CS 31: Introduction to Computer Systems

05-06: Digital Logic

February 04, 06

SWARTHMORE COLLEGE

Reading Quiz

Today

- C Programming Wrap Up
  - Arrays, Strings
  - Structs
  - Functions

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

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

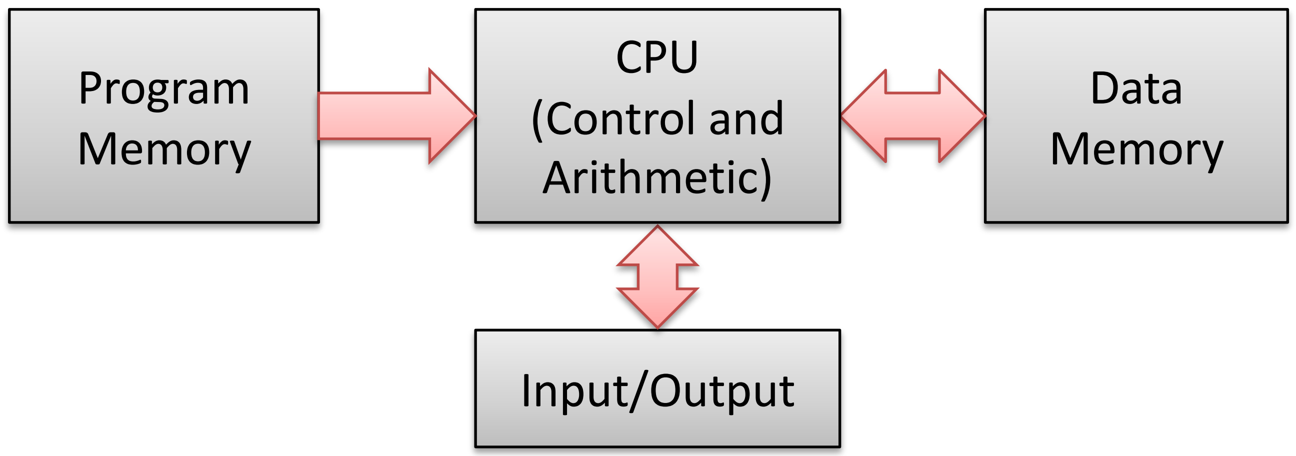
Circuits: Borrow some paper if you need to!

Hardware Models (1940's)

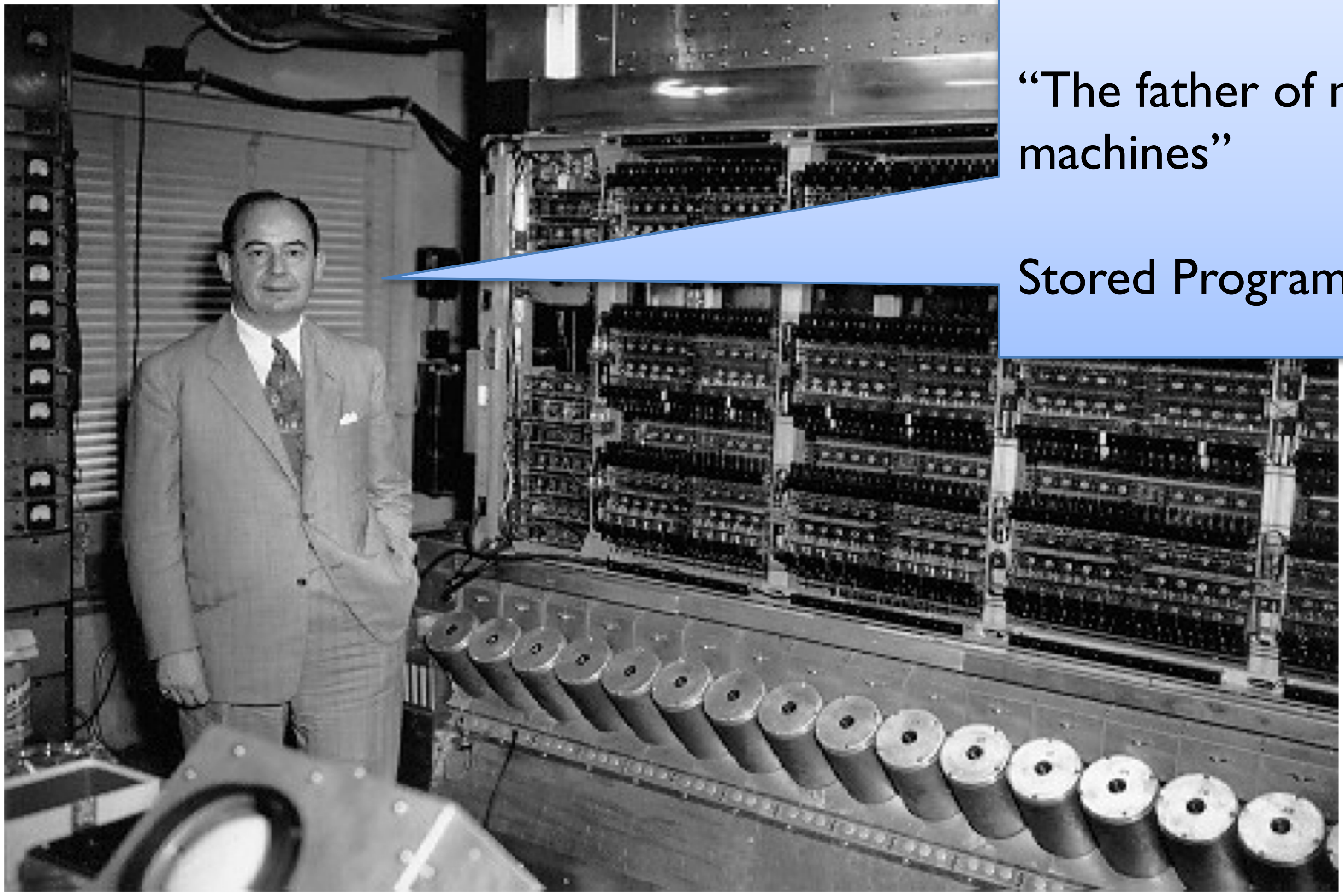
- Harvard Architecture:

Hardware Models (1940's)

- Harvard Architecture:



Von Neumann



John von Neumann

"The father of modern machines"

Stored Program Concept

EDVAC 1945

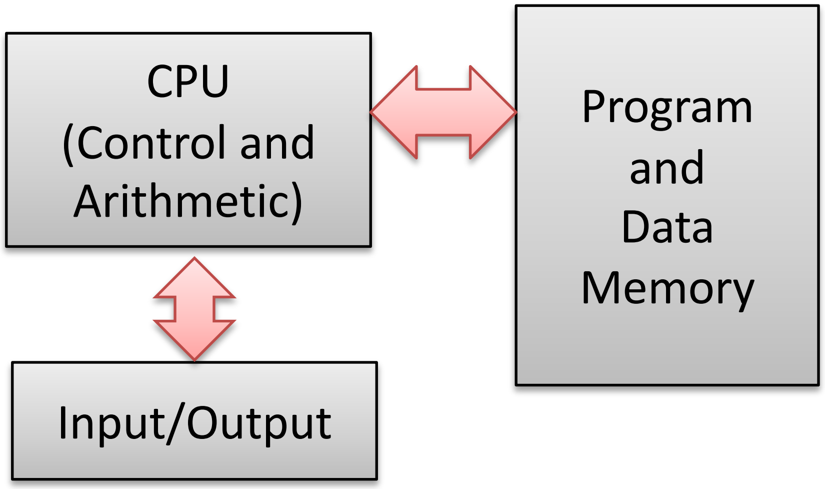
Stored Program Concept

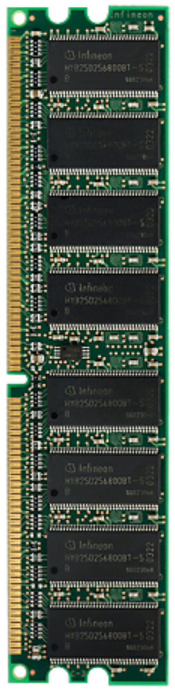
- **Fixed Machines**
  - Early machines had “fixed” programs
  - Can’t be used for other purpose
  - Change required re-design & re-wiring!
- **General Purpose Machine**
  - Need more versatility
  - Programs stored in a “memory”
  - Machine can be re-programmed!

Instructions encode functionality – programs!

Von Neumann Architecture

- Von Neumann Architecture:





Von Neumann Architecture Model

Based on Alan Turing's Universal Turing Machine

Computer is a generic computing machine:

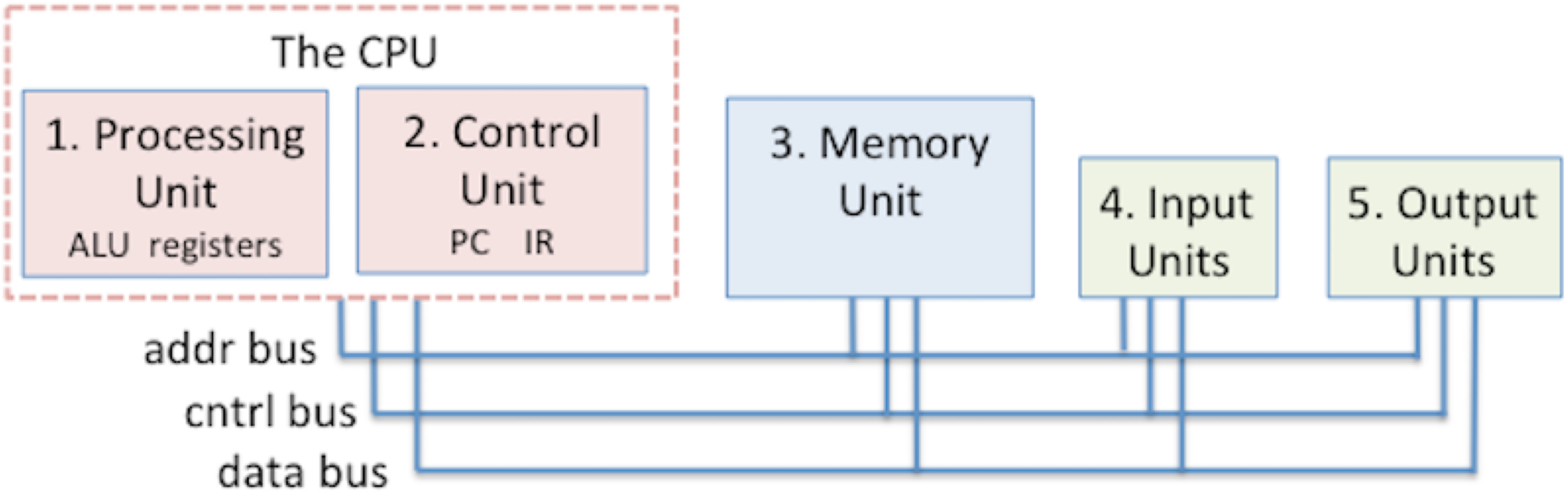
- Stored program model: computer stores program rather than encoding it (feed in data and instructions)

No distinction between data and instructions memory

Von Neumann Architecture Model

5 parts connected by buses (wires):

Control, Memory, Processing, Input, Output

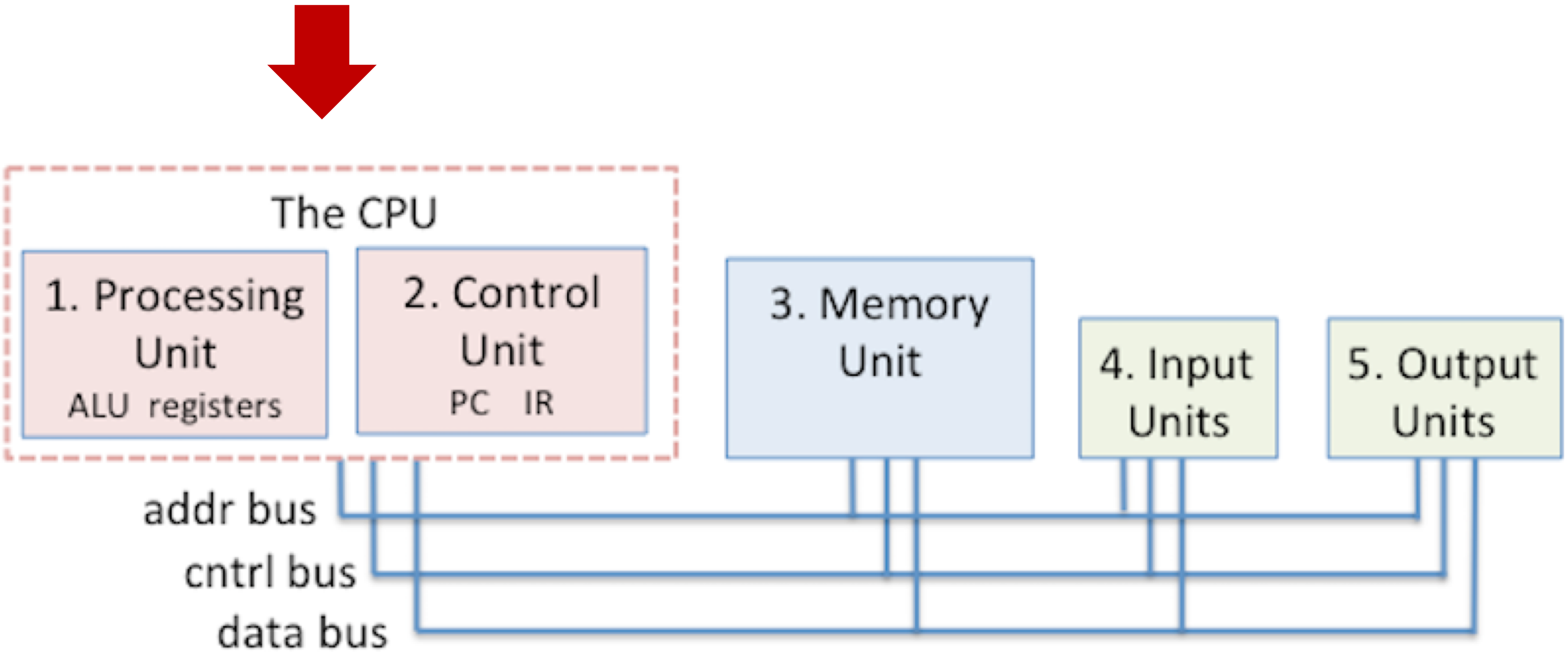


Control: The CPU

Processing Unit: executes instrs selected by cntrl unit

ALU (artithmetic logic unit): simmple functional units: ADD, SUB…

Registers: temporary storage directly accessible by instructions



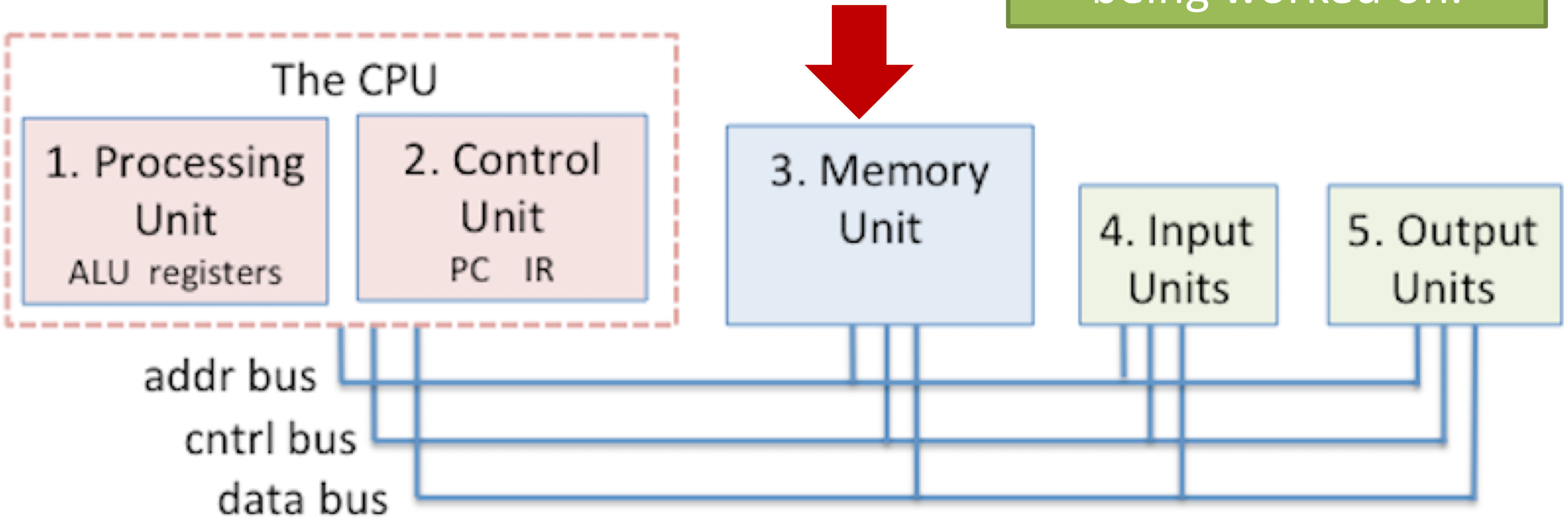
Memory

Memory: data and instructions are stored

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

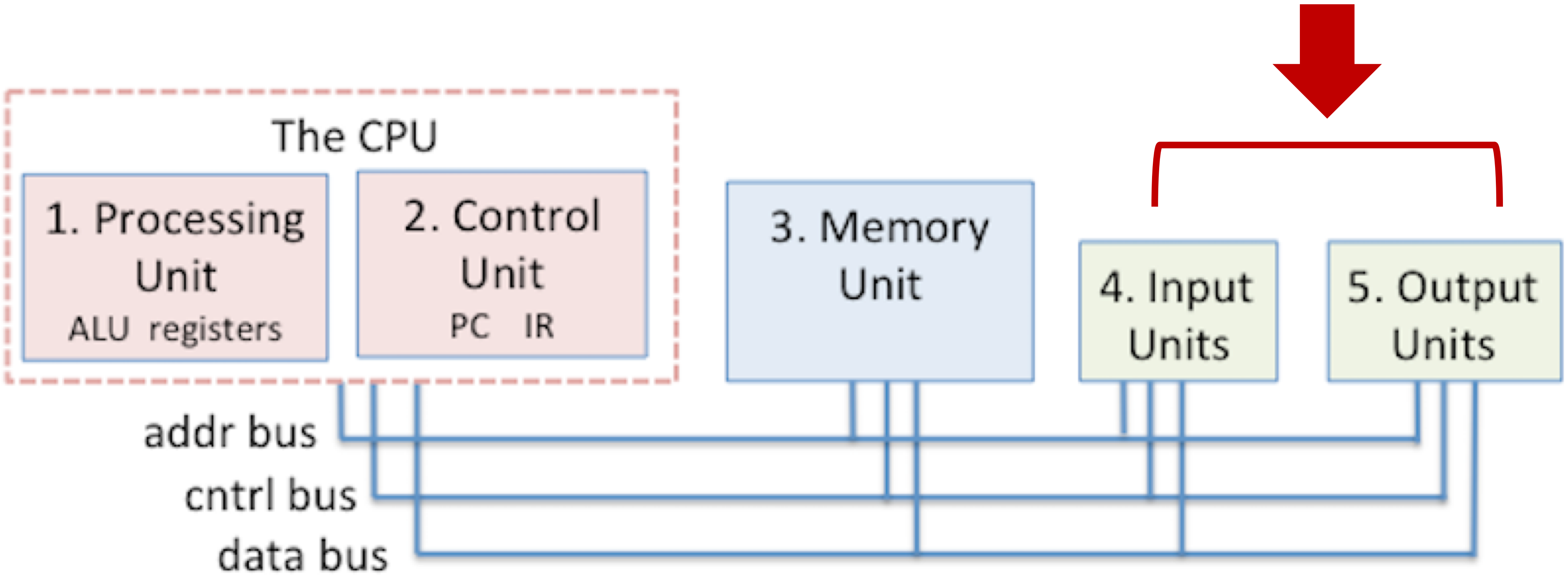
“Register”
Small, very vast storage space.
Fixed size (e.g., 32 bits).

Stores what is currently being worked on.



