

# **CSC 252: Computer Organization**

## **Spring 2019: Lecture 8**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

### **Action Items:**

- **Assignment 1 grades are out**
- **Assignment 2 is due soon!**

# Announcement

- Programming Assignment 2 is out
  - Due on **This Friday, Feb 15, 11:59 PM**
  - Tell us beforehand if you want to use slip days
  - Again, no slip days for trivia (in your best interests)
- Programming Assignment 1 grades are out

3	4	5	6	7	8	9
10	11	12	13	14	15	16
		Today			Due	

# Announcement

- A problem set for arithmetics: [http://  
www.cs.rochester.edu/courses/252/spring2019/  
handouts.html](http://www.cs.rochester.edu/courses/252/spring2019/handouts.html)
- Not to be turned in
- Form study groups
- Solution to be released soon

# Switch Statement Example

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Fall-through case

Multiple case labels

For missing cases,  
fall back to default

Converting to a cascade of if-else statements is simple, but cumbersome with too many cases.

# Implementing Switch Using Jump Table

## Switch Form

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

# Implementing Switch Using Jump Table

## Switch Form

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

## Jump Targets

Targ0: Code Block  
0

Targ1: Code Block  
1

Targ2: Code Block  
2

•  
•  
•

Targ $n-1$ : Code Block  
 $n-1$

# Implementing Switch Using Jump Table

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

# Implementing Switch Using Jump Table

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

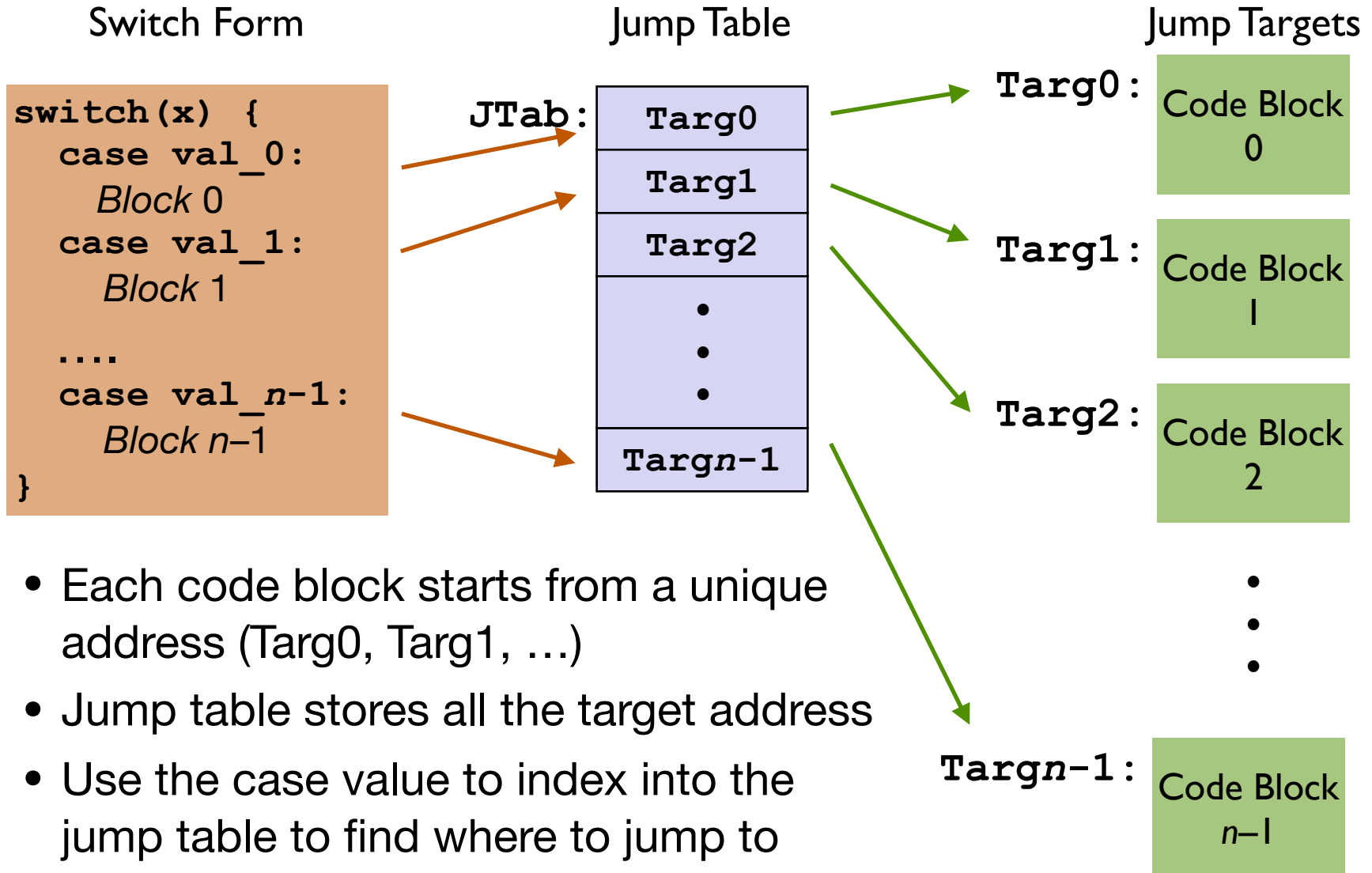
Targ2: Code Block 2

•  
•  
•

Targn-1: Code Block n-1



# Implementing Switch Using Jump Table



# Jump Table and Jump Targets

## Jump Table

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

## Jump Targets

```
.L3:                                # Case 1
    movq    %rsi, %rax
    imulq    %rdx, %rax
    jmp     .done

.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq    %rcx

.L9:                                # Case 3
    addq     %rcx, %rax
    jmp     .done

.L7:                                # Case 5,6
    subq     %rdx, %rax
    jmp     .done

.L8:                                # Default
    movl     $2, %eax
    jmp     .done
```

# Implementing Switch Using Jump Table

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

# Implementing Switch Using Jump Table

Switch Form

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

Jump Table

.L4:	.L8
	.L3
	.L5
	•
	•
	•
	.L7

Jump Targets

.L8: Code Block 0

.L3: Code Block 1

.L5: Code Block 2

•

•

•

.L7: Code Block n-1

# Implementing Switch Using Jump Table

Switch Form

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

Jump Table

.L4:	.L8
	.L3
	.L5
	•
	•
	•
	.L7

Jump Targets

.L8: Code Block 0

.L3: Code Block 1

.L5: Code Block 2

•

•

•

.L7: Code Block n-1

- The only thing left...
  - How do we jump to different locations in the jump table depending on the case value?

# Indirect Jump Instruction

```
.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
```

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

```
# assume x in %rdi
movq    .L4(,%rdi,8), %rax
jmp     *%rax
```



# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

```
# assume x in %rdi
movq    .L4(,%rdi,8), %rax
jmp     *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to (PC = `%rax`)

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to ( $PC = \%rax$ )
- Direct Jump (**jmp .L4**), directly specifies the jump address

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - `%rax` specifies the address to jump to ( $PC = \%rax$ )
- Direct Jump (**jmp .L4**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

# Indirect Jump Instruction

Address we want =  $.L4 + 8 * x$

```
.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
```

```
# assume x in %rdi
movq .L4(,%rdi,8), %rax
jmp  *%rax
```

- Indirect Jump: **jmp \*%rax**
  - %rax specifies the address to jump to (PC = %rax)
- Direct Jump (**jmp .L4**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

```
jmp  *.L4(,%rdi,8)
```

# Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

# Example of a Go Program Structure

```
main()
{
    /* place pieces on board */
    SetupBoard();

    /* choose black/white */
    DetermineSides();

    /* Play game */
    do {
        WhitesTurn();
        BlacksTurn();
    } while (NoOutcomeYet());
}
```

# Example of a Go Program Structure

```
main()
{
    /* place pieces on board */
    SetupBoard();

    /* choose black/white */
    DetermineSides();

    /* Play game */
    do {
        WhiteTurn();
        BlackTurn();
    } while (NoOutcomeYet());
}
```

Structure of program  
is evident, even without  
knowing implementation.

# Function

- Smaller, simpler, subcomponent of program that:
  - hides low-level details
  - gives high-level structure to program, easier to understand overall program flow
  - enables separable, independent development
- C functions
  - zero or multiple arguments passed in
  - single result returned (optional)
  - return value is always a particular type
- In other languages, called procedures, subroutines, ...



# Functions Declaration in C

Declaration (also called prototype)

- States return type, name, types of arguments

```
int Factorial(int) ;
```



type of  
return value



name of  
function



types of all  
arguments

# Function Definition

- Must match function declaration
- Implement the functionality of the function

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```



gives control back to  
calling function and  
returns value

# Mechanisms in Procedures

```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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

# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point

```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```


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

# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point

```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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

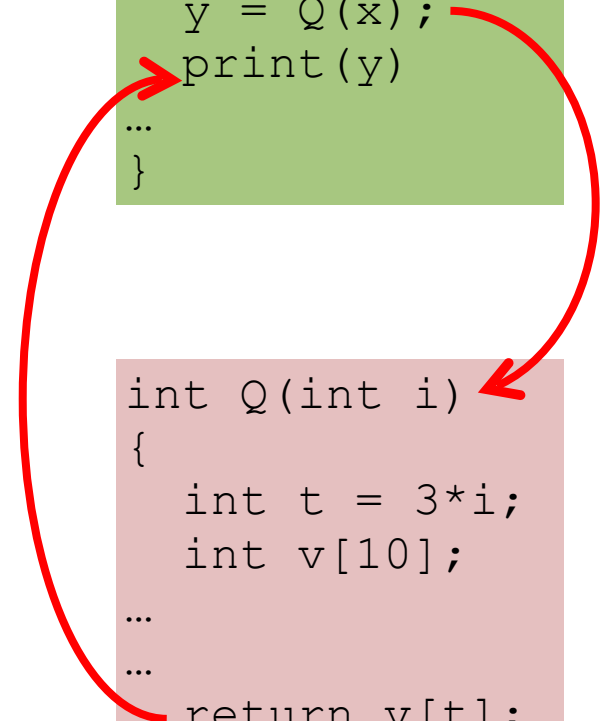


# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point

```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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

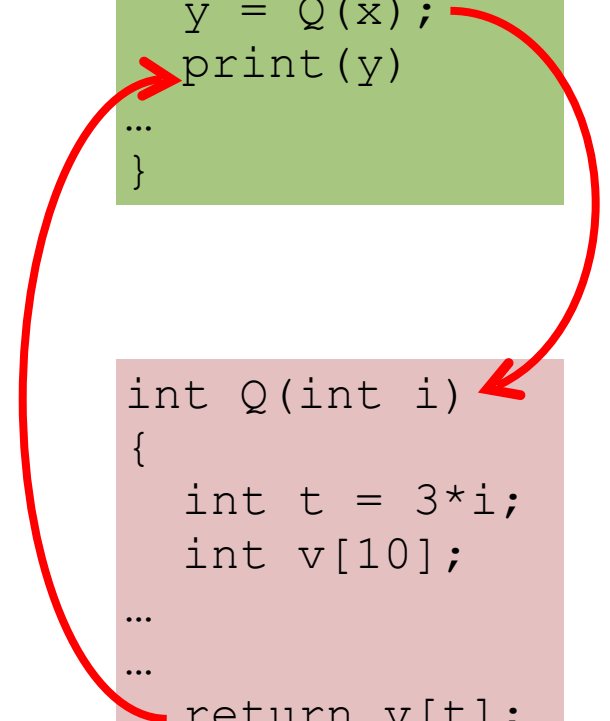


# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value

```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```


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



# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value

```
P (...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```



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

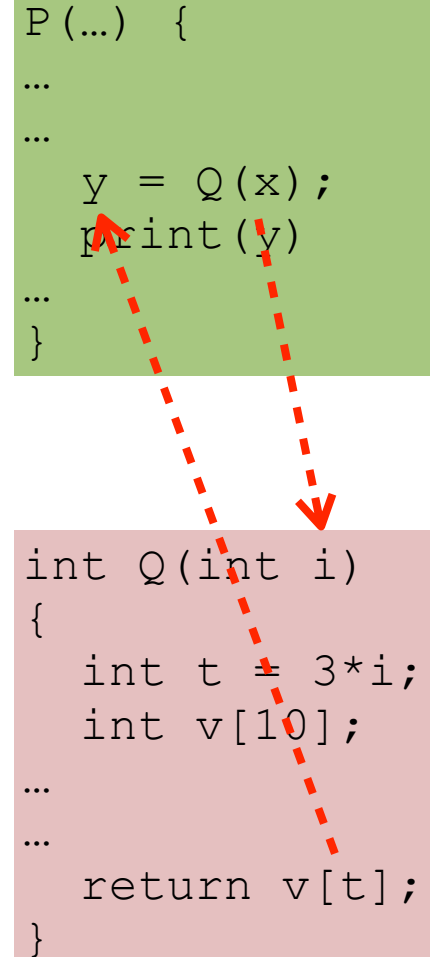


# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value

```
P (...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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



The diagram illustrates the control flow between two procedures. A green box at the top contains the code for procedure P, which calls Q. A red dashed arrow points from the call to Q in P to the start of procedure Q's code in a pink box below. Another red dashed arrow points from the return statement in Q back to the point in P where the call to Q occurred, showing the return path.

