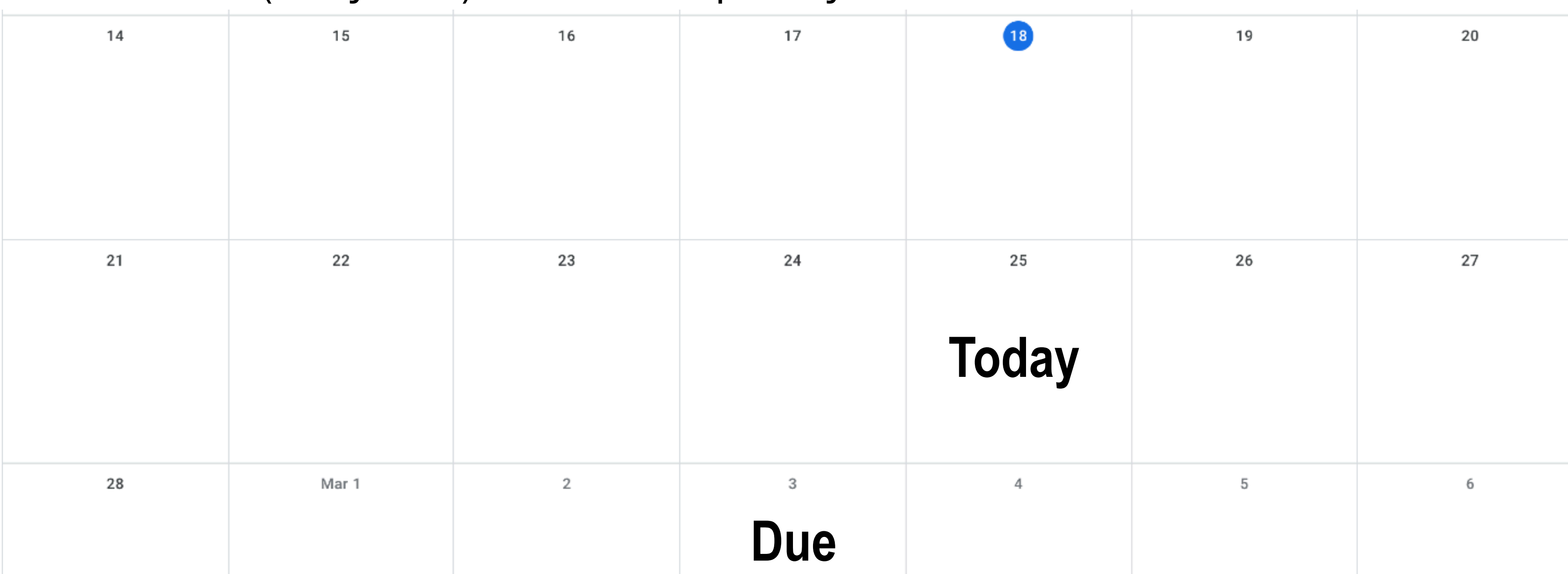
CSC 252: Computer Organization
Spring 2021: Lecture 8

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 2 is out
    - Details: https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment2.html
    - Due on **March 3**, 11:59 PM
    - You (may still) have 3 slip days



Announcement

- Programming assignment 2 is in x86 assembly language.
- Read the instructions before getting started!!!
  - You get 1/4 point off for every wrong answer
  - Maxed out at 10
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
- Problem set on arithmetics: https://www.cs.rochester.edu/courses/252/spring2021/handouts.html.
  - Not to be turned in.

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

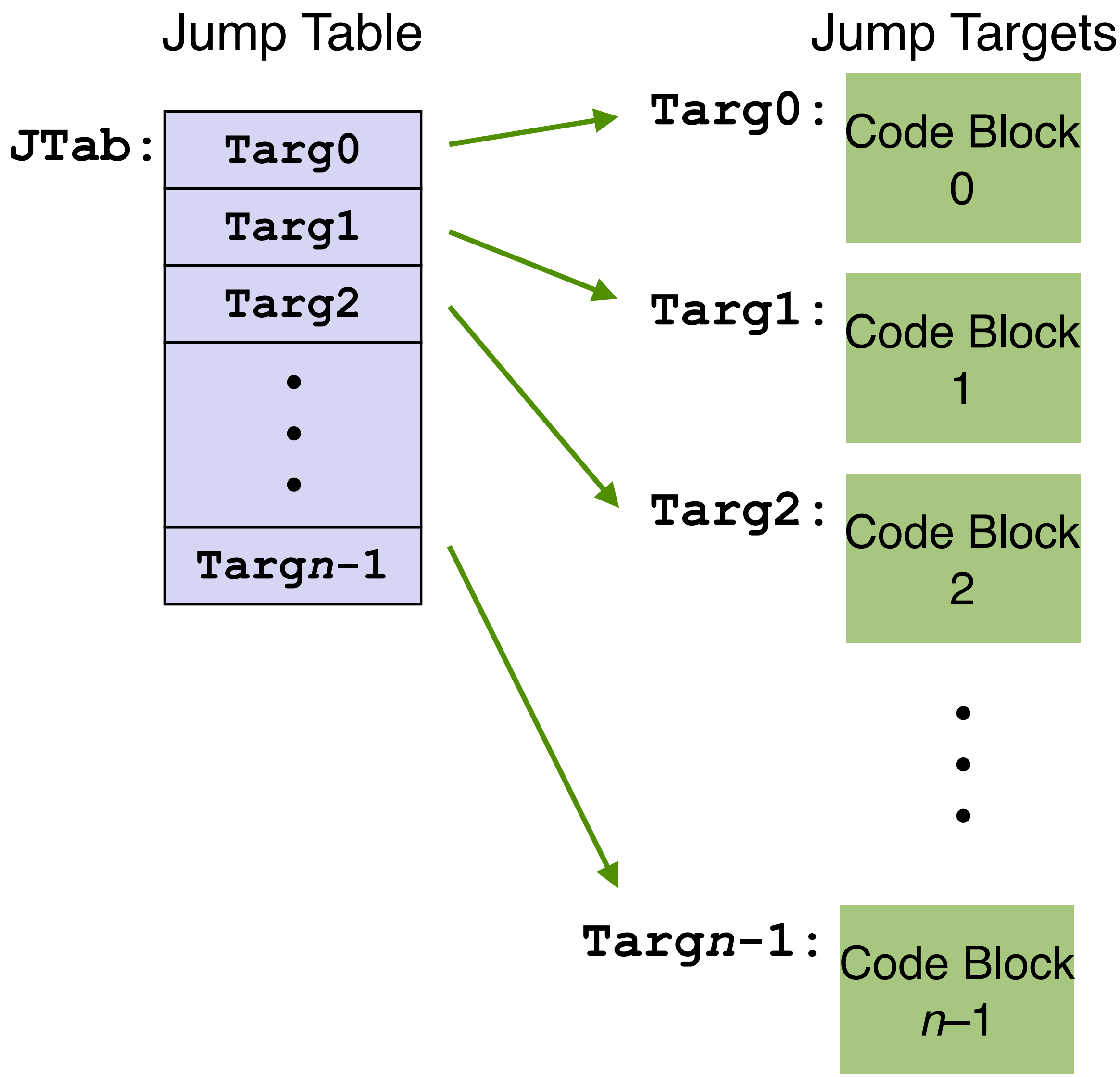
•
•
•

Targ*n*-1: Code Block *n*–1

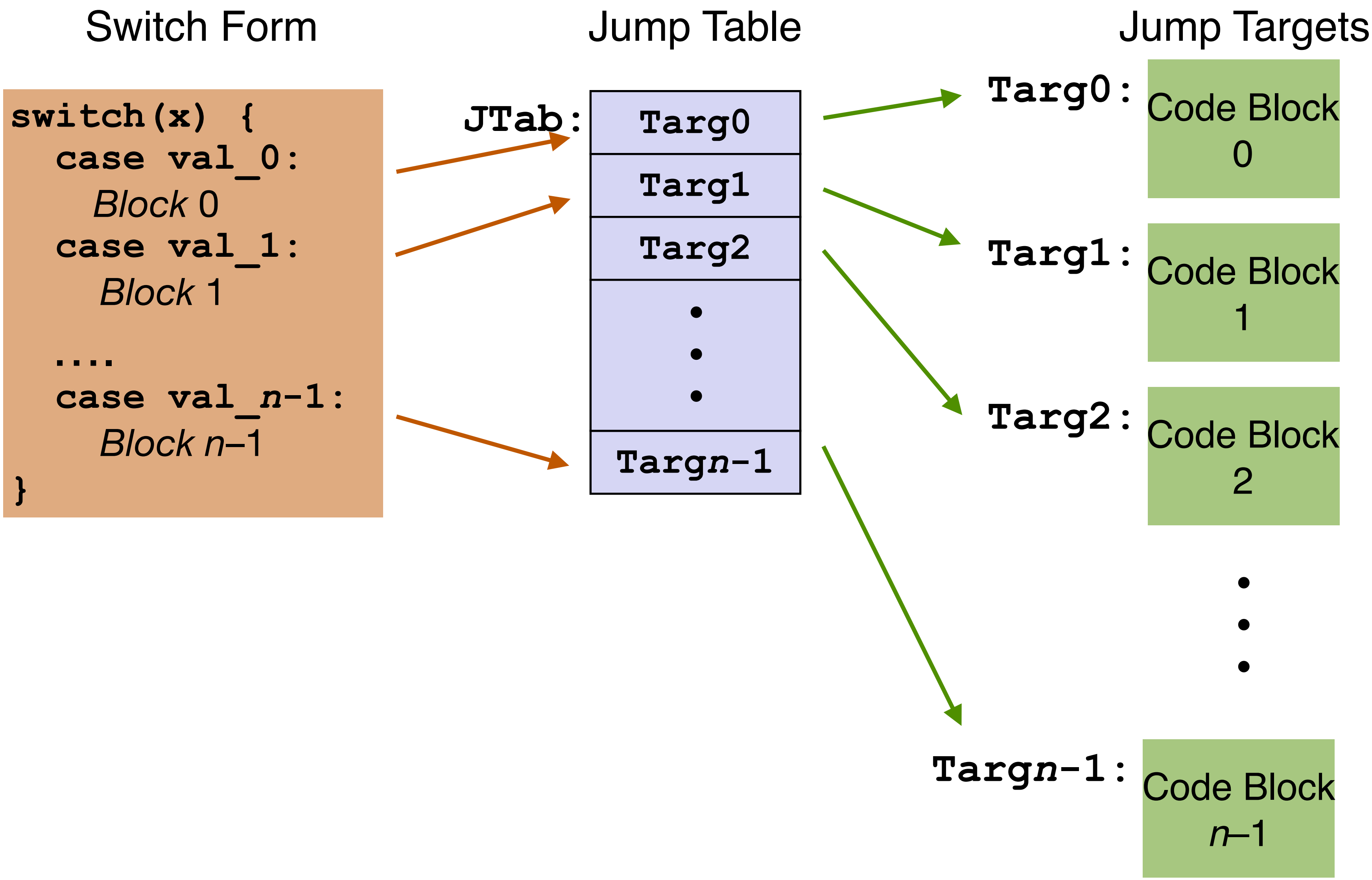
Implementing Switch Using Jump Table

Switch Form

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



Implementing Switch Using Jump Table



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

- Each code block starts from a unique address (Targ0, Targ1, …)
- Jump table stores all the target address
- Use the case value to index into the jump table to find where to jump to

Jump Table and Jump Targets

Jump Table

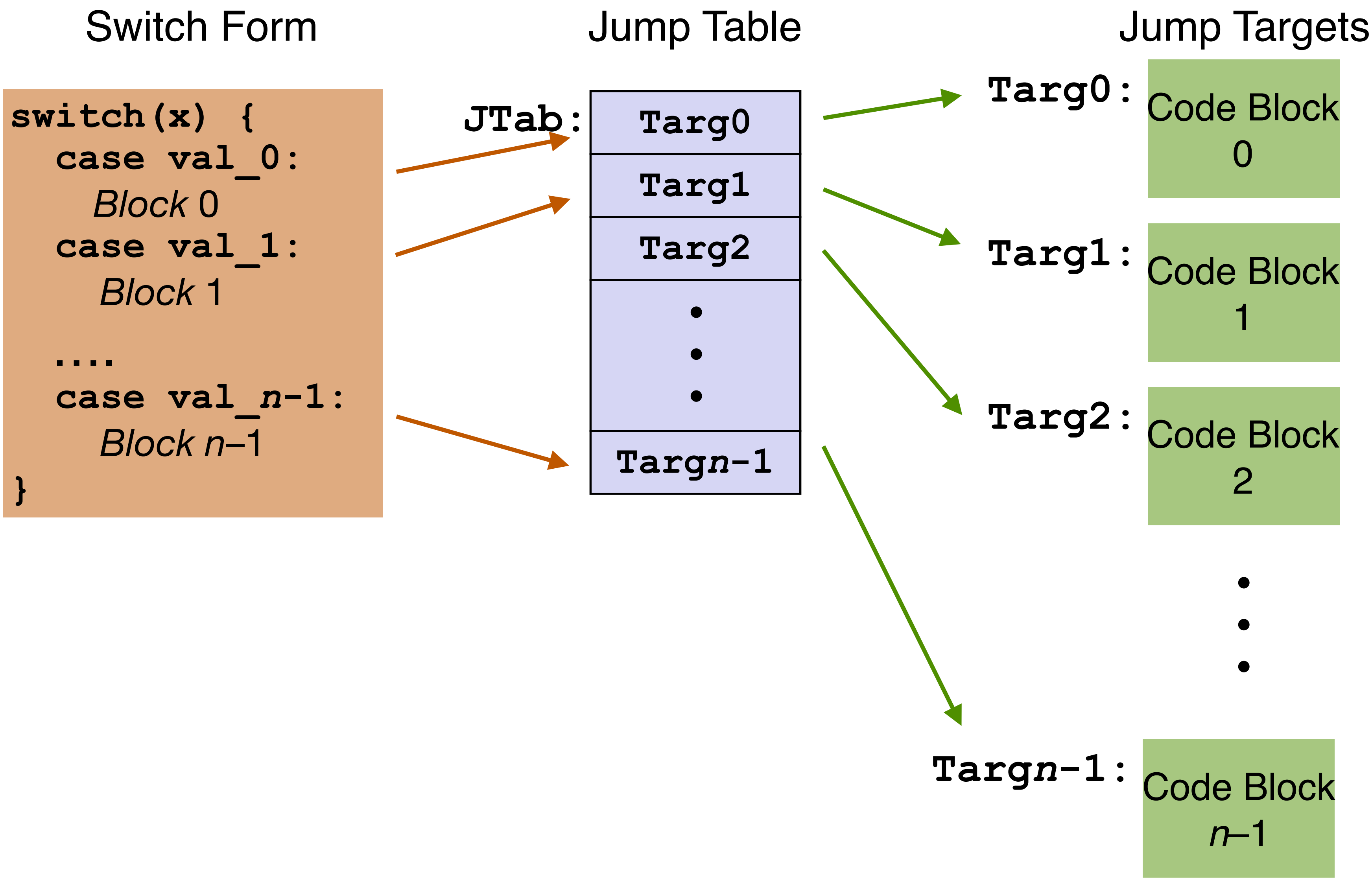
```
.section   .rodata
  .align 8
.L4:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

`jmp .L3` will go to `.L3` and start executing from there

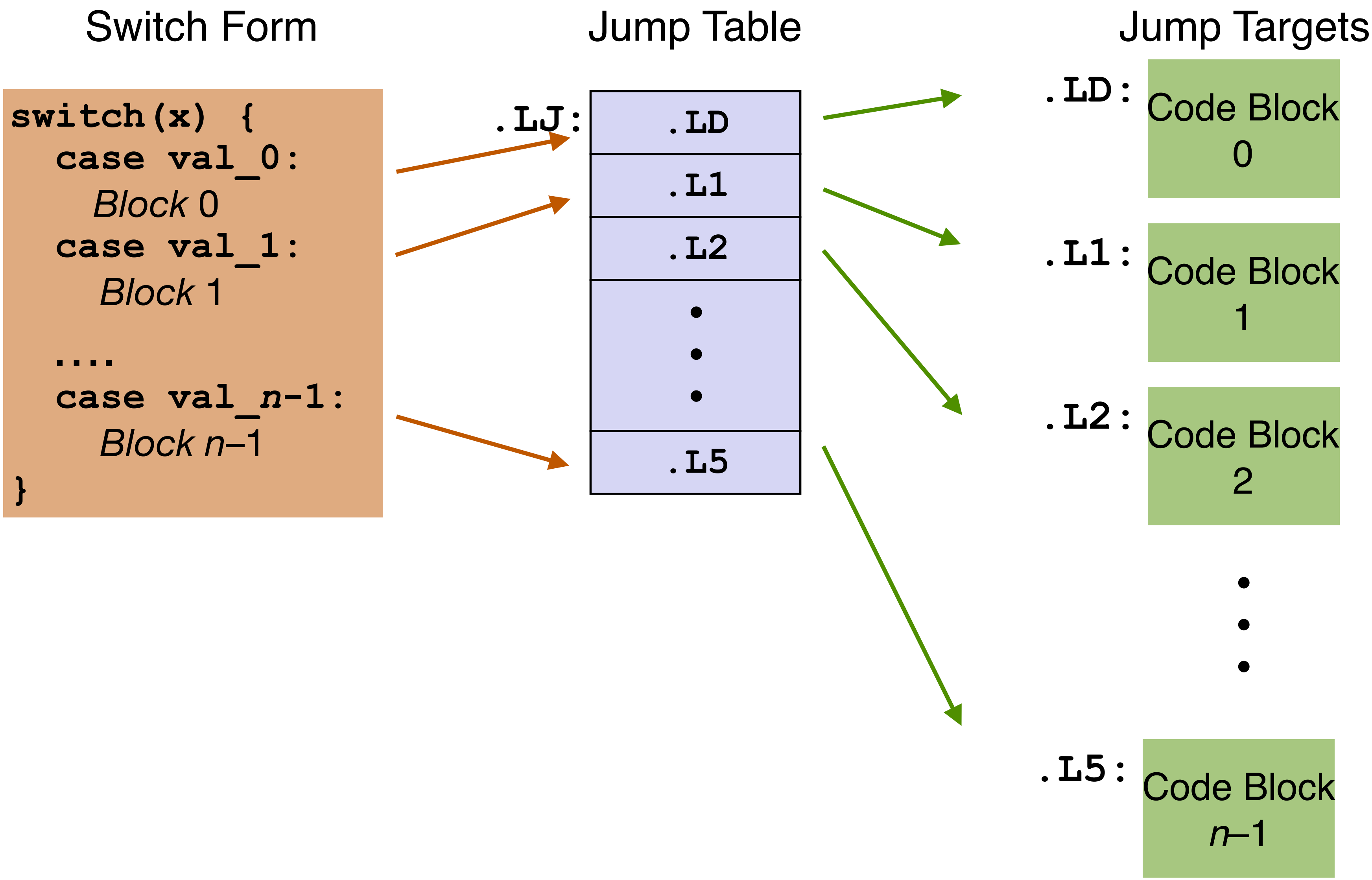
Jump Targets

