第四章　汇编语言程序格式

4.1 汇编程序功能

4.2 伪操作(汇编、连接工具可识别的说明符)

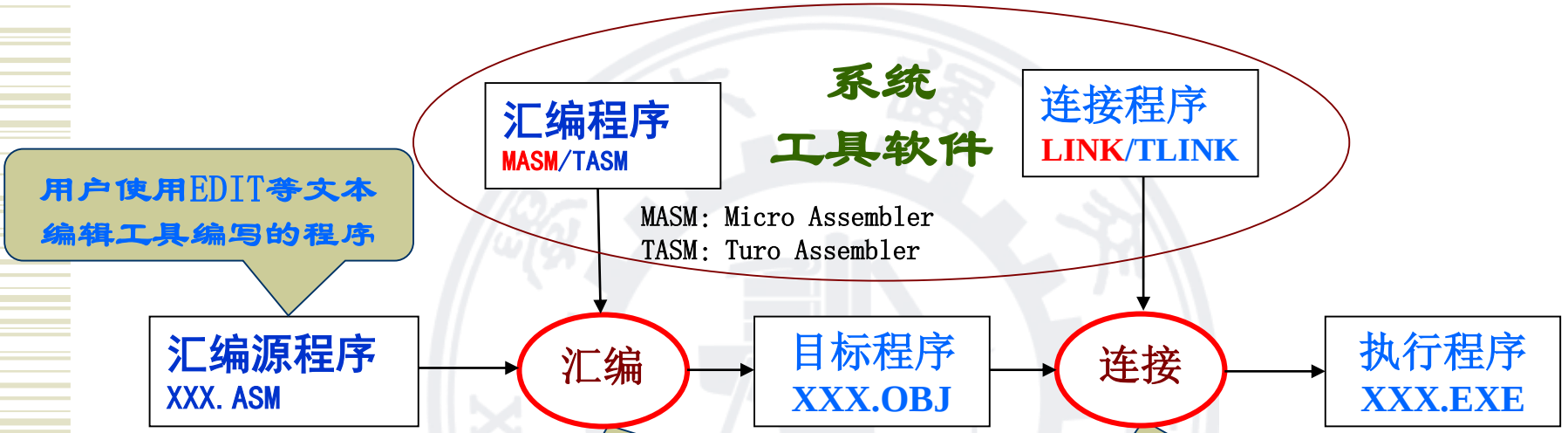
4.3 汇编语言程序格式

4.4 汇编语言程序上机过程

本章目标

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

4.1 汇编语言程序的功能



汇编程序主要功能：

1、检查程序；

2、测出源程序中的语法错误，并给出错误信息；

3、展开宏指令；

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

连接程序主要功能：

1、连接分别产生的目标模块

2、按要求进行段合并组合

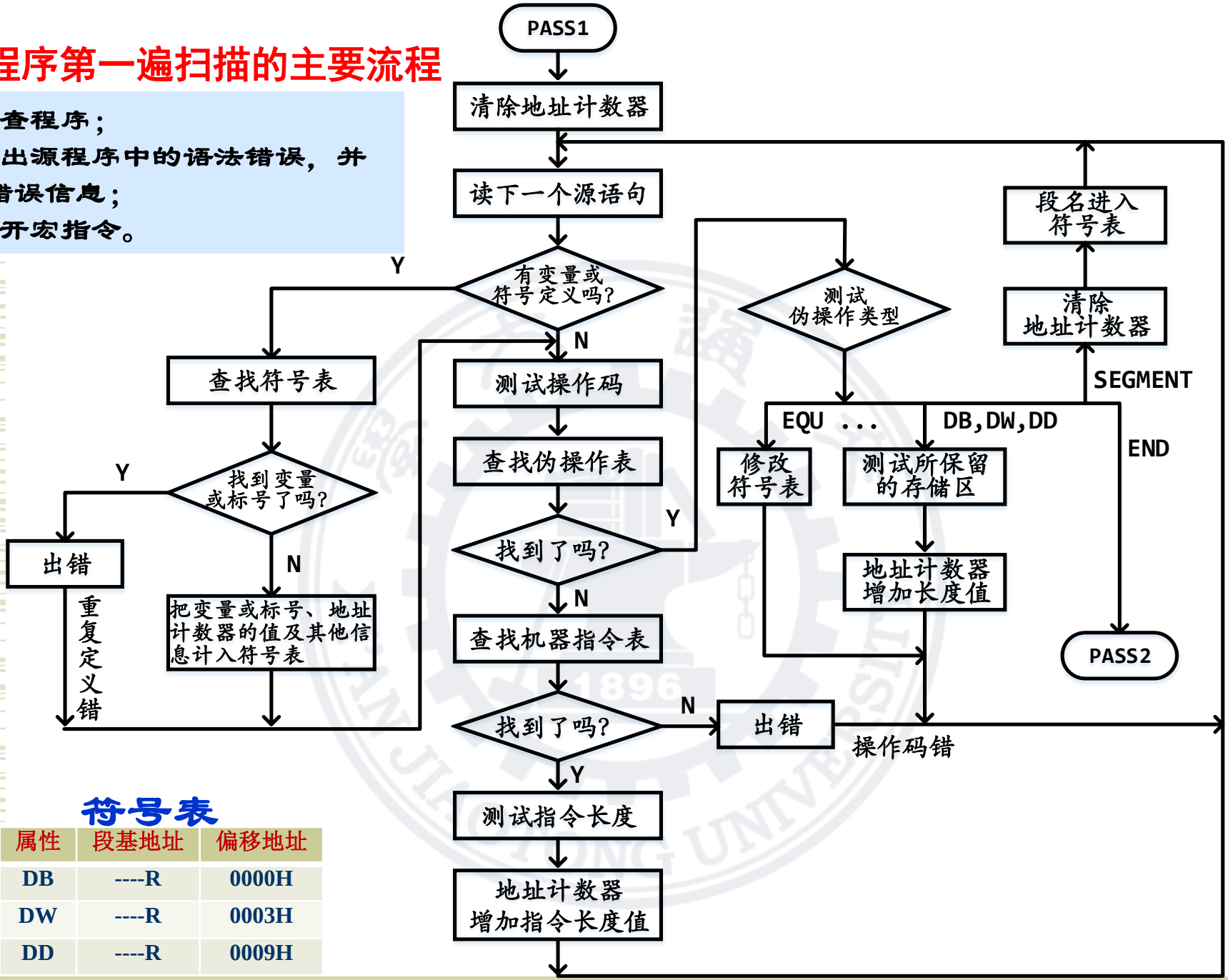
3、解决外部交叉调用

4、产生可重定位的装入模块

5、产生可选的内存映象文件等

汇编程序第一遍扫描的主要流程

1、检查程序；
2、测出源程序中的语法错误，并给出错误信息；
3、展开宏指令。



符号表

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

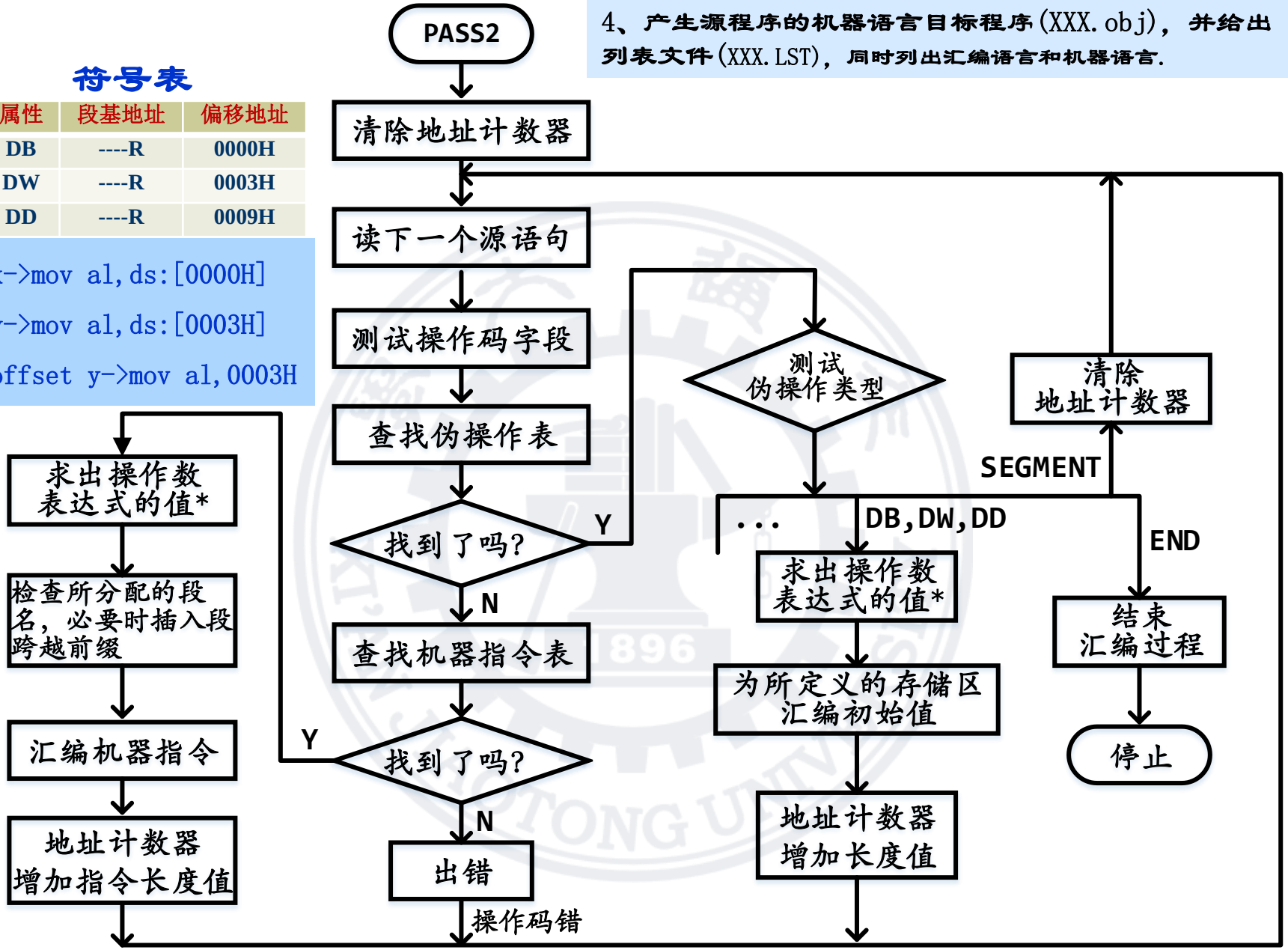
汇编程序第二遍扫描的主要流程

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

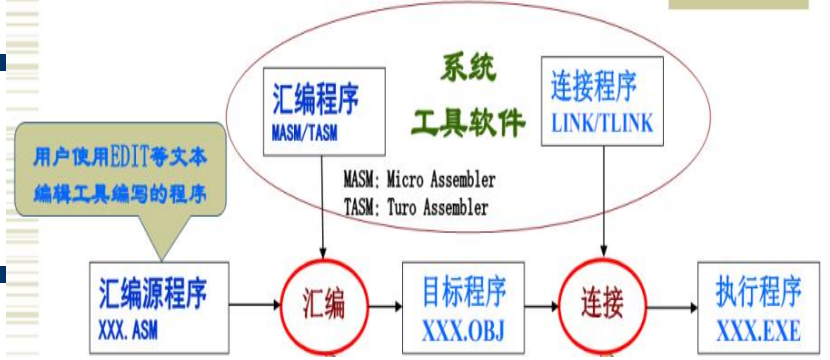
符号表

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

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



汇编语言指令



汇编语言指令分类

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

4.2 伪操作

4.2.1 处理器选择伪操作

4.2.2 段定义伪操作

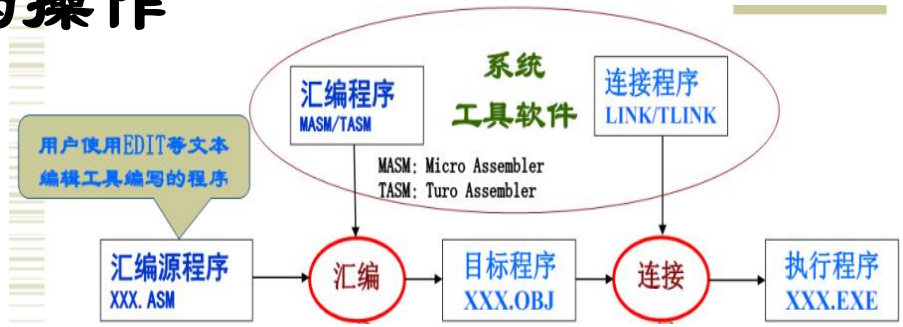
4.2.3 程序开始和结束伪操作

4.2.4 数据定义及存储器分配伪操作

4.2.5 表达式赋值伪操作EQU

4.2.6 地址计数器对准伪操作

4.2.7 基数控制伪操作



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

4.2.1 处理器选择伪操作

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

此类伪操作参看P135

缺省时默认选择8086指令系统

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

4.2.2 段定义伪操作

格式：

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

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

```
code_seg    segment
            ......
            mov   al,  x
            mov   ah,  x+1
            ......
