Agenda

How a computer runs a program

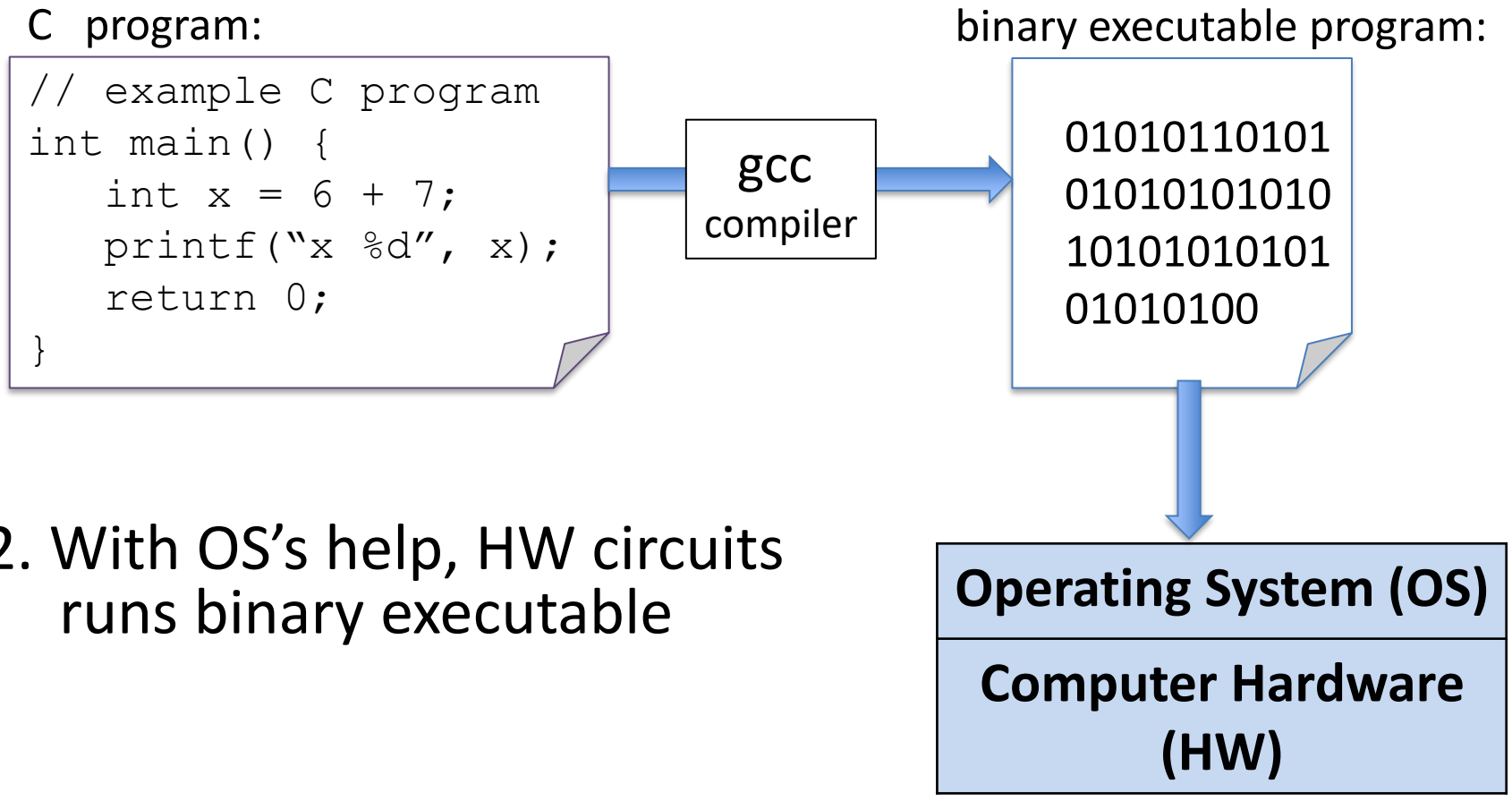
Von Neuman architecture overview

Building a CPU

- Logic gates
- Circuits
- ALU (Addition, Subtraction, Multiplexor)
- Storage (R-S Latches, registers)
- Control

Recall: Building and Running a C program

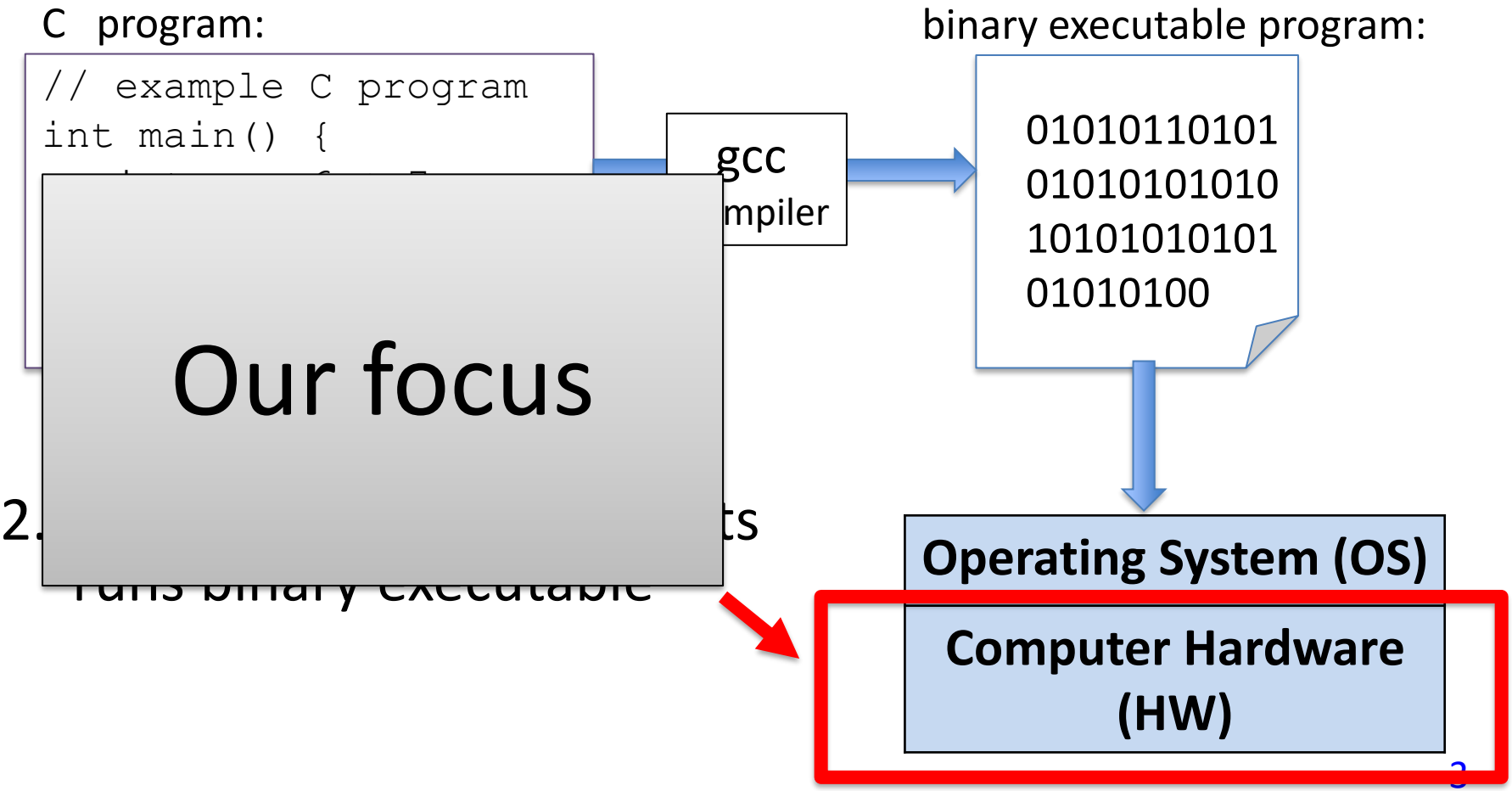
1. **Compiling** a C program translates it to binary (0's and 1's)
   - The binary file is an **executable**, meaning "we can run it"



