程序的机器级表示III：过程

教师：夏文
计算机科学与技术学院
哈尔滨工业大学深圳硬件和系统教研室

章节要求

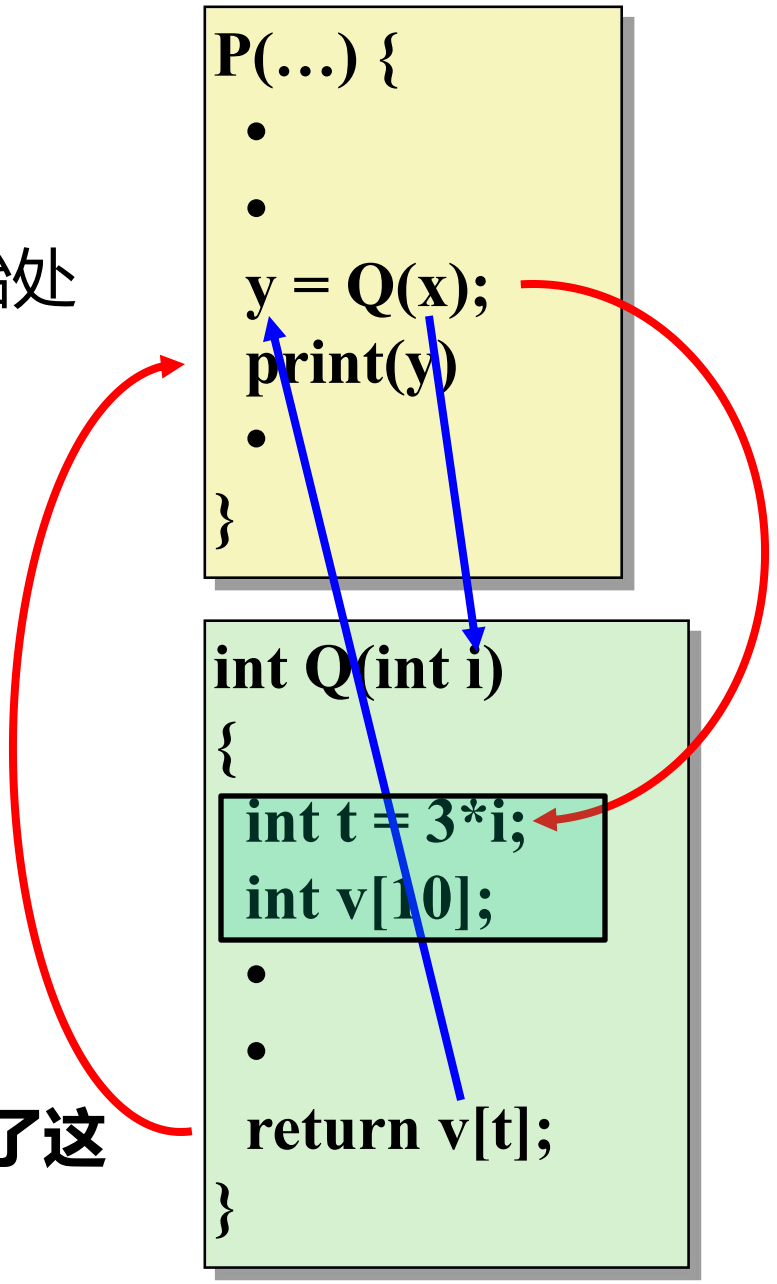
- **本节所有内容都是重点内容**
- **要求：**
  - 理解和掌握包括但不限于（过程的机制，运行时栈，栈帧，控制和数据的传递，被调用保存和调用者保存，通用寄存器的作用，局部数据的管理）。
  - 能够熟练将c代码和汇编代码相互转换（建议可以使用编译器如Visual Studio进行反汇编学习）
  - 完成并理解课后习题

主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - 管理局部数据
  - 递归

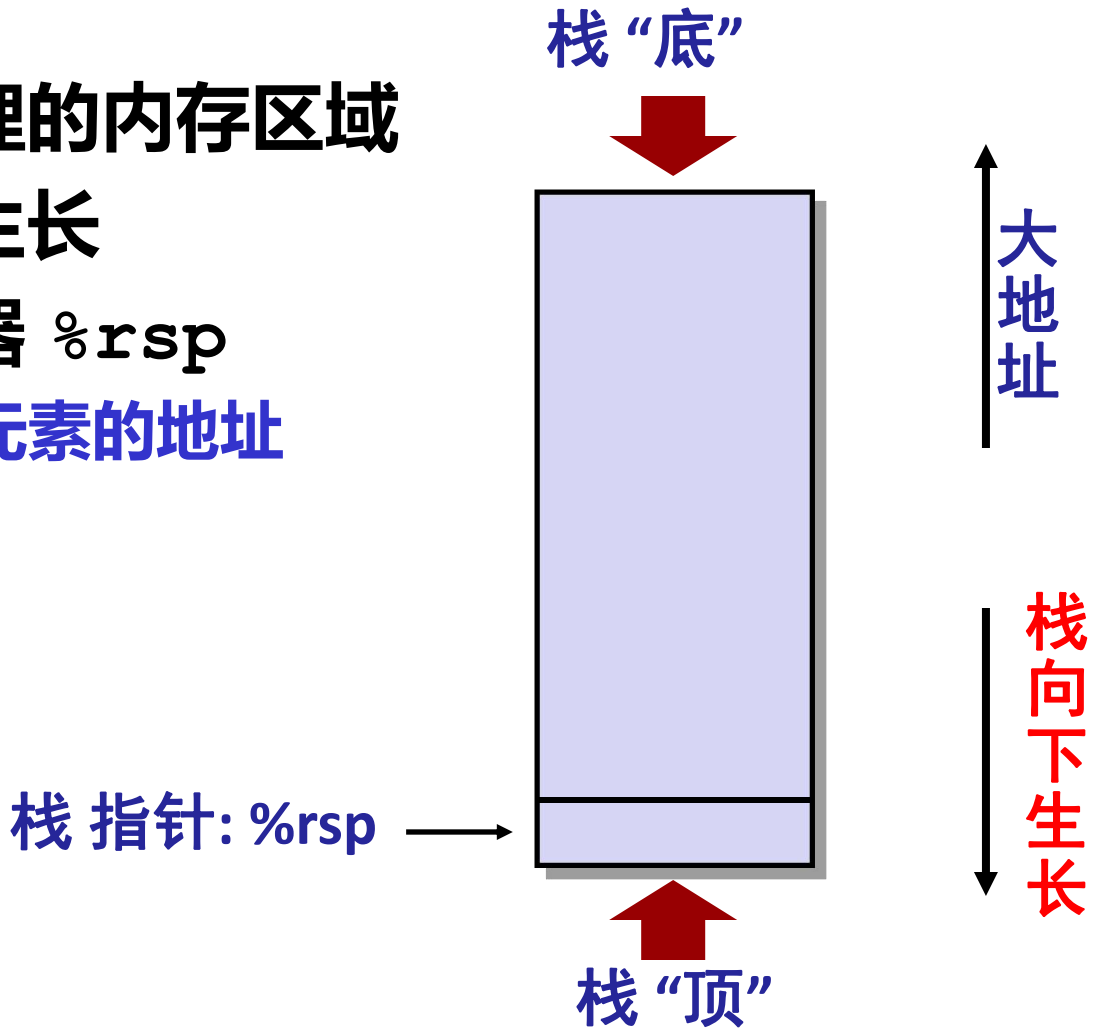
过程的机制

- **传递控制**
  - 调用：转到过程代码的起始处
  - 结束：回到返回点
- **传递数据**
  - 过程参数
  - 返回值
- **内存管理**
  - 过程运行期间申请
  - 返回时解除分配
- **该机制全部由机器指令实现**
- **x86-64 过程的实现只是使用了这些机制**



x86-64 栈

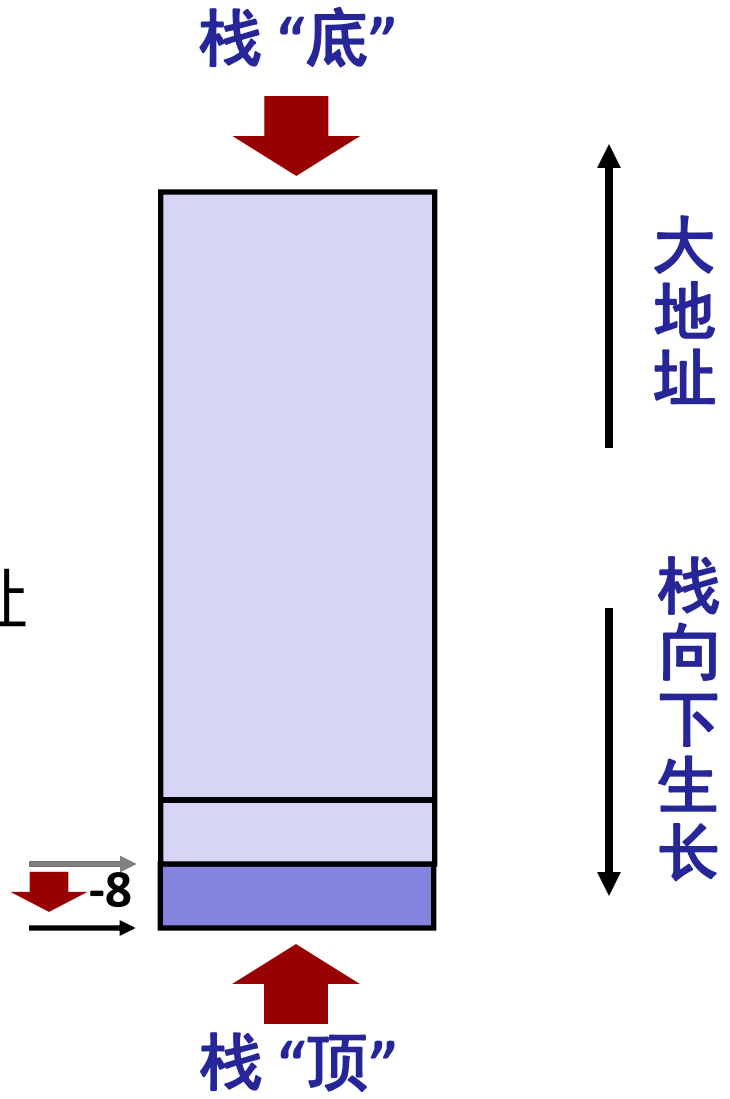
- 使用栈规则管理的内存区域
- 向低地址方向生长
- 栈指针：寄存器 `%rsp`
  - 保存 栈 “顶”元素的地址



x86-64 入栈指令: push

- **入栈指令 pushq**
  - 格式:

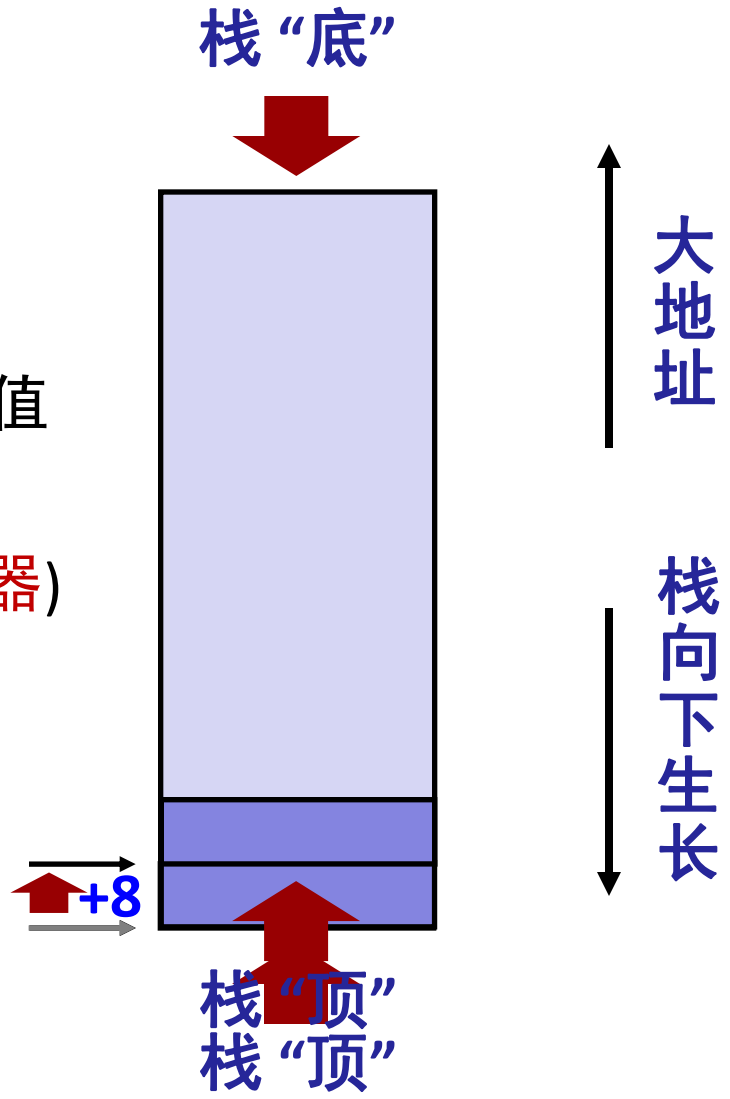
    pushq Src
  - 从Src取操作数
  - 将%rsp减8
  - 将操作数写到%rsp指向的地址



x86-64 出栈指令: pop

- 出栈指令 popq
  - 格式：

    popq Dst
  - 从%rsp中保存的地址值读取数值
  - 将 %rsp加 8
  - 将数值保存到Dst (必须是寄存器)



主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - 管理局部数据
  - 递归与指针的解释

代码示例

```
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                    # Save %rbx
  400541: mov    %rdx,%rbx               # Save dest
  400544: callq  400550 <mult2>          # mult2(x,y)
  400549: mov    %rax,(%rbx)             # Save at dest
  40054c: pop    %rbx                    # Restore %rbx
  40054d: retq                           # return
```

```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax# a
  400553:  imul   %rsi,%rax # a * b
  400557:  retq                    # return
```

过程控制流

- **栈：支持过程的调用、返回**
- **过程调用**

```
call  func_label
```

  - 返回地址入栈(Push)
  - 跳转到func_label (函数名字就是函数代码段的起始地址)
- **返回地址:**
  - 紧随call指令的下一条指令的地址（考虑PC——RIP的含义)
- **过程返回**

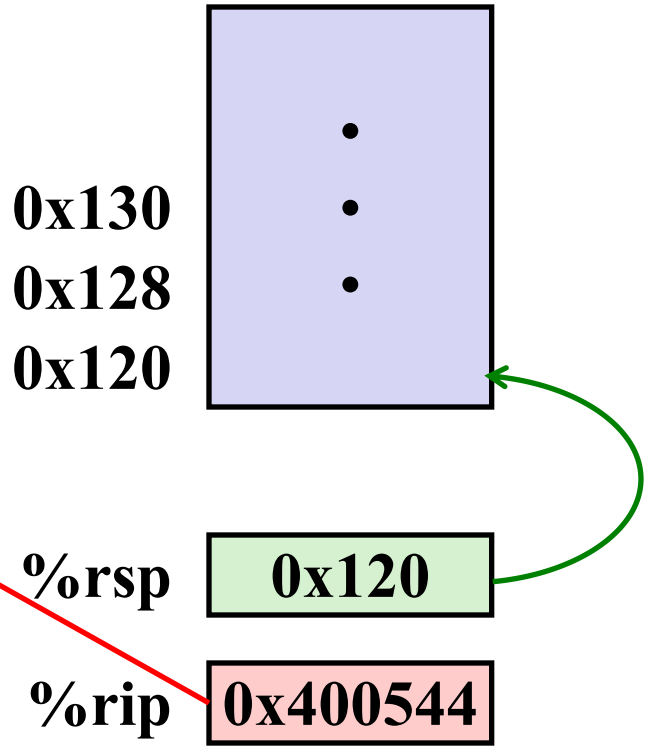
```
ret
```

  - 从栈中弹出返回地址(pop)
  - 跳转到返回地址

控制流—1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

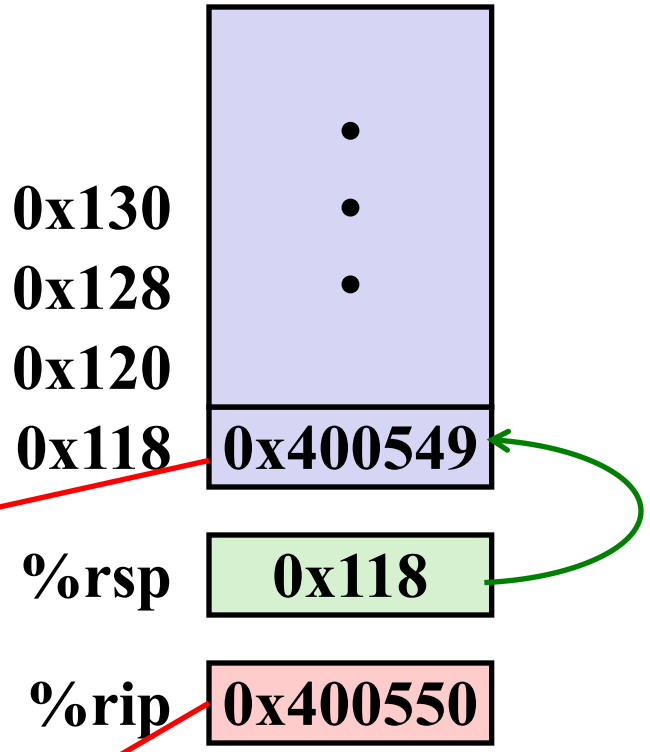


下一条指令是过程调用

控制流 —2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

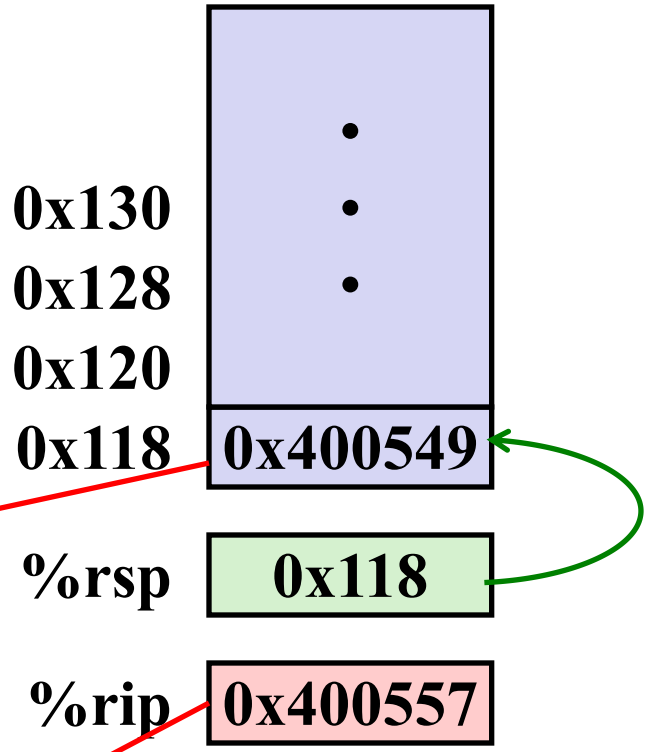


返回地址入栈，
下一条指令是call指令后的地址

控制流 —3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

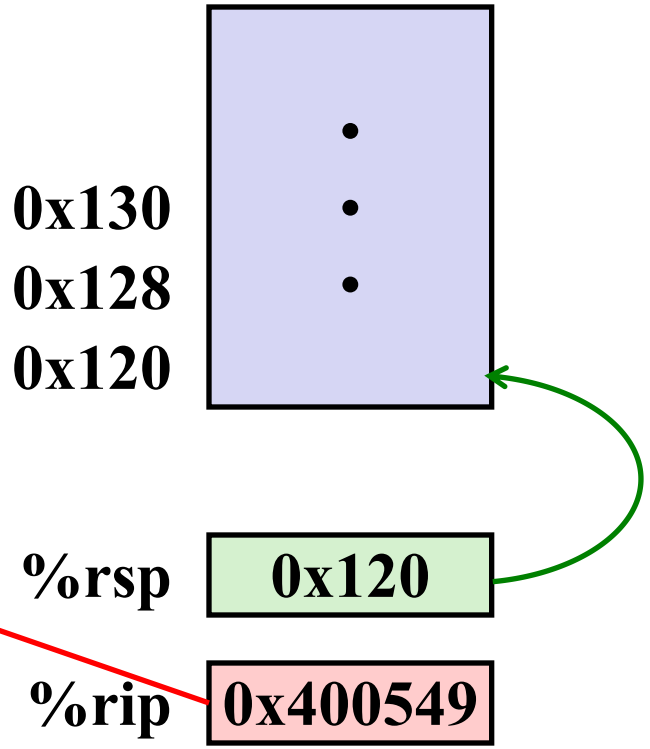


下一条是返回指令

控制流 —4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



返回地址出栈作为
下一条指令地址

主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - **传递数据**
    - 管理局部数据
  - 递归与指针的解释

过程数据流

- 参数传递
  - 前6个参数用寄存器

%rdi	Arg 1
%rsi	Arg 2
%rdx	Arg 3
%rcx	Arg 4
%r8	Arg 5
%r9	Arg 6

  - 其余参数用栈（注意顺序）

…
Arg *n*
…
Arg *8*
Arg *7*

返回地址

- 返回值

%rax

- 局部变量：仅在需要时申请栈空间

数据流示例

```
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
 # x in %rdi, y in %rsi, dest in %rdx
 ...
 400541: mov    %rdx,%rbx                # Save dest
 400544: callq  400550 <mult2>      # mult2(x,y)
 # t in %rax
 400549: mov    %rax,(%rbx)          # Save at dest
 ...
```

```
long mult2  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax     # a * b
  # s in %rax
  400557:  retq                          # return
```

主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - **管理局部数据**
  - 递归

基于栈的语言

- **支持递归的语言**
  - C、Pascal、Java
  - 代码必须可重入“Reentrant”
    - 单个过程有多个并发实例(simultaneous instantiations)
  - 需要保存每个实例的状态
    - 参数
    - 局部变量
    - 返回地址
- **栈的使用原则**
  - 有限时间内，保存给定程序的状态：从调用的发生到返回
  - 被调用者先于调用者返回
- **栈分配单位——帧，栈中单个过程实例的状态数据**

调用链示例

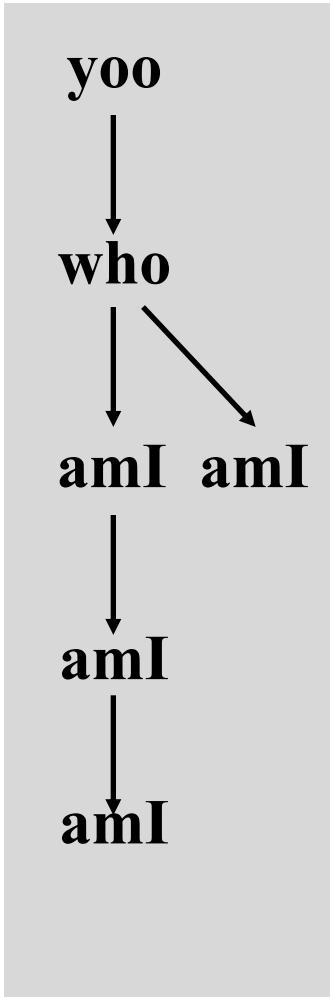
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

