photo by CGP Grey

COMP201

Computer Systems & Programming

Lecture #24 – x86-64 Procedures

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

Good news, everyone!

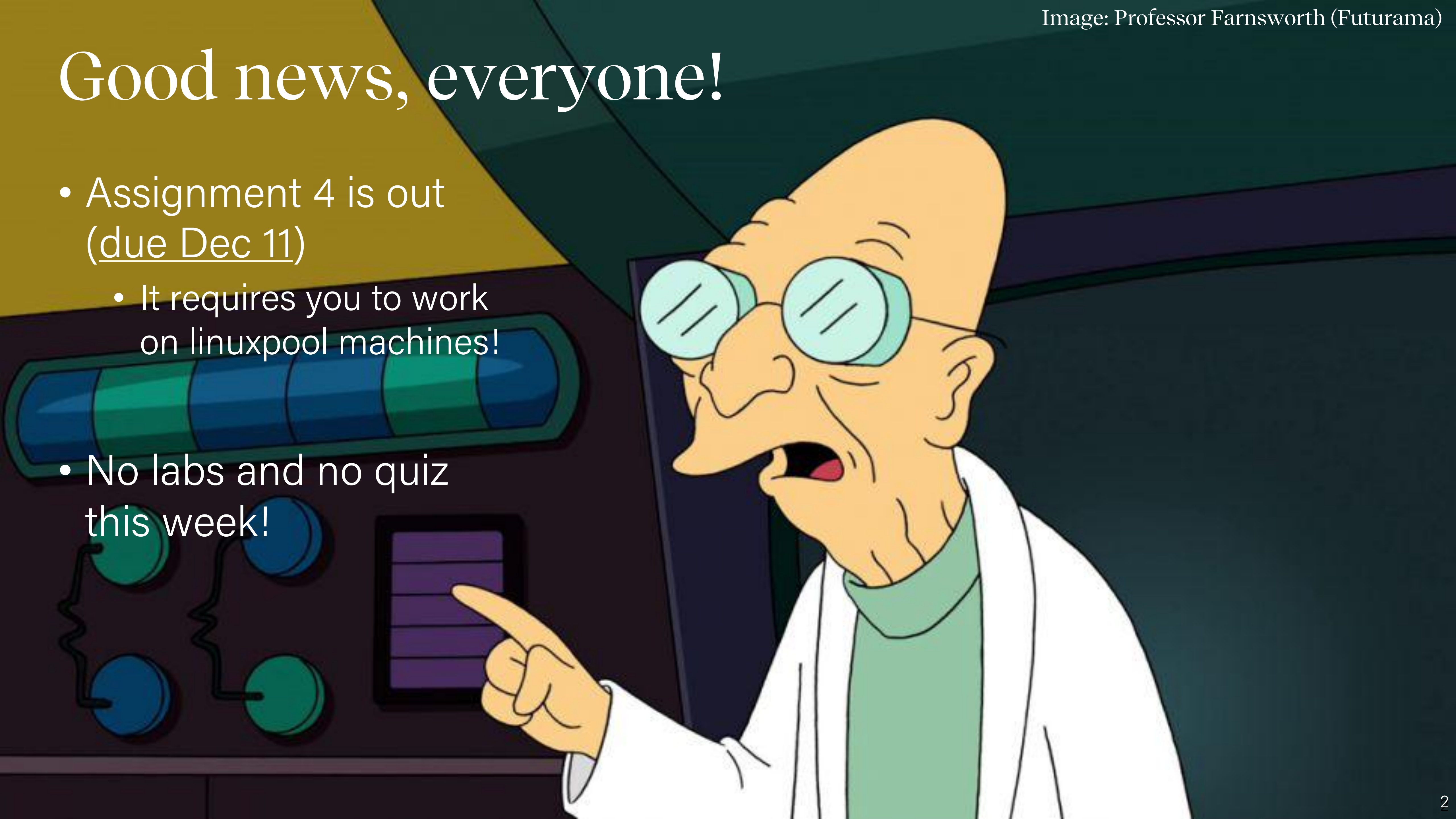


Image: Professor Farnsworth (Futurama)

- Assignment 4 is out (due Dec 11)
  - It requires you to work on linuxpool machines!
- No labs and no quiz this week!

Recap

- Assembly Execution and %rip
- Control Flow Mechanics
    - Condition Codes

        Assembly Instructions
- If statements
- Loops
    - While loops
    - For loops
- Other Instructions That Depend On Condition Codes

Practice 1: Fill In The Blank

Note: .L2/.L3 are “labels” that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = _______;
    while (_________) {
      result = ____________;
      a = _________;
    }
    return result;
}
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice 1: Fill In The Blank

slido Event code: 73165

Note: .L2/.L3 are "labels" that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = ____1____;
    while (__a < b__) {
      result = _result*(a+b)_;
      a = __a + 1__;
    }
    return result;
}
```

Common while loop construction:

```
Jump to test
Body
Test
Jump to body if success
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice 2: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

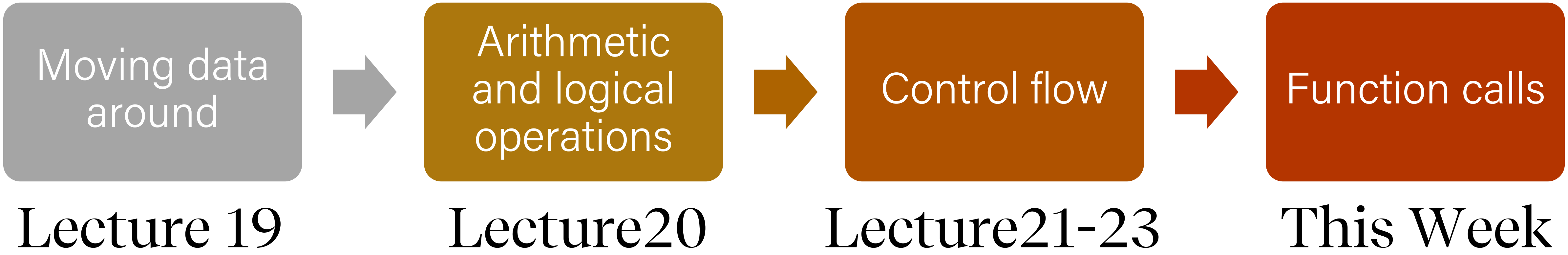
Practice 2: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Learning Assembly



Moving data around

Lecture 19

Arithmetic and logical operations

Lecture20

Control flow

Lecture21-23

Function calls

This Week

Learning Goals

- Learn how assembly calls functions and manages stack frames.
- Learn the rules of register use when calling functions.

