

CS 31: Introduction to Computer Systems

06: Computer Architecture

02-06-2025



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. Introduction to C

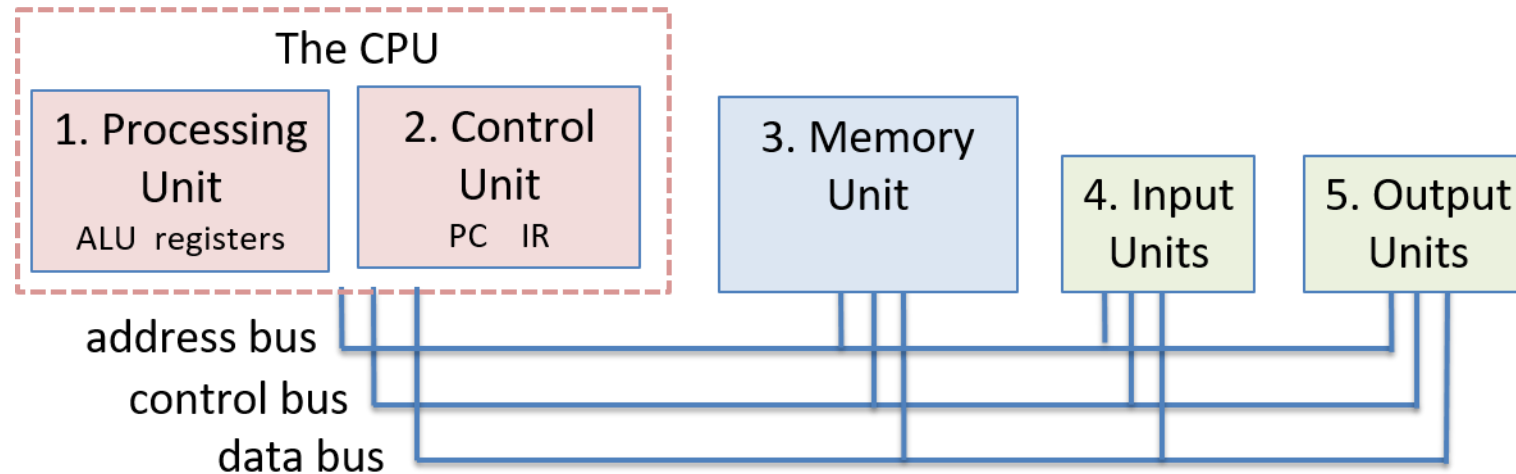
- Data organization and strings
- Functions

2. Computer Architecture

- Machine memory models
- Digital signals
- Logic gates

Von Neumann Model

5 units **connected** by buses (wires) to communicate



Processing & Control Units:

- implement CPU \execute program instructions on program data

Memory: stores program instructions and data

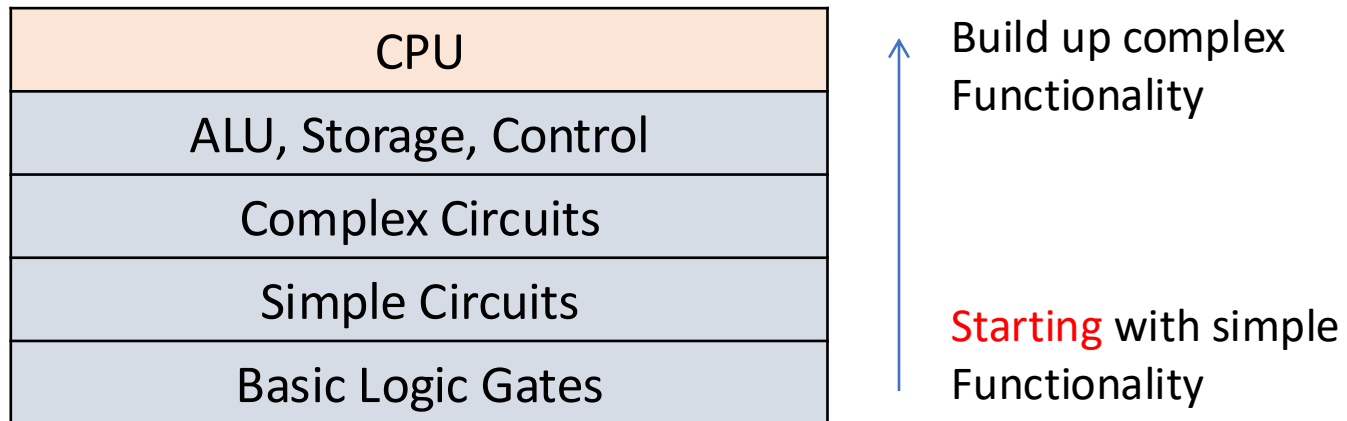
- memory is addressable: addr 0, 1, 2, ...

Input, Output: interface to compute

- trigger actions: load program, initiate execution, ...
- display/store results: to terminal, save to disk, ...

Our Goal: Build a CPU (model)

Start with very simple functionality, and add complexity

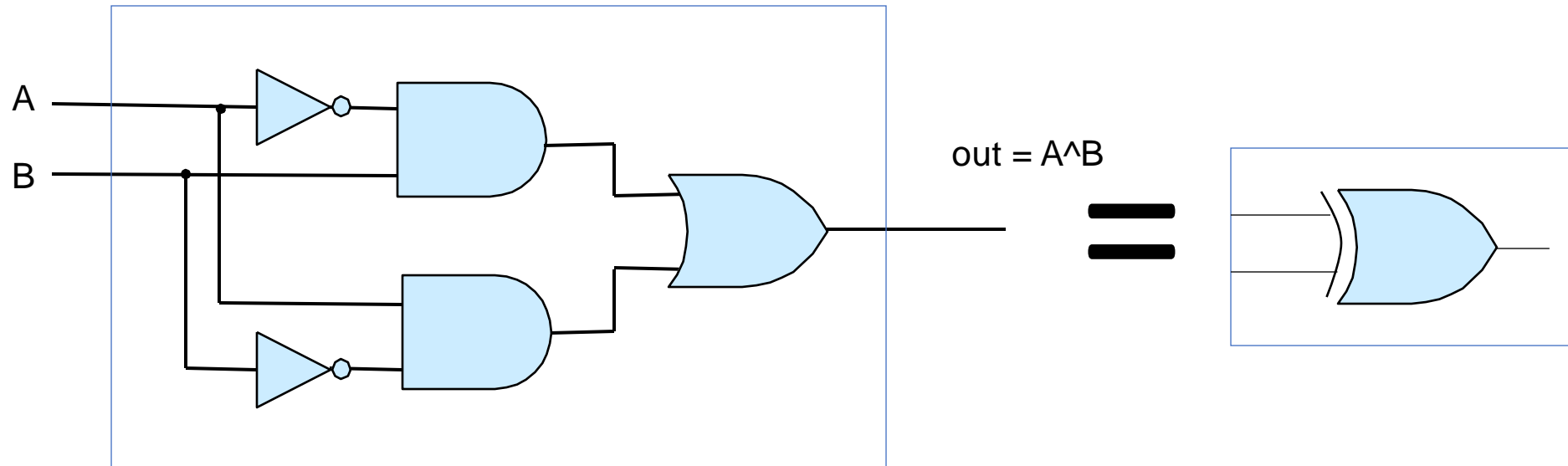


Abstraction!

- Hide away the complex internals of how the system functions, and focus on what functionality we expect. I.e., the guaranteed output of a system given the set of allowed inputs, and treating the functionality of the system as a black box.
- What are examples of abstractions you have experienced in daily life?

XOR Circuit: Abstraction

$$A \oplus B == (\sim A \ \& \ B) \ | \ (A \ \& \ \sim B)$$



A:0 B:0 A^B:

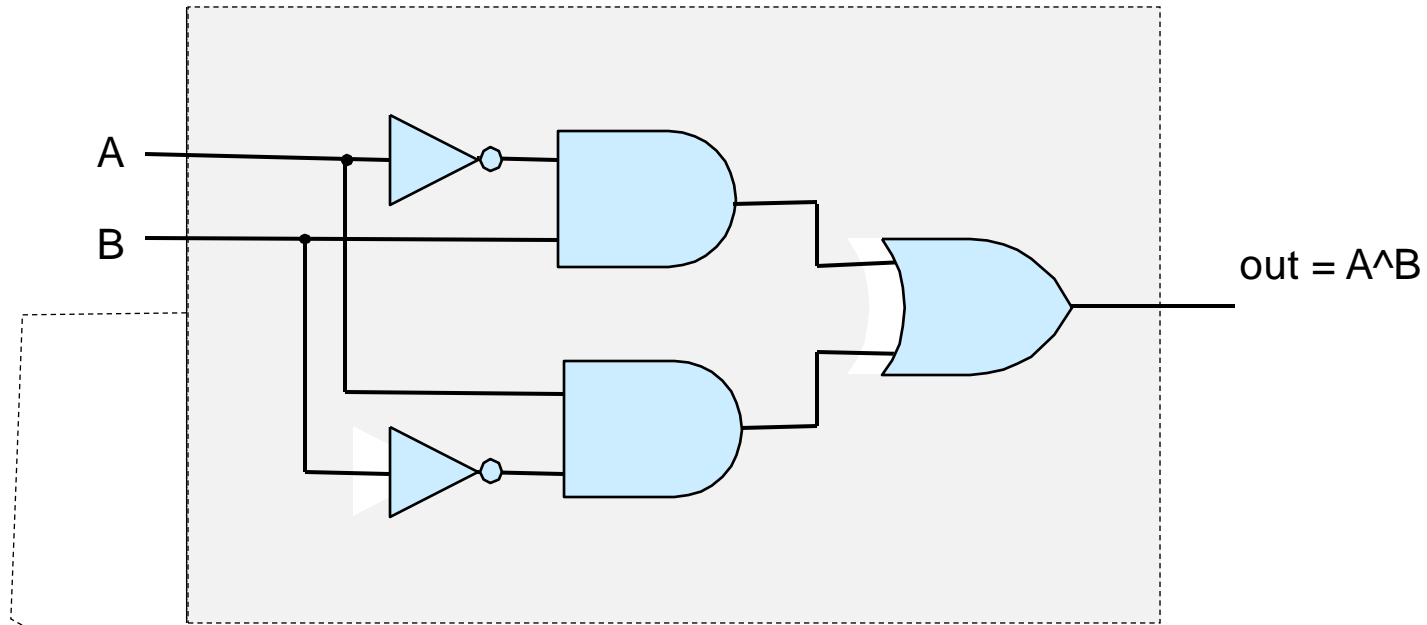
A:0 B:1 A^B:

A:1 B:0 A^B:

A:1 B:1 A^B:

XOR Circuit: Abstraction!

$$A \oplus B == (\sim A \ \& \ B) \ | \ (A \ \& \ \sim B)$$

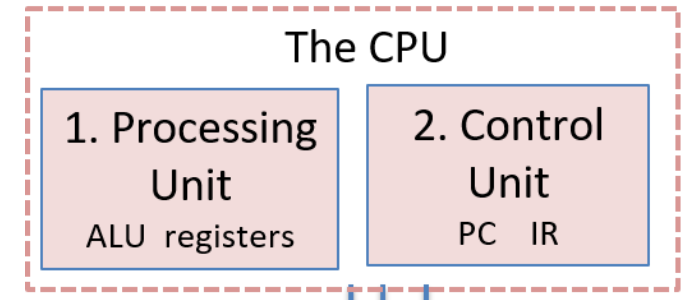


Treat XOR Circuit as a building block for other circuits!

Recall Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. ALU: implement arithmetic & logic functionality
 - Example: adder circuit to add two values together
2. Storage: to store binary values
 - Example: set of CPU registers (“register file”) to store temporary values
3. Control: support/coordinate instruction execution
 - Example: circuitry to fetch the next instruction from memory and decode it



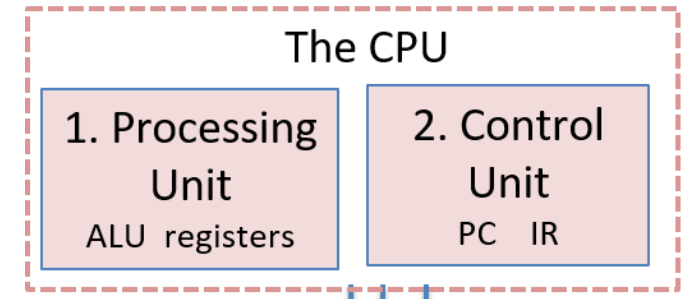
Recall Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. **ALU: implement arithmetic & logic functionality**
 - Example: adder circuit to add two values together

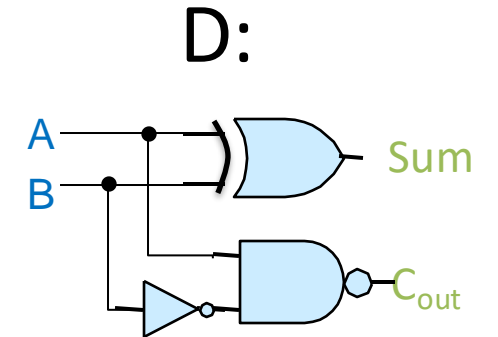
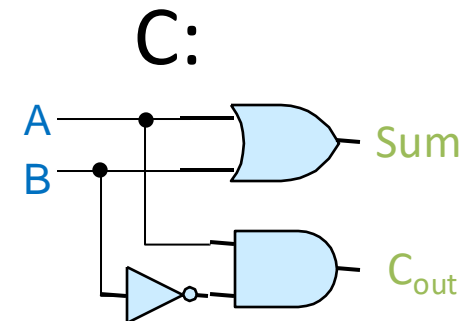
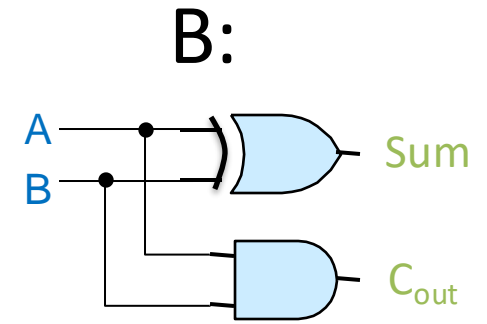
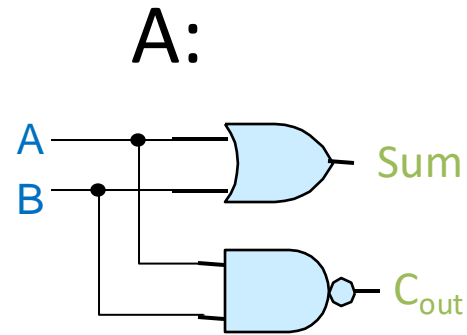
Start with ALU components (e.g., adder circuit, bitwise operator circuits)

Combine component circuits into ALU!



Which of these circuits is a one-bit adder?

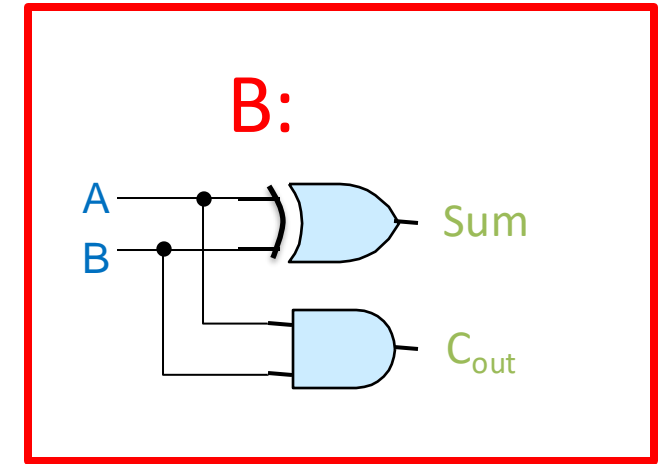
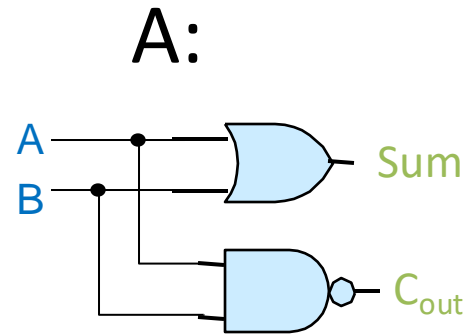
A	B	Sum (A + B)	C _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



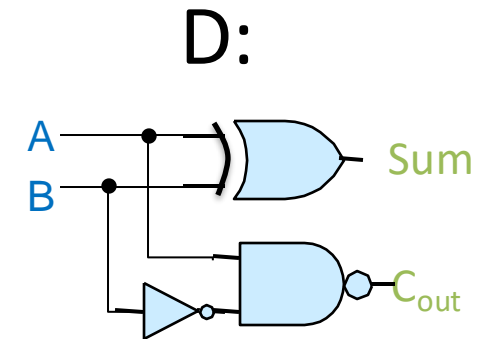
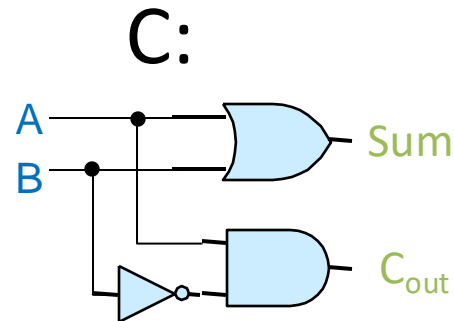
A	B	Sum (A + B)	C _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Which of these circuits is a one-bit adder?

A	B	Sum (A + B)	C _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



A	B	Sum (A + B)	C _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



More than one bit?

- When adding, sometimes have *carry in* too

$$\begin{array}{r} 1111 \\ 0011010 \\ + \underline{0001111} \end{array}$$

Write Boolean expressions for

$$\text{Sum} = 1 \text{ and } C_{\text{out}} = 1$$

When is Sum 1?

When is C_{out} 1?

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Write Boolean expressions for

Sum = 1 and $C_{out} = 1$

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

$$\sim C_{in} \ \& \ (A \wedge B) \ | \ C_{in} \ \& \ \sim (A \wedge B) \ == \ (C_{in} \ \wedge \ (A \wedge B))$$

When is C_{out} 1?

Write Boolean expressions for

Sum = 1 and $C_{out} = 1$

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

$$\sim C_{in} \ \& \ (A \wedge B) \ | \ C_{in} \ \& \ \sim (A \wedge B) \ == \ (C_{in} \ \wedge \ (A \wedge B))$$

When is C_{out} 1?

$$(A \ \& \ B) \ | \ ((A \wedge B) \ \& \ C_{in})$$

Write Boolean expressions for

Sum = 1 and $C_{out} = 1$

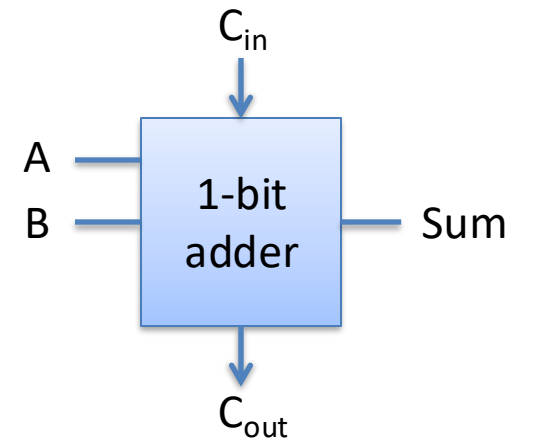
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

$$\sim C_{in} \ \& \ (A \wedge B) \ | \ C_{in} \ \& \ \sim (A \wedge B) \ == \ (C_{in} \ \wedge \ (A \wedge B))$$

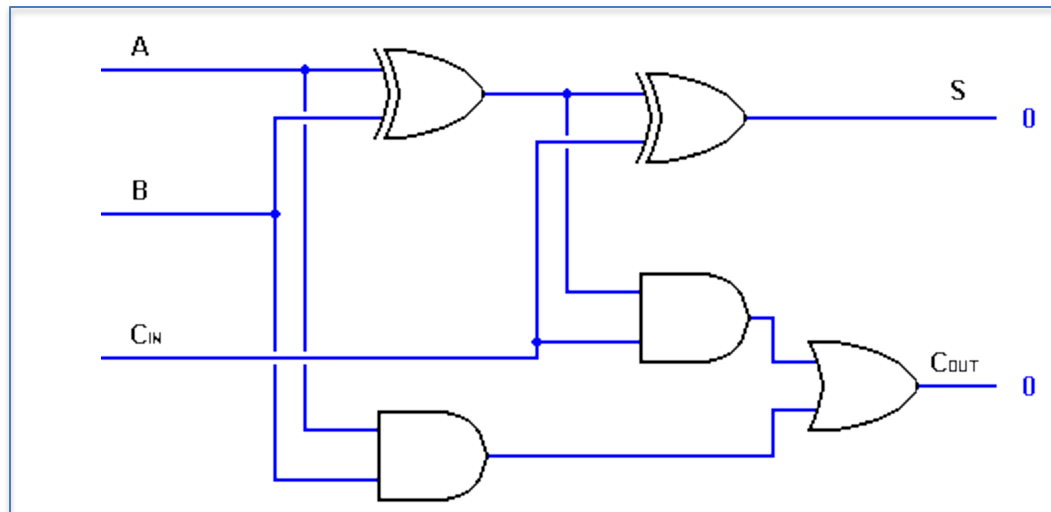
When is C_{out} 1?

$$(A \ \& \ B) \ | \ ((A \wedge B) \ \& \ C_{in})$$

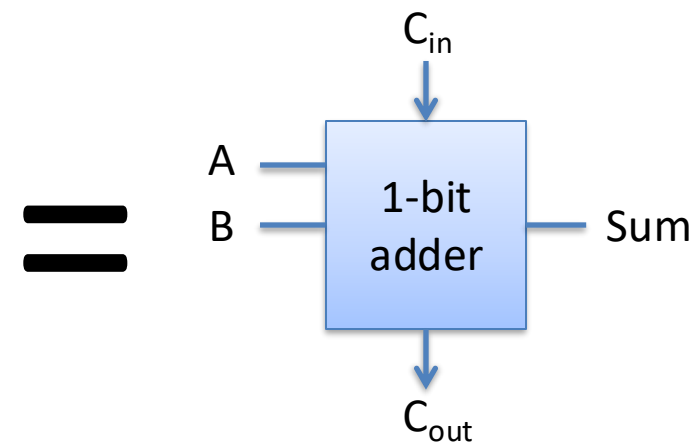


One-bit (full) adder

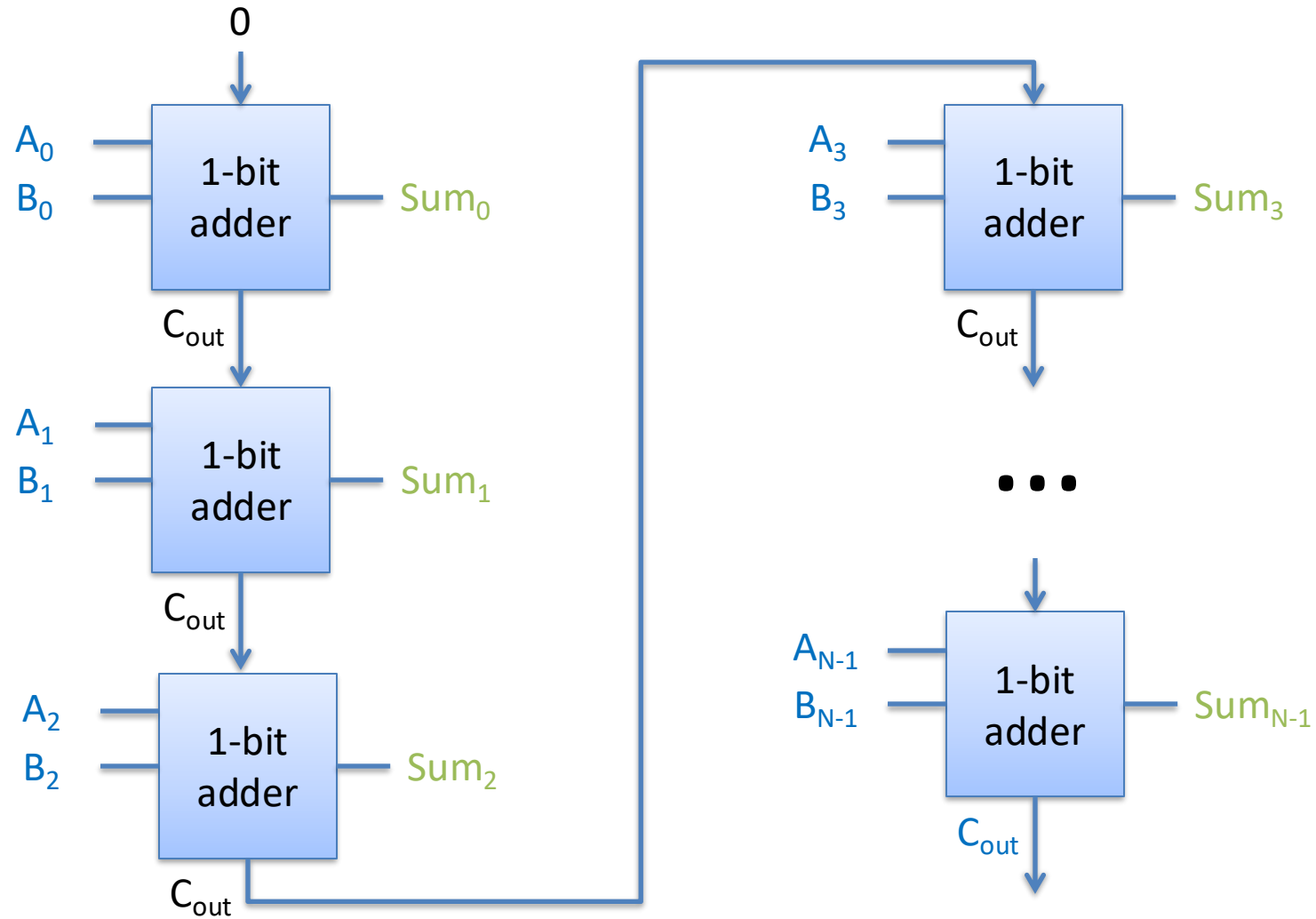
- Need to include:
carry-in and **carry-out**