过程 amI() 是递归的

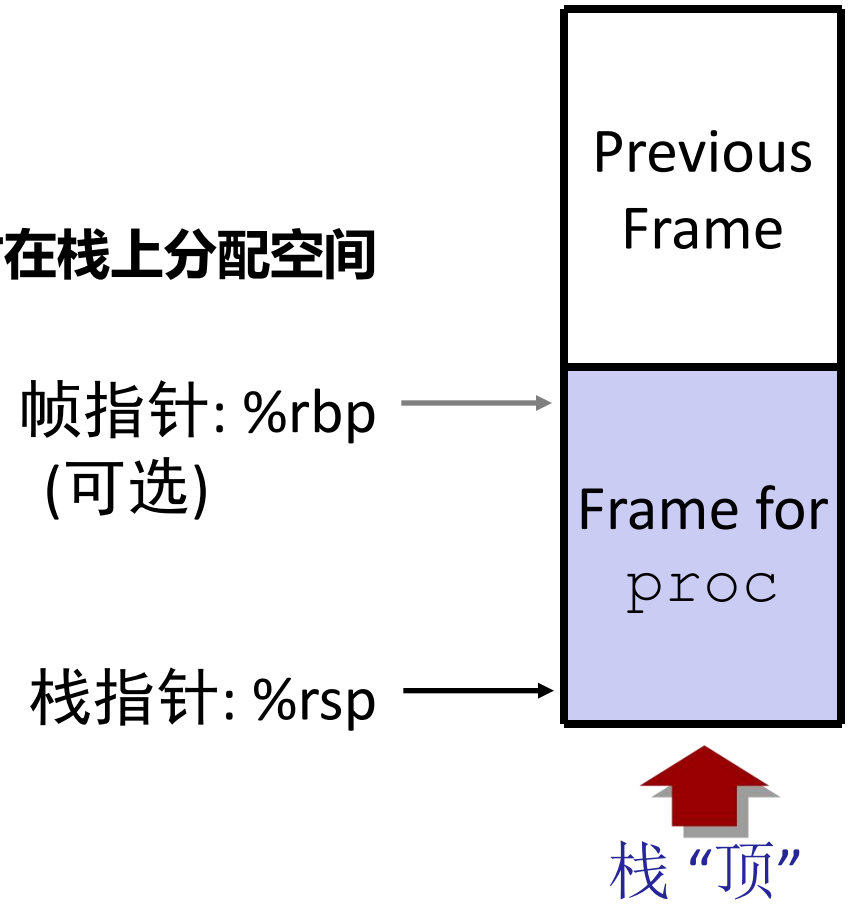
```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

调用链示例

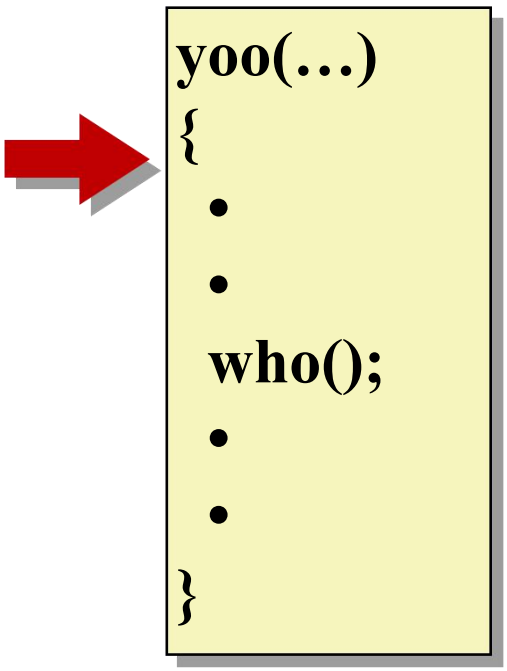


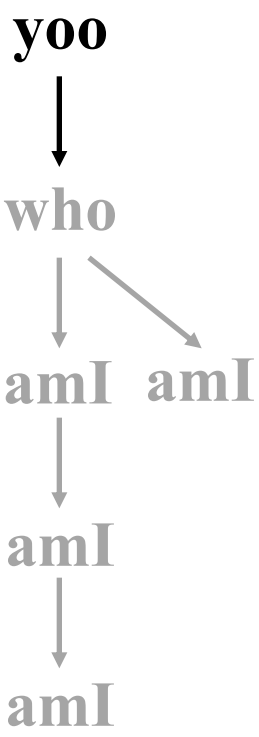
栈帧

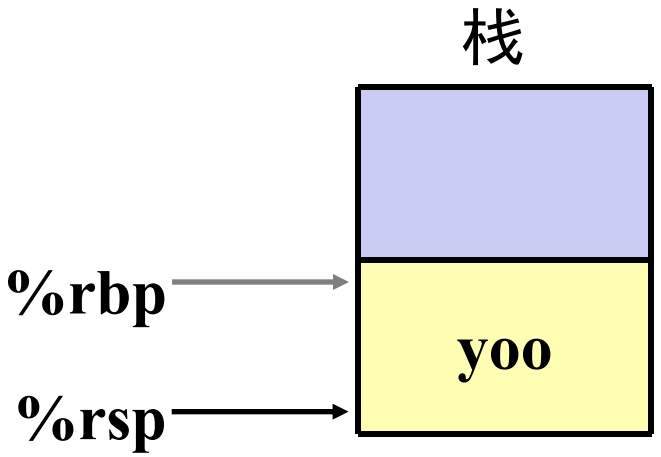
- **过程需要的空间超出寄存器大小时在栈上分配空间**
- **内容**
  - 返回信息
  - 局部存储(如需要)
  - 临时空间(如需要)
- **管理**
  - 进入过程时申请空间
    - 生成代码——构建栈帧
    - 包括**call指令产生的push操作**
  - 当返回时解除申请
    - 结束代码——清理栈帧
    - 包括ret指令产生的pop操作



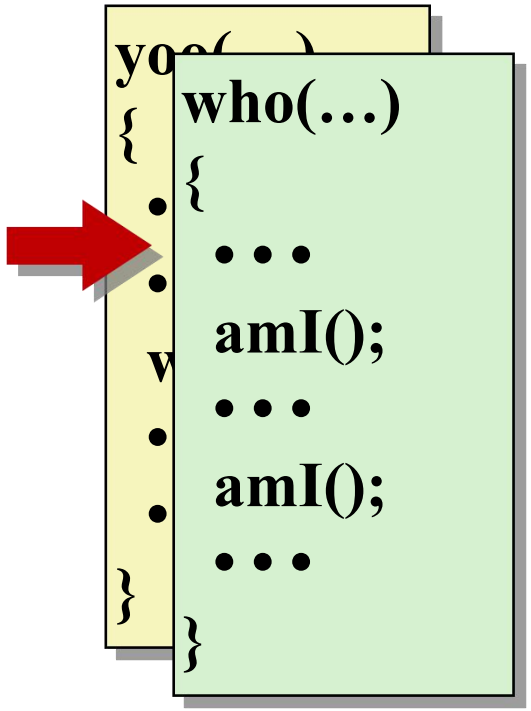
栈帧示例

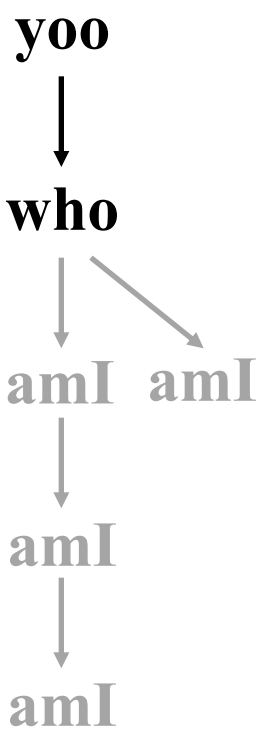


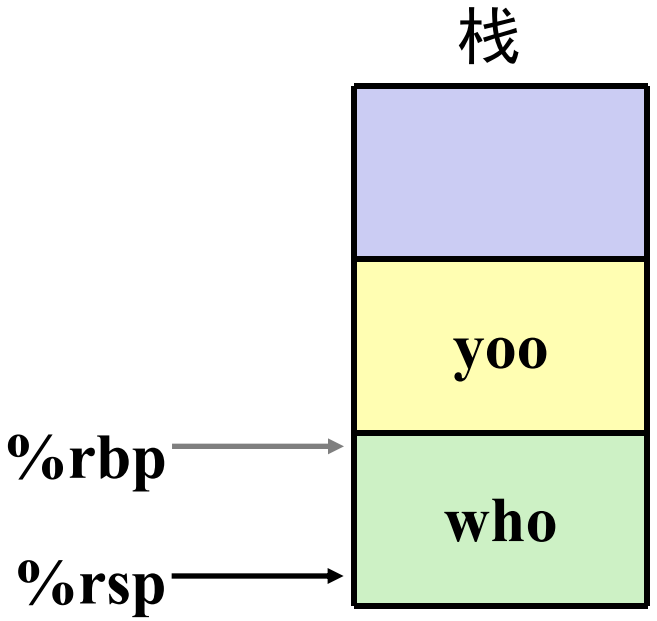




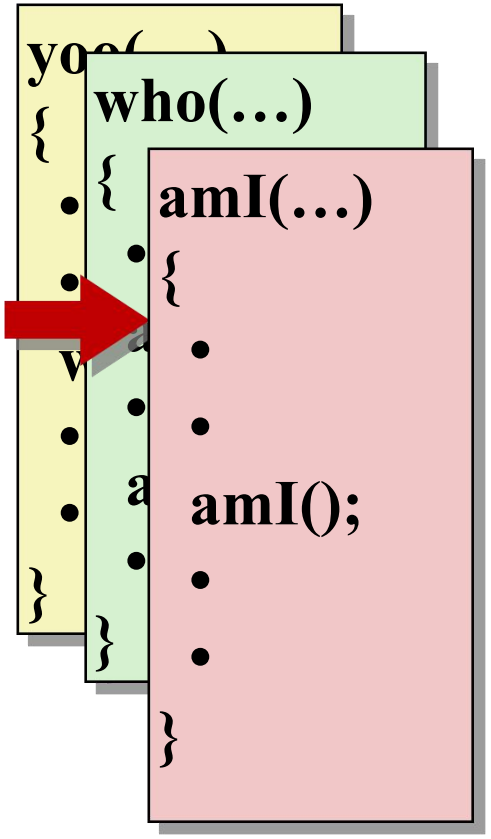
栈帧示例

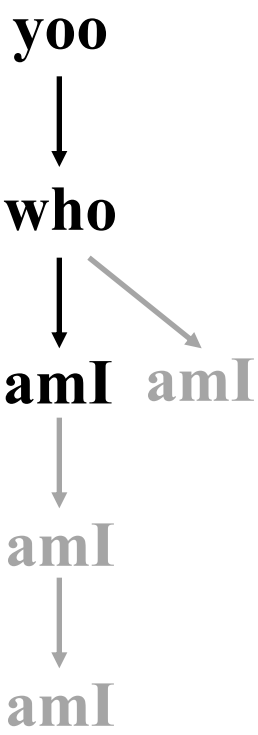


