

CSC 252/452: Computer Organization

Fall 2025: Lecture 8

Instructor: Yanan Guo

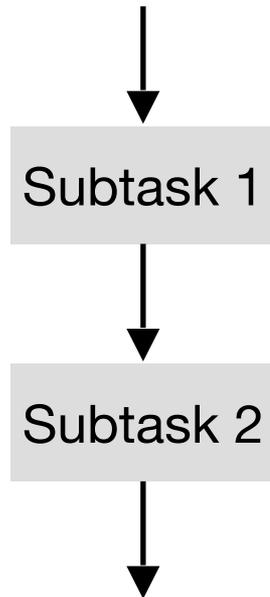
Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 2 is out
 - Details: <https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment2.html>
 - Due on **Oct. 1st**, 11:59 PM
 - You (may still) have 3 slip days

Summary

Sequential



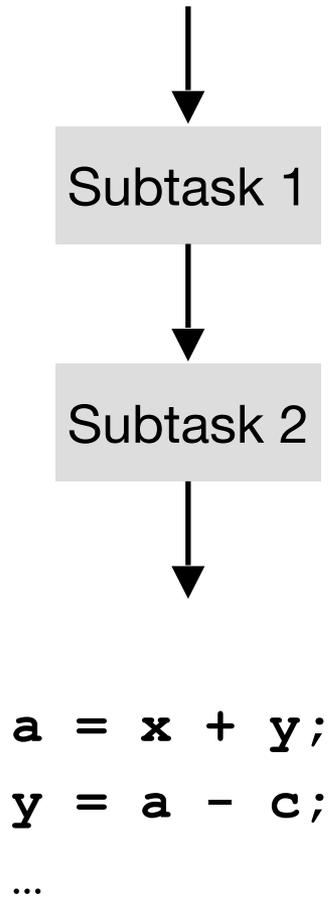
```
a = x + y;
```

```
y = a - c;
```

```
...
```

Summary

Sequential

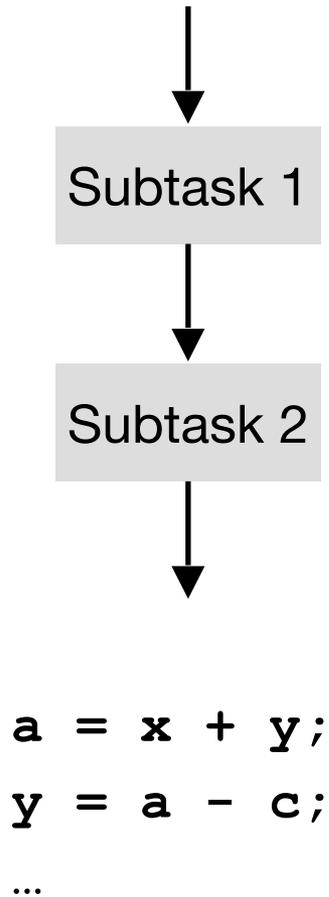


Memory

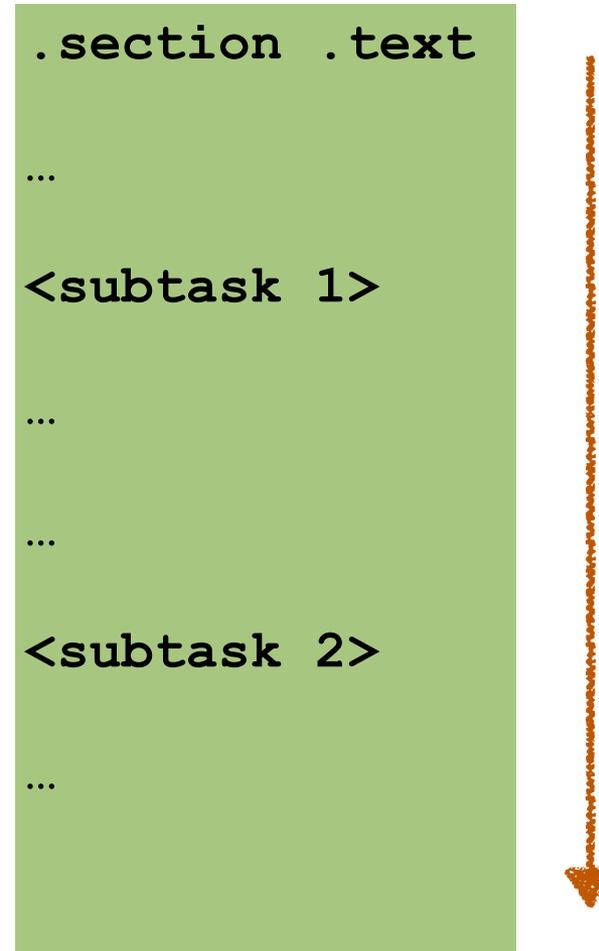
```
.section .text  
...  
<subtask 1>  
...  
...  
<subtask 2>  
...
```

Summary

Sequential

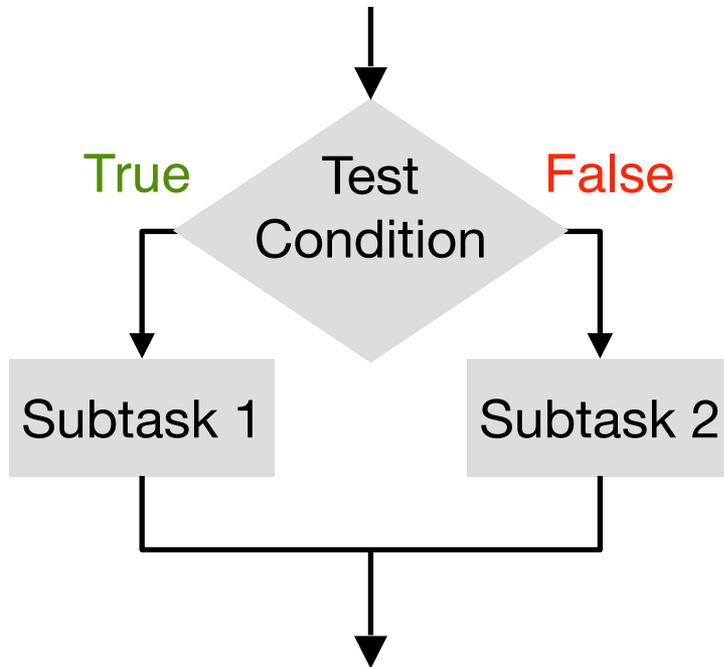


Memory



Summary

Conditional



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

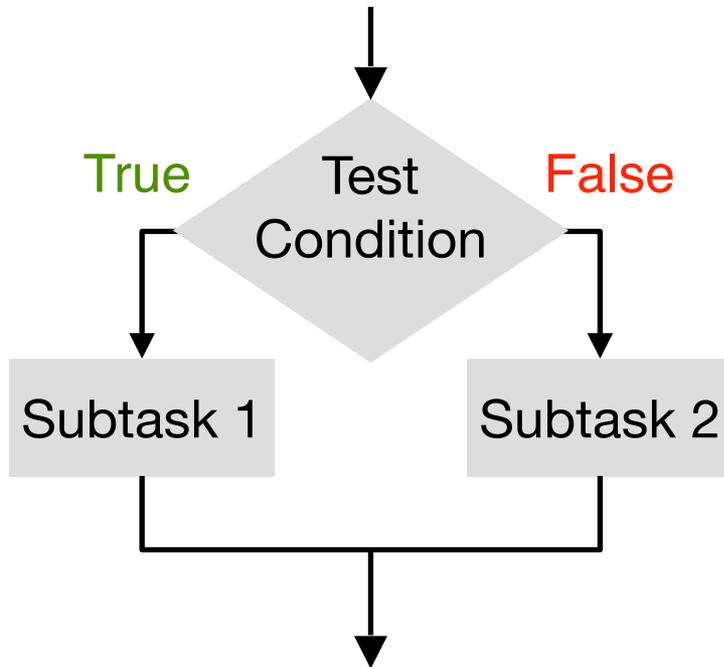
Direct Jump Instructions

- Jump to different part of code (designated by a label) depending on condition codes

Instruction	Jump Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jae	$\sim CF$	Above or Equal (unsigned)
jb	CF	Below (unsigned)
jbe	$CF \ \ ZF$	Below or Equal (unsigned)

Summary

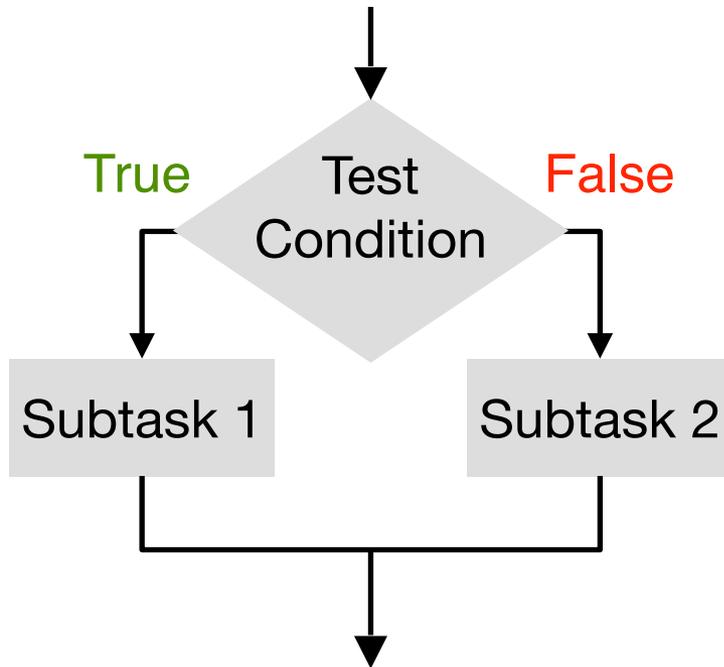
Conditional



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

Summary

Conditional



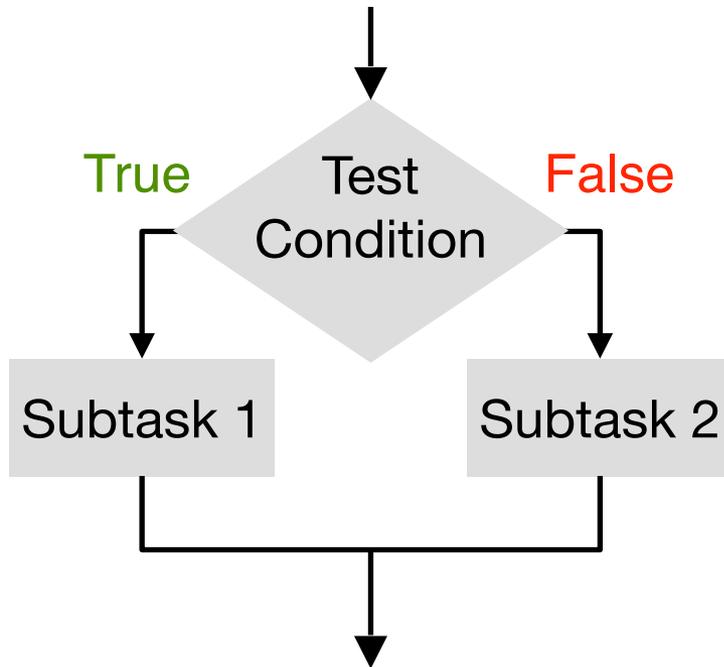
```
if (x > y) r = x - y;  
else r = y - x;
```

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;
```

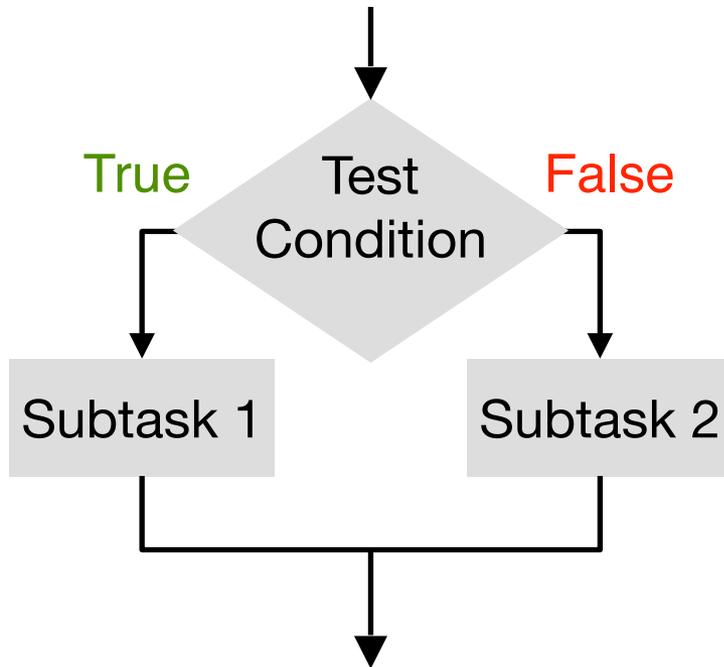
Memory

```
.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;
```

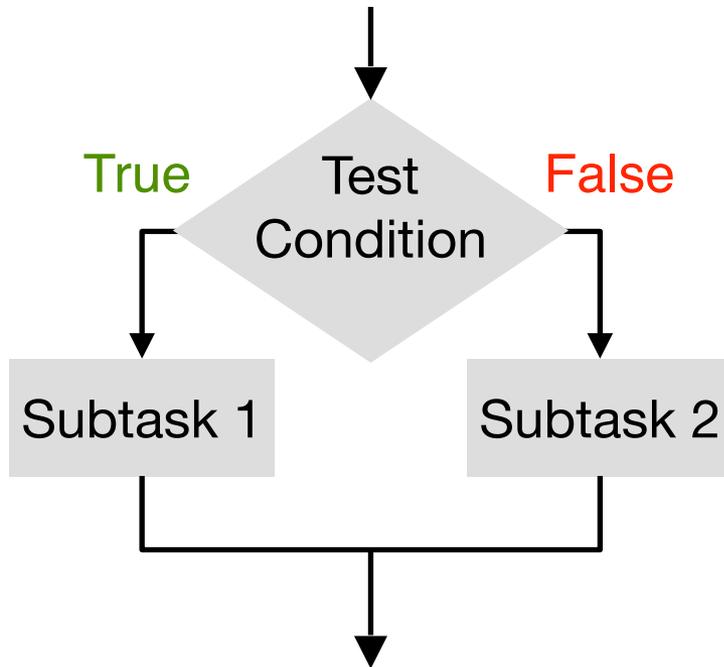
Memory

```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```



Summary

Conditional



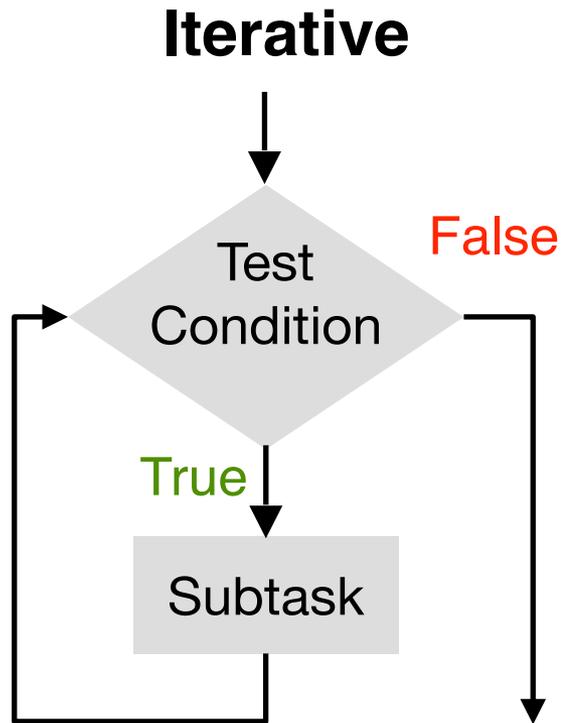
```
if (x > y) r = x - y;  
else r = y - x;
```

