保存信息并执行MOV BP，SP后

```
POP  DI
POP  DX
POP  CX
 POP AX
RET  6   ;返回并从堆栈中弹出6个字节
BINASC ENDP
COSEG  END BEGIN
```




执行RET 6前


执行RET 6后

## 3、用地址表传递参量

传递参数也可以采用传递参量的地址来实现。

在调用子程序前，将所有参量的地址依次存放在一个地址表中， 然后将该表的首地址传送给子程序。

数据段部分改为：

```
DATA    SEGMENT
BIN1    DB  35H
BIN2    DW  0AB48H
CUNT    DB  8，16
ASCBUP  DB 20H DUP(?)
ADR_TAB DW 3 DUP(0)   ;存放参量地址表
DATA    ENDS
```

主程序中有关指令序列修改为：

```
……
MOV  ADR_TAB,OFFSET BIN1   ;存参量地址
MOV  ADR_TAB+2,OFFSET CUNT
MOV  ADR_TAB+4,OFFSET ASCBUP
MOV  BX,OFFSET ADR_TAB ;传表首址
CALL BINASC8
MOV  ADR_TAB,OFFSET BIN2
MOV  ADR_TAB+2,OFFSET CUNT+1
MOV  ADR_TAB+4,OFFSET ASCBUP+8
MOV  BX,OFFSET ADR_TAB ;传表首址
CALL BINASC16
……
```

转换子程序设置两个入口，一个是转换8位数据的入口BINASC8，另一个是转换16位数据的入口BINASC16。

```
BINASC    PROC
BINASC8:  MOV DI，[BX]；取待转换8位数据
          MOV DH，[DI]
          JMP TRAN
BINASC16:MOV DI,[BX]   ;取待转换16位数据
          MOV DX,[DI]
TRAN:     MOV DI,[BX+2] ;取待转换数据位数
          MOV CL,[DI]
          XOR CH,CH
          MOV DI,[BX+4]:取存ASCII码首址
LOP:      ROL DX,1
          MOV AL,DL  ;待转换的1位送到AL中转换
          AND AL,1
          ADD AL,30H    ;构成相应的ASCII码
          MOV [DI],AL   ;存结果
          INC DI
          LOOP LOP
          RET
```

# §6.4 DOS 功能子程序调用

DOS操作系统为程序设计人员提供了可以直接调用的功能子程序。调用这些子程序可以实现从键盘输入数据，将数据送显示器显示，以及磁盘操作等功能。

调用这些子程序时，需要使用软中断指令 INT 21H，并且在执行该指令之前，需要将要调用的功能号送入寄存器AH中，有关的参量送入指定的寄存器。

调用过程包括以下三个步骤：

* 送入口参量给指定寄存器
* AH<=功能号
* INT 21H

## 1、带显示的键盘输入（1号功能）

调用该功能子程序将等待键盘输入，直到按下一个键。将字符的ASCII码送入AL寄存器，并在屏幕上显示该字符。如果是Ctrl-C组合键，则停止程序运行。该功能调用无入口参量。

例如：MOV AH，01H

INT 21H

## 2、不带显示的键盘输入（8号功能）

该功能调用与1号功能的作用相似，区别是8号功能将不显示输入的字符。调用方法为：

MOV AH，8

INT 21H

## 3、不带显示的键盘字符输入（7号功能）

该功能与8号功能相似，但对Ctrl-C组合键和TAB制表键无反应。调用方法：

```
MOV AH，7
INT  21H
```

## 4、字符串输入（0AH号功能）

该功能调用可实现从键盘输入一个字符串，其长度可达255个字符。调用该功能前，应在内存中建立一个输入缓冲区。

缓冲区第一个字节是可输入的最大字符数+1；第二个字节是系统在调用该功能时，自动填入的本次调用时实际输入的字符个数；从第三个字节开始存放输入字符的ASCII码。

当用户输入回车键时，结束输入，并将回车键的ASCII码（0DH）作为最后一个字符送入缓冲区。但它不计入实际输入字符个数。

调用入口参量：

DS和DX寄存器分别装入输入缓冲区的段基值和偏移量

```
CHAR_BUF DB 31H      ;缓冲区的最大长度
         DB 0        ;存实际输入字符数
         DB 31H DUP(0);输入缓冲区
         ......
  MOV DX,SEG CHAR_BUF;如果DS已经指向CHAR_BUF所在
  MOV DS,DX          ;数据段，则可以省去这两条指令
  MOV DX,OFFSET CHAR_BUF
  MOV AH,0AH
  INT 21H
```

## 5、字符显示（2号功能）

该功能实现在屏幕上显示单个字符。

入口参数：DL<=要显示字符的ASCII码。

例如：MOV　DL ‘A’

　　MOV AH，2

　　INT 21H

## 6、字符打印（5号功能）

该功能将字符送入打印机接口，实现单个字符的打印操作。

入口参数：DL<= 打印字符的ASCII码

```
MOV  DL，  ‘A’
MOV  AH，5
INT    21H
```

## 7、字符串显示（9号功能）

该功能实现将一个字符串显示到屏幕上。

入口参数：

（1）将待显示的字符串存放在一个数据缓冲区，字符串以符号“$”作为结束标志。

（2）将字符串的首址的段基值和偏移量分别送入DS和DX中

```
例如：CHAR    DB  ‘This is a test.’,0AH,0DH,’$’
          ..........
          MOV  DX,OFFSET CHAR
          MOV AH,9
          INT  21H
```

## 8、直接输入输出（6号功能）

该功能可以实现键盘输入，也可以实现屏幕显示操作。两种操作通过DL的内容确定。

**（1）（DL）=00—0FEH，显示输出。DL中是所显示字符的ASCII码。**

例如：显示美元符号“$”的程序段为：

```
MOV  DL，24H ；$的ASCII码为24H
MOV AH，06
INT   21H
```

## （2）（DL）=FFH，从键盘输入字符

该功能的字符输入不等待键盘输入，而是从键盘缓冲区中读取。读取的字符ASCII码送入AL中，如果没有键按下，则标志位ZF=1。

例如：

```
WAIT: MOV  DL，0FFH
      MOV  AH，6
      INT 21H
      JZ  WAIT
```

## 9、读出系统日期（2AH号功能）

读出的日期信息放入指定的寄存器中：

CX：年（1980—2099）

DH：月（1—12）

DL：日（1—31）

AL：星期（0—星期日，1—星期一……）

例如：

```
YEAR    DW  ?
MONTH DB  ?
DAY     DB  ?
        ......
        MOV  AH,2AH
        INT  21H
        MOV  YEAR,CX
        MOV  MONTH,DH
        MOV  DAY,DL
```

## 10、设置系统日期（2BH号功能）

该功能用来改变计算机CMOS中的系统日期。入口参数：

CX<=年号（1980—2099）

DH<= 月号（1—12）

DL<= 日（1—31）

返回参数在AL中，成功设置，则返回(AL)=0，否则（AL）=0FFH

例如：

```
MOV CX,2000
MOV DH,11
MOV DL,2
MOV AH,2BH
INT 21H
CMP AL,0
JNE ERROR ;转出错处理
……
```

## 11、读出系统时间（2CH号功能）

执行该功能将获得系统的当前时间。返回的时间参数存放在指定的寄存器中：

**CH：小时（0—23）**
**CL：分（0—59）**
**DH：秒（0—59）**
**DL：百分秒（0—99）**

## 12、设置系统时间（2DH号功能）

调用该功能，将设定系统时间。其入口参数为：

**CH：小时 （0—23） CL：分 （0—59）**
**DH：秒 （0-59） DL：百分秒（0-99）**

该功能执行后返回时，如果调用成功，则（AL）=0。否则（AL）=0FFH

# 第七章 汇编语言简单应用程序设计

本章主要内容：

- 算术运算调整指令及其应用
- 串和表的处理
- 代码转换及其应用

# §7.1 算术运算调整指令及其应用

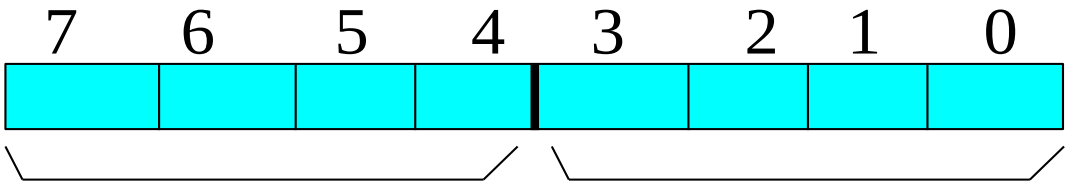
## 一. 十进制数的表示

在计算机中采用BCD码来表示十进制数。BCD码就是使用四位二进制数表示一位十进制数。

在8086/8088系统中，将BCD码分为两种格式：

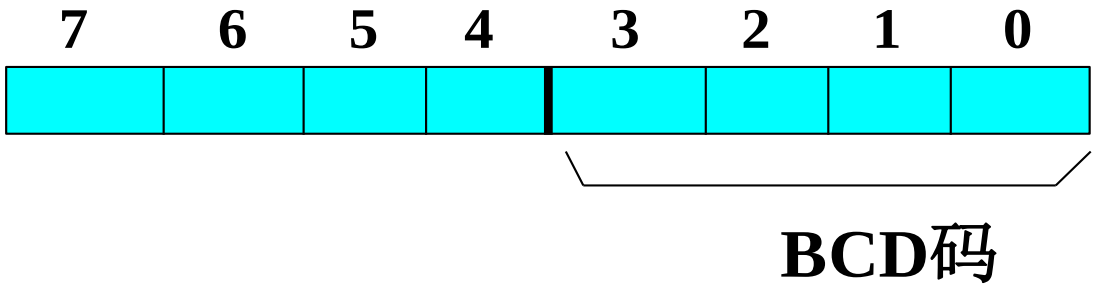
- 组合型（压缩型、装配型、PACKED）
- 非组合型（非压缩型、拆散型、UNPACKED）