Input/Output

Input/Output: keyboard (can trigger actions), terminal, disk



Digital Computers

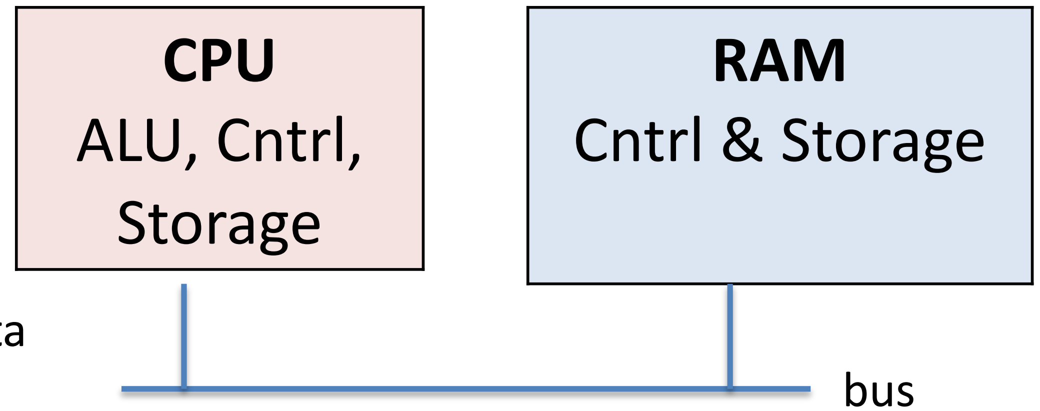
- All input is discrete (driven by periodic clock)

- All signals are binary (0: no voltage, 1: voltage)

  data, instructions, control signals, arithmetic, clock

- To run program, need different types of circuits

Circuits to execute program instructions that act on program data



bus

Circuits to store program data and instructions and support reading and writing addressable storage locations

Goal: Build a CPU (model)

Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality

   (ex) adder to add two values together

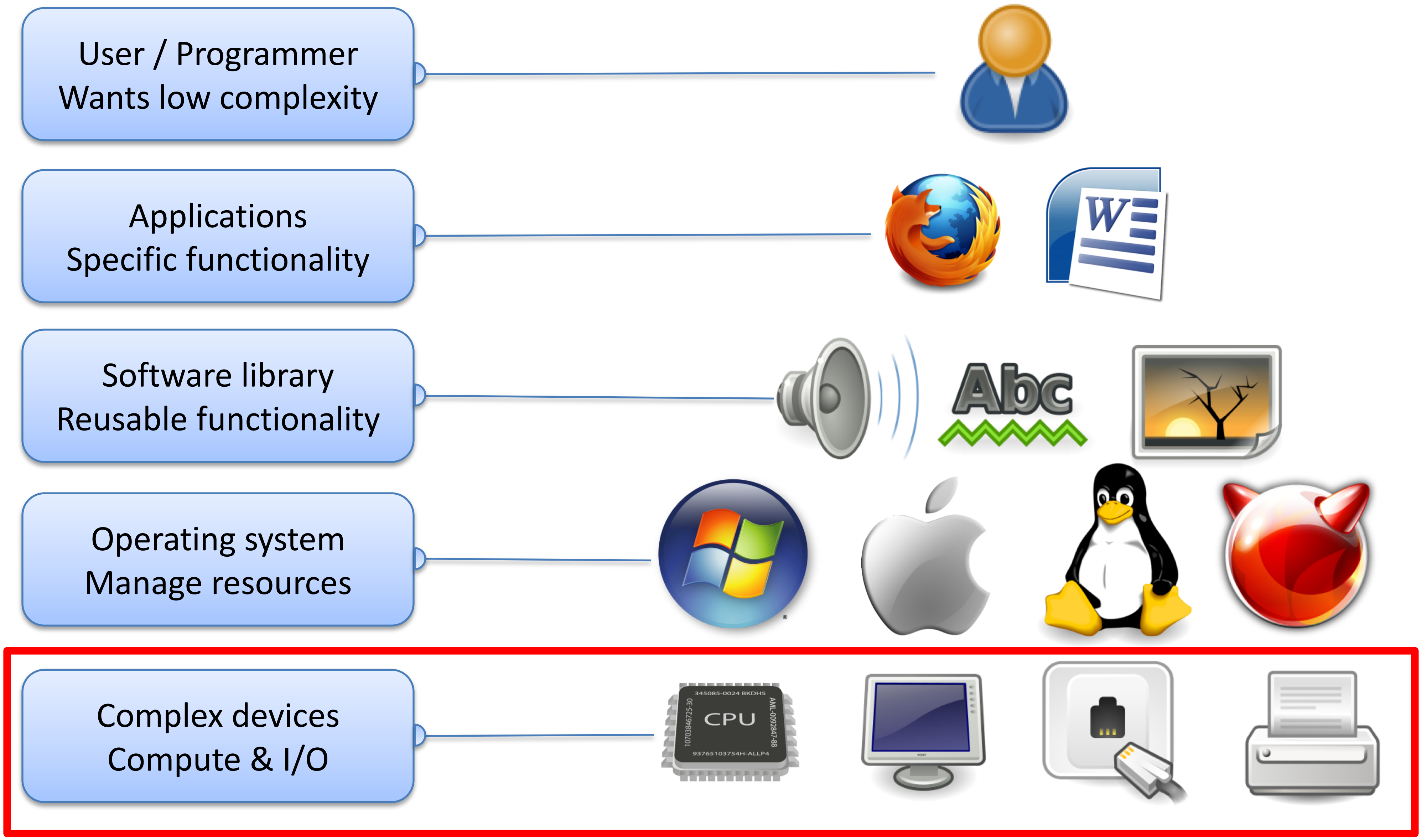
2. Storage: to store binary values

   (ex) Register File: set of CPU registers, Also: main memory (RAM)

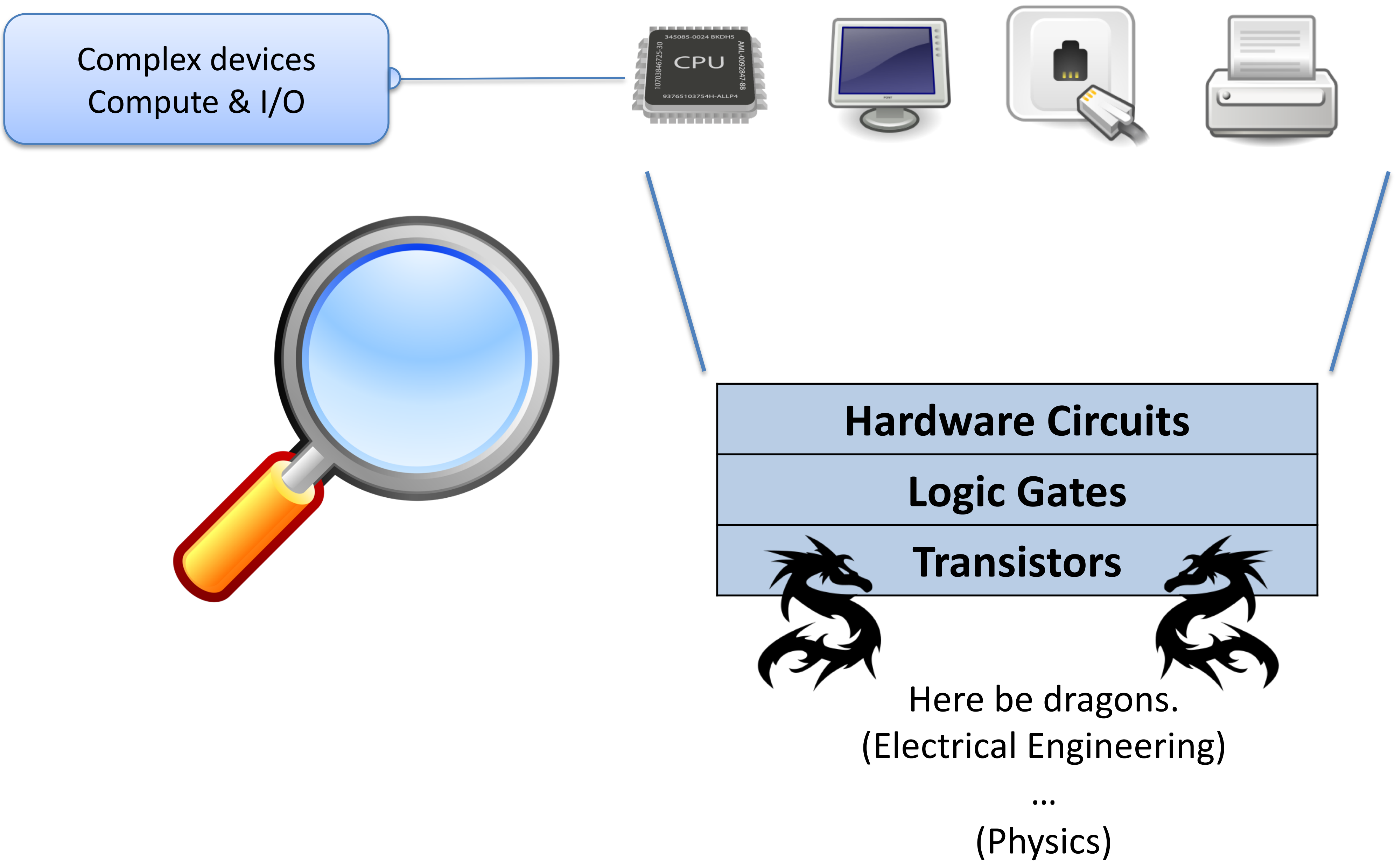
3. Control: support/coordinate instruction execution

   (ex) fetch the next instruction to execute

Abstraction



Abstraction

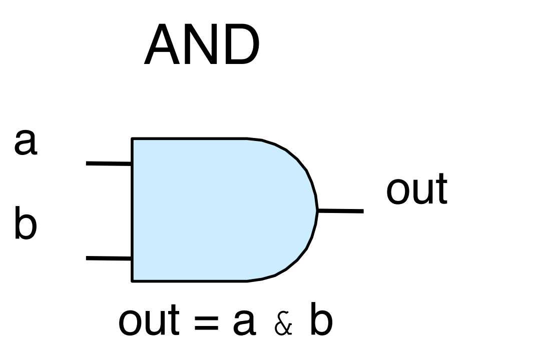


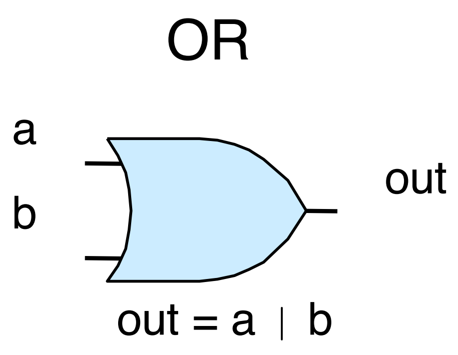
Logic Gates

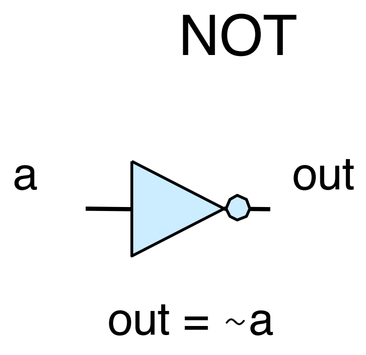
Input: Boolean value(s) (high and low voltages for 1 and 0)

Output: - Boolean value result of boolean function

- Always present, but may change when input changes





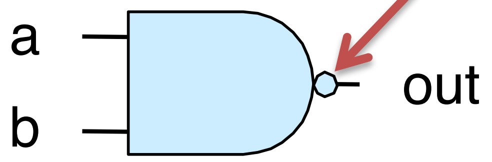


A	B	A & B	A | B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

More Logic Gates

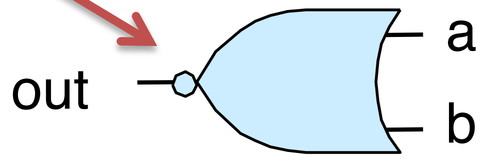
Note the circle on the output.
This means “not.” (flip bits)

NAND



out = ~(a & b)

NOR



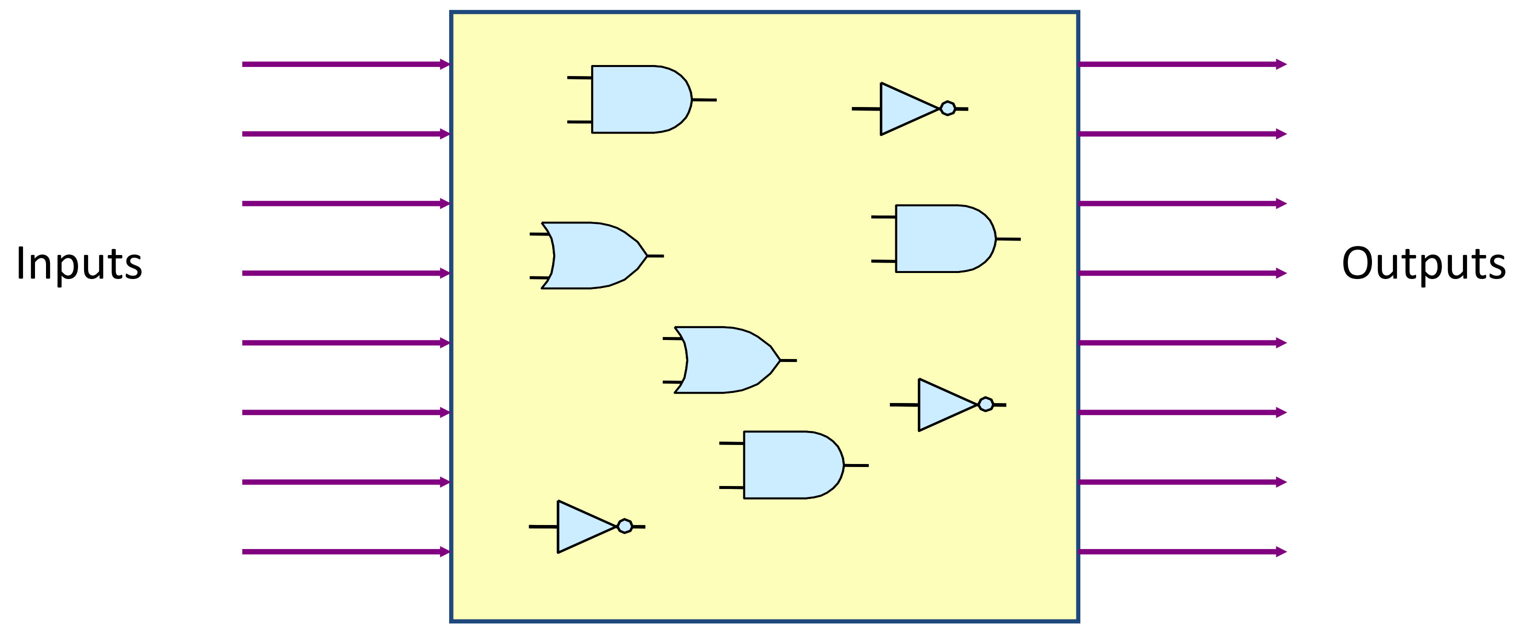
out = ~(a | b)

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

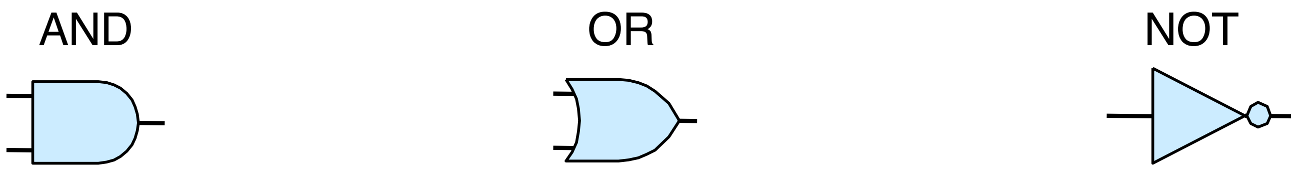
Acyclic Network of Gates

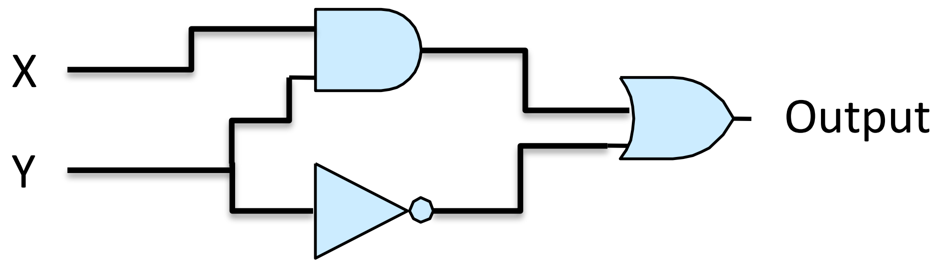


Outputs are Boolean functions of inputs

Outputs continuously respond to changes to inputs

What does this circuit output?

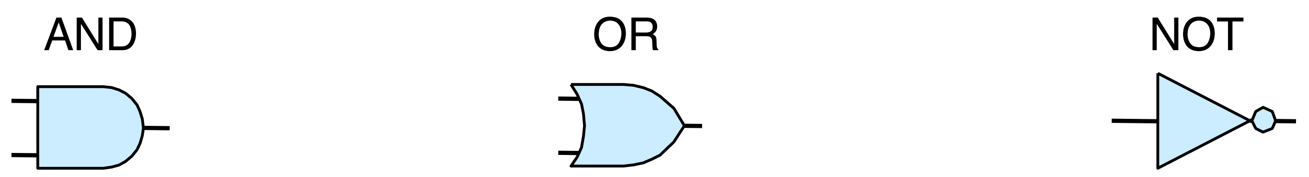


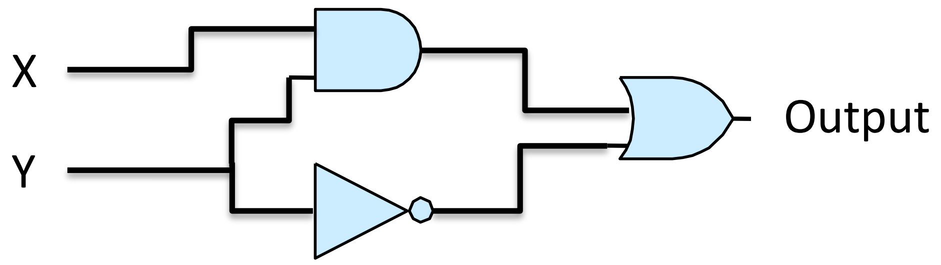


Clicker Choices

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

What does this circuit output?

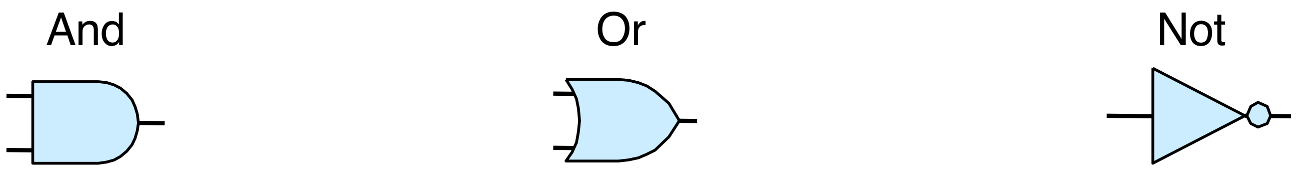


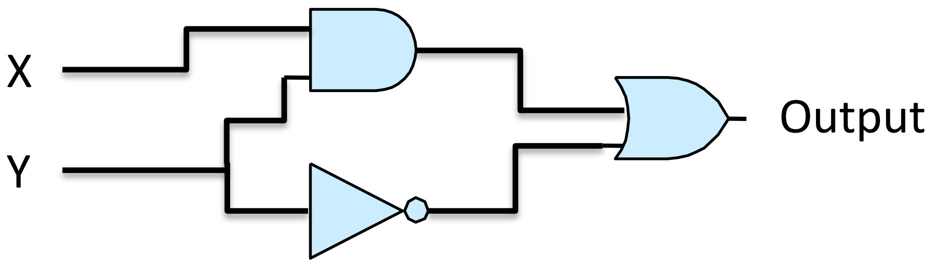


Clicker Choices

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~~

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~~

What can we do with these?

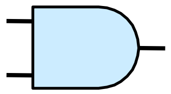
- Build-up XOR from basic gates (AND, OR, NOT)

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

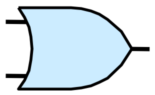
Q: When is A^B ==1?

Which of these is an XOR circuit?

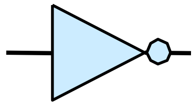
And



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.

Build a Circuit from Basic Gates

- Build XOR using (AND, OR, NOT)

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

How:

(1) Construct Logic Table

(2) When is A^B ==1?

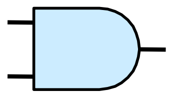
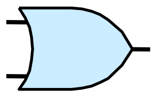
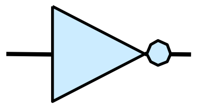
(a) express condition of each row of 1 result, in terms of input values A, B combined with &, ~

(b) combine each row expression with |

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

(3) : Translate expression to circuit

Build XOR circuit from basic gates (AND, OR, NOT)

And  Or  Not 

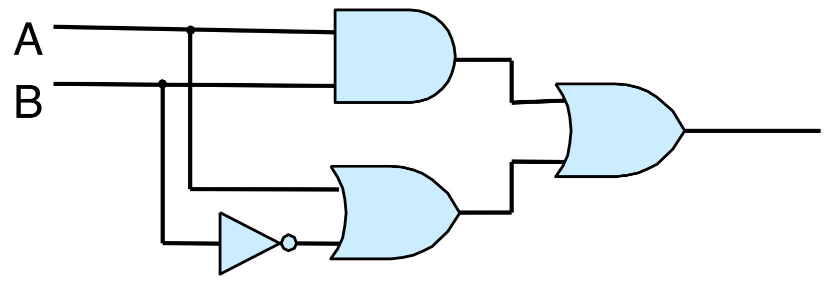
Use `A^B == 1: (~A & B) | (A & ~B)`

Test out circuit

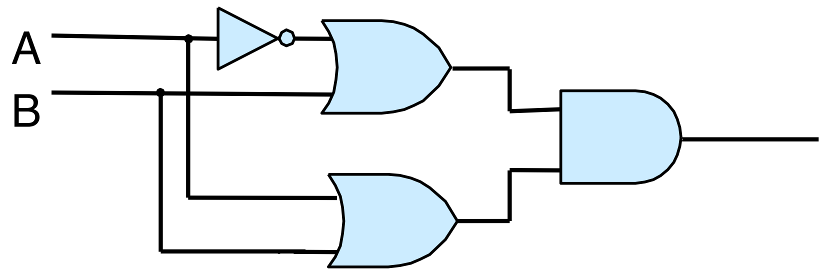
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Which of these is an XOR circuit?

