# 第六章 子程序结构

6.1 子程序的设计方法

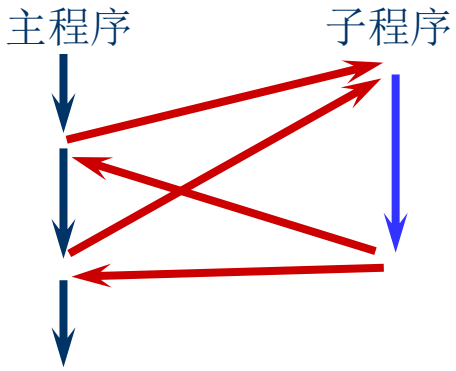
6.2 嵌套与递归子程序

6.3 子程序举例

6.4 DOS系统功能调用

# 子程序结构

- 子程序又称为过程，它相当于高级语言中的过程和函数
- 为什么需要子程序：
  - 程序段共享
  - 模块化设计
  - 简化程序设计
    - 简化功能和结构形式相同的程序段在程序中多次使用的设计复杂性
  - 节省存储空间
- 使用子程序的主要优点：
  1. 节省存储空间
  2. 减少程序设计时间



# 6.1 子程序的设计方法

## 6.1.1 过程（子程序）定义

- 格式： procedure name PROC Attribute

  ……

  procedure name ENDP

  - 过程名（procedure name）是子程序入口的符号地址
  - 类型属性（Attribute）：NEAR、 FAR

- 定义语句是伪操作，只告诉汇编程序如何处理；机器硬件不能识别和处理，即机器指令集中没有的语句。是汇编源程序设计和存储器分配的有效辅助手段

- **过程属性的确定原则：**
  - 调用程序和子程序在同一代码段中，则使用NEAR属性
  - 调用程序和子程序不在同一代码段中，则使用FAR属性
- **80X86汇编程序在汇编时用过程属性确定CALL和RET指令属性**

涉及到返回地址保存和恢复


  - 用户要在定义过程时考虑属性； CALL和RET指令属性可以不考虑，让汇编程序确定，但注意“向前引用”“向后引用”情况

# 例6.1　调用程序和子程序在同一代码段中

```
code  segment

main  proc  far
      ......
      call subr1
      ......
      ret
main  endp

subr1  proc near
       ......
       ret
subr1  endp

code  ends
```

主过程应定义为FAR属性。它是DOS调用的一个子过程

保存返回地址IP；确定目标指令的IP

NEAR属性

只恢复返回地址的IP

SP→ | | IP原 |

```
code  segment

main  proc  far
      ......
      call subr1
      ......
      ret

subr1  proc near
       ......
       ret
subr1  endp

main  endp
code  ends
```

堆栈

SP→ | | IP原 | CS原 |

复返回地址的IP、CS

过程定义可以嵌套，即一个过程定义中包含多个过程定义

Call subr1“向前引用”的调用有问题吗？ Call NEAR PTR subr1

# 例6.2　调用程序和子程序不在同一代码段中

```
segx   segment
subt   proc  far
       ......
       ret
subt   endp
       ......
       call  subt
       ......
segx   ends

segy   segment
       ......
       call  subt
       ......
segy   ends
```

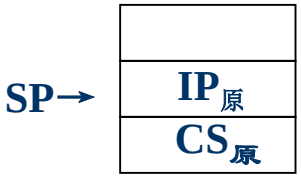
FAR属性

恢复返回地址的IP，CS

保存返回地址CS, IP；
确定目标指令的CS, IP

保存返回地址CS, IP；
确定目标指令的CS, IP

堆栈




- FAR属性子程序可以被同一段内或不同段内程序调用
- NEAR属性子程序只能被同一段内程序调用

# 6.1.2 子程序调用和返回

- **子程序调用和返回**

  （1）CALL　（2）RET　CALL和RET一定要根据属性配对使用

- **子程序调用CALL：**

  ① 隐含使用堆栈保存返回地址(IP或者CS,IP)

  ② 让程序指针(CS,IP)指向子程序入口

  call near ptr subp

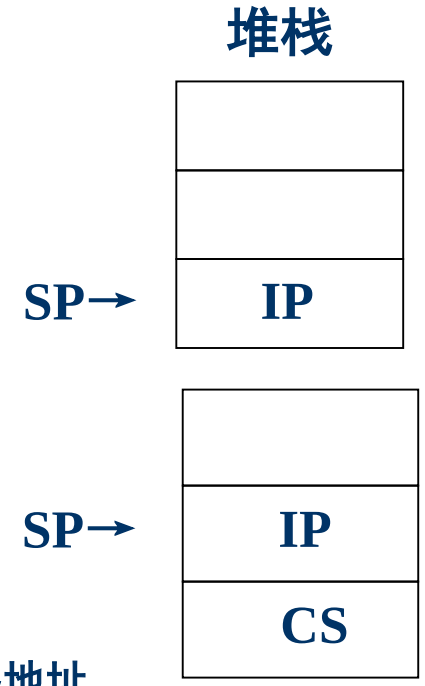
  （1）保存返回地址：隐含执行 PUSH IP

  （2）转子程序： IP ← subp的偏移地址

  call far ptr subp

  （1）保存返回地址： 隐含执行 PUSH CS 和 PUSH IP

  （2）转子程序： CS ← subp的段地址； IP ← subp的偏移地址



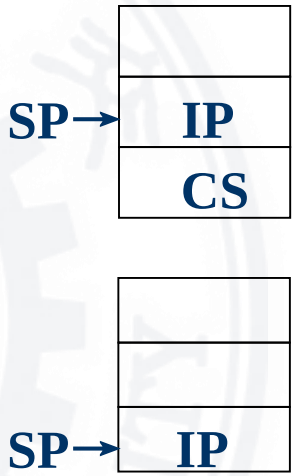

DOS功能用中断调用

# 子程序设计时应特别注意正确使用堆栈，及堆栈状态变化；

## 注意：子程序调用时属性不可以改变！

```
subp  proc far
        ……
        ……
      ret   ;汇编后对应retf指令，恢复IP，CS
subp endp

        ……
        ……

    call near ptr subp  ;调用时保存IP
        ……
        ……
```

堆栈




```
E8 000D          call near ptr bb
9A 0000 ---- R   call far ptr bb
```

```
subr1  proc near
        ……
        ret
subr1  endp
```

该子程序中的所有RET汇编时翻译成机器指令RET(C3H)

```
subr1  proc far
        ……
        ret
subr1  endp
```

该子程序中的所有RET汇编时翻译成机器指令RETF(CBH)

在汇编时用过程属性确定CALL和RET的具体机器指令

- int n （n:中断类型号）：DOS功能调用

(1) 自动保存现场（FLAGS）和返回地址（CS, IP）

- 隐含执行 PUSHF, PUSH CS 和 PUSH IP

(2) 中断标志位清0（关中断）等

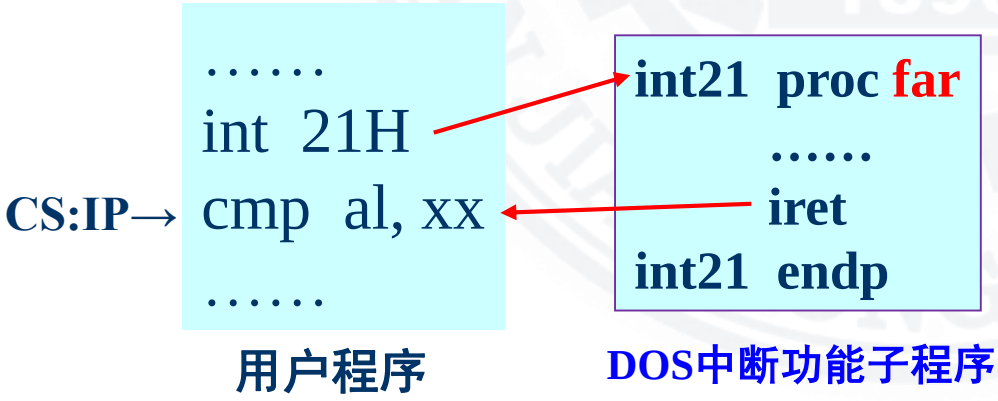
(3) 转中断处理程序

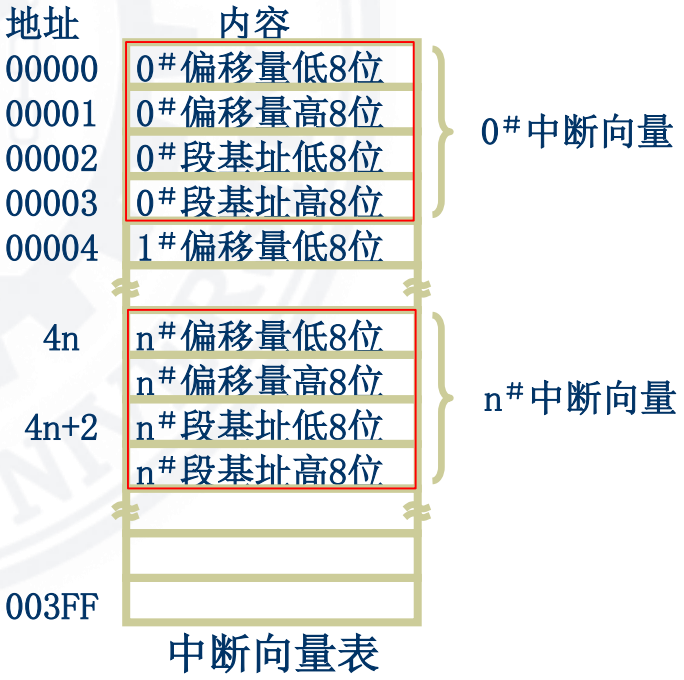
IP ← (n*4)

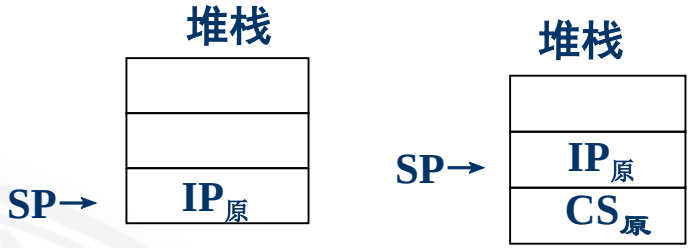
CS ← (n*4+2)

} 中断处理程序入口地址


# DOS功能调用 int 21H

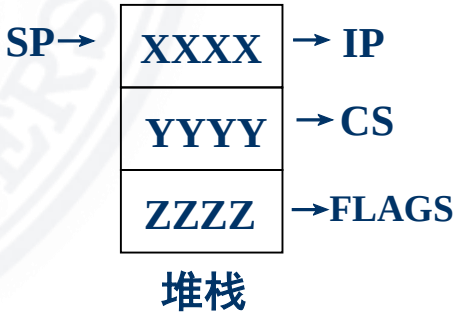





- 子程序返回：
  - 指令格式　ret　或者 ret imm8
    - 恢复返回地址：IP 或者　IP,CS
      - 段内（NEAR）：隐含执行 POP IP
      - 段间（FAR）：隐含执行 POP IP 和 POP CS


- 中断返回：
  - 指令格式　iret
    - 恢复现场（FLAGS）和返回地址（IP, CS）
    - 隐含执行 POP IP, POP CS 和 POPF




**注意：返回恢复和调用保存时次序相反**

```
0000                                cseg1   segment
                                            assume cs:cseg1
0000                                start   proc far
                                    ;
0000 E8 000D R                              call near ptr aa   ;近调用可以省略属性?
0003 9A 0000 ---- R                         call far ptr bb
                                    ;
0008  B8 4C00                       exit:   mov ax,4c00h
000B  CD 21                                 int 21h
000D                                start   endp
                                    ;
000D                                aa      proc near
000D  90                                    nop
000E  C3                                    ret
000F                                aa      endp
000F                                cseg1   ends
                                    ;
                                    ;
0000                                cseg2   segment
                                            assume cs:cseg2
0000                                bb      proc far
0000  90                                    nop
0001  CB                                    retf
0002                                bb      endp
0002                                cseg2   ends
                                    ;
                                            end  start
```

机器指令3个字节

机器指令5个字节

代码段1

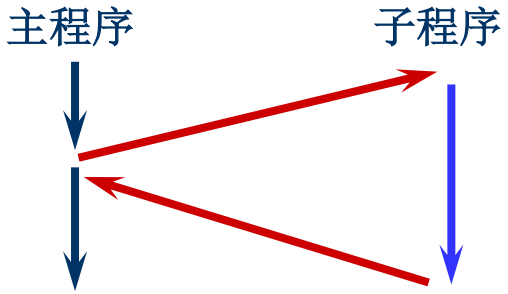
代码段2

.LIST文件

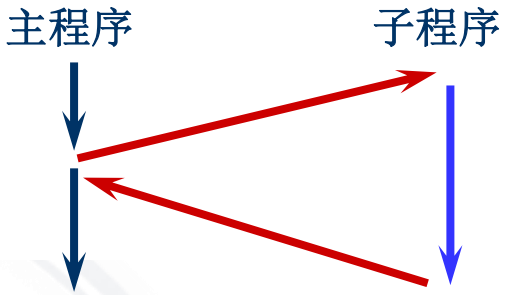
## Debug反汇编查看内存

```
-u 1424:0000 0b
1424:0000 E80A00          CALL    000D
1424:0003 9A0000 2514     CALL    1425:0000
1424:0008 B8004C          MOV     AX,4C00
1424:000B CD21            INT     21
-
-u 1424:000d 0e
1424:000D 90              NOP
1424:000E C3              RET
-
-u 1425:0000 01
1425:0000 90              NOP
1425:0001 CB              RETF
```

# 6.1.3 保存和恢复寄存器




- 为什么子程序中要保存和恢复寄存器
  - 子程序是独立的共享模块，对寄存器使用具有独立性，这样会产生主程序和子程序使用寄存器冲突
  - 为了解决主程序和子程序使用寄存器冲突，保证主程序正确运行，子程序中必须保存相关使用的寄存器

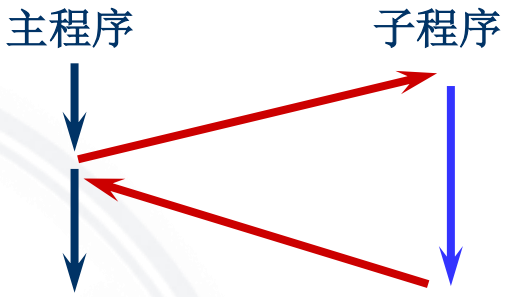



# (1) 保护和恢复寄存器的方法

- 子程序开始时，使用PUSH指令保存
- 子程序返回前，使用POP指令恢复
- 保存和恢复次序应该相反

```
subt    proc    far

        push    ax
        push    bx
        push    cx
        push    dx
        ……
        ……
        pop     dx
        pop     cx
        pop     bx
        pop     ax

        ret

subt    endp
```

- 子程序设计时应特别注意正确使用堆栈，及堆栈状态变化。一般情况下，子程序中PUSH和POP指令必须配对执行！
- 保证SP指向正确位置的情况下，可以灵活使用PUSH/POP/RET等指令




## (2) 确定保护哪些寄存器的原则

- 保护子程序中将要使用的寄存器及标志寄存器即可
  - 子程序独立性强，不了解调用程序的寄存器使用情况
  - 如果了解调用程序的寄存器使用情况，可适量保存
- 用寄存器向主程序回送结果的寄存器不必保存
- FLAGS寄存器保存优先，恢复时最后恢复

