CS 31: Introduction to Computer Systems

10: ISA and Assembly

02-20-2025

SWARTHMORE COLLEGE

Announcements

- Lab 3 Due Today. Please submit your lab questionnaire
- HW Groups will rotate after midterm 1 – Let me know your preferences!

Reading Quiz

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

<u>Check your frequency:</u>

- 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. C Pointers, and Main Memory
   - Parts of Program Memory
   - C's support for dynamic memory allocation
   - C pointer variables that refer to memory locations
   - Where are instructions, stack, etc., in program's memory space?

2. 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

Applications
Specific functionality

This week: Machine Interface

Operating system
Manage resources

Complex d
Compute

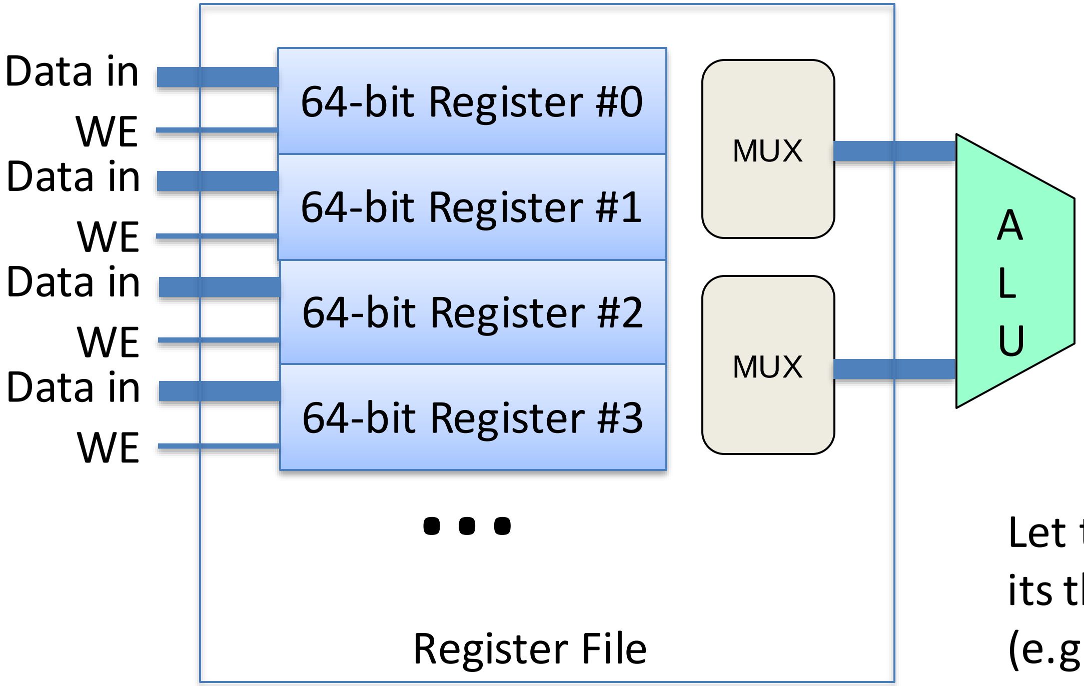
Last week: Circuits, Hardware Implementation

Hardware: Control, Storage, ALU circuitry

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

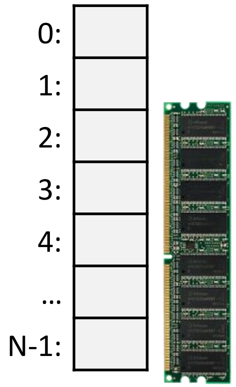
Instruction Register (IR): OP Code | Reg A | Reg B | Result

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

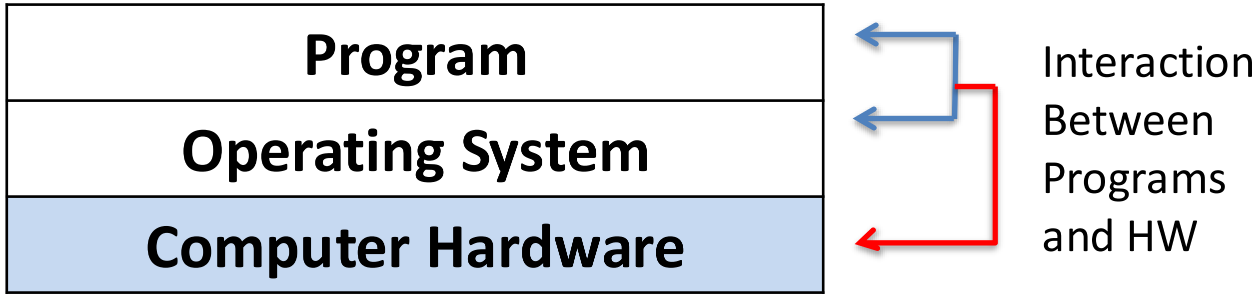


Let the ALU do its thing. (e.g., Add)

(Memory)

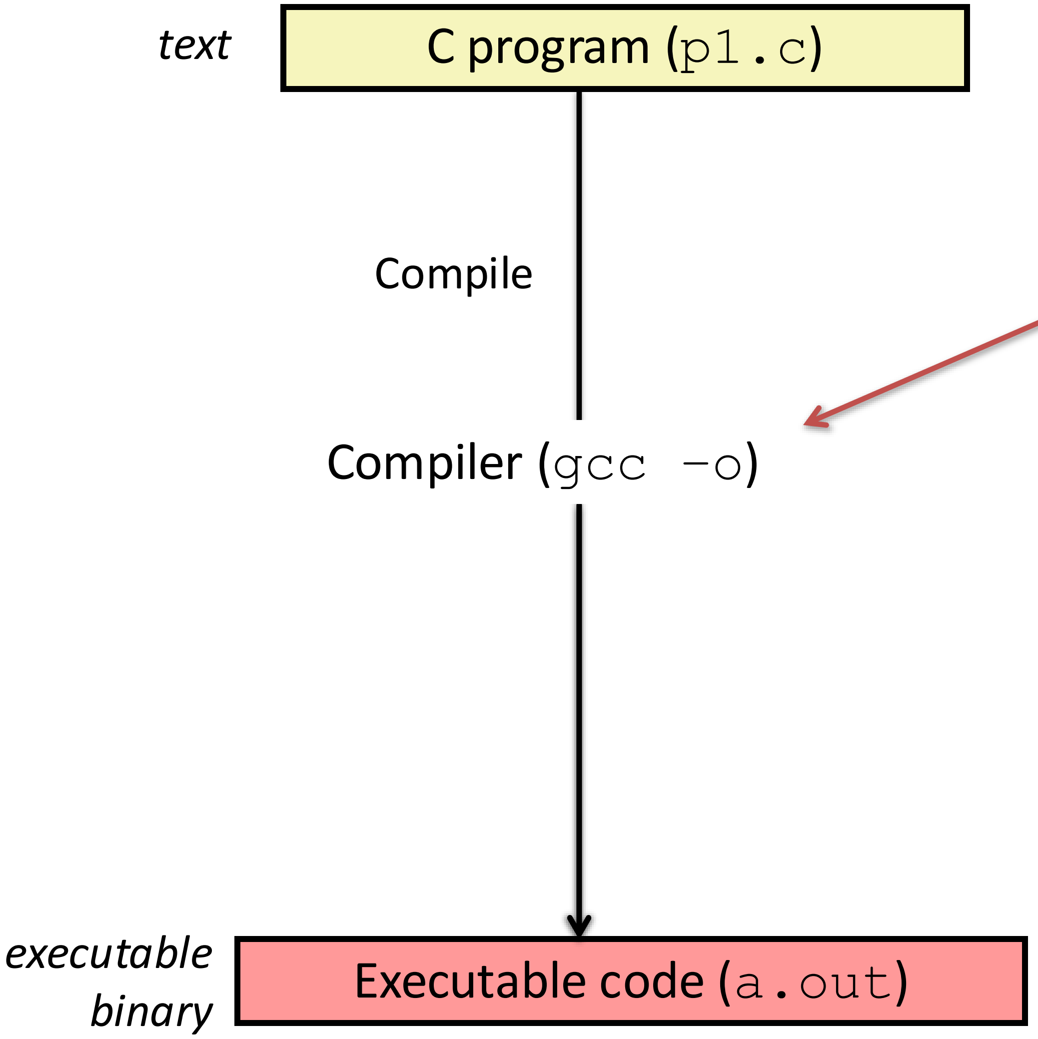


How a computer runs a program:



- 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)



Usually compile to a.out in a single step: gcc –m32 p1.c

-m32 tells gcc to compile for 32-bit Intel machines

Reality is more complex: there are intermediate steps!

machine code instructions

Compilation Steps (.c to a.out)

CS75

text — C program (`p1.c`)

↓

Compiler (`gcc -S`)

↓

text — Assembly program (`p1.s`)

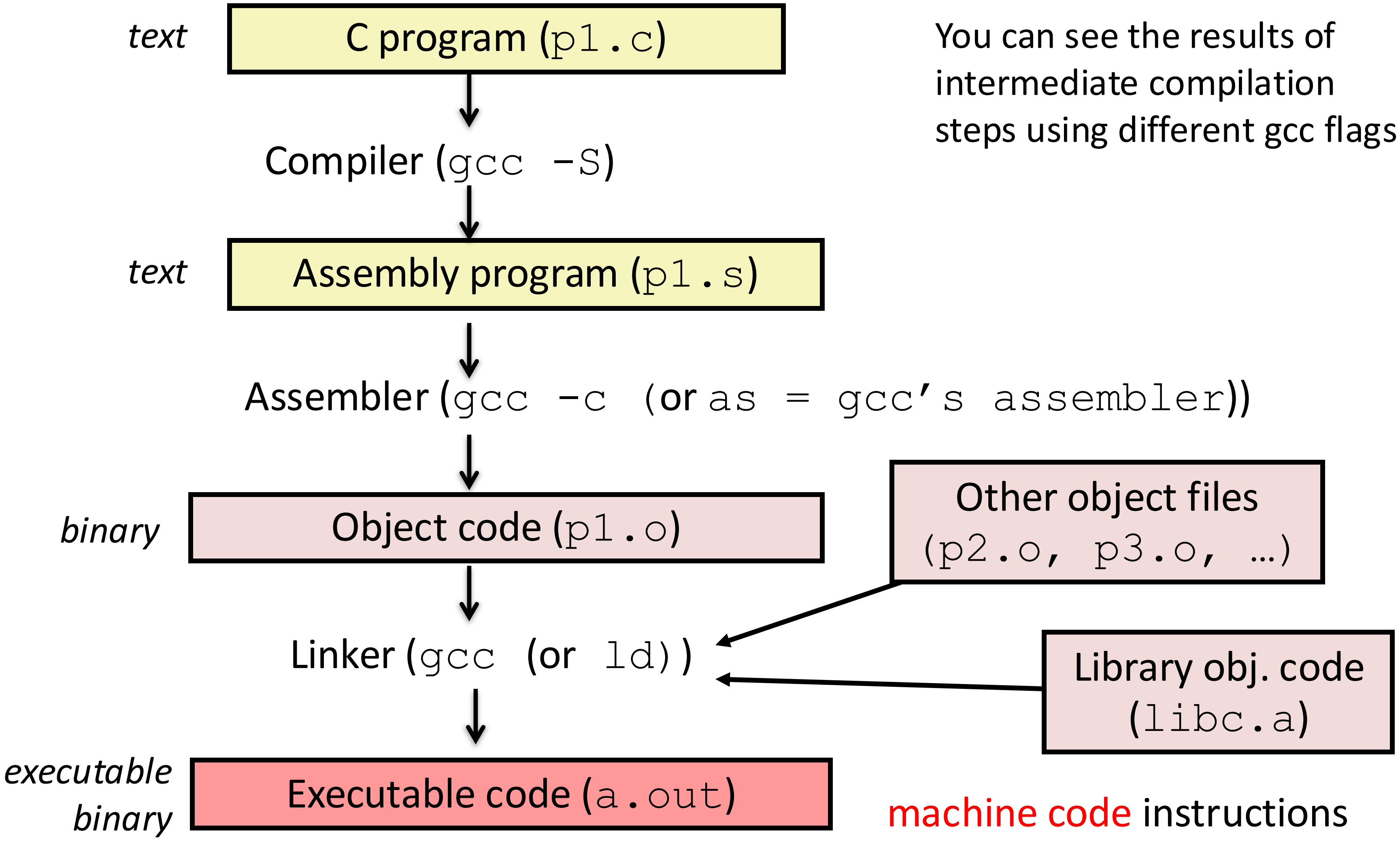
You can see the results of intermediate compilation steps using different gcc flags

executable binary — Executable code (`a.out`)

machine code instructions

Compilation Steps (.c to a.out)

