CS 31: Intro to Systems
ISAs and Assembly

Vasanta Chaganti & Kevin Webb

Swarthmore College

September 26, 2023

Overview

- How to directly interact with hardware
- Instruction set architecture (ISA)
  - Interface between programmer and CPU
  - Established instruction format (assembly lang)
- Assembly programming (x86_64)

Abstraction

User / Programmer
Wants low complexity

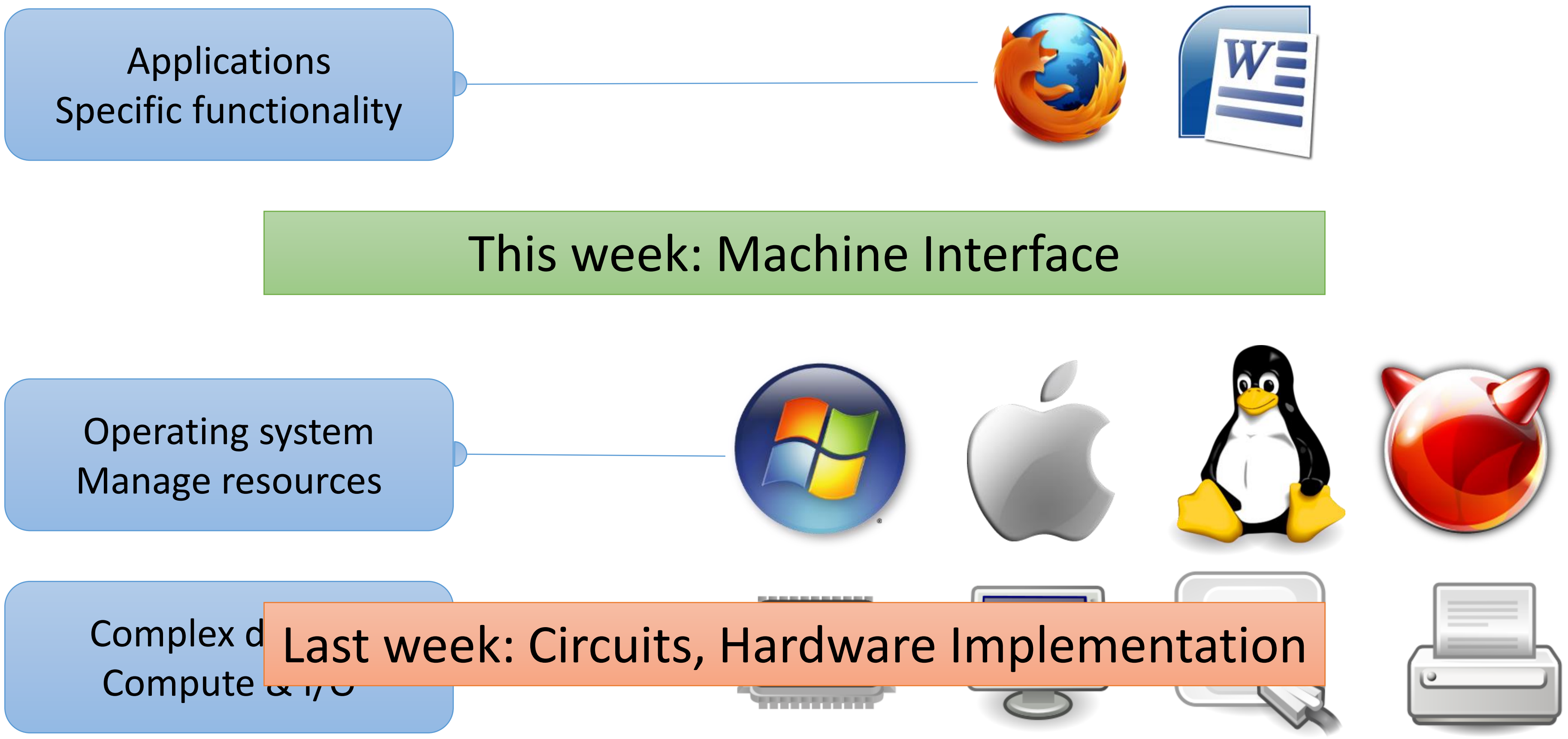
Applications
Specific functionality

Software library
Reusable functionality

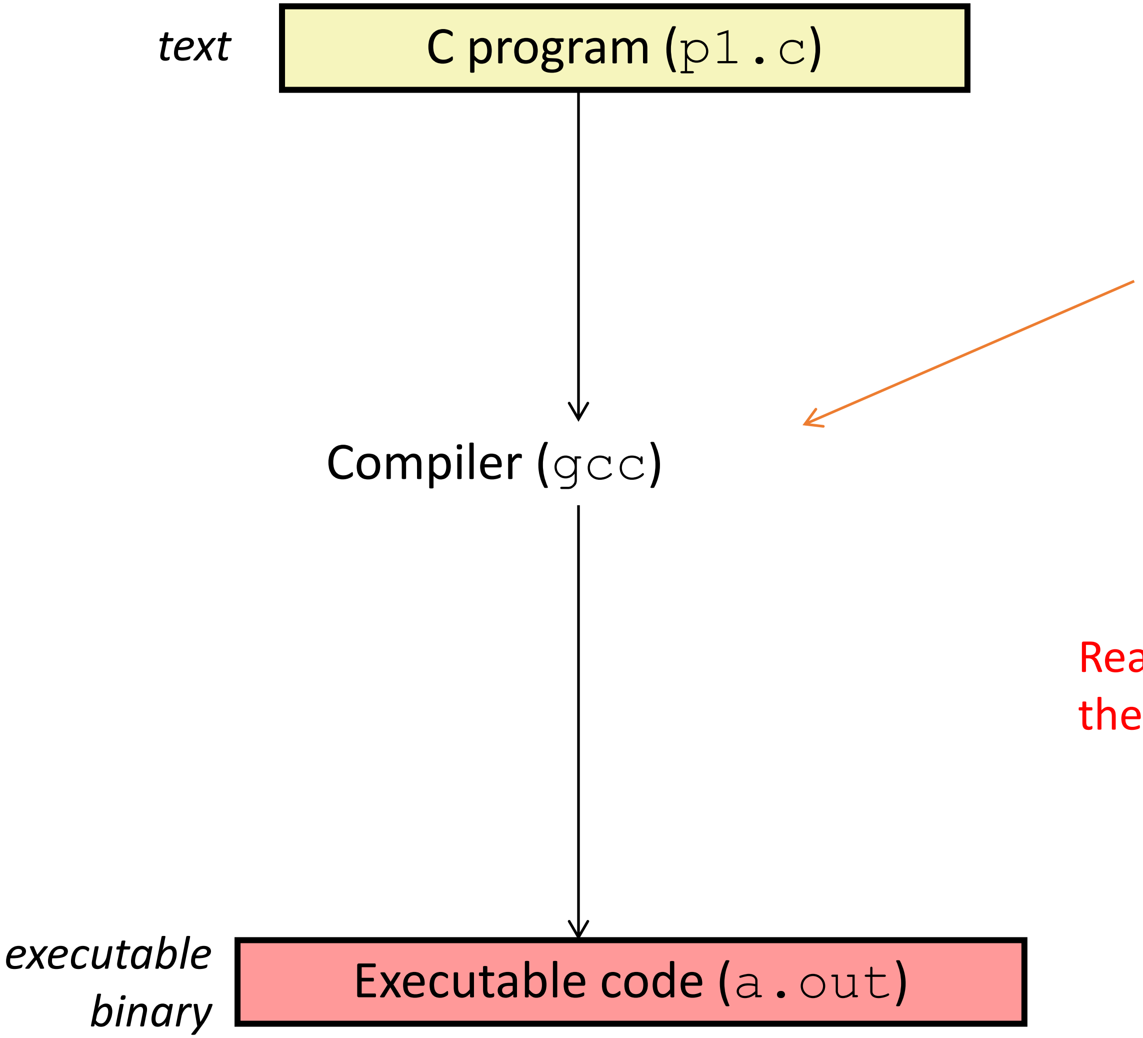
Operating system
Manage resources

Complex devices
Compute & I/O

Abstraction



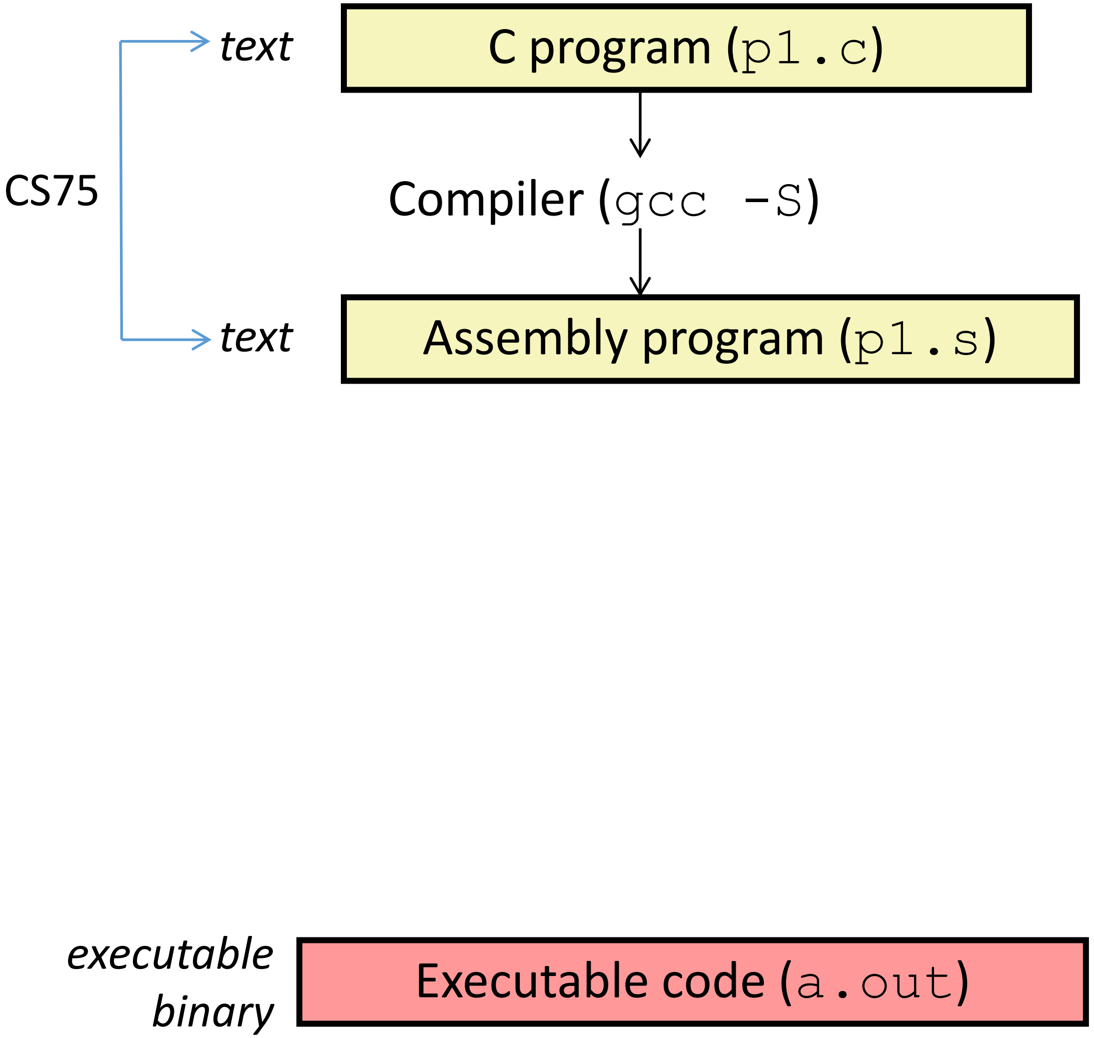
Compilation Steps (.c to a.out)



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

Reality is more complex:
there are intermediate steps!