# 6.1.4 子程序的参数传送

- 参数传送：调用程序和子程序之间的信息传送
  - 调用时，主程序传送参数给子程序
  - 返回时，子程序返回参数给主程序

CALL/RET指令只是改变指令指针

- 参数传送的一般途径
  - 寄存器
  - 存储器

参数放在主程序和子程序都可以访问到的地方

- 参数放在主程序和子程序都可以访问到的地方，根据各种寻址方式访问
- 在基本原理的基础上, 实现方法的创新、编程技巧

## 参数传送的具体方法：实际编程中创新

（1）通过寄存器传送参数

（2）通过存储器传送参数：（共享存储单元）

*子程序和调用程序在同一程序模块中，则子程序可直接访问模块中的变量（存储单元）（本章）

*子程序和调用程序不在同一程序模块中，则有两种传送方式：建立公共数据区和使用外部符号(13章)

（3）通过地址表传送变量地址

（4）通过堆栈传送参数或变量地址

达到主程序和子程序能访问到相同存储单元的目的即可

组合类型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

# （1）通过寄存器传送参数

- 这种传递方式使用方便，适用于参数较少的情况

例6.3 十进制到十六进制的转换程序，**通过寄存器BX传送参数**

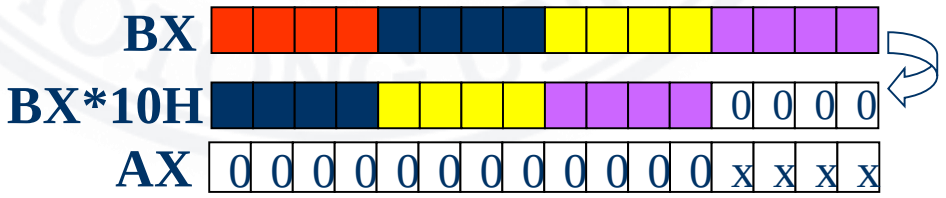
```
decihex  segment                    ; 10→16
         assume  cs: decihex

main     proc   far
         push   ds
         sub    ax, ax
         push   ax
repeat:  call   decibin   ; 从键盘取10进制数，10→2，保存在BX中
         call   crlf      ; 显示回车换行，防止屏幕显示重叠
         call   binihex    ; 2→16，并在屏幕上显示
         call   crlf
         jmp    repeat
         ret
main     endp
```

# 从键盘取10进制数，10→2，保存在BX中

```
decibin   proc  near
          mov   bx, 0     ; bx初始化，也可以为空格的ASCII码“      ”
newchar:  mov   ah, 1     ; 从键盘读键的ASCII码子功能
          int   21h       ; 从键盘取10进制数0-9键的ASCII码→al
          sub   al, 30h   ; 0-9的ASCII码30-39
          jl    exit      ; <0退出
          cmp   al, 9
          jg    exit      ; >9退出
          cbw             ; AL符号扩展到AH
          mov   cl, 4
          shl   bx, cl
          add   bx, ax      ;BX=BX*10H+AX
          jmp   newchar
exit:     ret               ;返回的10进制数的二进制数在BX中
decibin   endp
```

小于30<br>大于39<br>子程序<br>返回

**10进制数以四位2进制数形式保存在BX中**



## •BX中2进制数→16进制数，并在屏幕上显示


```
binihex   proc   near      ；要显示的二进制数在BX中
          mov    ch, 4
rotate:   mov    cl, 4
          rol    bx, cl
          mov    al, bl
          and    al, 0fh
          add    al, 30h
          cmp    al, 3ah
          jl     printit
          add    al, 7h
printit:  mov    dl, al
          mov    ah, 2
          int    21h
          dec    ch
          jnz    rotate
          ret
binihex   endp
```

16进制数转换成ASCII码

调用DOS功能在屏幕上显示1个字符
请参看605页附录4约定

## •显示回车换行

```
crlf    proc   near
        mov    dl, 0dh   ; “回车”的ASCII码=0dH
        mov    ah, 2
        int    21h
        mov    dl, 0ah   ; “换行”的ASCII码=0aH
        mov    ah, 2
        int    21h
        ret
crlf    endp

decihex ends      ; 程序代码在一个代码段中与前边“decihex  segment  ”配对
        end  main      ; 程序从main开始执行
```

# （2）通过存储器传送参数（共享）

- 子程序和调用程序在同一程序模块中，则子程序与主程序一样直接访问数据段中的变量

例6.4 累加数组中的元素

```
data   segment
       ary     dw   1,2,3,4,5,6,7,8,9,10
       count   dw   10
       sum     dw   ?
data   ends
code   segment
main   proc  far
       assume cs:code, ds:data
start:
       push   ds
       sub    ax, ax
       push   ax
       mov    ax, data
       mov    ds, ax
       call   near ptr  proadd
       ret
main   endp
```

```
proadd  proc  near
        push  ax
        push  cx
        push  si
        lea   si, ary
        mov   cx, count
        xor   ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next    ;cx-1≠0循环
        mov   sum, ax
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
code    ends
        end  start
```

该程序中不影响主程序，可以不保存

## 问题：假设数据段定义如下

```
data    segment

ary     dw   1,2,3,4,5,6,7,8,9,10
count   dw   10
sum     dw   ?

ary1    dw   10,20,30,40,50,60,70,80,90
count1  dw   9
sum1    dw   ?

data    ends
```

```
proadd  proc  near
        push  ax
        push  cx
        push  si
        lea   si, ary
        mov   cx, count
        xor   ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next
        mov   sum, ax
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
```

* 由于处理的存储单元在子程序中有固定的约定，不能用同一个子程序proadd。多编写几个子程序？子程序意义何在！

解决办法：

1、设置临时共享参数存放区，调用时主程序先将参数放在临时存放区，子程序处理临时参数存放区中数据

- 参数少时可以，参数多时主程序效率不高

2、调用时主程序只传送变量地址表给子程序

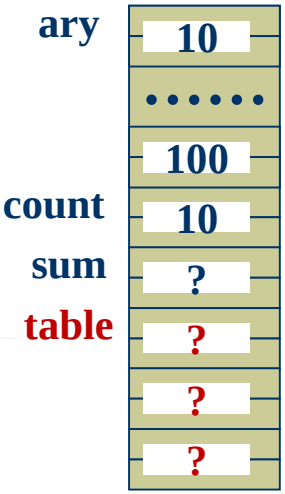
根据参数多少权衡采用方法

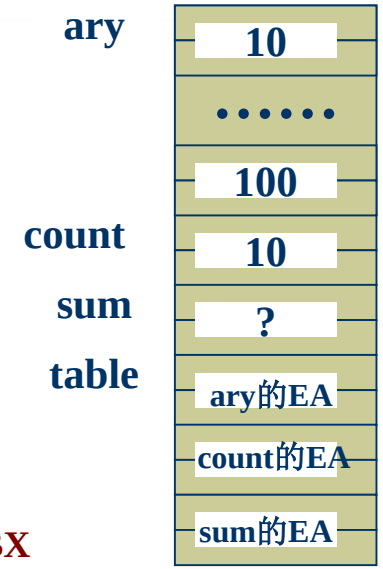
# （3）通过地址表传送变量地址

- **适用于参数较多的情况。**具体方法是先建立一个地址表，该表由参数地址构成。然后把表的首地址通过寄存器或堆栈传递给子程序

例6.4 累加数组中的元素

```
data  segment
    ary     dw  10,20,30,40,50,60,70,80,90,100
    count   dw  10
    sum     dw  ?
    table   dw  3  dup  (?)        ; 地址表，存放ary,count,sum的EA
data  ends

code  segment
    main  proc  far
          assume cs:code, ds:data
    start: push  ds
          sub   ax, ax
          push  ax
          mov   ax, data
          mov   ds, ax
          mov   table,     offset  ary
          mov   table+2,   offset  count
          mov   table+4,   offset  sum
          mov   bx,        offset  table   ;table的EA→BX
          call  proadd
          ret
      main  endp
```




将立即数（变量count的有效地址）送到table+2给出的有效地址的两个存储器单元

请解释mov table+2, offset count的功能

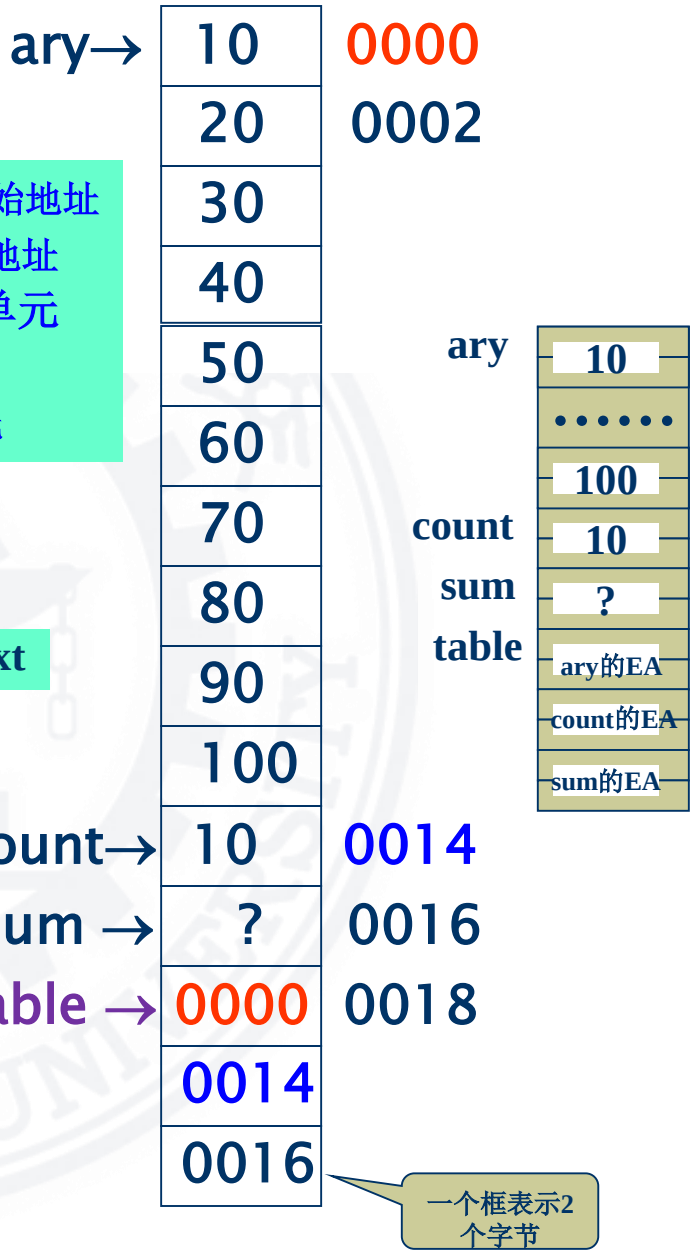
```
proadd  proc  near
        push  ax
        push  cx
        push  si
        push  di
        mov   si, [bx]
        mov   di, [bx+2]
        mov   cx, [di]
        mov   di, [bx+4]
        xor   ax, ax
next:   add   ax, [si]
        add   si, 2
        loop  next
        mov   [di], ax
        pop   di
        pop   si
        pop   cx
        pop   ax
        ret
proadd  endp
code    ends
        end  start
```

table的EA在BX中

bx=0018H指向table启始地址
si=0000H指向ary启始地址
di=0014H指向count单元
cx=10　　计数值count
di=0016H指向sum单元

si指向ary的下一个元素

cx=cx-1; 如cx≠0, 转next



# （4）通过堆栈传送变量或变量地址

- 适用于参数较少，或子程序有多层嵌套、递归调用的情况

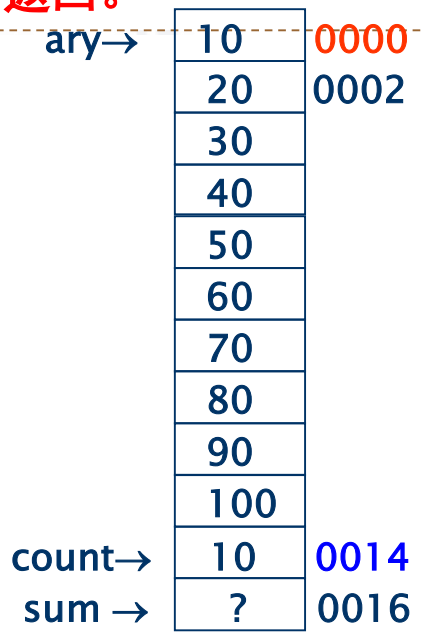
参数/变量地址表大小动态变化、无法确定，使用堆栈

- 步骤：

1. 主程序把参数或参数地址压入堆栈；
2. 子程序使用堆栈中的参数或通过栈中参数地址取到参数；
3. 子程序返回时使用RET n指令调整SP指针，以便删除堆栈中已用过的参数，保持堆栈平衡，保证嵌套子程序的正确返回。

## 例6.4 累加数组中的元素

```
data  segment
      ary     dw   10,20,30,40,50,60,70,80,90,100
      count   dw   10
      sum     dw   ?
data  ends

stack  segment
         dw     100  dup  (?)
  tos   label   word
stack  ends
```

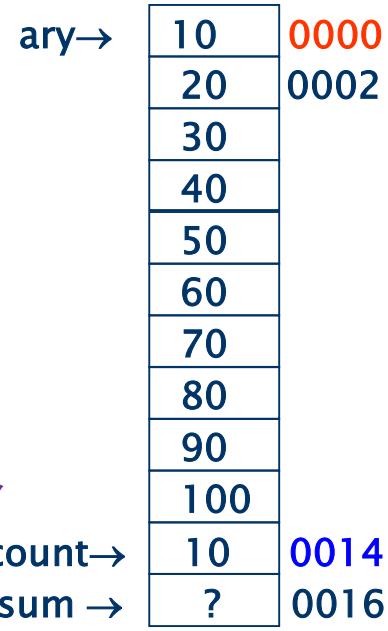


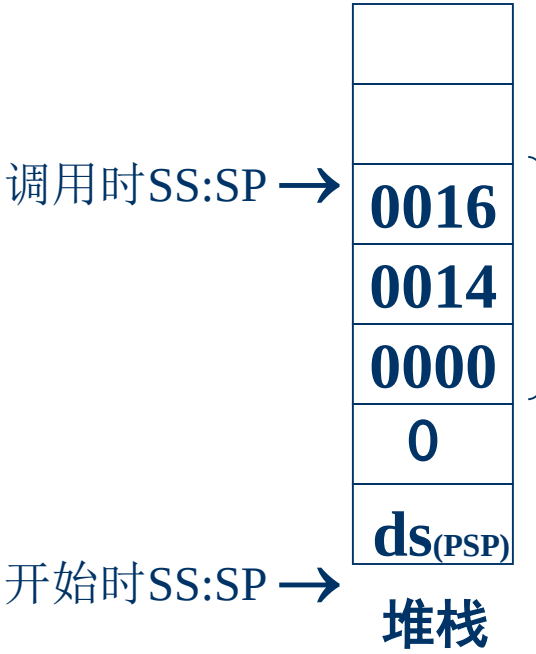

参数少时：传送参数
参数多时：传送参数地址

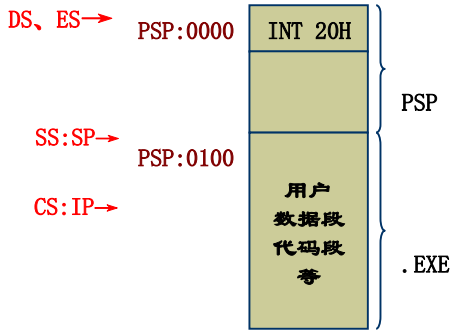
```
code1 segment
main  proc  far
      assume  cs:code1, ds:data, ss:stack
start:
      mov   ax,  stack
      mov   ss,  ax
      mov   sp,  offset  tos
      push  ds
      sub   ax,  ax
      push  ax
      mov   ax,  data
      mov   ds,  ax
      mov   bx, offset  ary
      push  bx
      mov   bx, offset  count
      push  bx
      mov   bx, offset  sum
      push  bx
       call   far ptr  proadd
       ret
main   endp
code1 ends
```

```
stack  segment
       dw     100  dup  (?)
 tos   label   word
stack  ends
```