# 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

```
P (...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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

# Mechanisms in Procedures

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

```
P (...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

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

# Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

# General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute that function, e.g., arguments and local variables

# General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute that function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory

# General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute that function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory

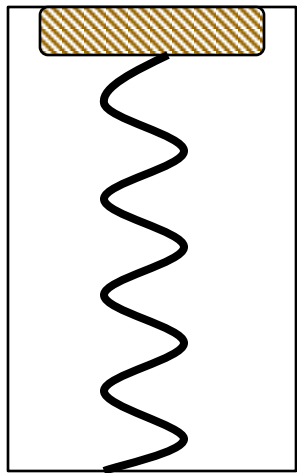
# General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute that function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory
- Frames are stored in memory in a *stack* fashion

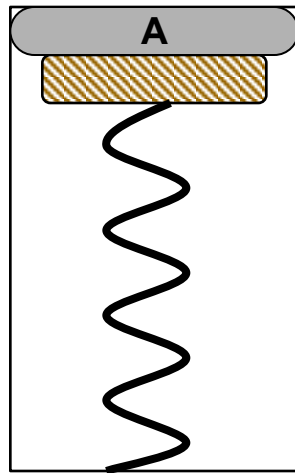


# A Physical Stack: A Coin Holder

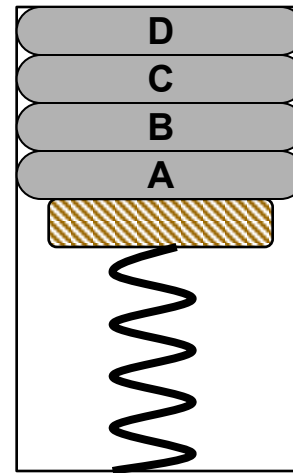
First quarter out is the last quarter in.



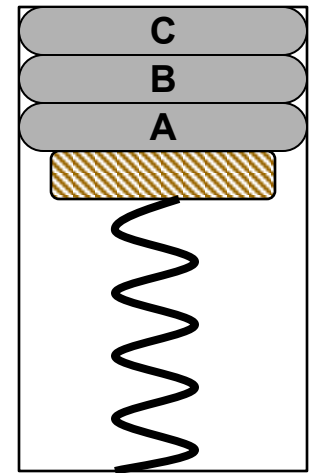
Initial State



After  
One Push



After Three  
More Pushes



After  
One Pop

- Stack is the right data structure for function call / return
  - If A calls B, then B returns before A

# Run-Time Stack During Function Call

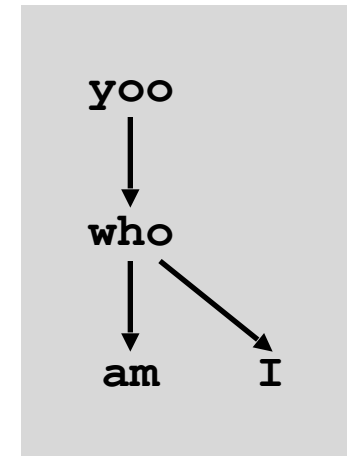
Example Call Chain

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


```
who (...)  
{  
  . . .  
  am ();  
  . . .  
  I ();  
  return;  
}
```

```
am (...)  
{  
  .  
  .  
  .  
  return;  
}
```

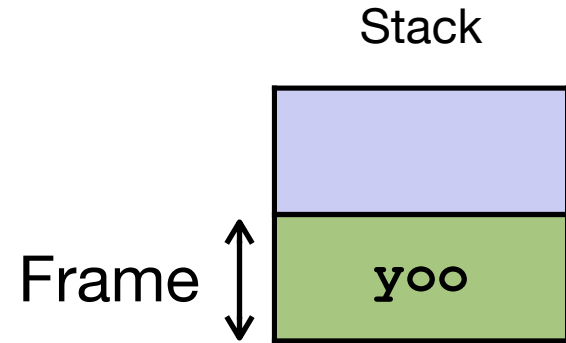
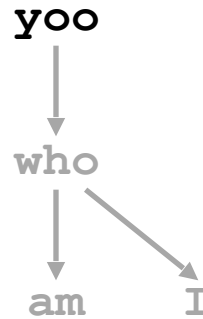
```
I (...)  
{  
  .  
  .  
  .  
  return;  
}
```