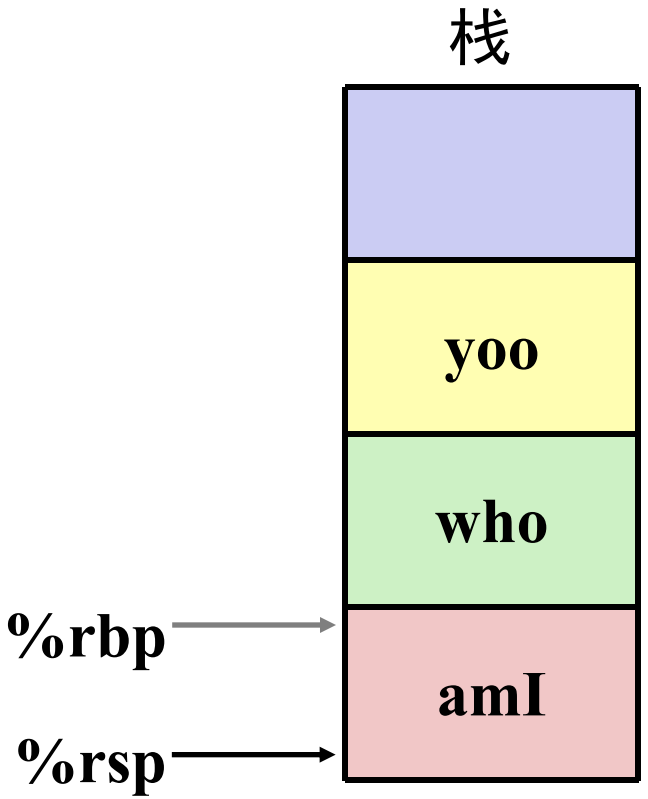




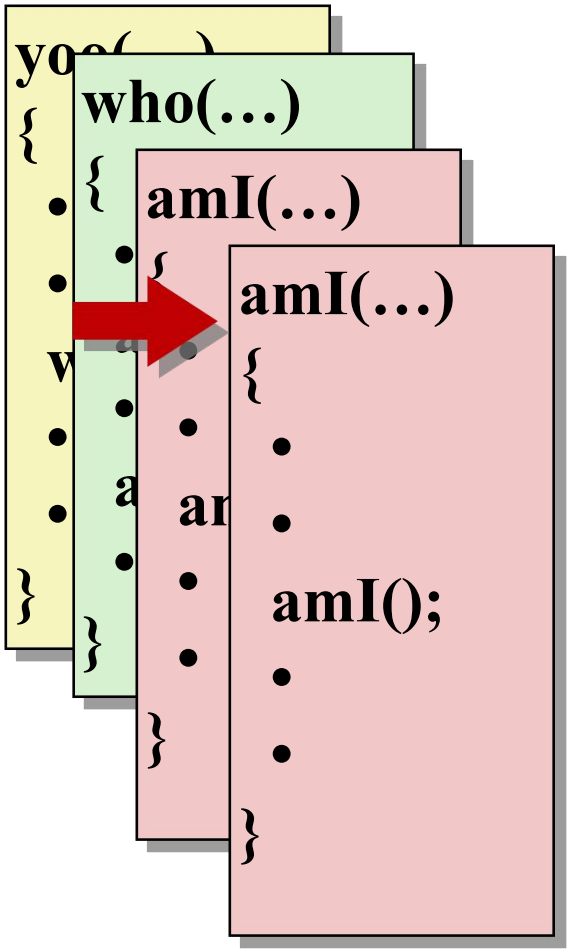
栈帧示例

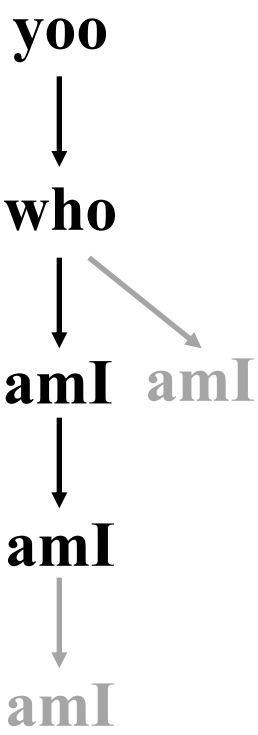


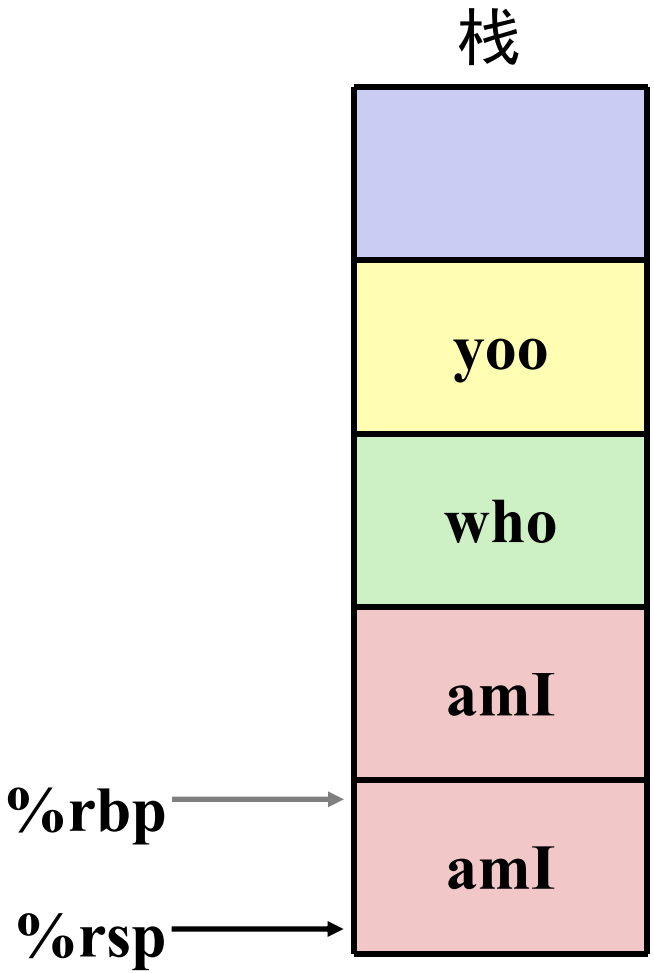




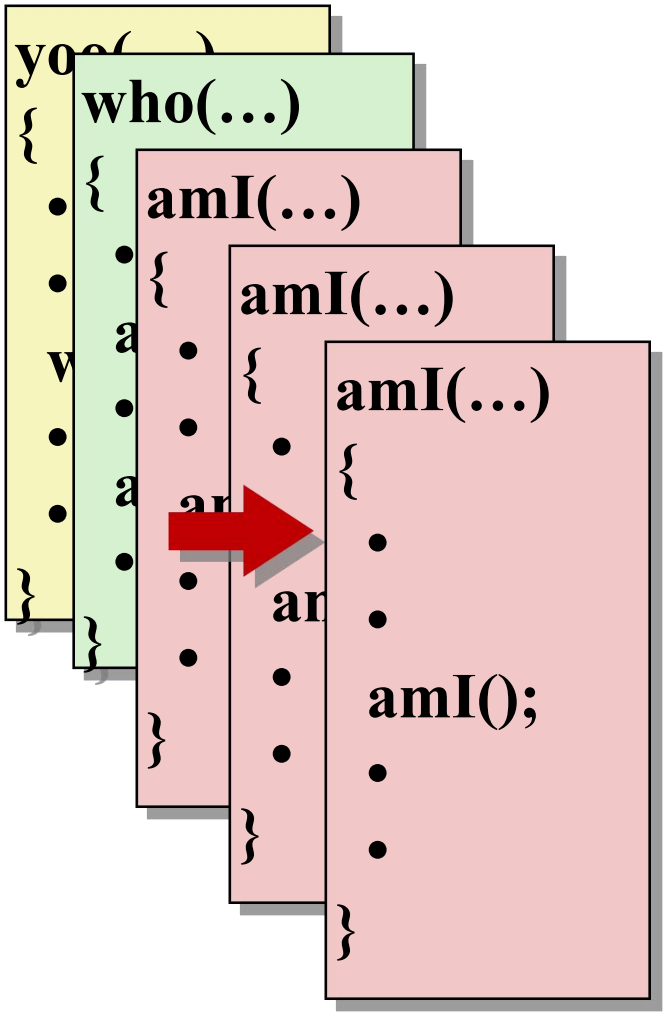
栈帧示例

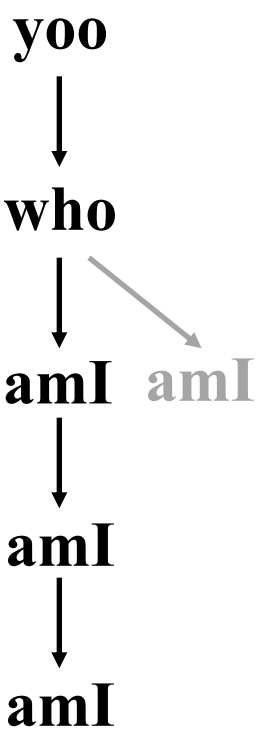


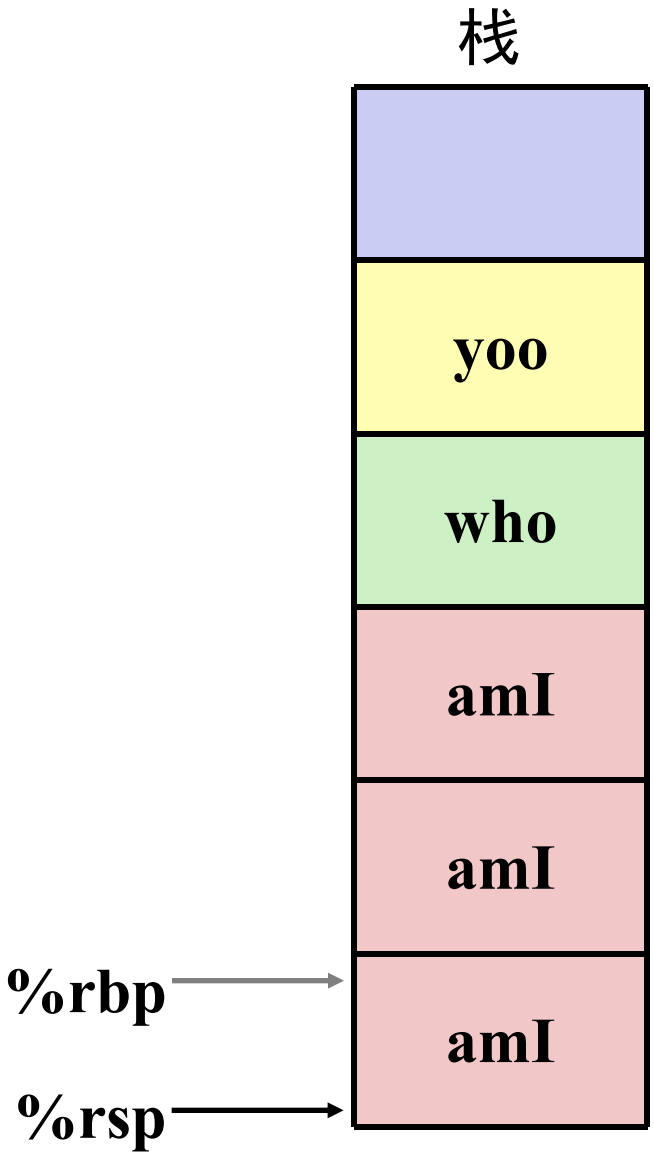




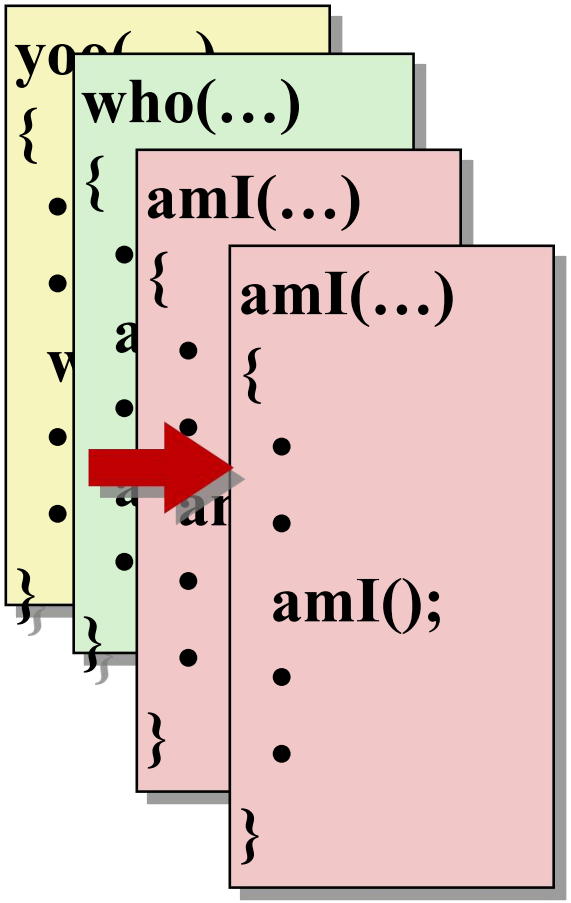
栈帧示例

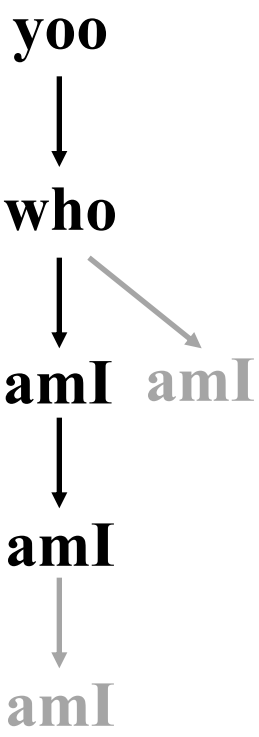

who(…)
amI(…)
amI(…)
amI(…)
amI();

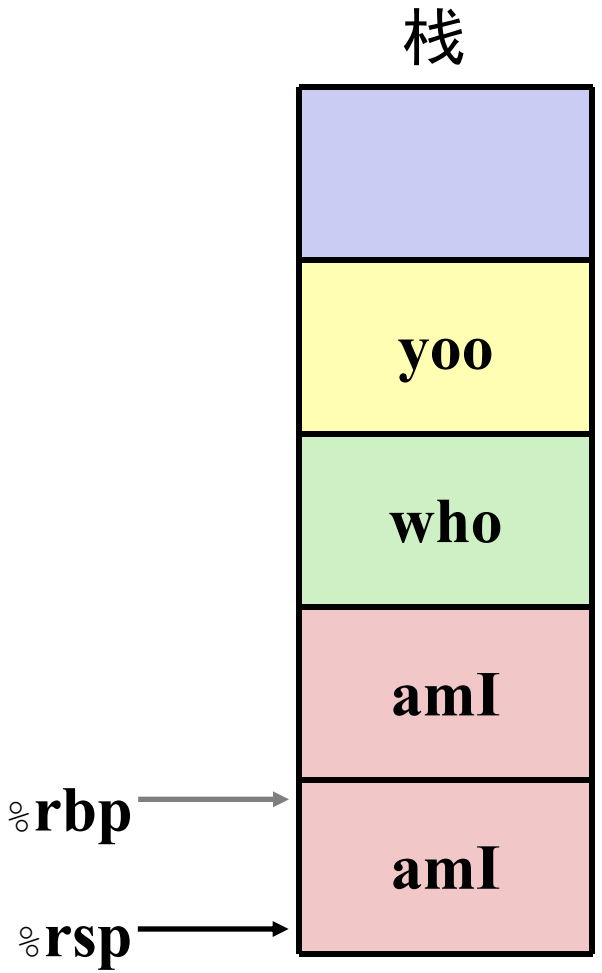

yoo
who
amI
amI
amI
amI


栈
yoo
who
amI
amI
amI
%rbp
%rsp

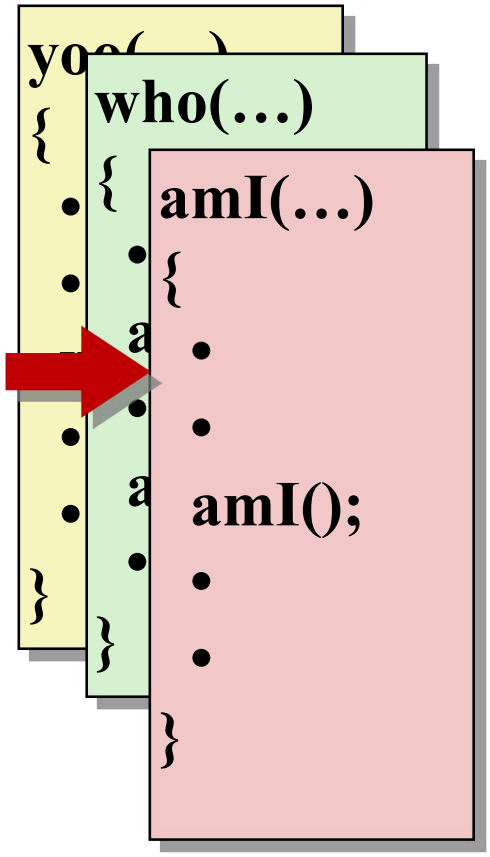
栈帧示例

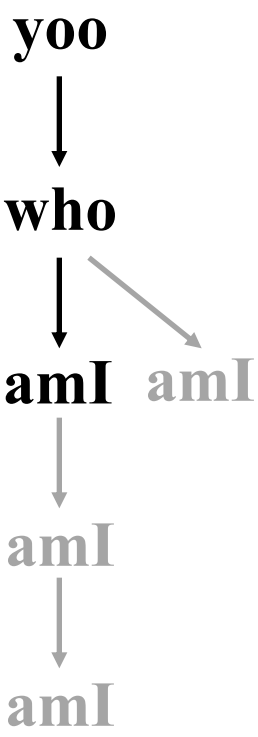


