Machine Level Programming: Switch Statements and Procedures

CSCI 237: Computer Organization
11th Lecture, Mar 5, 2025

Jeannie Albrecht

Administrative Details

- Lab 2:
  - First three phases due this week (we won't grade anything until next week)
  - All phases due next week
  - If you're finished, check out phase 6 and secret phase
  - Or try another bomb! ☺
- HW 3 due Friday on Glow
- Sample midterm links fixed
- Alternate final exam date TBD
- No class or office hours on Friday (but please watch video!)
  - Office hours Thur 2:30-3:30 instead

Last Time

- Discussed data dependent control
  - Loops!

- (I felt like many of you were confused during Monday’s lecture. Don’t fall behind! Get help if you need it.)

Today

- Wrap up data dependent control
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

Switch Statements

- Switch statements evaluate a given expression and execute statements based on the evaluated value
- Used to perform different actions based on different conditions (or cases)
- Alternative to long if-else if statements that compare a single variable to several values
- A “multiway branch” statement that provides an easy way to dispatch execution to different parts of code based on the value of the expression

Switch Statements

```
switch(expression) {
   case value1:
       statement_1;
       break;
   case value2:
       statement_2;
       break;
   .
   .
   .
   case value_n:
       statement_n;
       break;
   default:
       default_statement;
}
```

- Rules
  - The “case value” must be of “char” or “int” type in C
  - There can be one or N number of cases
  - The values in the case must be unique
  - Each statement of the case can have an *optional* break statement
  - The default statement is also optional

Switch Statement

```
int var = 1;
switch (var) {
   case 1:
       printf("Case 1 is Matched.");
       break;

   case 2:
       printf("Case 2 is Matched.");
       break;

   case 3:
       printf("Case 3 is Matched.");
       break;

   default:
       printf("Default case is Matched.");
}
```

Logically the same as:

```
if (var == 1) {
   printf("Case 1 is Matched.");
else if (var == 2)
   printf("Case 2 is Matched.");
. . .
else {
   printf("Default case is Matched.");
}
```

Switch Statements

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

JTab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block n–1

Translation (Extended C)

```
goto *JTab[x];
```

gcc often translates switch statements into "extended C" with jump tables (when # cases > 4) -> *indirect jumping*

Switch Statement Example

C code

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
       . . .
    }
    return w;
}
```

Jump table in x86-64

```
.section   .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Switch statement in x86-64:

Direct jump ➡ ja

Indirect jump ➡ jmp

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default, goto L8
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

Interpreting the Jump Table

Jump table in x86-64

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

```
switch(x) {
 case 1:       // .L3
   w = y*z;
   break;
 case 2:       // .L5
   w = y/z;
   /* Fall Through */
 case 3:       // .L9
   w += z;
   break;
 case 5:
 case 6:       // .L7
   w -= z;
   break;
 default:      // .L8
   w = 2;
}
```

Code Blocks (x == 1)

```
switch(x) {
 case 1:          // .L3
   w = y*z;
   break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

```
.section   .rodata
  .align 8
.L4:
  .quad    .L8   # x = 0
  .quad    .L3   # x = 1
  .quad    .L5   # x = 2
  .quad    .L9   # x = 3
  .quad    .L8   # x = 4
  .quad    .L7   # x = 5
  .quad    .L7   # x = 6
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Code Blocks (x == 2, x == 3)

```
long w = 1;
. . .
switch(x) {
 . . .
 case 2:
   w = y/z;
   /* Fall Through */
 case 3:
   w += z;
   break;
 . . .
}
```

```
.L5:                      # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx           #  y/z
   jmp     .L6            #  goto merge
.L9:                      # Case 3
   movl    $1, %eax       #  w = 1
.L6:                      # merge:
   addq    %rcx, %rax     #  w += z
   ret
```

```
.section        .rodata
  .align 8
.L4:
  .quad         .L8    # x = 0
  .quad         .L3    # x = 1
  .quad         .L5    # x = 2
  .quad         .L9    # x = 3
  .quad         .L8    # x = 4
  .quad         .L7    # x = 5
  .quad         .L7    # x = 6
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

Code Blocks (x == 5, x == 6, default)

```
switch(x) {
 . . .
 case 5:        // .L7
 case 6:        // .L7
   w -= z;
   break;
 default:       // .L8
   w = 2;
}
```

```
.L7:                      # Case 5,6
  movl   $1, %eax    #  w = 1
  subq   %rdx, %rax  #  w -= z
  ret
.L8:                      # Default:
  movl   $2, %eax    #  w = 2
  ret
```

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Summary of Ch 3.6

- C Data-dependent Control
  - if-then-else, do-while, while, for, switch
- Assembler Control
  - Conditional jump, Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use “decision trees” (if-else if-else if-else)

Today

- Wrap up data dependent control
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

Mechanisms in Procedures

- Passing control
  - To beginning of procedure (function) code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with assembly instructions

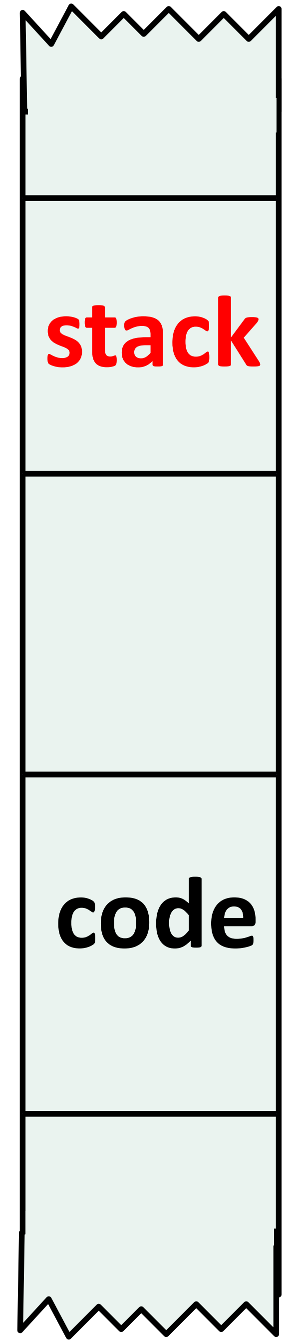
```
void P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

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

x86-64 Stack

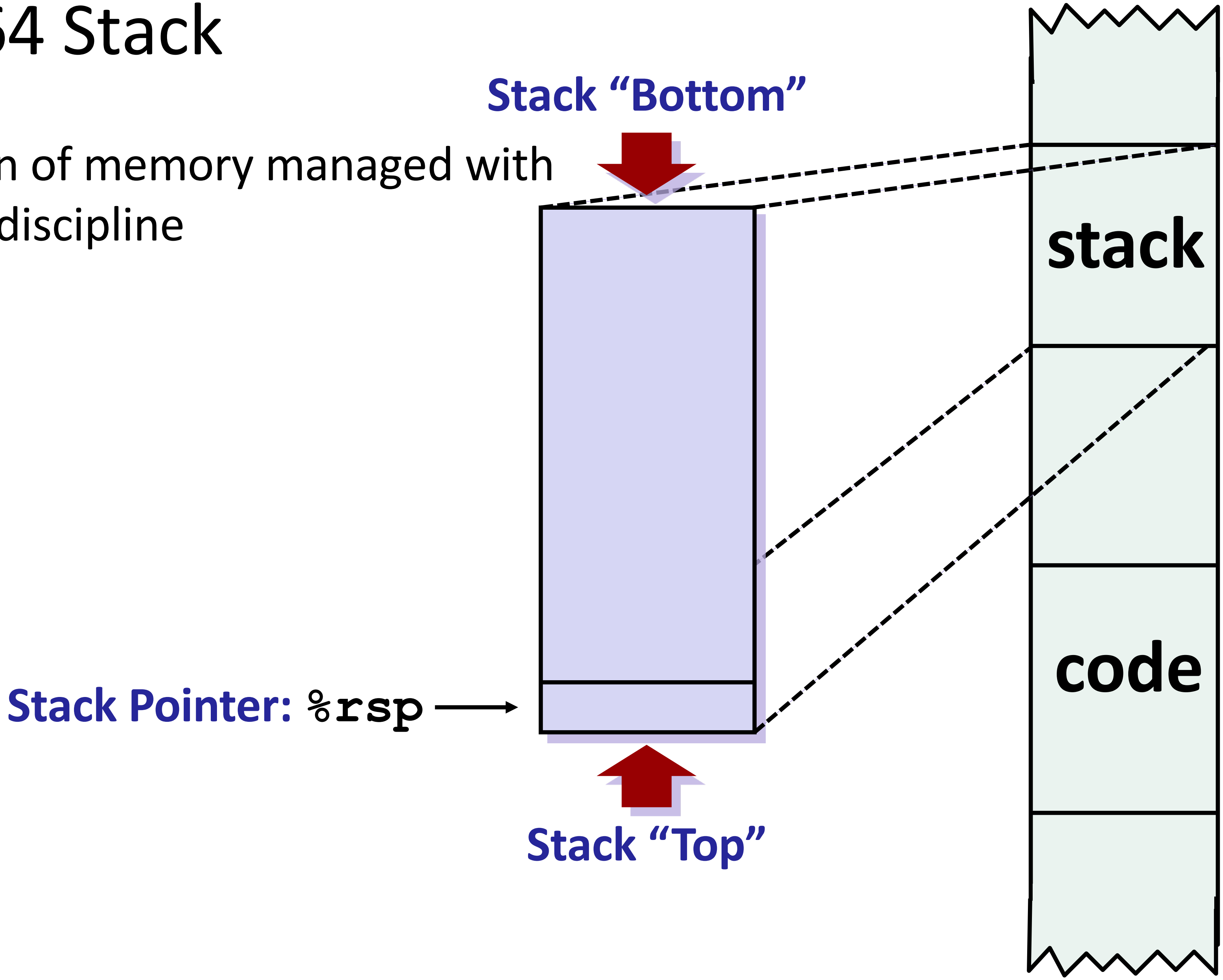
- Region of memory managed with stack discipline (access using push and pop)

