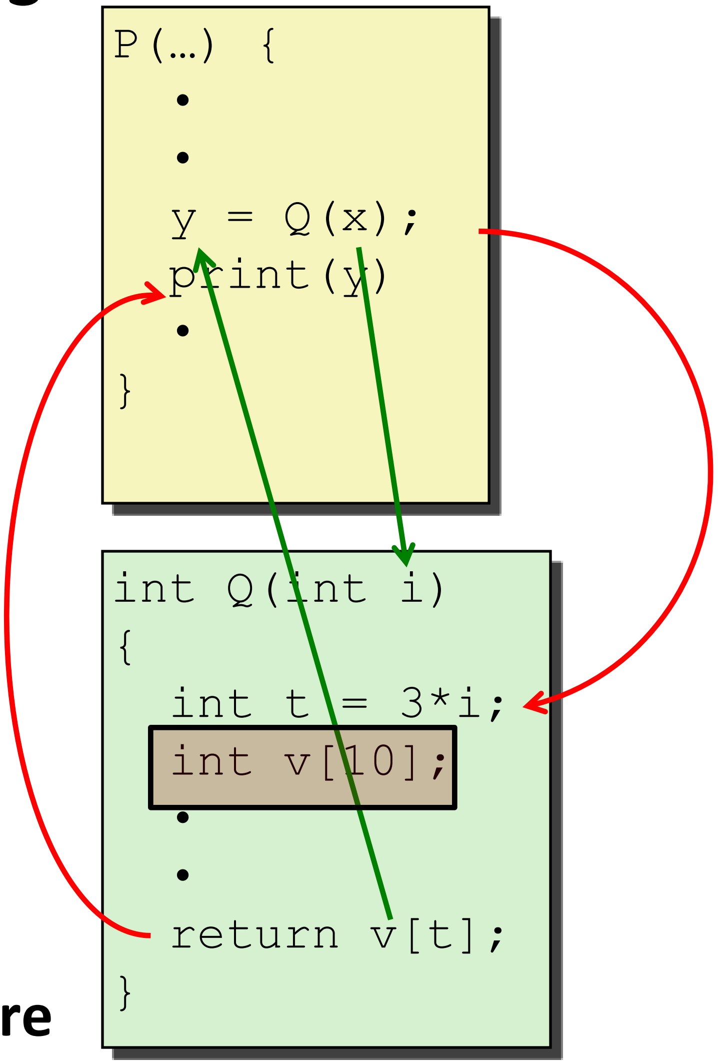
Machine-Level Programming III: Procedures

Instructor:

R. Shathanaa

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

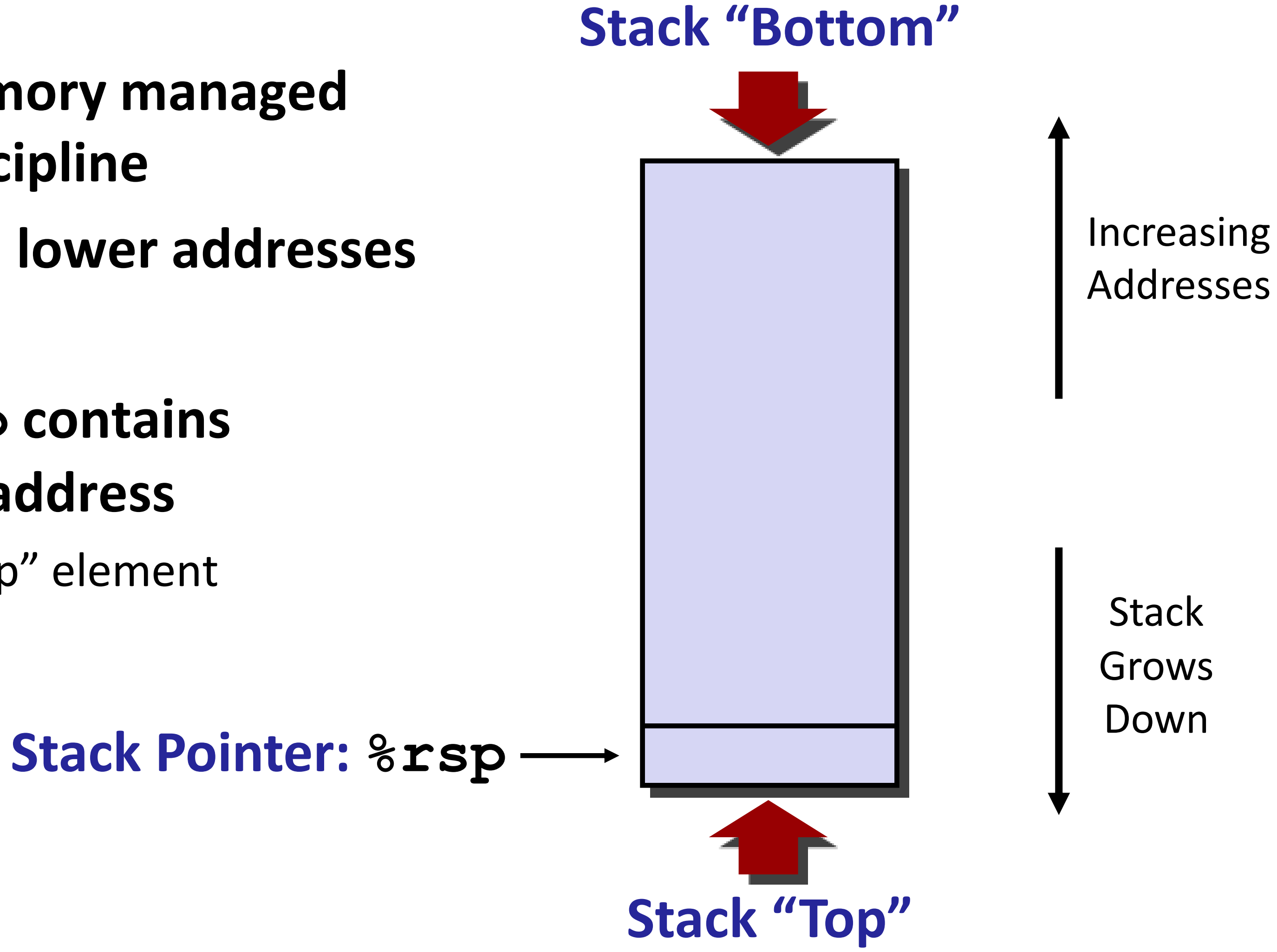
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion

x86-64 Stack

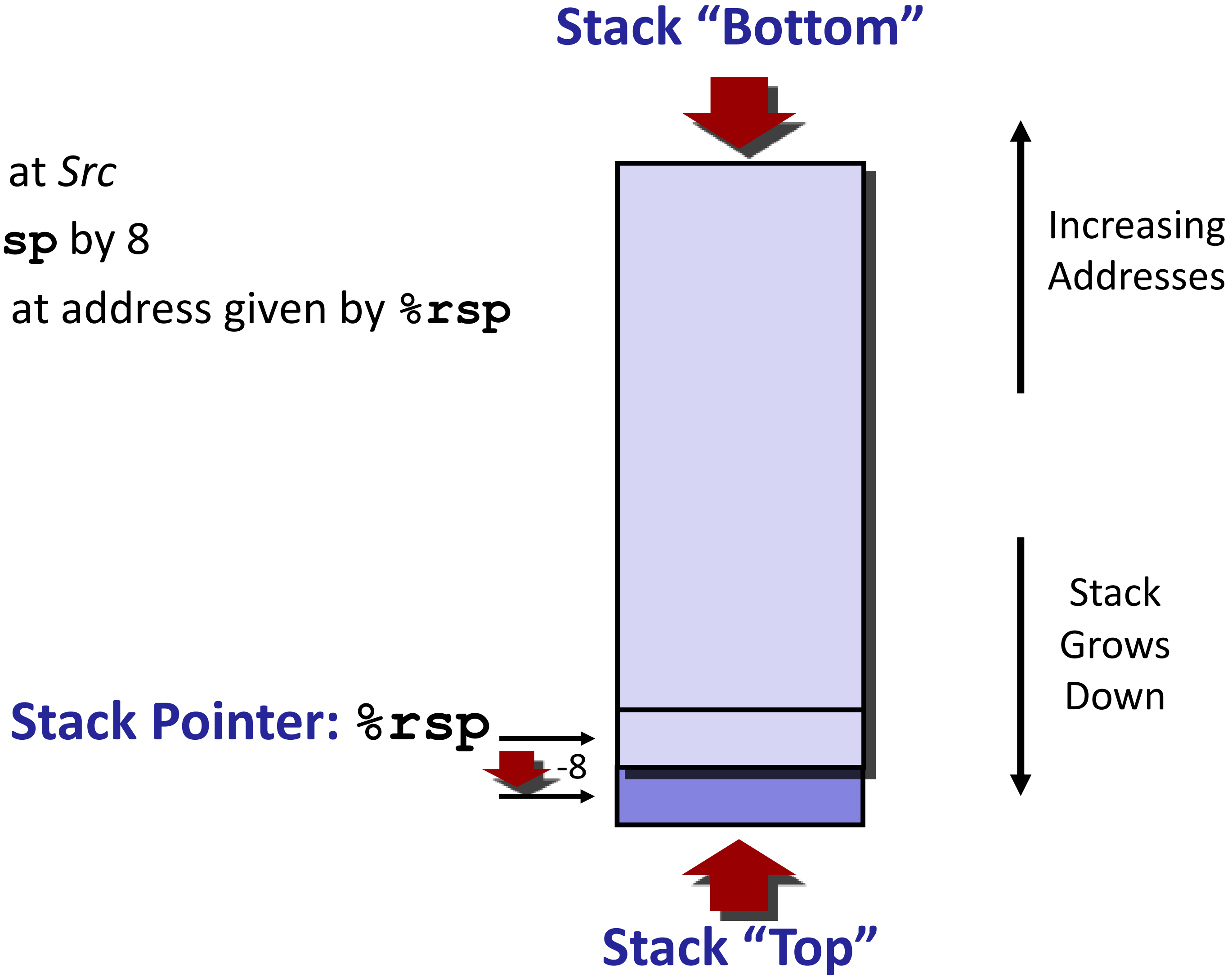
- **Region of memory managed with stack discipline**
- **Grows toward lower addresses**