Memory

```
.section .text  
  
...  
cmpq  
jle .L2  
.L1 <subtask 1>  
  
...  
  
jmp .done  
.L2 <subtask 2>  
  
...  
.done
```



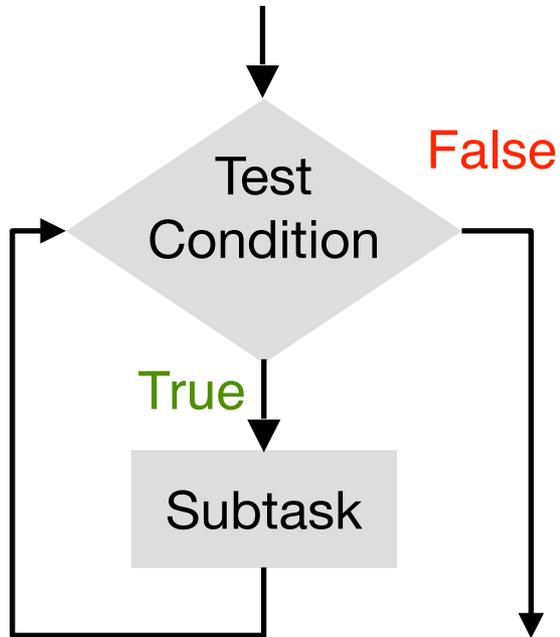
Summary



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

Summary

Iterative



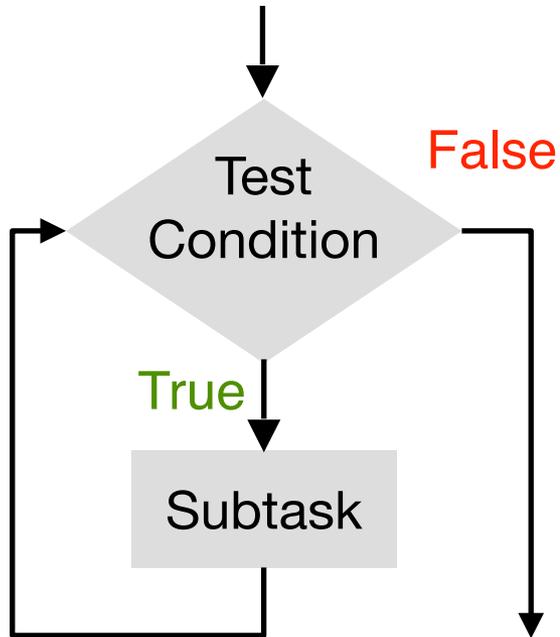
```
while (x > 0) {  
    x--;  
}
```

Memory

```
.section .text  
  
...  
addq  
jmp .L2  
.L1:  
  
...  
    <subtask>  
  
...  
.L2:  
    cmpq A, B  
    jg .L1  
  
...  
...
```

Summary

Iterative



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

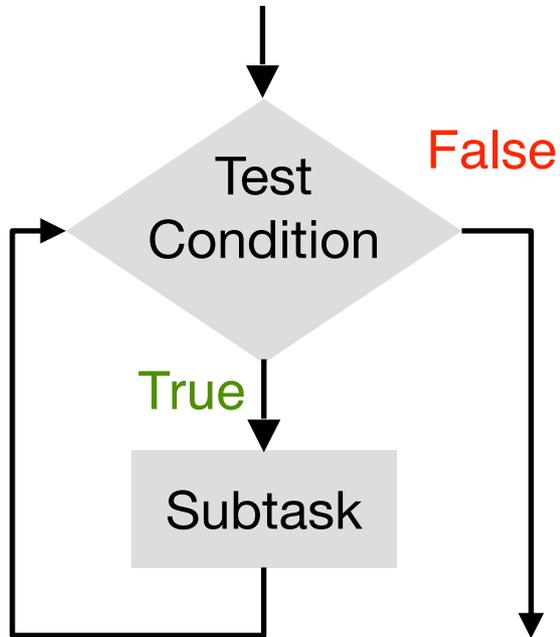
Memory

```
.section .text  
  
...  
addq  
jmp .L2  
.L1:  
  
...  
    <subtask>  
  
...  
.L2:  
    cmpq A, B  
    jg .L1  
  
...  
...
```

An orange arrow points downwards from the right side of the assembly code block, indicating the flow of execution from the assembly level to the iterative logic shown in the diagram.

Summary

Iterative



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

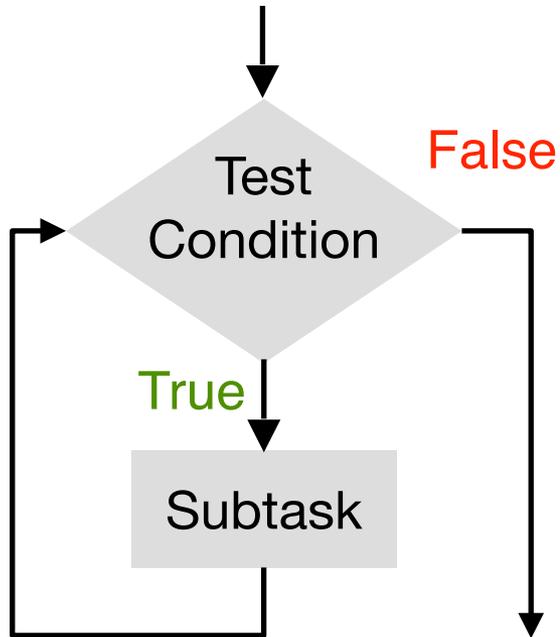
Memory

```
.section .text  
  
...  
addq  
jmp .L2  
.L1:  
  
...  
    <subtask>  
  
...  
.L2:  
    cmpq A, B  
    jg .L1  
  
...  
...
```

A vertical orange arrow with a dotted texture points downwards, starting from the top of the assembly code and ending at the bottom, indicating the sequential flow of execution through the code blocks.

Summary

Iterative



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

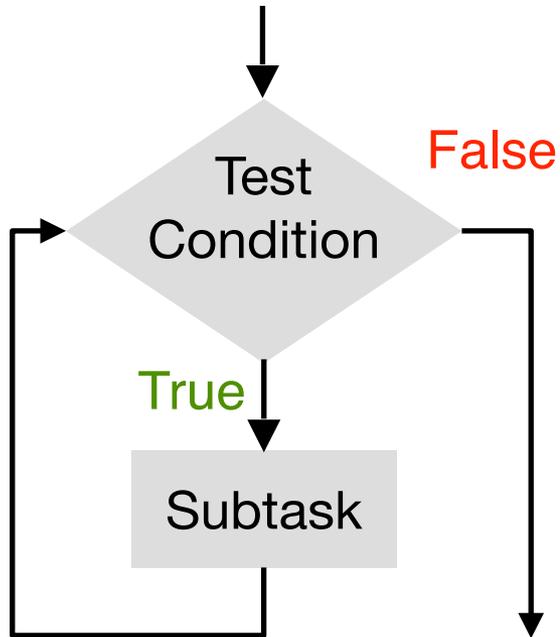
Memory

```
.section .text  
  
...  
addq  
jmp .L2  
.L1:  
  
...  
    <subtask>  
  
...  
.L2:  
    cmpq A, B  
    jg .L1  
  
...  
...
```



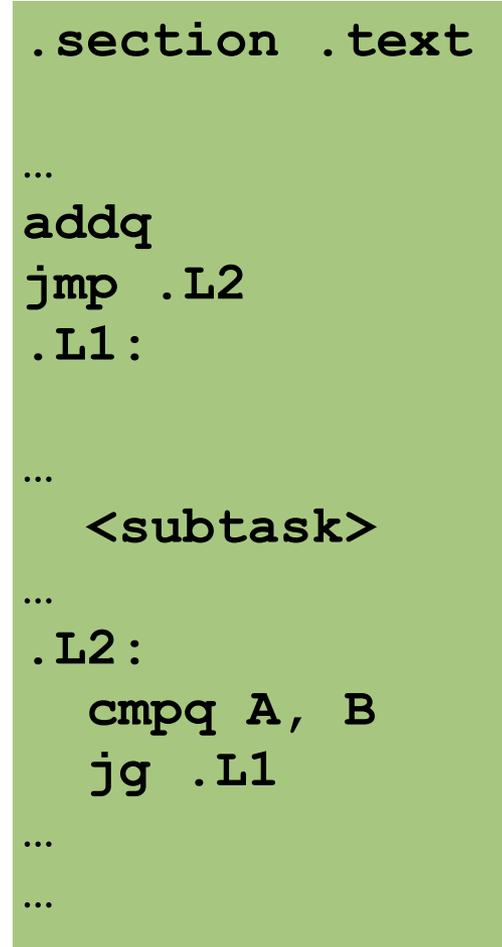
Summary

Iterative



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

Memory



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

•
•
•

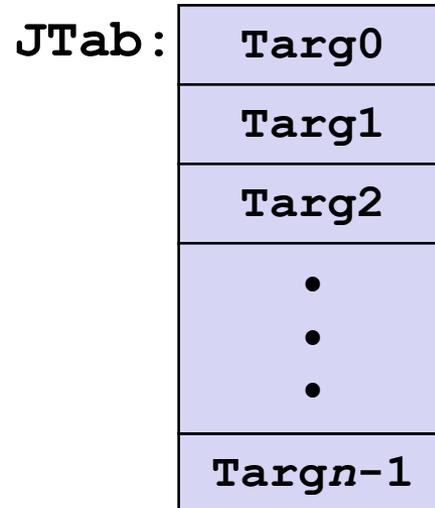
Targ $n-1$: Code Block
 $n-1$

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



Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

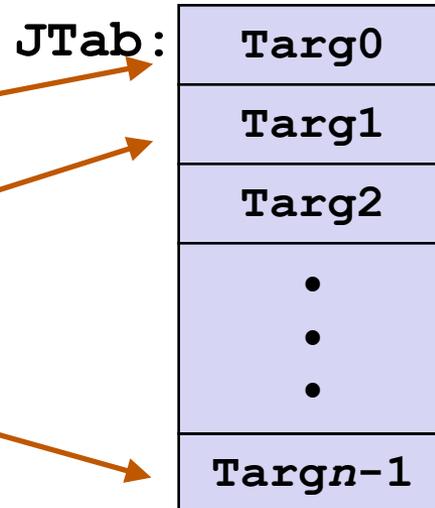
Targn-1: Code Block n-1

Implementing Switch Using Jump Table

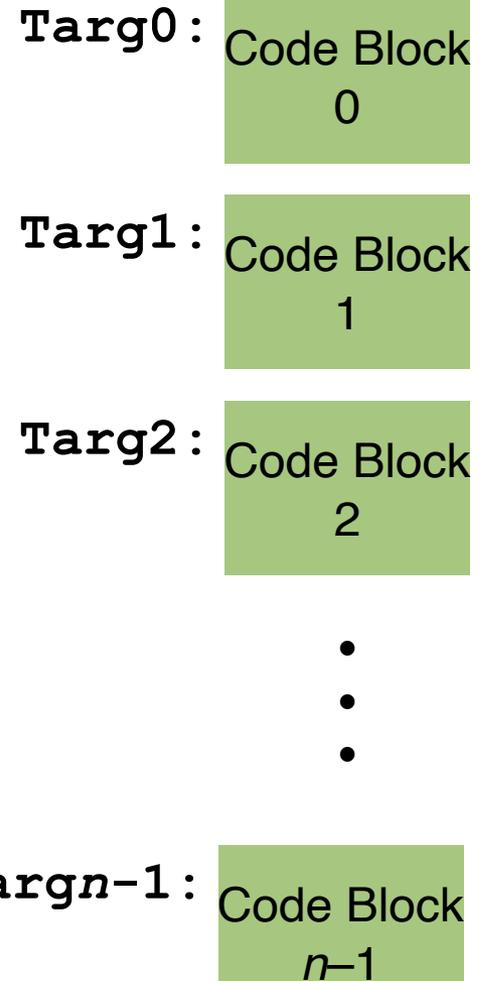
Switch Form

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

Jump Table



Jump Targets

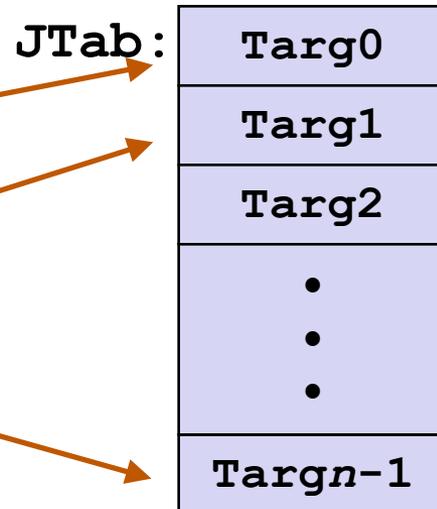


Implementing Switch Using Jump Table

Switch Form

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

Jump Table



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

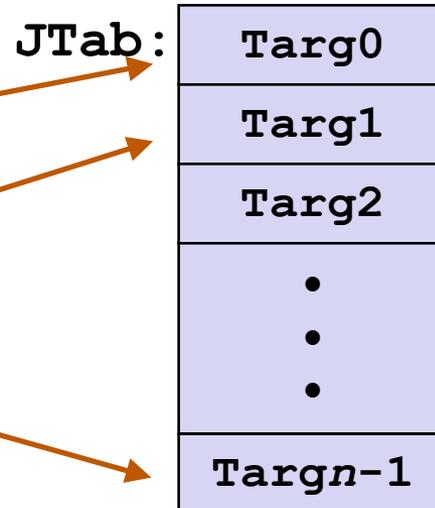
```
.L1: # Case 1
movq %rsi, %rax
imulq %rdx, %rax
jmp .done
.L2: # Case 2
movq %rsi, %rax
cqto
idivq %rcx
.L3: # Case 3
addq %rcx, %rax
jmp .done
.L5: # Case 5,6
subq %rdx, %rax
jmp .done
.LD: # Default
movl $2, %eax
jmp .done
```