- Memory viewed as array of bytes
- Different regions have different purposes
- Stack portion of memory facilitates passing info between procedures (among other things)



x86-64 Stack

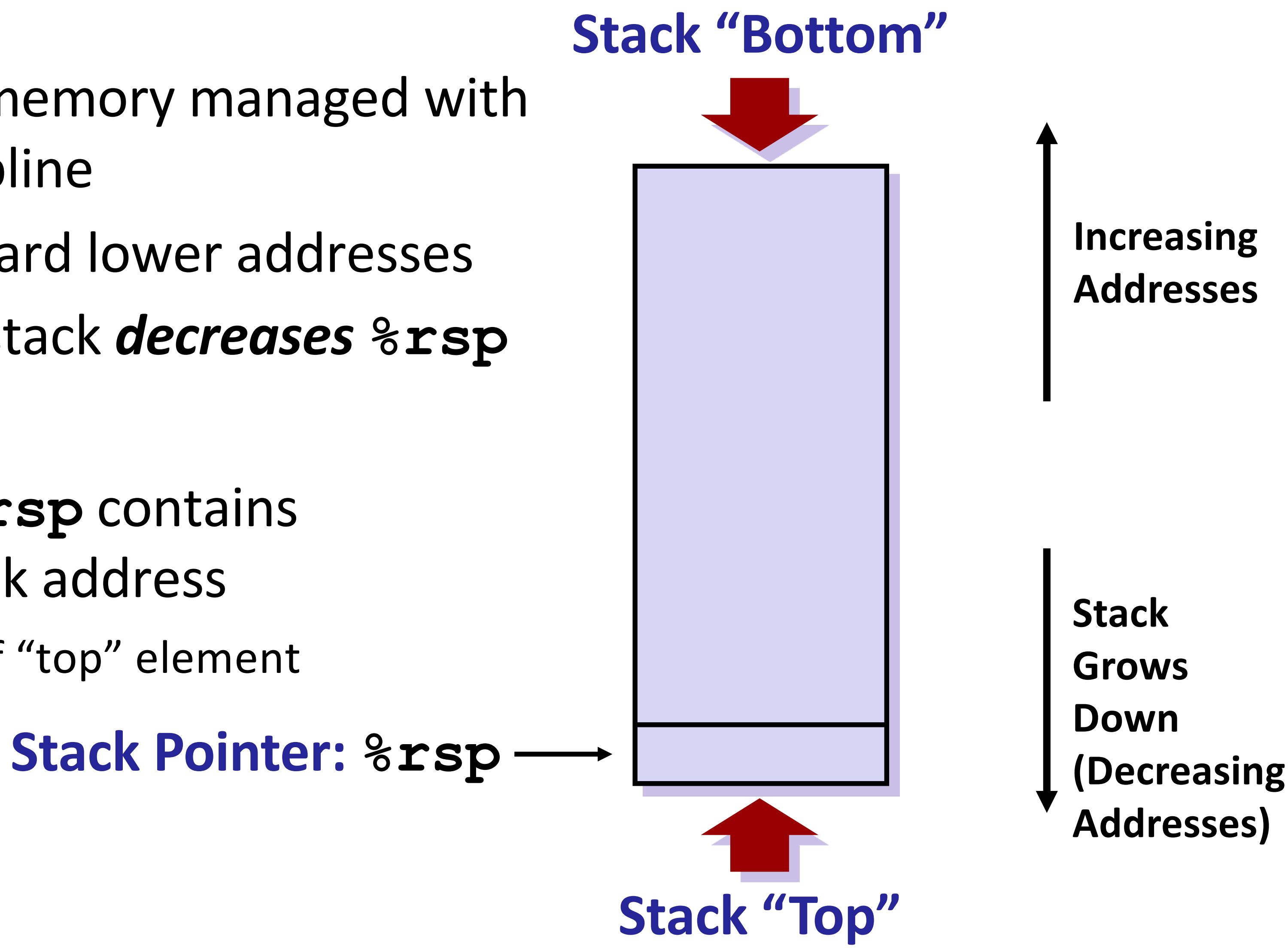
- Region of memory managed with stack discipline



x86-64 Stack

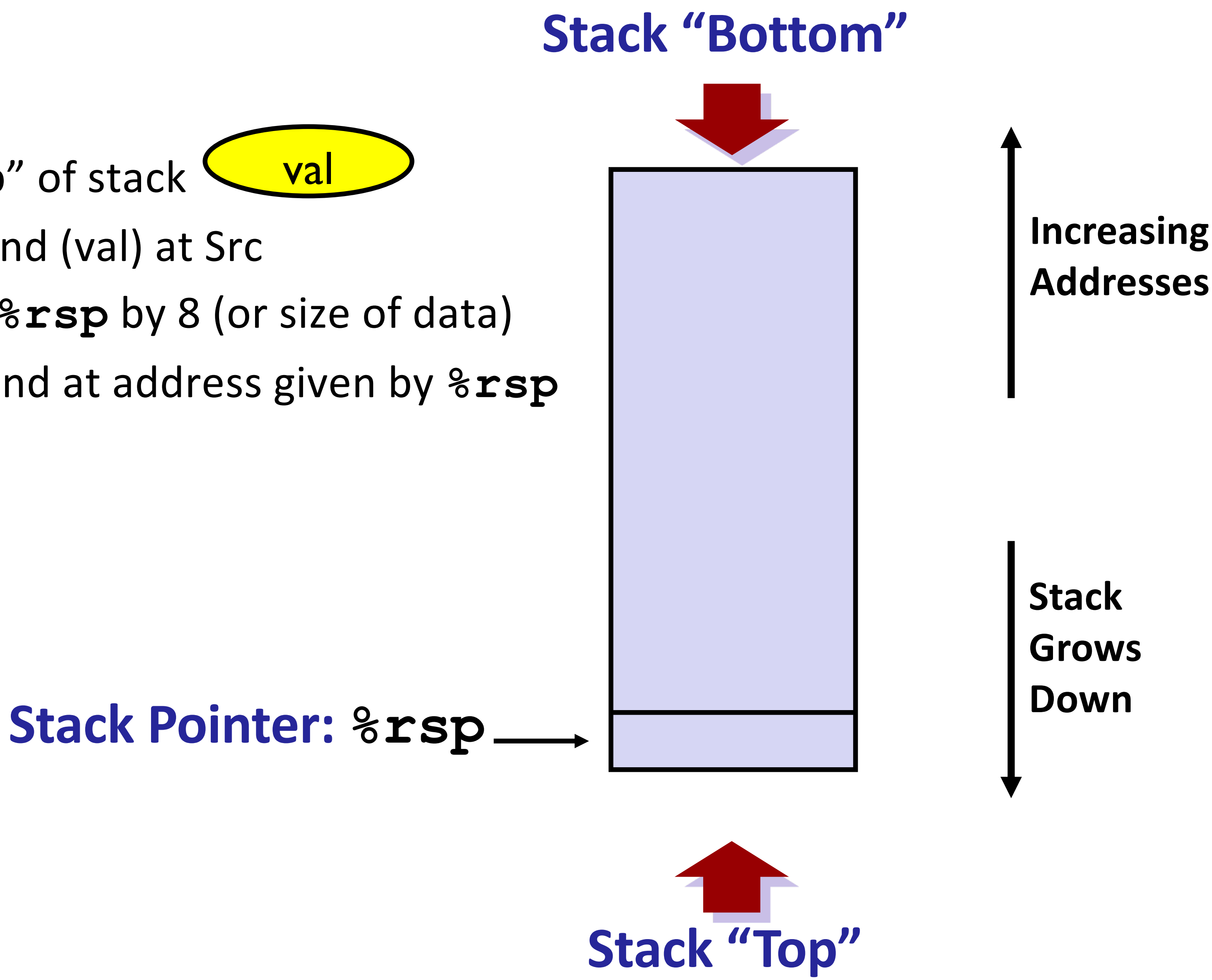
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Adding to stack ***decreases*** **`%rsp`**