- **Register `%rsp` contains lowest stack address**
  - address of "top" element



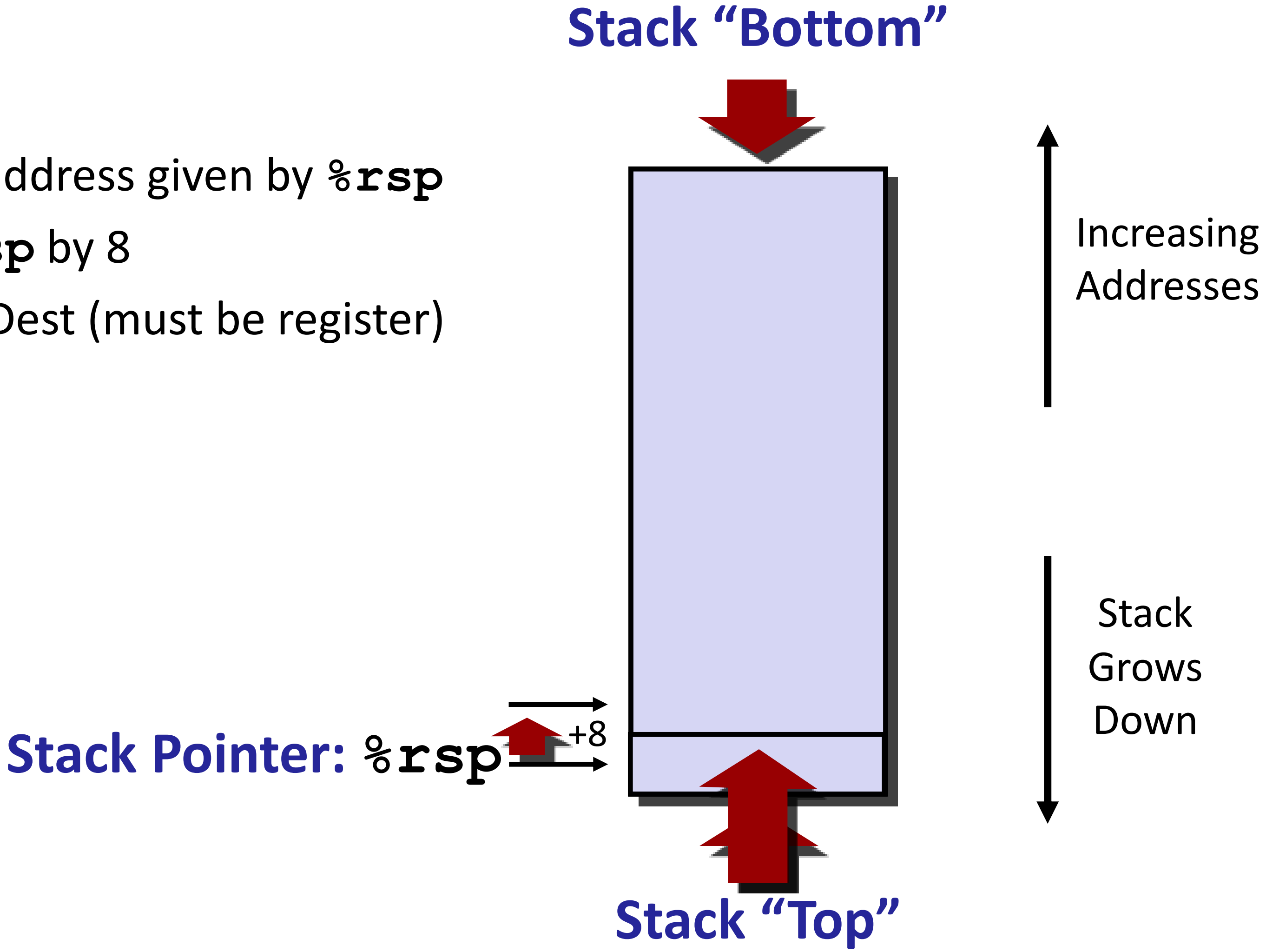
x86-64 Stack: Push

- `pushq` ***Src***
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



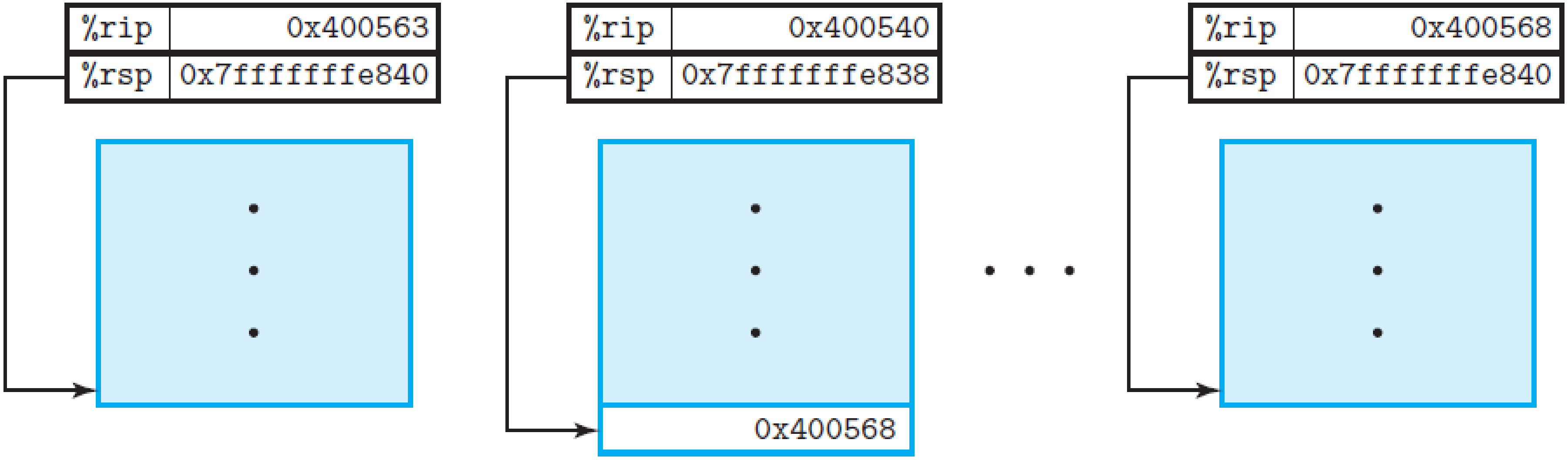
x86-64 Stack: Pop

- `popq` ***Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (must be register)



Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - **Passing control**
    - Passing data
    - Managing local data
  - Illustration of Recursion



(a) Executing `call`

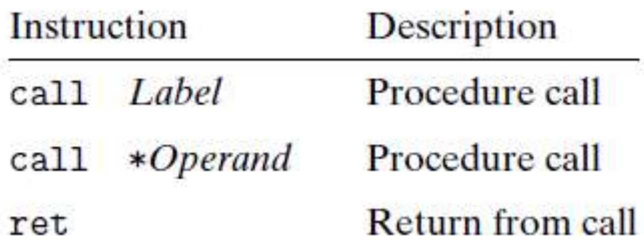
(b) After `call`

(c) After `ret`

```
Beginning of function multstore
0000000000400540 <multstore>:
 400540:  53                    push   %rbx
 400541:  48 89 d3              mov    %rdx,%rbx
 . . .
Return from function multstore
 40054d:  c3                    retq
 . . .
Call to multstore from main
 400563:  e8 d8 ff ff ff        callq  400540 <multstore>
 400568:  48 8b 54 24 08        mov    0x8(%rsp),%rdx
```

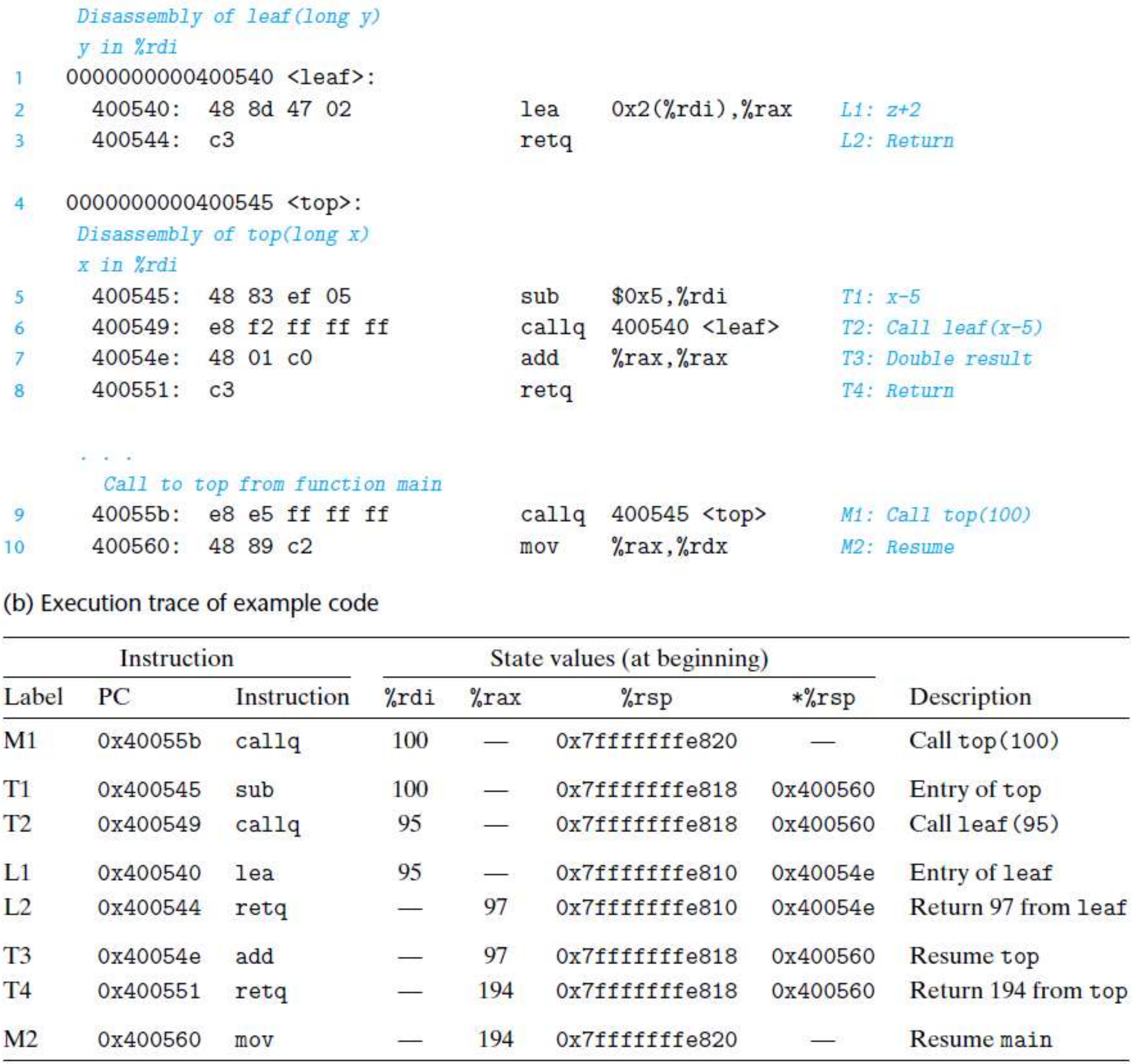
Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to ***label***



Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`ret`		Return from call

- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address



```
   Disassembly of leaf(long y)
   y in %rdi
0000000000400540 <leaf>:
  400540:  48 8d 47 02             lea    0x2(%rdi),%rax    L1: z+2
  400544:  c3                      retq                     L2: Return

0000000000400545 <top>:
   Disassembly of top(long x)
   x in %rdi
  400545:  48 83 ef 05             sub    $0x5,%rdi         T1: x-5
  400549:  e8 f2 ff ff ff          callq  400540 <leaf>     T2: Call leaf(x-5)
  40054e:  48 01 c0                add    %rax,%rax         T3: Double result
  400551:  c3                      retq                     T4: Return

   . . .
    Call to top from function main
  40055b:  e8 e5 ff ff ff          callq  400545 <top>      M1: Call top(100)
  400560:  48 89 c2                mov    %rax,%rdx         M2: Resume
```

