第10章
CALL和RET指令

第10章 call 和 ret 指令

- 10.1 ret 和 retf
- 10.2 call 指令
- 10.3 依据位移进行转移的call指令
- 10.4 转移的目的地址在指令中的call指令
- 10.5 转移地址在寄存器中的call指令
- 10.6 转移地址在内存中的call指令
- 10.7 call 和 ret 的配合使用
- 10.8 mul 指令
- 10.9 模块化程序设计
- 10.10 参数和结果传递的问题
- 10.11 批量数据的传递
- 10.12 寄存器冲突的问题

引言

- **转移指令**修改IP，或同时修改CS和IP。
  - jmp
  - jcxz
  - loop
  - call
  - ret和retf

10.1 ret 和 retf

- ret指令用栈中的数据，**只修改IP**的内容，实现近转移；
  - 操作

（1）(IP)=((ss)*16+(sp))

（2）(sp)=(sp)+2

  - CPU执行ret指令时，相当于进行：

pop IP

10.1 ret 和 retf

- retf指令用栈中的数据，**修改CS和IP**的内容，实现远转移；
  - 操作

    （1）(IP)=((ss)*16+(sp))

    （2）(sp)=(sp)+2

    （3）(CS)=((ss)*16+(sp))

    （4）(sp)=(sp)+2
  - CPU执行retf指令时，相当于进行：

    pop IP

    pop CS

10.1 ret 和 retf

- 示例程序
  - ret指令

  执行ret后，返回到哪？

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
        mov ax,4c00h
        int 21h

start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,0
        push ax
        mov bx,0
        ret
code ends

end start
```

10.1 ret 和 retf

- 示例程序
  - ret指令

执行ret后，返回到哪？

(IP)=0，

CS:IP指向代码段的第一条指令。

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
        mov ax,4c00h
        int 21h

start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,0
        push ax
        mov bx,0
        ret
code ends

end start
```

特别提示

- 检测点10.1（p191）

补全程序，实现从内存1000:0000处继续执行指令。

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,_________
        push ax
        mov ax,_________
        push ax
        retf
code ends

end start
```

没有完成此检测点，请不要向下进行。

10.2 call 指令

- CPU执行call指令，进行两步操作：

  （1）**压栈**：将IP或（CS和IP） 入栈；

  （2）**转移**：相对或绝对位置转移。

- call 指令不能实现短转移（8位长的位移），除此之外，call指令实现转移的方法和 jmp 指令的原理相同。

- 以下按给出转移目的地址的不同讲解call指令的格式。

10.3 依据位移进行转移的call指令

- 指令格式

  call 标号

- 功能：相对转移——先将IP压栈，再通过位移转到标号处执行指令

- 说明1：CPU相当于进行如下的操作：

  (1) (sp) = (sp) – 2

  ((ss)*16+(sp)) = (IP)

  (2) (IP) = (IP) + 16位位移

10.3 依据位移进行转移的call指令

- 指令格式

  call 标号

- 功能：相对转移——先将IP压栈，再通过位移转到标号处执行指令

- 说明2：
  - 16位位移=“标号”处的地址－call指令后的第一个字节的地址；
  - 16位位移的范围为 -32768~32767，用补码表示；
  - 16位位移由编译程序在编译时算出。

10.3 依据位移进行转移的call指令

- 指令格式

  call 标号

- 功能：相对转移——先将IP压栈，再通过位移转到标号处执行指令

- 说明3：
  - 用汇编语法来解释，相当于进行：

    push IP

    jmp near ptr 标号

10.3 依据位移进行转移的call指令

- 指令格式

  call 标号

- 功能：相对转移——先将IP压栈，再通过位移转到标号处执行指令

- 说明4：
  - call指令对应的机器指令中并没有转移的目的地址，而是相对于当前IP的转移位移。

特别提示

- 检测点10.2（p181）

下面程序执行后，ax中的值为多少？

```
内存地址        机器码          汇编指令
1000: 0         b8 00 00        mov ax,0
1000: 3         e8 01 00        call s
1000: 6         40              inc ax
1000: 7         58            s:pop ax
```

没有完成此检测点，请不要向下进行。

10.4 转移的目的地址在指令中的call指令

- 指令格式　　call far ptr 标号

绝对转移——先压栈CS:IP，再转到标号处执行指令。

- 说明1:
  - (1) (sp) = (sp) – 2

    ((ss) ×16+(sp)) = (CS)

    (sp) = (sp) – 2

    ((ss) ×16+(sp)) = (IP)
  - (2) (CS) = 标号所在的段地址

    (IP) = 标号所在的偏移地址

10.4 转移的目的地址在指令中的call指令

- 指令格式　　call far ptr 标号

绝对转移——先压栈CS:IP，再转到标号处执行指令。

- 说明2:
  - 用汇编语法来解释此种格式的 call 指令，相当于:

```
push CS
push IP
jmp far ptr  标号
```

特别提示

- 检测点10.3（p181）

- 没有完成此检测点，请不要向下进行。

10.5 转移地址在寄存器中的call指令

- 指令格式：　　call 16位寄存器
  - 功能：

（1）(sp) = (sp) – 2

((ss)*16+(sp)) = (IP)

（2）(IP) = (16位寄存器)

10.5 转移地址在寄存器中的call指令

- 指令格式：　　call 16位寄存器
  - 汇编语言解释：

    push IP

    jmp 16位寄存器

特别提示

- 检测点10.4（p182）

- 没有完成此检测点，请不要向下进行。

10.6 转移地址在内存中的call指令

- 转移地址在内存中的call指令有两种格式：
  - (1) call word ptr 内存单元地址
  - (2) call dword ptr 内存单元地址

10.6 转移地址在内存中的call指令

- (1) call word ptr 内存单元地址
  - 汇编语法解释：

    push IP

    jmp word ptr 内存单元地址

10.6 转移地址在内存中的call指令

- (1) call word ptr 内存单元地址（示例）
  - 比如下面的指令：

```
mov sp,10h
mov ax,0123h
mov ds:[0],ax
call word ptr ds:[0]
```

执行后，

(IP)= 0123H

(sp)= 0EH

10.6 转移地址在内存中的call指令

- (2) call dword ptr 内存单元地址
  - 汇编语法解释：

    push CS

    push IP

    jmp dword ptr 内存单元地址

10.6 转移地址在内存中的call指令

- (2) call dword ptr 内存单元地址（示例）
  - 比如，下面的指令：