栈指针设置最好用LSS SP,[1000H]指令

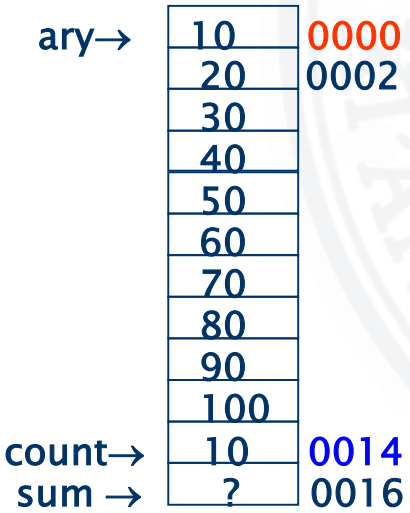


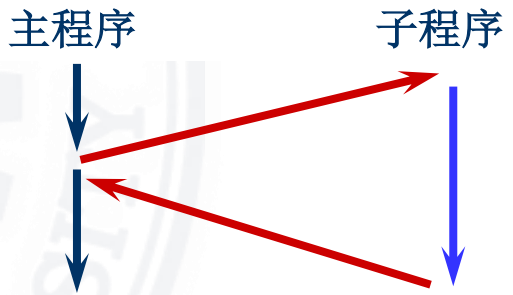
```
code2 segment
   assume  cs: code2
proadd  proc  far

      push  bp
      mov   bp, sp
      mov   si, [bp+0ah]  ;  (SS:[bp+0ah])=0000→si, si指向ary首址
      mov   di, [bp+8]    ;  (SS:[bp+08h])=0014→di, di指向count首址
      mov   cx, [di]      ;  (DS:[di])=10→cx,   cx是数组长度
      mov   di, [bp+6]    ;  (SS:[bp+06h])=0016→di, di指向sum首址
```

默认段寄存器是什么?
[bp+0ah]=SS:[bp+0ah]





一个框表示2个字节



```
code2 segment
  assume  cs: code2
proadd  proc  far

      push  bp
      mov   bp, sp

      push  ax
      push  cx
      push  si
      push  di

      mov   si, [bp+0ah]
      mov   di, [bp+8]
      mov   cx, [di]
      mov   di, [bp+6]
      xor   ax, ax
next: add   ax, [si]
      add   si, 2
      loop  next
      mov   [di], ax

      pop   di
      pop   si
      pop   cx
      pop   ax

      pop   bp

      ret   6
proadd  endp
code2   ends
        end  start
```

ary→

| | |
|---|---|
| 10 | 0000 |
| 20 | 0002 |
| 30 | |
| 40 | |
| 50 | |
| 60 | |
| 70 | |
| 80 | |
| 90 | |
| 100 | |
| 10 | 0014 |
| ? | 0016 |

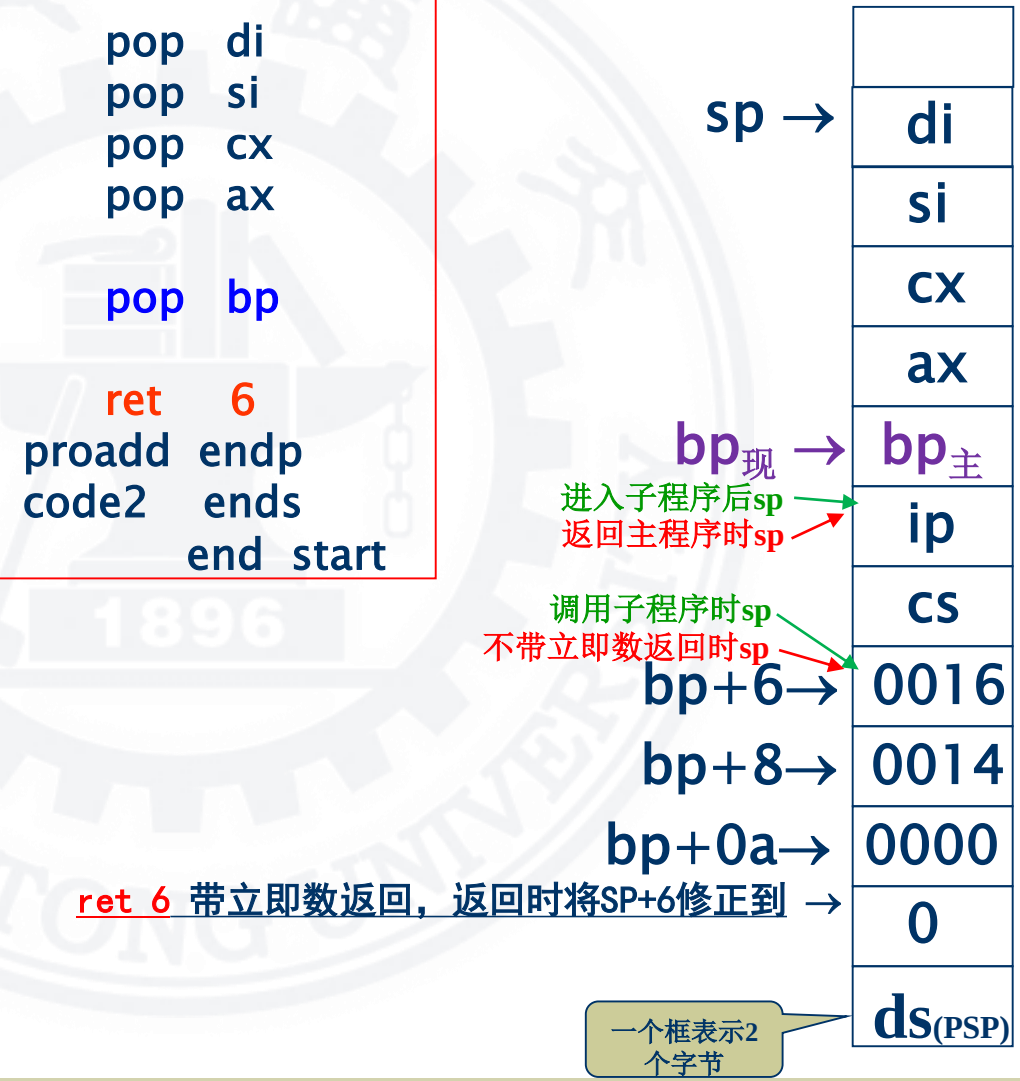
count→

sum →

## 进栈操作

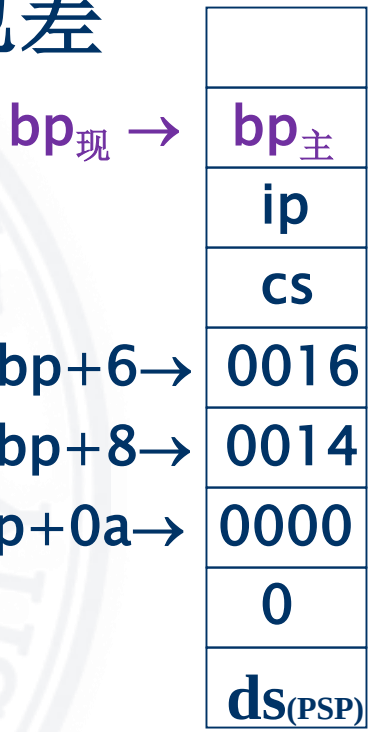
**执行操作：** SP ← SP - 2

( SS*16+SP+1, SS*16+SP ) ← SRC



mov si, [bp+0ah]

用0ah编程不方便不灵活，可读性也差




解决方法：

① 定义 arryad equ 0ah

然后 mov si, [bp+arryad]

② 定义结构体，然后使用结构体的名字段

mov si, [bp].par1_addr

■ **结构伪操作STRUC**：只是定义一种可包含不同类型数据的结构模式，只有具体使用时才有对应存储单元的具体含义

格式：

```
结构名  STRUC
字段名1   DB   ?
字段名2   DW   ?
字段名3   DD   ?
    ……
结构名  ENDS
```

对一组不同类型数据的结构格式定义

□字段名就是变量名，可用变量名表示字段起始地址

例：学生个人信息

```
STUDENT_DATA   STRUC     ; 4个字段,18个字节的结构模式
    NAME       DB    5 DUP(?)
    ID         DW    0
    AGE        DB    ?
    DEP        DB   10 DUP(?)
STUDENT_DATA   ENDS
```

说明了字段名数据的数据类型及其与结构体首址的位移量

■ **结构预置语句**：为结构中各字段的数据分配存储器单元，并可为存储单元重新输入字符串和数值

格式1： 变量名 结构名 < >

•采用结构定义中的赋值

格式2： 变量名 结构名 <预赋值说明>

•重新定义结构中的值

```
STUDENT_DATA  STRUC
  NAME    DB   5 DUP(?)
  ID      DW   0
  AGE     DB   ?
  DEP     DB  10 DUP(?)
STUDENT_DATA  ENDS
```

数据段定义中使用

例：S991000 STUDENT_DATA < >

```
等同于数据段中如下定义：
  S991000.NAME   DB   5 DUP(?)
  S991000.ID     DW   0
  S991000.AGE    DB   ?
  S991000.DEP    DB  10 DUP(?)
```

S991001 STUDENT_DATA < ,1001,22, >

```
S991000.NAME   DB   5 DUP(?)
S991000.ID     DW   1001
S991000.AGE    DB   22
S991000.DEP    DB  10 DUP(?)
```

STUDENT STUDENT_DATA 100 DUP (< >)

■ **访问结构数据变量方法**：

```
MOV  AL, S991000.NAME[SI]
MOV  AL, [BX].NAME[SI]
```

.name可以理解为相对于结构首址的位移量
Bx中存的是结构首址， si给出name字段的第几项
[BX].NAME[SI]= [BX+.NAME+SI]

## 结构伪操作举例：

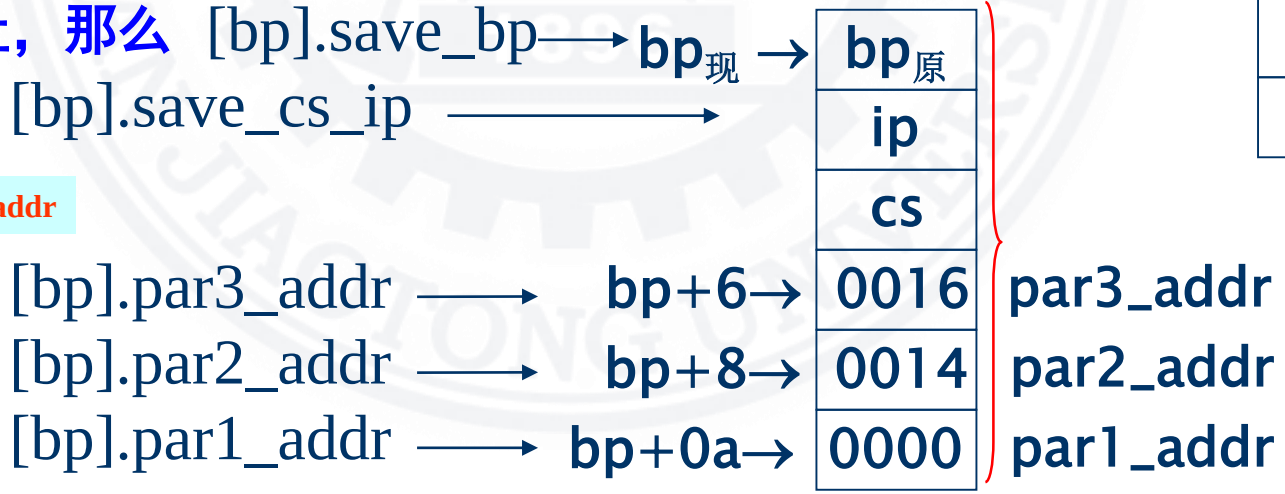
改写例6.4 累加数组中的元素

```
stack_strc      struc

    save_bp     dw    ?
    save_cs_ip  dw    2 dup (?)
    par3_addr   dw    ?
    par2_addr   dw    ?
    par1_addr   dw    ?

stack_strc      ends
```

定义这个存储数据格式为结构，便于访问编程


## 结构伪操作定义相当于存储结构格式说明

让bp指向结构首址，那么 [bp].save_bp ⟶ bp现 →




指令中使用

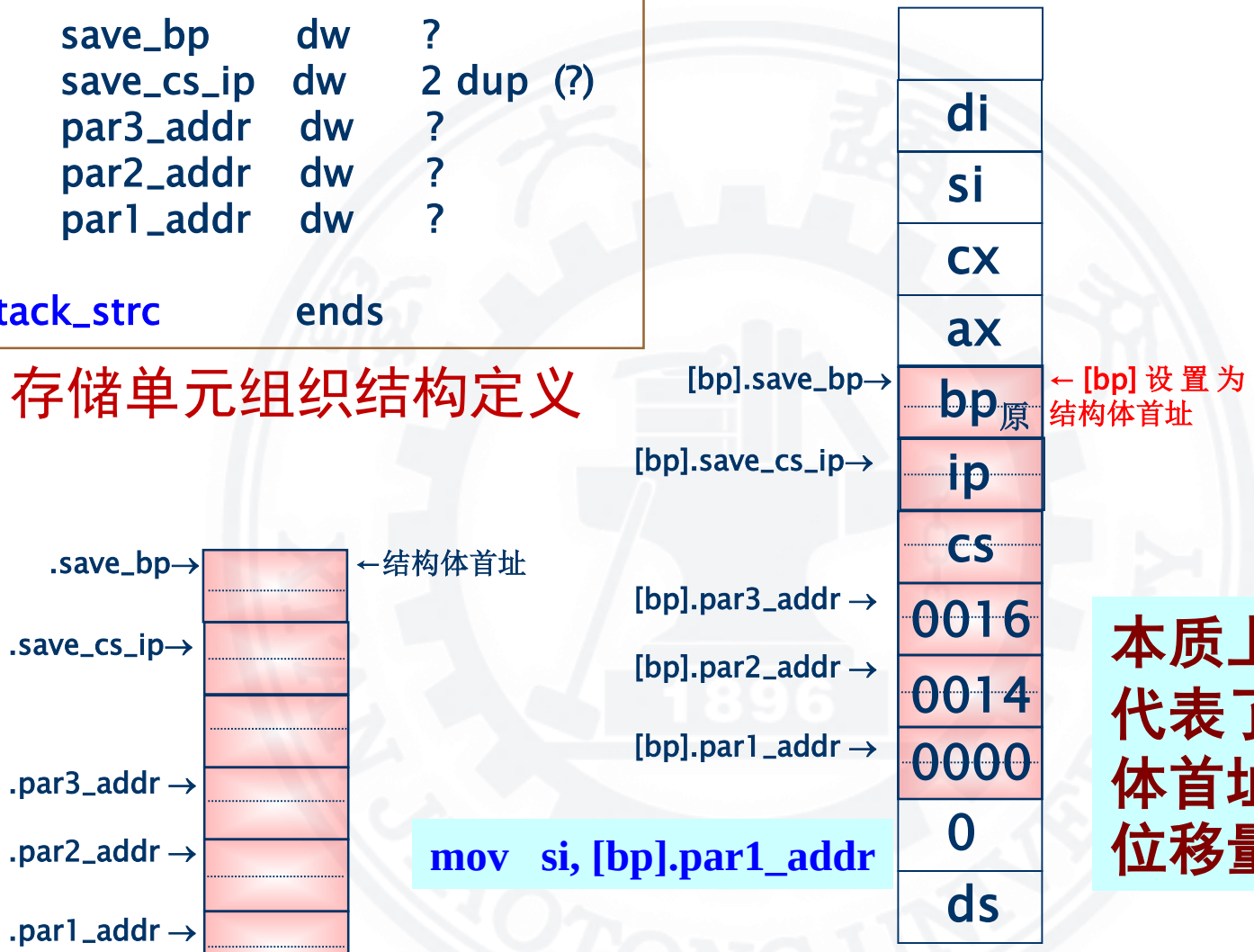
```
mov   si, [bp].par1_addr
```

汇编后

```
mov   si, [bp+0a]
```

```
stack_strc      struc

    save_bp     dw    ?
    save_cs_ip  dw    2 dup (?)
    par3_addr   dw    ?
    par2_addr   dw    ?
    par1_addr   dw    ?

stack_strc      ends
```

