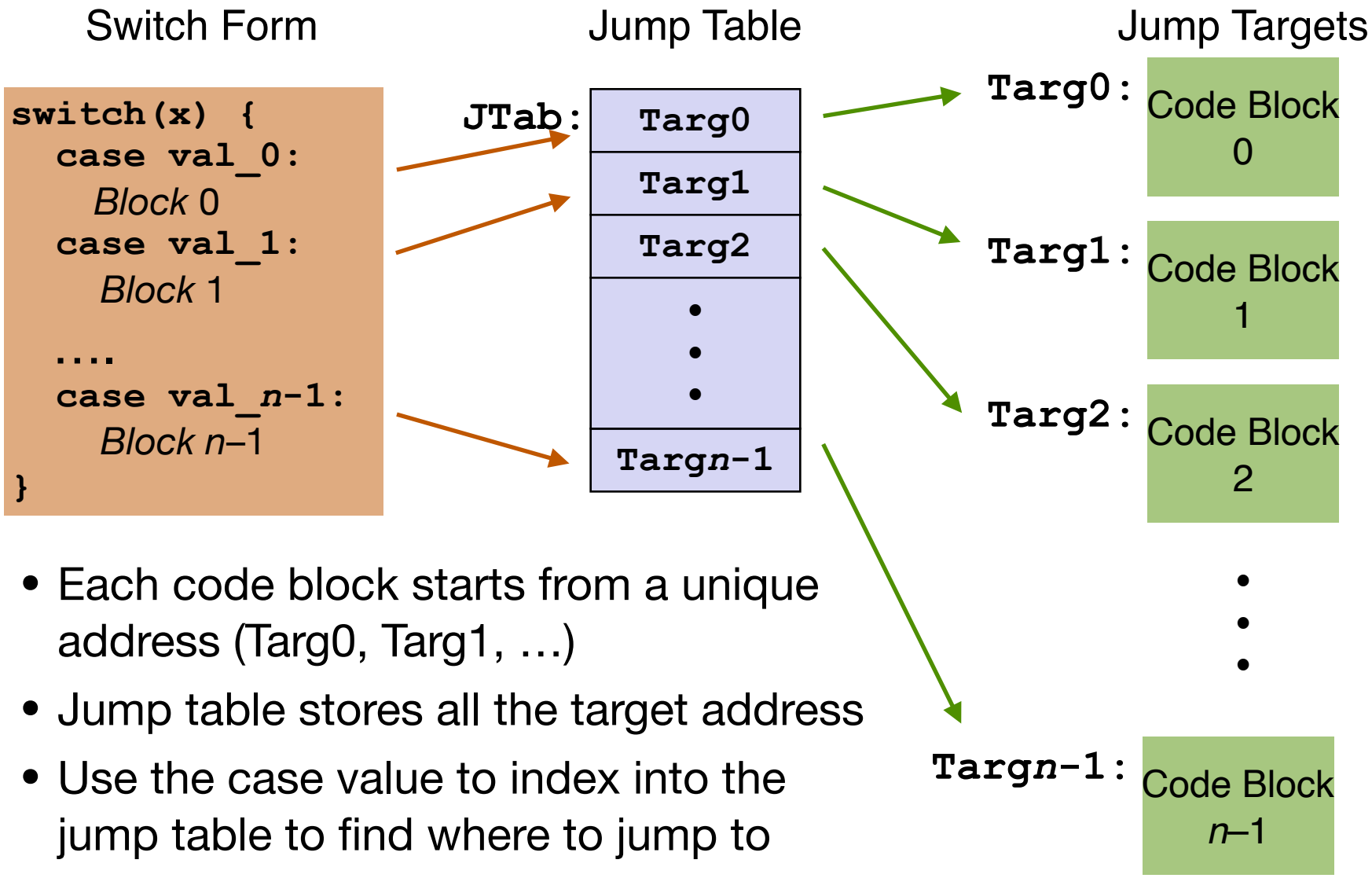
CSC 252: Computer Organization
Spring 2025: Lecture 9

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Implementing Switch Using Jump Table



- 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

Assembly Directives (Pseudo-Ops)

```
.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
```

- Directives:
  - Not real instructions, but assist assembler. Think of them as messages to help the assembler in the assembly process.

- .quad: tells the assembler to set aside the next 8 bytes in memory and initialize with the value of the operand (a label here, which itself is an address)
- .align: tells the assembler that addresses of the the following data will be aligned to 8 bytes
- .section: denotes different parts of the object file
- .rodata: read-only data section

Jump Table and Jump Targets

Jump Table

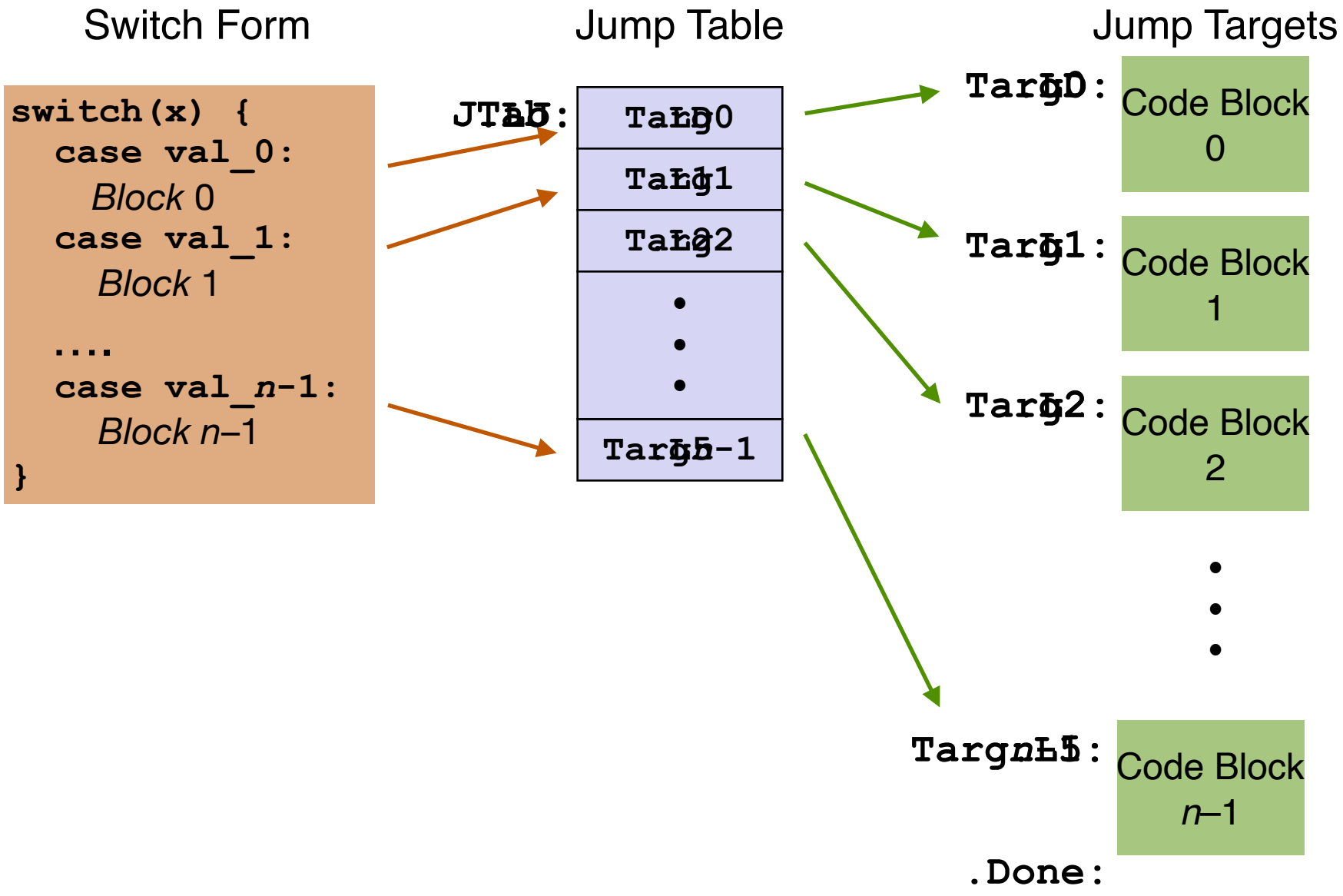
```
.section  .rodata
  .align 8
.L4:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

`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
```

