CS107, Lecture 10
Introduction to Assembly

Reading: B&O 3.1-3.4

What is Assembly Code?

- Computers execute "machine code," which is a sequence of bytes that encode low-level operations for manipulating data, managing memory, read and write from storage, and communicate with networks.

- The "assembly code" for a computer is a textual representation of the machine code giving the individual instructions to the underlying machine.

What is Assembly Code?

- `gcc` generates assembly code from C code
- Assembly is raw — there is no type checking, and the instructions are simple. It is unique to the type of processor (e.g., the assembly for your computer cannot run on your phone)
- Humans can write assembly (and, in fact, in the early days of computing they had to write assembly), but it is more productive to be able to read and understand what the compiler produces, than to write it by hand.
- `gcc` is almost always going to produce better optimized code than a human could, and understanding what the compiler produces is important.

x86 Assembly

- The Intel-based computers we use are direct descendants of Intel's 16-bit, 1978 processor with the name 8086.
- Intel has taken a strict backwards-compatibility approach to new processors, and their 32- and 64-bit processors have built upon the original 8086 Assembly code.
- These days, when we learn x86 assembly code, we have to keep this history in mind. Naming of "registers," for example, has historical roots, so bear with it.

Machine-Level Code

- Before we look at some assembly code, let's talk about some things that have been hidden from us when writing C code.
- Machine code is based on the "instruction set architecture" (ISA), which defines the behavior and layout of the system. Behavior is defined as if instructions are run one after the other, and memory appears as a very large byte array.

Machine-Level Code

- New things that have been hidden:
  - The *program counter* (PC), called "`%rip`" indicates the address of the next instruction ("r"egister "i"nstruction "p"ointer". We cannot modify this directly.
  - The "register file" contains 16 named locations that store 64-bit values. Registers are the fastest memory on your computer. They *are not* in main memory, and *do not* have addresses. You cannot pass a pointer to a register, but a pointer may *hold* a register as its value.
    - Registers can hold addresses, or integer data. Some registers are used to keep track of your program's state, and others hold temporary data.
    - Registers are used for arithmetic, local variables, and return values for functions.
  - The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to control program flow — e.g., if the result of an addition is negative, exit a loop.
  - There are vector registers, which hold integer or floating point values.

Machine-Level Code

- Unlike C, there is no model of different data types, and memory is simply a large, byte-addressable array.

- There is no distinction between signed and unsigned integers, between different types of pointers, or even between pointers and integers.

- A single machine instruction performs only a very elementary operation. For example:
  - there is an instruction to add two numbers in registers. That's all the instruction does.
  - there is an instruction that transfers data between a register and memory.
  - there is an instruction that conditionally branches to a new instruction address.

- Often, one C statement generates multiple assembly code instructions.

Learning Goals

- Learn what assembly language is and why it is important
- Become familiar with the format of human-readable assembly and x86
- Learn the **mov** instruction and how data moves around at the assembly level

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

Bits all the way down

Data representation so far

- Integer (`unsigned int`, 2's complement signed `int`)
- `char` (ASCII)
- Address (unsigned long)
- Aggregates (arrays, `structs`)

The code itself is binary too!

- Instructions (machine encoding)

GCC

- **GCC** is the compiler that converts your human-readable code into machine-readable instructions.
- C, and other languages, are high-level abstractions we use to write code efficiently. But computers don't really understand things like data structures, variable types, etc. Compilers are the translator!
- Pure machine code is 1s and 0s – everything is bits, even your programs! But we can read it in a human-readable form called **assembly**. (Engineers used to write code in assembly before C).
- There may be multiple assembly instructions needed to encode a single C instruction.
- We're going to go behind the curtain to see what the assembly code for our programs looks like.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

Demo: Looking at an Executable (objdump -d)

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

What does this look like in assembly?

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

make
objdump -d sum

```
0000000000401136 <sum_array>:
  401136:    b8 00 00 00 00      mov    $0x0,%eax
  40113b:    ba 00 00 00 00      mov    $0x0,%edx
  401140:    39 f0               cmp    %esi,%eax
  401142:    7d 0b               jge    40114f <sum_array+0x19>
  401144:    48 63 c8            movslq %eax,%rcx
  401147:    03 14 8f            add    (%rdi,%rcx,4),%edx
  40114a:    83 c0 01            add    $0x1,%eax
  40114d:    eb f1               jmp    401140 <sum_array+0xa>
  40114f:    89 d0               mov    %edx,%eax
  401151:    c3                  retq
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00          mov     $0x0,%eax
  40113b:   ba 00 00 00 00          mov     $0x0,%edx
  401140:   39 f0                   cmp     %esi,%eax
  401142:   7d 0b                   jge     40114f <sum_array+0x19>
  401144:   48 63 c8                movslq  %eax,%rcx
  401147:   03 14 8f                add     (%rdi,%rcx,4),%edx
  40114a:   83 c0 01                add     $0x1,%eax
  40114d:   eb f1                   jmp     401140 <sum_array+0xa>
  40114f:   89 d0                   mov     %edx,%eax
  401151:   c3                      retq
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
                                 p      %esi,%eax
                                 e      40114f <sum_array+0x19>
                                 vslq   %eax,%rcx
                                 d      (%rdi,%rcx,4),%edx
  40114a:  83 c0 01              add    $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```

This is the name of the function (same as C) and the memory address where the code for this function starts.

Our First Assembly

```
0000000000401136 <sum_array>:
  401136: b8 00 00 00 00          mov    $0x0,%eax
  40113b: ba 00 00 00 00          mov    $0x0,%edx
  401140: 39 f0                   cmp    %esi,%eax
  401142: 7d                                  array+0x19>
  401144: 48
  401147: 03                                  4),%edx
  40114a: 83
  40114d: eb f1                   jmp    401140 <sum_array+0xa>
  40114f: 89 d0                   mov    %edx,%eax
  401151: c3                      retq
```

These are the memory addresses where each of the instructions live. Sequential instructions are sequential in memory.

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00
  40113b:   ba 00 00 00 00
  401140:   39 f0
  40114d:   eb f1
  40114f:   89 d0
  401151:   c3
```

This is the assembly code: "human-readable" versions of each machine code instruction.

```
mov     $0x0,%eax
mov     $0x0,%edx
cmp     %esi,%eax
jge     40114f <sum_array+0x19>
movslq  %eax,%rcx
add     (%rdi,%rcx,4),%edx
add     $0x1,%eax
jmp     401140 <sum_array+0xa>
mov     %edx,%eax
retq
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00
  40113b:   ba 00 00 00 00
  401140:   39 f0
  401142:   7d 0b
  401144:   48 63 c8
  401147:   03 14 8f
  40114a:   83 c0 01
  40114d:   eb f1
  40114f:   89 d0
  401151:   c3
```

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00          mov    $0x0,%eax
  40113b:  ba 00 00 00 00          mov    $0x0,%edx
  401140:  39 f0                   cmp    %esi,%eax
  401142:  7d 0b                   jge    40114f <sum_array+0x19>
  401144:  48 63 c8                movslq %eax,%rcx
  401147:  03 14 8f                add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01                add    $0x1,%eax
  40114d:  eb f1                   jmp    401140 <sum_array+0xa>
  40114f:  89 d0                   mov    %edx,%eax
  401151:  c3                      retq
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
  401140:  39 f0                 cmp    %esi,%eax
  401142:  7d 0b                 jge    40114f <sum_array+0x19>
  401144:  48 63 c8              movslq %eax,%rcx
  401147:  03 14 8f              add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01              add    $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```

Each instruction has an operation name (“opcode”).

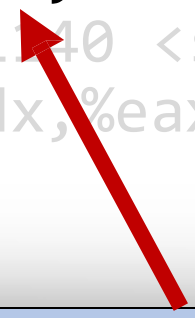
Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
  401140:  39 f0                 cmp    %esi,%eax
  401142:  7d 0b                 jge    40114f <sum_array+0x19>
  401144:  48 63 c8              movslq %eax,%rcx
  401147:  03 14 8f              add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01              add    $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3
```

Each instruction can also have arguments (“operands”).

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00        mov    $0x0,%eax
  40113b:  ba 00 00 00 00        mov    $0x0,%edx
  401140:  39 f0                 cmp    %esi,%eax
  401142:  7d 0b                 jge    40114f <sum_array+0x19>
  401144:  48 63 c8              movslq %eax,%rcx
  401147:  03 14 8f              add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01              add    $0x1,%eax
  40114d:  eb f1                 jmp    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```



$[number] means a constant value, or “immediate” (e.g. 1 here).

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:  b8 00 00 00 00      mov    $0x0,%eax
  40113b:  ba 00 00 00 00      mov    $0x0,%edx
  401140:  39 f0               cmp    %esi,%eax
  401142:  7d 0b               jge    40114f <sum_array+0x19>
  401144:  48 63 c8            movslq %eax,%rcx
  401147:  03 14 8f            add    (%rdi,%rcx,4),%edx
  40114a:  83 c0 01            add    $0x1,%eax
  40114d:  eb f1               jmp    401140 <sum_array+0xa>
  40114f:  89 d0               mov    %edx,%eax
  401151:  c3                  retq
```

%[name] means a register, a storage location on the CPU (e.g. edx here).

Lecture Plan

- **Overview:** GCC and Assembly 7
- **Demo:** Looking at an executable 11
- **Registers and The Assembly Level of Abstraction** 24
- The **mov** instruction 35

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

Assembly Abstraction

- C abstracts away the low-level details of machine code. It lets us work using variables, variable types, and other higher-level abstractions.
- C and other languages let us write code that works on most machines.
- Assembly code is just bytes! No variable types, no type checking, etc.
- Assembly/machine code is processor-specific.
- What is the level of abstraction for assembly code?

Registers

Registers


%rax


%rsi


%r8


%r12


%rbx


%rdi


%r9


%r13


%rcx


%rbp


%r10


%r14


%rdx


%rsp


%r11


%r15

Registers

What is a register?

A register is a fast read/write memory slot right on the CPU that can hold variable values.

Registers are **not** located in memory.

Registers

- A **register** is a 64-bit space inside the processor.
- There are 16 registers available, each with a unique name.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold parameters and return values for functions.
- Registers are extremely *fast* memory!
- Processor instructions consist mostly of moving data into/out of registers and performing arithmetic on them. This is the level of logic your program must be in to execute!

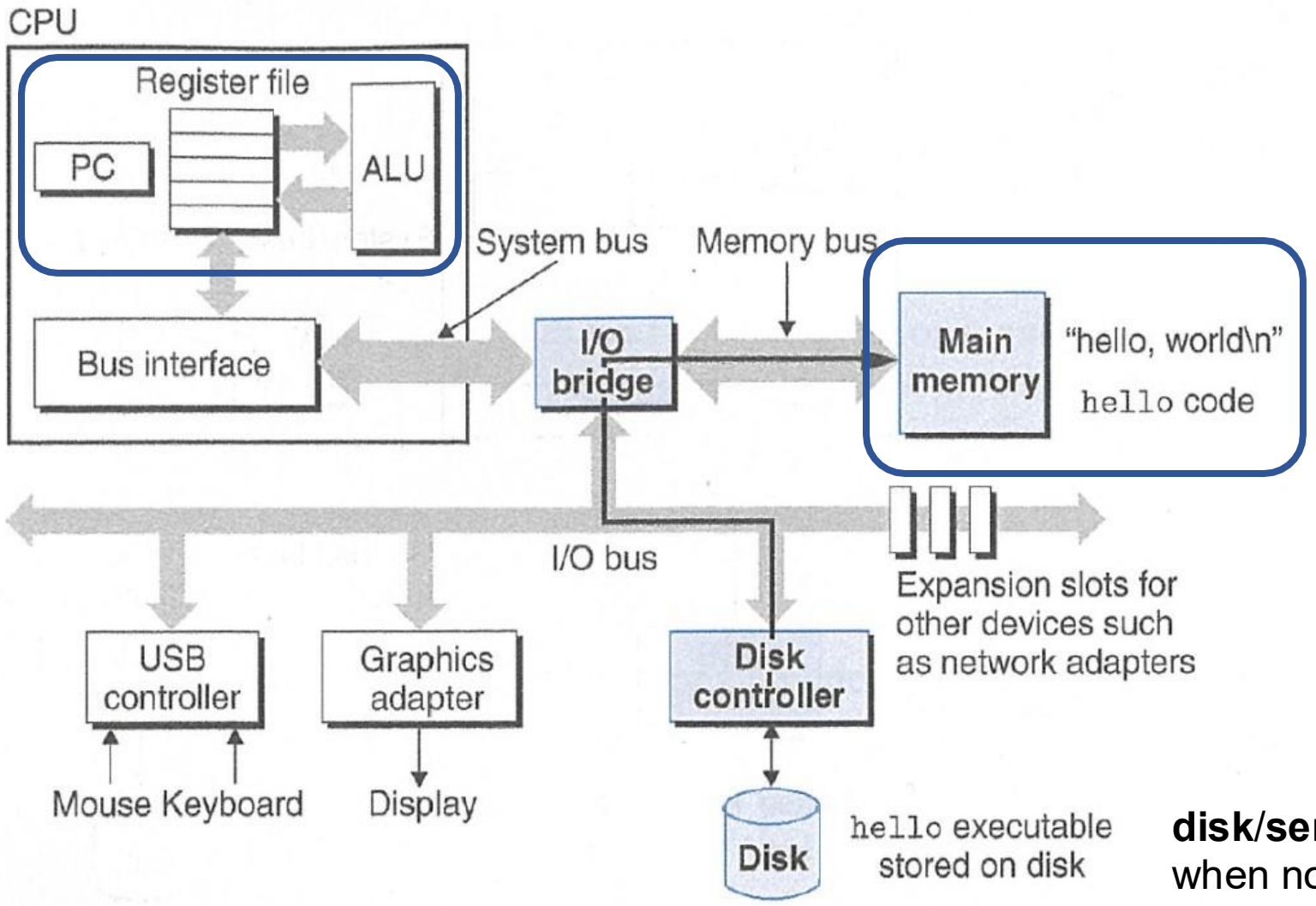
Machine-Level Code

Assembly instructions manipulate these registers. For example:

- One instruction adds two numbers in registers
- One instruction transfers data from a register to memory
- One instruction transfers data from memory to a register

Computer architecture

registers accessed by name
ALU is main workhorse of CPU



memory needed for program execution (stack, heap, etc.) accessed by address

disk/server stores program when not executing

GCC And Assembly

- GCC compiles your program – it lays out memory on the stack and heap and generates assembly instructions to access and do calculations on those memory locations.
- Here's what the "assembly-level abstraction" of C code might look like:

C	Assembly Abstraction
`int sum = x + y;`	1) Copy x into register 1 2) Copy y into register 2 3) Add register 2 to register 1 4) Write register 1 to memory for sum

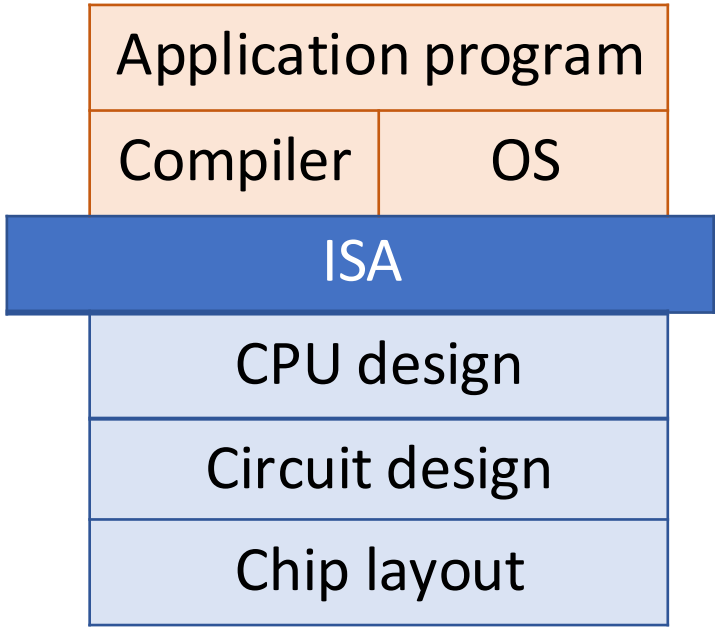
Assembly

- We are going to learn the **x86-64** instruction set architecture. This instruction set is used by Intel and AMD processors.
- There are many other instruction sets: ARM, MIPS, etc.

Instruction set architecture (ISA)

A contract between program/compiler and hardware:

- Defines operations that the processor (CPU) can execute
- Data read/write/transfer operations
- Control mechanisms



Intel originally designed their instruction set back in 1978.

- Legacy support is a huge issue for x86-64
- Originally 16-bit processor, then 32 bit, now 64 bit.
  These design choices dictated the register sizes
  (and even register/instruction names).

Lecture Plan

- **Overview:** GCC and Assembly 7
- **Demo:** Looking at an executable 11
- Registers and The Assembly Level of Abstraction 24
- **The mov Instruction** 35
- Live Session 57

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

mov

The **mov** instruction copies bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov     src,dst
```

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only **src**)* — `$0x104`
- Register — `%rbx`
- Memory Location
  (*at most one of **src, dst**)* — Direct address `0x6005c0`

Operand Forms: Immediate

```
mov     $0x104,____
```

Copy the value 0x104 into some destination.

Operand Forms: Registers

Copy the value in register %rbx into some destination.

```
mov     %rbx,____
```

```
mov     ____,%rbx
```

Copy the value from some source into register %rbx.

Operand Forms: Absolute Addresses

Copy the value at address 0x104 into some destination.

```
mov     0x104,_____
```

```
mov     _____,0x104
```

Copy the value from some source into the memory at address 0x104.

Practice #1: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *5* is stored at address 0x42, and the value *8* is stored in %rbx.

```
1. mov    $0x42,%rax

2. mov     0x42,%rax

3. mov     %rbx,0x55
```

Operand Forms: Indirect

Copy the value at the address stored in register %rbx into some destination.

```
mov     (%rbx),_____
```

```
mov     _____,(%rbx)
```

Copy the value from some source into the memory at the address stored in register %rbx.

Operand Forms: Base + Displacement

```
mov    0x10(%rax),_____________
```

*Copy the value at the address (**0x10 plus** what is stored in register %rax) into some destination.*

```
mov    _____________,0x10(%rax)
```

*Copy the value from some source into the memory at the address (**0x10 plus** what is stored in register %rax).*

Operand Forms: Indexed

Copy the value at the address which is (the sum of the values in registers %rax and %rdx) into some destination.

```
mov    (%rax,%rdx),_____________

mov    _____________,(%rax,%rdx)
```

Copy the value from some source into the memory at the address which is (the sum of the values in registers %rax and %rdx).

Operand Forms: Indexed

*Copy the value at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx) into some destination.*

```
mov    0x10(%rax,%rdx),_____
```

```
mov    _____,0x10(%rax,%rdx)
```

*Copy the value from some source into the memory at the address which is (the sum of **0x10 plus** the values in registers %rax and %rdx).*

Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x11* is stored at address *0x10C*, *0xAB* is stored at address *0x104*, *0x100* is stored in register %rax and *0x3* is stored in %rdx.

```
1. mov    $0x42,(%rax)
2. mov    4(%rax),%rcx
3. mov    9(%rax,%rdx),%rcx
```

Imm(r_b, r_i) is equivalent to address **Imm + R[r_b] + R[r_i]**

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**4 times** the value in register %rdx) into some destination.*

```
mov    (,%rdx,4), _____
```

The *scaling factor* (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

```
mov    _____,(,%rdx,4)
```

*Copy the value from some source into the memory at the address which is (**4 times** the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (4 times the value in register %rdx, **plus 0x4**), into some destination.*

```
mov     0x4(,%rdx,4),_______
```

```
mov     _______,0x4(,%rdx,4)
```

*Copy the value from some source into the memory at the address which is (4 times the value in register %rdx, **plus 0x4**).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**the value in register %rax** plus 2 times the value in register %rdx) into some destination.*

```
mov    (%rax,%rdx,2),_____
```

```
mov    _____,(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**the value in register %rax** plus 2 times the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**0x4 plus** the value in register %rax plus 2 times the value in register %rdx) into some destination.*

```
mov    0x4(%rax,%rdx,2),_____
```

```
mov    _____,0x4(%rax,%rdx,2)
```

*Copy the value from some source into the memory at the address which is (**0x4 plus** the value in register %rax plus 2 times the value in register %rdx).*

Most General Operand Form

Imm(r_b,r_i,s)

is equivalent to…

Imm + R[r_b] + R[r_i]*s

Most General Operand Form

`Imm(`r_b`, `r_i`, s)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]*s`**

Displacement: pos/neg constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Scale must be 1,2,4, or 8 (if missing, = 1)

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$r_!$	$\mathrm{R}[r_!]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	$(r_!)$	$\mathrm{M}[\mathrm{R}[r_!]]$	Indirect
Memory	$Imm(r_{"})$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}]]$	Base + displacement
Memory	$(r_{"}, r_{\#})$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$	Indexed
Memory	$Imm(r_{"}, r_{\#})$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$	Indexed
Memory	$(, r_{\#}, s)$	$\mathrm{M}[\mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$Imm(, r_{\#}, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$(r_{"}, r_{\#}, s)$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$Imm(r_{"}, r_{\#}, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor *s* must be either, 1, 2, 4, or 8."

Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x1* is stored in register %rcx, the value *0x100* is stored in register %rax, the value *0x3* is stored in register %rdx, and value *0x11* is stored at address *0x10C*.

1. `mov $0x42,0xfc(,%rcx,4)`

2. `mov (%rax,%rdx,4),%rbx`

`Imm(`r_b`, `r_i`, s)` is equivalent to address **`Imm + R[`r_b`] + R[`r_i`]*s`**

Displacement Base Index Scale (1,2,4,8)

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're 1/4th of the way to understanding assembly!
What looks understandable right now?

Some notes:

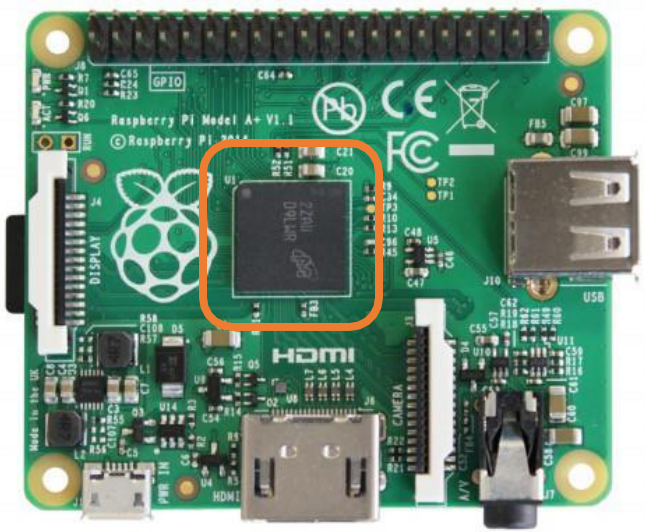
- Registers store addresses and values
- `mov src, dst` ***copies*** value into dst
- `sizeof(int)` is 4
- Instructions executed sequentially

```
0000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00      mov    $0x0,%edx
  4005bb:    b8 00 00 00 00      mov    $0x0,%eax
  4005c0:    eb 09               jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca            movslq %edx,%rcx
  4005c5:    03 04 8f            add    (%rdi,%rcx,4),%eax
  4005c8:    83 c2 01            add    $0x1,%edx
                                 cmp    %esi,%edx
                                 jl     4005c2 <sum_array+0xc>
                                 repz retq
```

We'll come back to this example in future lectures!

Central Processing Units (CPUs)

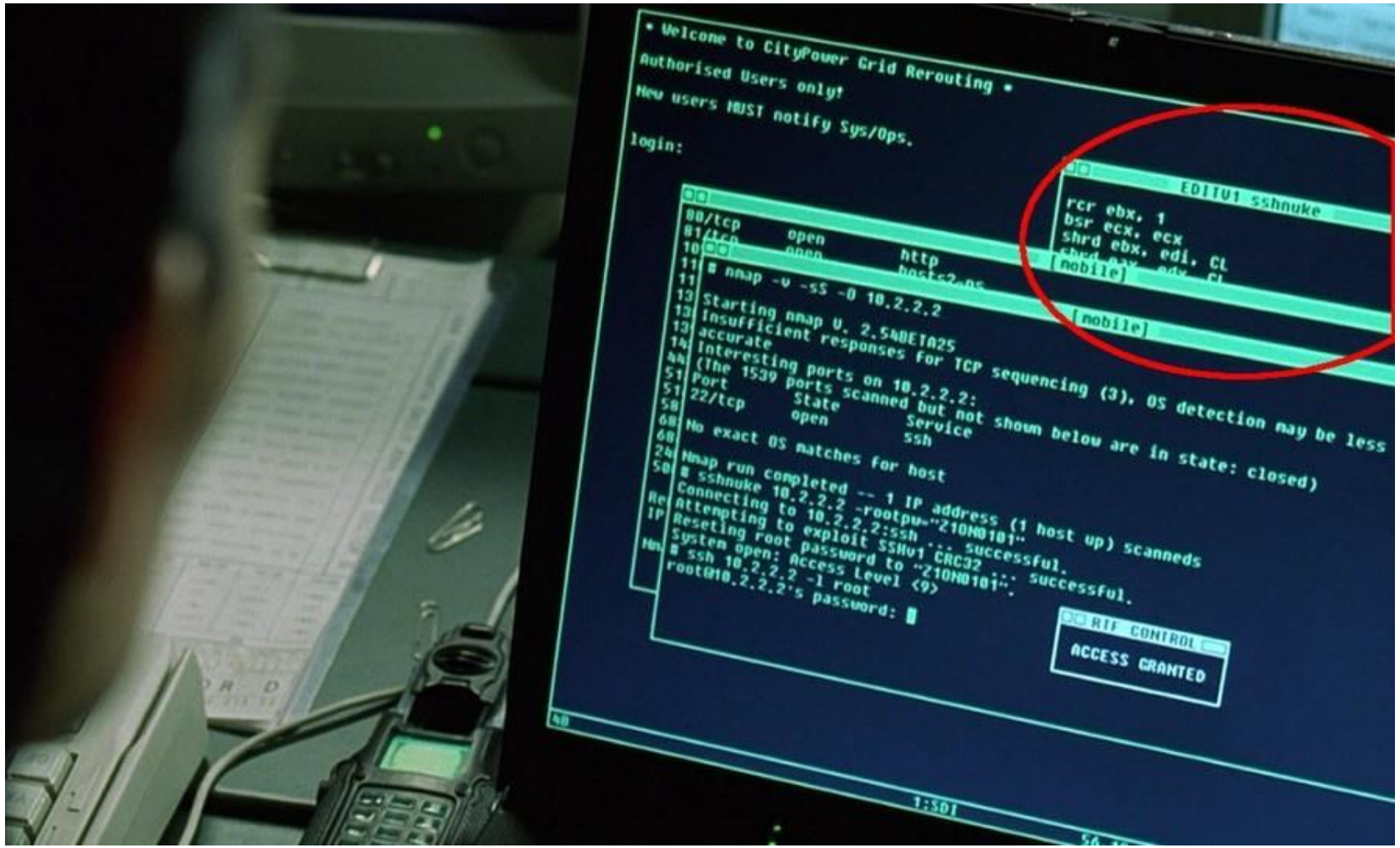
Intel 8086, 16-bit microprocessor ($86.65, 1978)



Raspberry Pi BCM2836 32-bit **ARM** microprocessor ($35 for everything, 2015)

Intel Core i9-9900K 64-bit 8-core multi-core processor ($449, 2018)

Assembly code in movies

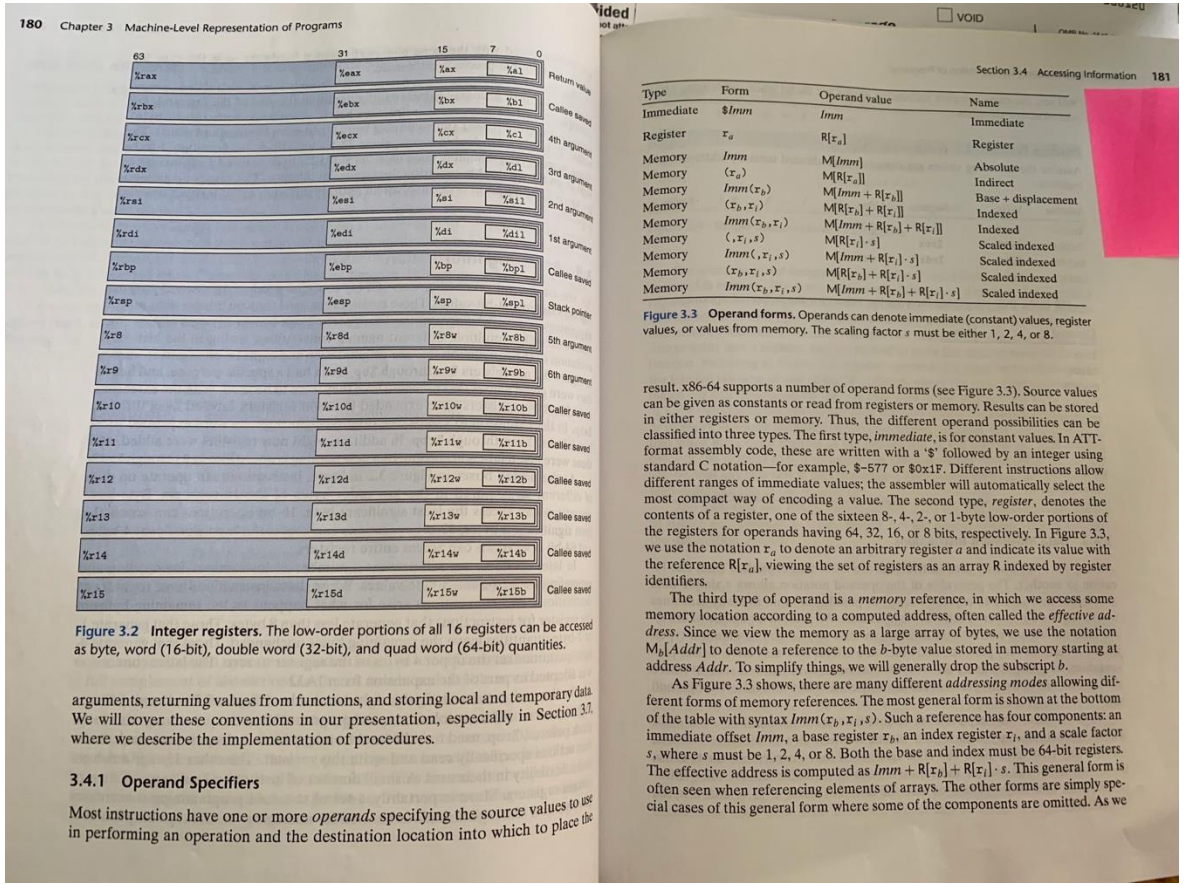


Trinity saving the world by hacking into the power grid using Nmap Network Scanning
The Matrix Reloaded, 2003

Keep a resource guide handy

- https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf
- B&O book:
  - Canvas -> Files
    -> Bryant_OHallaron_ch3.1-3.8.pdf

- It's like study abroad:
  - You took LANG 1A
  - Your tools give too much/too little information (a book reference, a rudimentary translator)
  - No one expects you to **speak** the language fluently…
  - …But the more you internalize, the better you can use tools to **read** the language



180 Chapter 3 Machine-Level Representation of Programs

63	31	15	7	0	
%rax	%eax	%ax	%al		Return value
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th argument
%rdx	%edx	%dx	%dl		3rd argument
%rsi	%esi	%si	%sil		2nd argument
%rdi	%edi	%di	%dil		1st argument
%rbp	%ebp	%bp	%bpl		Callee saved
%rsp	%esp	%sp	%spl		Stack pointer
%r8	%r8d	%r8w	%r8b		5th argument
%r9	%r9d	%r9w	%r9b		6th argument
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

arguments, returning values from functions, and storing local and temporary data. We will cover these conventions in our presentation, especially in Section 3.7, where we describe the implementation of procedures.

