

第4章 第1个程序

- ▶ 4.1 程序执行的基本过程
- ▶ 4.2 程序的基本结构
- ▶ 4.3 程序执行过程的跟踪

4.1 程序执行的基本过程

- ▶ 一个汇编语言程序从写出到最终执行的简要过程：

编写--> 编译--> 连接--> 执行

4.2 程序的基本结构

```
assume cs:codesg
```

```
codesg segment
```

```
start:  mov ax,0123H
```

```
        mov bx,0456H
```

```
        add ax,bx
```

```
        add ax,ax
```

```
        mov ax,4c00h
```

```
        int 21h
```

```
codesg ends
```

```
end
```

▶ 汇编指令

▶ 伪指令

XXX segment

XXX ends

end

assume

4.2 源程序

▶ 源程序中的“程序”

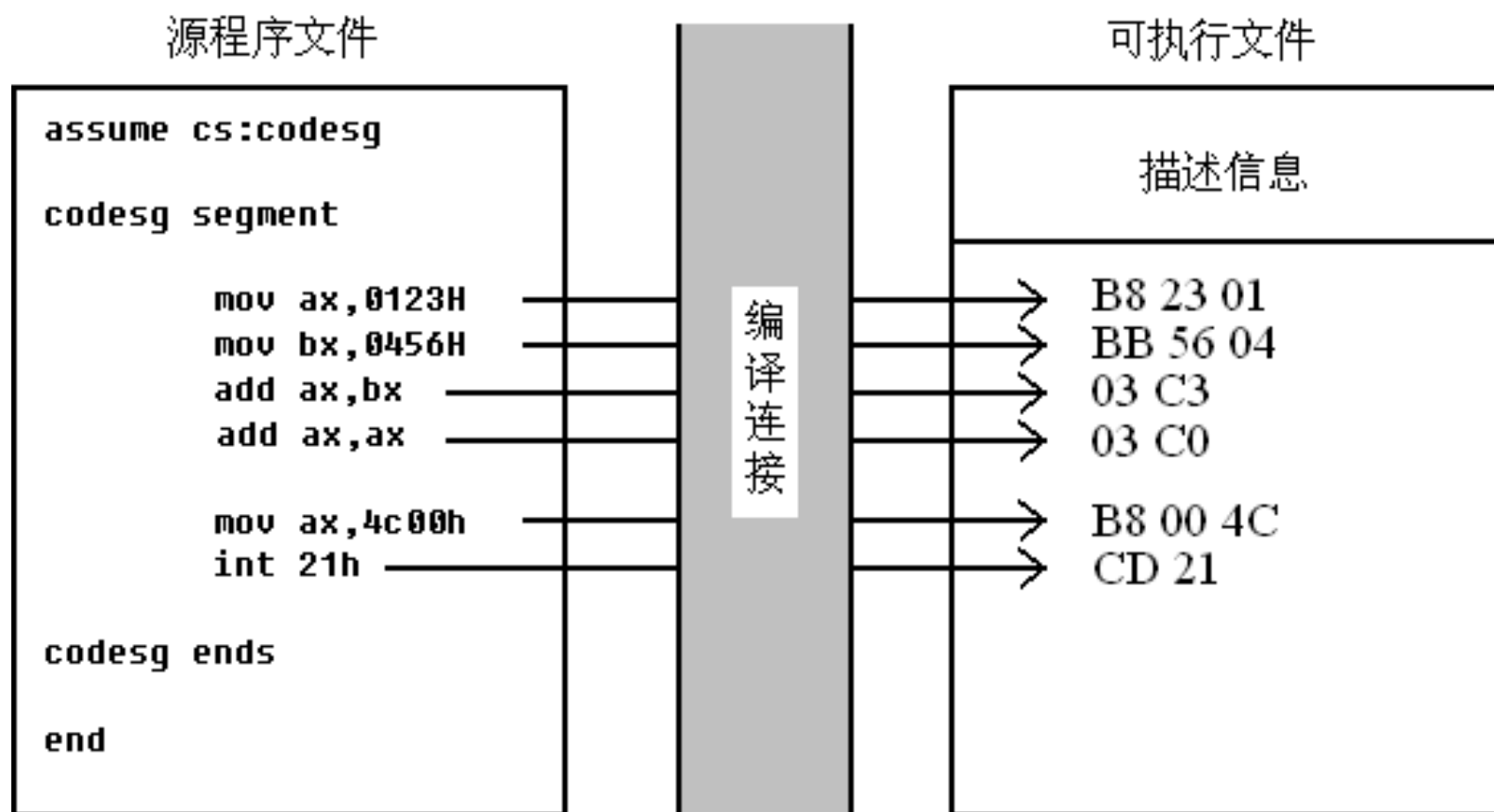
- 汇编源程序：

 伪指令 （编译器处理）

 汇编指令（编译为机器码）

- 程序：源程序中最终由计算机执行、处理的指令或数据。

程序经编译连接后变为机器码



4.2 程序的基本结构

▶ 标号

- 一个标号指代了一个地址。
- `codesg`: 放在`segment`的前面, 作为一个段的名称, 这个段的名称最终将被编译、连接程序处理为一个段的段地址。

4.2 程序的基本结构

▶ 程序的结构

- 任务：编程运算 2^3 。
 - 定义一个段
 - 实现处理任务
 - 程序结束
 - 段与段寄存器关联

汇编程序

```
assume cs:abc
abc segment
    mov ax,2
    add ax,ax
    add ax,ax
abc ends
end
```

4.2 程序的基本结构

▶ 程序返回

- 程序结束后，将CPU的控制权交还给使它得以运行的程序，这个过程为：程序返回。
- 如何返回呢？

4.2 程序的基本结构

▶ 程序返回

- 应该在程序的末尾添加返回的程序段。

```
mov ax,4c00H
```

```
int 21H
```

- 这两条指令所实现的功能就是程序返回。

▶ 几个和结束相关的内容

段结束、程序结束、程序返回

目 的	相关指令	指令性质	指令执行者
通知编译器一个段结束	段名 ends	伪指令	编译时，由编译器执行
通知编译器程序结束	end	伪指令	编译时，由编译器执行
程序返回	mov ax,4c00H int 21H	汇编指令	编译时，由CPU执行

