



河南大學
Henan University

汇编语言与接口技术

——第 4 章 80x86 汇编语言程序设计

主讲教师：舒高峰

电子邮箱：gaofeng.shu@henu.edu.cn

联系电话：13161693313

简答题

- 乘法指令和除法指令的两操作数分别是什么？运算的结果存放在哪个寄存器？
- 对操作数进行位处理（置零、置一、取反）的指令分别是哪些？
- 移位指令对 CF 和 OF 的影响分别是什么？

目录

- 01 80x86 汇编语言程序设计基础
- 02 汇编语言顺序、分支程序设计
- 03 汇编语言循环、串处理程序设计
- 04 子程序设计

01 | 程序设计基础-子目录

80x86 汇编语言程序设计基础

- MASM 宏汇编语句结构及开发过程
 - 指令语句格式
 - 汇编语言程序开发过程
- 汇编语句表达式和运算符
- 程序段的定义和属性
- 常用的系统功能调用

01 | 程序设计基础-指令语句格式

指令书写格式

- [标号:] 助记符 [操作数] [;注释]
 - 标号与助记符用冒号分隔，表示本条指令的**存放位置**
 - 标号不是每条指令中都需要，必要时定义
 - 16 位系统中，汇编指令的操作数一般为 0-2 个

伪指令书写格式

- [符号名] 伪指令助记符 操作数 [;注释]
 - 符号名与伪指令之间不需要冒号分隔
 - 伪指令的操作数至少 1 个，多个时用逗号分隔

01 | 程序设计基础-硬指令语句格式

指令语句格式

[标号:] 指令助记符 [操作数] [;注释]

- **标号**：机器指令语句存放地址的符号表示，代表该指令目标代码的第一个字节地址，后面必须紧跟“冒号”
- **指令助记符**：语句的核心成分，表示了该语句的操作类型
- **操作数**：表示指令助记符的操作对象，各个操作数之间必须以“逗号”分隔
- **注释**：以“分号”开始，它可占一行或多行，一般放在一条语句的后面

这里说的冒号、逗号、分号均为半角符号

01 程序设计基础-伪指令语句格式

伪指令语句格式

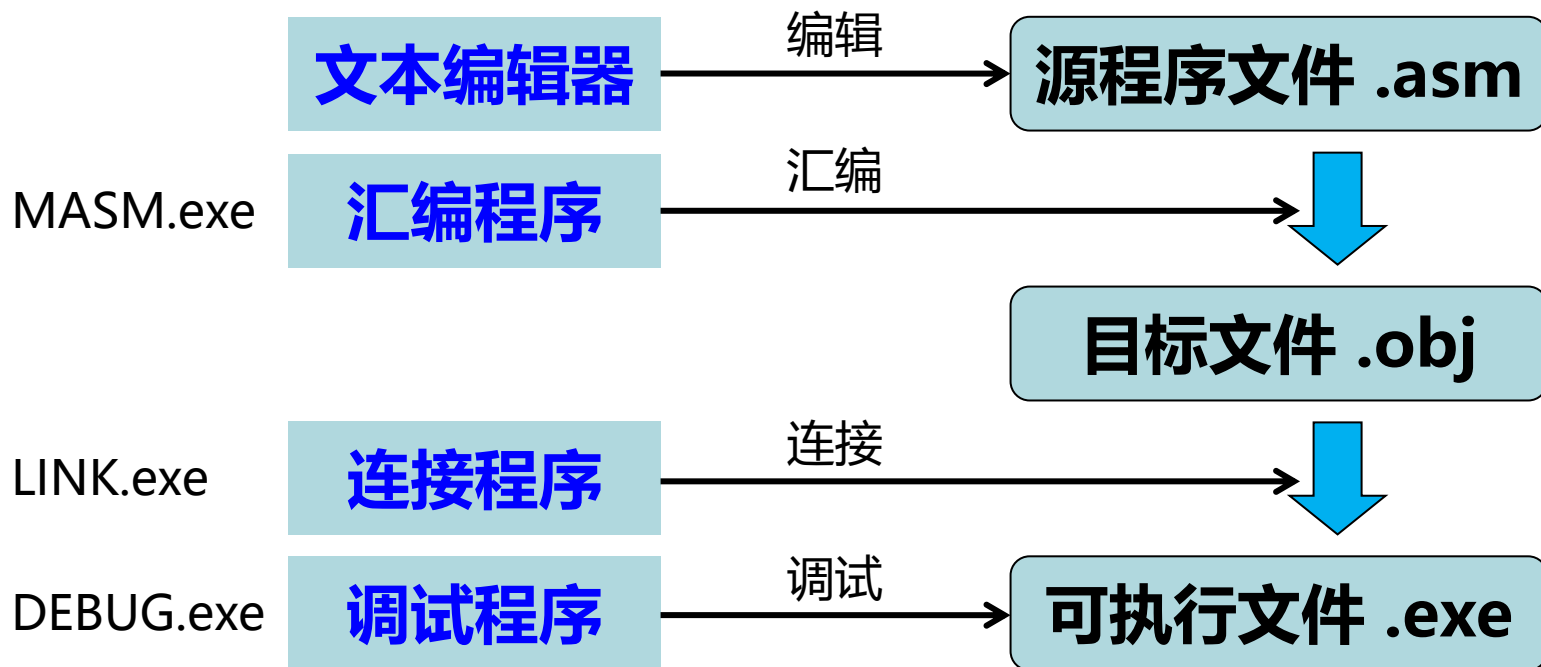
[符号名] 伪指令助记符 操作数 [;注释]

- **符号名**：伪指令语句的一个可选项，既反映逻辑地址又具有自身的各种属性
- **伪指令符**：指定汇编程序要完成的具体操作，如数据定义伪指令 DB、DW、DD，段定义伪指令 SEGMENT，假定伪指令 ASSUME 等
- **操作数**：至少有 1 个操作数，可以是常数、字符串、变量、表达式等，各个操作数之间必须以“逗号”分隔
- **注释**：伪指令的注释必须以“分号”开始，其作用同指令语句中的注释部分

	汇编指令与伪指令的区别
指令	产生目标代码，由 CPU 执行，完成各种数据传送、算术运算等功能
伪指令	不产生目标代码。由汇编程序执行，告诉汇编程序该汇编语言程序段的数目、名称，定义变量和常量、子程序，分配存储空间等

01 | 程序设计基础-源程序的开发过程

汇编语言程序的主要开发过程



01 | 程序设计基础-子目录

80x86 汇编语言程序设计基础

- MASM 宏汇编语句结构及开发过程
- 汇编语句表达式和运算符
 - 常量、变量和表达式
 - 标号
 - 表达式中的运算符
- 程序段的定义和属性
- 常用的系统功能调用

01 | 程序设计基础-数字常量

各种形式数字常量格式对照表

常量形式	格式	X 的取值	常例	说明
二进制常量	XX...XB	0、1	01110011B	数据类型后缀为 B
八进制常量	XX...XO	0~7	12537O	数据类型后缀为 O
十进制常量	XX...X XX...XD	0~9	1234 1234D	后缀可省略 数据类型后缀为 D
十六进制常量	XX...XH	0~9 A~F	0AB12H	如果第一位数是 A~F, 则必须在数的前面加 0

01 | 程序设计基础-字符串常量、符号常量

字符串常量

- 字符串常量是用**单引号**或**双引号**括起来的一个或多个字符
 - 字符串常量是**以各字符的 ASCII 码表示**的。如 'A' 用 41H 表示, 字符串 'A1B2' 用 41H, 31H, 42H, 32H 表示

符号常量

- 利用一个标识符表达的一个数值
 - 常量若使用有意义的符号名来表示, 可以提高程序的可读性, 同时更具有通用性
 - MASM 提供等价机制, 用来为常量定义符号名, 符号定义伪指令有 **"等价伪指令 (EQU)"** 和 **"等号 (=)"**

01 | 程序设计基础-常量定义伪指令EQU

等价伪指令 EQU (**E**quate) 格式

- 标号名 **EQU** 数值表达式
- 标号名 **EQU** <字符串>

- 功能：给符号名定义一个**数值**，或定义成一个**字符串**，这个字符串甚至可以是一条**处理器指令**
- 要求：字符串必须用尖括号括起来，同一符号名只能定义一次数值表达式

```
DATA EQU 2  
CALLDOS EQU <INT 21H>  
CARRIAGERETURN EQU 13
```

注意

- 该语句仅为**标识符来赋值**，并**不会分配存储空间**！
- 汇编程序会将源程序中的标识符替换成对应的数值
- 在同一程序中，**EQU** 语句对一个符号名只能定义一次

01 | 程序设计基础-等号=伪指令

等号 = 伪指令格式

```
X=7  
X=X+5
```

- 标号名 = 数值表达式
 - 功能：定义标识符，来代替表达式的值

注意

- = 伪指令**只能**定义数值
- 在同一程序中可对一个符号名重复定义

01 | 程序设计基础-变量

变量

- 存储器中某个**数据区的名字**，指示所定义变量的**开始地址**，在指令中可以作为存储器操作数
- 变量属性：
 - **段属性**
 - **偏移地址**属性
 - **类型**属性 (字节、字、双字)

Define **Byte**
Define **Word**
Define **Doubleword**

Define **Quadword**
Define **Ten byte**

助记符

- 助记符用于确定操作数的类型
 - 每个操作数在内存中存放所占的**字节数**
- 常用的助记符：**DB** (字节)、**DW** (字)、**DD** (双字)
- 其他助记符：**DF** (三字)、**DQ** (四字)、**DT** (十字节)

01 | 程序设计基础-变量的定义1

数值表达式

- 标号名 DB/DW/DD 数值或数值表
 - 功能：数据定义伪指令可以为一个或连续的存储单元设置数值初值
 - 要求：根据不同的伪指令符给变量分配字节 (DB)、字 (DW)、双字 (DD) 或字节串、字串、双字串
 - 在内存中存放时，低位字在前，高位字在后

示例

```
1. DATA SEGMENT
2.   X DB 11H,12H,13H
3.   Y DW 1122H,3344H
4.   Z DD 12345678H
5. DATA ENDS
```

X	11	DATA
	12	
	13	
Y	22	
	11	
	44	
	33	
Z	78	
	56	
	34	
	12	

01 | 程序设计基础-变量的定义2

字符串表达式

- 标号名 DB/DW/DD 字符或字符串

- 功能：将字符串中的各字符均以 ASCII 码形式存放在相应的存储单元，但表示形式各不相同
- 要求：根据不同的伪指令符给变量分配字节 (DB)、字 (DW)、双字 (DD) 或字节串、字串、双字串

示例

```
1. DATA SEGMENT
2.     STR1 DB '123'
3.     STR2 DW 'AB', 'A'
4.     STR3 DD 'AB'
5. DATA ENDS
```

STR1	31	DATA
	32	
	33	
STR2	42	
	41	
	41	
	00	
STR3	42	
	41	
	00	
	00	

01 程序设计基础-变量的定义3

? 表达式

- 标号名 DB/DW/DD ?

- 功能：? 表达式只分配存储单元，不赋值
- 要求：根据不同的伪指令符给变量分配字节 (DB)、字 (DW)、双字 (DD) 或字节串、字串、双字串

示例

```
1. DATA SEGMENT
2.     BUF1 DB 5, 6, ?
3.     BUF2 DW 100H, ?
4.     BUF3 DD ?
5. DATA ENDS
```

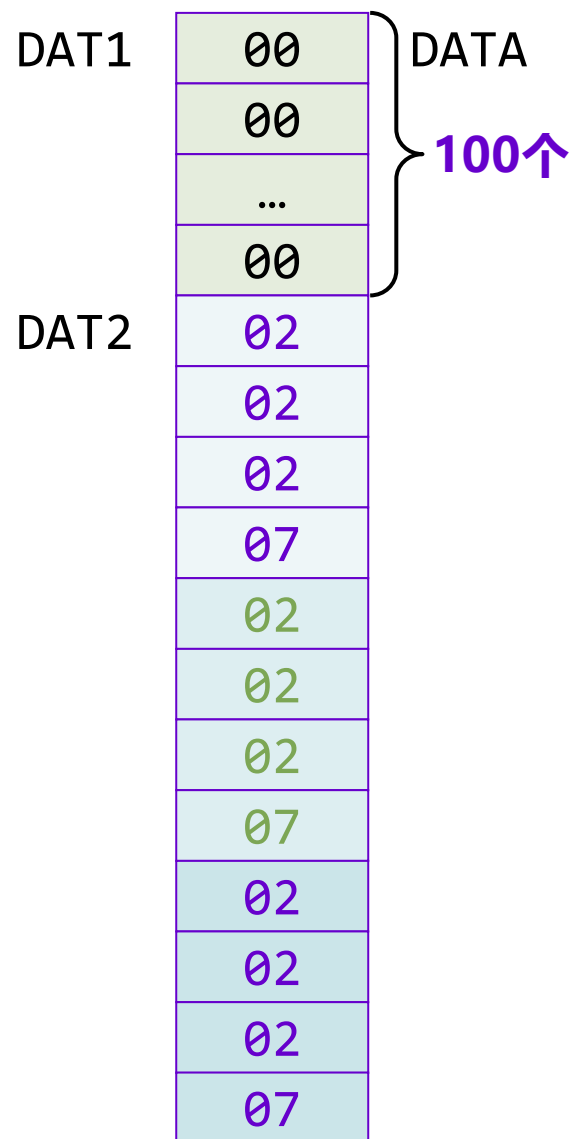
BUF1	05	DATA
	06	
	--	
BUF2	00	
	01	
	--	
	--	
BUF3	--	
	--	
	--	
	--	

01 程序设计基础-变量的定义4

带 DUP (Duplicate) 的表达式

- 标号名 DB/DW/DD N DUP (表达式)

- 功能：为连续的存储单元提供重复数据
- 要求：其中 N 为重复因子，只能取正整数，表示定义了 N 个重复数据存储单元，其类型由它前面的数据定义伪指令确定，而每个数据存储单元中的初值由 DUP 后面圆括号中的表达式给定



示例

```
1. DATA SEGMENT
2.   DAT1 DB 100 DUP(0)
3.   DAT2 DB 3 DUP(3 DUP(2), 7)
4. DATA ENDS
```

01 | 程序设计基础-变量定义伪指令练习

请写出下列变量定义伪指令的内存分配情况

```
1. DATA1 DB 01H, 'A', '123', 'BCD'
2.         DB 1, 2 DUP(2, 3)
3. DATA2 DW 0AB1CH, ?, 0
4.         DW 20, 14H, -1
5. DATA3 DD 234H, ?
```

DATA1

...	01	41	31	32	33	42	43	44	01	02	03	02	03	...
-----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

DATA2

...	1C	AB	--	--	00	00	14	00	14	00	FF	FF	...
-----	----	----	----	----	----	----	----	----	----	----	----	----	-----

DATA3

...	34	02	00	00	--	--	--	--	...
-----	----	----	----	----	----	----	----	----	-----

要在程序中定义缓冲区，保留 9 个字节存储空间的指令是（ ）

- ☐ A BUF DB 9
- ☐ B BUF DW 9
- ☒ C BUF DB 9 DUP (?)
- ☐ D BUF DW 9 DUP (?)

01 程序设计基础-定位伪指令ORG

定位伪指令 ORG (Origin) 格式

● ORG 数值表达式

- 功能：将其后的数据或指令从“数值表达式”所指定的位置开始存放

示例

```
1. ORG 100H
2. X1 DW 23H
3.    DB 23H
4.
5. ORG 200H
6. X2 DB 'ABC', 0DH, 0AH
7. X3 DB ?
```

	数据段	
X1	23	0100H
	00	0101H
	23	0102H
	...	
X2	41	0200H
	42	0201H
	43	0202H
	0D	0203H
	0A	0204H
X3	--	0205H
	

01 程序设计基础-定位伪指令EVEN

定位伪指令 EVEN 格式

● EVEN

- 功能：使当前偏移地址指针指向偶数地址
- 若原地址指针已指向偶地址，则不作调整；否则将地址指针加 1，使地址指针偶数化

示例

```
1. ORG 100H
2. X1 DW 23H
3.    DB 23H
4.
5. EVEN
6. X2 DB 'ABC', 0DH, 0AH
7. X3 DB ?
```

	数据段	
X1	23	0100H
	00	0101H
	23	0102H
		0103H
X2	41	0104H
	42	0105H
	43	0106H
	0D	0107H
X3	0A	0108H
	--	0109H
	

01 程序设计基础-定位伪指令ALIGN

定位伪指令 ALIGN 格式

● ALIGN n

功能

- 将当前偏移地址指针指向 n（n 是 2 的乘方）的整数倍的地址
 - 若原地址指针已指向 n 的整数倍地址，则不作调整；否则将指针加上 1 ~ n-1 中的一个数，使地址指针指向下一个 n 的整数倍地址

注意

- ORG、EVEN 和 ALIGN 指令也可在代码段使用，用于指定随后指令的偏移地址

01 程序设计基础-定位伪指令举例

示例

```
1. DATA SEGMENT
2.   X1 DB 1,2,3
3.   EVEN
4.   X2 DW 5
5.   ALIGN 4
6.   X3 DD 6
7.   ORG $+10H
8.   X4 DB 'ABC'
9. DATA ENDS
```

\$ 为数据、程序分配的当前地址

	数据段			
x1	01	0000H	x4	41	001CH
	02	0001H		42	001DH
	03	0002H		43	001EH
		0003H			001FH
x2	05	0004H		
	00	0005H			
		0006H			
		0007H			
x3	06	0008H			
	00	0009H			
	00	000AH			
	00	000BH			
		000CH			
				

01 | 程序设计基础-标号1

标号

- 一条**指令**语句的符号地址
 - 在汇编源程序中，只有在需要转向一条指令语句时，才为该指令语句设置标号，以便在转移类指令（含子程序调用指令）中直接引用这个标号
 - 标号可作为转移类指令的操作数，即转移地址

标号属性

- **段属性**：标号对应存储单元所在段的**段地址**
- **偏移地址**属性：标号对应存储单元所在段的段内**偏移地址**
- **类型**属性：标号、子程序名的类型可以是 **NEAR (近)** 和 **FAR (远)**，分别为段内和段间

01 | 程序设计基础-标号2

标号

- 一条指令语句的符号地址
 - 在汇编源程序中，只有在需要转向一条指令语句时，才为该指令语句设置标号，以便在转移类指令（含子程序调用指令）中直接引用这个标号
 - 标号可作为转移类指令的操作数，即转移地址

示例

```
1. NEXT: MOV AL, [SI] ;带标号 NEXT 的指令
2.      ...
3.      DEC CX
4.      JNE NEXT ;跳转到标号 NEXT 处执行指令
```

01 | 程序设计基础-运算符

数值运算符 (面向操作数)

- 算术运算符、逻辑运算符、关系运算符

属性运算符 (面向变量或标号)

- 段属性 **SEG**、偏移属性 **OFFSET**、类型属性 **TYPE**、长度属性 **LENGTHOF**、容量属性 **SIZEOF**

属性修改运算符

- 强制类型 **PTR**

01 | 程序设计基础-数值运算符-算术运算符

运算符类型

- +(正号)、-(负号)
- +(加)、-(减)、*(乘)、/(除)、MOD(取模)

注意

- 算术运算符与**立即数**、**常量**、**括号**等构成数值表达式。
变量不能参与算术运算构成数值表达式

示例

$$\begin{aligned} 120 + (321 - 90) \bmod 3 &= 120 + 231 \bmod 3 \\ &= 120 + 0 = 120 = 78\text{H} \end{aligned}$$

$$322 * 5 / 32 = 1610 / 32 = 50 = 32\text{H}$$

01 | 程序设计基础-数值运算符-逻辑运算符

运算符类型

- AND(与)、OR(或)、NOT(非)、XOR(异或)
- SHL(左移)、SHR(右移)

注意

- 逻辑运算符与**立即数**、**常量**、**括号**等构成数值表达式。
变量不能参与算术运算构成数值表达式

示例

1 SHL 3 = 08H

47H AND 0FH = 07H

NOT 56H = 0A9H

01 | 程序设计基础-数值运算符-关系运算符

运算符类型

- EQ(等于)、NE(不等)、LT(小于)、GT(大于)、LE(小于等于)、GE(大于等于)

注意

- 关系运算符与立即数、常量、变量、括号等构成数值表达式
- 关系运算只能有 0(假) 和 0FFH(真) 两个返回值
 - 返回值以二进制补码形式表示，位数由目的操作数决定

示例

MOV AX, 120 LT 100H ; (AX)=0FFFFH

MOV AL, 21 EQ 21H ; (AL)=0

01 | 程序设计基础-属性运算符

属性运算符

- 属性运算符（属性操作符）是面向**变量或标号**的
- 属性操作符的类型
 - **返回值型**：获取变量或标号的相关**属性**返回值
 - **强制转换型**：强制**改变**变量或标号的相关**属性**

变量和标号

- 在数据定义和程序中设置的标识符
 - **变量**：表示数据的地址
- **标号**：表示指令的地址

B1 DB 12H

LOOP1: MOV AX, BX

01 程序设计基础-段属性操作符SEG

段属性操作符 SEG (Segment) 格式

- SEG 标识符

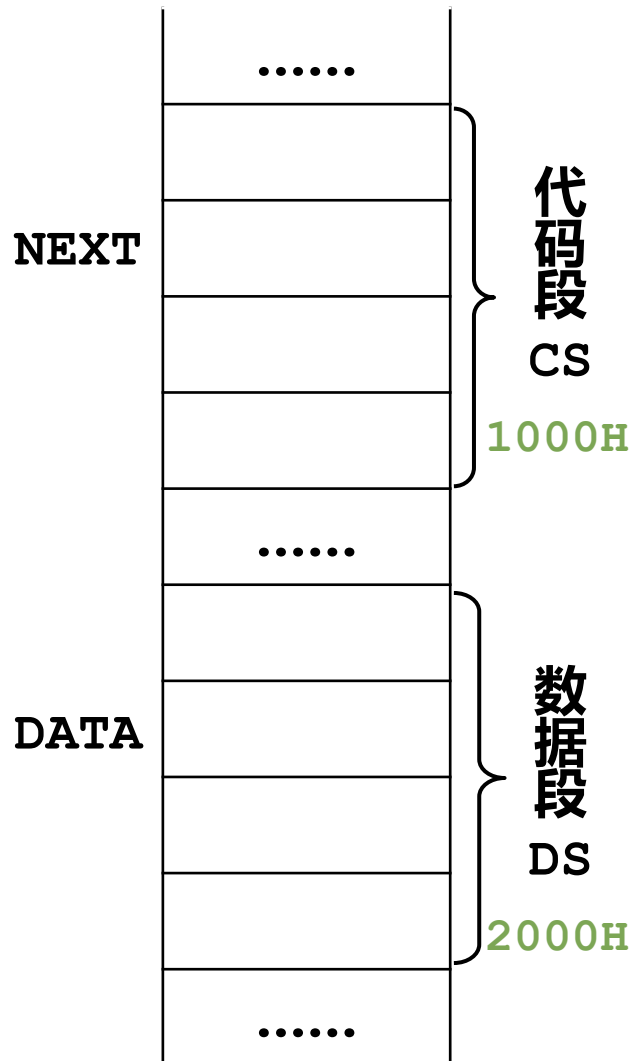
- 功能：返回标识符所在段的段地址

示例

```
1. MOV AX, SEG NEXT
2. MOV BX, SEG DATA
```

- 若存储器如右所示，则以上指令执行后

✎ (AX) = 1000H, (BX) = 2000H



01 程序设计基础-偏移量属性操作符OFFSET

偏移量属性操作符 OFFSET 格式

- **OFFSET 标识符**

- 功能：返回标识符的段内偏移地址

示例

```
1. X1 DW 12H, 100H
2. X2 DD 0
3. X3 DB ?, 'A'
```

- 若执行指令：`MOV AX, OFFSET X1`

`MOV BX, OFFSET X3`

 (AX) = 0000H, (BX) = 0008H

	数据段	
X1	12	0000H
	00	0001H
	00	0002H
	01	0003H
X2	00	0004H
	00	0005H
	00	0006H
	00	0007H
X3	--	0008H
	41	0009H
	

01 | 程序设计基础-类型属性操作符TYPE

类型属性操作符 TYPE 格式

- TYPE 标识符

- 功能：返回标识符的类型值

类型值

- 变量：每个变量所占字节数
- 标号：标号的 NEAR、FAR 类型

标识符类型	TYPE 值
字节 (DB)	1
字 (DW)	2
双字 (DD)	4
近标号 (NEAR)	-1
远标号 (FAR)	-2

01 | 程序设计基础-类型属性操作符举例

类型属性操作符示例

- 数据段定义如下:

```
1. X1 DW 12H, 100H
2. X2 DD 0
3. X3 DB ?, 'A'
```

- 若执行指令: `MOV AX, TYPE X1`

`MOV BL, TYPE X2`

 `(AX) = 0002H, (BL) = 04H`

	数据段	
X1	12	0000H
	00	0001H
	00	0002H
	01	0003H
X2	00	0004H
	00	0005H
	00	0006H
	00	0007H
X3	--	0008H
	41	0009H
	

01 程序设计基础-长度属性操作符LENGTHOF

长度属性操作符 LENGTHOF 格式

- **LENGTHOF** 变量名
 - 功能：返回变量所定义的数据个数

示例

```
1. ARRAY1 DW 2 DUP(0,1), 1
```

- 若执行指令：**MOV AL, LENGTHOF ARRAY1**
✎ (AL) = 05H

ARRAY1

...
00
00
01
00
00
00
01
00
01
00
...

01 | 程序设计基础-容量属性操作符SIZEOF

容量属性操作符 SIZEOF 格式

- **SIZEOF** 变量名

- 功能：返回变量所占存储单元的数目 (字节数)

示例

```
1. ARRAY1 DW 2 DUP(0,1), 1
```

- 若执行指令：**MOV AL, SIZEOF ARRAY1**

 (AL) = 0AH

SIZEOF 返回值 = LENGTHOF 返回值 × TYPE 返回值

ARRAY1

...
00
00
01
00
00
00
01
00
01
00
...

