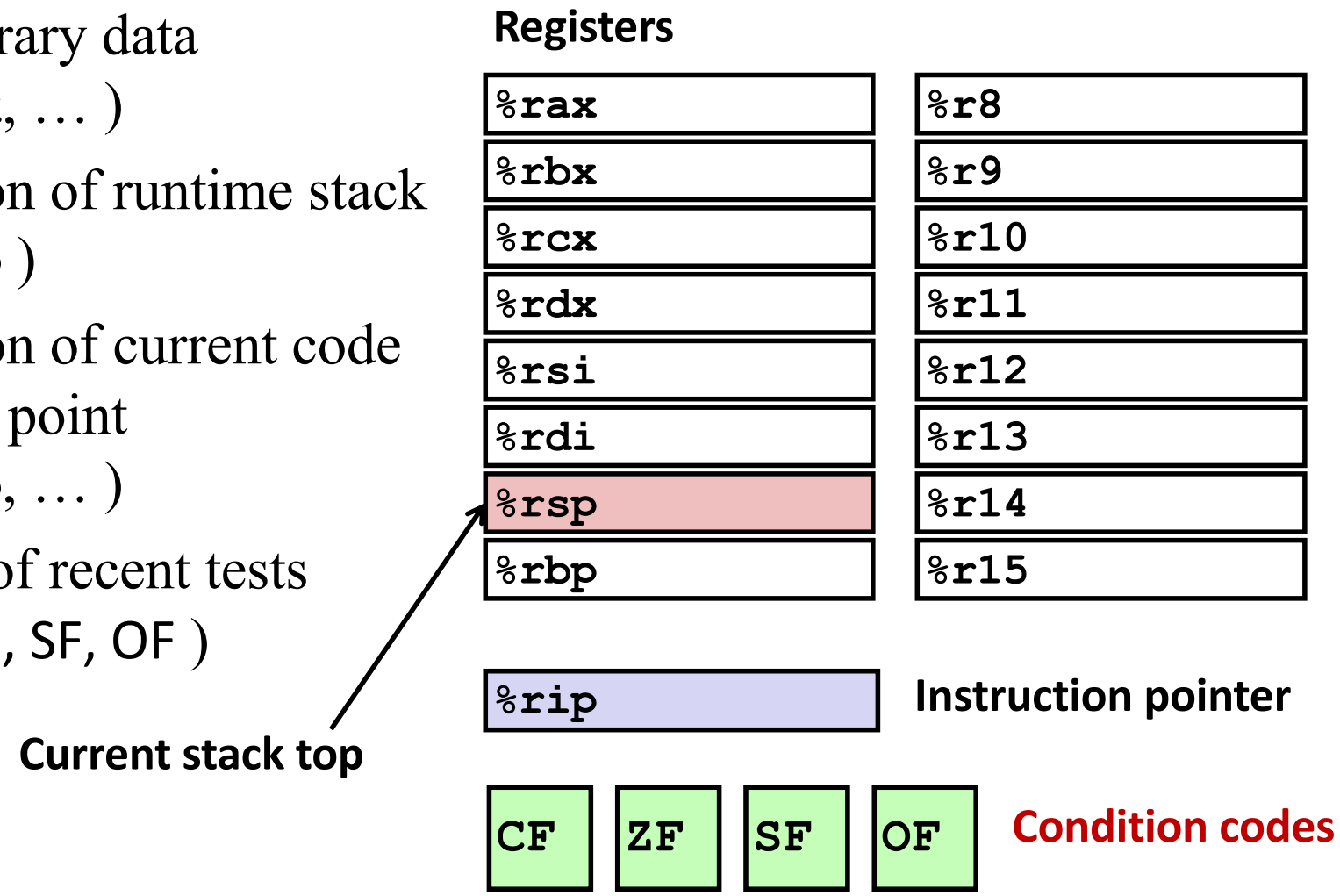
程序的机器级表示

- 分以下五个部分介绍
  - 第一讲： x86-64指令系统
    - 高级语言程序转换为机器代码的过程
    - 机器指令和汇编指令
    - x86-64指令系统
  - 第二讲： C语言程序的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
    - 过程调用的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：内存越界引用和缓冲区溢出
  - 第五讲：浮点指令和代码

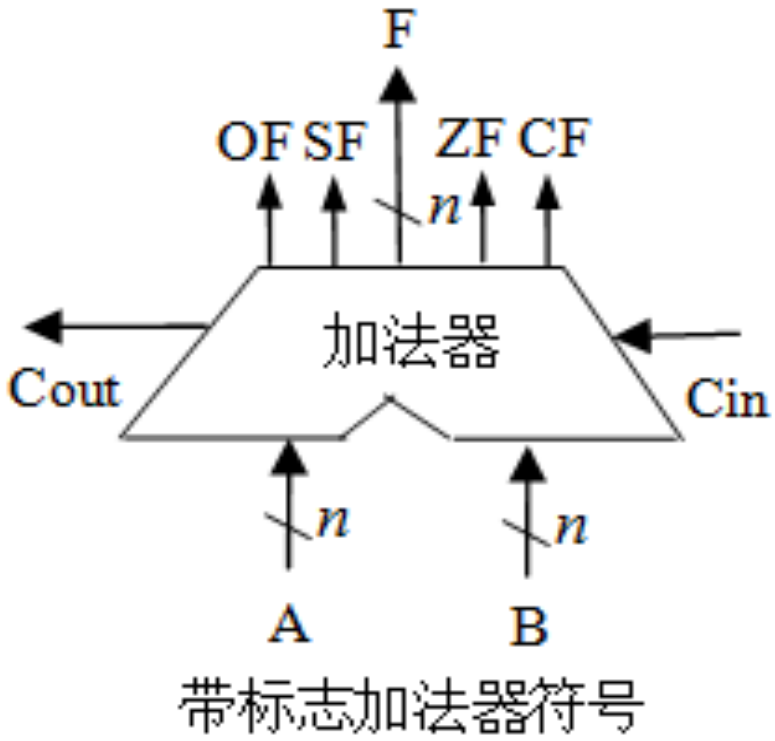
围绕C语言中的语句，解释其在底层机器级的实现方法

x86-64处理器寄存器（部分）

- 正在执行程序的相关信息
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (CF, ZF, SF, OF)



条件码



带标志加法器符号

- 零标志ZF、溢出标志OF、进/借位标志CF、符号标志SF称为条件标志。
- 条件标志（Flag）在运算电路中产生，被记录到专门的寄存器中
- 存放标志的寄存器通常称为程序/状态字寄存器或标志寄存器或条件码寄存器。每个标志对应寄存器中的一个标志位。

条件码

比较cmp指令

做减法操作，不改变目标操作数，仅影响标志: CF OF ZF SF

包括: cmpb、cmpw、cmpl，cmpq

cmpq Src2, Src1

Src1-Src2,不改变**Src1**的值

测试test指令

做“与”操作测试，不改变目标操作数，仅影响标志: SF ZF

CF=0 OF=0

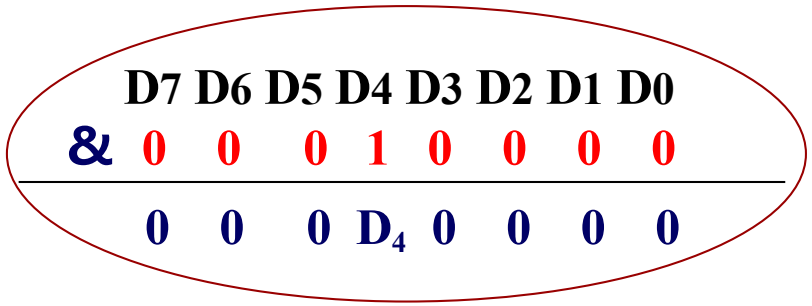
包括: testb、testw、testl，testq

testq Src2, Src1

Src1&Src2 不改变**Src1**的值

例：检测AL中的D4是否为1

testb $0x10,%al



SF=0
ZF=0 if D4=1
ZF=1 if D4=0

TEST 通常用于检测操作数某些位是1还是0，该指令后通常带有条件转移指令。

访问条件码

条件码通常不会直接读取，访问条件码三种方法：

（1）可以根据条件码的某种组合，将一个字节设置为0或者1。

（2）可以条件跳转到程序的某个其他的部分。

（3）可以有条件地传送数据。

访问条件码

- SetCC 指令
  - SET 指令根据条件码设置目的操作数低位单字节寄存器或者内存操作数

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setae	~CF	Above or Equal (unsigned)
setb	CF	Below (unsigned)
setbe	CF|ZF	Below or Equal (unsigned)

ge
~(SF^OF)
=1 iff.
SF=ZF=0
　无溢出正
SF=ZF=1
　负溢出 实际正

有符号数

无符号数

访问条件码


```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x-y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

- **movzbl** 使目标寄存器的高位清0
- 32 位操作数清零高 32 位规则

跳转指令

- 无条件跳转指令

  jmp DST：无条件转移到目标指令DST处执行

  （1）直接跳转

  jmp 标号

  （2）间接跳转

  例如： jmp *%rdx　　jmp *(%rdx)

- 条件跳转指令

  JCC DST：CC为条件码，根据标志（条件码）判断是否满足条件，若满足，则转移到目标指令DST处执行，否则按顺序执行

分三类：

(1)根据单个标志的值转移

(2)按无符号整数比较转移

(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A>B
10	jae/jnb label	CF=0	无符号整数 A≥B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A<B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A≤B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A>B
14	jge/jnl label	SF=OF	带符号整数 A≥B
15	jl/jnge label	SF≠OF	带符号整数 A<B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A≤B

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。

跳转指令的编码

```
1       movq    %rdi, %rax
2       jmp     .L2
3     .L3:
4       sarq    %rax
5     .L2:
6       testq   %rax, %rax
7       jg      .L3
8       rep; ret
```

汇编器产生的“.o”格式的反汇编版本如下：

```
1       0:   48 89 f8                mov    %rdi,%rax
2       3:   eb 03                   jmp    8 <loop+0x8>
3       5:   48 d1 f8                sar    %rax
4       8:   48 85 c0                test   %rax,%rax
5       b:   7f f8                   jg     5 <loop+0x5>
6       d:   f3 c3                   repz retq
```

汇编器产生的“.o”格式的反汇编版本如下：

```
1      0:   48 89 f8                mov    %rdi,%rax
2      3:   eb 03                   jmp    8 <loop+0x8>
3      5:   48 d1 f8                sar    %rax
4      8:   48 85 c0                test   %rax,%rax
5      b:   7f f8                   jg     5 <loop+0x5>
6      d:   f3 c3                   repz retq
```

```
1      4004d0:   48 89 f8                mov    %rdi,%rax
2      4004d3:   eb 03                   jmp    4004d8 <loop+0x8>
3      4004d5:   48 d1 f8                sar    %rax
4      4004d8:   48 85 c0                test   %rax,%rax
5      4004db:   7f f8                   jg     4004d5 <loop+0x5>
6      4004dd:   f3 c3                   repz retq
```

例子：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
   int  i，  sum = 0;
   for (i = 0; i <= len–1; i++)
         sum += a[i];
   return sum;
}
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。 Why?

i 和 len-1 分别存放在哪个寄存器中？ %eax？ %ecx？

```
L3: ...

 ...
leal    -0x1(%rsi),%ecx
cmpl    %eax,%ecx
jae     .L3
 ...
```

i 在%eax中，len-1 在%ecx中
%eax: 0000 0000
%ecx: 1111 1111
cmpl 指令的执行结果是什么?

jae .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0	无符号数A≥B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A≤B
JG/JNLE label	SF=OF AND ZF=0	有符号数A > B
JGE/JNL label	SF=OF	有符号数A≥B
JL/JNGE label	SF≠OF	有符号数A < B
JLE/JNG label	SF≠OF OR ZF=1	有符号数A≤B

显然，对于每个 i 都满足条件，因为任何无符号数都比32个1小，因此循环体被不断执行，最终导致数组访问越界而发生存储器访问异常。

例子：程序的机器级表示与执行

例:

```
int sum(int a[ ], int len)
{
   int  i，sum = 0;
   for (i = 0; i <= len–1; i++)
        sum += a[i];
   return sum;
}
```

正确的做法是将参数len声明为int型。 Why?

```
L3: …

 …
leal     -0x1(%rsi),%ecx
cmpl     %eax,%ecx
jge      .L3
 …
```

i 在%eax中，len-1 在%ecx中

%eax: 0000 …… 0000

%ecx: 1111 …… 1111

cmpl 指令的执行结果是什么?

条件控制

gcc –Og –S control.c

```
long absdiff
(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # y, x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

goto 代码表示

- C 允许 **goto**表达式
  - 跳转到标签位置

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

```
if (test-expr)
  then-statement
else
  else-statement
```

```
t=test-expr;
if (!t) goto false;
then-statement
goto done;
false:
else-statement
done:
```

```
t=test-expr;
if (t) goto true;
else-statement
goto done;
true:
then-statement
done:
```

条件控制


```
long absdiff
(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # y,x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

```
absdiff:
    cmpq    %rsi, %rdi   # y,x
    jg      .L4
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
.L4:                     # x > y
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
```

```
void  cond(long a, long* p)
{
    if (p&&a>*p)
        *p=a;
}
```

```
cond:
testq %rsi, %rsi
je .L1
cmpq %rdi, (%rsi)
jge  .L1
movq %rdi, (%rsi)
.L1:
rep; ret
```

条件传送

- 条件传送:

  if (Test) Dest ← Src

- 条件传送指令更符合现代处理器的性能特性

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

条件传送


```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff
(long x, long y)
{
    long rval = x-y;
    long eval = y-x;
    long ntest = x <= y;
    if (ntest) rval=eval;
    return rval;
}

...
{ return x<=y?y-x:x-y; }
```

```
absdiff:
    movq    %rdi, %rax   # x
    subq    %rsi, %rax   # return x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx   # eval = y-x
    cmpq    %rsi, %rdi   # x-y
    cmovle  %rdx, %rax   # if <=, result = eval
    ret
```

指令	同义名	传送条件	描述
cmove S, R	cmovz	ZF	相等/零
cmovne S, R	cmovnz	~ZF	不相等/非零
cmovs S, R		SF	负数
cmovns S, R		~SF	非负数
cmovg S, R	cmovnle	~(SF ^ OF) & ~ZF	大于（有符号>）
cmovge S, R	cmovnl	~(SF ^ OF)	大于或等于（有符号>=）
cmovl S, R	cmovnge	SF ^ OF	小于（有符号<）
cmovle S, R	cmovng	(SF ^ OF) | ZF	小于或等于（有符号<=）
cmova S, R	cmovnbe	~CF & ~ZF	超过（无符号>）
cmovae S, R	cmovnb	~CF	超过或相等（无符号>=）
cmovb S, R	cmovnae	CF	低于（无符号<）
cmovbe S, R	cmovna	CF | ZF	低于或相等（无符号<=）

“Do-While” 循环

C Code

```
do
   Body
while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- Body:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

“Do-While” 循环

C Code

```
long pcount_do
(unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
(unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- 计算x中的1的个数

“Do-While” 循环的汇编代码

Goto Version

```
long pcount_goto
(unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   rep; ret
```

“While” 循环#1

While version

```
while (Test)
   Body
```

Goto Version

```
goto test;
loop:
   Body
test:
   if (Test)
      goto loop;
done:
```

“While” 循环#1

C Code

```
long pcount_while
(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- 和do-while 做比较

“While” 循环#2

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

- “Do-while” conversion
- Used with **-O1**

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

“While” 循环#2

C Code

```
long pcount_while
(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

```
long loop_while(long a , long b)
{
  long result=__1__;
  while (__a<b__) {
    result=__result*(a+b)__;
    a=__a+1__;
  }
  return result;
}
```

```
loop_while:
movl $1, %eax
jmp .L2
.L3:
leaq (%rdi, %rsi), %rdx
imulq %rdx, %rax
addq $1, %rdi
.L2:
cmpq %rsi, %rdi
jl .L3
rep; ret
```

“For” 循环

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(long)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

“For” 循环 → While循环

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

“For” 循环

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
(unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

“For” 循环

C Code

```
long pcount_for
(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Goto Version

```
long pcount_for_goto_dw
(unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {
    unsigned bit =
       (x >> i) & 0x1;            Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

```
long pcount_for
(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
    movl    $0x0,%edx
    movl    $0x0,%ecx
    jmp     L1
L2: movq    %rdi,%rax
    shrq    %cl,%rax
    andl    $0x1,%eax
    addq    %rax,%rdx
    addq    $0x1,%rcx
L1: cmpq    $63,%rcx
    jle     L2
    movq    %rdx,%rax
    ret
```

switch 语句

switch(开关)语句可以根据一个整数索引值进行多重分支。

- 提高了C 代码的可读性
- 通过使用跳转表(jump table)使得实现更加高效。

```
long switch_eg
(long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
       /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