(b) Execution trace of example code

Instruction			State values (at beginning)				
Label	PC	Instruction	%rdi	%rax	%rsp	*%rsp	Description
M1	0x40055b	callq	100	—	0x7fffffffe820	—	Call top(100)
T1	0x400545	sub	100	—	0x7fffffffe818	0x400560	Entry of top
T2	0x400549	callq	95	—	0x7fffffffe818	0x400560	Call leaf(95)
L1	0x400540	lea	95	—	0x7fffffffe810	0x40054e	Entry of leaf
L2	0x400544	retq	—	97	0x7fffffffe810	0x40054e	Return 97 from leaf
T3	0x40054e	add	—	97	0x7fffffffe818	0x400560	Resume top
T4	0x400551	retq	—	194	0x7fffffffe818	0x400560	Return 194 from top
M2	0x400560	mov	—	194	0x7fffffffe820	—	Resume main

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**
    - Managing local data
  - Illustrations of Recursion & Pointers

Procedure Data Flow

Registers

- First 6 arguments

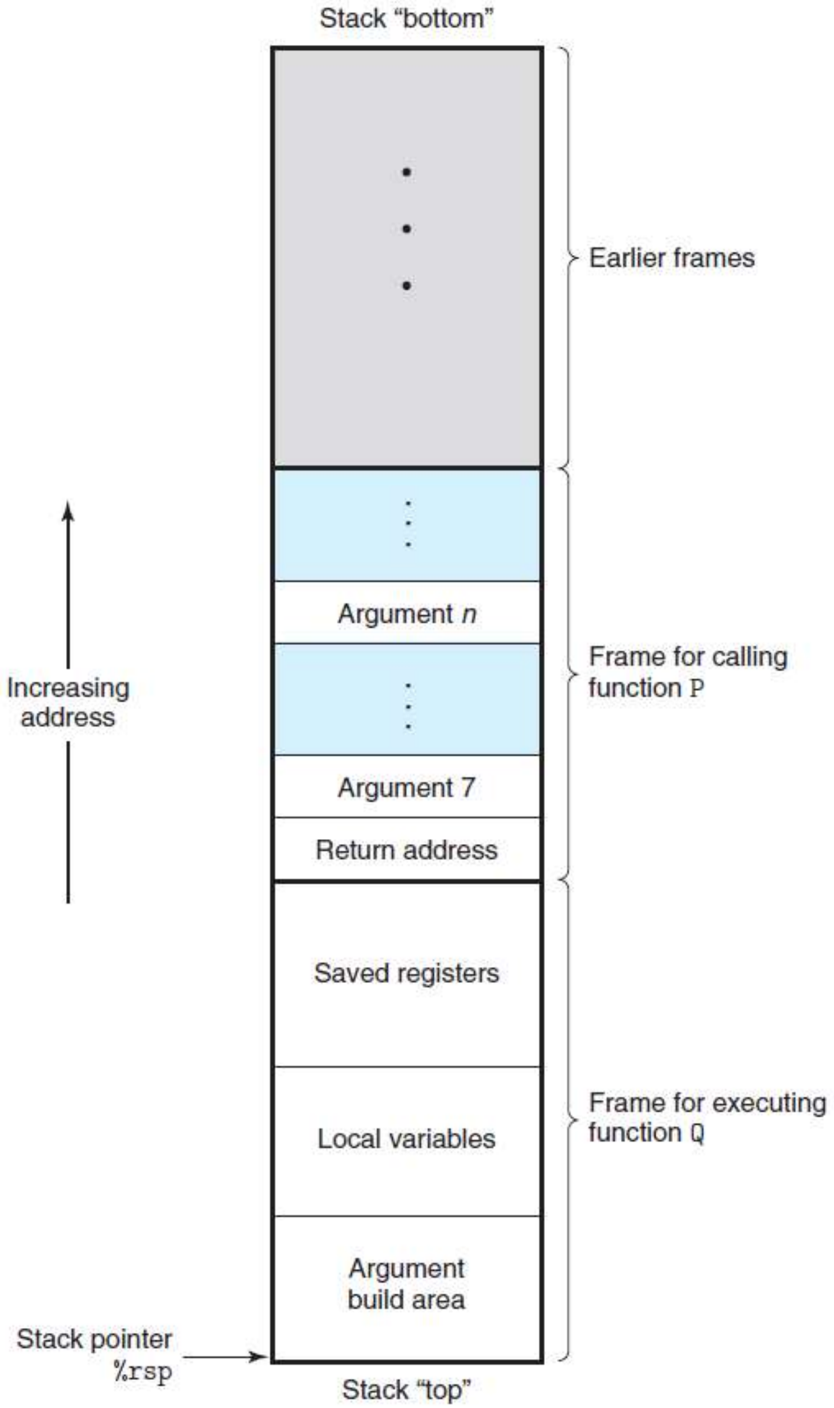
```
%rdi
%rsi
%rdx
%rcx
%r8
%r9
```

- Return value

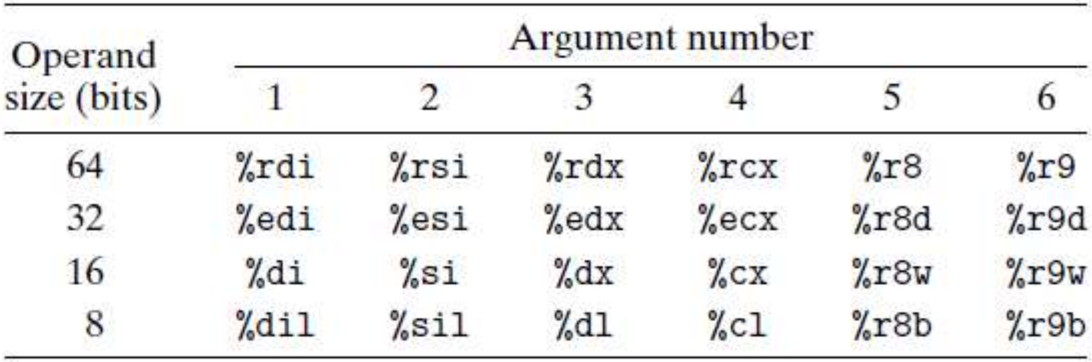
```
%rax
```

Stack

- Only allocate stack space when needed



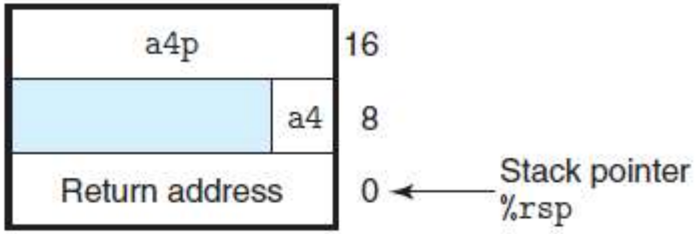
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

(a) C code

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```



(b) Generated assembly code

```
    void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
    Arguments passed as follows:
      a1  in %rdi          (64 bits)
      a1p in %rsi          (64 bits)
      a2  in %edx          (32 bits)
      a2p in %rcx          (64 bits)
      a3  in %r8w          (16 bits)
      a3p in %r9           (64 bits)
      a4  at %rsp+8        ( 8 bits)
      a4p at %rsp+16       (64 bits)
    proc:
      movq    16(%rsp), %rax    Fetch a4p    (64 bits)
      addq    %rdi, (%rsi)      *a1p += a1   (64 bits)
      addl    %edx, (%rcx)      *a2p += a2   (32 bits)
      addw    %r8w, (%r9)       *a3p += a3   (16 bits)
      movl    8(%rsp), %edx     Fetch a4     ( 8 bits)
      addb    %dl, (%rax)       *a4p += a4   ( 8 bits)
      ret                       Return
```

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data**
  - Illustration of Recursion

Call Chain Example

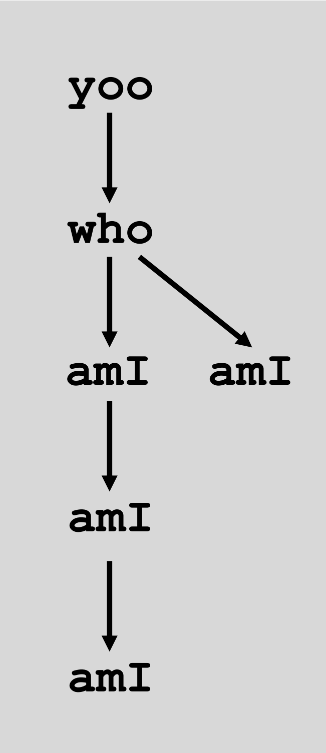
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

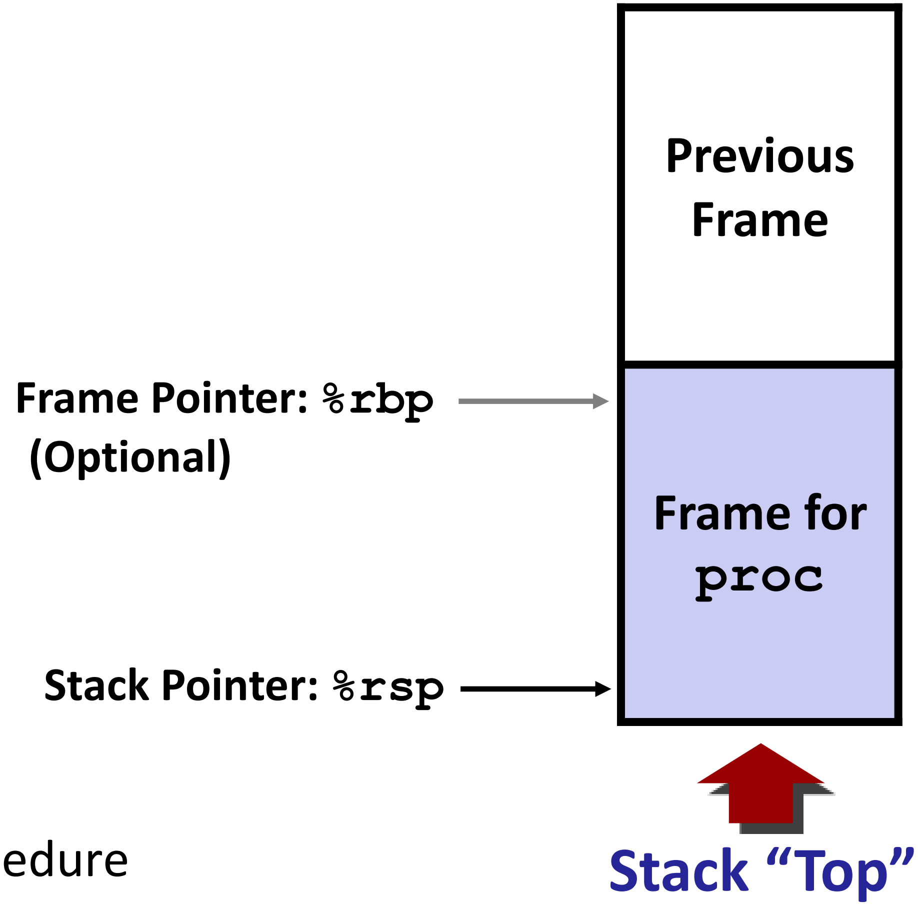
Procedure **amI()** is recursive

Example Call Chain



Stack Frames

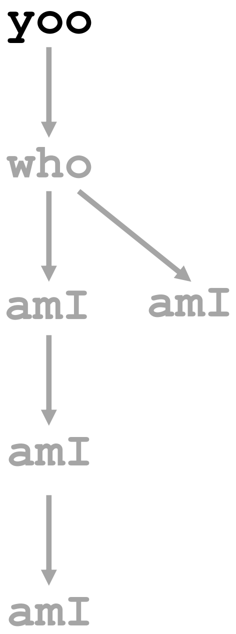
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

