CS 31: Introduction to Computer Systems

12-13: Assembly Arithmetic and Control

03-04-2025 - 03-05-2025

SWARTHMORE COLLEGE

Announcements

- New HW Groups Posted!

- Clicker scores up on Github.

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

Check your frequency:

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:

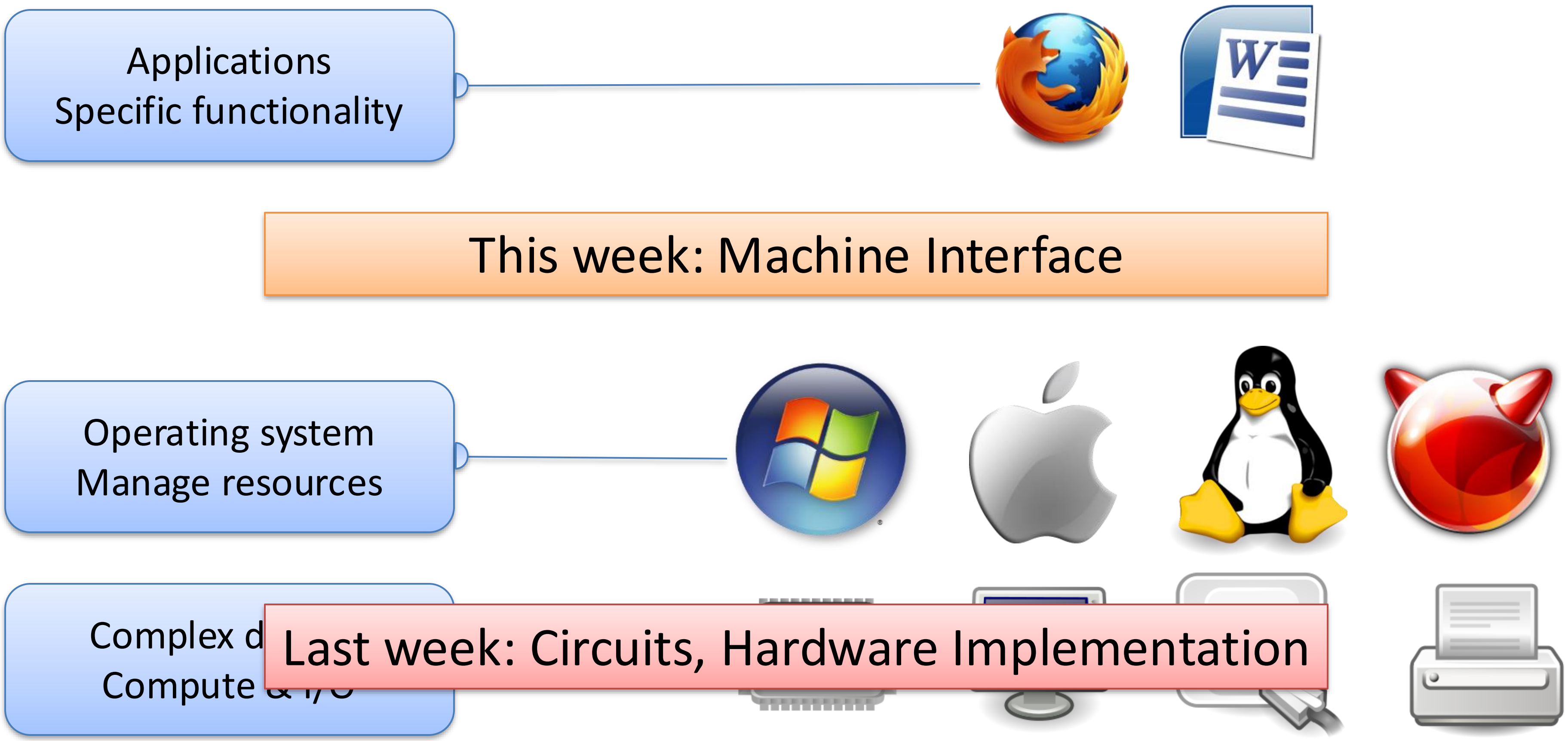
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success

What we will learn this week

1. Instruction set architecture (ISA)
    - Interface between programmer and CPU
    - Accessing Memory and Registers
    - Arithmetic Instructions
    - Control Flow

- 2. Functions & the stack
    - Stack data structure, applied to memory
    - Behavior of function calls
    - Storage of function data, at assembly level

Abstraction



Applications
Specific functionality

This week: Machine Interface

Operating system
Manage resources

Complex d
Compute & I/O

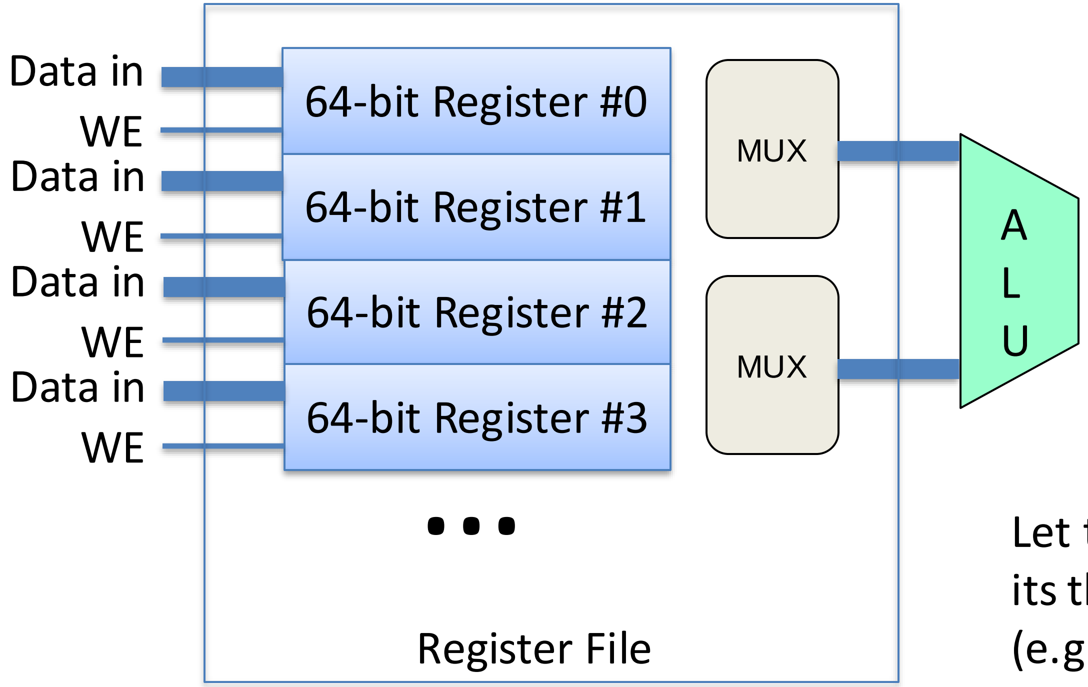
Last week: Circuits, Hardware Implementation

Hardware: Control, Storage, ALU circuitry

Program Counter (PC): **Address 0**

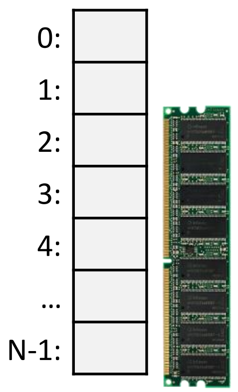
Instruction Register (IR): OP Code | Reg A | Reg B | Result

- acts on instruction bits to execute individual instructions
- PC value used to determine next instruction to execute

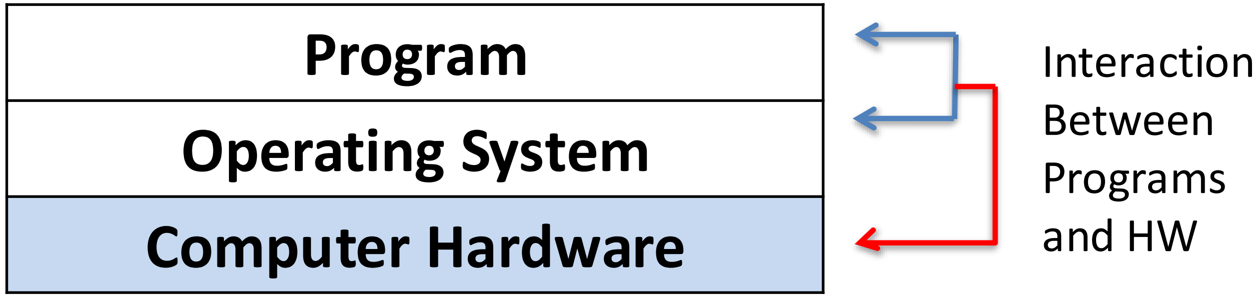


Let the ALU do its thing. (e.g., Add)

(Memory)



How a computer runs a program:



Program

Operating System

Computer Hardware

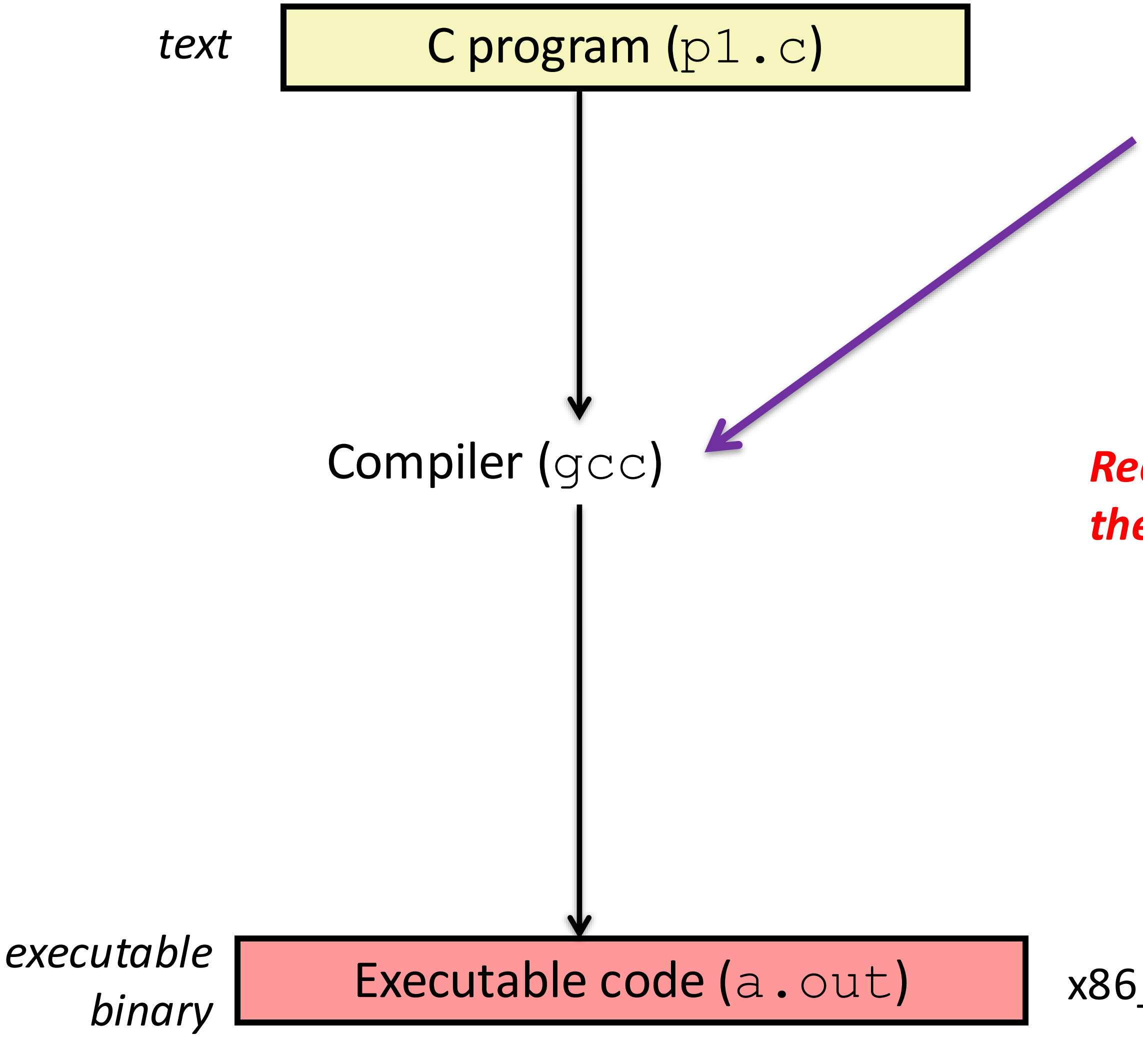
Interaction
Between
Programs
and HW

Instruction Set Architecture (ISA) Defines:

1. **Set and Encoding of Instructions**: defines a set of instructions and specifies their machine code format
2. **Processor State**: memory, registers, flags
   - makes CPU resources (registers, flags) available to the programmer
   - Allows instructions to access main memory (potentially with limitations)
3. **State Machine**: transitions from one processor state to another as a result of instruction execution
   - E.g., executing: `ADD %r1 %r2` (ADD source destination)

     state change:
     - `%r2 -> %r2+%r1`
     - `ALU flags: Overflow Flag (signed overflow)?`
       `Carry Flag (unsigned overflow)?    Zero Flag?`
     - `PC ← address of next instruction`

ISA and the Compiler



text

C program (`p1.c`)

Compiler (`gcc`)

executable binary

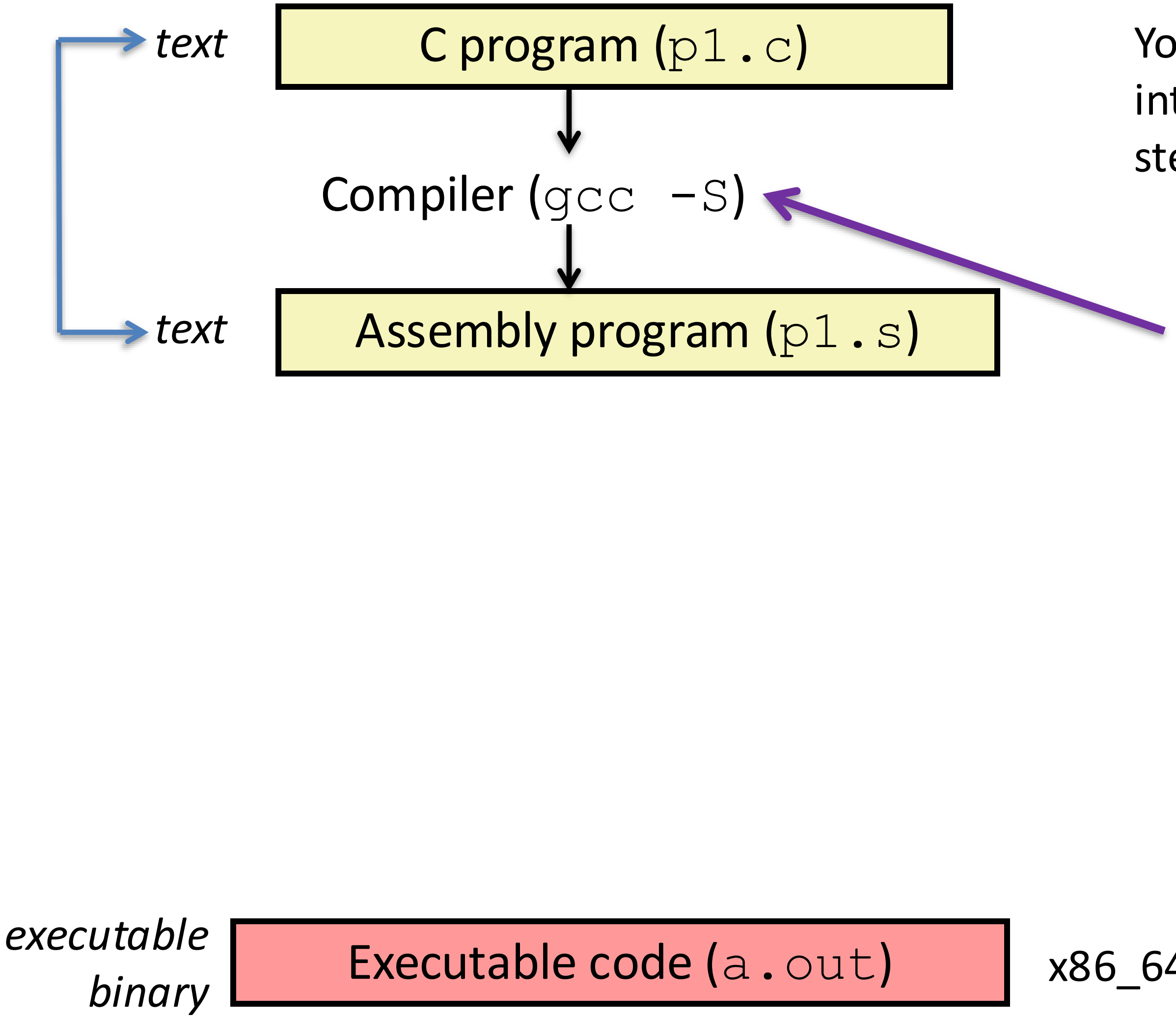
Executable code (`a.out`)

Usually compile to a.out in a single step: gcc p1.c

Reality is more complex: there are intermediate steps!

x86_64 machine code instructions

ISA and the Compiler



text C program (`p1.c`)

Compiler (`gcc -S`)

text Assembly program (`p1.s`)

executable binary Executable code (`a.out`)

You can see the results of intermediate compilation steps using different gcc flags

ISA is Interface between CPU and Compiler: Compiler translates program source code to machine code of a target ISA
e.g., C program → gcc → ISA machine code (0's and 1's)

x86_64 machine code instructions

Assembly Code

Human-readable form of CPU instructions

- Almost a 1-to-1 mapping to hardware instructions (Machine Code)
- Hides some details:
  - Registers have names rather than numbers
  - Instructions have names rather than variable-size codes

We're going to use x86_64 Assembly

- Can compile C to x86_64 Assembly on our system:

```
gcc -S code.c          # open code.s in an editor to view
```

C statement: A = A*B

Simple instructions:

```
LOAD A, R1
LOAD B, R2
PROD R1, R2
STORE R2, A
```

Powerful instructions:

```
MULT B, A
```

Translation:
Load the values ‘A’ and ‘B’ from memory into registers (R1 and R2) , compute the product, store the result in memory where ‘A’ was.

Instruction Set Architecture (ISA)

- Above ISA: High-level language (C, Python, …)
  - Hides ISA from users
  - Allows a program to run on any machine (after translation by human and/or compiler)

- **ISA**: Interface between CPU and high-level language/compiler
  - Compiler translates program source code to machine code of a target ISA

    e.g., C program → gcc → ISA machine code (0’s and 1’s)

- Below ISA: Hardware implementing ISA can change (faster, smaller, …)
  - ISA is like a CPU “family”

RISC versus CISC (Historically)

- Complex Instruction Set Computing (**CISC**)
  - Large, rich instruction set
  - More complicated instructions built into hardware
  - Multiple clock cycles per instruction
  - Easier for humans to reason about

- Reduced Instruction Set Computing (**RISC**)
  - Small, highly optimized set of instructions
  - Memory accesses are specific instructions
  - One instruction per clock cycle
  - Compiler: more work, more potential optimization

So . . . Which System “Won”?

- Most ISAs (after mid/late 1980’s) are RISC
- The ubiquitous Intel x86 is CISC; while ARM is RISC
  - Tablets and smartphones (ARM) taking over
- x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions
- Distinction between RISC and CISC is less clear
  - Some RISC instruction sets have more instructions than some CISC sets

Intel’s Woes with ARM Foe

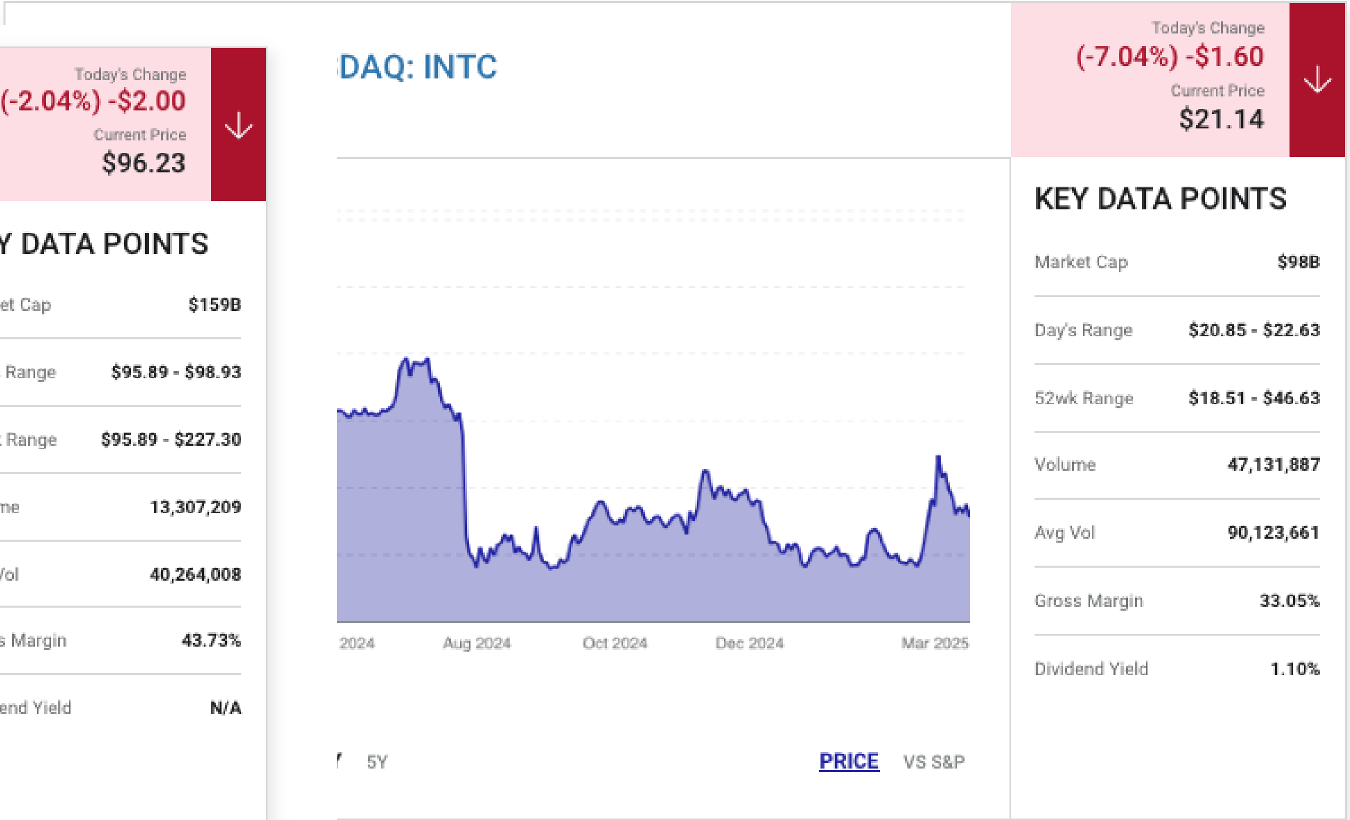
NASDAQ: AMD
Advanced Micro Devices

Today's Change
(-2.04%) -$2.00
Current Price
$96.23

KEY DATA POINTS

Market Cap	$159B
Day's Range	$95.89 - $98.93
52wk Range	$95.89 - $227.30
Volume	13,307,209
Avg Vol	40,264,008
Gross Margin	43.73%
Dividend Yield	N/A

AMD has struggled against Nvidia, largely due to its inferior software. In a recent study, SemiAnalysis called AMD's out-of-the-box GPUs "unusable" for AI training, noting it needed "multiple teams of AMD engineers" to help it fix software bugs. However, AMD has been able to carve out a niche in AI inference, with SemiAnalysis saying its customers typically use AMD's GPUs for narrow, well-defined inference use cases.



DAQ: INTC

Today's Change
(-7.04%) -$1.60
Current Price
$21.14

KEY DATA POINTS

Market Cap	$98B
Day's Range	$20.85 - $22.63
52wk Range	$18.51 - $46.63
Volume	47,131,887
Avg Vol	90,123,661
Gross Margin	33.05%
Dividend Yield	1.10%

its revenue decline last quarter by 6% to $13.3 billion, and its adjusted EPS flip to
ofit of $0.41 a year ago. The one bright spot last quarter was its data center and AI
e rise 9% to $3.3 billion. However, when compared to Nvidia and AMD, that is a very
t.

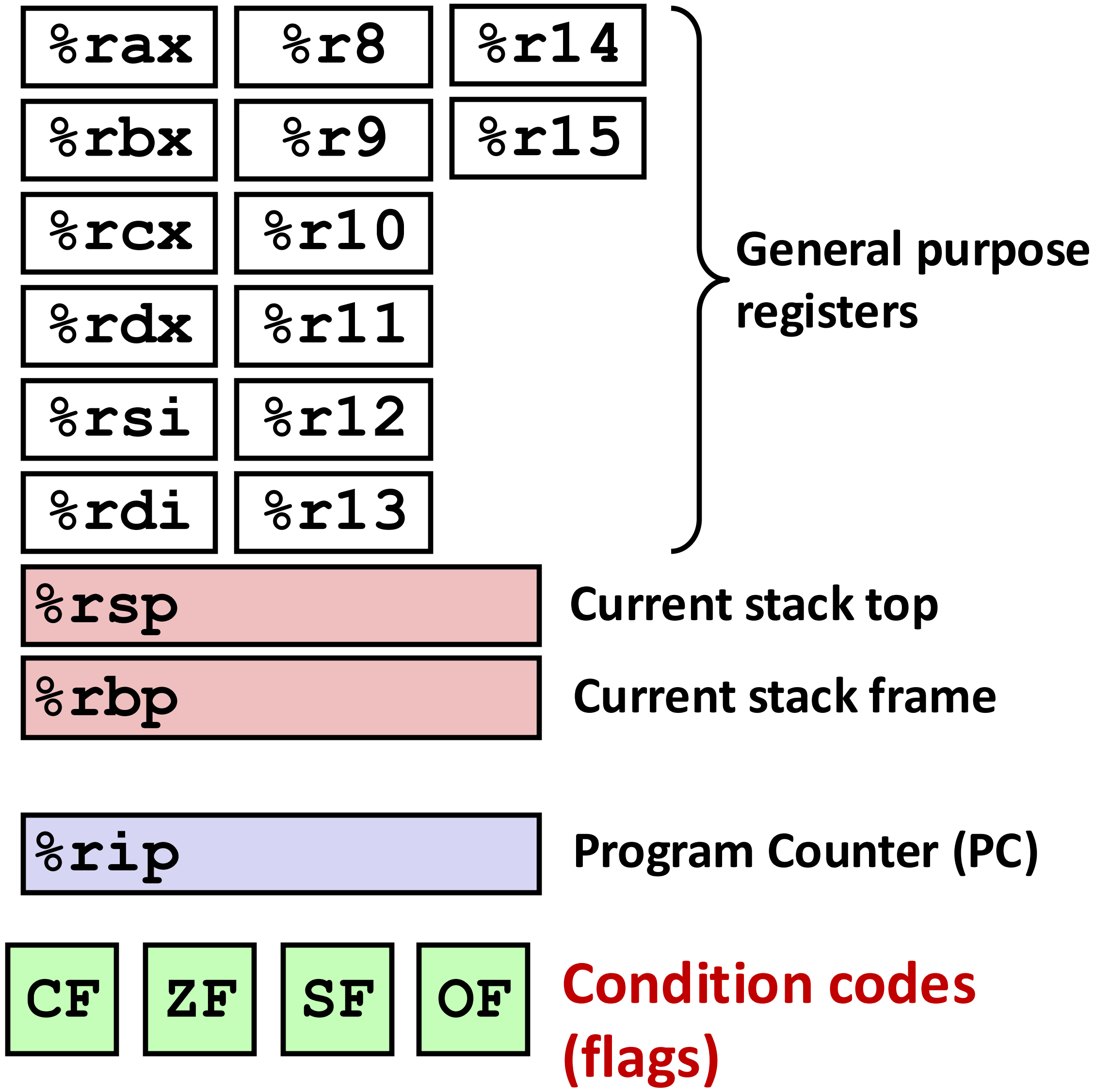
Instruction Set Architecture (ISA) Defines:

1. **Set and Encoding of Instructions**: defines a set of instructions and specifies their machine code format
2. **Processor State**: memory, registers, flags
   - makes CPU resources (registers, flags) available to the programmer
   - Allows instructions to access main memory (potentially with limitations)
3. **State Machine**: transitions from one processor state to another as a result of instruction execution
   - E.g., executing: `ADD %r1 %r2` (ADD source destination)