3.4.1 Operand Specifiers

Most instructions have one or more *operands* specifying the source values to use in performing an operation and the destination location into which to place the

Section 3.4 Accessing Information 181

Type	Form	Operand value	Name
Immediate	$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

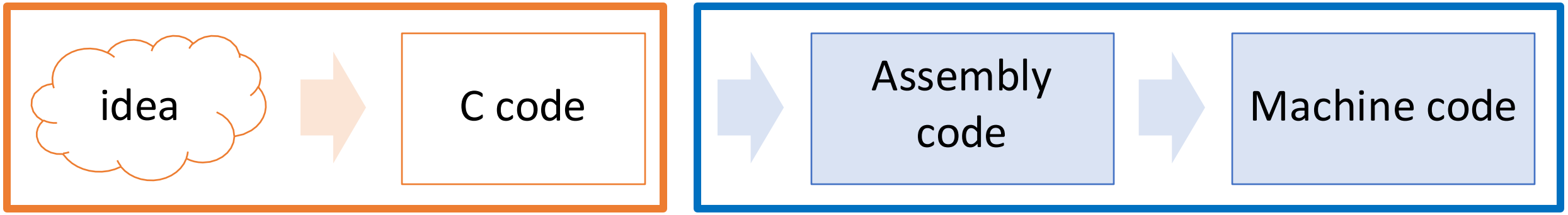
result. x86-64 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. In ATT-format assembly code, these are written with a '$' followed by an integer using standard C notation—for example, $-577 or $0x1F. Different instructions allow different ranges of immediate values; the assembler will automatically select the most compact way of encoding a value. The second type, *register*, denotes the contents of a register, one of the sixteen 8-, 4-, 2-, or 1-byte low-order portions of the registers for operands having 64, 32, 16, or 8 bits, respectively. In Figure 3.3, we use the notation r_a to denote an arbitrary register a and indicate its value with the reference $R[r_a]$, viewing the set of registers as an array R indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. Since we view the memory as a large array of bytes, we use the notation $M_b[Addr]$ to denote a reference to the b-byte value stored in memory starting at address $Addr$. To simplify things, we will generally drop the subscript b.

As Figure 3.3 shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm(r_b, r_i, s)$. Such a reference has four components: an immediate offset Imm, a base register r_b, an index register r_i, and a scale factor s, where s must be 1, 2, 4, or 8. Both the base and index must be 64-bit registers. The effective address is computed as $Imm + R[r_b] + R[r_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we

Chapter 3, Figures 3.2-3.3 (p. 180-181)

Why are we reading assembly?



Programmer-generated

gcc (compiler+assembler) generated

Main goal: Information retrieval

- We will not be writing assembly! (that's the compiler's job)
- Rather, we want to translate the assembly ***back*** into our C code.
- Knowing how our C code is converted into machine instructions gives us insight into how to write more efficient, cleaner code.

Extended warmup: Information Synthesis

Spend a few minutes thinking about the main paradigms of the `mov` instruction.

- What might be the equivalent C-like operation?
- Examples (note %r__ registers are 64-bit):

1. `mov $0x0,%rdx`
2. `mov %rdx,%rcx`
3. `mov $0x42,(%rdi)`
4. `mov (%rax,%rcx,8),%rax`

Extended warmup: Information Synthesis

Spend a few minutes thinking about the main paradigms of the `mov` instruction.

- What might be the equivalent C-like operation?
- Examples (note %r__ registers are 64-bit):

Indirect addressing is like pointer arithmetic/deref!

1. `mov $0x0,%rdx -> maybe long x = 0`
2. `mov %rdx,%rcx -> maybe long x = y;`
3. `mov $0x42,(%rdi) -> maybe *ptr = 0x42;`
4. `mov (%rax,%rcx,8),%rax -> maybe long x = arr[i];`

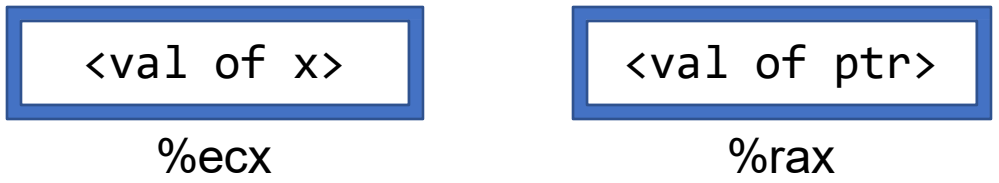
1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;
```

```
mov %ecx,(%rax)
```



(Pedantic: You should sub in <x> and <ptr> with actual values, like 4 and 0x7fff80)

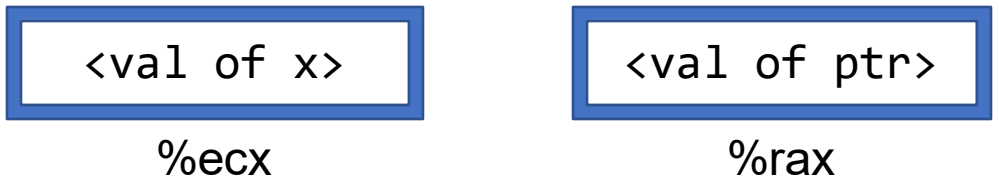
1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;     *ptr = x;
```

```
mov %ecx,(%rax)
```



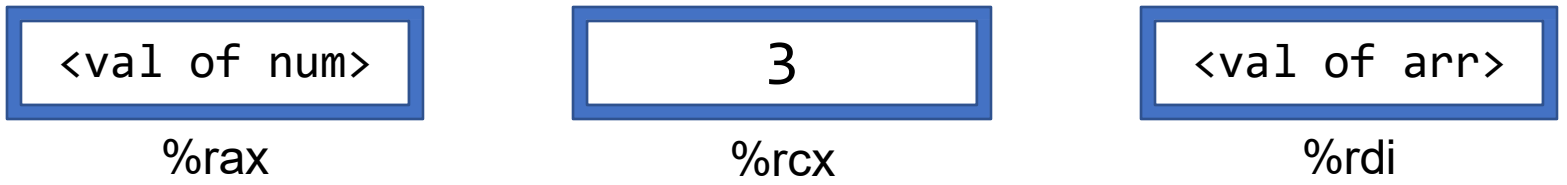
2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

```
mov (%rdi, %rcx, 8),%rax
```



2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

```
long num = arr[3];
long num = *(arr + 3);
long num = *(arr + y);
```

(assume `long y = 3;` declared earlier)

```
mov (%rdi, %rcx, 8),%rax
```

<val of num>	3	<val of arr>
%rax	%rcx	%rdi

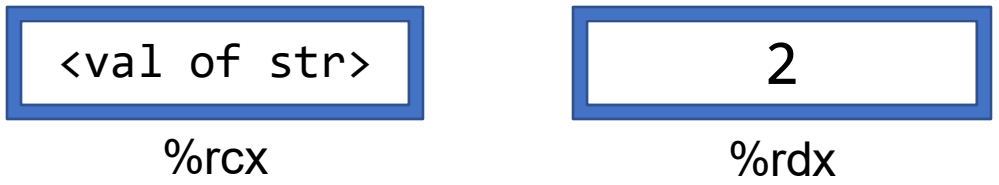
3. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

```
mov $0x63,(%rcx,%rdx,1)
```



3. Extra Practice

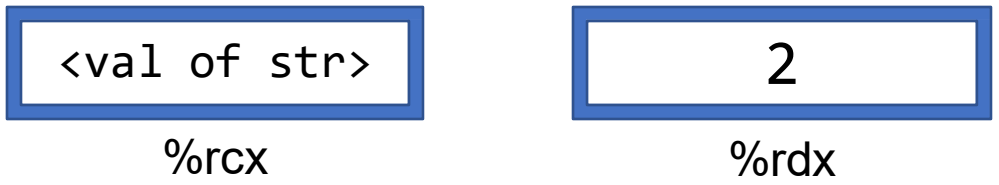
Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

```
str[2] = 'c';
*(str + 2) = 'c';
```

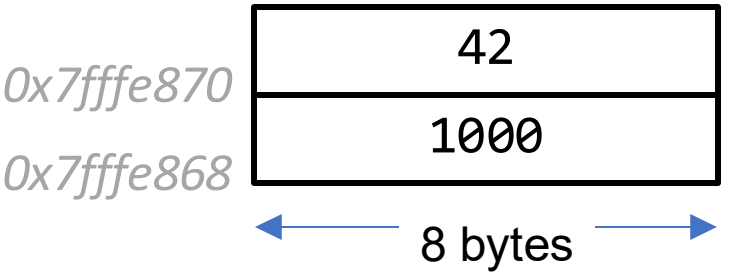
```
mov $0x63,(%rcx,%rdx,1)
```



Coming Up Soon To A Slide Near You

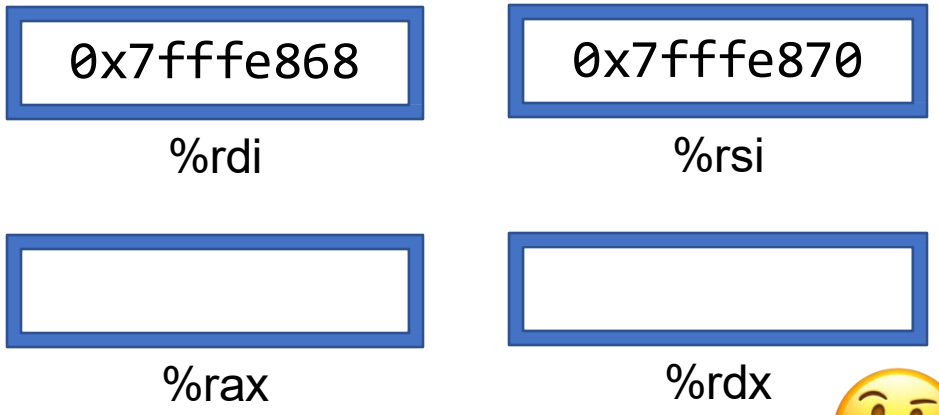
- The below code is the objdump of a C function, foo.
  - foo keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov     (%rdi),%rax
0x4005b9 <foo+3>    mov     (%rsi),%rdx
0x4005bc <foo+6>    mov     %rdx,(%rdi)
0x4005bf <foo+9>    mov     %rax,(%rsi)
```



1. **What does this function do?**
2. **What C code could have generated this assembly?**

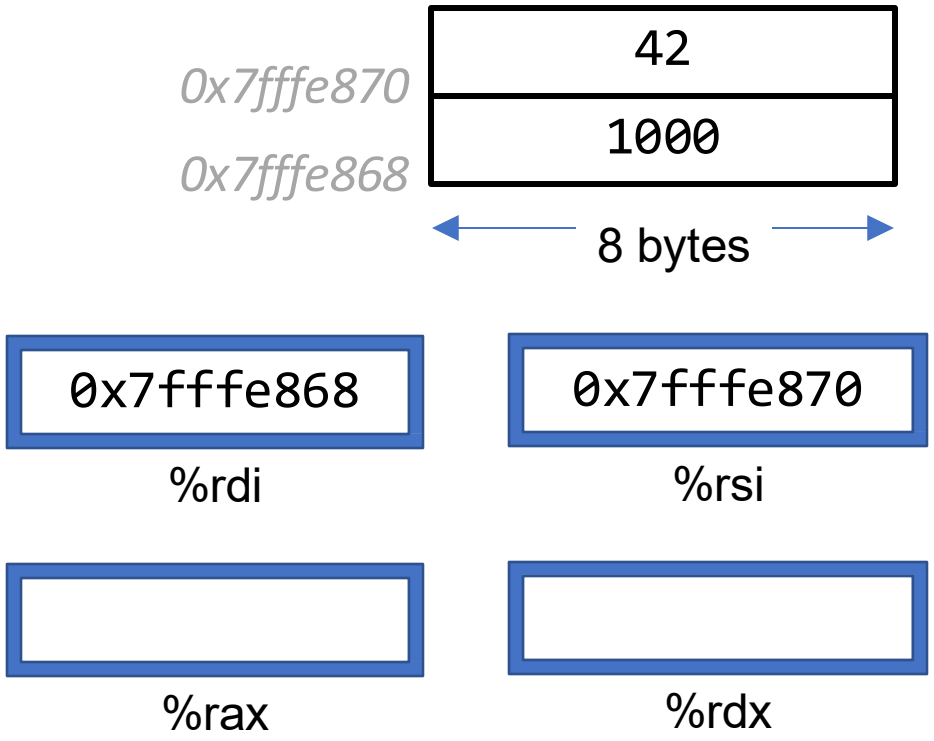
(Hints: make up C variable names as needed, assume all regs 64-bit)



Coming Up Soon To A Slide Near You

- The below code is the objdump of a C function, `foo`.
  - `foo` keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov     (%rdi),%rax
0x4005b9 <foo+3>    mov     (%rsi),%rdx
0x4005bc <foo+6>    mov     %rdx,(%rdi)
0x4005bf <foo+9>    mov     %rax,(%rsi)
```



Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Helpful Assembly Resources

- **Course textbook** (reminder: see relevant readings for each lecture on the Schedule page, http://cs107.stanford.edu/schedule.html)
- **CS107 Assembly Reference Sheet:** http://cs107.stanford.edu/resources/x86-64-reference.pdf
- **CS107 Guide to x86-64**: http://cs107.stanford.edu/guide/x86-64.html

References and Advanced Reading

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64

Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

mov

The **`mov`** instruction copies bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov     src,dst
```

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only **src**)*
- Register
- Memory Location
  (*at most one of **src, dst**)*

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	What's in %rax
4(%rax)	What's in %rax, plus 4
(%rax, %rdx)	Sum of what's in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	What's in %rcx, times 4 (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx
8(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx, plus 8

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$r_!$	$\mathrm{R}[r_!]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	$(r_!)$	$\mathrm{M}[\mathrm{R}[r_!]]$	Indirect
Memory	$Imm(r_{"})$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}]]$	Base + displacement
Memory	$(r_{"}, r_\#)$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_\#]]$	Indexed
Memory	$Imm(r_{"}, r_\#)$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_\#]]$	Indexed
Memory	$(, r_\#, s)$	$\mathrm{M}[\mathrm{R}[r_\#] \ . \ s]$	Scaled indexed
Memory	$Imm(, r_\#, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_\#] \ . \ s]$	Scaled indexed
Memory	$(r_{"}, r_\#, s)$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_\#] \ . \ s]$	Scaled indexed
Memory	$Imm(r_{"}, r_\#, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_\#] \ . \ s]$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor *s* must be either, 1, 2, 4, or 8."

Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Data Sizes

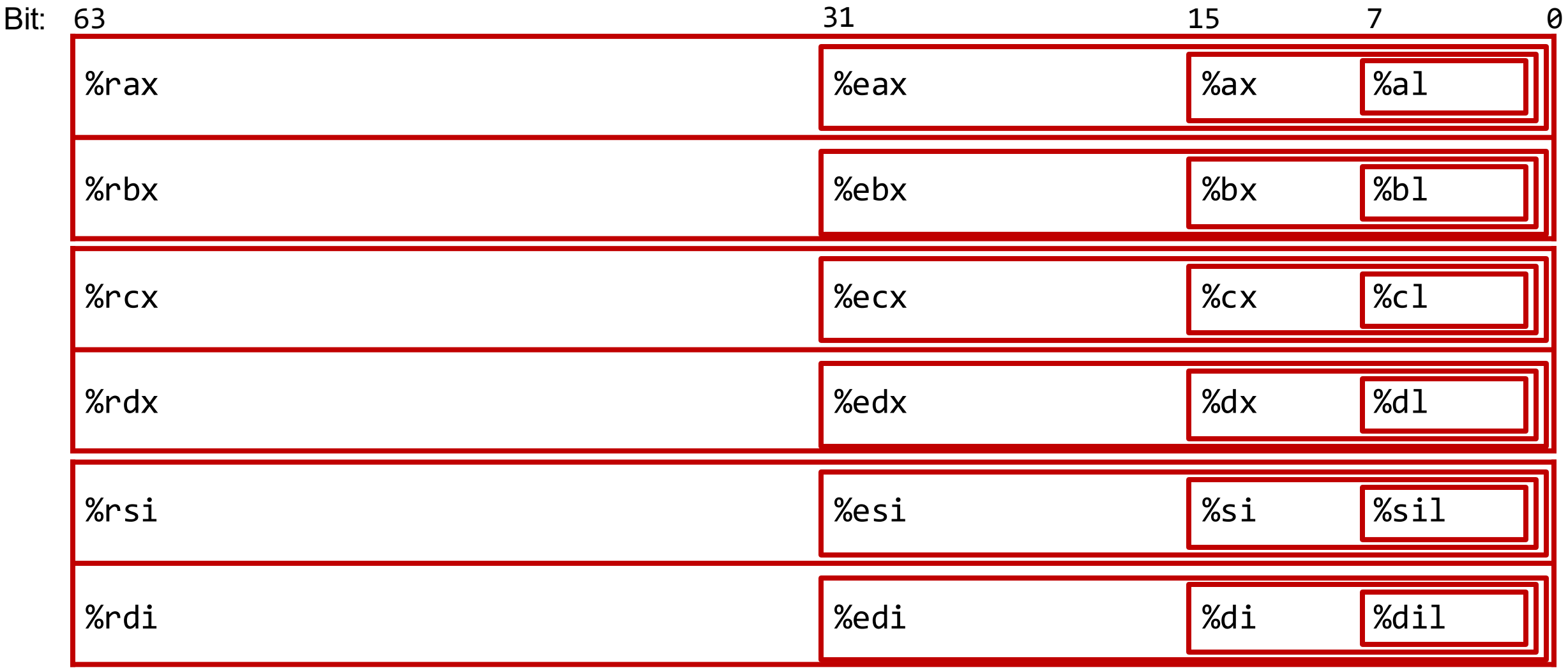
Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

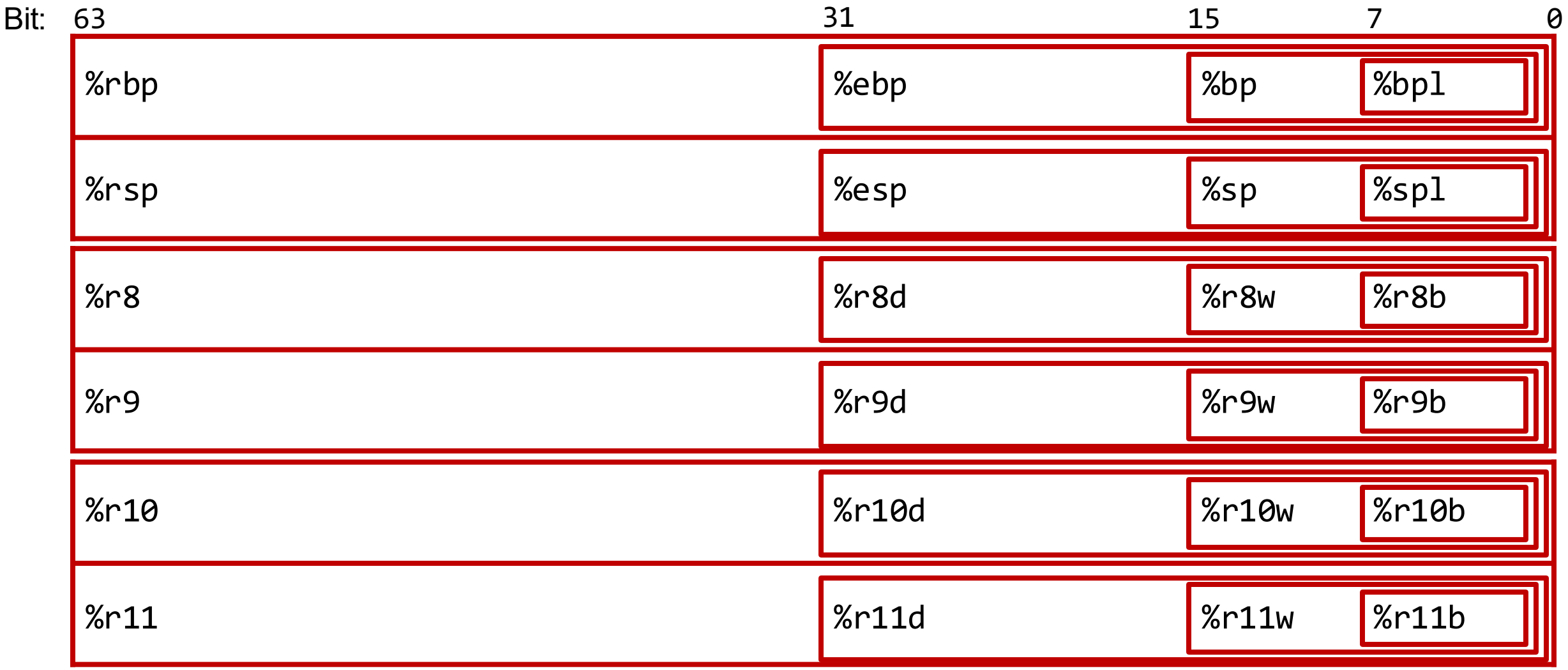
Assembly instructions can have suffixes to refer to these sizes:

- `b` means **byte**
- `w` means **word**
- `l` means **double word**
- `q` means **quad word**

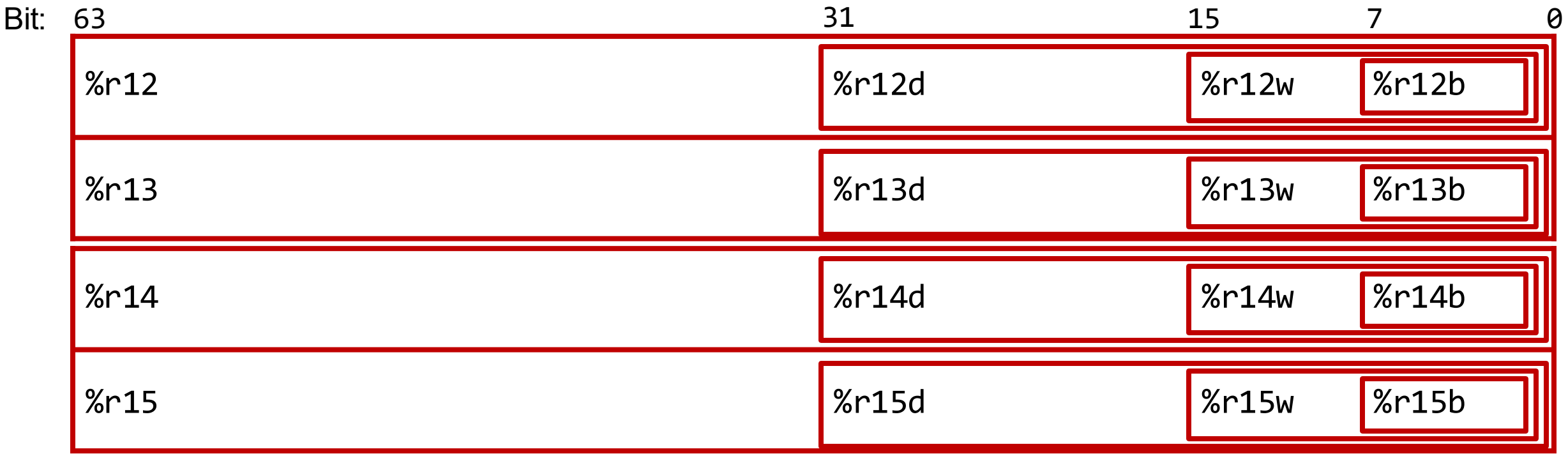
Register Sizes



Register Sizes



Register Sizes



Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

mov Variants

- **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move: `movb, movw, movl, movq`
- **mov** only updates the specific register bytes or memory locations indicated.
  - **Exception: movl** writing to a register will also set high order 4 bytes to 0.

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

```
1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
```

mov

- The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

movabsq $0x0011223344556677, %rax

movz and movs

- There are two mov instructions that can be used to copy a smaller source to a larger destination: **movz** and **movs**.
- **movz** fills the remaining bytes with zeros
- **movs** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

movz and movs

```
MOVZ S,R        R ← ZeroExtend(S)
```

Instruction	Description
`movzbw`	Move zero-extended byte to word
`movzbl`	Move zero-extended byte to double word
`movzwl`	Move zero-extended word to double word
`movzbq`	Move zero-extended byte to quad word
`movzwq`	Move zero-extended word to quad word

movz and movs

MOVS S,R R ← SignExtend(S)

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax %rax <- SignExtend(%eax)

Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

lea

The **lea** instruction copies an "effective address" from one place to another.

```
lea     src,dst
```

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **mov**. The difference is how it handles the src.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **imul**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D      D ← D * S
```

- If you specify one operand, it multiplies that by **%rax**, and splits the product across **2** registers. It puts the high-order 64 bits in **%rdx** and the low-order 64 bits in **%rax**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- Terminology: **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient + remainder**
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **cqto** instruction sign-extends the 64-bit value in **%rax** into **%rdx** to fill both registers with the dividend, as the division instruction expects.

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **%cl** (and only that register!)