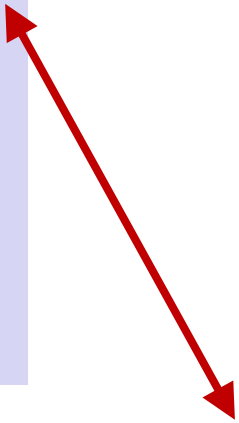
Implementing Switch Using Jump Table



Code Blocks (x == 1)

```
.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
```

```
switch(x) {
case 1:          // .L1
     w = y*z;
     break;
     …
}
```



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

```
.L1:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   jmp      .done
```

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

```
.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
```

```
switch(x) {
…
case 2:          // .L2
    w = y/z;
    /* Fall Through */
case 3:          // .L3
    w += z;
    break;
    …
}
```

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

```
.L2:                        # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx            #  y/z

.L3:                        # Case 3
    addq    %rcx, %rax      #  w += z
    jmp     .done
```

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

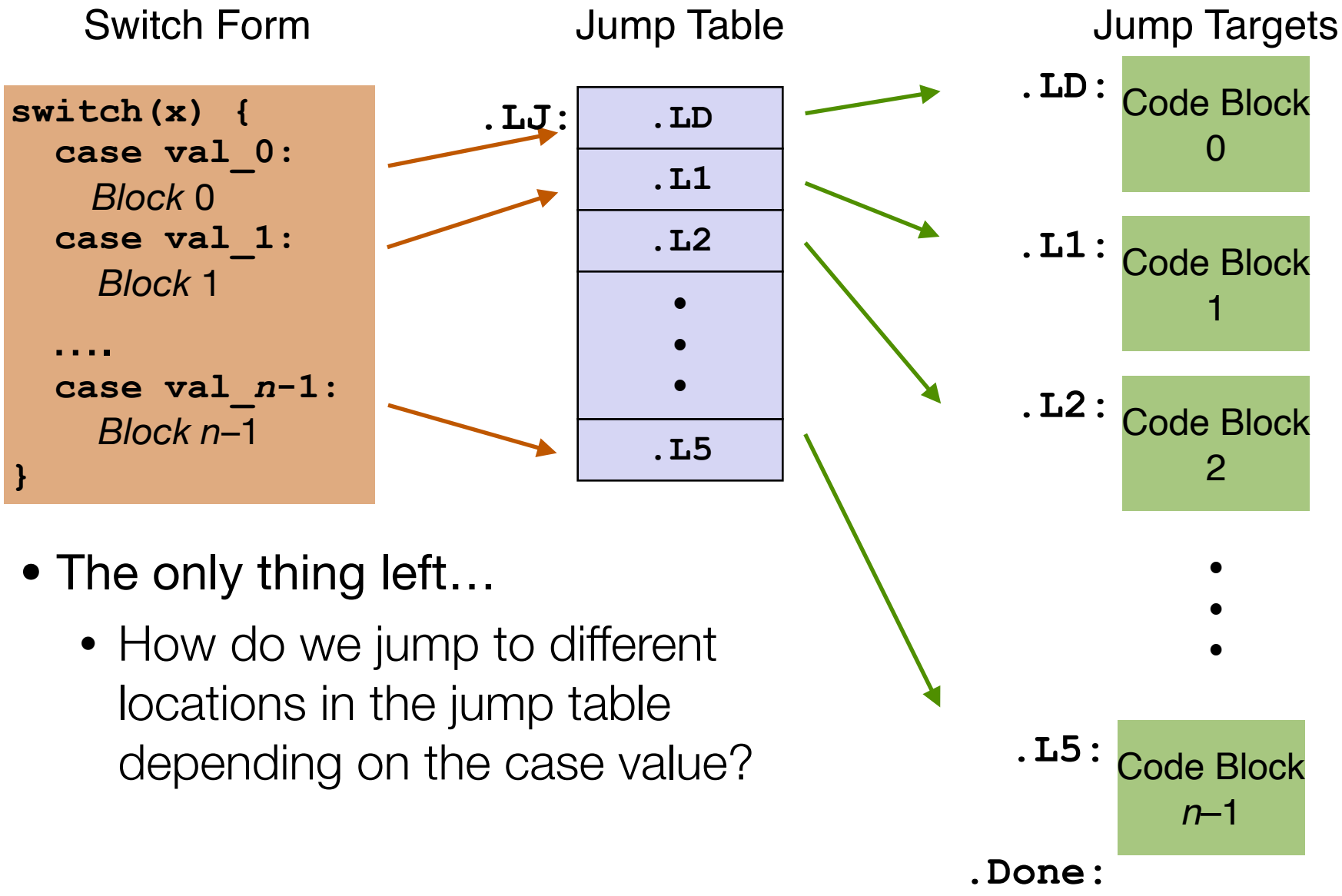
```
.section   .rodata
  .align 8
.L4:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

```
switch(x) {
…
case 5:   // .L5
case 6:   // .L5
     w -= z;
     break;
default: // .LD
     w = 2;
}
```

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

```
.L5:                    # Case 5,6
  subq   %rdx, %rax #  w -= z
  jmp    .done
.LD:                    # Default:
  movl   $2, %eax    #  2
  jmp    .done
```

Implementing Switch Using Jump Table



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

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
```

