CS 31: Intro to Systems Digital Logic

Vasanta Chaganti & Kevin Webb

Swarthmore College

September 19, 2023

Reading Quiz

Today

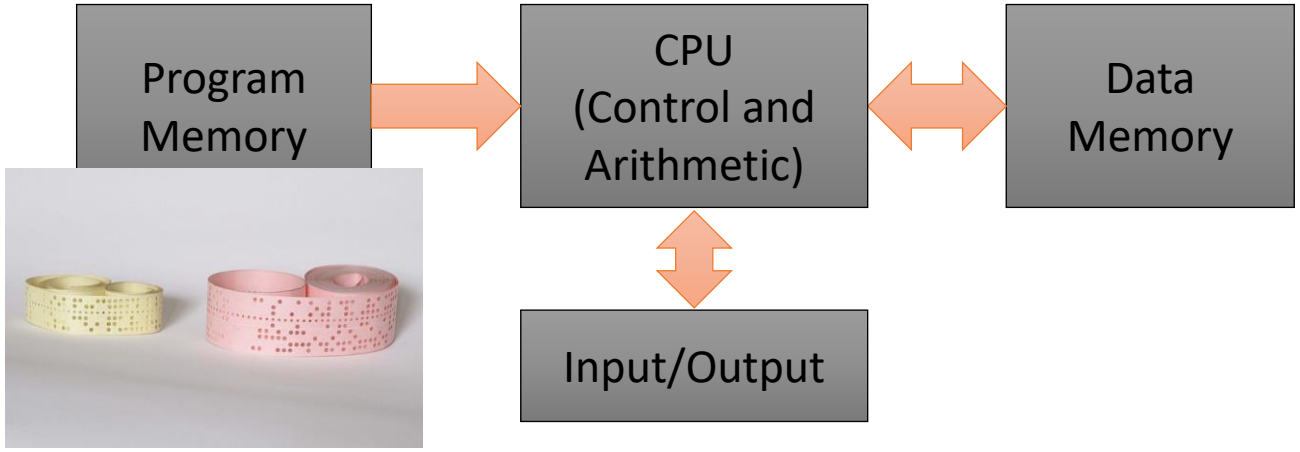
- Hardware basics
  - Machine memory models
  - Digital signals
  - Logic gates

Drawing Circuits: Borrow some paper if you need to!

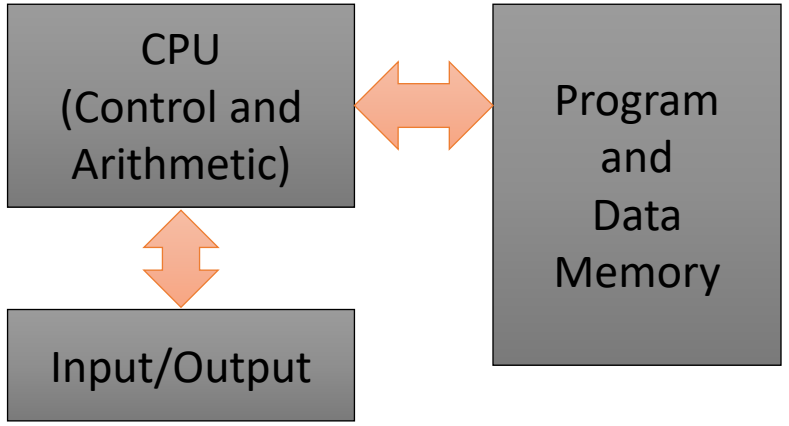
- Manipulating/Representing values in hardware
  - Adders
  - Storage & memory (latches)

Hardware Models (1940's)

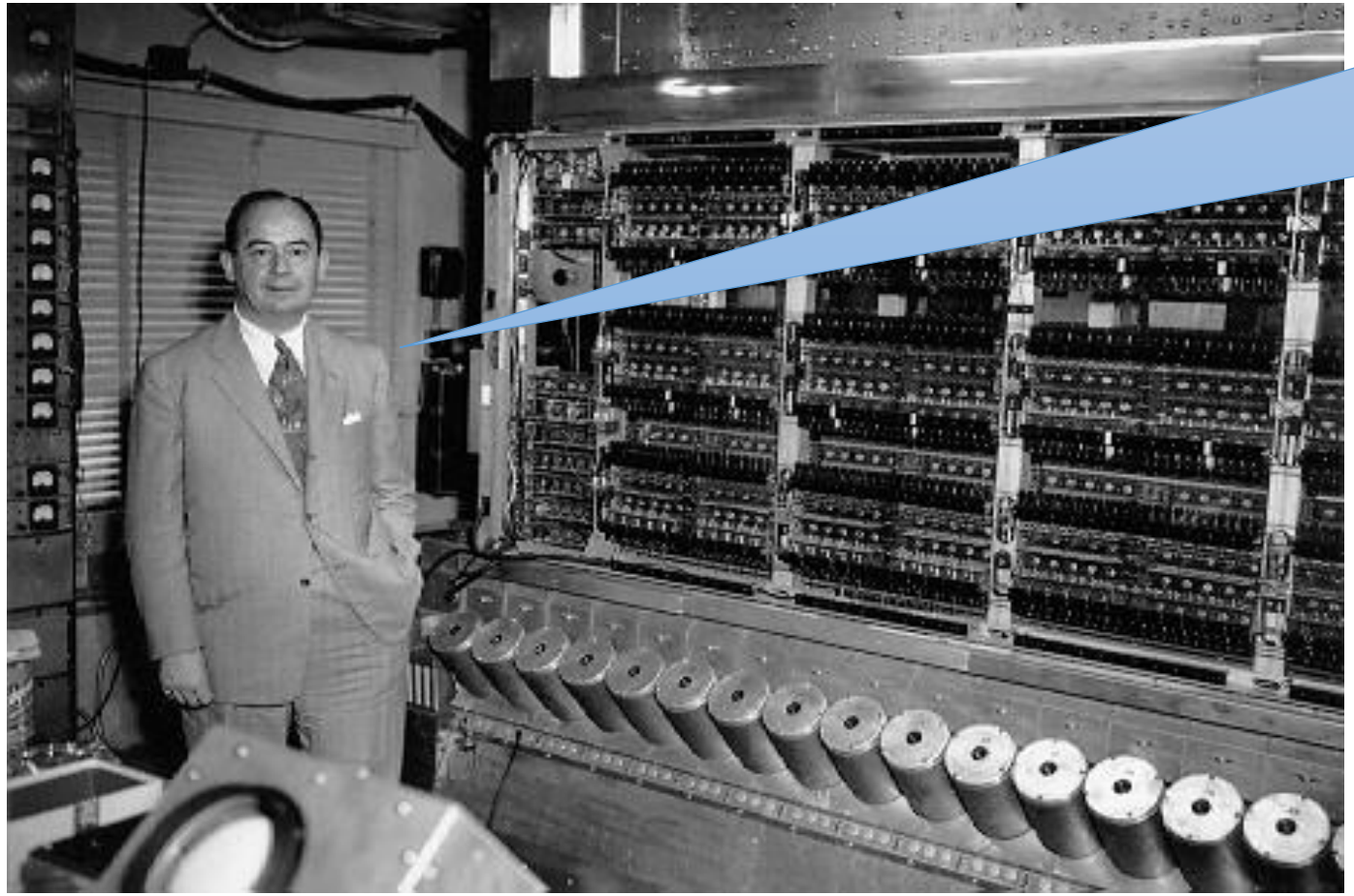
- Harvard Architecture:



- Von Neumann Architecture:



Von Neumann



John von Neumann

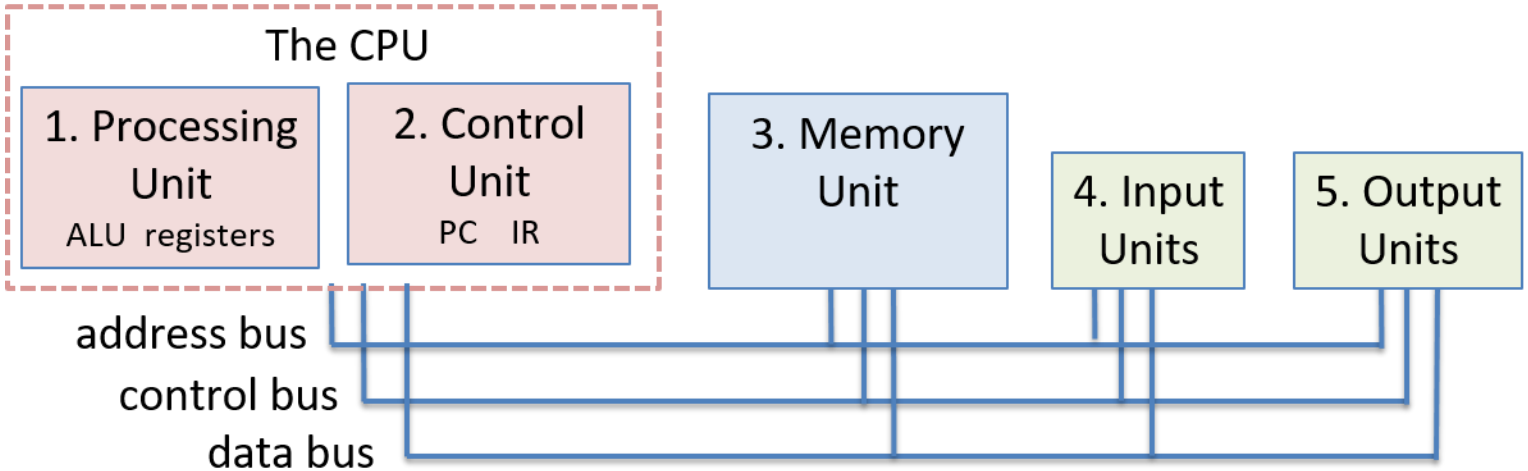
"The father of modern machines"

Stored Program Concept

EDVAC 1945

Von Neumann Architecture Model

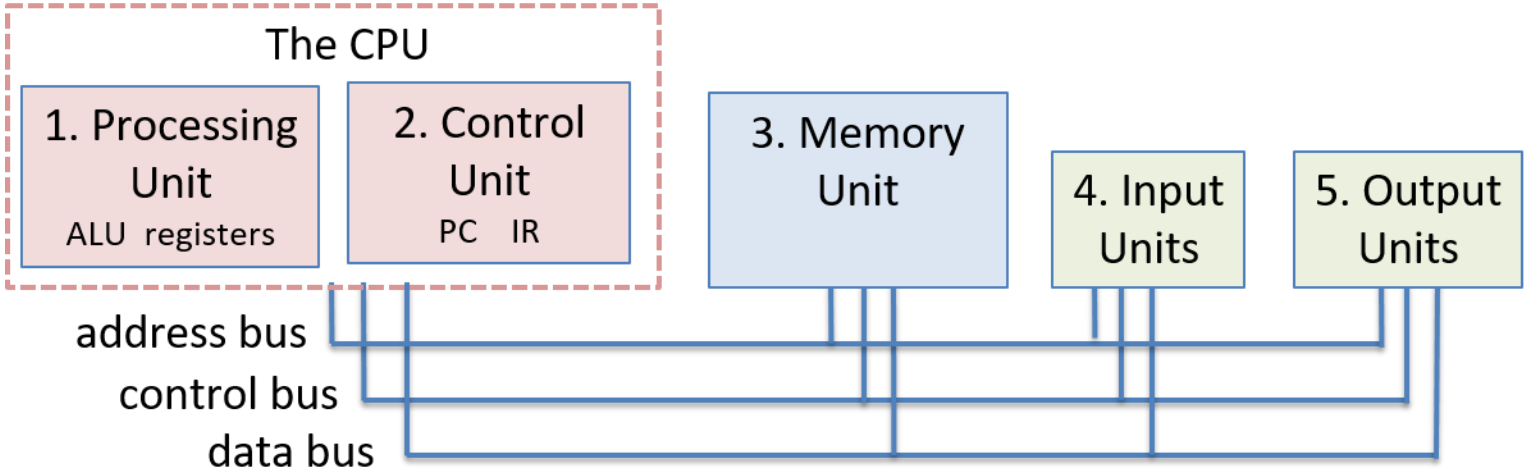
- Computer is a generic computing machine:
  - Based on Alan Turing's Universal Turing Machine
  - Stored program model: computer stores program rather than encoding it (feed in data and instructions)
  - No distinction between data and instructions memory
- 5 parts connected by buses (wires):
  - Memory,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



The CPU

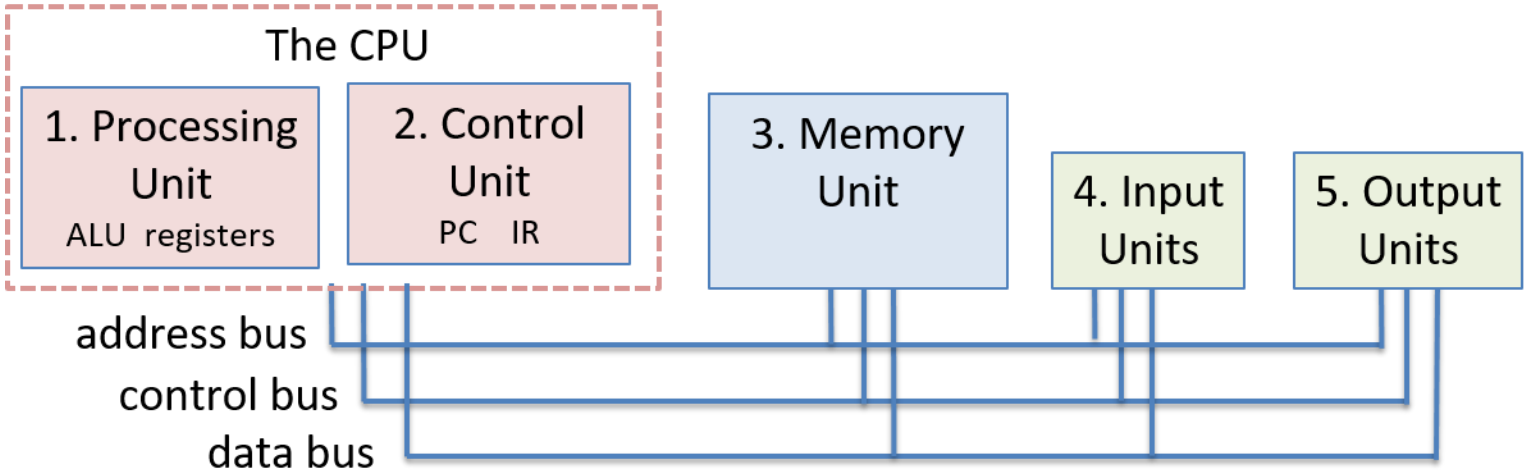
1. Processing Unit: Execute instructions to produce a result
   - ALU (arithmetic logic unit): set of circuits for arithmetic (ADD, SUB, etc.)
   - Registers: temporary storage for instructions (scratch space)

2. Control Unit: Keep track of which instruction to execute next and what that instruction says to do.



Memory

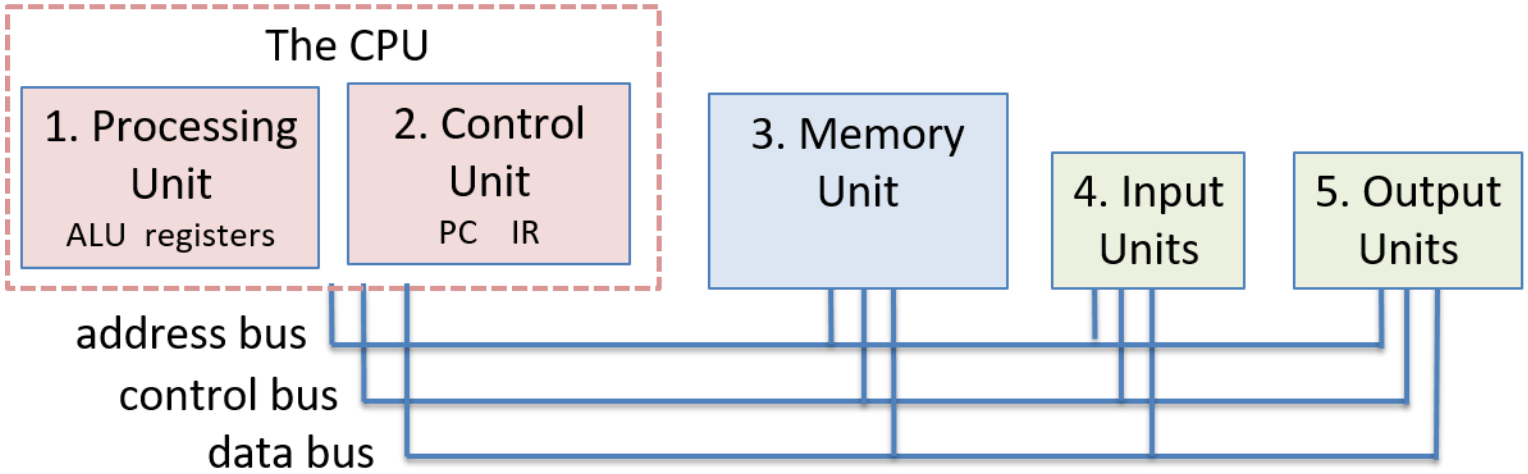
3. Data and instruction storage in “main memory” (RAM)
   - Each byte in memory has a unique address



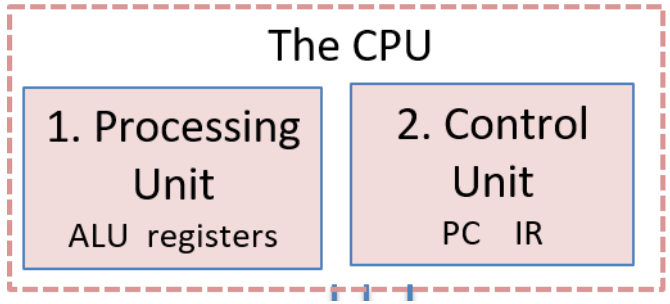
Memory

4. Input: Data coming into the CPU from outside sources
   - keyboard, mouse, network, hard drive

5. Output: Data leaving the CPU to the outside world
   - video display, audio, network, hard drive, printer



Goal: Build a CPU (model)



<u>Three main classifications of hardware circuits:</u>

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

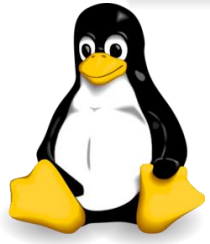
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

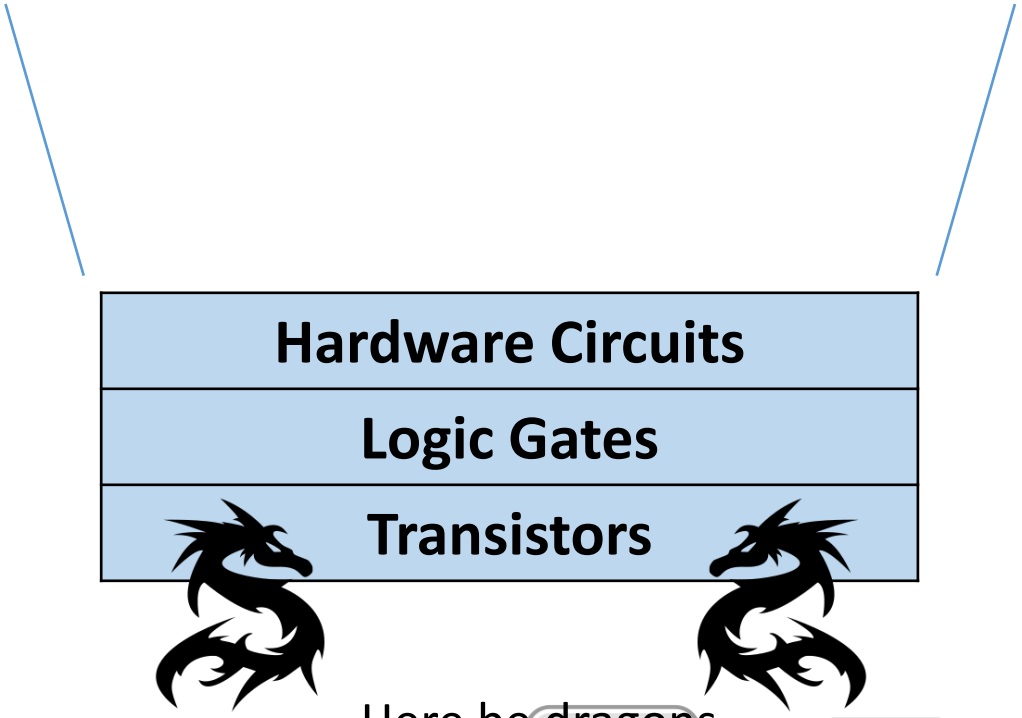
Software library
Reusable functionality

Operating system
Manage resources



Complex devices
Compute & I/O

Abstraction



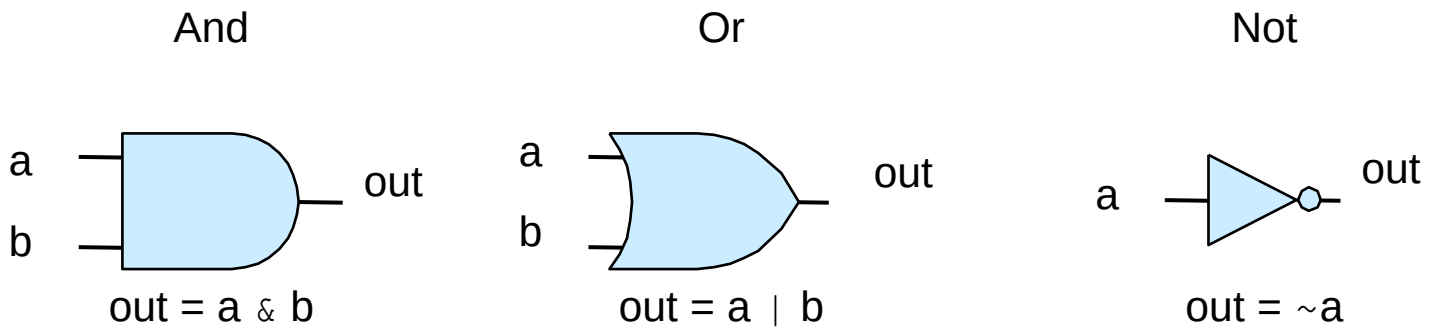
Complex devices
Compute & I/O

Here be dragons.
(Electrical Engineering)
…
(Physics)

Logic Gates

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

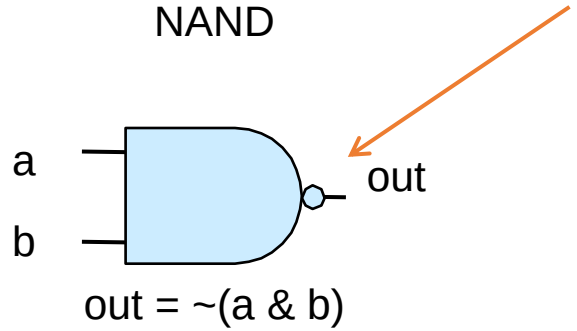
Output: Boolean value result of Boolean function
Always present, but may change when input changes

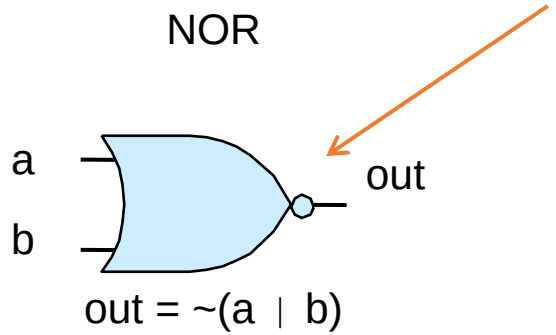


A	B	A & B	A | B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

More Logic Gates

Note the circle on the output. This circle means bitwise “not” (flip bits).