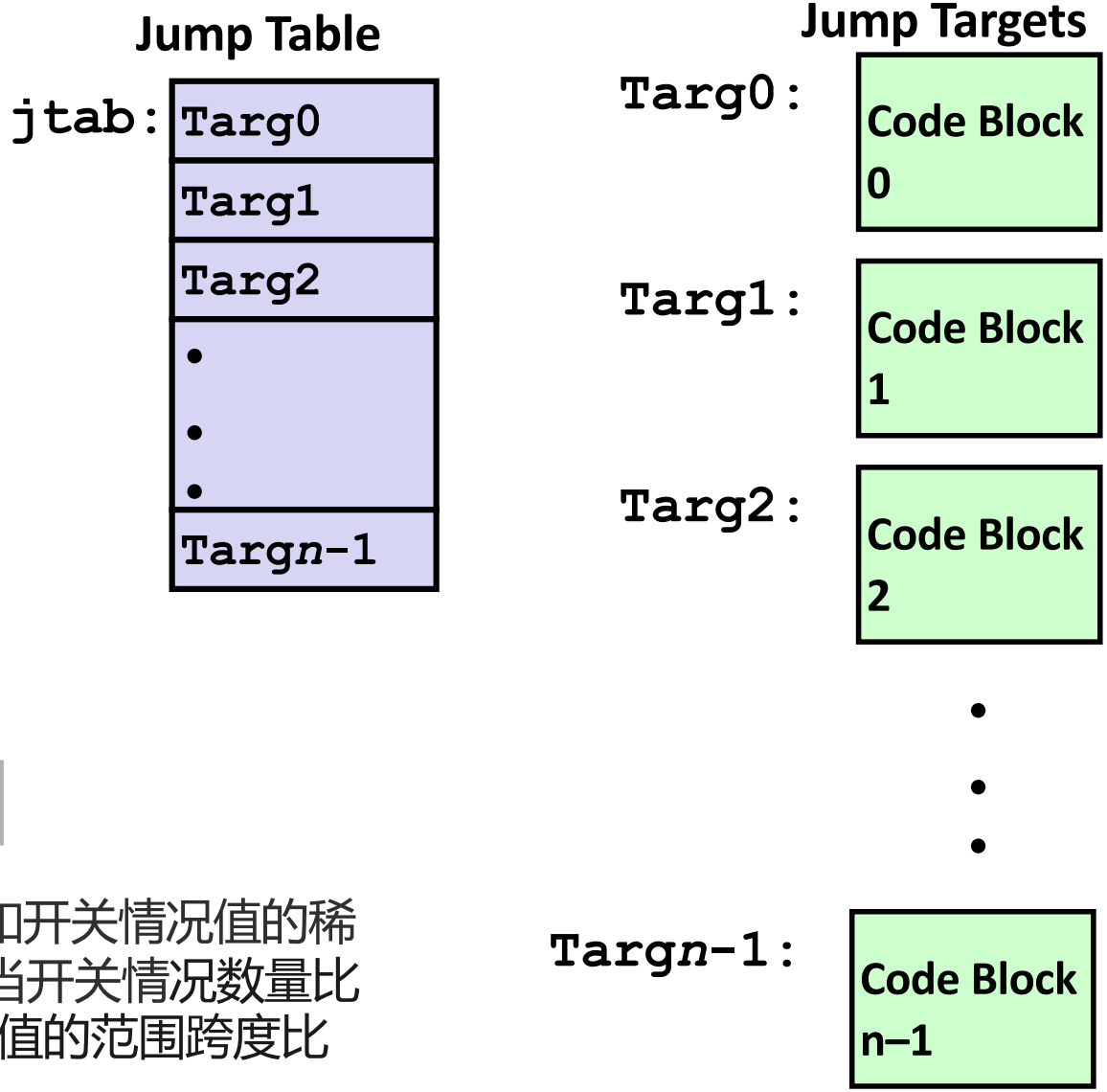
跳转表结构

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
  ...
  case val_n-1:
    Block n–1
}
```

Translation (Extended C)

```
goto *JTab[x];
```



GCC 根据开关情况的数量和开关情况值的稀疏程度来翻译开关语句。当开关情况数量比较多(例如4个以上)，并且值的范围跨度比较小时，就会使用跳转表。

Switch 语句


```
long switch_eg
(long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
       /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- Multiple case labels
  - 5 & 6
- Fall through cases
  - 2
- Default cases
  - 4, ...

Switch 语句


```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Switch 语句


```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

间接跳转

Assembly Setup Explanation

- 跳转表结构
- 每个跳转地址： 8 bytes
  - 基地址： **.L4**
- 跳转
  - 直接: **jmp .L8**
  - 间接: **jmp *.L4(,%rdi,8)**
    - 跳转表开始: .L4
    - %rdi*8 (地址 8 bytes)
    - 根据有效地址 .L4 + x*8 获取相应的跳转地址
    - Only for 0 ≤ x ≤ 6

跳转表

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

跳转表

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

Code Blocks (x == 1)


```
switch(x) {
case 1:           // .L3
    w = y*z;
    break;
...
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

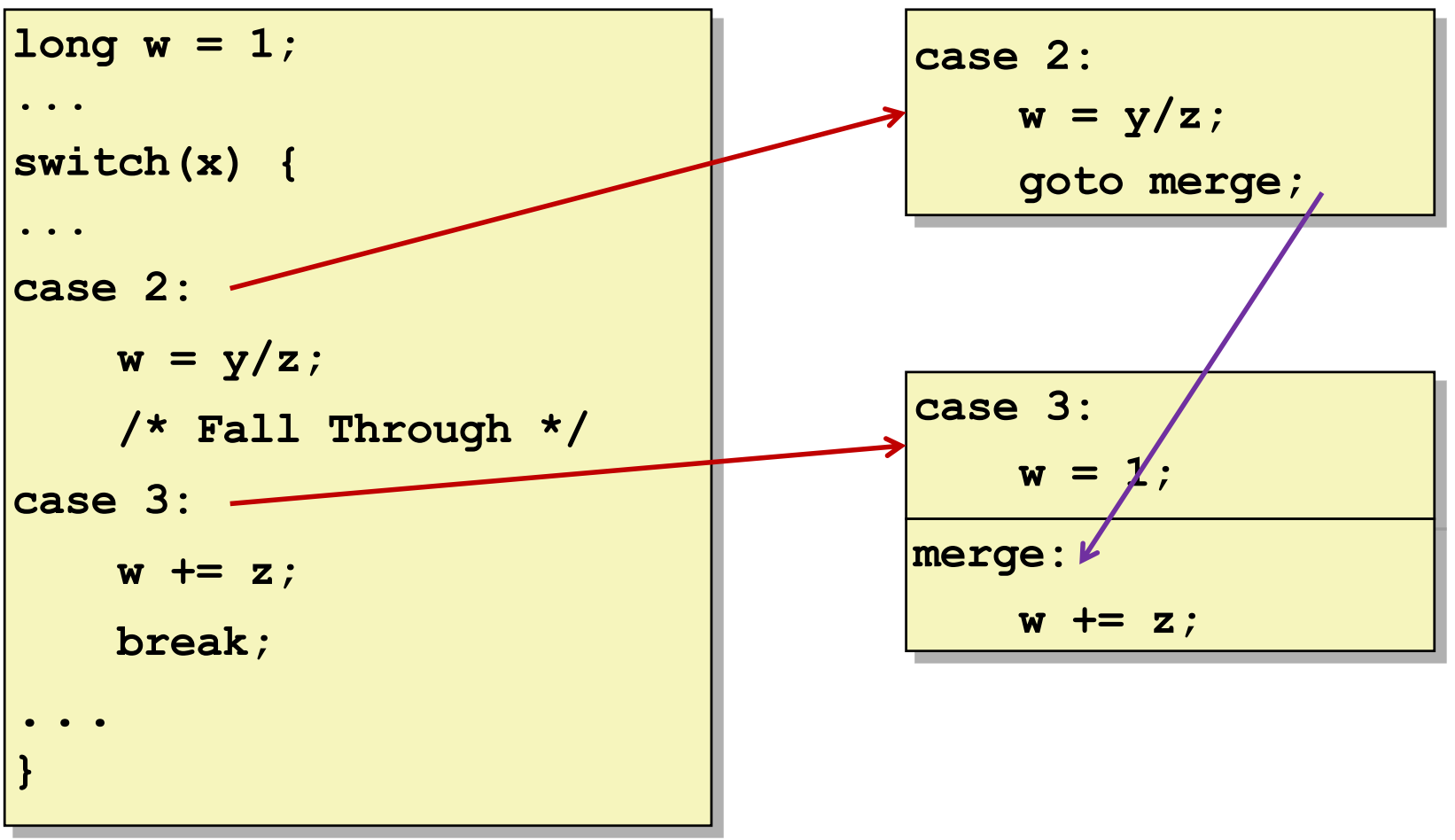
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through


```
long w = 1;
...
switch(x) {
...
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
...
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```



Code Blocks (x == 2, x == 3)


```
long w = 1;
...
switch(x) {
...
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
...
}
```

```
.L5:                        # Case 2
   movq    %rsi, %rax
   movq    %rdx, %rcx
   idivq   %rcx            # y/z
   jmp     .L6             # goto merge
.L9:                        # Case 3
   movl    $1, %eax        # w = 1
.L6:                        # merge:
   addq    %rcx, %rax      # w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks (x == 5, x == 6, default)


```
switch(x) {
...
  case 5:  // .L7
  case 6:  // .L7
    w -= z;
    break;
  default: // .L8
    w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

```
void switcher
(long a , long b , long c , long *dest) {
  long val;
  switch(a) {
  case ____4____:
    c= ___b^15___;
  case____1____:
  case ____6____ :
    val=_____(c+b)<<2_____;
    break;
  case ____3____:
    val= _____a_____;
    break;
  default:
    val=_____b_____;
  }
  *dest= val;
}
```

```
switcher:
cmpq $6 , %rdi
ja .L2
jmp *.L4( ,%rdi ,8)
.L7:
xorq $15 , %rsi
movq %rsi , %rdx
.L5:
leaq (%rdx , %rsi) , %rdi
salq $2 , %rdi
jmp . L6
.L2:
movq %rsi , %rdi
.L6:
movq %rdi , (%rcx)
ret
```

```
.L4:
.quad .L2
.quad .L5
.quad .L2
.quad .L6
.quad .L7
.quad .L2
.quad .L5
```

