CS 31: Introduction to Computer Systems

11: Assembly Arithmetic and Control

02-27-2025

SWARTHMORE COLLEGE

Announcements

- Lab 4 Due Today. Please submit your lab questionnaire
- HW Groups will rotate this week – Let me know your preferences!

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

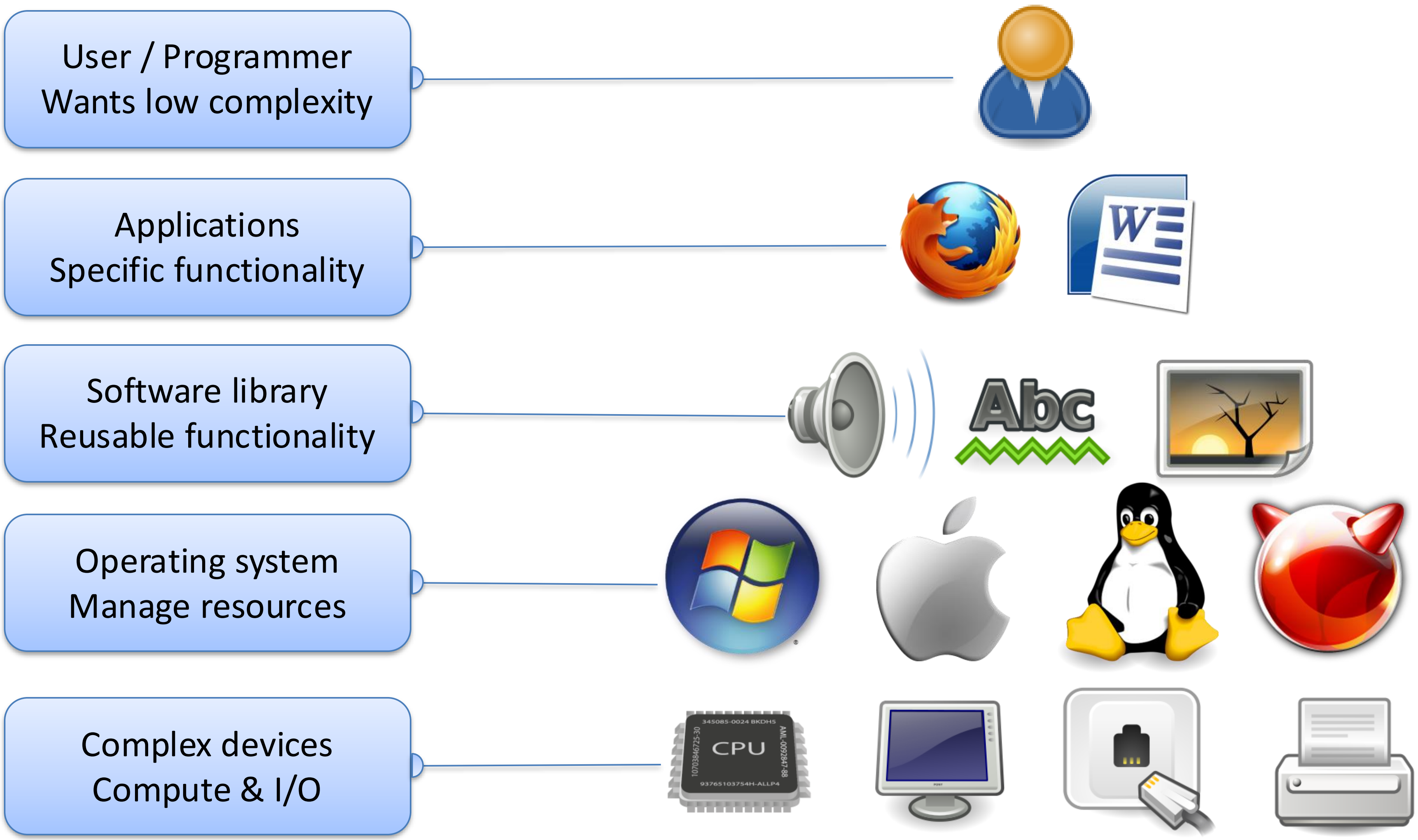
For new devices this should be okay,
For used you may need to reset frequency

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

Abstraction



User / Programmer
Wants low complexity

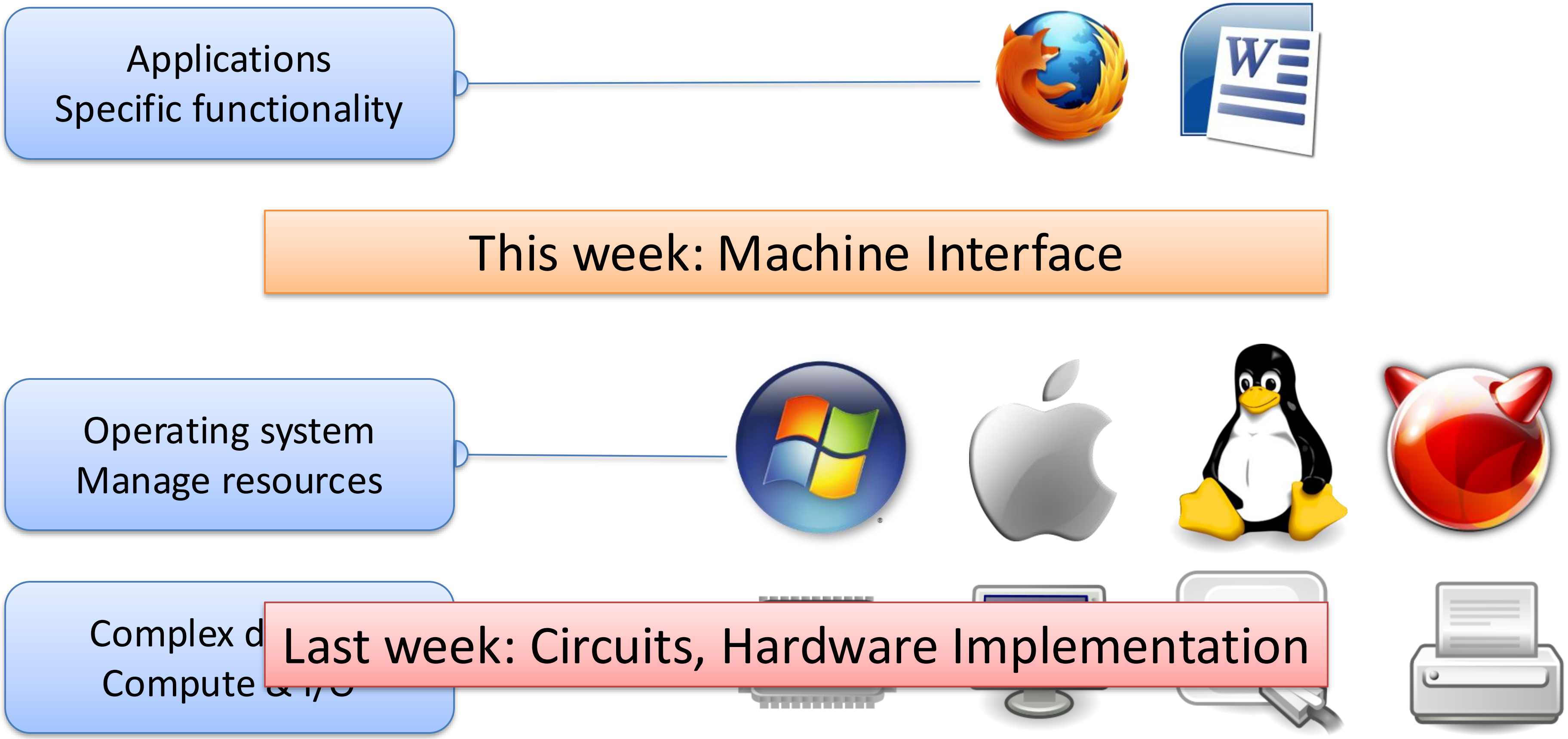
Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O