2. With OS's help, HW circuits runs binary executable

Recall: Building and Running a C program

1. **Compiling** a C program translates it to binary (0's and 1's)
   - The binary file is an **executable**, meaning "we can run it"



Slides by Tia Newhall, CS31, Swarthmore

Von Neumann Architecture (1945)

Computer is a generic computing machine

- Can be used to compute anything that is computable
- Based on Alan Turing's Universal Turing Machine

Uses a **stored program model**

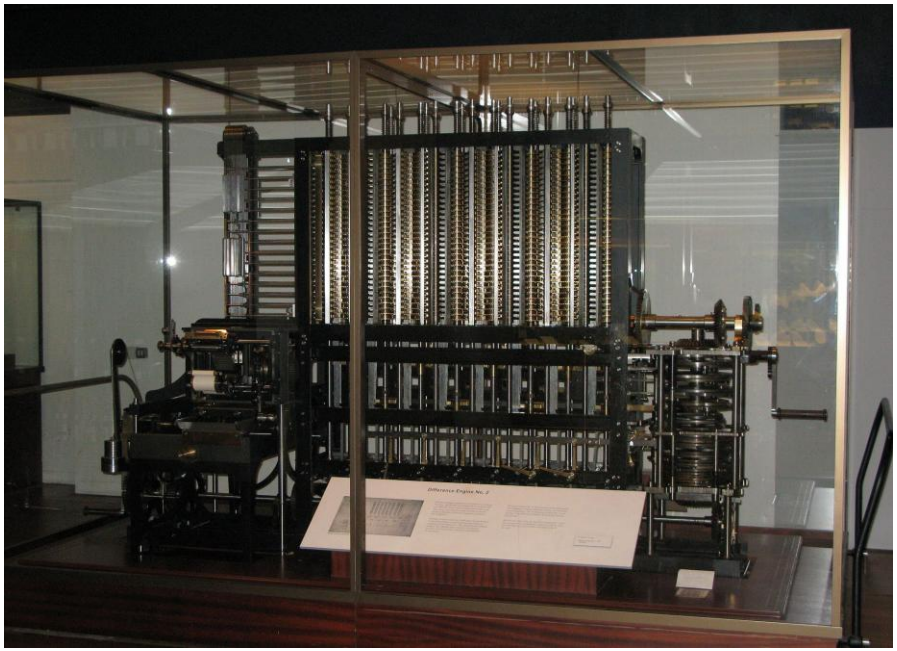
- Both program & data loaded into computer memory
- No distinction between data & instructions in memory

All modern computers based on the Von Neumann model

Early computers

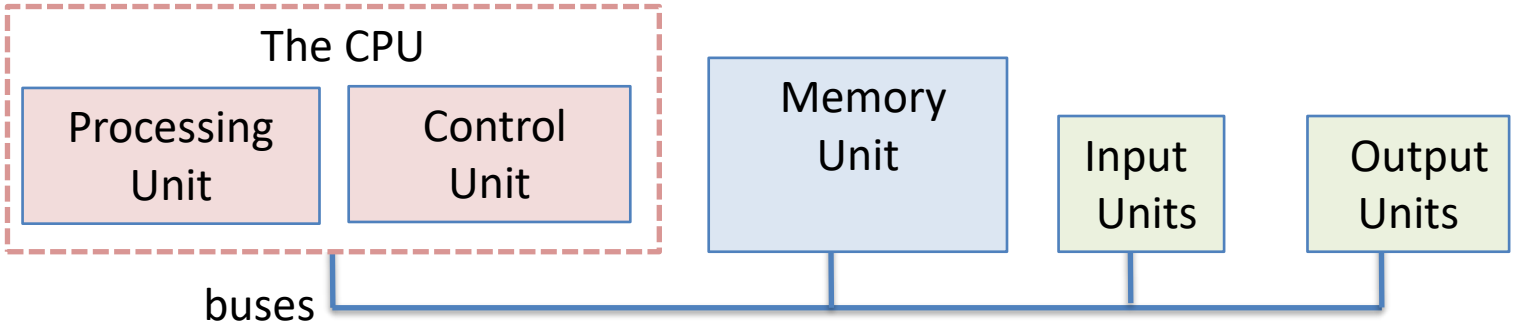
Earlier computers used fixed program encoded on machine, data loaded and run by fixed program

Left, The difference engine (Babbage) could estimate polynomial functions useful for making tables