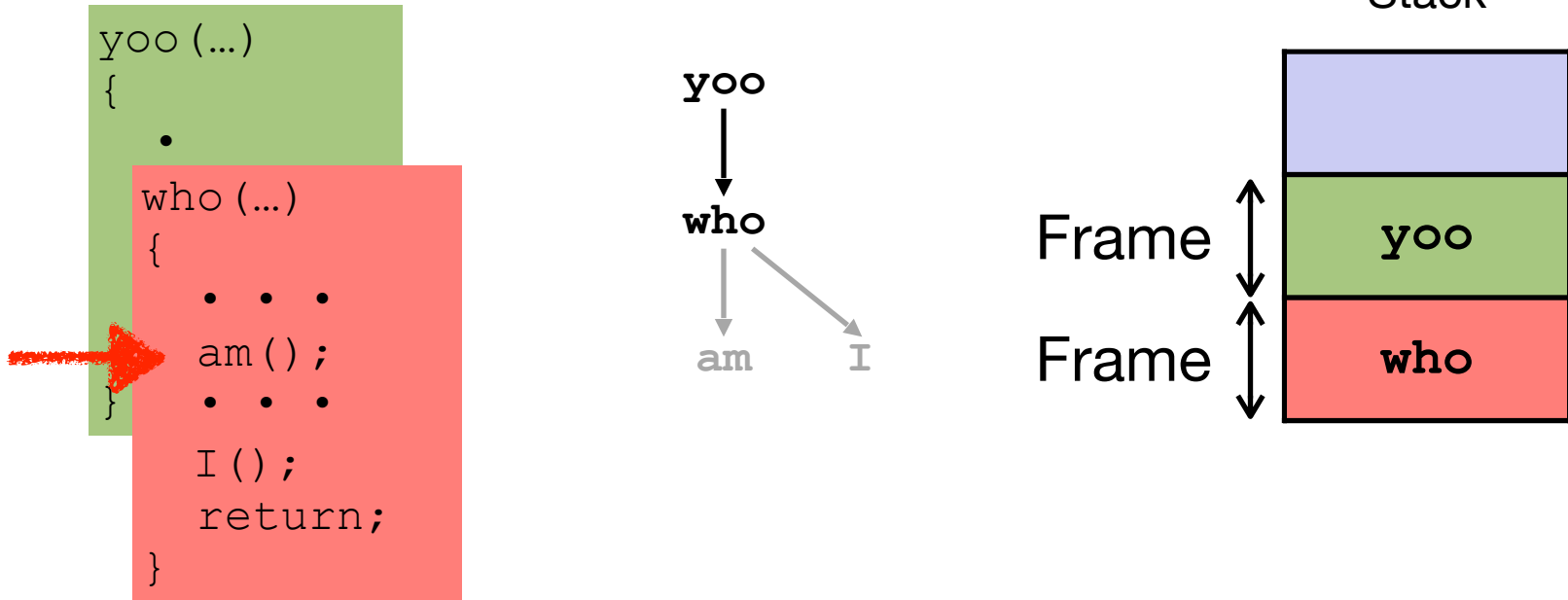
# Run-Time Stack During Function Call



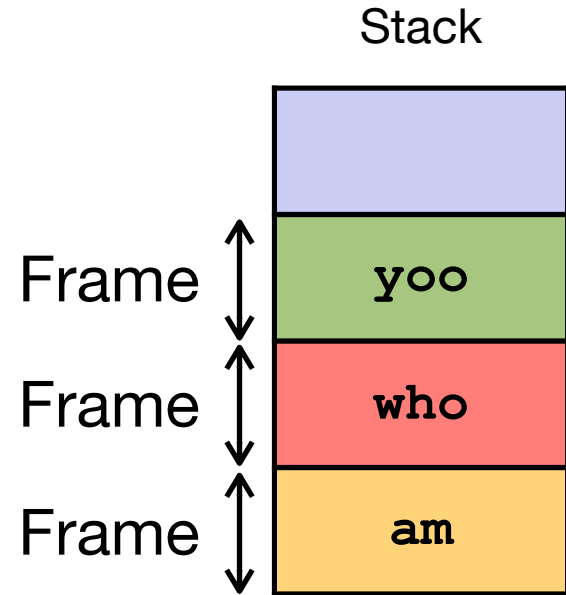
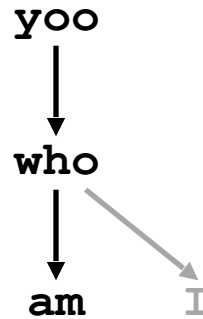
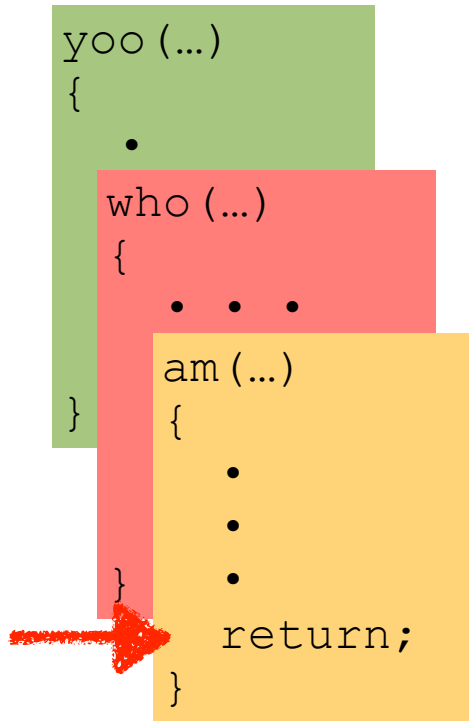
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  return  
}
```



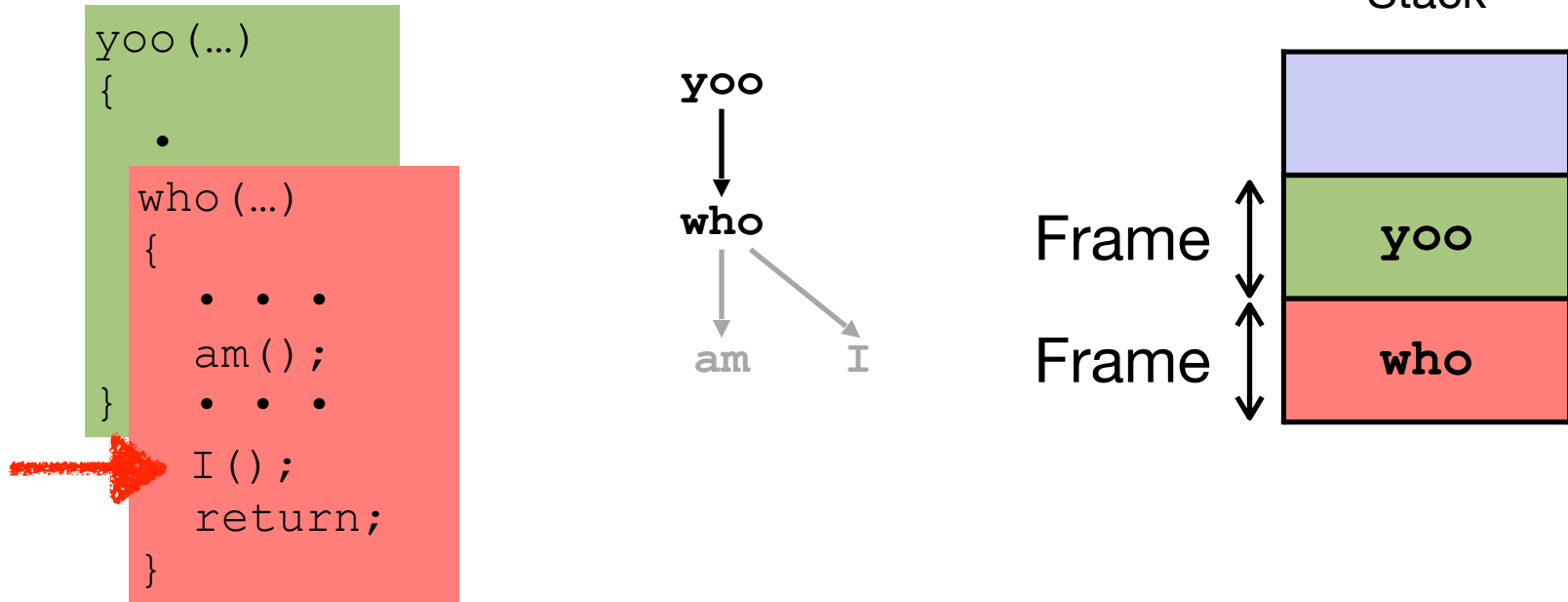
# Run-Time Stack During Function Call



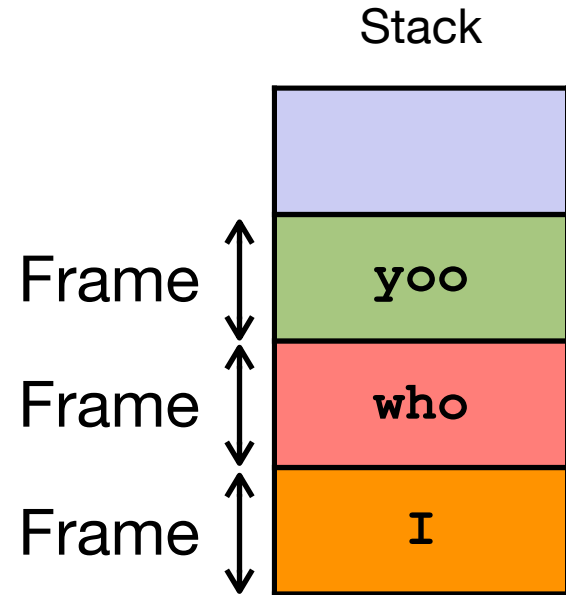
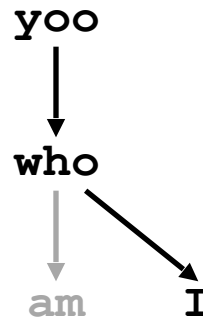
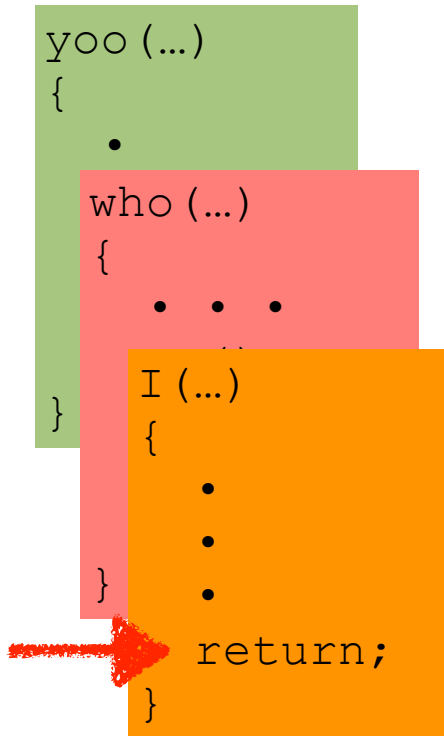
# Run-Time Stack During Function Call



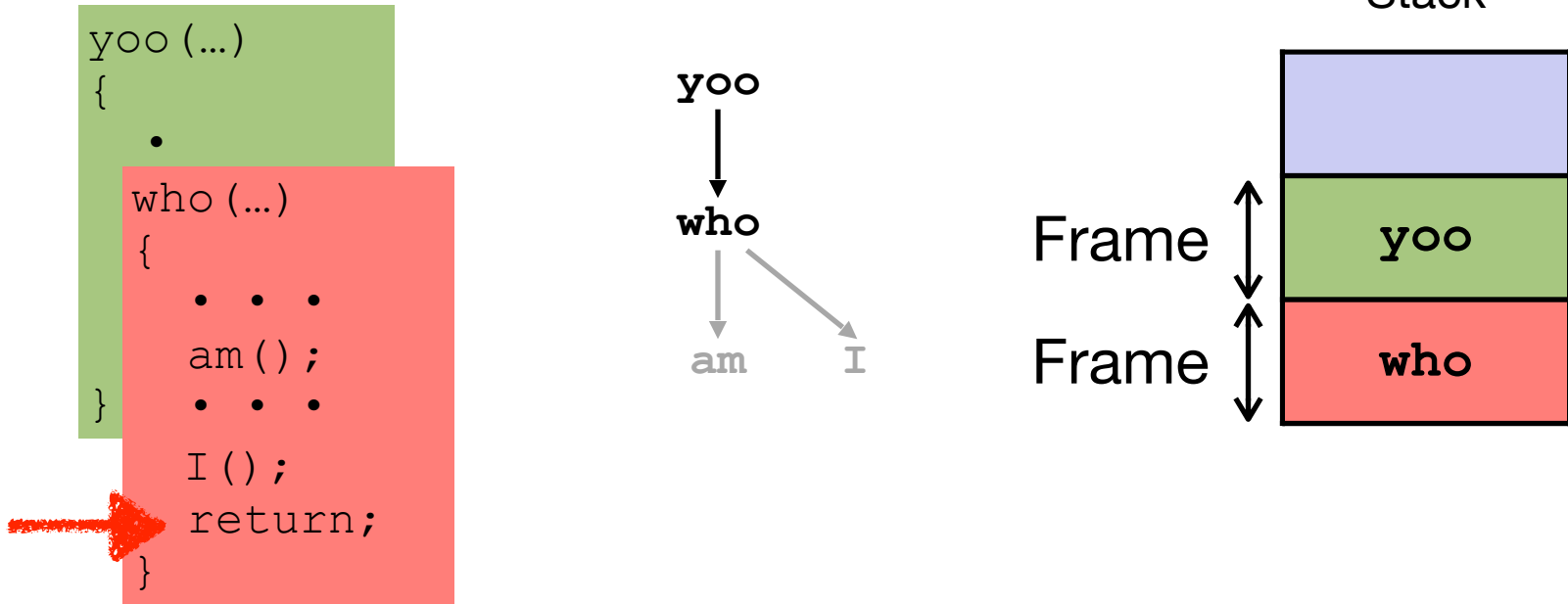
# Run-Time Stack During Function Call



# Run-Time Stack During Function Call



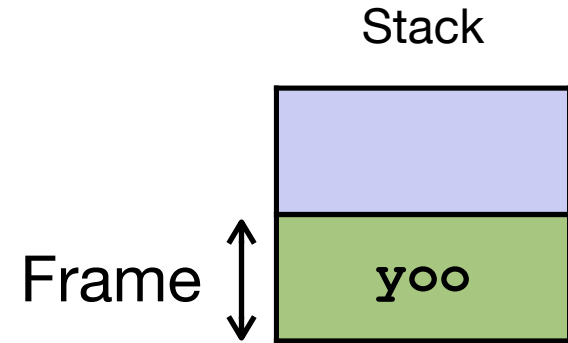
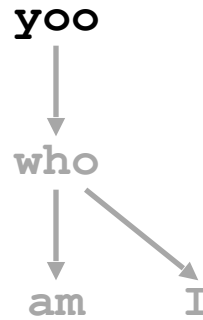

# Run-Time Stack During Function Call





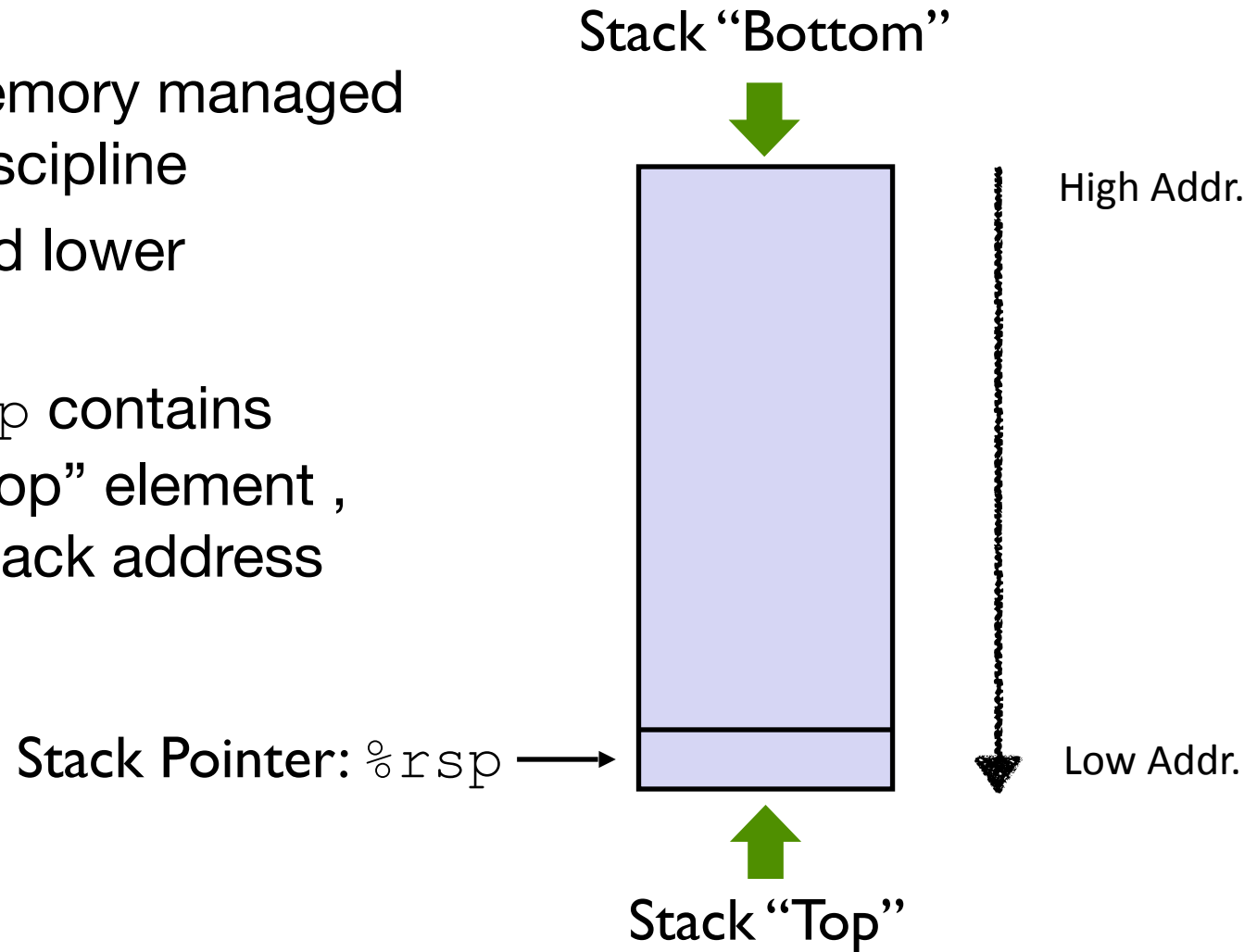
# Run-Time Stack During Function Call

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

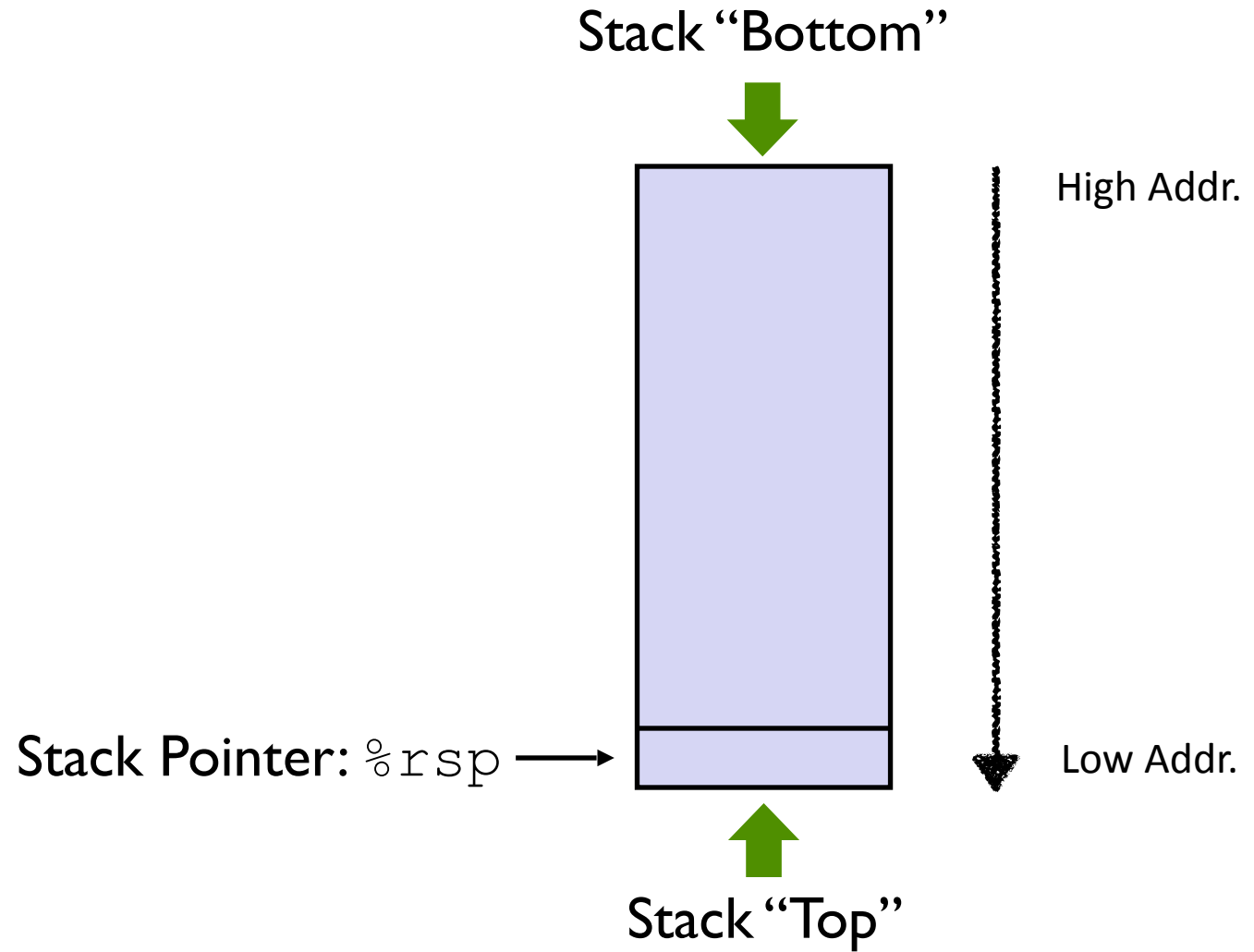