01 | 程序设计基础-强制属性操作符PTR

强制属性操作符 PTR (Pointer) 格式

- 类型名 PTR 标识符
- 类型名 PTR 存储单元寻址方式
 - 功能：将标识符的类型属性临时性地强制改为指定的类型，且该强制只在本条指令内有效

常用的类型

- 变量：BYTE、WORD、DWORD
- 标号：NEAR、FAR

01 程序设计基础-强制属性操作符举例

类型属性操作符示例

● 数据段定义如下：

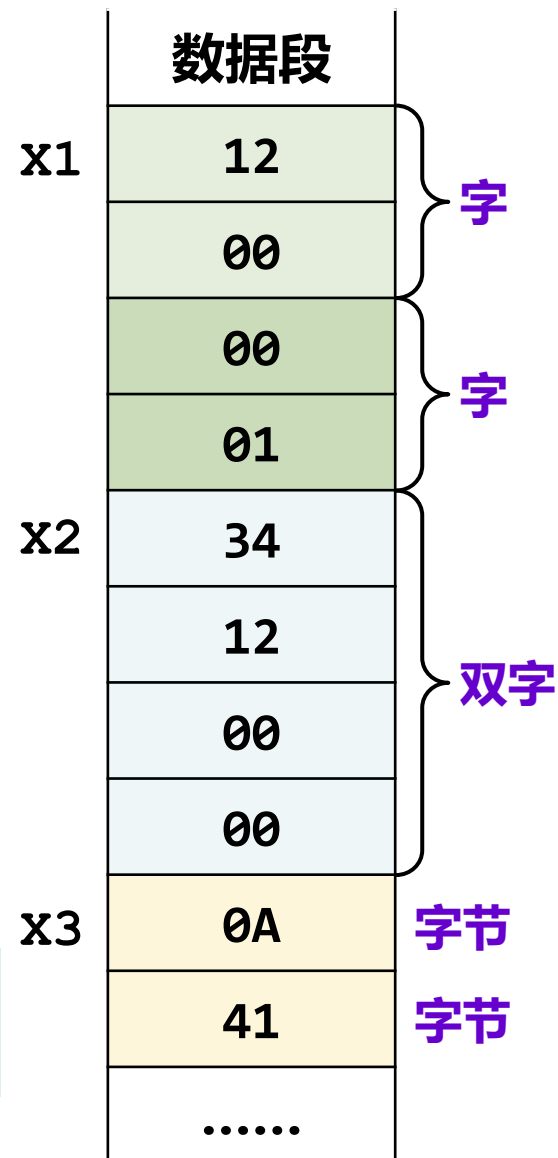
```
1. X1 DW 12H, 100H
2. X2 DD 1234H
3. X3 DB 10, 'A'
```

请问以下指令是否正确：

```
1. MOV AX, X1 ; ✓ (AX) = 0012H
2. MOV AX, X2 ; ✗
3. MOV AX, X3 ; ✗
```

应该修改为：

```
2. MOV AX, WORD PTR X2 ; ✓ (AX) = 1234H
3. MOV AX, WORD PTR X3 ; ✓ (AX) = 410AH
```



01 程序设计基础-运算符的优先顺序

运算符的优先级

- ① ()、<>、[]、LENGTHOF、SIZEOF
- ② PTR、SEG、OFFSET、TYPE
- ③ *、/、MOD、SHL、SHR
- ④ +、-
- ⑤ EQ、NE、LT、GT、LE、GE
- ⑥ NOT
- ⑦ AND
- ⑧ OR、XOR

优先权依次降低

为防止出错，建议使用 ()

01 | 程序设计基础-表达式

表达式

- 由**运算符**、**操作符**、**常量**和**变量**等构成的式子

注意

- 表达式要在程序的**汇编过程中**计算出具体数值的，各个部分的值需要在汇编期间完全确定
- 表达式中**不能出现寄存器**，因为在汇编期间程序尚未执行，寄存器中的数值不确定！！

表达式构成

- **数值表达式**：表示**数据**，可作为立即数使用
- **地址表达式**：表示**地址**，一般由符号地址等构成

01 | 程序设计基础-表达式的构成

数值表达式

- 由立即数、常量、字符或字符串与数值运算符构成
 - 字符或字符串使用时，应用引号括起来
- 由变量、标号与属性操作符构成
- 由**两个符号地址相减**构成表示二者之间距离的表达式

地址表达式

- 由变量、标号、运算符 (+、-)、地址计数器 \$ 构成;
- 典型的地址表达式: **符号地址 ± IMM**

01 程序设计基础-地址表达式举例1

地址表达式示例1

- 数据段定义如下:

```
1. X1 DW 12H, 100H
2. X2 DD 1234H
3. X3 DB 10, 'A'
```

- 地址表达式

```
1. MOV AX, X1+2 ; (AX)=0100H
2. MOV AX, X1+1 ; (AX)=0000H
3. MOV AL, BYTE PTR X2-2 ; (AL)=00H
4. MOV AL, X3+1 ; (AL)=41H
```

注意

- 地址表达式的类型与标识符的类型相同

	数据段
X1	12
	00
	00
	01
X2	34
	12
	00
	00
X3	0A
	41

01 | 程序设计基础-地址表达式举例2

地址表达式示例2

- 变量定义伪指令的操作数可以是表达式，包括**数值表达式**和**地址表达式**

```
1.      ORG 100H
2.      DATLIST DB 1,2,3
3.      ADDR1 DW DATLIST
4.      ADDR2 DD NEXT
5.      .....
6. NEXT: MOV AL, 12H
```

- 上述指令执行后：

 (ADDR1) = 0100H

 (ADDR2) 占用 4 个字节保存**指令的地址**（低字保存偏移地址，高字保存段地址）

01 | 程序设计基础-地址计数器引用符号\$

符号使用格式

- \$ [±IMM]

- 功能：用于构成地址或数据表达式
- 其值表示当前偏移量计数器的值
±IMM 的立即数

示例

```
1. ORG 100H
2. D1 DB 12H
3. W1 DW $, $
4. ORG $+3
5. B1 DB 43H
```

	数据段	
D1	12	0100H
W1	01	0101H
	01	0102H
	03	0103H
	01	0104H
空出 3个 单元		0105H
		0106H
		0107H
B1	43	0108H
	

01 | 程序设计基础-注意区分

变量的类型

- 字节、字、双字，可使用 PTR 等操作符改变类型

符号变量和符号常量

- 伪指令 DB、DW、DD 等定义符号变量，标识符表示地址
- 定义符 EQU、= 等定义符号常量，标识符表示数值

数值表达式和地址表达式

- 数值表达式中可使用多种运算符
 - 除关系运算、属性操作外，数值表达式大多不允许变量参与
- 地址表达式中变量的运算一般只使用 +、- 运算符
 - 一般形式为：符号地址 ± 立即数

某伪指令 `X EQU 12H` 中所定义的 `X` 是 ()

- ☐ A 变量
- ☒ B 常量
- ☐ C 标号
- ☐ D 段名

01 | 程序设计基础-子目录

80x86 汇编语言程序设计基础

- MASM 宏汇编语句结构及开发过程
- 汇编语句表达式和运算符
- 程序段的定义和属性
- 常用的系统功能调用

01 | 程序设计基础-源程序结构

汇编语言源程序结构

- 一个汇编语言源程序应包含**数据段**、**堆栈段**和**代码段**
 - **数据段**可以没有，也可以有一个或两个。用一个时一般为 DS 段，可多扩展一个 ES 段
 - **堆栈段**可以直接定义，也可以使系统默认分配
 - **代码段** CS 必须要有

注意

- 源程序中的各逻辑段顺序可以随意安排，但**通常数据段在前，代码段在后**
- 源程序形式有**完整结构**和**简化结构**两种

01 | 程序设计基础-完整源程序的一般结构

完整源程序的一般结构

可添加
堆栈段的
定义

```
DATA1 SEGMENT
```

```
.....
```

```
DATA1 ENDS
```

```
CODE1 SEGMENT
```

```
ASSUME CS:CODE1,DS:DATA1
```

```
START: MOV AX, DATA1
```

```
MOV DS, AX
```

```
;.....
```

```
.....
```

```
;.....
```

```
MOV AX, 4C00H
```

```
INT 21H
```

```
CODE1 ENDS
```

```
END START
```

段起始的标号，
合法标识符即可

源程序结束伪
指令 END

逻辑段的定义

段说明 ASSUME

只要有数据段，均需
指令为段寄存器赋值

返回 DOS 的功能调用，
是所有汇编语言源程
序的结束语句

01 | 程序设计基础-段定义伪指令

段定义伪指令格式

```
段名 SEGMENT  
.....  
;段体  
段名 ENDS
```

段名

- 合法标识符，**首尾段名要一致**
- 段名作为操作数时，表示**立即数**，其值为**段地址**

段体

- 数据段中主要为**数据定义伪指令**
- 代码段中主要为**汇编指令**

01 | 程序设计基础-段约定伪指令ASSUME

段约定伪指令格式

- **ASSUME** 段寄存器名:段名[, 段寄存器名:段名, ...]

功能

- 指明**逻辑段**和**段寄存器**的对应关系
 - 并不会为段寄存器赋值，需要指令完成赋值

示例

- **ASSUME CS:CODE1, DS:DATA1, ES:DATA2**
- 段寄存器与逻辑段之间**不一定**是一一对应的关系

取消段指定

- **ASSUME 段寄存器名:NOTHING**

01 | 程序设计基础-关于堆栈段

堆栈段的定义

```
段名 SEGMENT STACK
.....
;段体
段名 ENDS
```

注意

- 若源程序中无堆栈段定义，则**系统会自动分配一个堆栈段**，但连接时会产生一个警告信息：
warning xxxx: no stack segment
 - 警告信息不影响程序正常运行
- SS 和 SP 的赋值可以指令完成，也可通过参数 STACK 自动设置

01 | 程序设计基础-源程序结束伪指令

源程序结束伪指令格式

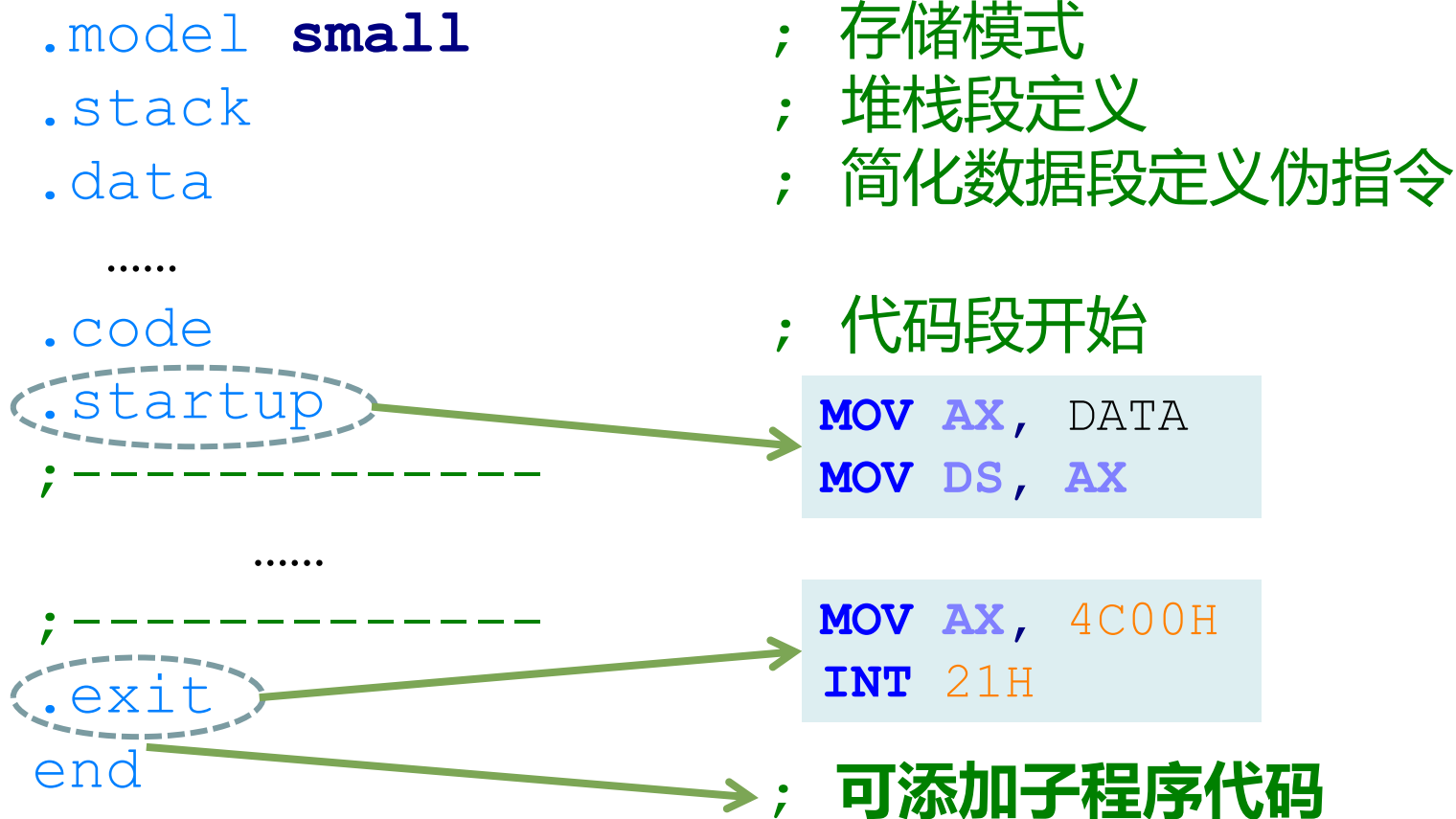
- **END** [地址]

功能

- 表示汇编语言源程序结束
- 可选的地址用于**指出程序的起点**，一般为标号或过程名表示的符号地址

01 | 程序设计基础-简化的源程序结构

简化的源程序结构



01 | 程序设计基础-存储模式说明

存储模式说明

- 简化模式下，必须使用 **.MODEL** 伪指令指明存储模式
 - 汇编程序根据该伪指令生成相应的 **ASSUME** 语句等

可选的存储模式

- **TINY**: COM 文件使用
- **SMALL**: 一个数据段、一个代码段，只支持段内转移
- **COMPACT**: 多个数据段，一个代码段
- **MEDIUM**: 一个数据段，多个代码段，可以做段间转移
- **LARGE**: 多个数据段，多个代码段，必须 64KB 以下数组
- **HUGE**: 多个数据段，多个代码段，可用 64KB 以上数组
- **FLAT**: 80386 以上的 CPU 模式下使用

01 | 程序设计基础-简化段定义伪指令

简化段段名

- `.CODE`: 代码段, 其后是汇编指令
- `.DATA`: 数据段, 其后是变量定义伪指令
- `.STACK [堆栈字节数]`: 堆栈段, 缺省 1024 字节
- `.STARTUP`: 用于代码段的开始, 可自动初始化段寄存器
- `.EXIT`: 用于结束程序的运行

01 | 程序设计基础-处理器选择伪指令

缺省方式

- 缺省方式下，汇编程序只处理 16 位系统的指令

32 位系统

- 需要在源程序开始指明处理器类型
 - `.8086`: 默认类型
 - `.286`、`.286P`、**`.386`**、`.386P`、`.486`、`.486P`、`.586`、`.586P`、`.686`、`.686P`
 - `P` 表示可使用特权指令
- 一个源程序中可混合使用多种处理器选择伪指令

01 | 程序设计基础-子目录

80x86 汇编语言程序设计基础

- MASM 宏汇编语句结构及开发过程
- 汇编语句表达式和运算符
- 程序段的定义和属性
- 常用的系统功能调用

01 | 程序设计基础-DOS、BIOS功能调用

BIOS (Basic Input Output System) 功能调用

- BIOS 例行程序是系统加电自检时，所用到的主要 I/O 设备程序以及接口控制等功能模块；
 - 直接调用这些模块，使程序员不必了解硬件接口的特性

DOS (Disk Operating System) 功能调用

- DOS 磁盘操作系统，有 IO.SYS 和 MSDOS.SYS 两个模块
- DOS 模块提供了更多更必要的测试，使用 DOS 调用比使用相应功能的 BIOS 操作更简易，而且对硬件的依赖性更少些

01 程序设计基础-DOS功能调用的使用方法

DOS 功能调用使用方法

1. 设置入口参数

- ① (AH)=功能调用号
- ② 其他参数根据功能需要而不同，也可没有

2. 执行指令 **INT 21H**，调用相应的 DOS 模块

3. 获得出口参数

- ① 根据不同的功能从不同的位置获取，也可没有

常用的 DOS 功能调用

- 单字符输入 → **01 号**功能调用
- 单字符输出 → **02 号**功能调用
- 字符串输出 → **09 号**功能调用
- 字符串输入 → **0A 号**功能调用
- 程序结束返回 → **4C 号**功能调用

01 程序设计基础-单字符输入(01号功能调用)

01 号功能调用

- 功能调用号：01H
- 入口参数：无
- 出口参数：(AL)=输入字符的 ASCII 码

01 号功能调用示例

- 从键盘上获取一位十进制数据
 - 指令序列：

```
MOV AH, 01H
INT 21H
```
 - 执行时状态：光标等待键盘输入，输入一个字符“1”后返回，在屏幕上可看到输入的字符
 - 出口参数：(AL)=31H
 - 结果处理：所需的十进制数据 = (AL) - 30H

01 程序设计基础-单字符输出(02号功能调用)

02 号功能调用

- 功能调用号：02H
- 入口参数：(DL)=输出字符的 ASCII 码
- 出口参数：无

02 号功能调用示例

- 输出显示变量 CHAR 所存放的字母

- 指令序列：

```
MOV AH, 02H  
MOV DL, CHAR  
INT 21H
```

- 若 CHAR 变量定义为 CHAR DB 'F'，则以上指令序列执行完毕会在屏幕上显示字符 "F"

01 程序设计基础-字符串输出(09号功能调用)

09 号功能调用

- 功能调用号：09H
- 入口参数：(DS:DX)=待输出字符串的起始地址
 - 输出字符串应事先存放于存储器，且必须以“\$”为结束
- 出口参数：无

09 号功能调用示例

- 试将变量 String 保存的字符串显示出来
 - 变量定义：String DB "ABCD\$"
 - 指令序列：

```
MOV AX, SEG String
MOV DS, AX
LEA DX, String
MOV AH, 09H
INT 21H
```


01 程序设计基础-字符串输入(0A号功能调用)

0A 号功能调用

- 功能调用号：0AH
- 入口参数：(DS:DX)=待输入字符串的起始地址
 - 存放输入字符串的缓冲区必须事先按格式定义
- 缓冲区定义格式如： BUF DB n, ?, n DUP(?)

缓冲区可接收
的最大字符数

实际接收的
字符个数

接收的
字符串

- 出口参数：在缓冲区中保存输入字符串及其长度

01 | 程序设计基础-0A号功能调用注意事项

输入字符串的长度限制

- 输入字符串的长度受限于缓冲区的第一个字节数据
- 若缓冲区定义：BUF DB 10, ?, 10 DUP(0)
则最多可输入 9 个字符
 - 字符串输入的回车结束符也会保存于缓冲区中

缓冲区初始化

- 若输入的字符串最终需要输出，则可在缓冲区定义时，将其初始化为 "\$"，以使输出的字符串能够正常结束
- 注意最后输入的回车符的处理

01 | 程序设计基础-DOS功能调用举例1

DOS 功能调用示例

- 数据段的定义:

```
DATA1 SEGMENT
    buf DB 5, ?, 5 DUP(0)
DATA1 ENDS
```

输入字符串的指令序列:

```
MOV AX, SEG buf
MOV DS, AX
LEA DX, buf
MOV AH, 0AH
INT 21H
```

buf
	05
	04
	61
	62
	63
	64
	0D

以上指令序列执行, 光标等待键盘输入, 输入字符串“abcd”, 以回车结束, 即返回, 存储单元的状态如右图

01 | 程序设计基础-DOS功能调用举例2

DOS 功能调用示例

```
DATA1 SEGMENT
    buf DB 10, ?, 10 DUP("$")
DATA1 ENDS
CODE1 SEGMENT
    ASSUME CS:CODE1, DS:DATA1
START: MOV AX, DATA1
        MOV DS, AX
        LEA DX, buf
        MOV AH, 0AH
        INT 21H
        MOV AH, 2
        MOV DL, 10
        INT 21H
        MOV DL, 13
        INT 21H
        LEA DX, buf+2
        MOV AH, 09H
        INT 21H
        MOV AX, 4C00H
        INT 21H
CODE1 ENDS
END START
```

最长 9 个字符串输入 (0A)

显示换行、回车 (02)

显示输入的字符串 (09)

最后结束程序 (4C00)

输入字符串的 0A 号 DOS 功能调用的入口参数是（ ）

☒ A DS:DX

☐ B SS:BP

☐ C DS:AX

☐ D DX:AX

本节小结

- 必须熟练掌握汇编语言源程序的完整结构格式
- 掌握常用的变量、常量定义伪指令的使用
 - 理解常量和变量的区别
- 熟悉常用的属性操作符，能够适当应用数值和地址表达式
- 掌握并能熟练使用常用的 DOS 功能调用
 - 理解 DOS 功能调用的实质，熟悉进出口参数的设置

目录

- 01 80x86 汇编语言程序设计基础
- 02 汇编语言顺序、分支程序设计
- 03 汇编语言循环、串处理程序设计
- 04 子程序设计

02 | 上节回顾-串处理、转移、循环指令

简答题

- 串处理操作中，传送的数据个数和方向分别由什么寄存器或标志位指出？
- 条件转移指令的条件。例如：JGE 与 JAE、JC 和 JB 的区别。
- 循环指令 LOOP/LOOPZ/LOOPNZ 退出循环的条件分别是什么？

02 | 顺序程序-程序设计概述

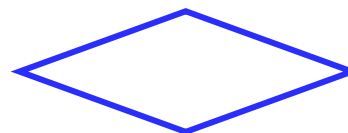
汇编源程序编程基本步骤

1. 分析问题，建立数学模型
2. 确定算法
3. 设计程序流程，画出**程序流程图**
4. 定义变量，分配内存单元 (定义数据段)
5. 编写源程序
6. 上机调试、修改

02 | 顺序程序-程序流程图的基本元素

汇编源程序编程基本步骤

- **起止框**：圆角矩形
 - 一个出口，**或**一个入口
- **执行框**：矩形
 - 一个出口**和**一个入口
- **判断框**：菱形
 - 一个入口和**多个出口**
- **流向线**：直线/折线箭头
 - 有向连接线



02 | 顺序程序-特点

顺序程序特点

- 顺序结构是**最基本**的程序结构
 - 一般较为简单
 - 线性结构程序
 - 从第一条指令开始，**按顺序**执行程序中的每条指令

示例1：将字节变量 X 的数据乘以 2，结果存入字变量 Y 中

- 假设字节变量 X 为无符号数
- 可用不同的方式实现：
 - 加法指令： $X+X \rightarrow Y$
 - 乘法指令 (**MUL**)
 - 移位指令 (**SHL**)

02 | 顺序程序-示例1程序

示例1：将字节变量 X 的数据乘以 2，结果存入字变量 Y 中

```
DATA SEGMENT
    X DB 12H
    Y DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AL, X
        MOV CL, 2
        MUL CL
        MOV Y, AX
        MOV AX, 4C00H
        INT 21H
CODE ENDS
    END START
```

```
DATA SEGMENT
    X DB 12H
    Y DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AL, X
        MOV AH, 0
        SHL AX, 1
        MOV Y, AX
        MOV AX, 4C00H
        INT 21H
CODE ENDS
    END START
```

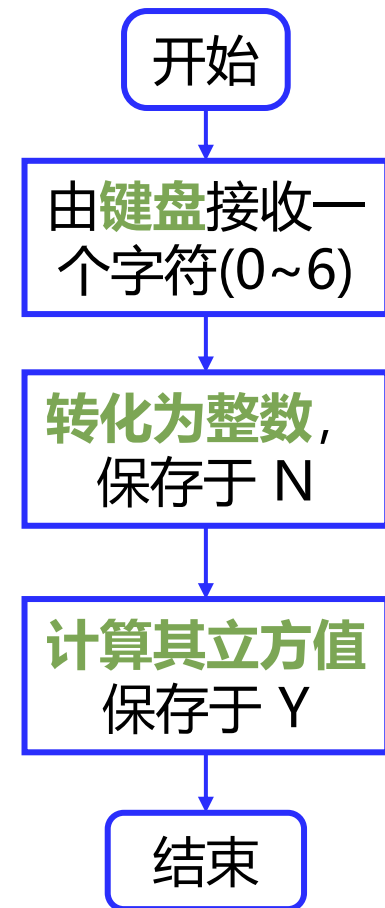
02 | 顺序程序-示例2

示例2：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序。
数据 N 由键盘输入， N 、 Y 均为无符号字节数

分支结构示例3

循环结构示例1

- 由键盘输入数据 N
 - 要做字符与数值的转换
- 方案一：直接求解
 - 由 N 的数值直接进行乘法
- 方案二：查表求解
 - 事先建立 $0 \sim 6$ 的立方值表，根据输入值查表确定其立方值



02 | 顺序程序-示例2程序-方案一直接求解

示例2：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序

- 数据段定义

- `N DB ?`
`Y DB ?`

- 从键盘输入 N 的值

- 注意输入的是字符，要转换成数值

`SUB AL, 30H`

- 求立方值

- $Y=N \times N \times N$

```
DATA SEGMENT
    N DB ?
    Y DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE,DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AH, 1 ; 单字符输入, 存于 AL 中
        INT 21H
        AND AL, 0FH ; 取输入字符的低 4 位
        MOV N, AL
        MUL AL ; AL=N*N
        MUL N ; AL=N*N*N
        MOV Y, AL ; Y=N*N*N
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

02 | 顺序程序-示例2程序-方案二查表求解1

示例2：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序

- 数据段定义
 - 建立 0~6 的立方表
 - `TAB1 DB 0, 1, 8, 27, 64, 125, 216`
- 代码段的改变
 - 获得 0~6 的整数后，采用**相对寻址**的方式查得立方值
- 指令序列：
 - `MOV BL, AL`
 - `MOV BH, 0`
 - `MOV AL, TAB1[BX]`
 - `MOV Y, AL`

02 | 顺序程序-示例2程序-方案二查表求解2

示例2：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序

```
DATA SEGMENT
    TAB1 DB 0,1,8,27,64,125,216
    N DB ?
    Y DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AH, 01H
        INT 21H
```

3 次方表