(1) 在DOS中直接执行 1.exe 时，是正在运行的command将1.exe中的程序加载入内存。

(2) command设置CPU的CS:IP指向程序的第一条指令（即程序的入口），从而使程序得以运行。

(3) 程序运行结束后，返回到command中，CPU继续运行command。

程序运行的原理

▶ 汇编程序从写出到执行的过程：

编程 → 1.asm → 编译 → 1.obj → 连接 → 1.exe → 加载 → 内存中的程序 → 运行
(edit) (masm) (link) (command) (CPU)

4.3 程序执行过程的跟踪

- ▶ 观察程序的运行过程，可以使用Debug。
- ▶ Debug 可以将程序加载入内存，设置 CS:IP指向程序的入口，但Debug并不放弃对CPU 的控制。
- ▶ 所以，我们可以使用Debug 的相关命令来单步执行程序，查看每条指令指令的执行结果。





4.3 程序执行过程的跟踪

- ▶ 用R命令可以看各个寄存器的设置情况：

```
C:\masm>debug 1.exe
-r
AX=0000  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=0000  NU UP EI PL NZ NA PO NC
12AE:0000 0000 B82301      MOV     AX,0123
-
```

- ▶ Debug将程序加载入内存后，cx中存放的是程序的长度。
- ▶ 1.exe 中程序的机器码共有15个字节。

► DOS中.EXE文件中的程序的加载过程如下

1		2		3		4	
<p>找到一段起始地址为SA:0000（即起始地址的偏移地址为0）的容量足够的空闲内存区；</p>		<p>在这段内存区的前256个字节中，创建一个称为程序的前缀（PSP）的数据区，DOS要利用PSP来和被加载程序进行通信；</p> <p>（读者可能不理解PSP的作用，不过没有关系。我们并不研究DOS的原理，只要知道有这个东西就可以了。）</p>		<p>从这段内存区的256字节处开始（在PSP的后面），将程序装入，程序的地址被设为SA+10H:0；</p> <p>（空闲的内存区从SA:0开始，0~255字节为PSP，从256字节处开始存放程序，为更好地区分PSP和程序，DOS一般将它们划分到不同的段中，所以，有了这样的地址安排：</p> <p>空闲内存区：SA:0 PSP区：SA:0 程序区：SA+10H:0</p> <p>注意：PSP区和程序区虽然物理地址连续，却有不同的段地址。）</p>		<p>将该内存区的段地址存入DS中，初始化其他相关寄存器后，设置CS:IP指向程序的入口。</p>	