过程调用的机器级表示

- 传递控制

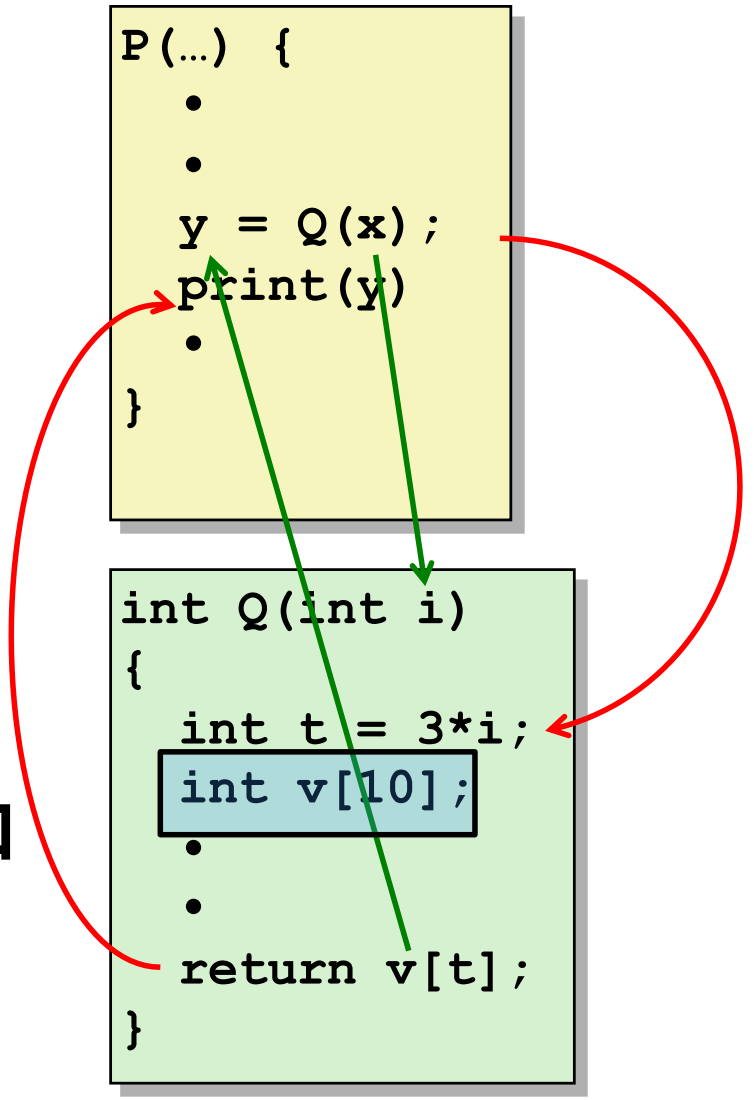
  过程的调用和返回

- 传递数据

  参数的传递和返回值

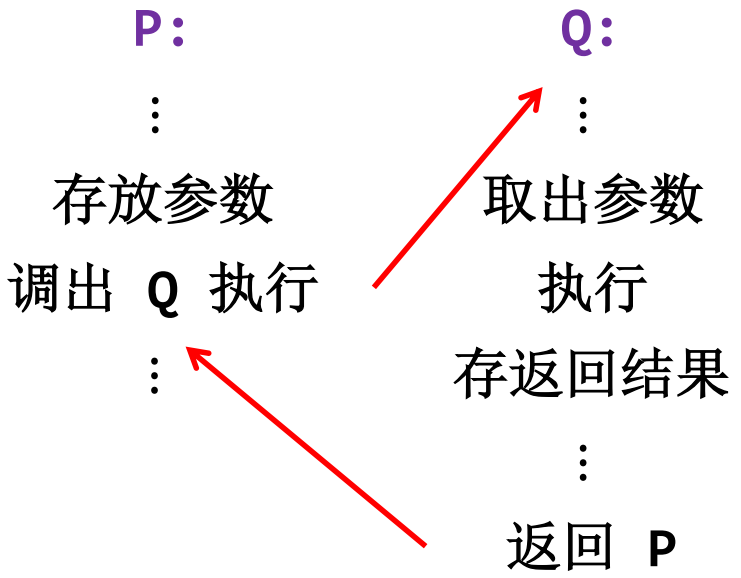
- 内存的分配和释放

  为局部变量分配空间及释放



x86-64 的过程实现包括过程调用和返回指令和一些对机器资源(例如寄存器和程序内存)使用的约定规则。

过程调用的机器级表示



何为现场？

通用寄存器的内容。

为何要保存现场？

因为所有过程共享一套通用寄存器。

想象：两个人做菜时共用同一套盘子。

过程调用的执行步骤（P 为调用者，Q 为被调用者）

（1）P 将入口参数（实参）放到 Q 能访问到的地方；

（2）P 保存返回地址，然后将控制转移到 Q；　CALL 指令

（3）Q 保存 P 的现场，并为自己的非静态局部变量分配空间；

（4）执行 Q 的过程体；　处理阶段

（5）Q 恢复 P 的现场，释放局部变量空间；

（6）Q 取出返回地址，将控制转移到 P。　RET 指令

P 过程

准备阶段

结束阶段

Q 过程

Code Examples


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

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

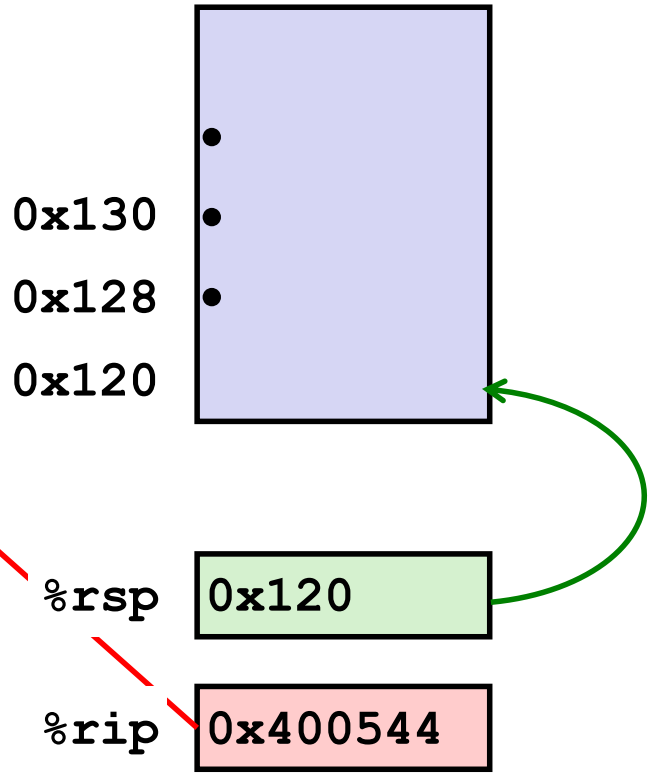
```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: mov    %rdx,%rbx             # Save dest
  400544: callq  400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: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

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

控制流 Example #1


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

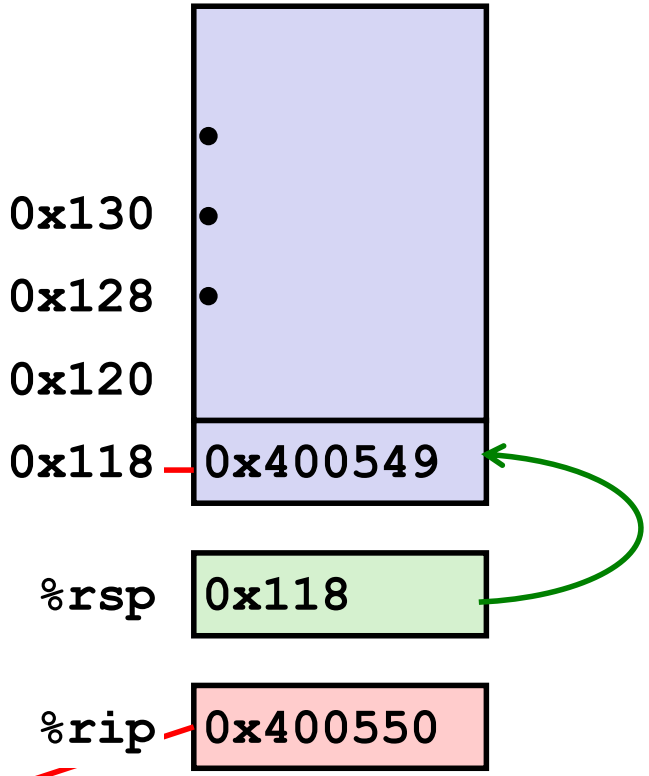
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



控制流 Example #2


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

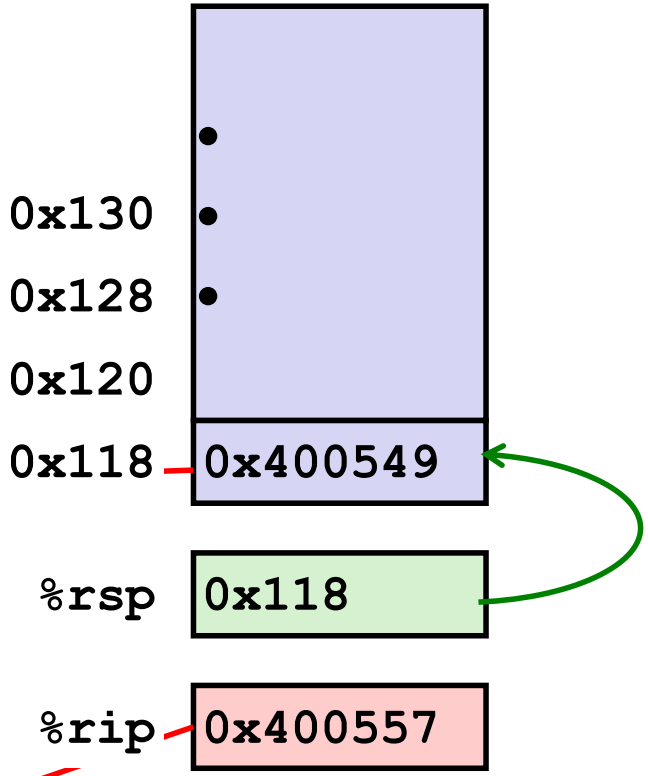
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



控制流 Example #3


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

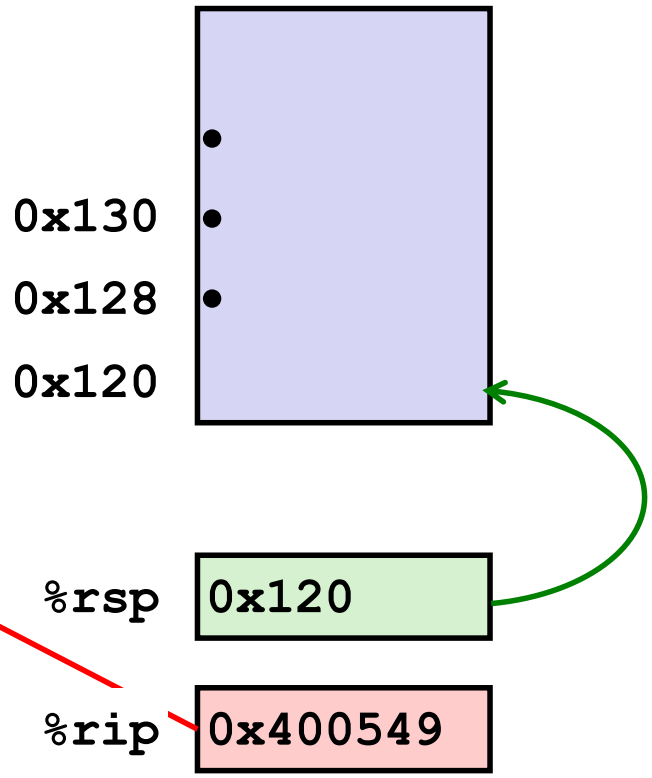
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



控制流 Example #4


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

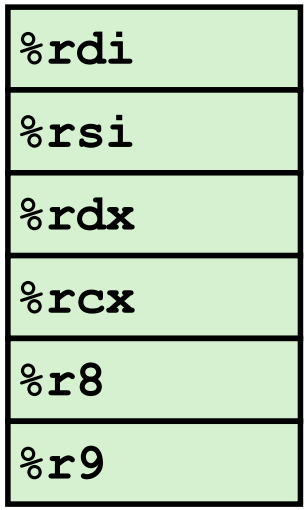
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



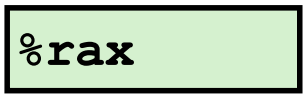
数据传送

寄存器

- 通过寄存器最多传递6个整型(例如整数和指针)参数

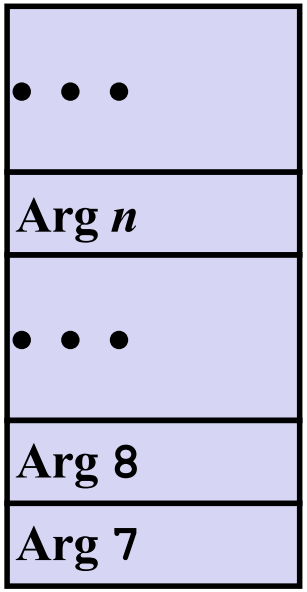


- 返回值



堆栈

- 需要的时候分配堆栈空间



操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

数据传送


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

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

```
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
  ...
```

```
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
```

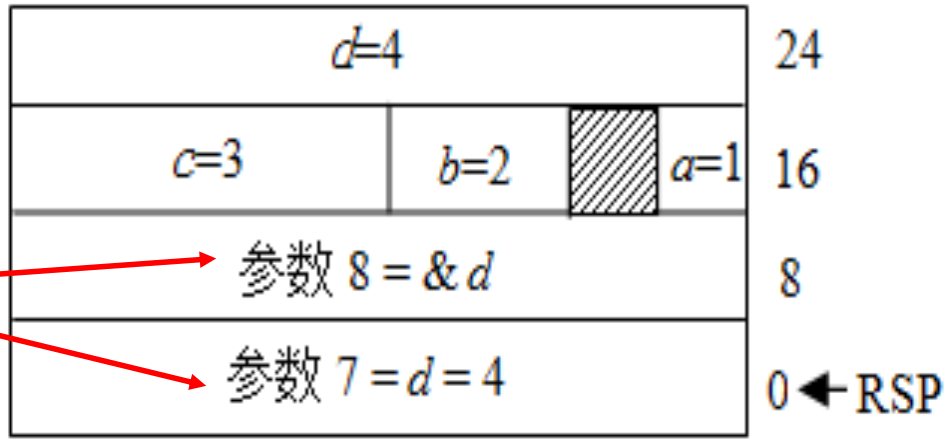
数据传送


```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

其他6个参数在哪里?

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

执行到 caller 的 call 指令时栈中情况



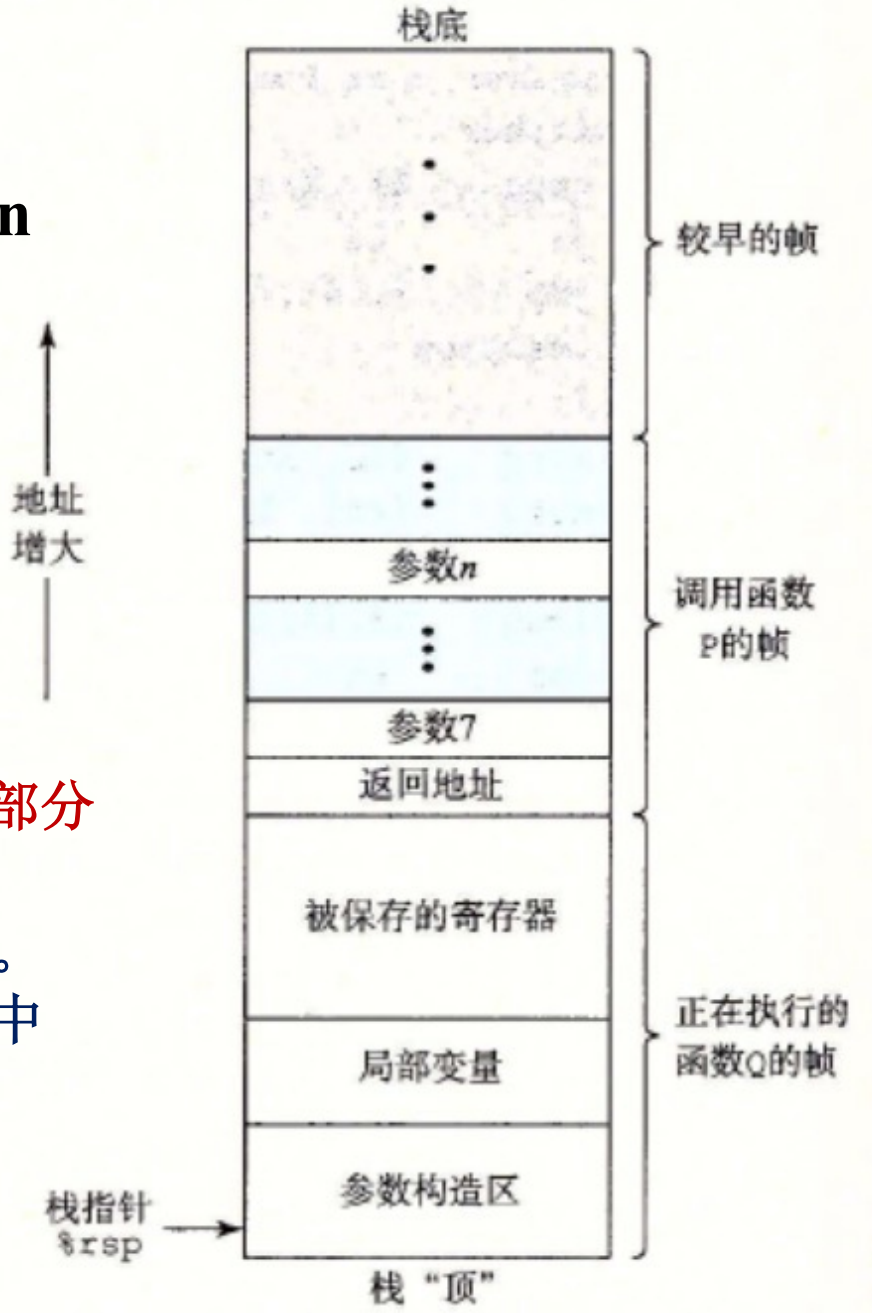
执行到caller的call指令,

栈中的状态如何?

操作数宽度（字节）	入口参数 1	2	3	4	5	6	返回参数
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

x86-64栈帧

- 内容
  - 返回地址
  - 调用另一个过程设置的参数7~n
  - 局部变量
    - (1) 寄存器不足够存放
    - (2) 使用地址运算符 &
    - (3) 数组或结构体
  - 被保存的现场寄存器