组合型：一个字节表示两个BCD码，即两位十进制数。




例如：0010 0011 表示十进制数的23

非组合型：一个字节的低四位表示一个BCD码，而高四位对所表示的十进制数没有影响。常为0000B或0011B。




例如：0000 1001与0011 1001都是十进制数9的非组合型的BCD码

在计算机中实现十进制数(BCD码)的运算有两种方法:

1. 数制转换: 先把十进制数转换为二进制数,然后用计算机中的二进制运算指令进行运算。最后将结果由二进制数转换为十进制数。

2. 使用计算机中的BCD码指令，直接进行十进制数运算。

在计算机内部实现直接对BCD码运算的方法有两种：

（1）指令系统提供专门实现BCD码运算的加、减、乘、除运算指令。

（2）先用二进制数的加、减、乘、除运算指令对BCD码运算，再用BCD码校正指令对结果校正。在8086/8088系统中就是使用这种方法。

## 二、BCD码加减校正指令

8086/8088系统共有六条BCD码校正指令。本节先介绍四条加减校正指令。

### 1、非组合型加法校正指令AAA

AAA指令实现对一位十进制数进行校正。

在AAA指令执行前，应使用ADD或ADC指令完成了BCD码加法，且结果是在AL中。AAA指令对AL中内容进行校正。

校正过程为：

当AL中的低4位>9或者AF=1，则AL<=（AL）+6，AH<=（AH）+1，AL中高4位清0，AF和CF置1。

例如：从键盘输入两个一位数的十进制数，然后相加，结果放在AH和AL中。

```
MOV  AH,1
INT  21H
MOV  BL,AL ;BL中为输入的一位十进制数的ASCII码，低4位为该数的BCD码
MOV  AH,1
INT  21H ; AL中为输入的另一位十进制数的ASCII码
MOV  AH,0
ADD  AL,BL
AAA
```

## 2、组合型加法校正指令DAA

DAA指令实现对二位十进制数进行校正。

在执行DAA指令前，应使用ADD或ADC完成了二位BCD码的加法操作，且加的结果放在AL中。

其校正过程为：

若AL中低4位>9或AF=1，则 AL<=(AL)+6, AF<=1

若AL中高4位>9或CF=1，则AL<=(AL)+60H, CF<=1

例：实现两个4位十进制数的加法4678+2556

```
NUM1  DB  78H,46H
NUM2  DB  56H,25H
SUM    DB  ?,?
          ......
          MOV  AL,NUM1
          ADD  AL,NUM2   ;低字节BCD码相加
          DAA                    ;结果低字节校正
          MOV  SUM,AL
          MOV  AL,NUM1+1
          ADC  AL,NUM2+1;高字节BCD码相加
          DAA                    ;结果高字节校正
          MOV  SUM+1,AL
```

## 3、非组合型减法校正指令AAS

执行AAS指令前，应使用SUB或SBB完成了BCD码的减法运算，且结果放在AL中。

其校正过程为：

若AL中低4位>9或AF=1，则AL<= (AL)-6 , AH<= (AH)-1, 同时将AL中高4位清零，CF和AF置1。

## 4、组合型减法校正指令DAS

执行DAS指令前，应使用SUB或SBB完成了二位BCD码减法运算，且结果放在AL中。

其校正过程为：

* 若AL中低4位>9或AF=1，则AL <=(AL)-6，AF置1；

* 若AL中高4位>9或CF=1，则AL<=(AL)-60H,CF置1。

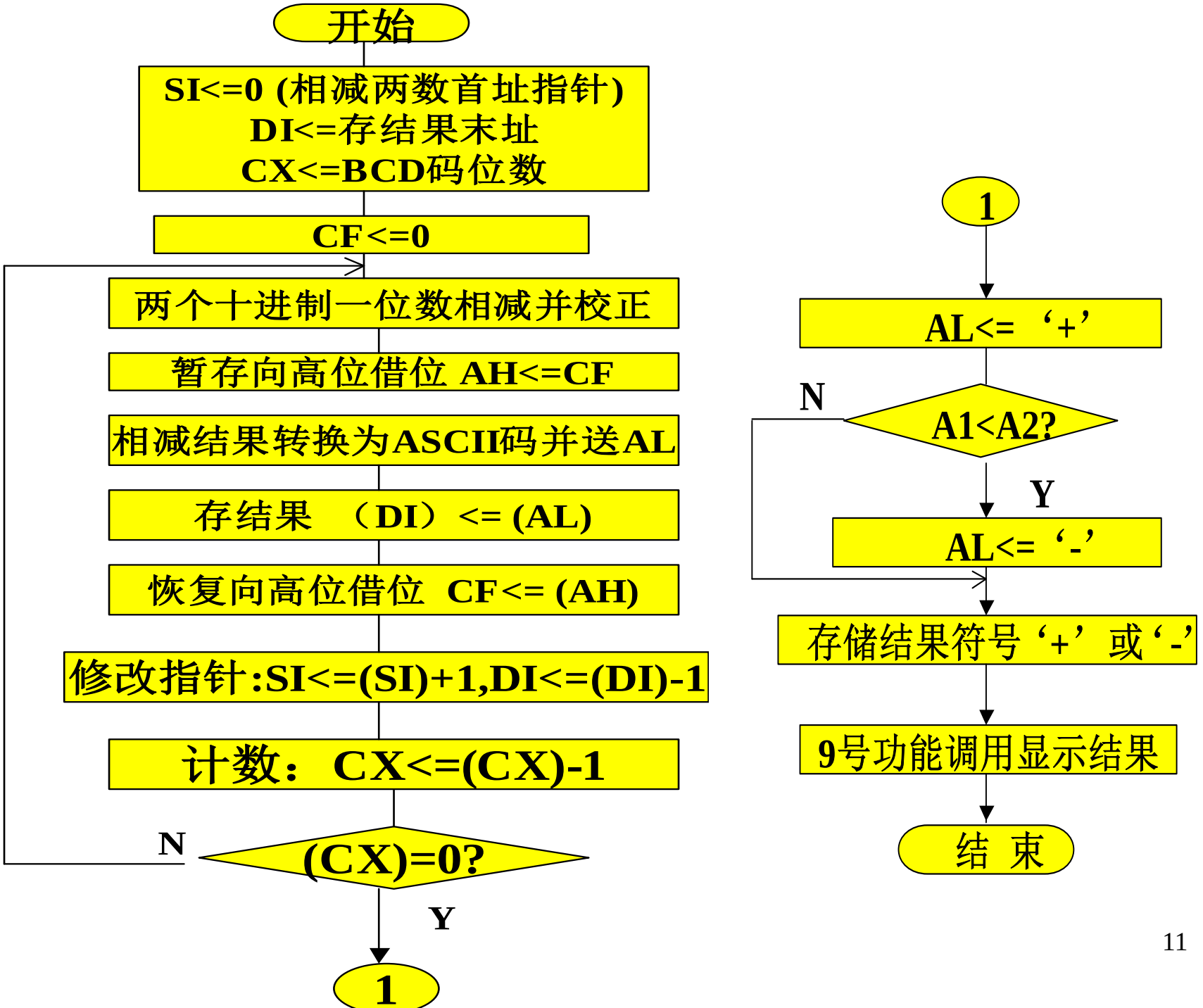
例1 试编制一程序，实现非组合型BCD码减法并显示结果。

设数据段有两个4位十进制数（非组合型BCD码）A1和A2。分别放在以DA1和DA2为首址的存储单元中（低字节放低位，高字节放高位）。

结果存放在以DA3为首址的存储单元中。为了显示方便，结果采用低字节放高位，高字节放低位。

为了表示A1和A2的相对大小，若A1≥ A2，则结果前加‘+’号，否则加‘-’号。

结果的显示使用9号DOS功能调用。


开始
SI<=0 (相减两数首址指针)
DI<=存结果末址
CX<=BCD码位数
CF<=0
两个十进制一位数相减并校正
暂存向高位借位 AH<=CF
相减结果转换为ASCII码并送AL
存结果 （DI）<= (AL)
恢复向高位借位 CF<= (AH)
修改指针:SI<=(SI)+1,DI<=(DI)-1
计数： CX<=(CX)-1
N
(CX)=0?
Y
1
1
AL<= ‘+’
N
A1<A2?
Y
AL<= ‘-’
存储结果符号‘+’或‘-’
9号功能调用显示结果
结 束

```
    TITLE  DECIMAL SUBTRACTION
DATA     SEGMENT
DA1      DB   1,2,3,4
DA2      DB   0,1,2,3
DA3      DB   5 DUP(0),'$'
DATA     ENDS
STACK1   SEGMENT  PARA  STACK
         DW   20H DUP(0)
STACK1   ENDS
CODE     SEGMENT
         ASSUME  CS:CODE,DS:DATA,SS:STACK1
START:   MOV  AX,DATA
         MOV  DS,AX
         MOV  SI,0
         LEA  DI,DA3+4     ;  存结果单元末址送DI
         MOV  CX,4         ;  十进制位数送CX
         CLC
```

```
LOP:  MOV  AL,DA1[SI]
      SBB  AL,DA2[SI]    ;  两数相减
      AAS                ;  校正
      LAHF               ;  暂存向高位的借位
      AND  AL,0FH        ;  转换成ASCII码
      OR   AL,30H
      MOV  [DI],AL       ;  存结果
      SAHF               ;  恢复向高位的借位
      INC  SI
      DEC  DI
      LOOP LOP
      MOV  AL,'+ '
      JNC  NEXT
      MOV  AL,'-'        ;  有向更高位的借位，  存 '-'号
```

```
NEXT: MOV  [DI],AL
      MOV  DX,OFFSET DA3
      MOV  AH,9        ；9号功能调用显示结果
      INT   21H
      MOV  AH,4CH
      INT   21H
CODE ENDS
      END  START
```