Implementing Switch Using Jump Table

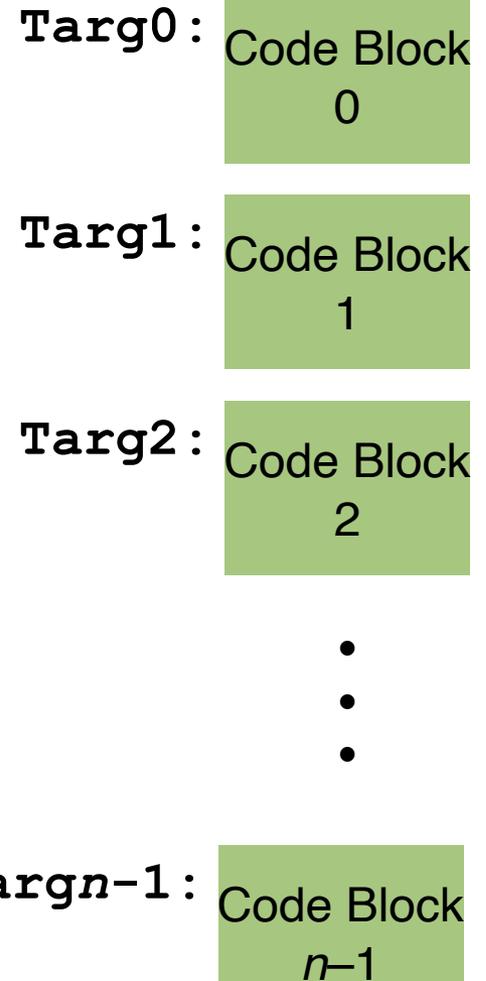
Switch Form

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

Jump Table



Jump Targets

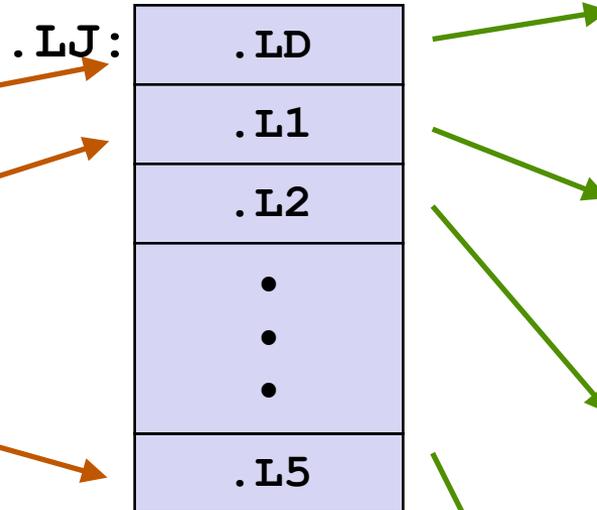


Implementing Switch Using Jump Table

Switch Form

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

Jump Table



Jump Targets

.LD: Code Block 0

.L1: Code Block 1

.L2: Code Block 2

•
•
•

.L5: 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

.LJ:	.LD
	.L1
	.L2
	• • •
	.L5

Jump Targets

.LD: Code Block 0

.L1: Code Block 1

.L2: Code Block 2

•
•
•

.L5: 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
.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)
```

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

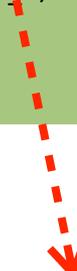
```
P(...) {  
...  
...  
    y = Q(x);  
    print(y)  
...  
}
```

```
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

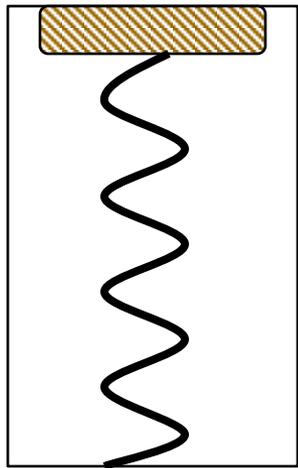
- Frame (Active Record)
 - A frame refers to a piece of memory that contains (almost) all the information needed to execute a function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory

General Idea

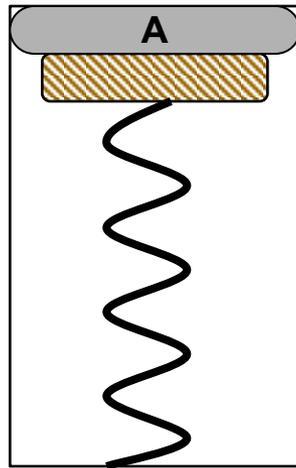
- Frame (Active Record)
 - A frame refers to a piece of memory that contains (almost) all the information needed to execute a function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory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

A Physical Stack: A Coin Holder

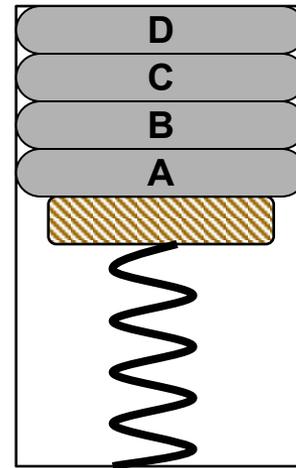
First quarter out is the last quarter in.



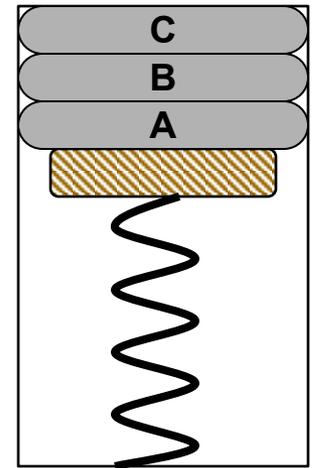
Initial State



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

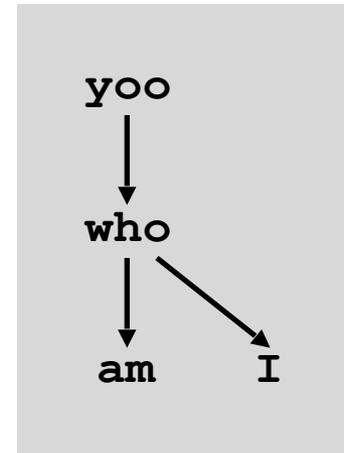
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  return  
}
```

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

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

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

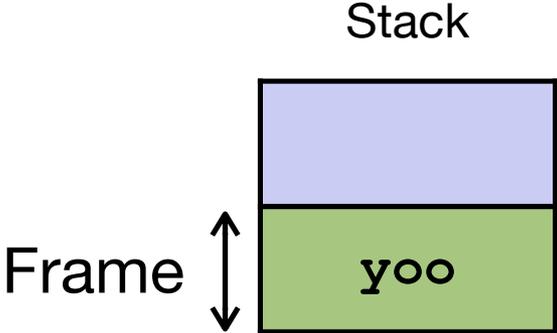
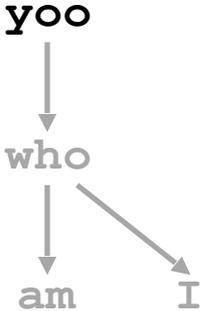
Example Call Chain



Run-Time Stack During Function Call

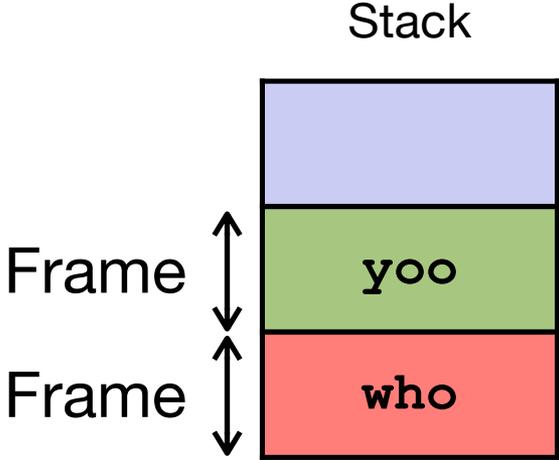
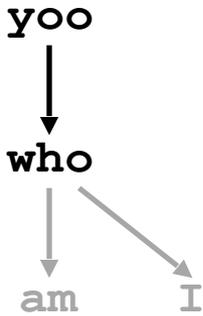