# 存储单元组织结构定义




**mov si, [bp].par1_addr**

**本质上.XXX代表了到结构体首址的一个位移量**

**[bp].save_bp = [bp+.save_bp]**

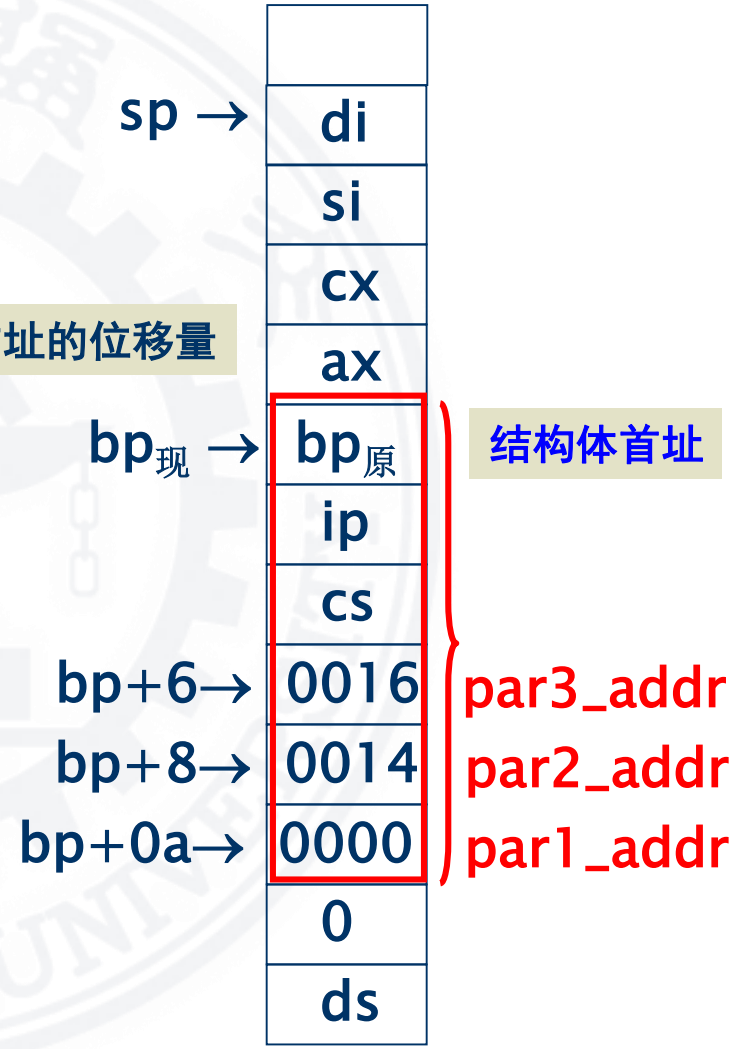
**汇编工具汇编是自动用[bp+00H]替代[bp].save_bp**

```
stack_strc      struc
    save_bp     dw    ?
    save_cs_ip  dw    2 dup  (?)
    par3_addr   dw    ?
    par2_addr   dw    ?
    par1_addr   dw    ?
stack_strc      ends

proadd proc  far
    push  bp
    mov   bp, sp
    push  ax
    push  cx
    push  si
    push  di
    mov   si, [bp].par1_addr
    mov   di, [bp].par2_addr
    mov   cx, [di]
    mov   di, [bp].par3_addr
    xor   ax, ax
next:
    add   ax, [si]
    add   si, 2
    loop  next
    mov   [di], ax
    pop   di
    pop   si
    pop   cx
    pop   ax
    pop   bp
    ret   6
proadd  endp
```

**结构伪操作定义相当于数据存储结构格式和相关变量指针说明**
**编程、阅读、修改方便**

bp指向结构体首址

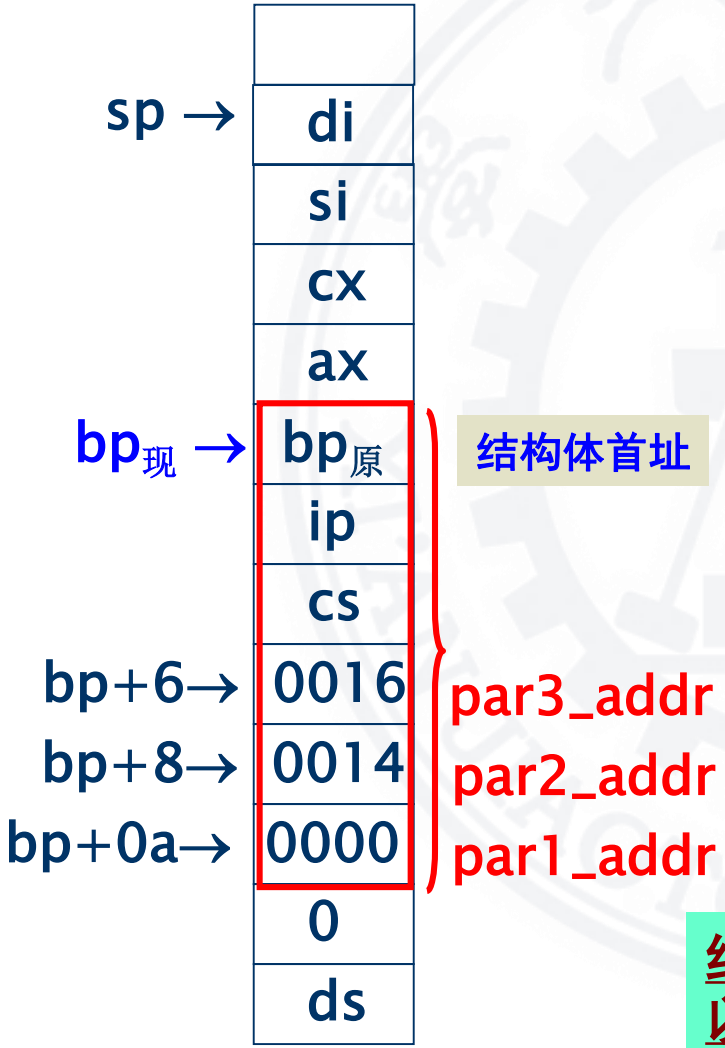
.par1_addr可以理解为相对于结构首址的位移量

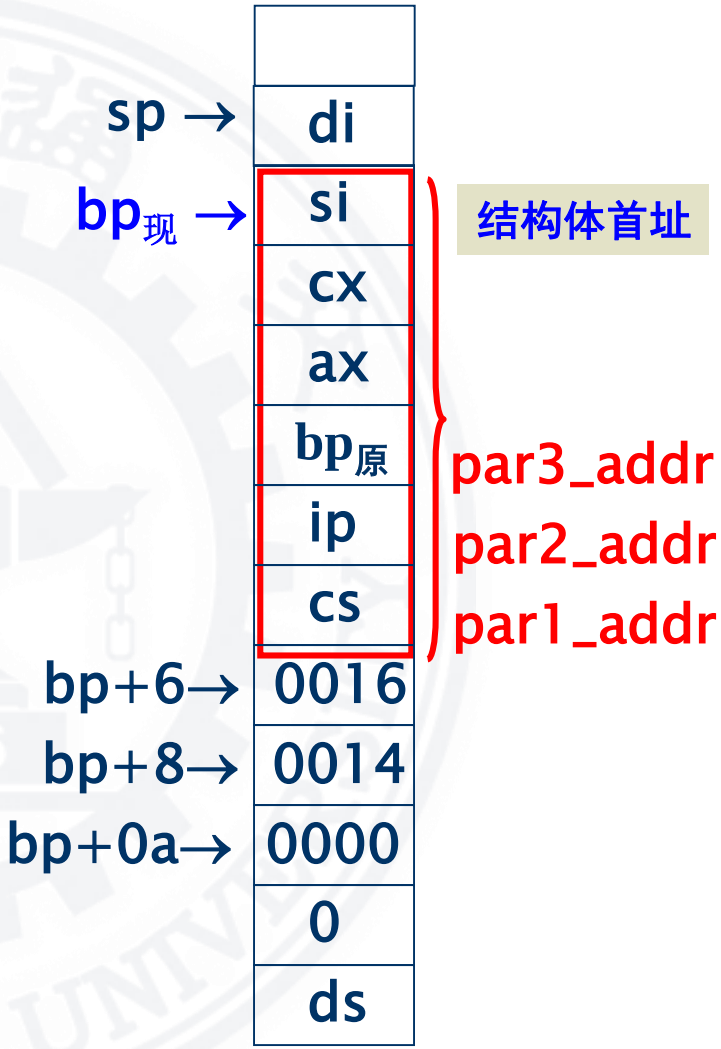


# 注意正确设置结构体首址

**bp指向结构体首址**

```
mov   si, [bp].par1_addr
mov   di, [bp].par2_addr
mov   di, [bp].par3_addr
```



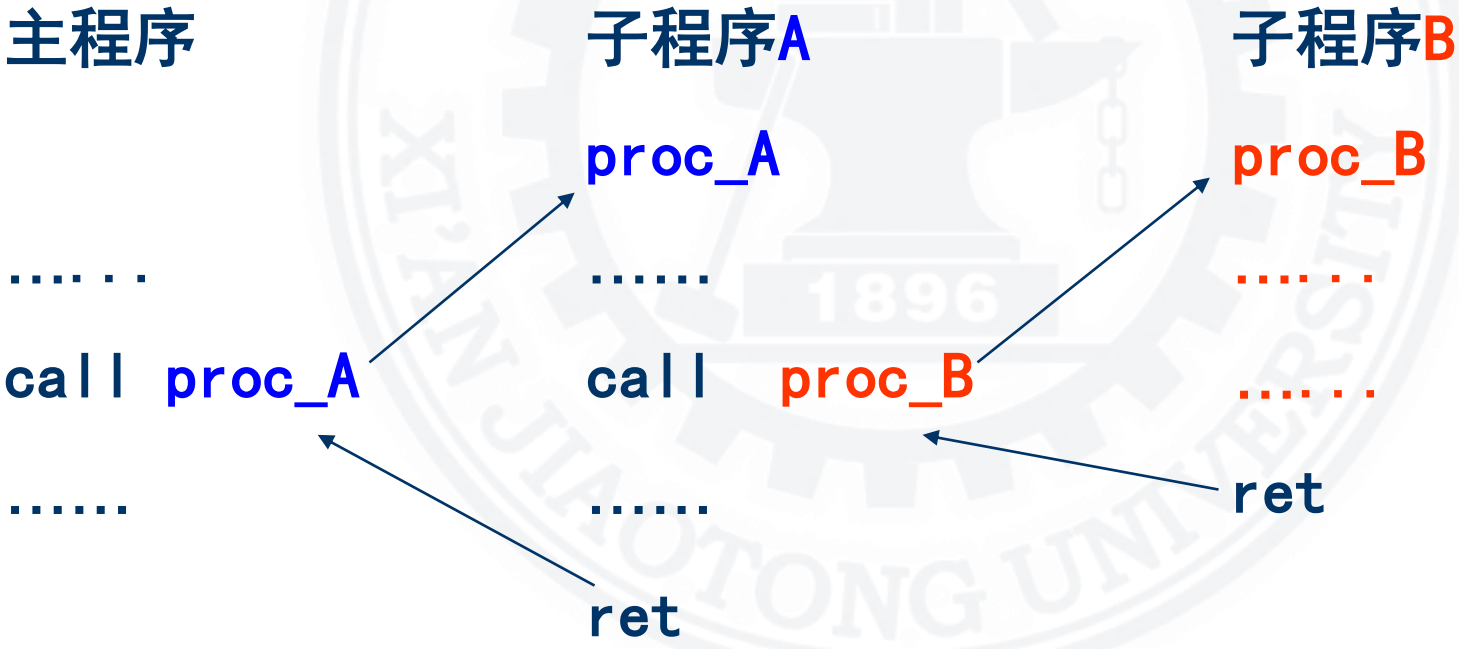




**结构伪操作只是定义了数据存储的结构格式以及相关变量指针说明**

# 6.2 嵌套与递归子程序

## 6.2.1 子程序的嵌套

■子程序嵌套：一个子程序作为调用程序调用另一个子程序



# 6.2.1 递归子程序

- 递归子程序
  - 递归调用：子程序调用的子程序是它自身
  - 递归子程序：递归调用中的子程序
    - 是子程序嵌套的特殊情况
- 嵌套深度：是嵌套的层次，层次不限

但实际中由堆栈大小和现场保护情况决定

  - 堆栈大小是嵌套深度的关键因素，特别是递归调用
  - 特别注意堆栈状态和正确使用
- 注意事项同一般子程序调用，但要特别注意子程序结束返回条件

- 堆栈大小由嵌套子程序调用等需要确定
- 堆栈大小确定后，程序对堆栈使用不能超出

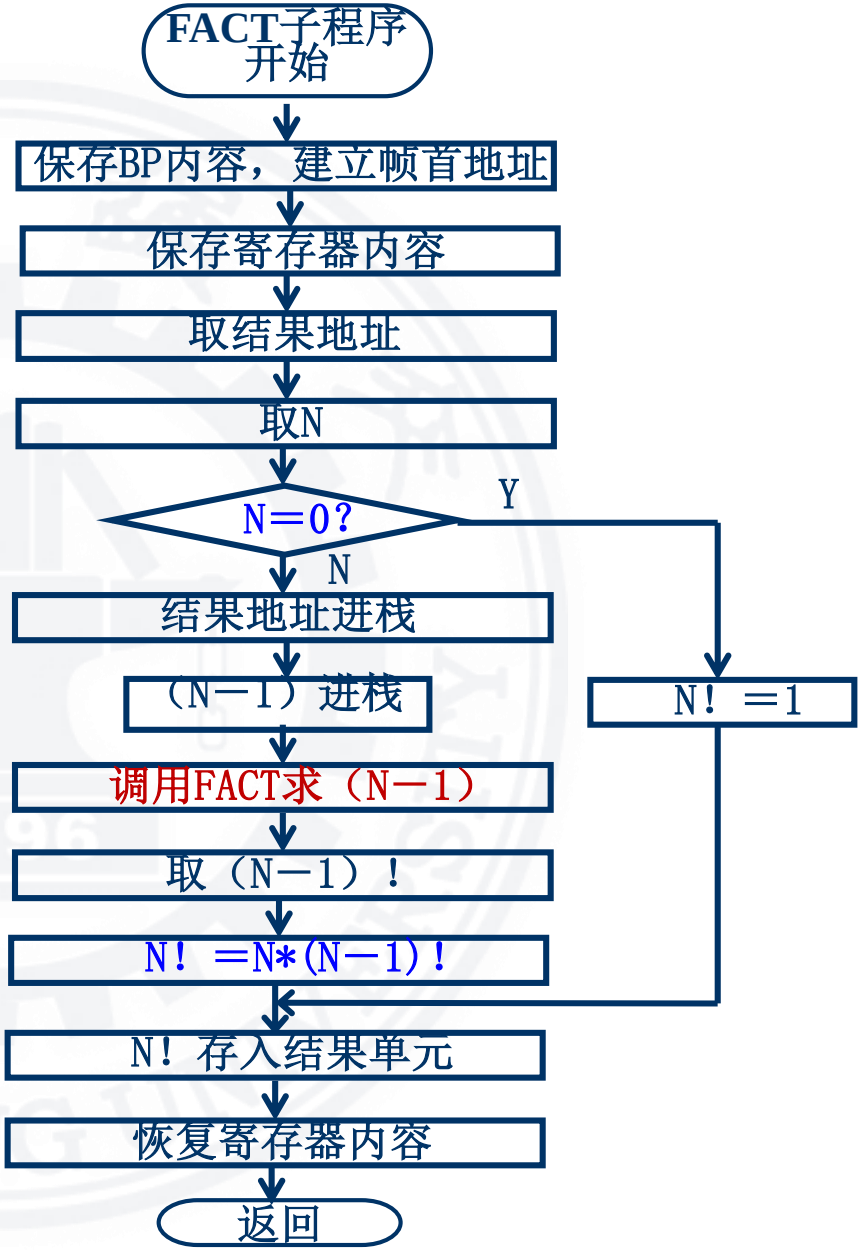
# 例6.7 计算N! (N≥0)

