

# CS 31: Intro to Systems Digital Logic

Kevin Webb

Swarthmore College

September 13, 2022

# Reading Quiz

# Today

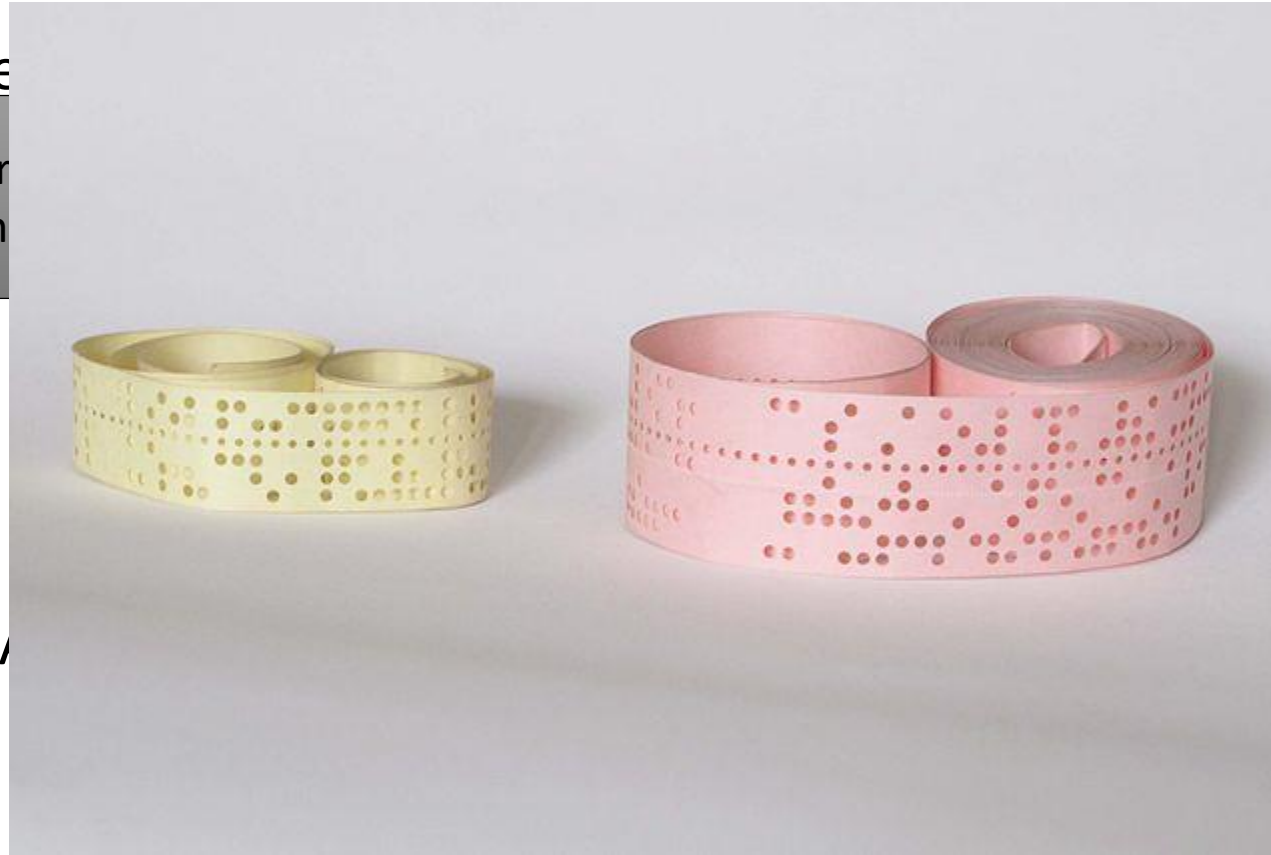
- Hardware basics
  - Machine memory models
  - Digital signals
  - Logic gates
- Manipulating/Representing values in hardware
  - Adders
  - Storage & memory (latches)

Circuits: Borrow some paper if you need to!

# Hardware Models (1940's)

- Harvard Architecture

Program  
Memory



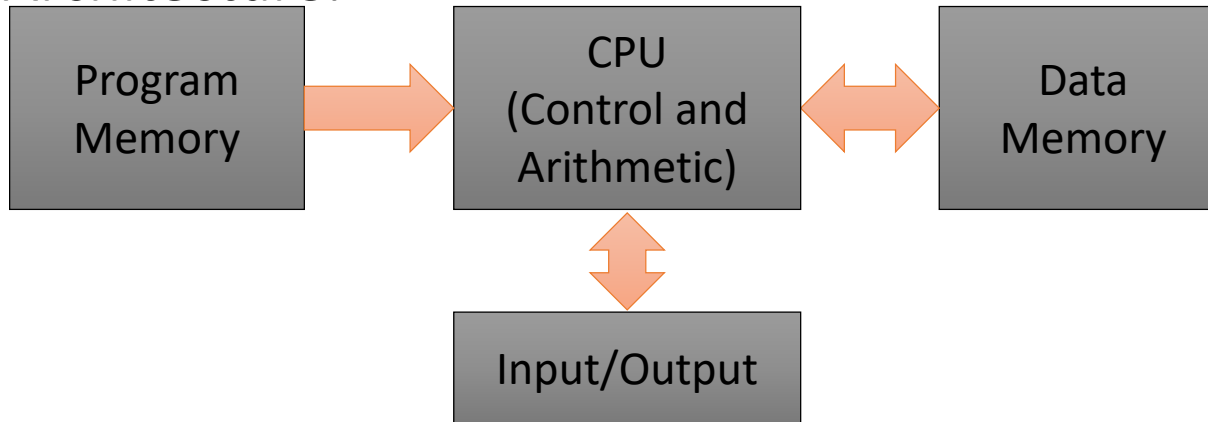
- Von Neumann Architecture

Data  
Memory

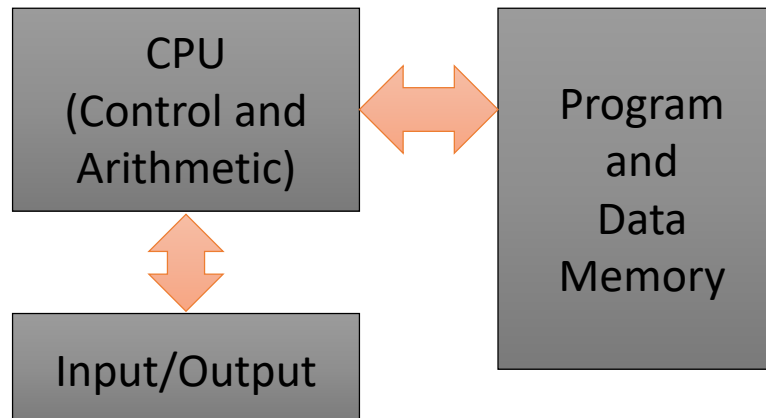


# Hardware Models (1940's)

- Harvard Architecture:

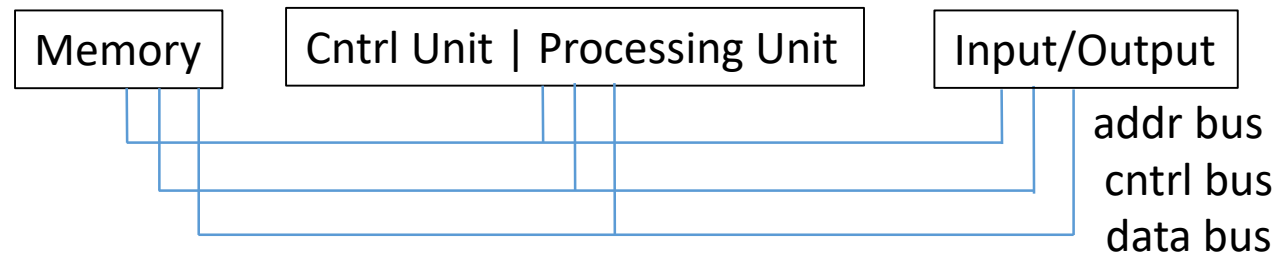


- Von Neumann Architecture:



# Von Neumann Architecture Model

- Computer is a generic computing machine:
  - Based on Alan Turing's Universal Turing Machine
  - Stored program model: computer stores program rather than encoding it (feed in data and instructions)
  - No distinction between data and instructions memory
- 5 parts connected by buses (wires):
  - Memory, Control, Processing, Input, Output



# Goal: Build a CPU (model)

## 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, Also: main memory (RAM)
3. Control: support/coordinate instruction execution  
(ex) fetch the next instruction to execute

# Abstraction

User / Programmer  
Wants low complexity



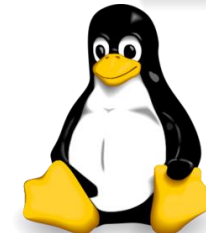
Applications  
Specific functionality



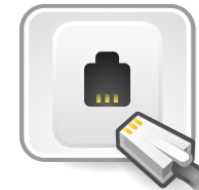
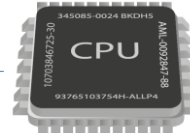
Software library  
Reusable functionality



Operating system  
Manage resources

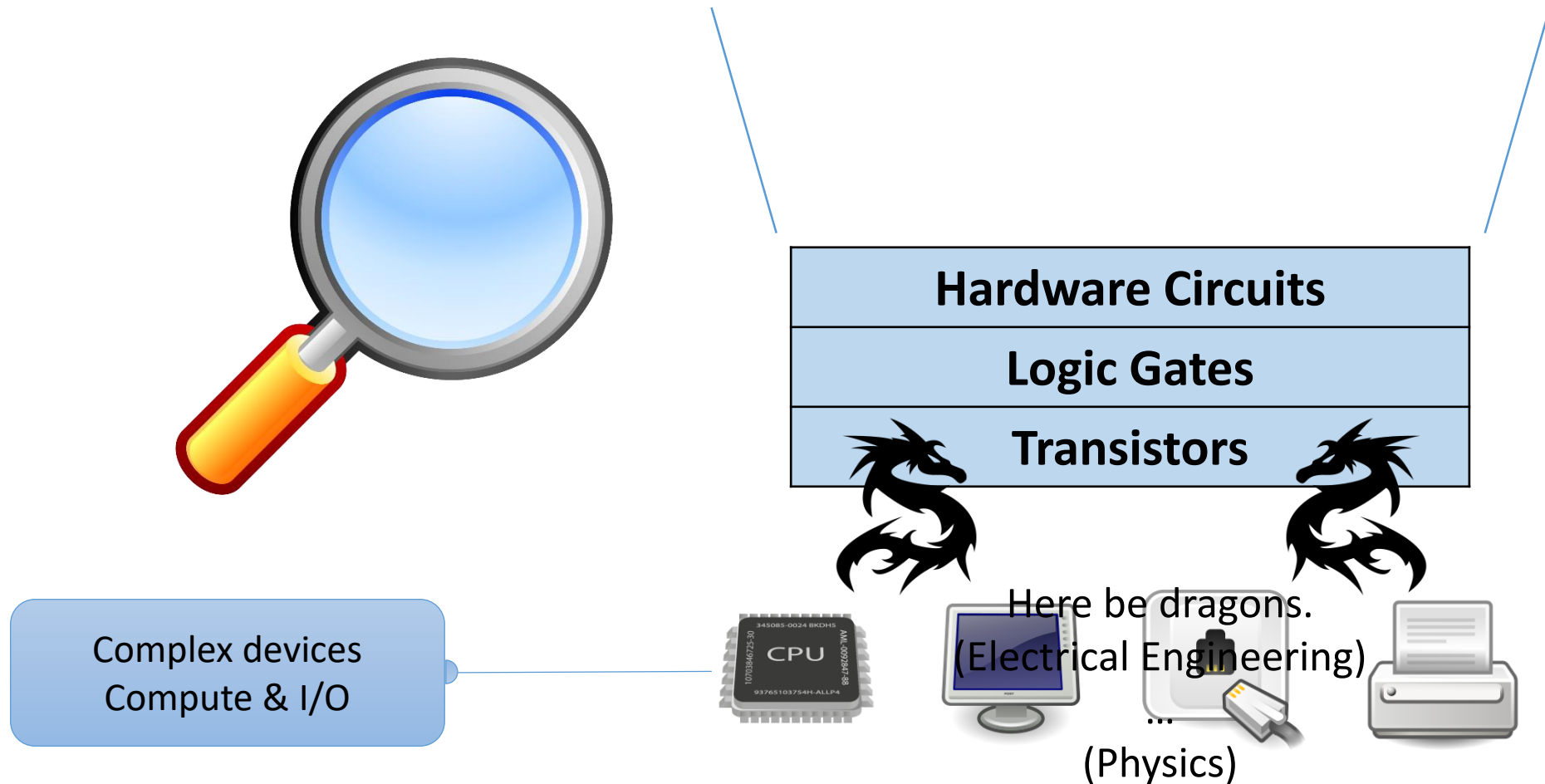


Complex devices  
Compute & I/O





# Abstraction

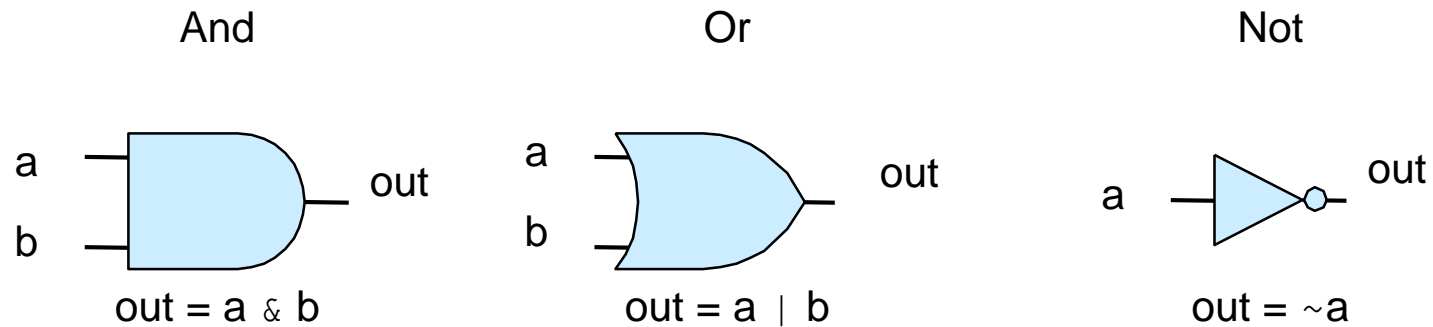


# Logic Gates

Input: Boolean value(s) (high and low voltages for 1 and 0)

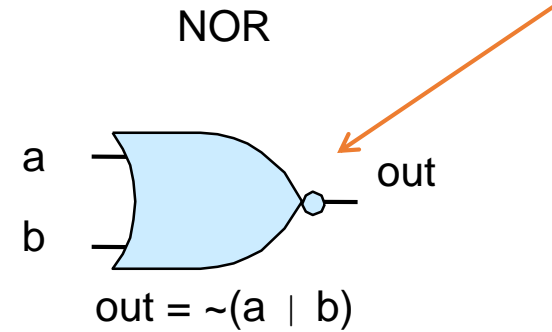
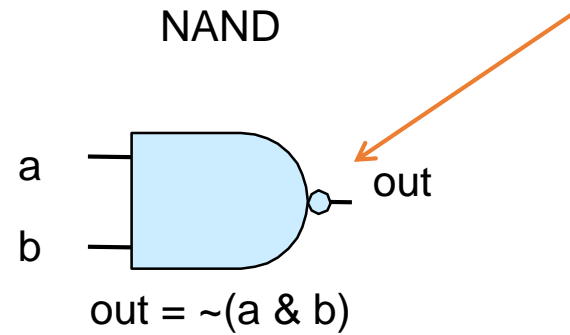
Output: Boolean value result of Boolean function

Always present, but may change when input changes



A	B	A & B	A   B	$\sim A$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

# More Logic Gates

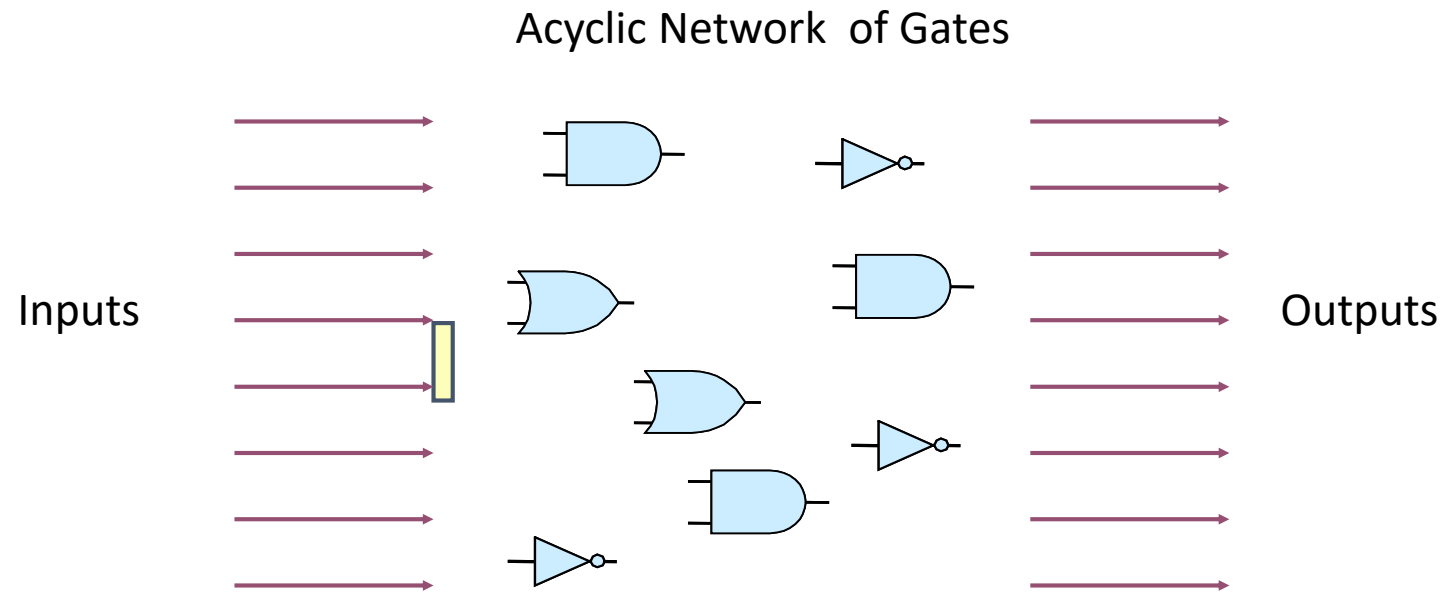


Note the circle on the output.  
This means "negate it."