```
.L1:                    # Case 1
    movq    %rsi, %rax
    imulq   %rdx, %rax
    jmp     .done
.L2:                    # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx
.L3:                    # Case 3
    addq    %rcx, %rax
    jmp     .done
.L5:                    # Case 5,6
    subq    %rdx, %rax
    jmp     .done
.LD:                    # Default
   movl    $2, %eax
   jmp    .done
```

Implementing Switch Using Jump Table



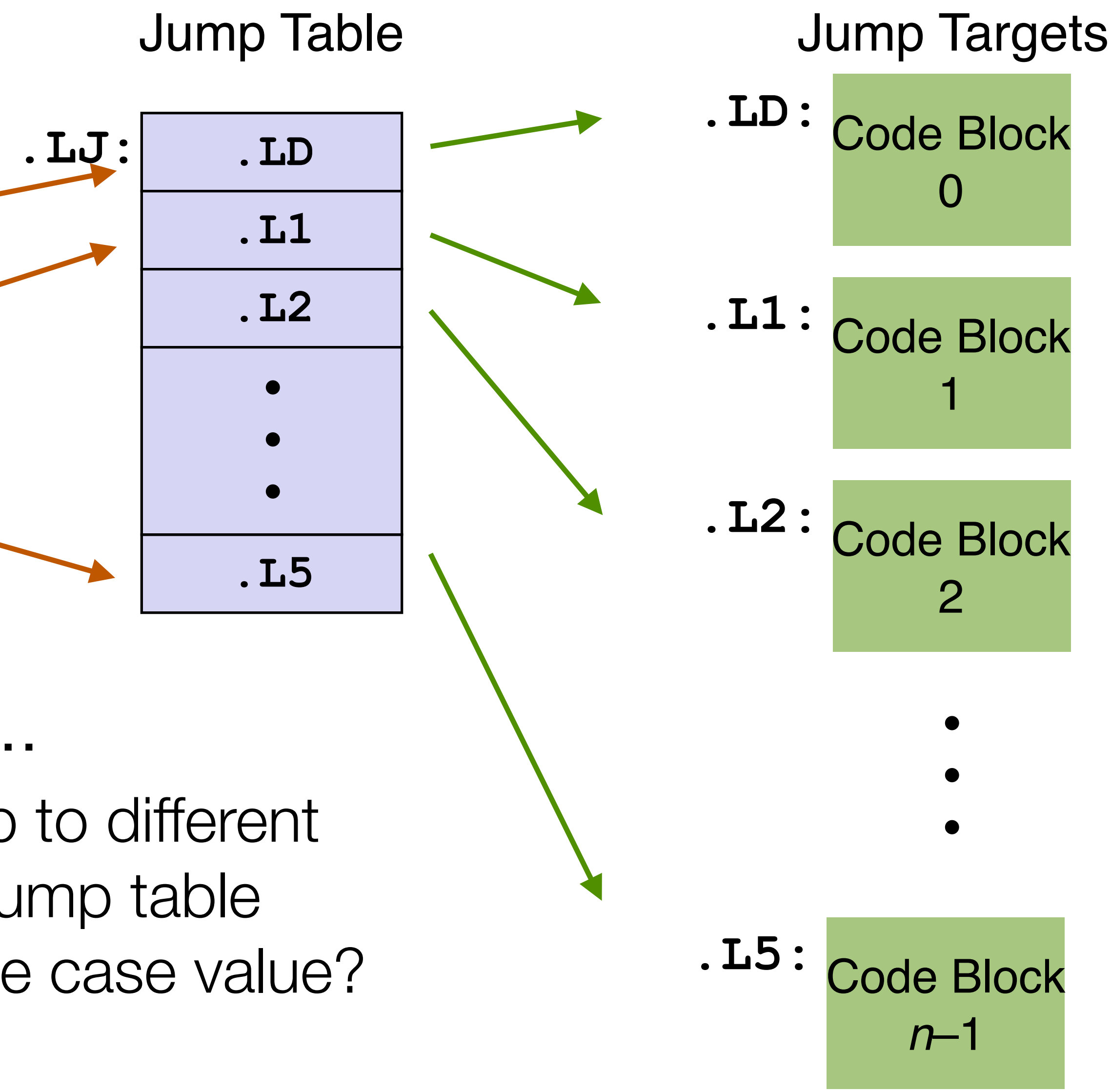
Implementing Switch Using Jump Table



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
}
```



- 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
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section  .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section   .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section   .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

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

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section  .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

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

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section   .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

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

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.section   .rodata
  .align 8
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

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

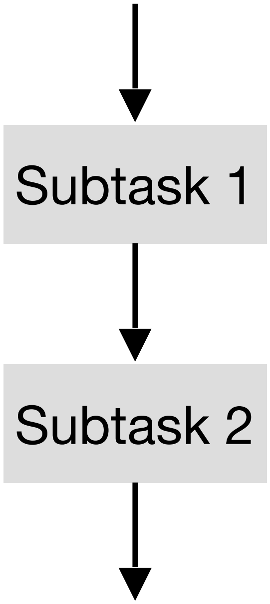
- Indirect Jump: **`jmp *%rax`**
  - `%rax` specifies the address to jump to (PC = `%rax`)
- Direct Jump (**`jmp .LJ`**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

An equivalent syntax in x86:

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

Summary

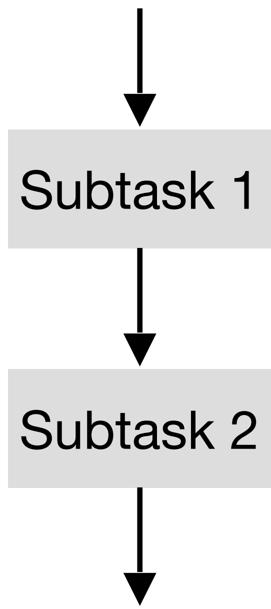
Sequential




```
a = x + y;
y = a - c;
...
```

Summary

Sequential



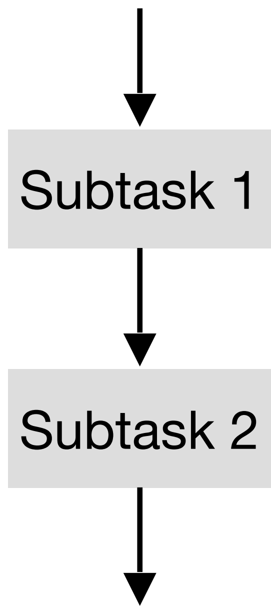

```
a = x + y;
y = a - c;
…
```

Memory

```
.section .text

…

<subtask 1>

…

…

<subtask 2>

…
```

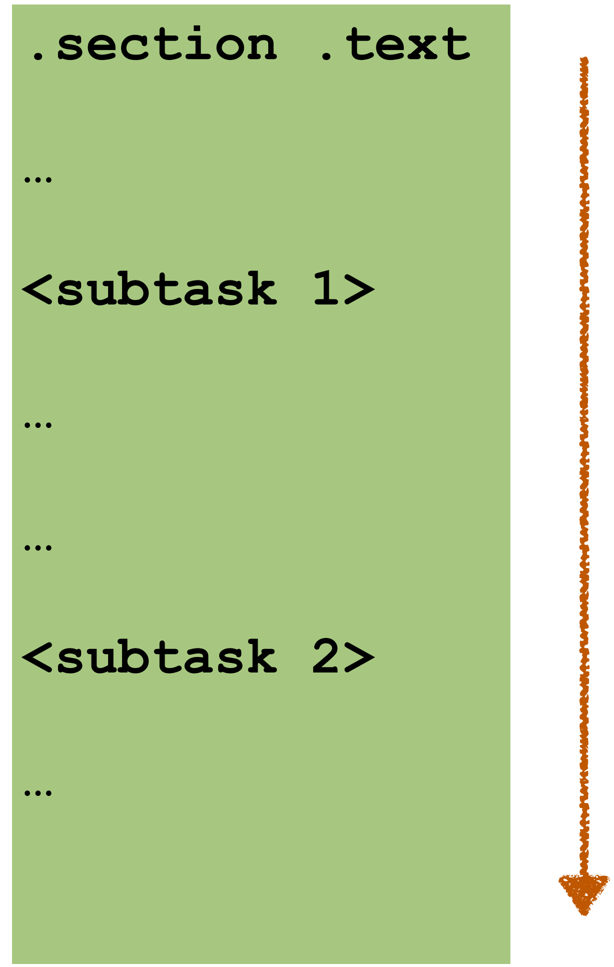
Summary

Sequential



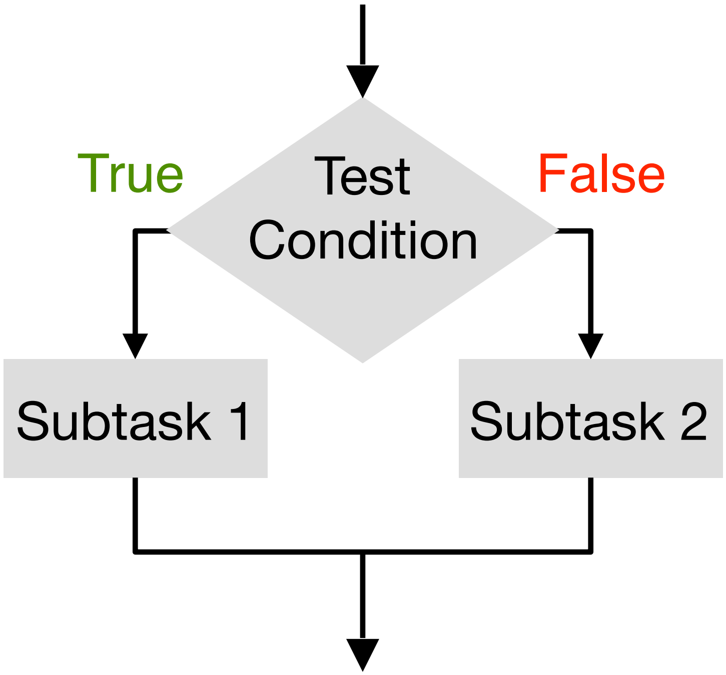

```
a = x + y;
y = a - c;
…
```

Memory



Summary

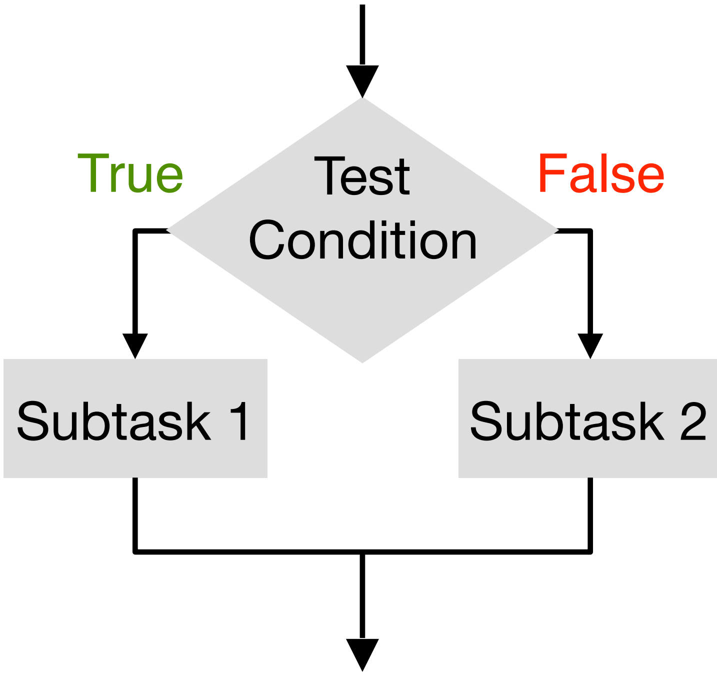
Conditional




```
if (x > y) r = x - y;
else r = y - x;
```

Summary

Conditional



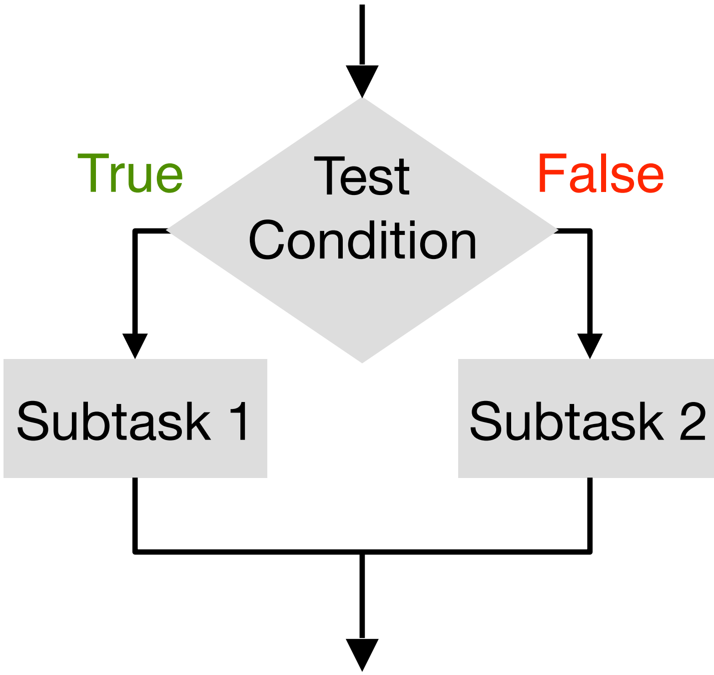

```
if (x > y) r = x - y;
else r = y - x;
```

Memory

```
.section .text

…
cmpq
jle .L2
.L1 <subtask 1>

…

jmp .done
.L2 <subtask 2>

…
.done
```

Summary

Conditional



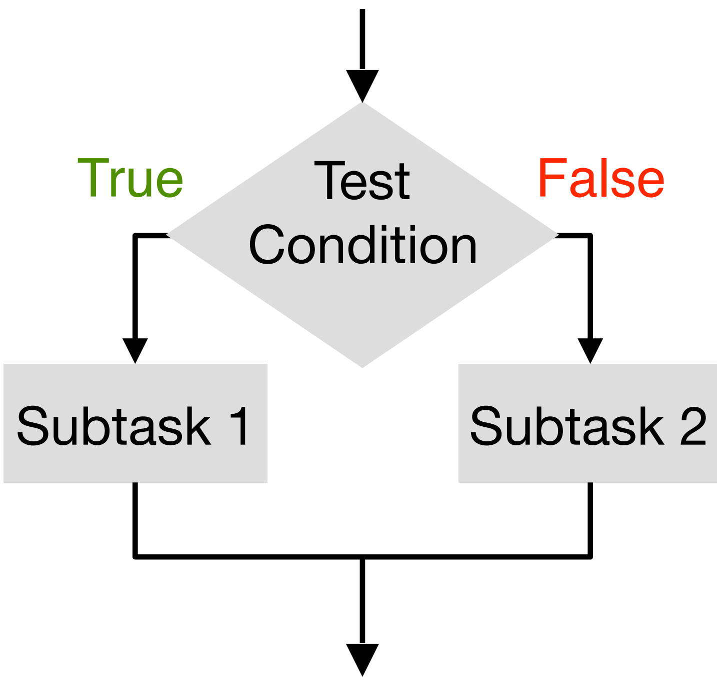

```
if (x > y) r = x - y;
else r = y - x;
```

Memory

```
.section .text

…
cmpq
jle .L2
.L1 <subtask 1>

…

jmp .done
.L2 <subtask 2>

…
.done
```