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log2(w)** bits of **%cl** to know how much to shift.
  - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Lecture Plan

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Assembly Exploration

- Let's pull these commands together and see how some C code might be translated to assembly.
- Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let's check it out!
- https://godbolt.org/z/WPzz6G4a9

Code Reference: add_to_first

```
// Returns the sum of x and the first element in
arr
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}

----------

add_to_first:
  movl %edi, %eax
  addl (%rsi), %eax
  ret
```

Code Reference: full_divide

```
// Returns x/y, stores remainder in location stored in
remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}
-------

full_divide:
  movq %rdi, %rax
  movq %rdx, %rcx
  cqto
  idivq %rsi
  movq %rdx, (%rcx)
  ret
```

Assembly Exercise 1

```
000000000040116e <sum_example1>:
  40116e: 8d 04 37              lea  (%rdi,%rsi,1),%eax
  401171: c3                    retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

```
// B)
int sum_example1(int x, int y) {
    return x + y;
}
```

Assembly Exercise 2

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov  0xc(%rdi),%eax
      401175: 03 07           add  (%rdi),%eax
      401177: 2b 47 18        sub  0x18(%rdi),%eax
      40117a: c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

%eax

Assembly Exercise 3

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov  0xc(%rdi),%eax
      401175: 03 07           add  (%rdi),%eax
      401177: 2b 47 18        sub  0x18(%rdi),%eax
      40117a: c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

0x18

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're 1/2 of the way to understanding assembly!
What looks understandable right now?

```
0000000000401136 <sum_array>:
  401136:    b8 00 00 00 00      mov    $0x0,%eax
  40113b:    ba 00 00 00 00      mov    $0x0,%edx
  401140:    39 f0               cmp    %esi,%eax
  401142:    7d 0b               jge    40114f <sum_array+0x19>
  401144:    48 63 c8            movslq %eax,%rcx
  401147:    03 14 8f            add    (%rdi,%rcx,4),%edx
  40114a:    83 c0 01            add    $0x1,%eax
  40114d:    eb f1               jmp    401140 <sum_array+0xa>
  40114f:    89 d0               mov    %edx,%eax
  401151:    c3                  retq
```

A Note About Operand Forms

- Many instructions share the same address operand forms that **mov** uses.
  - Eg. 7(%rax, %rcx, 2).
- These forms work the same way for other instructions, e.g. sub:
  - sub 8(%rax,%rdx),%rcx -> Go to 8 + %rax + %rdx, subtract what's there from %rcx
- The exception is **lea:**
  - It interprets this form as just the calculation, *not the dereferencing*
  - lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx

Shift Amount

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log2(w)** bits of **%cl** to know how much to shift.
  - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Division and Remainder

Instruction	Effect	Description
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S`	Unsigned divide

- Terminology: **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.

Extra Practice

https://godbolt.org/z/hGKPWszq4

Reverse Engineering 1

```
int add_to(int x, int arr[], int i)
    { int sum = _?___;
    sum += arr[___?___];
    return ___?___;
}

----------

add_to:
  movslq %edx, %rdx
  movl %edi, %eax
  addl (%rsi,%rdx,4), %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i)
    { int sum = _?___;
    sum += arr[___?___];
    return ___?___;
}


----------
// x in %edi, arr in %rsi, i in
%edx add_to:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i)
    { int sum = x;
    sum += arr[i];
    return sum;
}


----------
// x in %edi, arr in %rsi, i in
%edx add_to:
  movslq %edx, %rdx            // sign-extend i into full register
  movl %edi, %eax              // copy x into %eax
  addl (%rsi,%rdx,4), %eax     // add arr[i] to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y)
    { int z = nums[_?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------

elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y)
    { int z = nums[_?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y)
    { int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull (%rdi), %eax       // multiply %eax by nums[0]
  subl 4(%rdi), %eax       // subtract nums[1] from %eax
  sarl $2, %eax            // shift %eax right by 2
  addl $2, %eax            // add 2 to %eax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------

func:
  movq %rdi, %rax
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  cqto
  idivq %rcx
  movq %rdx, %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax          // copy x into %rax
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x +
    1
    return result;
}---------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax          // copy x into %rax
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```

Side Note: Old GCC Output

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x +
    1
    return result;
}---------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx        // put x + 1 into %rcx
  movq %rcx, (%rsi)         // copy %rcx into *ptr
  movq %rdi, %rax           // copy x into %rax
  cqto                      // sign-extend x into %rdx
  idivq %rcx                // calculate x / (x + 1)
  movq %rdx, %rax           // copy the remainder into %rax
  ret
```

Learning Goals

- Learn about how assembly stores comparison and operation results in condition codes
- Understand how assembly implements loops and control flow

Lecture Plan

Lecture Plan

Executing Instructions

What does it mean for a program to execute?

Executing Instructions

So far:

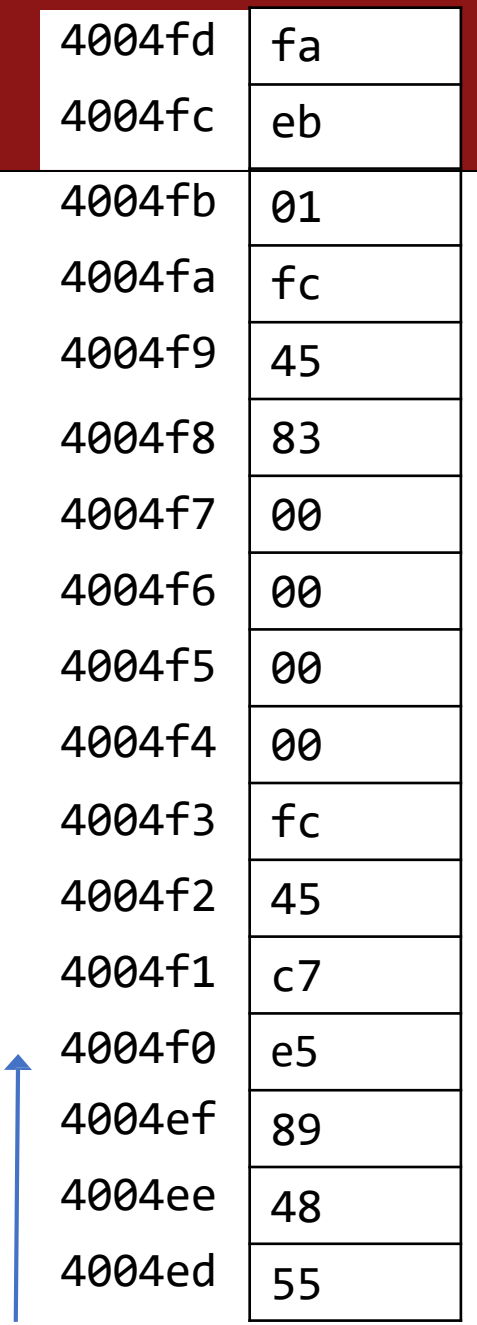
- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Today:

- **Who controls the instructions**?
  How do we know what to do now or next?

Answer:

- The **program counter** (PC), %rip.



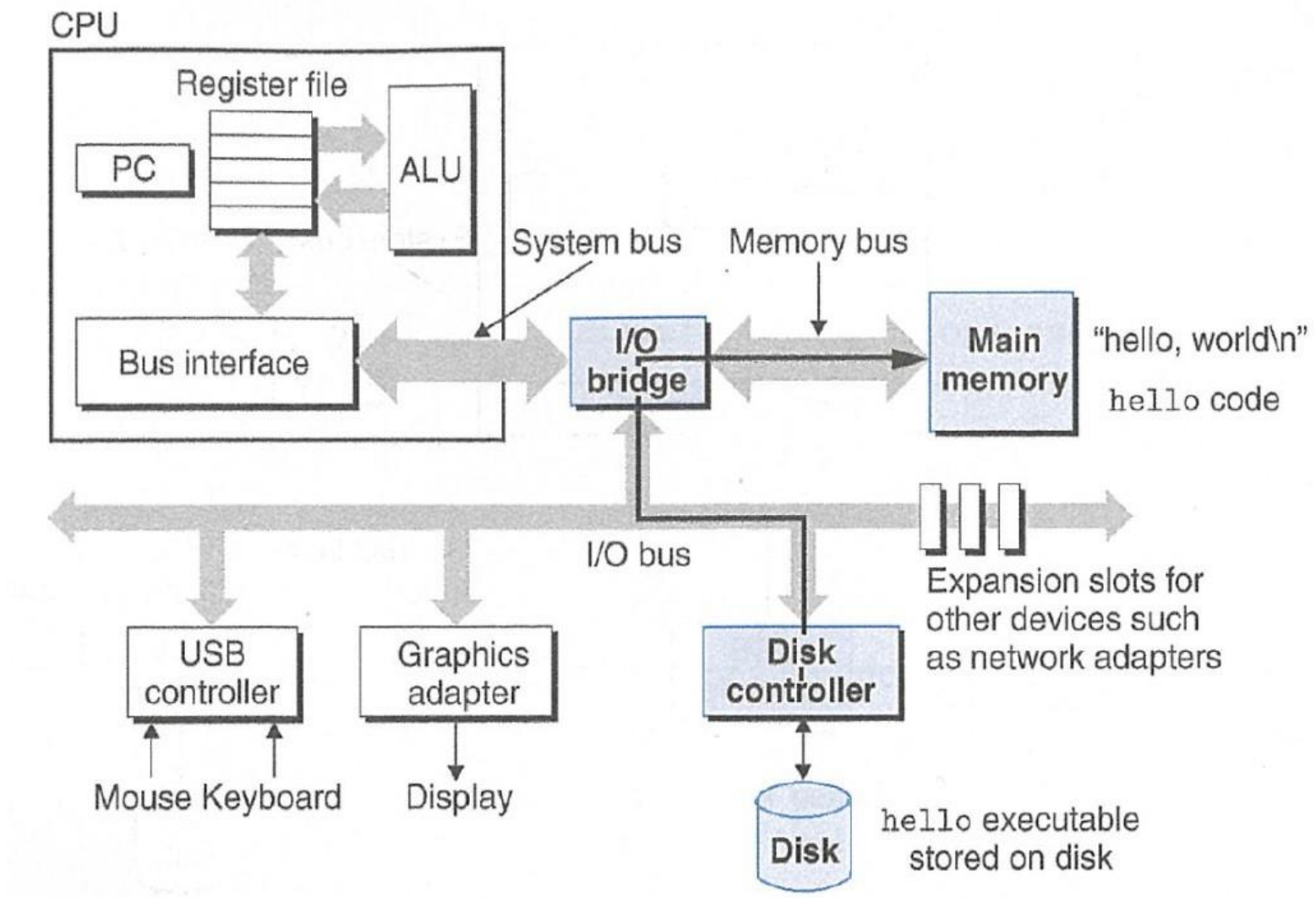
Register Responsibilities

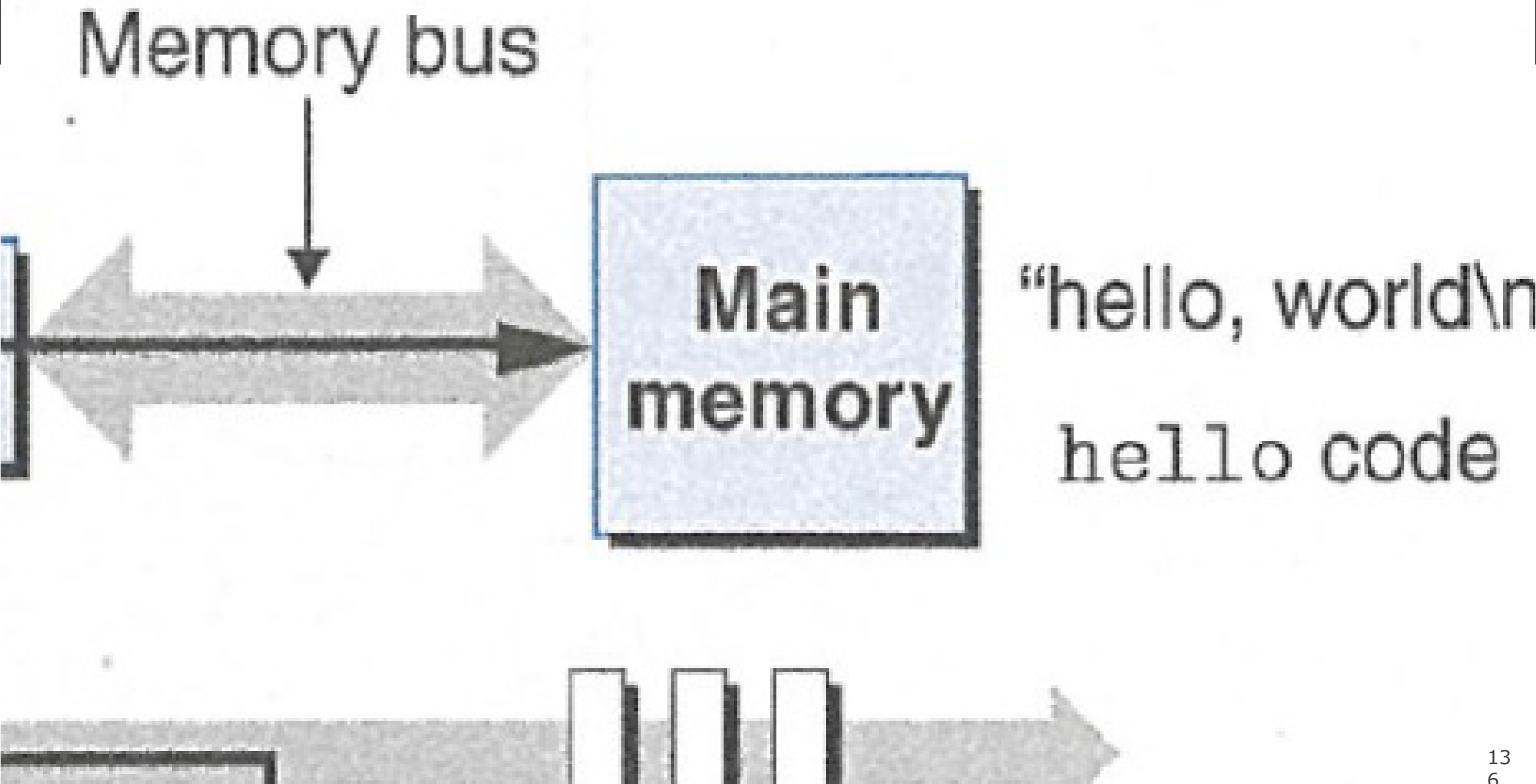
Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

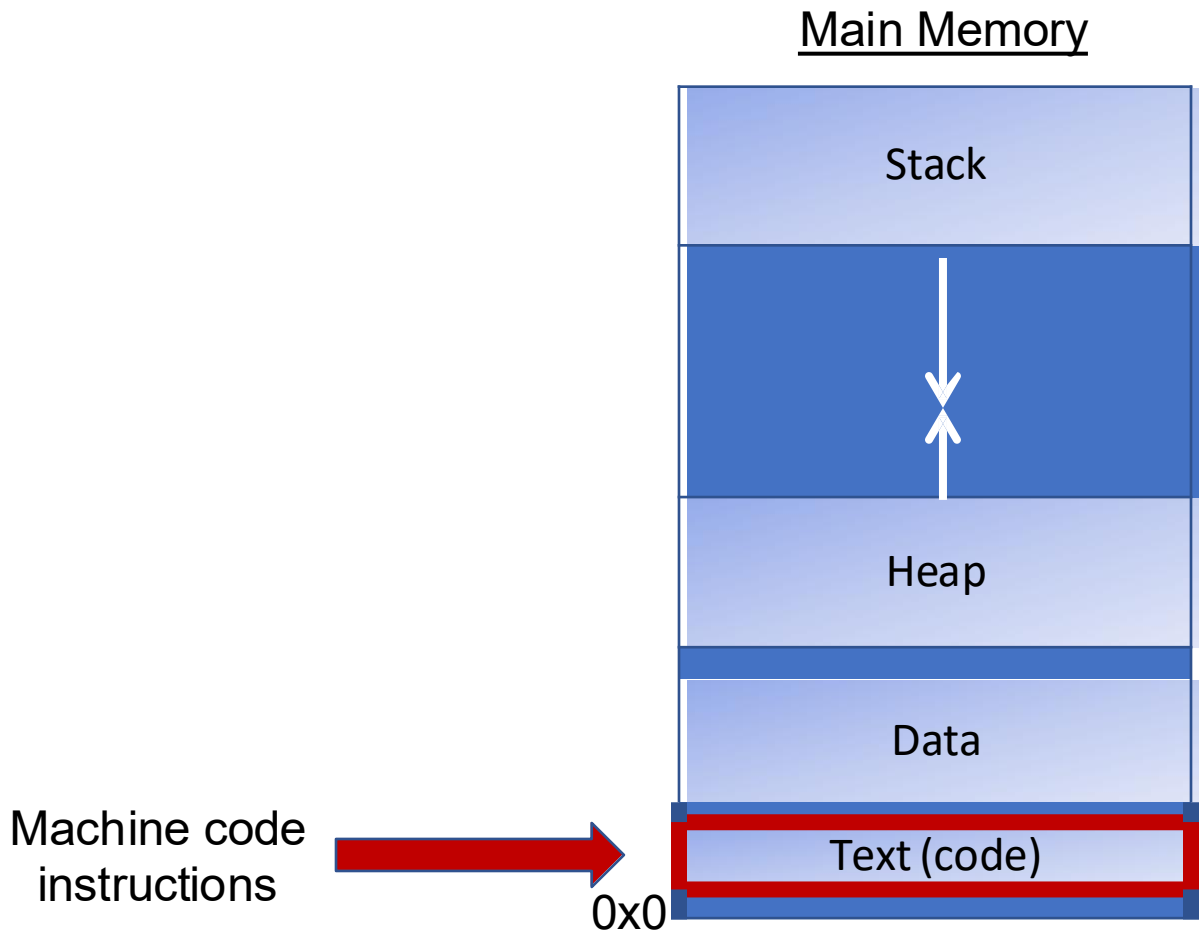
See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

Instructions Are Just Bytes!




Memory bus
Main memory
"hello, world\n
hello code

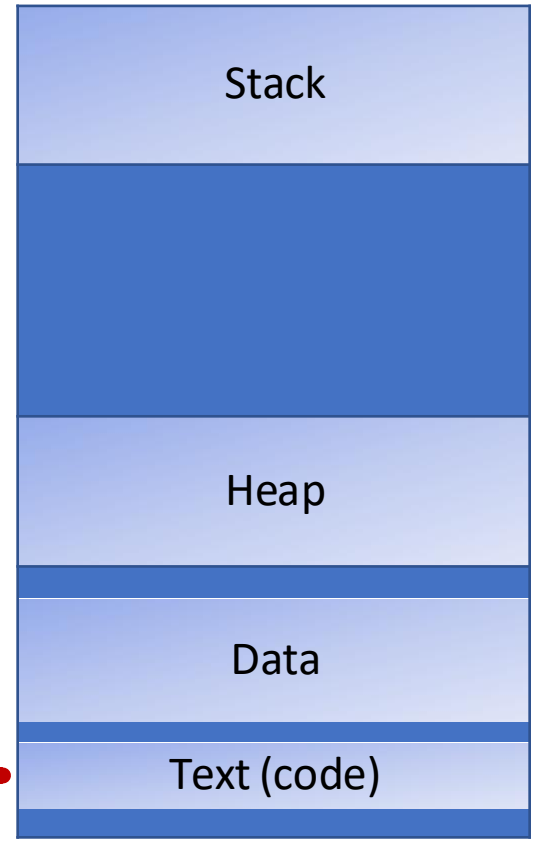
Instructions Are Just Bytes!



%ri

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

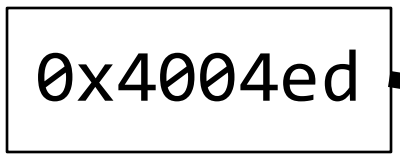
	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



%rip

```
00000000004004ed <loop>:
4004ed:  55                    push   %rbp
4004ee:  48 89 e5              mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01           addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                 jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

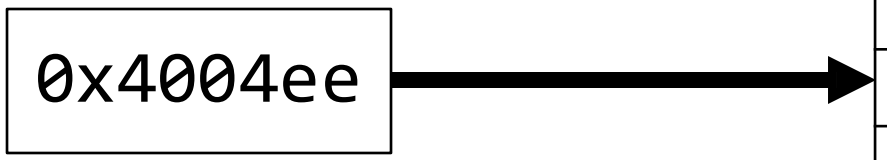


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                          push   %rbp
4004ee: 48 89 e5                    mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                 addl   $0x1,-0x4(%rbp)
4004fc: eb fa                       jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

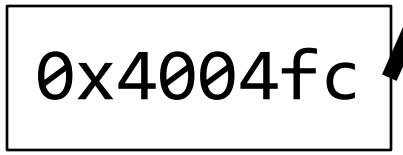
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
00000000004004ed <loop>:
4004ed: 55                          push   %rbp
4004ee: 48 89 e5                    mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                 addl   $0x1,-0x4(%rbp)
4004fc: eb fa                       jmp    4004f8 <loop+0xb>
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Going In Circles

- How can we use this representation of execution to represent e.g. a **loop**?
- **Key Idea:** we can ”interfere” with **%rip** and set it back to an earlier instruction!

Jump!

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed:  55                              push   %rbp
4004ee:  48 89 e5                        mov    %rsp,%rbp
4004f1:  c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8:  83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc:  eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

%rip

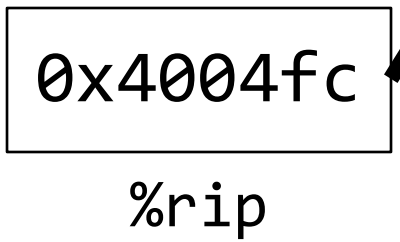
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Jump!

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

This assembly represents an infinite loop in C!

```
while (true) {…}
```



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

jmp

The **jmp** instruction jumps to another instruction in the assembly code ("Unconditional Jump").

```
jmp Label      (Direct Jump)
jmp *Operand   (Indirect Jump)
```

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be one of the usual operand forms (indirect jump):

```
jmp *%rax     # jump to instruction at address in %rax
```

"Interfering" with %rip

1. How do we repeat instructions in a loop?

`jmp [target]`

- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a **conditional jump**?

Lecture Plan

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.
- How is this represented in assembly?

Control

```
if (x > y) {
      // a
} else {
      // b
}
```

In Assembly:

1. Calculate the condition result
2. Based on the result, go to a or b

Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. `cmp S1, S2` // compare two values

2. je [target] *or* **jne [target]** *or* **jl [target]** *or* … // conditionally jump

"jump if equal"

"jump if not equal"

"jump if less than"

Conditional Jumps

There are also variants of **jmp** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero
`jne` *Label*	`jnz`	Not equal / not zero
`js` *Label*		Negative
`jns` *Label*		Nonnegative
`jg` *Label*	`jnle`	Greater (signed >)
`jge` *Label*	`jnl`	Greater or equal (signed >=)
`jl` *Label*	`jnge`	Less (signed <)
`jle` *Label*	`jng`	Less or equal (signed <=)
`ja` *Label*	`jnbe`	Above (unsigned >)
`jae` *Label*	`jnb`	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	Below (unsigned <)
`jbe` *Label*	`jna`	Below or equal (unsigned <=)

Control

Read **cmp S1,S2** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Control

Read **cmp S1,S2** as *"compare S2 to S1"*:

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?

Control

- The CPU has special registers called *condition codes* that are like “global variables”. They *automatically* keep track of information about the most recent arithmetic or logical operation.
  - **cmp** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded zero.
- **SF:** Sign flag. The most recent operation yielded a negative value.
- **OF:** Overflow flag. The most recent operation caused a two's-complement overflow-either negative or positive.

Condition Codes

Alongside normal registers, the CPU also has single-bit *condition code* registers. They store the results of the most recent arithmetic or logical operation.

Example: if we calculate t = a + b, condition codes are set according to:

- **CF:** Carry flag (Unsigned Overflow). `(unsigned) t < (unsigned) a`
- **ZF:** Zero flag (Zero). `(t == 0)`
- **SF:** Sign flag (Negative). `(t < 0)`
- **OF:** Overflow flag (Signed Overflow). `(a<0 == b<0) && (t<0 != a<0)`

Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 – S1
```

Instruction	Description
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Control

Read **cmp S1,S2** as *"compare S2 to S1".* It calculates S2 – S1 and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
// calculates %edi - 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
```

Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero (ZF = 1)
`jne` *Label*	`jnz`	Not equal / not zero (ZF = 0)
`js` *Label*		Negative (SF = 1)
`jns` *Label*		Nonnegative (SF = 0)
`jg` *Label*	`jnle`	Greater (signed >) (ZF = 0 and SF = OF)
`jge` *Label*	`jnl`	Greater or equal (signed >=) (SF = OF)
`jl` *Label*	`jnge`	Less (signed <) (SF != OF)
`jle` *Label*	`jng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`ja` *Label*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *Label*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb` *Label*	`jnae`	Below (unsigned <) (CF = 1)
`jbe` *Label*	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Setting Condition Codes

The **test** instruction is like **cmp**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

Condition Codes

- Previously-discussed arithmetic and logical instructions update these flags. **lea** does not (it was intended only for address computations).
- Logical operations (**xor**, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.
- For more complicated reasons, **inc** and **dec** set the overflow and zero flags, but leave the carry flag unchanged.

Exercise 1: Conditional jump

```
je target                jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

Exercise 1: Conditional jump

```
je target
```

jump if ZF is 1

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don't jump

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi **0x5**

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. `4004d9`
   - B. `4004db`
   - C. `4004de`
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. `4004e1`
   - B. `0x2`
   - C. `0xa`
   - D. `0xc`
   - E. Other

Exercise 2: Conditional jump

```
0000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi **0x5**

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. 4004d9
   - B. 4004db
   - C. 4004de
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. 4004e1
   - B. 0x2
   - C. 0xa
   - D. 0xc
   - E. Other

Lecture Plan

If Statements

How can we use instructions like **cmp** and *conditional jumps* to implement if statements in assembly?

Practice: Fill In The Blank

```
int if_then(int param1) {
  if ( __________ ) {
      __________;
  }

  return __________;
}
```

```
0000000000401126 <if_then>:
  401126:    cmp     $0x6,%edi
  401129:    je      40112f
  40112b:    lea     (%rdi,%rdi,1),%eax
  40112e:    retq
  40112f:    add     $0x1,%edi
  401132:    jmp     40112b
```

Practice: Fill In The Blank

```
int if_then(int param1) {
   if ( param1 == 6 ) {
         param1++;
   }

   return param1 * 2;
}
```

```
0000000000401126 <if_then>:
  401126:    cmp     $0x6,%edi
  401129:    je      40112f
  40112b:    lea     (%rdi,%rdi,1),%eax
  40112e:    retq
  40112f:    add     $0x1,%edi
  401132:    jmp     40112b
```

Common If-Else Construction

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (x < y) {
        result = y - x;
    } else {
        result = x - y;
    }

    return result;
}
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (________) {

        ____________________ ;
    } else {

        ____________________ ;
    }

    return result;
}
```

```
401134 <+0>:   mov     %rsi,%rax
401137 <+3>:   cmp     %rsi,%rdi
40113a <+6>:   jge     0x401140 <absdiff+12>
40113c <+8>:   sub     %rdi,%rax
40113f <+11>:  retq
401140 <+12>:  sub     %rsi,%rdi
401143 <+15>:  mov     %rdi,%rax
401146 <+18>:  retq
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (  x < y  ) {
        result = y - x ;
    } else {
        result = x - y ;
    }
    return result;
}
```

```
401134 <+0>:   mov     %rsi,%rax
401137 <+3>:   cmp     %rsi,%rdi
40113a <+6>:   jge     0x401140 <absdiff+12>
40113c <+8>:   sub     %rdi,%rax
40113f <+11>:  retq
401140 <+12>:  sub     %rsi,%rdi
401143 <+15>:  mov     %rdi,%rax
401146 <+18>:  retq
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body
```

If-Else Construction Variations

C Code

```
int test(int arg) {
    int ret;
    if (arg > 3) {
        ret = 10;
    } else {
        ret = 0;
    }

    ret++;
    return ret;
}
```

Assembly

```
401134 <+0>:   cmp    $0x3,%edi
401137 <+3>:   jle    0x401142 <test+14>
401139 <+5>:   mov    $0xa,%eax
40113e <+10>:  add    $0x1,%eax
401141 <+13>:  retq
401142 <+14>:  mov    $0x0,%eax
401147 <+19>:  jmp    0x40113e <test+10>
```

Lecture Plan

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Set %eax (i) to 0.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

jg means “jump if greater than”. This jumps if %eax > 0x63. The flags indicate this is false, so we do not jump.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Add 1 to %eax (i).

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Jump to another instruction.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We continue in this pattern until we make this conditional jump. When will that be?

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We will stop looping when this comparison says that %eax – 0x63 > 0!

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Then, we return from the function.

GCC Common While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Test
Skip loop if test passes
Body
Jump back to test
```

From Previous Slide:

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

GCC Other While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

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

From Previous Slide:

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Lecture Plan

Common For Loop Construction

C For loop

```
for (init; test; update) {
      body
}
```

C Equivalent While Loop

```
init
while(test) {
     body
     update
}
```

Assembly pseudocode

```
Init
Test
Skip loop if test passes
Body
Update
Jump back to test
```

For loops and while loops are treated (essentially) the same when compiled down to assembly.