# 三. 乘除法指令及调整指令

## 1、无符号数乘法指令MUL

指令格式：**MUL　OPRD**

其中：**OPRD**提供乘法运算的一个操作数，它只能是寄存器或存储器操作数。另一操作数隐含使用**AL**或**AX**寄存器。

运算结果存放在**AX**（字节运算）或**DX：AX**（字乘法）中。

- 字节运算：AX<=(AL) ×(OPRD)
- 字运算 ：DX:AX<= (AX)×(OPRD)

MUL只影响CF和OF标志，其它标志位的值不确定。若结果的AH（字节运算）或DX（字运算）为全0，CF=0、OF=0，否则 CF=1、OF=1。

## 2、带符号数乘法指令IMUL

指令格式：IMUL　OPRD

该指令的功能除了操作数是带符号外，其余与MUL指令相同。

对标志位的影响：若乘积的高半部AH（字节乘法）或DX（字乘法）是低半部的符号扩展(不是有效数值），则CF=0、OF=0 。否则CF=1、OF=1 。

例如，对字节乘法有：

若乘积的(AH)=11111111，且AL最高为1，则表示符号扩展，则CF=0、OF=0；

若乘积的(AH)=00000000，且AL最高为0，则表示符号扩展，则CF=0、OF=0；

若乘积的(AH)=11111110，不是符号扩展，则CF=1、OF=1；

若乘积的(AH)=00000010，不是符号扩展，则CF=1、OF=1。

## 3、无符号数除法指令DIV

**指令格式：DIV OPRD**

其中OPRD是除法运算中的除数，它可以是字节(字节除法）或字（字除法）操作数，只能是寄存器或存储器操作数。

被除数和结果隐含使用以下的寄存器：

- 字节除法：被除数为AX，AL<=商 ，AH<= 余数
- 字除法：被除数为DX和AX，AX<=商, DX<=余数

对标志影响：该指令对标志各位无有效影响。

下面两种情况将产生0型中断，转入除法出错处理：

- （OPRD）=0
- 商>0FFH(字节运算）或商>0FFFFH（字运算）

## 4、带符号数除法指令IDIV

除操作数是带符号数外，该指令的功能与DIV指令相同。

当结果商的值超过-127~+127（字节运算）或-32767~+32767（字运算）范围时，将产生0型中断。

## 5、字节/字扩展指令CBW/CWD

这两条指令主要用于除法指令前，形成双倍长度的被除数。它们都是无操作数指令，隐含使用AX或DX。

指令功能：

CBW：扩展AL中符号位到AH中

CWD：扩展AX中符号位到DX中

两条指令对标志都无影响。

## 6. 非组合型乘法校正指令AAM

执行AAM指令前，必须是用MUL完成了无符号数乘法操作，且结果放在AL中。

调整规则： AH←AL/10(商)，AL←AL/10(余数)

对标志位的影响：根据结果设置PF、SF和ZF，AF、CF和OF等不确定。

## 7. 非组合型除法校正指令AAD

AAD用于除法指令DIV之前，将寄存器AH和AL组成的两位非组合型BCD码，调整成一个在AL中的二进制数。

调整规则： AL←AH×10+AL，AH←0

对标志位的影响：与指令AAM相同。

例2 设数据段有X、Y两变量（无符号数），且$X^Y$不超过一个字的表示范围（65535），试编制一计算$X^Y$的程序。

由于没有乘方指令，因此需要将乘方转换为乘法运算。即：

$$X^Y = \underbrace{X \times X \cdots\cdots \times X}_{Y}$$

```
    TITLE  Mathematical power
DATA   SEGMENT
VARX   DW  5
VARY   DW  6
POWER   DW  ?
DATA   ENDS
STACK1 SEGMENT PARA STACK
       DW  20H DUP(0)
STACK1 ENDS
```

```
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA,SS:STACK1
MATH:  MOV  AX,DATA
           MOV  DS,AX
           MOV  AX,VARX
            MOV  CX,VARY
            DEC  CX
           JE   EXIT
           MOV  DX,0  ;乘积的高位清0
LOP:    MUL VARX
           LOOP LOP
EXIT:  MOV  POWER,AX;存结果
           MOV  AH,4CH
           INT 21H
CODE  ENDS
            END MATH
```

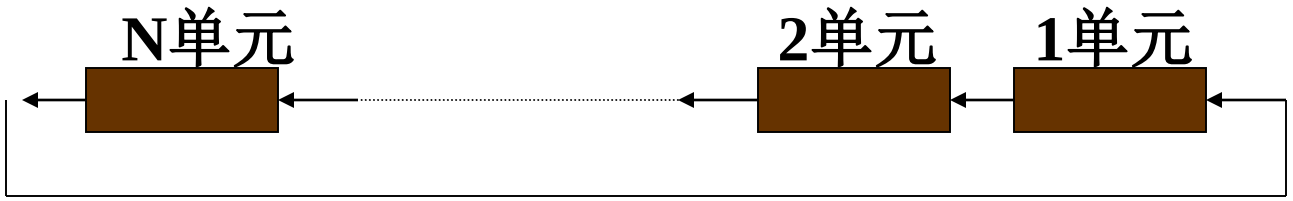
多精度数运算：8086/8088微处理器的每条指令只能处理8位或16位二进制数，其表示的数值范围有限，有时不能满足需要。为了提高运算精度，常用多字节或多字来表示一个完整的数据。

例3 试编写对一个多精度数求补的子程序。

设多精度数的首址放在SI中。数据存储时，低字节放低地址单元，高字节放高地址单元。多精度数的字节数在CL中。程序中，求补采用的是“变反加1”的方法。

```
COMP  PROC
          MOV  CS:RESV,SI    ;暂存多精度数首址
          XOR  CH,CH
          MOV  AL,CL         ;暂存多精度字节数
LOP1: NOT  BYTE PTR [SI]    ;变反
          INC  SI
          LOOP  LOP1
          MOV  SI,CS:RESV   ;恢复多精度数首址
          MOV  CL,AL        ;恢复字节数
          STC                ;  置CF为1
LOP2: ADC  BYTE PTR [SI],0    ;完成最低位加1，
                                ;其它方式？
          INC  SI
          LOOP  LOP2
          RET
RESV  DW  0
COMP  ENDP
```

例4 编制一个多精度数的循环左移子程序。



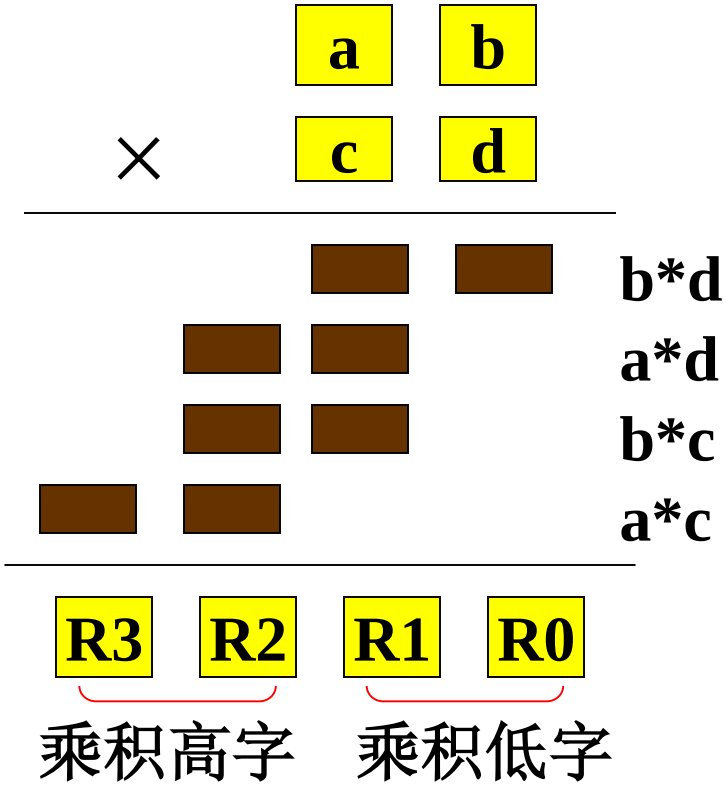

为了将最后一个单元（N单元）的最高位移入第一个单元（1单元）的最低位，先测试N单元的最高位，将测试的结果记录在CF中，在1单元做循环左移时，将其移入最低位。

设多精度数的首址在SI中，字节数在CL中。

```
ROTATE  PROC
        XOR  CH,CH
        MOV  BX,CX
        TEST   BYTE PTR [BX-1][SI],80H
         ;测试最后单元的最高位，并清CF
        JNS  LOP              ;最高位=0,保持CF=0
        STC                   ;最高位=1,CF<=1
LOP:  RCL   BYTE PTR [SI],1  ;循环左移一位
        INC  SI
        LOOP  LOP
        RET
ROTATE  ENDP
```