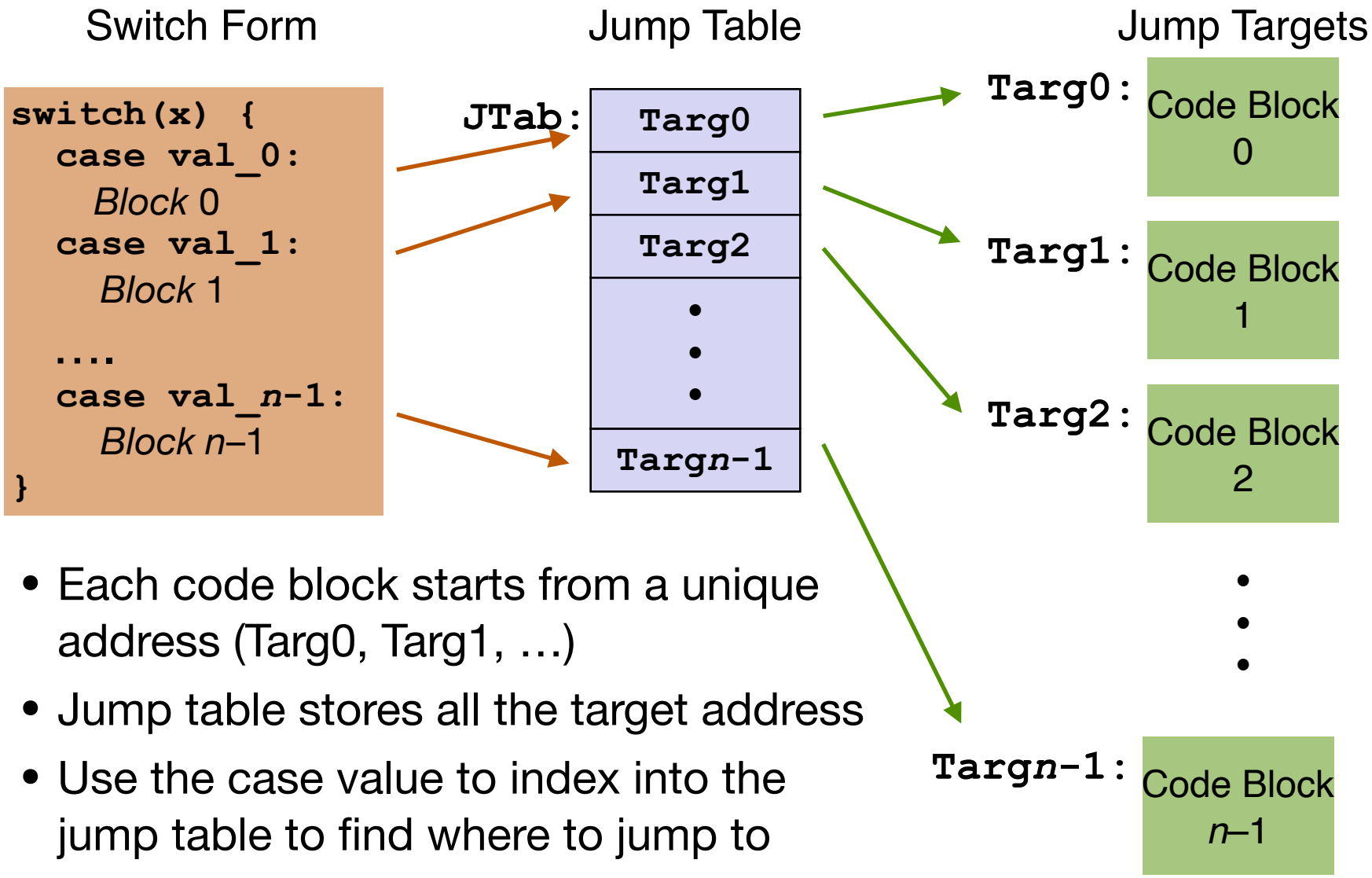
```
# assume x in %rdi
movq  .LJ(,%rdi,8), %rax
jmp   *%rax
```

- 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



- 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

Not The Only Way

- Jump table might not the most efficient implementation; certainly not the only way to implement switch-case.
- What if x can take a very large value range. Do we need to have a giant jump table?
- Let's say x can be any integer from 1 to 1 million, but anything between 8 and 1 million fall back to the default case. Can we avoid a 1 million entry jump table (which isn't too bad if you calculate the size)?
  - Have an if-else check first followed by an 8-entry table.