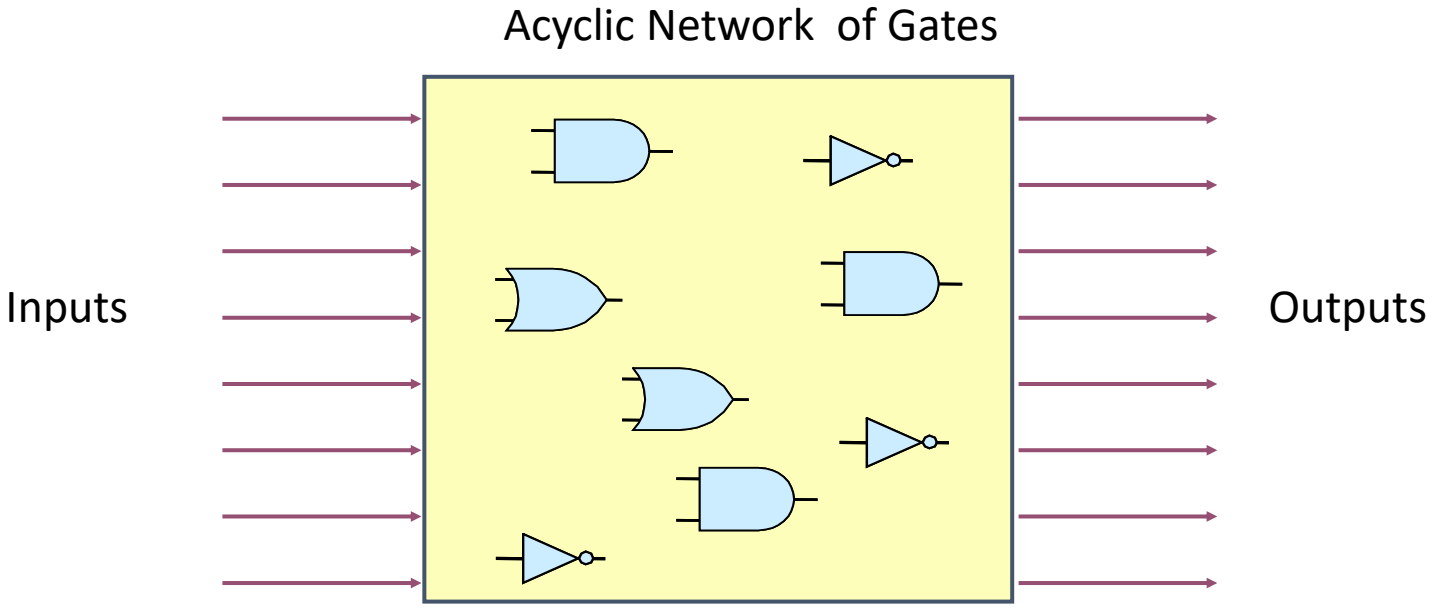




A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

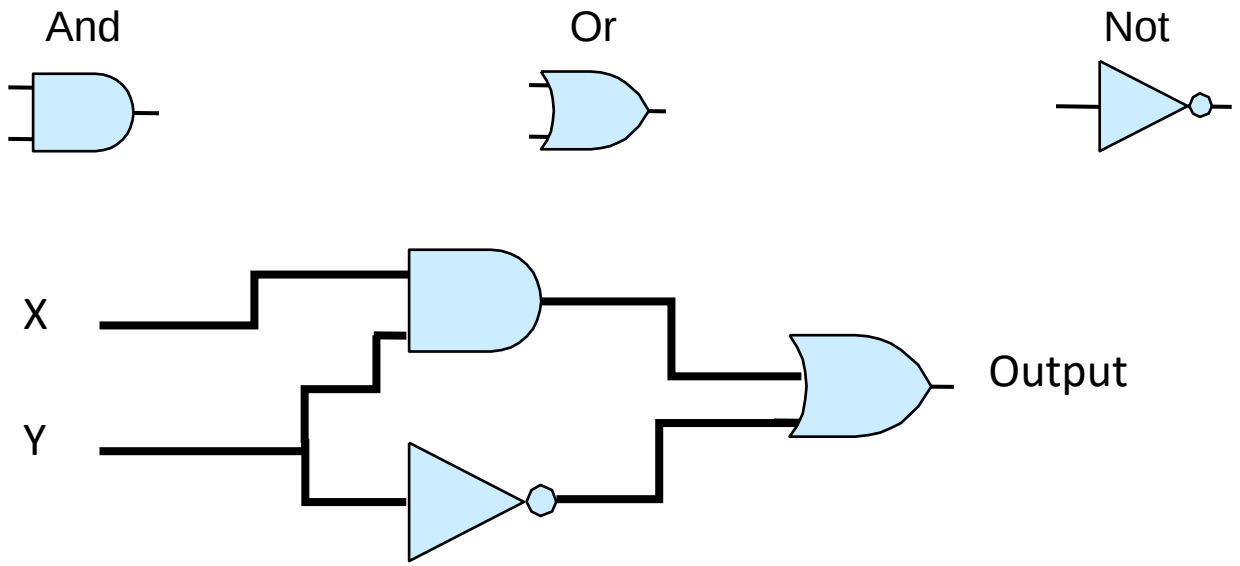
Combinational Logic Circuits

- 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?



Clicker Choices

X	Y	Out_A	Out_B	Out_C	Out_D	Out_E
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

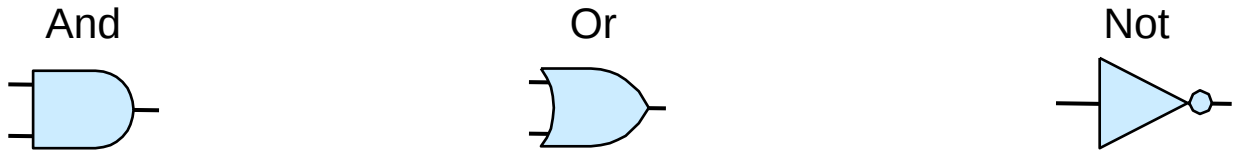
Building more interesting circuits…

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

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- Q: When is A^B ==1?

Which of these is an XOR circuit?



- General strategy:

1. Determine truth table (given ->)

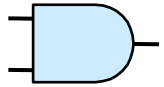
2. Determine for which rows the result is 1
   - express each row with 1 result in terms of input values A, B combined with AND, NOT
   - combine each row expression with OR

3. Translate expression to a circuit

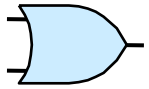
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Which of these is an XOR circuit?

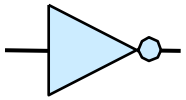
And



Or



Not



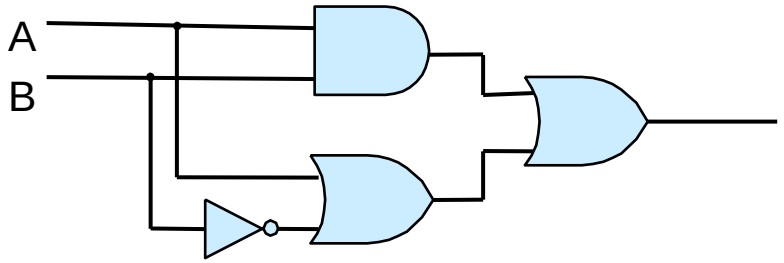
Draw an XOR circuit using AND, OR, and NOT gates.

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

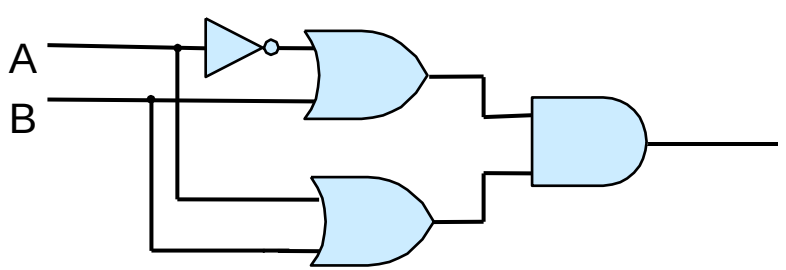
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Which of these is an XOR circuit?

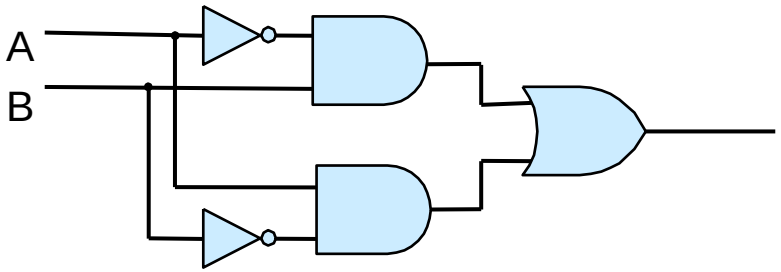
A:



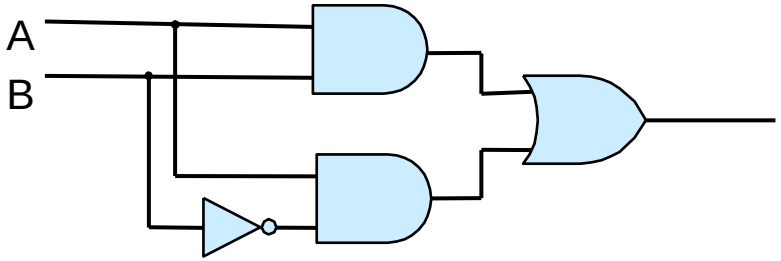
B:



C:



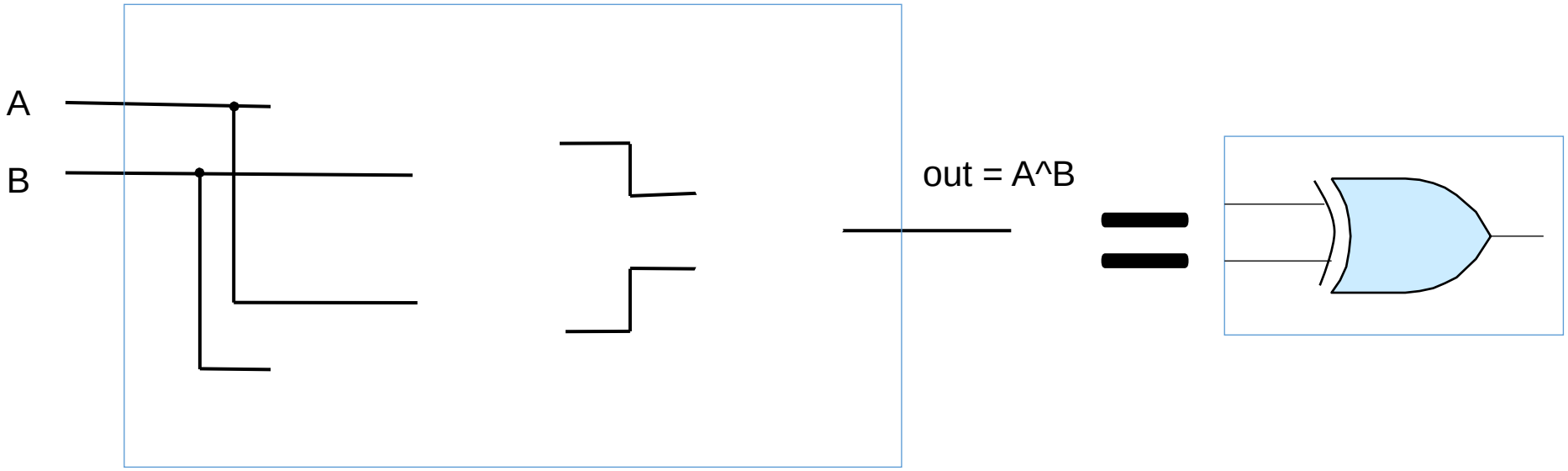
D:



E: None of these are XOR.

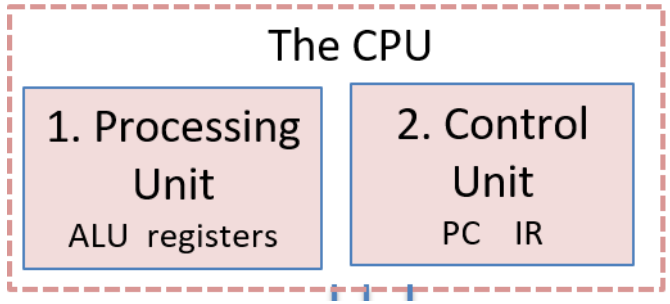
XOR Circuit: Abstraction

```
A^B  == (~A & B) | (A & ~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:
```

Recall Goal: Build a CPU (model)



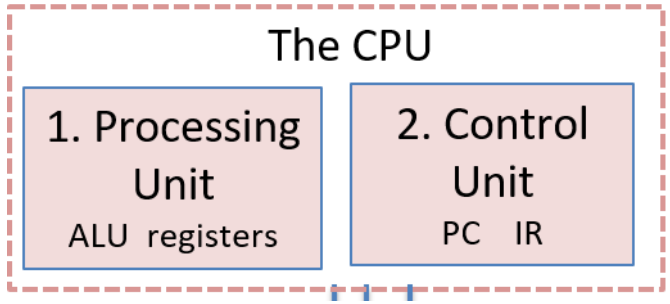
<u>Three main classifications of hardware circuits:</u>

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)



<u>Three main classifications of hardware circuits:</u>

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!

Arithmetic Circuits

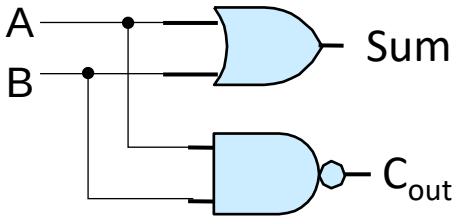
- 1 bit adder: A+B
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

A	B	Sum (A + B)	C_{out}
0	0		
0	1		
1	0		
1	1		

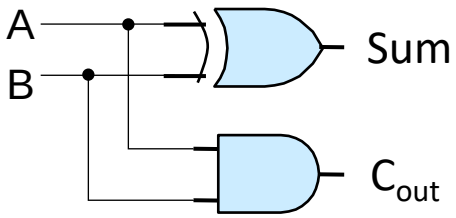
Which of these circuits is a one-bit adder?

A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

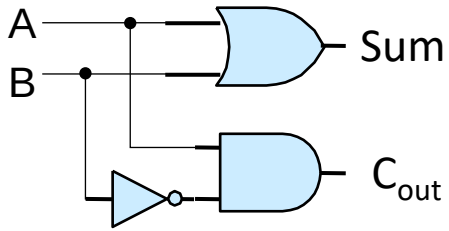
A:



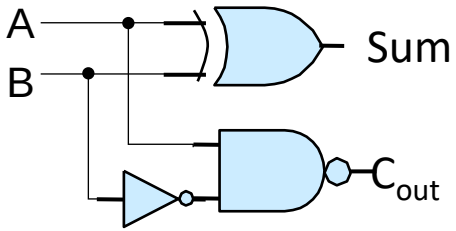
B:



C:



D:



More than one bit?

- When adding, sometimes have *carry in* too

```
    0011010
+   0001111
```

More than one bit?

- When adding, sometimes have *carry in* too

```
   1111
  0011010
+ 0001111
```

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?

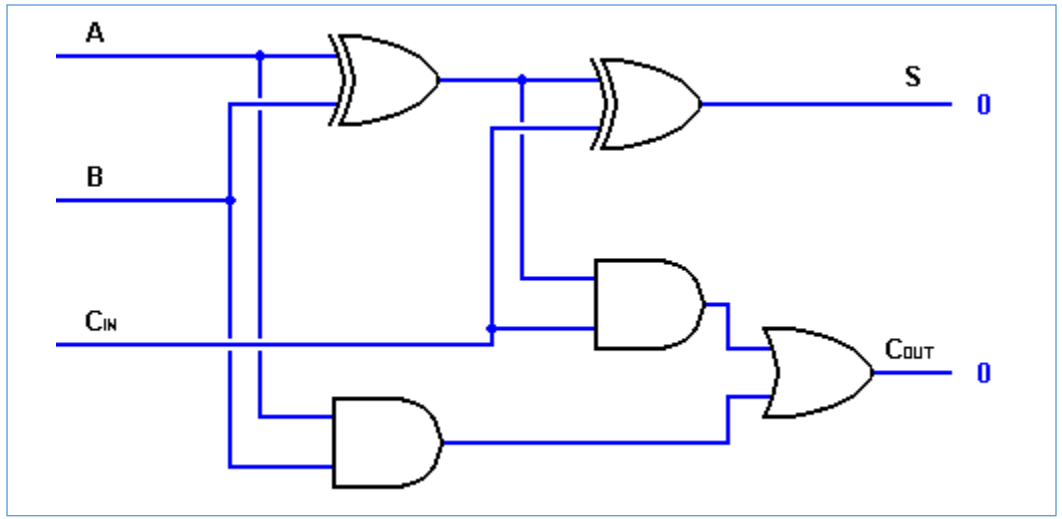
- When is C_{out} 1?