Abstraction



Applications
Specific functionality

This week: Machine Interface

Operating system
Manage resources

Complex d
Compute

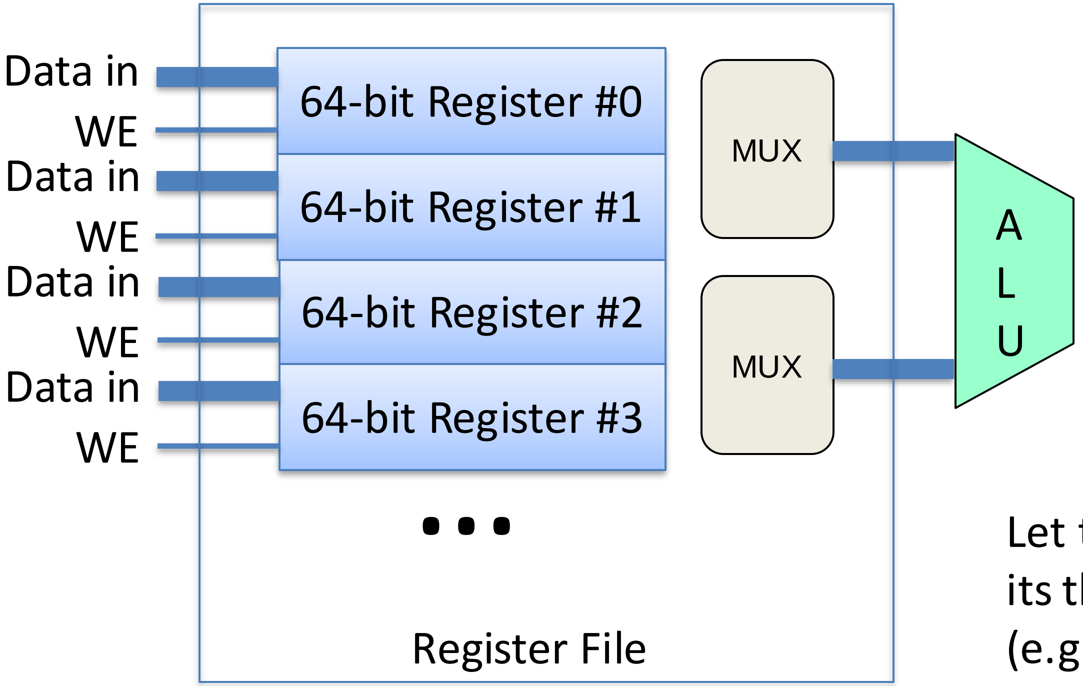
Last week: Circuits, Hardware Implementation

Hardware: Control, Storage, ALU circuitry

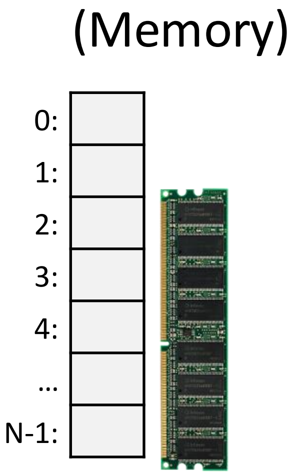
Program Counter (PC): **Address 0**

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

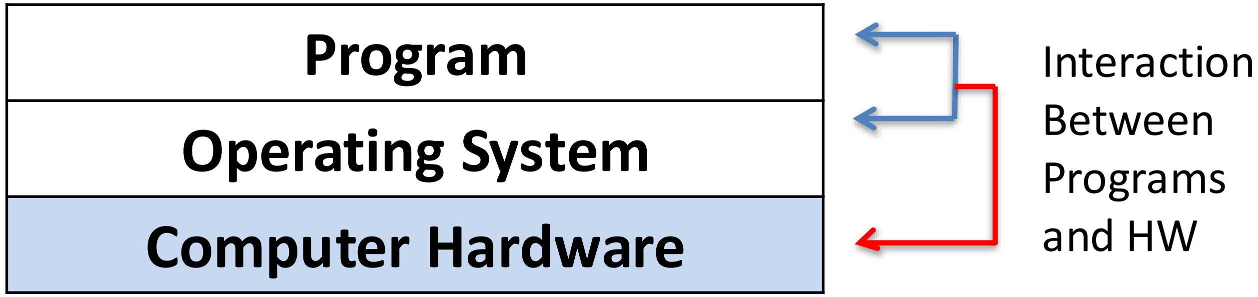
- acts on instruction bits to execute individual instructions
- PC value used to determine next instruction to execute



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

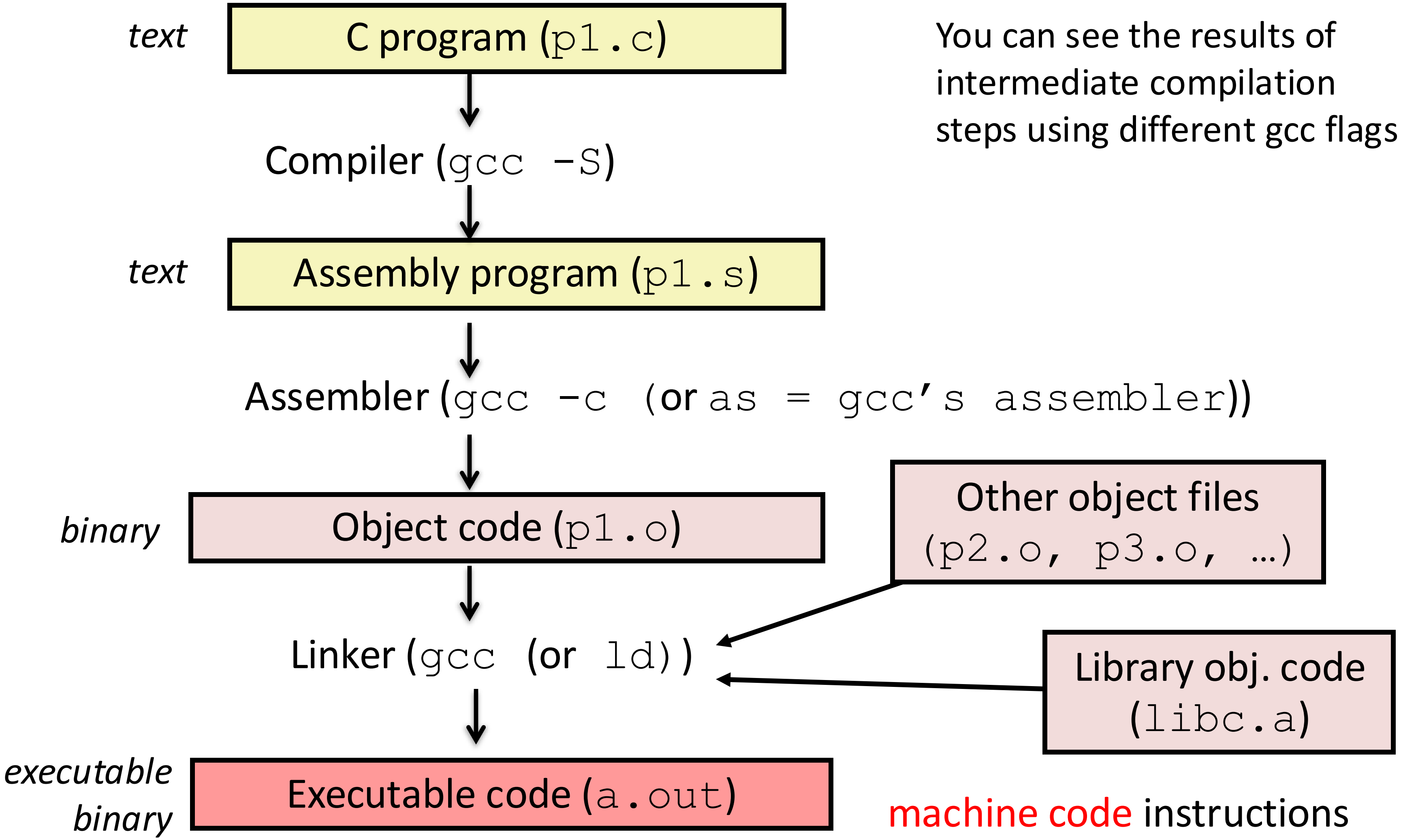


How a computer runs a program:



- We know: How HW Executes Instructions:
- This Week: Instructions and ISA
  - Program Encoding: C code to assembly code
  - Learn IA32 Assembly programming

Compilation Steps (.c to a.out)



text C program (`p1.c`)

Compiler (`gcc -S`)

text Assembly program (`p1.s`)

Assembler (`gcc -c` (or `as = gcc's assembler`))

binary Object code (`p1.o`)

Other object files (`p2.o, p3.o, …`)

Linker (`gcc` (or `ld`))