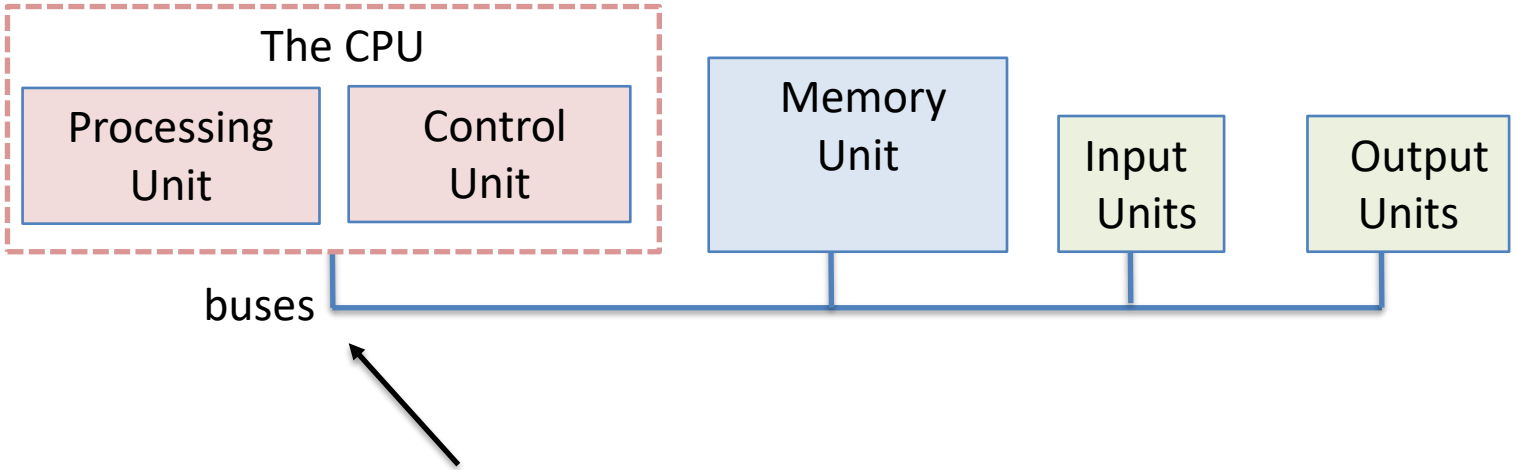
Von Neumann Model

units connected by buses (wires) to communicate



Von Neumann Model

5 units connected by buses (wires) to communicate

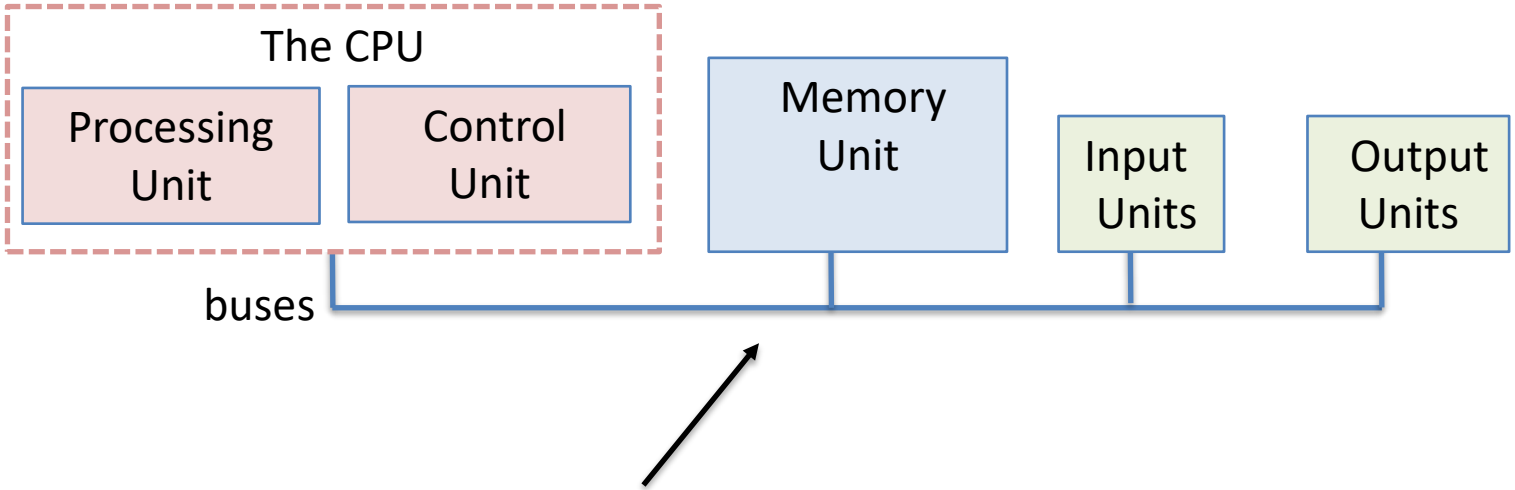


Processing & Control Units: implement CPU

execute program instructions on program data

Von Neumann Model

5 units connected by buses (wires) to communicate

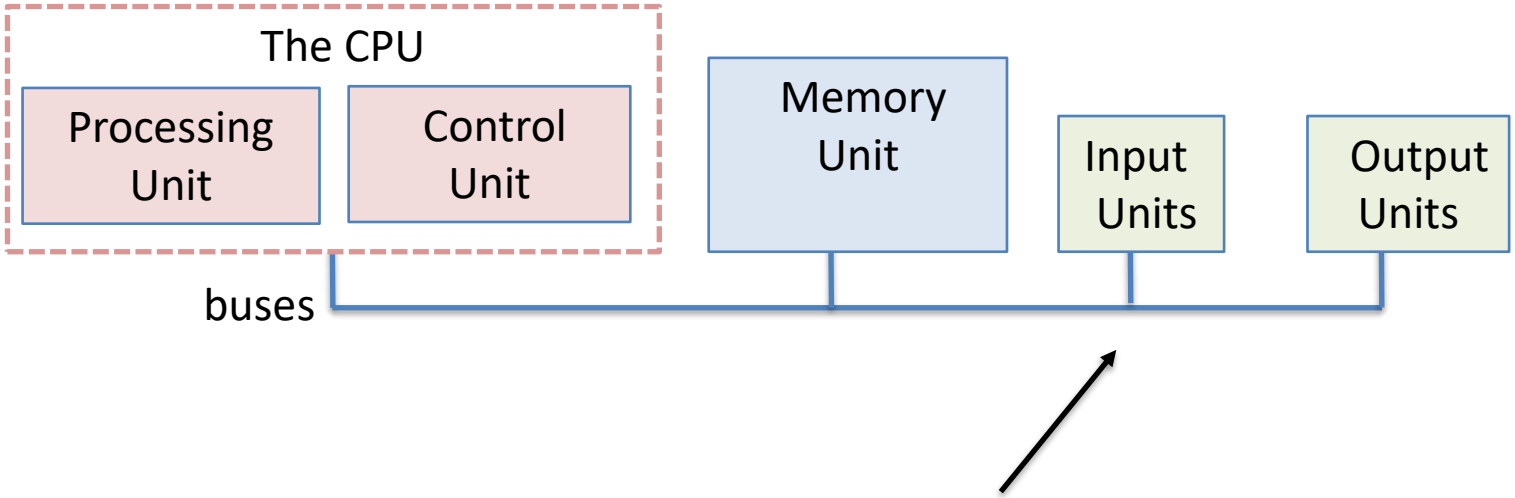


Memory: stores program instructions and data

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

Von Neumann Model

5 units connected by buses (wires) to communicate

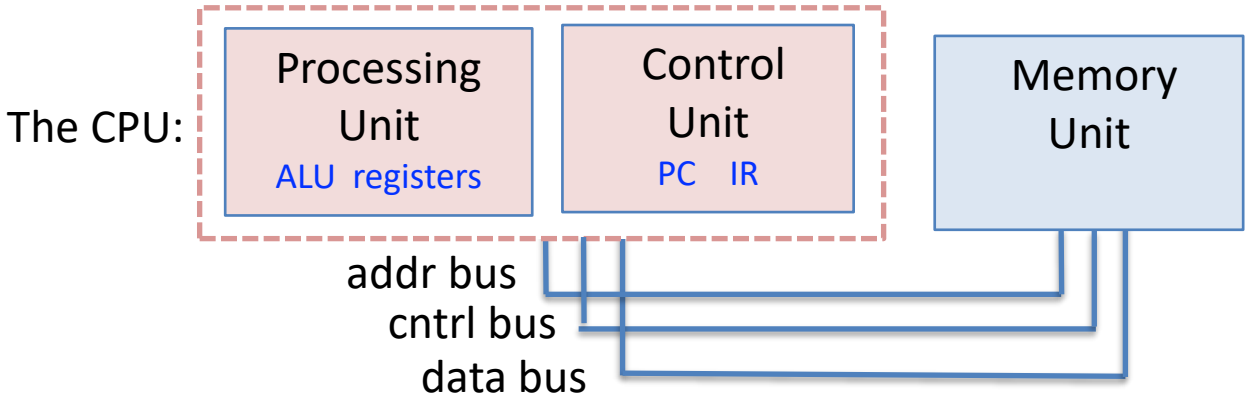


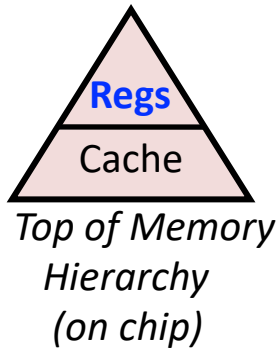
Input, Output: interface to computer

trigger actions: load program, initiate execution, ...

display/store results: to terminal, save to disk, ...