One-bit (full) adder

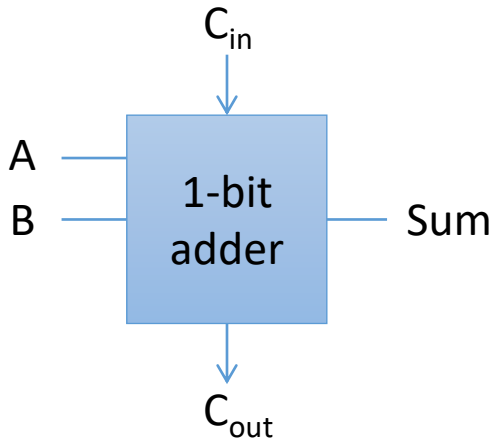
- Need to include:

  carry-in and carry-out

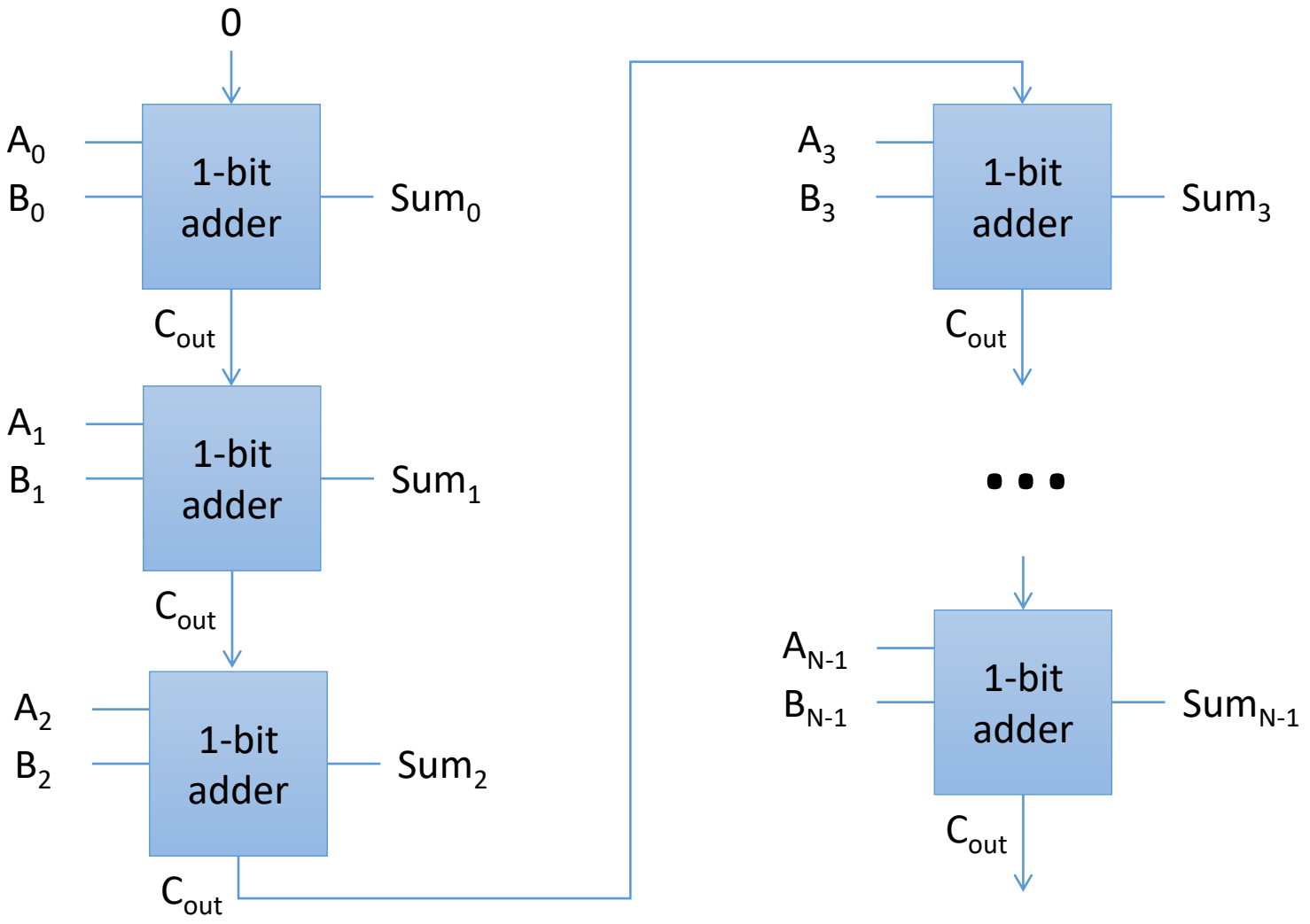
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



=

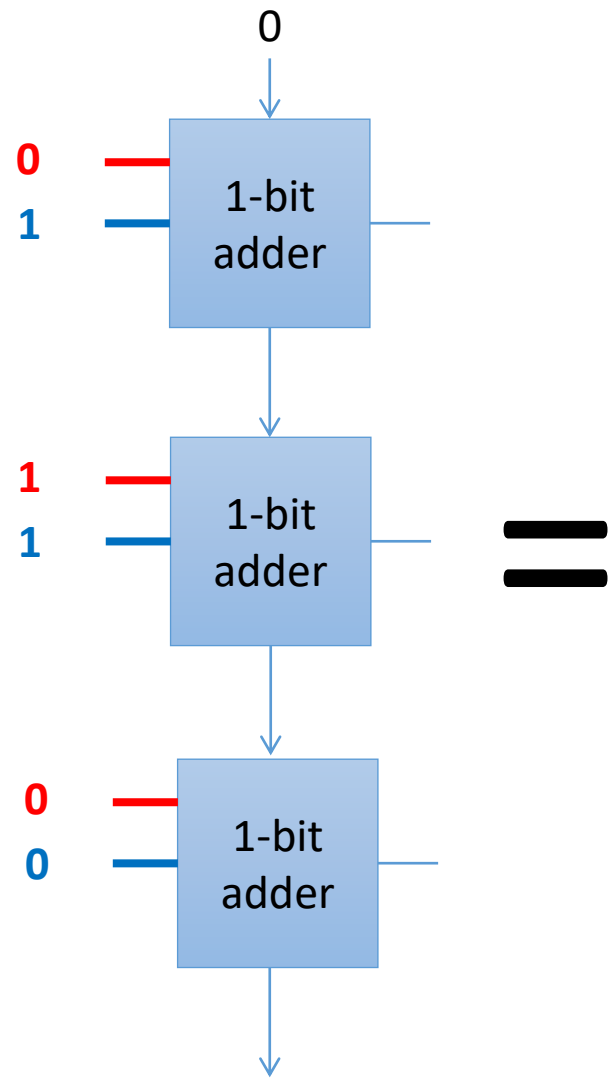


Multi-bit Adder (Ripple-carry Adder)

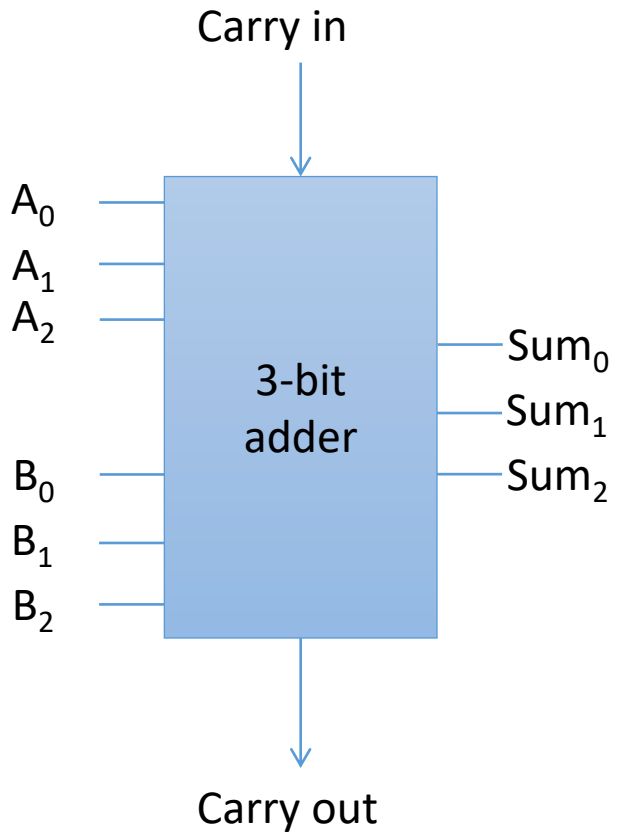


Three-bit Adder (Ripple-carry Adder)

010
+ 011



=

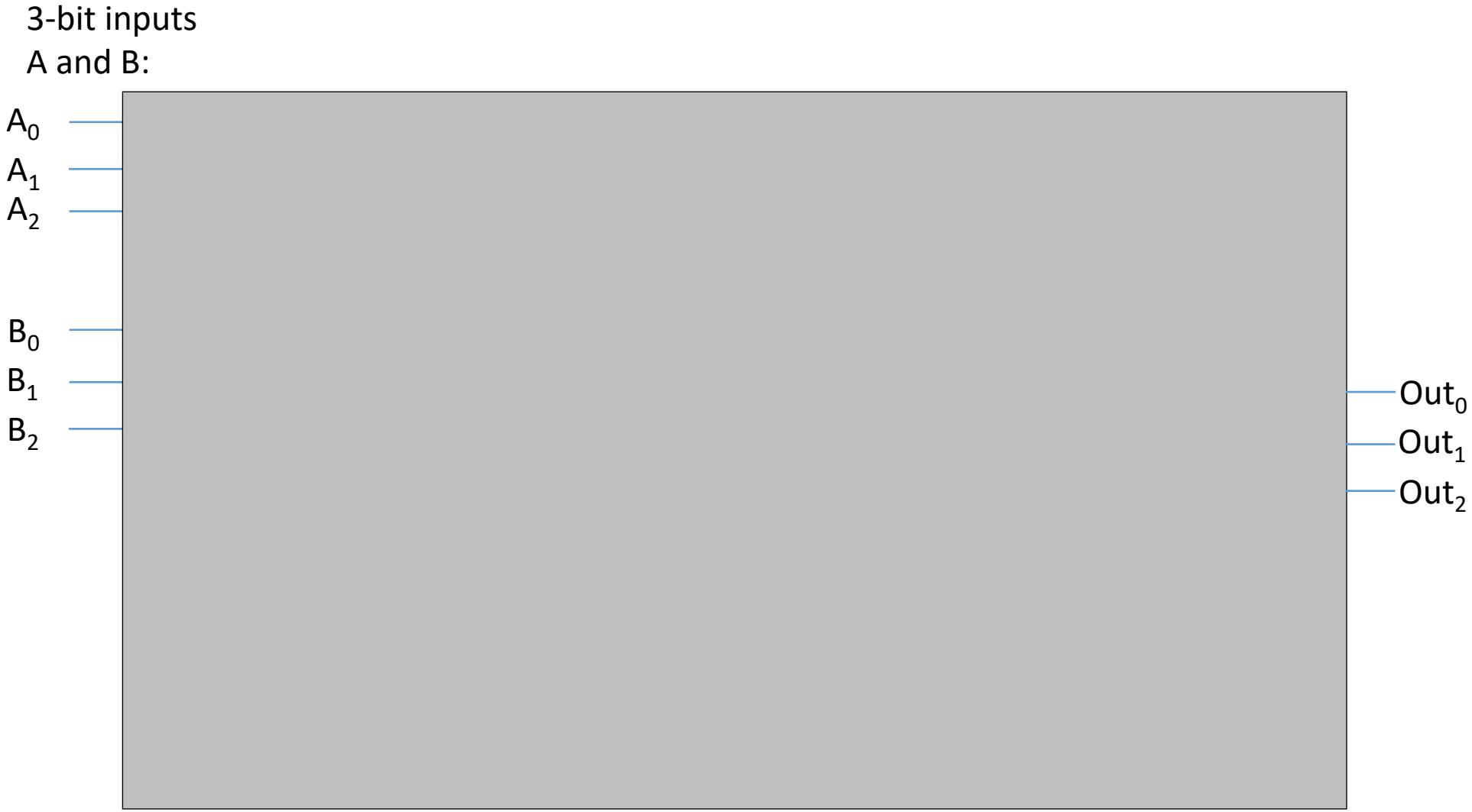


Arithmetic Logic Unit (ALU)

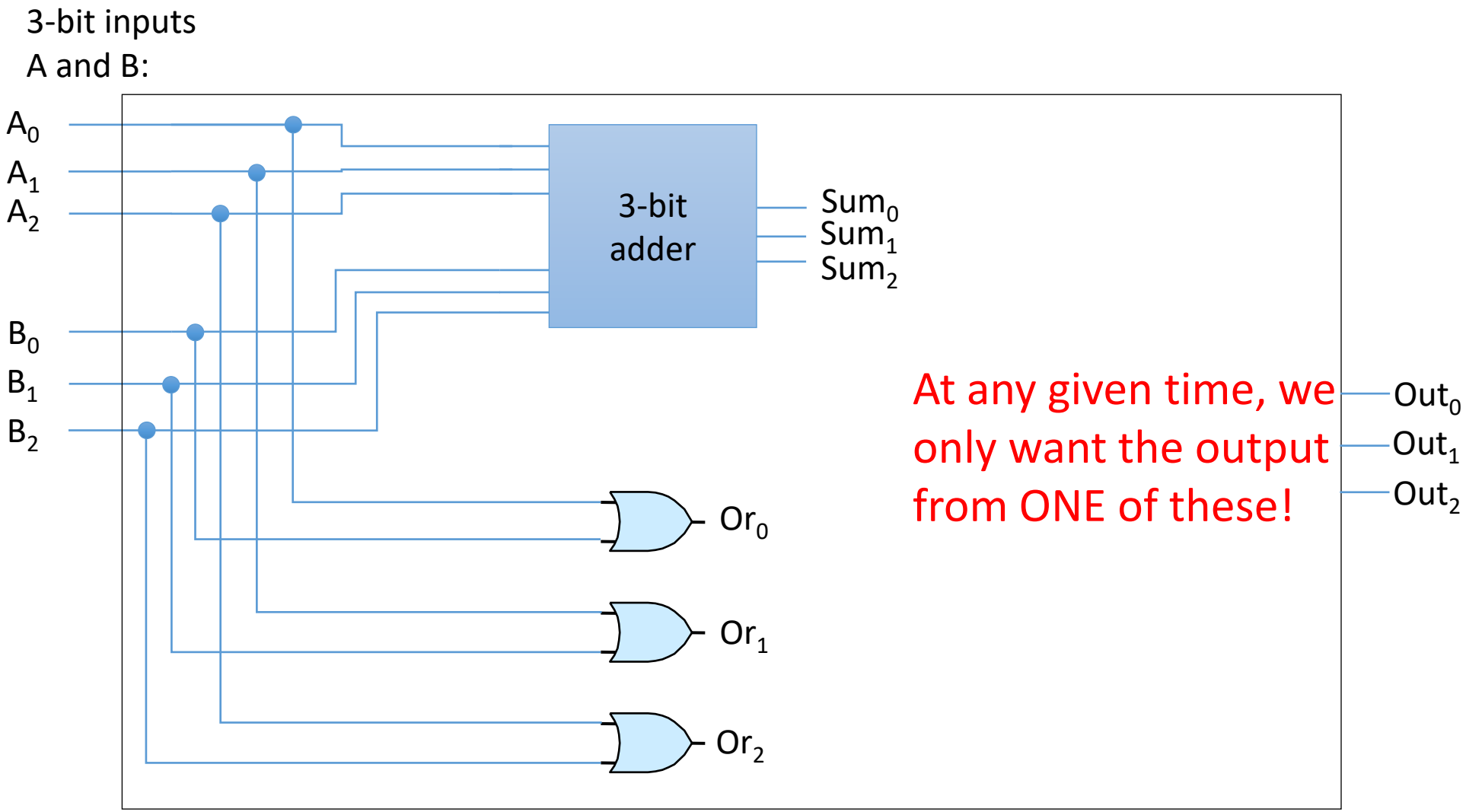
- One component that knows how to manipulate bits in multiple ways
  - Addition
  - Subtraction
  - Multiplication / Division
  - Bitwise AND, OR, NOT, etc.

- Built by combining components
  - Take advantage of sharing HW when possible (e.g., subtraction using adder)

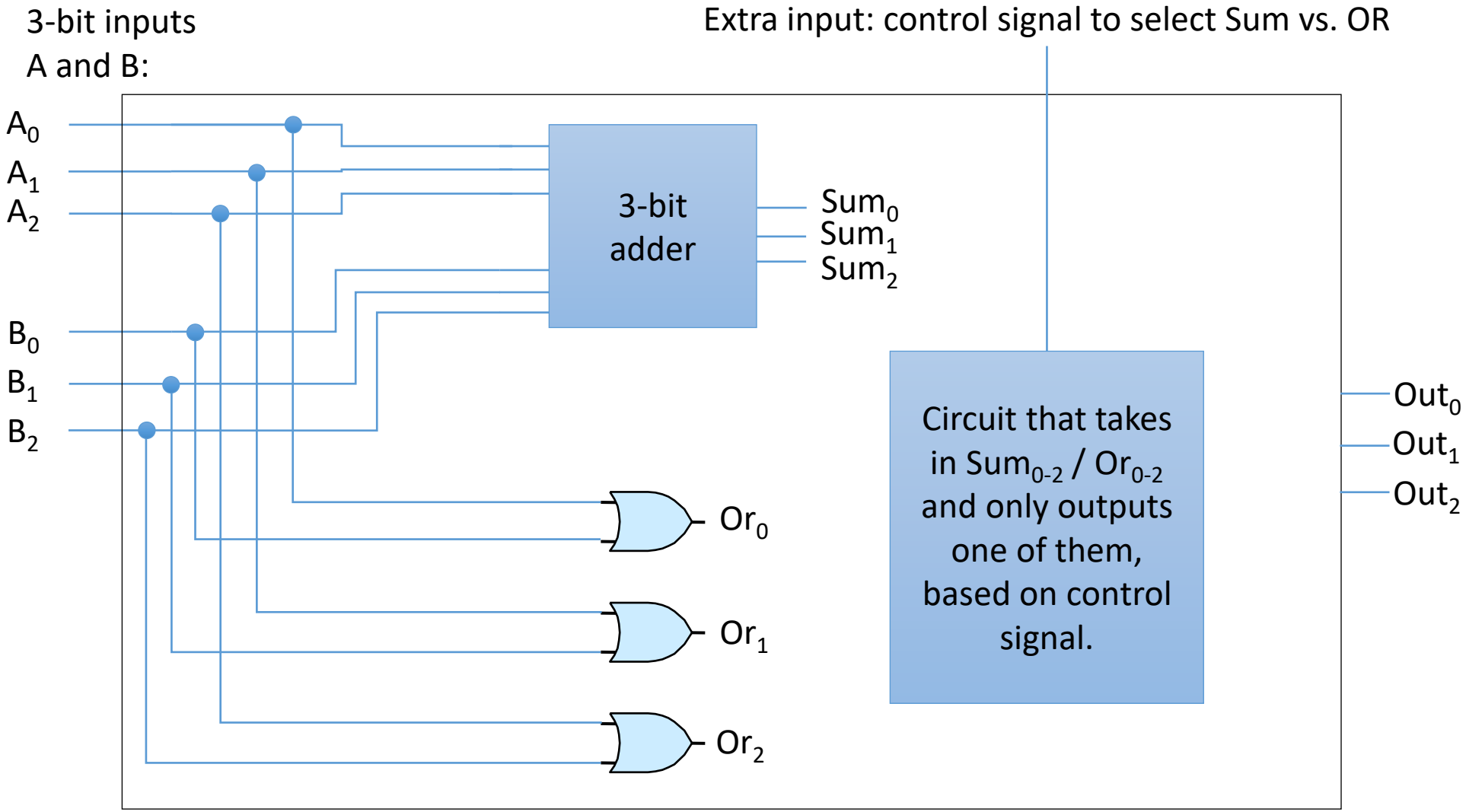
Simple 3-bit ALU: Add and bitwise OR



Simple 3-bit ALU: Add and bitwise OR

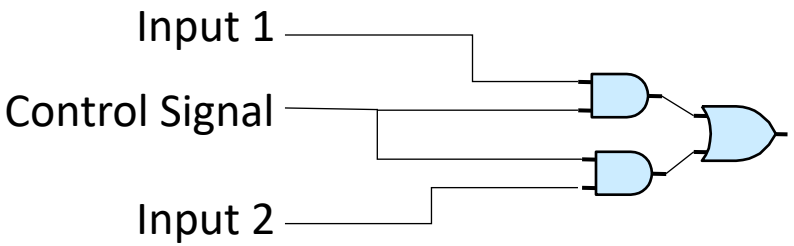


Simple 3-bit ALU: Add and bitwise OR

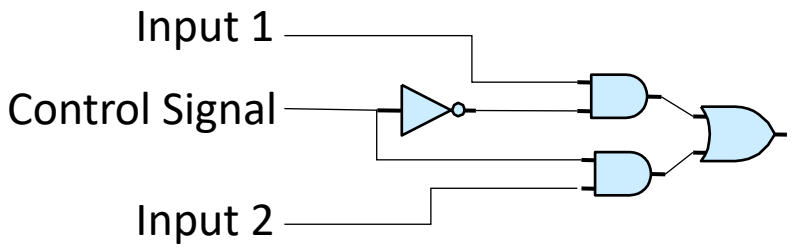


Which of these circuits lets us select between two inputs?

