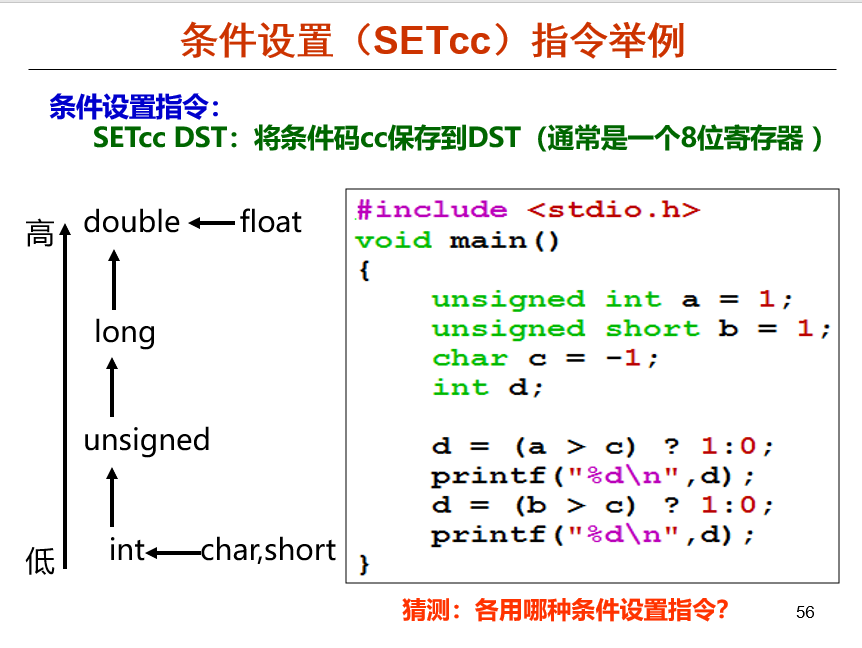
条件设置指令举例



类型转换：横向 – 必须转换 纵向 – 按需要转换

该程序输出为：

0

1

原因：第一句输出 char类型转换为unsigned int类型，c为111…（最大的数）

第二局输出 unsigned char、unsigned short 类型转化为 int类型，1 > -1，d = 1

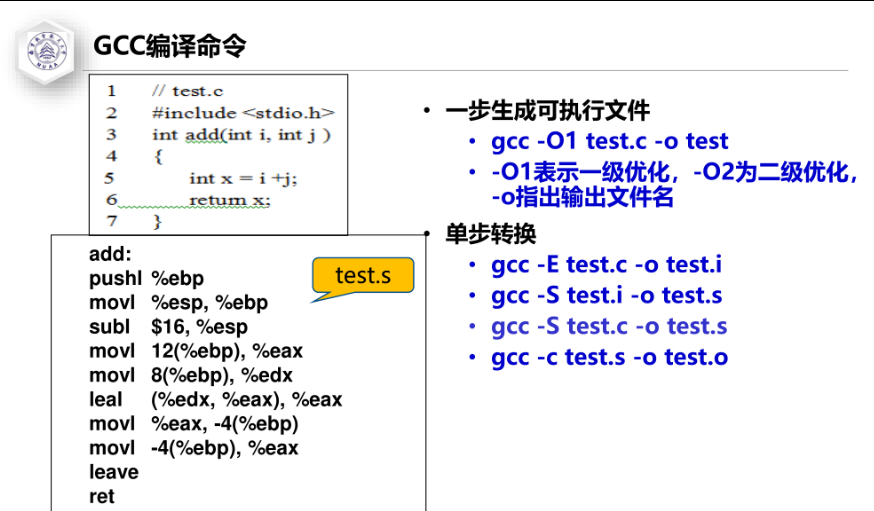


cmp指令：右边 – 左边

Setcc指令：如果条件成立，R[al] = 1；否则，R[al] = 0

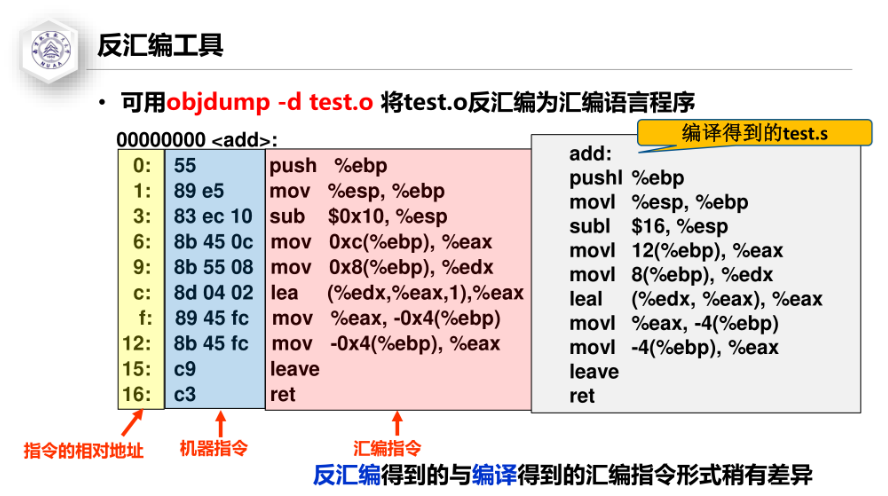
对b无符号扩展为int类型，对c有符号扩展为int类型，cmp对二者作比较，b – c > 0

