Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

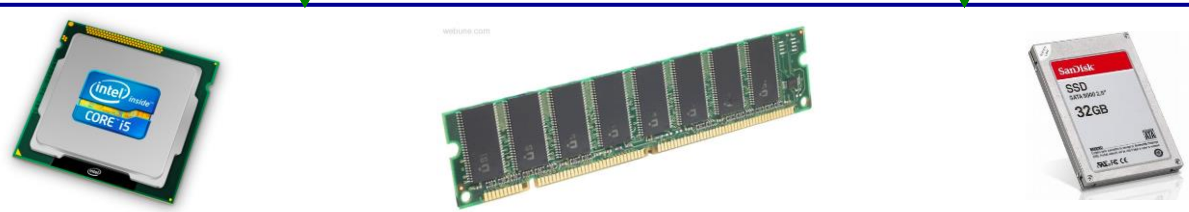
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

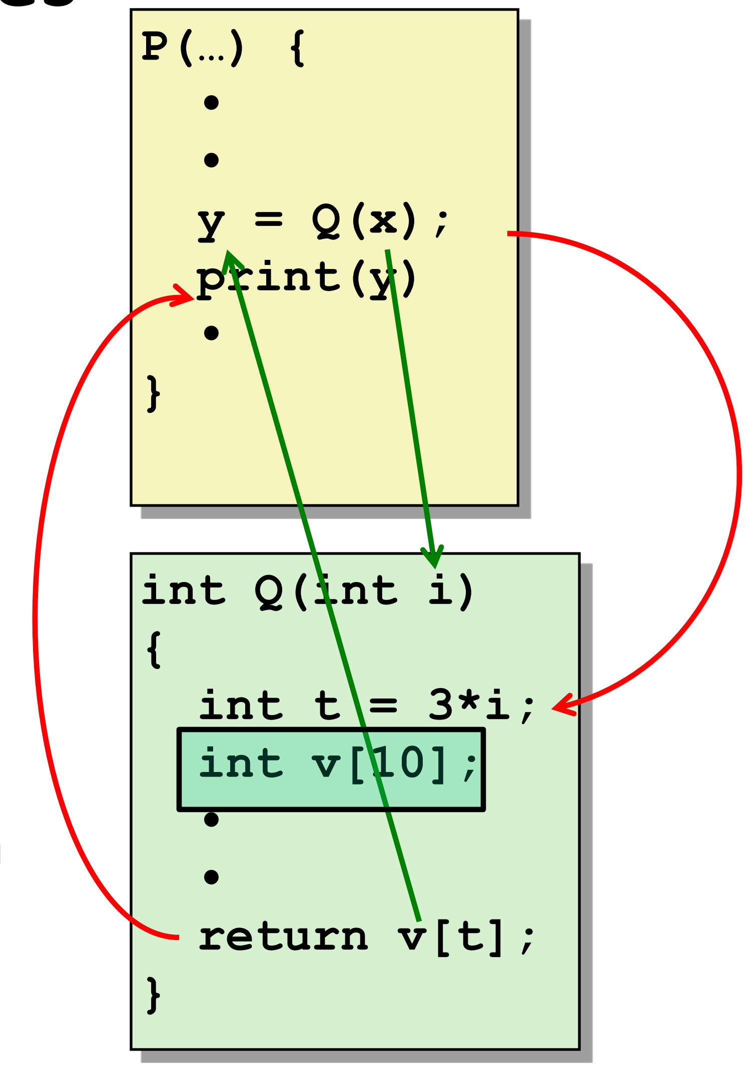
Computer system:



Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

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**



Questions to answer about Procedures

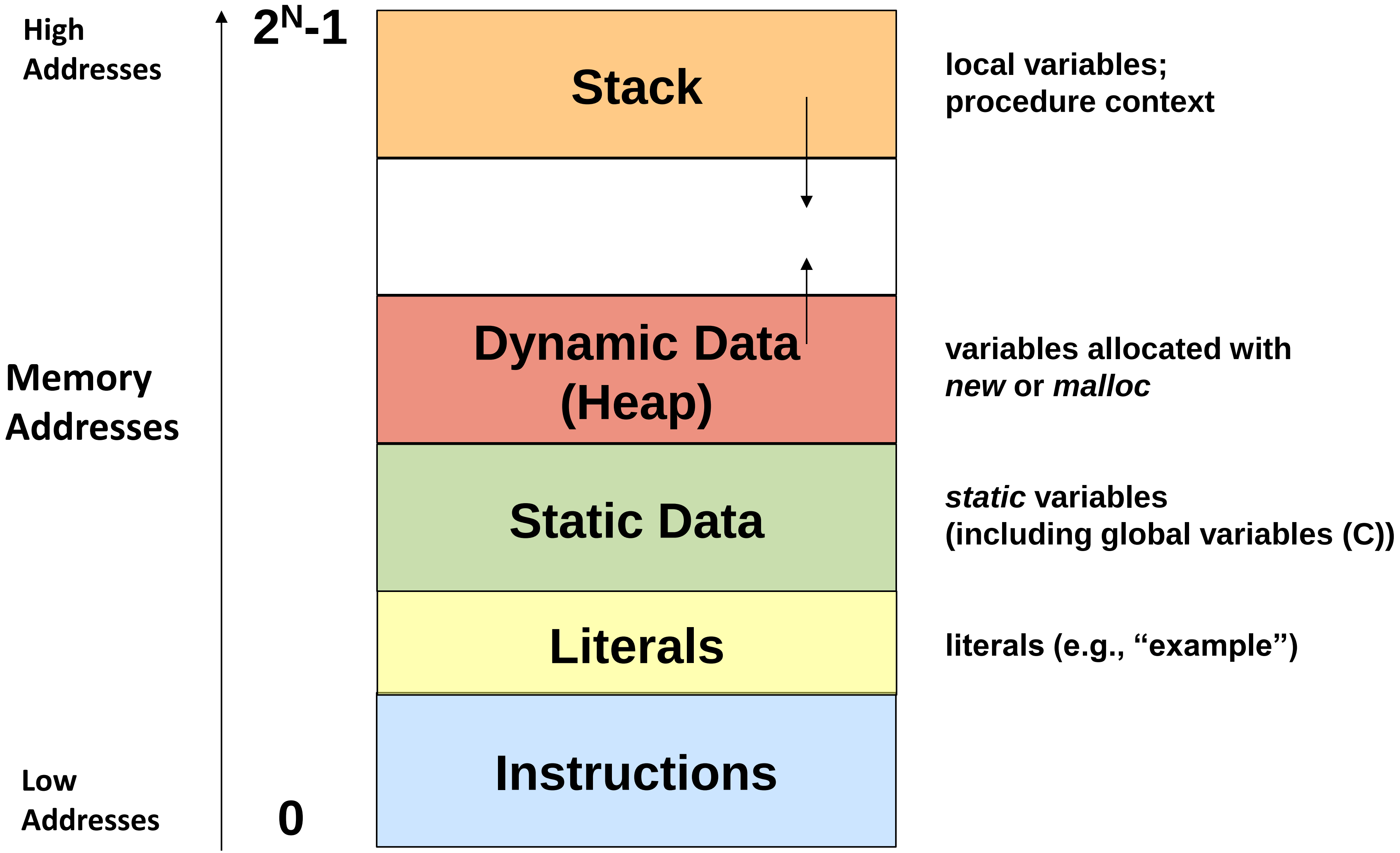
- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer some of these questions, we need a *call stack* …

Today

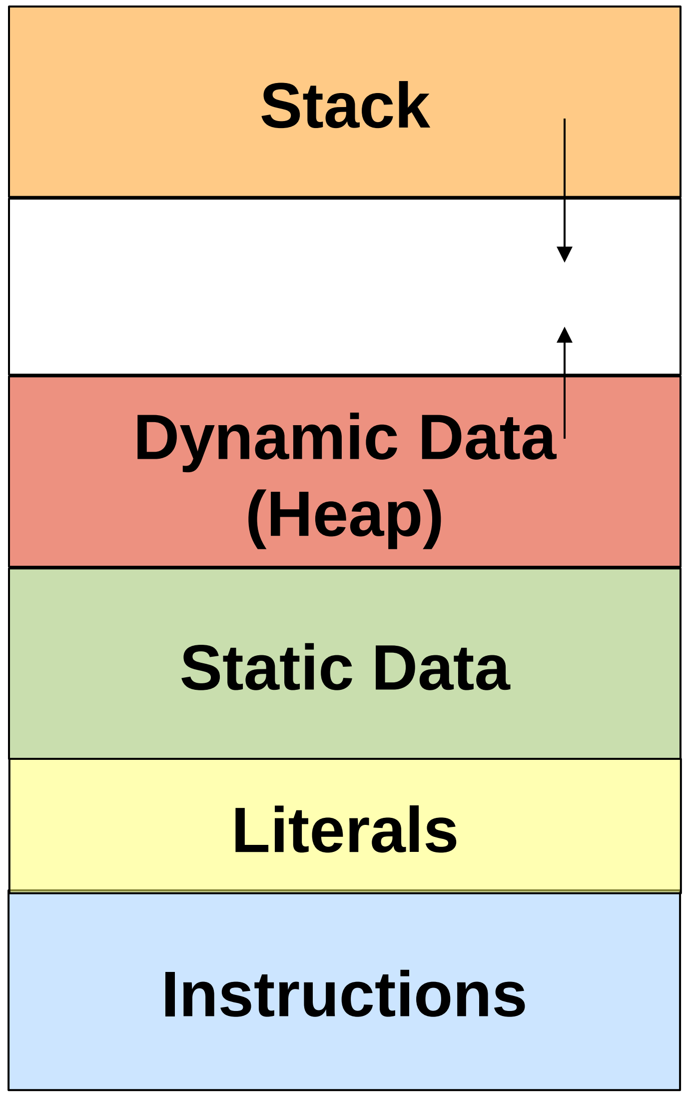
- **Procedures**
  - **Stack Structure**
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion

Memory Layout



segmentation faults?

Memory Layout

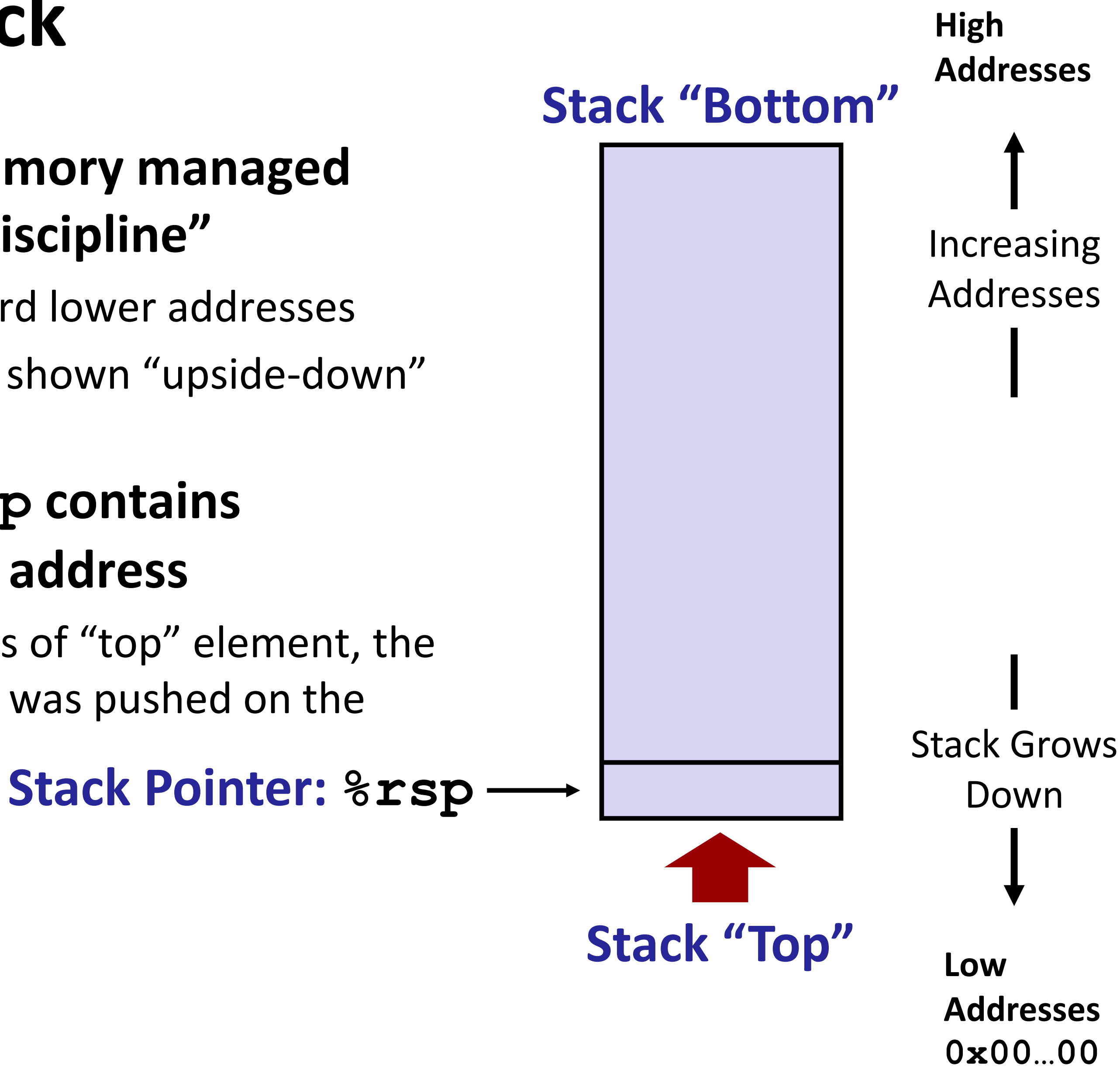


writable; not executable	Stack	Managed “automatically” (by compiler)
writable; not executable	Dynamic Data (Heap)	Managed by programmer
writable; not executable	Static Data	Initialized when process starts
read-only; not executable	Literals	Initialized when process starts
read-only; executable	Instructions	Initialized when process starts