EXE文件中的程序的加载过程

- ▶ 注意： 有一步称为**重定位**的工作我们没有讲解，这个问题和操作系统的关系较大，我们不作讨论。

EXE文件中的程序的加载过程

▶ 总结

- 程序加载后，ds中存放着程序所在内存区的段地址，这个内存区的偏移地址为 0，则程序所在的内存区的地址为：ds:0；
- 这个内存区的前256 个字节中存放的是PSP，dos用来和程序进行通信。
- 从 256字节处向后的空间存放的是程序。

EXE文件中的程序的加载过程

▶ 总结（续）

- 所以，我们从ds中可以得到PSP的段地址SA，PSP的偏移地址为 0，则物理地址为 $SA \times 16 + 0$ 。
- 因为PSP占256（100H）字节，所以程序的物理地址是：

$$SA \times 16 + 0 + 256 = SA \times 16 + 16 \times 16 = (SA + 16) \times 16 + 0$$

可用段地址和偏移地址表示为： $SA + 10:0$ 。

4.3 程序执行过程的跟踪

- ▶ 用U命令查看一下其他指令：

```
C:\masm>debug 1.exe
-r
AX=0000  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=0000  NU UP EI PL NZ NA PO NC
12AE:0000 0000 B82301          MOV     AX,0123
-u
12AE:0000 B82301          MOV     AX,0123
12AE:0003 BB5604          MOV     BX,0456
12AE:0006 03C3          ADD     AX,BX
12AE:0008 03C0          ADD     AX,AX
12AE:000A B8004C          MOV     AX,4C00
12AE:000D CD21          INT     21
12AE:000F 83E201          AND     DX,+01
12AE:0012 BA85E2          MOV     DX,E285
12AE:0015 2E          CS:
12AE:0016 A385E2          MOV     [E285],AX
12AE:0019 E9A5FC          JMP     FCC1
12AE:001C 803EE70400      CMP     BYTE PTR [04E7],00
```

4.3 程序执行过程的跟踪

- ▶ 用T命令单步执行程序中的每一条指令，并观察每条指令的执行结果，到了 int 21，我们要用P命令执行：

```
AX=0AF2  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=13F2  ES=13F2  SS=1402  CS=1402  IP=000A  NU UP EI PL NZ AC PO NC
1402:000A B8004C          MOV     AX,4C00
-t

AX=4C00  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=13F2  ES=13F2  SS=1402  CS=1402  IP=000D  NU UP EI PL NZ AC PO NC
1402:000D CD21          INT     21
-p

Program terminated normally
```

4.9 程序执行过程的跟踪

- ▶ int 21 执行后，显示 “Program terminated normally”，返回到Debug中。
- ▶ 表示程序正常结束。
- ▶ 注意，要使用P命令执行int 21。