## 例5 用乘法指令实现32位（双字长）二进制数的乘法

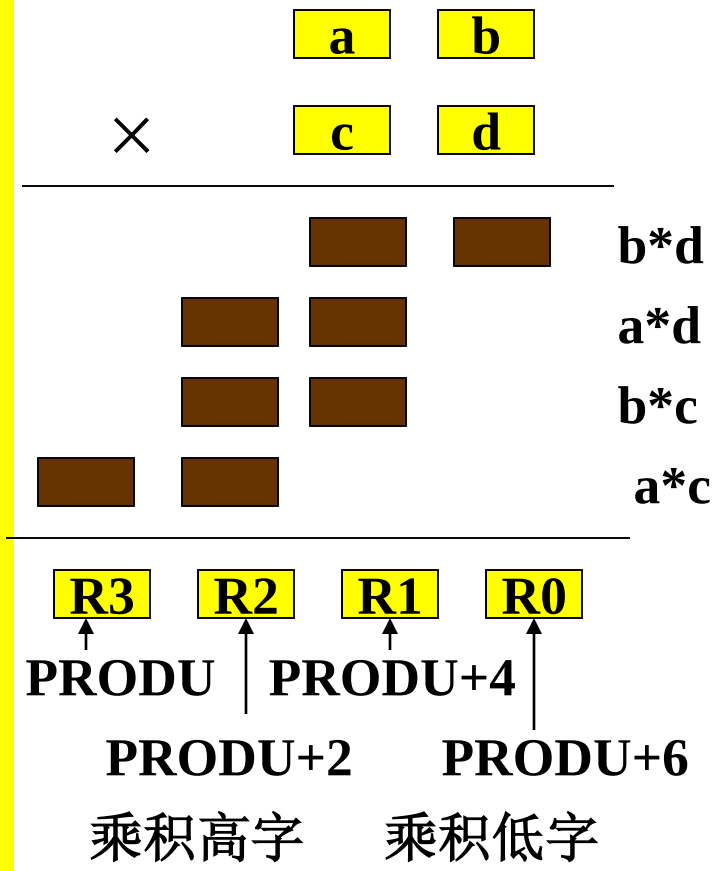
设被乘数两个字用a、b表示，乘数两个字用c、d表示。乘积则为64位（4个字长）。由于乘法指令只能完成单字乘法，对于双字乘法的处理过程如下图所示。




设乘数和乘积存放为:低字存于高地址单元,高字存于低地址单元。

```
 TITLE  32-BIT MULTIPLICATION
DATA   SEGMENT
NUM1   DW  1220H,48A2H
NUM2   DW  2398H,0AE41H
PRODU  DW  4 DUP(0)
DATA   ENDS
STACK1 SEGMENT PARA STACK
       DW  20H DUP(0)
STACK1 ENDS
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA,SS:STACK1
START: MOV  AX,DATA
       MOV  DS,AX
       XOR  DX,DX
       MOV  AX,NUM2+2
       MUL  NUM1+2   ;完成b*d
       MOV  PRODU+6,AX;  存R0
       MOV  PRODU+4,DX;  暂存R1的部分
```

```
    MOV  AX,NUM2+2
    MUL  NUM1            ;完成a*d
    ADD  PRODU+4,AX;
    ADC  PRODU+2,DX
    ADC  PRODU,0 ;将进位送PRODU中
    MOV  AX,NUM2
    MUL  NUM1+2          ;完成b*c
    ADD  PRODU+4,AX;  完成R1存放
    ADC  PRODU+2,DX
    ADC  PRODU,0
    MOV  AX,NUM2
    MUL  NUM1            ;完成a*c
    ADD  PRODU+2,AX ;  完成R2存放
    ADC  PRODU,DX     ;  完成R3存放
    MOV  AH,4CH
    INT 21H
CODE   ENDS
    END START
```