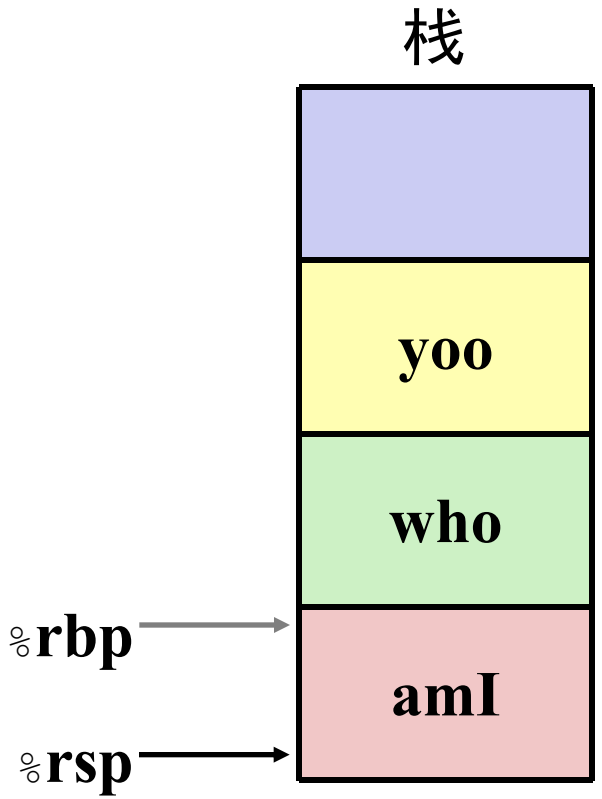




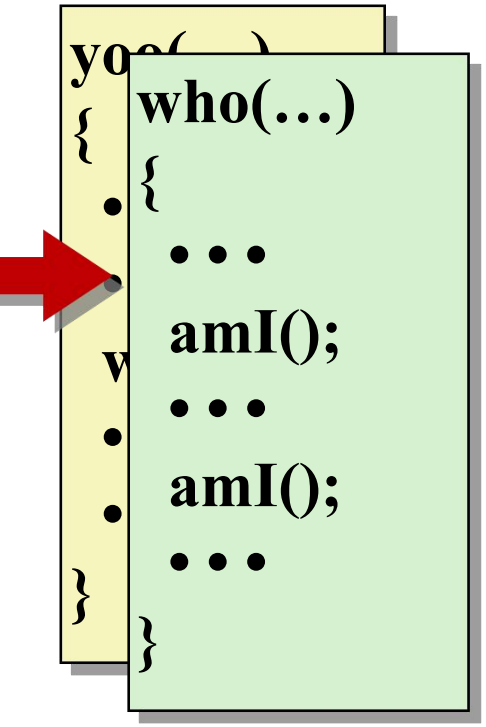
栈帧示例

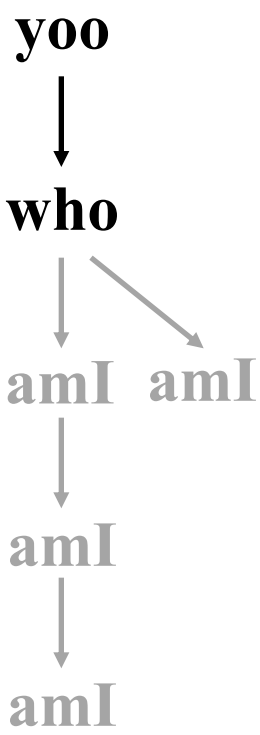


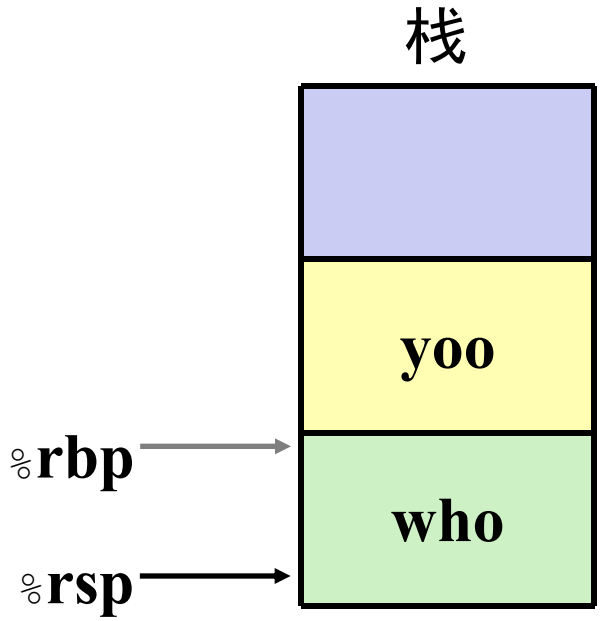




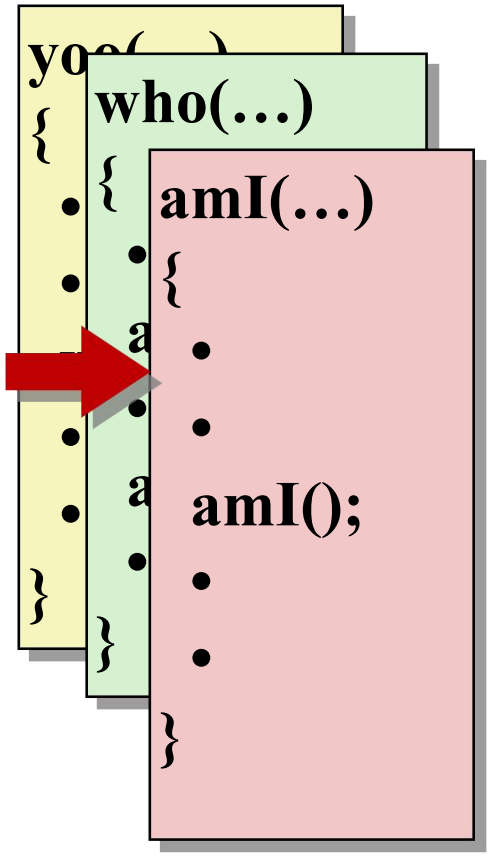
栈帧示例

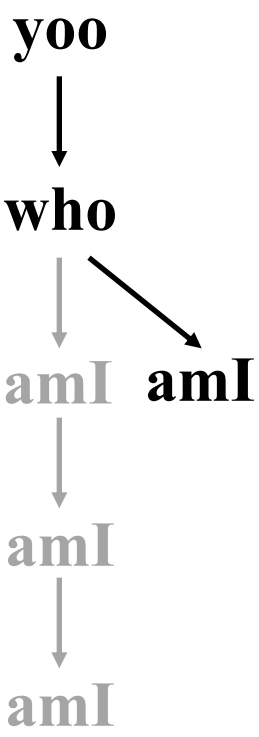


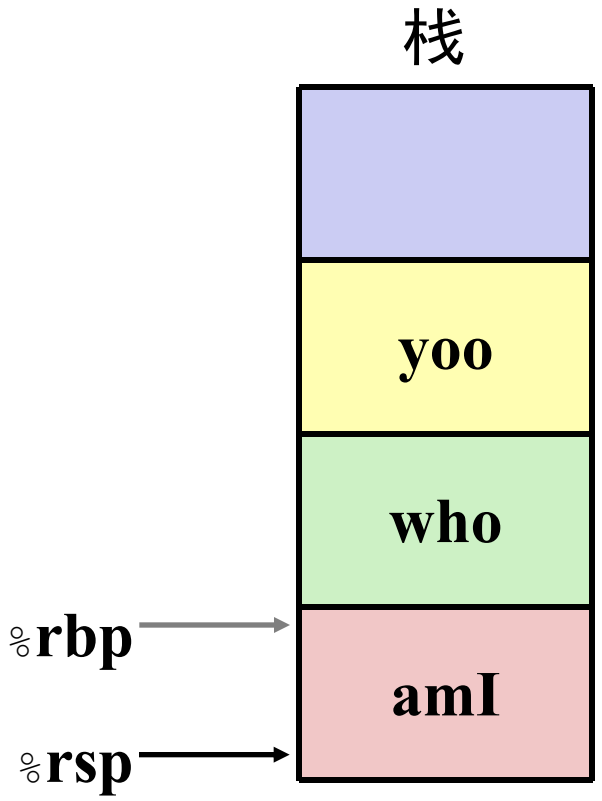




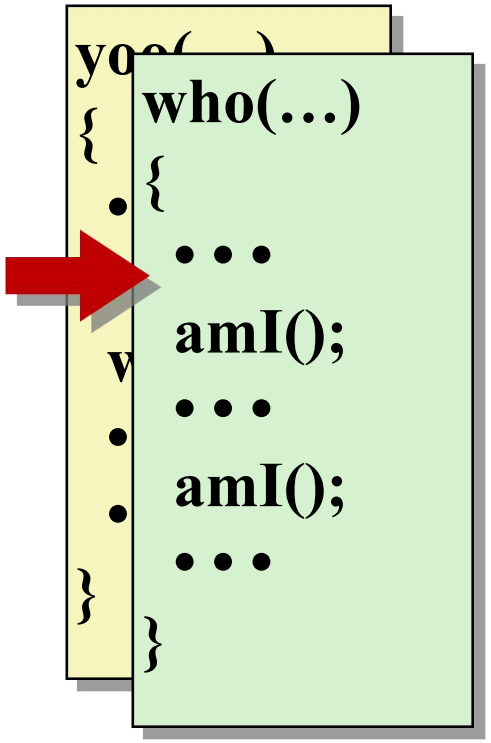
栈帧示例

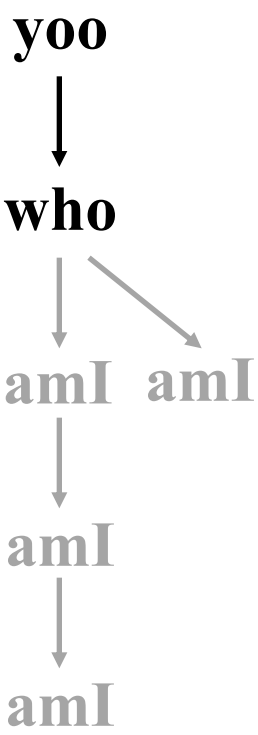


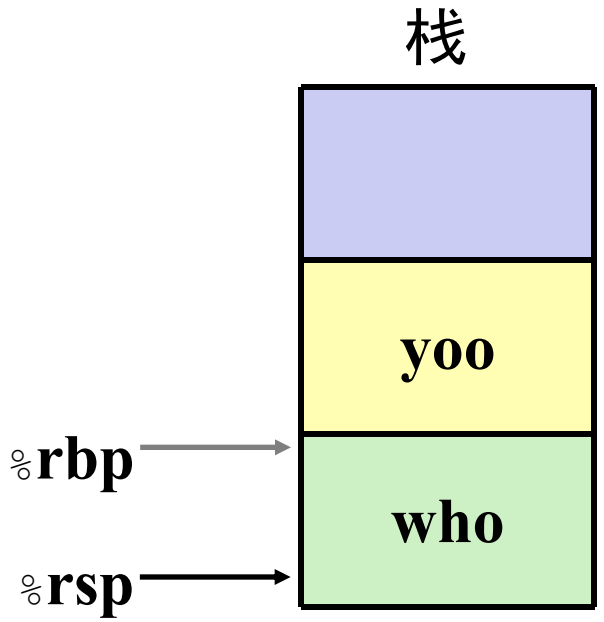




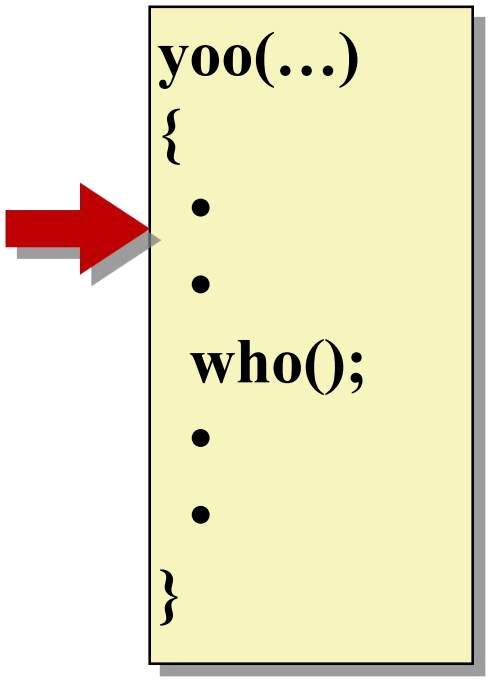
栈帧示例

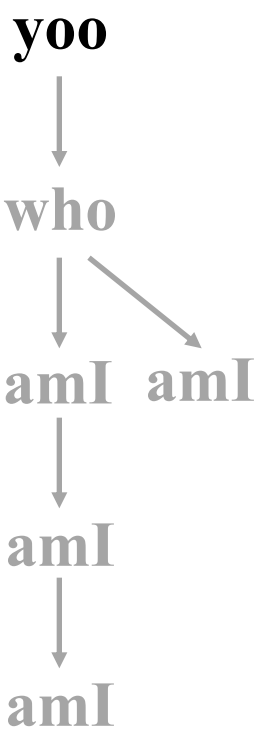


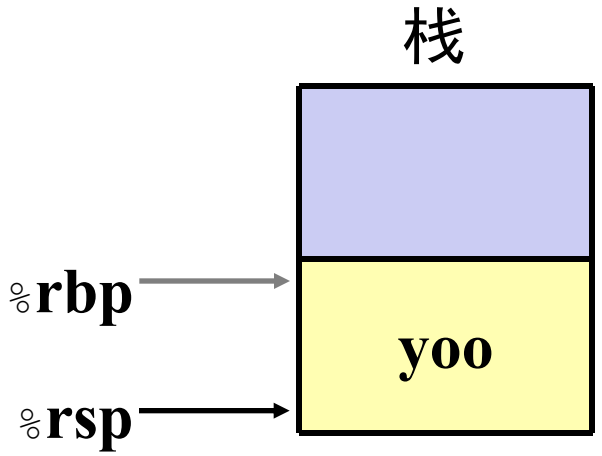




栈帧示例

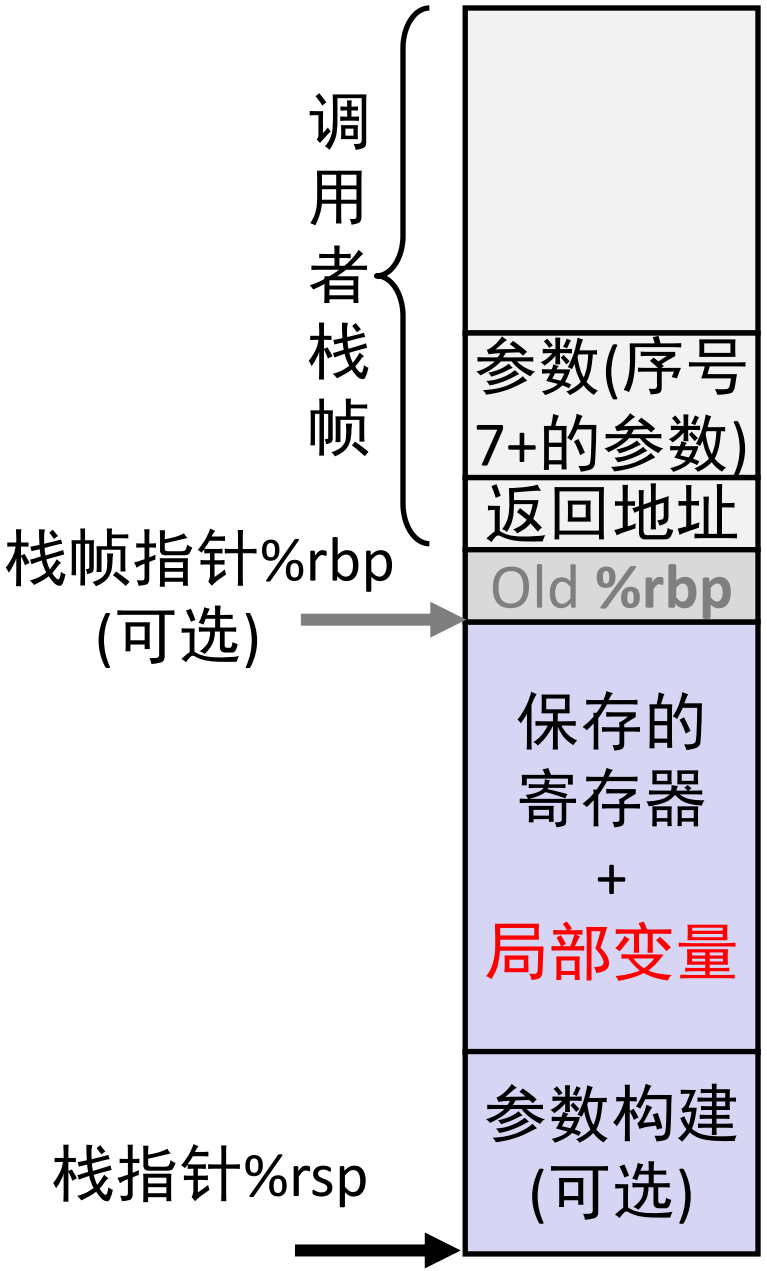






x86-64/Linux 栈帧