A	B	A	NAND	B	A	NOR	B
0	0		1			1	
0	1		1			0	
1	0		1			0	
1	1		0			0	

# Combinational Logic Circuits

- Build up higher level processor functionality from basic gates

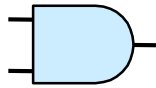


Outputs are boolean functions of inputs

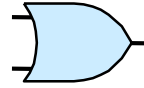
Outputs continuously respond to changes to inputs

# What does this circuit output?

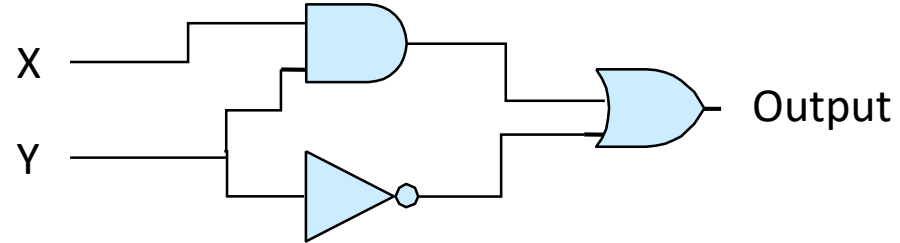
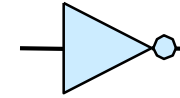
And



Or



Not



Clicker Choices

Clicker Choices						
X	Y	Out <sub>A</sub>	Out <sub>B</sub>	Out <sub>C</sub>	Out <sub>D</sub>	Out <sub>E</sub>
0	0	0	1	0	1	0
0	1	0	1	0	0	1
1	0	1	0	1	1	1
1	1	0	0	1	1	0

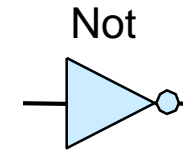
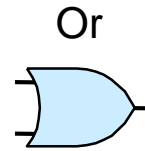
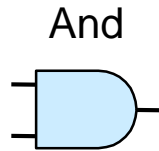
# What can we do with these?

- Build-up XOR from basic gates (AND, OR, NOT)

A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Q: When is  $A \wedge B == 1$ ?

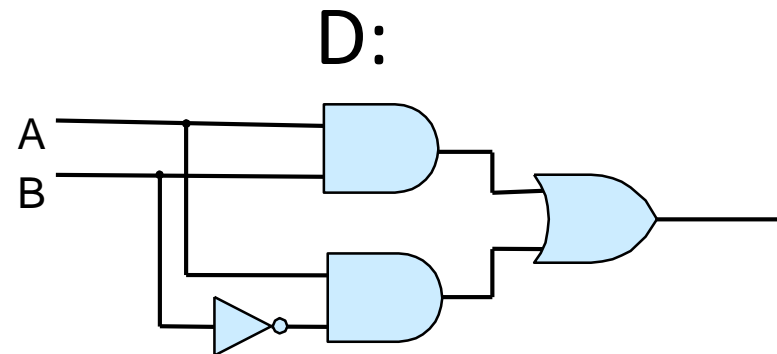
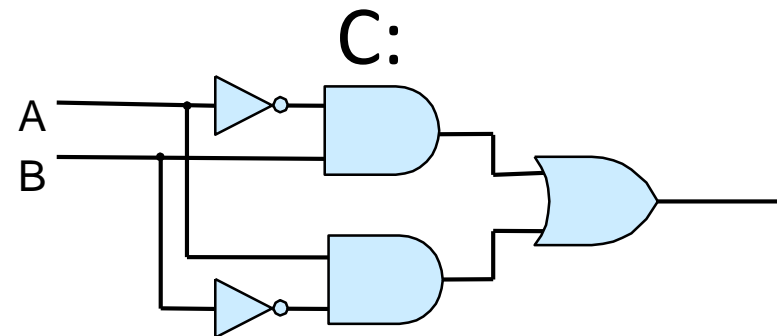
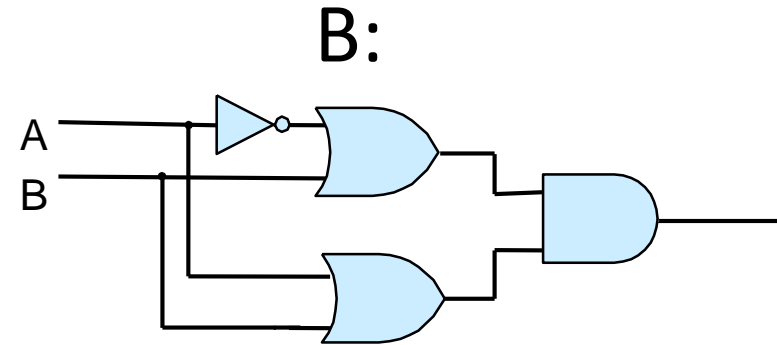
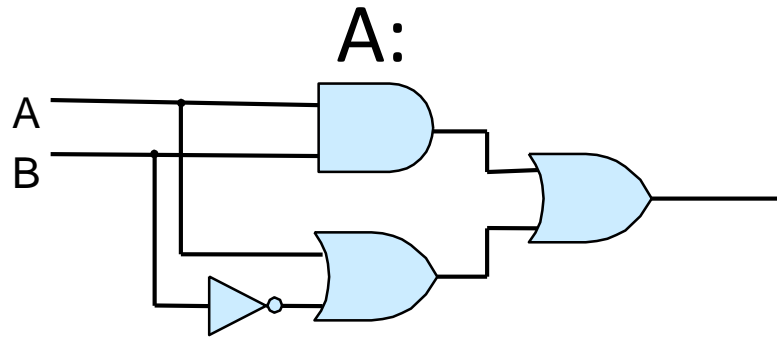
# Which of these is an XOR circuit?



Draw an XOR circuit using AND, OR, and NOT gates.

I'll show you the clicker options after you've had some time.

# Which of these is an XOR circuit?

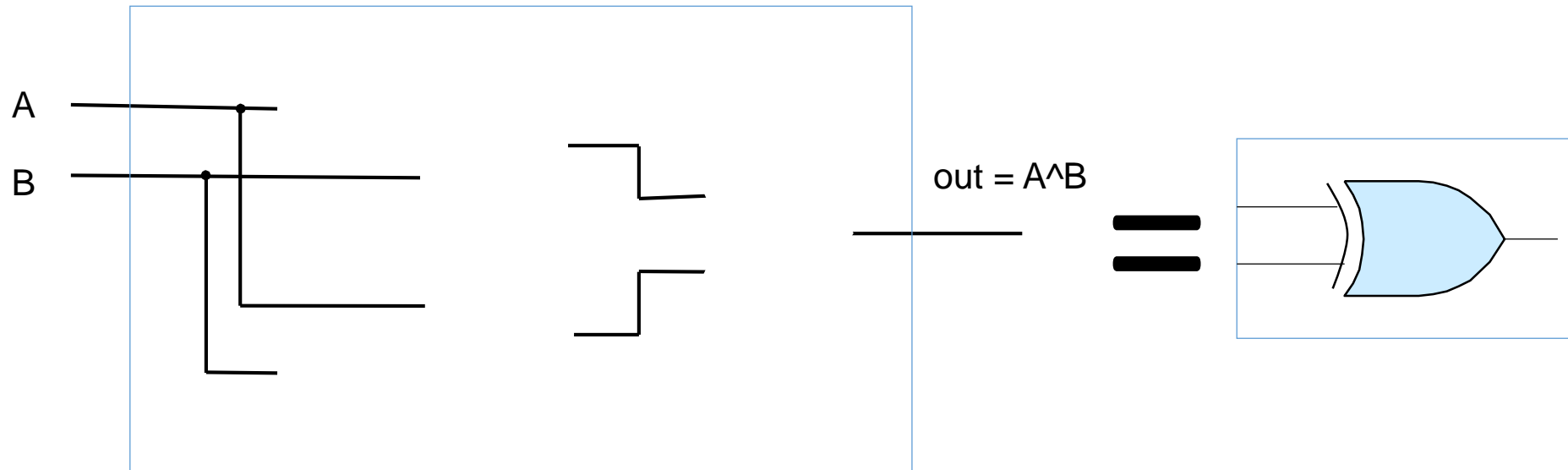


E: None of these are XOR.



# XOR Circuit: Abstraction

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



A:0 B:0 A^B:

A:0 B:1 A^B:

A:1 B:0 A^B:

A:1 B:1 A^B:

# 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

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

# Digital Circuits - Building a CPU

Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality  
(ex) adder to add two values together

Start with ALU components (e.g., adder)

Combine into ALU!

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

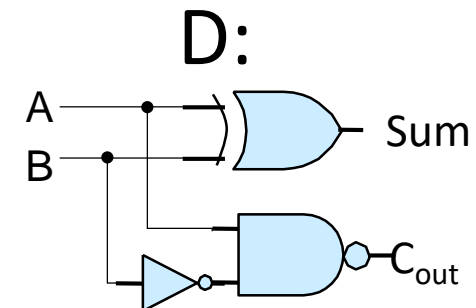
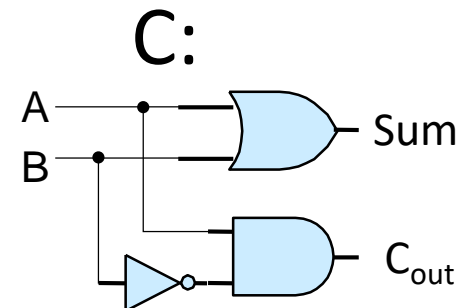
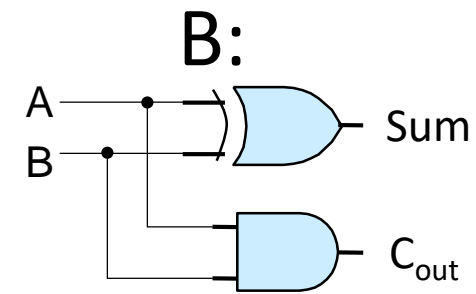
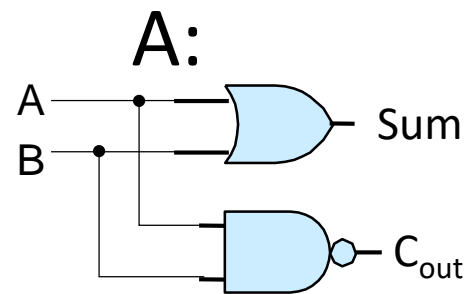
# Arithmetic Circuits

- 1 bit adder:  $A+B$
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

A	B	Sum (A + B)	C <sub>out</sub>
0	0		
0	1		
1	0		
1	1		

# Which of these circuits is a one-bit adder?

A	B	Sum (A + B)	C <sub>out</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



# More than one bit?

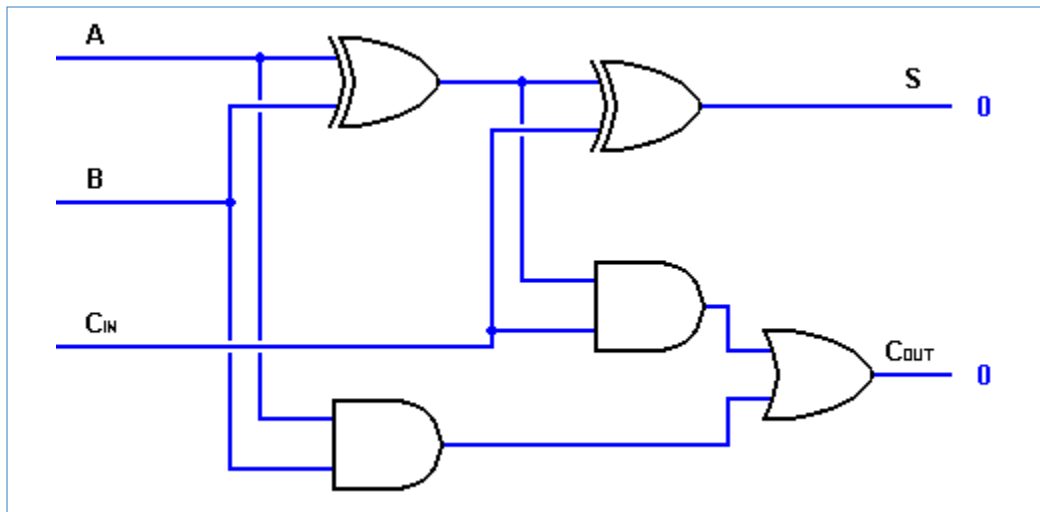
- When adding, sometimes have *carry in* too

$$\begin{array}{r} 0011010 \\ + 0001111 \\ \hline \end{array}$$

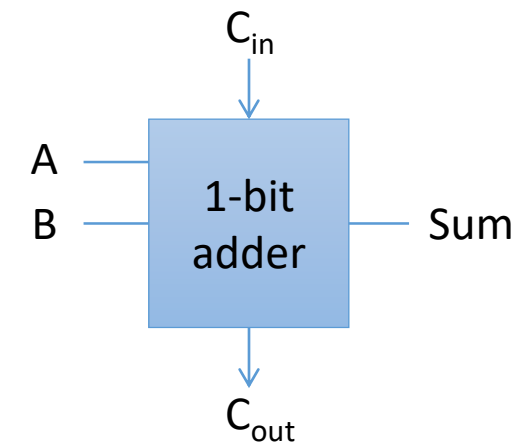
# One-bit (full) adder

Need to include:  
Carry-in & Carry-out

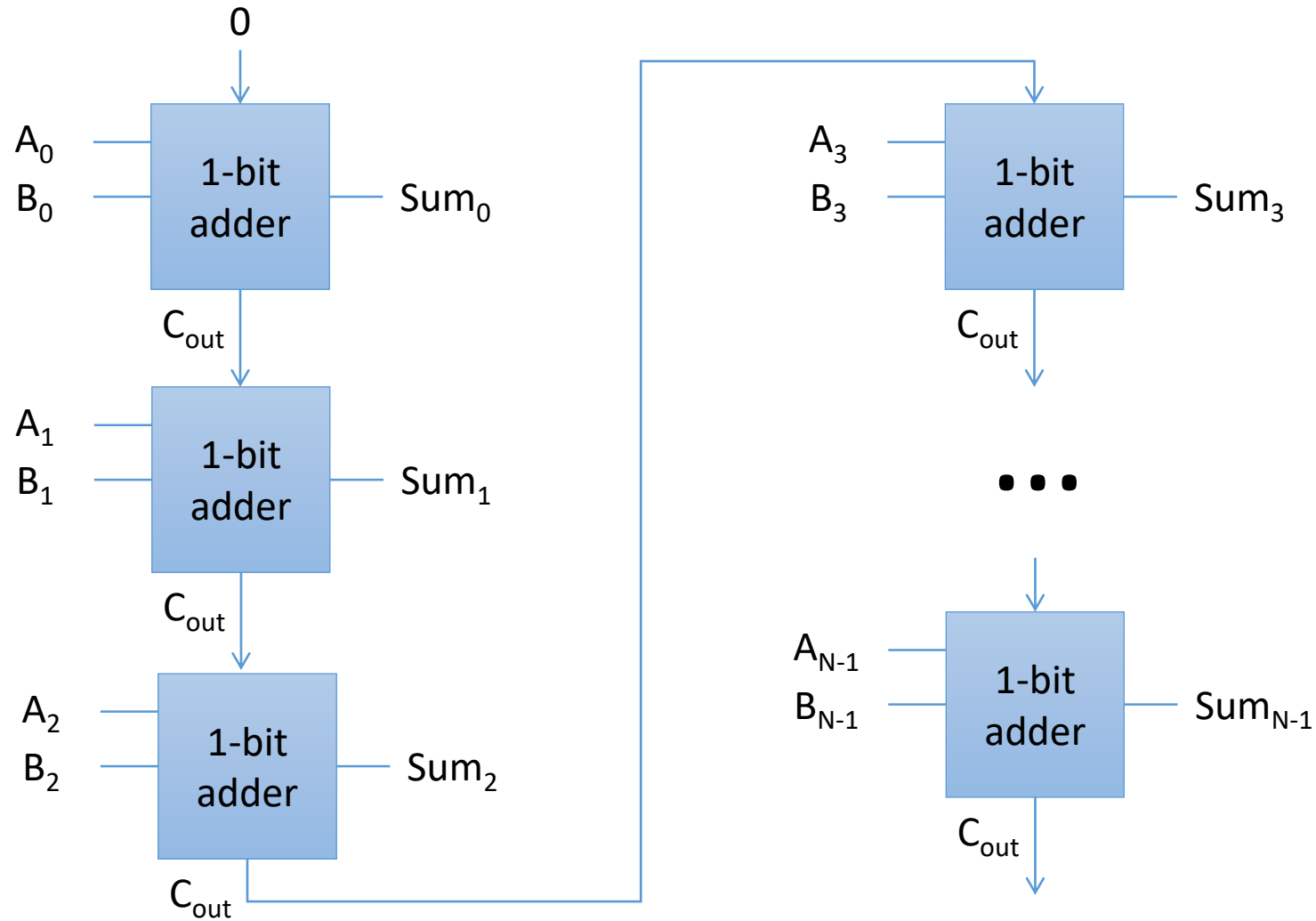
A	B	$C_{in}$	Sum	$C_{out}$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



