UNM SCHOOL of ENGINEERING
Department of Computer Science

# Procedure Call: Basic Control and Data Linkage Section 3.7

Prof. Soraya Abad-Mota, PhD

# Answer to Quiz from 9/23

1. Condition codes: how many are there? 4

   How many bits each? 1 bit each

1. Used for? Conditional branching
2. How to set them implicitly? All arithmetic and logical instructions (in assembly) set them up.
3. How to set them explicitly? `cmp` and `test` instructions
4. Which instruction allows you to read them? `set` and `jump` instructions allow to read and use them

# Conclusions from the quiz

- Average 2.5/5
- Please keep up to date with the topics; have all the tables handy while we are studying assembly code
- Mechanical: e.g. understanding how each instruction works and the names and sizes of the registers,
- Conceptual: notion of the condition codes, how they are set and used and what for
- Today we cover both, concepts related to procedures, how they translate into assembly and how the system stack works!

# Questions?

- How are the control structures/statements of C translated into assembly/machine code?
- Did you try doing some of the suggested exercises?

UNM SCHOOL of ENGINEERING
Department of Computer Science

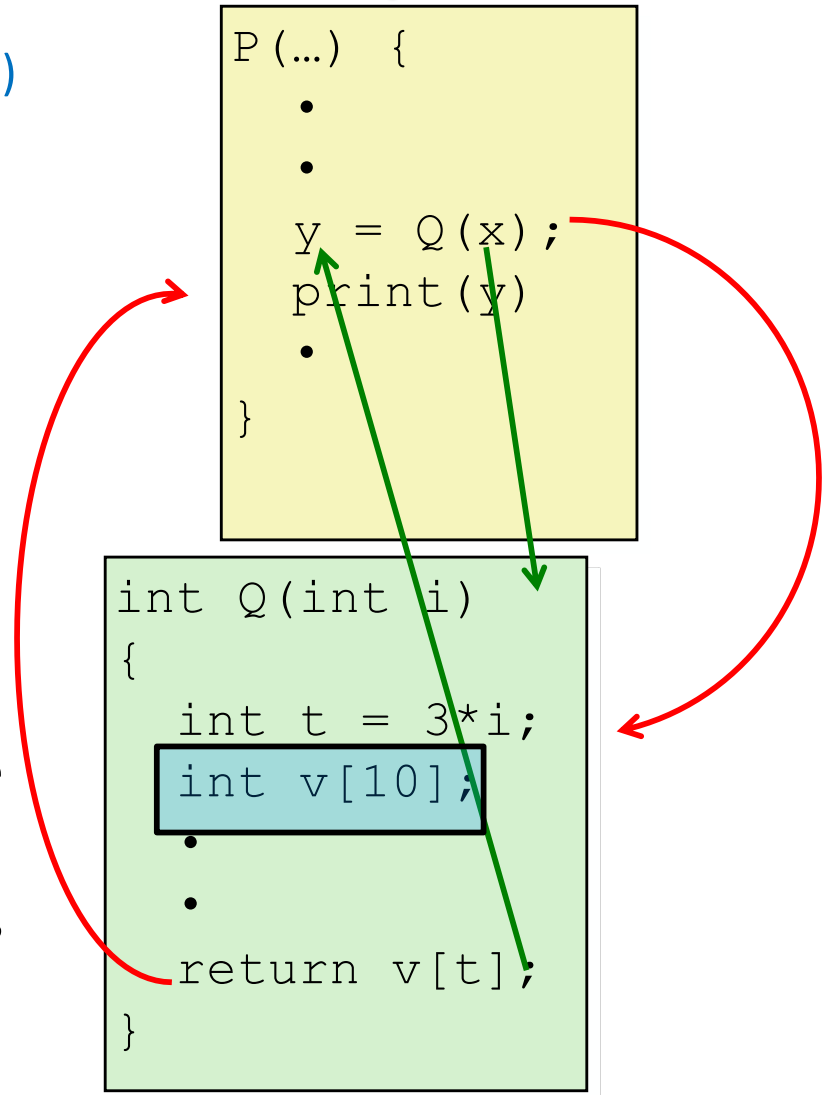
# Procedures (Sec. 3.7)

## Lect #11

Prof. Soraya Abad-Mota, PhD

# Mechanisms for Procedure call/return

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



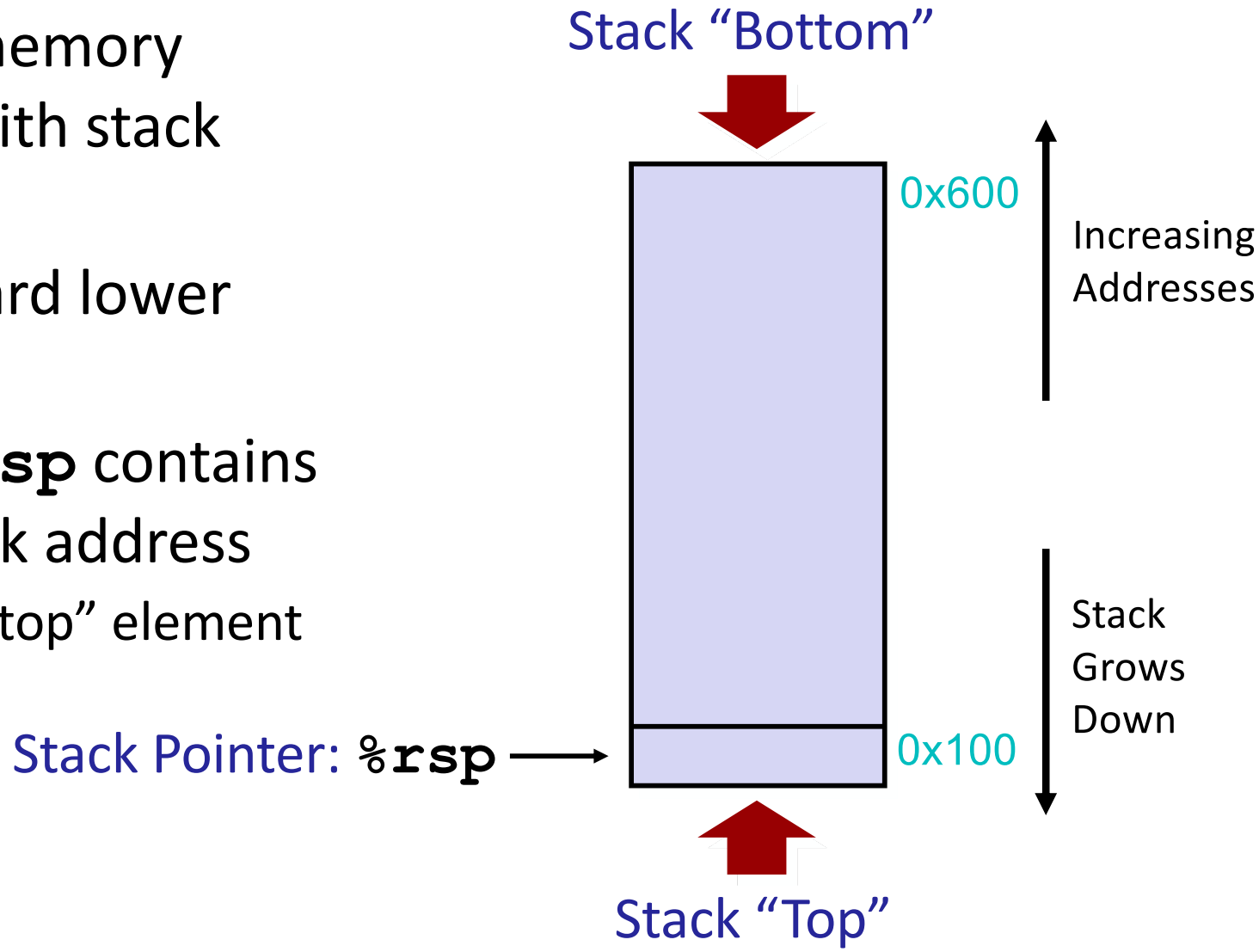
# Topics on Procedures

- **Runtime Stack**
  - **Structure and addressing conventions**
  - **pushq and popq insructions**

- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

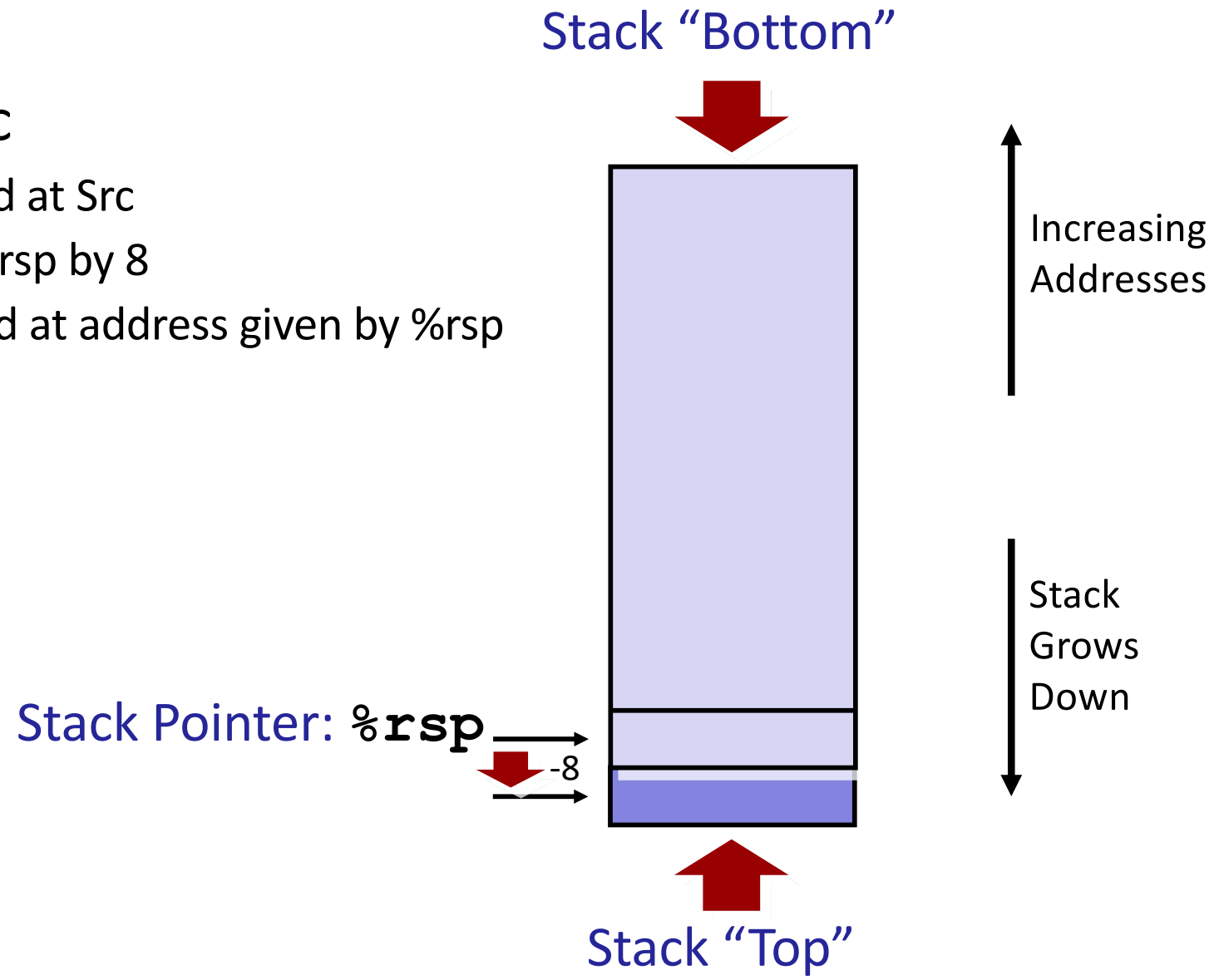
# x86-64 Stack

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



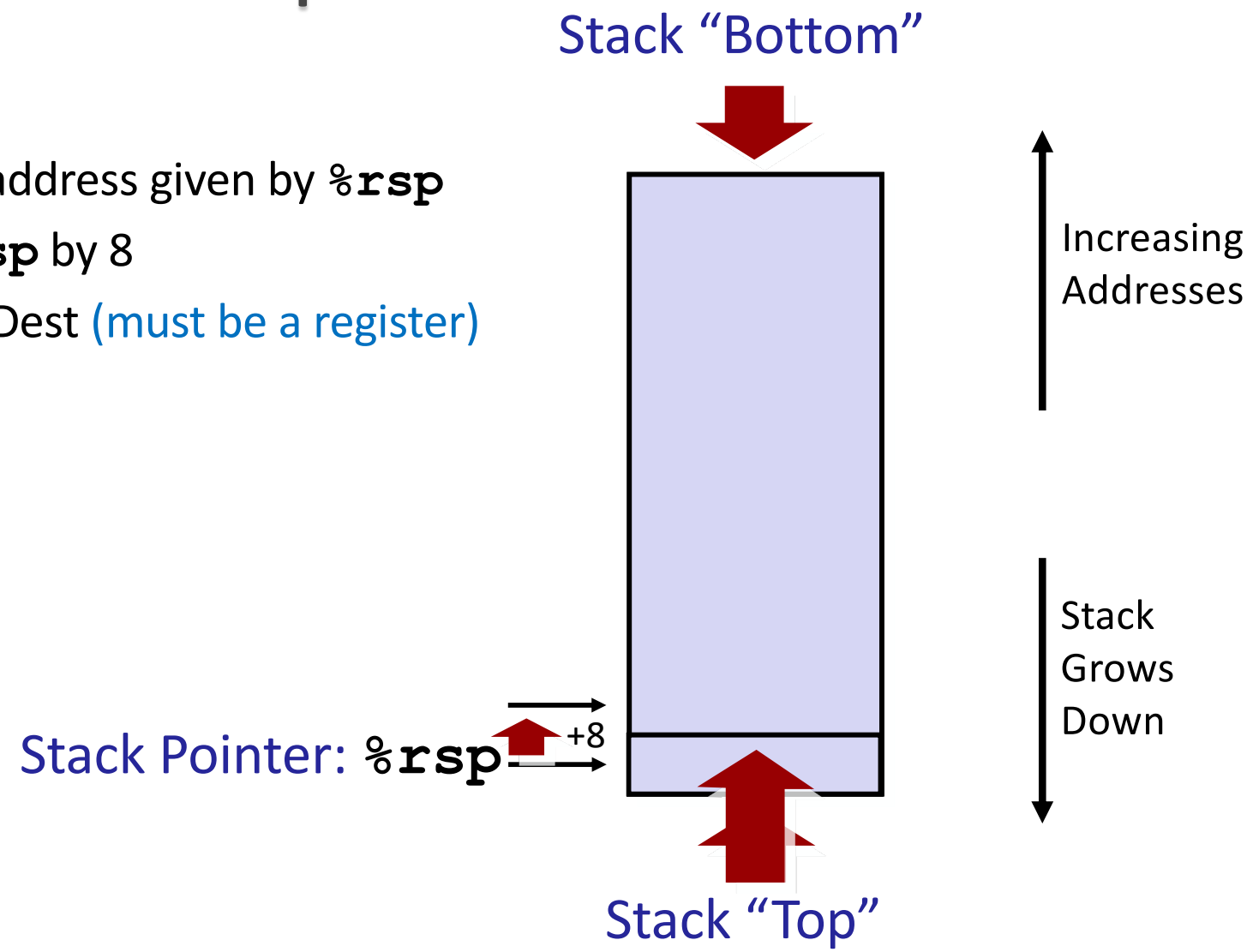
# x86-64 Stack: Push

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



# x86-64 Stack: Pop

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



# Topics on Procedures

- Procedures
  - 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;
}
```

```
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
```

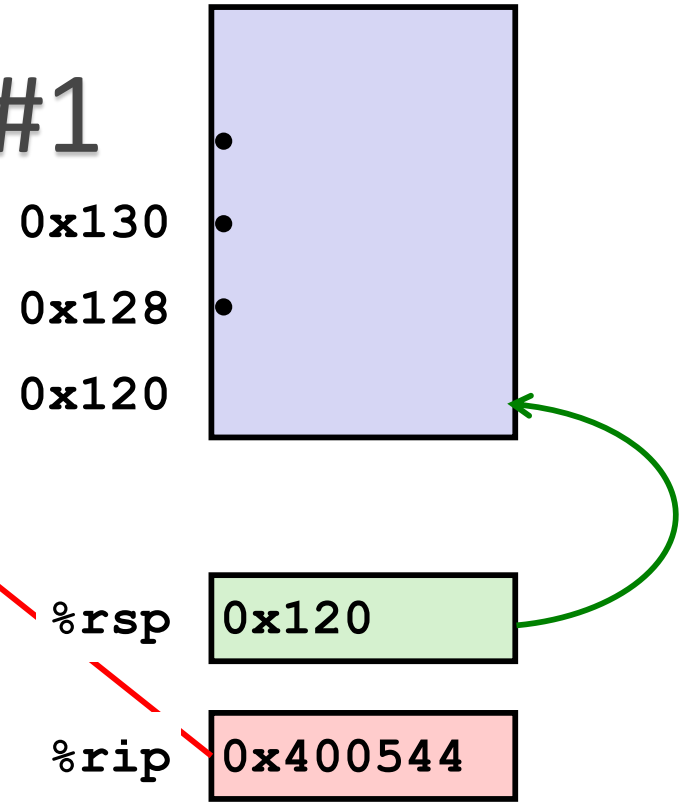
# Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - Push *return address (*on stack): address of the next instruction right after call (in calling prog.)
  - Jump to label (address of 1st instruction in called procedure)
- Procedure return: **`ret`**
  - Pop address from stack
  - Jump to address

# Control Flow Example #1

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

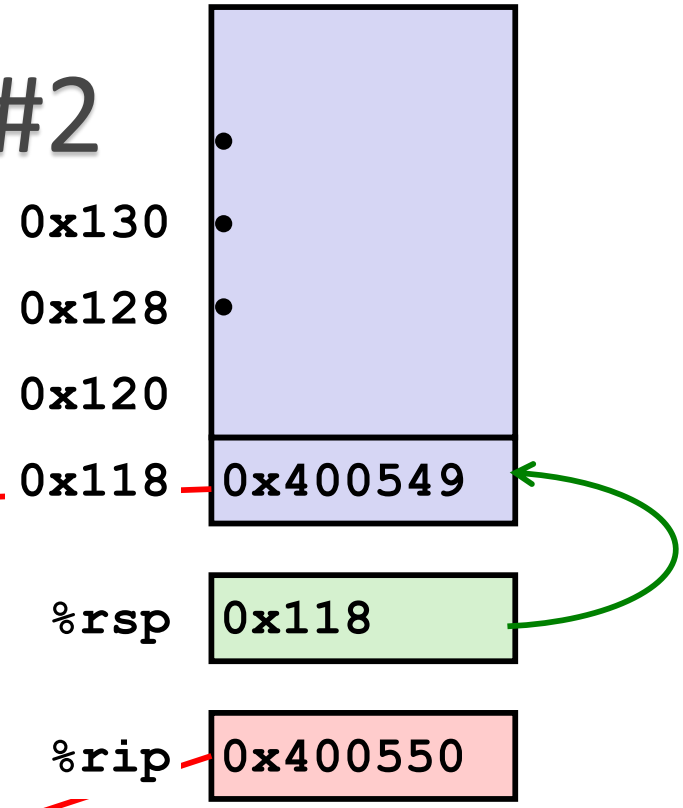
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



# Control Flow Example #2

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

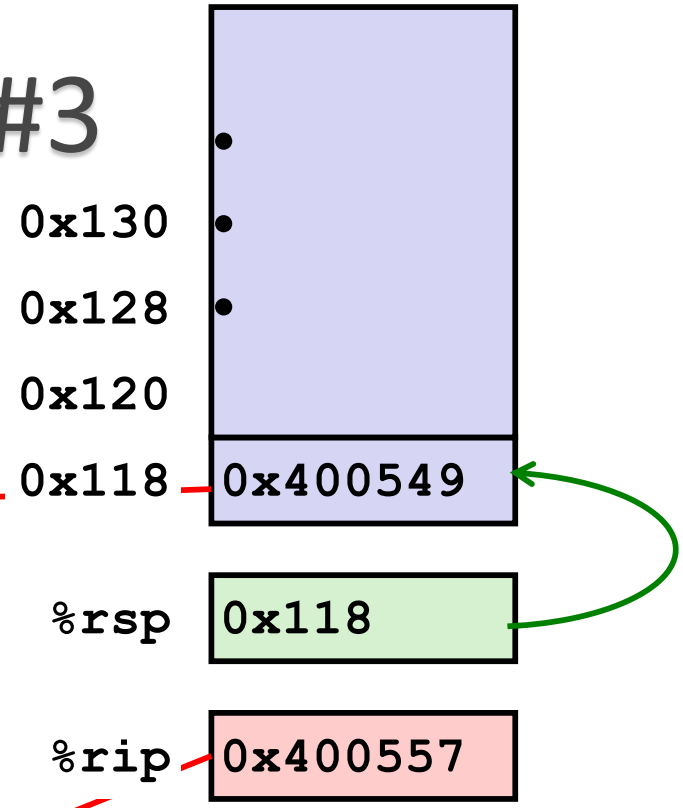
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



# Control Flow Example #3

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

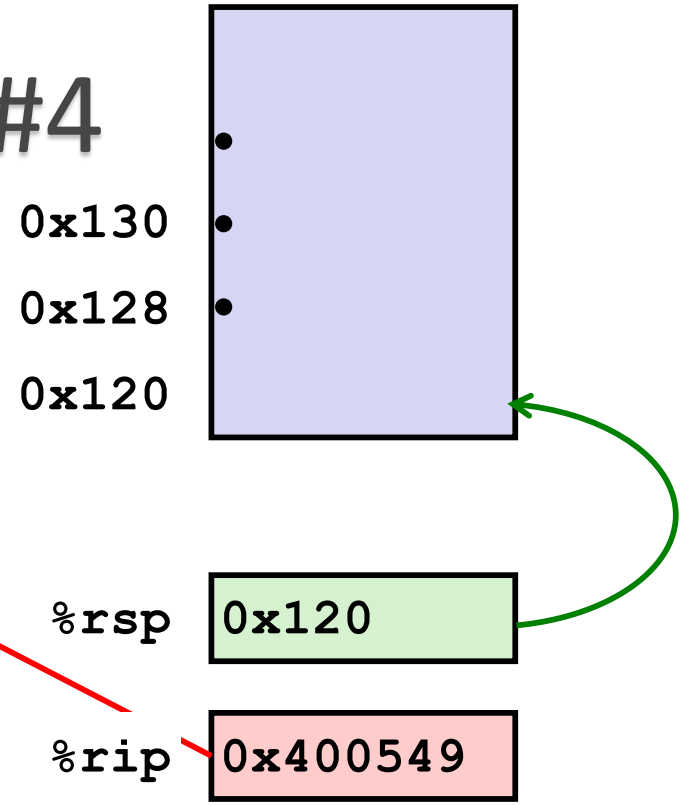
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



# Control Flow Example #4

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

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



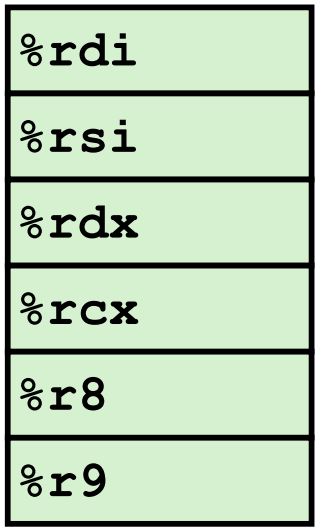
# Topics

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

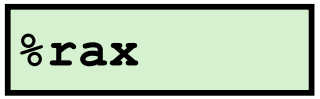
# Procedure Data Flow & conventions

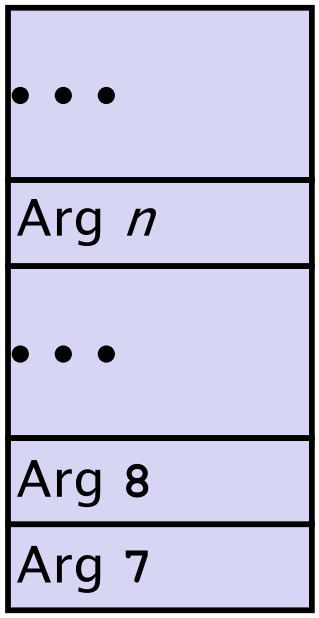
(sec. 3.7.3)

- First 6 arguments (in order)




- Return value




- Only allocate stack space for more args when needed

Registers

Stack

# 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
```