```
mov sp,10h
mov ax,0123h
mov ds:[0],ax
mov word ptr ds:[2],0
call dword ptr ds:[0]
```

执行后，(CS) = **0**，(IP)= **0123H**，(sp)= **0CH**

call小结

- 通过**标号**转移
  - call 标号 ; 相对转移：IP入栈，按16位位移转移到标号
  - call far ptr 标号 ; 绝对长转移：cs ip 依次入栈，再转移
- 通过**寄存器**转移
  - call 寄存器 ; 绝对近转移：ip入栈，ip新值为寄存器数据
- 通过**内存**转移
  - call 内存地址 ;绝对近转移：ip入栈， ip新值为地址上的内存数据
  - call far ptr 内存地址 ;绝对长转移：cs ip 依次入栈，
    ;再转移地址上的内存表示的cs ip

特别提示

- 检测点10.5（p183）

- 没有完成此检测点，请不要向下进行。

10.7 call 和 ret 的配合使用

- 下面学习 ret 和call指令配合使用来实现子程序的机制。

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
   s:   add ax,ax
        loop s
        ret
code ends
end start
```

- 问题10.1

  右面程序返回前，bx中的值是多少？

思考后看分析。

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:   mov ax,1
         mov cx,3
         call s
         mov bx,ax        ;(bx) = ?
         mov ax,4c00h
         int 21h
    s:   add ax,ax
         loop s
         ret
code ends
end start
```

- 问题10.1（分析）
  - （1）CPU取指令call s指令后，IP指向了call mov bx,ax。然后CPU执行call s指令，将当前的IP值压栈，并将 IP 的值改变为标号 s处的偏移地址；

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
    s:  add ax,ax
        loop s
        ret
code ends
end start
```

- 问题10.1（分析）

- （2）CPU从标号 s 处开始执行指令，loop循环完毕，(ax)=8；

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
   s:   add ax,ax
        loop s
        ret
code ends
end start
```

- 问题10.1（分析）

- （3）CPU将ret指令的机器码读入，IP指向了ret指令后的内存单元，然后CPU 执行 ret 指令，从栈中弹出一个值（即call 先前压入的mov bx,ax 指令的偏移地址）送入 IP 中。则CS:IP指向指令mov bx,ax；

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
   s:   add ax,ax
        loop s
        ret
code ends
end start
```

- 问题10.1（分析）

- CPU从 mov bx,ax 开始执行指令，直至完成。

程序返回前，(bx)=8 。

10.7 call 和 ret 的配合使用

```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
   s:   add ax,ax
        loop s
        ret
code ends
end start
```

- 问题10.1 （分析）

- 可看出，这段程序的作用是计算2的N次方，计算前，N的值由CX提供。

10.7 call 和 ret 的配合使用

源程序　　内存中的情况（假设程序从内存1000:0处装入）

```
assume cs:code
stack segment
        db  8 dup (0)     1000:0000  00 00 00 00 00 00 00 00
        db  8 dup (0)     1000:0008  00 00 00 00 00 00 00 00
stack ends

code segment
start:  mov ax,stack      1001:0000  B8 05 14
        mov ss,ax         1001:0003  8E D0
        mov sp,16         1001:0005  BC 10 00
        mov ax,1000       1001:0008  B8 E8 03
        call s            1001:000B  E8 05 00

        mov ax,4c00h      1001:000E  B8 00 4C
        int 21h           1001:0011  CD 21

    s:  add ax,ax         1001:0013  03 C0
        ret               1001:0015  C3
code ends

end start
```

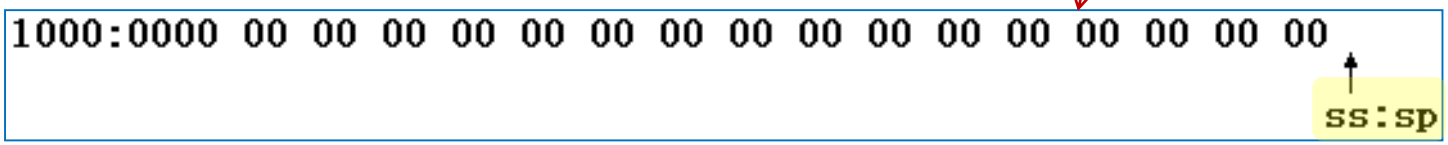
- 分析这个示例程序

10.7 call 和 ret 的配合使用

源程序　　内存中的情况（假设程序从内存1000:0处装入）

```
assume cs:code
stack segment
        db  8 dup (0)     1000:0000  00 00 00 00 00 00 00 00
        db  8 dup (0)     1000:0008  00 00 00 00 00 00 00 00
stack ends

code segment
start:  mov ax,stack      1001:0000  B8 05 14
        mov ss,ax         1001:0003  8E D0
        mov sp,16         1001:0005  BC 10 00
        mov ax,1000       1001:0008  B8 E8 03
        call s            1001:000B  E8 05 00

        mov ax,4c00h      1001:000E  B8 00 4C
        int 21h           1001:0011  CD 21

    s:  add ax,ax         1001:0013  03 C0
        ret               1001:0015  C3
code ends

end start
```

（1）初始化栈ss和sp，结果如下：



10.7 call 和 ret 的配合使用

源程序

```
assume cs:code
stack segment
        db  8 dup (0)
        db  8 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,1000
        call s

        mov ax,4c00h
        int 21h

    s:  add ax,ax
        ret
code ends

end start
```

内存中的情况（假设程序从内存1000:0处装入）

```
1000:0000  00 00 00 00 00 00 00 00
1000:0008  00 00 00 00 00 00 00 00

1001:0000  B8 05 14
1001:0003  8E D0
1001:0005  BC 10 00
1001:0008  B8 E8 03
1001:000B  E8 05 00

1001:000E  B8 00 4C
1001:0011  CD 21

1001:0013  03 C0
1001:0015  C3
```

（2）CPU取指令call后，修改 IP=000EH，然后执行call，当前IP入栈（如下图） 并修改IP，(IP)=(IP)+0005=0013H。

```
1000:0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 00
                                                  ↑
                                                 ss:sp
```

10.7 call 和 ret 的配合使用

源程序　　内存中的情况（假设程序从内存1000:0处装入）

```
assume cs:code
stack segment
        db  8 dup (0)     1000:0000  00 00 00 00 00 00 00 00
        db  8 dup (0)     1000:0008  00 00 00 00 00 00 00 00
stack ends

code segment
start:  mov ax,stack      1001:0000  B8 05 14
        mov ss,ax         1001:0003  8E D0
        mov sp,16         1001:0005  BC 10 00
        mov ax,1000       1001:0008  B8 E8 03
        call s            1001:000B  E8 05 00

        mov ax,4c00h      1001:000E  B8 00 4C
        int 21h           1001:0011  CD 21

    s:  add ax,ax         1001:0013  03 C0
        ret               1001:0015  C3
code ends

end start
```