# §7.2 串和表的处理

## 一. 串操作指令

“串”是指存储器中的一段连续的字单元或字节单元。这些单元中存放的内容可以是字符或数据。

“串操作”就是对这些单元进行某种相同的操作。比如将一段数据从一个存储区传送到另一个存储区。

串操作指令具有以下共有特点：

（1）串操作指令使用专用的寻址方式：源操作数地址由DS:[SI]提供，目的操作数由ES:[DI]提供。

（2）串操作指令每次只处理串中的一个字或一个字节单元，并自动修改SI和(或)DI，使其指向下一个字或字节单元。

（3）当标志位DF=0时，SI和DI的修改方向为递增，即加2（字操作）或加1（字节操作）。当DF=1时，SI和DI的修改为递减，即减2或减1。

## 1. 取串指令

指令格式：　LODS　源串

功能：　将源串中的一个字（或字节）内容送入AX（或AL）中，并根据DF修改SI。

由于源串是由SI指定，如果程序中在执行该指令时已经明确是字或字节，则可以用无操作数指令LODSB（字节操作）或LODSW（字操作）替代。

该指令执行后对标志无影响。

## 2. 存串指令

指令格式：　**STOS　目的串**

功能：将**AX**(或**AL**)中的内容送入目的串中的一个字单元(或字节单元)，并根据DF修改DI。

同样，指令可以用无操作数指令**STOSB**或**STOSW**替代。

该指令对标志无影响。

## 3. 串传送指令

指令格式：**MOVS 目的串，源串**

功能：将由SI指向的源串的一个字(或字节)传送到DI所指向的目的串中。并根据DF修改SI和DI。

同样，指令可以用无操作数指令MOVSB或MOVSW替代。

指令对标志无影响。

## 4. 串比较指令

指令格式：　CMPS　源串，目的串

功能：　将源串中的一个字(或字节)减去目的串中的一个字(或字节)，结果不回送。但将影响标志寄存器。同时，将根据DF修改SI和DI。

同样，指令可以用无操作数指令CMPSB或CMPSW替代。

## 5、串搜索指令

指令格式：　SCAS　目的串

功能：　在目的串中查找AX或AL指定的内容。

查找的方法：用AX(或AL)的内容减去目的串中的一个字(或字节)，相减的结果反映在标志寄存器中。每查找一次，将按照DF修改DI。

同样，指令可以用无操作数指令SCASB或SCASW替代。

## 6、重复前缀指令

指令格式：　**REP**

为了方便对若干个字或字节进行多次同样的操作，可在上述各种指令的前面加上**REP**指令。

重复操作的次数由**CX**控制，每执行一次串操作指令，**CX**内容减**1**，直到**CX**内容为**0**。

例如：**REP MOVSB**

设在执行该指令前,DF=0,(SI)=0020H,(DI)=100H,(CX)=0030H。

执行该指令将使数据段中从0020H开始的30H个字节传送到附加段从0100H开始的存储区。

如果改成不使用串操作指令，则它相当于下面的程序段：

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

另外还有两条重复前缀指令：

**REPE/REPZ**

**重复执行串操作指令的条件是：（CX）≠0和ZF=1**

**REPNE/REPNZ**

**重复执行串操作指令的条件是：（CX）≠0和ZF=0**

**由于这两条指令的执行要由标志位ZF来控制结束，而LODS、STOS和MOVS三条指令不影响标志，因此不适合与这些指令配合使用。**

# 二. 串操作指令应用举例

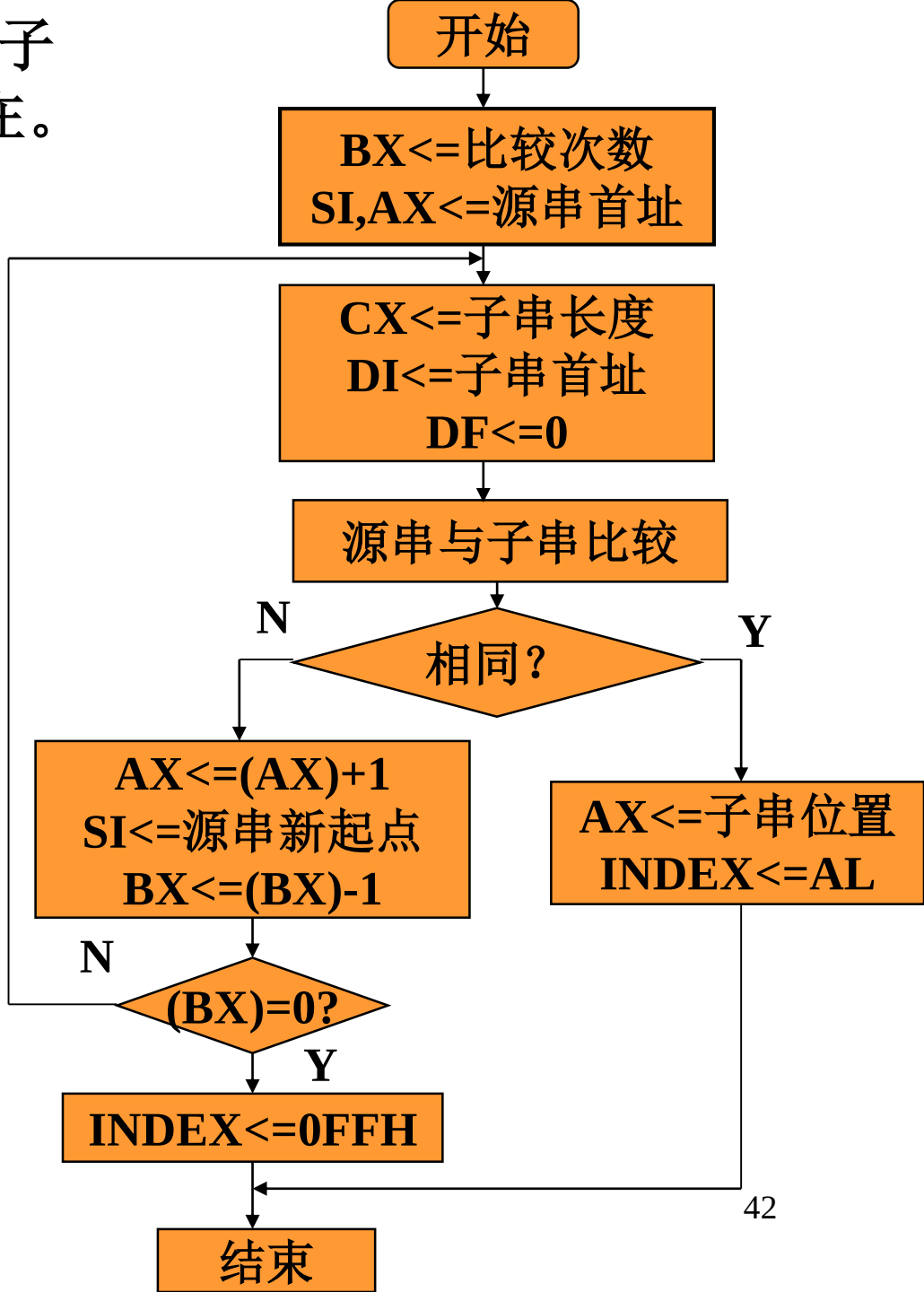
例1 试编制一程序，在TXTBUF字符串中查找STRING变量指定的字符。若查到，则把该字符所在位置（1～n）送入INDEX单元中。若未查到，则把0FFH送INDEX单元中。

```
DATA     SEGMENT
TXTBUF  DB 'ABCDEFGHIJKLMNOP'
CUNT      EQU $-TXTBUF
STRING    DB 'G'
INDEX     DB ?
DATA      ENDS
STACK1   SEGMENT PARA STACK
             DW 20H DUP(0)
STACK1   ENDS
CODE      SEGMENT
     ASSUME CS:CODE,DS:DATA,SS:STACK1
```

```
START: MOV  AX,DATA
       MOV   DS,AX
       MOV   ES,AX
       MOV DI,OFFSET TXTBUF;取目的串首址
       MOV CX,CUNT
       MOV AL,STRING
       CLD
      REPNE SCASB;查找字符
      MOV BL,0FFH
      JNE END0   ;未查到，ZF=0，转移
       SUB DI,OFFSET TXTBUF;DI指向了后一个字符
       MOV BX,DI
END0:MOV INDEX,BL
       MOV AH,4CH
       INT 21H
CODE   ENDS
 END START
```

例2 试编写一程序，确定某子字符串是否在另一字符串中存在。若在，则记录其所在起始位置。若不在则设置标志0FFH。

该例是上例的更一般情况。

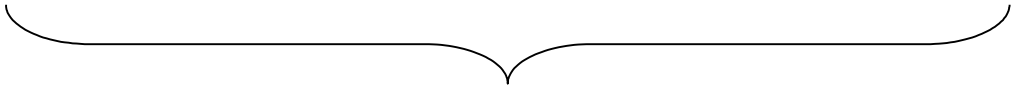


最大比较次数=（源串长度-子串长度）+1

第1次比较：

A B C D E F G H I J K L M N O P

E F



比较次数-1

第2次比较：

A B C D E F G H I J K L M N O P

E F

第3次比较：

A B C D E F G H I J K L M N O P

E F

```
DATA   SEGMENT
TXTBUF DB 'ABCDEFGHIJKLMNOP'
CUNT1  EQU $-TXTBUF
STRING DB 'EF'
CUNT2  EQU $-STRING ；子串长度
INDEX  DB 0FFH
DATA   ENDS
STACK1 SEGMENT PARA STACK
       DW 20H DUP(0)
STACK1 ENDS
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA,SS:STACK1
START: MOV AX,DATA
       MOV DS,AX
       MOV ES,AX
       MOV BX,CUNT1-CUNT2+1;最大比较次数
       MOV SI,OFFSET TXTBUF;取源串首址
       MOV AX,SI ； 信息保护，SI要变化
```

```
LOP:   LEA DI,STRING;取子串首址
       MOV CX,CUNT2
       CLD ;SI,DI递增
       REPZ CMPSB; 比较(SI和DI同时？ )
       JZ  MATCH   ;（CX)=0且ZF=1,相同,转MATCH
       INC AX    ; 未匹配，比较位置往后移一位
       MOV SI,AX
       DEC BX  ;  比较次数计数
       JNZ LOP
       JMP EXIT
MATCH:  SUB AX,OFFSET TXTBUF;位置序号从0开始
       INC  AL                          ;位置序号从1开始
       MOV INDEX,AL
EXIT:   MOV AH,4CH(?,INDEX初始化的值为0FFH)
       INT 21H
CODE   ENDS
       END START
```

# 三. 表的排序与查找

排序是指将一组没有规律的无序数据，排列成有序数据。
查找是从一组数据中搜索指定的数据。

排序和查找一般都是在表中进行的。数据表是指一组连续存放在存储器中的数据。

数据表分有序表和无序表。有序表指各数据项在存储器中按顺序（递增或递减）排列。否则称为无序表。

为了缩短数据查找时间，一般应先将无序表排列成有序表后再查找。

## 1. 气泡排序算法

设有一组无序数据为$b_1$、$b_2$、.....$b_{n-1}$、$b_n$，要求以递减顺序排列。气泡排序法的处理过程如下：

（1）将表中第一个数$b_1$与第二数$b_2$比较，若b1<b2，则交换两数，否则不交换。然后将b2与b3比较，这样重复比较，直至最后两个数。这称为一个循环。

（2）重复执行上述步骤，直至一次循环中没有数据交换为止。

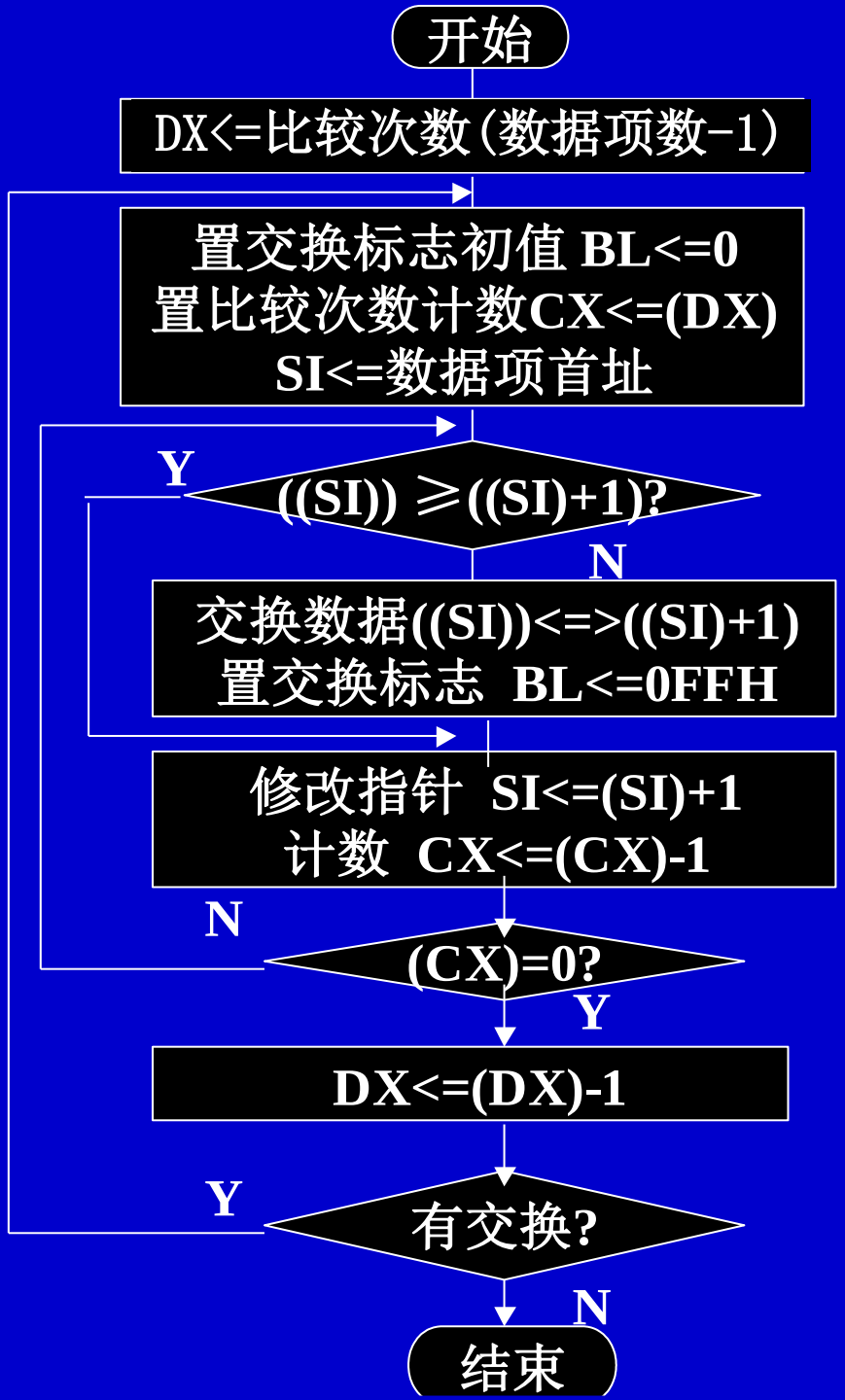
执行完第一次比较循环，将使得最小一个数移动到最后，即一个气泡上浮。

以后每次循环比较将使一个较小的数上浮。并且比较的次数将依次递减。这样直至整个表排列成一个有序表为止。

如果采用递增方式进行排序，其处理方法与上述递减排列方法类似，只需将交换条件变为前面的数大于后面的数。

例如有10个数，经过5次循环比较后，第6次循环比较时已没有数据交换，表明已排列完成。为此，设置一个标识，以确定每次循环比较中是否有数据交换。

| 遍数 / 数据 | | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|---|
| b10 | 42 | -32 | -32 | -32 | -32 | -32 | -32 |
| b9 | -32 | 42 | -20 | -20 | -20 | -20 | -20 |
| b8 | 62 | -20 | 42 | -6 | -6 | -6 | -6 |
| b7 | -6 | 62 | -6 | 42 | 3 | 3 | 3 |
| b6 | 120 | -6 | 62 | 3 | 42 | 9 | 9 |
| b5 | 9 | 120 | 3 | 62 | 9 | 42 | 42 |
| b4 | 116 | 9 | 120 | 9 | 62 | 62 | 62 |
| b3 | -20 | 116 | 9 | 120 | 80 | 80 | 80 |
| b2 | 3 | 3 | 116 | 80 | 120 | 116 | 116 |
| b1 | 80 | 80 | 80 | 116 | 116 | 120 | 120 |