```
yoo (...)  
{  
  •  
  •  
  who ();  
  •  
  return  
}
```

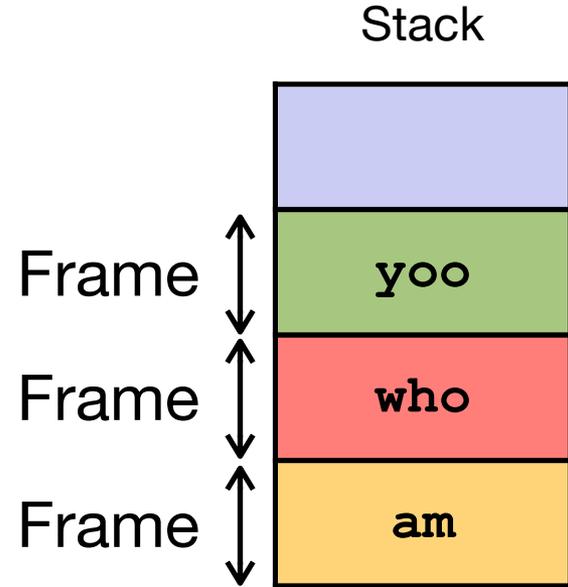
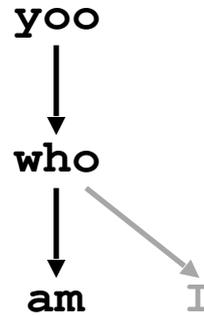
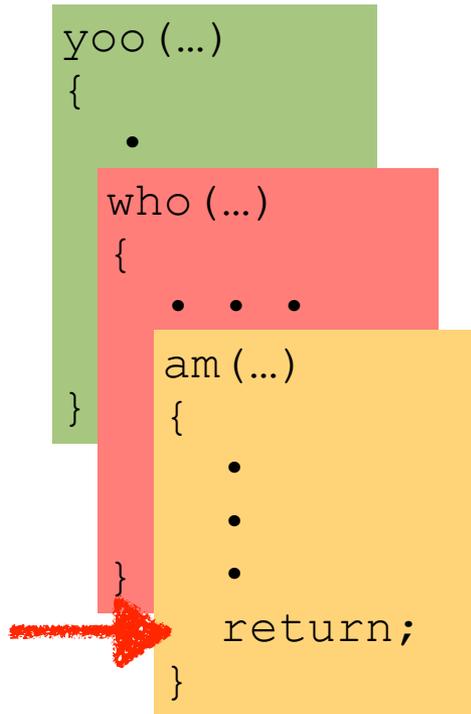


Run-Time Stack During Function Call

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

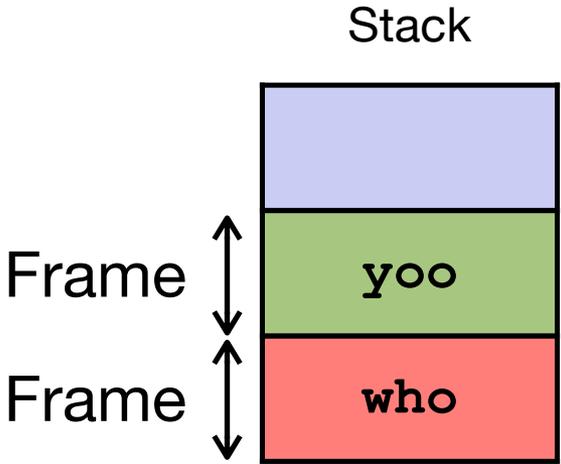
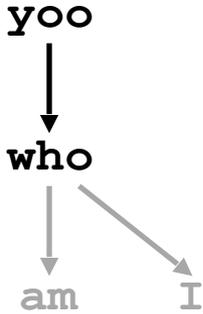


Run-Time Stack During Function Call

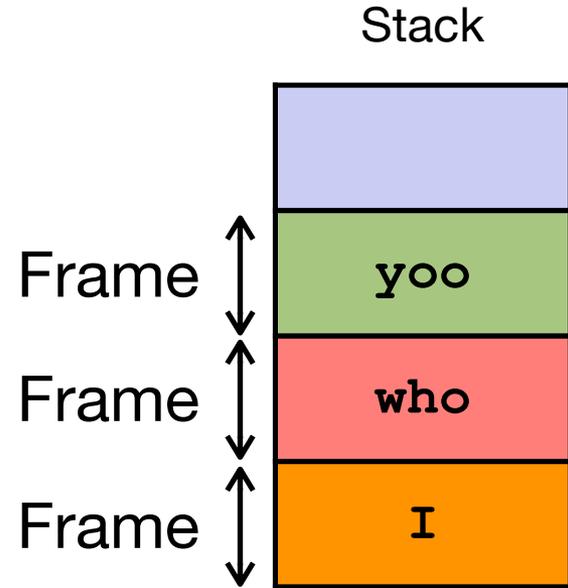
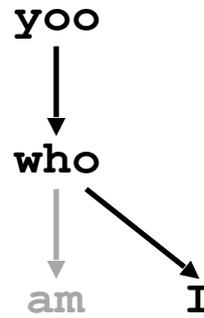
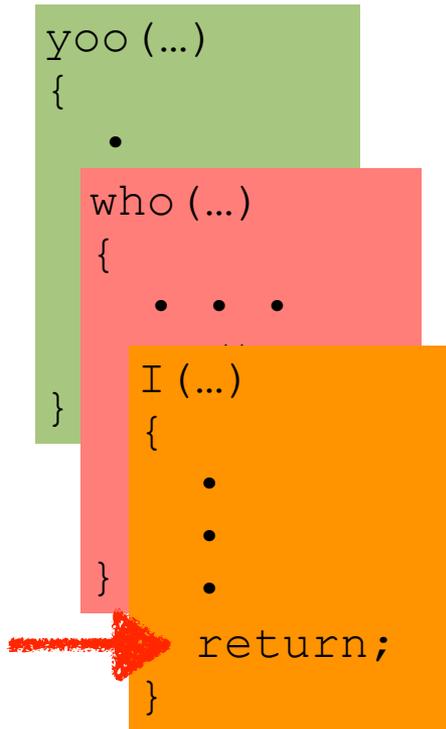


Run-Time Stack During Function Call

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

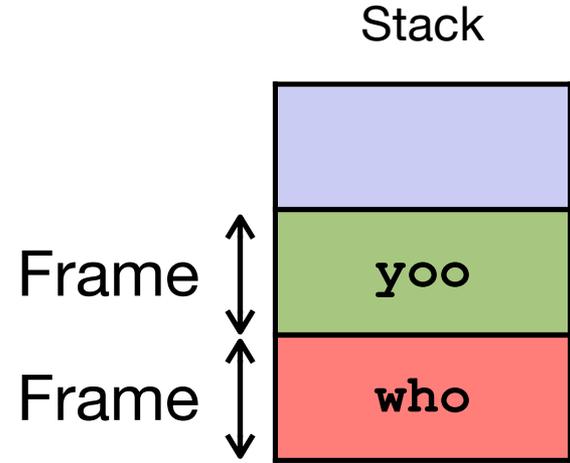
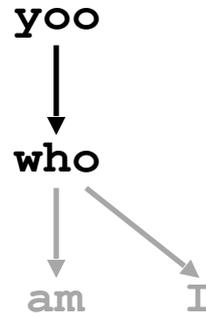


Run-Time Stack During Function Call



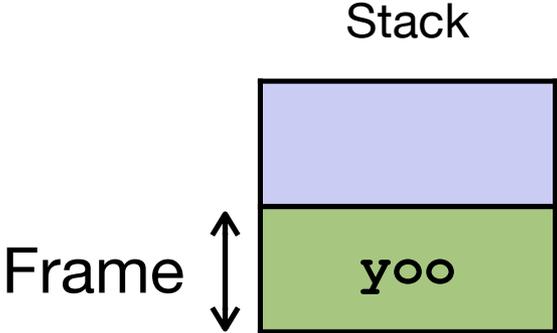
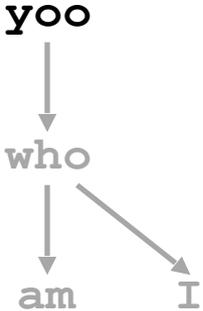
Run-Time Stack During Function Call

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



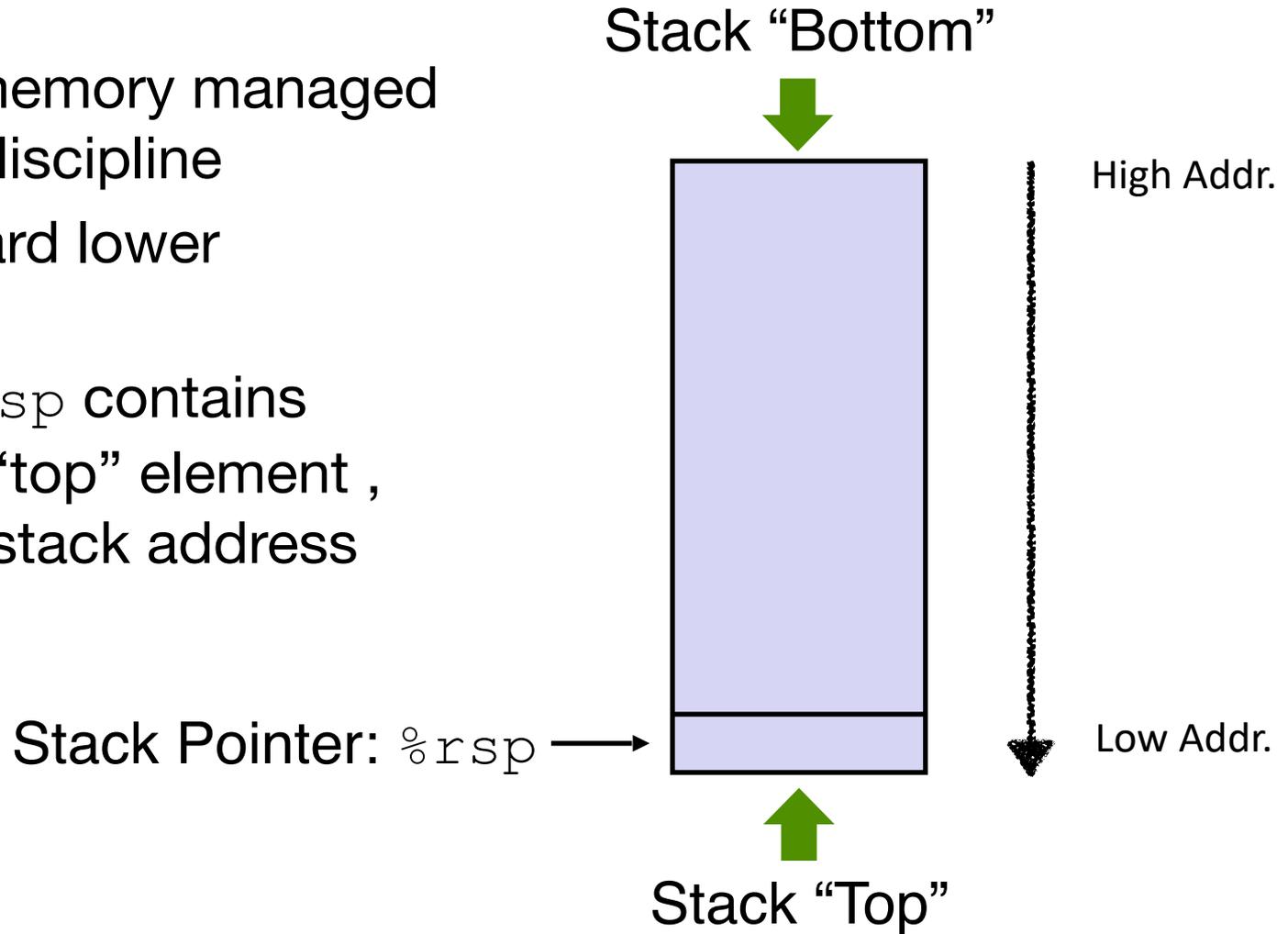
Run-Time Stack During Function Call

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



Stack in X86-64

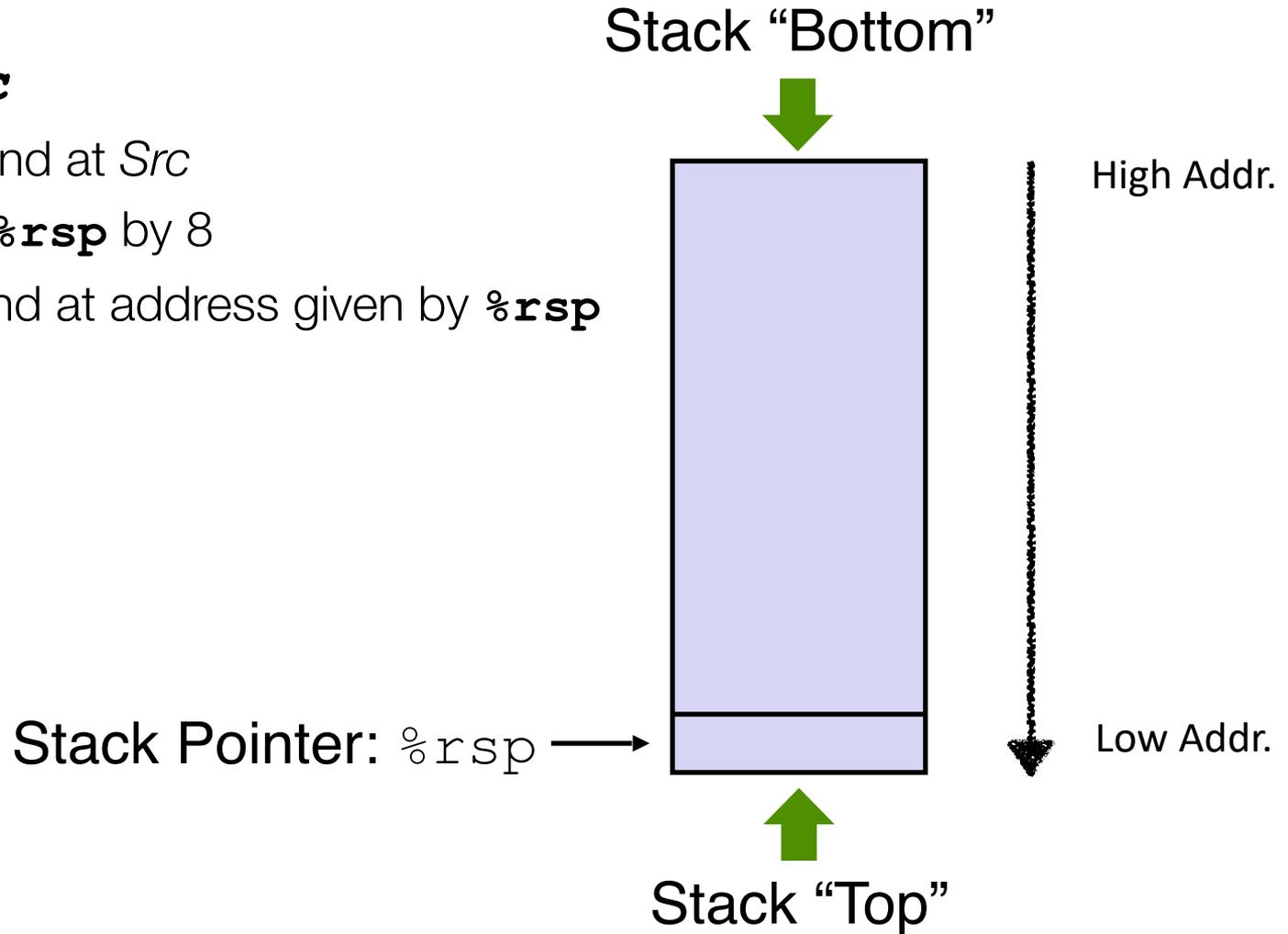
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register $\%rsp$ contains address of “top” element, i.e., lowest stack address



x86-64 Stack: Push

- **pushq Src**

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

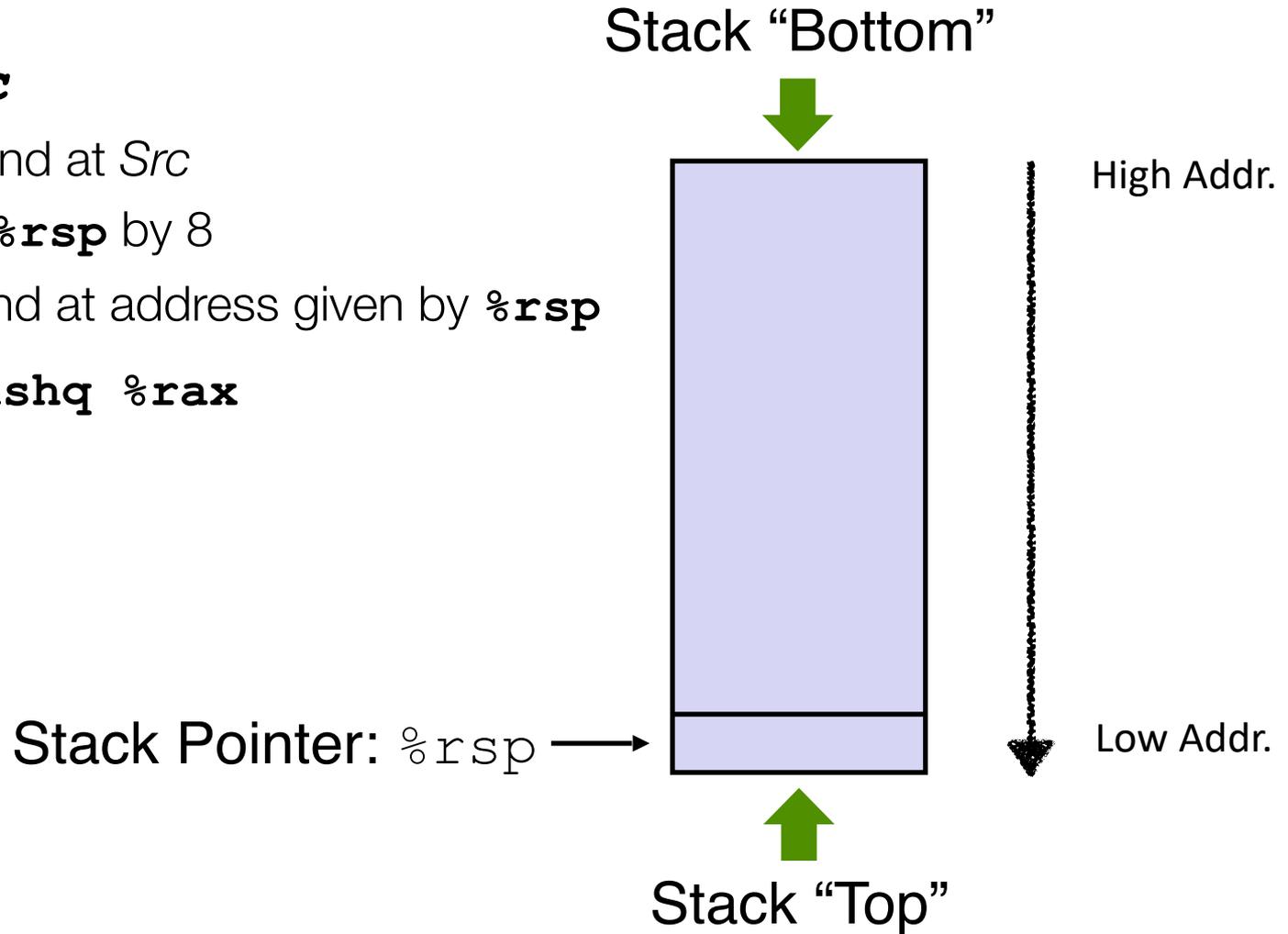


x86-64 Stack: Push

- **pushq Src**

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

- Example: **pushq %rax**

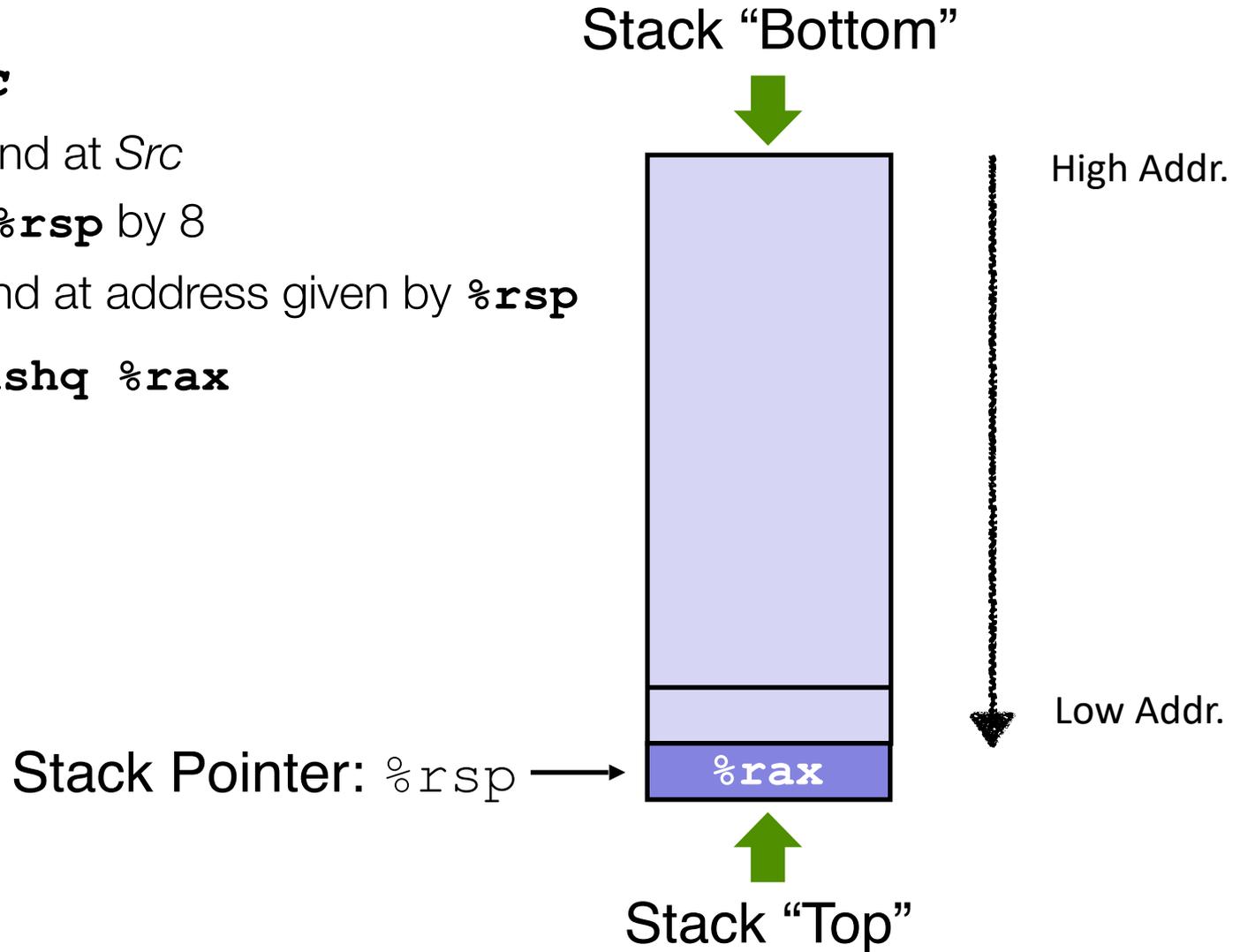


x86-64 Stack: Push

- **pushq Src**

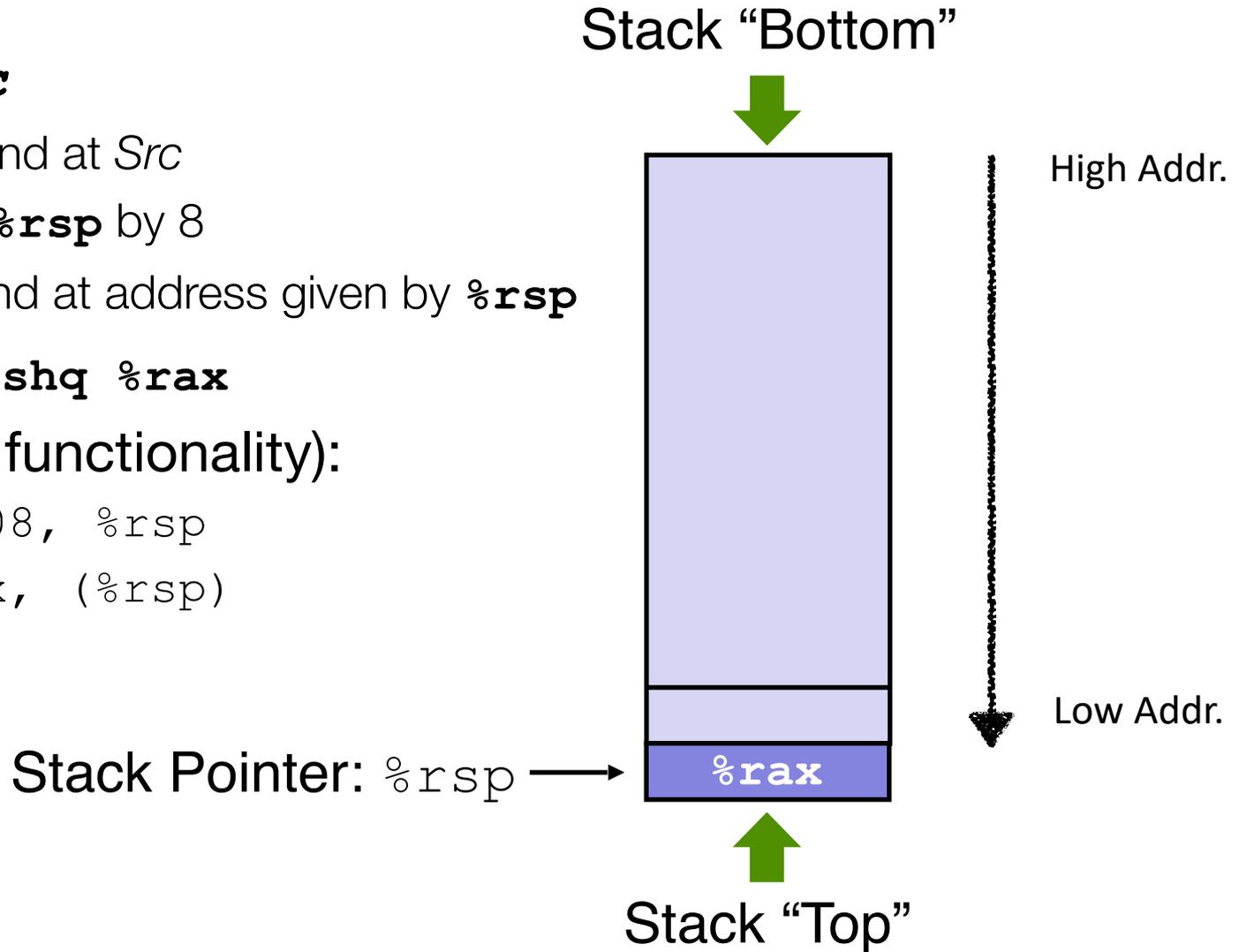
- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

- Example: **pushq %rax**



x86-64 Stack: Push

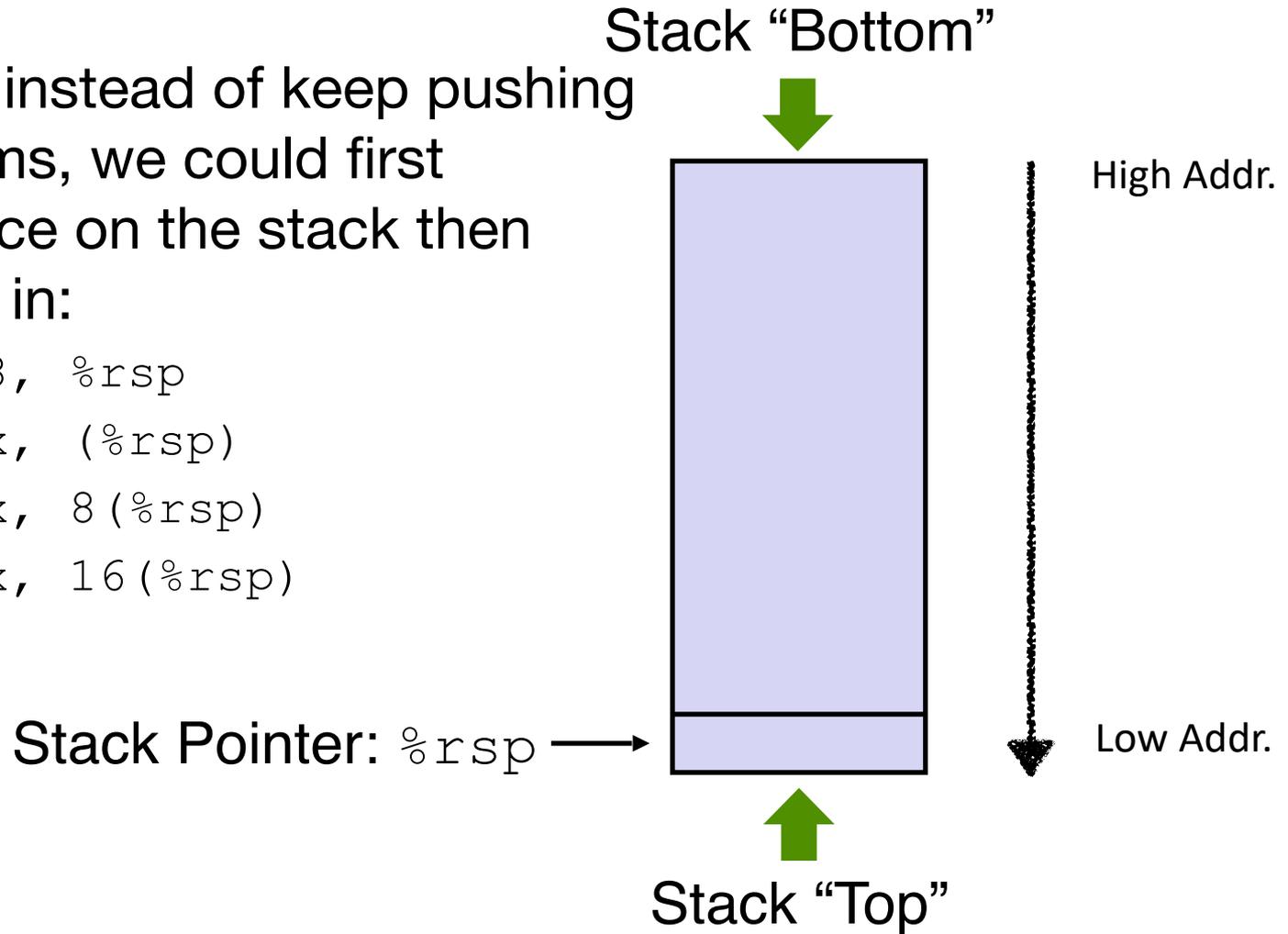
- **pushq Src**
