

# 第四章 汇编语言程序格式

4.1 汇编程序功能

4.2 伪操作

4.3 汇编语言程序格式

4.4 汇编语言程序上机过程

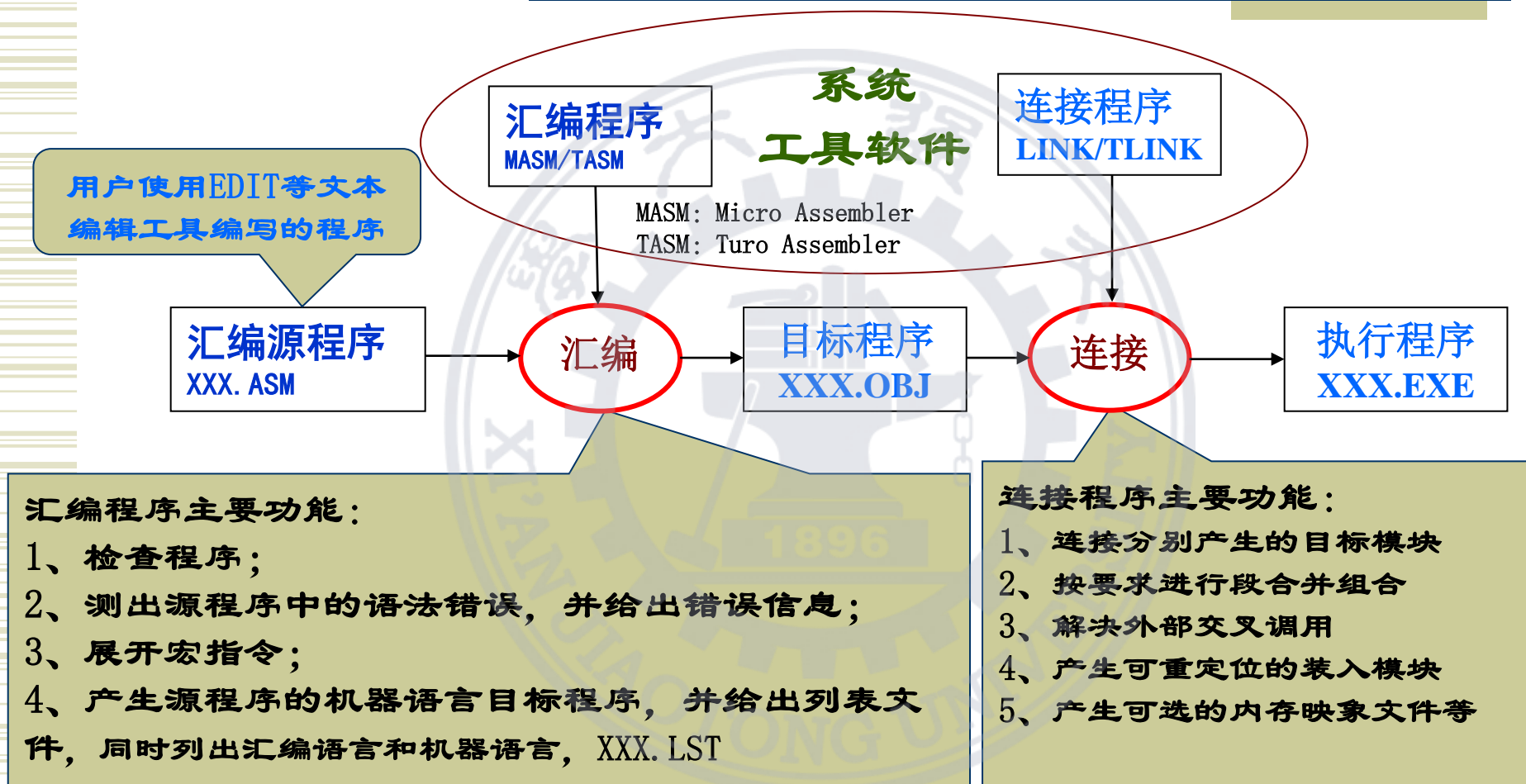
4.5 ARM64伪操作

4.6 ARM64汇编语言程序格式

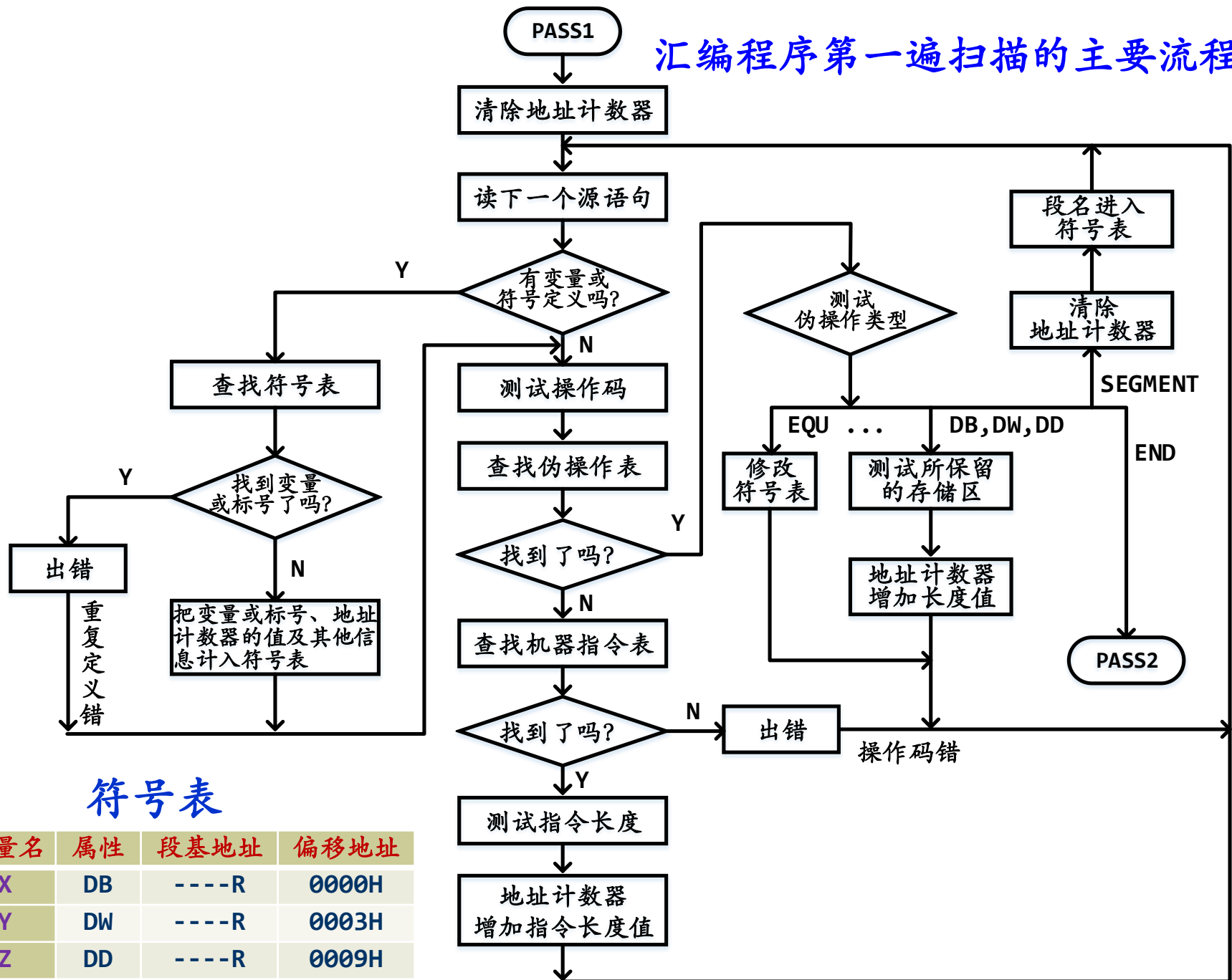
# 本章目标

- ❑ 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



# 汇编程序第一遍扫描的主要流程



## 符号表

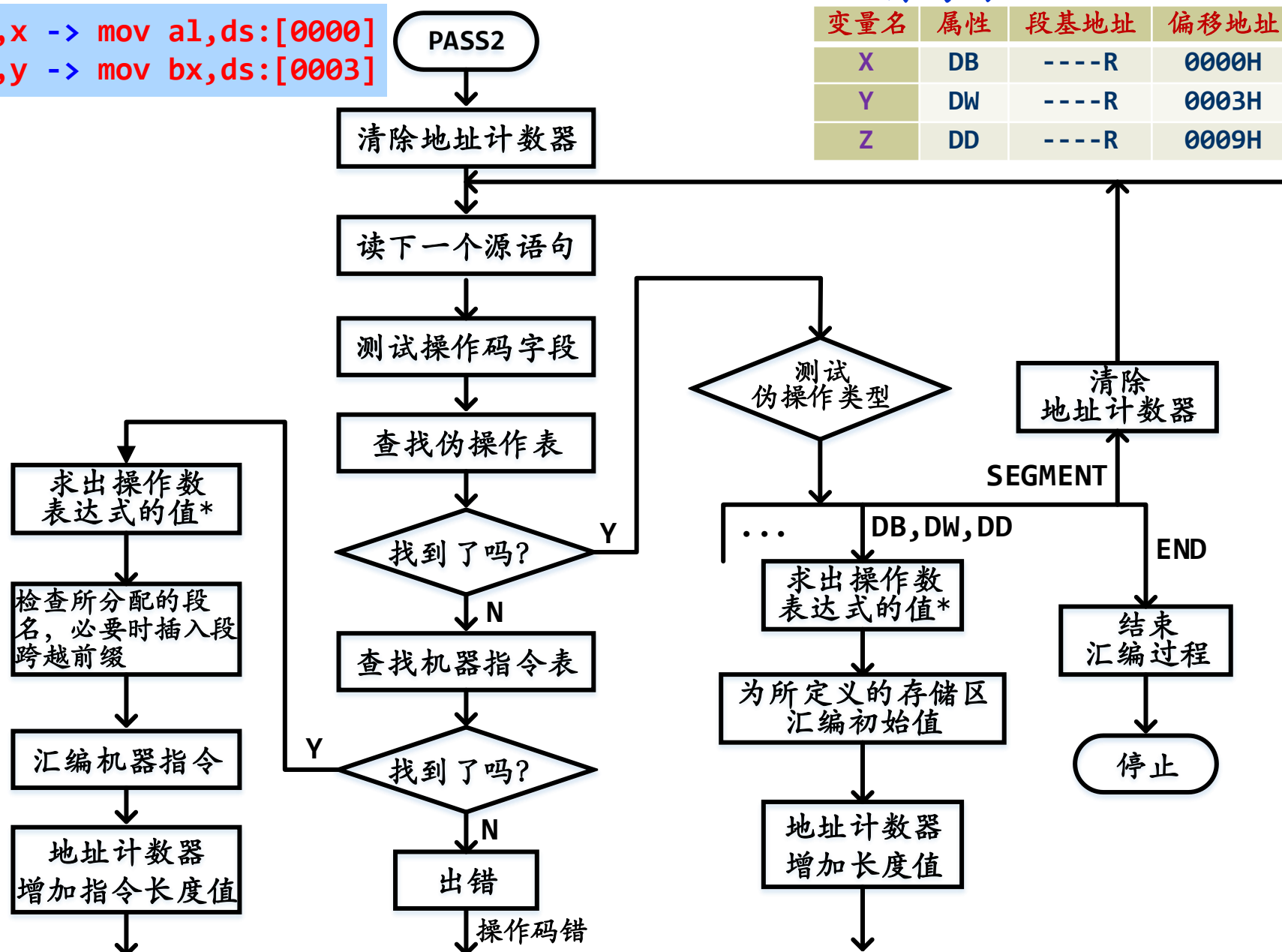
变量名	属性	段基地址	偏移地址
X	DB	----R	0000H
Y	DW	----R	0003H
Z	DD	----R	0009H

# 汇编程序第二遍扫描的主要流程

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



# 汇编语言指令

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

## 4.2 伪操作

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

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

## 4.2.1 处理器选择伪操作

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

此类伪操作参看P135

缺省时默认选择8086指令系统

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



## 4.2.2 段定义伪操作

格式:

segment\_name      segment

...

segment\_name      ends

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

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

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

## Segment 伪操作还可以增加存储器分配组合方式类型

### 格式：

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

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

◆ **定位类型：** BYTE、WORD、DWORD、PARA、PAGE指定段在存储器分配时的对齐属性

■ **PARA：** 段从小段地址开始 (**缺省默认**)

XX...XX0000

■ **BYTE：** 段从任意字节开始

XX...XXXXXX

■ **WORD：** 段从下一字地址开始，偶数地址开始

XX...XXXXX0

■ **DWORD：** 段从下一双字地址开始，开始地址最低2位为0，4的倍数

XX...XXXX00

■ **PAGE：** 段从下一页地址开始 (256字节为一页)

XX...XX00000000

## Segment 伪操作还可以增加存储器分配组合方式类型

### 格式：

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

- ◆ **组合类型**：PRIVATE、PUBLIC、COMMON、AT、STACK、MEMORY，  
程序连接时段的合并方式
  - **PRIVATE**：该段为私有段，不与其他模块中的同名段合并（缺省默认）
  - **PUBLIC**：同名段连接成一个物理段，连接次序由连接命令指定
  - **COMMON**：同名段起始地址相同，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