# Topics

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data (Secs. 3.7.4 and 3.7.5):** includes conventions for register usage and saving
  - Illustration of Recursion

# Stack-Based Languages

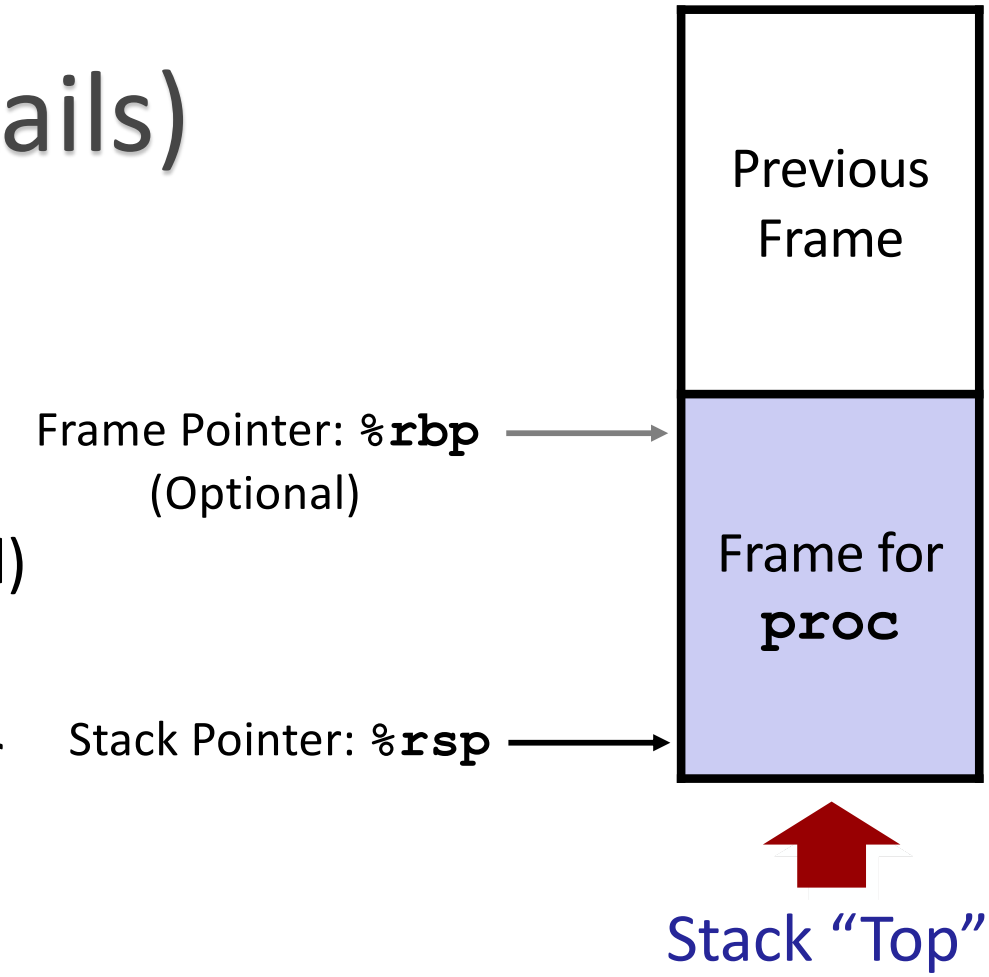
- Languages that support recursion (e.g., C, Pascal, Java)
  - Code must be "Reentrant"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
  - Caller same procedure as callee → procedure call is recursive
- Need to store state for recursive and non-recursive calls
- Stack discipline
  - State for given procedure needed for limited time
    - **From** when it was called **to** when it returns
  - Callee returns before caller does

# Stack Frames

- Each procedure call needs to save data
- The saved data goes in the *stack*
- Each procedure call allocates data in the Stack in
  - Frames, one stack frame for each procedure while executing
  - state for single procedure instantiation or call (regardless of the procedure call being recursive or not)

# Stack Frames (details)

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



# Call Chain Example

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

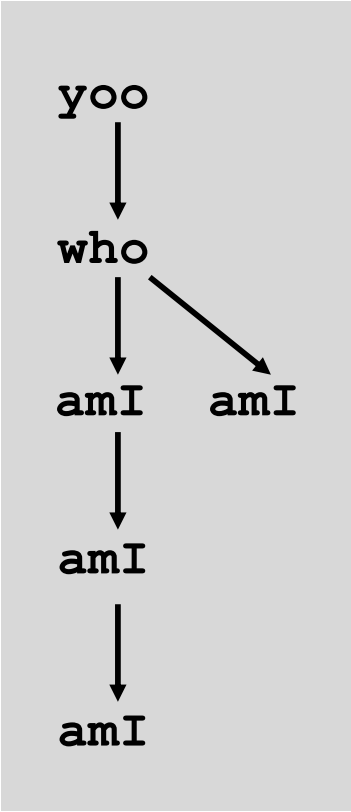
(high-level example of stack dynamic with procedure calls, recursive and non-recursive)

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

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

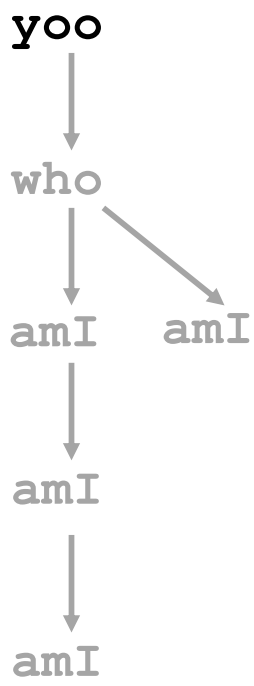
Procedure **amI()** is recursive

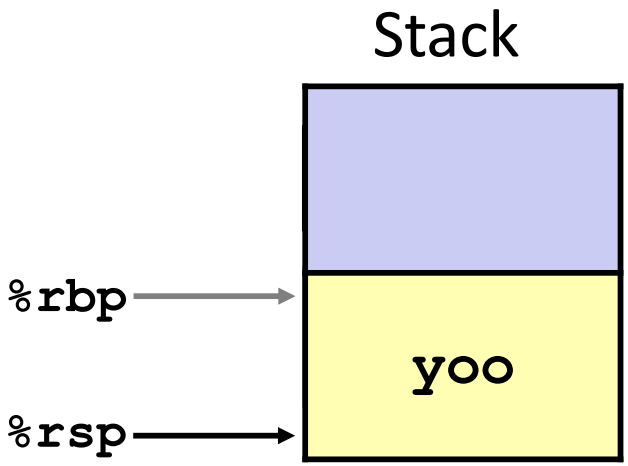
Example
Call Chain



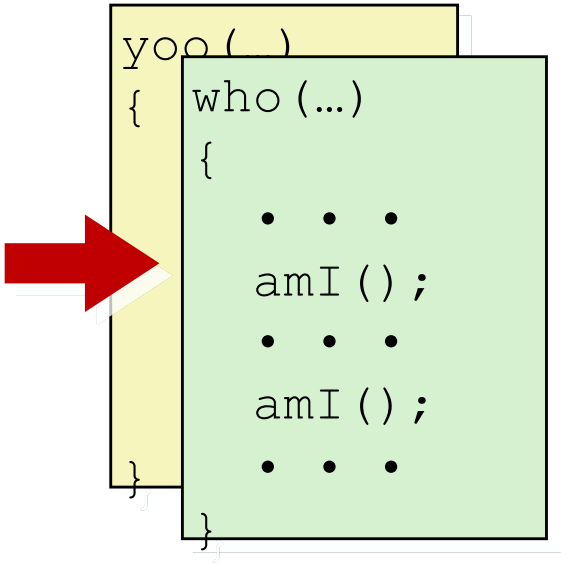
# Example

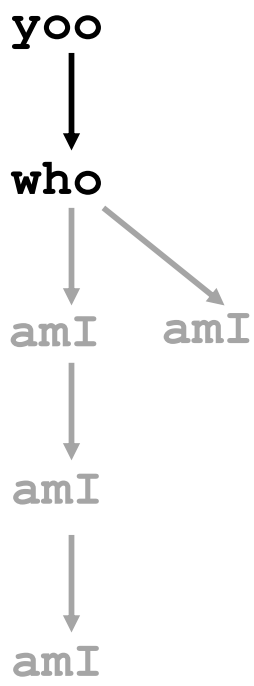
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