```
    AND AL, 0FH
    MOV N, AL
    MOV BL, AL
    MOV BH, 0
    MOV AL, TAB1[BX]
    MOV Y, AL
    MOV AX, 4C00H
    INT 21H
CODE ENDS
END START
```

转换成数值

MOV AL, TAB1 [BX] 源操作数的寻址方式是 ()

- ☐ A 寄存器寻址
- ☐ B 寄存器间接寻址
- ☒ C 寄存器相对寻址
- ☐ D 立即寻址

02 | 分支程序-分支程序设计

指令寻址方式

- 程序转移地址的寻址方式
 - 无条件转移指令
 - 条件转移指令

分支程序

- 单分支程序 **IF-THEN**
 - 满足某个条件，则跳转
- 双分支程序 **IF-THEN-ELSE**
 - 由一个条件的真假，判断跳转执行两个不同的处理
- 多分支程序 **SWITCH-CASE**
 - 多层嵌套分支，或多个判断条件，决定多个不同的处理

02 | 分支程序-程序转移寻址方式

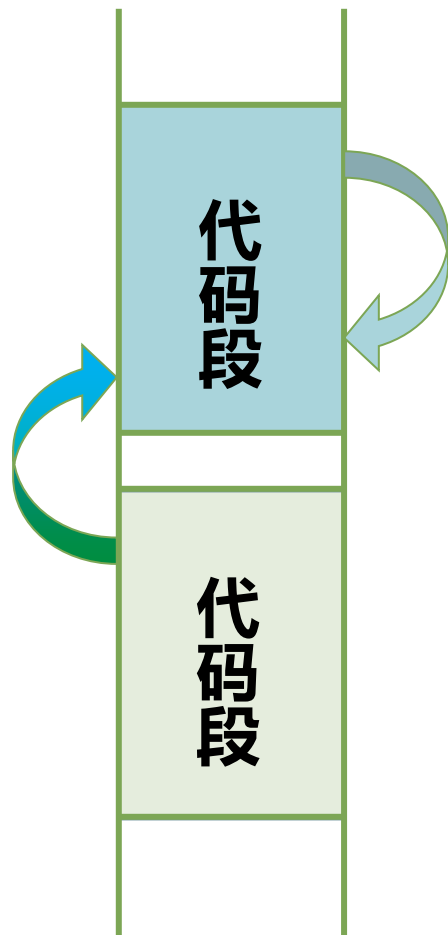
转移分类

- 根据转移的范围分为：**段内转移**和**段间转移**
- 根据操作数寻址方式分为：**直接转移**和**间接转移**

转移类型

- **段内转移**：在同一代码段内转移，只修改 IP，CS 不变
 - 段内直接转移
 - 段内间接转移
- **段间转移**：在不同代码段之间的转移，同时修改 CS 和 IP
 - 段间直接转移
 - 段间间接转移

1.	JMP 1000H	; 段内直接转移
2.	JMP CX	; 段内间接转移
3.	JMP 2000H:0100H	; 段间直接转移
4.	JMP DWORD PTR [SI]	; 段间间接转移



02 | 分支程序-转移指令

无条件转移指令

- **JMP** <OPRD>

- 使程序**无条件地**转向 OPRD 所指定的位置执行
- OPRD 可用寄存器寻址和存储器寻址方式表示

条件转移指令

- **无符号数比较**: JE/JZ、JNE/JNZ、JA/JNBE、JAE/JNB、JB/JNAE、JBE/JNA
- **有符号数比较**: JE/JZ、JNE/JNZ、JG/JNLE、JGE/JNL、JL/JNGE、JLE/JNG
- **单个标志位测试**: JZ/JNZ、JS/JNS、JC/JNC、JO/JNO、JP/JPE、JNP/JPO、JCXZ

02 | 分支程序-条件转移指令1

无符号数比较的转移指令

指令	英文全称	检测的转移条件	功能描述
JE/JZ	Jump if Equal Jump if Zero	ZF=1	等于，则转移
JNE/JNZ	Jump if Not Equal Jump if Not Zero	ZF=0	不等，则转移
JA/JNBE	Jump if Above Jump if Not Below or Equal	CF=0 且 ZF=0	大于（不小于等于），则转移
JAE/JNB	Jump if Above or Equal Jump if Not Below	CF=0	大于等于（不小于），则转移
JB/JNAE	Jump if Below Jump if Not Above or Equal	CF=1	小于（不大于等于），则转移
JBE/JNA	Jump if Below or Equal Jump if Not Above	CF=1 或 ZF=1	小于等于（不大于），则转移

- A — 大于； B — 小于； E/Z — 等于； N — 否

02 | 分支程序-条件转移指令2

有符号数比较的转移指令

指令	英文全称	检测的转移条件	功能描述
JE/JZ	Jump if Equal Jump if Zero	ZF=1	等于，则转移
JNE/JNZ	Jump if Not Equal Jump if Not Zero	ZF=0	不等，则转移
JG/JNLE	Jump if Greater Jump if Not Less or Equal	ZF=0 且 SF=OF	大于（不小于等于），则转移
JGE/JNL	Jump if Greater or Equal Jump if Not Less	SF=OF	大于等于（不小于），则转移
JL/JNGE	Jump if Less Jump if Not Greater or Equal	SF≠OF	小于（不大于等于），则转移
JLE/JNG	Jump if Less or Equal Jump if Not Greater	ZF=1或SF≠OF	小于等于（不大于），则转移

- G — 大于； L — 小于； E/Z — 等于； N — 否

02 | 分支程序-条件转移指令3

单个标志位测试的转移指令

指令	英文全称	检测的转移条件	功能描述
JZ	Jump if Zero	ZF=1	结果为0，则转移
JNZ	Jump if Not Zero	ZF=0	结果不为0，则转移
JS	Jump if Signed	SF=1	结果为负数，则转移
JNS	Jump if Not Signed	SF=0	结果为正数，则转移
JC	Jump if Carry	CF=1	产生进/借位，则转移
JNC	Jump if Not Carry	CF=0	未产生进/借位，则转移
JO	Jump if Overflow	OF=1	结果溢出，则转移
JNO	Jump if Not Overflow	OF=0	结果未溢出，则转移
JP/JPE	Jump if Parity / Parity Even	PF=1	结果有偶数个1，则转移
JNP/JPO	Jump if No Parity / Parity Odd	PF=0	结果有奇数个1，则转移
JCXZ	Jump if Register CX is Zero	CX=0	CX为零，则转移

02 | 分支程序-分支结构程序

双分支结构

- 双分支程序一般为简单的程序，类似于高级语言的 **if 语句**
 - 由一个**条件转移指令**构成
 - 可以是**单分支结构** (一条分支为空)，也可以是**双分支结构**

多分支结构

- 多分支程序类似于高级语言的 **switch 语句**
 - 可由**无条件转移指令**构成，也可由**条件转移指令**构成
 - 常用的方法有**逻辑分解法**、**地址表法**、**转移表法**等

注意

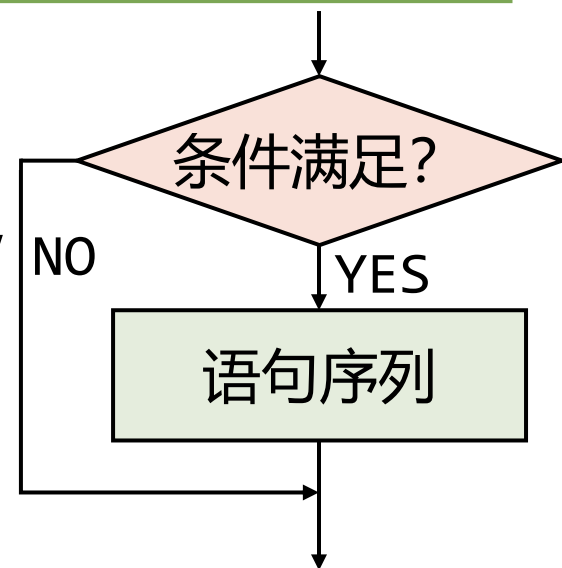
- 汇编语言程序中，注意各分支的安排，**适当地使用无条件转移指令**

02 | 分支程序-单分支结构程序

特点

- **单分支程序**实际上是双分支程序的一种, 另一分支为空, 或直接退出程序
 - 程序执行时, 通过条件的判断, **可能跳过某些语句序列**
 - 类似语句

```
IF (.....)
THEN {.....}
```
- 一般将分支置于条件转移指令之后, 若**不需执行**, 则可**直接跳过**



推荐形式

```
JNcc EXIT
<语句序列>
```

```
EXIT: .....
```

注意

- 注意不要使用多余的转移指令

不推荐形式!

```
Jcc NEXT
JMP EXIT
```

```
NEXT: <语句序列>
EXIT: .....
```

02 | 分支程序-单分支结构程序示例1

编写程序，完成 $z = |x - y|$ 的计算，假定 x 、 y 为有符号字变量

- 思路

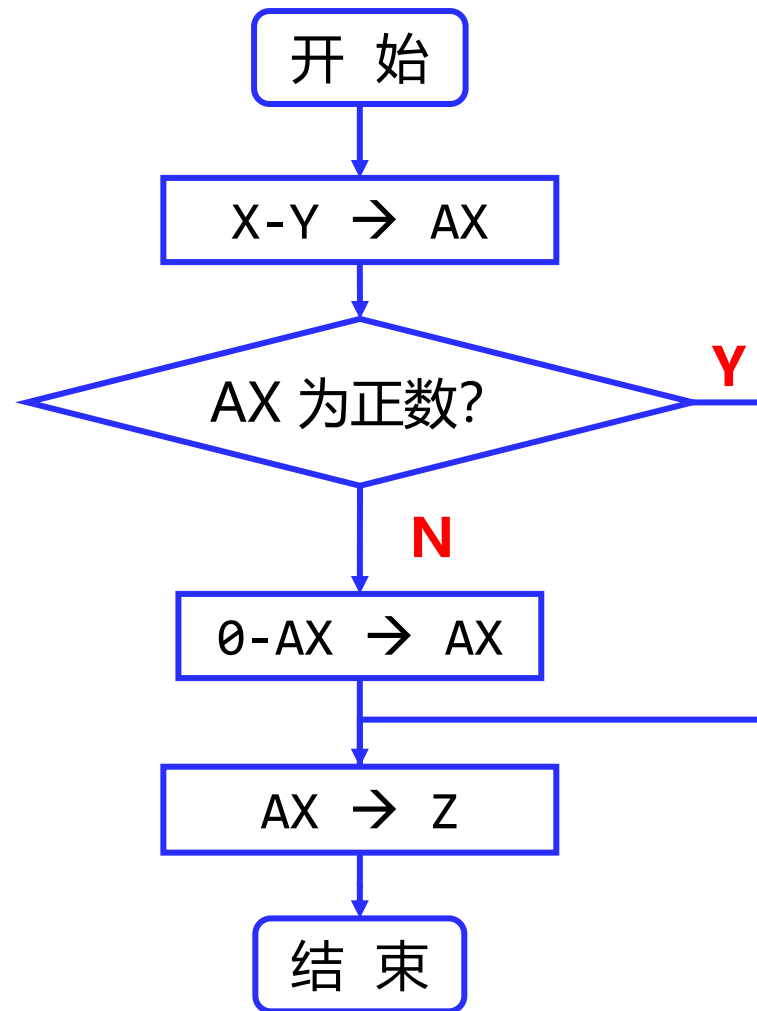
- 减法： $x - y \rightarrow AX$
- 若 $AX \geq 0$ ，则结果即为所求；
否则，求 AX 的相反数即可

- 程序段：

```
SUB AX, Y  
JGE EXIT  
NEG AX
```

EXIT:

若不溢出，则也可以用 **JNS**，思考为什么？



02 | 分支程序-单分支结构程序示例1参考

编写程序，完成 $z = |x - y|$ 的计算，假定 x 、 y 为有符号字变量

```
DATA SEGMENT
```

```
    X DW 4
```

```
    Y DW 6
```

```
    Z DW ?
```

```
DATA ENDS
```

```
CODE SEGMENT
```

```
    ASSUME CS:CODE, DS:DATA
```

```
START: MOV AX, DATA
```

```
        MOV DS, AX
```

```
    MOV AX, X
```

```
    SUB AX, Y
```

```
    JGE EXIT ;如果AX≥0, 则退出
```

```
    NEG AX ;否则取相反数
```

```
EXIT: MOV Z, AX
```

```
    MOV AX, 4C00H
```

```
    INT 21H
```

```
CODE ENDS
```

```
END START
```

若 **AX**、**BX** 均表示**有符号数**，执行指令 **CMP AX, BX** 后，OF 标志位为 0，则符合 **AX ≥ BX** 的转移指令为 ()

A

JAE

B

JGE

C

JNS

D

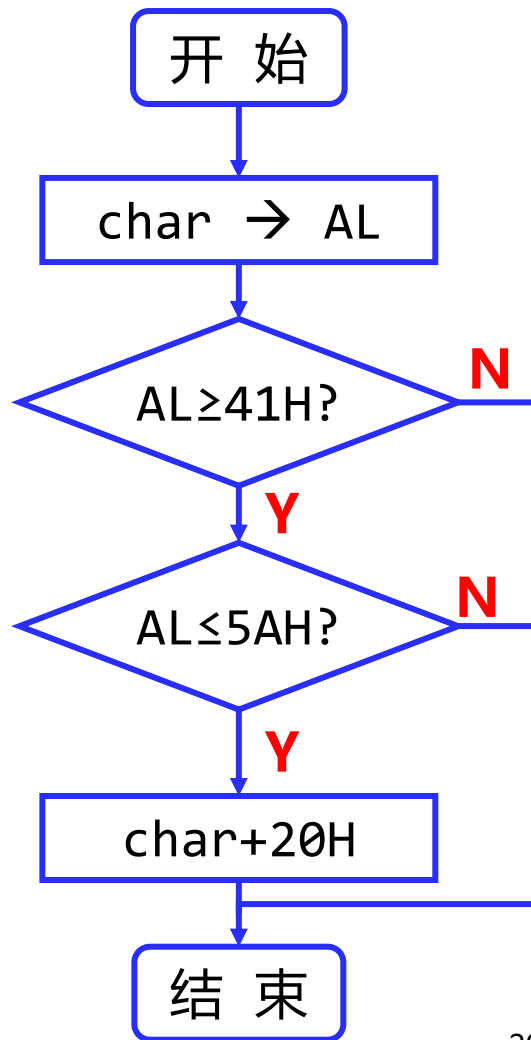
JNC

02 | 分支程序-单分支结构程序示例2

试判断字节变量 char 是否为大写字母，若是，编写程序将其变成小写字母

- 思路

- 先判断 char 变量是否为大写字母
- 两次条件转移的区间判断
- 若是，则转换 (+20H)
- 否则，不处理，或给出提示信息



02 | 分支程序-单分支结构程序示例2参考

试判断字节变量 char 是否为大写字母，若是，编写程序将其变成小写字母

- 大写字母变小写字母程序段

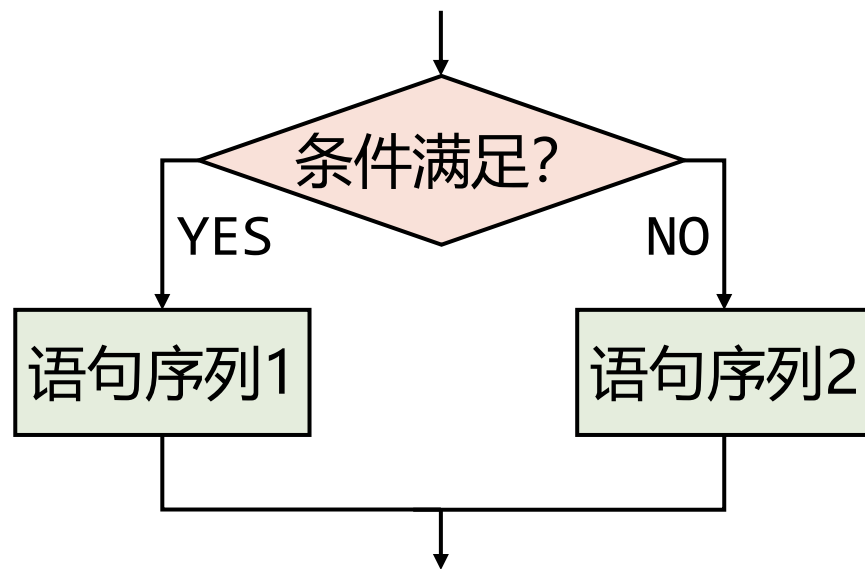
```
.....  
char DB 'F'  
.....  
MOV AL, char ;取变量 char  
CMP AL, 'A' ;判断 char 是否为 'A'-'Z' 之间的值  
JB Next ;ASCII 码小于 'A', 则跳出  
CMP AL, 'Z'  
JA Next ;ASCII 码大于 'Z', 则跳出  
ADD char, 'a' - 'A' ;若是大写字母, 则转换为小写  
Next: .....
```

02 | 分支程序-双分支结构程序

特点

- 双分支程序通过一个条件真假，执行**两种**不同的处理
 - 程序执行时，通过条件的判断，**可能跳过某些语句序列**
 - 类似语句

```
IF (.....)
THEN {.....}
ELSE {.....}
```



注意

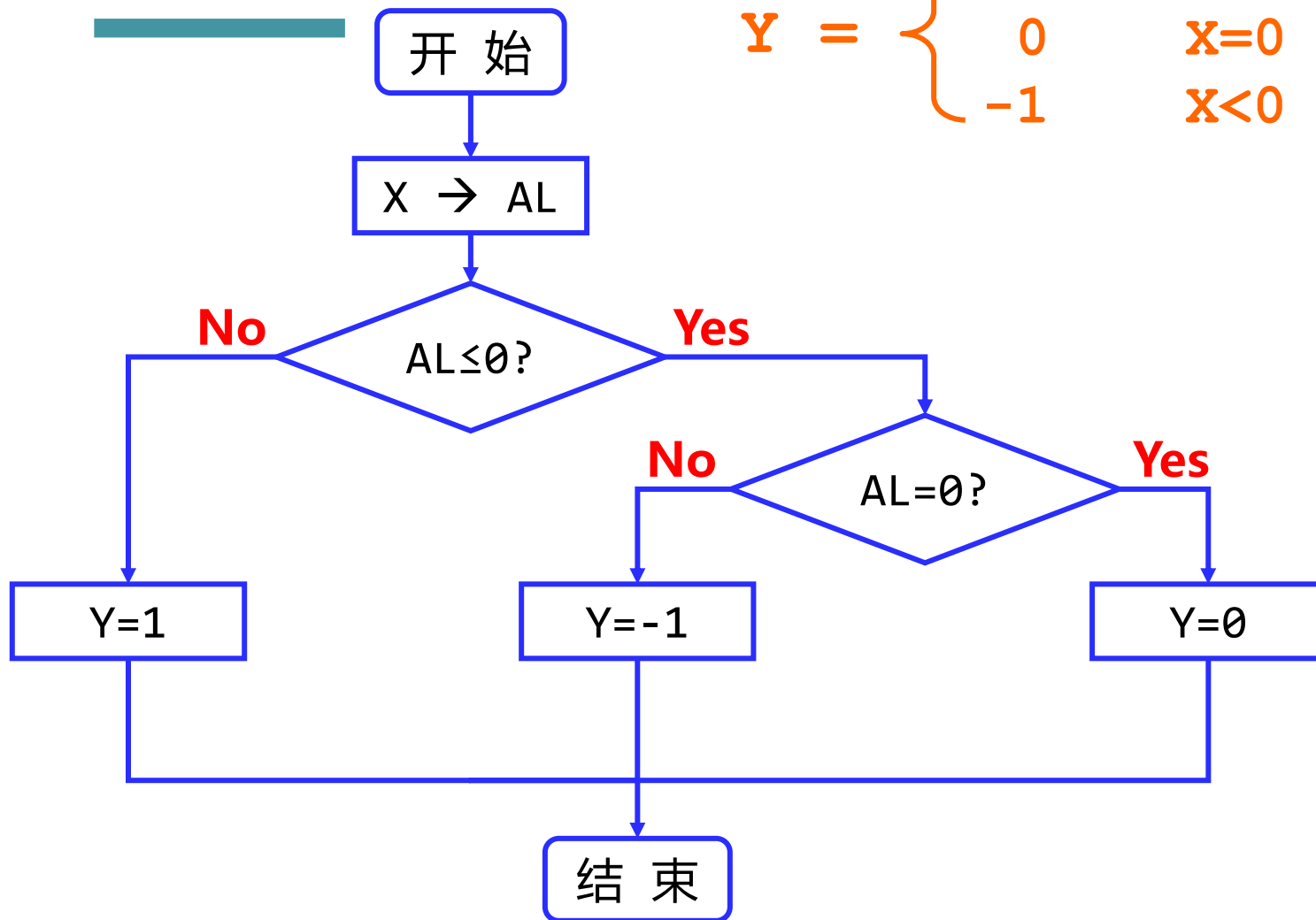
- 各个分支的代码是顺序存放的，注意**多个分支不能同时执行**
- 各分支之后应有转移指令，转向程序的后续部分

```
Jcc NEXT
<语句序列2>
JMP EXIT
NEXT: <语句序列1>
EXIT: .....
```

02 | 分支程序-双分支结构程序示例1

编写程序段，完成下面计算公式，其中变量 X 和 Y 都是有符号的字节变量

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



02 | 分支程序-双分支结构程序示例1参考

编写程序段，完成下面计算公式，其中变量 X 和 Y 都是有符号的字节变量

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

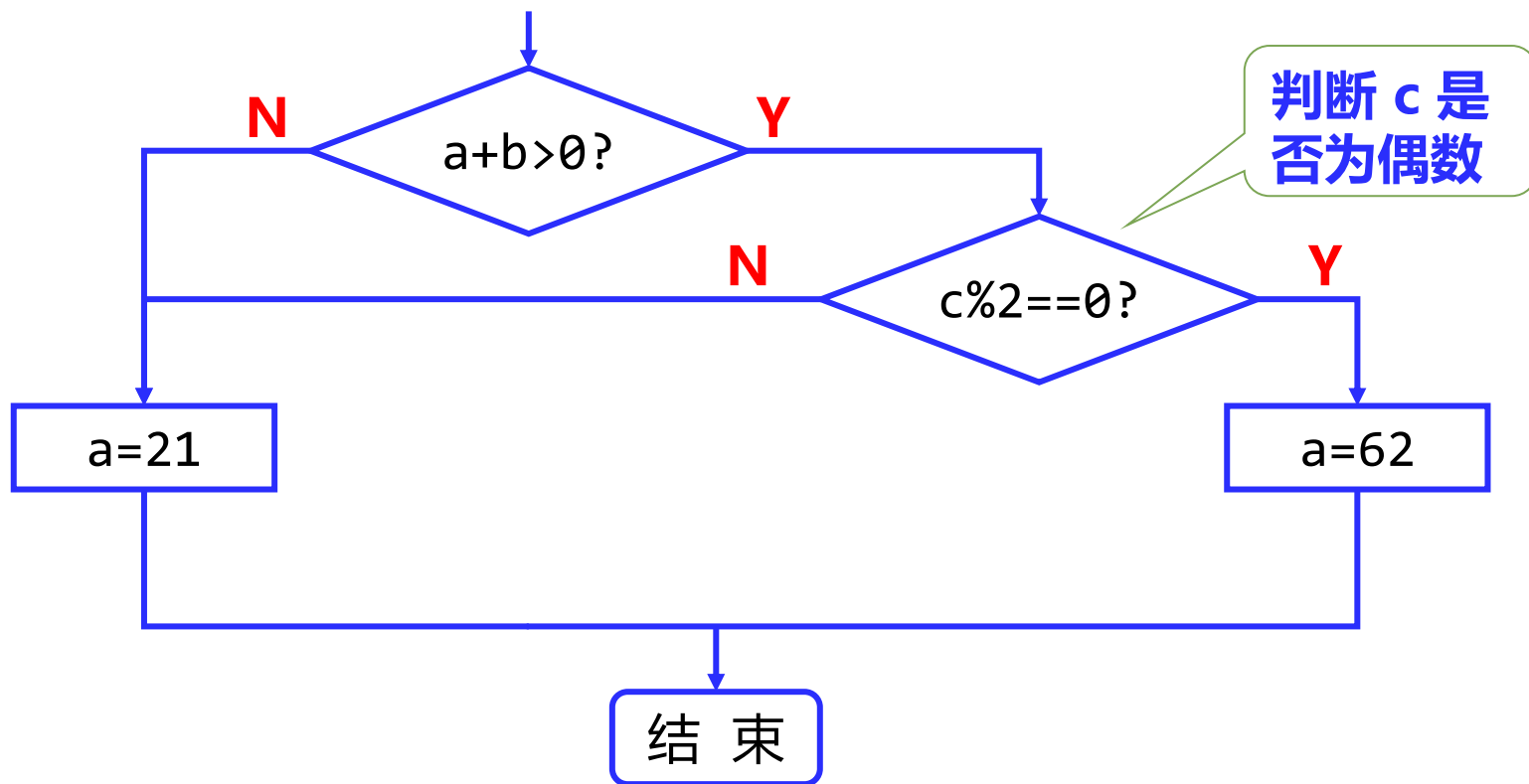
```
...  
X DB ?  
Y DB ?  
...  
  
