

# Machine-Level Programming III: Procedures

Kai Zhang  
Fudan University  
[zhangk@fudan.edu.cn](mailto:zhangk@fudan.edu.cn)

# Review

$$\begin{array}{r}
 1100 \\
 + 1100 \\
 \hline
 1 \quad 1000
 \end{array}$$

$$ZF = 0$$

$$SF = 1$$

$$CF = 1$$

$$\begin{array}{r}
 1xxxxxxx... \\
 + 1xxxxxxx... \\
 \hline
 1 \quad xxxxxx... \\
 \\
 1 \quad 0xxxxxxx... \\
 - 1xxxxxxx... \\
 \hline
 1xxxxxxx...
 \end{array}$$

Carry

Borrow

For **unsigned** arithmetic, this reports **overflow**

$$\begin{array}{r}
 12 \\
 + 12 \\
 \hline
 8
 \end{array}$$

$$OF = 0$$

$$\begin{array}{r}
 yxxxxxxx... \\
 + yxxxxxxx... \\
 \hline
 zxxxxxxx...
 \end{array}$$

a

b

t

$$Z = \sim y$$

$$\begin{array}{r}
 -4 \\
 + -4 \\
 \hline
 -8
 \end{array}$$

For **signed** arithmetic, this reports **overflow**

# Jump Table

## 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
```

.L4 = → 0x40010

0x40010

0x40014

0x40018

0x4001C

0x40020

0x40024

0x40028

.L8
.L3
.L5
.L9
.L8
.L7
.L7

```
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;
}
```

# Finding Jump Table in Binary

```
00000000004005e0 <switch_eg>:
4005e0:    48 89 d1                mov     %rdx,%rcx
4005e3:    48 83 ff 06            cmp     $0x6,%rdi
4005e7:    77 2b                  ja      400614 <switch_eg+0x34>
4005e9:    ff 24 fd f0 07 40 00   jmpq    *0x4007f0(,%rdi,8)
4005f0:    48 89 f0                mov     %rsi,%rax
4005f3:    48 0f af c2            imul    %rdx,%rax
4005f7:    c3                     retq
4005f8:    48 89 f0                mov     %rsi,%rax
4005fb:    48 99                  cqto
4005fd:    48 f7 f9                idiv    %rcx
400600:    eb 05                  jmp     400607 <switch_eg+0x27>
400602:    b8 01 00 00 00        mov     $0x1,%eax
400607:    48 01 c8                add     %rcx,%rax
40060a:    c3                     retq
40060b:    b8 01 00 00 00        mov     $0x1,%eax
400610:    48 29 d0                sub     %rdx,%rax
400613:    c3                     retq
400614:    b8 02 00 00 00        mov     $0x2,%eax
400619:    c3                     retq
```

# Finding Jump Table in Binary (cont.)

```
00000000004005e0 <switch_eg>:
```

```
. . .
```

```
4005e9:      ff 24 fd f0 07 40 00      jmpq    *0x4007f0(,%rdi,8)
```

```
. . .
```

```
% gdb switch
```

```
(gdb) x /8xg 0x4007f0
```

0x4007f0:	0x0000000000400614	0x00000000004005f0
0x400800:	0x00000000004005f8	0x0000000000400602
0x400810:	0x0000000000400614	0x000000000040060b
0x400820:	0x000000000040060b	0x2c646c25203d2078

```
(gdb)
```

# Finding Jump Table in Binary (cont.)

```
% gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:      0x000000000000400614      0x0000000000004005f0
0x400800:      0x0000000000004005f8      0x000000000000400602
0x400810:      0x000000000000400614      0x00000000000040060b
0x400820:      0x00000000000040060b      0x2c646c25203d2078
```

4005f0:	48 89 f0	mov %rsi,%rax
4005f3:	48 0f af c2	imul %rdx,%rax
4005f7:	c3	retq
4005f8:	48 89 f0	mov %rsi,%rax
4005fb:	48 99	cqto
4005fd:	48 f7 f9	idiv %rcx
400600:	eb 05	jmp 400607 <switch_eg+0x27>
400602:	b8 01 00 00 00	mov \$0x1,%eax
400607:	48 01 c8	add %rcx,%rax
40060a:	c3	retq
40060b:	b8 01 00 00 00	mov \$0x1,%eax
400610:	48 29 d0	sub %rdx,%rax
400613:	c3	retq
400614:	b8 02 00 00 00	mov \$0x2,%eax
400619:	c3	retq

# Today

## ■ Procedures

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

# 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];  
}
```



# Mechanisms in Procedures

## ■ Passing **control**

- To beginning of procedure code
- Back to return point

## ■ Passing **data**

- Procedure arguments
- Return value

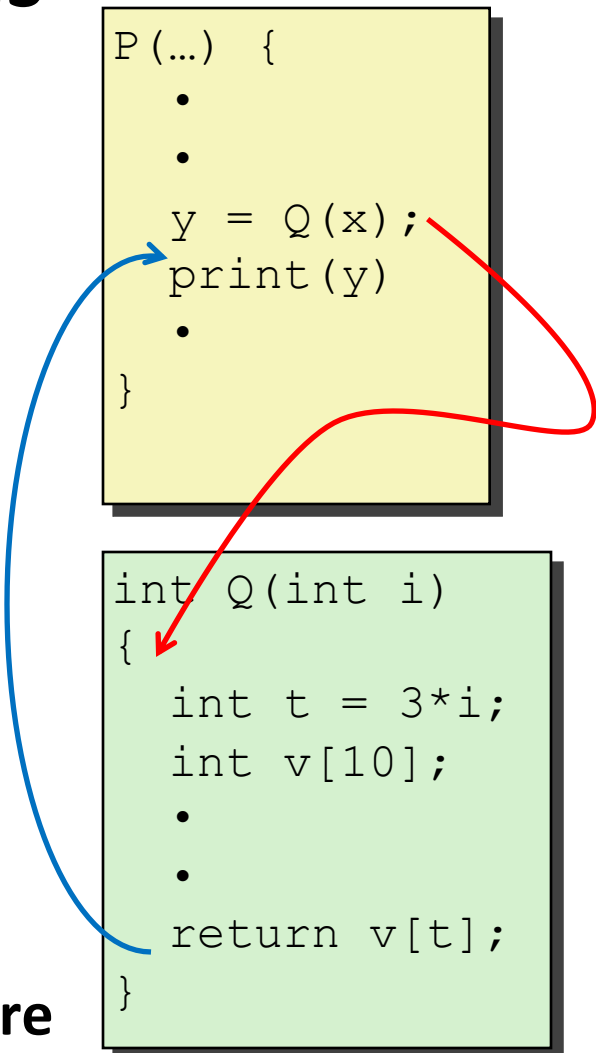
## ■ **Memory** management

- 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];  
}
```

# Mechanisms in Procedures

## ■ Passing **control**

- To beginning of procedure code
- Back to return point

## ■ Passing **data**

- Procedure arguments
- **Return value**

## ■ **Memory** management

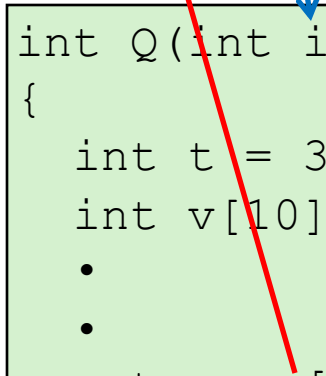
- Allocate during procedure execution
- Deallocate upon return

## ■ 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];  
}
```



# 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];  
}
```

# Mechanisms in Procedures

## ■ Passing **control**

- To beginning of procedure code

```
P (...) {  
    •  
    •  
    v = O(x);  
}
```

## Application Binary Interface (ABI)

defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format

[https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions)

- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
·  
·  
return v[t];  
}
```

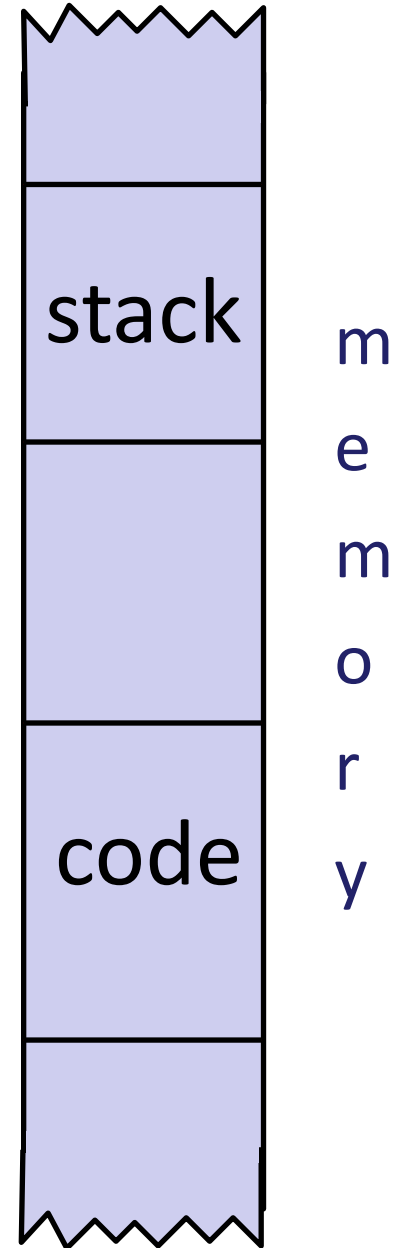
# Today

## ■ Procedures

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

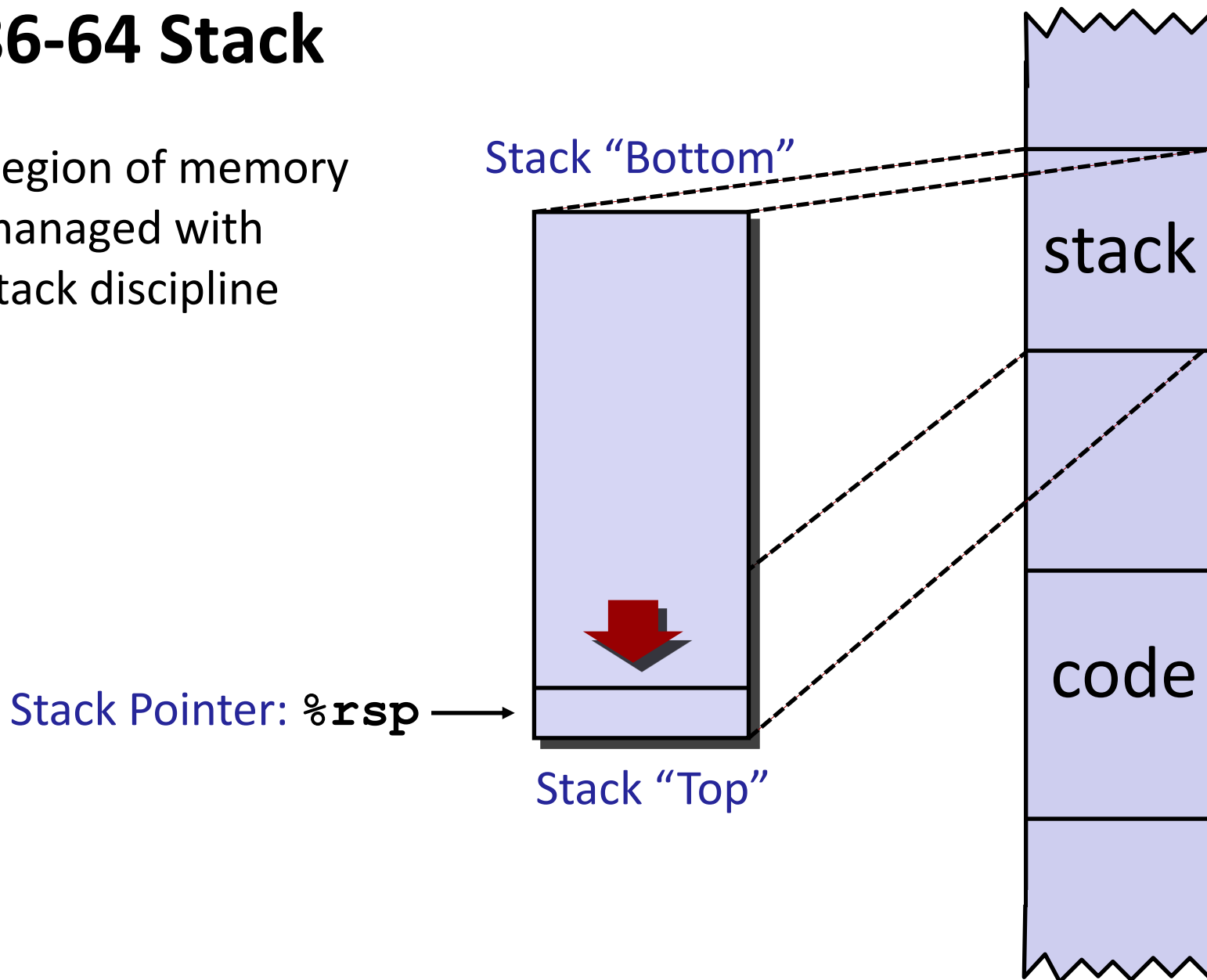
# x86-64 Stack

- Region of memory managed with stack discipline
  - Memory viewed as [array of bytes](#).
  - Different regions have different purposes.
  - (Like ABI, a policy decision)