N!=N× (N-1) ×(N-2) × … × 1

递归定义：
$$\begin{cases} 0! = 1 \\ N! = N \times (N-1)! \quad N>0 \end{cases}$$


图6.7

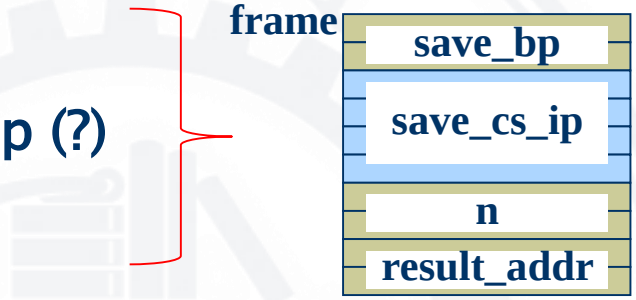
3!=3*2!
2!=2*1!
1!=1*0!
0! =1

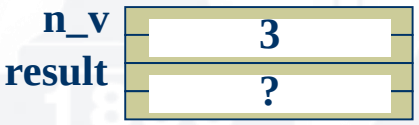


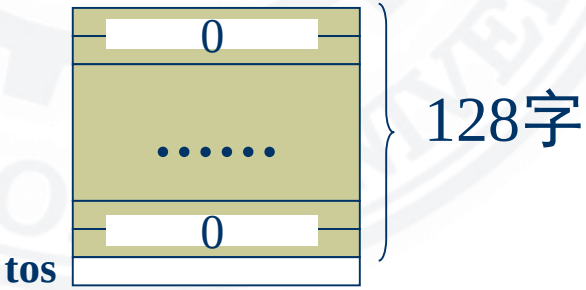
# 例6.7 计算n! 假设n=3

用堆栈、寄存器或存储单元返回数据

```
frame  struc
       save_bp      dw    ?
       save_cs_ip   dw    2 dup (?)
       n            dw    ?
       result_addr  dw    ?
frame  ends

data   segment
n_v    dw    3
result dw    ?
data   ends

stack  segment
       dw  128  dup (0)
   tos label  word
stack  ends
```




3!=3*2!

2!=2*1!

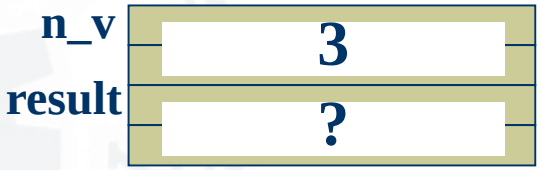
1!=1*0!

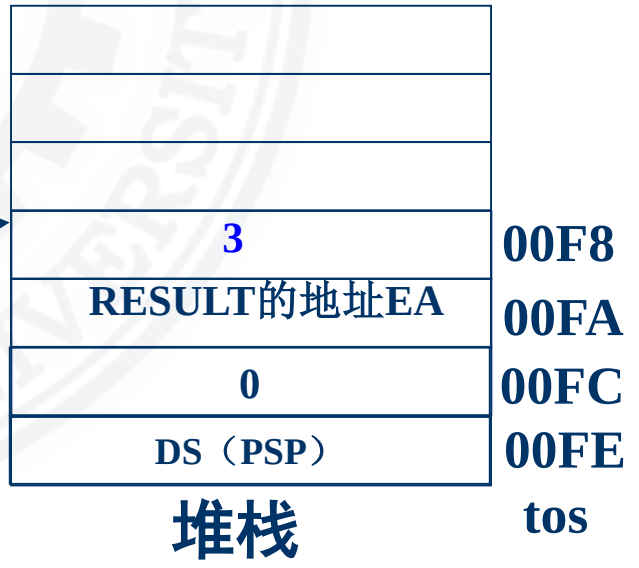
0!=1

```
code segment
main  proc  far
        assume  cs:code, ds:data, ss:stack
start:
      mov  ax, stack
      mov  ss, ax
      mov  sp, offset tos
      push  ds
      sub  ax, ax
      push  ax
      mov  ax, data
      mov  ds, ax
      mov  bx, offset result
      push  bx
      mov  bx, n_v
      push  bx
      call  far  ptr  fact
      ret
main endp
code  ends
```

```
stack   segment
        dw  128  dup (0)
   tos  label  word
stack   ends
```

```
data    segment
n_v     dw    3
result  dw    ?
data    ends
```

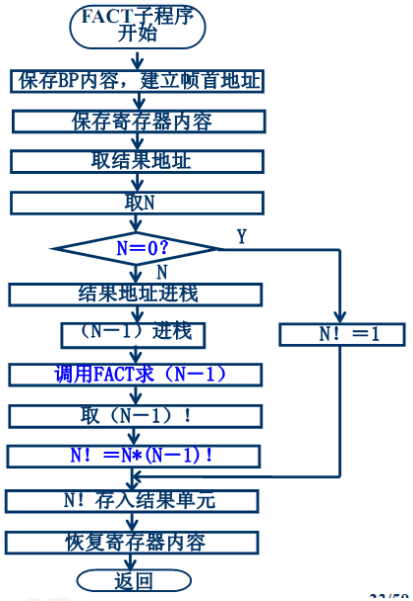



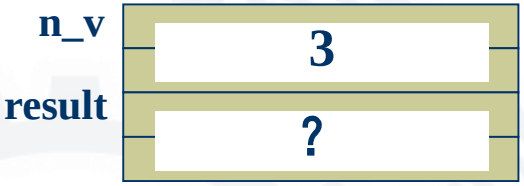




为了子程序通用，
将结果地址和N传给子程序

```
code1  segment
       assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

```
frame  struc
       save_bp     dw     ?
       save_cs_ip  dw     2 dup (?)
       n           dw     ?
       result_addr dw     ?
frame  ends
```



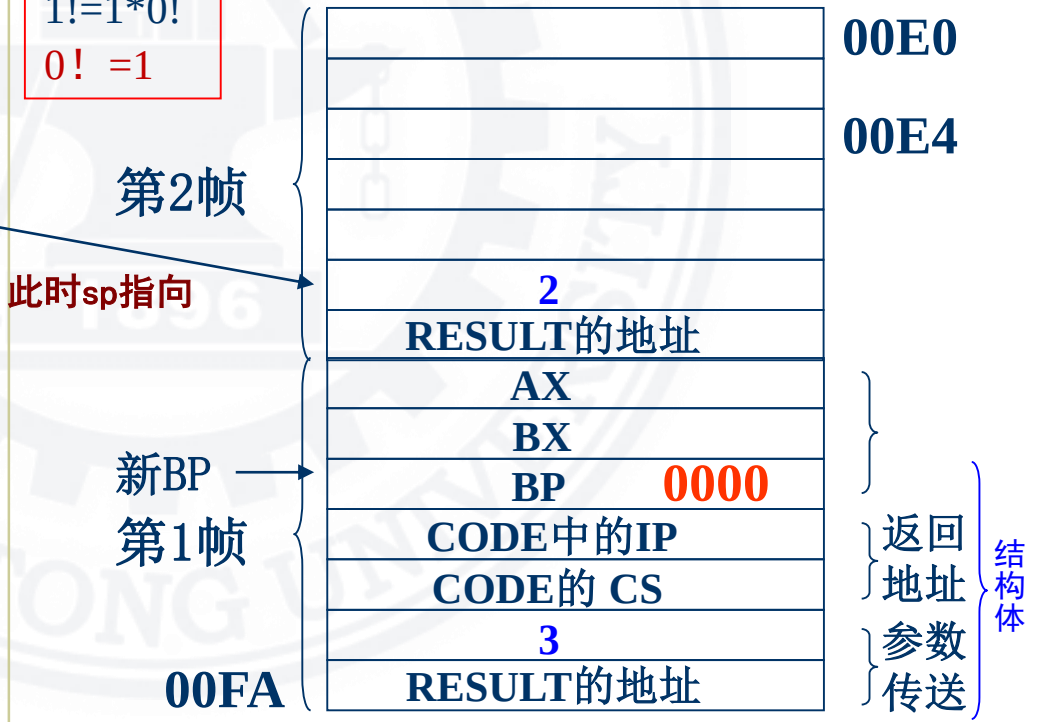




3!=3*2!

2!=2*1!

1!=1*0!

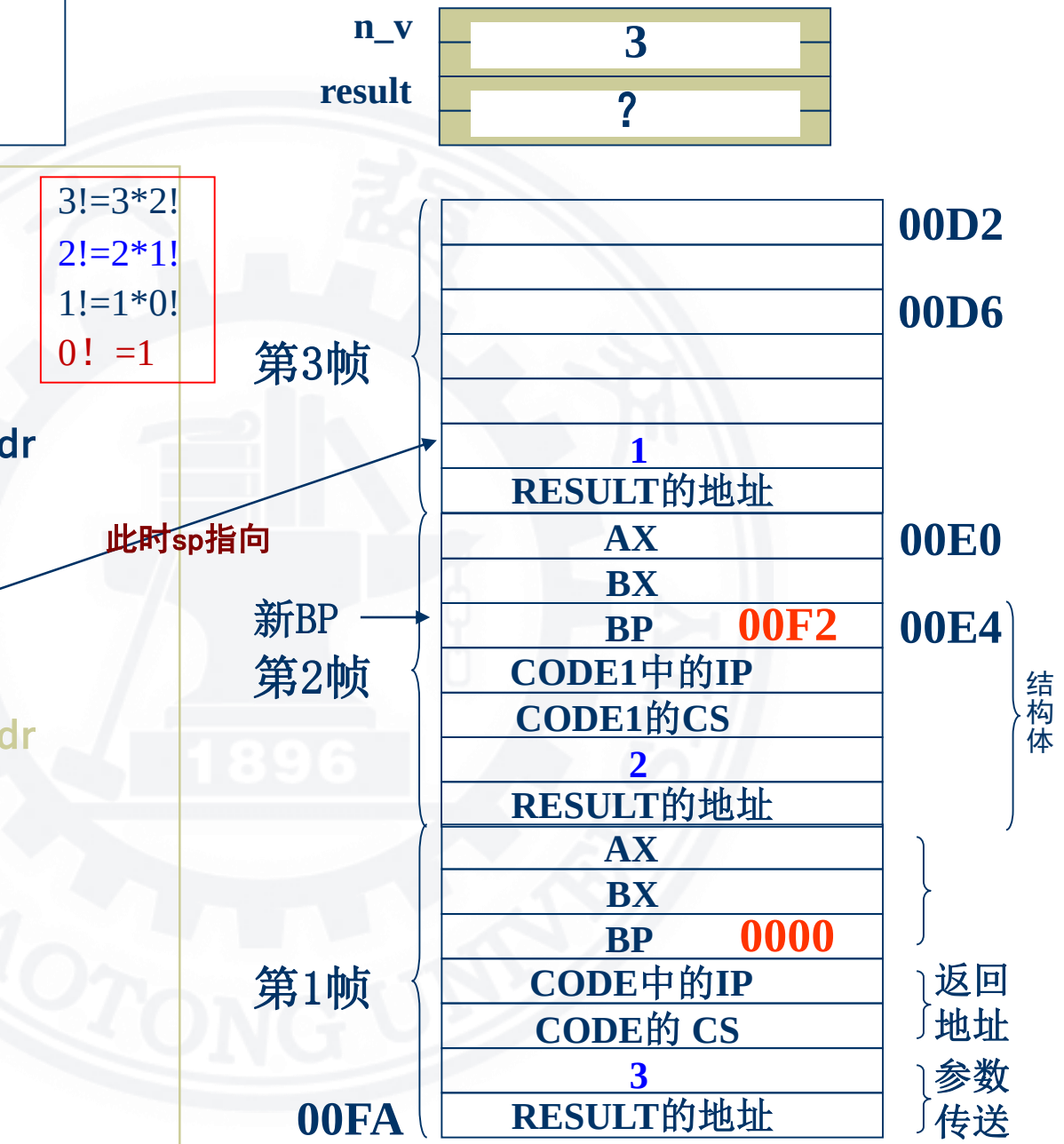
0! =1



```
frame  struc
       save_bp      dw    ?
       save_cs_ip   dw    2 dup (?)
       n            dw    ?
       result_addr  dw    ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

3!=3*2!
2!=2*1!
1!=1*0!
0！ =1



```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1  ends
```

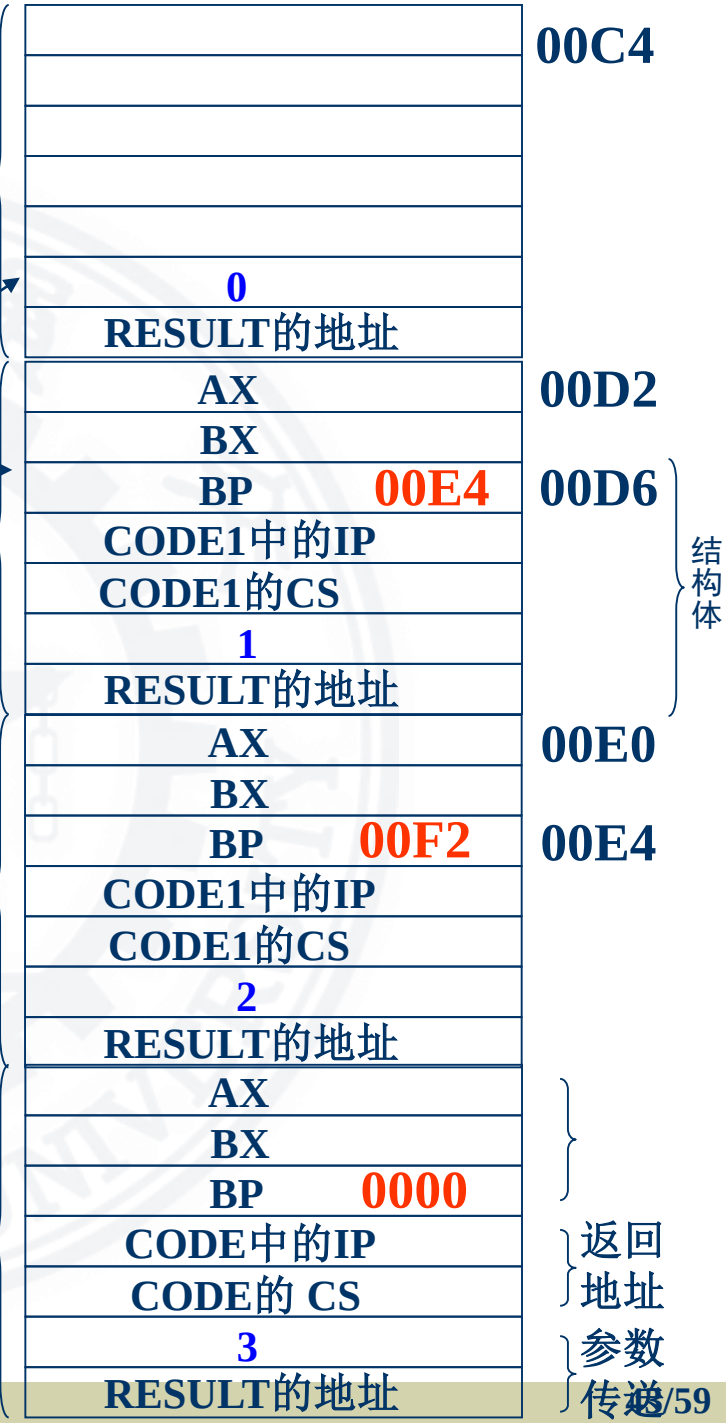
3!=3*2!