A	B	C _{in}	Sum	C _{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

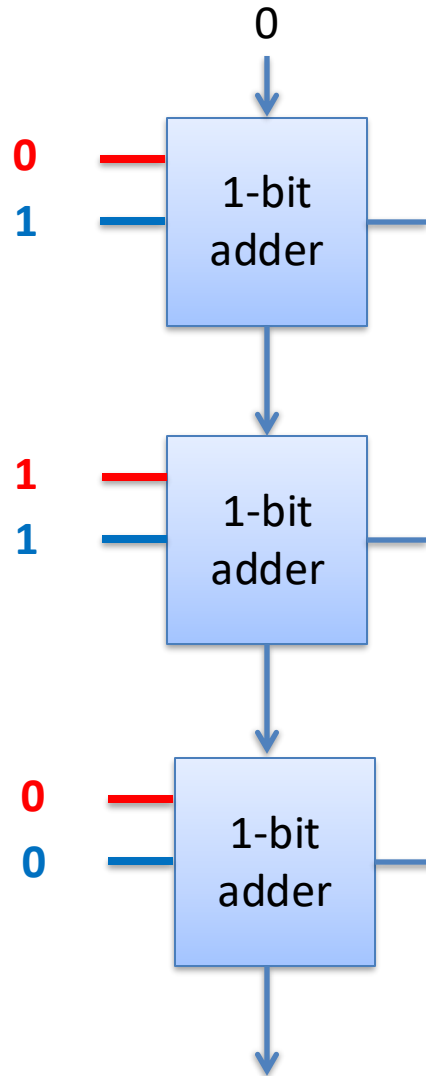


Multi-bit Adder (Ripple-carry Adder)

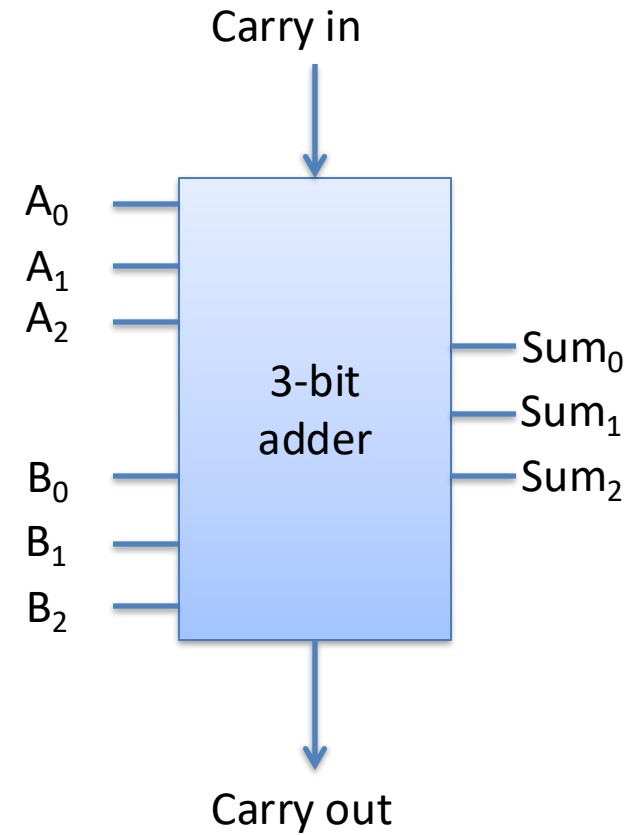


Three-bit Adder (Ripple-carry Adder)

$$\begin{array}{r} 010 \\ + 011 \\ \hline \end{array}$$



=



Arithmetic Logic Unit (ALU)

- One component that knows how to manipulate bits in multiple ways
 - Addition
 - Subtraction
 - Multiplication / Division
 - Bitwise AND, OR, NOT, etc.
- Built by combining components
 - Take advantage of sharing HW when possible (e.g., subtraction using adder)

Simple 3-bit ALU: Add and bitwise OR

3-bit inputs

A and B:

A_0

A_1

A_2

B_0

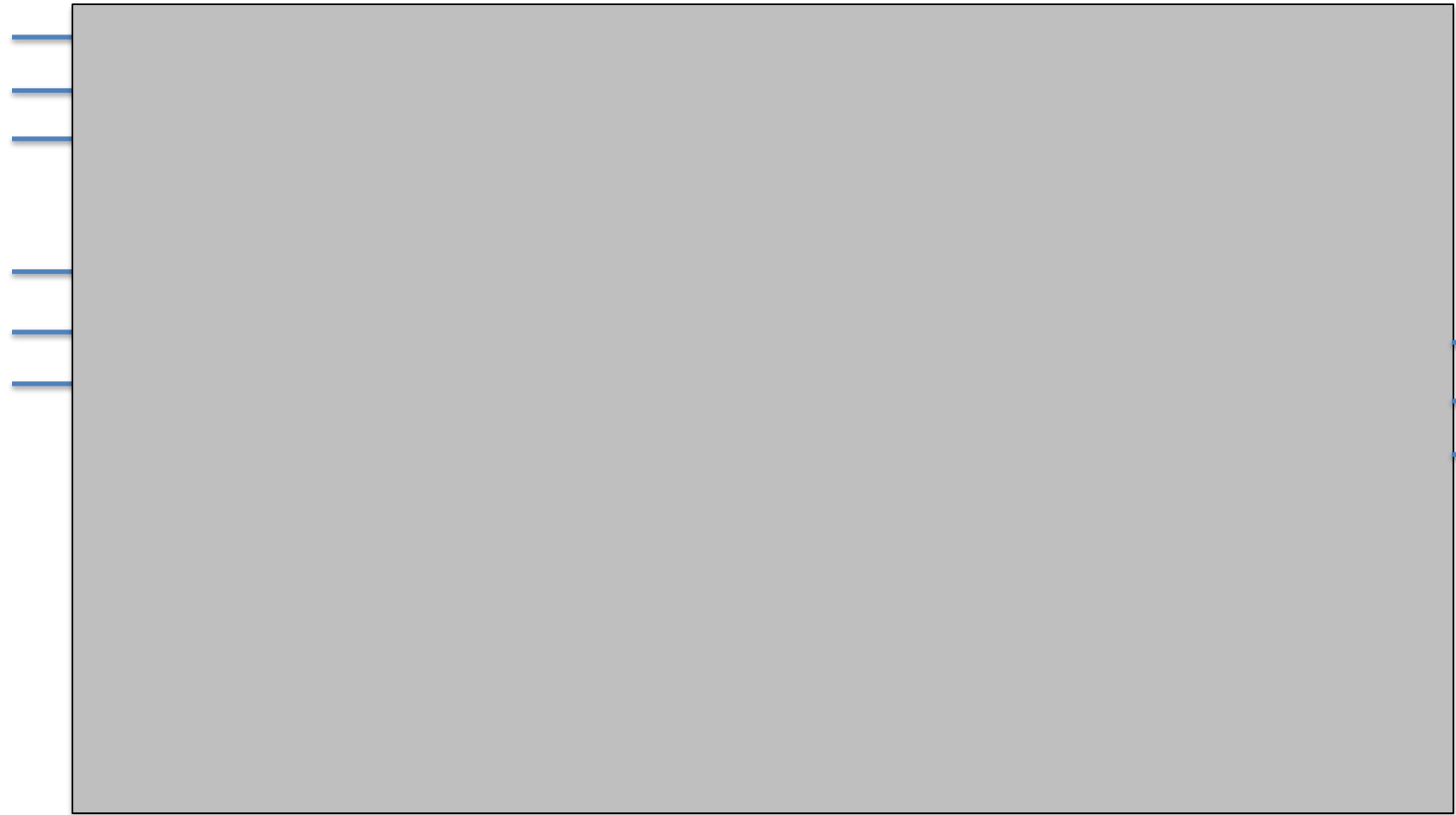
B_1

B_2

Out_0

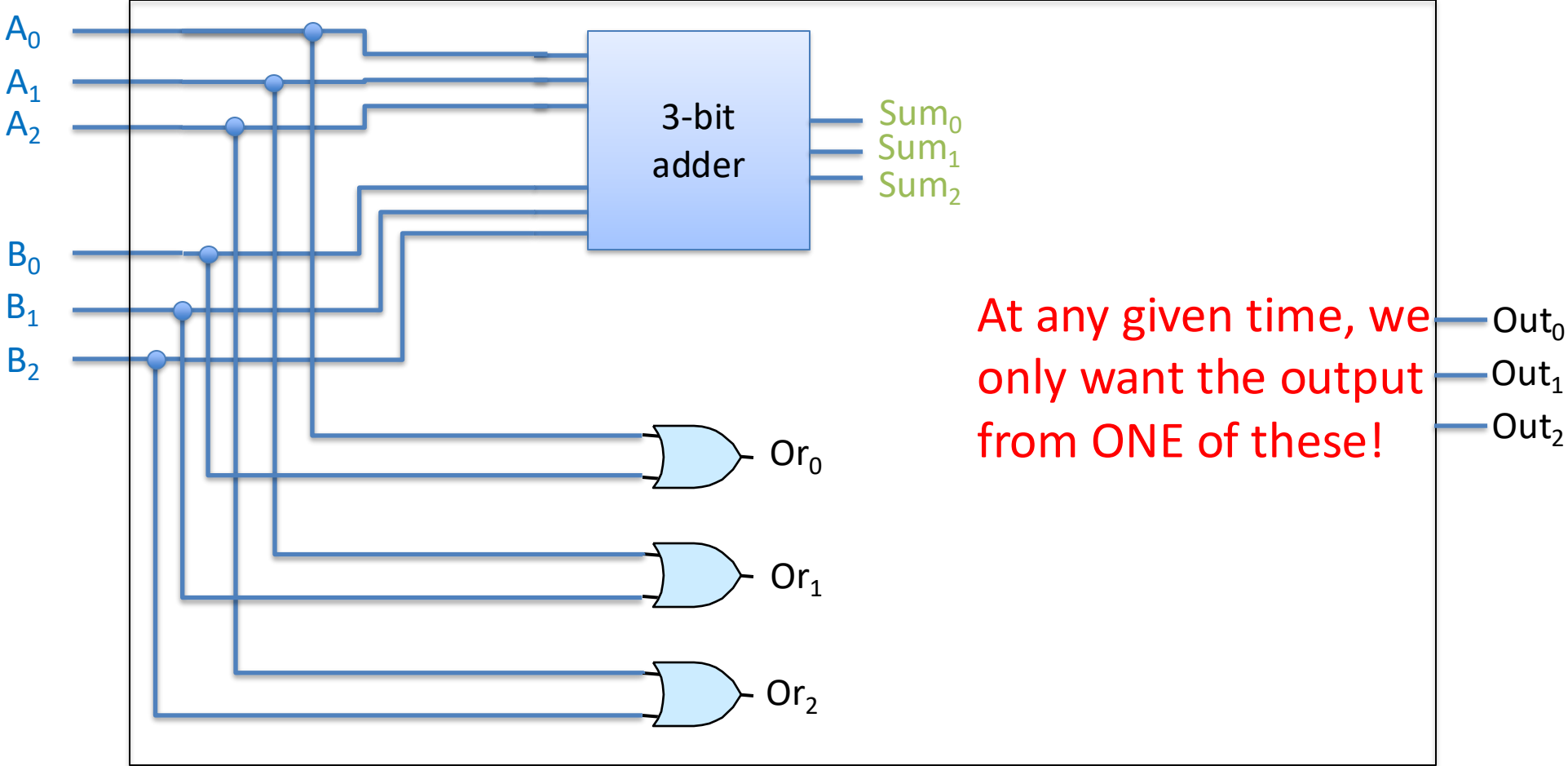
Out_1

Out_2



Simple 3-bit ALU: Add and bitwise OR

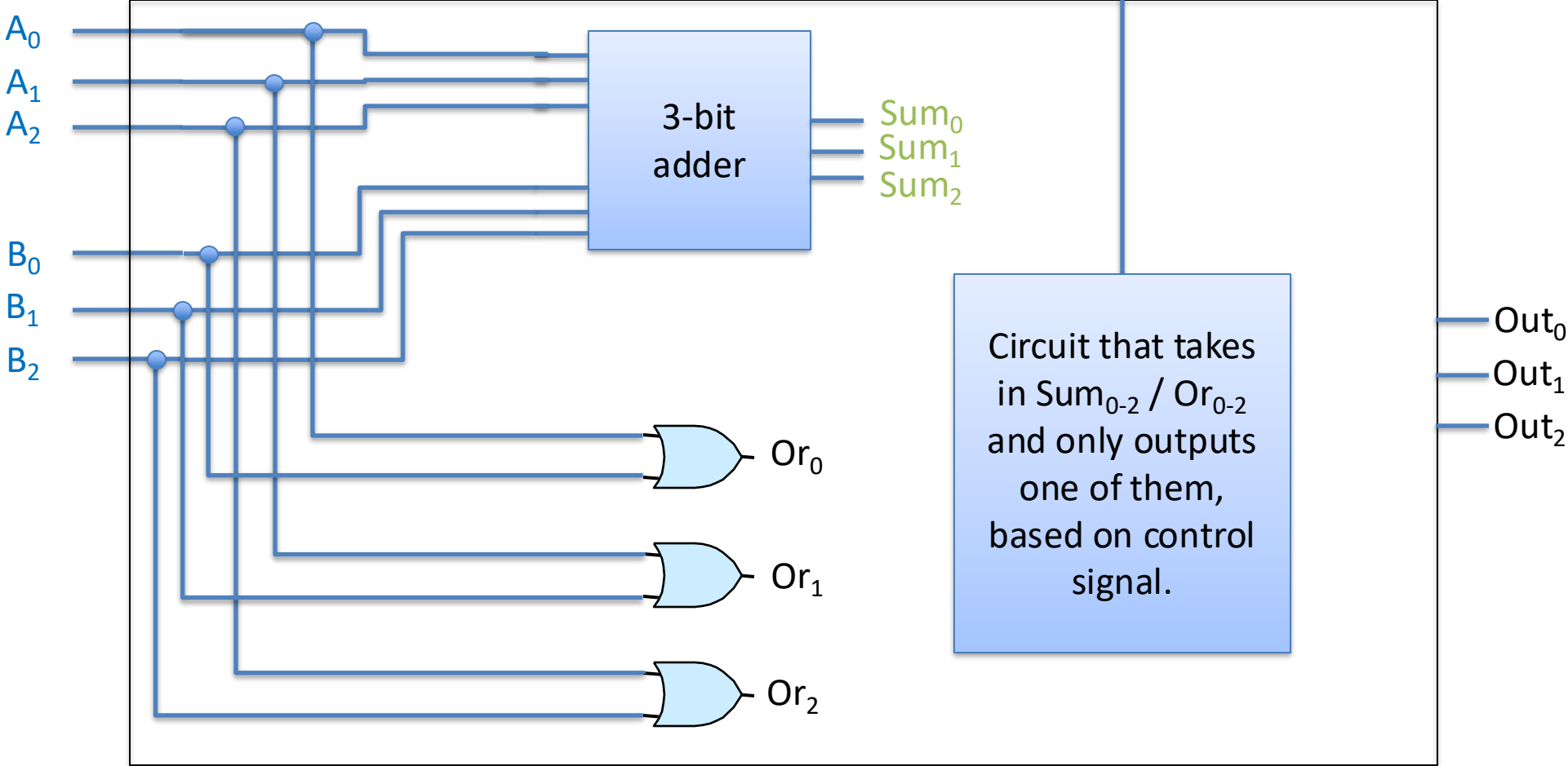
3-bit
inputs
A and B:



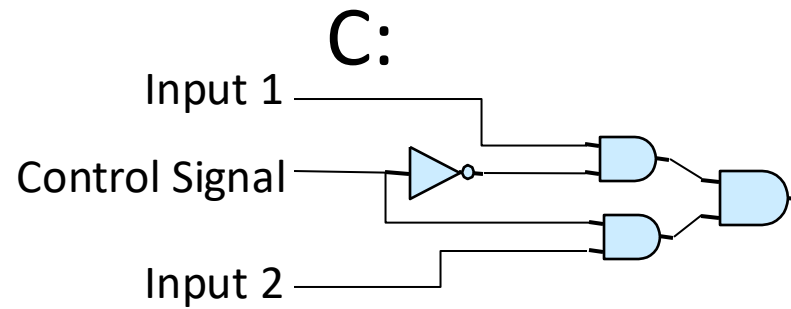
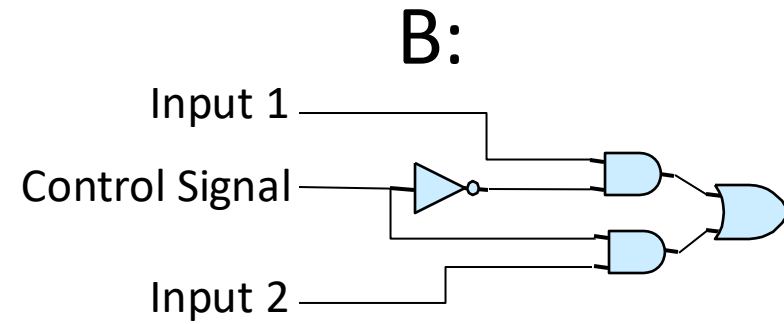
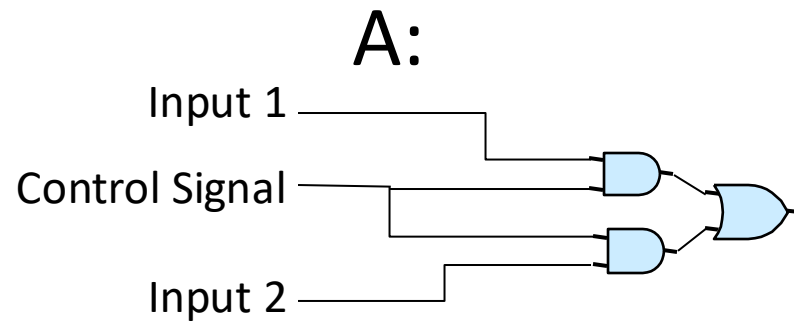
Simple 3-bit ALU: Add and bitwise OR

Extra input: control signal to select Sum vs. OR

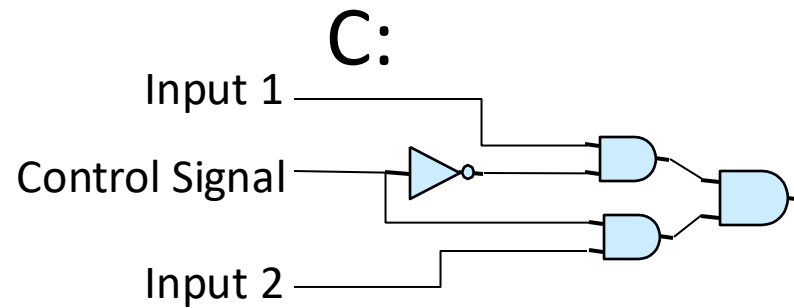
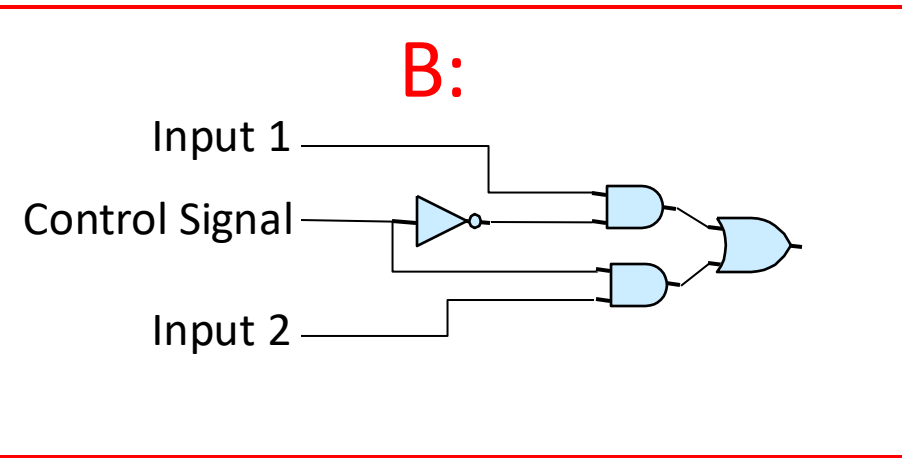
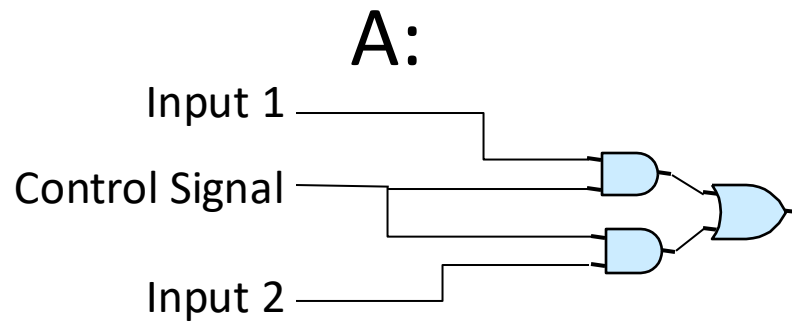
3-bit
inputs
A and B:



Which of these circuits lets us select between two inputs?



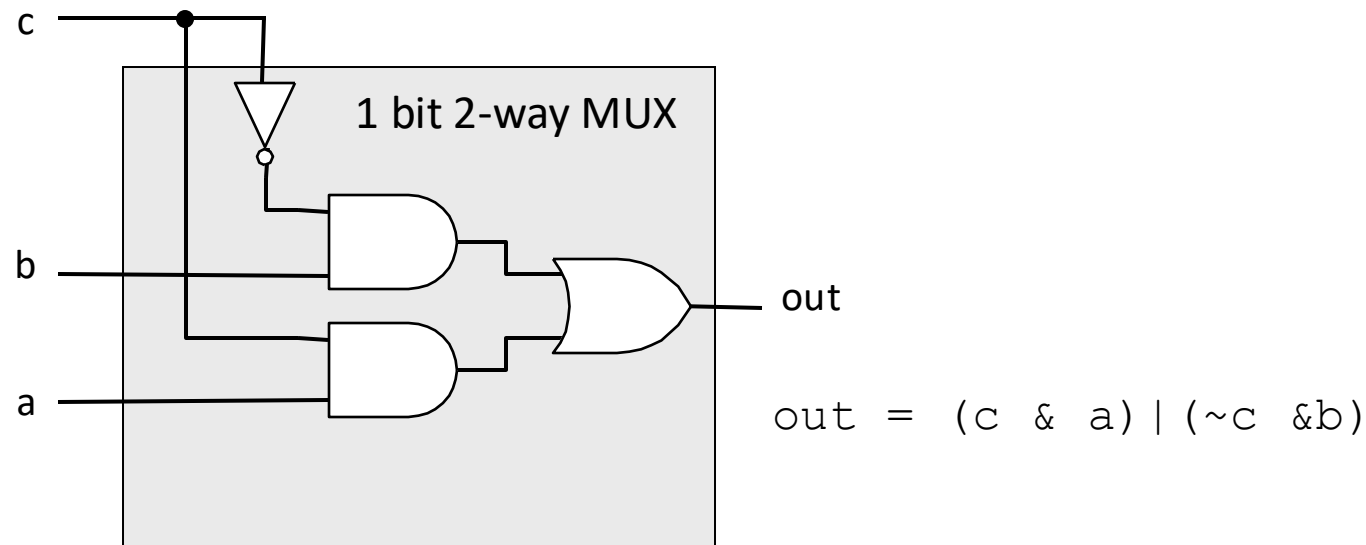
Which of these circuits lets us select between two inputs?