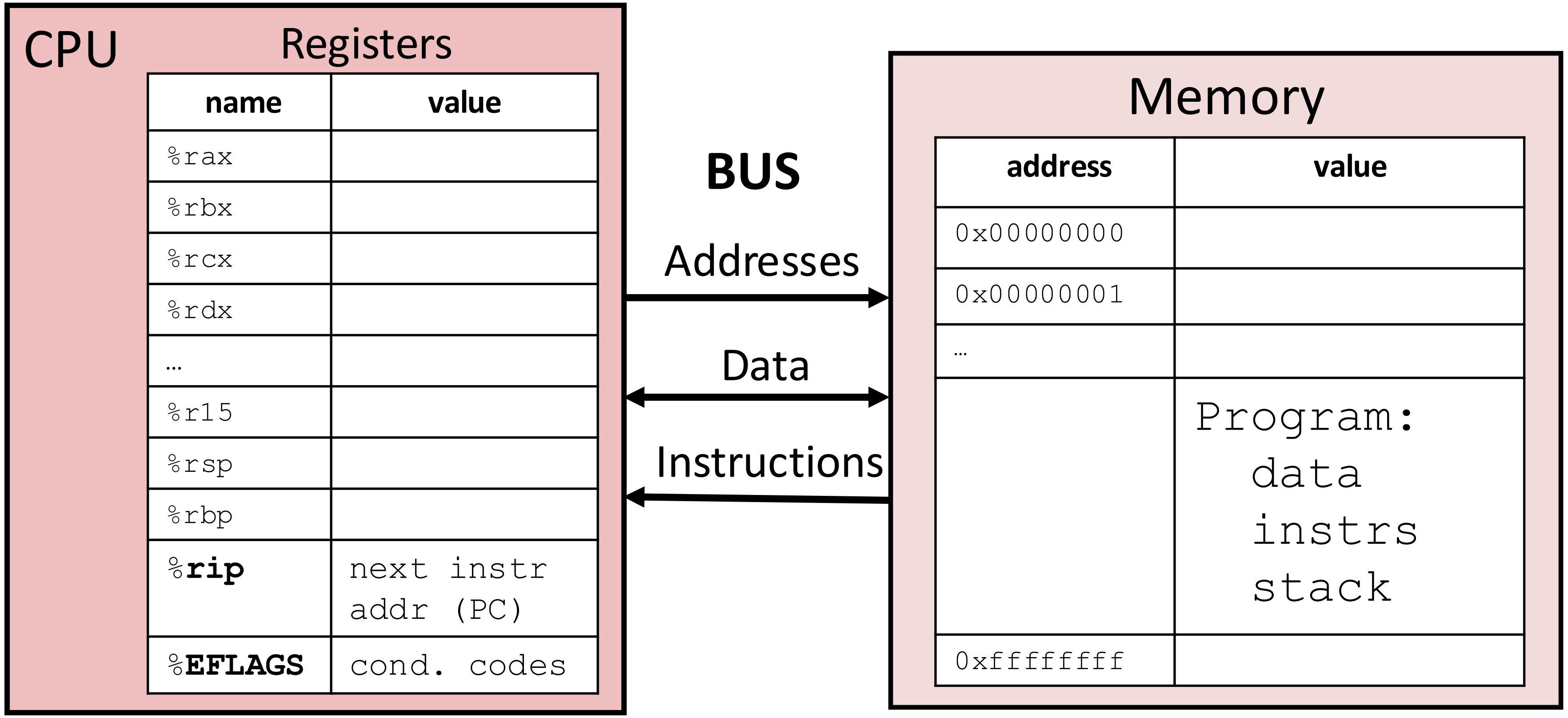
     state change:
     - `%r2 -> %r2+%r1`
     - `ALU flags: Overflow Flag (signed overflow)?`
       `Carry Flag (unsigned overflow)?    Zero Flag?`
     - `PC ← address of next instruction`

Processor State in Registers

- Working memory for currently executing program
  - 14 for temporary data (%rax - %r15)
  - 2 for location of runtime stack (%rbp, %rsp)
  - 1 for address of next instruction to execute (%rip)
  - 1 for status of recent ALU tests (CF, ZF, SF, OF)



Assembly Programmer’s View of State



CPU

Registers

name	value
`%rax`	
`%rbx`	
`%rcx`	
`%rdx`	
…	
`%r15`	
`%rsp`	
`%rbp`	
`%rip`	`next instr addr (PC)`
`%EFLAGS`	`cond. codes`

BUS

Addresses

Data

Instructions

Memory

address	value
`0x00000000`	
`0x00000001`	
…	

	`Program:`
	`data`
	`instrs`
	`stack`

`0xffffffff`	

Registers:

PC: Program counter (%rip)

Condition codes (%EFLAGS)

General Purpose (%rax - %r15)

Memory:

- Byte addressable array
- Program code and data
- Execution stack

Four Types of Assembly Instructions

1. Arithmetic: use ALU to compute a value
   - a + b a << 2 a | b …

2. Data Movement: load and store
   - move data/instructions between registers and memory
   - x = y + z

3. Control Flow: branch, jump, etc.
   - Change PC based on ALU condition code state

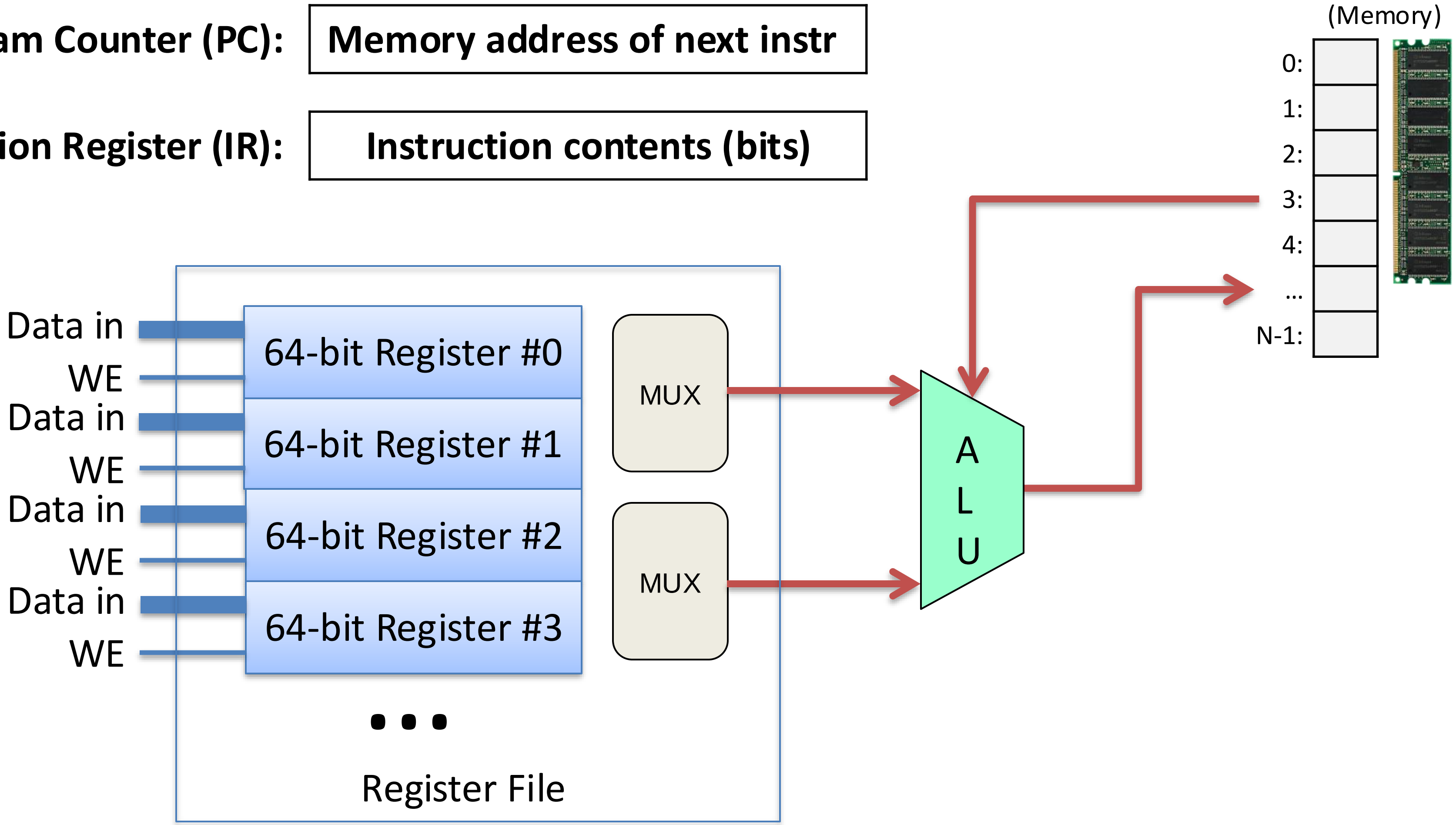
4. Stack Instructions: push and pop stack frames

Arithmetic

Use ALU to **compute** a value, **store** result in register / memory.

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**



Four Types of Assembly Instructions

1. Arithmetic: use ALU to compute a value
2. Data Movement: load and store
    - move data/instructions between registers and memory
    - x = y + z

    - Examples: `mov, movl, movq`

    - **Load**: move data from memory to register
    - **Store**: move data from register to memory

The suffix letters specify how many bytes to move (not always necessary, depending on context).

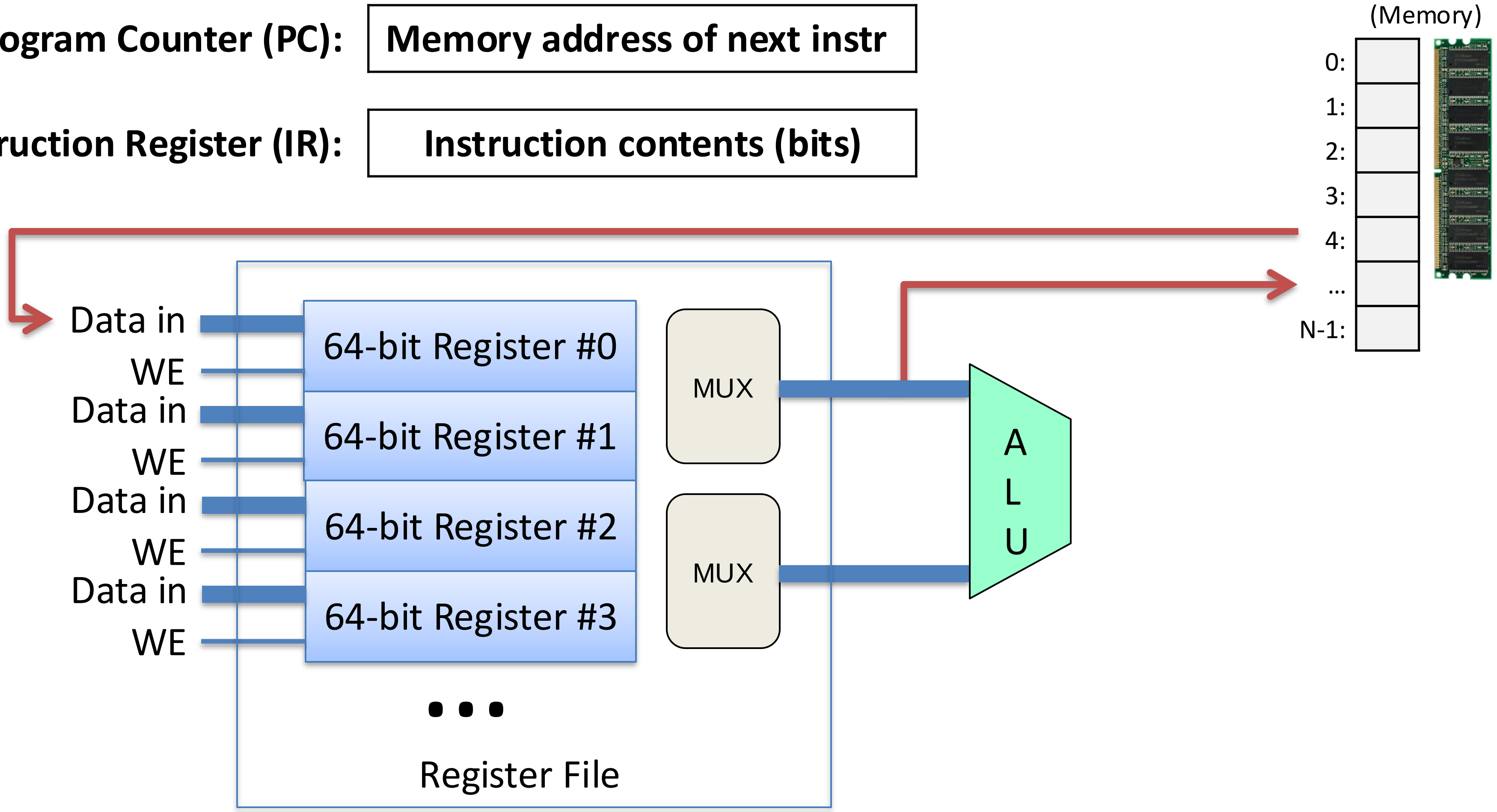
l -> 32 bits
q -> 64 bits

Data Movement

Move values between memory and registers or between two registers.

Program Counter (PC): **Memory address of next instr**

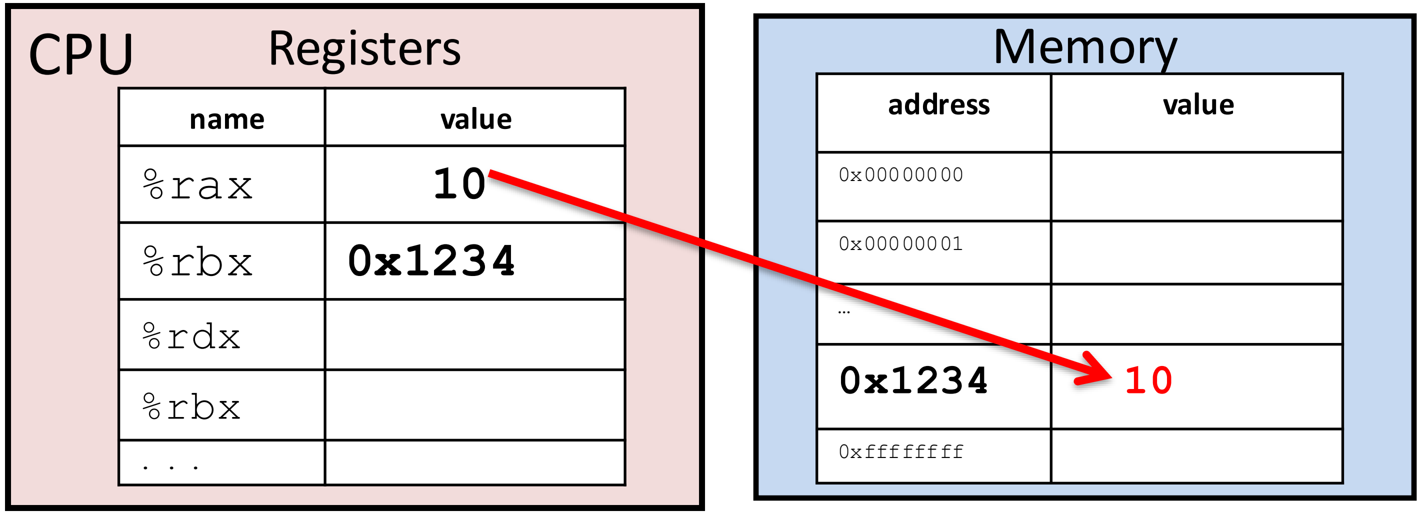
Instruction Register (IR): **Instruction contents (bits)**



Data Movement: Four Addressing Modes

- Instructions need to be told where to get operands or store results
- Variety of options for how to *address* those locations

- A location might be:
  1. A register
  2. A literal/immediate value
  3. A location in memory
  4. An offset from a location in memory



CPU Registers

name	value
`%rax`	**10**
`%rbx`	**0x1234**
`%rdx`	
`%rbx`	
. . .	

Memory

address	value
0x00000000	
0x00000001	
…	
0x1234	10
0xffffffff	

- In x86_64, an instruction can access at most one memory location (e.g., one memory location and register OR two registers)

Addressing Modes

- Instructions need to be told where to get operands or store results

- Variety of options for how to *address* those locations

- A location might be:
  - A register
  - A location in memory

- In x86_64, an instruction can access *at most* one memory location

Four Addressing Modes: Register

- Instructions can refer to the name of a register
- Examples: `MOV S,D  # D ← S`
  - `mov %rax, %rbx`
  - `# Copy the contents of %rax into %rbx –  overwrites %rbx, no change to %rax`
  - `add %r9, %rdx`
  - `# Add the contents of %r9 and %rdx, store the result in %rdx, no change to %r9`

Four Addressing Modes: Immediate

- Refers to a constant or "literal" value, starts with **$**
- Allows programmer to hard-code a number
- Can be either decimal (no prefix) or hexadecimal (0x prefix)

```
mov $10, %rax  # Put the constant value 10 in register rax.
add $0xF, %rdx # Add 15 (0xF) to %rdx and store result in %rdx
```

Four Addressing Modes: Memory

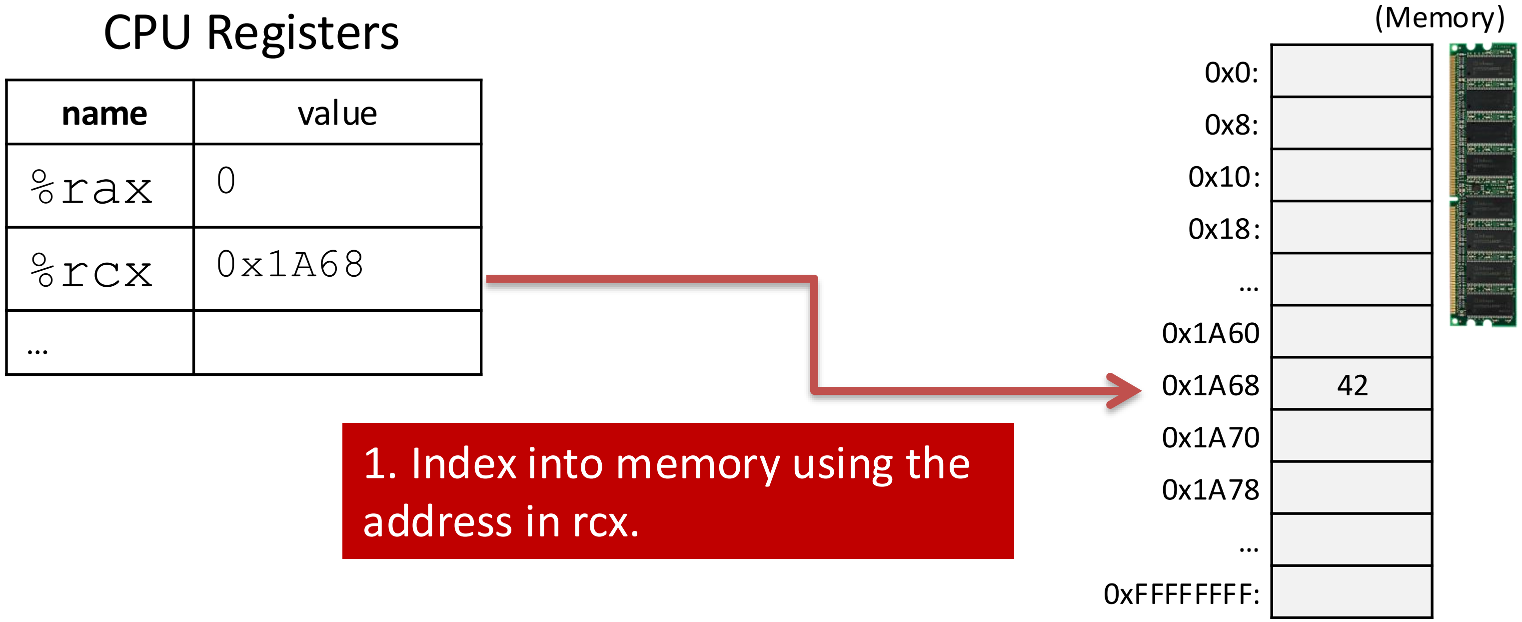
- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with () around the register’s name.

```
mov (%rcx), %rax
# Treat the value %rcx as an index into main memory, retrieve the value ,
and store the value in register %rax
```

Addressing Mode: Memory

`movq (%rcx), %rax`

- Use the address in register %rcx to access memory,
- then, *store result at that memory address* in register %rax



CPU Registers

name	value
`%rax`	`0`
`%rcx`	`0x1A68`
…	

(Memory)

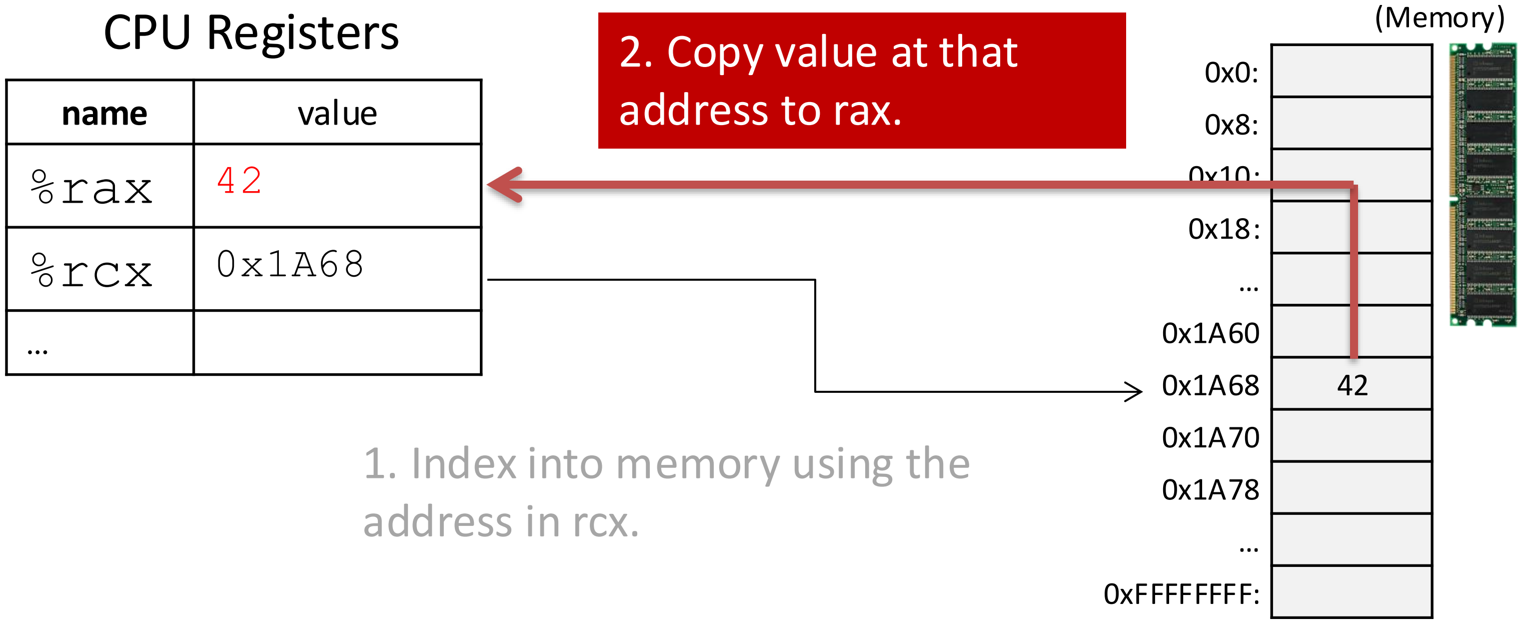
0x0:	
0x8:	
0x10:	
0x18:	
…	
0x1A60	
0x1A68	42
0x1A70	
0x1A78	
…	
0xFFFFFFFF:	

1. Index into memory using the address in rcx.

Addressing Mode: Memory

`movq (%rcx), %rax`

- Use the address in register %rcx to access memory,
- then, *store result at that memory address* in register %rax



CPU Registers

name	value
`%rax`	42
`%rcx`	`0x1A68`
…	

2. Copy value at that address to rax.

1. Index into memory using the address in rcx.

(Memory)

0x0:	
0x8:	
0x10:	
0x18:	
…	
0x1A60	
0x1A68	42
0x1A70	
0x1A78	
…	
0xFFFFFFFF:	

Addressing Mode: Register

- Instructions can refer to the name of a register

- Examples:
  - `movq %rax, %r15`
    (Copy the contents of %rax into %r15 -- overwrites %r15, no change to %rax)
  - `addq %r9, %rdx`
    (Add the contents of %r9 and %rdx, store the result in %rdx, no change to %r9)

Addressing Mode: Immediate

- Refers to a constant or “literal” value, starts with $
- Allows programmer to hard-code a number
- Can be either decimal (no prefix) or hexadecimal (0x prefix)

```
movq $10, %rax
```

  – Put the constant value 10 in register rax.

```
addq $0xF, %rdx
```

  – Add 15 (0xF) to %rdx and store the result in %rdx.

Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with () around the register’s name.

```
movq (%rcx), %rax
```

- Use the address in register %rcx to access memory, store result in register %rax

Addressing Mode: Displacement

- Like memory mode, but with a constant offset
  - Offset is often negative, relative to %rbp

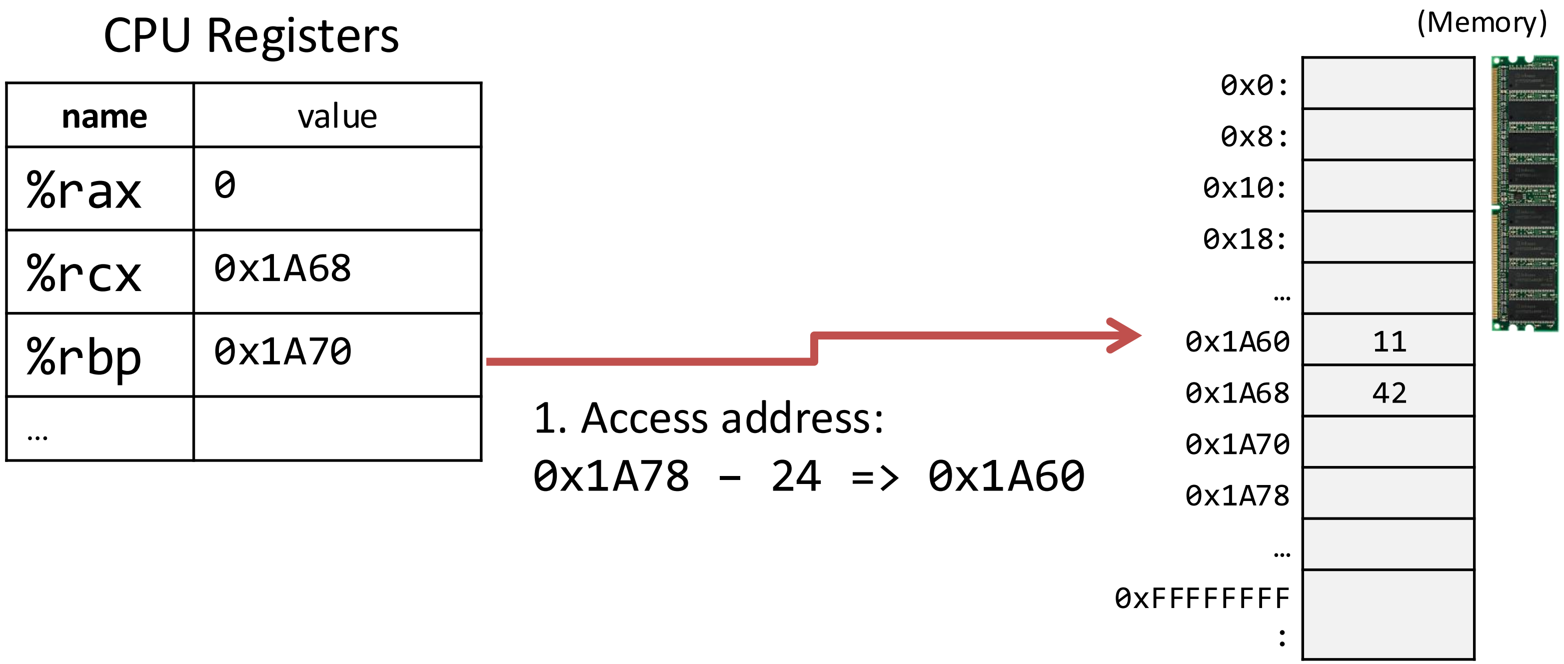
```
movq -16(%rbp), %rax
```

  - Take the address in %rbp, subtract 16 from it, index into memory and store the result in %rax.

Addressing Mode: Displacement

`movl -16(%rbp), %rax`

- Take the address in %rbp, subtract 16 from it, index into memory and store the result in %rax.