过程调用的机器级表示



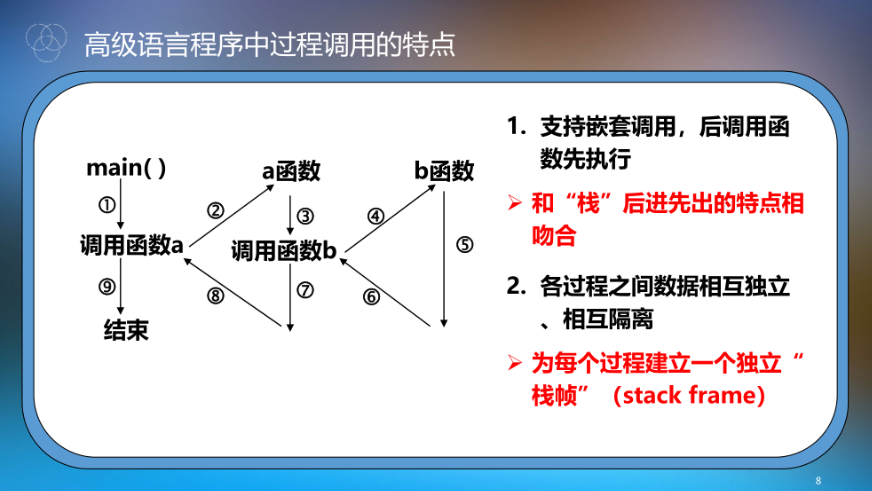
优化级别越高，汇编指令越难看懂

Linux下可执行文件没有后缀（windows下为.exe）

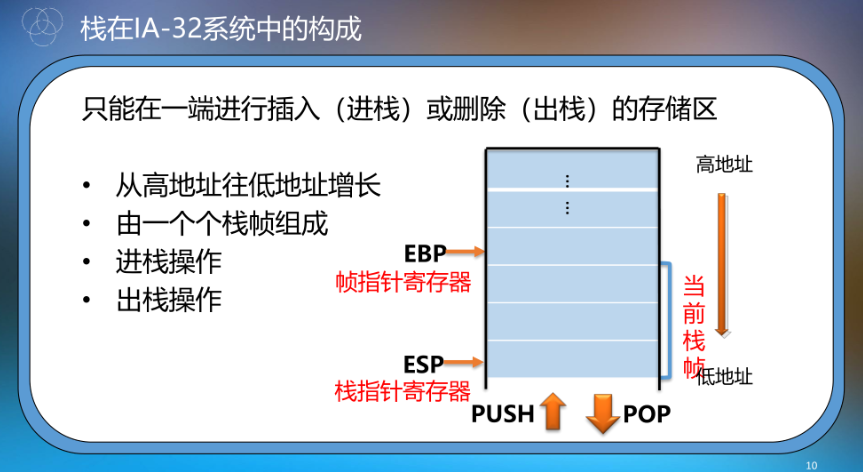


反向：文件有后缀（test.o），立即数是十六进制，地址为相对地址

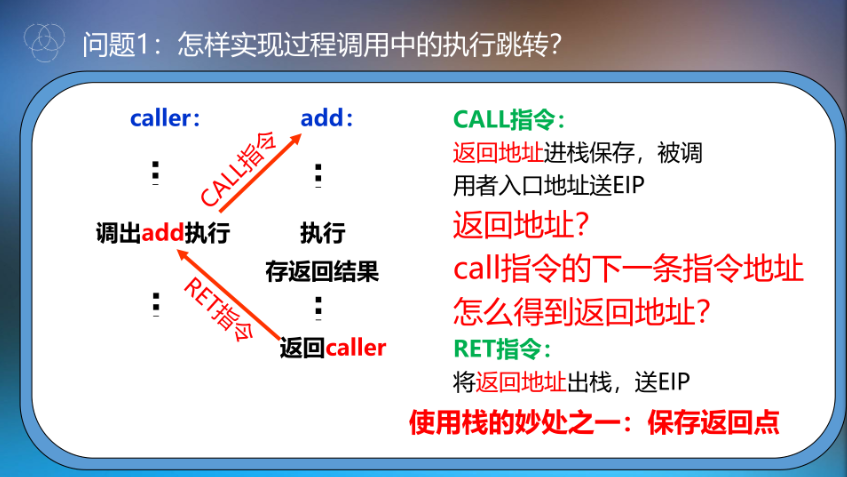
正向：文件无后缀（test），立即数是十进制，地址绝对地址



2. 为每个过程建立一个独立的存储区，并动态管理



分析：函数调用时，上一级调用者的帧底被压入当前ebp内容所指的地址，也就是当前帧的帧底位置保存了上一级调用者的ebp指针值（帧底），而每个ebp的前一个单元存放的就是当前函数的返回地址（它是由调用者在call指令中入的栈）。这样就可以根据当前ebp的值回溯出整个任务的调用栈（调用过程）。