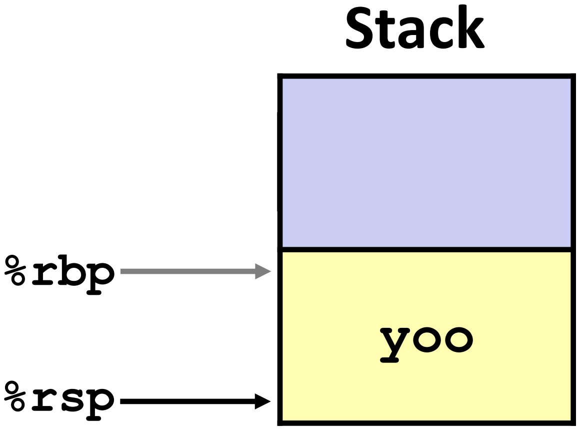


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by `ret` instruction

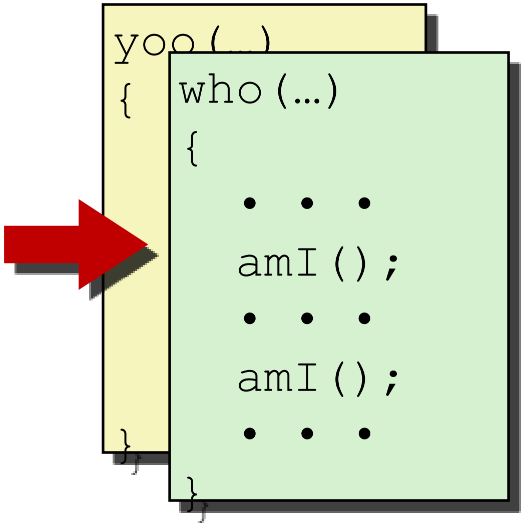
Example

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



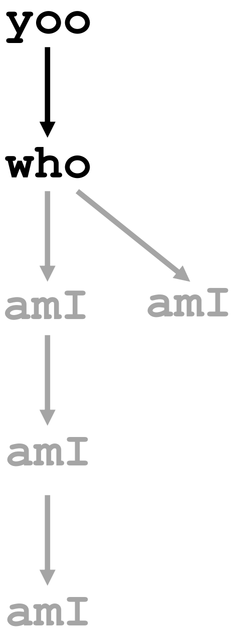


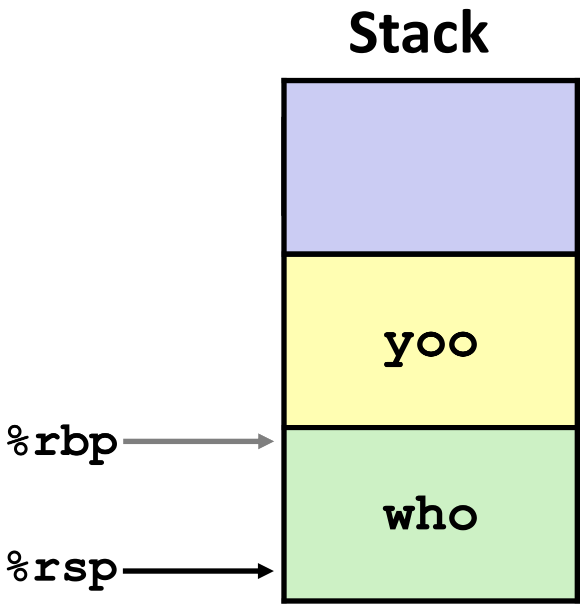
Example



```
yoo(…)
{

}
```

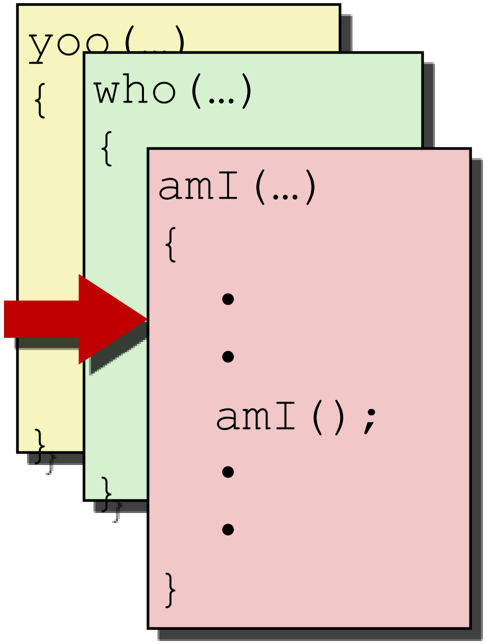
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```





