

# Lecture 08

# Procedures

CS213 – Intro to Computer Systems  
Branden Ghena – Spring 2021

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

# Administrivia

- Homework 2 due today
  - It will be good practice for the exam
- Midterm Exam 1 Thursday, during class
  - I have already contacted you if you're at a different time
  - Covers material including last week Thursday
    - Not today's material
  - Canvas quiz: 80 minutes to complete (starts at 2pm sharp)
    - Plus 10 minutes to submit images of work afterwards

# Today's Goals

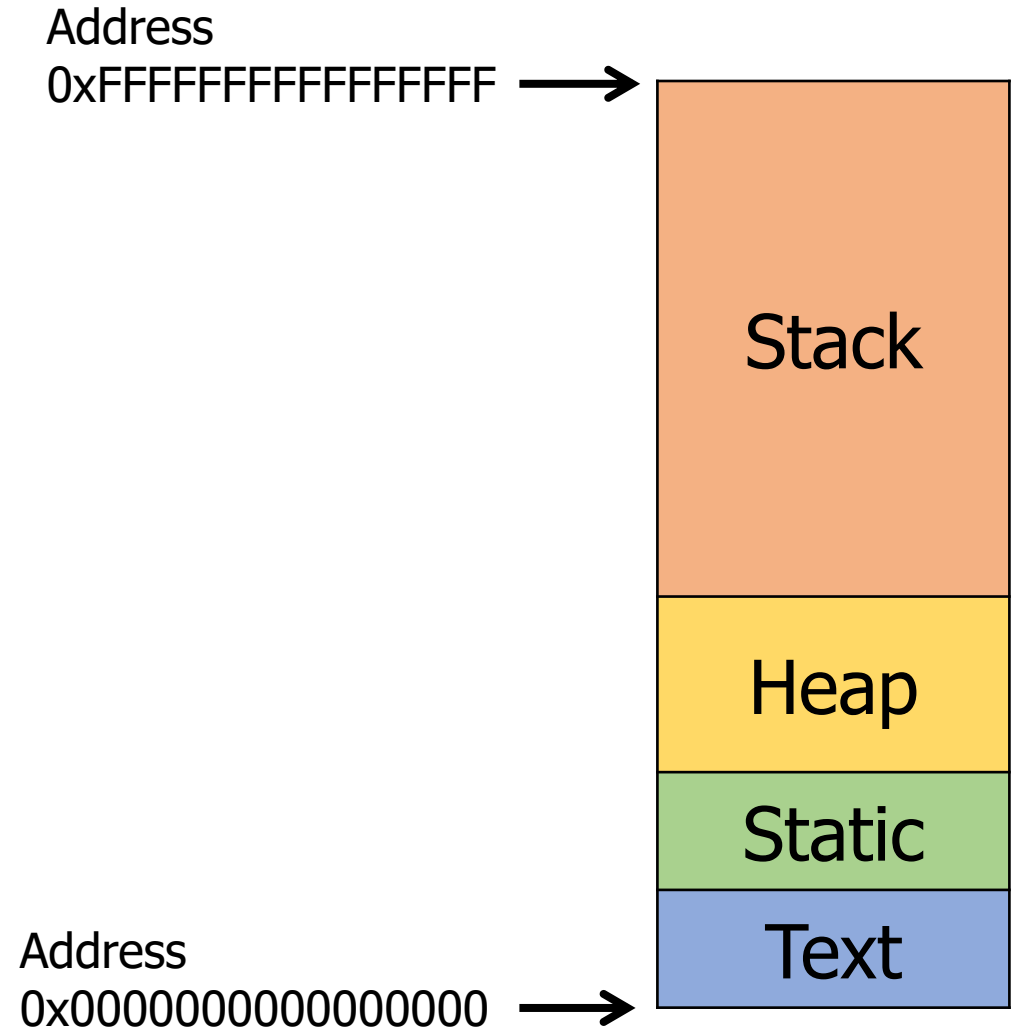
- Describe C memory layout
- Explore functions in assembly
  - How do we call them and return from them?
  - How do we create local variables?
- Understand how we manage register use between functions

# Outline

- **C Code Layout**
- x86-64 Calling Convention
- Managing Local Data
- Register Saving
  - Recursion Example

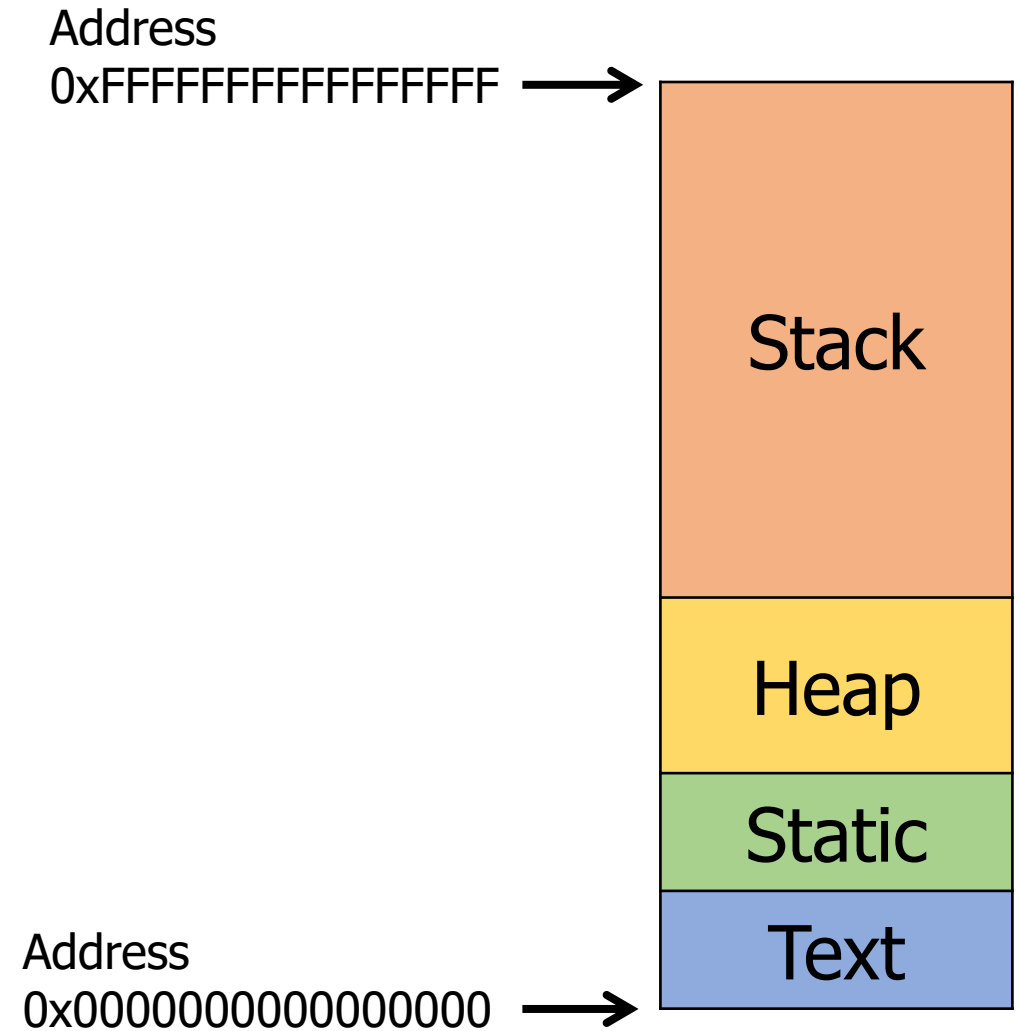
# C memory layout

- Stack Section
  - Local variables
  - Function arguments
- Heap Section
  - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
  - Global variables
  - Static function variables
- Text Section (a.k.a Code Section)
  - Program code



# C memory layout

```
int a;  
  
void foo(short b) {  
    static int c = 3;  
  
    char* d;  
    d = (char*) malloc(4);  
  
    printf("Hello CS213\n");  
}
```



# C memory layout

```
int a;
```

```
void foo(short b) {
```

```
    static int c = 3;
```

```
    char* d;
```

```
    d = (char*) malloc(4);
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

# C memory layout

```
int a;
```

```
void foo(short b) {  
    static int c = 3;
```

```
    char* d;
```

```
    d = (char*) malloc(4);
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →



# C memory layout

```
int a;
```

```
void foo(short b) {
```

```
    static int c = 3;
```

```
    char* d;
```

```
    d = (char*) malloc(4);
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

# C memory layout

```
int a;
```

```
void foo(short b) {
```

```
    static int c = 3;
```

```
    char* d;
```

```
    d = (char*) malloc(4);
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

# C memory layout

```
int a;
```

```
void foo(short b) {
```

```
    static int c = 3;
```

```
    char* d;
```

```
    d = (char*) malloc(4);
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

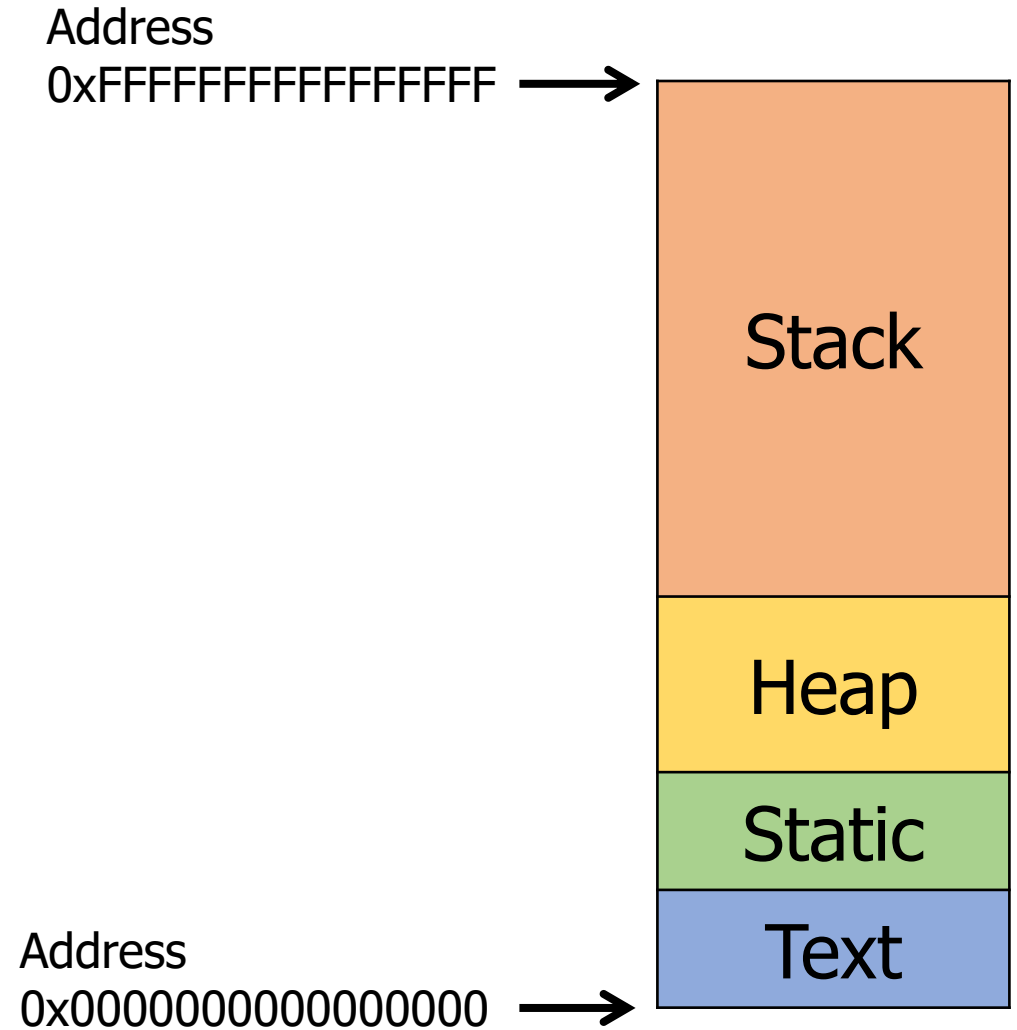
Address

0x0000000000000000 →

# C memory layout

```
int a;  
  
void foo(short b) {  
    static int c = 3;  
  
    char* d;  
    d = (char*) malloc(4);  
  