x86-64 Stack

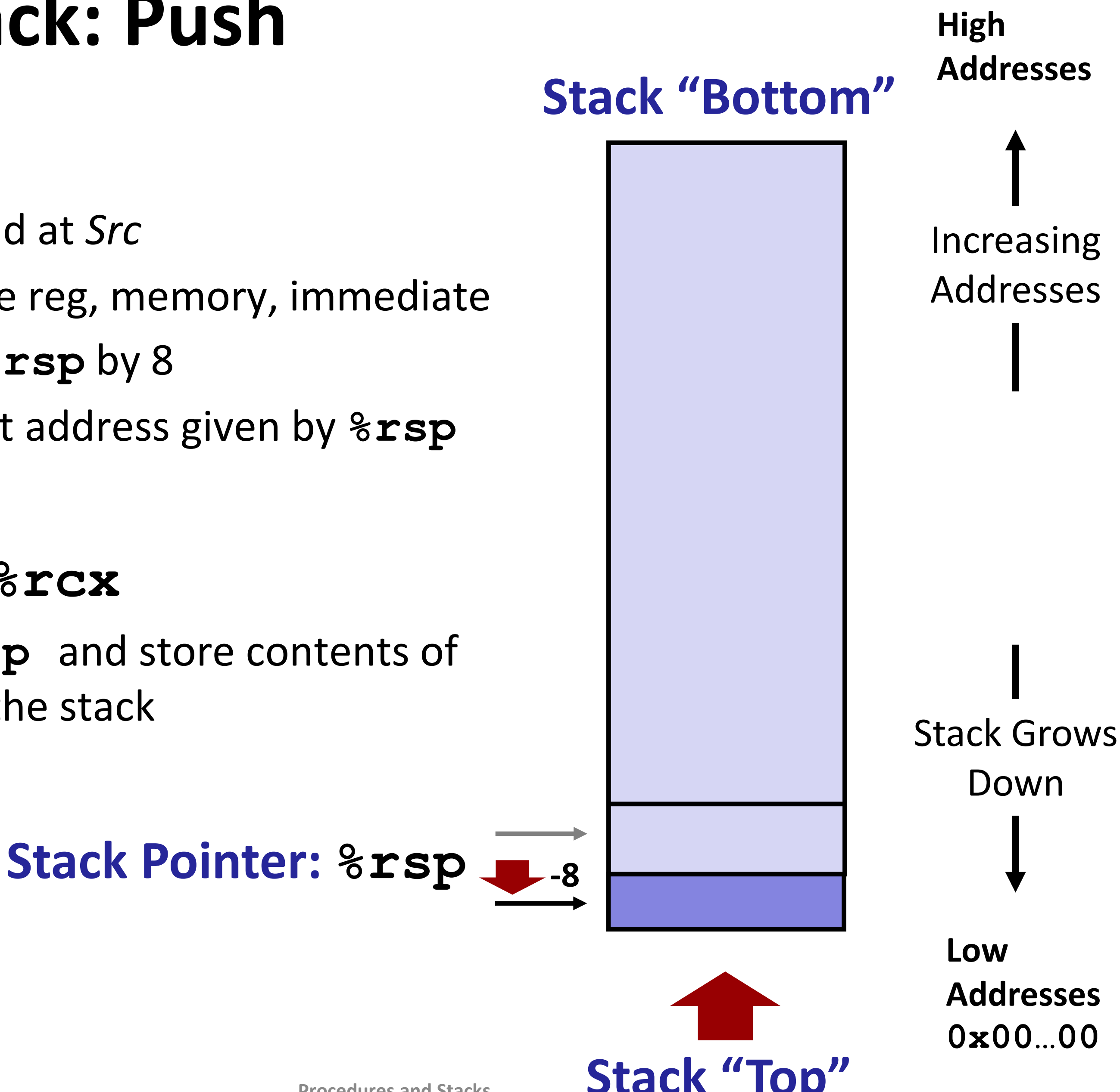
- **Region of memory managed with stack "discipline"**
  - Grows toward lower addresses
  - Customarily shown "upside-down"

- **Register `%rsp` contains lowest stack address**
  - %rsp = address of "top" element, the last thing that was pushed on the stack



x86-64 Stack: Push

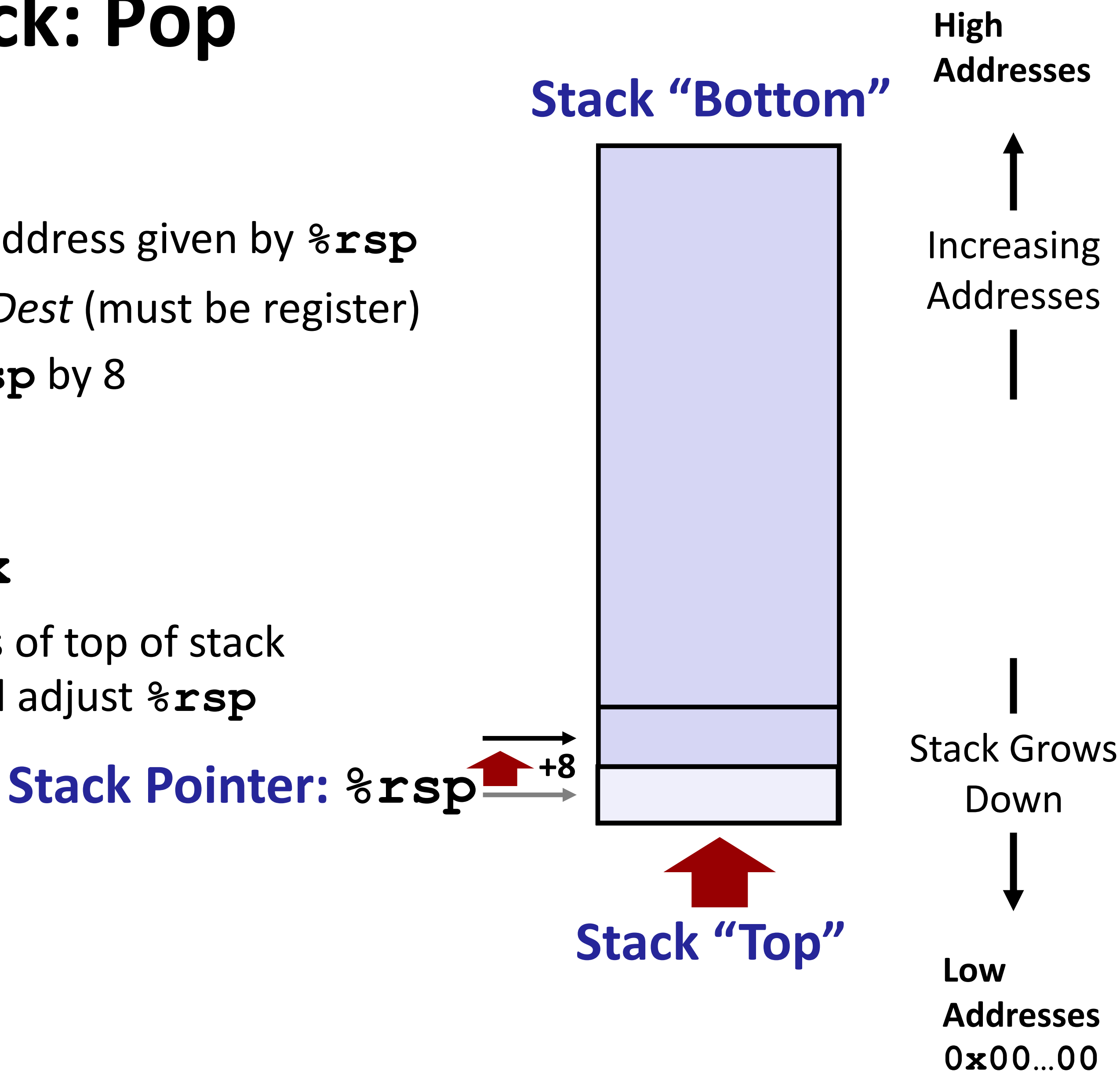
- `pushq` *Src*
  - Fetch operand at *Src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- **Example:**
  - `pushq %rcx`
  - Adjust `%rsp` and store contents of `%rcx` on the stack



x86-64 Stack: Pop

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

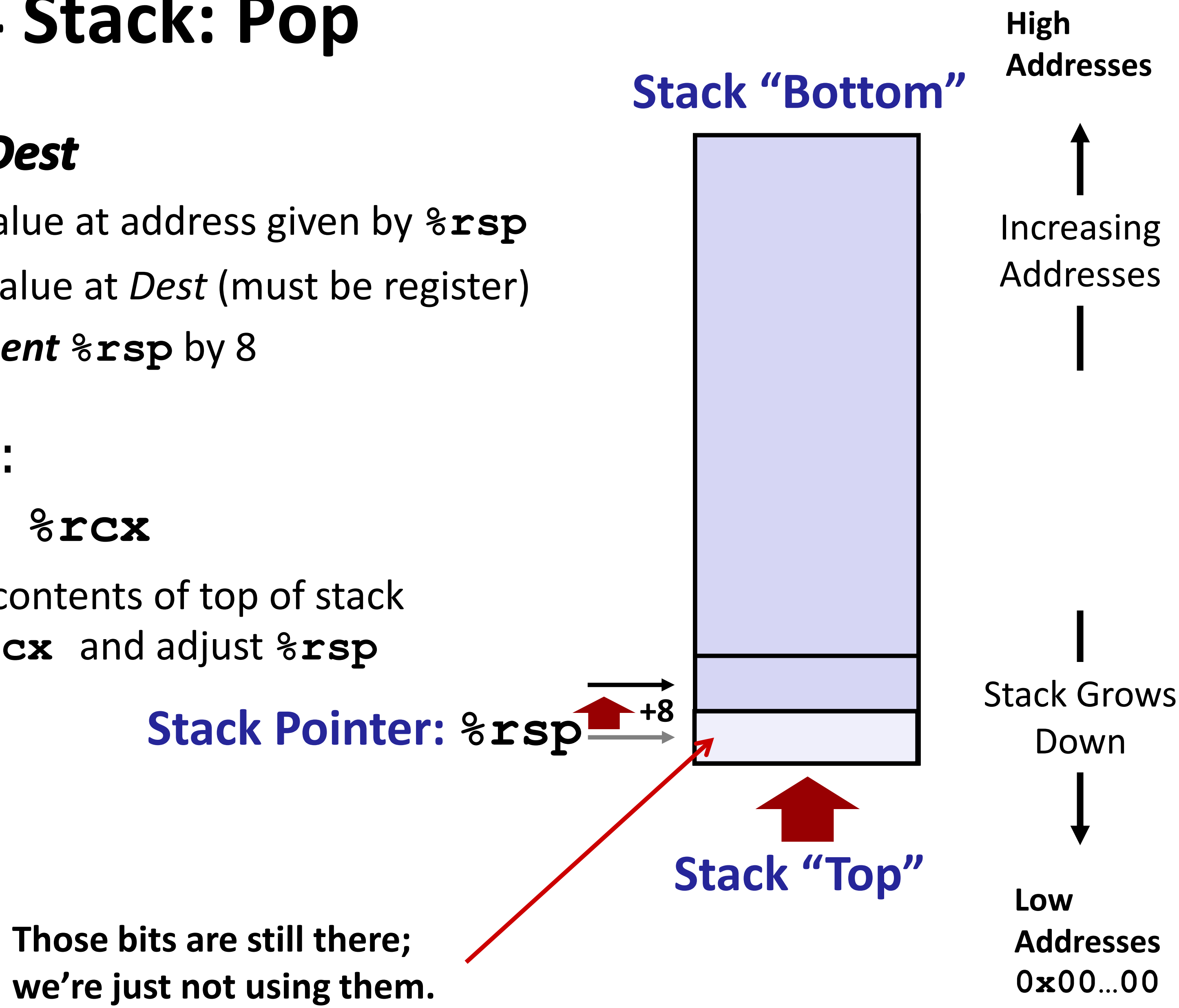
- **Example:**
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



x86-64 Stack: Pop

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

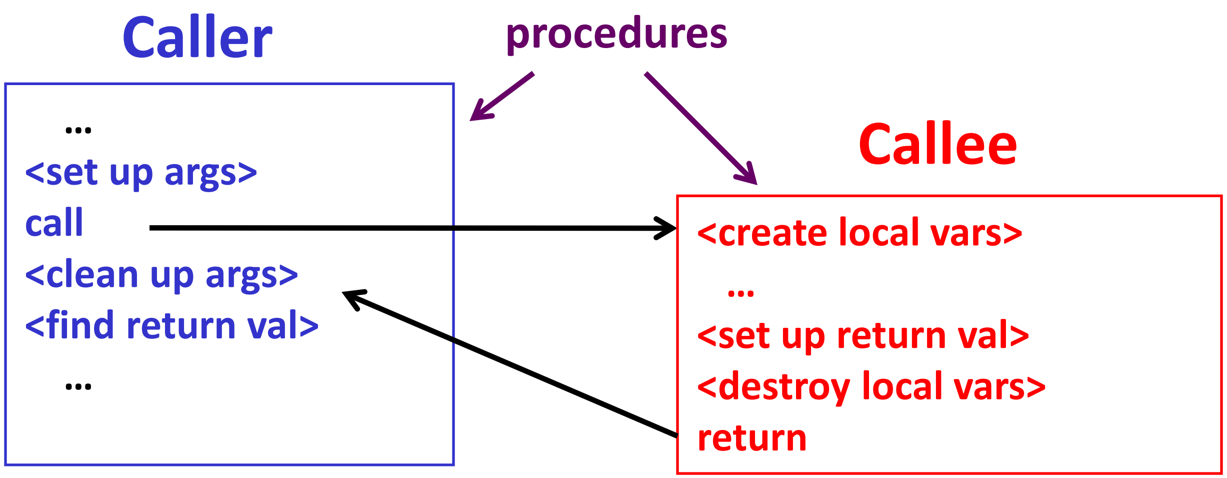
- **Example:**
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Today

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

Procedure Call Overview



- **Callee must know where to find args**
- **Callee must know where to find "return address"**
- **Caller must know where to find return val**
- **Caller and Callee run on same CPU → use the same registers**
  - So how do we deal with register reuse?
- Not all steps are always performed (e.g. no clean up of args or no need to find return val if not used)

Procedure Call Overview

Caller

…
<save regs>
<set up args>
call
<clean up args>
<restore regs>
<find return val>
…

Callee

<save regs>
<create local vars>
…
<set up return val>
<destroy local vars>
<restore regs>
return

- **The <u>convention</u> of where to leave/find things is called the calling convention (or procedure call linkage).**
  - Details vary between systems
  - We will see the convention for <u>x86-64/Linux</u> in detail
  - What could happen if our program didn't follow these conventions?