### 多个模块组合时的连接情况

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

## 例13.2:

### 源程序模块1

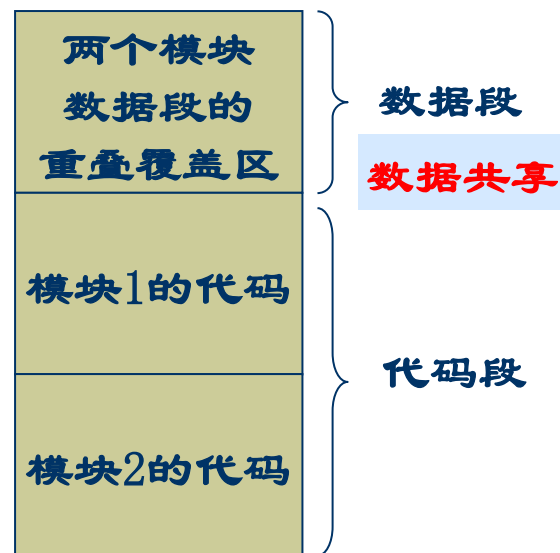
```
data segment common
...
data ends
code segment public
...
code ends
end start
```

### 源程序模块2

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

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

### 连接后装入模块的 存储区分配情况



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

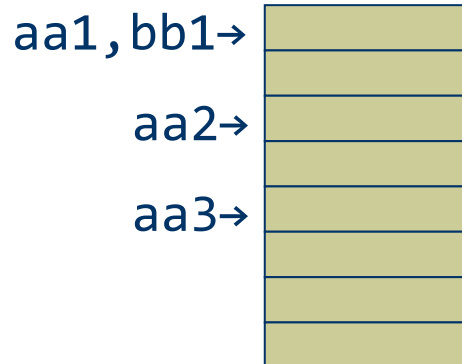
## 源程序模块1

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

## 源程序模块2

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

# 数据段覆盖组合定义



## 例13.3:

### 源程序模块1

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

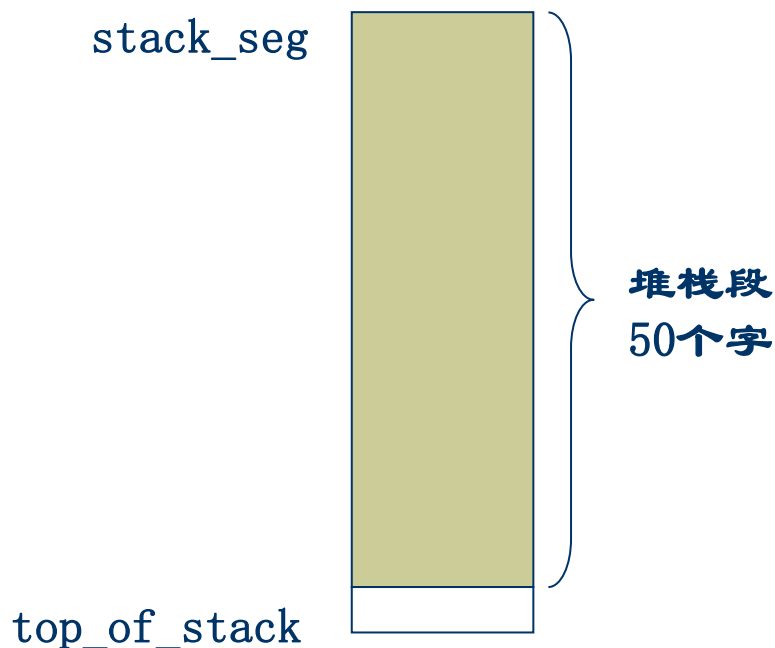
### 源程序模块2

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

模块1和模块2的堆栈段组成一段，之间没有留空

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！

连接后的堆栈段  
存储区分配情况



## Segment 伪操作还可以增加存储器分配组合方式类型

### 格式：

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

#### ◆ 使用类型：386及后续机型，指定偏移量长度

- USE16：16位寻址方式（缺省默认），段长64KB
- USE32：32位寻址方式，段长4GB
- 实模式下，应该使用USE16

#### ◆ 类别：‘CLASS’，给出组成段组的类型名

- 同类别的段装配在相邻的位置，组成段组
- 如 ‘code’，‘data’，‘bss’



例：定义用户堆栈

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

```
code_seg segment
```

.....

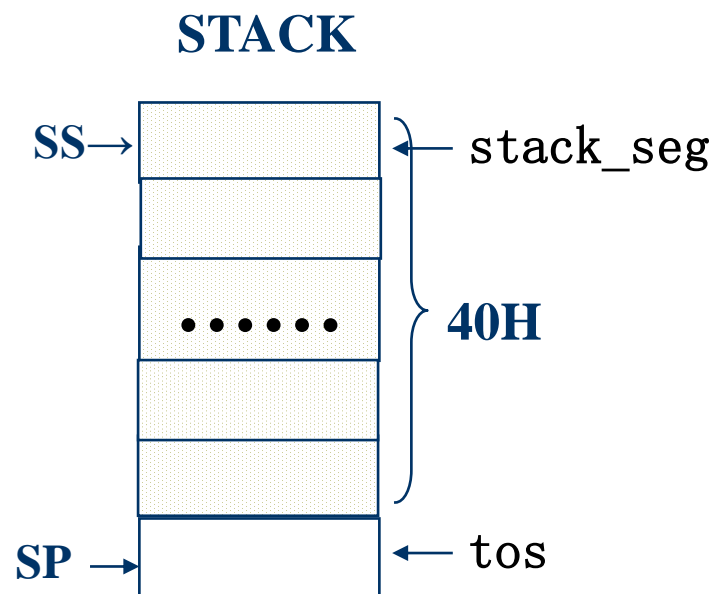
```
mov ax, stack_seg
mov ss, ax
mov sp, offset tos
```

.....

```
code_seg ends
```

注意段  
寄存器  
设置方  
式

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



例:

*data\_seg1 segment*  
...  
*data\_seg1 ends* ; 定义数据段

*data\_seg2 segment*  
...  
*data\_seg2 ends* ; 定义附加段

*code\_seg segment*  
    *assume cs:code\_seg, ds:data\_seg1, es:data\_seg2*  
    *start:*  
        *mov ax, data\_seg1*  
        *mov ds, ax*  
        *mov ax, data\_seg2*  
        *mov es, ax* ; 段地址→段寄存器  
        ...  
    *code\_seg ends*  
    *end start*

# ASSUME (约定) 伪指令

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

CS: CSSEGNAME

DS: DSSEGNAME

ES: ESSEGNAME

SS: SSSEGNAME

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

**使用段寄存器前必须给出约定**

- ◆ 应放在引用段寄存器之前，通常放在代码段或主过程的第一个语句位置
- ◆ 若一个段寄存器与NOTHING关联，则表示取消前边对该段寄存器的假设  
DS:nothing ; 取消原来段寄存器DS的预约分配
- ◆ ASSUME语句并不给段寄存器赋值