    printf("Hello CS213\n");  
}
```

Assembly code goes in the  
Text section



# Interacting with data sections in assembly

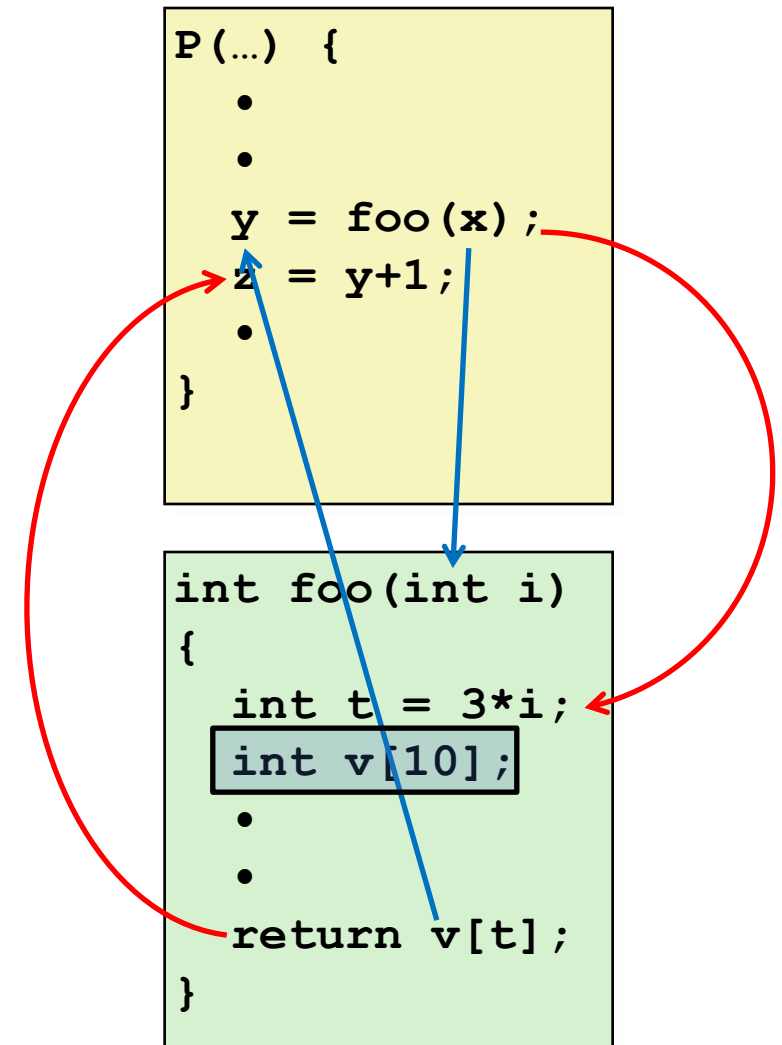
- Stack
  - Stack pointer is saved in `%rsp` and can be moved as needed
  - We'll discuss this today
- Heap
  - C library (malloc) handles this above the machine level
  - i.e. from the machine point of view, there is no heap
- Static
  - Arbitrary pointers to memory can be created and used
    - With memory addressing instructions
  - Assembly directive can place values into Static section
- Text
  - Assembly code is placed here automatically
  - Labels are just addresses within the Text section

# Outline

- C Code Layout
- **x86-64 Calling Convention**
- Managing Local Data
- Register Saving
  - Recursion Example

# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Local memory management
  - Allocate during procedure execution
  - Deallocate upon return
- No one instruction does all that
  - Need instructions for each
- The stack is the key to all 3 of these!



# Procedure control flow

- Use stack to support procedure call and return!

- Procedure call

`callq label`      Push return address on stack; jump to *label*

- Procedure return

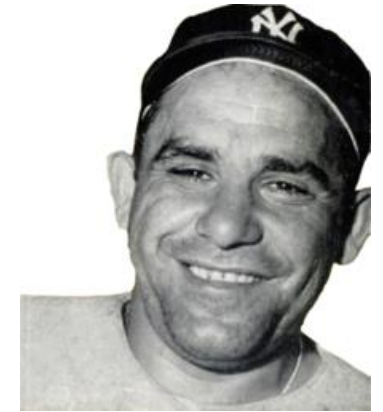
`retq`      Pop address from stack; jump there  
(stack should be as it was when the call began)

- Return address value

- Address of instruction immediately following `callq`
- Example from disassembly

```
400544: call 400550 <mult2>
400549: mov  %rax, (%rbx)
```

Return address: 0x400549



If you don't know where  
you're going, you may  
not get there.

— Yogi Berra

Just `call` and `ret` are fine  
`q` is assumed (and can't be changed)



# Code Examples

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

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

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

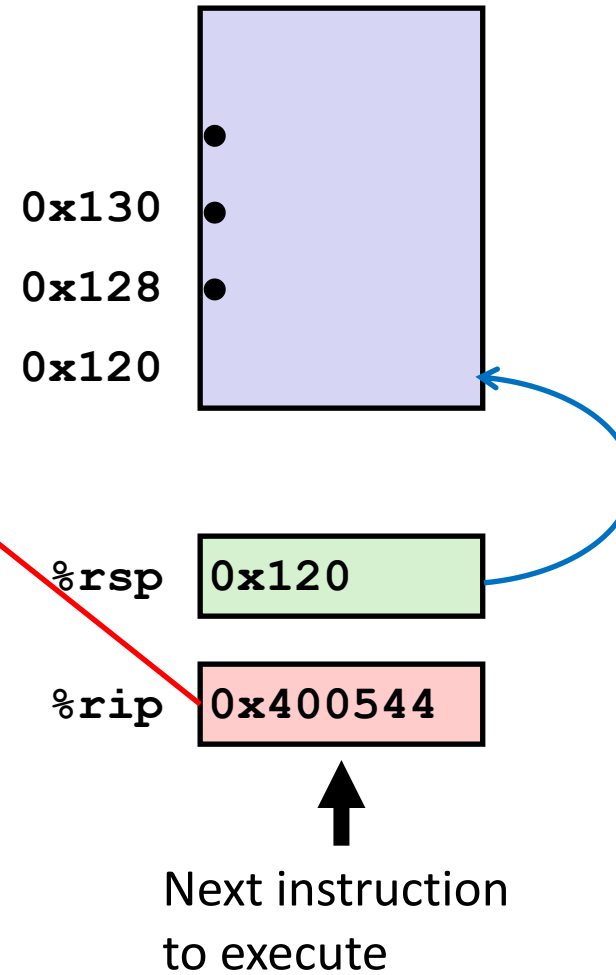
```
0000000000400550 <mult2>:  
400550: mov     %rdi,%rax      # a  
400553: imul    %rsi,%rax      # a * b  
400557: retq                    # Return
```

# Control Flow Example

about to execute `callq`

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

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

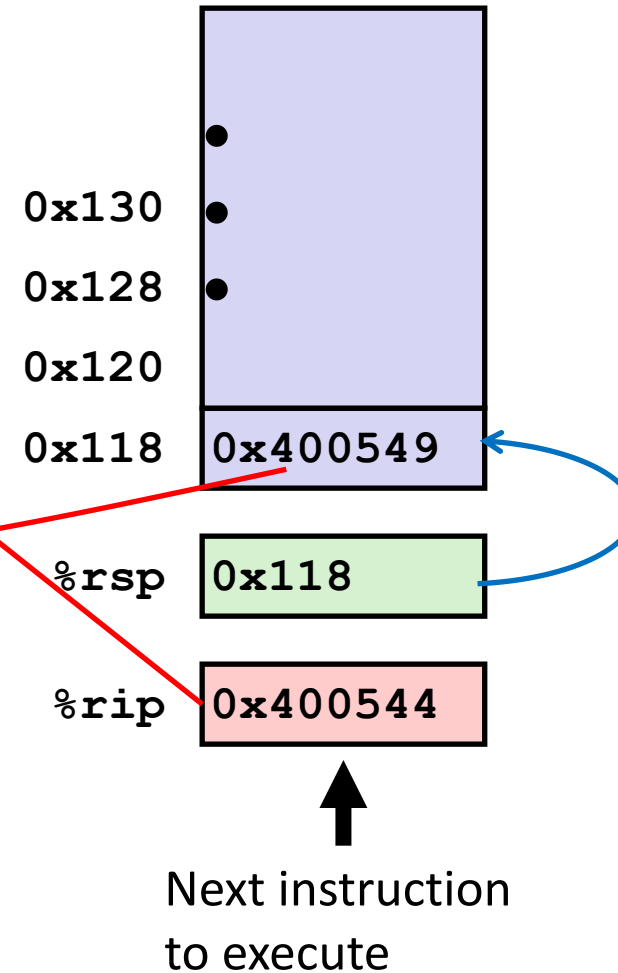


# Control Flow Example

callq step 1

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

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

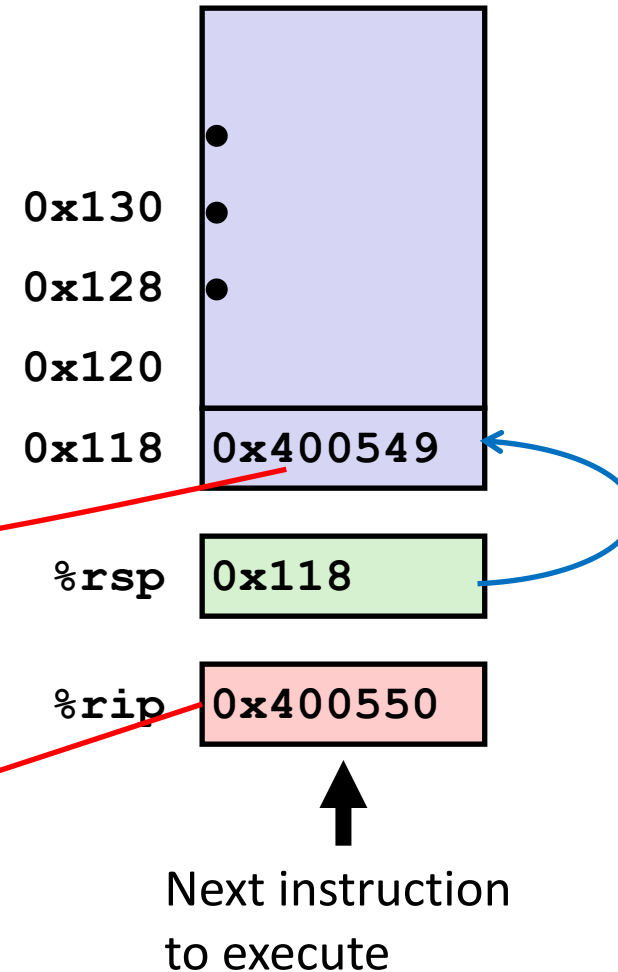


# Control Flow Example

callq step 2

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

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

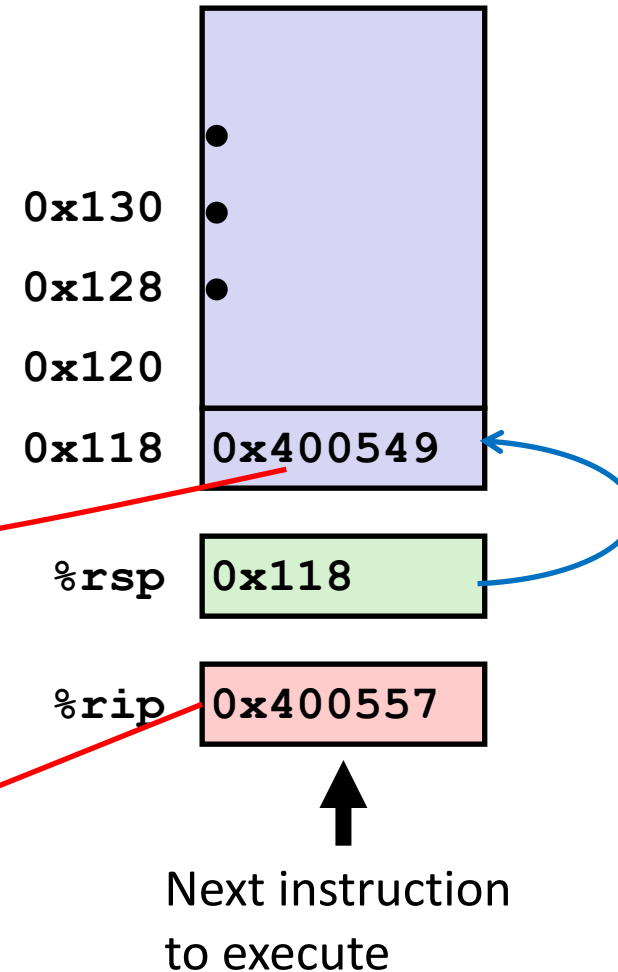


# Control Flow Example

about to execute `retq`

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

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



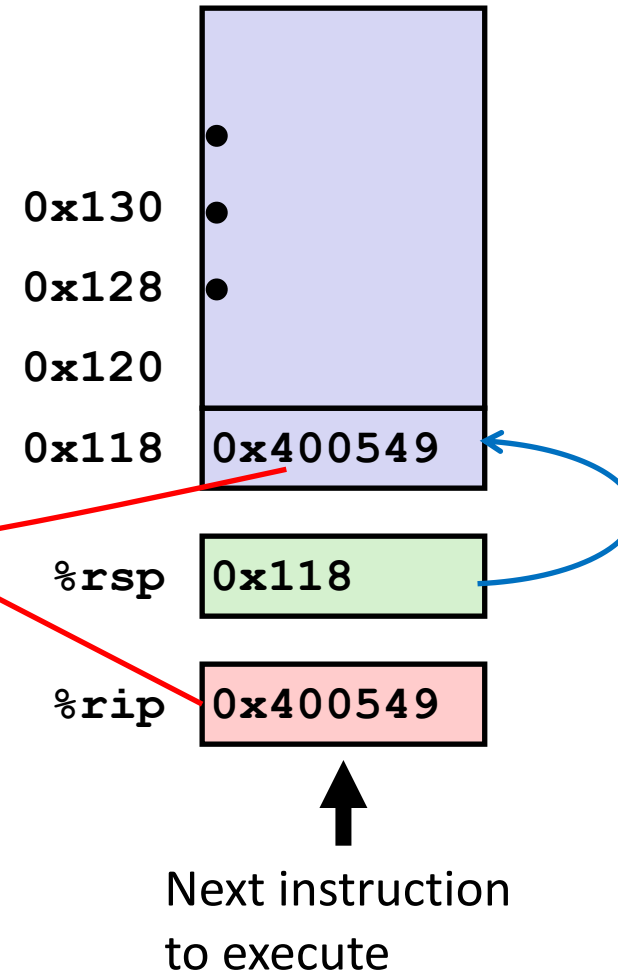
**QUIZ:** What is the address of the instruction we execute after `retq`?

# Control Flow Example

retq step 1

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

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

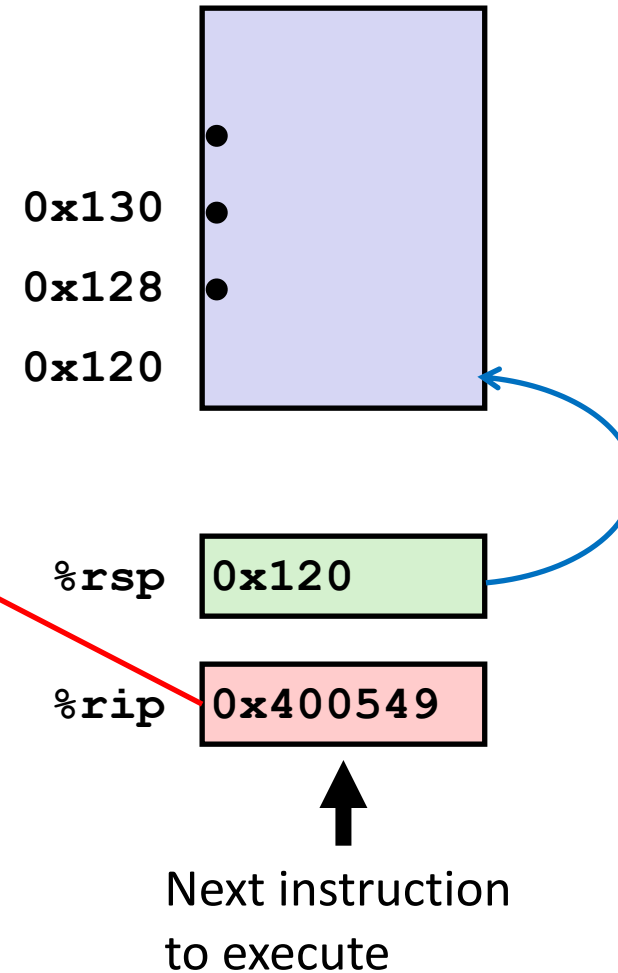


# Control Flow Example

retq step 2

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

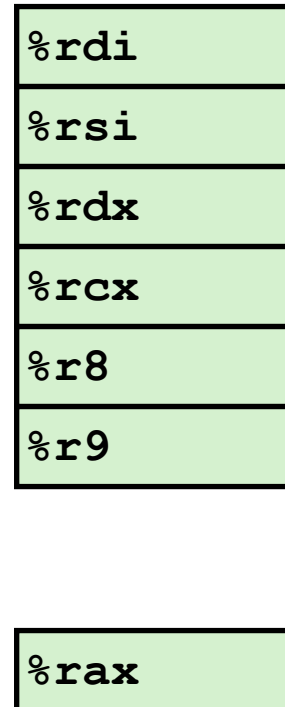
```
0000000000400550 <mult2>:  
400550: mov  %rdi, %rax  
.  
.  
400557: retq
```



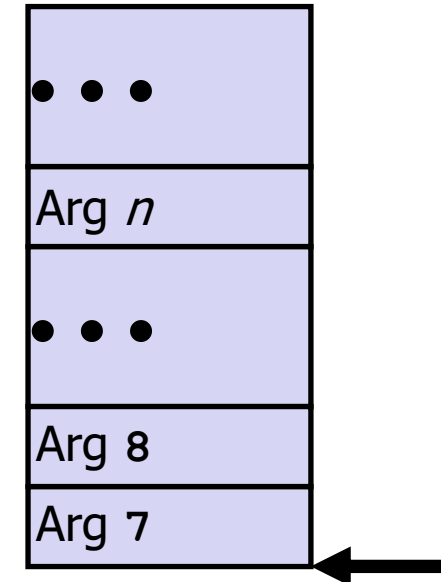
# Function data flow

- First 6 arguments are in registers
  - `%rdi` is first argument
- Next `n` arguments are on the stack
  - This means more arguments is slower
- Return value is in `%rax`

## • Registers



## • Stack



top

- 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) # \*dest = t

• • •