=



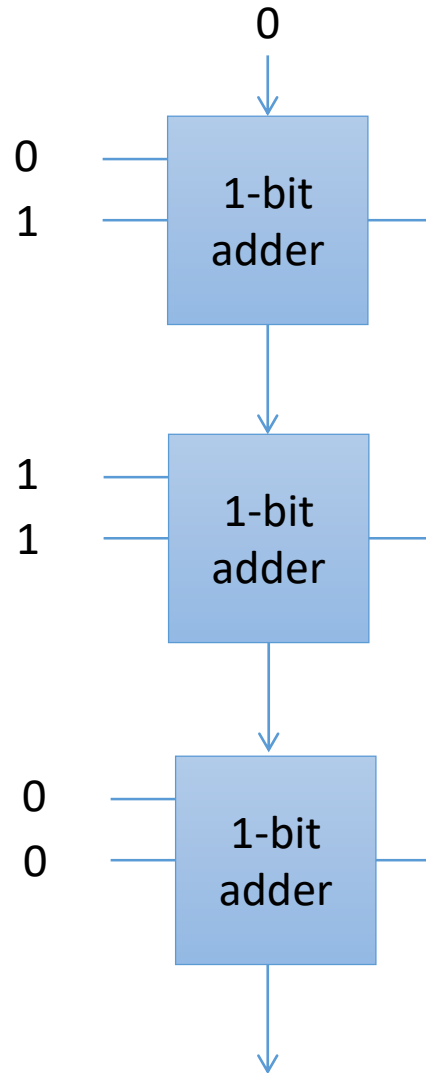
# Multi-bit Adder (Ripple-carry Adder)



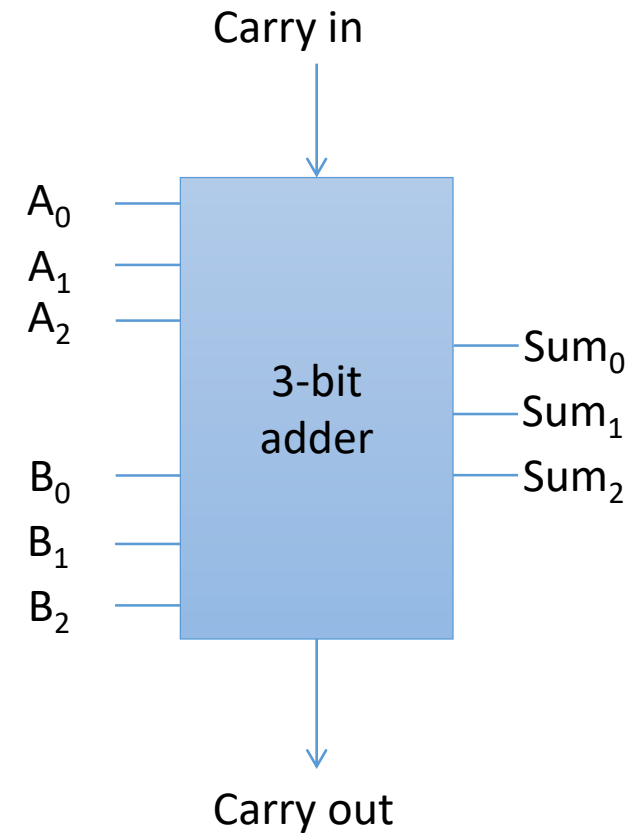


# Three-bit Adder (Ripple-carry Adder)

$$\begin{array}{r} 010 \text{ (2)} \\ + \underline{011 \text{ (3)}} \\ \hline \end{array}$$



=



# 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:

$A_0$

$A_1$

$A_2$

$B_0$

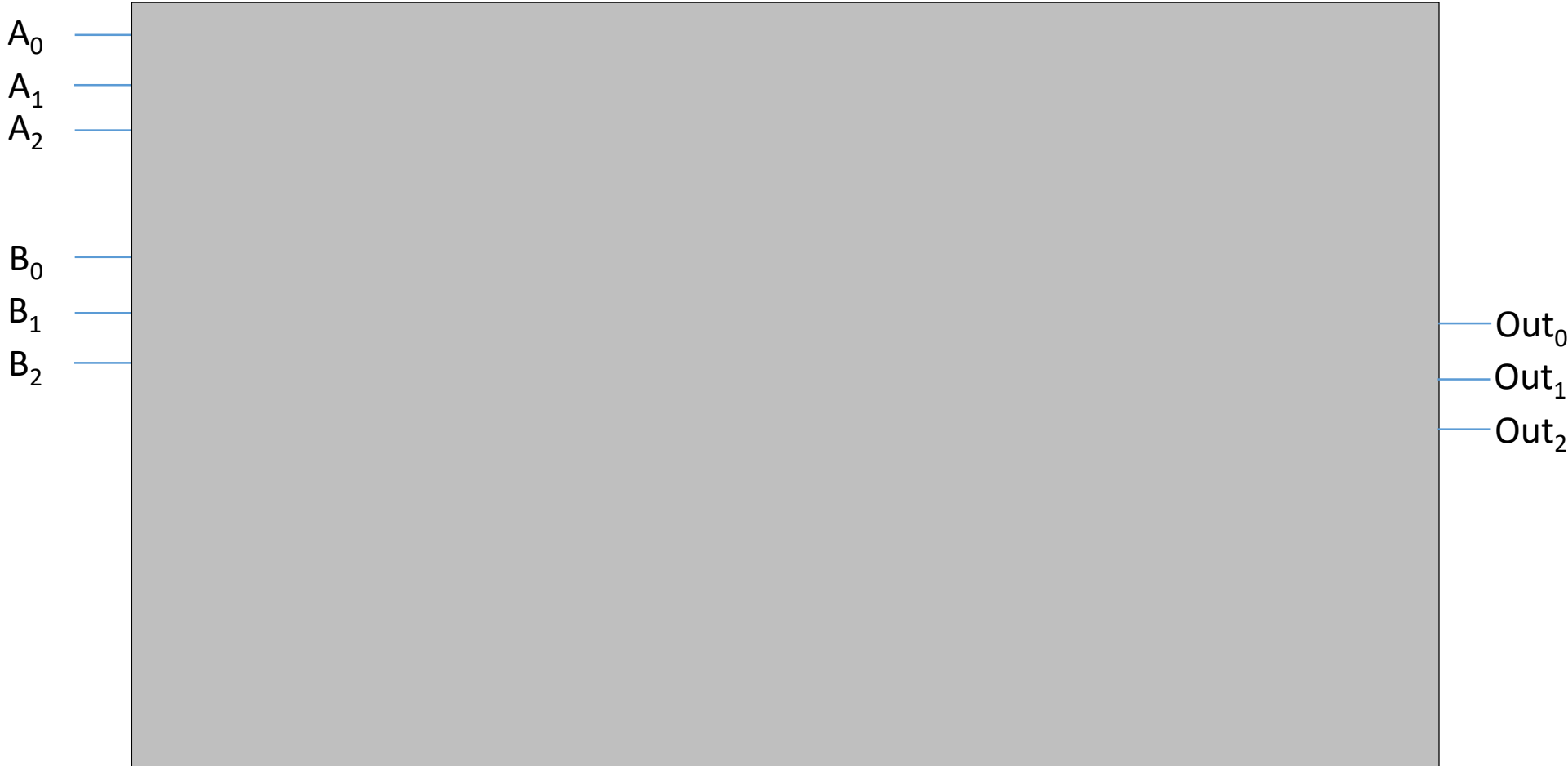
$B_1$

$B_2$

$Out_0$

$Out_1$

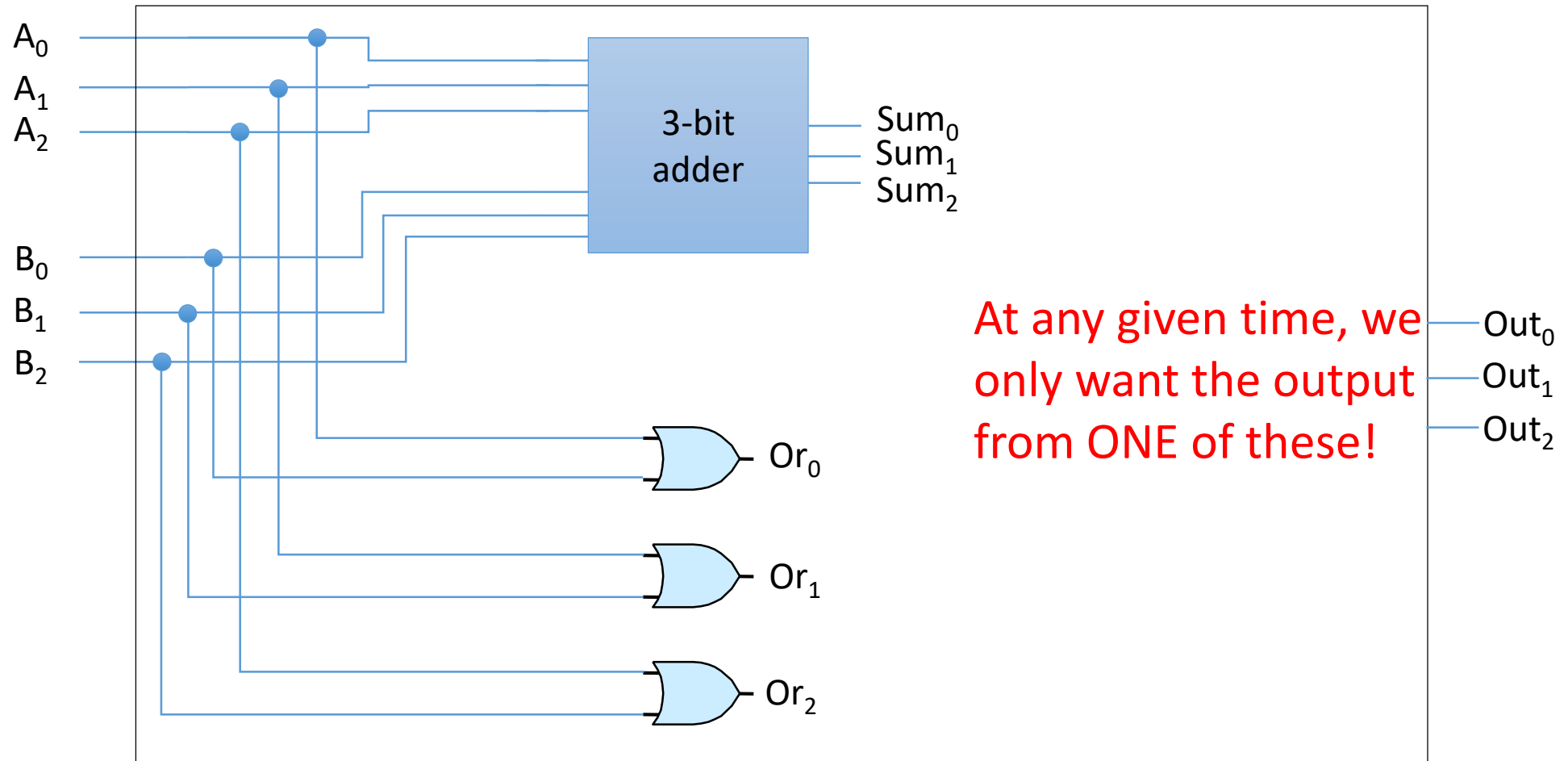
$Out_2$



# Simple 3-bit ALU: Add and bitwise OR

3-bit inputs

A and B:

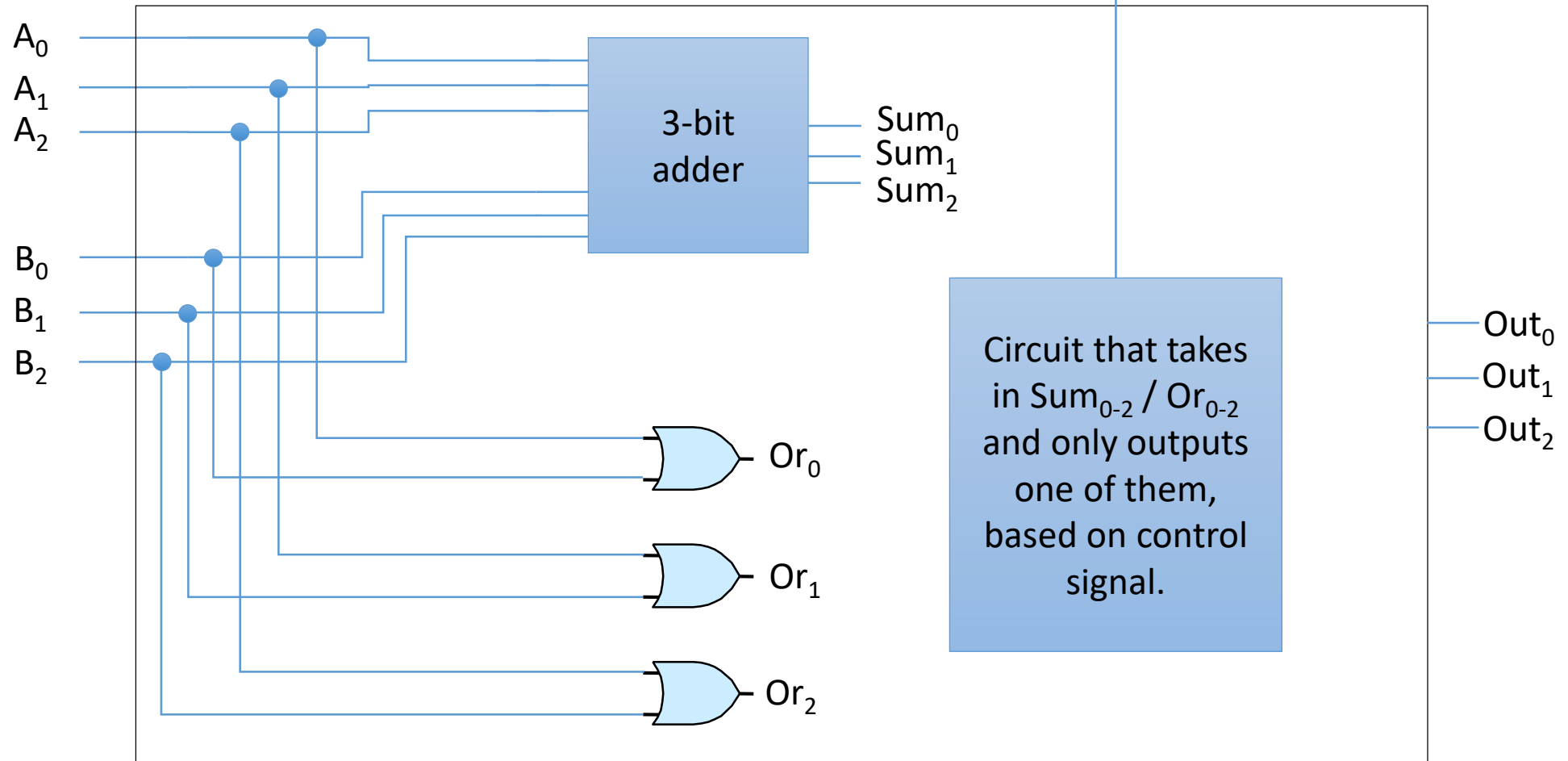


# Simple 3-bit ALU: Add and bitwise OR

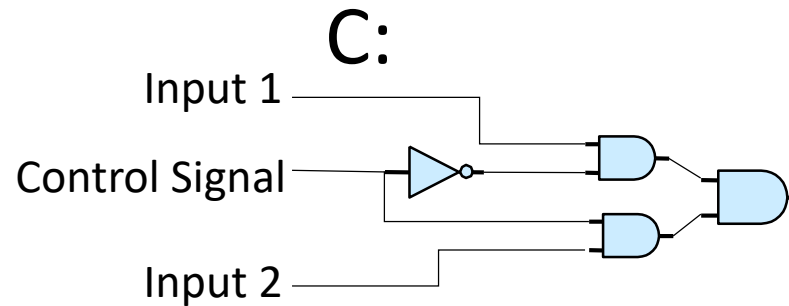
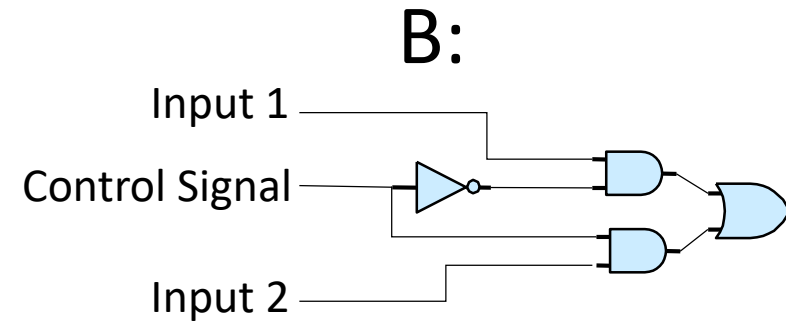
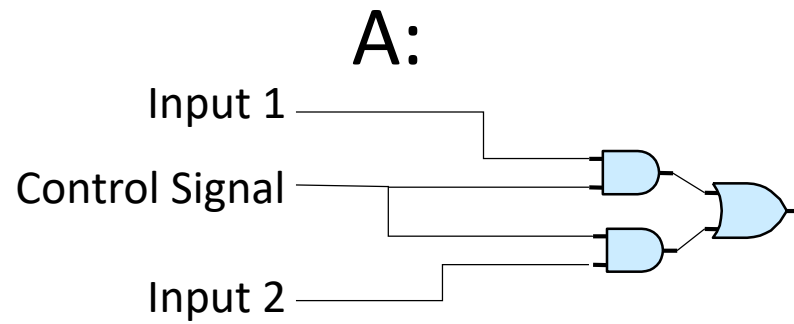
3-bit inputs

A and B:

Extra input: control signal to select Sum vs. OR



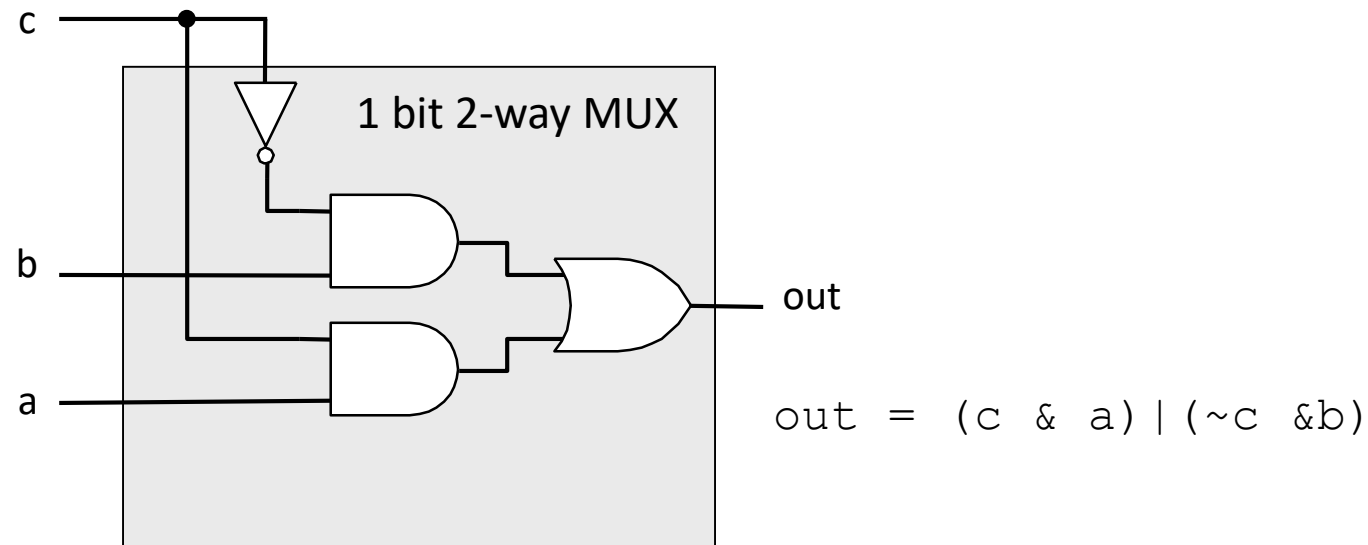
# Which of these circuits lets us select between two inputs?