X86-64 过程只分配自己所需要的栈帧部分

实际上，许多函数甚至根本不需要栈帧。
若所有的局部变量都可以保存在寄存器中，而且该函数不会调用任何其他函数。

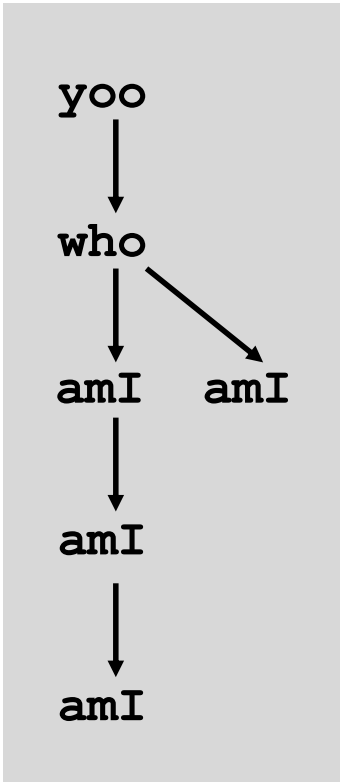
过程调用链举例

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

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

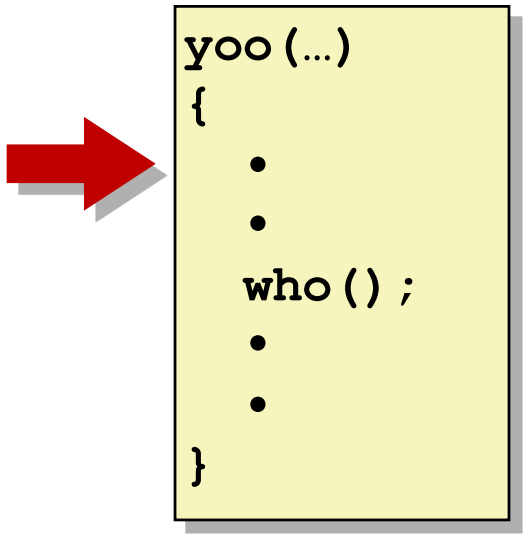
```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

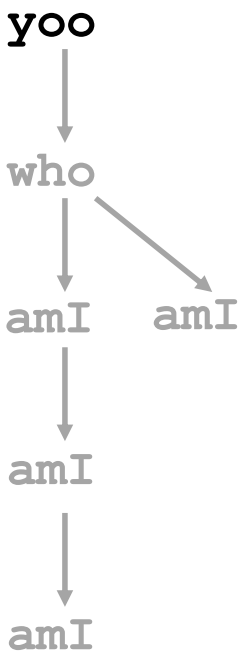
Example Call Chain

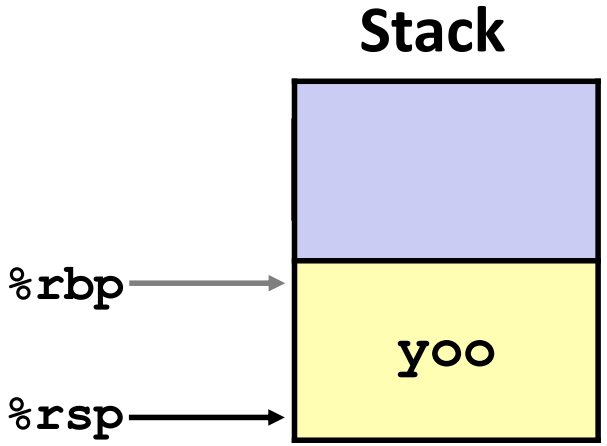


Procedure amI() is recursive

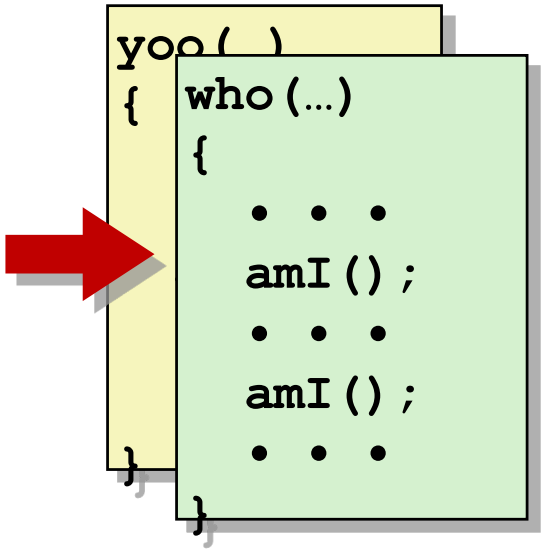
Example

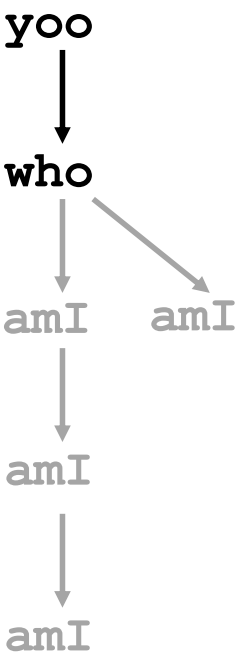


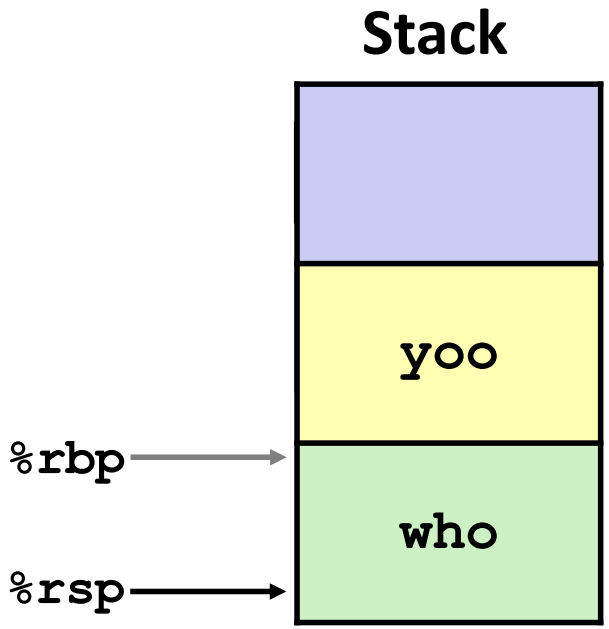




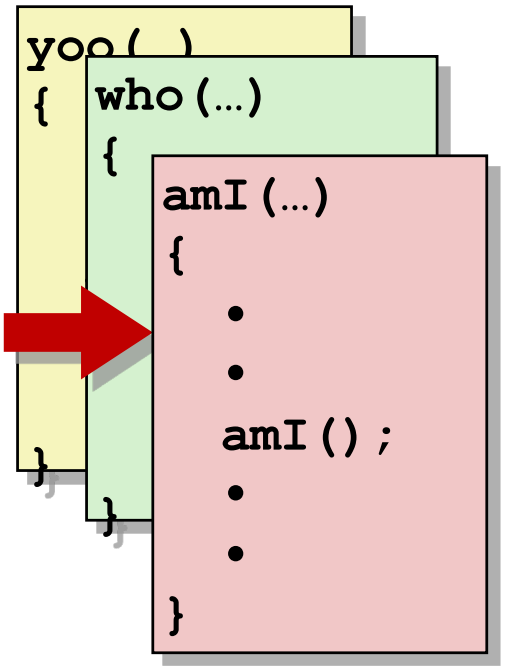
Example

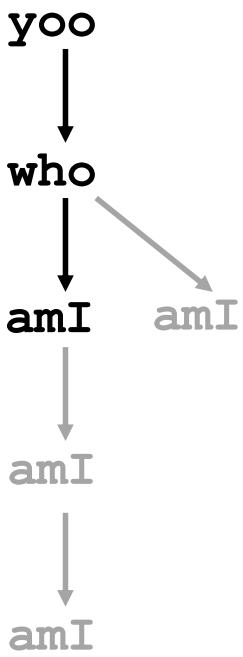


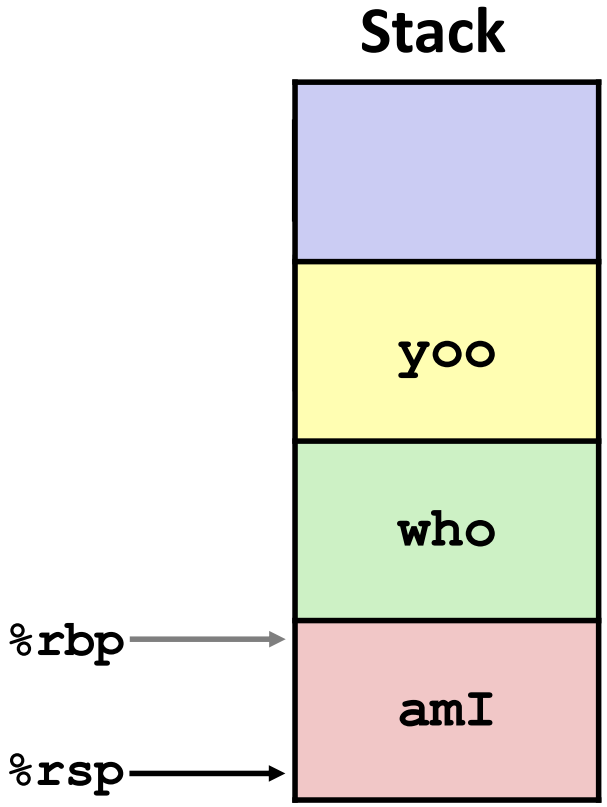




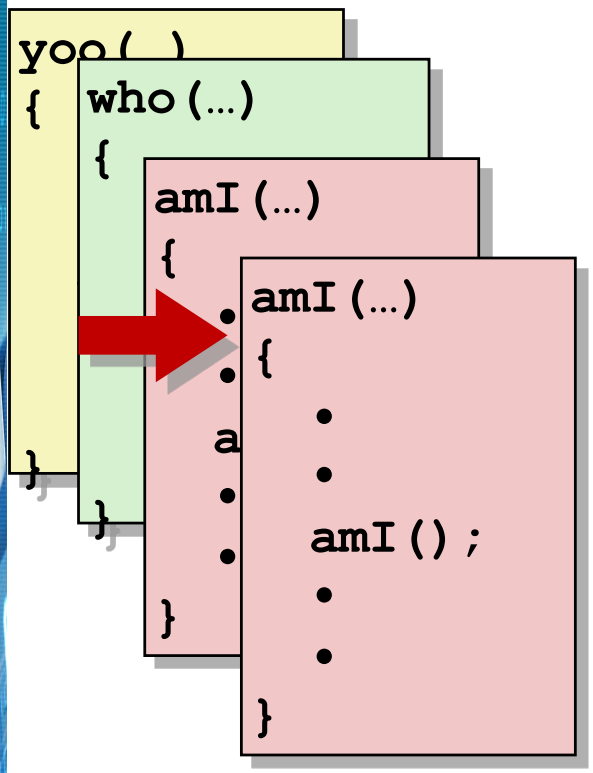
Example

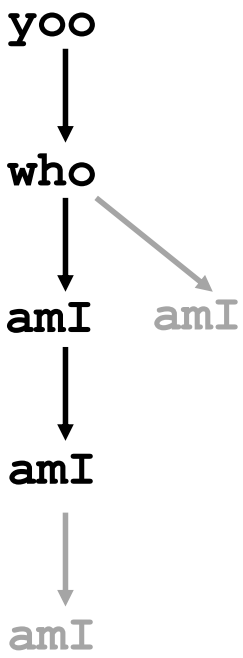

yoo( )
{
who(...)
{
amI(...)
{
•
•
amI();
•
•
}
}
}

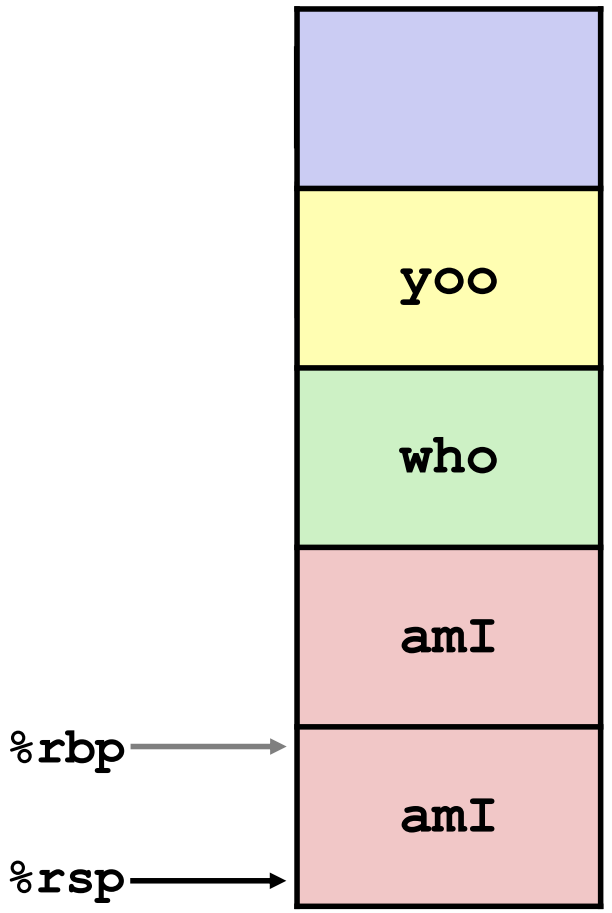

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

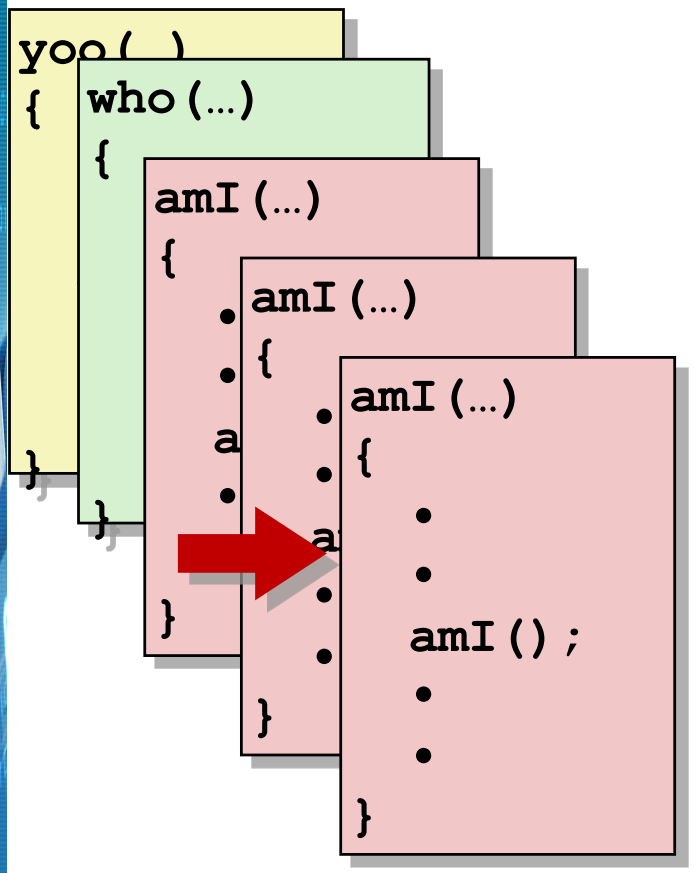
Example

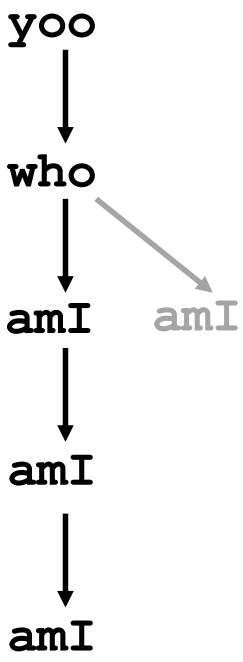

yoo( )
{
who(...)
{
amI(...)
{
amI(...)
{
amI();
}
}
}
}

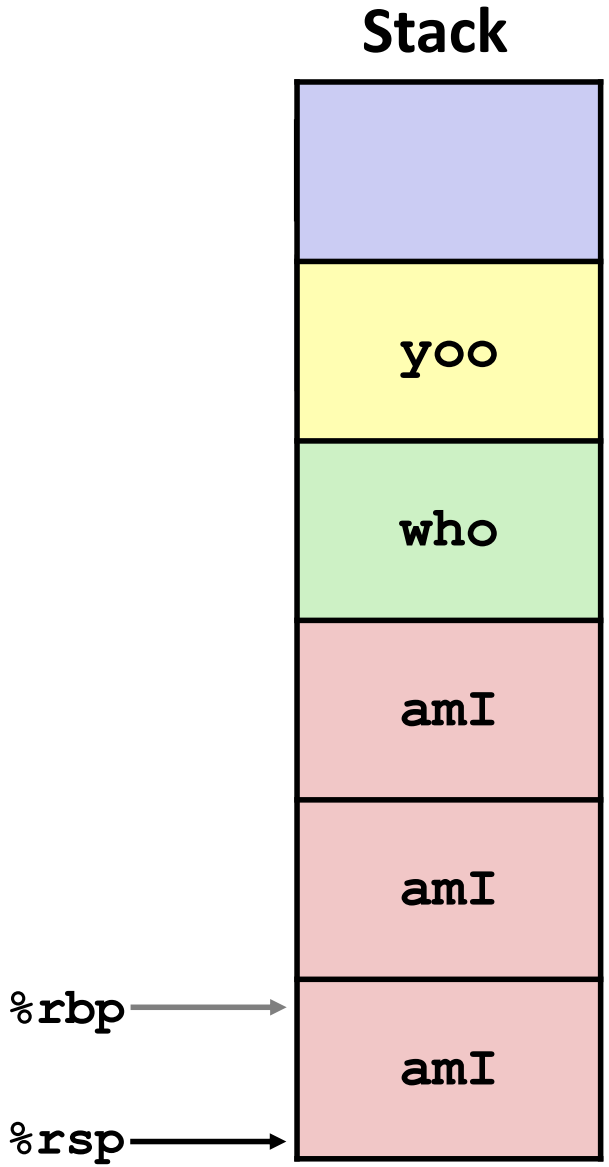

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

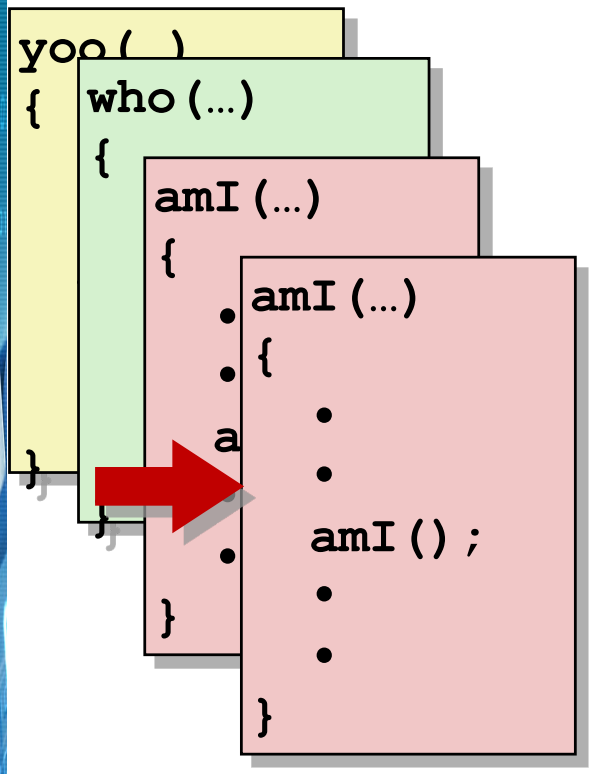
Example

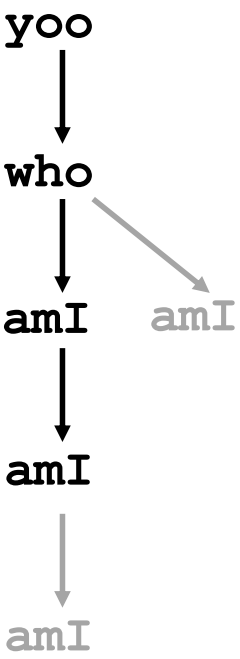

yoo ( )
{
who (...)
{
amI (...)
{
amI (...)
{
amI (...)
{
amI () ;
}

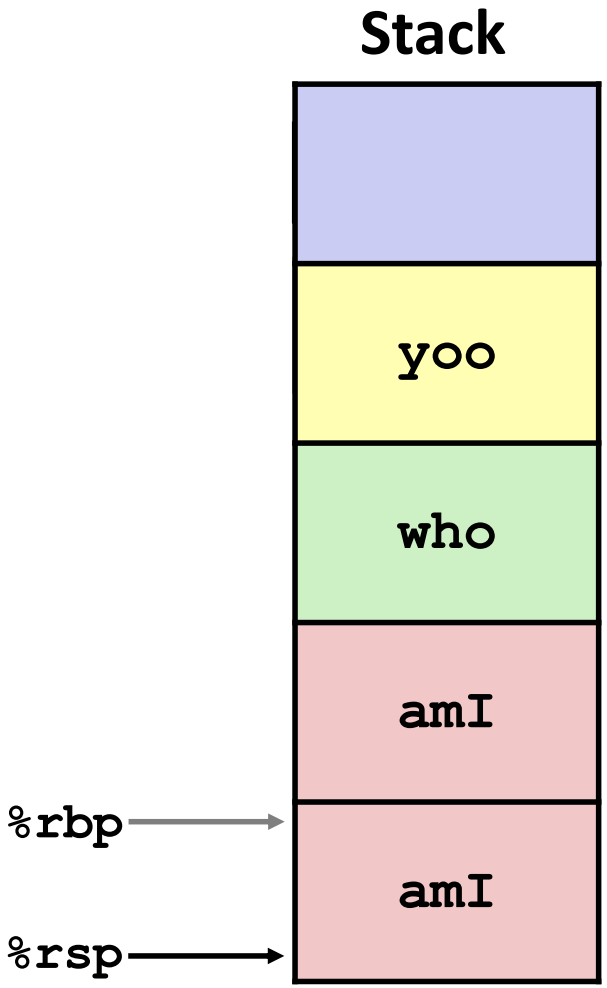

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

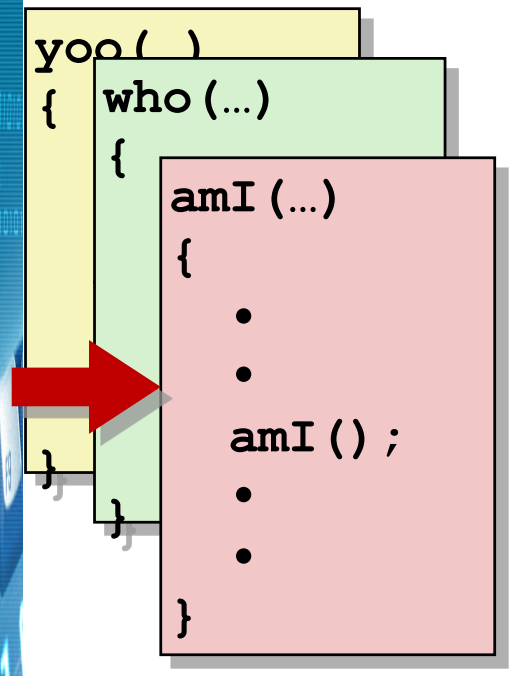
Example

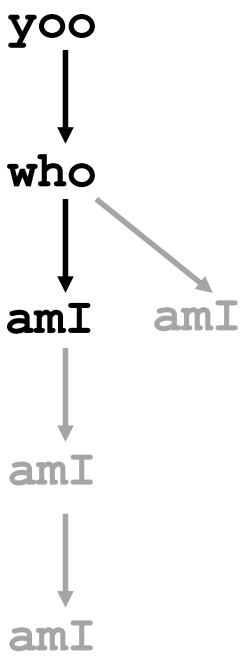

yoo( )
{
who(...)
{
amI(...)
{
amI(...)
{
amI();
}

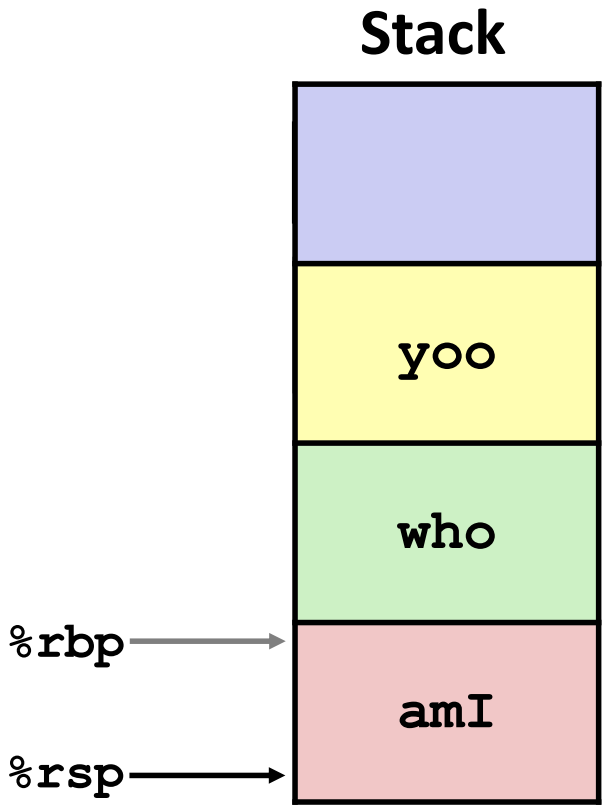

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

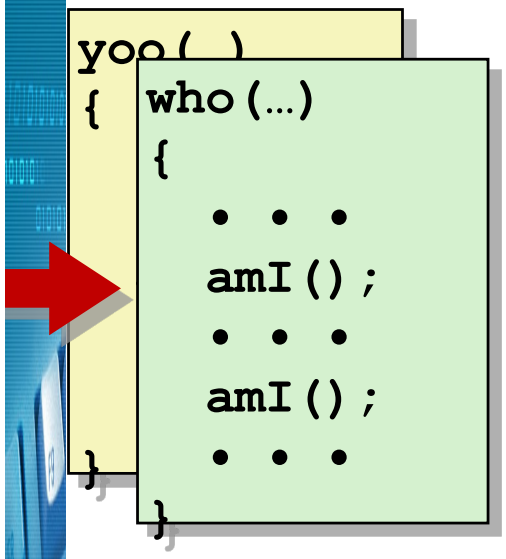
Example

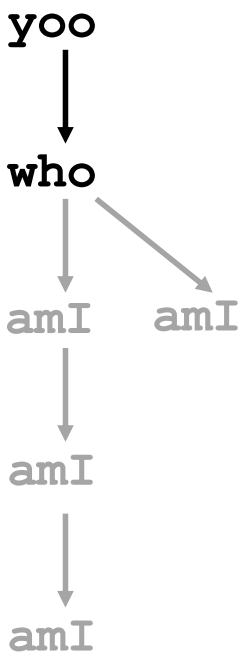

yoo( )
{
who(...)
{
amI(...)
{
•
•
amI();
•
•
}
}
}

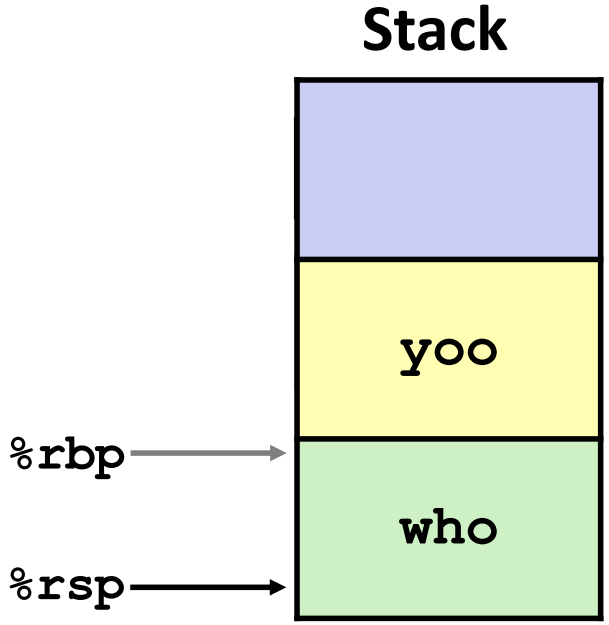

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

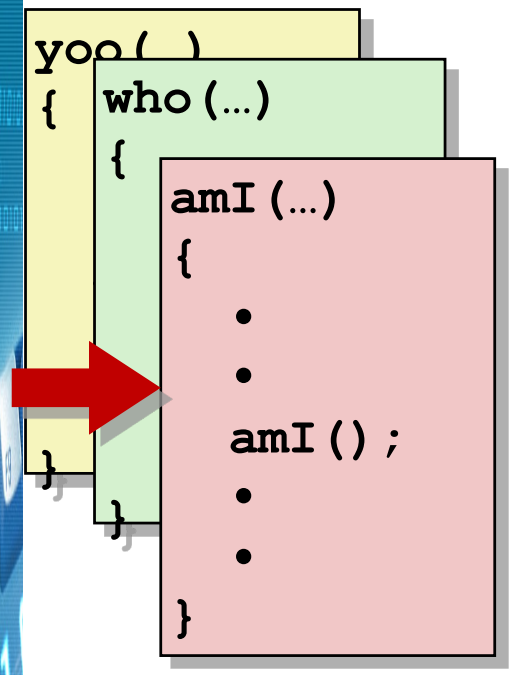
Example

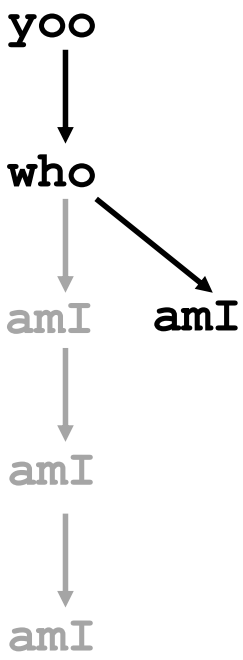


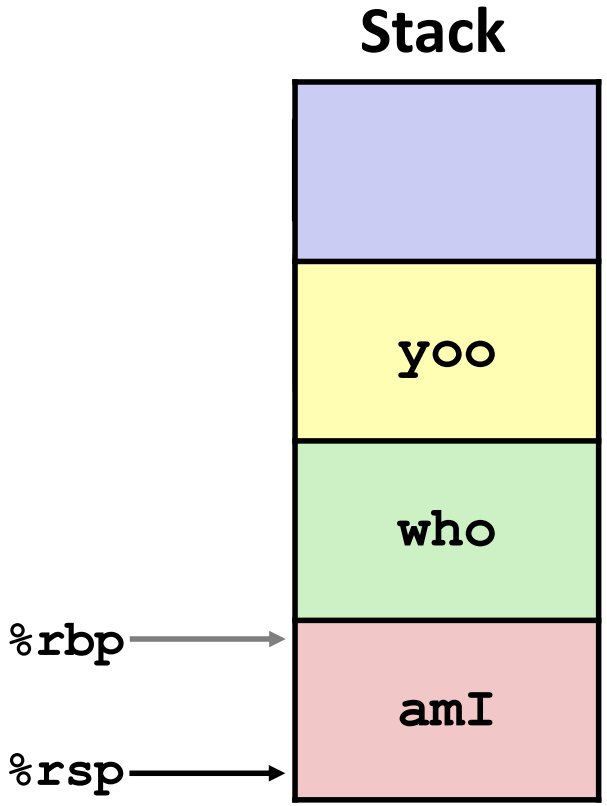




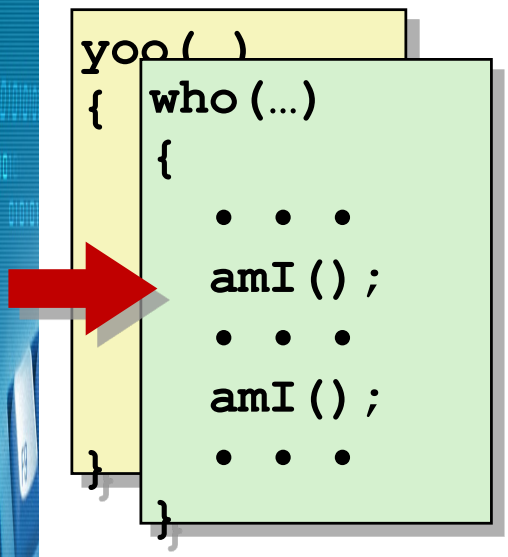
Example

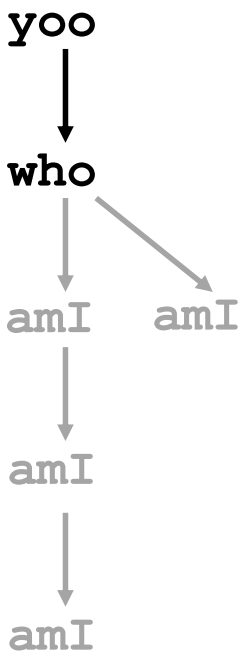

yoo( )
{
who(...)
{
amI(...)
{
•
•
amI();
•
•
}
}
}

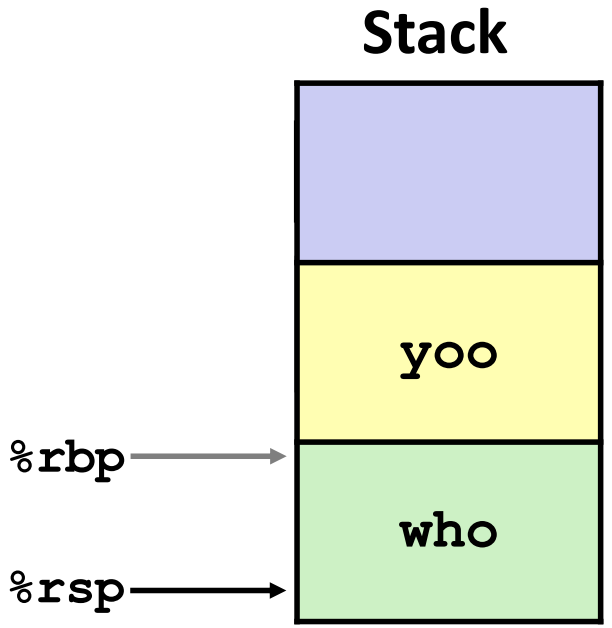

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example

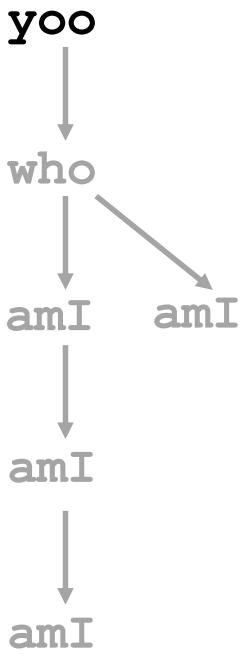


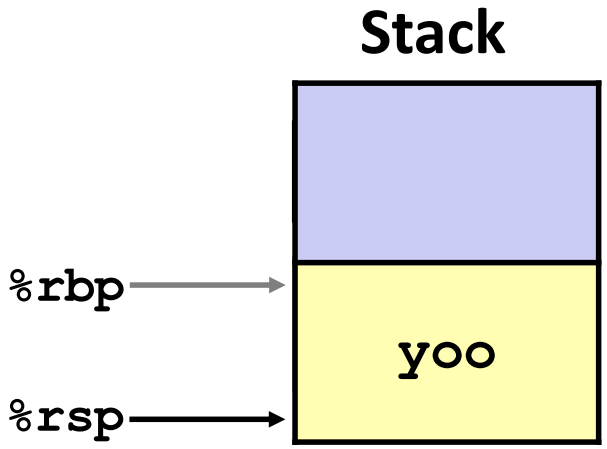




Example

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





Example: `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
```