# 存储模型与简化段定义伪操作

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

# 1. MODEL伪操作

格式：

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

- ◆ **memory-model**：根据代码段和数据段在存储器中的安放合并方式，可以建立7种存储模型
  - ① TINY：所有数据 and 代码都放在一个段内，数据和代码都是近访问
    - TINY程序可以写成 .com文件，com程序必须从0100H存储单元开始，一般用于小程序
  - ② SMALL：所有数据放在一个64KB的数据段内，所有代码放在另一个64KB的代码段内，都是近访问 `.model small`
  - ③ MEDIUM：代码一个模块一个段，所有数据放在一个64KB的数据段内，数据近访问，代码远访问
  - ④ COMPACT：数据放在多个段内，所有代码放在一个64KB的代码段内
  - ⑤ LARGE：代码和数据都可以用多个段，数据和代码都可以远访问
  - ⑥ HUGE：代码和数据都可以用多个段，但允许数据段大小超过64KB
  - ⑦ FLAT：允许使用32位偏移量

**格式：**

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

◆ **model options:**

- ① **高级语言接口：**该汇编语言程序作为高级语言程序的过程调用，如C, BASIC, FORTRAN, PASCAL等
- ② **操作系统：**说明运行于哪种操作系统环境下，如OS\_DOS, OS\_OS2（缺省是OS\_DOS）
- ③ **堆栈距离：**NEARSTACK, FARSTACK
  - NEARSTACK：堆栈段和数据段组合到一个DGROUP段中，(DS)=(SS)
    - ◆ TINY、SMALL、MEDIUM、FLAT时缺省默认NEARSTACK
  - FARSTACK：堆栈段和数据段不合并
    - ◆ COMPACT、LARGE、HUGE时缺省默认FARSTACK

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

## 2. 简化的段定义伪操作

### ◆ 将数据段分得更详细

- 常数段、初始化段、未初始化段、远初始化段、远未初始化段
- 便于与高级语言兼容

### ◆ 格式：

- `.CODE [name]`      代码段
- `.DATA`      近初始化数据段
- `.DATA ?`      近未初始化数据段
- `.FARDATA [name]`      远初始化数据段, name缺省使用FAR\_DATA
- `.FARDATA ? [name]`      远初始化数据段, name缺省使用FAR\_BSS
- `.CONST`      常数段
- `.STACK [size]`      size缺省值为1KB

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

- ◆ 使用简化段伪操作时, 必须在程序一开始先用`.MODEL`
- ◆ 每个段的定义开始就是上一段的结束, 段结束符`ENDS`可以省略

# 与简化段定义有关的预定义符号

- ◆ @MODEL 用数值表示当前所用的存储模型

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

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

- ◆ @code: 由.CODE伪操作定义的代码段段名或段组名

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

- ◆ @codesize: 以数值表示当前代码段情况

- 只有一个代码段 ( TINY、SMALL、COMPACT 、FLAT ) , 数值为0
- 多个代码段 ( MEDIUM、LARGE、HUGE ) , 数值为1

mov ax, @codesize ;相当于 mov ax, 0或者1



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

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

立即数寻址方式

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

代替段名, 取段基地址

- ◆ 用汇编语言编写程序可以使用两种基本格式。完整段定义、简化段定义
- ◆ 完整段定义格式虽然需要较复杂的语法，但它可以提供完整的控制，也是大多数汇编程序通用的定义格式，兼容性好

# Why Even Bother With Segments?

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

## Why Even Bother With Segments?

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

## 4.2.3 程序命名和结束伪操作

### 程序命名：

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

### 程序结束

END      [执行的起始地址]

# 程序开始

## ◆ 模块命名

- 格式：NAME      module\_name
  - module\_name: 模块的名字

## ◆ 模块标题

- 格式：TITLE    text
  - 这样可在列表 (LST) 文件中, TITLE的标题名从第二页开始列出
  - text最多60个字符

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

## ◆ 程序中NAME、TITLE都没有

- 用源文件名作为模块名
- 但一般经常使用TITLE

# 程序结束

格式：

END [label]

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

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

- ◆ MASM6.0增加了定义程序入口点和出口点的伪操作  
(应该归为宏指令)

- **.STARTUP**

- 定义程序的初始入口点
- 汇编程序汇编时自动产生设置DS、SS和SP的代码
- 这时END伪指令可以不指定程序开始执行地址标号

- **.EXIT [return\_value]**

- 汇编程序汇编时自动产生退出程序并返回操作系统的代码
- return\_value返回给操作系统的值，常用 0

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

## 4.2.4 数据定义及存储器分配伪操作

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

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

- Mnemonic: 伪操作助记符, 是数据类型的符号表示

字节定义伪指令      DB

字定义伪指令        DW

双字定义伪指令     DD

四字定义伪指令     DQ

10个字节定义伪指令 DT

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

- Operand:

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

- Comments: 注释, 必须以 “;” 开始

X	0A	10
	04	4
	10	10H
Y	64	100
	00	
	00	100H
	01	
	?	?
Z	?	
	3C	60
	00	
	00	
	00	
	FD	0FFFDH
	FF	
	00	
	00	



## 为数据分配存储单元，并初始化单元内容

*data segment*    完整段定义

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

*data ends*

*.data*    简化段定义

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

X	0A	10
	04	4
Y	10	10H
	64	100
	00	
	00	100H
	01	
	FB	-5
Z	FF	
	3C	60
	00	
	00	
	00	
	FD	0FFFDH
	FF	
	00	
	00	

# 变量和标号属性：

## 所有的变量和标号都有三种属性

段基值 (SEG)

偏移量 (OFFSET)

类型 (TYPE)：变量 (字节/字/双字/四字/十字节)

标号 (NEAR / FAR)

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

```
NE: MOV AL, X
```

变量和标号作用是什么？

在汇编源程序中指明数据和指令的存储单元地址和可处理的方式

## 变量的类型属性总结：

(1) 指令中使用了变量名，表示的是该组数据的第一个字节的偏移地址

(2) 该操作中的每一个数据项

- DB伪操作 字节型 1字节

- DW伪操作 字型 2字节

- DD伪操作 双字型 4字节

- DF伪操作 6字节型 6字节

- ◆ 存放远地址（16位段地址，32位偏移地址）

- DQ伪操作 四字型 8字节

- DT伪操作 10字节型 10字节

(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

例 4.14

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

P148, 例4.14

(4) 指令中PTR指定的变量类型属性优先于隐含的类型属性

格式: **type** **PTR** **变量名**

■ type可以是: BYTE WORD DWORD FWORD QWORD TBYTE

■ 这样可使同一个变量具有不同的类型属性

Mov al, **byte ptr** yy

(5) 变量的另一种属性可以用LABEL伪操作来定义

格式: **name** **LABEL** **type**

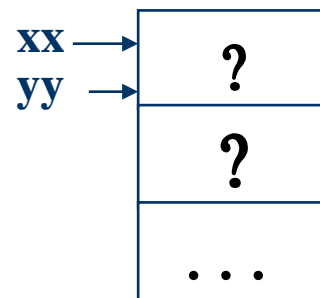
■ 数据项类型type可以是: BYTE WORD DWORD FWORD QWORD TBYTE

■ 对于可执行代码类型可以是: **NEAR** **FAR**

<i>xx</i>	<i>LABEL</i>	<i>BYTE</i>
<i>yy</i>	<i>DW</i>	<i>50 DUP (?)</i>

(6) 操作数字段可以使用复制操作符

repeat\_count **DUP** (operand, ..., operand)



**BYTE\_ARRAY** **LABEL** **BYTE**

**WORD\_ARRAY** **DW** **25 DUP(?, 1234H)**

• 为数组分配100个存储单元, 各单元字节内容:

**?, ?, 34H, 12H, ?, ?, 34H, 12H, ... , ?, ?, 34H, 12H**

• 指令中使用**BYTE\_ARRAY**, 以字节访问

• 指令中使用**WORD\_ARRAY**, 以字访问

• 有两个类型的变量名

**MOV** **BYTE\_ARRAY+2, 0**  
**; 该数组第3个字节置0**  
**MOV** **WORD\_ARRAY+2, 0**  
**; 该数组第3、4个字节置0**

例.