开始
DX<=比较次数(数据项数-1)
置交换标志初值 BL<=0
置比较次数计数CX<=(DX)
SI<=数据项首址
Y
((SI)) ≥((SI)+1)?
N
交换数据((SI))<=>((SI)+1)
置交换标志 BL<=0FFH
修改指针 SI<=(SI)+1
计数 CX<=(CX)-1
N
(CX)=0?
Y
DX<=(DX)-1
Y
有交换?
N
结束

```
 TITLE BUBBLE SORT
DATA     SEGMENT
DA       DB 80,3,-20,116,9,120,-6,62,-32,42
COUNT    EQU $-DA
DATA     ENDS
STACK1 SEGMENT PARA STACK
         DW 20H DUP(0)
STACK1 ENDS
COSEG    SEGMENT
       ASSUME CS:COSEG,DS:DATA,SS:STACK1
SORT:    MOV  AX,DATA
         MOV  DS,AX
         MOV  DX,COUNT-1  ;置比较次数初值
SORT1:   MOV  BL,0        ;置交换标志初值
         MOV  CX,DX      ;置内循环比较次数
         MOV  SI,0       ;置数据指针初值
```

```
SORT2:    MOV  AL,DA[SI]
          CMP  AL,DA[SI+1] ;比较
          JGE  NOXCHG      ;大于等于则转不交换
          XCHG AL,DA[SI+1] ;交换
          MOV  DA[SI],AL
          MOV  BL,0FFH   ;置已交换标志
NOXCHG:INC  SI         ;修改地址
          LOOP SORT2
          DEC  DX        ;修改比较次数
          CMP  BL,0      ;检查交换标志
          JNE  SORT1     ;有交换,继续
          MOV  AH,4CH
          INT  21H
COSEG     ENDS
          END  SORT
```

## 2. 二分法查找算法

二分法查找算法是对有序表进行的查找方法。如果是一个无序表，则必须先将其排列成一个有序表。

算法要点：将整个数据表对分成两个部分，判断所查找的数据属于哪个部分。再将所属部分对分成两个区域，并判断所查找数据属于哪个区域。如此重复操作，直至找到数据或区域长度<1。

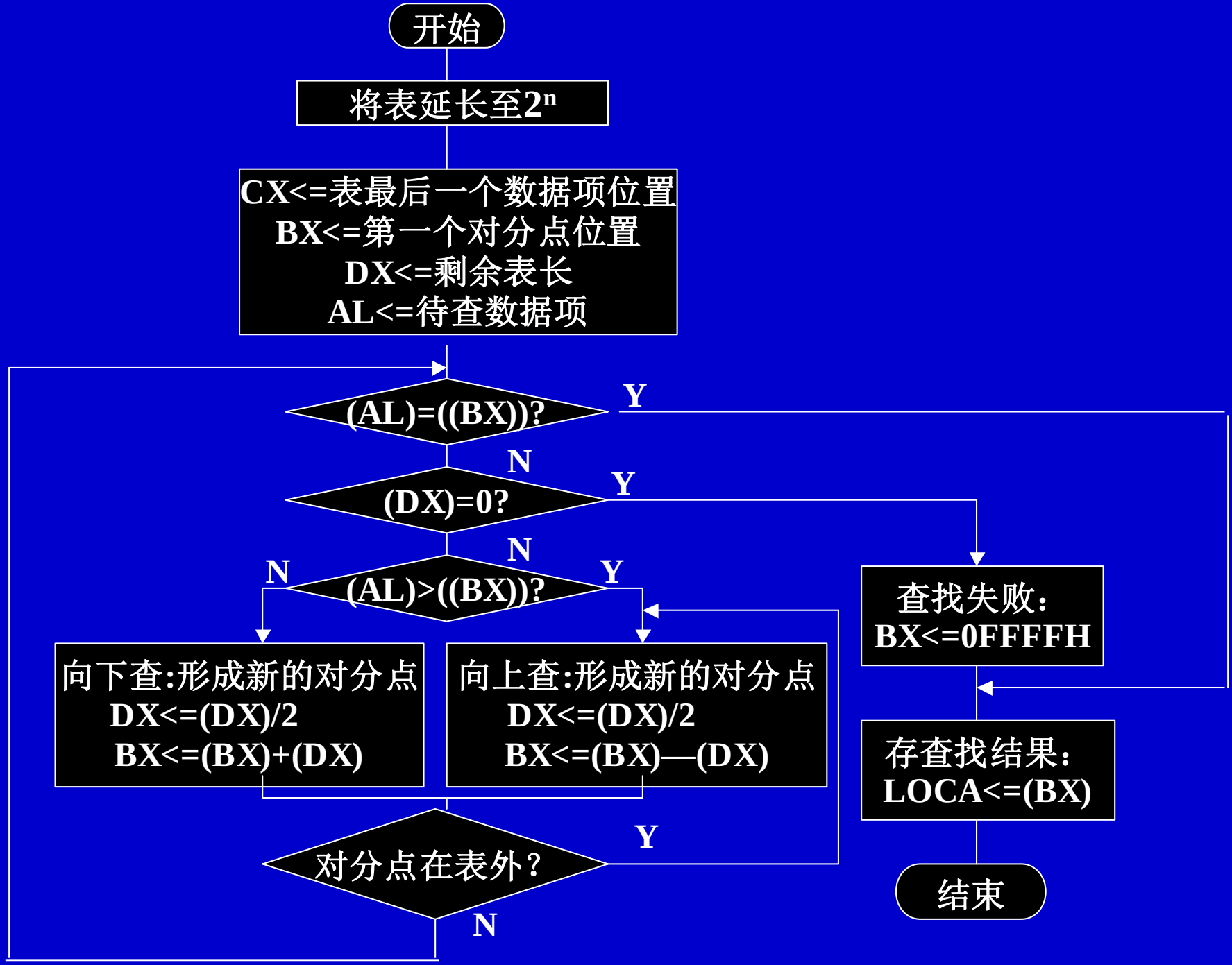
二分法查找算法与顺序查找算法相比，可以减少查找次数，特别是对数据表很大的查找特别明显。对于数据项为N的数据表，二分法查找算法的最多查找次数为：$Log_2N$。

例如当N=1024时，采用顺序查找的平均查找次数为1024/2=512，而二分查找算法的最多查找次数为$Log_2 2^{10}=10$ 。

为了达到对数据表的合理对分，数据表的长度应为$2^n$。为此，对不满足该长度要求的数据表，将其延长至$2^n$。但是，在进行对分时不能使对分的中点落入延长部分。

例　试用二分查找算法在有序表中查找指定的数据。设数据表已经是递减排列。

使用BX作对分中点地址指针，DX存放对分后的查找范围长度。（DX）/2就是下次查找范围长度。若(DX)=0则查找结束。如果找到数据，则将其位置（1~n）送LOCA单元，如果未找到，则送全“1”到LOCA单元。