- **当前栈帧(从“顶” 到底)**
  - “参数建立”：把即将调用的函数所需参数入栈
  - 局部变量
    如果不能用寄存器实现，则在栈中实现
  - 保存的寄存器内容
  - 旧栈帧指针 (rbp可选)
- **调用者栈帧**
  - 返回地址
    - 由`call`指令压入栈
  - 本次调用的参数



实例: incr

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

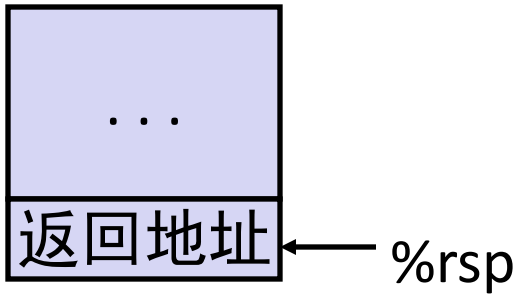
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

寄存器	用途
%rdi	参数 p
%rsi	参数 val, y
%rax	x, 返回值

实例: 调用incr#1

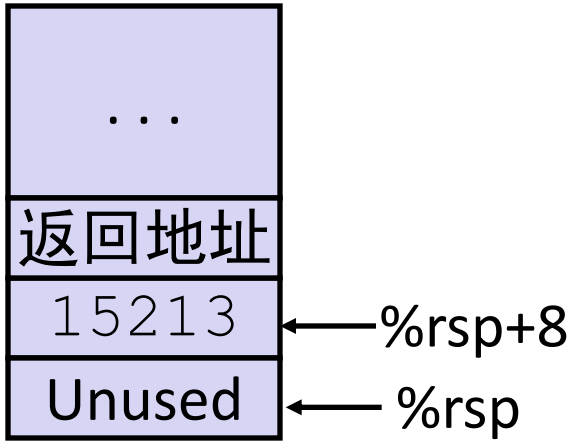
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

初始栈结构




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

结果栈结构

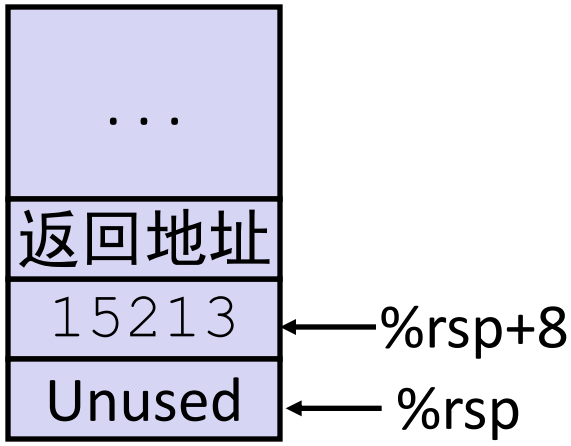


实例: 调用incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈结构