MOV AL, X  
CMP AL, 0  
JG positive  
CMP AL, 0 ;可省略  
JZ zero  
MOV Y, -1
```

```
        JMP exit  
zero:  
        MOV Y, 0  
        JMP exit  
positive:  
        MOV Y, 1  
exit:  
        MOV AX, 4C00H  
        INT 21H
```

02 | 分支程序-双分支结构程序示例2

把下列 C 语言的语句改写成等价的汇编语言程序段 (不考虑运算过程中的溢出), 变量 a、b 和 c 都是有符号数的字变量

```
If (a+b>0 && c%2==0 ) a = 62  
else a = 21
```



02 | 分支程序-双分支结构程序示例2参考

把下列 C 语言的语句改写成等价的汇编语言程序段 (不考虑运算过程中的溢出), 变量 a、b 和 c 都是有符号数的字变量

```
If (a+b>0 && c%2==0 ) a = 62
else a = 21
```

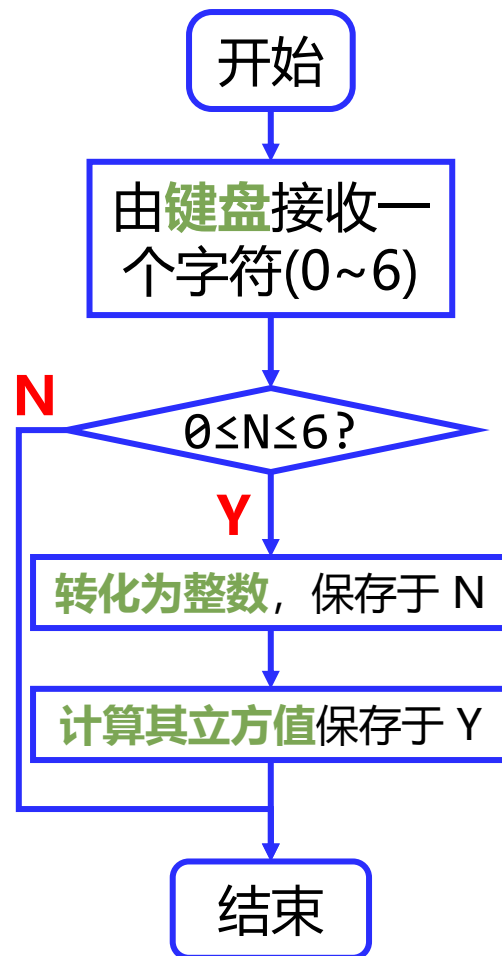
```
DATA1 SEGMENT
    A DW 5
    B DW 6
    CC DW 7 ;C 是保留字, 不能使用
DATA1 ENDS
CODE1 SEGMENT
    ASSUME CS:CODE1, DS:DATA1
START: MOV AX, DATA1
        MOV DS, AX
        MOV AX, A
        ADD AX, B
```

```
    CMP AX, 0
    JLE _ELSE ;判断 >0?
    TEST CC, 1
    JNZ _ELSE ;判断偶数?
    MOV A, 62
    JMP NEXT
_ELSE: MOV A, 21
NEXT:  MOV AX, 4C00H
        INT 21H
CODE1 ENDS
    END START
```

02 | 分支程序-双分支结构程序示例3

示例3：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序。 顺序结构示例2
数据 N 由键盘输入， N 、 Y 均为无符号字节数。 添加输入控制

- 由键盘输入数据 N
 - 对于输入的数据进行范围判别
 - 若 $0 \leq N \leq 6$ ，则直接求 N^3
 - 否则，结束
- 对于取值范围的判断
 - 分别判断 $N \geq 0$ 、 $N \leq 6$



02 | 分支程序-双分支结构程序示例3参考

示例3：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序。
数据 N 由键盘输入，N、Y 均为无符号字节数。添加输入控制

```
DATA SEGMENT
    N db ?
    Y db ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AH, 1
        INT 21H
        CMP AL, 30H
        JB EXIT ;小于30, 则退出
```

```
        CMP AL, 36H
        JA EXIT ;大于36, 则退出
        AND AL, 0FH
        MOV N, AL
        MUL AL
        MUL N
        MOV Y, AL
EXIT: MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

02 | 分支程序-多分支结构程序

多分支程序的实现方法

- 逻辑分解法

- 将多分支转化为多个双分支

- 地址表法

- 数据段中保存多个分支入口地址的地址表
- 程序执行时，使用段内间接转移指令 **JMP**，转入正确的分支

- 转移表法

- 代码段中设置对应于每一分支的转移指令——转移表；
- 程序执行时，使用段内间接转移指令 **JMP**，转到转移表中对应的指令，从而转入正确的分支

02 | 分支程序-多分支结构程序示例

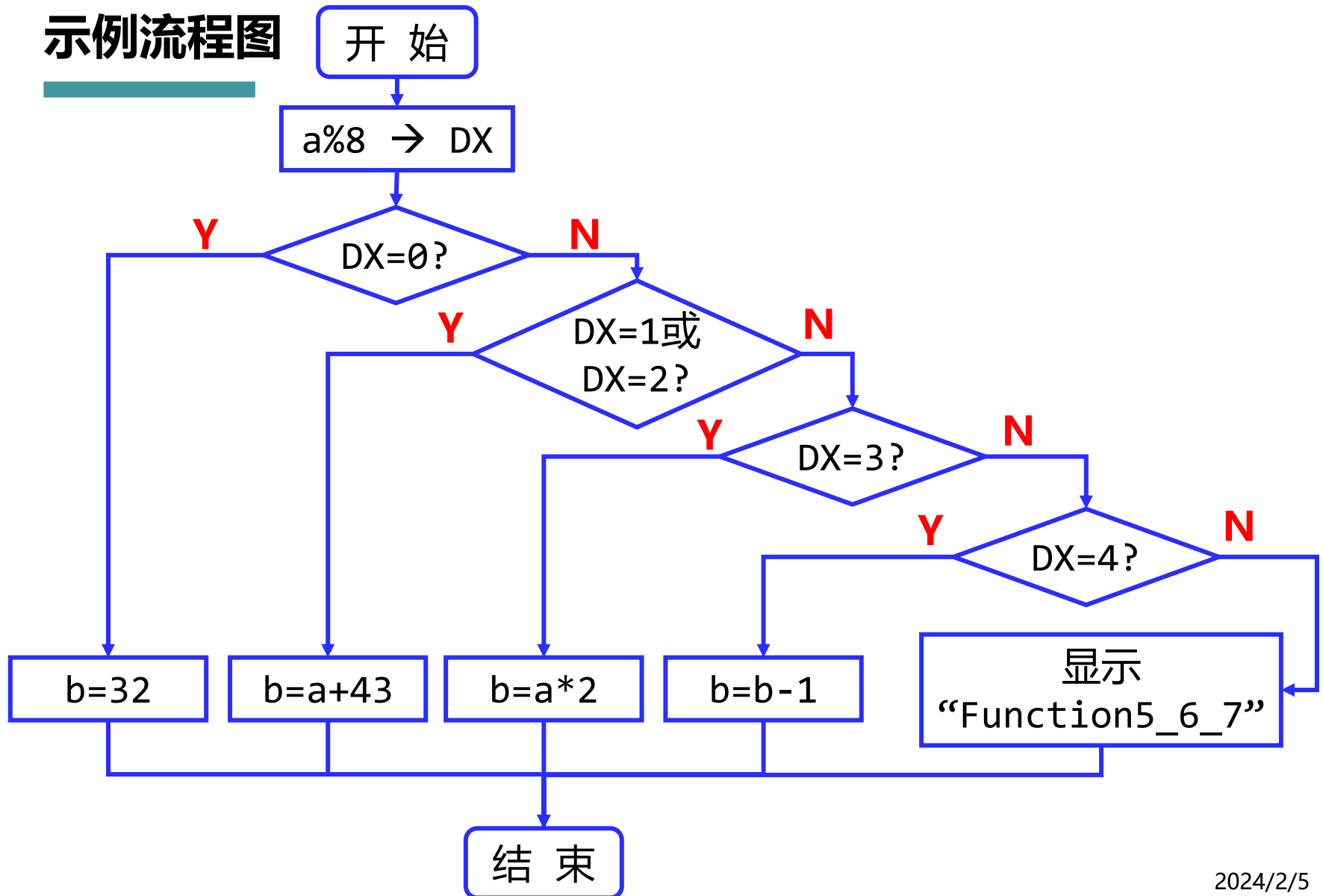
示例：使用不同的方法实现 C 语言 switch 语句的功能，其中 a 和 b 是有符号的字变量

- 多分支程序，条件 (a%8) 的可能值为 0~7，与每一个分支一一对应
 - 求 a%8 的方法
 - ① 除法求余数
 - ② a 的高 13 位清零
- 使用逻辑分解法、地址表法、转移表法分别实现

```
switch(a%8){
    case0: b=32;
           break;
    case1:
    case2: b=a+43;
           break;
    case3: b=2*a;
           break;
    case4: b--;
           break;
    case5:
    case6:
    case7:
           printf("Function5_6_7");
           break;
}
```

02 | 分支程序-多分支程序-逻辑分解法

示例流程图



02 | 分支程序-多分支程序-逻辑分解法

```
DATA SEGMENT
    A DW 12H
    B DW ?
    MESS DB 'Function5_6_7','$'
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AX, A
        CWD
        MOV CX, 8
        IDIV CX
        CMP DX, 0
        JZ CASE0 ;模为 0
        CMP DX, 1
        JZ CASE12 ;模为 1
        CMP DX, 2
        JZ CASE12 ;模为 2
        CMP DX, 3
        JZ CASE3 ;模为 3
```

为防止商溢出的除法错中断,必须使用 32 位被除数!

```
        CMP DX, 4
        JZ CASE4 ;模为 4
CASE567: MOV AH, 9
        LEA DX, MESS
        INT 21H
        JMP EXIT
CASE0: MOV B, 32D
        JMP EXIT
CASE12: MOV AX, A
        ADD AX, 43D
        MOV B, AX
        JMP EXIT
CASE3: MOV AX, A
        SHL AX, 1
        MOV B, AX
        JMP EXIT
CASE4: DEC B
EXIT: MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

02 | 分支程序-多分支程序-地址表法

构建地址表

- 设置地址表
 - 将各分支的入口地址 (**case0**、**case12**、**case3**、**case4**、**case567**) 存放于一段存储区域中
 - 每个地址占 2 个字节, 共 8 个地址
- 分支转入
 - 间址寄存器 BX 中设置为:
 $(a\%8)*2$
 - 指令 **JMP** TABLE [**BX**] 转入分支

TABLE	数据段	
	case0	00H 01H
	case12	02H 03H
	case12	04H 05H
	case3	06H 07H
	case4	08H 09H
	case567	0AH 0BH
	case567	0CH 0DH
	case567	0EH 0FH
	

02 | 分支程序-多分支程序-地址表法

示例程序

```
DATA SEGMENT
    TABLE DW CASE0,CASE12,CASE12, CASE3
            DW CASE4,CASE567,CASE567
            DW CASE567
    A DW 12H
    B DW ?
    MESS DB 'Function5_6_7','$'
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV BX, A
        AND BX, 7 ; A%8
        SHL BX, 1 ; (A%8)*2
        JMP TABLE[BX] ; 根据地址表跳转
```

地址表

求 A%8

```
CASE0: MOV B, 32D
        JMP EXIT
CASE12: ADD AX, 43D
        MOV B, AX
        JMP EXIT
CASE3: SHL AX, 1
        MOV B, AX
        JMP EXIT
CASE4: DEC B
        JMP EXIT
CASE567: LEA DX, MESS
        MOV AH, 9
        INT 21H
EXIT: MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

02 | 分支程序-多分支程序-转移表法

构建转移指令表

- 设置转移指令列表
 - 设置列表标号
 - 每条转移指令占用 3 个字节的
空间，利用 $(a\%8)$ 的结果寻址
每一条转移指令
- 分支转入
 - 间址寄存器 BX 中设置为：
 $(a\%8)*3$
 - 指令 **JMP BX** 转入分支
 - 每一个分支最后都加 **JMP**
EXIT，最后一个除外

```
JMPTB:  JMP NEAR PTR CASE0  
        JMP NEAR PTR CASE12  
        JMP NEAR PTR CASE12  
        JMP NEAR PTR CASE3  
        JMP NEAR PTR CASE4  
        JMP NEAR PTR CASE567  
        JMP NEAR PTR CASE567  
        JMP NEAR PTR CASE567
```

这里的强制类型是
为了使每条指令所
占的字节数相同！

02 | 分支程序-多分支程序-转移表法

```
DATA SEGMENT
    A DW 12H
    B DW ?
    MESS DB 'Function5_6_7','$'
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV BX, A
        AND BX, 7 ; A%8
        MOV CX, BX
        SHL BX, 1 ; (A%8)*2
        ADD BX, CX
        ADD BX, OFFSET JMPTB
        JMP BX
JMPTB: JMP NEAR PTR CASE0
        JMP NEAR PTR CASE12
        JMP NEAR PTR CASE12
        JMP NEAR PTR CASE3
```

```
        JMP NEAR PTR CASE4
        JMP NEAR PTR CASE567
        JMP NEAR PTR CASE567
        JMP NEAR PTR CASE567
CASE0: MOV B, 32D
        JMP EXIT
CASE12: ADD AX, 43D
        MOV B, AX
        JMP EXIT
CASE3: SHL AX, 1
        MOV B, AX
        JMP EXIT
CASE4: DEC B
        JMP EXIT
CASE567: LEA DX, MESS
        MOV AH, 9
        INT 21H
EXIT: MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

转移指令
地址表

能不能换成 `JMP JMPTB[BX]` ?

不能! 段内转移 `JMP` 指令后接偏移地址, 而 `JMPTB[BX]` 是存储器寻址方式, 默认段是数据段, 转移表在代码段

本节小结



- 熟悉掌握顺序结构、分支结构汇编语言程序的编写方法
- 掌握多分支结构的三种分支设计方法
 - 逻辑分解法
 - 地址表法（数据段中建立分支入口地址表）
 - 转移表法（程序段中建立转移分支指令表）

目录

- 01 80x86 汇编语言程序设计基础
- 02 汇编语言顺序、分支程序设计
- 03 汇编语言循环、串处理程序设计
- 04 子程序设计

03 | 上节回顾-分支结构程序设计

简答题

- 汇编语言如何实现多条件分支结构程序？
- 分支结构程序需要用到哪些指令？

03 | 循环程序-循环程序设计

循环程序的结构

- WHILE 结构
- UNTIL 结构

循环指令

- LOOP、LOOPZ/LOOPE、LOOPNZ/LOOPNE
- JCXZ

循环程序类型

- 计数控制的循环程序
- 条件控制的循环程序
- 计数+条件控制的循环程序

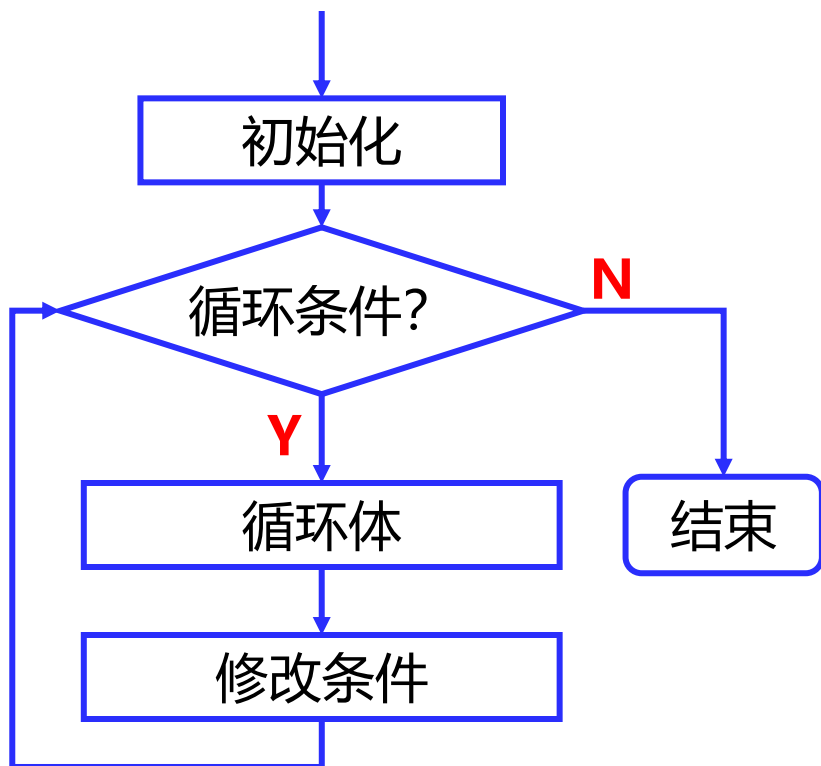
03 | 循环程序-循环程序的结构

循环程序的结构

● WHILE 结构

■ 先判断后执行

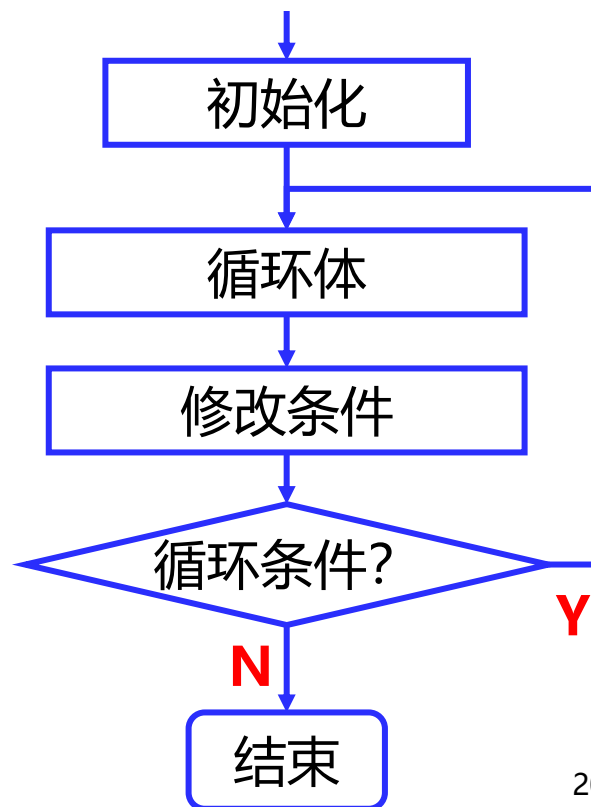
■ 循环体可以一遍也不执行



● UNTIL 结构

■ 先执行后判断

■ 循环体至少执行一遍



03 | 循环程序-循环程序的组成部分

循环程序的结构

● 初始化

- 循环的准备，主要有**指针**、**计数器**、**初值**等的初始设置

● 循环体

- **核心部分**，动态地执行功能相同的操作

● 修改条件

- 为下次循环体的执行做准备，如**指针**、**计数器**等的修改

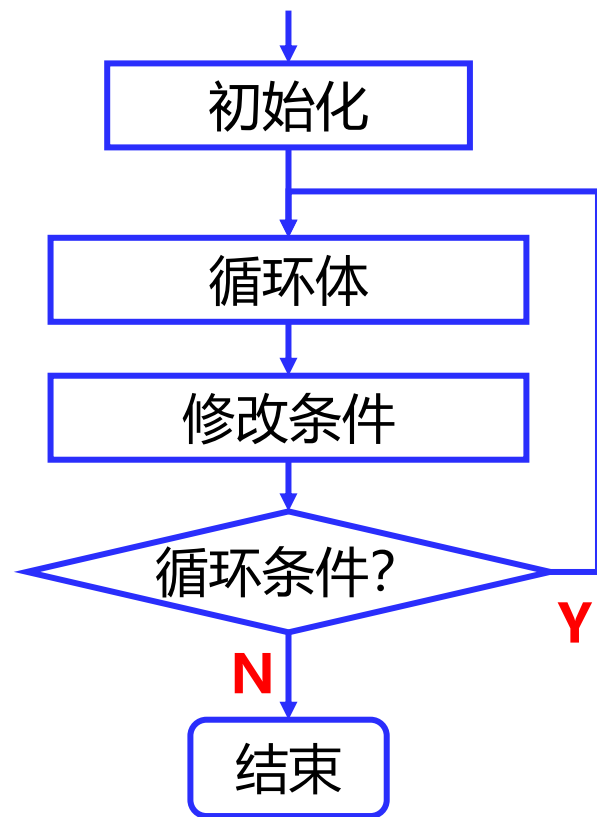
● 循环条件

- 决定是否继续循环

● 结束处理

- 完成对循环结果的分析、存储等操作

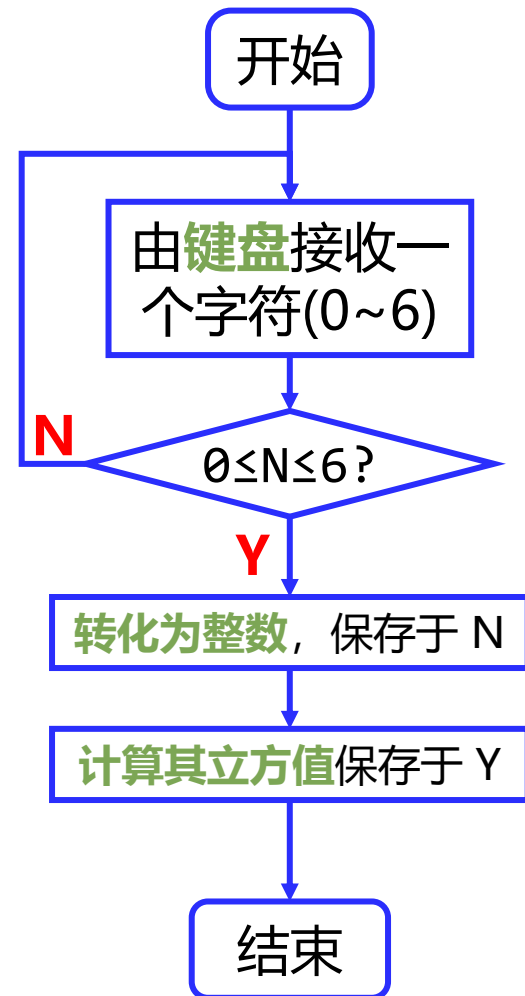
UNTIL 结构



03 | 循环程序-循环程序示例1

示例1：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序。 顺序结构示例2
数据 N 由键盘输入， N 、 Y 均为无符号字节数。 添加循环控制

- 由键盘输入数据 N
 - 对于输入的数据进行范围判别
 - 若 $0 \leq N \leq 6$ ，则直接求 N^3
 - 否则，重新输入
- 对于取值范围的判断
 - 分别判断 $N \geq 0$ 、 $N \leq 6$



计数循环示例1扩展

03 | 循环程序-循环程序示例1参考

示例1：设计一个求 $Y=N^3$ ($0 \leq N \leq 6$) 的程序。
数据 N 由键盘输入，N、Y 均为无符号字节数。添加循环控制

```
DATA SEGMENT
    N db ?
    Y db ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
INPUT: MOV AH, 7
        INT 21H
        CMP AL, 30H
        JB INPUT ; 小于30, 则重输
        CMP AL, 36H
```

仅输入不显示

```
        JA INPUT; 大于36, 则重输
        MOV DL, AL
        MOV AH, 02H
        INT 21H
        AND AL, 0FH
        MOV N, AL
        MUL AL
        MUL N
        MOV Y, AL
EXIT:  MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

仅显示符合
要求的字符

完成 N^3 操作

03 | 循环程序-循环指令LOOP

指令格式

- **LOOP** Label

该标号指向循环体的起始位置

指令执行过程

- $(CX) - 1 \rightarrow CX$
- 判断 CX 是否为 0
 - 当 $CX \neq 0$, 则转移至标号位置, 继续循环
 - 当 $CX = 0$, 则顺序向下执行程序, 退出循环

注意

- **LOOP** 指令用于构成循环次数已知的循环程序
 - 初始化部分, 应预先设置循环次数于 CX 寄存器中

03 | 循环程序-循环指令 LOOPZ/LOOPE

指令格式

- **LOOPZ/LOOPE** Label

LOOPZ/LOOPE NEXT
的等价指令

```
NEXT: .....  
.....  
JNZ EXIT  
DEC CX  
JNZ NEXT  
EXIT: .....
```

指令执行过程

- $(CX) - 1 \rightarrow CX$
- 判断 CX 是否为 0，再判断 ZF 是否为 1
 - 当 $CX \neq 0$ 且 $ZF = 1$ ，则转移至标号位置，继续循环
 - 否则，按顺序向下执行程序，退出循环

注意

- **LOOPZ/LOOPE** 跳出循环的情况
 - ① $CX = 0$ ；② $ZF = 0$

03 | 循环程序-循环指令 LOOPNZ/LOOPNE

指令格式

- LOOPNZ/LOOPNE Label

LOOPNZ/LOOPNE NEXT
的等价指令

```
NEXT: .....  
.....  
JZ EXIT  
DEC CX  
JNZ NEXT  
EXIT: .....
```

指令执行过程

- $(CX) - 1 \rightarrow CX$
- 判断 CX 是否为 0，再判断 ZF 是否为 0
 - 当 $CX \neq 0$ 且 $ZF = 0$ ，则转移至标号位置，继续循环
 - 否则，按顺序向下执行程序，退出循环

注意

- LOOPNZ/LOOPNE 跳出循环的情况
 - ① $CX = 0$ ；② $ZF = 1$

循环控制指令 **LOOPNZ/LOOPNE** 控制循环继续执行的条件是 ()

- ☐ A $CX \neq 0$ 且 $ZF = 1$
- ☒ B $CX \neq 0$ 且 $ZF = 0$
- ☐ C $CX = 0$ 或 $ZF = 1$
- ☐ D $CX = 0$ 或 $ZF = 0$

03 | 循环程序-循环指令JCXZ

指令格式

- JCXZ Label

判断条件

- 若 $CX=0$ ，则转移至标号位置执行程序

用途

- 循环体开始执行之前，构成先判断后循环的结构
- 循环结束之后，**判断循环结束的条件**，以避免结束处理中的运算错误
 - 例如 **条件循环示例2**
- 该转移属于段内短转移