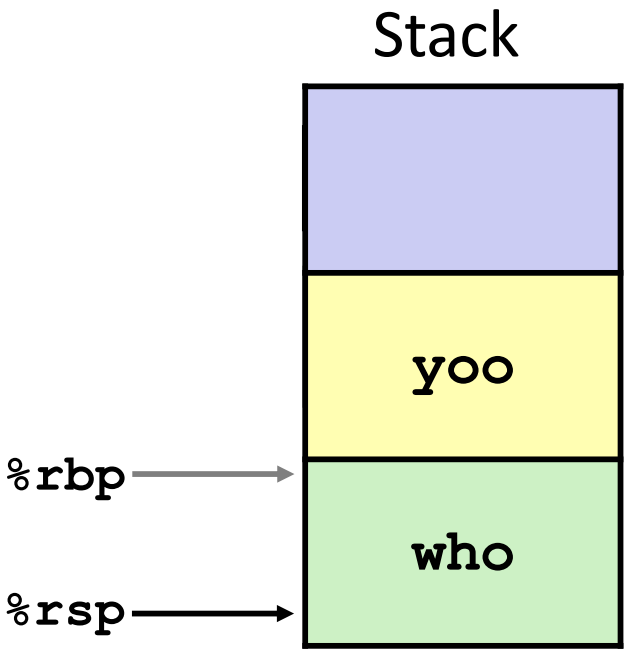


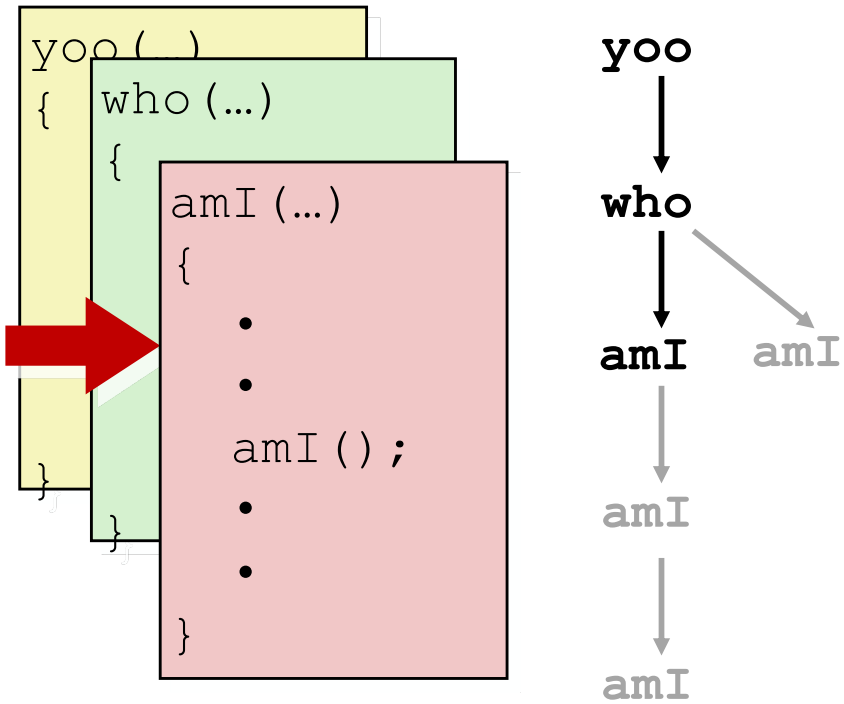
# Example



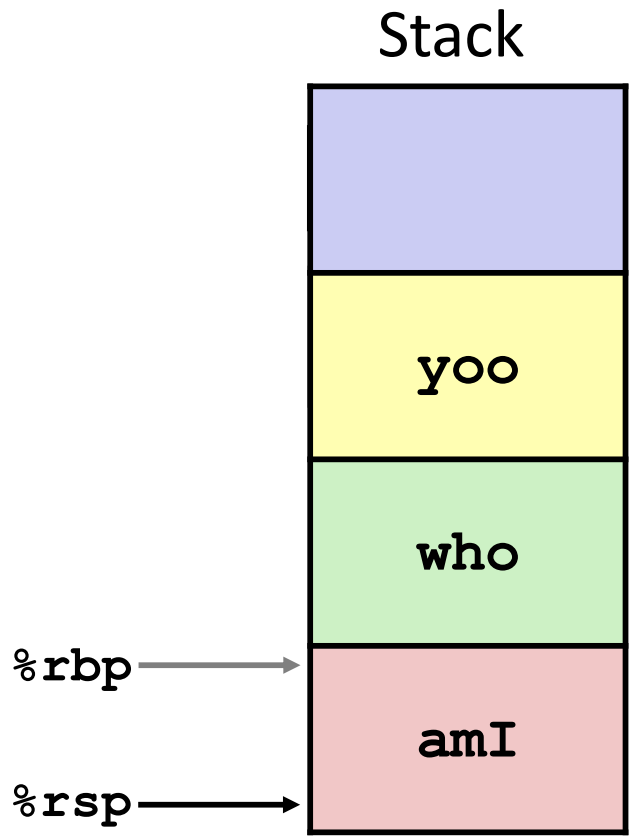



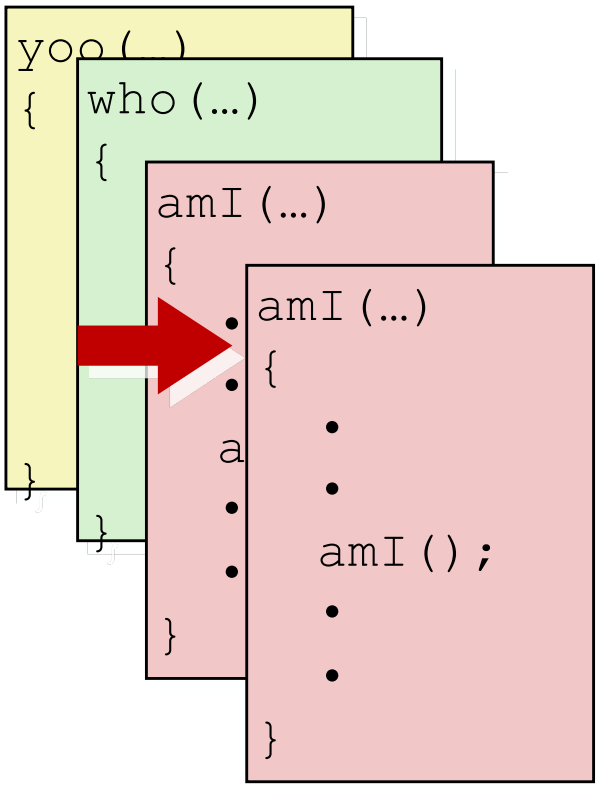
# Example

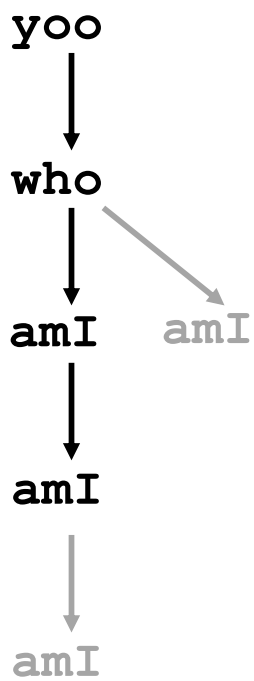

yoo(…)
{
}
who(…)
{
}
amI(…)
{
amI();
}

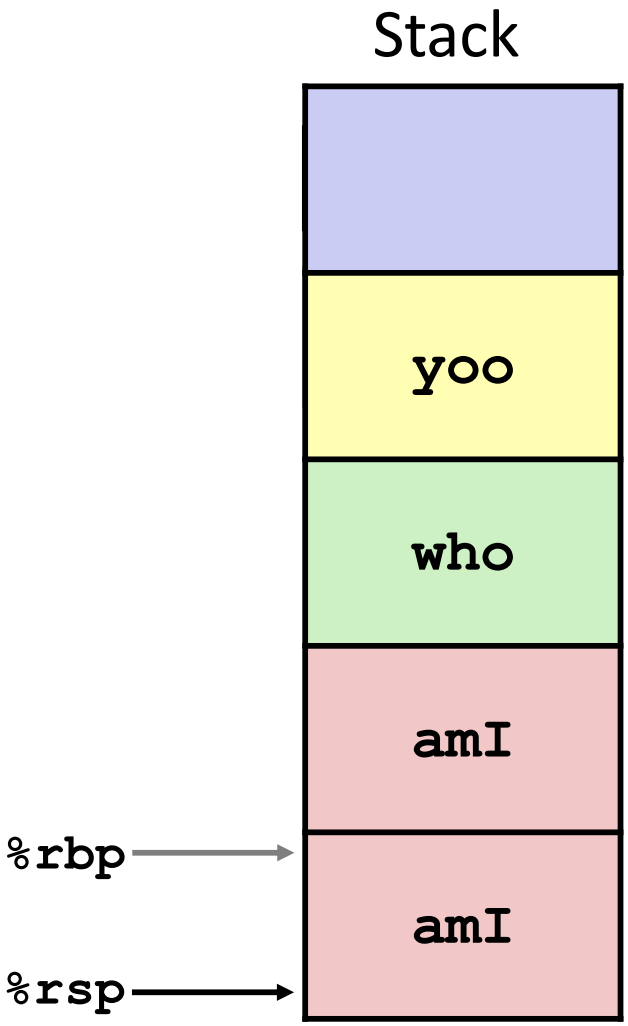

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

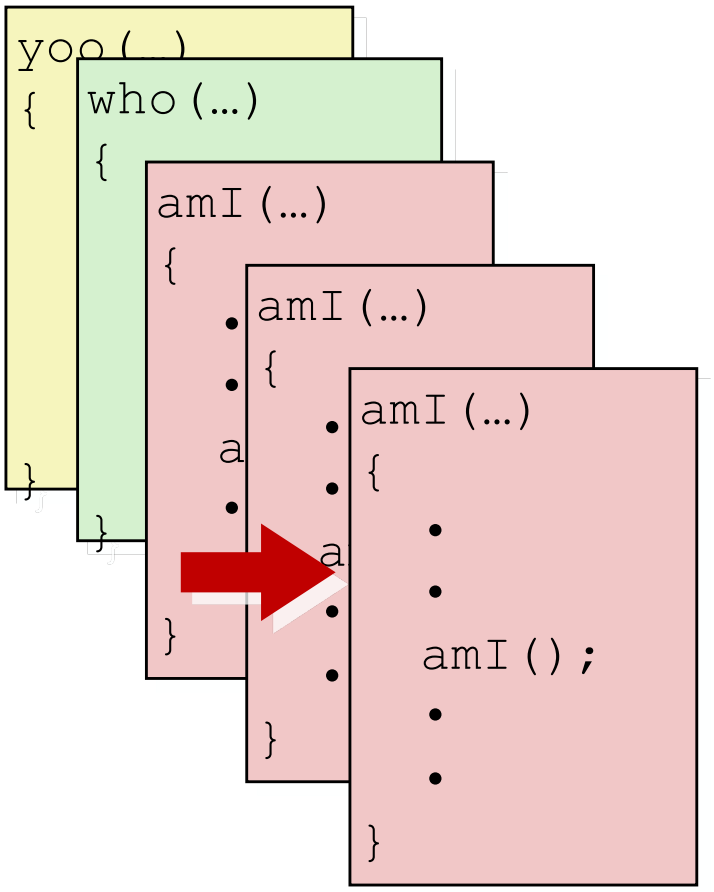
# Example

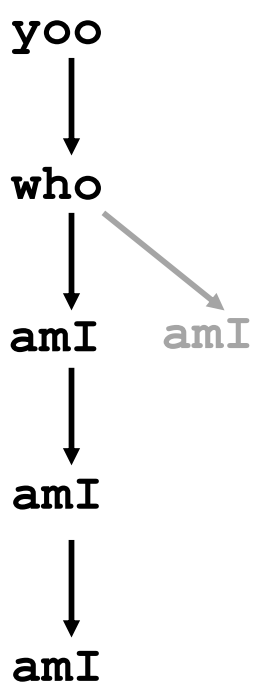

yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}

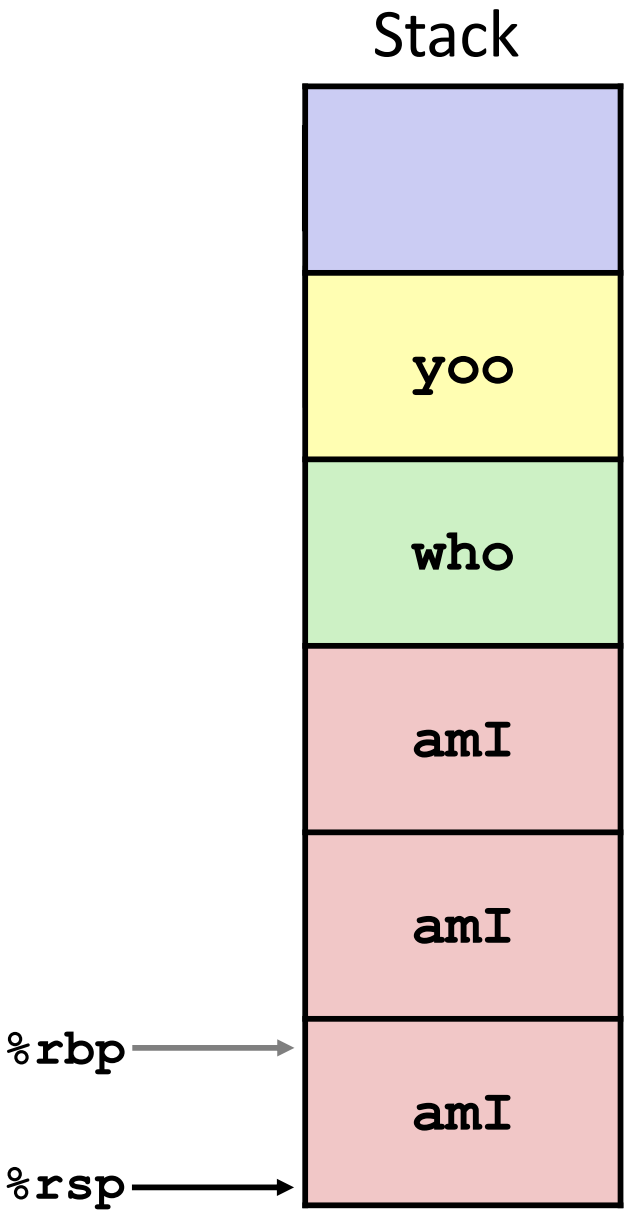

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