# Stack in X86-64

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains address of “top” element , i.e., lowest stack address



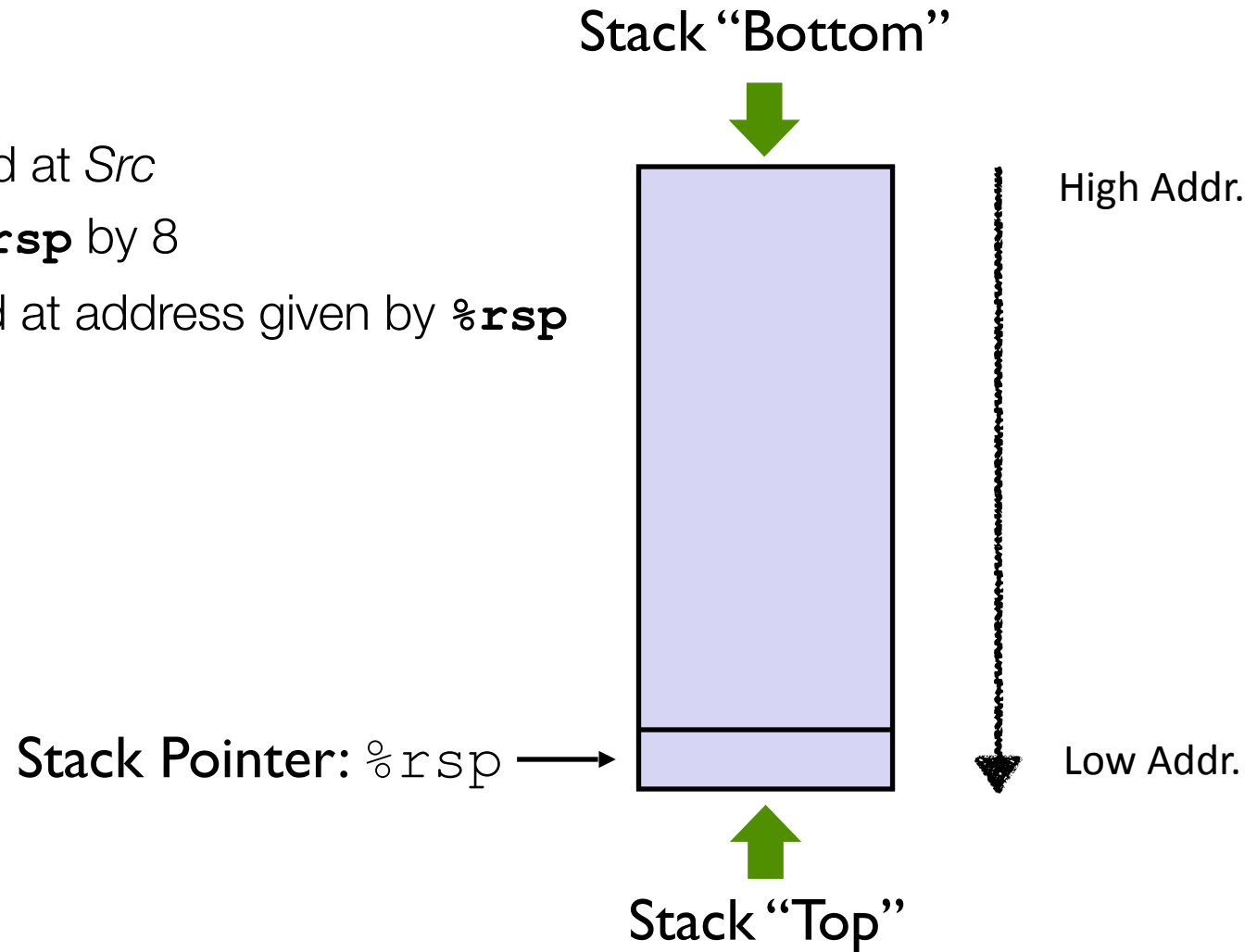
# x86-64 Stack: Push



# x86-64 Stack: Push

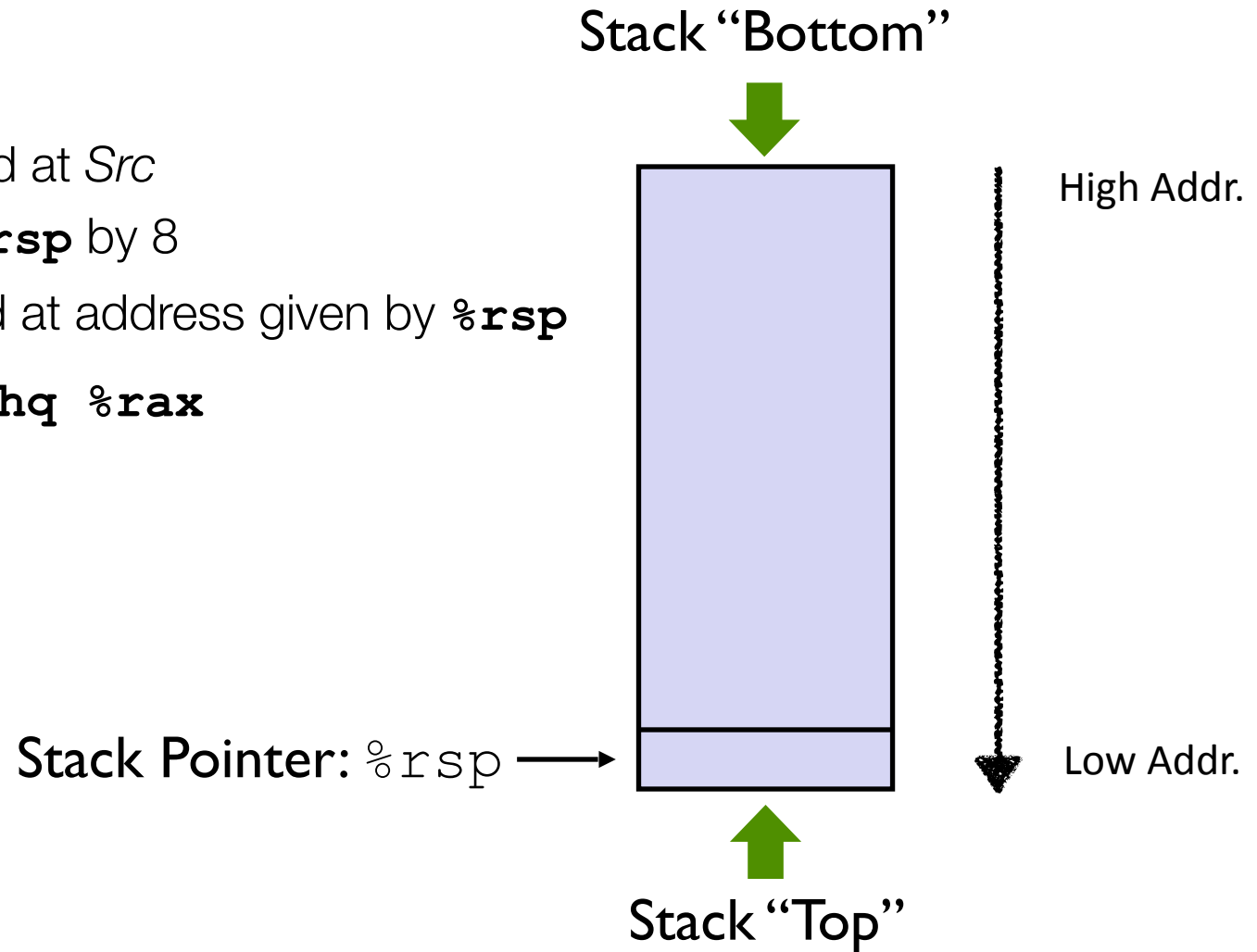
- **pushq Src**

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



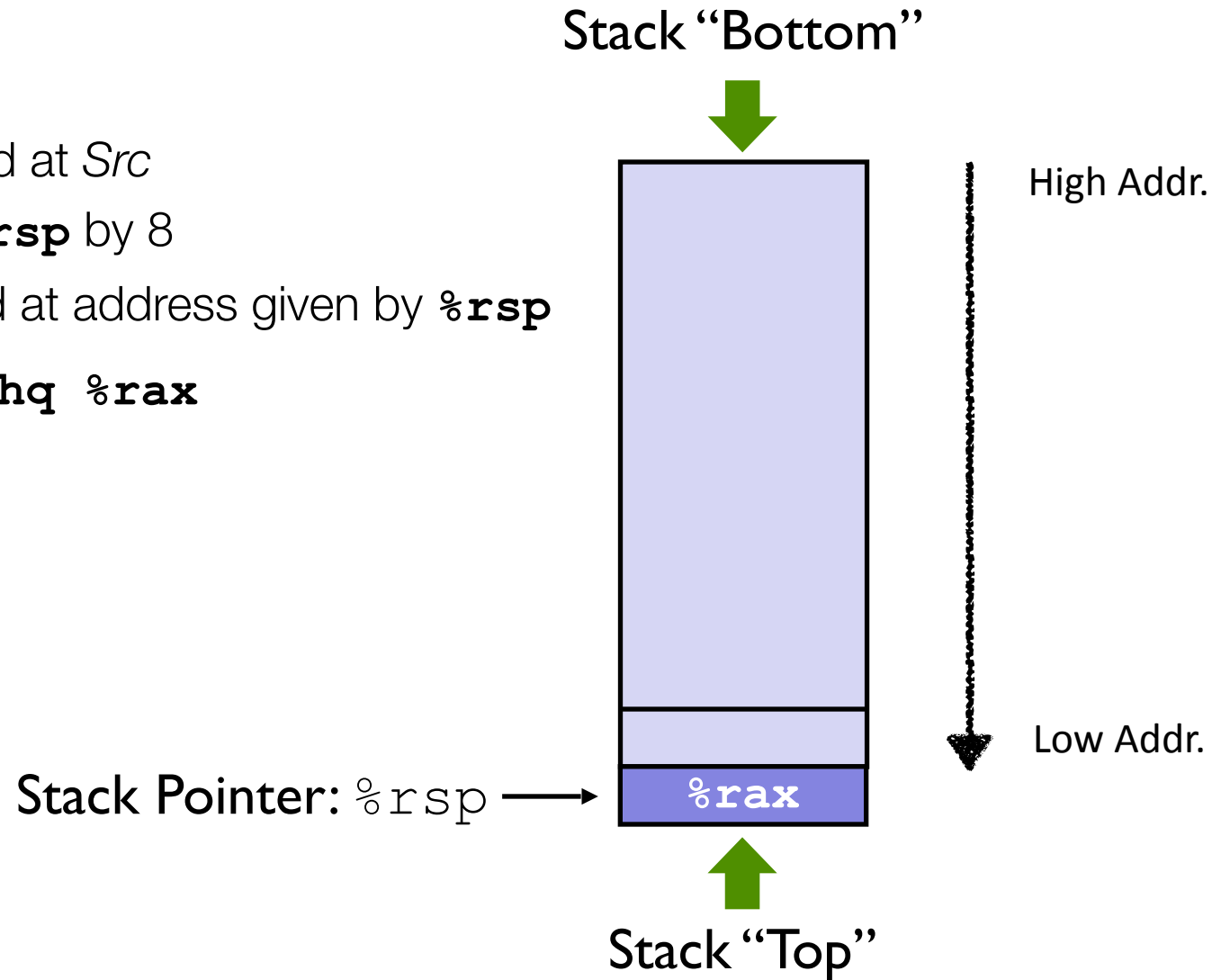
# x86-64 Stack: Push

- **pushq Src**
  - Fetch operand at *Src*
  - Decrement **%rsp** by 8
  - Write operand at address given by **%rsp**
- Example: **pushq %rax**



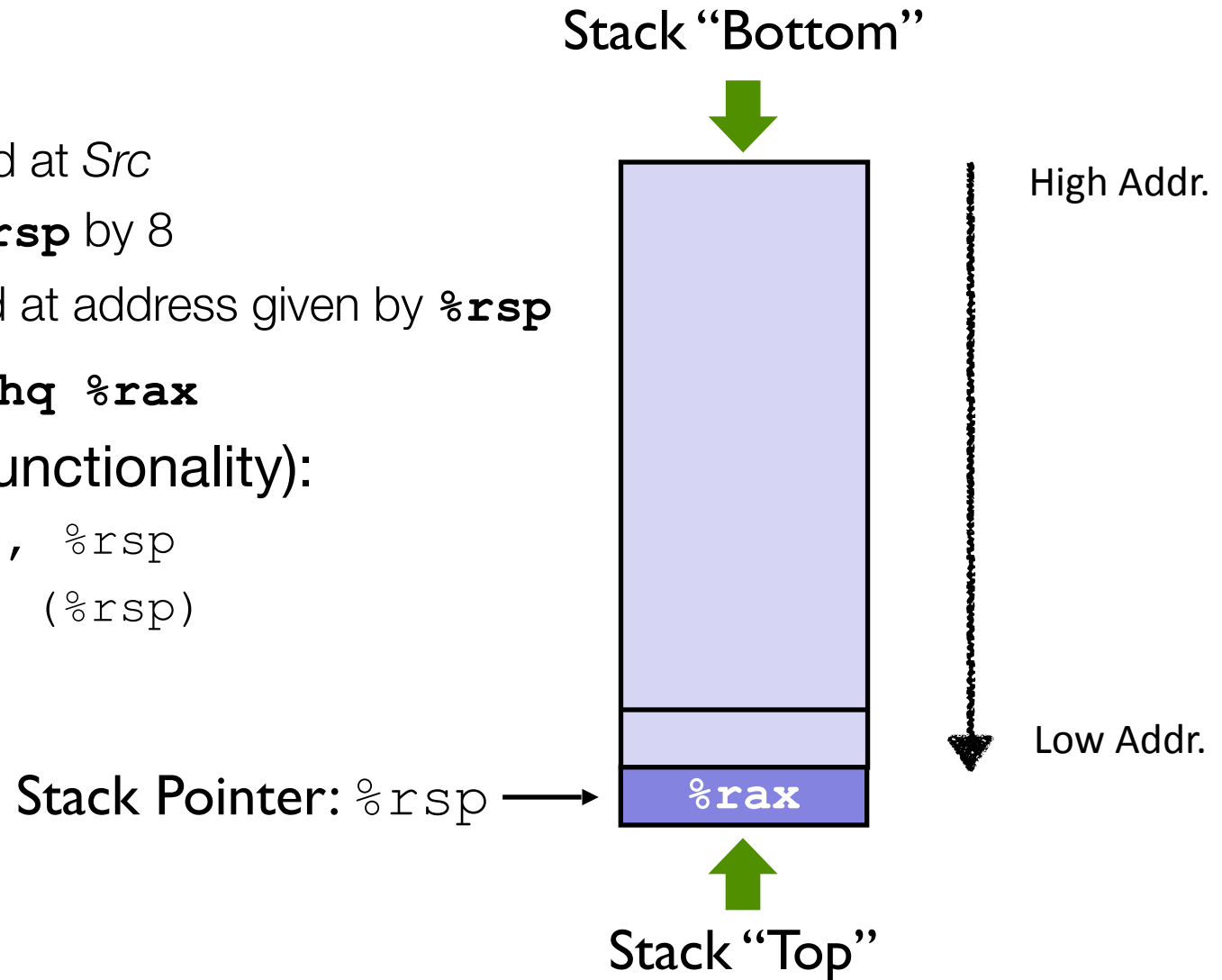
# x86-64 Stack: Push

- **pushq Src**
  - Fetch operand at *Src*
  - Decrement **%rsp** by 8
  - Write operand at address given by **%rsp**
- Example: **pushq %rax**



# x86-64 Stack: Push

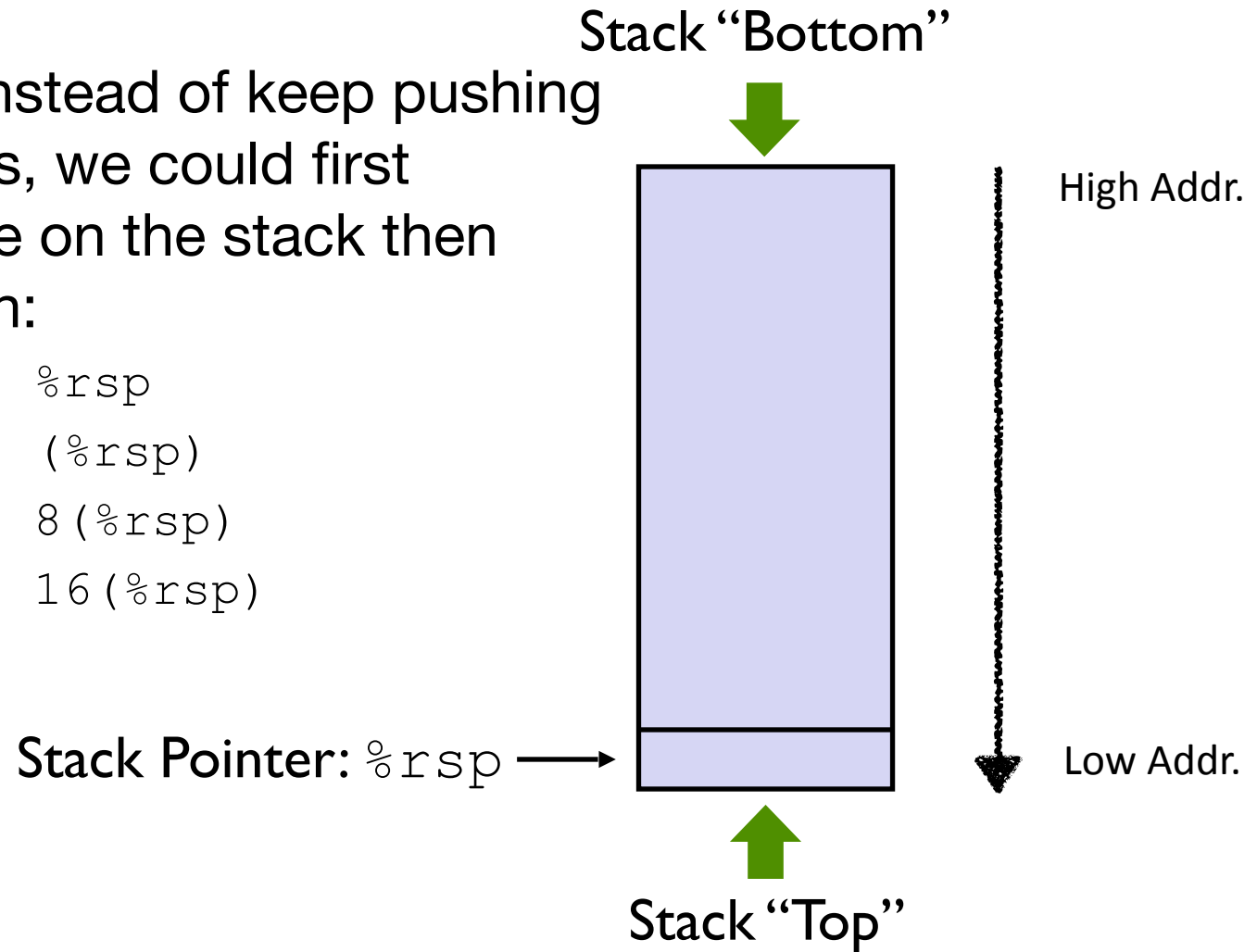
- **pushq Src**
  - Fetch operand at *Src*
  - Decrement **%rsp** by 8
  - Write operand at address given by **%rsp**