Back to Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```
0000000000401136 <sum_array>:
  401136 <+0>:   mov    $0x0,%eax
  40113b <+5>:   mov    $0x0,%edx
  401140 <+10>:  cmp    %esi,%eax
  401142 <+12>:  jge    0x40114f <sum_array+25>
  401144 <+14>:  movslq %eax,%rcx
  401147 <+17>:  add    (%rdi,%rcx,4),%edx
  40114a <+20>:  add    $0x1,%eax
  40114d <+23>:  jmp    0x401140 <sum_array+10>
  40114f <+25>:  mov    %edx,%eax
  401151 <+27>:  retq
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jge`** instructions doing?
   (**`jge`**: signed jump greater than/equal)

Demo: GDB and Assembly

`sum_array.c`

gdb tips

`layout split` (ctrl-x a: exit, ctrl-l: resize)	View C, assembly, and gdb (lab5)
`info reg`	Print all registers
`p $eax`	Print register value
`p $eflags`	Print all condition codes currently set
`b *0x400546`	Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`	Set **conditional breakpoint**
`ni`	Next assembly instruction
`si`	Step into assembly instruction (will step into function calls)

gdb tips

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address

Lecture Plan

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **jmp** instructions conditionally jump to a different next instruction
- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data

`set`: Read condition codes

`set` instrucnons condinonally set a byte to 0 or 1.

- Reads current state of flags
- Desnnanon is a single-byte register (e.g., %al) or single-byte memory locanon
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

cmov: Conditional move

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition `x` holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp     %edi,%esi
mov     %edi, %eax
cmovge  %esi, %eax
retq
```

cmov: Conditional move

Instruction	Synonym	Move Condition
cmove S,R	cmovz	Equal / zero (ZF = 1)
cmovne S,R	cmovnz	Not equal / not zero (ZF = 0)
cmovs S,R		Negative (SF = 1)
cmovns S,R		NonnegaOve (SF = 0)
cmovg S,R	cmovnle	Greater (signed >) (SF = 0 and SF = OF)
cmovge S,R	cmovnl	Greater or equal (signed >=) (SF = OF)
cmovl S,R	cmovnge	Less (signed <) (SF != OF)
cmovle S,R	cmovng	Less or equal (signed <=) (ZF = 1 or SF! = OF)
cmova S,R	cmovnbe	Above (unsigned >) (CF = 0 and ZF = 0)
cmovae S,R	cmovnb	Above or equal (unsigned >=) (CF = 0)
cmovb S,R	cmovnae	Below (unsigned <) (CF = 1)
cmovbe S,R	cmovna	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Last Lab: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

Put x + 3 into %eax

Check the sign of x

If x is positive, put x into %eax

Divide %eax by 4

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

Next time: Function calls in assembly

How to remember cmp/jmp

- `CMP S1, S2` is `S2 – S1` (just sets condition codes). **But generally**:

```
cmp S1, S2
jg  …
```

⟺ **`S2 > S1`** ⟺ **`S2 - S1 > 0`**

- Much less important to remember exact condition codes
  - Yes, they fully explain conditional jmp…
  - …but more important to know how to translate assembly back into C
  - If you're interested, B&O p. 206 has details

Instruction	Synonym	Jump condition	Description
`jmp` *Label*		1	Direct jump
`jmp` **Operand*		1	Indirect jump
`je` *Label*	`jz`	ZF	Equal / zero
`jne` *Label*	`jnz`	~ZF	Not equal / not zero
`js` *Label*		SF	Negative
`jns` *Label*		~SF	Nonnegative
`jg` *Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge` *Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl` *Label*	`jnge`	SF ^ OF	Less (signed <)
`jle` *Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja` *Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae` *Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	CF	Below (unsigned <)
`jbe` *Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Remember `test` exists

- `TEST S1, S2` is `S2 & S1`

```
test %edi, %edi
jns  …
```

`%edi & %edi is nonnegative`

`%edi is nonnegative`

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test

Jump past loop if fails

Body

Jump to test

https://godbolt.org/z/zrW6c5MGa

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
      result = ____(3)_____;
      a = ___(4)___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax
<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>
<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi
<+22>:  jmp    0x113e <loop+5>
<+24>:  retq
```

GCC common while loop construction:

Test
Jump past loop if fails
Body
Jump to test

Practice: Fill in the blanks

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

```
<+0>:    mov     $0x1,%eax

<+5>:    cmp     %rsi,%rdi
<+8>:    jge     0x1151 <loop+24>

<+10>:   lea     (%rdi,%rsi,1),%rdx
<+14>:   imul    %rdx,%rax
<+18>:   add     $0x1,%rdi

<+22>:   jmp     0x113e <loop+5>

<+24>:   retq
```

Practice: Fill in the blanks

```
long loop(long a, long b) {
    long result = ____1____;
    while (__a < b___) {
      result = result*(a+b);
      a = __a + 1___;
    }
    return result;
}
```

```
<+0>:   mov    $0x1,%eax

<+5>:   cmp    %rsi,%rdi
<+8>:   jge    0x1151 <loop+24>

<+10>:  lea    (%rdi,%rsi,1),%rdx
<+14>:  imul   %rdx,%rax
<+18>:  add    $0x1,%rdi

<+22>:  jmp    0x113e <loop+5>

<+24>:  retq
```

`test` practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

`test` practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;
}
```

(or anything like this)

Practice: "Escape Room"

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

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

You don't have to reverse-engineer C code exactly!

Practice: "Escape Room"

```
<escape_room+0>     lea    (%rdi,%rdi,1),%eax
<escape_room+3>     cmp    $0x5,%eax
<escape_room+6>     jg     0x114c <escape_room+19>
<escape_room+8>     cmp    $0x1,%edi
<escape_room+11>    je     0x1152 <escape_room+25>
<escape_room+13>    mov    $0x0,%eax
<escape_room+18>    retq
<escape_room+19>    mov    $0x1,%eax
<escape_room+24>    retq
<escape_room+25>    mov    $0x1,%eax
<escape_room+30>    retq
```

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

First param > 2 or == 1.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

%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++;
    }
}
```

```
0x40113f <+0>:  b8 00 00 00 00  mov  $0x0,%eax
0x401144 <+5>:  83 f8 63        cmp  $0x63,%eax
0x401147 <+8>:  7f 05           jg   40114e <loop2+15>
0x401149 <+10>: 83 c0 01        add  $0x1,%eax
0x40114c <+13>: eb f6           jmp  401144 <loop2+5>
0x40114e <+15>: c3              retq
```

%rip

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

```
0x40113f <+0>:  b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:  83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:  7f 05            jg    40114e <loop2+15>
0x401149 <+10>: 83 c0 01         add   $0x1,%eax
0x40114c <+13>: eb f6            jmp   401144 <loop2+5>
0x40114e <+15>: c3               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++;
    }
}
```

```
0x40113f <+0>:  b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:  83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:  7f 05            jg    40114e <loop2+15>
0x401149 <+10>: 83 c0 01         add   $0x1,%eax
0x40114c <+13>: eb f6            jmp   401144 <loop2+5>
0x40114e <+15>: c3               retq
```

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

%rip

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

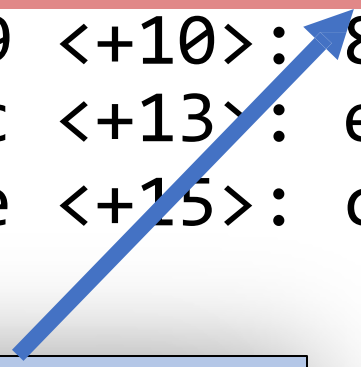
```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

%rip

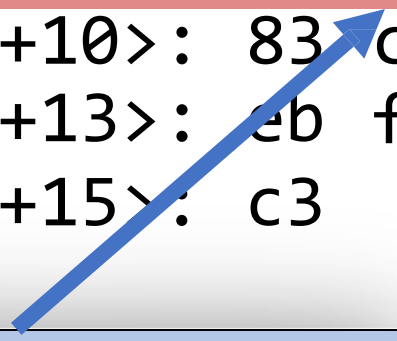
```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



0x7f means **jg**.

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

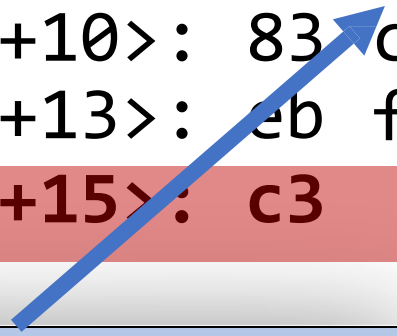


0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jg** says to <u>then</u> go **5** bytes further!

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

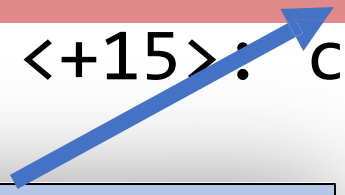


0x05 is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jg** says to <u>then</u> go **5** bytes further!

%rip

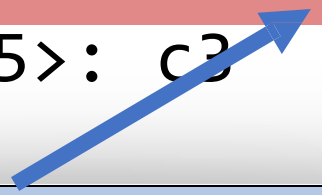
```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



0xeb means **jmp**.

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```

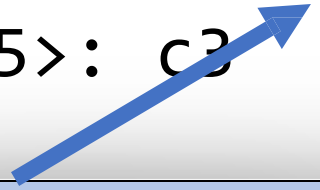


0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **10** bytes back!

%rip

```
0x40113f <+0>:   b8 00 00 00 00   mov   $0x0,%eax
0x401144 <+5>:   83 f8 63         cmp   $0x63,%eax
0x401147 <+8>:   7f 05            jg    40114e <loop2+15>
0x401149 <+10>:  83 c0 01         add   $0x1,%eax
0x40114c <+13>:  eb f6            jmp   401144 <loop2+5>
0x40114e <+15>:  c3               retq
```



0xf6 is the number of instruction bytes to jump relative to %rip. This is -10 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **10** 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.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

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?

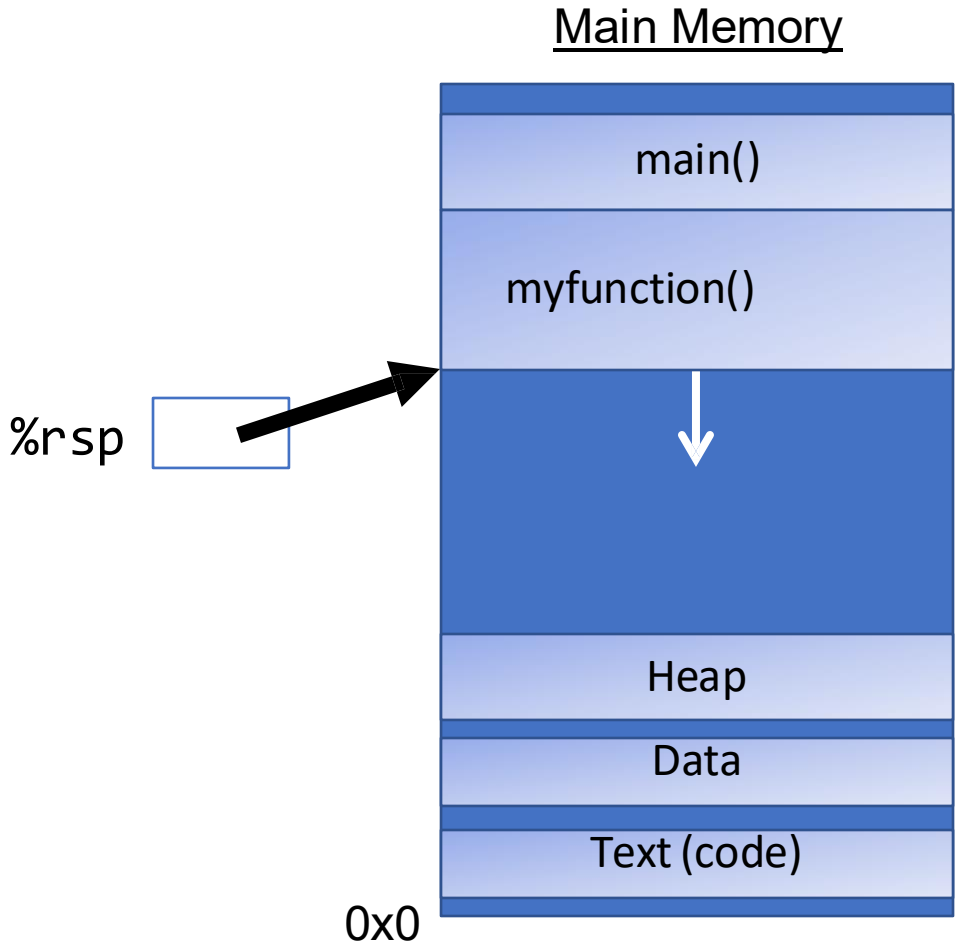
Terminology: **caller** function calls the **callee** function.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

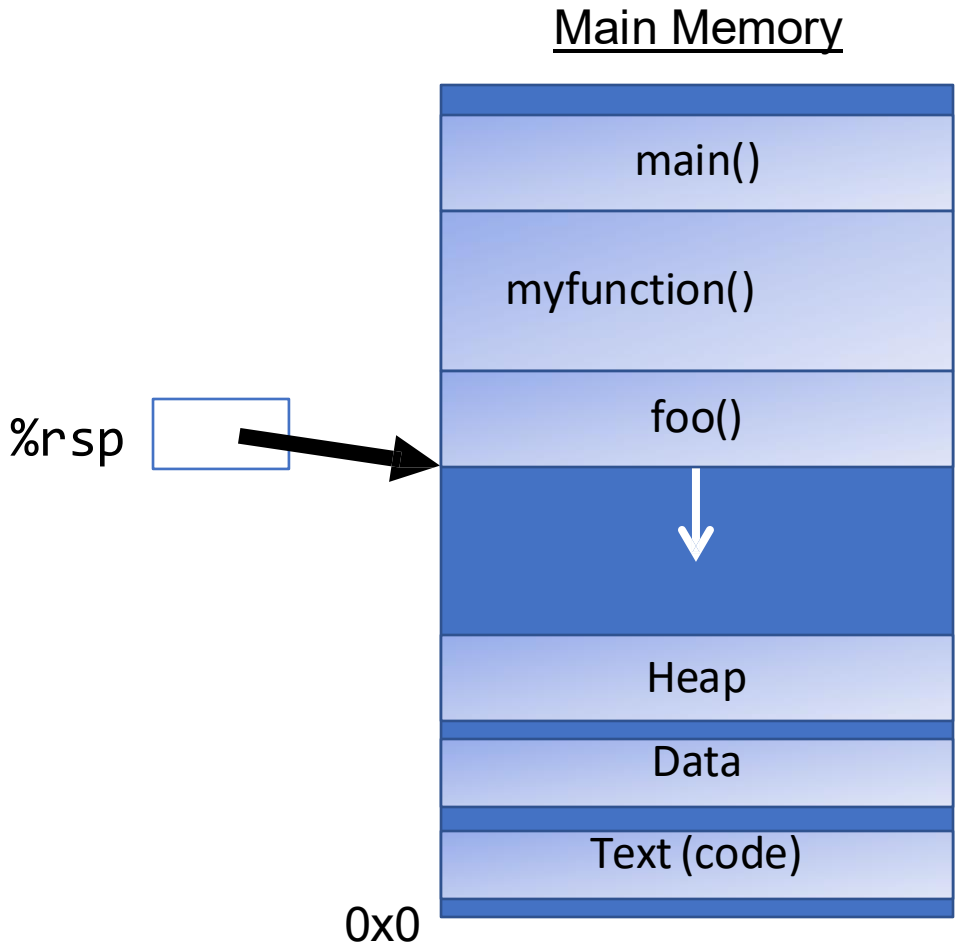
%rsp

- **%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).



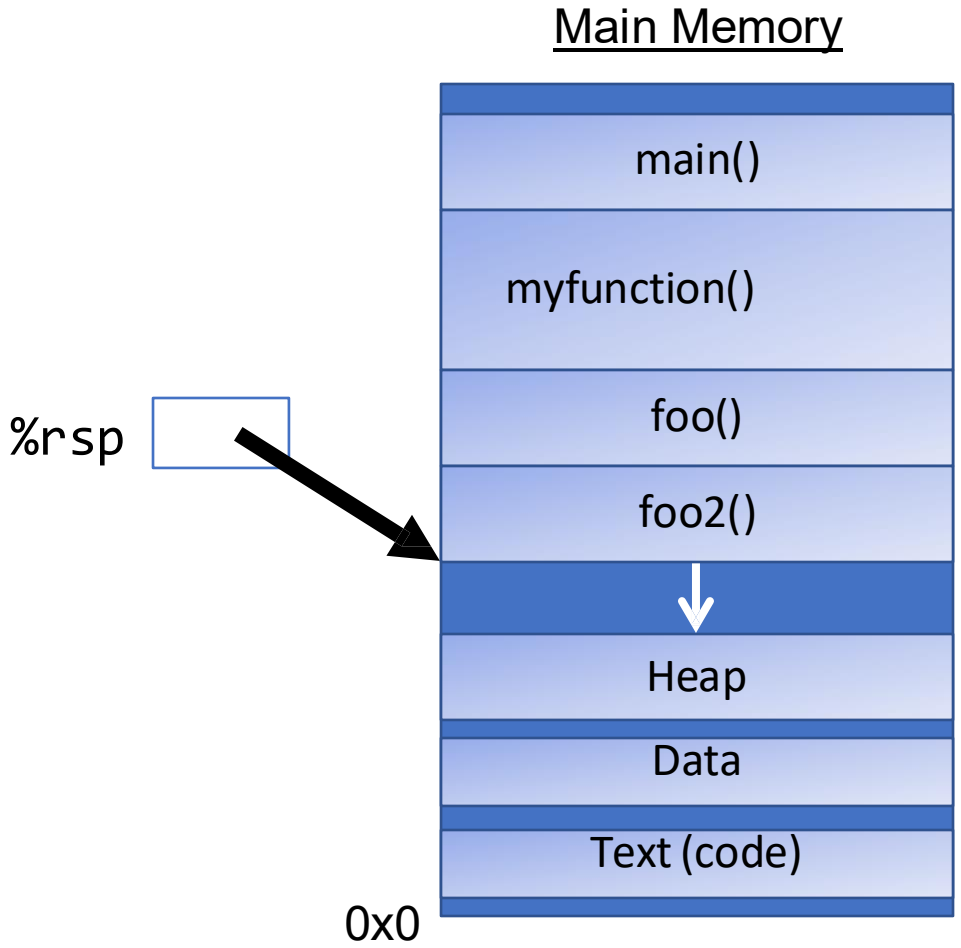
%rsp

- **%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).



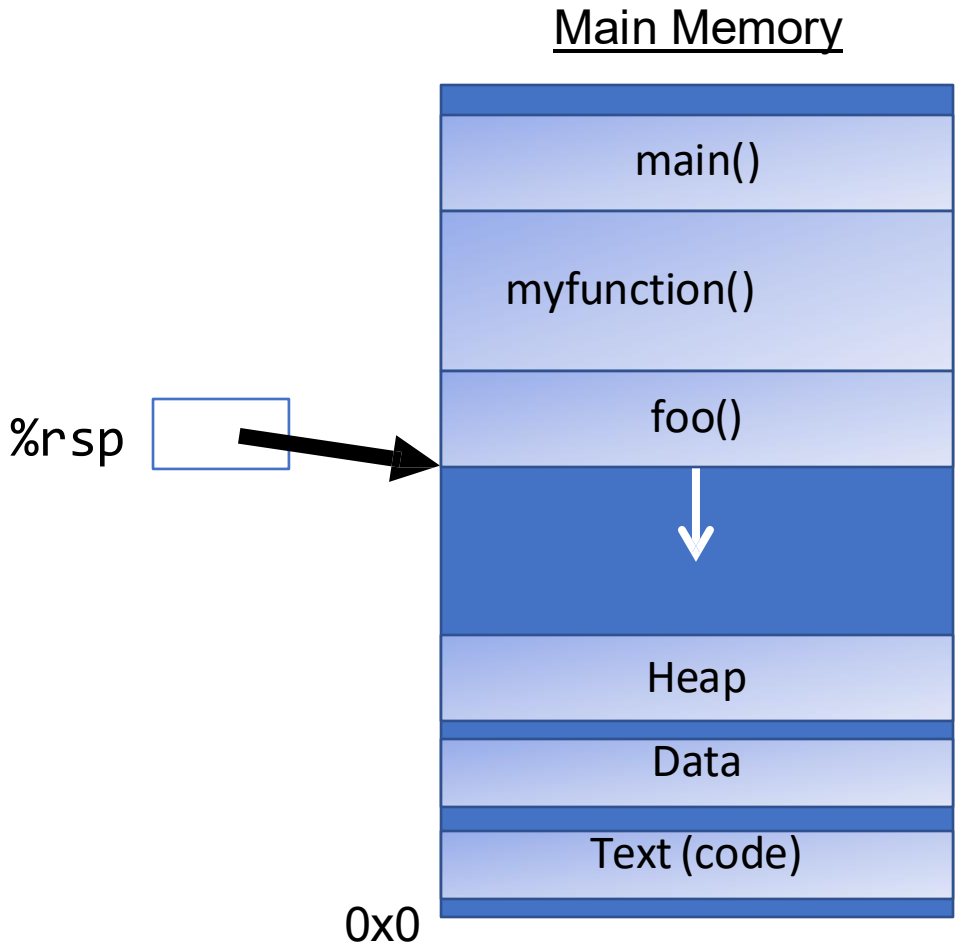
%rsp

- **%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).



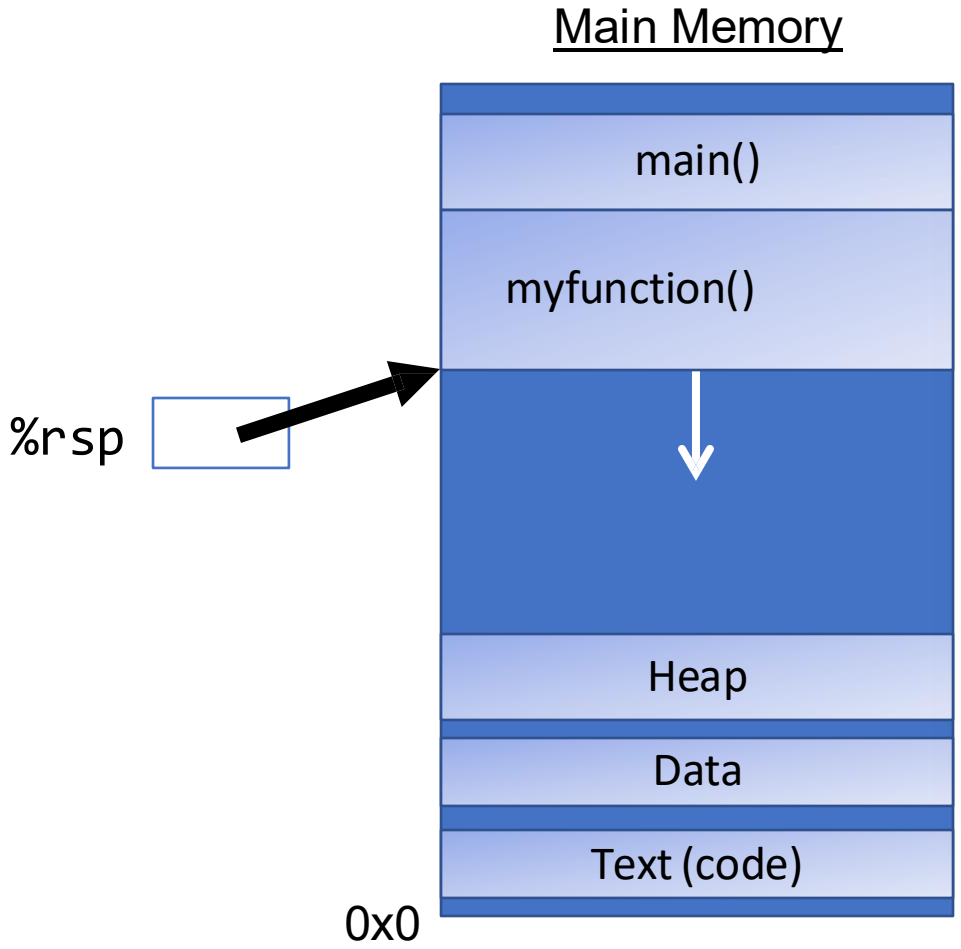
%rsp

- **%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.

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
```

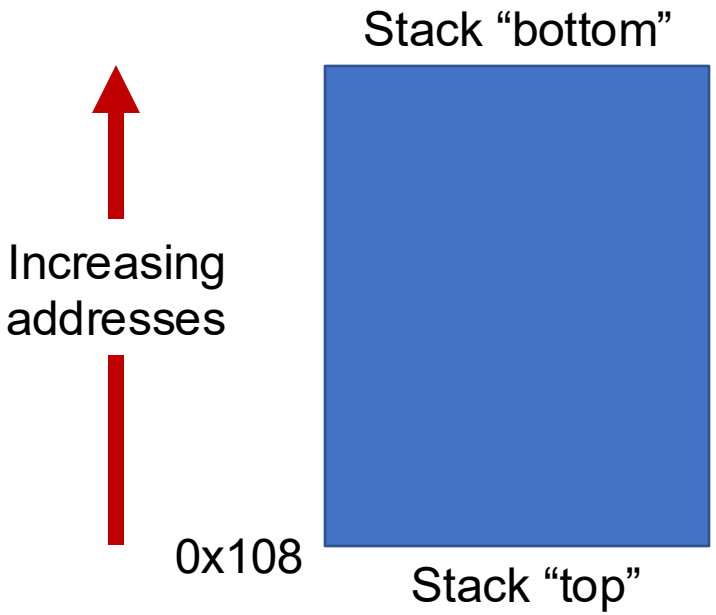
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

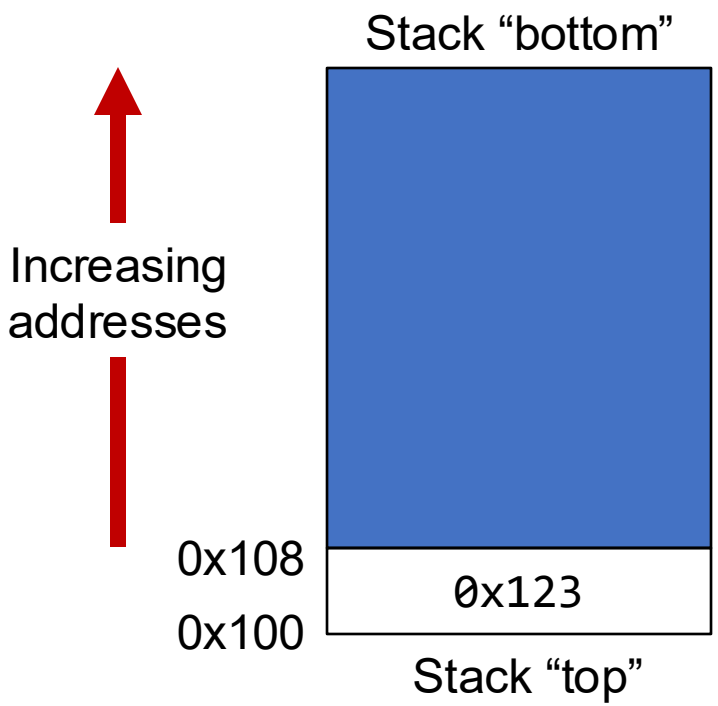
Stack Example

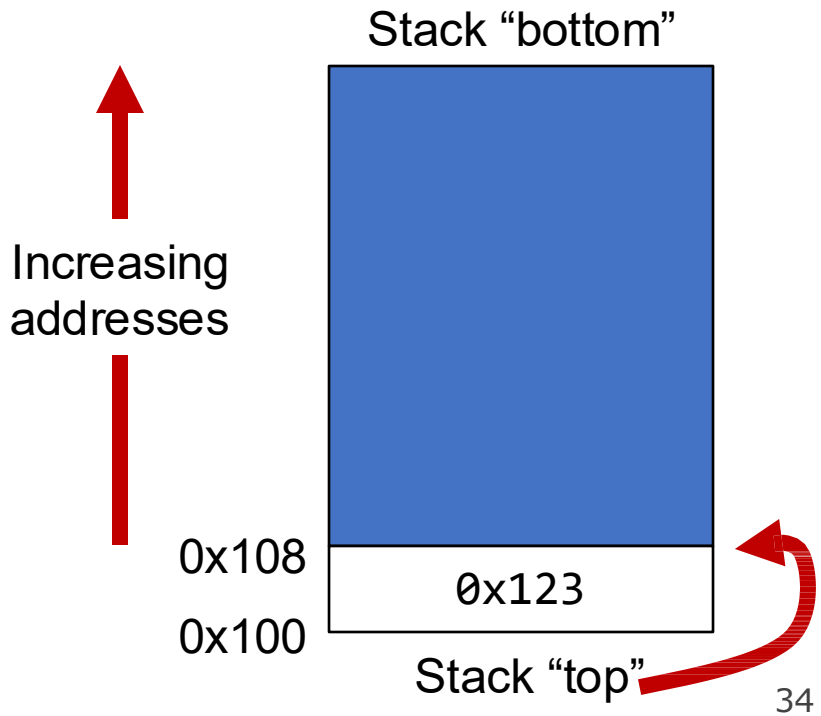
Initially	
%rax	0x123
%rdx	0
%rsp	0x108

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.