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



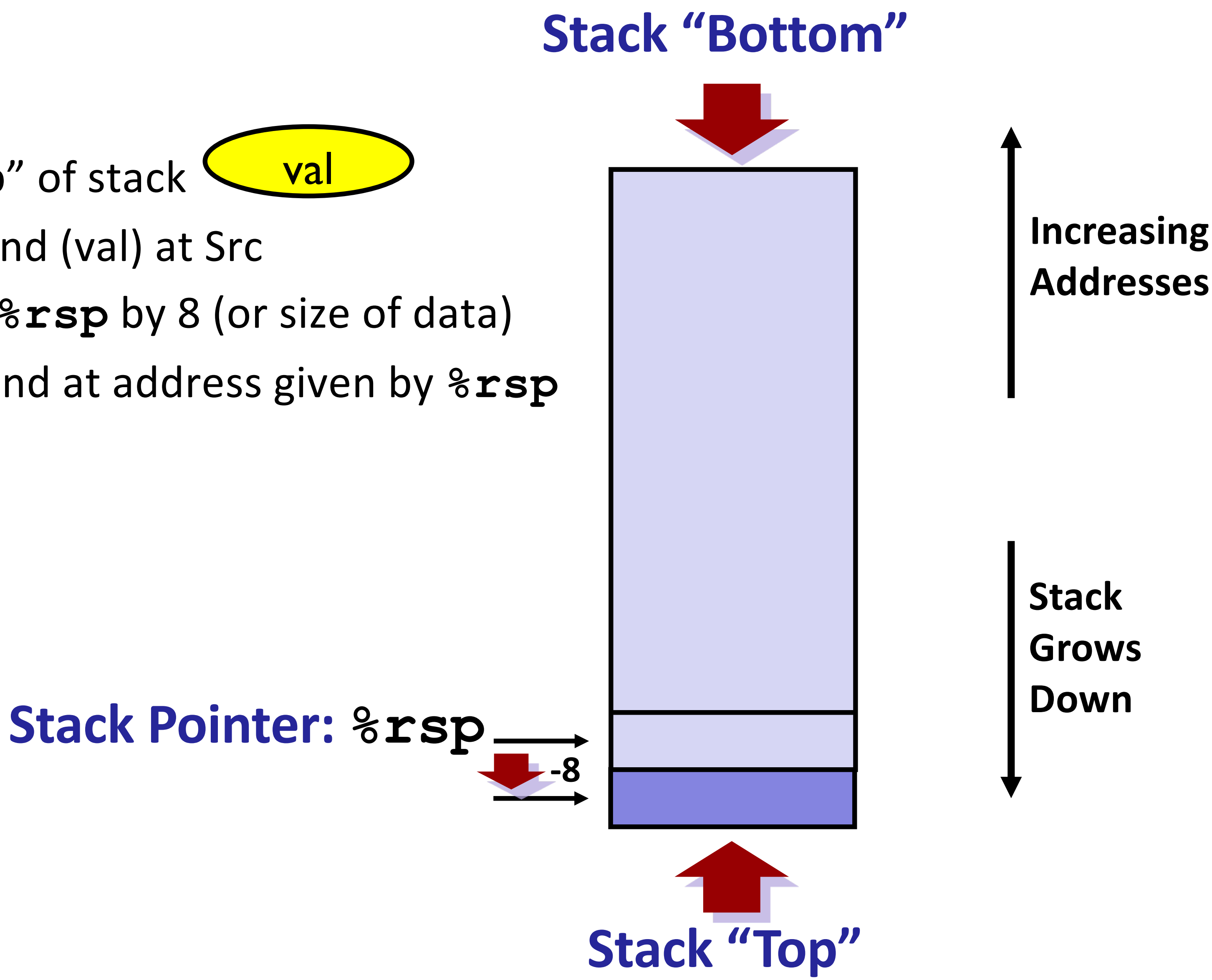
x86-64 Stack: Push

- **`pushq`** Src
  - Add val to "top" of stack (val)
    - Fetch operand (val) at Src
    - Decrement **`%rsp`** by 8 (or size of data)
    - Write operand at address given by **`%rsp`**



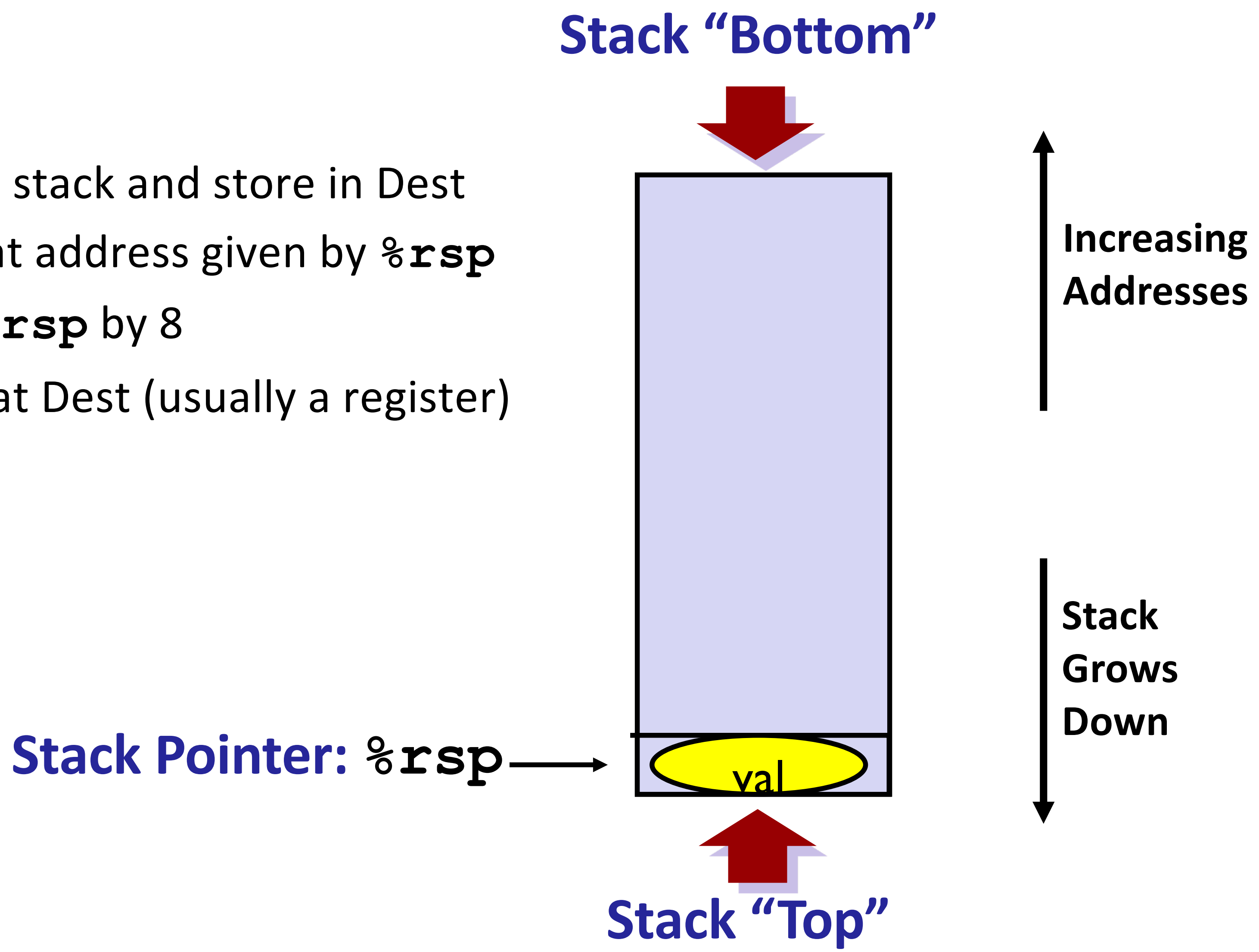
x86-64 Stack: Push

- **`pushq`** Src
  - Add val to "top" of stack
    - Fetch operand (val) at Src
    - Decrement **`%rsp`** by 8 (or size of data)
    - Write operand at address given by **`%rsp`**



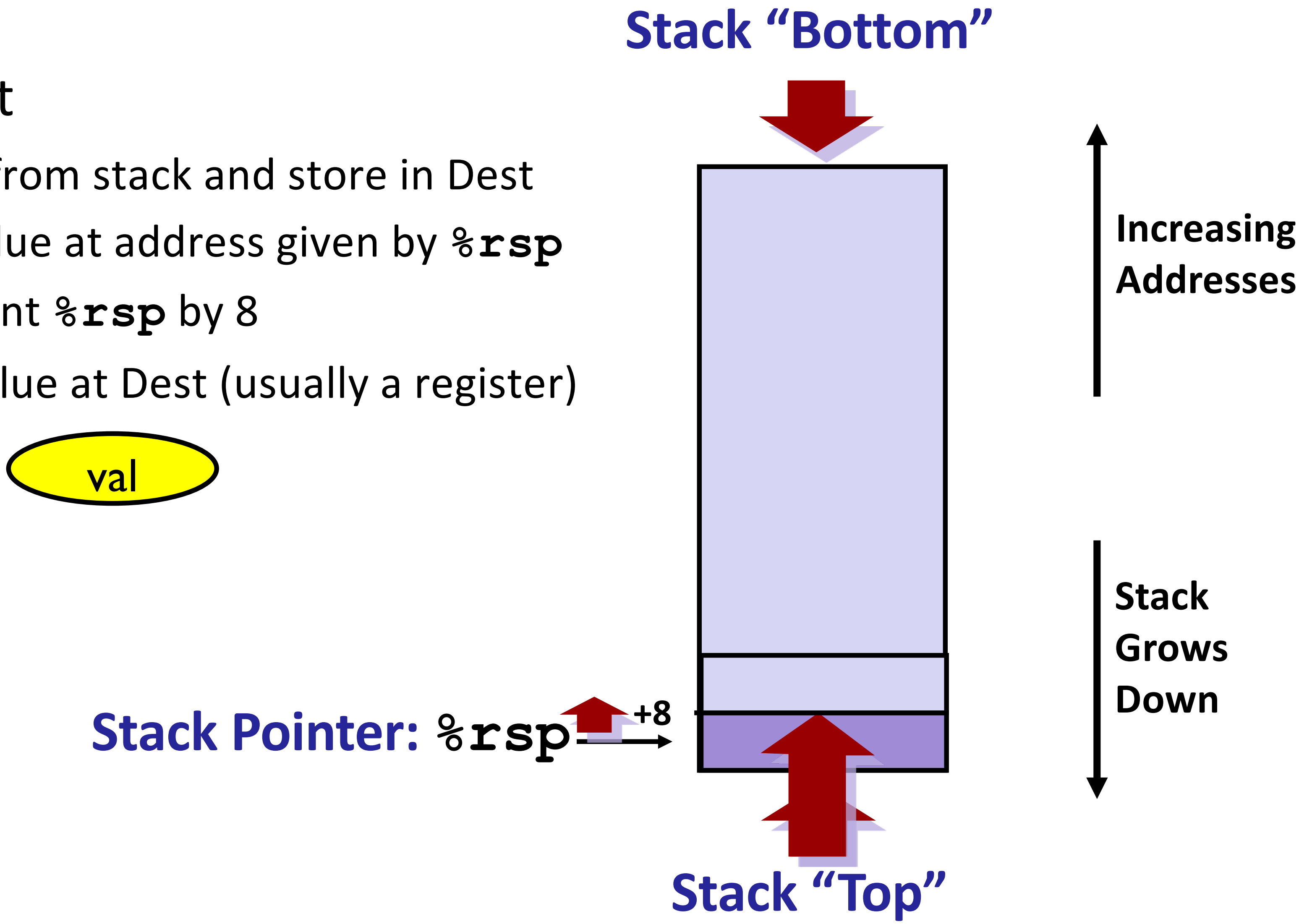
x86-64 Stack: Pop

- **`popq`** Dest
  - Pop value from stack and store in Dest
    - Read value at address given by **`%rsp`**
    - Increment **`%rsp`** by 8
    - Store value at Dest (usually a register)



