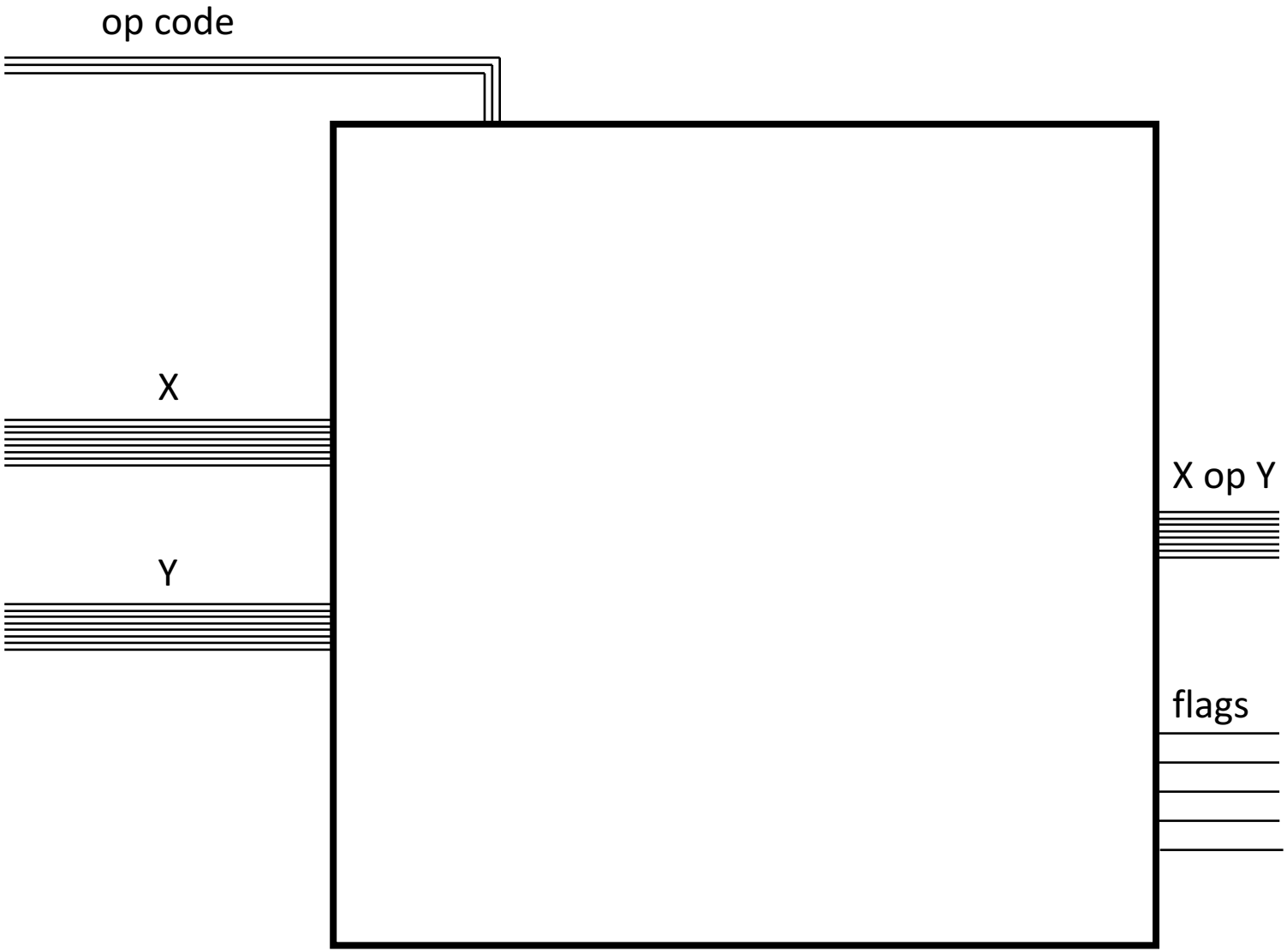
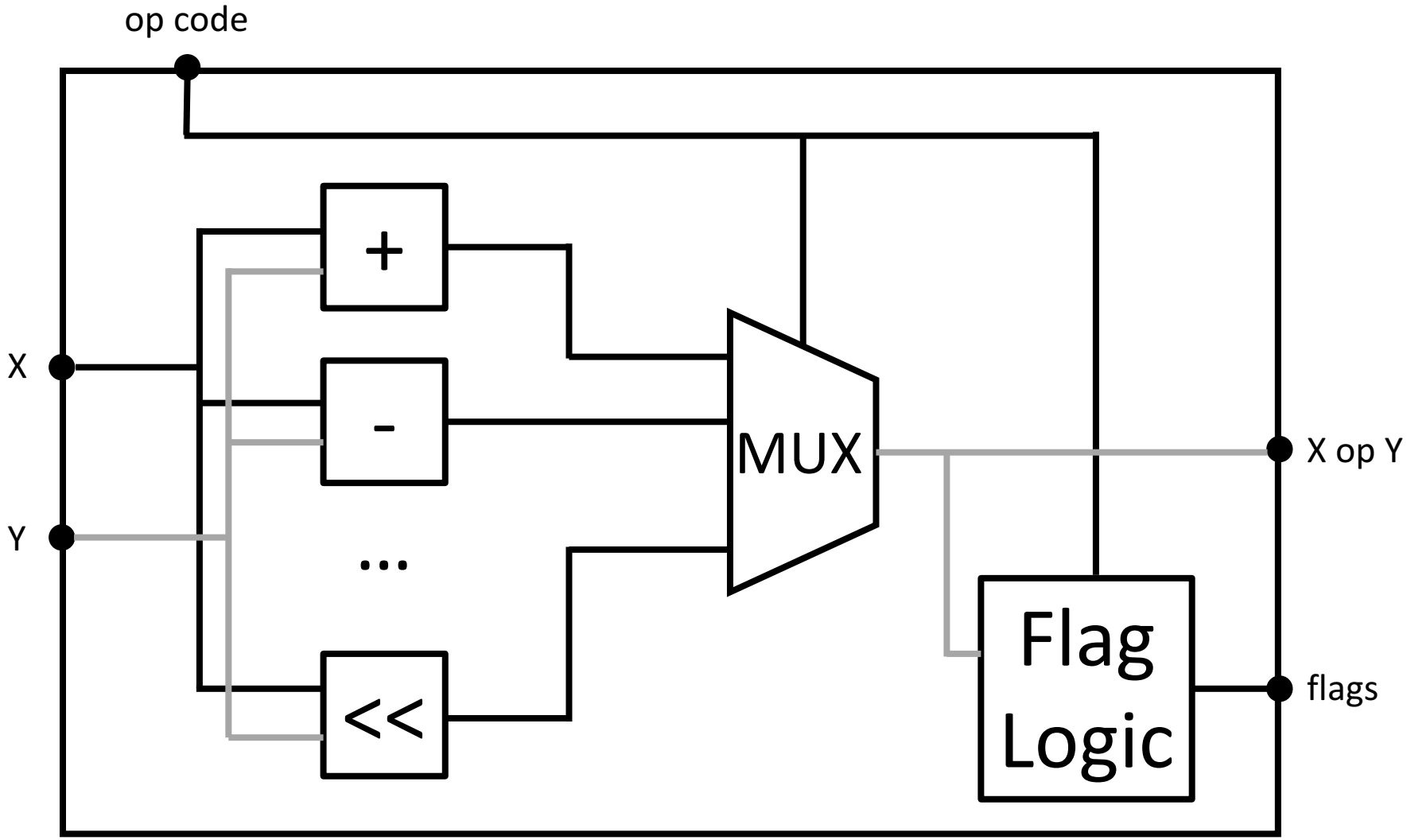
Building a CPU

9/15/16

Abstraction of your Lab 3 ALU:



Inside your Lab 3 ALU:



Circuits inside the ALU

- Arithmetic circuits
  - Generally one circuit for each possible operation.
- Control circuits
  - Select the right output
  - Set appropriate flags

Circuits around the ALU

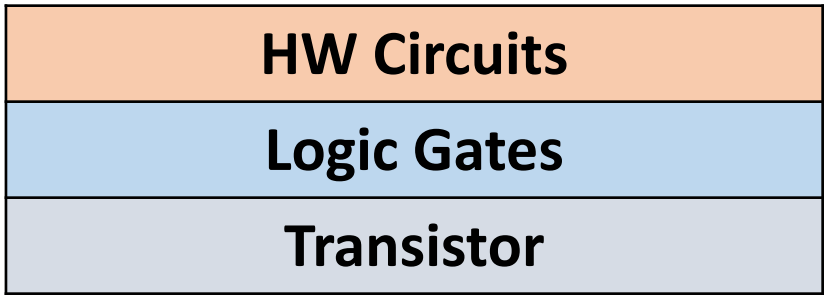
- Where do the inputs come from?
  - X, Y
  - opcode
- Where do the outputs go?
  - X op Y
  - Flags

Recall from last time…

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

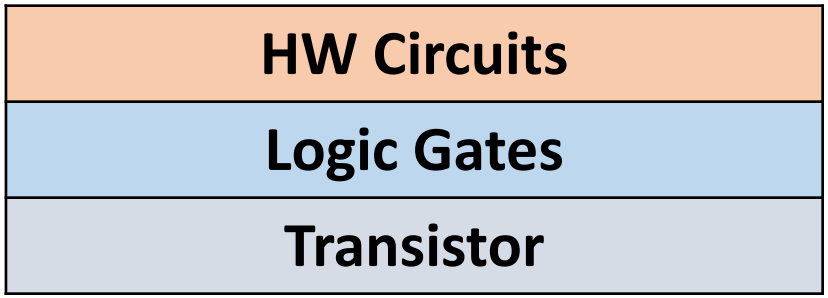


Recall from last time…

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.



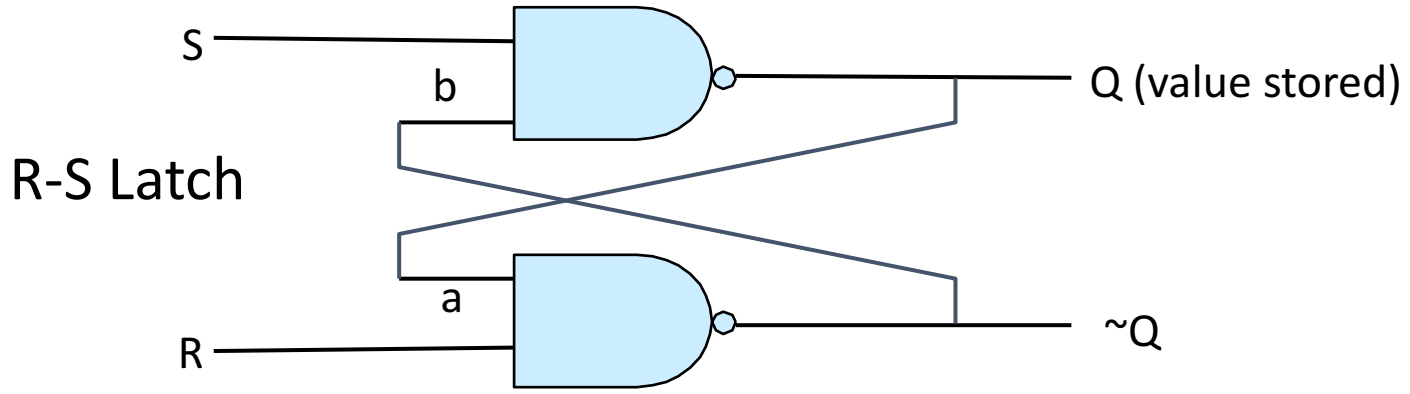
Memory Circuits: Starting Small

- 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 an S are both 1: Store a value