Compilation Steps (.c to a.out)



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

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 to Assembly

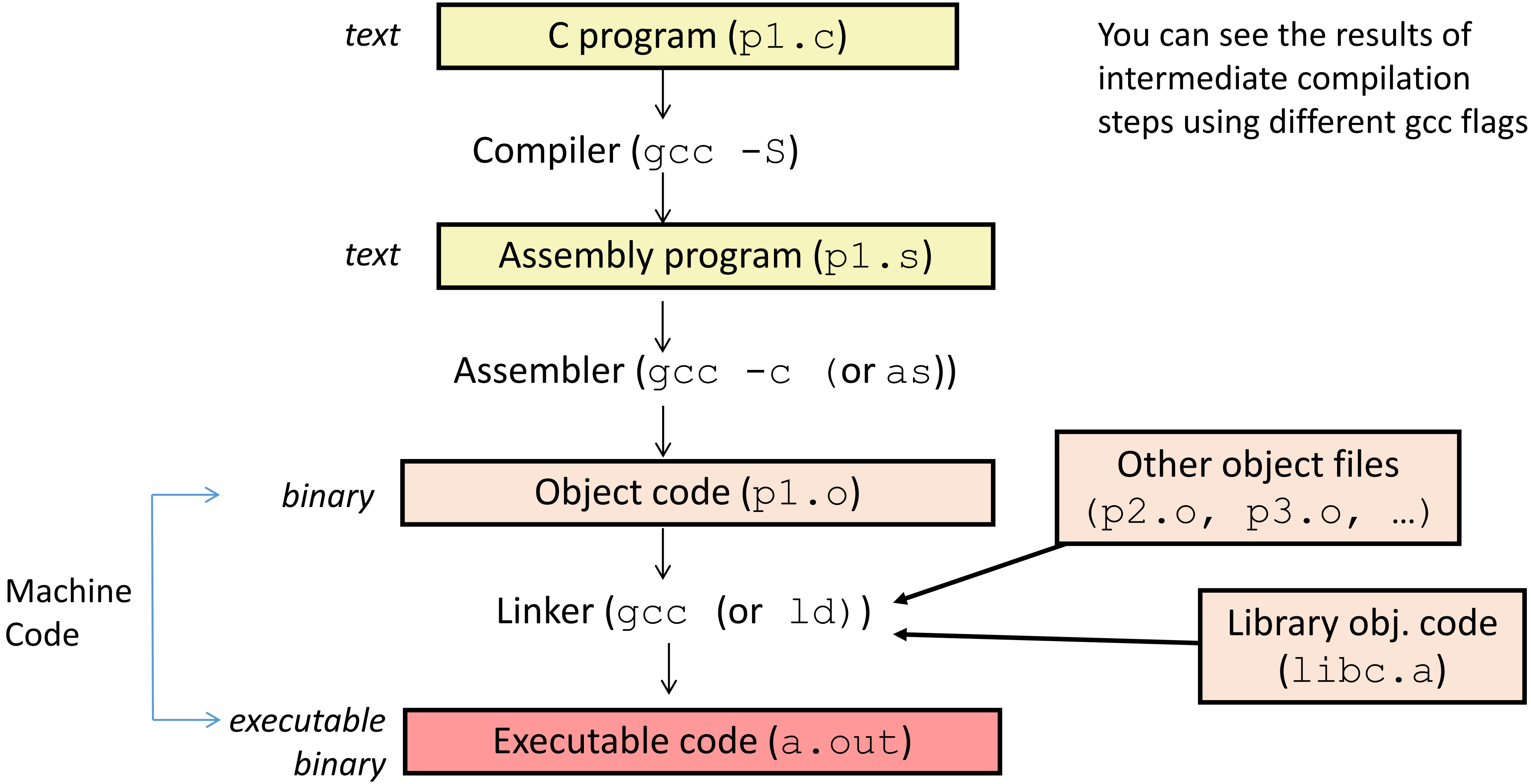
C

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

    a = a + b;

    return a;
}
```

x86_64 Assembly

```
push    %rbp
mov     %rsp,%rbp
movq    $10,-0x10(%rbp)
movq    $20,-0x8(%rbp)
mov     -0x8(%rbp),%rax
add     %rax,-0x10(%rbp)
mov     -0x10(%rbp),%rax
pop     %rbp
ret
```

Compilation Steps (.c to a.out)



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

Machine Code

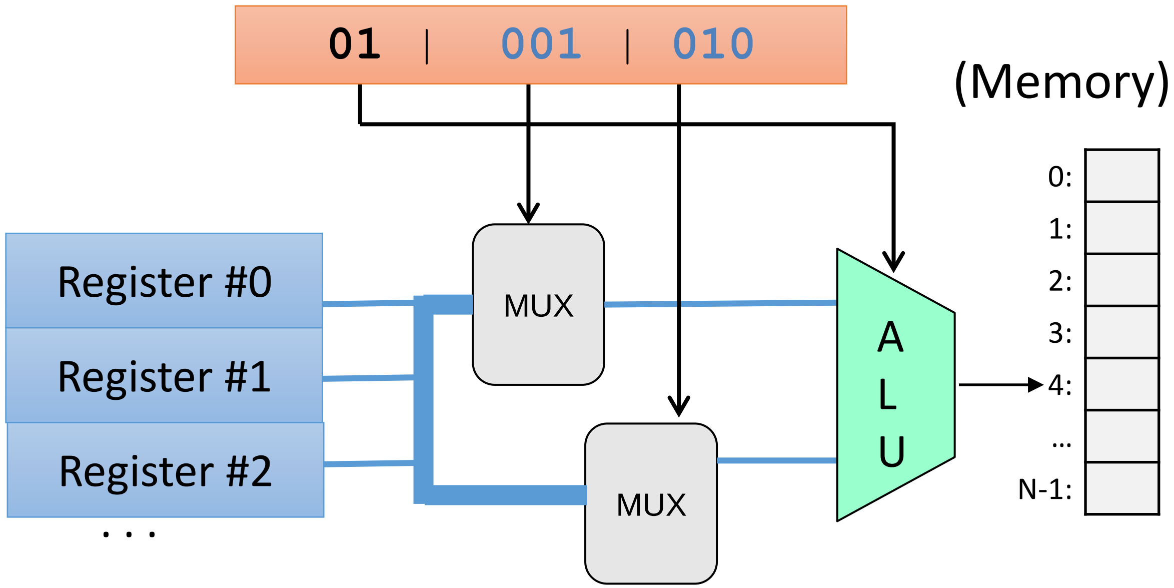
Binary (0's and 1's) encoding of instructions

- Opcode bits identify the instruction
- Other bits encode operand(s), where to store the results

(ex) `01001010` **opcode** operands