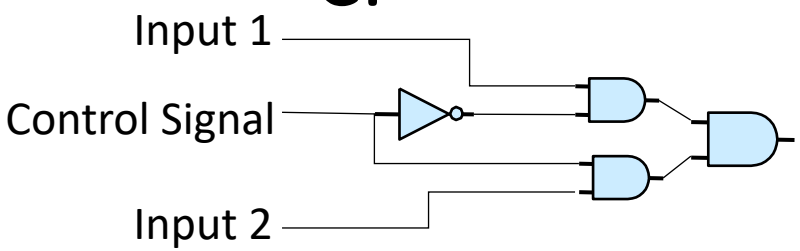
A:



B:



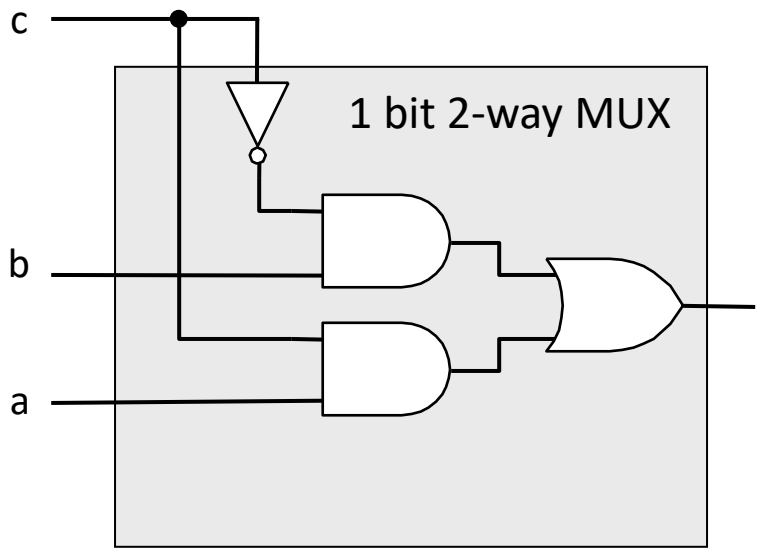
C:



Multiplexor: Chooses an input value

Inputs: 2^N data inputs, N signal bits

Output: is one of the 2^N input values

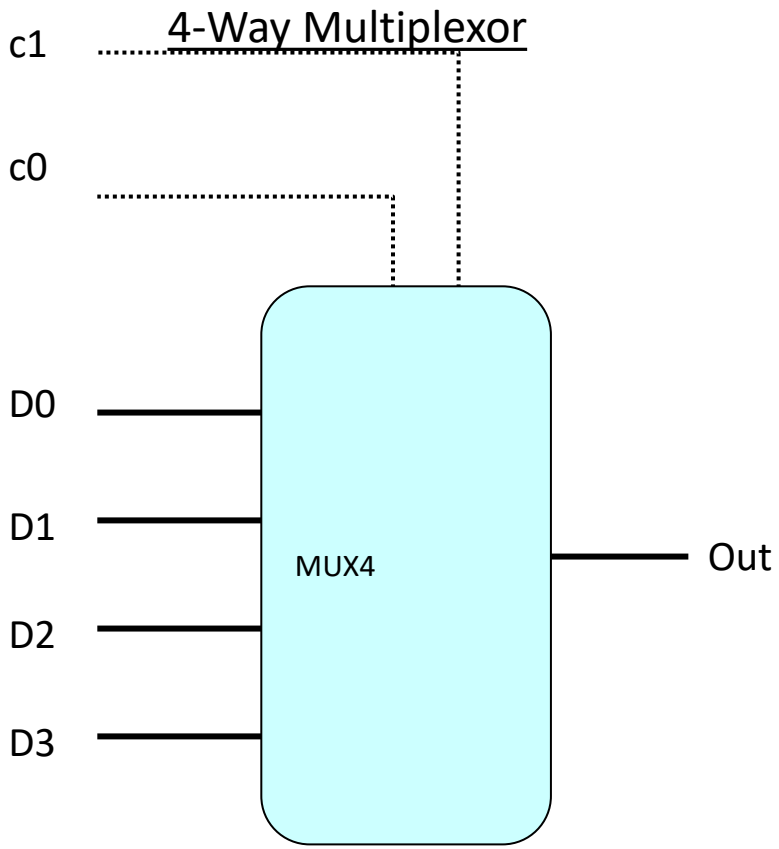



```
out = (c & a)|(~c &b)
```

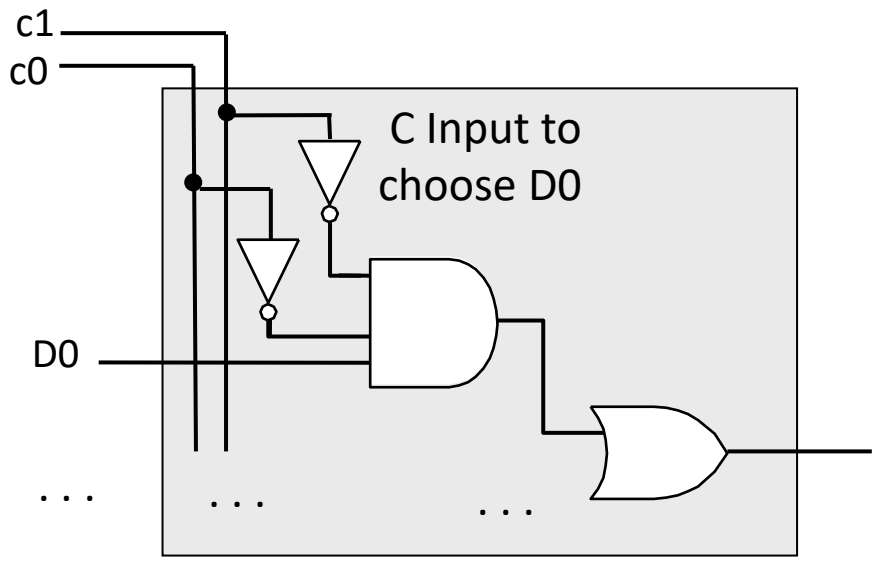
- Control signal c, chooses the input for output
  - When c is 1: choose a, when c is 0: choose b

N-Way Multiplexor

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

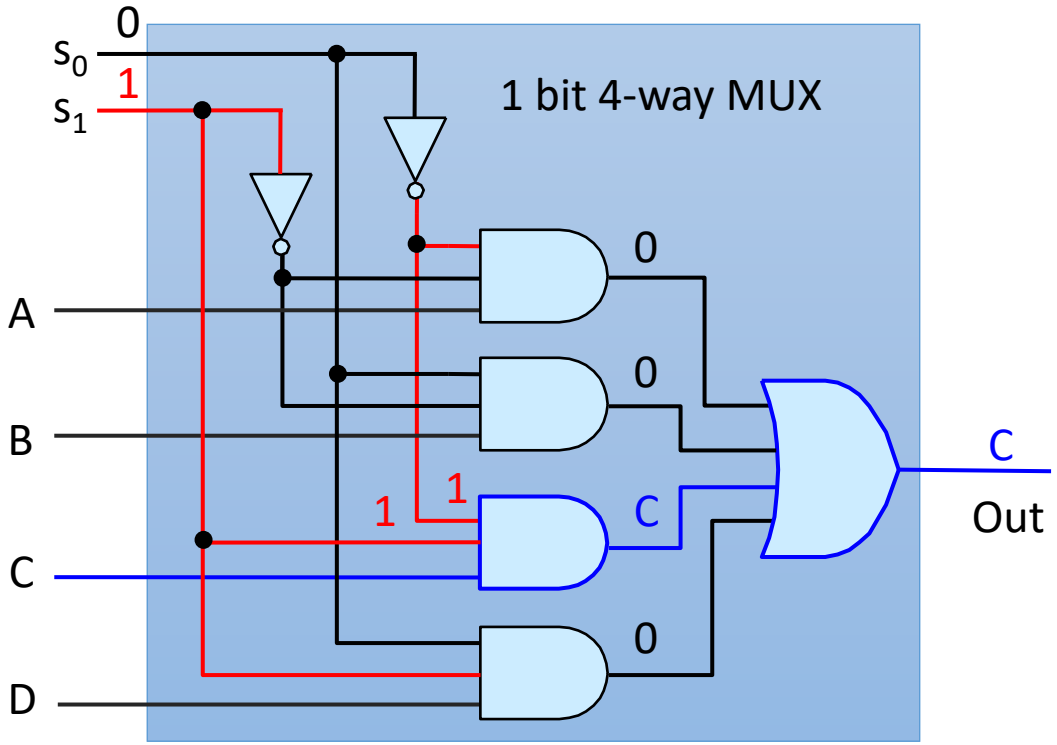


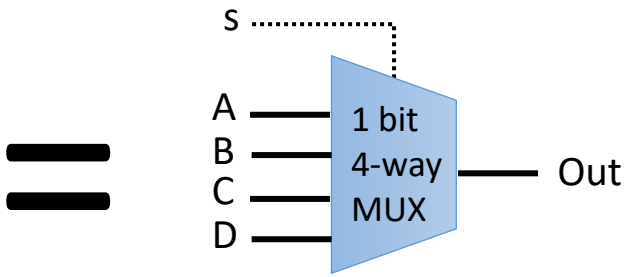
c_1	c_2	Output
0	0	D0
0	1	D1
1	0	D2
1	1	D3



Example 1-bit, 4-way MUX

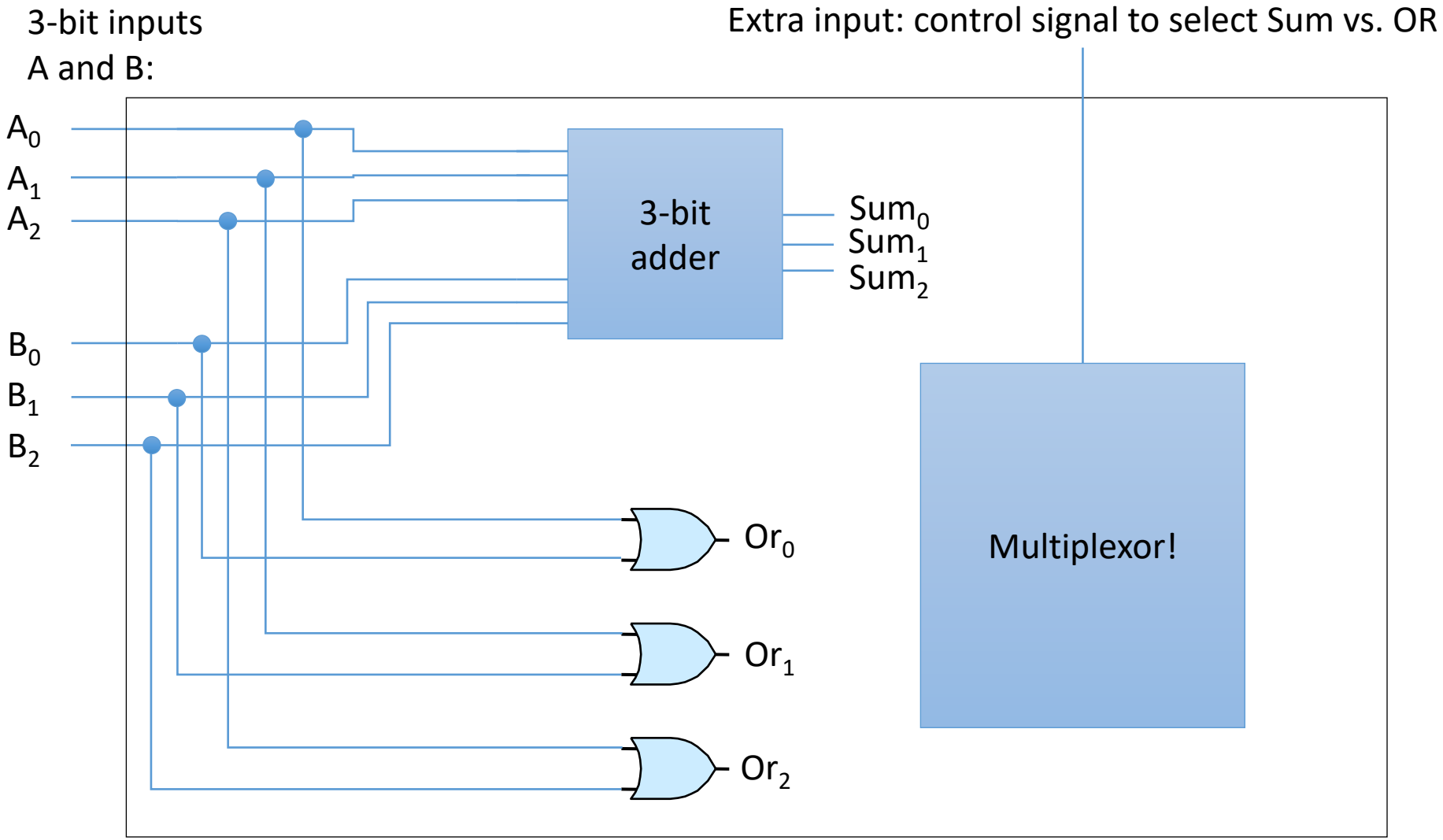
- When select input is 2 (0b10): C chosen as output



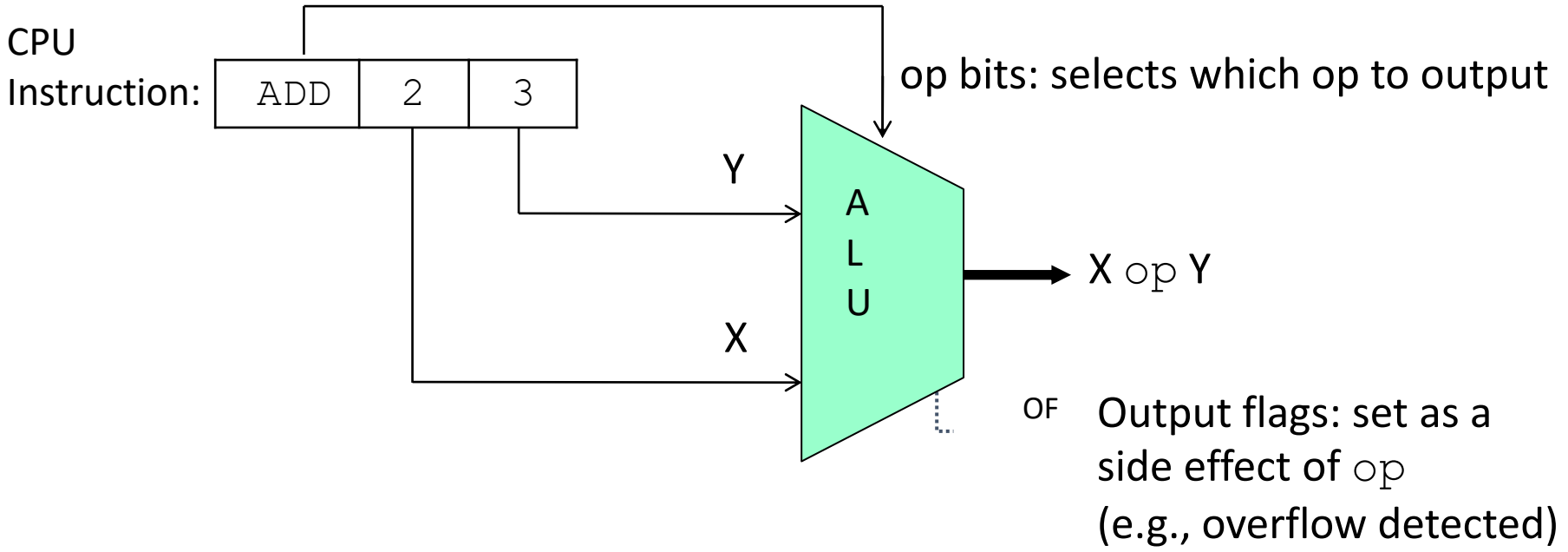


S	Out
0	A
1	B
2	C
3	D

Simple 3-bit ALU: Add and bitwise OR

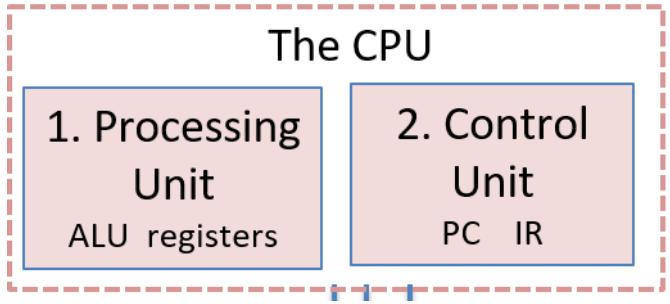


ALU: Arithmetic Logic Unit



- 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)



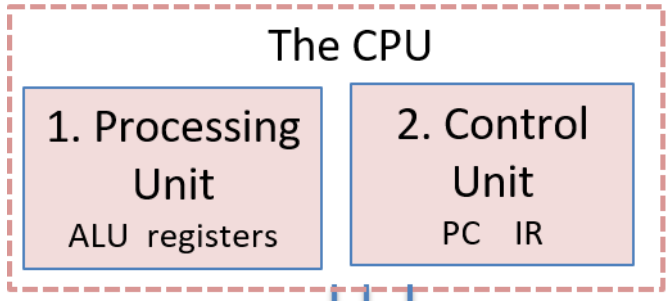
<u>Three main classifications of hardware circuits:</u>

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)



<u>Three main classifications of hardware circuits:</u>

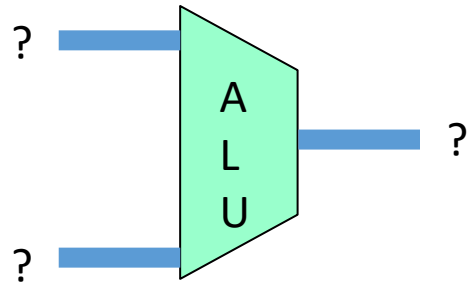
2. Storage: to store binary values
   - Example: set of CPU registers ("register file") to store temporary values

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 to the ALU input values come from?
  - Where do we store the result?
  - What does this “register” thing mean?