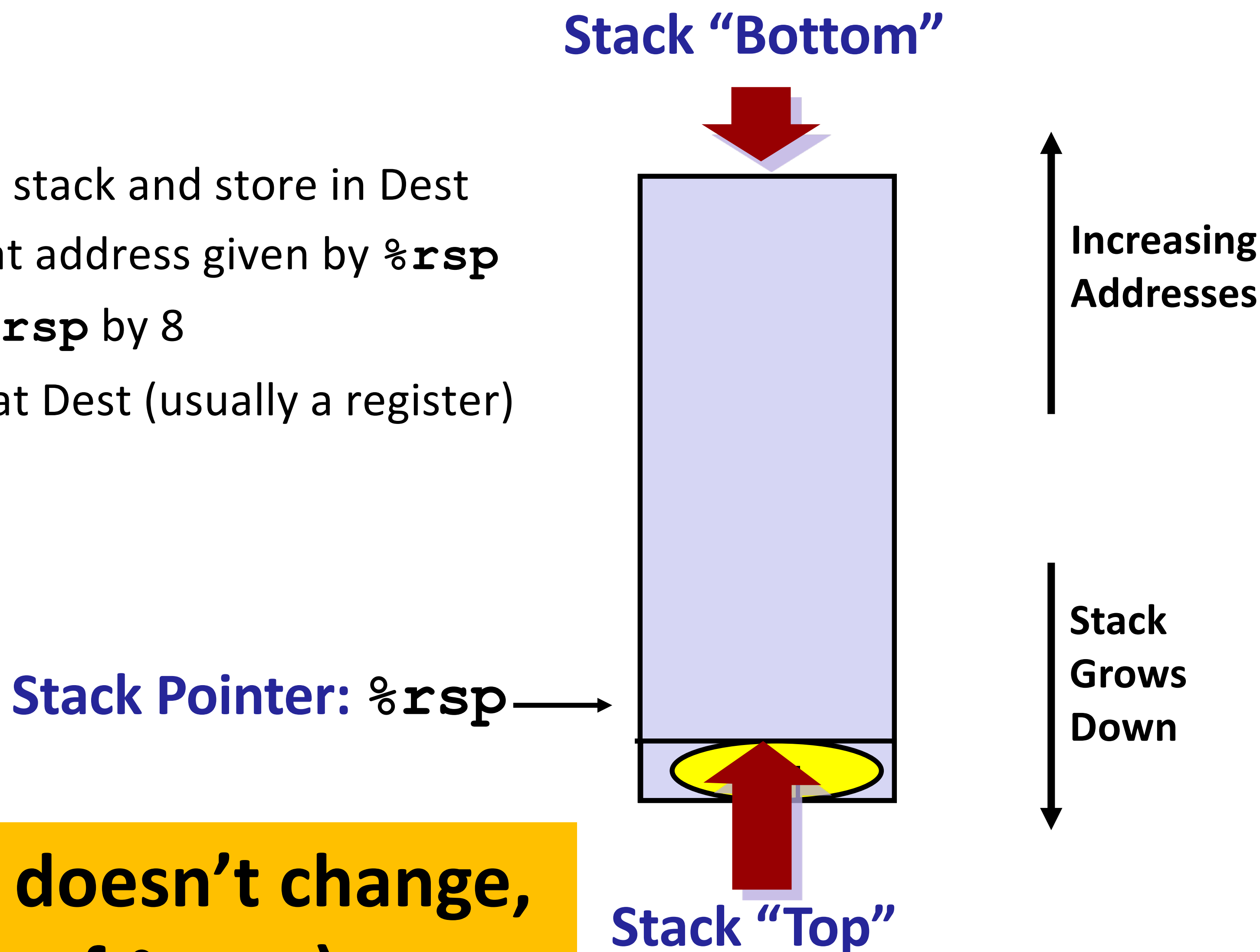
x86-64 Stack: Pop

- **`popq`** Dest
  - Pop value from stack and store in Dest
    - Read value at address given by `%rsp`
    - Increment `%rsp` by 8
    - Store value at Dest (usually a register)



x86-64 Stack: Pop

- **`popq`** Dest
  - Pop value from stack and store in Dest
    - Read value at address given by **`%rsp`**
    - Increment **`%rsp`** by 8
    - Store value at Dest (usually a register)



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

Today

- Wrap up data dependent control
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - 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
```

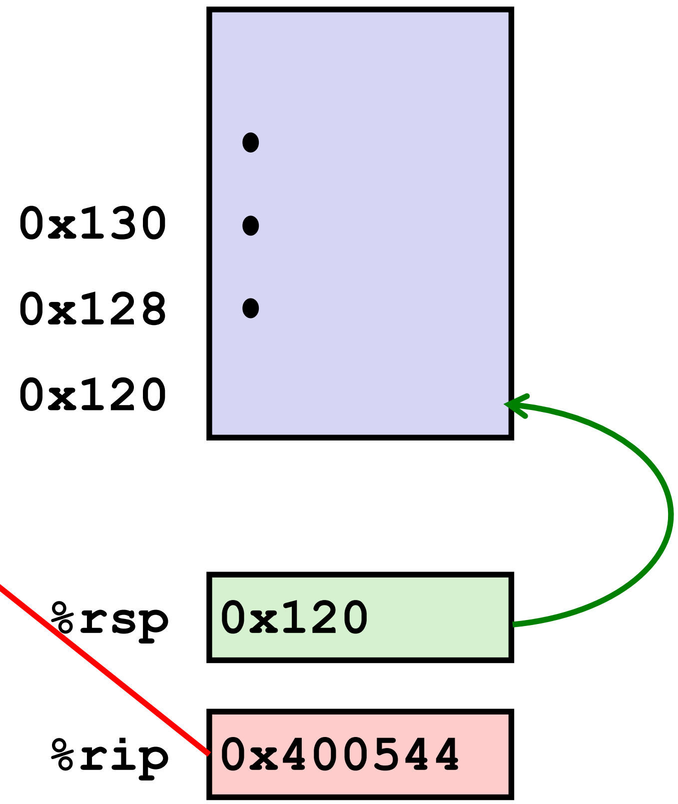
Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - Push return address on stack (which decrements **`%rsp`**)
  - Jump to label (address of function being called)
- Return address:
  - Address of the next instruction right after function call returns
  - (Example on next slide)
- Procedure return: **`ret`**
  - Pop return address from stack (which increments **`%rsp`**)
  - Jump to address (back in calling function)

Control Flow Example #1

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

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

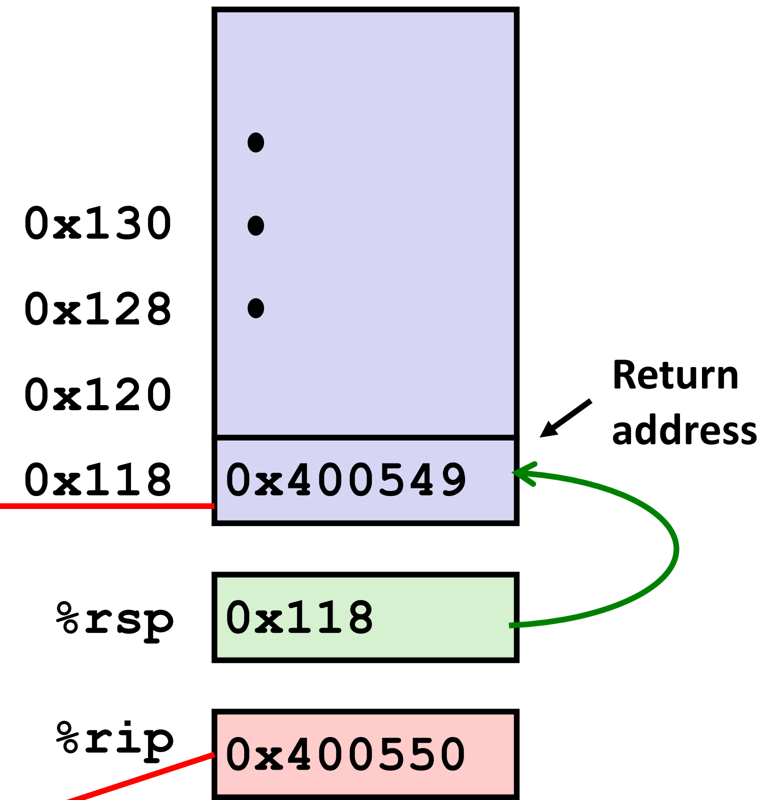


callq: Push return addr onto stack, jump to label (address).

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
```

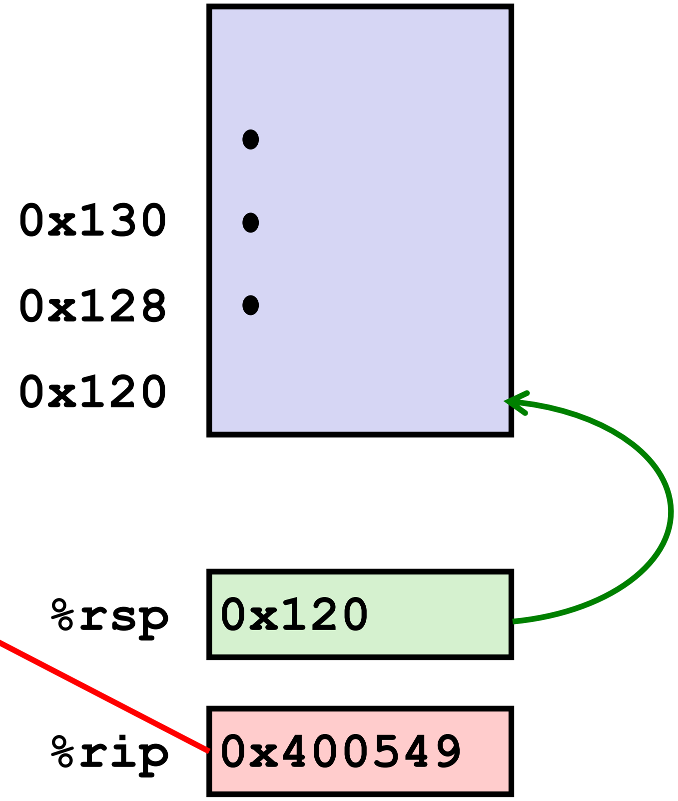
	•
0x130	•
0x128	•
0x120	
0x118	0x400549
%rsp	0x118
%rip	0x400557

retq: Pop return addr from stack, jump to addr.