```
01 001 010
ADD %r1 %r2
```

- bits fed through different CPU circuitry:



Assembly to Machine Code

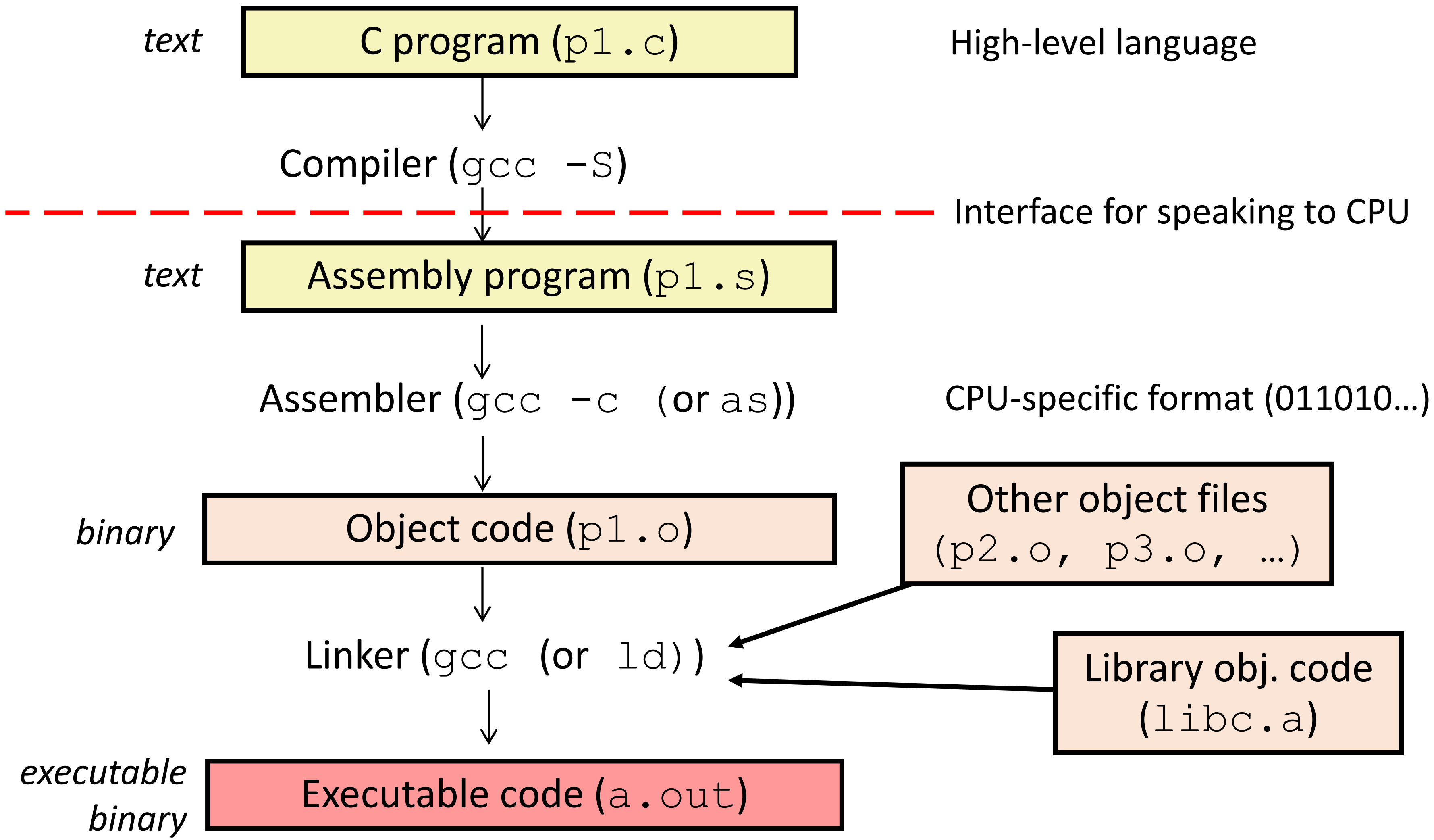
x86_64 Assembly

```
push    %rbp
mov     %rsp,%rbp
movq    $10,-0x10(%rbp)
movq    $20,-0x8(%rbp)
mov     -0x8(%rbp),%rax
add     %rax,-0x10(%rbp)
mov     -0x10(%rbp),%rax
pop     %rbp
ret
```

x86_64 Machine Code

```
55
48 89 e5
48 c7 45 f0 0a 00 00 00
48 c7 45 f8 14 00 00 00
48 8b 45 f8
48 01 45 f0
48 8b 45 f0
5d
c3
```

Compilation Steps (.c to a.out)



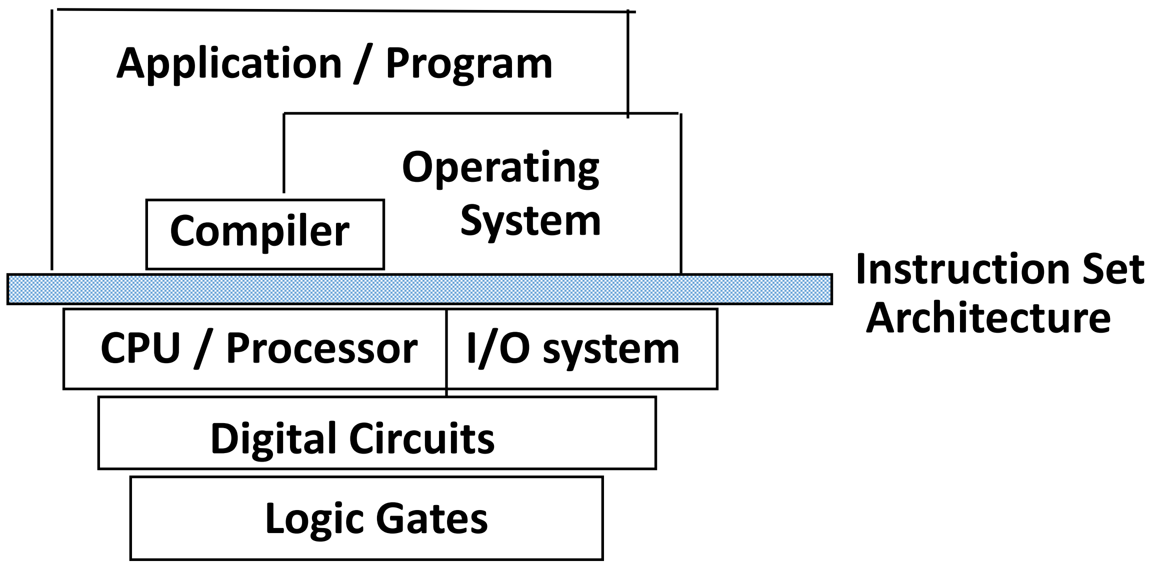
Instruction Set Architecture (ISA)

- ISA (or simply architecture):
  Interface between lowest software level and the hardware.

- Defines the language for controlling CPU state:
  - Defines a set of instructions and specifies their machine code format
  - Makes CPU resources (registers, flags) available to the programmer
  - Allows instructions to access main memory (potentially with limitations)
  - Provides control flow mechanisms (instructions to change what executes next)

Instruction Set Architecture (ISA)

- The agreed-upon interface between all software that runs on the machine and the hardware that executes it.



ISA Examples

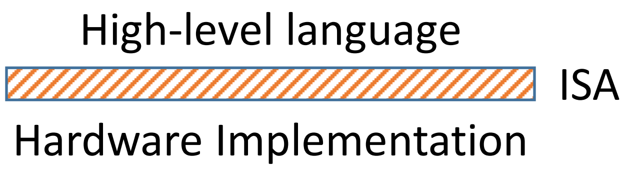
- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel x86_64
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360

How many of these ISAs have you used? (Don’t worry if you’re not sure. Try to guess based on the types of CPUs/devices you interact with.)

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel x86_64
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360

A. 0
B. 1-2
C. 3-4
D. 5-6
E. 7+

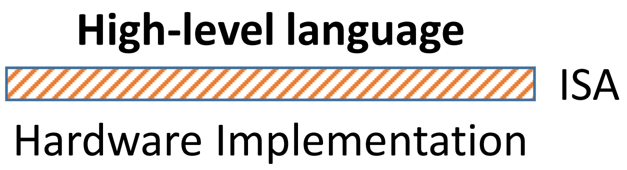
ISA Characteristics



- 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)

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

ISA Characteristics



- 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)

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

Instruction Translation

`sum.c (High-level C)`