# Multiplexor: Chooses an input value

Inputs:  $2^N$  data inputs,  $N$  signal bits

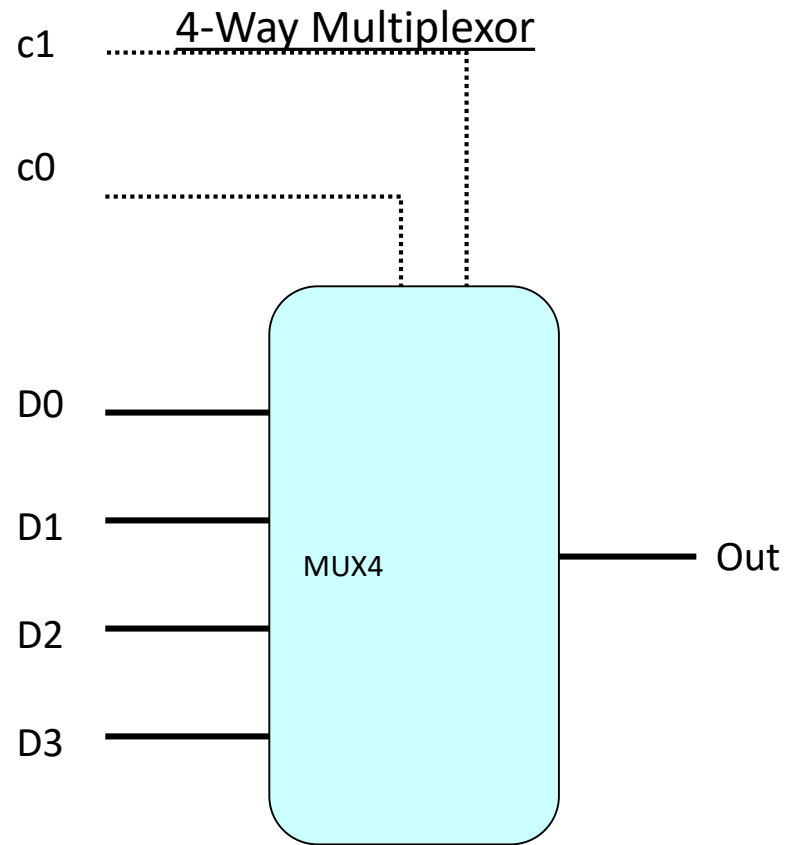
Output: is one of the  $2^N$  input values



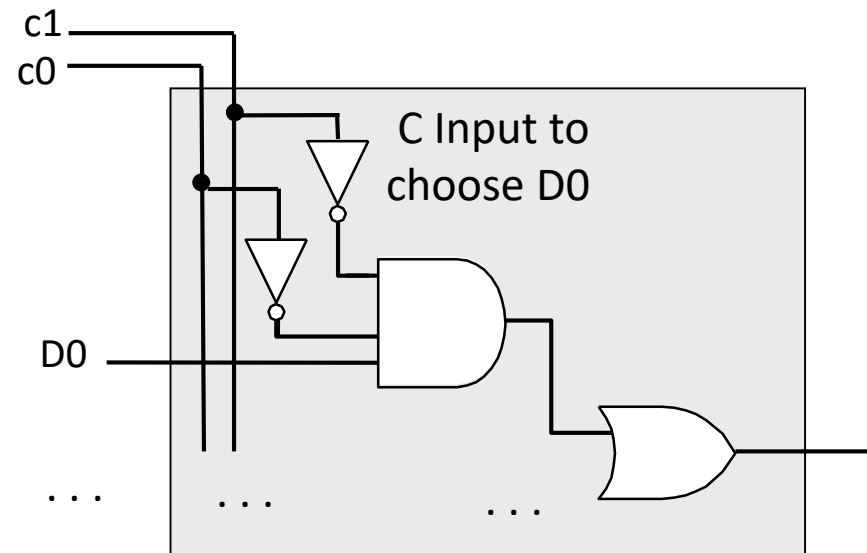
- Control signal  $c$ , chooses the input for output
  - When  $c$  is 1: choose  $a$ , when  $c$  is 0: choose  $b$

# 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



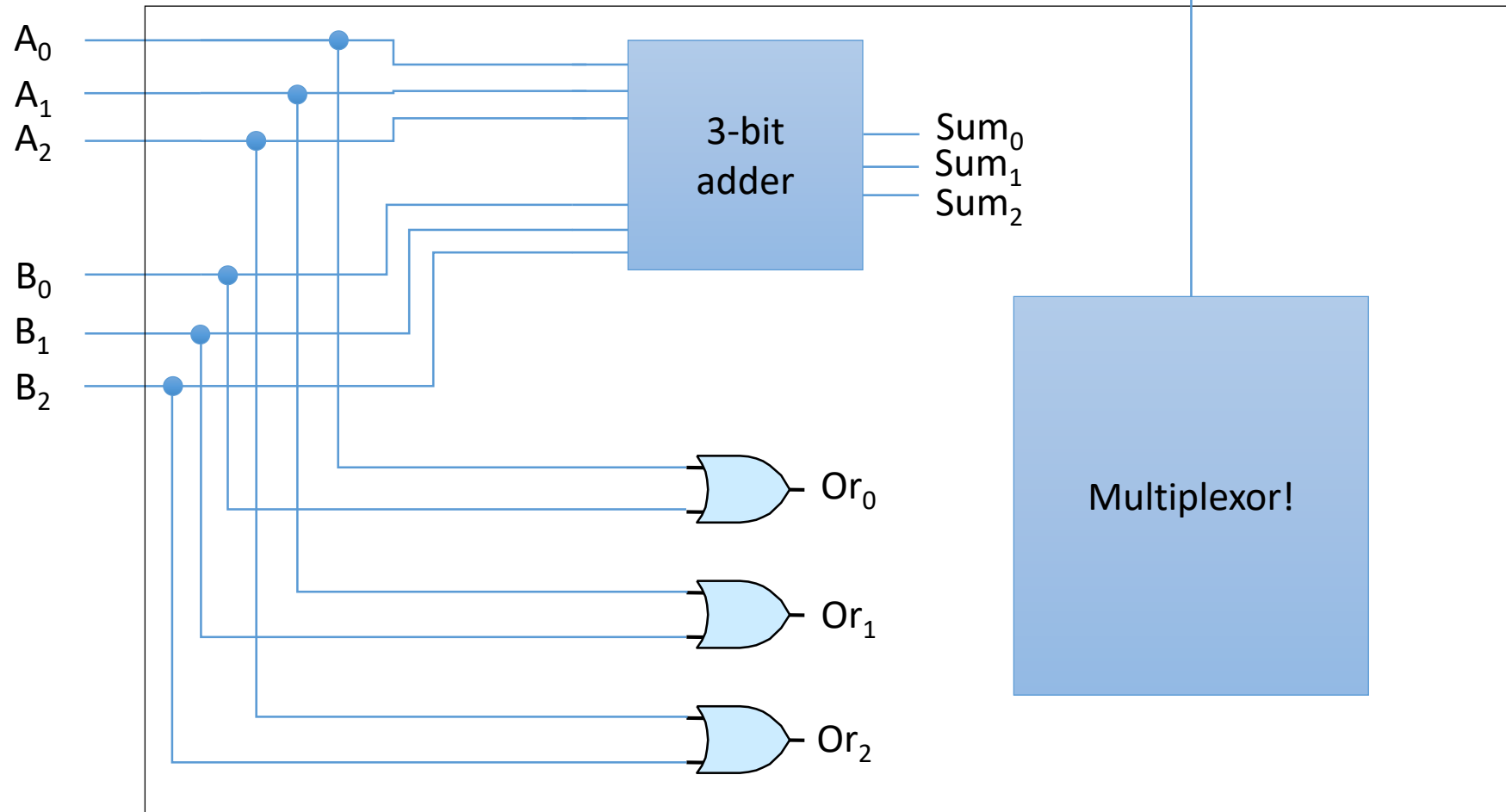


# Simple 3-bit ALU: Add and bitwise OR

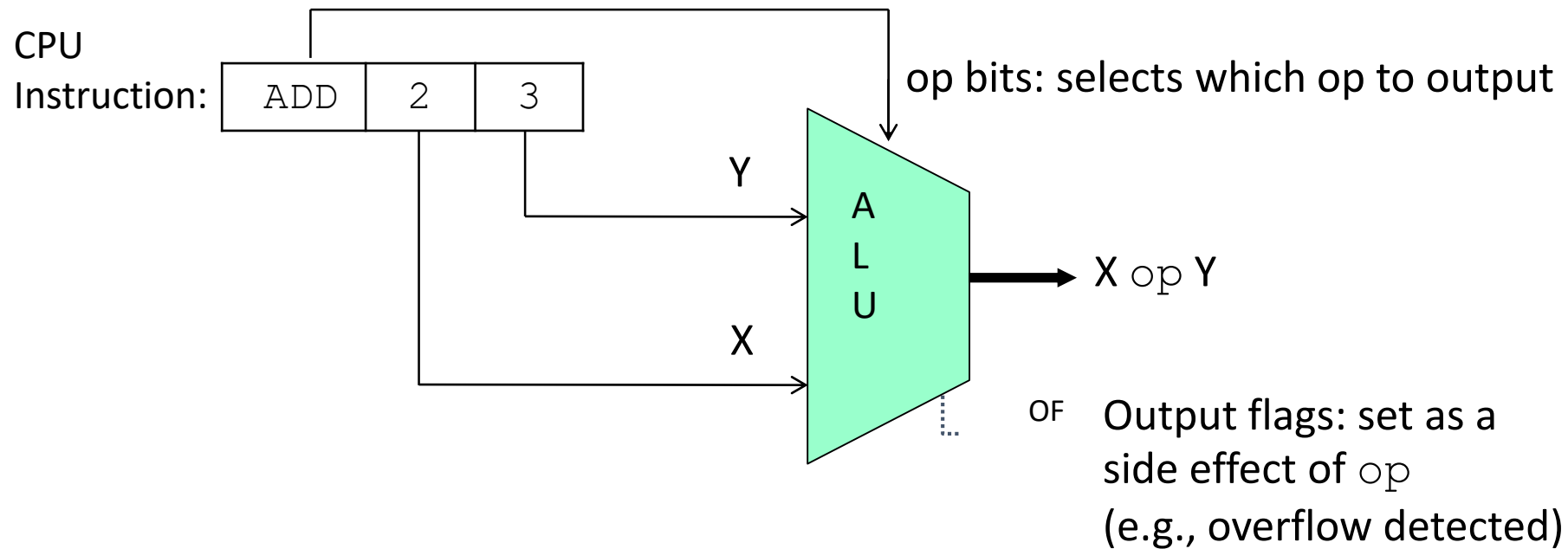
3-bit inputs

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

# 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

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

# Digital Circuits - Building a CPU

Three main classifications of HW circuits:

2. Storage: to store binary values

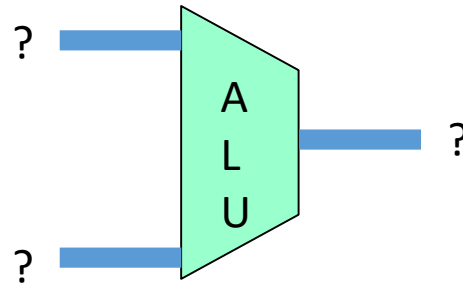
(ex) Register File: set of CPU registers

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

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

# 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?



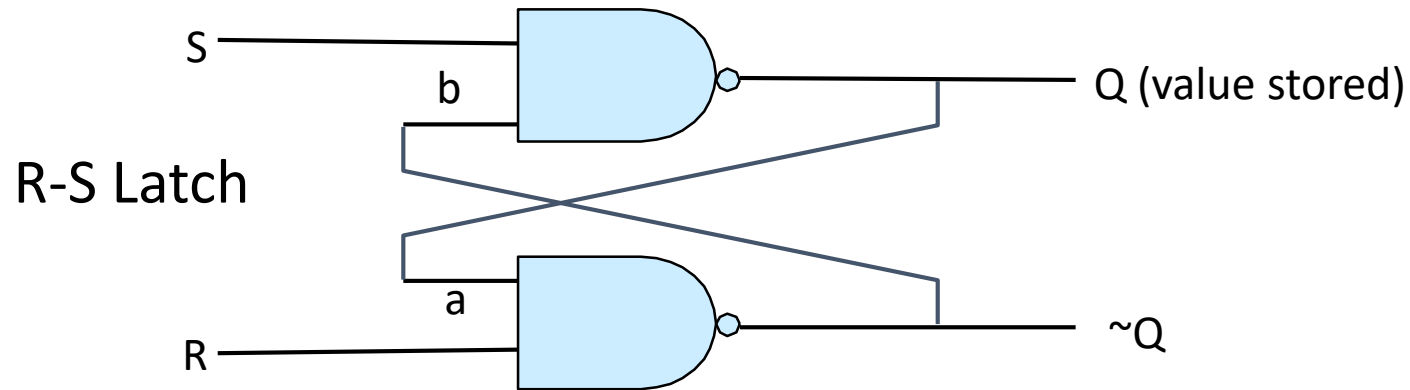
# Memory Circuit Goals: Starting Small

- 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

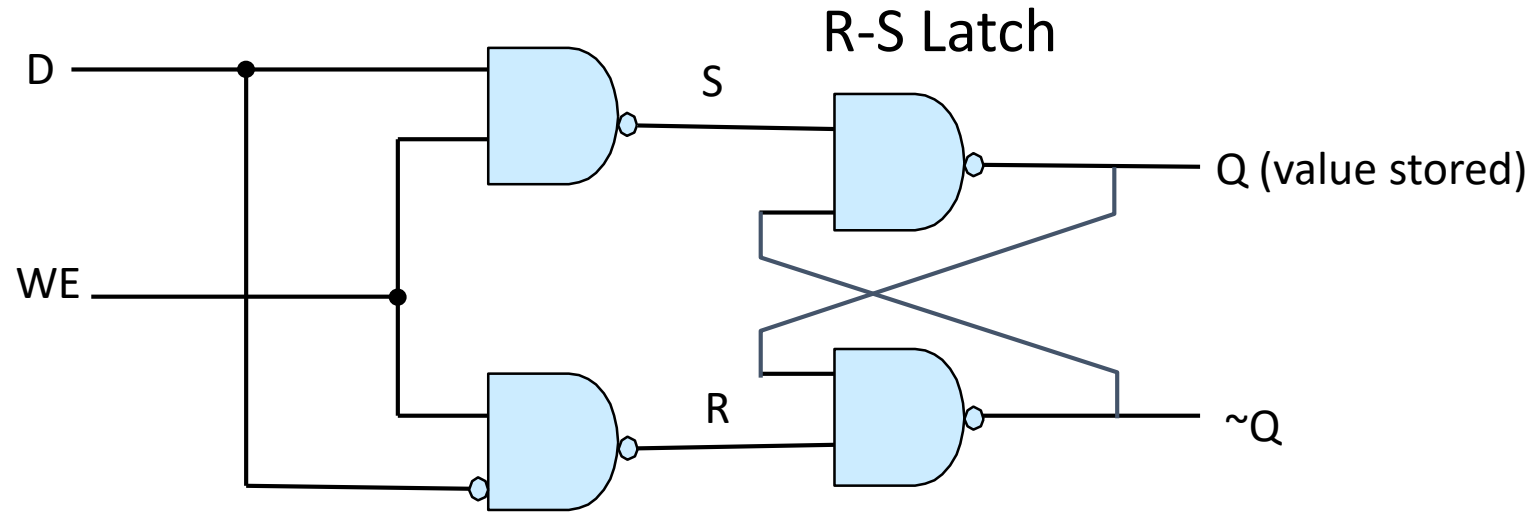
R and S are never both simultaneously 0



- 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

# Gated D Latch

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



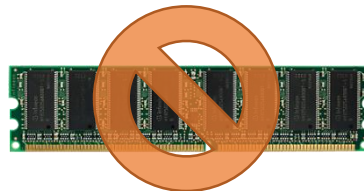
D: into top NAND,  $\sim D$  into bottom NAND