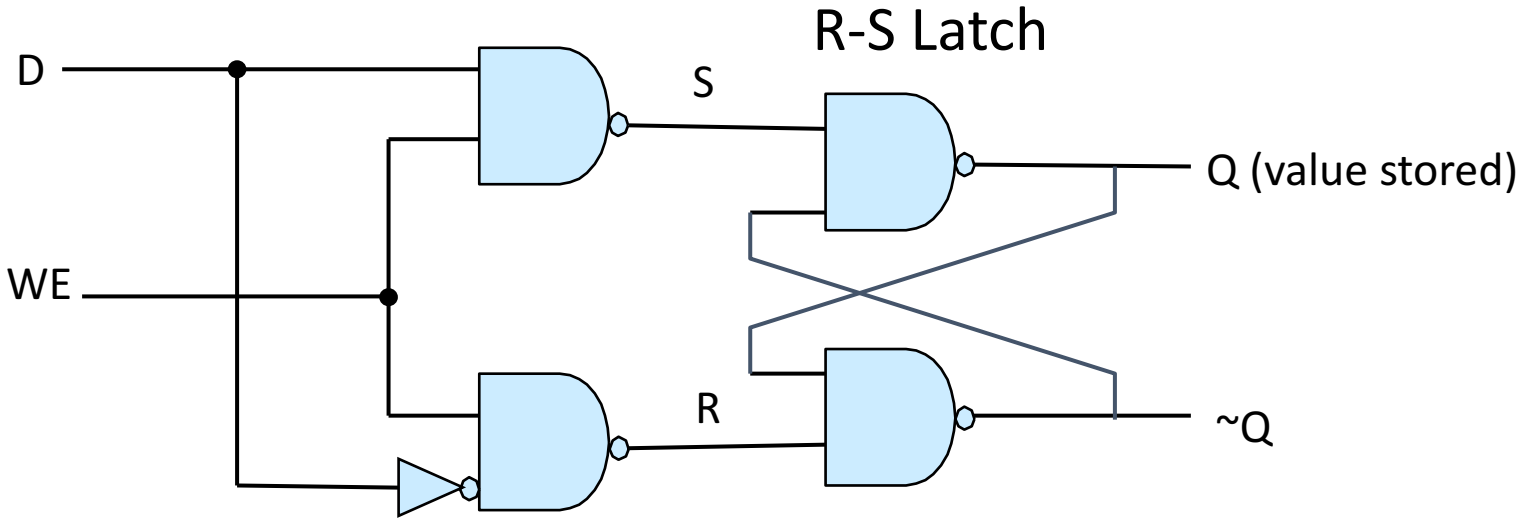
R and S are never both simultaneously 0



- 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

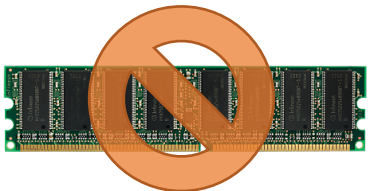
Controls S-R latch writing, ensures S & R never both 0



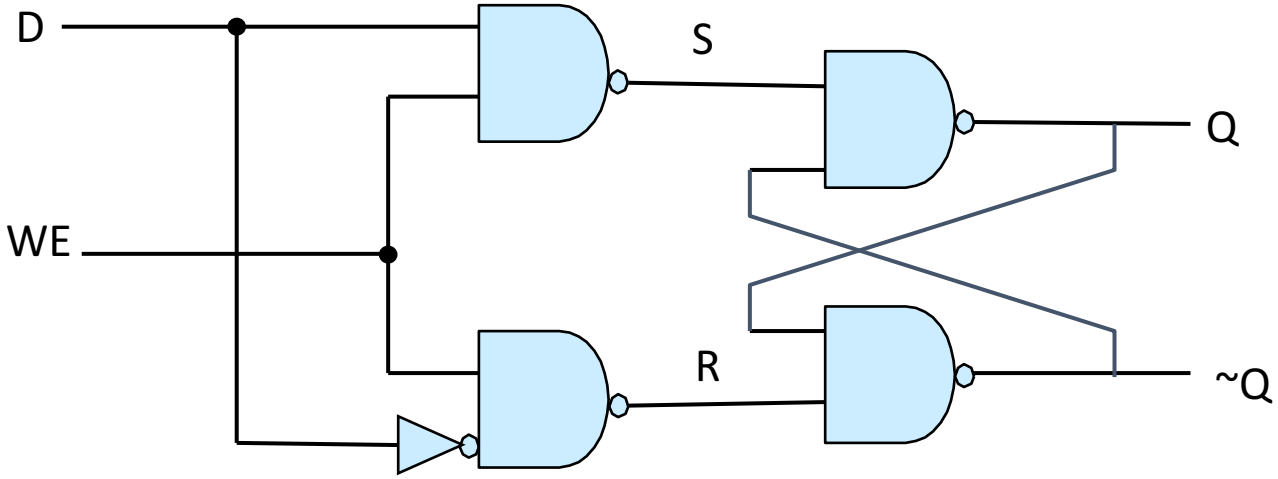
D: data we want to store
WE: write-enable: allow data to be stored

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

DRAM: capacitor-based:



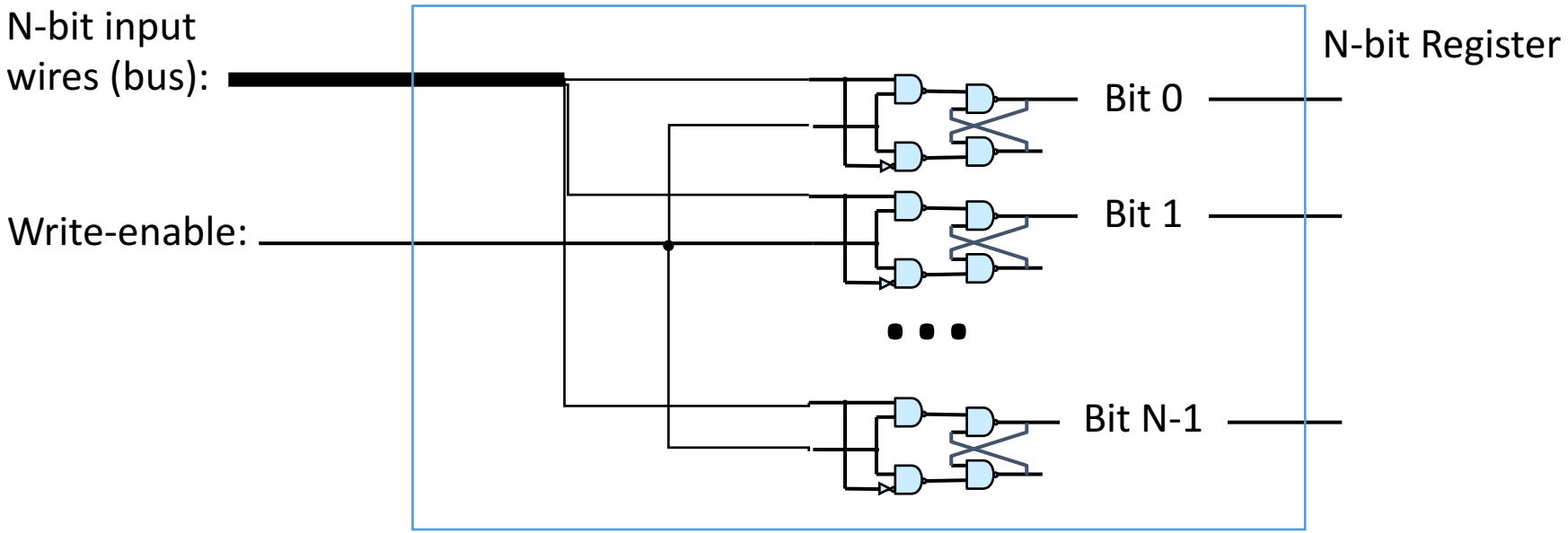
What gets stored when WE=1?



A. Q = 0
B. Q = 1
C. Q = D
D. Q = ~D
E. Something else.

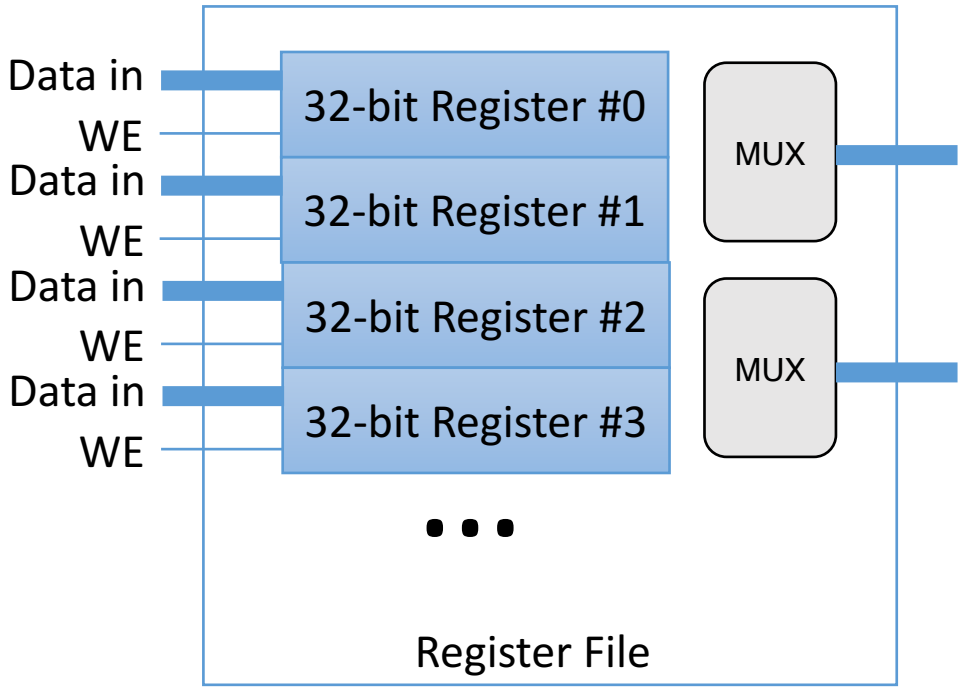
Registers

- 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.
- You (the programmer) can directly interact with the register file.
- 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.

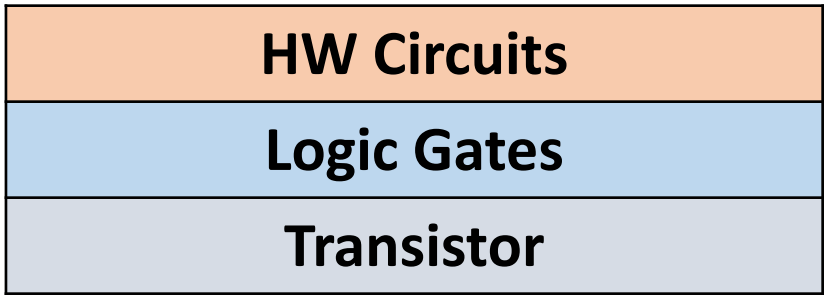
- Register file's simple interface:
  - Read R_x's value, use for calculation
  - Write R_y's value to store result

Recall again…

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



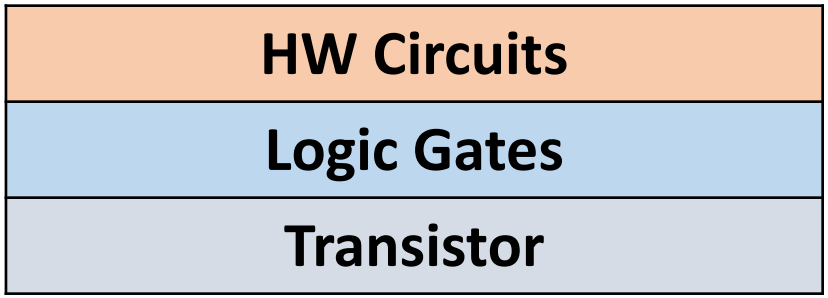
Recall again…

Three main classifications of HW circuits:

Keep track of where we are in the program.

Execute instruction, move to next.