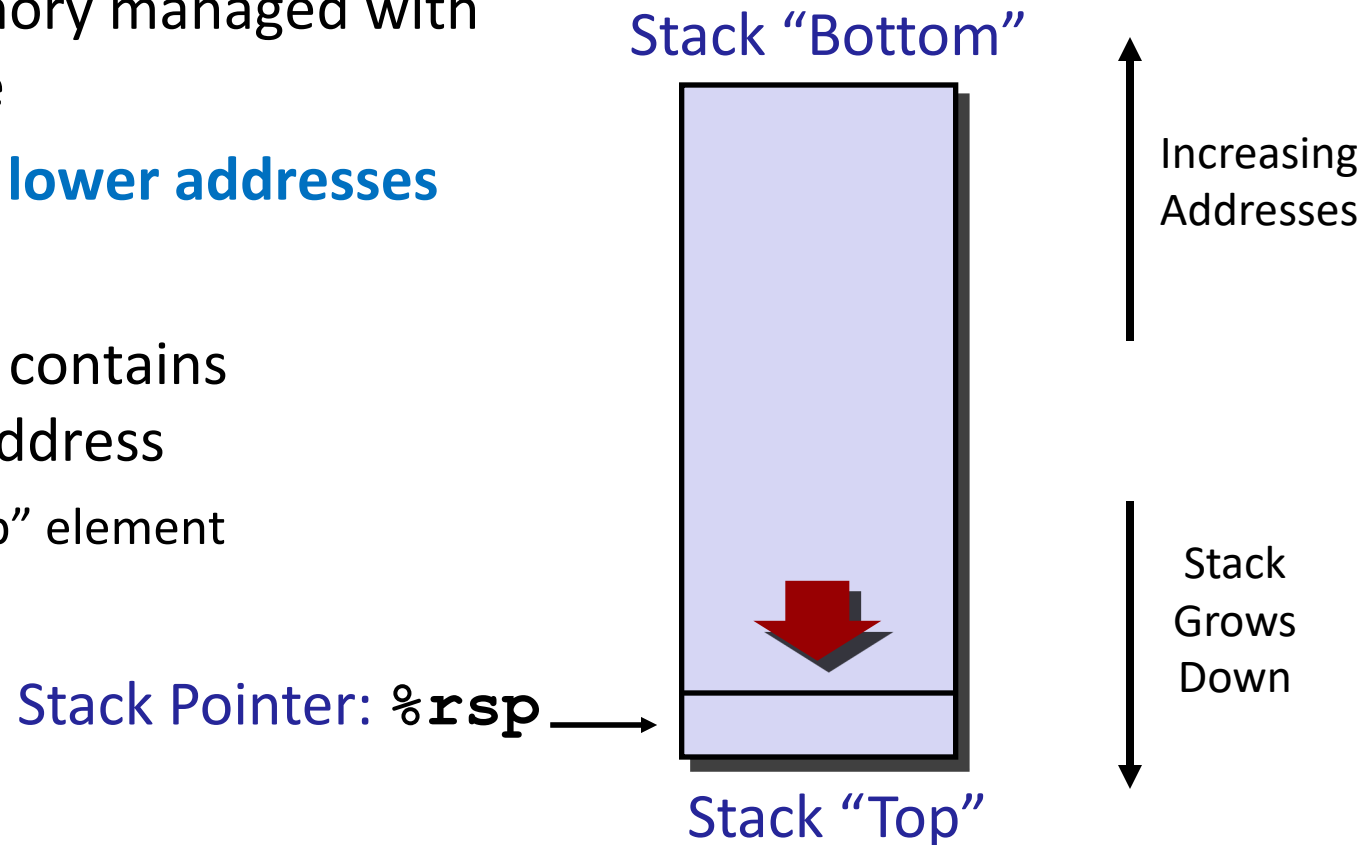
# x86-64 Stack

- Region of memory managed with stack discipline



# x86-64 Stack

- Region of memory managed with stack discipline
- **Grows toward lower addresses**
- Register **%rsp** contains lowest stack address
  - address of “top” element

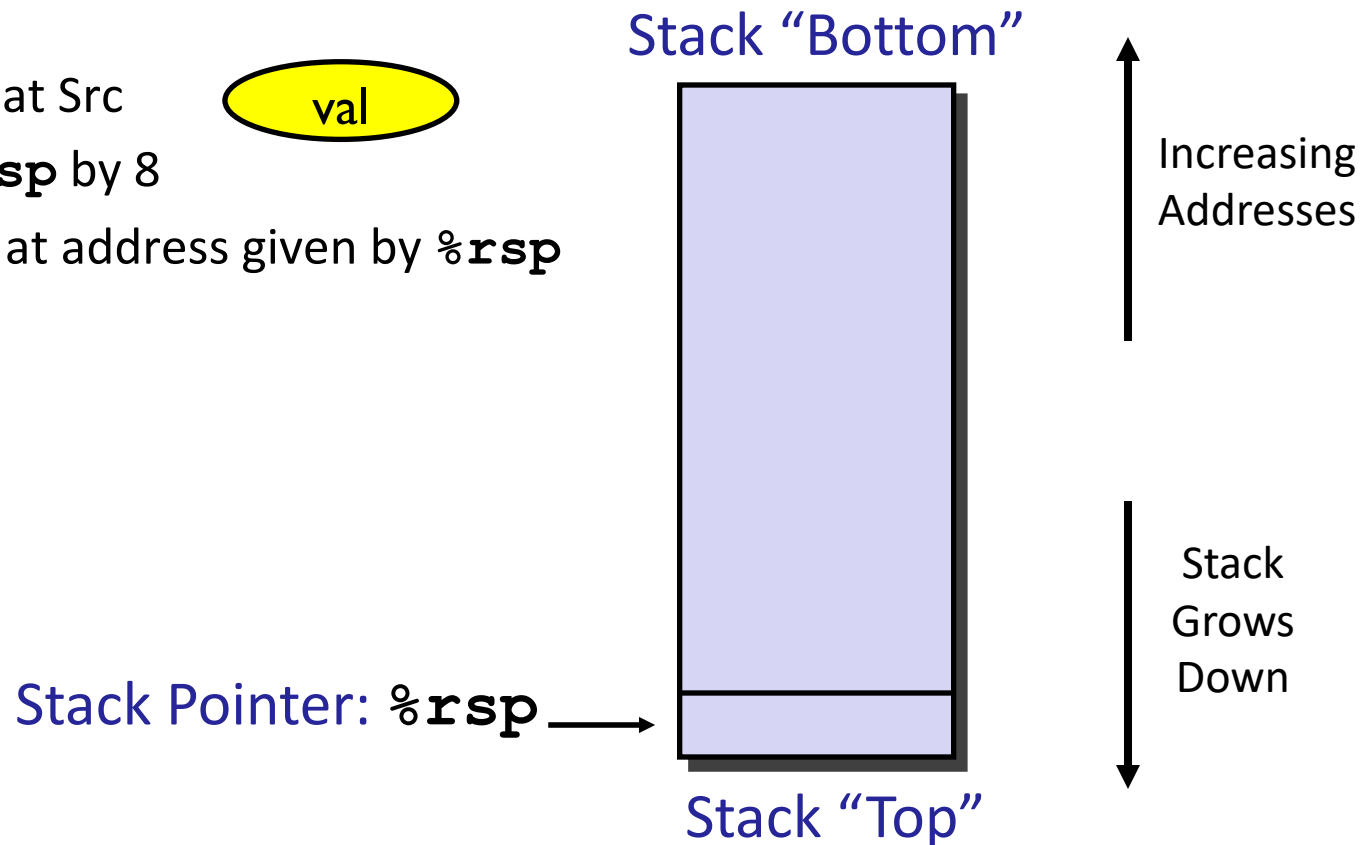




# x86-64 Stack: Push

## ■ `pushq Src`

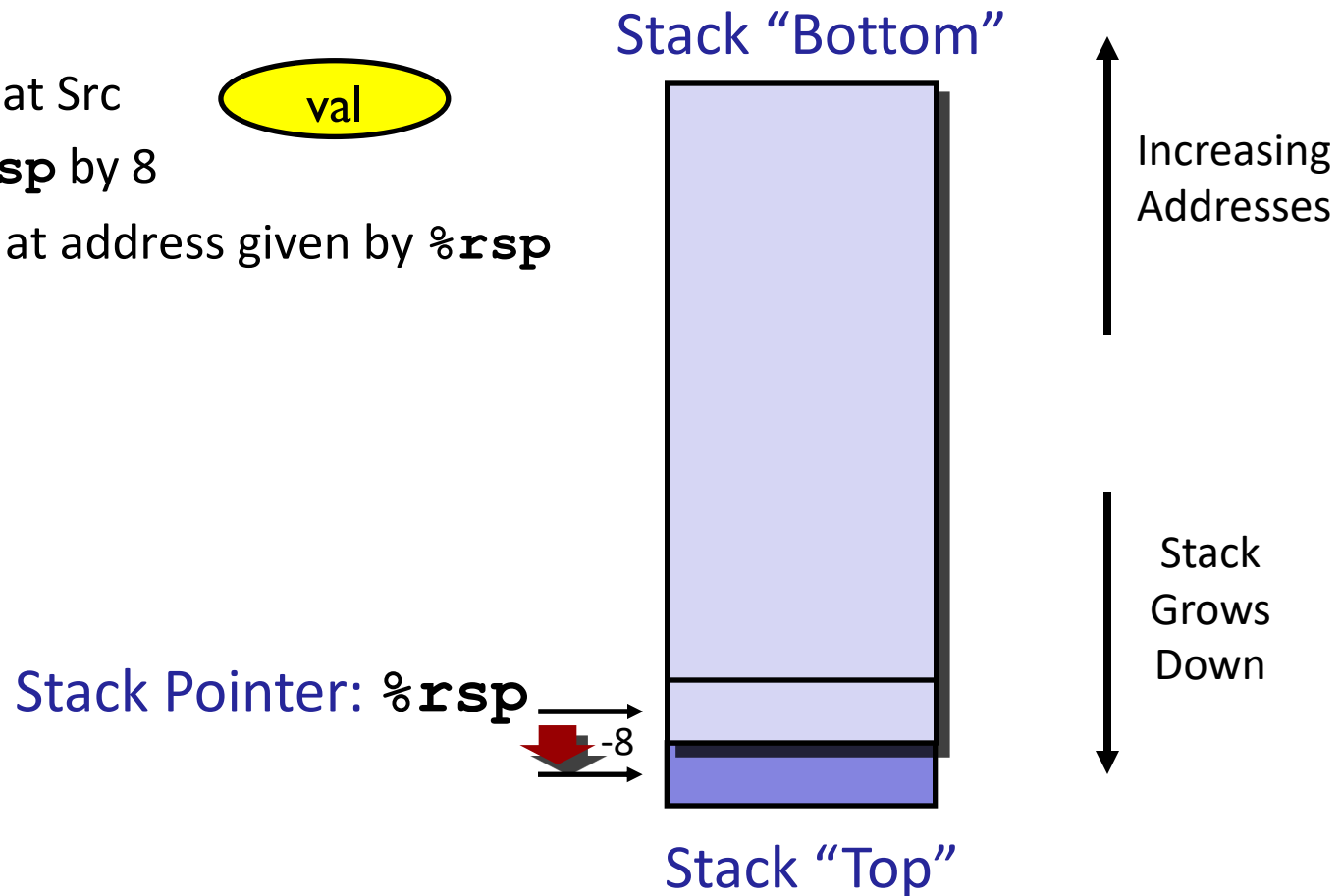
- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