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

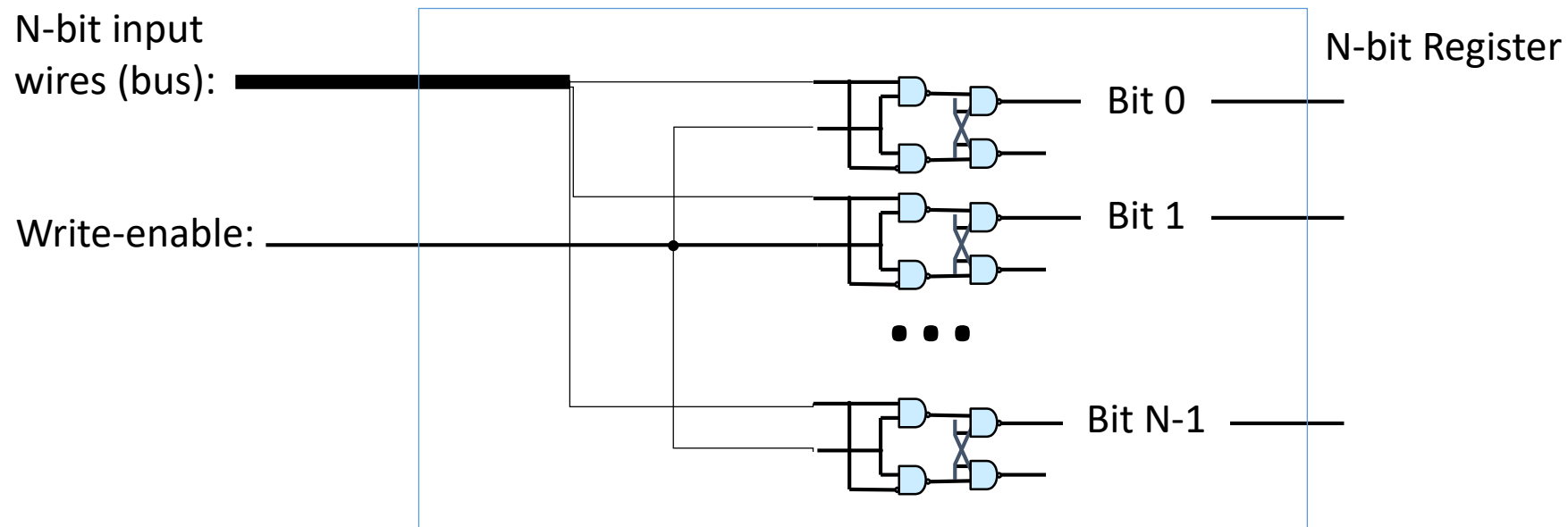
DRAM: capacitor-based:





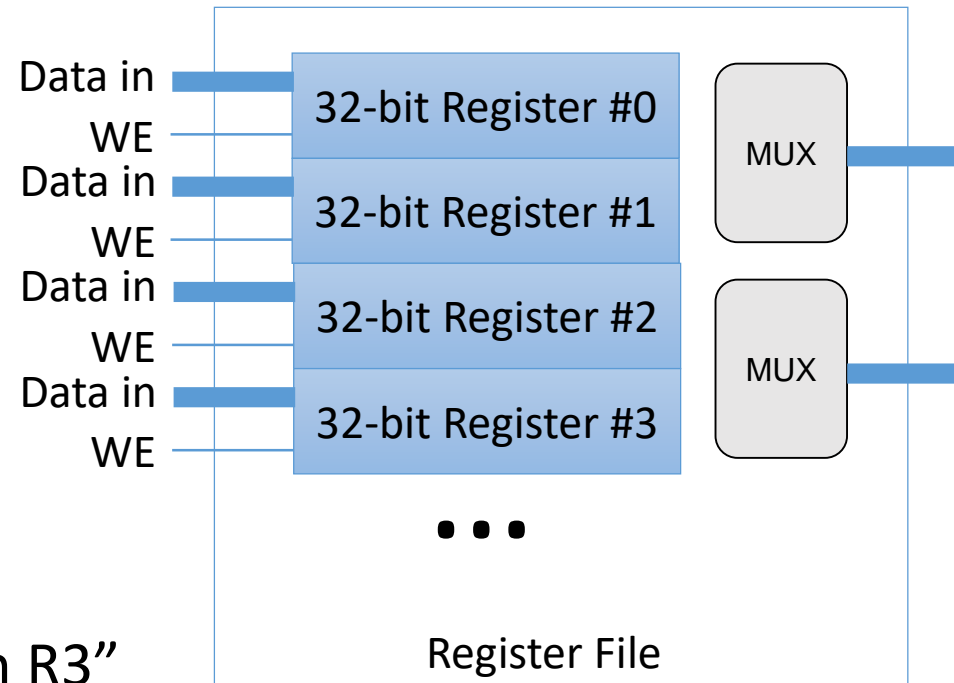
# Registers

- Fixed-size storage (8-bit, 32-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

# 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

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

# 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.

<b>HW Circuits</b>
<b>Logic Gates</b>
<b>Transistor</b>

# 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.

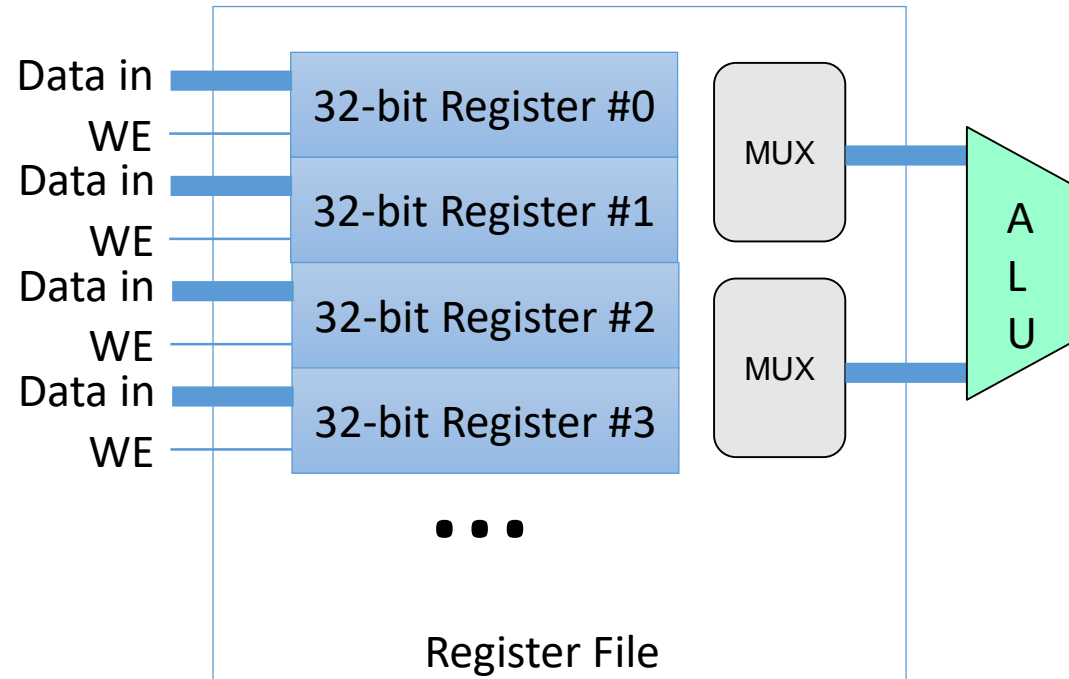
Remaining questions:

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

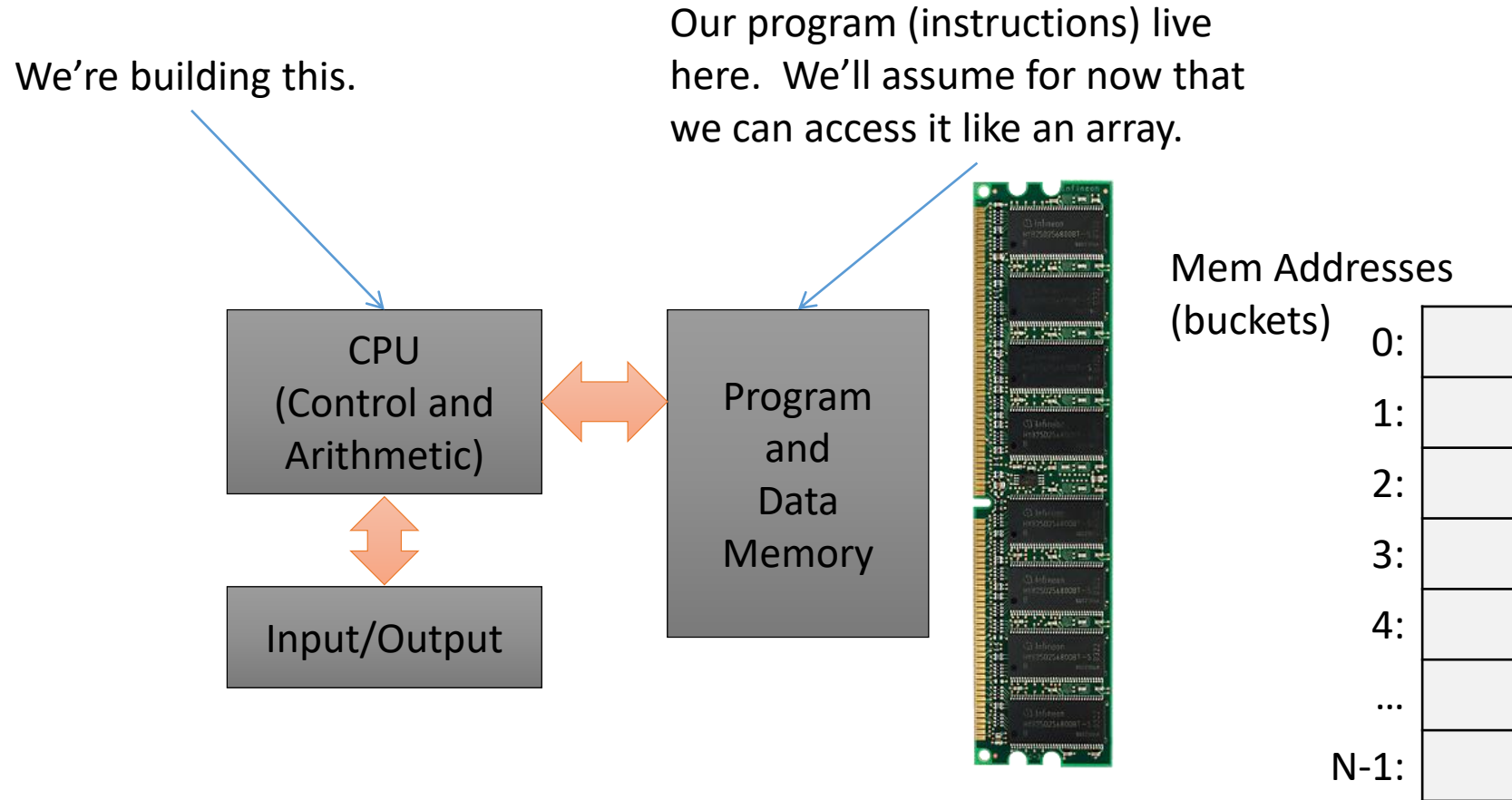
Which operation should the ALU perform?

To which register should we store the result?

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



# Recall: Von Neumann Model



# 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

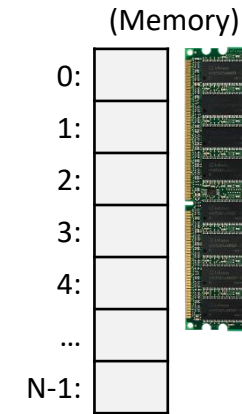
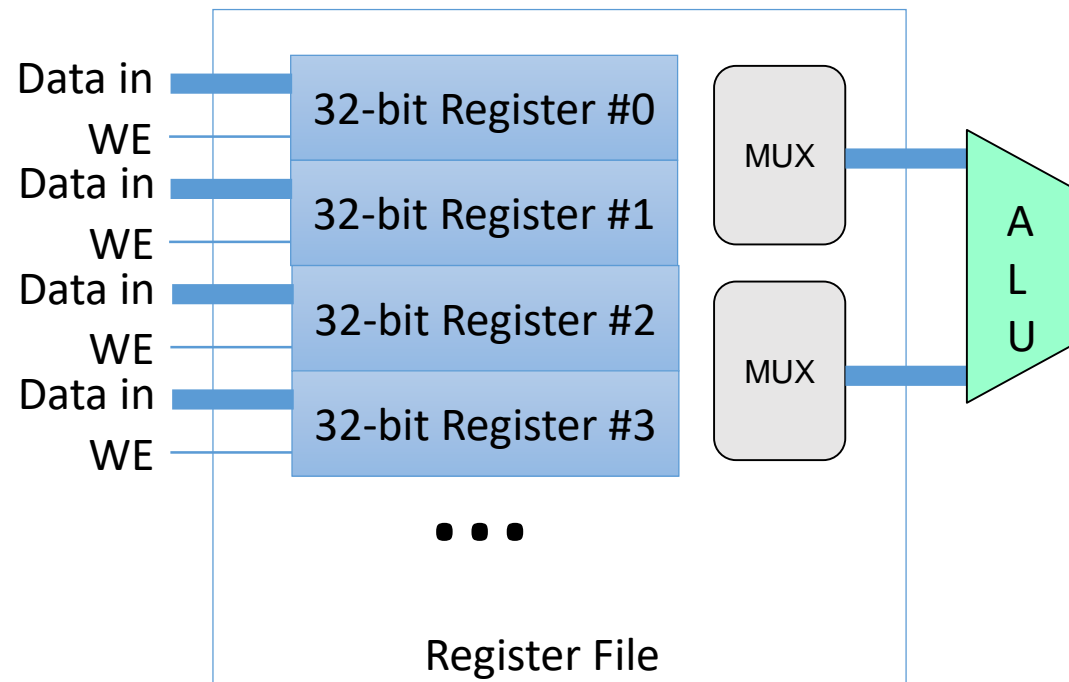


# Program State

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.

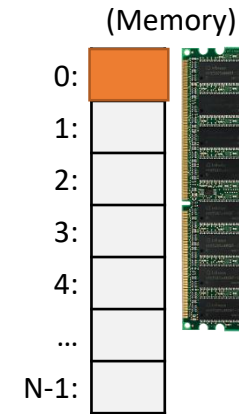
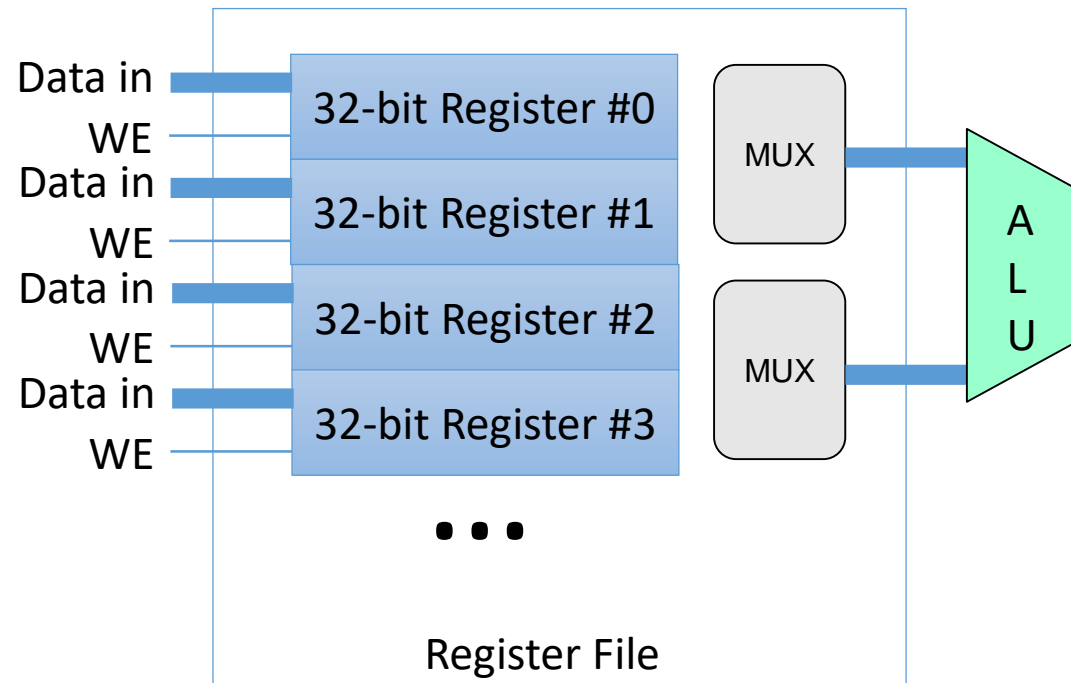
Load IR with the contents of memory at the address stored in the PC.