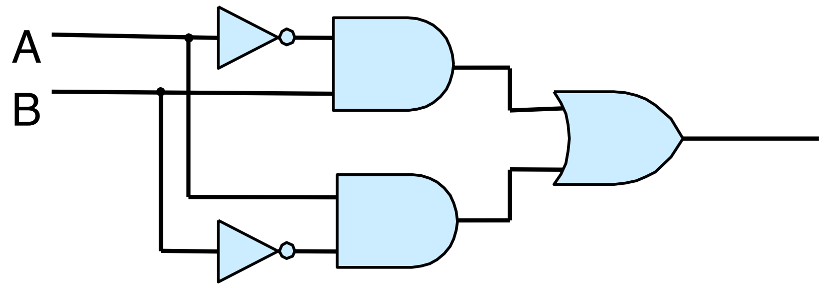
A:



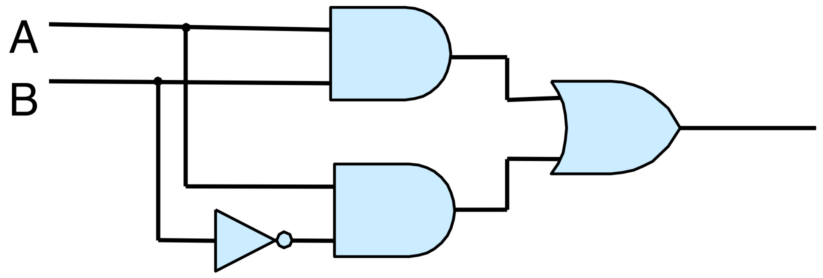
B:



C:



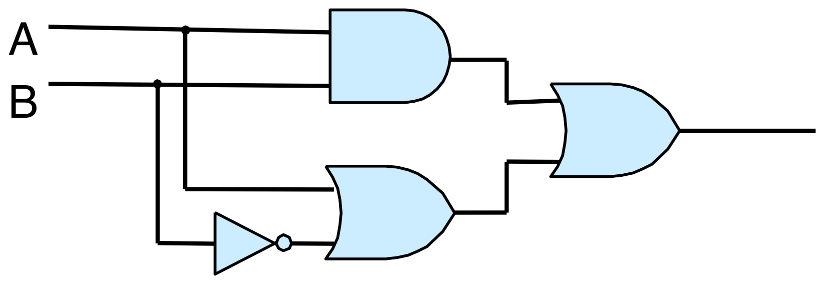
D:



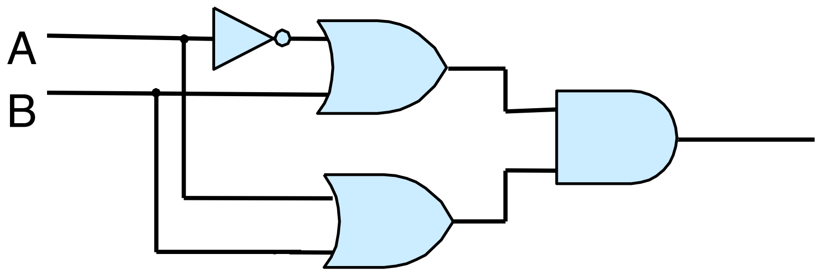
E: None of these are XOR.

Which of these is an XOR circuit?

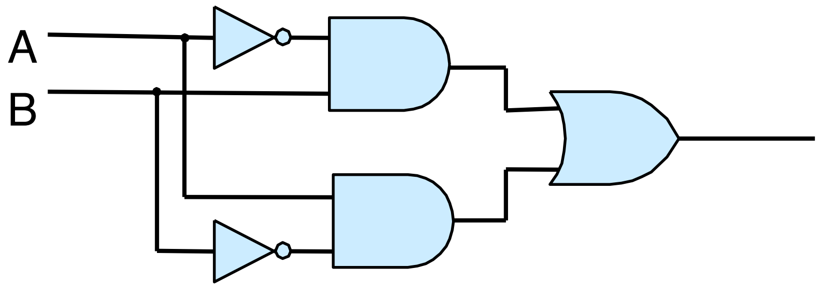
A:



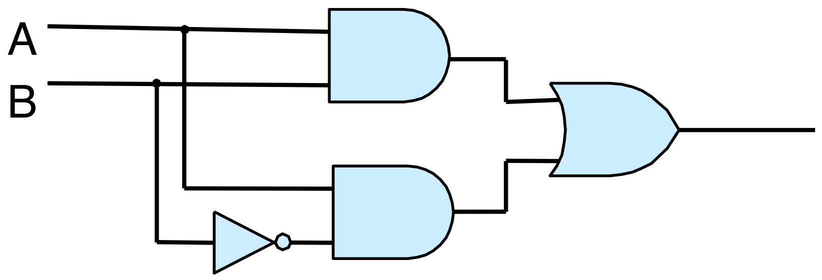
B:



C:



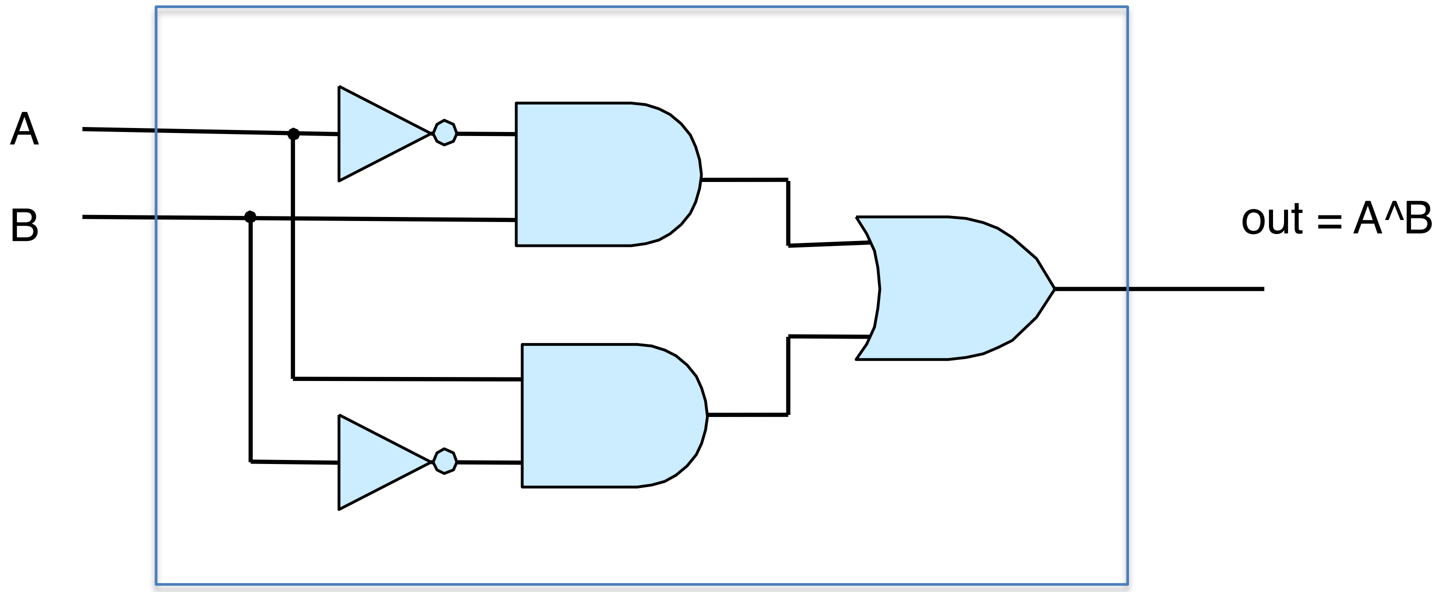
D:



E: None of these are XOR.

XOR Circuit

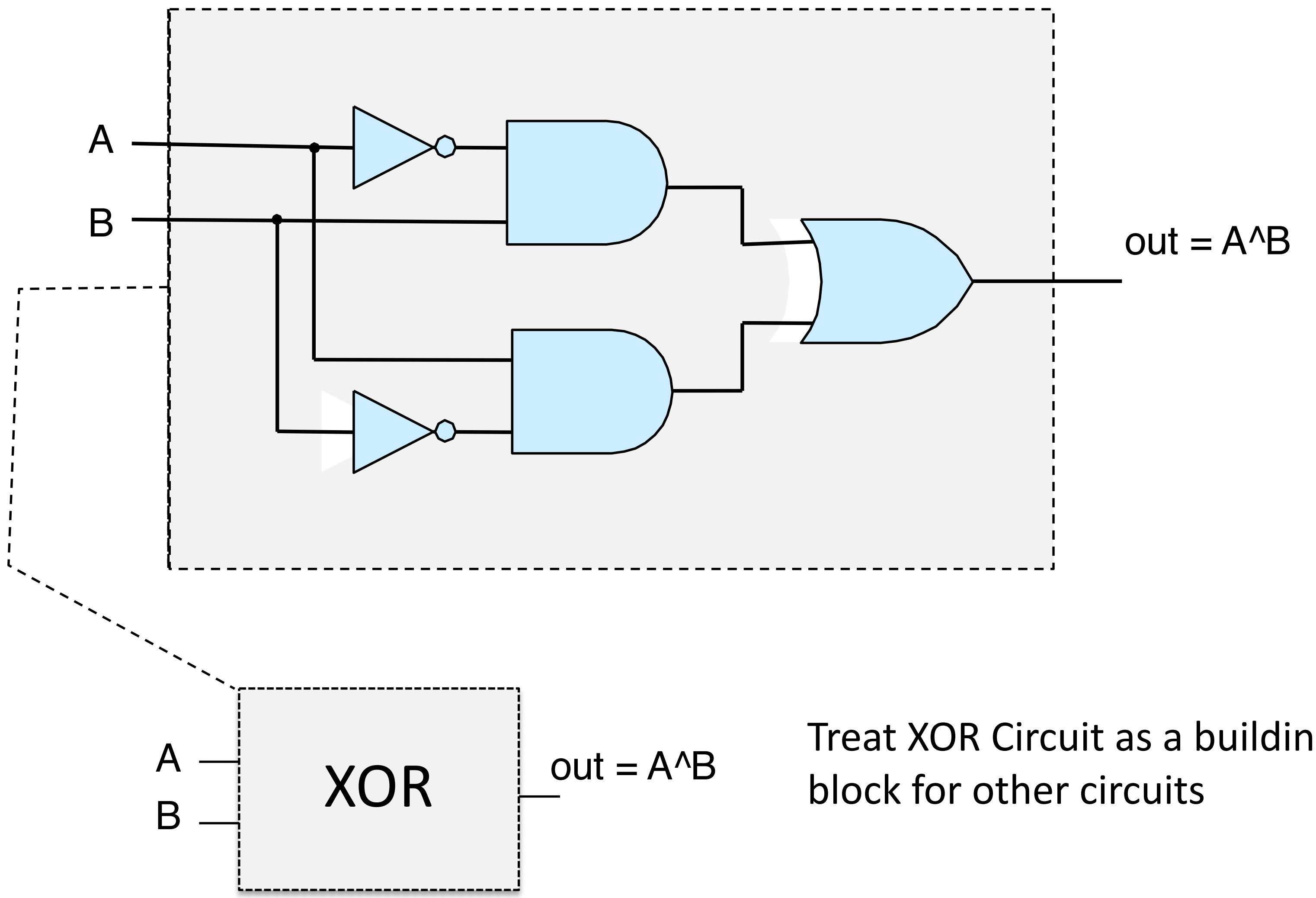
```
A^B  == (~A & B) | (A & ~B)
```



```
A:0 B:0 A^B:              A:1 B:0 A^B:
A:0 B:1 A^B:              A:1 B:1 A^B:
```

XOR Circuit: Abstraction!

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



Treat XOR Circuit as a building block for other circuits

Digital Circuits - Building a CPU

Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality

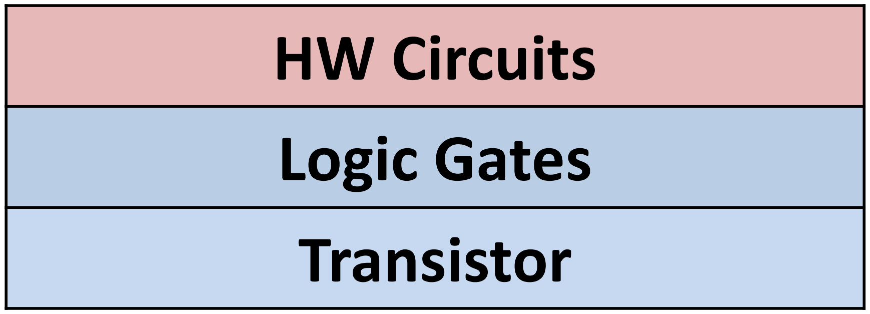
   (ex) adder to add two values together

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



Digital Circuits - Building a CPU

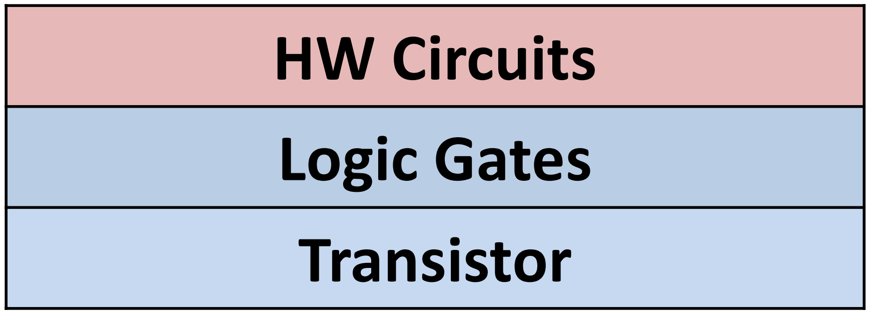
<u>Three main classifications of HW circuits:</u>

1. ALU: implement arithmetic & logic functionality

   (ex) adder to add two values together

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

Combine into ALU!



Arithmetic Circuits

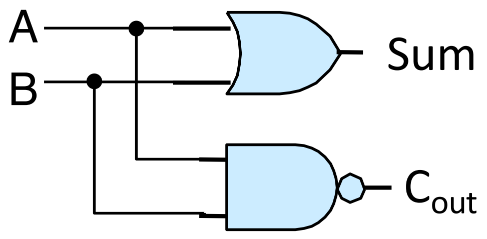
- 1 bit adder: A+B
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

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

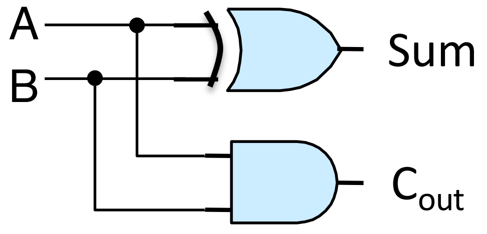
Which of these circuits is a one-bit adder?

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

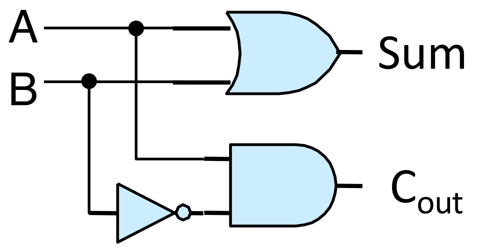
A:



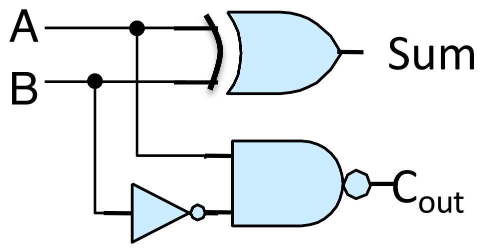
B:



C:



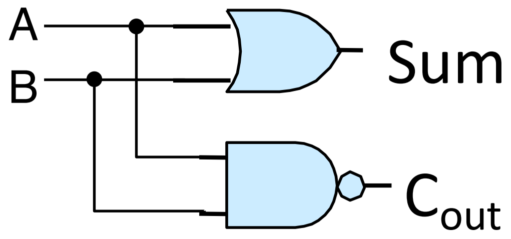
D:



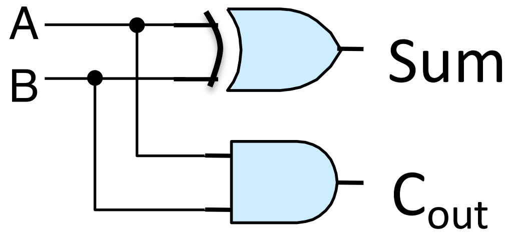
Which of these circuits is a one-bit adder?

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

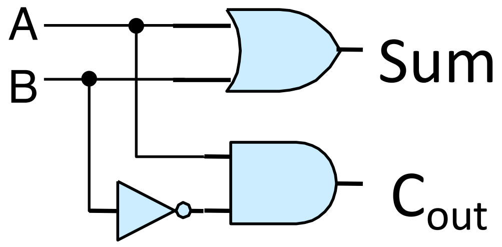
A:



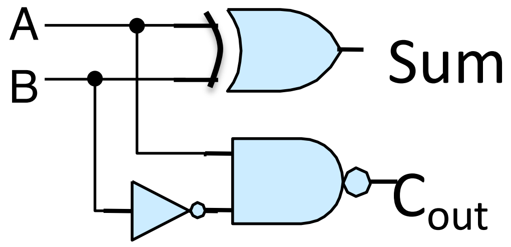
B:



C:



D:



More than one bit?

- When adding, sometimes have *carry in* too

```
      1111
     0011010
+    0001111
   ---------
    0101001
```

One-bit (full) adder

Need to include:

Carry-in & Carry-out

A	B	Cin	Sum	Cout
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 & Carry-out

A	B	Cin	Sum	Cout
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^B) | C_{in} & ~(A^B) == (C_{in} ^(A^B))

- When is C_{out} 1?

One-bit (full) adder

Need to include:

Carry-in & Carry-out

A	B	Cin	Sum	Cout
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^B) | C_{in} & ~(A^B) == (C_{in} ^(A^B))

- When is C_{out} 1?

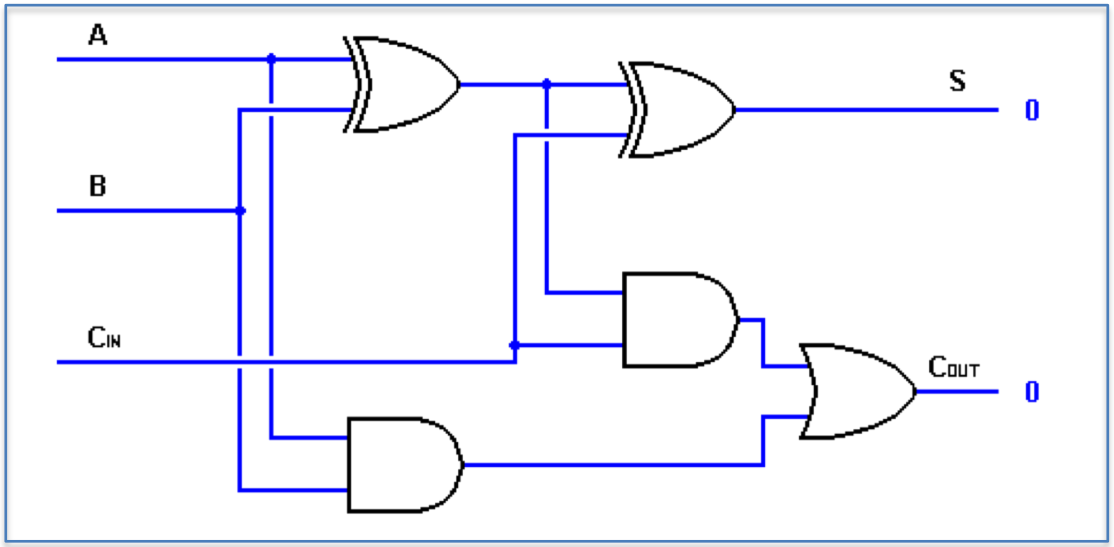
(A & B) | ((A^B) & C_{in})

One-bit (full) adder

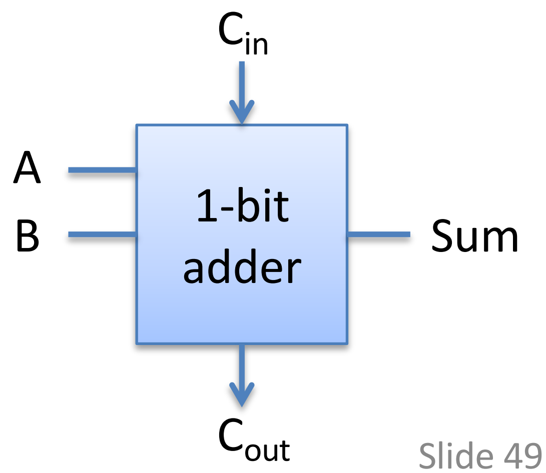
Need to include:

Carry-in & Carry-out

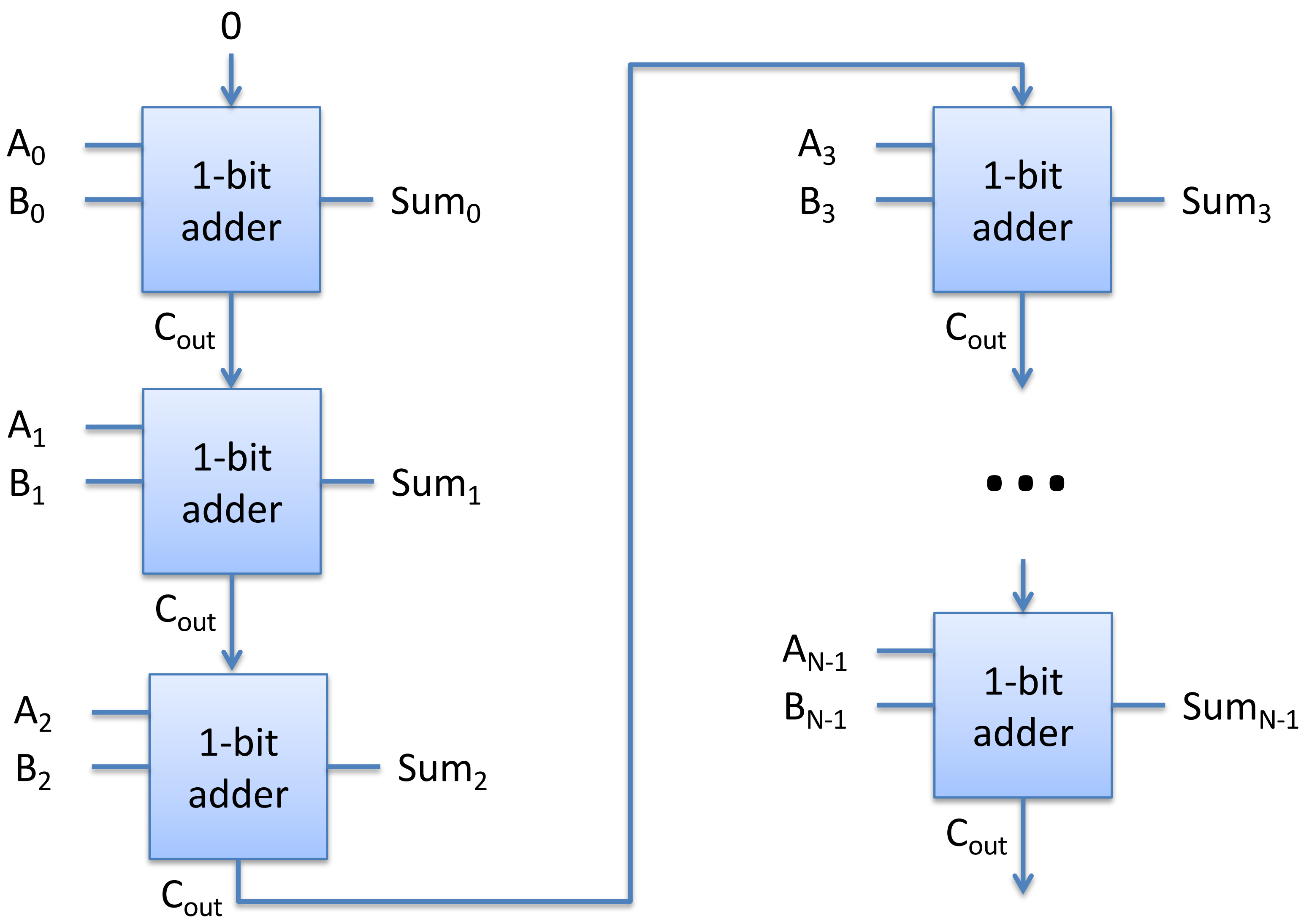
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