CPU Registers

name	value
`%rax`	`0`
`%rcx`	`0x1A68`
`%rbp`	`0x1A70`
…	

1. Access address:
`0x1A78 – 24 => 0x1A60`

(Memory)

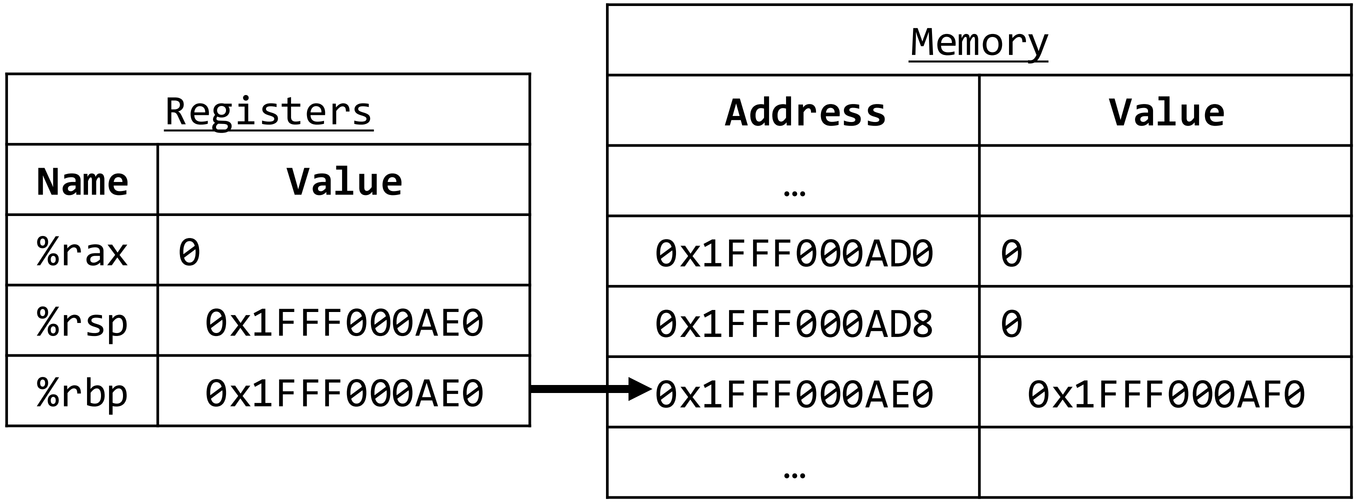
`0x0:`	
`0x8:`	
`0x10:`	
`0x18:`	
…	
`0x1A60`	11
`0x1A68`	42
`0x1A70`	
`0x1A78`	
…	
`0xFFFFFFFF:`	

What will the state of registers and memory look like after executing these instructions?

```
sub  $16, %rsp
movq $3, -8(%rbp)
mov  $10, %rax
sal  $1, %rax
add  -8(%rbp), %rax
movq %rax, -16(%rbp)
add  $16, %rsp
```

x is stored at rbp-8

y is stored at rbp-16



Registers

Name	Value
%rax	0
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

Memory

Address	Value
…	
0x1FFF000AD0	0
0x1FFF000AD8	0
0x1FFF000AE0	0x1FFF000AF0
…	

What will the state of registers and memory look like after executing these instructions?

```
subq  $16, %rsp
movq  $3, -8(%rbp)
movq  $10, %rax
sal   $1, %rax
addq -8(%rbp), %rax
movq  %rax, -16(%rbp)
addq  $16, %rsp
```

x is stored at `rbp-8`

y is stored at `rbp-16`

A.

Registers	
Name	**Value**
%rax	2
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

Memory	
Address	**Value**
0x1FFF000AD0	3
0x1FFF000AD8	10
0x1FFF000AE0	0x1FFF000AF0

B.

Registers	
Name	**Value**
%rax	10
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

Memory	
Address	**Value**
0x1FFF000AD0	23
0x1FFF000AD8	10
0x1FFF000AE0	0x1FFF000AF0

C.

Registers	
Name	**Value**
%rax	23
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

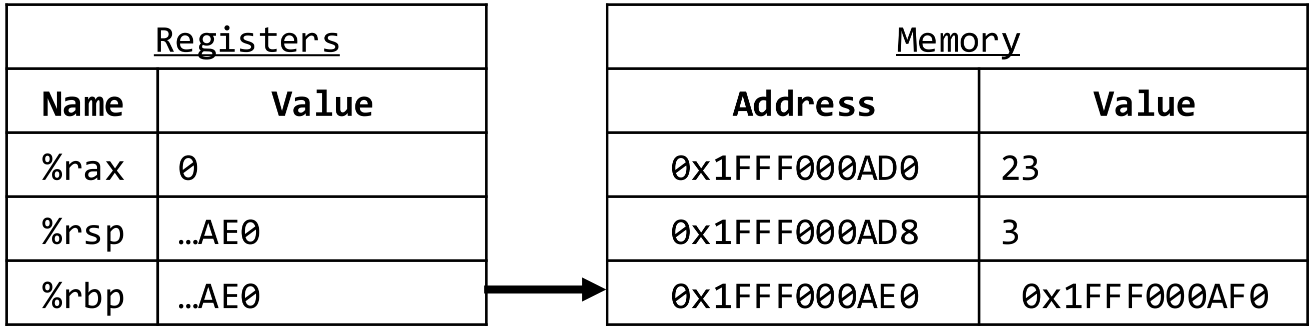
Memory	
Address	**Value**
0x1FFF000AD0	23
0x1FFF000AD8	3
0x1FFF000AE0	0x1FFF000AF0

Solution

```
subq  $16, %rsp
movq  $3, -8(%rbp)
movq  $10, %rax
sal   $1, %rax
addq -8(%rbp), %rax
movq  %rax, -16(%rbp)
addq  $16, %rsp
```

x is stored at rbp-8

y is stored at rbp-16



Registers

Name	Value
%rax	0
%rsp	…AE0
%rbp	…AE0

Memory

Address	Value
0x1FFF000AD0	23
0x1FFF000AD8	3
0x1FFF000AE0	0x1FFF000AF0

Assembly Visualization Tool

- The authors of Dive into Systems, including Swarthmore faculty with help from Swarthmore students, have developed a tool to help visualize assembly code execution:
- https://asm.diveintosystems.org
- For this example, use the arithmetic mode.

```
subq  $16, %rsp
movq  $3, -8(%rbp)
movq  $10, %rax
sal   $1, %rax
addq -8(%rbp), %rax
movq  %rax, -16(%rbp)
addq  $16, %rsp
```

x is stored at rbp-8

y is stored at rbp-16

Solution

C code equivalent:

```
x = 3;
y = x + (10 << 1);
```

```
subq  $16, %rsp           Subtract constant 16 from %rsp
movq  $3, -8(%rbp)        Move constant 3 to address %rbp-8
movq  $10, %rax           Move constant 10 to register %rax
sal   $1, %rax            Shift the value in %rax left by 1 bit
addq  -8(%rbp), %rax       Add the value at address %rbp-8 to %rax
movq  %rax, -16(%rbp)    Store the value in %rax at address rbp-16
addq  $16, %rsp           Add constant 16 to %rsp
```

x is stored at rbp-8

y is stored at rbp-16

Registers	
Name	**Value**
%rax	23
%rsp	…AE0
%rbp	…AE0

Memory	
Address	**Value**
0x1FFF000AD0	23
0x1FFF000AD8	3
0x1FFF000AE0	0x1FFF000AF0

What will the state of registers and memory look like after executing these instructions?

```
…
movq  %rbp, %rcx
subq  $8, %rcx
movq (%rcx), %rax
or   %rax, -16(%rbp)
neg  %rax
```

Registers

Name	Value
%rax	0
%rcx	0
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0 →

Memory

Address	Value
…	
0x1FFF000AD0	8
0x1FFF000AD8	5
0x1FFF000AE0	0x1FFF000AF0
…	

How might you implement the following C code in assembly?

z = x ^ y

x is stored at %rbp-8

y is stored at %rbp-16

z is stored at %rbp-24

Registers

Name	Value
%rax	0
%rdx	0
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

Memory

Address	Value
0x1FFF000AC8	(z)
0x1FFF000AD0	(y)
0x1FFF000AD8	(x)
0x1FFF000AE0	0x1FFF000AF0
…	

A:

```
movq -8(%rbp), %rax
movq -16(%rbp), %rdx
xor  %rax, %rdx
movq %rax, -24(%rbp)
```

B:

```
movq -8(%rbp), %rax
movq -16(%rbp), %rdx
xor  %rdx, %rax
movq %rax, -24(%rbp)
```

C:

```
movq -8(%rbp), %rax
movq -16(%rbp), %rdx
xor  %rax, %rdx
movq %rax, -8(%rbp)
```

D:

```
movq -24(%rbp), %rax
movq -16(%rbp), %rdx
xor  %rdx, %rax
movq %rax, -8(%rbp)
```

How might you implement the following C code in assembly?

x = y >> 3 | x * 8

x is stored at %rbp-8

y is stored at %rbp-16

z is stored at %rbp-24

Registers

Name	Value
%rax	0
%rdx	0
%rsp	0x1FFF000AE0
%rbp	0x1FFF000AE0

Memory

Address	Value
0x1FFF000AC8	(z)
0x1FFF000AD0	(y)
0x1FFF000AD8	(x)
0x1FFF000AE0	0x1FFF000AF0
…	

Solutions (other instruction sequences can work too!)

- z = x ^ y

```
movq -8(%rbp), %rax
movq -16(%rbp), %rdx
xor  %rdx, %rax
movq %rax, -24(%rbp)
```

- `x = y >> 3 | x * 8`

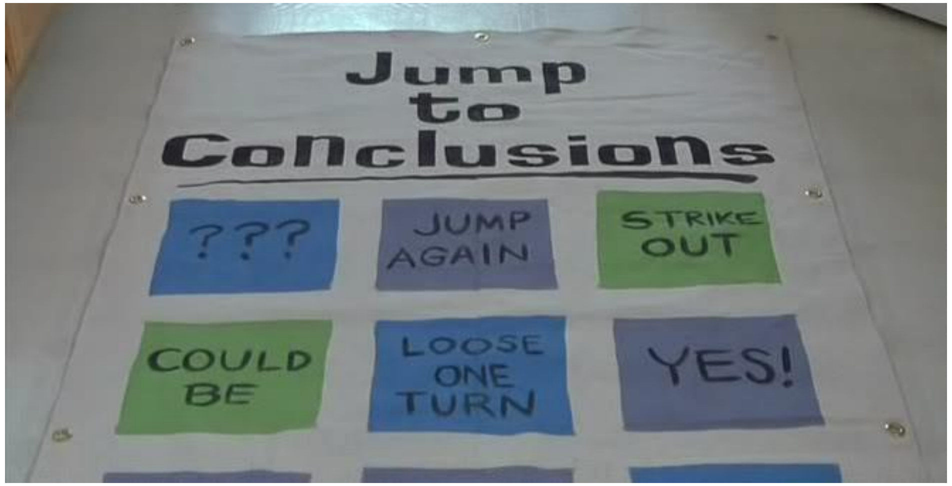
```
mov  -8(%rbp), %rax
imul $8, %rax
movq -16(%rbp), %rdx
sar  $3, %rdx
or   %rax, %rdx
movq %rdx, -8(%rbp)
```

Control Flow

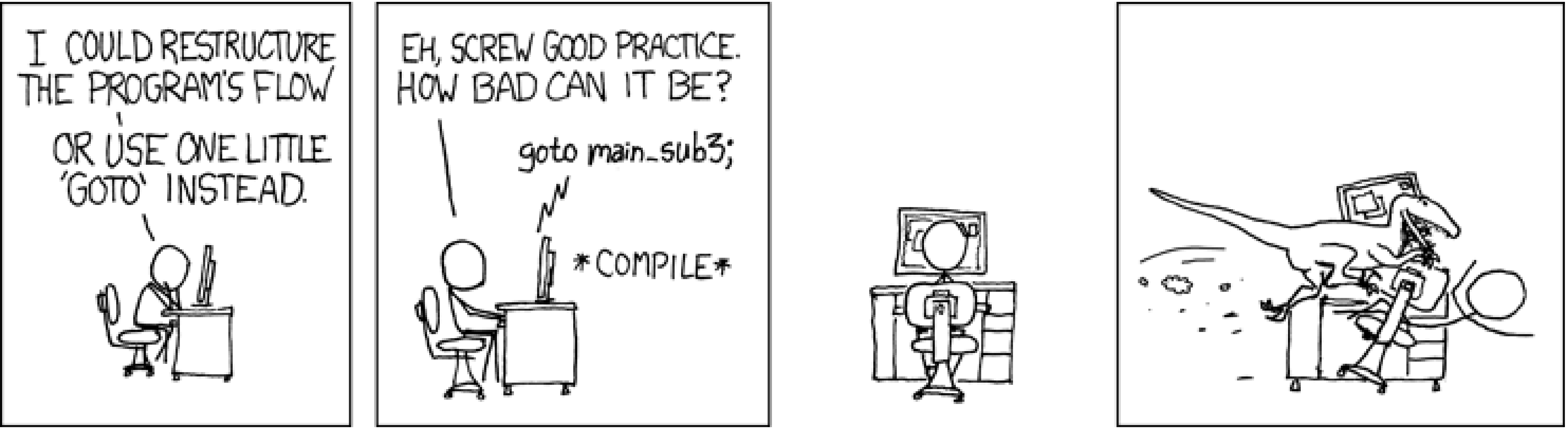
- Previous examples focused on:
  - data movement (mov, movq)
  - arithmetic (add, sub, or, neg, sal, etc.)

- Up next: Jumping!

  (Changing which instruction we execute next.)



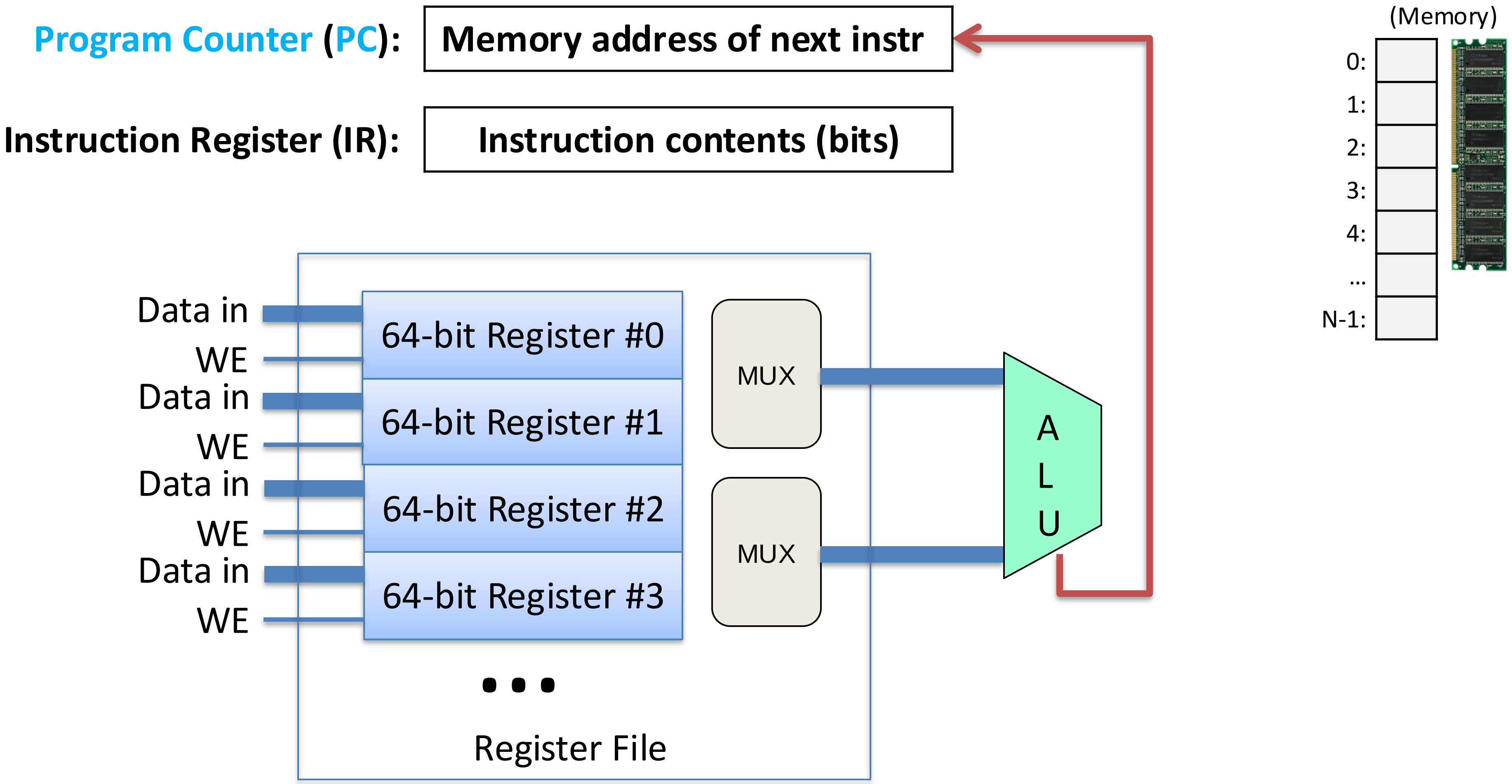
Relevant XKCD



xkcd #292

3. Control

Change **PC** based on ALU **condition code** state.



Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Unconditional Jumping / Goto

```
int main(void) {
  long a = 10;
  long b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

A label is a place you <u>might</u> jump to.

Labels ignored except for goto/jumps.

(Skipped over if encountered)

```
  int x = 20;
L1:
  int y = x + 30;
L2:
  printf(“%d, %d\n”, x, y);
```

Unconditional Jumping / Goto

```c
int main(void) {
   long a = 10;
   long b = 20;

   goto label1;
   a = a + b;

label1:
   return;
```

```
  pushq %rbp
  mov   %rsp, %rbp
  sub   $16, %rsp
  movq $10, -16(%ebp)
  movq $20, -8(%ebp)
  jmp   label1
  movq -8(%rbp), $rax
  add   $rax, -16(%rbp)
  movq -16(%rbp), %rax
label1:
  leave
```

Unconditional Jumping / Goto

Use of unconditional jumping besides `goto`?

  - infinite loop
  - break;
  - continue;
  - functions (handled differently)

- Often, we only want to jump when *something* is true / false

- Need some way to compare values, jump based on comparison results

```
  pushq %rbp
  mov   %rsp, %rbp
  sub   $16, %rsp
  movq $10, -16(%ebp)
  movq $20, -8(%ebp)
  jmp   label1
  movq -8(%rbp), $rax
  add  $rax, -16(%rbp)
  movq -16(%rbp), %rax
label1:
  leave
```

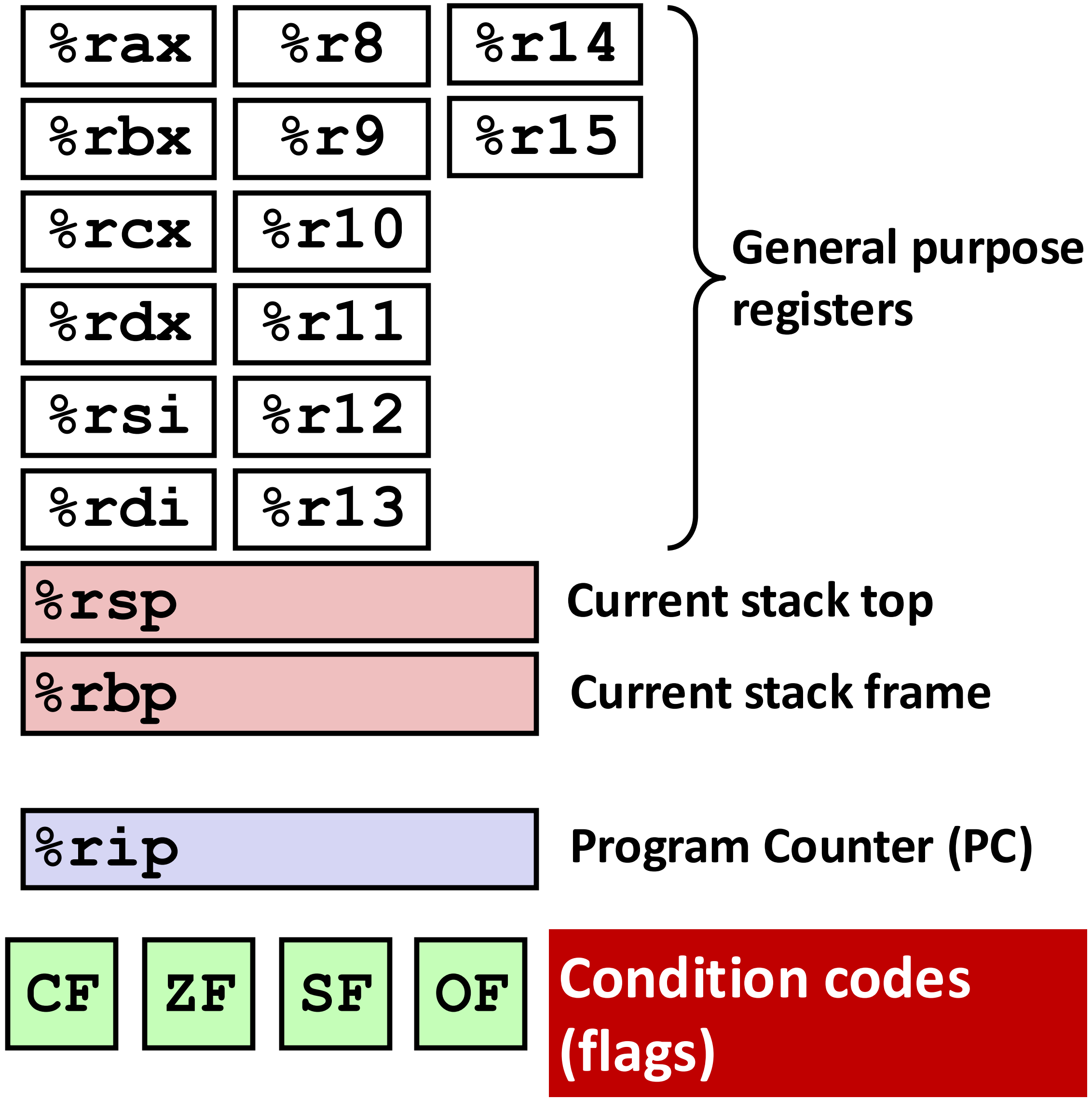
Condition Codes (or Flags)

- Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions (e.g., cmp, test)

- x86_64 condition codes tell you:
  - ZF — zero flag — if the result is zero
  - SF — sign flag — if the result’s first bit is set (negative if signed)
  - CF — carry flag — if the result overflowed (assuming unsigned) [“carried”]
  - OF — overflow flag —if the result overflowed (assuming signed)

Processor State in Registers

- Working memory for currently executing program
  - Temporary data (%rax - %r15)
  - Location of runtime stack (%rbp, %rsp)
  - Address of next instruction to execute (%rip)
  - Status of recent ALU tests (CF, ZF, SF, OF)



`%rax` `%r8` `%r14`
`%rbx` `%r9` `%r15`
`%rcx` `%r10`
`%rdx` `%r11`
`%rsi` `%r12`
`%rdi` `%r13`

General purpose registers

`%rsp` **Current stack top**

`%rbp` **Current stack frame**

`%rip` **Program Counter (PC)**

`CF` `ZF` `SF` `OF` **Condition codes (flags)**

Control Instructions

Change control flow (next instr is not sequential)

- Sometimes conditional: if(cond)-else, for(cond)
- Sometimes not: foo(), return

Use Condition Codes: %EFLAGS bit vector

Describe attributes of most recent arithmetic/logic op

- **CF** Carry Flag (did op result in unsigned overflow?)
  (a carry-out bit for ADD and no-carryout bit for SUB)
- **SF** Sign Flag (is the result negative? (is high-order bit 1?))
- **ZF** Zero Flag (is the result zero?)
- **OF** Overflow Flag (did op result in signed overflow?)

Implicitly set as the result of some (not all) ops:

`addq %eax, %ecx`   #adds and sets %EFLAGS bits

Instructions that set condition codes

1. Arithmetic/logic side effects (add, sub, or, etc.)

2. CMP and TEST: Does not change state of registers, only condition codes

`cmp b, a` like computing **`a-b`** without storing result

- Sets OF if overflow, Sets CF if carry-out,
  Sets ZF if result zero, Sets SF if results is negative

`test b, a` like computing **`a&b`** without storing result

- Sets ZF if result zero, sets SF if a&b < 0
  OF and CF flags are zero (there is no overflow with &)

Which flags would this `sub` set?

Suppose **%rax** holds **5**, **%rcx** holds **7**

```
sub $5, %rax
```

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and OF

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

Which flags would this `sub` set?

Suppose **%rax** holds **5**, **%rcx** holds **7**

```
sub $5, %rax
```

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and OF

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

Which flags would this `sub` set?

Suppose **%rax** holds **5**, **%rcx** holds **7**

```
cmp $5, %rax
```

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and OF

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

Which flags would this `sub` set?

Suppose **%rax** holds **5**, **%rcx** holds **7**

```
cmp $5, %rax
```

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and OF

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

How could we use jumps/CCs to implement this C code?

```
long userval;
scanf("%ld", &userval);

if (userval == 42) {
  userval = userval + 5;
} else {
  userval = userval - 10;
}
```

Assume userval is stored in %rax at this point.

(A)

```
 cmp $42, %rax
 je L2
L1:
 sub $10, %rax
 jmp DONE
L2:
 add $5, %rax
DONE:
```

(B)

```
 cmp $42, %rax
 jne L2
L1:
 sub $10, %rax
 jmp DONE
L2:
 add $5, %rax
DONE:
```

(C)

```
 cmp $42, %rax
 jne L2
L1:
 add $5, %rax
 jmp DONE
L2:
 sub $10, %rax
DONE:
```

How could we use jumps/CCs to implement this C code?

```
long userval;
scanf("%ld", &userval);

if (userval == 42) {
  userval = userval + 5;
} else {
  userval = userval - 10;
}
```

Assume userval is stored in %rax at this point.

(A)

```
  cmp $42, %rax
  je L2
L1:
  sub $10, %rax
  jmp DONE
L2:
  add $5, %rax
DONE:
```

(B)

```
  cmp $42, %rax
  jne L2
L1:
  sub $10, %rax
  jmp DONE
L2:
  add $5, %rax
DONE:
```

(C)

```
  cmp $42, %rax
  jne L2
L1:
  add $5, %rax
  jmp DONE
L2:
  sub $10, %rax
DONE:
```

C Loops to x86_64

do-while: do { loop body } while (cond);	C goto translations: loop: loop body if(cond) goto loop

while:
 	
while(cond) {
loop body
}
	if(!cond) goto done
	loop:
	loop body
	if(cond) goto loop
	done:

for:
 	
for(init; cond; step){
loop body
}
	init code
	if(!cond) goto done
	loop:
	loop body
	step
	if(cond) goto loop
	done:

Convert to C goto:

```
x = 0;
for(i=0; i < 10; i++) {
 x = x + 1;
}
z = x * 3;
```

Example goto code

```
int main(void) {
  long a = 10;
  long b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

for:
for(init; cond; step){
loop body
}
	init code
	<fill in your answer here>

Convert to C goto:

```
x = 0;
for(i=0; i < 10; i++) {
 x = x + 1;
}
z = x * 3;
```

Example goto code

```
int main(void) {
  long a = 10;
  long b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

<u>for:</u>

```
for(init; cond; step){
  loop body
}
```

```
  init code
 if(!cond) goto done
loop:
  loop body
  step
  if(cond) goto loop
done:
```

Using Jump Instructions

- `jmp label` # unconditional jump (ex. `jmp .L2`)
- `jge label` # conditional jump (ex. if >=) (je, jne, js, jg, …)

(A label is a place you might jump to. Labels ignored except for goto/jumps)

Try out this code: what does it do?