（3）CPU从处即标号s处开始执行。

10.7 call 和 ret 的配合使用

源程序

```
assume cs:code
stack segment
        db  8 dup (0)
        db  8 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,1000
        call s

        mov ax,4c00h
        int 21h

    s:  add ax,ax
        ret
code ends

end start
```

内存中的情况（假设程序从内存1000:0处装入）

```
1000:0000  00 00 00 00 00 00 00 00
1000:0008  00 00 00 00 00 00 00 00

1001:0000  B8 05 14
1001:0003  8E D0
1001:0005  BC 10 00
1001:0008  B8 E8 03
1001:000B  E8 05 00

1001:000E  B8 00 4C
1001:0011  CD 21

1001:0013  03 C0
1001:0015  C3
```

（4）CPU读入ret指令后并执行，IP值更新为栈顶数据，栈结果如下。

(IP)=000EH;

```
1000:0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 00
                                                  ↑
                                                 ss:sp
```

10.7 call 和 ret 的配合使用

源程序

```
assume cs:code
stack segment
        db  8 dup (0)
        db  8 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,1000
        call s

        mov ax,4c00h
        int 21h

    s:  add ax,ax
        ret
code ends

end start
```

内存中的情况（假设程序从内存1000:0处装入）

```
1000:0000  00 00 00 00 00 00 00 00
1000:0008  00 00 00 00 00 00 00 00

1001:0000  B8 05 14
1001:0003  8E D0
1001:0005  BC 10 00
1001:0008  B8 E8 03
1001:000B  E8 05 00

1001:000E  B8 00 4C
1001:0011  CD 21

1001:0013  03 C0
1001:0015  C3
```

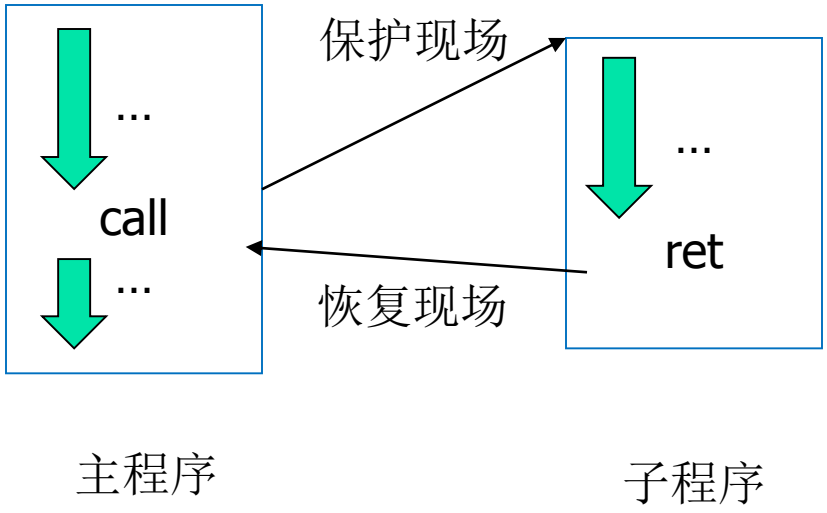
（5）ret执行完毕后，CPU回到cs:000EH处继续执行。

10.7 call 和 ret 的配合使用

- 以上程序中有子程序（具有一定功能的程序段）：

子程序**调用**：用call指令调用子程序，转去执行子程序之前，call指令后面的指令的地址将存储在栈中，

子程序**返回**：在子程序的后面使用 ret 指令返回主调程序，用栈中的数据设置IP的值，从而转到 call 指令后面的代码处继续执行。



10.7 call 和 ret 的配合使用

- 可利用call和ret来实现子程序的机制。

子程序的框架：

标号：指令

ret

具有子程序的源程序的框架：

```
assume cs:code
code segment
main:   :
        :
        call sub1          ;调用子程序sub1
        :
        :
        mov ax,4c00h
        int 21h

sub1:   :                  ;子程序sub1开始
        :
        call sub2          ;调用子程序sub2
        :
        :
        ret                ;子程序返回

sub2:   :                  ;子程序sub2开始
        :
        :
        ret                ;子程序返回
code ends
end main
```

10.7 call 和 ret 的配合使用

- 请从子程序的角度再回过头本节前面的两个程序。

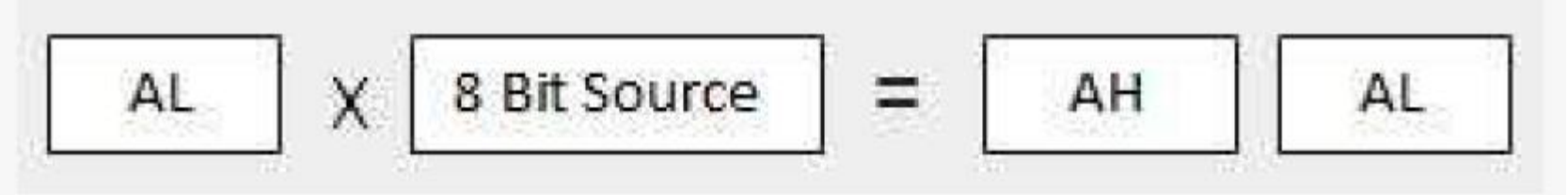
```
assume cs:code
code segment
start:  mov ax,1
        mov cx,3
        call s
        mov bx,ax       ;(bx) = ?
        mov ax,4c00h
        int 21h
   s:   add ax,ax
        loop s
        ret
code ends
end start
```

10.8 mul 指令

- 乘法指令mul：
  - 格式：　　mul reg
    mul 内存单元
    - 分为：
      - 两个8 位数相乘
      - 两个16 位数相乘：

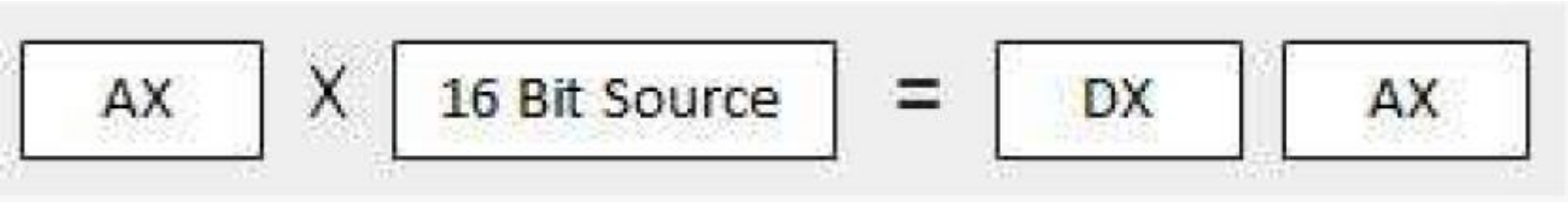
10.8 mul 指令

- 乘法指令mul：
  - 两个8 位数相乘
    - 被乘数在AL中，
    - 乘数在8位寄存器或内存字节单元中；
    - 结果保存在AX中；



10.8 mul 指令

- 乘法指令mul：
  - 两个16 位数相乘：
    - 被乘数在AX，
    - 乘数在16 位寄存器或内存字单元中。
    - 结果保存在DX（高位）和AX（低位）中。