Von Neumann Model: CPU/Memory

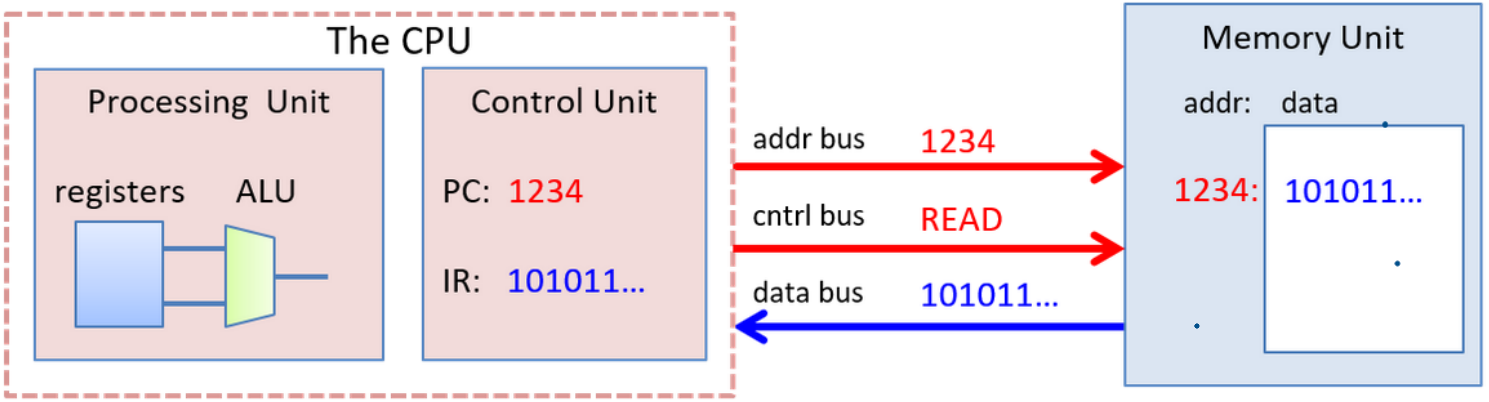




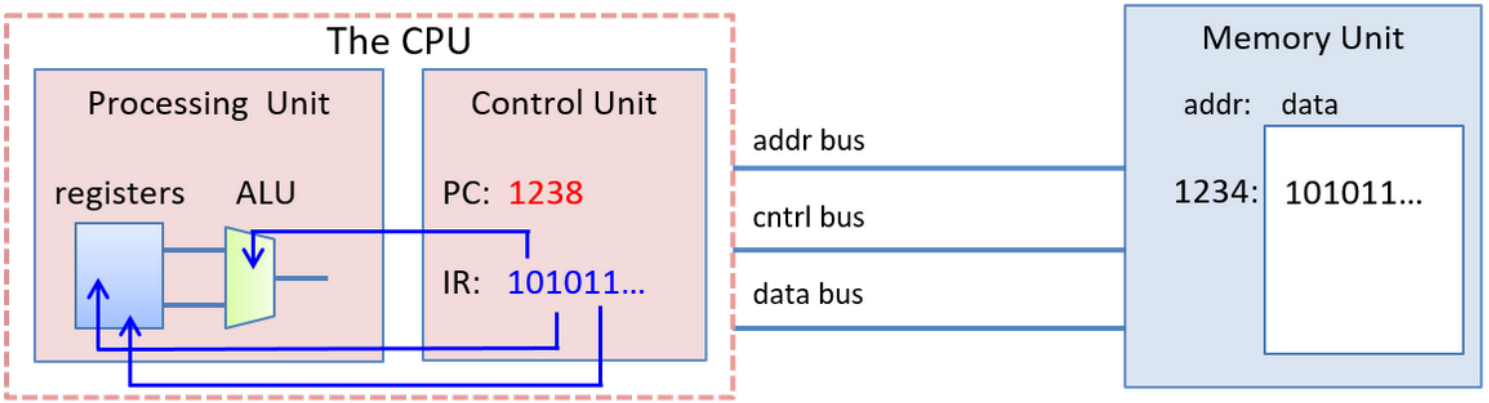
Instruction Execution: controlled by Control Unit

1. Fetch instruction from Memory (its addr in PC) into IR (and increment address in PC to next instruction address)
2. Decode instruction bits to determine operation & operands
3. Execute instruction on ALU
4. Store instruction results to Memory

