**-CALL和RET指令**

2016年04月27日 21:58:08 [poetliu](https://me.csdn.net/u013018721) 阅读数：17233 标签： [汇编语言](http://so.csdn.net/so/search/s.do?q=%E6%B1%87%E7%BC%96%E8%AF%AD%E8%A8%80&t=blog) 更多

个人分类： [汇编语言](https://blog.csdn.net/u013018721/article/category/6182121)

所属专栏： [汇编语言学习](https://blog.csdn.net/column/details/masm.html)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/u013018721/article/details/51264199

本博文系列参考自<<汇编语言>>第三版，作者:王爽

call与ret都是转移指令，它们可以改变IP值，或者同时改变CS与IP的值，往往在程序中使用它们进行子程序模块的设计。

10.1  ret和retf

ret用栈的数据修改IP的内容，实现近转移

retf用栈中的数据修改CS与IP的内容，实现远转移

ret执行步骤:

(1):(IP)=((SS)\*16+SP)  
(2):(SP)=(SP)+2

retf执行步骤:

(1):(IP)=((SS)\*16+SP)  
(2):(SP)=(SP)+2

(3):(CS)=((SS)\*16+SP)  
(4):(SP)=(SP)+2

可以将ret理解为:POP IP

可以将retf理解为:POP IP  POP CS

如下例子:

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 cs

push ax

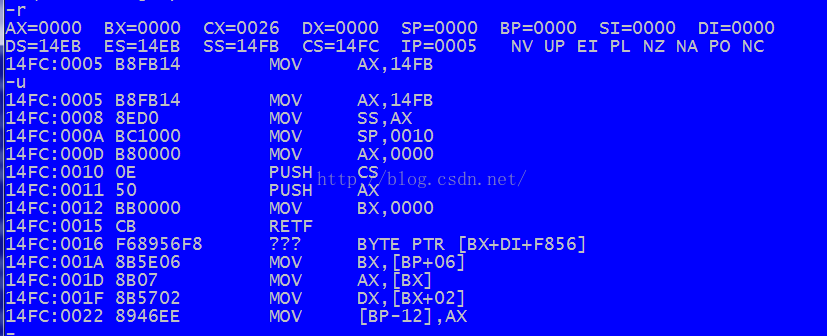
mov bx,0

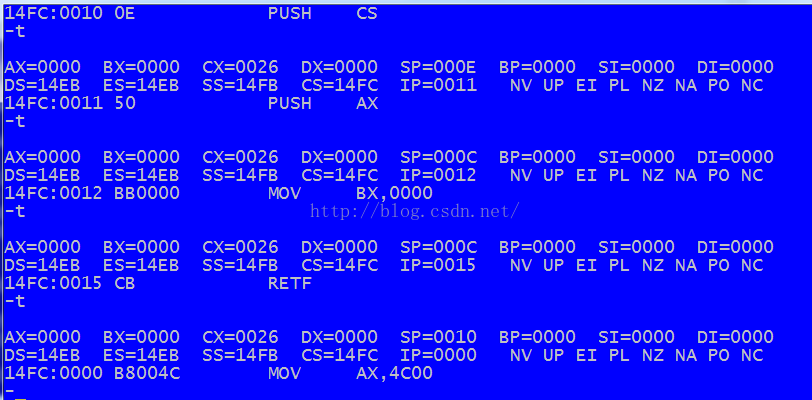
retf

code ends

end start

将其用debug工具单步执行如下:





可以看出，执行完retf后，CS:IP指向第一句指令:mov ax,4c00h

10.2  call指令

CALL指令的执行步骤(call不能实现短转移):

(1)将当前的IP或CS与IP压入栈中

(2)转移

10.3依据位移进行转移的call指令

call 标号(将当前IP压栈，转移到标号处执行)

执行步骤:

(1) (sp)=(sp)-2

    ((ss)\*16+(sp))=(ip)   (实现当前IP值压栈)

(2)(ip)=(ip)+16位位移    (转移到标号处)

16位位移=标号处地址-call指令后的第一个字节的地址

位移范围为-32768-32767由补码给出

位移在编译时计算

call 标号 相当于  call near ptr 标号

10.4 转移的目的地址在指令中的call指令

call 标号指令对应的机器码中没有目的地址,只包括标号处针对当前IP的位移

call far ptr 标号 在机器码中直接给出目的地址，实现段间转移

执行步骤为:

(1) (sp)=(sp)-2

     ((ss)\*16+(sp))=(CS)

     (sp)=(sp)-2

     ((ss)\*16+(sp))=(IP)(IP与CS压栈)

(2) (CS)=标号处所在的[段地址](https://www.baidu.com/s?wd=%E6%AE%B5%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