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



CPU so far…

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

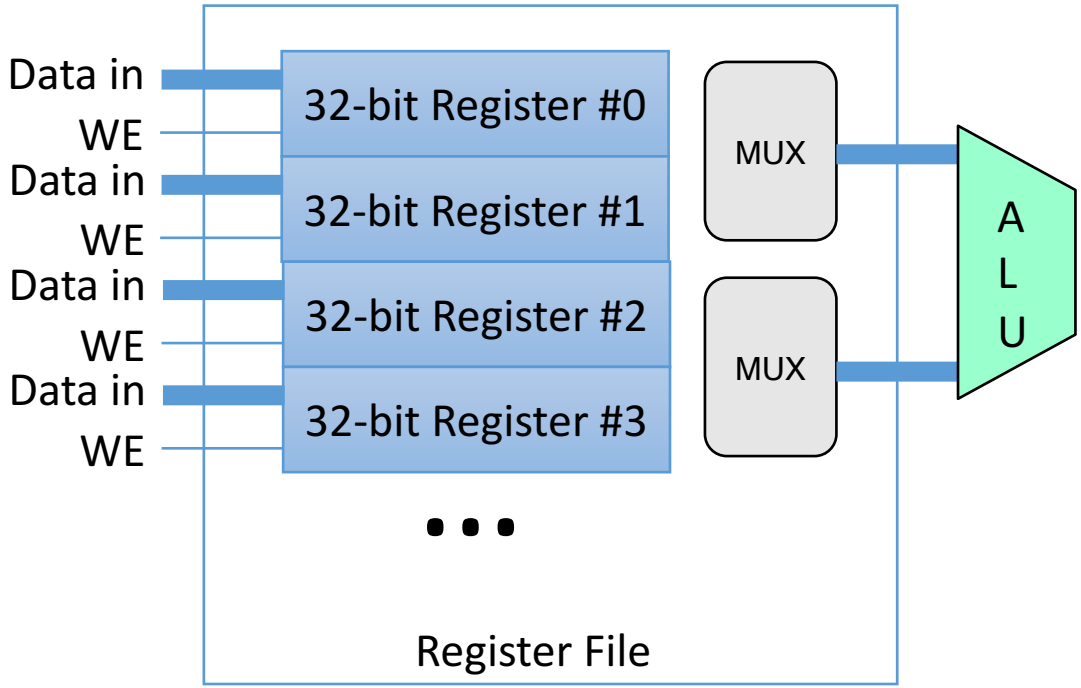
We know how to perform arithmetic on it, by feeding it to ALU.

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?

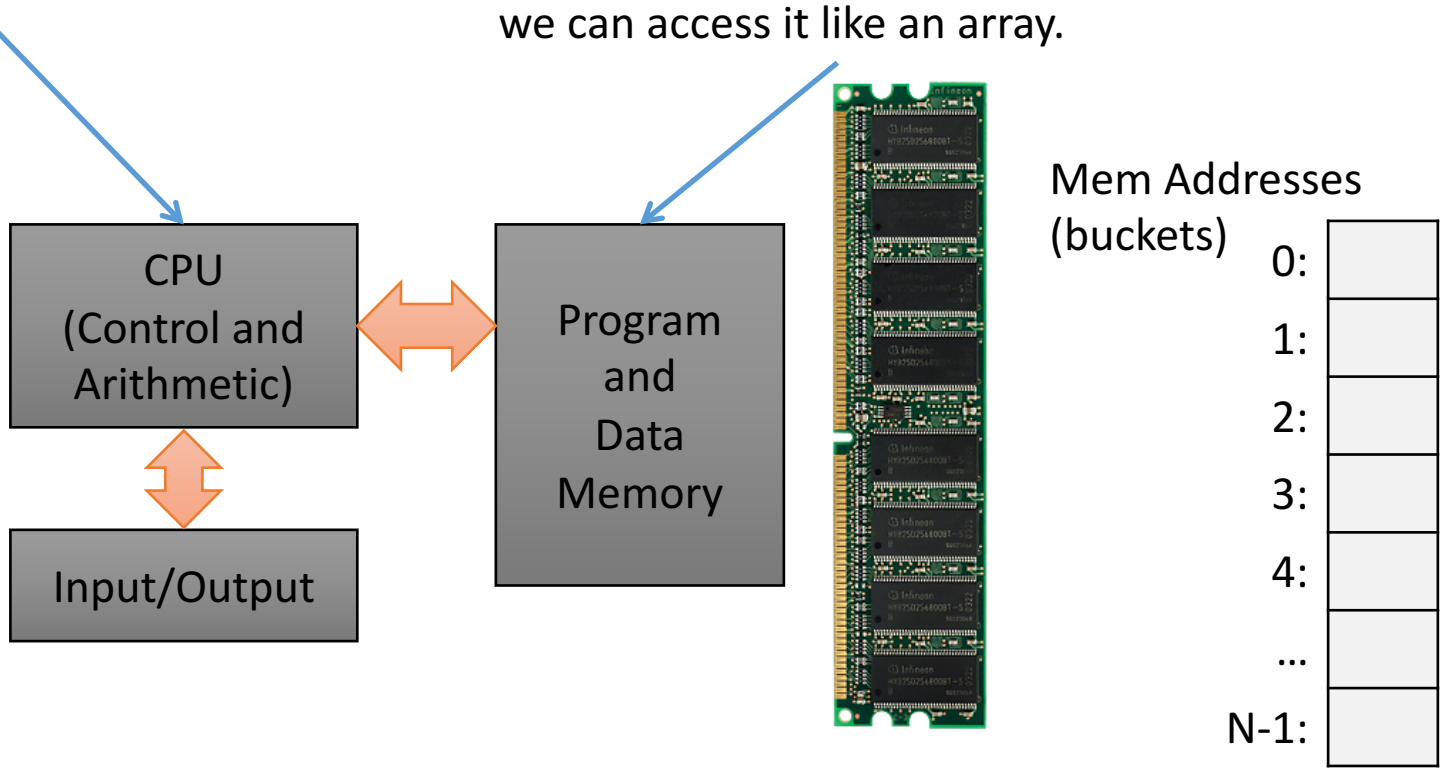


All this info comes from our compiled program: a series of instructions.

Recall: Von Neumann Model

We're building this.

Our program (instructions) live here. We'll assume for now that we can access it like an array.



CPU Game Plan

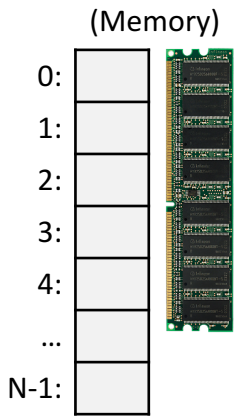
- Fetch instruction from memory
- Decode what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands
- Execute the instruction (arithmetic, etc.)
- Store the result

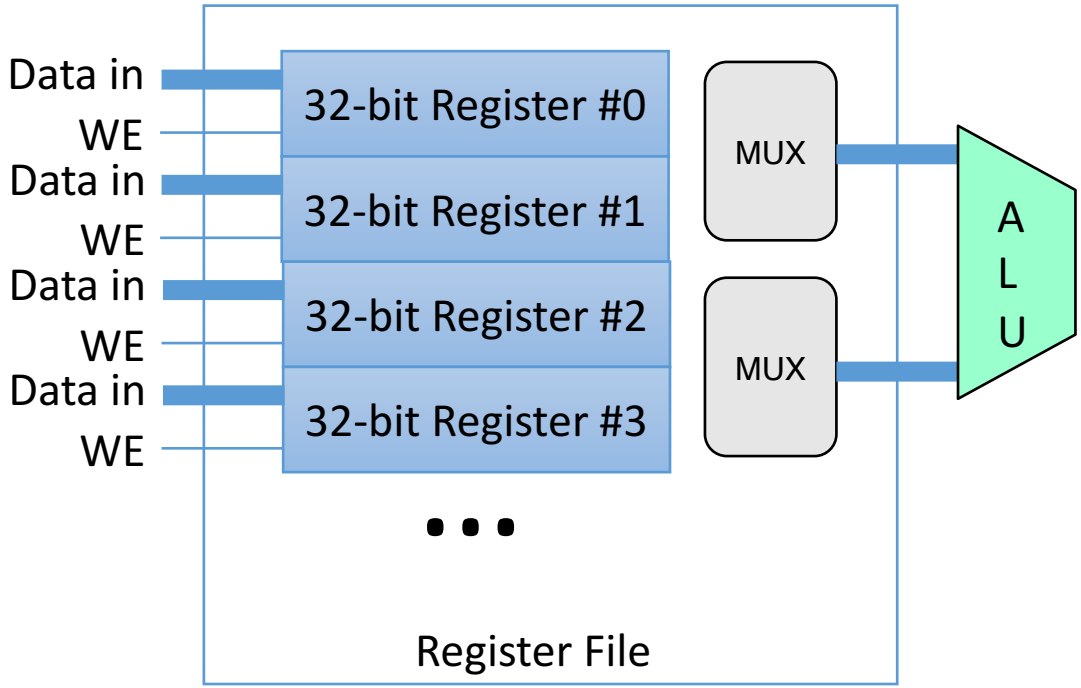
Program State

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)**



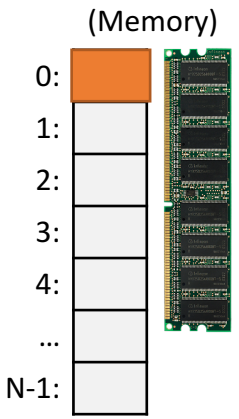


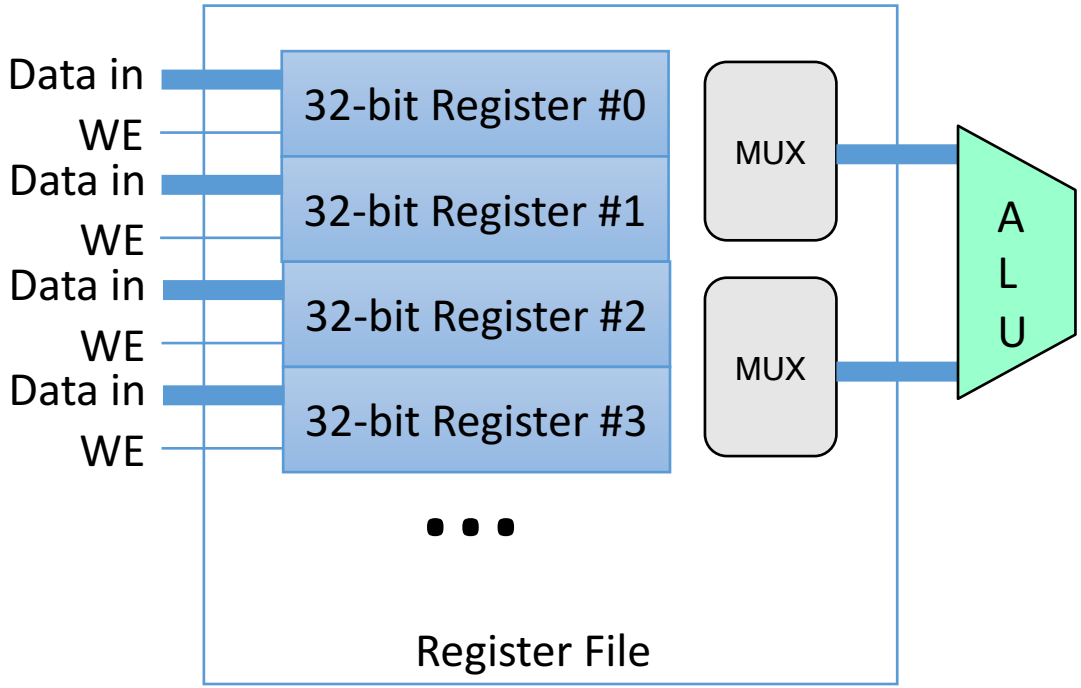
Fetching instructions.