*data segment*

M1 DB 15, 67H, 11110000B, ?

M2 DB '15', 'AB\$'

M3 DW 4\*5

M4 DD 1234H

M5 DB 2 DUP (5, 'A')

M6 DW M2 ;M2的偏移量

M7 DD M2 ;M2的偏移量、段基址

*data ends*

**注意：常数和字符串在存储器中的放置次序！**

1470: 0000  
1470: 0001  
1470: 0002  
1470: 0003  
1470: 0004  
1470: 0005  
1470: 0006  
1470: 0007  
1470: 0008  
1470: 0009  
1470: 000A  
1470: 000B  
1470: 000C  
1470: 000D  
1470: 000E  
1470: 000F  
1470: 0010  
1470: 0011  
1470: 0012  
1470: 0013  
1470: 0014  
1470: 0015  
1470: 0016  
1470: 0017  
1470: 0018  
1470: 0019  
1470: 001A

0F	← M1
67	
F0	
?	
31	← M2
35	
41	
42	
24	
14	← M3
00	
34	← M4
12	
00	
00	
05	← M5
41	
05	
41	
04	← M6
00	
04	← M7
00	
70	
14	

## 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

- ◆ 有效的操作数格式
- ◆ 可求出常数值表达式
- ◆ 任何有效的助记符

- 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
```

- 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]

**DATAS SEGMENT**

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

**DATAS ENDS**

**CODES SEGMENT**

```
ASSUME CS:CODES,DS:DATAS
```

**START:**

```
MOV AX,DATAS
```

```
MOV DS,AX
```

```
mov al, adr
```

```
MOV AH,4CH
```

```
INT 21H
```

**CODES ENDS**

```
END START
```

## 4.2.6 地址计数器对准伪操作

### 1、地址计数器

- ◆ 在汇编程序对源程序汇编过程中，使用地址计数器保存当前正在汇编的指令地址
  - 用在指令中：本条指令的第一个字节的地址
  - 用在参数中：地址计数器的当前值
- ◆ 指令中地址计数器的值用 ‘\$’ 来表示，汇编语言允许用户直接用 ‘\$’ 来引用地址计数器的值

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



## 例1: cs:0074 MOV AX, \$+6

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

用在指令中

地址计数器的值

## 例2: ARRAY DW 1, 2, \$+4, 3, 4, \$+4

假设该数组的首地址0074H

- ① 汇编程序处理第一个\$+4时，地址计数器=0078H

因此，\$+4=0078+4=007CH

- ② 汇编程序处理第二个\$+4时，地址计数器=007EH

因此，\$+4=007E+4=0082H

ARRAY	01	}	0074
	00		
	02		
	00		
	7C	}	0078
	00		
	03		
	00		
	04	}	007E
	00		
	82		
	00		

用在参数中

## 2、ORG 伪操作

- 用来设置当前地址计数器的值，即分配后续数据、指令的存储器开始地址

- 格式如下：

ORG 常数表达式 (n)

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

- 例如：

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

### 3、EVEN 伪操作

- **格式：** EVEN
  - **功能：** 使下一个变量或指令地址开始于偶字节地址
- ```
A DB 'morning'
EVEN
B DW 2 DUP (?)
```

### 4、ALIGN 伪操作

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

ALIGN 2 = EVEN

```
...
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

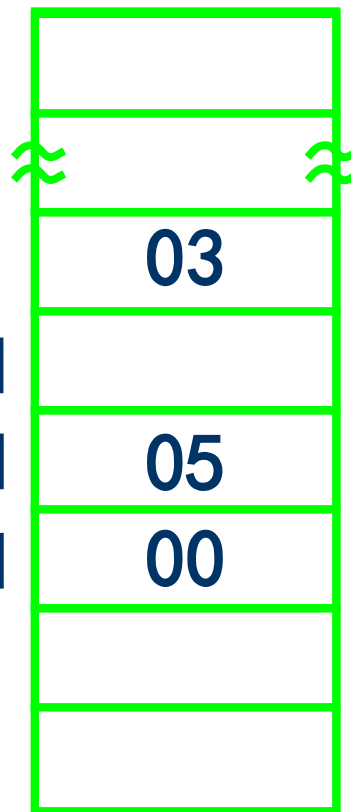
A2 DW 5

DS:0050H

DS:0051H

DS:0052H

DS:0053H



EVEN语句的效果

## 4.2.7 基数控制伪操作

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

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

### RADIX 伪操作

**格式：**RADIX 表达式 (n) ； 表达式表示基数值，用十进制数表示

**功能：**用于改变汇编语言默认的基数（范围为：2~16）

例如：

...

**RADIX 16**

MOV BL, 0FF

MOV BX, 199D

如果199D是十六进制，后边必须带H

**MOV BX, 199DH**

## 4.3 汇编语言指令格式

[名字项]

↓  
变量  
标号

操作助记符

↓  
机器指令  
伪指令  
宏指令

操作数

↓  
寄存器  
存储器  
标号  
变量  
常数  
表达式

[; 注释]

↓  
说明程序  
或语句  
的功能

带 [ ] 的两项是可缺省的

表达式：数字表达式，地址表达式

**[名字项]    操作项    操作数项    [; 注释项]**

◆ **机器指令语句格式：**

**[标号:]    操作项    [操作数1 [, 操作数2]]    [; 注释]**

◆ **伪指令语句格式：**

**[变量]    伪操作项    [操作数1 [, 操作数2]]    [; 注释]**

〔名字项〕 操作项 操作数项 〔; 注释项〕

### 4.3.1 名字项

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



## 1、标号

- 是用符号表示的指令地址，也叫符号地址
- 标号有3个属性：**段地址** **偏移地址** **类型**
  - 标号的段地址和偏移地址是指标号对应的指令首字节所在的段基地址和段内的偏移地址
  - 标号的类型属性有NEAR和FAR类型
- **标号的定义**：直接在指令助记符前加上标识符，必须以冒号“:”结束，如 **NEXT:**  
`next: mov ax, data_seg1`
- 标号经常在转移指令或CALL指令的操作数字段出现，用以表示转向地址