Example: Adding two numbers

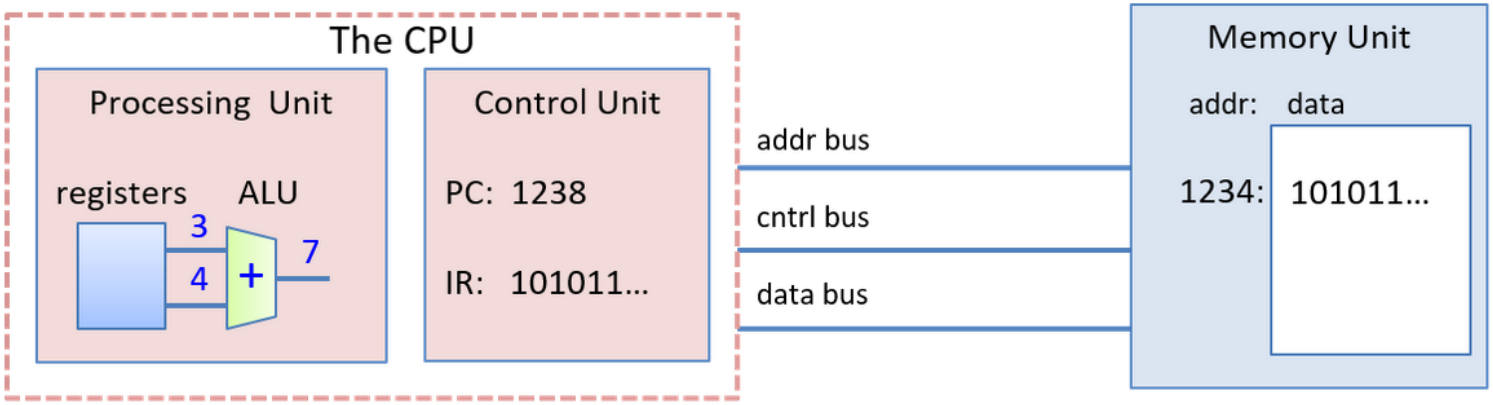


1. Fetch: Read instruction bits from memory at address in PC (1234), and store in IR

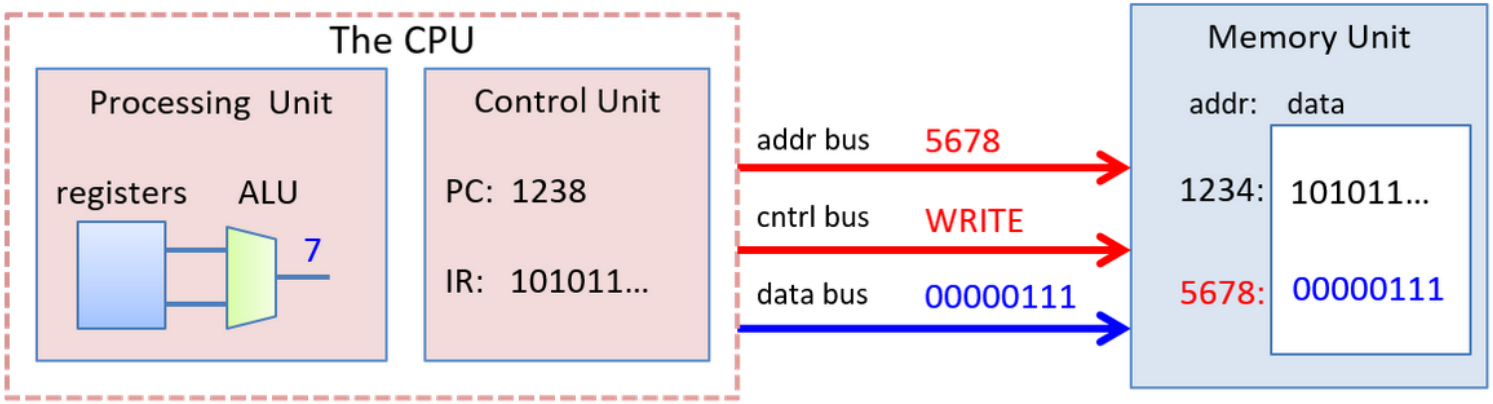


2. Decode: instruction bits in IR encode which registers store operands & the ALU operation

Example: Adding two numbers



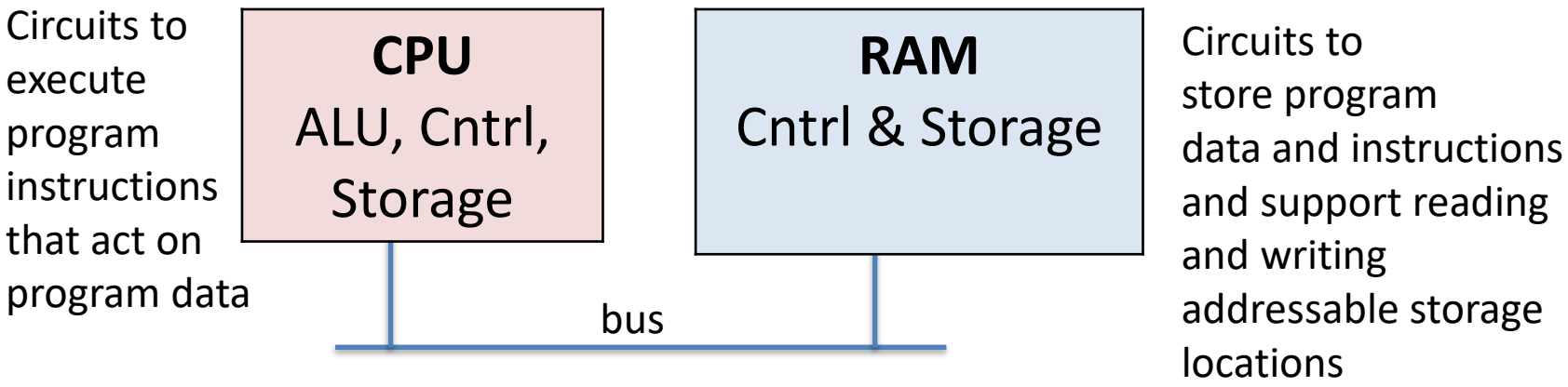
3. Execute: ALU performs instruction operation (+) on operands (3,4) to compute result (7)



4. Store: the control unit stores the ALU result (7, binary 00000111) to memory

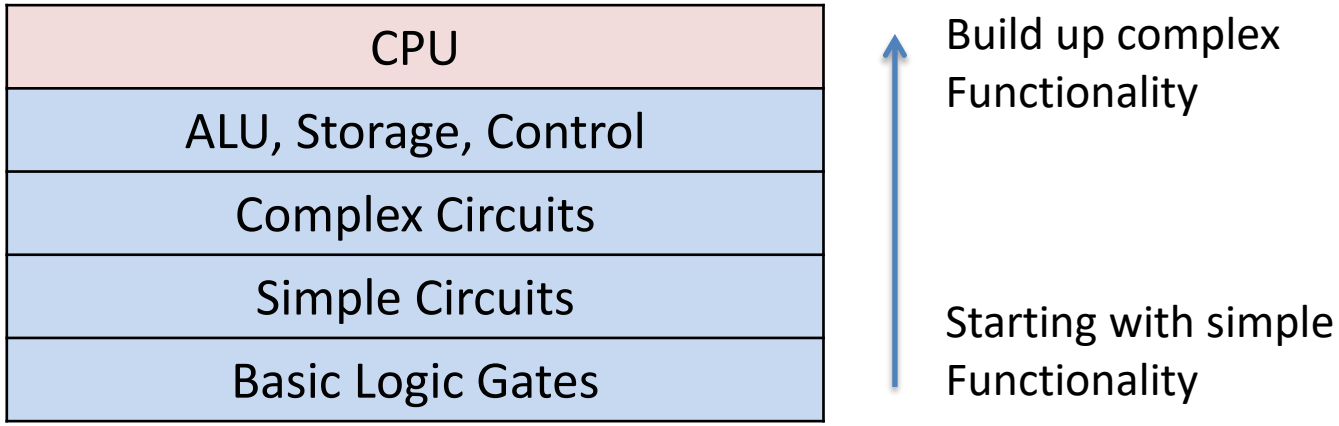
Digital Computers

- All input & output are discrete and binary
  - data, instructions, control signals (0: no voltage, 1: voltage)
  - execution is driven by a clock (will discuss later)
  - time is discrete: time 1, time 2, time 3, ...
- To run program, need different types of circuits



Building a CPU (model)

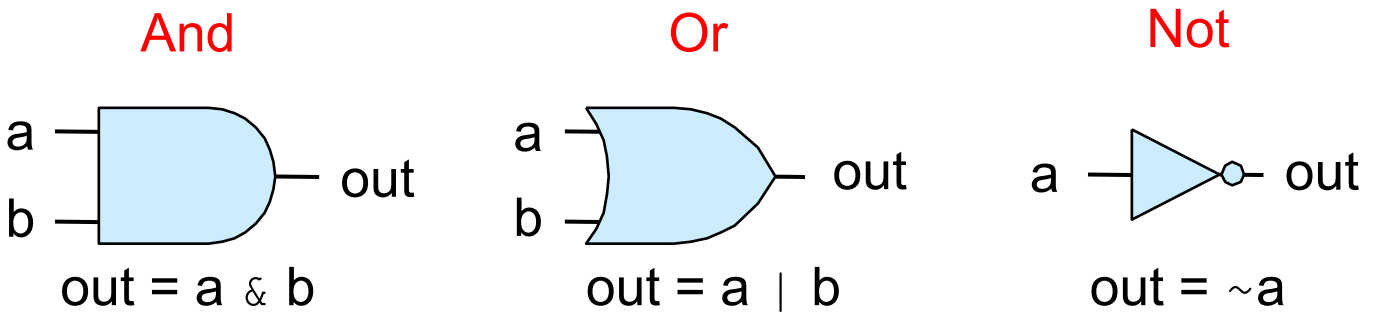
Levels of abstraction: start with very simple functionality, and add complexity



Logic Gates: Basic Building Blocks

Input: Boolean value(s) (high and low voltages for 1 and 0)
Output: Boolean value (0 or 1) result of boolean function
Always present, but may change when inputs change

C bit-wise operators:
`&`: AND
`|`: OR
`~`: NOT



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

Review: C Bitwise Operators

Bit-wise operators: applied to bits of operand(s)