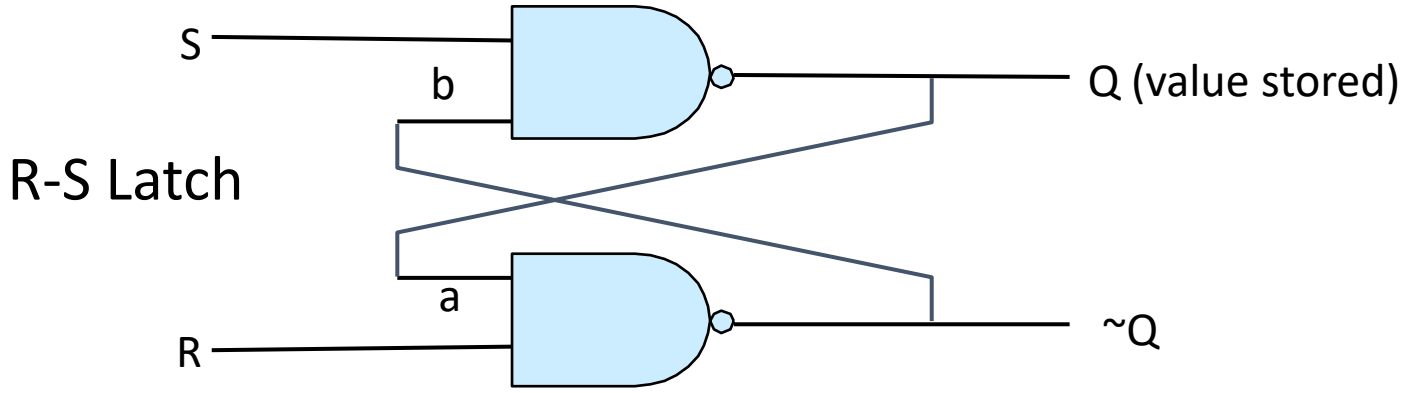
Memory Circuit Goals: Starting Small

- 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

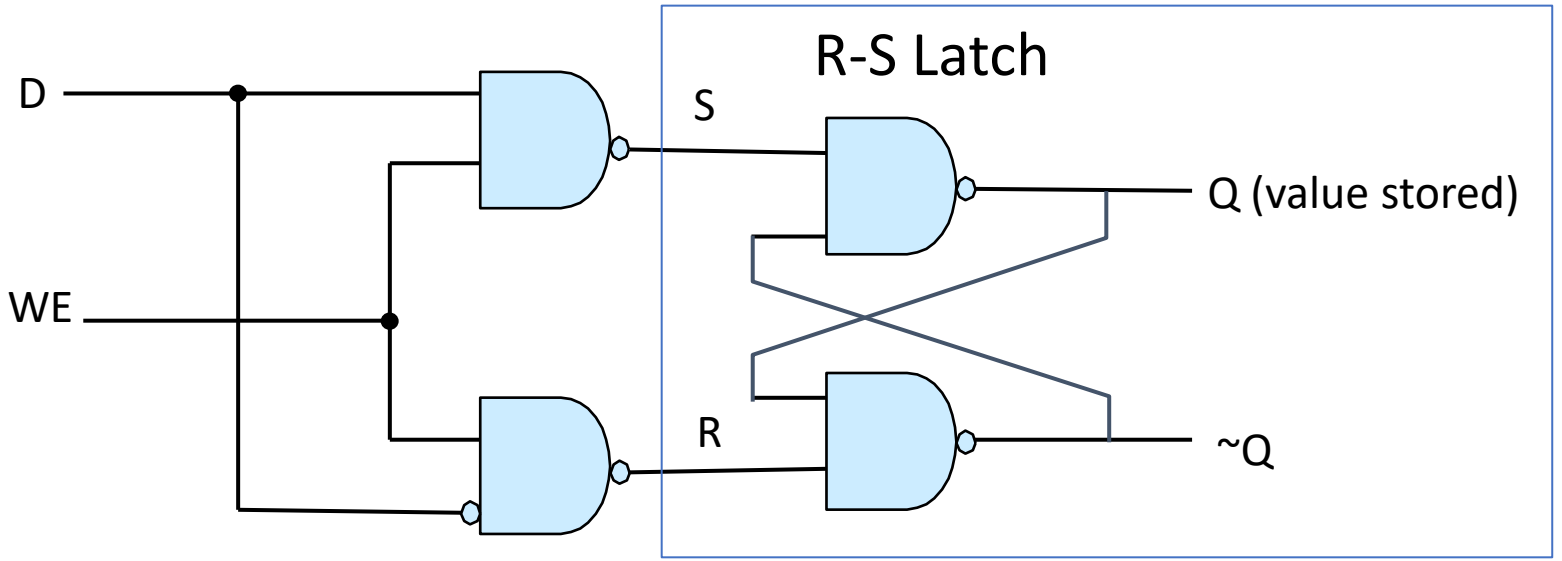
R and S are never both simultaneously 0



- 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

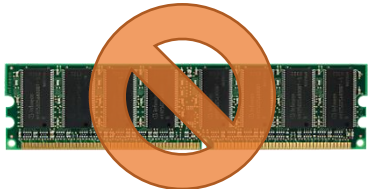
Controls S-R latch writing, ensures S & R never both 0



D: into top NAND, ~D into bottom NAND
WE: write-enabled, when set, latch is set to value of D

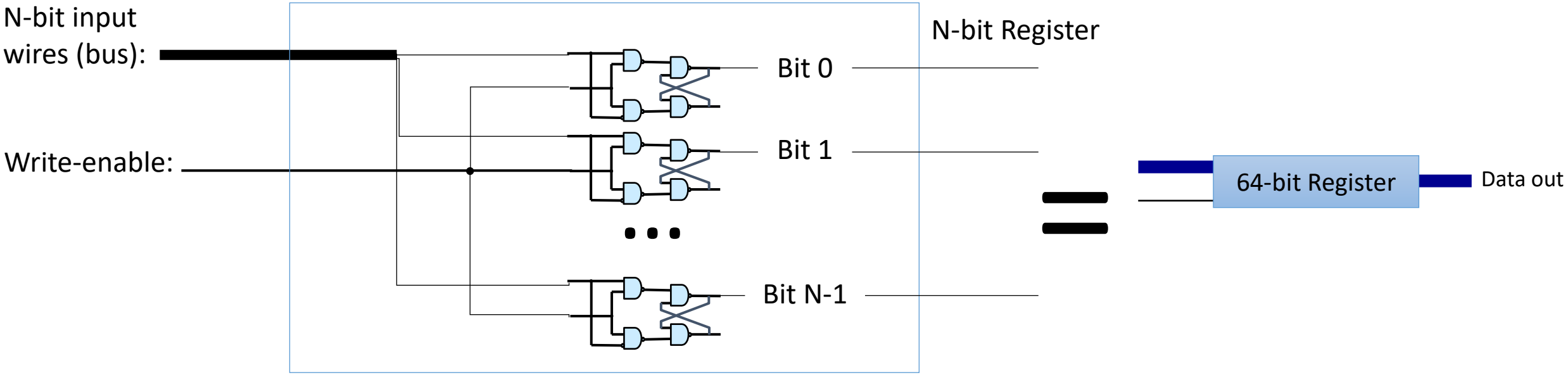
Latches used in registers (up next) and SRAM (caches, later)
Fast, not very dense, expensive

DRAM: capacitor-based:



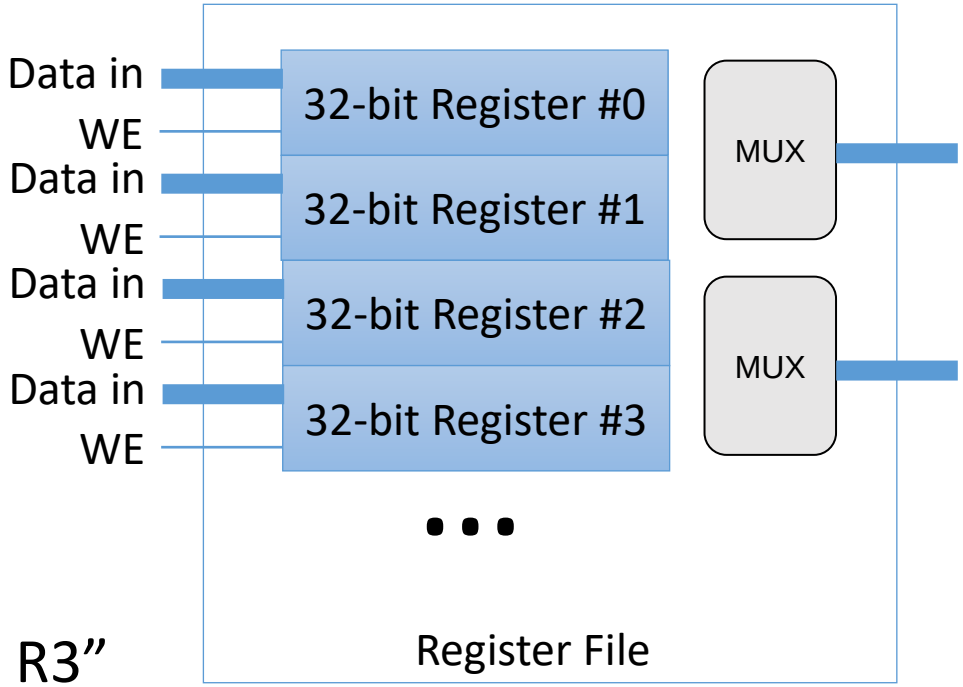
An N-bit Register

- 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

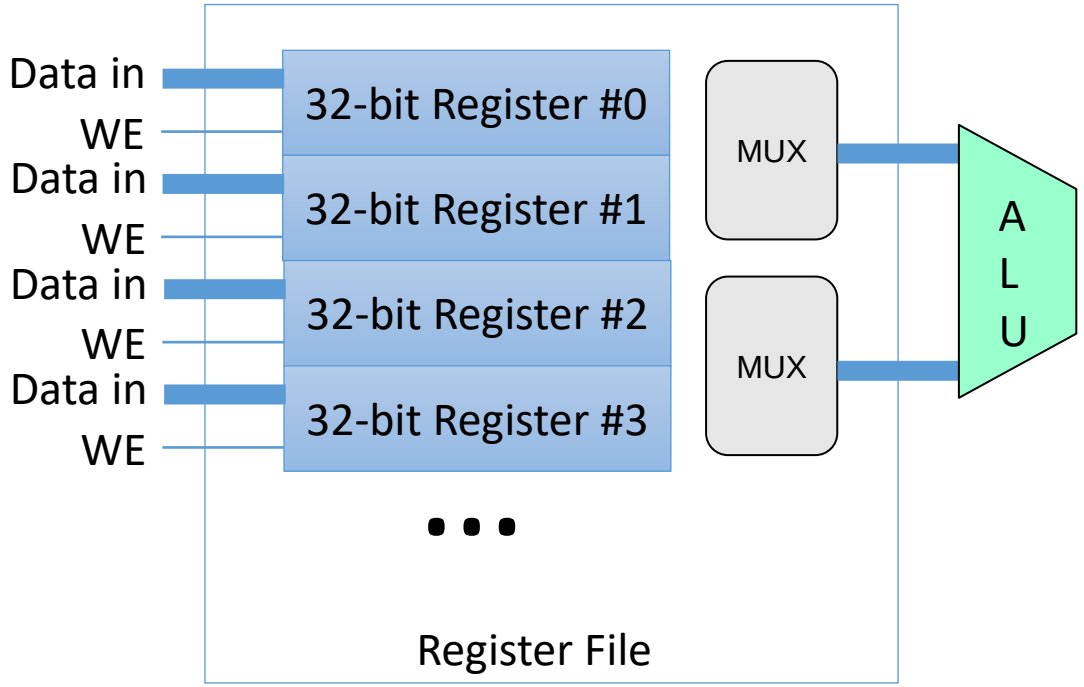
CPU so far…

We know how to store data (in register file).
We know how to perform arithmetic on it, by feeding it to ALU.

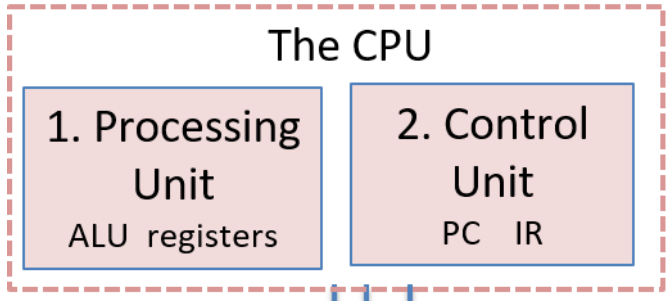
Remaining questions:

- 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 our program:
a series of instructions.



Goal: Build a CPU (model)



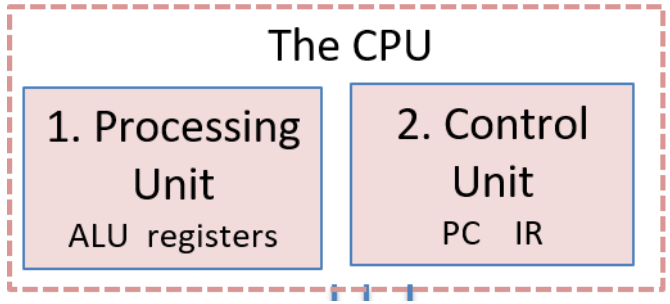
<u>Three main classifications of hardware circuits:</u>

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)



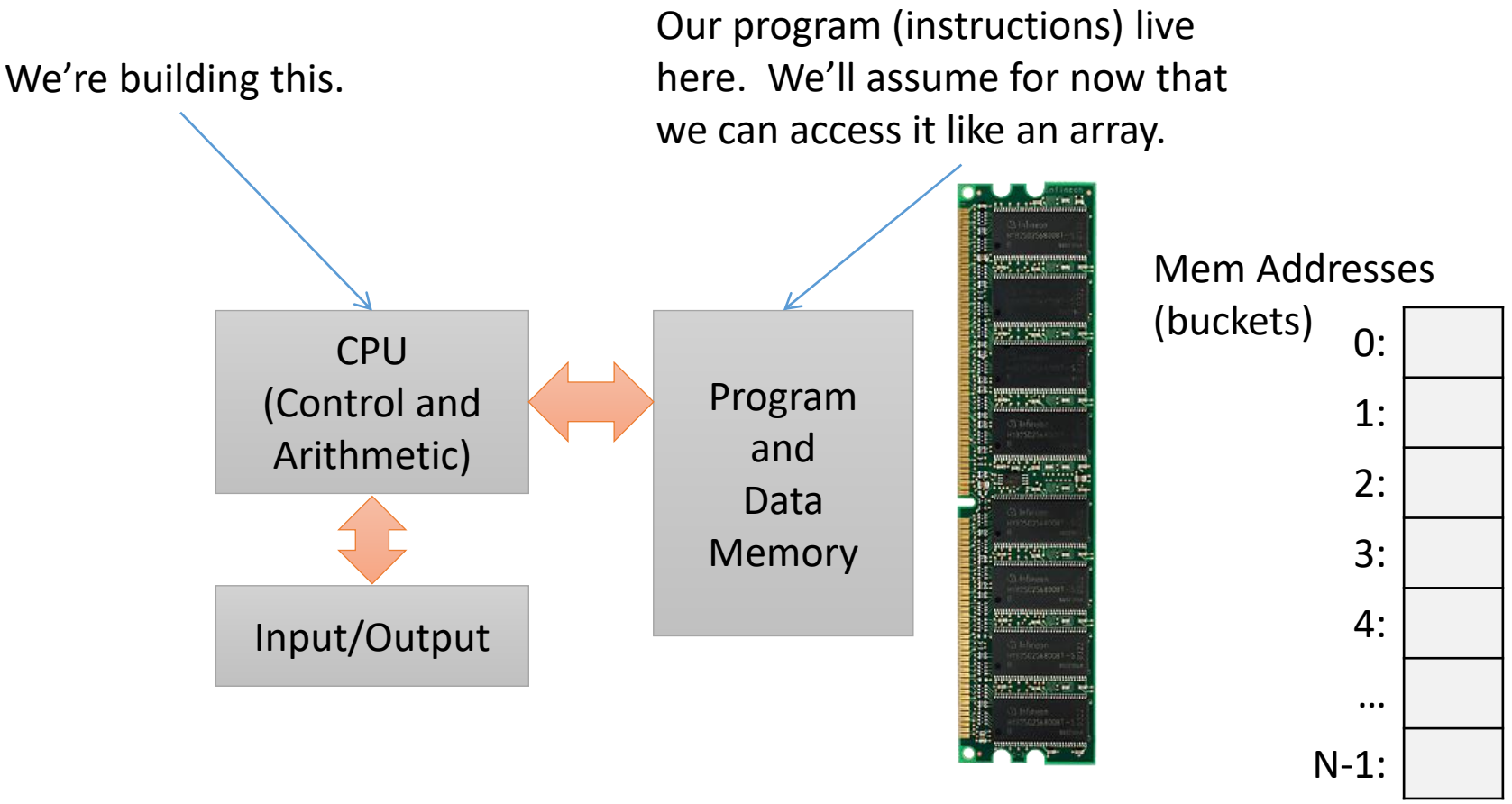
<u>Three main classifications of hardware circuits:</u>

3. Control: support/coordinate instruction execution
   - Example: circuitry to fetch the next instruction from memory and decode it

Keep track of where we are in the program.

Execute an instruction, move on to the next…

Recall: Von Neumann Model



CPU Game Plan

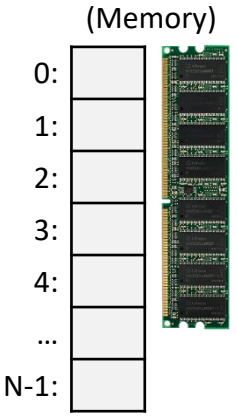
- Fetch instruction from memory
- Decode what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands
- Execute the instruction (arithmetic, etc.)
- Store the result

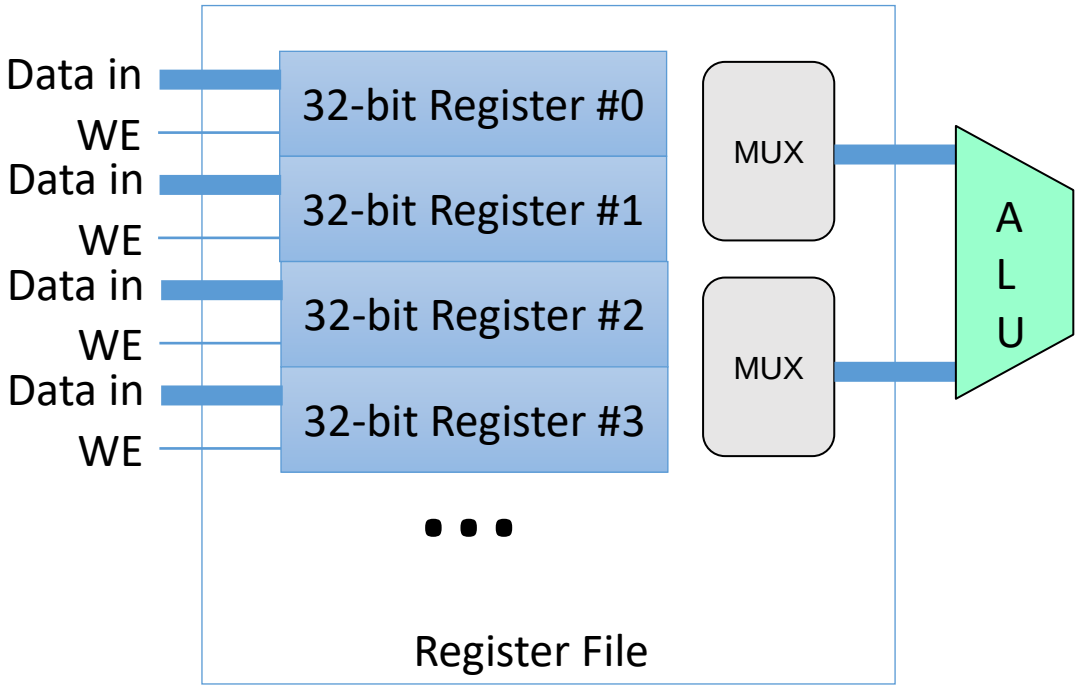
Program State

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

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

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



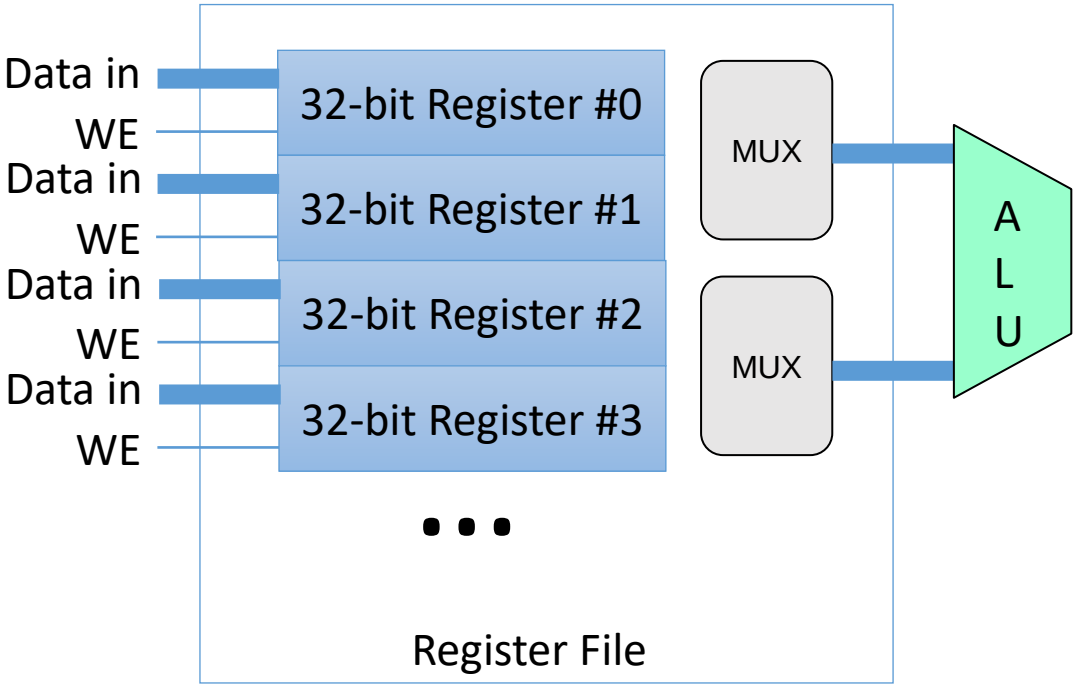


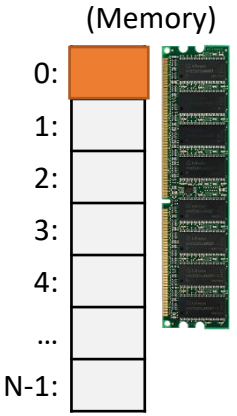
Fetching instructions.