## 2、变量

- 是数据项的第一个字节相对应的标识符
- 变量有3个属性：**段地址** **偏移地址** **类型**
  - 变量的段地址和偏移地址：是指变量对应的数据项首字节所在的段地址和段内的偏移地址
  - 变量类型属性：定义该变量所保留的字节数，如 BYTE（1个字节长），WORD（2个字节长），DWORD（4个字节长）...
- 变量的定义：
  - (1) 变量在数据段或附加段中定义，后面不跟冒号
  - (2) 用LABEL、DB、DW、DD伪操作定义变量属性

A DB 'morning'
- 变量经常在操作数字段出现

## 4.3.2 操作项

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

### 4.3.3 操作数项

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

# 表达式

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

# 操作数有关的常用操作符

## 1、算术操作符

- 算术操作符有：+、-、\*、/和MOD

其中MOD是指除法运算后得到的余数

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

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

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

**汇编时将ARRAY认为地址偏移量来计算**

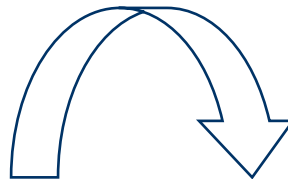
## 2、逻辑与移位操作符

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

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

$$\begin{array}{r} 00011001 \\ 00000111 \\ \hline 00000001 = 01H \end{array}$$

汇编后等同



AND AX, OPR1 AND OPR2

AND AX, 0001H

### 3、关系操作符

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

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

X: .....

.....

Y: .....

若 $\leq 128$  (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 |
| 值：  | 1  | 2  | 4  | 8  | 10 |

|     |      |     |
|-----|------|-----|
| 标号： | NEAR | FAR |
|     | -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
```

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

### ■ 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

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

## 4.3.4 注释项

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

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

## 4.4 汇编语言程序的上机过程

4.4.1 建立汇编语言的工作环境

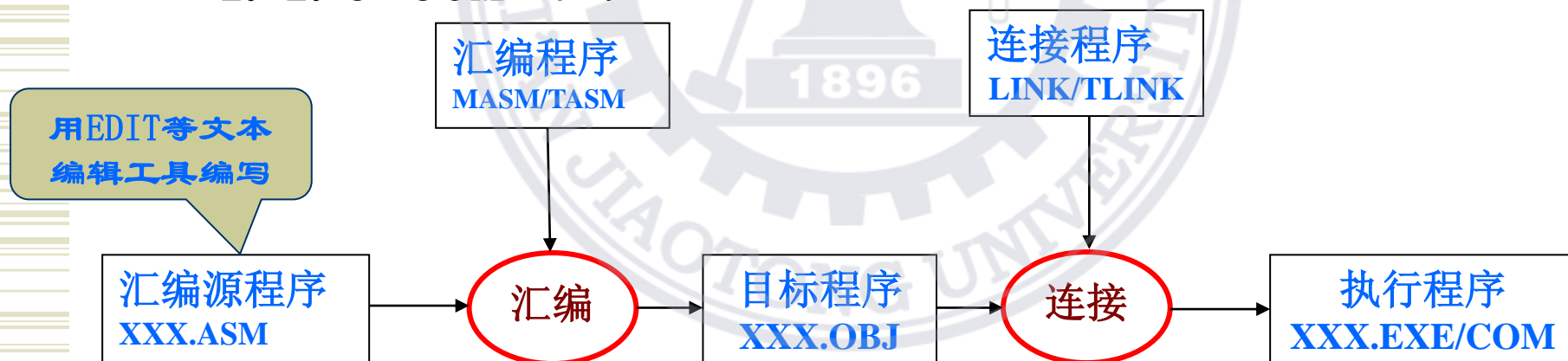
4.4.2 建立ASM文件

4.4.3 用MASM(或TASM)程序产生OBJ文件

4.4.4 用LINK(或TLINK)程序产生EXE文件

4.4.5 程序的运行

4.4.6 COM文件



## 4.4.1 建立汇编语言的工作环境

建议新建一个文件夹设为D\ASM

### ➤ 微软公司的产品

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

### ➤ Borland公司的产品

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

在DOS平台上使用的较普遍的汇编器是MASM和TASM

## 4.4.2 建立ASM文件

**第一步：编辑输入汇编语言源程序**

**方法：** D\ASM>EDIT pname.asm ↙

◆ 用编辑程序（如DOS6.2的EDIT）将汇编语言源程序输入计算机，经修改认为无误后，存入磁盘的文件系统

- 例如，设源文件定名为pname.asm
- 汇编语言源程序的后缀（扩展名）为“.asm”
- P162 例4.30

◆ 若启动时带有文件名但该文件不存在，则启动后可以输入新文件，否则把已存在的文件调入编辑

例如：

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

则屏幕显示：

```
File Edit Search View Options Help
```

```
|----- D:\masm\pname.ASM -----|
```

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

## 4.4.3 用MASM(或TASM)程序产生OBJ文件

### 第二步：汇编产生OBJ文件

D\ASM> MASM pname ✓

调用MASM (或者TASM) 汇编程序对源文件进行汇编

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



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

▪ 例如 D:\MASM>MASM pname; ↙

屏幕显示:

Microsoft (R) Macro Assembler Version 5.10

. . .

Source filename [pname.ASM]:

(汇编命令中没有输入源文件名时输入源文件名, 不必输入扩展名)

Object filename [pname.OBJ]:

(要求回答目标文件名, 可直接按Enter确认)

Source listing [NUL.LST]: pname

(列表文件名, 需要时输入名字部分, 缺省情况不生成)

Cross-reference [NUL.CRF]:

(交叉引用文件名, 需要时输入名字部分, 缺省情况不生成)

51058 + 421678 Bytes symbol space free

0 Warning Errors

0 Severe Errors

- ◆ 也可以用命令行的形式按顺序对四个提示予以回答，格式是：

**MASM 源文件名, 目标文件名, 列表文件名, 交叉引用文件名**

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

D:\MASM>MASM pname, , pname;

- ◆ 另外如果需要得到列表文件pname.lst，可按如下方法进行汇编：

D\ASM>MASM pname.asm/l

## ◆ 可产生的3个文件：

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

- 源程序和机器语言程序清单

偏移量 目标码

汇编格式