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



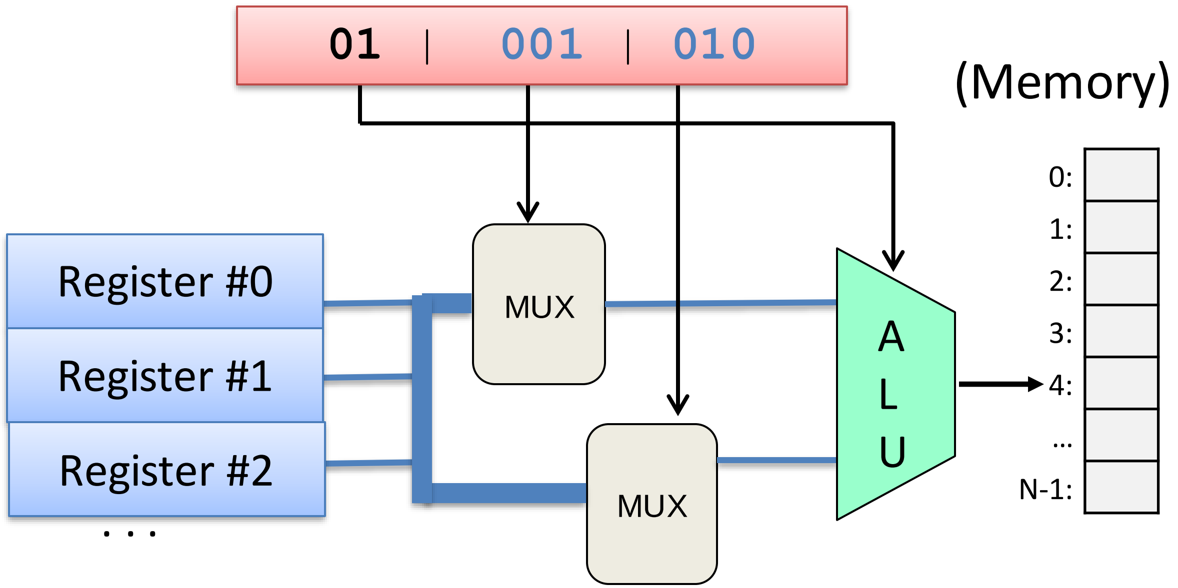
Machine Code

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

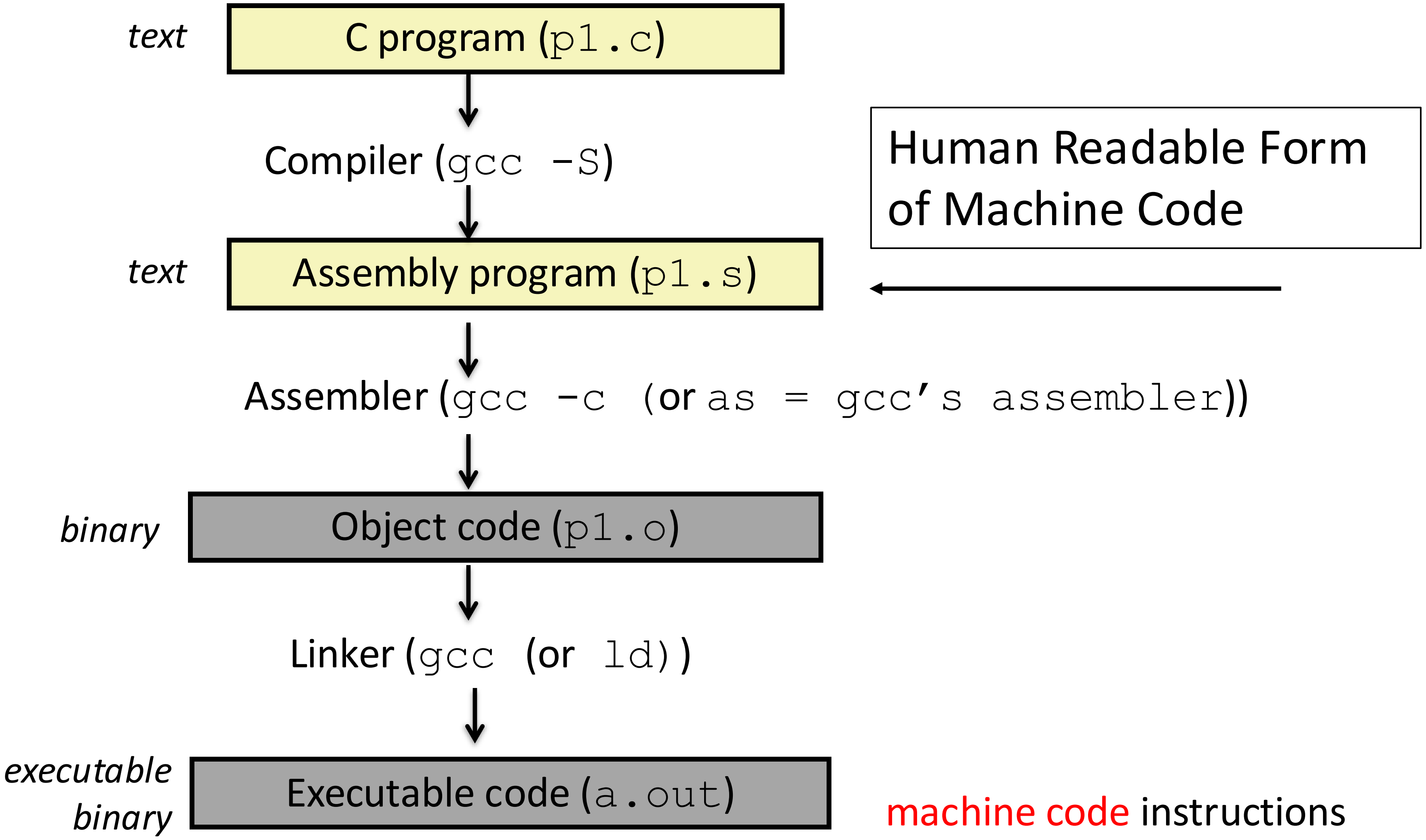
- some bits: encode the instruction (opcode bits)
- others encode operand(s)

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

- different bits fed through different CPU circuitry:



Assembly Code



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

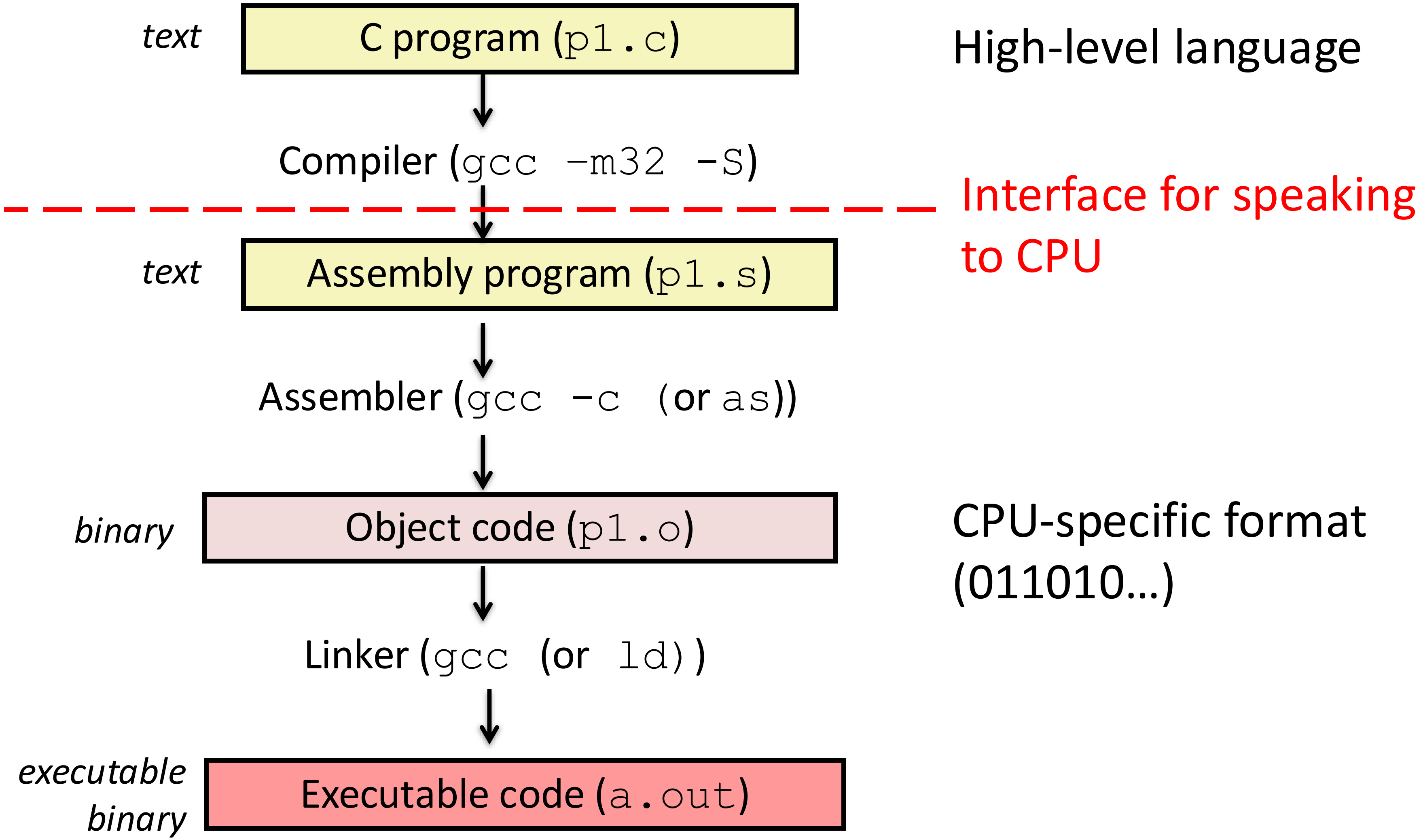
```
pushq %rbp
movq  %rsp,      %rbp
subq  $16,       %rsp
movq  $10,  -16(%rbp)
movq  $20,   -8(%rbp)
movq  -8(%rbp), $rax
addq  $rax,  -8(%rbp)
movq  -8(%rbp), %rax
leaveq
```