code_seg    ends
```

为了程序结构清晰

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

Segment 伪操作还可以增加存储器分配组合方式类型

格式：

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

注意：s_name只是给段起了个有助于程序阅读的段名，不说明段的任何属性！

- **定位类型**：BYTE、WORD、DWORD、PARA、PAGE

  指定段在存储器分配时的对齐属性

  - PARA：段从小段地址开始（缺省默认） XX...XX0000
  - BYTE：段从任意字节开始 XX...XXXXXX
  - WORD：段从下一字地址开始，偶数地址开始 XX...XXXXX0
  - DWORD：段从下一双字地址开始，开始地址最低2位为0，4的倍数 XX...XXXX00
  - PAGE：段从下一页地址开始（能被256整除） XX...XX00000000

Segment 伪操作还可以增加存储器分配组合方式类型

格式：

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

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

连接程序
如何对程序模块中的段合并

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

1. 多个模块组合时的连接情况

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

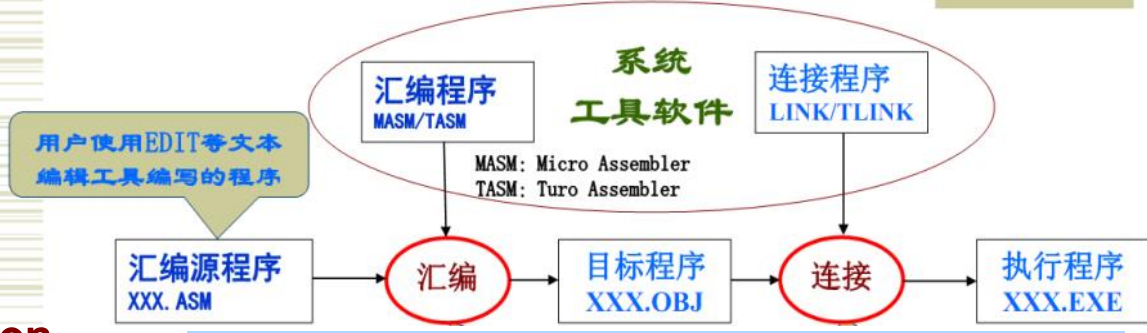
例13.2：

程序员A：源程序模块

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

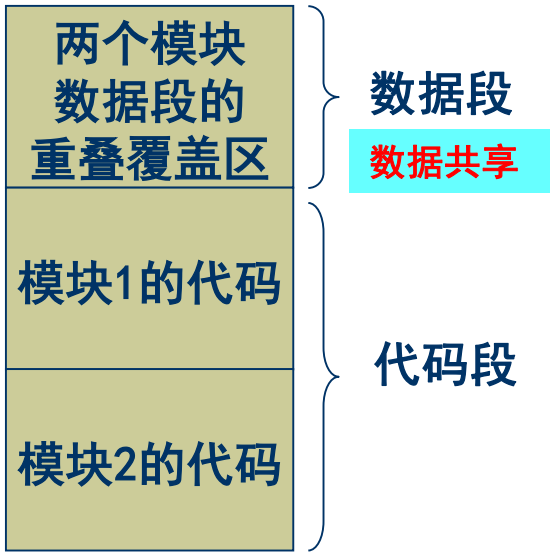
程序员B：源程序模块2

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

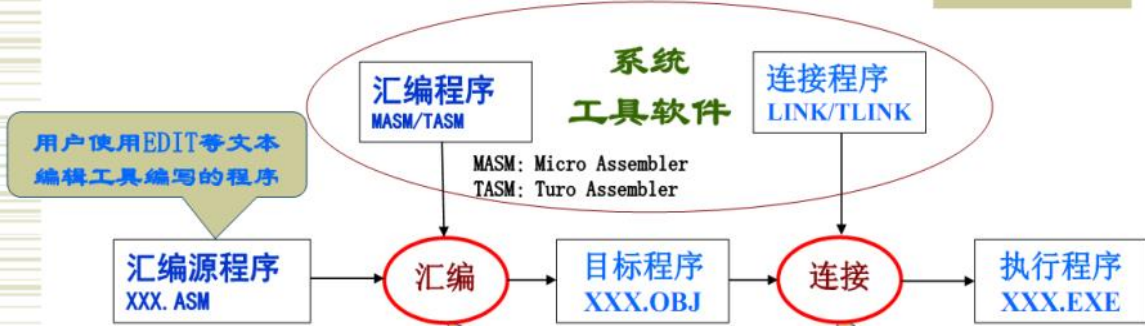


注意：data、code段名助记符只是有助于程序阅读的段名，不说明段的任何属性！

连接后装入模块的存储区分配情况



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



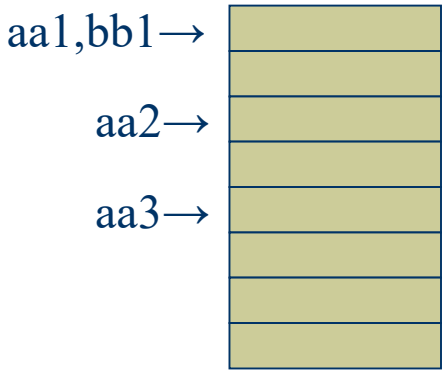
程序员A：源程序模块1

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

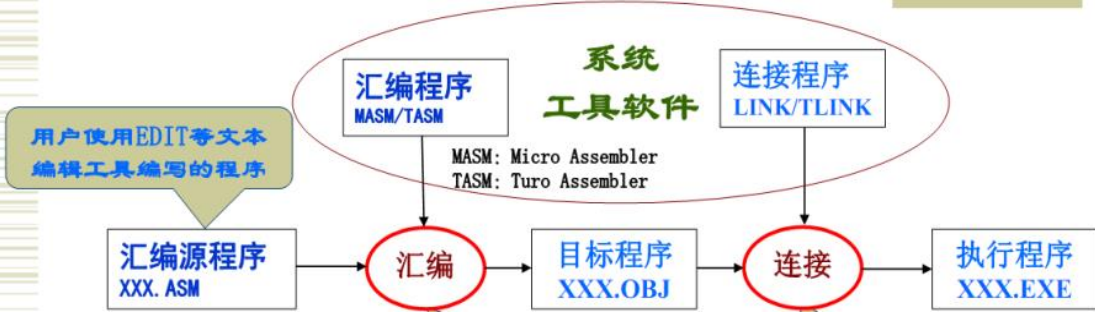
程序员B：源程序模块2

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

数据段覆盖组合定义



例13.3：



程序员A：源程序模块1

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

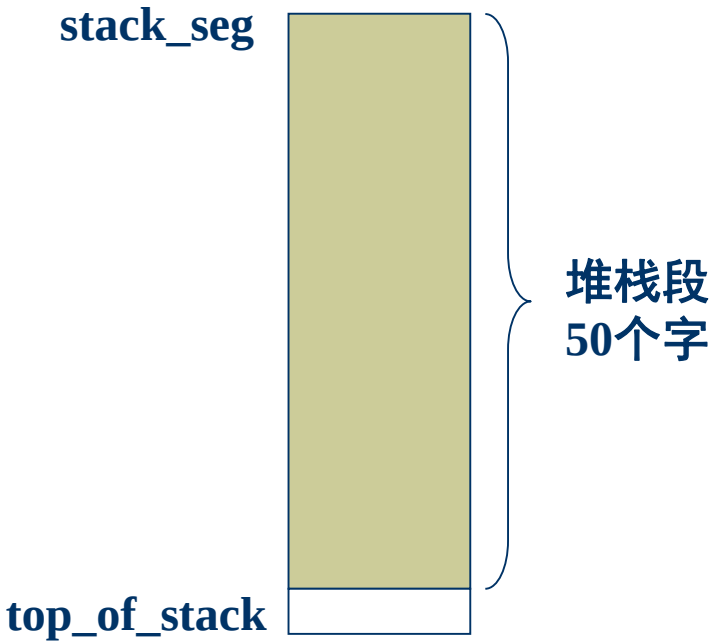
注意：stack_seg段名助记符只是有助于程序阅读的段名，不说明段的任何属性！