Multiplexor: Chooses an input value

Inputs: 2^N data inputs, N signal bits

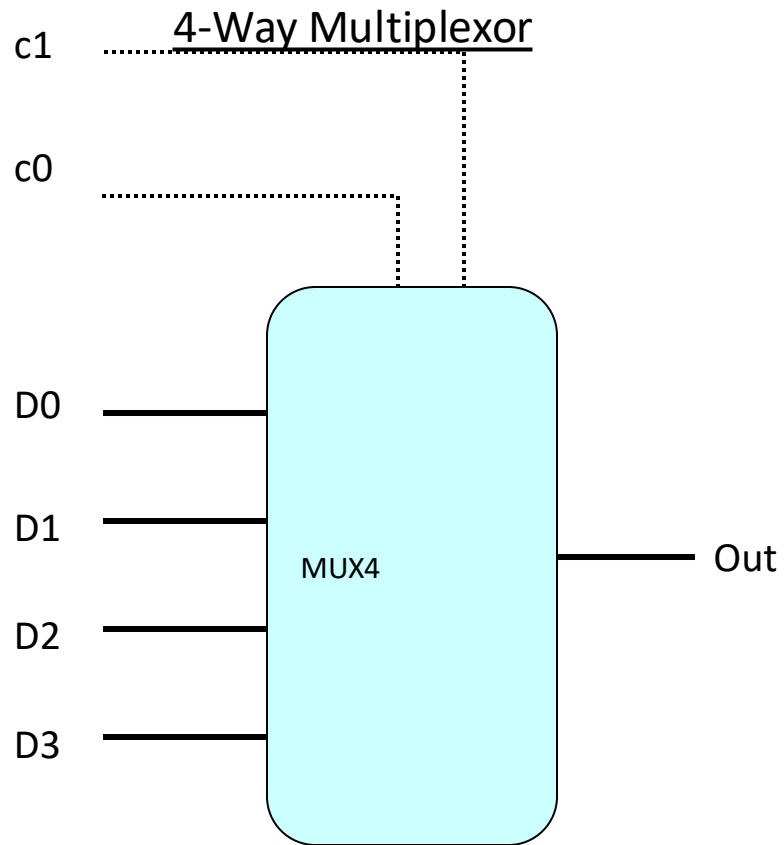
Output: is one of the 2^N input values



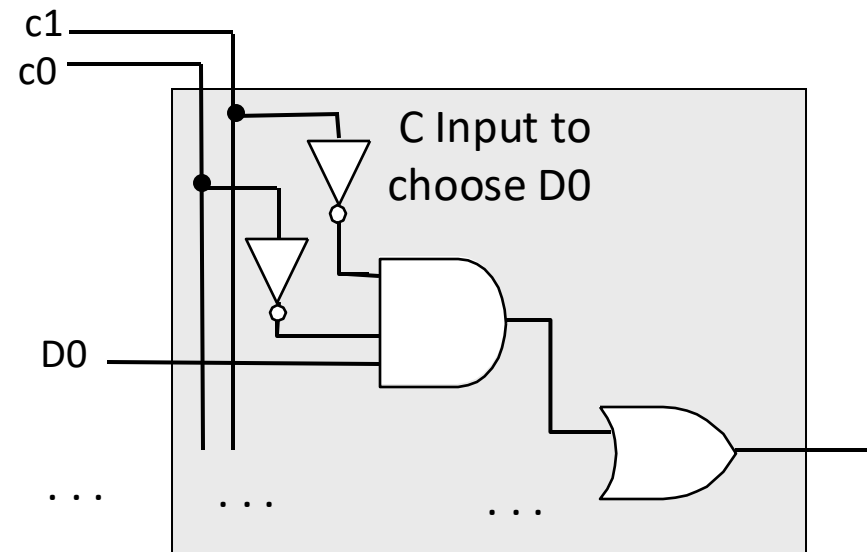
- Control signal c , chooses the input for output
 - When c is 1: choose a , when c is 0: choose b

N-Way Multiplexor

Choose one of N inputs, need $\log_2 N$ select bits

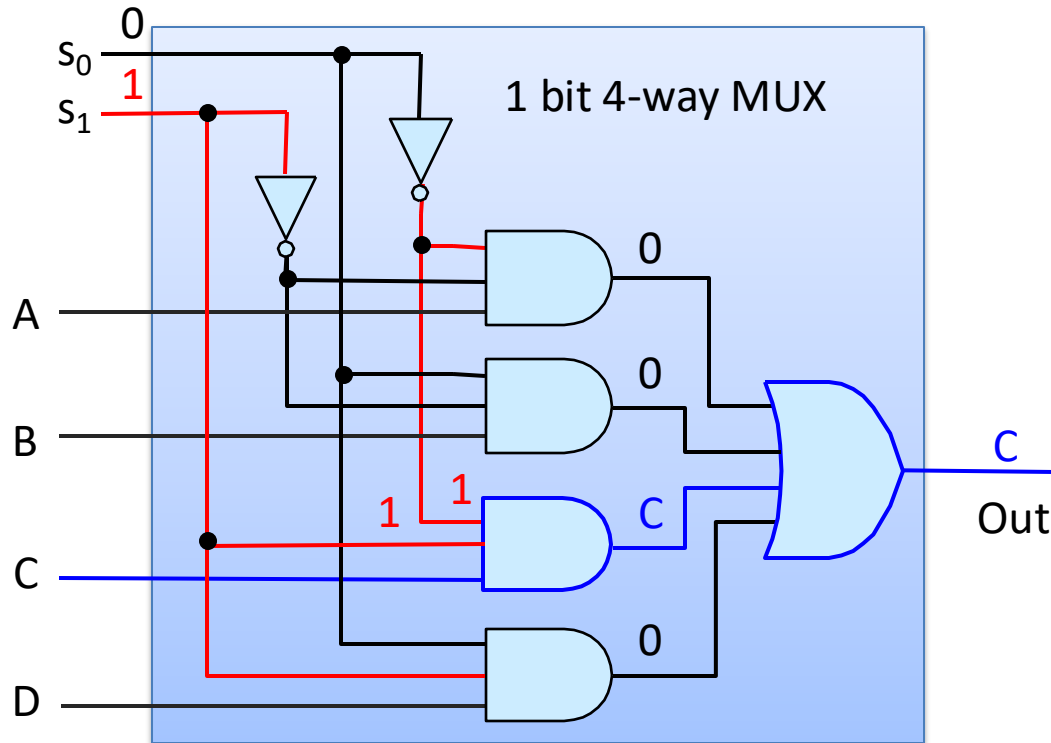


c_1	c_2	Output
0	0	D0
0	1	D1
1	0	D2
1	1	D3

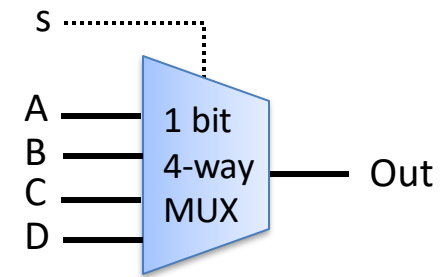


Example 1-bit, 4-way MUX

- When select input is 2 (0b10): C chosen as output



=



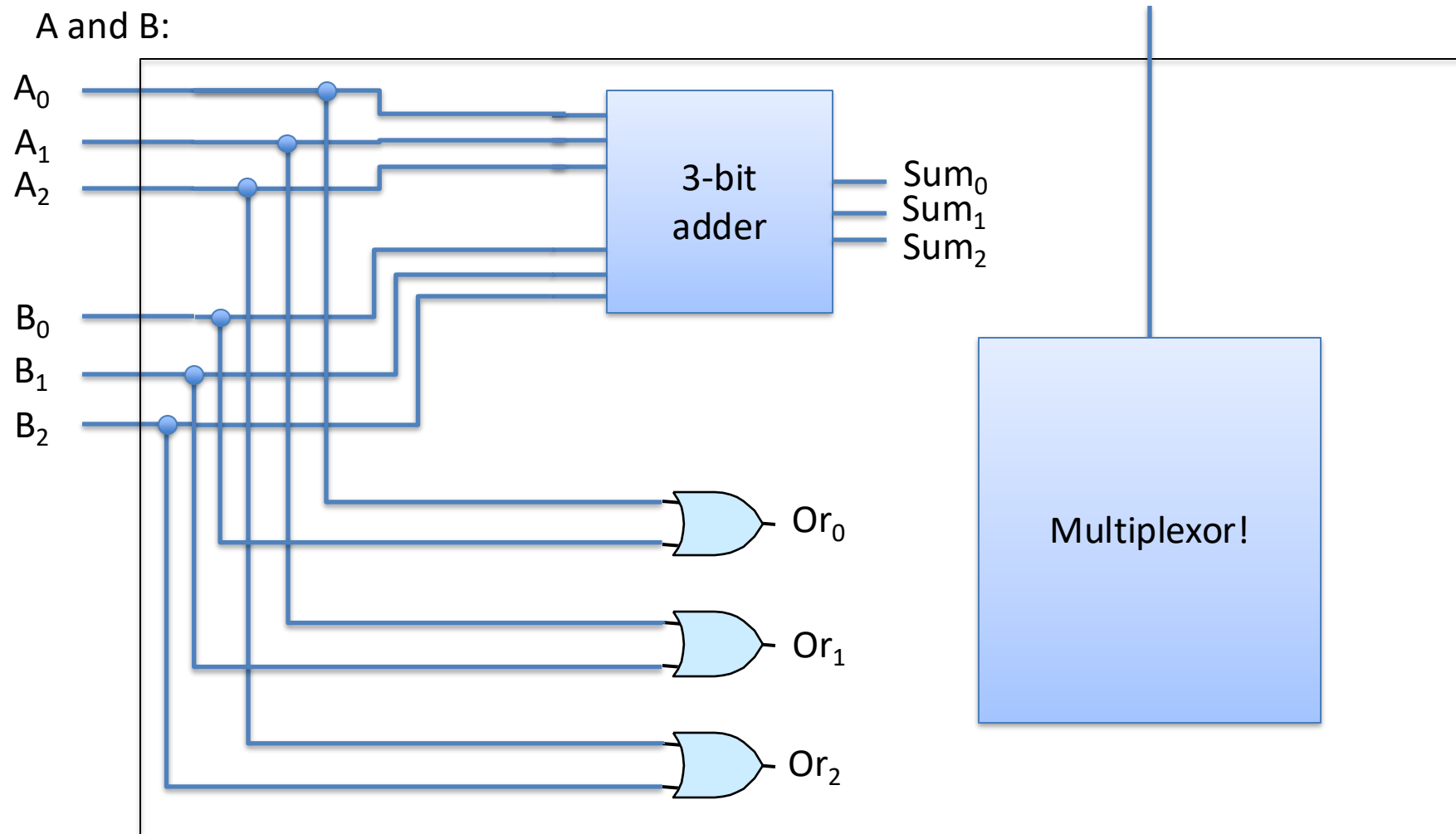
S	Out
0	A
1	B
2	C
3	D

Simple 3-bit ALU: Add and bitwise OR

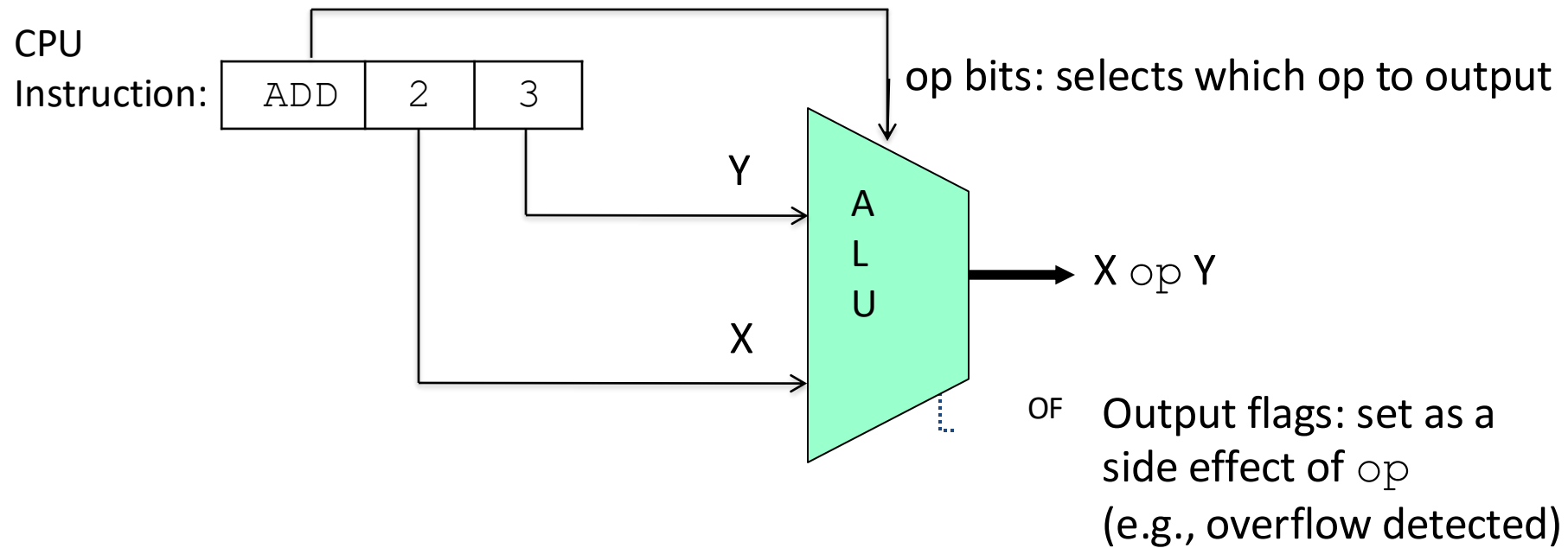
3-bit inputs

A and B:

Extra input: control signal to select Sum vs. OR



ALU: Arithmetic Logic Unit

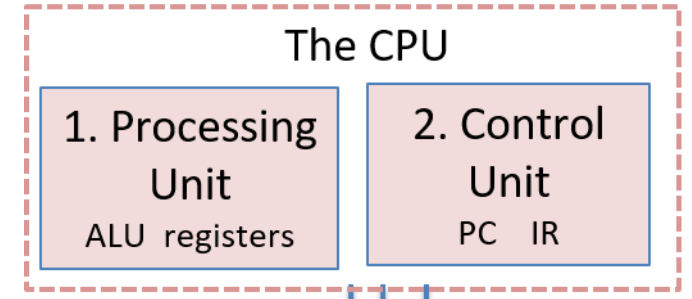


- Arithmetic and logic circuits: ADD, SUB, NOT, ...
- Control circuits: use op bits to select output
- Circuits around ALU:
 - Select input values X and Y from instruction or register
 - Select op bits from instruction to feed into ALU
 - Feed output somewhere

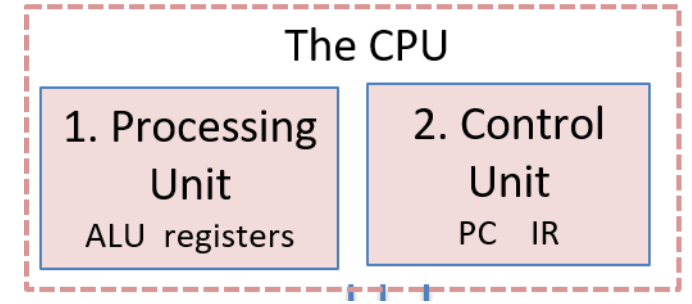
Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. ALU: implement arithmetic & logic functionality
 - Example: adder circuit to add two values together
2. Storage: to store binary values
 - Example: set of CPU registers (“register file”) to store temporary values
3. Control: support/coordinate instruction execution
 - Example: circuitry to fetch the next instruction from memory and decode it



Goal: Build a CPU (model)



Three main classifications of hardware circuits:

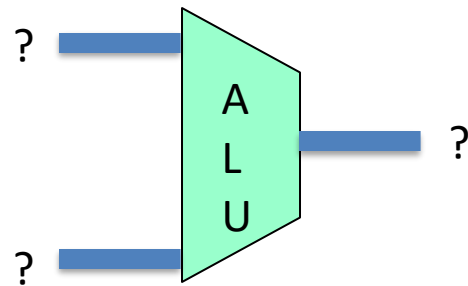
2. Storage: to store binary values

- Example: set of CPU registers (“register file”) to store temporary values

Give the CPU a “scratch space” to perform calculations and keep track of the state its in.

CPU so far...

- We can perform arithmetic!
- Storage questions:
 - Where do the ALU input values come from?
 - Where do we store the result?
 - What does this “register” thing mean?



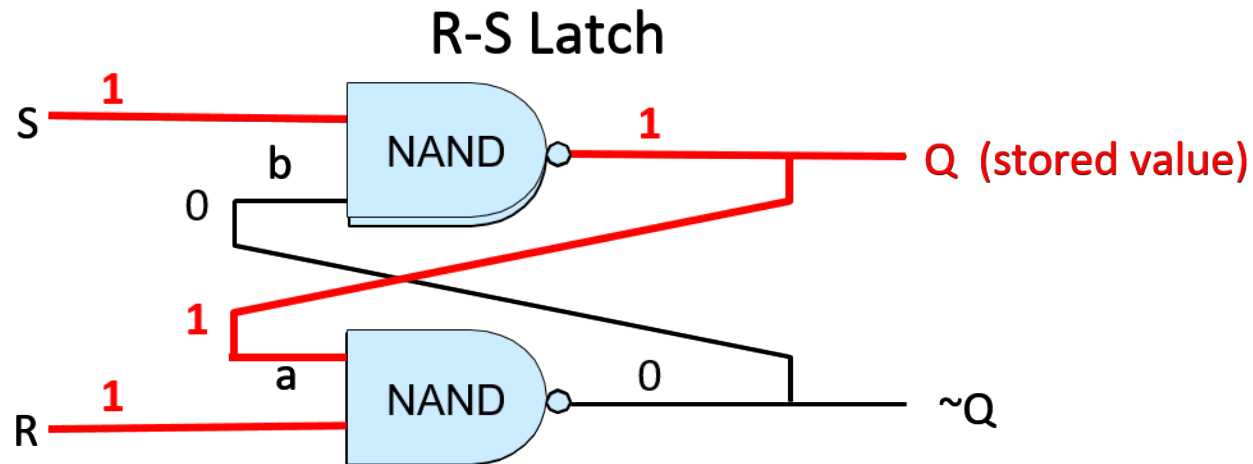
Memory Circuit Goals: Starting Small

- Store a 0 or 1
- Retrieve the 0 or 1 value on demand (read)
- Set the 0 or 1 value on demand (write)

R-S Latch: Stores Value Q

When R and S are both 1: Maintain a value

R and S are never both simultaneously 0

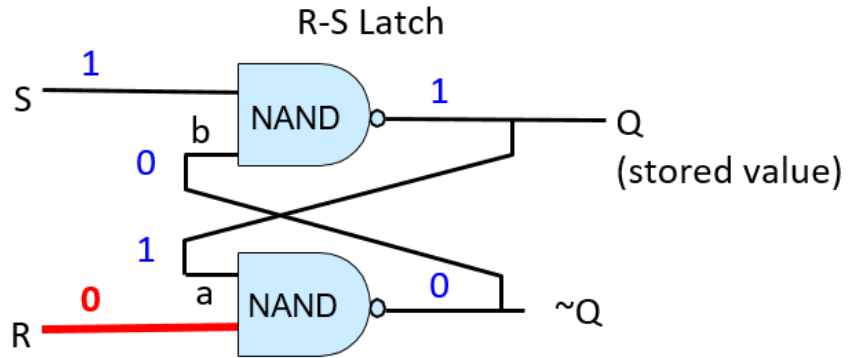


- To write a new value:
 - Set S to 0 momentarily (R stays at 1): to write a 1
 - Set R to 0 momentarily (S stays at 1): to write a 0

R-S Latch: Stores Value Q

Assume that the RS Latch currently stores 1.

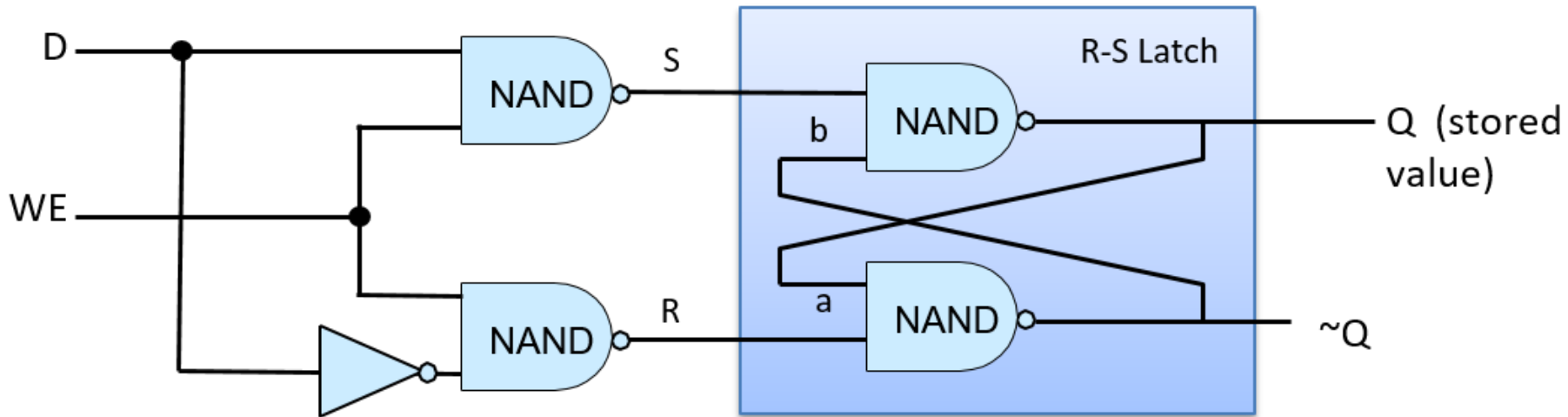
To write 0 into the latch, set R's value to 0.



A. Set R to 0 to store 0

Gated D Latch

Controls S-R latch writing, ensures S & R never both 0



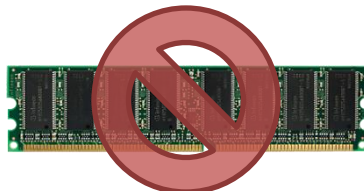
D: into top NAND, $\sim D$ into bottom NAND

WE: write-enabled, when set, latch is set to value of D

Latches used in registers (up next) and SRAM (caches, later)

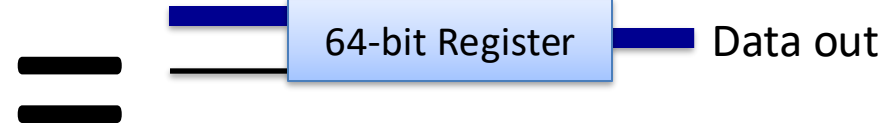
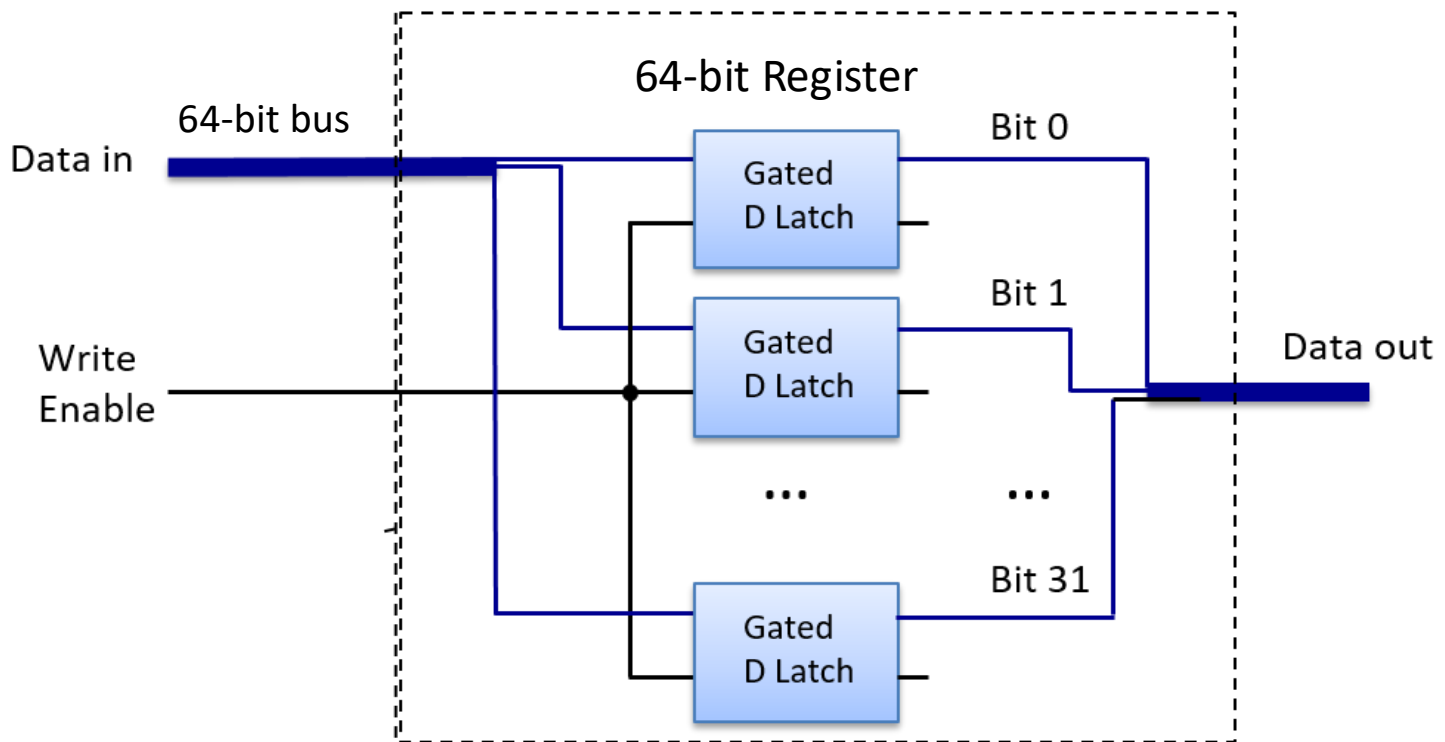
Fast, not very dense, expensive

DRAM: capacitor-based



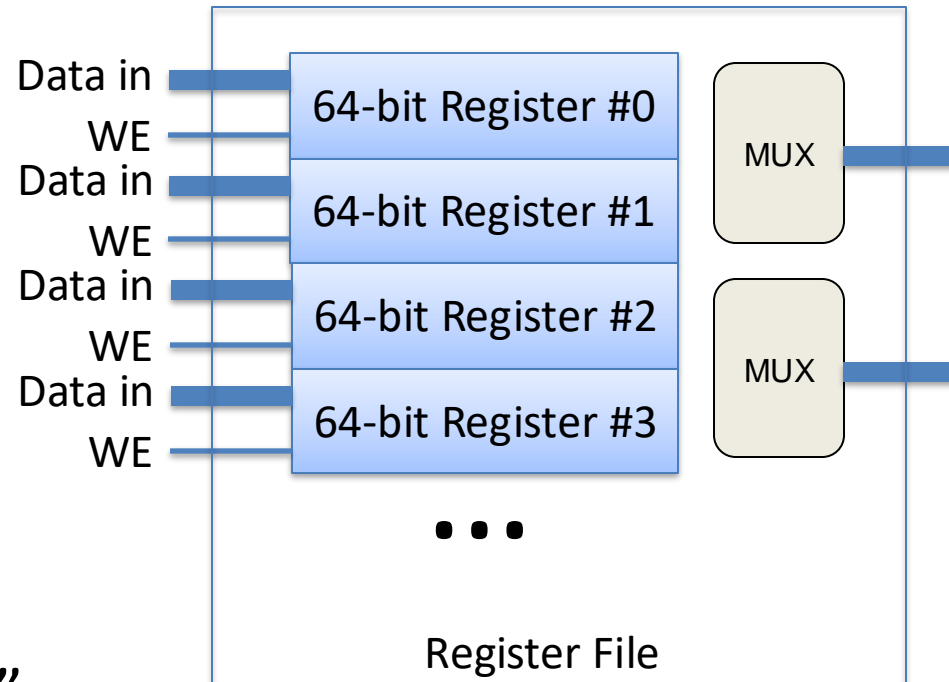
An N-bit Register

- Fixed-size storage (8-bit, 32-bit, 64-bit, etc.)
- Gated D latch lets us store one bit
 - Connect N of them to the same write-enable wire!