Control Flow Example #4

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

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



Today

- Wrap up data dependent control
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

Procedure Data Flow

Registers

- First 6 arguments

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

- Return value

`%rax`

Stack

• • •
Arg *n*
• • •
Arg 8
Arg 7

- Use registers or stack to “pass” data between functions
- But only allocate stack space when needed!

Data Flow Examples

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  • • •
```

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

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

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data in procedures
  - Register saving conventions
  - Illustration of Recursion

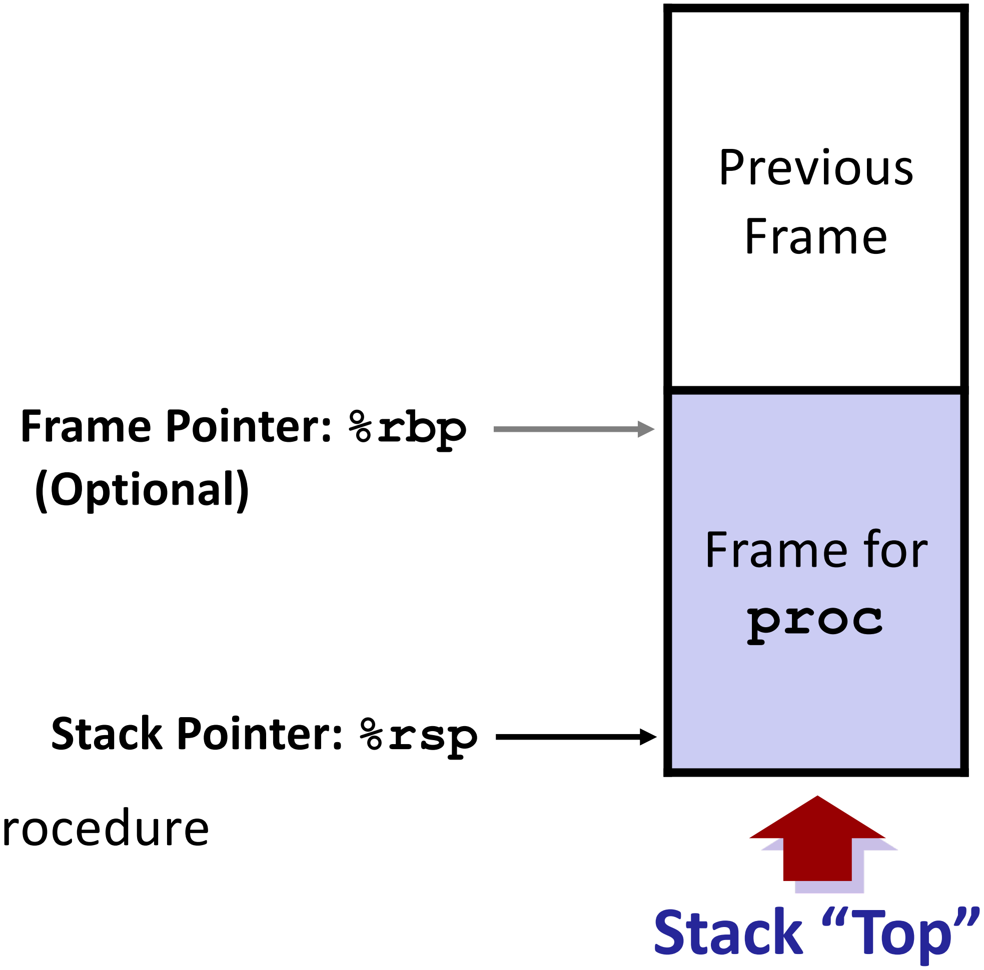
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, Python
  - Code must be "reentrant"
    - Allows for multiple simultaneous instantiations of single procedure
  - Thus we need some place to store state (data) of each *instantiation*
    - Arguments, local variables, return address pointer