# x86-64 Stack: Push

## ■ `pushq Src`

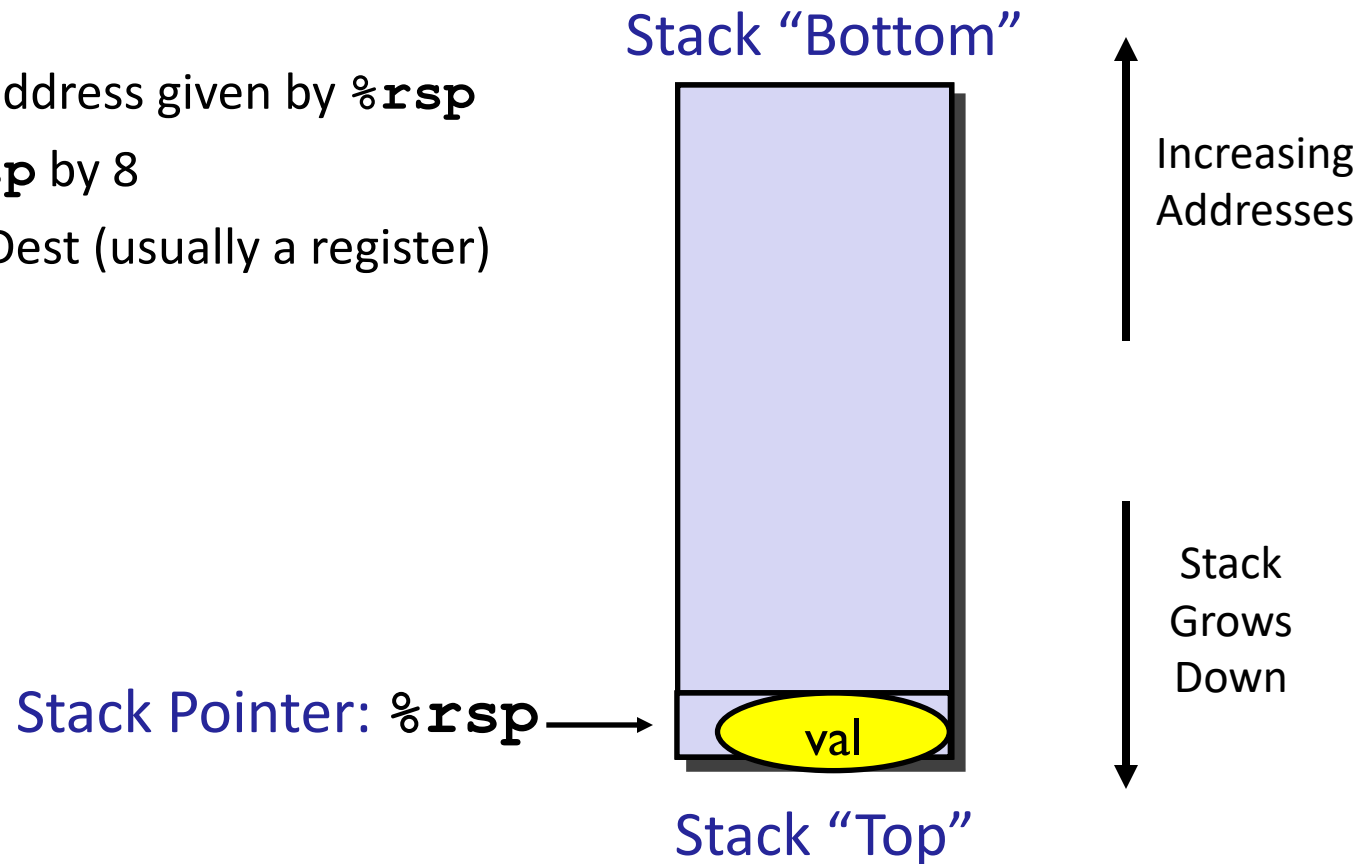
- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



# x86-64 Stack: Pop

## ■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at `Dest` (usually a register)



# x86-64 Stack: Pop

## ■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at `Dest` (usually a register)

val

Stack Pointer: `%rsp`



Stack "Bottom"



Stack "Top"

Increasing  
Addresses

Stack  
Grows  
Down

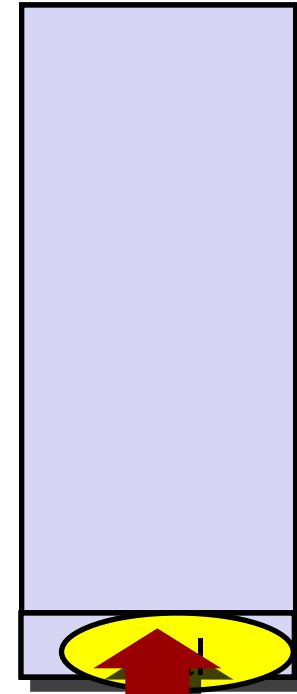
# x86-64 Stack: Pop

## ■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at `Dest` (usually a register)

Stack Pointer: `%rsp` →

Stack “Bottom”



Increasing  
Addresses

Stack  
Grows  
Down

Stack “Top”

(The memory doesn't change,  
only the value of `%rsp`)

# Today

## ■ Procedures

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

# Code Examples

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

```
00000000000400540 <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;
}
```

```
00000000000400550 <mult2>:
400550: mov     %rdi,%rax           # a
400553: imul    %rsi,%rax           # a * b
400557: retq                      # Return
```

# Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the `next instruction` right after call
  - Example from disassembly
- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address



# Procedure Control Flow

## ■ Program Counter (PC)

- Commonly called Instruction Counter (IP)
- Sometimes called Instruction Address Register (IAR), Instruction Counter

## ■ PC controls execution

- PC is increased after fetching an instruction
- Different instructions have different sizes ( $PC+=1/2/4$ )
- Holds the memory address of the next instruction would be executed
- `jmp/call/ret` changes PC in control transfer

## ■ In X86-64: `%rip`

# Control Flow Example #1

```
0000000000400540 <multstore>:
```

•

•

```
400544: callq 400550 <mult2>
```

```
400549: mov    %rax, (%rbx)
```

•

•

```
0000000000400550 <mult2>:
```

```
400550: mov    %rdi, %rax
```

•

•

```
400557: retq
```

0x130

0x128

0x120

%rsp

%rip

0x120

0x400544

# Control Flow Example #2

```
00000000000400540 <multstore>:
```

•

•

```
400544: callq 400550 <mult2>
```

```
400549: mov    %rax, (%rbx) ←
```

•

•

```
00000000000400550 <mult2>:
```

```
400550: mov    %rdi, %rax ←
```

•

•

```
400557: retq
```

0x130

0x128

0x120

0x118

%rsp

%rip

0x400549

0x118

0x400550

**callq**: push **return address** to stack;  
**jump** to the **first instruction** of the function

# Control Flow Example #3

0000000000400540 <multstore>:

•

•

400544: callq 400550 <mult2>

400549: mov %rax, (%rbx) ←

•

•

0000000000400550 <mult2>:

400550: mov %rdi, %rax

•

•

400557: retq ←

0x130

0x128

0x120

0x118

0x400549

%rsp

0x118

%rip

0x400557

# Control Flow Example #4

```
00000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
.  
.
```

```
00000000000400550 <mult2>:  
400550: mov    %rdi,%rax  
.  
.  
400557: retq
```

0x130

0x128

0x120

%rsp

%rip

0x120

0x400549

# Quiz Time!

## Exercise 3.32

# Today

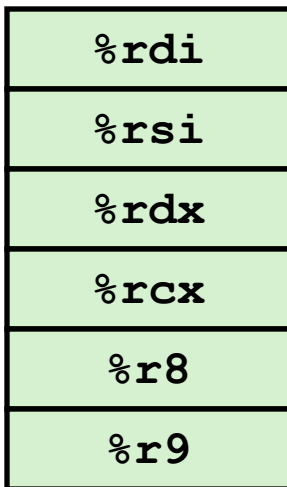
## ■ Procedures

- Mechanisms
- tack Structure
- Calling Conventions
  - Passing control
  - **Passing data**
  - Managing local data
- Illustrations of Recursion & Pointers

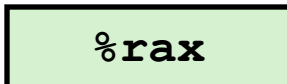
# Procedure Data Flow

## Registers

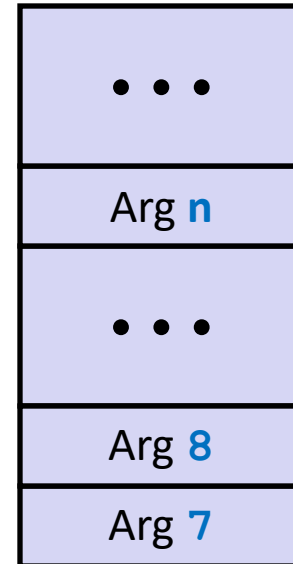
### ■ First 6 arguments



### ■ Return value



## Stack



### ■ Only allocate stack space when needed



# Data Flow Examples

```
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
```

# Quiz Time!

## Exercise 3.33

# Today

## ■ Procedures

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

# Stack-Based Languages

## ■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “Reentrant”
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

## ■ Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

## ■ Stack allocated in Frames (栈帧)

- state for single procedure instantiation

# Call Chain Example

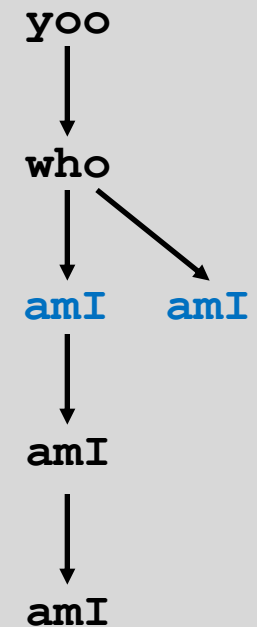
```
yoo (...)  
{  
  .  
  .  
  who () ;  
  .  
  .  
}
```

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

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

Procedure **amI ()** is recursive

Example  
Call Chain



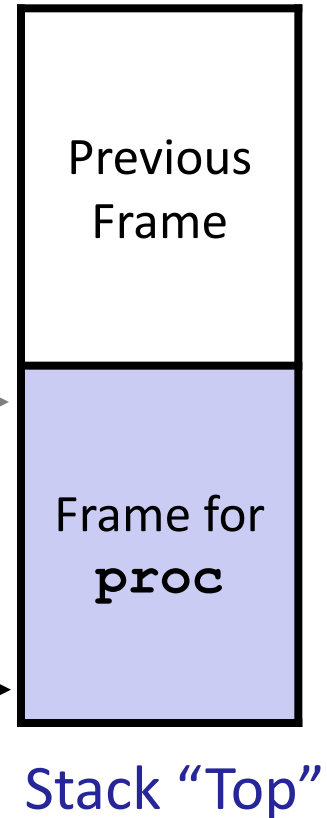
# Stack Frames

## ■ Contents

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

Frame Pointer: `%rbp`  
(Optional)


Stack Pointer: `%rsp`



## ■ Management

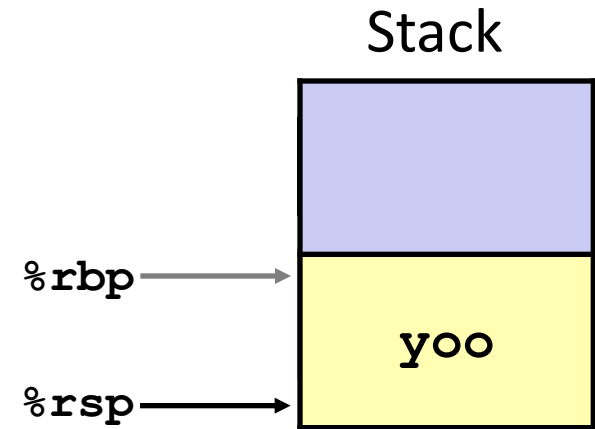
- Space **allocated when enter** procedure
  - “Set-up” code
  - Includes **push by `call`** instruction
- **Deallocated when return**
  - “Finish” code
  - Includes **pop by `ret`** instruction

# Example

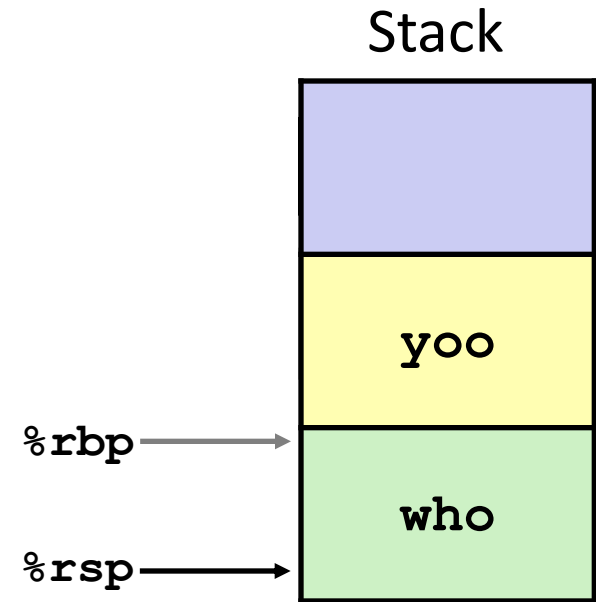
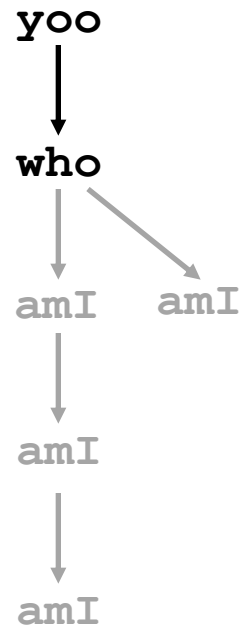
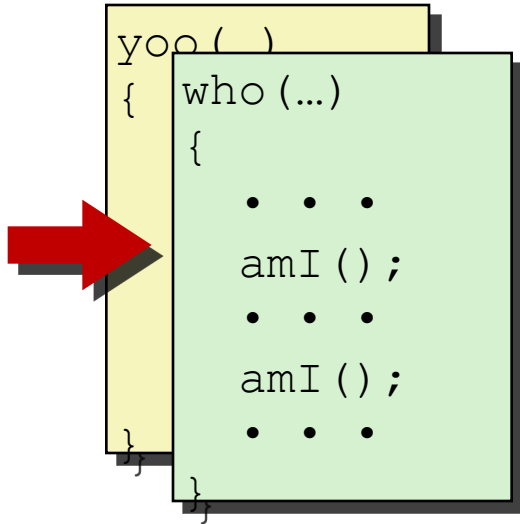


```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
yoo  
  ↓  
who  
  ↓  ↘  
amI  amI  
  ↓  
amI  
  ↓  
amI
```