CALL指令：调出函数执行

1. 返回地址进栈保存

pushl %eip

R[esp] <- R[esp] - 4

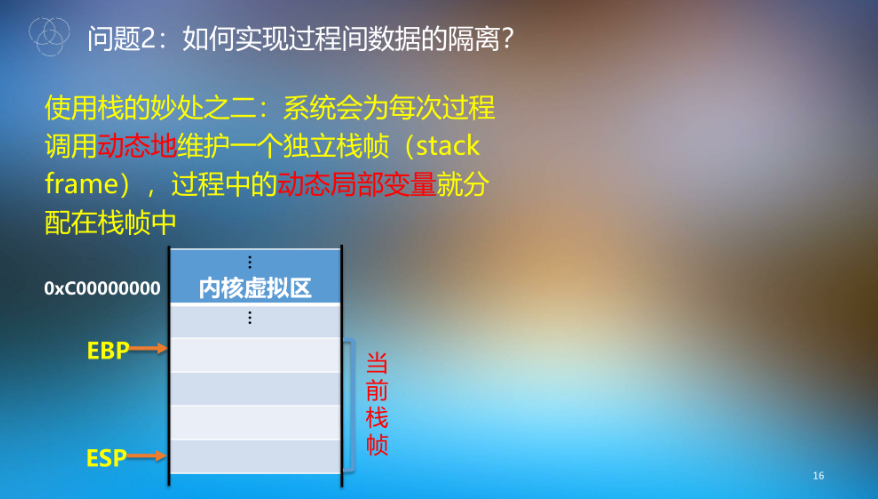
1. 被调用者入口地址（相对寻址）送EIP

R[eip] <- R[eip] + 4

RET指令：

将返回地址出栈，送EIP（绝对寻址）

popl %eip

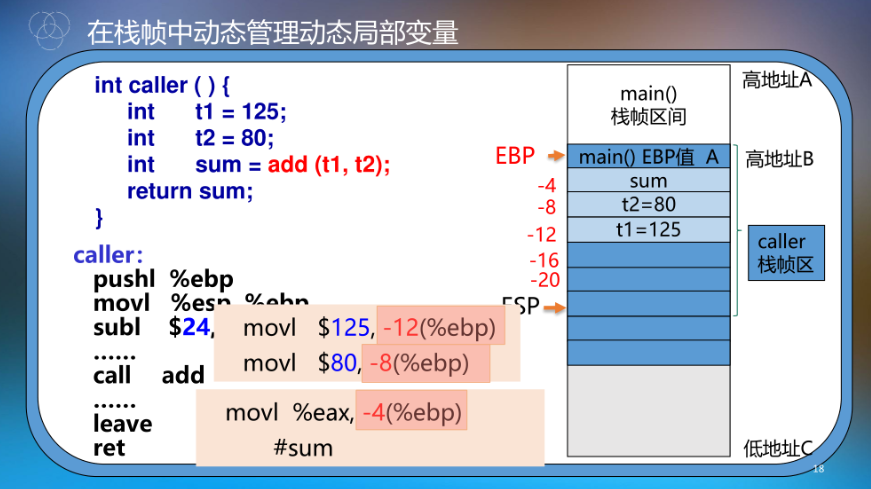


ESP 指向栈顶

EBP 指向栈底

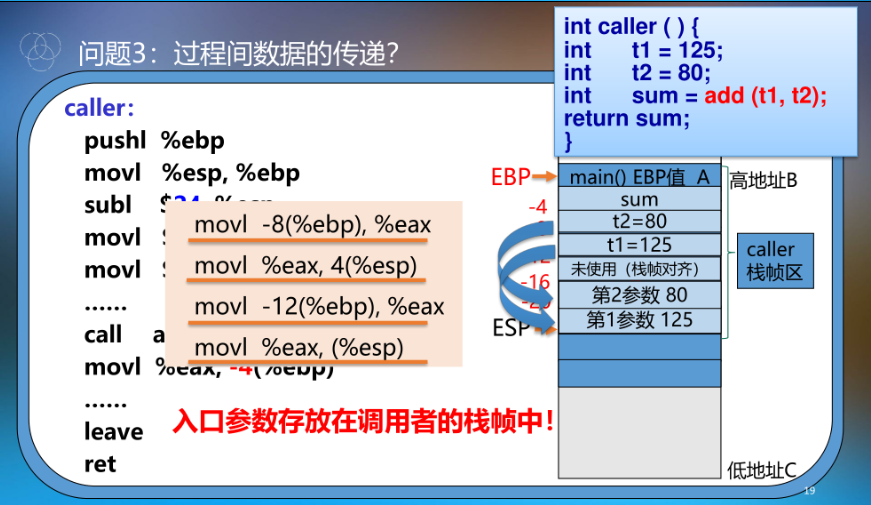


$24是编译器算出来的



eax寄存器存放整型返回值

顺序和编译器有关，可以反过来