- Stack discipline
  - State for given procedure is only needed for limited time
    - From when called to when return occurs
  - **Callee** (function being called) returns before **caller** (function calling) does
- Stack space allocated in frames
  - Stores state for single procedure instantiation

Stack Frames

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

- Management
  - Space allocated when entering procedure
    - “Set-up” code
    - Includes push by **`call`** instruction
  - Space deallocated when return occurs
    - “Finish” code
    - Includes pop by **`ret`** instruction



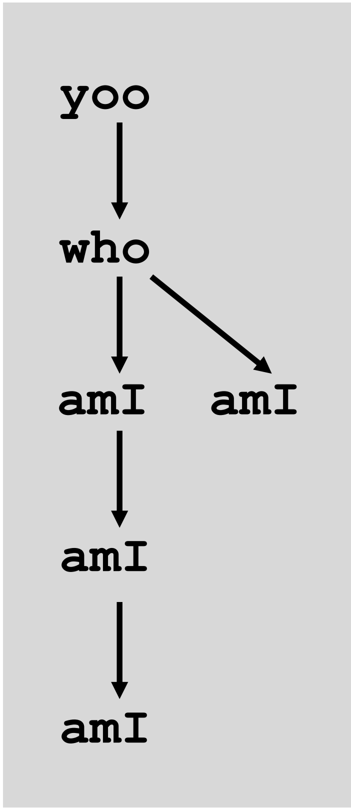
Call Chain Example

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

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

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

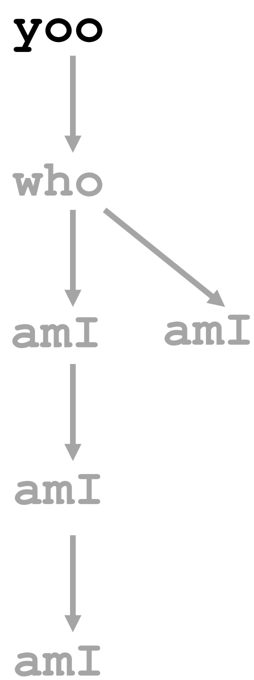
Example Call Chain

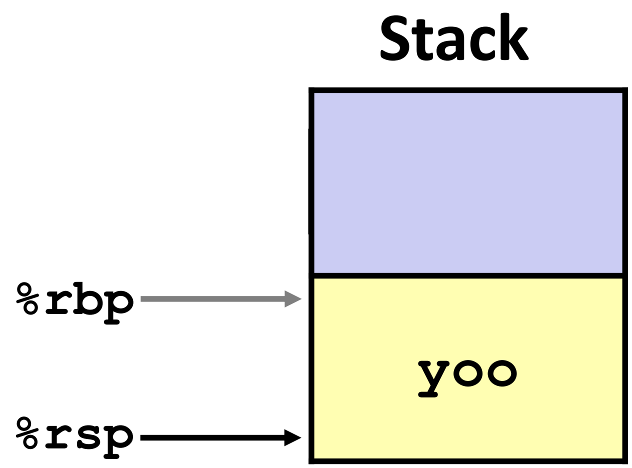


Note: Procedure `amI()` is recursive!

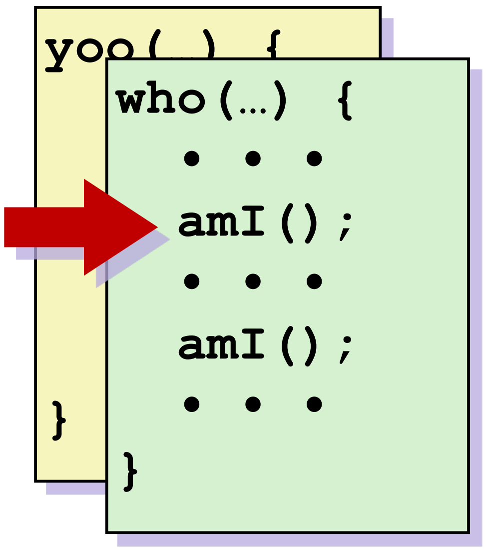
Example

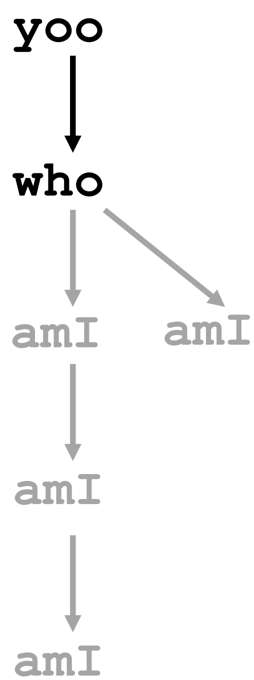
```
yoo(…) {
   •
   •
   who();
   •
   •
}
```

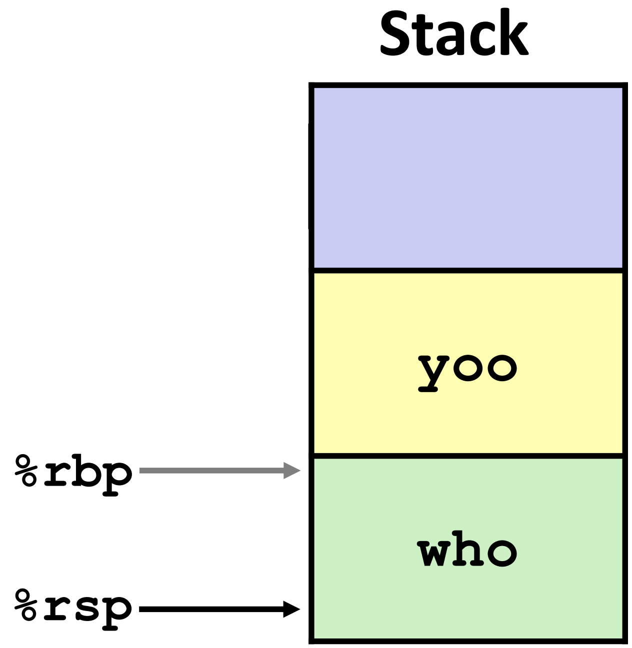