Code Examples

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

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

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

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

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call:** `call` ***`label`***
  1. Push return address on stack (*why?, and which exact address?*)
  2. Jump to ***`label`***

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  1. Push return address on stack
  2. Jump to *`label`*
- **Return address:**
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

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

Return address = **0x400549**

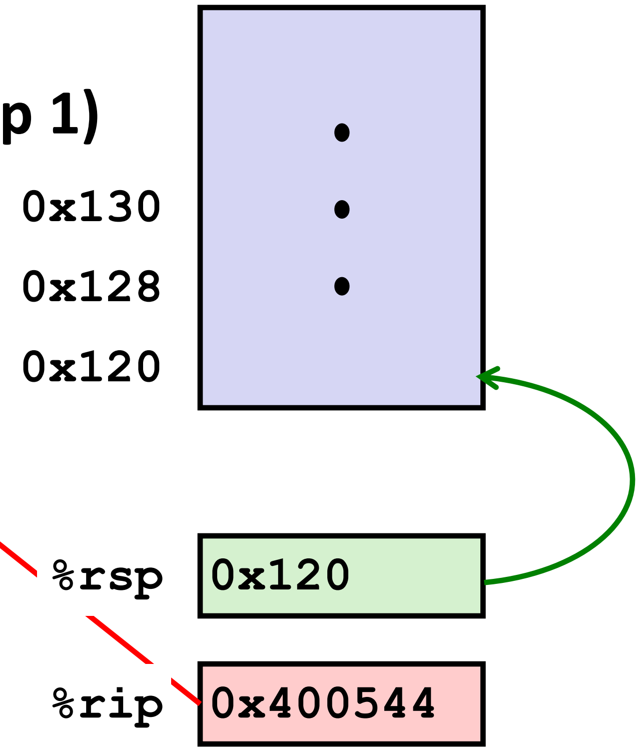
- **Procedure return: `ret`**
  1. Pop return address from stack
  2. Jump to address

next instruction just happens to be an mov, but could be anything

Procedure Call Example (step 1)

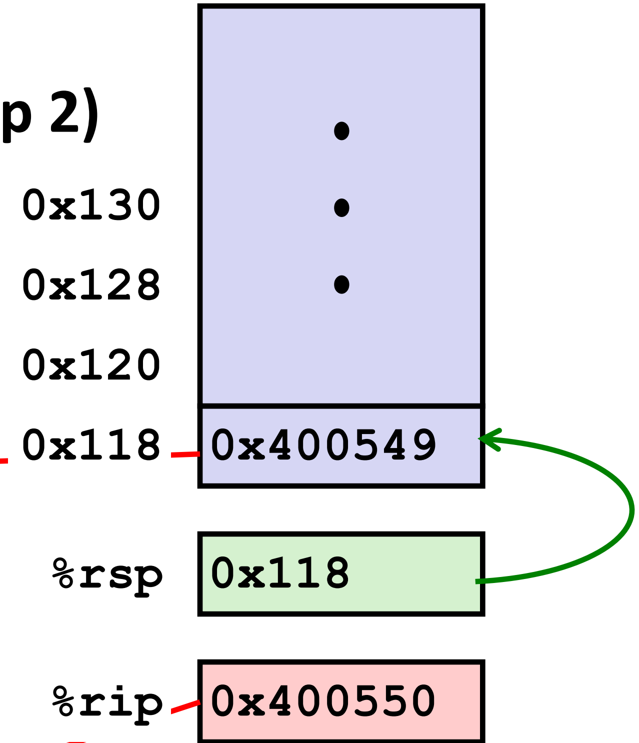
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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



Procedure Call Example (step 2)

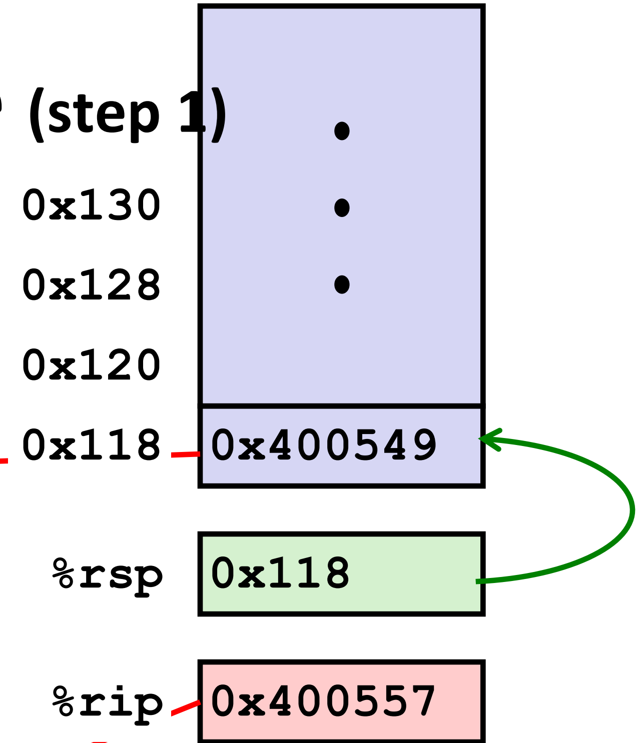
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```




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

Procedure Return Example (step 1)

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

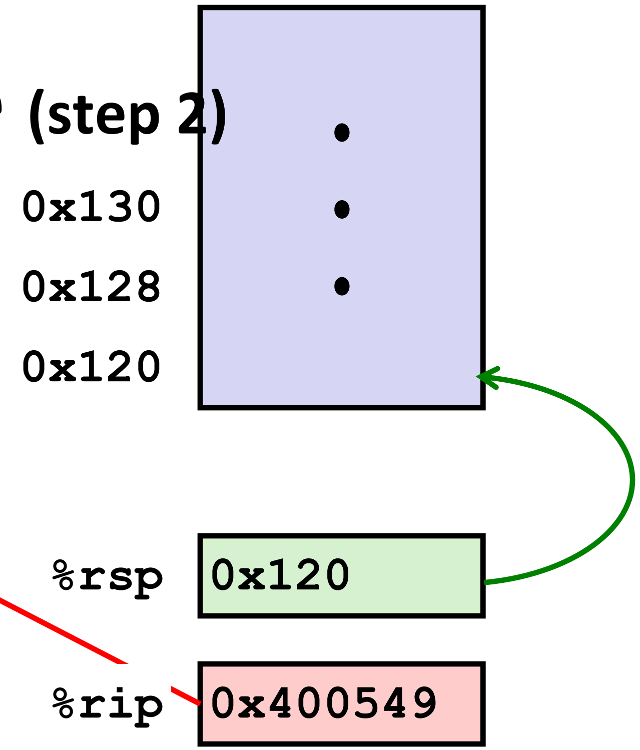



```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Procedure Return Example (step 2)

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

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



Today

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

Procedure Data Flow

Registers – NOT in Memory!

- First 6 arguments

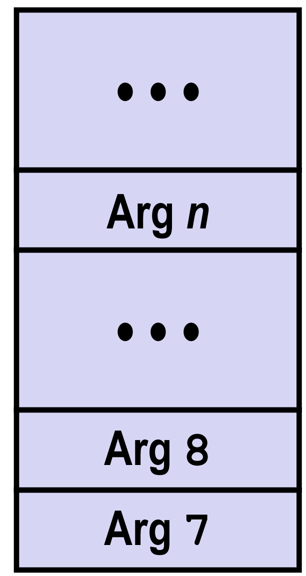
`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

- Return value

`%rax`

Stack – in Memory!

High Addresses



Low Addresses
`0x00…00`

- Only allocate stack space when needed

X86-64 Return Values

- **By convention, values returned by procedures are placed in the %rax register**
  - Choice of %rax is arbitrary, could have easily been a different register
- **Caller must make sure to save the contents of %rax before calling a callee that returns a value**
  - Part of register-saving convention
- **Callee places return value into the %rax register**
  - Any type that can fit in 8 bytes – integer, float, pointer, etc.
  - For return values greater than 8 bytes, best to return a pointer to them
- **Upon return, caller finds the return value in the %rax register**

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
```

Today

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

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
- **Stack allocated in *frames***
  - State for a single procedure instantiation

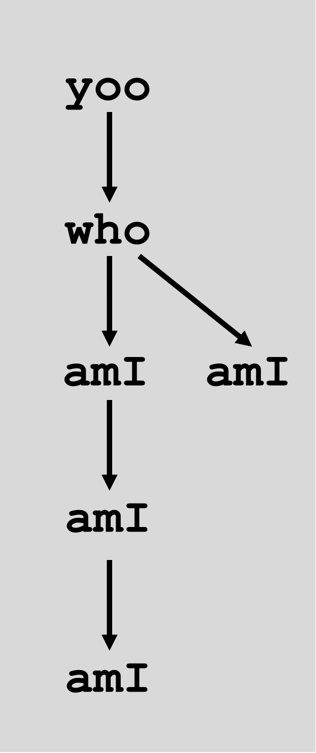
Call Chain Example

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

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

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

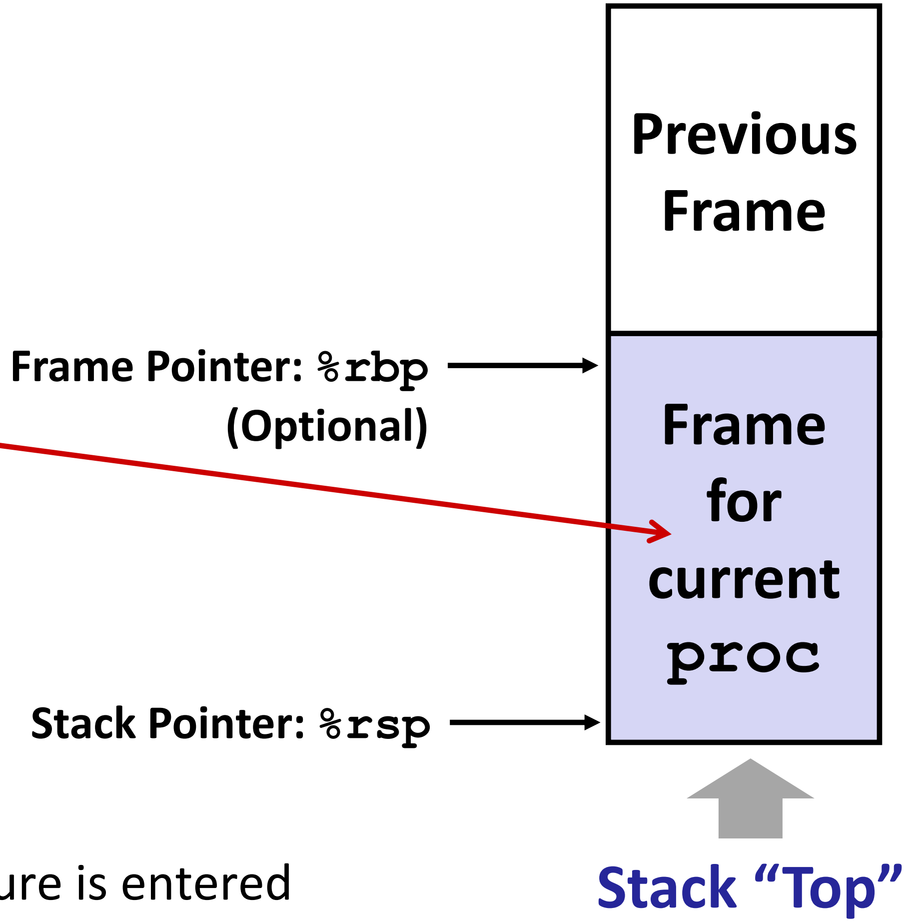
Example Call Chain



Procedure `amI` is recursive (calls itself)