Plan for Today

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

%rip

- **%rip** is a special register that points to the next instruction to execute.
- **Let's dive deeper into how %rip works, and how jumps modify it.**

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

These are 0-based offsets in bytes for each instruction relative to the start of this function.

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

These are bytes for the machine code instructions. Instructions are variable length.

%rip

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

%rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xeb means **jmp**.

%rip

```
0x400570 <+0>:   b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:   eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:   83 c0 01       add $0x1,%eax
0x40057a <+10>:  83 f8 63       cmp $0x63,%eax
0x40057d <+13>:  73 f8          jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3          repz retq
```

0x03 is the number of instruction bytes to jump relative to `%rip`.

With no jump, `%rip` would advance to the next line. This **`jmp`** says to then go **3** bytes further!

%rip

```
0x400570 <+0>:   b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:   eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:   83 c0 01       add $0x1,%eax
0x40057a <+10>:  83 f8 63       cmp $0x63,%eax
0x40057d <+13>:  73 f8          jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3          repz retq
```

0x03 is the number of instruction bytes to jump relative to `%rip`.

With no jump, `%rip` would advance to the next line. This **`jmp`** says to <u>then</u> go **3** bytes further!

%rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0x73 means **jle**.

%rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xf8 is the number of instruction bytes to jump relative to %rip. This is -8 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **8** bytes back!

%rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03          jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01       add $0x1,%eax
0x40057a <+10>: 83 f8 63       cmp $0x63,%eax
0x40057d <+13>: 73 f8          jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3          repz retq
```

0xf8 is the number of instruction bytes to jump relative to `%rip`. This is -8 (in two's complement!).

With no jump, `%rip` would advance to the next line. This **`jmp`** says to then go **8** bytes back!

Summary: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.
- **`%rip`** is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **`jmp`** instructions work by adjusting **`%rip`** by a specified amount.

Question Break

Lecture Plan

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

How do we call functions in assembly?

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

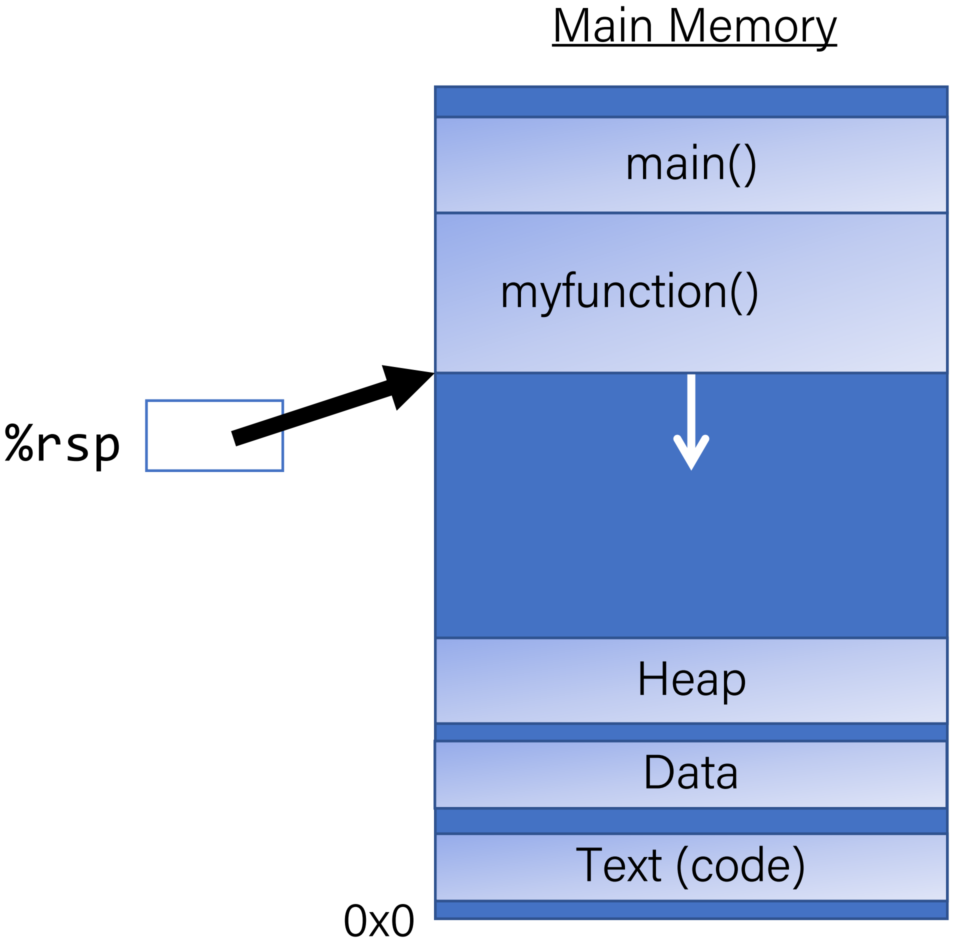
Terminology: **caller** function calls the **callee** function.