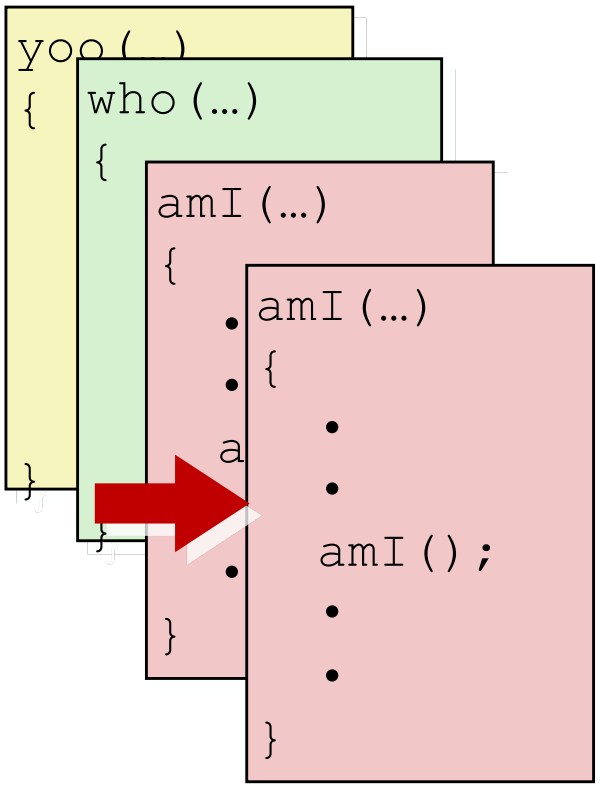
# Example

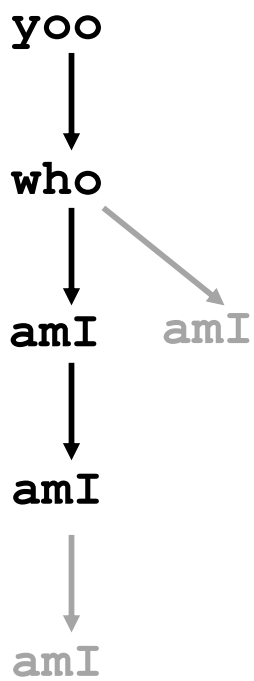

yoo(...)
{
}
who(...)
{
}
amI(...)
{
}
amI(...)
{
}
amI(...)
{
amI();
}

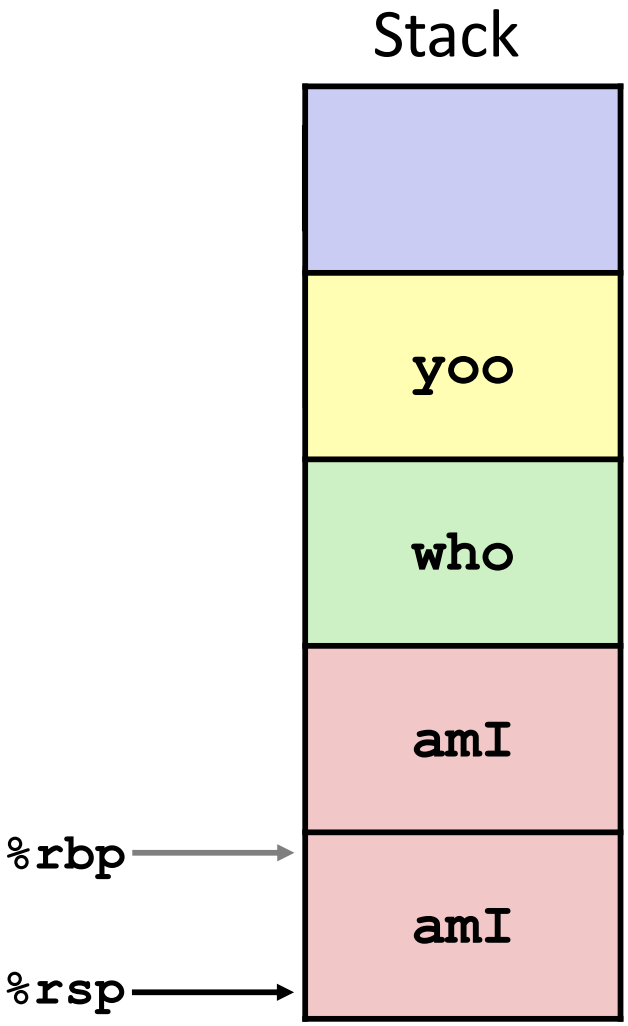

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

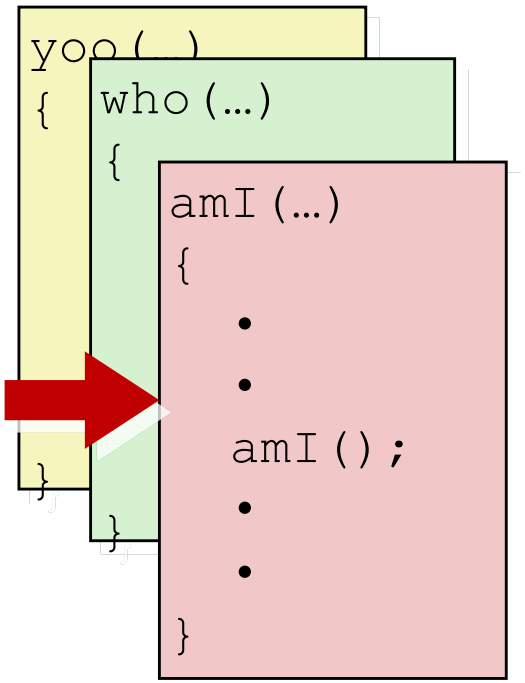
# Example

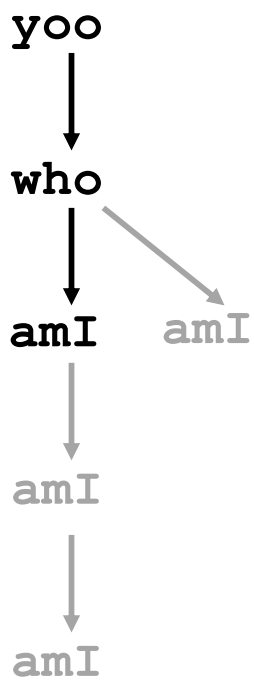

yoo(...)
{
}
who(...)
{
}
amI(...)
{
}
amI(...)
{
amI();
}

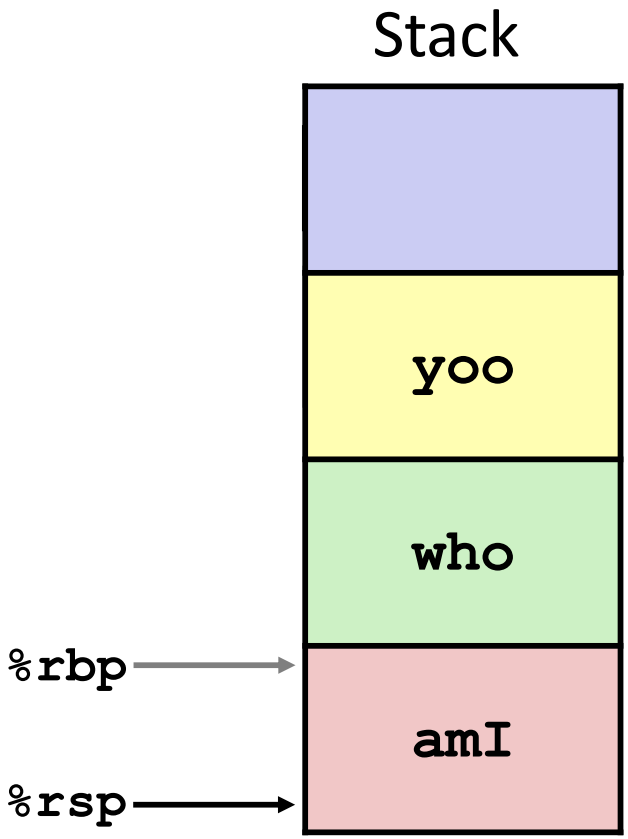

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

# Example

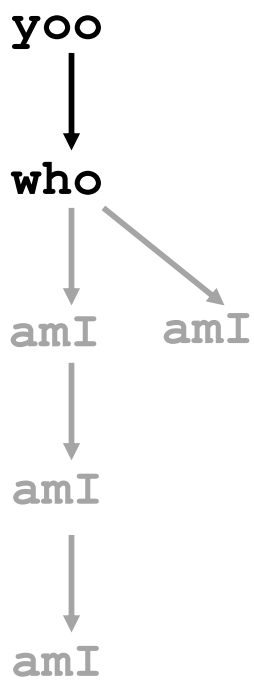

yoo(…)