```
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

# Break + Open Question

- How did we decide how many registers to use for arguments and return values?

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

- Do all functions have to use this same convention?

<code>%rax</code>
-------------------

# Break + Open Question

- How did we decide how many registers to use for arguments and return values?
  - Testing lots of real-world programs
  - Many style guides suggest four or less
  - x86 (32-bit) only had four arguments
    - x86-64 added two more
  - C only has one return result, so one register is fine
- Do all functions have to use this same convention?
  - All functions within a program must, or they won't work
  - Different programs, or different OSes, could choose different

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

<code>%rax</code>
-------------------

# Outline

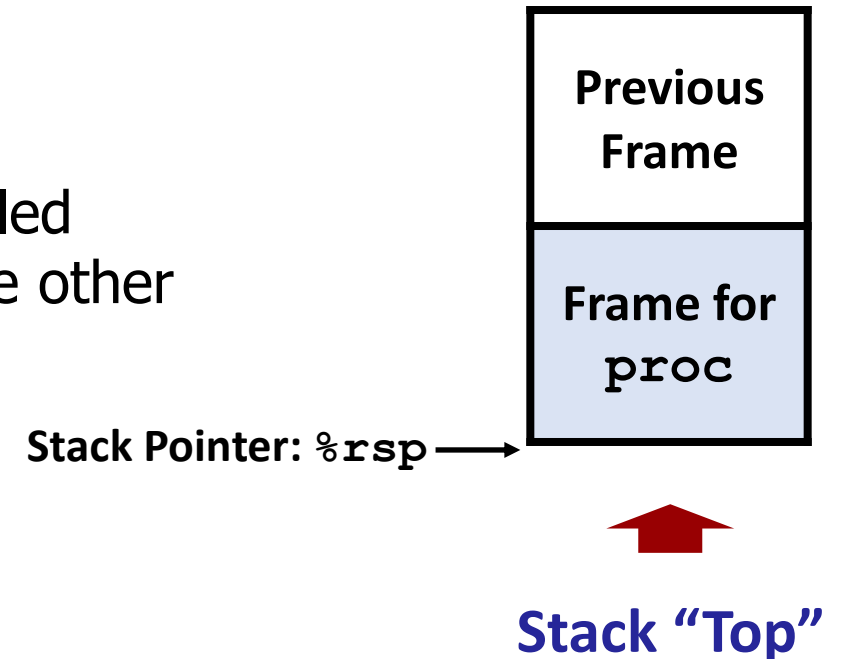
- C Code Layout
- x86-64 Calling Convention
- **Managing Local Data**
- Register Saving
  - Recursion Example

# Call-Local State

- Need some place to store state for each call
  - Return address
  - Arguments
  - Local variables
  - Temporary space (if needed)
- Note: these are separate for each call, not each function
  - Function could be called recursively, but each needs its own local variables
- State only needs to exist until the function returns

# Using the Stack for Call-Local State

- Place local state on the stack
- Stack discipline
  - That state is only needed for limited time
    - Starts when proc is called; ends when it returns
  - **Callee** returns before **caller** does
    - **Callee**: for a specific call, the function being called
    - **Caller**: for a specific call, the function calling the other
- Stack allocated in **Frames**
  - Frame = State for a single procedure invocation
  - Allocated by “setup” code at the start of proc
  - Deallocated by “teardown” code before returning



# Call Chain Example

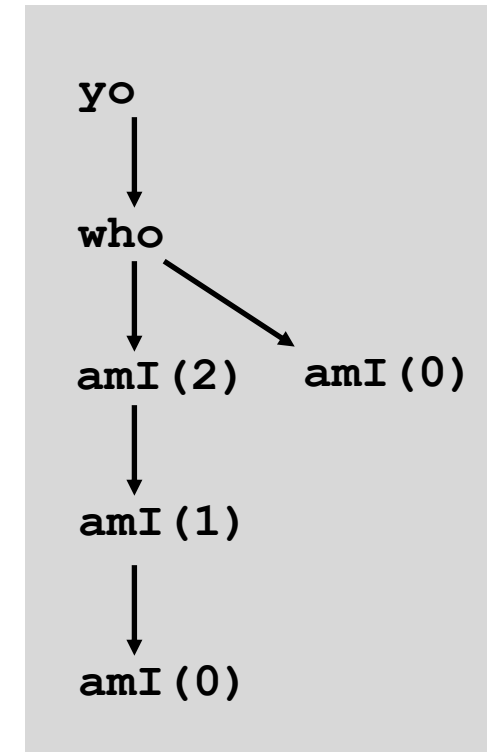
```
yo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI (2) ;  
  . . .  
  amI (0) ;  
  . . .  
}
```

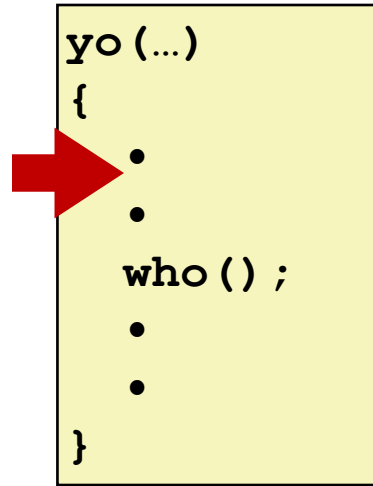
```
amI (int x)  
{  
  .  
  if (x)  
    amI (x-1) ;  
  .  
  .  
}
```

Procedure amI ( ) is recursive

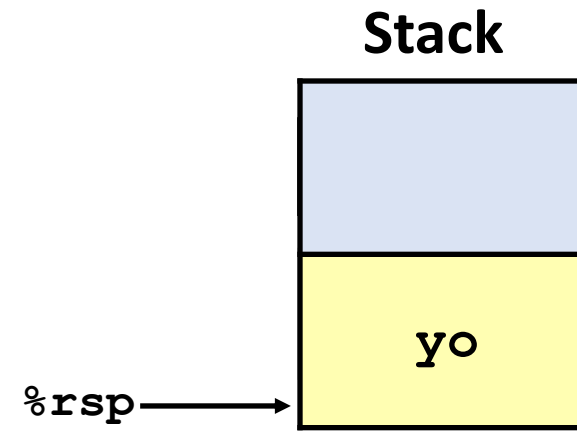
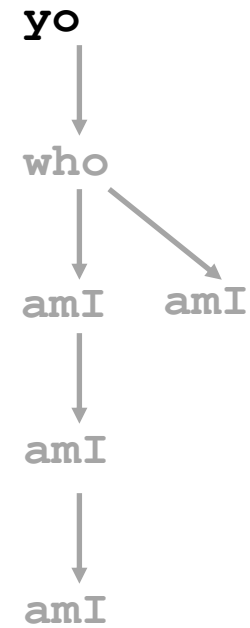
## Example Call Chain



# Example

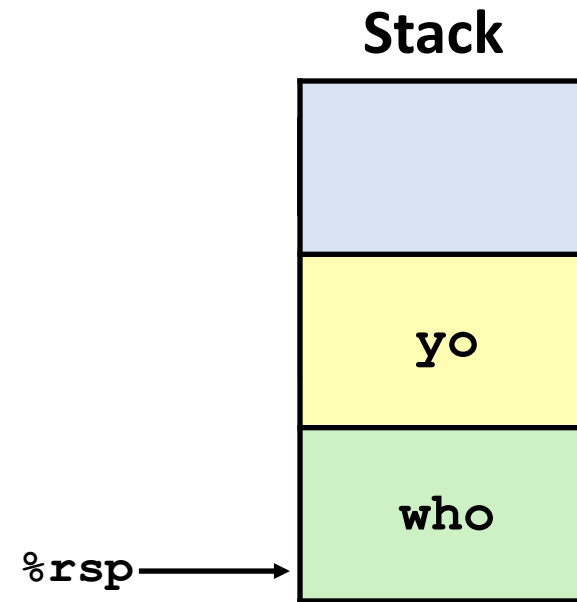
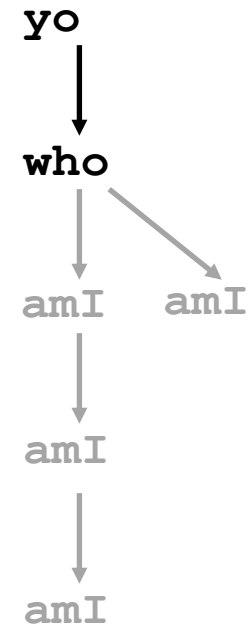
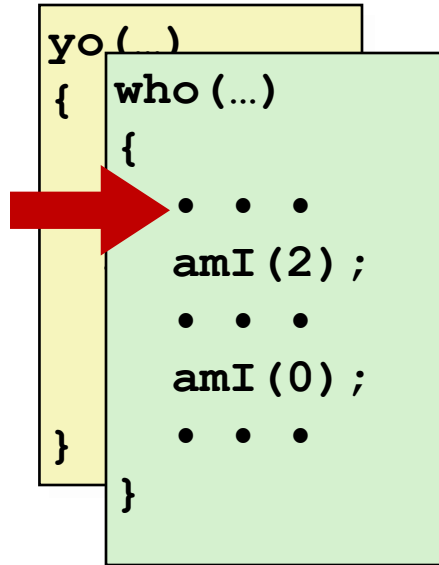


## Call Chain

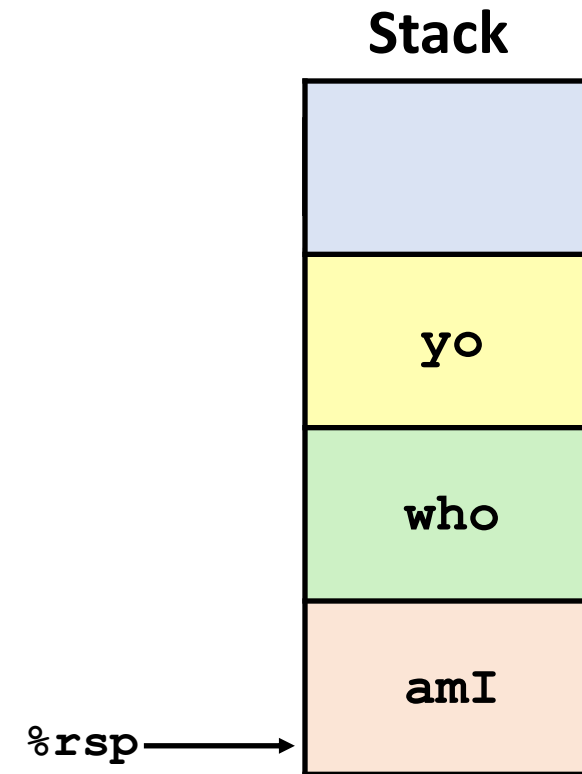
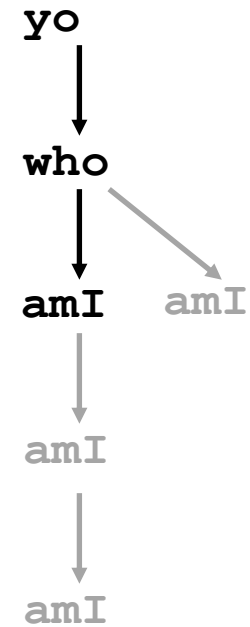
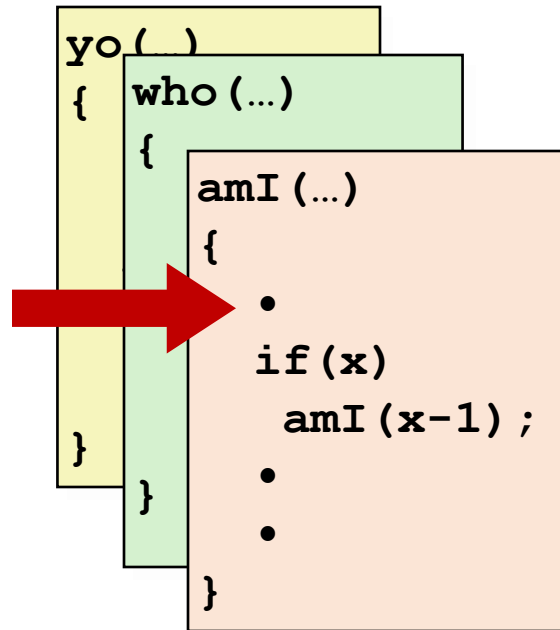