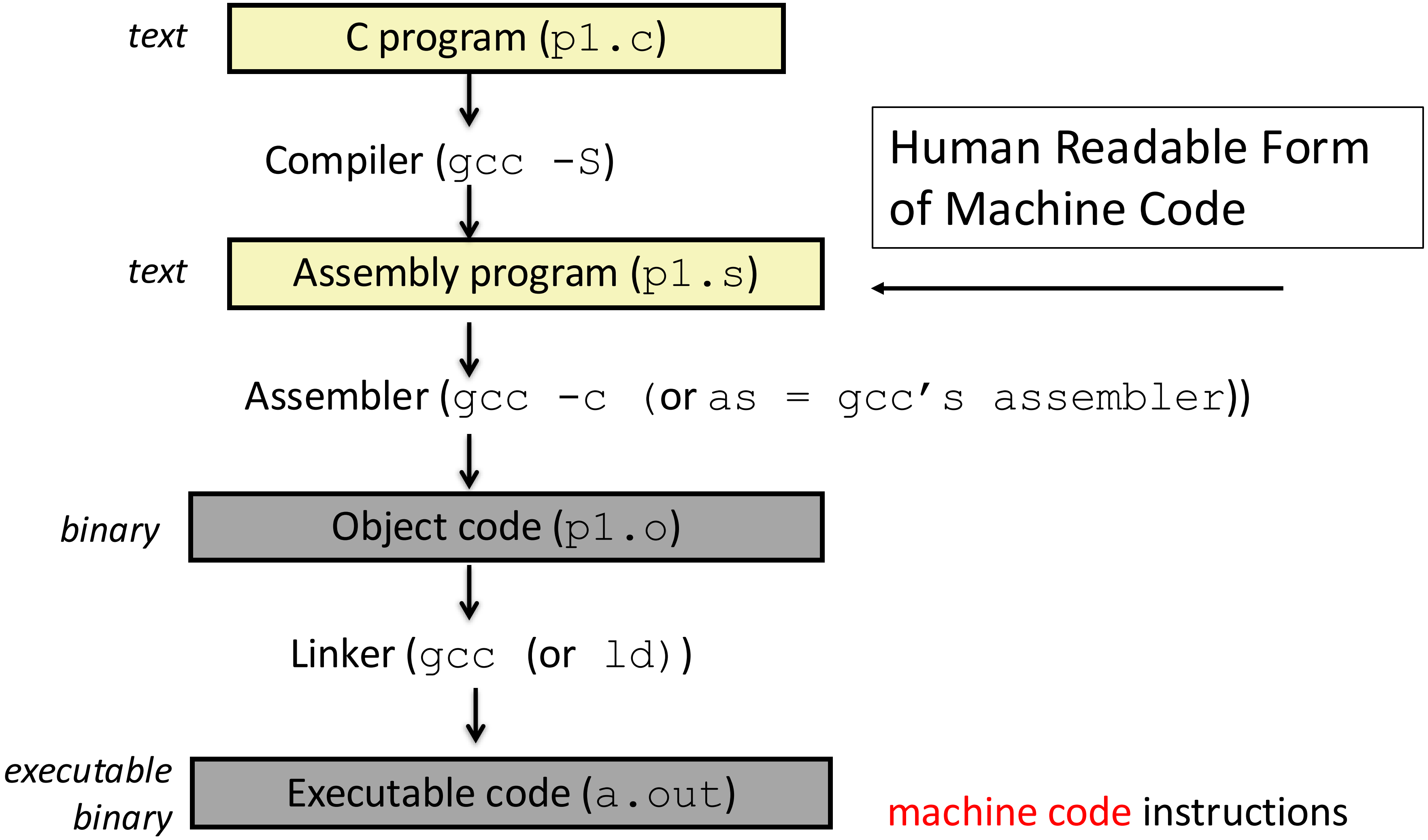
Library obj. code (`libc.a`)

executable binary Executable code (`a.out`)

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

machine code instructions

Assembly Code



text

C program (`p1.c`)

Compiler (`gcc -S`)

text

Assembly program (`p1.s`)

Human Readable Form of Machine Code

Assembler (`gcc -c` (or `as = gcc's assembler`))

binary

Object code (`p1.o`)

Linker (`gcc` (or `ld`))

executable binary

Executable code (`a.out`)

machine code instructions

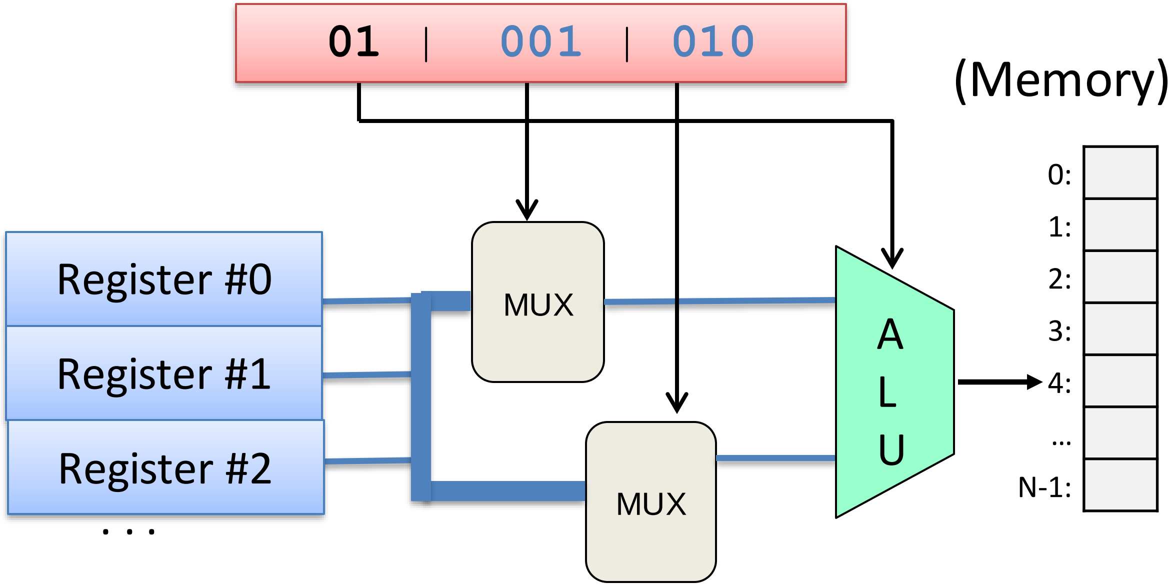
Machine Code

Binary (0's and 1's) Encoding of ISA Instructions

- some bits: encode the instruction (opcode bits)
- others encode operand(s)

```
(eg) 01001010   opcode operands
     01 001 010
    ADD %r1 %r2
```

- different bits fed through different CPU circuitry:



What is “assembly”?

```
pushq %rbp
movq  %rsp,     %rbp
subq  $16,      %rsp
movq  $10,  -16(%rbp)
movq  $20,   -8(%rbp)
movq  -8(%rbp), $rax
addq  $rax,  -8(%rbp)
movq  -8(%rbp), %rax
leaveq
```

Assembly is the “human readable” form of the instructions a machine can understand.

```
objdump -d a.out
```

Object / Executable / Machine Code

Assembly	**Machine Code (Hexadecimal)**
`pushq %rbp`	`55`
`movq  %rsp,      %rbp`	`89 E5`
`subq  $16,       %rsp`	`83 EC 10`
`movq  $10,   -16(%rbp)`	`C7 45 F8 0A 00 00 00`
`movq  $20,    -8(%rbp)`	`C7 45 FC 14 00 00 00`
`movq  -8(%rbp), $rax`	`8B 45 FC`
`addq  $rax,   -8(%rbp)`	`01 45 F8`
`movq  -8(%rbp), %rax`	`B8 45 F8`
`leaveq`	`C9`

Almost a 1-to-1 mapping to Machine Code
Hides some details like num bytes in instructions

Object / Executable / Machine Code

Assembly

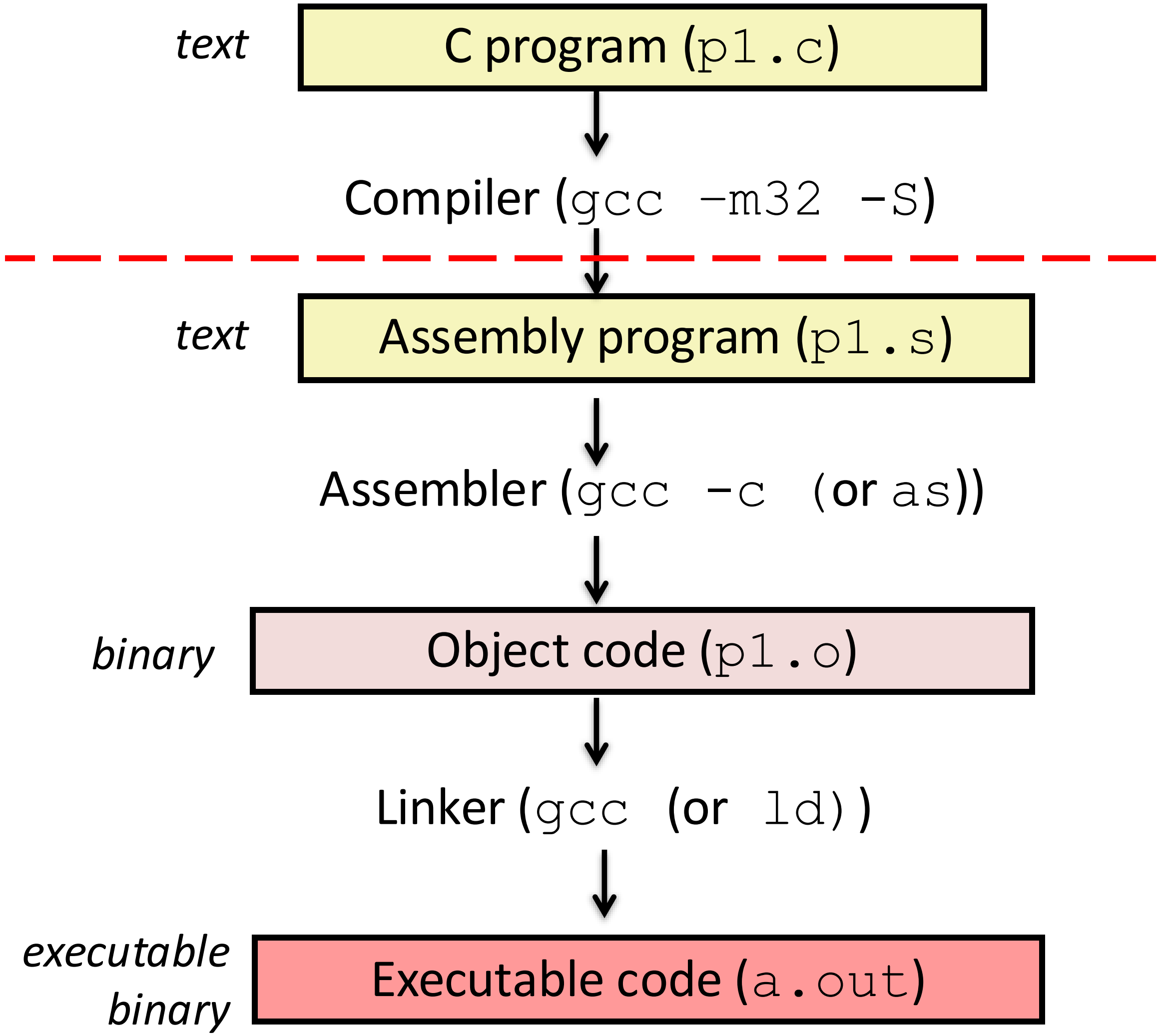
```
pushq %rbp
movq  %rsp,      %rbp
subq  $16,       %rsp
movq  $10,  -16(%rbp)
movq  $20,   -8(%rbp)
movq  -8(%rbp), $rax
addq  $rax,  -8(%rbp)
movq  -8(%rbp), %rax
leaveq
```