- Example: **pushq %rax**
- Same as (in functionality):
  - `subq $0x08, %rsp`
  - `movq %rax, (%rsp)`



# x86-64 Stack: Push

- Sometimes instead of keep pushing multiple items, we could first reserve space on the stack then move items in:

- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`

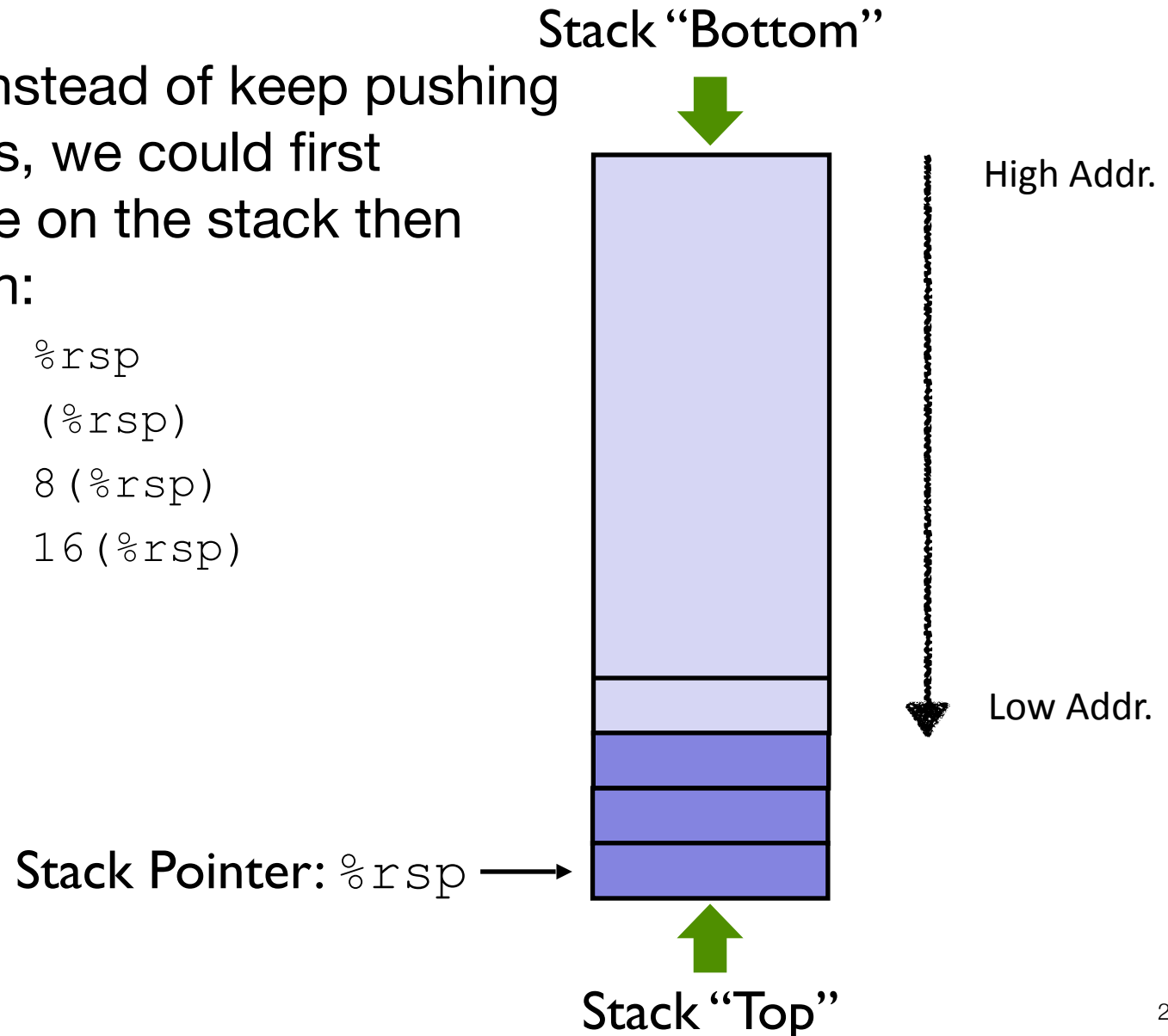




# x86-64 Stack: Push

- Sometimes instead of keep pushing multiple items, we could first reserve space on the stack then move items in:

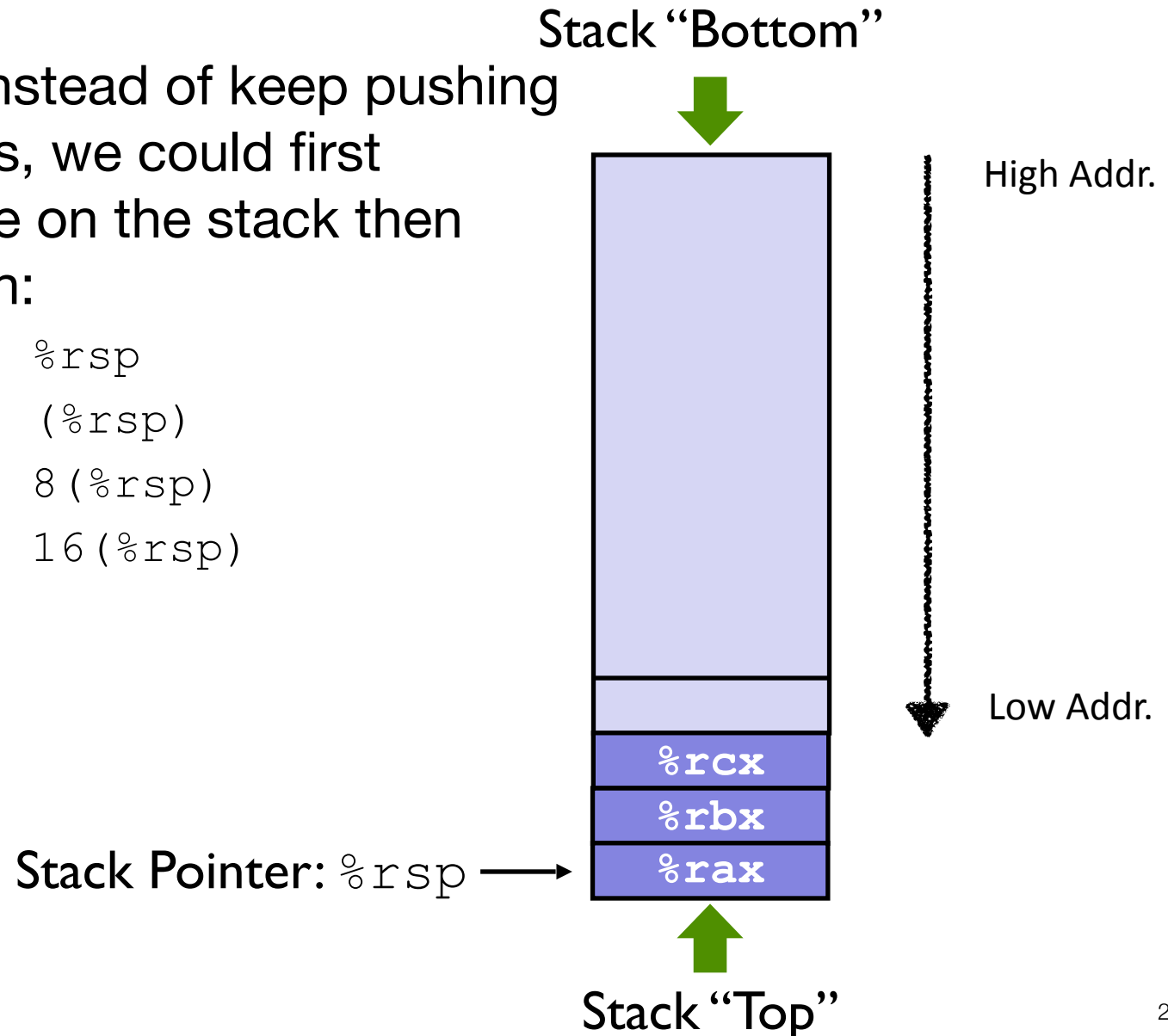
- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`



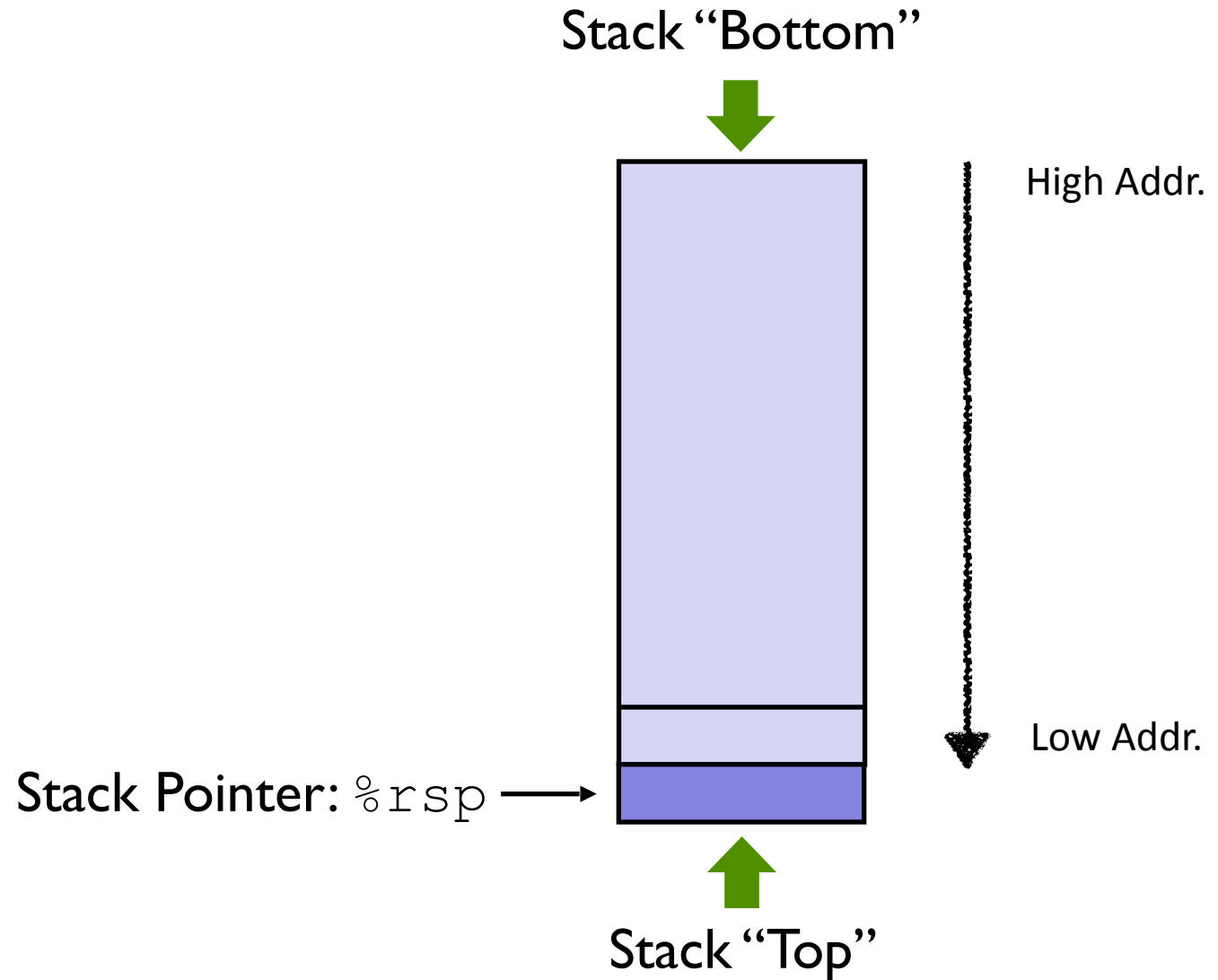
# x86-64 Stack: Push

- Sometimes instead of keep pushing multiple items, we could first reserve space on the stack then move items in:

- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`



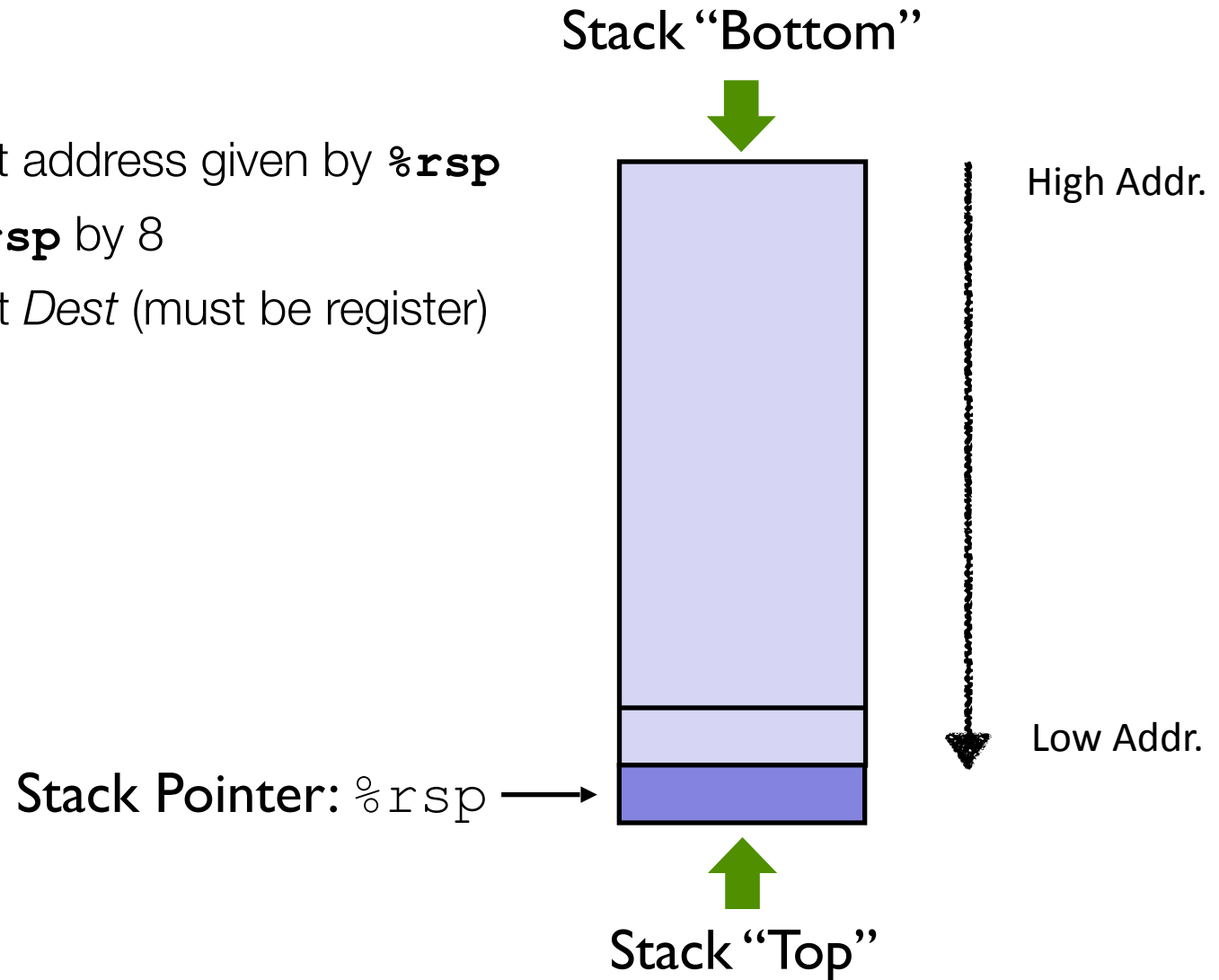
# x86-64 Stack: Pop



# x86-64 Stack: Pop

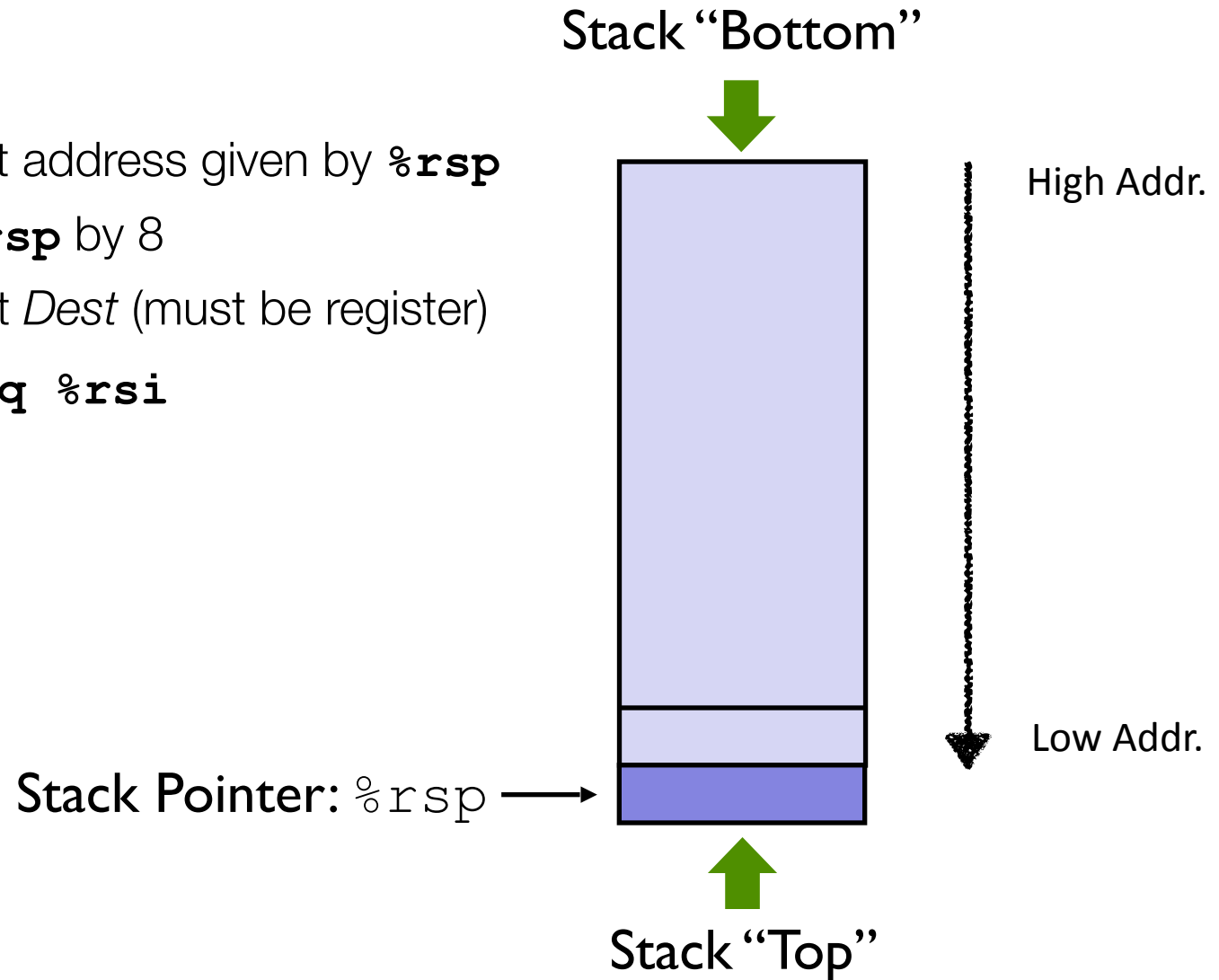