程序员B：源程序模块2

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

模块1和模块2的堆栈段组成一段，之间没有留空

连接后的堆栈段存储区分配情况



Segment 伪操作还可以增加存储器分配组合方式类型

格式：

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

注意：s_name只是给段起了个有助于程序阅读的段名，不说明段的任何属性！

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

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

如何建立堆栈段，并初始化栈指针？

例：定义用户堆栈

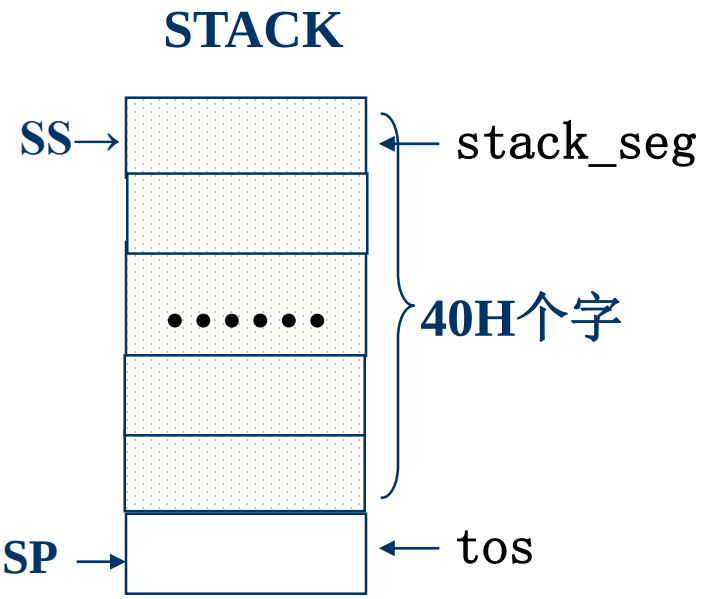
```
stack_seg   segment
            dw   40H   dup   (?)
    tos     label  word
stack_seg   ends
```

缺省：PARA PRIVATE USE16

分配内存时：从小段开始、私有段、16位偏移量寻址方式

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

注意段寄存器设置方式



注意：
data_seg1、 data_seg2、 code_seg只是给段起了个有助于程序阅读的段名，不说明段的任何属性！

例：

```
data_seg1  segment
        x  db  10h
           …
data_seg1  ends          ; 定义数据段
data_seg2  segment
           …
data_seg2  ends          ; 定义附加段

code_seg  segment
        assume  cs:code_seg, ds:data_seg1, es:data_seg2
start:
        mov  ax, data_seg1
        mov  ds, ax
        mov  ax, data_seg2
        mov  es, ax              ; 段基地址→段寄存器
        mov  al, x               ; mov  al, ds:[x]
        …
code_seg  ends
          end start
```

ASSUME(约定)伪指令

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

CS：**CSSEGNAME**

DS：**DSSEGNAME**

ES：**ESSEGNAME**

SS：**SSSEGNAME**

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

使用段寄存器前必须给出约定！

◆应放在引用段寄存器之前，通常放在代码段或主过程的第一个语句位置

◆若一个段寄存器与NOTHING关联，则表示取消前边对该段寄存器的假设

DS:nothing ；取消原来段寄存器DS的预约分配

◆ASSUME语句并不给段寄存器赋值，只是约定，便于汇编程序查错

4.2.3 程序命名和结束伪操作

程序命名：

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

程序结束

END　[执行的起始地址]

```
code_seg  segment
          assume  cs:code_seg,
ds:data_seg1
start:    mov  ax, data_seg1
          mov  ds, ax
          …
          …
code_seg  ends
          end start
```

程序从哪开始执行

装入内存后操作系统如何设置CS/IP值

- **模块命名**
  - **格式：** NAME　　module_name
    - module_name：**模块的名字**
- **模块标题**
  - **格式：** TITLE　text
    - **这样可在列表（LST）文件中，TITLE标题名从第二页开始列出**
    - **text最多60个字符**

- **如果没有NAME伪操作命令，将用text中的前6个字符作为模块名**

- **程序中NAME、TITLE都没有**
  - **用源文件名作为模块名**
  - **但一般经常使用TITLE**

程序结束

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

格式：

END [label]

告诉OS如何初始设置程序的CS,IP

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

过程定义伪指令

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

格式：

过程名　PROC　[类型]

. . .

过程名　ENDP

功能：定义一个过程

- 汇编语言中无论是主程序还是子程序都可以以过程形式出现
  - 一个代码段可以含有多个过程
  - 主过程必须是FAR型
  - 过程名由程序员自己起名
- 类型选项指明该过程的类型，可以是：

  NEAR（或缺省）——说明该过程只能在段内被调用

  FAR ——说明该过程可以在段间被调用，也可以在段内被调用

```
SCODE   SEGMENT
S_NAME  PROC   NEAR/FAR
        …
        RET
S_NAME  ENDP
SCODE   ENDS
```

子程序涉及到RET返回方式，一定要用过程方式定义，并且明确属性

4.2.4 数据定义及存储器分配伪操作

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

- 汇编工具为变量(数据)分配存储单元，并设置初始值或者只预留空间
- Variable:变量名，可有可无，是变量的符号地址，如果指令使用了变量名，表示(指向)第一个字节的偏移地址

```
MOV AL,X
MOV AL,DS:[0000H]
```

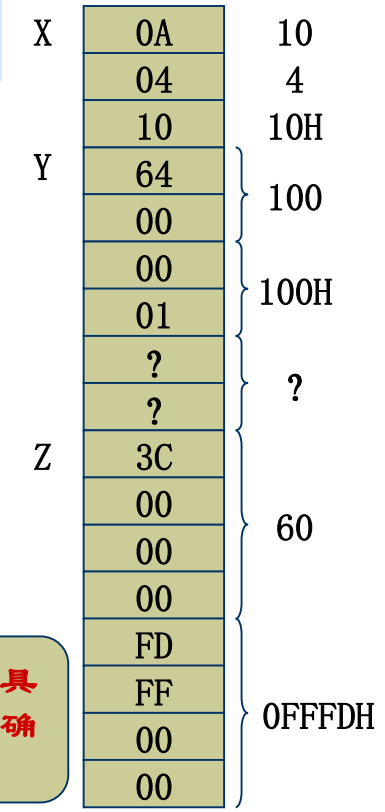
- Mnemonic: 数据类型助记符，是数据类型的符号表示

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

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

每个操作数分配几个存储单元

操作数的具体数值



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

注意：汇编工具汇编时必须能确定具体数值

为数据分配存储单元，并初始化单元内容

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

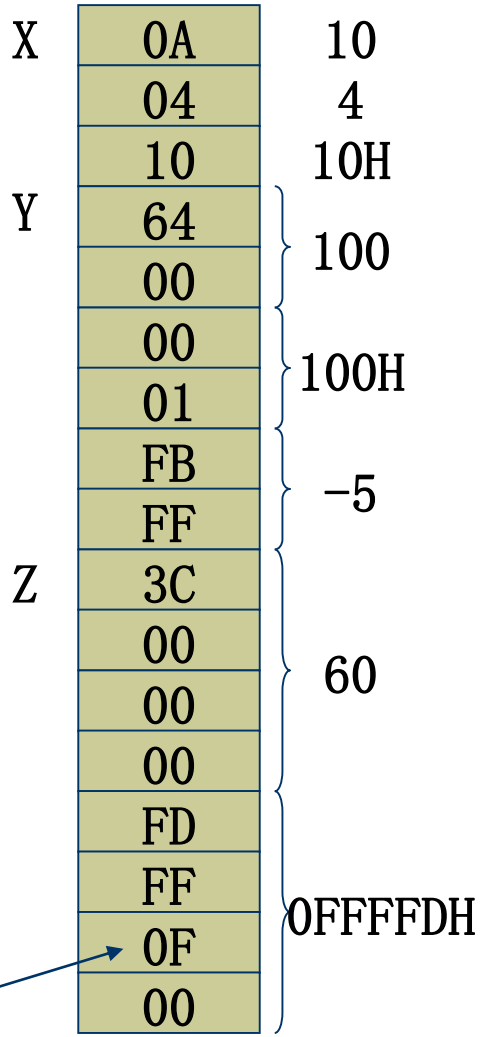
完整段定义

```
MOV  AX,OFFSET Y
        ↓
MOV  AX,0003H
```

```
MOV  AX,Y
        ↓
MOV  AX,DS:[0003H]
```

符号表

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



存储器中实际上以二进制代码00001111存放，一般为了可读性使用十六进制书写

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

简化段定义

变量和标号属性：

变量和标号作用是什么？
在汇编源程序中指明数据和指令的存储单元地址和可处理的方式

所有的变量和标号都有三种属性

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

段基值(SEG)

偏移量(OFFSET)

类型(TYPE)：变量 (字节/字/双字/四字/十字节)

标号 (NEAR / FAR)

例： *BYTE_ARRAY LABEL BYTE*

WORD_ARRAY DW 50 DUP (?)