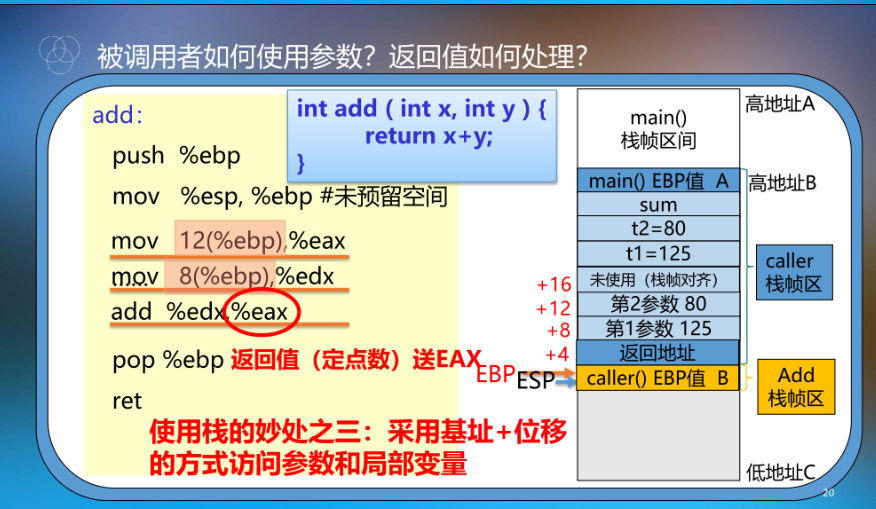


为什么要用eax寄存器作为中转？

mov指令用于寄存器之间、寄存器和存储单元之间的数据传送，不允许两个存储单元数据进行传递

为什么？

因为速度会很慢

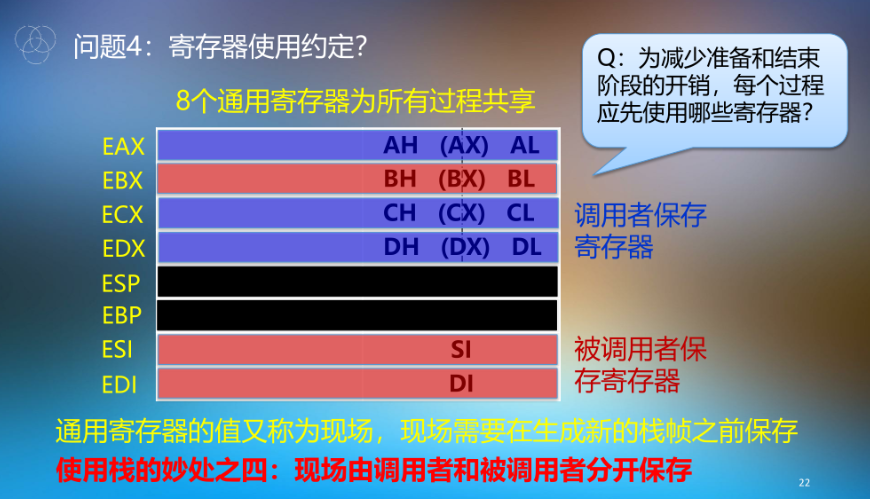


最右边的参数先进栈（因为后进先出）

入口参数存放在调用者的栈帧中，准备好了入口参数，再call add

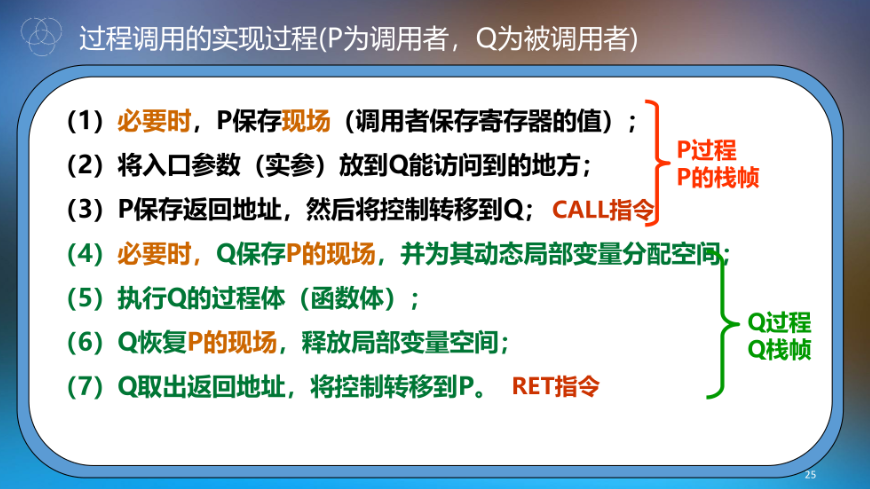
中间未使用是为了栈帧对齐

EBP +4 返回地址，EBP + 8 始终是第一个参数，EBP + 12始终是第二个参数

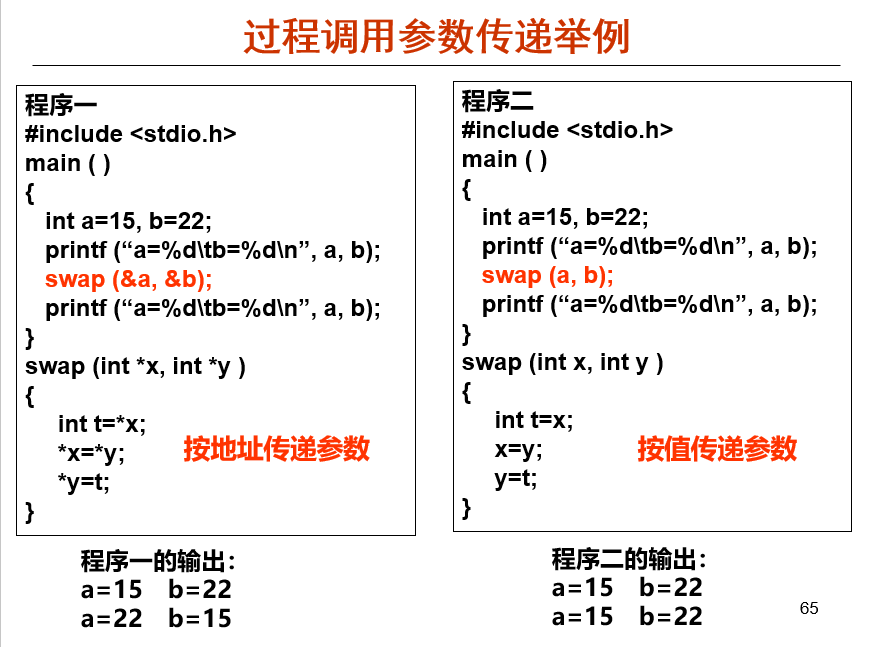