**Program Counter (PC):**

**Address 0**

**Instruction Register (IR):**

Instruction at Address 0



# Decoding instructions.

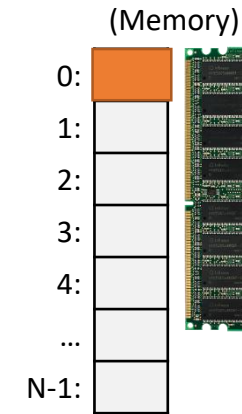
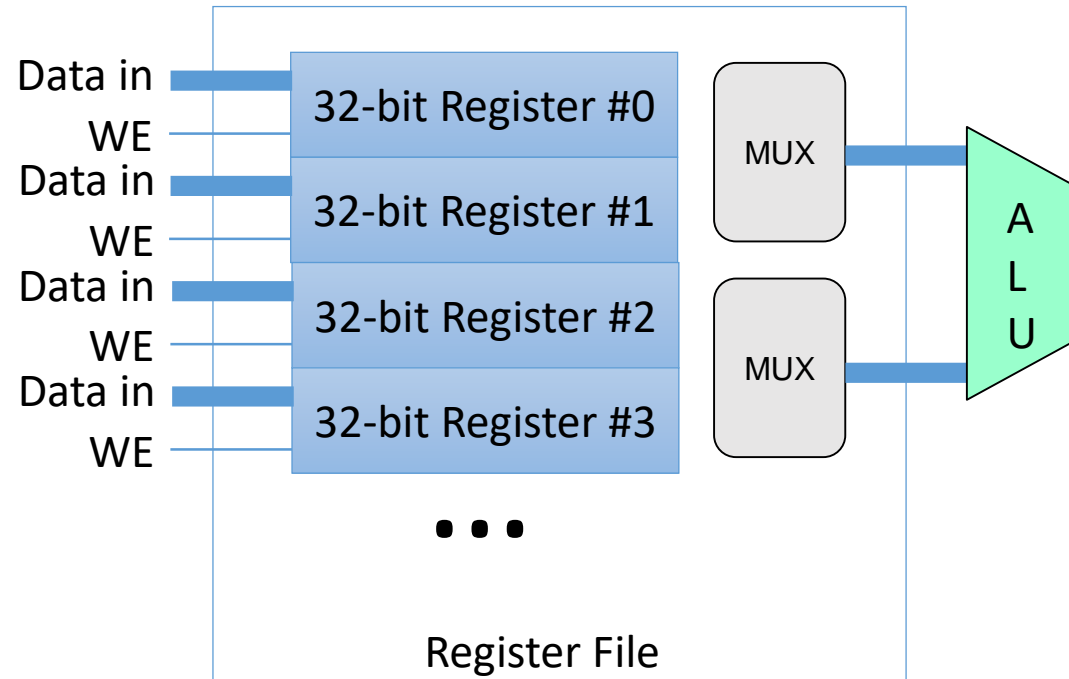
Interpret the instruction bits: What operation? Which arguments?

**Program Counter (PC):**

**Address 0**

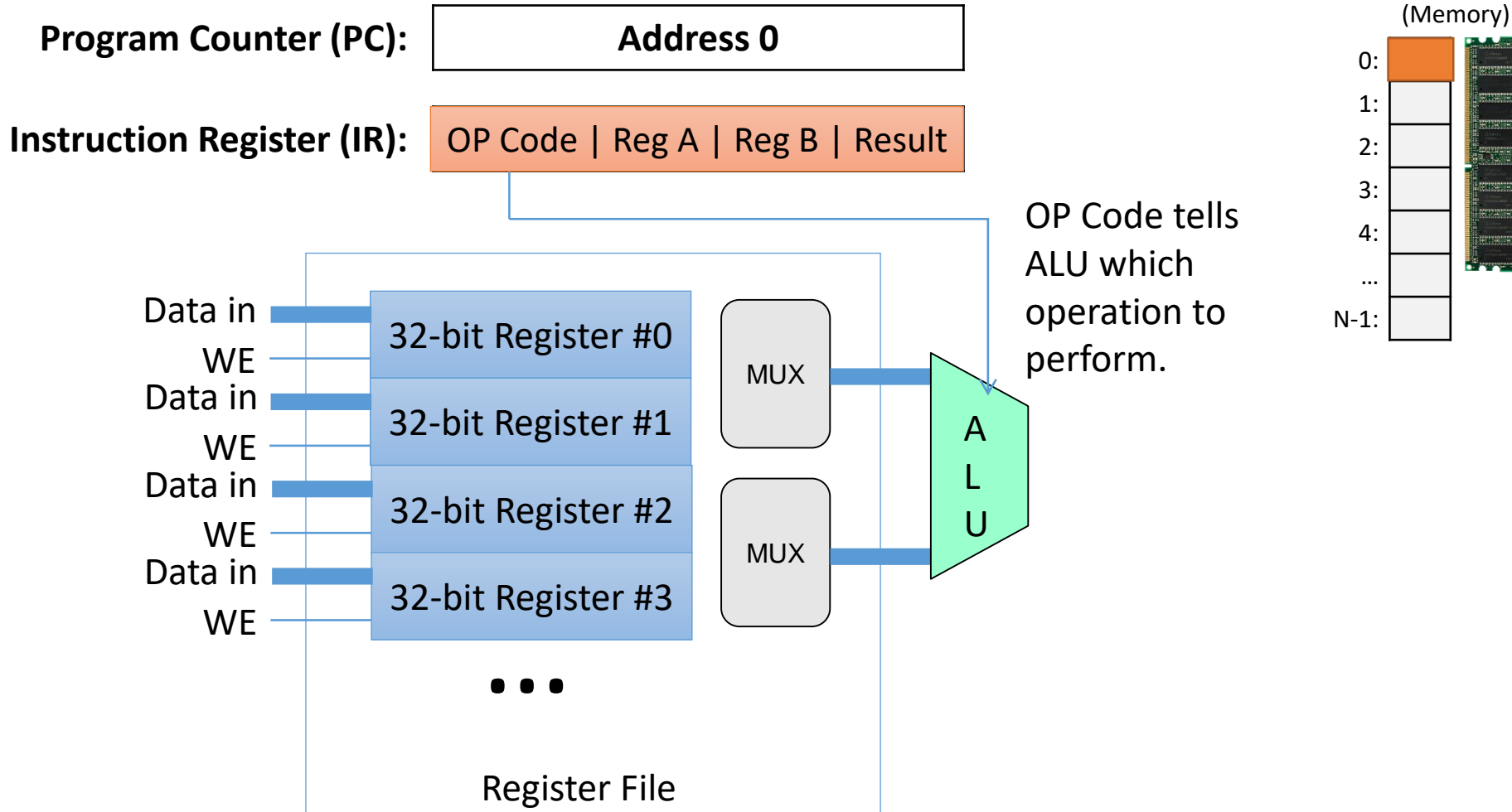
**Instruction Register (IR):**

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?

**Program Counter (PC):**

**Address 0**

**Instruction Register (IR):**

OP Code | Reg A | Reg B | Result

Data in

WE

Data in

WE

Data in

WE

Data in

WE

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3

...

Register File

MUX

MUX

A  
L  
U

Register ID #'s  
specify input  
arguments.

(Memory)

0:

1:

2:

3:

4:

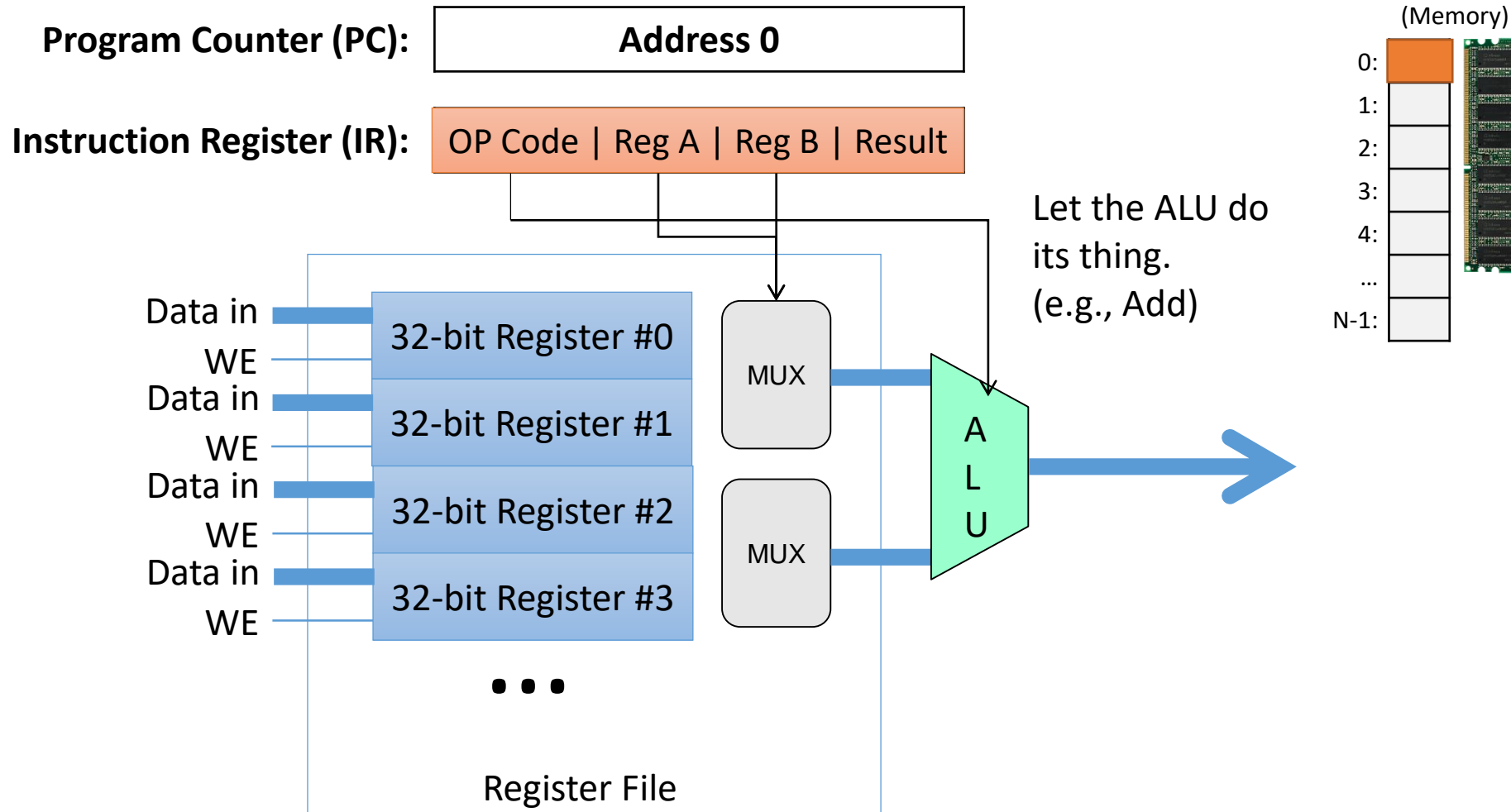
...

N-1:



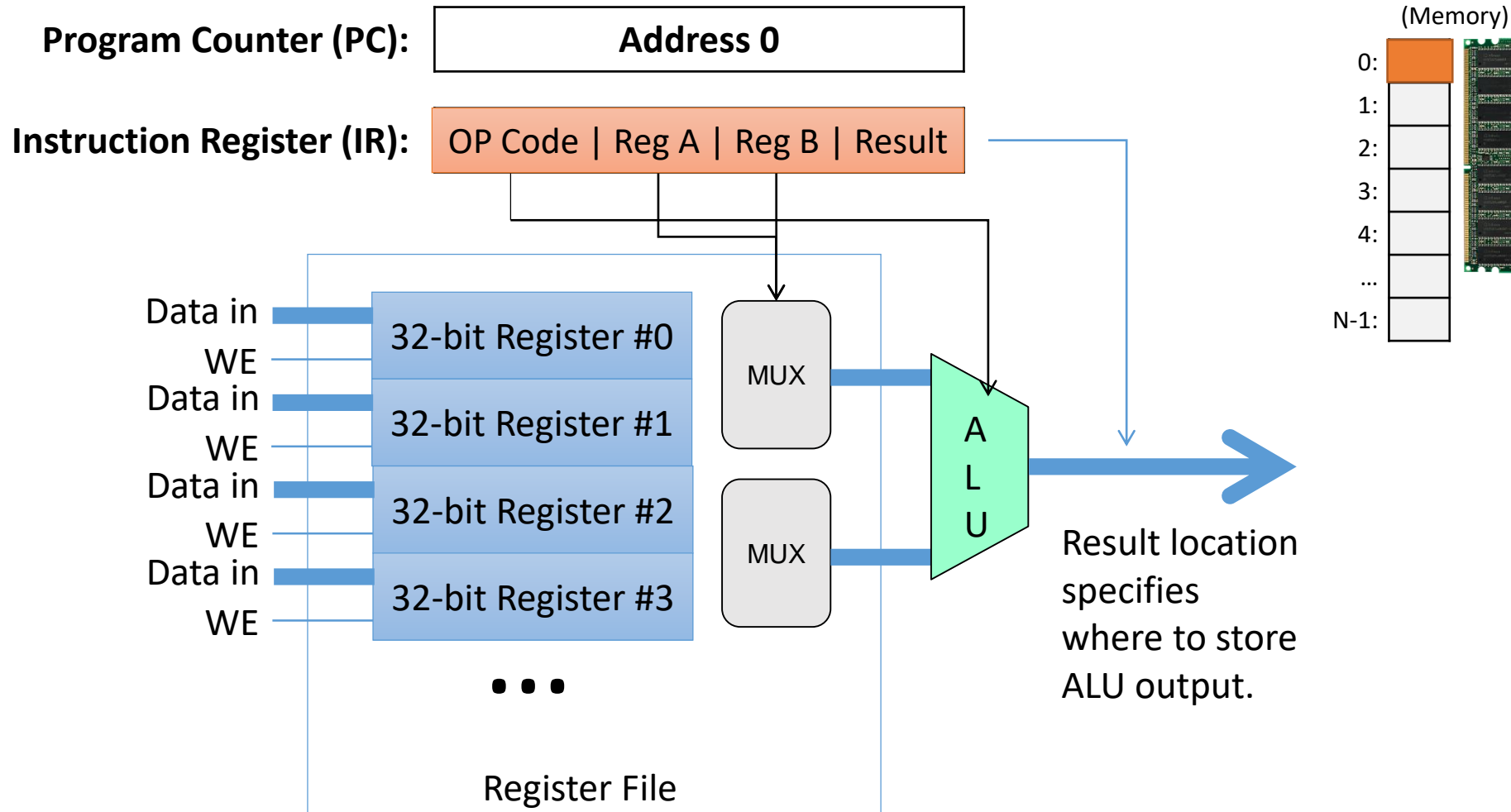
# Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



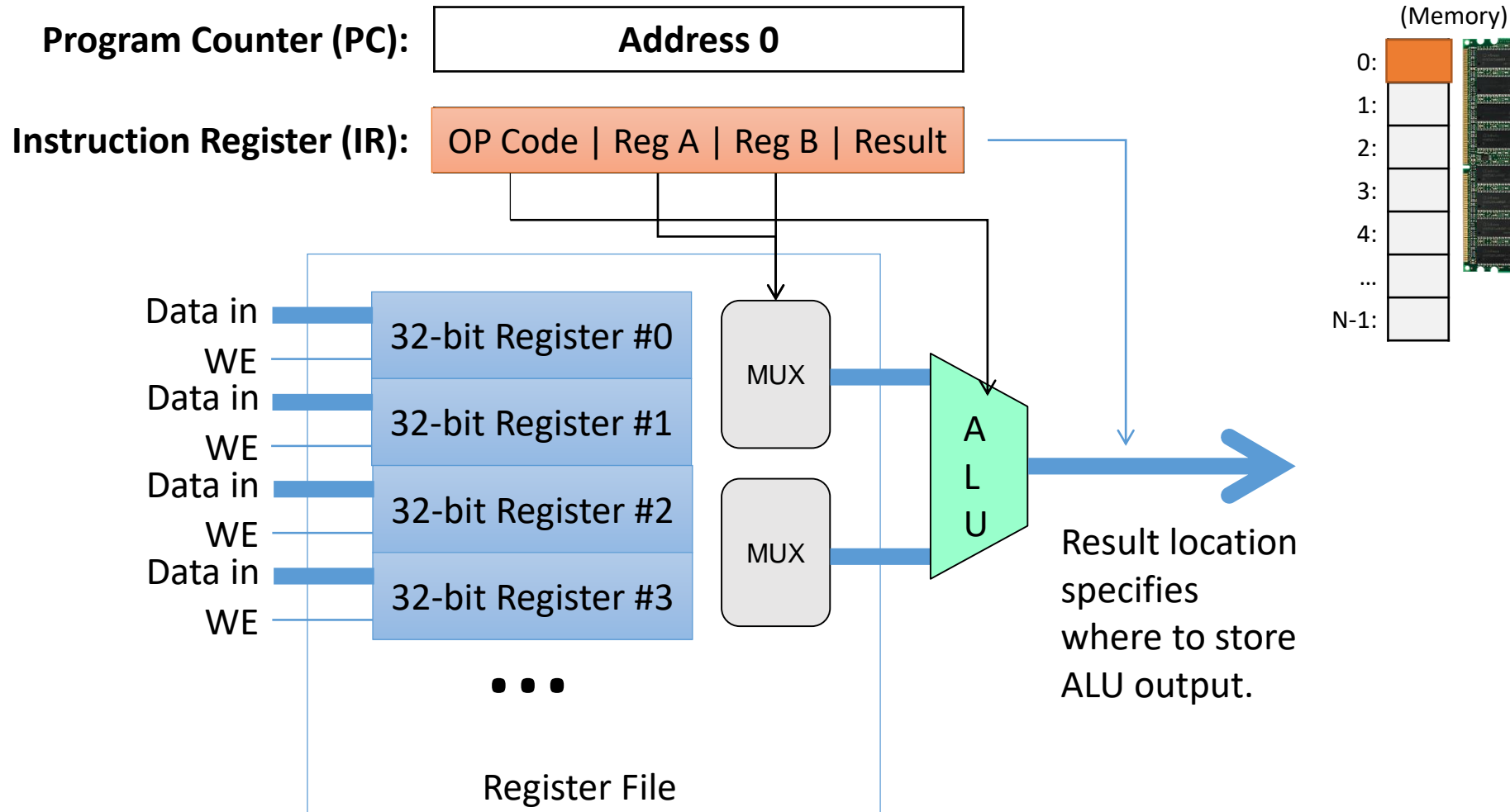
# Storing results.