```
.....  
0000    1E                PUSH DS        ;save old data segment  
.....
```

- 段名表：段名、段大小、属性

Segments and Groups:

| Name       | Length | Align | Combine | Class |
|------------|--------|-------|---------|-------|
| CODE ..... | 001D   | PARA  | NONE    |       |
| DATA ..... | 0028   | PARA  | NONE    |       |
| EXTRA ...  | 0028   | PARA  | STACK   |       |

- 符号表：用户定义的符号名、类型、属性

- 交叉引用文件(.CFER)：

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

- P166

Microsoft Cross - Reference Version 5.00

Wed Mar 04 00:34:07 1998

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

|                     |      |    |    |    |
|---------------------|------|----|----|----|
| CODE .....          | 17 # | 21 | 50 |    |
| DATA .....          | 3 #  | 8  | 21 | 31 |
| DEST. BUFFER.....   | 11 # | 41 |    |    |
| EXTRA.....          | 10 # | 15 | 21 | 35 |
| MAIN .....          | 19 # | 48 |    |    |
| SOURCE. BUFFER..... | 4 #  | 39 |    |    |
| START.....          | 23 # | 53 |    |    |

7 Symbols

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

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

## 4.4.4 用LINK(或TLINK)程序产生EXE文件

### 第三步：连接程序产生EXE文件

D\ASM>LINK pname; ✓

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

例如：

D:\MASM>LINK

Object Modules [.OBJ]: pname

(输入由汇编产生的.OBJ目标文件名)

Run File [pname.EXE]:

(直接回车确认系统给出的默认可执行文件名，也可改为其他名字)

List File [NUL.MAP]: pname

(输入内存映象文件名，可缺省不产生，直接按回车键)

Libraries [.LIB]:

(程序用到的库文件，如无特殊需求，直接按回车键)

◆ **pname.OBJ经连接后在当前目录下产生了  
pname.EXE和pname.MAP文件**

- **可执行文件（.EXE）：操作系统可装入、动态重新分配内存空间的执行程序**
- **内存映象文件（.MAP）：给出每个段在存储器中的分配情况**

- **pname.MAP文件的内容为：**

| Start  | Stop   | Length | Name  | Class |
|--------|--------|--------|-------|-------|
| 00000H | 00027H | 00028H | DATA  |       |
| 00030H | 00057H | 00028H | EXTRA |       |
| 00060H | 0007CH | 0001DH | CODE  |       |

Program entry point at 0006:0000

- ◆ 00060H=00060+00000
- ◆ 从小段的开始地址分配段基地址

◆ **程序装入时，动态分配实际的物理段基地址，设置段寄存器内容**

- **如P162：程序段基址0006变成了08FE；根据MAP文件，数据段基址0000应该是08FE-0006=08F8**



# 进一步说明的问题

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

## 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，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`

```
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)，这样符号表中符号的偏移地址才能正确形成

## 2、“浮动”地址概念

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

## 4.4.5 程序的运行

### 第四步：运行程序

D: \ASM>pname ↙ 或 pname.exe ↙

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

# DOS装入 .EXE文件的过程

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

- ② 把文件头读入内存
- ③ 计算可执行模块的大小
- ④ 计算装入的起始段地址
- ⑤ 完成重定位



## ⑥ 初始化段寄存器和指针寄存器。

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

## ⑦ 把控制权交给 .EXE 程序。



## ◆ 需要使DS指向用户程序的数据段

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



# 如何返回DOS

## 方法1、

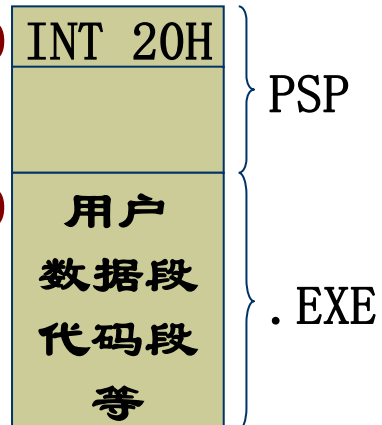
```
MAIN  PROC  FAR
        ASSUME  ...
        PUSH  DS
        XOR   AX, AX
        PUSH  AX
        ...
        RET
```

```
;PSP段基址入栈
;清0
;数字0入栈
;完成程序指定功能
;PSP:0 送 CS:IP
```

DS、ES → PSP:0000

SS:SP → PSP:0100

CS:IP →





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

## ◆ 方法2、

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

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



- 
- 
- ◆ 若想查看内存中的结果，请借助动态调试软件DEBUG/TD
  - ◆ 若执行结果有错误，请借助动态调试软件DEBUG/TD

# DEBUG简介

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

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

- ◆ DEBUG常用命令：

u、t、p、g、d、r、q

## (1) 反汇编命令u

格式： u /u cs:偏移量

例如：

-u

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

## (2) 跟踪执行命令t

格式：t/t=cs: 偏移量

例如：

-t

AX=0243 BX=0000 CX=0064 DX=0000 SP=003C  
BP=0000 SI=0000 DI=0000 DS=1306 ES=12F2  
SS=1302 CS=1307 IP=0010 NV UP EI PL NZ AC  
PO NC

1307:0010 A30400 MOV [0004], AX  
DS:0004=0000

## (3) 过程执行命令p

p (CALL REP LOOP INT等)

## (4) 运行命令 g

格式：g程序断点

例如：

-g9

AX=1306 BX=0000 CX=0064 DX=0000 SP=003C BP=0000  
SI=0000 DI=0000

DS=1306 ES=12F2 SS=1302 CS=1307 IP=0009 NV UP EI  
PL ZR NA PE NC

1307:0009 A10000 MOV AX, [0000] DS:0000=007B

## (5) 显示存储单元内容命令 d

格式：d 段地址：偏移地址

例如：

-d ds:0 f

1306:0000 7B 00 C8 01 00 00 00 00-00 00 00 00 00 00 00 00

## (6) 寄存器显示/修改命令r

格式：r 寄存器名称

例如：

-r

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

标志位值的符号表示：

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

## (7) 存储单元修改命令e

格式：e 段地址：偏移地址

## (8) debug退出命令q

格式：q

# Turbo debugger

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

# TD的机器指令级调试界面

| 主 菜 单 条 |              |         |
|---------|--------------|---------|
| 代码显示区域  | 寄存器值<br>显示区域 | 标志值显示区域 |
| 数据显示区域  | 堆栈显示区域       |         |
| 操作提示区域  |              |         |



|         |          |      |                  |    |      |     |
|---------|----------|------|------------------|----|------|-----|
| cs:0100 | 80740883 | xor  | byte ptr [si+08] | ax | 0000 | c=0 |
| cs:0104 | C602E2   | mov  | byte ptr [bp+si] | bx | 0000 | z=0 |
| cs:0107 | E4F9     | in   | al,F9            | cx | 0000 | s=0 |
| cs:0109 | EB01     | jmp  | 010C             | dx | 0000 | o=0 |
| cs:010B | F8       | clc  |                  | si | 0000 | p=0 |
| cs:010C | 5F       | pop  | di               | di | 0000 | a=0 |
| cs:010D | 5E       | pop  | si               | bp | 0000 | i=1 |
| cs:010E | 1F       | pop  | ds               | sp | 0080 | d=0 |
| cs:010F | 59       | pop  | cx               | ds | 1817 |     |
| cs:0110 | 5B       | pop  | bx               | es | 1817 |     |
| cs:0111 | 58       | pop  | ax               | ss | 1817 |     |
| cs:0112 | C3       | ret  |                  | cs | 1817 |     |
| cs:0113 | 51       | push | cx               | ip | 0100 |     |