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

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

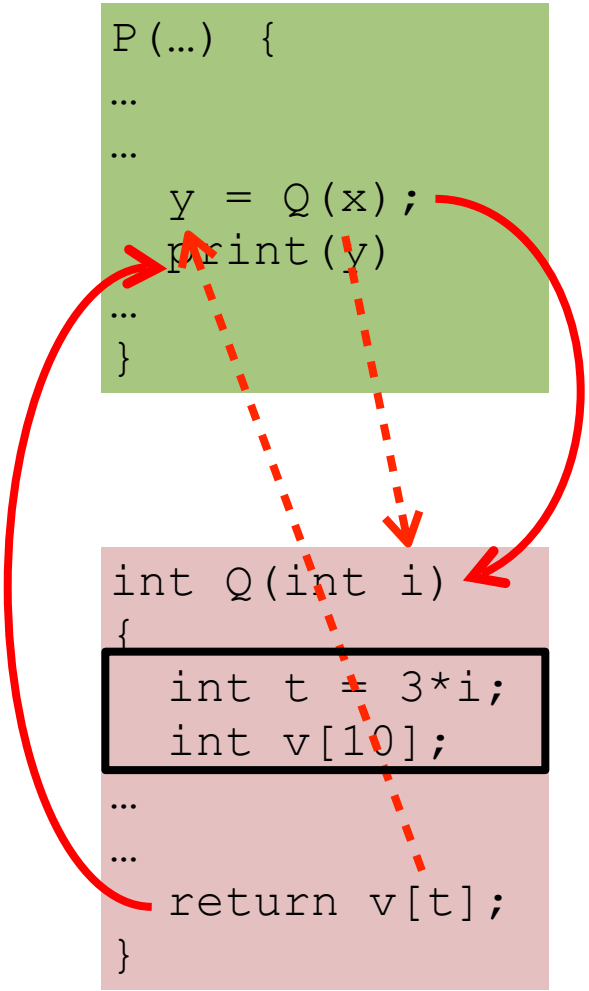
- Must match function declaration
- Implement the functionality of the function

```
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

- 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
- The ISA must provide ways (instructions) to realize all these.



Today: How to Implement Function Call

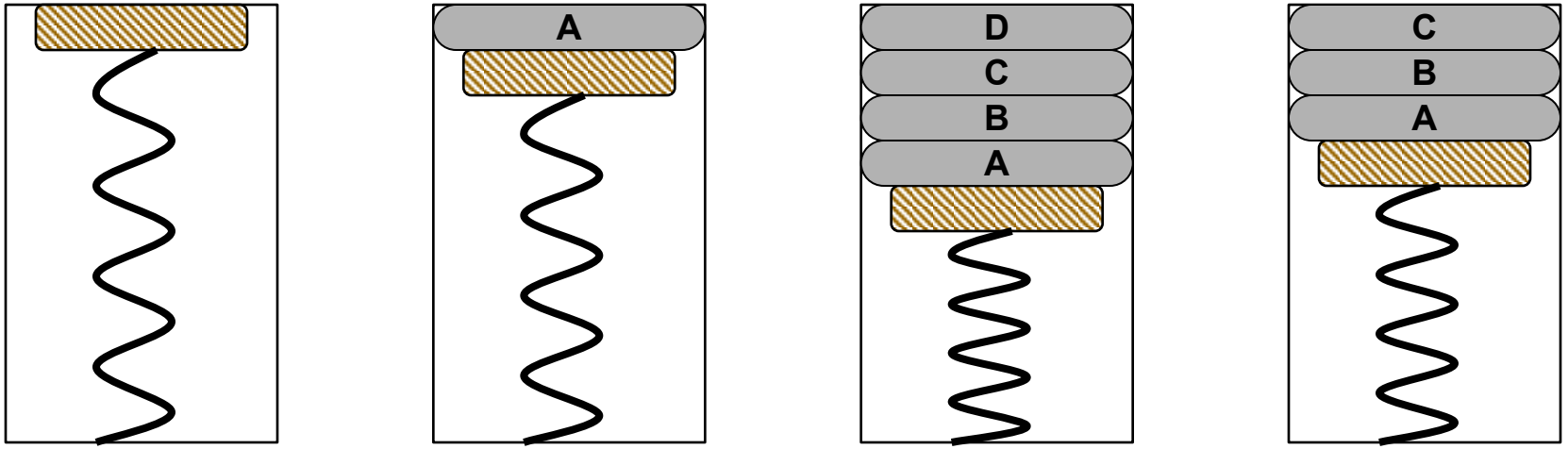
- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

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
- 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

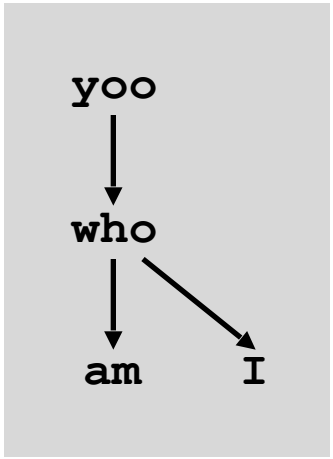
```
yoo(…)
{
   •