Stack Frames

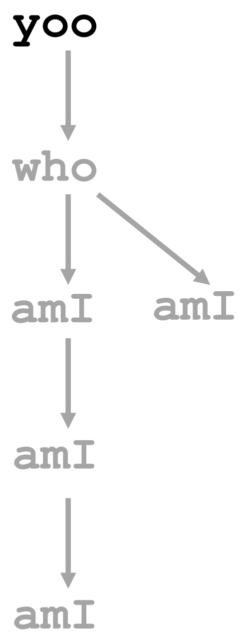
- **Contents**
  - Return address
  - If Needed:
    - Local variables
    - Temporary space
- **Management**
  - Space allocated when procedure is entered
    - "Set-up" code
  - Space deallocated upon return
    - "Finish" code



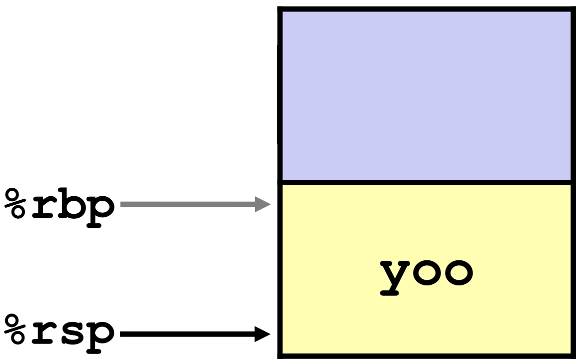
Example:

call to `yoo`

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



Stack

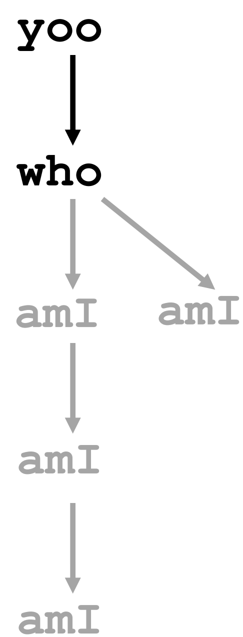


Example:

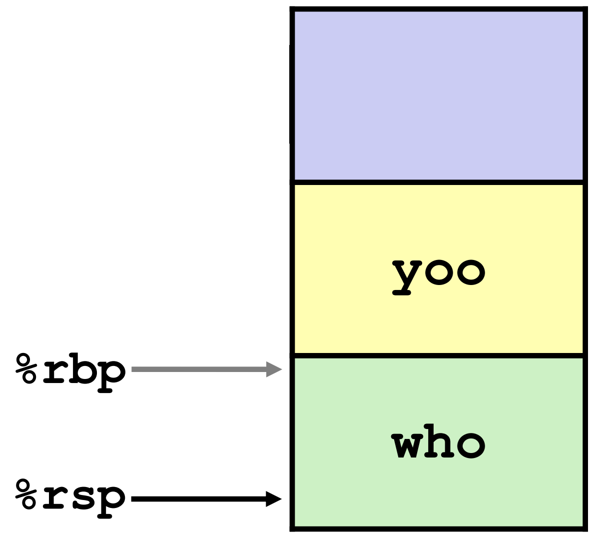
call to who

```
yoo(…)
{
}
```

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

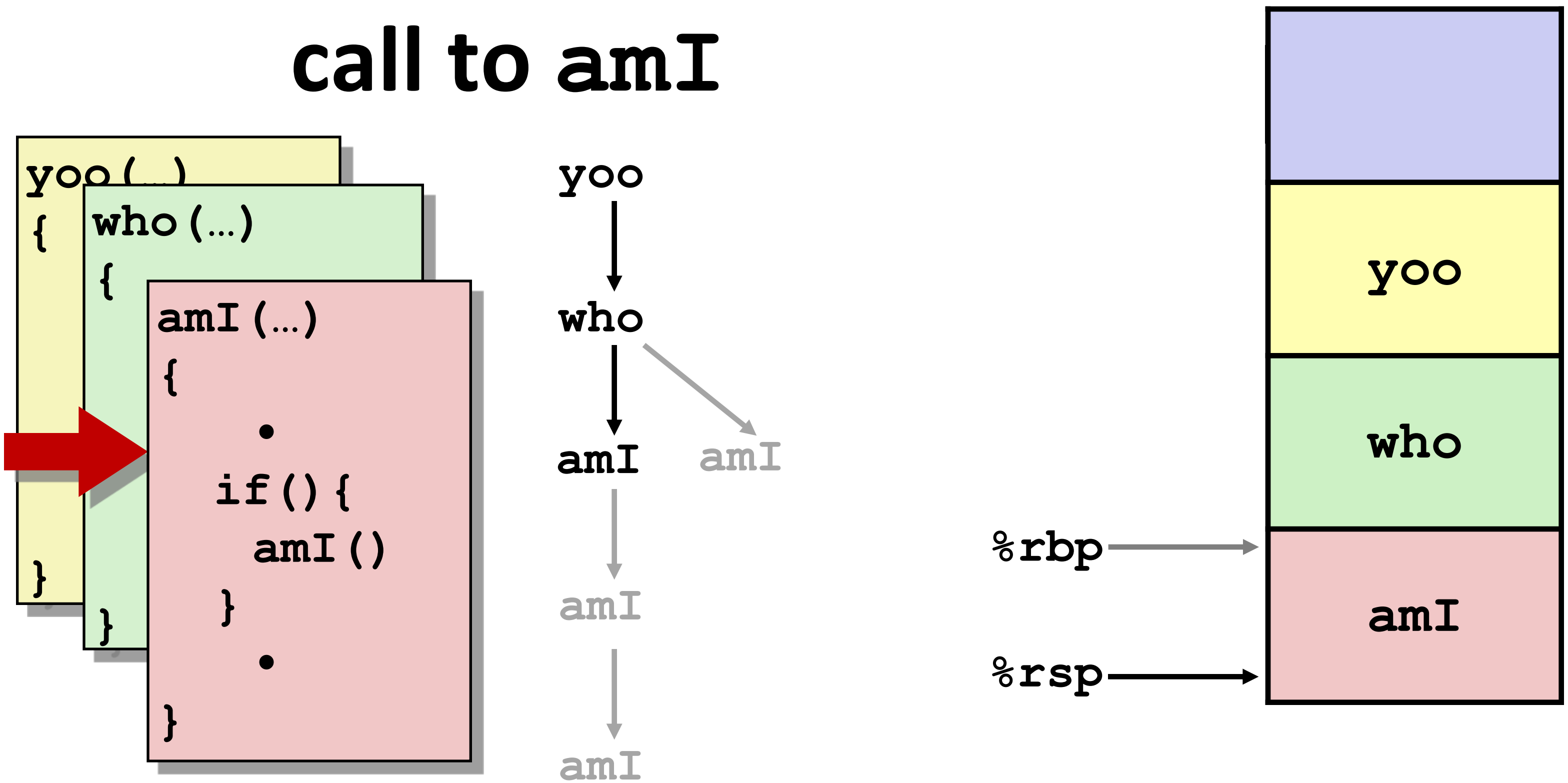


Stack

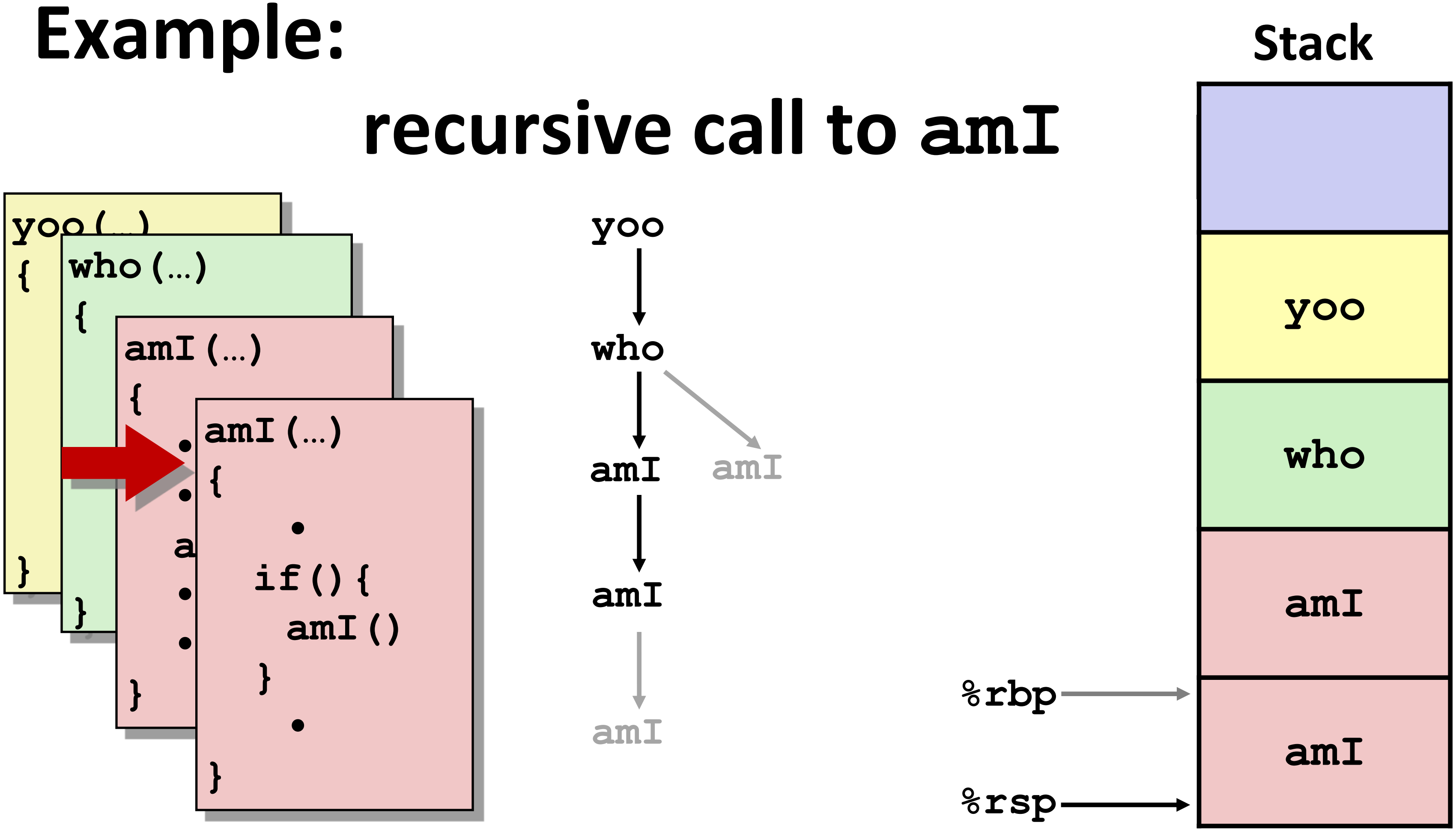


Example:

call to amI

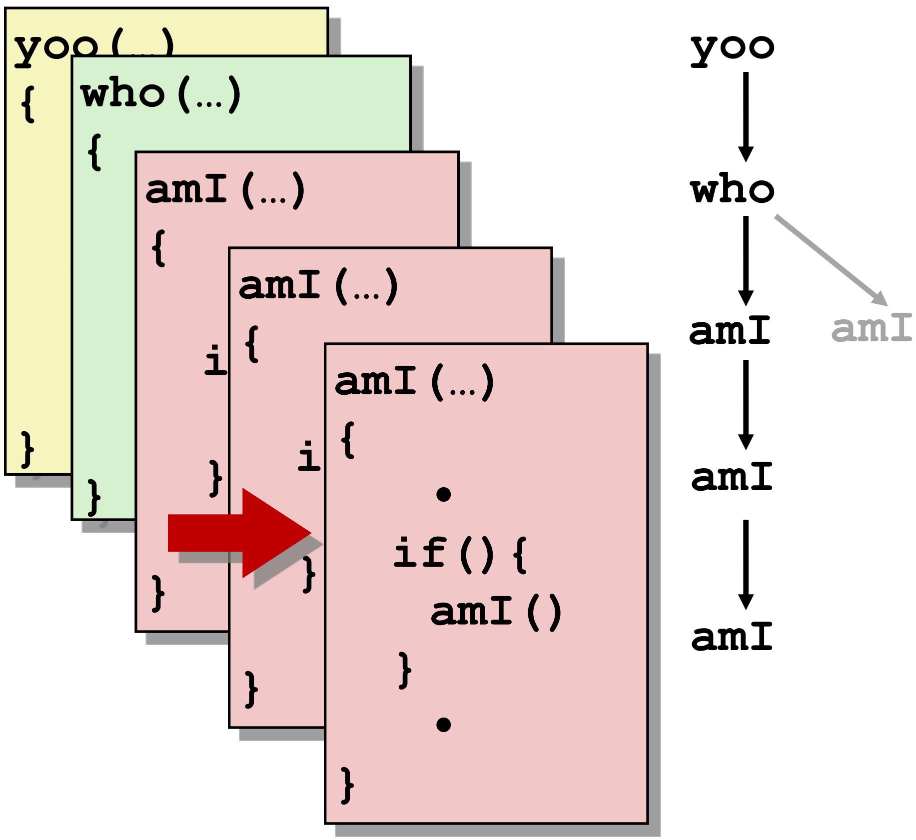


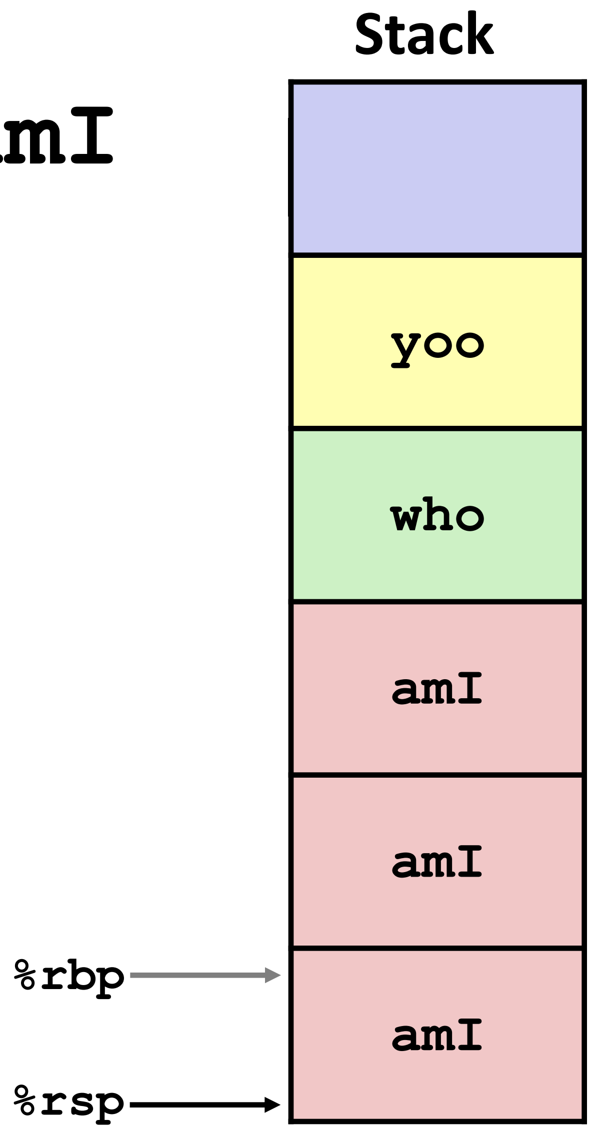
Example: recursive call to amI



Example:

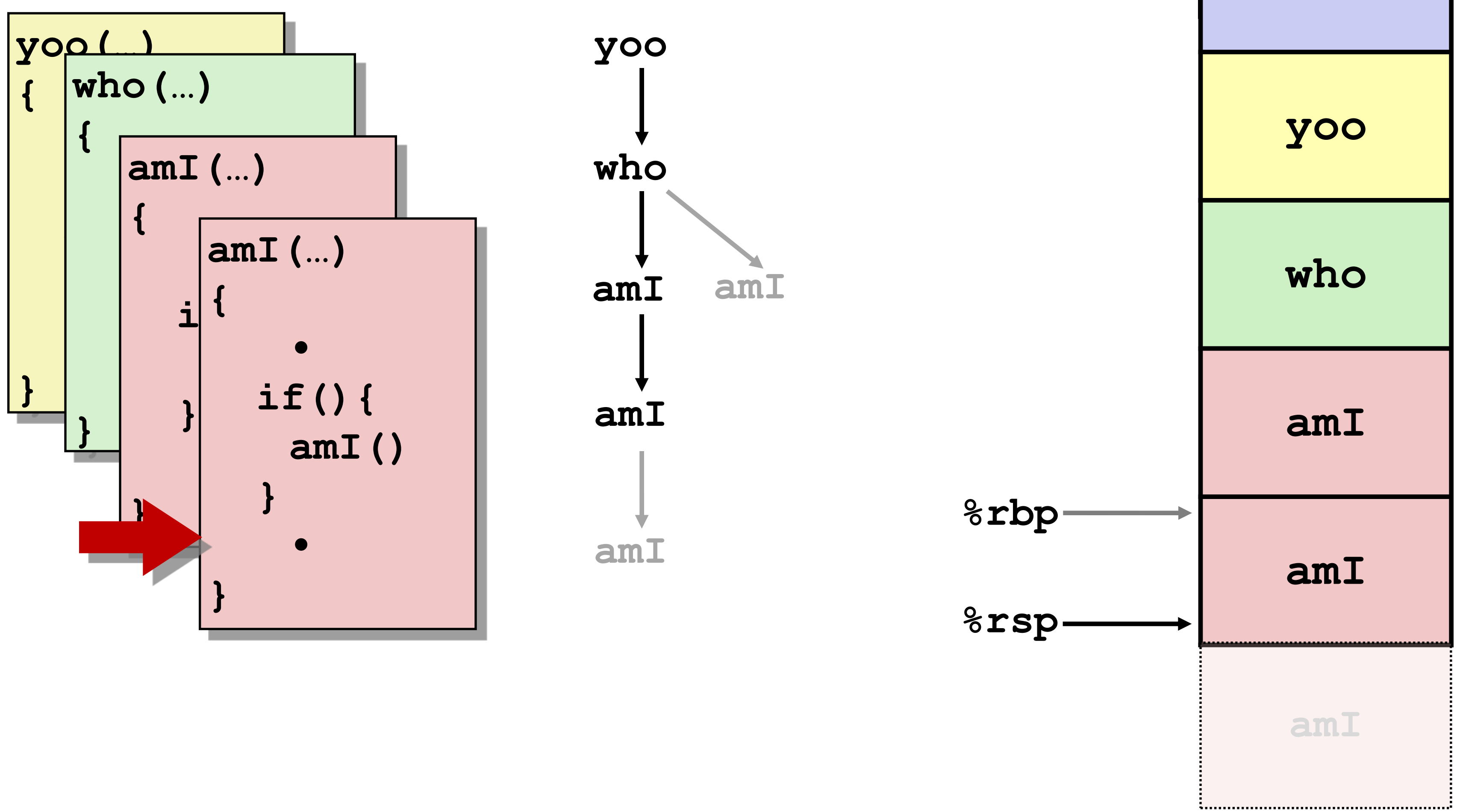
(another) recursive call to amI



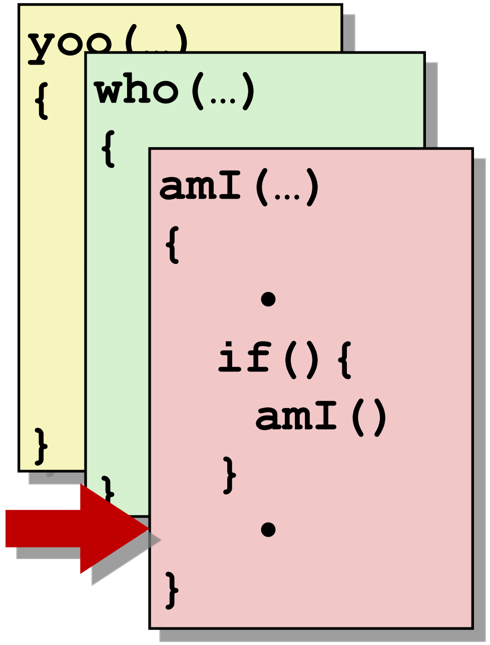


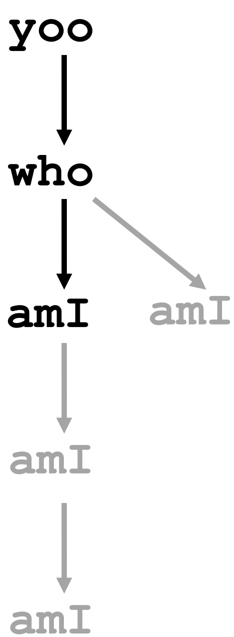
Return from:

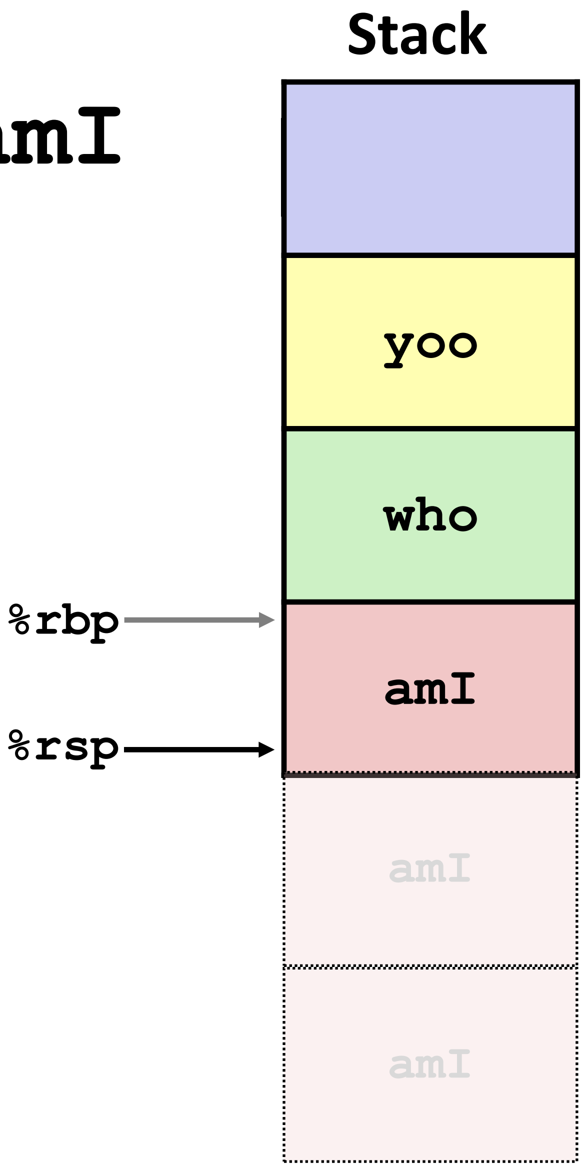
(another) recursive call to amI



Return from: recursive call to `amI`

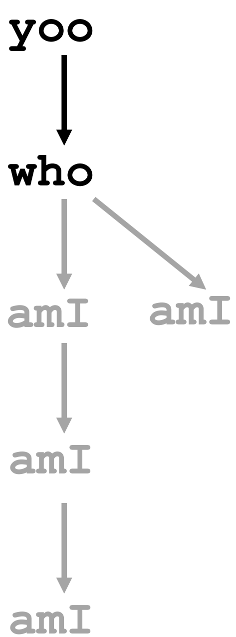


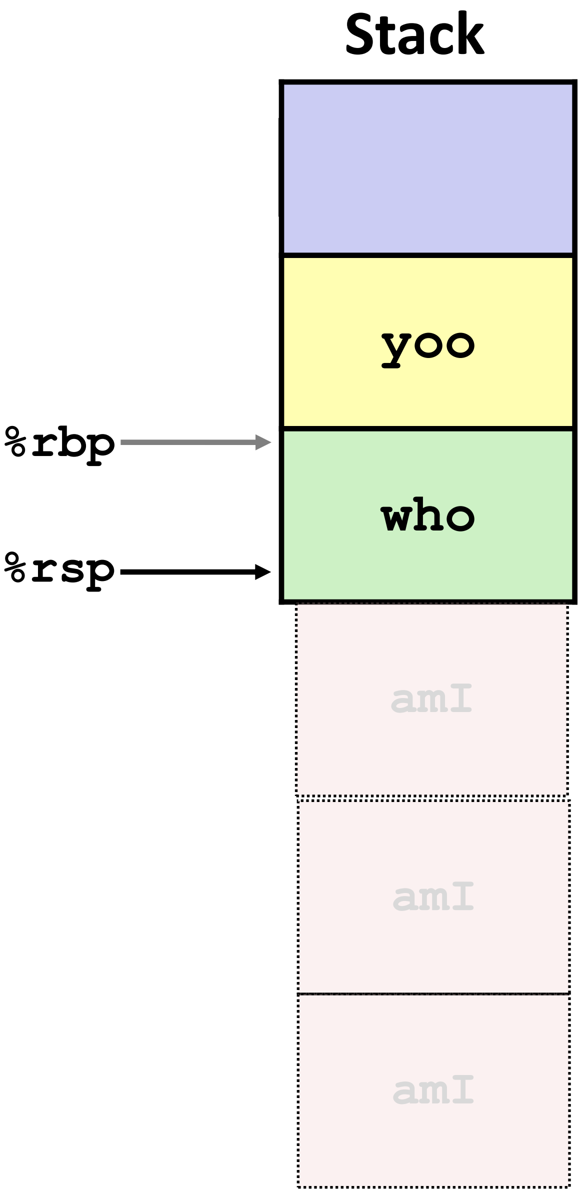




Return from: call to amI

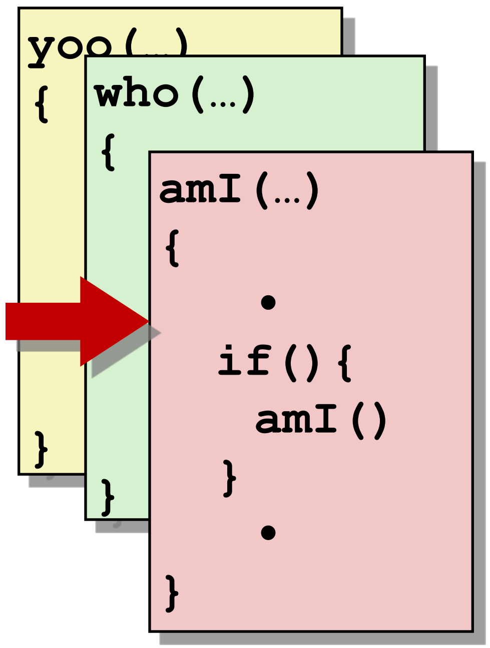
```
yoo(…)
{
who(…)
{
    •
    amI();
    •
    amI();
    •
}
}
```