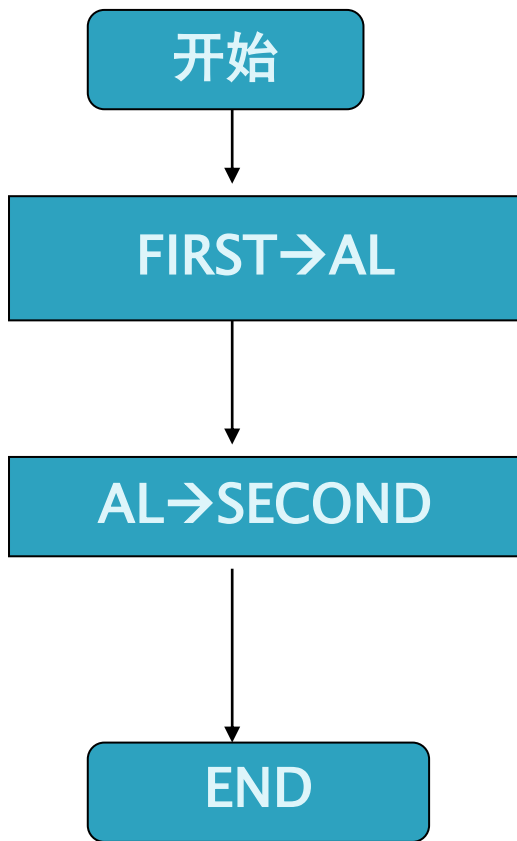
4.3 程序执行过程的跟踪

- ▶ 使用Q命令退出Debug，将返回到command中，因为Debug是由command加载运行的。

4.3 程序执行过程的跟踪

- ▶ 我们在 DOS 中用 “Debug 1.exe” 运行 Debug 对 1.exe 进行跟踪时，程序加载的顺序是：command 加载 Debug，Debug 加载 1.exe。
- ▶ 返回的顺序是：从 1.exe 中的程序返回到 Debug，从 Debug 返回到 command。

练习 ▶ 将一字节数据从数据段的某个单元传送到另一个单元



```
DATA SEGMENT
    FIRST DB 4CH
    SECOND DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
    MOV AX,DATA
    MOV DS,AX
    MOV AL,FIRST
    MOV SECONT,AL
CODE ENDS
END
```


第5章 [bx]和loop指令

- ▶ 5.1 [bx]
- ▶ 5.2 Loop指令
- ▶ 5.3 在Debug中跟踪用loop指令实现的循环程序
- ▶ 5.4 Debug和汇编编译器Masm对指令的不同处理
- ▶ 5.5 loop和[bx]的联合应用
- ▶ 5.6 段前缀
- ▶ 5.7 一段安全的空间
- ▶ 5.8 段前缀的使用

[bx]和内存单元的描述

- ▶ 要完整地描述一个内存单元，需要两种信息：
 - (1) 内存单元的**地址**；
 - (2) 内存单元的**长度**（类型）。

如：[0]表示一个内存单元，
偏移地址为0，段地址默认在ds中，
单元的长度（类型）可以由具体指令中的
其他操作对象（比如说寄存器）指出。

[bx]和内存单元的描述

- ▶ [bx]同样也表示一个内存单元，它的偏移地址在bx中，比如下面的指令：
 - `mov ax,[bx]`
 - `mov al,[bx]`

loop

- ▶ 英文单词 “loop”有循环的含义，显然这个指令和循环有关。

5.1 [bx]

▶ 我们看一看下面指令的功能：

◦ `mov ax,[bx]`

功能：bx 中存放的数据作为一个偏移地址，段地址 默认在ds 中，将ds:[bx]处的数据送入ax中。

即： $(ax) = (ds * 16 + (bx))$;

5.1 [bx]

► `mov [bx],ax`

功能：bx中存放的数据作为一个偏移地址，段地址默认在ds中，将ax中的数据送入内存ds:[bx]处。

即： $(ds * 16 + (bx)) = (ax)$ 。

5.1 [bx]

▶ 问题5.1

程序和内存中的情况如下图所示，写出程序执行后，21000H~21007H 单元中的内容。

BE	21000H
00	21001H
	21002H
	21003H
	21004H
	21005H
	21006H
	21007H

5.1 [bx]

BE	21000H
00	21001H
	21002H
	21003H
	21004H
	21005H
	21006H
	21007H

```
mov ax,2000H
```

```
mov ds,ax
```

```
mov bx,1000H
```

```
mov ax,[bx]
```

```
inc bx
```

```
inc bx
```

```
mov [bx],ax
```

```
inc bx
```

```
inc bx
```

```
mov [bx],ax
```

```
inc bx
```

```
mov [bx],al
```

```
inc bx
```

```
mov [bx],al
```