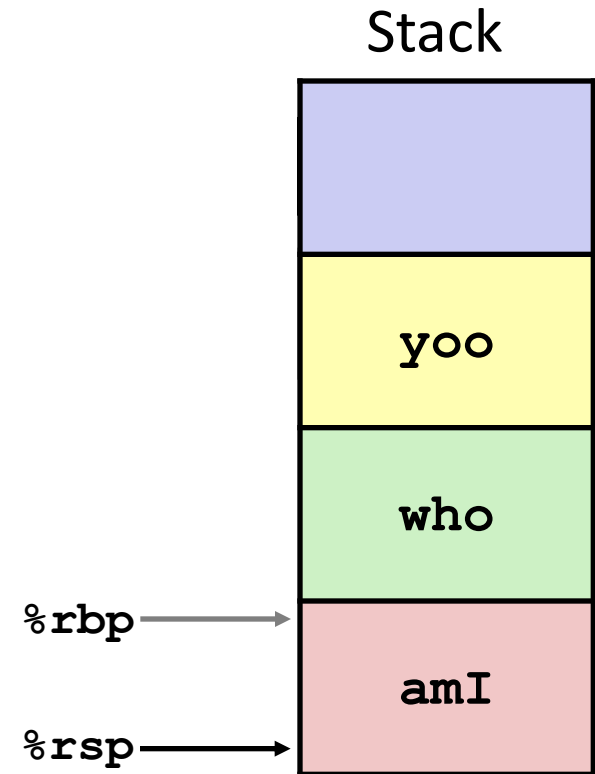
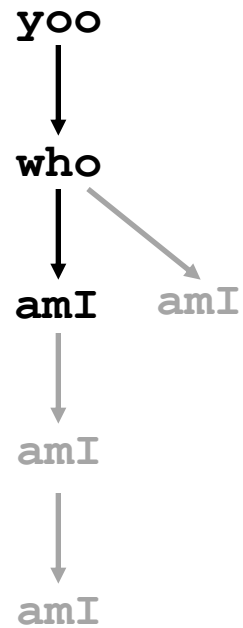
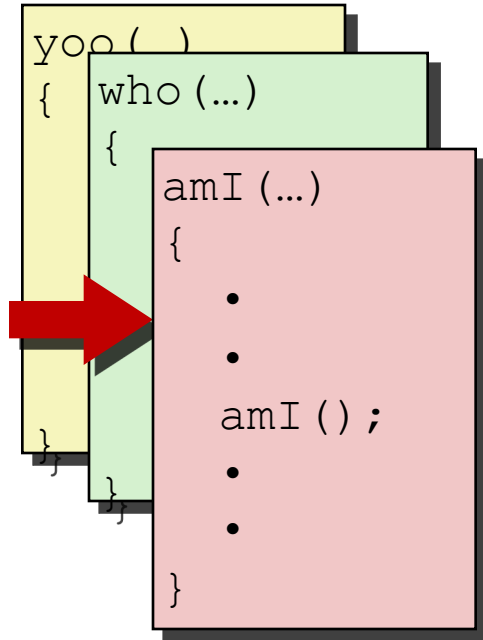


# Example

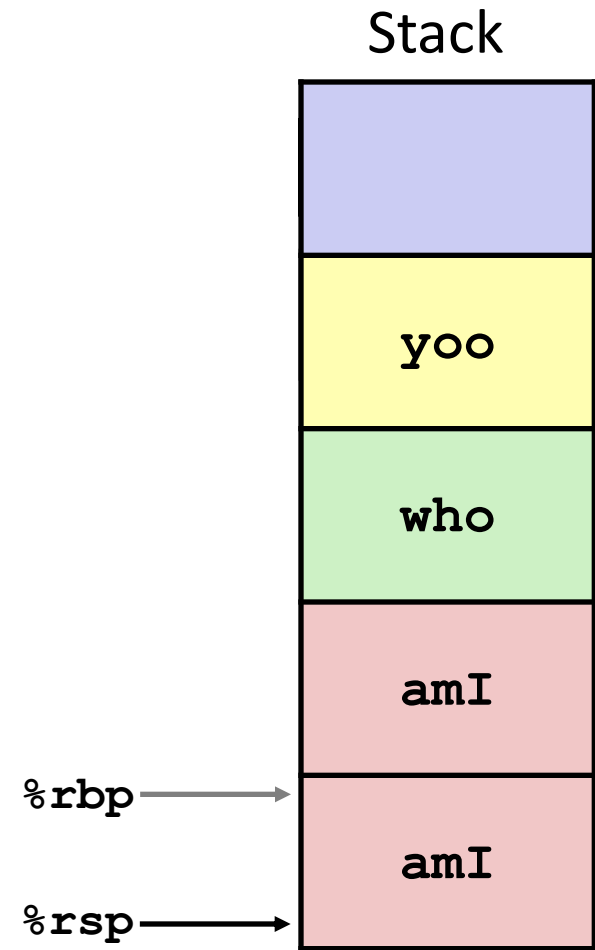
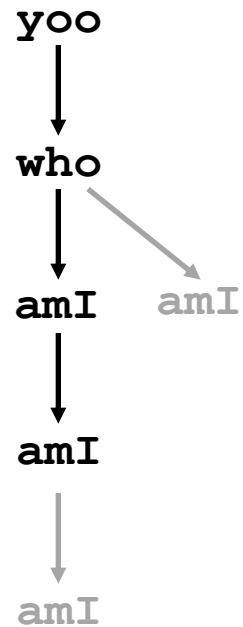
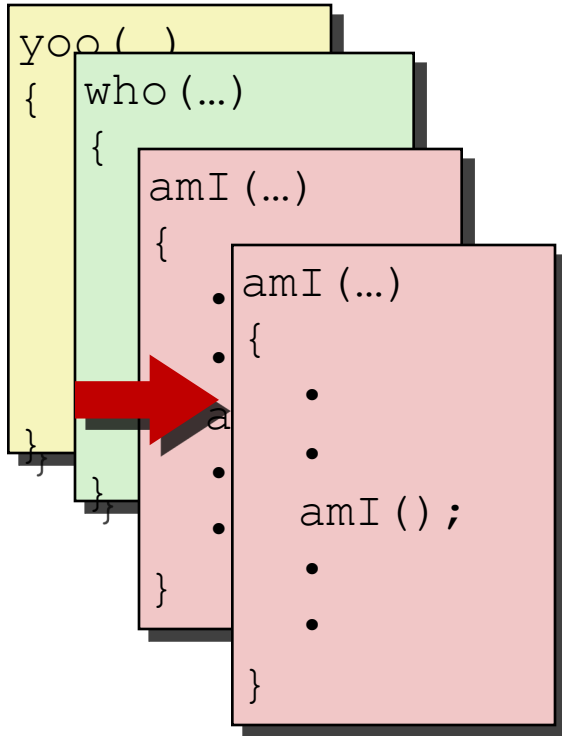




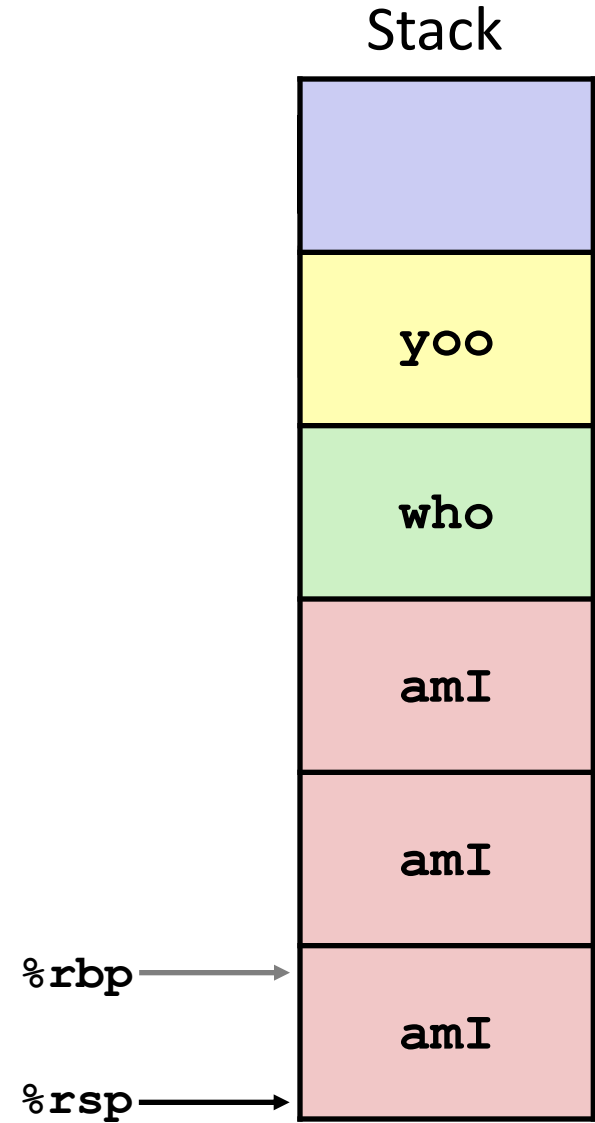
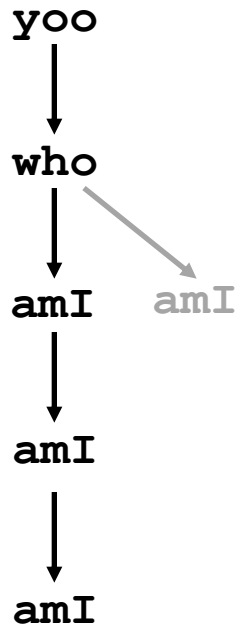
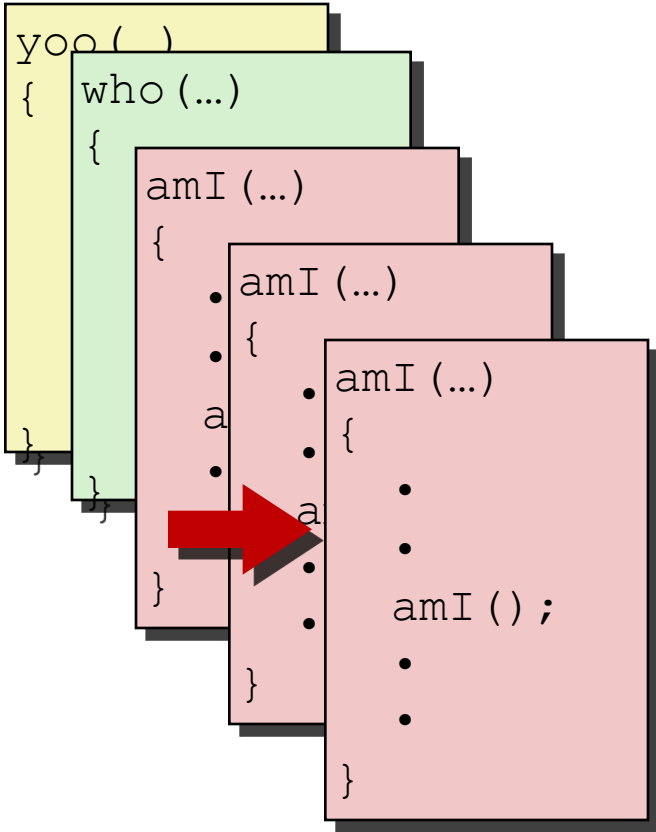
# Example