```
   movq $0, %rax
   movq $4, %rbx
   movq $0, %rdx
   jmp  .L2
 .L1:
   addq $1, %rax
 .L2:
   addq %rax, %rdx
   cmp  %rax, %rbx   # R[%rbx] - R[%rax]
   jge .L1
```

CPU Registers

`%rax`	
`%rdx`	
`%rbx`	

Summary

- ISA defines what programmer can do on hardware
  - Which instructions are available
  - How to access state (registers, memory, etc.)
  - This is the architecture's *assembly language*

- In this course, we'll be using x86_64
  - Instructions for:
    - moving data (mov, movl, movq)
    - arithmetic (add, sub, imul, or, sal, etc.)
    - control (jmp, je, jne, etc.)
  - Condition codes for making control decisions
    - If the result is zero (ZF)
    - If the result's first bit is set (negative if signed) (SF)
    - If the result overflowed (assuming unsigned) (CF)
    - If the result overflowed (assuming signed) (OF)

Four Types of Assembly Instructions

1. Arithmetic: use ALU to compute a value
2. Data movement: load and store
3. Control Flow: branch, jump, etc.
4. Stack Instructions: push and pop stack frames
    - Shortcut instructions for common operations (we’ll cover these in detail later)

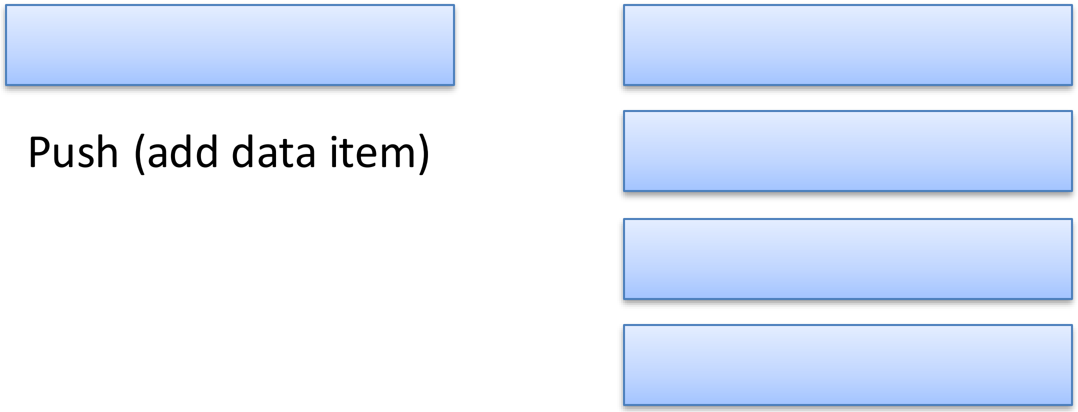
Overview

- Stack data structure, applied to memory
- Behavior of function calls
- Storage of function data, at assembly level

“A” Stack

- A stack is a basic data structure
  - Last in, first out behavior (LIFO)
  - Two operations
    - Push (add item to top of stack)
    - Pop (remove item from top of stack)

Pop (remove and return item)



Newest data

Push (add data item)

Oldest data

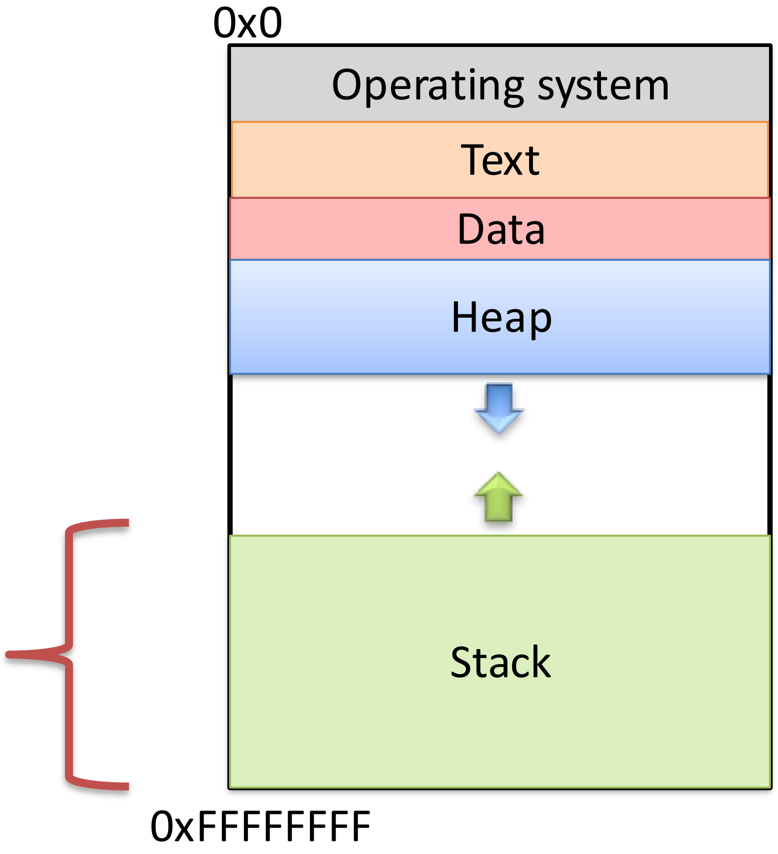
“The” Stack

- Apply stack data structure to memory
  - Store local (automatic) variables
  - Maintain state for functions (e.g., where to return)

- Organized into units called *frames*
  - One frame represents all of the information for one function.
  - Sometimes called *activation records*

Memory Model

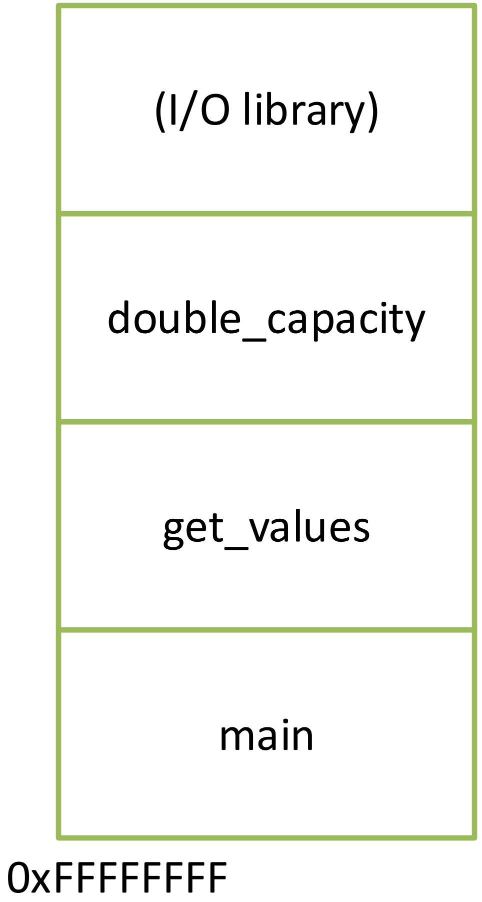
- Starts at the highest memory addresses, grows into lower addresses.



Stack Frames

- As functions get called,
  new frames added to stack.

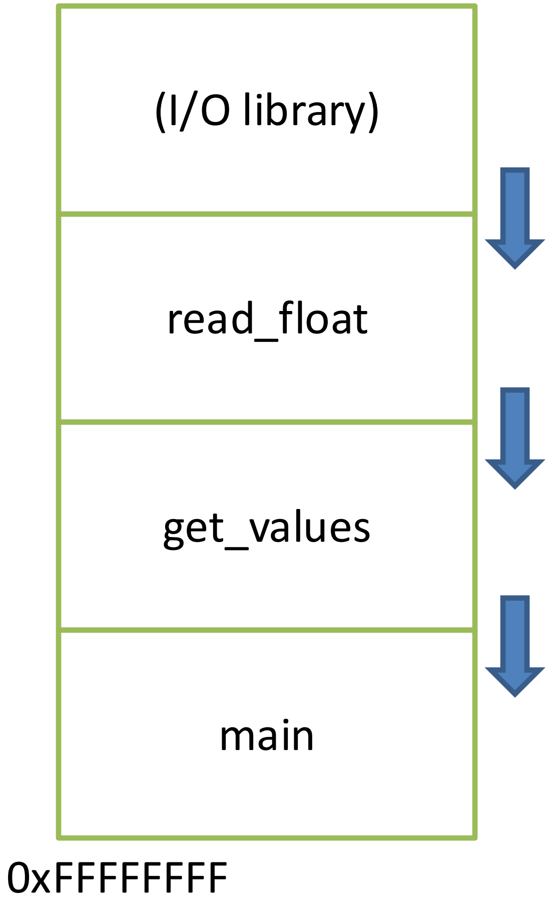
- Example: Lab 4
  - main calls get_values()
  - get_values calls double_capacity()
  - double_capacity calls I/O library



Stack Frames

- As functions get called,
  new frames added to stack.
- Example: Lab 4
  - main calls get_values()
  - get_values calls double_capacity()
  - double_capacity calls I/O library

All of this stack growing/shrinking happens automatically (from the programmer's perspective).



What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

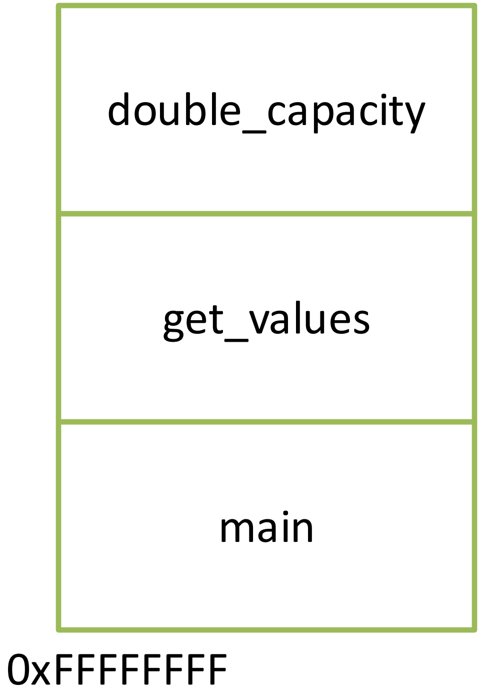
E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know / access?

- Local variables



double_capacity

get_values

main

0xFFFFFFFF

Local Variables

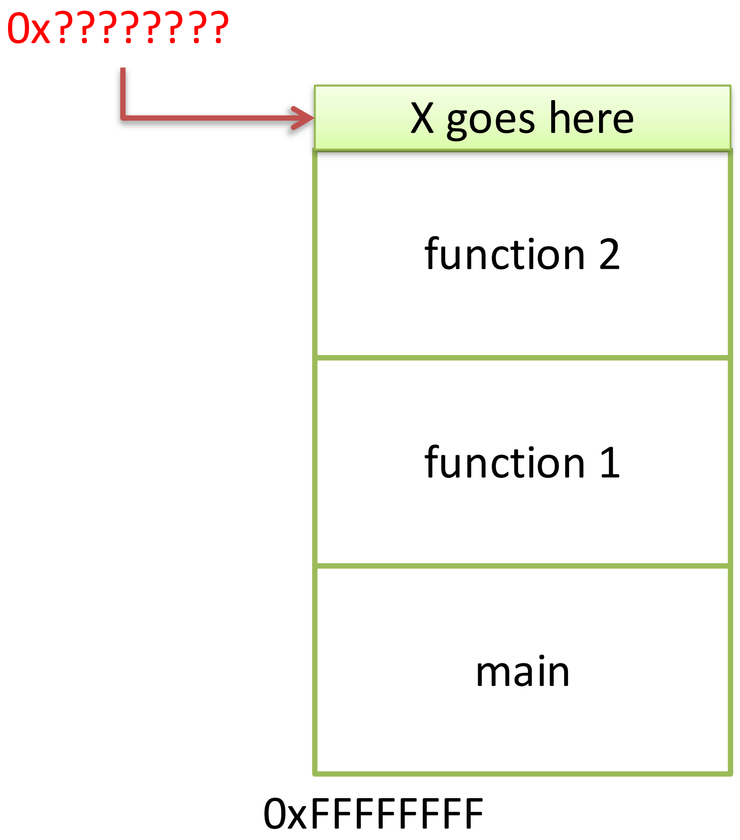
If the programmer says:

`int x = 0;`

Where should x be stored?

(Recall basic stack data structure)

Which memory address is that?



How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.

How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.

Program Characteristics

- Compile time (static)
  - Information that is known by analyzing your program
  - Independent of the machine and inputs

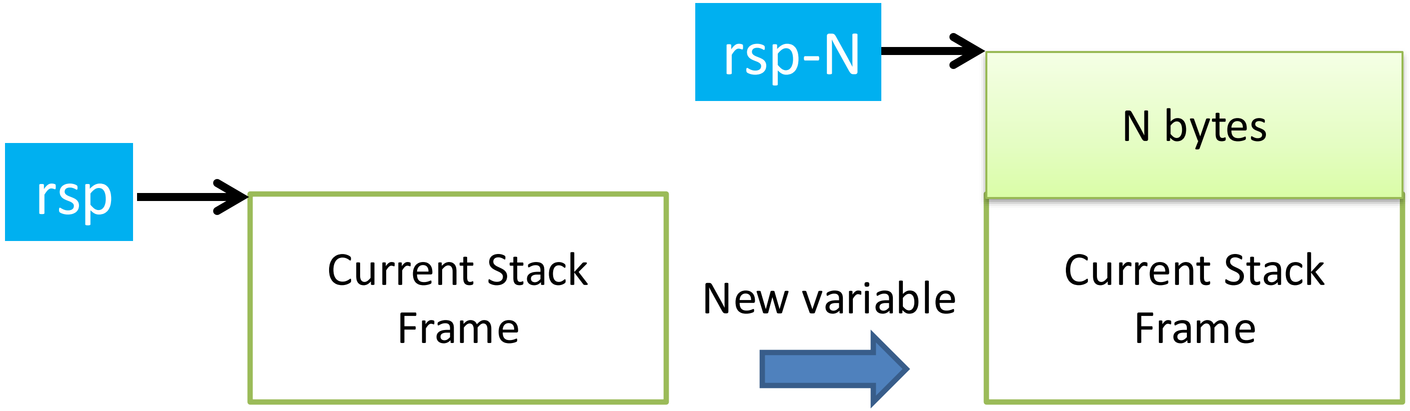
- Run time (dynamic)
  - Information that isn’t known until program is running
  - Depends on machine characteristics and user input

The Compiler Can…

- Perform type checking.
- Determine how much space you need on the stack to store local variables.
- Insert assembly instructions for you to set up the stack for function calls.
  - Create stack frames on function call
  - Restore stack to previous state on function return

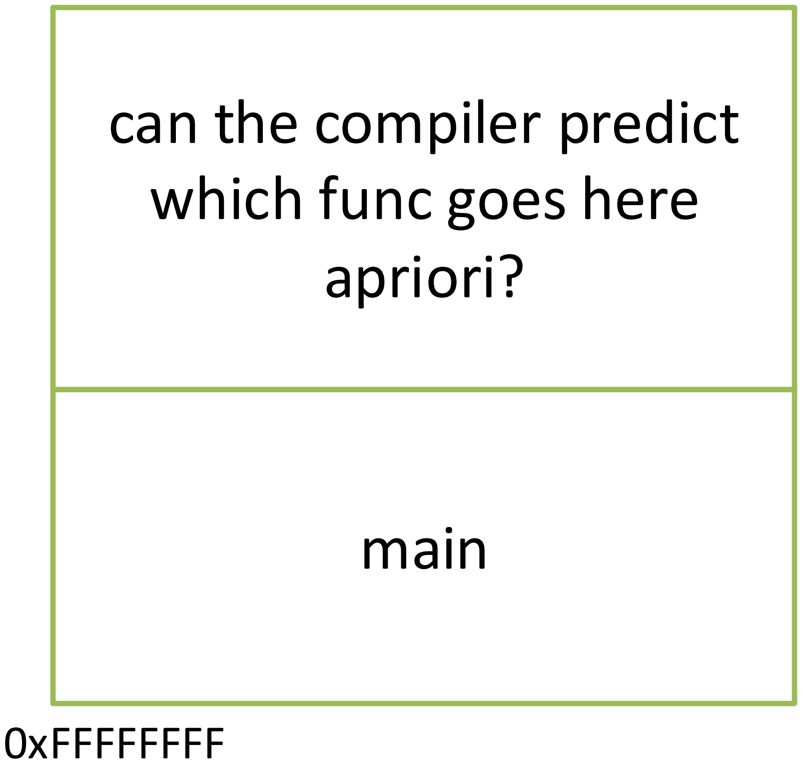
Local Variables

Compiler can allocate N bytes on the stack by subtracting N from the stack pointer: (rsp)



The Compiler Can't…Predict User Input

```
int main(void) {
  int decision = [read user input];
  if(decision > 5){
         funcA();
  }
   else{
         funcB();
  }
}
```



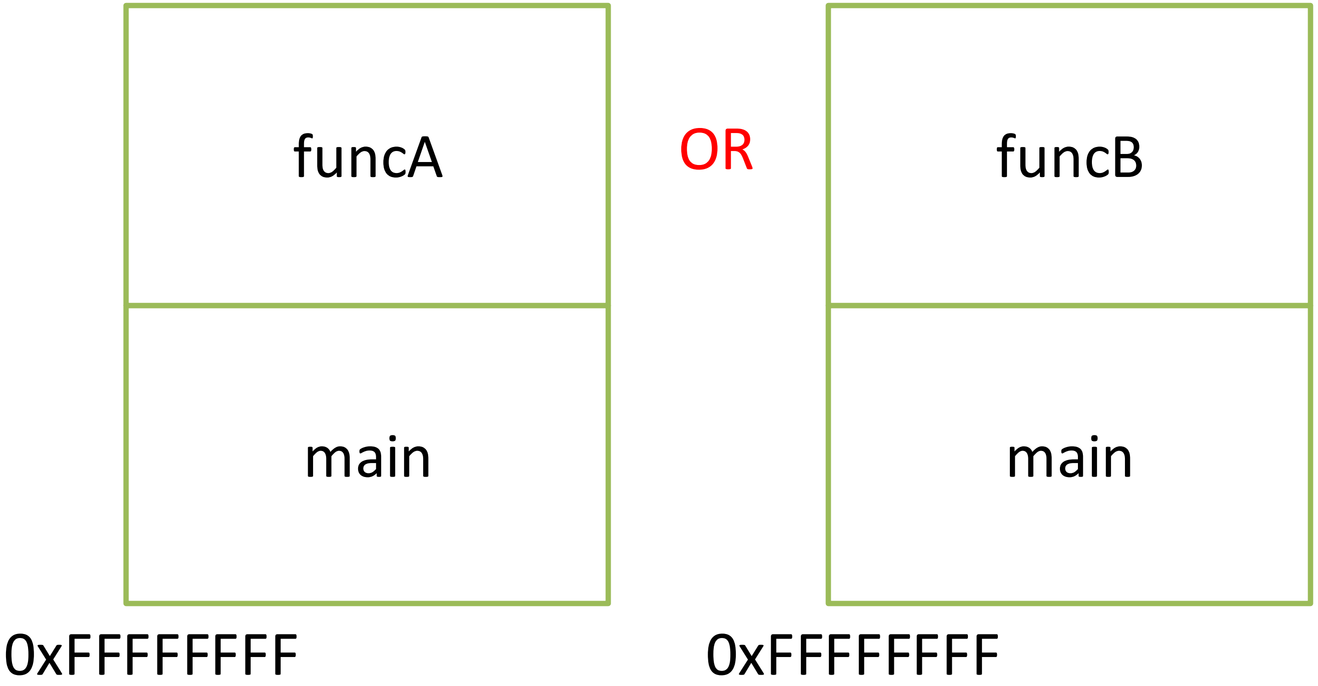
can the compiler predict which func goes here apriori?

main

0xFFFFFFFF

The Compiler Can't…Predict User Input