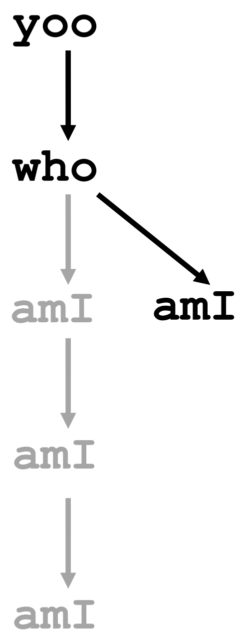


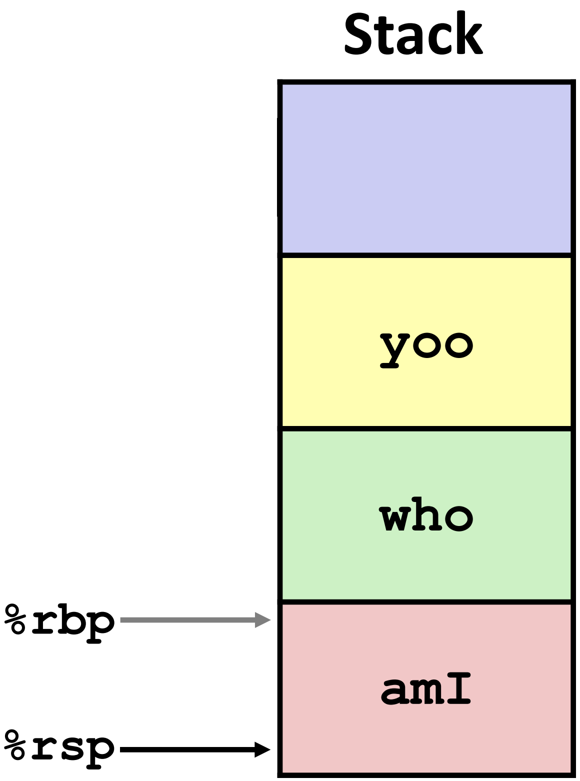


Example:

(second) call to amI



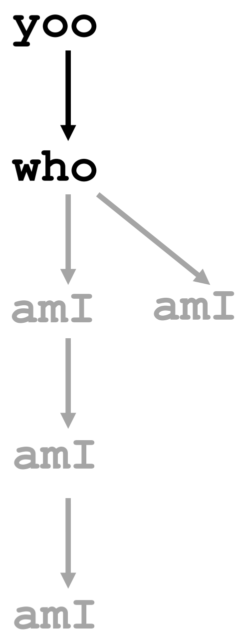




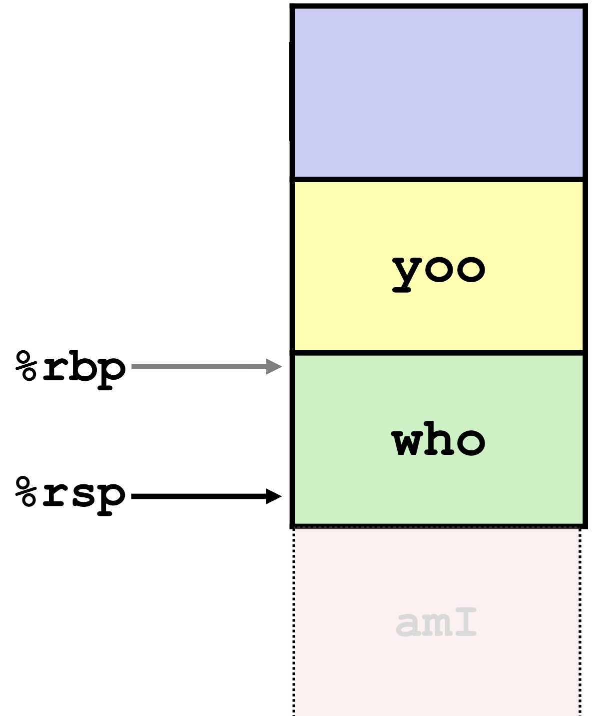
Return from:

(second) call to `amI`

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

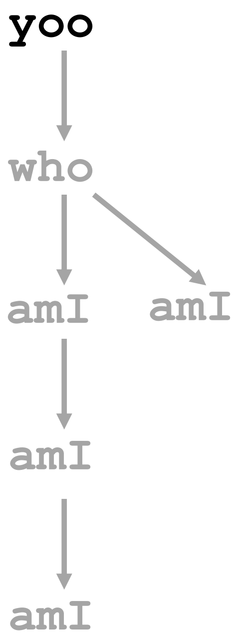


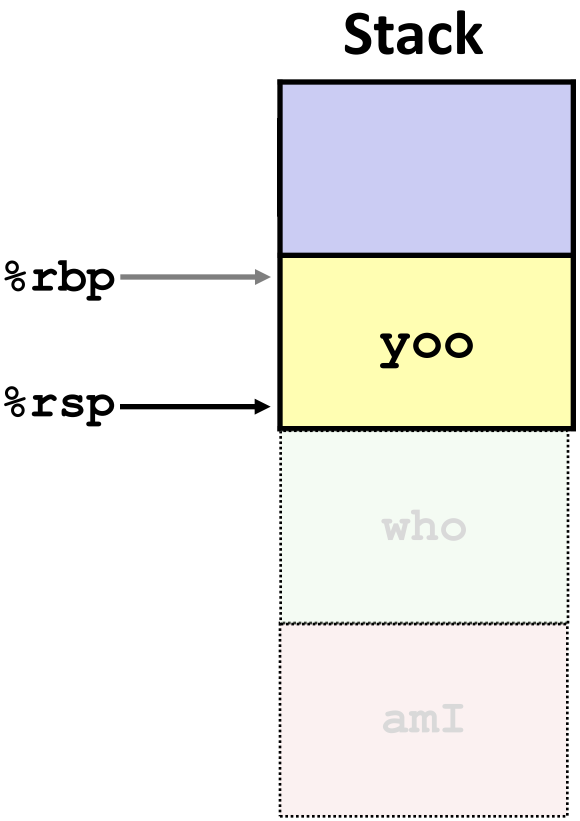
Stack



Return from: call to who

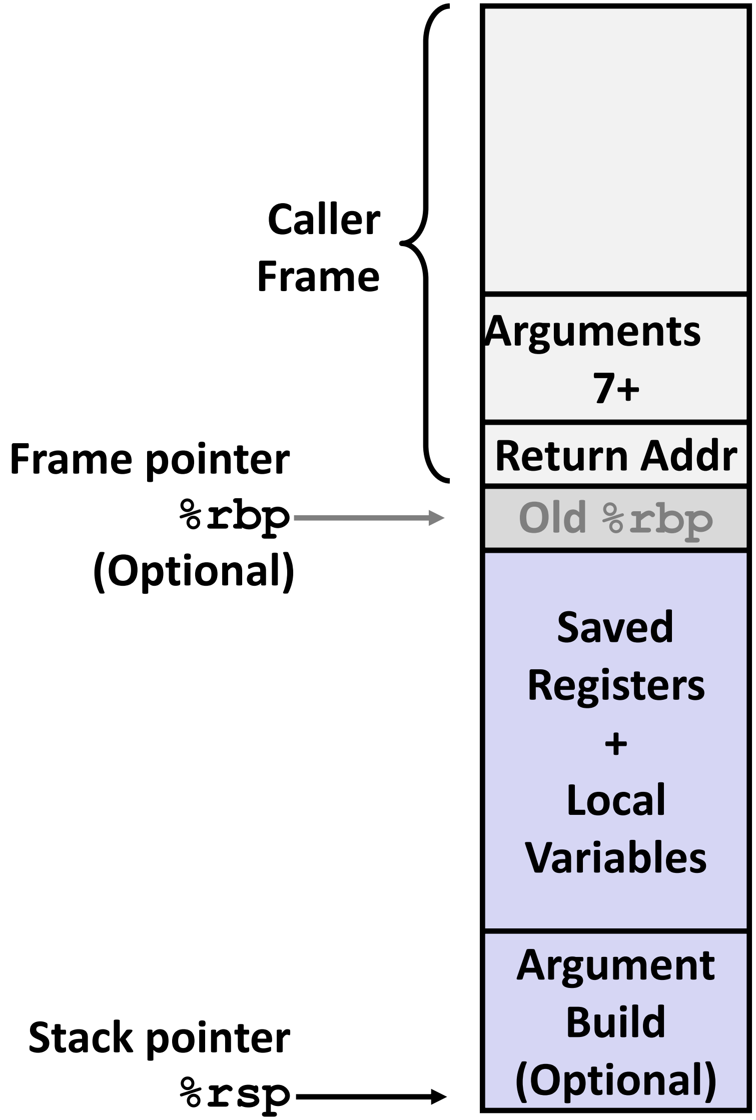
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```





x86-64/Linux Stack Frame

- **Caller's Stack Frame**
  - Arguments (if > 6 args) for this call
  - Return address
    - Pushed by `call` instruction
- **Current/ Callee Stack Frame**
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (If can't be kept in registers)
  - "Argument build" area (If callee needs to call another function - parameters for function about to call, if needed)



Example: `increment`

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

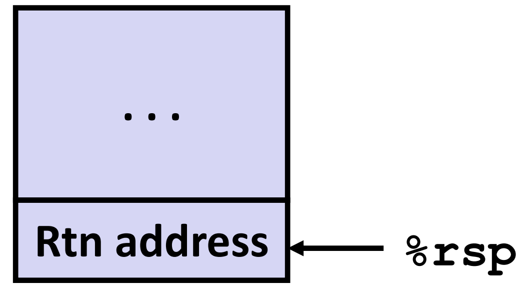
```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

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

Procedure Call Example

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Initial Stack Structure



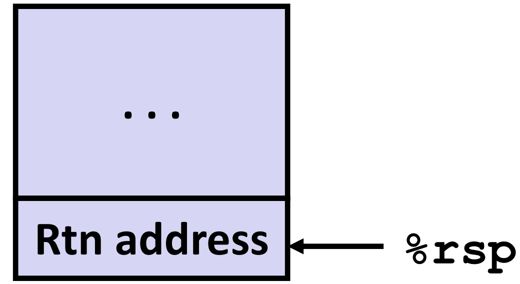

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rdi	
%rsi	
%rax	

Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Initial Stack Structure



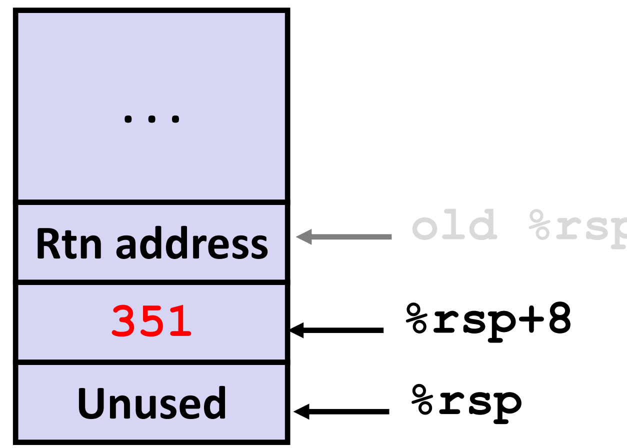

```
call_incr:
  subq     $16, %rsp
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Return address on stack is address of instruction immediately following the call to "`call_incr`" (not shown here).

Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

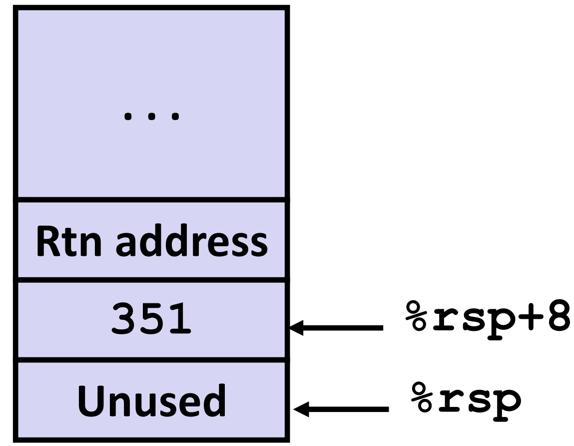
Allocate space for local vars

State after setup of space for local variables, in this case only v1 needs space on the stack. The compiler allocated extra space and often does this for a variety of reasons including alignment.

Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure




```
call_incr:
  subq     $16, %rsp
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes one less byte to encode a movl than a movq.

Set up parameters for call to `increment`

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`

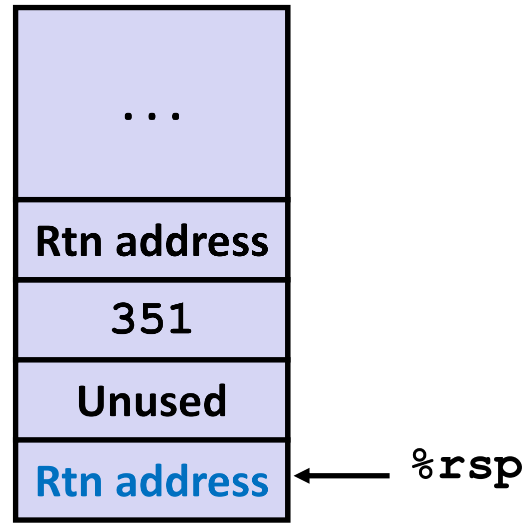
Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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

Stack Structure



State while inside the function `increment`. Return address on top of stack is address of the **`addq`** instruction immediately following call to `increment`

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`

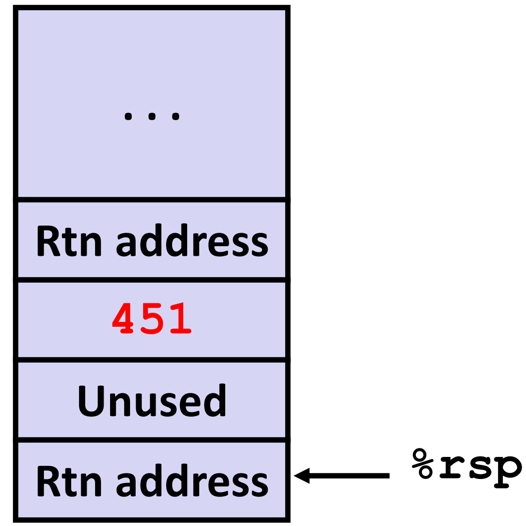
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

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

Stack Structure



State while inside the function `increment`. After code in body of `increment` has been executed.

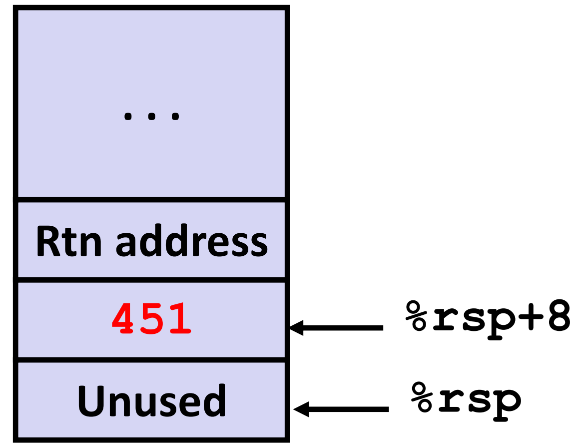
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



State after returning from call to `increment`. Registers and memory have been modified and return address has been popped off stack.

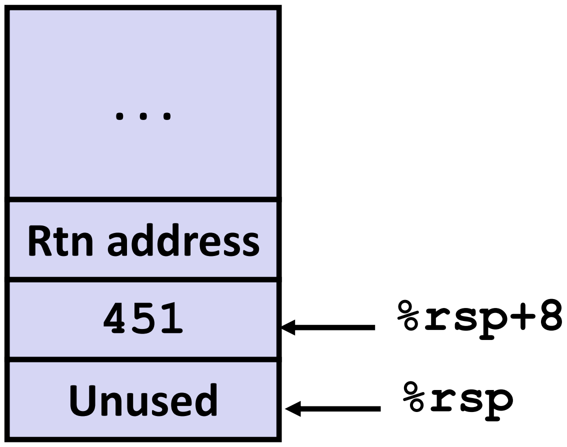
Register	Use(s)
`%rdi`	`&v1`
`%rsi`	451
`%rax`	351

Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



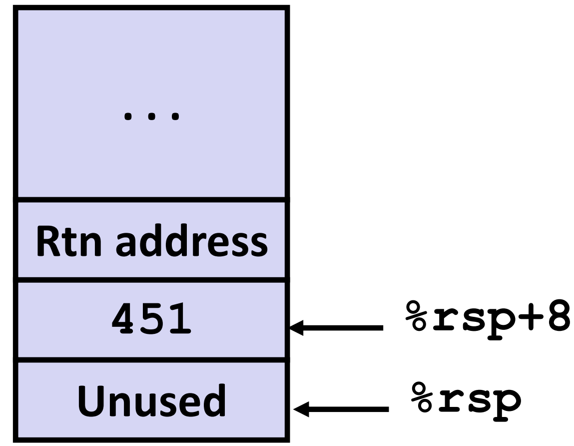
Update `$rax` to contain v1 + v2

Register	Use(s)
%rax	451 + 351

Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure



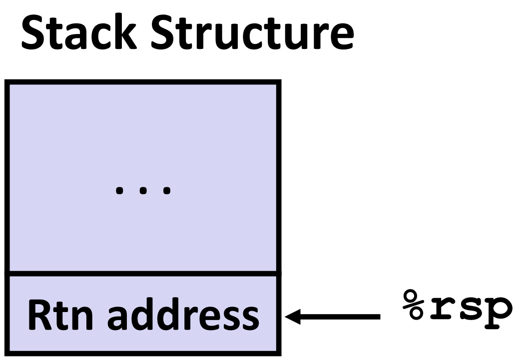

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

De-allocate space for local vars

Register	Use(s)
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

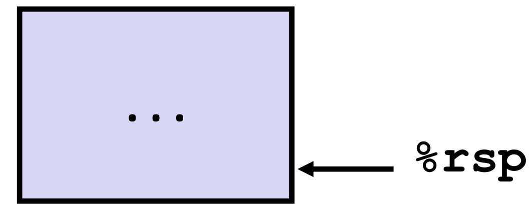
State just before returning from call to `call_incr`.

Register	Use(s)
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Final Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

State immediately AFTER returning from call to `call_incr`. Return addr has been popped off stack and control has returned to the instruction immediately following the call to `call_incr` (not shown here).

Register	Use(s)
`%rax`	802

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! Either:
    - *caller* should save `%rdx` before the call (and restore it after the call)
    - *callee* should save `%rdx` before using it (and restore it before returning)

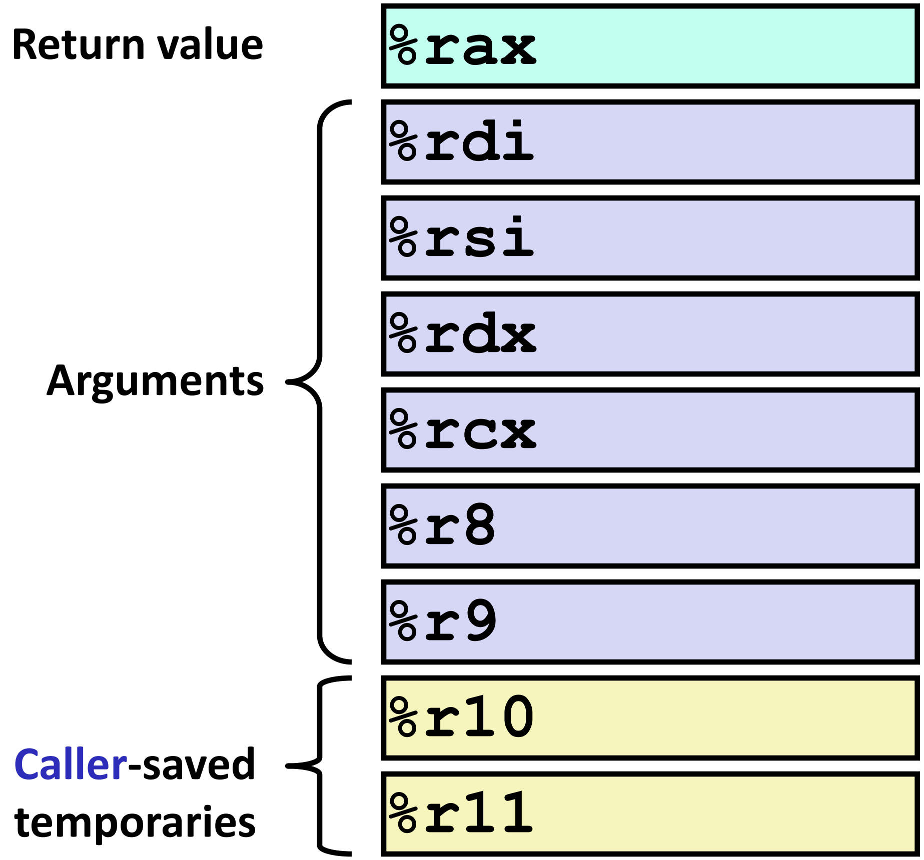
Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***

- **Can a register be used for temporary storage?**
- **Conventions**
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before calling
    - Caller restores values after the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

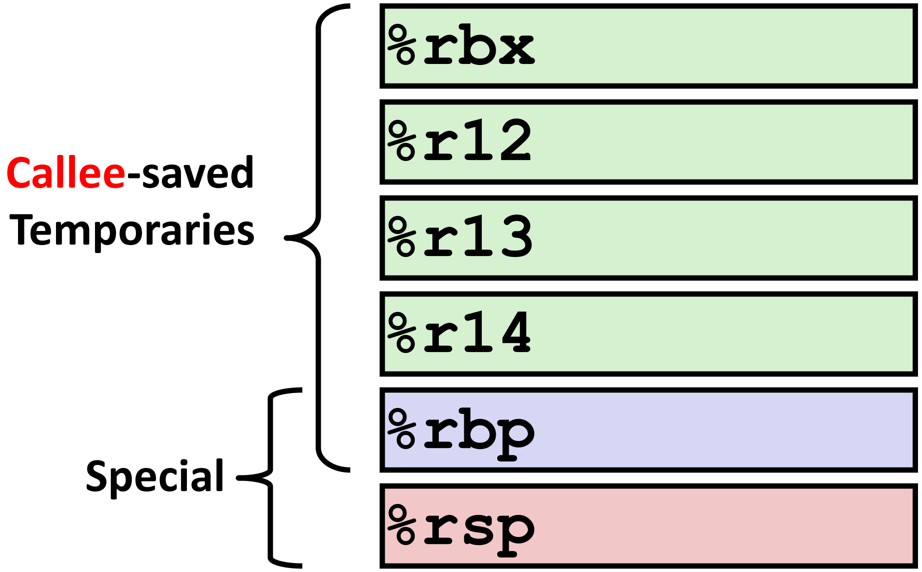
x86-64 Linux Register Usage, part1

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



x86-64 Linux Register Usage, part 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



x86-64 64-bit Registers: Usage Conventions

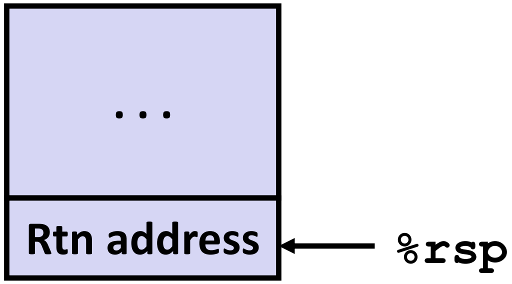
`%rax`	Return value - Caller saved	`%r8`	Argument #5 - Caller saved
`%rbx`	Callee saved	`%r9`	Argument #6 - Caller saved
`%rcx`	Argument #4 - Caller saved	`%r10`	Caller saved
`%rdx`	Argument #3 - Caller saved	`%r11`	Caller Saved
`%rsi`	Argument #2- Caller saved	`%r12`	Callee saved
`%rdi`	Argument #1 - Caller saved	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

Callee-Saved Example, (step 1)

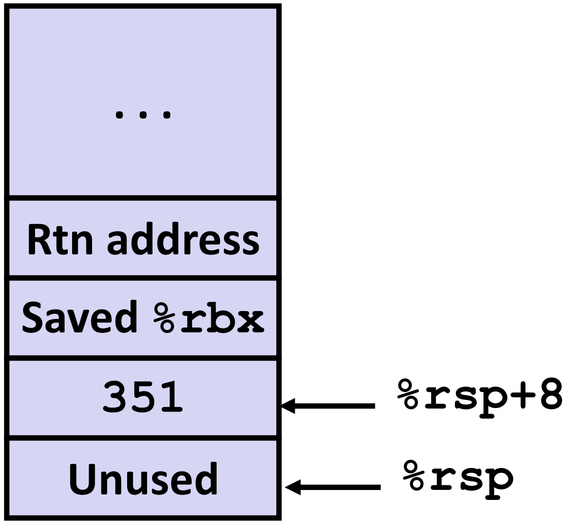
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

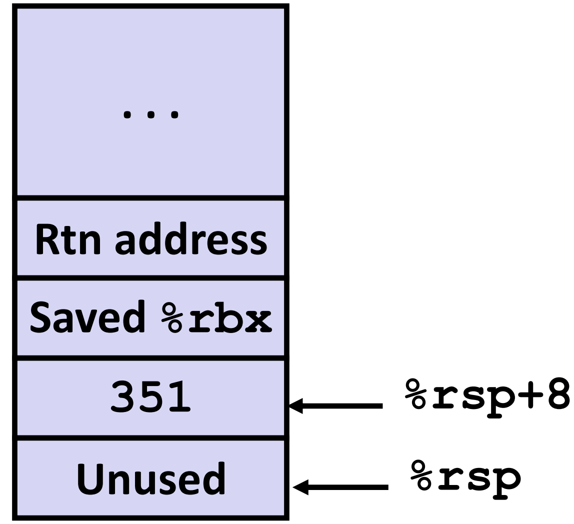


Callee-Saved Example (step 2)

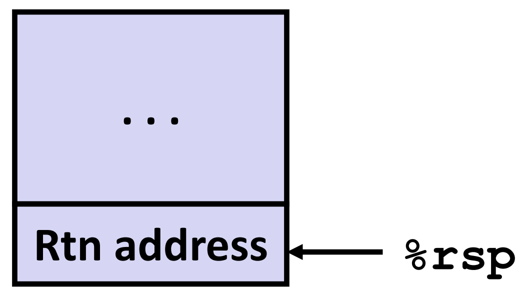
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



Today

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

Recursive Function

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

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

Recursive Function: Base 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
```

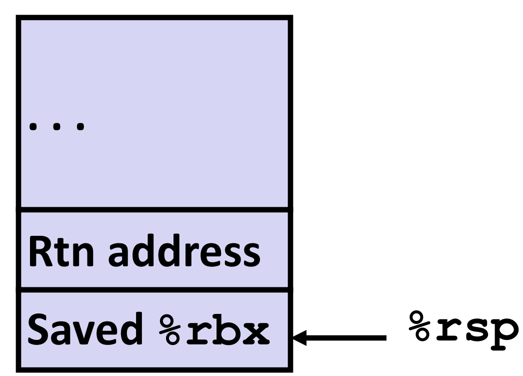
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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`	Rec. argument
`%rbx`	`x & 1`	Callee-saved