- **popq *Dest***

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



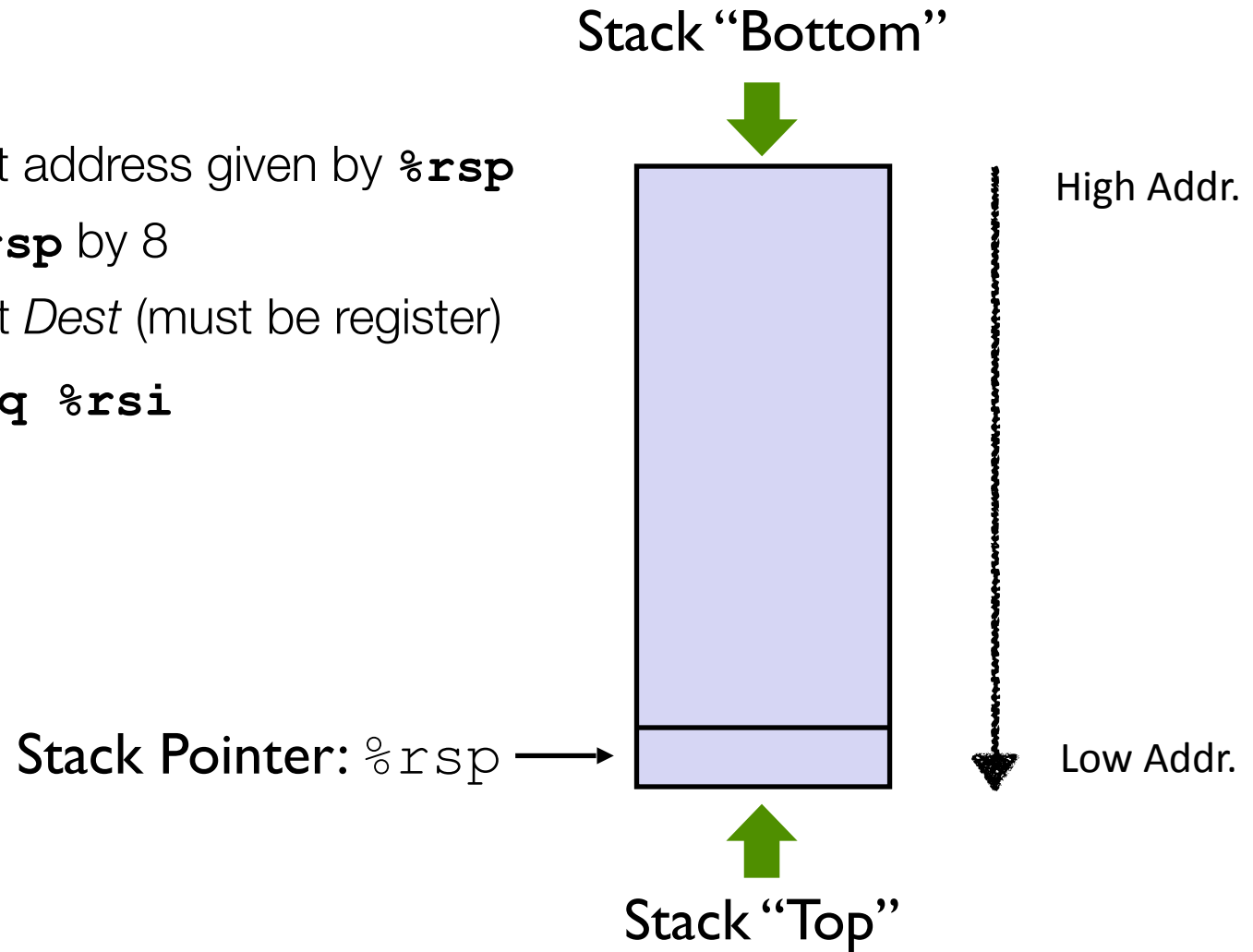
# x86-64 Stack: Pop

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



# x86-64 Stack: Pop

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



# x86-64 Stack: Pop

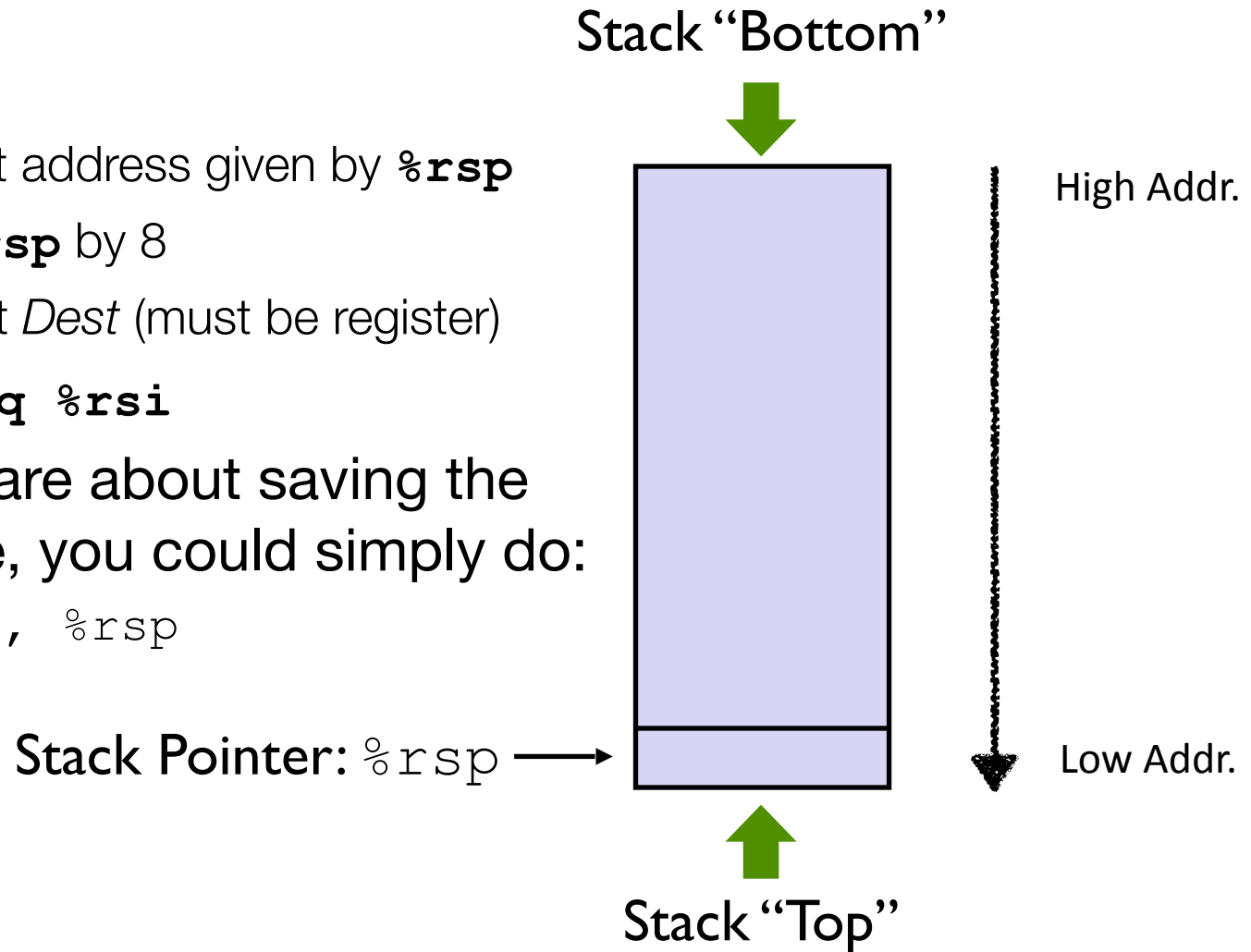
- **popq *Dest***

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

- Example: **popq %rsi**

- If you don't care about saving the popped value, you could simply do:

- **addq \$0x08, %rsp**



# Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- **Passing control**
- Passing data
- Managing local data



# Code Examples

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

...

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

# Code Examples

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

...

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