2!=2*1!

1!=1*0!

0！ =1

此时sp指向




| | |
|---|---|
| n_v | 3 |
| result | ? |

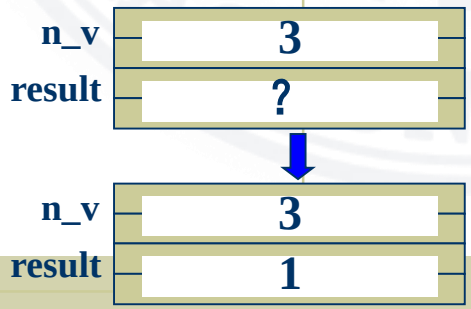
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```

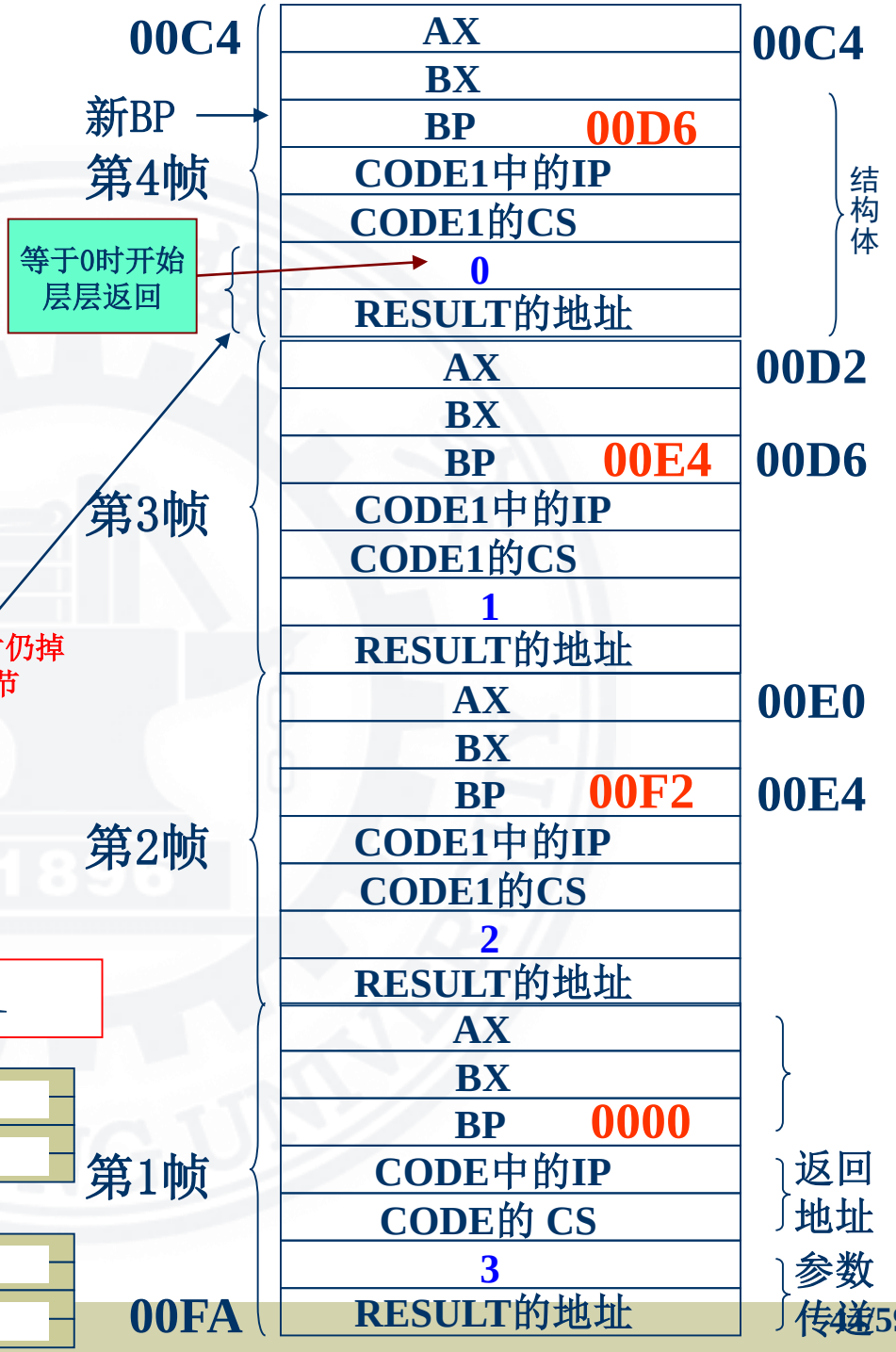
```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec     ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

3!=3*2!
2!=2*1!
1!=1*0!
0! =1

0!=1

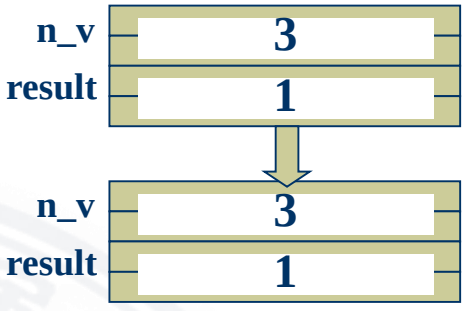
返回时仍掉4个字节



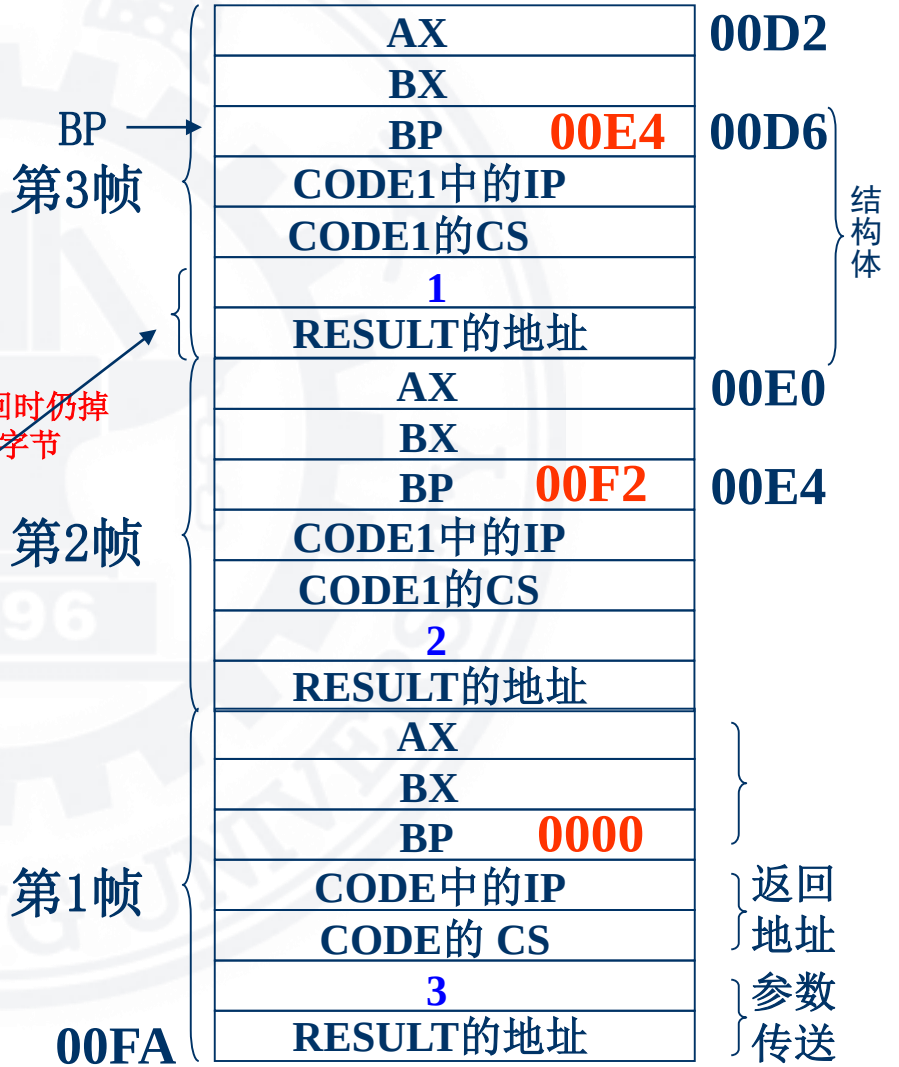
```
frame  struc
       save_bp      dw    ?
       save_cs_ip   dw    2 dup (?)
       n            dw    ?
       result_addr  dw    ?
frame  ends
```

```
code1  segment
       assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je    done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n        ;ax=1!=1*0!
       jmp   short  return
done:  mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1  ends
```



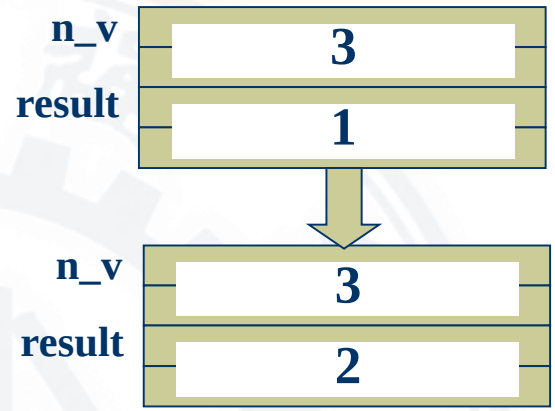

0!=1

1!=1*0!



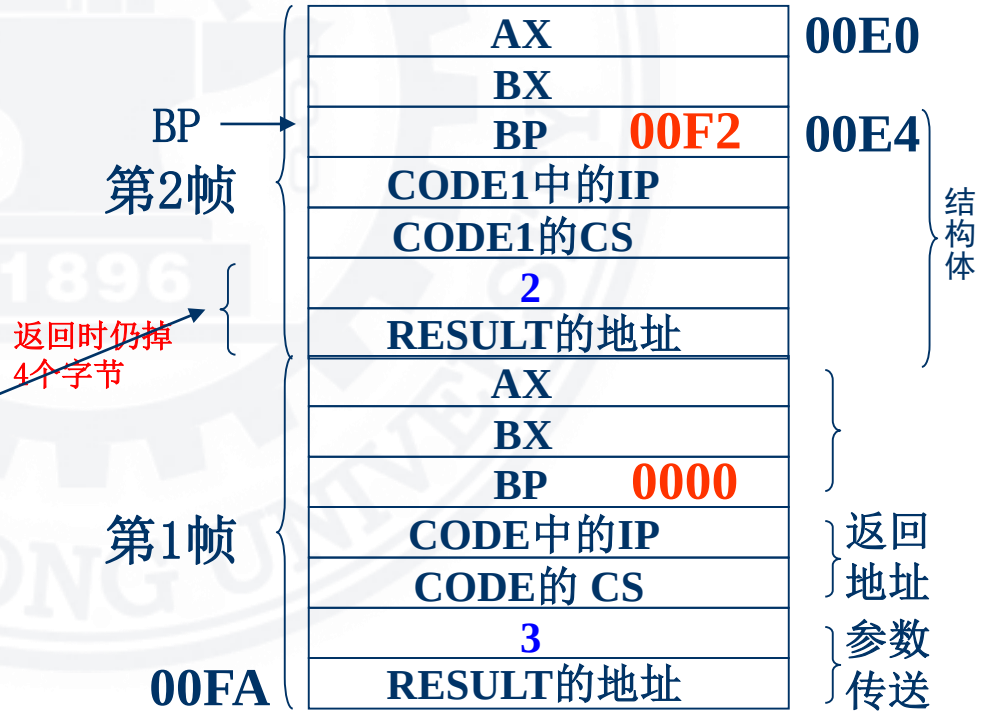
```
code1  segment
        assume  cs:code1
fact   proc  far
        push  bp
        mov   bp, sp
        push  bx
        push  ax
        mov   bx, [bp].result_addr
        mov   ax, [bp].n
        cmp   ax, 0
        je      done
        push  bx
        dec    ax
        push  ax
        call  far ptr  fact
        mov   bx, [bp].result_addr
        mov   ax, [bx]
        mul   [bp].n      ;ax=2!=2*1!
        jmp   short  return
done: mov   ax,1
return:
        mov   [bx], ax
        pop   ax
        pop   bx
        pop   bp
        ret   4
fact   endp
code1   ends
```

```
frame  struc
       save_bp     dw    ?
       save_cs_ip  dw    2 dup (?)
       n           dw    ?
       result_addr dw    ?
frame  ends
```




1!=1*0!

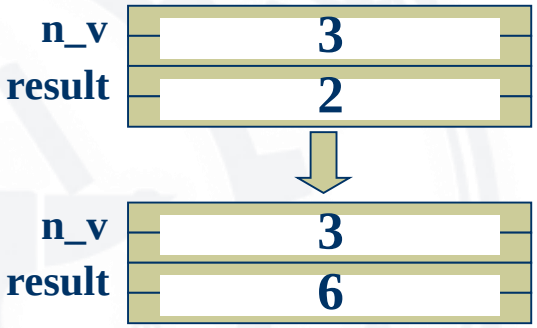
2!=2*1!



```
code1  segment
       assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je    done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n        ;ax=3!=3*2!
       jmp   short  return
done: mov    ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

返回时仍掉4个字节

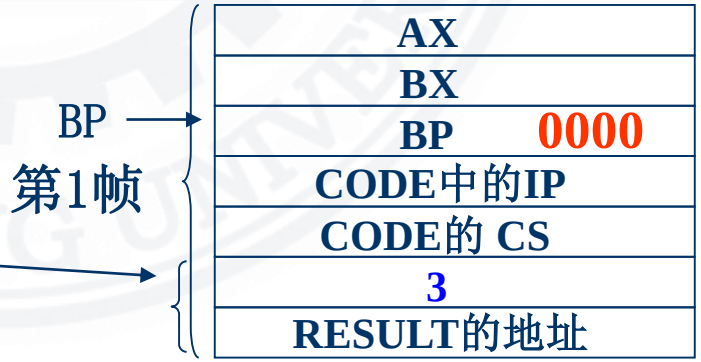
```
frame  struc
       save_bp      dw     ?
       save_cs_ip   dw     2 dup (?)
       n            dw     ?
       result_addr  dw     ?
frame  ends
```



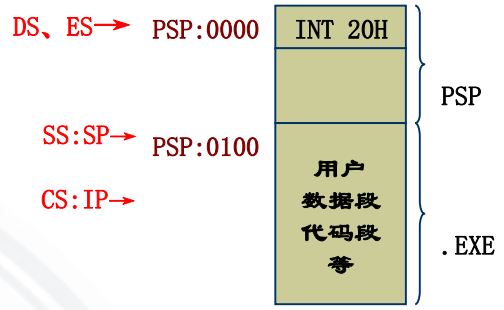

2!=2*1!