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

        a = a + b;

        return a;
}
```

Compilation Steps (.c to a.out)



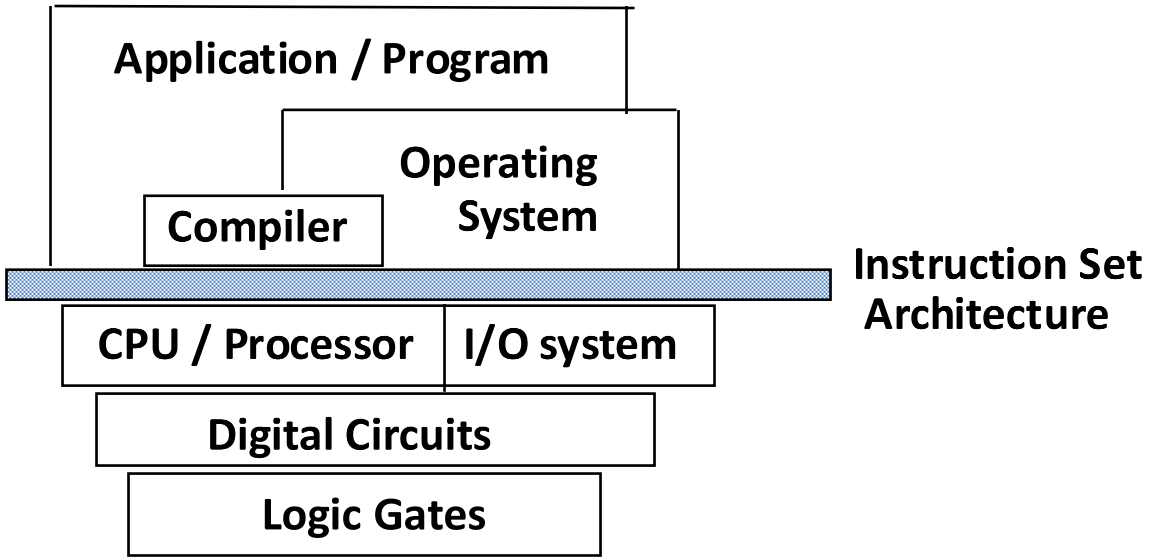
Instruction Set Architecture (ISA)

- 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)

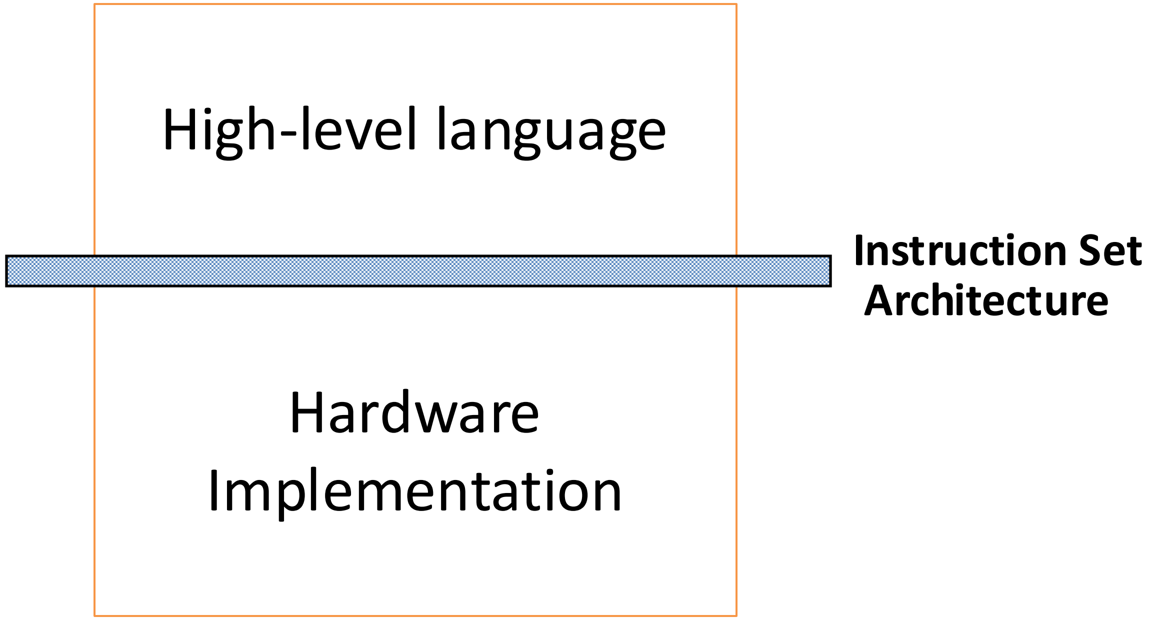
Instruction Set Architecture (ISA)

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



Instruction Set Architecture (ISA)