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



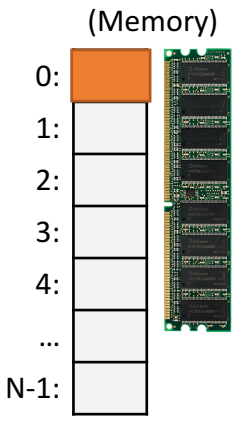


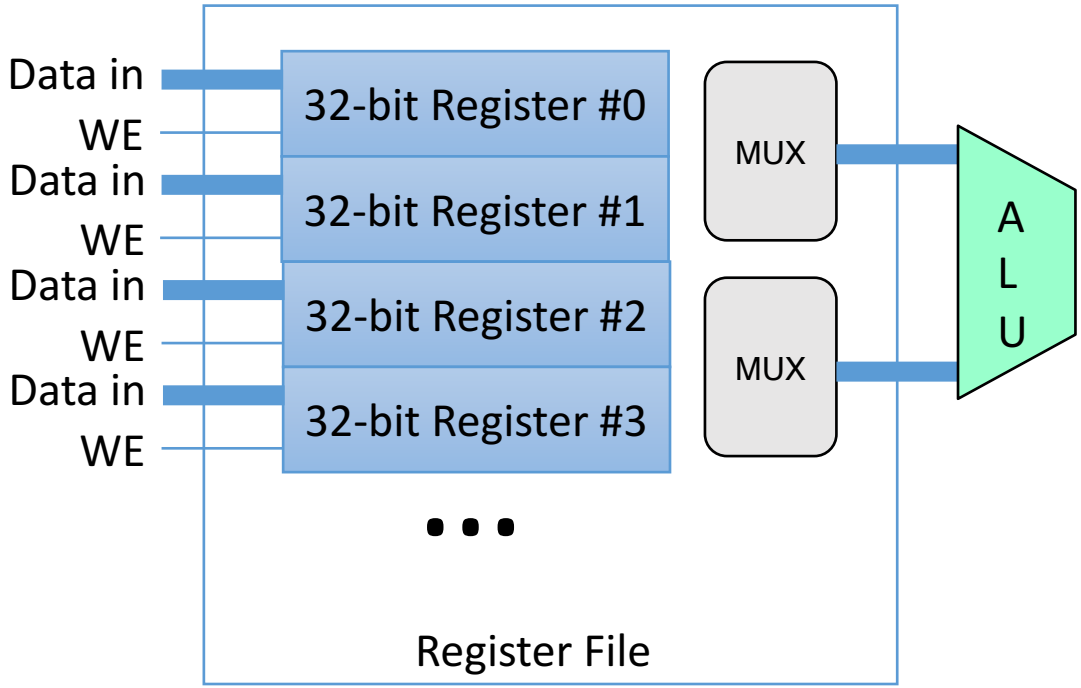
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

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

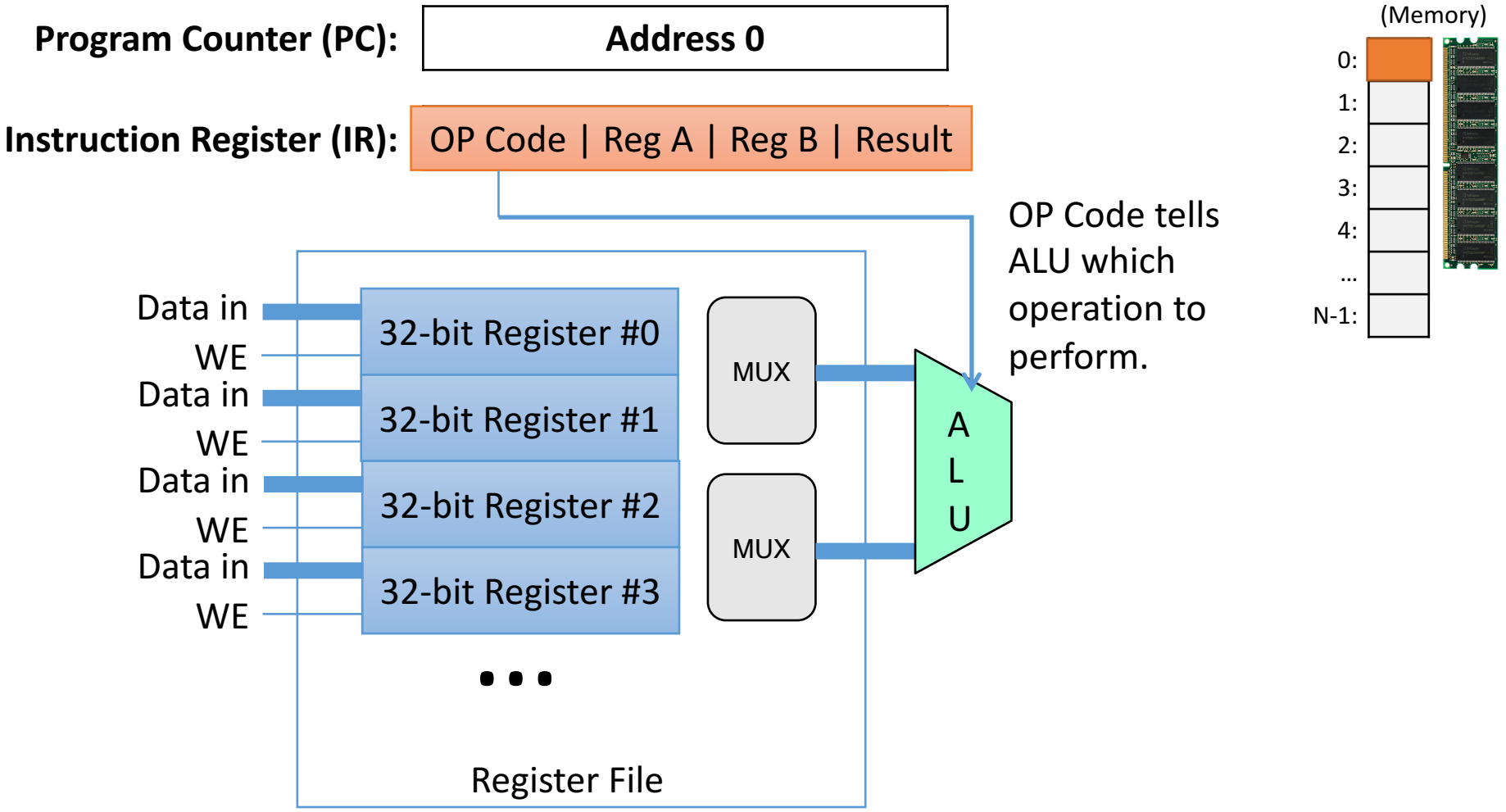
Instruction Register (IR): OP Code | Reg A | Reg B | Result





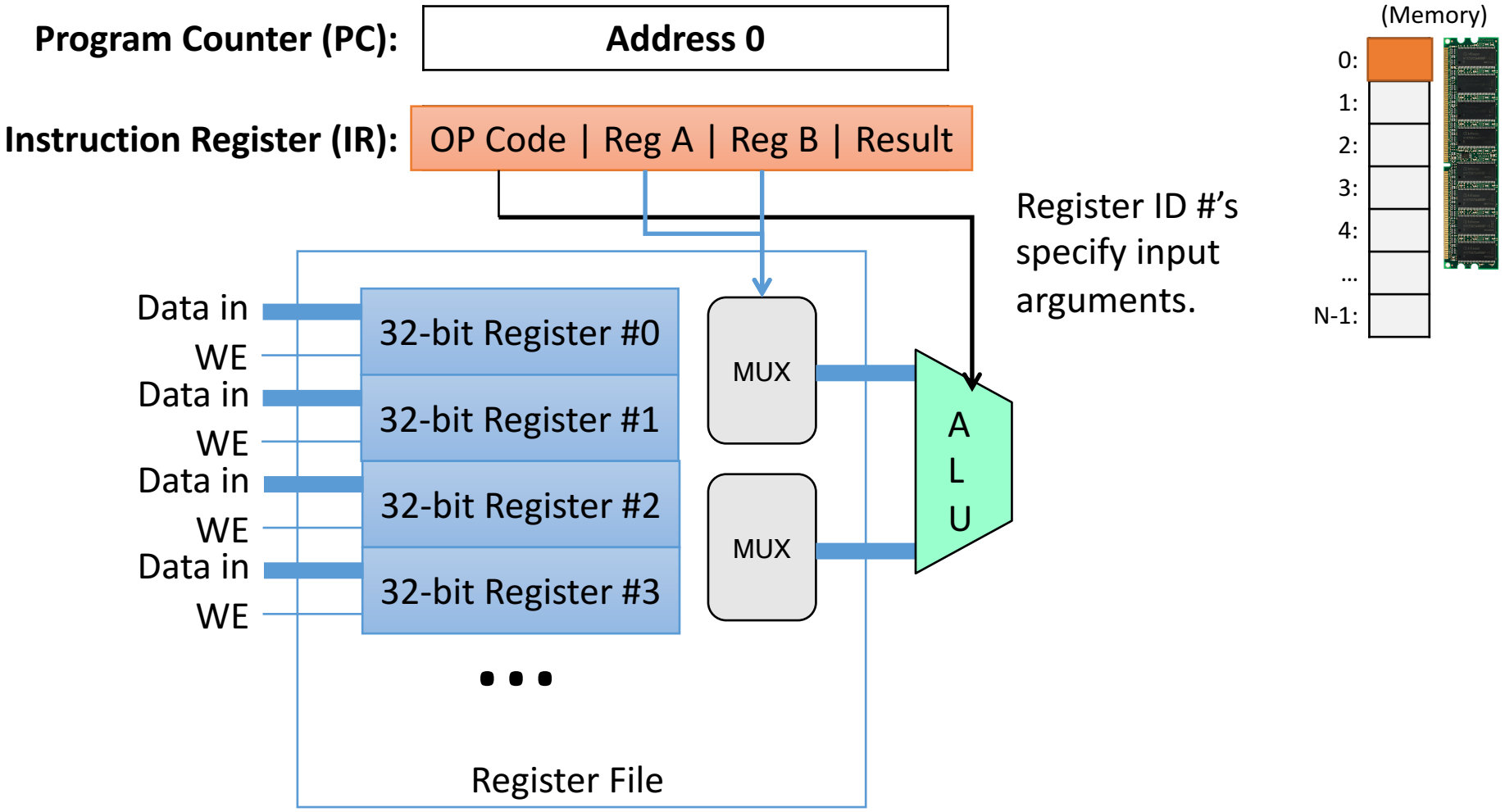
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