```
long sum(long x, long y) {
  long result;
  result = x + y;
  return result;
}
```

`sum.s (Assembly)`

```
push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x18(%rbp)
mov     %rsi,-0x20(%rbp)
mov     -0x18(%rbp),%rdx
mov     -0x20(%rbp),%rax
add     %rdx,%rax
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
pop     %rbp
ret
```

`sum.s from sum.c:`

```
gcc -S sum.c
```

- Instructions to set up the stack frame and get argument values
- An add instruction to compute sum
- Instructions to return from function

Instruction Translation

`sum.c (High-level C)`

```
long sum(long x, long y) {
  long result;
  result = x + y;
  return result;
}
```

`sum.s (Assembly)`

```
push    %rbp
mov     %rsp,%rbp
mov     %rdi,-0x18(%rbp)
mov     %rsi,-0x20(%rbp)
mov     -0x18(%rbp),%rdx
mov     -0x20(%rbp),%rax
add     %rdx,%rax
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
pop     %rbp
ret
```

`sum.s from sum.c:`

`gcc -S sum.c`

- What should these instructions do?
- What is/isn't allowed by hardware?
- How complex should they be?

Example: supporting multiplication

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.

Which would you use if you were designing an ISA for your CPU? (Why?)

Simple instructions:

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

Powerful instructions:

```
MULT B, A
```

A. Simple
B. Powerful
C. Something else

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
  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

ISA Examples

- Intel IA-32 (CISC)
- ARM (RISC)
- MIPS (RISC)
- PowerPC (RISC)
- IBM Cell (RISC)
- Motorola 68k (CISC)
- Intel x86_64 (CISC)
- Intel IA-64 (Neither, VLIW)
- VAX (CISC)
- SPARC (RISC)
- Alpha (RISC)
- IBM 360 (CISC)

ISA Characteristics

- 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)

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

Intel x86 Family

Intel i386 (1985)

- 12 MHz - 40 MHz
- ~300,000 transistors
- Component size: 1.5 µm

Intel Core i9 9900k (2018)

- ~4,000 MHz
- ~7,000,000,000 transistors
- Component size: 14 nm

Everything in this family uses the same ISA (Same instructions)!

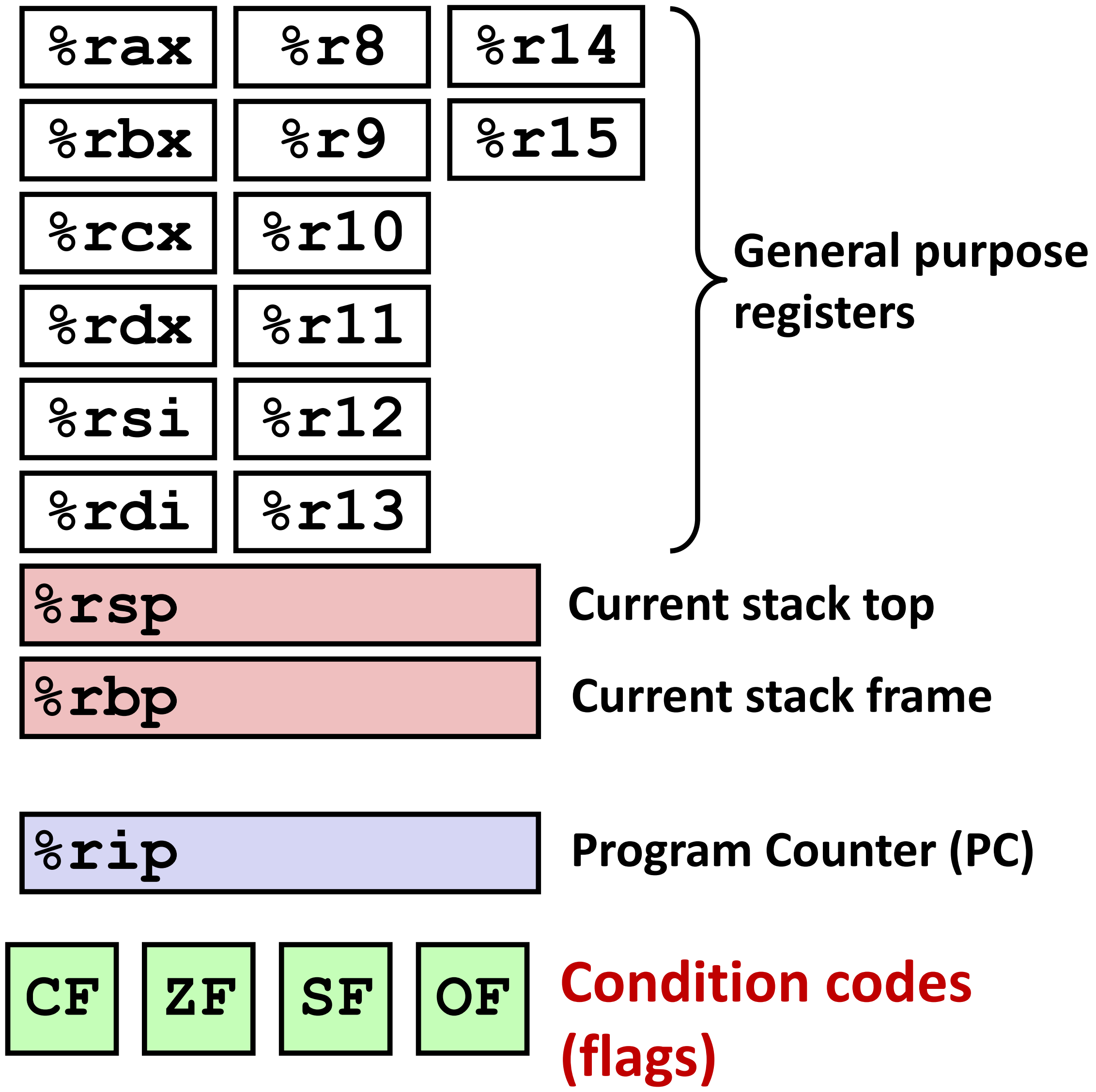
Recall: Instruction Set Architecture (ISA)

- ISA (or simply architecture):
  Interface between lowest software level and the hardware.

- Defines the language for controlling CPU state:
  - Defines a set of instructions and specifies their machine code format
  - Makes CPU resources (registers, flags) available to the programmer
  - Allows instructions to access main memory (potentially with limitations)
  - Provides control flow mechanisms (instructions to change what executes next)

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)



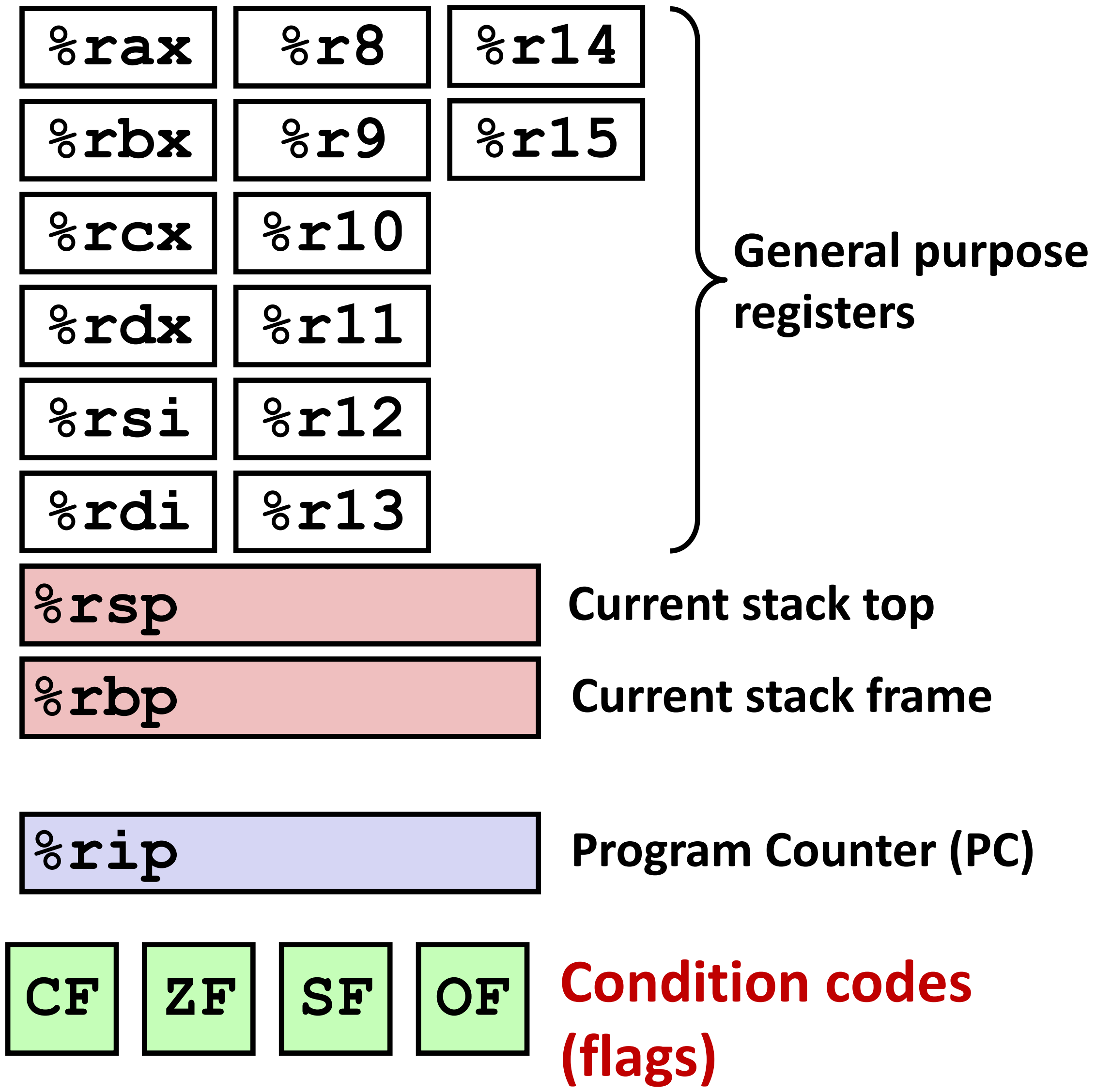
Component Registers

- Registers starting with “r” are 64-bit registers

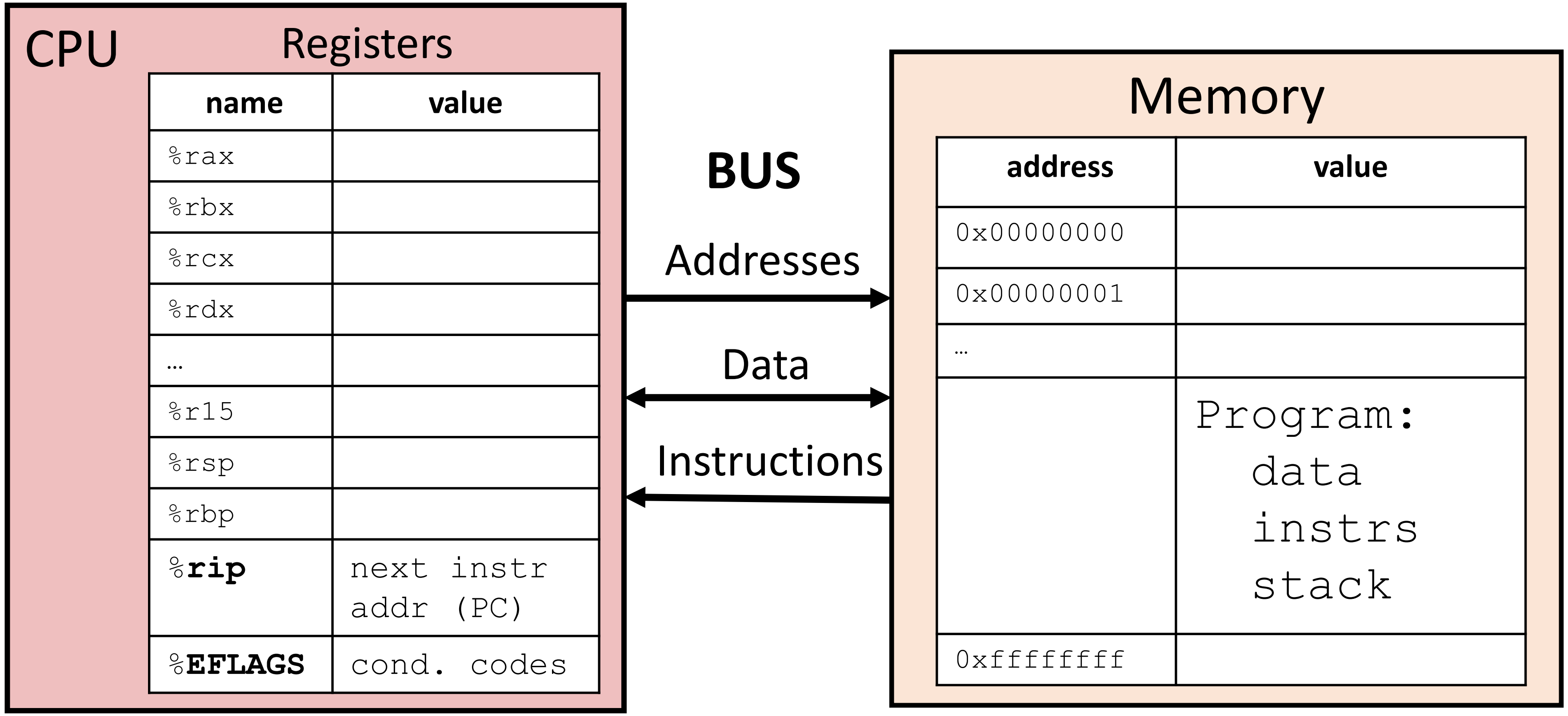
- Sometimes, you might only want to store 32 bits (e.g., `int` variable)

- You can access the lower 32 bits of a register:
  - with a prefix of e rather than r for registers %rax - %rdi (e.g., %eax, %ebx, …, %esi, %edi)

  - with a suffix of d for registers %r8 - %r15 (e.g., %r8d, %r9d, …, %r15d)