10.8 mul 指令

- 内存单元可以用不同的寻址方式给出，比如：
  - mul byte ptr ds:[0]

    含义为： (ax)=(al)*((ds)*16+0);

  - mul word ptr [bx+si+8]

    含义为：

    (ax)=(al)*((ds)*16+(bx)+(si)+8)结果的低16位;

    (dx)=(al)*((ds)*16+(bx)+(si)+8)结果的高16位;

10.8 mul 指令

- 例如：
  - （1）计算100*10

    100和10小于255，可以做8位乘法，程序如下：

```
mov al,100
 mov bl,10
 mul bl
```

结果： (ax)=1000（03E8H）

10.8 mul 指令

- 例如：
  - （2）计算100*10000

    100小于255，可10000大于255，所以必须做16位乘法，程序如下：

```
mov ax,100
mov bx,10000
mul bx
```

结果：(ax)=4240H，(dx)=000FH

（F4240H=1000000）

10.9 模块化程序设计

- 在实际编程中，程序的**模块化**是必不可少的。
  - 因为现实的问题比较复杂，对现实问题进行分析时，把它转化成为相互联系、不同层次的子问题，是必须的解决方法。

- call 与 ret 指令共同支持了汇编语言编程中的模块化设计。
  - 利用 call和ret指令，可用简洁的方法，实现多个互相联系、功能独立的子程序来解决一个复杂的问题。

10.9 模块化程序设计

- 子程序一般都要根据提供的参数处理一定的事务，处理后，将结果（返回值）提供给调用者。

- 参数和返回值传递的问题
  - 如何存储子程序需要的参数和产生的返回值？

10.10 参数和结果传递的问题

- 设计一个子程序，可以根据提供的N，来计算N的3次方。
  - 分析

两个问题：

（1）将参数N存储在什么地方？

（2）计算得到的数值，存储在什么地方？

  - 思路：

(1)可以用寄存器来存储，可以将参数放到 bx 中；

(2) 在子程序中使用多个 mul 指令计算 N×N×N，，为了方便可将结果放到 dx 和 ax中

10.10 参数和结果传递的问题

- 设计一个子程序，可以根据提供的N，来计算N的3次方。(续)

- 子程序：

```
;说明：计算N的3次方
;参数：  (bx)=N
;结果：  (dx:ax)=N^3
cube: mov ax,bx
      mul bx
      mul bx
      ret
```

10.10 参数和结果传递的问题

- 为了便于自己和或他人子程序的使用，应保持良好的编程风格，对于程序要有详细的**注释**：
  - 功能
  - 参数
  - 结果

10.10 参数和结果传递的问题

- 用寄存器来存储参数和结果是最常使用的方法。
  - 用寄存器传递参数的过程：

（1）调用者将参数写入寄存器，然后call子程序；

（2）子程序从参数寄存器中取到参数，将返回值送入结果寄存器。

  - 用寄存器回传结果的过程：

（1）子程序将结果保存到寄存器中，并ret

（2）调用者将参数送入参数寄存器，从结果寄存器中取到返回值；

10.10 参数和结果传递的问题

- 编程：计算data段中第一组数据的3次方，结果保存在后面一组dword单元中。

```
assume cs:code
data segment
   dw 1,2,3,4,5,6,7,8
   dd  0,0,0,0,0,0,0,0
data ends
```

10.10 参数和结果传递的问题

- 编程：计算data段中第一组数据的3次方，结果保存在后面一组dword单元中。

程序代码

```
code segment

 start:mov ax,data
       mov ds,ax
       mov si,0                ;ds:si 指向第一组 word 单元
       mov di,16               ;ds:di 指向第二组 dword 单元

       mov cx,8
    s: mov bx,[si]
       call cube
       mov [di],ax
       mov [di].2,dx
       add si,2                ;ds:si 指向下一个 word 单元
       add di,4                ;ds:di 指向下一个 dword 单元
       loop s

       mov ax,4c00h
       int 21h

cube:  mov ax,bx
       mul bx
       mul bx
       ret

code ends
end start
```

10.11 批量数据的传递

- 如果子程序传递的数据太多，传参或者传递返回值时，寄存器数量不够用怎么办？

  ~想想C语言中的数组传参如何处理的？

- 将参数放在内存中，然后将它们所在内存空间的首地址放在寄存器中，传递给需要的子程序。

10.11 批量数据的传递

- 示例：设计子程序
  - 功能：将一个全是字母的字符串转化为大写。
  - 分析
    - 因为字符串可能很长，不便于将整个串中所有字母都用寄存器直接传递给子程序。
    - 可以将字符串在内存中的首地址放在寄存器中传递给子程序。在子程序中要用到循环loop 指令，而循环的次数为字符串的长度——将字符串的长度放到CX中。

10.11 批量数据的传递

- 示例：设计子程序
  - 功能：将一个全是字母的字符串转化为大写。
  - 子程序

```
capital:and byte ptr [si],11011111b      ;将ds:si所指单元
                                         ;中的字母转化为大写
        inc si                           ;ds:si指向下一个单元

        loop capital

        ret
```

10.11 批量数据的传递

- 编程：将data段中的字符串转化为大写。

```
assume cs:code

data segment
  db 'conversation'
data ends

code segment
  start:mov ax,data
        mov ds,ax
        mov si,0              ;ds:si 指向字符串(批量数据)所在空间的首地址
        mov cx,12             ;cx 存放字符串的长度
        call capital
        mov ax,4c00h
        int 21h

capital:and byte ptr [si],11011111b
        inc si
        loop capital
        ret
code ends
end start
```

10.11 批量数据的传递

- 编程：将data段中的字符串转化为大写。

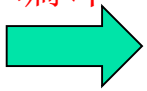
  - 注意：除了寄存器传递参数外，还有一种通用的方法使用栈来传递参数——参看附注4。

C中传参的汇编实现

```
void add(int,int,int);

main()
{
 int a=1;
 int b=2;
 int c=0;
 add(a,b,c);
 c++;
}

void add(int a,int b,int c)
{
 c=a+b;
}
```