改变数据的属性

分配50个字数据的存储单元，单元初始值未定


```
MOV AL, BYTE_ARRAY
MOV AL, WORD_ARRAY
MOV AX, WORD_ARRAY
```

这3条指令区别？ 结果、寻址方式

```
mov ax, word_array
mov ax, offset word_array
mov ax, seg word_array
```

对应机器指令

```
mov ax, [5678H]
mov ax, 5678H
mov ax, 1234H
```

变量的类型属性总结：

(1) 指令中使用了变量名，表示的是该组数据的第一个字节的偏移地址

```
MOV AL, XX
MOV AX, YY
```

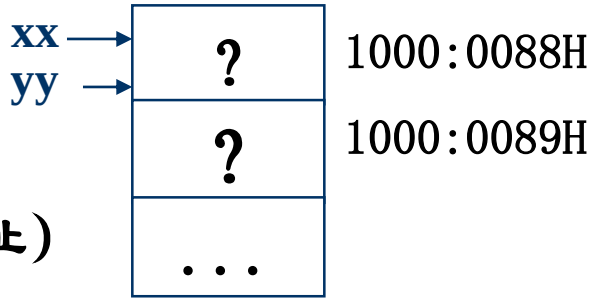
↓

```
MOV AL, DS:[0088H]
MOV AX, DS:[0088H]
```

(2) 该操作中的每一个数据项

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

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



(3) 汇编程序可以用这种隐含的类型属性来确定某些指令是字指令还是字节指令

```
MOV AL, xx  = MOV AL, DS:[xx]
MOV AX, yy  = MOV AX, DS:[yy]
```

(4) 指令中PTR指定的变量类型属性优先于隐含的类型属性

格式：type PTR 变量名

```
mov al, byte ptr yy
```

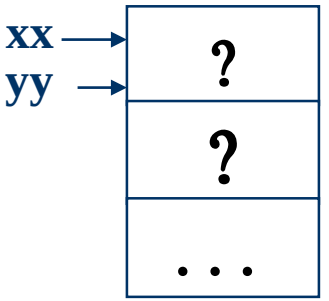
- type可以是：BYTE WORD DWORD FWORD QWORD TBYTE
- 这样可使同一个变量具有不同的类型属性

(5) 变量的另一种属性可以用LABEL伪操作来定义。

格式： name LABEL type

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

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



(6) 操作数字段可以使用复制操作符

repeat_count DUP(operand,...,operand)

```
X   DW   25 DUP(?, 1234H)
```

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

•为数组分配100个存储单元，各单元字节内容：
?,?,34H,12H, ?,?,34H,12H,... ,?,?,34H,12H

•指令中使用BYTE_ARRAY，以字节访问

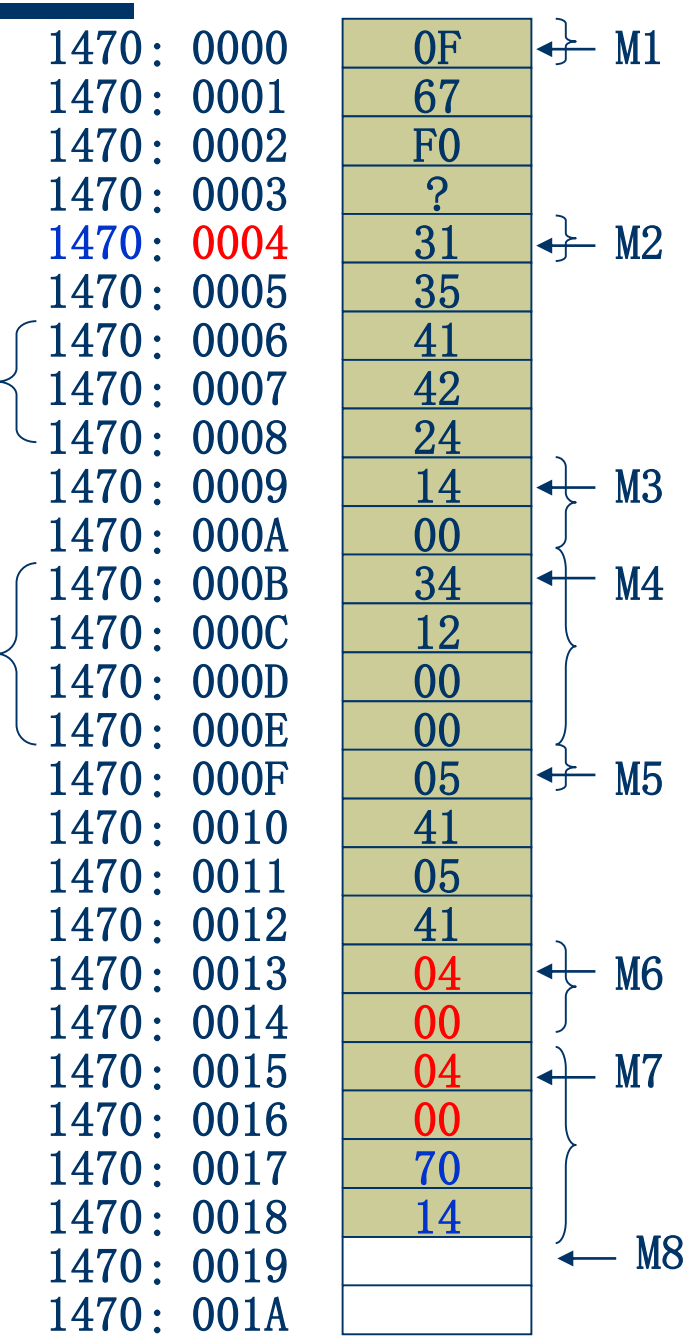
•指令中使用WORD_ARRAY，以字访问

•有两个类型的变量名

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

例.

```
data    segment
        M1  DB  15,67H,11110000B, ?
        M2  DB  '15', 'AB$'
        M3  DW  4*5
        M4  DD  1234H
        M5  DB  2 DUP(5,'A')
        M6  DW  M2      ;M2的偏移量
        M7  DD  M2      ;M2的偏移量、段基址
        M8   LABEL    BYTE
data    ends
```



注意：常数和字符串在存储器中的放置次序！

存储单元14713H可以对应表示为不同的逻辑地址1470:0013,1471:0003等等

第一遍扫描： 检查操作数部分的数据类型是否匹配等，用ASCII值替换字符、用地址替换变量等，并按类型组织数据；

第二遍扫描： 根据变量类型分配存储单元，并初始化为相应数值。

```
M1 DB  '15', 'AB'
M2 DW  '5', 'AB' ;
M3 DW  M2      ;M2的偏移量
M4 DD  M2      ;M2的偏移量、段基址
M5 DB  2 DUP(5,'A')
```

第一遍扫描后

```
M1 DB 31H, 35H, 41H, 42H
M2 DW 0035H, 4142H
M3 DW 0004H
M4 DD 14700004H
M5 DB 2 DUP(5,41H)
```

'AB' 超出不错

'5' 前边补0

'ABC' 超出错

为什么？

汇编工具按程序员意图设计

1470: 0000	31	M1
1470: 0001	35	
1470: 0002	41	
1470: 0003	42	
1470: 0004	35	M2
1470: 0005	00	
1470: 0006	42	
1470: 0007	41	
1470: 0008	24	
1470: 0009	04	M3
1470: 000A	00	
1470: 000B	04	M4
1470: 000C	00	
1470: 000D	70	
1470: 000E	14	
1470: 000F	05	M5
1470: 0010	41	
0011	05	
0012	41	

4.2.5 表达式赋值伪操作EQU

注意： 不分配占用存储单元，只是相当定义了一个常数的数值

格式：

表达式名称 EQU 表达式

```
k EQU 66
mov al,k  ;mov al,66
```

另外有一个与 equ 相类似的“=”赋值伪操作

格式：

表达式名 = 表达式

注意：表达式值汇编时一定要能确定具体数值！

- 区别：EQU伪操作中的表达式名是不允许重复定义的，而“=”伪操作中的表达式名则允许重复定义

```
k=1                 k EQU 1
k=k+5  允许         k EQU k+5  不允许
```

MOV AL, k 汇编后等同 MOV AL, 6

“=”可以看作“表达式”定义
“EQU”可以看作“等同”定义

- 表达式中的变量名是指变量的数值 BETA EQU ALPHA-2
- 字符串、变址引用都可以赋以符号名 B EQU [BP+8]

MOV AL, B 汇编后等同 MOV AL, [BP+8]

4.2.6 地址计数器对准伪操作

汇编程序汇编时最主要任务有两个：

为数据/机器指令分配存储单元；

数据初始化、汇编指令翻译机器指令

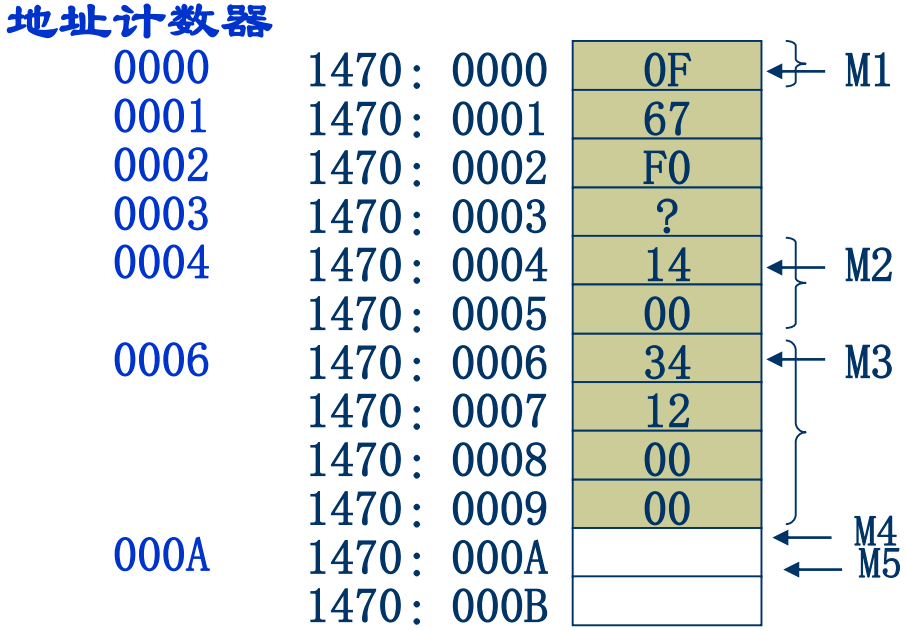
偏移地址由地址计数器确定

例.

```
data    segment
        M1 DB 15,67H,11110000B, ?
        M2 DW 4*5
        M3 DD 1234H
        M4 LABEL BYTE
        M5 DB 10
data    ends
```

符号表

变量名	属性	段基地址	偏移地址
M1	DB	----R	0000H
M2	DW	----R	0004H
M3	DD	----R	0006H
M4	DB	----R	000AH



4.2.6 地址计数器对准伪操作

- 汇编源程序编写时地址偏移量计算很麻烦，如何让汇编程序帮我们计算？ 简化程序可编程性、易读性？

1、地址计数器

- 在汇编程序对源程序汇编过程中，使用地址计数器保存当前正在汇编的基本处理单位在存储器中的首地址（如指令的首地址，操作数首地址）
- 指令中地址计数器的值用‘$’来表示，汇编语言允许用户直接用‘$’来引用地址计数器的值

```
M2 DW 10, $*5
```