“Register file”

- A set of registers for the CPU to store temporary values.
- This is (finally) something you will interact with!
- Instructions of form:
 - “add R1 + R2, store result in R3”



Memory Circuit Summary

- Lots of abstraction going on here!
 - Gates hide the details of transistors.
 - Build R-S Latches out of gates to store one bit.
 - Combining multiple latches gives us N-bit register.
 - Grouping N-bit registers gives us register file.
- Register file's simple interface:
 - Read R_x 's value, use for calculation
 - Write R_y 's value to store result

CPU so far...

We know how to store data (in register file).

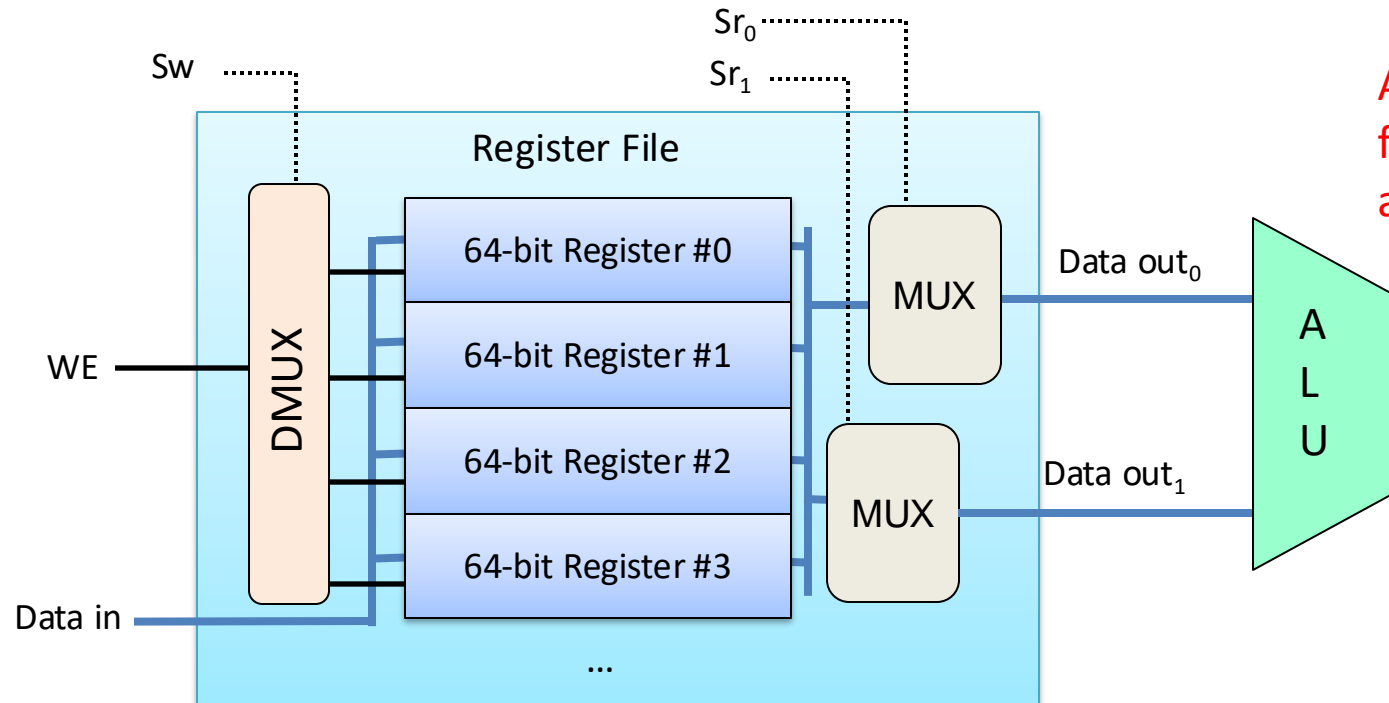
We know how to perform arithmetic on it, by feeding it to ALU.

Remaining questions:

Which register(s) do we use as input to ALU?

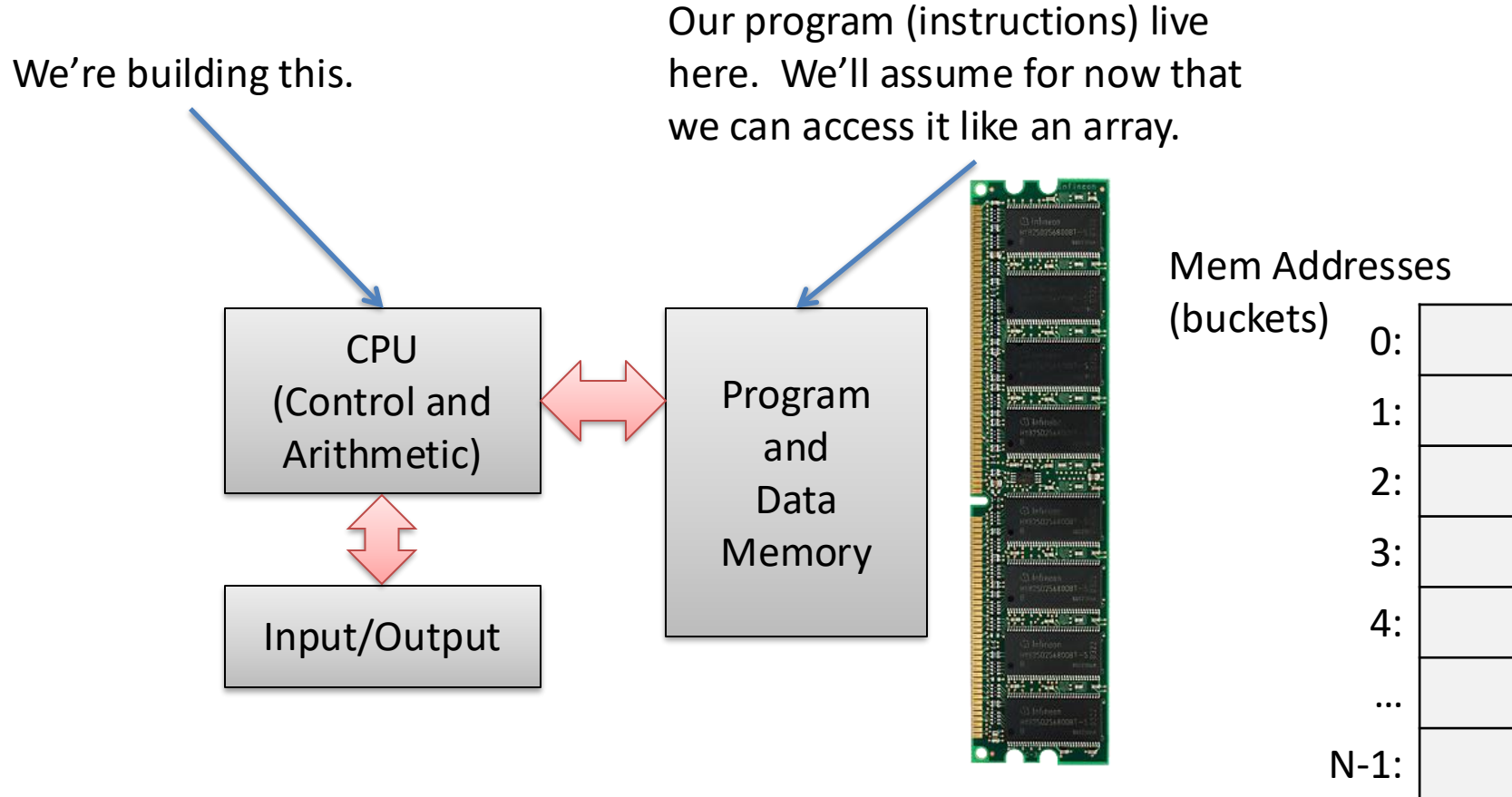
Which operation should the ALU perform?

To which register should we store the result?



All this info comes from the program: a series of instructions.

Recall: Von Neumann Model



Digital Circuits - Building a CPU

Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality
(ex) adder to add two values together
2. Storage: to store binary values
(ex) Register File: set of CPU registers
3. Control: support/coordinate instruction execution
(ex) fetch the next instruction to execute

Circuits are built from Logic Gates which are built from transistors

HW Circuits
Logic Gates
Transistor

Digital Circuits - Building a CPU

Three main classifications of HW circuits:

3. Control: support/coordinate instruction execution
(ex) fetch the next instruction to execute

Keep track of where we are in the program.

Execute instruction, move to next.

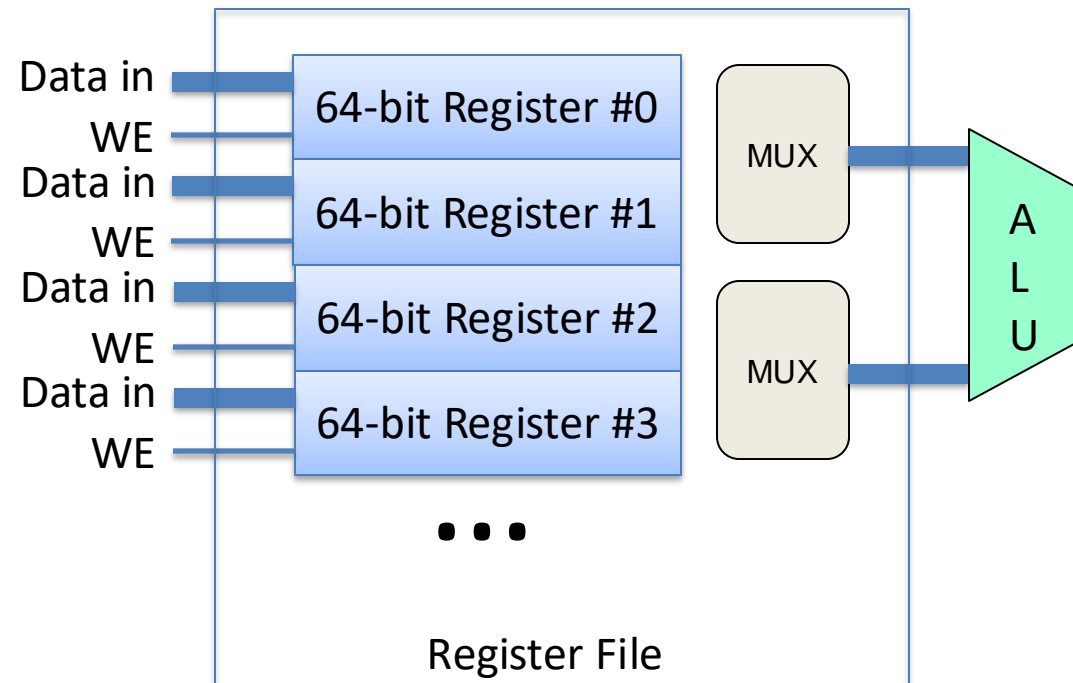
HW Circuits
Logic Gates
Transistor

Control Unit

Which register(s) do we use as input to ALU?

Which operation should the ALU perform?

To which register should we store the result?



All this info
comes from our
program:
a series of
instructions.

CPU Game Plan

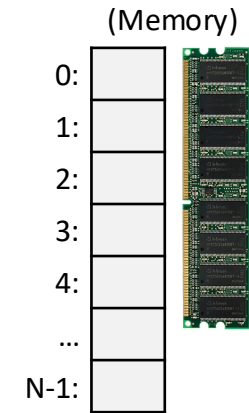
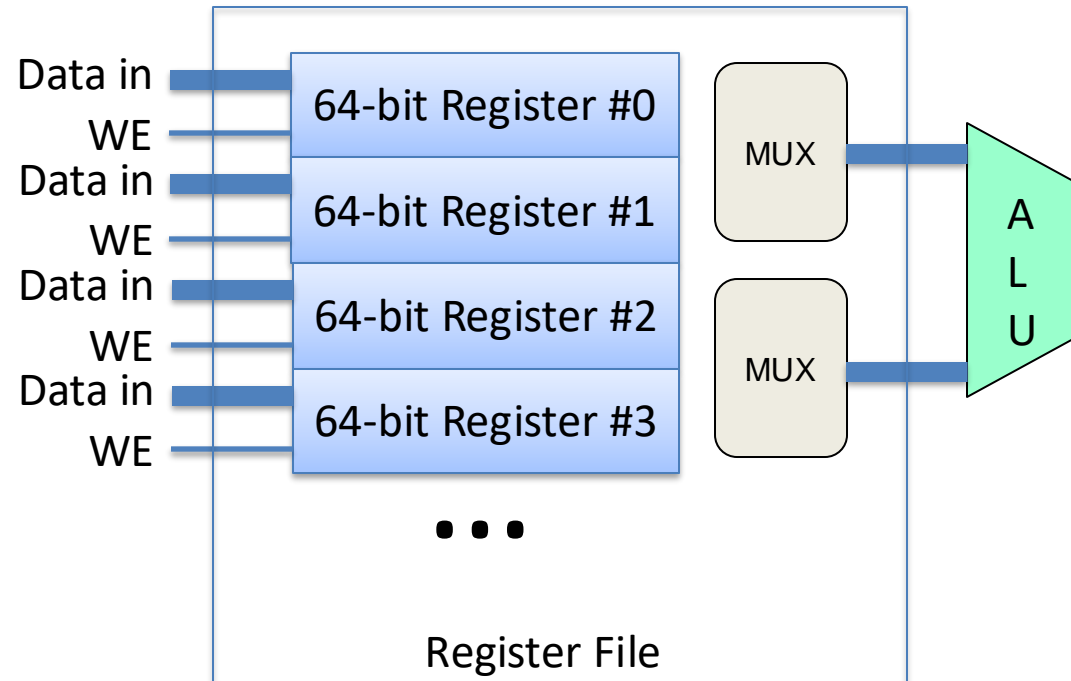
- Fetch instruction from memory
- Decode what the instruction is telling us to do
 - Tell the ALU what it should be doing
 - Find the correct operands
- Execute the instruction (arithmetic, etc.)
- Store the result

Program State

Let's add two more special registers (not in register file) to keep track of program.

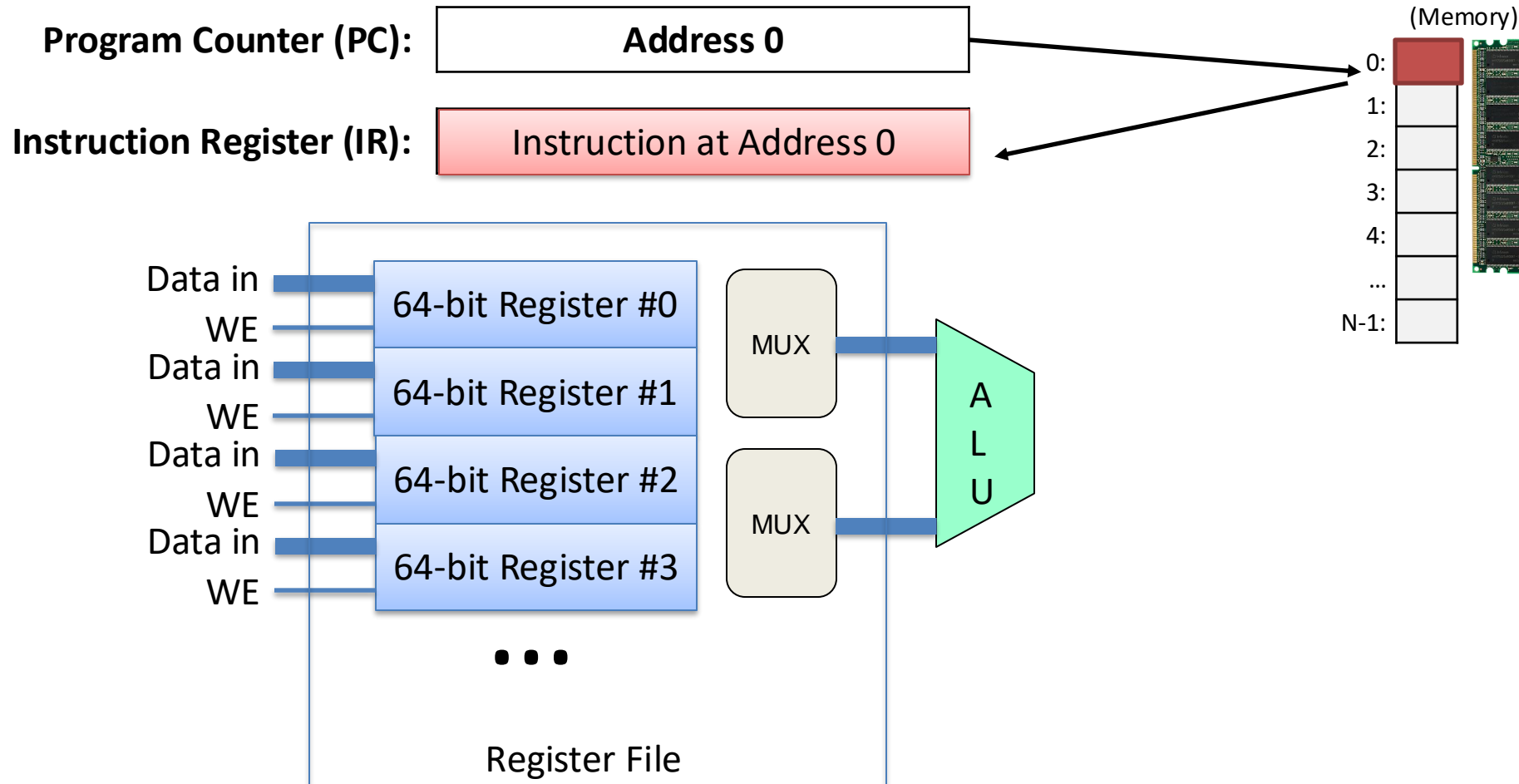
Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)



Fetching instructions.

Load IR with the contents of memory at the address stored in the PC.

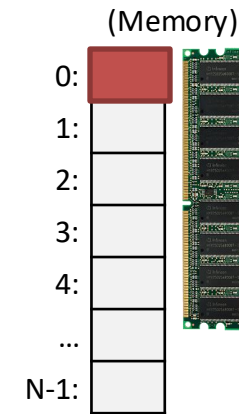
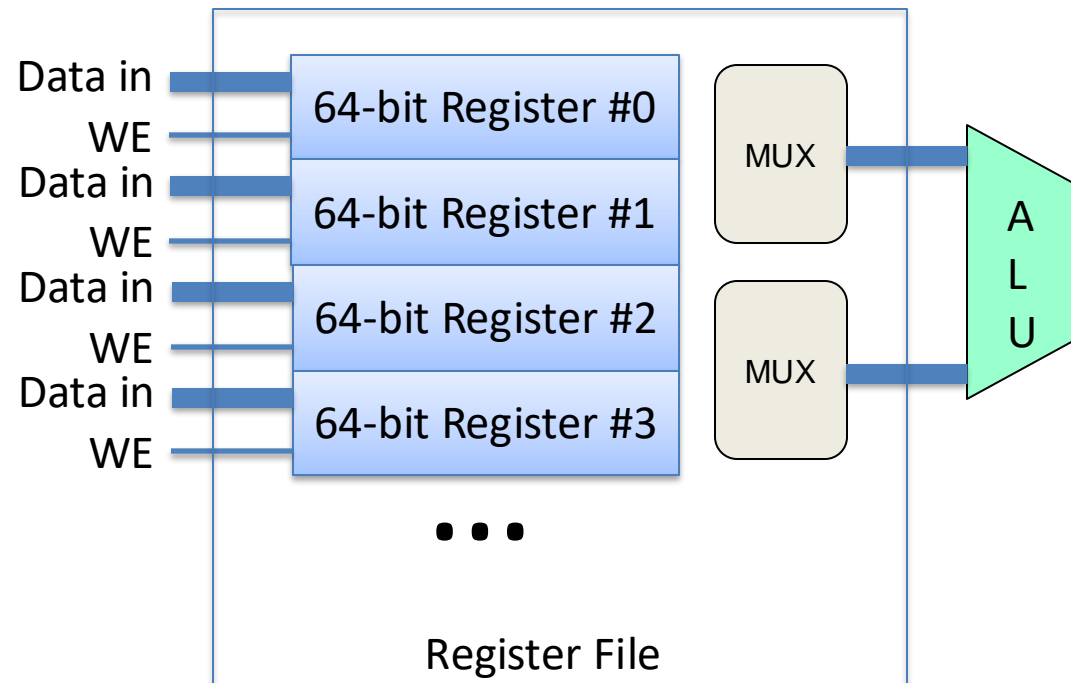


Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

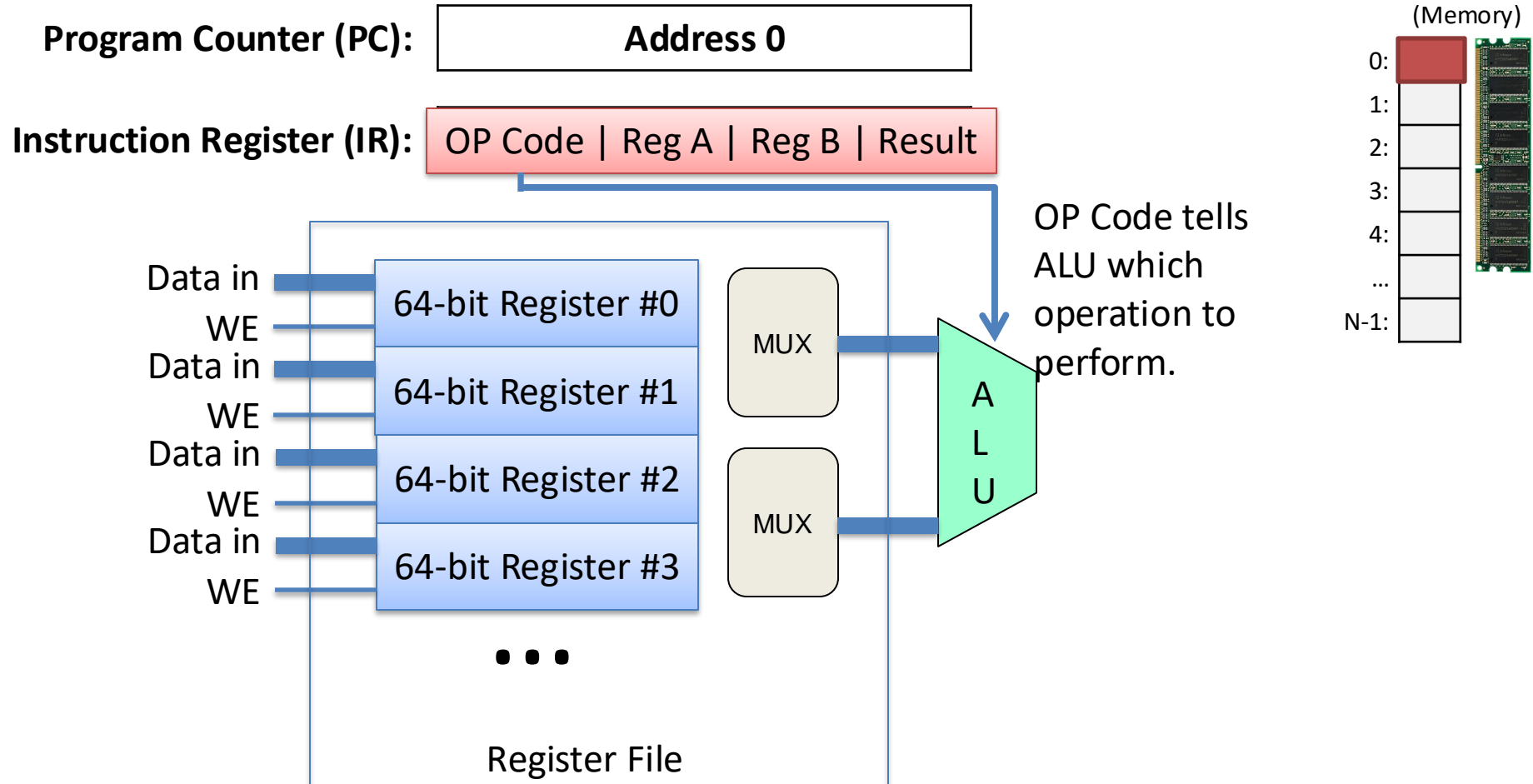
Program Counter (PC): Address 0

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



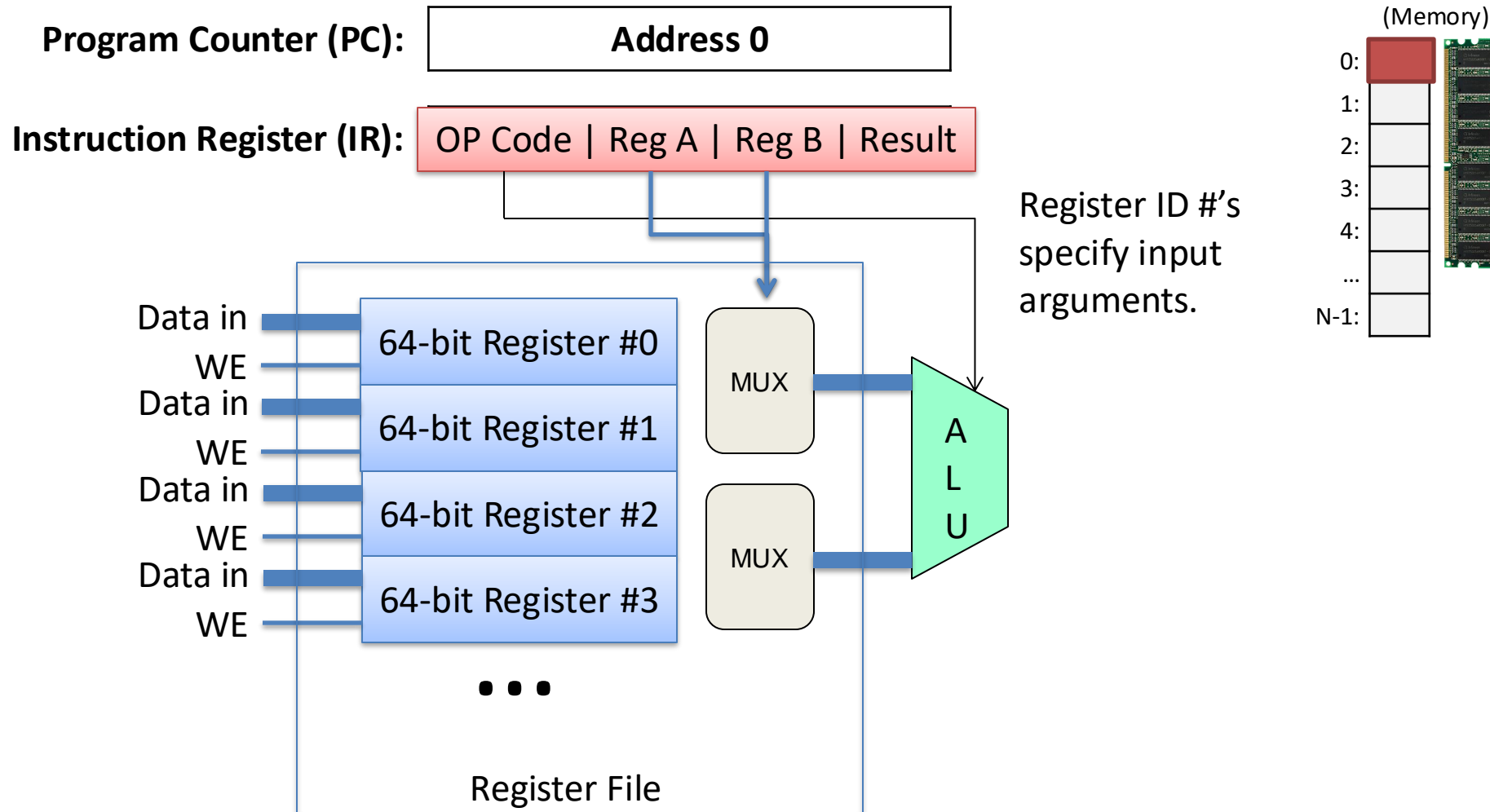
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