=



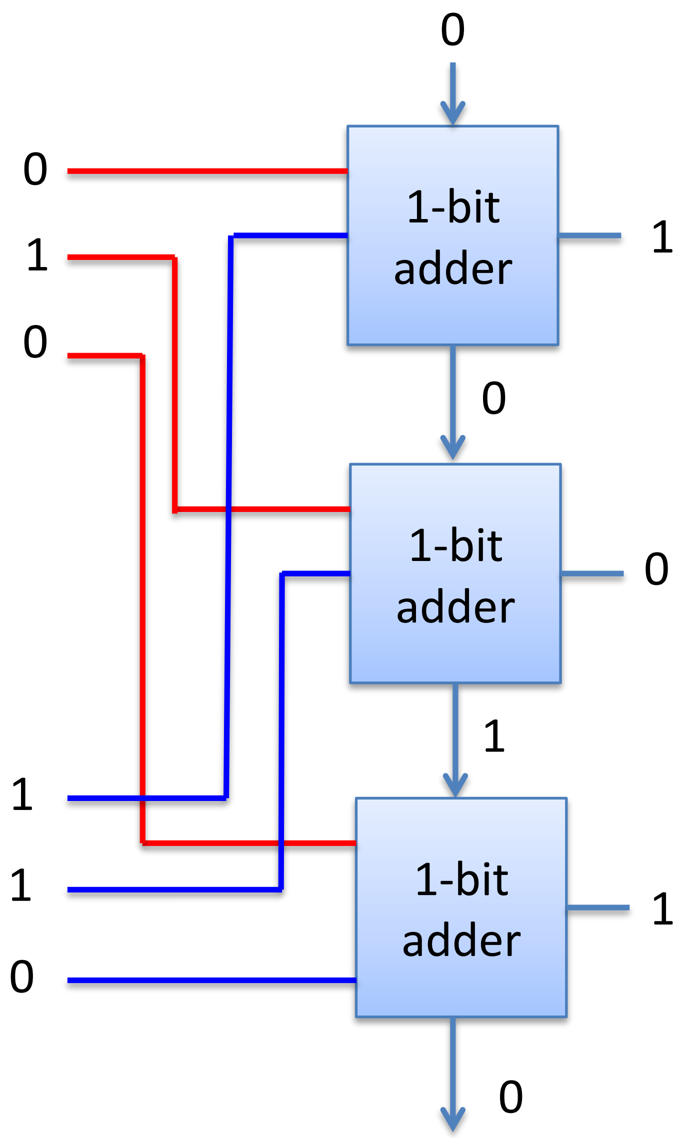
Multi-bit Adder (Ripple-carry Adder)



Three-bit Adder (Ripple-carry Adder)

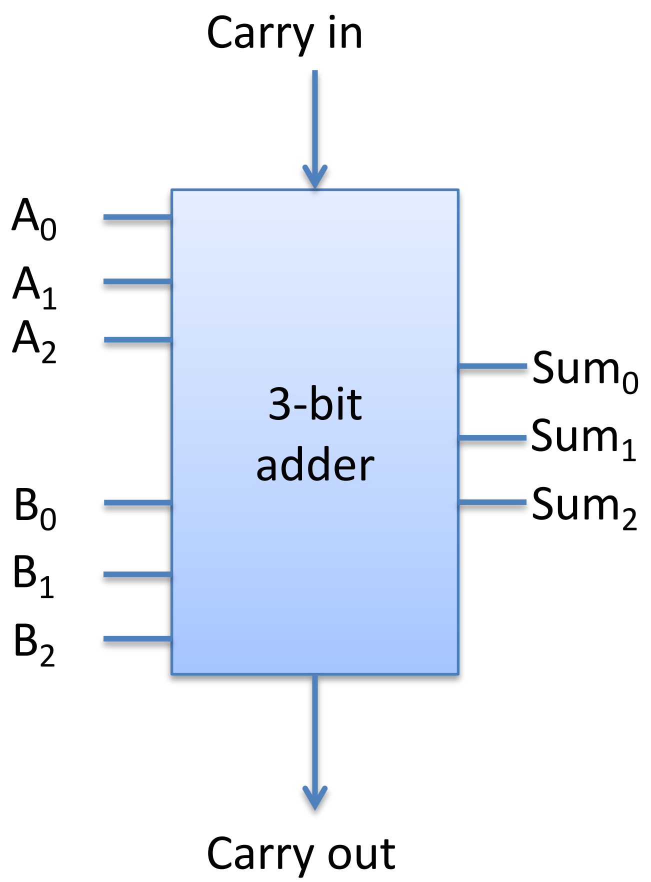
010

$+\ 011$



=101

=

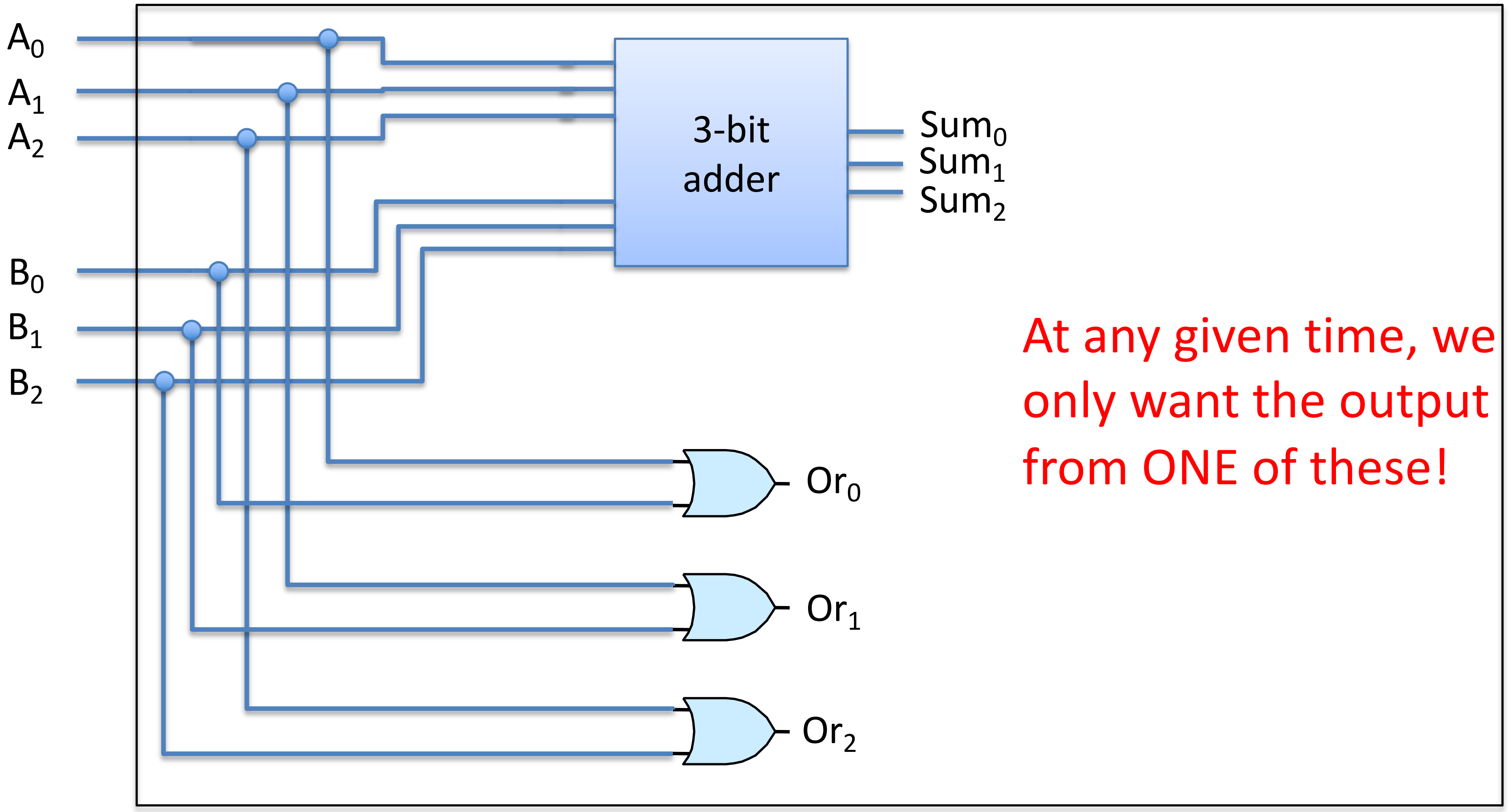


Arithmetic Logic Unit (ALU)

- 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)

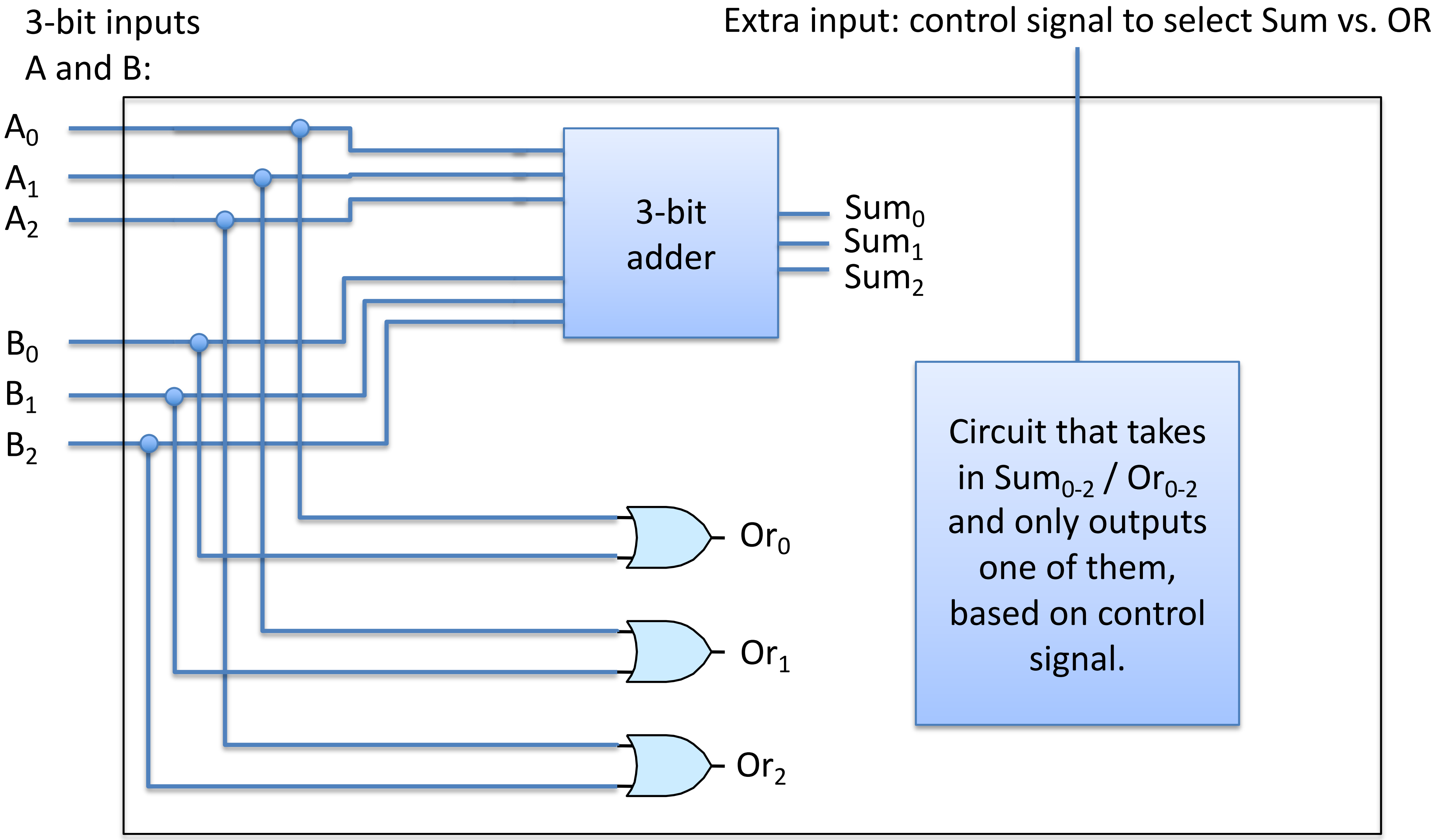
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:



At any given time, we only want the output from ONE of these!

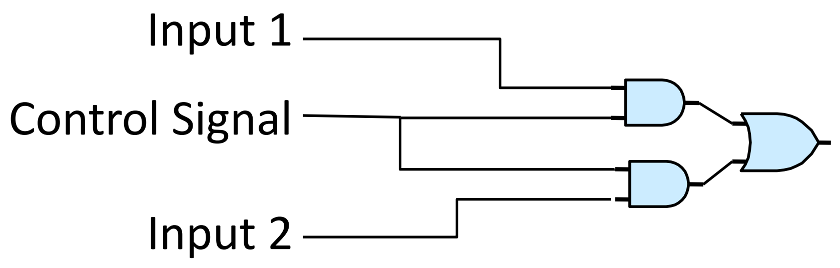
Simple 3-bit ALU: Add and bitwise OR



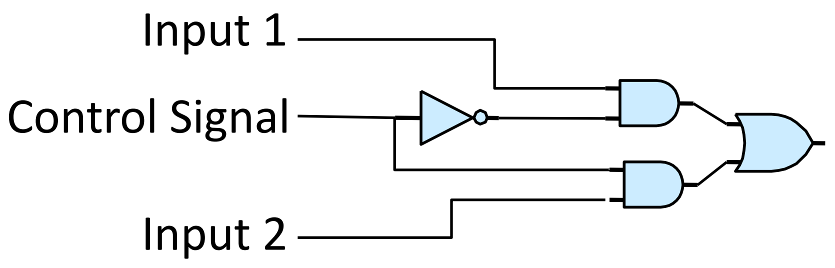
3-bit inputs A and B:
Extra input: control signal to select Sum vs. OR
A_0
A_1
A_2
B_0
B_1
B_2
3-bit adder
Sum_0
Sum_1
Sum_2
Or_0
Or_1
Or_2
Circuit that takes in Sum_{0-2} / Or_{0-2} and only outputs one of them, based on control signal.

Which of these circuits lets us select between two inputs?

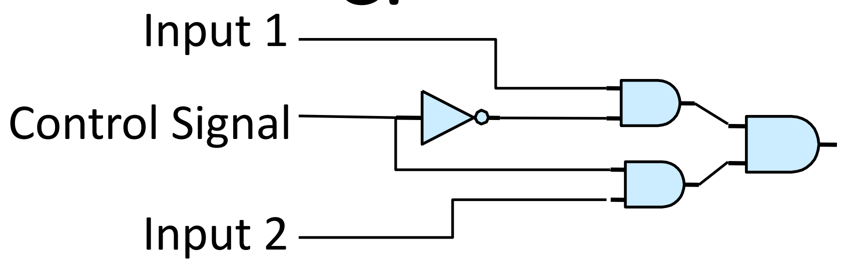
A:



B:

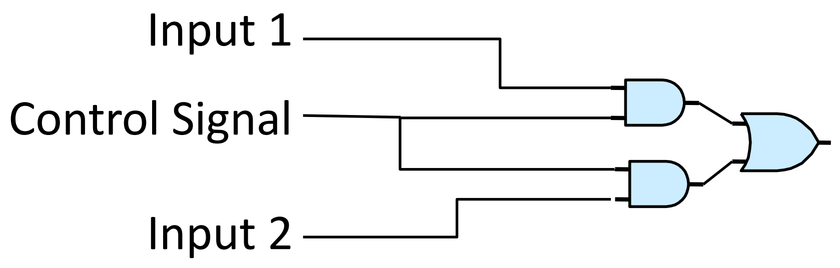


C:

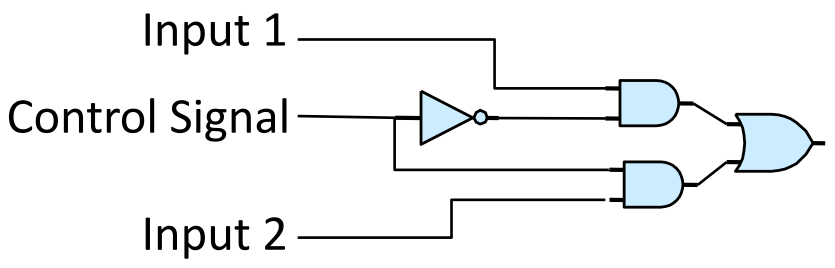


Which of these circuits lets us select between two inputs?

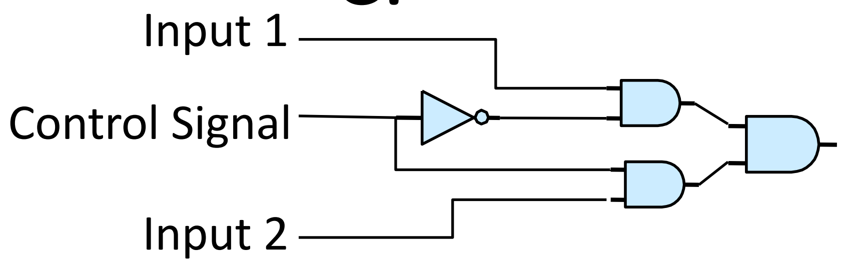
A:



B:



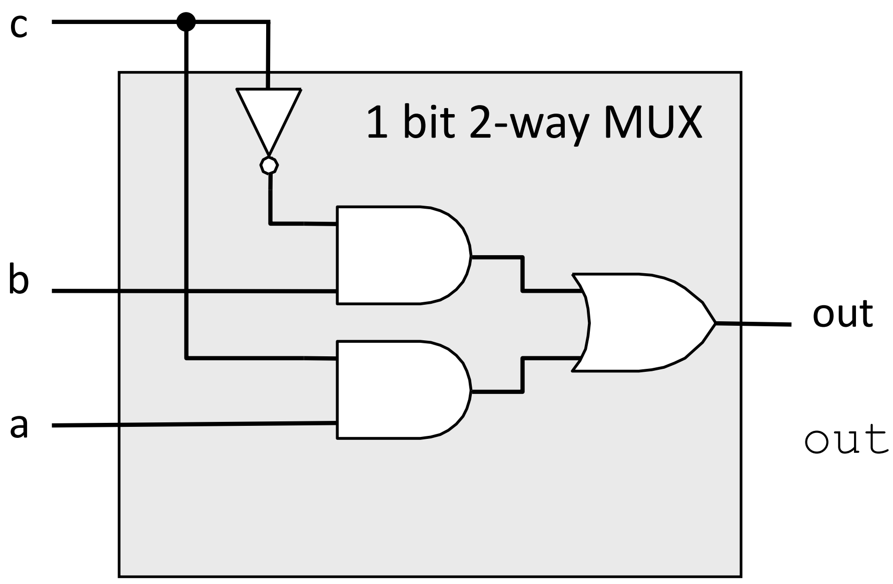
C:



Multiplexor: Chooses an input value

Inputs: 2^N data inputs, N signal bits

Output: is one of the 2^N input values

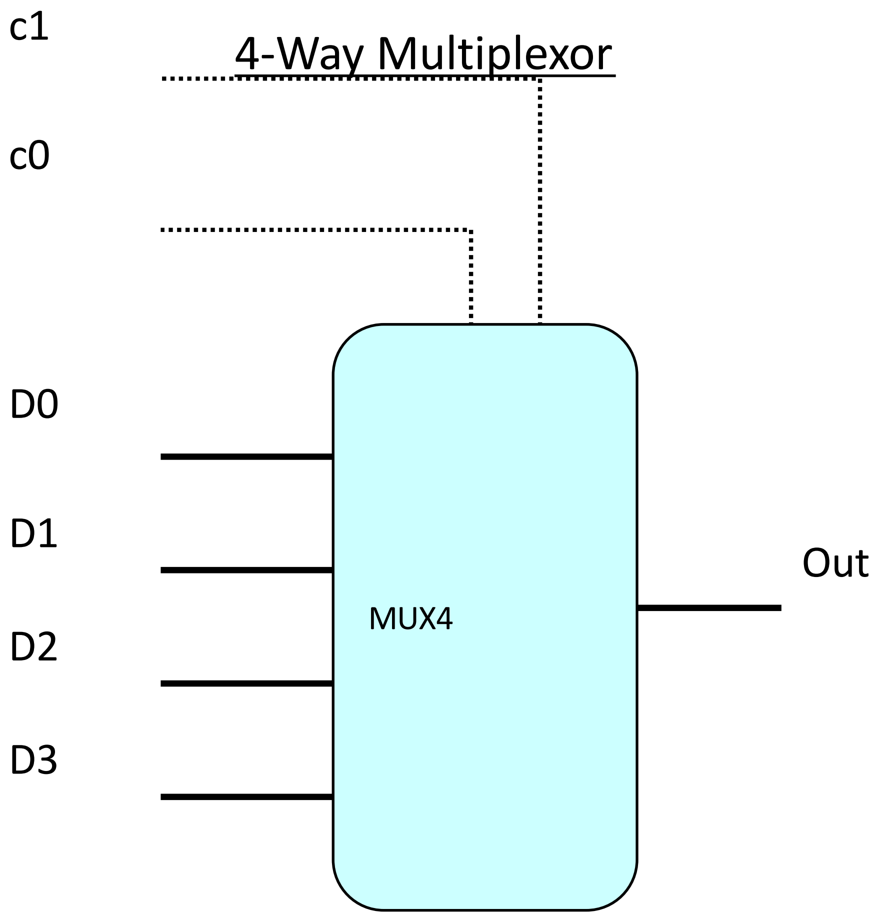


```
out = (c & a)|(~c &b)
```

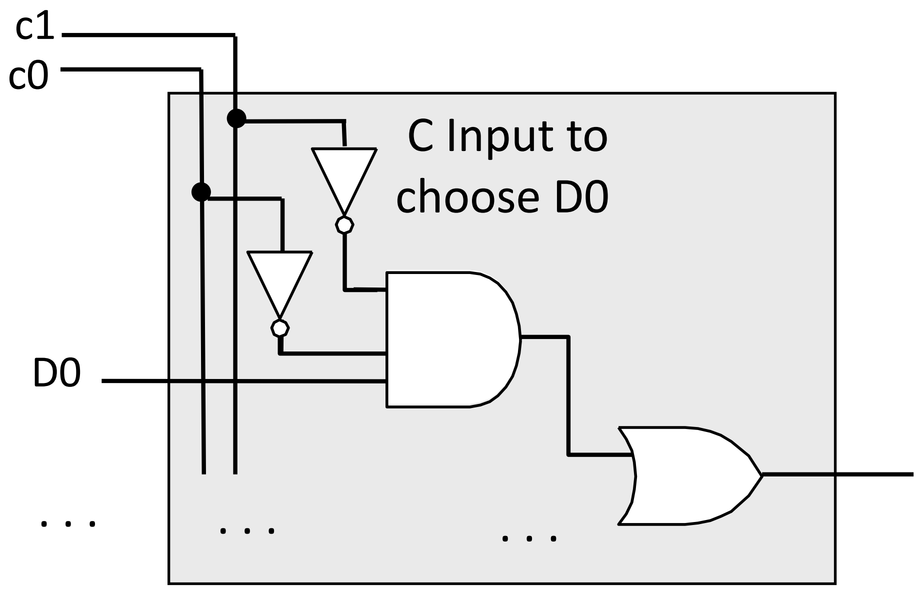
- Control signal c, chooses the input for output
  - When c is 1: choose a, when c is 0: choose b