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

地址计数器

- 又称单元计数器或位置计数器
  - 内容为指令或数据首字节存储单元的偏移地址（相对于本段开始位置的相对地址）
- 啥时将地址计数器初始化为零
  - 开始汇编
  - 每一段开始
- 地址计数器修改
  - 汇编程序扫描源文件时，每处理一条指令，根据指令所需的字节数就增加一个值
  - 根据伪操作指令ORG xx设置为某个值

```
data_seg1  segment
    x  db  10h
        …
data_seg1  ends
```

汇编工具第一遍扫描时，地址计数器的值用来确定每条指令的第一个字节的偏移地址和数据段中变量名的值（数据的第一个字节偏移地址）

扫描时确定偏移地址

地址计数器的值

```
        .model   small                  (location      length
; ********************************      counter)       (bytes)
        .data                             0              0
        org      10                       0              10
num     db       -29                      0ah(10)        1
array   db       100 dup(?)               0bh(11)        100
count   dw       5                        6fh(111)       2
masks   db       82h,04h,2ah              71h(113)       3
; ********************************
        .code                             0              0
main    proc     far                      0              0
start:                                    0              0
        mov      ax,@data                 0              3
        mov      ds,ax                    03h(3)         2
```

汇编工具扫描时一定要能确定指令长度 JMP SHORT OPR

JMP NEAR PTR OPR

例1： cs:0074 MOV AX, $+6

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

cs:0074 MOV AX, 007AH

地址计数器的值

例2： ARRAY DW 1,2,$+4,3,4,$+4

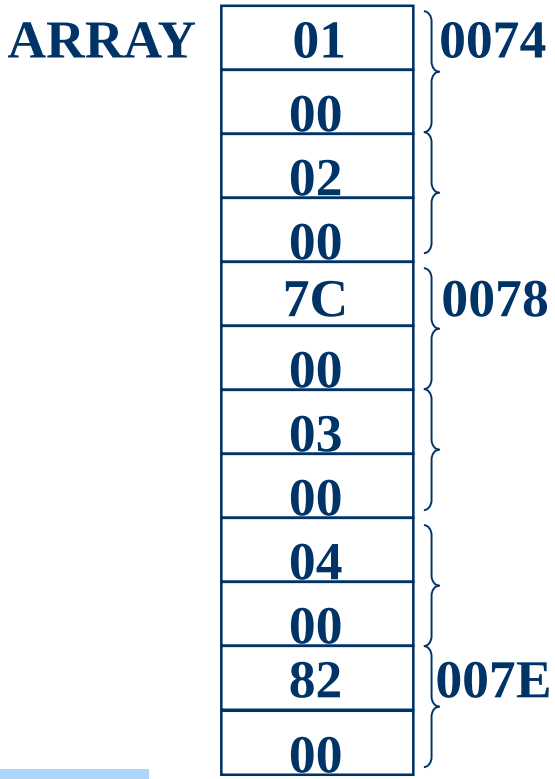
假设该数组的首地址0074H

① 汇编程序处理第一个$+4时，地址计数器=0078H

因此，$+4=0078+4=007CH

② 汇编程序处理第二个$+4时，地址计数器=007EH

因此，$+4=007E+4=0082H



地址计数器值＝当前指令/操作数的首地址；

地址计数器的值汇编程序用$表示

2、ORG 伪操作

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

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

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

3、EVEN 伪操作

地址计数器值=2的倍数

- 格式：　EVEN
- 功能：使下一个变量或指令地址开始于偶字节地址

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

4、ALIGN 伪操作

ALIGN 2 = EVEN

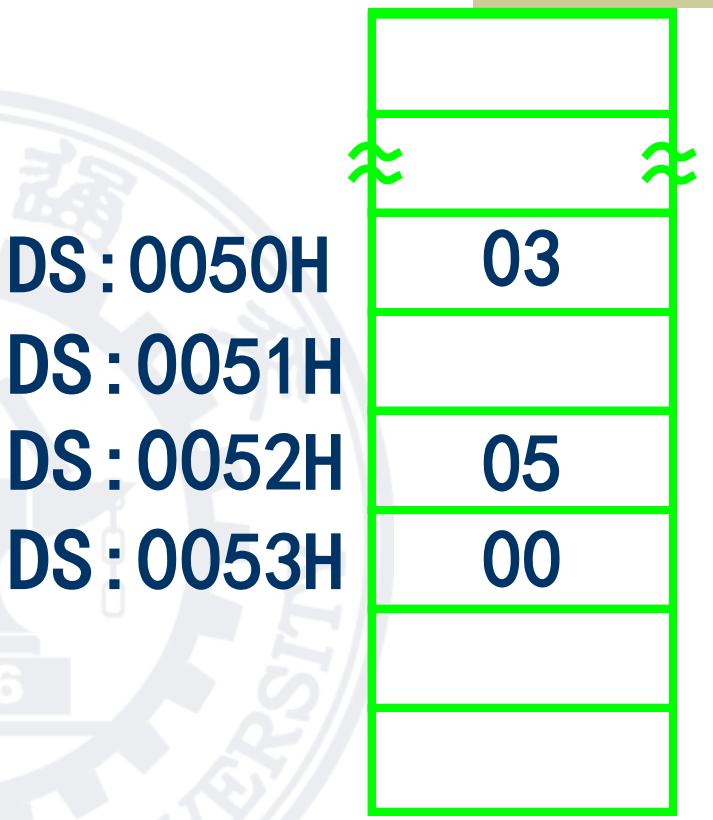
- 格式：ALIGN　boundary
  - 其中boundary必须是2^n
- 保证双字数组边界从4的倍数地址存储单元开始
  - ALIGN 4
- 例子：

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

ARRAY的地址偏移量为4的倍数

例.

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



EVEN语句的效果

4.2.7 基数控制伪操作

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

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

RADIX 伪操作

格式：RADIX 表达式（n） ；表达式表示基数值，用十进制数表示

功能：用于改变汇编语言默认的基数（范围为：2~16）

例如：

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

如果有二义性，必须加后缀

如果199D是十六进制，后边必须带H

```
MOV BX, 199DH
```

4.3 汇编语言指令格式

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

带〔 〕的两项是可缺省的

表达式：数据表达式，地址表达式

4.3 汇编语言指令格式

〔名字项〕 操作项 操作数项 〔；注释项〕

- **机器指令语句格式：**

  [标号：] 操作项 [操作数1 [，操作数2]] [；注释]

  ```
  start:  mov  si, offset next
  ```

- **伪指令语句格式：**

  [变量] 伪操作项 [操作数1 [，操作数2]] [；注释]

  ```
  X DB 10,4,10H
  ```

  ```
  code segment
  ```

注意：

- 程序中使用的是半角英文字符和标点符号，不能用中文或全角英文字符和标点符号
- 注释中可以用中文

```
X DB 10,4，10H
```
哪里错了？

〔名字项〕 操作项 操作数项 〔；注释项〕

4.3.1 名字项

- 名字项可以是指令标号或伪操作的变量、过程名、段名

```
NEXT:     MOV  AX,  DATA
A         DB  'morning'
SCODE     SEGMENT
S_NAME    PROC  NEAR/FAR
```

只有标号加“：”

- 由1-26个字符组成，数字不能出现在名字的第一个字符位置

〔名字项〕　操作项　操作数项　〔；注释项〕

1、标号

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

  next: mov ax, data_seg1

- 标号经常在转移指令或CALL指令的操作数字段出现，用以表示转向地址

〔名字项〕 操作项 操作数项 〔；注释项〕

2、变量

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

  (1) 变量在数据段或附加段中定义，后面不跟冒号

  (2) 用LABEL、DB、DW、DD伪操作定义变量属性

  A DB ‘morning’
- 变量经常在操作数字段出现 MOV AL, A

4.3.2 操作项

- 操作项：给出操作的符号表示，可以是机器指令、伪指令、宏指令的操作功能助记符
  - 对于机器指令：汇编程序将其翻译为机器语言操作码

    mov ax, data_seg1
  - 对于伪指令：汇编程序将根据其要求的功能进行处理

    A DB ‘morning’
  - 对于宏指令：汇编程序将根据宏定义展开

〔名字项〕 操作项 操作数项 〔；注释项〕

4.3.3 操作数项

- 操作数项：由一个或多个表达式组成，提供操作项操作所需要的数据信息
- 操作数项可以是常数（立即数）/寄存器/存储器地址/标号/变量/表达式

表达式值汇编时要保证能计算出具体值，否则汇编工具无法处理

- 每条指令语句的操作数个数已由系统确定
  - 例如加法指令有两个操作数

表达式

〔名字项〕　操作项　操作数项　〔；注释项〕

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

```
ARRAY DW 1,2,$+4
MOV AX, $+6
```

表达式的值在汇编时一定要能够确定！

操作数有关的常用操作符

1、算术操作符

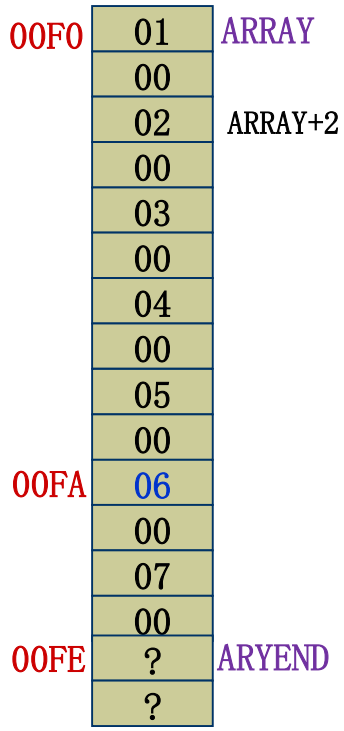
- 算术操作符有：+、 -、 *、 /和MOD

其中MOD是指除法运算后得到的余数

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

        MOV DX,ARRAY+(6-1)*2     ; 00F0+(6-1)*2=00FA
        ;汇编后, MOV DX,[00FA]
        ;执行后, 0006H→DX
        MOV CX,(ARYEND-ARRAY)/2 ;计算数组长度
        ;汇编后, MOV CX,7
```

汇编时将ARRAY认为地址偏移量来计算



与第3章指令操作符区别?

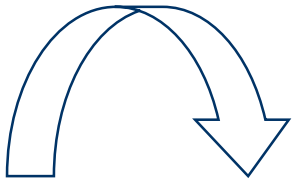
2、逻辑与移位操作符

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

例： OPR1 EQU 25 ；00011001

OPR2 EQU 7 ；00000111

汇编后等同



00011001
00000111
00000001=01H

AND AX, OPR1 AND OPR2 AND AX, 0001H

3、关系操作符

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

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