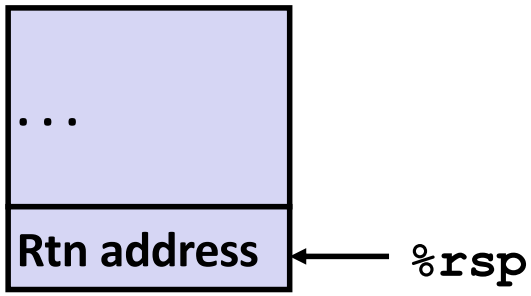
Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	Return value `x`

Example: Calling `incr` #1

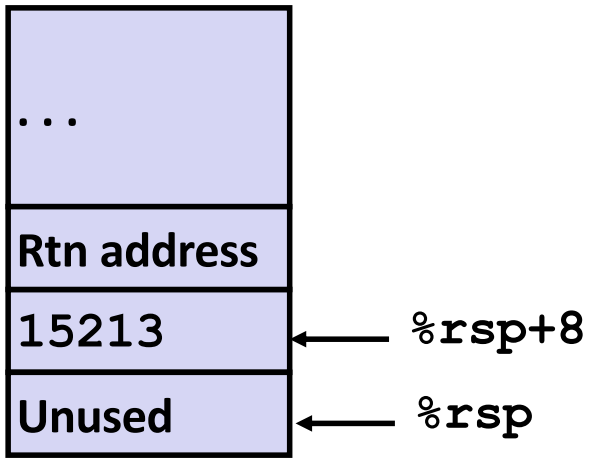

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

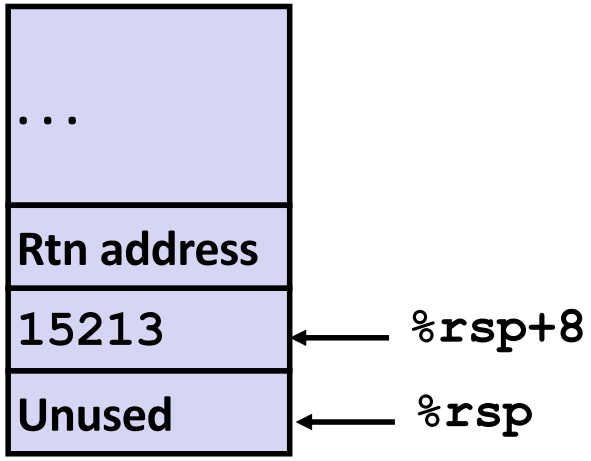


Example: Calling `incr` #2


```
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
```