Recursive Function: Call

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

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

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

Recursive Function: Result

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

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

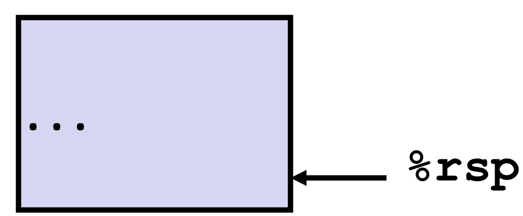
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function: Completion

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

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

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



Observations About Recursion

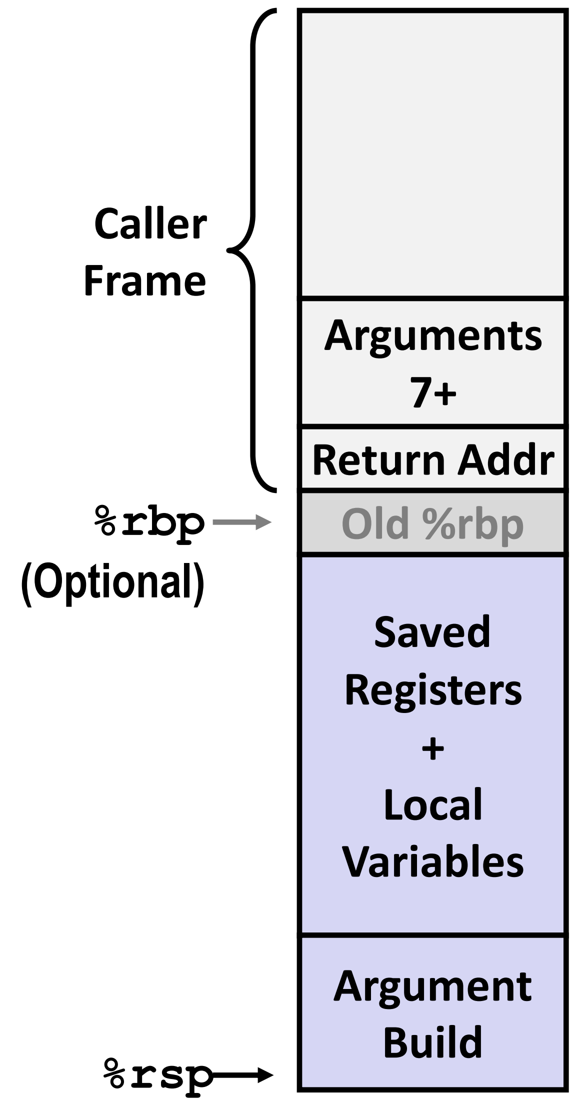
- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow - described in future lecture)
  - 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 Stack Frames

- **Often (ideally), x86-64 functions need no stack frame at all**
  - Just a return address is pushed onto the stack when a function call is made
- **A function *does* need a stack frame when it:**
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of calle**r**-save registers before calling a procedure
  - Needs to save the state of calle**e**-save registers before modifying them

x86-64 Procedure Summary

- **Important Points**
  - Procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **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`



One more x86-64 example

- Example of passing more than 6 parameters and passing addresses of local variables
- The following example, along with a brief re-cap of x86-64 calling conventions is in this video:

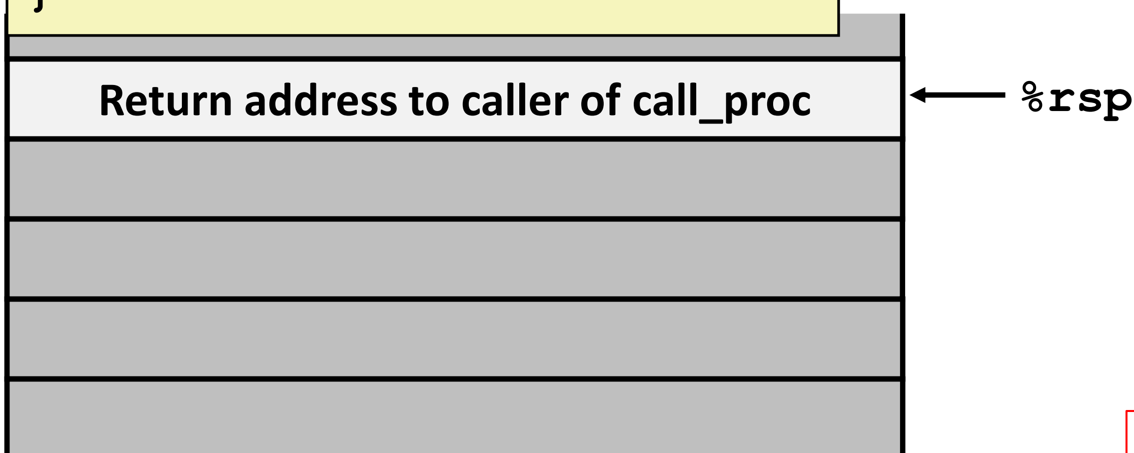
- 5. Procedures and Stacks
  - …
  - 6. x86-64 Calling Conventions

- https://courses.cs.washington.edu/courses/cse351/videos/05/056.mp4

x86-64 Example (1)

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

```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)  # x1
  movl  $2,24(%rsp)  # x2
  movw  $3,28(%rsp)  # x3
  movb  $4,31(%rsp)  # x4
  • • •
```

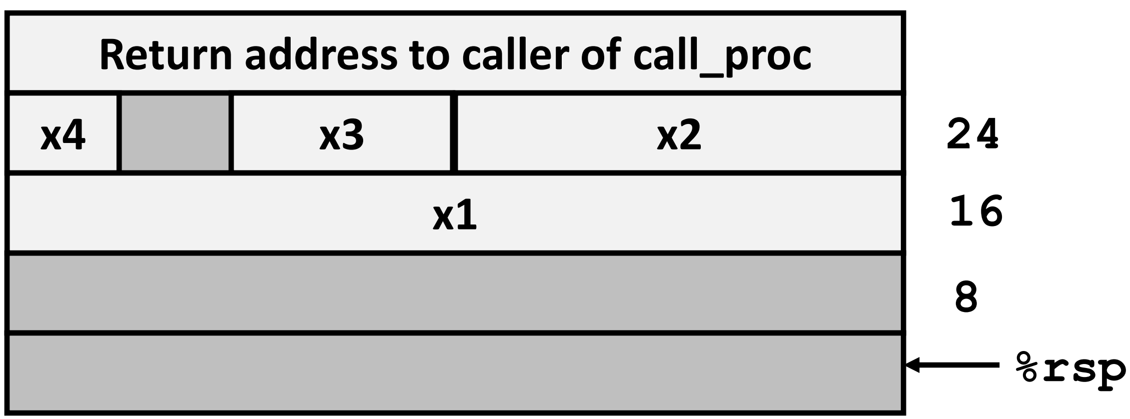


NB: Details may vary depending on compiler.

x86-64 Example (2) – Allocate local vars

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

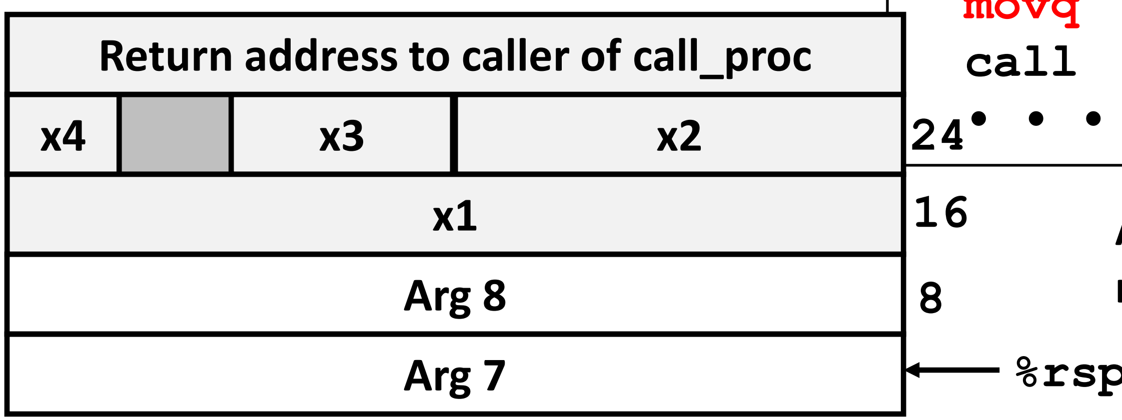
```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)  # x1
  movl  $2,24(%rsp)  # x2
  movw  $3,28(%rsp)  # x3
  movb  $4,31(%rsp)  # x4
  • • •
```



x86-64 Example (3) – setup params to `proc`

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

```
call_proc:
  • • •
  leaq  24(%rsp),%rcx # %rcx=&x2
  leaq  16(%rsp),%rsi # %rsi=&x1
  leaq  31(%rsp),%rax # %rax=&x4
  movq  %rax,8(%rsp)  # arg8=&4
  movl  $4,(%rsp)     # arg7=4
  leaq  28(%rsp),%r9  # %r9=&x3
  movl  $3,%r8d  # %r8 = 3
  movl  $2,%edx  # %rdx = 2
  movq  $1,%rdi  # %rdi = 1
  call  proc
  • • •
```



Arguments passed in (in order):
rdi, rsi, rdx, rcx, r8, r9

Same instructions as in video, just a different order.

x86-64 Example (4) – setup params to `proc`

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

```
call_proc:
   • • •
   leaq   24(%rsp),%rcx
   leaq   16(%rsp),%rsi
   leaq   31(%rsp),%rax
   movq   %rax,8(%rsp)
   movl   $4,(%rsp)
   leaq   28(%rsp),%r9
   movl   $3,%r8d
   movl   $2,%edx
   movq   $1,%rdi
   call   proc
   • • •
```

Note sizes

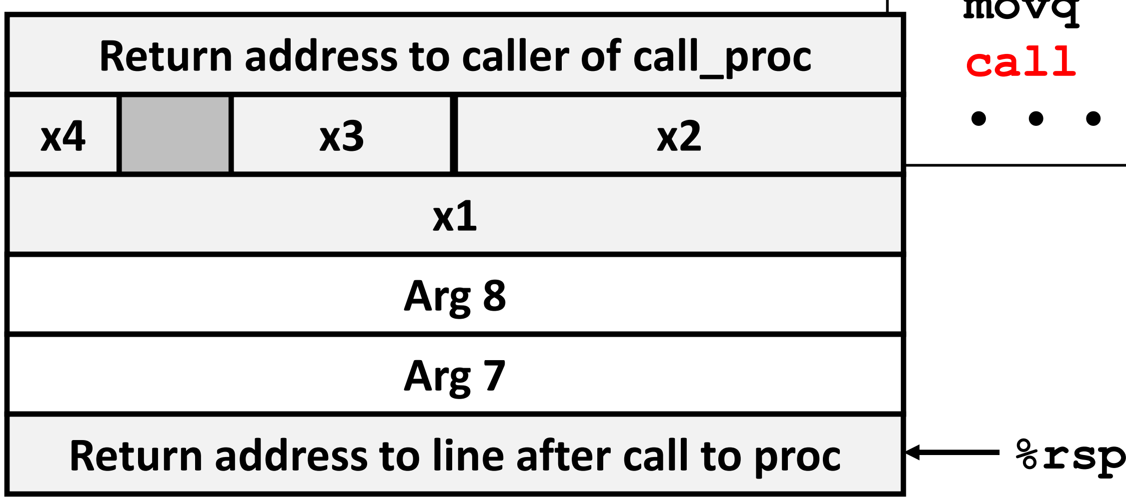
Return address to caller of call_proc				
x4		x3	x2	24
x1				16
Arg 8				8
Arg 7				← %rsp

Arguments passed in (in order):
rdi, rsi, rdx, rcx, r8, r9

x86-64 Example (5) – call `proc`

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

```
call_proc:
  • • •
  leaq   24(%rsp),%rcx
  leaq   16(%rsp),%rsi
  leaq   31(%rsp),%rax
  movq   %rax,8(%rsp)
  movl   $4,(%rsp)
  leaq   28(%rsp),%r9
  movl   $3,%r8d
  movl   $2,%edx
  movq   $1,%rdi
  call   proc
  • • •
```



x86-64 Example (6) – after call to `proc`

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

```
call_proc:
  • • •
  movswl 28(%rsp),%eax # %eax=x3
  movsbl 31(%rsp),%edx # %edx=x4
  subl   %edx,%eax # %eax=x3-x4
  cltq
  movslq 24(%rsp),%rdx # %rdx=x2
  addq   16(%rsp),%rdx #%rdx=x1+x2
  imulq  %rdx,%rax # %rax=rax*rdx
  addq   $32,%rsp
  ret
```

Return address to caller of call_proc				
x4		x3	x2	24
x1				16
Arg 8				8
Arg 7				← `%rsp`

`movs` – move and sign extend

`cltq` - sign extend `%eax` into `%rax`

x86-64 Example (7) – de-allocate local vars

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

```
call_proc:
  • • •
  movswl 28(%rsp),%eax
  movsbl 31(%rsp),%edx
  subl   %edx,%eax
  cltq
  movslq 24(%rsp),%rdx
  addq   16(%rsp),%rdx
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```

Return address to caller of call_proc ← `%rsp`