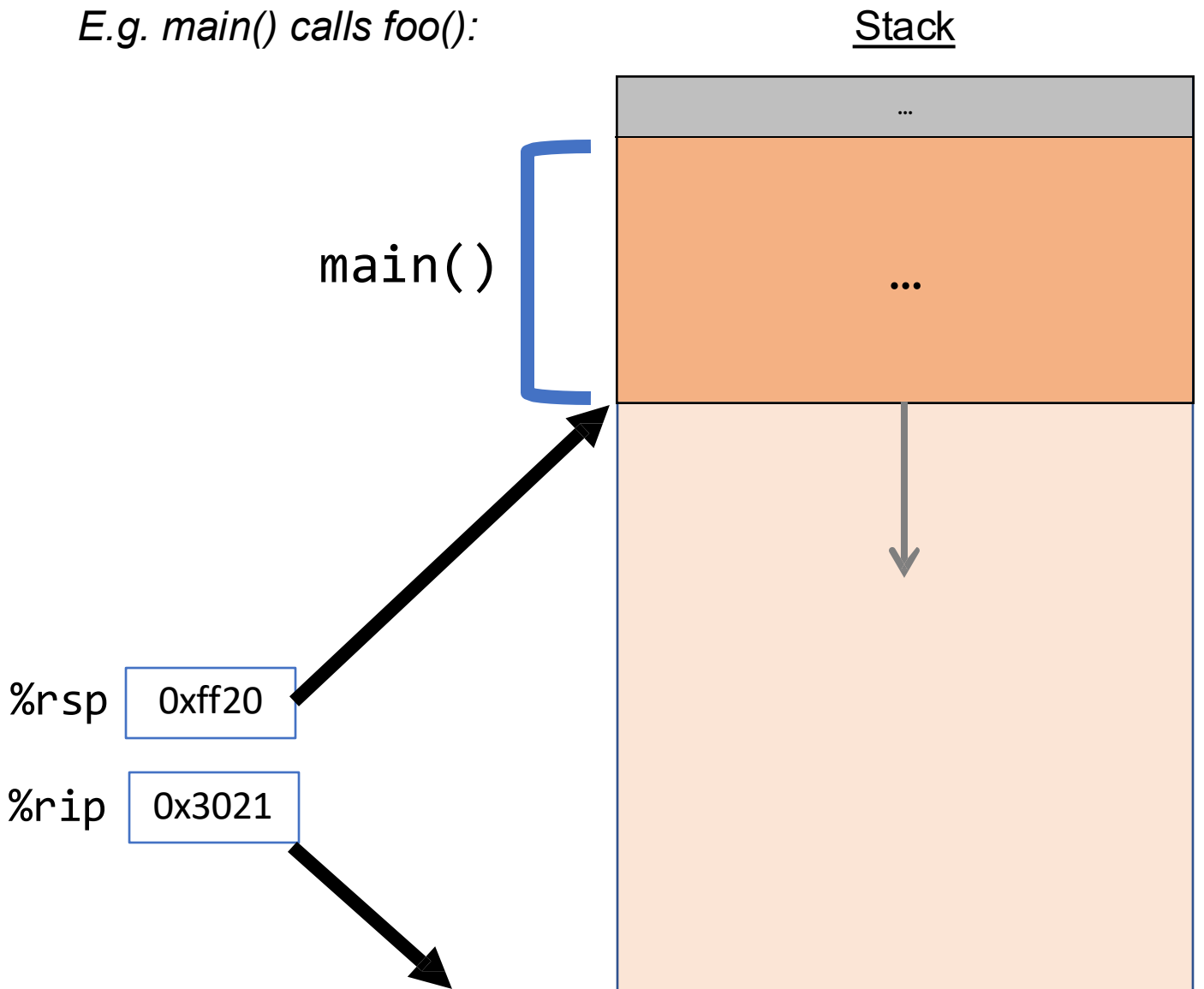
Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

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.

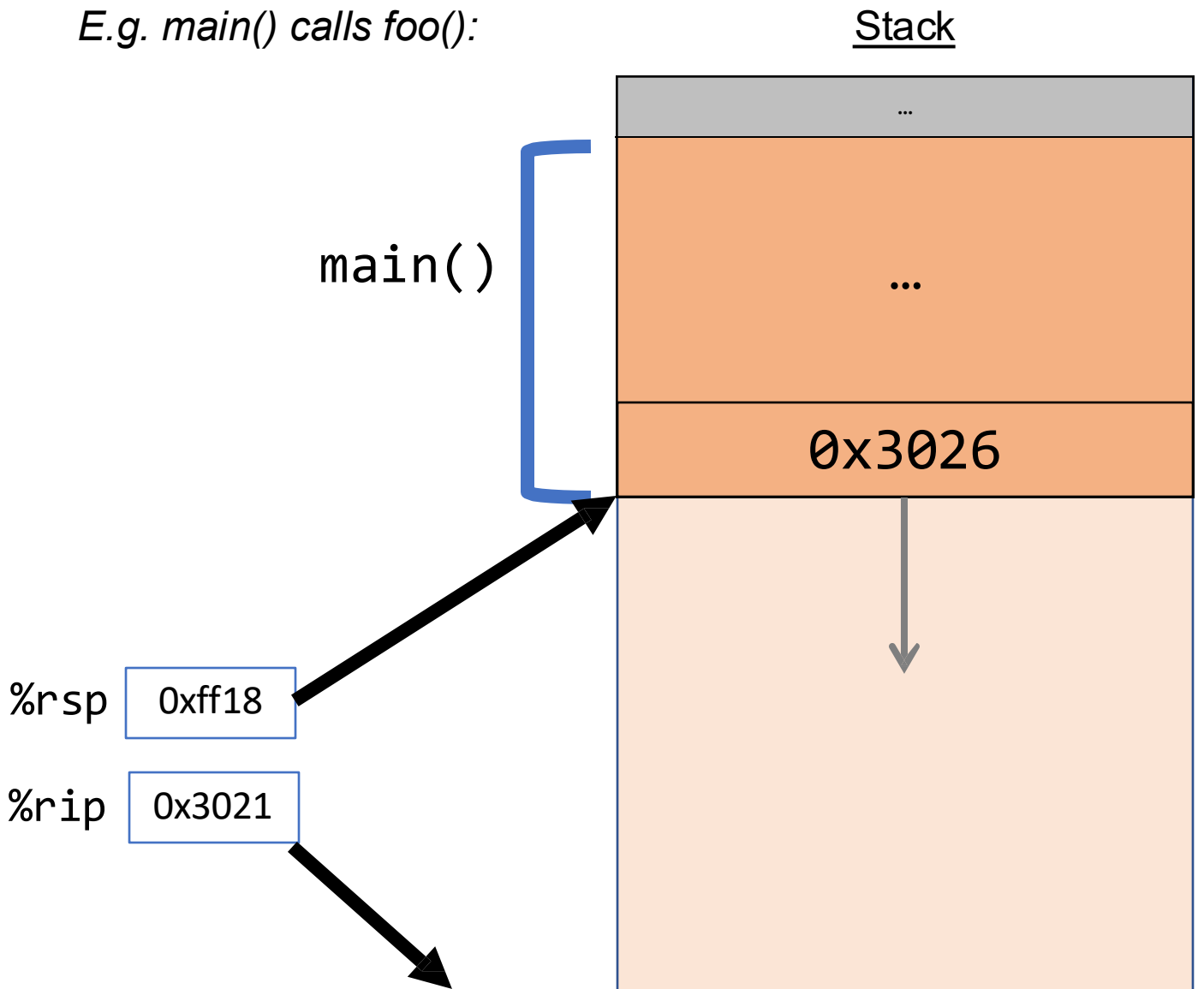
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

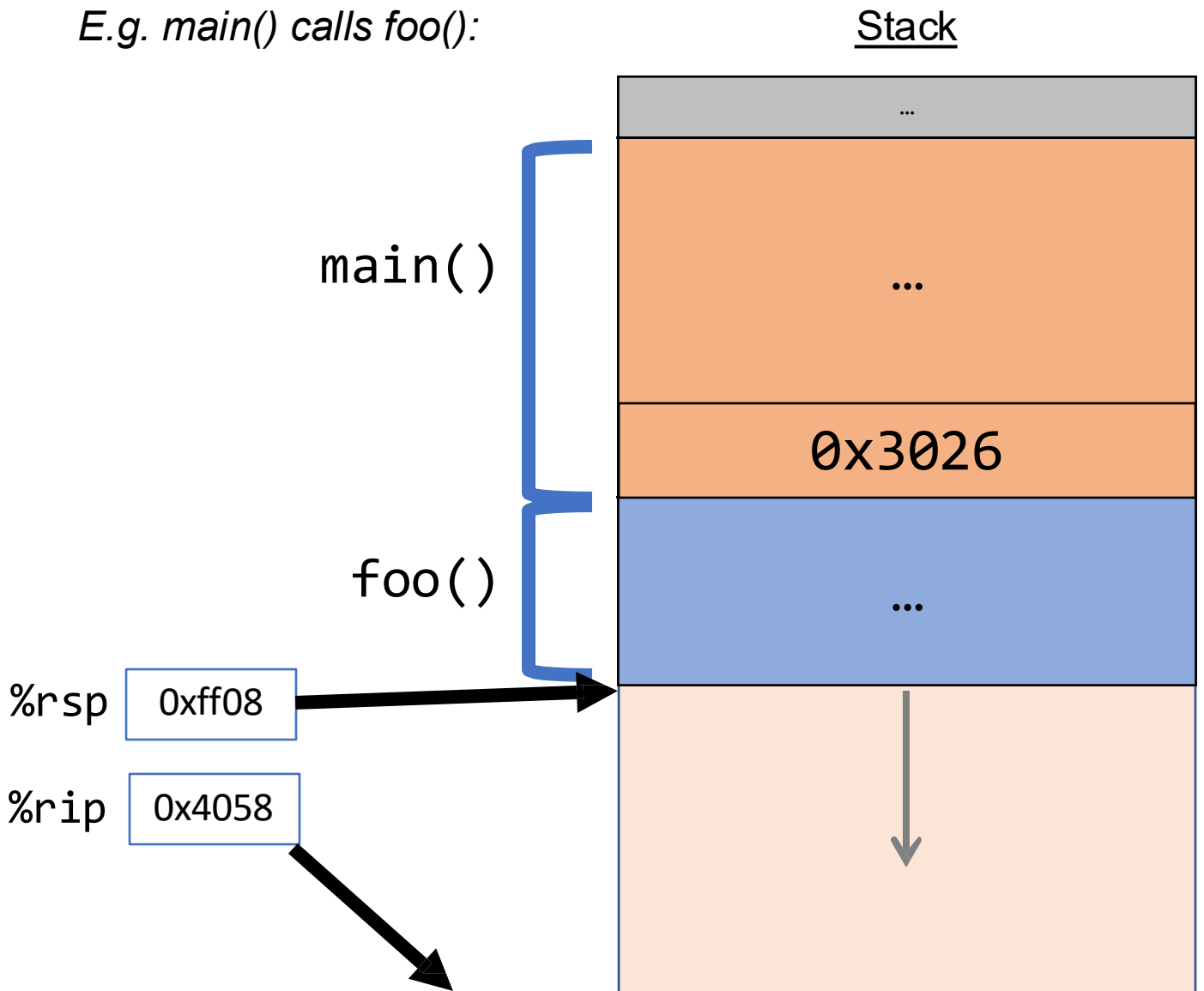
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

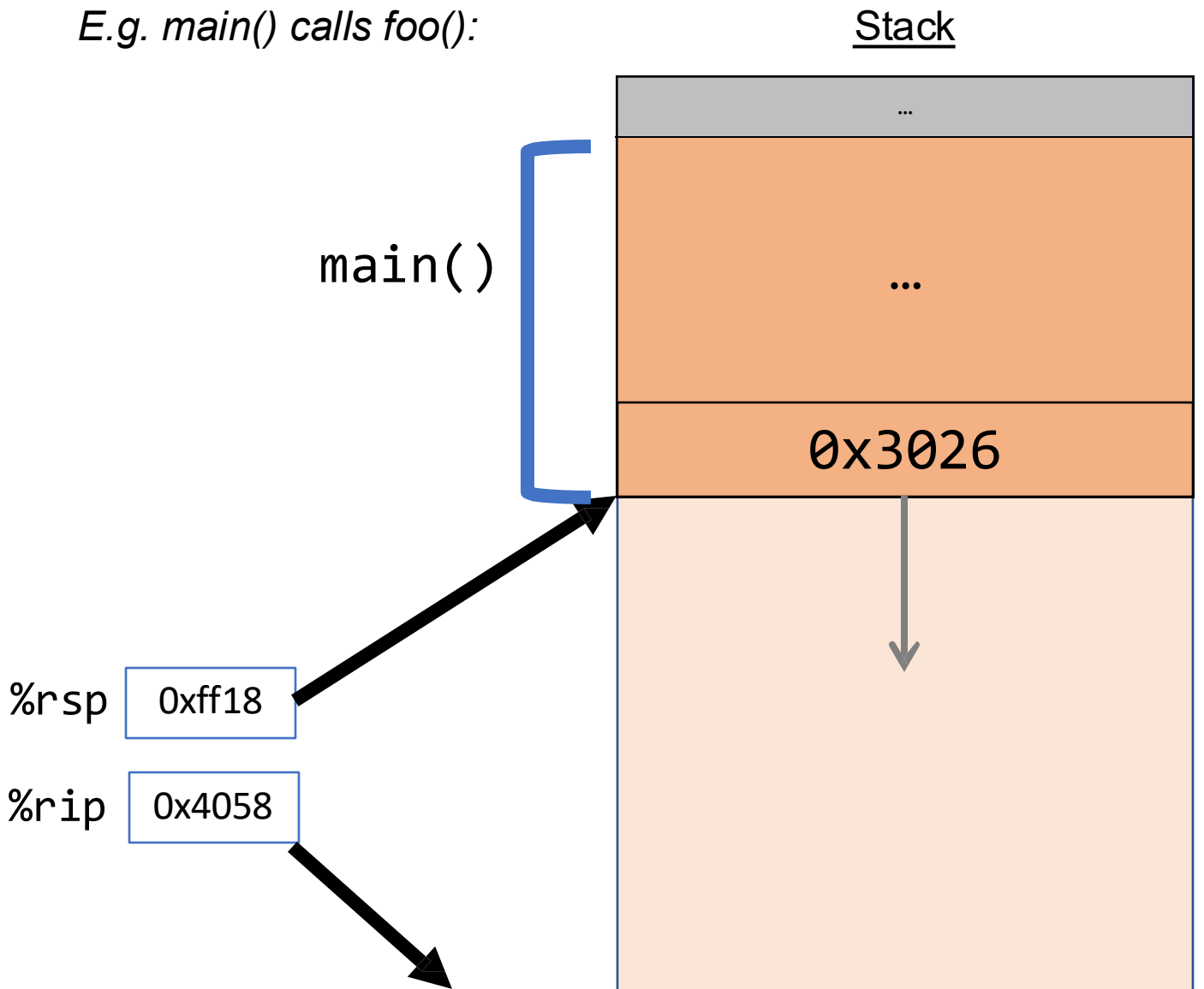
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.

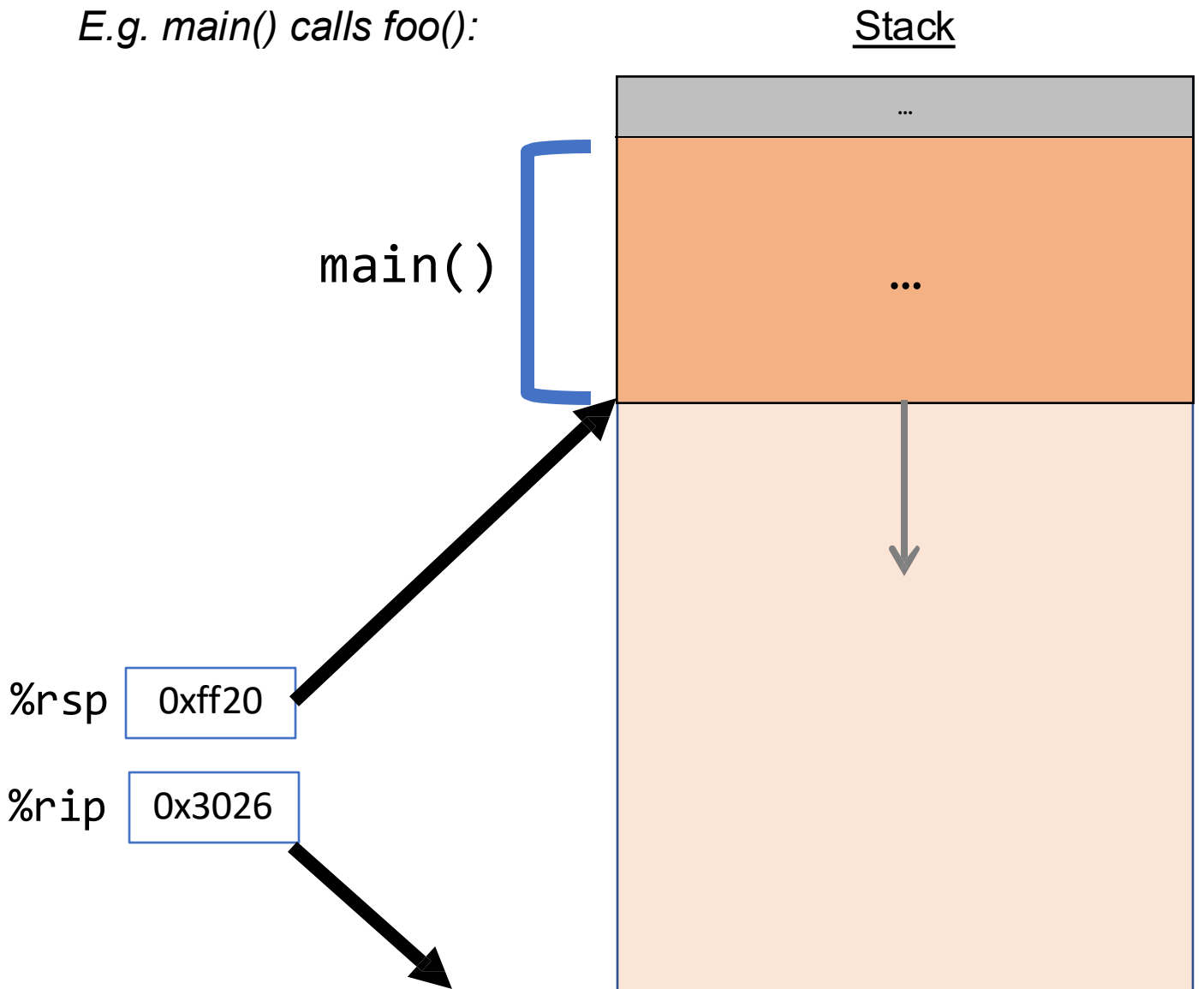
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



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.



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).

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.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Parameters and Return

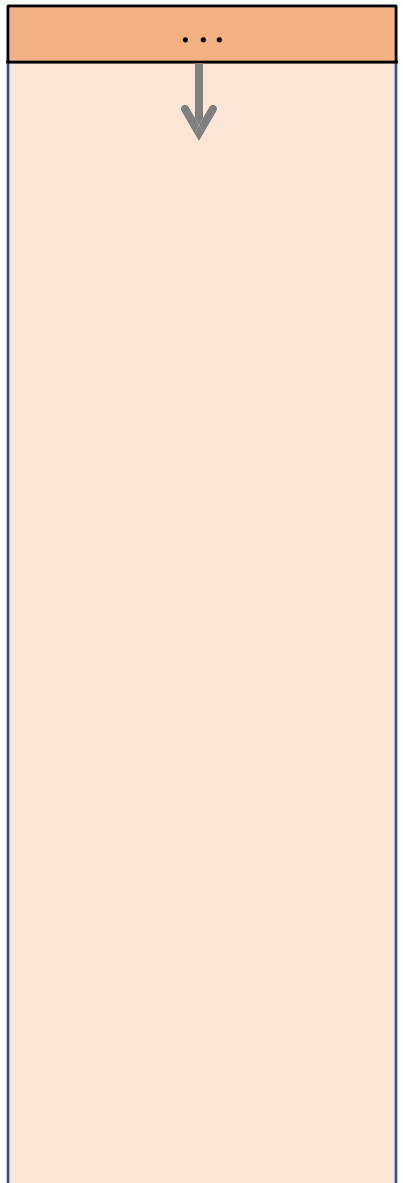
- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return value, it looks in %rax after the callee completes.

Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

main()



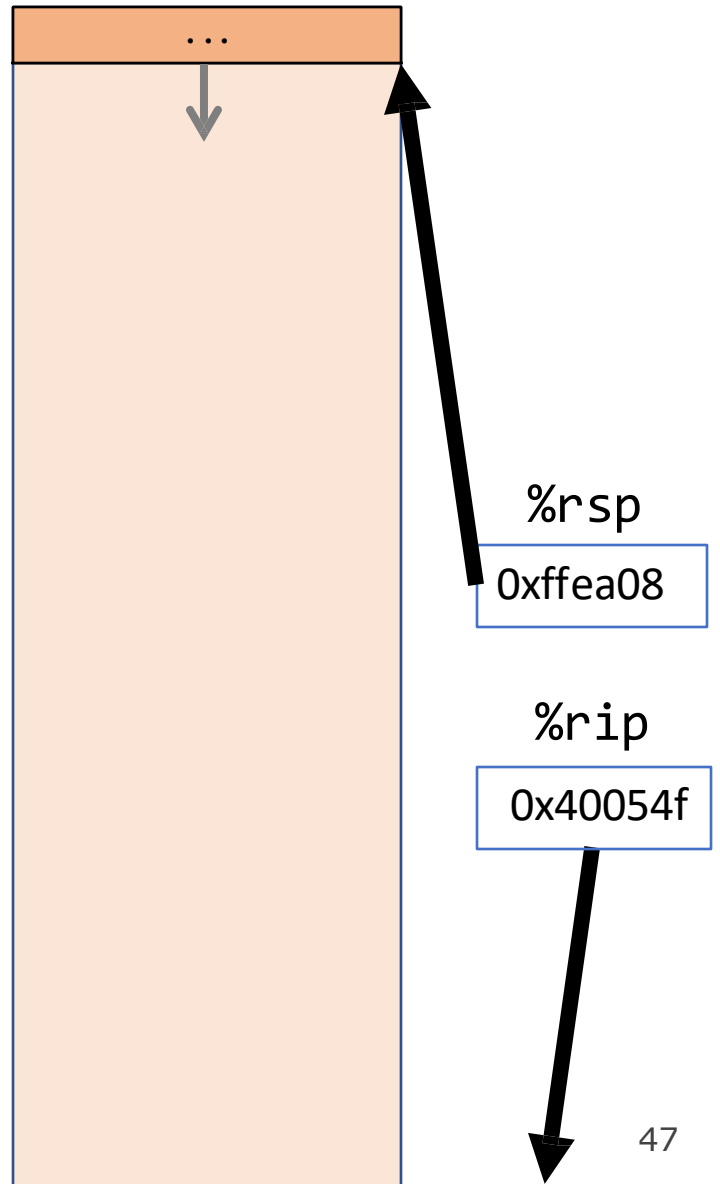
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

main()