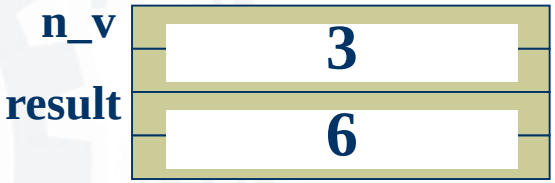
3!=3*2!

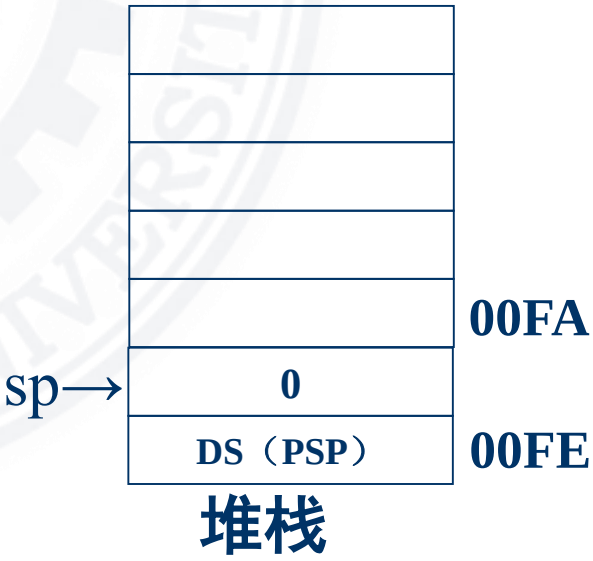
子程序存在隐患：没保存DX



```
code segment
main  proc  far
        assume  cs:code, ds:data, ss:stack
start:
      mov  ax, stack
      mov  ss, ax
      mov  sp, offset tos
      push  ds
      sub  ax, ax
      push  ax
      mov  ax, data
      mov  ds, ax
      mov  bx, offset result
      push  bx
      mov  bx, n_v
      push  bx
      call  far  ptr  fact
      ret
main endp
code  ends
```

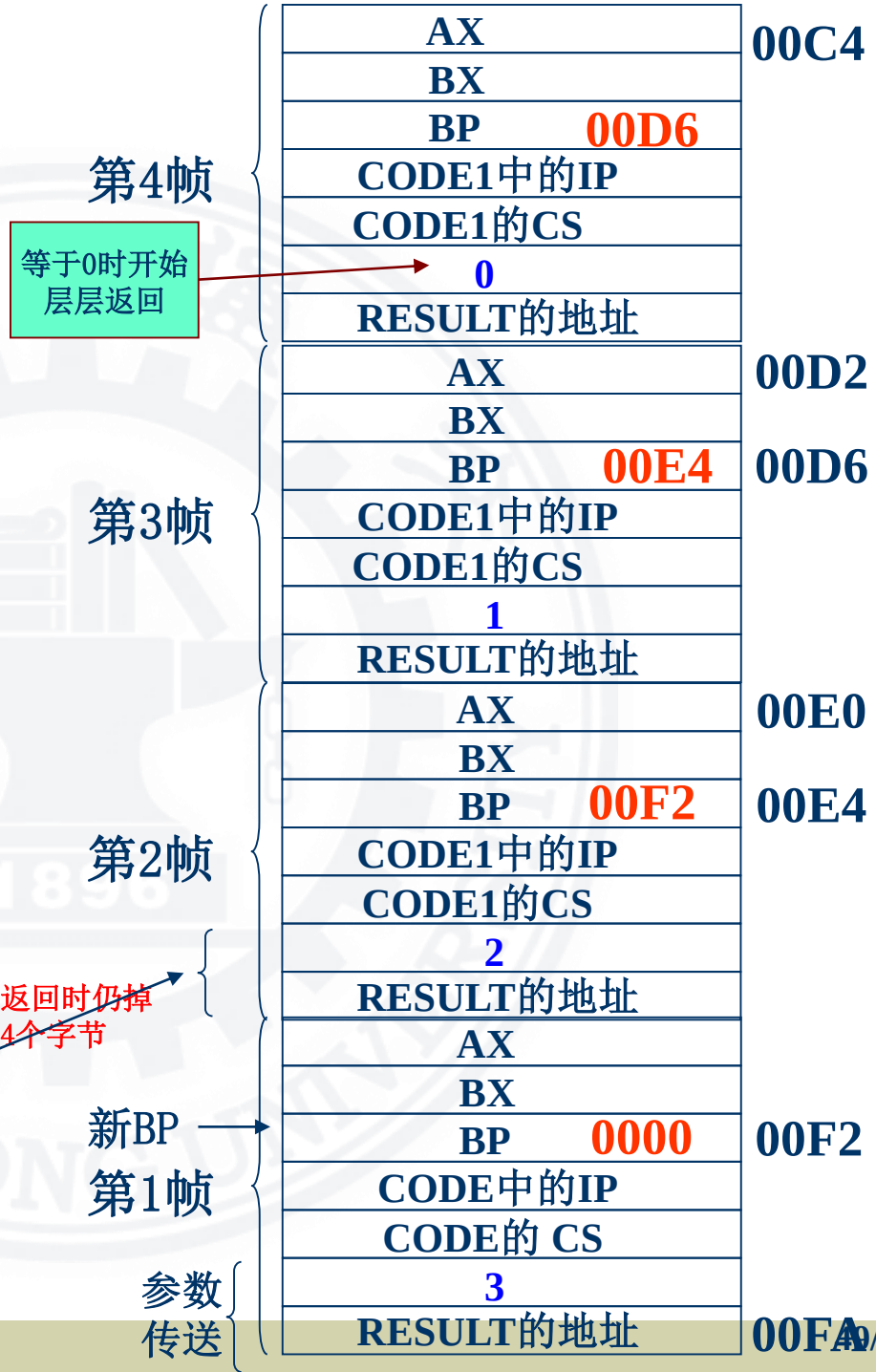


```
frame  struc
       save_bp      dw    ?
       save_cs_ip   dw    2 dup (?)
       n            dw    ?
       result_addr  dw    ?
frame  ends
```

```
code1  segment
         assume  cs:code1
fact   proc  far
       push  bp
       mov   bp, sp
       push  bx
       push  ax
       mov   bx, [bp].result_addr
       mov   ax, [bp].n
       cmp   ax, 0
       je      done
       push  bx
       dec   ax
       push  ax
       call  far ptr  fact
       mov   bx, [bp].result_addr
       mov   ax, [bx]
       mul   [bp].n
       jmp   short  return
done: mov   ax,1
return:
       mov   [bx], ax
       pop   ax
       pop   bx
       pop   bp
       ret   4
fact   endp
code1   ends
```

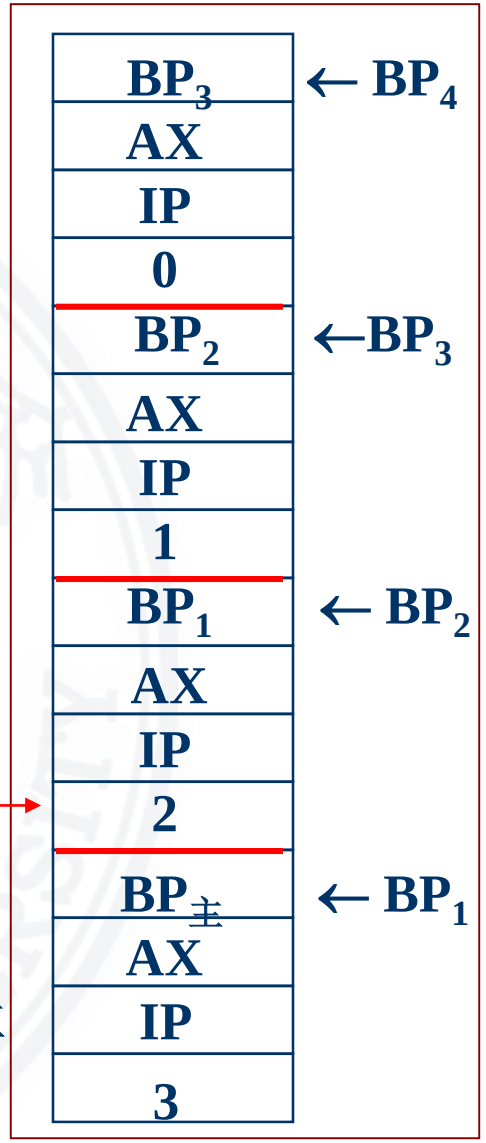


# 例6.7 计算n! 不使用STRUC定义

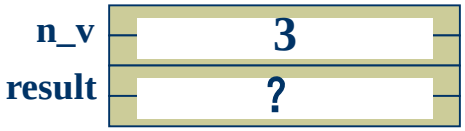
```
mov  bx, n
push bx
call fact
pop  result
```

主程序部分

```
fact  proc near
      push ax    ;可以不保存ax，为什么？
      push bp
      mov  bp, sp
      mov  ax, [bp+6]
      cmp  ax, 0
      jne  fact1
      inc  ax    ;比mov ax, 1效率高
      jmp  exit
fact1: dec  ax
      push ax
      call fact
      pop  ax
      mul  word ptr[bp+6]
exit: mov  [bp+6], ax ；假设结果16位
      pop  bp
      pop  ax
      ret
fact  endp
```




传递参数与返回结果共用该单元




使用STRUC定义，结构清晰，不易出错，修改方便！

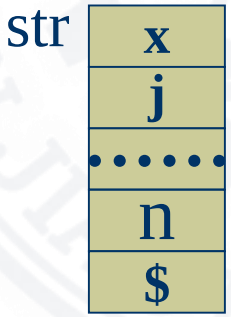
# 例：将字符串反序输出

字符串长度未知，但以“$”结束

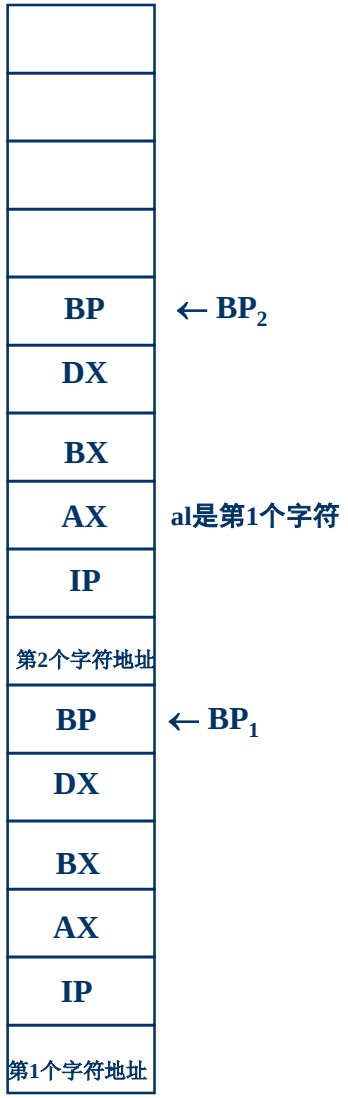
```
mov   bx, offset str
push  bx
call  revers
```

主程序部分代码

试画出当字符串为abc$时堆栈变化情况



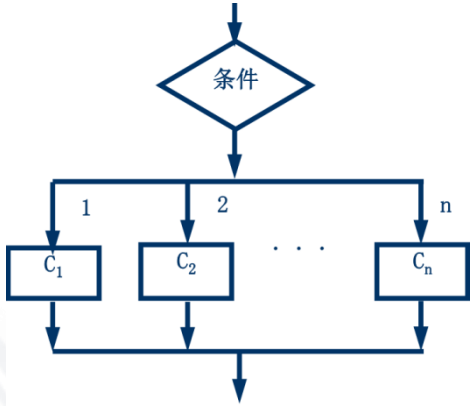

```
revers  proc  near
        push  ax
        push  bx
        push  dx
        push  bp
        mov   bp, sp
        mov   bx, [bp+10]
        mov   al, [bx]
        cmp   al, '$'
        jne   re_call
        jmp   return
re_call:
        inc   bx
        push  bx
        call  revers
        mov   dl, al
        mov   ah, 2
        int   21h
return:
        pop   bp
        pop   dx
        pop   bx
        pop   ax
        ret 2
revers  endp
```



# 6.4 DOS系统功能调用

- 系统功能调用是DOS为系统程序员及用户提供的一组常用功能程序
  - 用户可在程序中调用DOS提供的功能
- DOS规定用INT 21H中断指令作为进入各功能调用程序的总入口，再为每个功能调用规定一个功能号，以便进入相应选择各子功能程序的入口。
- DOS系统功能调用的分类：

  设备管理、文件管理、目录管理




- DOS系统功能调用的使用方法（约定）：
  - ① 在AH寄存器中存入所要调用功能的功能号
  - ② 根据所调用功能的规定设置入口参数
  - ③ 用INT 21H指令转入DOS系统功能调用程序入口
  - ④ 相应的功能程序运行完后，可以按规定取得出口参数
- 一般调用格式：
  - ① 设置调用参数
  - ② MOV AH，功能号
  - ③ INT 21H
  - ④ 取返回参数

```
MOV AH, 1   ；键盘输入并回显
INT 21H
```

- 简单举例：参看P605 附录四——键盘输入单个字符，显示器输出单个字符等

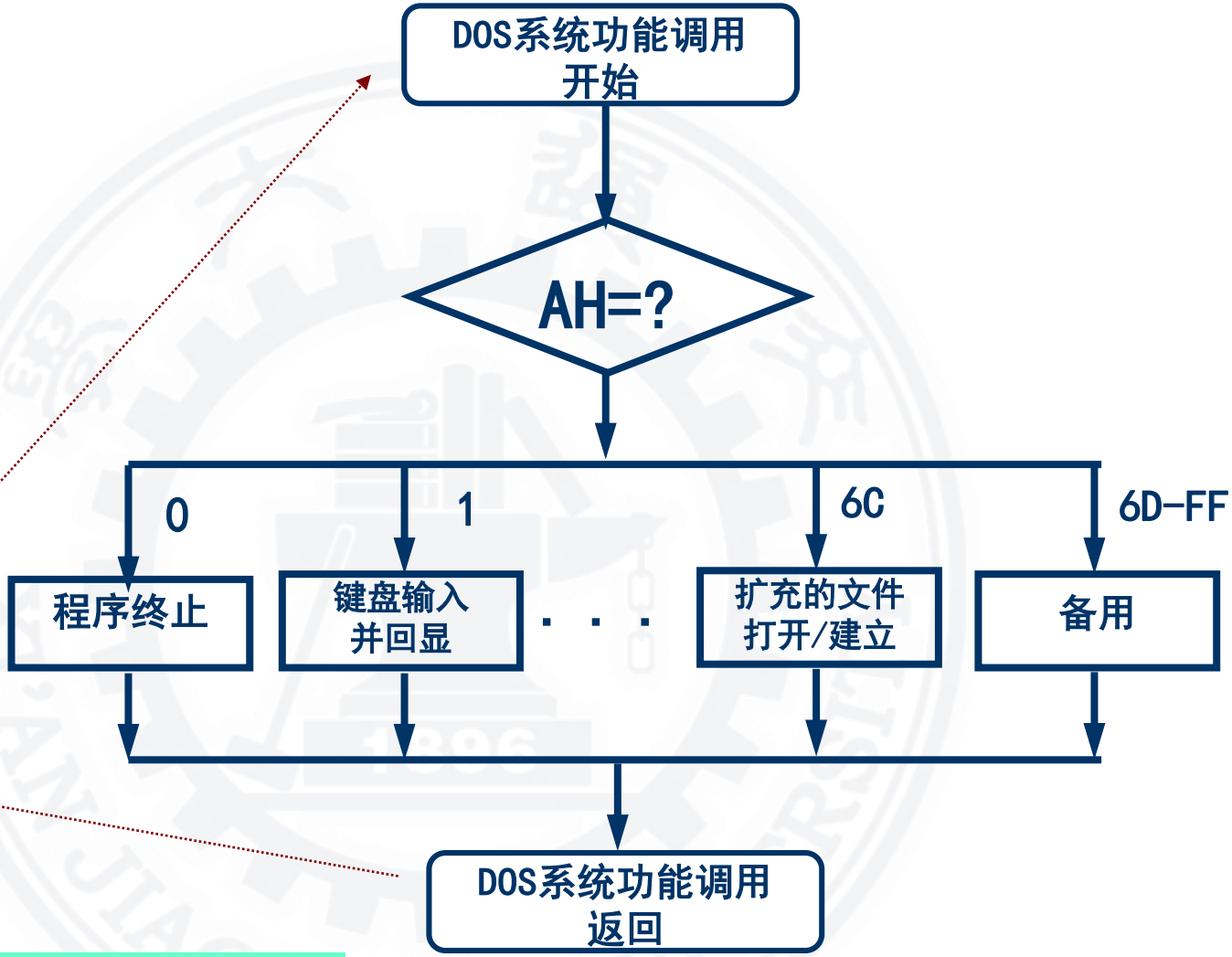
用户程序中：

设置调用参数指令

MOV AH, 功能号

INT 21H

取返回参数指令




**大家想一想：DOS系统功能调用中如何根据AH内容（功能号）实现多分支？**

## 多分支结构

## （1）DOS键盘功能调用 ( AH=1, 6, 7, 8, A, B, C )

例：单字符输入 ( AH=1 ： 键盘输入并回显)

```
get-key:  mov   ah, 1    ；键盘输入并回显
          int   21h      ；返回时，键盘输入字符在AL中
          cmp   al, 'Y'
          je    yes
          cmp   al, 'N'
          je    no
          jne   get_key
yes:
          ……
no:
          ……
```