 - Fetch operand at *Src*
 - Decrement `%rsp` by 8
 - Write operand at address given by `%rsp`
- Example: `pushq %rax`
- Same as (in functionality):
 - `subq $0x08, %rsp`
 - `movq %rax, (%rsp)`



x86-64 Stack: Push

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

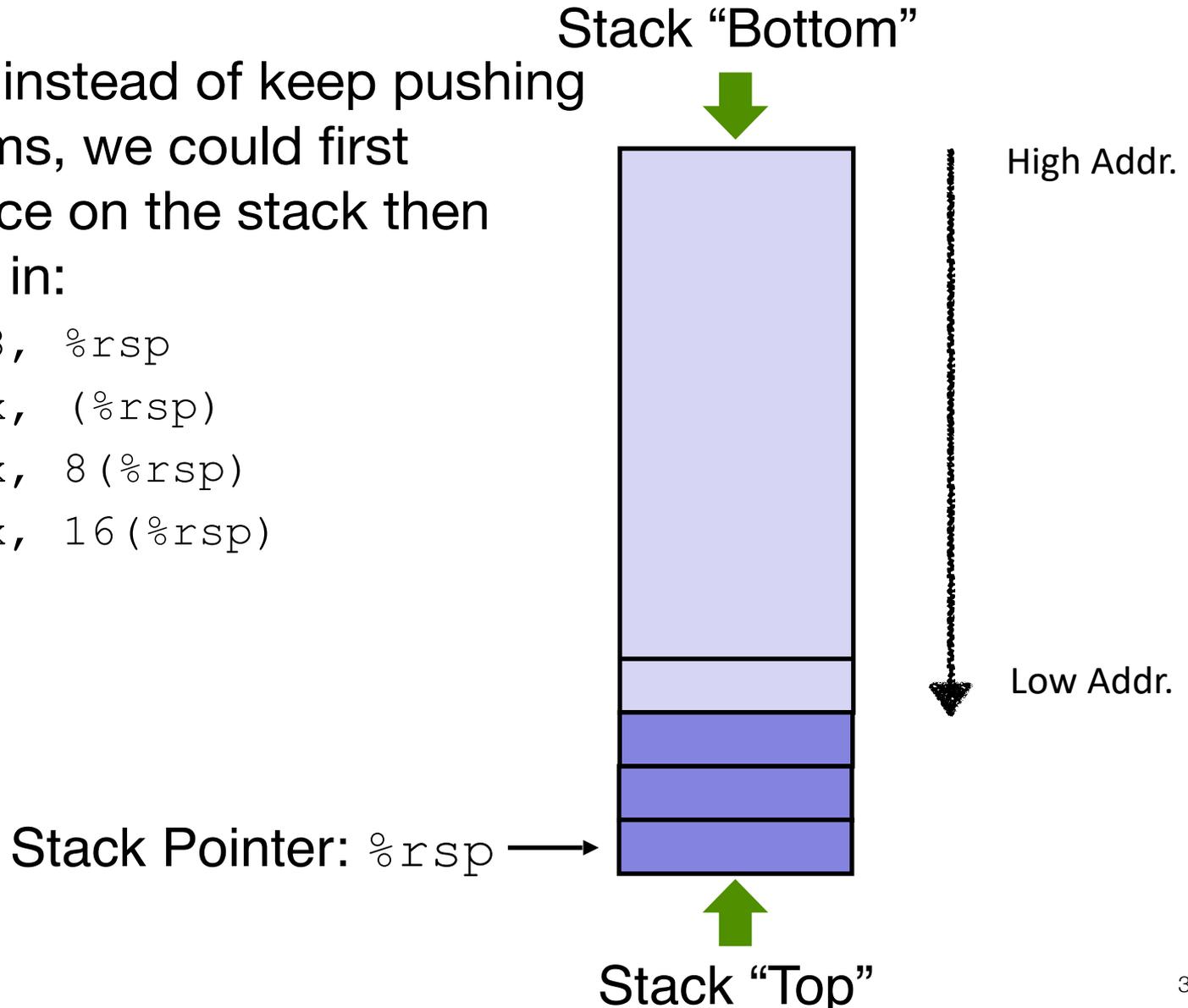
- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`



x86-64 Stack: Push

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

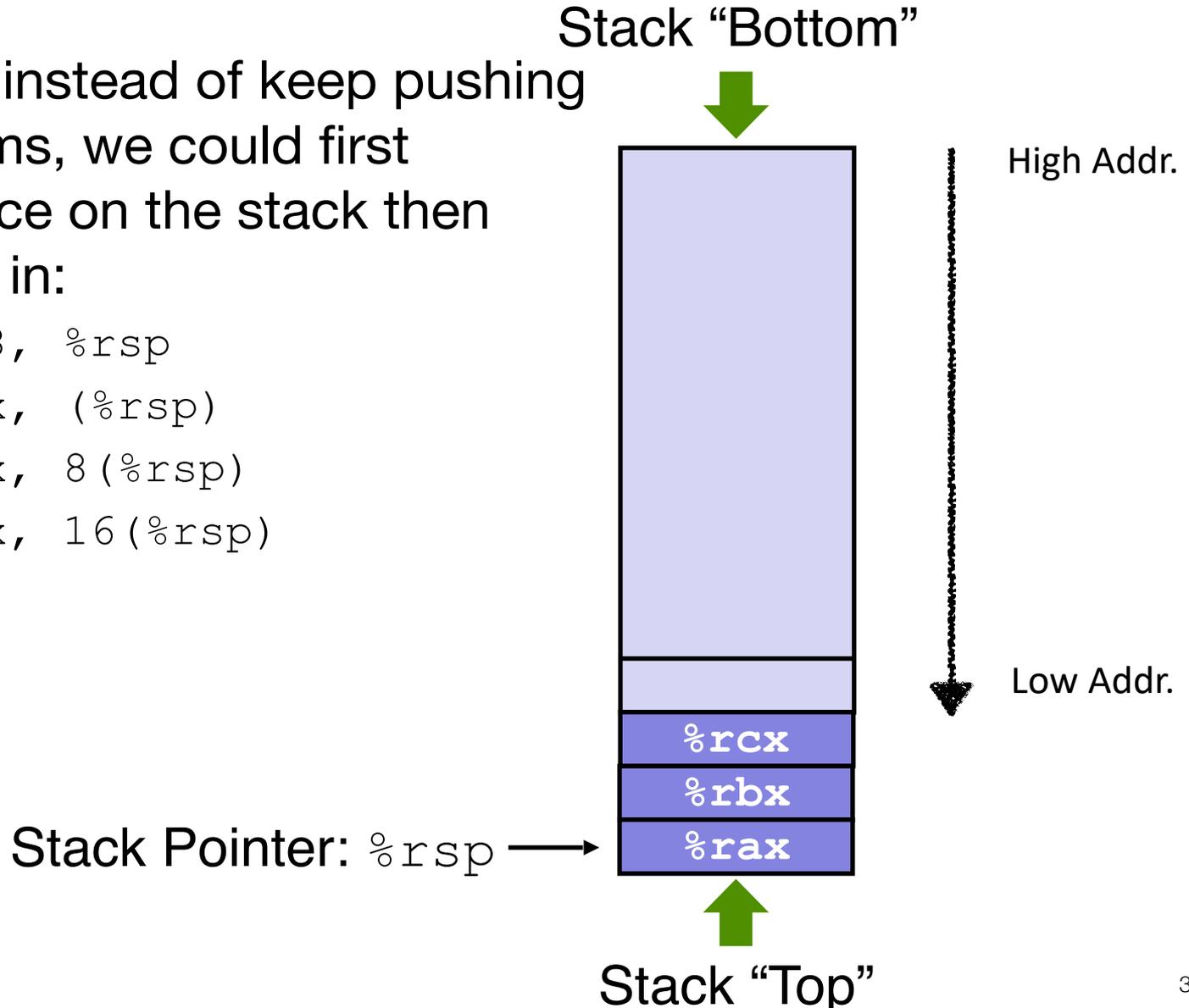
- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`



x86-64 Stack: Push

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

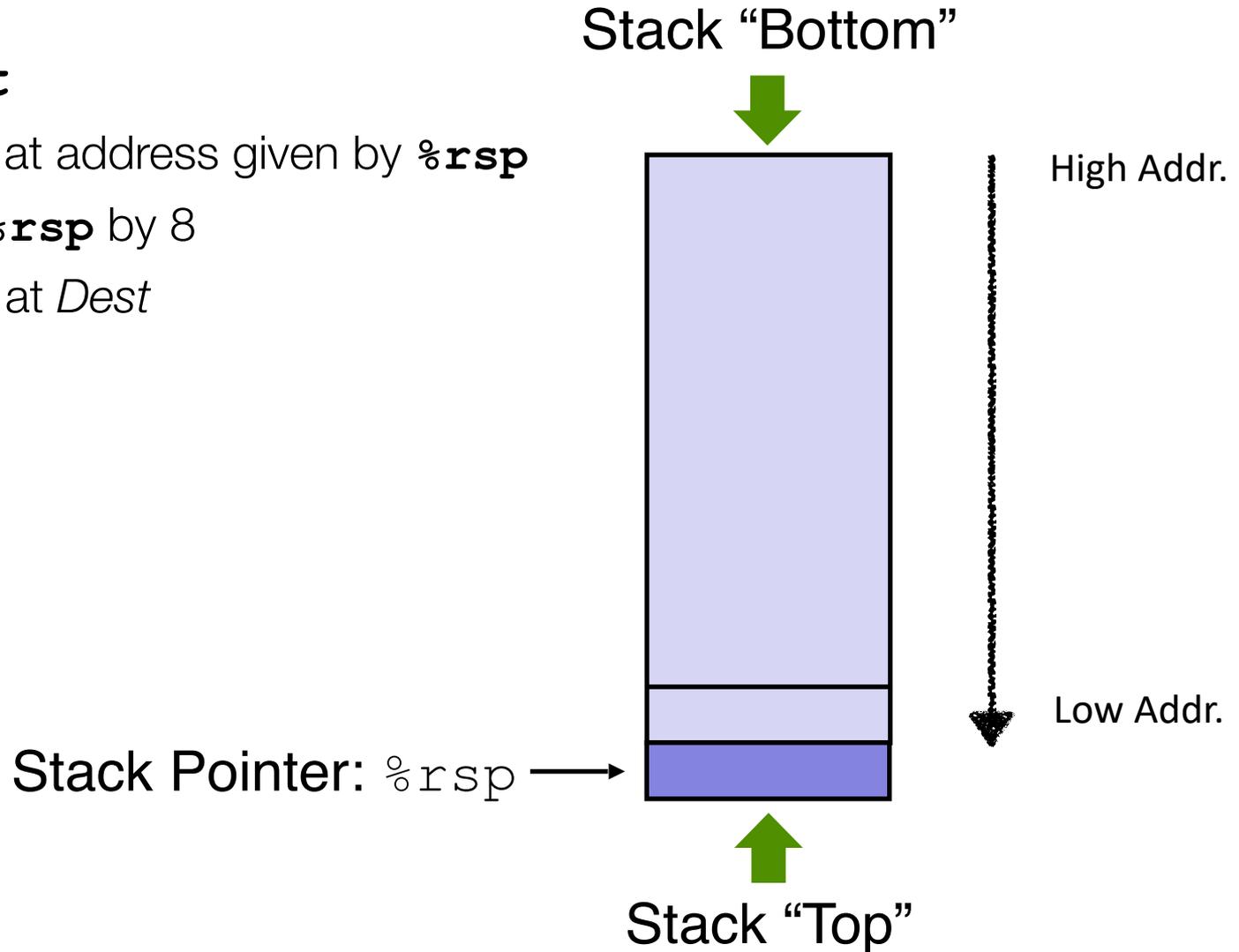
- `subq 0x18, %rsp`
- `movq %rax, (%rsp)`
- `movq %rbx, 8(%rsp)`
- `movq %rcx, 16(%rsp)`



x86-64 Stack: Pop

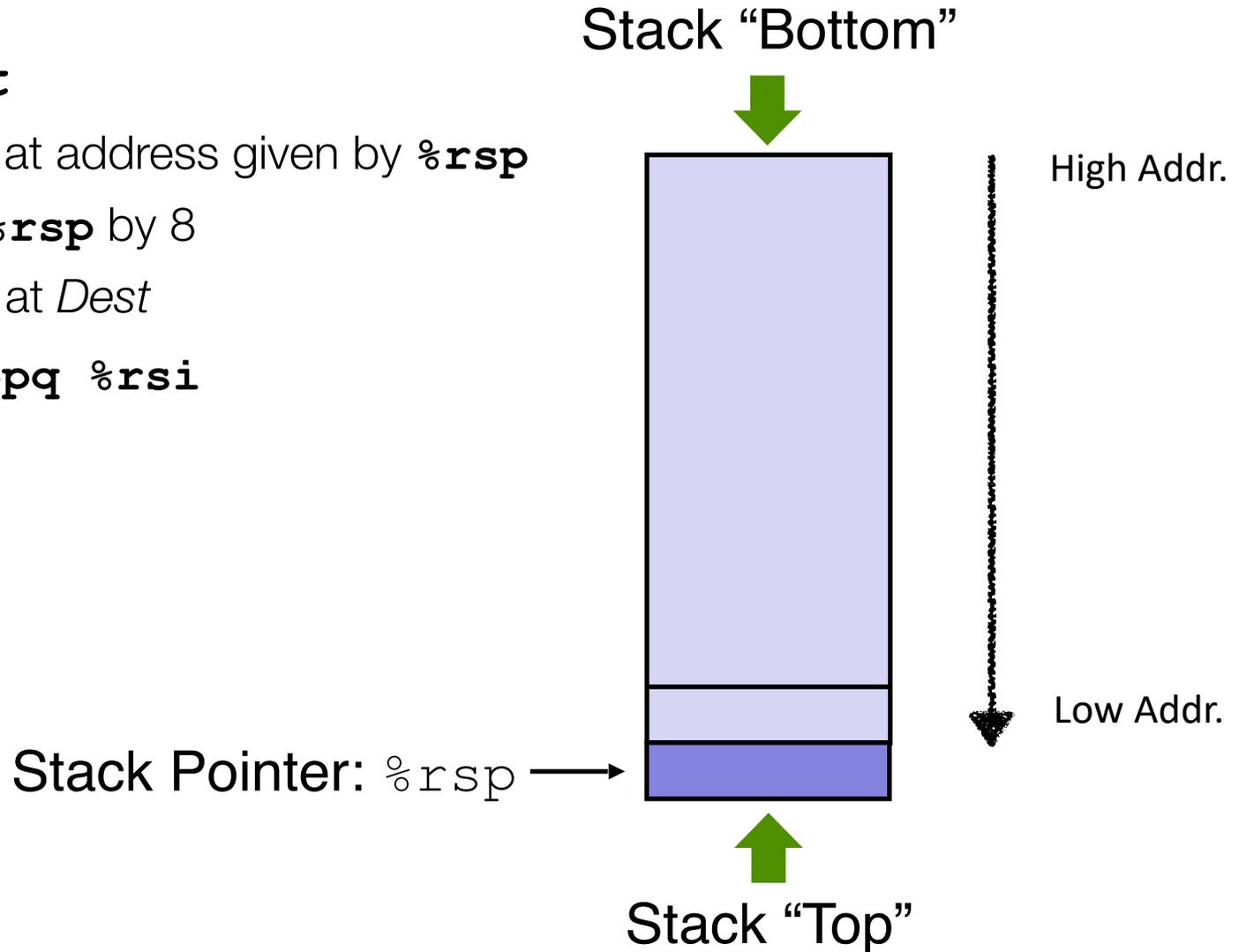
- **popq *Dest***

- Read value at address given by **%rsp**
- Increment **%rsp** by 8
- Store value at *Dest*



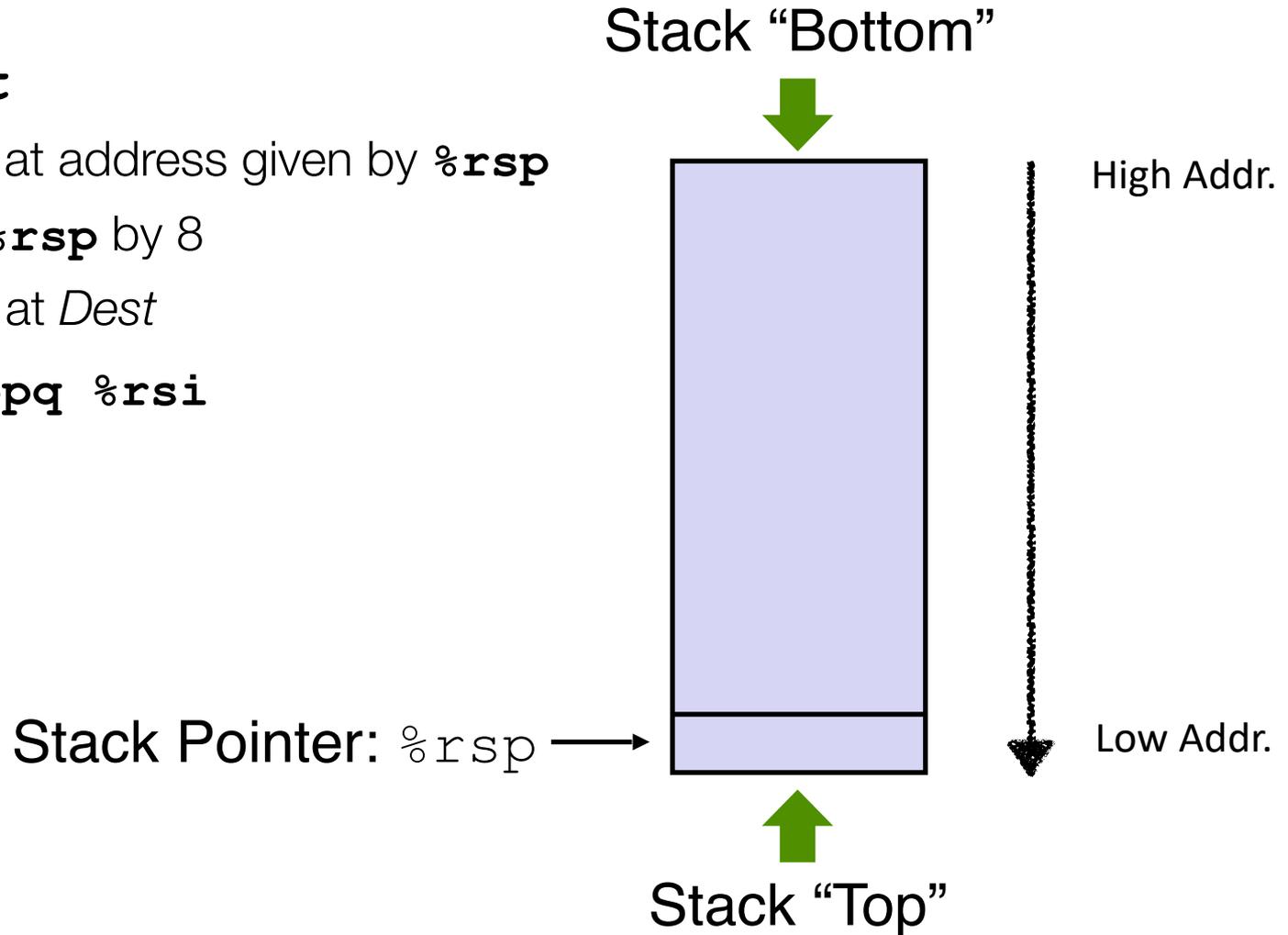
x86-64 Stack: Pop

- **popq *Dest***
 - Read value at address given by **%rsp**
 - Increment **%rsp** by 8
 - Store value at *Dest*
- Example: **popq %rsi**



x86-64 Stack: Pop

- **popq *Dest***
 - Read value at address given by **%rsp**
 - Increment **%rsp** by 8
 - Store value at *Dest*
- Example: **popq %rsi**



x86-64 Stack: Pop

- **popq *Dest***

- Read value at address given by **%rsp**
- Increment **%rsp** by 8
- Store value at *Dest*

- Example: **popq %rsi**

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

- `addq $0x08, %rsp`

Stack Pointer: `%rsp` →

Stack "Bottom"



High Addr.

Low Addr.

Stack "Top"

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

...

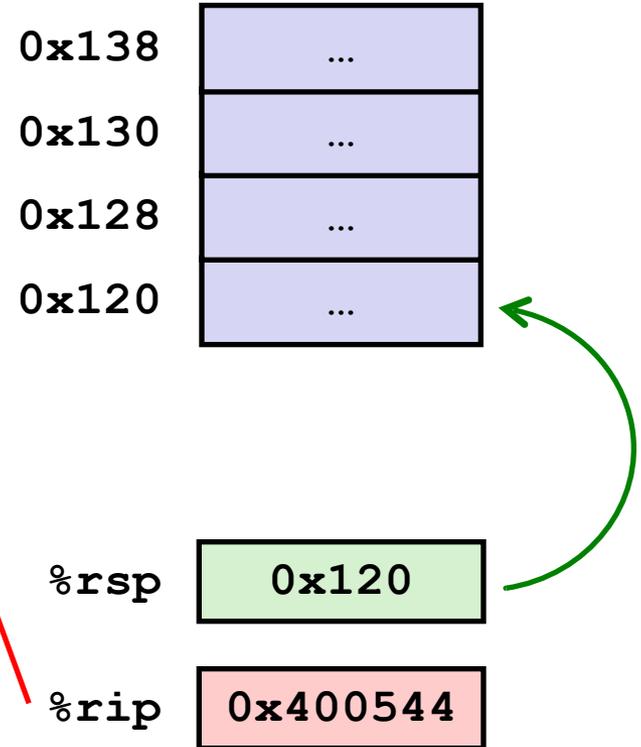
400550 <mult2>:
400550: mov    %rdi,%rax
400553: imul  %rsi,%rax
400557: retq
```