|         |                         |       |        |
|---------|-------------------------|-------|--------|
| ds:0000 | CD 20 00 A0 00 9A F0 FE | =     | a ü    |
| ds:0008 | 1D F0 10 08 2E 13 0F 07 | ≡     | .!!    |
| ds:0010 | 2E 13 5D 08 3B 11 11 13 | .!!   | ; <<!! |
| ds:0018 | 01 01 01 00 02 08 FF FF | ☹☹☹ ☹ |        |

ss:0082 0000

ss:0080 0000

## 4.4.6 .COM文件

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

- ◆ 程序段前缀PSP段长为100H字节，PSP:0处存放一条INT 20H指令。程序的二进制代码紧跟PSP之后装入
- ◆ .COM程序的代码、数据及堆栈数据在同一段中，所以对所有的段寄存器都初始化为指向PSP的段基址。IP = 100H, 为PSP之后的下一个地址偏移量，SP指向栈顶，栈顶中存入一个字型数字0，如下图所示

CS、DS、ES、SS

CD

20

PSP

IP

$\leq 64K$

SP

00

FFFE

00

## .COM文件装入内存示意图

```

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

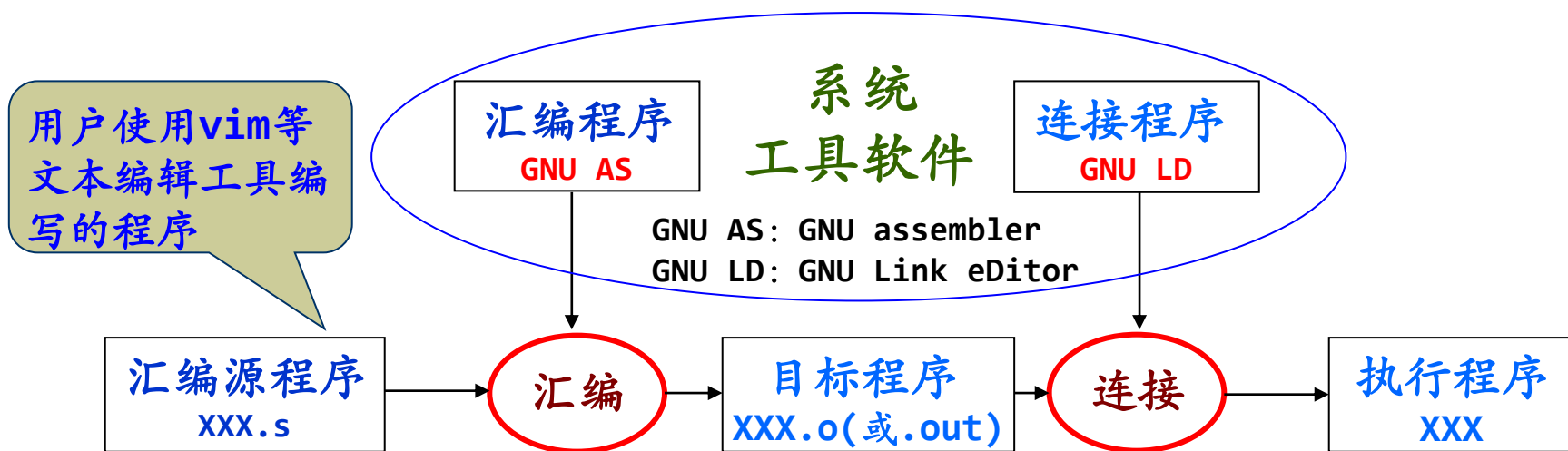
```

# COM文件的上机步骤

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

## 4.5 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



## 4.5.1 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。



## 4.5.1 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

`.data`

`data`段内容

`.section my_section`

自定义内容

①

`.data`

`data`段内容

`.text`

代码段内容

②

`.data`

`data`段内容

文件结束

③

## 4.5.2 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
```

## 4.5.3 ARM64过程定义

### ◆ 格式:

- **.type 函数名, %function**
- 功能: 定义一个函数
- **%function**说明类型是函数

```
.text
.type main, %function
.global main

main:
    stp x29, x30, [sp,#-16]!
    ...
    ret
```

- ◆ 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

## 4.5.4 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
```

|   |    |   |     |
|---|----|---|-----|
| X | 00 | } | 0   |
|   | 00 |   |     |
| Y | 41 | } | 'A' |
|   | 42 |   |     |
| Z | 00 | } | 0   |
|   | 00 |   |     |
|   | 00 |   |     |
|   | 00 |   |     |
|   | 01 | } | 1   |
|   | 00 |   |     |
|   | 00 |   |     |
|   | 00 |   |     |
|   | 03 | } | 3   |
|   | 00 |   |     |
|   | 00 |   |     |
|   | 00 |   |     |

## 4.5.4 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
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
~
```

|   |    |         |
|---|----|---------|
| X | 0A | 10      |
|   | 04 | 4       |
| Y | 10 | 10H     |
|   | 64 | 100     |
|   | 00 |         |
|   | 00 | 0x100   |
|   | 01 |         |
|   | FB | -5      |
| Z | FF |         |
|   | 3C | 60      |
|   | 00 |         |
|   | 00 |         |
|   | 00 | 0xFFFFD |
|   | FD |         |
|   | FF |         |
|   | 0F |         |
|   | 00 |         |

## 4.5.5 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。

## 4.5.6 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。

## 4.5.6 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 地址计数器的值 page 1

扫描时确定偏移地址

| 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 |                       |



## 4.5.6 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
```

## 4.5.7 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**

## 4.6 ARM64汇编语言程序格式

[名字项]

↓  
变量  
标号

操作助记符

↓  
机器指令  
伪指令  
宏指令

操作数

↓  
寄存器  
存储器  
标号  
变量  
常数  
表达式

[//注释]

↓  
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

## 4.6.1 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

## 4.6.2 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

## 4.6.2 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

## 4.6.3 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

|      |    |        |
|------|----|--------|
| 0000 | 01 | ARRAY  |
|      | 00 |        |
|      | 02 |        |
|      | 00 |        |
|      | 03 |        |
|      | 00 |        |
| 000C | 04 | ARYEND |
|      | 00 |        |
|      | 05 |        |
|      | 00 |        |
|      | 06 |        |
|      | 00 |        |
| 000C | 07 | ARYEND |
|      | 00 |        |
|      | ?  |        |
|      | ?  |        |

## 4.6.4 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
    msg: .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, msg
      bl  printf
```



The background of the slide features a large, light gray watermark of the Xi'an Jiaotong University (XJTU) logo. The logo is circular, with a gear-like outer ring. Inside the ring, the university's name is written in Chinese characters '西安交通大学' at the top and 'XI'AN JIAOTONG UNIVERSITY' at the bottom. The center of the logo depicts a bell on a pedestal, with the year '1896' below it.

**谢谢！**