N-Way Multiplexor

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

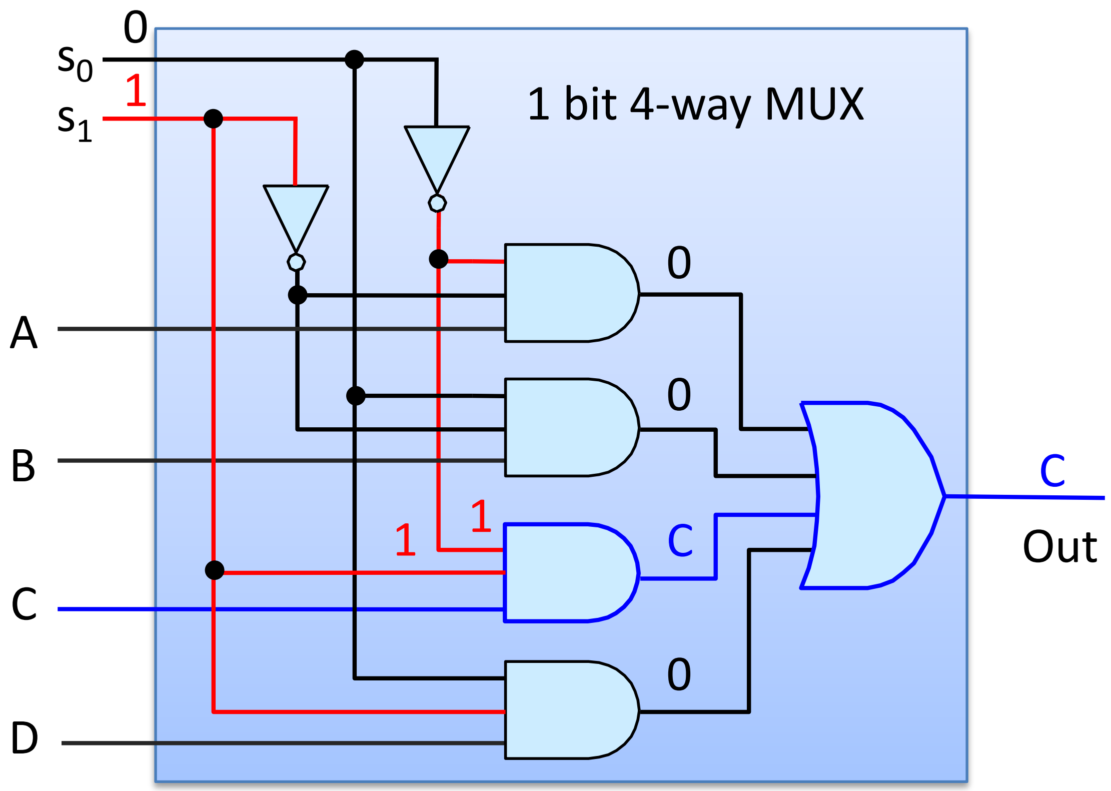


c_1	c_2	Output
0	0	D0
0	1	D1
1	0	D2
1	1	D3

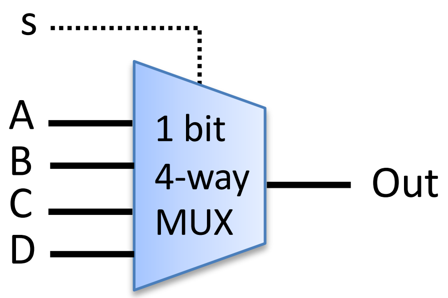


Example, 1-bit 4-way MUX

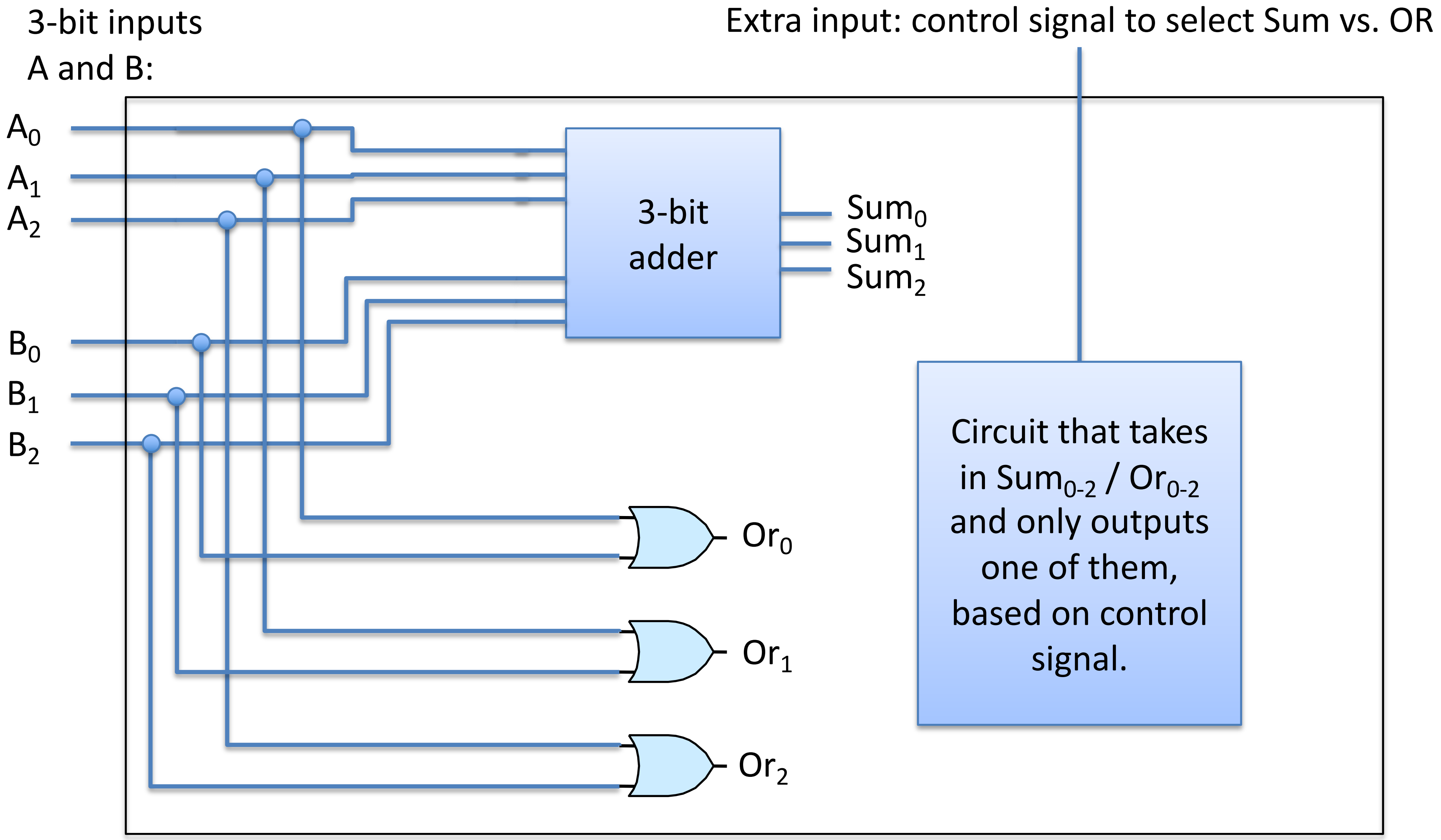
When select input is 2 (0b10): C chosen as output



S	Ou t
0	A
1	B
2	C
3	D



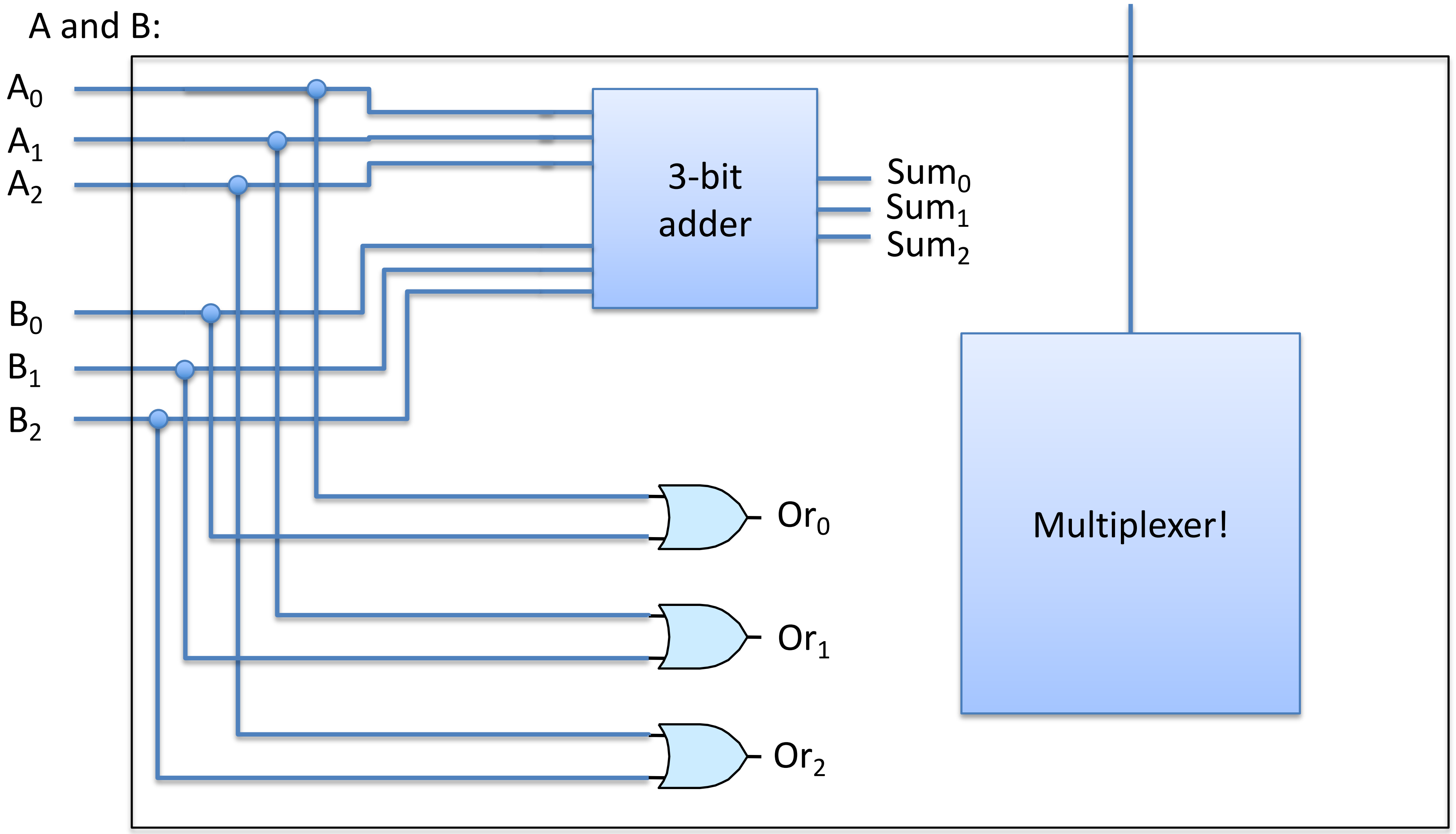
Simple 3-bit ALU: Add and bitwise OR


3-bit inputs
A and B:
Extra input: control signal to select Sum vs. OR
A_0
A_1
A_2
B_0
B_1
B_2
3-bit adder
Sum_0
Sum_1
Sum_2
Or_0
Or_1
Or_2
Circuit that takes in Sum_{0-2} / Or_{0-2} and only outputs one of them, based on control signal.

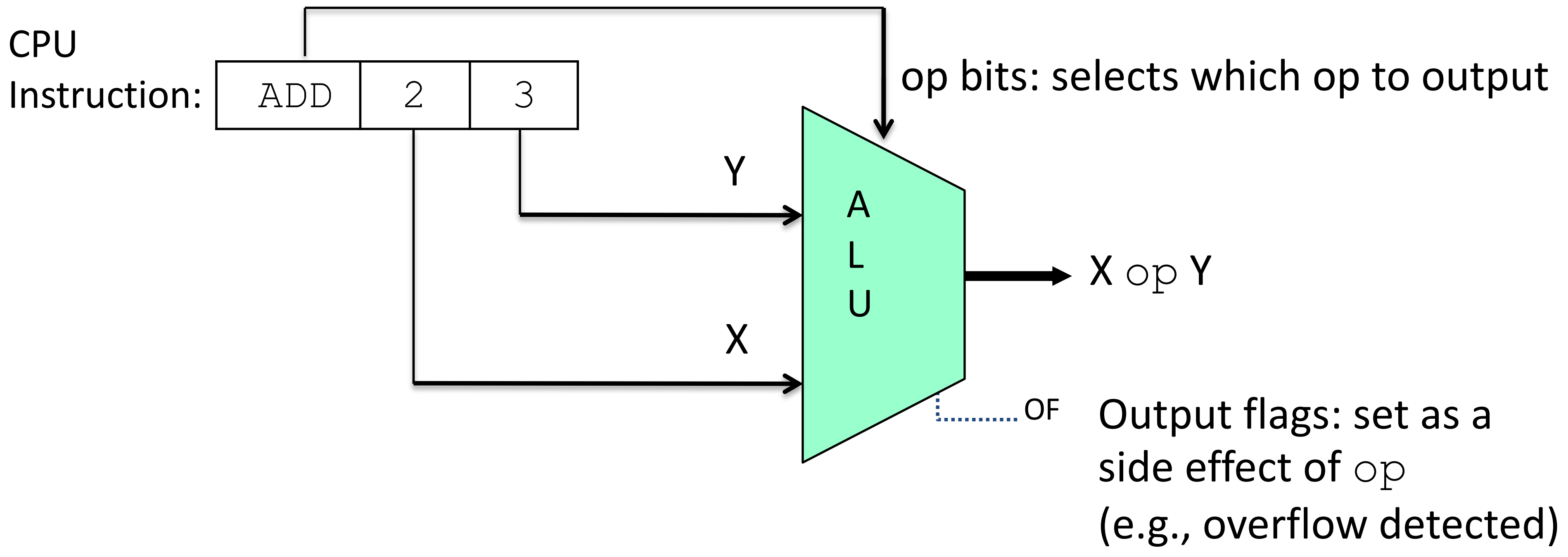
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:

Extra input: control signal to select Sum vs. OR



ALU: Arithmetic Logic Unit


CPU
Instruction:
ADD
2
3
op bits: selects which op to output
Y
X
A
L
U
X op Y
OF
Output flags: set as a side effect of op (e.g., overflow detected)

- 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

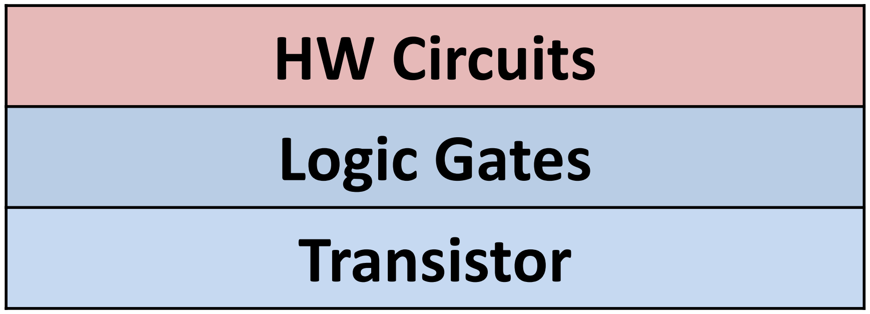
2. Storage: to store binary values

   (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



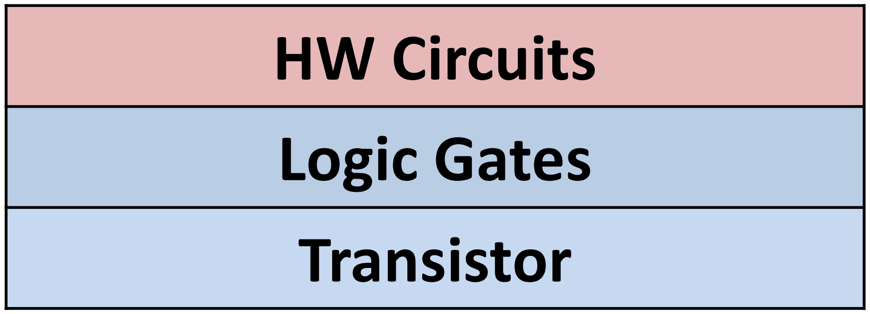
Digital Circuits - Building a CPU

Three main classifications of HW circuits:

2. Storage: to store binary values

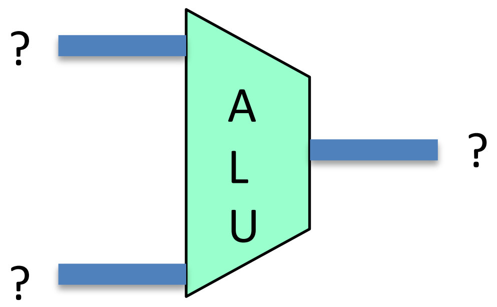
    (ex) Register File: set of CPU registers

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



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

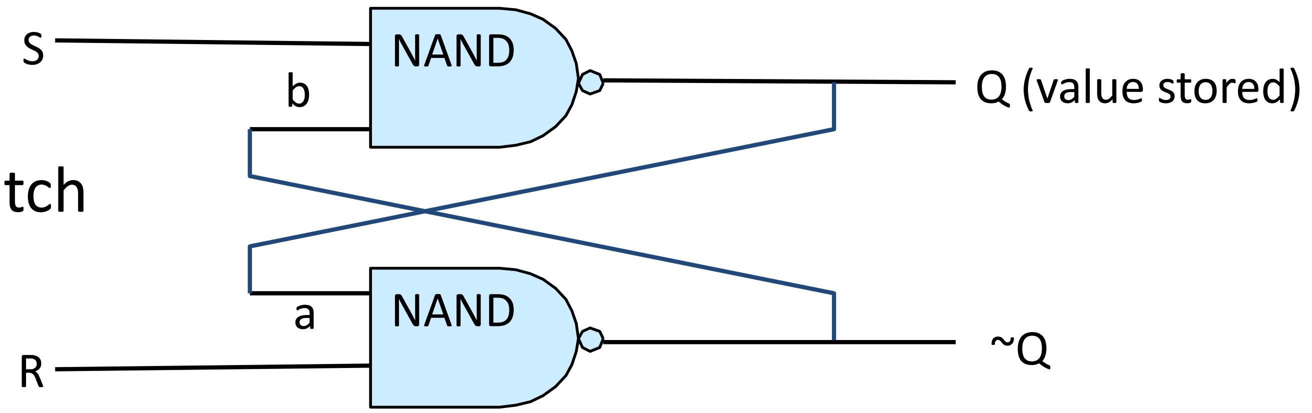
- Store a 0 or 1
- Retrieve the 0 or 1 value on demand (read)
- Set the 0 or 1 value on demand (write)

R-S (Reset-Set) Latch: Stores Value Q

When R and S are both 1: Maintain a value

R and S are never both simultaneously 0

R-S Latch

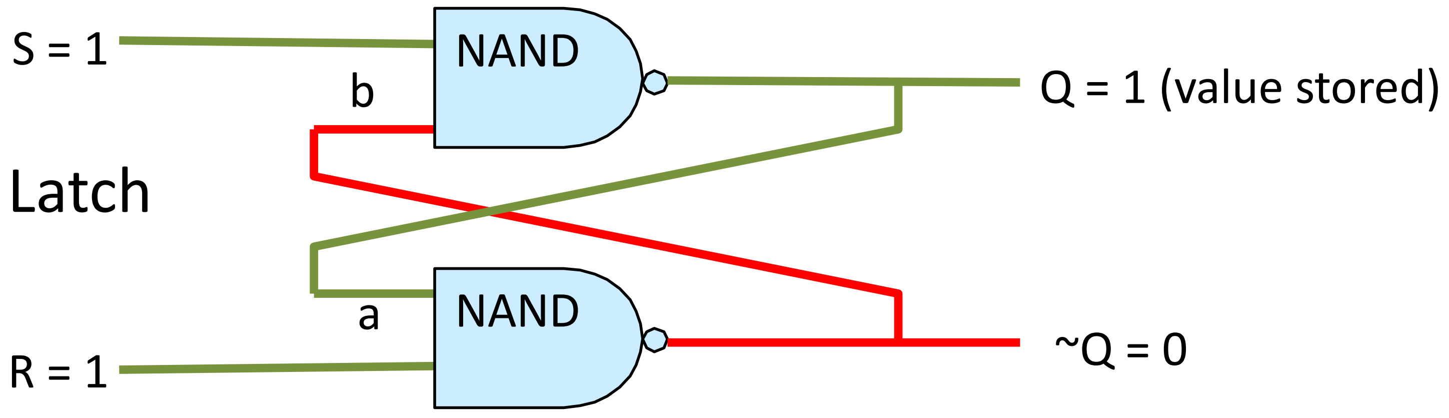


- To write a new value:
  - Set S to 0 momentarily (R stays at 1): to write a 1
  - Set R to 0 momentarily (S stays at 1): to write a 0