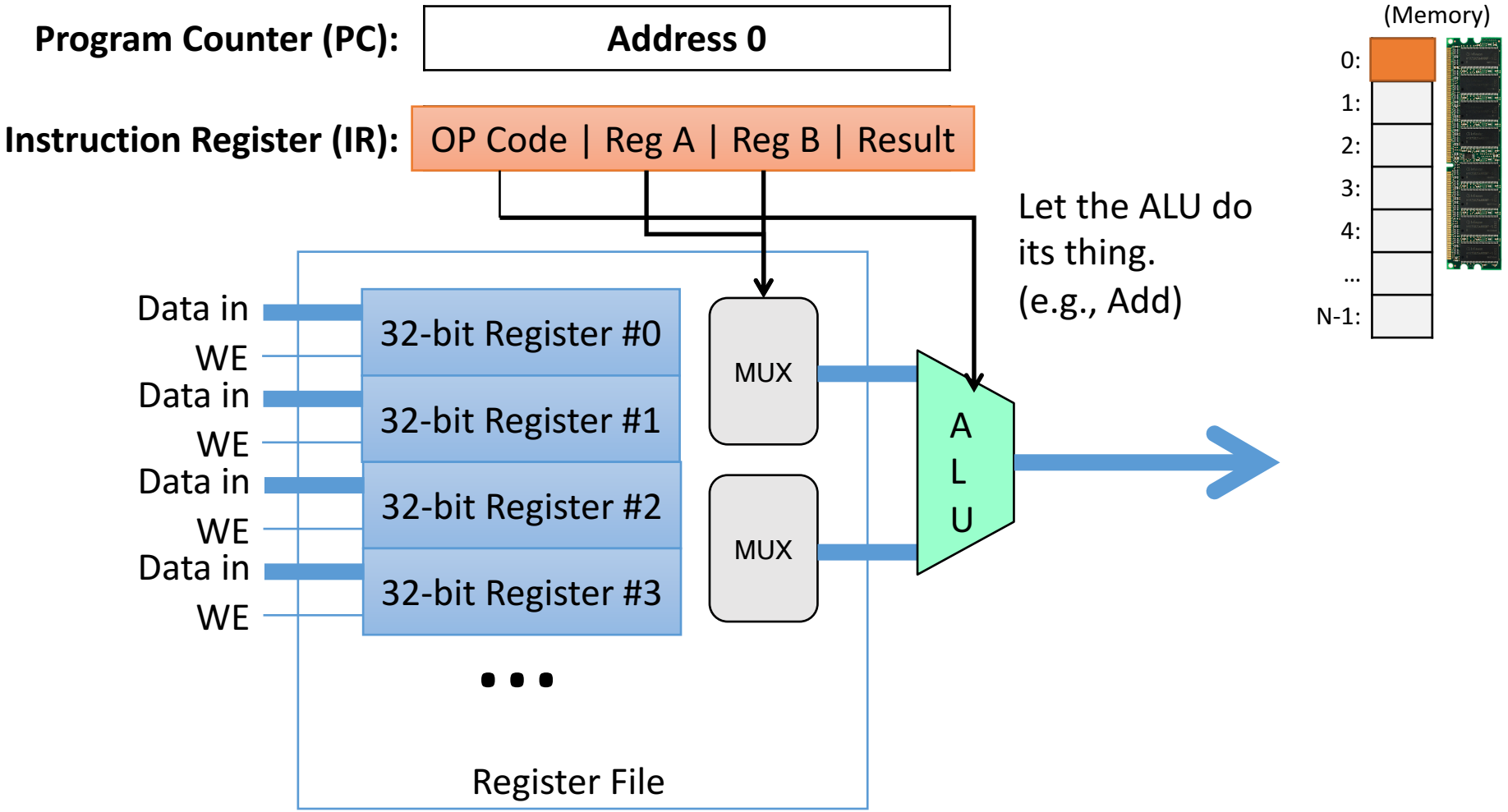
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



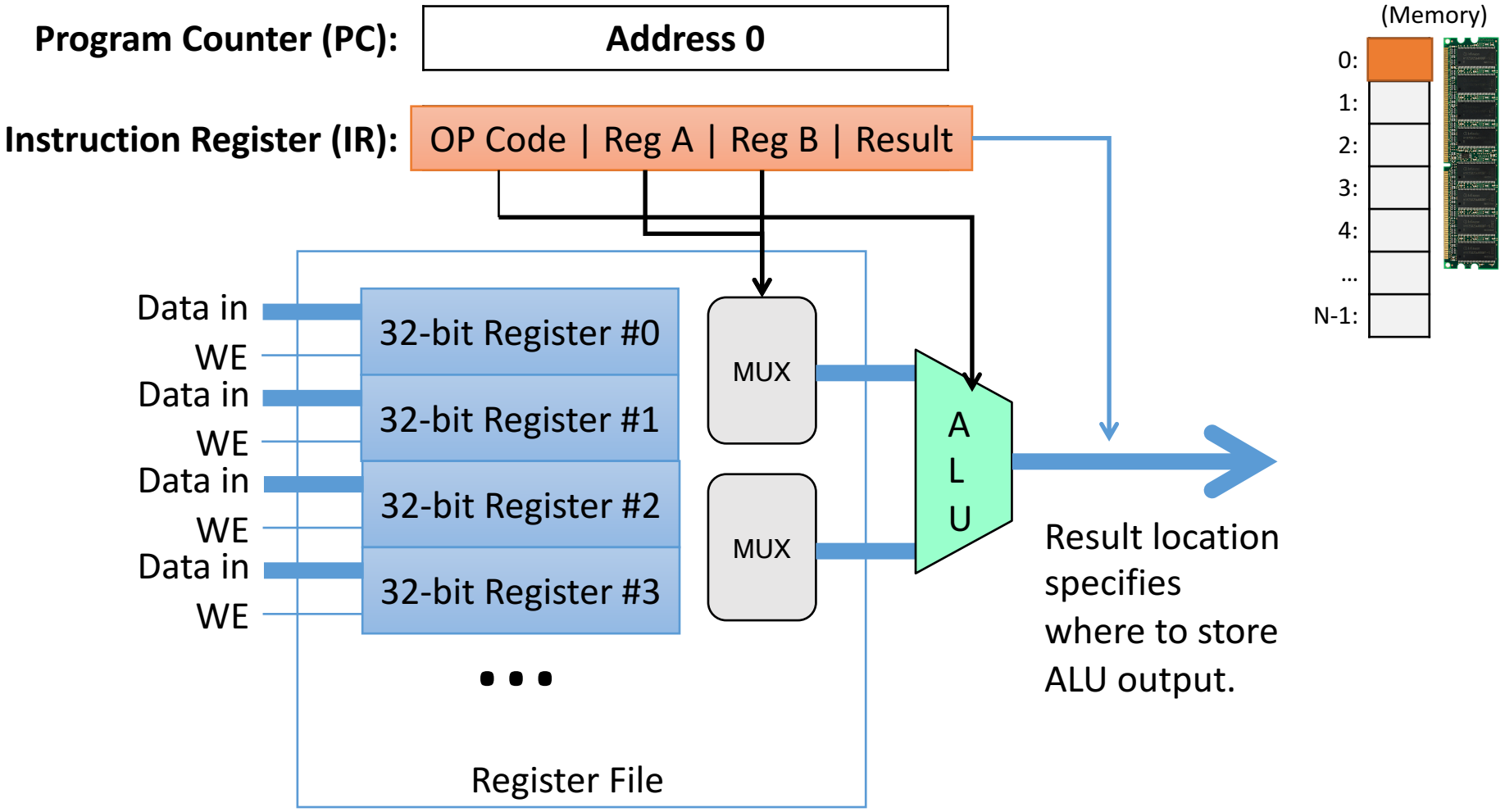
Executing instructions.

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.

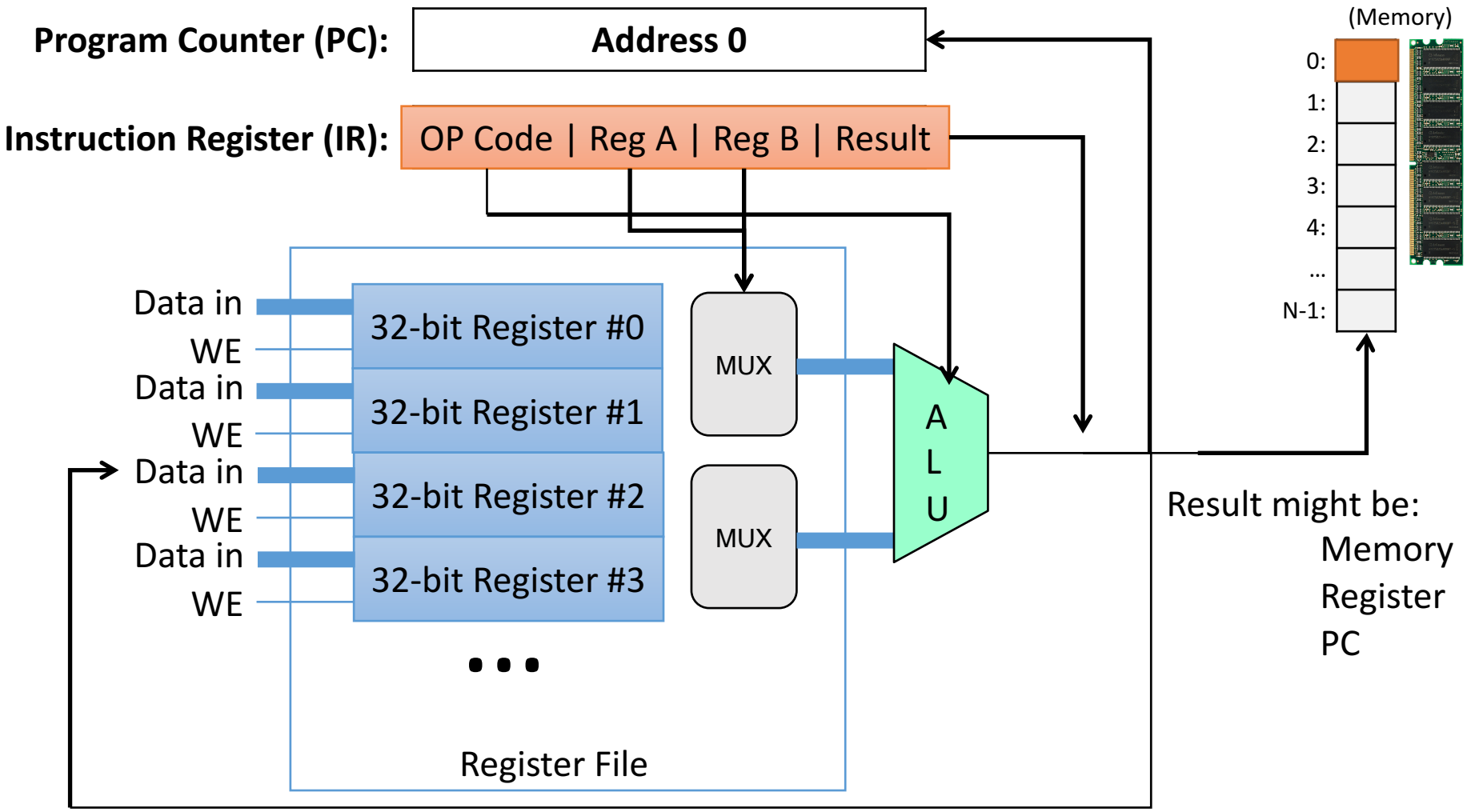
B. Some instructions might skip the PC forward by more than one.

C. Some instructions might adjust the PC backwards.

D. We need the PC for some other reason(s).

Storing results.

Interpret the instruction bits: What operation? Which arguments?