   •
   who();
   •
   return
}
```

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

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

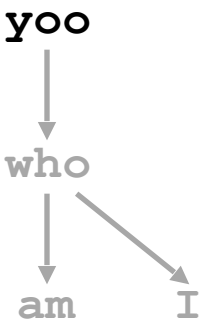
```
I(…)
{
   •
   •
   •
   return;
}
```

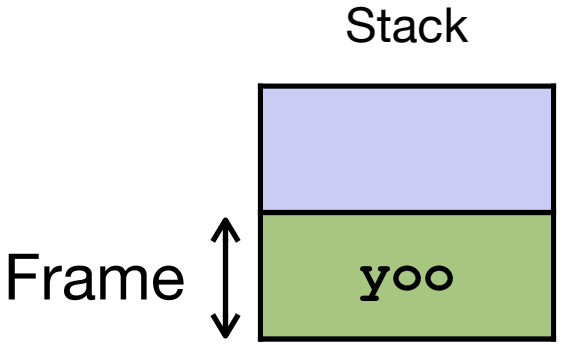
Example Call Chain



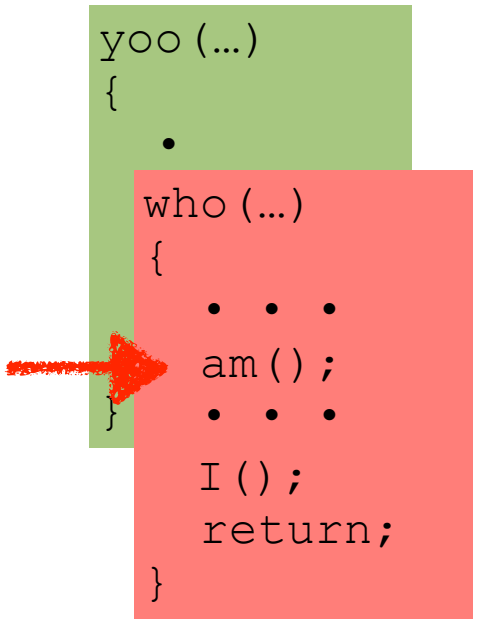
Run-Time Stack During Function Call

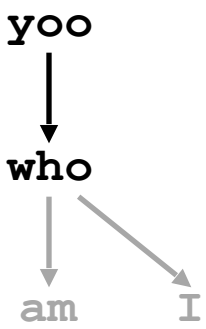
```
yoo(…)
{
   •
   •
   who();
   •
   return
}
```

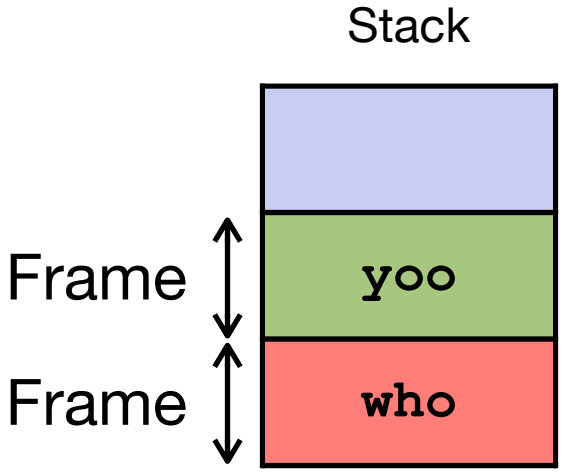




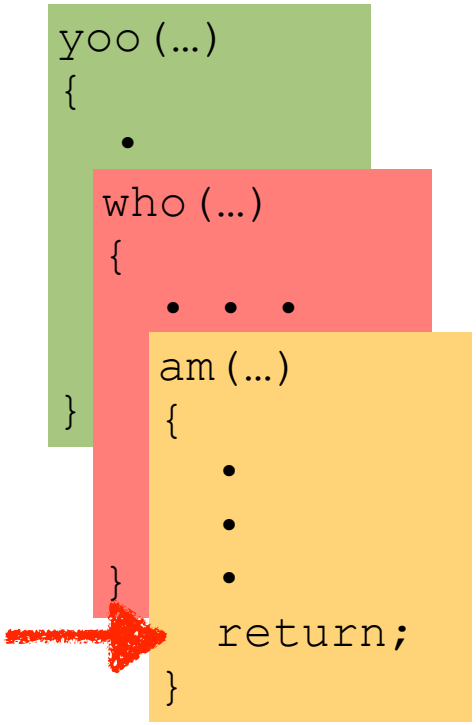
Run-Time Stack During Function Call

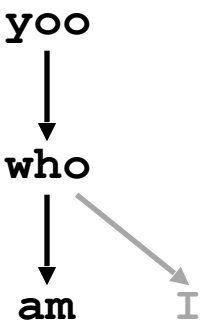


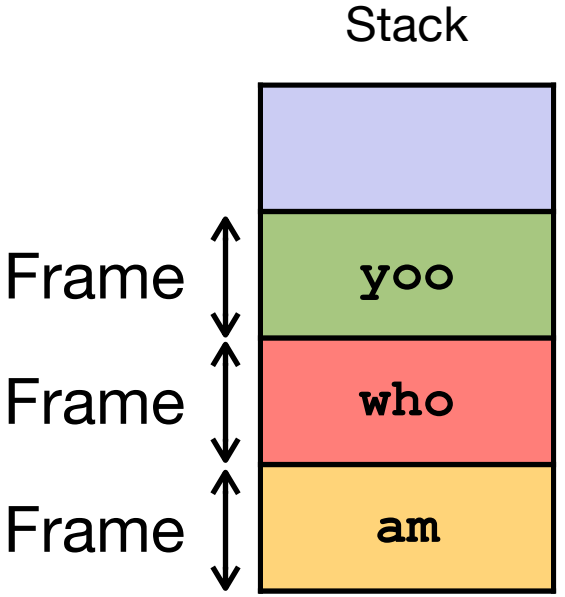




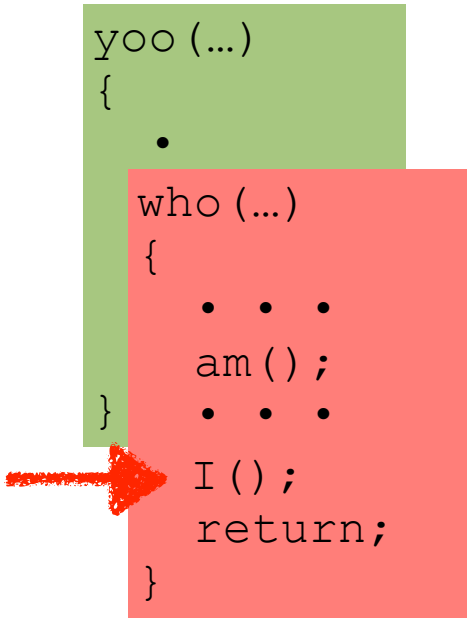