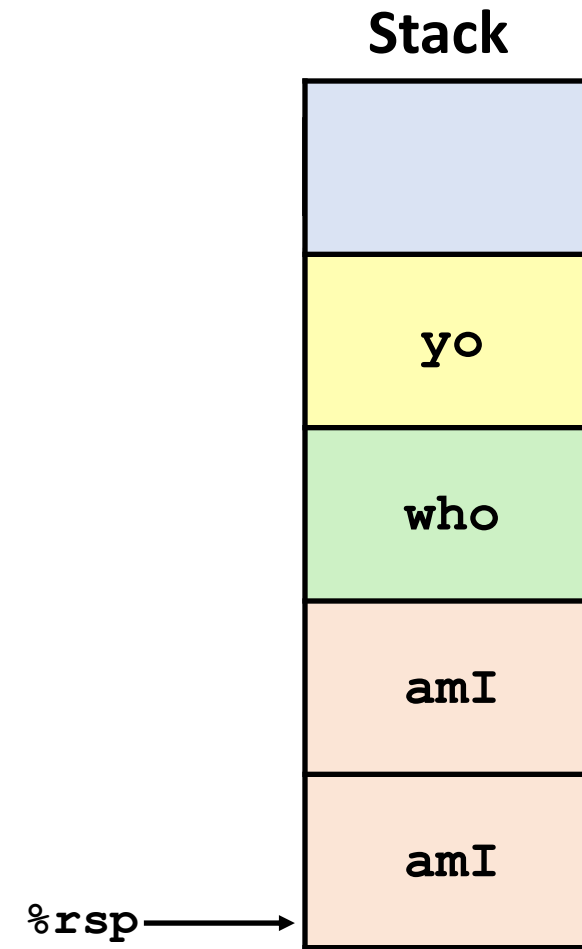
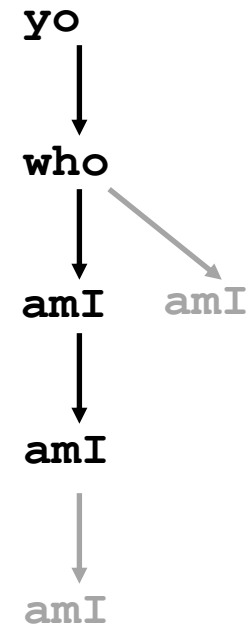
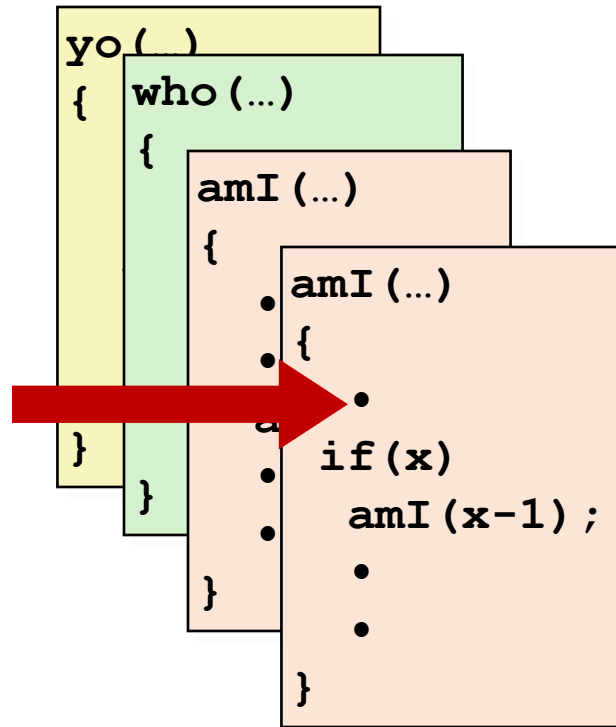
# Example



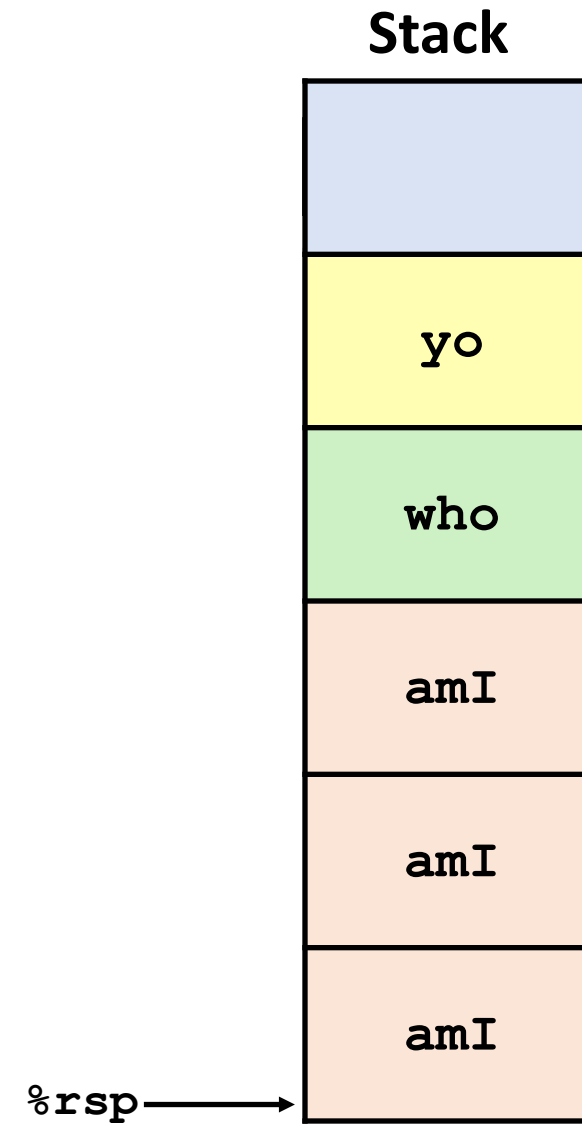
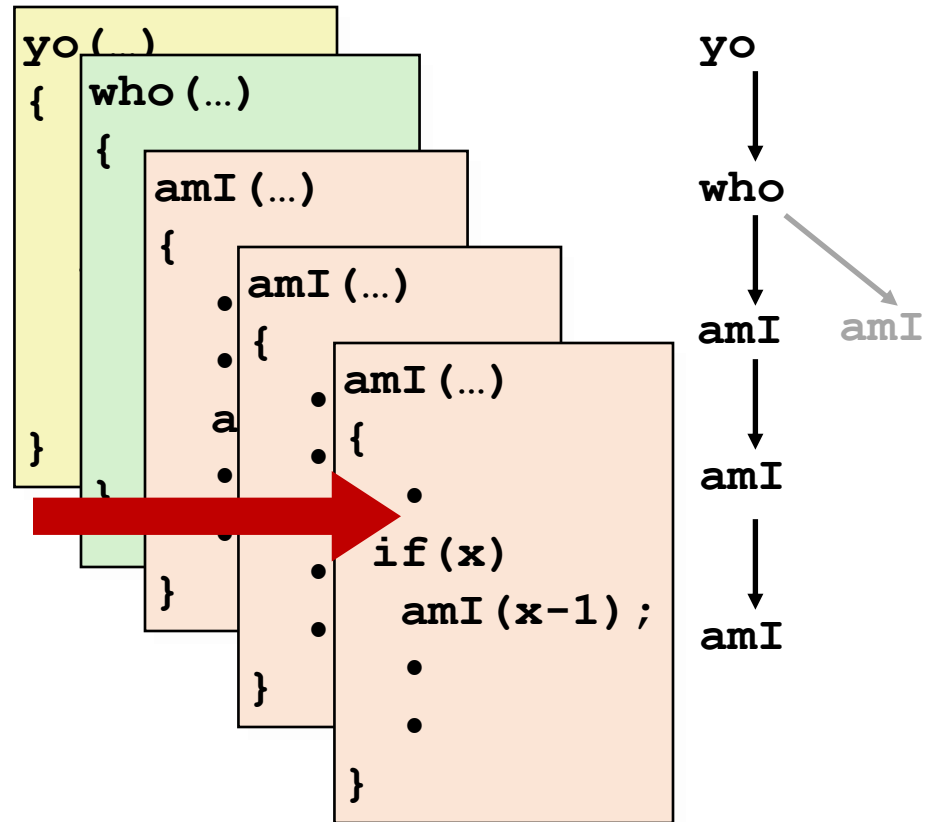
# Example



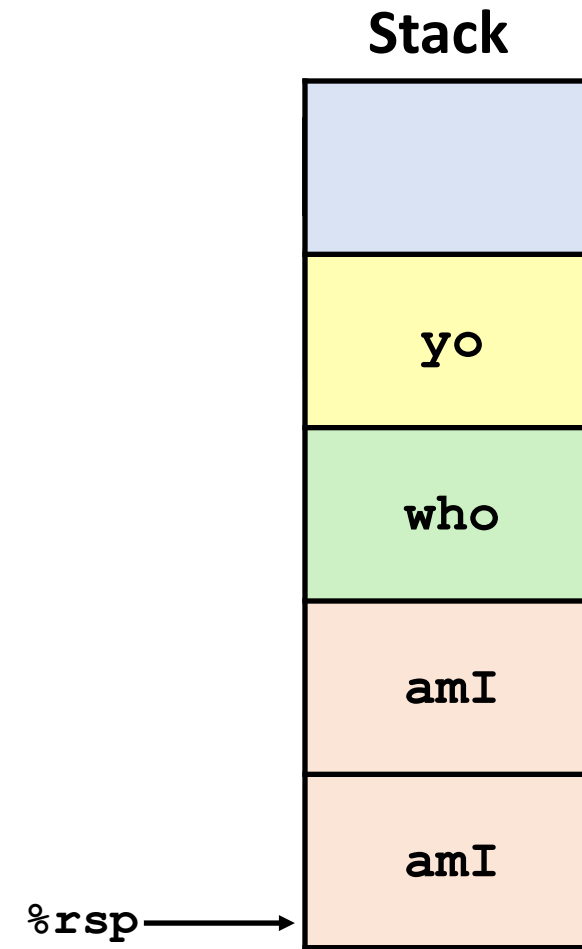
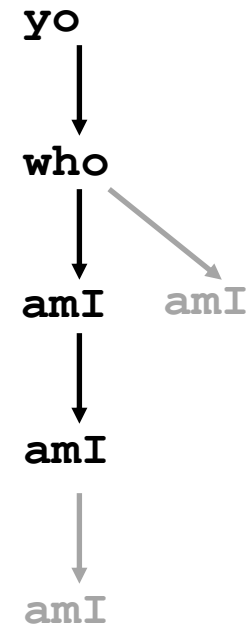
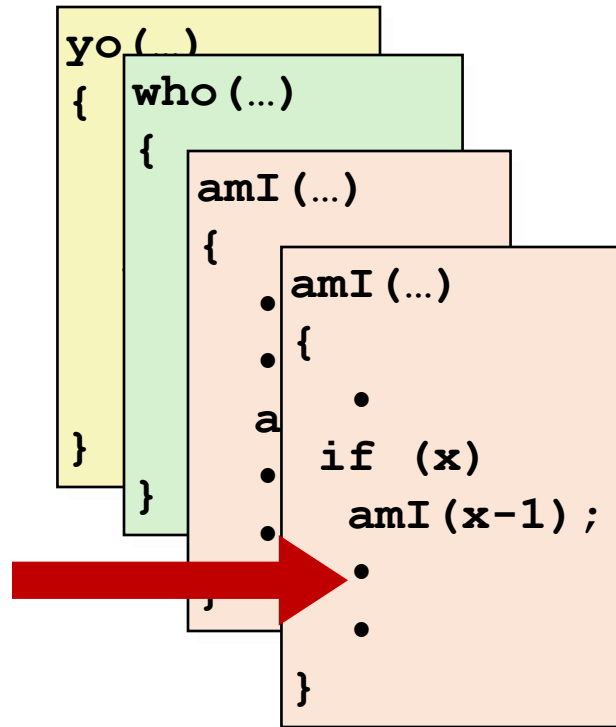
# Example



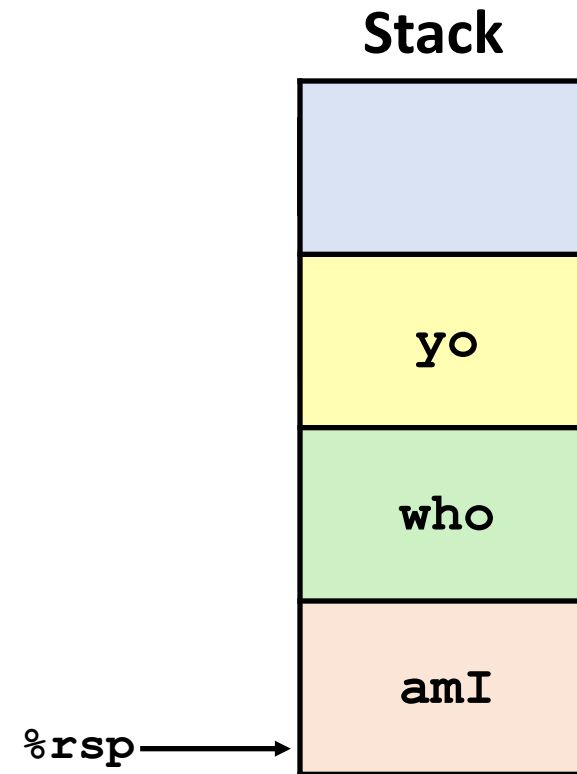
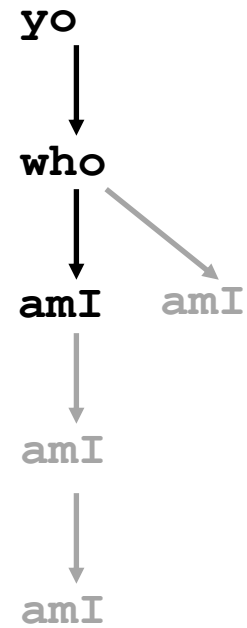
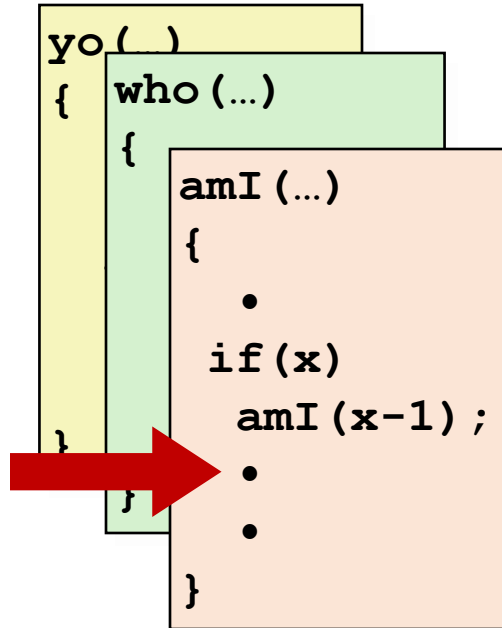
# Example