- The interface between lowest software level and the hardware.
- An 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

Intel x86 Family

Intel i386 (1985)

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

Intel Core i9 9900k (2018)

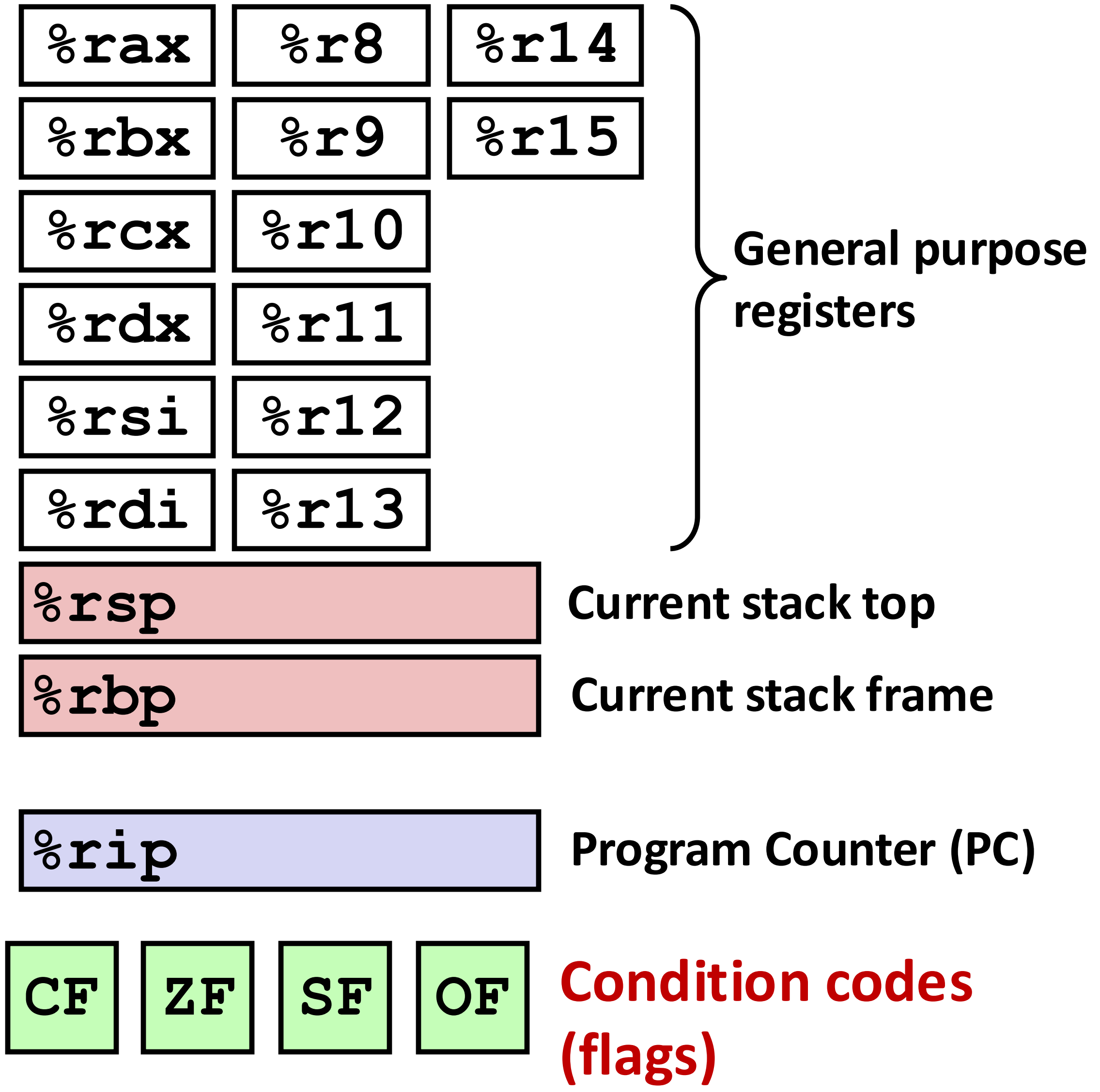
- ~4,000 MHz
- ~7,000,000,000 transistors
- Component size: 14 nm

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

Processor State in Registers