Stack Structure



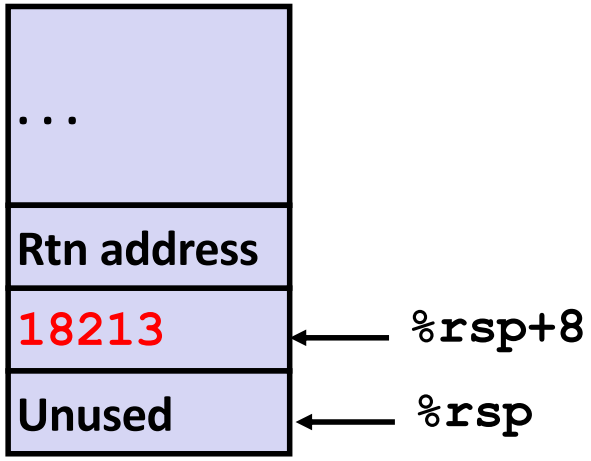
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `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
```

Stack Structure



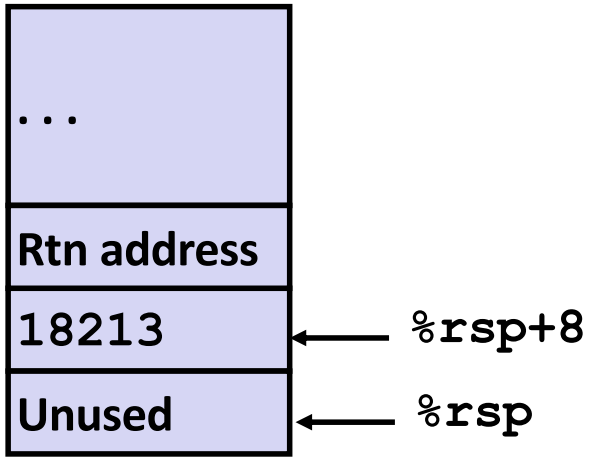
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4


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

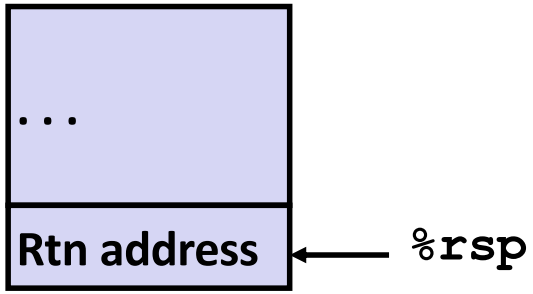
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure

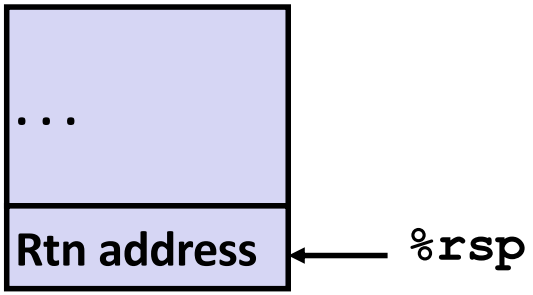


Example: Calling `incr` #5


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

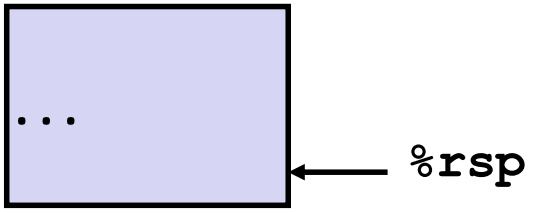
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure

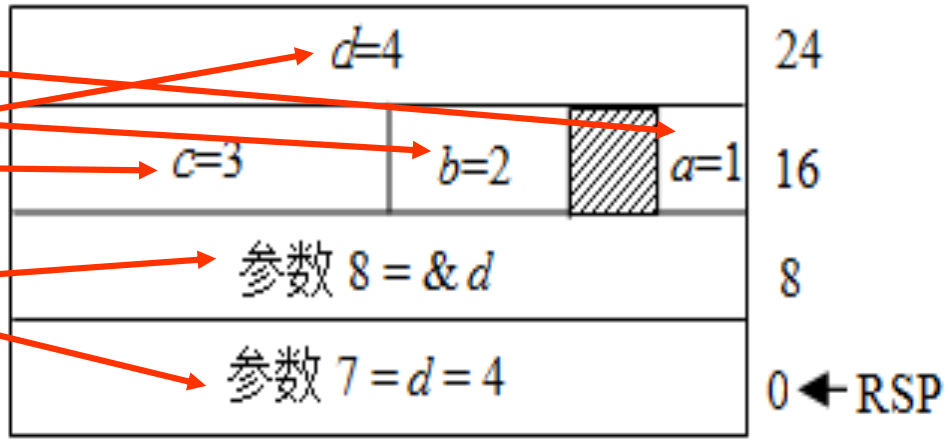


数据传送


```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}

void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
      *ap+=a;
      *bp+=b;
      *cp+=c;
      *dp+=d;
}
```

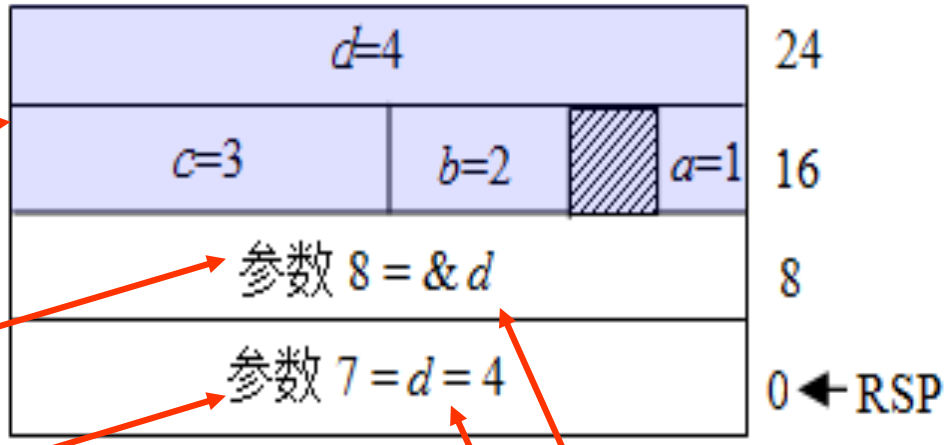
执行到 caller 的 call 指令时栈中情况



数据传送


```
subq $32, %rsp          //R[rsp]←R[rsp]-32
movb $1, 16(%rsp)       //M[R[rsp]+16]←1
movw $2, 18(%rsp)       //M[R[rsp]+18]←2
movl $3, 20(%rsp)       //M[R[rsp]+20]←3
movq $4, 24(%rsp)       //M[R[rsp]+24]←4
leaq 24(%rsp), %rax     //R[rax]←R[rsp]+24
movq %rax, 8(%rsp)      //M[R[rsp]+8]←R[rax]
movq $4, (%rsp)         //M[R[rsp]]←4
leaq 20(%rsp), %r9      //R[r9]←R[rsp]+20
movl $3, %r8d           //R[r8d]←3
leaq 18(%rsp), %rcx     //R[rcx]←R[rsp]+18
movw $2, %dx            //R[dx]←2
leaq 16(%rsp), %rsi     //R[rsi]←R[rsp]+16
movb $1, %dil           //R[dil]←1
call test   第15条指令
```

执行到 caller 的 call 指令时栈中情况




```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

数据传送


```
movq 16(%rsp), %r10 //R[r10]←M[R[rsp]+16]
addb %dil, (%rsi)   //M[R[rsi]]←M[R[rsi]]+R[dil]
addw %dx, (%rcx)    //M[R[rcx]]←M[R[rcx]]+R[dx]
addl %r8d, (%r9)    //M[R[r9]]←M[R[r9]]+R[r8d]
movq 8(%rsp), %rax  //R[rax]←M[R[rsp]+8]
addq %rax, (%r10)   //M[R[r10]]←M[R[r10]]+R[rax]
ret
```

R[r10]←&d

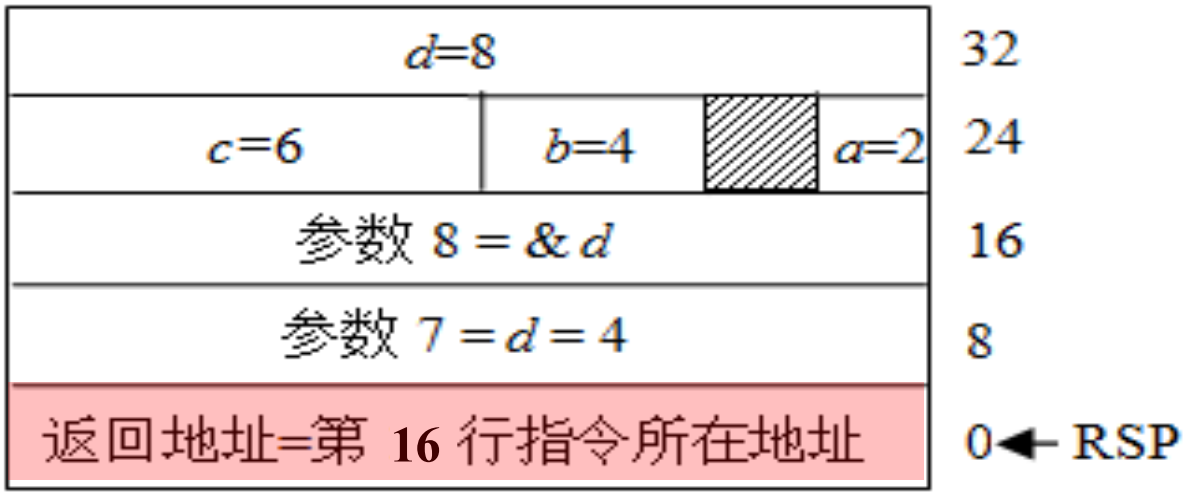
*ap+=a;

*bp+=b;

*cp+=c;

*dp+=d;

执行到test的ret指令前，栈中的状态如何？ret执行后怎样?



DIL、RSI、DX、RCX、R8D、R9

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

数据传送

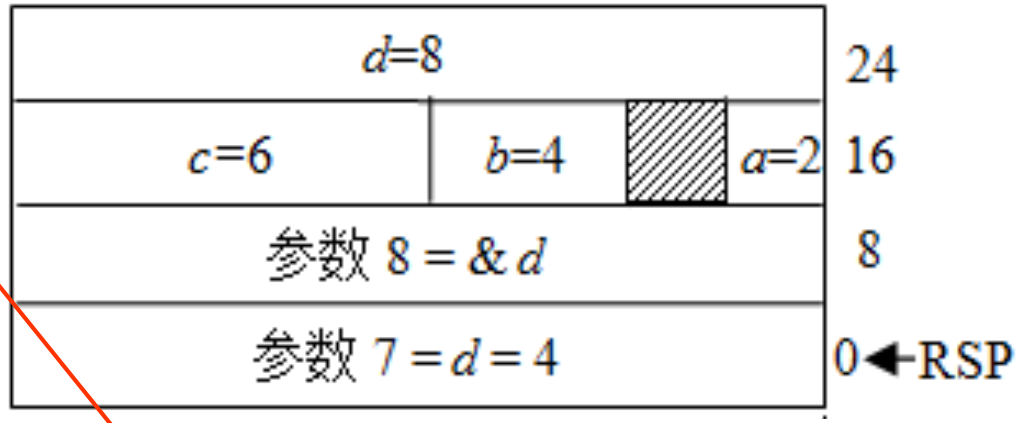
从第16条指令开始

```
movslq 20(%rsp), %rcx
movq   24(%rsp), %rdx
imulq  %rdx, %rcx
movsbw 16(%rsp), %ax
movw   18(%rsp), %dx
imulw  %dx, %ax
movswq %ax, %rax
leaq   (%rax, %rcx),%rax
addq   $32, %rsp
ret
```

释放caller的栈帧

执行到ret指令时，RSP指向调用caller函数时保存的返回值

执行test的ret指令后，栈中的状态如何?




```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

寄存器保存约定

- 当过程 **main**调用过程 **who**:
  - **mian**是caller调用者
  - **who**是callee被调用者

```
main:
    . . .
    movq $15210, %rdx
    movq $15213, %rbx
    call who
    movq %rdx, %rax
    addq %rbx, %rax
    . . .
```

```
who:
    . . .
    movq $1000, %rdx
    subq $18213, %rbx
    . . .
    ret
```

  - 寄存器 **%rbx,%rdx** 被 **who**覆盖

x86-64 采用了一组统一的寄存器使用惯例，所有的过程(包括程序库)都必须遵循。

寄存器保存约定

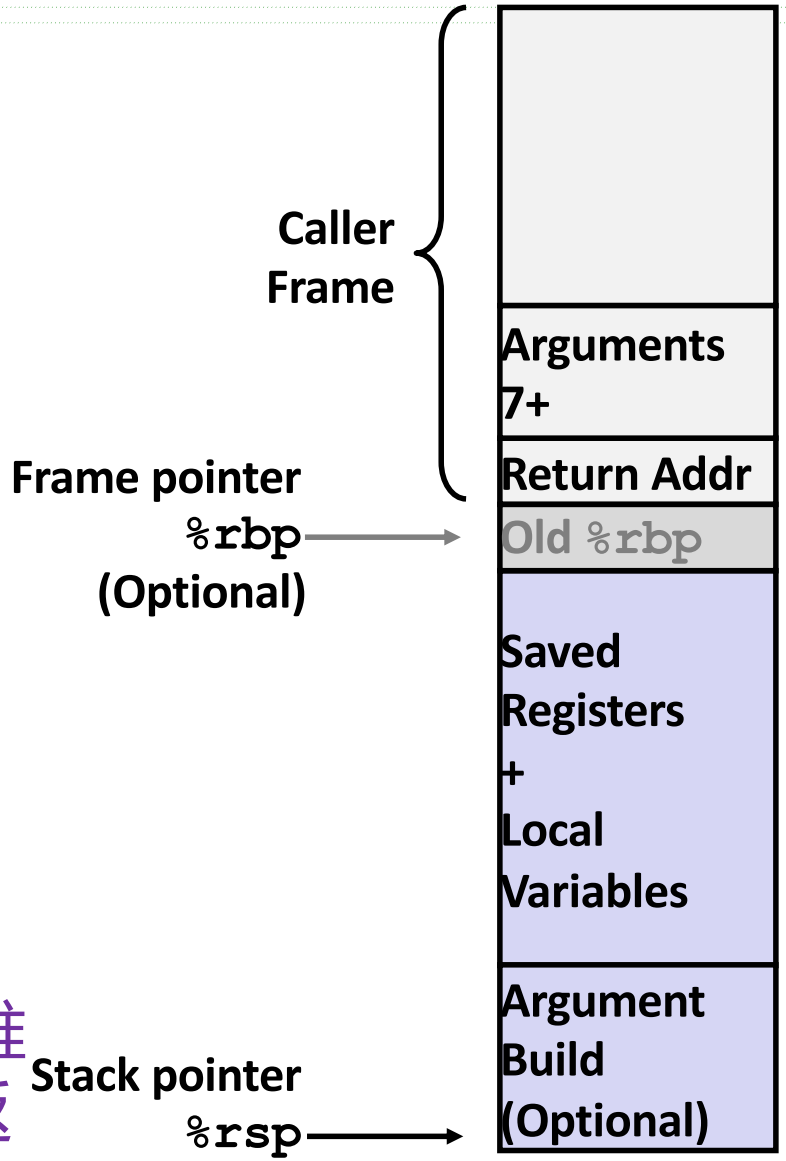
- 约定
  - “调用者保存寄存器”

    调用者在call调用之前保存这些寄存器的值。
  - “被调用者保存寄存器”

    被调用者负责在使用前保存这些寄存器的值。

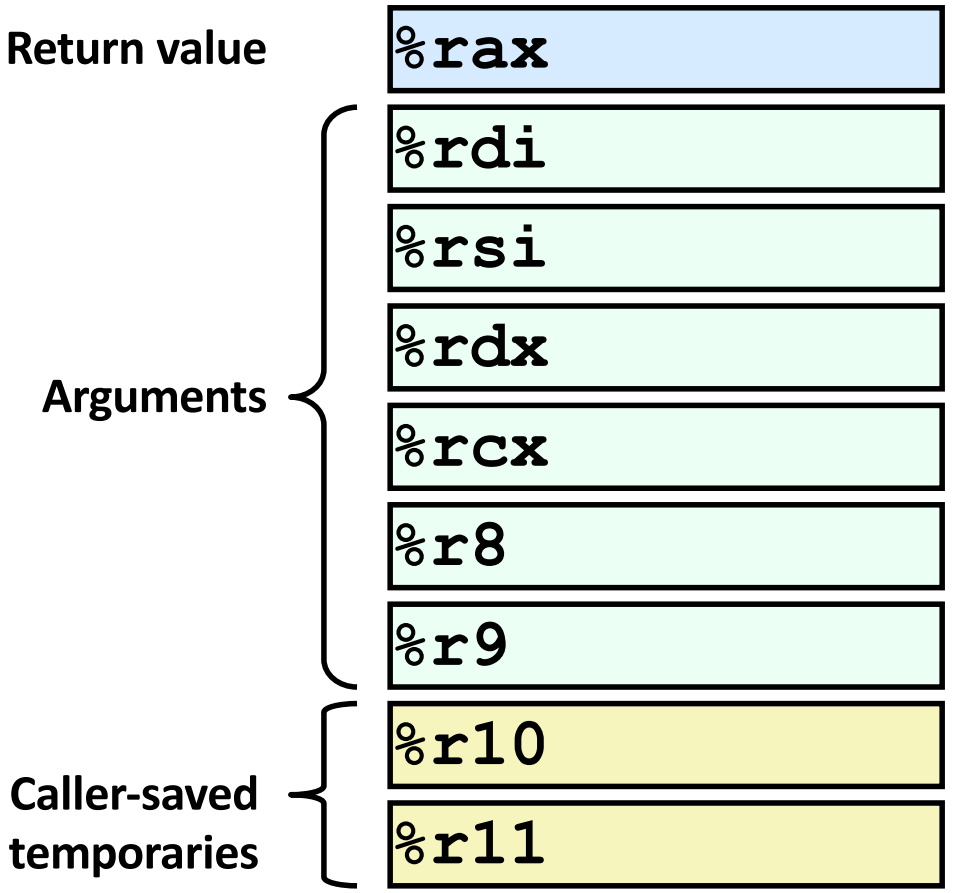
    被调用者负责在过程返回前恢复原来的值

保存方法：把寄存器原始值压入堆栈中，改变寄存器的值，然后在返回前从栈中弹出旧值。



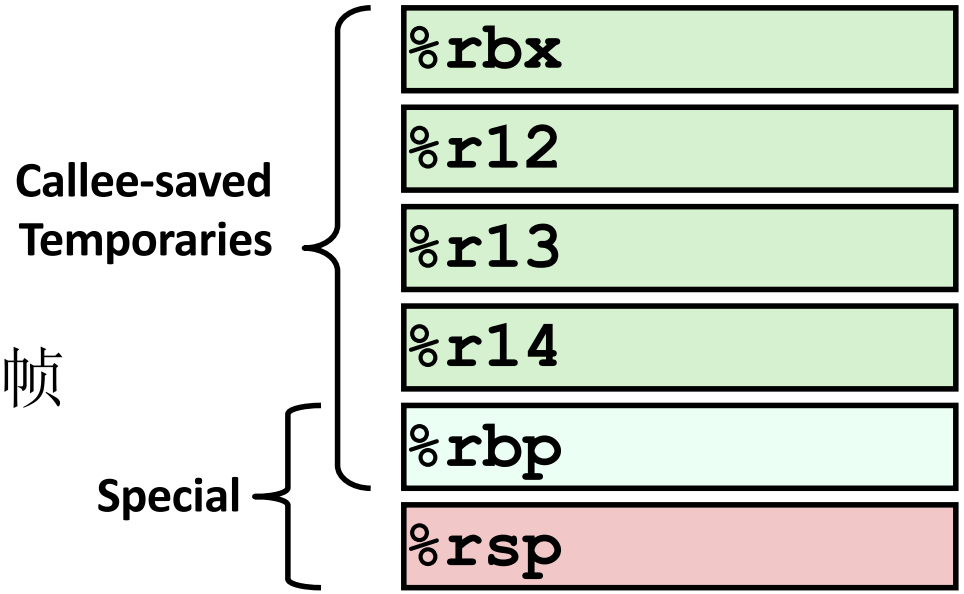
x86-64 Linux Register Usage #1

- **%rax**
  - 返回值
  - 调用者保护
  - 值可以被过程改变
- **%rdi**, ..., **%r9**
  - 参数
  - 调用者保护
  - 值可以被过程改变
- **%r10**, **%r11**
  - 调用者保护
  - 值可以被过程改变



x86-64 Linux Register Usage #2

- **%rbx**, **%r12**, **%r13**, **%r14**
  - 被调用者保护
  - 被调用者必须保存和恢复
- **%rbp**
  - 被调用者保护
  - 被调用者必须保存和恢复栈帧