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



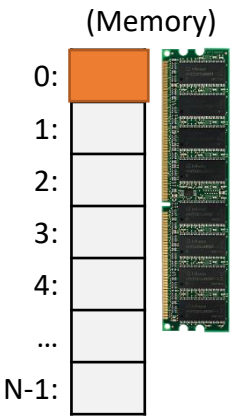


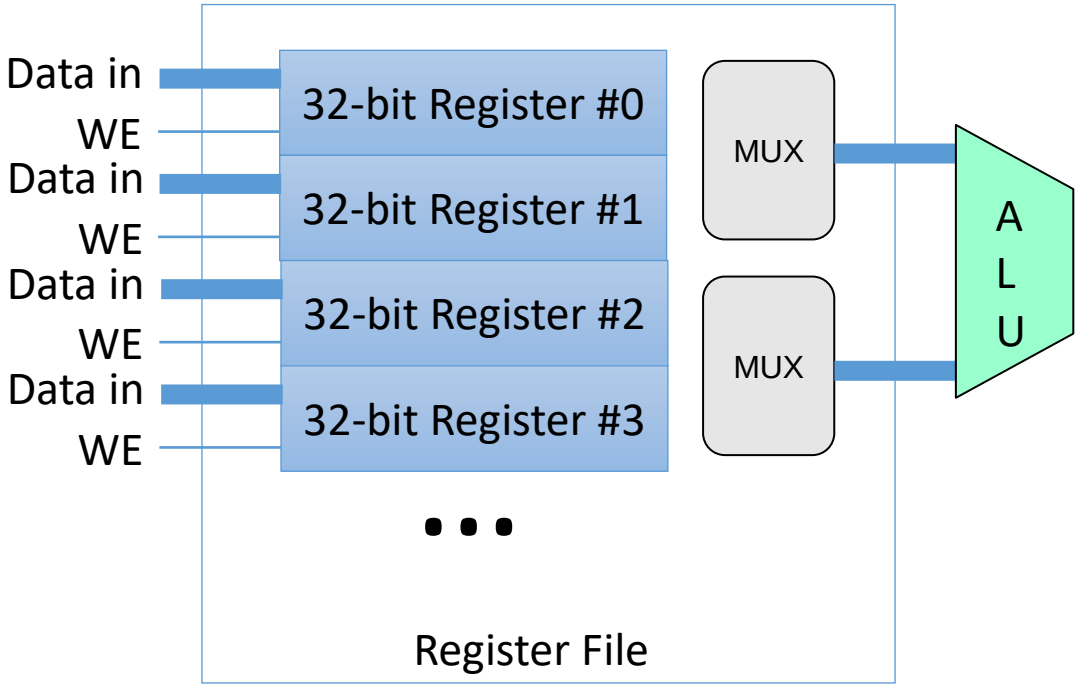
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

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

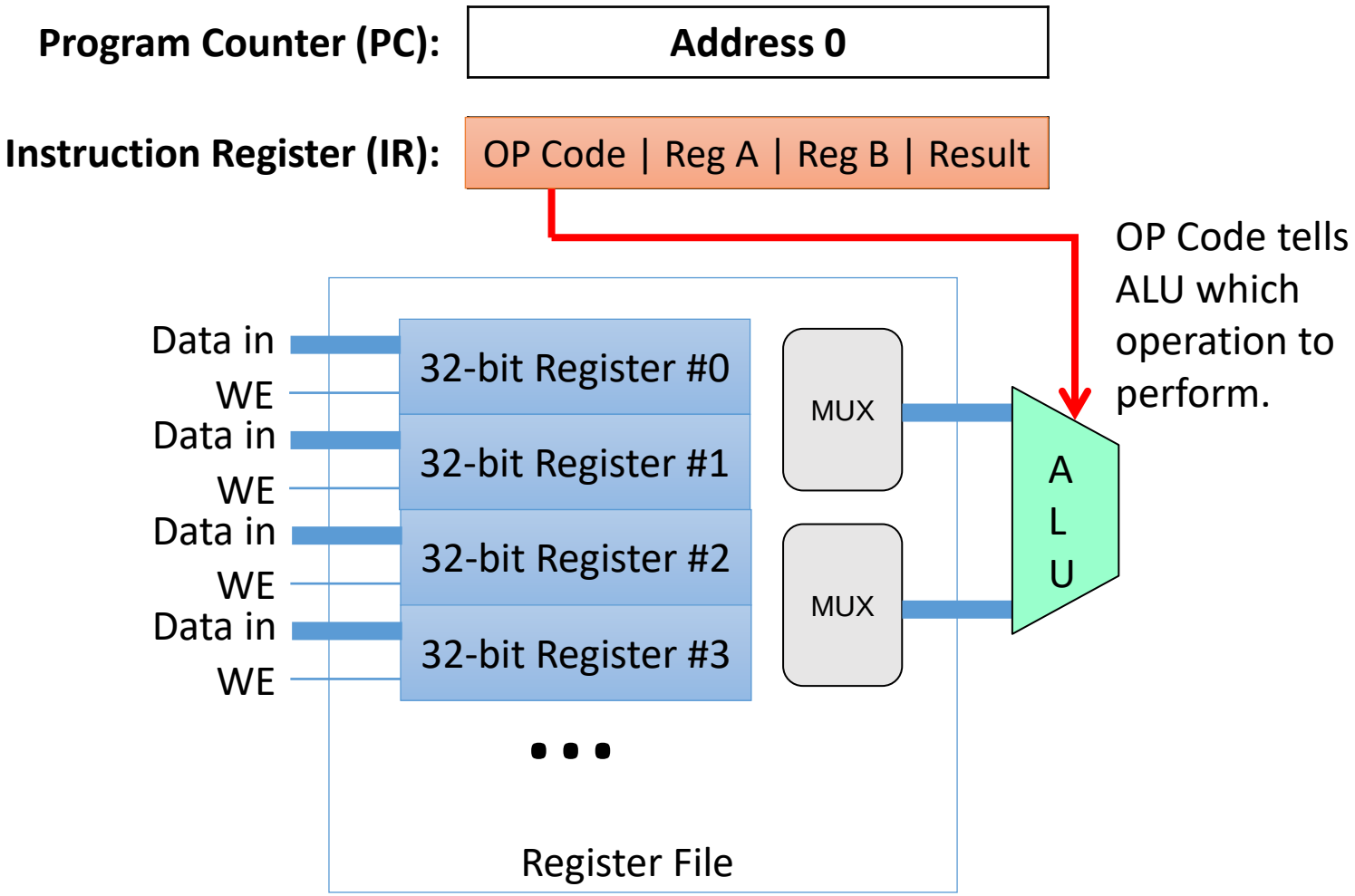
Instruction Register (IR): OP Code | Reg A | Reg B | Result

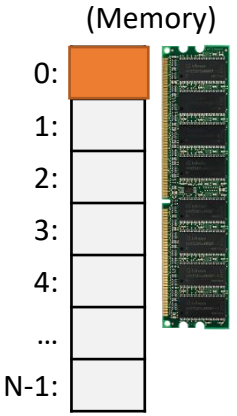




Decoding instructions.

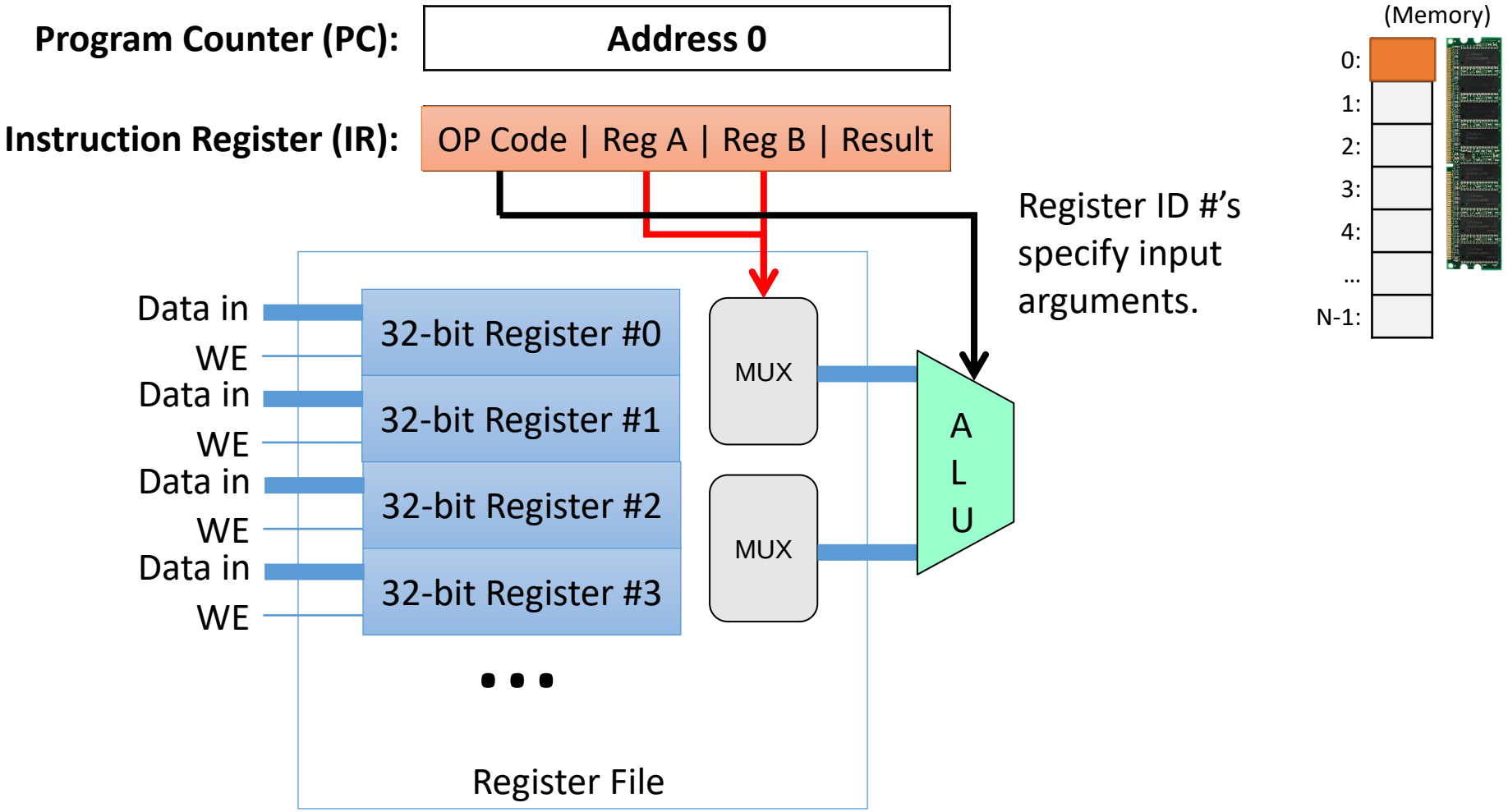
Interpret the instruction bits: What operation? Which arguments?





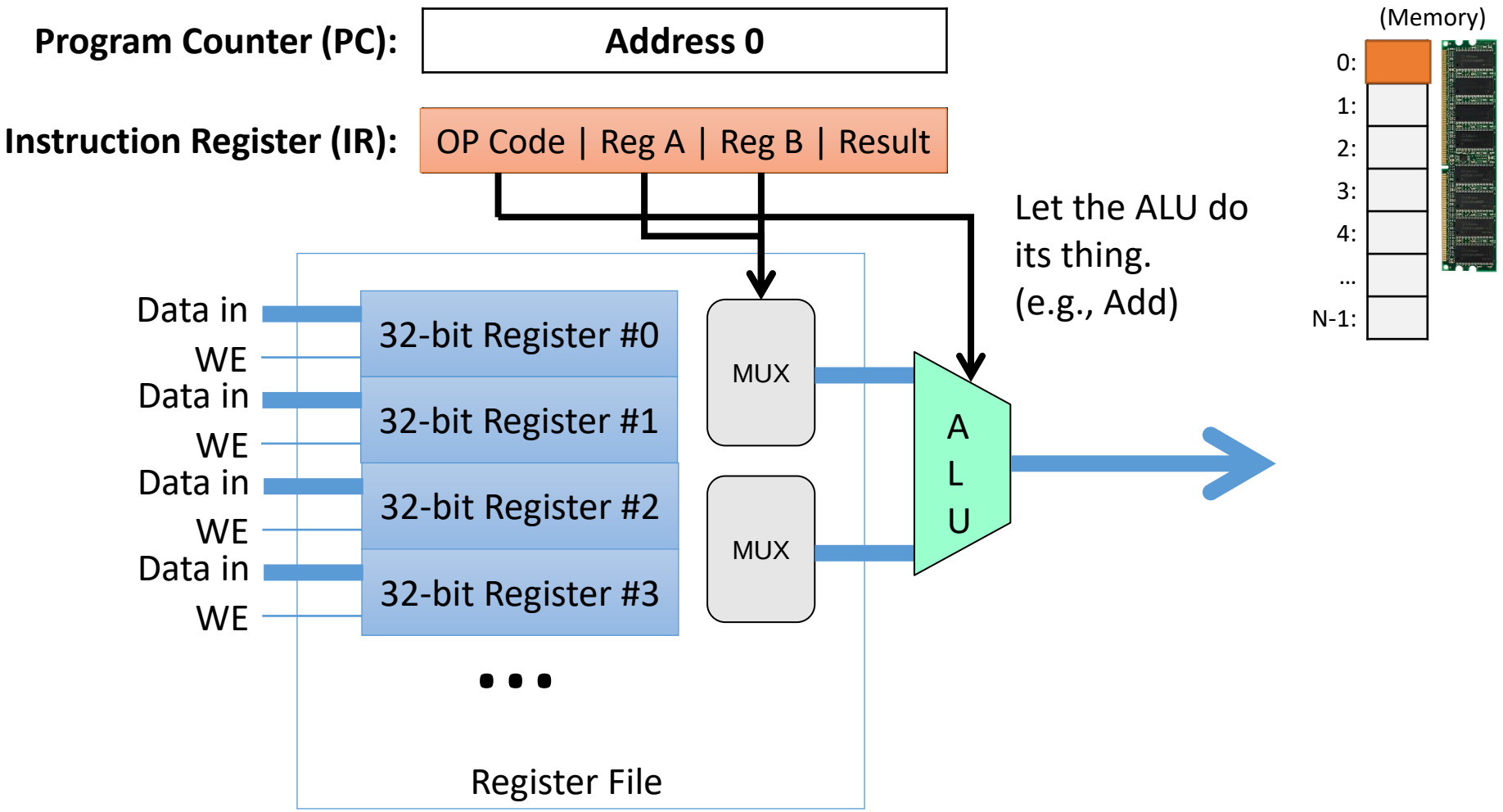
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



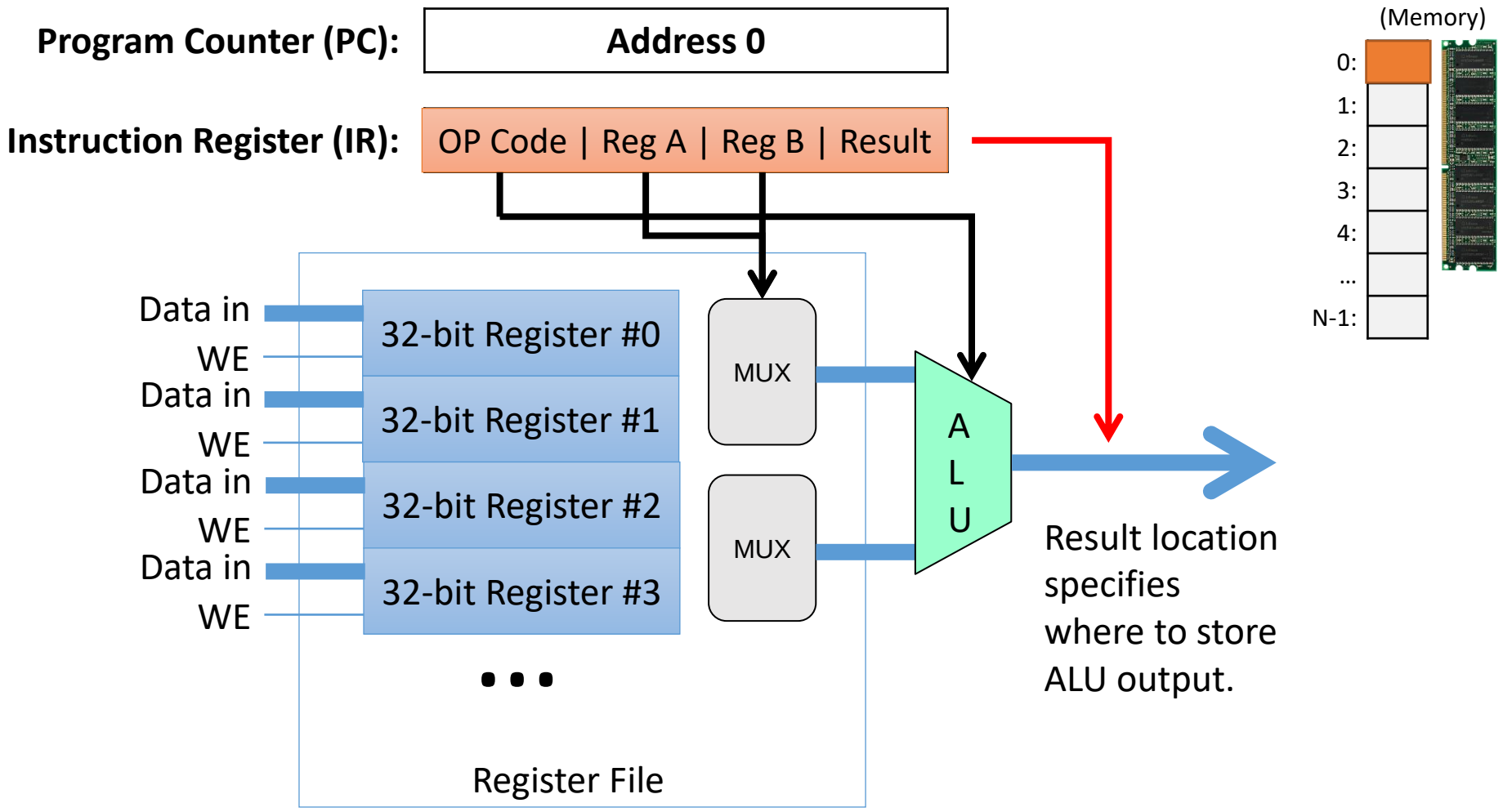
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



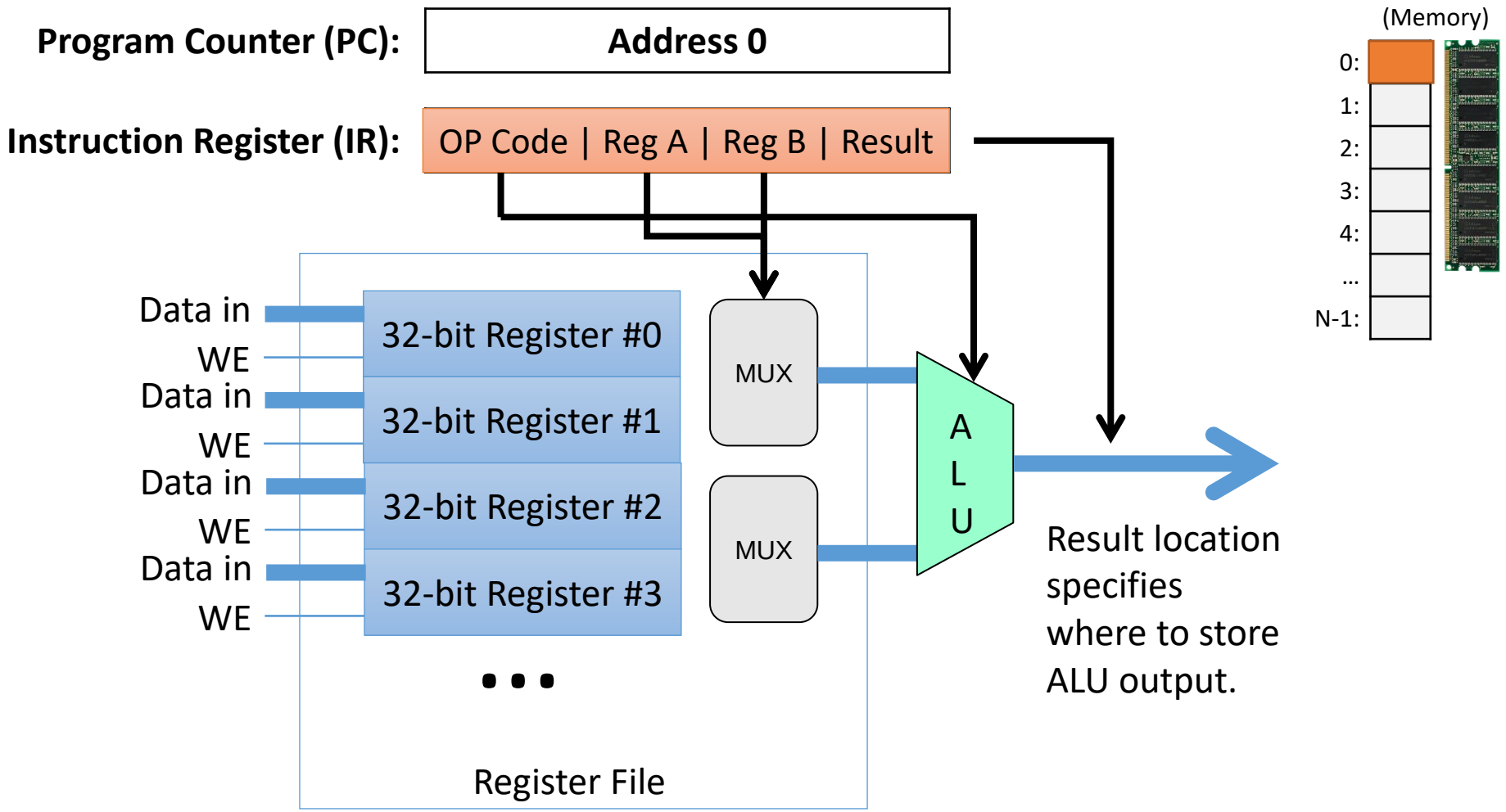
Storing results.