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

	a = a + b;

	return a;
}
```

Compilation Steps (.c to a.out)



text C program (`p1.c`)

High-level language

Compiler (`gcc -m32 -S`)

Interface for speaking to CPU

text Assembly program (`p1.s`)

Assembler (`gcc -c` (or `as`))

binary Object code (`p1.o`)

CPU-specific format (011010…)

Linker (`gcc` (or `ld`))

executable binary Executable code (`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)

Intel x86 Family

Intel i386 (1985)

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

Intel Core i9 9900k (2018)

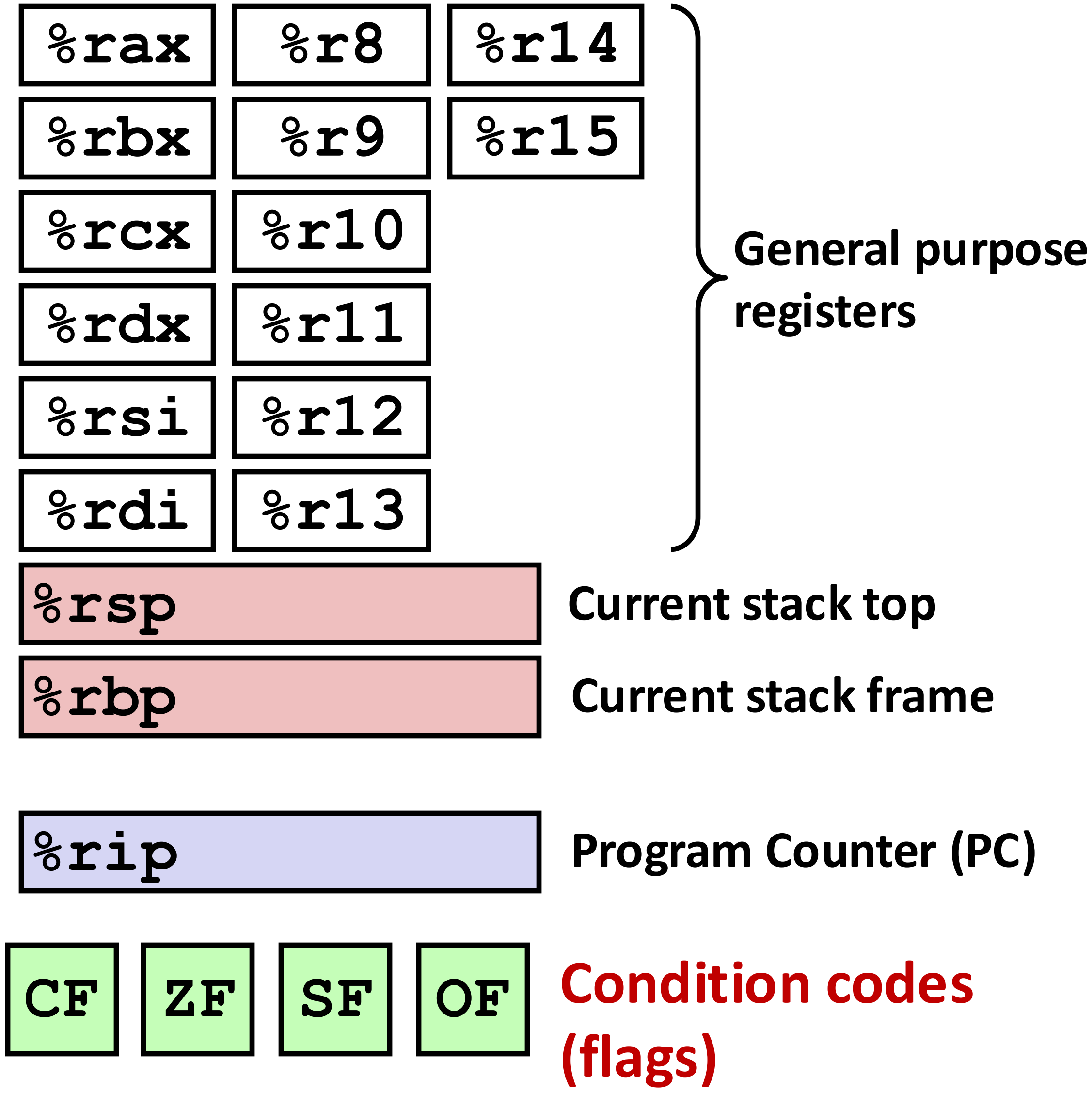
- ~4,000 MHz
- ~7,000,000,000 transistors
- Component size: 14 nm

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

Processor State in Registers

Working memory for currently executing program

- Temporary data: %rax - %r15
- Current stack frame
- %rbp: base pointer
- %rsp: stack pointer