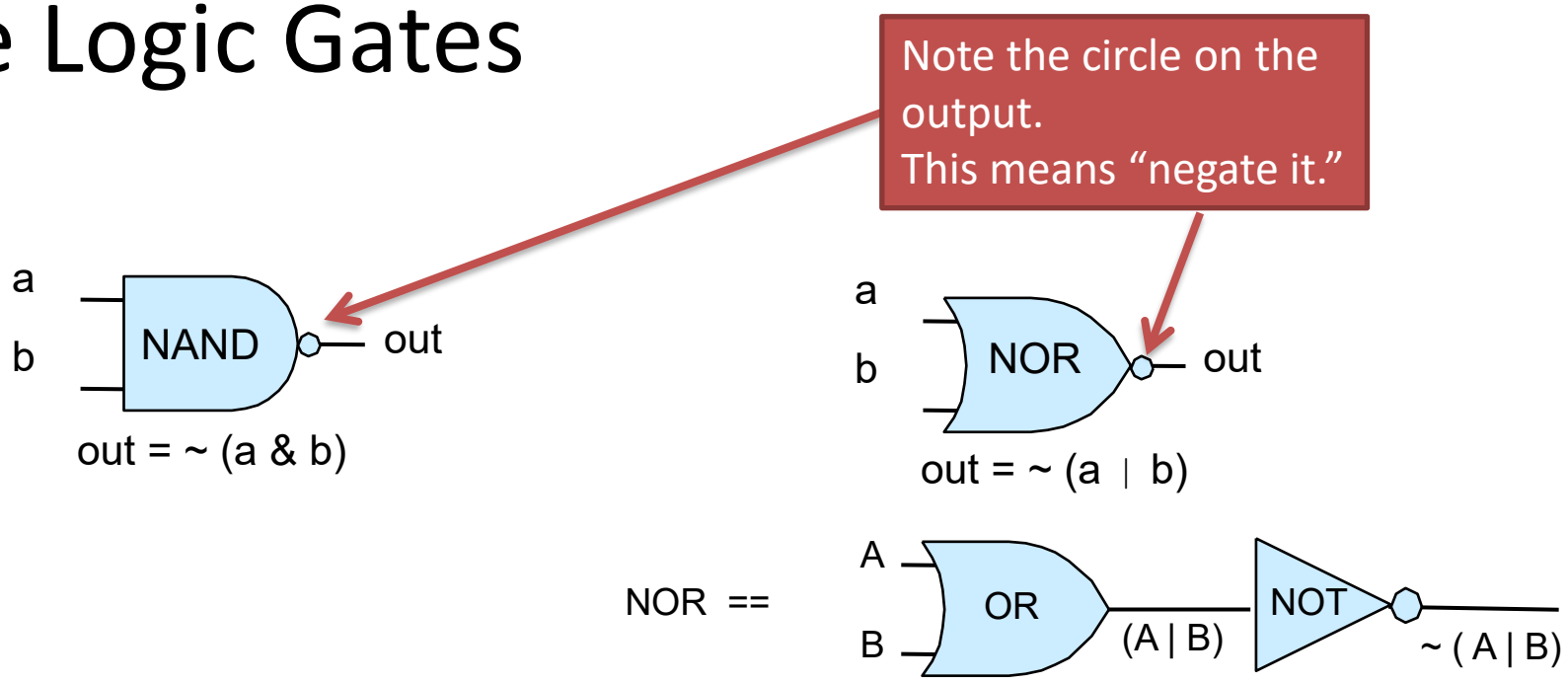
& (AND) | (OR) ^(XOR) ~(NOT)

```
  01010101      01101010      10101010     ~10101111
& 00100001    | 10111011    ^ 01101001      01010000
  00000001      11111011      11000011
```

Evaluate Differently than C logical operators

```
char x=8, y=7;
x & y    // 8 bitwise AND 7 is 0
x && y   // 8 logical AND 7  is a non-zero value (evals to true)
```

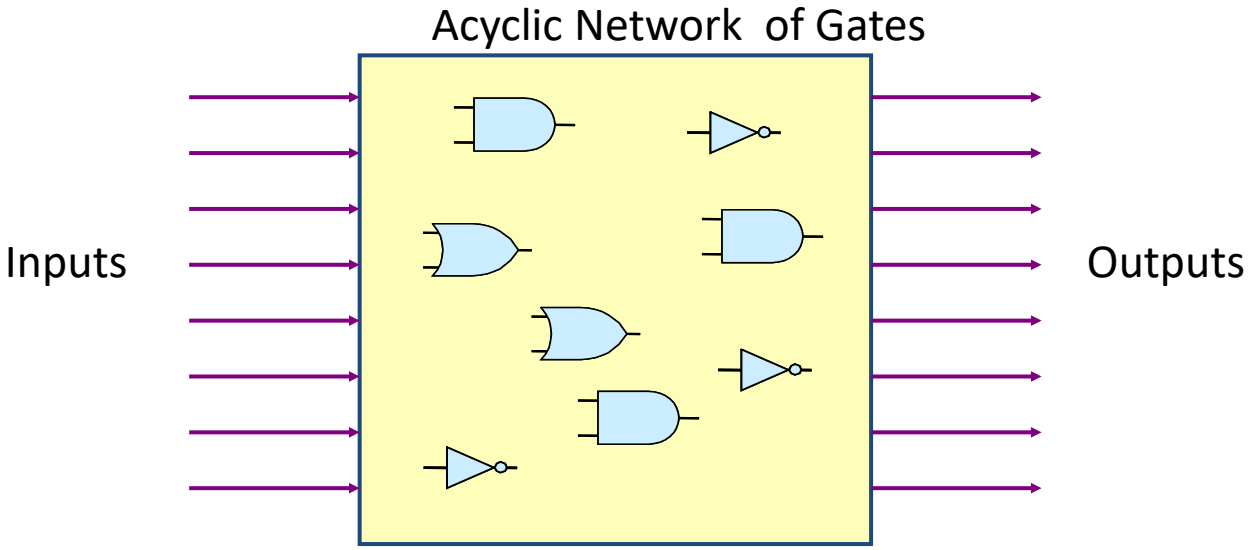
More Logic Gates



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

Combinatorial Logic Circuits

Idea: Combine logic circuits together to implement higher-level functionality

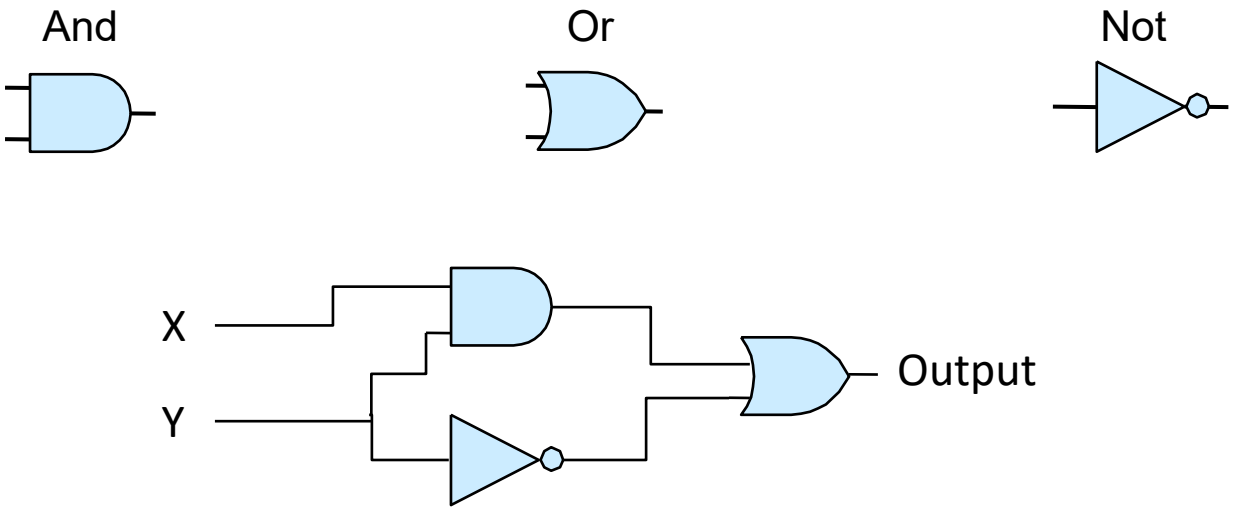


Outputs are boolean functions of inputs

Outputs continuously respond to changes to inputs

Use this new functionality as a building block for even higher level functionality (Abstraction!)