Assembly Programmer's View of State



Registers:

PC: Program counter (%rip)

Condition codes (%EFLAGS)

General Purpose (%rax - %r15)

Memory:

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

Types of assembly instructions

- Data movement
  - Move values between registers and memory
  - Examples: `mov`, `movl`, `movq`

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

l -> 32 bits
q -> 64 bits

- Load: move data from memory to register

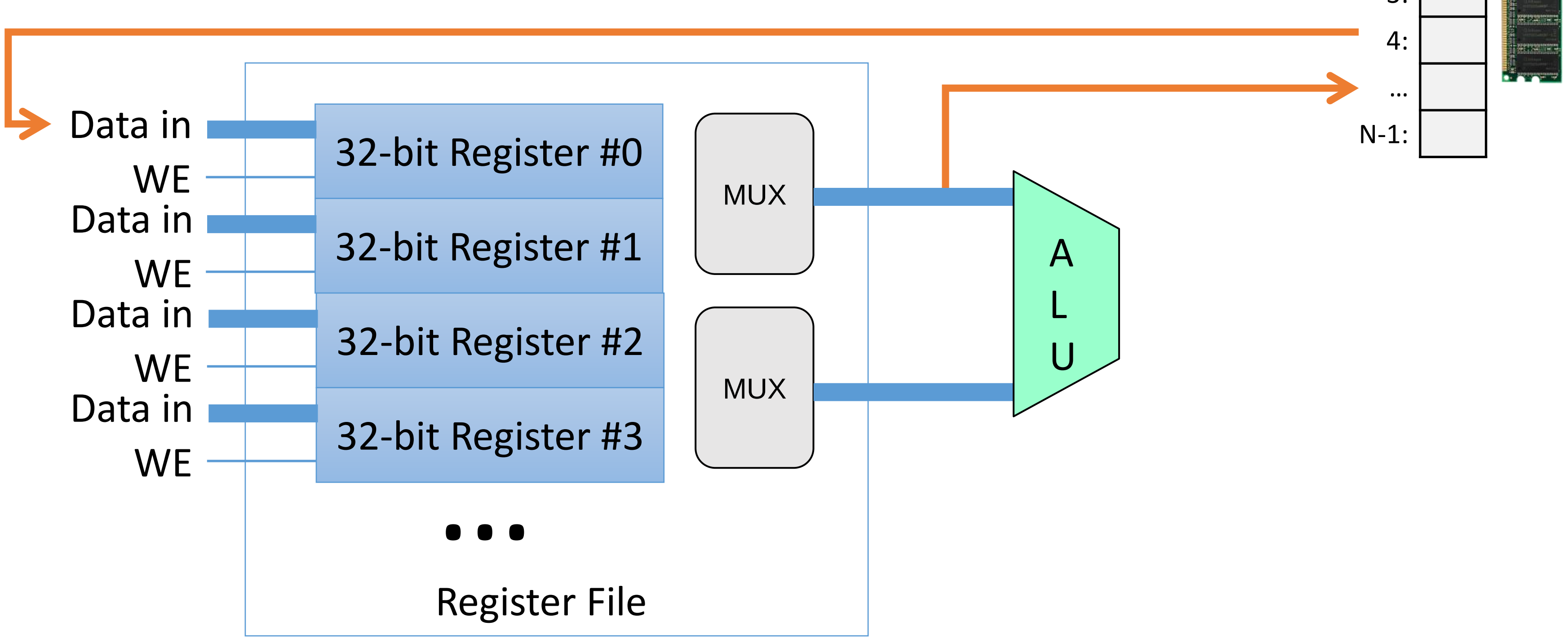
- Store: move data from register to memory

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)**



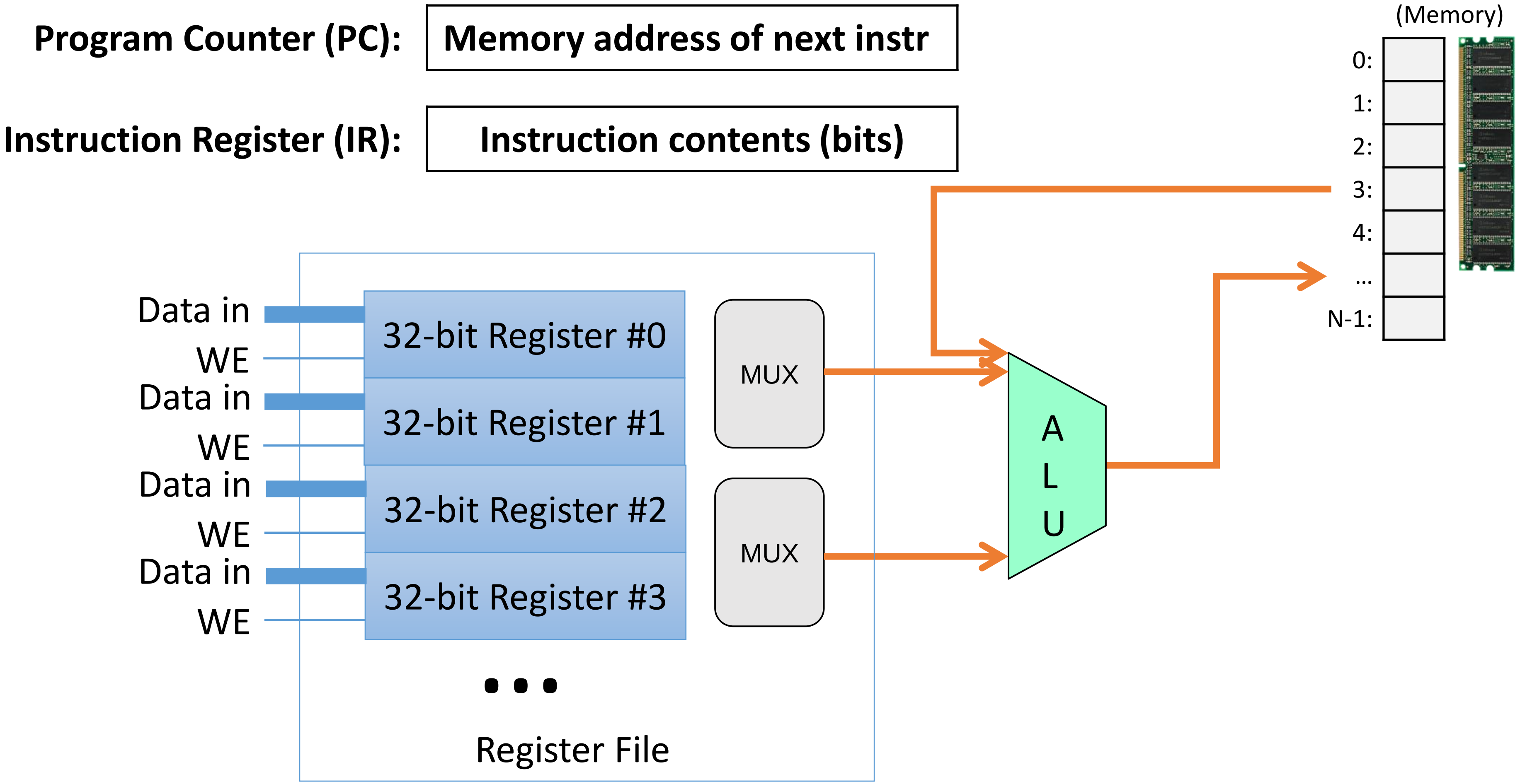
Types of assembly instructions

- Data movement
  - Move values between registers and memory

- Arithmetic
  - Uses ALU to compute a value
  - Examples: `add, addl, addq, sub, subl, subq`…

Arithmetic

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

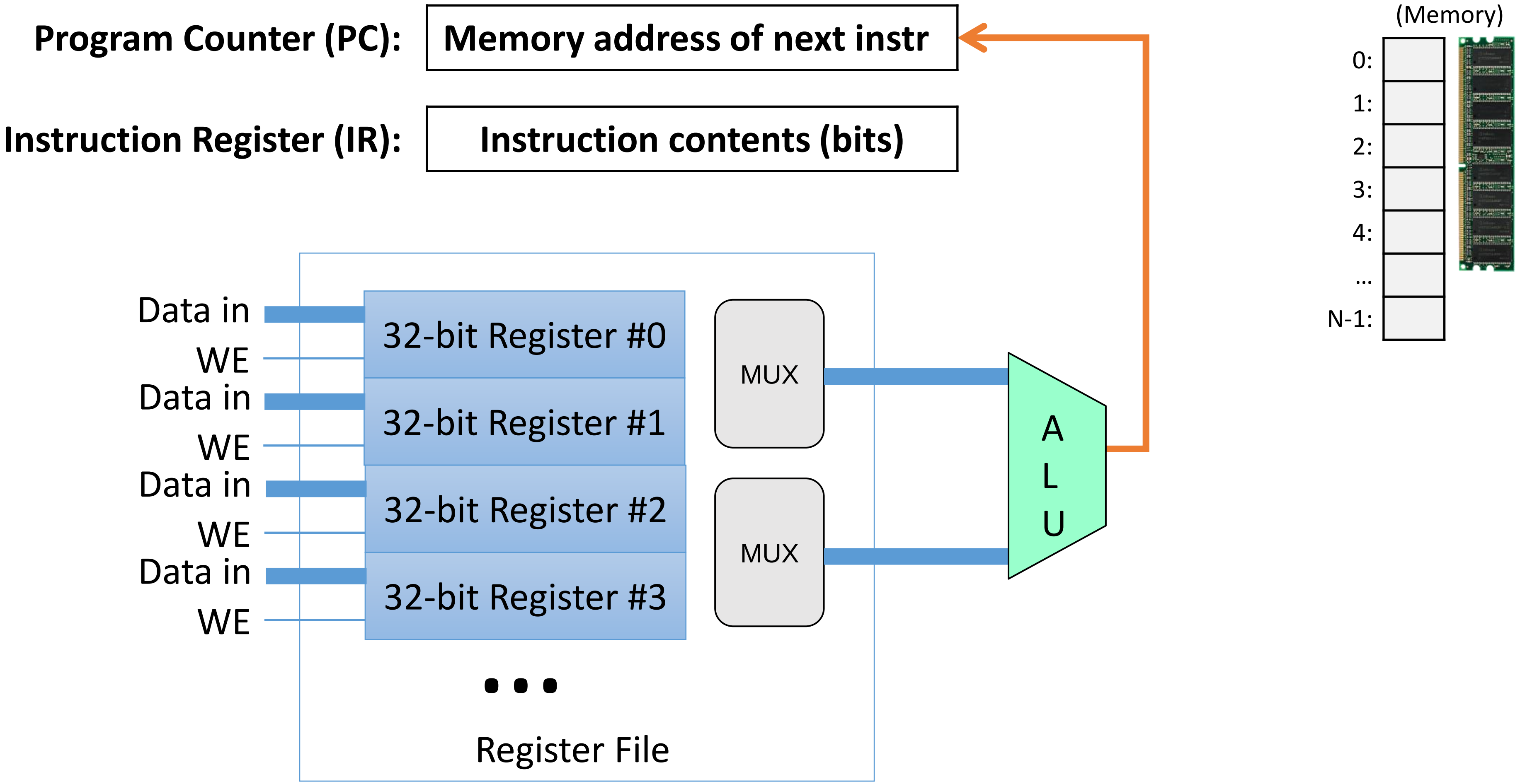


Types of assembly instructions

- Data movement
  - Move values between registers and memory
- Arithmetic
  - Uses ALU to compute a value
- Control
  - Change PC based on ALU condition code state
  - Example: `jmp`

Control

Change PC based on ALU condition code state.



Types of assembly instructions

- Data movement
  - Move values between registers and memory
- Arithmetic
  - Uses ALU to compute a value
- Control
  - Change PC based on ALU condition code state
- Stack / Function call (We'll cover these in detail later)
  - Shortcut instructions for common operations

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

Addressing Mode: Register

- Instructions can refer to the name of a register

- Examples:
  - `mov %rax, %r15`
    (Copy the contents of %rax into %r15 -- overwrites %r15, no change to %rax)

  - `add %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)