Lecture Plan

- Revisiting %rip
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

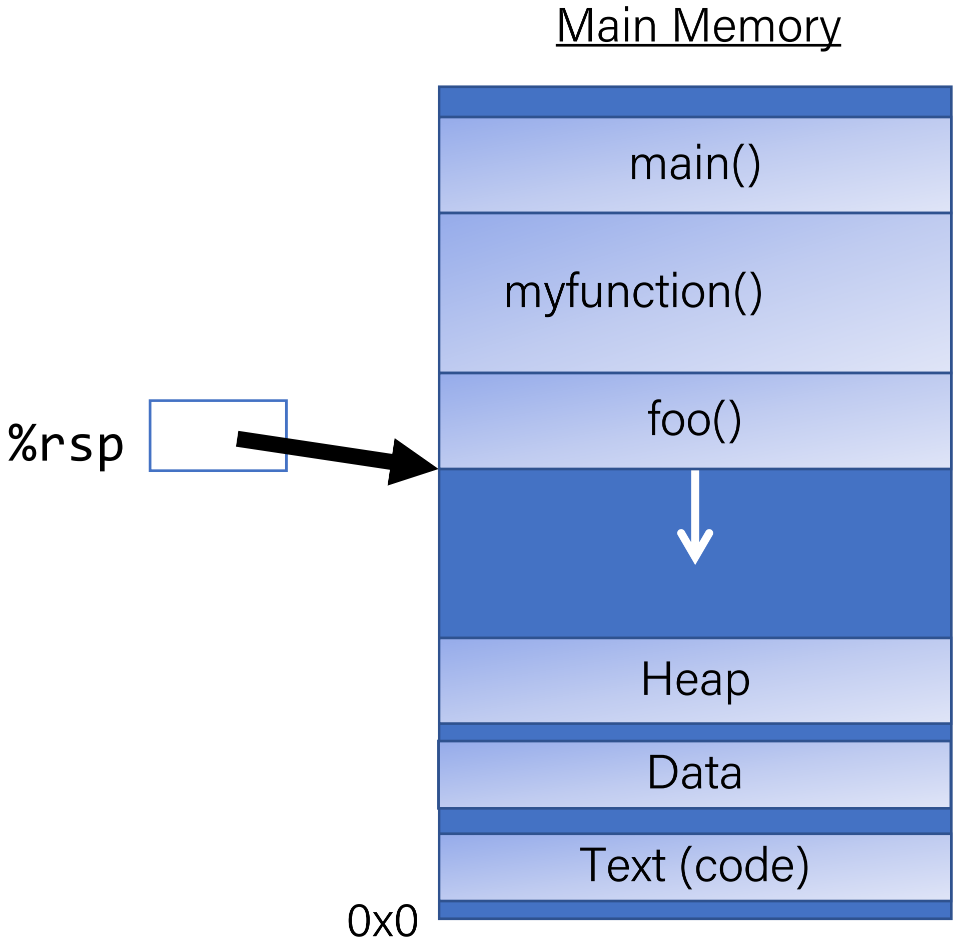
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



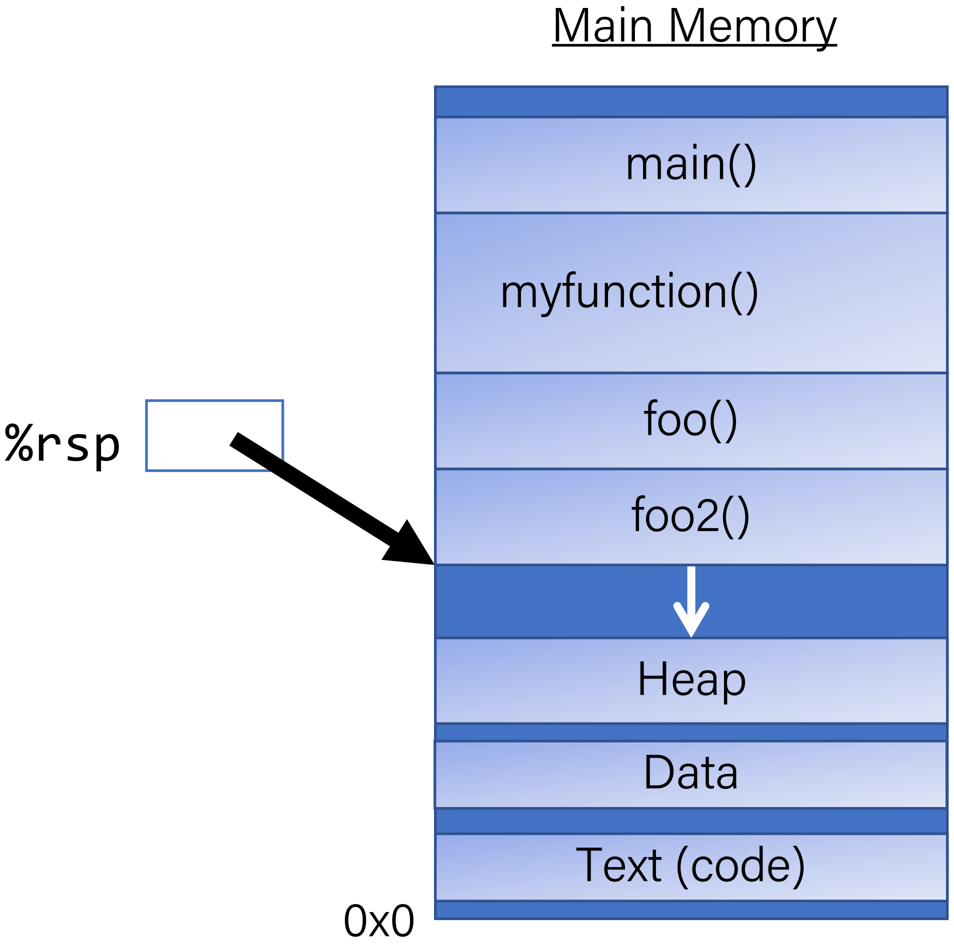
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



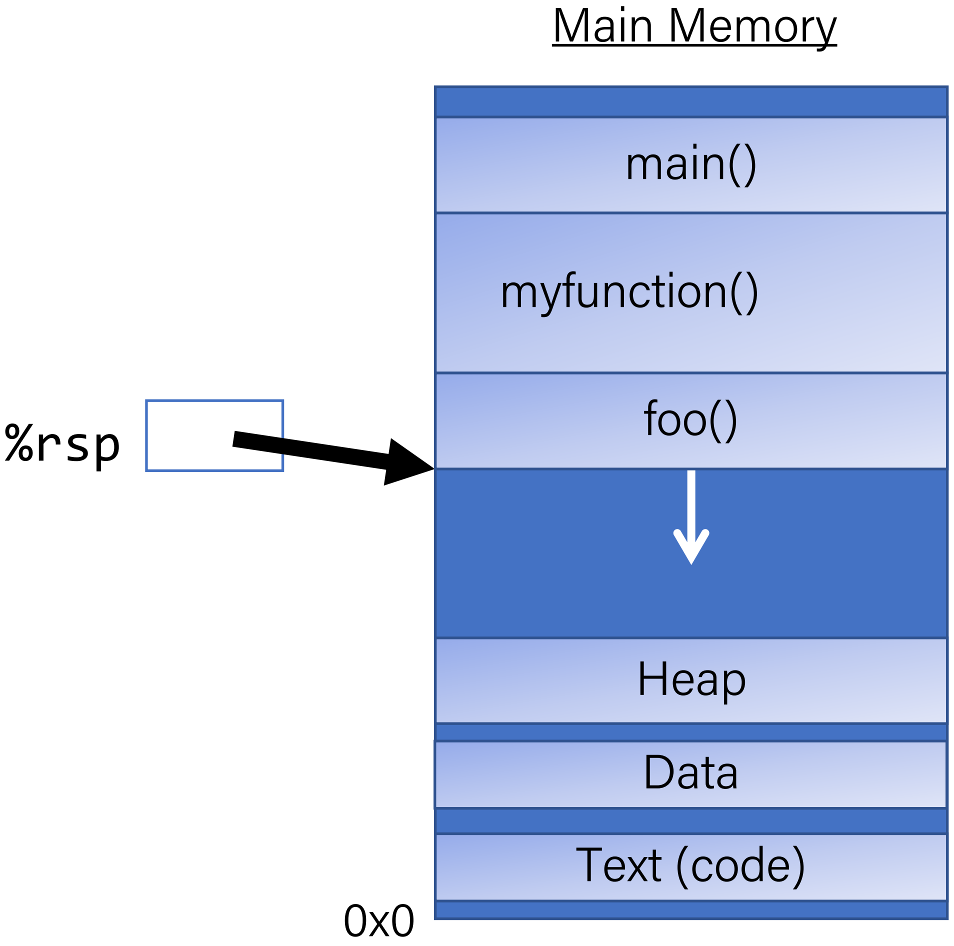
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



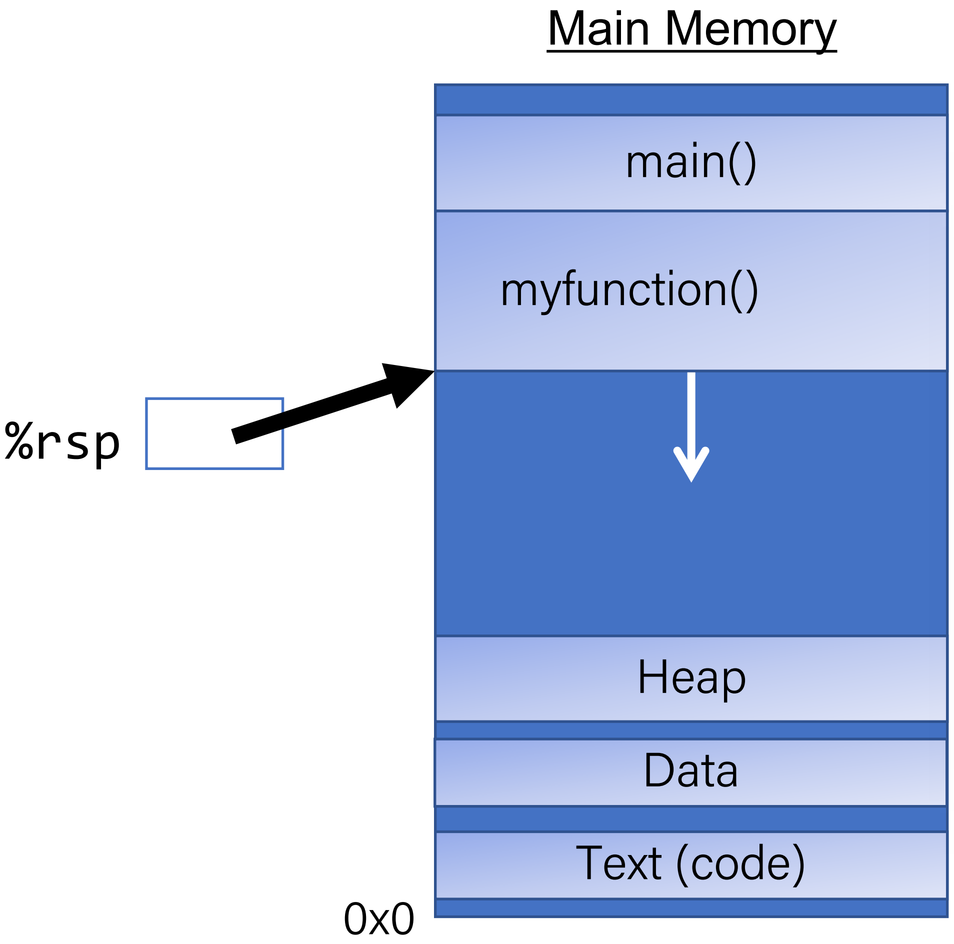
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	R[%rsp] ⟵ R[%rsp] - 8; M[R[%rsp]] ⟵ S

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	**R[%rsp] ⟵ R[%rsp] - 8;** M[R[%rsp]] ⟵ S

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect

`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;`
	`M[R[%rsp]] ⟵ S`

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
pushq S	R[%rsp] ⟵ R[%rsp] - 8; M[R[%rsp]] ⟵ S

- This behavior is equivalent to the following, but **pushq** is a shorter instruction:

```
 subq $8, %rsp
movq  S, (%rsp)
```

- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for future data. More on this later!

pop

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **`%rsp`** accordingly.

Instruction	Effect

`popq D`	`D ⟵ M[R[%rsp]]`
	`R[%rsp] ⟵ R[%rsp] + 8;`

- **Note:** this *does not* remove/clear out the data! It just increments `%rsp` to indicate the next push can overwrite that location.

pop

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **`%rsp`** accordingly.

Instruction	Effect

`popq D`	`D ⟵ M[R[%rsp]]`
	`R[%rsp] ⟵ R[%rsp] + 8;`

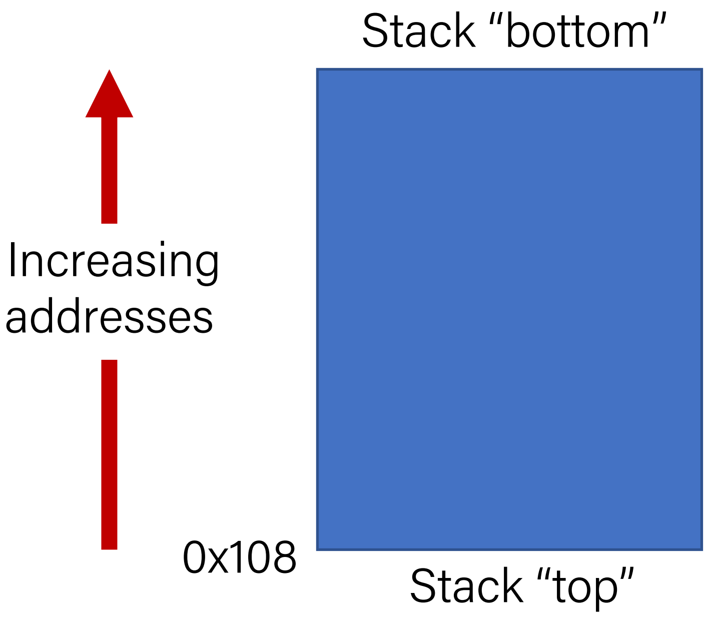
- This behavior is equivalent to the following, but **popq** is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

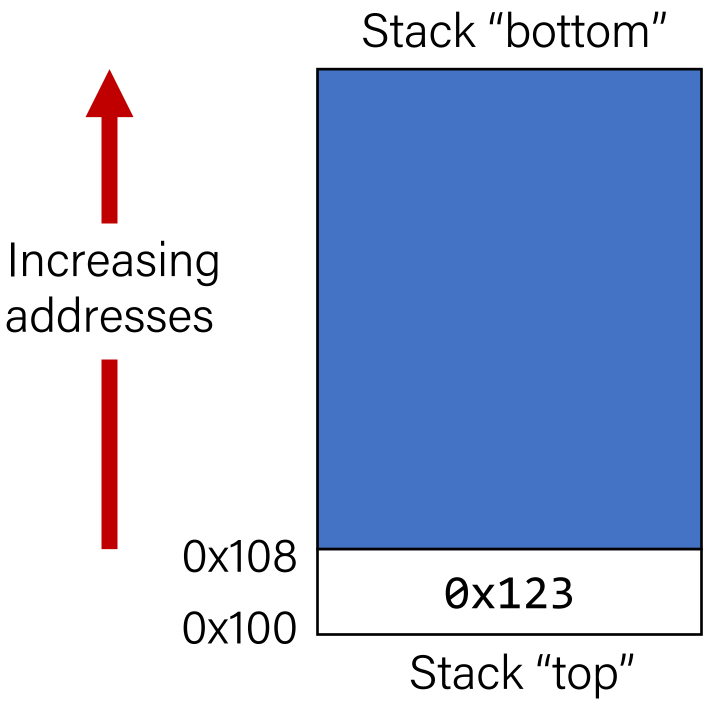
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

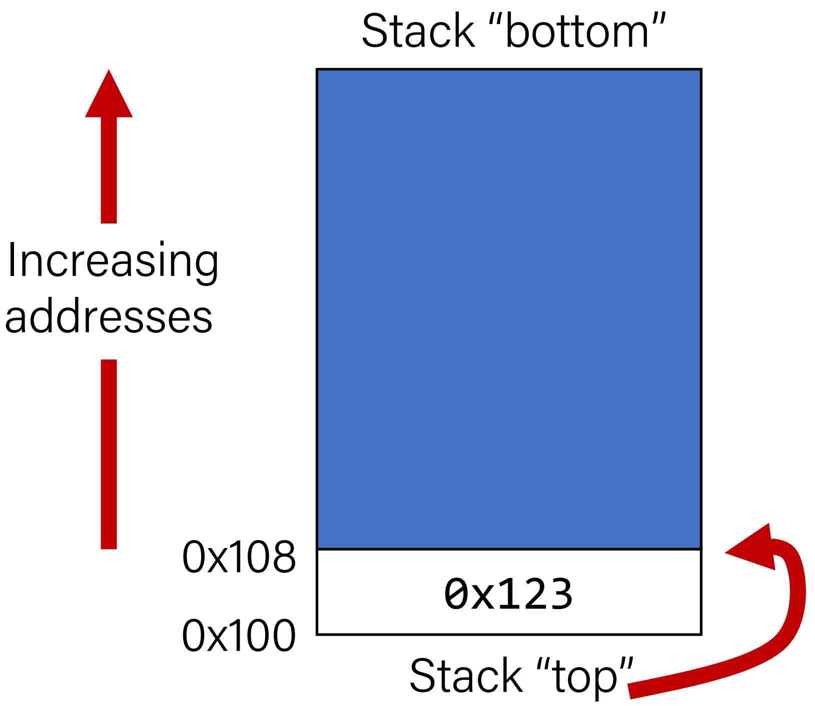
Initially	
%rax	0x123
%rdx	0
%rsp	0x108



pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100



popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108



Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Question Break

Lecture Plan