```
int main(void) {
   int decision = [read user input];
   if(decision > 5){
           funcA();
   }
   else{
           funcB();
   }
}
```



The Compiler Can't…

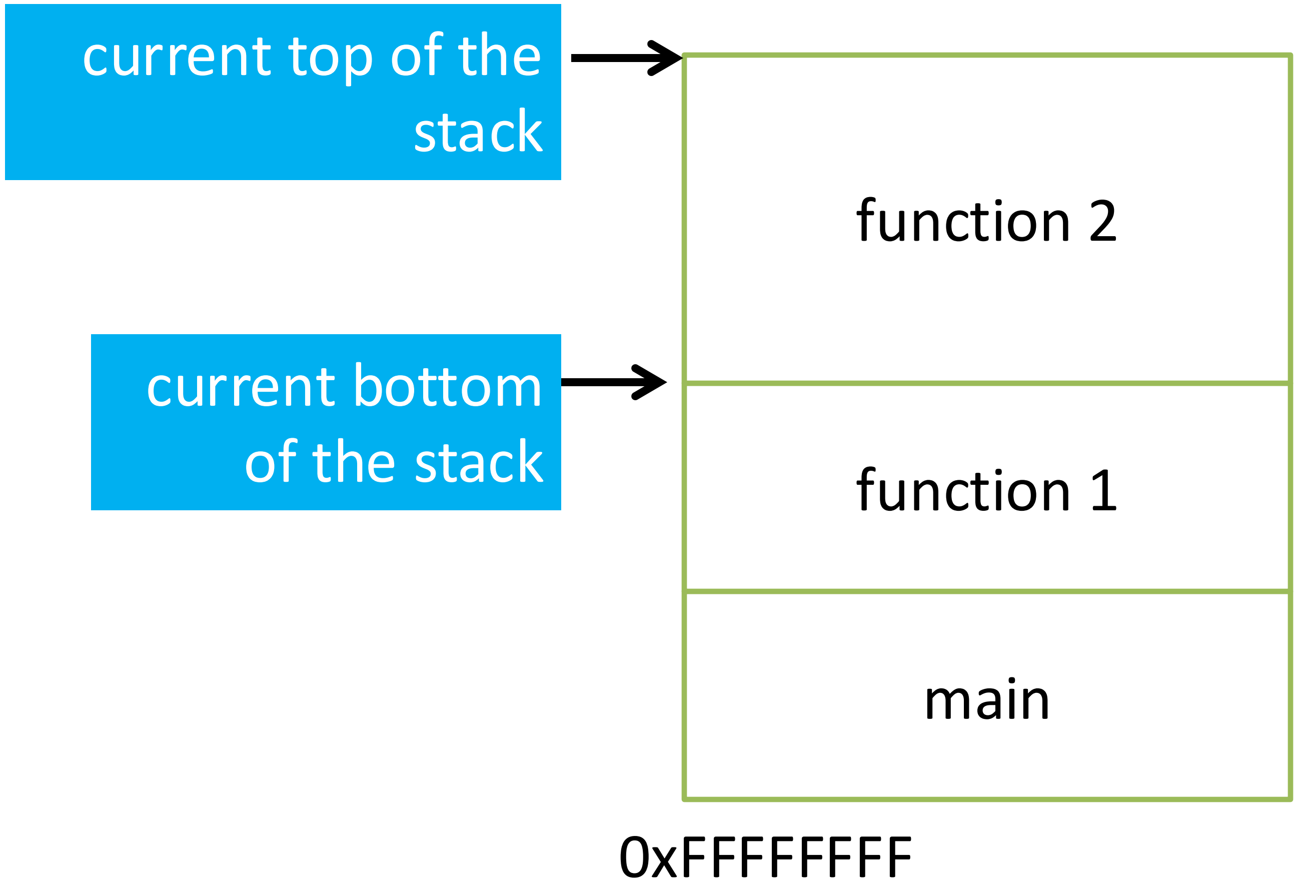
Predict user input.

Can't assume a function will always be at a certain address on the stack.

Alternative: create stack frames relative to the current (dynamic) state of the stack.

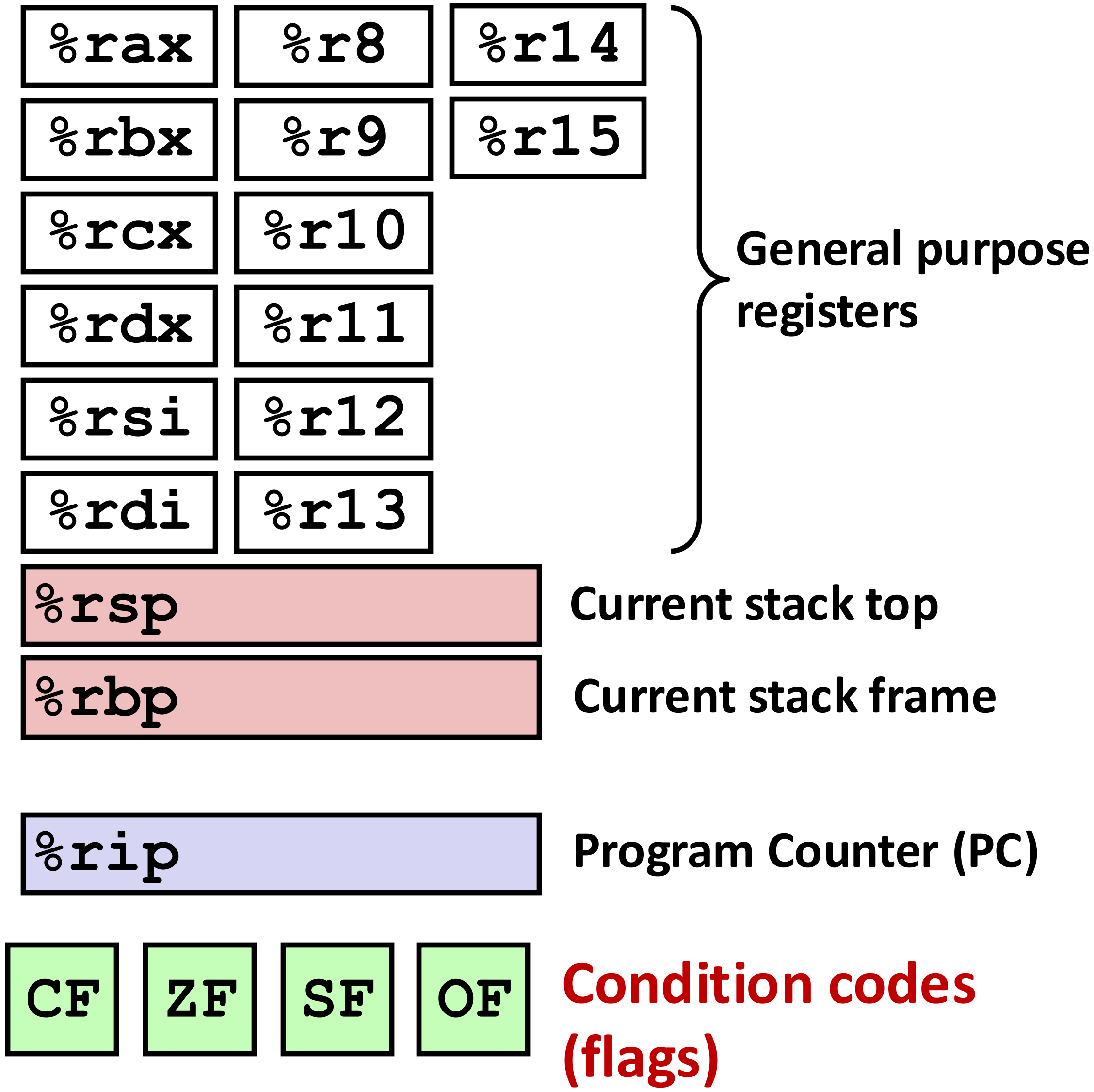
Stack Frame Location

Where in memory is the current stack frame?



Recall: x86_64 Register Conventions

- Working memory for currently executing program
  - Address of next instruction to execute (%rip)
  - Location of runtime stack (%rbp, %rsp)
  - Temporary data (%rax - %r15)
  - Status of recent ALU tests (CF, ZF, SF, OF)



`%rax` `%r8` `%r14`

`%rbx` `%r9` `%r15`

`%rcx` `%r10`

`%rdx` `%r11`

`%rsi` `%r12`

`%rdi` `%r13`

General purpose registers

`%rsp` **Current stack top**

`%rbp` **Current stack frame**

`%rip` **Program Counter (PC)**

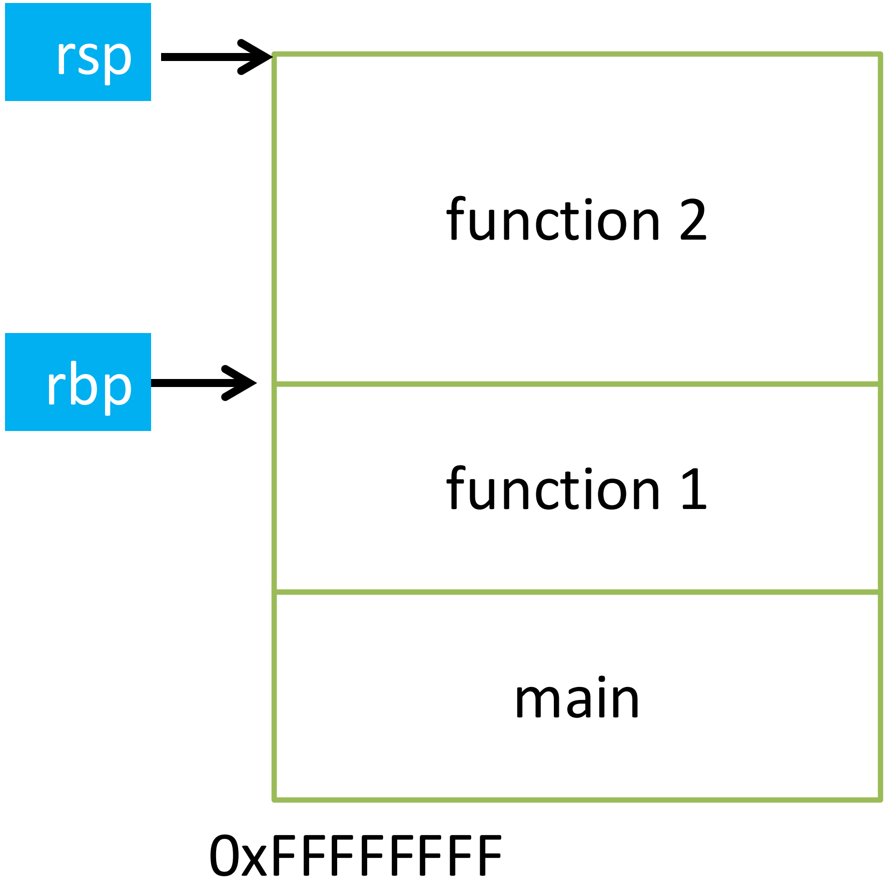
`CF` `ZF` `SF` `OF` **Condition codes (flags)**

Stack Frame Location

Where in memory is the current stack frame?

- rsp: stack pointer
- rbp: frame pointer (base pointer)

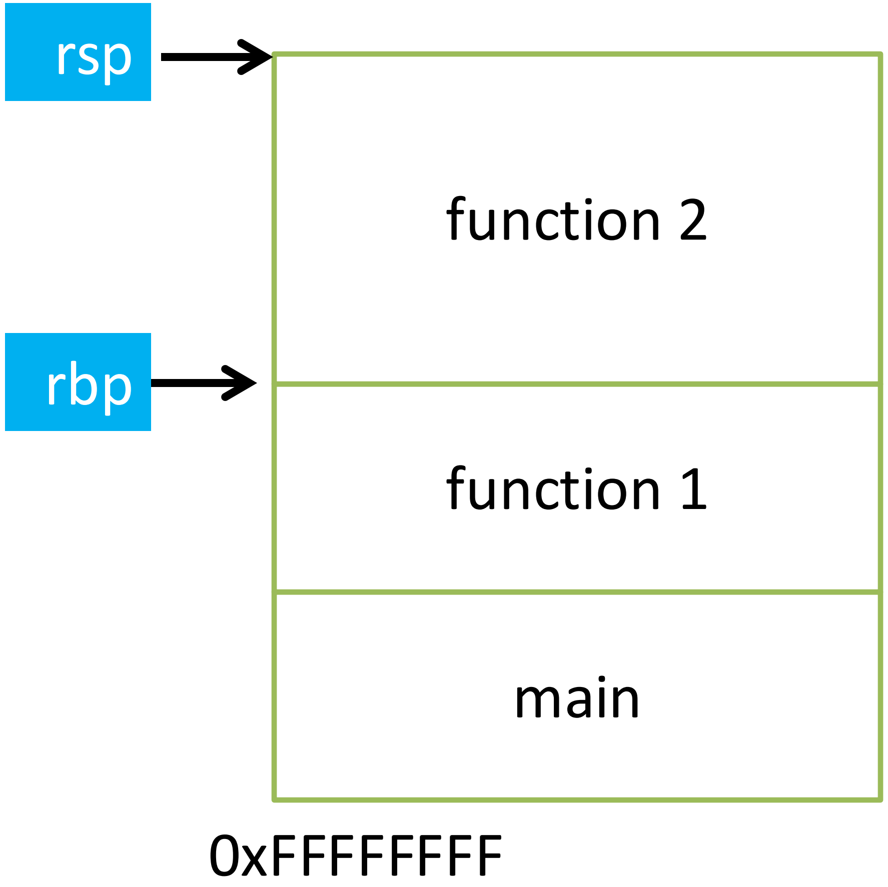
invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`



Stack Frame Location

- Compiler ensures that this invariant holds.
- This is why all local variables we've seen in assembly are relative to `rbp` or `rsp`!

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



How would we implement pushing x to the top of the stack in x86_64?

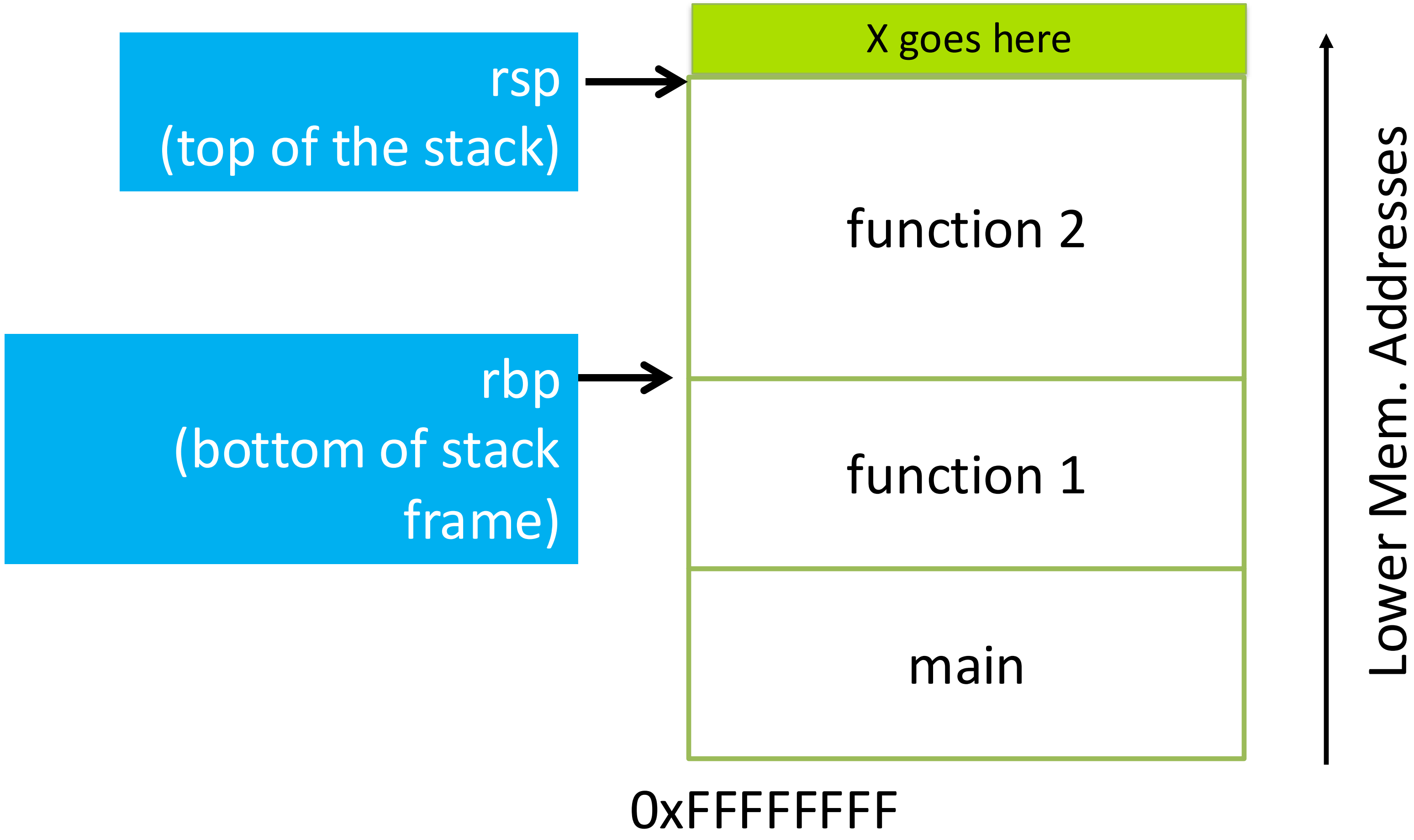
A. Increment rsp
Store x at (rsp)

B. Store x at (rsp)
Increment rsp

C. Decrement rsp
Store x at (rsp)

D. Store x at (rsp)
Decrement rsp

E. Copy rsp to rbp
Store x at rbp


rsp
(top of the stack)

rbp
(bottom of stack frame)

X goes here

function 2

function 1

main

Lower Mem. Addresses

0xFFFFFFFF

How would we implement pushing x to the top of the stack in x86_64?

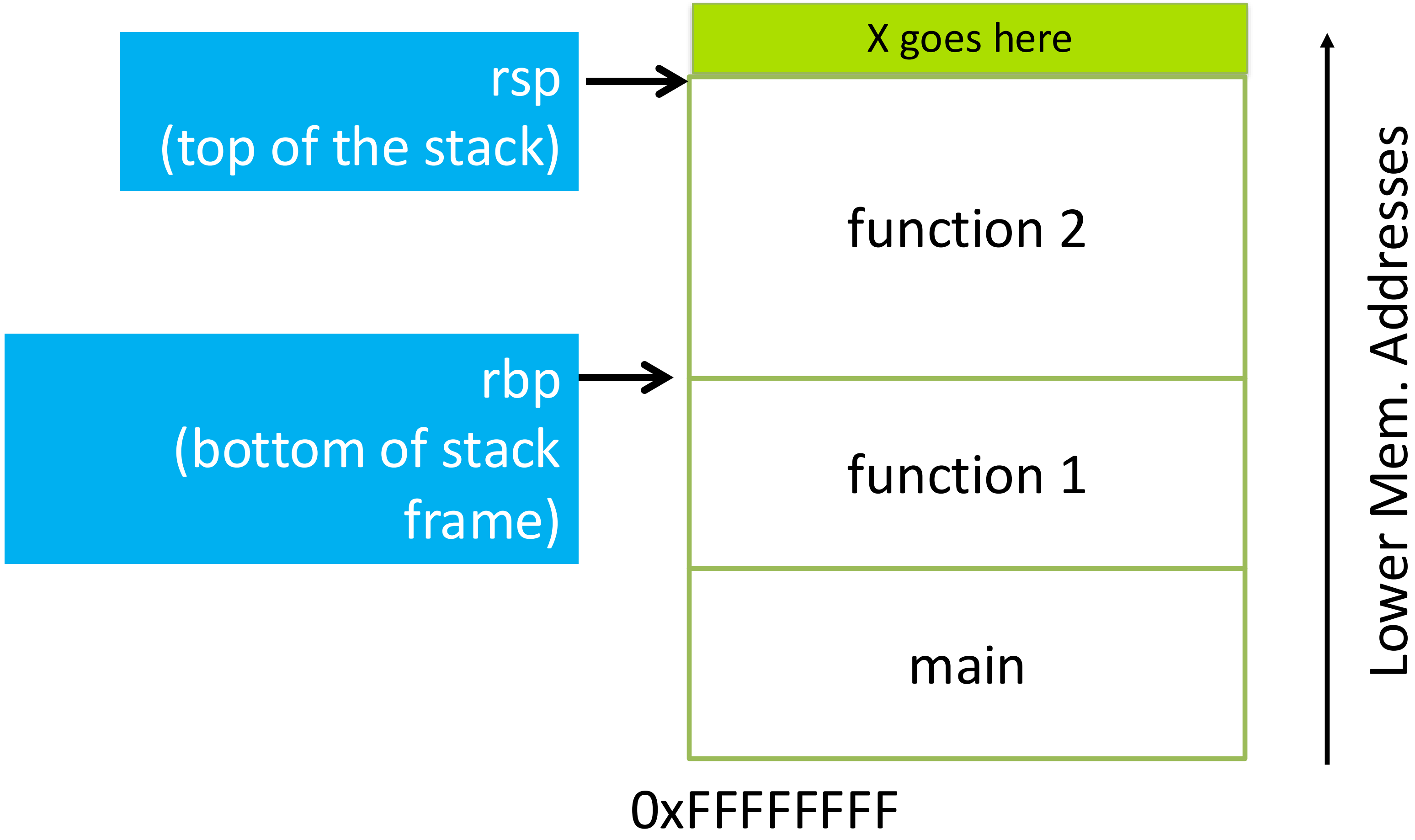
A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp



rsp
(top of the stack)

rbp
(bottom of stack frame)

X goes here

function 2

function 1

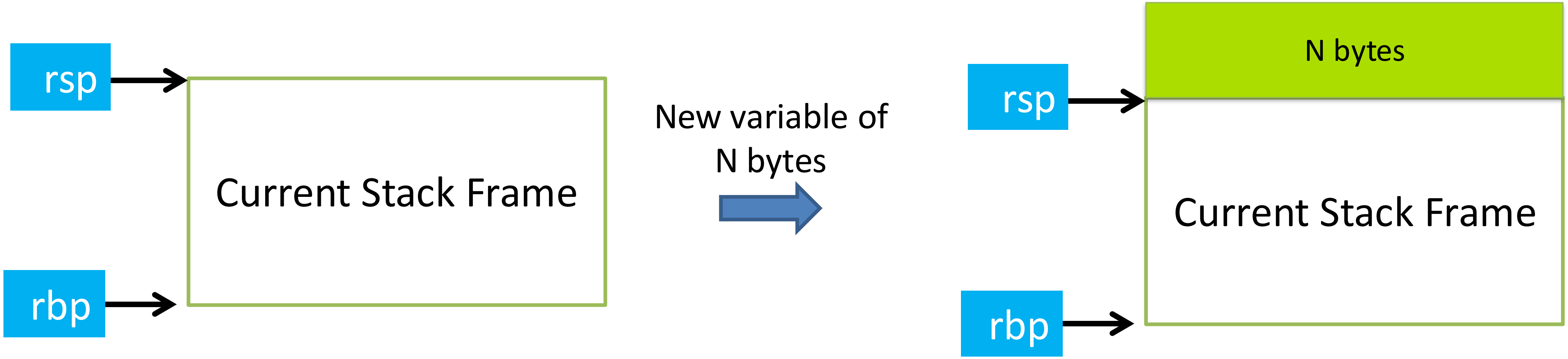
main

0xFFFFFFFF

Lower Mem. Addresses

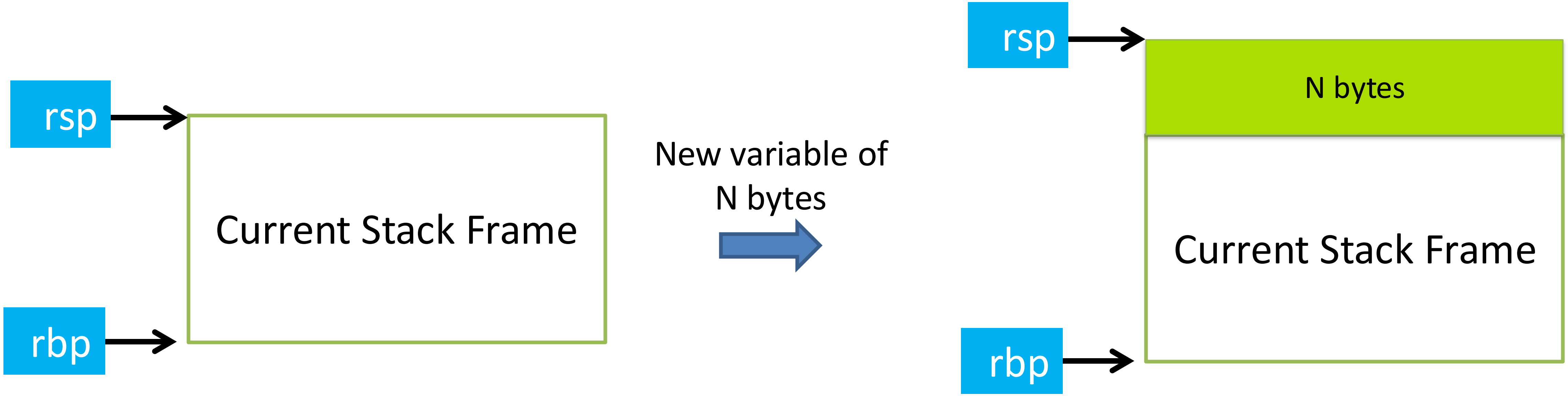
Local Variables

- Generally, we can make space on the stack for N bytes by:
  - <u>subtracting N from rsp</u>



Local Variables

- When we're done, free the space by adding N back to `rsp`
  - `rsp + N`

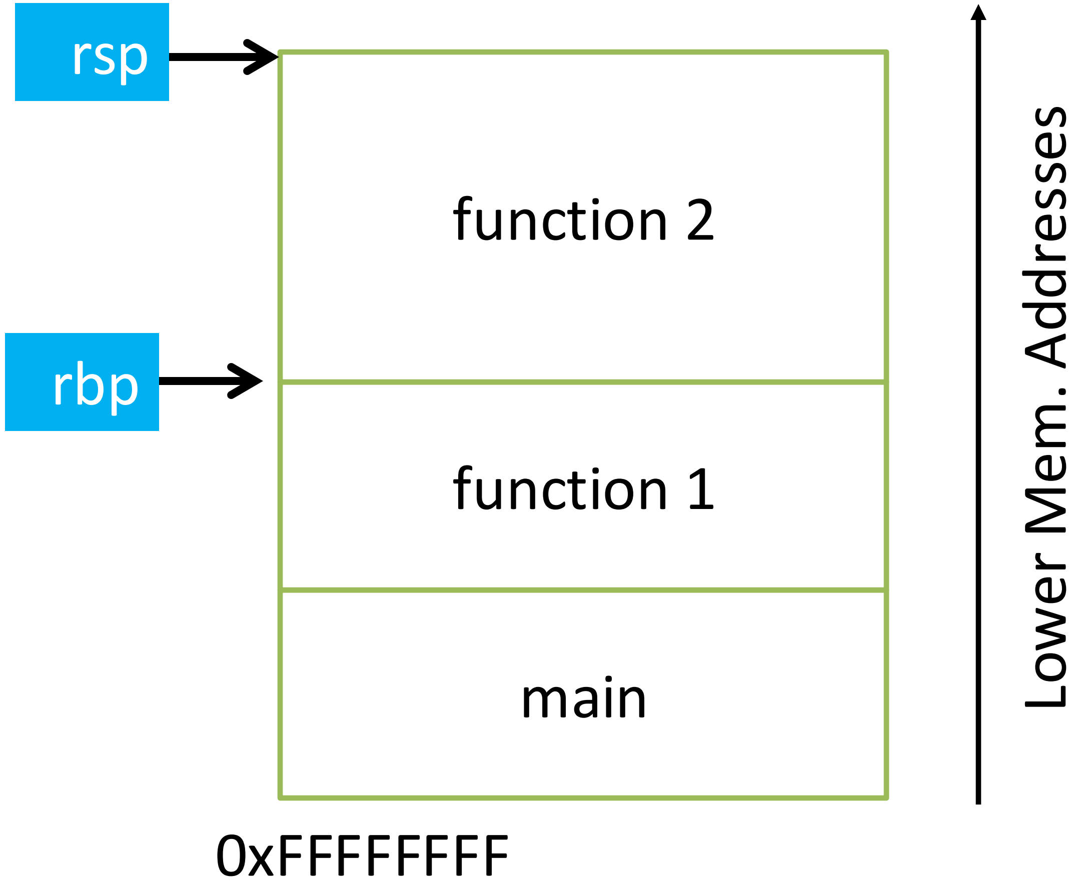


Stack Frame Contents

What needs to be stored in a stack frame? What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries



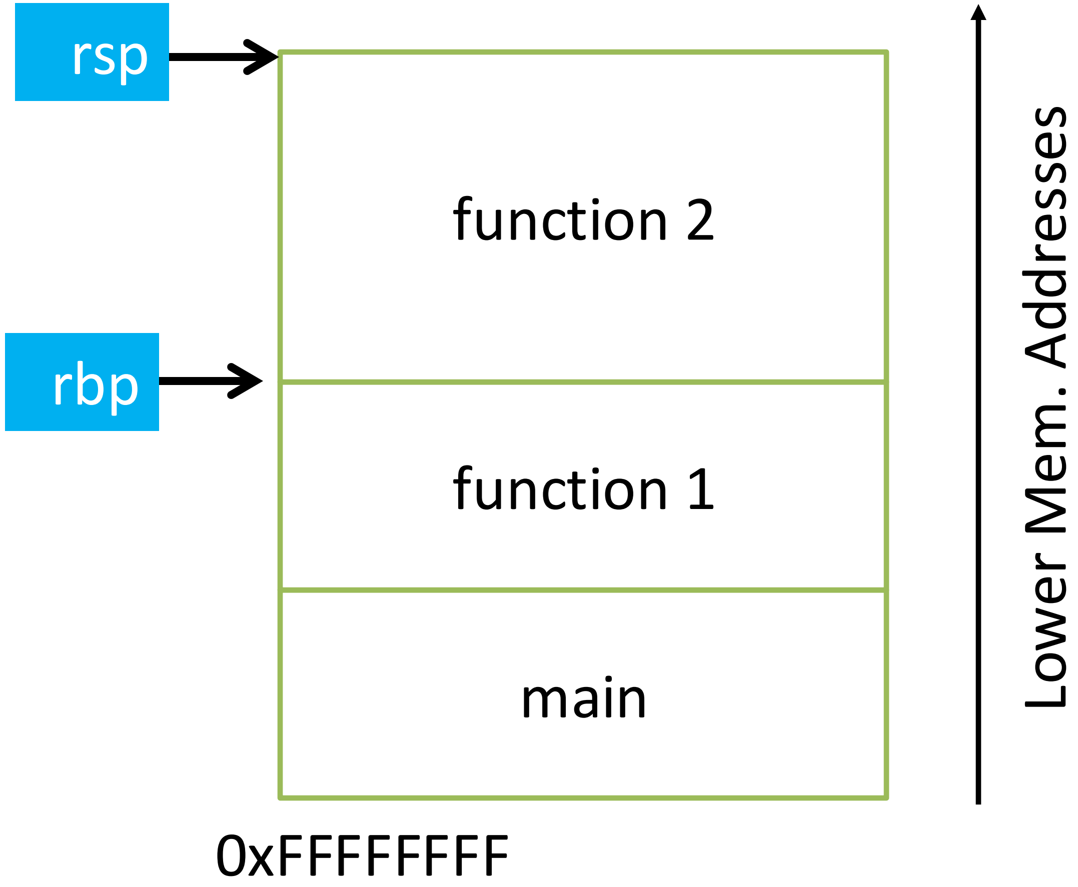
Stack Frame Contents

What needs to be stored in a stack frame?

- Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

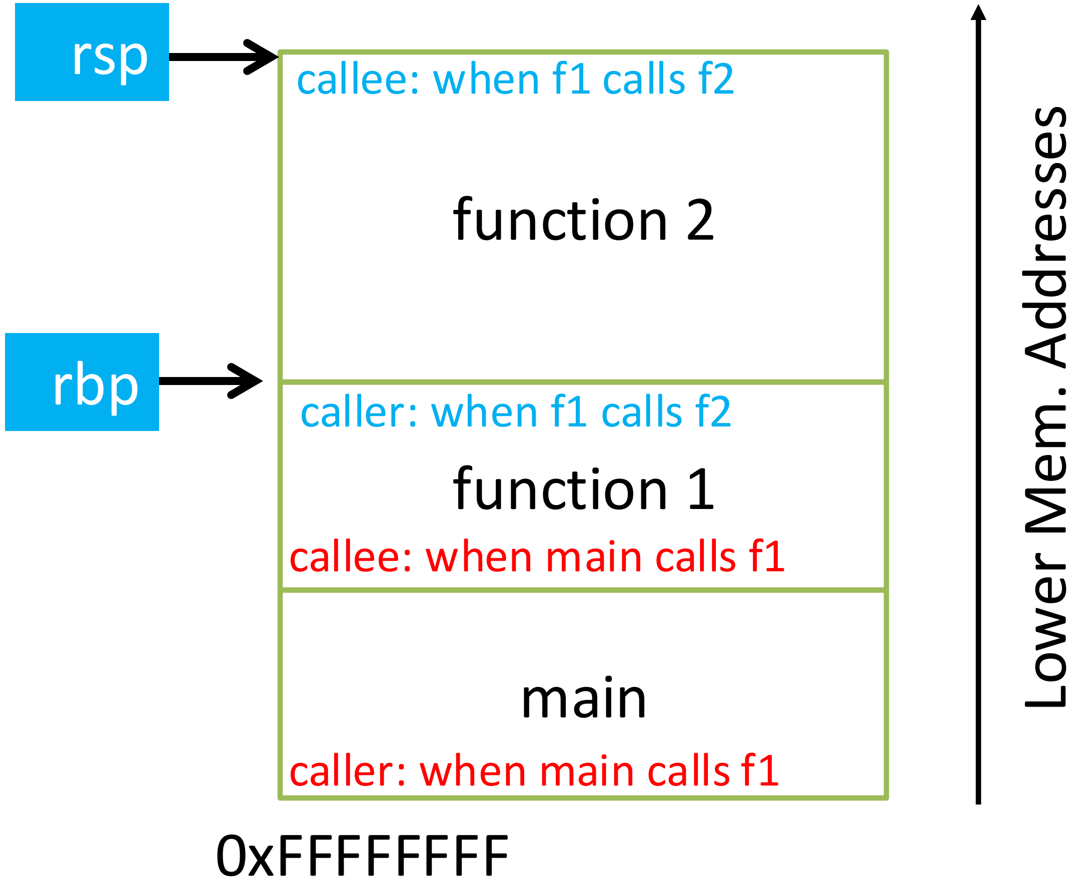
- Saved registers
- Spilled temporaries



Stack Frame Relationships

- If function 1 calls function 2:
  - function 1 is the *caller*
  - function 2 is the *callee*

- With respect to main:
  - main is the *caller*
  - function 1 is the *callee*



Where should we store the following stuff?