- 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**

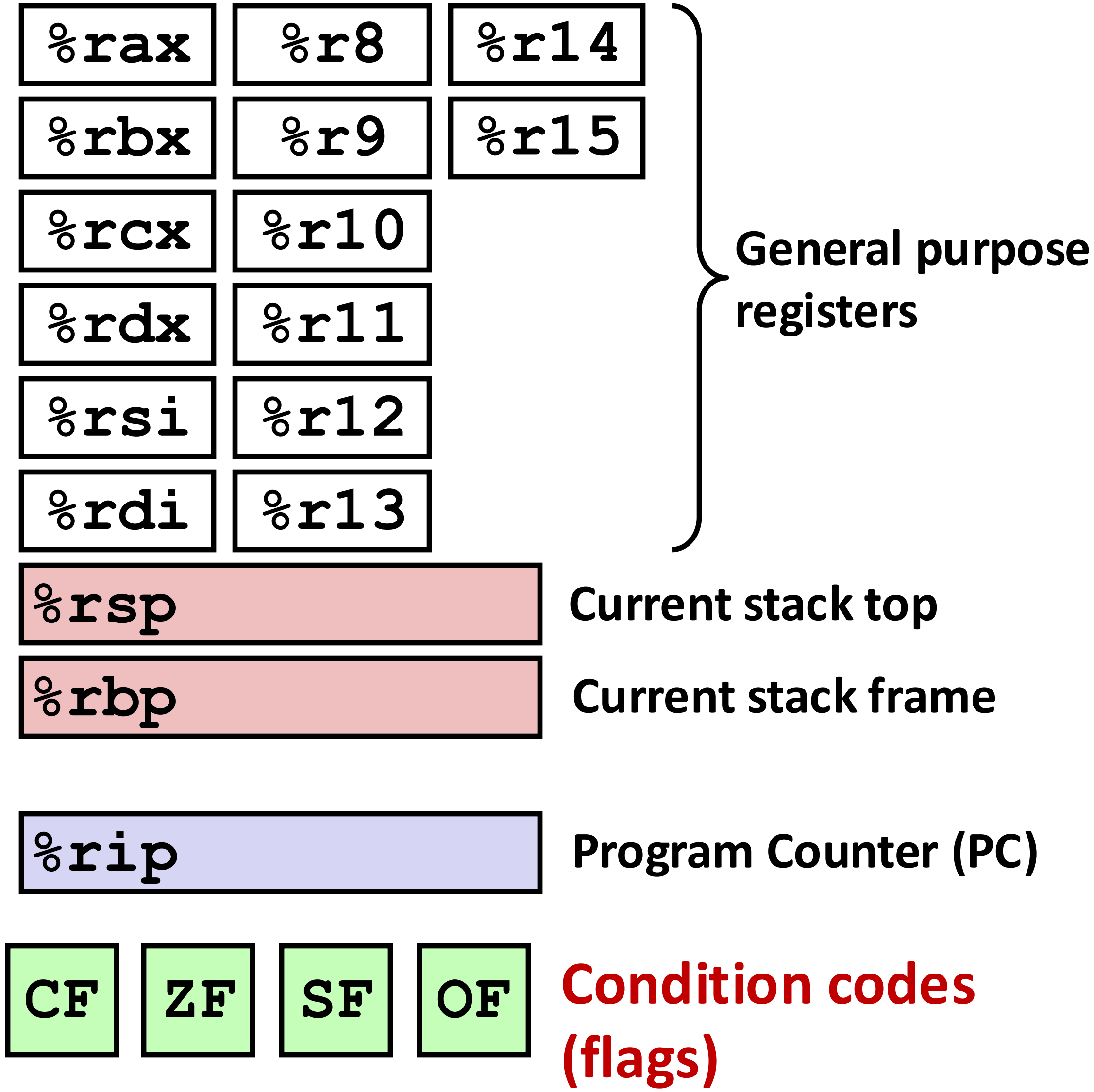
`%rip` **Program Counter (PC)**

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

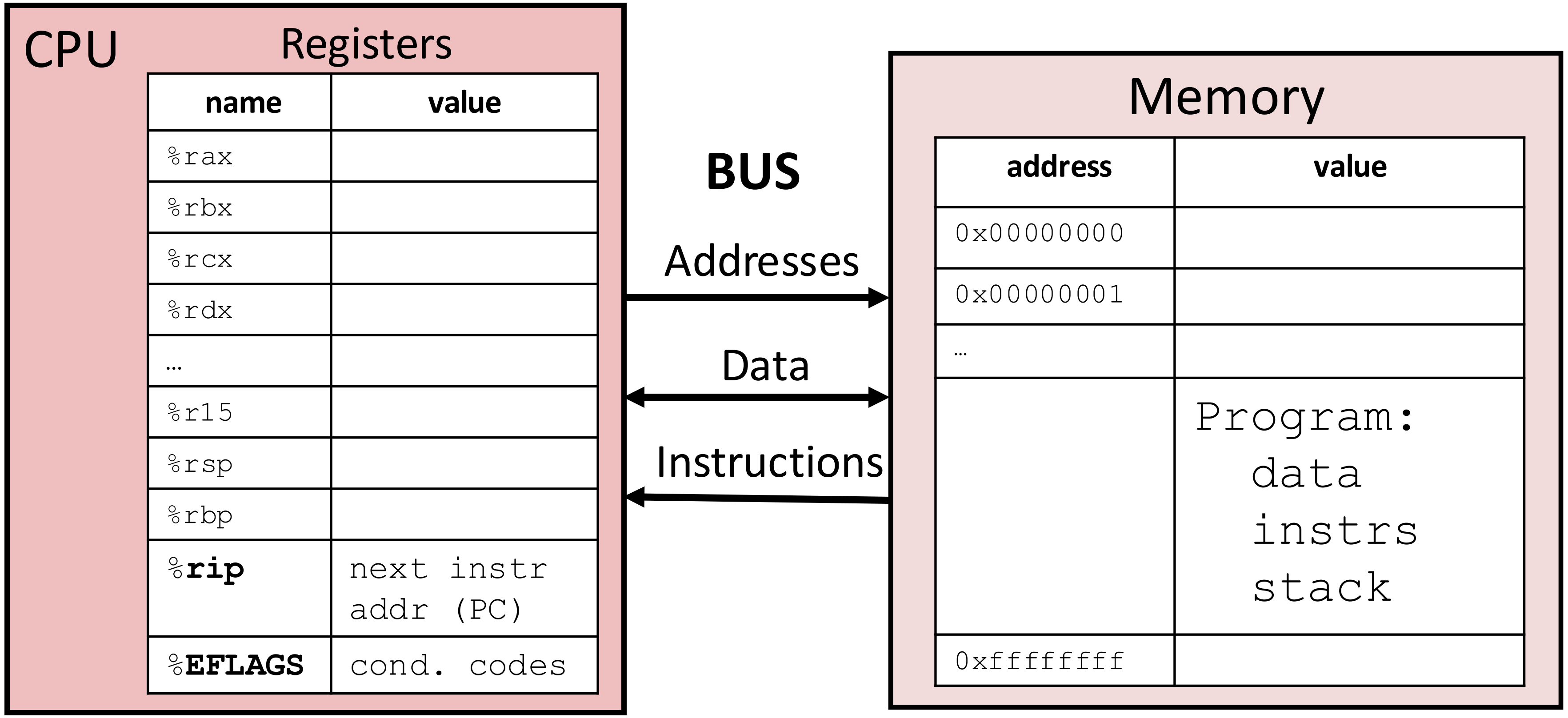
Component Registers

- Registers starting with "r" are 64-bit registers
  - %rax, %rbx, …, %rsi, %rdi

- Sometimes, you might only want to store 32 bits (e.g., `int` variable)
  - You can access the lower 32 bits of a register with prefix e:
  - %eax, %ebx, …, %esi, %edi
  - with a suffix of d for registers %r8 to %r15
  - %r8d, %r9d, …, %r15d



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

Types of assembly instructions

- Data movement
  - Move values between registers and memory
  - Examples: `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).

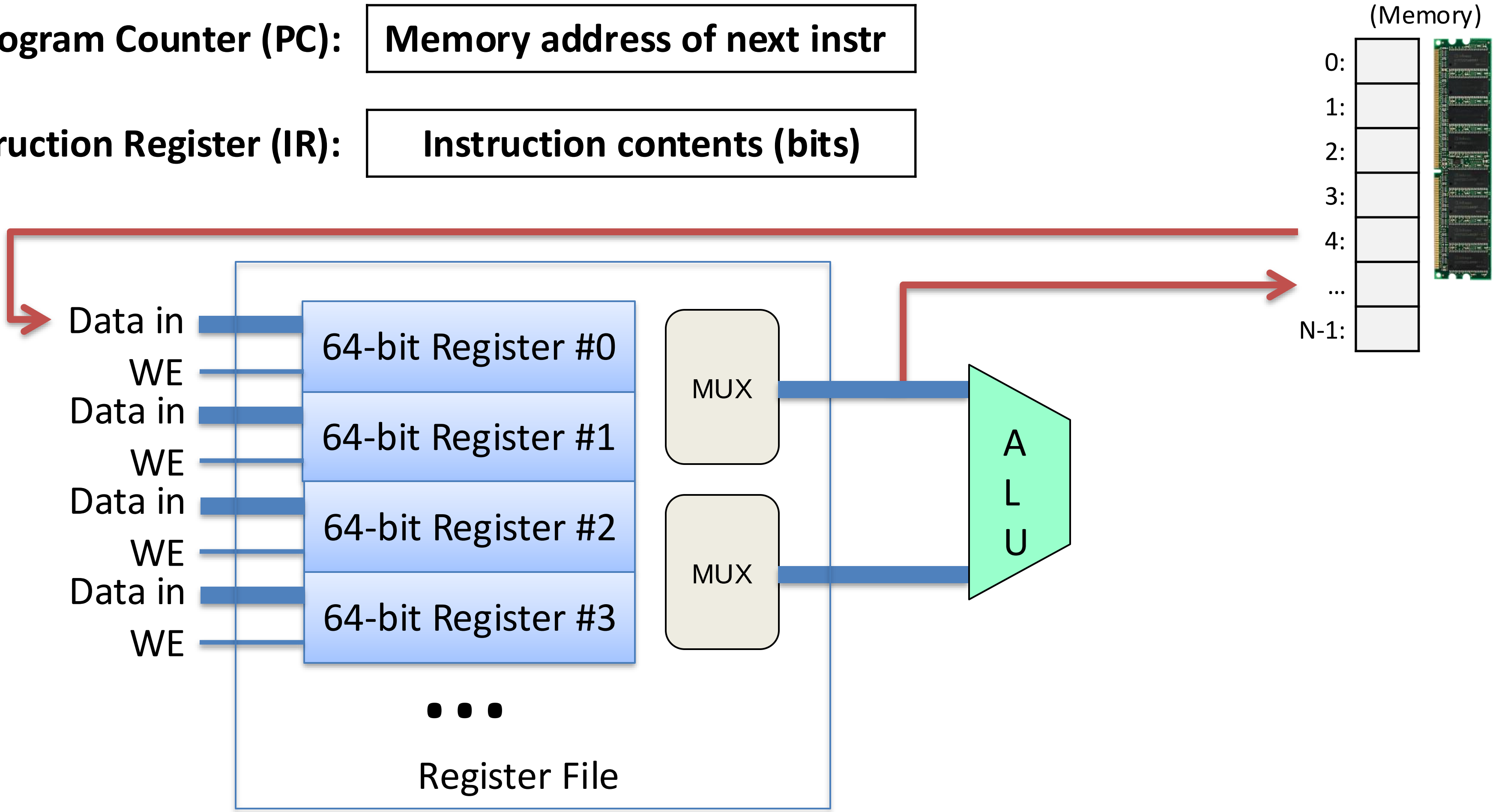
l -> 32 bits
q -> 64 bits

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)**



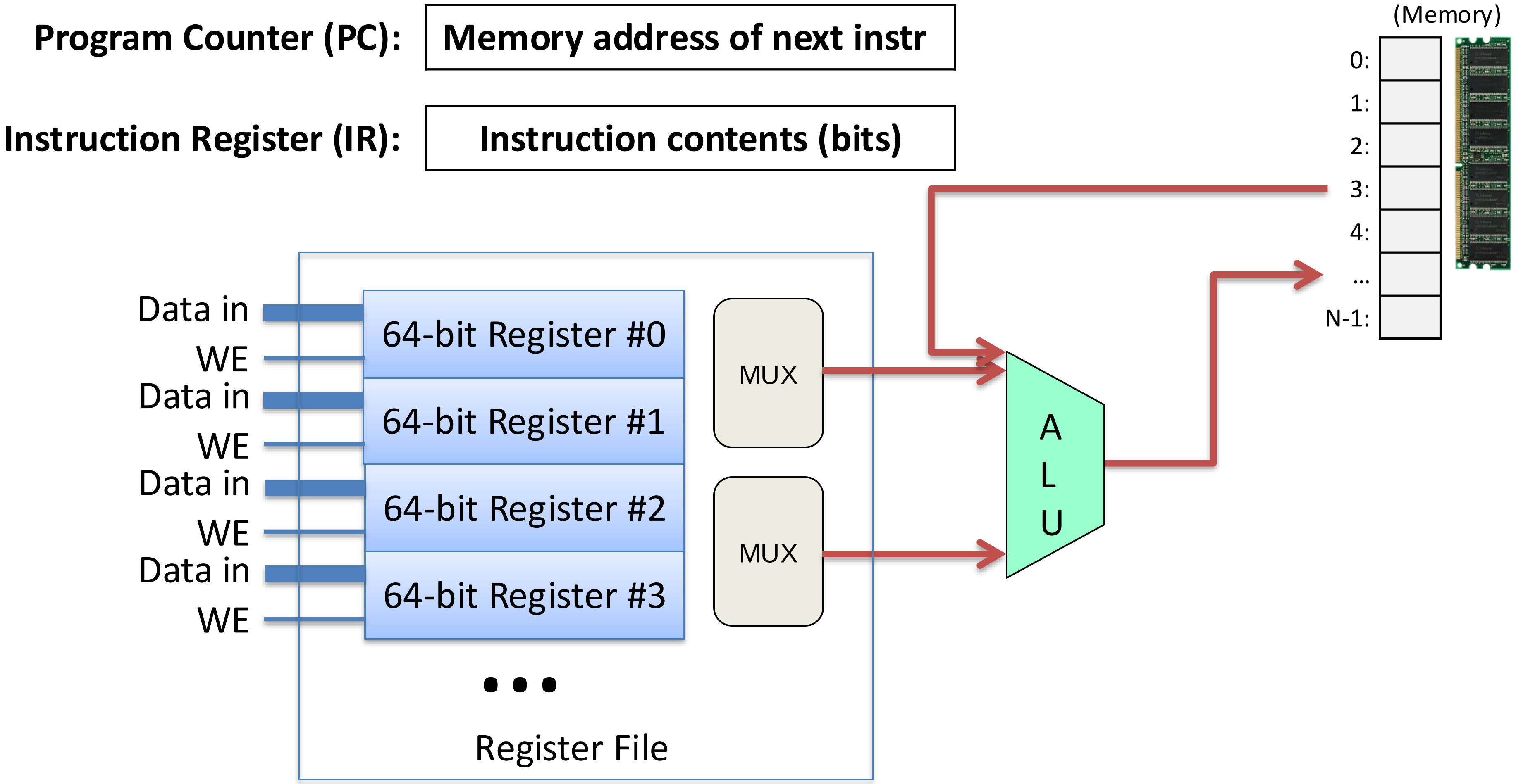