```
main:
    • • •
    movq $15210, %rdx
    movq $15213, %rbx
    pushq %rdx
    call who
    popq %rdx
    movq %rdx, %rax
    addq %rbx, %rax
    • • •
```

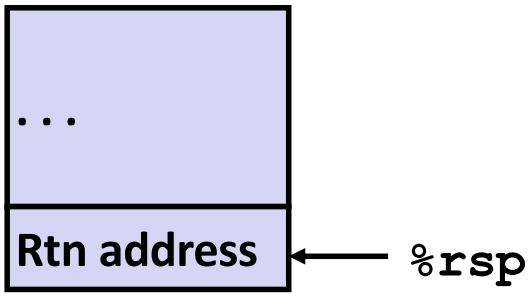
```
who:
  pushq  %rbx
  movq $1000, %rdx
  subq $18213, %rbx
  popq %rbx
  • • •
  ret
```

被调用者保护 Example

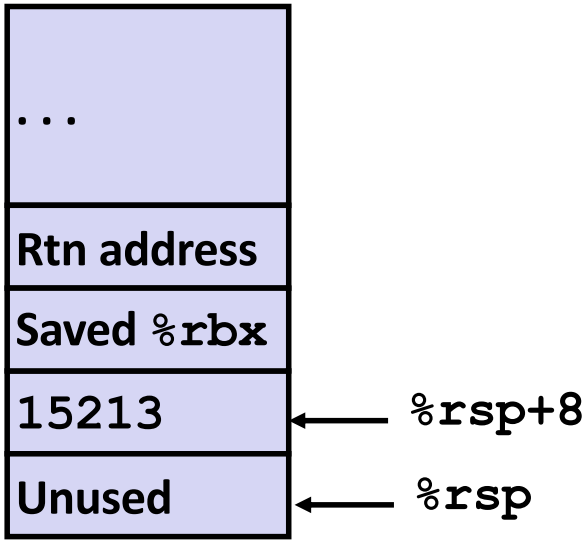
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  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
```

Initial Stack Structure



Resulting Stack Structure

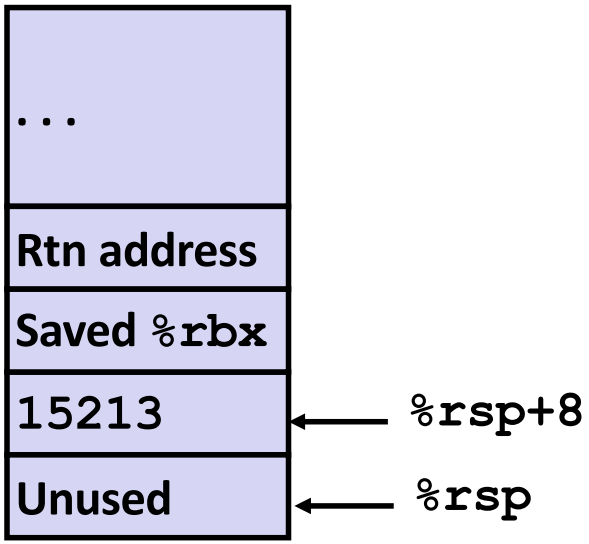


被调用者保护 Example

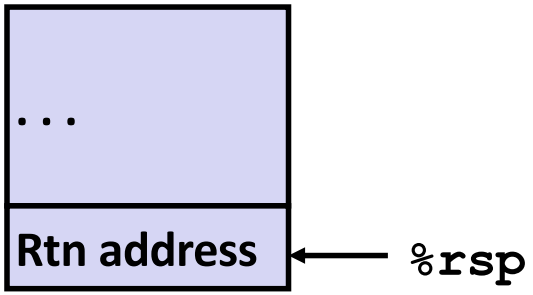

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  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
```

Resulting Stack Structure



Pre-return Stack Structure



递归过程


```
/* Recursive pcount */
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
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归过程


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

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

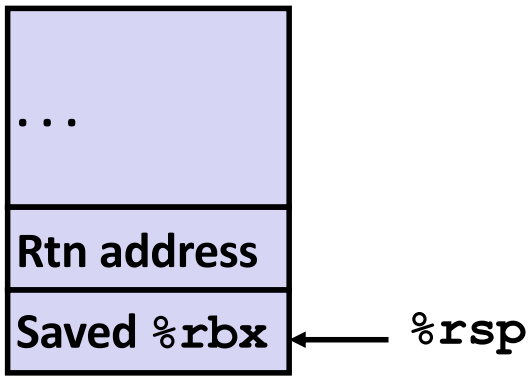
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归过程