```
        X: ......
           ......
        Y: ......
```

若≤128(真)　　汇编后　　MOV FID, 0FFFFH

若>128(假)　　汇编后　　MOV FID, 0

4、数值回送操作符

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

OFFSET / SEG 变量（或标号）

功能：回送变量或标号的偏址 / 段址

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

TYPE 变量（或标号）

变量：DB DW DD DQ DT　　标号：NEAR FAR

值：　1　2　4　8　10　　-1　-2

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

LENGTH 变量

功能：回送由DUP定义的变量的数据个数，其它情况回送1

SIZE 变量

功能：LENGTH*TYPE

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

5、属性操作符

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

  - PTR操作符

    格式：类型 PTR 地址表达式

    功能：指定地址表达式的类型

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

  - THIS 类型操作符

    格式：THIS 类型

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

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

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

  - SHORT 标号操作符

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

```
JMP  SHORT  NEXT
```

  - HIGH、LOW 字节分离操作符

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

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

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

〔名字项〕 操作项 操作数项 〔；注释项〕

4.3.4 注释项

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

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

4.4 汇编语言程序的上机过程

4.4.1 建立汇编语言的工作环境

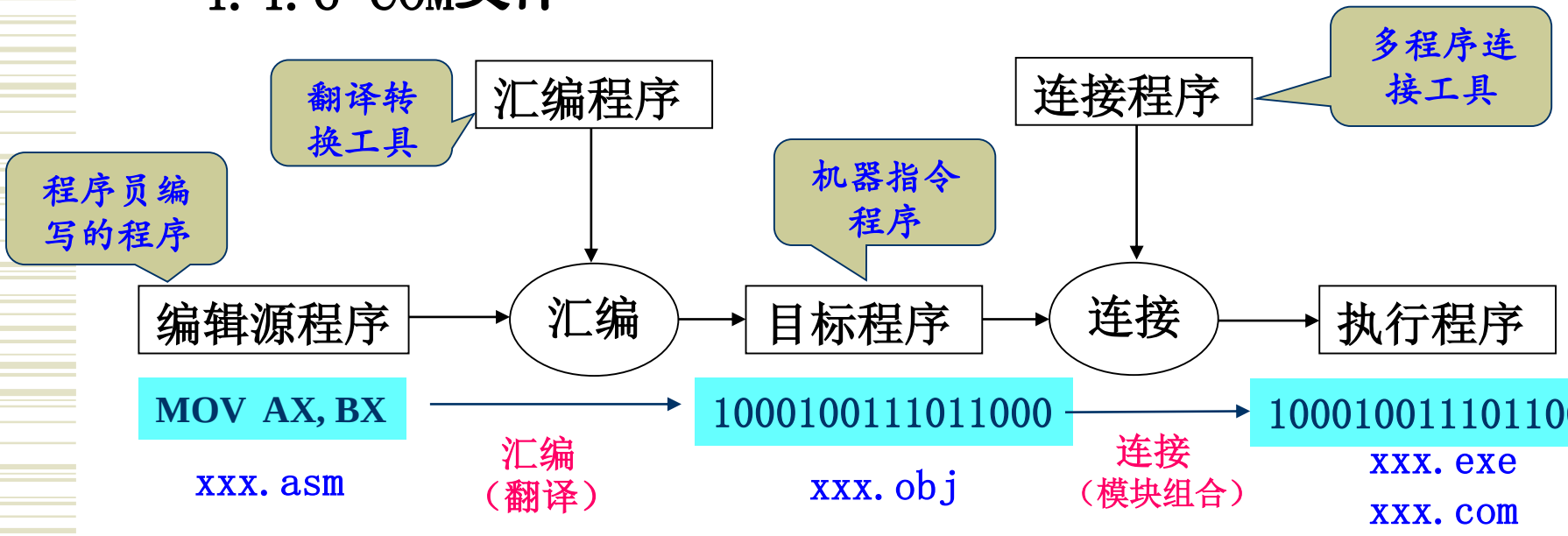
4.4.2 建立ASM文件

4.4.3 用MASM(或TASM)程序产生OBJ文件

4.4.4 用LINK(或TLINK)程序产生EXE文件

4.4.5 程序的运行

4.4.6 COM文件



进一步说明的问题

1、“向前引用”、“向后引用”

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

JMP NEAR PTR EXIT

主要看汇编程序扫描到这里时能否确定变量或标号的属性

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

```
     JMP   NEAR PTR EXIT
     .......
EXIT:
```

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

2、“浮动”地址概念

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

```
006F  0005                          count   dw      5
0071  82  04  2A                    masks   db      82h,04h,2ah
                        ; ***** ******************************
0000                                .code
0000                    main        proc    far
0000                    start:
0000  B8  ---- R                    mov     ax,@data
0003  8E  D8                        mov     ds,ax
                        ;
0005  8B  0E  006F  R               mov     cx,count
0009                    repeat:
0009  49                            dec     cx
```

装入后确定

连接段组合后确定，可能段合并

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

DOS装入.EXE文件的过程

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

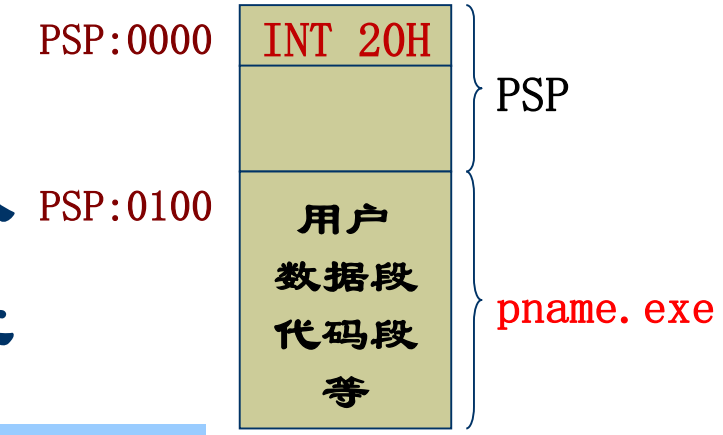
② 把文件头读入内存

③ 计算可执行模块的大小

④ 计算装入的起始段地址

⑤ 完成重定位

到这一步.EXE文件被装入内存



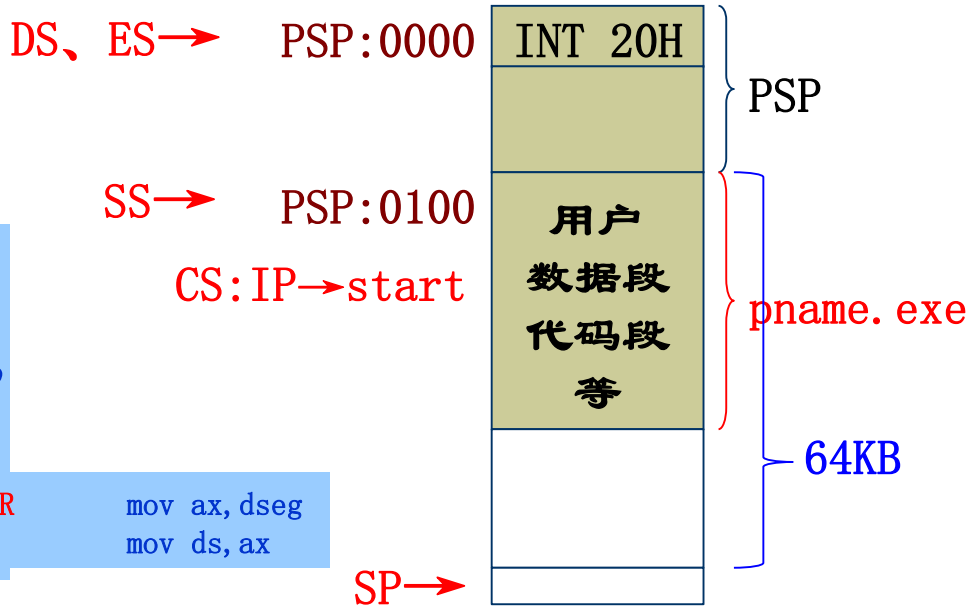

```
0004   B8 ---- R        mov ax,dseg
0007   8E D8             mov ds,ax
```

```
code_seg  segment
          assume  cs:code_seg,
ds:data_seg1
start:    mov  ax,  data_seg1
          …
code_seg  ends
          end start
```

⑥ 初始化段寄存器和指针寄存器(OS设置)

- SS=PSP+0100，SP=0000指向栈顶
- 其他段寄存器全部被初始化为指向PSP的段基址，以便用户能够访问PSP中的信息
- 装入程序对段和指针寄存器设置为：CS:IP为主程序的入口地址(程序装入后执行的第一条指令地址)

⑦ 把控制权交给.EXE程序，开始执行用户程序

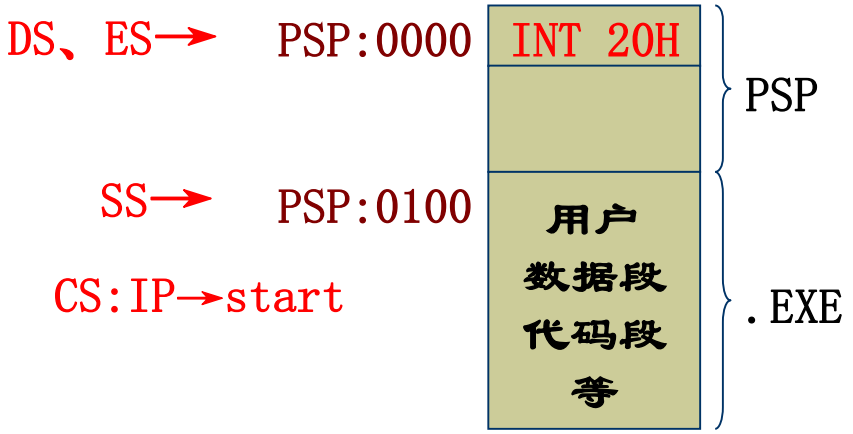


注意：用户程序中必须根据需要重新设置DS\ES\SS、SP，指向自己的数据段等，如：