如不会编程，请熟记课堂中的这些典型小程序，然后实际中灵活运用

例：输入字符串 ( AH=0ah )

定义缓冲区：

方法1
```
maxlen  db  32
actlen  db  ?
string  db  32  dup  (?)
```

方法2
```
maxlen  db  32, 0, 32 dup (?)
```

方法3
```
maxlen  db  32, 33  dup  (?)
```

输入字符串
```
lea  dx, maxlen
mov  ah, 0ah
int  21h
```

## （2） DOS显示功能调用 ( AH=2, 6, 9 )

例：显示单个字符 （ AH=2 ）

```
mov  ah, 2
mov  dl, 'A'
int  21h
```

例：显示字符串 （ AH=9 ）

```
string  db  'HELLO', 0dh, 0ah, '$'    ; 以$结束
mov  dx, offset  string ;DS:DX=字符串首地址
mov  ah, 9
int  21h
```

## （3） DOS打印功能 ( AH=5 )

例：输出单个字符到打印机 ( AH=5 )

```
mov  ah, 5
mov  dl, 'A'
int  21h
```

# 设计子程序时应注意的问题

1. 子程序功能定义与说明
2. 参数传递方法
3. 寄存器的保存与恢复
4. 密切注意堆栈状态

# 6.5.1 ARM64函数/子程序结构

## 1、函数/子程序结构定义

### (1) 函数/子程序属性

- 全局函数：**.global** func_name
- 未使用.global声明的函数仅可在本文件内被调用

### (2) 函数/子程序声明

- **.type** func_name **%function**

### (3) 函数/子程序定义

```
func_name:
        ... // 指令
        ... // 指令
```

### (4) 函数/子程序长度

- .size func_name ( . – func_name )

# 6.5.1 ARM64函数/子程序结构

## 2、函数/子程序调用和返回

### (1) 函数/子程序调用

- 直接调用：**bl func_name**　　//函数名字func_name
- 间接调用：**blr Xn**　　//Xn中保存了func_name的地址
- 函数调用，会将返回地址存入**X30**

### (2) 函数/子程序返回

- 缺省返回方式：**ret**　　//把**X30**的内容赋值给PC
- 显式返回方式：**ret Xn**　　//把**Xn**的内容赋值给PC
  - 适用于该函数嵌套调用其它函数时，将**X30**值保存到了**Xn**寄存器

# 6.5.1 ARM64函数/子程序结构

## 3、函数/子程序嵌套调用和返回

与80X86不同点，将返回地址保存在X30

### (1) 函数/子程序嵌套调用

- 函数f调用函数g，函数g的返回地址保存在X30
- 函数g再调用函数h，函数h的返回地址也会保存在X30中
- 如果函数g中不保存X30值，函数g将无法正确返回

### (2) 当有函数嵌套调用时，需要保存X30的值

- 若函数g需要调用其它函数，则需要在进入函数后保存X30的值。
  - 保存到堆栈中，STP X29, X30, [SP, #-16]!
  - 保存在其它寄存器中，例如 MOV X20， X30

# 6.5.1 ARM64函数/子程序结构

## 4、参数传送

(1) 自定义函数间的参数和返回值可以按照自己的习惯定义和实现

- 用寄存器传递参数或返回值
- 用内存单元传递
  - 内存单元传递特例：使用堆栈传递

(2) 若函数会被C函数调用，须遵循特定规则

- 参数数量少于8时：按顺序使用X0到X7
- 参数数量大于8时：还需要把其余的参数按照逆序保存到堆栈中。
  - 先存第n个参数，最后再存第9个参数，最前面的8个参数用X0到X7传送
- 被调用函数按照规则，会保存X19-X28，其它寄存器值在调用函数后可能会改变，需要调用函数自行保存
- 使用X0传递返回值

# 6.5.2 函数全局属性

- 源文件addsub.S
  - 仅能在本模块内被调用的函数
    - myadd
    - mysub
  - 可以被全局（其它模块）调用的函数
    - testfunc
    - 由global声明

```
.global testfunc
// x0 = x1 + x2
myadd:    // 本文件内被调用
    add x0, x1, x2
    ret

// x0 = x1 - x2
mysub:    // 本文件内被调用
    sub x0, x1, x2
    ret

// 可被其它文件内的函数调用
testfunc:
    stp x29, x30, [sp, #-16]!
    bl myadd
    bl mysub
    ldp x29, x30, [sp], #16
    ret
```

# 6.5.2 函数全局属性

- 源文件testfunc.S
  - 定义了main函数
  - 在main函数中可以调用在addsub.S中定义的全局函数testfunc
  - 但若调用在addsub.S中定义的myadd函数，则会报下面的错误
    - 函数myadd未定义

```
.data
    X: .dword  0x2222
    Y: .dword  0x1111
    Z: .dword  0x0

.text
.global main
main:
  stp x29, x30, [sp, #-16]!
  ldr x1, X //x1 ← X变量的值
  ldr x2, Y //x2 ← Y变量的值
  bl testfunc  //调用全局函数
  //bl  myadd  // 调用失败
  ldp x29, x30, [sp], #16
  ret
```

```
gcc -o ttt addsub.o testfunc.o
testfunc.o:testfunc.S:13: undefined reference to `myadd'
collect2: error: ld returned 1 exit status
```

# 6.5.3 函数参数传递

- 使用寄存器传送参数
- 源文件reg.S
  - 在main函数中调用myadd函数
  - myadd有两个输入参数，分别是x1和x2
  - myadd用x0返回计算结果
  - 即myadd的参数和返回值均使用寄存器进行传送

```
.data
    X: .dword  0x2222
    Y: .dword  0x1111
    Z: .dword  0x0

.text
.global main
main:
  stp x29, x30, [sp, #-16]!
  ldr x1, X     //x1 ← X变量的值
  ldr x2, Y     //x2 ← Y变量的值
  bl  myadd
  adr x3, Z      //得到变量Z的地址
  str x0, [x3]    //Z ← 计算结果
  ldp x29, x30, [sp], #16
  ret

// x0 = x1 + x2
// X1和X2传送参数，X0传送返回值
myadd:
    add x0, x1, x2
    ret
```

# 6.5.3 函数参数传递

- 使用存储器直接访问传送参数
- 源文件**mem.S**
  - proadd函数计算数组中所有元素的累加值
  - proadd函数直接访问数组arr，数组长度count，并把结果存入sum中
  - 如果有第2个数组，第3个数组呢？

```
.data
  arr: .dword 1, 2, 3, 4, 5, 6
  count: .dword (.-arr)/8   //数组长度
  sum:   .dword 0
.text
.global main
main: stp x29, x30, [sp, #-16]!
      bl proadd
      ldp x29, x30, [sp], #16
      ret

// 函数直接访问数据段定义的变量
proadd: adr x0, arr // 数组首地址
        ldr x1, count // 数组长度
        eor x2, x2, x2
next: ldr x3, [x0], #8 // 取数组元素
      add x2, x2, x3 // 累加数组元素
      subs x1, x1, 1 // 剩余元素个数
      bne next   // 未累加完，则继续
      adr x0, sum  //得到sum变量地址
      str x2, [x0] //sum ← 累加结果
      ret
```

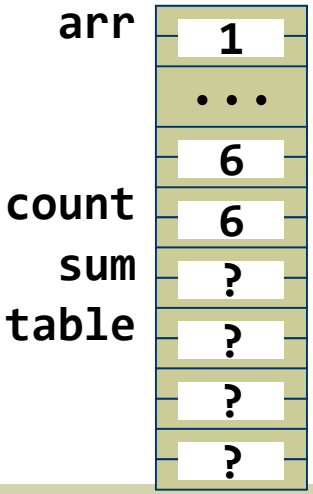
# 6.5.3 函数参数传递

- 使用地址表传送变量地址
- 源文件table.S
  - 适用于参数较多的情况
  - 具体方法：

    ①先建立一个地址表，该表由参数地址构成

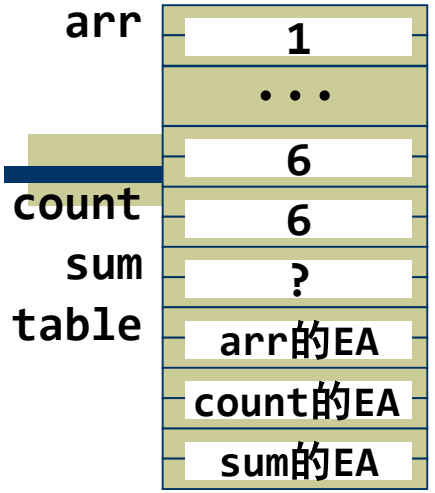
    ②然后把表的首地址通过寄存器或堆栈传递给子程序

```
.data
    arr:   .dword 1, 2, 3, 4, 5, 6
    count: .dword (.-arr)/8  // 数组长度
    sum:   .dword 0
// 地址表，依次保存变量arr、count和sum
// 的地址，在程序中分别对其进行赋值
    table: .dword 0, 0, 0
```



# 6.5.3 函数参数传递

- table的地址通过X0寄存器传送给proadd函数




```
.text
.global main
main: stp x29, x30, [sp, #-16]!
      adr x0, table
//arr的地址存入地址表的第1个元素
    adr x1, arr // arr的地址
    str x1, [x0]
//count的地址存入地址表第2个元素
    adr x1, count
    str x1, [x0, #8]
//sum的地址存入地址表的第3个元素
    adr x1, sum
    str x1, [x0, #16]
// X0保存了地址表的首地址
    bl proadd
    ldp x29, x30, [sp], #16
    ret
```

```
proadd:     //从地址表得到
  ldr x4, [x0] //数组地址
// 从地址表中得到数组长度
  ldr x5, [x0, #8]
  ldr x1, [x5] ; x1=数组长度
// 从地址表得到sum的地址
  ldr x5, [x0, #16]
  eor x2, x2, x2
next: ldr x3, [x4], #8
      add x2, x2, x3
      subs x1, x1, 1
      bne next
      str x2, [x5]
      ret
```

# 6.5.3 函数参数传递

- 使用堆栈传送变量或变量地址
- 源文件**stack.S**
  - 适用于参数较少，或子程序嵌套、递归调用的情况
  - 步骤：

    (1) 主程序把参数或参数地址压入堆栈

    (2) 子程序使用堆栈中的参数或通过栈中参数地址取到参数

    (3) 主程序在子程序返回后，需要调整堆栈SP指针

    **add sp, sp, #n　(n为16的倍数)**

  - 举例：通过堆栈传送变量地址（64位）

# 6.5.3 函数参数传递

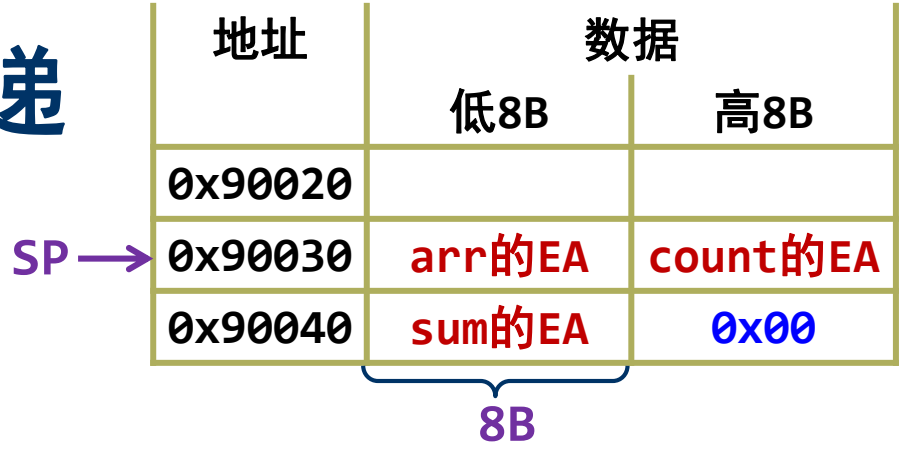
- 使用堆栈传送变量或变量地址
- 使用建议
  - 若被调用函数是叶子函数，即它不会再调用其它函数，同时被调用函数不使用X30，则它的返回地址（X30）就不需要保存
  - 堆栈使用时，一定要16字节对齐。建议一次压入2个64位的寄存器
    - STP X29, X30, [SP, #-16]!

      //此时，X29在低8字节，X30在高8字节
  - 若要传递多个参数，建议使用逆序压入堆栈；若参数个数是奇数，则建议最后一个参数和XZR一起压入堆栈
    - 假设有3个参数

      STP 参数3, XZR, [SP, #-16]!  //XZR是0寄存器

      STP 参数1, 参数2, [SP, #-16]!

# 6.5.3 函数参数传递




```
.data
  arr: .dword 1, 2, 3, 4, 5, 6
  count: .dword (.-arr)/8
  sum:   .dword 0
.text
.global main
main: stp x29, x30, [sp, #-16]!
    adr x0, arr       //arr的地址
    adr x1, count     //count地址
    adr x2, sum      //sum的地址
    stp x2, xzr, [sp, #-16]!
                  //先将sum地址压入堆栈
    stp x0, x1,  [sp, #-16]!
            //将arr和count地址压入堆栈
    bl proadd
    add sp, sp, #32    // 恢复堆栈
    ldp x29, x30, [sp], #16
    ret
```

```
proadd:
   ldr x4, [sp]          //取arr地址
   ldr x5, [sp, #8]  //取count地址
   ldr x1, [x5]          //取数组长度
   ldr x5, [sp, #16]
                        //取得sum的地址
   eor x2, x2, x2     //x2清零

next: ldr x3, [x4], #8
      add x2, x2, x3
      subs x1, x1, 1
      bne next
      str x2, [x5]
      ret
```

# 谢谢！