```
400540 <multstore>:
400540: push    %rbx
400541: mov     %rdx,%rbx
400544: callq   400550 <mult2>
400549: mov     %rax, (%rbx)
40054c: pop     %rbx
40054d: retq
```

...

```
400550 <mult2>:
400550: mov     %rdi,%rax
400553: imul    %rsi,%rax
400557: retq
```

# Code Examples

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

...

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

```
400540 <multstore>:
400540: push    %rbx
400541: mov     %rdx,%rbx
400544: callq   400550 <mult2>
400549: mov     %rax, (%rbx)
40054c: pop     %rbx
40054d: retq
```

...

```
400550 <mult2>:
400550: mov     %rdi,%rax
400553: imul    %rsi,%rax
400557: retq
```

`retq` returns to (by changing the PC) 400549.  
But how would `retq` know where to return?

# Non-Solution

- Replace `callq` with `jmp`
- assign a label to the instruction next to `callq` (e.g., `.L1`)
- replace `retq` with `jmpq .L1`

```
400540 <multstore>:
  400540:  push    %rbx
  400541:  mov     %rdx,%rbx
  400544:  callq   400550 <mult2>
  400549:  mov     %rax, (%rbx)
  40054c:  pop     %rbx
  40054d:  retq

...

400550 <mult2>:
  400550:  mov     %rdi,%rax
  400553:  imul    %rsi,%rax
  400557:  retq
```

# Non-Solution

- Replace `callq` with `jmp`
- assign a label to the instruction next to `callq` (e.g., `.L1`)
- replace `retq` with `jmpq .L1`

```
400540 <multstore>:
400540: push    %rbx
400541: mov     %rdx,%rbx
400544: jmp     400550 <mult2>
.L1 400549: mov     %rax, (%rbx)
40054c: pop     %rbx
40054d: retq

...

400550 <mult2>:
400550: mov     %rdi,%rax
400553: imul    %rsi,%rax
400557: jmp     .L1
```

# Non-Solution

- Replace `callq` with `jmp`
- assign a label to the instruction next to `callq` (e.g., `.L1`)
- replace `retq` with `jmpq .L1`
- Will this work?!
- How about when other functions call `mult2`?

```
400540 <multstore>:
400540: push    %rbx
400541: mov     %rdx,%rbx
400544: jmp     400550 <mult2>
.L1 400549: mov     %rax, (%rbx)
40054c: pop     %rbx
40054d: retq

...

400550 <mult2>:
400550: mov     %rdi,%rax
400553: imul    %rsi,%rax
400557: jmp     .L1
```

# Using Stack for Function Call and Return

- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call (400549 here)
- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address

```
400540 <multstore>:
400540: push    %rbx
400541: mov     %rdx,%rbx
400544: callq   400550 <mult2>
400549: mov     %rax, (%rbx)
40054c: pop     %rbx
40054d: retq

...

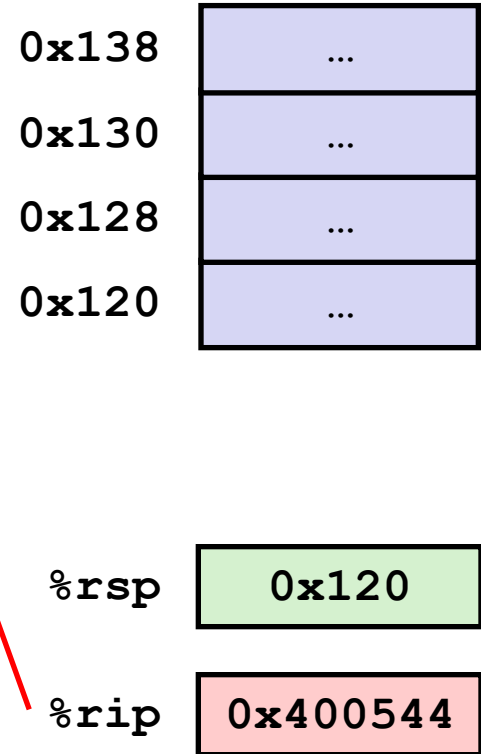
400550 <mult2>:
400550: mov     %rdi,%rax
400553: imul    %rsi,%rax
400557: retq
```

# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)



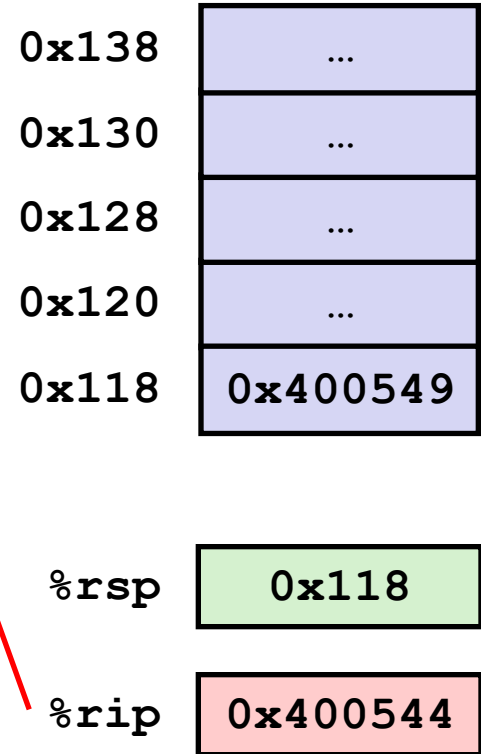


# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)

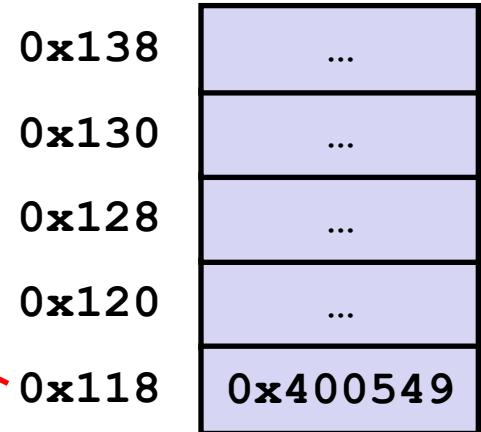


# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)



%rsp 0x118

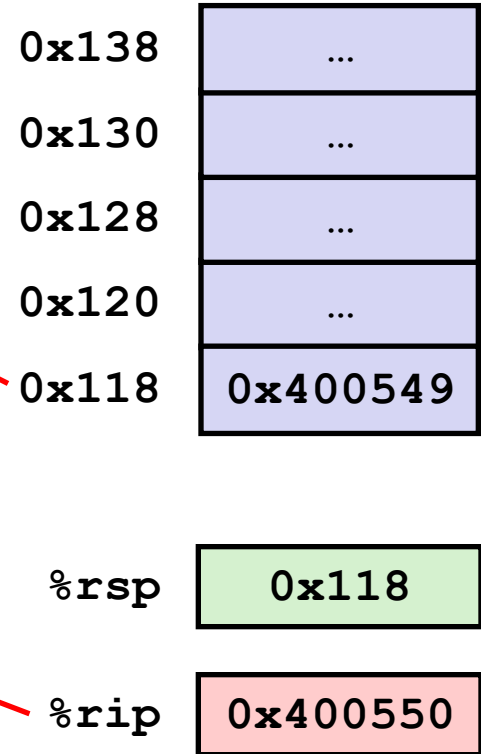
%rip 0x400544

# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)



# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)

0x138

...

0x130

...

0x128

...

0x120

...

0x118

0x400549

%rsp

0x118

%rip

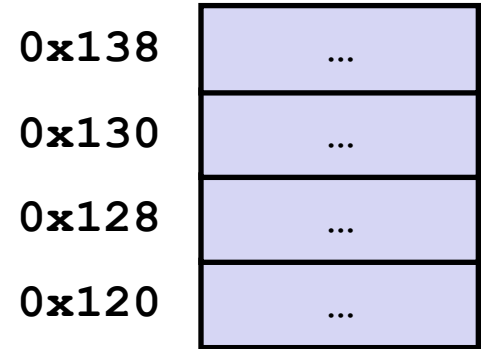
0x400557

# Function Call Example

```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
...  
...
```

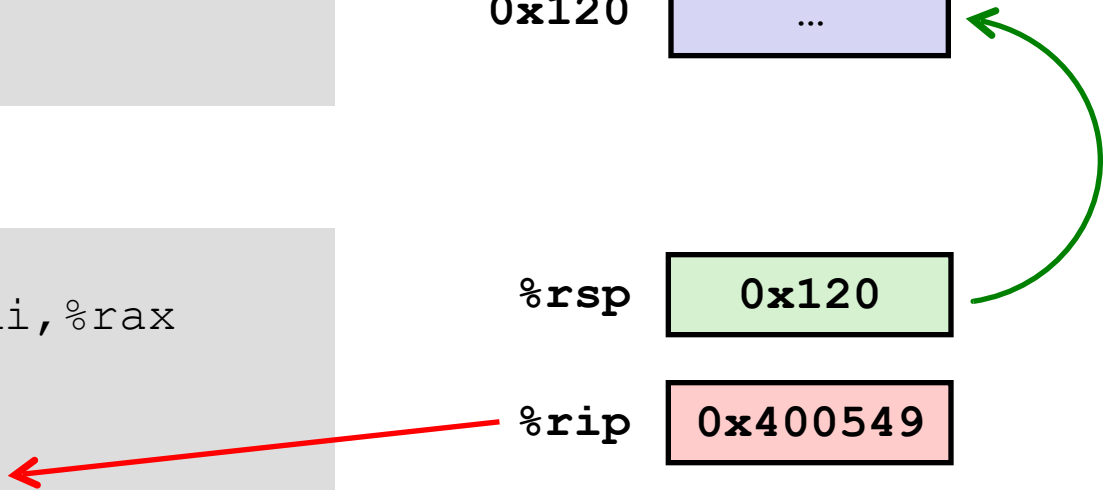
```
400550 <mult2>:  
400550: mov    %rdi, %rax  
...  
...  
400557: retq
```

Stack  
(Memory)



%rsp 0x120

%rip 0x400549

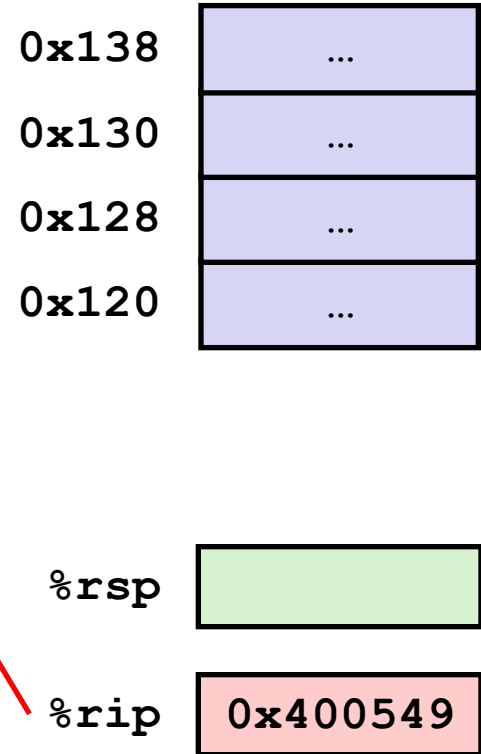


# Function Call Example

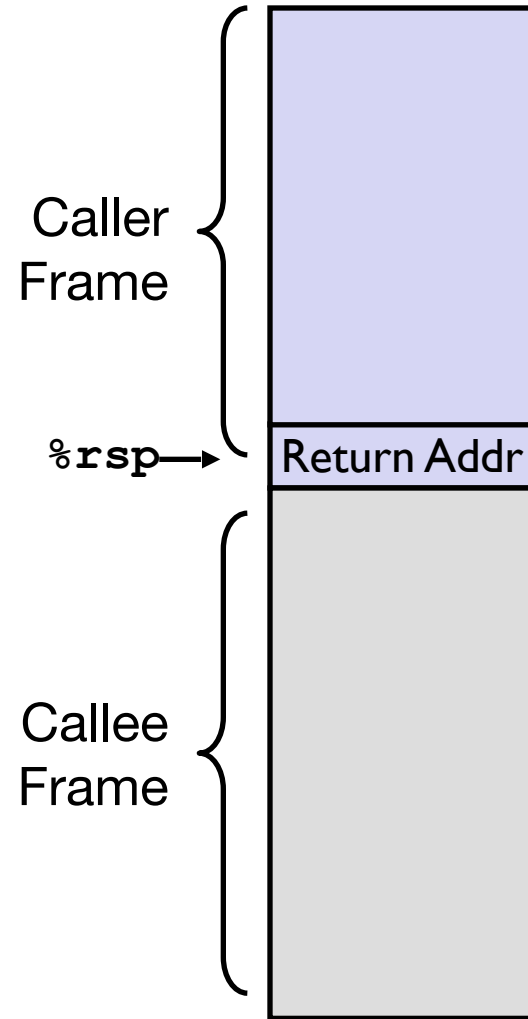
```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