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



Questions so far?

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

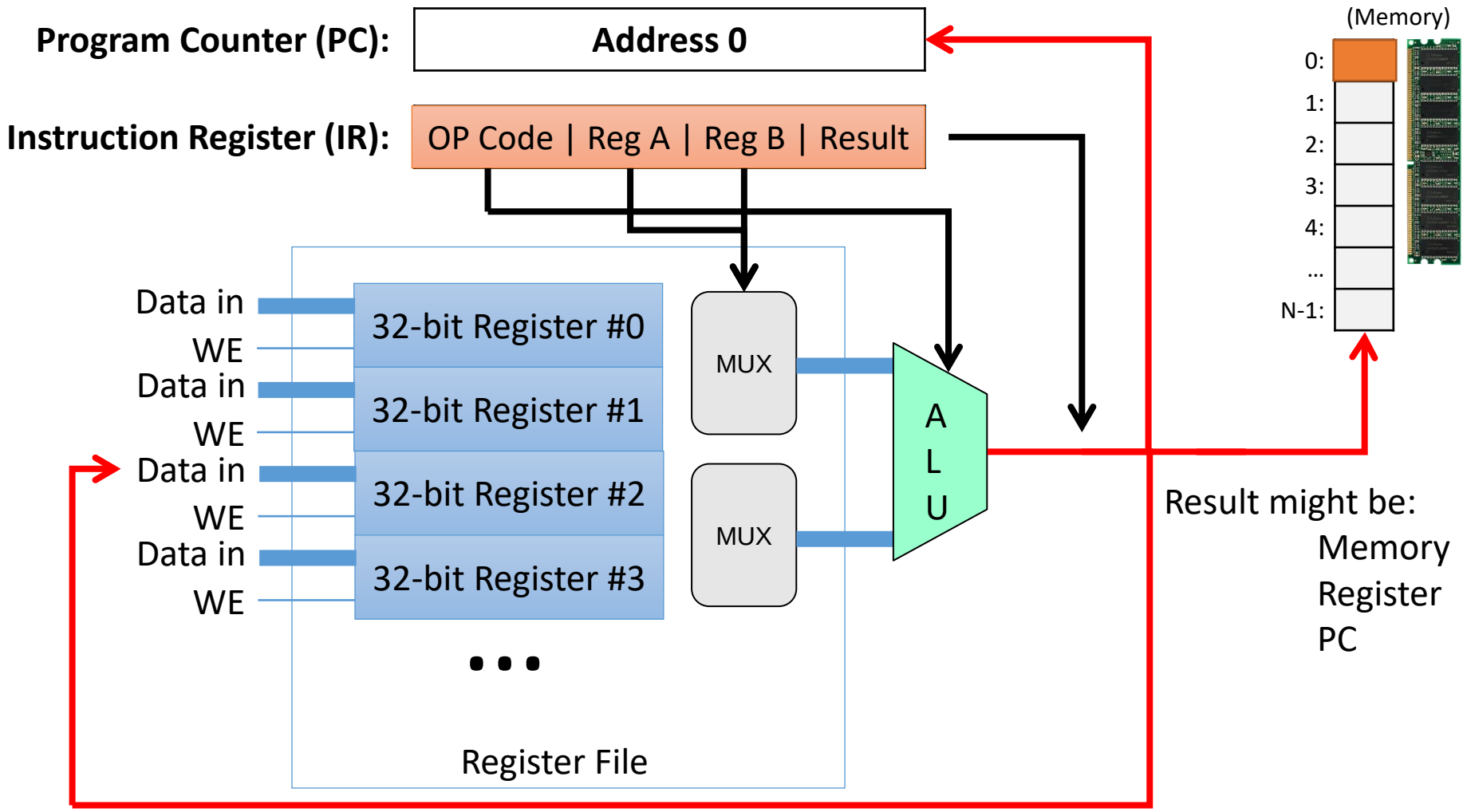


Why do we need a program counter? Can't we just start executing instruction at address 0 and count up one at a time from there?

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?

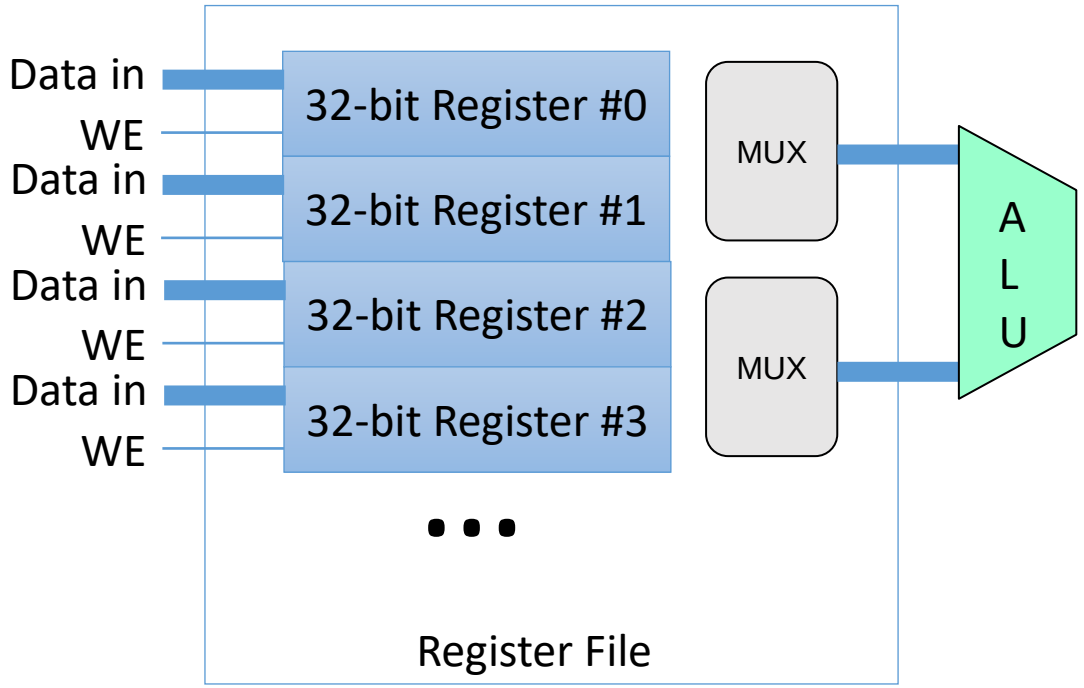


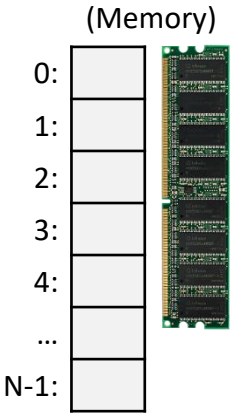
Recap CPU Model

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

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

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



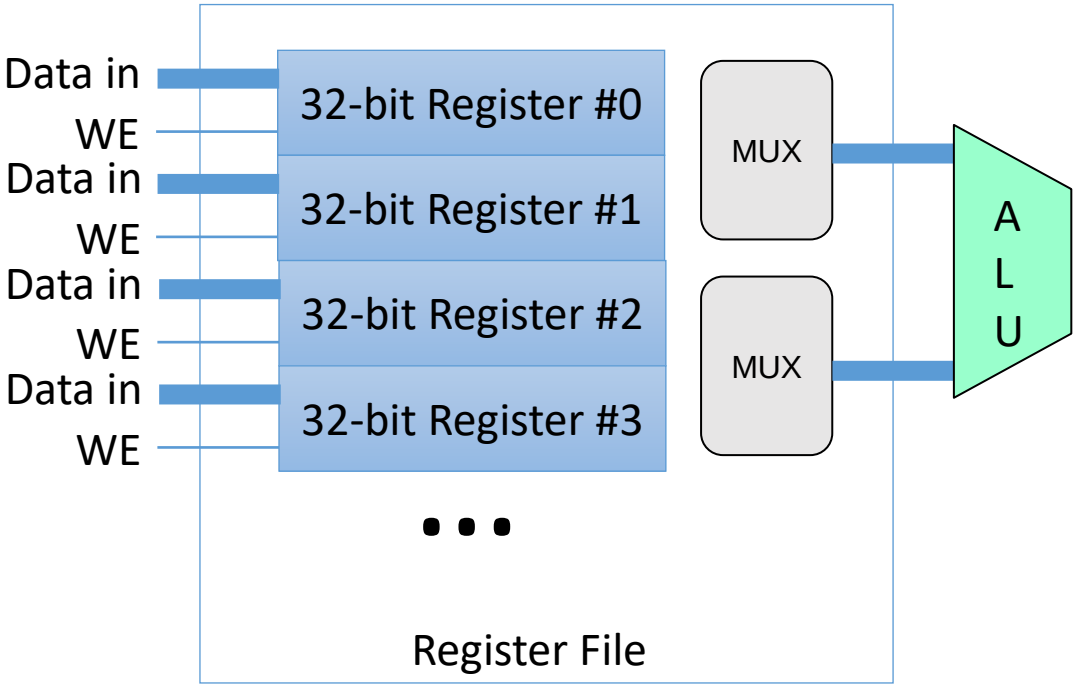


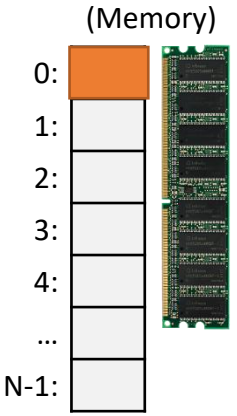
Fetching instructions.

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



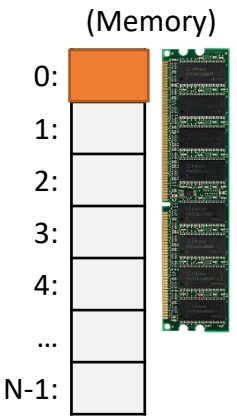


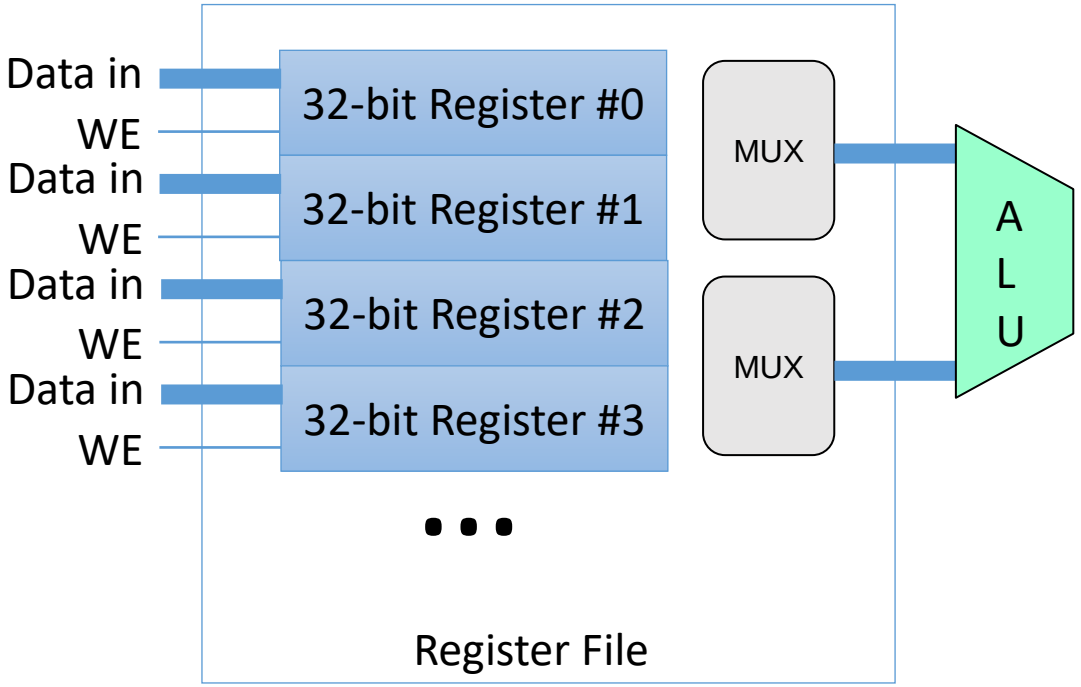
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

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

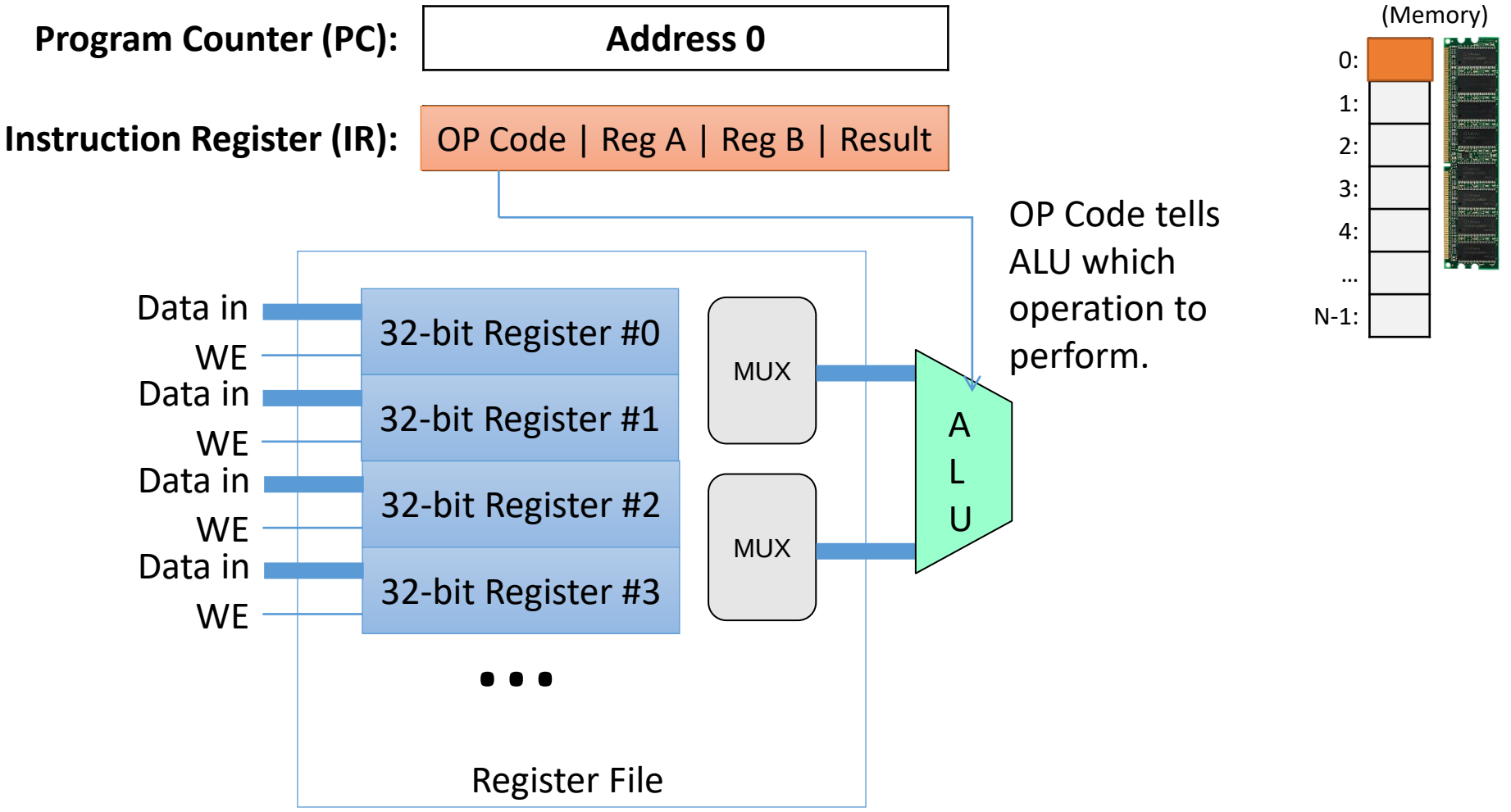
Instruction Register (IR): OP Code | Reg A | Reg B | Result





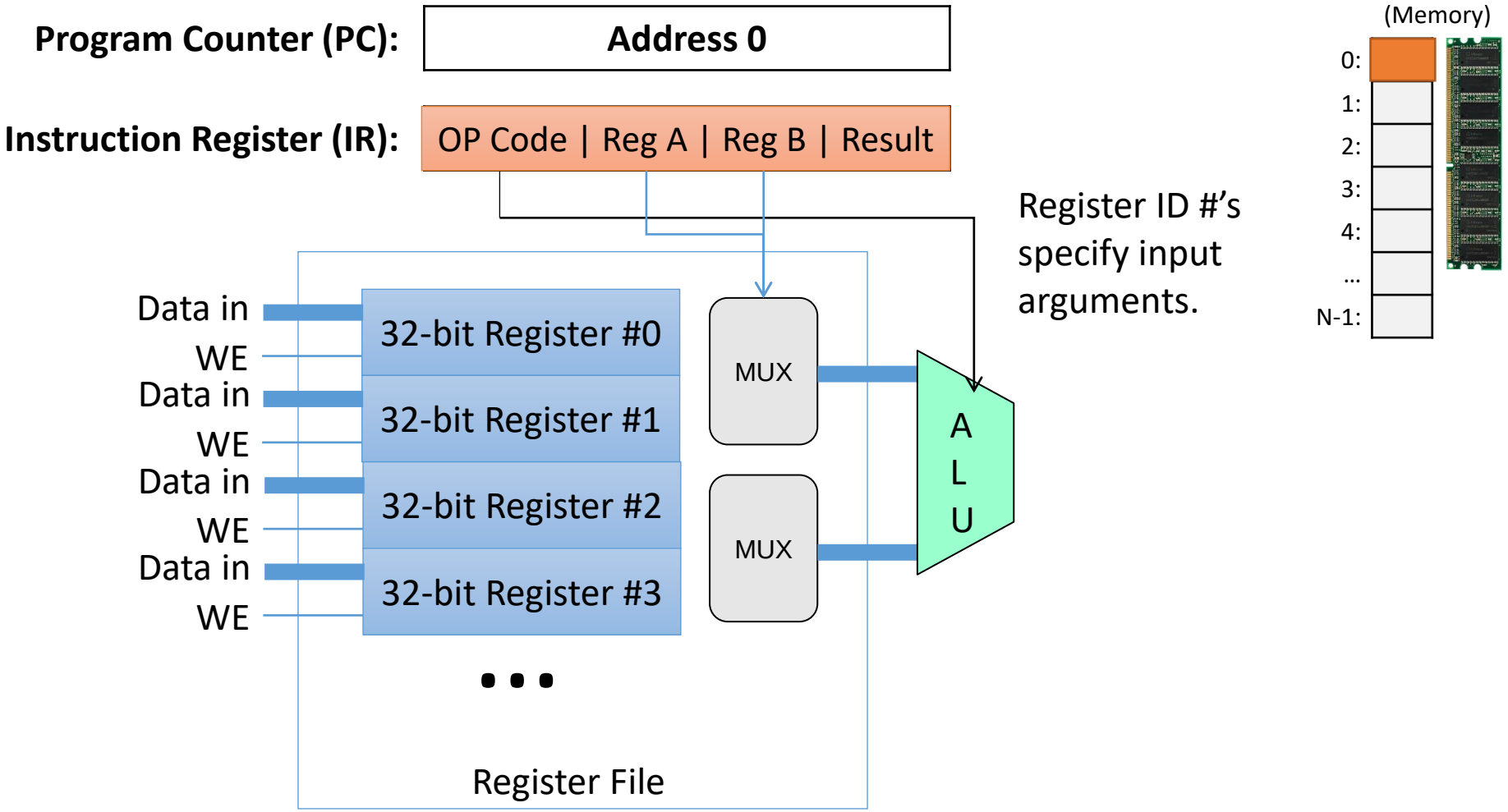
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