      (IP)=标号处所在的[偏移地址](https://www.baidu.com/s?wd=%E5%81%8F%E7%A7%BB%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

call far ptr 标号相当于:

push CS

push IP

jmp far ptr 标号

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

执行步骤:

(SP)=(SP)-2

((SS)\*16+(SP))=(IP)

(IP)=(16位寄存器)

call  16位寄存器相当于:

PUSH IP

jmp 16位寄存器

10.6转移地址在内存中的call指令

两种格式：

一 call word ptr [内存地址](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

    相当于: push IP

                jmp word ptr 内存地址

二 call dword ptr 内存地址

    相当于:push  cs

               push  ip

               jmp  dword ptr 内存地址

10.7  call和ret的配合使用

如下代码执行后,bx寄存器中的值为多少:

assume cs:code

code segment

start:mov ax,1

mov cx,3

call s

mov bx,ax

mov ax,4c00h

int 21h

s:add ax,ax

loop s

ret

code ends

end start

debug单步执行结果看出(bx)=8



分析步骤如下:

1.cpu将call s的机器码读入,IP指向了 call后的指令mov bx,ax，然后CPU执行call s指令，将当前IP(指令mov bx,ax的偏移地址)压栈，并将IP改为标号S的偏移地址。

2.cpu从标号s开始执行，循环结束后,(ax)=8

3.cpu将ret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(即先前压入栈的mov bx,ax的偏移地址)弹出一个值存入IP，此时CS:IP指向mov bx,ax

4.cpu 从mov bx,ax开始执行，直至结束

10.8 mul指令

mul作为乘法指令，由如下规则

1.均为8位的乘数，一个放在AL中，另外一个放在8位寄存器或内存中

  均为16位的乘数，一个放在AX中，另外一个放在16位寄存器或内存中

2.结果：如果乘数为8位，结果放在AX中，如果乘数为16位，高位放在DX，低位放在AX

格式:

mul reg

mul 内存单元

比如:

mul byte ptr ds:[0]  含义:(AX)=(AL)\*((DS)\*16+0);

mul word ptr [bx+si+8]  含义:(DX)=(AX)\*((DS)\*16+(bx)+(si)+8)结果的高16位

     (DX)=(AX)\*((DS)\*16+(bx)+(si)+8)结果的低16位

10.9模块化程序设计及参数和结果传递问题

call与ret配合使用可以使得程序达到模块化设计的问题，这在编写功能庞大的程序时候是很用的，模块化程序的逻辑清楚，且可以重复复用模块代码。

子程序需要将程序功能执行完成后将结果返回给调用者。这里我们需要考虑如何存储和传递子程序需要的参数和返回值。

比如一个问题，设计一个子程序，提供一个参数N，计算N的三次方。

这里存在两个问题:

(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。

这里我们可以将参数N放入寄存器BX中，将结果放到DX与AX中，因为是求三次方，我们可以用两次mul指令完成。

代码如下:

cube:mov ax,bx

mul bx

mul bx

ret

用寄存器是常用的传递参数和结果的方法，在调用的时候将参数传入参数寄存器。在调用返回时候，将结果存入结果寄存器。这种方式虽然常用，但是有个缺陷就是传递的参数和结果的个数有限，因为寄存的个数是有限的。

10.10 批量数据的传递

如何解决前面通过寄存器传递参数和结果的时候寄存器数量的限制，导致有很多参数的时候这种寄存器传递参数或者结果的方式不可取。

这种时候，我们需要将要传递的参数放到内存中，然后将内存地址的首地址放在寄存器中，传递给子程序。对于批量返回结果也使用此方法。

有如下例子，需要将一个字符串中字符全部转换为大写字符。

我们可以将字符串存储于内存中的一段连续区域，然后将该内存地址存入寄存器。通过LOOP指令循环遍历内存取出每个字符然后进行大小写转换。结果如下:

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

captical:add byte ptr [si],11011111b

loop captical

ret

code ends

end start

10.11寄存器冲突的问题

寄存器冲突是指在子程序使用的寄存器很有可能在主程序中也有使用，这会造成寄存器在使用上的冲突。

解决这个问题是在执行子程序开始时，将子程序中所有用到的寄存器的值全部保存起来，在子程序返回前，再将其恢复。可以用栈来保存寄存器的内容。

以后编写子程序的框架如下:

子程序开始:子程序使用的寄存器压栈

子程序内容

子程序使用的寄存器出栈

返回(ret,retf)

我们对10.10中的子程序capital修改以防止寄存冲突,修改后如下:

captical: push cx

push si

change: mov cl,[si]

mov ch,0

jcxz ok

and byte ptr [si],11011111b

inc si

imp short change

ok: pop si

pop cx

ret