Exercise: Draw the truth table for this circuit



X	Y	Output

Exercise: Build a circuit that implements XOR using AND, OR, and NOT

Step 1: Construct the logic table

Exercise: Build a circuit that implements XOR using AND, OR, and NOT

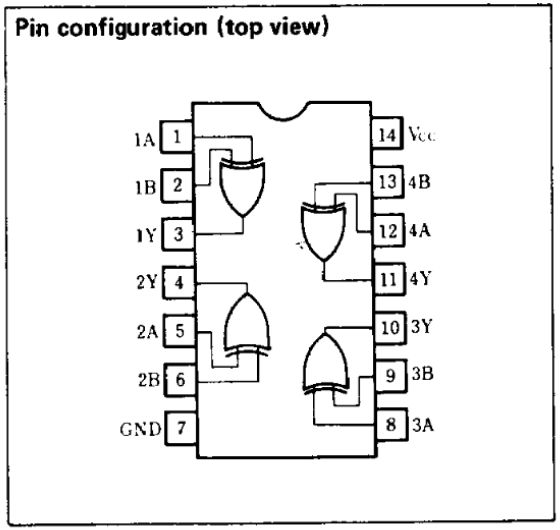
Step 2: Find an expression for X^Y using binary operations

Exercise: Build a circuit that implements XOR using AND, OR, and NOT

Step 3: Translate the expression to a circuit

Abstractions in action

Treat XOR Circuit as a building block for other circuits!



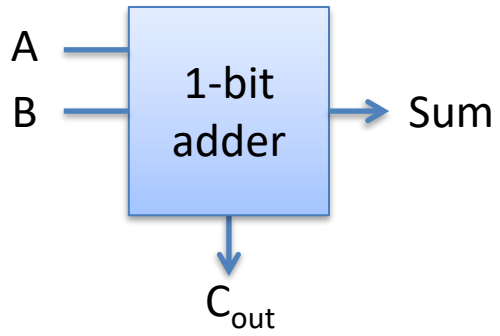
Arithmetic logic unit (ALU)

The **ALU** performs addition, subtraction, or, add, etc.

- Adder
- Subtractor
- Multiplication / Division (we are not covering these)
- Bit-wise AND, OR, NOT, XOR
- Bit shift operations
- Goal: reuse hardware components whenever possible

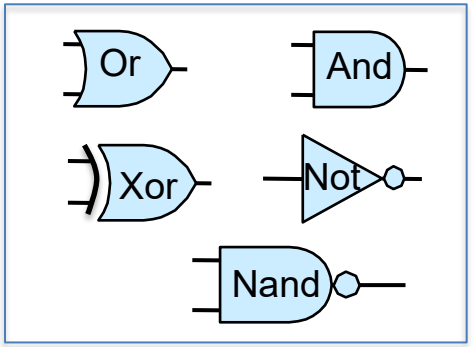
ALU: a 1-bit adder circuit

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

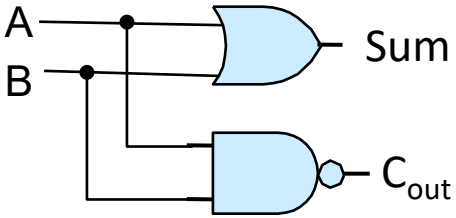


A	B	Sum(A+B)	Cout
0	0		
0	1		
1	0		
1	1		

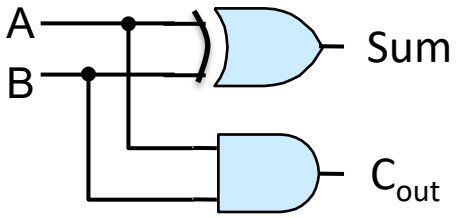
Which of these circuits is a one-bit adder?



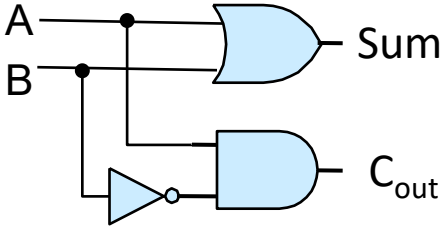
A:



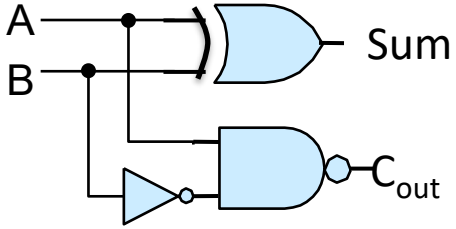
B:



C:



D:



More than one bit addition?

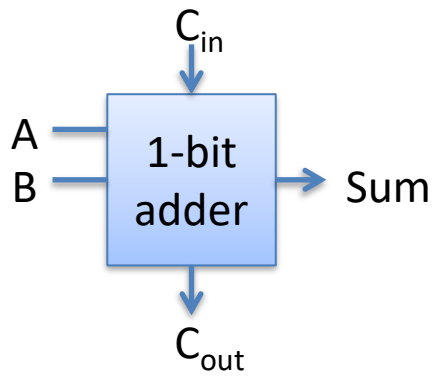
- When adding, sometimes have *carry in* too

```
  00011010
+ 00001111
```

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?

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

- When is C_{out} 1?

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

One-bit (full) adder

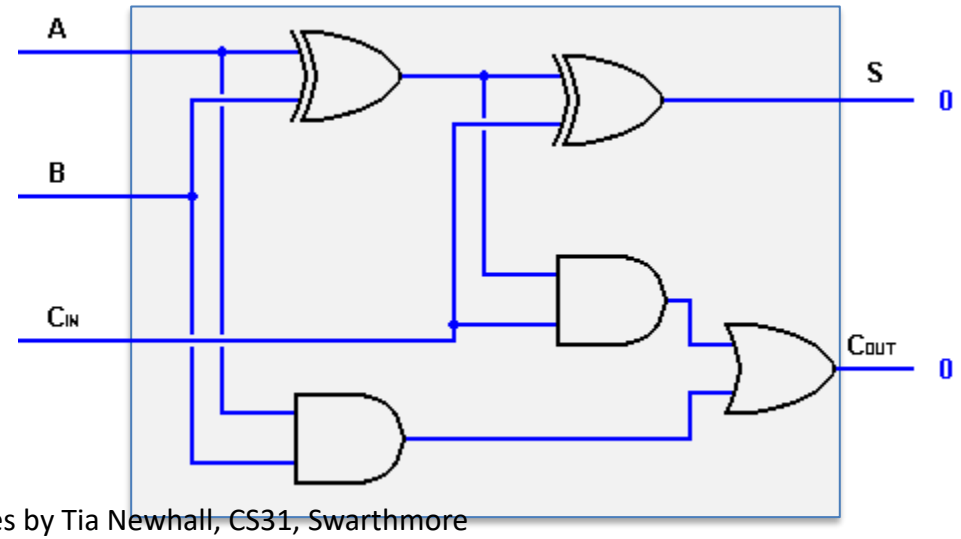
Need to include:

Carry-in & Carry-out

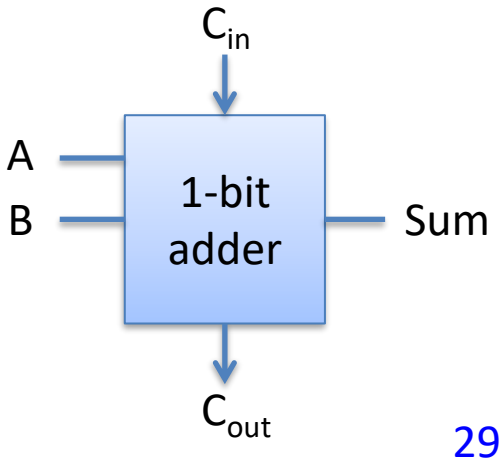
Sum: C_{in} ^ (A^B)