```
mov  ax, dseg
mov  ds, ax
mov  ax, x  ;DS:[x]
```

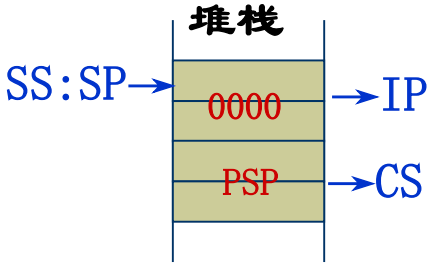
```
0004  B8 ---- R      mov  ax, dseg
0007  8E D8          mov  ds, ax
```

如何返回DOS



方法1、

```
MAIN   PROC   FAR
       ASSUME  ...
       先设置栈指针或使用默认栈
       PUSH   DS          ;PSP段基址入栈
       XOR    AX,AX       ;清0
       PUSH   AX          ;数字0入栈
       ...                ;完成程序指定功能
       RET                ;PSP:0 送 CS:IP
```



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

如果使用自定义堆栈，栈指针一定要在开始设置好！

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

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

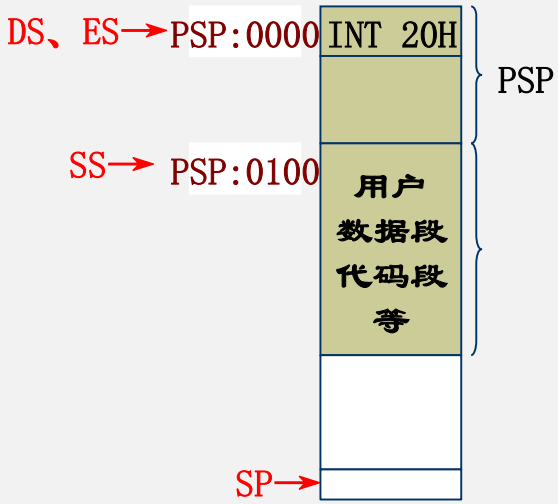
~~MOV AH, 4CH~~

无法保证AL=00

```
; 定义数据段
data_seg1   segment
      X1    DB 10,4,10H
      Y1    DW 100,100H,-5
      Z1    DD 3*20,0FFFDH
data_seg1   ends;
; 定义附加段
data_seg2   segment
      X2    DB 10,4,10H
      Y2    DW 100,100H,-5
      Z2    DD 3*20,0FFFDH
data_seg2   ends
;定义堆栈段
stack_seg   segment
            dw  40H  dup  (?)
     tos     label  word
stack_seg   ends
;
```

```
code_seg   segment
           assume  cs:code_seg, ds:data_seg1, es:data_seg2,
ss:stack_seg
p-name     proc   far            ; (或者near, 主程序最好使用far)
;
           cli                   ; 关中断
           mov    ax, stack_seg  ; 设置堆栈指针
           mov    ss, ax
           mov    sp, offset  tos
           sti                   ; 开中断
;
           push   ds
           xor    ax, ax
           push   ax
;
           mov    ax, data_seg1  ; 设置数据段段寄存器
           mov    ds, ax
;
           mov    ax, data_seg2  ; 设置附加数据段段寄存器
           mov    es, ax
;
           ......   ;在这里写主程序代码
           ......
;
exit-dos:  ret
;
p-name     endp
;
sp-name    proc   near; 子程序开始
;
           ......   ;在这里写子程序代码
           ......
           ret         ; 子程序返回
sp-name    endp
;
code_seg   ends
           end p-name
```

栈指针设置应该放在程序开始！

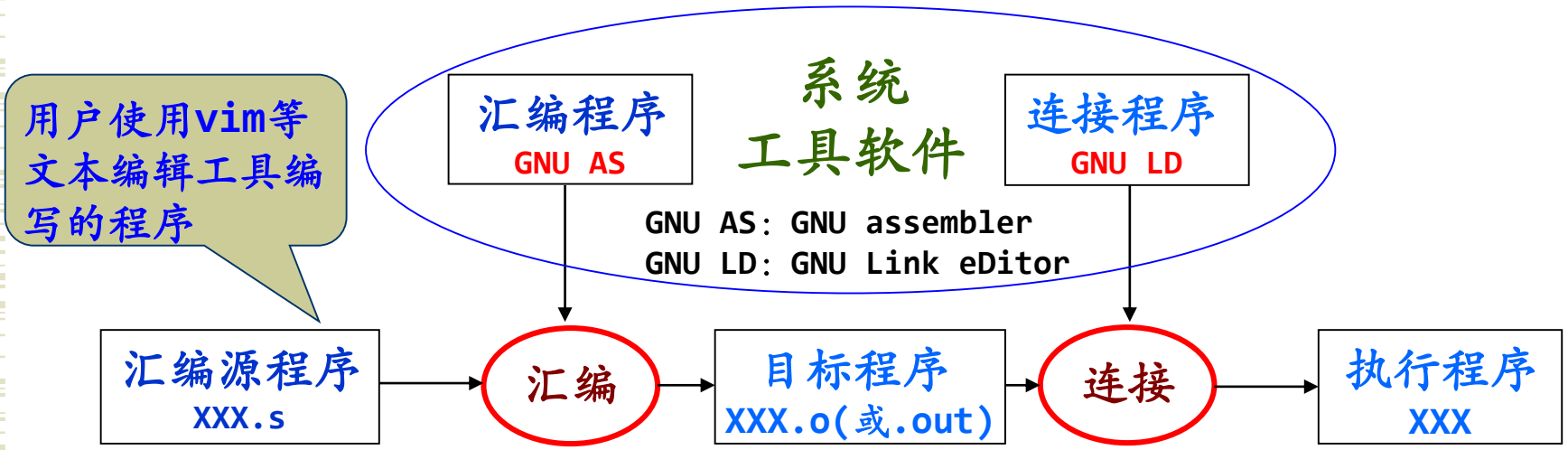


```
; 定义数据段
data_seg1     segment
       X1     DB 10,4,10H
       Y1     DW 100,100H,-5
       Z1     DD 3*20,0FFFDH
data_seg1     ends;
; 定义附加段
data_seg2     segment
       X2     DB 10,4,10H
       Y2     DW 100,100H,-5
       Z2     DD 3*20,0FFFDH
data_seg2     ends
;定义堆栈段
stack_seg     segment
              dw  40H  dup  (?)
     tos      label  word
stack_seg     ends
;
```

```
code_seg   segment
           assume  cs:code_seg, ds:data_seg1, es:data_seg2, ss:stack_seg
p-name     proc    far            ; (或者near，主程序最好使用far)
;
           mov     ax, data_seg1  ; 设置数据段段寄存器
           mov     ds, ax
;
           mov     ax, data_seg2  ; 设置附加数据段段寄存器
           mov     es, ax
;
           cli                    ; 关中断
           mov     ax, stack_seg  ; 设置堆栈指针
           mov     ss, ax
           mov     sp, offset  tos
           sti                    ; 开中断
;
           ......       ;在这里写主程序代码
           ......
;
exit-dos:  mov     ax, 4c00h      ; 另外一种返回DOS操作系统方法
           int     21h
p-name     endp
;
sp-name    proc    near           ; 子程序开始
;
           ......       ;在这里写子程序代码
           ......
           ret                    ; 子程序返回
sp-name    endp
;
code_seg   ends
           end p-name
```

4.4 ARM64伪操作

- ARM64汇编环境：GNU as
  - GNU as 可以运行在不同处理器架构的机器上
  - 在本课程中，使用GNU as在ARM –A处理器架构的机器上运行汇编程序。
  - 在ARM编程中使用AT&T语法,例如GNU汇编器



4.4.1 ARM64段定义

- 格式:

  .section name

  ...

  .section next_name

  ...

- 在GNU中，所有的伪指令都是由“.”开头。
  - 自定义段操作：**.section name** 表明开始一个段，从此定义语句以后的所有汇编语句都属于这个**section name**段，直到定义下一个段，或者使用了另一个系统预定义段名，或者到文件结束。
  - 系统预定义段名：**.text**、**.data**、**.bss**，分别表示代码段、初始化数据段、未初始化数据段。可以直接使用。

4.4.1 ARM64段定义：举例

- 段结束的三种方式

① data段的内容从.data开始，到自定义段命令.section my_section段结束；

② .data段的内容从.data伪指令开始，到系统另一个预定义的段名.text结束

③ data段的内容从定义.data开始，到文件结束

①

```
.data
        data段内容
.section  my_section
        自定义内容
```

②

```
.data
        data段内容
.text
        代码段内容
```

③

```
.data
        data段内容
文件结束
```

4.4.2 ARM64程序命名

- 标题格式:
  - .title text
  - 这样可在列表（LST）文件中，TITLE的标题名会在每一页文件名和页序号行的后一行列出
- 举例：
  - 文件名称为title.s文件，用.title设置title为“tryToUseTitle”，在list文件中显示如图所示。

```
AARCH64 GAS  title.s                    page 1
tryToUseTitle


   1                            .title "tryToUseTitle"
   2                            .data
   3 0000 01000200      ARRAY:  .2byte  1,2,3,4,5,6,7
   3      03000400
   3      05000600
   3      0700
   4 000e 0100          ARYEND: .2byte  1
   5 0010 400180D2              mov X0, ARRAY + (6-1)*2
```

4.4.3 ARM64过程定义

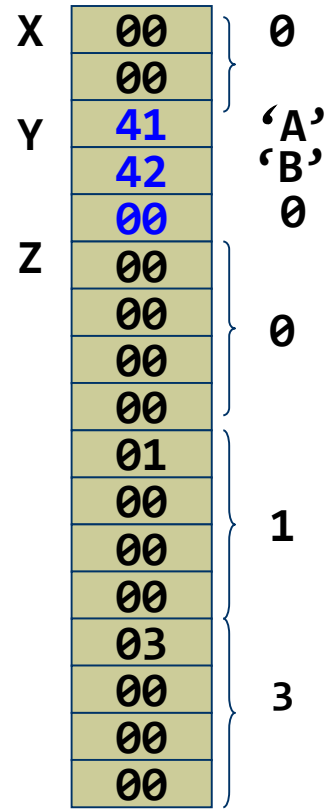
```
        .text
        .type main, %function
        .global main
main:
        stp x29, x30, [sp,#-16]!
        …
        ret
```

- 格式：
  - .type 函数名, %function
  - 功能：定义一个函数
  - %function说明类型是函数
- 汇编语言中无论是主程序还是子程序都可以以函数（过程）形式出现
  - 一个代码段可以含有多个函数
  - 主程序和子程序结构一样，无特殊要求
  - 函数（过程）名由用户自己起
- 作用范围说明符指明该函数的作用范围，可以是：
  - 缺省 ——说明该函数只能在本文件中被调用
  - .global ——说明该函数可以被任意一个与此文件连接的文件调用

4.4.4 ARM64数据定义

- 格式： 变量名: .数据类型, [表达式] [//注释]
- 汇编工具为变量(数据)分配存储单元，并设置初始值
  - 变量名：可有可无，是变量的符号地址，如果指令使用了变量名，表示(指向)第一个字节的偏移地址
  - 数据类型：是数据类型的助记符
    - 字节定义伪指令 byte
    - 半字定义伪指令 2byte/hword/short
    - 字定义伪指令 4byte/word/long
    - 8字节定义伪指令 8byte/quad
  - 表达式：表达式可以是简单整数，也可以是c样式的表达式；
  - 注释：以“//”开始，代表之后都是注释