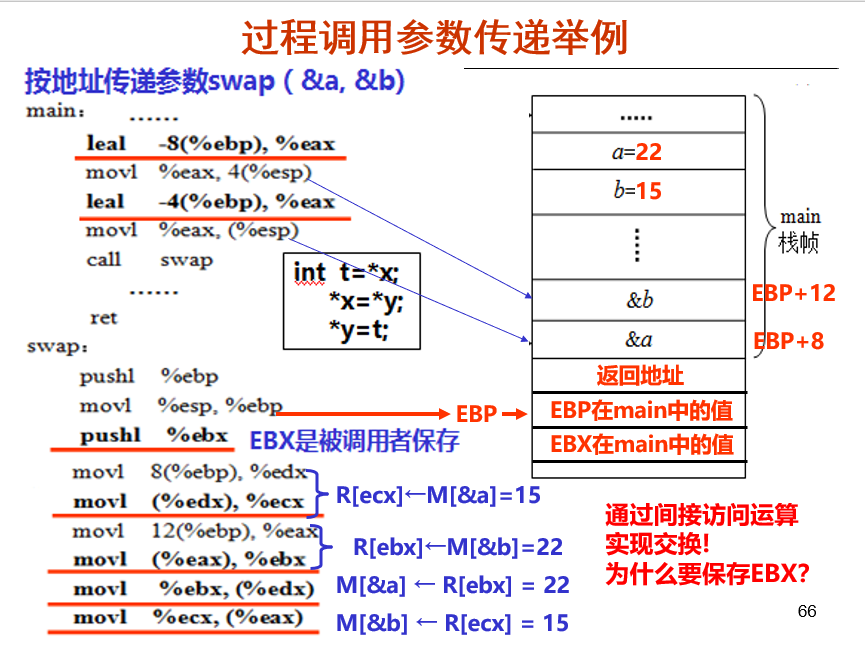
为减少准备和结束阶段的开销，每个过程应先使用哪些寄存器？

调用者保存寄存器



传地址调用 vs 传值调用





main:

1、2：&b -> 4(%esp)

3、4：&a -> (%esp)

call指令: 1. push返回地址%eip 2.转到swap执行（加上相对位移）

swap:

1：存旧的ebp

2：生成新的栈帧的栈底

3： ebx是被调用者保存寄存器

功能：保存现场

为什么用到了ebx？下边把eax、ecx、edx寄存器（调用者保存寄存器）都已经用了，不够用了，所以得用被调用者保存寄存器，由于ebx寄存器的编号在前面，所以一般先用ebx寄存器