Run-Time Stack During Function Call

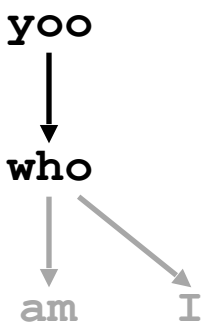


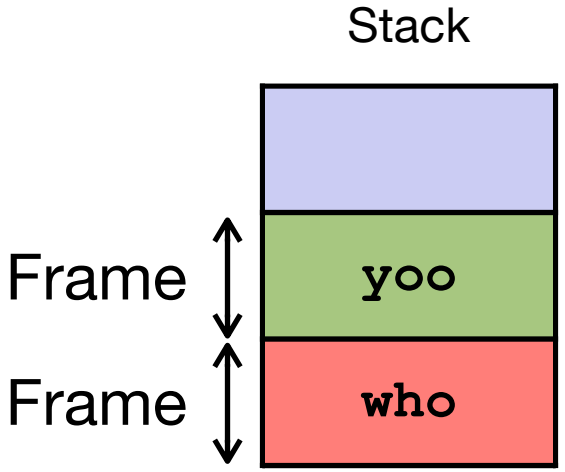




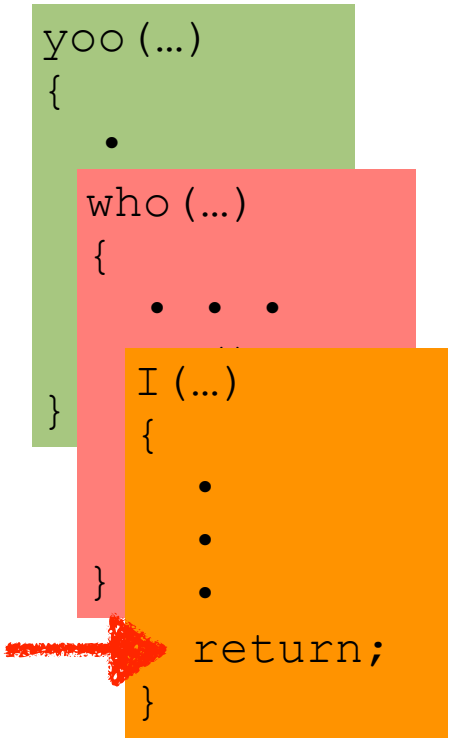
Run-Time Stack During Function Call

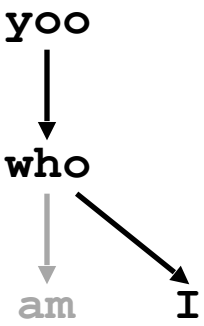


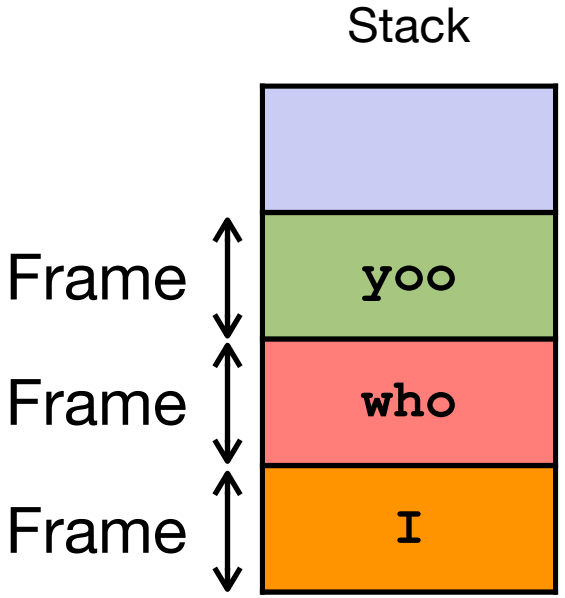




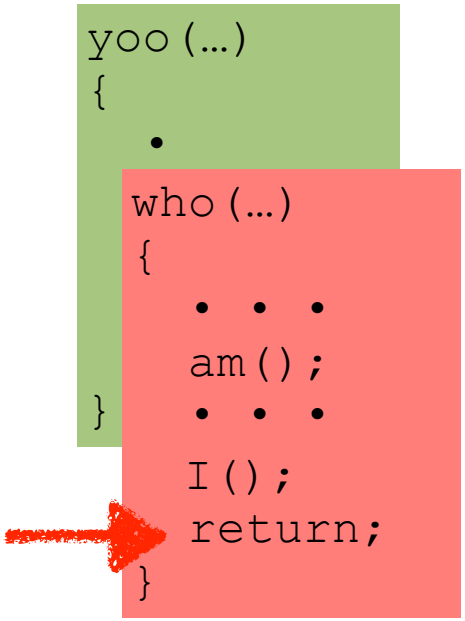
Run-Time Stack During Function Call

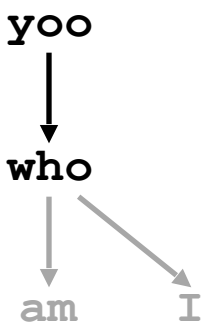


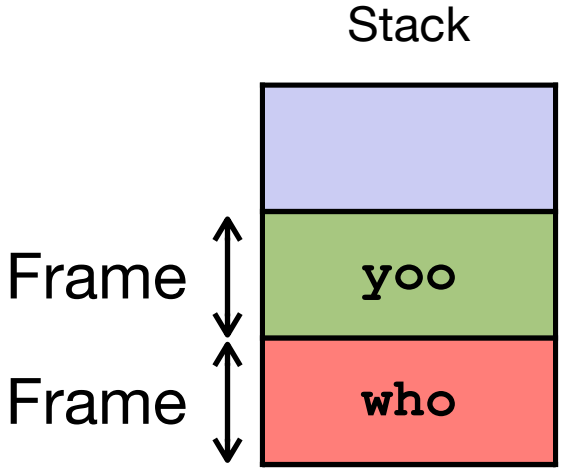




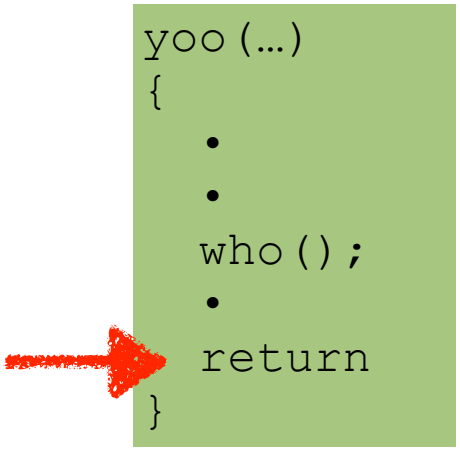
Run-Time Stack During Function Call

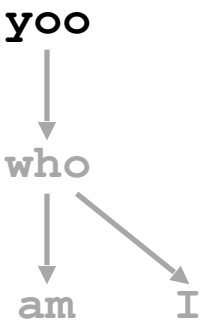


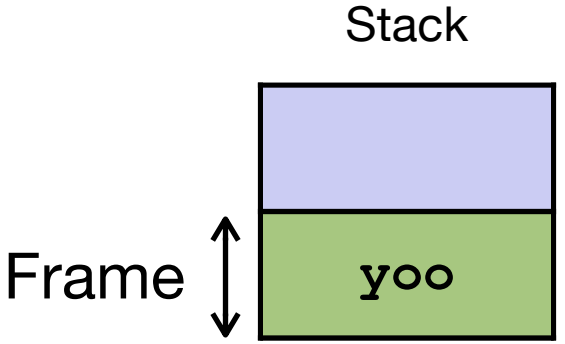




Run-Time Stack During Function Call

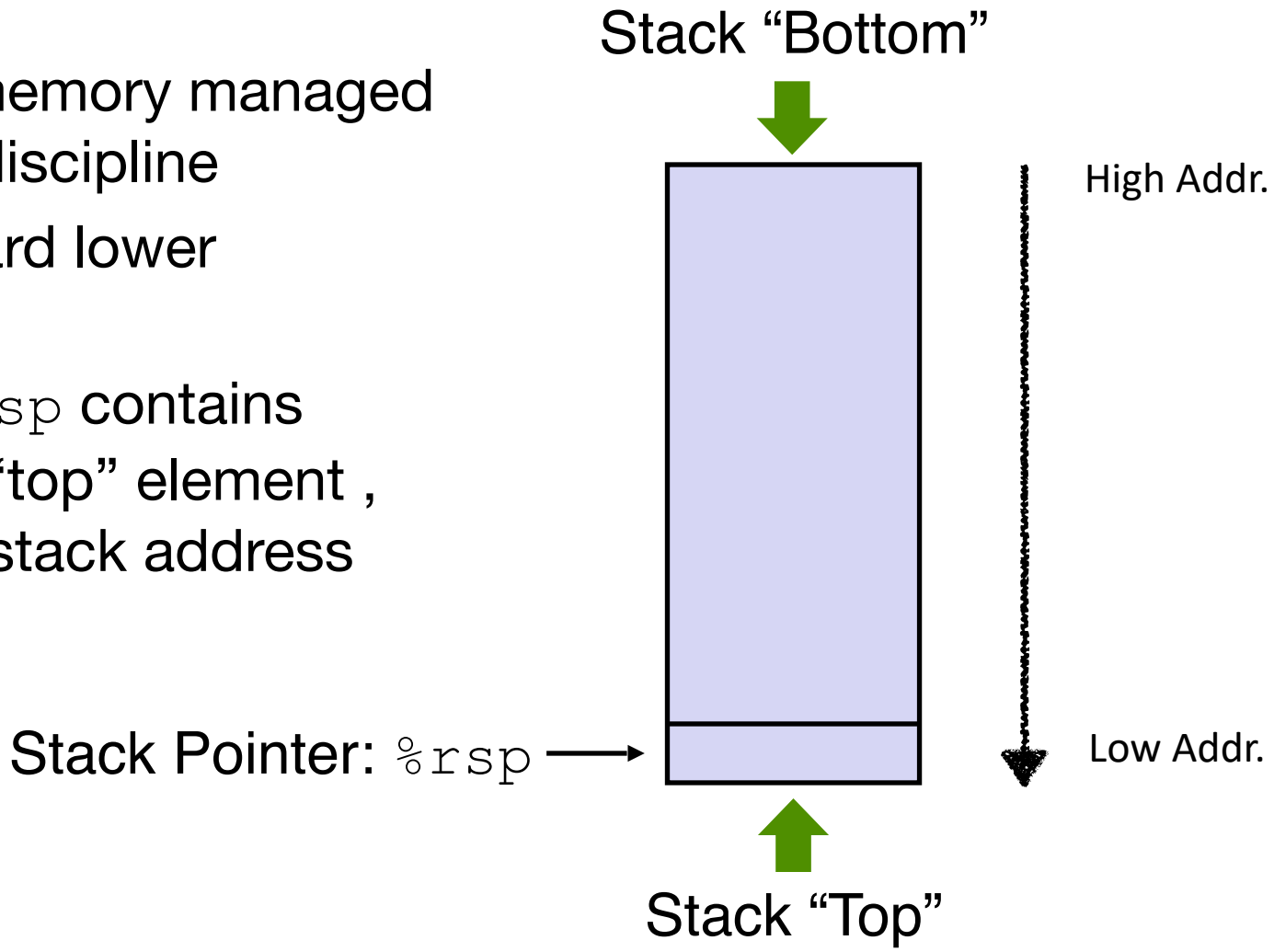