5.2 Loop指令

- ▶ 指令格式：

- Loop 标号

- ▶ CPU 执行loop指令的时候，要进行两步操作：

- ① $(cx) = (cx) - 1$ ；

- ② 判断cx中的值，不为零则转至标号处执行程序，如果为零则向下执行。

5.2 Loop指令

- ▶ cx中的值影响着loop指令的执行结果。
- ▶ 通常，我们可以用loop指令来实现循环功能，
cx 中存放循环次数。

5.2 Loop指令

▶ 应用举例：

- 任务1：编程计算 2^2 ，结果存放在ax中。
- 任务2：编程计算 2^3 。
- 任务3：编程计算 2^{12} 。

5.2 Loop指令

- ▶ cx和loop指令结合实现循环的框架如下：

```
mov cx,循环次数  
s:  
    循环执行的程序段  
loop s
```

5.2 Loop指令

- ▶ 问题5.2用加法循环，实现 123×236 ，结果存在ax 中。

```
assume cs:code
code segment
    mov ax,0
    mov cx,236
s:add ax,123
    loop s
    mov ax,4c00h
    int 21h
code ends
end
```

5.3 在Debug中跟踪loop指令

- ▶ 问题：计算ffff:0006单元中的数乘以3，结果存储在dx中。

（1）运算后的结果是否会超出dx所能存储的范围？

5.3 在Debug中跟踪loop指令

(2) 用循环累加实现乘法

我们将ffff:0006单元中的数赋值给ax，用dx进行累加。先设(dx)=0，然后做3次(dx)=(dx)+(ax)。

(3) ffff:0006单元是一个字节单元，ax是一个16位寄存器，数据长度不一样，如何赋值？

5.3 在Debug中跟踪loop指令

```
assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax
    mov bx,6
    mov al,[bx]
    mov ah,0
    mov dx,0

    mov cx,3
s: add dx,ax
   loop s

    mov ax,4c00h
    int 21h
code ends
end
```


5.4 Debug和汇编编译器Masm对指令的不同处理

- ▶ 我们在Debug中写过类似的指令：
 `mov ax,[0]`
 表示将ds:0处的数据送入ax中。
- ▶ 但是在汇编源程序中，指令“`mov ax,[0]`”
 被编译器当作指令“`mov ax,0`”处理。
- ▶ 示例

5.4 Debug和汇编编译器Masm对指令的不同处理

- ▶ 任务：将内存2000:0、2000:1 、2000:2、2000:3单元中的数据送入al, bl, cl, dl中。
 - (1) 在Debug中编程实现
 - (2) 汇编程序实现

5.5 loop和[bx]的联合应用

- ▶ 问题:

计算ffff:0~ffff:b单元中的数据的和,
结果存储在dx中。

5.5 loop和[bx]的联合应用

- ▶ 计算ffff:0~ffff:b单元中的数据的和，结果存储在dx中。

- ▶ 分析：

（1）运算后的结果是否会超出 dx 所能存储的范围？

5.5 loop和[bx]的联合应用

(2) 类型的匹配和结果的不超界。

$(dx) = (dx) + \text{内存中的8位数据};$

$(dl) = (dl) + \text{内存中的8位数据};$

第一种方法中的问题是两个运算对象的类型不匹配，第二种方法中的问题是结果有可能超界。

5.5 loop和[bx]的联合应用

- ▶ 将内存单元中的 8 位数据赋值到一个16位寄存器ax中，再将ax中的数据加到dx上，从而使两个运算对象的类型匹配并且结果不会超界.

程序代码:

Assume cs:code
Code segment

mov ax,0fffh

mov ds,ax

mov bx,0

mov dx,0

mov cx,12

S: mov al,[bx]

mov ah,0

add dx,ax

inc bx

Loop s

mov ax,4c00h

int 21h

Code ends

end

5.6 段前缀

- ▶ 显式地指明内存单元的段地址的
“ds:”、“cs:”、“ss:”或“es:”，在
汇编语言中称为段前缀。

5.7 一段安全的空间

- ▶ 在8086模式中，随意向一段内存空间写入内容是很危险的，因为这段空间中可能存放着重要的系统数据或代码。
- ▶ 比如下面的指令：
 mov ax,1000h
 mov ds,ax
 mov al,0
 mov ds:[0],al