开始
将表延长至2n
CX<=表最后一个数据项位置
BX<=第一个对分点位置
DX<=剩余表长
AL<=待查数据项
(AL)=((BX))?
Y
N
(DX)=0?
Y
N
N
(AL)>((BX))?
Y
查找失败:
BX<=0FFFFH
向下查:形成新的对分点
DX<=(DX)/2
BX<=(BX)+(DX)
向上查:形成新的对分点
DX<=(DX)/2
BX<=(BX)—(DX)
存查找结果:
LOCA<=(BX)
对分点在表外?
Y
N
结束

```
      TITLE  BINARY  SEARCH
DATA     SEGMENT
TABLE    DB CUNT-1,7FH,7AH,79H,73H,70H,6EH,6BH,6AH
         DB 5DH,5CH,5AH,59H,55H,54H,50H,4DH,4CH,4AH
         DB 49H,48H,44H,41H,40H,3EH,3DH,3AH,39H,36H
         DB 35H,32H,30H,2CH,25H,23H,1FH,19H,14H,00H
CUNT     EQU $-TABLE
DA0      DB  32H  ;被查找数据
LOCA     DW  ?
DATA     ENDS
STACK1 SEGMENT PARA STACK
         DW 20H DUP(0)
STACK1 ENDS
COSEG    SEGMENT
         ASSUME CS:COSEG,DS:DATA,SS:STACK1
BINARY:MOV AX,DATA
         MOV DS,AX
```

```
        MOV BX,1
        MOV AX,CUNT-1 ;CUNT-1为最后一个数的指针?
LENG:   CMP BX,AX ;此循环实现将表延长为2^n
        JAE SEARCH
        SAL BX,1  ;BX此时角色还不是对分点?
        JMP LENG
SEARCH:MOV CX,CUNT-1;CX<=最后一个数据项位置
        SHR BX,1     ;?,第一个对分点
        MOV DX,BX    ;DX<=对分后表长
        MOV AL,DA0   ;AL<=待查找数据
COMP:  CMP AL,TABLE[BX]
       JE END0  ;查到,转结束
       PUSHF    ;未查到，暂存标志
       CMP DX,0 ;表已查完?
       JE   NOFUND;是,转结束
       SHR DX,1 ;?,折半计算
       POPF
```

```
            JG  UP    ;大于对分点数据
            ADD BX,DX ; 向下查，计算出下半部的对分点
             JMP NEXT
UP:       SUB BX,DX ;  向上查，计算出上半部的对分点
NEXT:    CMP BX,CX ;检查对分点是否在表外（比较地址）
            JB COMP  ;在表内,转继续查找
            JMP   UP ;在表外,向上找对分点
NOFUND:MOV BX,0FFFFH;未查到,BX置常数
             POPF ;该操作的结果虽无用,但能保持堆栈操作的正确
END0:    MOV LOCA,BX ;存结果
            MOV AH,4CH
            INT 21H
COSEG  ENDS
           END BINARY
```

# §7.3 代码转换及其应用

代码转换是在计算机程序设计中经常碰到的问题。如二进制数与十进制数的转换，ASCII码表示的各种进制数与二进制数之间的转换等等。

代码转换可以用硬件快速实现，但更常用的方法还是用软件的方法来实现。用软件处理代码转换的方法通常有以下两种方法：

1.查表法：这种方法主要用于代码之间的转换关系比较复杂，但码元的数量必须是有限的情况。

2.直接转换法：依据转换规律，采用一定的算术运算或逻辑运算进行转换。

# 一．十六进制数的ASCII码与二进制数之间的转换

在编制源程序时，常用十六进制数。而从键盘输入时，在计算机中得到的是每个数符的**ASCII码**。因此需要将这些ASCII码表示的数转换为二进制数。

十六进制的每个数符所对应的ASCII码如下表所示。

从表中可以看出以下规律：

1、对于数字符0～9，其ASCII码的低4位就等于对应的二进制值。转换时，只需要将ASCII码的高4位去掉，就是其对应的二进制数。而在二进制数前加上0011B，就是ASCII码。

2、对于数符A～F，各个ASCII码值与对应的二进制数值之差都为37H。

3、对于数符a～f，各个ASCII码值与对应的二进制数值之差都为57H。

| 16 进制数符 | ASCII 码 | 二进制数 |
|---|---|---|
| 0 | 30H | 0000 |
| 1 | 31H | 0001 |
| 2 | 32H | 0010 |
| 3 | 33H | 0011 |
| 4 | 34H | 0100 |
| 5 | 35H | 0101 |
| 6 | 36H | 0110 |
| 7 | 37H | 0111 |
| 8 | 38H | 1000 |
| 9 | 39H | 1001 |
| A | 41H | 1010 |
| B | 42H | 1011 |
| C | 43H | 1100 |
| D | 44H | 1101 |
| E | 45H | 1110 |
| F | 46H | 1111 |
| a | 61H | 1010 |
| b | 62H | 1011 |
| c | 63H | 1100 |
| d | 64H | 1101 |
| e | 65H | 1110 |
| f | 66H | 1111 |

例1 将4位十六进制数的ASCII码分别转换为对应的4位二进制数，然后将它们组合成一个16位长的二进制数。

例如，十六进制数1A2CH，它的ASCII码的表示形式为：31H,41H,32H,43H，而对应的二进制数为0001101000101100B

对于数字0～9的ASCII码，将其高4位二进制数去掉，就是对应的二进制数，对应字母A～F或 a～f ，将其ASCII码减去7，则其低4位与对应的4位二进制数相同。再去掉高4位即可。

```
 TITLE  CONVERT HEX TO BINARY
DATA     SEGMENT
PROMPT DB 'INPUT HEXADECIMAL (4DIGIT):$'
HEX       DB 5,0,5DUP(0)
BIN       DW ?
ERR       DB 0AH,0DH,'ERROR ! NO-HEXADECIMAL ! $'
DATA     ENDS
STACK1 SEGMENT PARA STACK
            DW 20H DUP(0)
STACK1 ENDS
COSEG   SEGMENT
           ASSUME CS:COSEG,DS:DATA,SS:STACK1
HEXBIN:MOV  AX,DATA
            MOV  DS,AX
            LEA    DX,PROMPT;显示提示信息
            MOV  AH,09H
            INT 21H
```

```
        LEA   DX,HEX
        MOV AH,0AH  ;输入4位十六进制数
        INT 21H
        LEA SI,HEX+2 ;取输入字符首地址
        MOV CH,HEX+1        ;取字符数
        MOV AX,0
CONV:  MOV BL,[SI] ;代码转换
        CMP  BL,'0'
        JB      ERROR；<0，出错
        CMP BL,'9'
      JBE  BIN1  ;是0-9，转移
        CALL HEX1 ;是字母符，调用子程序
        JC  ERROR ;是错误的字符
BIN1:  AND BL,0FH
        MOV CL,4
        SAL AX,CL;空出低4位装新转换的值
        OR  AL,BL;  类似于加？
        INC SI
```

```
            DEC CH ；转换字符计数
            JNE CONV
            MOV BIN,AX ;存结果
             JMP END0
ERROR: MOV BIN,0
            LEA DX,ERR
            MOV AH,09H
            INT 21H
END0:    MOV AH,4CH
            INT 21H
;判断大小写子程序
HEX1     PROC
            CMP  BL,'F'
            JA  SMALL
            CMP  BL,'A' ;
            JB  ERROR1；<‘A’,出错
            JMP OUT1；是十六进制数符A-F
SMALL:CMP BL,‘a’ ;检查是否为a~f？
```

```
            JB  ERROR1；<‘a’，出错
            CMP BL,'f'
            JA  ERROR1；>’f’,出错
OUT1:    SUB BL,07H ;
            CLC  ;无错误CF<=0
            RET
ERROR1:STC;设置出错标志CF<=1
            RET
HEX1     ENDP
COSEG  ENDS
            END HEXBIN
```