```
Previous stack frame base address
Function arguments
Return value
Return address
```

A. In registers
B. On the heap
C. In the caller’s stack frame
D. In the callee’s stack frame
E. Somewhere else

Calling Convention

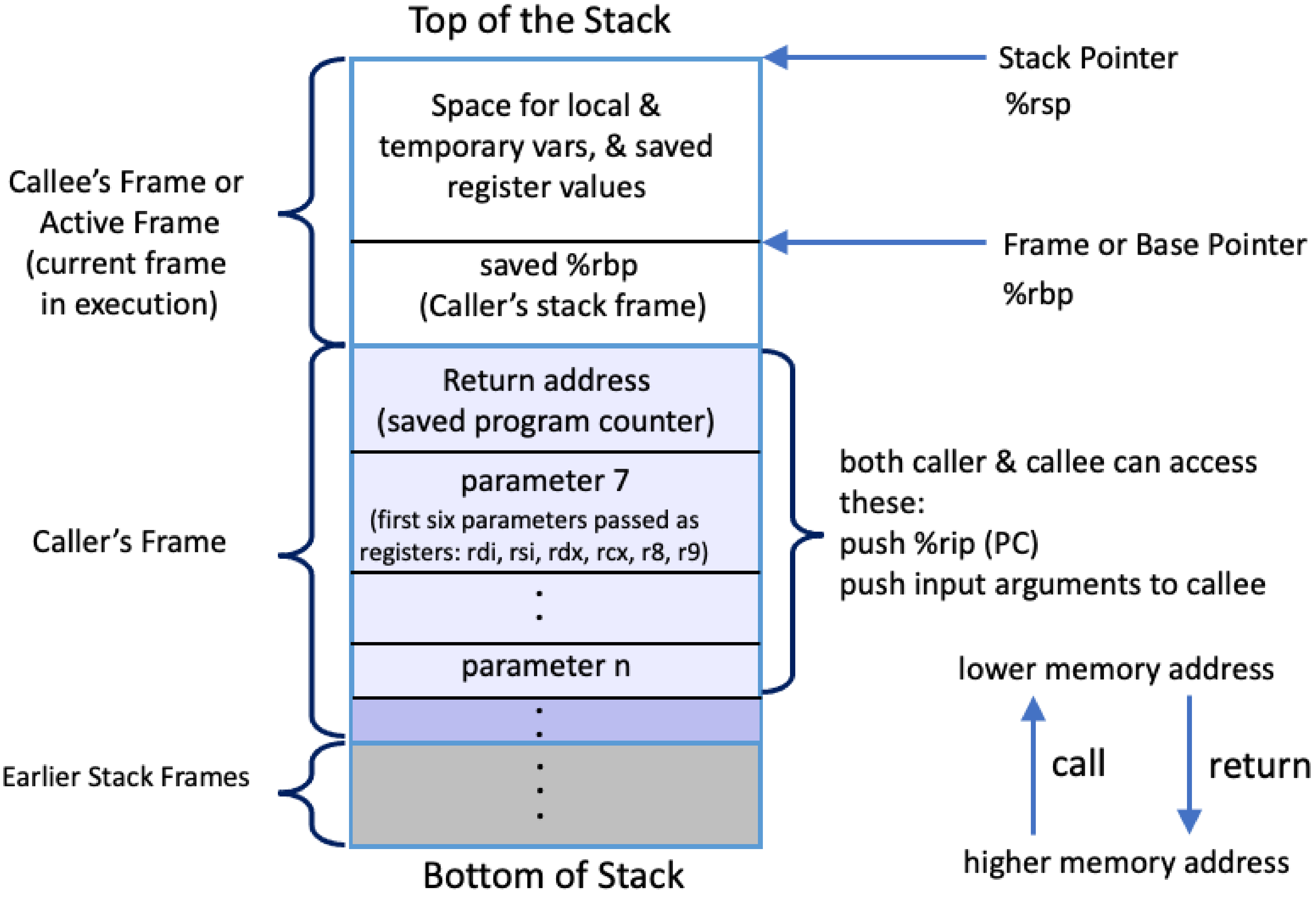
- You could store this stuff wherever you want!
  - The hardware does NOT care.
  - What matters: everyone agrees on where to find the necessary data.

- Calling convention: agreed upon system for exchanging data between caller and callee

- When possible, keep values in registers (why?)
  - Accessing registers is faster than memory (stack)

x86_64 Calling Convention

- The function’s return value: In register %rax
- The caller’s %rbp value (caller’s saved frame pointer)
  - Placed on the stack in the callee’s stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller’s stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller’s stack frame (shared with callee)


Top of the Stack
Stack Pointer
%rsp
Space for local & temporary vars, & saved register values
Callee's Frame or Active Frame (current frame in execution)
Frame or Base Pointer
%rbp
saved %rbp (Caller's stack frame)
Return address (saved program counter)
parameter 7
(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
parameter n
Caller's Frame
both caller & callee can access these:
push %rip (PC)
push input arguments to callee
lower memory address
call
return
higher memory address
Earlier Stack Frames
Bottom of Stack

x86_64 Calling Convention

- The function's return value: In register %rax

- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame

- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame

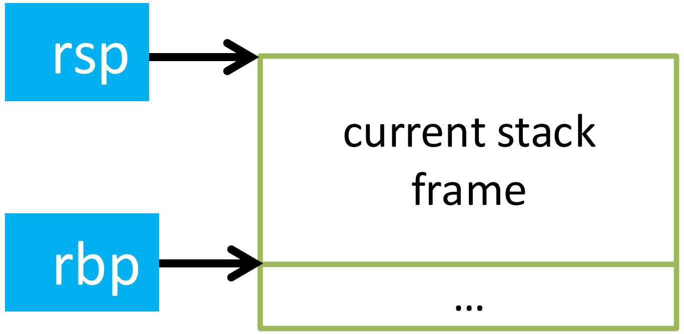
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

Return Value

- If the callee function produces a result, the caller can find it in %rax

- We saw this when we wrote our function in the weekly lab last friday
  - Copy the result to %rax before we finishing up

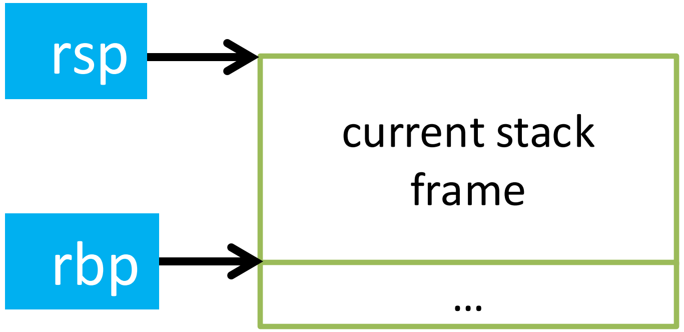
Dynamic Stack Accounting



- Dedicate CPU registers for stack bookkeeping
  - %rsp (stack pointer): Top of current stack frame
  - %rbp (frame pointer): Base of current stack frame
- Compiler maintains these pointers
  - Does the compiler know the exact address they point to?
  - Compiler doesn’t know or care! (job of the OS to figure that out)
- To the compiler: every variable access is relative to %rsp and %rbp!

Compiler: updates to rsp/rbp on function call/return

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

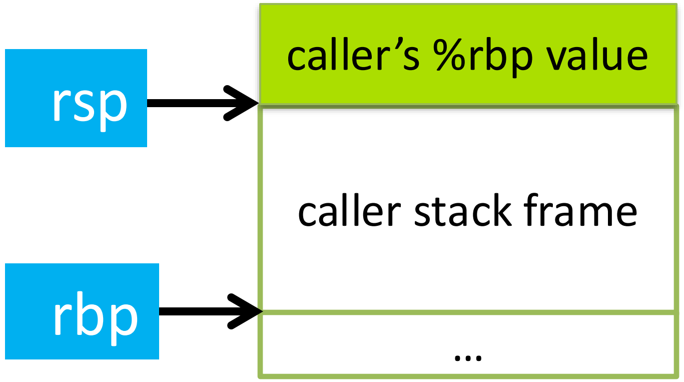


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

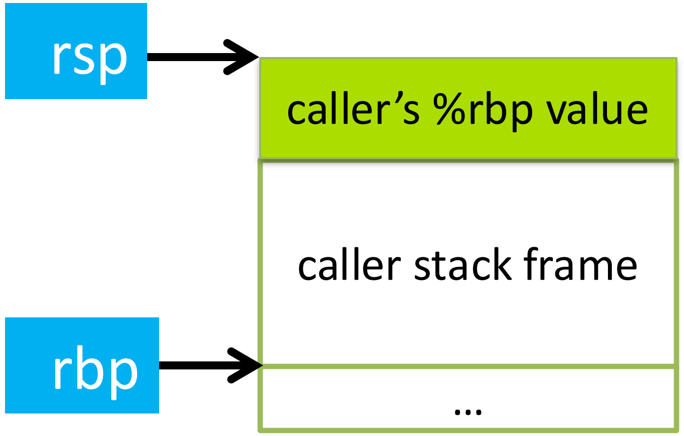


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`

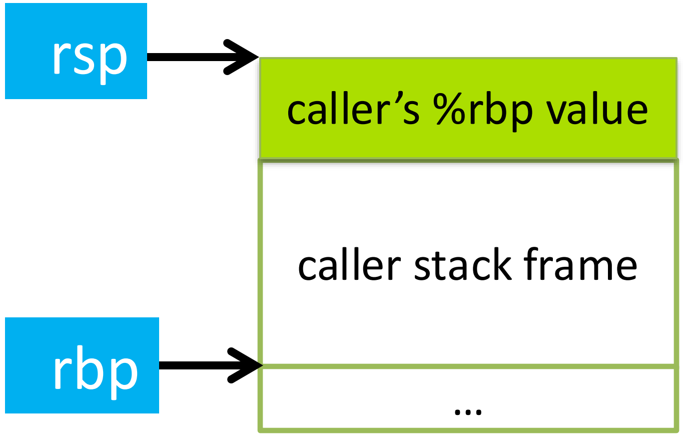


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

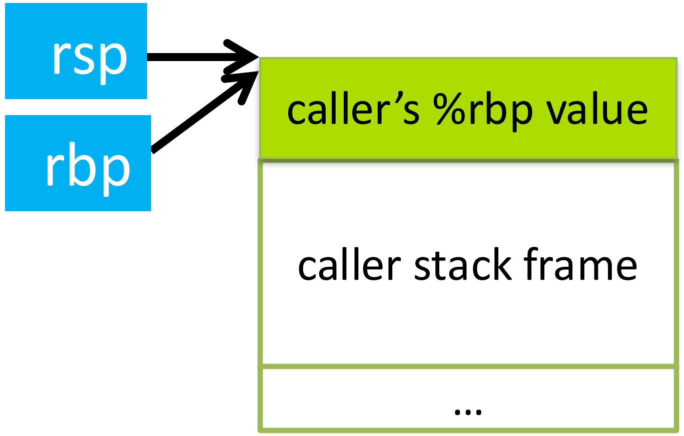


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

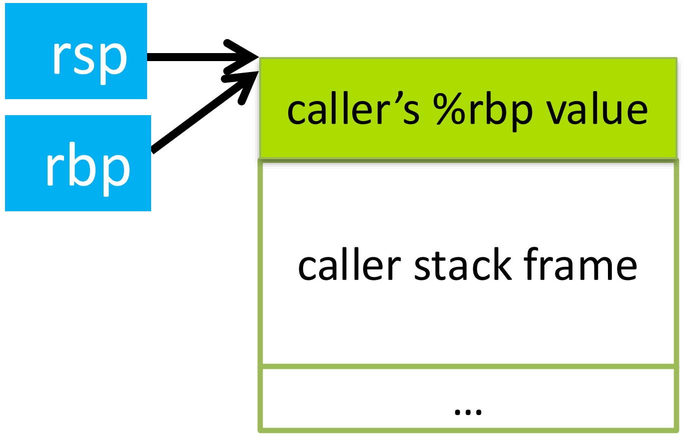


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

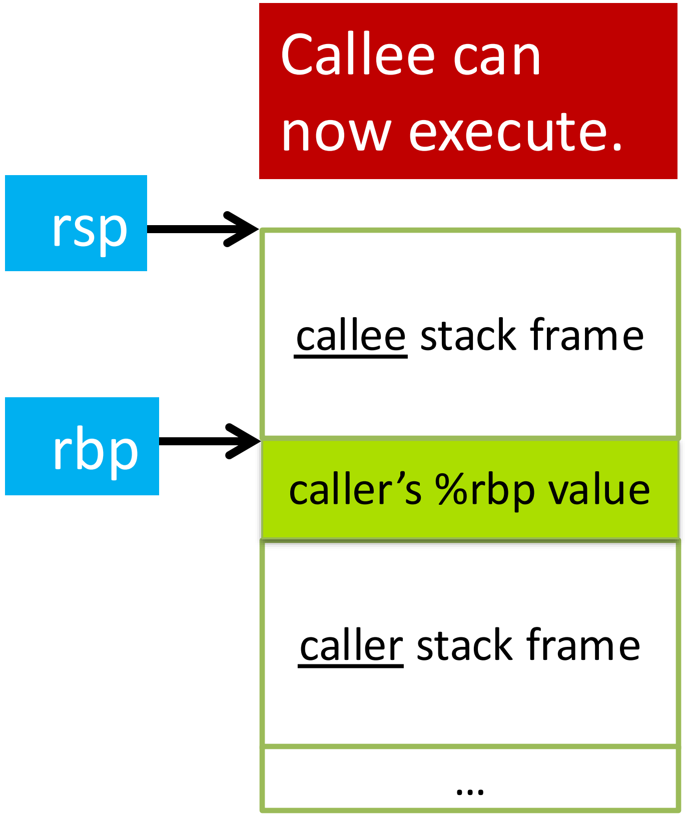


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

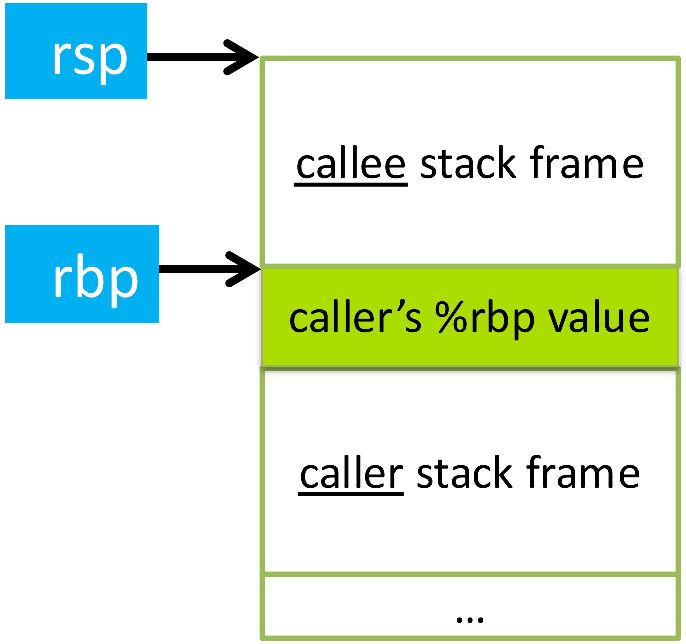


Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp

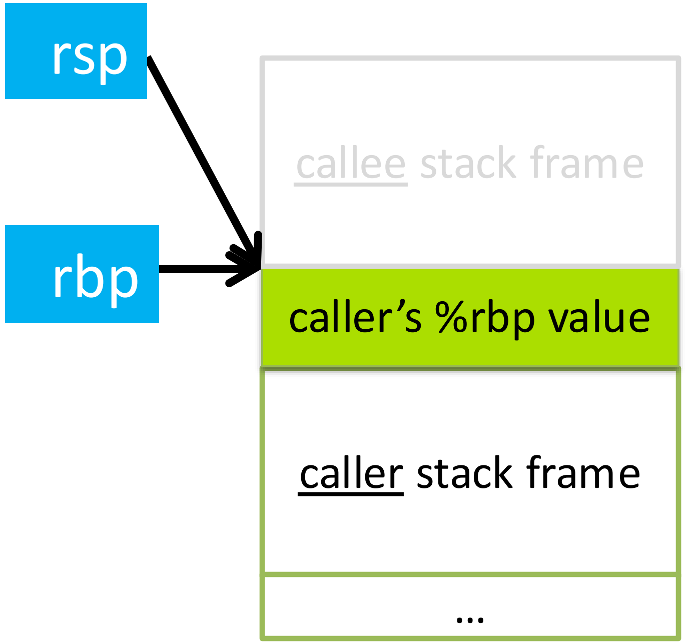
invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)



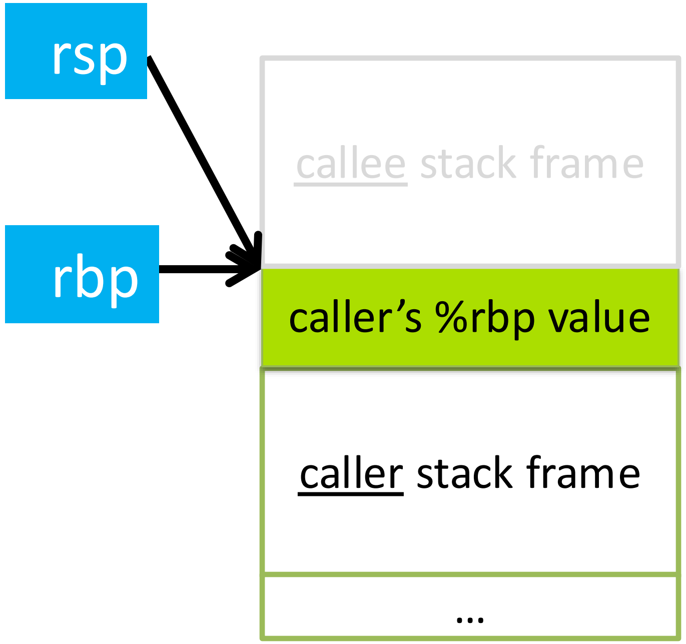
invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)
2. pop %rbp

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



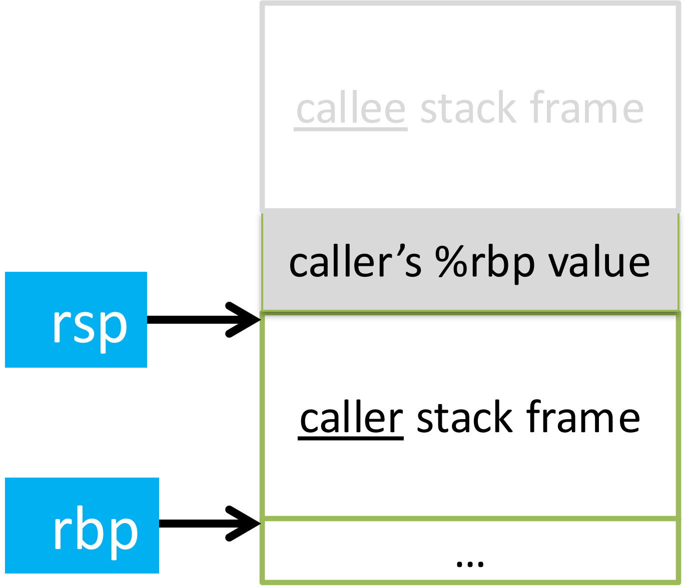
Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp
2. pop %rbp
   - pop caller's rbp off the stack and set it to the value of rbp
   - decrement rsp

X86_64 has another convenience instruction for this: leaveq

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp



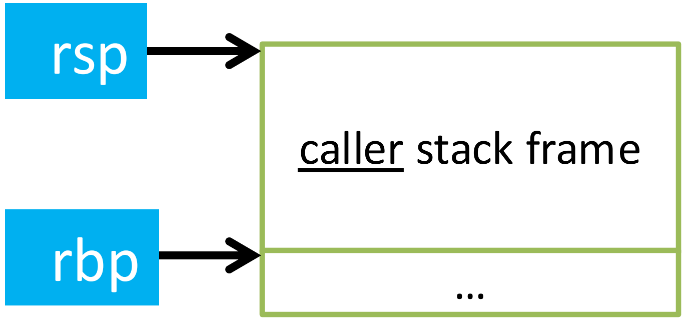
Compiler: Returning from a function call..

Returning from a function:

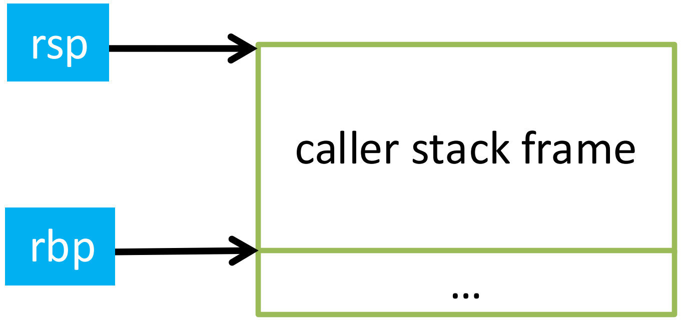
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller's rbp off the stack and set it to the value of rbp
   - decrement rsp

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`

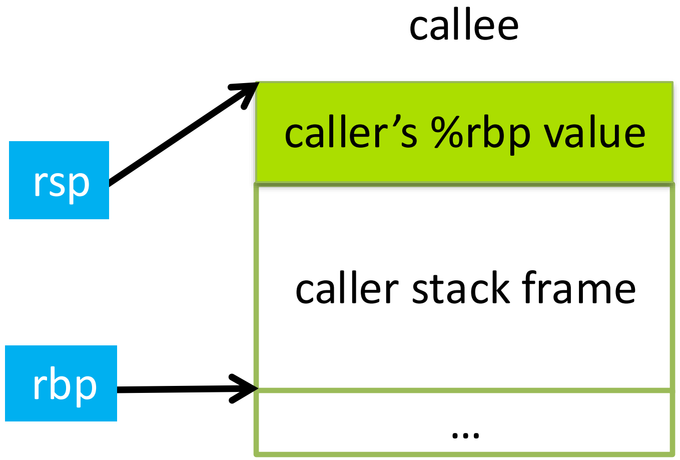
Back to where we started



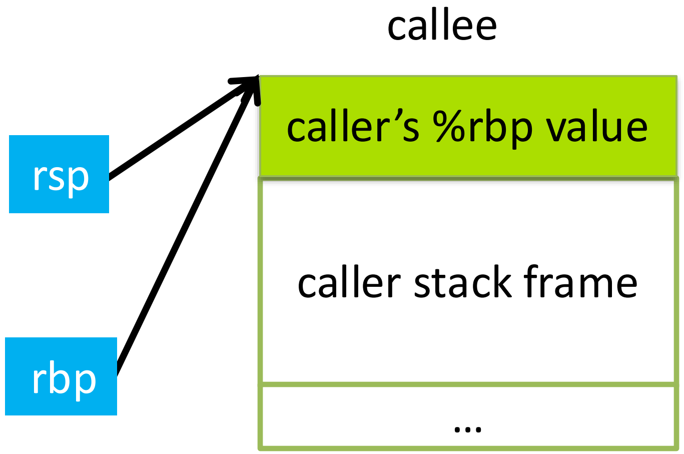
x86 Calling Conventions: Function Call



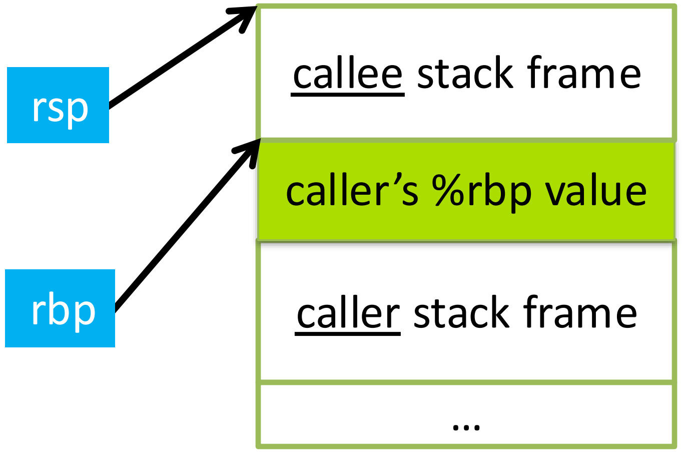
Initial state



push %rbp (store caller's base pointer)

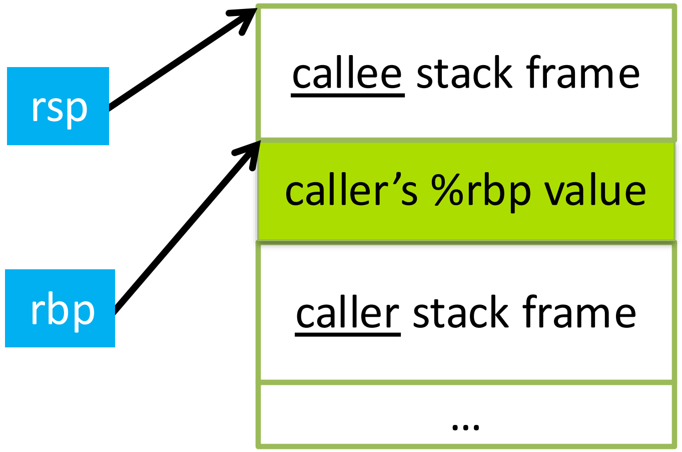


mov %rsp, %rbp
(establish callee's frame pointer)



sub $SIZE, %rsp
(allocate space for callee's locals)

x86 Calling Conventions: Function Return


rsp

rbp

callee stack frame

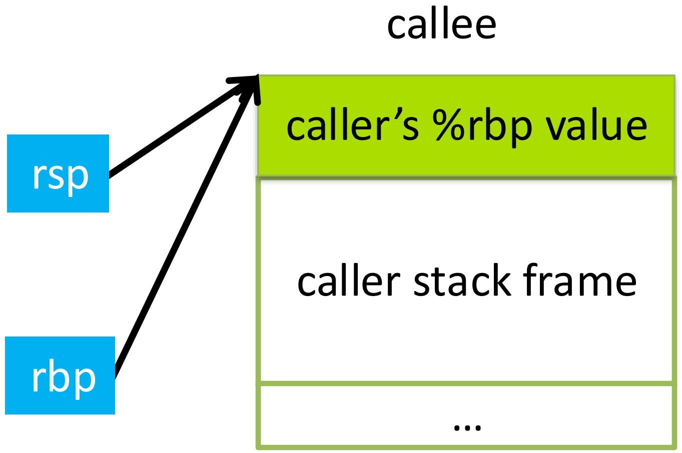
caller's %rbp value

caller stack frame

…

we want to restore the caller's frame

x86_64 provides a convenience instruction that does all of this:
`leaveq`


callee

rsp

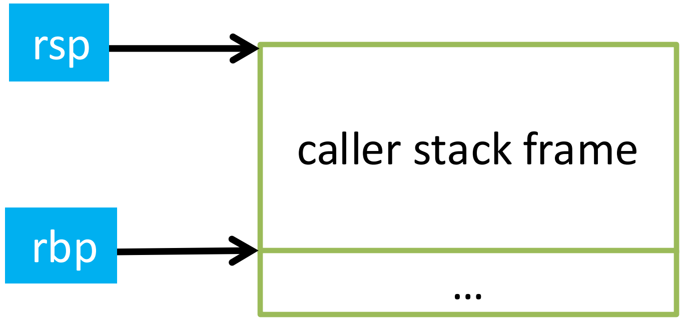
rbp

caller's %rbp value

caller stack frame

…

mov %rbp, %rsp
(restore caller's stack pointer)


rsp

rbp

caller stack frame

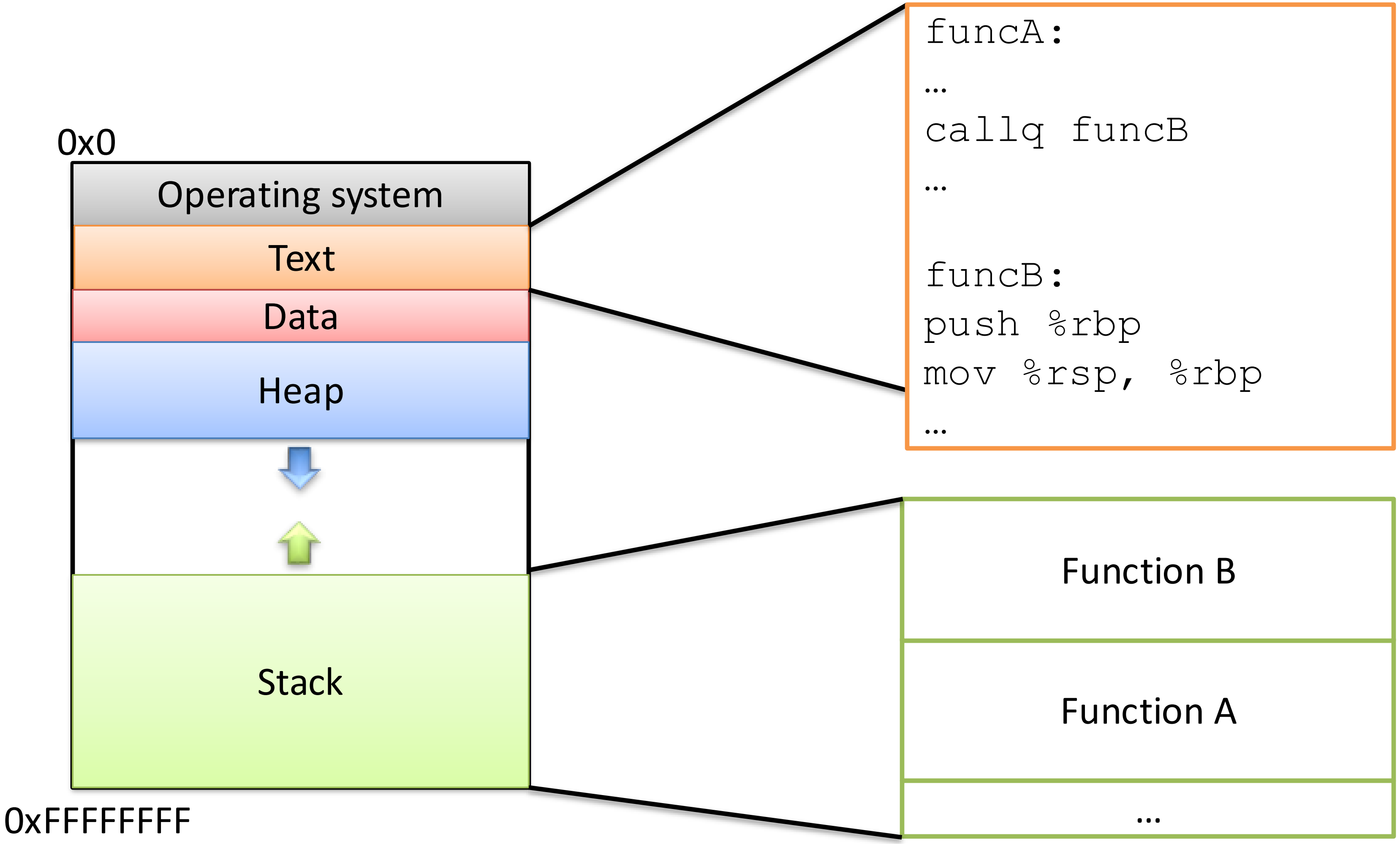
…

pop %rbp (restore caller's frame pointer)

x86_64 Calling Convention

- The function's return value:
  - In register %rax
- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

Instructions in Memory



0x0

Operating system

Text

Data

Heap

Stack

0xFFFFFFFF