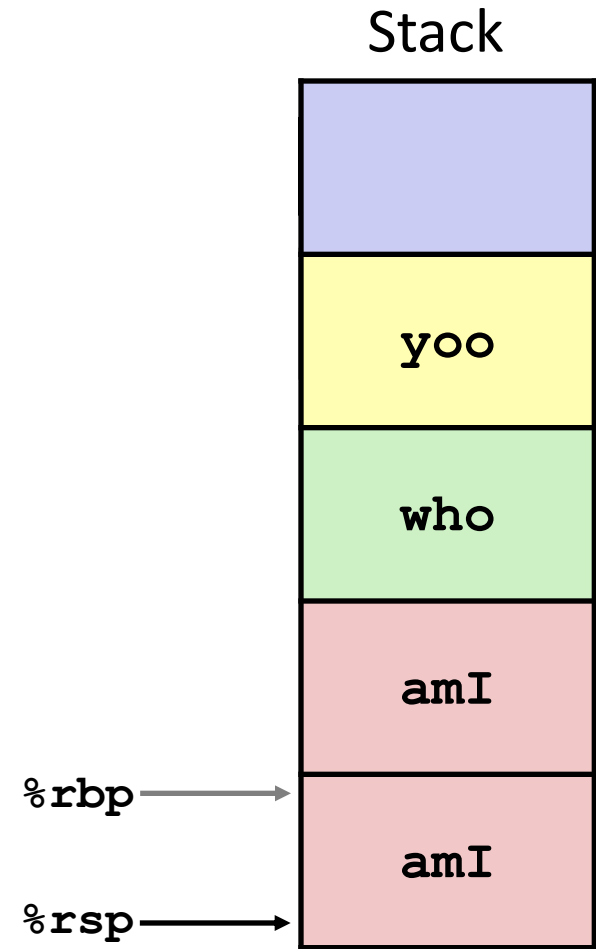
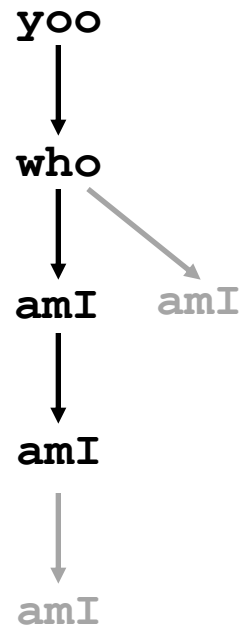
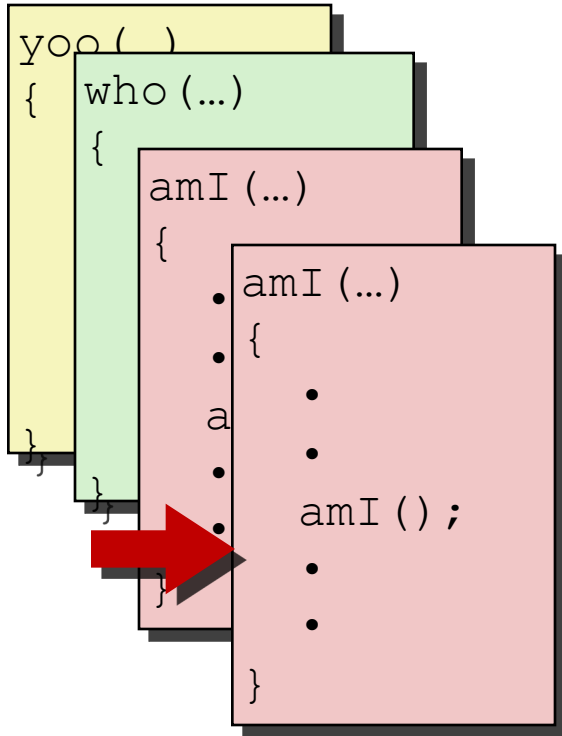
# Example



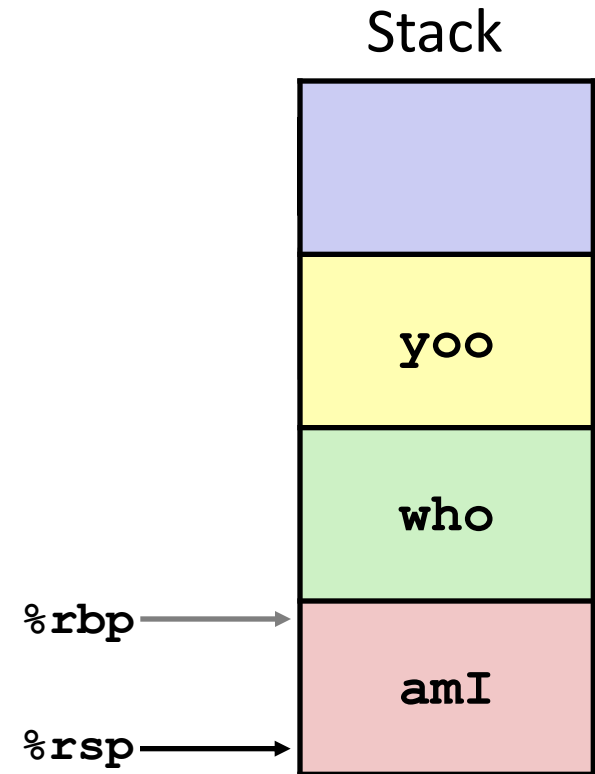
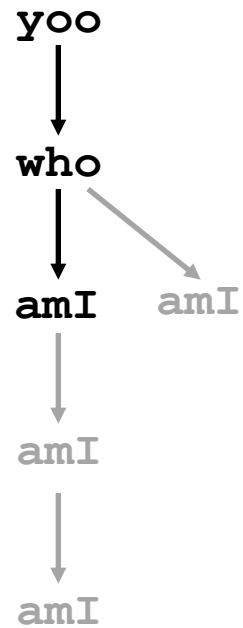
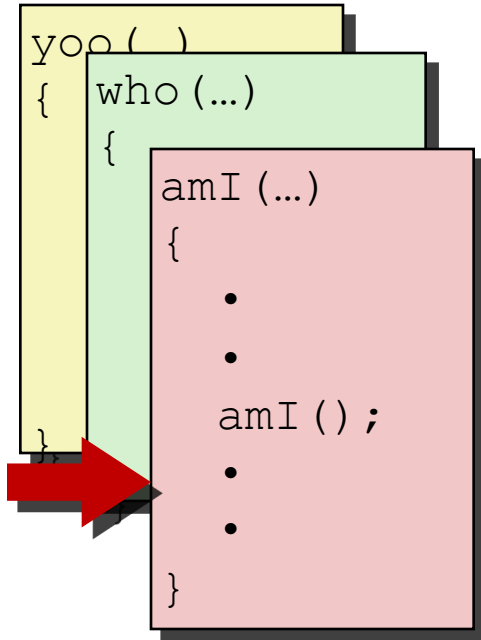
# Example



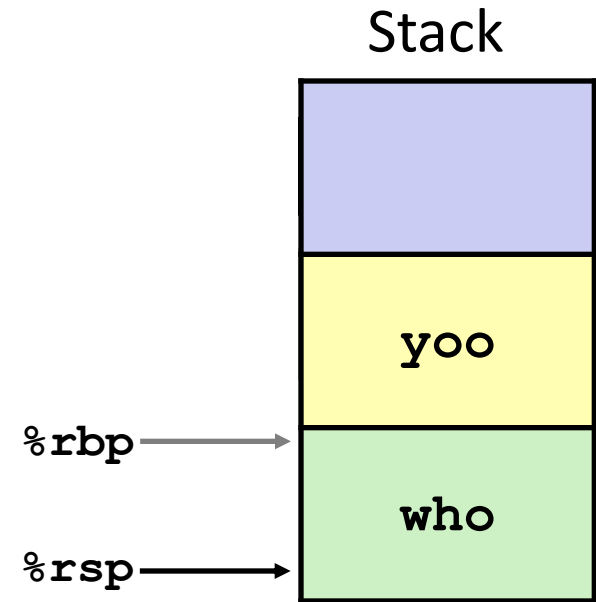
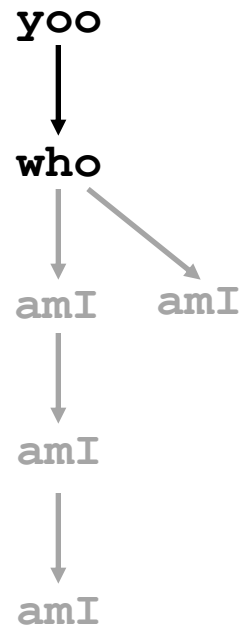
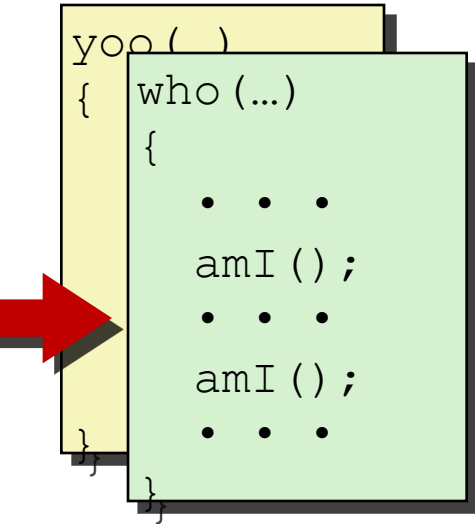
# Example



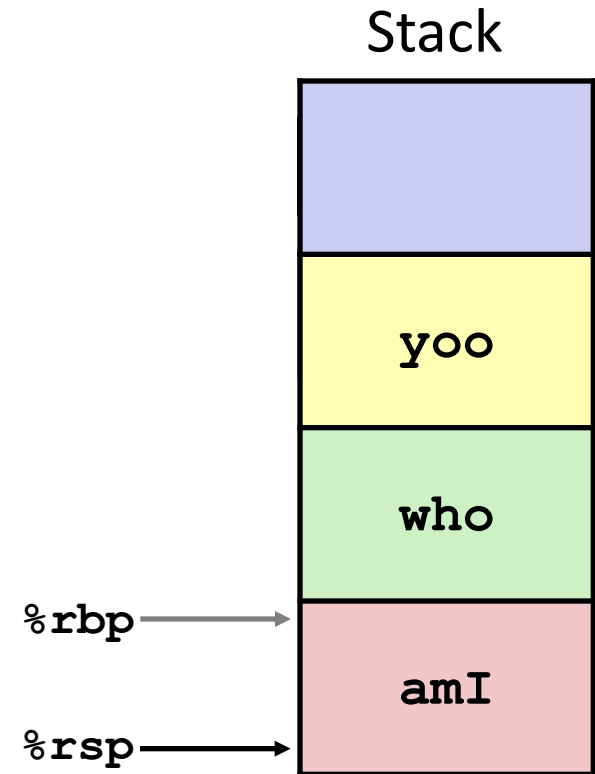
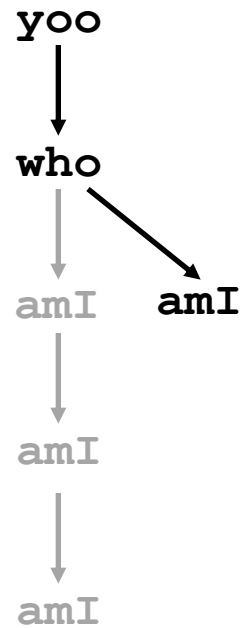
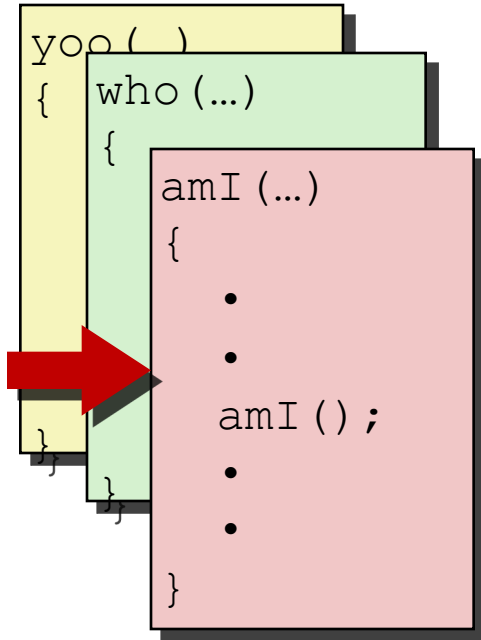
# Example



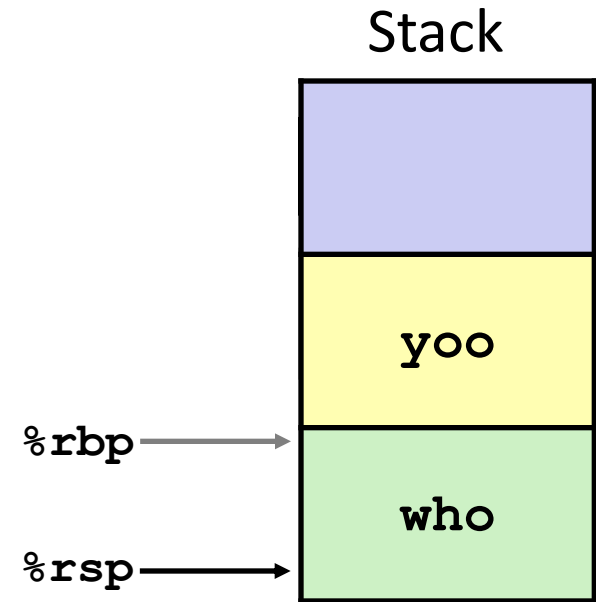
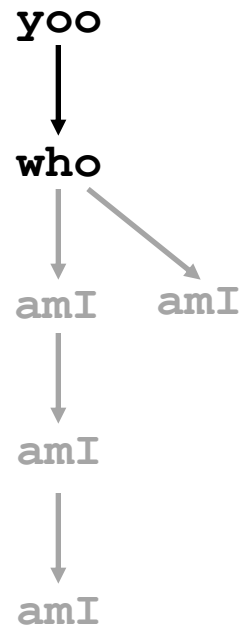
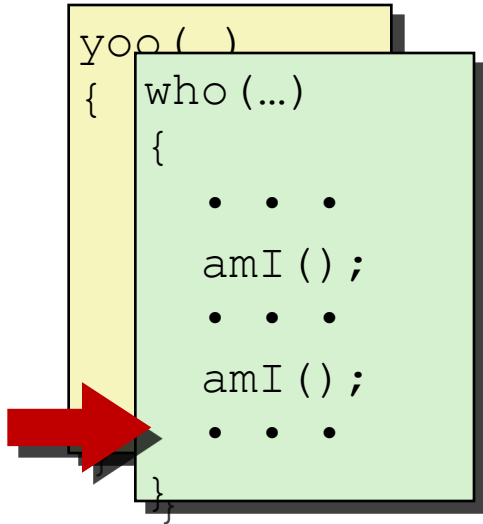
# Example