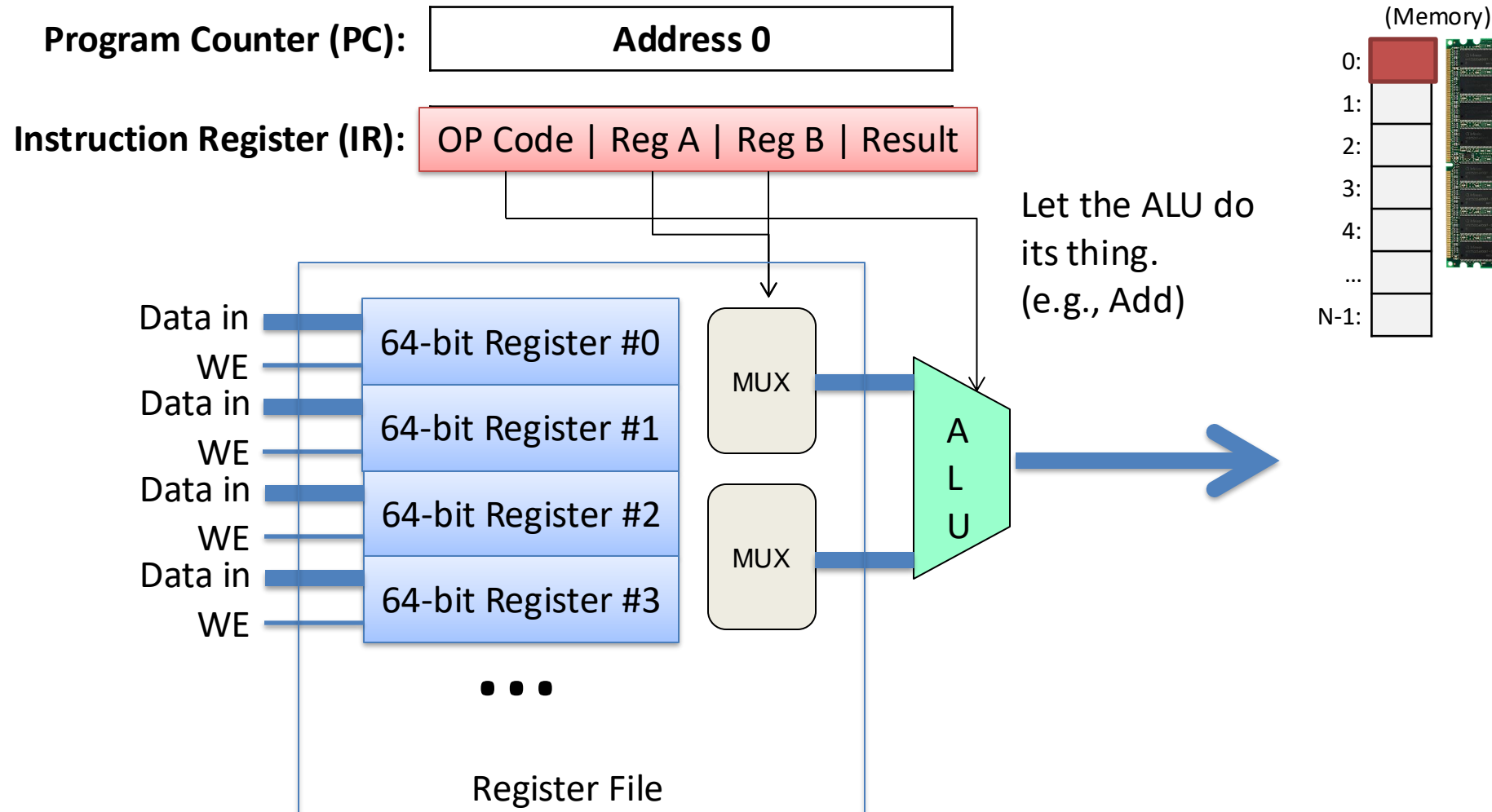
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



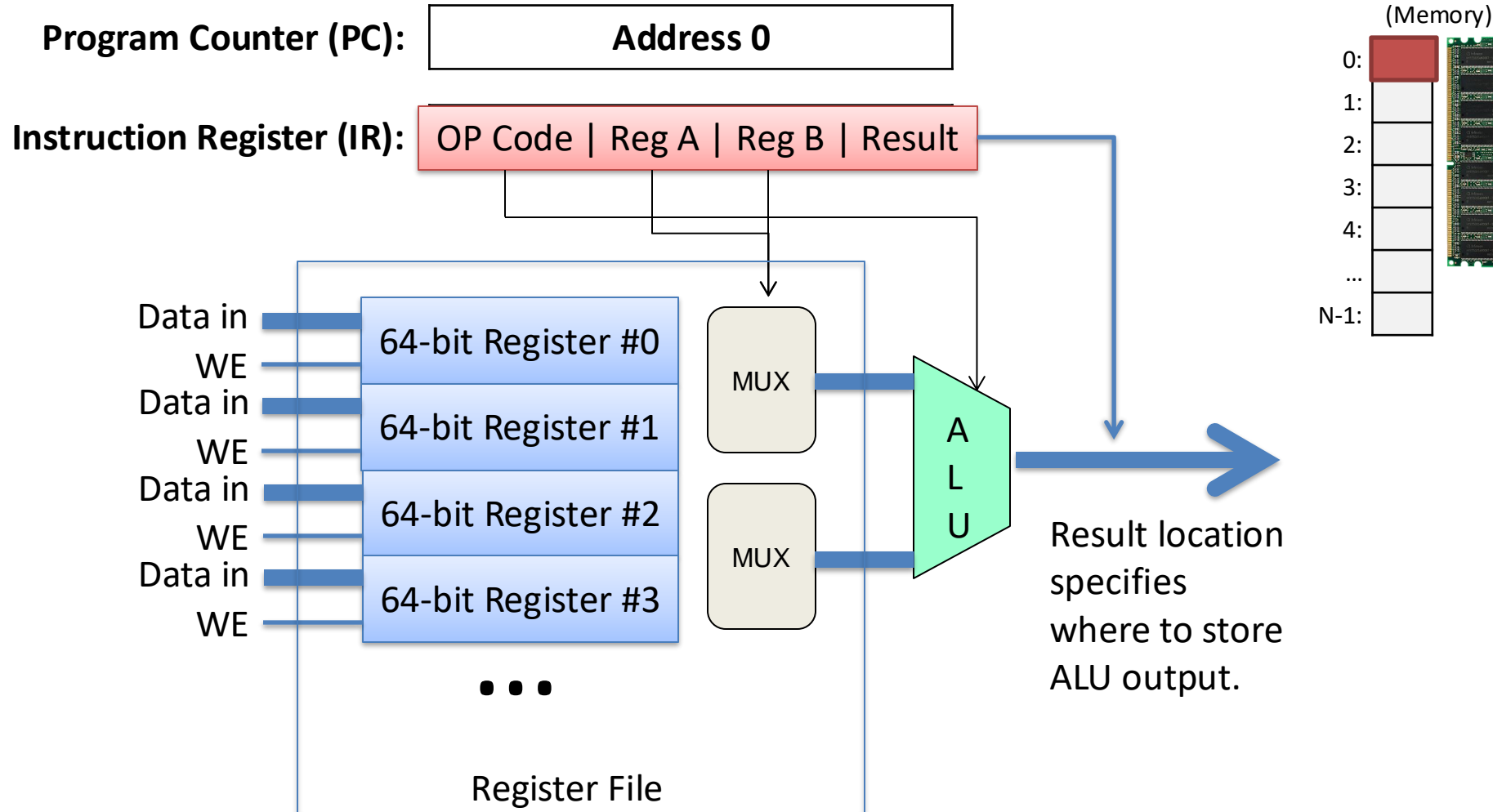
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



Storing results.

We've just computed something. Where do we put it?



Why do we need a program counter? Can't we just start at 0 and count up one at a time from there?

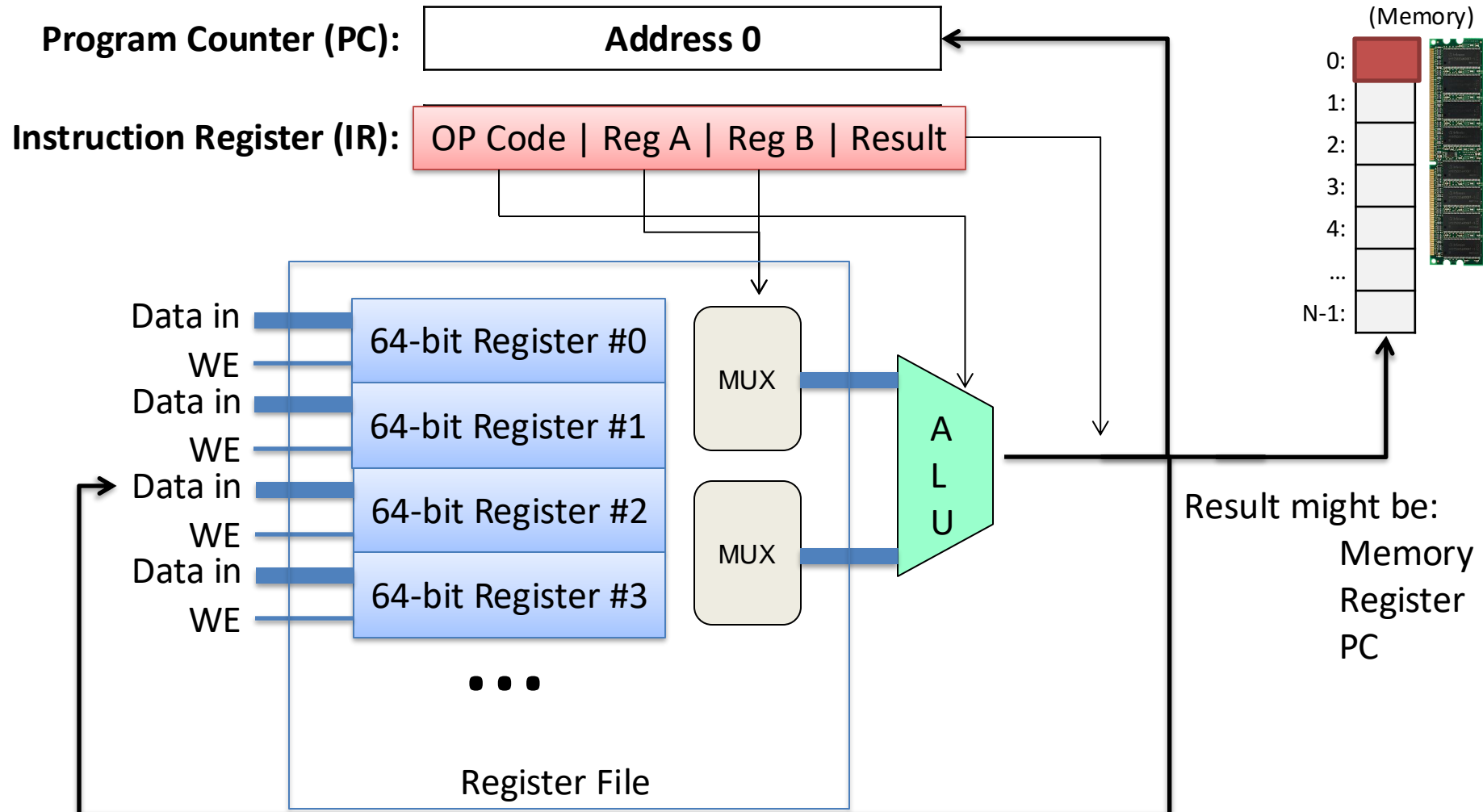
- A. We don't, it's there for convenience.
- B. Some instructions might skip the PC forward by more than one.
- C. Some instructions might adjust the PC backwards.
- D. We need the PC for some other reason(s).

Why do we need a program counter? Can't we just start at 0 and count up one at a time from there?

- A. We don't, it's there for convenience.
- B. Some instructions might skip the PC forward by more than one.
- C. Some instructions might adjust the PC backwards.
- D. We need the PC for some other reason(s).

Storing results.

Interpret the instruction bits: What operation? Which arguments?

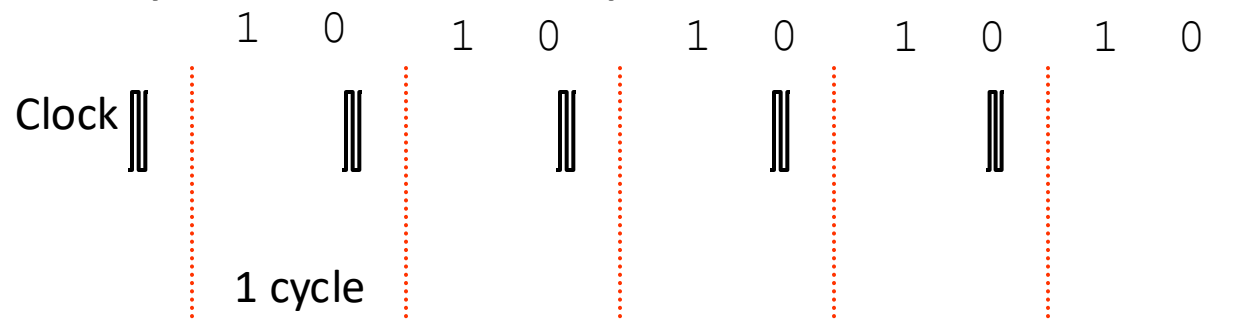


Clocking

- Need to periodically transition from one instruction to the next.
- It takes time to fetch from memory, for signal to propagate through wires, etc.
 - Too fast: don't fully compute result
 - Too slow: waste time

Clock Driven System

- Everything in is driven by a discrete clock
 - clock: an oscillator circuit, generates hi low pulse
 - clock cycle: one hi-low pair



- Clock determines how fast system runs
 - Processor can only do one thing per clock cycle
 - Usually just one part of executing an instruction
 - 1GHz processor:
 - 1 billion cycles/second → 1 cycle every nanosecond

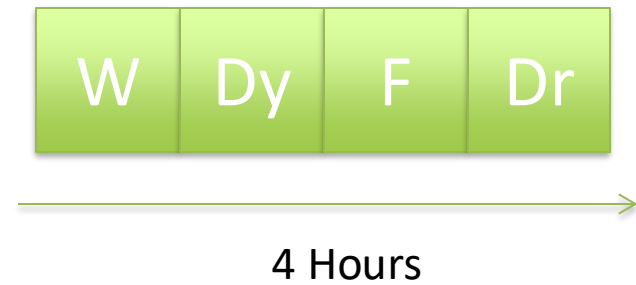
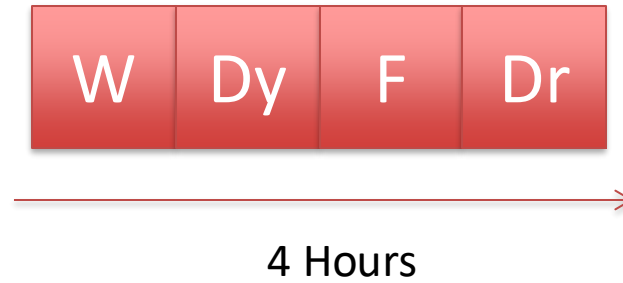
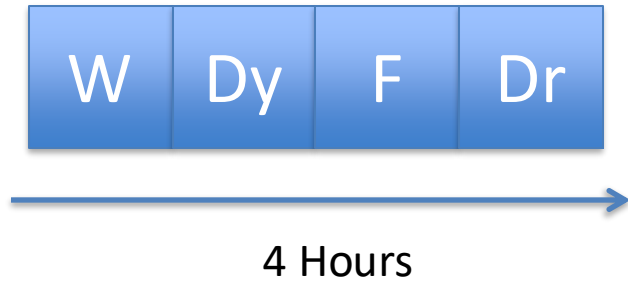
Cycle Time: Laundry Analogy

- Discrete stages: fetch, decode, execute, store
- Analogy (laundry): washer, dryer, folding, dresser



You have big problems if you have millions of loads of laundry to do....

Laundry



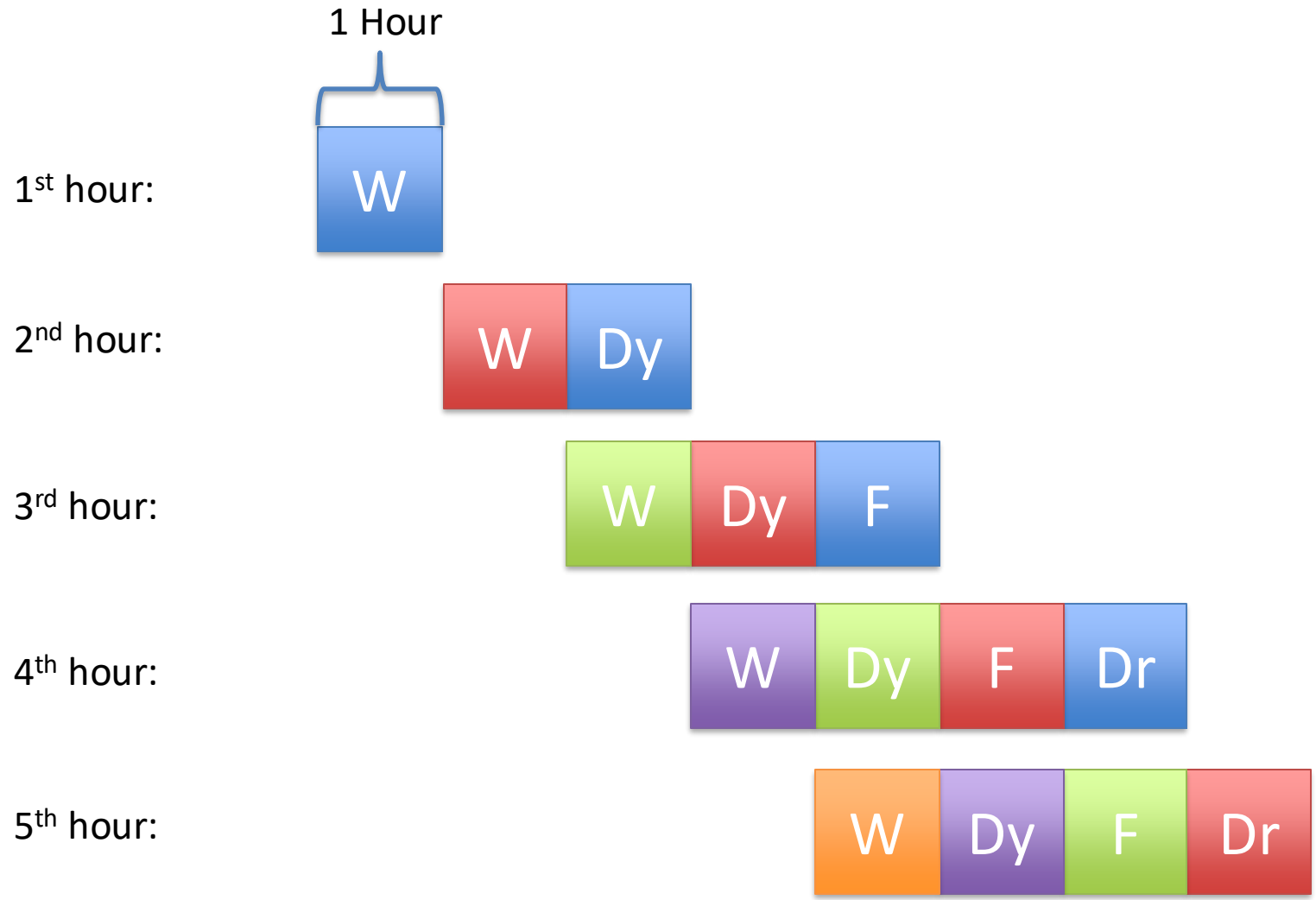
4-hour cycle time.

Finishes a laundry load every cycle.

(6 laundry loads per day)

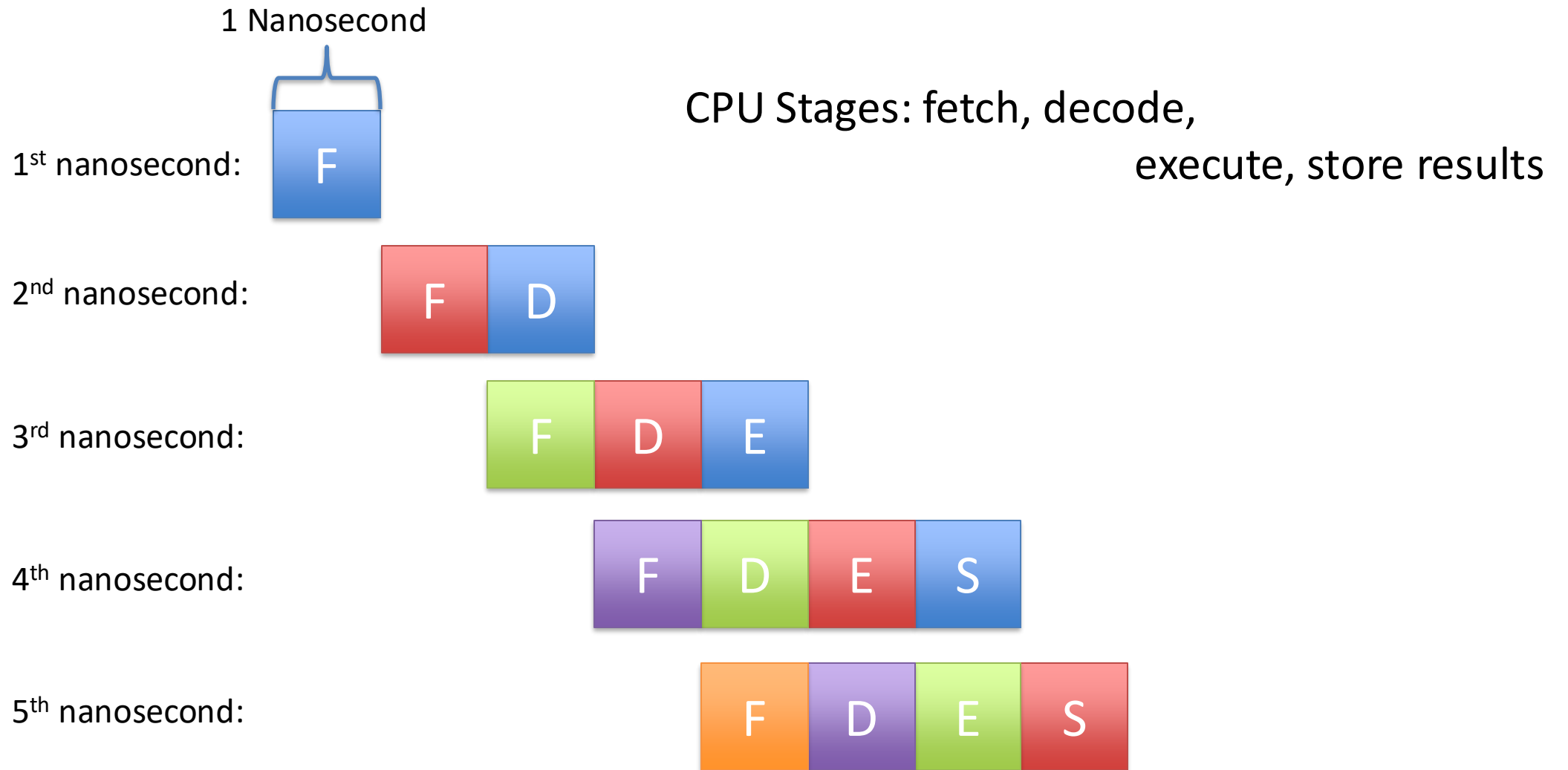


Pipelining (Laundry)



Steady state: One load finishes every hour!
(Not every four hours like before.)