Stack

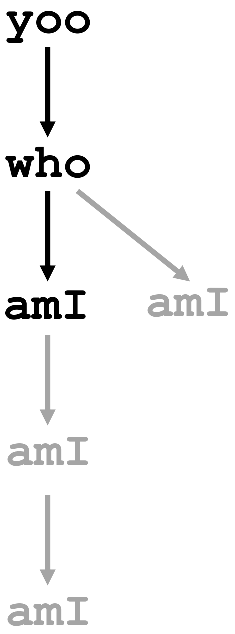
Example

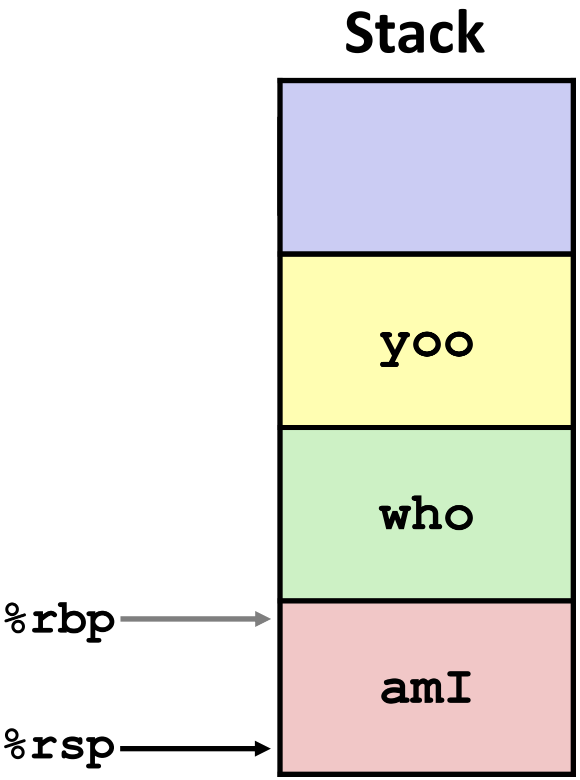


```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

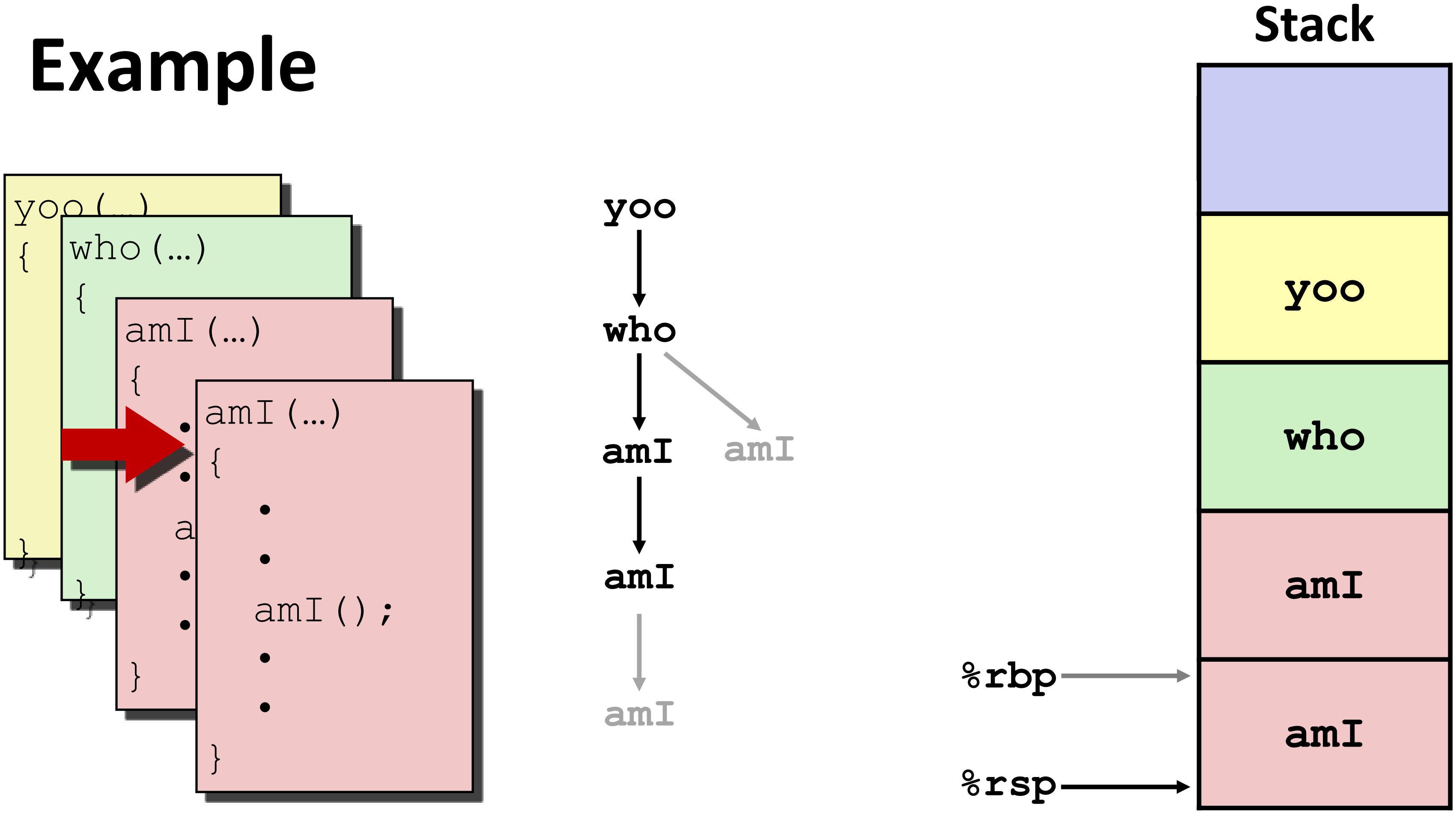
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

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

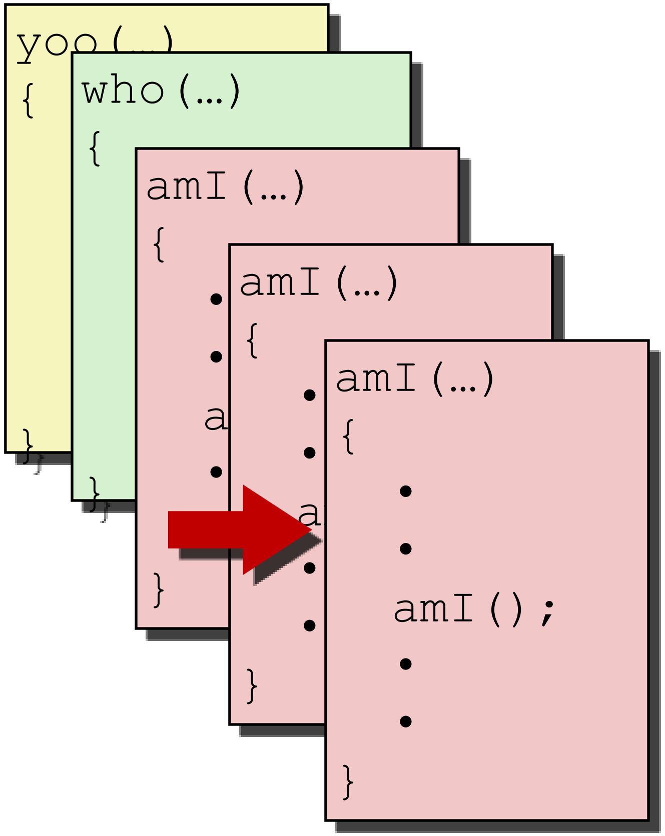


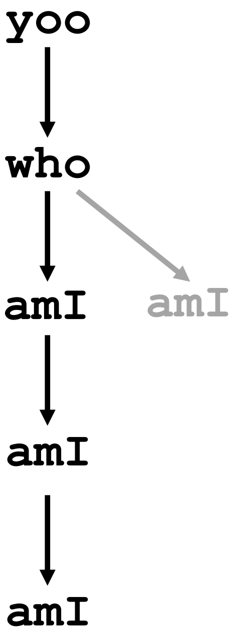


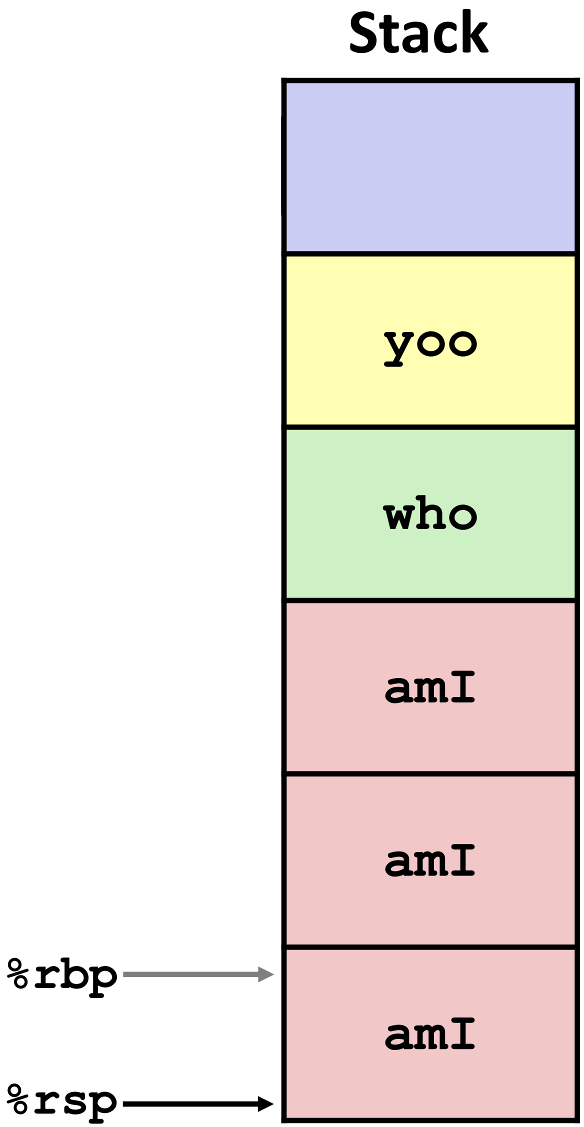
Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

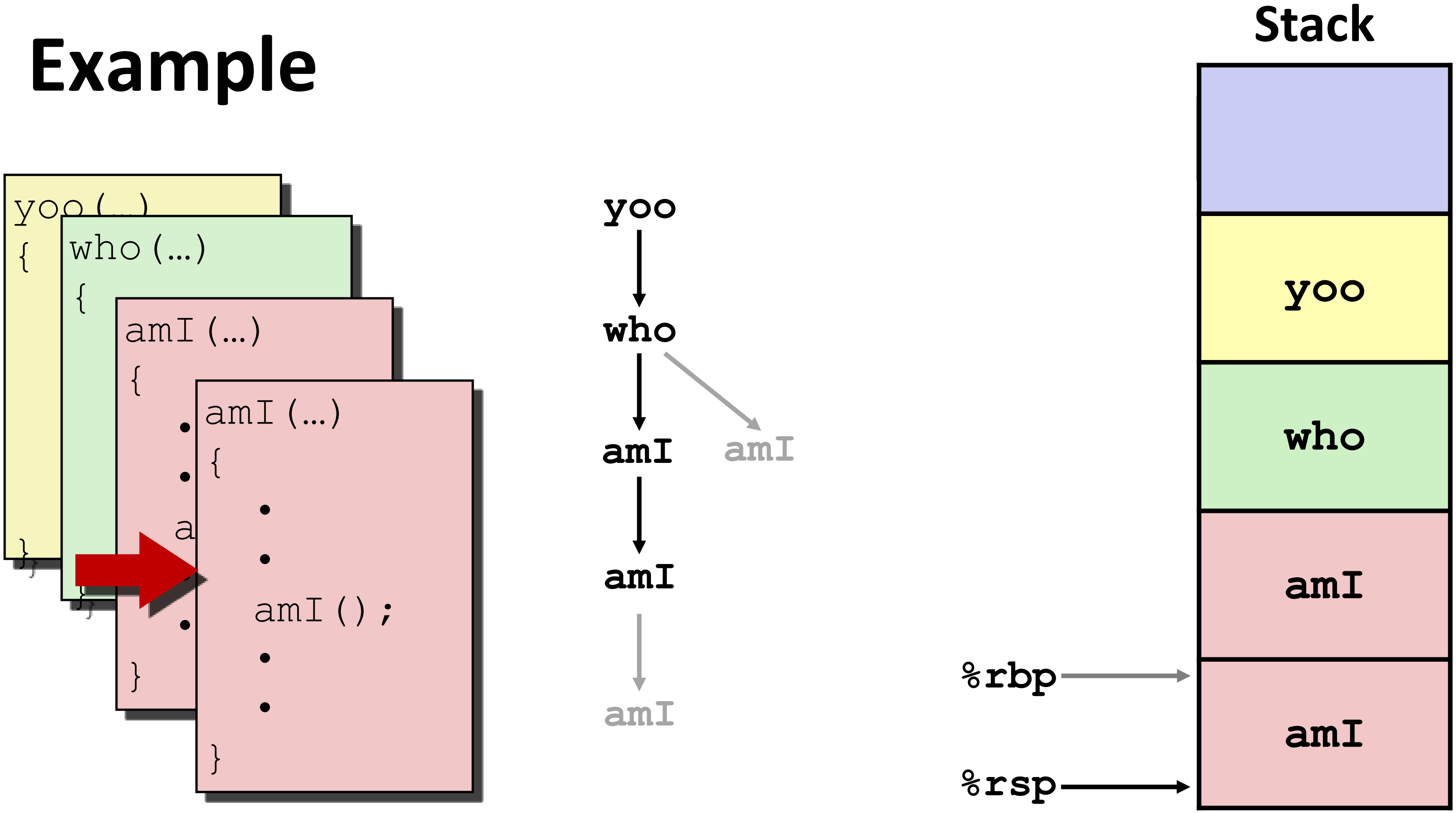
Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}


yoo
who
amI
amI
amI
amI

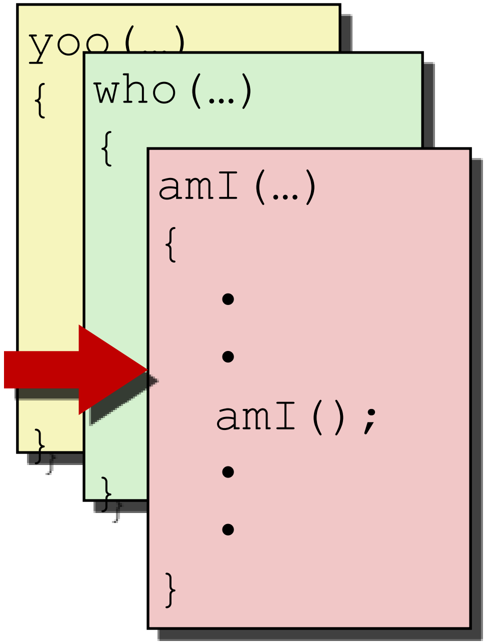

Stack
yoo
who
amI
amI
amI
%rbp
%rsp

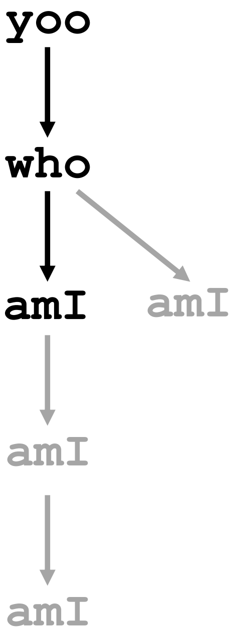
Example

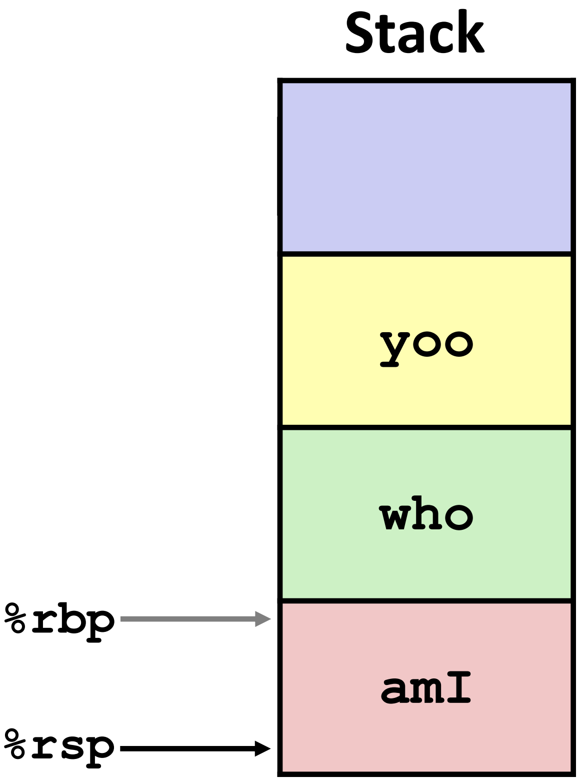