```
funcA:
…
callq funcB
…

funcB:
push %rbp
mov %rsp, %rbp
…
```

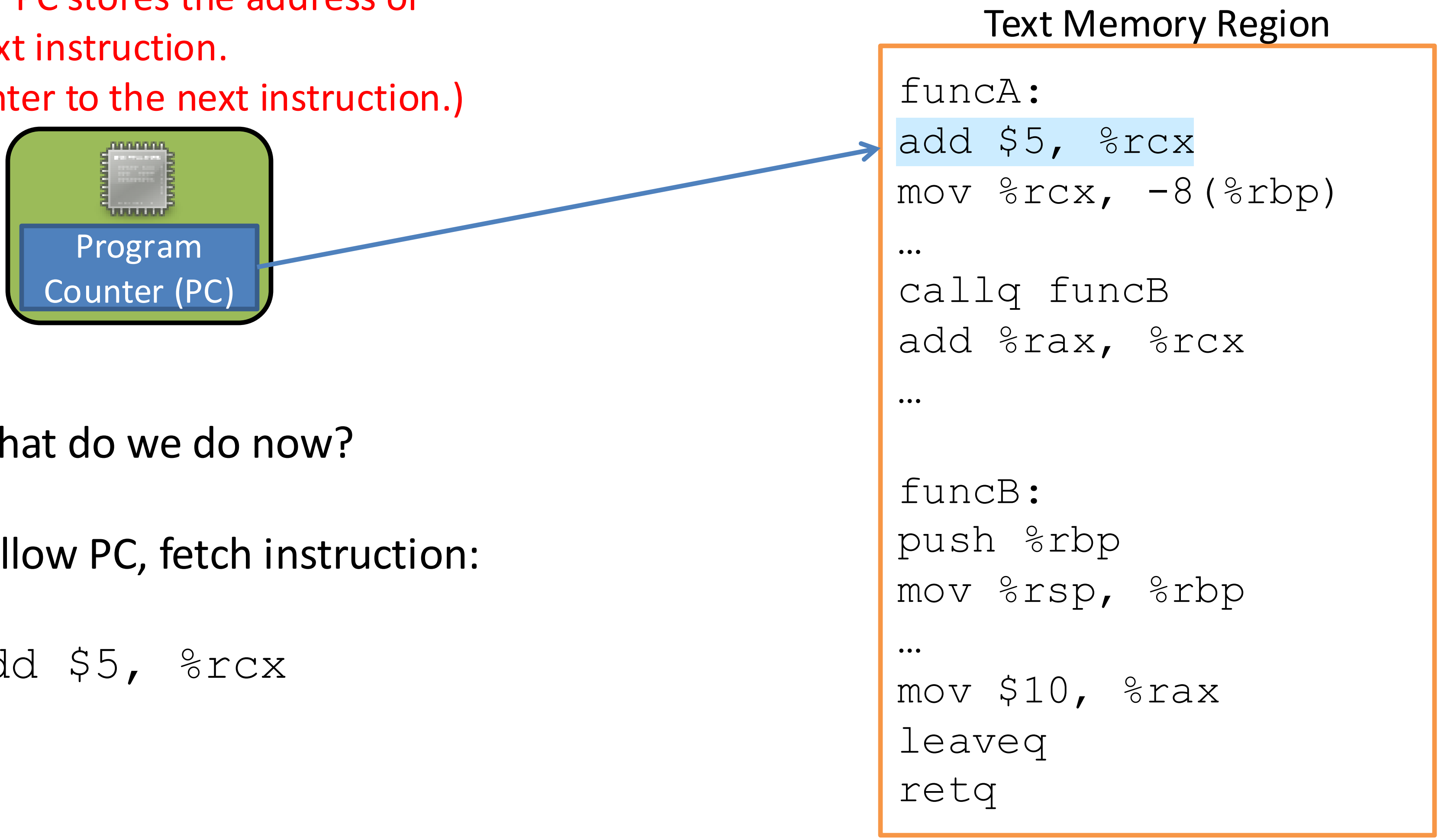
Function B

Function A

…

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

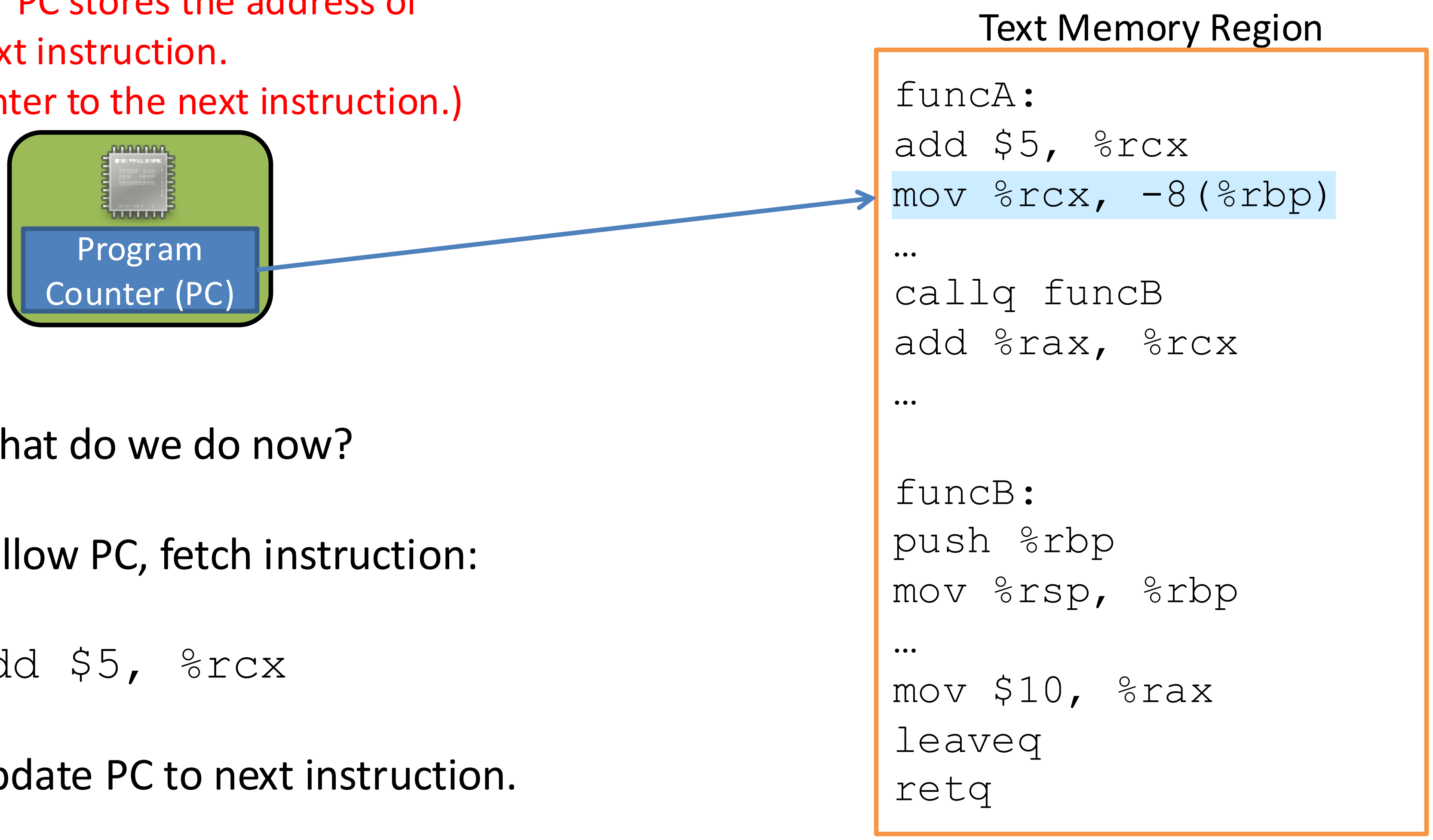
What do we do now?

Follow PC, fetch instruction:

```
add $5, %rcx
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Follow PC, fetch instruction:

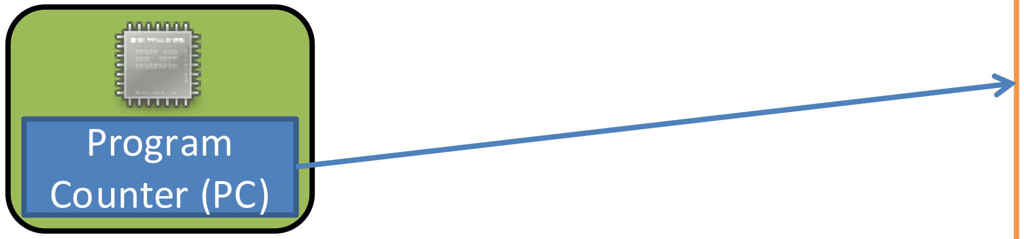
`add $5, %rcx`

Update PC to next instruction.

Execute the `addl`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

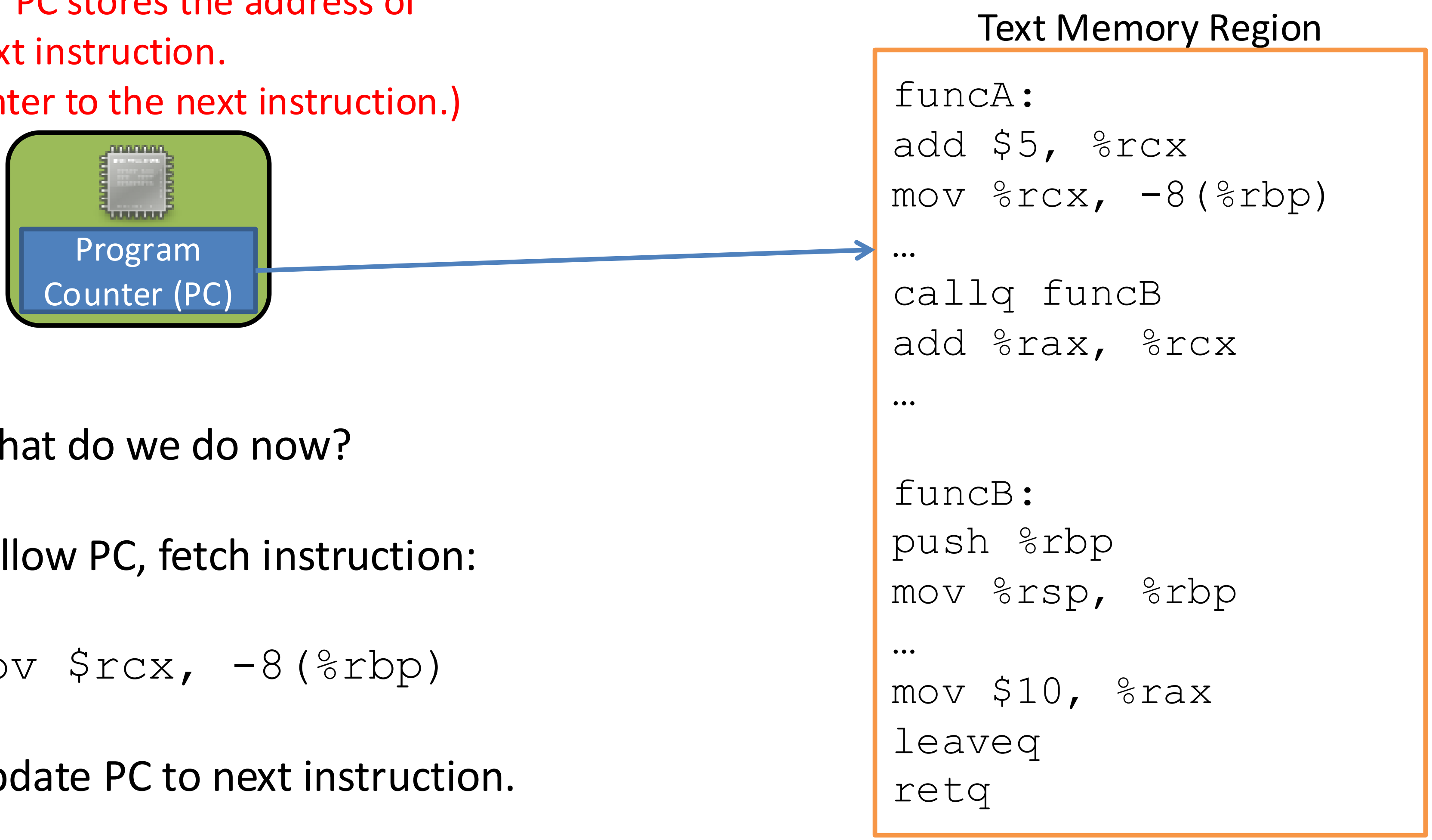
What do we do now?

Follow PC, fetch instruction:

```
mov $rcx, -8(%rbp)
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Follow PC, fetch instruction:

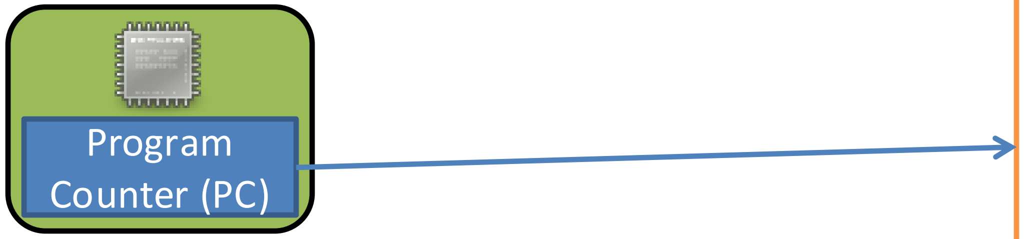
`mov $rcx, -8(%rbp)`

Update PC to next instruction.

Execute the `mov`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

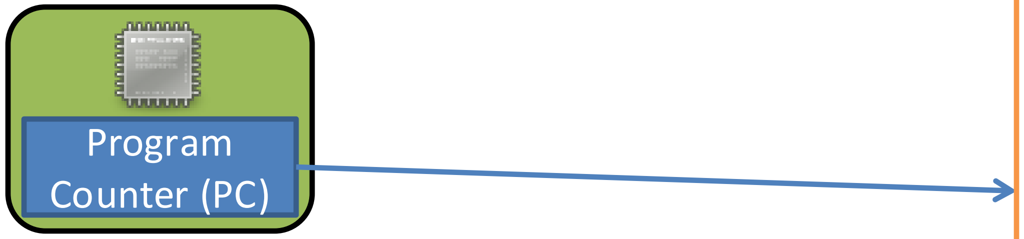
Changing the PC: Jump

- On a (non-function call) jump:
  - Check condition codes
  - Set PC to execute elsewhere (usually not the next instruction)

- Do we ever need to go back to the instruction after the jump?

Maybe (and if so, we'd have a label to jump back to), but usually not.

Changing the PC: Functions

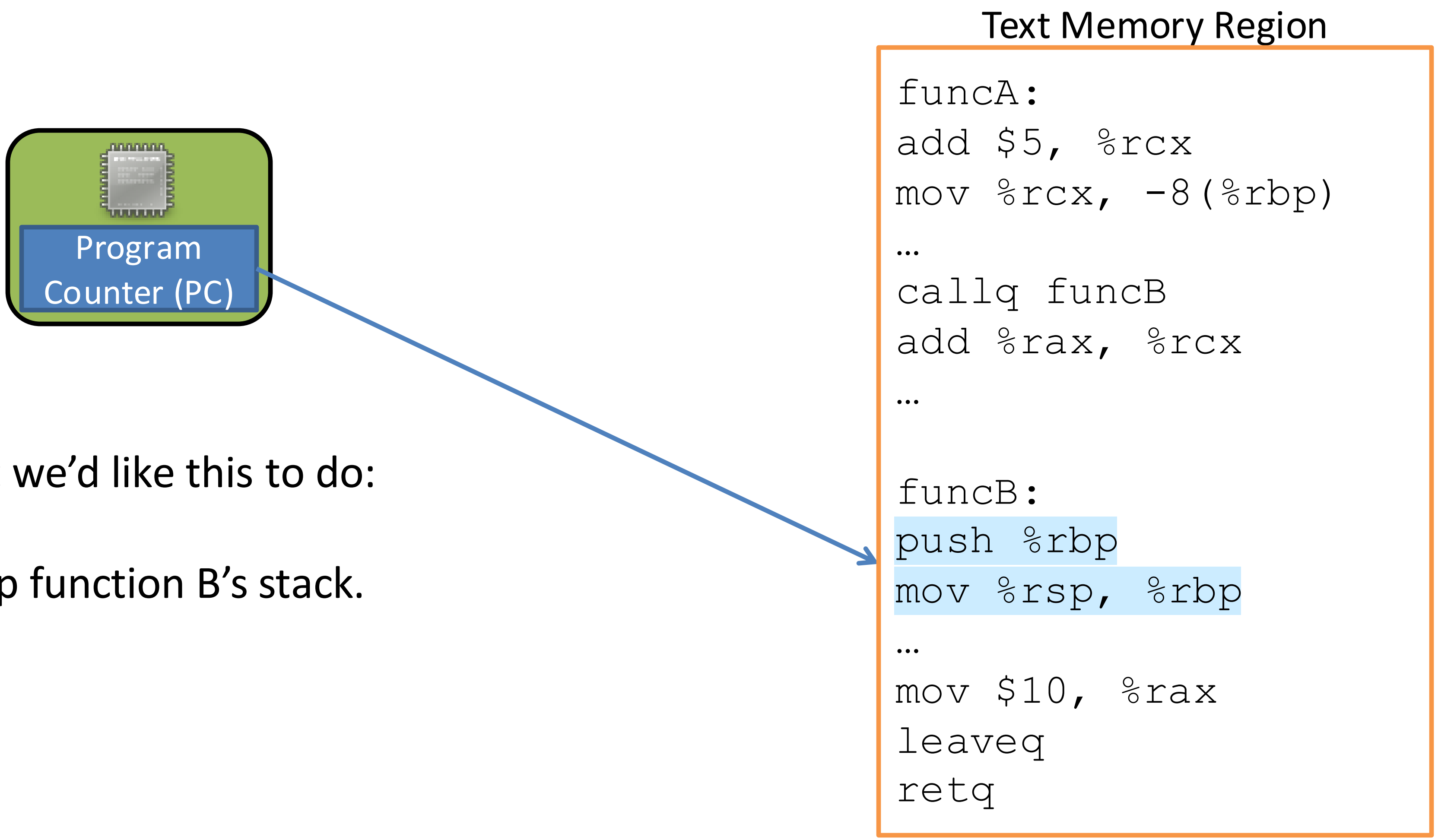


Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What we’d like this to do:

Changing the PC: Functions



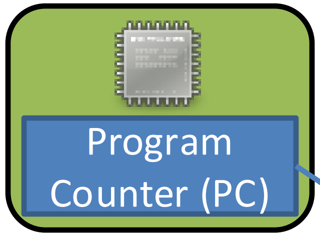
What we'd like this to do:

Set up function B's stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



What we'd like this to do:

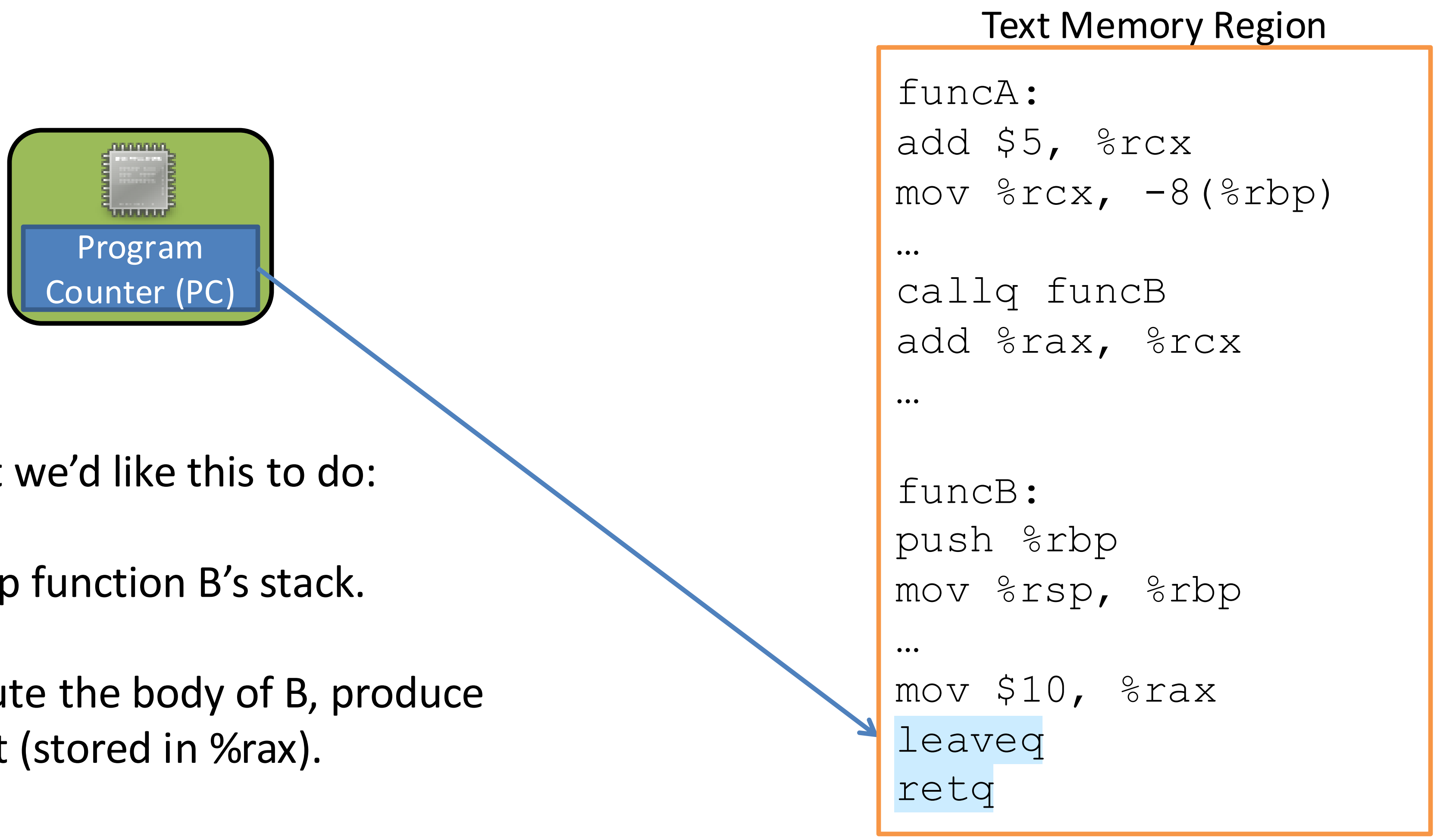
Set up function B's stack.

Execute the body of B, produce result (stored in %rax).

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

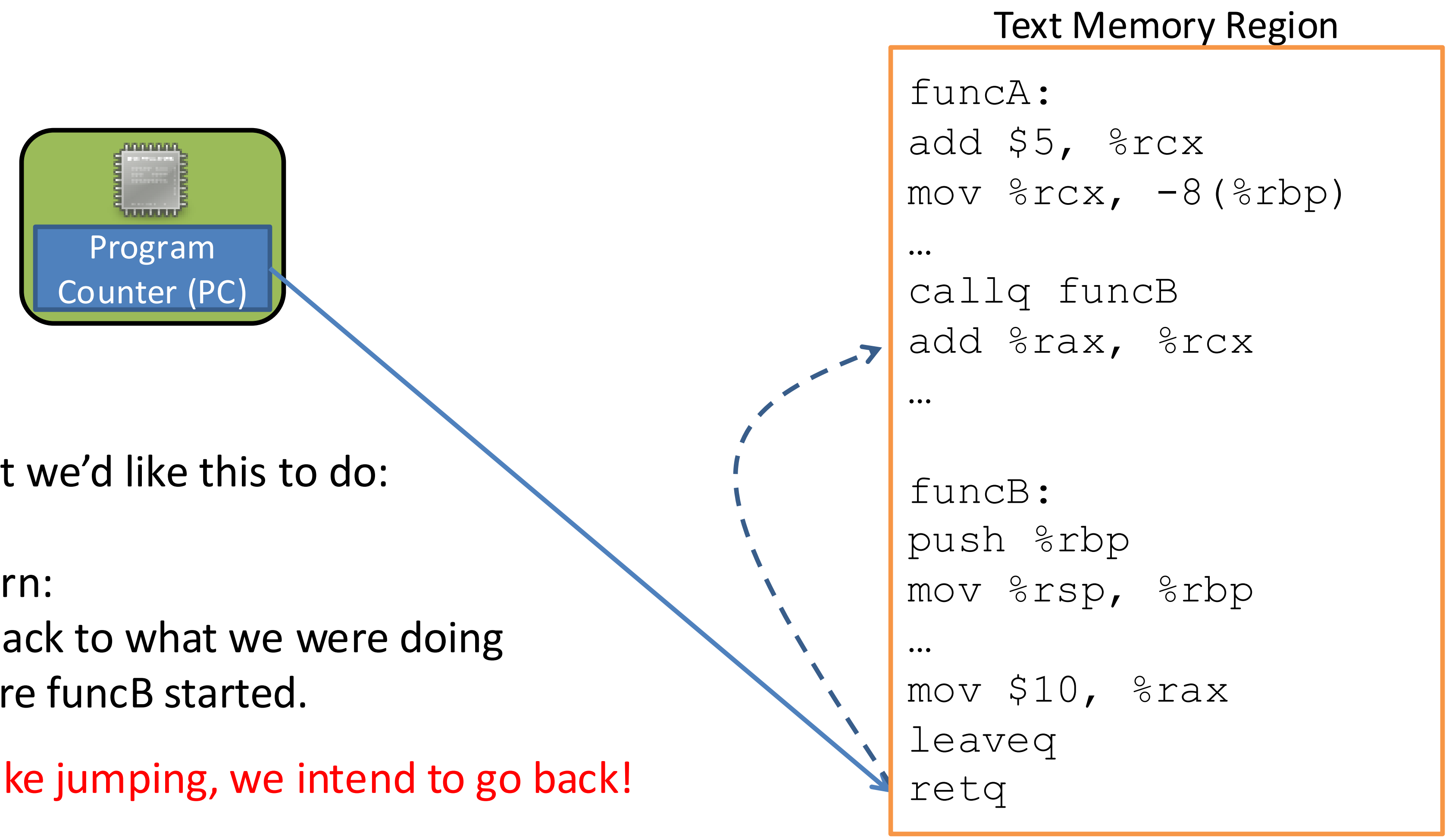
What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %rax).

Restore function A’s stack.

Changing the PC: Functions



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What we'd like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!

Like `push`, `pop`, and `leave`, `call` and `ret` are convenience instructions. What should they do to support the PC-changing behavior we need? (The PC is %rip.)

call

In words:

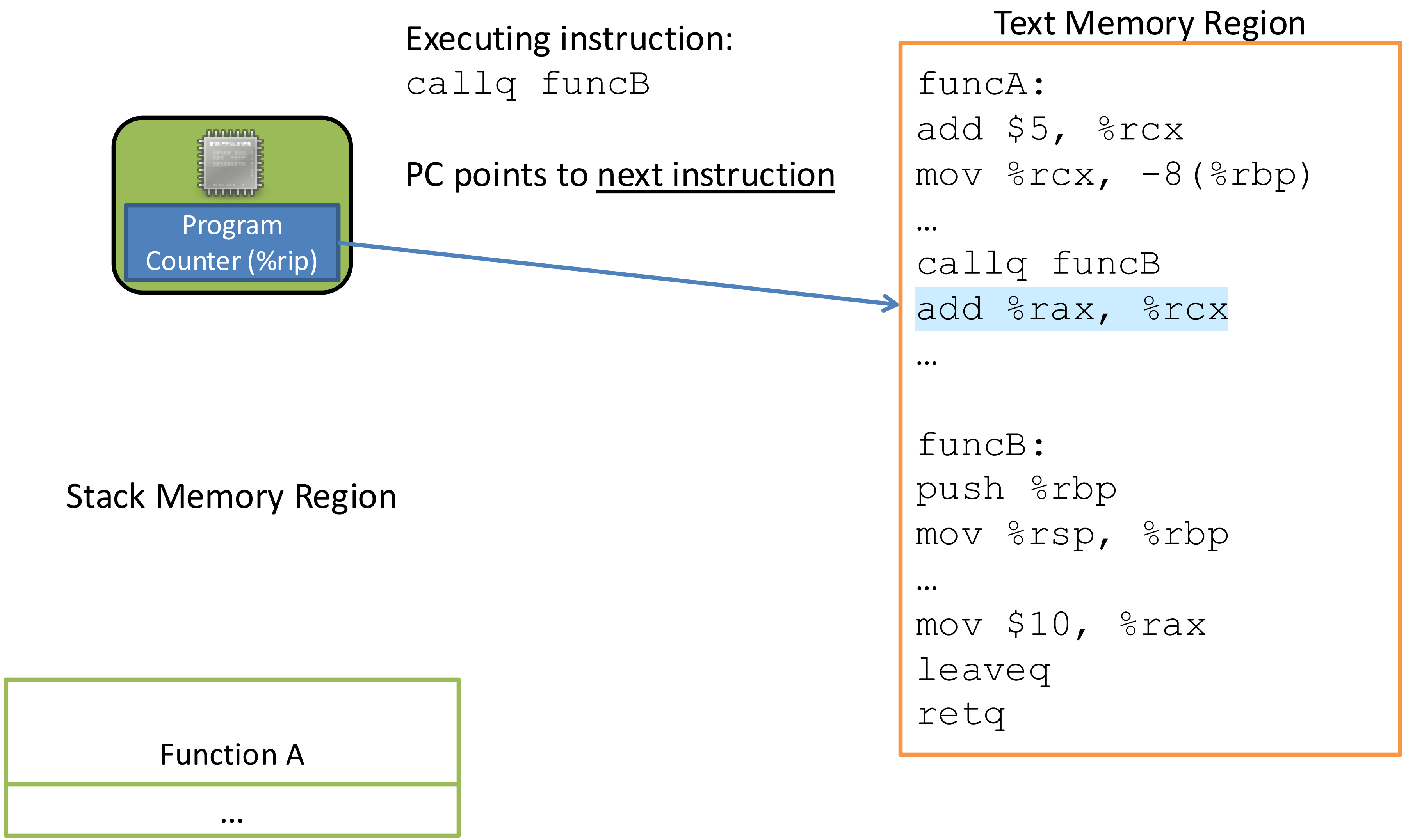
In instructions:

ret

In words:

In instructions:

Functions and the Stack



Executing instruction:
`callq funcB`

PC points to <u>next instruction</u>

Text Memory Region

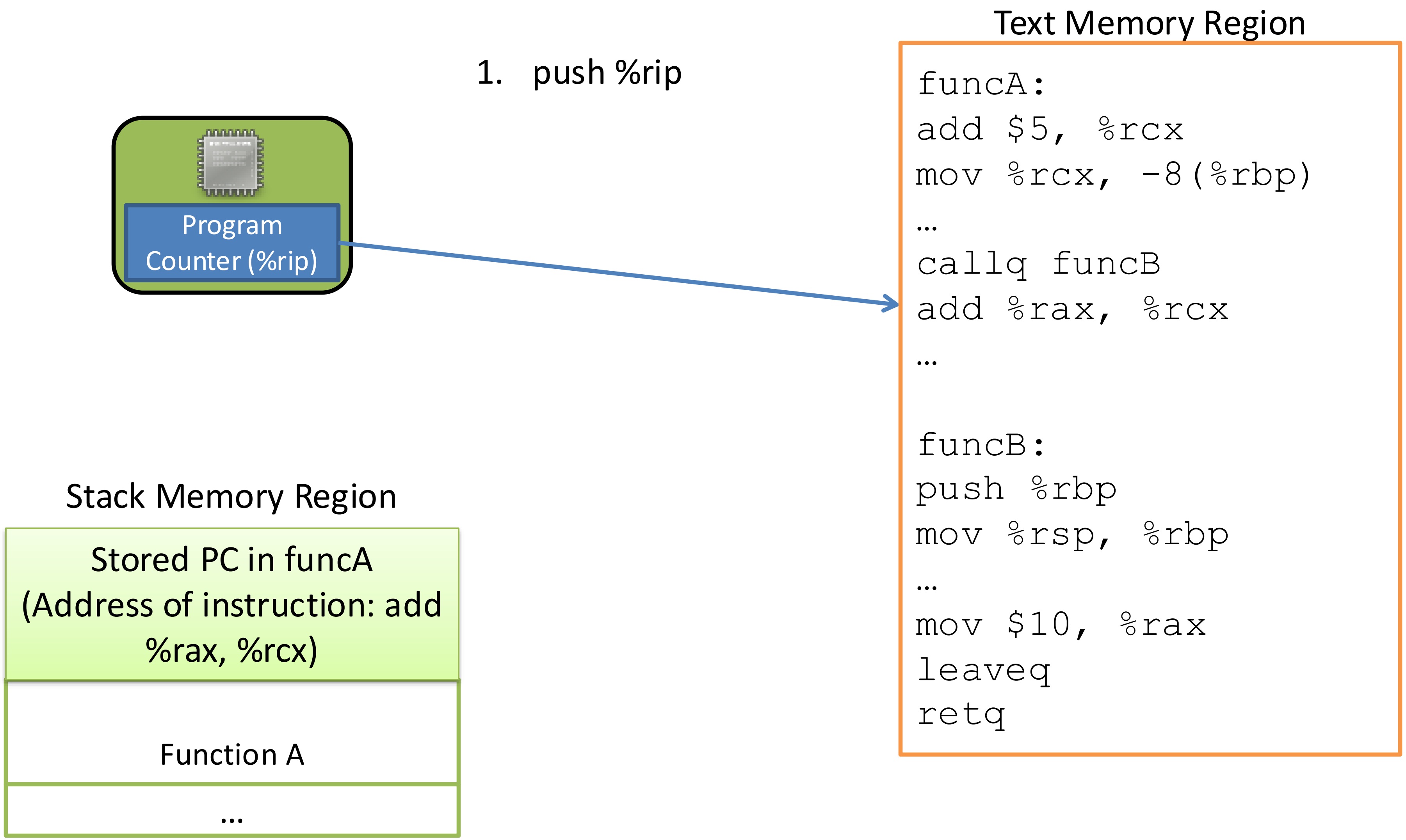
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

Function A
…

Functions and the Stack

1. push %rip



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

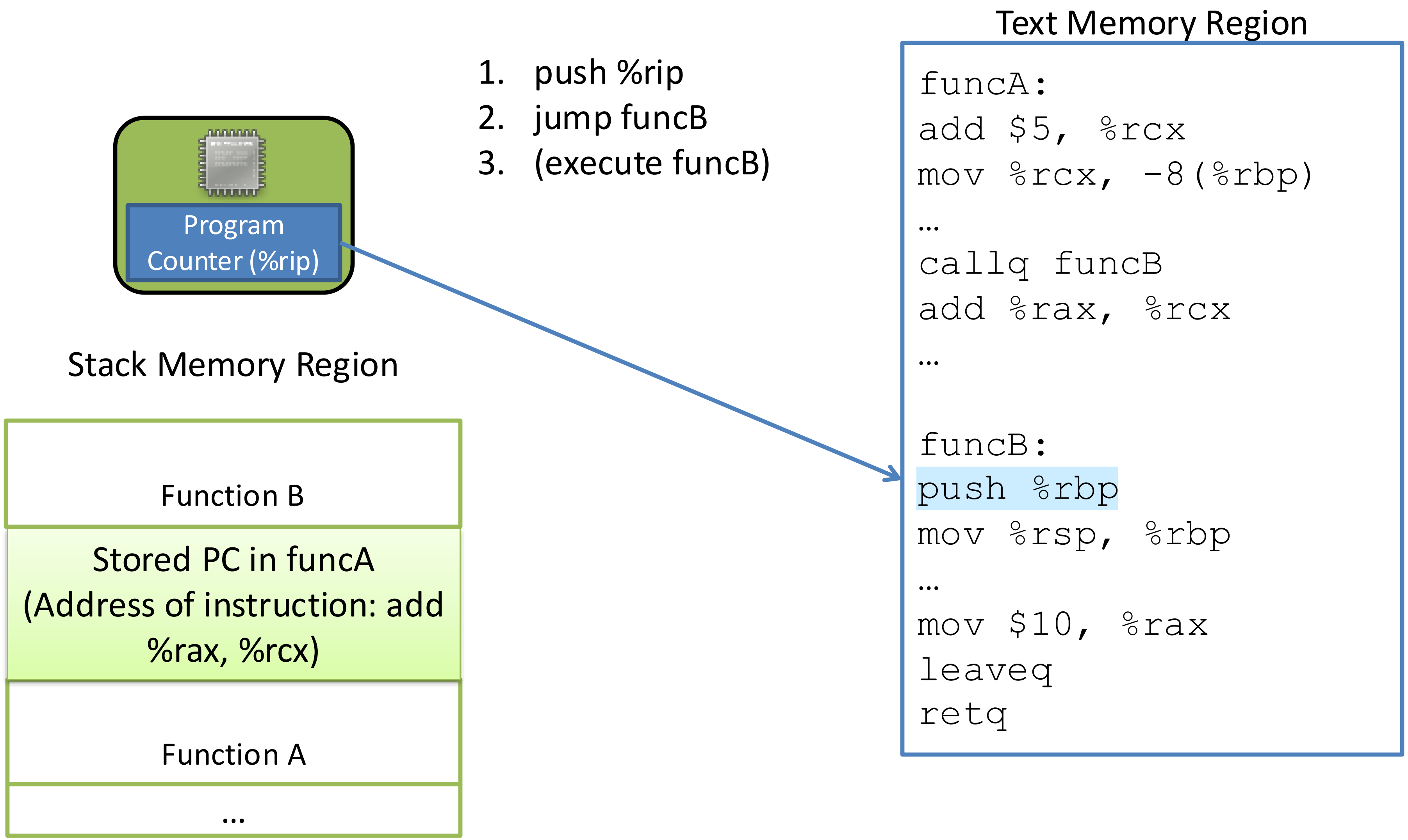
Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

…

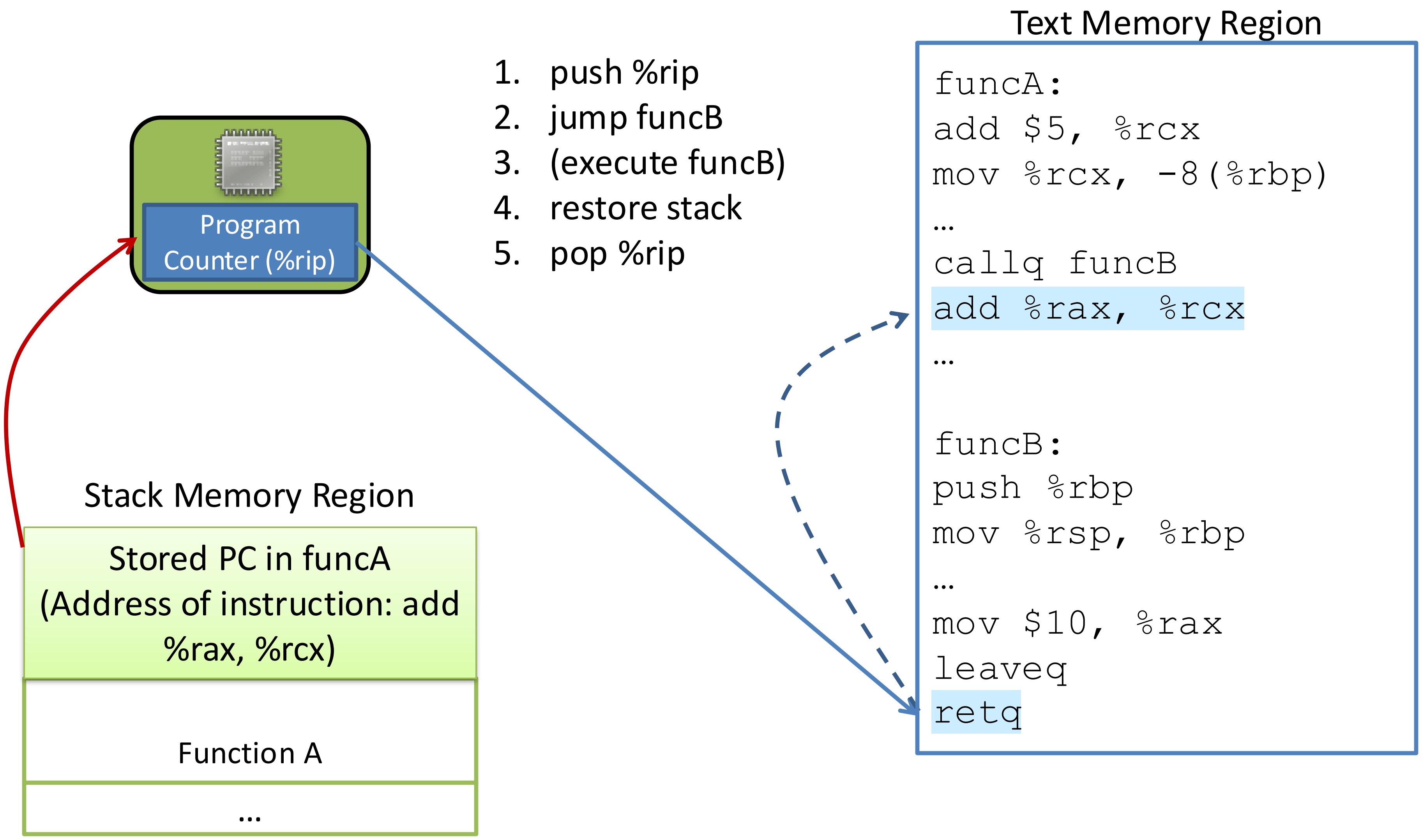
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)