Function Call Example

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

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

Stack (Memory)

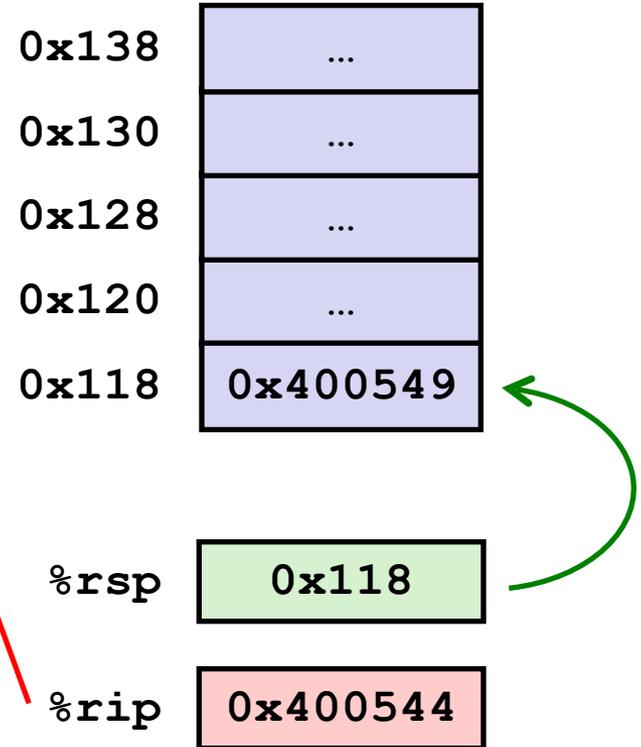


Function Call Example

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

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

Stack (Memory)

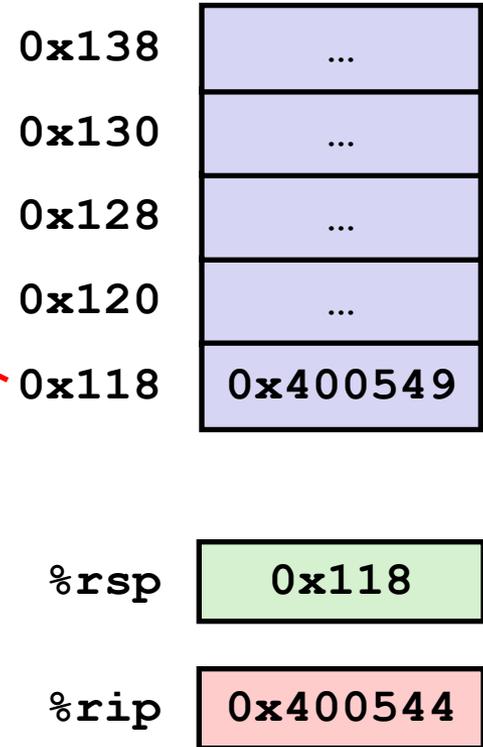


Function Call Example

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

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

Stack (Memory)