Summary

Conditional



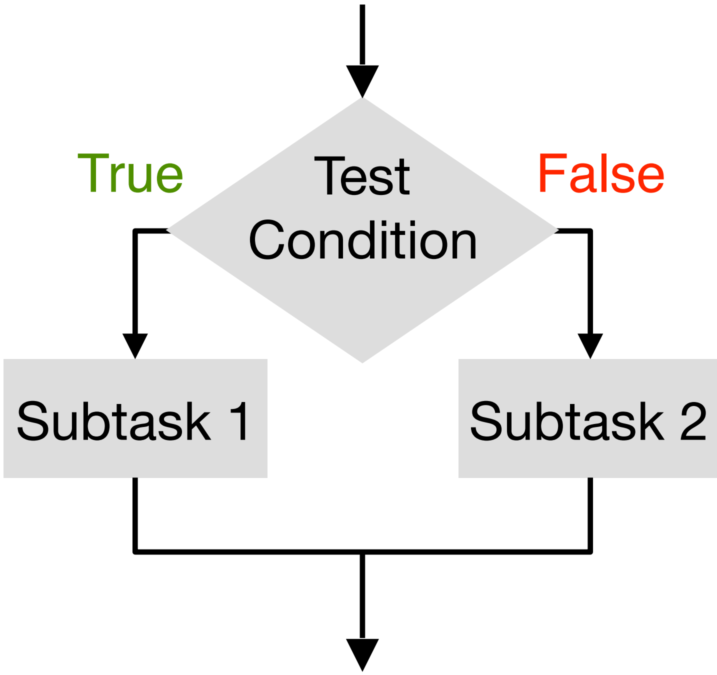

```
if (x > y) r = x - y;
else r = y - x;
```

Memory

```
.section .text

…
cmpq
jle .L2
.L1 <subtask 1>

…

jmp .done
.L2 <subtask 2>

…
.done
```

Summary

Conditional

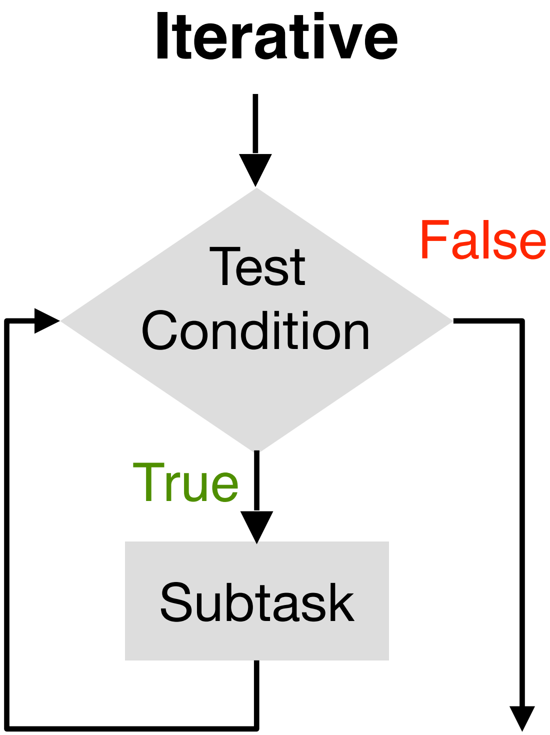



```
if (x > y) r = x - y;
else r = y - x;
```

Memory

```
.section .text

…
cmpq
jle .L2
.L1 <subtask 1>

…

jmp .done
.L2 <subtask 2>

…
.done
```

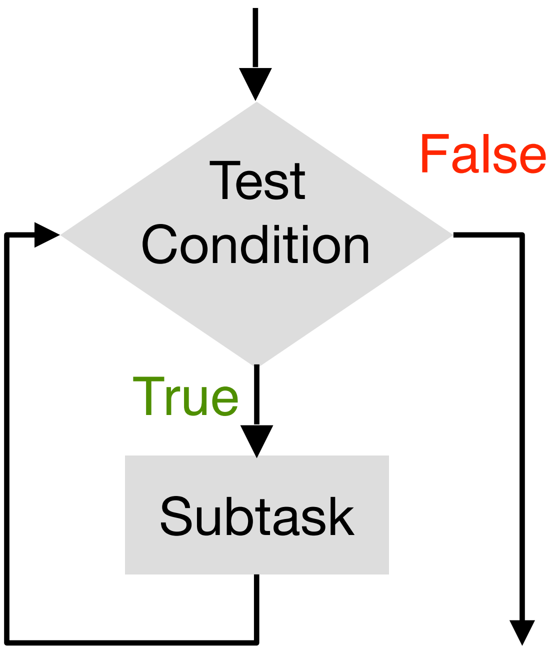
Summary




```
while (x > 0) {
  x--;
}
```

Summary

Iterative



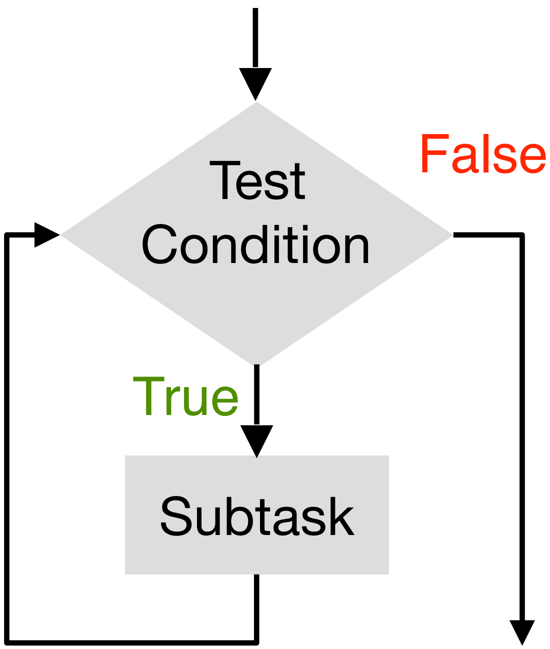

```
while (x > 0) {
   x--;
}
```

Memory

```
.section .text

…
addq
jmp .L2
.L1:


…
   <subtask>
…
.L2:
   cmpq A, B
   jg .L1
…
…
```

Summary

Iterative



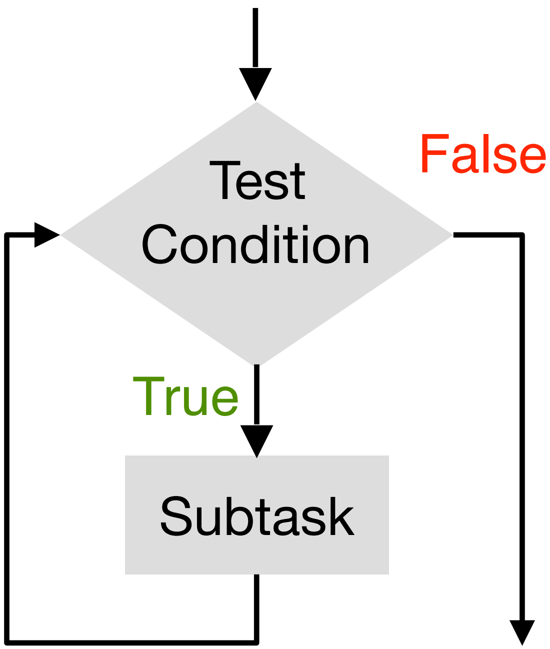

```
while (x > 0) {
   x--;
}
```

Memory

```
.section .text

…
addq
jmp .L2
.L1:

…
   <subtask>
…
.L2:
   cmpq A, B
   jg .L1
…
…
```

Summary

Iterative



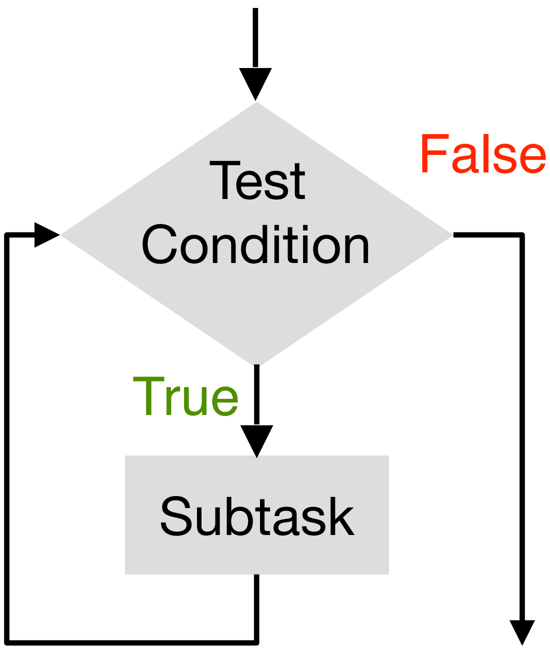

```
while (x > 0) {
  x--;
}
```

Memory

```
.section .text

…
addq
jmp .L2
.L1:

…
  <subtask>
…
.L2:
  cmpq A, B
  jg .L1
…
…
```

Summary

Iterative



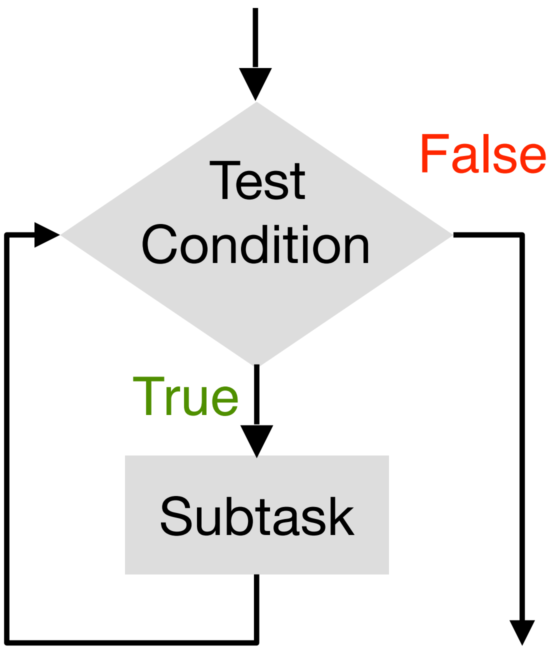

```
while (x > 0) {
  x--;
}
```

Memory

```
.section .text

…
addq
jmp .L2
.L1:

…
  <subtask>
…
.L2:
  cmpq A, B
  jg .L1
…
…
```

Summary

Iterative




```
while (x > 0) {
  x--;
}
```

Memory

```
.section .text

…
addq
jmp .L2
.L1:

…
   <subtask>
…
.L2:
   cmpq A, B
   jg .L1
…
…
```

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 {
    WhitesTurn();
    BlacksTurn();
  } while (NoOutcomeYet());
}
```

Structure of program is evident, even without knowing implementation.

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];
}
```

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 a 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 a function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory

General Idea

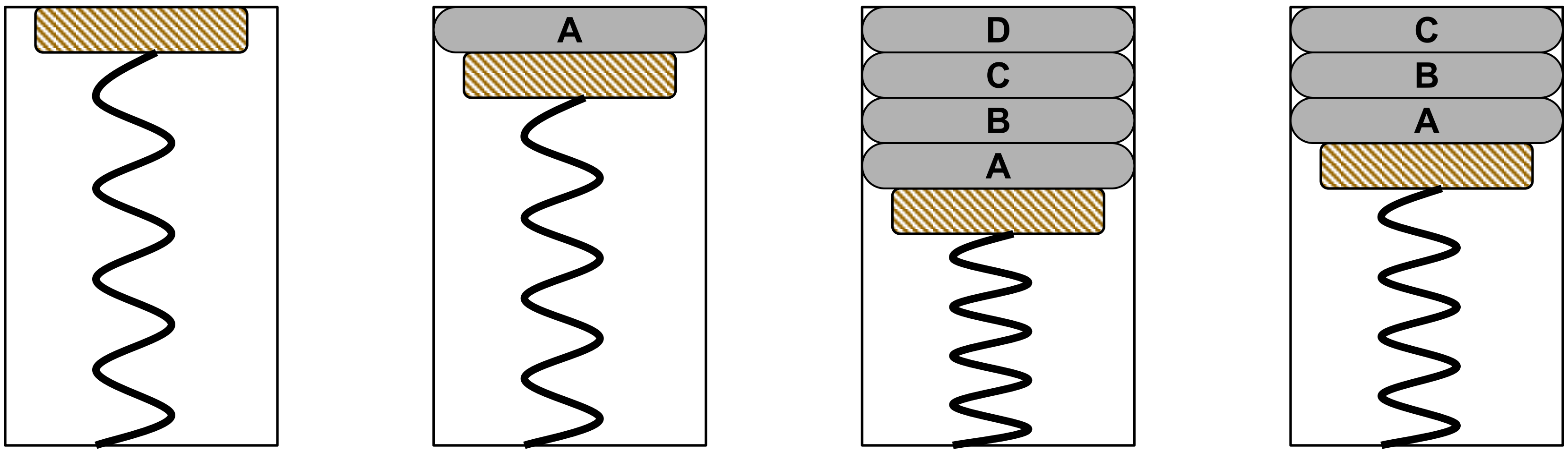
- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute a function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory

General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute a 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

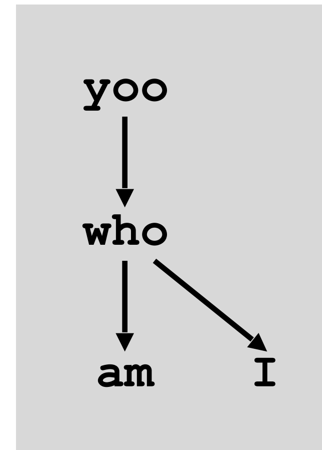
```
yoo(…)
{
   •
   •
   who();
   •
   return
}
```

```
who(…)
{
   •  •  •
   am();
   •  •  •
   I();
   return;
}
```

```
am(…)
{
   •
   •
   •
   return;
}
```

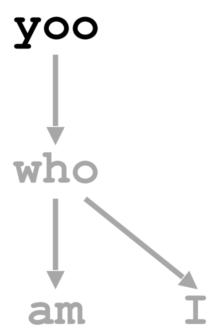
```
I(…)
{
   •
   •
   •
   return;
}
```