```
mov $10, %rax
```

- Put the constant value 10 in register rax.

```
add $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.

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

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

Addressing Mode: Memory

```
movl (%rcx), %rax
```

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

CPU Registers

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

(Memory)

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

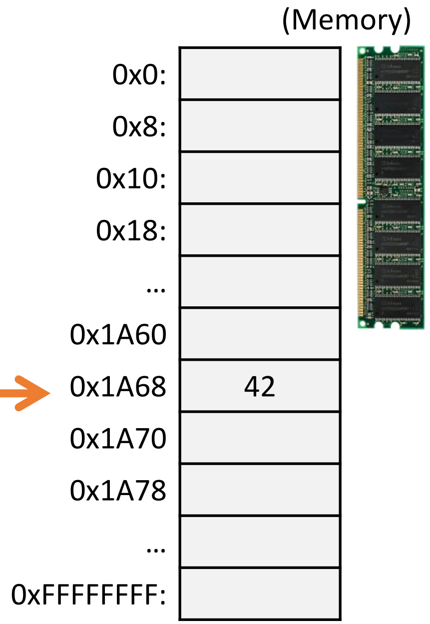
Addressing Mode: Memory

```
movl (%rcx), %rax
```

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

CPU Registers

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



1. Index into memory using the address in rcx.

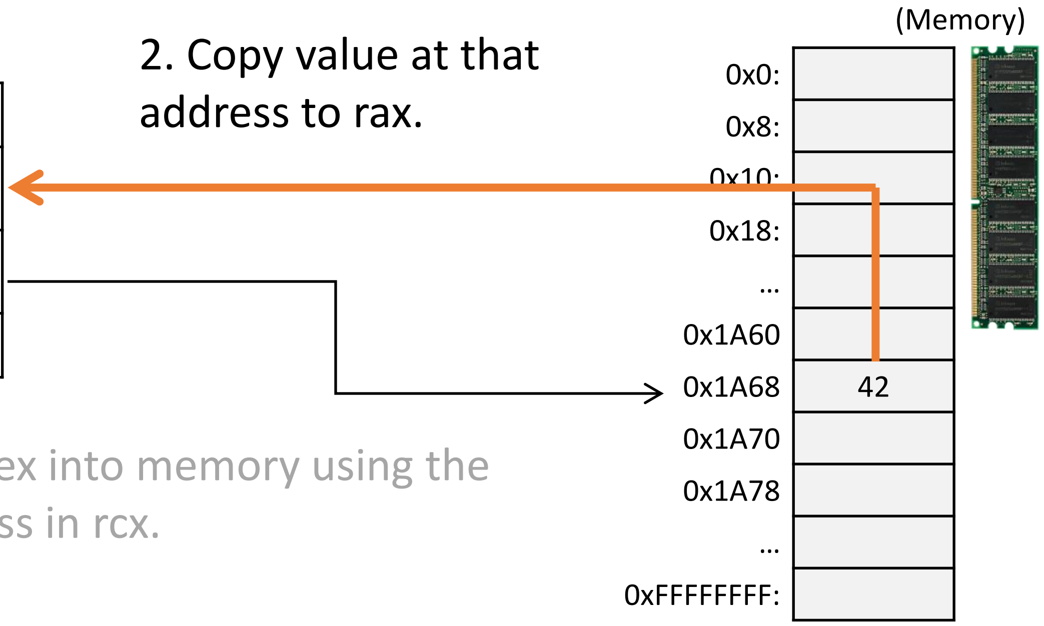
Addressing Mode: Memory

```
movl (%rcx), %rax
```

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

CPU Registers

name	value
%rax	42
%rcx	0x1A68
…	



Addressing Mode: Displacement

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

```
movl -24(%rbp), %rax
```

  - Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.

Addressing Mode: Displacement

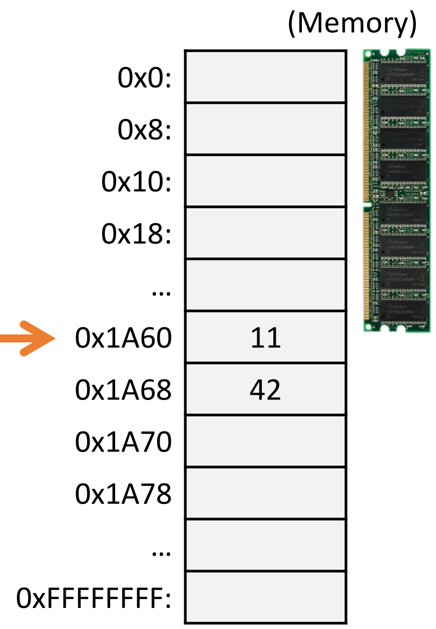
```
movl -24(%rbp), %rax
```

- Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.

CPU Registers

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

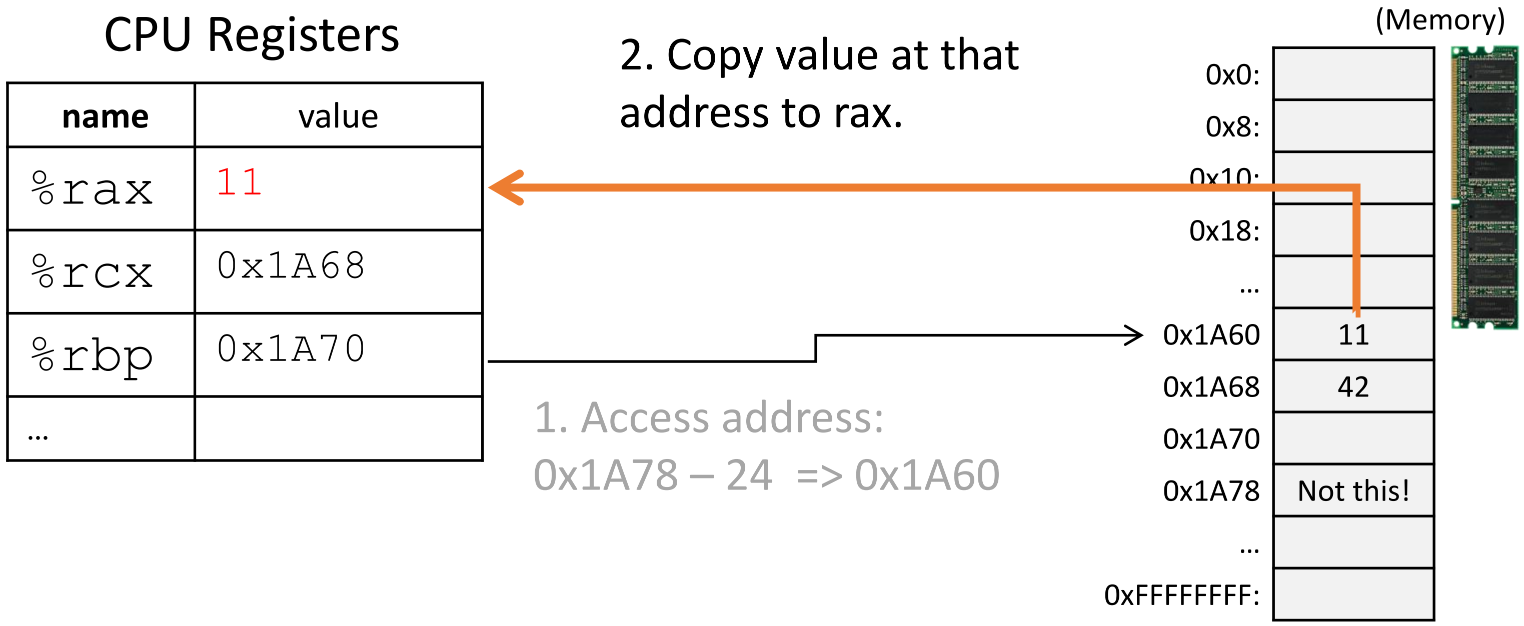
1. Access address:
0x1A78 – 24 => 0x1A60



Addressing Mode: Displacement

```
movl -24(%rbp), %rax
```

- Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.



Let’s try a few examples…

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?

```
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

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

```
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	…AE0
%rbp	…AE0

Memory	
Address	**Value**
0x1FFF000AD0	0
0x1FFF000AD8	0
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.