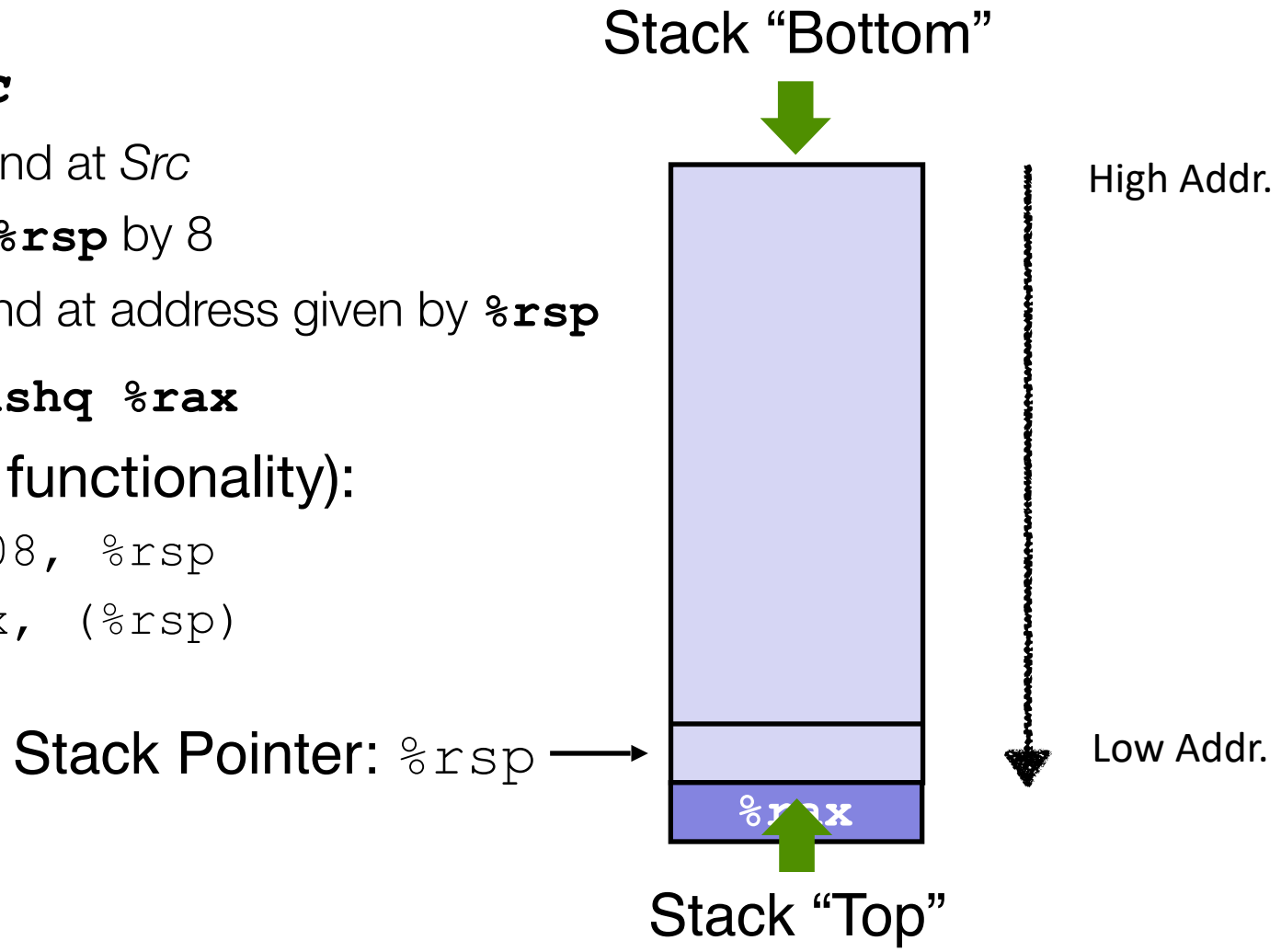
Stack in X86-64

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



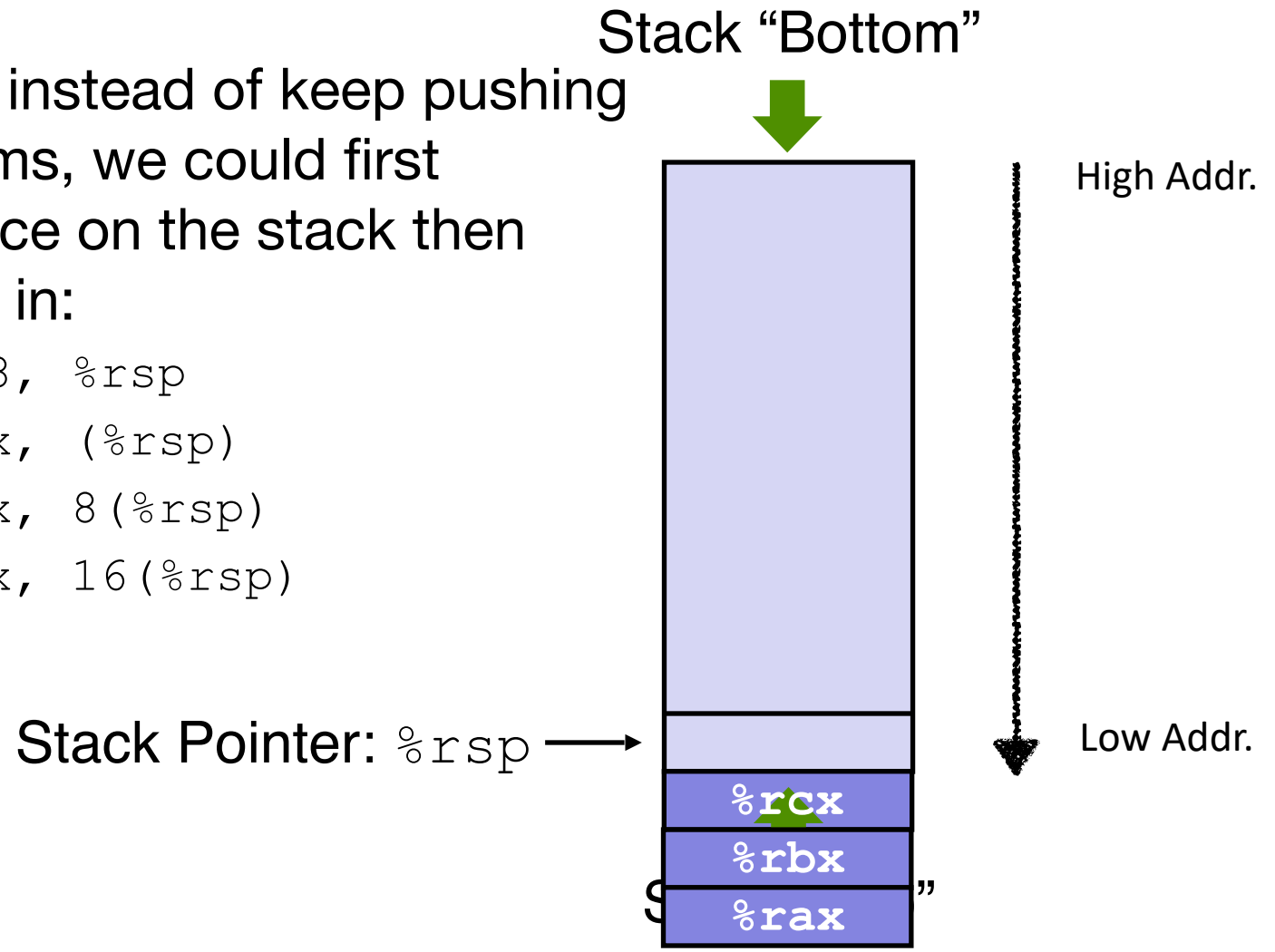
x86-64 Stack: Push

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



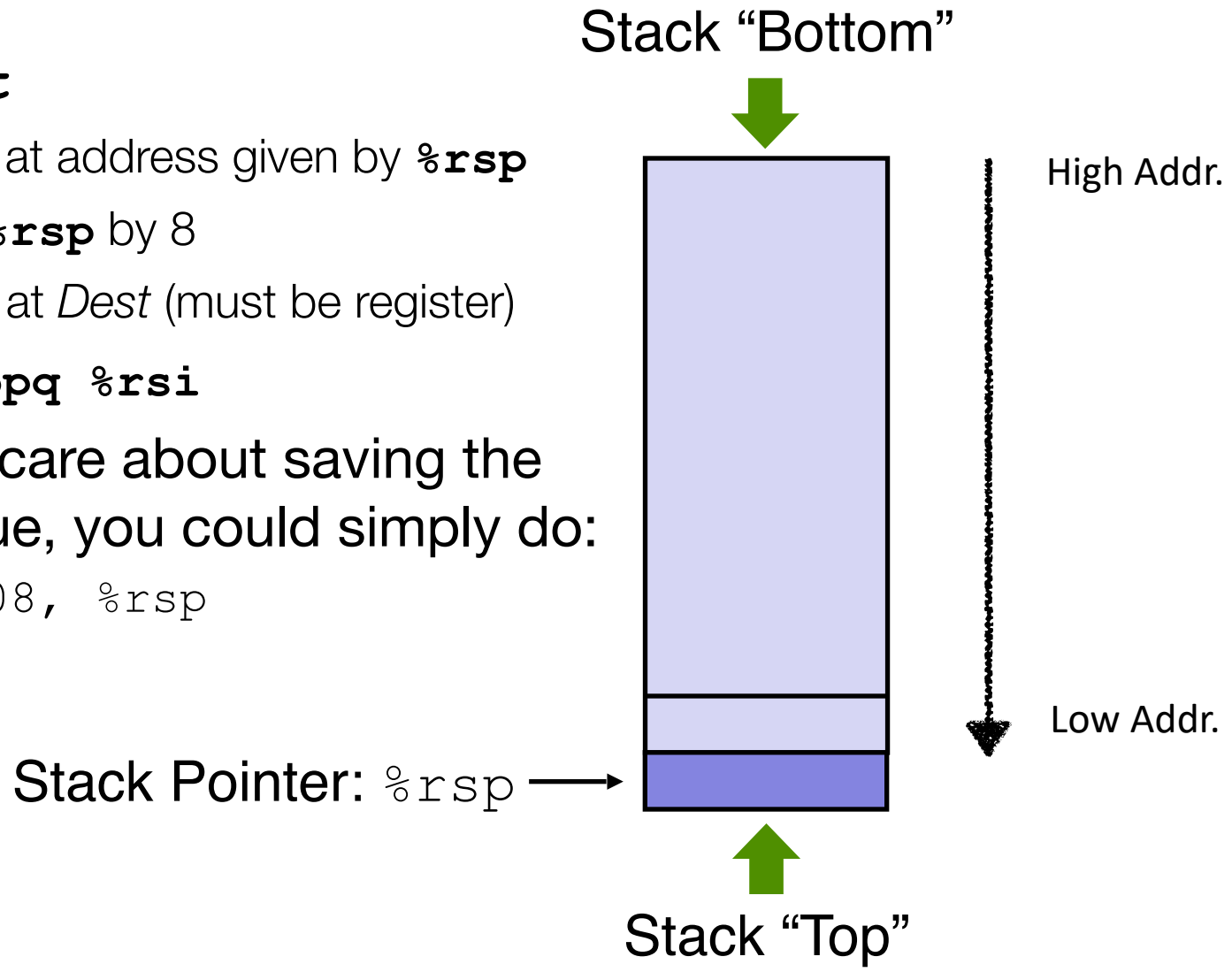
x86-64 Stack: Push

- 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: 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;
}
```

```
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`

- 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

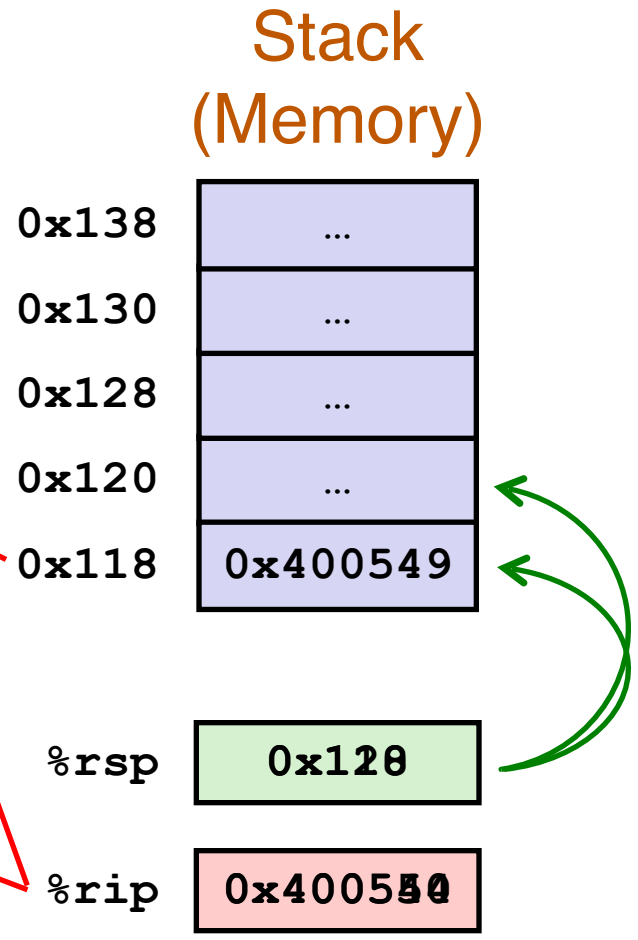
- Procedure call: `call label`
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call (`400549` here)
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