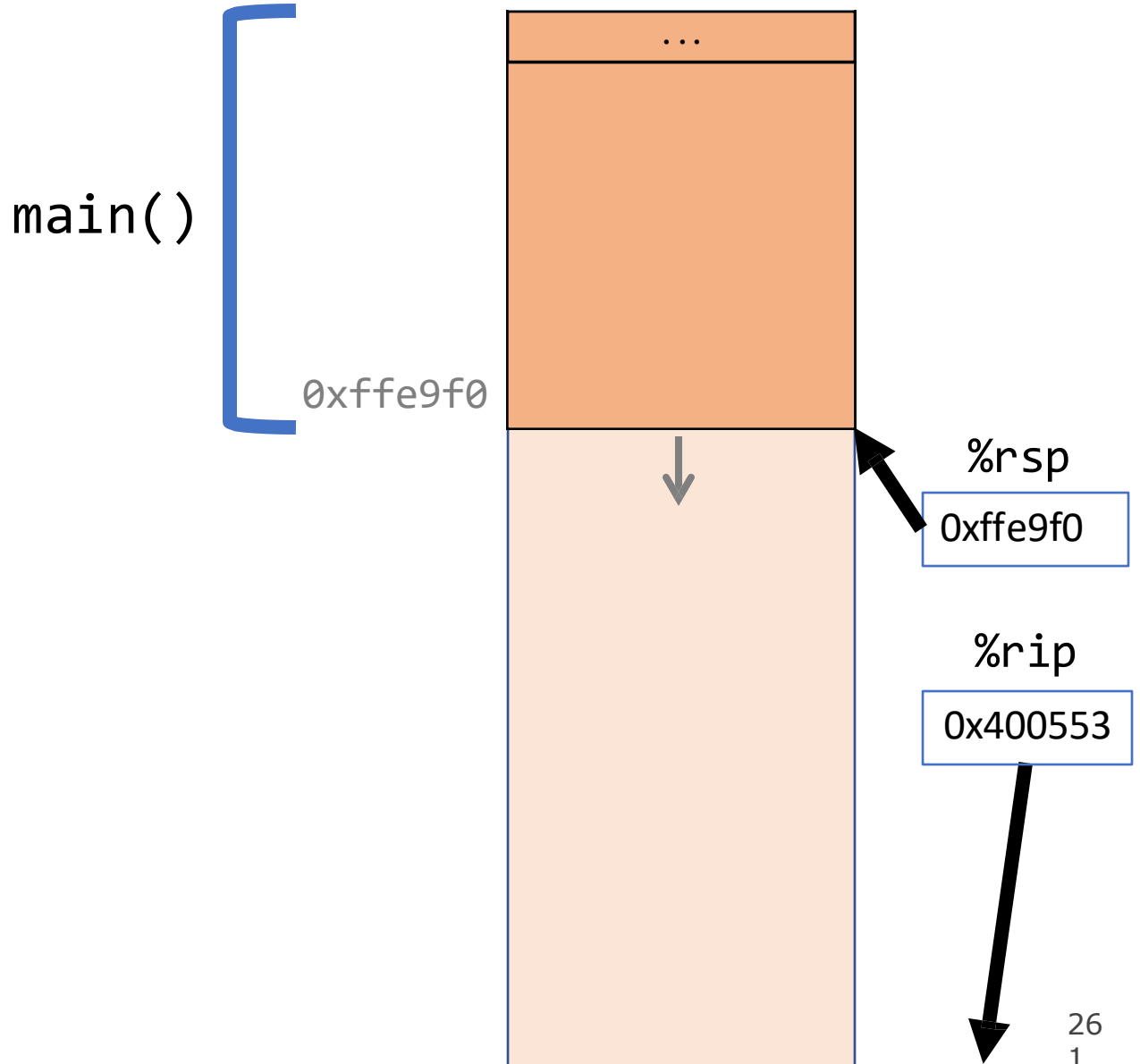


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

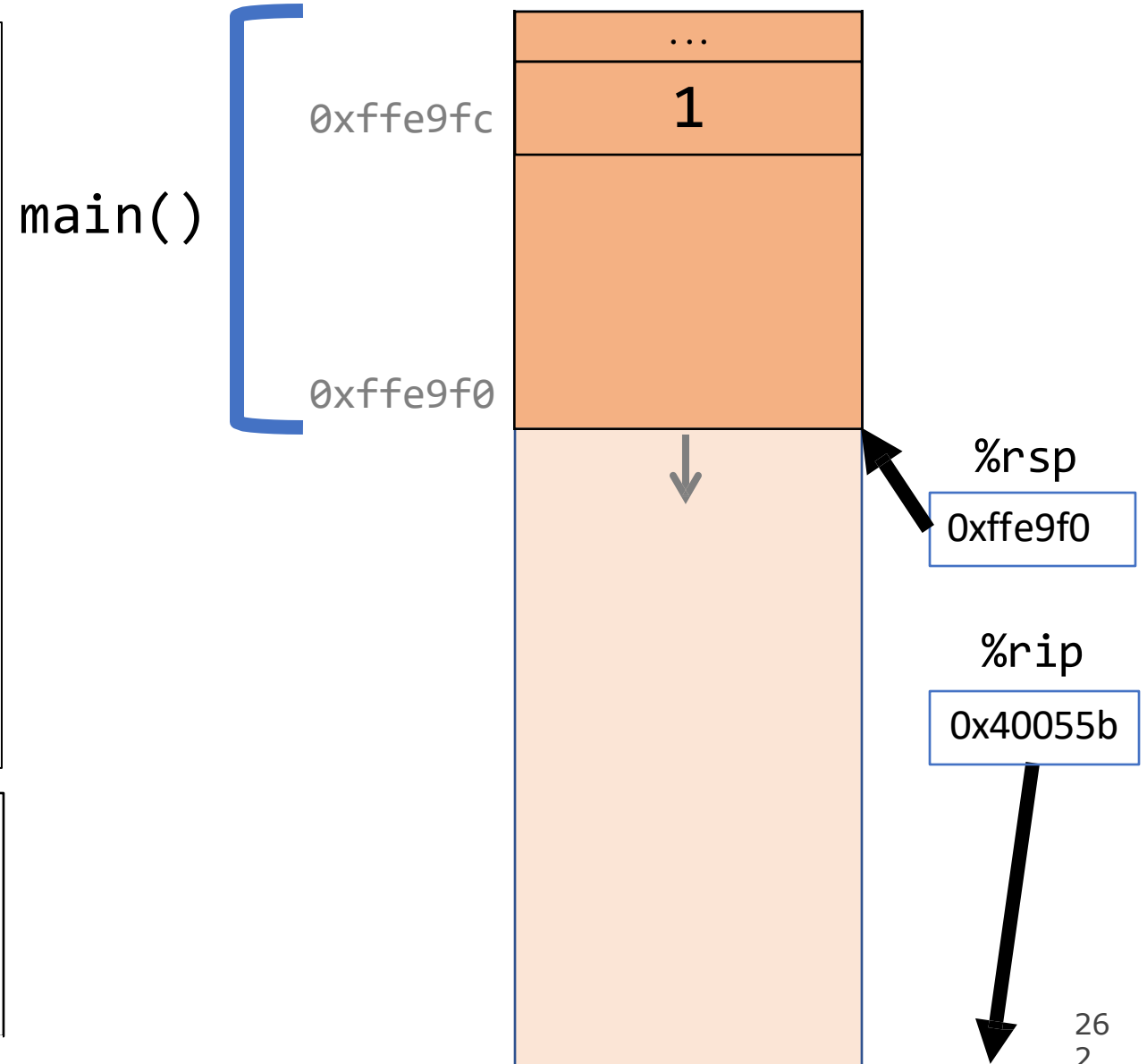


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

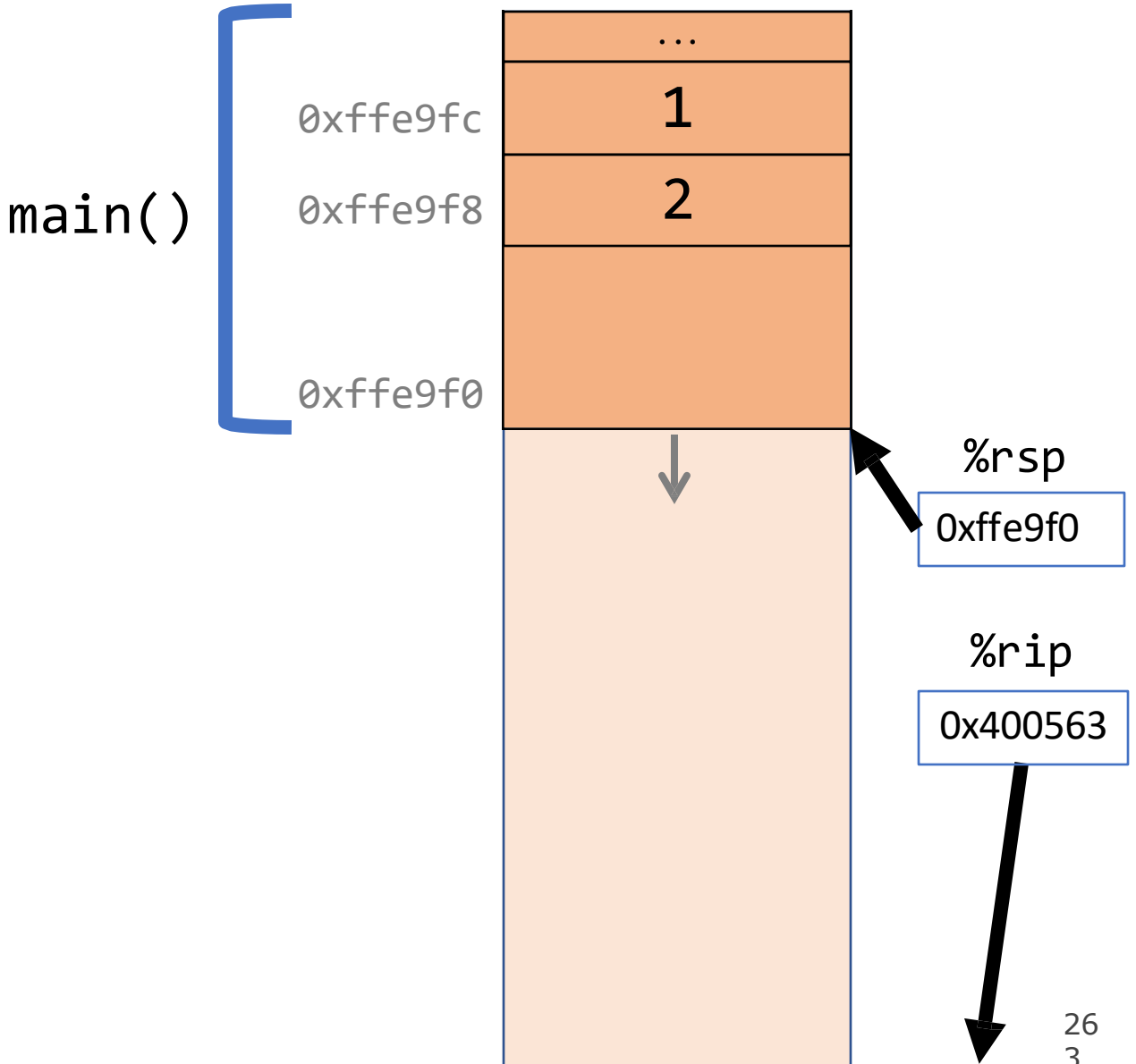


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

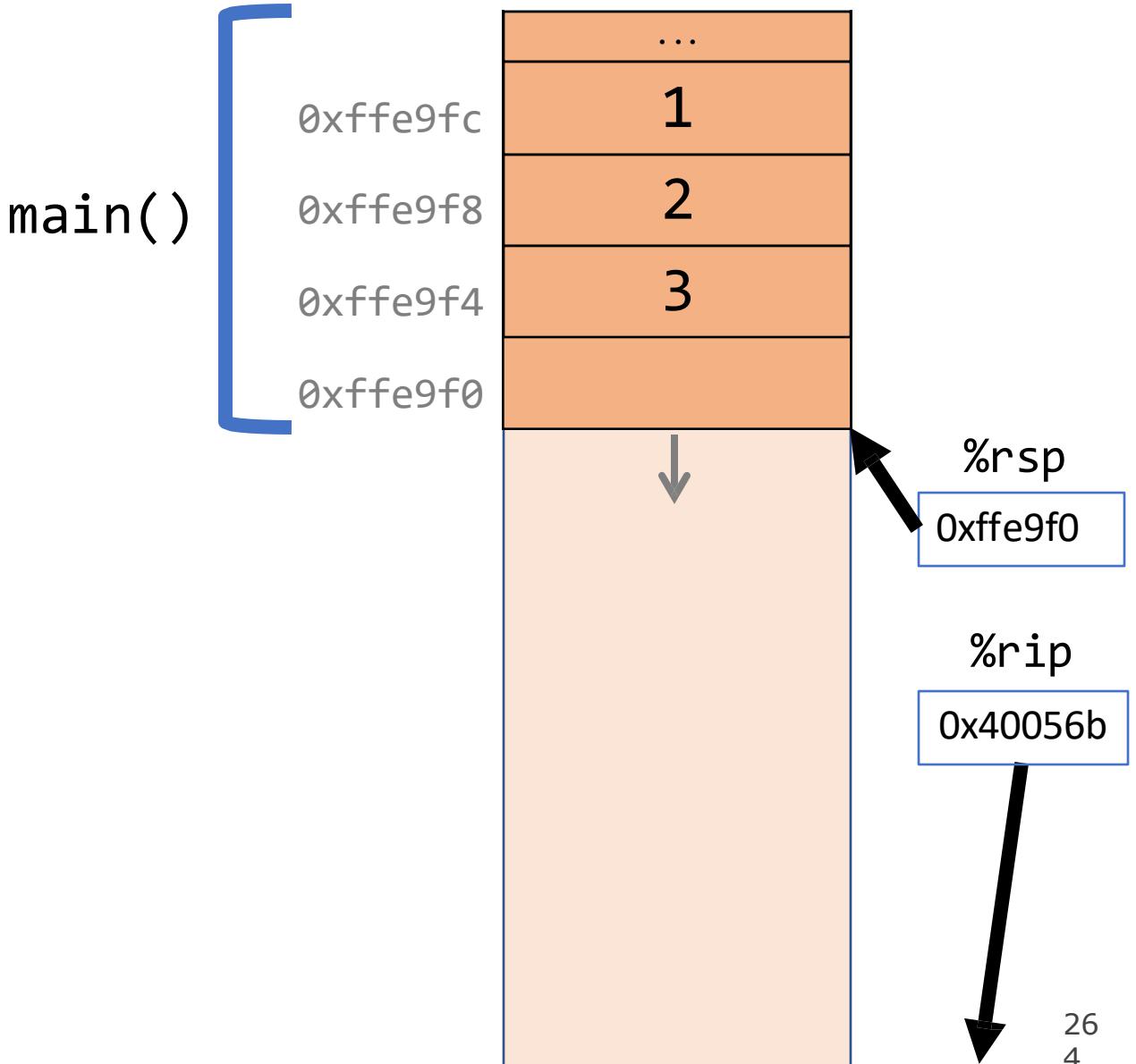


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
```

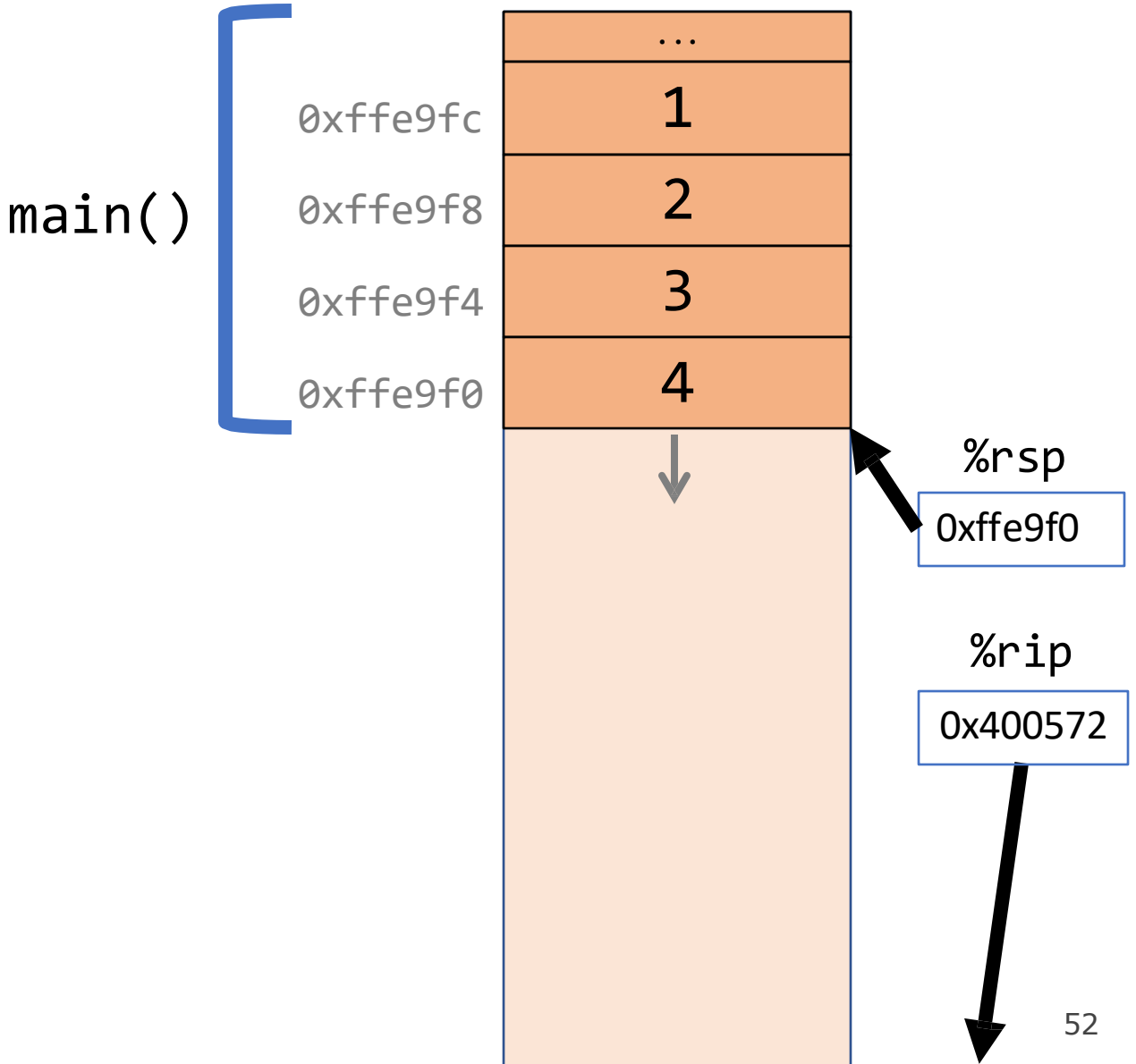


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40055b <+12>:    movl   $0x2,0x8(%rsp)
0x400563 <+20>:    movl   $0x3,0x4(%rsp)
0x40056b <+28>:    movl   $0x4,(%rsp)
0x400572 <+35>:    pushq  $0x4
0x400574 <+37>:    pushq  $0x3
```

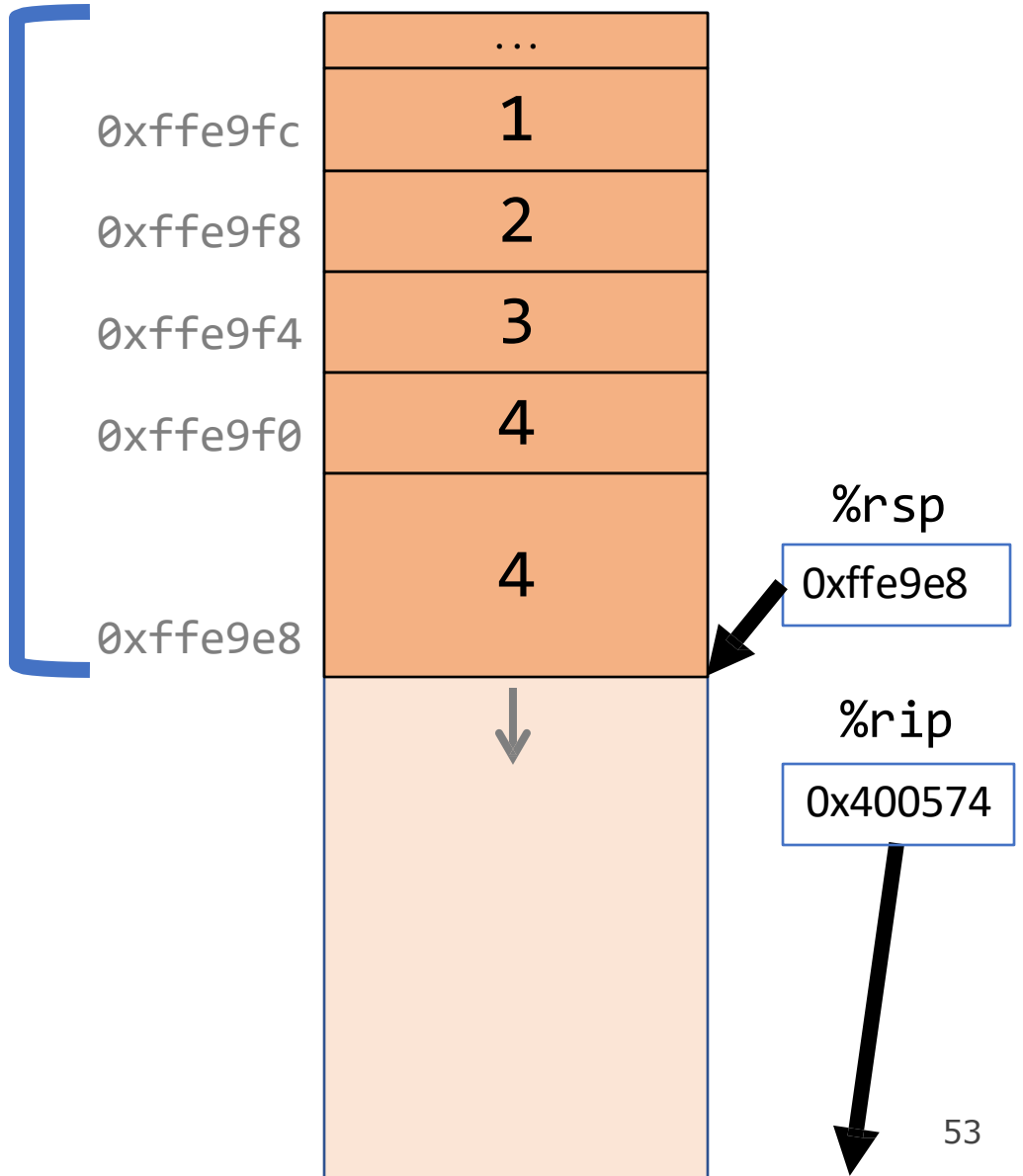


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400563 <+20>:    movl   $0x3,0x4(%rsp)
0x40056b <+28>:    movl   $0x4,(%rsp)
0x400572 <+35>:    pushq  $0x4
0x400574 <+37>:    pushq  $0x3
0x400576 <+39>:    mov    $0x2,%r9d
```

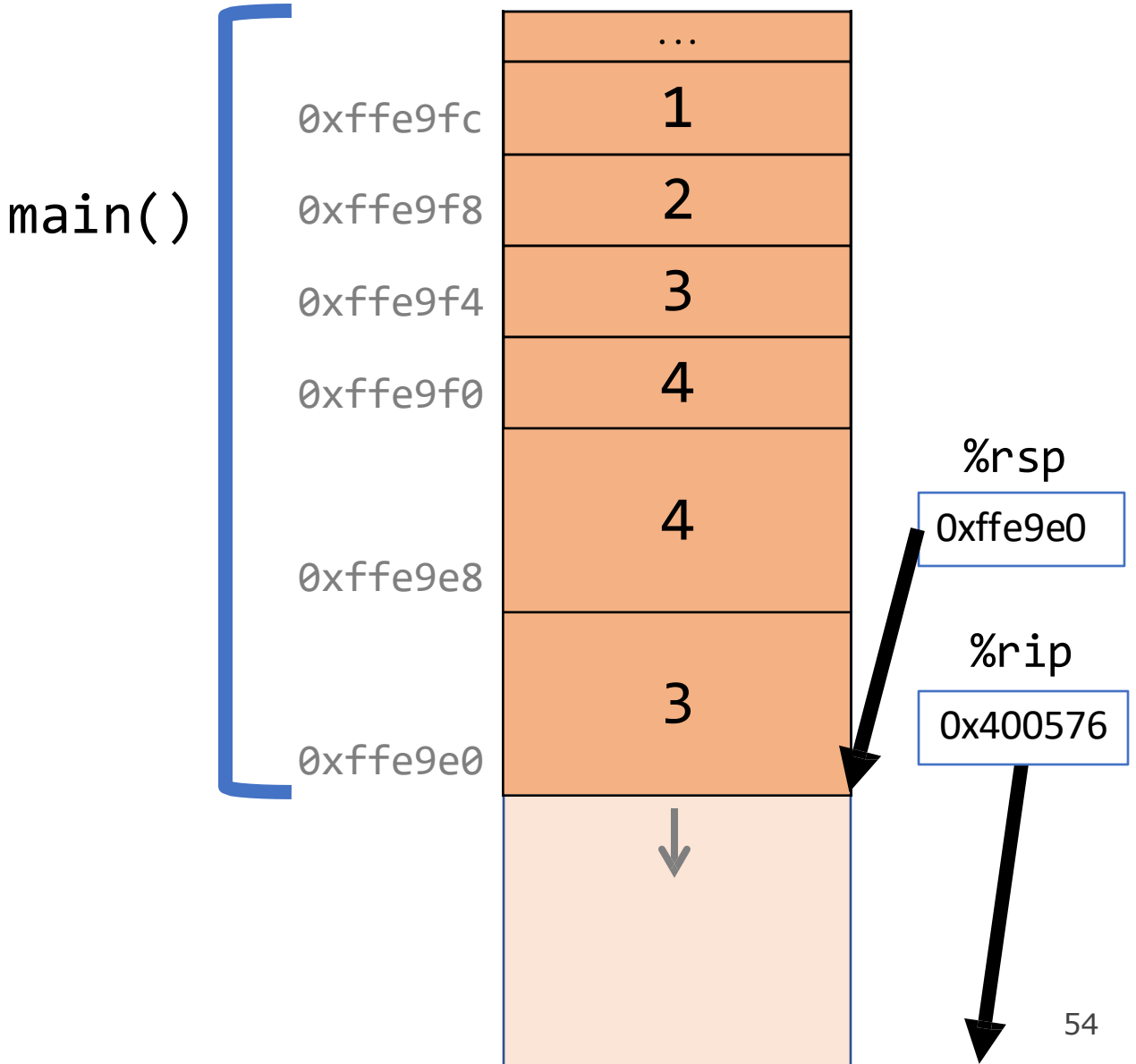


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40056b <+28>:    movl   $0x4,(%rsp)
0x400572 <+35>:    pushq  $0x4
0x400574 <+37>:    pushq  $0x3
0x400576 <+39>:    mov    $0x2,%r9d
0x40057c <+45>:    mov    $0x1,%r8d
```

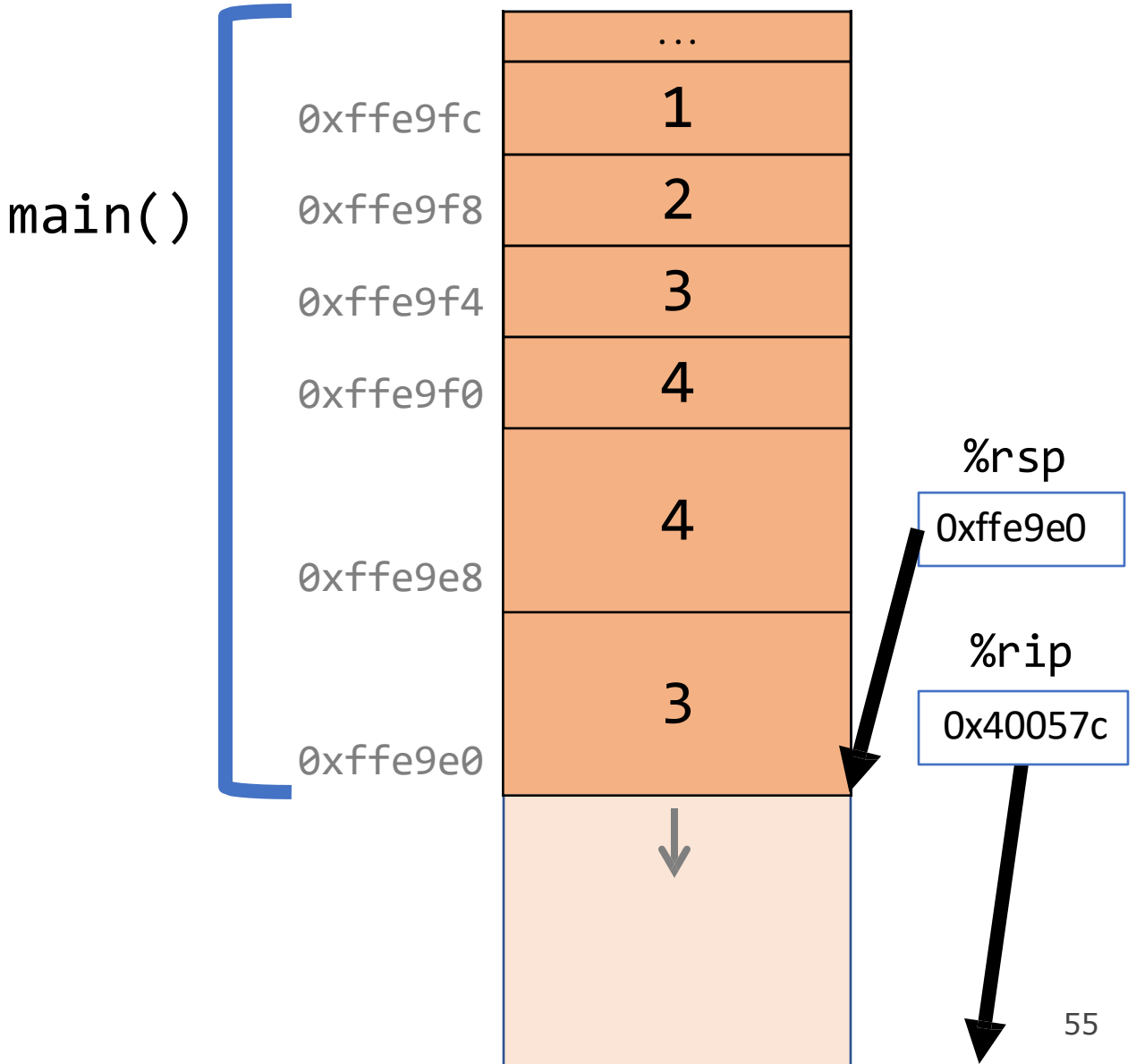


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```



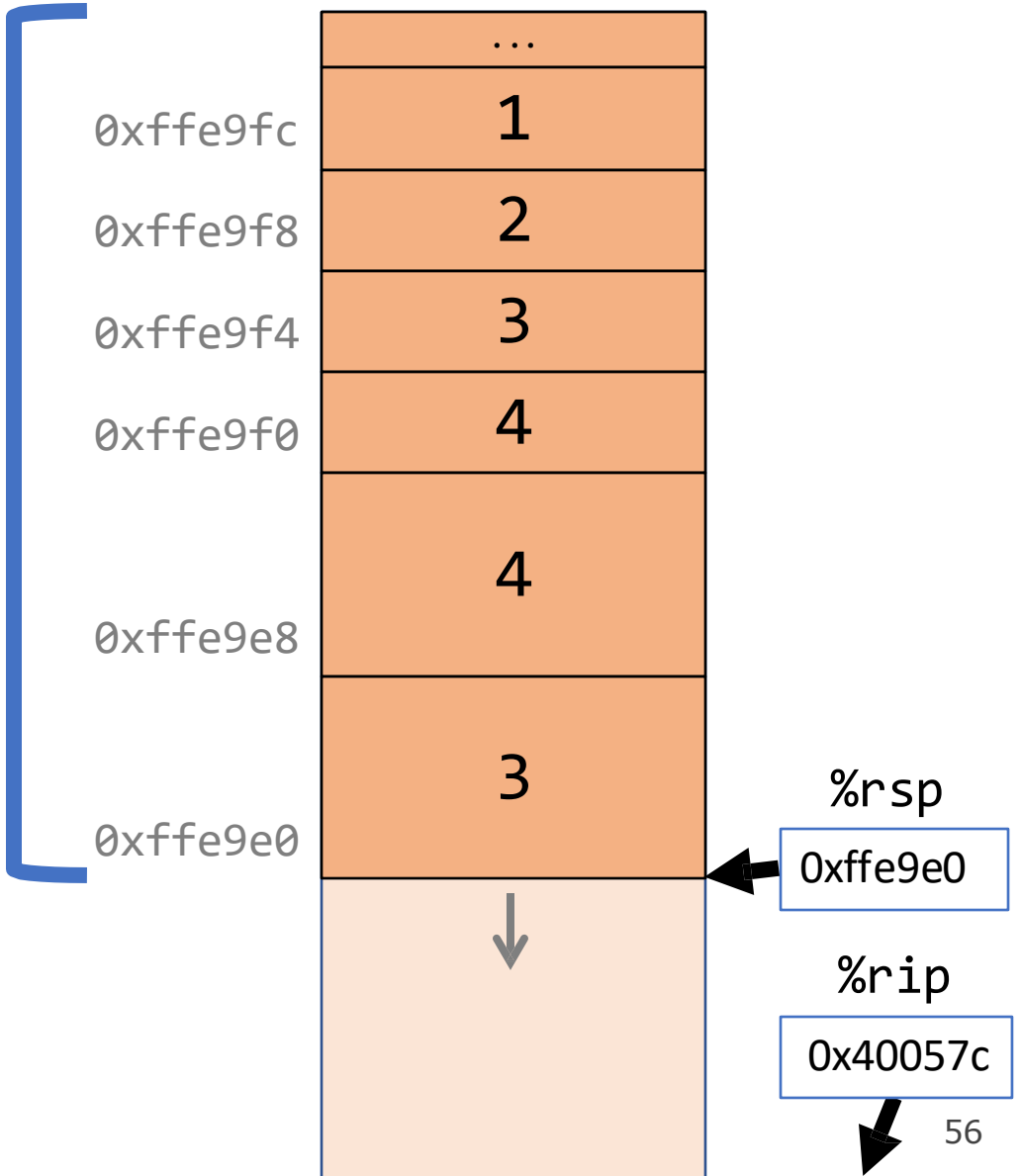
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```

main()