03 | 循环程序-循环程序示例2

示例2：查找 STRING 字符串 (长度为 100 个字符) 中是否存在字符“0”，若存在，则转向 NEXT 标号执行程序

- 初始化部分

- 循环计数器 CX

```
MOV CX, 100
```

- 字符串指针 BX

```
LEA BX, STRING
```

- 循环体

```
CMP BYTE PTR [BX], '0'
```

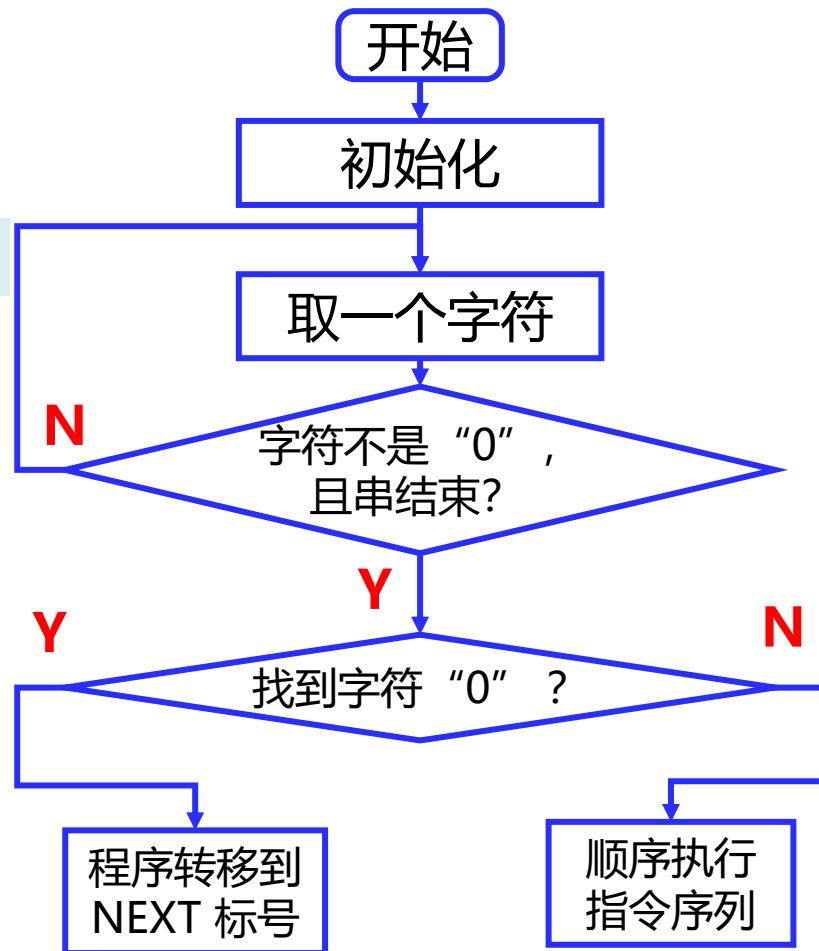
- 比较当前字符

- LOOPNZ 的使用

- 用在影响标志位的指令之后
- “修改”部分应写在循环体之前

- 循环退出的判断

- 应判断 ZF 状态，而非 CX 的值



03 | 循环程序-循环程序示例2参考

示例2：查找 STRING 字符串 (长度为 100 个字符) 中是否存在字符“0”，若存在，则转向 NEXT 标号执行程序

- 初始化部分

- 循环计数器 CX

```
MOV CX, 100
```

- 字符串指针 BX

```
LEA BX, STRING
```

- 循环体

```
CMP BYTE PTR [BX], '0'
```

- 比较当前字符

- **LOOPNZ** 的使用

- 用在影响标志位的指令之后
- “修改”部分应写在循环体之前

- 循环退出的判断

- 应判断 **ZF 状态**，而非 CX 的值

```
LEA BX, STRING
MOV CX, 100
DEC BX
AGAIN: INC BX
      CMP STRING[BX], '0'
      LOOPNZ AGAIN
      JZ NEXT
.....
```

03 | 循环程序-循环程序类型

循环程序的类型

- **计数**控制的循环程序（**循环次数已知**）
 - 使用**条件转移指令**控制循环，**正计数/倒计数**均可
 - 使用 **LOOP** 指令控制循环，只能**倒计数**
- **条件**控制的循环程序（**循环次数未知**）
 - 使用**条件转移指令**控制循环
- **计数+条件**控制的循环程序
 - 循环控制方式：**LOOPZ/LOOPNZ** 等指令控制
 - ◆ 使用转移指令，需要 **2 次判断**
 - 典型的程序类型：**串搜索、数组处理**等程序

03 | 循环程序-计数控制循环示例1

示例1：编写程序段，计算 $1+2+\dots+1000$ 之和，并把结果存入 AX 中

- 循环体：`ADD AX, BX`

BX 初始值为 1
终止值为 1000

`INC BX`

- 循环控制：`LOOP <标号>`

隐含操作： $CX-1 \rightarrow CX$;
CX 初值为 1000, 终止值为 0

- 因此，BX 和 CX 可以合二为一，代码如下：

```
MOV CX, 1000
XOR AX, AX      ;AX 清 0
again: ADD AX, CX
      LOOP again
```

03 | 循环程序-计数控制循环示例1扩展

示例1：编写程序段，计算 $1+2+\dots+N$ 之和，并把结果存入 AX 中，其中 N 为用户输入值 ($1 \leq N \leq 9$)

- 循环体1：输入参数控制，参考 循环结构示例1

```
INPUT: MOV AH, 7
        INT 21H
        CMP AL, 31H
        JB INPUT ;小于 31, 则重输
        CMP AL, 39H
        JA INPUT ;大于 39, 则重输
```

```
MOV AH, 0
MOV CX, AX
```

- 循环体2：累加 N 次，参考 计数循环示例1

```
        XOR AX, AX ;AX 清 0
again:  ADD AX, CX
        LOOP again
```

03 | 循环程序-计数控制循环示例2

示例2：编写程序，将 BUF 中的连续 100 个存储单元的内容清 0

- 循环初始化

- 循环计数器 CX

```
MOV CX, 100
```

- 数据指针 BX

```
LEA BX, BUF
```

- 循环体

- 将 BX 指向单元清 0

```
MOV BYTE PTR [BX], 0
```

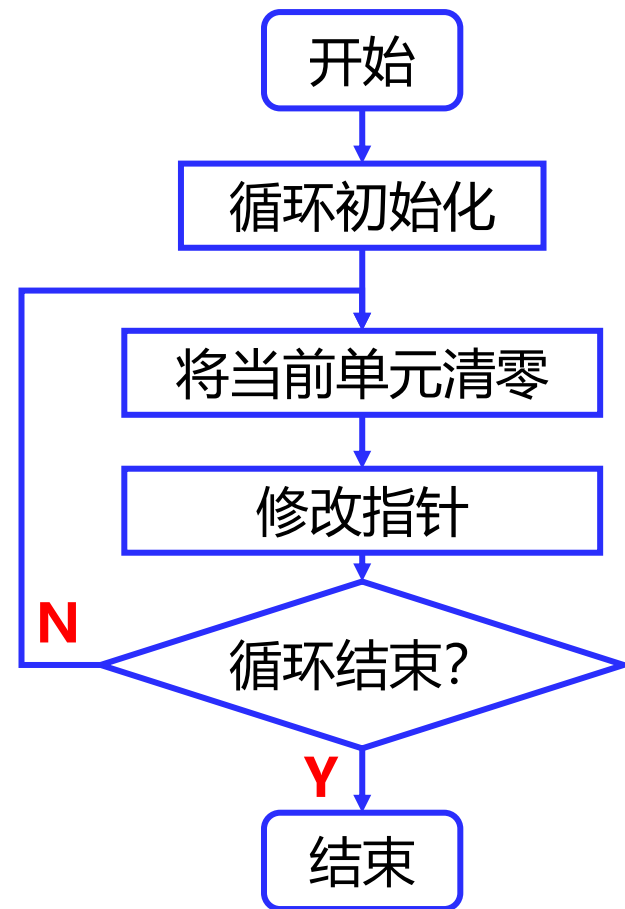
- 修改部分

- 数据指针增量

```
INC BX
```

- 循环控制

- 循环计数值是否为 0 `LOOP AGAIN`



03 | 循环程序-计数控制循环示例2参考

示例2：编写程序，将 BUF 中的连续 100 个存储单元的内容清 0

```
DATA SEGMENT
    BUF DB 100 DUP(15)
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
    MOV DS, AX
    MOV BX, 0
```

```
    MOV CX, 100
AGAIN: MOV BUF[BX], 0
    INC BX
    LOOP AGAIN
    MOV AX, 4C00H
    INT 21H
CODE ENDS
END START
```

- 可以使用串操作指令 **STOSB** 代替循环

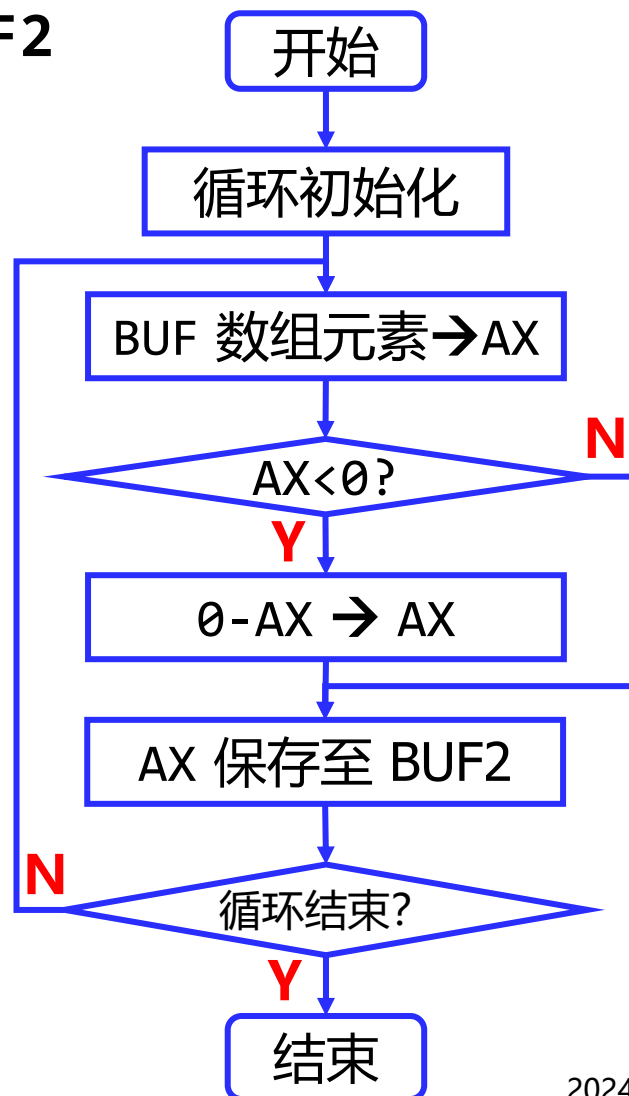
```
MOV ES, AX
CLD
```

```
LEA DI, BUF
XOR AL, AL
REP STOSB
```

03 | 循环程序-计数控制循环示例3

示例3：已知 BUF 数组中包含 100 个字数据，取 BUF 数组中每个元素的绝对值形成正数数组 BUF2

- 程序的主要任务：
 - 对 100 个字数据求绝对值
 - 应使用循环次数已知的循环结构程序
 - ◆ 循环次数为 100
- 循环体内部：
 - 求数据的绝对值：单分支结构程序



03 | 循环程序-计数控制循环示例3参考

示例3：已知 BUF 数组中包含 100 个字数据，取 BUF 数组中每个元素的绝对值形成正数数组 BUF2

```
DATA SEGMENT
    BUF DW 50 DUP(1), 50 DUP(-1)
    BUF2 DW 100 DUP(?)
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        LEA SI, BUF
        LEA DI, BUF2
        MOV CX, 100
```

源、目的数据指针

循环计数值

```
AGAIN: MOV AX, [SI]
        TEST AX, 8000H
        JZ NEXT
        NEG AX
NEXT: MOV [DI], AX
        ADD SI, 2
        ADD DI, 2
        LOOP AGAIN
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

求取每一个
数据的绝对值

修改指针

判断是否小于 0 也可以用右边两条指令

```
CMP AX, 0
JGE NEXT
```

AX 表示有符号数，要实现当 $AX < 0$ ，则跳转到 NEXT 标号位置，下列指令中能实现上述功能的是（ ）

A

```
TEST AX, 8000H
JZ NEXT
```

B

```
TEST AX, 8000H
JNZ NEXT
```

C

```
TEST AX, 8000H
JNS NEXT
```

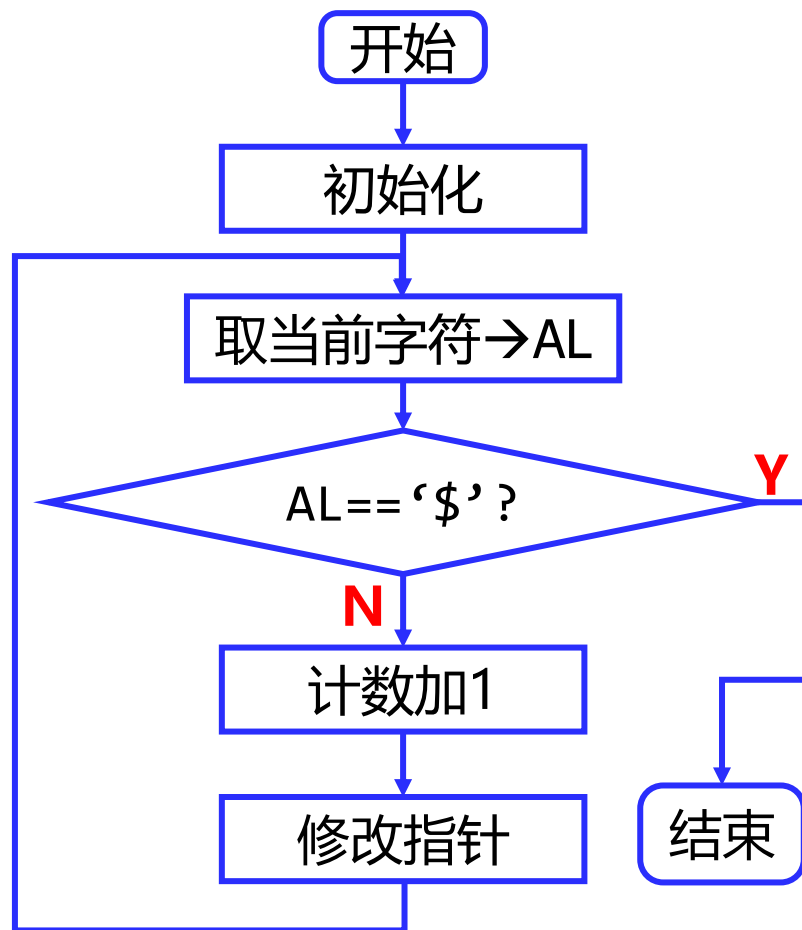
D

```
CMP AX, 0
JGE NEXT
```

03 | 循环程序-条件控制循环示例1

示例1：编写程序，求以 '\$' 符结束的字符串 BUF 的长度，并将结果保存于 CNT 单元

- 程序的主要任务
 - 统计字符串 BUF 的长度
 - 字符串以 "\$" 符结束
 - ◆ 逐个字符判别，统计
- 采用条件循环结构程序
 - 循环次数未知
 - 以查找到 "\$" 符为准



03 | 循环程序-条件控制循环示例1参考

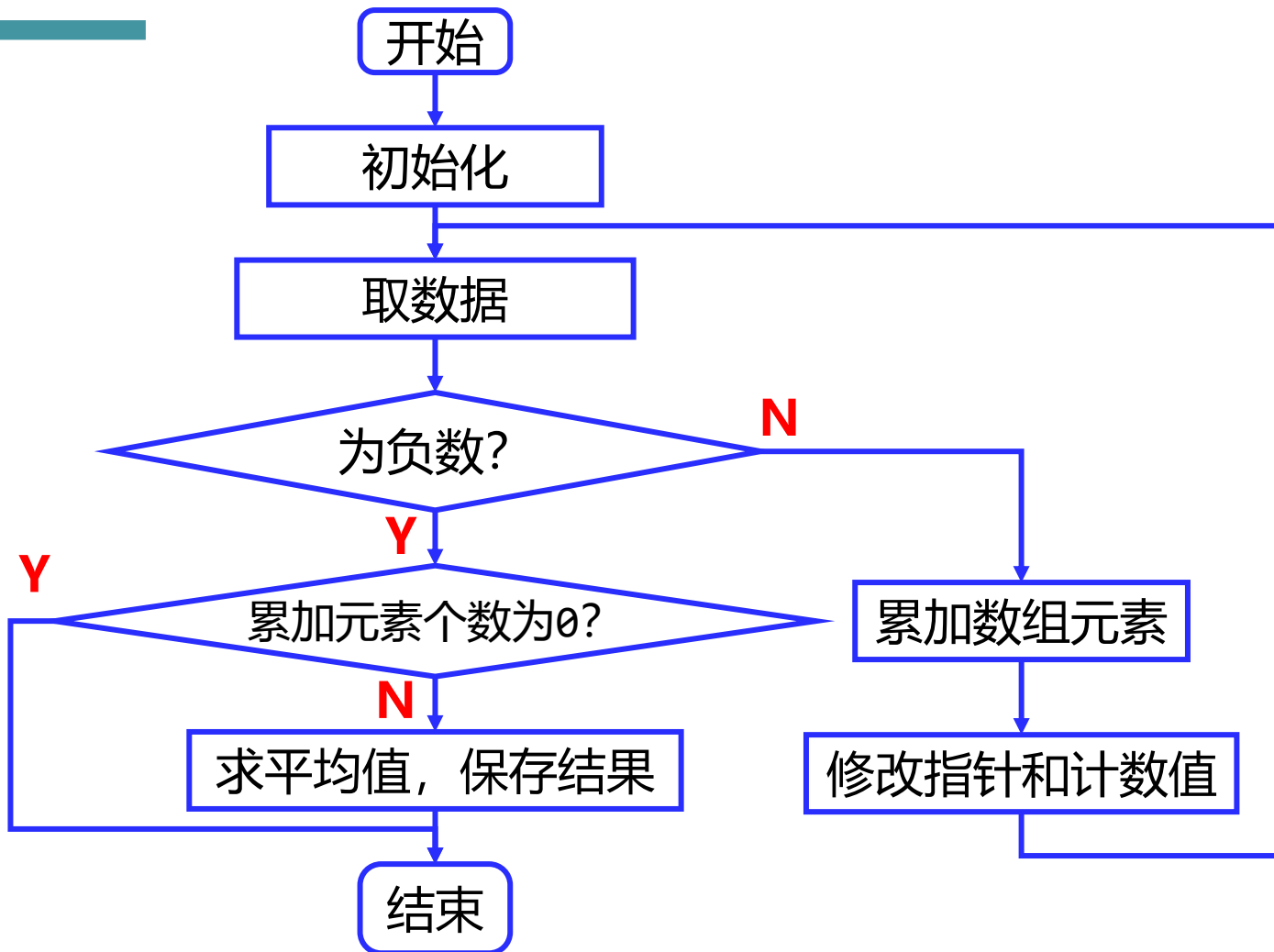
示例1：编写程序，求以 '\$' 符结束的字符串 BUF 的长度，并将结果保存于 CNT 单元

```
DATA SEGMENT
    BUF DB 'HOW DO YOU DO?$'
    CNT DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        LEA SI, BUF
        MOV CX, 0
```

```
AGAIN: MOV AL, [SI]
        CMP AL, '$'
        JZ EXIT  循环控制
        INC CX  长度计数值
        INC SI  修改数据指针
        JMP AGAIN
EXIT: MOV CNT, CX
        MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

03 | 循环程序-条件控制循环示例2

示例2：编写程序，把数组 score 的平均值 (取整) 存入字变量 average 中，数组以负数为结束标志



03 | 循环程序-条件控制循环示例2参考

示例2：编写程序，把数组 score 的平均值 (取整) 存入字变量 average 中，数组以负数为结束标志

```
DATA SEGMENT
    score DW 1,2,3,4,5,6,7,-1
    average DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        XOR AX, AX
        XOR DX, DX
        XOR CX, CX
        LEA SI, score
```

初始化

32位累加
和初始值

累加元素个数计数器

JCXZ指令

```
again: MOV BX, [SI]
        CMP BX, 0
        JL over
        ADD AX, BX
        ADC DX, 0
        INC CX
        ADD SI, 2
        JMP again
```

循环
控制

求累加和

累加次数计数值

修改指针

```
over: JCXZ exit
        DIV CX
        MOV average, AX
exit: MOV AX, 4C00H
        INT 21H
```

结束
处理

```
CODE ENDS
END START
```


03 | 循环程序-计数+条件控制循环示例1

示例1：编写程序，比较两个字符串 STR1 和 STR2 是否相同，若相同，则显示“Match”，否则显示“No Match”

- 数据段定义如下：

```
DATA SEGMENT
    STR1 DB "ABCD"
    LEN1 EQU $-STR1
    STR2 DB "ABCD"
    LEN2 EQU $-STR2
    MAT DB "Match$"
    NOMA DB "No Match$"
DATA ENDS
```

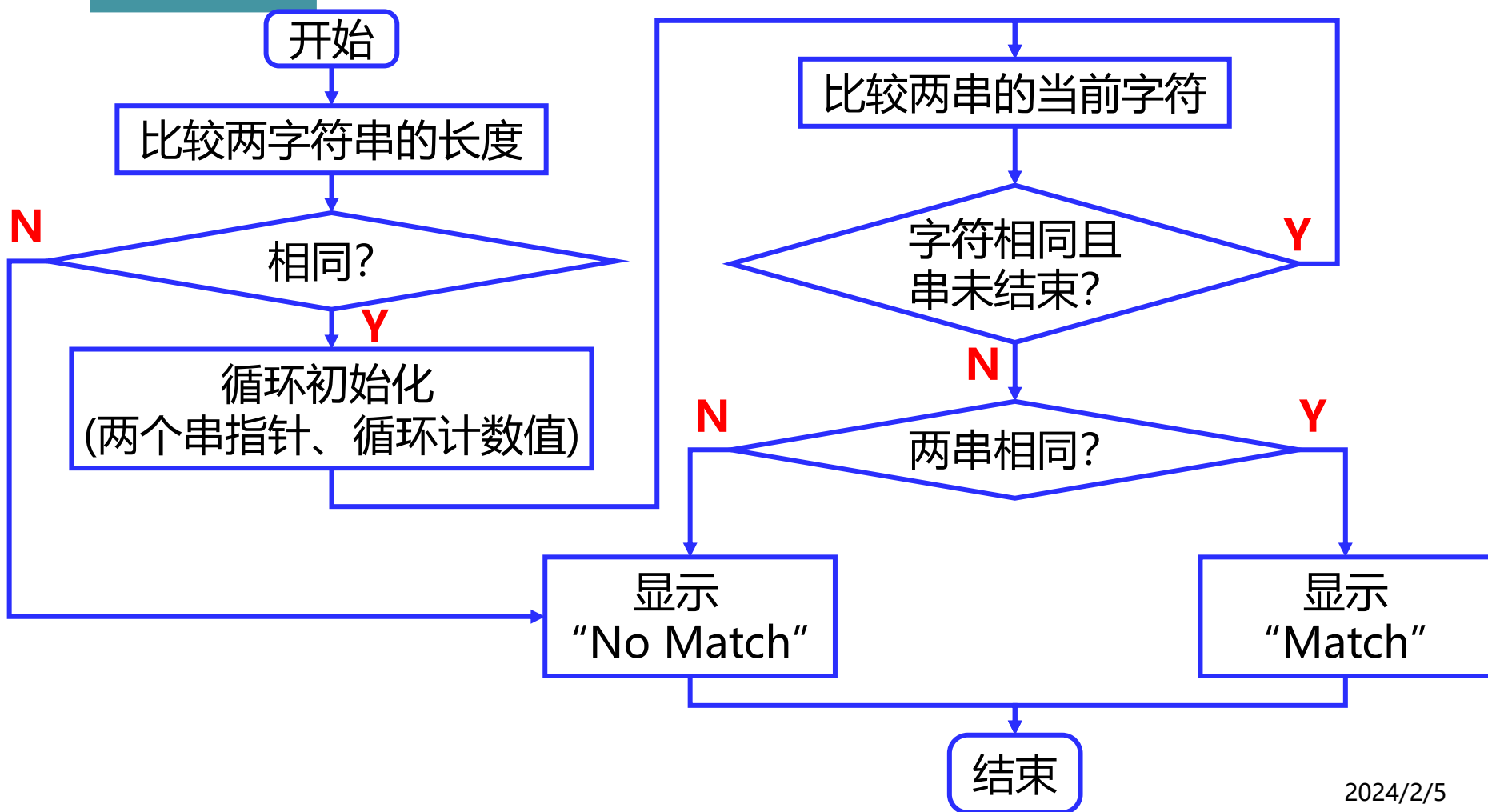
LEN1为常量，表示
STR1的长度

LEN2为常量，表示
STR2的长度

STR1	41
	42
	43
	44
STR2	41
	42
	43
	44
MAT	...
	...
NOMA	...
	...