R-S (Reset-Set) Latch: Stores Value Q

When R and S are both 1: Maintain a value

R and S are never both simultaneously 0

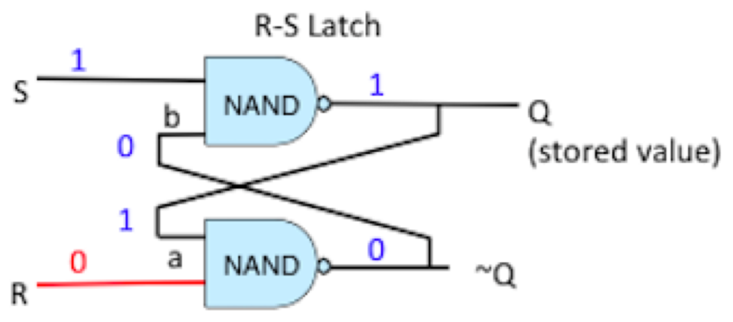


- To write a new value:
  - Set S to 0 momentarily (R stays at 1): to write a 1
  - Set R to 0 momentarily (S stays at 1): to write a 0

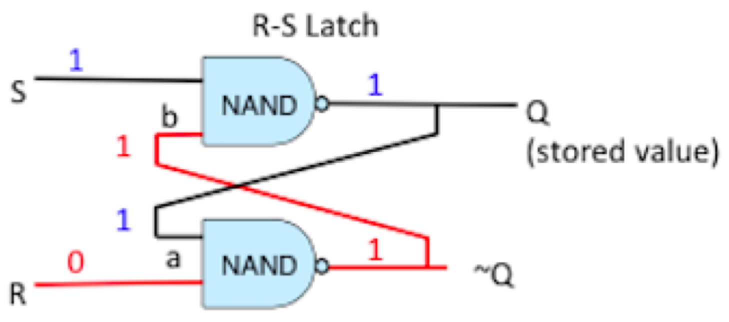
R-S (Reset-Set) Latch: Stores Value Q

To write 0 to an RS latch momentarily set R to 0

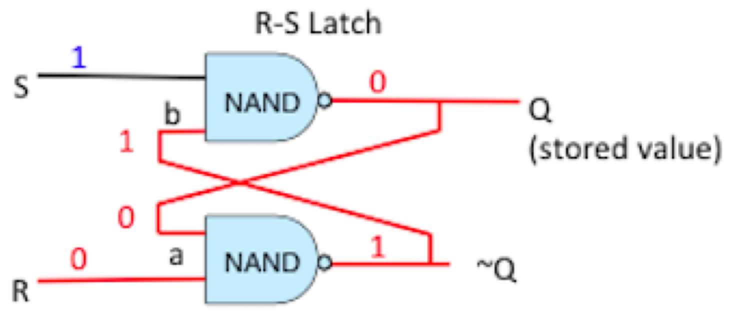
R and S are never both simultaneously 0



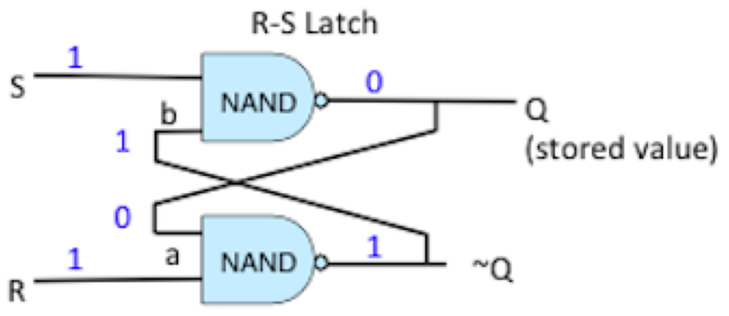
A. Set R to 0 to store 0



B. Changes lower NAND output to 1



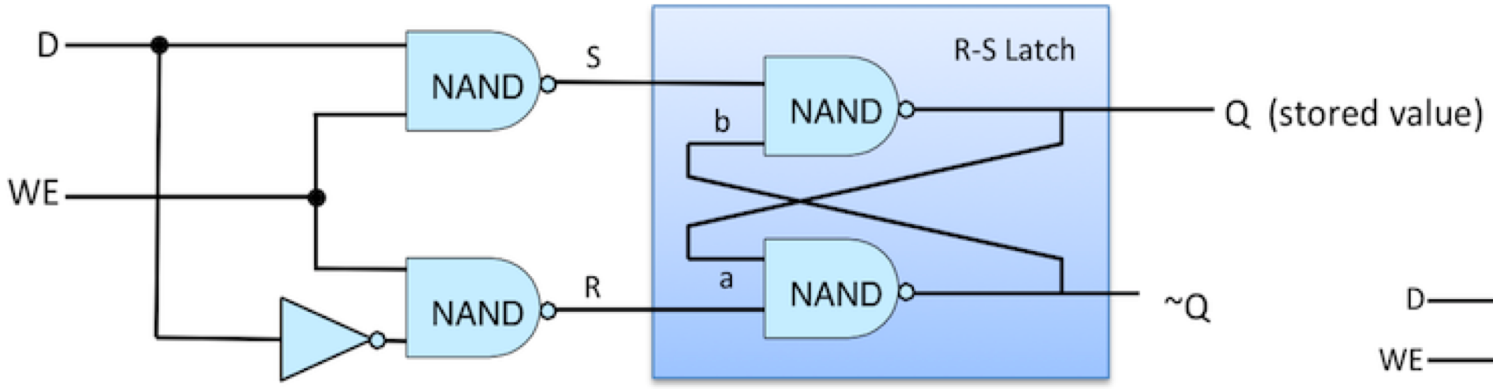
C. Changes upper NAND output to 0

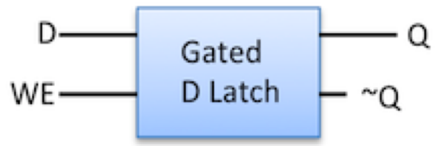


D. R-S Latch Now Stores 0
(R can be set back to 1 and still stores 0)

Gated D Latch

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





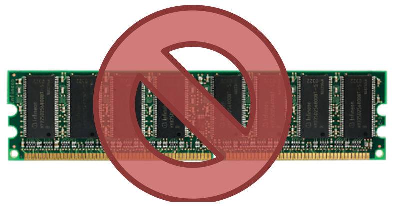
D: into top NAND, ~D into bottom NAND

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

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

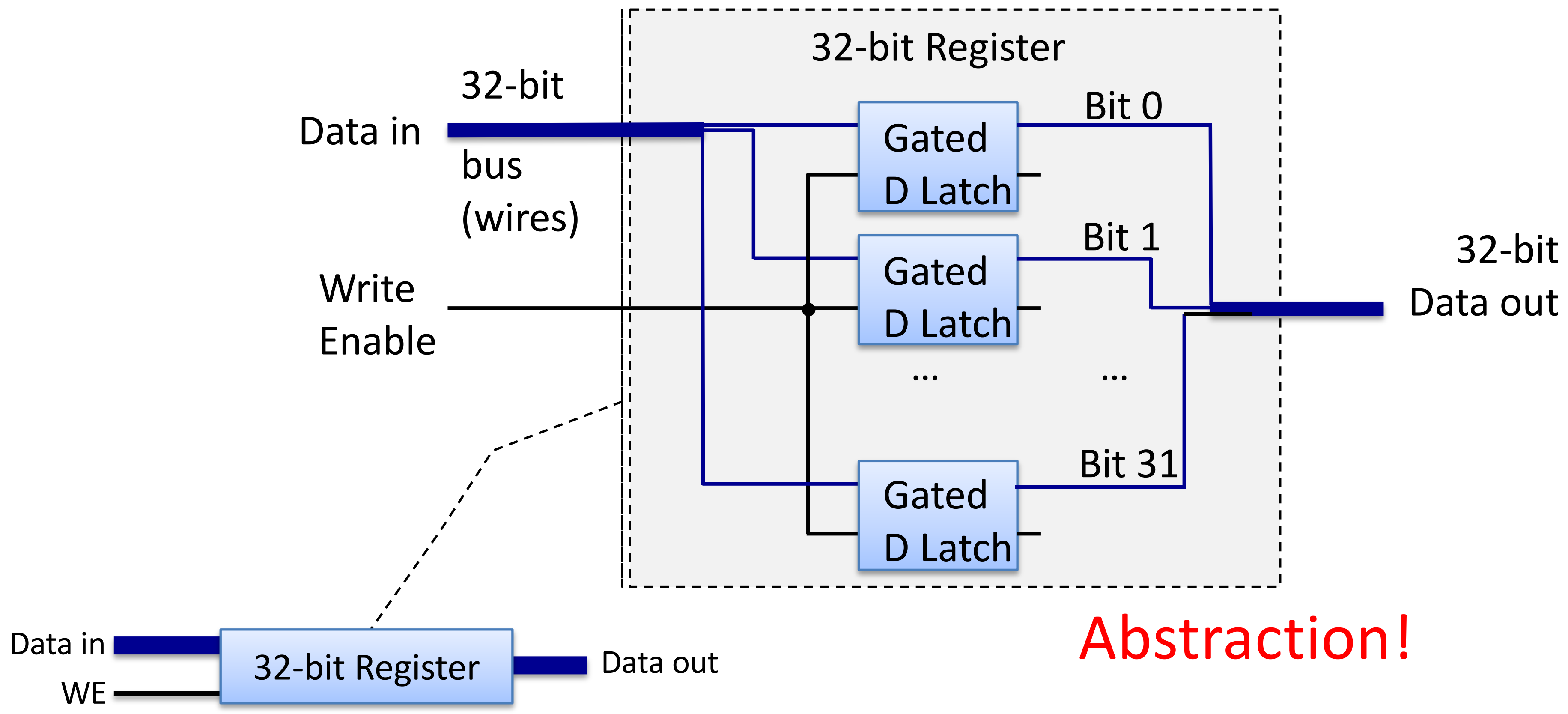
Fast, not very dense, expensive

DRAM: capacitor-based:



An N-bit Register

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

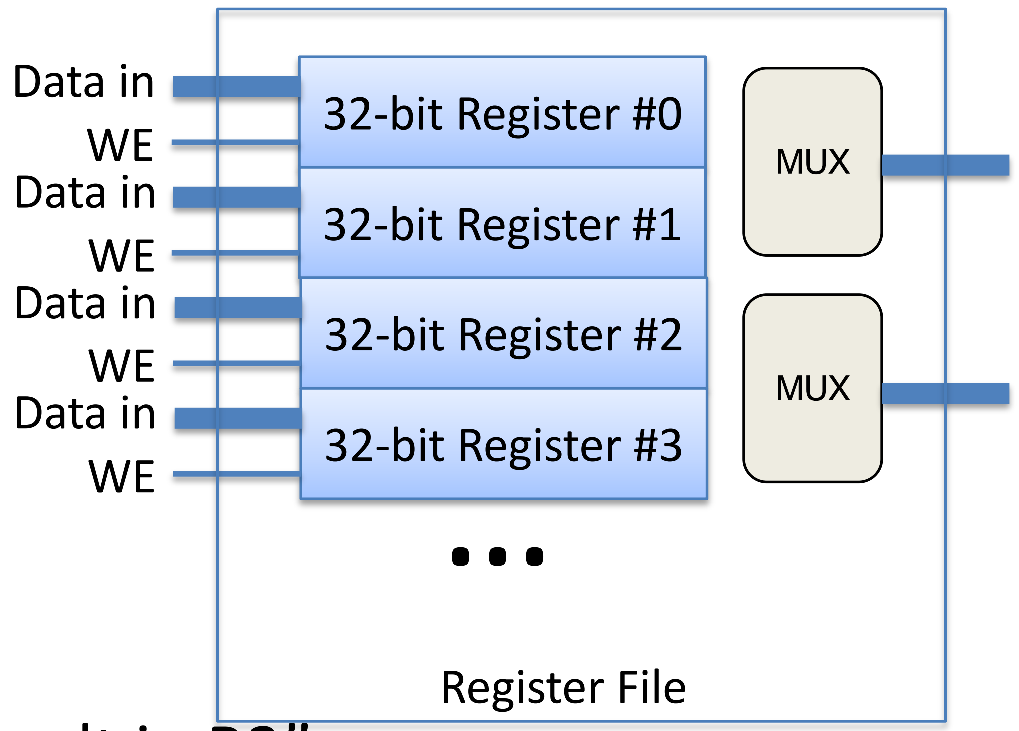


Abstraction!

"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

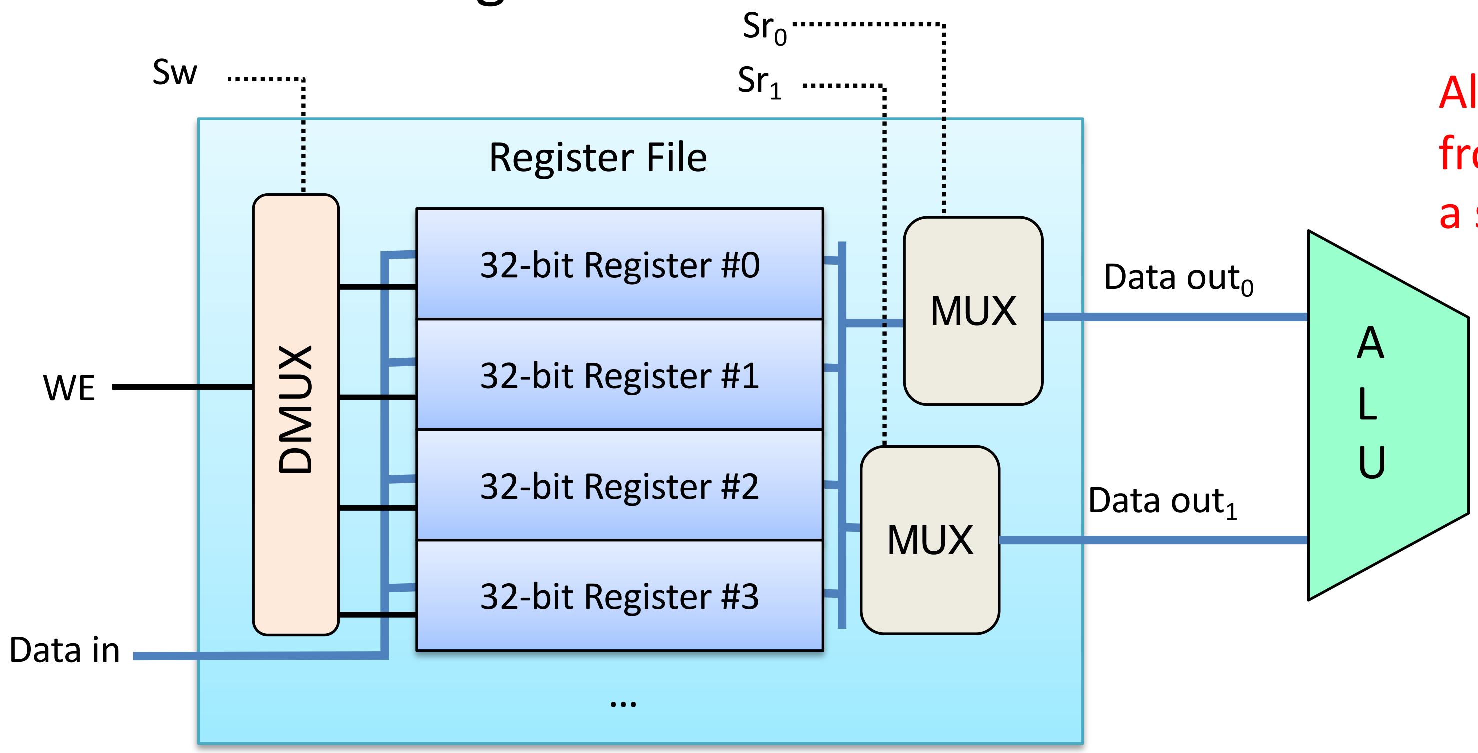
- Lots of abstraction going on here!
  - Gates hide the details of transistors.
  - Build R-S Latches out of gates to store one bit.
  - Combining multiple latches gives us N-bit register.
  - Grouping N-bit registers gives us register file.

- 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?

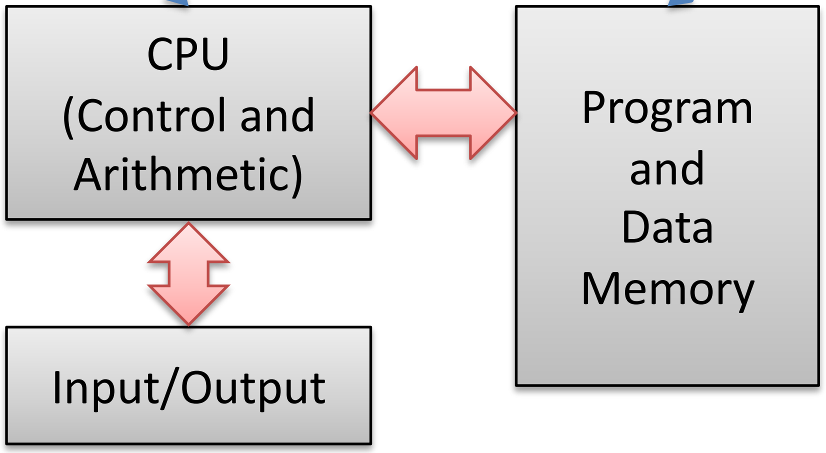


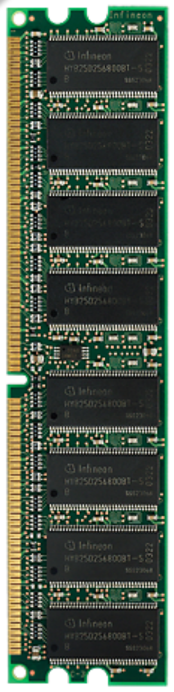
All this info comes from the program: a series of instructions.

Recall: Von Neumann Model

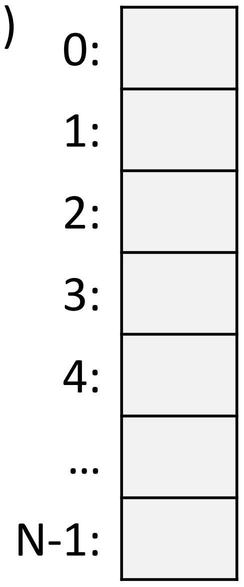
We're building this.

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





Mem Addresses (buckets)



Digital Circuits - Building a CPU

Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality

    (ex) adder to add two values together

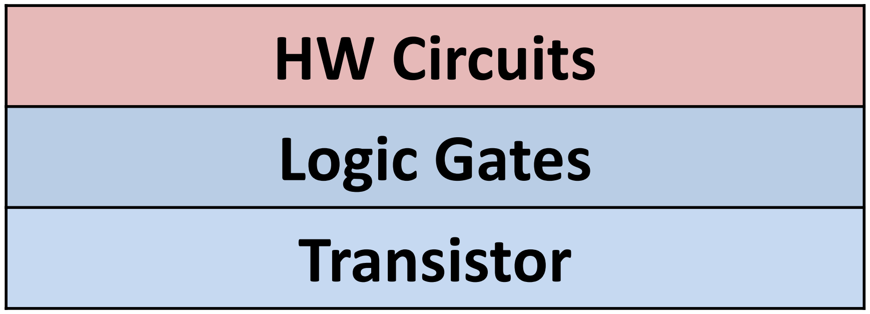
2. Storage: to store binary values

    (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



Digital Circuits - Building a CPU

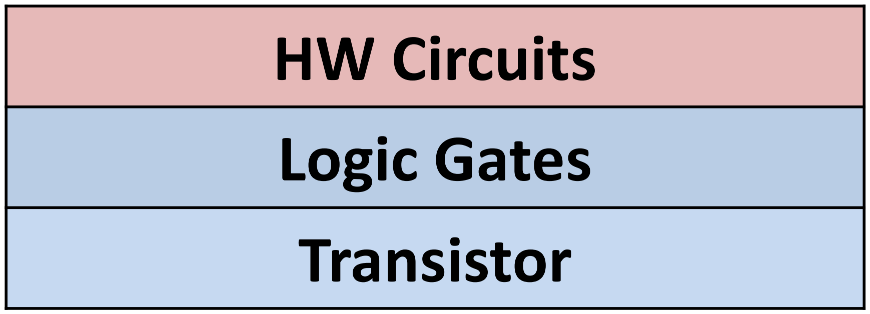
Three main classifications of HW circuits:

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.