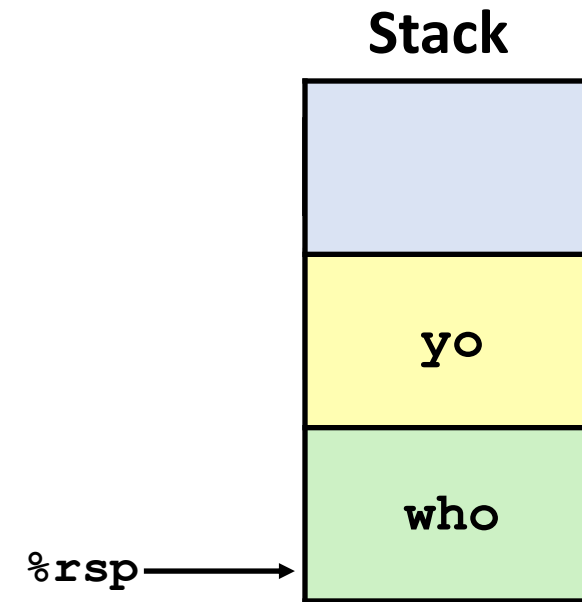
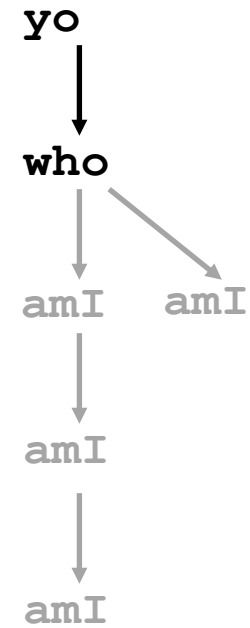
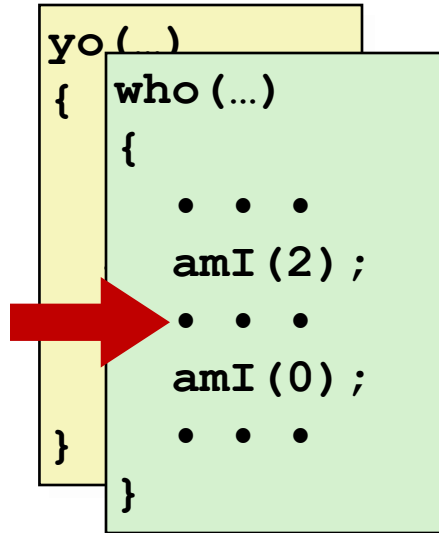
# Example



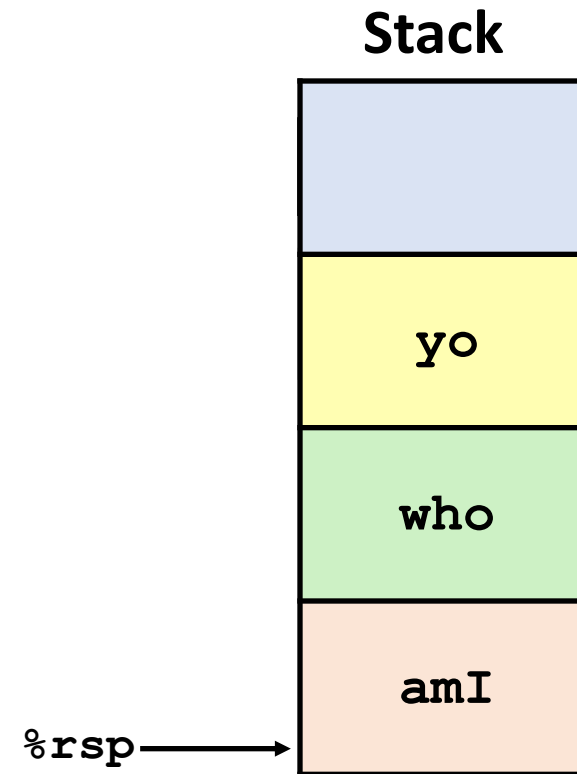
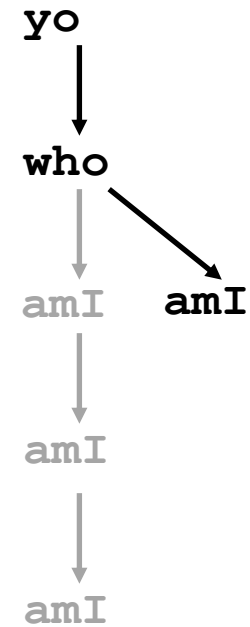
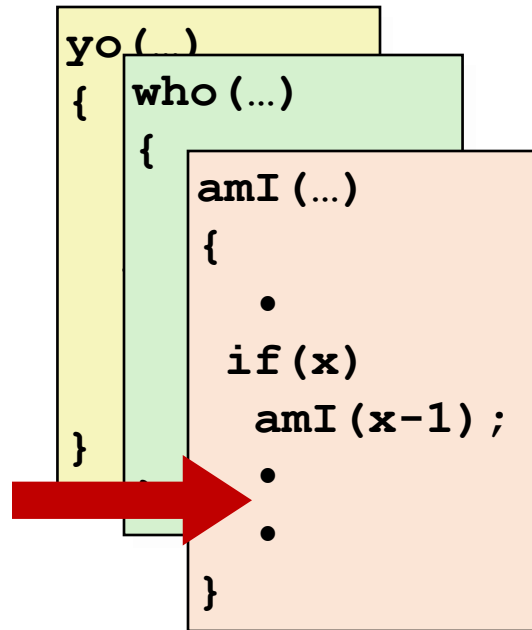
# Example



# Example

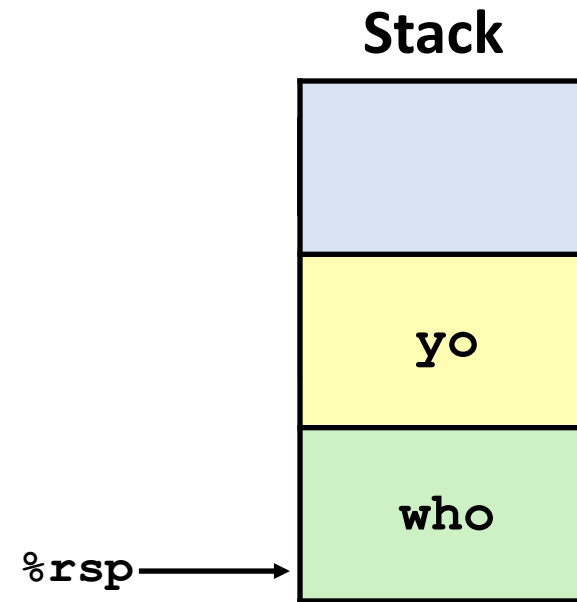
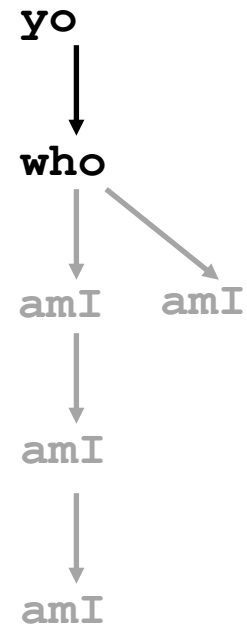
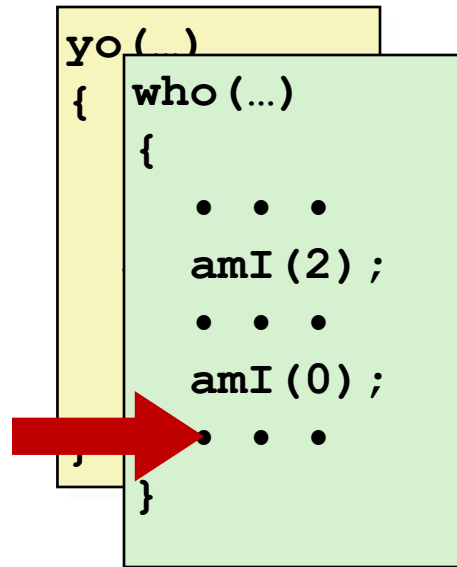


# Example

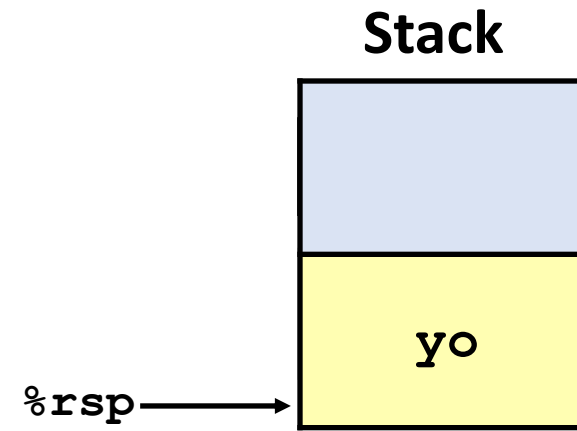
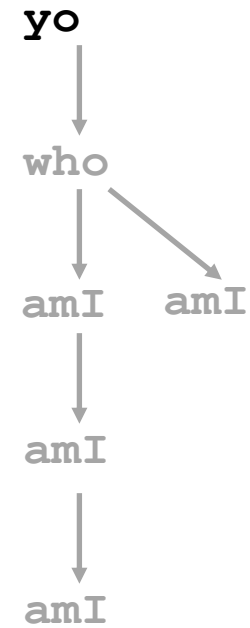
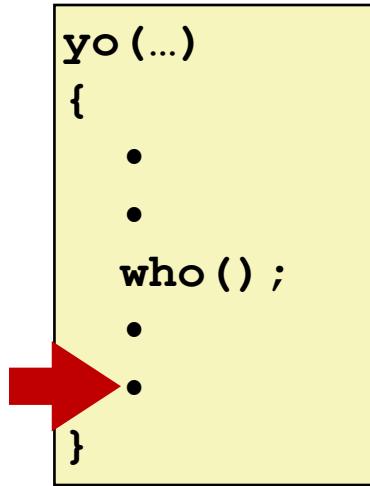




# Example

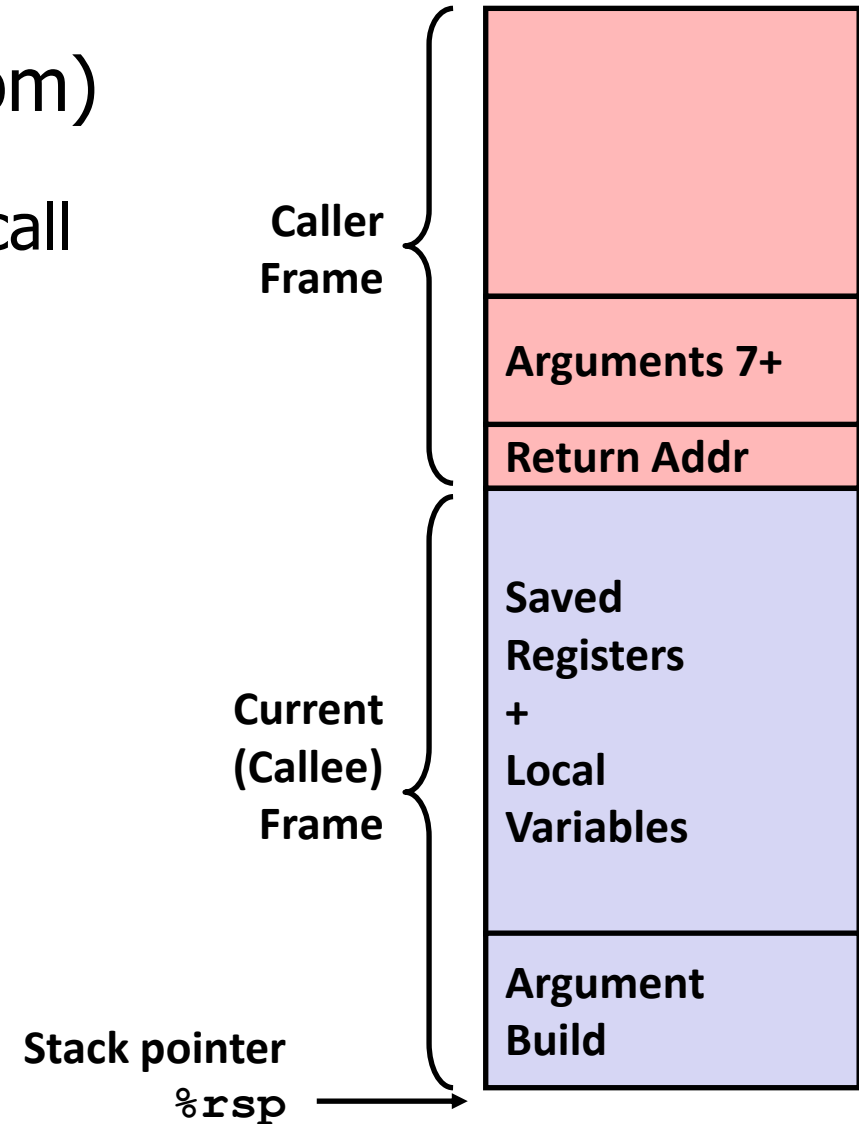


# Example



# x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build":  
Arguments for function we're about to call
  - Local variables  
If we can't keep them in registers  
(too many, or if must be in memory)
  - Saved register context  
(we'll get to that soon)
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