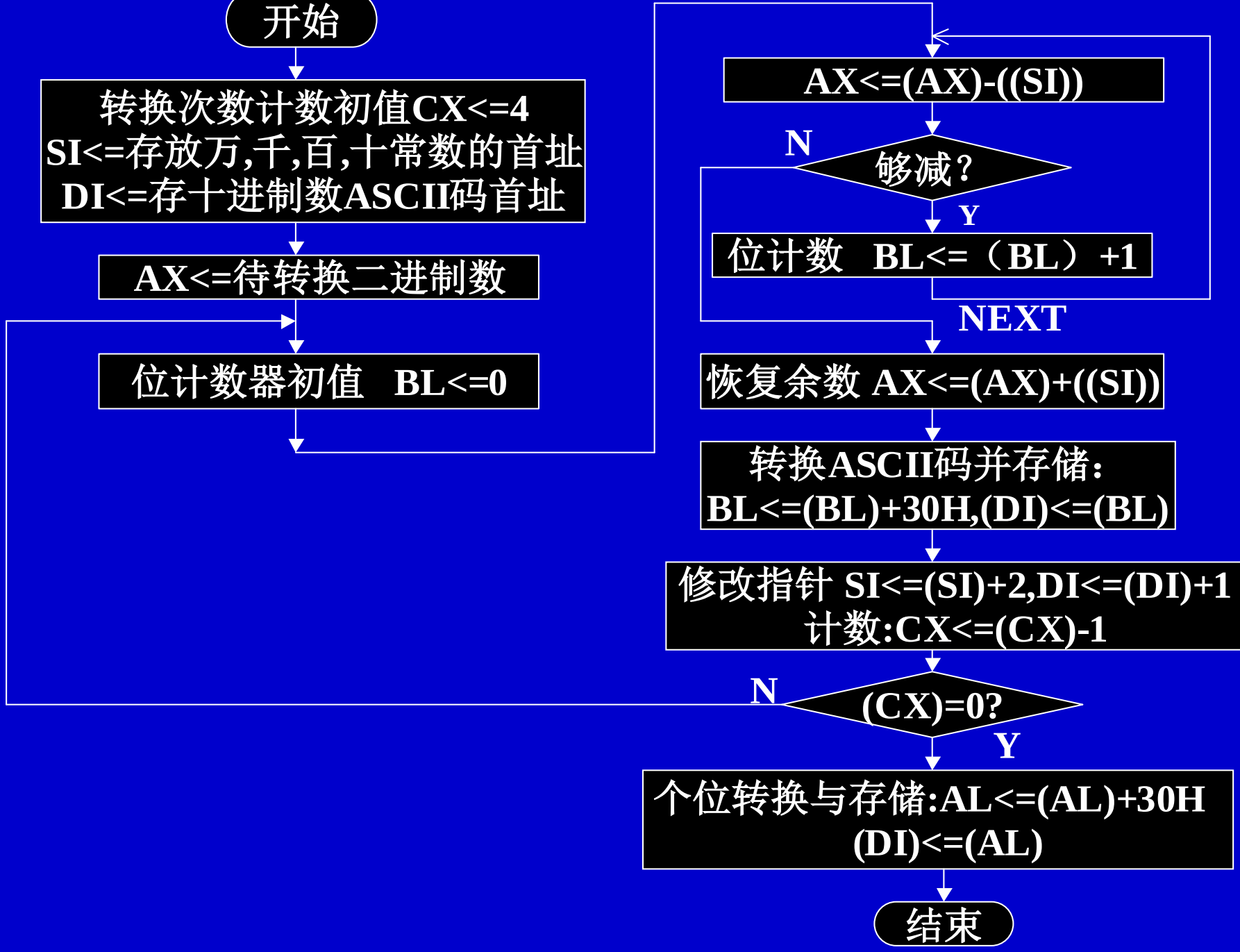
# 二. 二进制数与十进制数之间的转换

例1 将16位无符号二进制数转换为用ASCII码表示的十进制数

算法分析：

1. 16位无符号二进制数表示的十进制数范围为0~65535，需要分别求出万位、千位、百位、十位和个位的值。

2. 从16位二进制数中能够减10000的次数就是万位的值，剩下的数再用1000去减，这样依次进行下去，直至个位。


开始
转换次数计数初值CX<=4
SI<=存放万,千,百,十常数的首址
DI<=存十进制数ASCII码首址
AX<=待转换二进制数
位计数器初值 BL<=0
AX<=(AX)-((SI))
N
够减?
Y
位计数 BL<=（BL）+1
NEXT
恢复余数 AX<=(AX)+((SI))
转换ASCII码并存储:
BL<=(BL)+30H,(DI)<=(BL)
修改指针 SI<=(SI)+2,DI<=(DI)+1
计数:CX<=(CX)-1
N
(CX)=0?
Y
个位转换与存储:AL<=(AL)+30H
(DI)<=(AL)
结束

```
DATA      SEGMENT
BIN1      DW 0110110110101001B
CONST     DW 10000,1000,100,10
DEC5      DB  5 DUP(0)
DATA      ENDS
STACK1  SEGMENT PARA STACK
          DW  20H DUP(0)
STACK1  ENDS
CODE      SEGMENT
          ASSUME CS:CODE,DS:DATA,SS:STACK1
BINDEC: MOV  AX,DATA
          MOV  DS,AX
          MOV  CX,4  ;转换后十进制位数-1
          LEA    SI,CONST ;常数首址
          LEA    DI,DEC5 ;取存十进制数ASCII码的首址
          MOV  AX,BIN1 ;取待转换数
CONV3:  MOV   BL,0  ;位计数器初始化
```

```
LOP1:   SUB   AX,[SI];试减
        JC      NEXT  ;不够减，转NEXT
        INC     BL    ;够减，计数
        JMP     LOP1
NEXT:   ADD   AX,[SI];不够减，恢复余数
        OR      BL,30H ;形成ASCII码
        MOV   [DI],BL;存结果
        INC     SI  ; 取下一个常数地址
        INC     SI
        INC     DI ;修改存结果指针
        LOOP  CONV3  ;继续
       OR    AL,30H ;形成个位的ASCII码
       MOV [DI],AL ;存个位数
       MOV AH,4CH
       INT 21H
CODE   ENDS
       END  BINDEC
```

例2 将16位二进制数转换为非组合型BCD码表示的十进制数。

本例中将16位二进制数转换为5位十进制数的方法是：被转换二进制数除以10，所得余数为十进制数的个位。其商再除以10，所得余数为十位，如此反复，直到商为0。

程序中,用循环实现将二进制数转换为十进制数。十进制数的高位存放在高地址单元，低位存放在低地址单元。

```
DATA    SEGMENT
BIN16    DW  365AH      ;待转换的二进制数
DEC5   DB  5 DUP(0)   ;存转换后的十进制数BCD码
DATA    ENDS
STACK1  SEGMENT PARA  STACK
        DW  20H DUP(0)
STACK1  ENDS
COSEG   SEGMENT
        ASSUME CS:COSEG,DS:DATA,SS:STACK1
MAIN:MOV  AX,DATA
        MOV  DS,AX
        LEA  DI,DEC5 ;取存十进制数个位的单元地址
        MOV  AX,BIN16
        MOV  BX,10
LOP:XOR  DX,DX
        DIV  BX       ;二进制数除以10，余数在DX中
        MOV  [DI],DL  ;存1位十进制数
        INC  DI
```

```
        CMP  AX,0        ;商是否为0?
        JNE  LOP    ;否,则继续转换
        MOV  AH,4CH
        INT  21H
COSEG   ENDS
        END  MAIN
```

# 三. 十六进制数与BCD码的转换

从键盘上输入一个十六进制数字的ASCII码串，将它转换为十进制数的BCD码表示形式。

其转换过程通常分为两个步骤：先把十六进制数ASCII码转换为二进制数，再将二进制数转换为BCD码。这两个步骤我们在前面的例题中已经学习了。下面学习另一种将二进制数转换为BCD码的方法。

设有4位二进制数为$a_3a_2a_1a_0$，每个数符$a_i$的取值为0或1。该二进制数对应的十进制数可用以下公式计算：

$$(((0+a_3)\times 2+a_2)\times 2+a_1)\times 2+a_0$$

公式中需要作4次加法和3次乘法。如果是8位二进制数，则需要作8次加法和7次乘法，其余依此类推。

例1 从键盘输入4位十六进制数（它对应的二进制数是补码表示的带符号数），试编制一程序，把它们转换为带符号非组合型BCD码，并在屏幕上显示出来。

算法分析：

1、从键盘输入4位十六进制数，存放在以HEXBUF+2为首址的4个字节单元中。其中HEXBUF+1单元中为输入的数据个数。

2、将ASCII码表示的4位十六进制数转换为16位二进制数，并暂时存放在BX中；

3、确定十进制数的符号，并把符号(+或-)存放在BCDBUF单元中；

**4、对BX中的二进制数采用前述算法转换为十进制数（非组合型BCD码），转换结果存放在以BCDBUF+1为首址的5个字节单元中；**

**5、把转换结果的5个非组合型BCD码形成相应的ASCII码**

**6、显示结果。**

```
 TITLE  CONVERT HEXADECIMAL TO BCD
DATA    SEGMENT
PROMPT  DB "INPUT HEXADECIMAL(4DIGITS):$'
HEXBUF  DB 5,0,5 DUP(0)
DISP        DB 0AH,0DH
BCDBUF  DB 6 DUP(0),'$'
DATA        ENDS
STACK1  SEGMENT   PARA  STACK
                DW 20H DUP(0)
STACK1  ENDS
```

```
COSEG   SEGMENT
        ASSUME CS:COSEG,DS:DATA,SS:STACK1
HEXBCD: MOV AX,DATA
        MOV DS,AX
;输入4位十六进制数
        LEA DX,PROMPT   ;显示提示信息
        MOV AH,09H
        INT 21H
        LEA DX,HEXBUF   ;输入数据
        MOV AH,0AH
        INT 21H
;ASCII码转换为16位二进制数并存入BX中
        LEA SI,HEXBUF+2   ;取十六进制数ASCII码首址
        MOV BX,0   ;暂存二进制数的寄存器清零
        MOV CH,HEXBUF+1    ;取输入数据个数
HEX1:   MOV AL,[SI]        ;取一个十六进制数字符
        CMP AL,'9'
        JBE NUMB           ;小于等于‘9’是数字符
```

```
        SUB AL,07H          ;是字母符
NUMB:   AND AL,0FH
        MOV CL,4
        SAL BX,CL
        OR  BL,AL
        INC SI
        DEC CH
        JNE HEX1
;确定十进制数的符号
        MOV BCDBUF,'+'
        TEST BX,8000H
        JNS PLUS        ;是正数
        MOV BCDBUF,'-' ;是负数
        NEG BX          ;求补后变为原码
```

```
;将二进制数转换为非组合型BCD码，从二进制
;数高位起,进行15次加和乘
PLUS: MOV CH,0FH     ;“加乘”运算的次数
LOP0: SHL BX,1       ;最高位二进制数送CF
      CALL ADDIT     ;先加1位二进制数
      CALL MULTI     ;再乘2
      DEC CH
      JNE LOP0
      SHL BX,1       ;加最低位二进制数
      CALL ADDIT ;与上一起？
;把非组合型BCD码转换为ASCII码形式
      LEA DI,BCDBUF+1
      MOV CX,5
LOP1: OR  BYTE PTR [DI],30H
      INC DI
      LOOP LOP1
```

显示结果

```
        LEA DX,DISP
        MOV AH,09H
        INT 21H
        MOV AH,4CH ;程序结束
        INT 21H
;多字节BCD码加1位二进制数子程序
ADDIT   PROC
        LEA DI,BCDBUF+5 ;从低位开始
        MOV CL,5        ;取字节数
ADD1:   MOV AL,[DI]     ;取BCD码
        ADC AL,0        ;加二进制数位(CF)
        AAA             ;十进制数运算校正
        MOV [DI],AL     ;存BCD码
        DEC DI
        DEC CL
        JNE ADD1
        RET
ADDIT   ENDP
```

```
;多字节BCD码乘2子程序
MULTI   PROC
        LEA DI,BCDBUF+5   ;从低位开始
        MOV CL,5
        CLC
MUL1:   MOV AL,[DI]       ;取BCD码
        ADC AL,AL         ;乘2
        AAA               ;十进制数运算校正
        MOV [DI],AL       ;存BCD码
        DEC DI
        DEC CL
        JNE MUL1
        RET
MULTI   ENDP
COSEG   ENDS
        END  HEXBCD
```