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



# Questions so far?

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



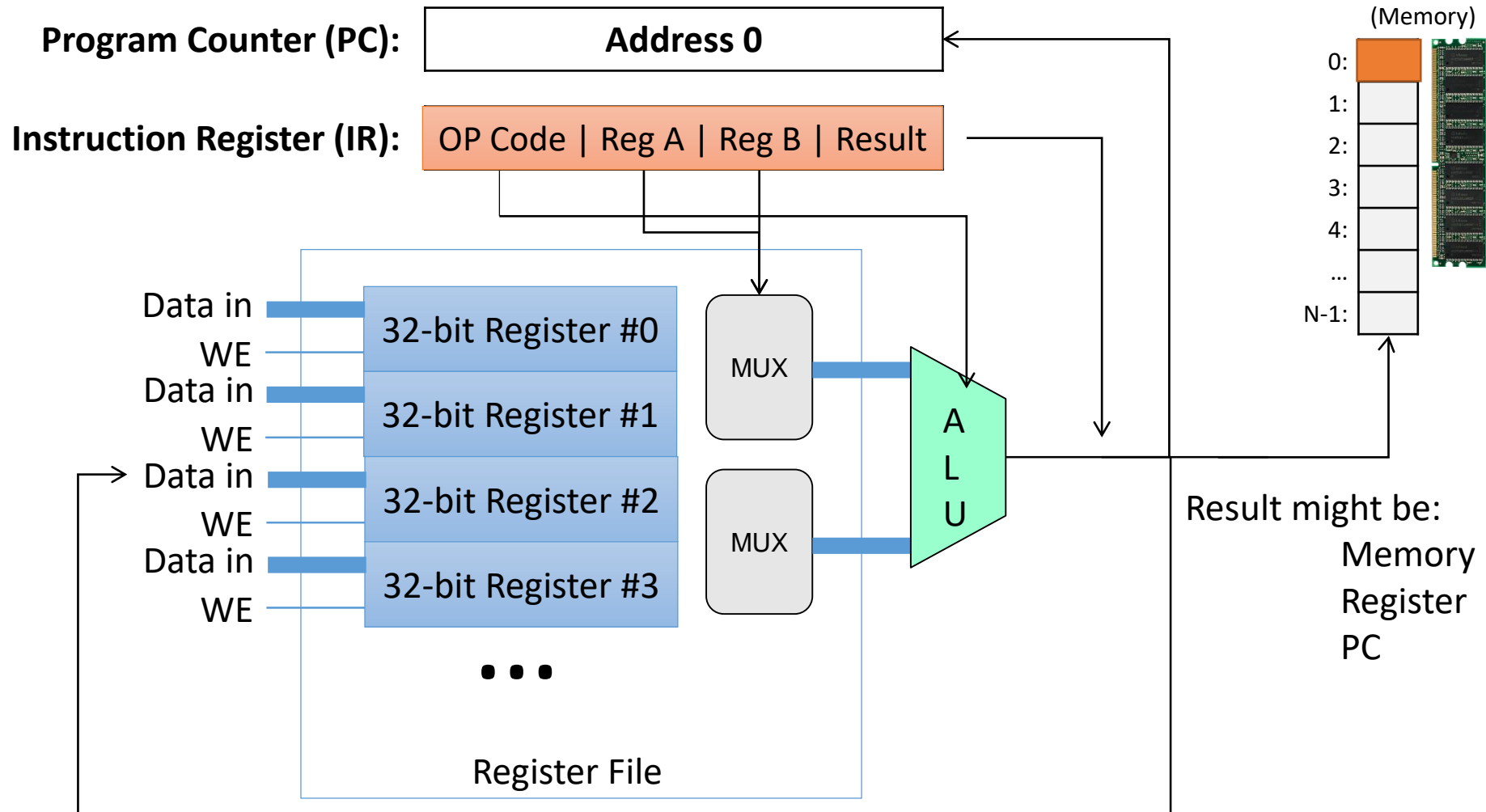


Why do we need a program counter? Can't we just start executing instruction at address 0 and count up one at a time from there?

- 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?

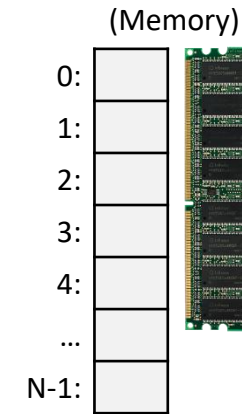
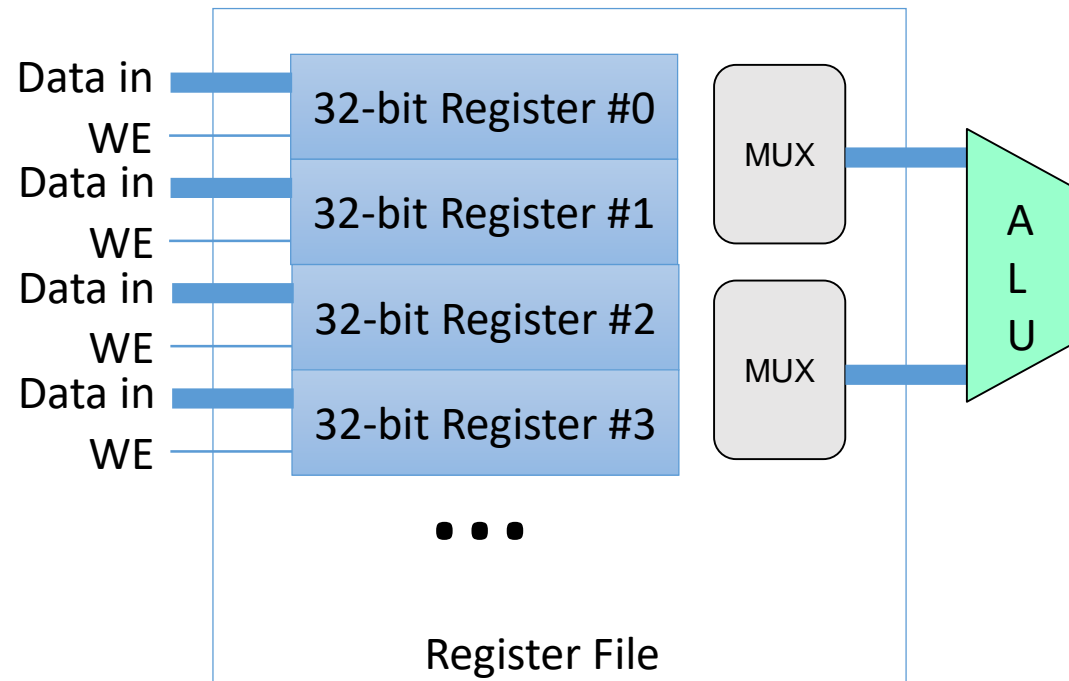


# Recap CPU Model

Four stages: fetch instruction, decode instruction, execute, store result

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

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



# Fetching instructions.

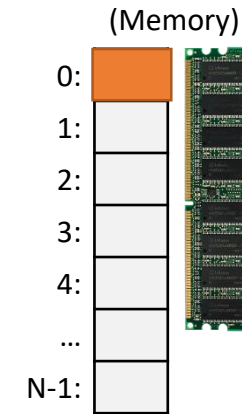
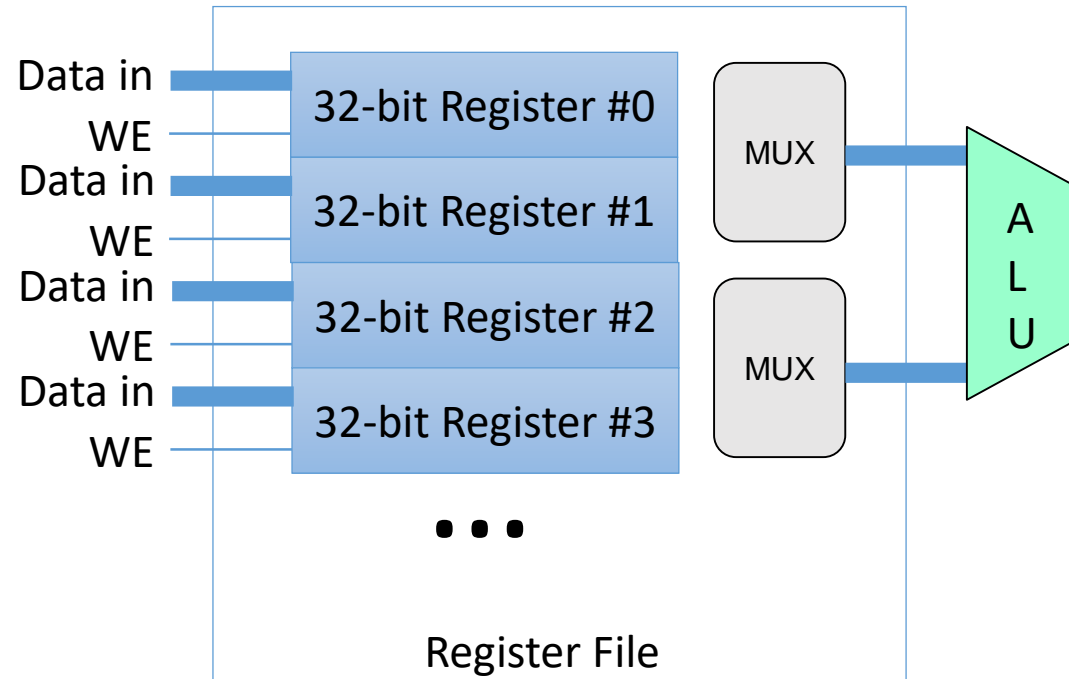
Load IR with the contents of memory at the address stored in the PC.

**Program Counter (PC):**

**Address 0**

**Instruction Register (IR):**

Instruction at Address 0



# Decoding instructions.

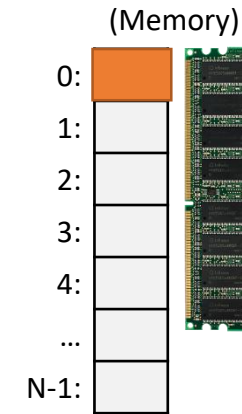
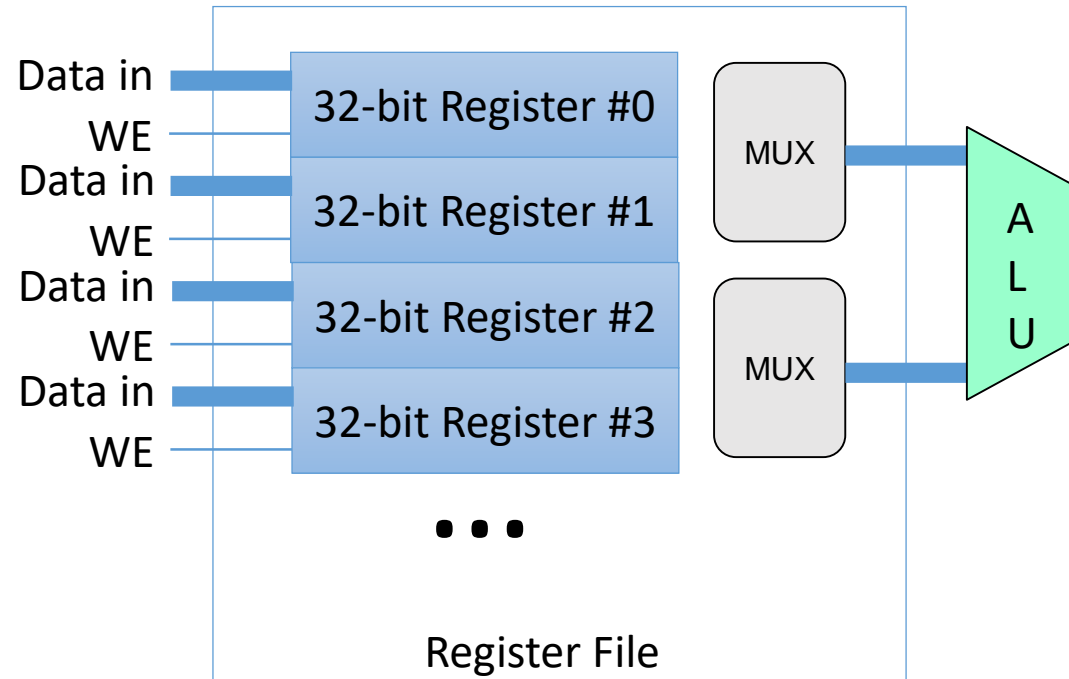
Interpret the instruction bits: What operation? Which arguments?

**Program Counter (PC):**

**Address 0**

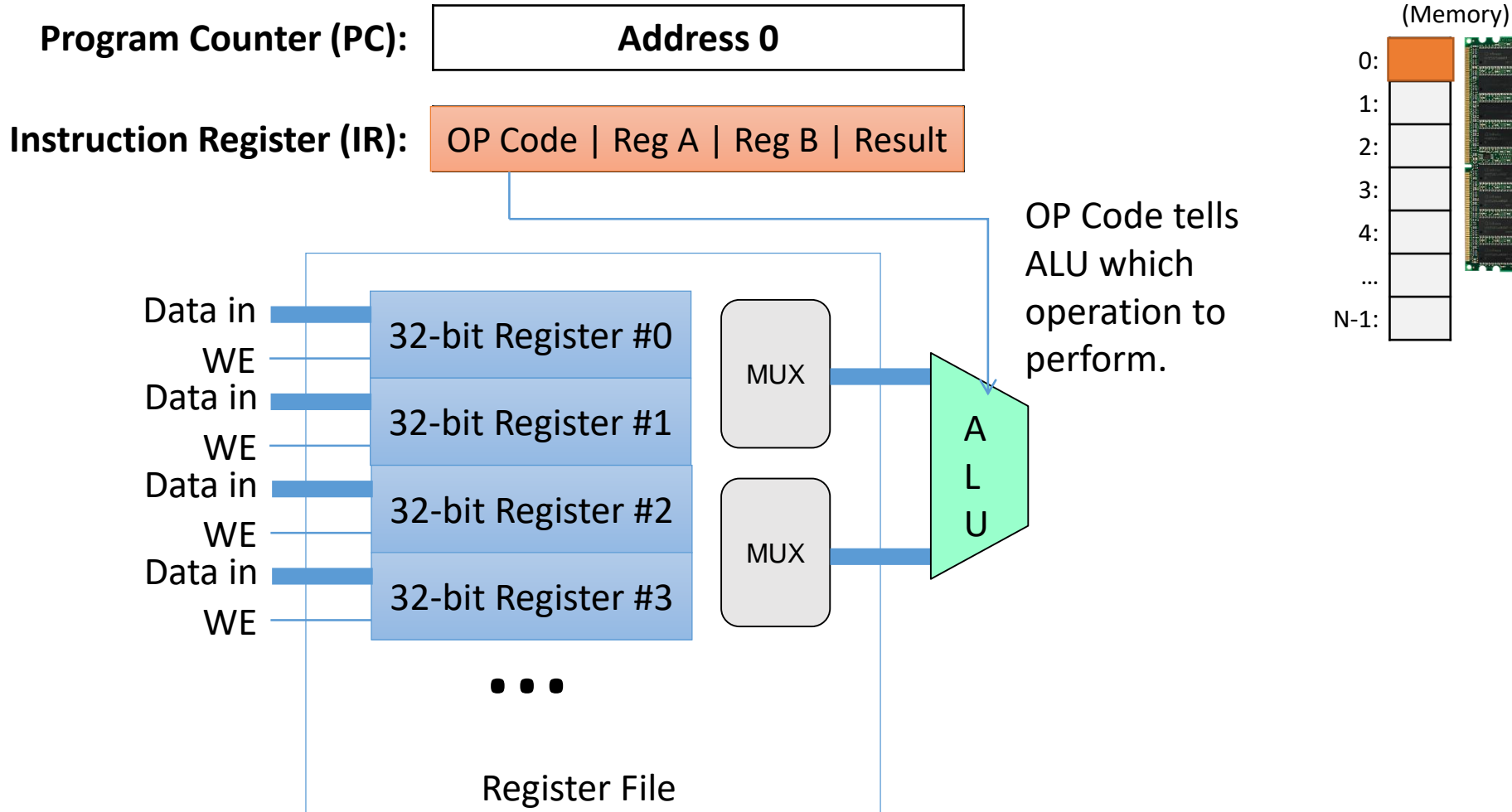
**Instruction Register (IR):**

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?

**Program Counter (PC):**

**Address 0**

**Instruction Register (IR):**

OP Code | Reg A | Reg B | Result

Data in

WE

Data in

WE

Data in

WE

Data in

WE

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3

...

Register File

MUX

MUX

A  
L  
U

Register ID #'s  
specify input  
arguments.