```
X:      .short  0
Y:      .byte   ‘A’,‘B’,0
Z:      .word   0, 1, 3
```



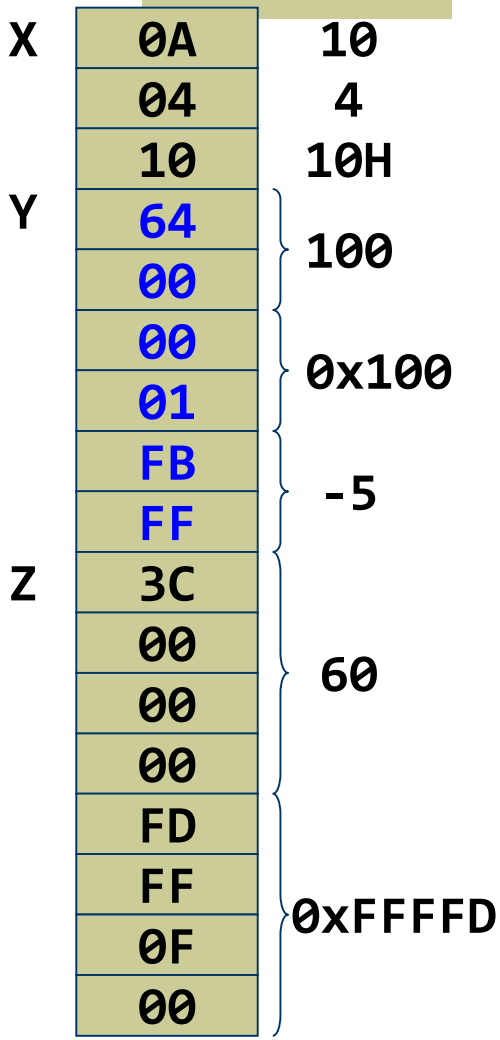
4.4.4 ARM64数据定义

- 为数据分配存储单元，并初始化单元内容

```
.data                    ARM64写法
X:      .byte   10,4,0x10
Y:      .hword  100,0x100,-5
Z:      .word    3*20,0xFFFFD
```

```
AARCH64 GAS  assignfordata.s                    page 1


   1                            .data
   2 0000 0A0410        X:      .byte  10, 4, 0X10
   3 0003 64000001      Y:      .hword 100, 0X100, -5
   3      FBFF
   4 0009 3C000000      Z:      .word  3*20, 0XFFFFD
   4      FDFF0F00
~
```



4.4.5 ARM64表达式赋值

- 格式：
  - .equ{v} 表达式名称，表达式
  - equv比equ多了一个功能，就是当表达式名称已经被定义过的时候汇编器会报错

```
.equ arysize, 10

MOV x1, arysize
; 汇编后等同 MOV x1, 10
```

- 在arm64中也是不分配存储单元，只是相当定义了一个常数的数值

注意：表达式值汇编时一定要能确定具体数值！

•使用表达式赋值伪操作的好处：提高可读性；常量值不需要在每个用到的地方修改，只需要更改表达式的赋值即可。

4.4.6 ARM64地址计数器

- 位置计数器 ‘.’
  - 在ARM64中使用符号点 ‘.’来表示当前位置计数器的值的引用，含义与x86中地址计数器相同
- 何时将地址计数器初始化为零？
  - 与x86汇编相同，在一个新段开始时
- 地址计数器修改
  - 汇编程序扫描源文件时，每处理一条指令，根据指令所需的字节数就增加一个值
  - Arm程序中经常在段的末尾有伪指令

    .size myfunc, (. – myfunc)

    - 用位置计数器 ‘.’ 减去myfunc标签的值，表示此段占用了多少字节的空间。这个伪指令用于给连接器、调试器提供信息。

4.4.6 ARM64地址计数器

- 在用GNU汇编ARM64汇编程序的时候GNU仅进行一遍扫描。地址计数器的值用来确定每条指令的第一个字节的偏移地址和数据段中变量名的值（数据的第一个字节偏移地址）

```
ARM GAS  variable2.S                    page 1

line addr value          code
   1                              .data
   2 0000 00000000       i:      .word   0
   3 0004 01000000       j:      .word   1
   4 0008 48656C6C       fmt:    .asciz  "Hello\n"
   4      6F0A00
   5 000f 414200         ch:     .byte   'A','B',0
   6 0012 0000                    .align  2
   7 0014 00000000       ary:    .word   0,1,2,3,4
   7      01000000
   7      02000000
   7      03000000
   7      04000000
```

地址计数器的值

扫描时确定偏移地址

4.4.6 ARM64地址计数器

- ARM中更改地址计数器伪操作
- 在ARM64中更改地址计数器的伪操作更为简单。
- 格式如下：
  - .= 常数表达式
- 功能：汇编时，使下一个操作数、指令等分配的存储器单元地址是常数表达式的值n
- 例如：

```
.data
. = . + 4                    // 4=0004H
i:    .word      0           // 偏移地址为0004H
. = 20                       // 20=0014H
J:    .word      1           // 偏移地址为0014H
```

4.4.7 ARM64边界对齐

- .align伪操作
  - 格式：.align abs-sxpr
  - 功能：使下一个变量或指令地址低位处零位个数，对齐过程中填充的数字

```
i:      .byte       0
.align 1
J:      .short 1
K:      .byte       2
```

要求地址在转化成二进制时最后1为是0，即地址是偶数，是2的倍数

.align 1 等价于x86中的EVEN

- .balign伪操作(arm)
  - 格式：.balign abs-expr
  - 其中abs-expr必须是2^n
  - 保证字数组边界从4的倍数地址存储单元开始

    .balign 4

4.5 ARM64汇编语言程序格式

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

- 带[ ]的两项是可缺省的
- 表达式：数字表达式，地址表达式

4.5.1 ARM64指令格式

- 机器指令语句格式：
  - [标号:] 操作项 [源操作数 [,目的操作数]][//注释]

start: mov X0, X1

- 伪指令语句格式：
  - [标号:] 伪操作项 [操作数1 [,操作数2]] [//注释]

x: .word 10, 4 //用字大小的空间保存两个数10和4

注意：程序中使用的是半角英文字符和标点符号
不能用中文或全角英文字符和标点符号
注释中可以用中文

4.5.2 ARM64标号和变量

1、标号

- 是用符号表示的指令地址，也叫符号地址
- 作用范围可以是缺省或者全局global

- 标号的定义：直接在指令助记符前加上标识符，必须以冒号“:”结束，如NEXT:

```
next: mov  X0, X1
```

  - Arm64中由于标号既可以作为子程序名，又可以作为变量名，因此可以在.data段和.text段出现

4.5.2 ARM64标号和变量

2、变量

- 是数据项的第一个字节相对应的标识符，在ARM中用标号方式来实现，是标号的用法之一。
- 变量有1个属性
- 作用范围:变量可以通过.global 变量名来使得变量是全局变量。

- 变量类型属性：定义该变量所保留的字节数，
  - byte（1个字节长），hword（2个字节长），word（4个字节长）…(与x86不同，ARM64一个字是4字节)
- 变量的定义：

（1）变量在数据段或附加段中定义，后面跟冒号

（2）用.byte，.short，.word等伪操作定义变量属性

A: .byte 10

4.5.3 ARM64与操作数有关的操作符

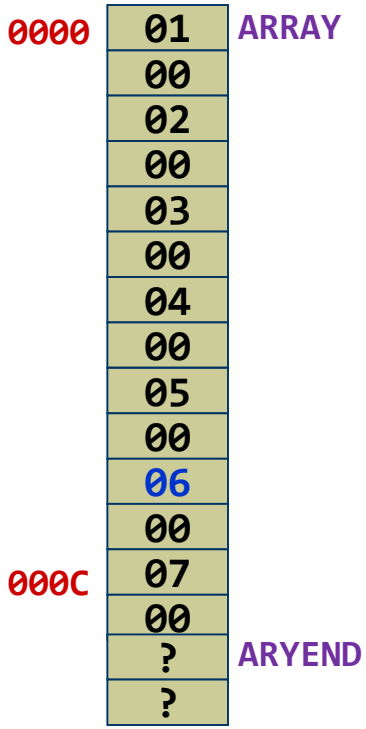
- 算术操作符
  - +、 -、 *、 /

```
ARRAY:   .short  1, 2, 3, 4, 5, 6, 7
ARYEND:

        MOV X0, ARRAY+(7-1)*2
        ;汇编后，MOV X0, #12
        ;执行后，12→X0
        MOV X1, (ARYEND-ARRAY)/2 ;计算数组长度
        ;汇编后，MOV X1, 7
```

MOV不是内存访问，按立即数处理

汇编时将ARRAY认为地址偏移量来计算



4.5.4 ARM64注释

- 在gnu中，不同的芯片架构使用不同的指令集，因此注释符号也有所不同。
- 通用的符号是/* */，在arm64中，还可以使用//作为行注释符号

```
.data
    mesg: .asciz "Hello World\n" // Define null-terminated
string
.text      // Text section
.global main
/*
* Prints "Hello World\n" and returns 0.
*/
main: stp x29, x30, [sp, #-16]!
    adr x0, mesg
    bl   printf
```

谢谢！