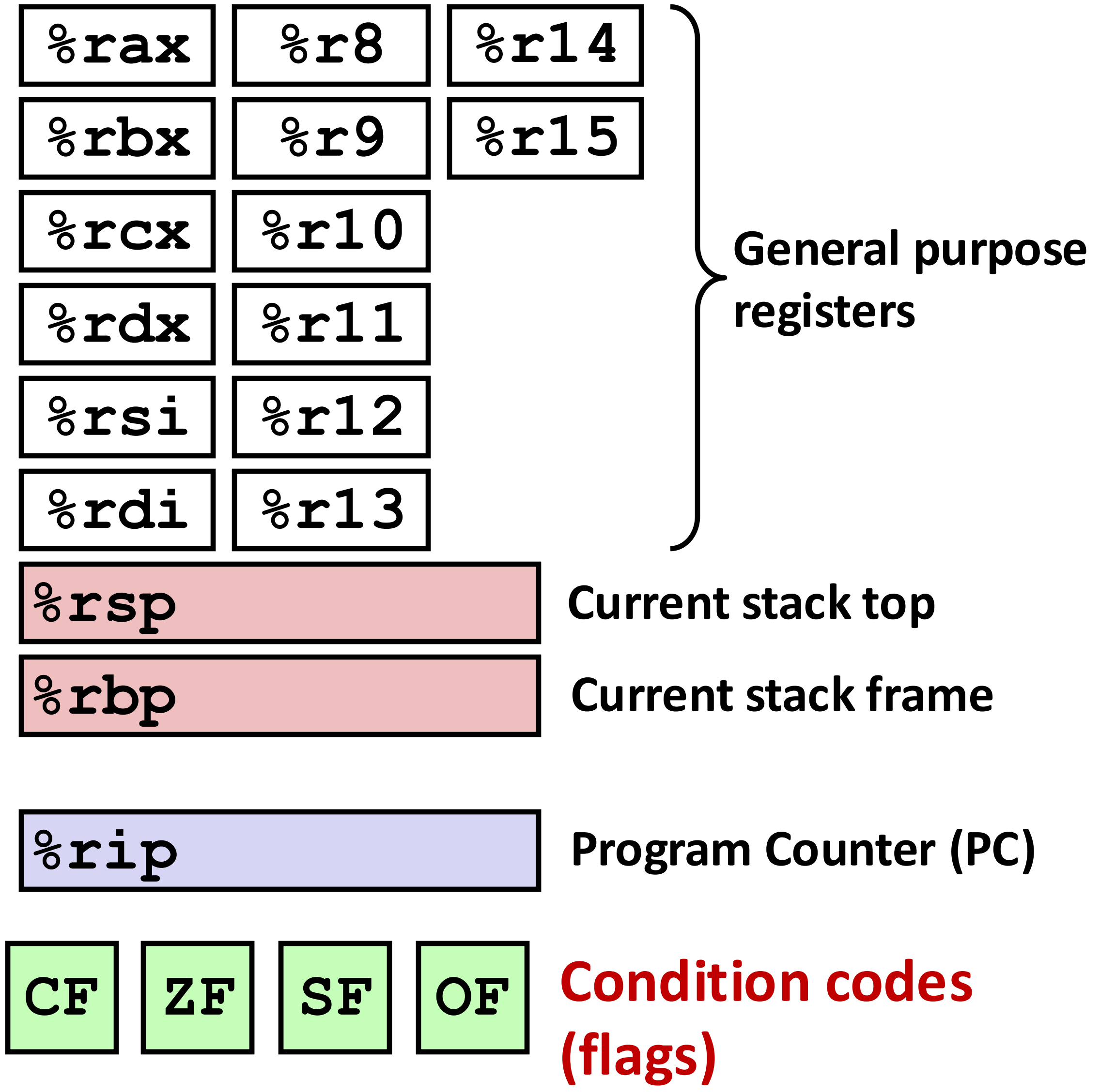
Working memory for currently executing program

- 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)

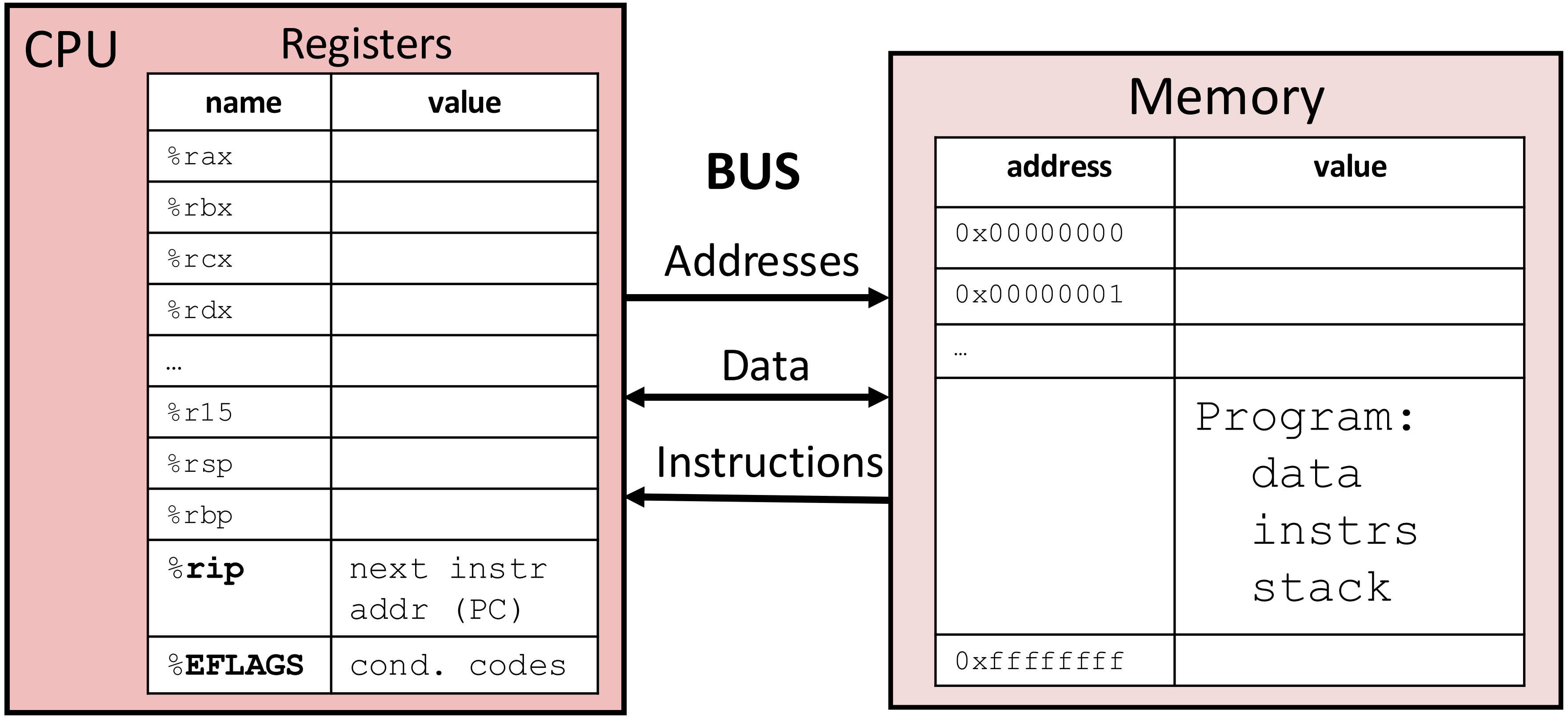


Component Registers

- 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



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

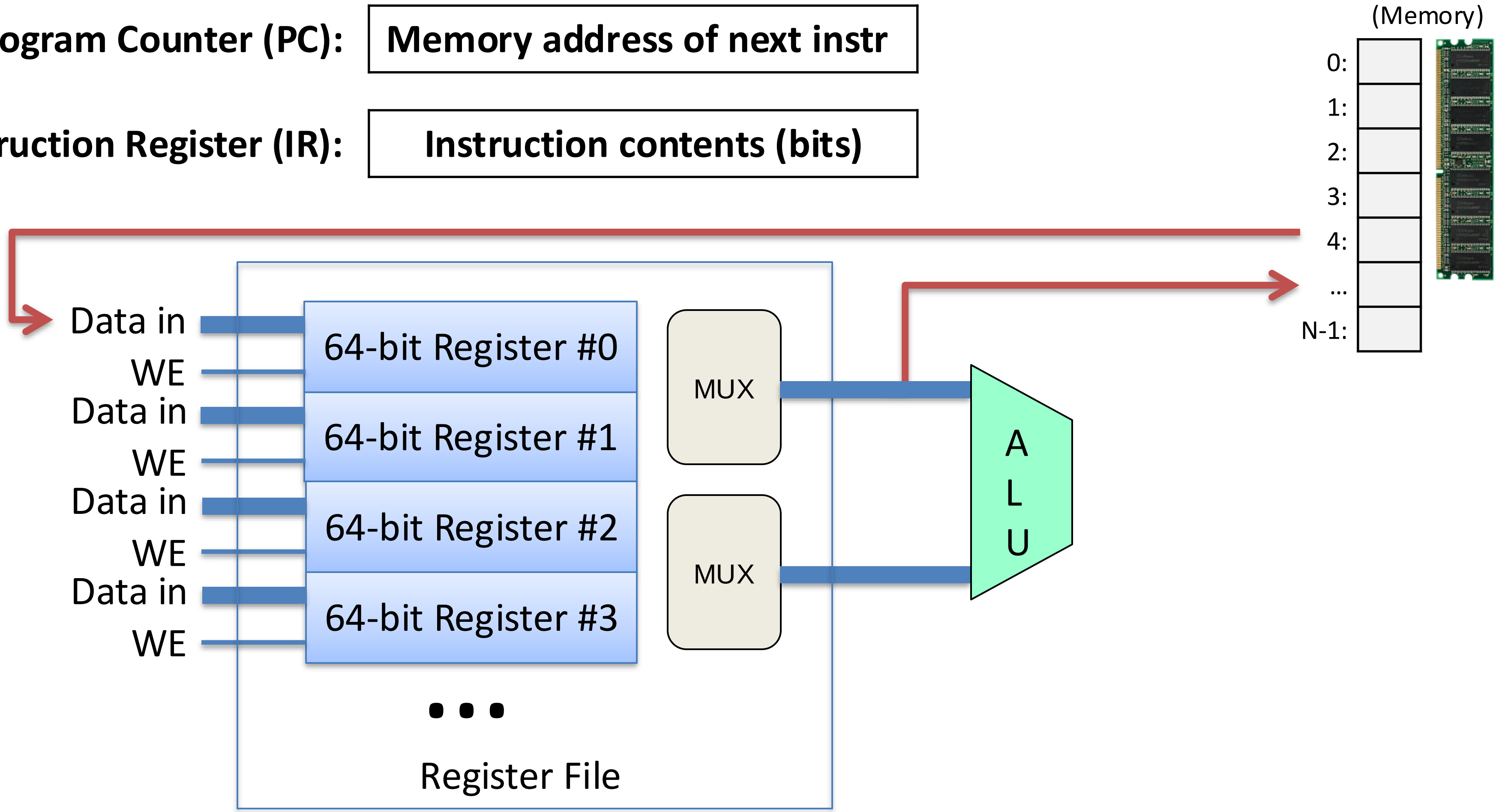
The suffix letters specify how many bytes to move (not always necessary, depending on context).