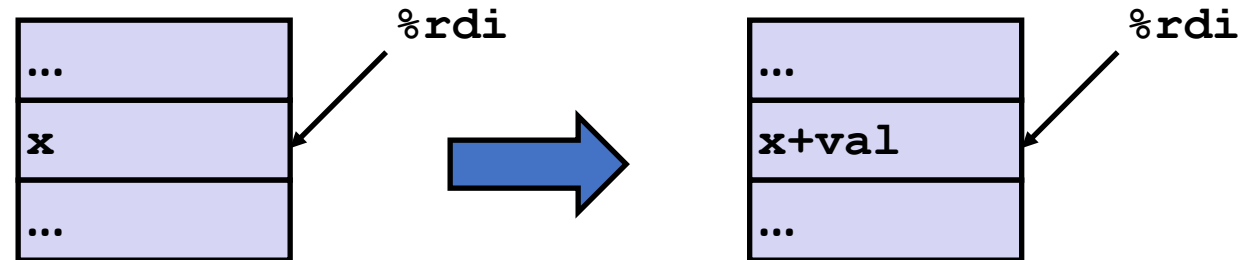
# Example: `incr`

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

```
incr:  
    movq    (%rdi), %rax    # x = *p  
    addq    %rax, %rsi      # y = x+val  
    movq    %rsi, (%rdi)    # *p = y  
    ret
```

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

Memory



# Example: Calling `incr` #1 (local variables)

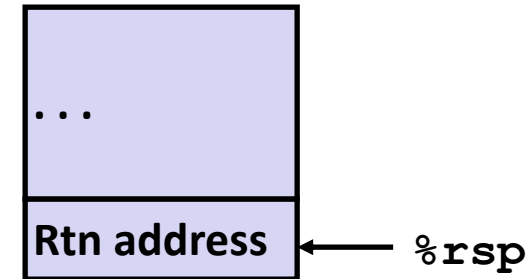
We take `v1`'s address, so must be in memory

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

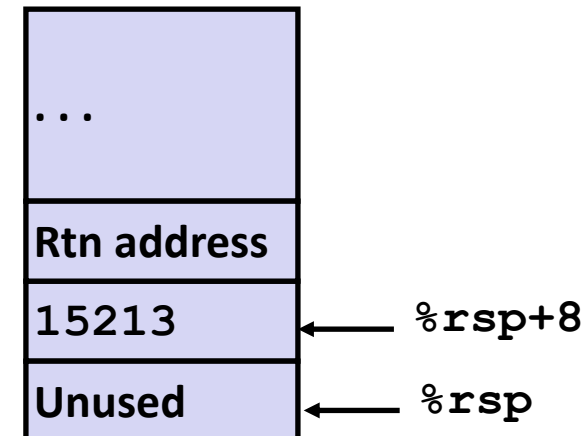
Stack pointer must be multiple of 16

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

## Initial Stack Structure



## Resulting Stack Structure



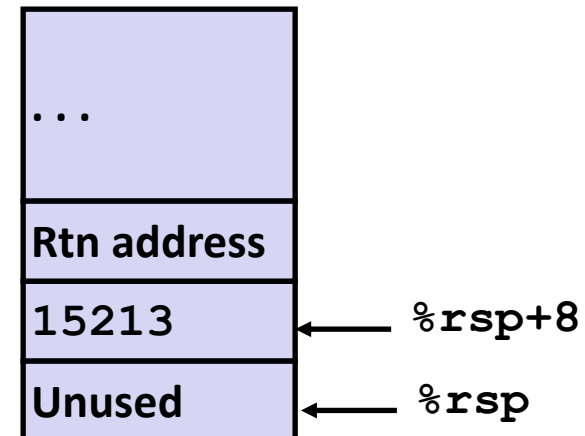
# Example: Calling `incr` #2 (argument build)

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

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

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

## Stack Structure



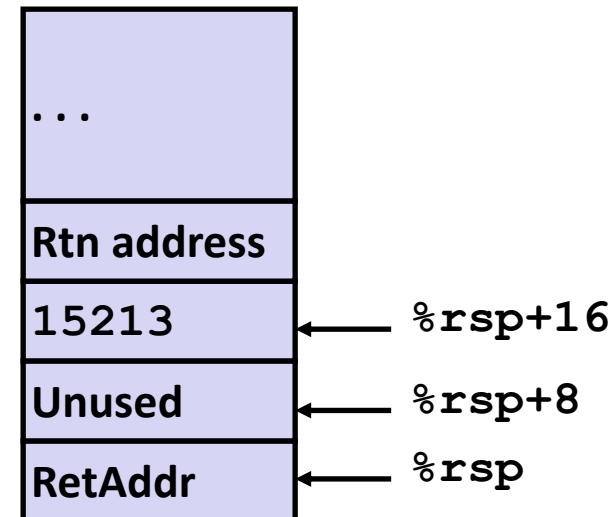
# Example: Calling `incr` #3 (control transfer)

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

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

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

Stack Structure



# Example: executing `incr`

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

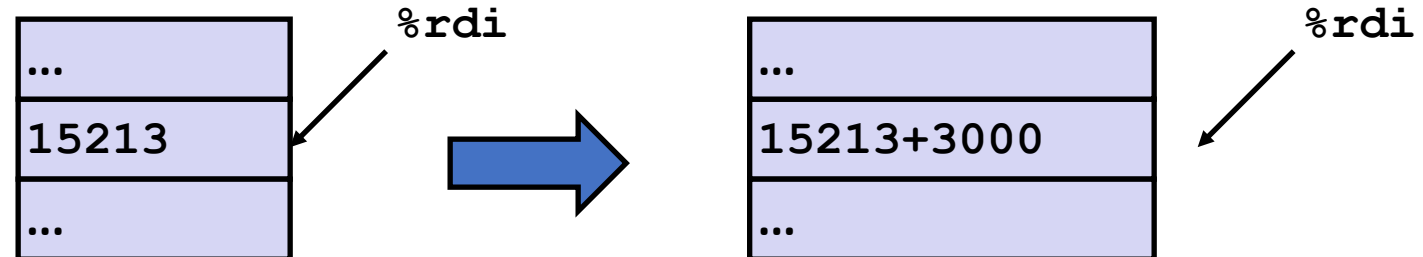
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	Argument <code>val</code> (3000)
%rax	...



Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	18213
%rax	15213 (return value)

Memory



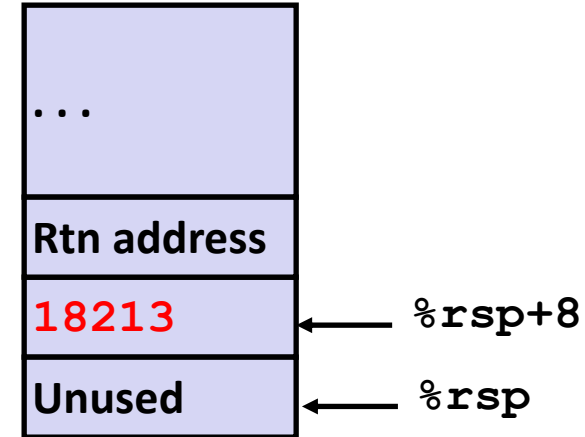


# Example: right after executing `incr`

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

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    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>	18213
<code>%rax</code>	15213

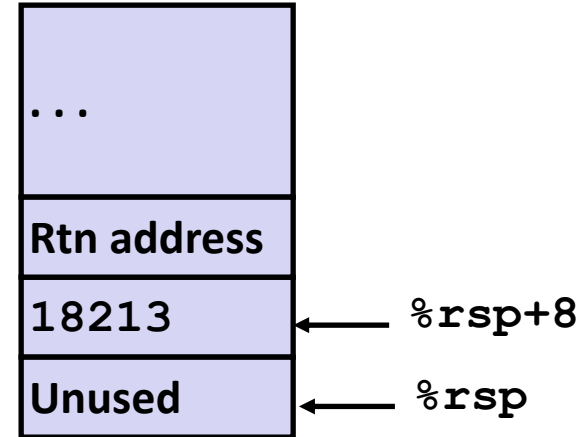
QUIZ: where do we find the return value of `incr`?

# Example: Calling `incr` #4 (cleanup)

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

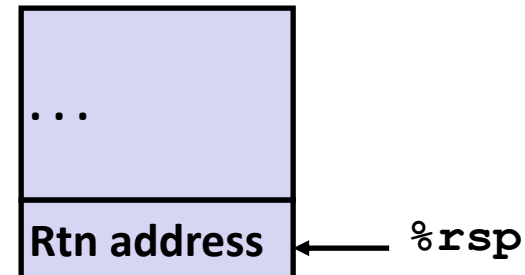
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    → addq   8(%rsp), %rax  
    → addq   $16, %rsp  
    ret
```

Previous stack Structure



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

Updated Stack Structure

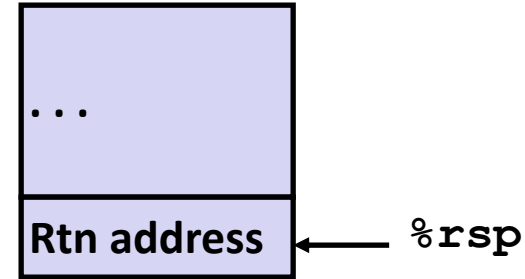


# Example: Calling `incr` #5

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

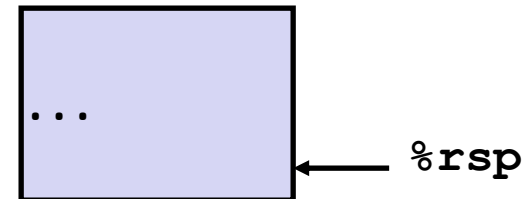
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    → ret
```

Updated Stack Structure



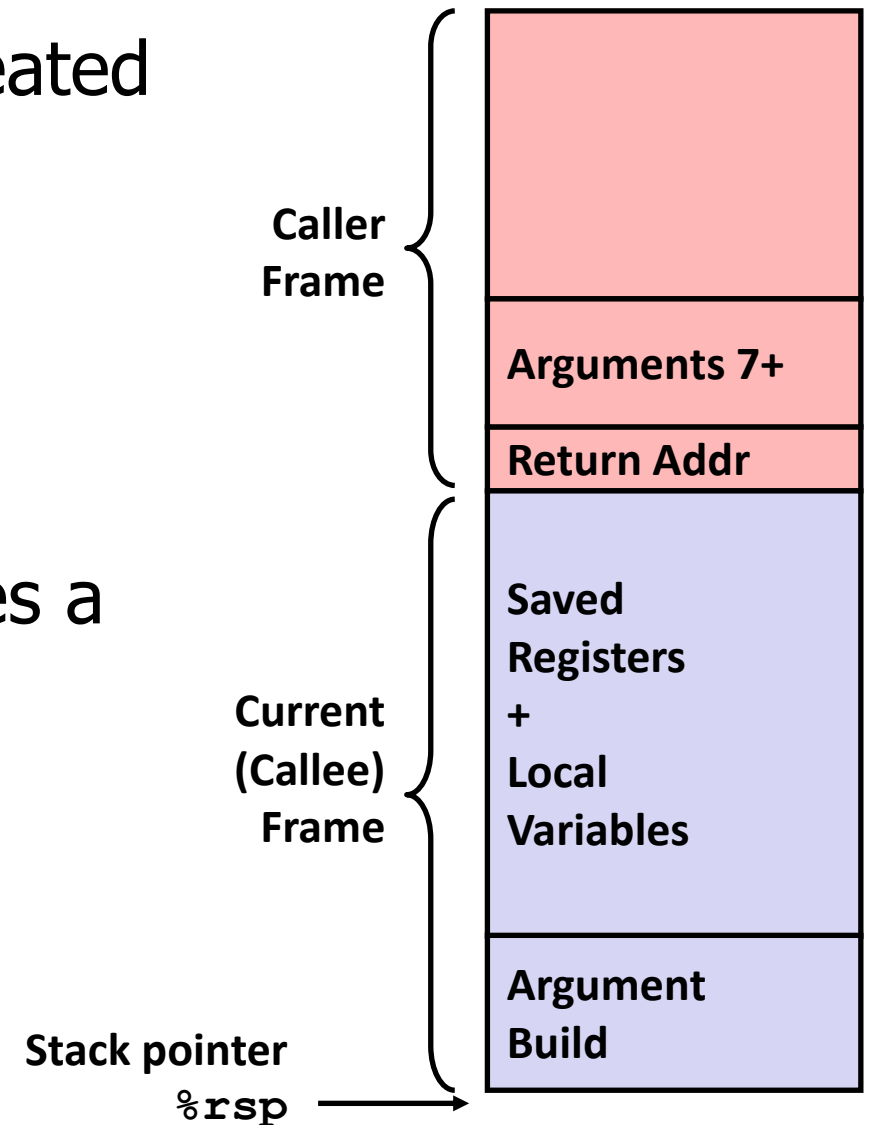
Register	Use(s)
%rax	Return value

Final Stack Structure



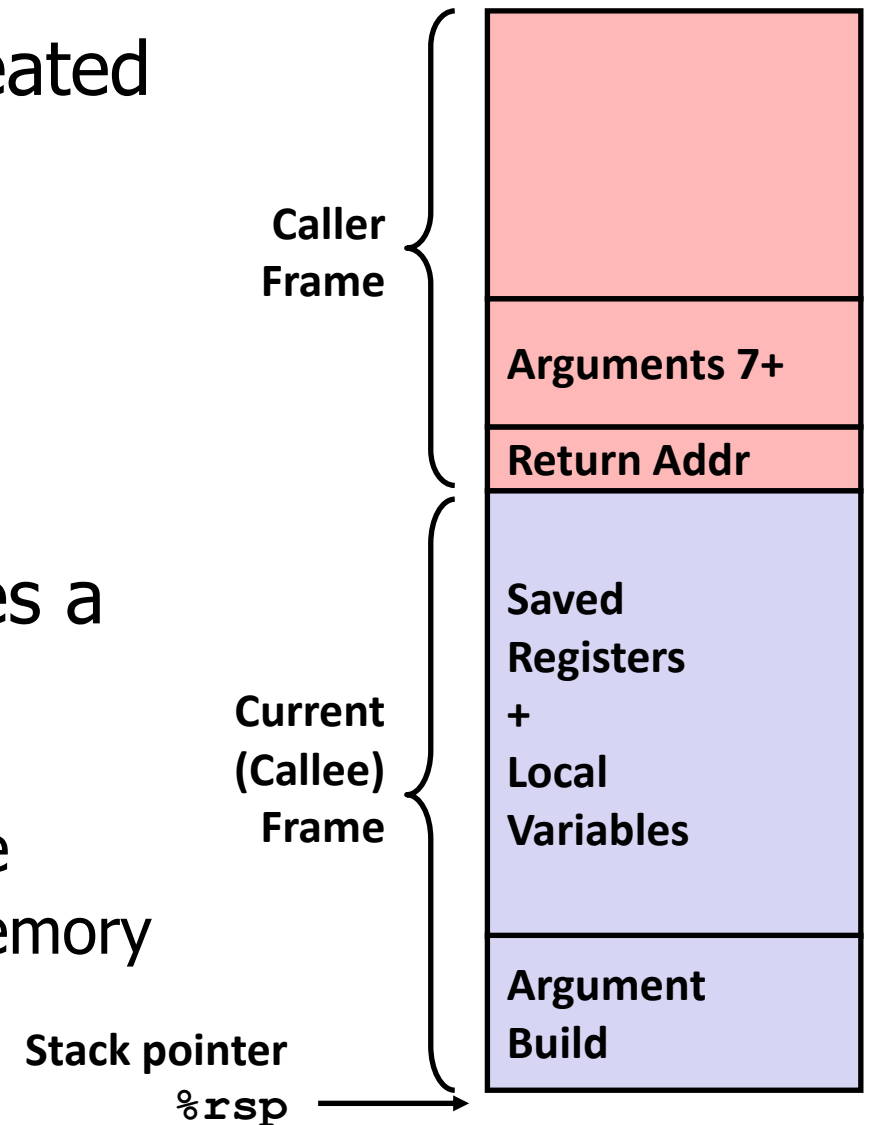
# Break + Open Questions

- What are the initial values of variables created on the stack?
- Is there a limit to how many local variables a function can have?