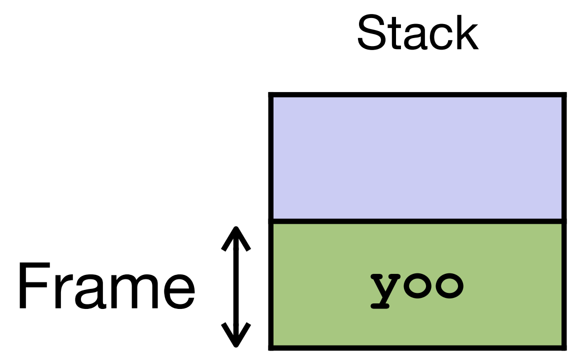
Example Call Chain



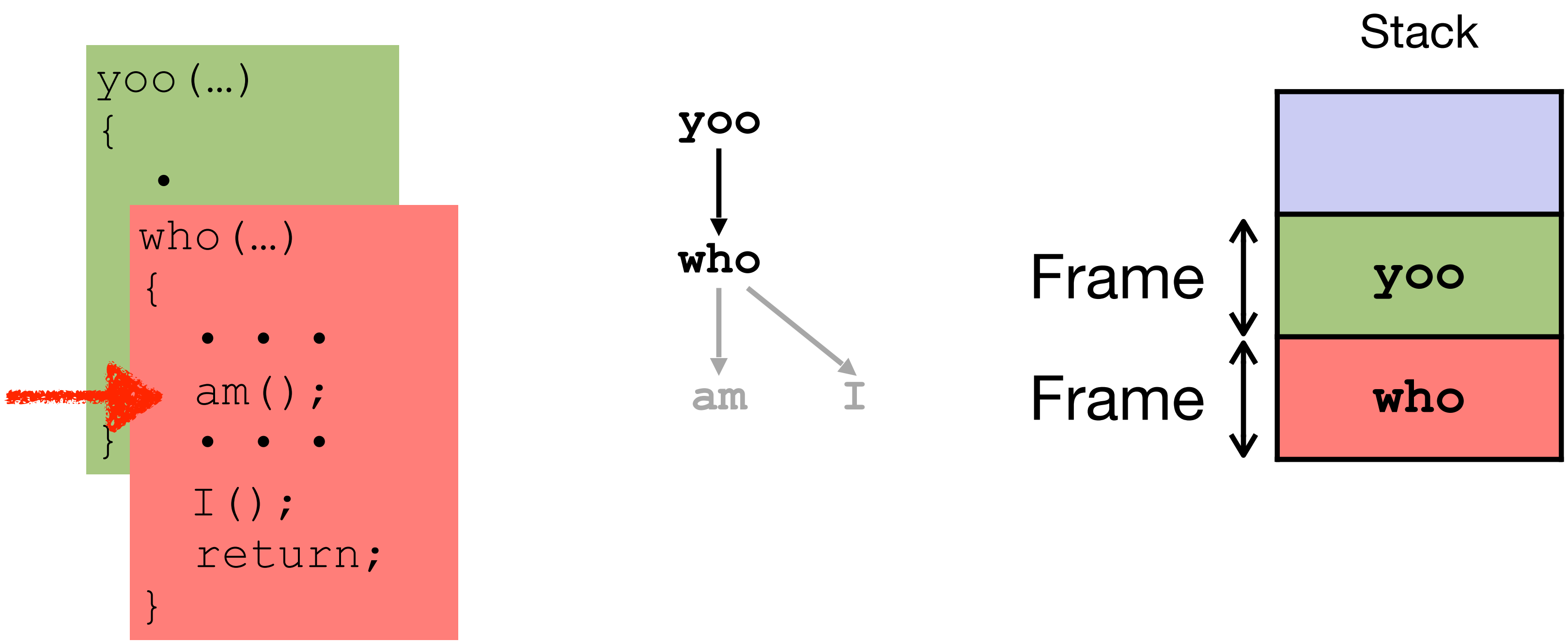
Run-Time Stack During Function Call

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

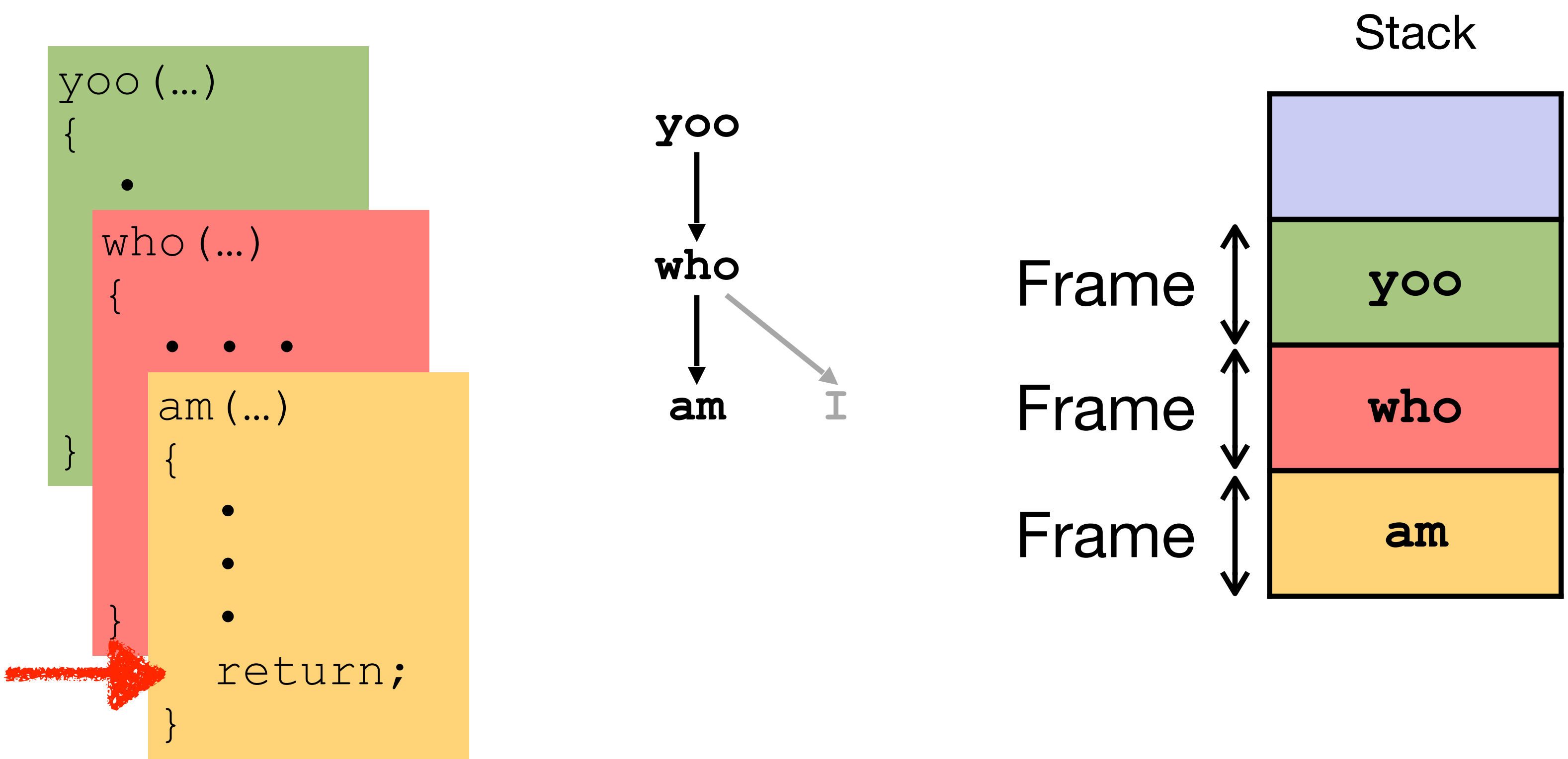




Run-Time Stack During Function Call



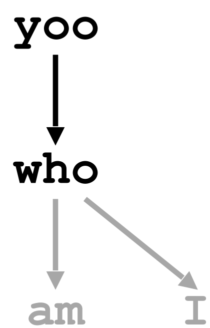
Run-Time Stack During Function Call

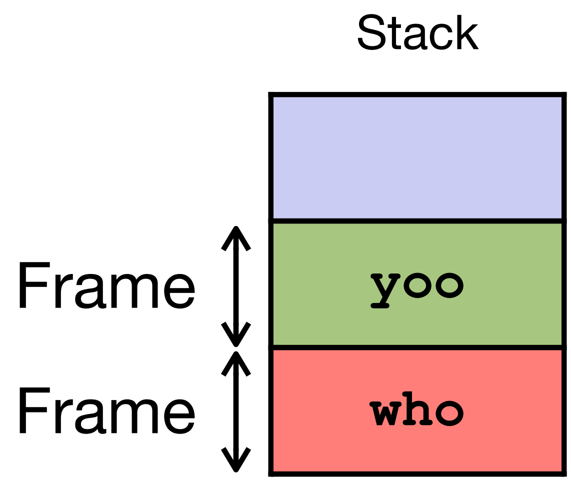


Run-Time Stack During Function Call

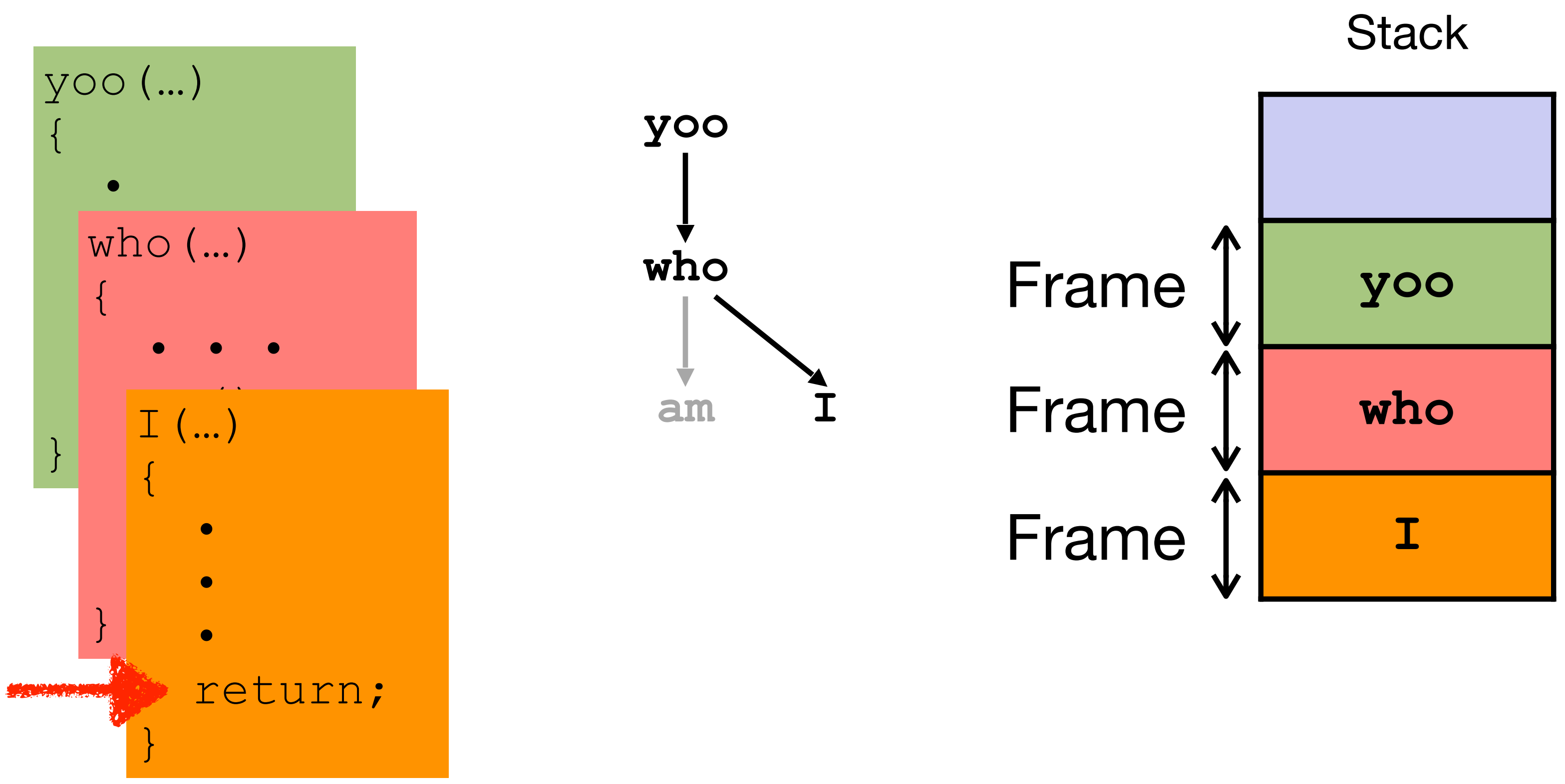
```
yoo(…)
{
  •
}
```

```
who(…)
{
  • • •
  am();
  • • •
  I();
  return;
}
```





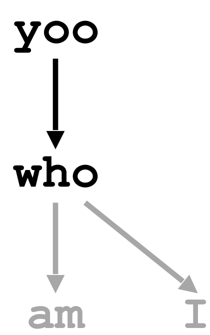
Run-Time Stack During Function Call

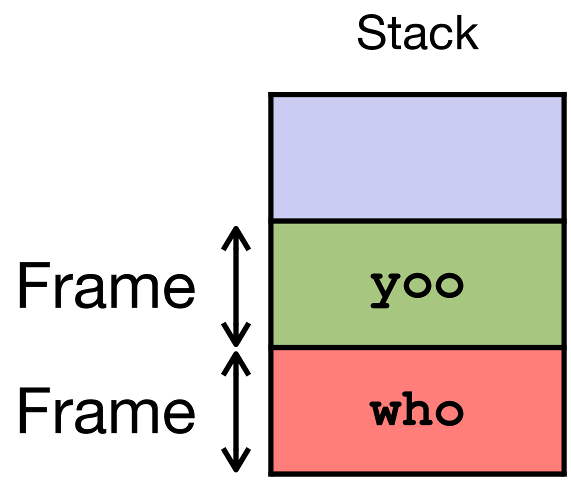


Run-Time Stack During Function Call

```
yoo(…)
{
  •
}
```

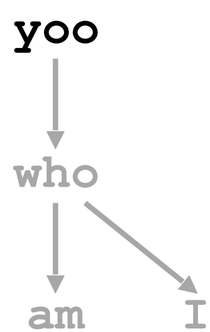
```
who(…)
{
  • • •
  am();
  • • •
  I();
  return;
}
```

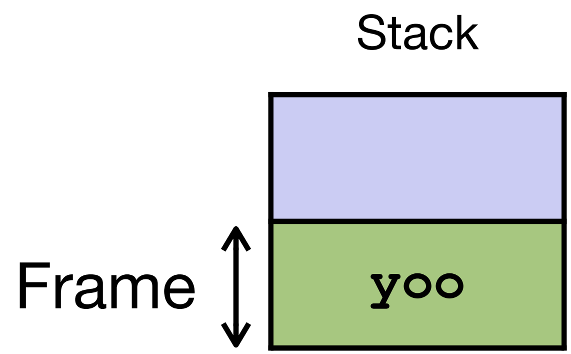




Run-Time Stack During Function Call

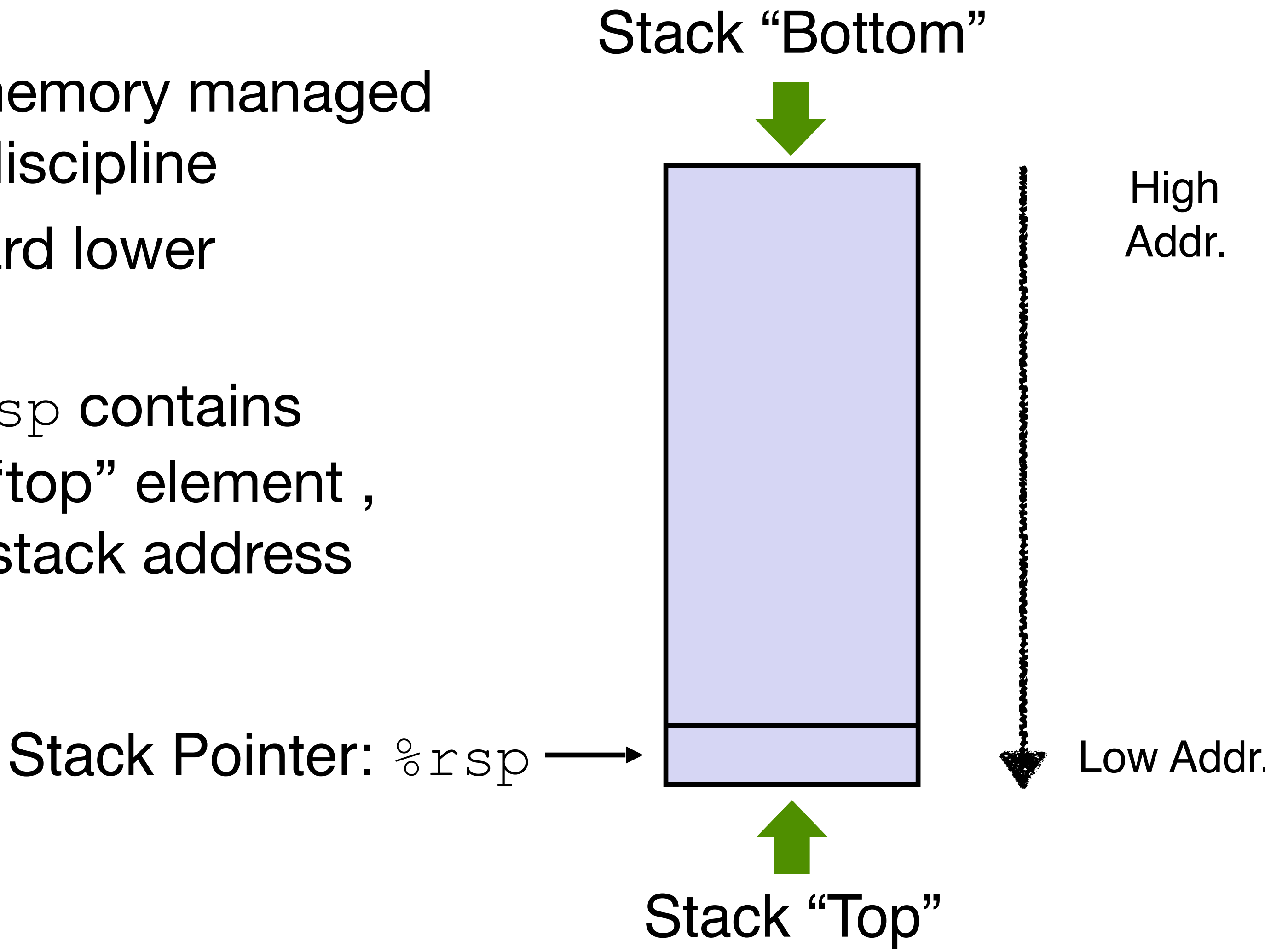
```
yoo(…)
{
   •
   •
   who();
   •
   return
}
```