C语言

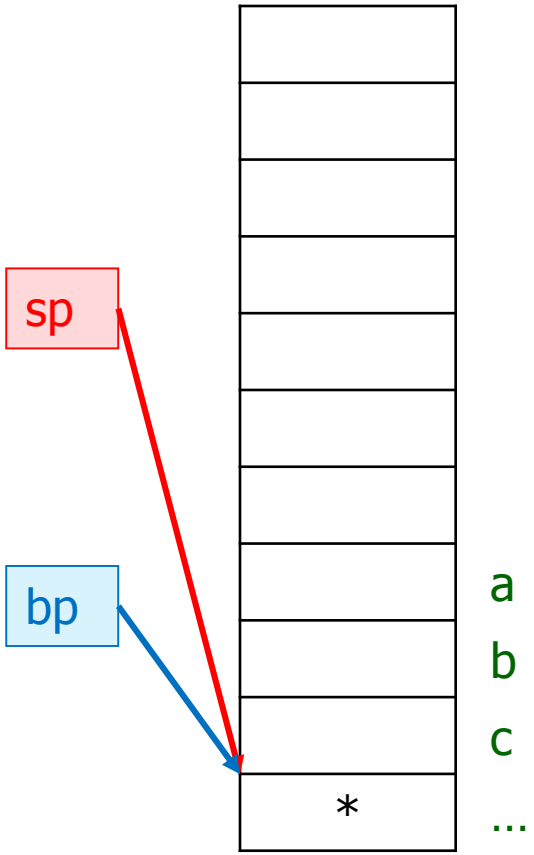
编译



```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

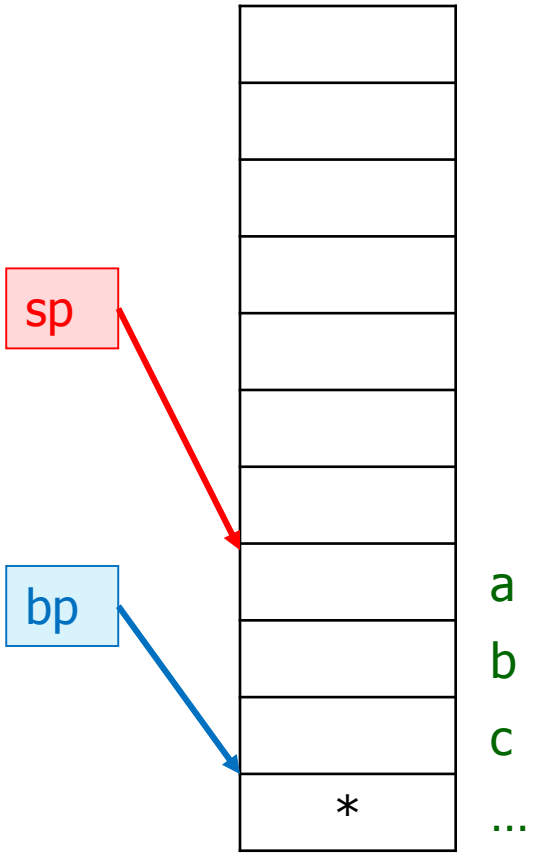
C中传参的汇编实现




```
mov bp,sp          保存进入main时的栈顶
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

C中传参的汇编实现

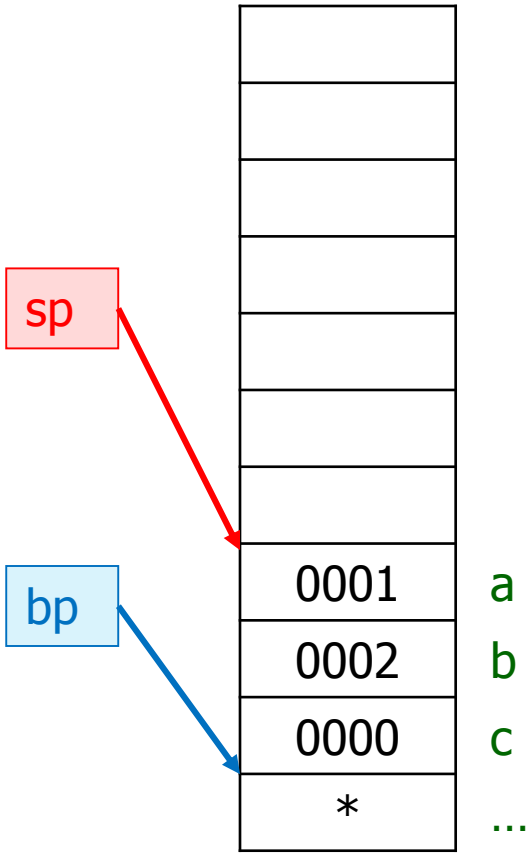



```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

在stack中定义局部变量

汇编语言

C中传参的汇编实现

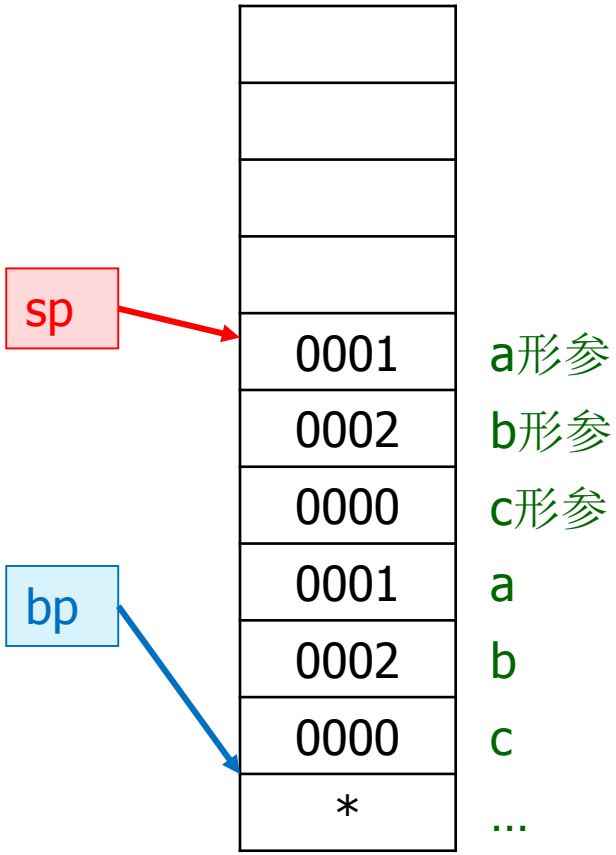


局部变量初始化