# Break + Open Questions

- What are the initial values of variables created on the stack?
  - Undefined behavior in C (compiler chooses)
  - Machine just creates a variable in the stack
    - Initial value is whatever was there before
- Is there a limit to how many local variables a function can have?
  - Based on memory limit of the process
  - Stack keeps growing until it runs out of space
    - OS can do lots of tricks to give it more memory



# Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- **Register Saving**
  - Recursion Example

# Register Saving

- Can a function use `%rdx` for temporary storage?

## Caller

```
yo:
    . . .
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    . . .
    ret
```

## Callee

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

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

# Reusing registers

- Problem: registers are shared between functions
  - Callee could overwrite caller's registers by accident
- How does each function know which registers are safe to use?
- Solution:
  - Save original register value to stack
  - Use register as needed
  - Restore original register value from stack
- New question: when should the saving happen? In advance or on demand?



# Saving registers in advance

- New question: who should save the registers, Caller or Callee?
- Attempt 1: Save everything in advance
  - Caller knows which registers it is using
  - Save all registers it is going to need after the call
- Downside: Caller doesn't know what Callee needs
  - Wasted stores to memory if Callee doesn't need those registers

# Saving registers on demand

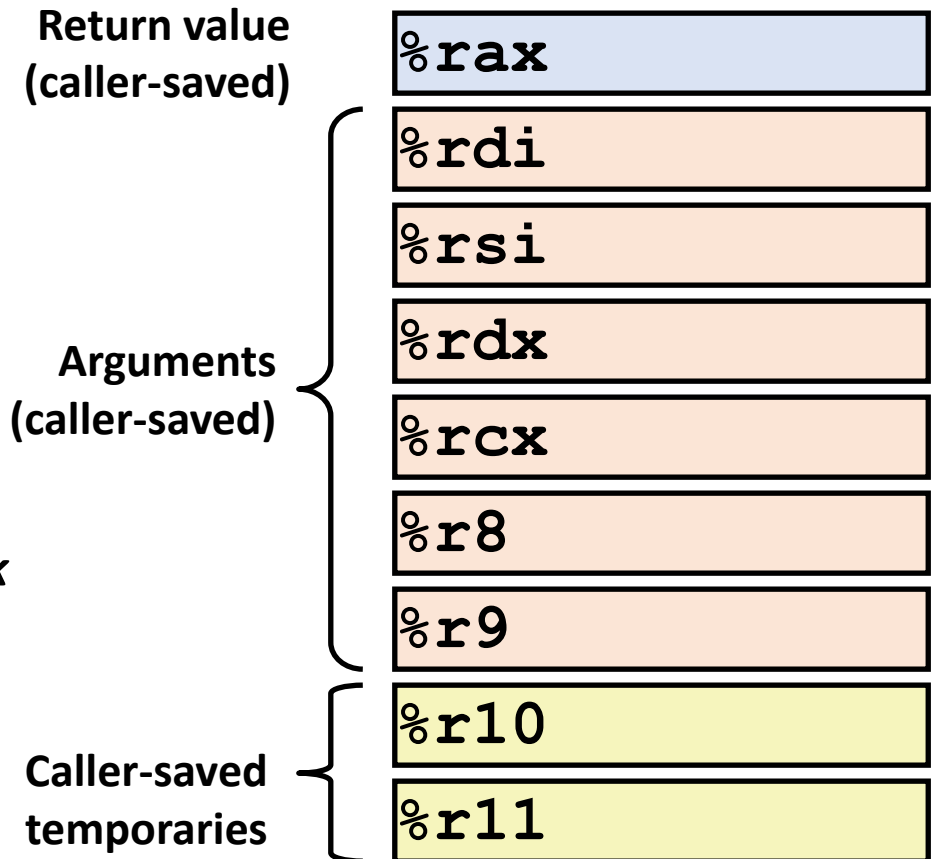
- New question: who should save the registers, Caller or Callee?
- Attempt 2: Save everything on demand
  - Callee knows which registers it is using
  - Save all registers it is going to use at the start of the function
- Downside: Callee doesn't know what Caller was using
  - Wasted stores to memory if Caller wasn't using those registers

# Compromise: some registers in advance, some on demand

- Neither the Caller nor the Callee has perfect knowledge of register availability
- Designate based on register which are saved when
  - Some are saved in advance: Caller saved
  - Some are saved on demand: Callee saved
- Remember: Caller and Callee are just designations for one call event
  - Functions can and do act as both at different times

# x86-64 Linux Register Usage #1 (caller-saved)

- **%rax**
  - Return value
  - Caller-saved
  - **Will** be modified by function we're about to call
- **%rdi, ..., %r9**
  - Arguments
  - Caller-saved
  - Can be modified by function we're about to call
  - If more than 6 arguments, then *pass the rest on the stack*
- **%r10, %r11**
  - Caller-saved
  - Can be modified by function we're about to call

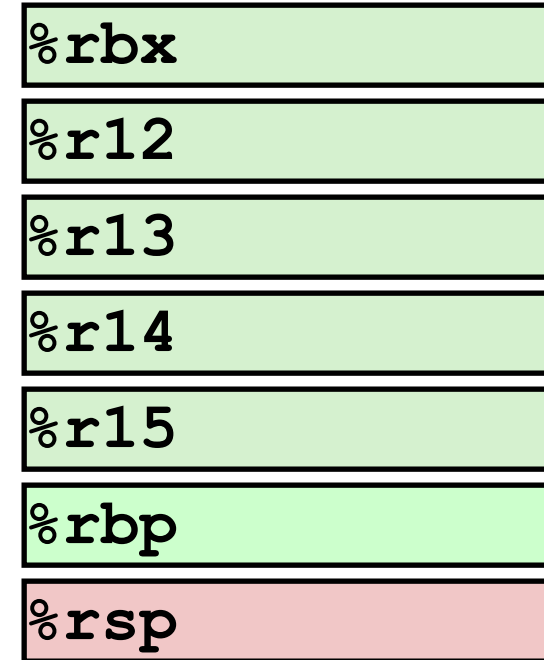


# x86-64 Linux Register Usage #2 (callee-saved)

- **%rbx, %r12, %r13, %r14**

- Callee-saved
- Callee must save & restore

Callee-saved  
Temporaries



Special

- **%rsp**

- Special form of callee-saved
- Restored to original value upon exit from procedure
  - Stack frame is removed

# x86-64 Integer Registers: Usage Conventions

Caller Saved

In advance

Callee saved

On demand

**%rax** Return value

**%rbx** Callee saved

**%rcx** Argument #4

**%rdx** Argument #3

**%rsi** Argument #2

**%rdi** Argument #1

**%rsp** Stack pointer

**%rbp** Callee saved

**%r8** Argument #5

**%r9** Argument #6

**%r10** Caller saved

**%r11** Caller Saved

**%r12** Callee saved

**%r13** Callee saved

**%r14** Callee saved

**%r15** Callee saved

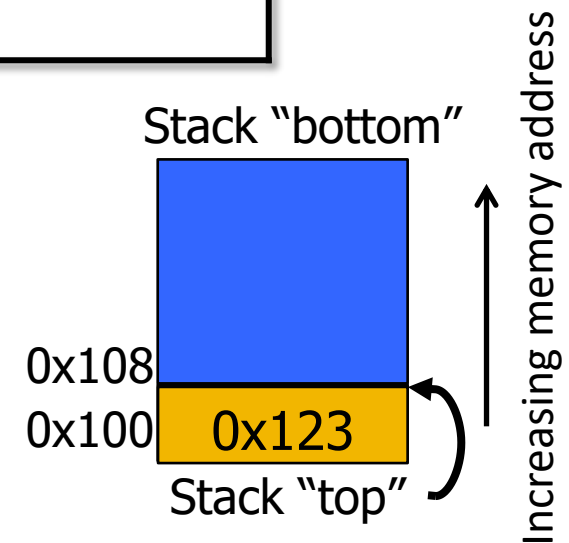
# Push and Pop instructions

Instruction	Effect	Description
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Store S onto the stack
<code>popq D</code>	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$	Retrieve D from the stack

- Example:

`%rax = 0x123, %rdx = 0x0, %rsp = 0x108`

<code>pushq %rax</code>	<code>%rsp = 0x100</code>
<code>popq %rdx</code>	<code>%rdx = 0x123; %rsp = 0x108</code>



- Remember, stack is just memory

- Can also use memory moves and modify `%rsp` manually!

# Register Saving Example #1

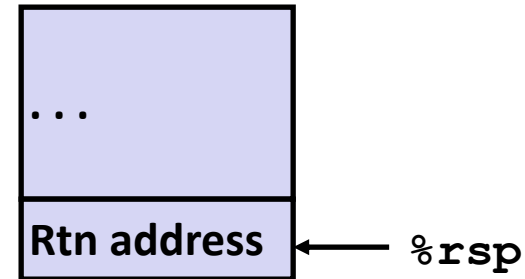
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

↑ Still need **x** after the call!

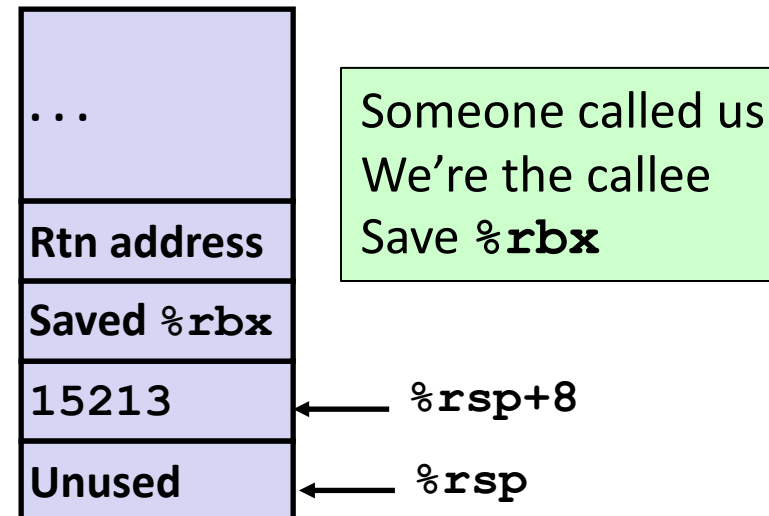
**%rbx** is callee-save (on demand)

```
call_incr2:  
→ pushq    %rbx  
→ subq     $16, %rsp  
    movq     %rdi, %rbx  
    movq     $15213, 8(%rsp)  
    movq     $3000, %rsi  
    leaq     8(%rsp), %rdi  
    call     incr  
    addq     %rbx, %rax  
    addq     $16, %rsp  
    popq     %rbx  
    ret
```

## Initial Stack Structure



## Resulting Stack Structure





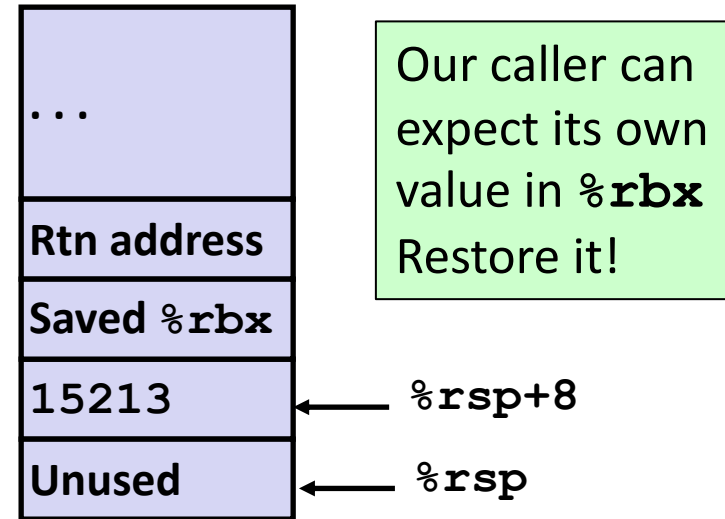
# Register Saving Example #2

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

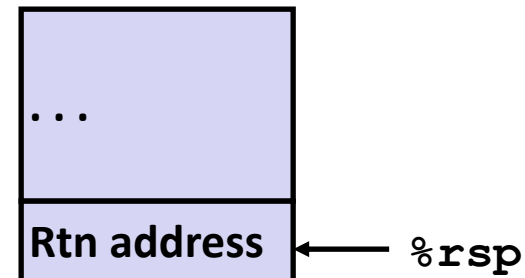
**%rbx** is callee-save (on demand)

```
call_incr2:  
    pushq    %rbx  
    subq     $16, %rsp  
    movq     %rdi, %rbx  
    movq     $15213, 8(%rsp)  
    movq     $3000, %rsi  
    leaq     8(%rsp), %rdi  
    call     incr  
    addq     %rbx, %rax  
    addq     $16, %rsp  
    popq     %rbx  
    ret
```

## Resulting Stack Structure



## Pre-return Stack Structure



# Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- **Register Saving**
  - **Recursion Example**

# 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:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Note: `rep` instruction inserted as no-op. You can ignore it.