Pipelining (CPU)



Steady state: One instruction finishes every nanosecond!
(Clock rate can be faster.)

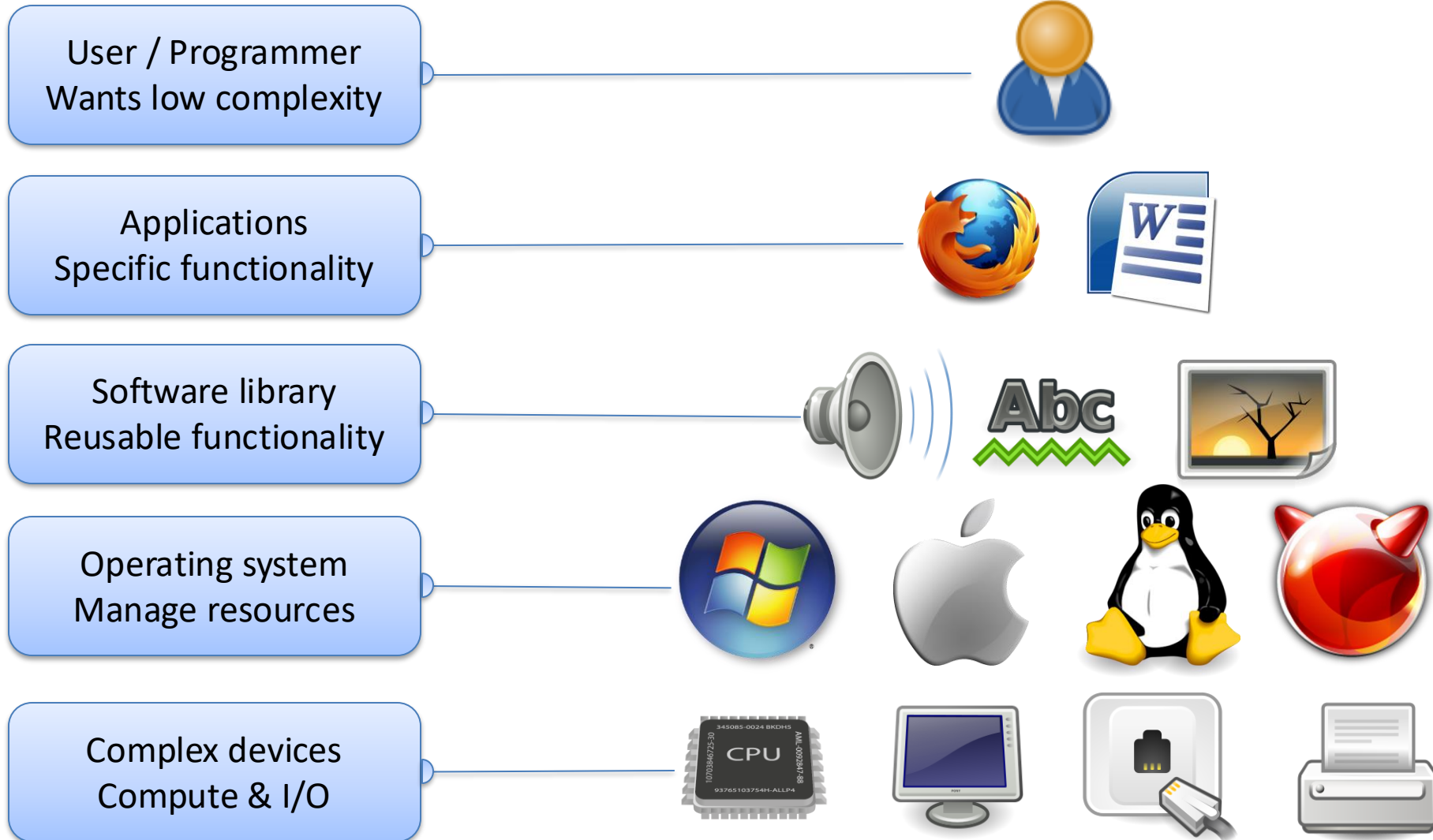
Pipelining

(For more details about this and the other things we talked about here, take architecture.)

Today

- 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



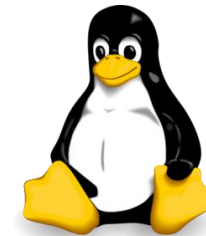
Abstraction

Applications
Specific functionality



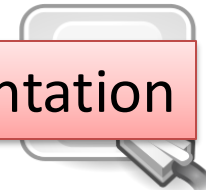
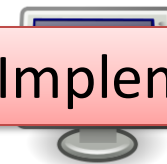
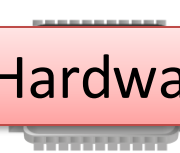
This week: Machine Interface

Operating system
Manage resources



Complex d
Compute & I/O

Last week: Circuits, Hardware Implementation

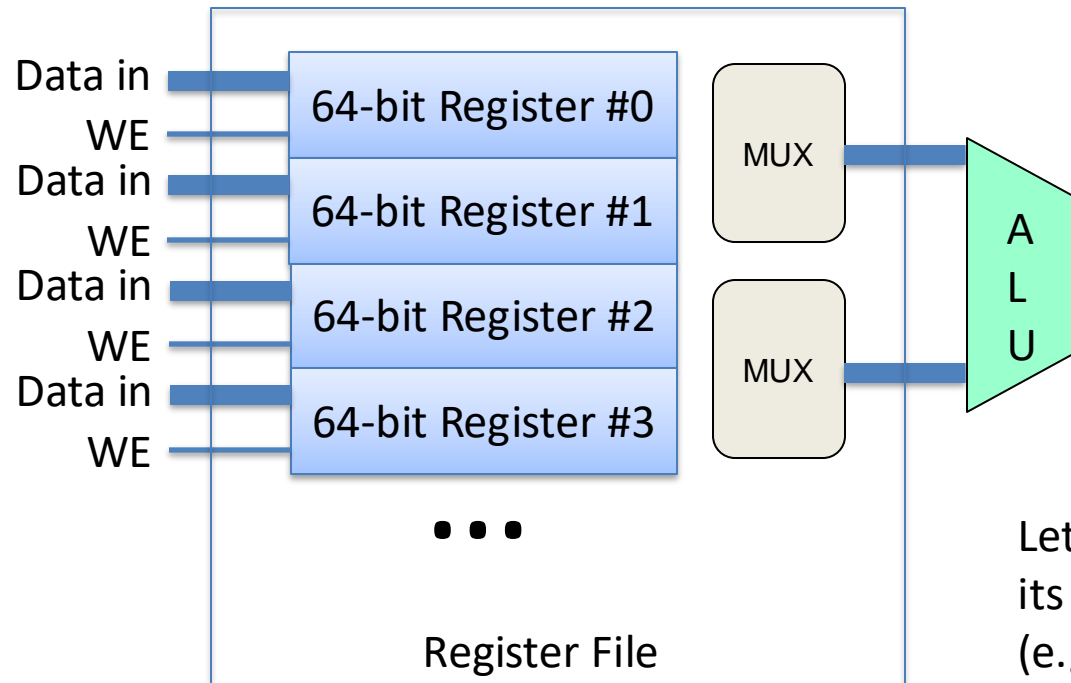


Hardware: Control, Storage, ALU circuitry

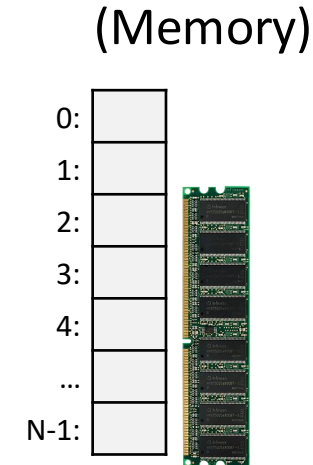
Program Counter (PC): Address 0

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

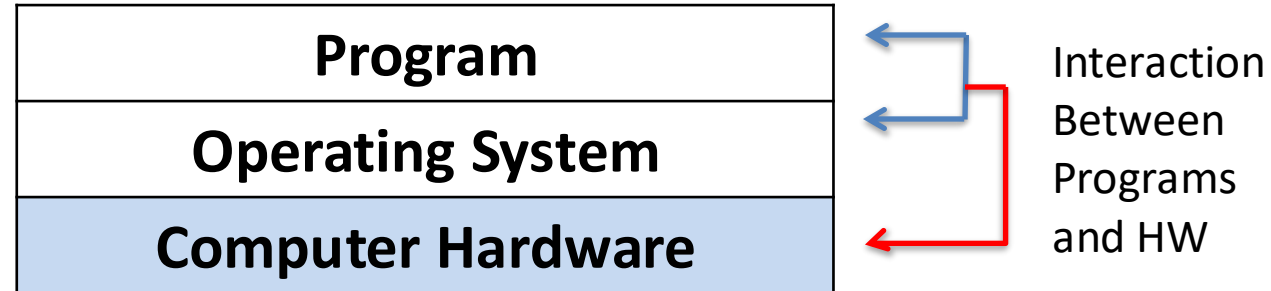
- acts on instruction bits to execute individual instructions
- PC value used to determine next instruction to execute



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

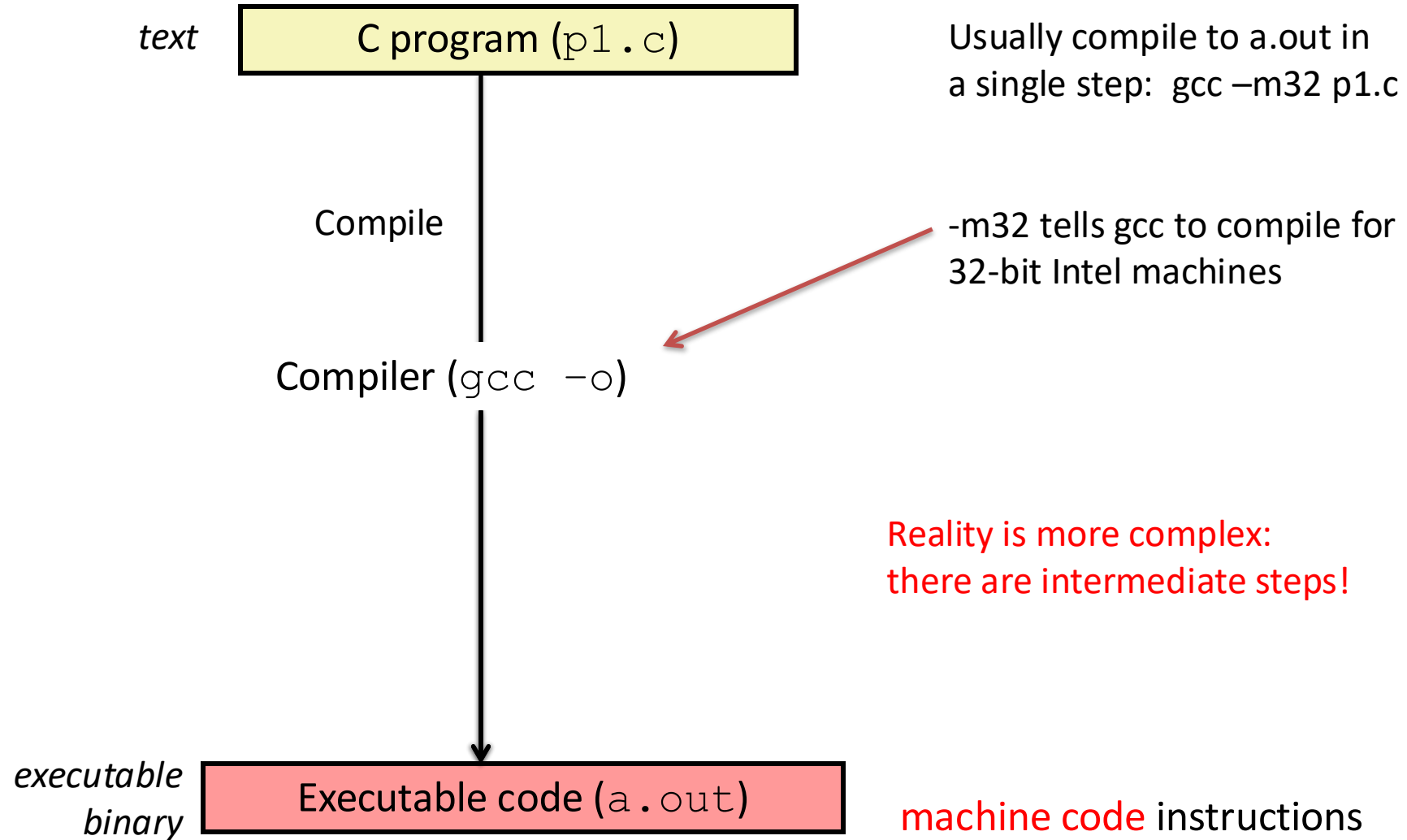


How a computer runs a program:

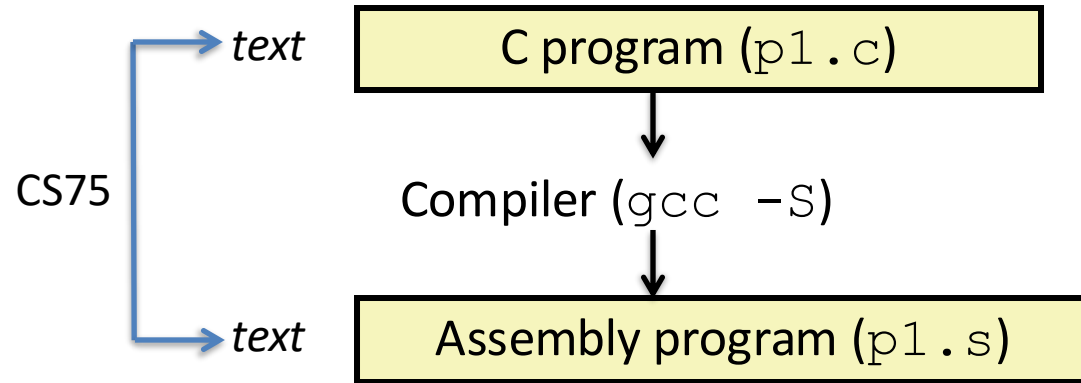


- We know: How HW Executes Instructions:
- **This Week: Instructions and ISA**
 - Program Encoding: C code to assembly code
 - Learn IA32 Assembly programming

Compilation Steps (.c to a.out)



Compilation Steps (.c to a.out)

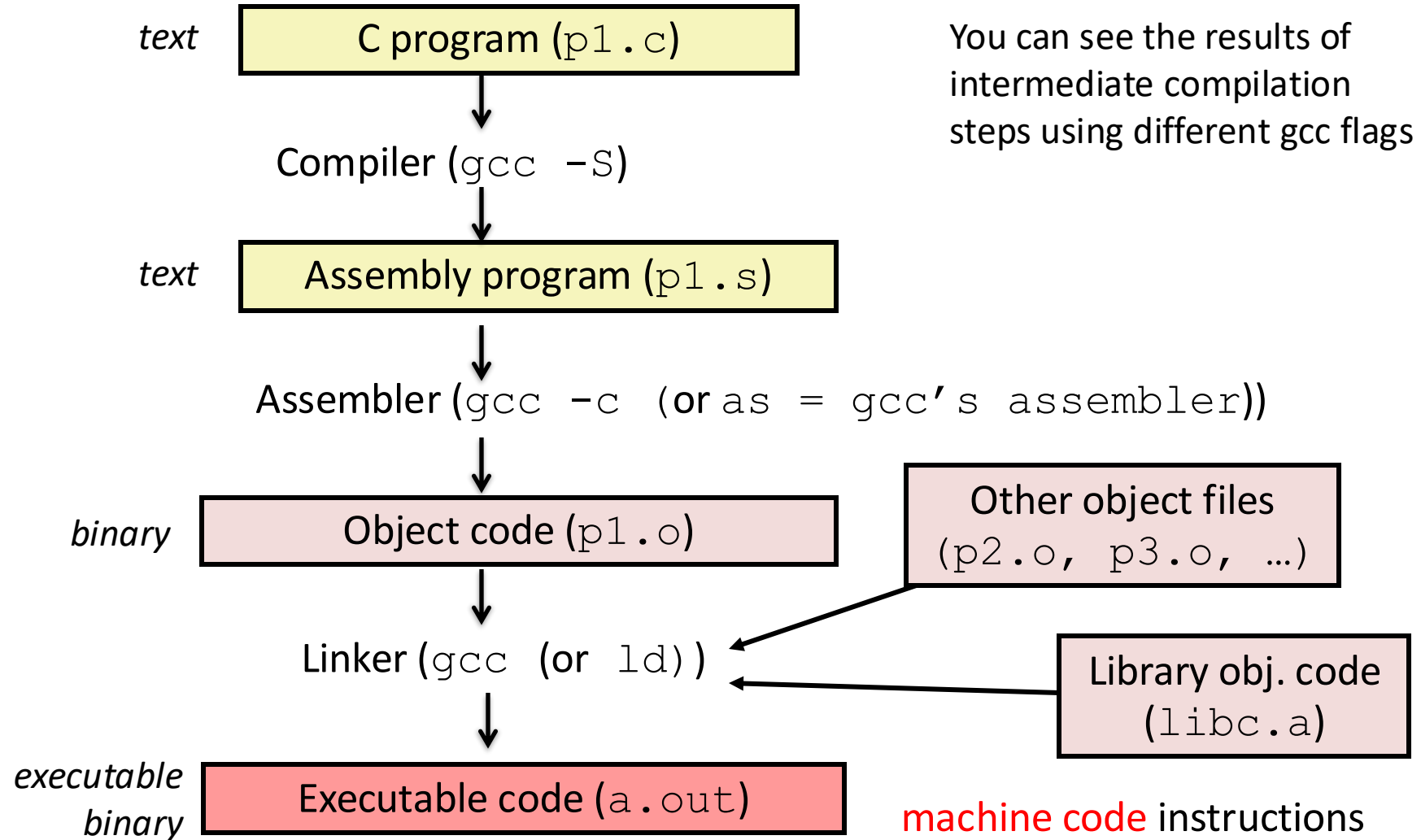


You can see the results of intermediate compilation steps using different gcc flags

executable binary Executable code (a.out)

machine code instructions

Compilation Steps (.c to a.out)



Machine Code

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

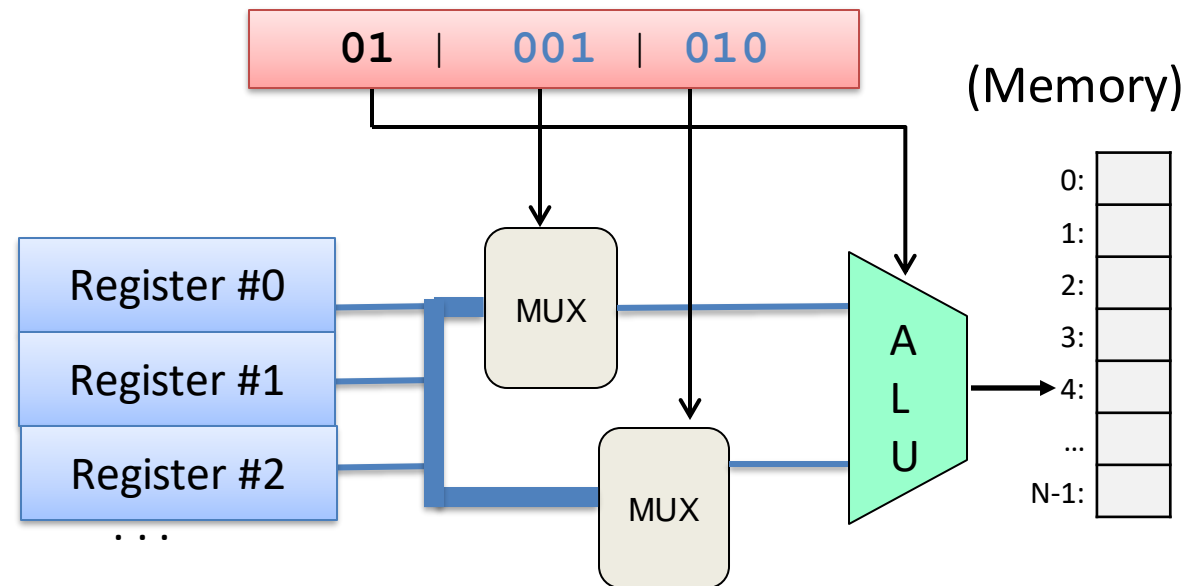
- some bits: encode the instruction (opcode bits)
- others encode operand(s)

(eg) **01**001010 **opcode** operands

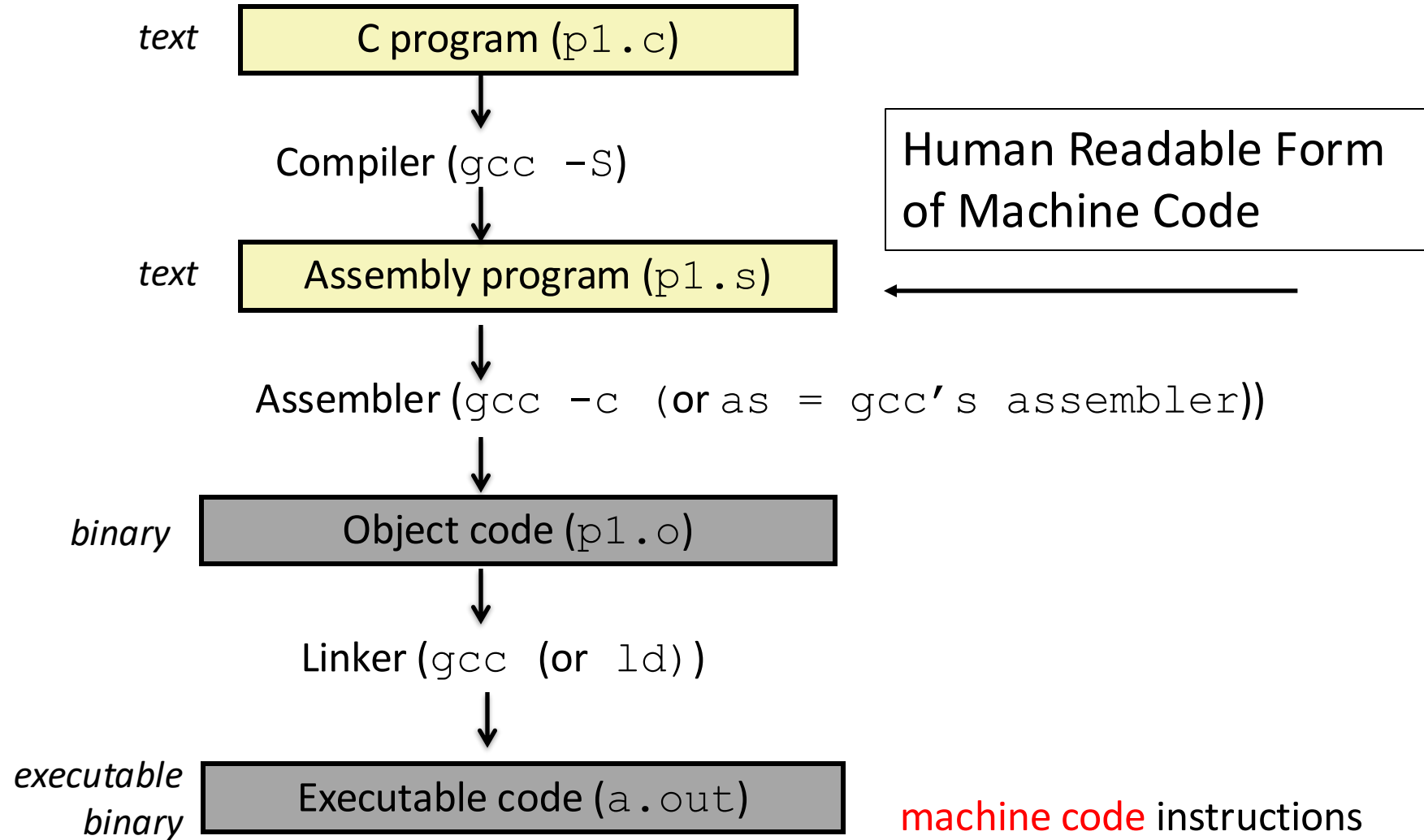
01 001 010

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

```
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
```