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

Solution

C code equivalent:

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

```
sub  $16, %rsp          Subtract 16 from %rsp, %rsp <- 0x…AD0
movq $3, -8(%rbp)       Move constant 3 to value at 0x…AD8 (x)
mov  $10, %rax          Move constant 10 to register %rax
sal  $1, %rax           Shift the value in %rax left by 1 bit
add  -8(%rbp), %rax     Add the value at 0x…AD8 (x) to %rax
movq %rax, -16(%rbp)    Store the value in %rax at 0x…AD0 (y)
add  $16, %rsp          Add 16 to %rsp, %rsp <- 0x…AE0
```

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?

```
…
mov  %rbp, %rcx
sub  $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)
```

Recall Memory Operands

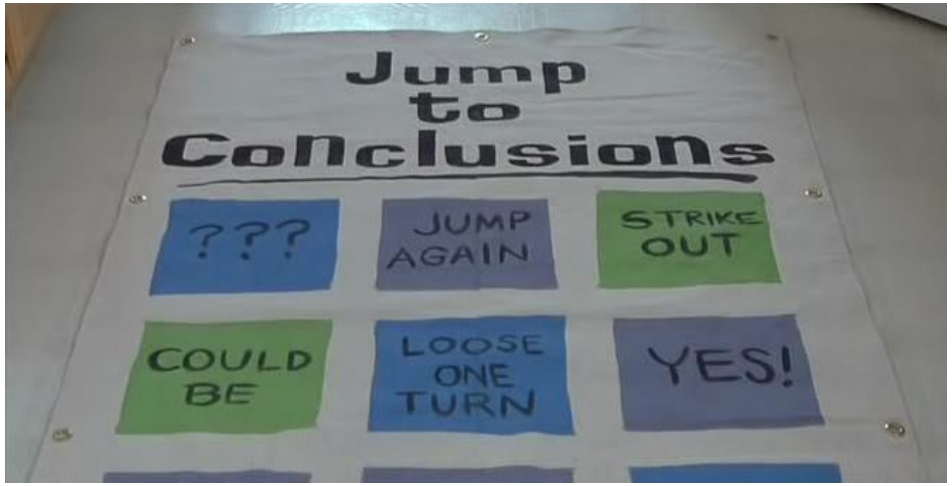
- `displacement(%reg)`
  - e.g., `add %rax, -8(%rbp)`
- x86_64 allows a memory operand as the source or destination, but NOT BOTH!
  - One of the operands must be a register
- This would not be allowed:
  - `add -8(%rbp), -16(%rbp)`
  - If you wanted this, `movq` one value into a register first

Control Flow

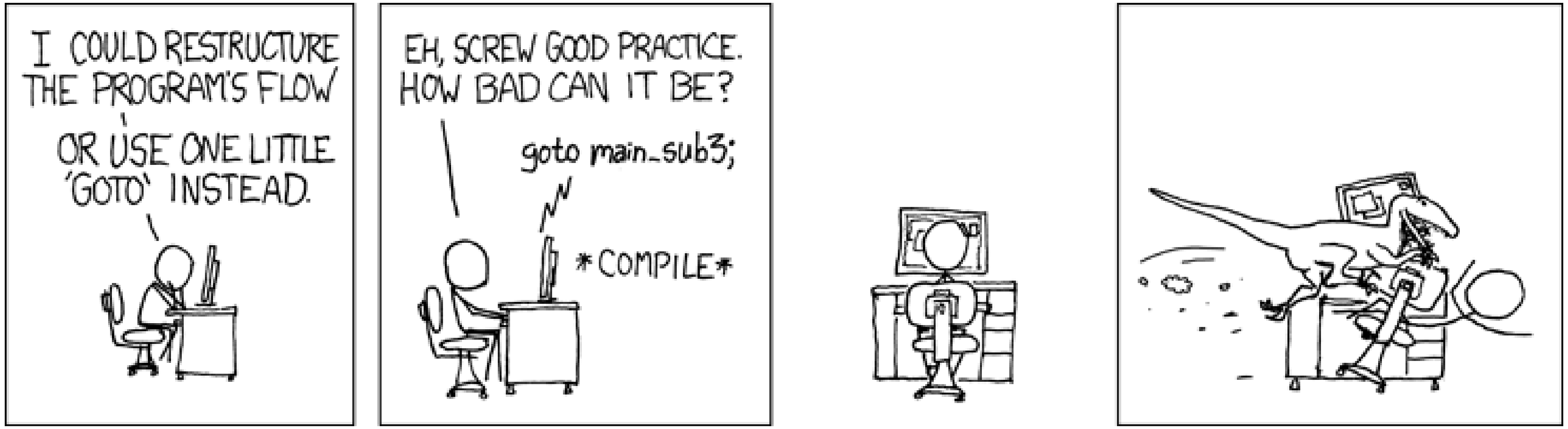
- Previous examples focused on:
  - data movement (mov, movq)
  - arithmetic (add, sub, or, neg, sal, etc.)

- Up next: Jumping!

  (Changing which instruction we execute next.)



Relevant XKCD



xkcd #292

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

Usage 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
```

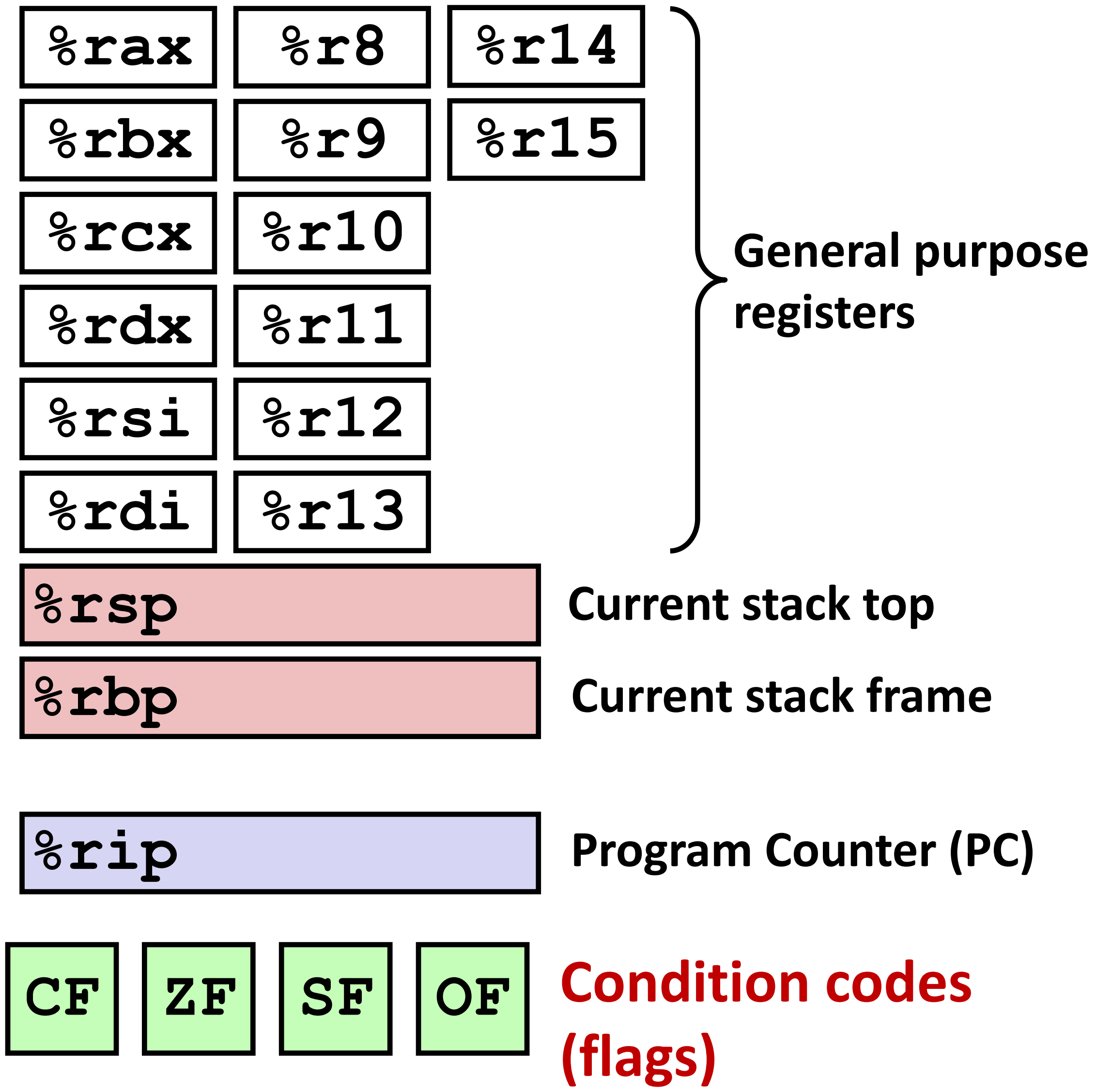
Condition Codes (or Flags)

- Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions

- x86_64 condition codes tell you:
  - 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)

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)



Instructions that set condition codes

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

2. CMP and TEST:

   `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
```

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)

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

Which flags would this `cmp` set?

- Suppose %rax holds 5, %rcx holds 7

```
cmp %rcx, %rax
```

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)

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

Conditional Jumping

- Jump based on which condition codes are set

Jump Instructions:
(See book section 7.4.1)

You do not need to memorize these!

	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned jg)
`jb`	`CF`	Below (unsigned)

Example Scenario

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

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

- Suppose user gives us a value via scanf
- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10

How would we use jumps/CCs for this?

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

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

Assume userval is stored in %eax at this point.

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

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

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

Assume userval is stored in %rax at this point.

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

(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:
  …
```

Visualization demo

- Try this in arithmetic mode:

https://asm.diveintosystems.org

Change the value 3 to 42 to alter the behavior.

```
  # Initialize rax
  mov $3, %rax

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

Loops

- We’ll look at these in the lab!

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)