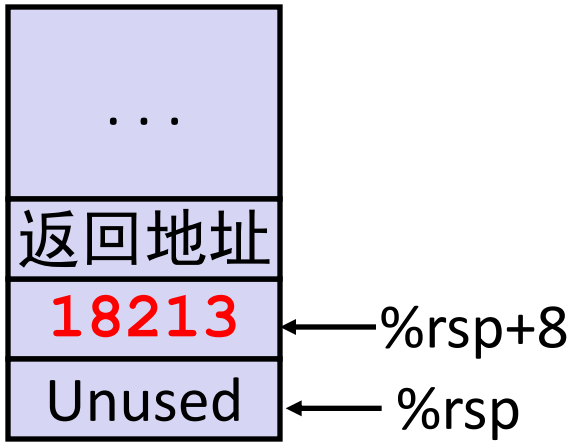
寄存器	用途
%rdi	&v1
%rsi	3000

实例: 调用incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈结构



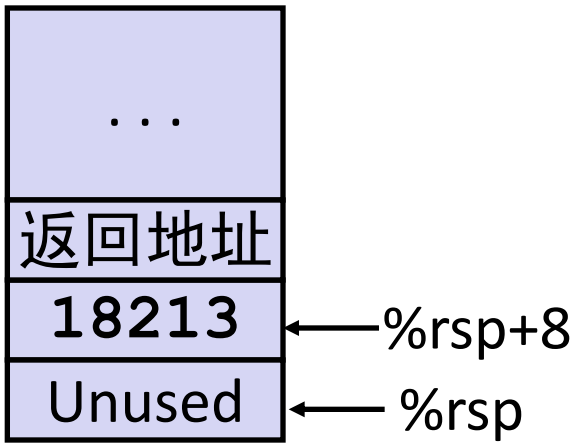
寄存器	用途
%rdi	&v1
%rsi	3000

实例: 调用incr #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

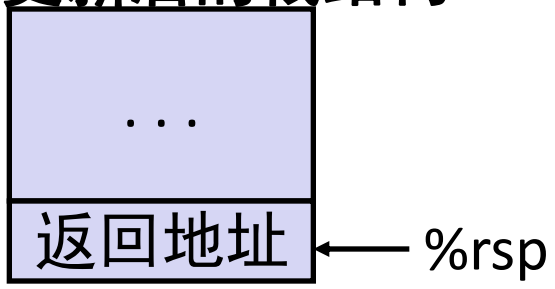
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈结构



寄存器	用途
%rax	返回值

更新后的栈结构

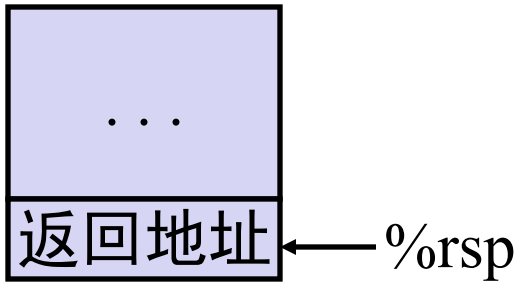


实例: 调用incr #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

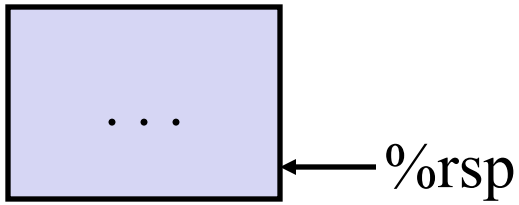
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

更新后的栈结构



寄存器	用途
%rax	返回值

最终栈结构



理解一个问题