03 | 循环程序-计数+条件控制循环示例1

示例1：编写程序，比较两个字符串 STR1 和 STR2 是否相同，若相同，则显示“Match”，否则显示“No Match”



03 | 循环程序-计数+条件控制循环示例1参考

示例1：编写程序，比较两个字符串 STR1 和 STR2 是否相同，若相同，则显示“Match”，否则显示“No Match”

```
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV AL, LEN1
        CMP AL, LEN2  比较两串长度相同?
        JNZ NOMATCH
        MOV SI, -1
        MOV CX, LEN1
AGAIN:  INC SI
        MOV AL, STR1[SI]
        CMP AL, STR2[SI]  比较两串元素相同?
        LOOPZ AGAIN
```

```
        JNZ NOMATCH
        LEA DX, MAT
        JMP DISPLAY
NOMATCH: LEA DX, NOMA
DISPLAY: MOV AH, 09H
        INT 21H
        MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

03 | 循环程序-计数+条件控制循环示例1改进

示例1改进：从键盘输入两个字符串，并比较是否相同，显示提示信息

- 输入字符串
 - 输入字符串的缓冲区设置

```
BUF1 DB 10, ?, 10 DUP(0)
BUF2 DB 10, ?, 10 DUP(0)
```
 - 以上缓冲区设置，实际可输入字符为 9 个
- 比较字符串的程序
 - 首先比较字符串的长度
 - ◆ BUF 缓冲区的第二个字节为输入串的实际长度
 - 再比较串中的每个字符

03 | 循环程序-循环嵌套

多重循环

- 循环指令使用 **CX** 或 **ECX** 进行计数，内循环计数必然会改变外循环计数值
- 对于**双重**循环，可将其中至少一个循环**改为转移指令**
- 对于**多重**循环，可在进入内层循环前**暂存 CX 的值**，退出内层循环前恢复 **CX** 的值
 - 将 CX 暂存至固定**内存单元**，需要保证该内存单元不被程序修改
 - 将 CX 暂存至通用**寄存器**，该寄存器不能被程序其他部分使用
 - 将 CX 暂存至**堆栈**，注意堆栈平衡

03 | 循环程序-循环嵌套示例

示例：打印一个 5 行的倒三角形，第 1 行 5 个 '*'，第 2 行 4 个，依次类推

- 循环初始化
 - 外层循环计数器 CX，内层循环计数值为 CX
- 循环体
 - 外层循环控制打印的行数并换行
 - 内层循环控制打印的 '*' 的个数
- 循环控制
 - 循环次数控制循环

```
CODE SEGMENT
START: MOV CX, 5
OUTLP: PUSH CX           ;可换成 MOV BX, CX
        MOV DL, '*'
        MOV AH, 2        ;打印*, CX次
        INT 21H
        LOOP INLP
        MOV DL, 10       ;显示换行符
        INT 21H
        POP CX           ;可换成 MOV CX, BX
        LOOP OUTLP
        MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

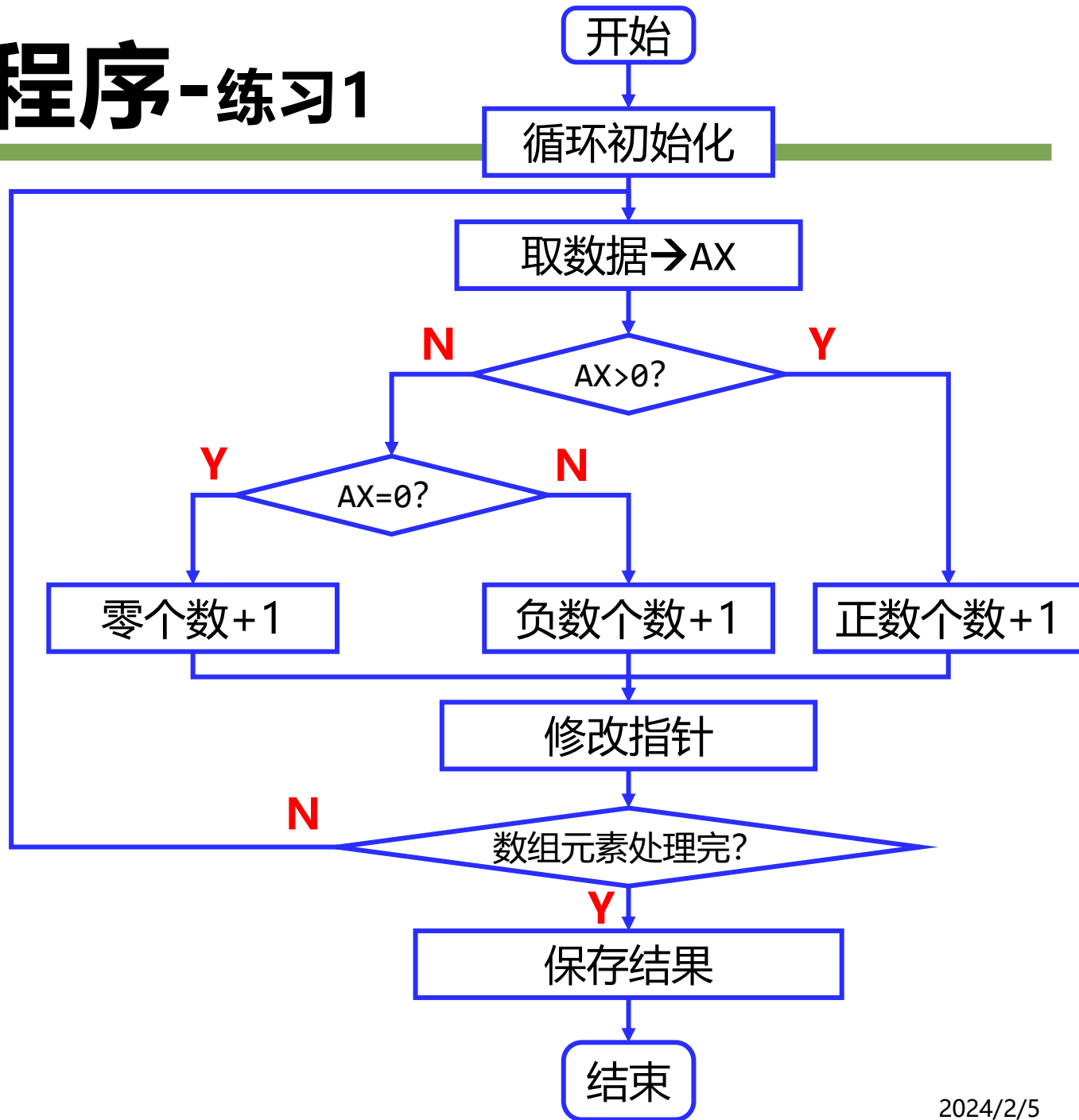
03 | 循环程序-练习1

练习1：分类统计数组 data 中正数、负数和零的个数，结果保存至字节变量 Pcount (正)、Ncount (负)、Zcount (零) 中，数组元素的个数保存于其第一个字数据中

- 循环初始化
 - 数组指针 SI、循环计数器 CX
 - 三个统计计数器 AX、BX、DX
- 循环体
 - 分支结构：判断每一个数组元素的性质
- 修改部分
 - 修改数据指针；修改循环计数值（包含在 LOOP 指令中）
- 循环控制
 - 循环次数控制循环
- 结束处理
 - 保存正负零的计数值

03 | 循环程序-练习1

练习1流程图



03 | 循环程序-练习1参考

练习1：分类统计数组 data 中正数、负数和零的个数，数组元素的个数保存于其第一个字数据中

初始化

```
.....  
XOR AX, AX  
XOR BX, BX  
XOR DX, DX  
MOV CX, data  
JCXZ save  
LEA SI, data+2  
again: CMP word ptr [SI], 0  
       JL lower  
       JE equal
```

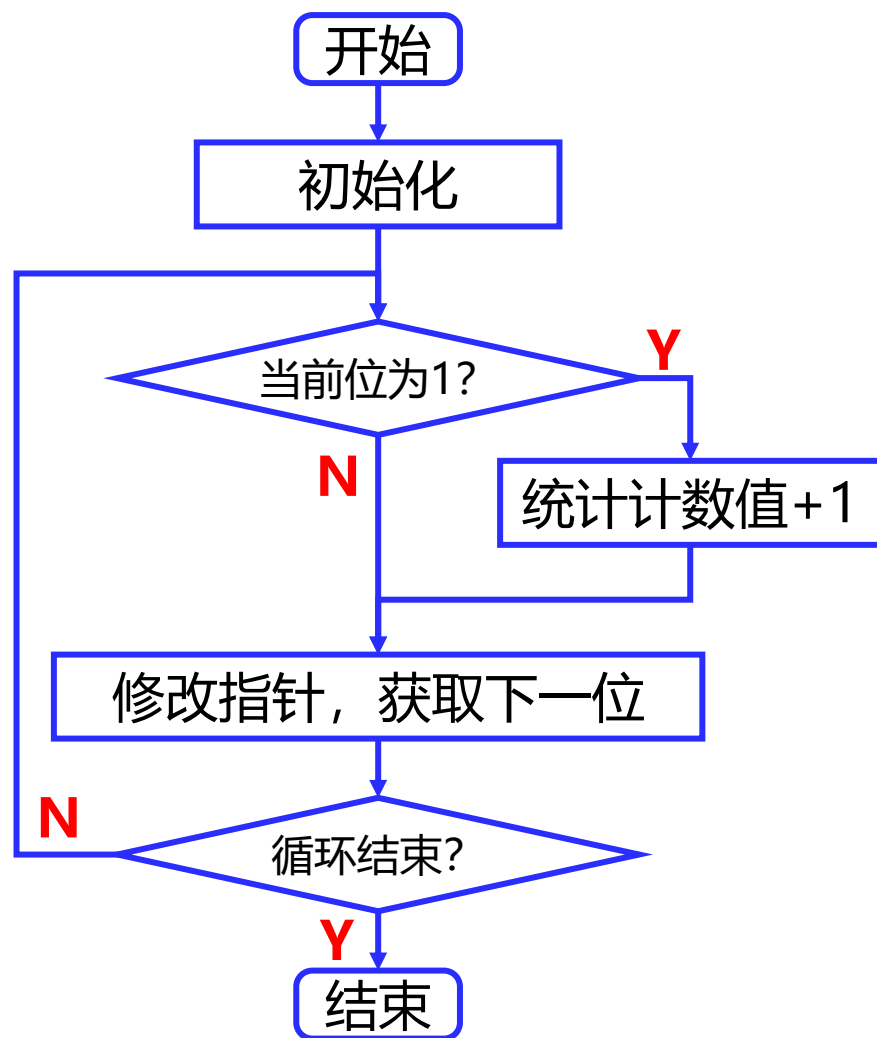
```
       INC AX  
       JMP loop1  
lower: INC BX  
       JMP loop1  
equal: INC DX  
loop1: ADD SI, 2 ;修改指针  
       LOOP again ;循环控制  
save: MOV Pcount, AX  
      MOV Ncount, BX  
      MOV Zcount, DX  
.....
```

结束
处理

03 | 循环程序-练习2

练习2：统计字变量 BUF 中所包含二进制位“1”的个数，将结果保存于字节变量 COUNT 中

- 逐位判断字变量 BUF 中数据
 - 循环次数为 16 次
 - 循环体中判断对应位的 0、1 状态
 - ◆ 方法1：TEST 指令测试
 - ◆ 方法2：移位指令逐位移出



03 | 循环程序-练习2解法1-TEST指令

练习2：统计字变量 BUF 中所包含二进制位“1”的个数，将结果保存于字节变量 COUNT 中

```
DATA SEGMENT
    BUF DW 1234H
    COUNT DB ?
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        MOV BX, BUF
        MOV CX, 16 ;循环计数值
        XOR DL, DL ;统计计数值
```

```
        MOV AX, 0001H
AGAIN:  TEST BX, AX
        JZ NEXT
        INC DL
NEXT:   SHL AX, 1
        LOOP AGAIN
        MOV COUNT, DL
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

测试数据初值

测试当前位

统计结果+1

修改测试数据

03 | 循环程序-练习2解法2-ROL指令

练习2：统计字变量 BUF 中所包含二进制位“1”的个数，将结果保存于字节变量 COUNT 中

```
DATA SEGMENT
    BUF DW 1234H
    COUNT DB ?
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        MOV BX, BUF
        MOV CX, 16 ;循环计数值
        XOR DL, DL ;统计计数值
```

```
AGAIN: ROL BX, 1
        JNC NEXT
        INC DL
NEXT: LOOP AGAIN
        MOV COUNT, DL
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

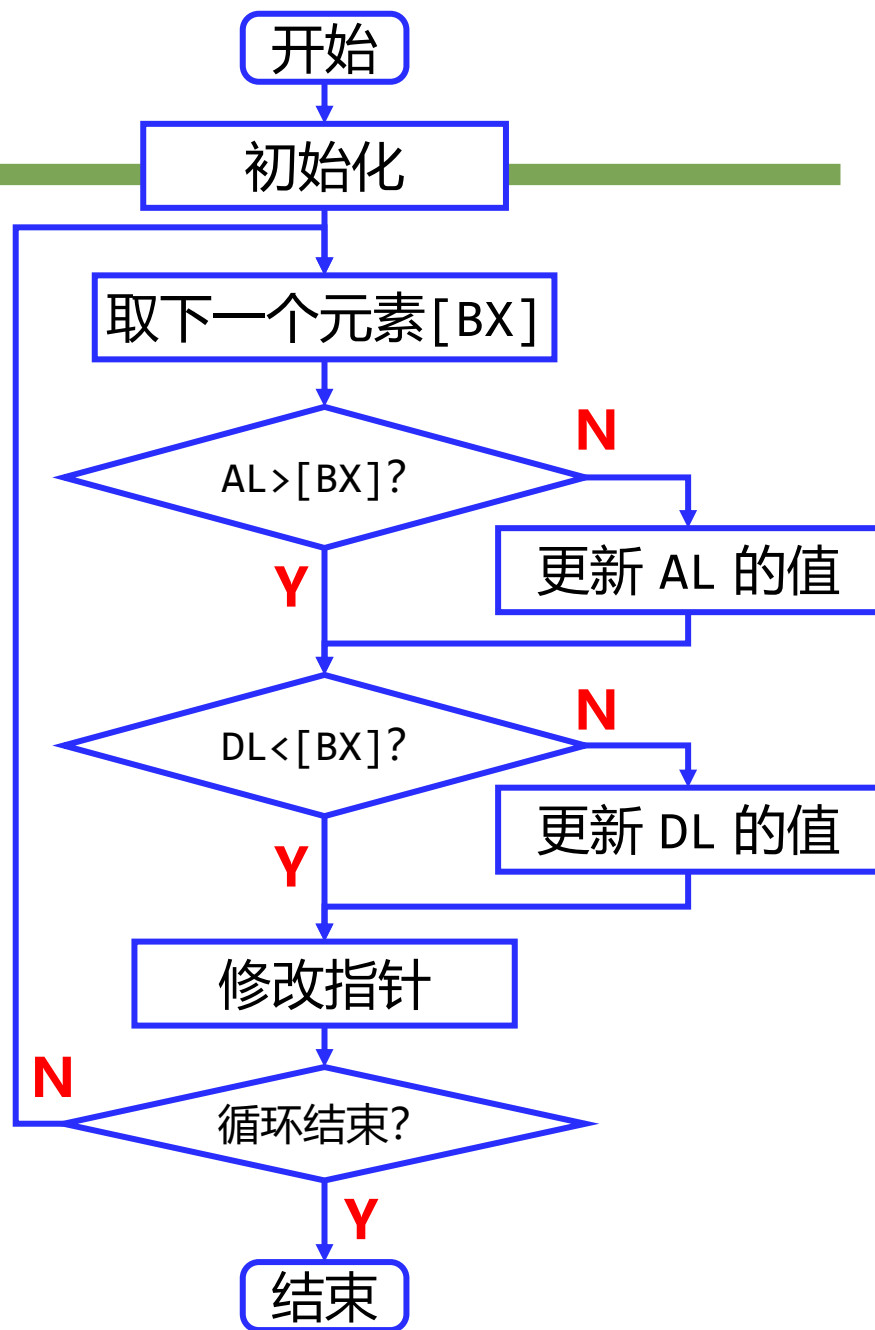
循环移出每一位

根据 CF 的值修改统计结果

03 | 循环程序-练习3

**练习3：在字节数组 ARRAY 中
求出最大值和最小值，结果存
放字节单元 MAX 和 MIN 中**

- 当数组元素确定时，该程序为**循环次数已知**的结构
- 关键问题：
 - **最大最小值的初始值**设置
 - **每次循环同时比较最大最小值**



03 | 循环程序-练习3参考

练习3：在字节数组 ARRAY 中
求出最大值和最小值，结果存
放字节单元 MAX 和 MIN 中

```
DATA SEGMENT
    ARRAY DB 1,2,34,4,-5,2,-7
    COUNT EQU $-ARRAY
    MAX DB ?
    MIN DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        LEA BX, ARRAY
        MOV AL, [BX]
        MOV DL, AL
        MOV CX, COUNT-1
```

取第一个元素作为
最大、最小值初值

从第二个元素开始比较

```
AGAIN: INC BX
        MOV AH, [BX]
        CMP AL, AH
        JGE NEXT
        MOV AL, AH
NEXT:   CMP DL, AH
        JLE OVER
        MOV DL, AH
OVER:  LOOP AGAIN
        MOV MAX, AL
        MOV MIN, DL
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

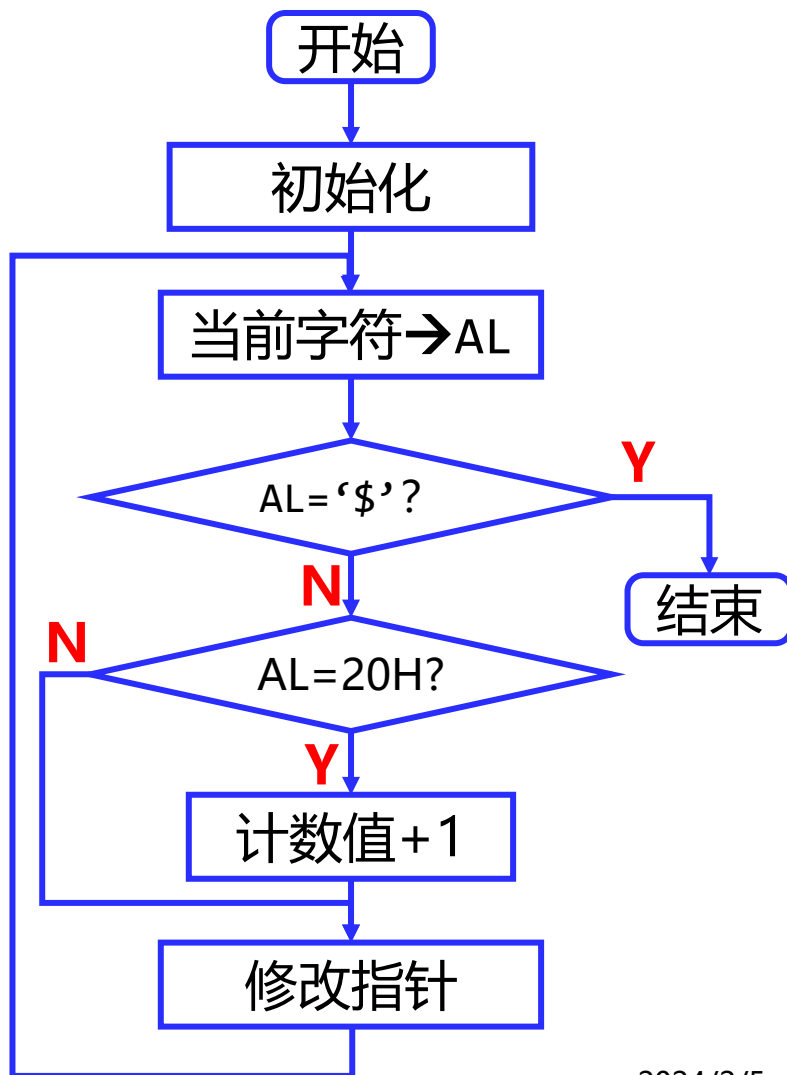
AL始终保存最大值

DL始终保存最小值

03 | 循环程序-练习4

练习4：统计一个以 “\$” 为结束符的字符串 String 中的空格个数

- 循环体中逐个字符检测
 - 若为 “\$” ，则**结束**
 - 若为 “ ” ，则**计数加 1**
- 循环次数不定的循环程序
 - 使用条件转移指令跳出循环
 - 循环体内是**分支结构程序**
 - ◆ 判断是否为空格，决定是否修改计数值



03 | 循环程序-练习4参考

练习4：统计一个以“\$”为结束符的字符串 String 中的空格个数

```
DATA SEGMENT
    STRING DB 'THIS IS A
STRING ! $'
    RESULT DW ?
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        LEA BX, STRING
        XOR CX, CX
AGAIN: MOV AL, [BX]
        CMP AL, '$'
        JZ EXIT
```

BX为数据指针

CX为计数值

判断是否结束

判断是否为空格

```
        CMP AL, 20H
        JNZ NEXT
        INC CX
NEXT: INC BX
        JMP AGAIN
EXIT: MOV RESULT, CX
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```


03 | 循环程序-练习5

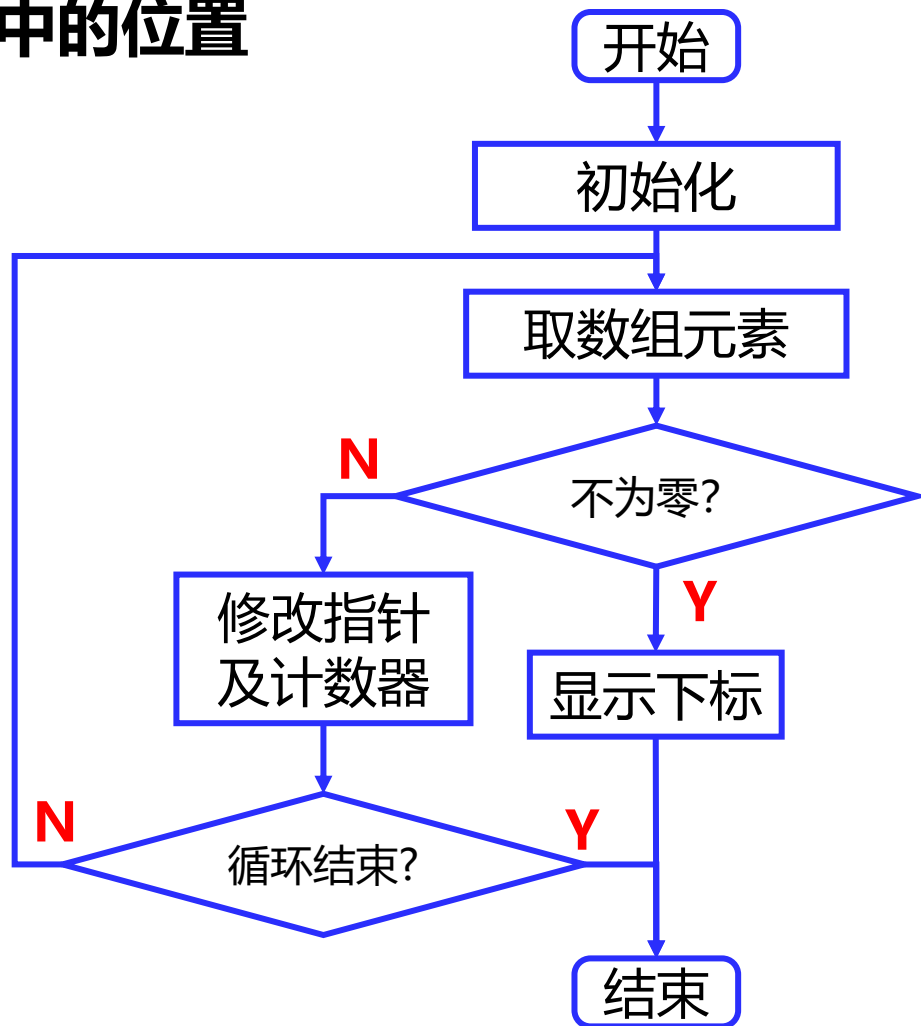
练习5：在包含 10 个字节数据的数组 array 中查找第一个非 0 数据，并显示其在数组中的位置

- 思路：

- 依次判断每个数组元素，**非零即退出**
- **循环次数**已知

- 使用 **LOOPZ** 指令

- 循环次数未到达，且数据仍然是 0 时，继续循环



03 | 循环程序-练习5参考

练习5：在包含 10 个字节数据的数组 array 中查找第一个非 0 数据，并显示其在数组中的位置

执行到该指令的两种情况：

1. 找到非 0 数据：CX 可能为 0 或其他值
2. 未找到非 0 数据：CX 为 0

```
DATA SEGMENT
    ARRAY DB 9 DUP(0),1
    MESS DB 'ALL ZERO!$'
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA
        MOV DS, AX
        MOV CX, 10
        MOV DI, -1
again: INC DI
        CMP array[DI], 0
        LOOPZ again
```

```
        JNZ NEXT
        MOV DX, offset MESS
        MOV AH, 9
        INT 21H
        JMP EXIT
NEXT: MOV DX, DI
        ADD DL, 30H
        MOV AH, 2
        INT 21H
EXIT: MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```

03 | 循环程序-练习6

练习6：在具有 10 个元素的字数组 ARRAY 中查找第一个 0，并显示其下标，若没有显示 n

判断触发退出循环的条件，是找到 0，还是数组结束

```
DATA SEGMENT
    ARRAY DW 15,0,22,43,78,35,44,68,90,76
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        MOV CX, 10
        MOV BX, -2
NEXT:  ADD BX, 2
        MOV AX, ARRAY[BX]
        CMP AX, 0
```

BX为字数据指针

```
        LOOPNZ NEXT
        JZ ZERO
        MOV DL, 'N'
        JMP EXIT
ZERO:  SHR BX, 1
        MOV DL, BL
        ADD DL, 30H
EXIT:  MOV AH, 2
        INT 21H
        MOV AX, 4C00H
        INT 21H
CODE ENDS
END START
```

输出单个字符的 2 号 DOS 功能调用的入口参数是（ ）

- ☐ A AL
- ☐ B AH
- ☒ C DL
- ☐ D DH

03 | 串处理程序-串操作

串操作

- 一般情况下，串操作处理的数据不只是一个字节或字，而是在内存中地址**连续的字节串或字串**
- 所有的串操作指令都用寄存器 (E)SI、(E)DI 操作数进行间接寻址
- 串操作时，地址的修改与方向标志 DF 有关

串操作指令

- 串操作指令包括
 - 串传送指令：MOVSB/W/D
 - 串比较指令：CMPSB/W/D
 - 串扫描指令：SCASB/W/D
 - 串装入指令：LODSB/W/D
 - 串存储指令：STOSB/W/D
 - 重复前缀：REP、REPE/Z、REPNE/NZ
 - 端口数据处理：INSB/W/D、OUTSB/W/D

03 | 串处理程序-串操作指令

串操作指令总结

指令	重复前缀	对标志位的影响	操作数	地址指针寄存器
MOVS	REP	无影响	目标	ES:DI
			源	DS:SI
CMPS	REPE/REPZ	有影响	目标	ES:DI
			源	DS:SI
SCAS	REPE/REPZ	有影响	目标	ES:DI
LODS	无	无影响	源	DS:SI
STOS	REP	无影响	目标	ES:DI

03 | 串处理程序-串操作-快速填充

快速填充

- **STOS** 指令结合 **REP** 指令，可实现一串内存单元的快速填充
 - 由 **ES:DI** 指定填充区域首地址
 - 由 **CX** 预先设置好填充数据大小

快速填充示例

```
DATA SEGMENT
    ORG 100H
    STR1 DB 'PLEASE ENTER A
    STRING. OK$'
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV ES, AX
```

```
LEA DI, BYTE PTR [100H]
CLD ; 设置填充方向
MOV CX, 25 ; 准备计数值
MOV AL, 'A' ; 准备填充值
REP STOSB ; 快速填充
MOV AX, 4C00H
INT 21H
CODE ENDS
END START
```

准备首地址

03 | 串处理程序-串操作-处理内存单元数据

处理内存单元数据并返回

- 使用 **LODS** 指令，结合**循环结构**，可以依次对串数据进行处理，并将处理结果返回原存储单元
 - 例子：将 [0700H] 指向的 9 个数据依次修改后，送回原存储单元

```
DATA SEGMENT
    ORG 700H
    STR1 DB '012345678'
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        CLD
```