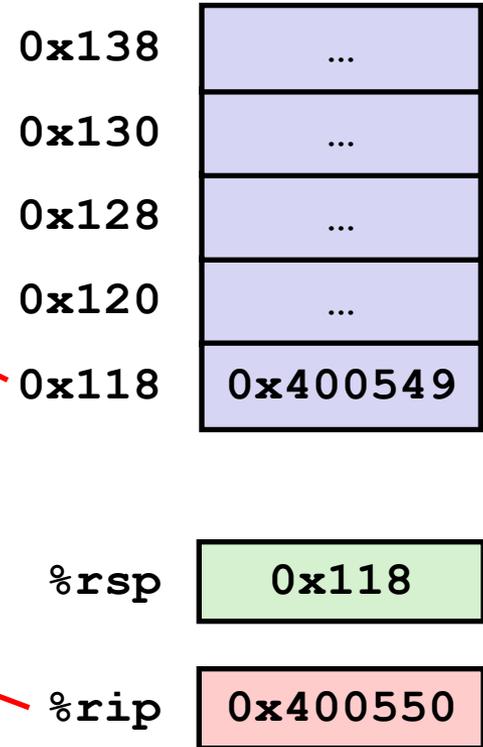


Function Call Example

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

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

Stack (Memory)

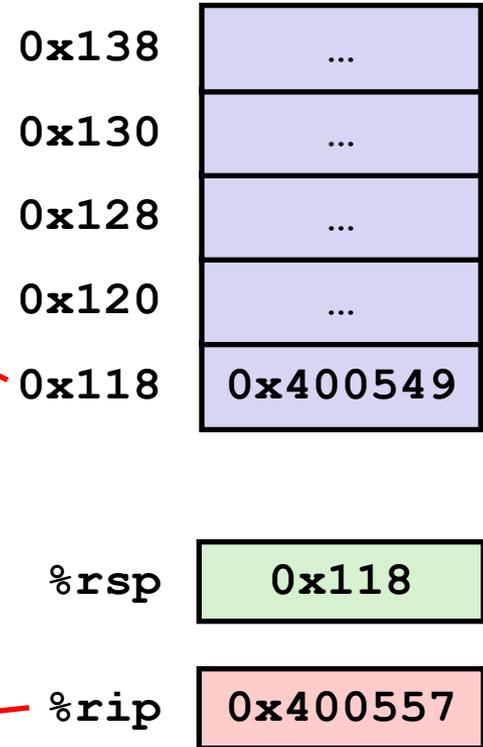


Function Call Example

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

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

Stack (Memory)

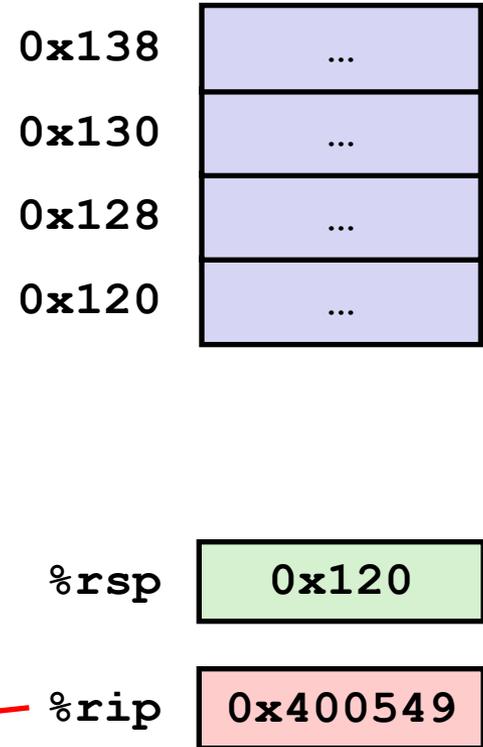


Function Call Example

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

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

Stack (Memory)

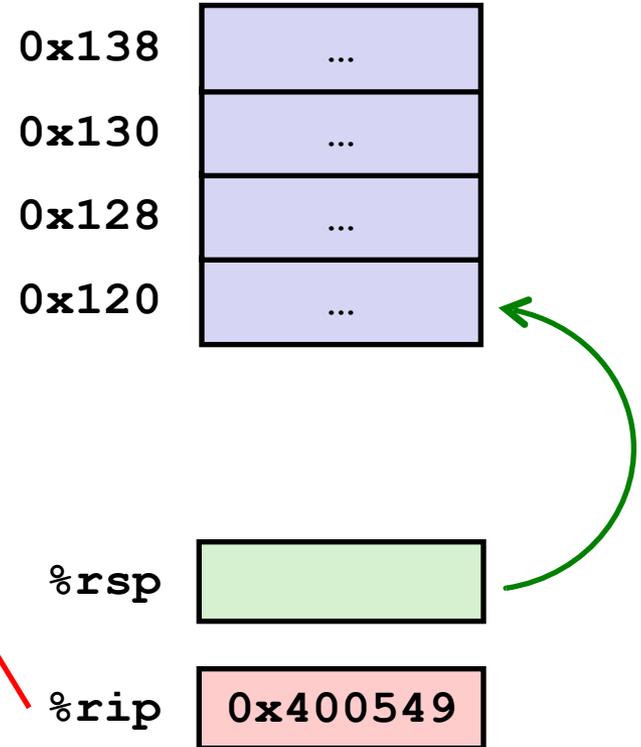


Function Call Example

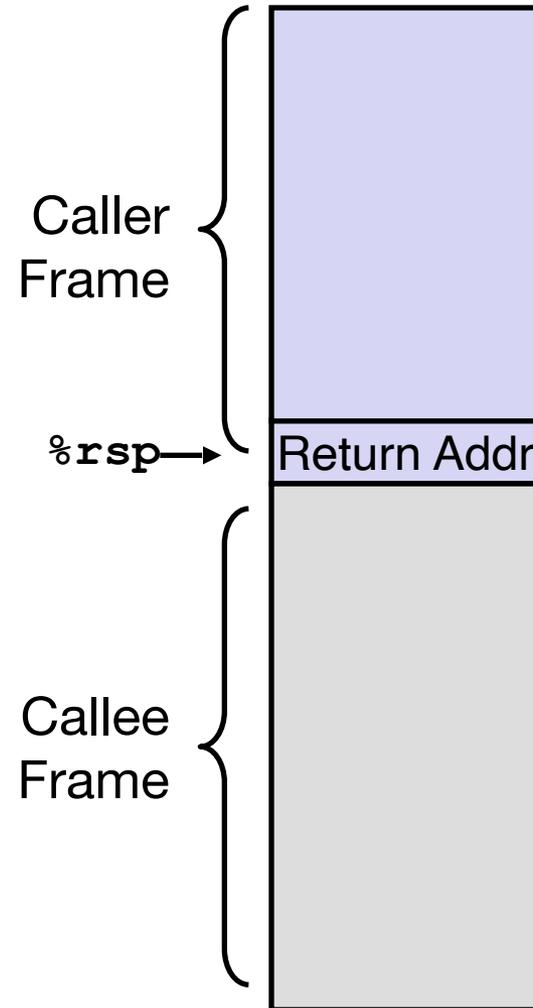
```
400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

```
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

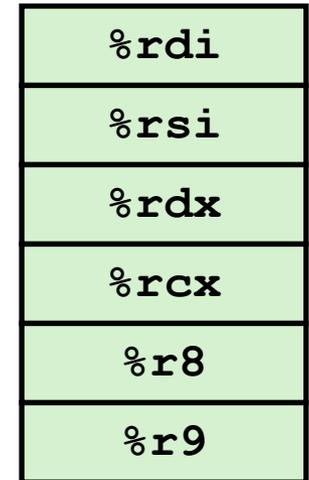
Passing Function Arguments

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

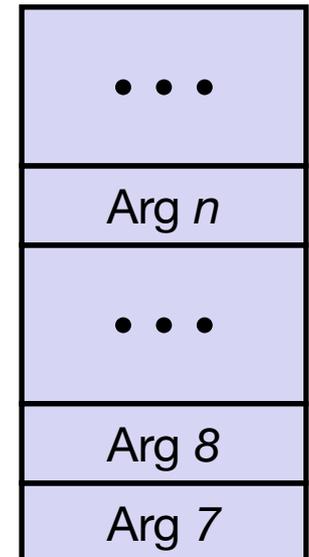
Registers

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`



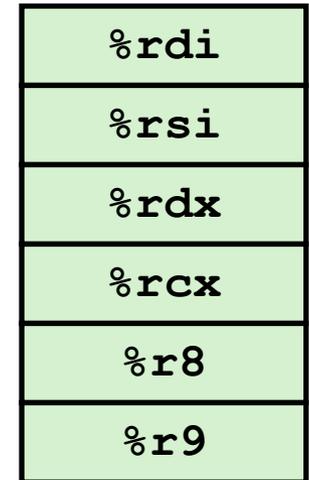
Stack



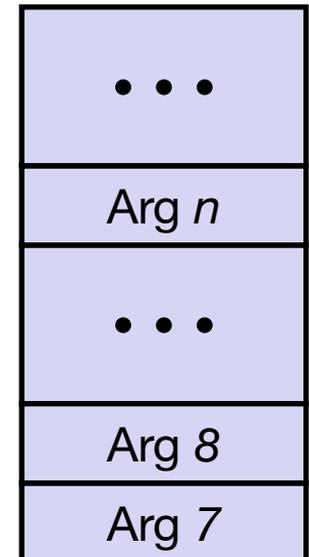
Registers

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`
- 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



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

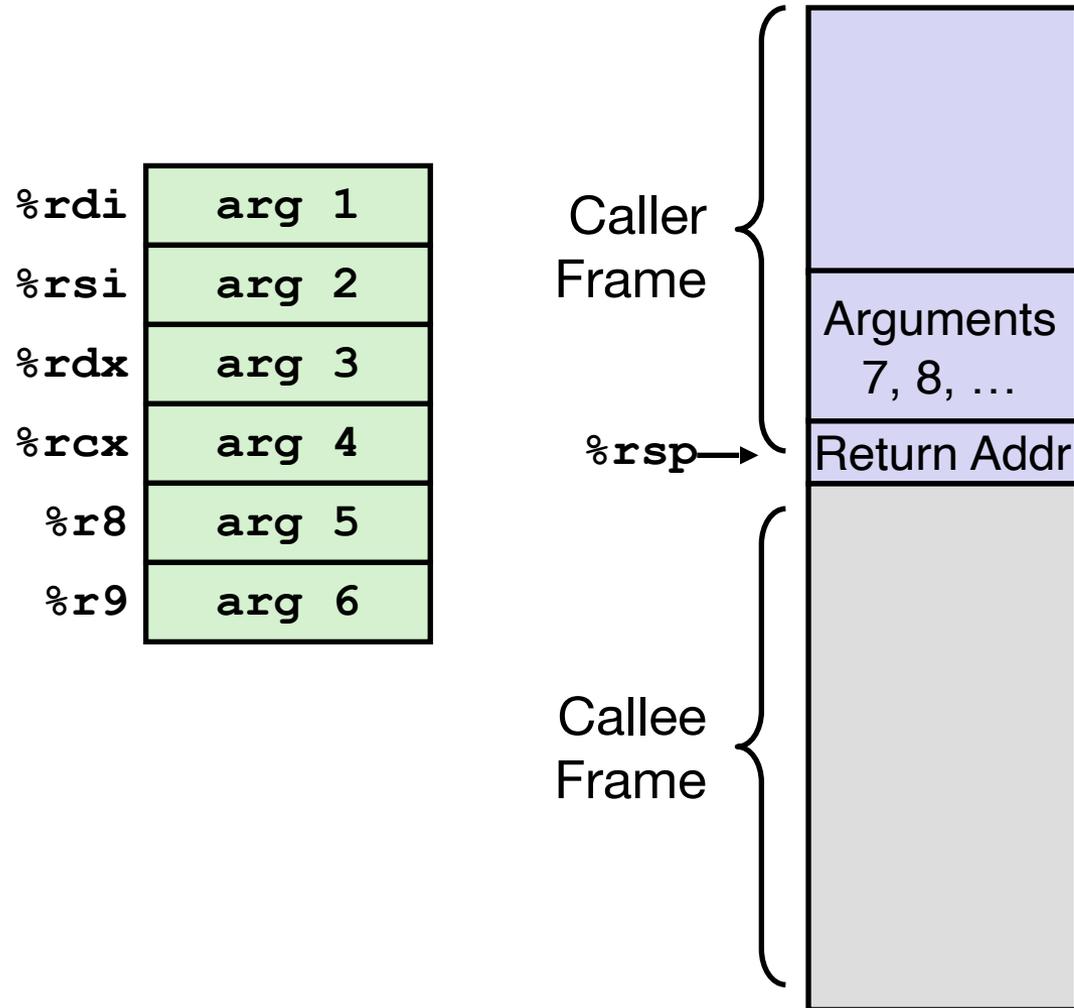
%rdi
%rsi
%rdx
%rcx
%r8
%r9

```
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. Take 255/455.

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

Register Example: `incr`

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

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

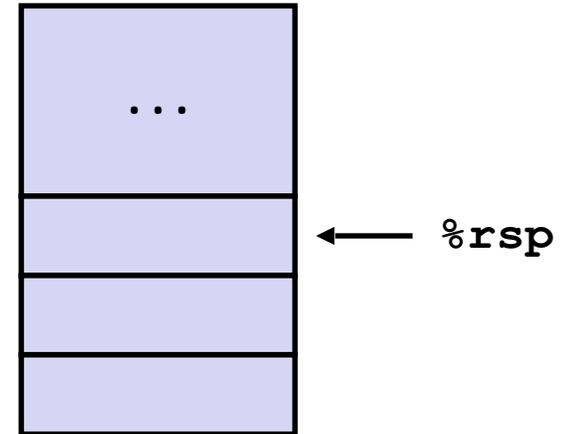
```
incr:  
    movq    (%rdi), %rax  
    addq   %rax, %rsi  
    movq   %rsi, (%rdi)  
    ret
```

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

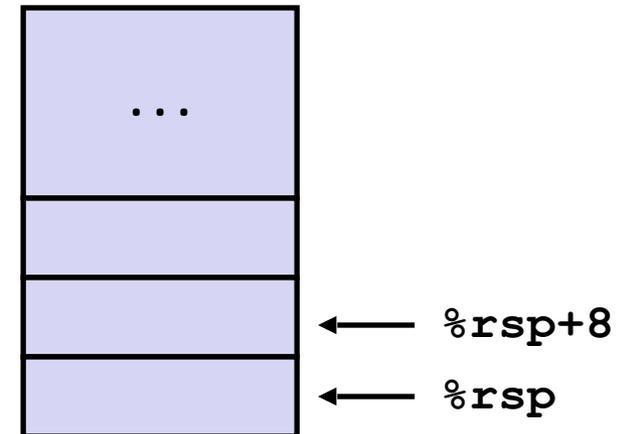


Stack Example: `call_add`