4、5：R[ecx]<-M[&a] = 15 %edx a的地址 (%edx) a的值

6、7：R[ebx]<-M[&b] = 22

8：b的原值传给a

9：b的原值传给b

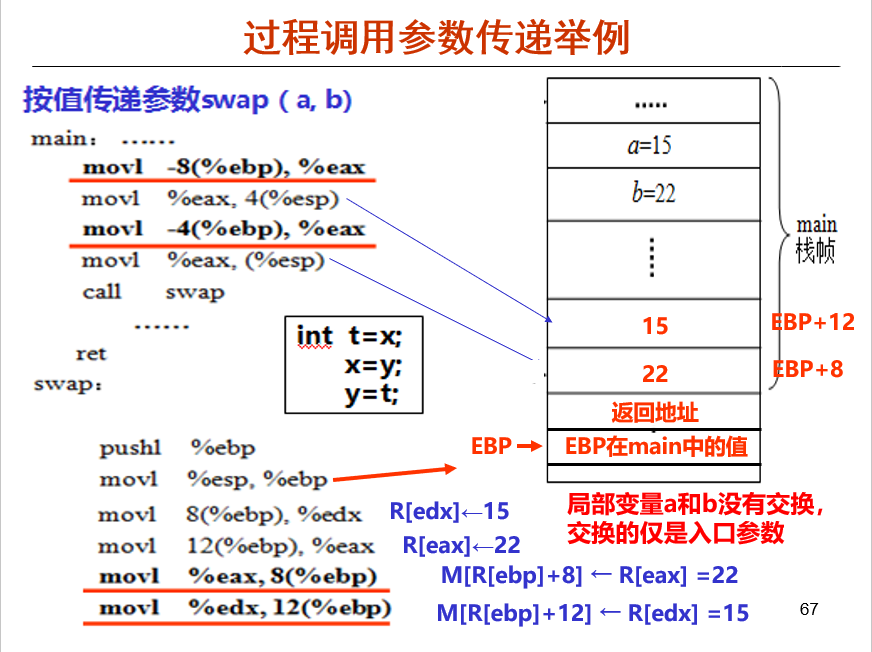
在这个过程中，ecx相当于C语言中的 t变量

C语言中用三个变量即可实现两个变量数据之间的交换，为什么这里用到四个寄存器？

因为\*x = \*y这个操作在两个内存单元进行，不能直接用mov指令传，所以必须用ebx寄存器中转，所以多了一个寄存器

汇编指令层面上的过程：

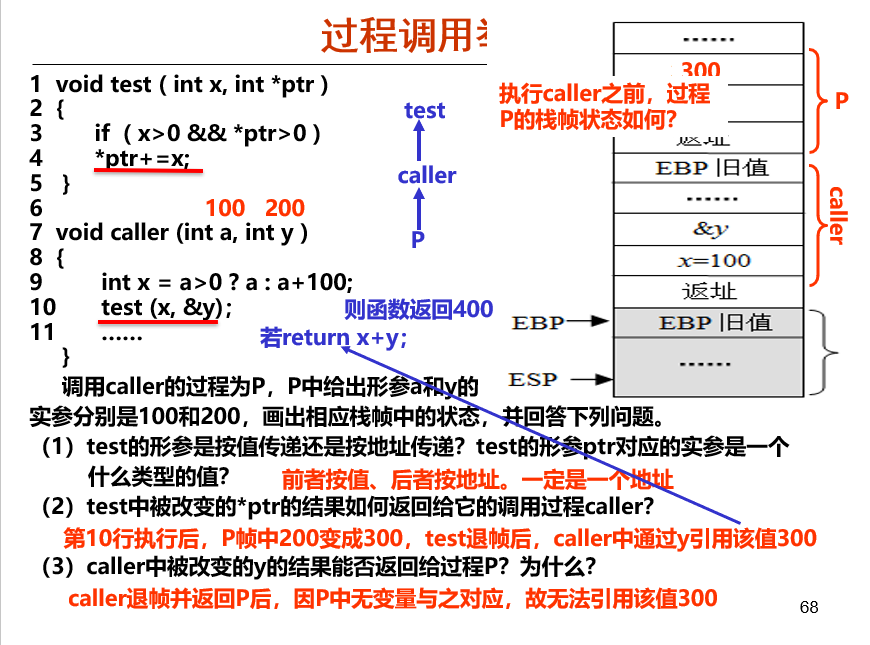
1. 传地址
2. 地址送到寄存器
3. （寄存器）基址寻址方式实现间接访问