5.7 一段安全的空间



The screenshot shows a DOS debug window with the following registers and memory addresses:

```
-r
AX=0000 BX=0000
DS=0B2D ES=0B2D
0B3D:0000 B800
-t
AX=0000 BX=0000
DS=0B2D ES=0B2D
0B3D:0003 8ED8
-t
AX=0000 BX=0000 CX=000D DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=0000 ES=0B2D SS=0B3D CS=0B3D IP=0005
0B3D:0005 A32600 MOV [0026],AX
-t
```

A command prompt window titled "16 位 MS-DOS 子系统" is overlaid on the debug window. It contains the following text:

```
命令提示符 - debug p7.exe
NTVDM CPU 遇到无效的指令。
CS:0000 IP:0460 OP:0f 0c 00 d4 03 选择“关闭”终止应用程序。
```

The command prompt window has two buttons: "关闭(C)" and "忽略(I)".

- ▶ 上图是在Windows2000的DOS方式中，在Debug里执行“mov [0026],ax”的结果。如果在实模式（即纯DOS方式）下执行程序p7.exe，将会引起死机。
产生这种结果的原因是0:0026处存放着重要的系统数据，而“mov [0026],ax”将其改写。

5.7 一段安全的空间

- ▶ 一般，DOS方式下，DOS和其他合法的程序一般都不会使用0:200~0:2FF（0:200h~0:2FFh）的256个字节的
空间。
- ▶ 所以，我们使用这段空间是安全的。

5.7 一段安全的空间

- ▶ 先用Debug 查看，如果0:200~0:2FF单元的内容都是0的话，则证明DOS和其他合法的程序没有使用这里。

5.8 段前缀的使用

▶ 问题：

将内存ffff:0~ffff:b段元中的数据
拷贝到 0:200~0:20b单元中。

第6章 包含多个段的程序

- ▶ 6.1 在代码段中使用数据
- ▶ 6.2 在代码段中使用栈
- ▶ 6.3 将数据、代码、栈放入不同的段

引言

- ▶ 前面我们写的程序中，只有一个代码段。
- ▶ 如果程序需要用其他空间来存放数据，我们使用哪里呢？

引言

- ▶ 第5章中，我们说0:200~0:300是相对安全的空间；
- ▶ 可这段空间的容量只有256个字节，如果需要的空间超过256个字节该怎么办呢？

6.1 在代码段中使用数据

- ▶ 编程计算以下8个数的和，结果存在`ax` 寄存器中：
0123H, 0456H, 0789H, 0abcH, 0defH,
0fedH, 0cbaH, 0987H。

6.1 在代码段中使用数据

▶ 程序6.1

```
assume cs:codesg
```

```
codesg segment
```

```
    dw
```

```
    0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
```

```
    mov bx,0
```

```
    mov ax,0
```

```
    mov cx,8
```

```
s: add ax,cs:[bx]
```

```
    add bx,2
```

```
    loop s
```

```
    mov ax,4c00h
```

```
    int 21h
```

```
codesg ends
```

```
end
```

6.1 在代码段中使用数据

▶ 程序6.2

```
assume cs:codesg
codesg segment
    dw
    0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
start: mov bx,0
        mov ax,0
        mov cx,8
        s: add ax,cs:[bx]
            add bx,2
            loop s
        mov ax,4c00h
        int 21h
codesg ends
end start
```

6.1 在代码段中使用数据

- ▶ 注意:

我们在程序的第一条指令的前面加上了一个标号start，而这个标号在伪指令end的后面出现。

- ▶ end的作用:

end 除了通知编译器程序结束外，还可以通知编译器程序的入口在什么地方。

6.1 在代码段中使用数据

- ▶ 有了这种方法，我们就可以这样来安排程序的框架：

```
assume cs:code
code segment
:
数据
:
start:
:
:
代码
:
:
code ends
end start
```

6.2 在代码段中使用栈

- ▶ 利用**栈**，将程序中定义的数据逆序存放。

```
assume cs:codesg
```

```
codesg segment
```

```
dw
```

```
0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
```

```
?
```

```
code ends
```

```
end
```

定义的数据存放在cs:0~cs:15单元中



可将cs:16 ~ cs:31 的内存空间当作栈来用，初始状态下栈为空，**SS:**
SP?