```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

C中传参的汇编实现

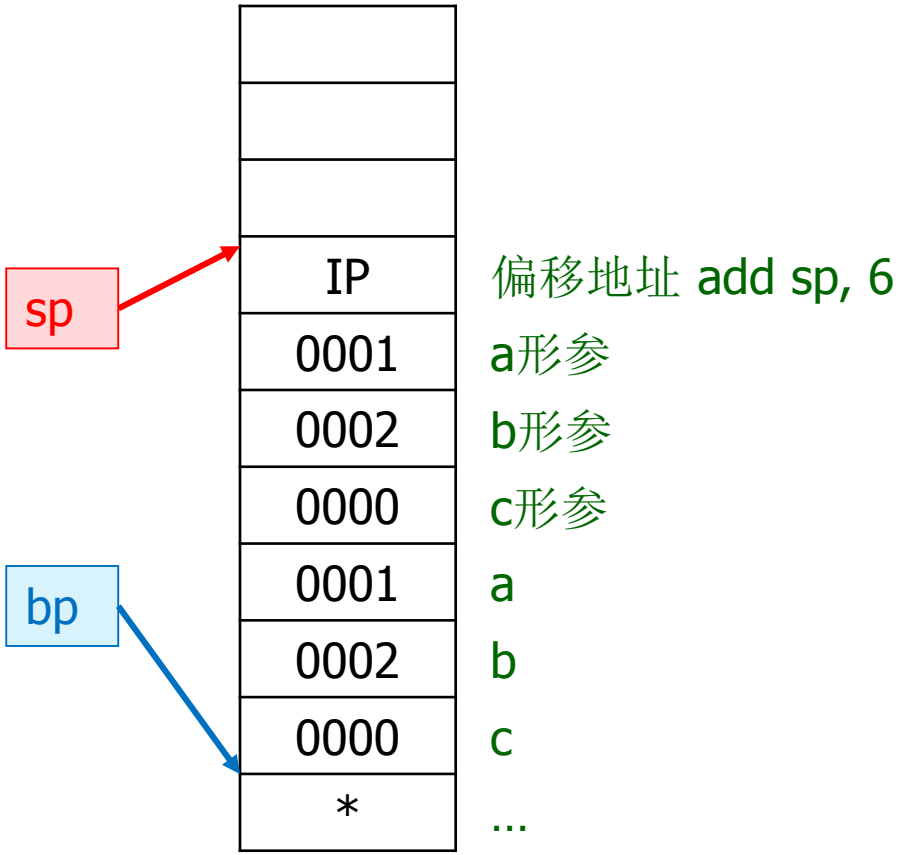



```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

参数入栈——从右到左
（开始函数调用）

汇编语言

C中传参的汇编实现

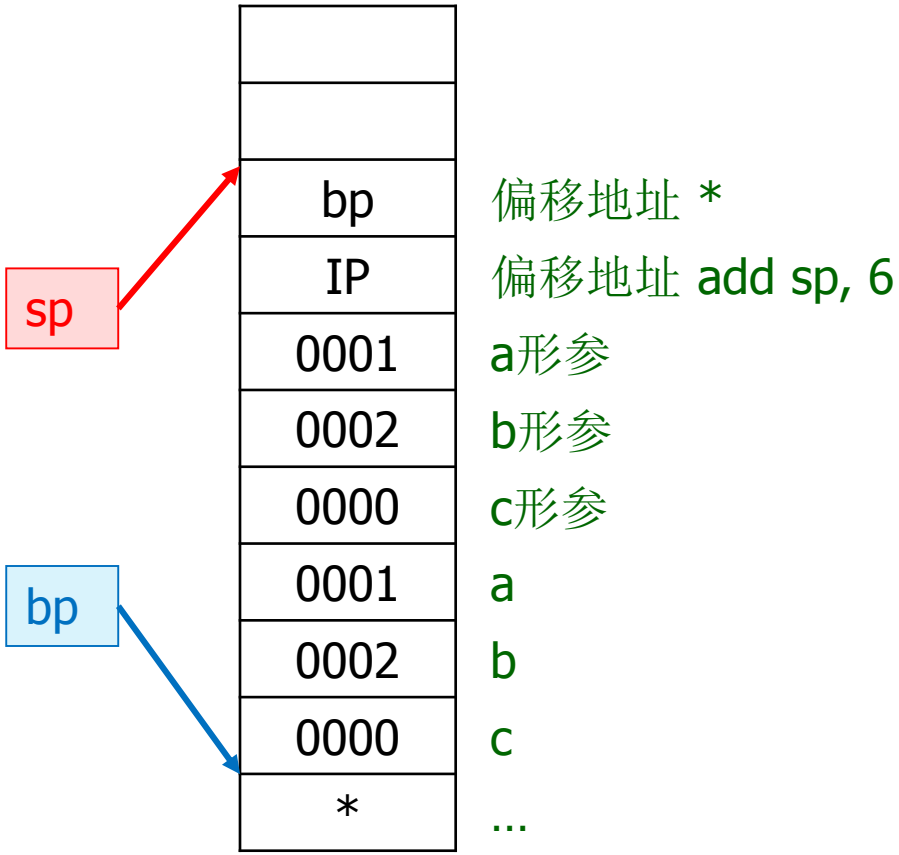