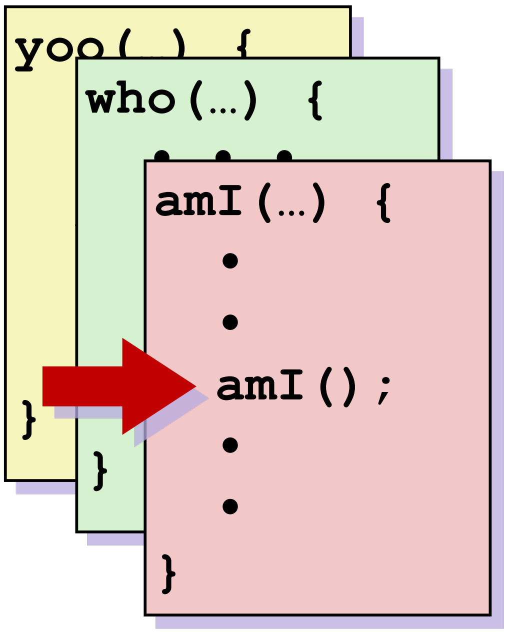
Example



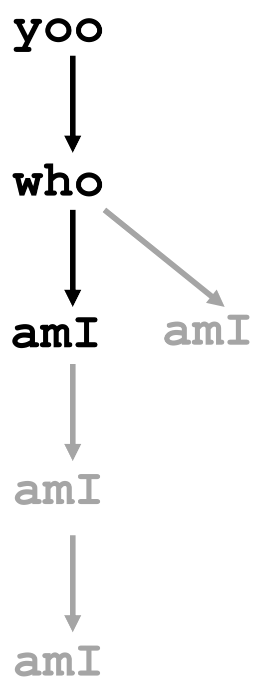




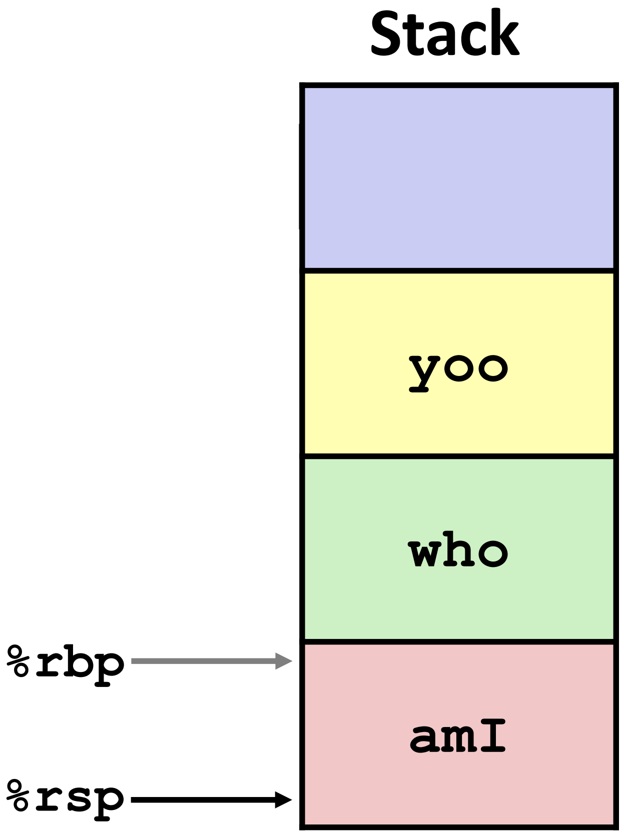
Example



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

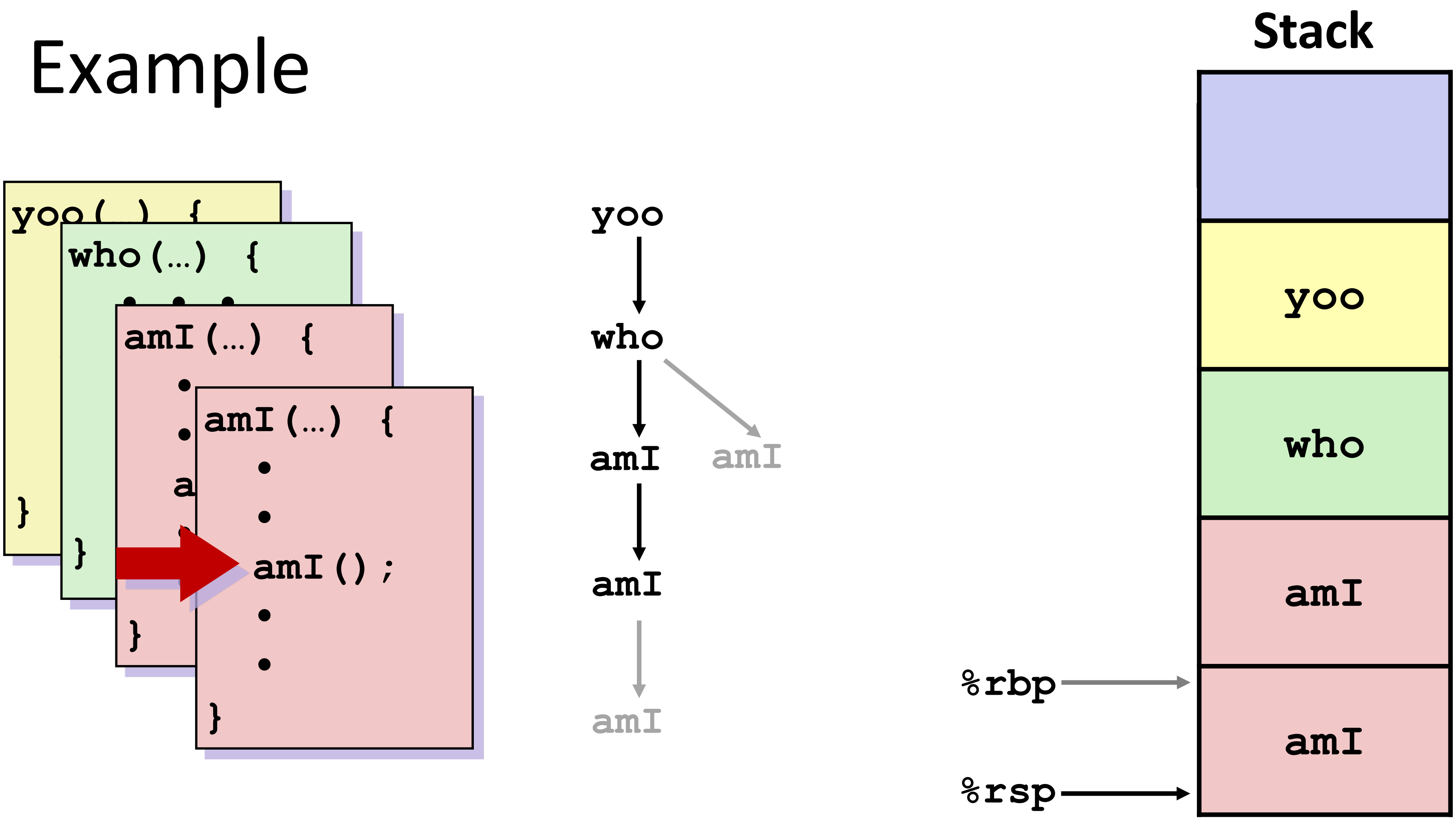


yoo
who
amI
amI
amI
amI



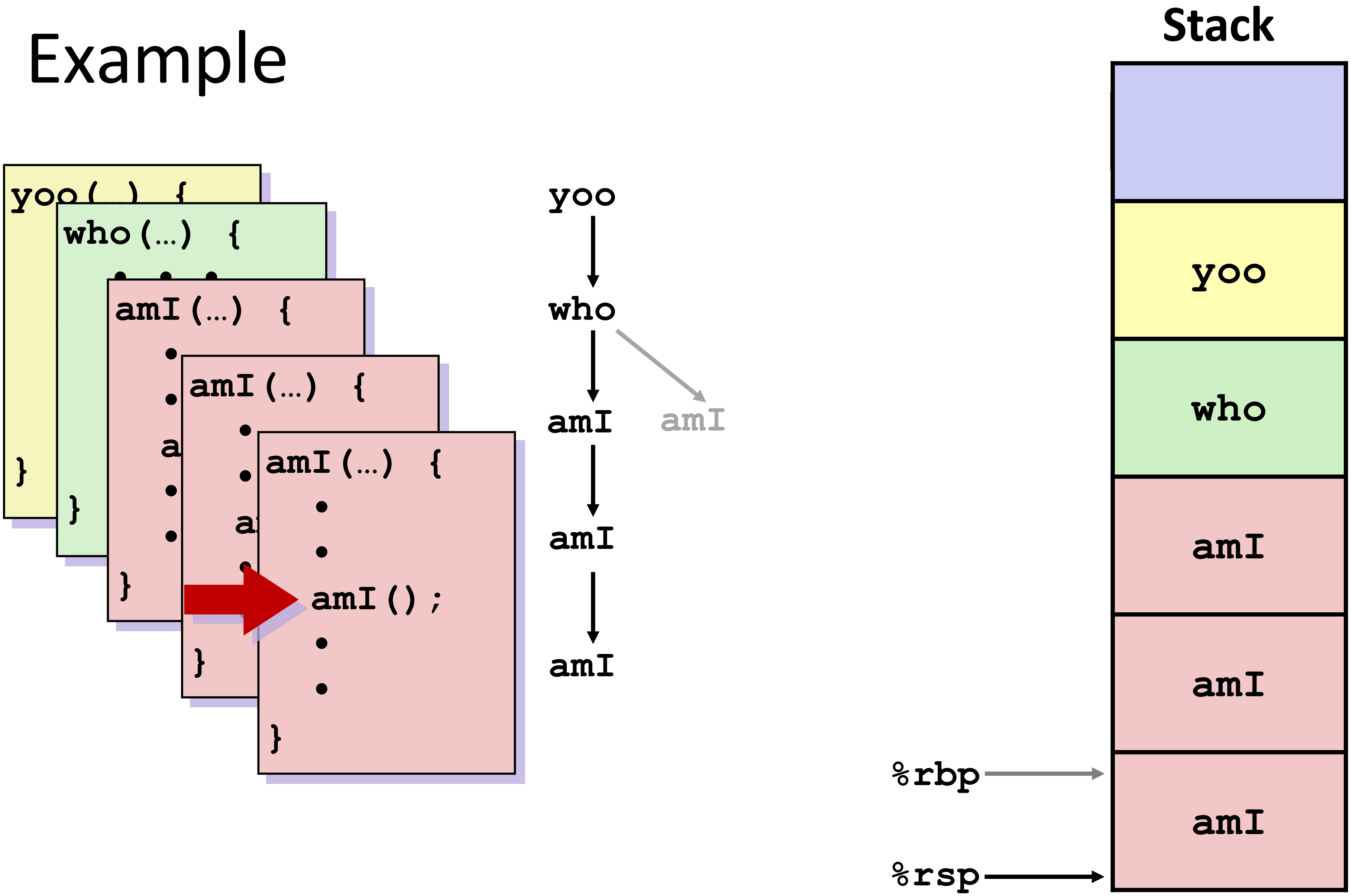
Stack
yoo
who
amI
%rbp
%rsp

Example



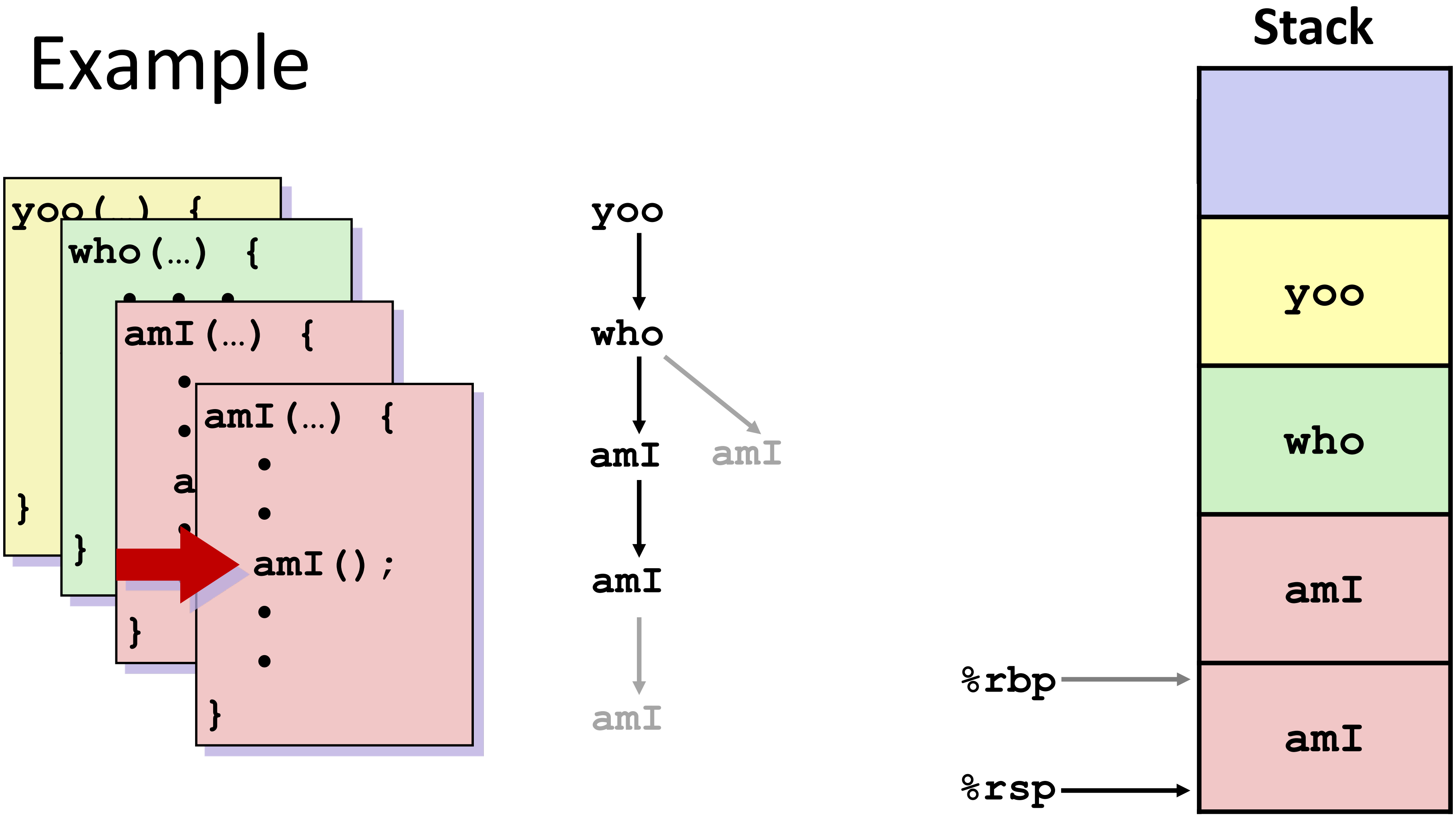
yoo(…) {
who(…) {
• • •
amI(…) {
•
•
a
•
}
}
}
amI(…) {
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

Example



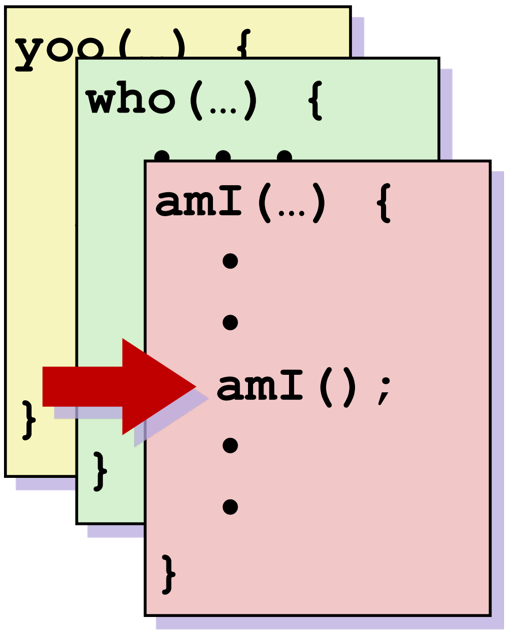
yoo(…) {
}
who(…) {
• • •
}
amI(…) {
•
•
}
amI(…) {
•
•
}
amI(…) {
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
amI
%rbp
%rsp

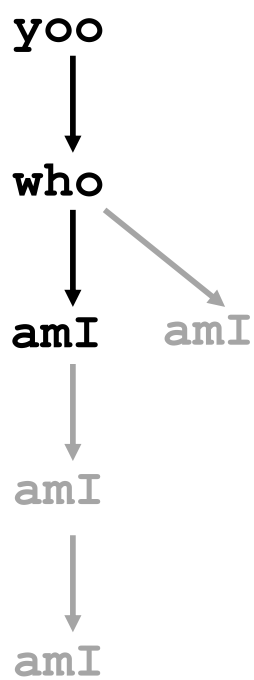
Example

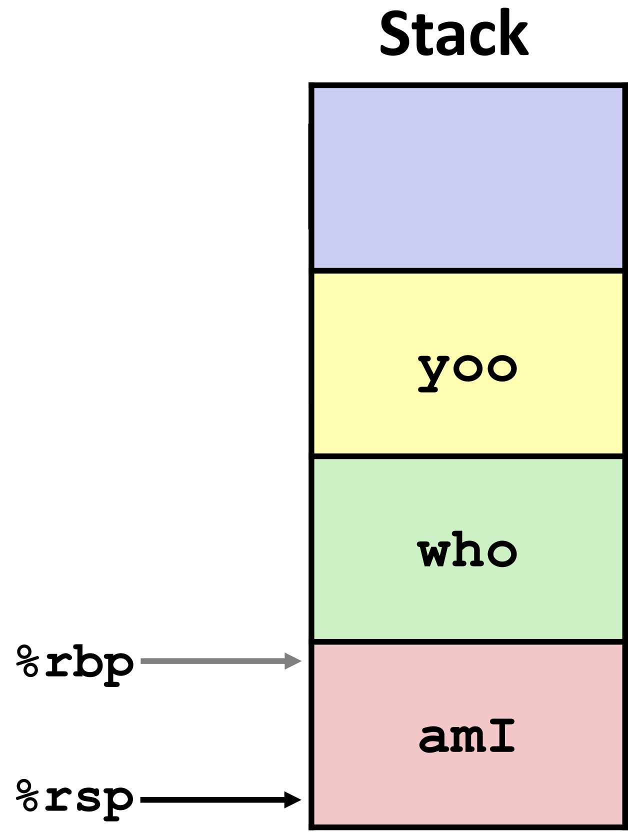


yoo(…) {
who(…) {
amI(…) {
amI(…) {
•
•
amI();
•
•
}
}
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

Example


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

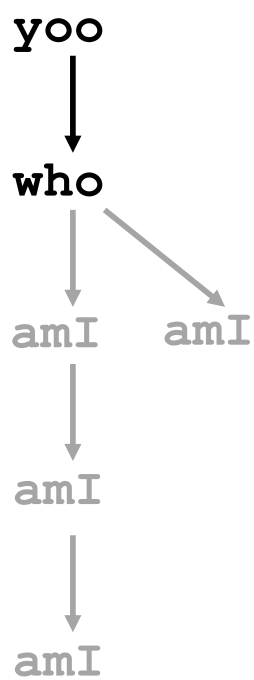

yoo
who
amI
amI
amI
amI

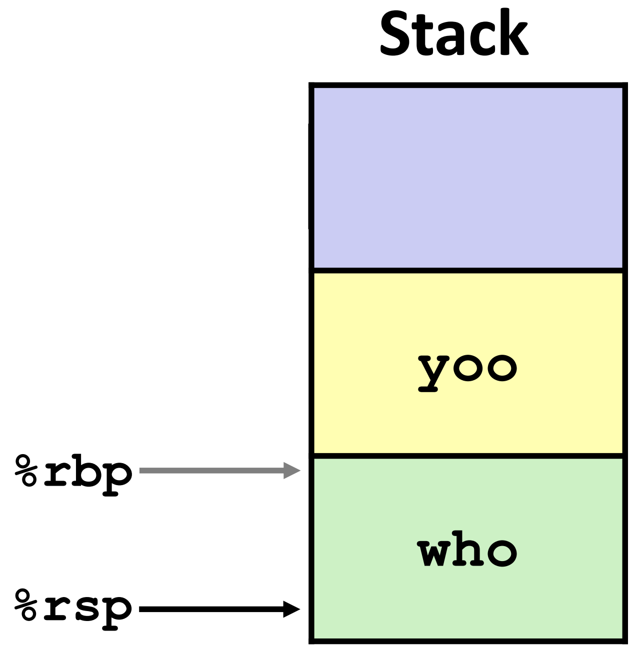

Stack
yoo
who
amI
%rbp
%rsp

Example

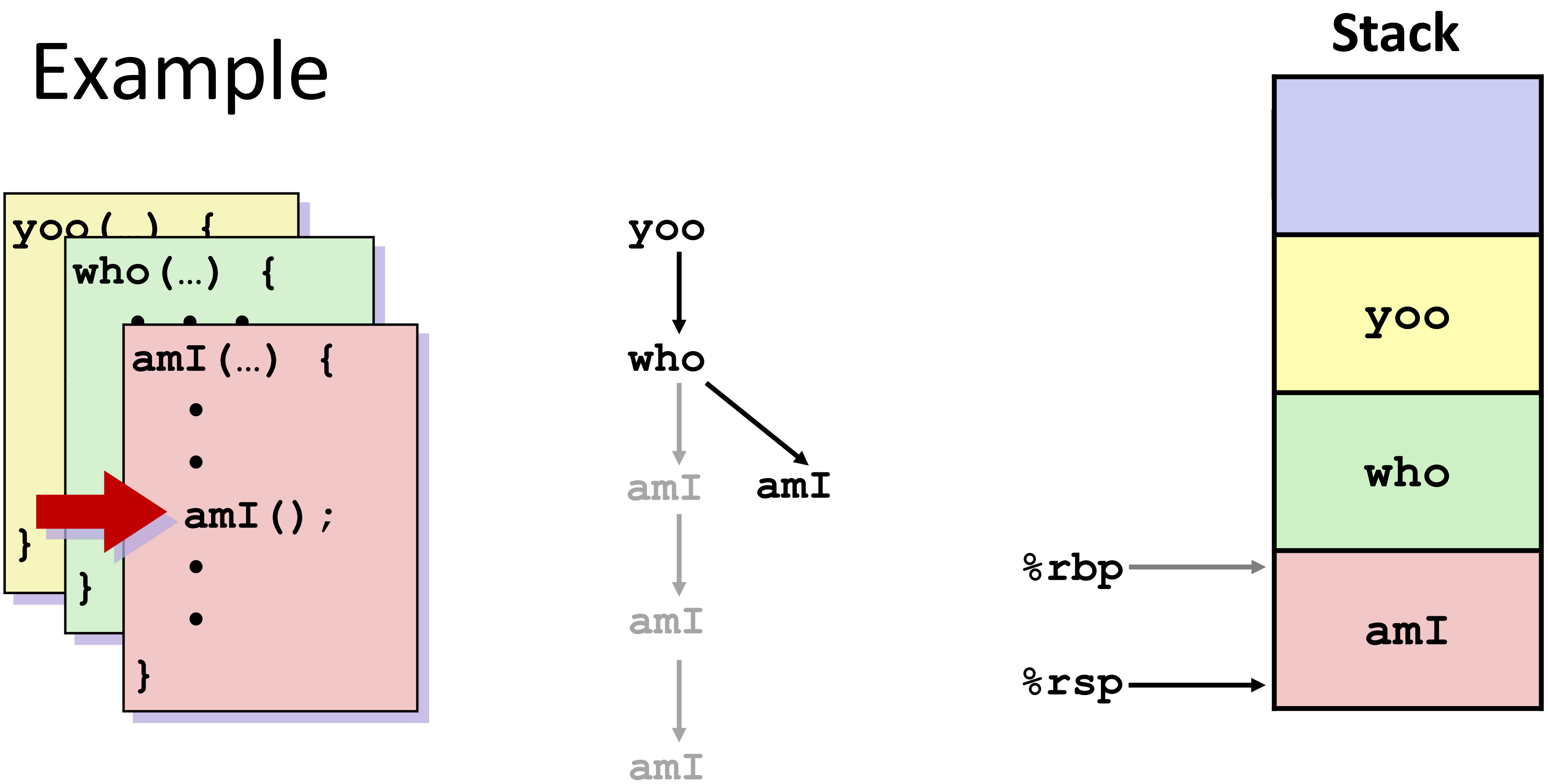
```
yoo(…) {