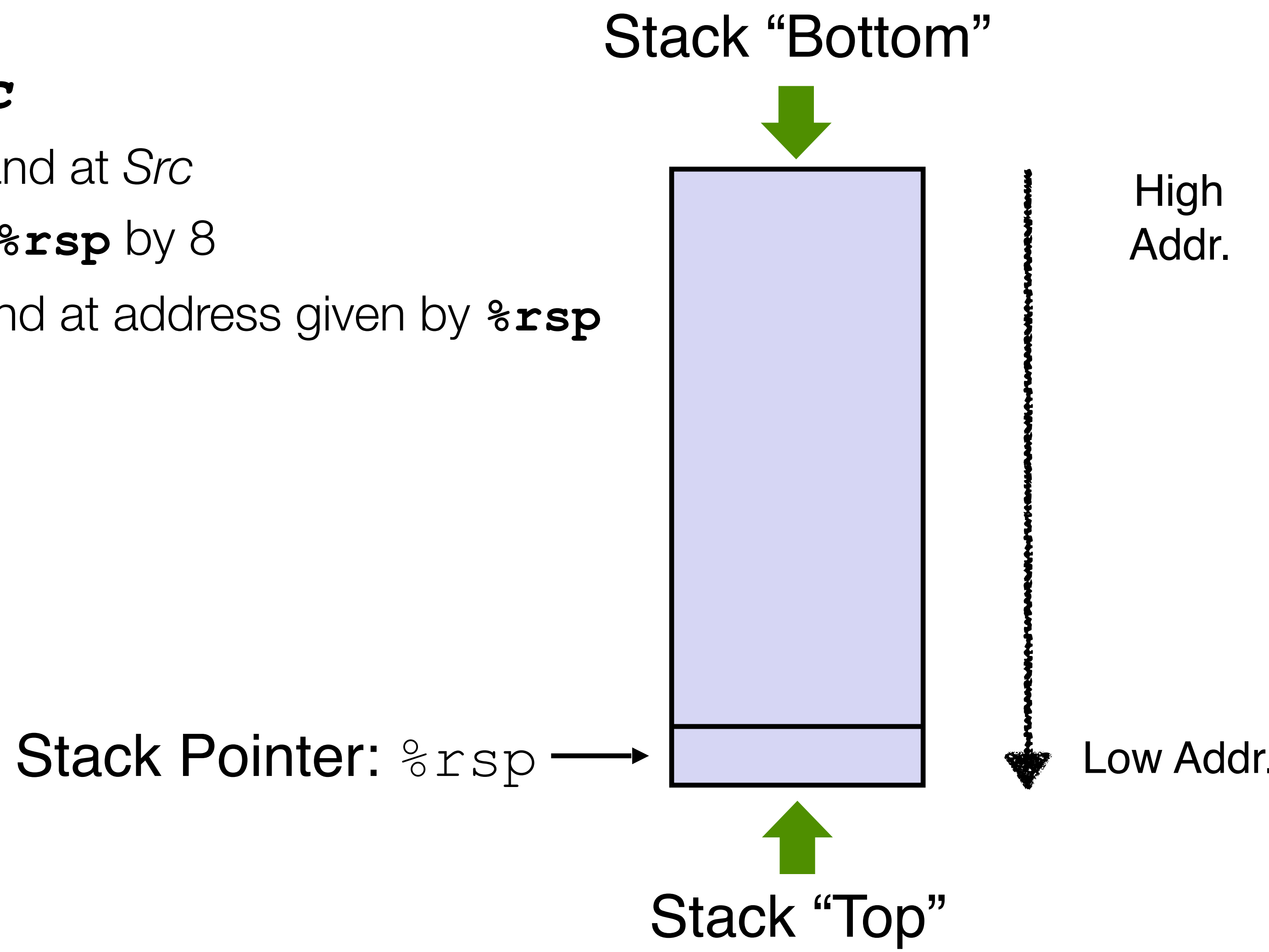
Stack in X86-64

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



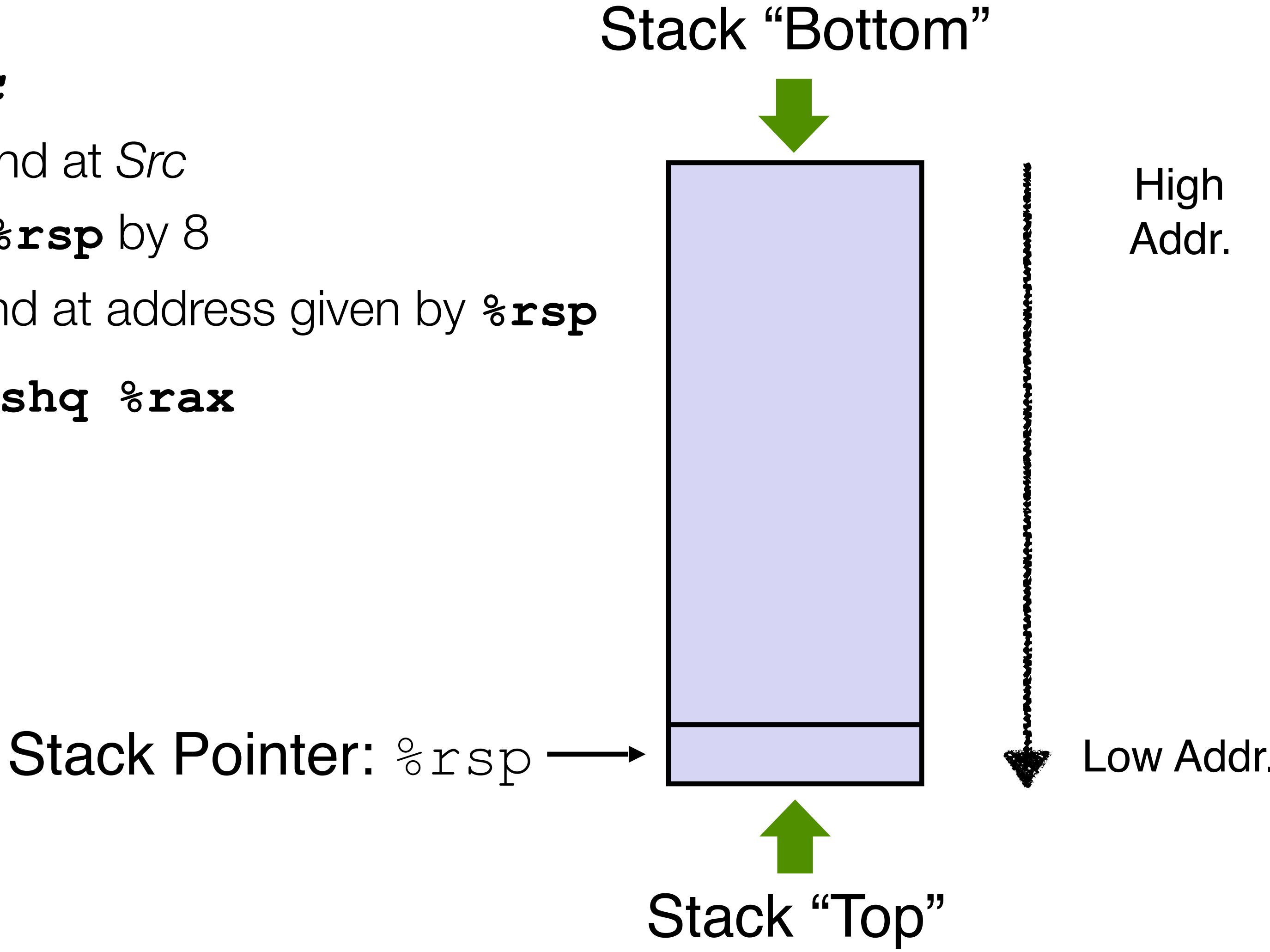
x86-64 Stack: Push

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



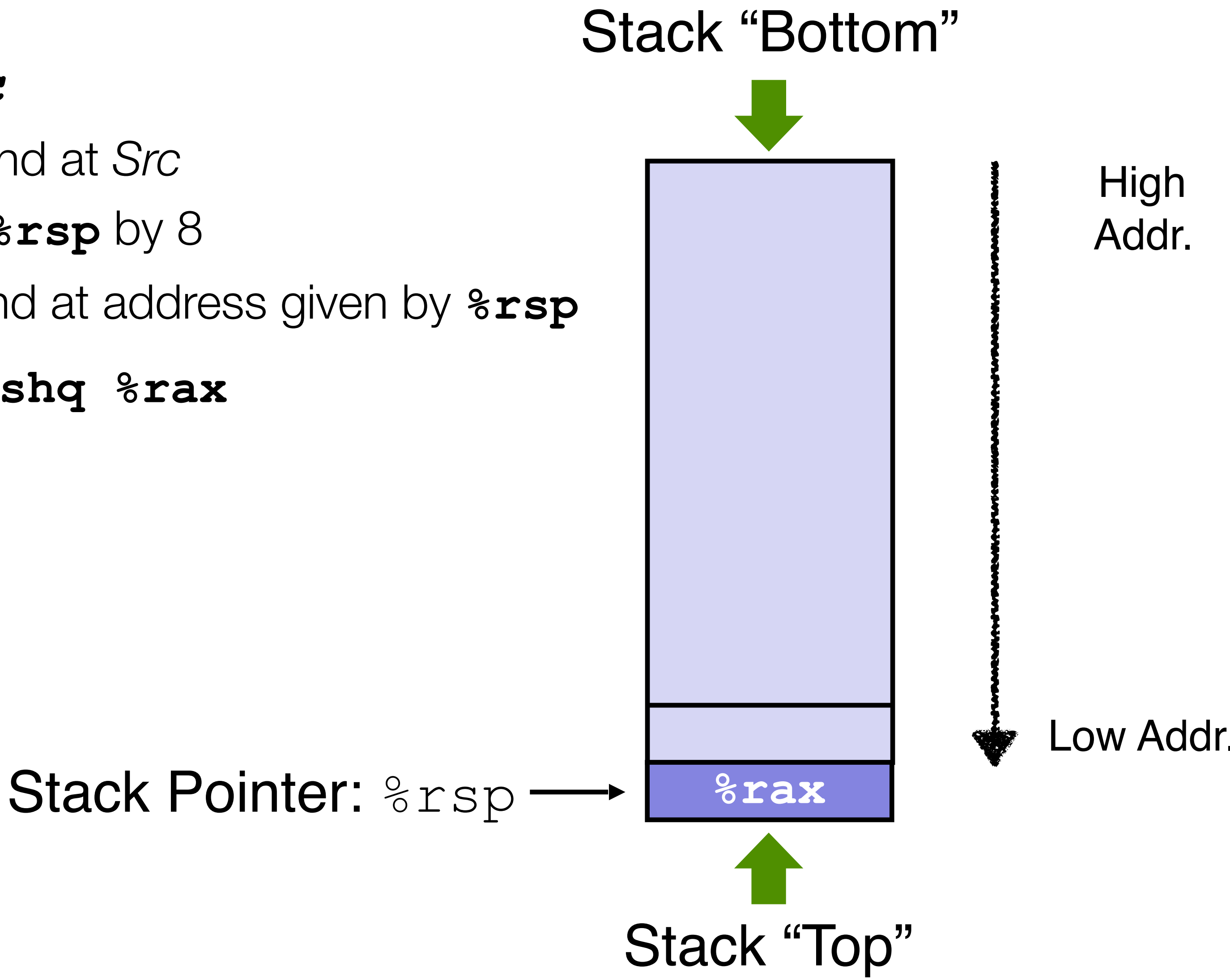
x86-64 Stack: Push

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



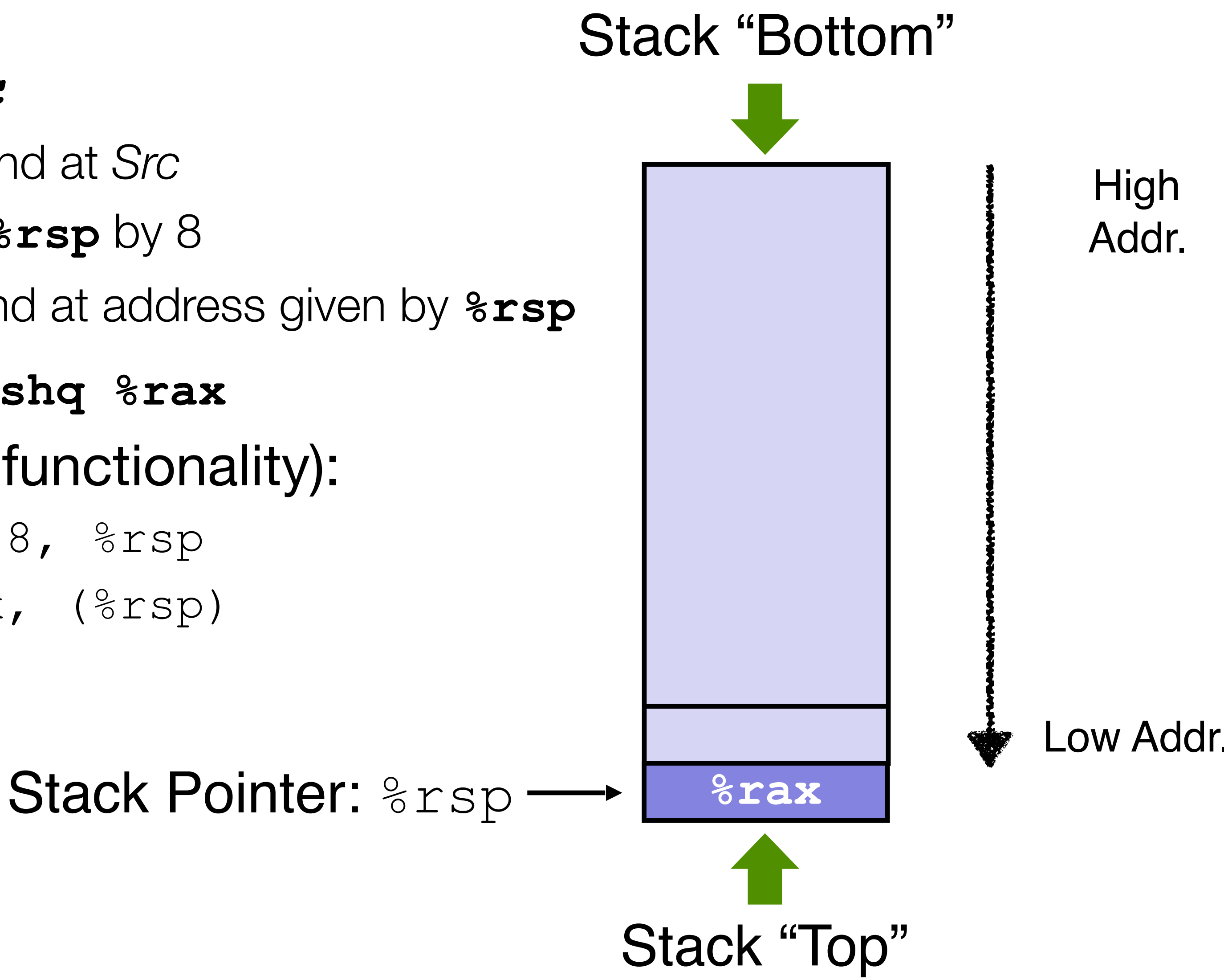
x86-64 Stack: Push

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



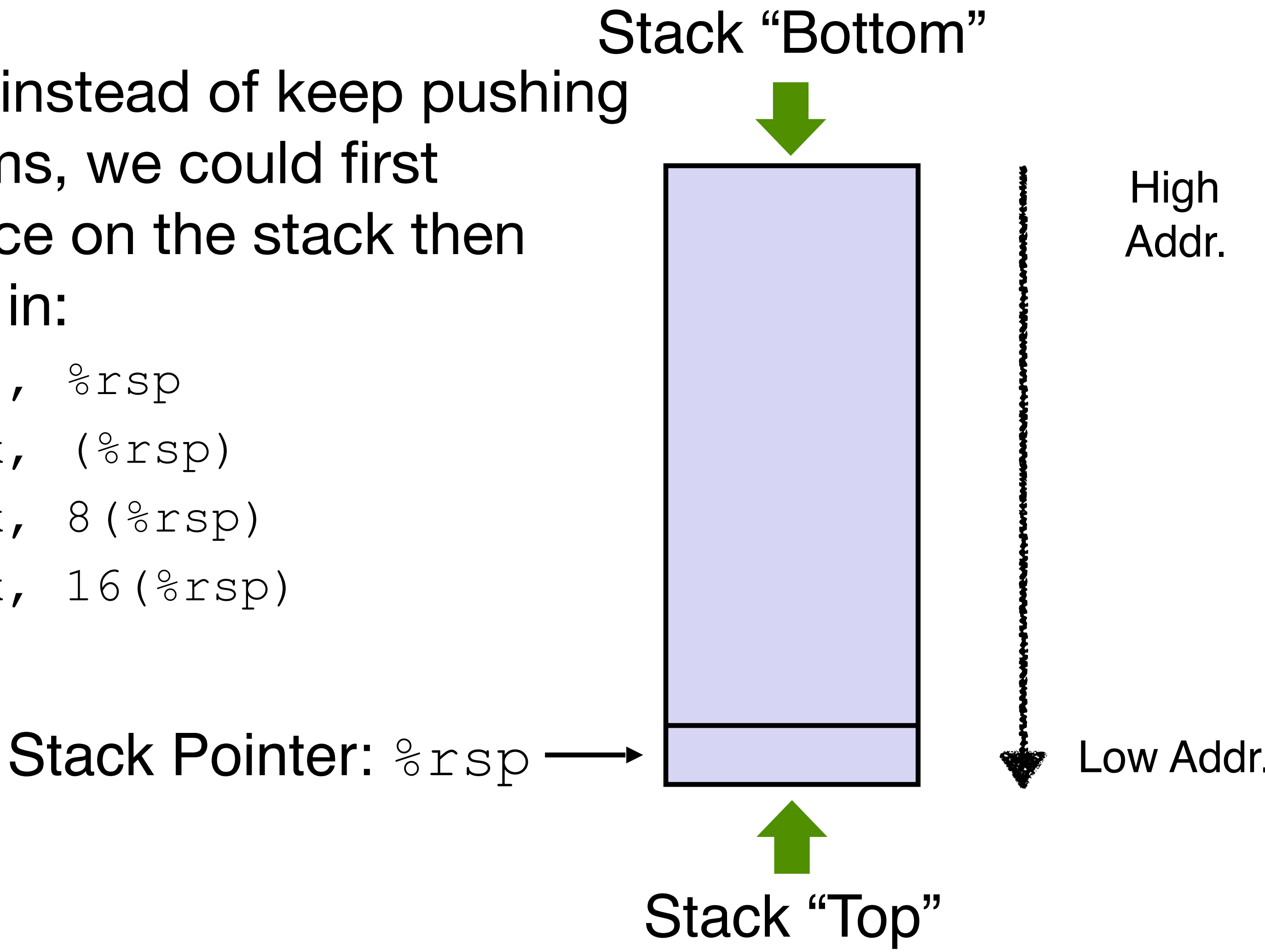
x86-64 Stack: Push

- **`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)`



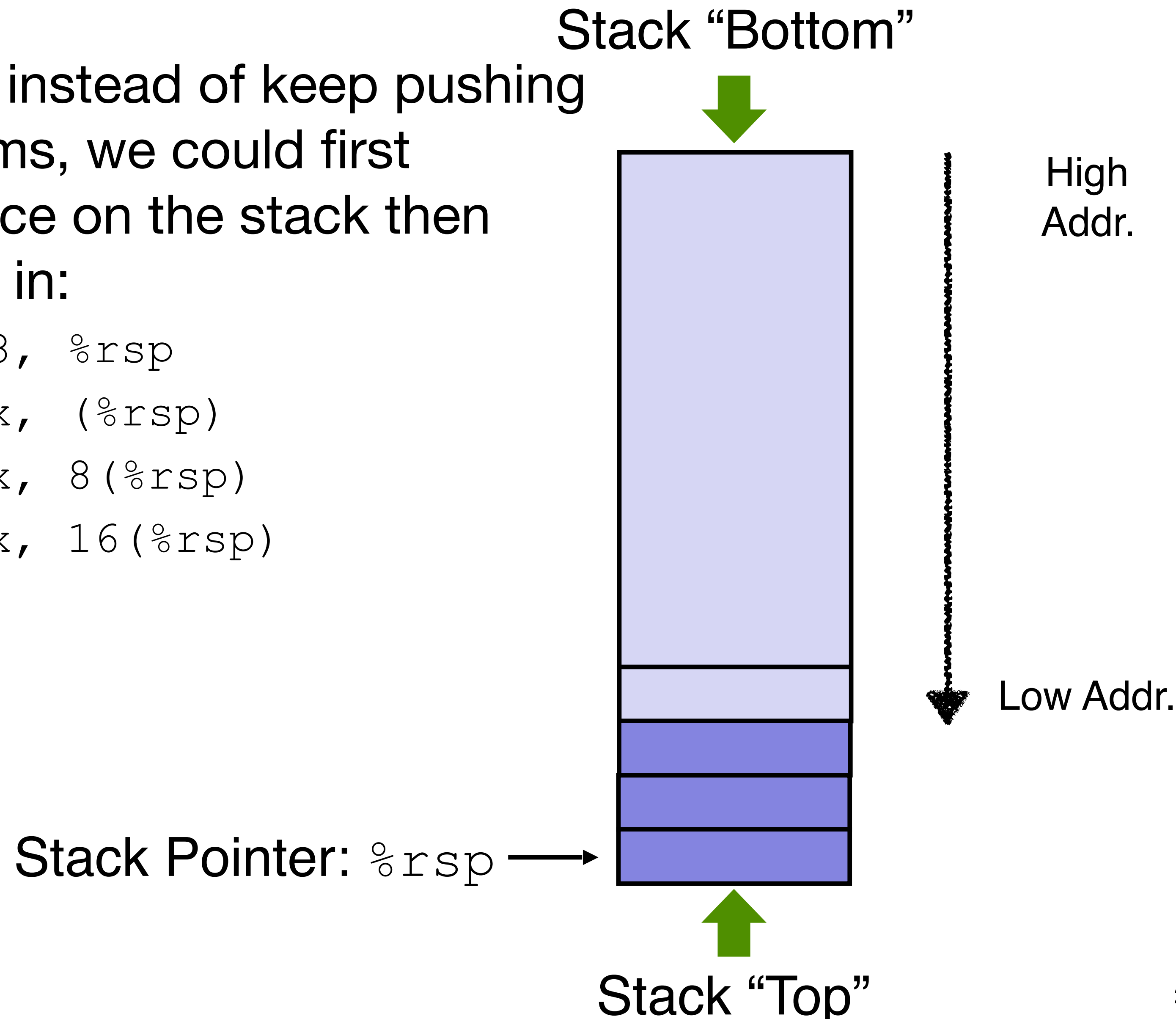
x86-64 Stack: Push

- 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)`



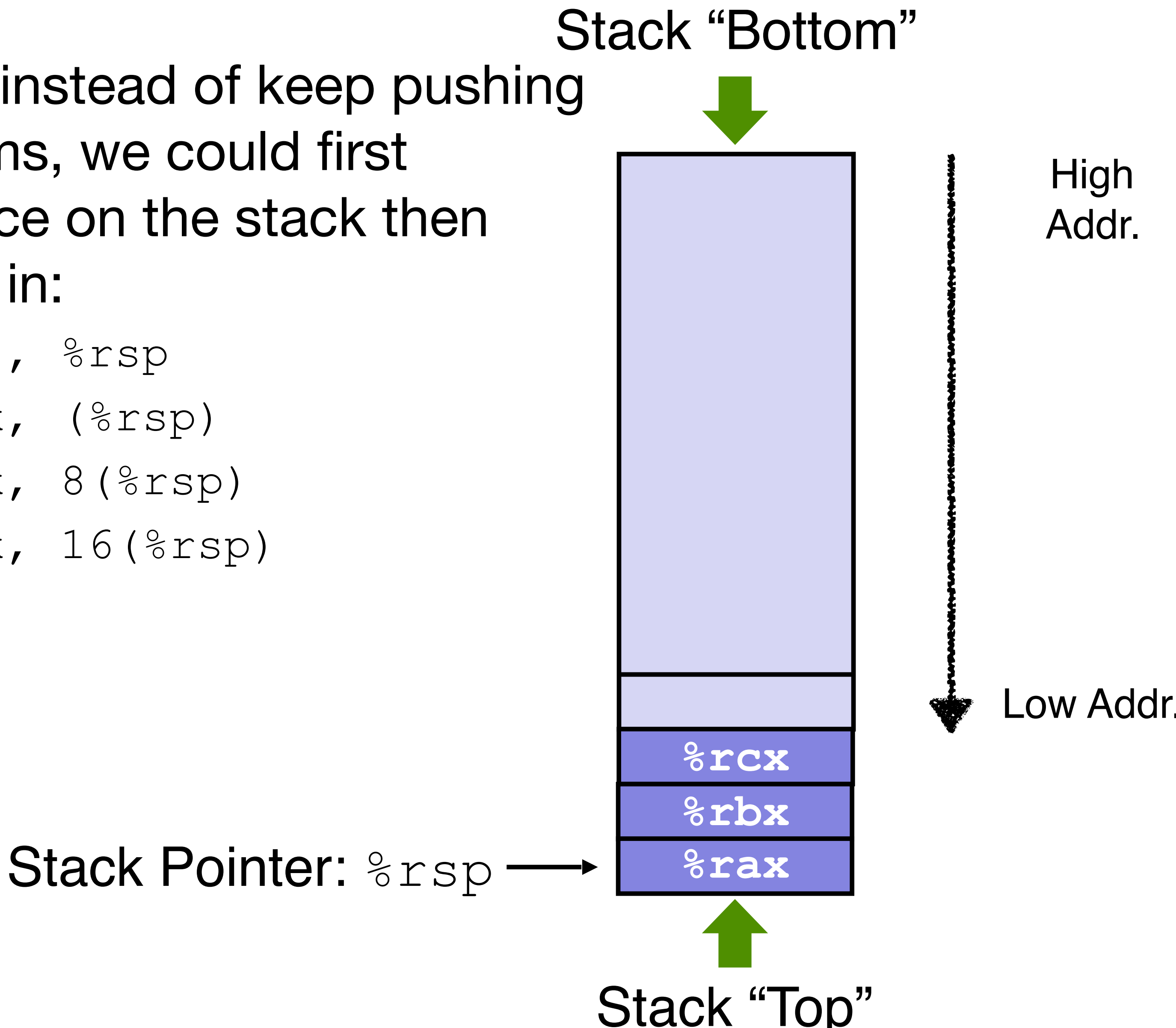
x86-64 Stack: Push

- 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)`



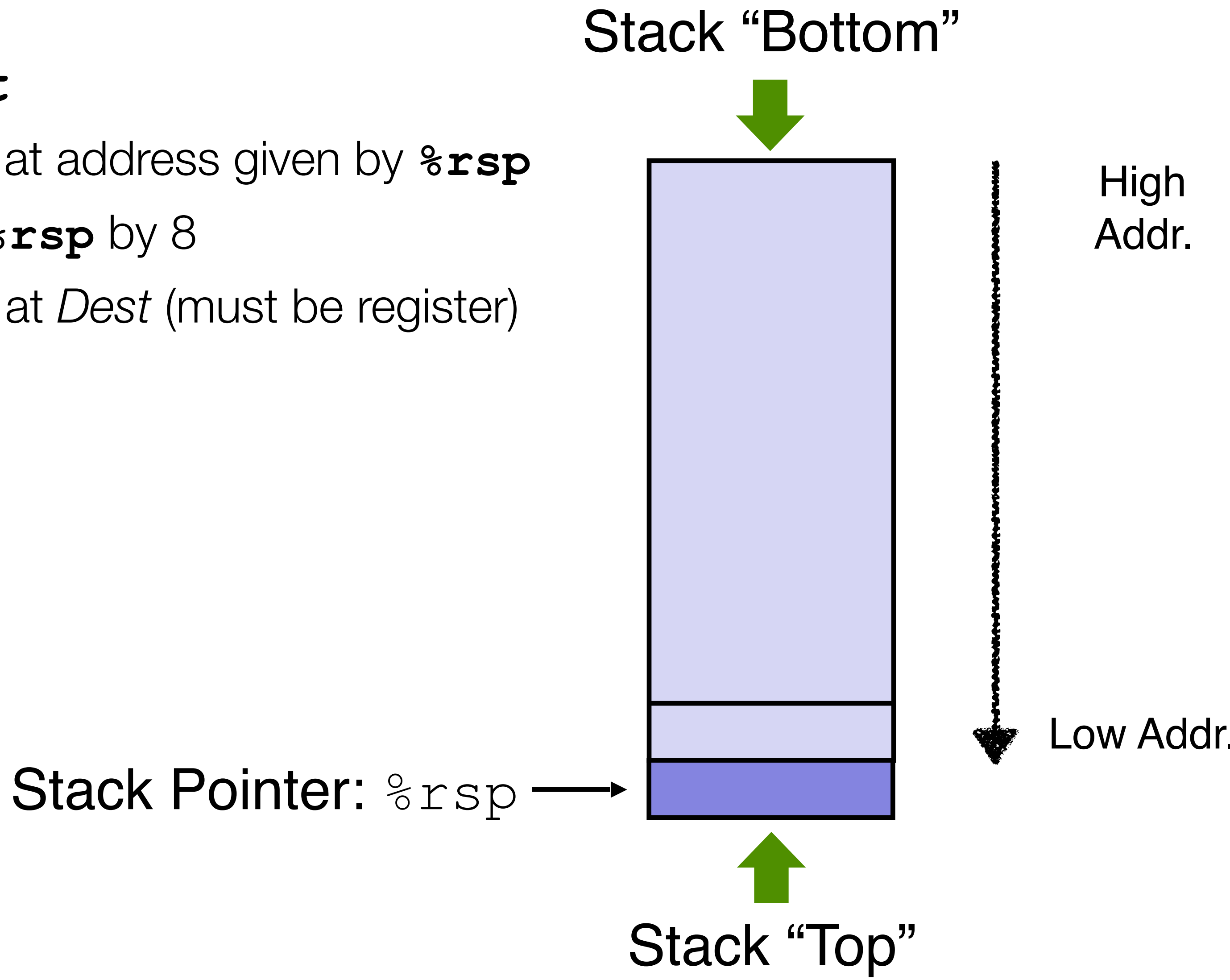
x86-64 Stack: Push

- 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)`