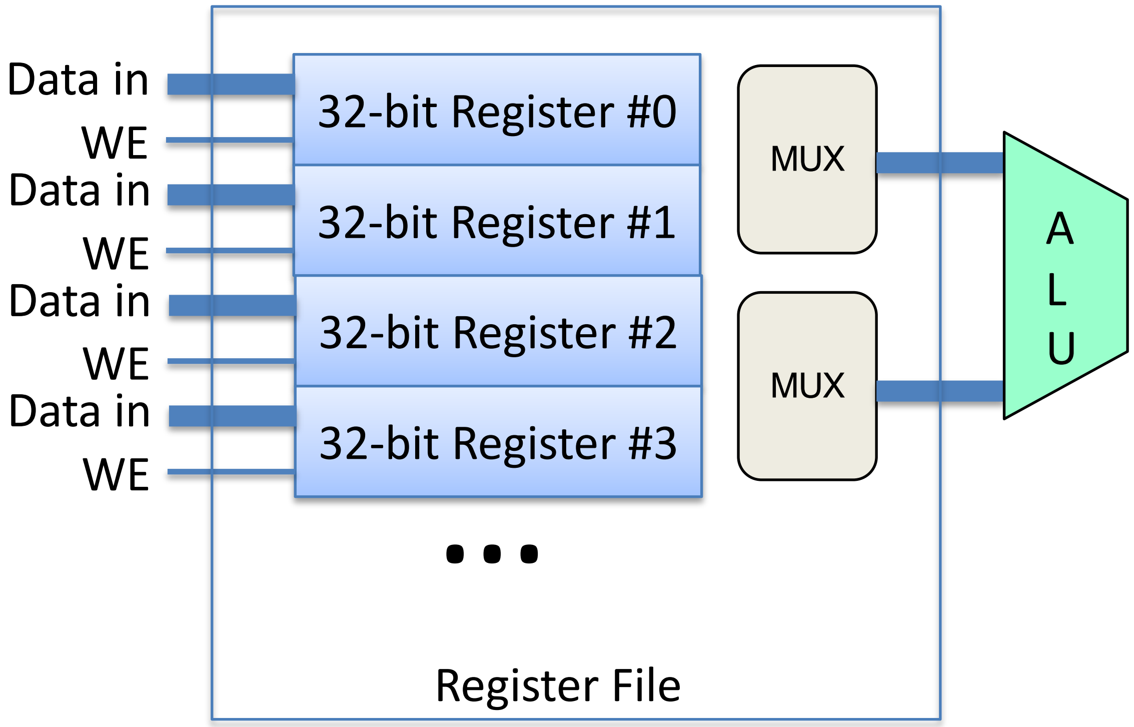


Control Unit

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

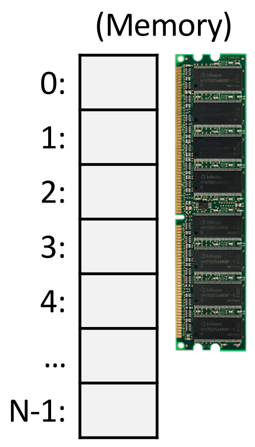
- Fetch instruction from memory
- Decode what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands
- Execute the instruction (arithmetic, etc.)
- Store the result

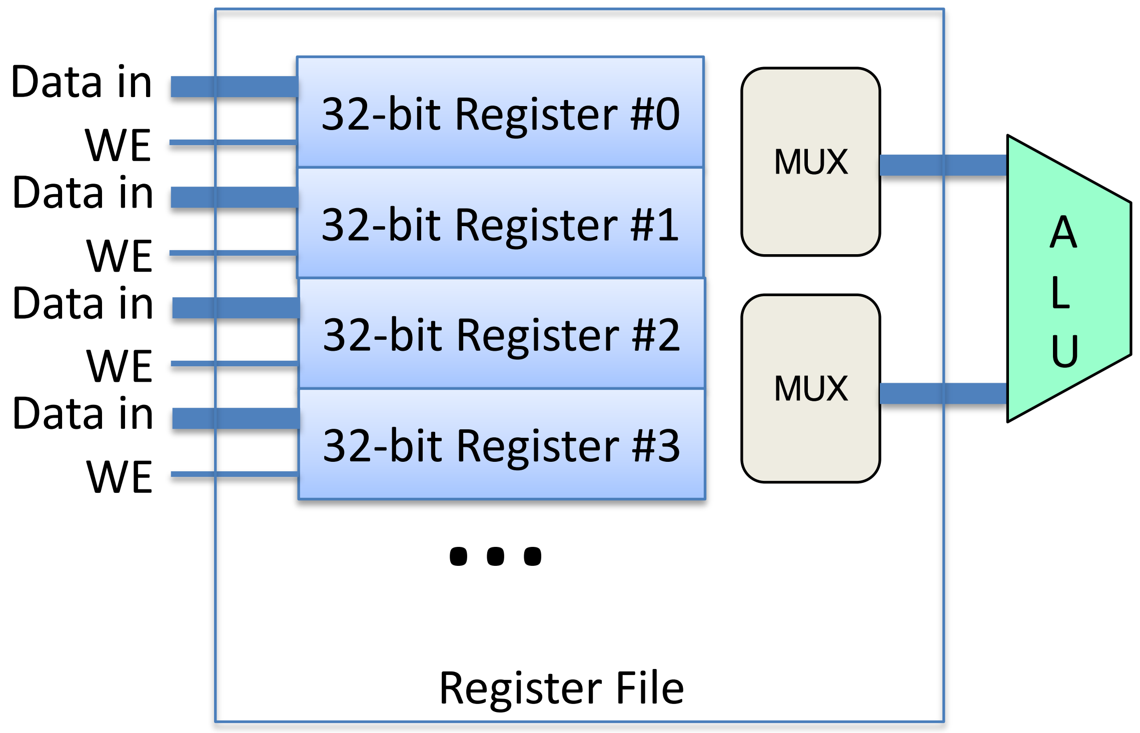
Program State

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)**



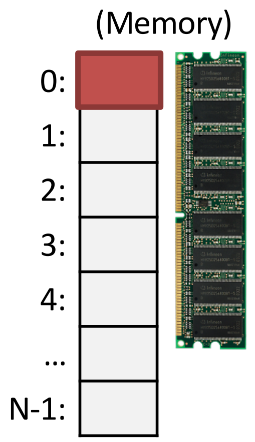


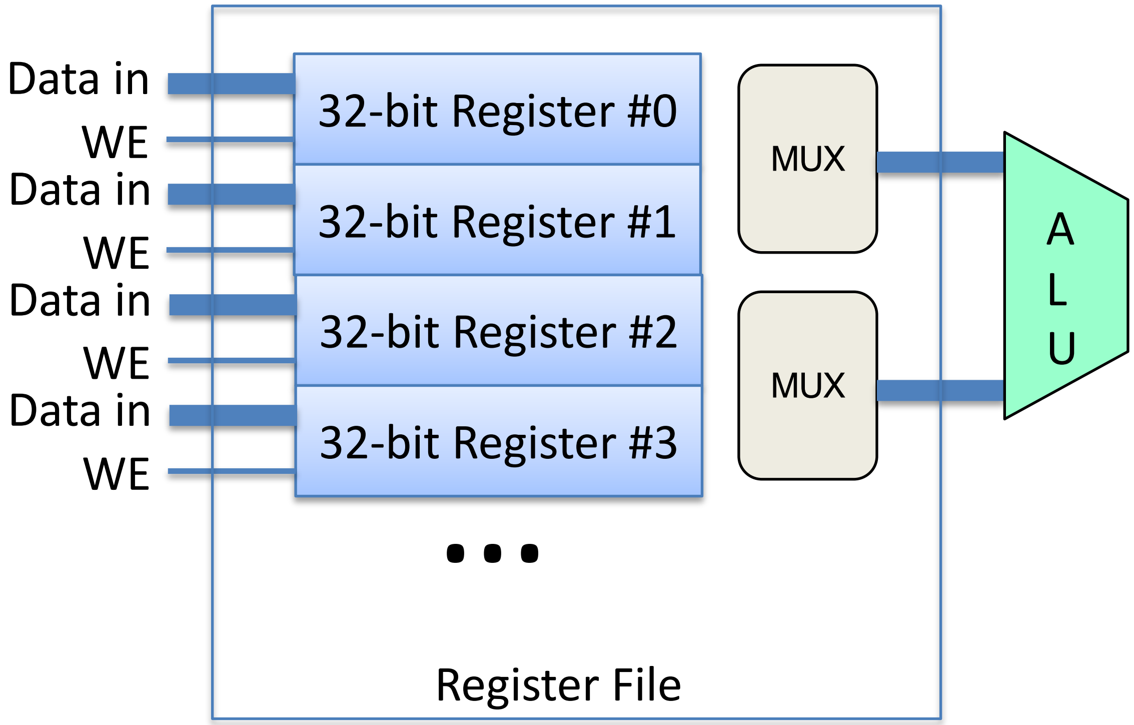
Fetching instructions.

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

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

Instruction Register (IR): Instruction at Address 0



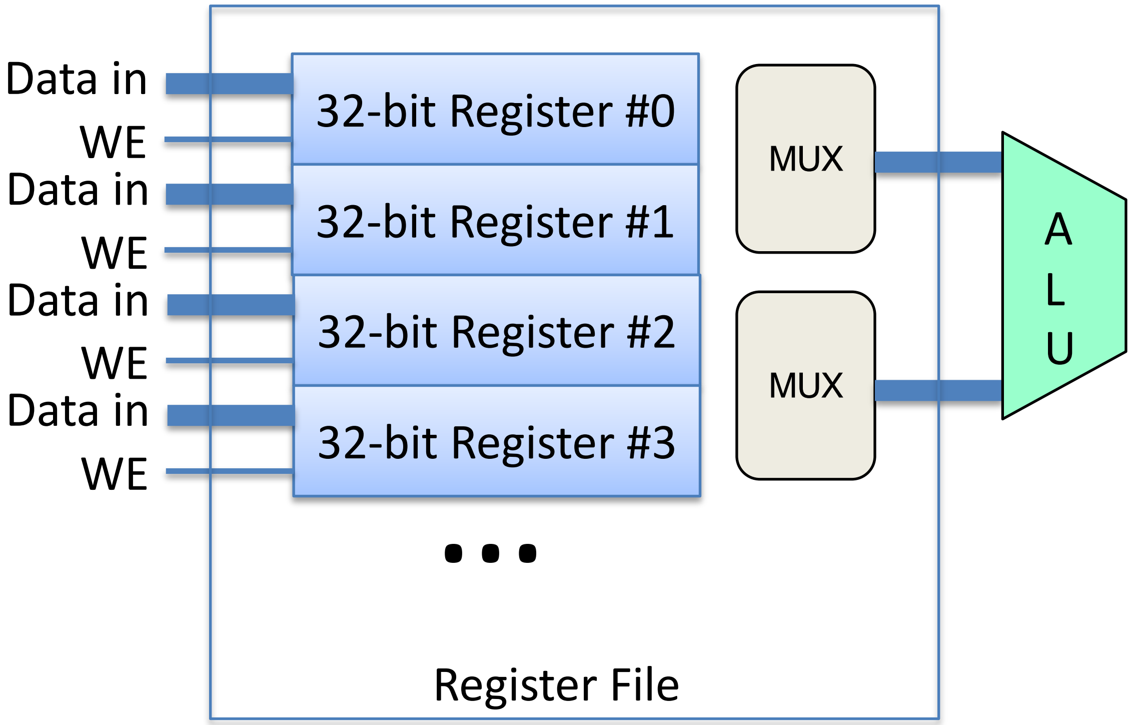


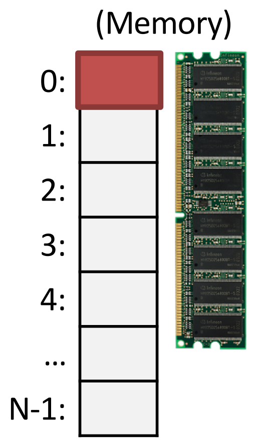
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

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

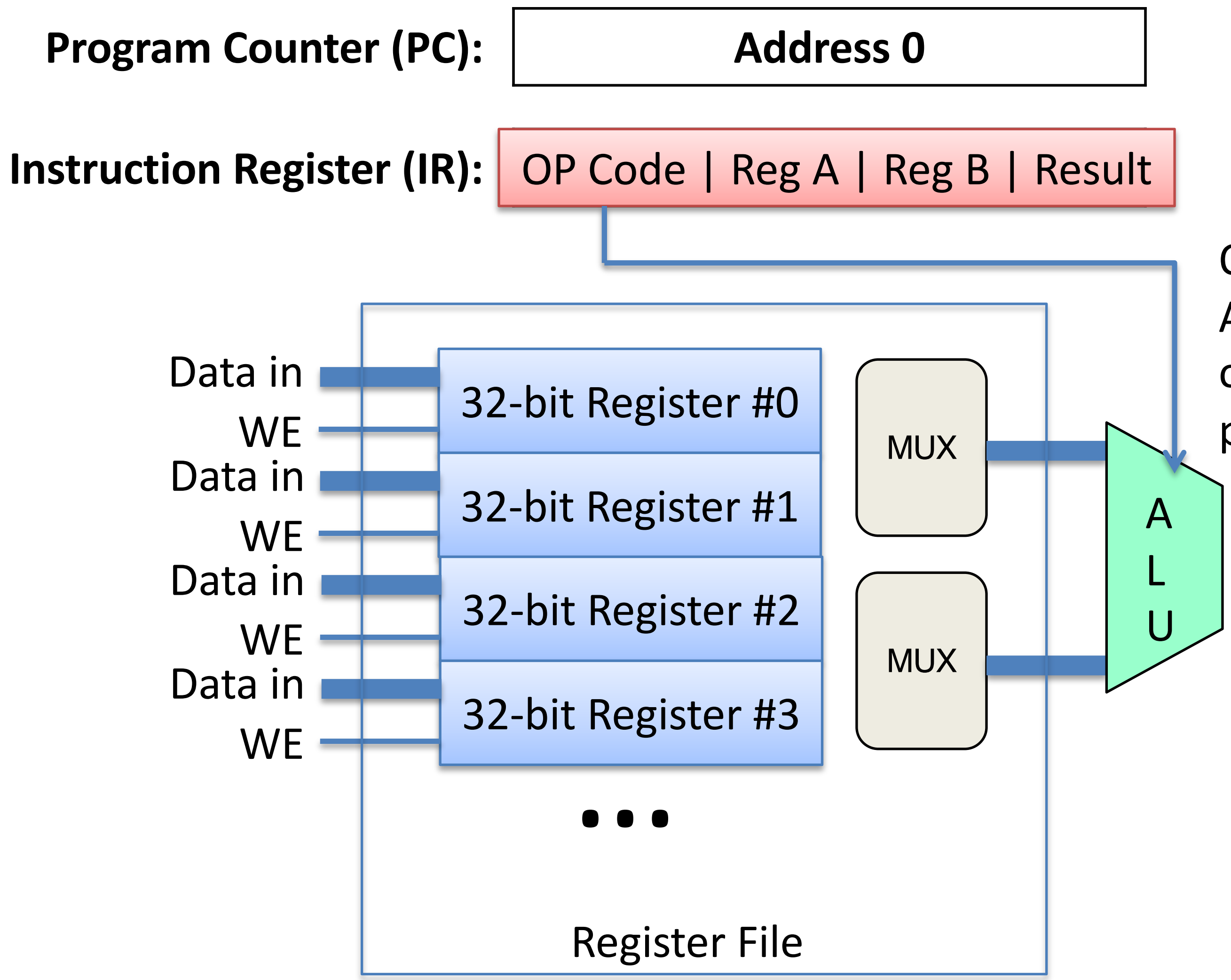
Instruction Register (IR): OP Code | Reg A | Reg B | Result



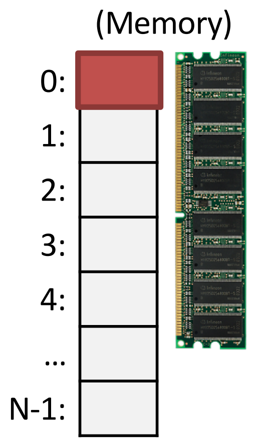


Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

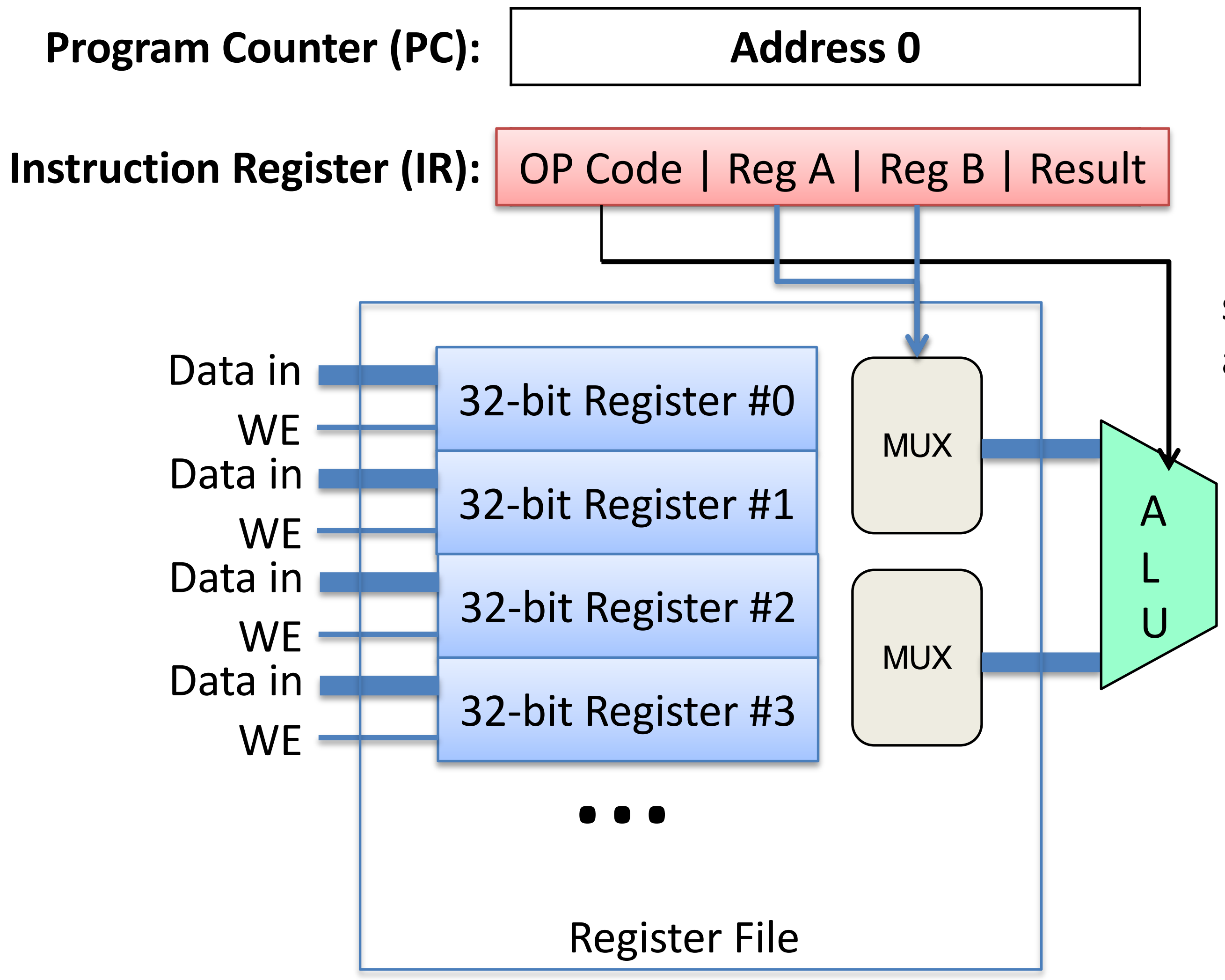


OP Code tells ALU which operation to perform.

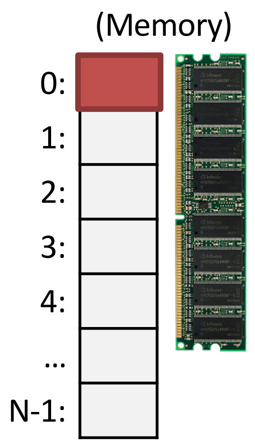


Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

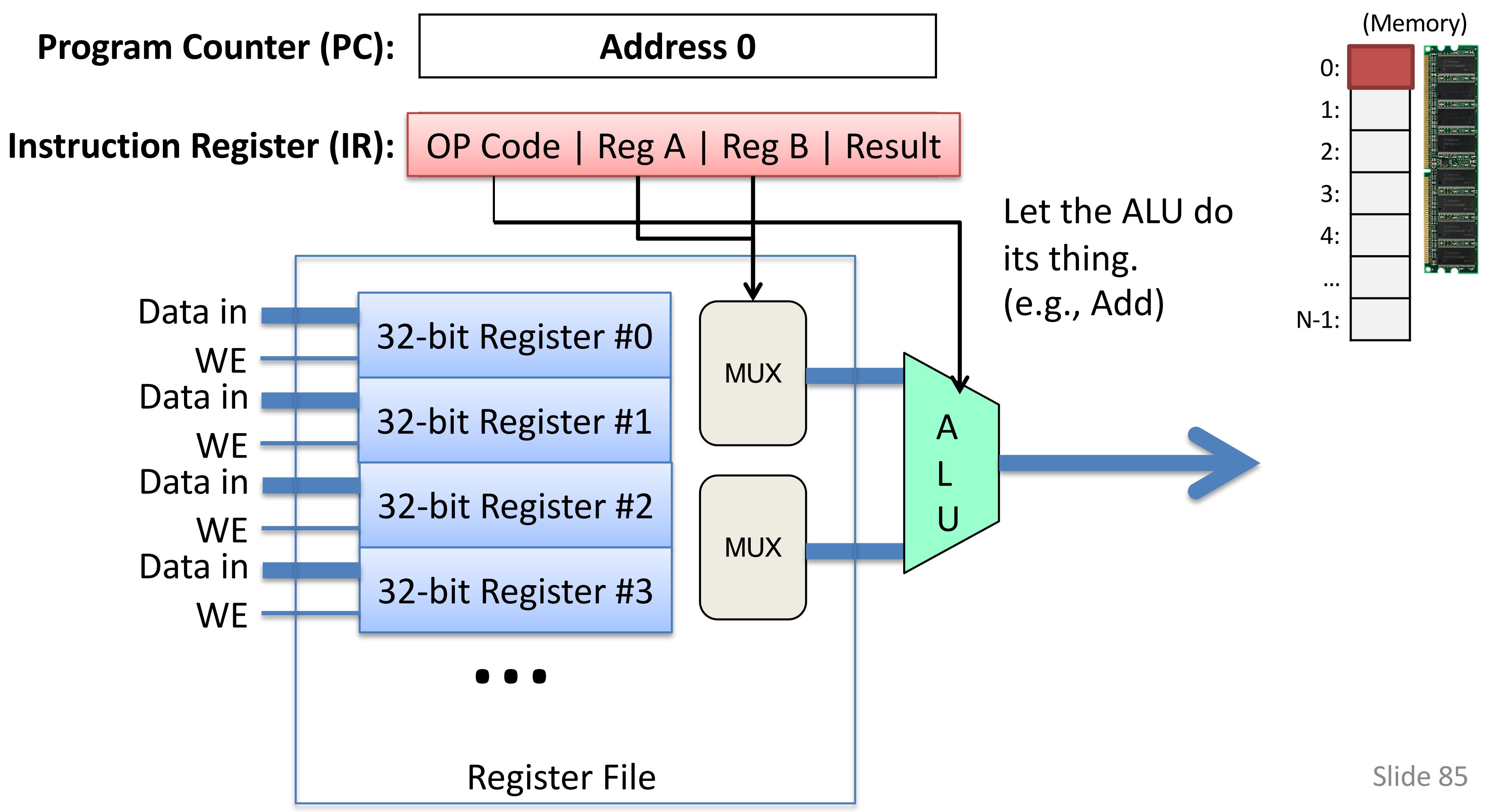


Register ID #'s specify input arguments.