}
```

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



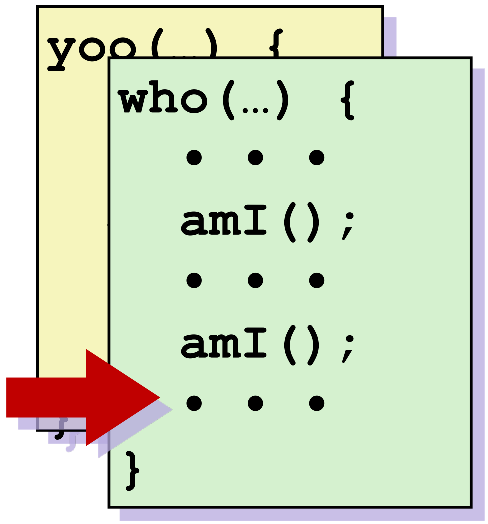


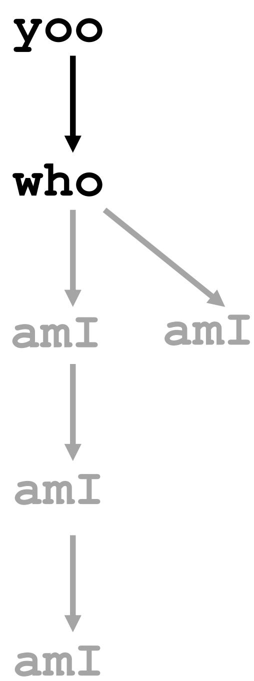
Example

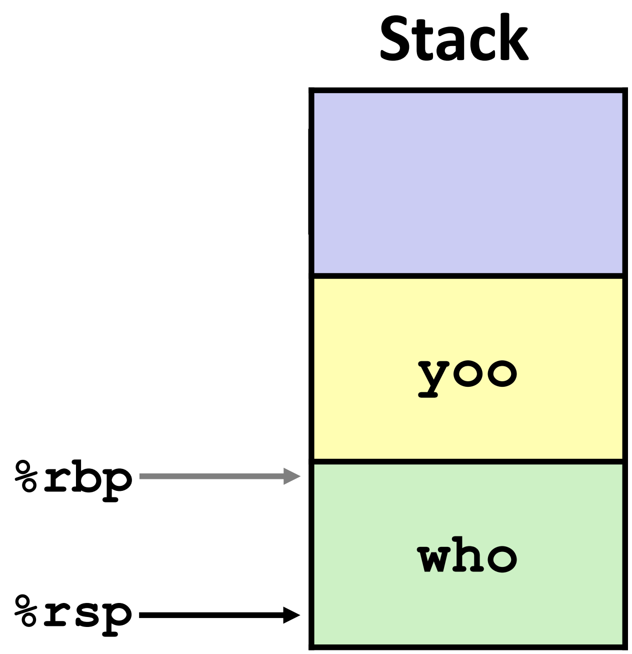


yoo(…) {
who(…) {
amI(…) {
amI();
}
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

Example







Example

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