```
        MOV SI, 0700H ; 准备首地址
        MOV CX, 9      ; 准备计数值
LI:     LODSB           ; 依次装入 AL
        INC AL         ; 处理数据
        MOV [SI-1], AL ; 送回内存单元
        LOOP LI
        MOV AX, 4C00H
        INT 21H
CODE ENDS
        END START
```


03 | 串处理程序-串操作-数据块的复制

数据块的复制

- **MOVS** 指令结合 **REP** 指令，可实现一串内存单元的数据块复制到另一片内存单元
 - 源数据块由 **DS:SI** 指定，目的数据块由 **ES:DI** 指定
 - 由 **CX** 预先设置需要复制的数据块长度 (字节数)
 - 复制方向由 **DF** 指定
- 源数据块与目的数据块可能出现部分重叠，也就是目的数据块起始地址位于源数据块范围内
- 当出现这种情况时，移动 (复制) 过程就**不能**简单地**从低地址向高地址**调整，而需要**从高地址向低地址**调整

03 | 串处理程序-串操作-数据块的复制示例

数据块的复制示例

- 将由 **DS:SI** 指定源数据块复制到由 **ES:DI** 指定的目的数据块

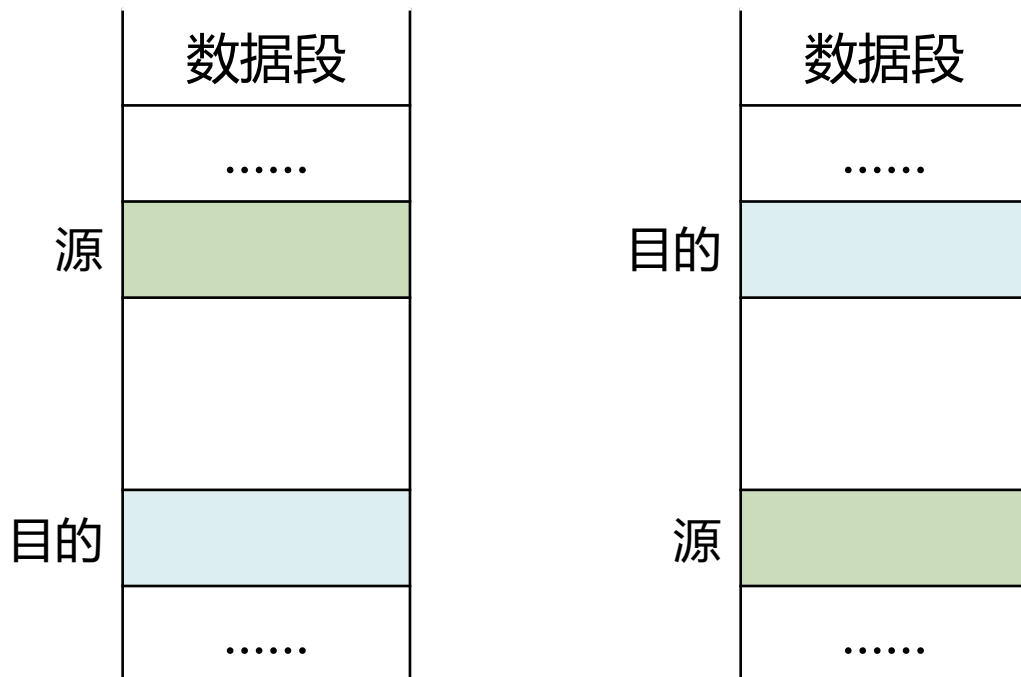
```
DATA SEGMENT
    ORG 2000H
    STR1 DB 64 DUP(12H)
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA, CS:CODE
START: MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
```

```
    MOV SI, 2000H ; 源地址
    MOV DI, 3000H ; 目的地址
    MOV CX, 64    ; 字符串长
    CLD           ; 复制方向
    REP MOVSB     ; 重复复制
    MOV AX, 4C00H
    INT 21H
CODE ENDS
    END START
```

03 | 串处理程序-串操作-数据块的复制方向

数据块的复制方向

- 对于不重叠的情况，无论源数据块位于低地址还是高地址，复制的方向可任意选择

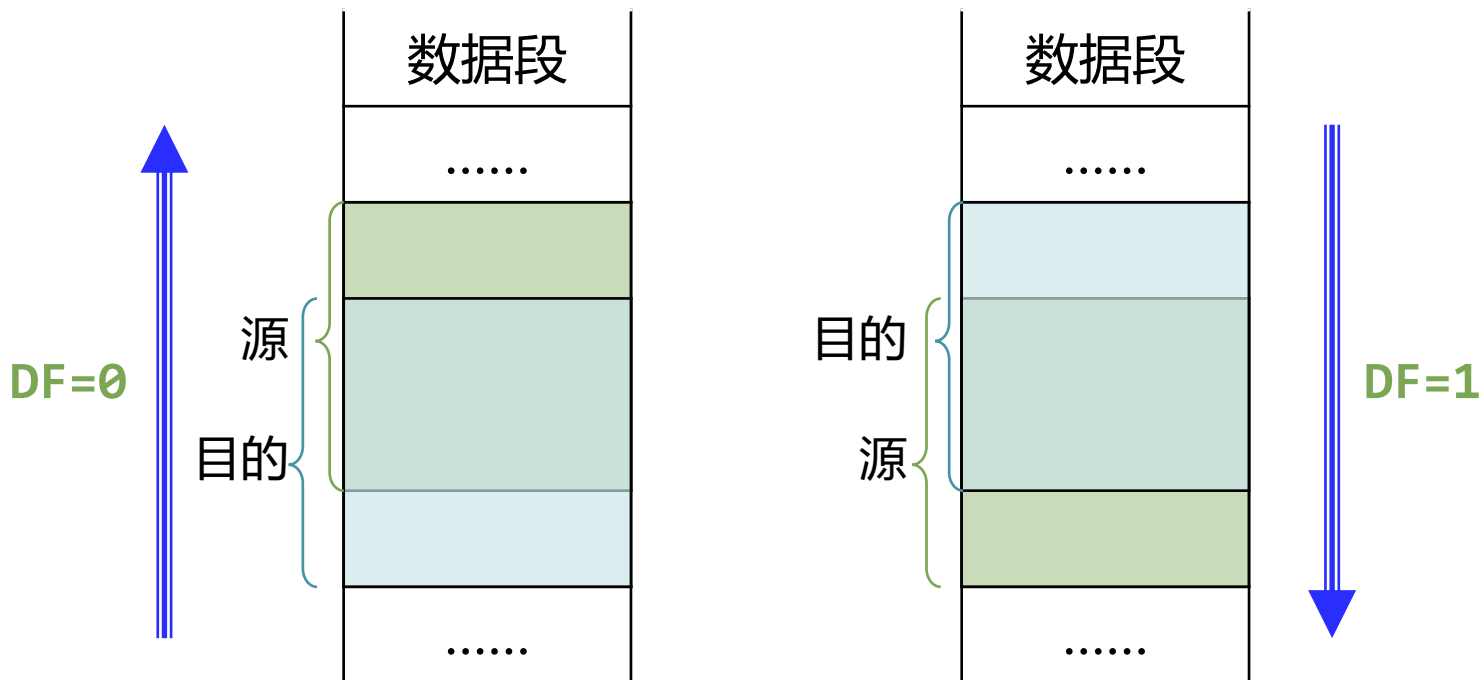


03 | 串处理程序-串操作-数据块的复制方向

数据块的复制方向

- 对于部分重叠的情况

- 当源数据块在高地址，目的数据块在低地址，则需要 $DF=0$
- 当源数据块在低地址，目的数据块在高地址，则需要 $DF=1$



本节小结

- 熟练掌握循环结构汇编语言程序的编写方法，会根据要求选择合适的循环结构
 - 计数控制的循环结构
 - 条件控制的循环结构
 - 计数+条件控制的循环结构
- 掌握串处理程序设计，理解它与循环结构的关系

目录

- 01 80x86 汇编语言程序设计基础
- 02 汇编语言顺序、分支程序设计
- 03 汇编语言循环、串处理程序设计
- 04 子程序设计**

04 | 上节回顾-循环结构程序设计

简答题

- 汇编语言程序中，以下循环结构可以选用哪些汇编指令？
 - 计数控制的循环结构
 - 条件控制的循环结构
 - 计数+条件控制的循环结构
 - 多层循环嵌套结构

04 | 子程序设计-子程序的概念

子程序 (Subroutine) 的概念

- 大型程序中，由**功能相对独立的程序段**构成的模块，相当于高级语言的**过程** (Procedure) 和**函数** (Function)

子程序的作用

- **简化主程序、增强可读性和可维护性，有利于代码重用**

例如：实验一题目3

- 从键盘接收两个不大于 5 的十进制数值，并以十进制数据形式显示其和
- 数据的输入代码：
 - 可定义为子程序，调用两次

```
MOV AH, 01H
INT 21H
SUB AL, 30H
MOV NUM, AL
MOV AH, 02H
MOV DL, 10
INT 21H
MOV AH, 02H
MOV DL, 13
INT 21H
```


04 | 子程序设计-子程序的定义

子程序定义伪指令

子程序名 PROC [NEAR/FAR]

..... ;子程序体

子程序名 ENDP

子程序名

- 合法标识符，首尾段名要一致，且在程序中唯一
- 子程序名作为操作数时，表示立即数，其值为子程序入口地址

说明

- 子程序的类型为可选项，16 位系统默认为近调用
NEAR 类型

04 | 子程序设计-关于子程序的定义

子程序定义的位置

- 必须在**代码段**中定义
- 常用的定义位置：
 - 主程序指令开始之前 (**START** 标号之前)
 - 主程序指令结束之后 (**<段名>** **ENDS**之前)

其他

- 子程序允许**嵌套** (子程序内包含有子程序的调用) 定义
- 要养成书写子程序**说明信息**的好习惯
 - **功能描述、入出口参数、所用寄存器、调用注意事项、编写人员及日期等**

04 | 子程序设计-子程序调用与返回

子程序的调用与返回

- 子程序的调用与返回是一个 CPU 执行程序的变化过程
 - 程序执行的转向由 CS、IP 寄存器中值的改变实现
 - 与转移指令的区别
 - ◆ 子程序执行完毕后，需要返回主程序

子程序的调用类型

- 近调用 (NEAR)：主程序与子程序在**同一逻辑段内**，程序转移只需改变 IP 寄存器的值
- 远调用 (FAR)：主程序与子程序在**不同逻辑段内**，程序转移需要同时改变 CS 和 IP 寄存器的值

04 | 子程序设计-子程序调用指令

子程序调用指令格式

- **CALL <OPRD>** ;OPRD 表示地址信息
- **OPRD**: REG16/MEM16/MEM32/LABEL

功能

- **OPRD** 为子程序标号时，为**段内/段间直接调用** (**最常用**)
 - 调用的类型由所调用**子程序定义的类型**决定
- **OPRD** 为 REG16/MEM16 时，为**段内间接调用**
 - 寄存器或存储单元中的值为**子程序的有效地址**
- **OPRD** 为 MEM32 时，为**段间间接调用**
 - 存储单元中的值为**子程序的段地址和有效地址**

注意

- 子程序调用与返回指令**不影响**标志位

04 | 子程序设计-子程序调用过程

子程序调用指令执行过程

1. 将当前 (CS、) IP 值 —— **CALL 指令之后**的指令地址，入栈保护
 - 段内调用 —— 只保护 IP
 - 段间调用 —— 同时保护 CS 和 IP
2. 将**子程序的起始地址**赋值给 (CS、) IP

CALL 指令与 JMP 指令的区别

- CALL 指令与 JMP 指令的区别在于**是否保护当前地址**
- 使用 JMP 指令模拟 CALL 指令：

CALL DELAY



PUSH IP
JMP DELAY

只是功能模拟，
指令为非法的！

假设为近调用

04 | 子程序设计-子程序返回指令

子程序返回 (Return) 指令格式

● RET

功能

- 指令执行：将堆栈中的内容出栈 → (CS、) IP
 - 可用 POP 和 JMP 指令模拟近返回
- 出栈数据数目由 RET 自行决定
 - 远调用 —— 出栈 CS、IP
 - 近调用 —— 出栈 IP

```
POP BX  
JMP BX
```

其他返回指令

- RET IMM ;子程序返回后, 再将 SP+IMM
 - 适用于使用堆栈传递参数的子程序, 可将入口参数移出堆栈

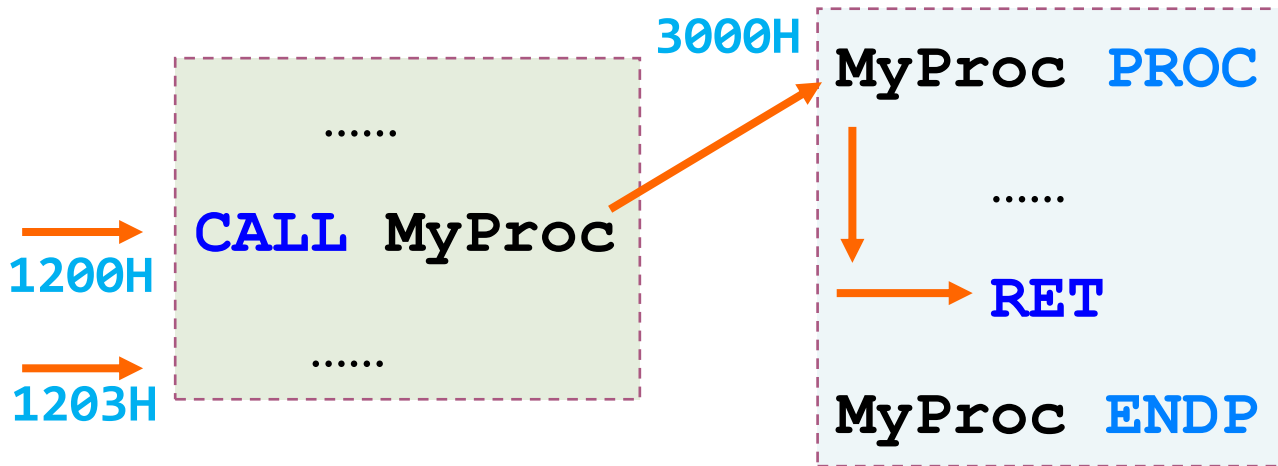
04 | 子程序设计-子程序调用与返回演示

主程序

子程序

IP

3000H



假设：本条 **CALL** 指令
占 3 个字节空间

子程序调用指令 **CALL** 执行后，一定**不会**影响的寄存器是（ ）


- ☐ A IP
- ☐ B SP
- ☐ C CS
- ☒ D AX

04 | 子程序设计-关于子程序的调用与返回

子程序执行过程

- 子程序的正确执行是依靠 **CALL**、**RET** 指令中的**入出栈操作**
 - 子程序调用时入栈返回地址
 - 子程序返回时出栈返回地址
- 子程序中必须保证 **RET** 指令的正常执行
- 例如：子程序 ROUTINE
 - 子程序中入栈**两个字**数据
 - 出栈**一个字**数据
 - 当执行 **RET** 返回时，**出栈的是 AX 的值**，无法正常返回
- 在某些病毒程序中，就是利用**修改返回地址**实现攻击的！

```
ROUTINE PROC
    PUSH AX
    PUSH BX
    .....
    POP BX
    RET
ROUTINE ENDP
```



04 | 子程序设计-子程序实现实验一题目3

实验一题目 3

```
INNUM PROC
    MOV AH, 01H
    INT 21H
    SUB AL, 30H
    MOV Y, AL
    MOV AH, 02H
    MOV DL, 10
    INT 21H
    MOV AH, 02H
    MOV DL, 13
    INT 21H
    RET
INNUM ENDP
```

```
LEA DX, STR1 ; 显示提示字符串
```

```
MOV AH, 09H
```

```
INT 21H
```

```
CALL INNUM ; 调用子程序
```

```
MOV AL, Y
```

```
MOV X, AL
```

```
LEA DX, STR1 ; 再次显示 提示字符串
```

```
MOV AH, 09H
```

```
INT 21H
```

```
CALL INNUM ; 调用子程序
```

```
MOV DL, X
```

```
ADD DL, Y ; 相加并显示结果
```

```
ADD DL, 30H
```

```
MOV AH, 02H
```

```
INT 21H
```

04 | 子程序设计-子程序的保护与恢复

子程序的保护与恢复

- 子程序的执行应**不影响**主程序中的数据
- 子程序调用和返回时，要对**通用寄存器**进行现场保护与恢复
 - 调用子程序时，**保护**所用的寄存器
 - 子程序返回时，**恢复**所用的寄存器
 - 注意：**出口参数除外**
- 如果使用堆栈进行保护，要注意**堆栈平衡**（入栈和出栈长度必须相等）

04 | 子程序设计-实验一题目3

```
DATA SEGMENT
    A DB ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
INNUM PROC
    MOV AH, 01H
    INT 21H
    SUB AL, 30H
    MOV A, AL
    MOV DL, 0AH
    MOV AH, 02H
    INT 21H
    MOV DL, 0DH
    MOV AH, 02H
    INT 21H
    RET
INNUM ENDP
```

```
START:
    MOV AX, DATA
    MOV DS, AX
    CALL INNUM
    MOV DL, A
    CALL INNUM
    ADD DL, A
    ADD DL, 30H
    MOV AH, 02H
    INT 21H
    MOV AX, 4C00H
    INT 21H
CODE ENDS
    END START
```

```
S:\>\test
2
3
@
S:\>\test
4
2
?
S:\>_
```

请同学们自行调试，解决问题！

04 | 子程序设计-参数传递

子程序的参数传递

- 在主程序与子程序之间的信息传递称为**参数传递**
 - **入口参数** (输入参数):
主程序传入子程序的, 需要子程序处理的数据
 - **出口参数** (输出参数):
子程序传回主程序的, 子程序处理的结果数据
- 参数的传递方法: **主程序与子程序事先约定好的**
 - 利用主程序与子程序都可使用的资源来进行传递
 - ◆ **寄存器**传递参数: 实现简单, 调用方便, 但只适用于**传递参数较少**的情形
 - ◆ **变量**传递参数: 不占用寄存器, **需要额外的存储单元**
 - ◆ **堆栈**传递参数: 不占用寄存器, 也无需额外的存储单元, 但较为复杂, 要注意**堆栈平衡**
- 并不是所有的子程序都要有参数, 根据需要设定

04 | 子程序设计-变量传递参数示例

示例：设计一个子程序，求指定数组 ARRAY 的最大值，最大值保存于 Max

- 数据段定义

- 该数据段中**变量是全局的**，子程序与主程序均可访问

- 子程序定义

- **子程序体**与主程序的编写相同
 - 循环结构程序，**循环次数**已知
- 在子程序的**开始之前保护现场**，**结束 (RET) 之前恢复现场**

- 也可使用寄存器传递参数，增强子程序的重用性！

```
DATA SEGMENT
    Count DW 5
    Array DW 8, -1, 32766,
0, 100
    Max DW ?
DATA ENDS
```

```
PUSH AX
PUSH BX
PUSH CX
```

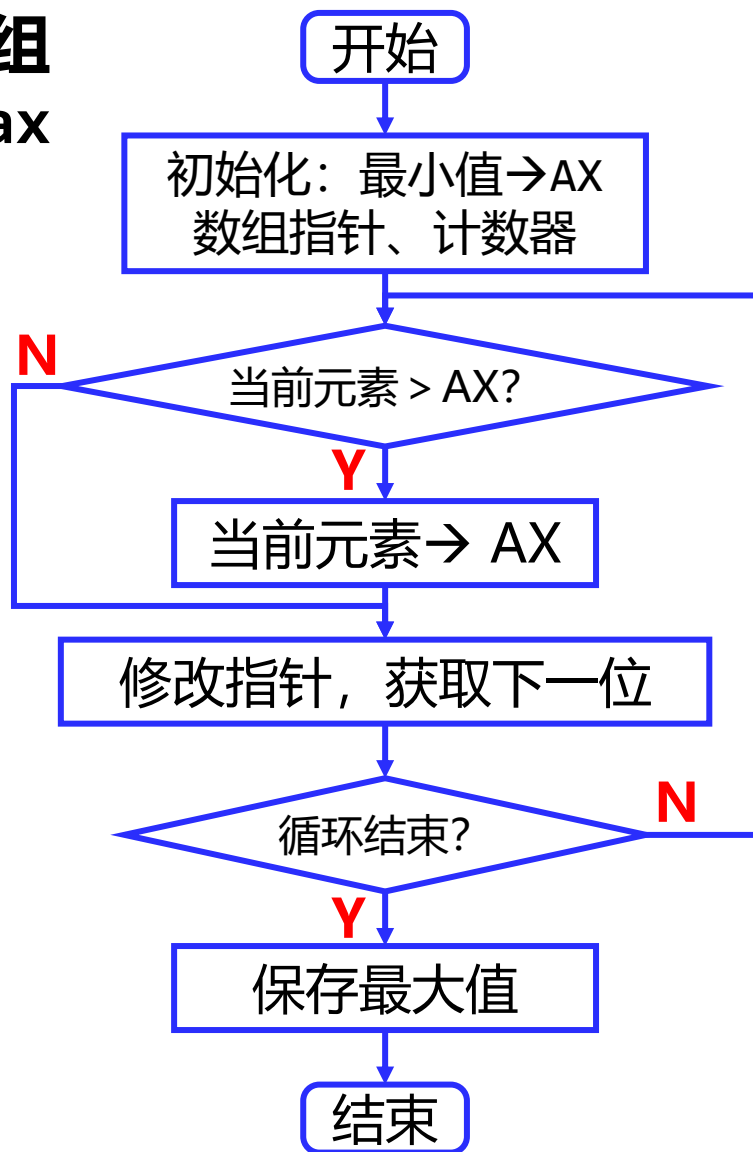
```
POP CX
POP BX
POP AX
```

04 | 子程序设计-变量传递参数示例-子程序

示例：设计一个子程序，求指定数组 ARRAY 的最大值，最大值保存于 Max

```
FindMax PROC
    MOV AX, 8000H
    MOV BX, 0
    MOV CX, Count
AGAIN:  CMP AX, Array[BX]
        JGE SKIP
        MOV AX, Array[BX]
SKIP:   ADD BX, 2
        LOOP AGAIN
        MOV Max, AX
        RET
FindMax ENDP
```

用到的通用寄存器
有：AX、BX、CX



04 | 子程序设计-变量传递参数示例-完整程序

示例：设计一个子程序，求指定数组 ARRAY 的最大值，最大值保存于 Max

```
DATA SEGMENT
    Count DW 5
    Array DW 8, -1, 32766, 0, 100
    Max DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        CALL FindMax
        MOV AX, 4C00H
        INT 21H
```

```
FindMax PROC
    PUSH AX
    PUSH BX
    PUSH CX
    MOV AX, 8000H
    MOV BX, 0
    MOV CX, Count
AGAIN:  CMP AX, Array[BX]
        JGE SKIP
        MOV AX, Array[BX]
SKIP:  ADD BX, 2
        LOOP AGAIN
        MOV Max, AX
    POP CX
    POP BX
    POP AX
    RET
FindMax ENDP
CODE ENDS
    END START
```

保护现场

恢复现场

04 | 子程序设计-寄存器传递参数示例

示例：设计一个子程序，求指定数组
ARRAY 的最大值，最大值保存于 Max

- 子程序体更改

- 入出口参数指定为寄存器
- 子程序体内不需要设置指针和计数值

- 需保护的寄存器

- SI、CX

- 主程序调用

- 设置入口参数
- 执行 CALL 指令
- 获取出口参数

```
PUSH SI  
PUSH CX
```

```
POP CX  
POP SI
```

```
LEA SI, Array  
MOV CX, Count  
CALL FindMax  
MOV Max, AX
```

```
FindMax PROC  
    MOV AX, 8000H  
MOV BX, 0  
MOV CX, Count  
AGAIN: CMP AX, Array[BX]  
        JGE SKIP  
        MOV AX, Array[BX]  
SKIP:  ADD BX, 2  
        LOOP AGAIN  
MOV Max, AX  
        RET  
FindMax ENDP
```

04 | 子程序设计-寄存器传递参数示例-完整程序

**示例：设计一个子程序，求指定数组
ARRAY 的最大值，最大值保存于 Max**


```
DATA SEGMENT
    Count DW 5
    Array DW 8, -1, 32766, 0, 100
    Max DW ?
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA
        MOV DS, AX
        LEA SI, Array
        MOV CX, Count
        CALL FindMax
        MOV Max, AX
        MOV AX, 4C00H
        INT 21H
```