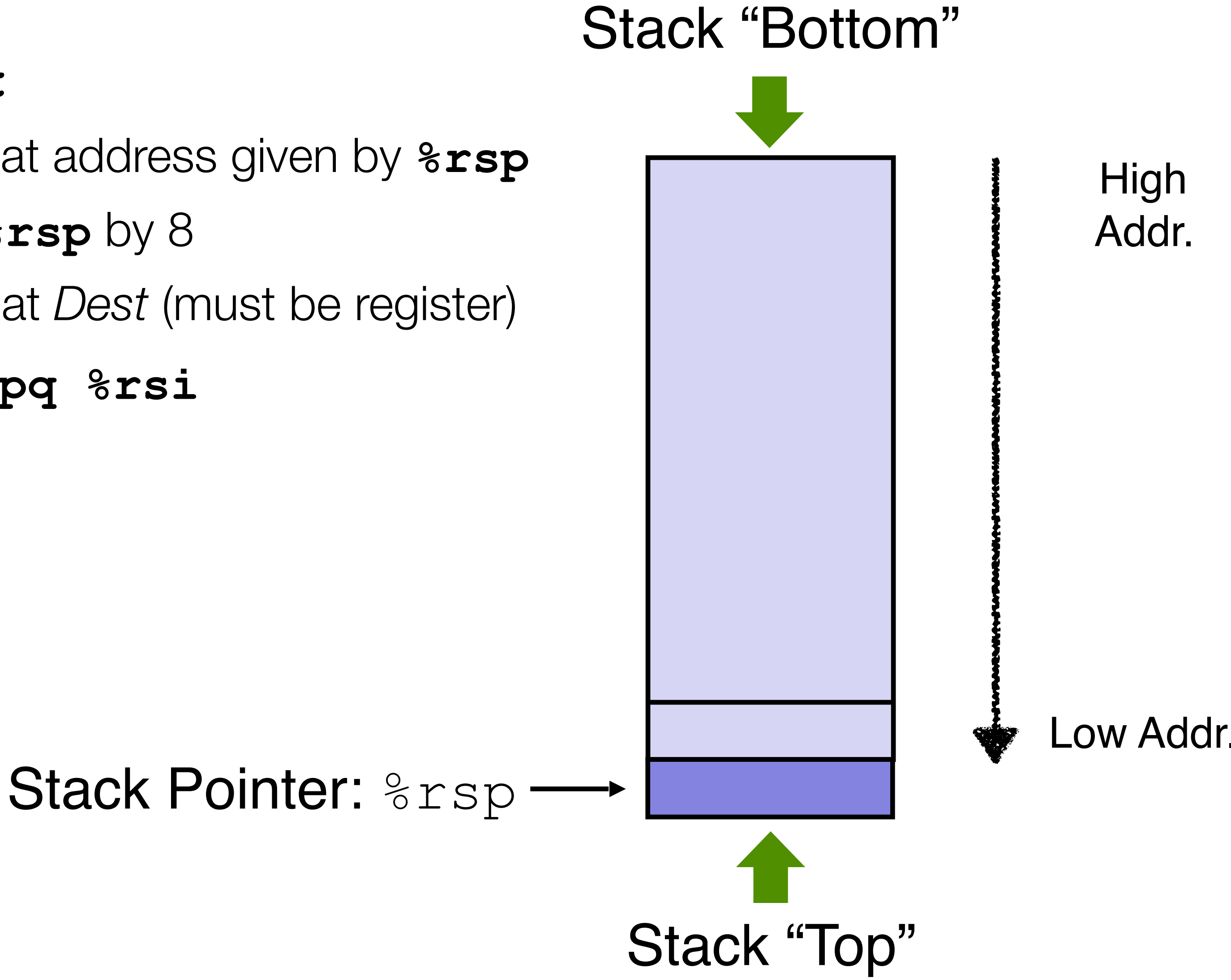
x86-64 Stack: Pop

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



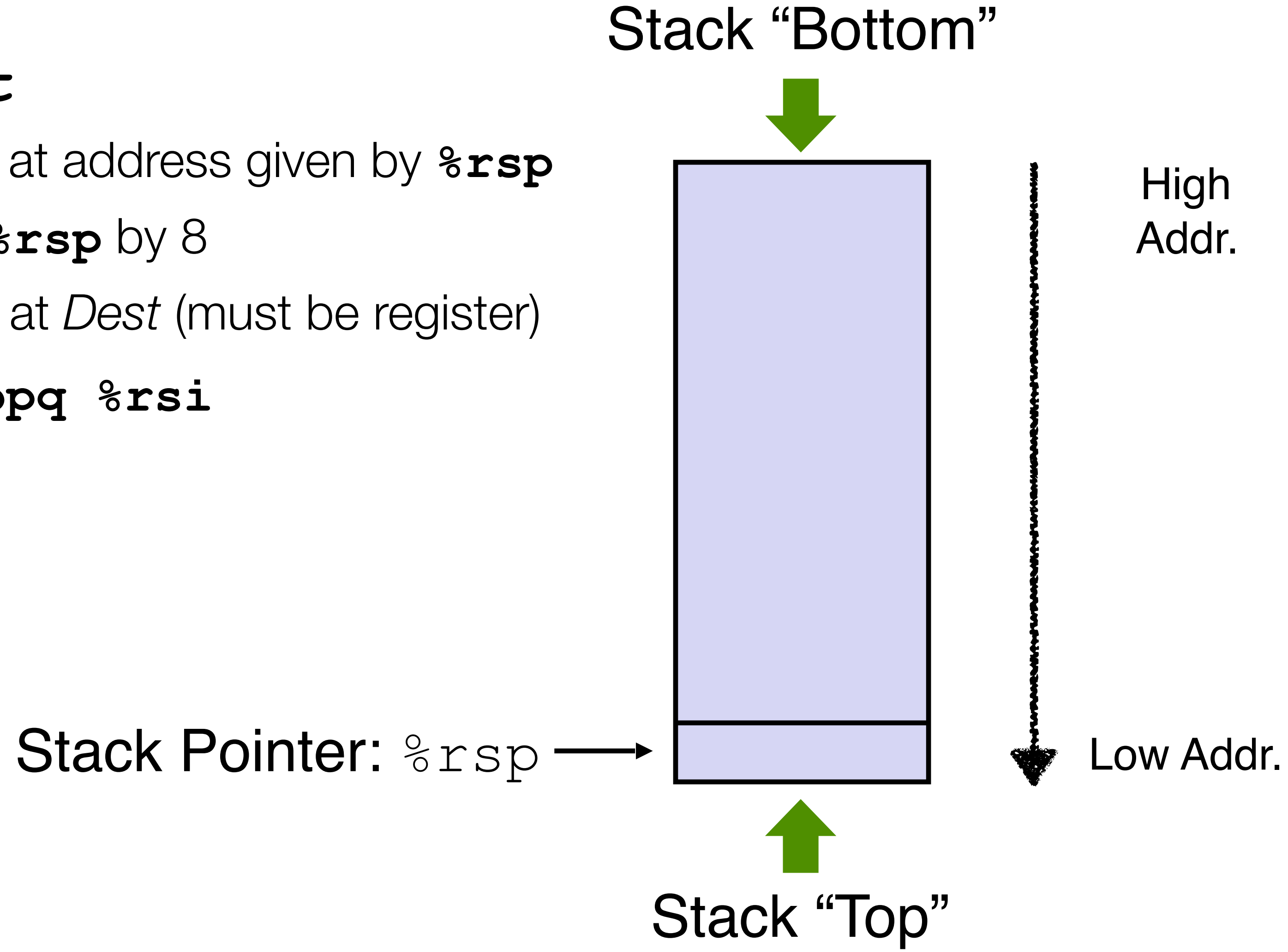
x86-64 Stack: Pop

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



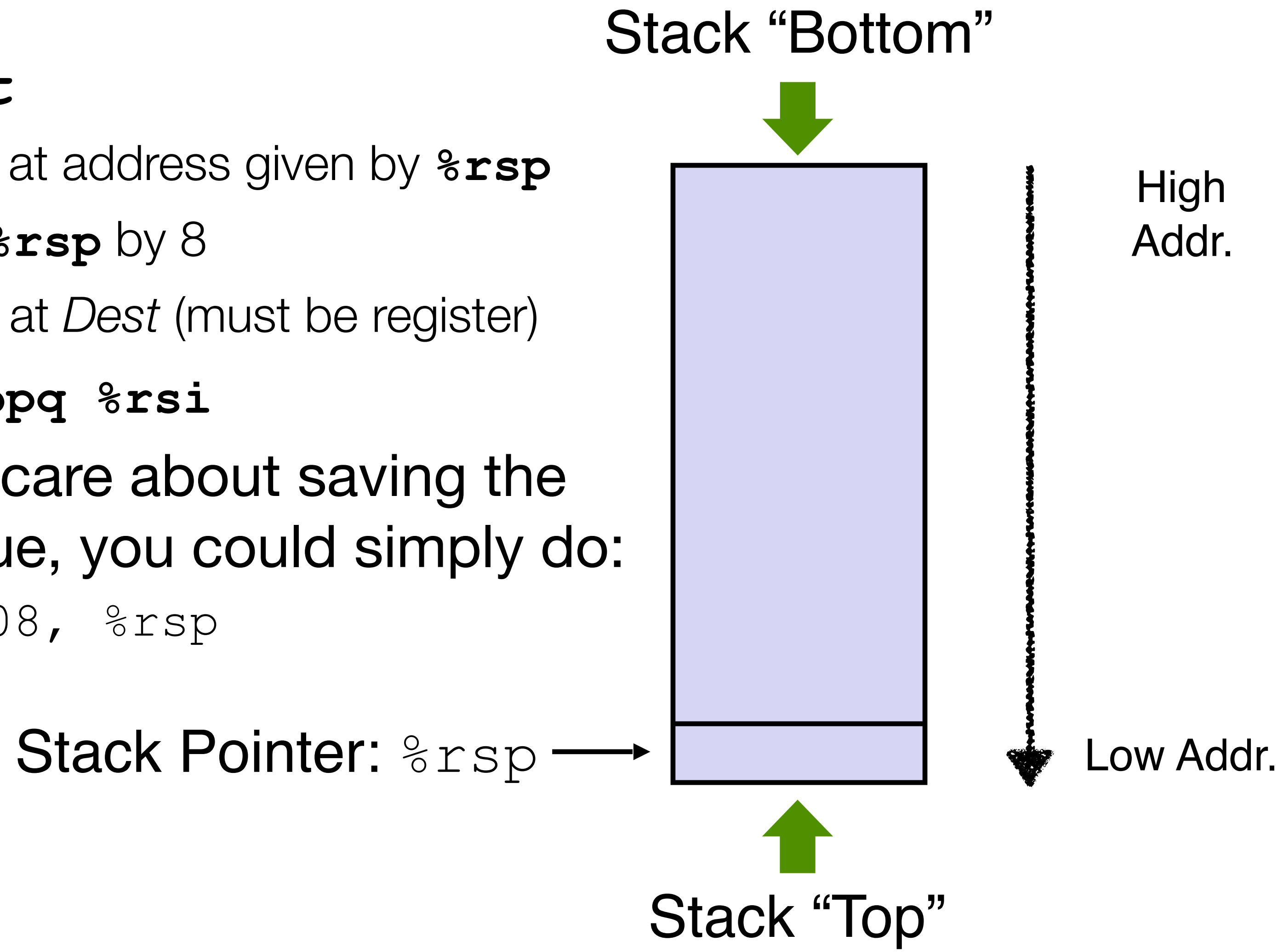
x86-64 Stack: Pop

- **`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 `jmq .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 `jmq .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 `jmq .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

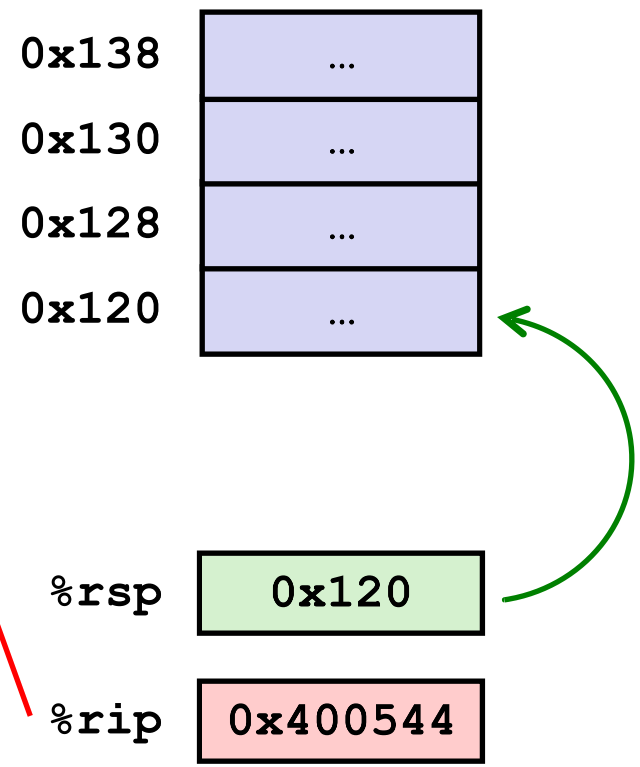
```
400540 <multstore>:
  400540: push    %rbx
  400541: mov     %rdx,%rbx
  400544: callq   400550 <mult2>
  400549: mov     %rax,(%rbx)
  40054c: pop     %rbx
  40054d: retq

…

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
```

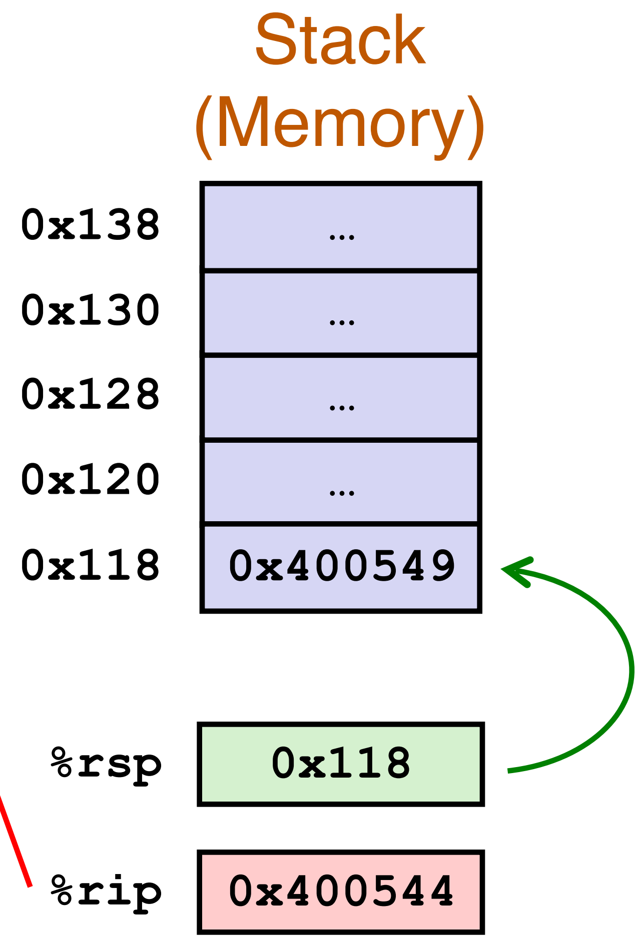
Stack
(Memory)



Function Call Example