```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

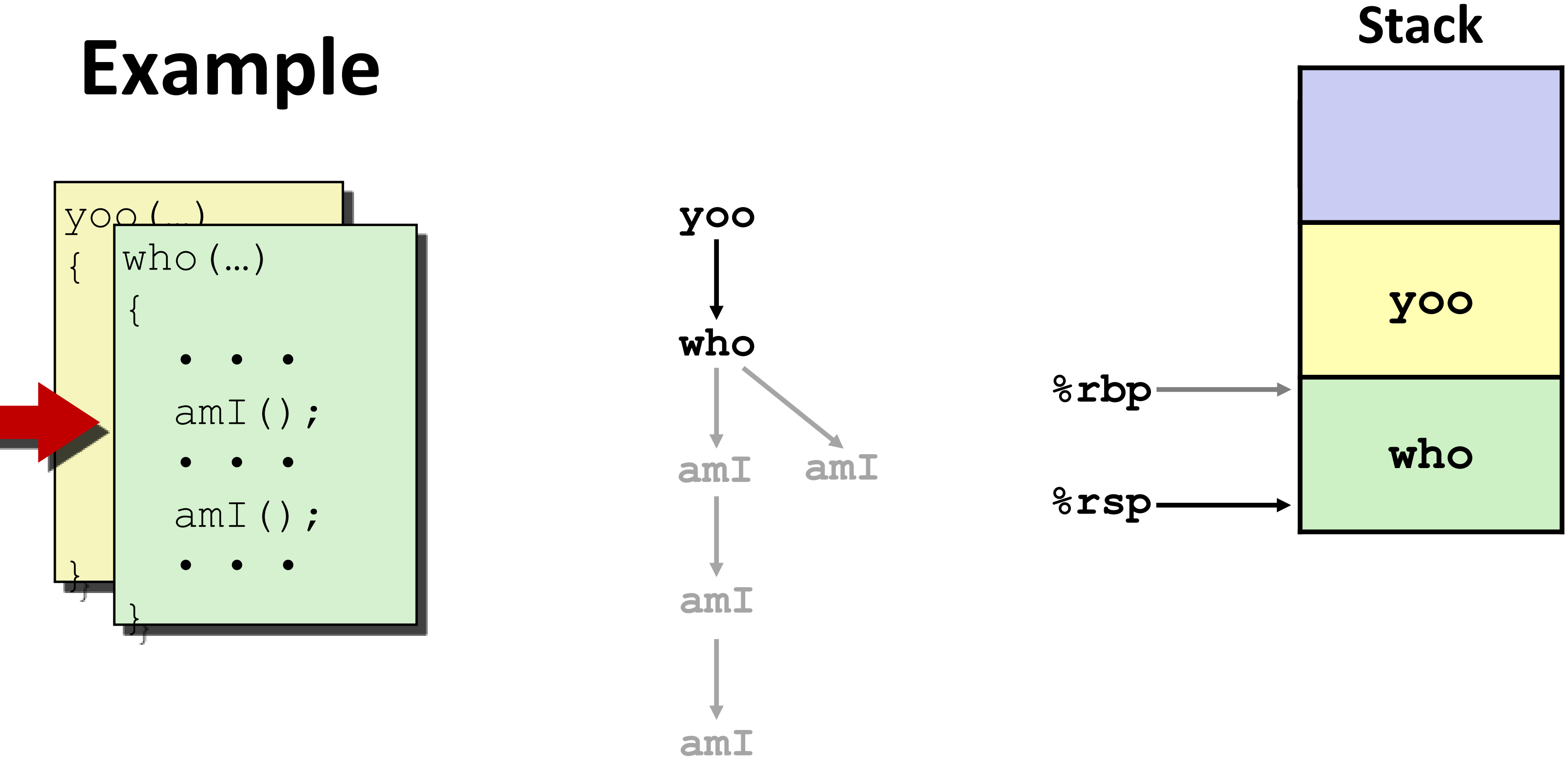
Example


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


yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

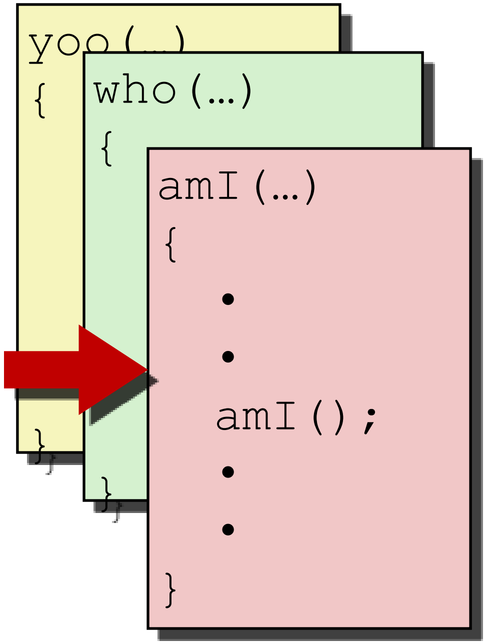
Example



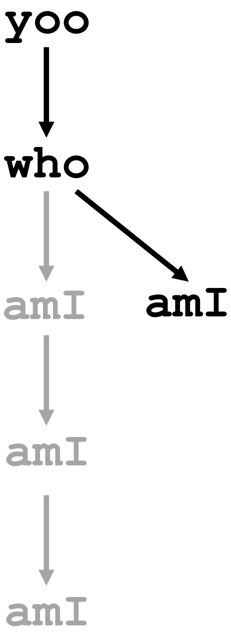
```
yoo(…)
{
```

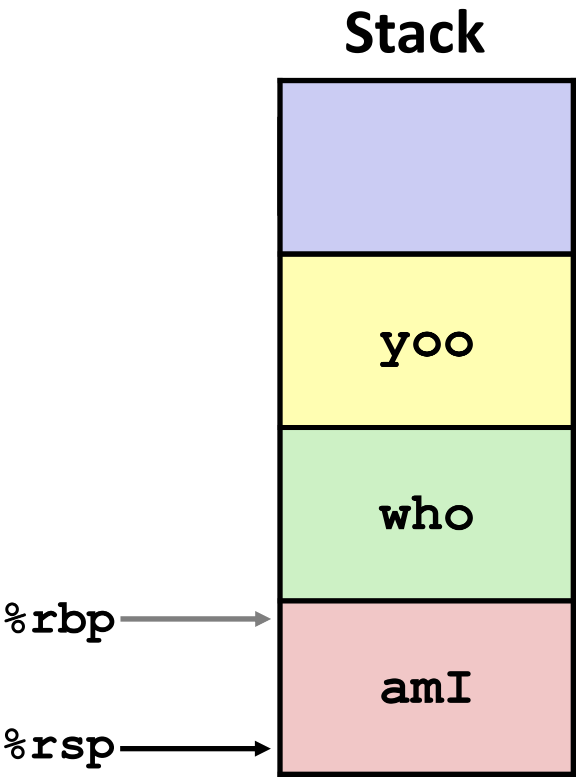
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Example

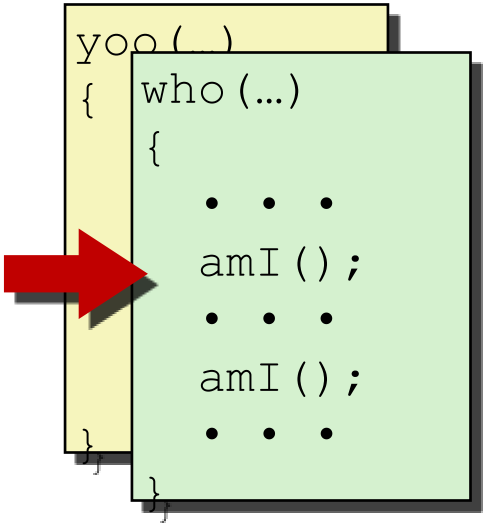


```
amI(…)
{
   •
   •
  amI();
   •
   •
}
```