```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

IP入栈，调用子过程

汇编语言

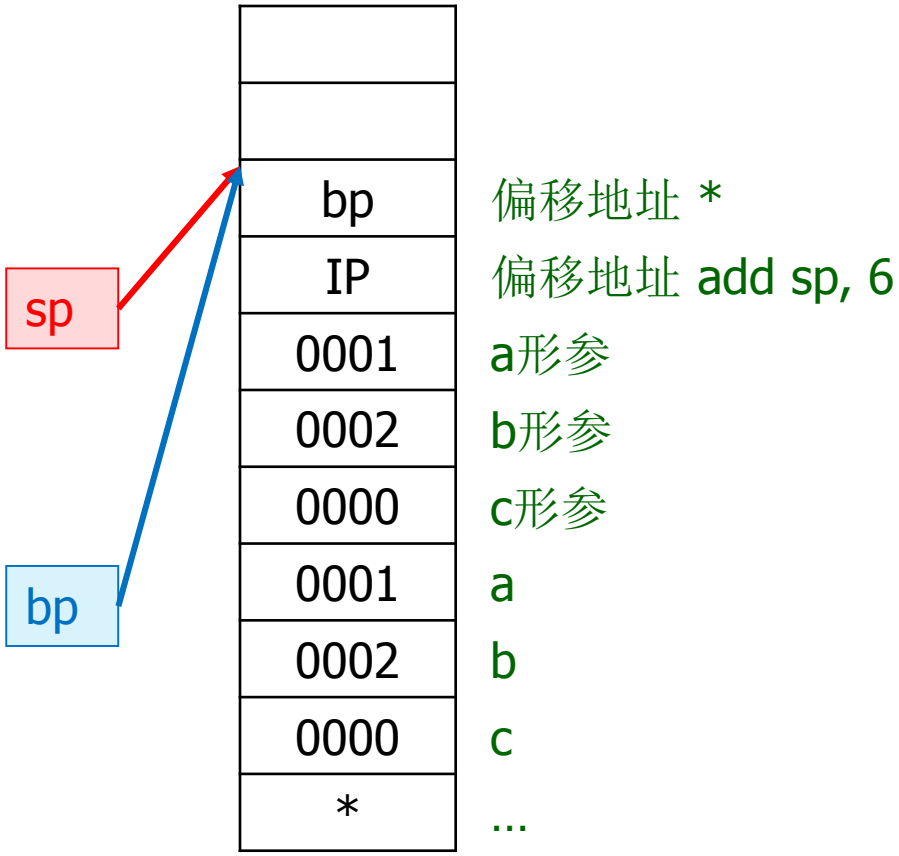
C中传参的汇编实现




```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp    bp入栈
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

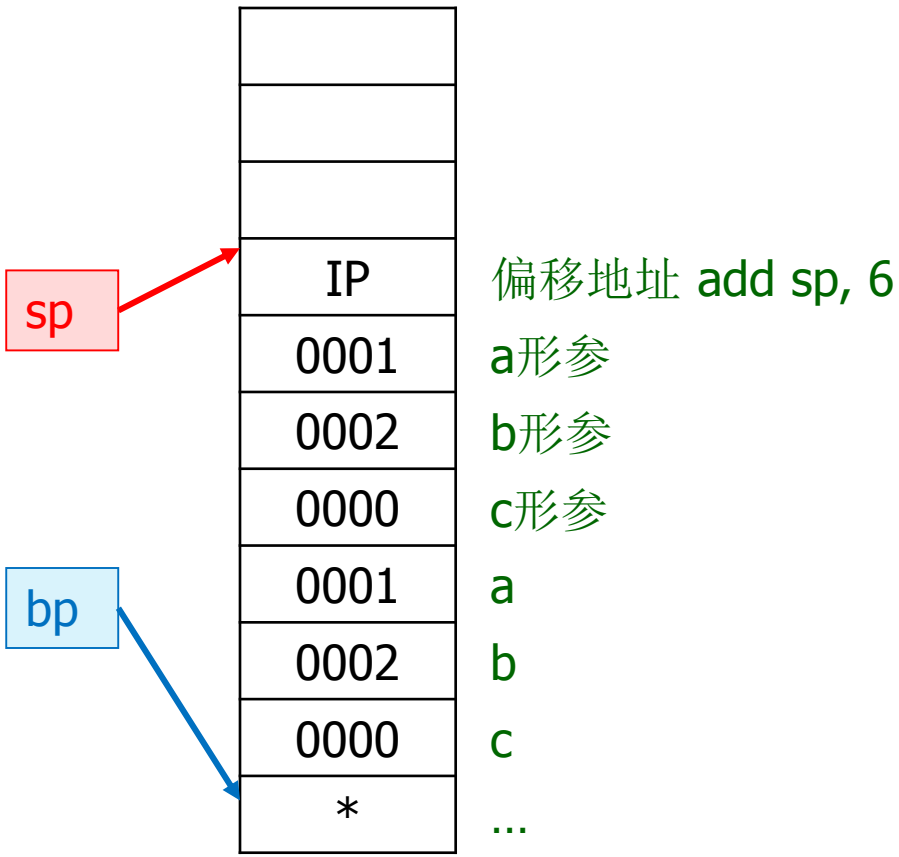
C中传参的汇编实现




```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]   形参a
        add ax,[bp+6]   形参b
        mov [bp+8],ax   形参c
        mov sp,bp
        pop bp
        ret
```

汇编语言

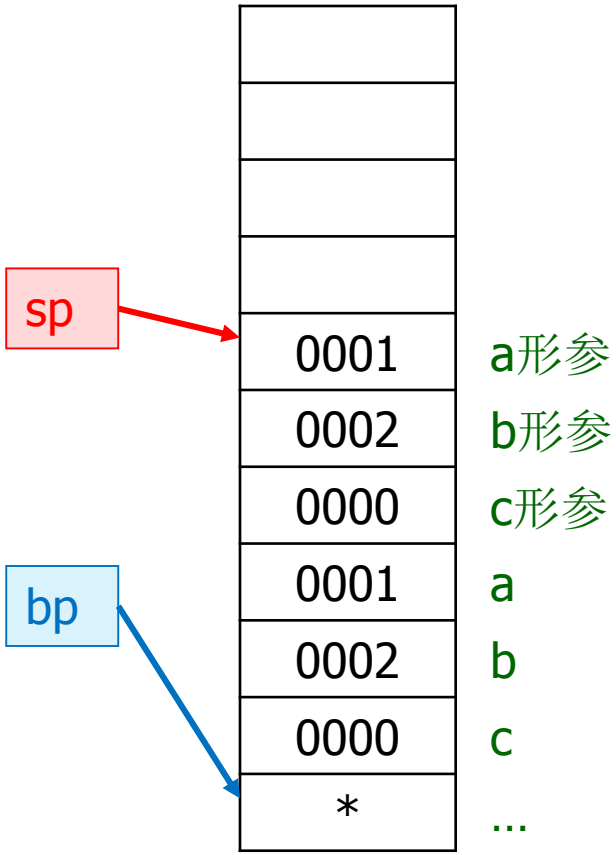
C中传参的汇编实现




```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

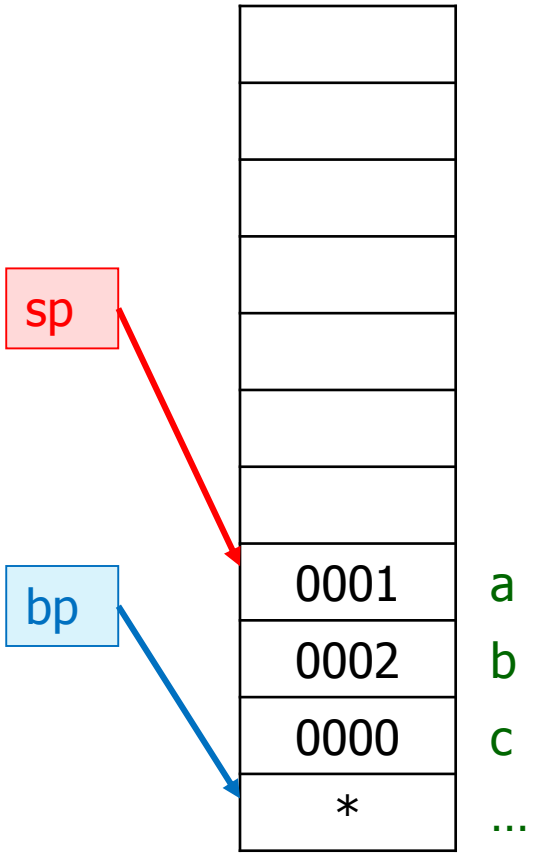
C中传参的汇编实现




```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

汇编语言

C中传参的汇编实现