# Recursive Function Base Case

```
/* 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
%rdi	x	Argument
%rax	Return value	Return value

pcount\_r:

```
movq    $0, %rax  
testq   %rdi, %rdi  
je       .L6
```

```
pushq   %rbx  
movq    %rdi, %rbx  
andq    $1, %rbx  
shrq    %rdi # (by 1)  
callq   pcount_r  
addq    %rbx, %rax  
popq    %rbx
```

.L6:

```
rep; ret
```

# 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);
}
```

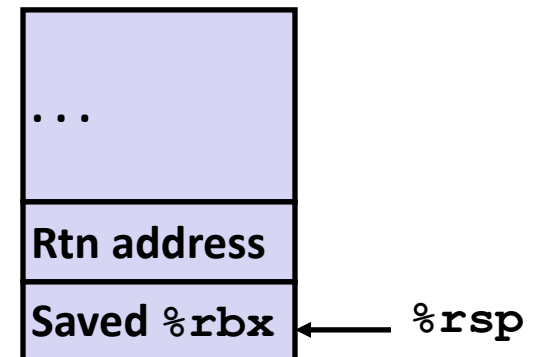
Register	Use(s)	Type
%rdi	x	Argument

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
    rep; ret
```



# 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);
}
```

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

# 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:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   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:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   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);
}
```

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

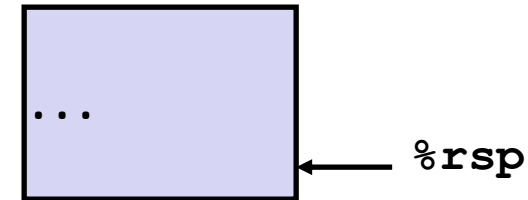
pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
```

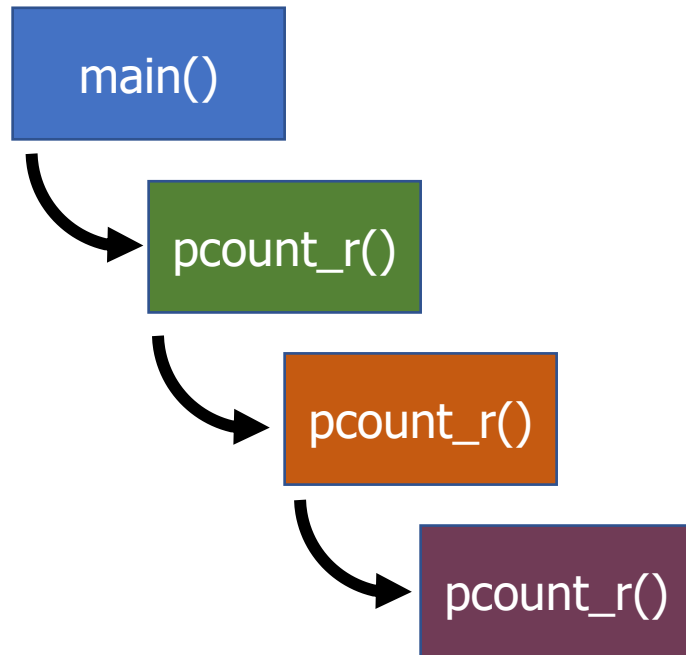
```
    popq    %rbx
```

.L6:

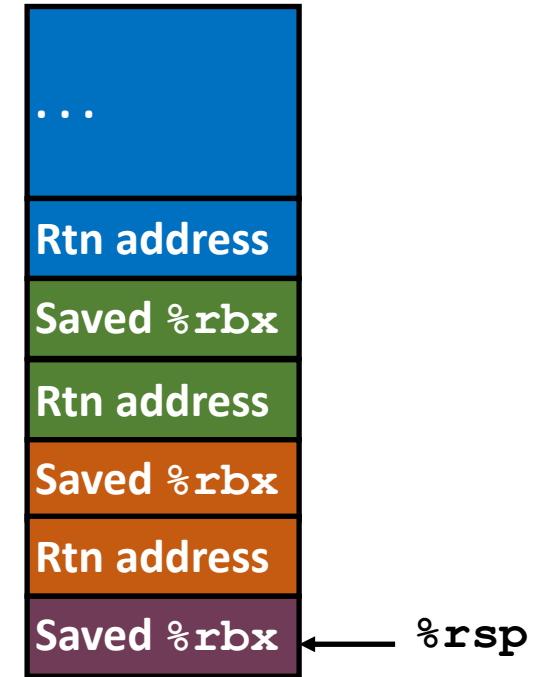
```
    rep; ret
```



# Example three recursions in



Executing, but has not yet called `pcount_r()` again



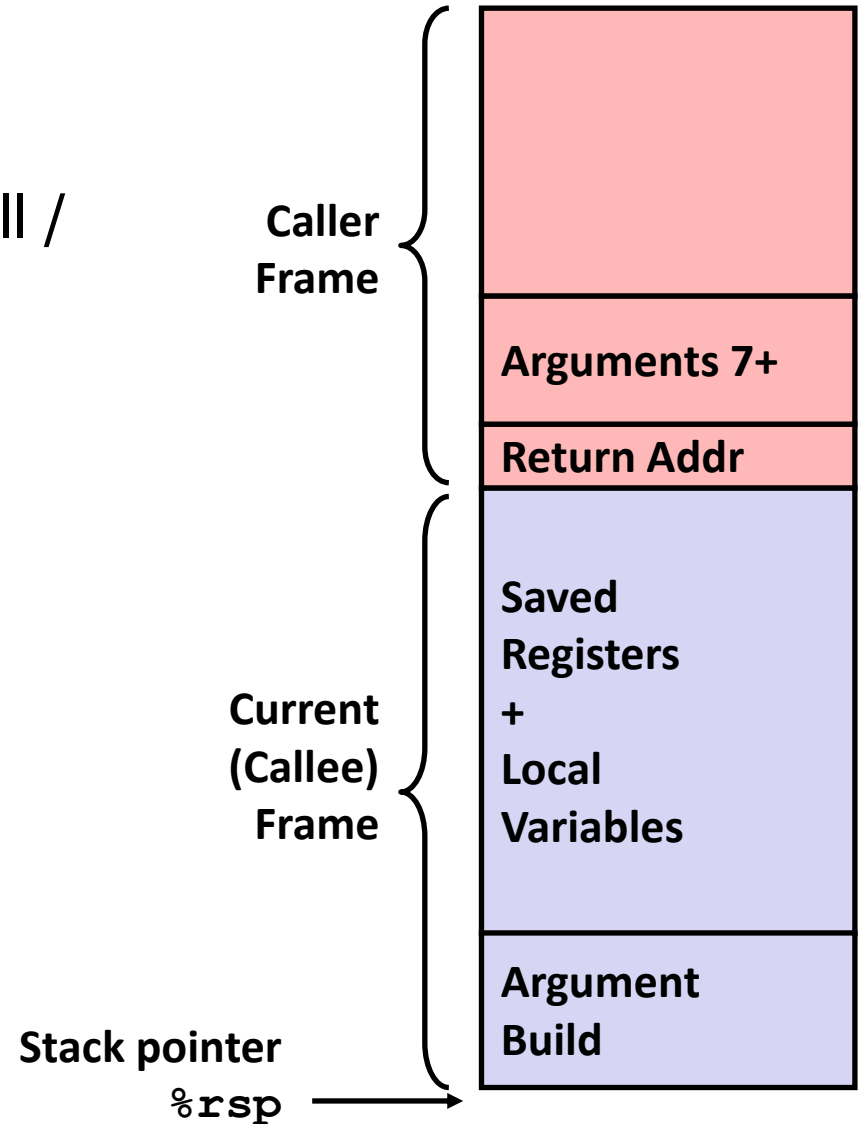
# x86-64 Procedure Summary

- Important Points

- A stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P
- The stack makes recursion work

- Calling convention

- Caller-saved registers saved in advance before call
- Put arguments in registers (1-6)
- Put further arguments on top of stack (7+)
- Put return address on top of stack
- Callee can safely store values in local stack frame and in callee-saved registers (after saving them)
- Result return in `%rax` and restore callee-saved registers before returning



# Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- Register Saving
  - Recursion Example

# Outline

- Bonus: Stack Frame Example

# x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void
swap_ele_su(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq   %esi, %rax
    leaq    8(%rdi, %rax, 8), %rbx
    leaq    (%rdi, %rax, 8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

# Understanding x86-64 Stack Frame

swap ele su:

<b>movq</b> <b>%rbx, -16(%rsp)</b>	# Save %rbx
<b>movq</b> <b>%rbp, -8(%rsp)</b>	# Save %rbp
<b>subq</b> <b>\$16, %rsp</b>	# Allocate stack frame
<b>movslq</b> <b>%esi, %rax</b>	# Extend i
<b>leaq</b> <b>8(%rdi, %rax, 8), %rbx</b>	# &a[i+1] (callee save)
<b>leaq</b> <b>(%rdi, %rax, 8), %rbp</b>	# &a[i] (callee save)
<b>movq</b> <b>%rbx, %rsi</b>	# 2 <sup>nd</sup> argument
<b>movq</b> <b>%rbp, %rdi</b>	# 1 <sup>st</sup> argument
<b>call</b> <b>swap</b>	
<b>movq</b> <b>(%rbx), %rax</b>	# Get a[i+1]
<b>imulq</b> <b>(%rbp), %rax</b>	# Multiply by a[i]
<b>addq</b> <b>%rax, sum(%rip)</b>	# Add to sum
<b>movq</b> <b>(%rsp), %rbx</b>	# Restore %rbx
<b>movq</b> <b>8(%rsp), %rbp</b>	# Restore %rbp
<b>addq</b> <b>\$16, %rsp</b>	# Deallocate frame
<b>ret</b>	

# Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```

`%rsp` → rtn addr

```
subq    $16, %rsp         # Allocate stack frame
```

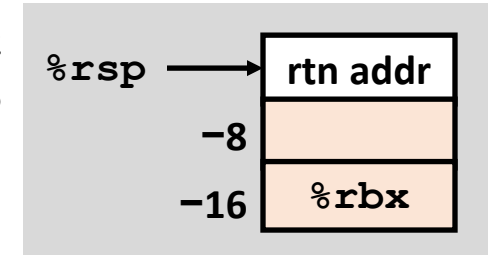
● ● ●

```
movq    (%rsp), %rbx      # Restore %rbx
movq    8(%rsp), %rbp     # Restore %rbp
addq    $16, %rsp         # Deallocate frame
```



# Understanding x86-64 Stack Frame

→ **movq** **%rbx, -16(%rsp)** # Save %rbx  
**movq** **%rbp, -8(%rsp)** # Save %rbp



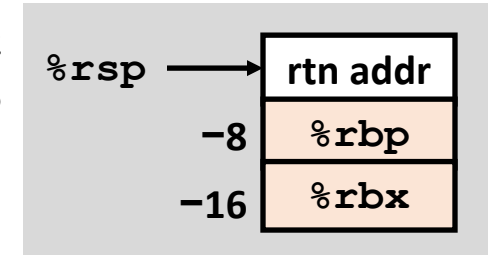
**subq** **\$16, %rsp** # Allocate stack frame

• • •

**movq** **(%rsp), %rbx** # Restore %rbx  
**movq** **8(%rsp), %rbp** # Restore %rbp  
**addq** **\$16, %rsp** # Deallocate frame

# Understanding x86-64 Stack Frame

→ `movq %rbx, -16(%rsp)` # Save %rbx  
→ `movq %rbp, -8(%rsp)` # Save %rbp



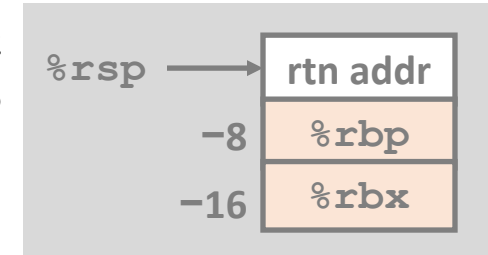
`subq $16, %rsp` # Allocate stack frame

• • •

`movq (%rsp), %rbx` # Restore %rbx  
`movq 8(%rsp), %rbp` # Restore %rbp  
`addq $16, %rsp` # Deallocate frame

# Understanding x86-64 Stack Frame

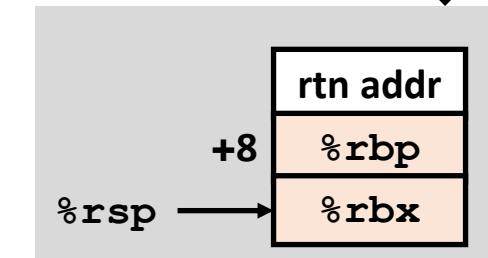
```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



→ **subq \$16, %rsp**

# Allocate stack frame

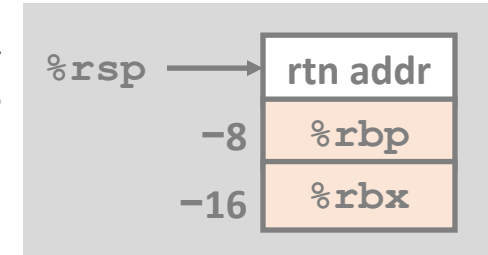
• • •



```
movq    (%rsp), %rbx       # Restore %rbx
movq    8(%rsp), %rbp      # Restore %rbp
addq    $16, %rsp          # Deallocate frame
```

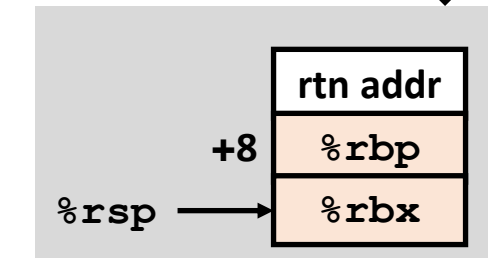
# Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



```
subq    $16, %rsp         # Allocate stack frame
```

...



```
movq    (%rsp), %rbx
movq    8(%rsp), %rbp
addq    $16, %rsp
```

```
# Restore %rbx
# Restore %rbp
# Deallocate frame
```