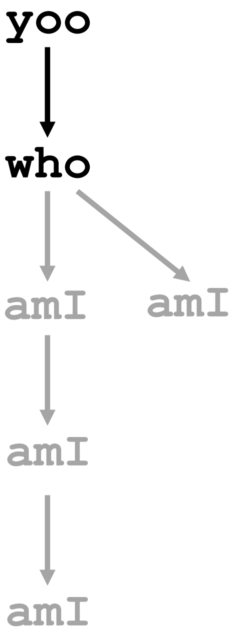


Example

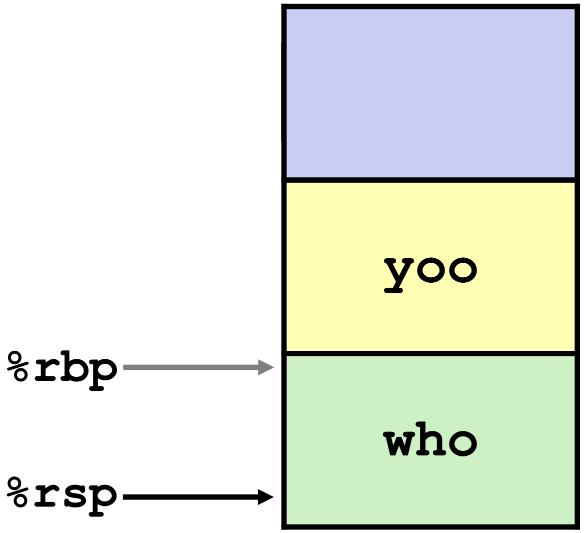


```
yoo(…)
{
   •
   •
}
```

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

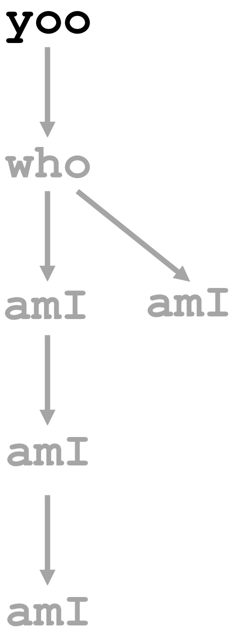


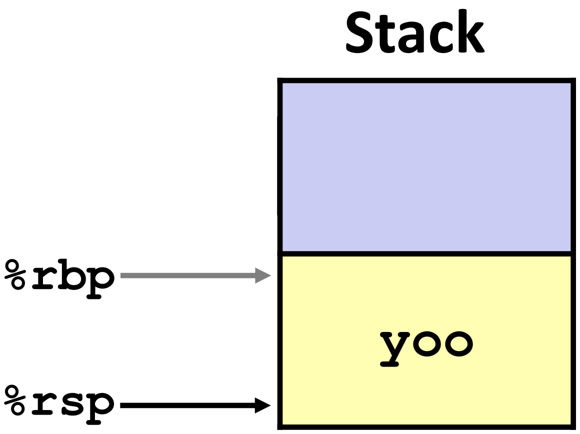
Stack



Example

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

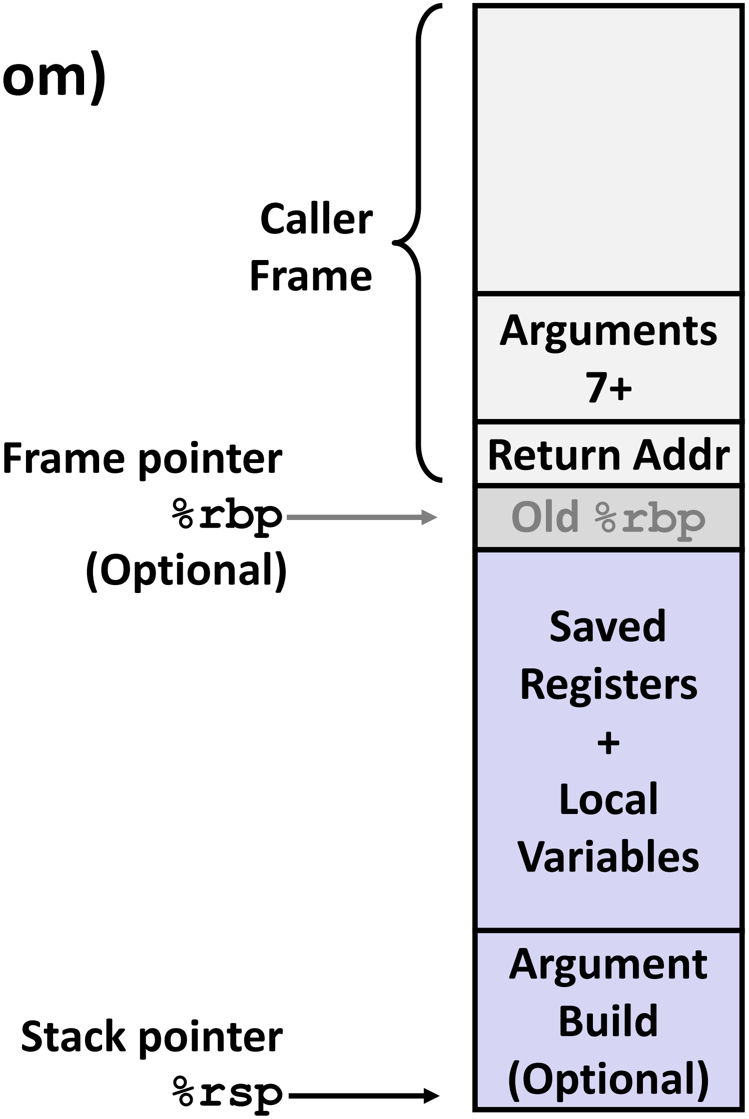


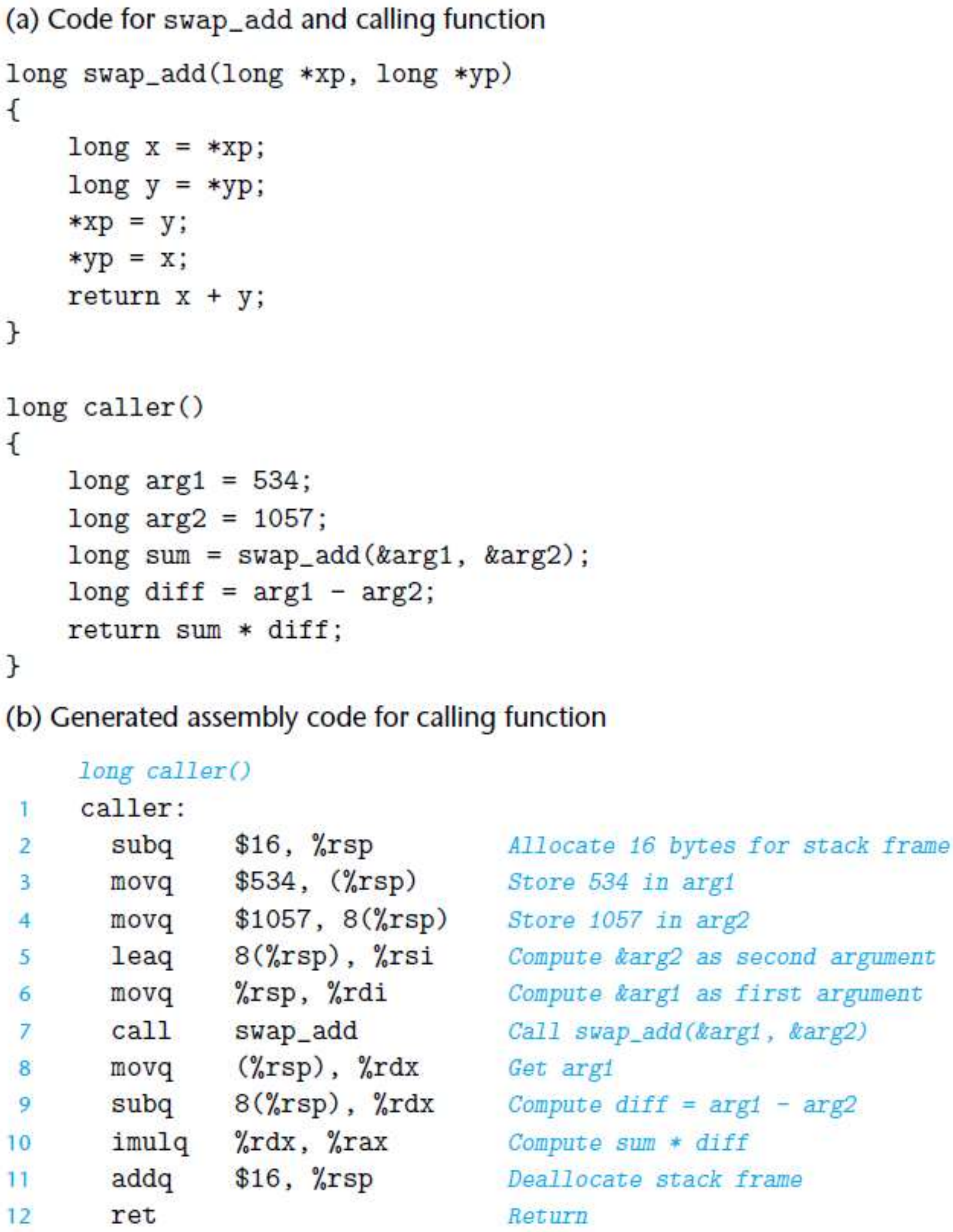


x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call





(a) Code for swap_add and calling function

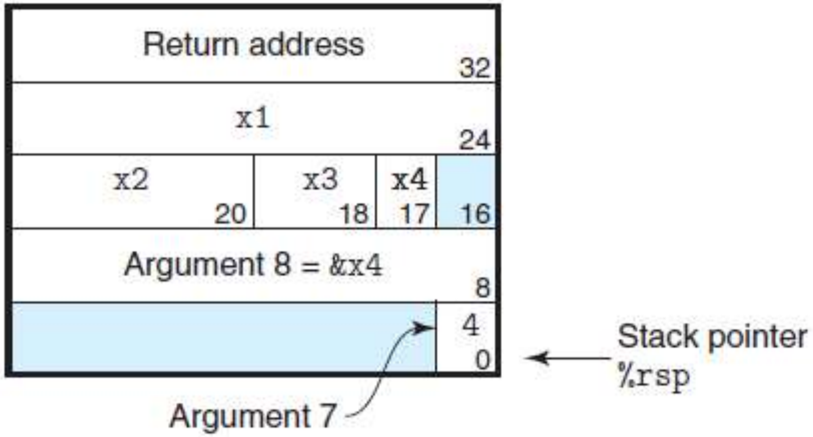
```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

(b) Generated assembly code for calling function

```
long caller()
caller:
  subq    $16, %rsp           Allocate 16 bytes for stack frame
  movq    $534, (%rsp)        Store 534 in arg1
  movq    $1057, 8(%rsp)      Store 1057 in arg2
  leaq    8(%rsp), %rsi       Compute &arg2 as second argument
  movq    %rsp, %rdi          Compute &arg1 as first argument
  call    swap_add            Call swap_add(&arg1, &arg2)
  movq    (%rsp), %rdx        Get arg1
  subq    8(%rsp), %rdx       Compute diff = arg1 - arg2
  imulq   %rdx, %rax          Compute sum * diff
  addq    $16, %rsp           Deallocate stack frame
  ret                         Return
```

(a) C code for calling function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```