# Example

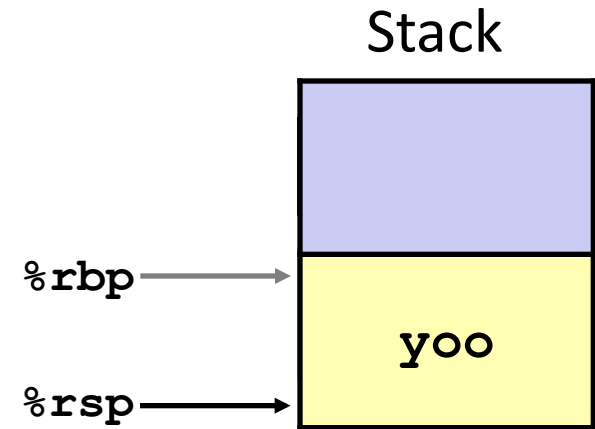
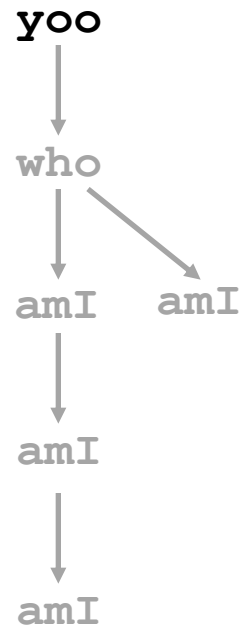
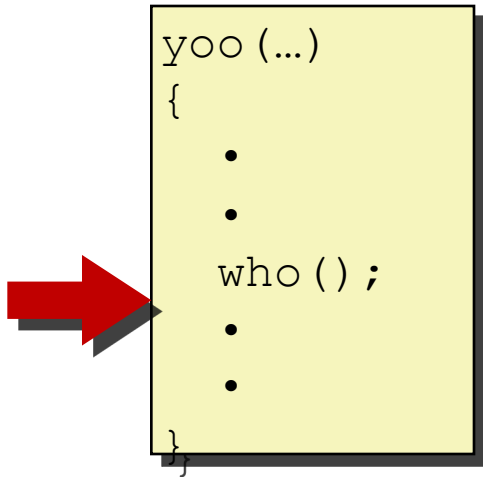


# Example





# Example



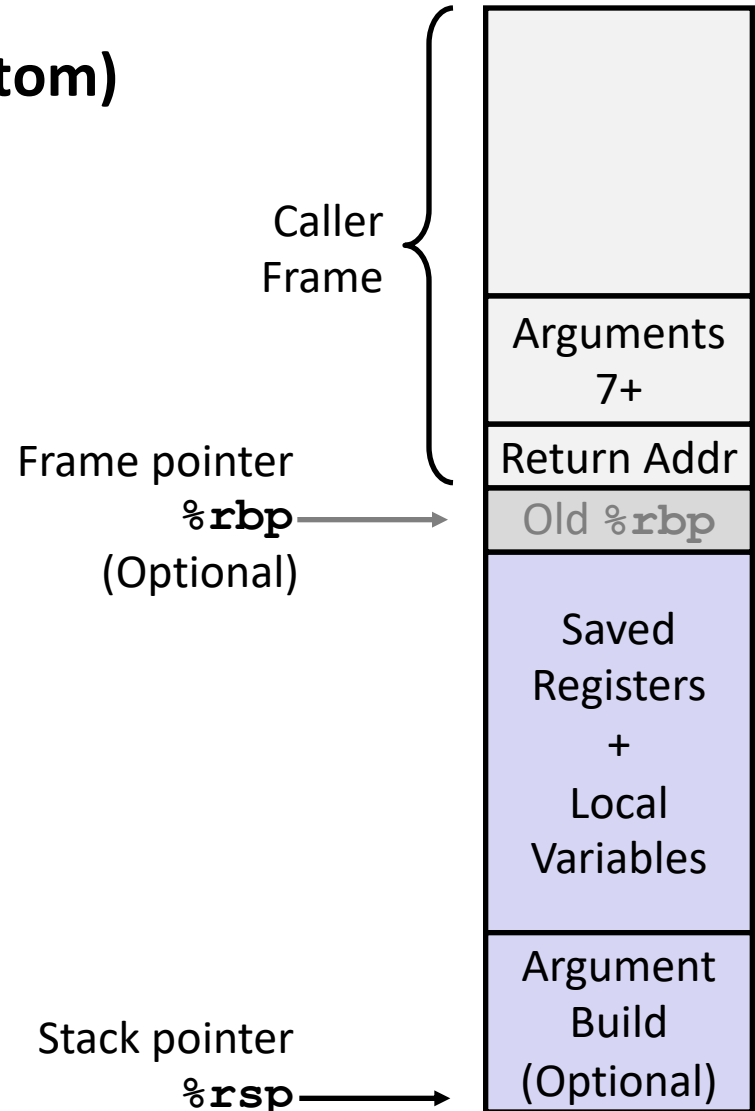
# x86-64/Linux Stack Frame

## ■ 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



# Example: `incr`

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

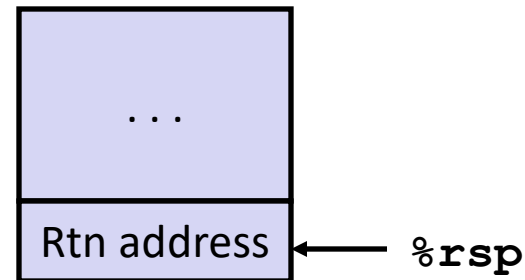
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	Argument <code>val, y</code>
%rax	<code>x</code> , Return value

# Example: Calling `incr` #1

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

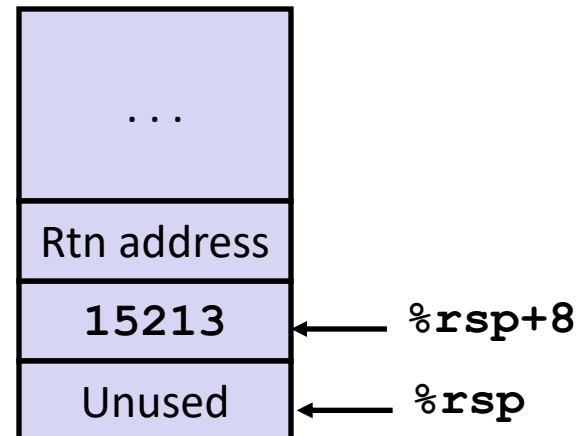
Initial Stack Structure



```
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
```

Allocate stack frame

Resulting Stack Structure

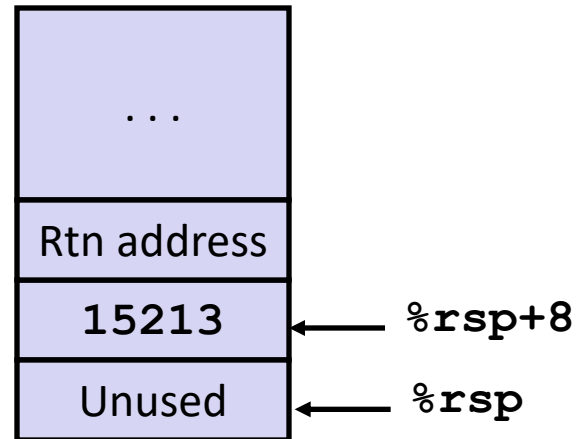


# Example: Calling `incr` #2

```
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

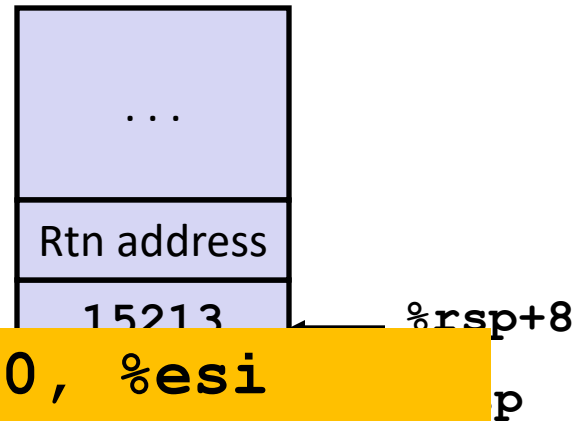


Register	Use(s)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	3000

# Example: Calling `incr` #2

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

Stack Structure



Aside 1: `movl $3000, %esi`

- Note: `movl` -> `%exx` zeros out high order 32 bits.

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

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

# Example: Calling `incr` #2

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

Stack Structure



Aside 2: `leaq 8(%rsp), %rdi`

- Computes `%rsp+8`

```
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
```

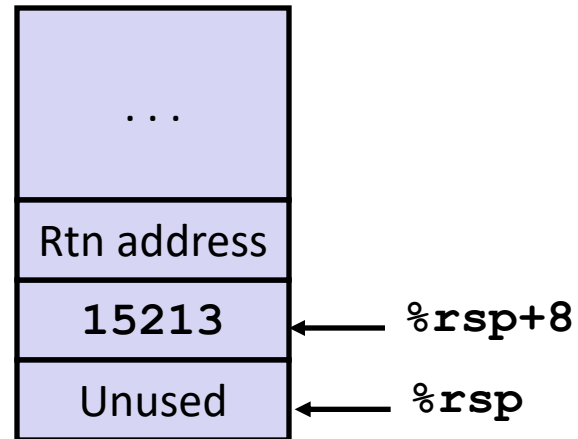
Register	Use(s)
%rdi	&v1
%rsi	3000

# 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)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	3000

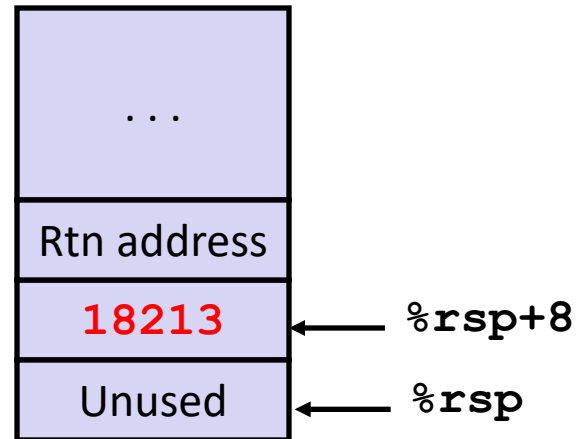


# Example: Calling `incr` #3

```
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



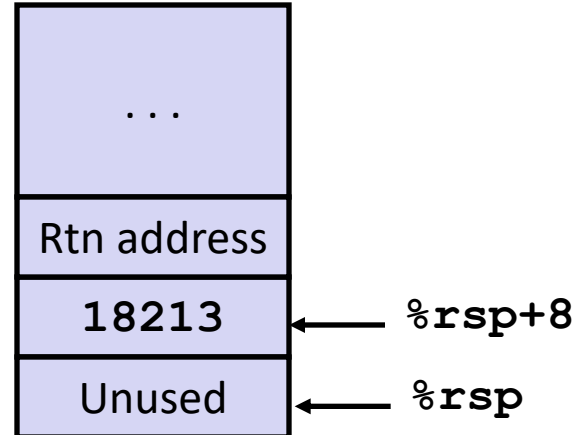
Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling `incr` #4