特别提示

- ▶ 检测点6.1（Page 129）
- ▶ 练习：实验5（P133）
- ▶ 没有通过检测点，请不要向下学习！

6.3 将数据、代码、栈放入不同的段

▶ 在前面的内容中，我们将数据、栈和代码都放到了一个段里面。

(1) 把它们放到一个段中使程序显得混乱；

(2) 处理的数据很少时放到一个段里面没有问题，当数据、栈和代码需要的空间超过64KB，就不能放在一个段中。

(——8086模式的限制)

—— 所以，我们应该考虑用多个段来存放数据、代码和栈。

6.3 将数据、代码、栈放入不同的段

▶ ② 伪指令

`assume`

`cs:code,ds:data,ss:stack`

将cs、ds和ss分别和code、data、stack段相连。

6.3 将数据、代码、栈放入不同的段

- ▶ **end start**: 说明了程序的入口，可执行文件中的程序被加载入内存后，CPU的CS:IP被设置指向这个入口，从而开始执行程序中的第一条指令。
- ▶ **Start**: 在“code”段中，这样CPU就将code段中的内容当作指令来执行了。

6.3 将数据、代码、栈放入不同的段

- ▶ 我们在code段中，使用指令：

```
mov ax,stack
```

```
mov ss,ax
```

```
mov sp,16
```

ss:sp指向stack:16， CPU 把stack段当栈空间用。

- ▶ CPU若要访问data段中的数据，则可用 **ds** 指向 **data 段**，用其他的寄存器（如：bx）来存放 data 段中数据的偏移地址。

6.3 将数据、代码、栈放入不同的段

- ▶ 总之，CPU对段中的内容，是当作指令执行，当作数据访问，还是当作栈空间，完全是靠程序中具体的汇编指令，和汇编指令对CS:IP、SS:SP、DS等寄存器的设置来决定的。