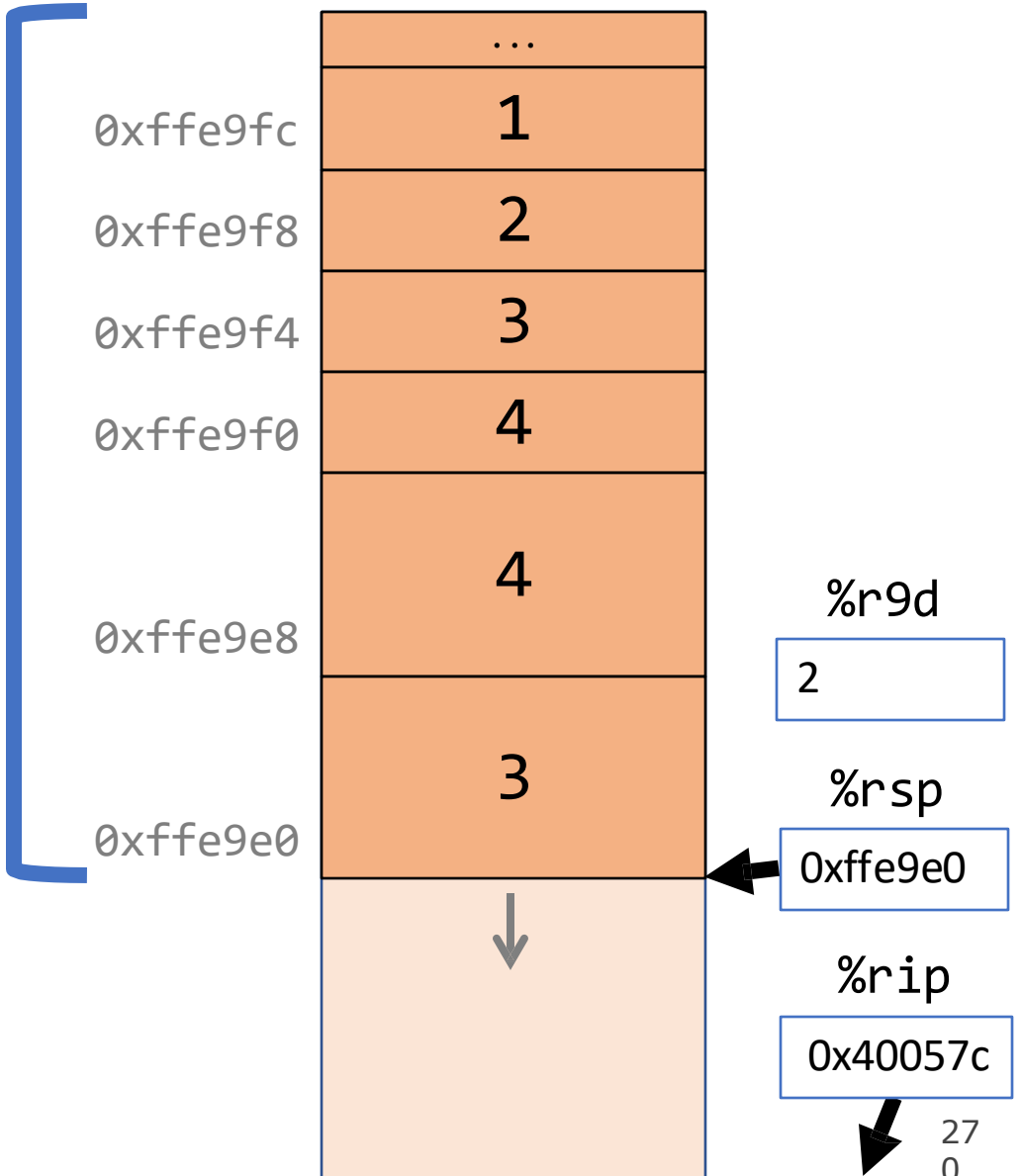
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```

main()

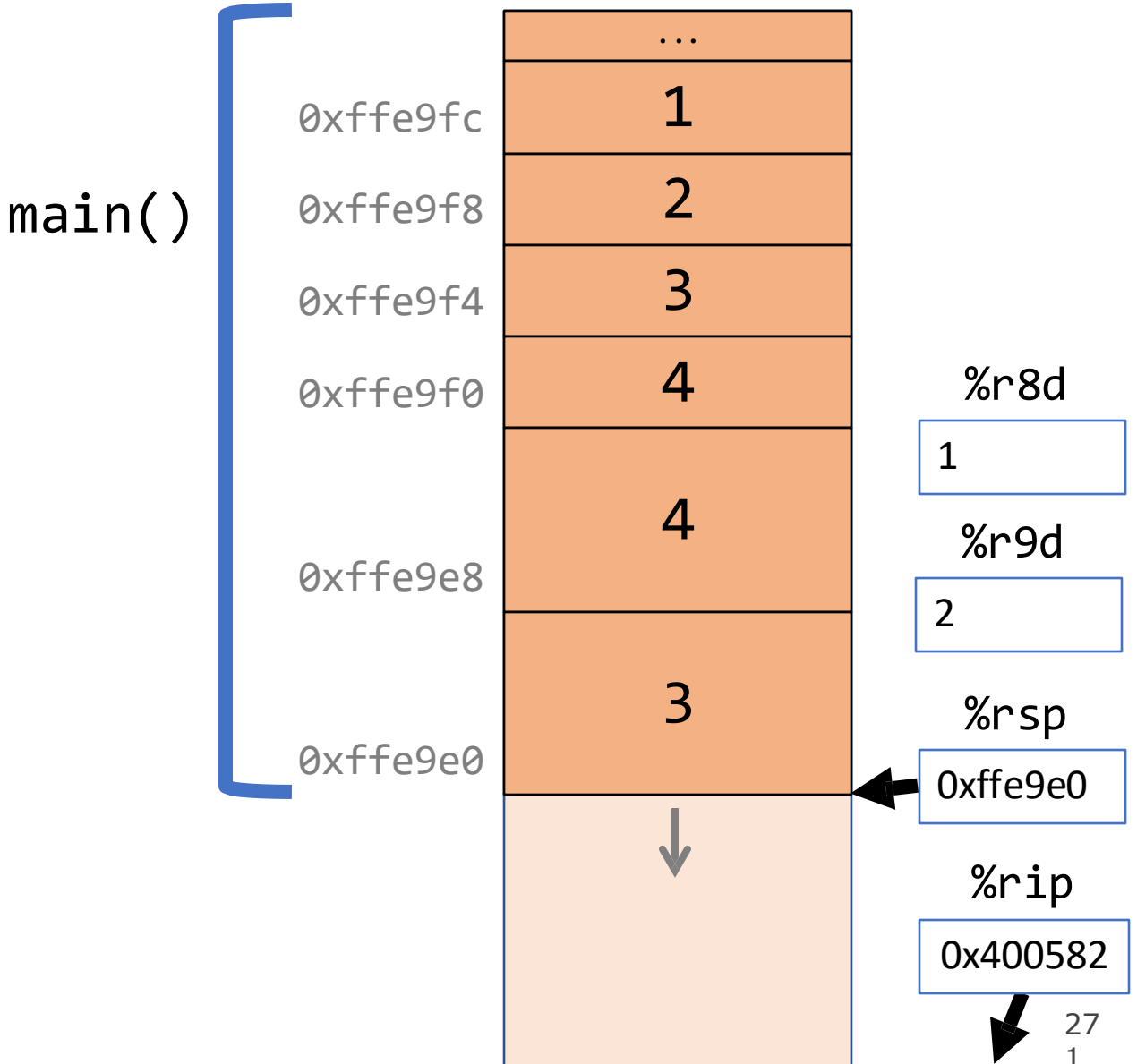


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
0x400587 <+56>:    lea     0x14(%rsp),%rdx
```

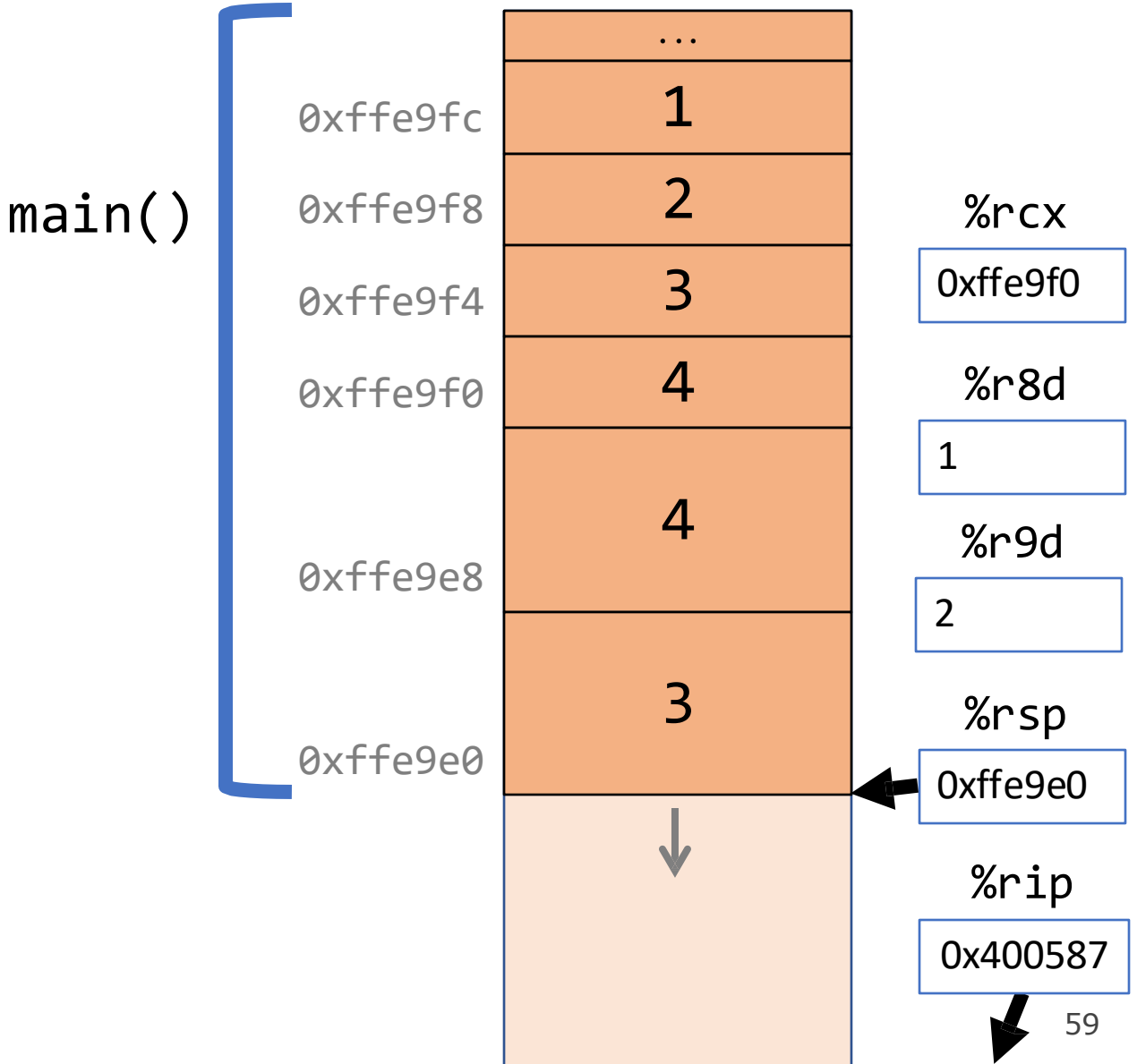


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400576 <+39>:    mov    $0x2,%r9d
0x40057c <+45>:    mov    $0x1,%r8d
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
```



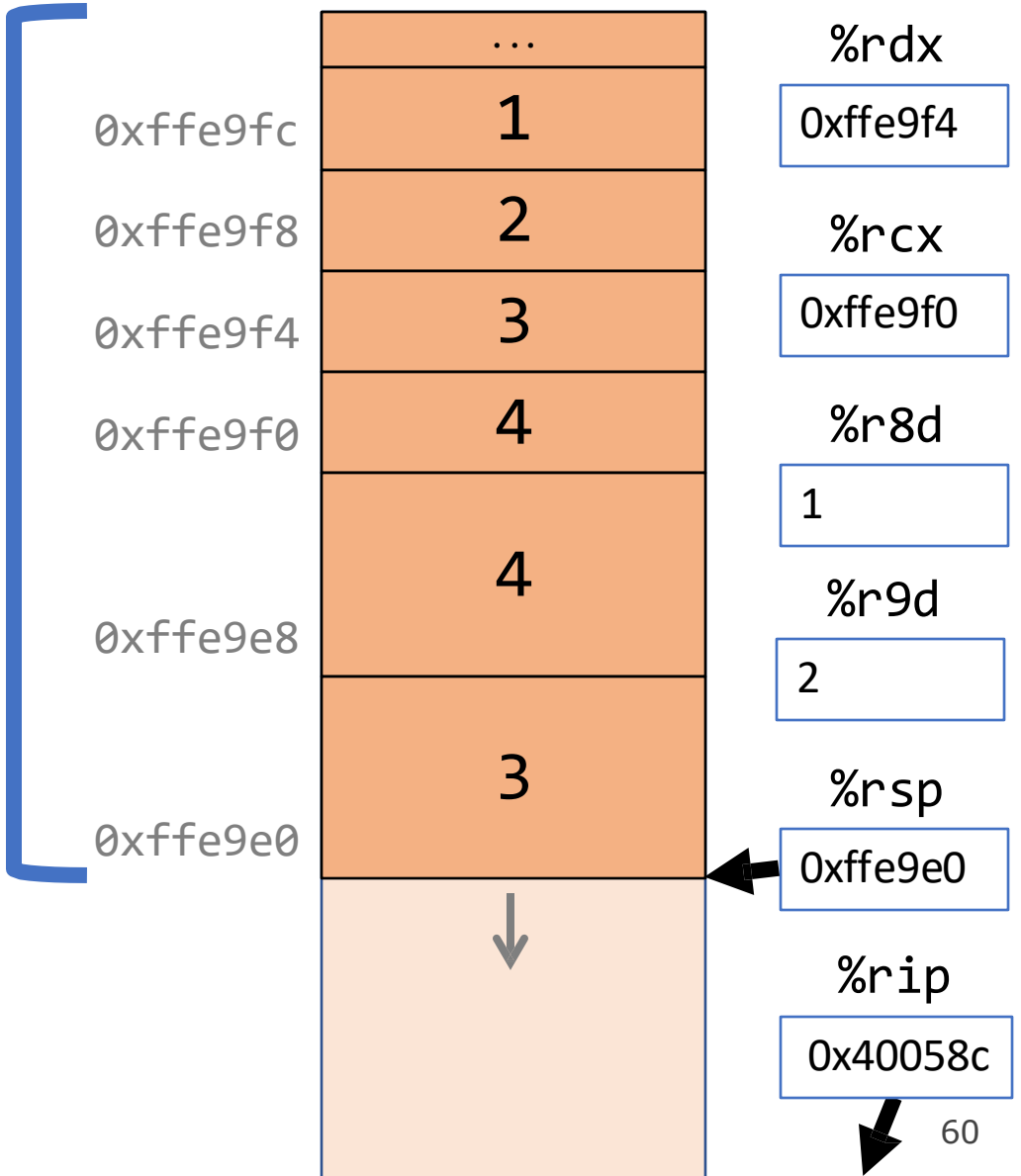
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40057c <+45>:    mov    $0x1,%r8d
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
```

main()

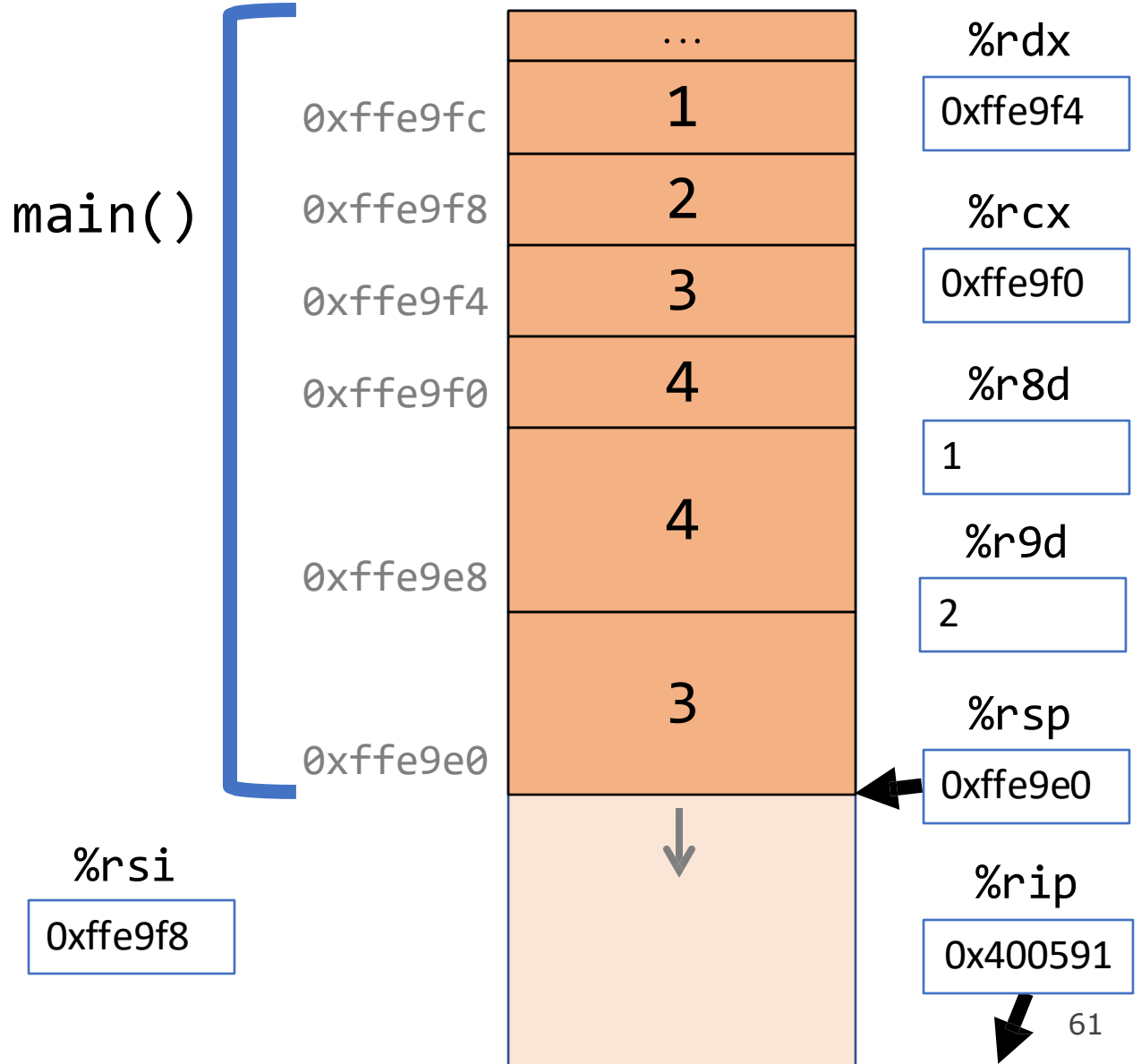


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x400546 <func>
```

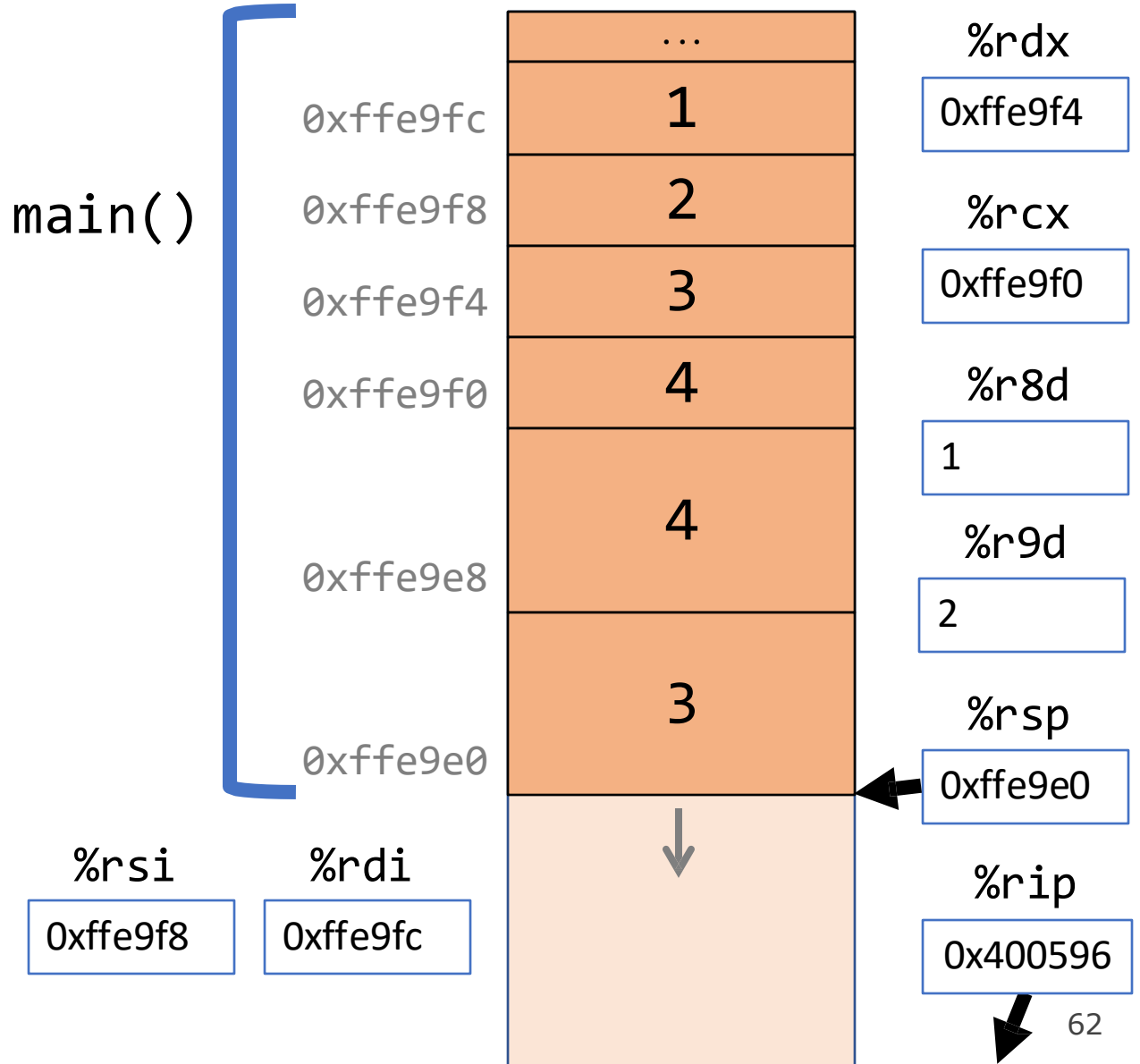


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x400546 <func>
0x40059b <+76>:    add    $0x10,%rsp
```

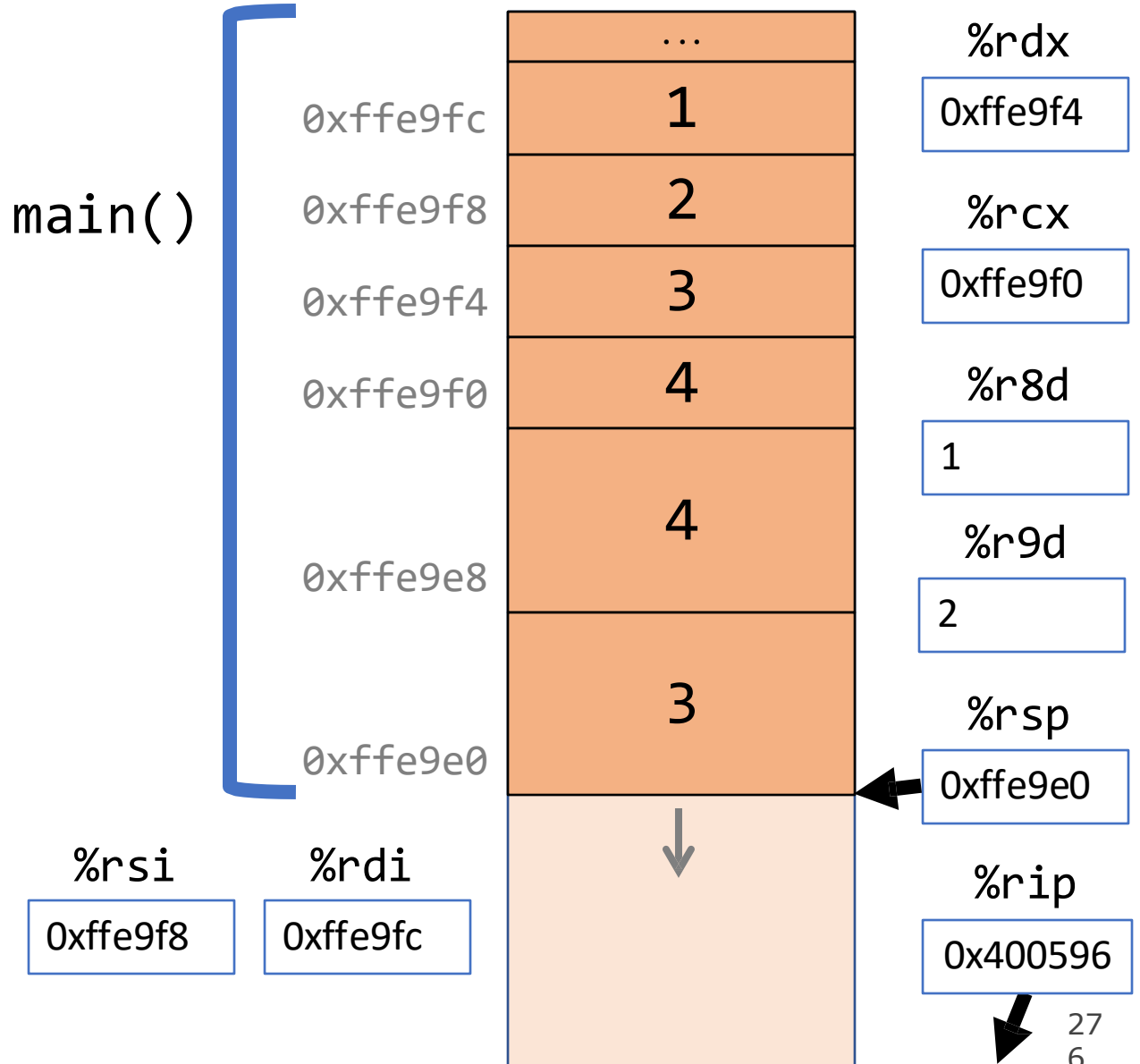


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x400546 <func>
0x40059b <+76>:    add    $0x10,%rsp
…
```