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)**

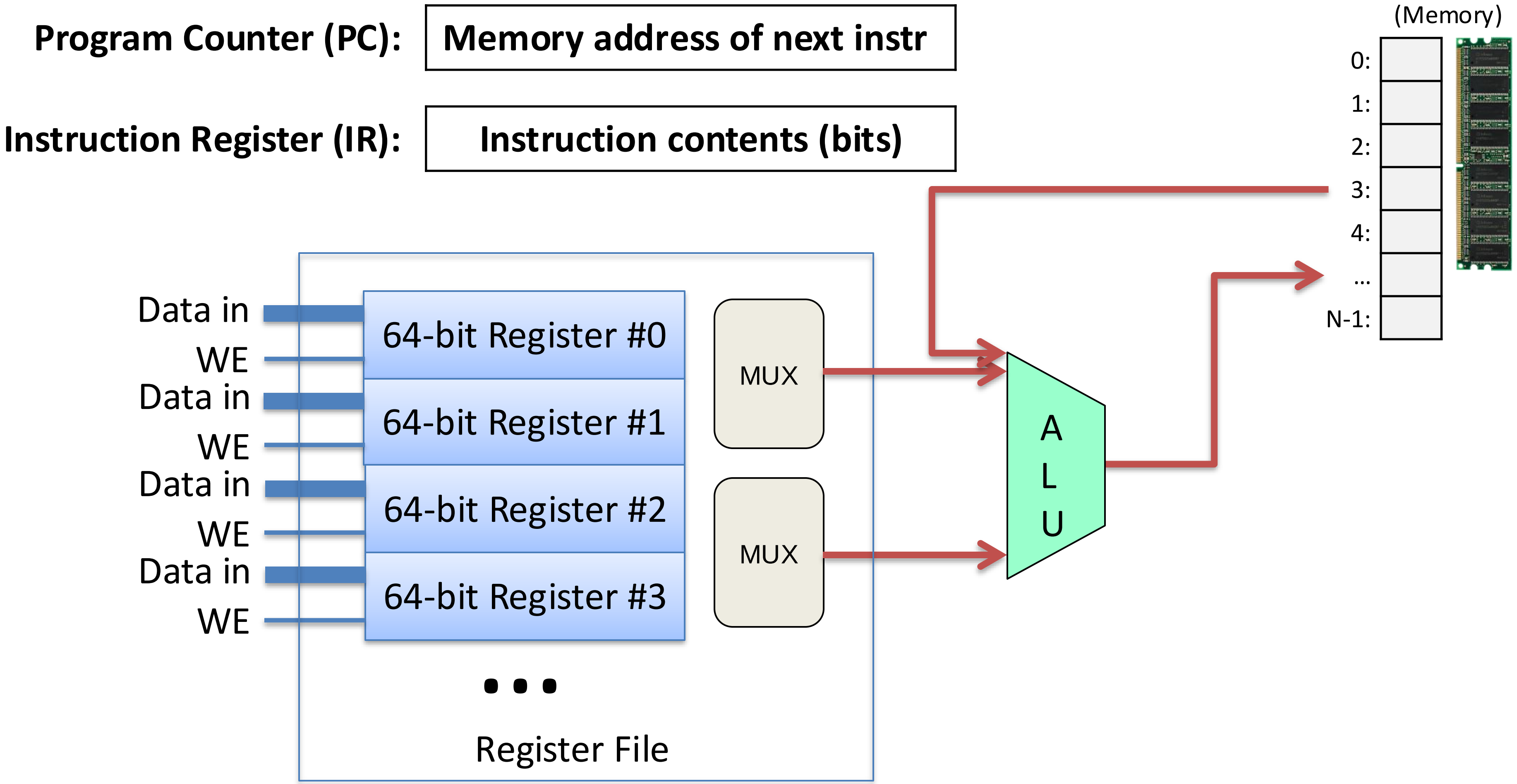


Types of assembly instructions

- 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.

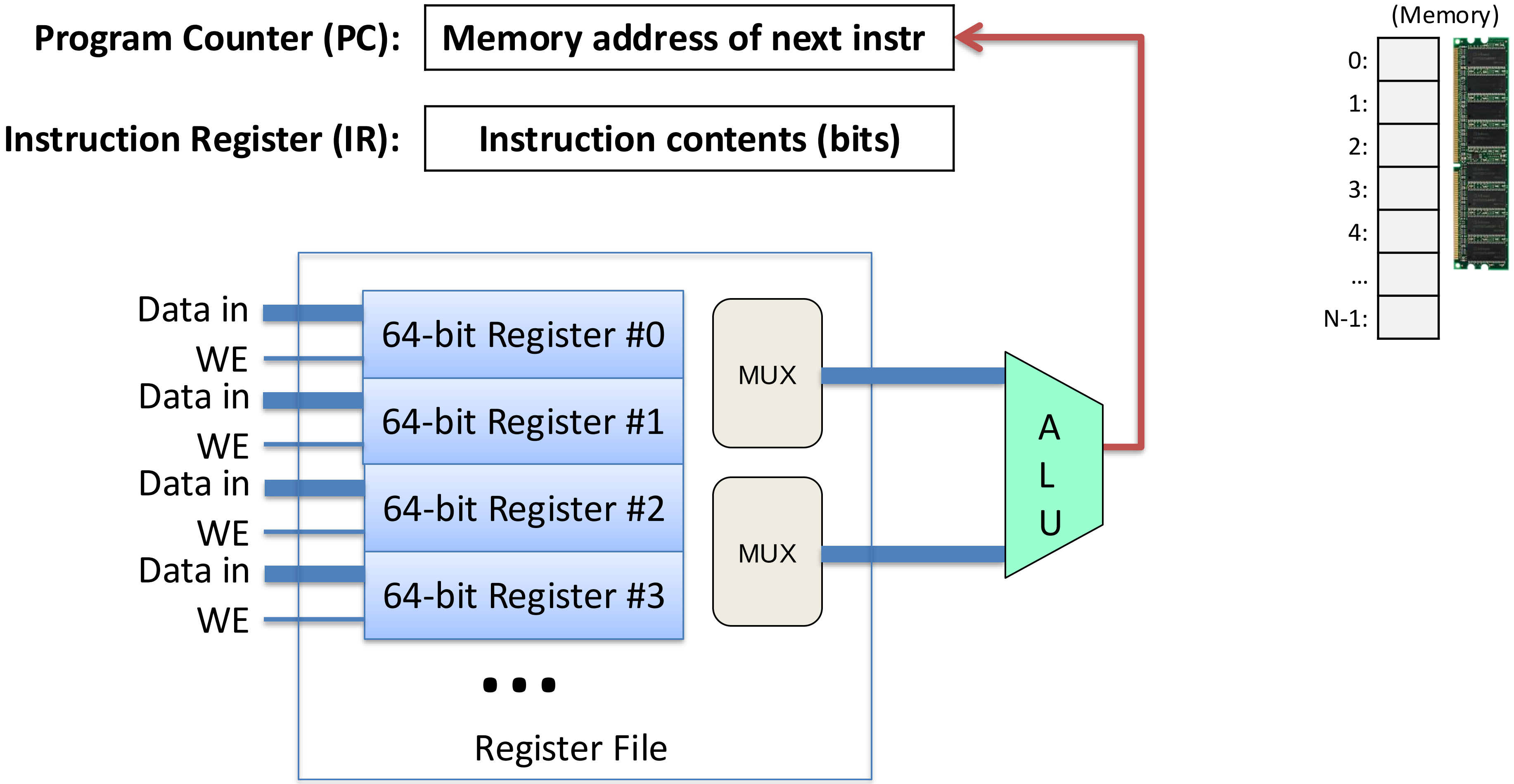


Types of assembly instructions

- 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.



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
```