Types of assembly instructions

- Data movement
  - Move values between registers and memory

- Arithmetic
  - Uses ALU to compute a value
  - Examples: `addq, subq`

Arithmetic

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



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

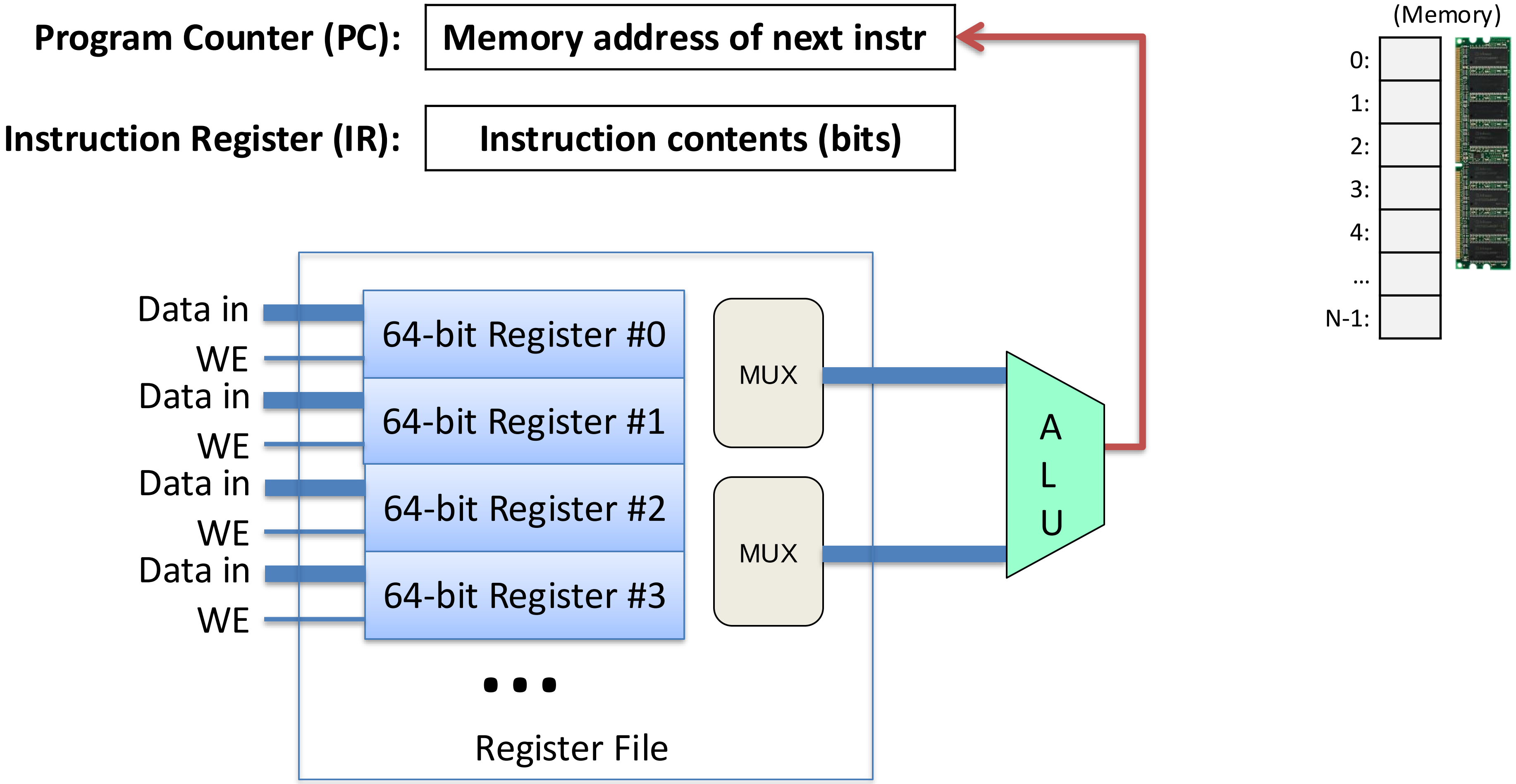
Instruction Register (IR): **Instruction contents (bits)**

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: `jmpq`

Control

Change PC based on ALU condition code state.



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
- 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

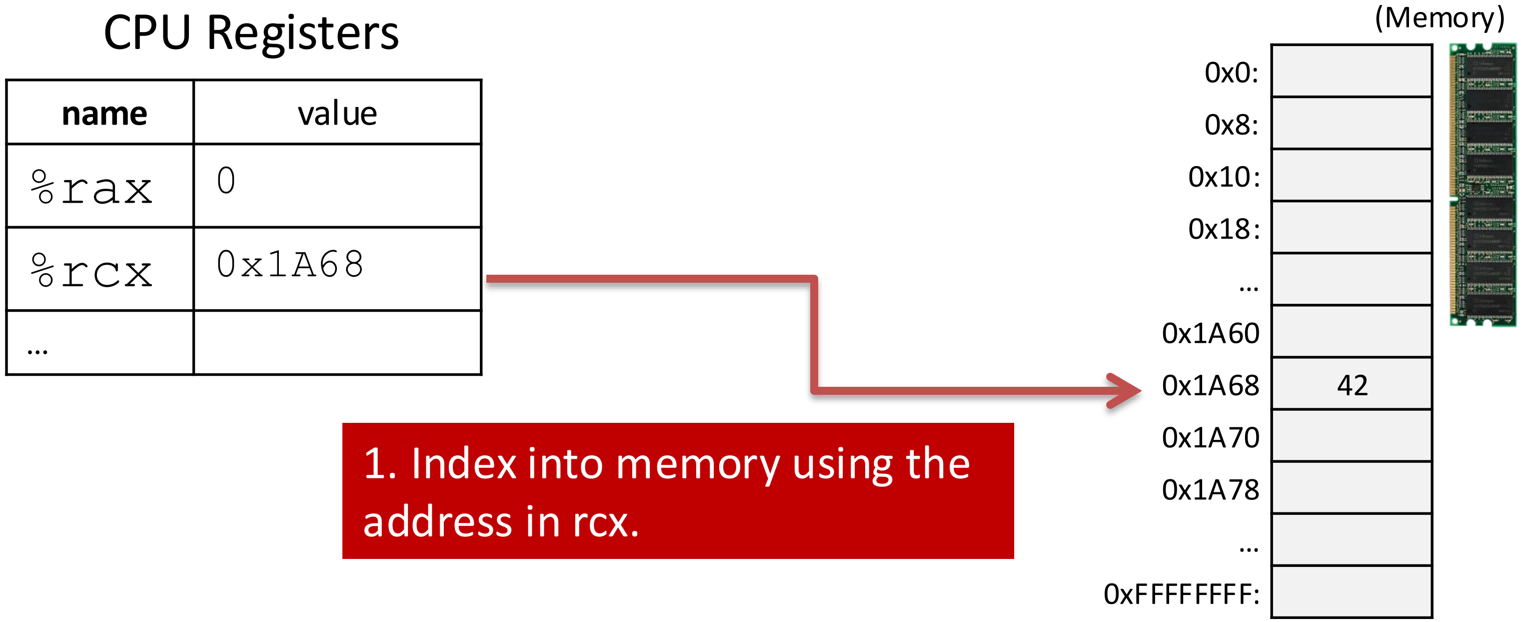
Addressing Modes

- Instructions can refer to:
  - the name of a register (%rax, %rbx, etc)
  - to a constant or “literal” value, starts with $
  - (%rax) : accessing memory
    - treat the value in %rax as a memory address,

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)