Machine Code (Hexadecimal)

```
55
89 E5
83 EC 10
C7 45 F8 0A 00 00 00
C7 45 FC 14 00 00 00
8B 45 FC
01 45 F8
B8 45 F8
C9
```

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

Object / Executable / Machine Code

Assembly

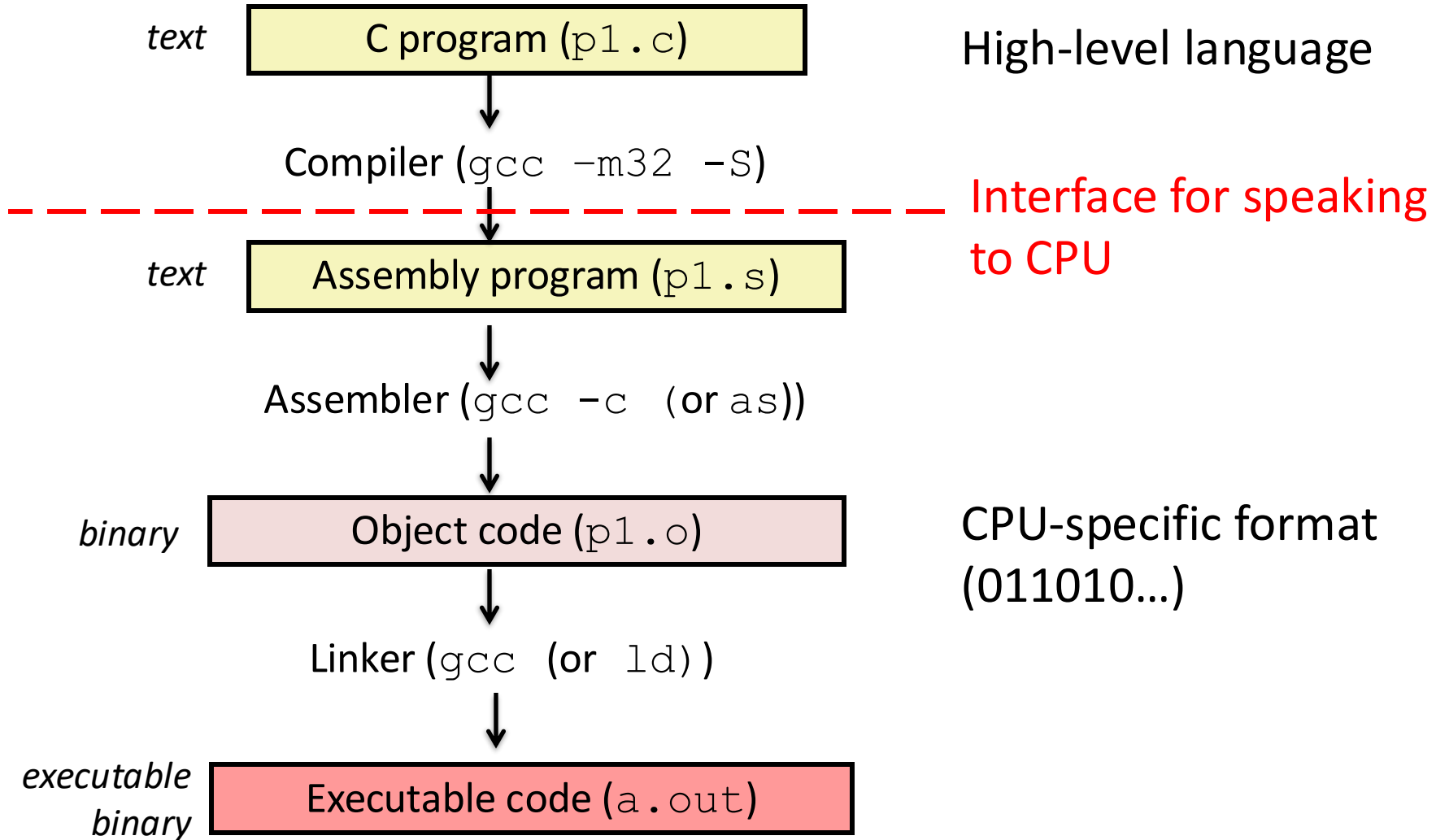
```
pushq %rbp
movq  %rsp,    %rbp
subq  $16,    %rsp
movq  $10,    -16(%rbp)
movq  $20,    -8(%rbp)
movq  -8(%rbp), %rax
addq  $rax,   -8(%rbp)
movq  -8(%rbp), %rax
leaveq
```

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

    a = a + b;

    return a;
}
```

Compilation Steps (.c to a.out)

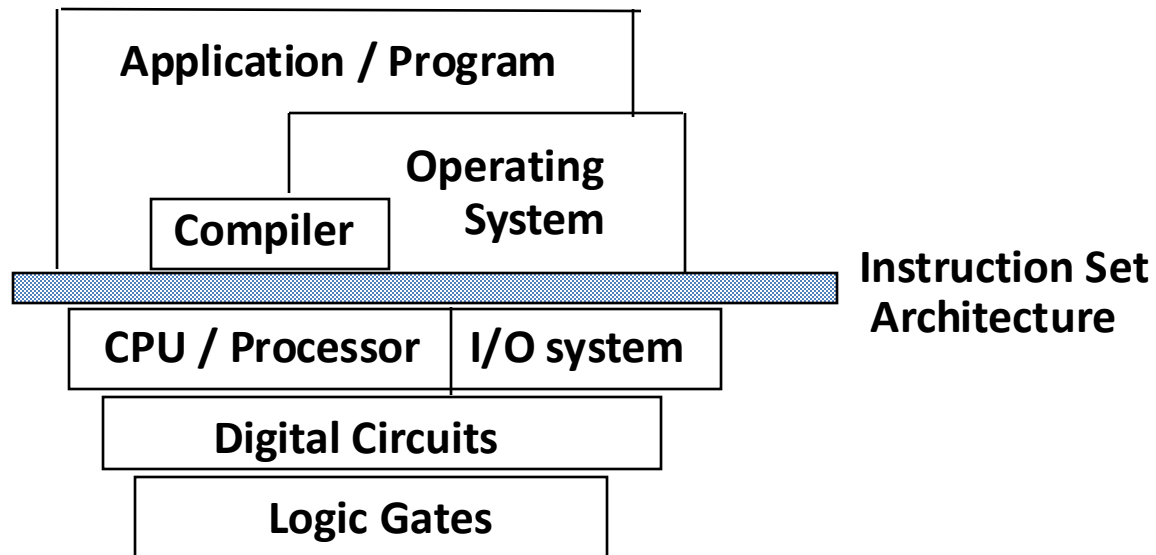


Instruction Set Architecture (ISA)

- 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)

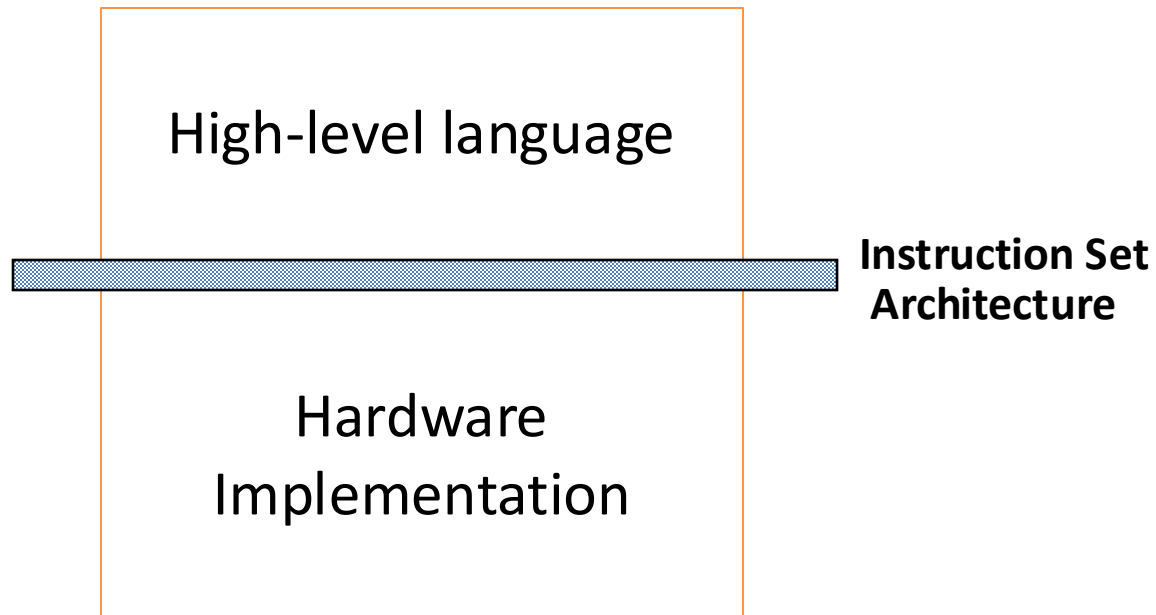
Instruction Set Architecture (ISA)

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



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

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)!

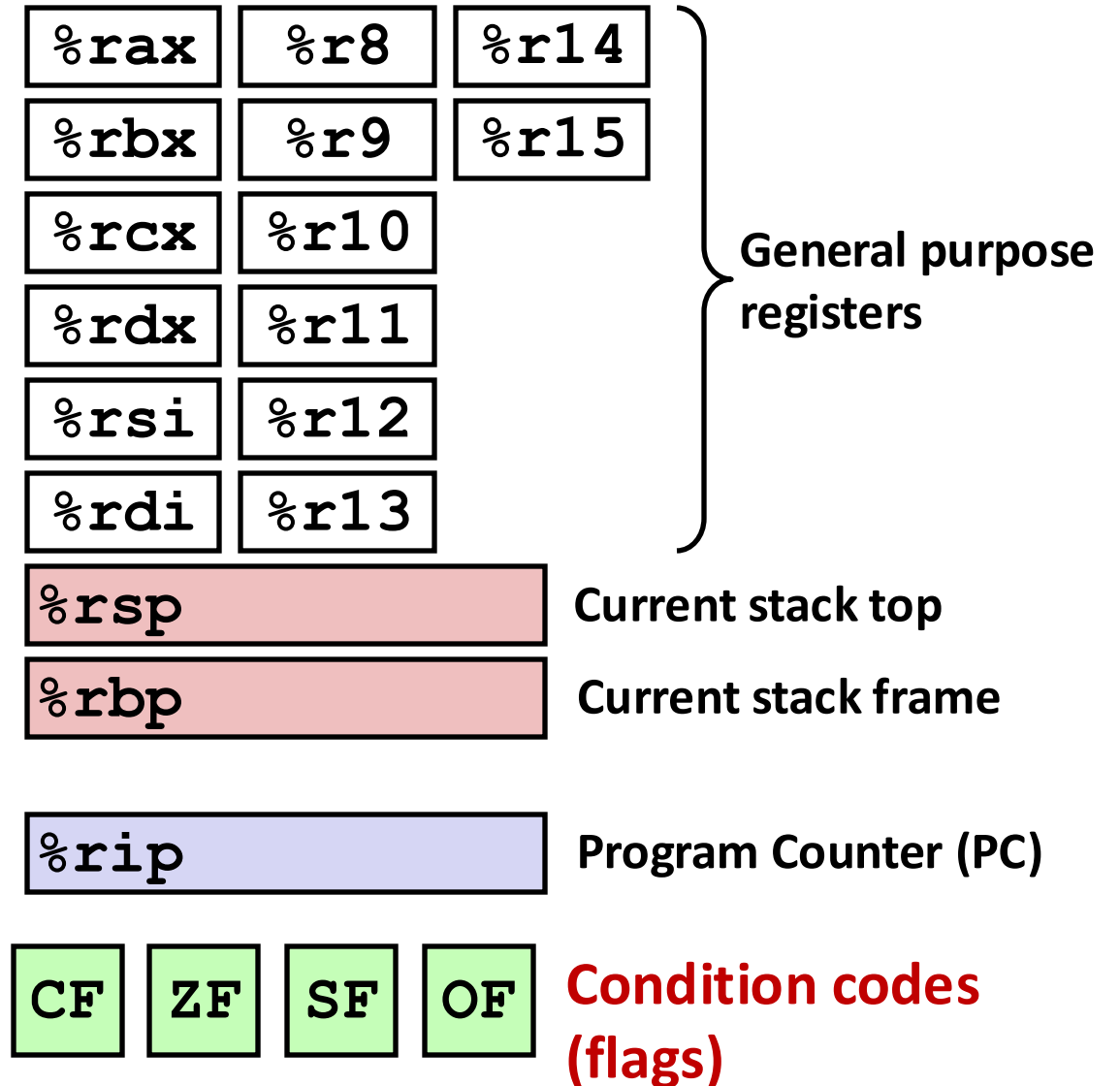
Instruction Set Architecture (ISA)

- 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

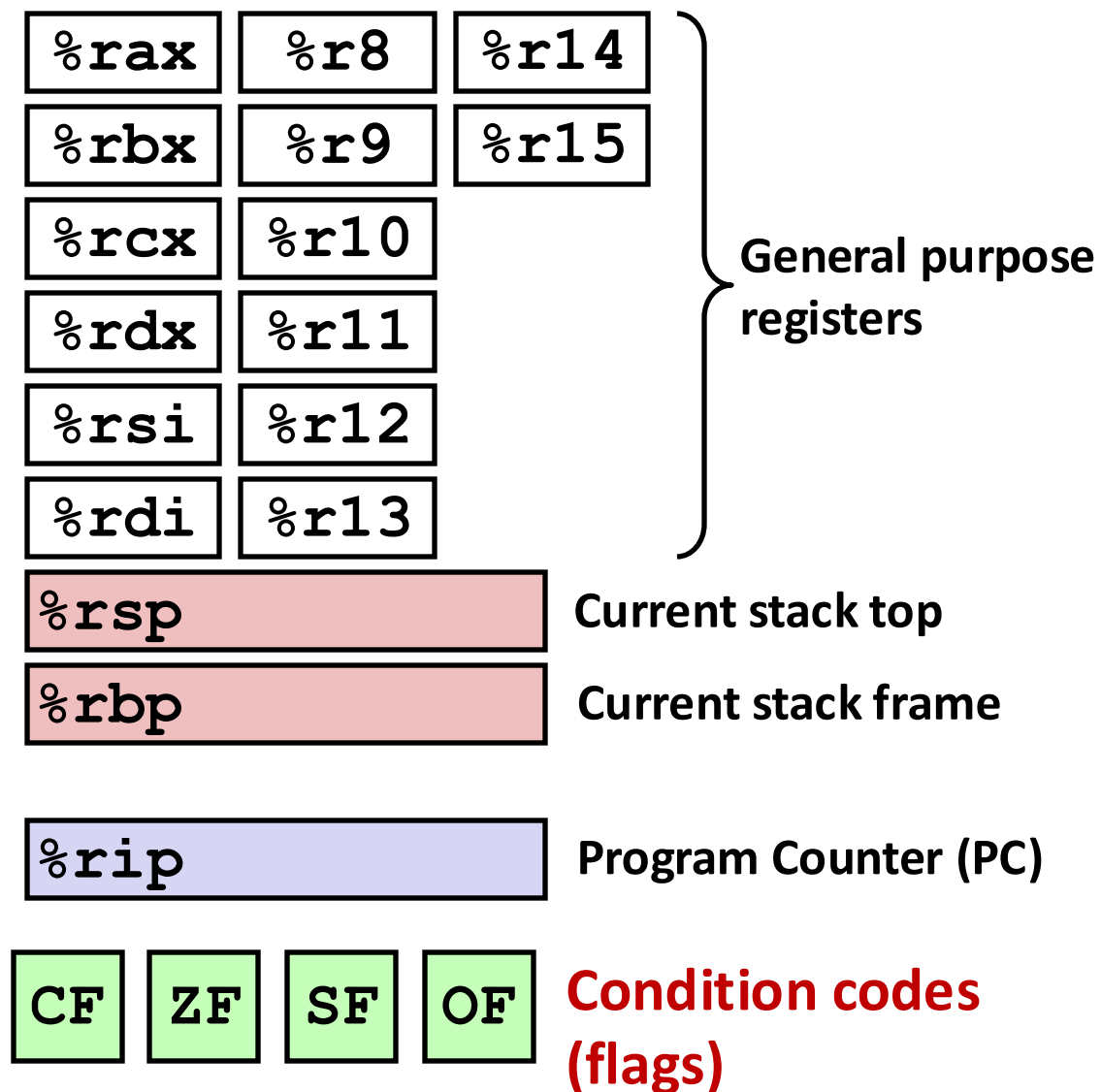
Working memory for currently executing program

- Temporary data: `%rax - %r15`
- 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)

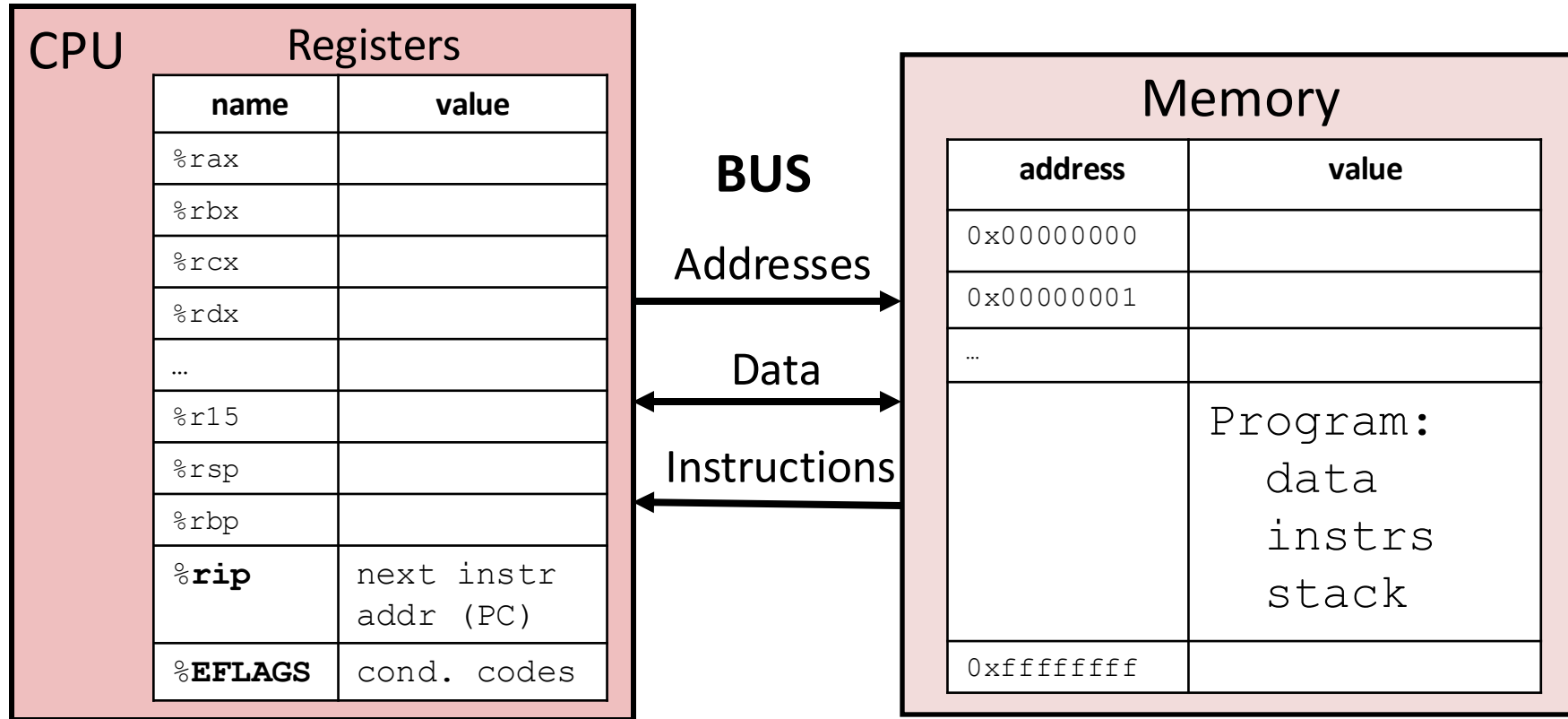


Component Registers

- 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).

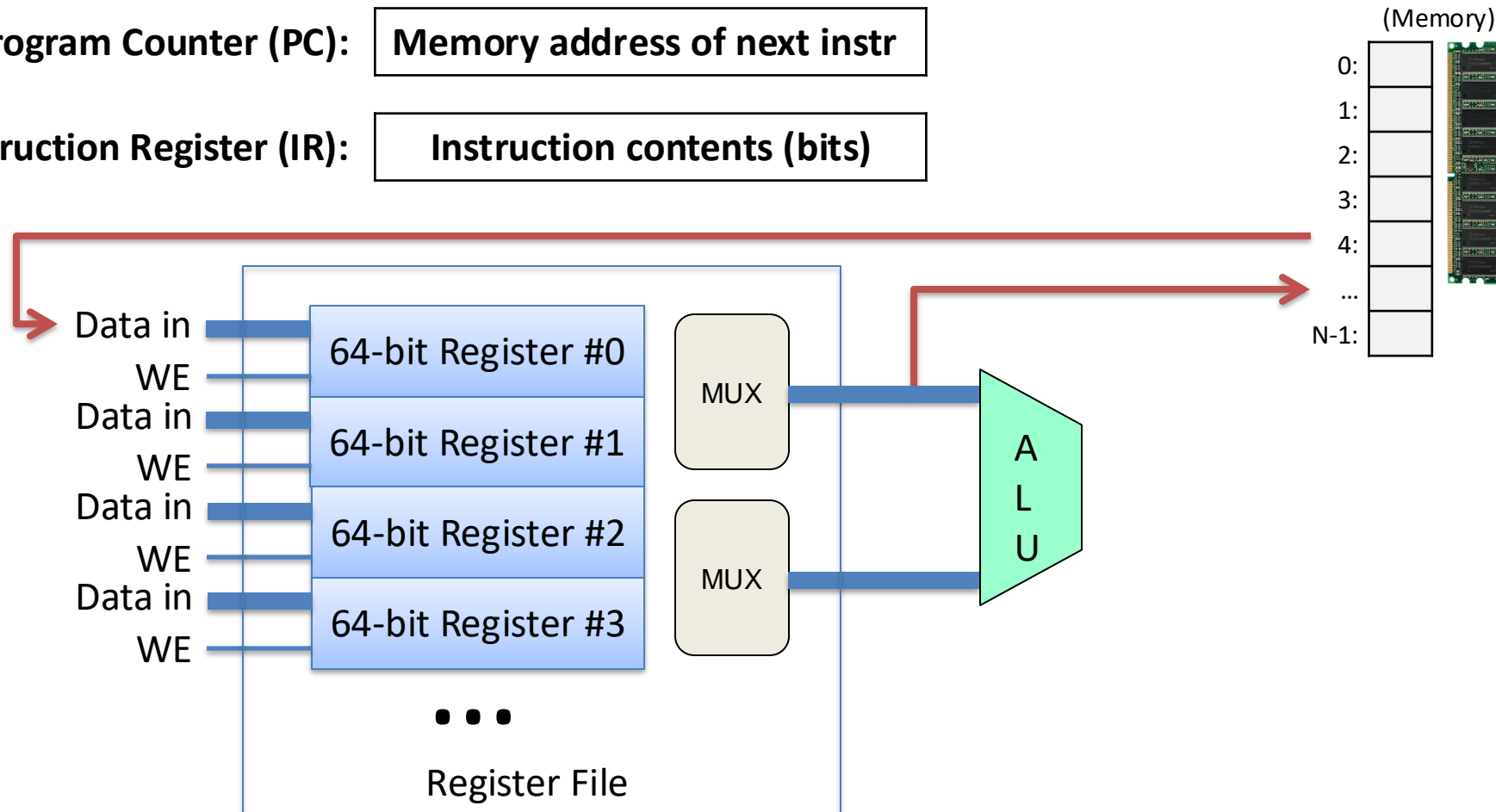
l -> 32 bits
q -> 64 bits

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

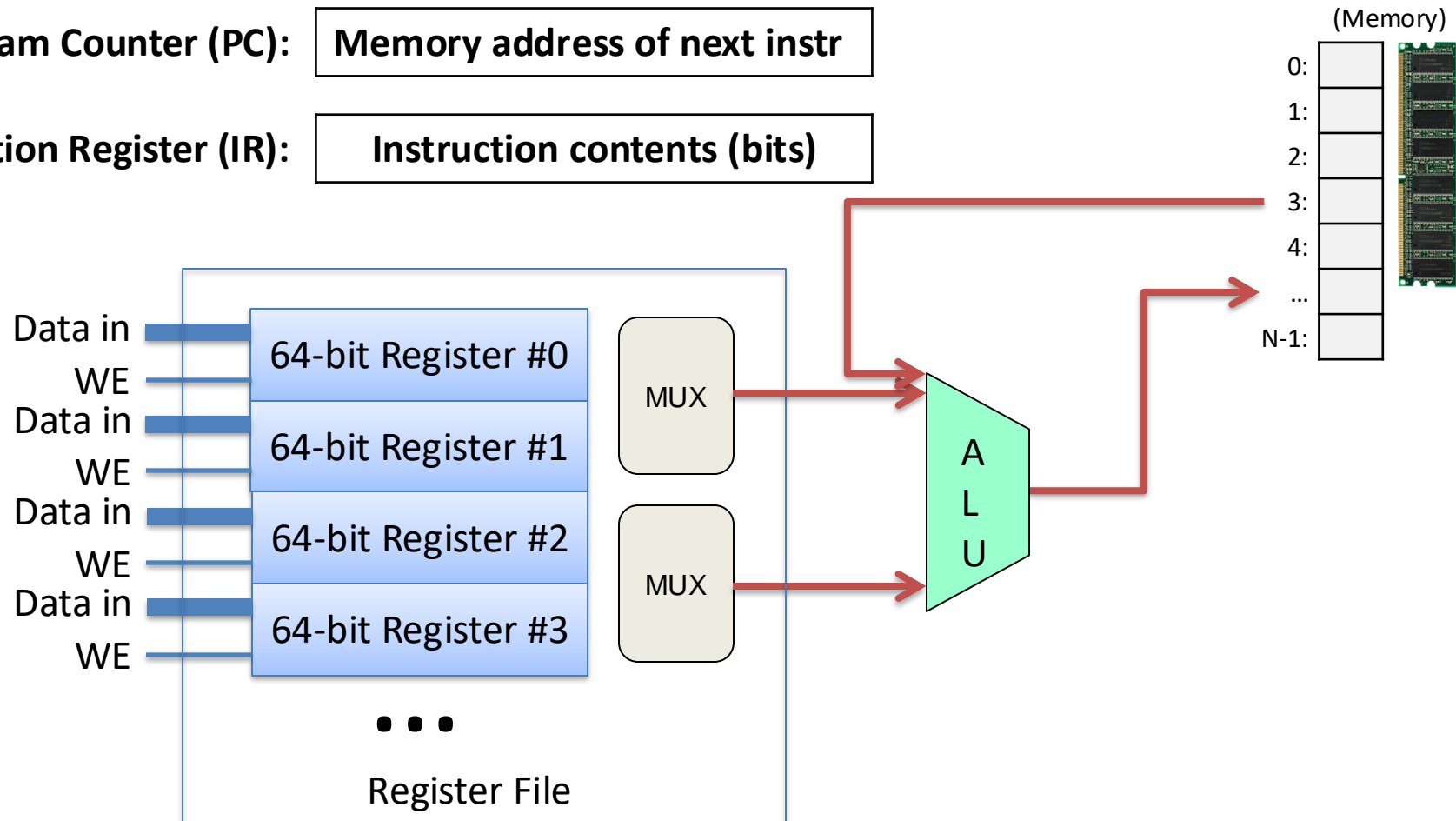
- Data movement
 - Move values between registers and memory
- Arithmetic
 - Uses ALU to compute a value
 - Examples: `addq`, `subq`

Arithmetic

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

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

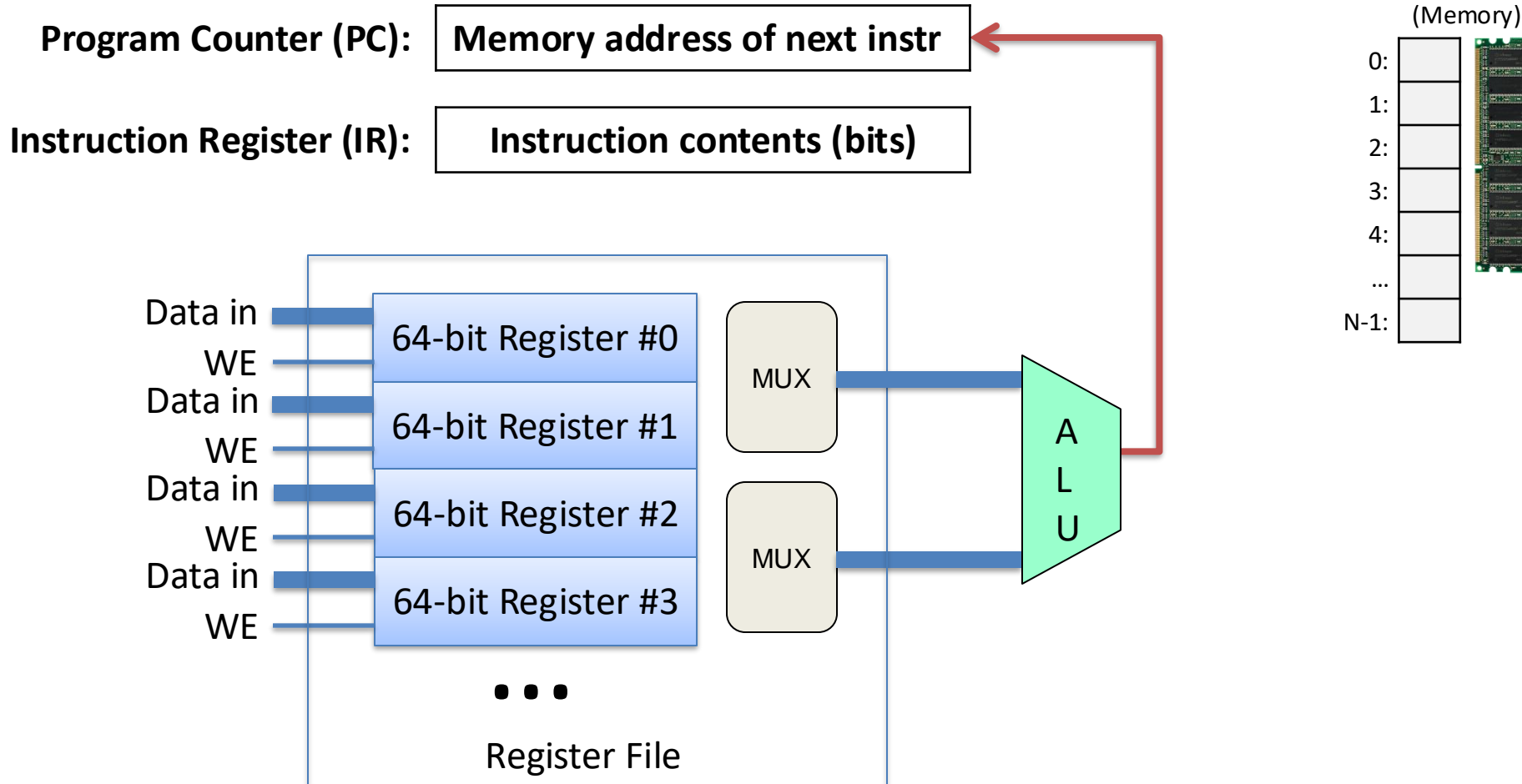


Types of assembly instructions

- 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.



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

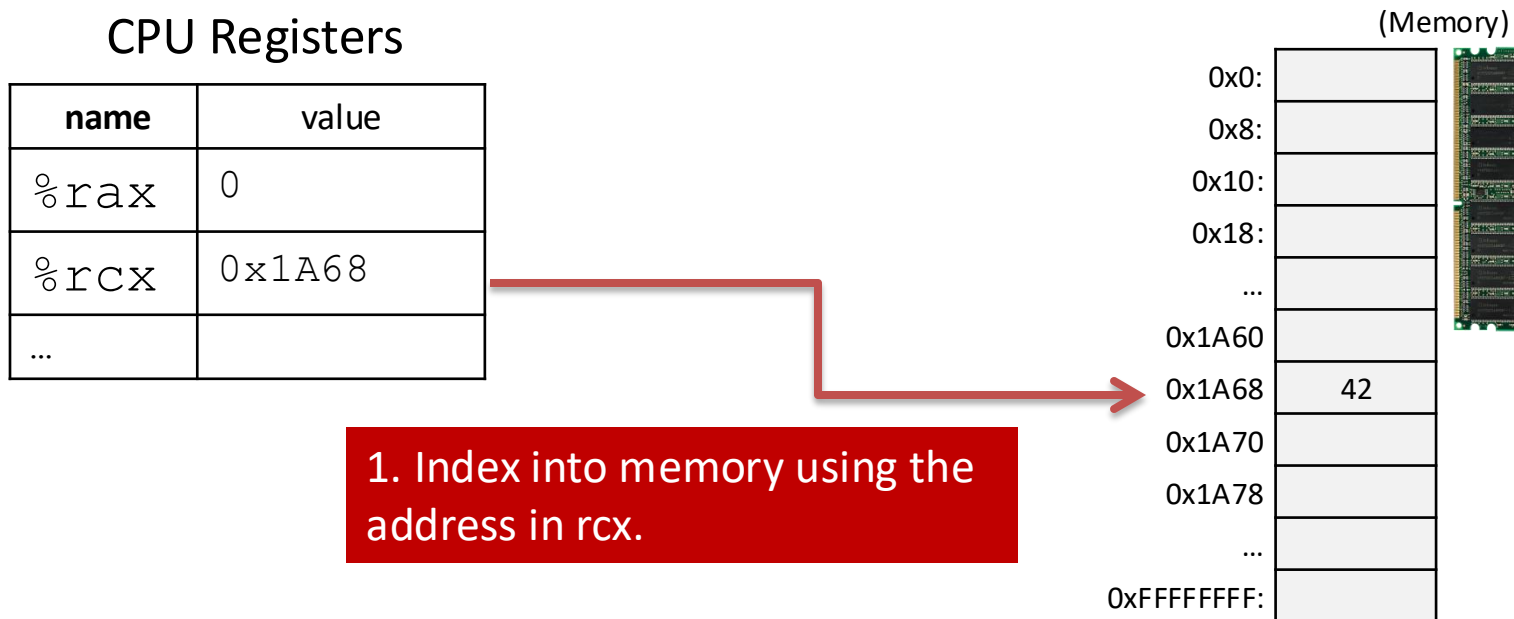
Addressing Modes

- 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

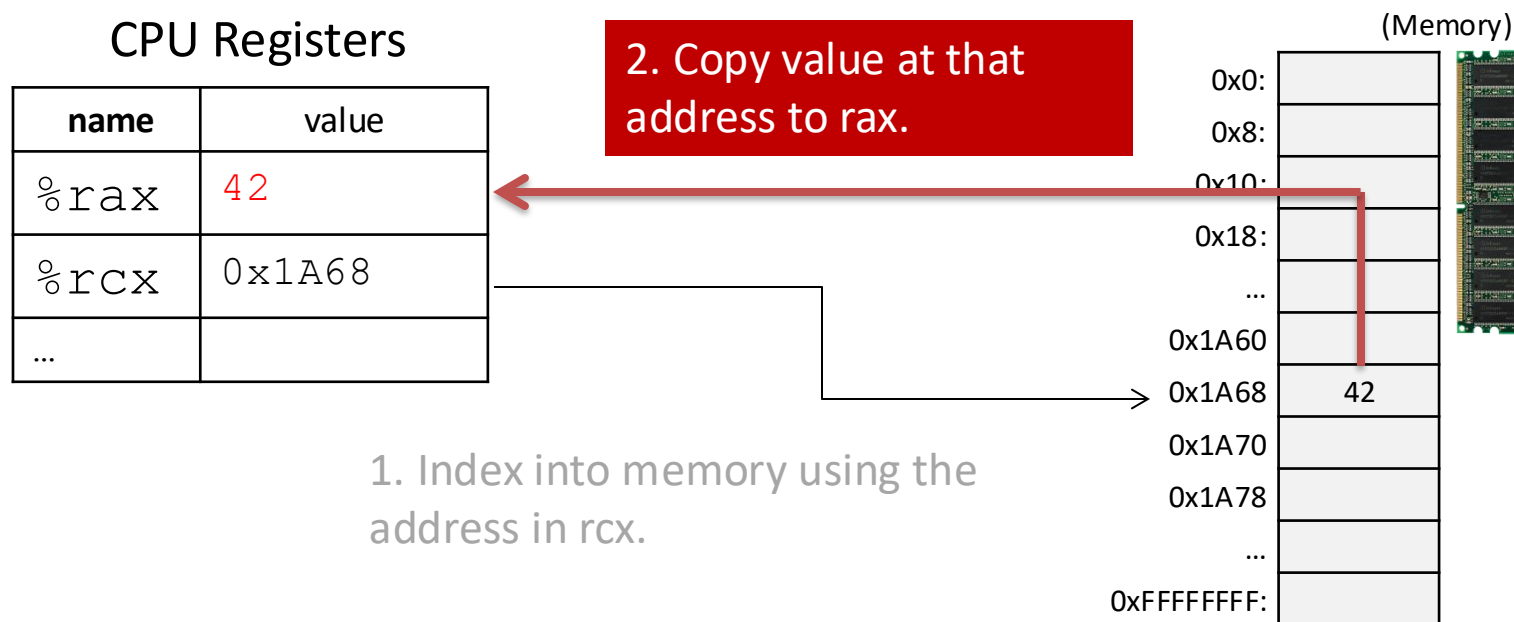
- Use the address in register %rcx to access memory,
- then, store result at that memory address in register %rax



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

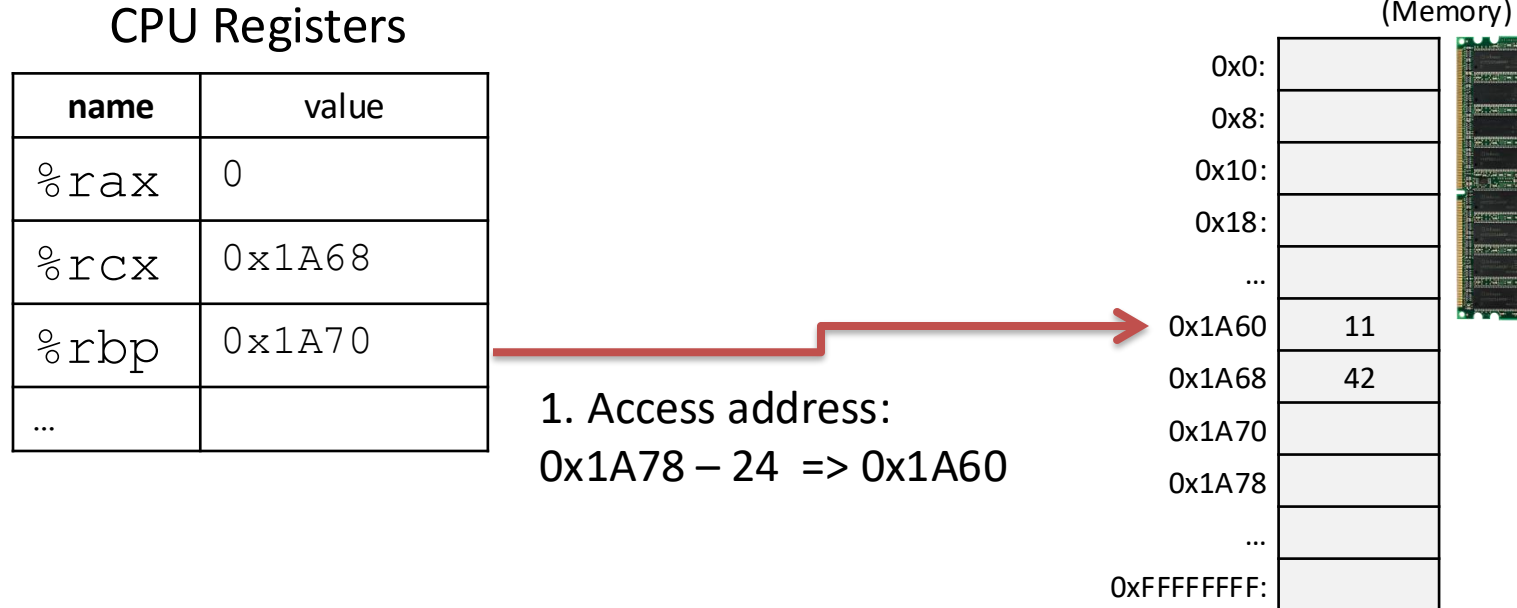
```
movq -24(%rbp), %rax
```

- 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
```