Stack Structure

```
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
```

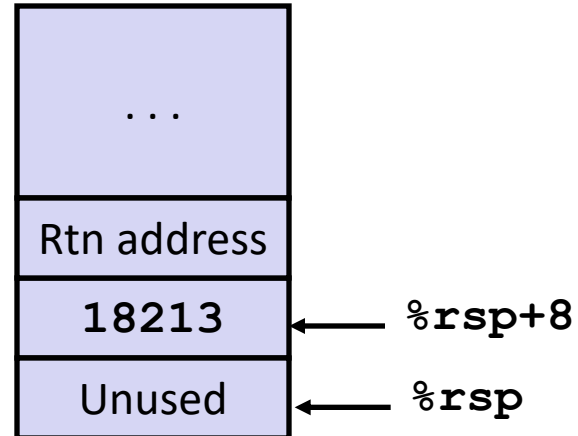


Register	Use(s)
%rax	Return value

# Example: Calling `incr` #5a

Stack Structure

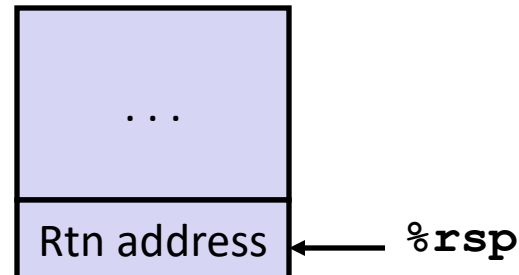
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```



```
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
```

Register	Use(s)
<code>%rax</code>	Return value

Updated Stack Structure

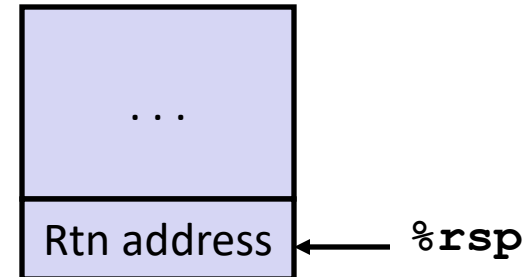


# Example: Calling `incr` #5b

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

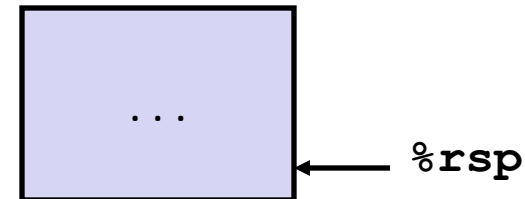
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Updated Stack Structure



Register	Use(s)
<code>%rax</code>	Return value

Final Stack Structure



# 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

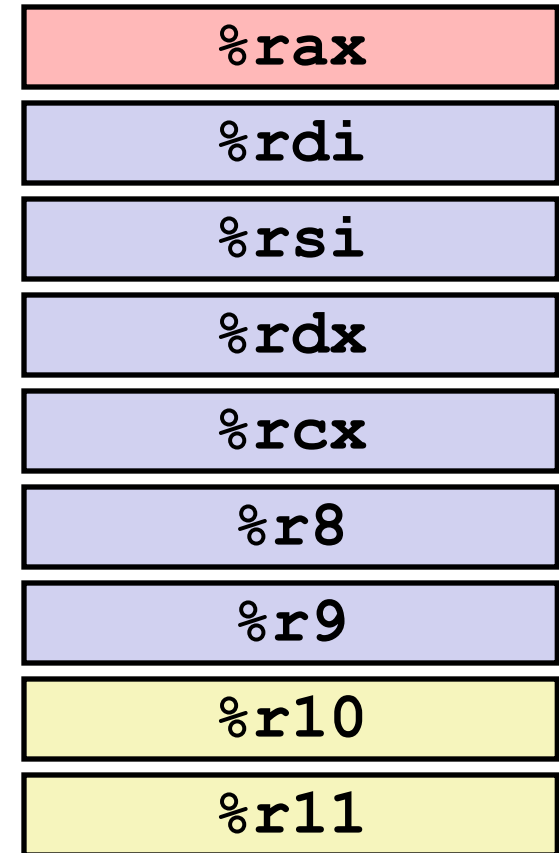
## ■ **%r10, %r11**

- Caller-saved
- Can be modified by procedure

Return value

Arguments

Caller-saved  
temporaries



**Caller-saved**

# x86-64 Linux Register Usage #2

## ■ **%rbx, %r12, %r13, %r14**

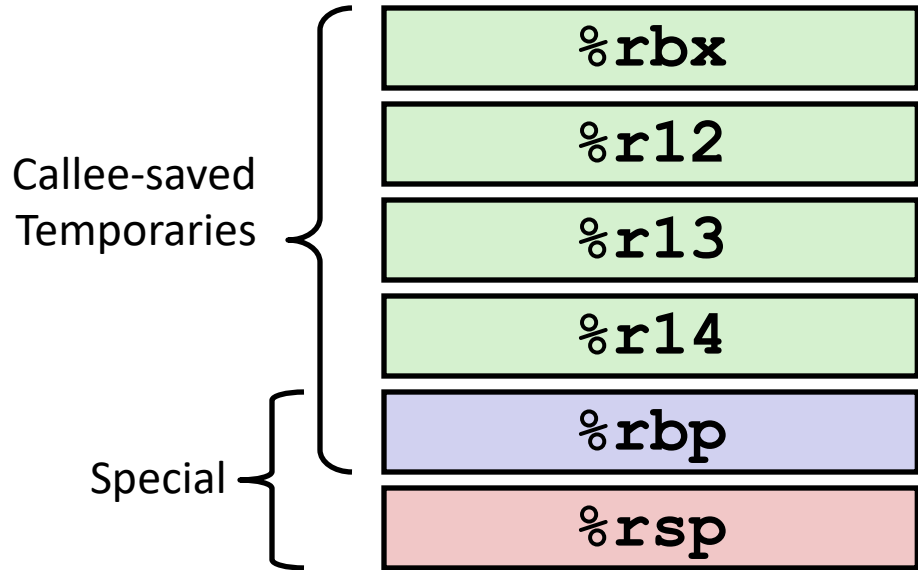
- Callee-saved
- Callee must save & restore

## ■ **%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



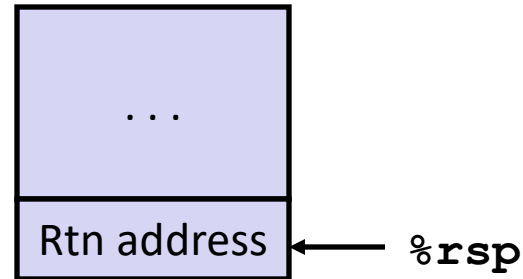
**Callee-saved**



# Callee-Saved Example #1

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

Initial Stack Structure



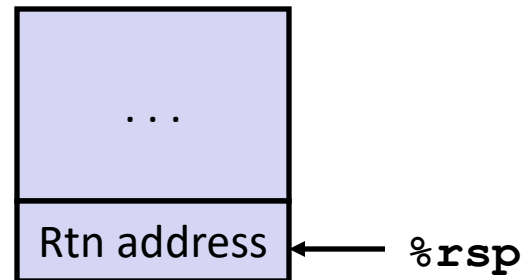
- X comes in register **%rdi**.
- We need **%rdi** for the call to incr.
- Where should we put x, so we can use it after the call to incr?

# Callee-Saved Example #2

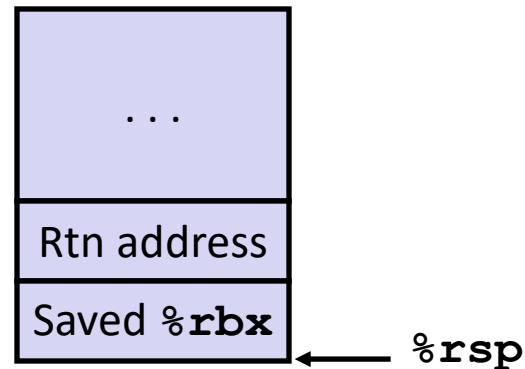
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
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



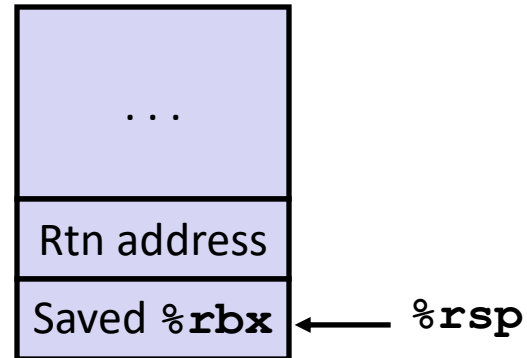
- **x** is saved in **%rbx**,  
a callee saved register

# Callee-Saved Example #3

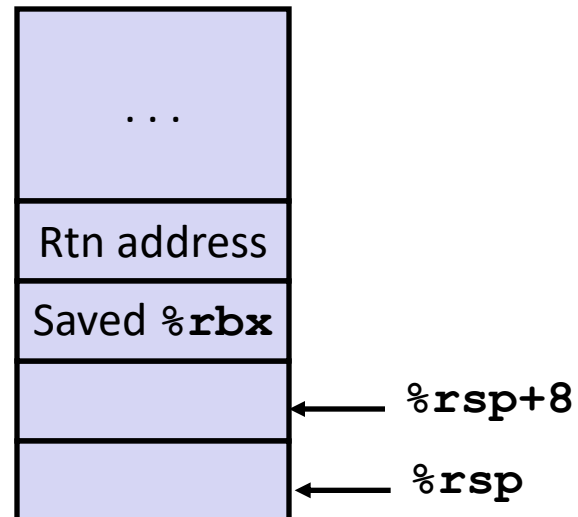
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
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

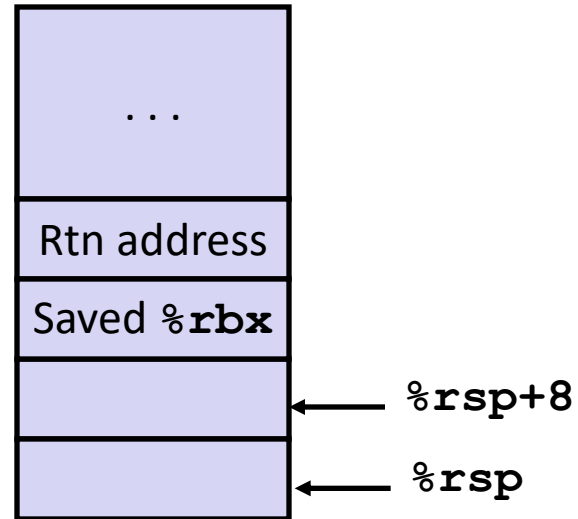


# Callee-Saved Example #4

```
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
```

Stack Structure



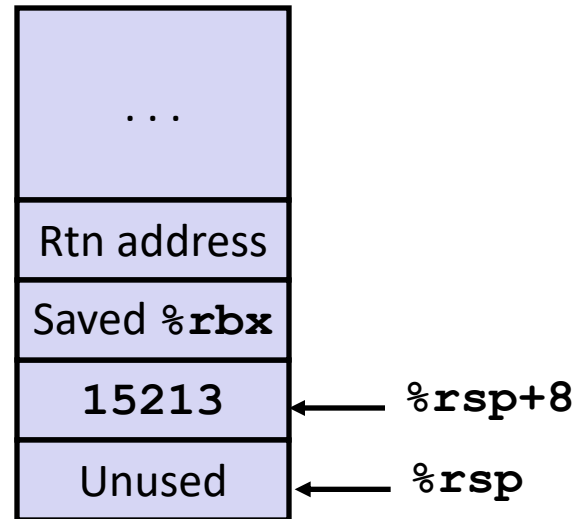
- **x** is saved in **%rbx**,  
a callee saved register

# Callee-Saved Example #5

```
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
```

Stack Structure



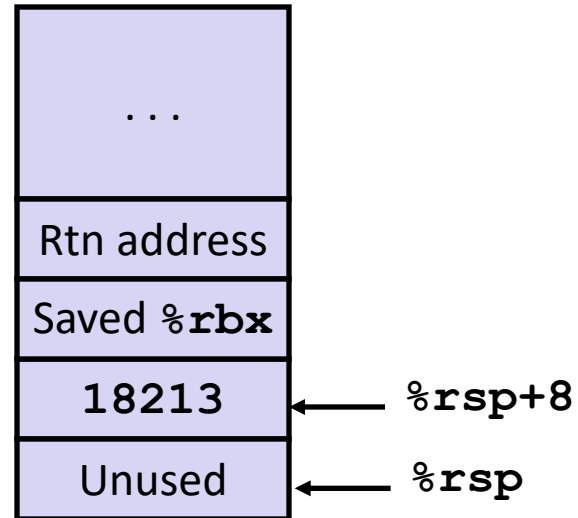
- **x** is saved in **%rbx**,  
a callee saved register

# Callee-Saved Example #6

```
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
```

Stack Structure



Upon return from incr:

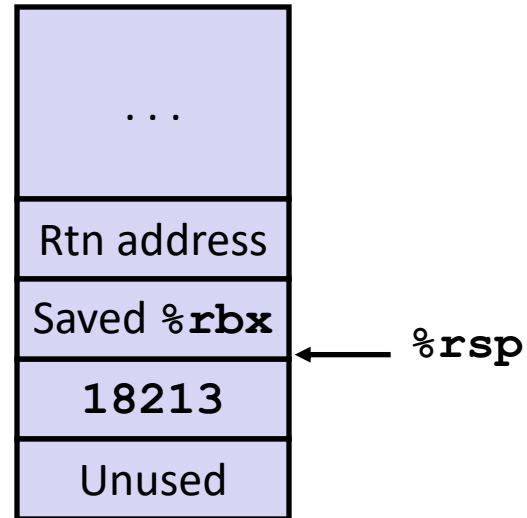
- **x** is safe in **%rbx**
- Return result **v2** is in **%rax**
- Compute **x+v2**

# Callee-Saved Example #7

```
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
```

Stack Structure



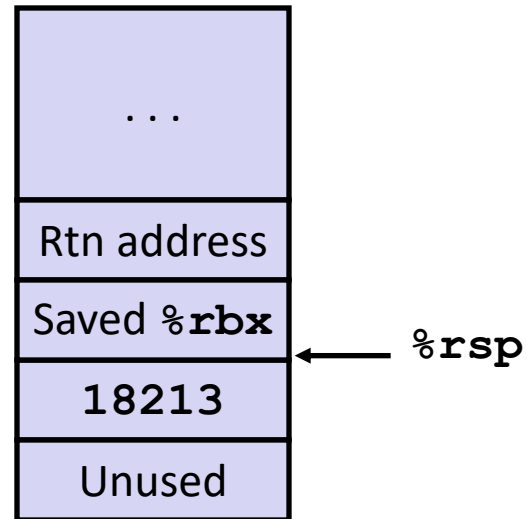
- Return result in **%rax**

# Callee-Saved Example #8

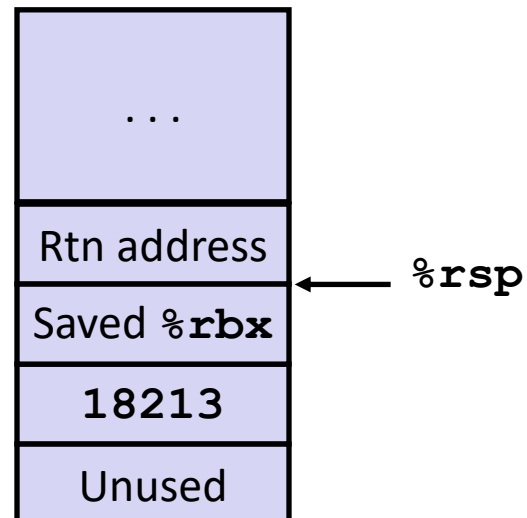
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
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



final Stack Structure





# Quiz Time!

## Exercise 3.34

# Today

## ■ Procedures

- Mechanisms
- 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);
}
```

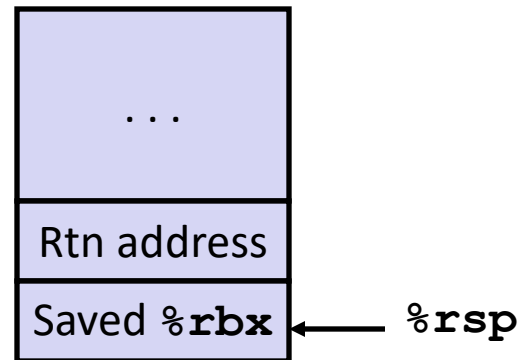
pcount\_r:

```
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



# 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	Recursive 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	

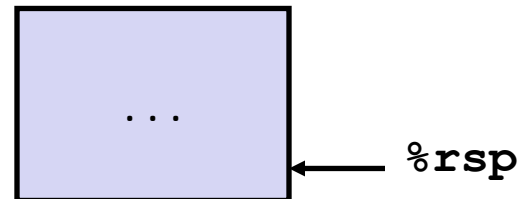


# Recursive Function Completion

```
/* 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 so (e.g., buffer overflow)
- 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

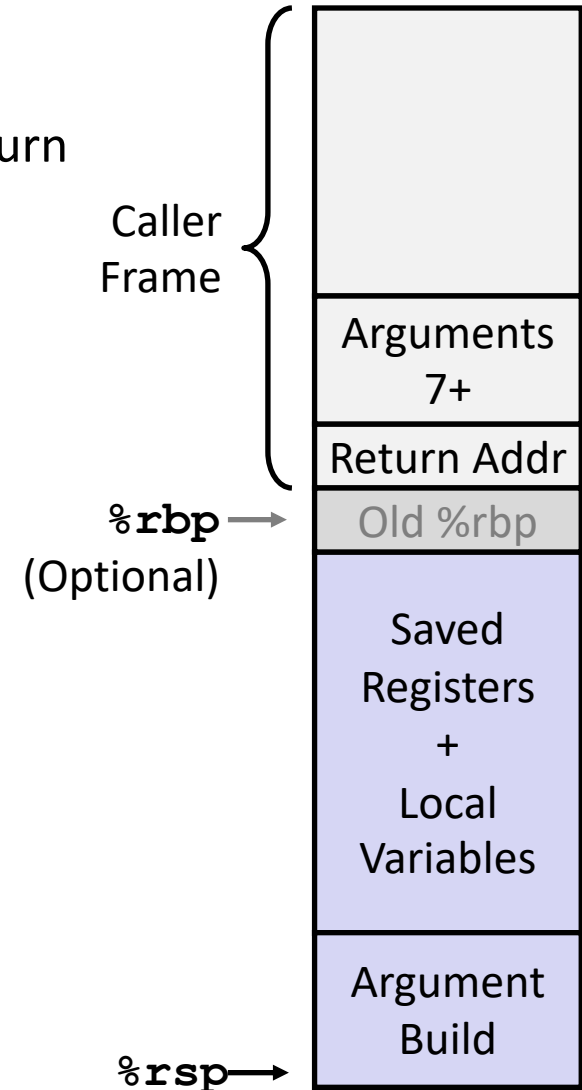
# x86-64 Procedure Summary

## ■ 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

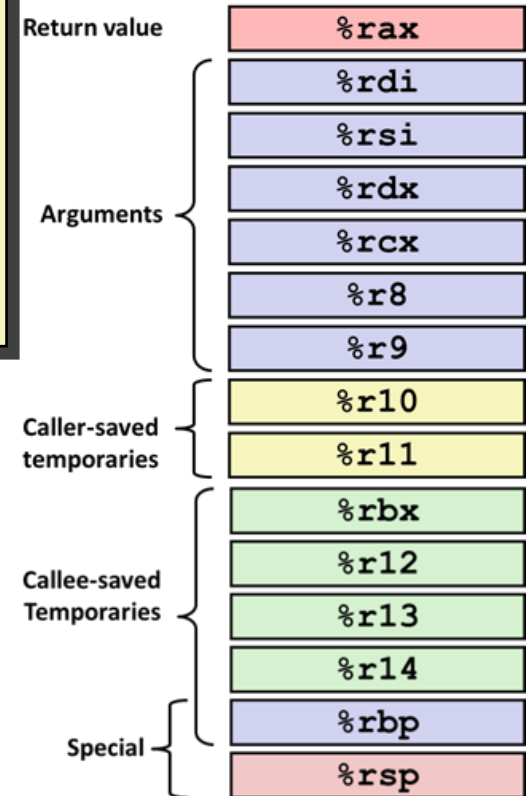
- Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in **%rax**
- ## ■ Pointers are addresses of values
- On stack or global



# Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {  
    return b0+b1+b2+b3+b4;  
}  
  
long add10(long a0, long a1, long a2, long a3, long a4, long a5,  
           long a6, long a7, long a8, long a9) {  
    return add5(a0, a1, a2, a3, a4)+  
           add5(a5, a6, a7, a8, a9);  
}
```

- Where are a0,..., a9 passed?  
**rdi, rsi, rdx, rcx, r8, r9, stack**
- Where are b0,..., b4 passed?  
**rdi, rsi, rdx, rcx, r8**
- Which registers do we need to save in add10?  
Ill-posed question. Need assembly.  
**rbx, rbp, r9 (during first call to add5)**

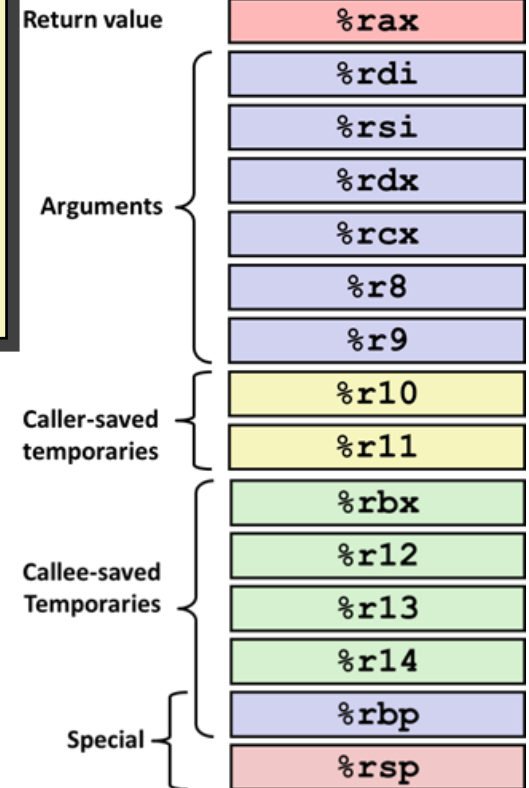


# Small Exercise

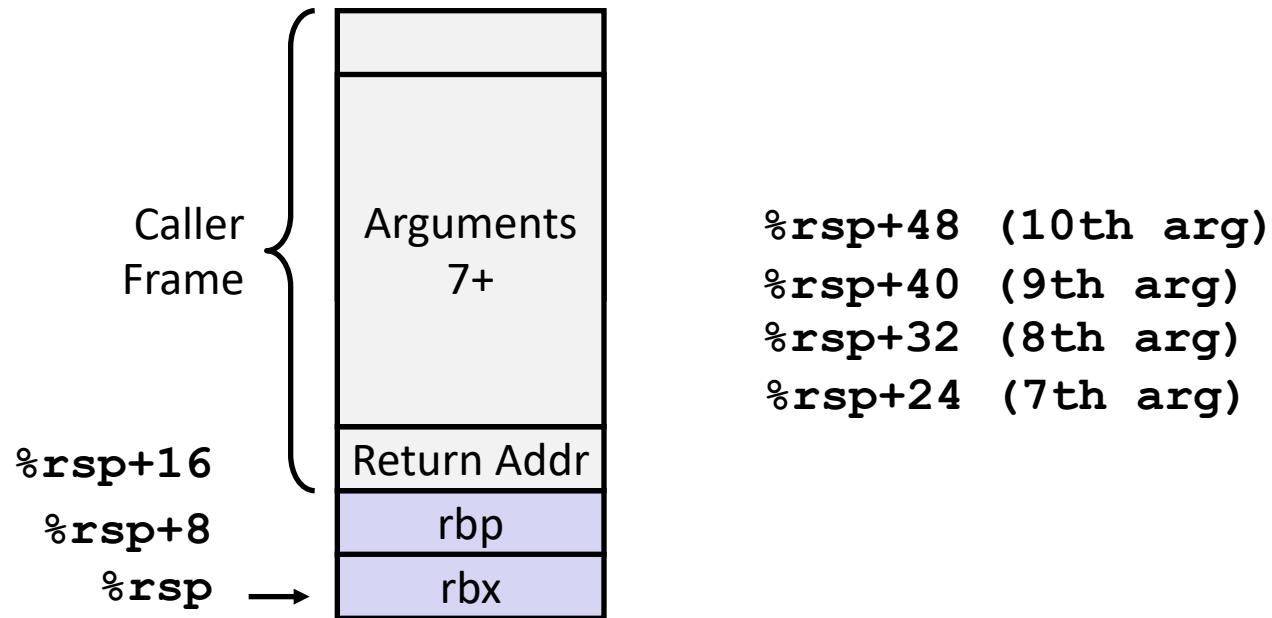
```
long add5(long b0, long b1, long b2, long b3, long b4) {  
    return b0+b1+b2+b3+b4;  
}  
  
long add10(long a0, long a1, long a2, long a3, long a4, long a5,  
           long a6, long a7, long a8, long a9) {  
    return add5(a0, a1, a2, a3, a4)+  
           add5(a5, a6, a7, a8, a9);  
}
```

```
add10:  
    pushq    %rbp  
    pushq    %rbx  
    movq     %r9, %rbp  
    call     add5  
    movq     %rax, %rbx  
    movq     48(%rsp), %r8  
    movq     40(%rsp), %rcx  
    movq     32(%rsp), %rdx  
    movq     24(%rsp), %rsi  
    movq     %rbp, %rdi  
    call     add5  
    addq     %rbx, %rax  
    popq     %rbx  
    popq     %rbp  
    ret
```

```
add5:  
    addq     %rsi, %rdi  
    addq     %rdi, %rdx  
    addq     %rdx, %rcx  
    leaq     (%rcx,%r8), %rax  
    ret
```



# Small Exercise



# Quiz Time!

## Exercise 3.35