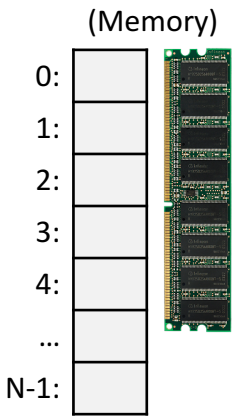


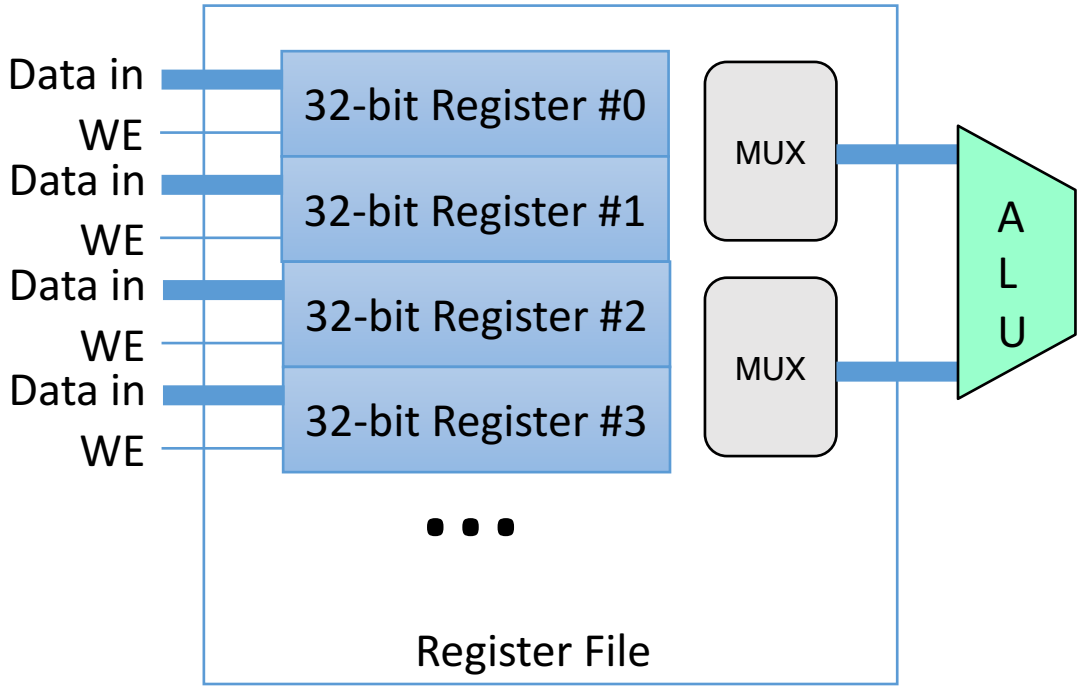
Recap CPU Model

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

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

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



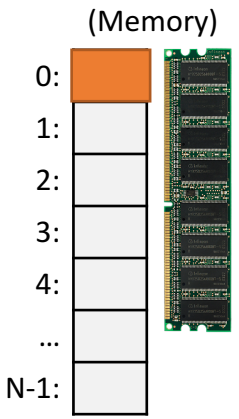


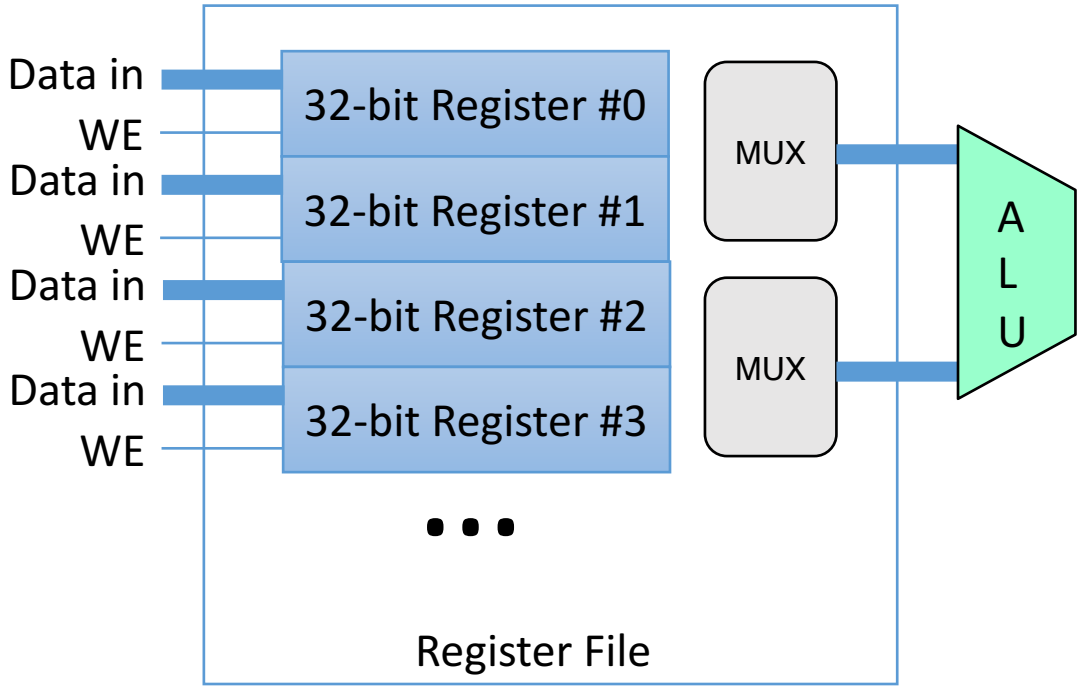
Fetching instructions.

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



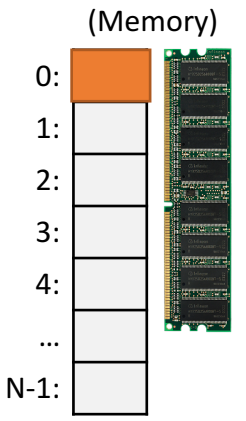


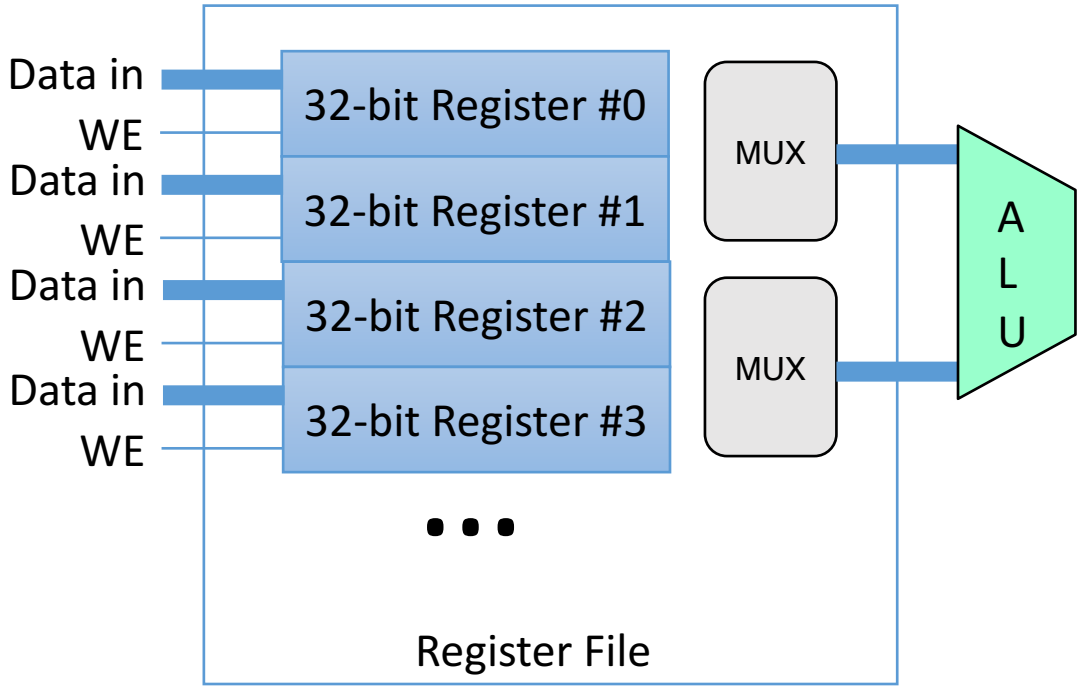
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

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

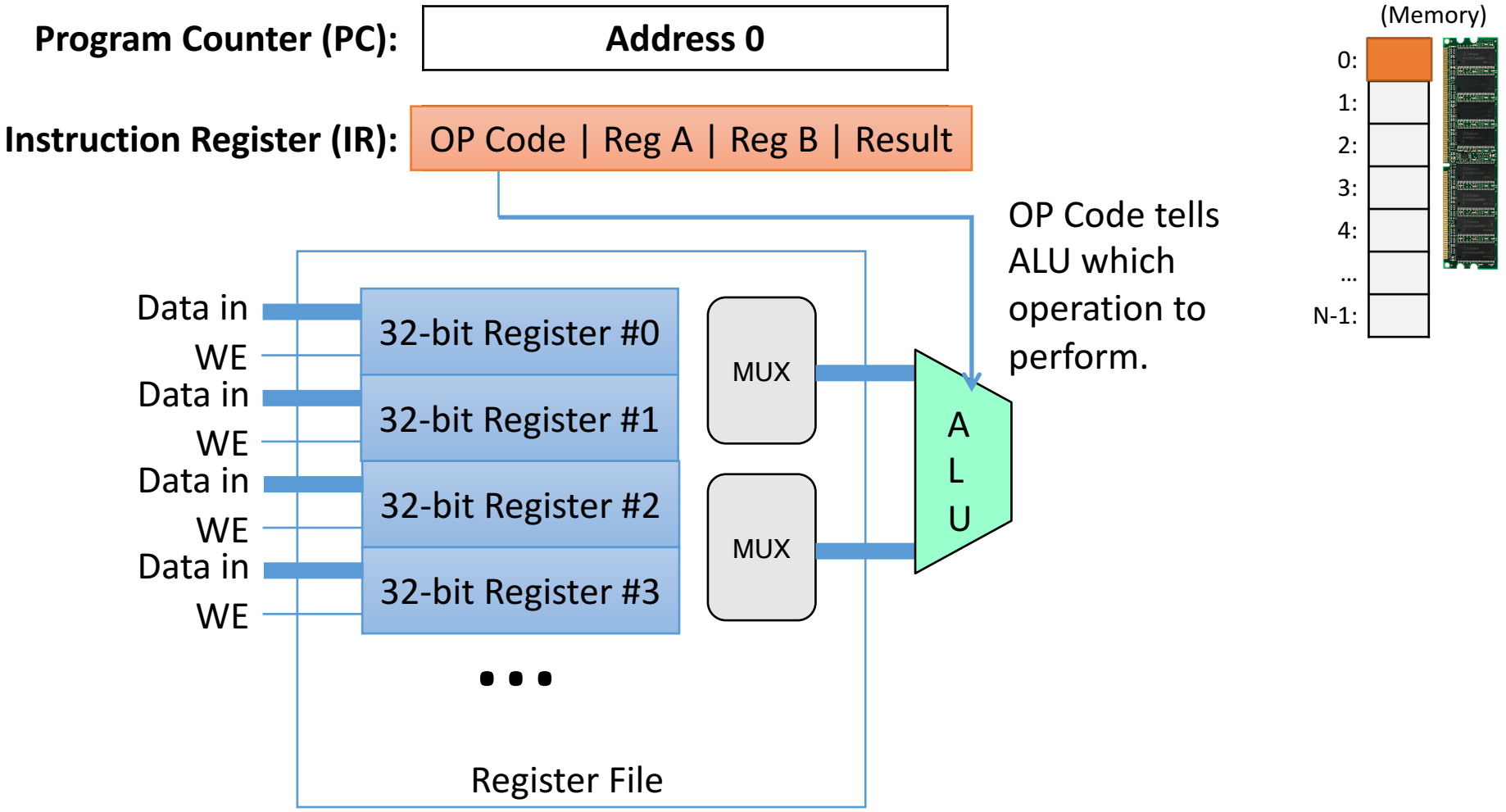
Instruction Register (IR): OP Code | Reg A | Reg B | Result