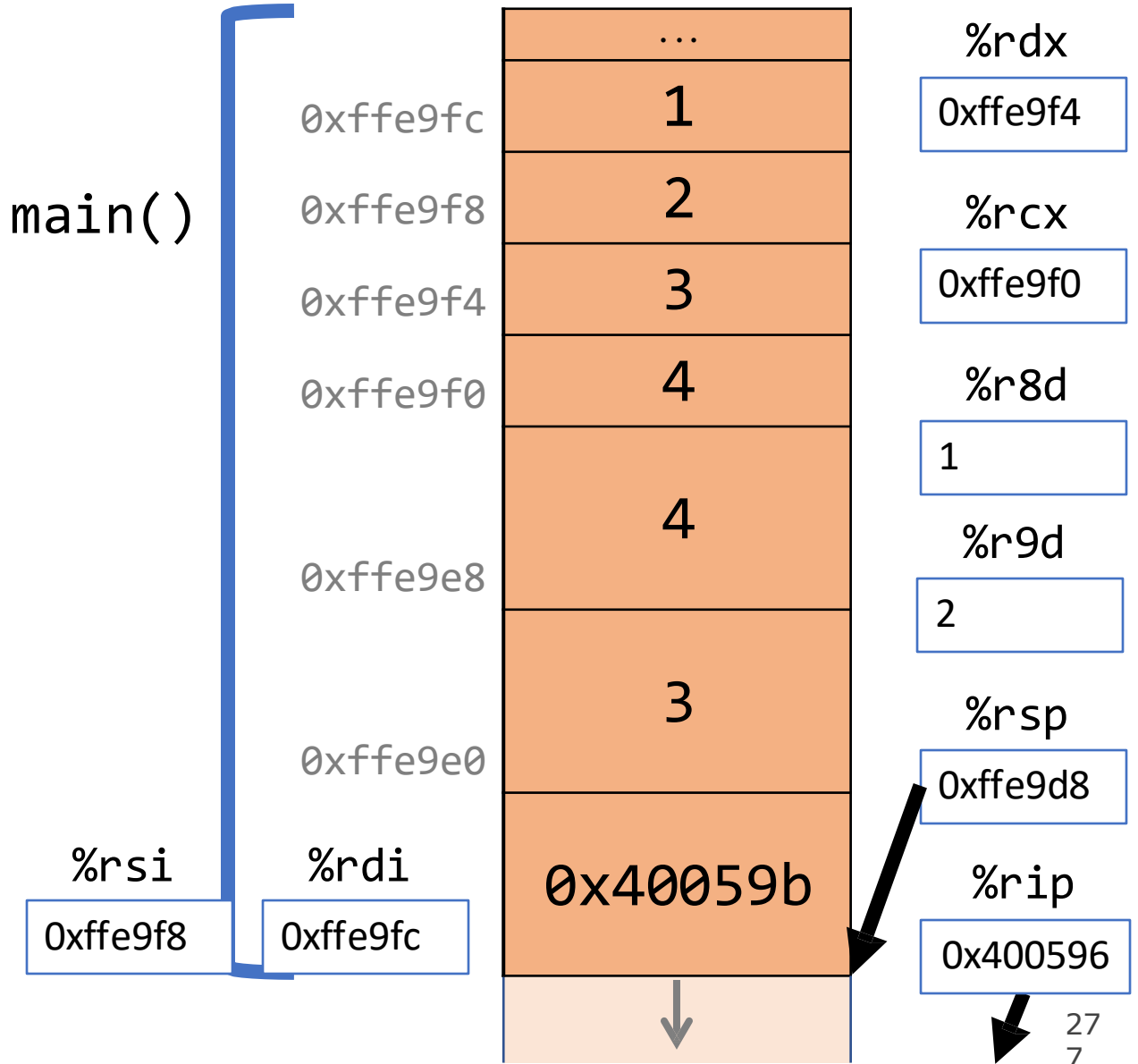


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x400546 <func>
0x40059b <+76>:    add    $0x10,%rsp
…
```



Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

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.

Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Local Storage

```
long caller() {
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    ...
}
```

```
caller:
    sub    $0x10, %rsp           // 16 bytes for stack frame
    movq   $0x216, 0x8(%rsp)     // store 534 in arg1
    movq   $0x421, (%rsp)        // store 1057 in arg2
    mov    %rsp, %rsi            // compute &arg2 as second arg
    lea    0x8(%rsp), %rdi       // compute &arg1 as first arg
    callq swap_add               // call swap_add(&arg1, &arg2)
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

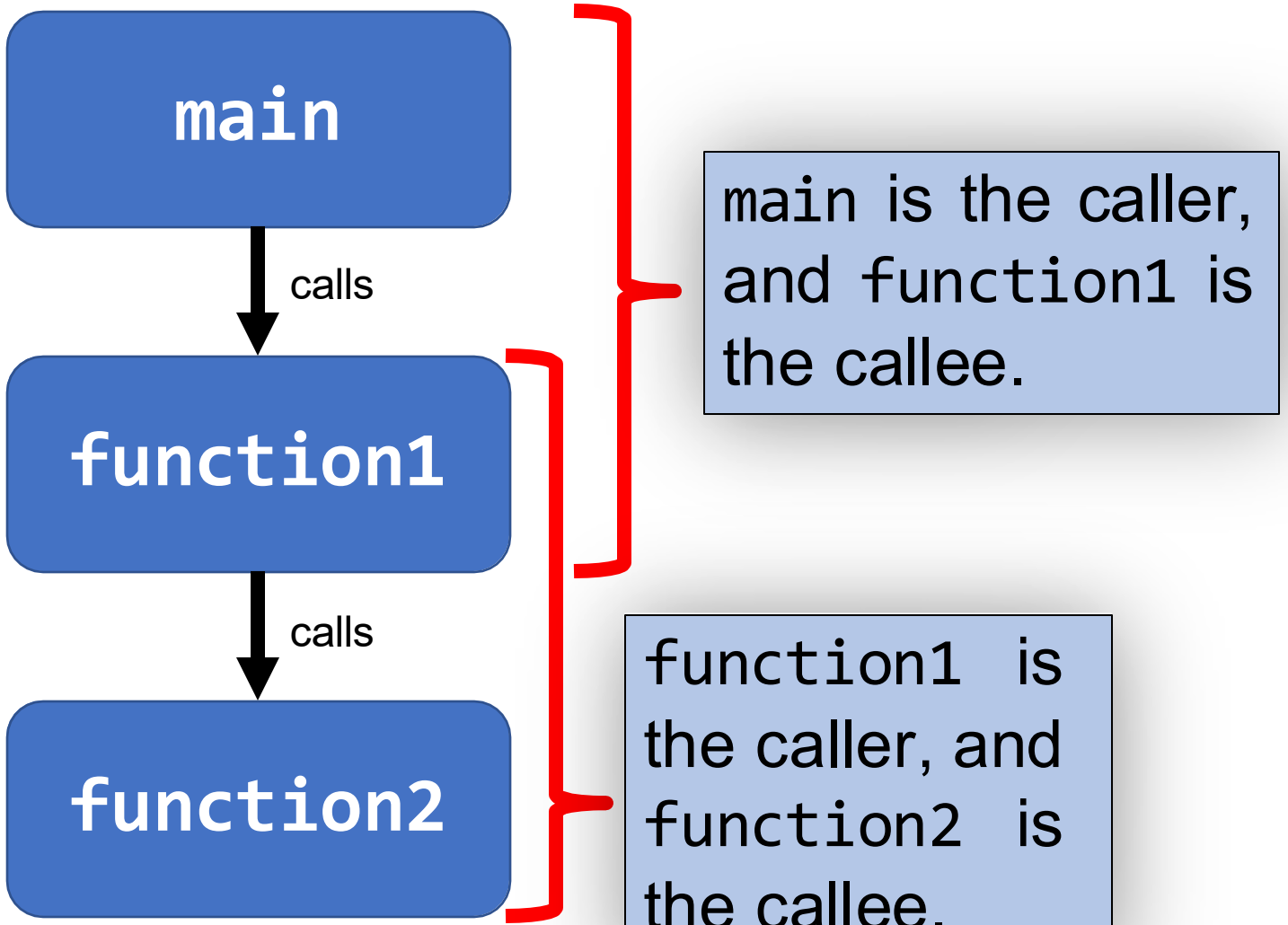
Register Restrictions

There is only one copy of registers for all programs and functions.

- **Problem:** what if ***funcA*** is building up a value in register %r10, and calls ***funcB*** in the middle, which also has instructions that modify %r10? ***funcA***'s value will be overwritten!
- **Solution:** make some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



Register Restrictions

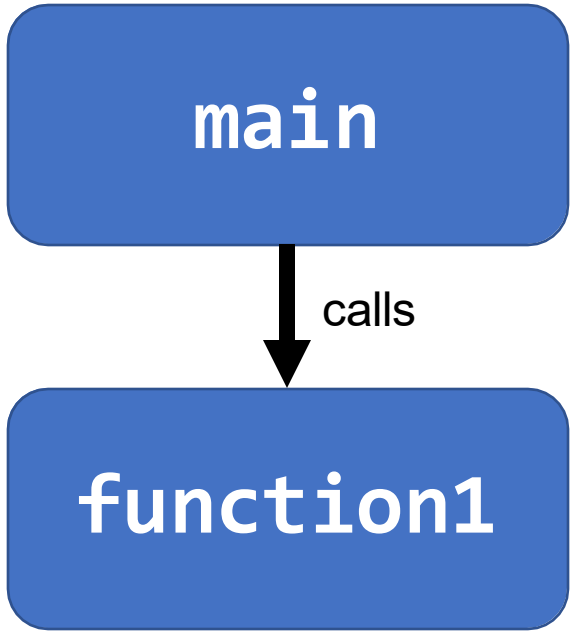
Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

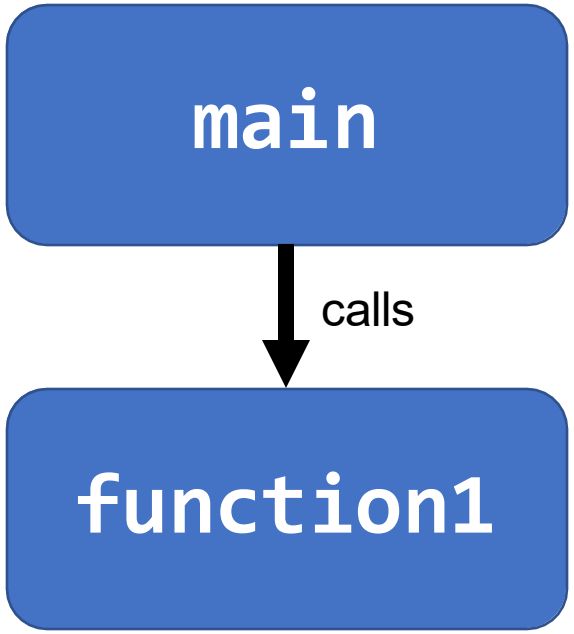
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

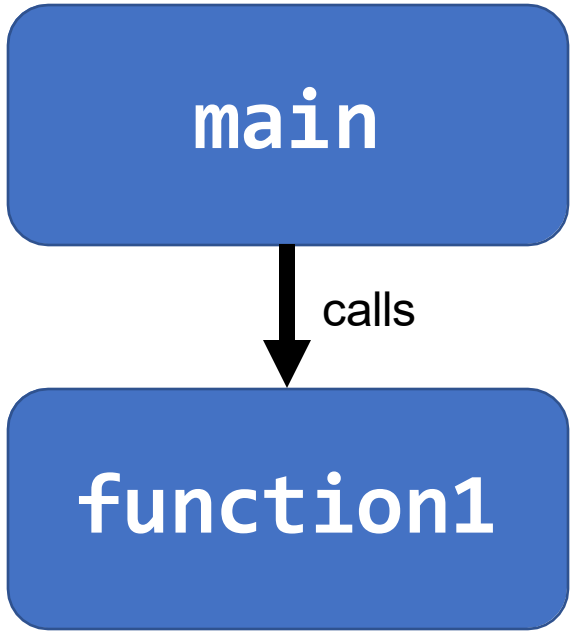
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers




```
function1:
   push %rbp
   push %rbx
   ...
   pop %rbx
   pop %rbp
   retq
```

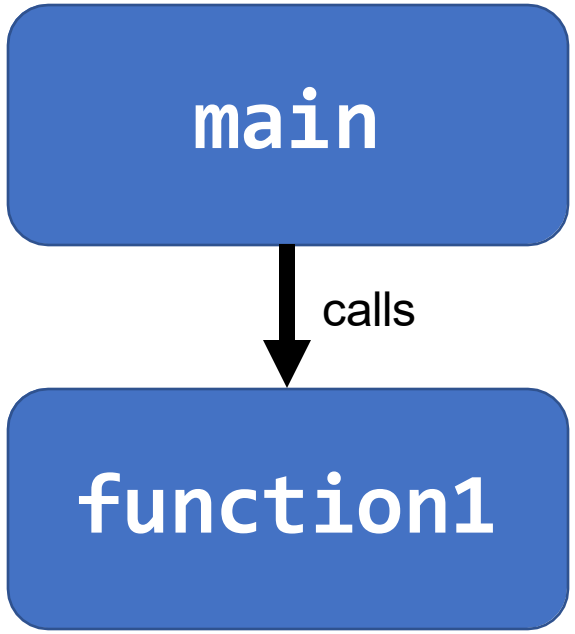
Callee-Owned Registers



`main` can use callee-owned registers but calling `function1` may permanently modify their values.

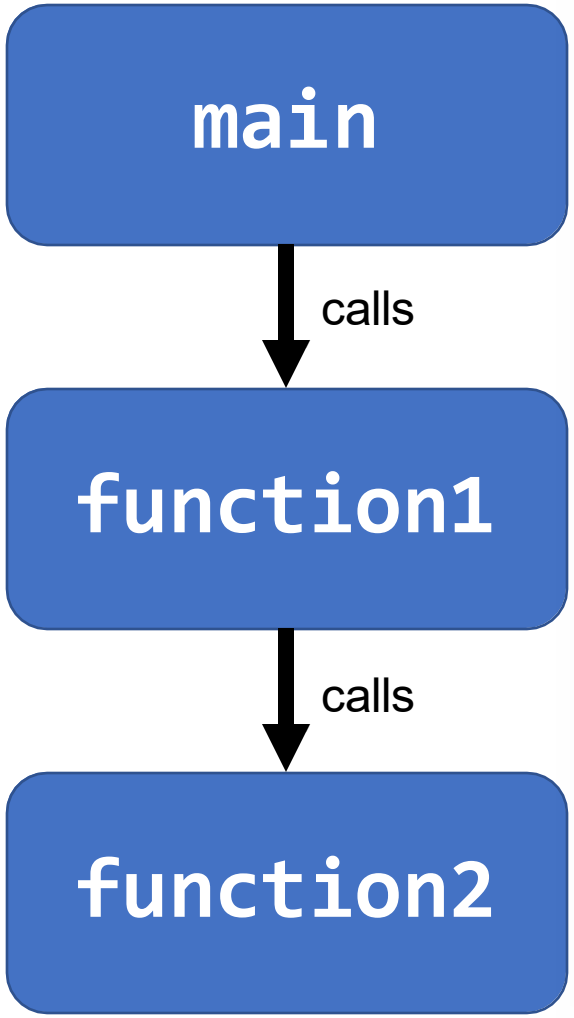
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers




```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

A Day In the Life of `function1`



Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Example: Recursion

- Let's look at an example of recursion at the assembly level.
- We'll use everything we've learned about registers, the stack, function calls, parameters, and assembly instructions!
- We'll also see how helpful GDB can be when tracing through assembly.

`factorial.c` and `factorial`

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're done with all our assembly lectures! Now we can fully understand what's going on in the assembly below, including how someone would call sum_array in assembly and what the **ret** instruction does.

```
0000000000401136 <sum_array>:
  401136 <+0>:   mov    $0x0,%eax
  40113b <+5>:   mov    $0x0,%edx
  401140 <+10>:  cmp    %esi,%eax
  401142 <+12>:  jge    0x40114f <sum_array+25>
  401144 <+14>:  movslq %eax,%rcx
  401147 <+17>:  add    (%rdi,%rcx,4),%edx
  40114a <+20>:  add    $0x1,%eax
  40114d <+23>:  jmp    0x401140 <sum_array+10>
  40114f <+25>:  mov    %edx,%eax
  401151 <+27>:  retq
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Optimizations you'll see

`nop`

- **`nop/nopl`** are "no-op" instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- "Sometimes, doing nothing is how to be most productive" – Philosopher Nick

`mov %ebx,%ebx`

- Zeros out the top 32 register bits (because a `mov` on an e-register zeros out rest of 64 bits).

`xor %ebx,%ebx`

- Optimizes for performance as well as code size (read more here):

```
b8 00 00 00 00          mov $0x0,%eax
31 c0                   xor %eax,%eax
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Possible Alternative

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

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

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

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

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

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

Possible Alternative

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

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)

Optimizing Instruction Counts

- Both versions have the same **static** **instruction count** (# of written instructions).
- But they have different **dynamic** **instruction counts** (# of executed instructions when program is run).
  - If n = 0, left (GCC common output) is best b/c fewer instructions
  - If n is large, right (alternative) is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

Optimizations

- **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.
- Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.

Recap

- Revisiting %rip
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example
- Optimizations

That's it for assembly! **Next time:** managing the heap

Key GDB Tips For Assembly

- Examine 4 giant words (8 bytes) on the stack:

```
(gdb) x/4g $rsp
0x7fffffffe870: 0x0000000000000005      0x0000000000400559
0x7fffffffe880: 0x0000000000000000      0x0000000000400575
```

- display/undisplay (prints out things every time you step/next)

```
(gdb) display/4w $rsp
1: x/4xw $rsp
0x7fffffffe8a8:
0xf7a2d830      0x00007fff      0x00000000      0x00000000
```

Key GDB Tips For Assembly

- `stepi`/`finish`: step into current function call/return to caller:

```
(gdb) finish
```

- Set register values during the run

```
(gdb) p $rdi = $rdi + 1
```

(Might be useful to write down the original value of $rdi somewhere)

- Tui things
  - `refresh`
  - `focus cmd` – use up/down arrows on gdb command line (vs `focus asm`, `focus regs`)
  - `layout regs`, `layout asm`

Extra Practice – Escape Room 2

https://godbolt.org/z/8e31fG4r5

`escape_room`

Escape room assembly code

```
000000000000115b <escape_room>:
    115b:  48 83 ec 08           sub    $0x8,%rsp
    115f:  ba 0a 00 00 00        mov    $0xa,%edx
    1164:  be 00 00 00 00        mov    $0x0,%esi
    1169:  e8 d2 fe ff ff        callq  1040 <strtol@plt>
    116e:  48 89 c7              mov    %rax,%rdi
    1171:  e8 d3 ff ff ff        callq  1149 <transform>
    1176:  a8 01                 test   $0x1,%al
    1178:  74 0a                 je     1184 <escape_room+0x29>
    117a:  b8 00 00 00 00        mov    $0x0,%eax
    117f:  48 83 c4 08           add    $0x8,%rsp
    1183:  c3                    retq
    1184:  b8 01 00 00 00        mov    $0x1,%eax
    1189:  eb f4                 jmp    117f <escape_room+0x24>
```

Escape room assembly code

```
0000000000001149 <transform>:
    1149:  8d 04 bd 00 00 00 00  lea    0x0(,%rdi,4),%eax
    1150:  8d 50 01              lea    0x1(%rax),%edx
    1153:  83 fa 32              cmp    $0x32,%edx
    1156:  7f 02                 jg     115a <transform+0x11>
    1158:  89 d0                 mov    %edx,%eax
    115a:  c3                    retq
```

Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are *very* good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type *T* and integer constant *N:*

T `A[`*N*`]`

- The starting location is designated as x_A
- The declaration allocates *N* `* sizeof(`*T*`)` bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of x_A.

Array Allocation and Access

- Example:

```
char      A[12];
char     *B[8];
int       C[6];
double   *D[5]
```

Array	Element Size	Total Size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access `C[3]` above. If the address of C is in register `%rdx`, and `3` is in register `%rcx`
- The following copies `C[3]` into `%eax`,

```
movl (%rdx,%rcx,4), %eax
```

Pointer

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference `A[i]` is identical to `*(A+i)`
- Example: if the address of array E is in `%rdx`, and the integer index, `i`, is in `%rcx`, the following are some expressions involving E:

Expression	Type	Value	Code
`E[0]`	int	M[x_E]	`movl (%rdx), %eax`
`E[i]`	int	M[x_E+4i]	`movl (%rdx,%rcx,4) %eax`
`&E[2]`	int *	x_E+8	`leaq 8(%rdx), %rax`
`E+i-1`	int *	x_E+4i-4	`leaq -4(%rdx,%rcx,4), %rax`
`*(E+i-3)`	int	M[x_E+4i-12]	`movl -12(%rdx,%rcx,4) %eax`
`&E[i]-E`	long	i	`movq %rcx,%rax`

`movq %rdx, %rax`

Pointer

- Practice: x_S is the address of a `short` integer array, S, stored in `%rdx`, and a long integer index, `i`, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

Expression	Type	Value	Assembly Code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

Pointer

- Practice: x_S is the address of a `short` integer array, S, stored in `%rdx`, and a long integer index, `i`, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

Expression	Type	Value	Assembly Code
`S+1`	`short *`	$x_S + 2$	`leaq 2(%rdx),%rax`
`S[3]`	`short`	$M[x_S + 6]$	`movw 6(%rdx),%ax`
`&S[i]`	`short *`	$x_S + 2i$	`leaq (%rdx,%rcx,2),%rax`
`S[4*i+1]`	`short`	$M[x_S + 8i + 2]$	`movw 2(%rdx,%rcx,8),%ax`
`S+i-5`	`short *`	$x_S + 2i - 10$	`leaq -10(%rdx,%rcx,2),%rax`

Structur

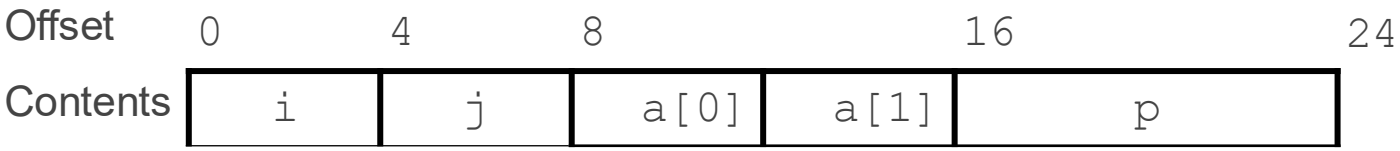
- The C `struct` declaration is used to group objects of different types into a single unit.
- Each "field" is referenced by a name, and can be accessed using dot (`.`) or (if there is a pointer to the struct) arrow (`->`) notation.
- Structures are kept in contiguous memory
- A pointer to a struct is to its first byte, and the compiler maintains the byte offset information for each field.
- In assembly, the references to the fields are via the byte offsets.

Structur

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:



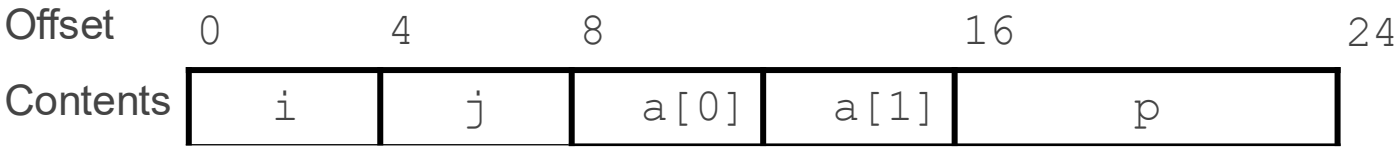
- The numbers along the top of the diagram are the byte offsets of the fields from the beginning of the structure.
- Note that the array is embedded in the structure.
- To access the fields, the compiler generates code that adds the field offset to the address of the structure.

Structur

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:



- Example: Variable `r` of type struct `rec *` is in register `%rdi`. The following copies element `r->i` to element `r->j`:

```
movl (%rdi), %eax    // get r->i
movl %eax, 4(%rdi)   // store in r->j
```

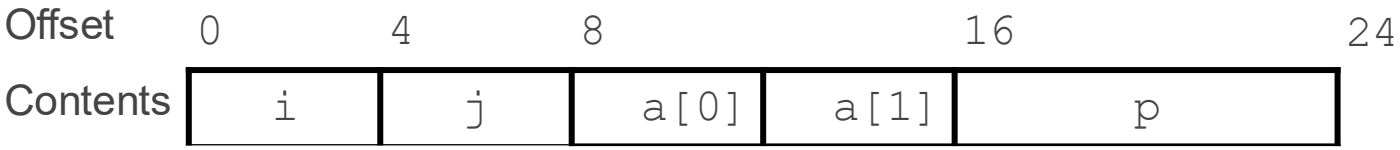
- The offset of `i` is `0`, so `i`'s field is `%rdi`. The offset of `j` is `4`, so the offset of `4` is added to the address of `%rdi` to store into `j`.

Structur

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:



- We can generate a pointer to a field by adding the field's offset to the struct address. To generate `&(r->a[1])` we add offset `8 + 4 = 12`. For a pointer `r` in register `%rdi` and long `int` variable `i` in `%rsi`, we can generate the pointer value `&(r->a[i])` with one instruction:

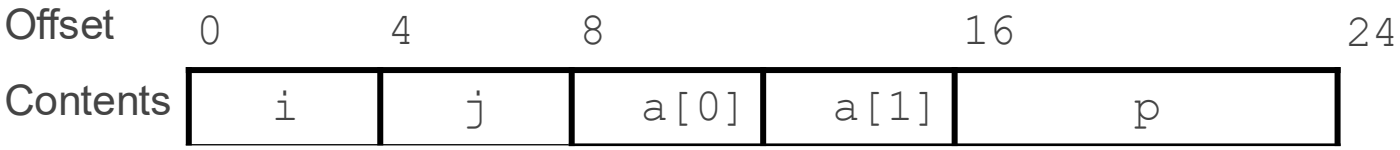
```
leaq 8(%rdi,%rsi,4), %rax // set %rax to &r->a[i]
```

Structur

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:



- The following code implements `r->p = &r->a[r->i + r->j];`

```
// r in %rdi

movl  4(%rdi),%eax          //  get r->j

addl  (%rdi),%eax           //  add r->i

leaq 8(%rdi,%rax,4), %rax // compute %rax as &r->a[r->i + r->j]
cltq                        // extend %eax to 8 bytes, %rax = r->j
movq %rax, 16(%rdi)         // store in r->p
```

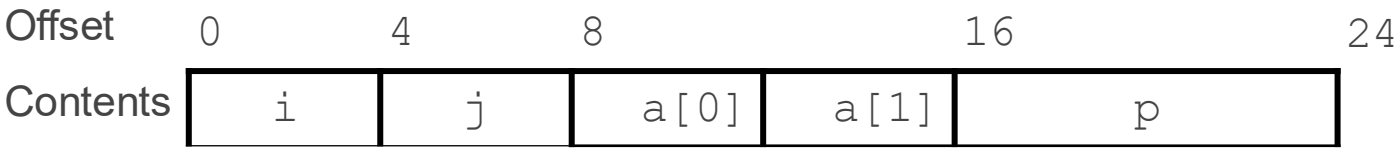
x

Structur

- Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:



- Notice that all struct manipulation is handled at compile time, and the machine code doesn't contain any information about the field declarations or the names of the fields.
- The compiler does all the work, keeping track as it produces the assembly code.
- BTW, if you're curious about how the compiler actually does the transformation from C to assembly, take a compilers class, e.g., CS143.

Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value *K* (normally 2, 4, or 8).
- These *alignment restrictions* simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

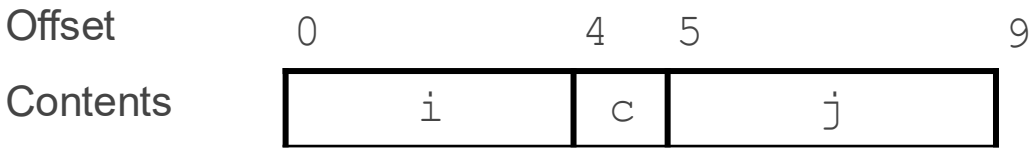
K	Types
1	`char`
2	`short`
4	`int, float`
8	`long, double, char *`

Data Alignment

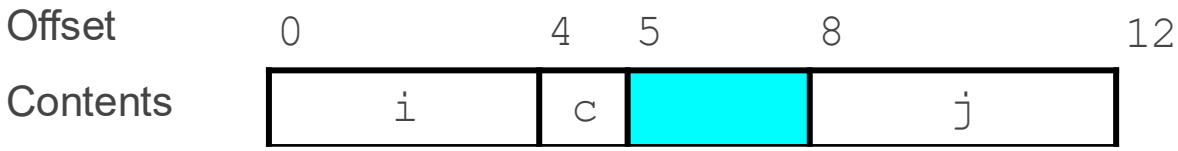
- The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
- For example, let's look at the following struct:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

- If the compiler used a minimal allocation:



- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:



- So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!

Function Pointers

- Let's look at the following code:

```
void *gfind_max(void *arr, int n, size_t elemsz,
                int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

Function Pointers

- Let's look at the following code:

```
void *gfind_max(void *arr, int n, size_t elemsz,
                int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

- Because `compar` is a function pointer, the compiler calls the function via the address that is in the compar variable.
- Let's take a look at this in gdb.

References and

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64