Stack  
(Memory)



# Stack Frame (So Far...)



# Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- **Passing data**
- Managing local data



# Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount

## Registers

# Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount
- x86-64 convention (Part of the *Calling Conventions*):
  - First 6 arguments in registers, in specific order
  - The rest are pushed to stack
  - *Return value* is always in `%rax`

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

## Stack

...
Arg <i>n</i>
...
Arg 8
Arg 7

## Registers

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

# Passing Function Arguments

- Two choices: memory or registers
  - Registers are faster, but have limited amount
- x86-64 convention (Part of the *Calling Conventions*):
  - First 6 arguments in registers, in specific order
  - The rest are pushed to stack
  - *Return value* is always in %rax
- Just conventions, not laws
  - Not necessary if you write both caller and callee as long as the caller and callee agree
  - But is necessary to interface with others' code

## Stack

...
Arg <i>n</i>
...
Arg 8
Arg 7

# Function Call Data Flow Example

```
void multstore
(long x, long y, long *res) {
    long t = mult2(x, y);
    *res = t;
}
...
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

%rdi

%rsi

%rdx

%rcx

%r8

%r9

# Function Call Data Flow Example

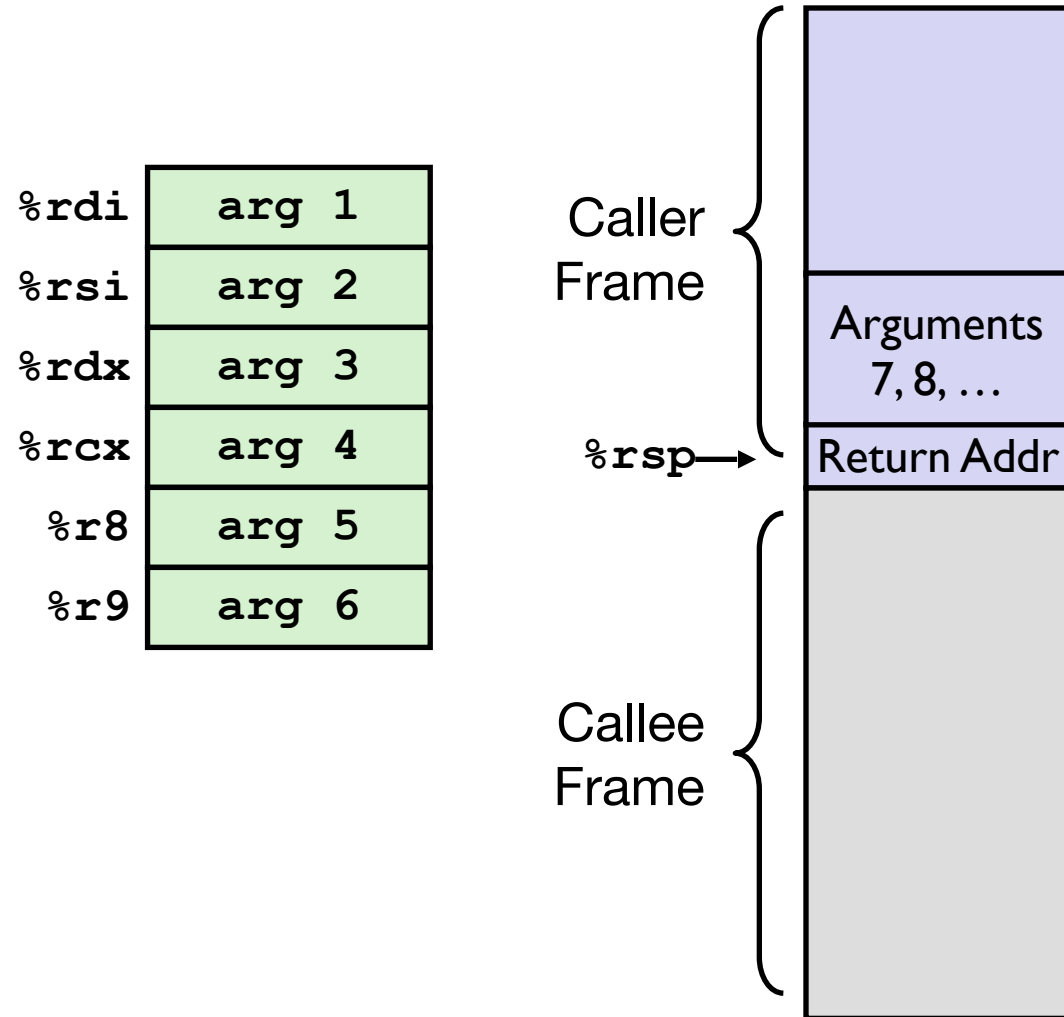
%rdi
%rsi
%rdx
%rcx
%r8
%r9

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

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

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, res in %rdx
...
400541: movq    %rdx,%rbx
400544: callq   400550 <mult2>
    # t in %rax
400549: movq    %rax, (%rbx)
...
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550: movq    %rdi,%rax
400553: imul    %rsi,%rax
    # s in %rax
400557: retq
```

# Stack Frame (So Far...)



# Today: How to Implement Function Call

- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

# Managing Function Local Variables

- Two ways: registers and memory (stack)
- Registers are faster, but limited. Memory is slower, but large. Smart compilers will optimize the usage.
- We will show different uses. Compiler optimizations later in the course.

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



# Register Example: `incr`

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

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

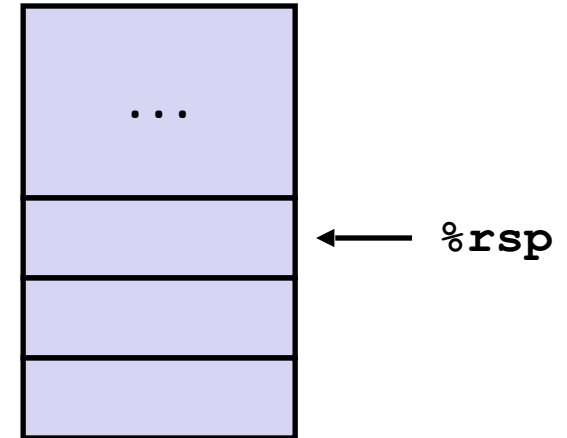
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v1+v3;  
}
```

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

Stack

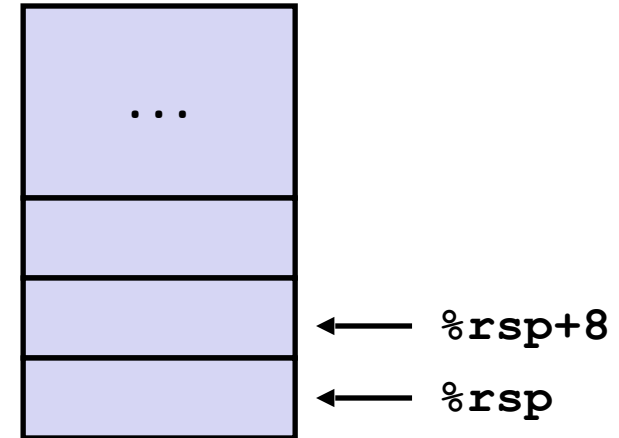


# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v1+v3;  
}
```

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

Stack

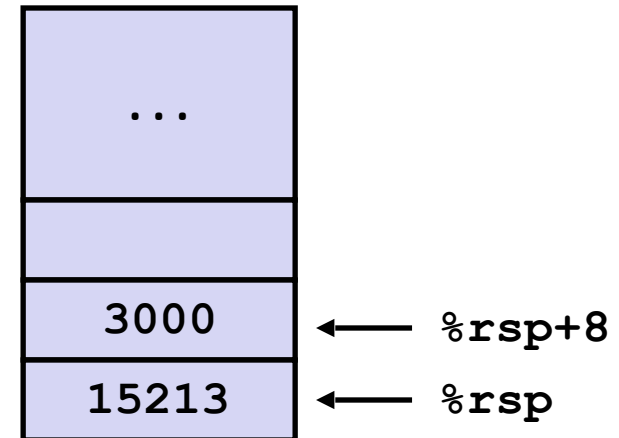


# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v1+v3;  
}
```

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

Stack

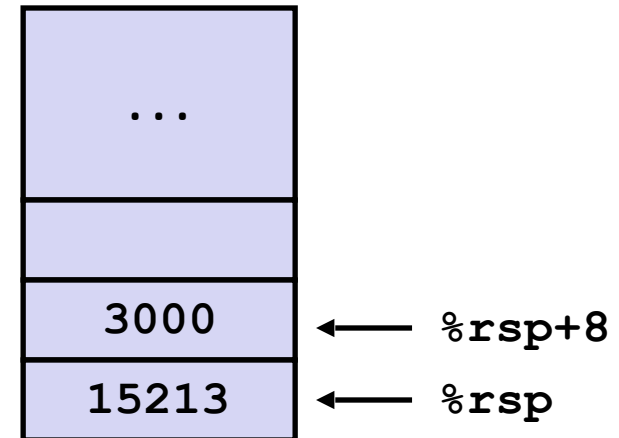


# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v1+v3;  
}
```

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

Stack



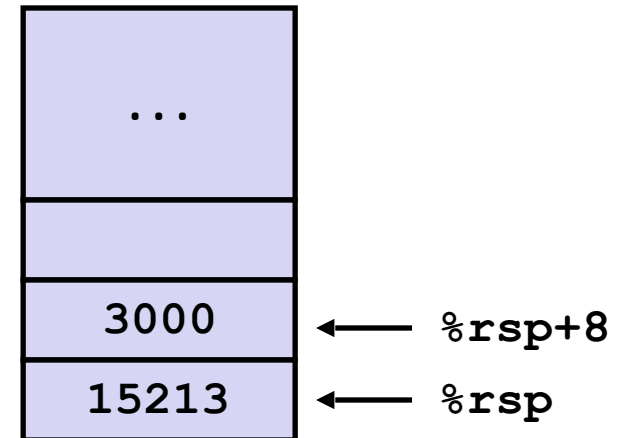
Register	Value(s)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	<code>&amp;v2</code>

# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v2+v3;  
}
```

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

Stack



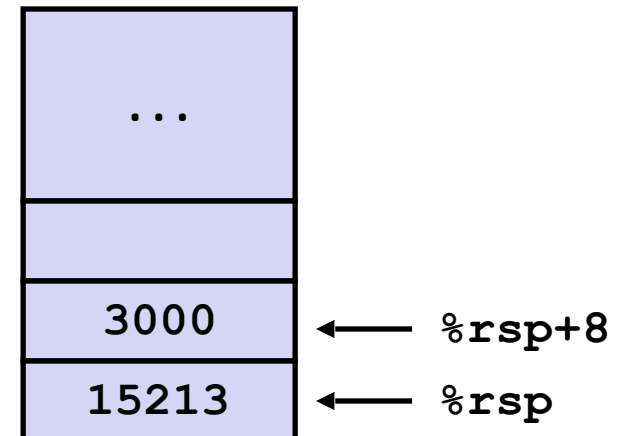
Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	18213

# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v2+v3;  
}
```

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

Stack



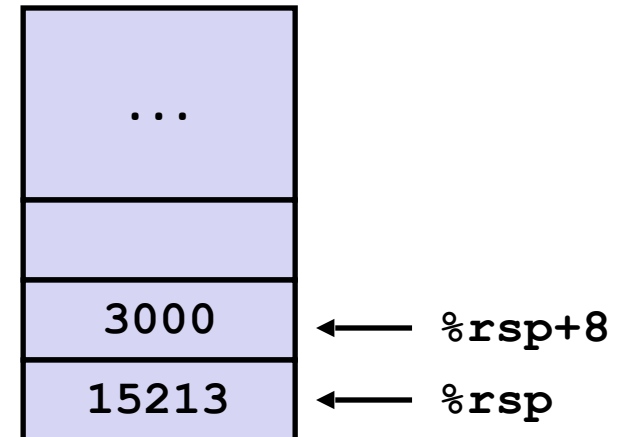
Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	21213

# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v2+v3;  
}
```

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

Stack



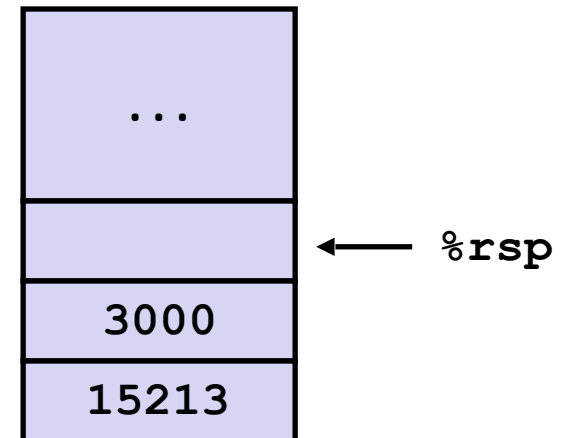


# Stack Example: `call_add`

```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v2+v3;  
}
```

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

Stack



# Stack Frame (So Far...)