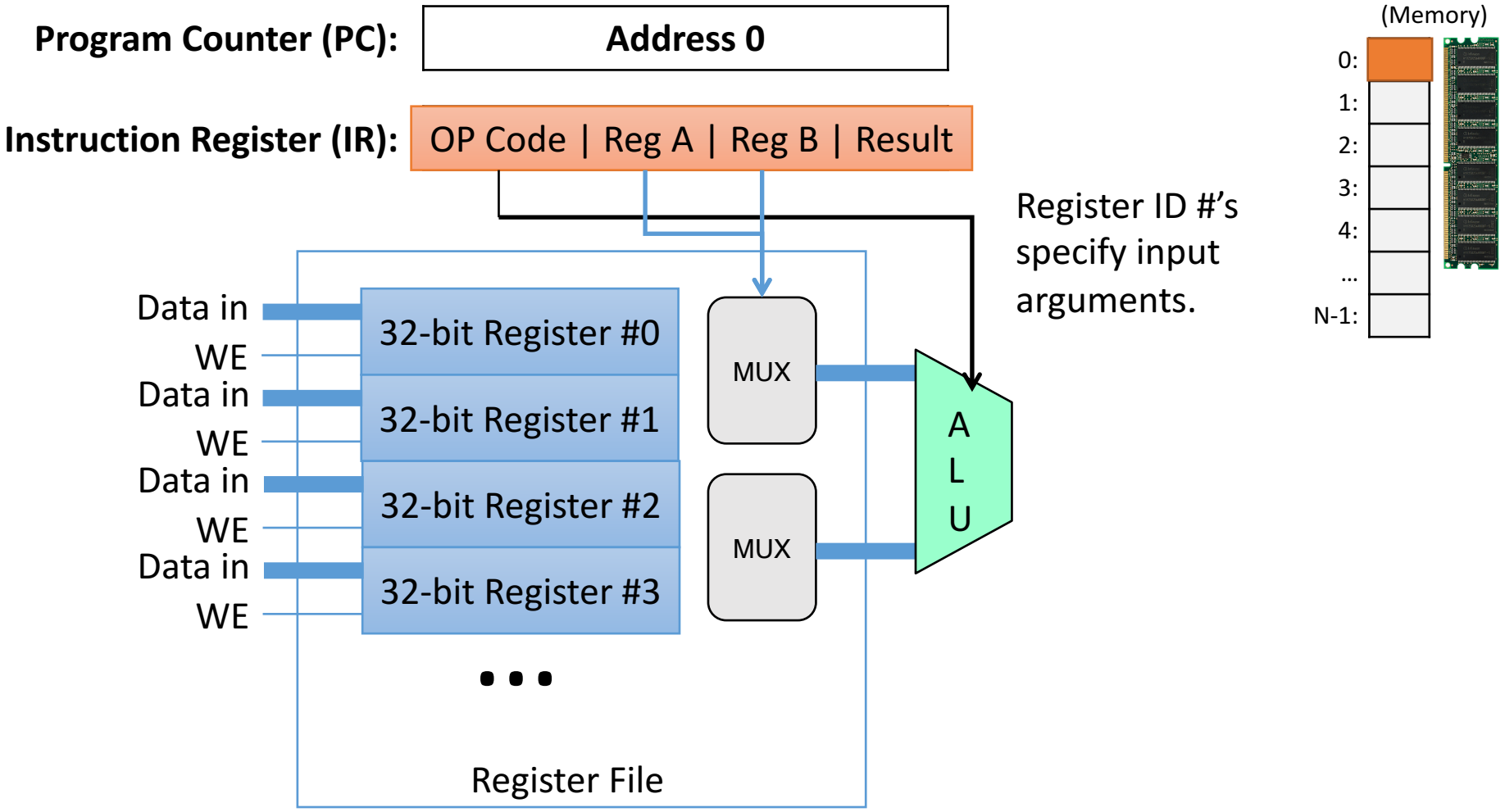
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



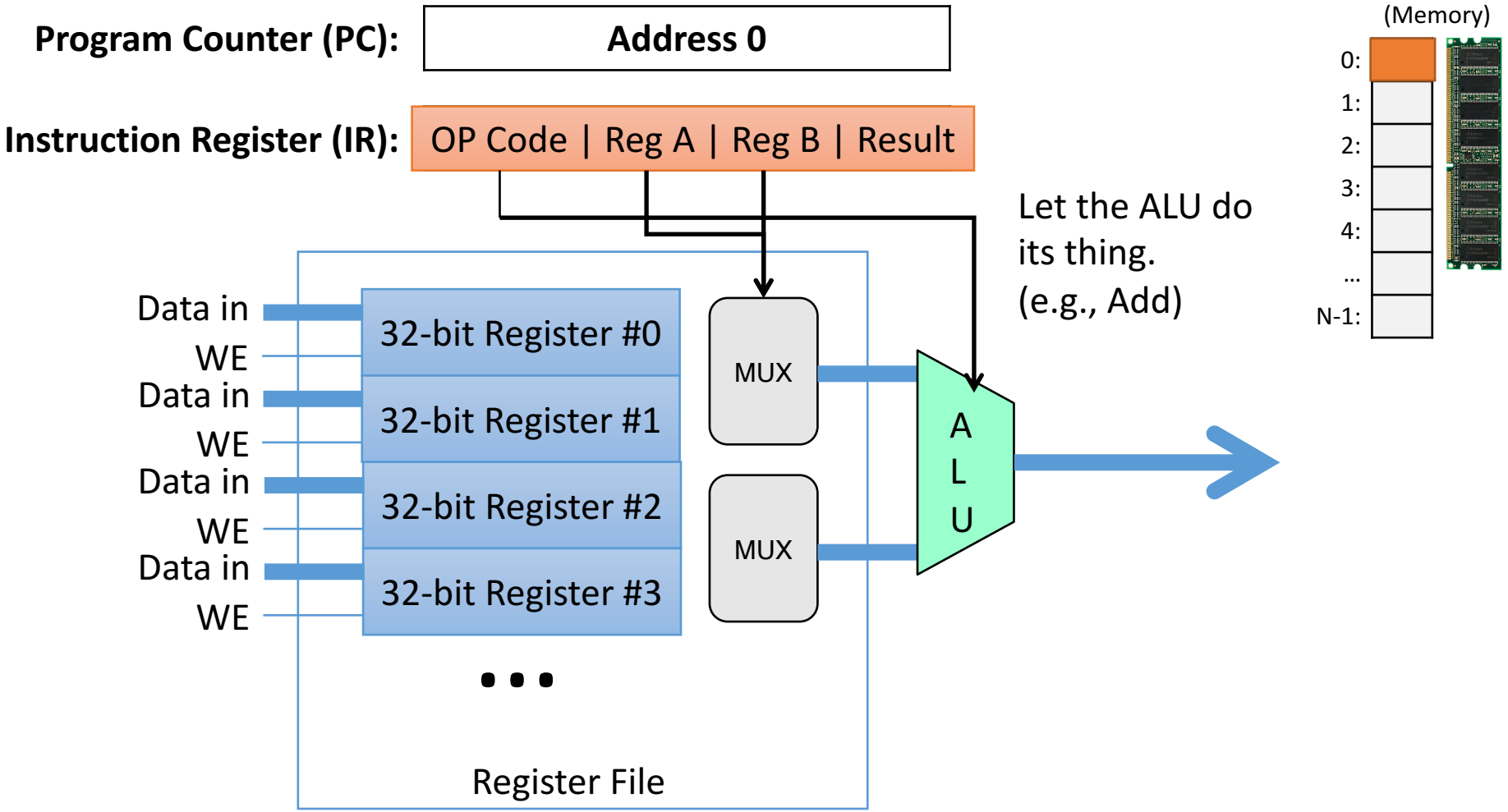
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



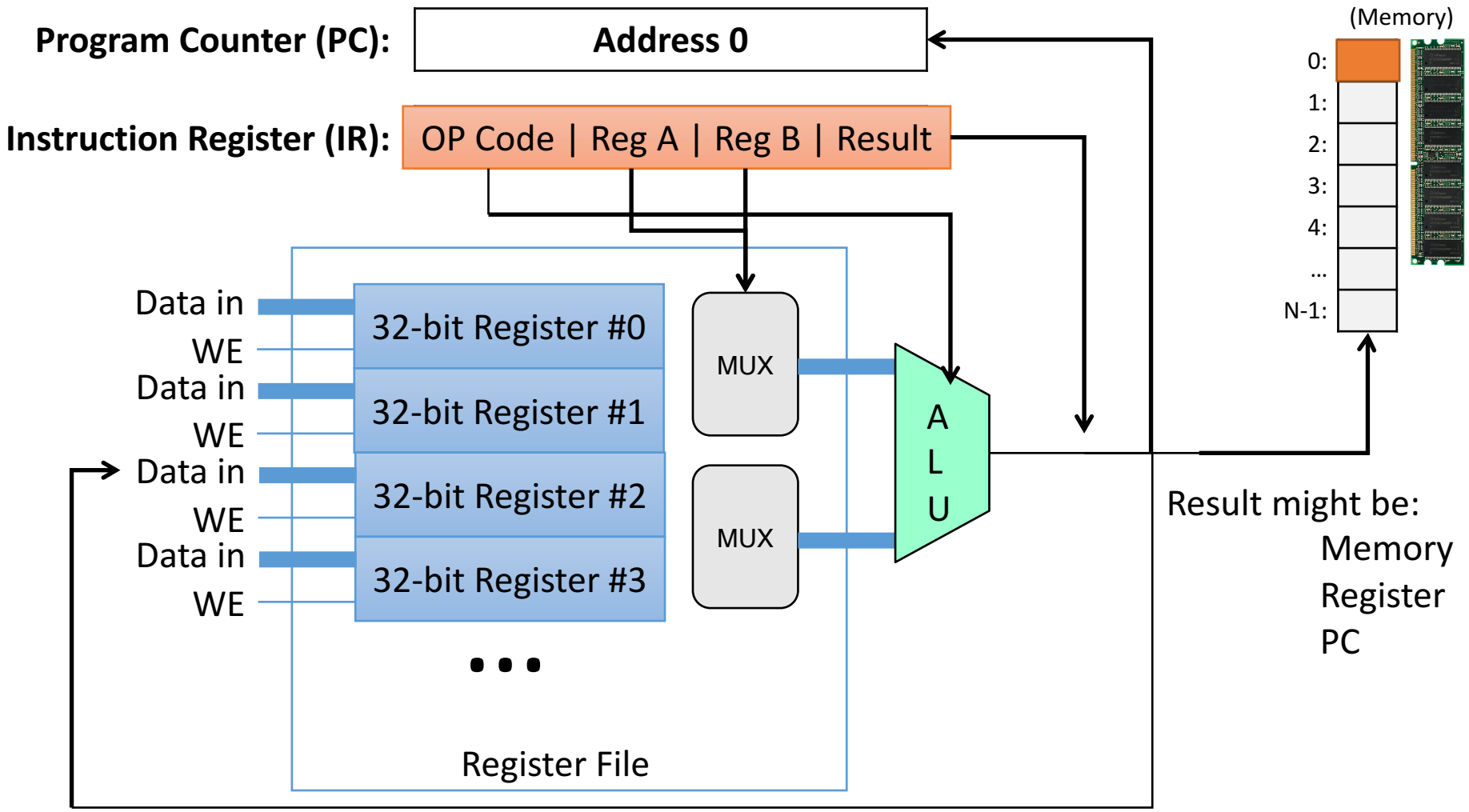
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



Storing results.