Executing instructions.

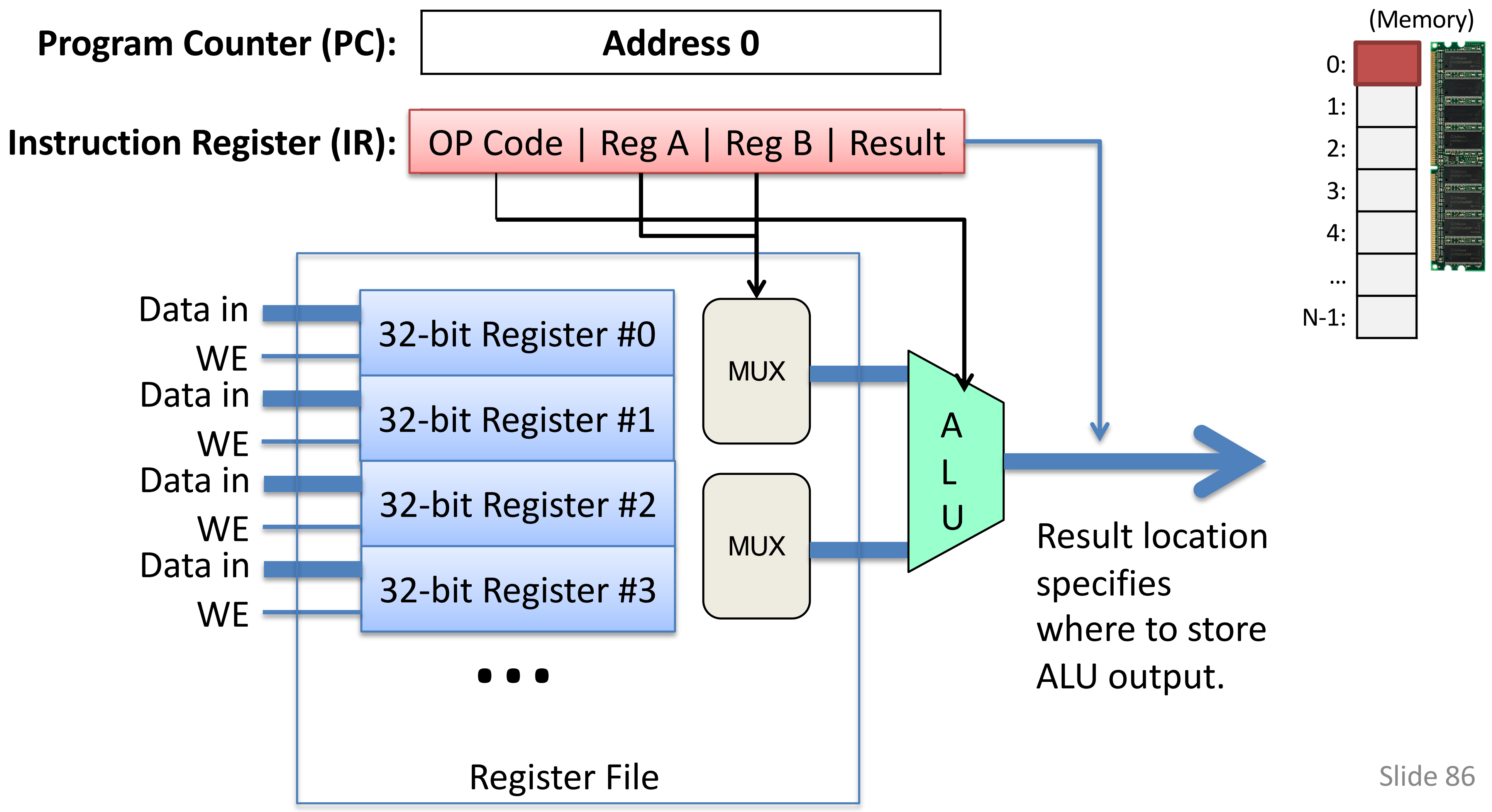
Interpret the instruction bits: What operation? Which arguments?



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

Storing results.

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



Result location specifies where to store ALU output.

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.

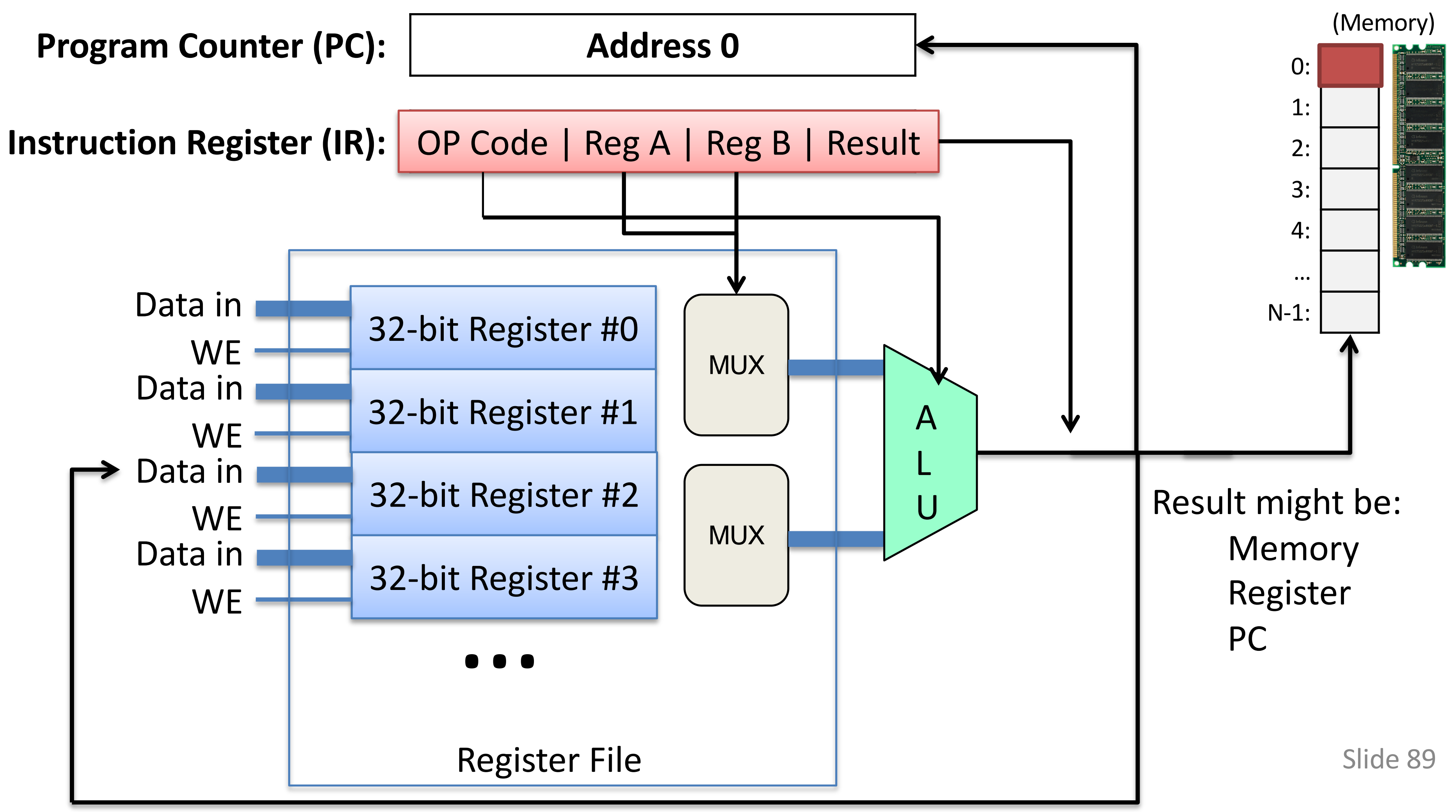
B. Some instructions might skip the PC forward by more than one.

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?



Result might be:
- Memory
- Register
- PC

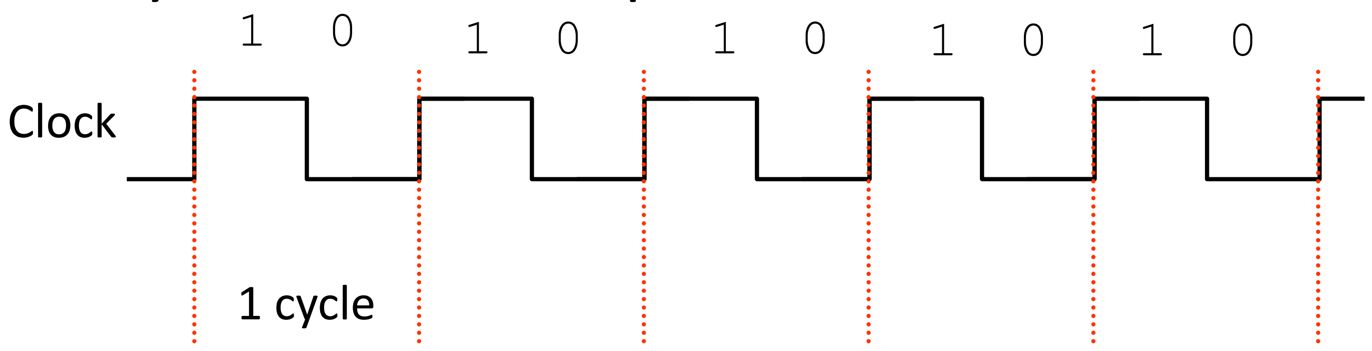
Clocking

- 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

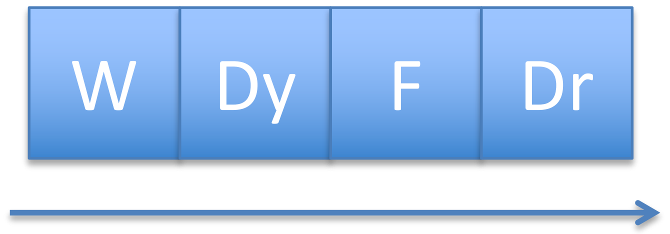


  - Clock determines how fast system runs
    - Processor can only do one thing per clock cycle
      - Usually just one part of executing an instruction
    - 1GHz processor:

      1 billion cycles/second → 1 cycle every nanosecond

Cycle Time: Laundry Analogy

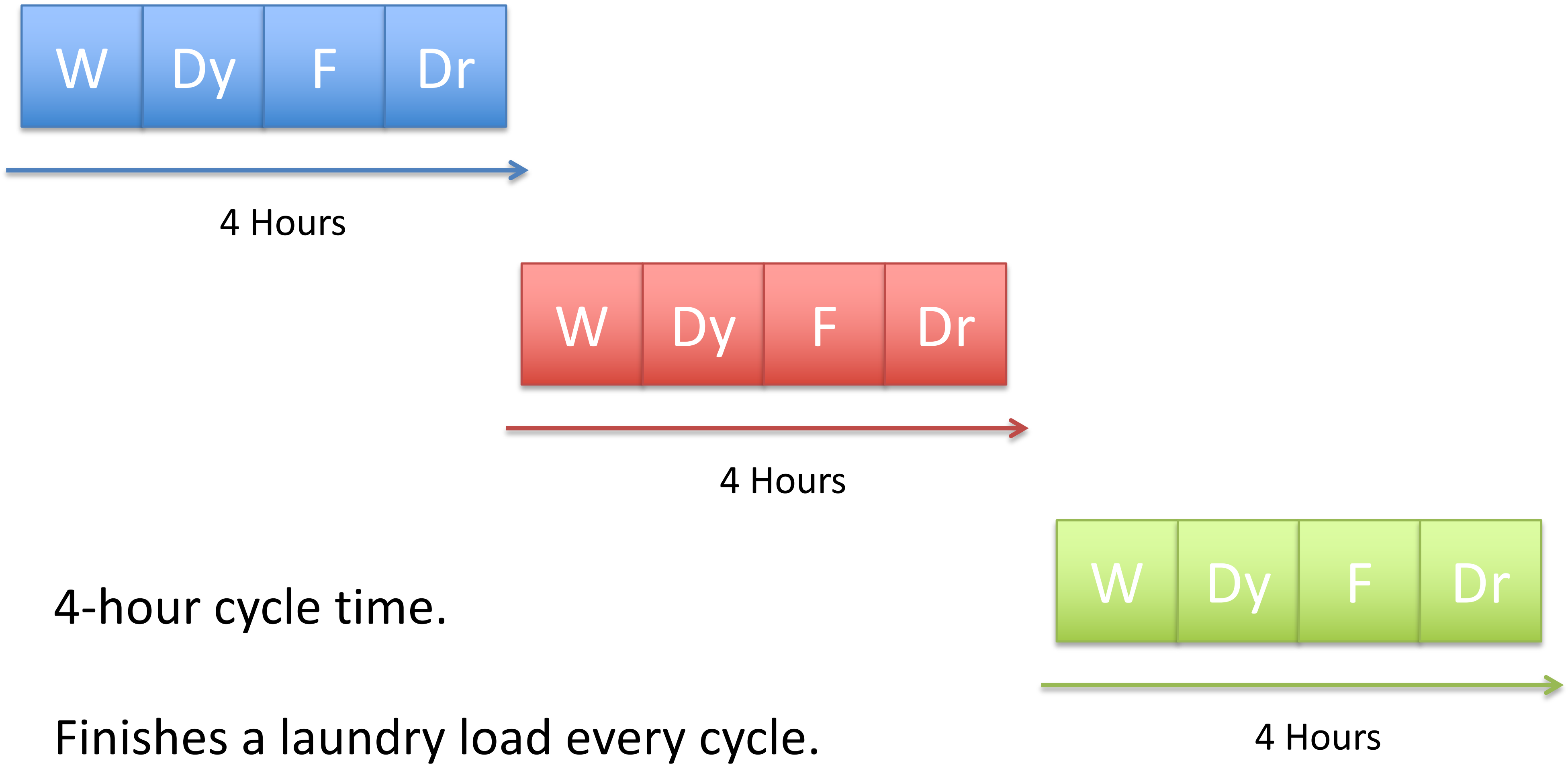
- Discrete stages: fetch, decode, execute, store
- Analogy (laundry): washer, dryer, folding, dresser



4 Hours (each stage takes 1 hour)

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

Laundry

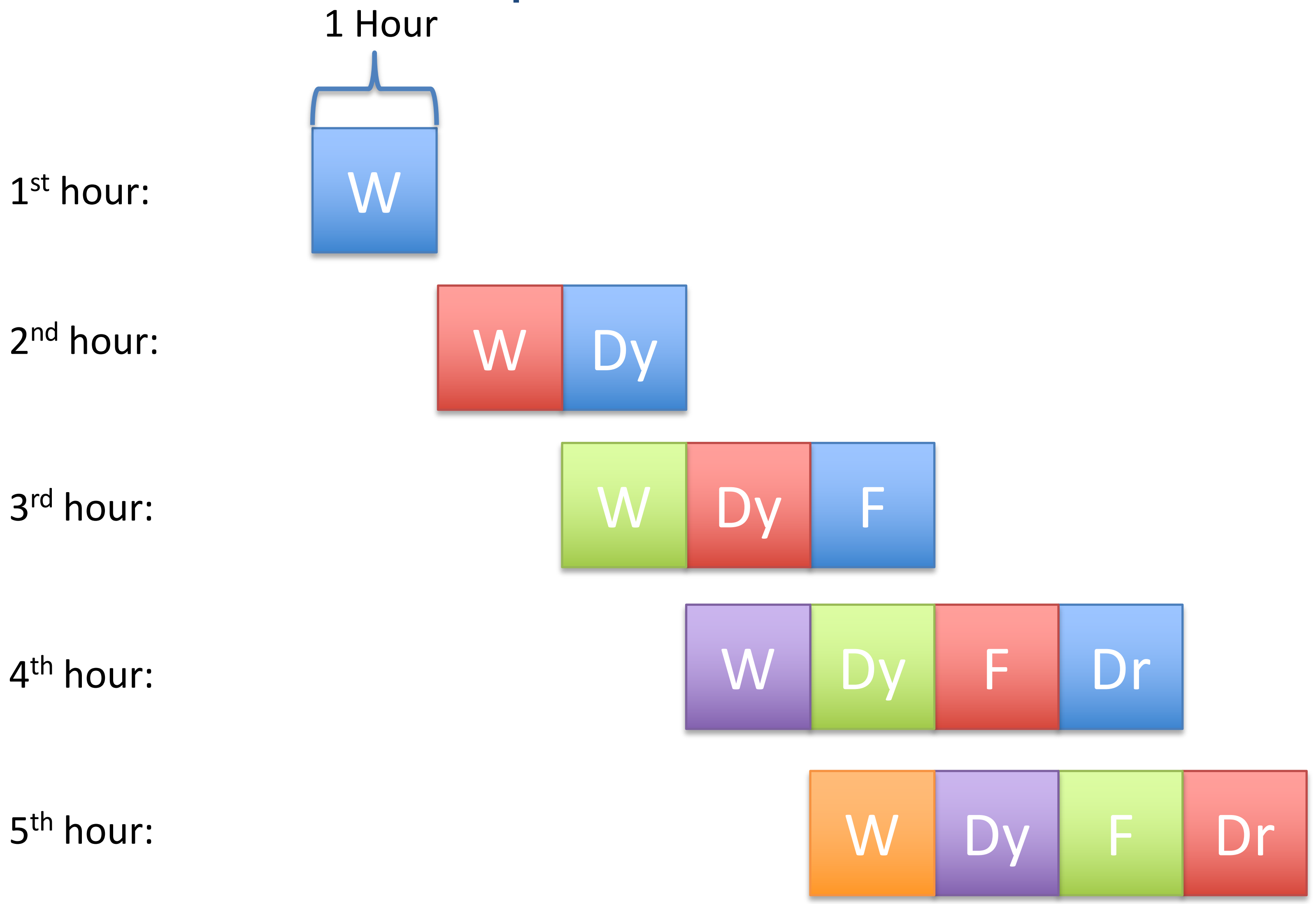


4-hour cycle time.

Finishes a laundry load every cycle.

(6 laundry loads per day)

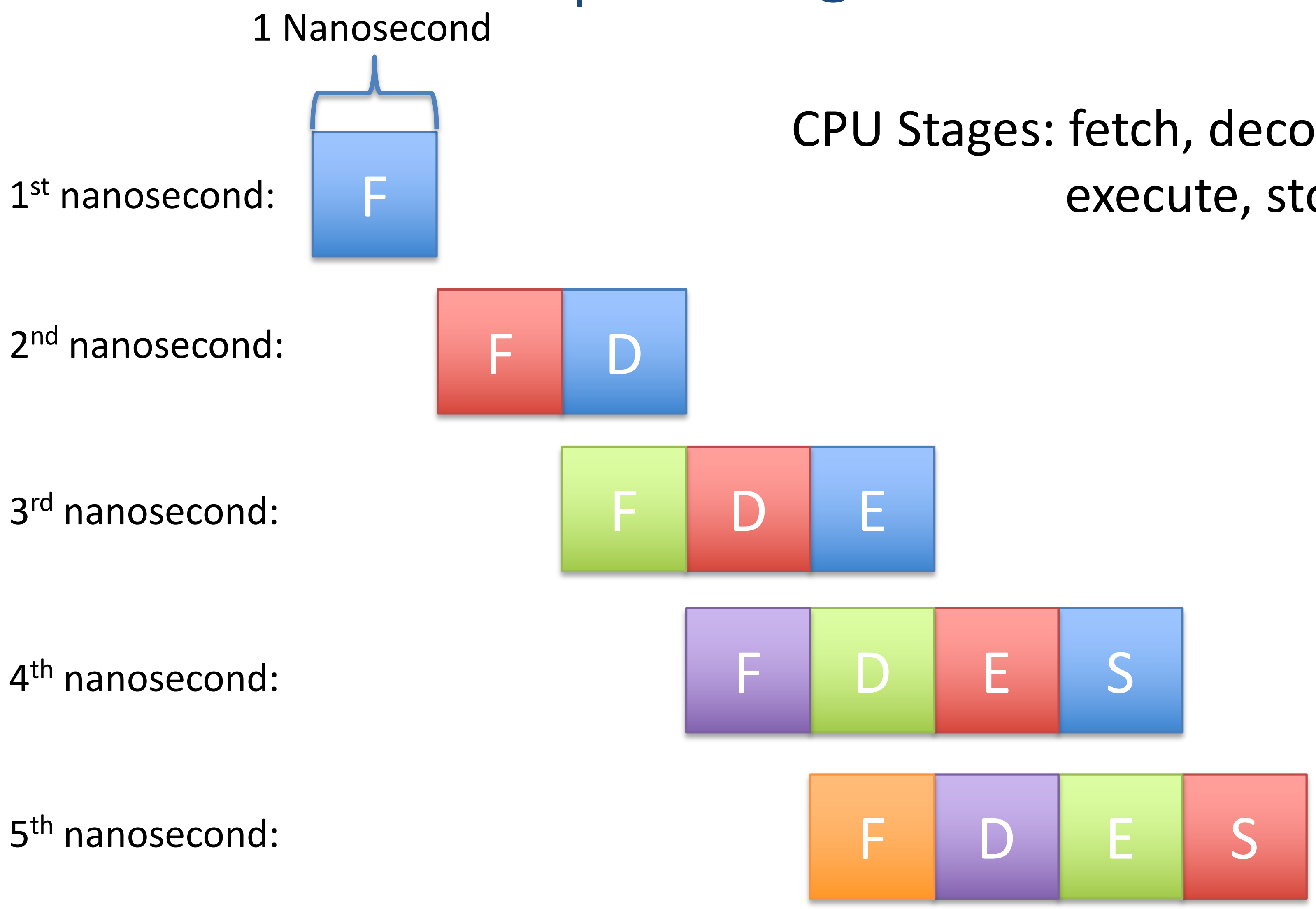
Pipelining (Laundry)



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