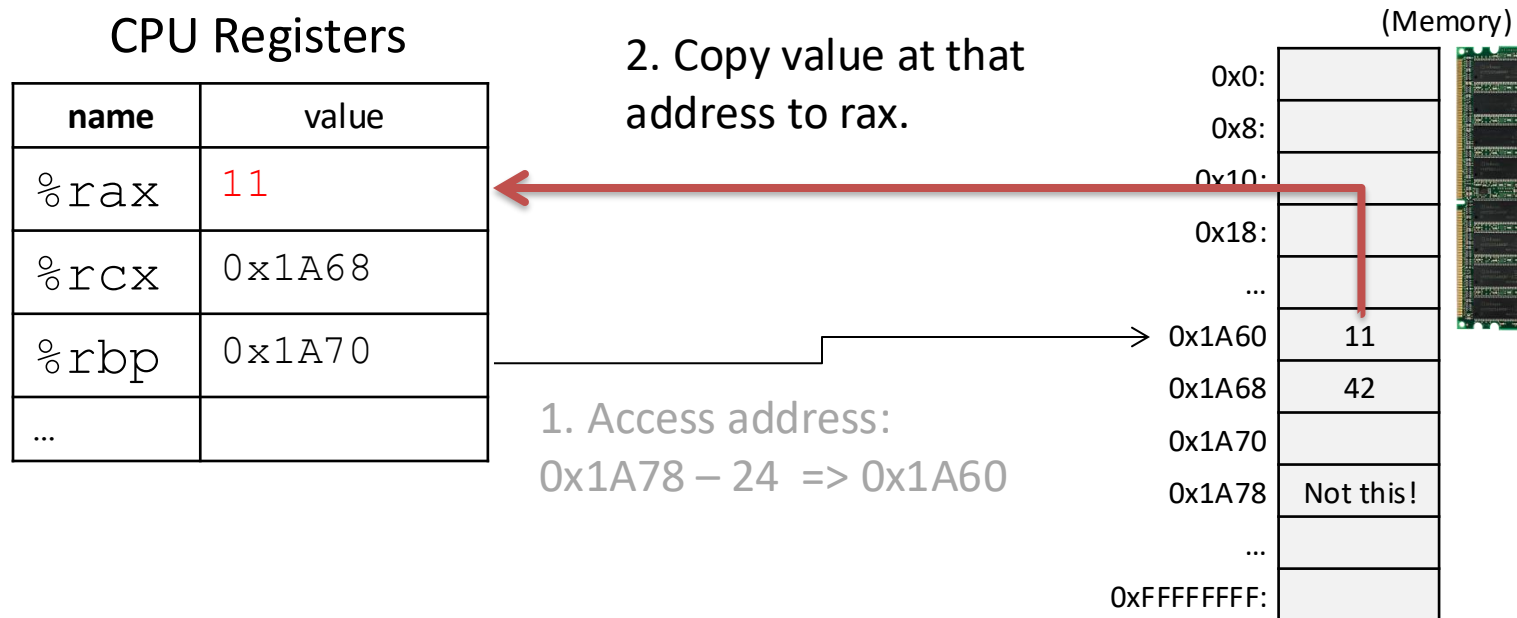
- Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.



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

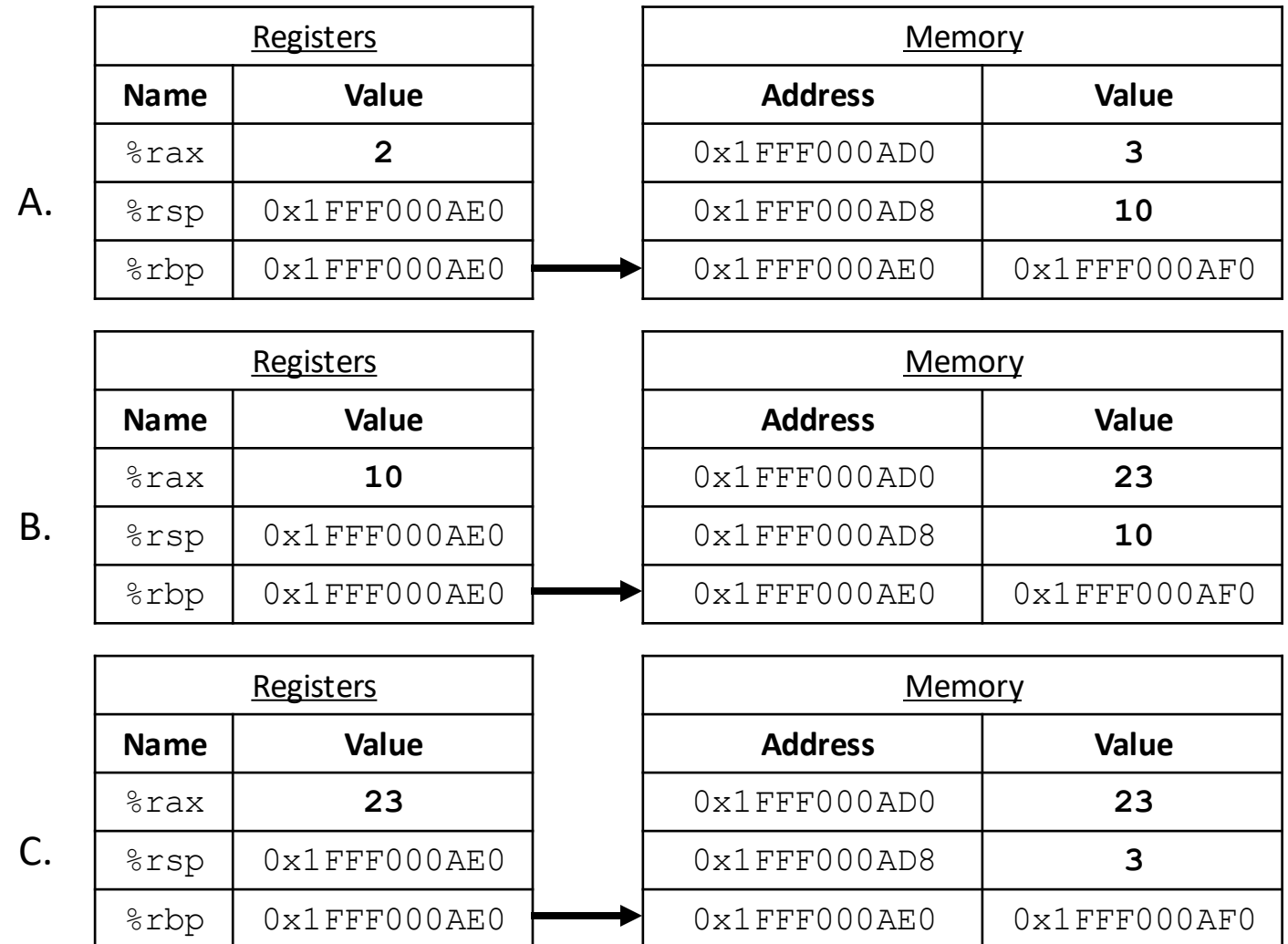
Memory	
Address	Value
...	
0x1FFF000AD0	0
0x1FFF000AD8	0
0x1FFF000AE0	0x1FFF000AF0
...	

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




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		Memory	
Name	Value	Address	Value
%rax	0	0x1FFF000AD0	0
%rsp	...AE0	0x1FFF000AD8	0
%rbp	...AE0	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
```

x is stored at rbp-8

y is stored at rbp-16

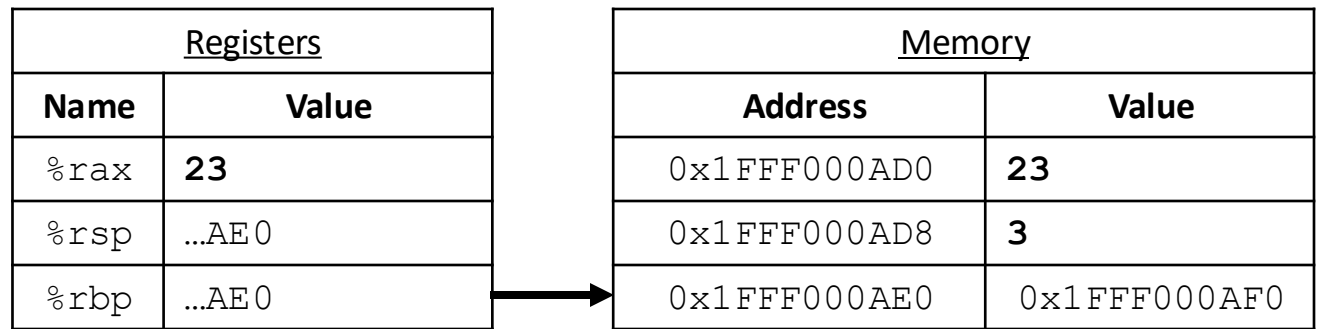
Solution

```
C code equivalent:  
    x = 3;  
    y = x + (10 << 1);
```

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

Subtract constant 16 from %rsp
Move constant 3 to address %rbp-8
Move constant 10 to register %rax
Shift the value in %rax left by 1 bit
Add the value at address %rbp-8 to %rax
Store the value in %rax at address rbp-16
Add constant 16 to %rsp

x is stored at rbp-8
y is stored at rbp-16



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?

$$z = x \wedge y$$

x is stored at %rbp-8

y is stored at %rbp-16

z is stored at %rbp-24

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)

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

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

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?

$x = y \gg 3 \mid x * 8$

x is stored at %rbp-8

y is stored at %rbp-16

z is stored at %rbp-24

Registers		Memory	
Name	Value	Address	Value
%rax	0	0x1FFF000AC8	(z)
%rdx	0	0x1FFF000AD0	(y)
%rsp	0x1FFF000AE0	0x1FFF000AD8	(x)
%rbp	0x1FFF000AE0	0x1FFF000AE0	0x1FFF000AF0
		...	

Solutions (other instruction sequences can work too!)

- $z = x \wedge y$

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

- $x = y \gg 3 \mid x * 8$

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