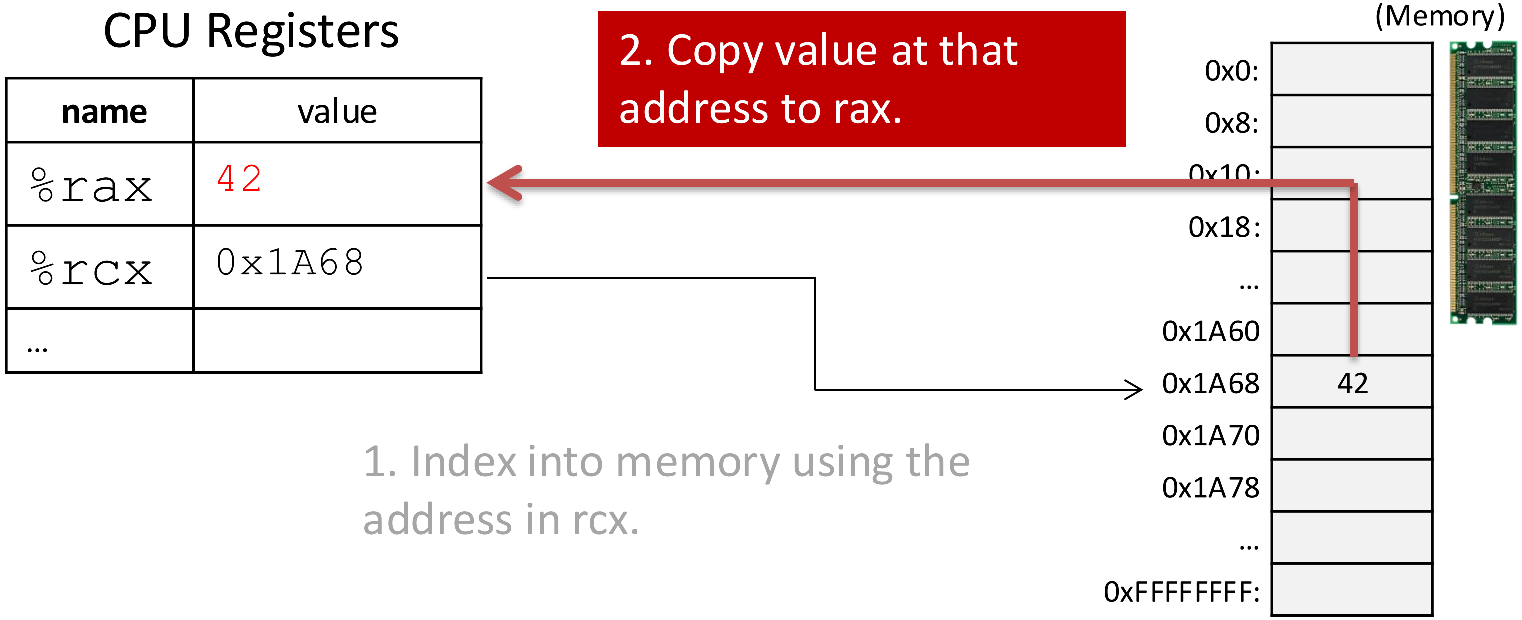
0x0:	
0x8:	
0x10:	
0x18:	
…	
0x1A60	
0x1A68	42
0x1A70	
0x1A78	
…	
0xFFFFFFFF:	

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



name	value
%rax	42
%rcx	0x1A68
...	

2. Copy value at that address to rax.

1. Index into memory using the address in rcx.

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

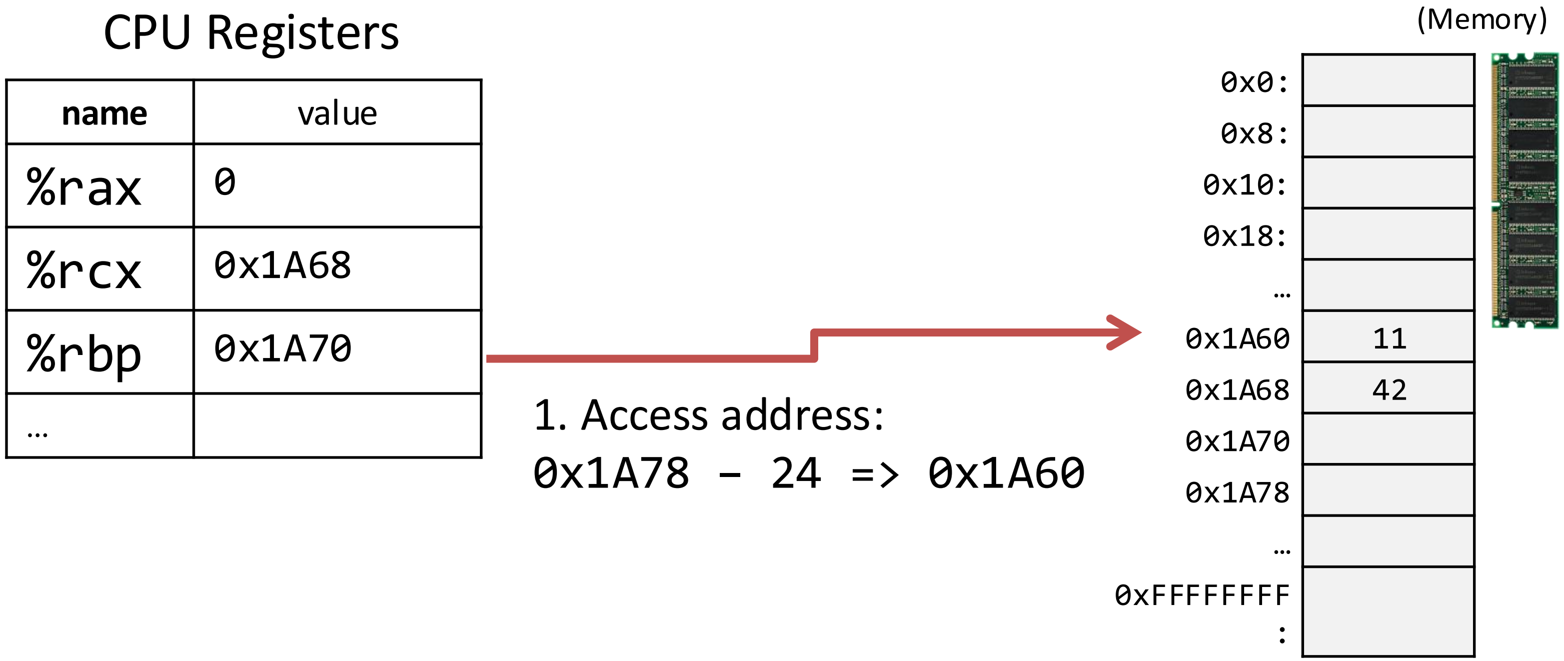
```
movq -16(%rbp), %rax
```

  - 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

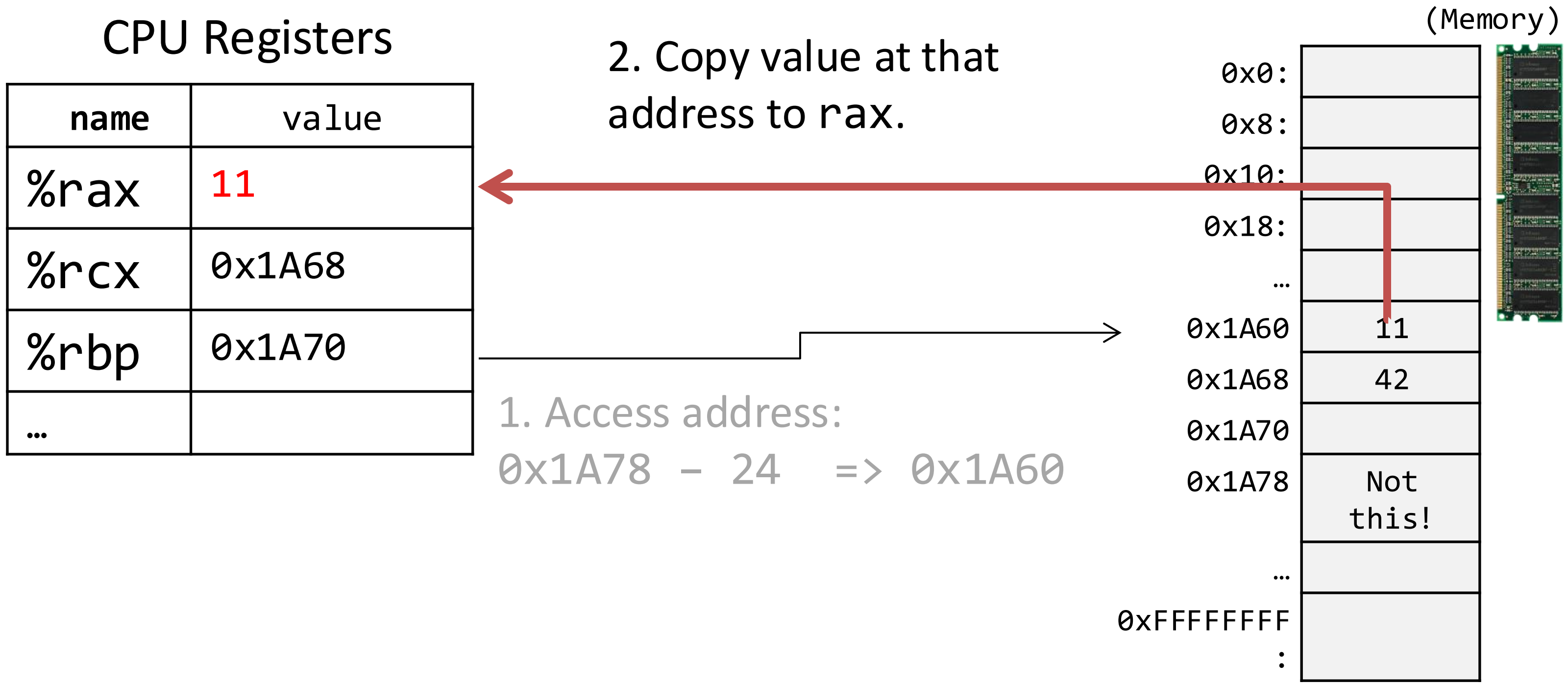
(Memory)

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

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	11
%rcx	0x1A68
%rbp	0x1A70
…	

2. Copy value at that address to `rax`.

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

(Memory)

0x0:	
0x8:	
0x10:	
0x18:	
…	
0x1A60	11
0x1A68	42
0x1A70	
0x1A78	Not this!
…	
0xFFFFFFFF:	

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?