{
}
who(…)
{
}
amI(…)
{
amI();
}

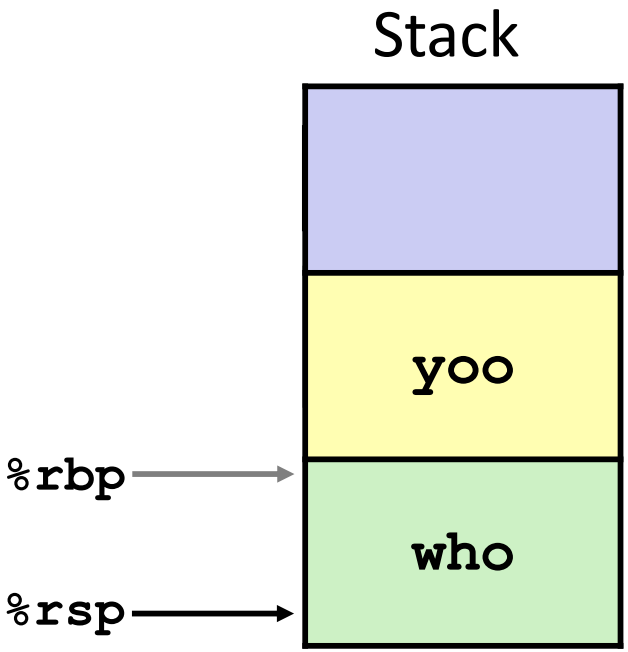

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

# Example

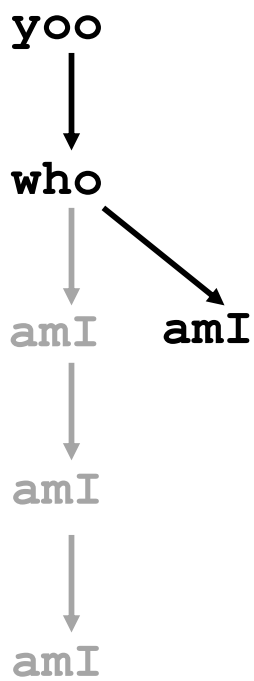
```
yoo(...)
{
    who(...)
    {
        • • •
        amI();
        • • •
        amI();
        • • •
    }
}
```

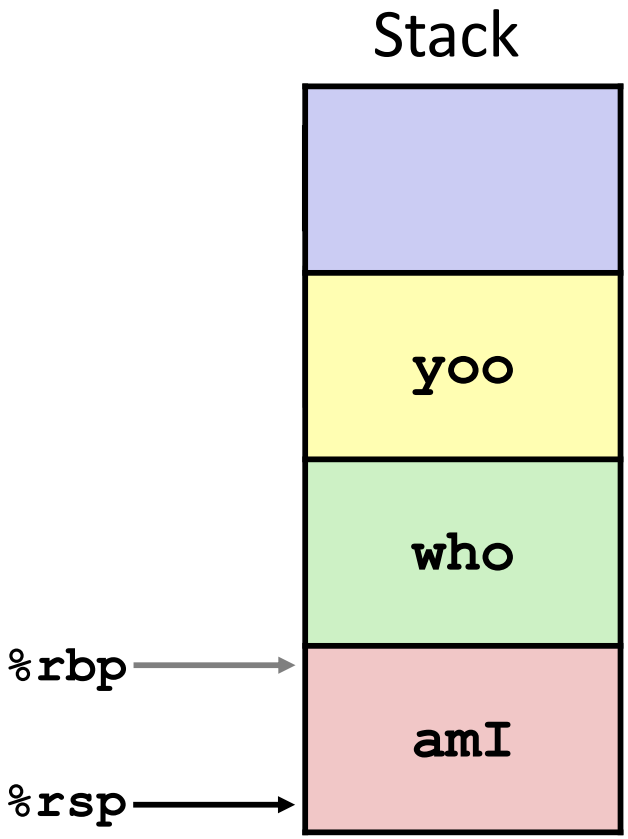


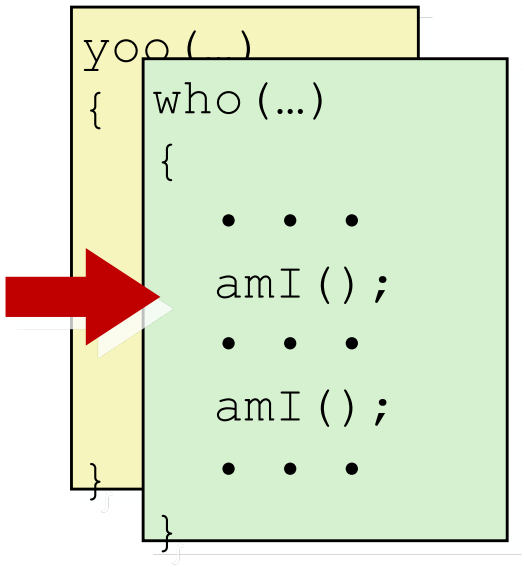
# Example

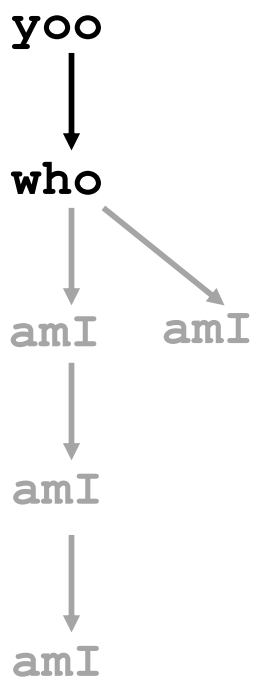
```
yoo(...)
{
    who(...)
    {
        amI(...)
        {
            •
            •
            amI();
            •
            •
        }
    }
}
```