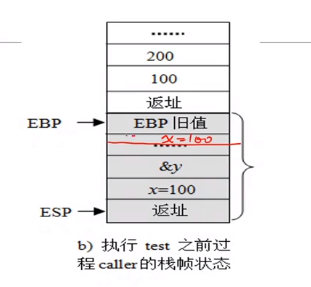
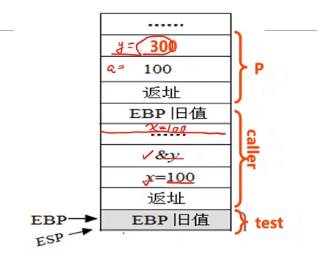


为啥没有用ebx寄存器？因为两个寄存器够用了，不需要用到四个

这里的交换操作实际上交换的是入口参数的值，原来的变量并没有实现交换

退帧后，栈帧中的形参数据仍存在，但是，C语言中没有形参变量对应，所以获取不了（作用域），不过可以想办法用指针直接访问



test对应的实参是 &y

y = 300是P为调用caller准备的形参

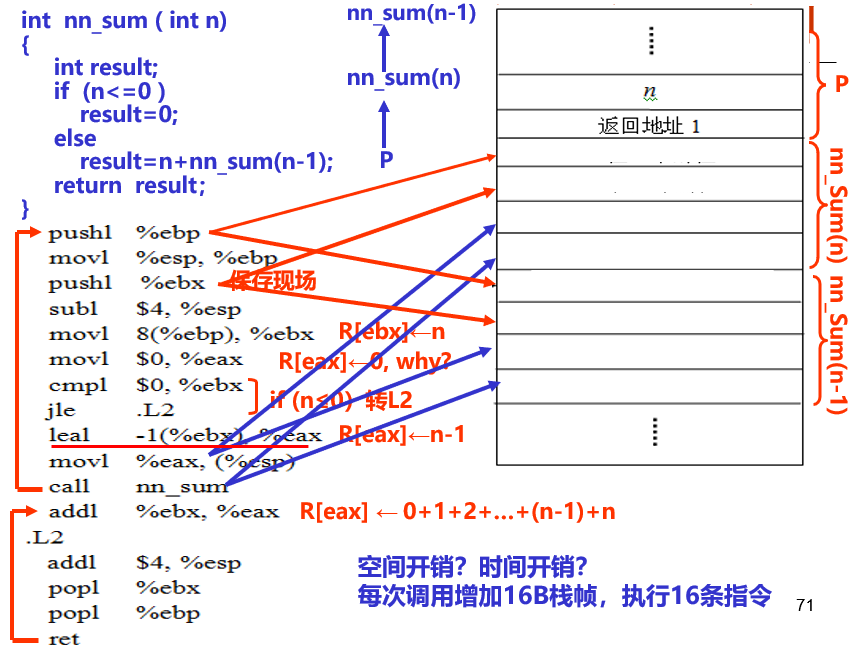
调用test之后执行的过程：

mov 12(%ebp), %eax

movl 8(%ebp), %ecx

add %ecx (%eax)