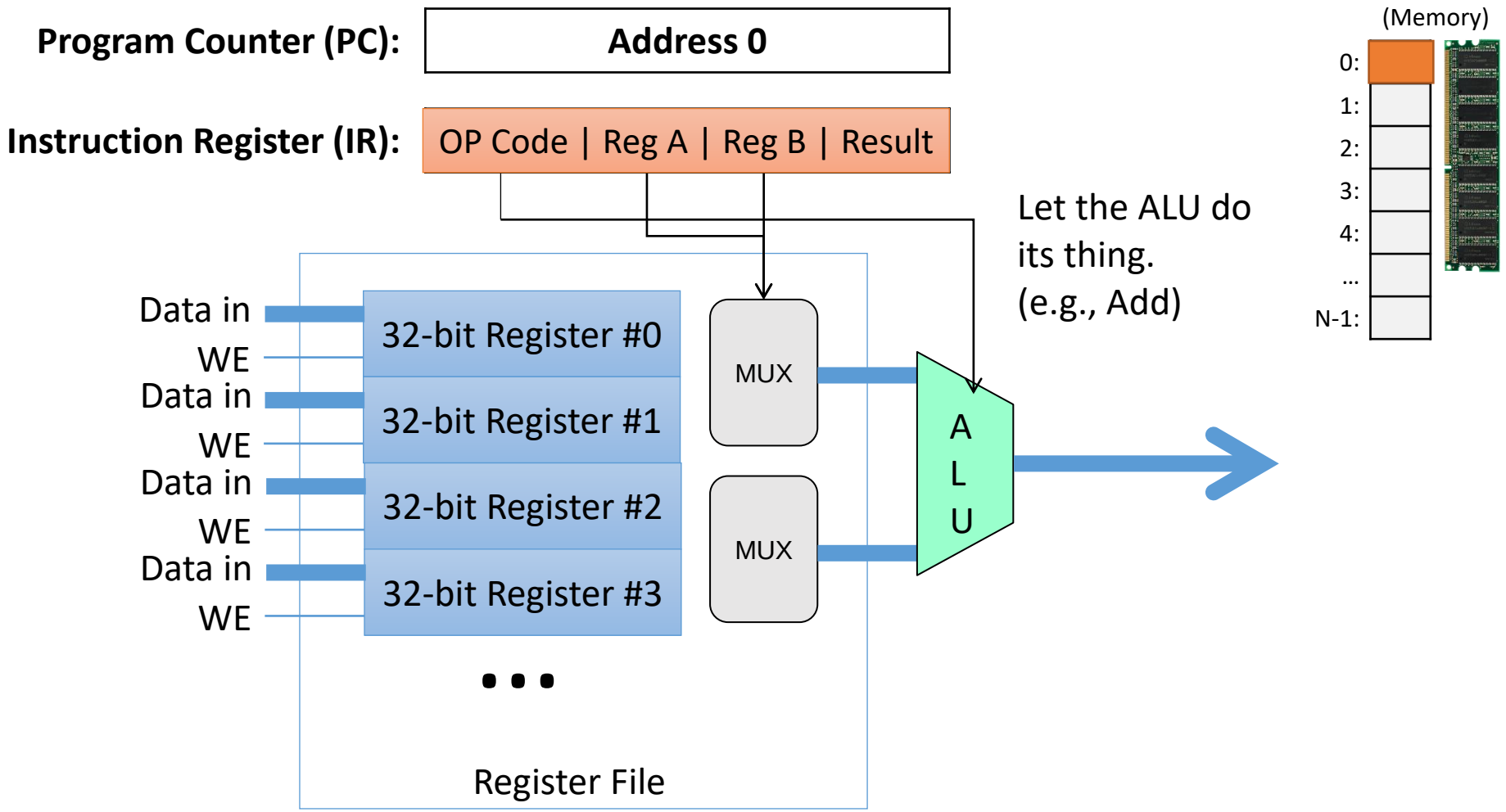
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



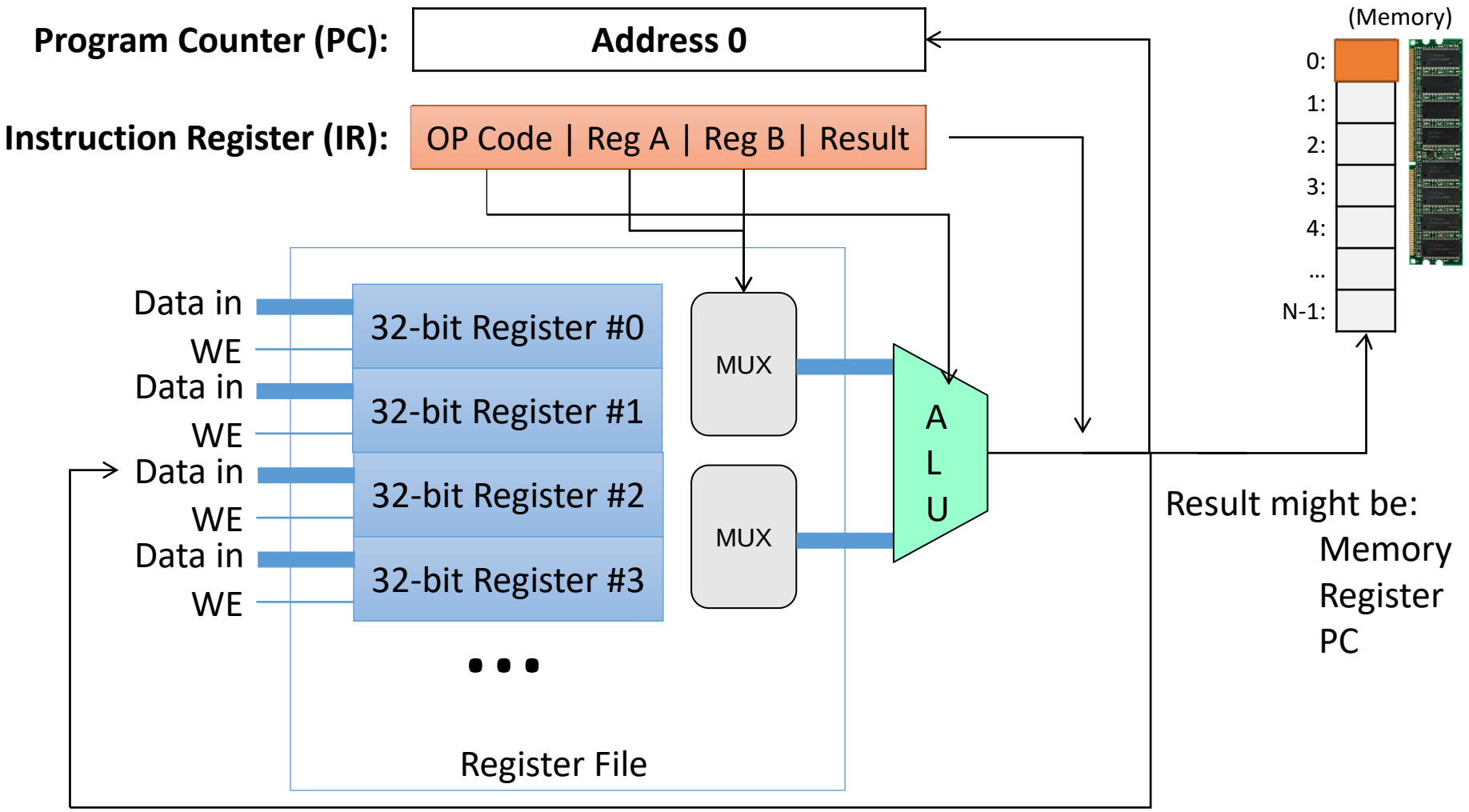
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



Storing results.

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



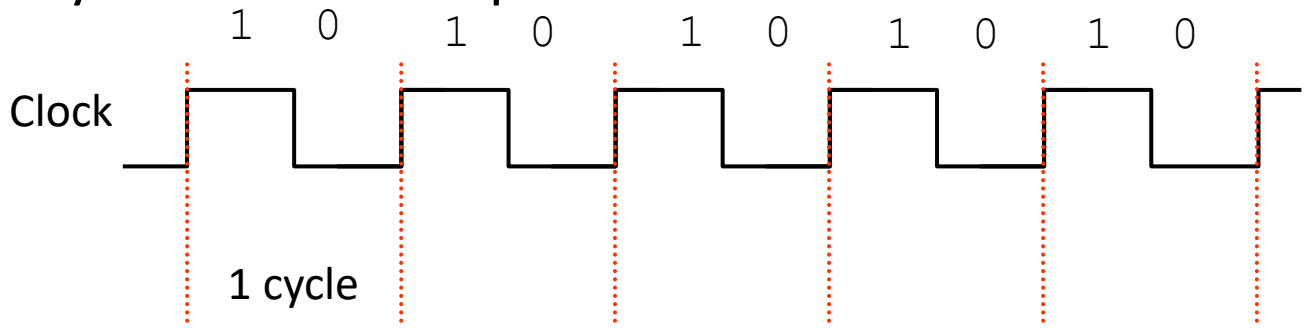
Clocking

- 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



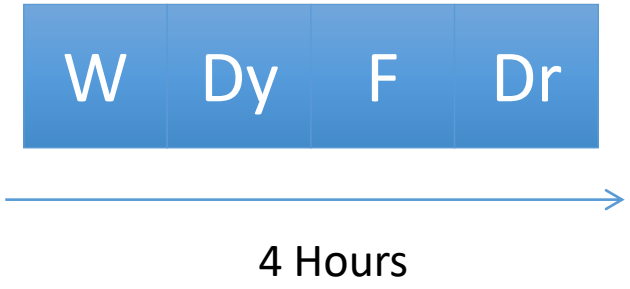
  - Clock determines how fast system runs
    - Processor can only do one thing per clock cycle
      - Usually just one part of executing an instruction
    - 1GHz processor:

      1 billion cycles/second → 1 cycle every nanosecond

Cycle Time: Laundry Analogy

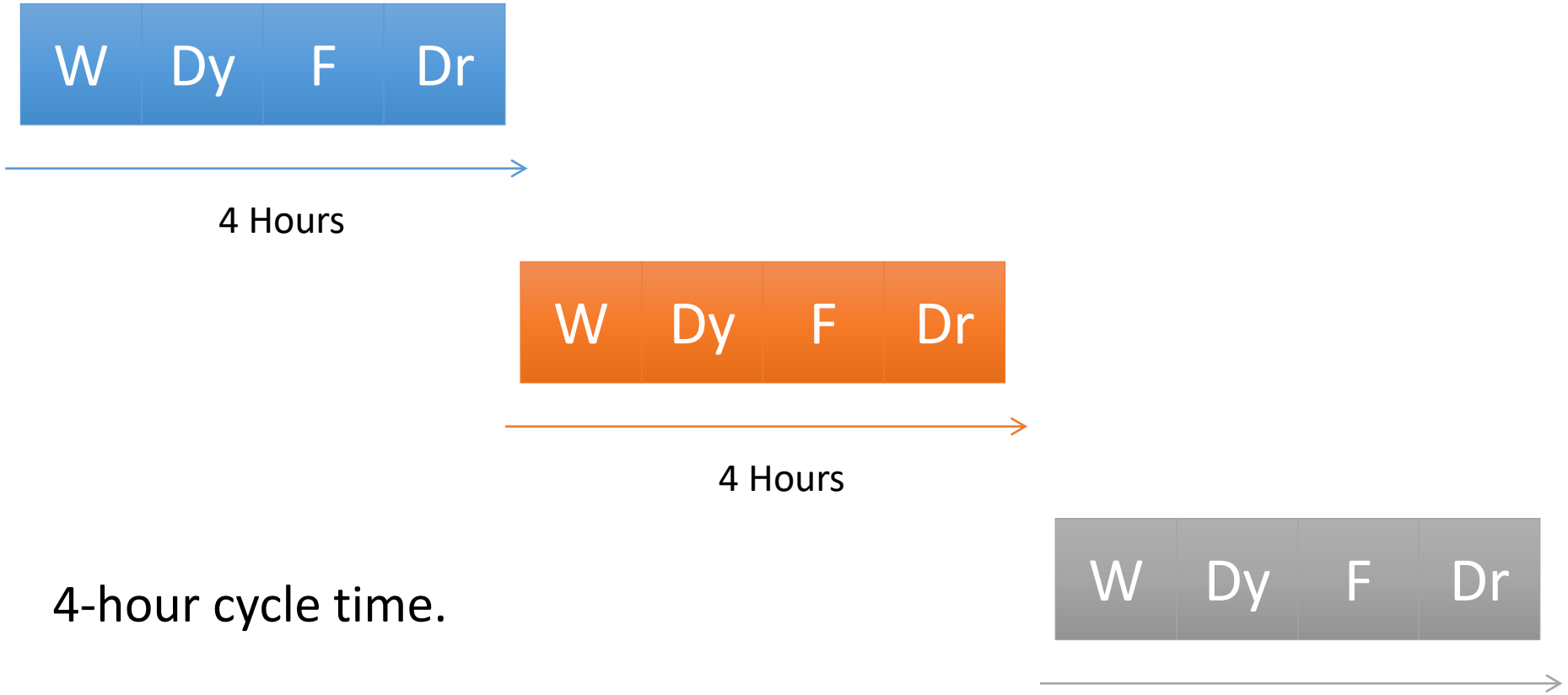
- Discrete stages: fetch, decode, execute, store

- Analogy (laundry): washer, dryer, folding, dresser



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

Laundry

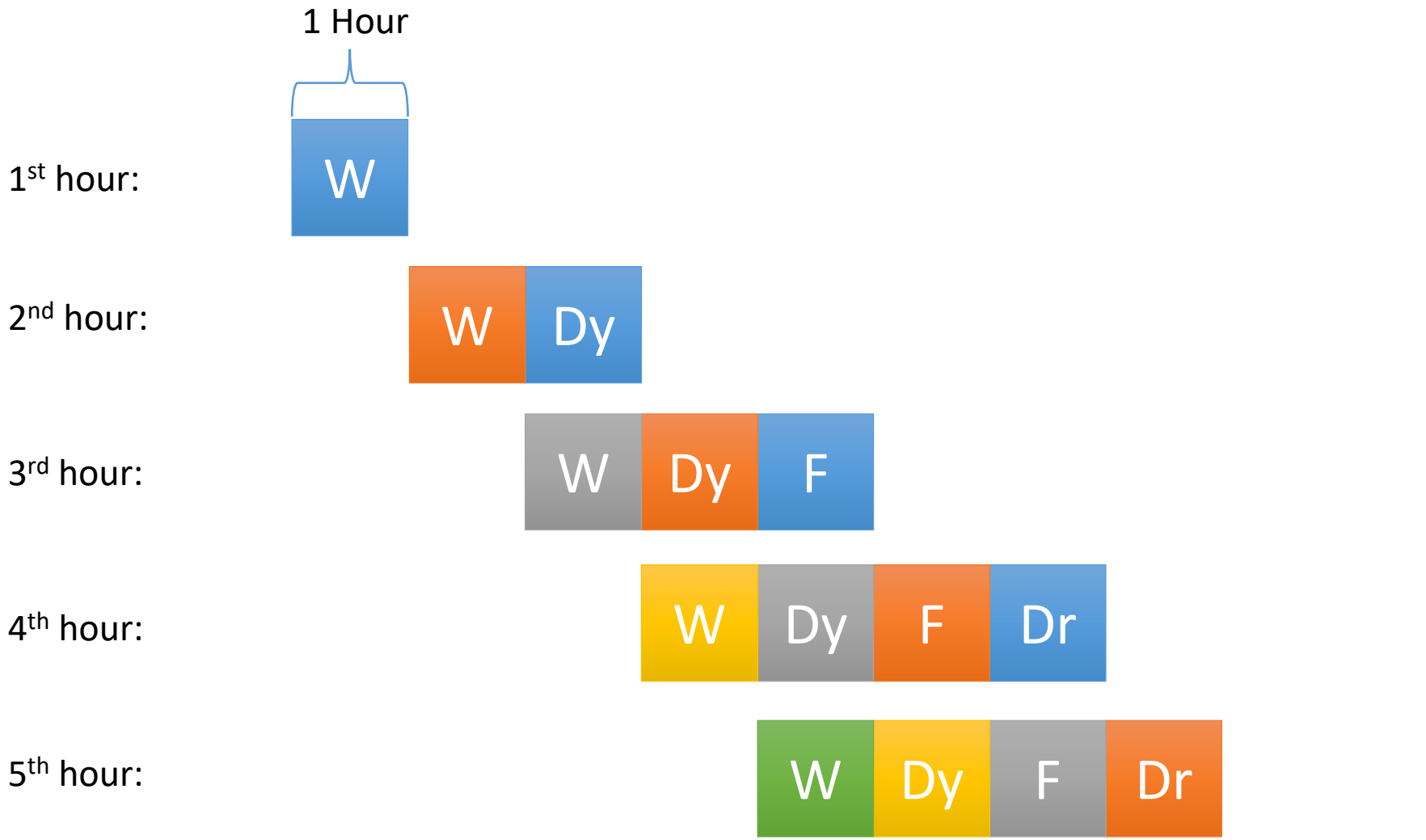


4-hour cycle time.

Finishes a laundry load every cycle.

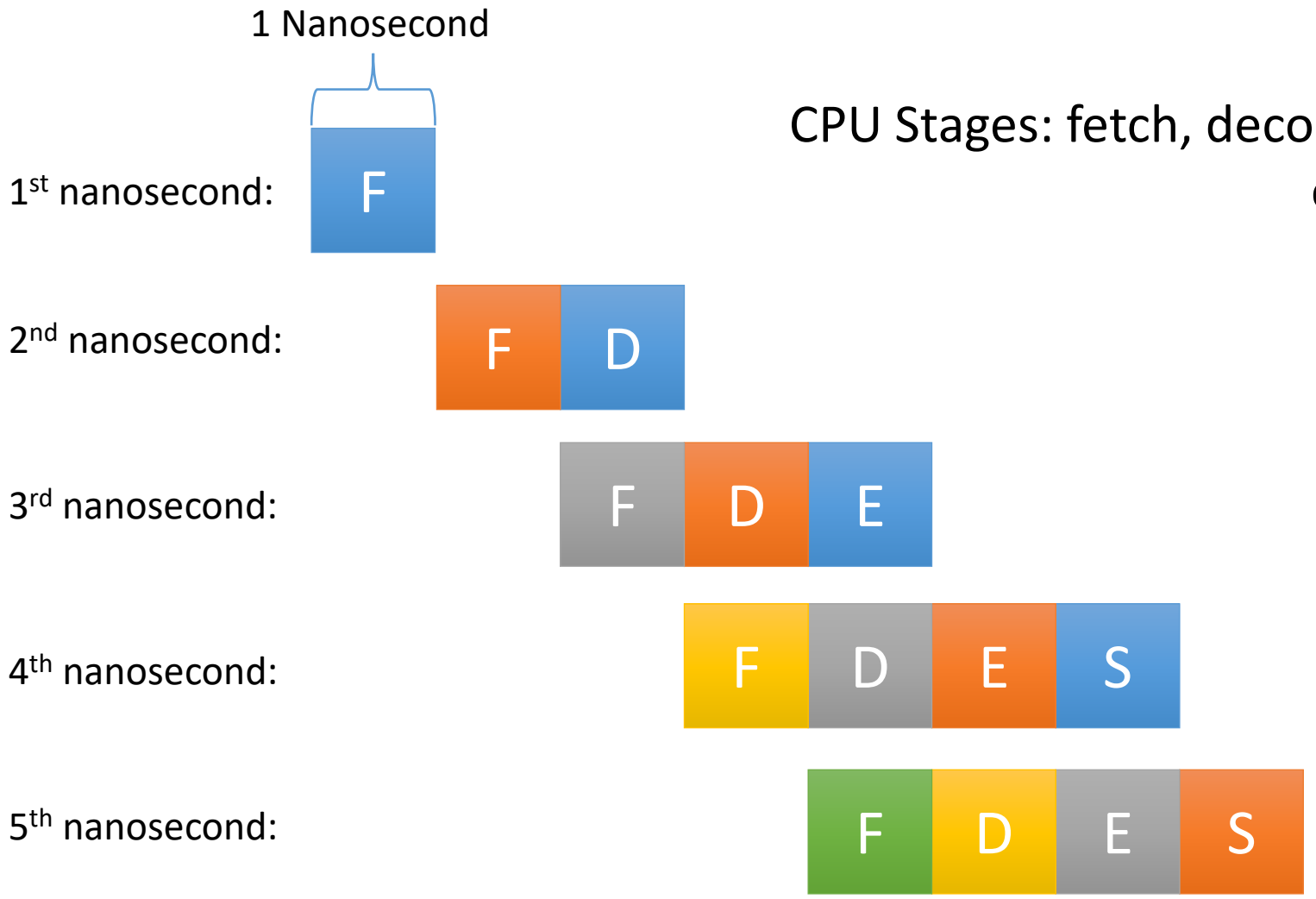
(6 laundry loads per day)

Pipelining (Laundry)



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

Pipelining (CPU)



CPU Stages: fetch, decode, execute, store results

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