```
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)
…
…
```

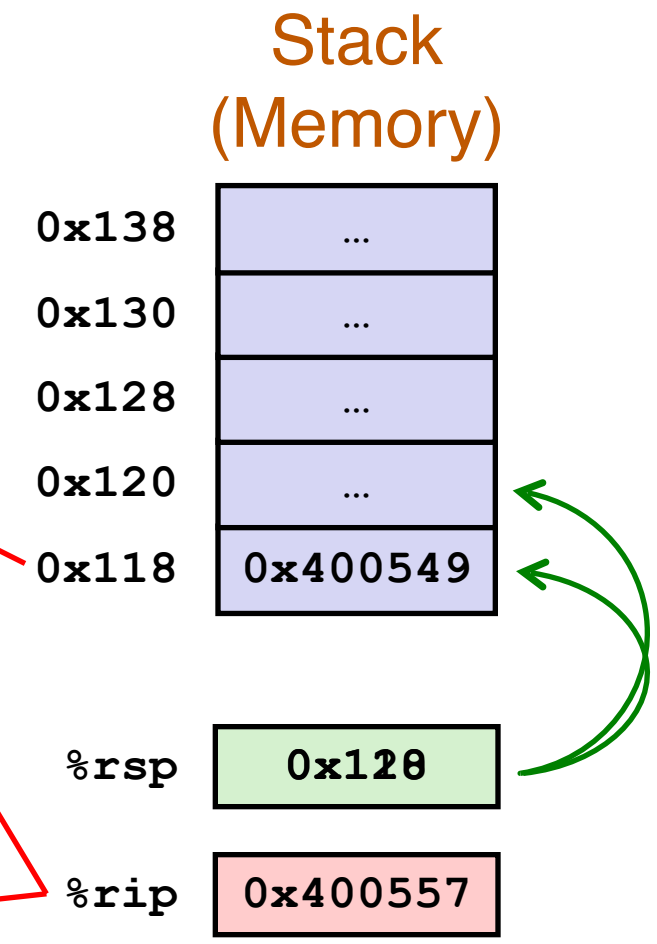
```
400550 <mult2>:
  400550:  mov    %rdi,%rax
…
…
  400557:  retq
```



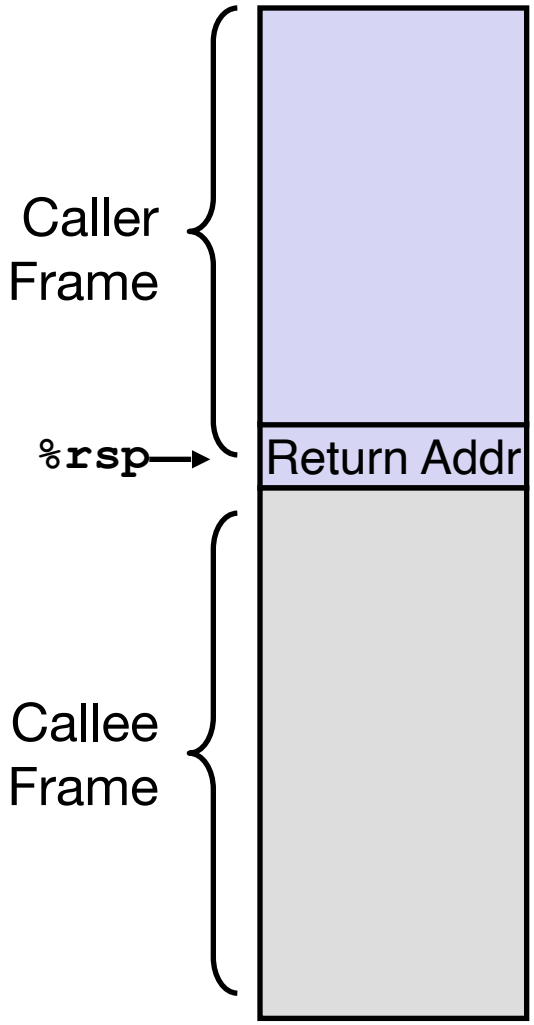
Function Call Example

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

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



Stack Frame (So Far…)

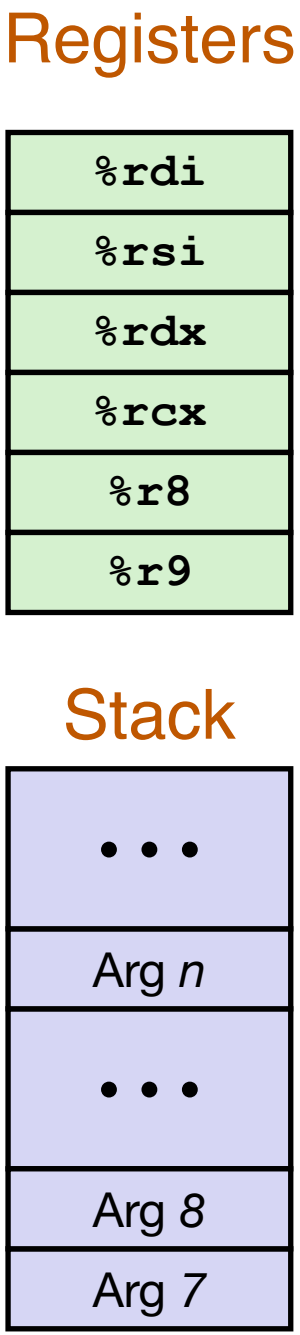


Today: How to Implement Function Call

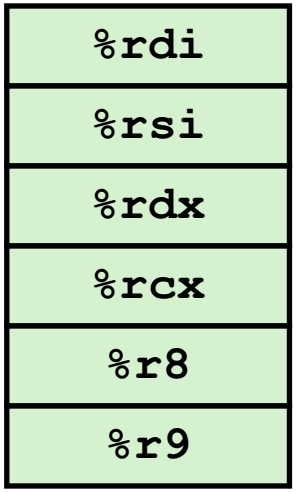
- What are functions and why do we use them?
- General idea of implementing functions: Stack
- Passing control
- Passing data
- Managing local data

Passing Function Arguments

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




```
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…)