递归调用实现举例



第一次调用，假设n = 2

1：ebp进栈

3：后面代码寄存器只用了eax，为什么要用ebx？ebx作用是什么？保存每一次的加数

4：为什么长出4个字节？保存入口参数

5：R[ebx] <- n

6：R[EAX] <- 0，为什么？对应于C语言代码中的语句：result = 0

7：n - 0

8：有符号判断是否小于等于0

9：leal指令的特殊用法：仅作运算，没有加载地址

9、10：R[eax] <- n-1，功能：准备入口参数

11：返回地址2进栈，调用递归函数执行

第二次调用

3：由第一次调用的5可知，保存了n

5：R[ebx] <- n-1 = 1

9、10：R[eax] <- n-2 = 0

11：返回地址3进栈，调用递归函数执行

第三次调用

3：由第二次调用的5可知，保存了n - 1

5：R[ebx] <- n-2 = 0

7、8：R[ebx] == 0，跳转到L2.

L2.

1：跳转到这，第三次调用的4空出的4个字节没有用，加回去

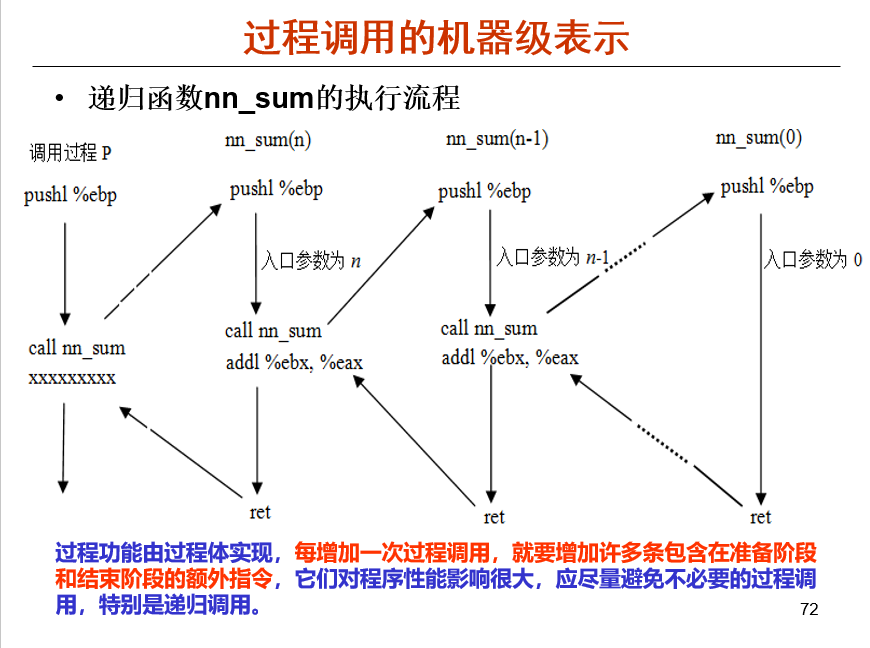
2：pop出第三次调用的3保存的1

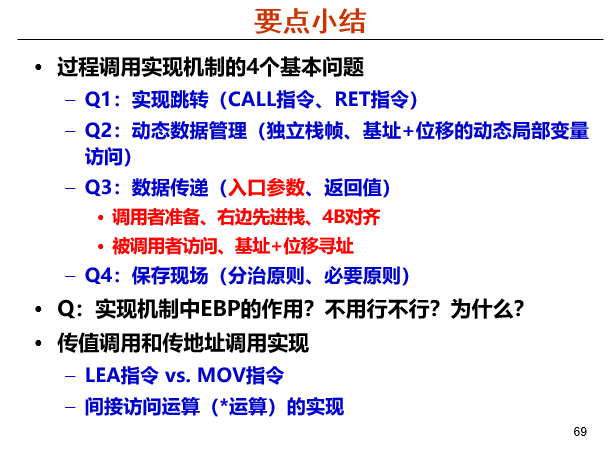
3：退帧，回到nn\_sum(1)栈帧中

回到指令12：实现 R[eax] + R[ebx] = R[eax]，即R[eax] = 0 + 1

继续退帧，pop出2，回到nn\_sum(2)的栈帧中，R[eax] = 0 + 1 + 2

前面四条指令是额外开销





4B对齐：方便基址+位移访问

Q：可以只用esp；

用ebp会占用很多时空资源，但是可以方便的访问变量

当只用esp时，最好要对齐