```
long call_proc()
call_proc:
  Set up arguments to proc
  subq     $32, %rsp           Allocate 32-byte stack frame
  movq     $1, 24(%rsp)        Store 1 in &x1
  movl     $2, 20(%rsp)        Store 2 in &x2
  movw     $3, 18(%rsp)        Store 3 in &x3
  movb     $4, 17(%rsp)        Store 4 in &x4
  leaq     17(%rsp), %rax      Create &x4
  movq     %rax, 8(%rsp)       Store &x4 as argument 8
  movl     $4, (%rsp)          Store 4 as argument 7
  leaq     18(%rsp), %r9       Pass &x3 as argument 6
  movl     $3, %r8d            Pass 3 as argument 5
  leaq     20(%rsp), %rcx      Pass &x2 as argument 4
  movl     $2, %edx            Pass 2 as argument 3
  leaq     24(%rsp), %rsi      Pass &x1 as argument 2
  movl     $1, %edi            Pass 1 as argument 1
  Call proc
  call     proc
  Retrieve changes to memory
  movslq   20(%rsp), %rdx      Get x2 and convert to long
  addq     24(%rsp), %rdx      Compute x1+x2
  movswl   18(%rsp), %eax      Get x3 and convert to int
  movsbl   17(%rsp), %ecx      Get x4 and convert to int
  subl     %ecx, %eax          Compute x3-x4
  cltq                         Convert to long
  imulq    %rdx, %rax          Compute (x1+x2) * (x3-x4)
  addq     $32, %rsp           Deallocate stack frame
  ret                          Return
```

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- **Can register be used for temporary storage?**

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

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

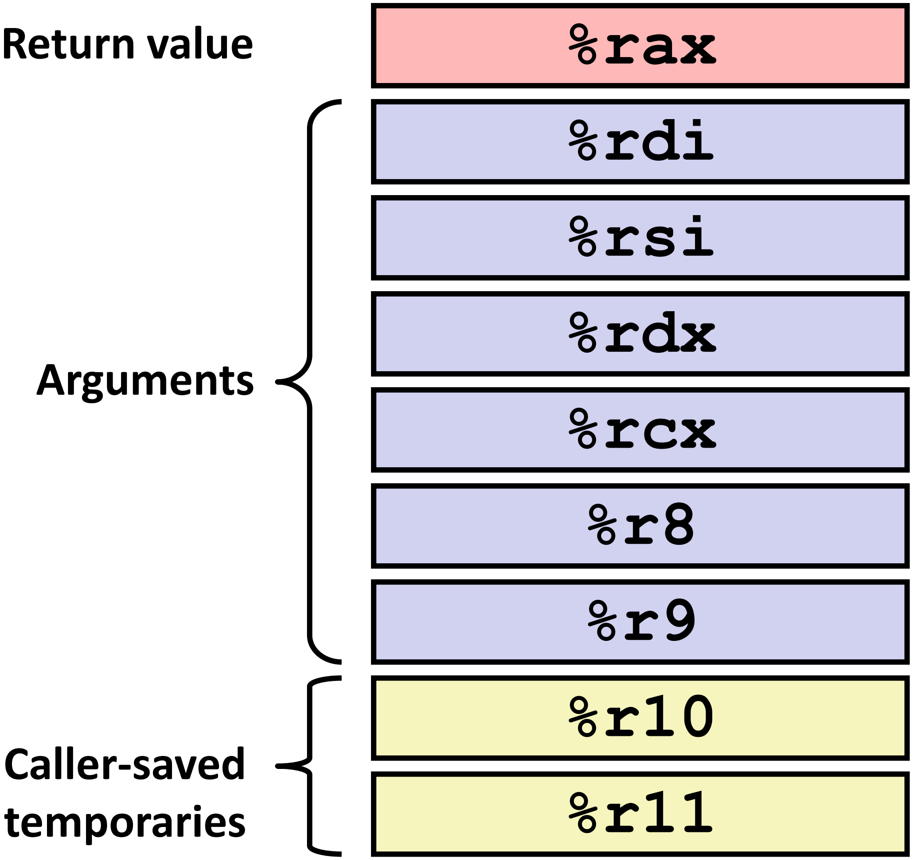
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- **Can register be used for temporary storage?**
- **Conventions**
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

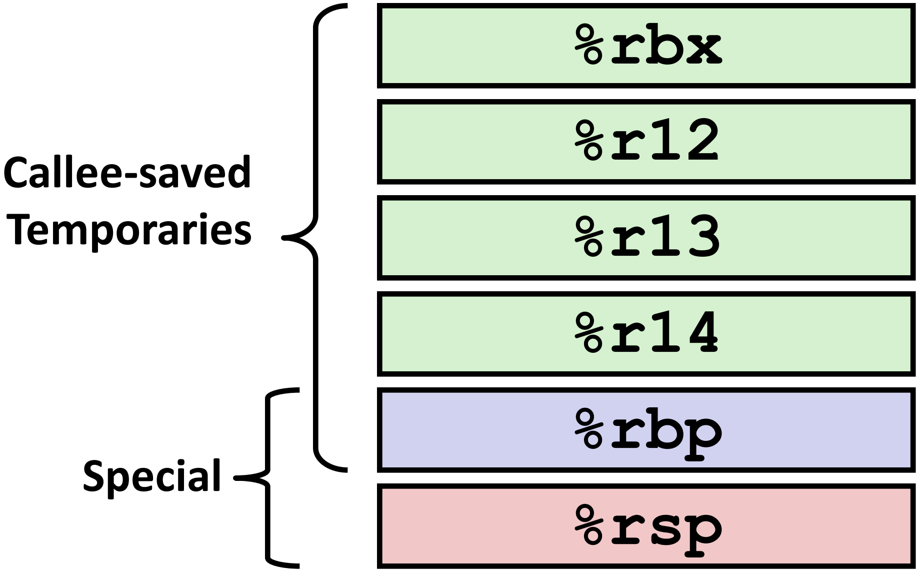
x86-64 Linux Register Usage #1

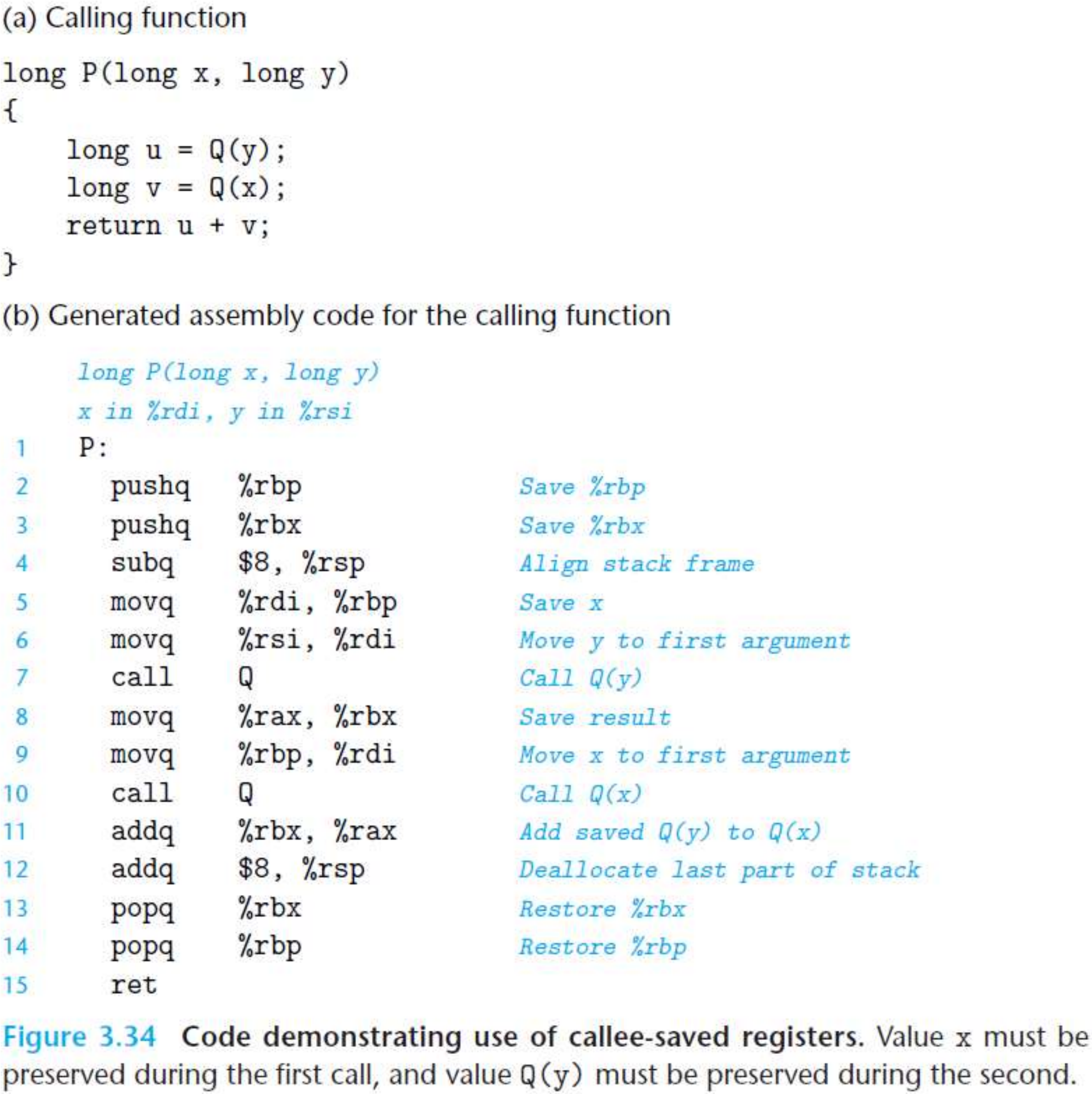
- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, …, `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure





(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Generated assembly code for the calling function

```
    long P(long x, long y)
    x in %rdi, y in %rsi
    P:
      pushq   %rbp              Save %rbp
      pushq   %rbx              Save %rbx
      subq    $8, %rsp          Align stack frame
      movq    %rdi, %rbp        Save x
      movq    %rsi, %rdi        Move y to first argument
      call    Q                 Call Q(y)
      movq    %rax, %rbx        Save result
      movq    %rbp, %rdi        Move x to first argument
      call    Q                 Call Q(x)
      addq    %rbx, %rax        Add saved Q(y) to Q(x)
      addq    $8, %rsp          Deallocate last part of stack
      popq    %rbx              Restore %rbx
      popq    %rbp              Restore %rbp
      ret
```

Figure 3.34 **Code demonstrating use of callee-saved registers.** Value x must be preserved during the first call, and value Q(y) must be preserved during the second.

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - **Illustration of Recursion**

Recursive Function

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

Recursive Function Terminal Case

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

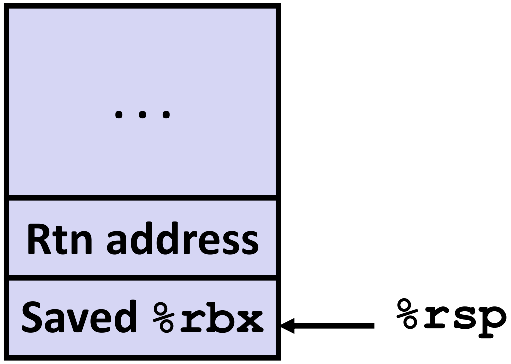
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

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

Register	Use(s)	Type
%rdi	x	Argument



. . .

Rtn address

Saved %rbx

%rsp

Recursive Function Call Setup

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

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

Recursive Function Call

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

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

Recursive Function Result

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

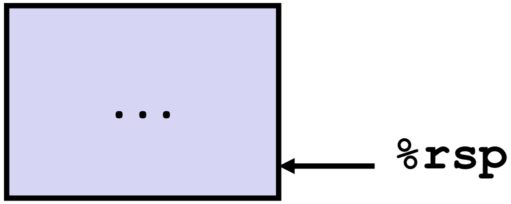
Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Return value	

Recursive Function Completion

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

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

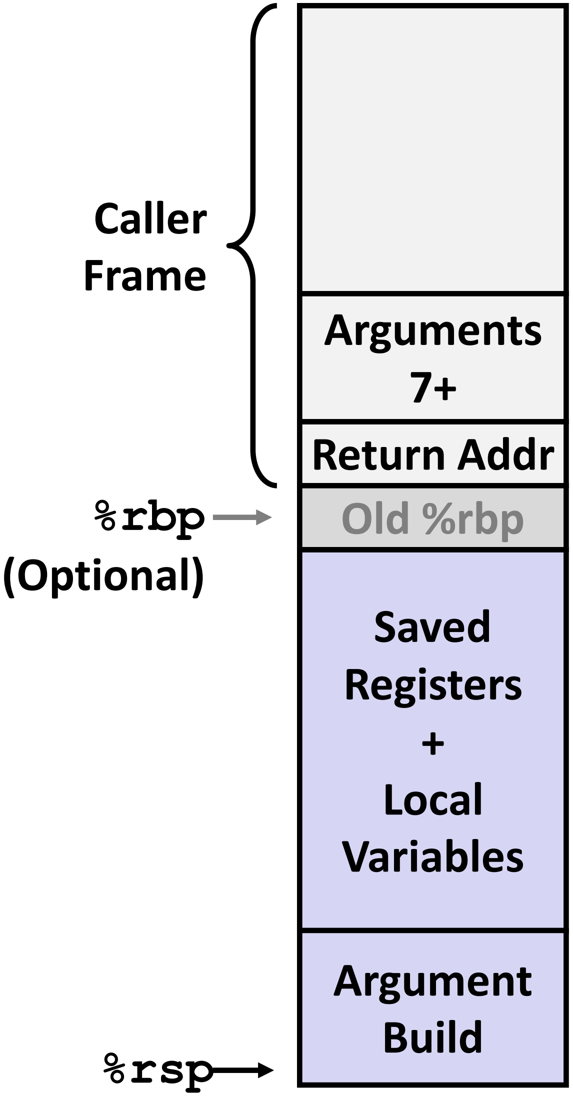


Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the C code explicitly does
    - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- **Pointers are addresses of values**
  - **On stack or global**


Caller Frame
Arguments 7+
Return Addr
%rbp
(Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build
%rsp