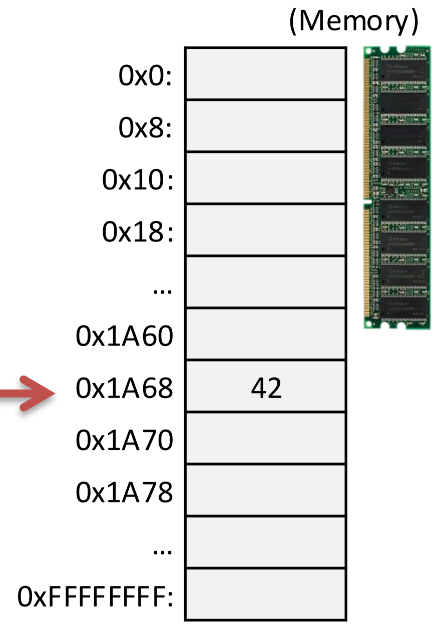
– Use the address in register %rcx to access memory,

– then, <u>*store result at that memory address*</u> in register %rax

CPU Registers

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

1. Index into memory using the address in rcx.

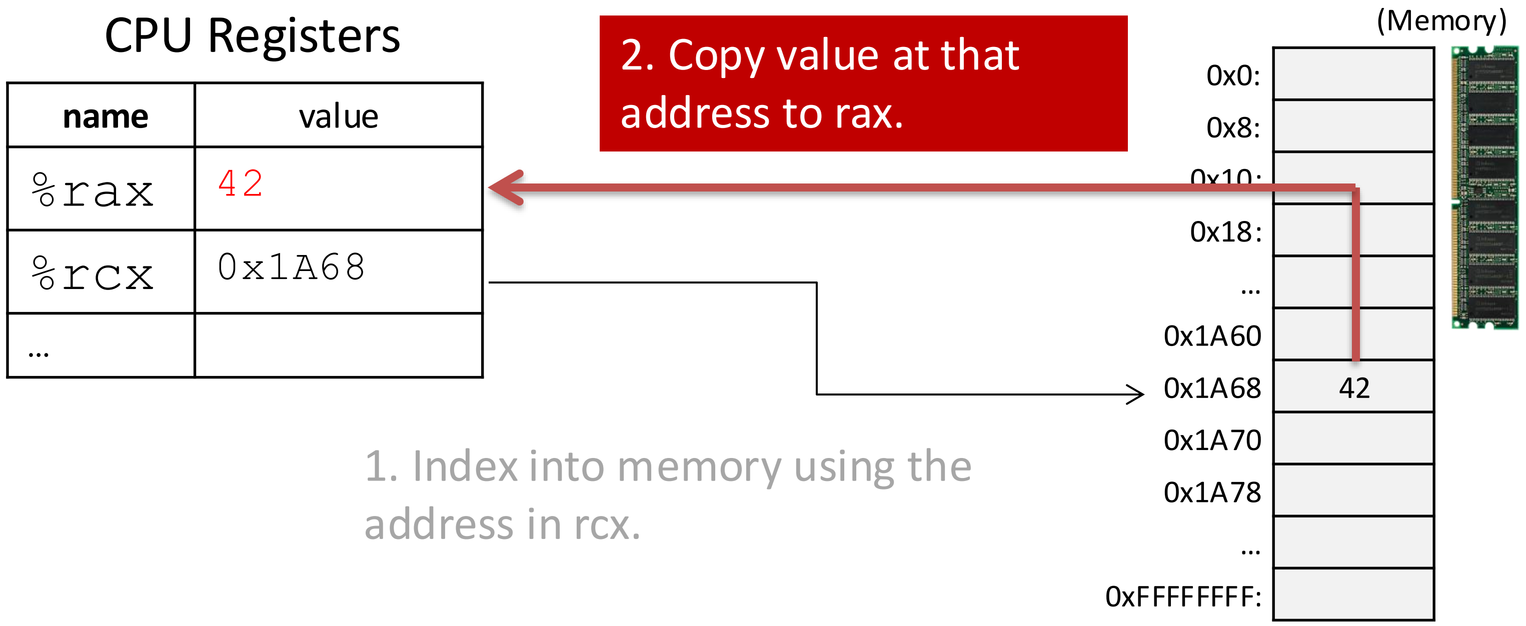


Addressing Mode: Memory

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

– Use the address in register %rcx to access memory,

– then, *store result at that memory address* in register %rax



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

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

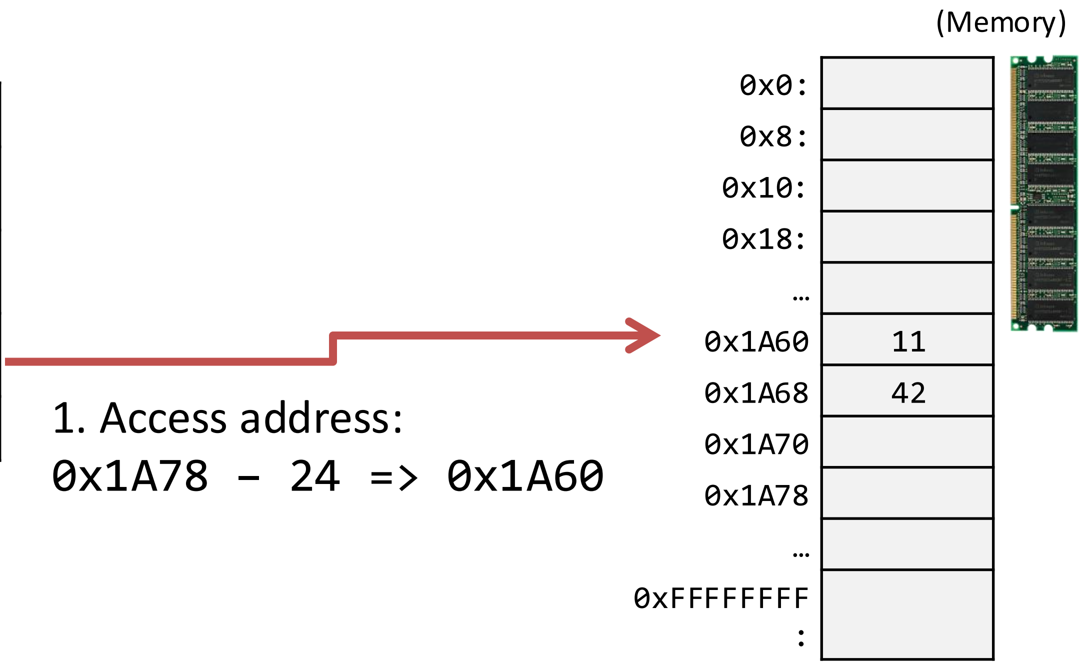
Addressing Mode: Displacement

`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
…	



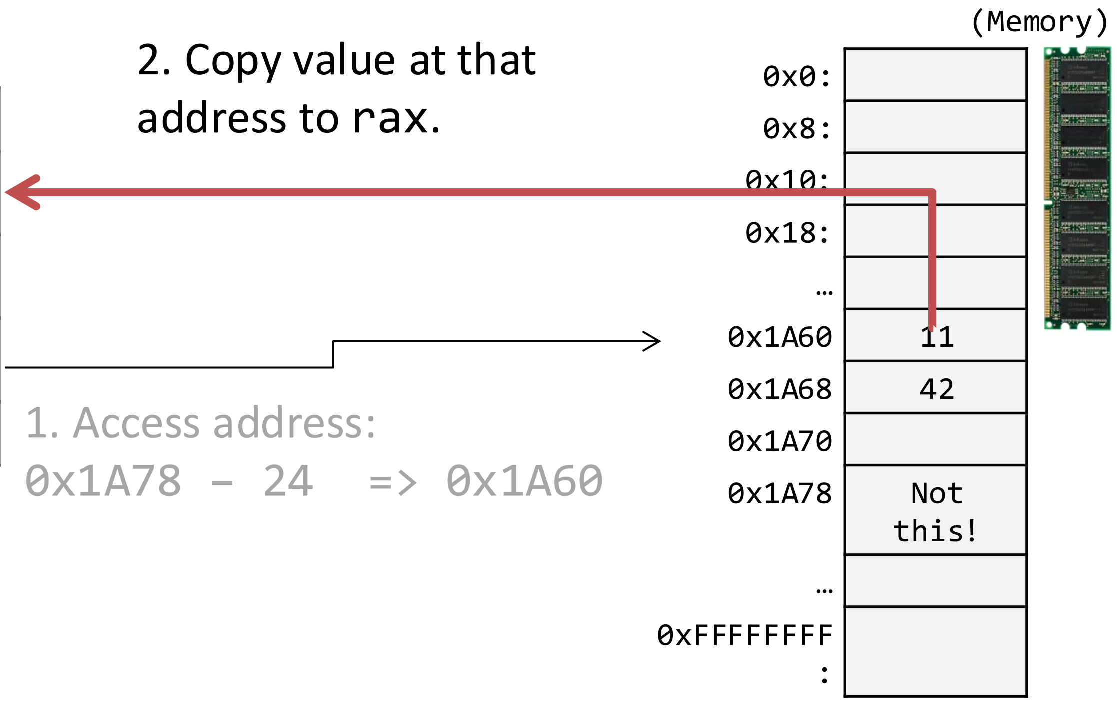
Addressing Mode: Displacement

```
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	11
%rcx	0x1A68
%rbp	0x1A70
…	



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	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.

```
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)
```