```
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

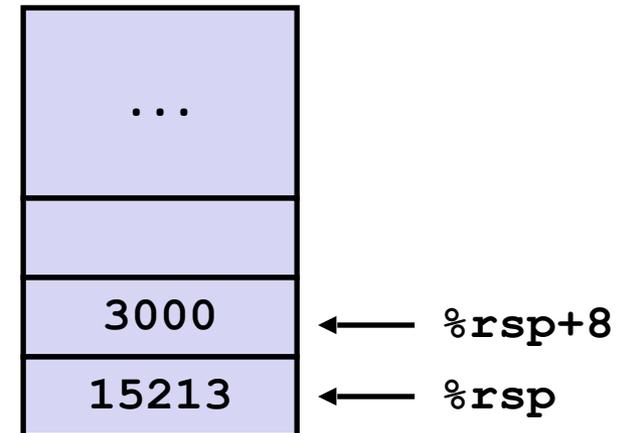


Stack Example: `call_add`

```
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

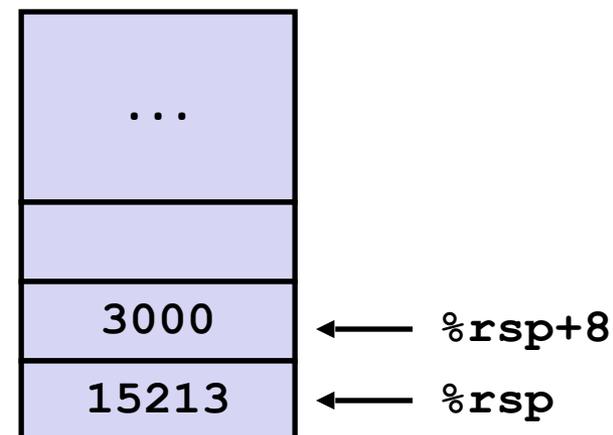


Stack Example: `call_add`

```
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



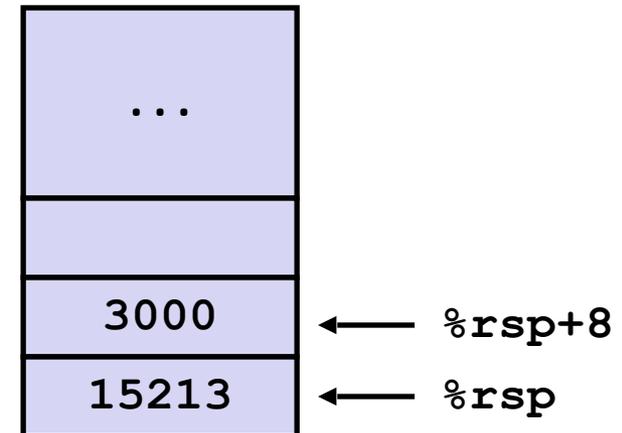
Register	Value(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>&v2</code>

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



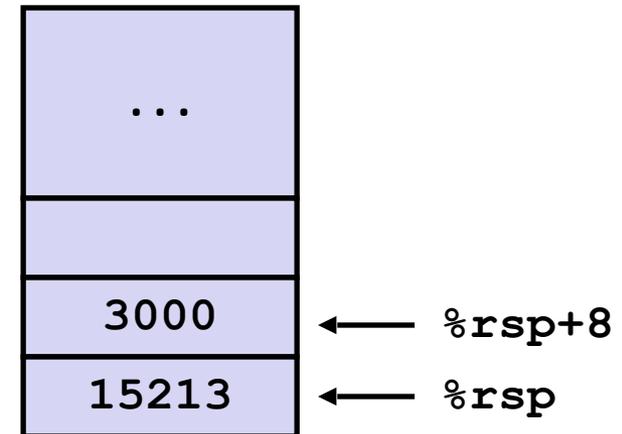
Register	Value(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>&v2</code>
<code>%rax</code>	18213

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



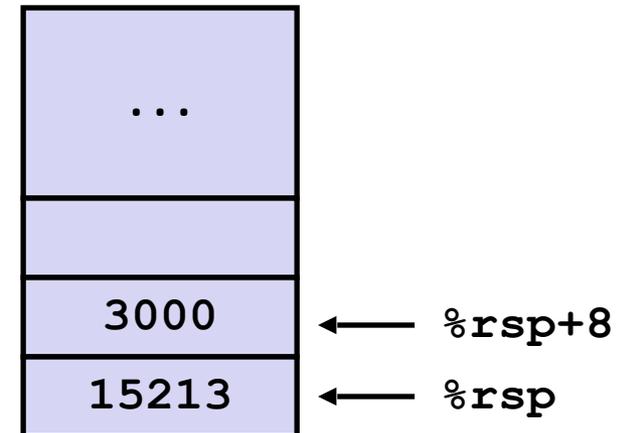
Register	Value(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>&v2</code>
<code>%rax</code>	21213

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

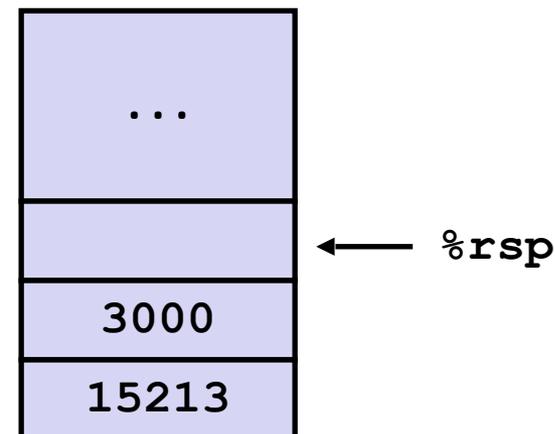


Stack Example: `call_add`

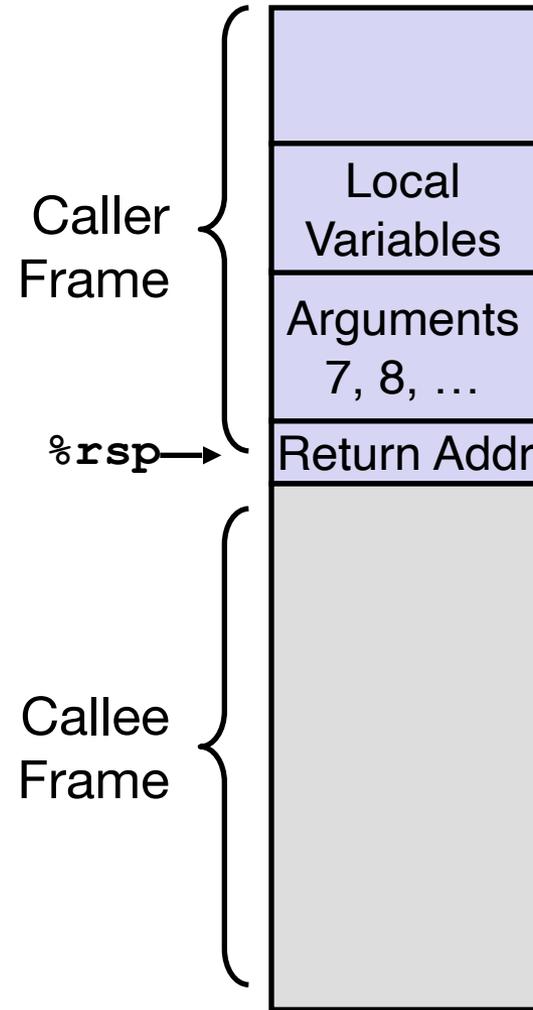
```
long call_add() {  
    long v1 = 15213;  
    long v2 = 3000;  
    long v3 = add(&v1, &v2);  
    return v2+v3;  
}
```

```
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



Stack Frame (So Far...)



Register Saving Conventions

Register Saving Conventions

- Any issue with using registers for temporary storage?

Caller

```
yoo:  
...  
movq $15213, %rdx  
call who  
addq %rdx, %rax  
...  
ret
```

Callee

```
who:  
...  
subq $18213, %rdx  
...  
ret
```

Register Saving Conventions

- Any issue with using registers for temporary storage?
 - Contents of register `%rdx` overwritten by `who()`

Caller

```
yoo:  
...  
movq $15213, %rdx  
call who  
addq %rdx, %rax  
...  
ret
```

Callee

```
who:  
...  
subq $18213, %rdx  
...  
ret
```

Register Saving Conventions

- Any issue with using registers for temporary storage?
 - Contents of register `%rdx` overwritten by `who()`
 - This could be trouble → Need some coordination

Caller

```
yoo:  
...  
movq $15213, %rdx  
call who  
addq %rdx, %rax  
...  
ret
```

Callee

```
who:  
...  
subq $18213, %rdx  
...  
ret
```

Register Saving Conventions

- Common conventions

- “*Caller Saved*”

- Caller saves temporary values in its frame (on the stack) before the call
 - Callee is then free to modify their values

- “*Callee Saved*”

- Callee saves temporary values in its frame before using
 - Callee restores them before returning to caller
 - Caller can safely assume that register values won't change after the function call

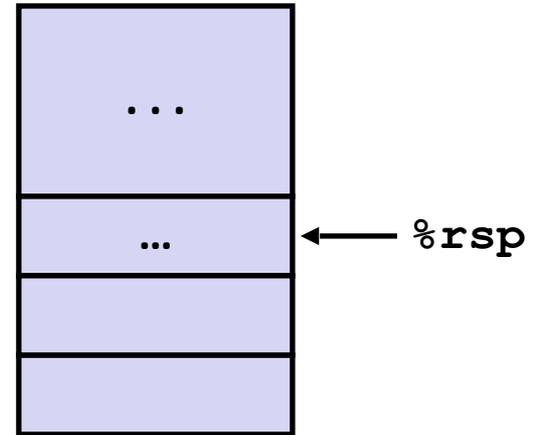
Register Saving Conventions

- Conventions used in x86-64 (*Part of the Calling Conventions*)
 - Some registers are saved by caller, some are by callee.
 - Caller saved: `%rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11`
 - Callee saved: `%rbx, %rbp, %r12, %r13, %r14, %r15`
 - `%rax` holds return value, so implicitly caller saved
 - `%rsp` is the stack pointer, so implicitly callee saved

Example

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

Stack

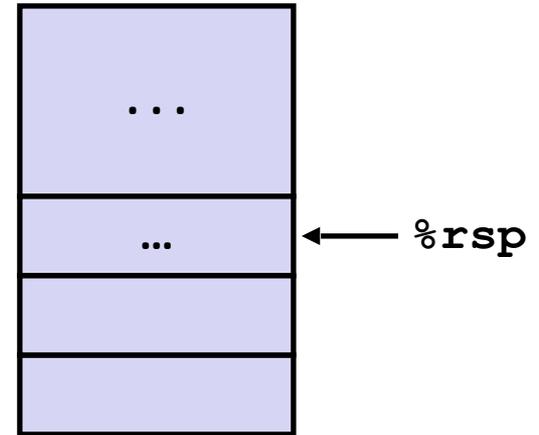


Example

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

```
call_incr2:  
    pushq    %rbx  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl     $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $8, %rsp  
    popq    %rbx  
    ret
```

Stack

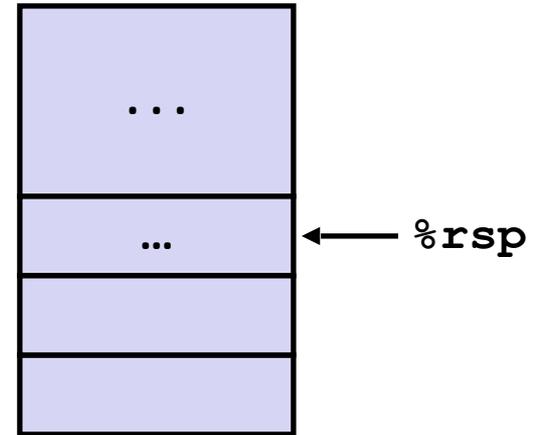


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq    %rdi, %rbx  
    movl    $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack

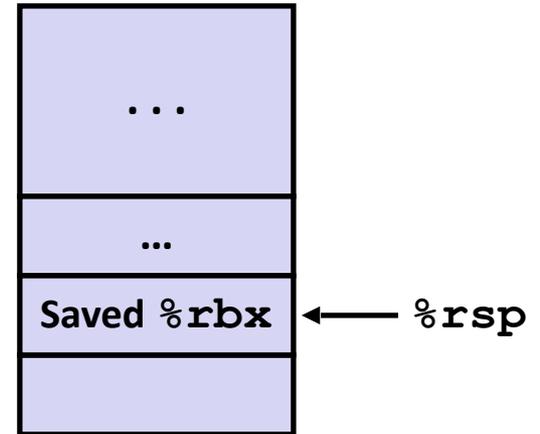


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl     $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq     %rbx, %rax  
    addq     $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack

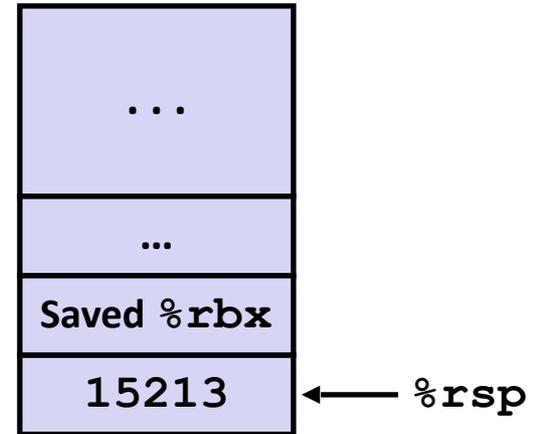


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl     $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq     %rbx, %rax  
    addq     $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack

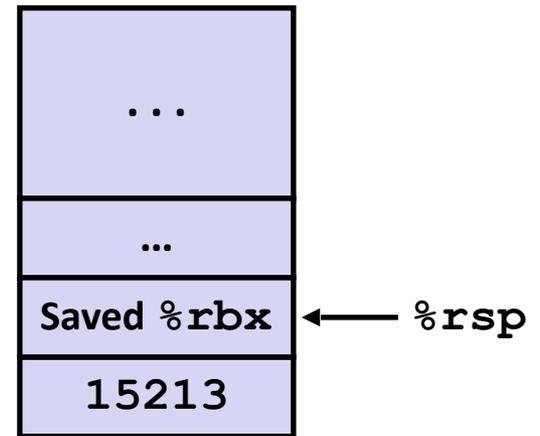


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl    $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack

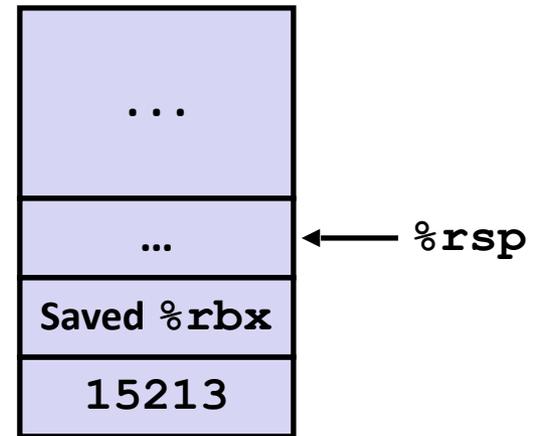


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl     $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq     %rbx, %rax  
    addq     $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack

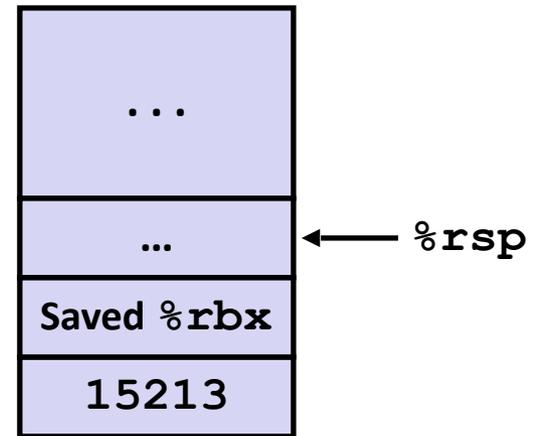


Example

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

```
call_incr2:  
    pushq    %rbx          ←  
    pushq    $15213  
    movq     %rdi, %rbx  
    movl     $3000, %esi  
    leaq    (%rsp), %rdi  
    call    incr  
    addq     %rbx, %rax  
    addq     $8, %rsp  
    popq    %rbx          ←  
    ret
```

Stack



- `call_incr2` needs to save `%rbx` (callee-saved) because it will modify its value
- It can safely use `%rbx` after `call incr` because `incr` will have to save `%rbx` if it needs to use it (again, `%rbx` is callee saved)

Stack Frame: Putting It Together