```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Functions and the Stack



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip

Program Counter (%rip)

Stack Memory Region

Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

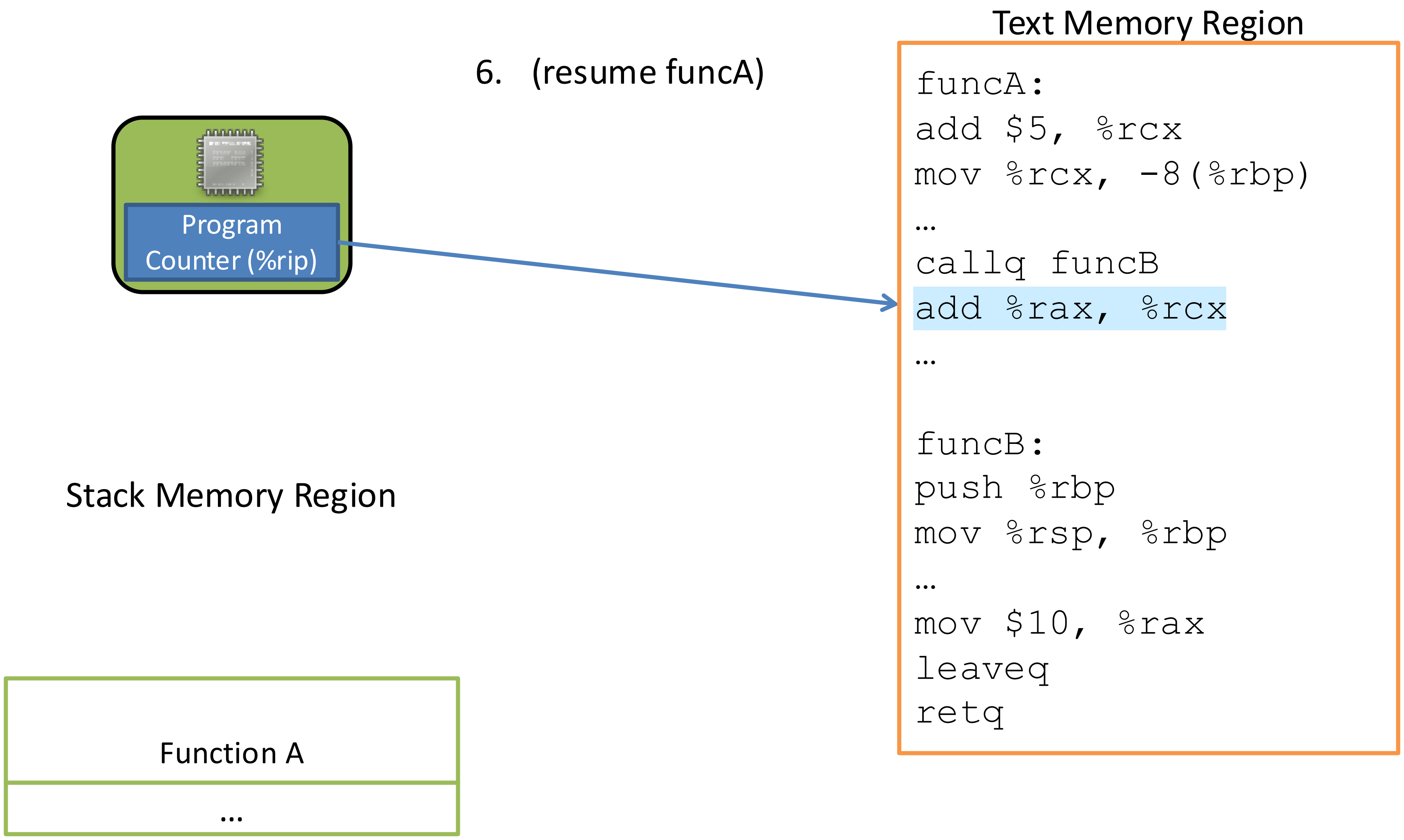
…

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

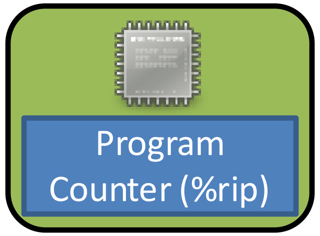
Functions and the Stack

6. (resume funcA)



```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Recap: PC upon a Function Call



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

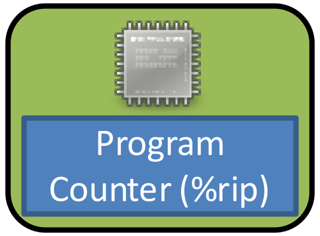
Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

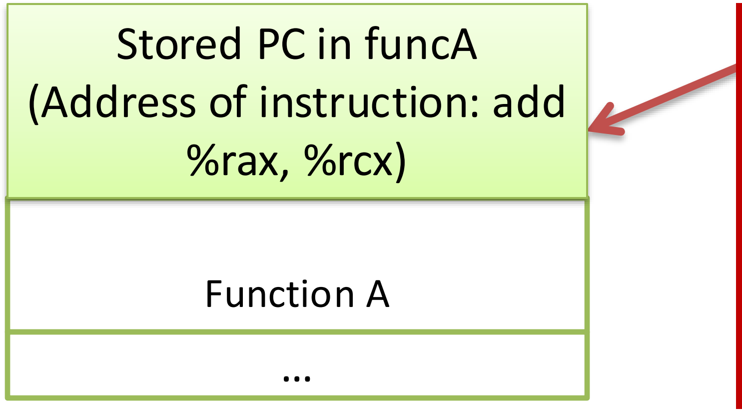
Stored PC in funcA (Address of instruction: add %rax, %rcx)
Function A
…

Functions and the Stack



1. push %rip ⎫ callq
2. jump funcB ⎭
3. (execute funcB)
4. restore stack } leaveq
5. pop %rip } retq
6. (resume funcA)

Stack Memory Region



Return address:

Address of the instruction we should jump back to when we finish (return from) the currently executing function.

x86_64 Stack / Function Call Instructions

`push`	Create space on the stack and place the source there.	`sub $8, %rsp` `mov src, (%rsp)`
`pop`	Remove the top item off the stack and store it at the destination.	`mov (%rsp), dst` `add $8, %rsp`
`callq`	1. Push return address on stack 2. Jump to start of function	`push %rip` `jmp target`
`leaveq`	Prepare the stack for return (restoring caller's stack frame)	`mov %rbp, %rsp` `pop %rbp`
`retq`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	`pop %rip`

x86_64 Calling Convention

- The function’s return value:
  - In register %rax

- The caller’s %rbp value (caller’s saved frame pointer)
  - Placed on the stack in the callee’s stack frame

- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller’s stack frame

- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller’s stack frame (shared with callee)

Function Arguments

- Most functions don’t receive more than 6 arguments, so x86_64 can simply use registers most of the time.
- If we *do* have more than 6 arguments though (e.g., perhaps a `printf` with lots of placeholders), we can’t fit them all in registers.
- In that case, we need to store the extra arguments on the stack.
  By convention, they go in the caller’s stack frame.

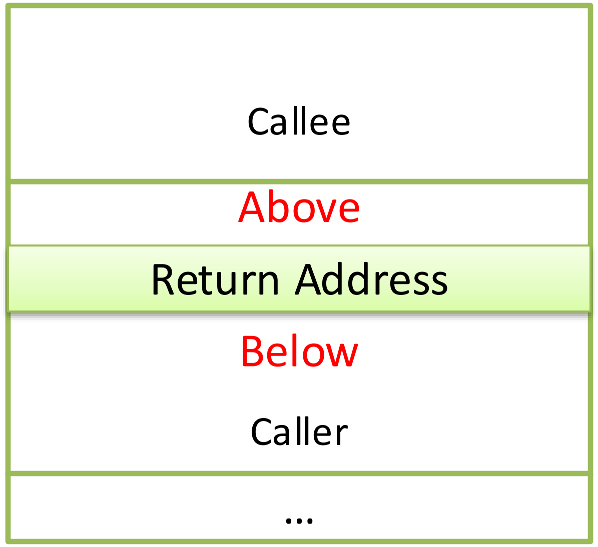
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



Callee

Above

Return Address

Below

Caller

…

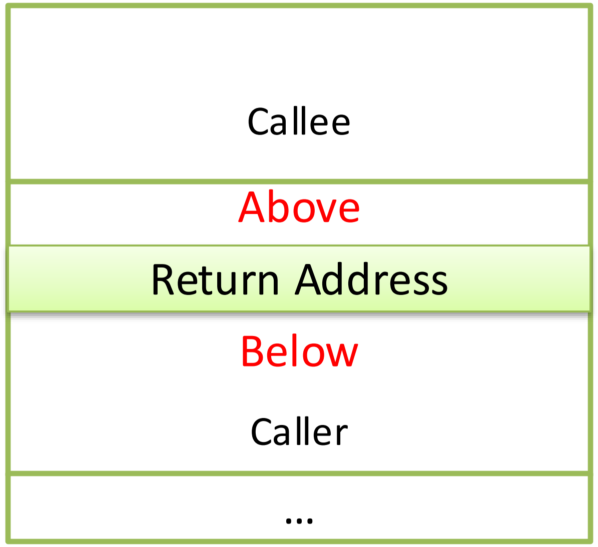
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



x86_64 Stack / Function Call Instructions

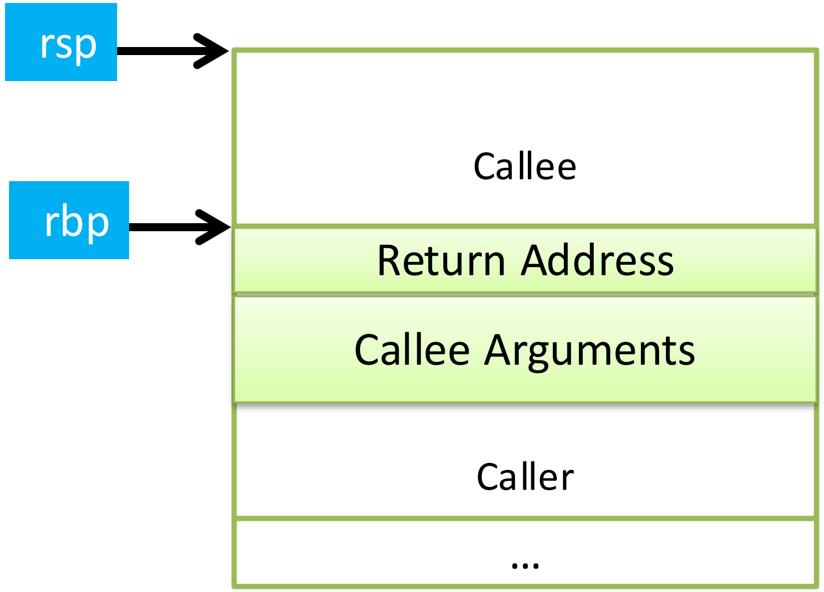
`push`	Create space on the stack and place the source there.	`sub $8, %rsp` `mov src, (%rsp)`
`pop`	Remove the top item off the stack and store it at the destination.	`mov (%rsp), dst` `add $8, %rsp`
`callq`	1. Push return address on stack 2. Jump to start of function	`push %rip` `jmp target`
`leaveq`	Prepare the stack for return (restoring caller’s stack frame)	`mov %rbp, %rsp` `pop %rbp`
`retq`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	`pop %rip`

Arguments

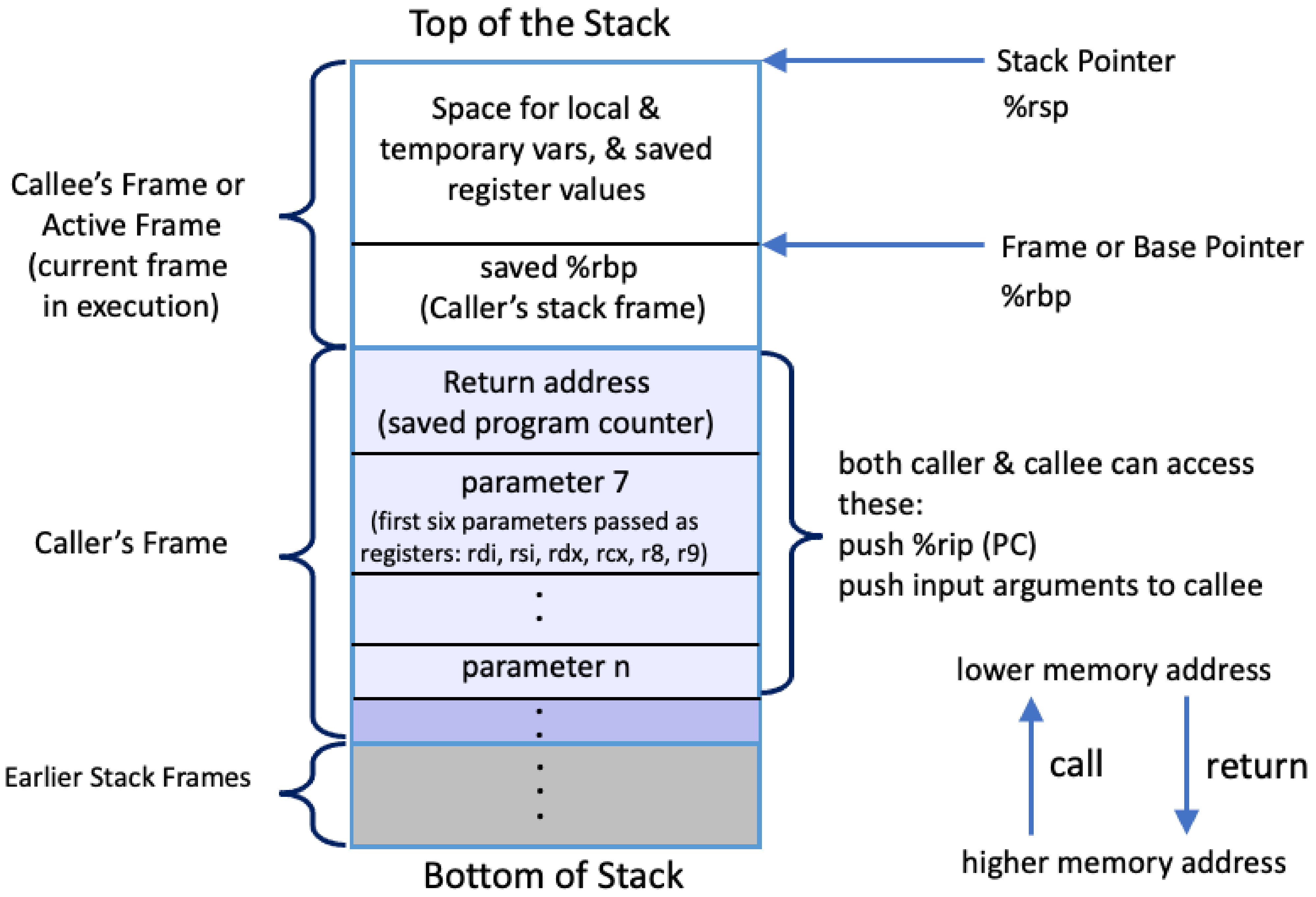
- Extra arguments to the callee are stored just underneath the return address.

- Does it matter what order we store the arguments in?

- Not really, as long as we're consistent (follow conventions).



This is why arguments can be found at positive offsets relative to %rbp.



Top of the Stack

Callee's Frame or Active Frame (current frame in execution)

Space for local & temporary vars, & saved register values

Stack Pointer %rsp

saved %rbp (Caller's stack frame)

Frame or Base Pointer %rbp

Caller's Frame

Return address (saved program counter)

parameter 7 (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)

⋮

parameter n

⋮

both caller & callee can access these:
push %rip (PC)
push input arguments to callee

Earlier Stack Frames

⋮

Bottom of Stack

lower memory address

call

return

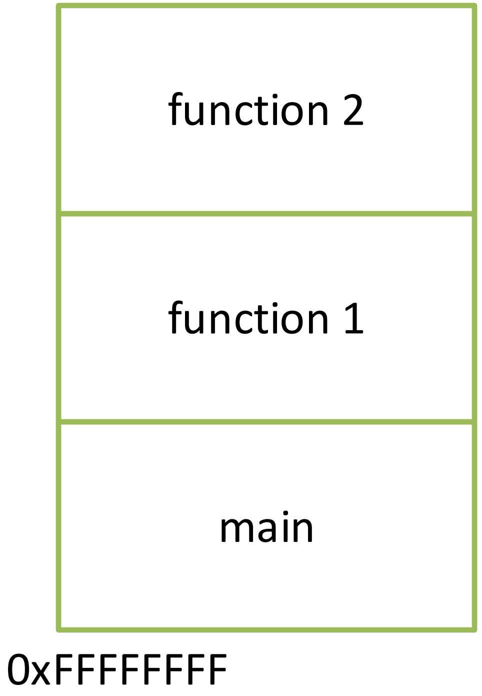
higher memory address

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries



Saving Registers

- Registers are a relatively scarce resource, but they’re fast to access. Memory is plentiful, but slower to access.

- Should the caller save its registers to free them up for the callee to use?
- Should the callee save the registers in case the caller was using them?
- Who needs more registers for temporary calculations, the caller or callee?

- Clearly the answers depend on what the functions do…

Splitting the difference…

- We can’t know the answers to those questions in advance…

- Divide registers into two groups:

Caller-saved: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

  Caller must save them prior to calling callee
  callee free to trash these,
  Caller will restore if needed

Callee-saved: %rbx, %r12, %r13, %r14, %r15

  Callee must save them first, and restore them before returning
  Caller can assume these will be preserved

Running Out of Registers

- Some computations require more than 16 general-purpose registers to store temporary values.

- *Register spilling*: The compiler will move some temporary values to memory, if necessary.
  - Values pushed onto stack, popped off later
  - No explicit variable declared by user
  - This is getting to the limits of CS 31!
    - – take CS 75 (compilers) for more details.

Up next…

- Connecting Arrays, Structs, and Pointers with assembly