(Memory)

0:

1:

2:

3:

4:

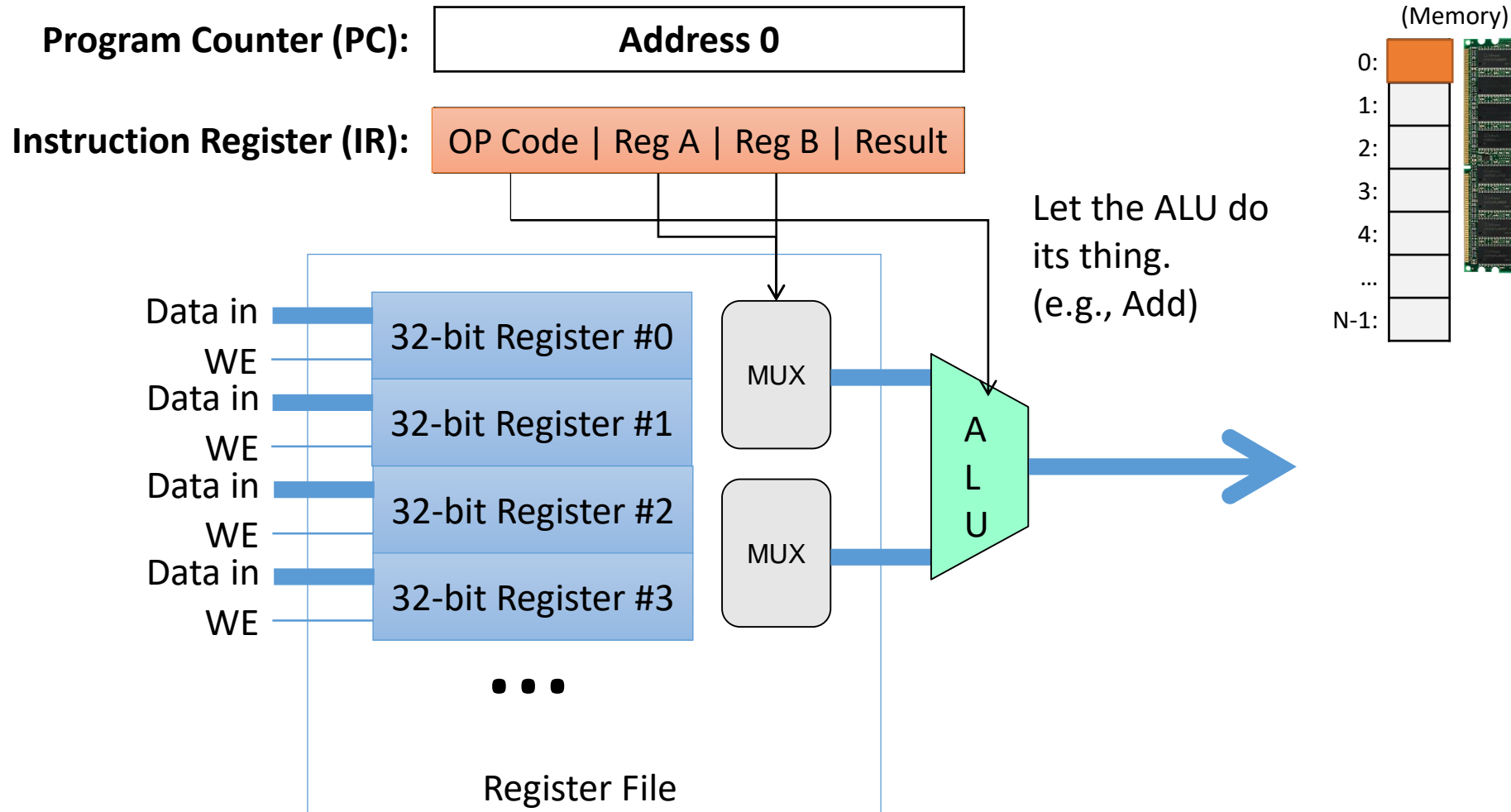
...

N-1:



# Executing instructions.

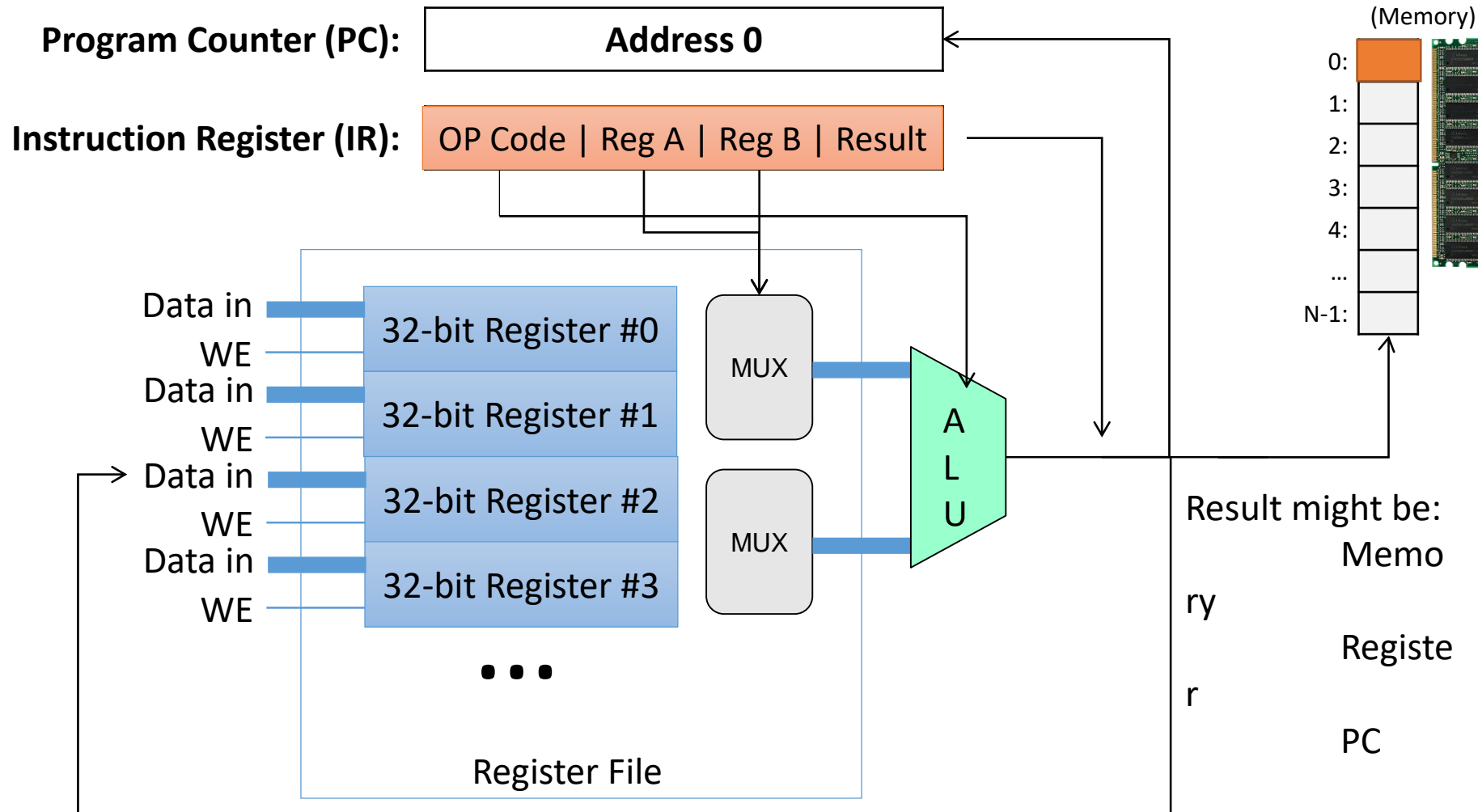
Interpret the instruction bits: What operation? Which arguments?





# Storing results.

Interpret the instruction bits: Store result in register, memory, PC.

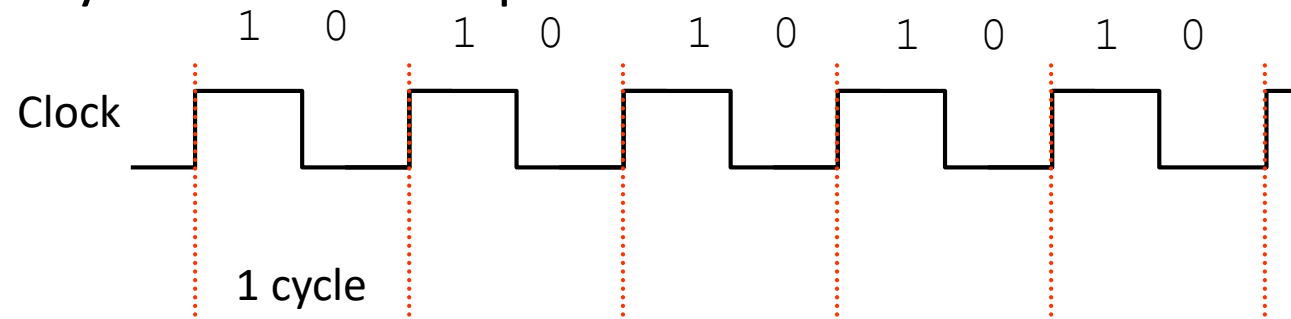


# Clocking

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

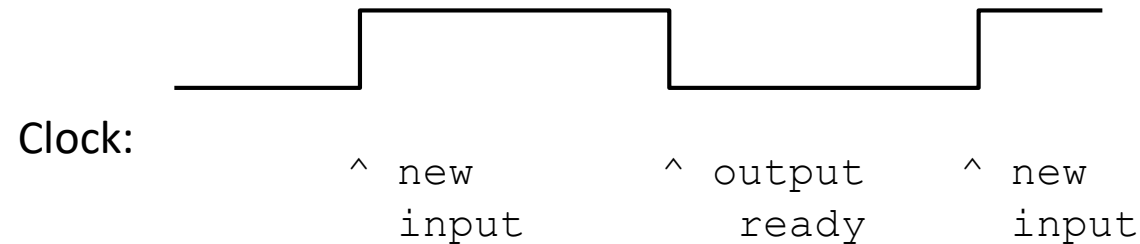


- Clock determines how fast system runs
  - Processor can only do one thing per clock cycle
    - Usually just one part of executing an instruction
  - 1GHz processor:  
1 billion cycles/second → 1 cycle every nanosecond

# Clock and Circuits

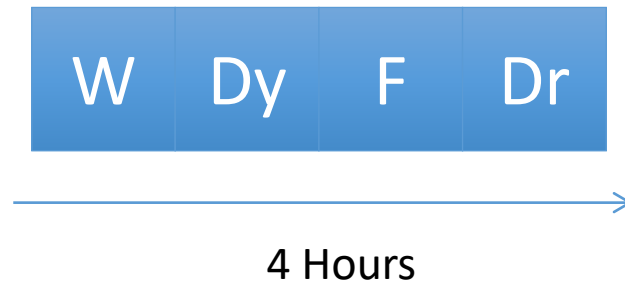
## Clock Edges Triggers events

- Circuits have continuous values
- Rising Edge: trigger new input values
- Falling Edge: consistent output ready to read
- Between rising and falling edge can have inconsistent state as new input values flow through circuit



# 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 Hours



4 Hours



4 Hours

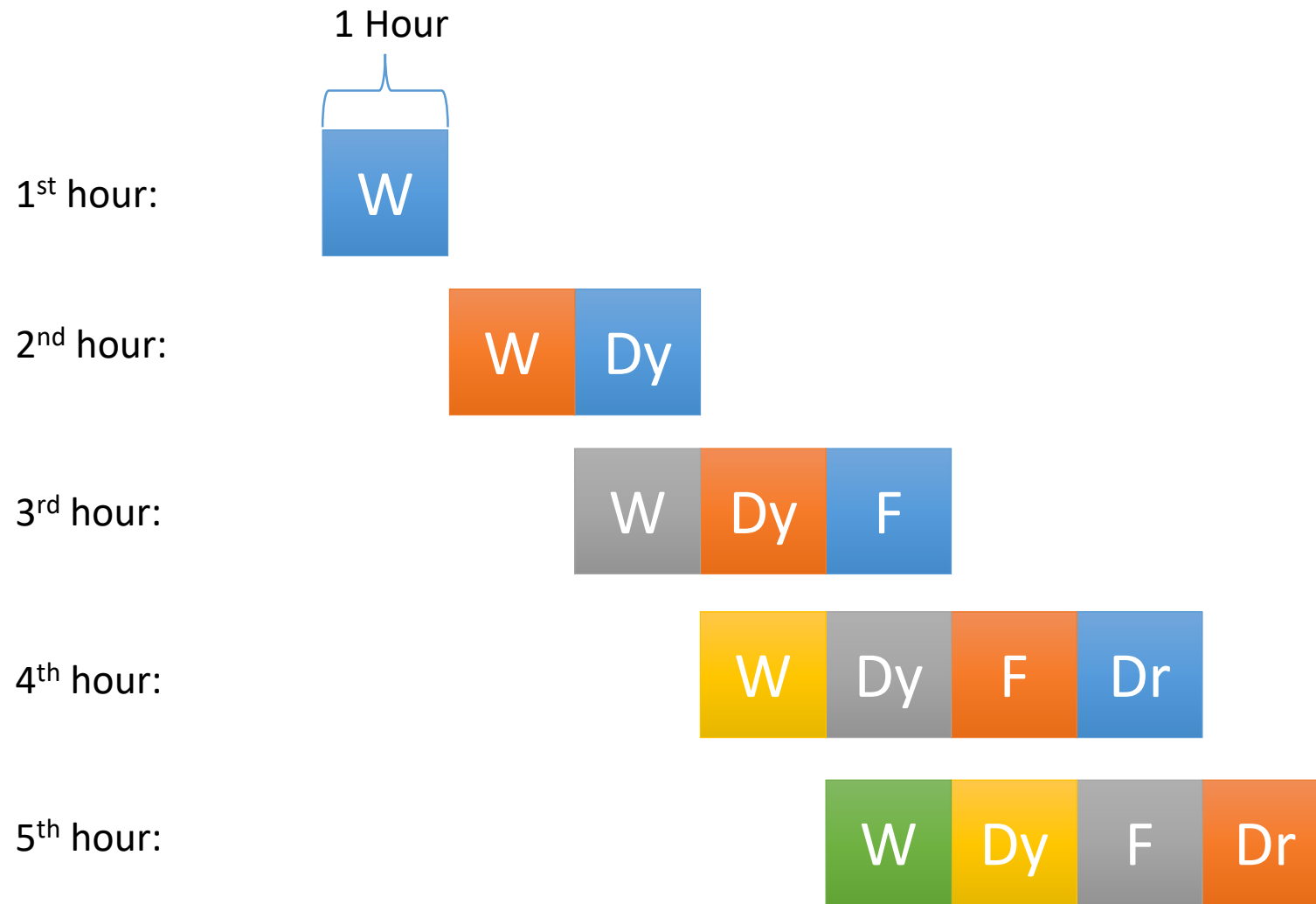
4-hour cycle time.

Finishes a laundry load every cycle.

(6 laundry loads per day)



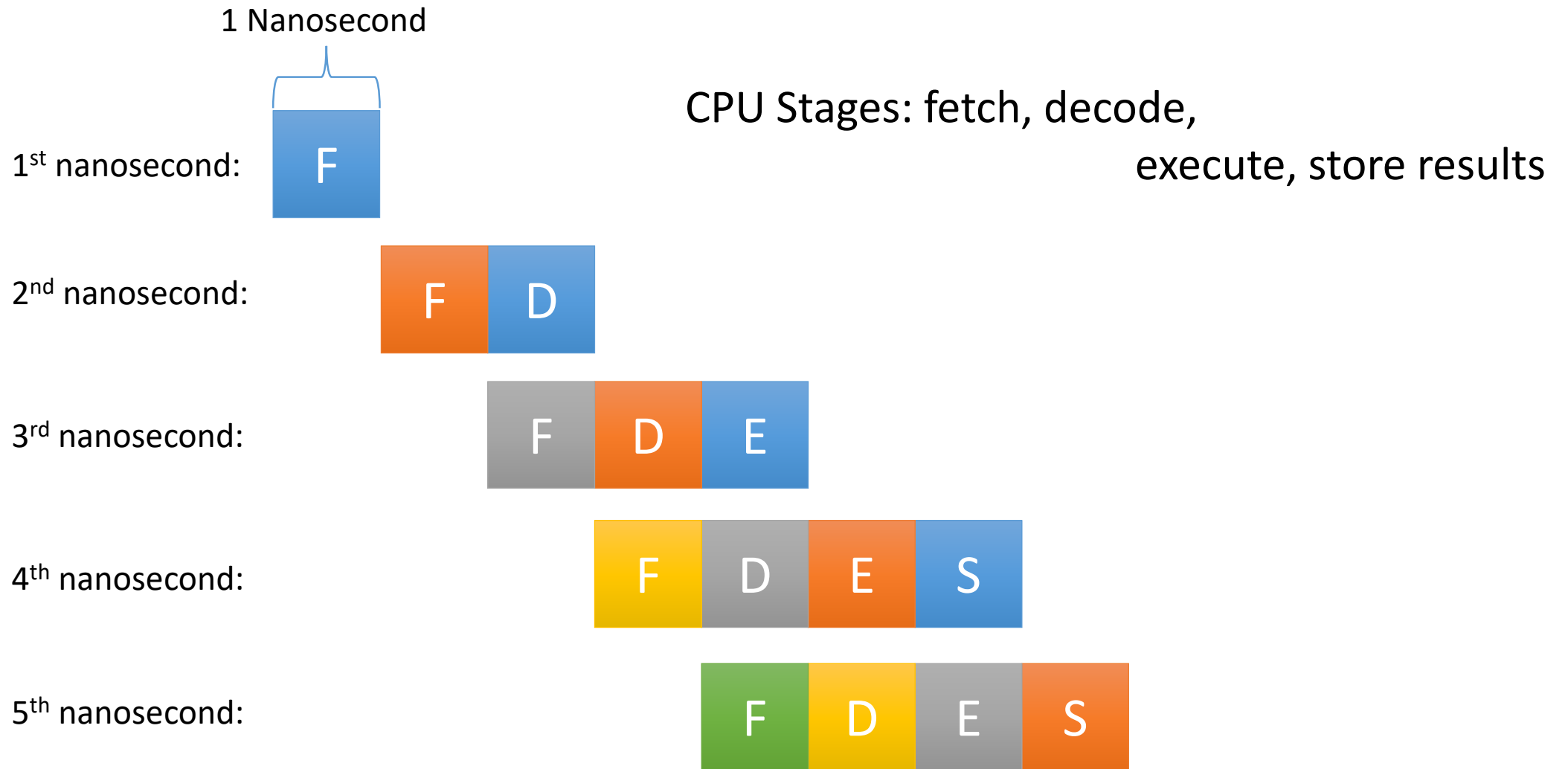
# Pipelining (Laundry)



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.)

# Up next

- Talking to the CPU: Assembly language