```
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

<u>Registers</u>

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

<u>Memory</u>

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)
```

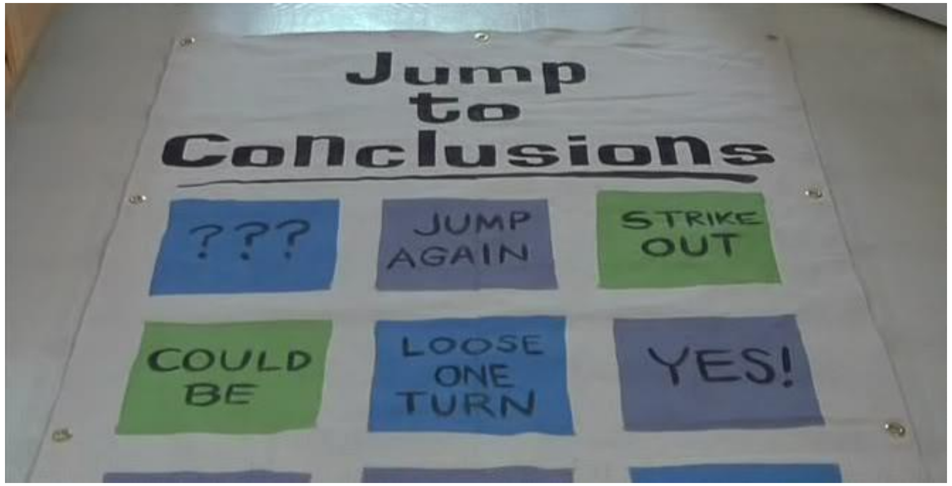
- x = y >> 3 | x * 8

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

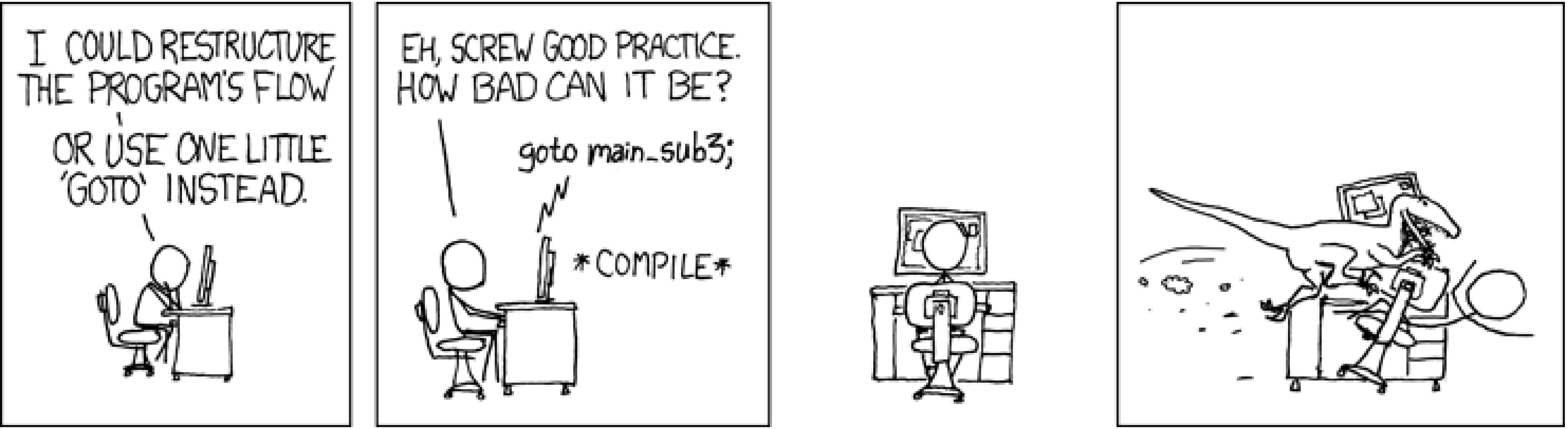
Control Flow

- 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

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);
```

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:	init code <fill in your answer here>

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

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 <u>might</u> jump to. Labels ignored except for goto/jumps)

<u>Try out this code: what does it do?</u>

```
  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)