综合练习

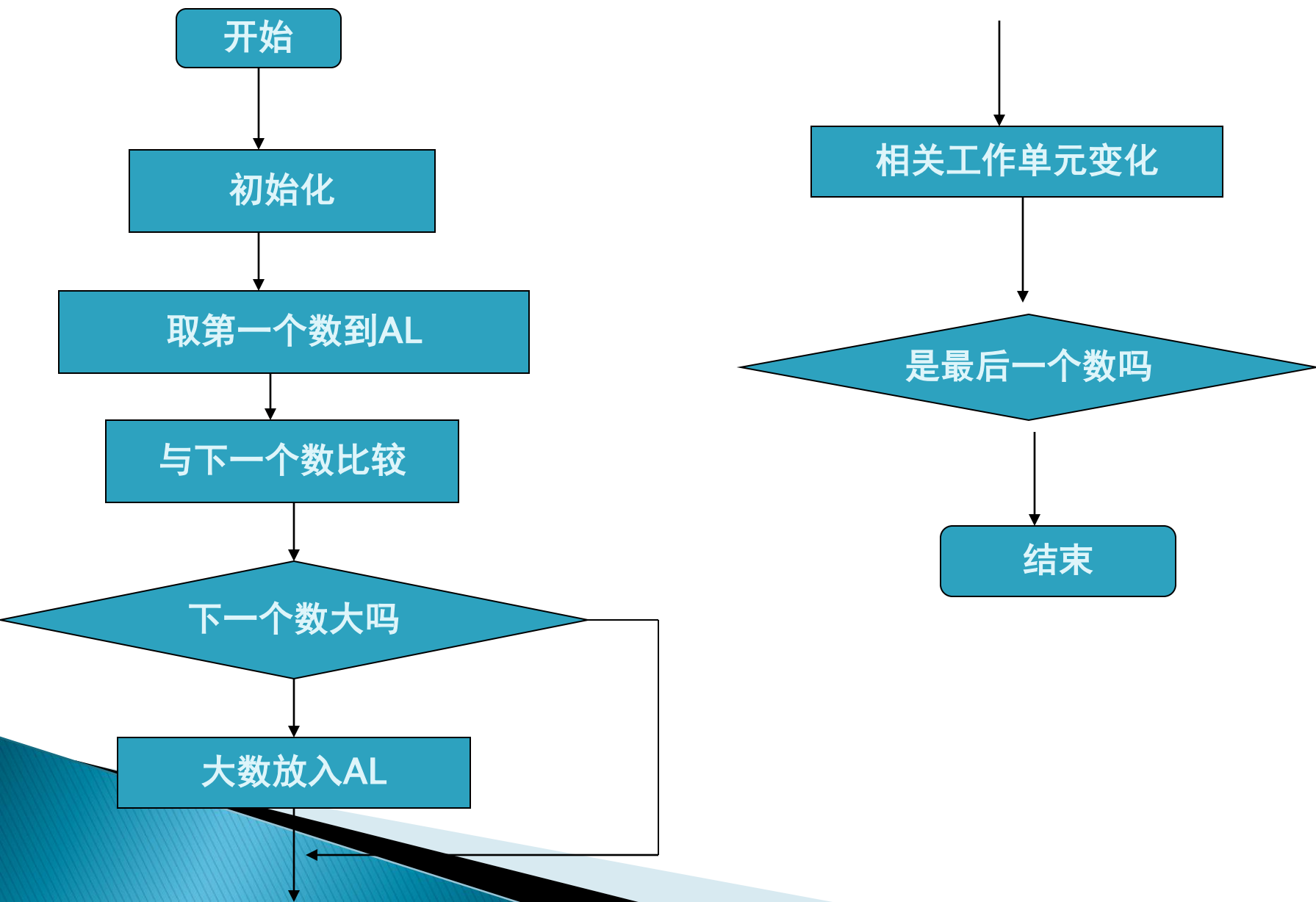
▶ 在无序的0-9中找出最大数。

1、分析问题

2、确定算法

建立一个数据区，并将一个数据指针指向数据区的首地址，从首地址开始顺序取一个数放入a1,并跟下一个数比较，将较大的数保存在a1中。直到最后一个数。

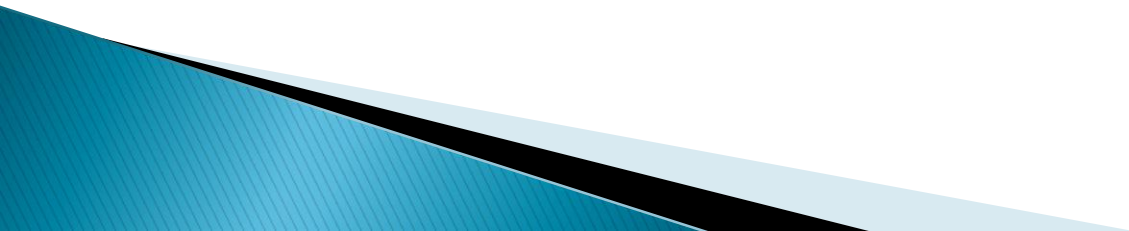
3、绘制流程图



4、确定存储空间和工作单元

BX做数据指针，CX做计数器，AL放最大数，

5、编写程序



DATAS SEGMENT

DB 2,5,4,7,0,8,1,3,9,6

DATAS ENDS

CODES SEGMENT

ASSUME CS:CODES,DS:DATAS,SS:STACKS

START: MOV AX,DATAS

MOV DS,AX

MOV BX,0

MOV CX,10

mov ah,0

mov al,0

L1: CMP AL,[BX]

Jnb NEXT

MOV AL,[BX]

next: inc bx

loop L1

;输出al中的字符数据

add al,48

mov dl,al

mov ah,02

int 21h

MOV AX,4C00H

INT 21H

CODES ENDS

END START

扩展：

1、找100个数中的最值

2、显示的问题：

如何显示一个任意的十进制数？

(AH=02h,int 21h

只能显示dl对应的字符)