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

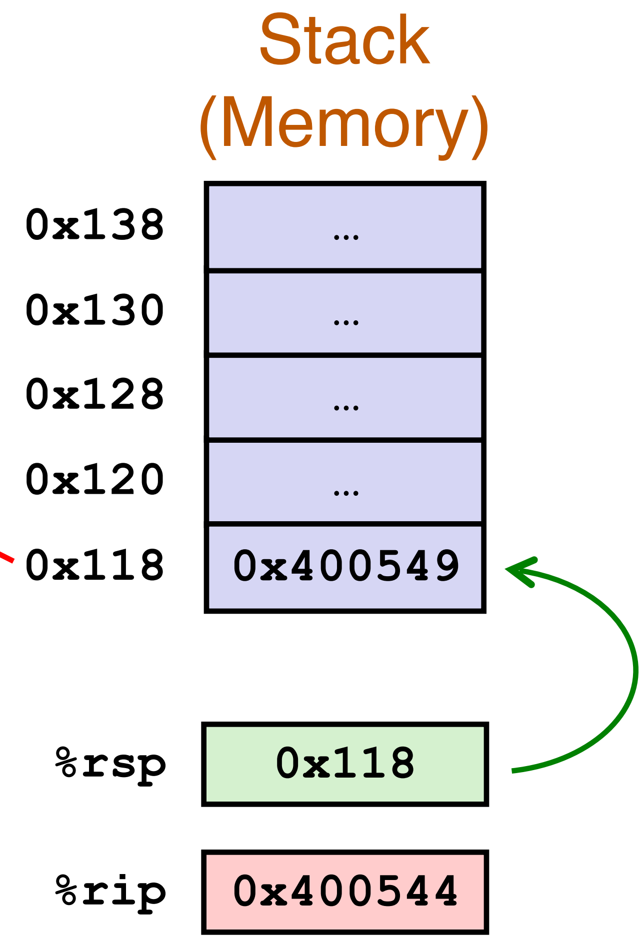
```
400550 <mult2>:
  400550:   mov     %rdi,%rax
…
…
  400557:   retq
```



Function Call Example

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

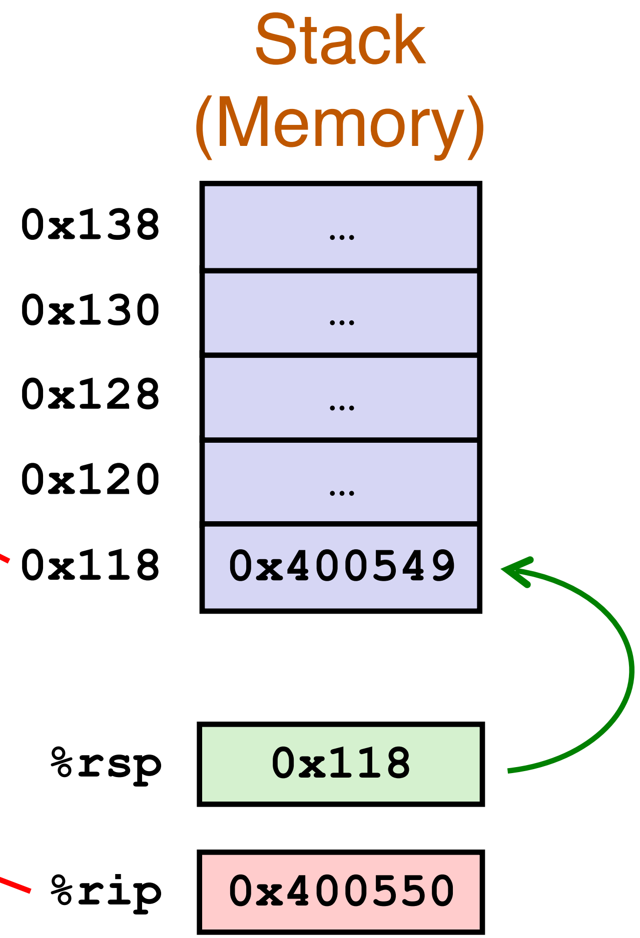
```
400550 <mult2>:
  400550:  mov     %rdi,%rax
…
…
  400557:  retq
```



Function Call Example

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

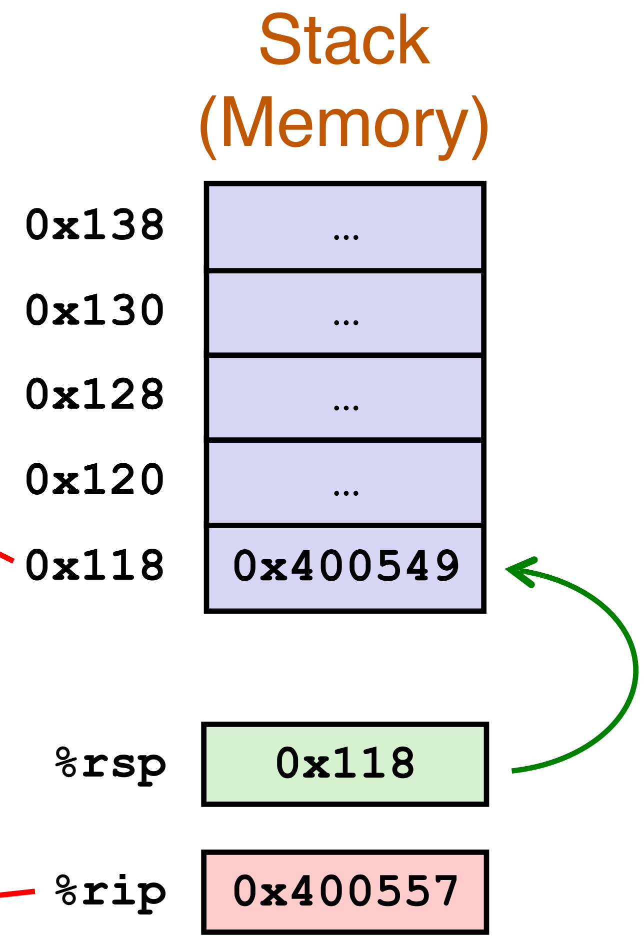
```
400550 <mult2>:
  400550:  mov    %rdi,%rax
…
…
  400557:  retq
```



Function Call Example

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

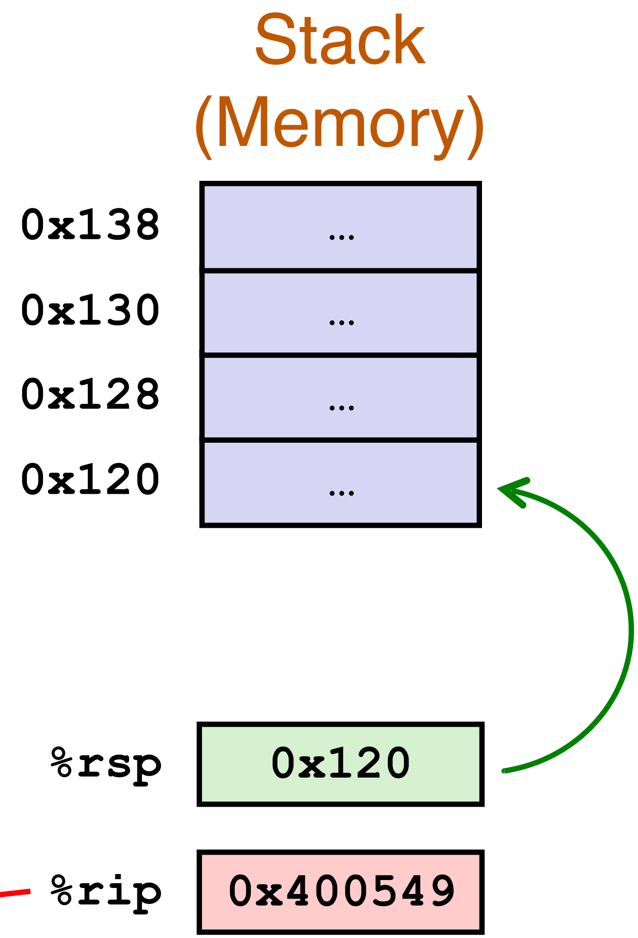
```
400550 <mult2>:
  400550:   mov     %rdi,%rax
…
…
  400557:   retq
```



Function Call Example

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

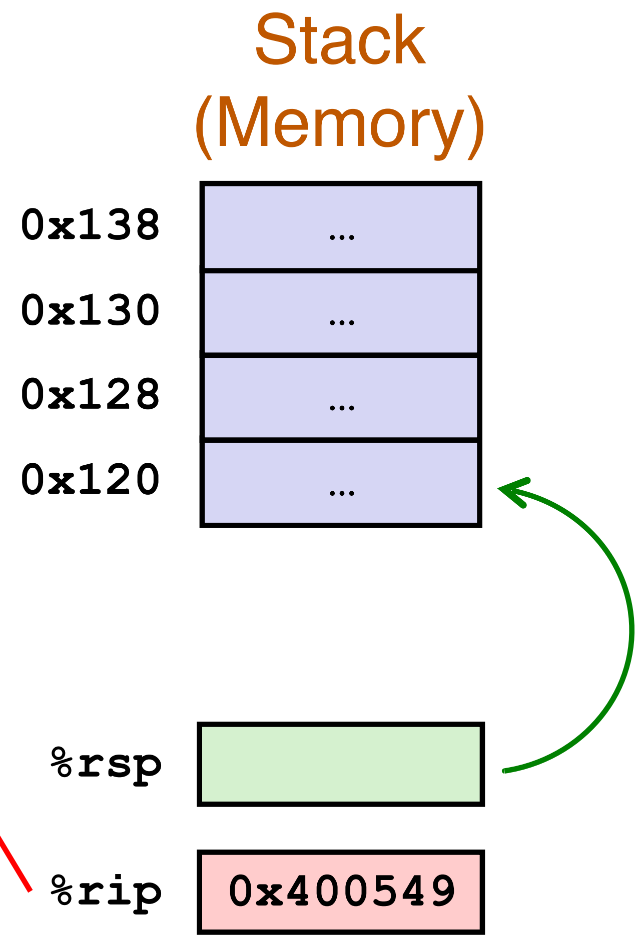
```
400550 <mult2>:
  400550:  mov    %rdi,%rax
…
…
  400557:  retq
```