```
int * call_p(){
        int a = 0;
        int *p = &a;
        return p;
}
```

这个函数返回的指针是非法的，为什么?

当过程调用时，为变量a在栈帧上分配了空间，返回时栈帧已经被释放了，指针p所指向的位置此时是未知的，修改p指向的数据可能会发生灾难性的后果

寄存器保存约定

- **当过程 yoo调用who时:**
  - yoo是调用者(caller)
  - who是被调用者(callee)
- **寄存器能否用于临时存储?**

```
yoo:
    …
    movq $15213, %rdx
    call who
    addq %rdx, %rax
      …
    ret
```

```
who:
      …
      subq $18213, %rdx
      …
    ret
```

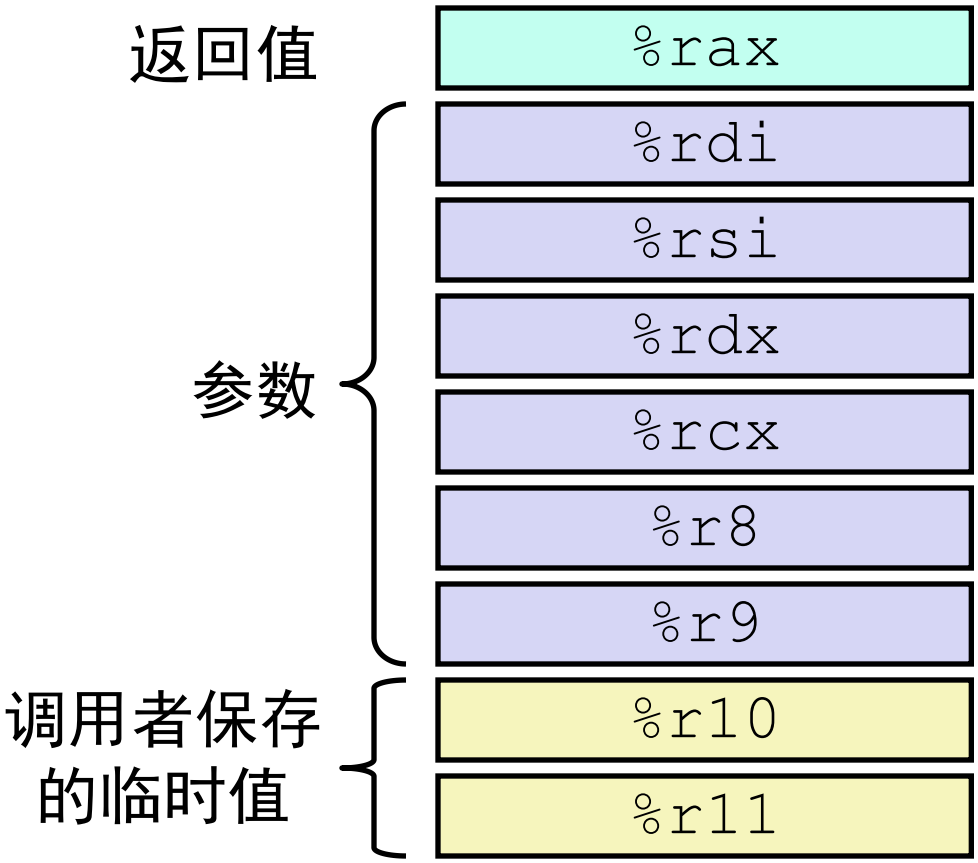
  - 寄存器%rdx的内容被 who覆盖写了
  - 这样会有问题，如何解决?
    - 需要调用者(caller)和被调用者(callee)之间的协调

寄存器保存约定

- **当过程 yoo调用who时:**
  - yoo是调用者(caller)
  - who是被调用者(callee)
- **寄存器能否用于临时存储?**
- **约定——谁来保存的问题**
  - 调用者保存“Caller Saved”
    - 调用者在调用前，在它的栈帧中保存临时值(寄存器)
  - 被调用者保存“Callee Saved”
    - 被调用者要先在自己的栈帧中保存，然后再使用(寄存器)
    - 返回到调用者之前，恢复这些保存的值

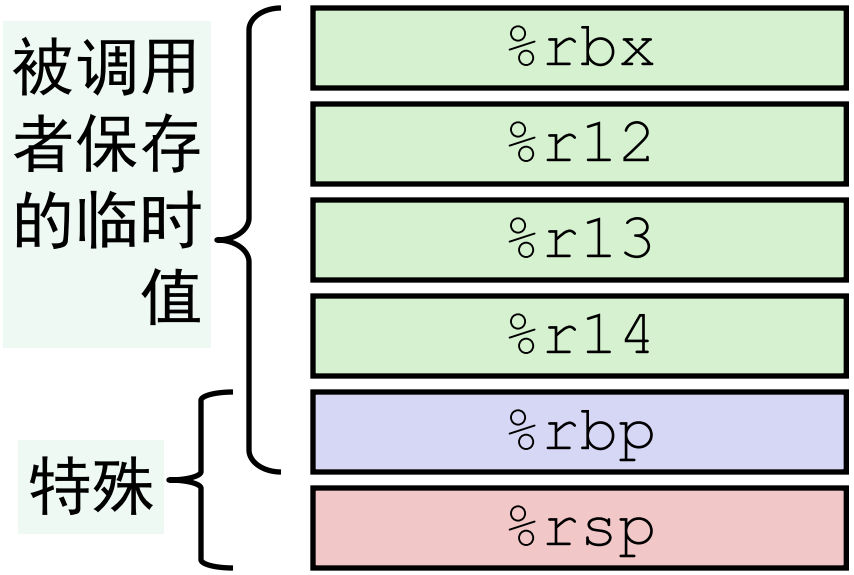
x86-64 Linux的寄存器用法#1

- **%rax**
  - 返回值
  - 调用者保存
  - 被调用过程可修改
- **%rdi, …, %r9**
  - 传递函数参数
  - 调用者保存
  - 被调用过程可修改
- **%r10, %r11**
  - 调用者保存
  - 被调用过程可修改



x86-64 Linux的寄存器用法#2

- **%rbx, %r12, %r13, %r14**
  - 被调用者保存并恢复
- **%rbp**
  - 被调用者保存并恢复
  - 可用作栈帧指针
- **%rsp**
  - 被调用者保存的特殊形式
  - 在离开过程时，恢复为原始值 (CALL之前的值)

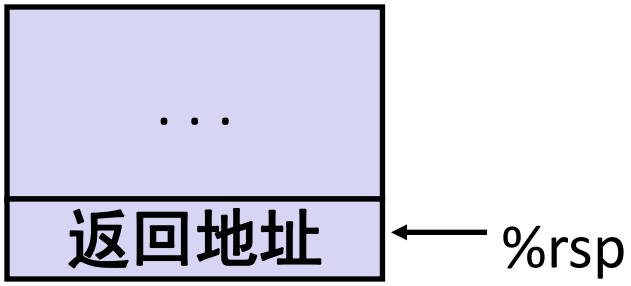


被调用者保存——实例#1

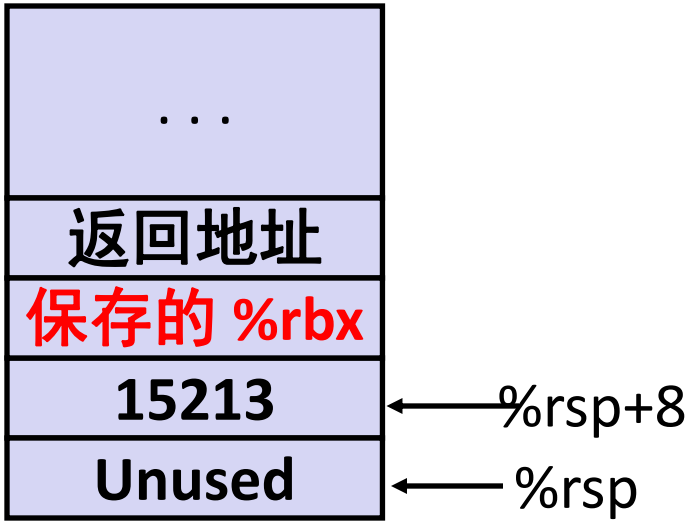
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx          #被调用者保存
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

初始化栈结构



结果栈结构

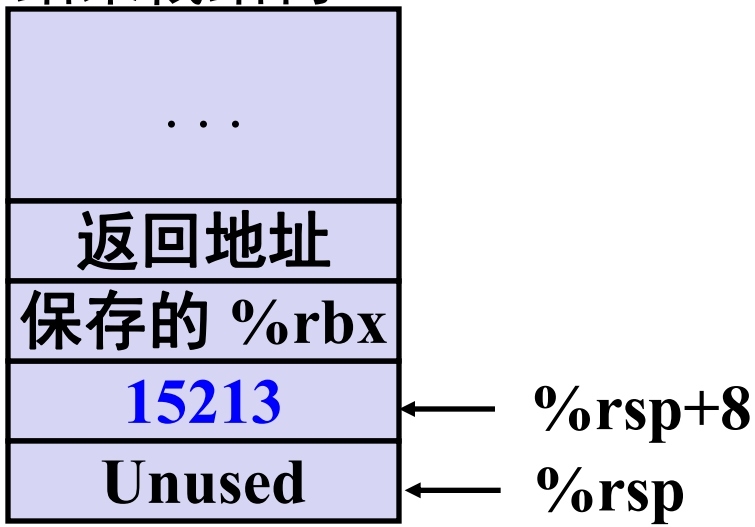


被调用者保存——实例#2

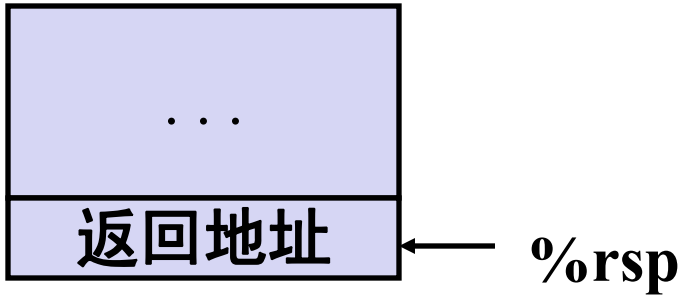
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx        #被调用者保存
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

结果栈结构



返回前的栈结构



主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - 管理局部数据
  - 递归

递归函数

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的终止条件

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rdi	x	参数
%rax	返回值	返回值

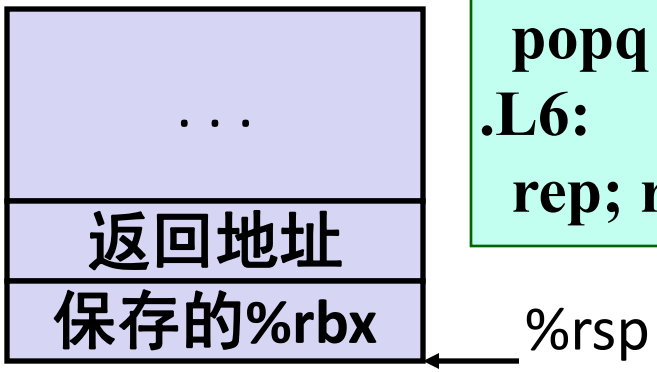
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的寄存器保存

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rdi	x	参数

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



递归函数的调用创建

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rdi	x >> 1	Rec. 参数
%rbx	x & 1	Callee-saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数调用

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rbx	x & 1	被调用者保存
%rax	递归调用的返回值	

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的结果

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)+ pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rbx	x & 1	被调用者保存
%rax	返回值	

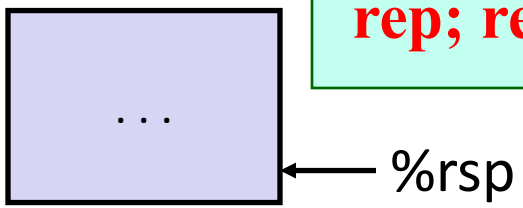
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的完成

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

寄存器	用途	类型
%rax	返回值	返回值

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



递归的观察

- **递归无需特殊的处理**
  - 栈帧意味着每个过程调用在栈上有私有的存储
    - 保存的寄存器 、局部变量
    - 保存的返回地址
  - 寄存器保存约定：防止过程调用损毁其他函数(调用)的数据
    - 除非C代码明确地这样做（如第9课中的缓冲区溢出）。
  - 栈的使用原则：遵循 调用/返回 模式
    - 如 P调用Q, 然后Q 在P结束之前返回
    - 后入、先出
- **对互递归同样有效**
  - P调用Q; Q调用P

经典例题

1.递归函数程序执行时，正确的是（ ）

A．使用了堆 B.可能发生栈溢出 C.容易有漏洞 D.必须用循环计数器

2.x86-64系统中，函数int sum (int x, int y)经编译后其返回值保存在（ ）

A.%rdi B.%rsi C.%rax D.%rdx

经典例题

3.有下列C函数:

```
long arith(long x, long y, long z)
{
  long t1 = _____ (1) ___;
  long t2 = _____ (2) ___;
  long t3 = _____ (3) ___;
  long t4 = _____ (4) ___;
  _____ (5) ___;
}
```

函数arith的汇编代码如下:

```
arith:
xorq   %rsi,%rdi
leaq   (%rdi,%rdi,4),%rax
leaq   (%rax,%rsi,2),%rax
subq   %rdx,%rax
retq
```

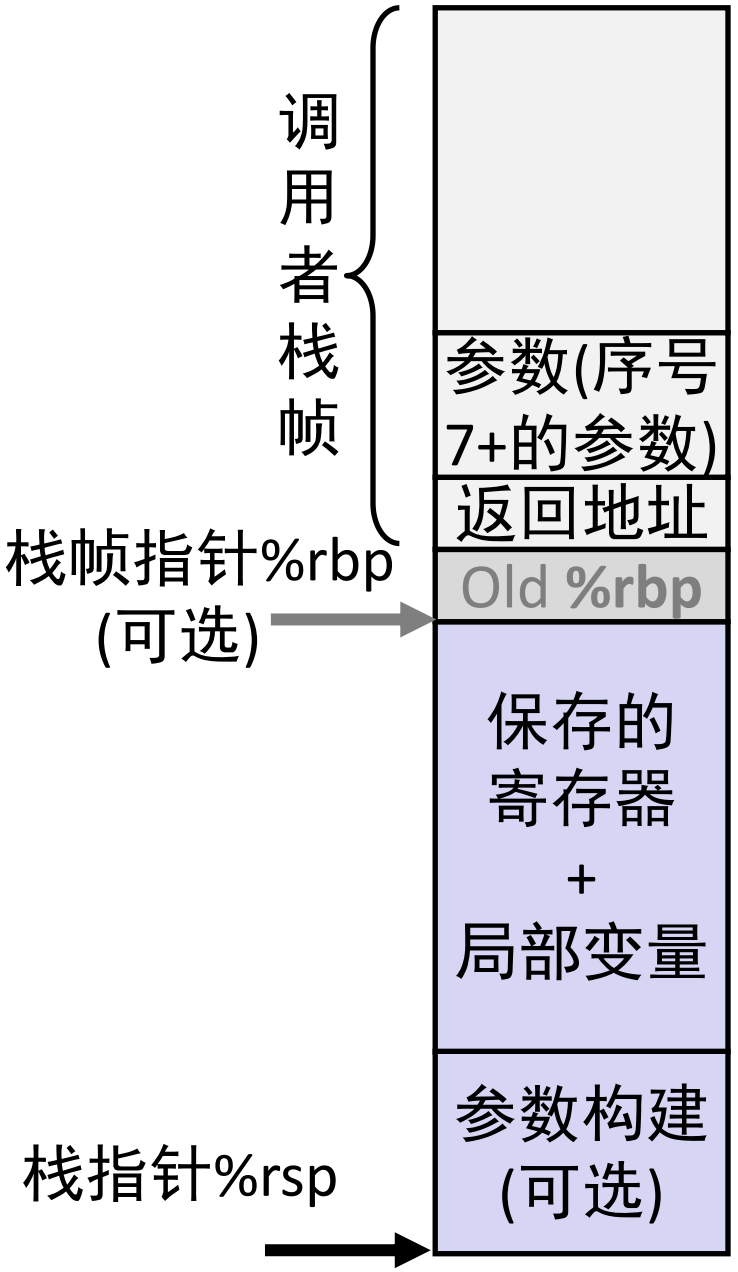
请填写出上述C语言代码中缺失的部分:

(1) x^y (2) t1+ t1 << 2 或5*t1 (3) t2 + (y << 1) (4) t3-z

(5) return t4

x86-64 过程总结

- **要点**
  - 栈是实现过程调用/返回所依赖的数据结构
  - 如P调用Q，则Q先返回P后返回
- **用正常调用约定处理递归(互递归)**
  - 可安全保存数值的地方：栈帧、被调用者保存的寄存器
  - 函数调用前将参数置于栈顶
  - 返回结果在 `%rax`中
- 指针就是数值的地址:
  - 在栈中的或是全局的



Enjoy!