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



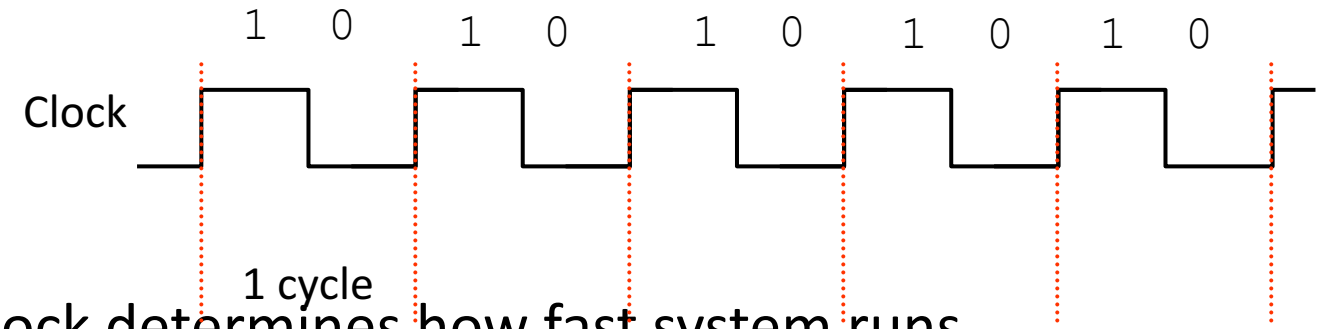
Clocking

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



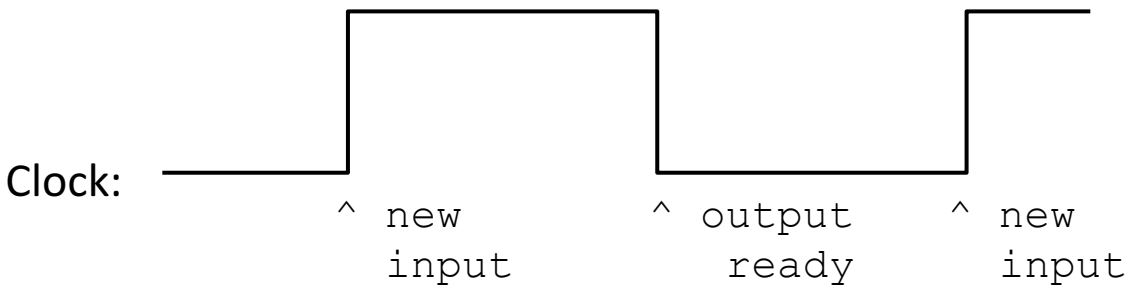
  - Clock determines how fast system runs
    - Processor can only do one thing per clock cycle
      - Usually just one part of executing an instruction
    - 1GHz processor:

      1 billion cycles/second → 1 cycle every nanosecond

Clock and Circuits

Clock Edges Triggers events

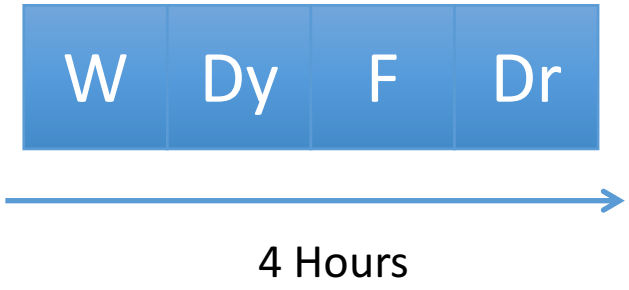
- Circuits have continuous values
- Rising Edge: trigger new input values
- Falling Edge: consistent output ready to read
- Between rising and falling edge can have inconsistent state as new input values flow through circuit



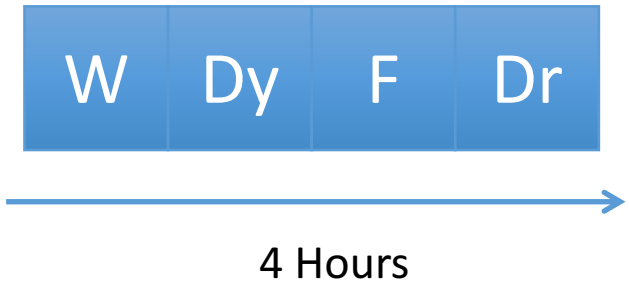
Time per instruction: Laundry Analogy

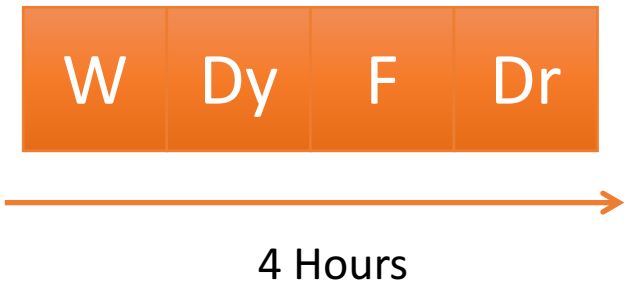
- Discrete stages: fetch, decode, execute, store

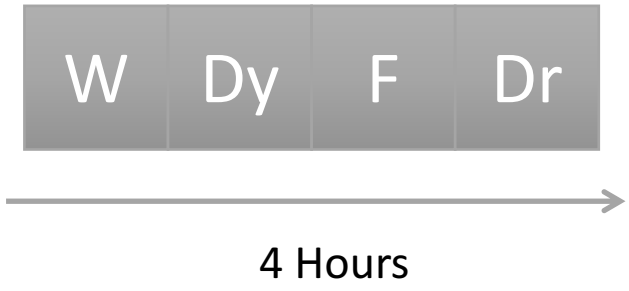
- Analogy (laundry): washer, dryer, folding, dresser



Laundry





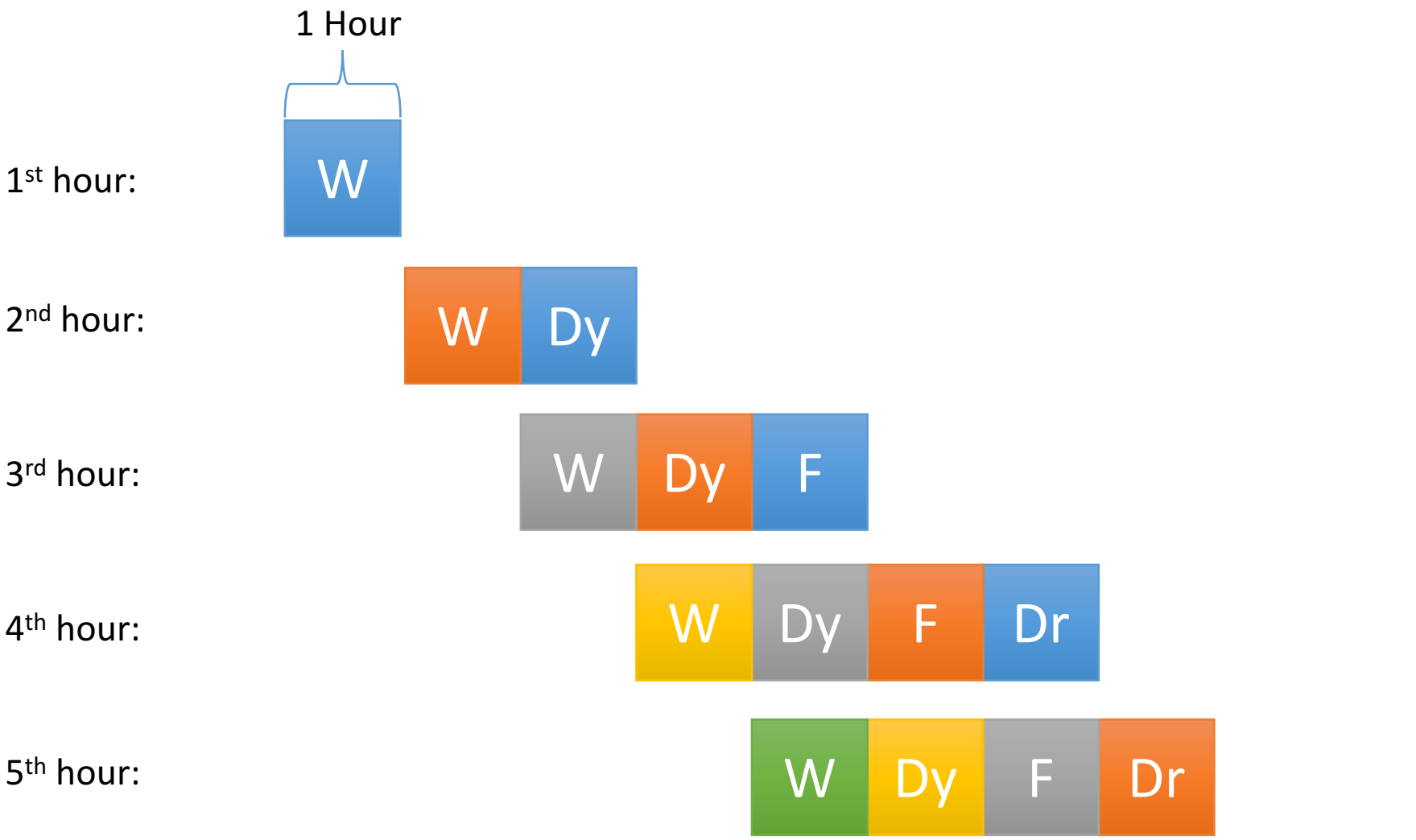


4-hour cycle time.

Finishes a laundry load every cycle.

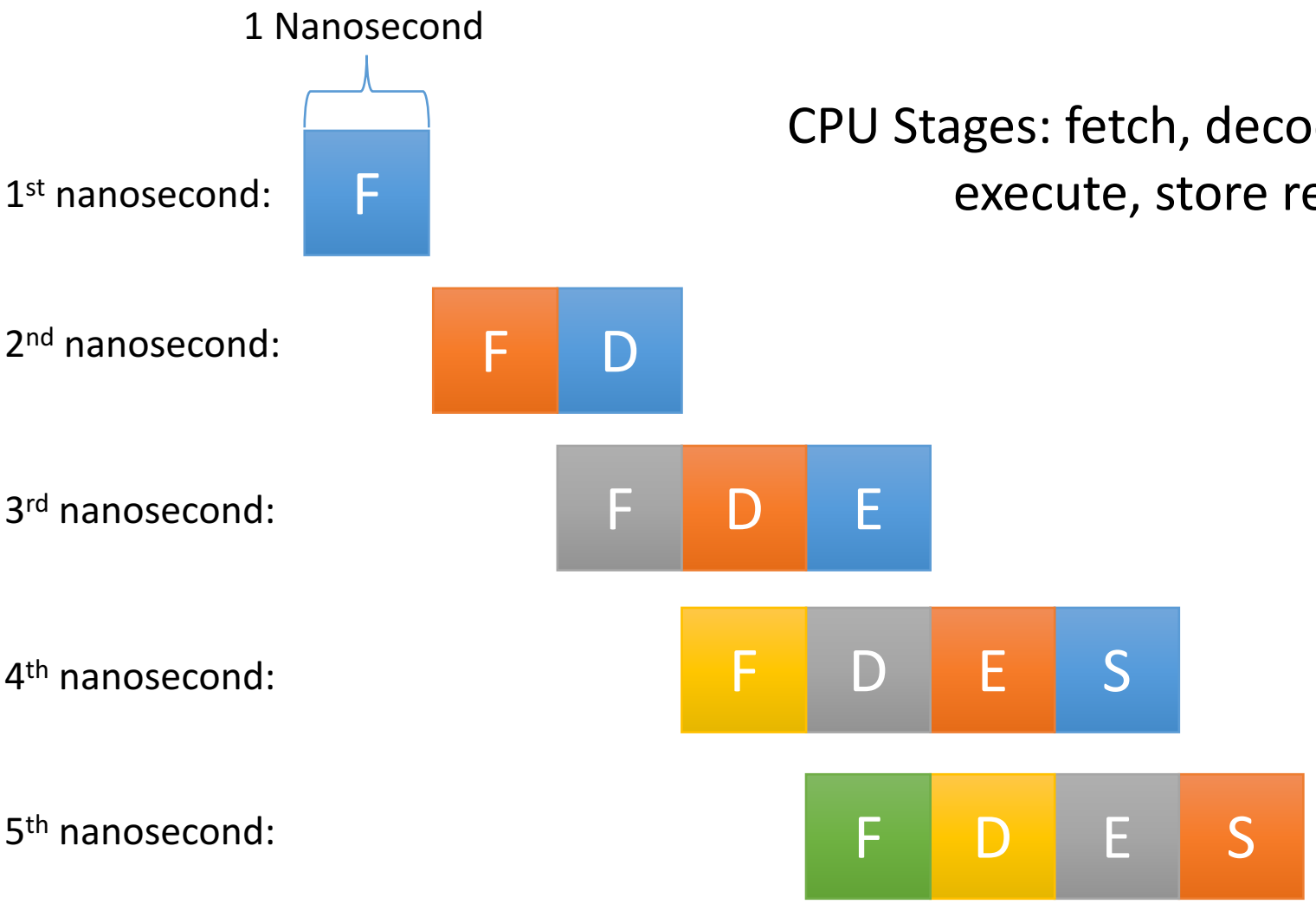
(6 laundry loads per day)

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.)

Coming up next week…

- Talking to the CPU: Assembly language