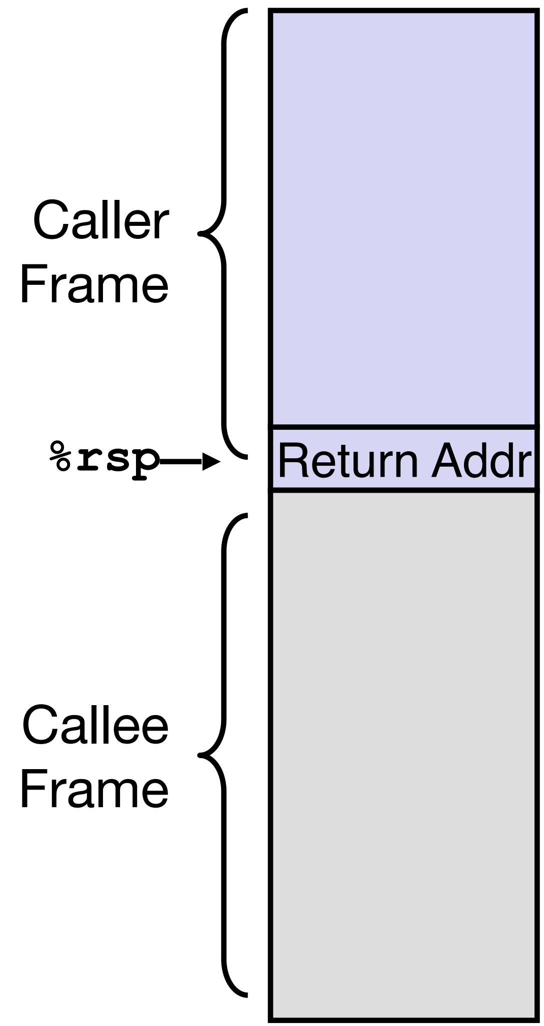
Function Call Example

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

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



Stack Frame (So Far…)



Today: How to Implement Function Call

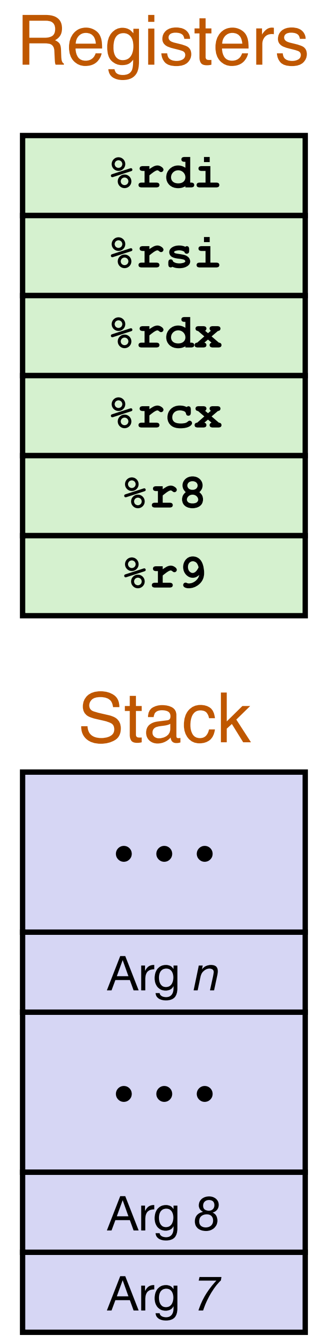
- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

Passing Function Arguments

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

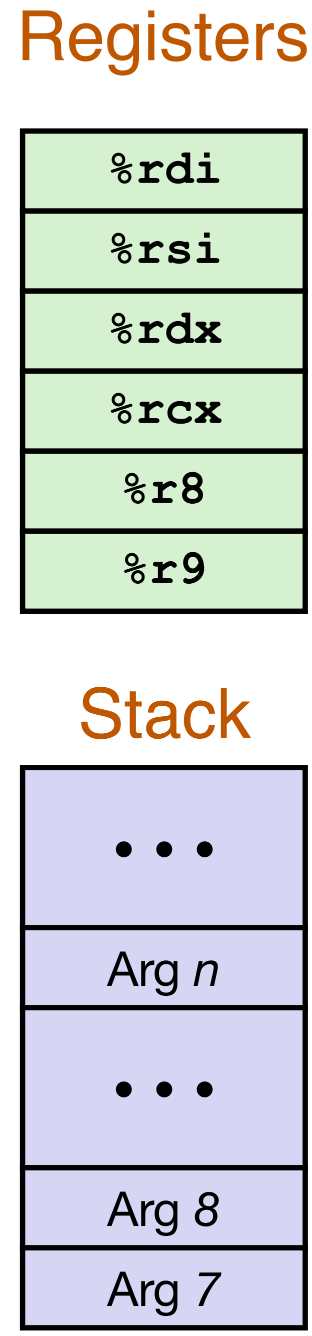
Passing Function Arguments

- 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`



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



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;
}
```

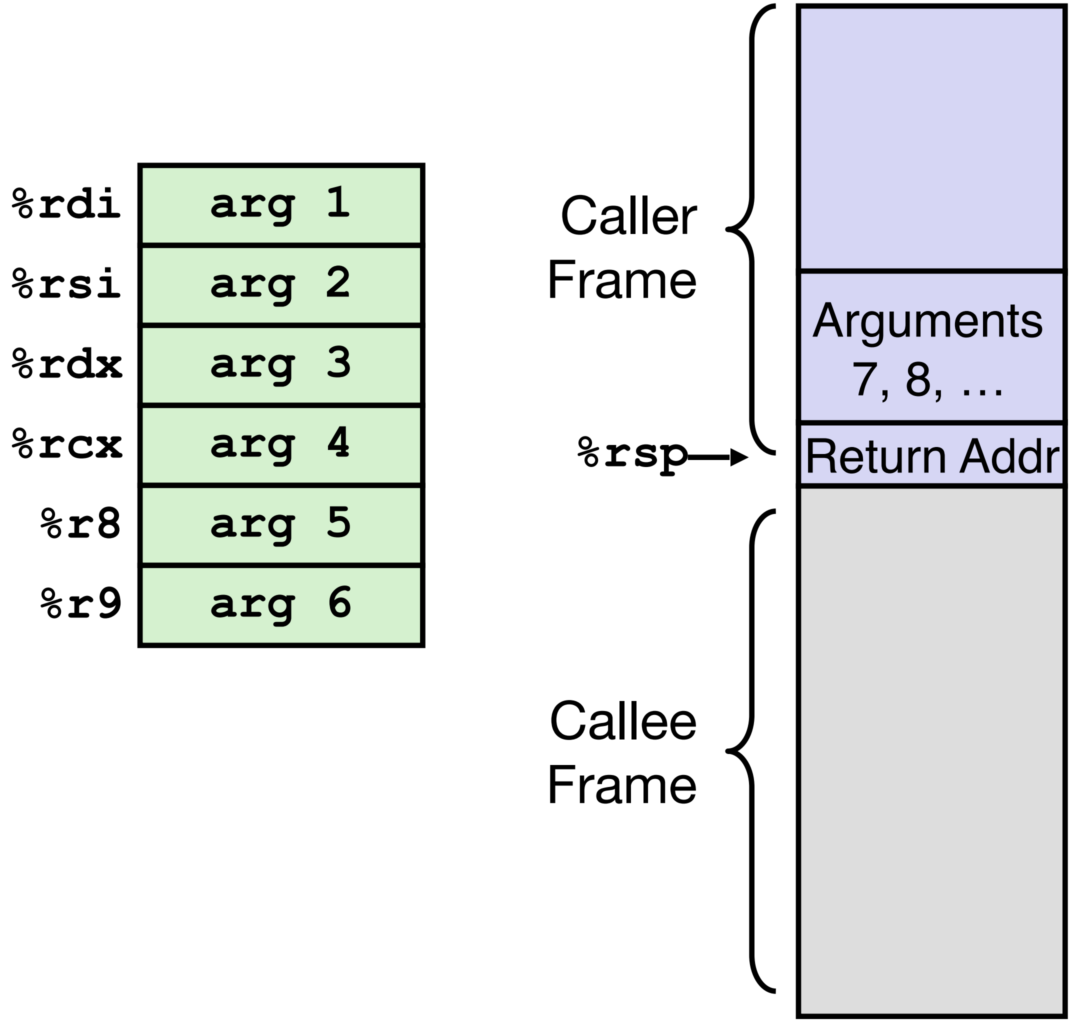
Function Call Data Flow Example

%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. Take 255/455.

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

Register Example: `incr`

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

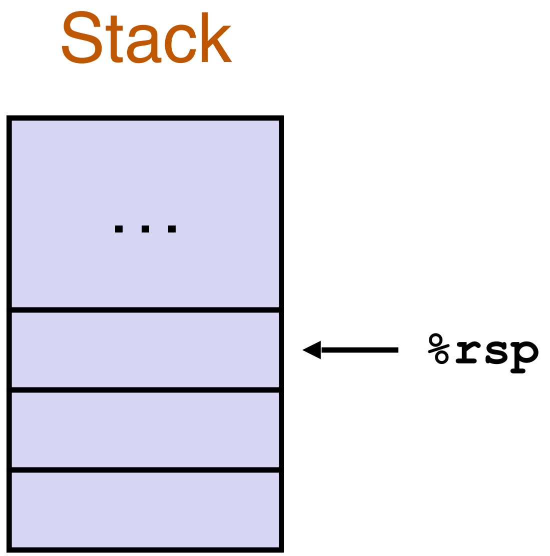
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
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 v2+v3;
}
```

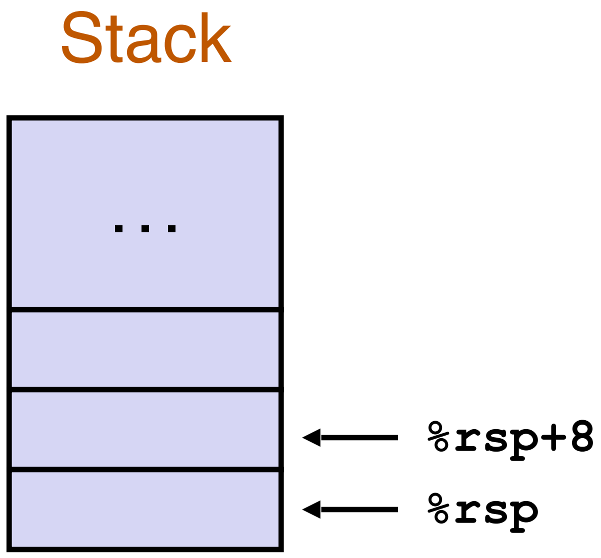
```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Example: `call_add`

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

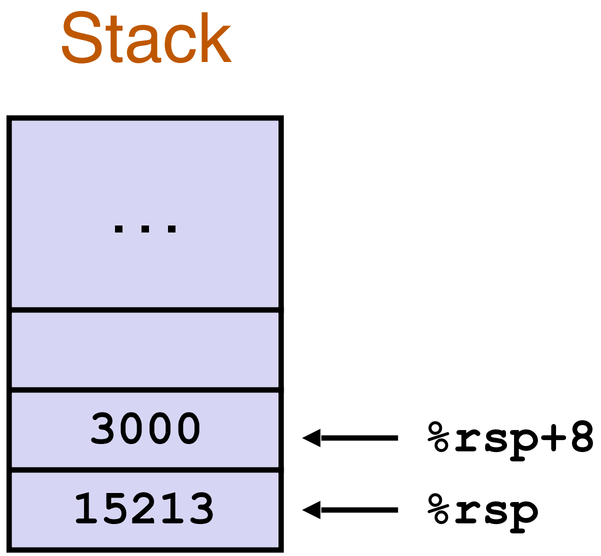
```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Example: `call_add`

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

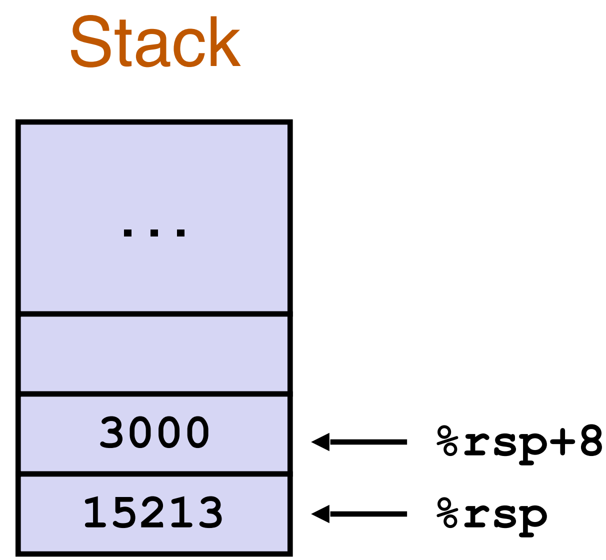
```
call_add:
  subq    $16, %rsp
  movq    $15213, (%rsp)
  movq    $3000, 8(%rsp)
  leaq    (%rsp), %rdi
  leaq    8(%rsp), %rsi
  call    add
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Stack Example: `call_add`

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

```
call_add:
  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
```



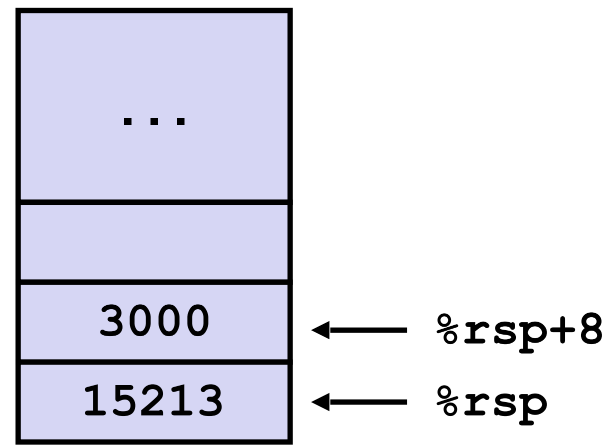
Register	Value(s)
%rdi	&v1
%rsi	&v2

Stack Example: `call_add`

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

```
call_add:
  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



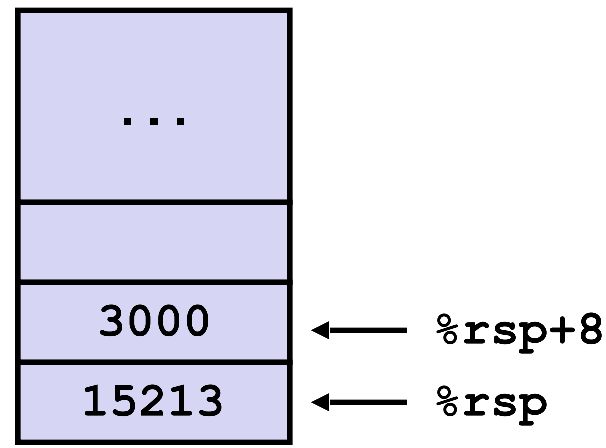
Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	18213

Stack Example: `call_add`

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

```
call_add:
  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



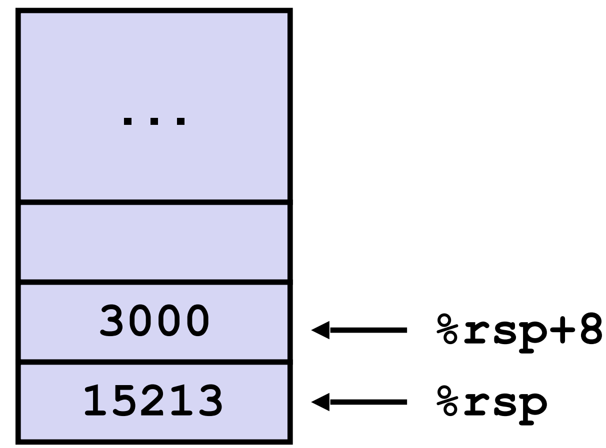
Register	Value(s)
%rdi	&v1
%rsi	&v2
%rax	21213

Stack Example: `call_add`

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

```
call_add:
  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
```

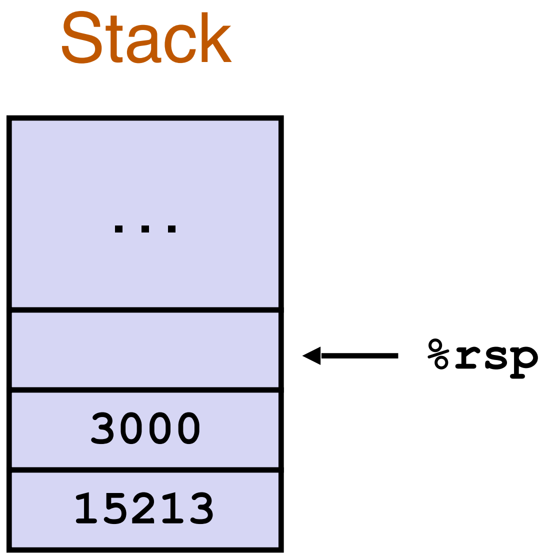
Stack



Stack Example: `call_add`

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

```
call_add:
  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 Frame (So Far…)