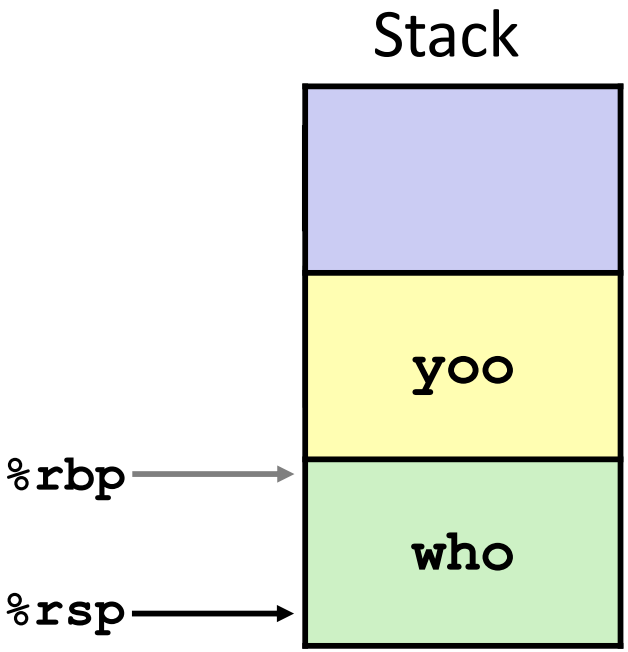


# Example

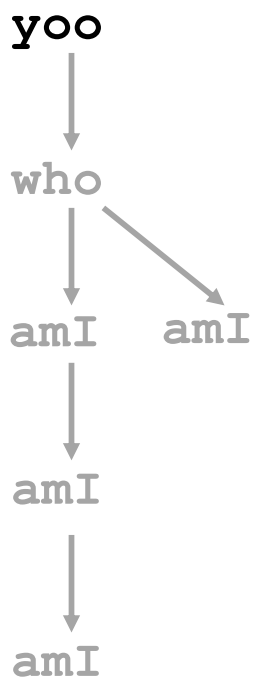


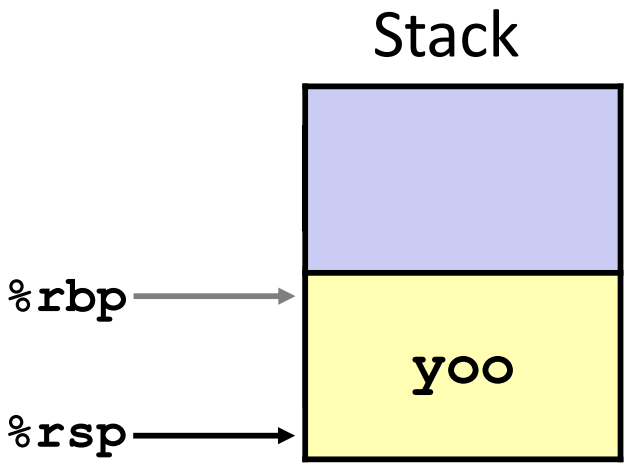



# Example

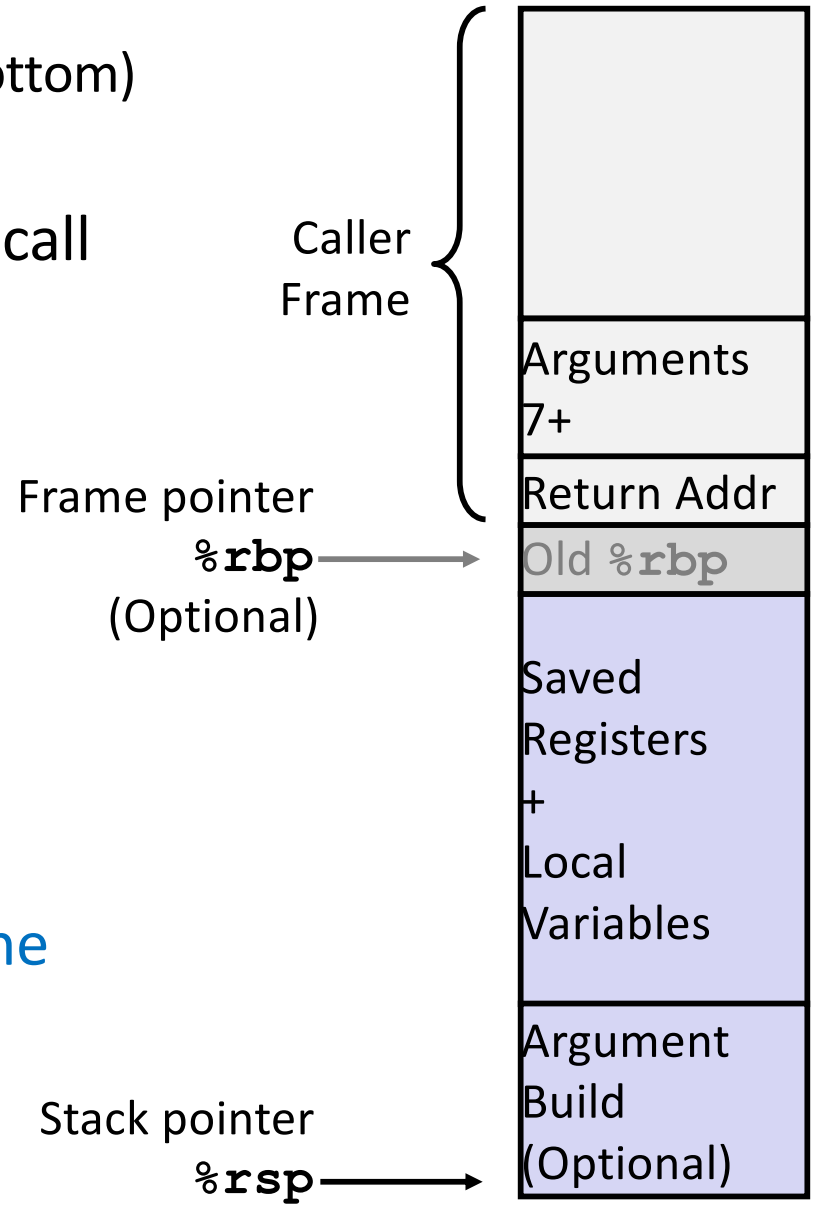
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```



# x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame
  - Return address
    - Pushed by **`call`** instruction
  - Arguments for this call (beyond the first 6 args. stored in registers)



# Topics

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
    - Complete the example of incr and call_incr
    - Caller-saved and callee-saved registers
    - Example
  - Illustration of Recursion

# Example: `incr`

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

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

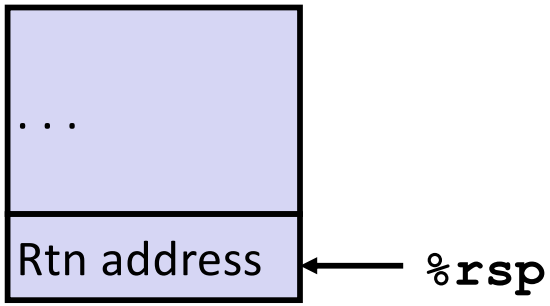
| Register | Use(s) |
|---|---|
| `%rdi` | Argument `p` |
| `%rsi` | Argument `val`, `y` |
| `%rax` | `x`, Return value |

# Example: Calling `incr` #1

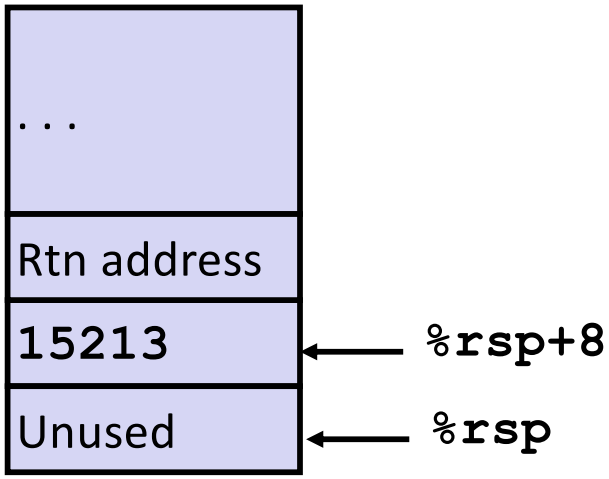
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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

Initial Stack Structure




Resulting Stack Structure

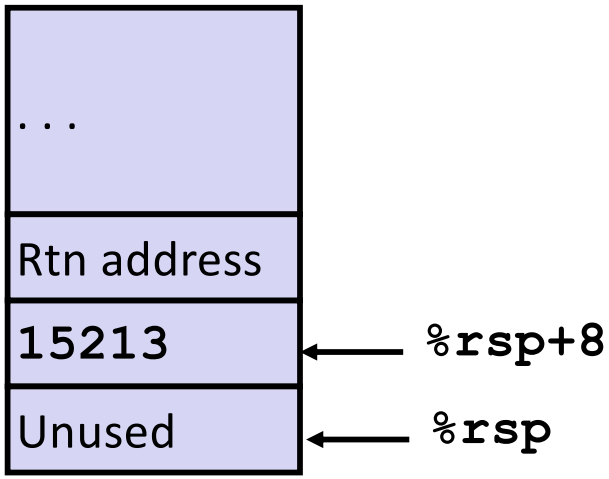


# Example: Calling `incr` #2

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

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

Stack Structure



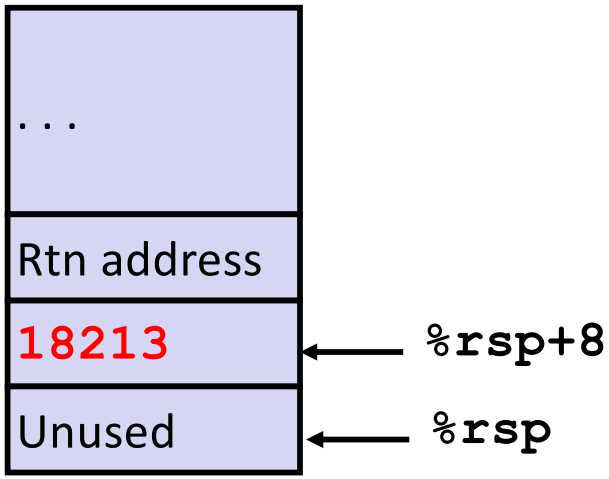

| Register | Use(s) |
|---|---|
| %rdi | &v1 |
| %rsi | 3000 |

# Example: Calling `incr` #3

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

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

Stack Structure



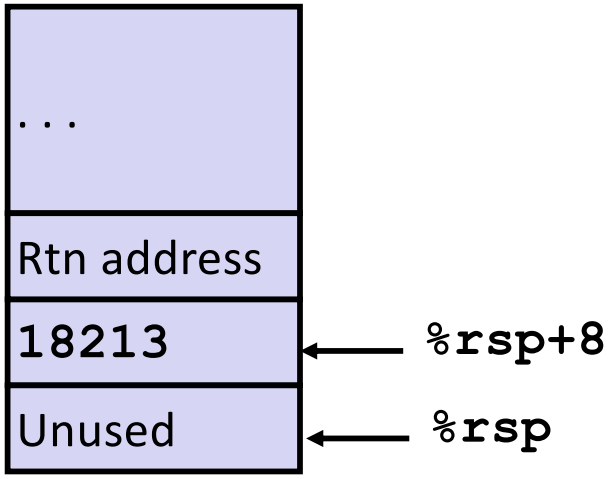

| Register | Use(s) |
|---|---|
| %rdi | &v1 |
| %rsi | 3000 |

# Example: Calling `incr` #4

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

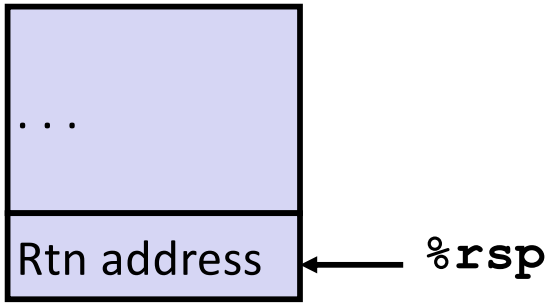
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure




| Register | Use(s) |
|---|---|
| `%rax` | Return value |

Updated Stack Structure

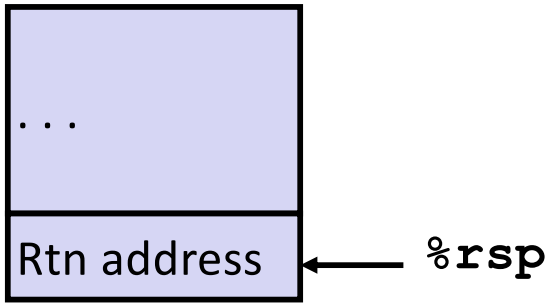


# Example: Calling `incr` #5

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

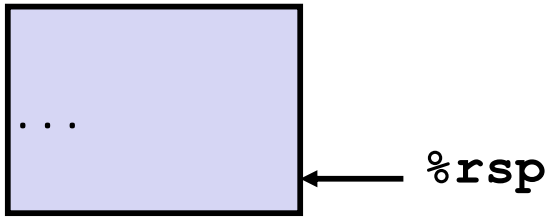
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure




| Register | Use(s) |
|---|---|
| `%rax` | Return value |

Final Stack Structure



# 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
```

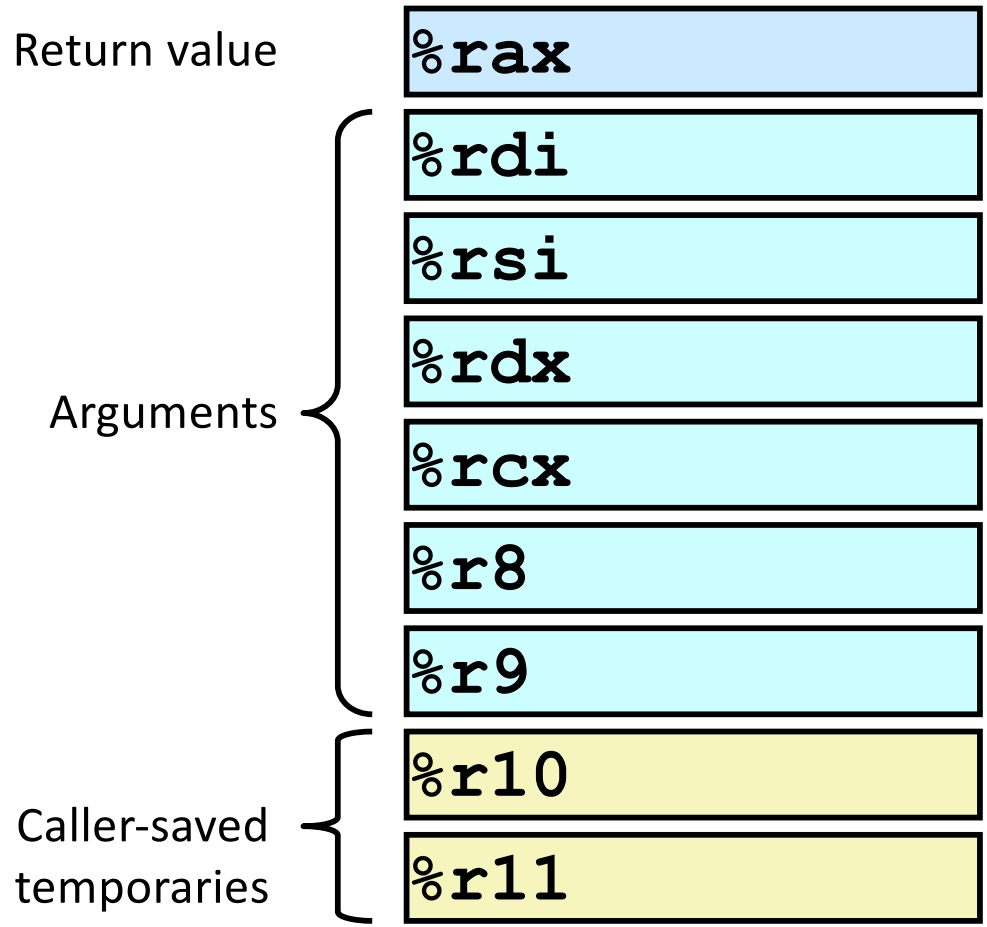
  - Contents of register **%rdx** overwritten by **who**
  - This could be trouble → something should be done!
    - Need some coordination

# Register Saving Conventions

- When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the caller
  - **`who`** is the callee
- Can register be used for temporary storage?
- Conventions (now can understand annotations on Fig. 3.2)
  - "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

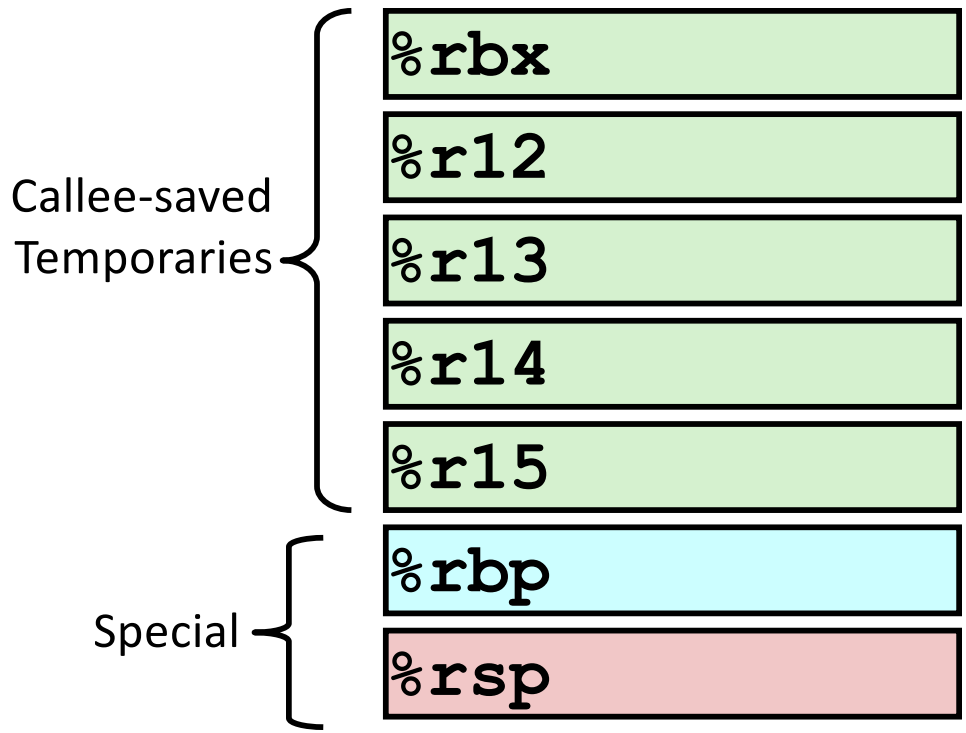
# x86-64 Linux Register Usage #1 (caller-saved)

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



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

- **%rbx, %r12, %r13, %r14, %r15**
  - 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

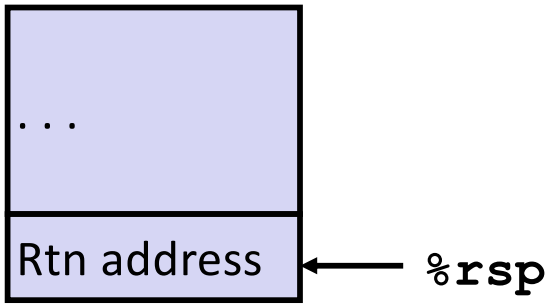


# Callee-Saved Example #1

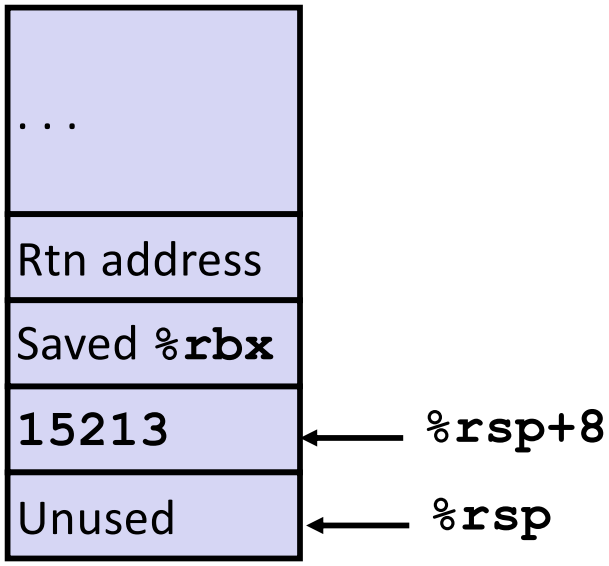
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

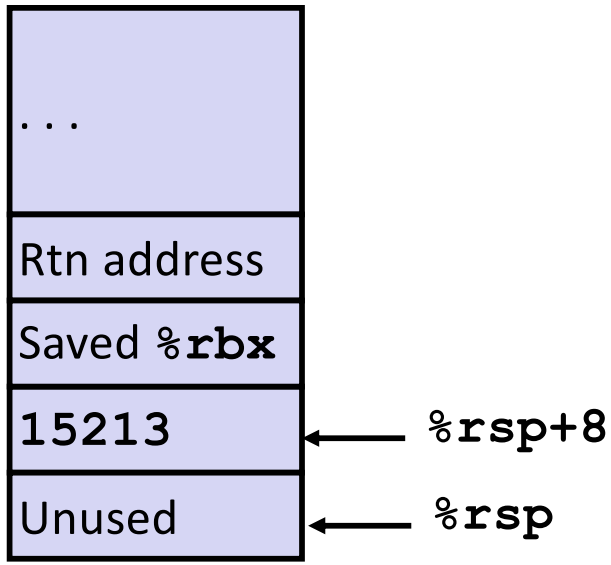


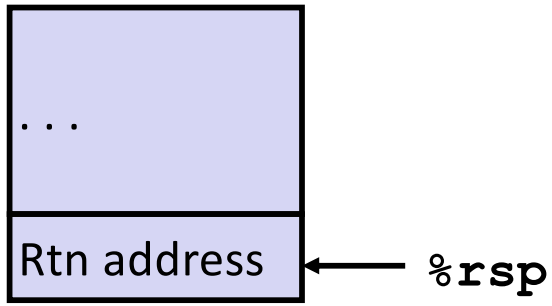
# Callee-Saved Example #2

```
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
```

Resulting Stack Structure




Pre-return Stack Structure