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

Register	Use(s)	Type
%rdi	x	Argument

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



递归过程


```
/* Recursive pcount */
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
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Recursive argument
%rbx	x & 1	Callee-saved

递归过程


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

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

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

递归过程


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

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

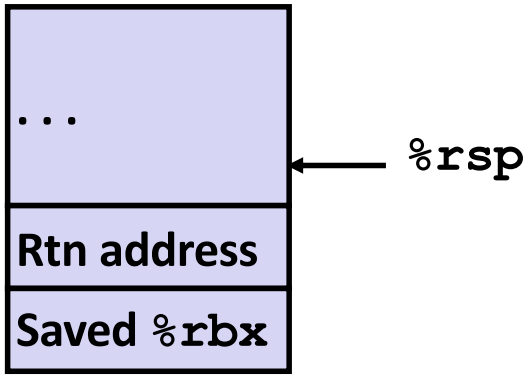
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归过程

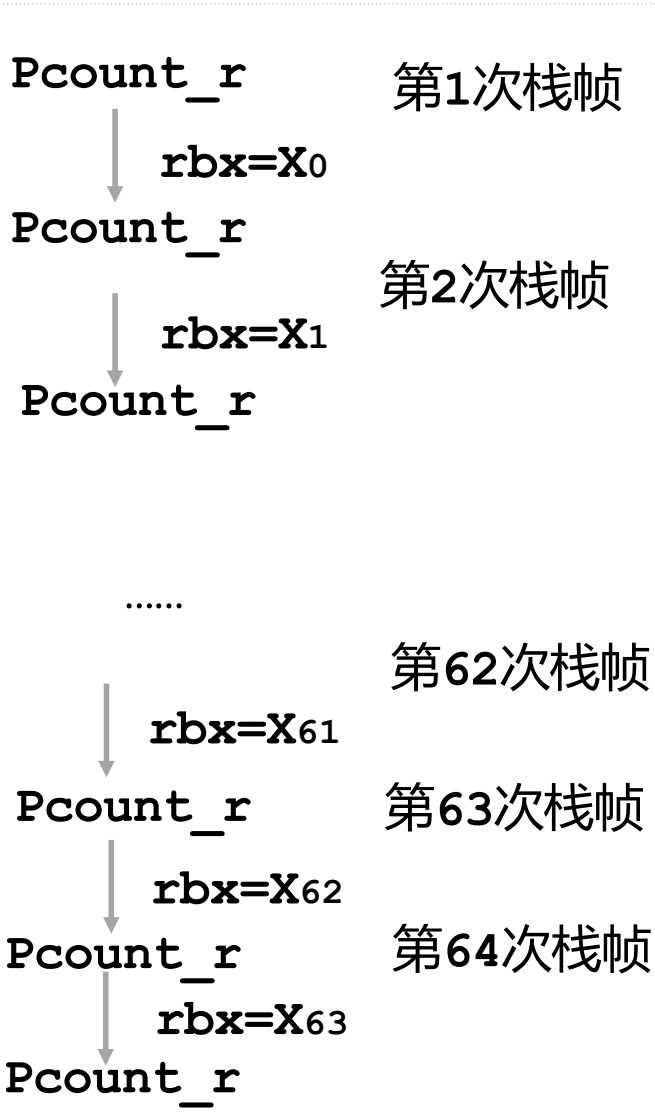

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value

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



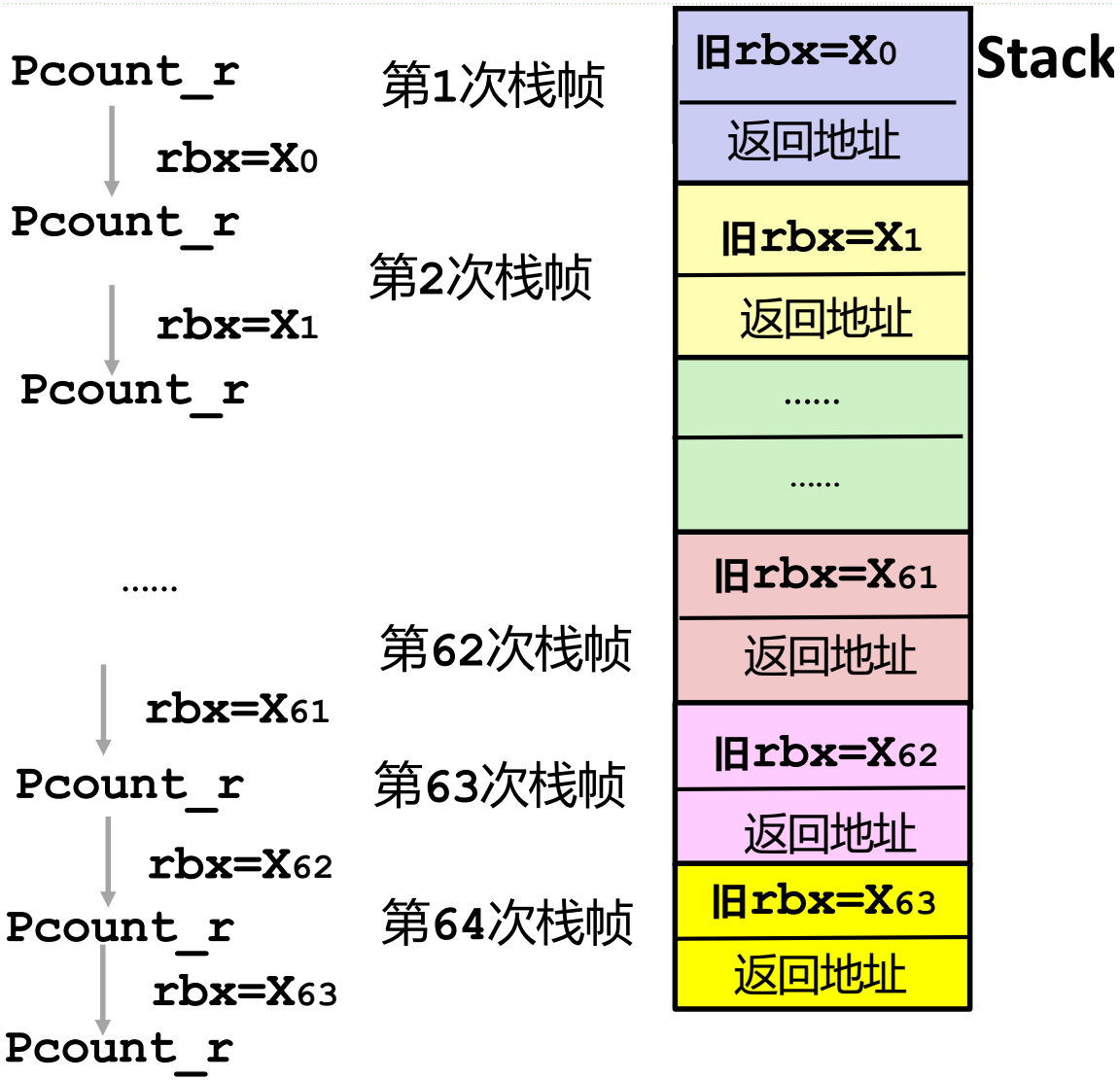
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```



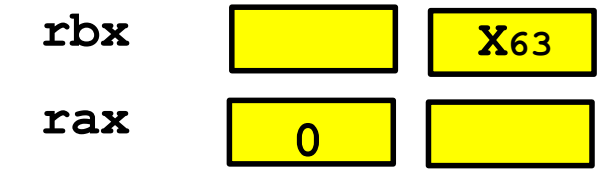
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

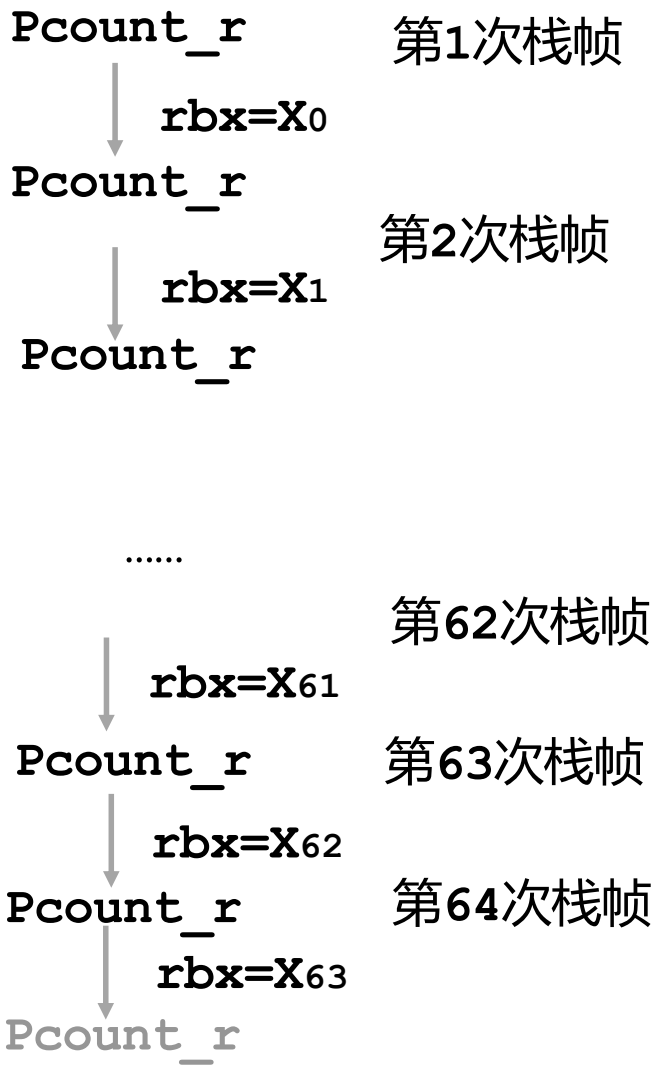
rbx

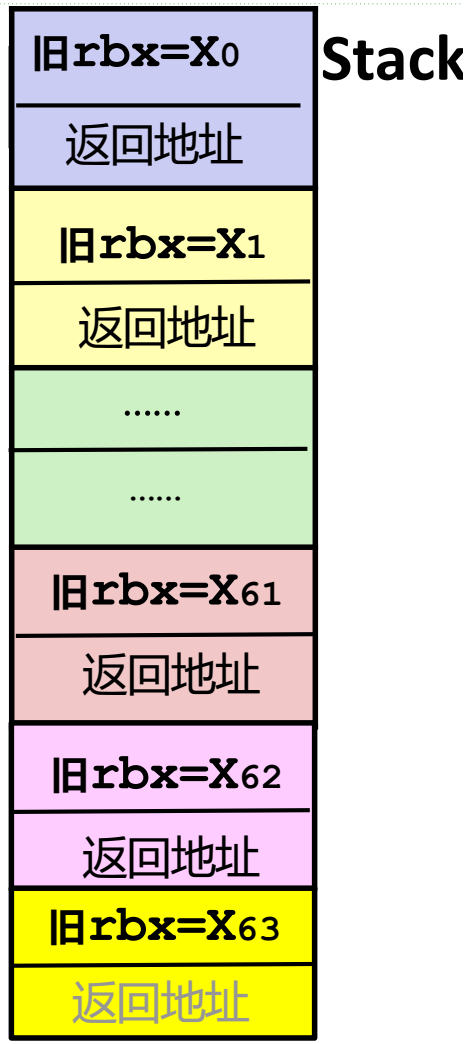
rax 0



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

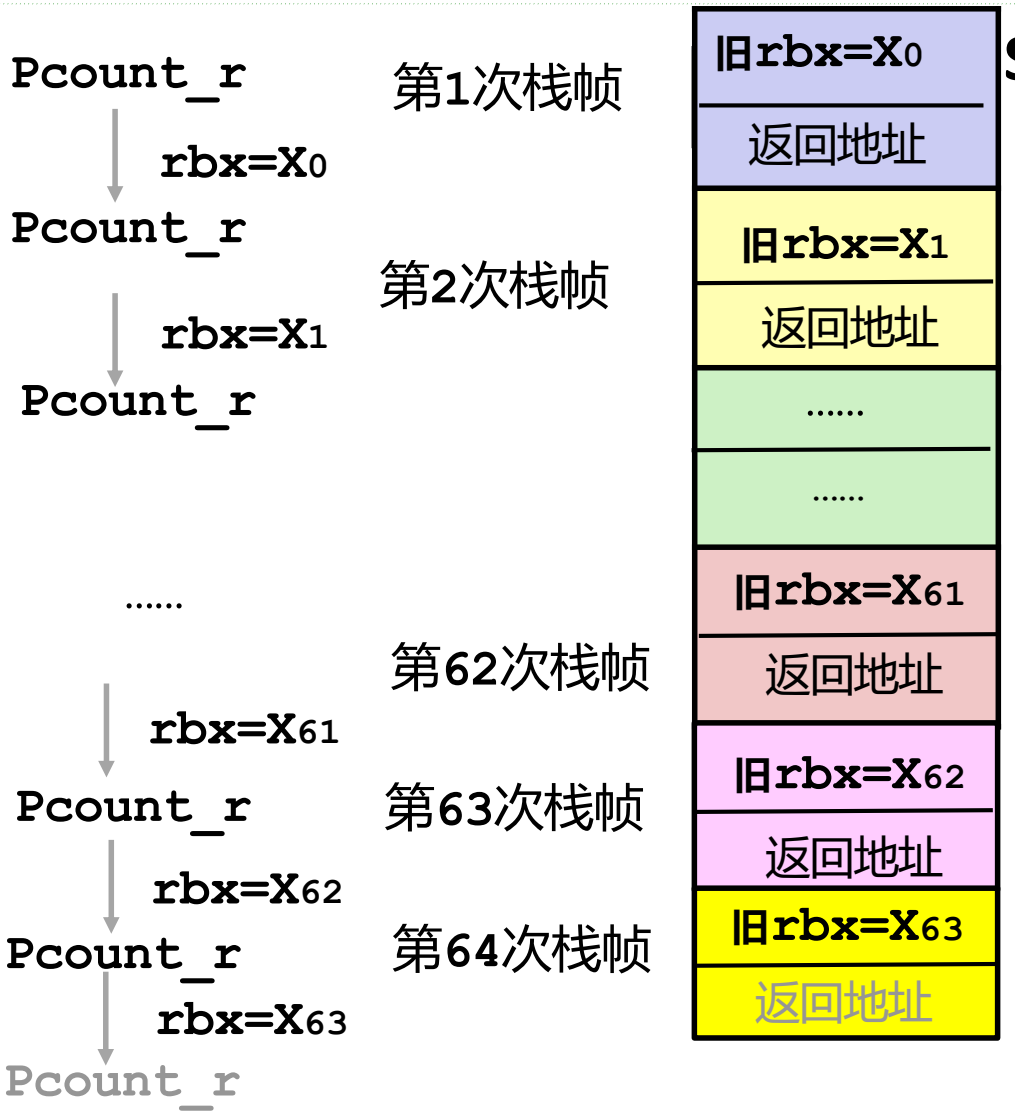




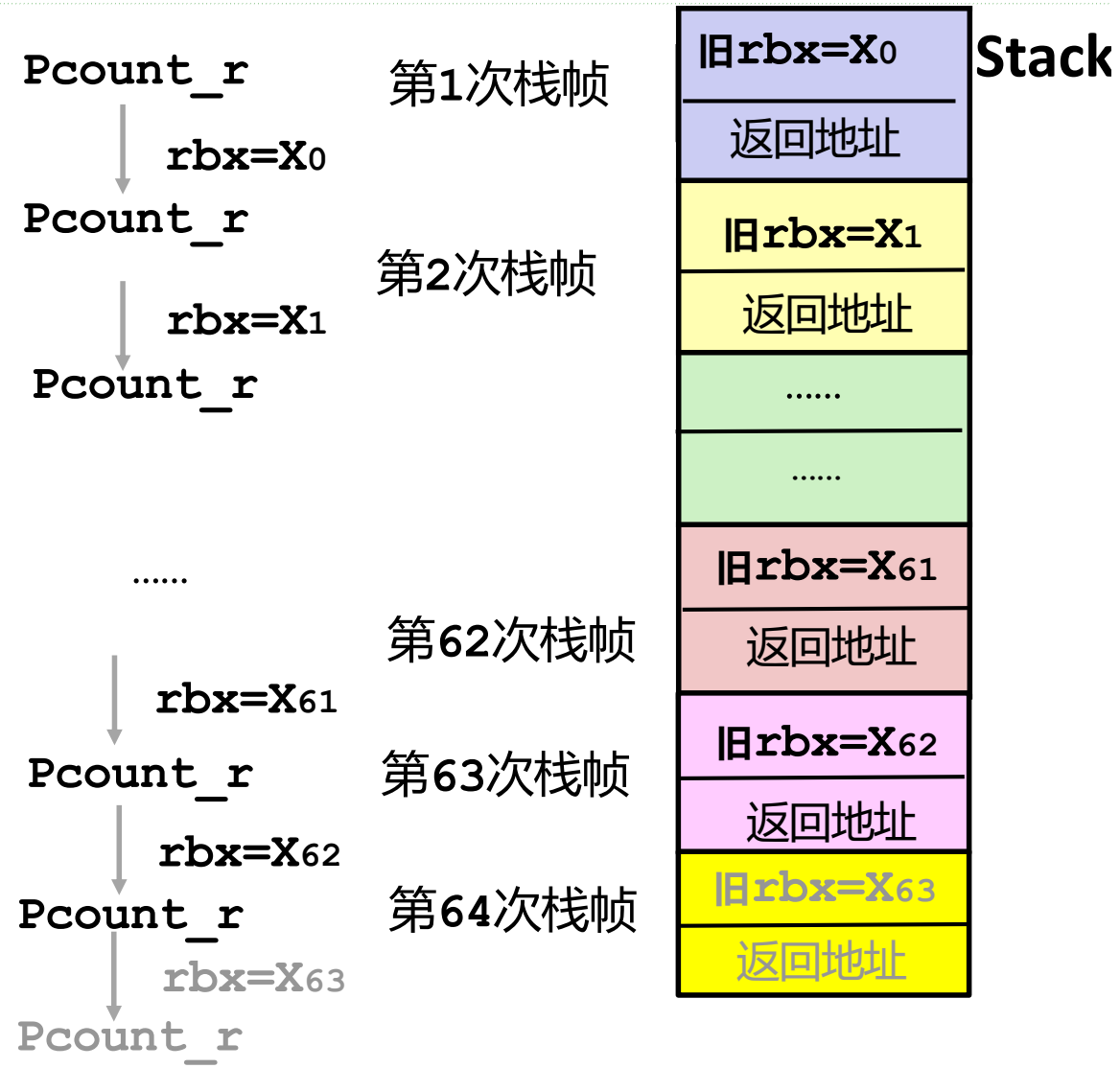


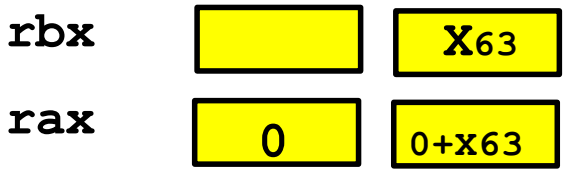
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

rbx		X63
rax	0	0+X63

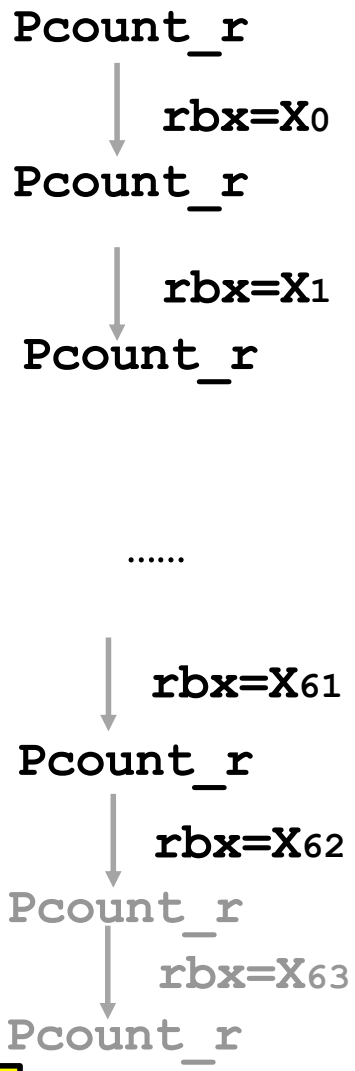


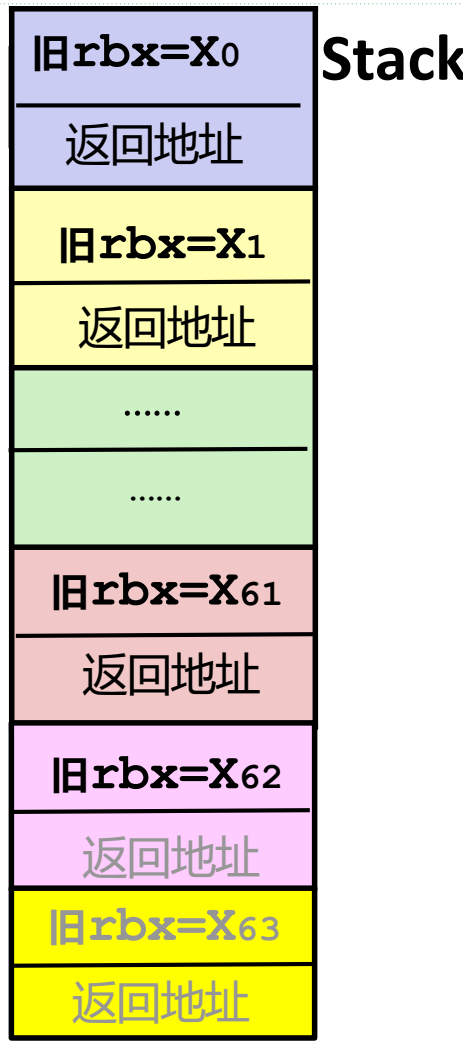
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

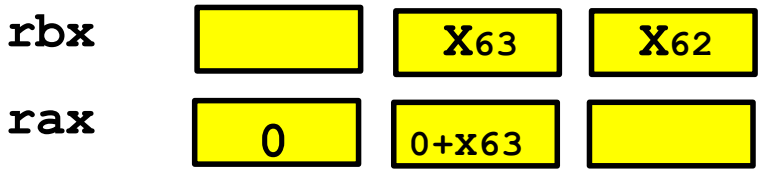




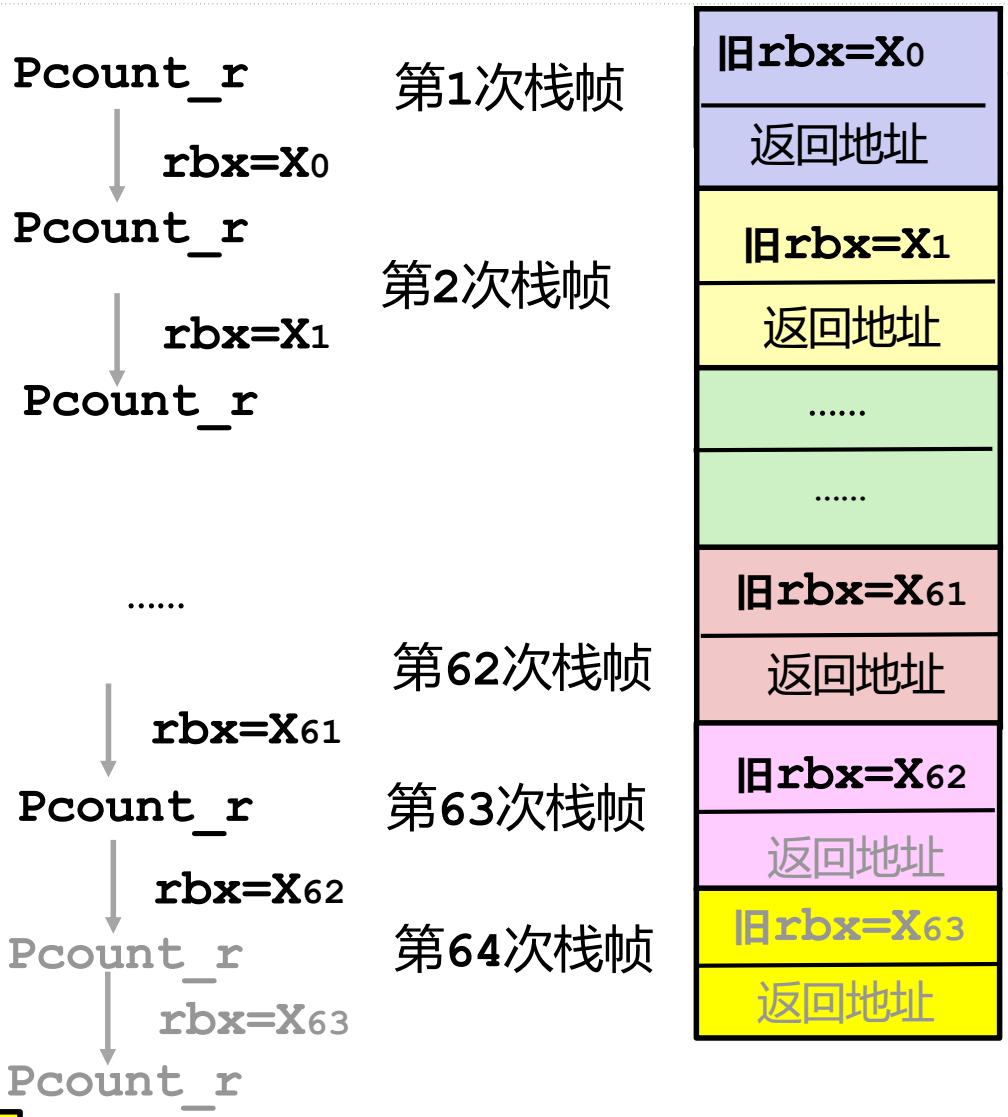
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

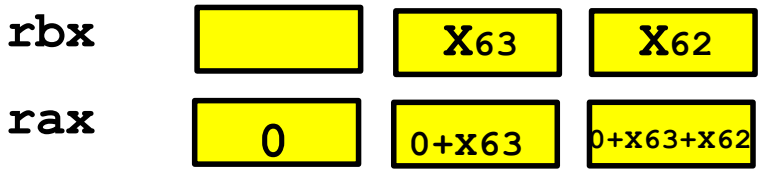






```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```





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