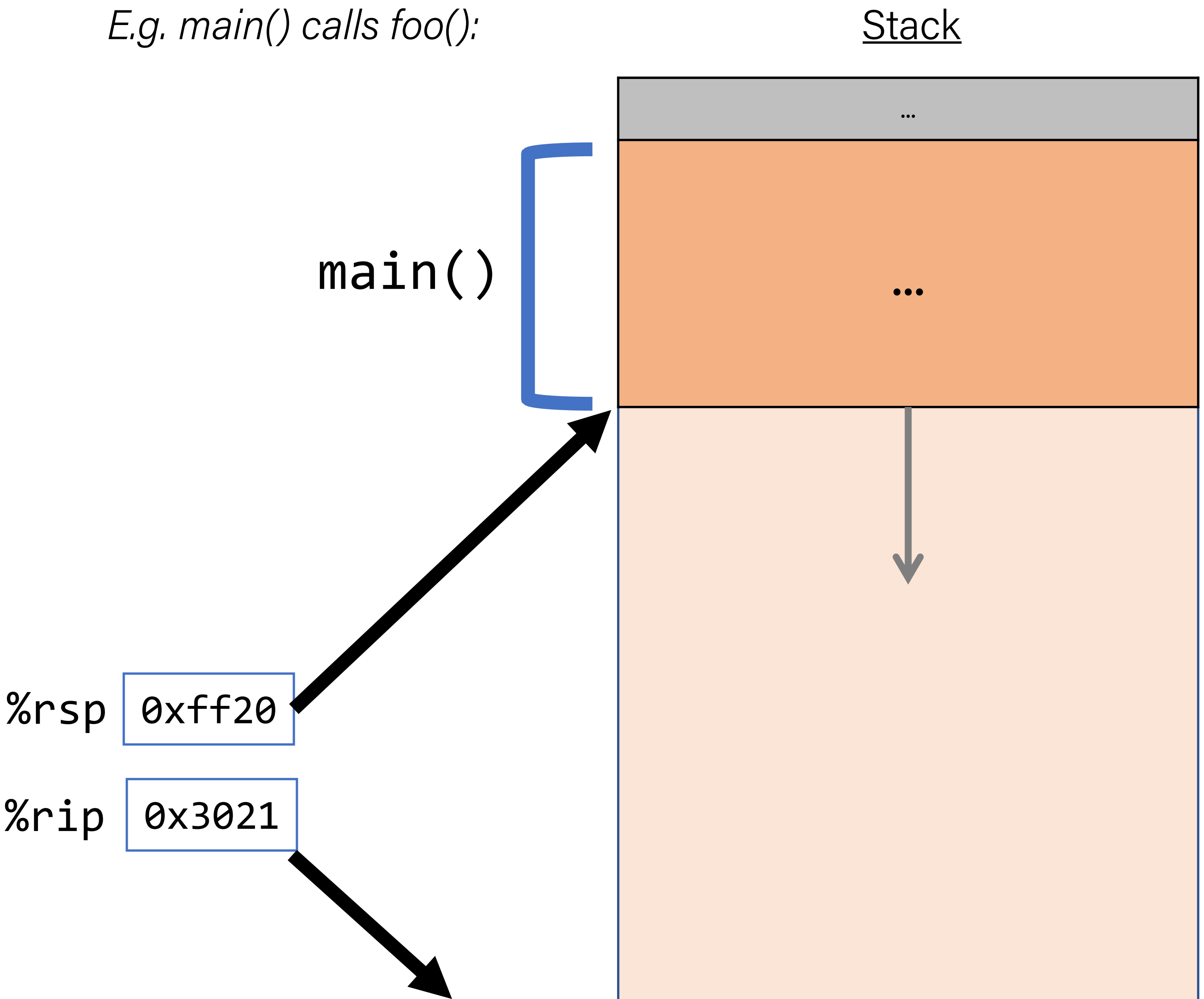
- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

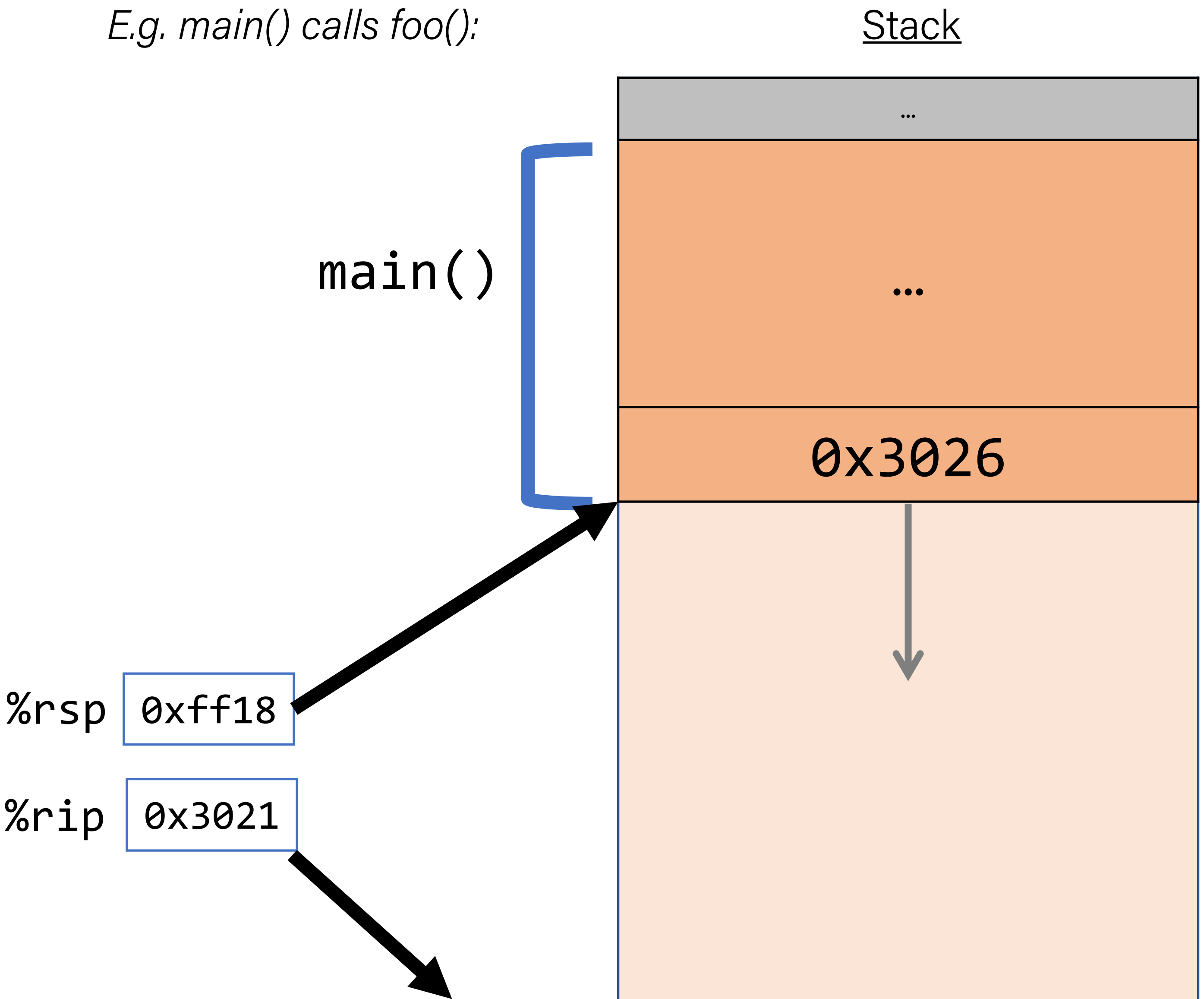


Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

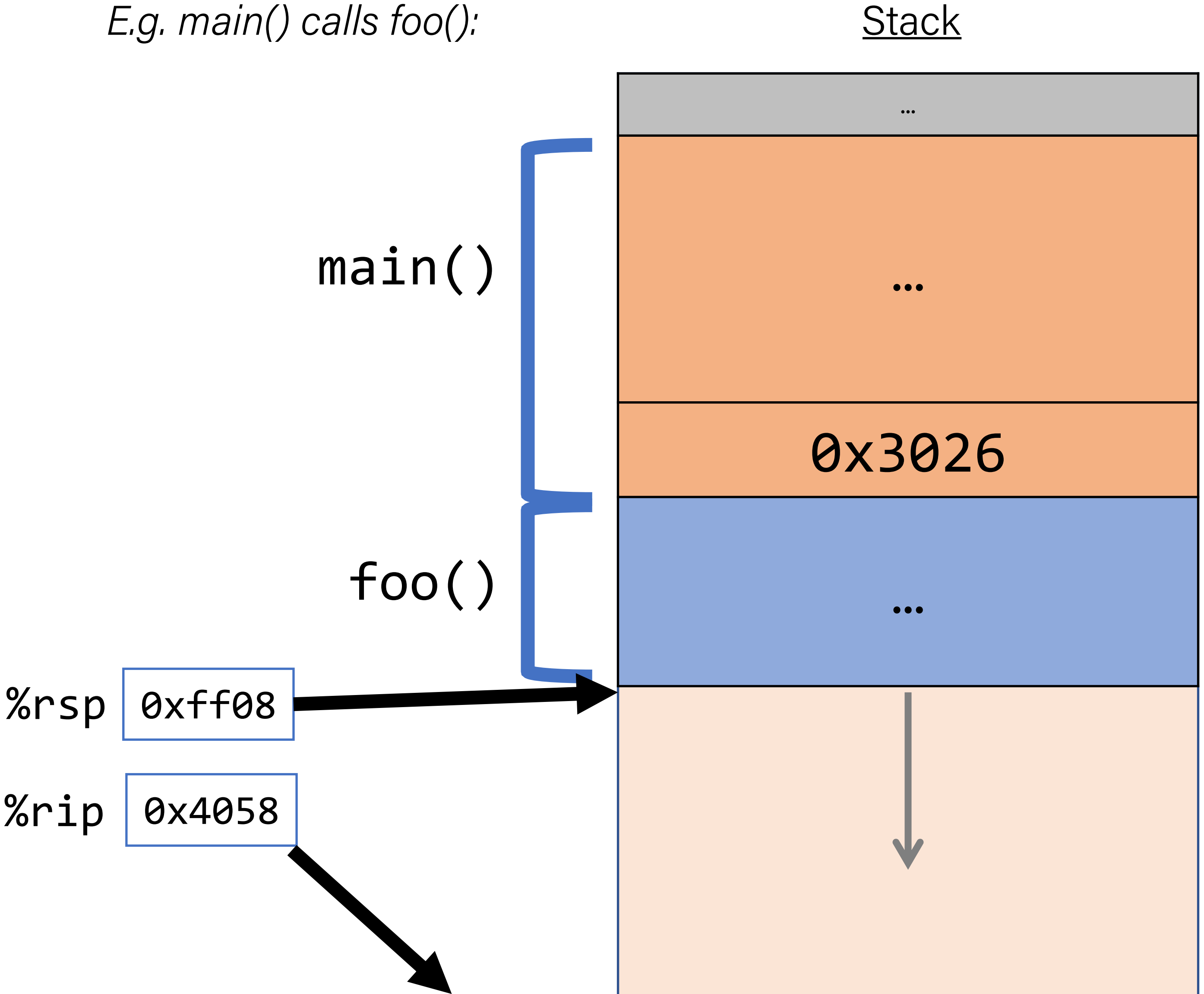


Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

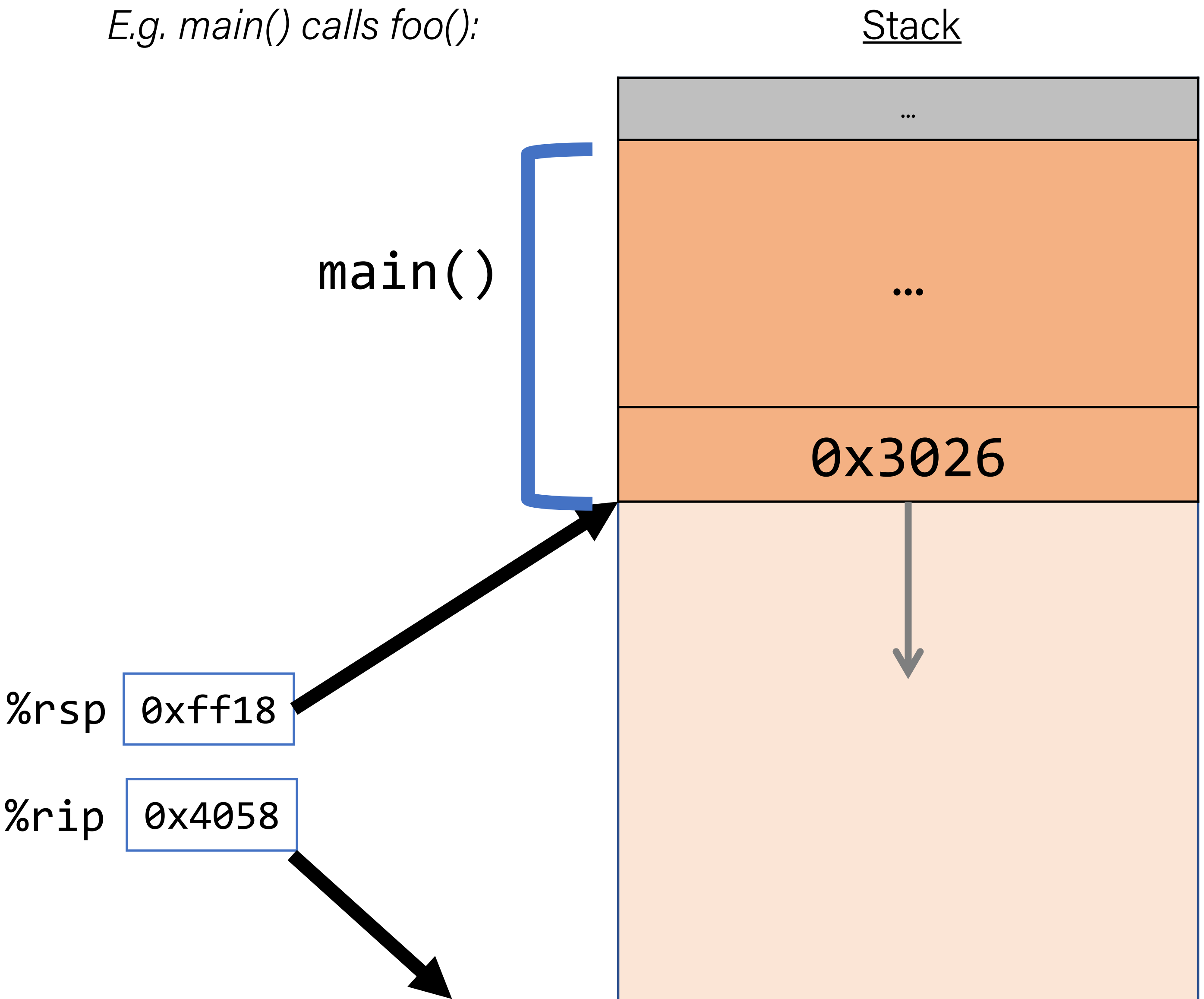


Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():

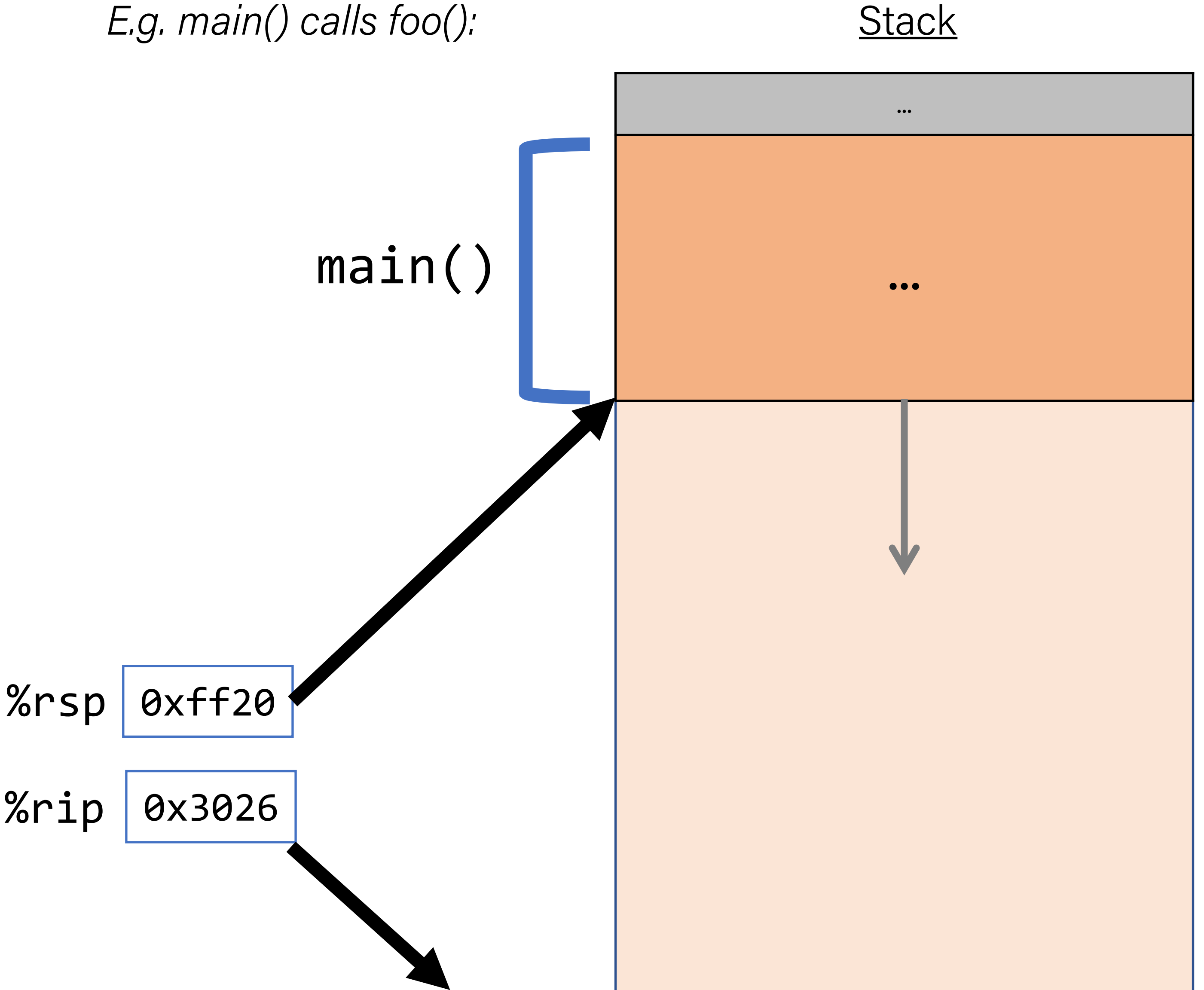


Remembering Where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.

E.g. main() calls foo():



Call And Return

The **`call`** instruction pushes the address of the instruction immediately following the **`call`** instruction onto the stack and sets `%rip` to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

The **`ret`** instruction pops this instruction address from the stack and stores it in `%rip`.

```
ret
```

The stored `%rip` value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

What’s left? Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Recap

- Revisiting `%rip`
- Calling Functions
    - The Stack
    - Passing Control

Next time: passing data, local storage, register restrictions