```
mov bp,sp
sub sp,6
mov word ptr [bp-6],0001    ;int a
mov word ptr [bp-4],0002    ;int b
mov word ptr [bp-2],0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]
call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:   push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret
```

释放形参内存空间

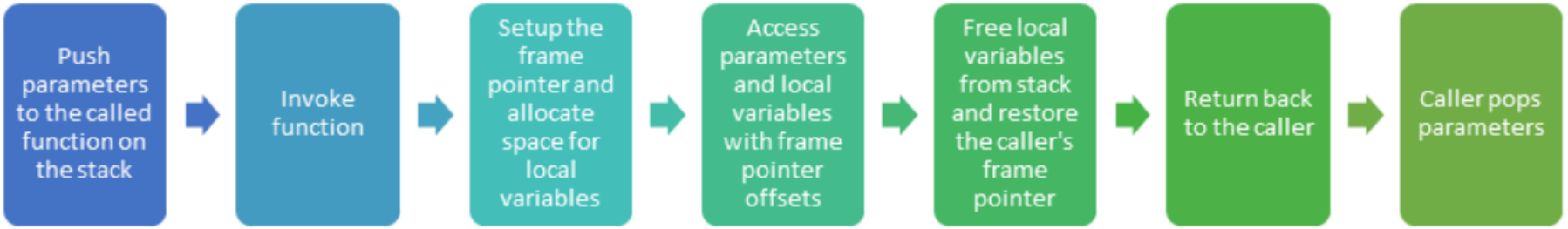
函数调用结束

汇编语言

C中传参的汇编实现

- 小结：

C/C++ 调用函数的执行过程如下：



10.12 寄存器冲突的问题

- 设计一个子程序：
  - 功能：将一个全是字母，以0结尾的字符串，转化为大写。

例如，以下定义了一个以0作为结尾符的字符串：

db ‘conversation’,0

10.12 寄存器冲突的问题

- 设计一个子程序：
  - 功能：将一个全是字母，以0结尾的字符串，转化为大写。

  - 分析

    子程序依次读取并检测每个字符：若不为0，则转化为大写字母，若为0，则结束就结束处理。

    子程序可以不需要字符串的长度作为参数。

    用jcxz来检测0。

10.12 寄存器冲突的问题

- 子程序代码

```
;说明：将一个全是字母，以 0 结尾的字符串，转化为大写
;参数：ds:si 指向字符串的首地址
;结果：没有返回值

capital:mov cl,[si]
        mov ch,0
        jcxz ok                              ;如果(cx)=0，结束；如果不是 0，处理
        and byte ptr [si],11011111b          ;将 ds:si 所指单元中的字母转化为大写
        inc si                               ;ds:si 指向下一个单元
        jmp short capital
     ok:ret
```

10.12 寄存器冲突的问题

- 子程序的应用示例1

（1）将data段中字符串转化为大写

```
assume cs:code
data segment
    db 'conversation',0
data ends
```

代码段中相关程序段如下：

```
mov ax,data
mov ds,ax
mov si,0
call capital
```

10.12 寄存器冲突的问题

- 子程序的应用示例2
  - （2）将data段中字符串全部转化为大写

```
assume cs:code
data segment
    db ‘word',0
    db ‘unix',0
    db ‘wind',0
    db ‘good',0
data ends
```

有4个字符串，每个字符串的长度都是5（算上结尾符0），

使用循环，重复调用子程序capital完成对4个字符串的处理。

10.12 寄存器冲突的问题

- 子程序的应用示例2（续）

完整代码

- 问题10.2

  程序有什么问题

思考后看分析。

```
code segment

 start: mov ax,data
        mov ds,ax
        mov bx,0

        mov cx,4
     s: mov si,bx
        call capital
        add bx,5
        loop s

        mov ax,4c00h
        int 21h

capital:mov cl,[si]
        mov ch,0
        jcxz ok
        and byte ptr [si],11011111b
        inc si
        jmp short capital
     ok:ret

code ends

end start
```

10.12 寄存器冲突的问题

- 子程序的应用示例2（续）

完整代码

- 问题10.2

程序有什么问题

问题在于cx的使用冲突

- 主程序用cx控制循环次数，
- 子程序用cx控制条件跳转

子程序修改cx值，使得主程序的循环出错。

```
code segment

 start: mov ax,data
        mov ds,ax
        mov bx,0

        mov cx,4
     s: mov si,bx
        call capital
        add bx,5
        loop s

        mov ax,4c00h
        int 21h

capital:mov cl,[si]
        mov ch,0
        jcxz ok
        and byte ptr [si],11011111b
        inc si
        jmp short capital
     ok:ret

code ends

end start
```

10.12 寄存器冲突的问题

- 问题10.2分析（续）

  问题：子程序与主程序中公用寄存器时发生冲突。

  如何避免冲突？两种思路：

  - 思路1：若子程序已经编写好，则编写调用程序时调用者使用子程序中未使用的寄存器；

    不可行，主程序编写很麻烦，因为必须小心检查所调用的子程序中是否有将产生冲突的寄存器。

  - 思路2：若调用程序编写好，则编写子程序的时候，不要使用表用程序中使用的寄存器。

    不可行，因为编写子程序的时候无法知道将来的调用情况。

10.12 寄存器冲突的问题

- 问题10.2分析（续）

  我们希望：

  - 编写主程序的时候不必关心子程序到底使用了哪些寄存器；
  - 编写子程序的时候不必关心主程序使用了哪些寄存器；
  - 不会发生寄存器冲突。

10.12 寄存器冲突的问题

- 问题10.2分析（续）
  - 解决方法：

在子程序的开始将子程序中所有用到的寄存器中的内容都**保存**起来，在子程序返回前再恢复。

用栈来保存

10.12 寄存器冲突的问题

- 问题10.2分析（续）

以后，我们编写子程序的标准框架如下：

子程序开始：子程序中使用的寄存器入栈

子程序内容

子程序使用的寄存器出栈

返回（ret、retf）

我们改进一下子程序capital的设计

10.12 寄存器冲突的问题

- 问题10.2改进的子程序capital的设计

```
capital: push cx
         push si
change: mov cl,[si]
         mov ch,0
          jcxz ok
          and byte ptr [si],11011111b
          inc si
          jmp short change
     ok: pop si
         pop cx
         ret
```

- 注意寄存器入栈和出栈的保证后进先出（LIFO）原则。