```
FindMax PROC
    PUSH SI
    PUSH CX
    MOV AX, 8000H
AGAIN: CMP AX, [SI]
        JGE SKIP
        MOV AX, [SI]
SKIP:  ADD SI, 2
        LOOP AGAIN
    POP CX
    POP SI
    RET
FindMax ENDP
CODE ENDS
    END START
```

04 | 子程序设计-堆栈传递参数示例

示例：设计一个子程序，求指定数组
ARRAY 的最大值，最大值保存于 Max

- 需要堆栈保存的数据有
 - 入口参数
 - 子程序调用的返回地址
 - 子程序中使用的通用寄存器
 - 出口参数



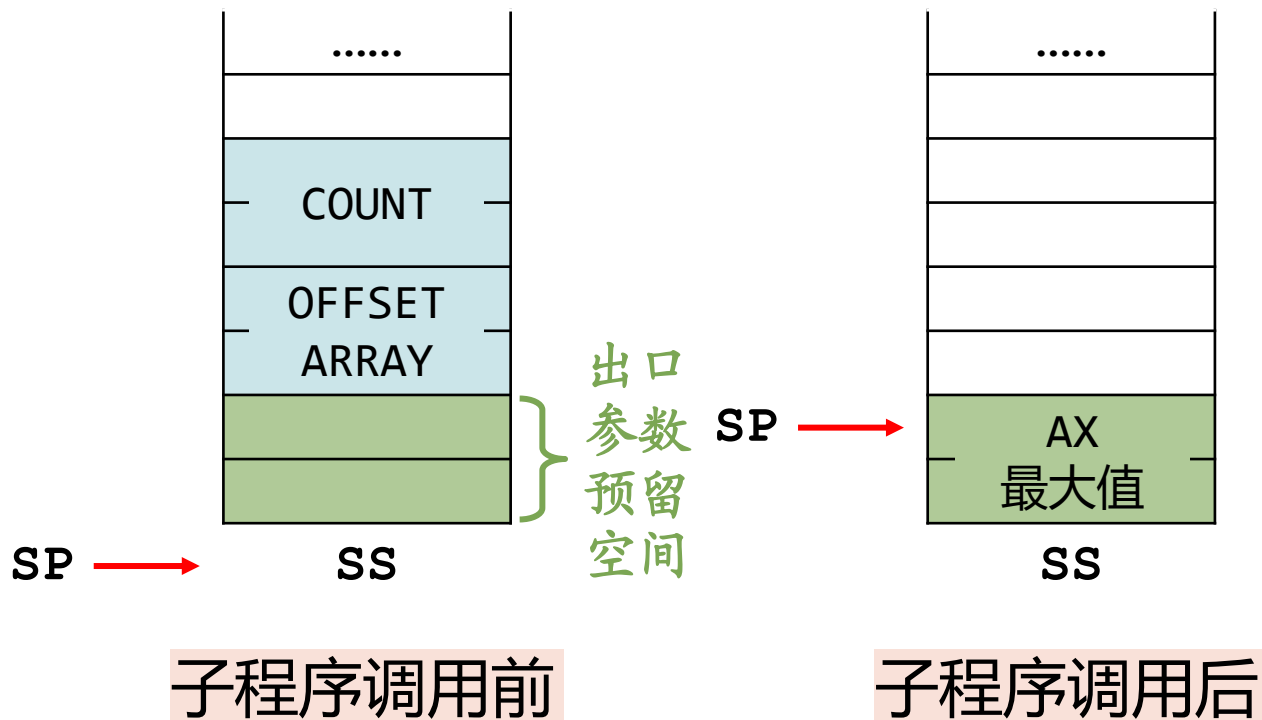
需要对子程序
调用的情况十
分清楚!

- 堆栈的使用
 - 主程序中
 - ① ◆ 入栈入口参数
 - ⑧ ◆ 出栈出口参数
 - 子程序调用和返回
 - ② ◆ 调用：入栈返回地址
 - ⑦ ◆ 返回：出栈返回地址
 - 子程序执行
 - ③ ◆ 起始：入栈通用寄存器
 - ④ ◆ 运算之前：出栈入口参数
 - ⑤ ◆ 运算结束：入栈出口参数
 - ⑥ ◆ 结束：出栈通用寄存器

04 | 子程序设计-堆栈传递参数-主程序

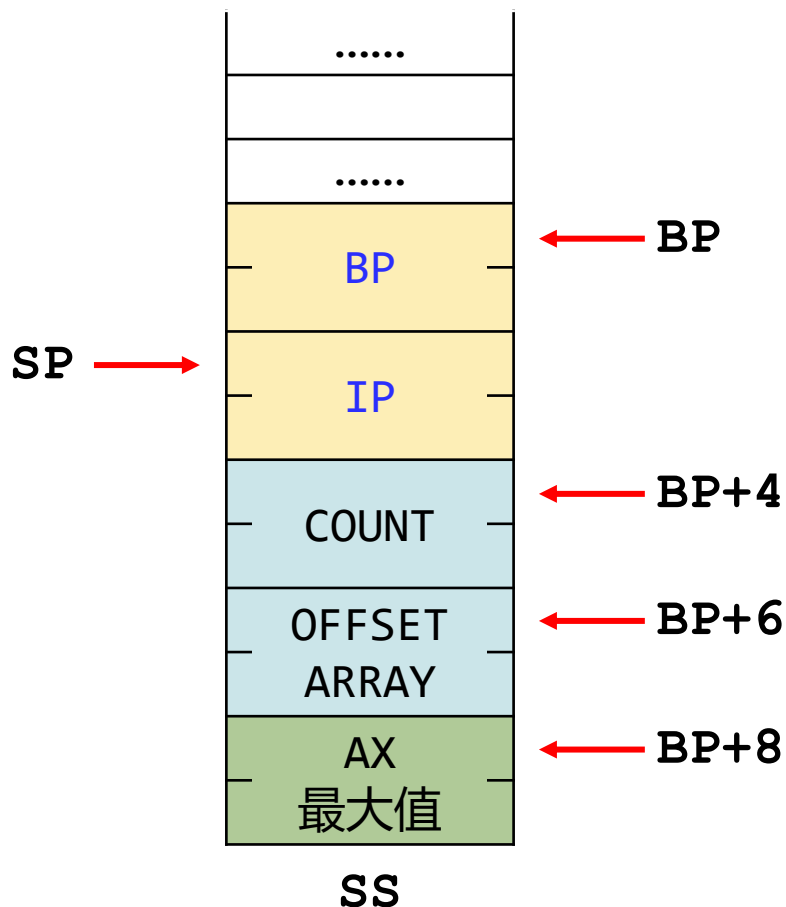
示例：设计一个子程序，求指定数组
ARRAY 的最大值，最大值保存于 Max

```
.....  
SUB SP, 2  
LEA AX, Array  
PUSH AX  
PUSH Count  
CALL FindMax  
POP Max  
.....
```



04 | 子程序设计-堆栈传递参数-子程序

示例：设计一个子程序，求指定数组
ARRAY 的最大值，最大值保存于 Max



```
FindMax PROC
    PUSH BP
    MOV BP, SP
    PUSH SI
    PUSH CX
    PUSH AX
    MOV CX, [BP+4]
    MOV SI, [BP+6]
    MOV AX, 8000H
AGAIN:  CMP AX, [SI]
        JG SKIP
        MOV AX, [SI]
SKIP:   ADD SI, 2
        LOOP AGAIN
        MOV [BP+8], AX
        POP AX
        POP CX
        POP SI
        POP BP
        RET 4
FindMax ENDP
```

04 | 子程序设计-堆栈传递参数注意事项

子程序的参数传递

- 调用子程序时
 - 主程序**为返回的参数预留空间**
 - 再将所需的**入口参数入栈**
- 子程序执行时
 - 正常使用堆栈，来暂存通用寄存器的值，保护现场
 - **使用 BP 访问堆栈的非栈顶元素，获取入口参数**
 - **使用 BP 设置预留空间的出口参数**
- 子程序返回时
 - 子程序应**使用 RET n 的返回指令，跳至堆栈中的出口参数**
 - 主程序**出栈当前的栈顶元素——出口参数**

完全理解这个过程，就能正确使用堆栈传递参数

04 | 子程序设计-子程序的嵌套与递归

子程序的嵌套与递归的定义

- 子程序的嵌套：子程序内调用**其他子程序**
- 子程序的递归：子程序内调用**本身**

注意事项

- **嵌套深度**理论上无限制，实际上受限于堆栈空间的大小
- 嵌套子程序的设计要正确使用 **CALL** 和 **RET** 指令
- 尤其需要注意**寄存器的保护和恢复**，避免各层子程序之间因寄存器使用冲突而出错
- 被递归调用的子程序必须具有**递归结束**的条件

04 | 子程序设计-子程序的嵌套示例

示例：将 AL 寄存器内的二进制数用十六进制显示出来

； 显示 AL 两位十六进制数

```
ALDISP PROC
    PUSH AX
    PUSH CX
    PUSH AX      ; 暂存数据
    MOV CL, 4
    SHR AL, CL   ; 转换 AL 高 4 位
    CALL H2ASC   ; 子程序调用 (嵌套)
    POP AX       ; 转换 AL 低 4 位
    CALL H2ASC   ; 再次调用子程序
    MOV AH, 2
    MOV DL, 'H'
    INT 21H
    POP CX
    POP AX
    RET
ALDISP ENDP
```

```
H2ASC PROC
    PUSH AX
    PUSH BX
    PUSH DX
    MOV BX, OFFSET ASCII ; BX 指向 ASCII 码表
    AND AL, 0FH ; 取得一位十六进制数

    XLAT CS:ASCII
    MOV DL, AL
    MOV AH, 2
    INT 21H
    POP DX
    POP BX
    POP AX
    RET
ASCII DB 30H,31H,32H,33H,34H,35H,36H,37H,
        38H, 39H
        DB 41H,42H,43H,44H,45H,46H
H2ASC ENDP
```

换码：AL ← CS:[BX+AL],
注意数据在代码段 CS 中

显示16进制数据(判断法)

04 | 子程序设计-子程序的递归示例

示例：计算 $SUM=1!+2!+3!+4!+5!$

● 子程序

- 功能：计算一个数据的阶乘
- 入口参数：数值 BX
- 出口参数：BX 的阶乘 AX

● 主程序

- 循环初始化
- 循环体内容完成各阶乘的累加

```
MOV CX, 5
MOV DX, 0
SUM: MOV BX, CX
CALL FACT
ADD DX, AX
LOOP SUM
```

```
; 计算 BX 的阶乘
; 入口参数为 BX, 出口参数为 AX
FACT PROC
    AND BX, BX
    JZ FACT1 ; BX=0, 则其阶乘为 1
    PUSH BX
    DEC BX
    CALL FACT ; 递归调用
    POP BX
    MUL BL
    RET
FACT1: MOV AX, 1
    RET
FACT ENDP
```

以下指令中，子程序中一定包含的指令是（ ）

- ☐ A CALL DELAY
- ☐ B PUSH AX
- ☐ C MOV AX, 0
- ☒ D RET

04 | 子程序设计-子程序应用示例1

示例1：将一个字节数据以二进制形式显示

- 子程序功能：将一个字节数据以二进制形式显示

- 入口参数：要转换的数据 AL
转换后保存的位置 DI
- 出口参数：以 [DI] 为首地址的存储单元中保存转换结果，并显示

- 例如：字节变量 B1=45H

- 以二进制显示，则屏幕上应看到“0100 0101B”
- 问题转换为：将字节数据 B1 转换为 BUF 字符串
 - ◆ 判断 B1 每个二进制位的状态，为 BUF 缓冲区赋 ASCII 码值

	数据段

BUF	30
	31
	30
	30
	30
	31
	30
	31
	42
	24

04 | 子程序设计-子程序应用示例1示意

示例1：将一个字节数据以二进制形式显示

字节变量 B1 = 45H

SHL AL, 1

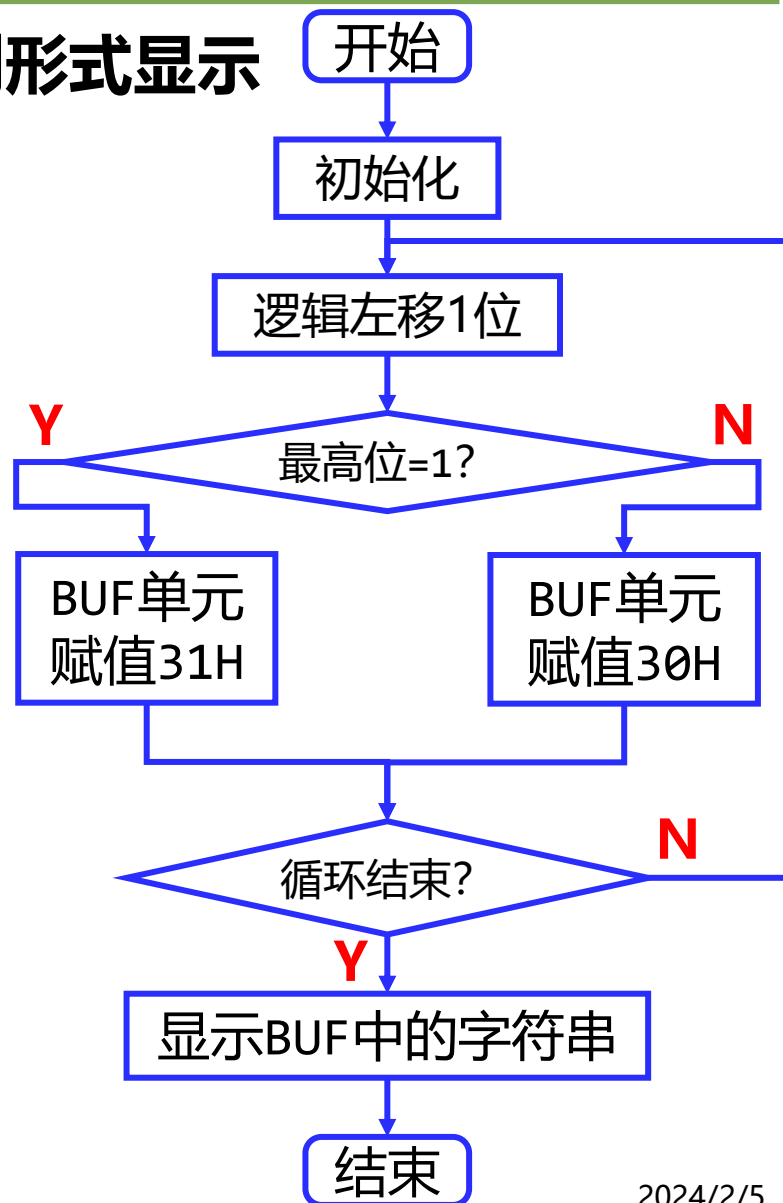


数据段	
.....	
BUF 30	'0'
31	'1'
30	'0'
30	'0'
30	'0'
31	'1'
30	'0'
31	'1'
42	'B'
24	'\$'
.....	

04 | 子程序设计-子程序应用示例1参考

示例1：将一个字节数据以二进制形式显示

```
B2ASC PROC
    MOV DX, DI
    MOV CX, 8
AGAIN: SHL AL, 1
    JC NEXT
    MOV BYTE PTR [DI], 30H
    JMP MODI
NEXT: MOV BYTE PTR [DI], 31H
MODI: INC DI
    LOOP AGAIN
    MOV AH, 09H
    INT 21H
    RET
B2ASC ENDP
```



04 | 子程序设计-子程序应用示例2

示例2：将一个字节数据以十六进制形式显示

- 子程序功能：将一个字节数据以十六进制形式显示

- 入口参数：要转换的数据 AL
转换后保存的位置 DI
- 出口参数：以 [DI] 为首地址的存储单元中保存转换结果，并显示

- 例如：字节变量 B1=4FH

- 以十六进制显示，则屏幕上应看到“4FH”
- 问题转换为：将字节数据 B1 转换为 BUF 字符串
 - ◆ 判断 B1 每个 4 位的值，为 BUF 缓冲区赋 ASCII 码值

数据段	
.....	
34	'4'
46	'F'
48	'H'
24	'\$'
.....	

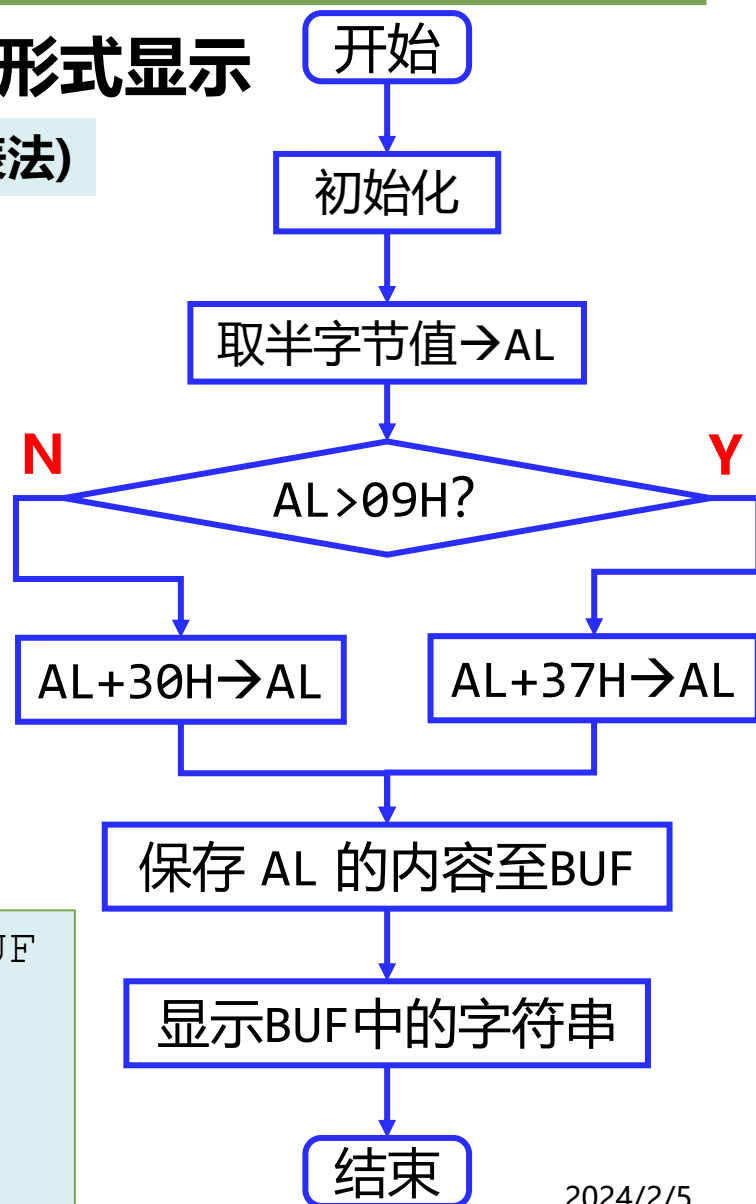
04 | 子程序设计-子程序应用示例2参考

示例2：将一个字节数据以十六进制形式显示

显示16进制数据(查表法)

```
H2ASC PROC
    MOV CL, 4
    MOV BL, AL
    SHR BL, CL
    AND AL, 0FH
    LEA DI, BUF+1
    MOV CX, 2
AGAIN:  CMP AL, 9
        JA  NEXT
        ADD AL, 30H
        JMP SAVE
NEXT:   ADD AL, 37H
SAVE:   MOV [DI], AL
        DEC DI
        MOV AL, BL
    LOOP AGAIN
```

```
.....
    MOV DX, OFFSET BUF
    MOV AH, 09H
    INT 21H
    RET
H2ASC ENDP
```



04 | 子程序设计-子程序应用示例3

示例3：输入十进制有符号数，并保存结果

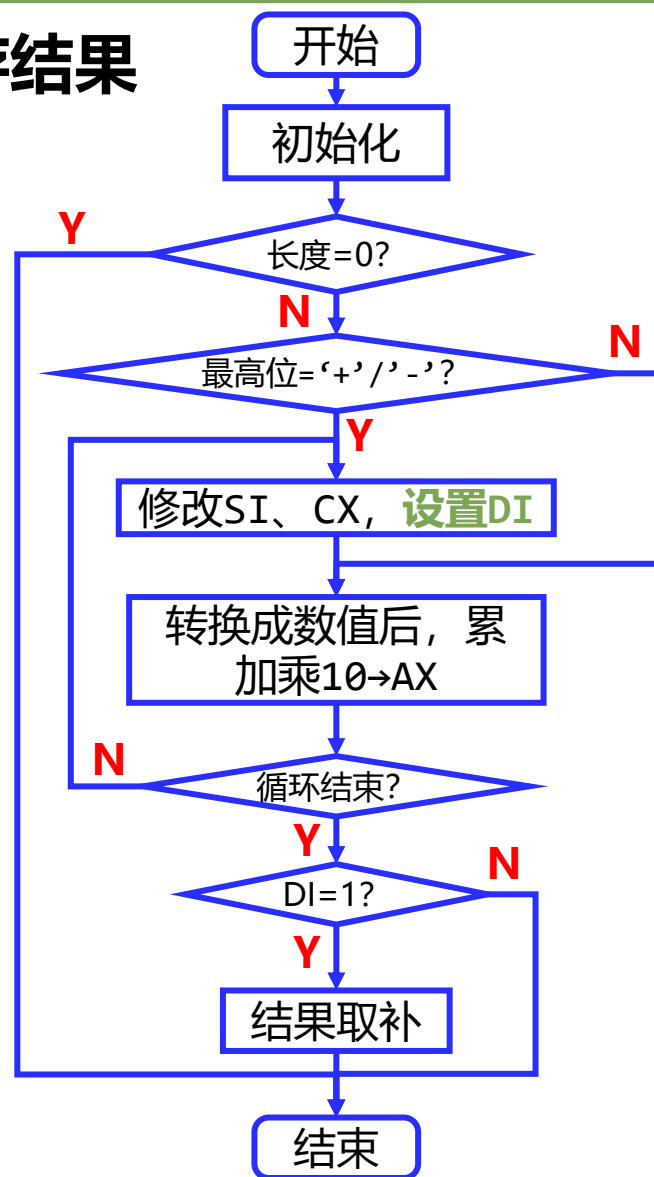
教材 P180

- 输入数据为**字符串或依次输入的字符**，要转换成数值
 - 例如，输入'-1234'，得到 2DH、31H、32H、33H、34H 字符串
 - 转换为数值为 01、02、03、04 四个字节的数据
 - 实际的数值为 $01*1000+02*100+03*10+04$
 - 如果输入的是负数，对结果进行取补 (NEG)
 - ◆ 计算出十进制数值，为 BUF 赋值即可
- 程序的主要任务
 - **将字符串计算出数值**
 - ◆ 数值 = $[(01*10+02)*10+03]*10+04$
 - ◆ 循环结构程序实现 —— $a1*10+a2$

04 | 子程序设计-子程序应用示例3流程图

示例3：输入十进制有符号数，并保存结果

- 程序处理对象：输入的字符串
 - 初始化设置内容：
 - ◆ 数据指针 SI、循环次数 CX、符号变量 DI
 - ◆ 累加和 AX、基数 BX(10)
 - 循环体的主要任务：
 - ◆ 从高位开始，累加和乘以 10，再加数串的一位
- 注意事项
 - 寄存器的使用
 - 数串使用时应将字符转换为数值



04 子程序设计-子程序应用示例3参考

示例3：输入十进制有符号数，并保存结果

READD PROC

```
PUSH BX  
PUSH CX  
PUSH DX  
PUSH SI  
PUSH DI
```

保护要使用的
寄存器

```
LEA DX, BUFD  
MOV AH, 0AH  
INT 21H
```

输入字符串

```
XOR SI, SI  
XOR AX, AX  
XOR CX, CX  
MOV BX, 10
```

循环初始化

```
MOV CL, BUFD+1  
JCXZ READ4
```

判断字符长度

```
LEA SI, BUFD+2  
CMP BYTE PTR [SI], '+'  
JZ READ0  
CMP BYTE PTR [SI], '-'  
JNZ READ2
```

判断输入的
正负符号

```
MOV DI, 1
```

设置正负标志、修
改指针、计数值

```
READ0: DEC CL
```

```
READ1: INC SI
```

```
READ2: MUL BX
```

```
MOV DL, [SI]
```

```
SUB DL, 30H
```

```
MOV DH, 0
```

```
ADD AX, DX
```

```
LOOP READ1
```

主循环体：
乘10累加 → AX

```
READ3: CMP DI, 0
```

```
JZ READ4
```

修正负数

```
NEG AX
```

```
READ4: POP DI
```

恢复堆栈

```
POP SI
```

```
POP DX
```

```
POP CX
```

```
POP BX
```

```
RET
```

```
READD ENDP
```

04 | 子程序设计-子程序应用示例4

示例4：输出有符号十进制数的子程序

● 程序处理对象：输出十进制字符串

■ 初始化设置内容：

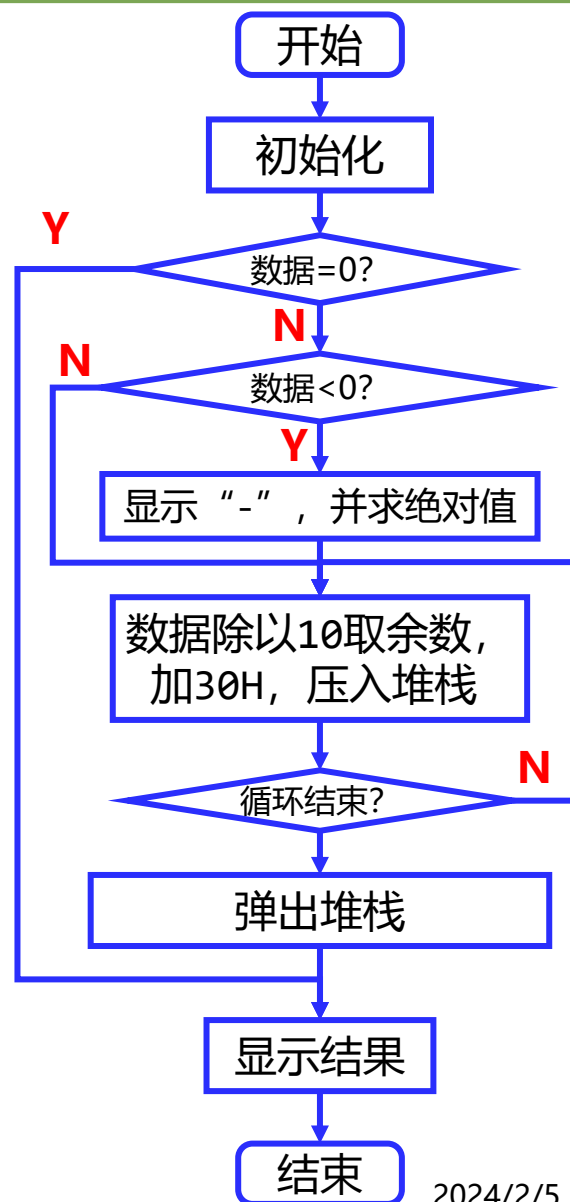
- ◆ 共享变量 WTEMP
- ◆ 商 DX、余数 AX、基数 BX(10)

■ 循环体的主要任务：

- ◆ 将数据除以 10 取余，转换成 ASCII 码后，压入堆栈

● 注意事项

- 堆栈中结束标志的设置
- 数串使用时应将数值转换为字符



04 | 子程序设计-子程序应用示例4参考

示例4：输出有符号十进制数的子程序

WRITE PROC

```
PUSH AX  
PUSH BX  
PUSH DX
```

保护要使用的
寄存器

```
MOV AX, WTEMP
```

```
TEST AX, AX
```

测试数据性质

```
JNZ WRITE1
```

```
MOV DL, '0'
```

```
MOV AH, 2
```

```
INT 21H
```

```
JMP WRITE5
```

数据为0

```
WRITE1: JNS WRITE2
```

```
MOV BX, AX
```

数据为负

```
MOV DL, '-'
```

```
MOV AH, 2
```

```
INT 21H
```

```
MOV AX, BX
```

```
NEG AX
```

```
WRITE2: MOV BX, 10
```

```
PUSH BX
```

设置结束符
和除数

```
WRITE3: CMP AX, 0
```

```
JZ WRITE4
```

```
SUB DX, DX
```

```
DIV BX
```

```
ADD DL, 30H
```

```
PUSH DX
```

```
JMP WRITE3
```

主循环体：
数据除以10，
取余数+30H
并压入堆栈

```
WRITE4: POP DX
```

```
CMP DL, 10
```

```
JE WRITE5
```

```
MOV AH, 2
```

```
INT 21H
```

```
JMP WRITE4
```

从堆栈中依次
弹出数据，并
显示

```
WRITE5: POP DX
```

```
POP BX
```

```
POP AX
```

恢复堆栈

```
RET
```

```
WRITE ENDP
```

本章小结

- **必须熟练掌握**汇编语言源程序的**完整结构**格式
- **掌握**常用的转移、循环指令功能及相应结构的程序设计方法，**能够在 DEBUG 下断点调试程序**
 - 给定的题目能够分析，画出流程图，编写并调试程序
- **熟悉**子程序的基本使用，**掌握**通过**变量、寄存器**传递参数的方法
 - 子程序使用时的**寄存器保护与恢复**
- **必须熟练编写**汇编语言程序



河南大學
Henan University

Q&A

主讲教师：舒高峰

电子邮箱：gaofeng.shu@henu.edu.cn

联系电话：13161693313