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

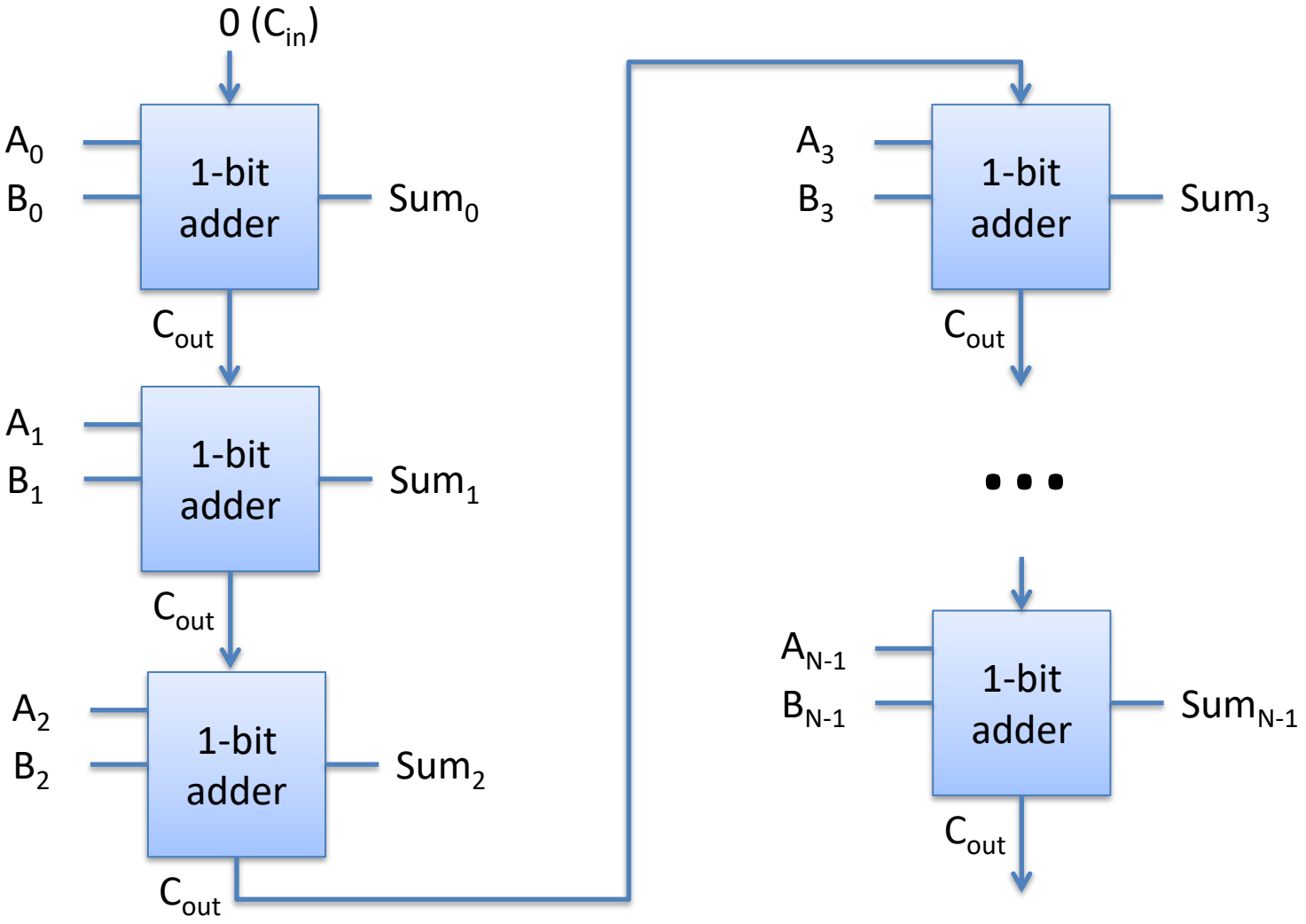
A	B	Cin	Sum	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



=

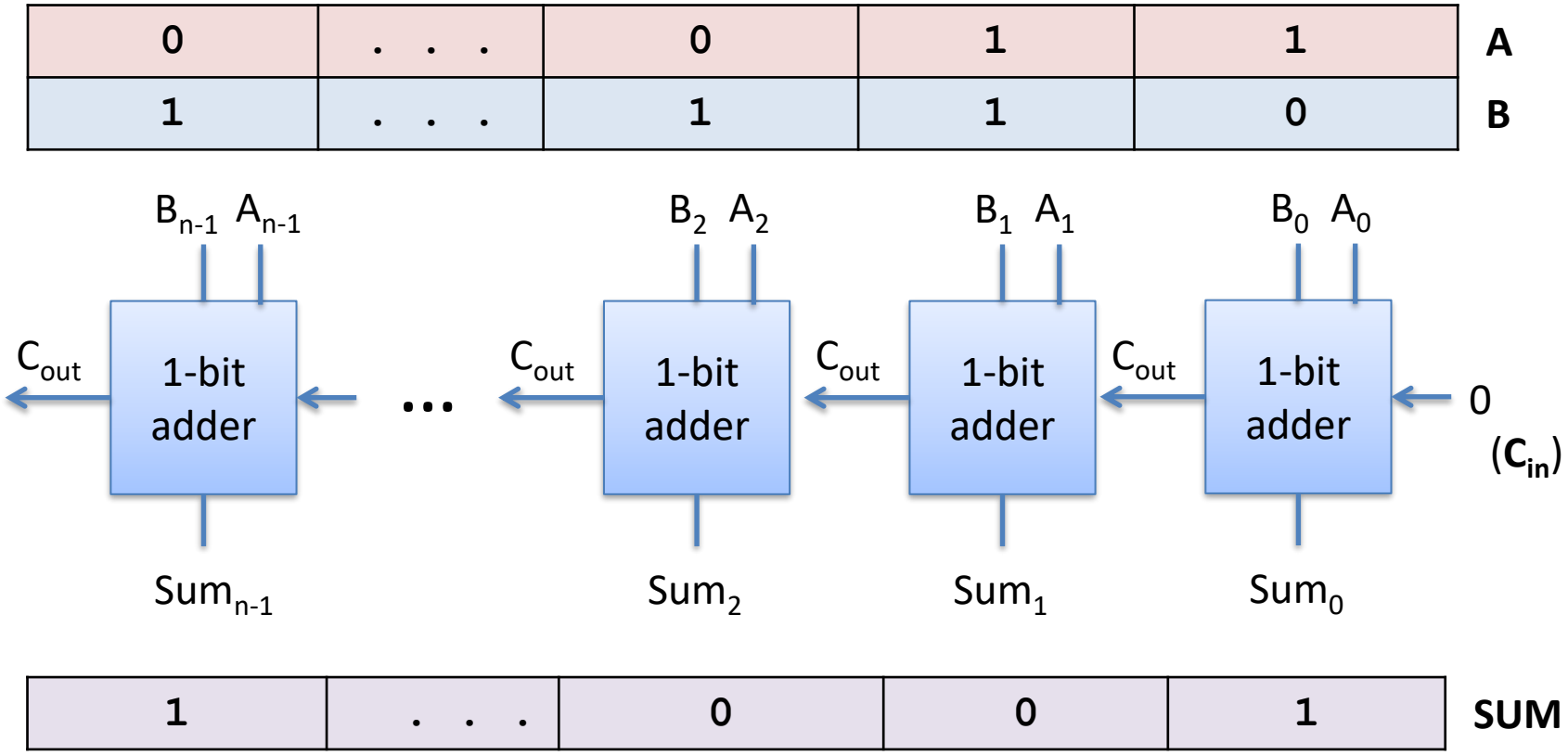


Multi-bit Adder (Ripple-carry Adder)

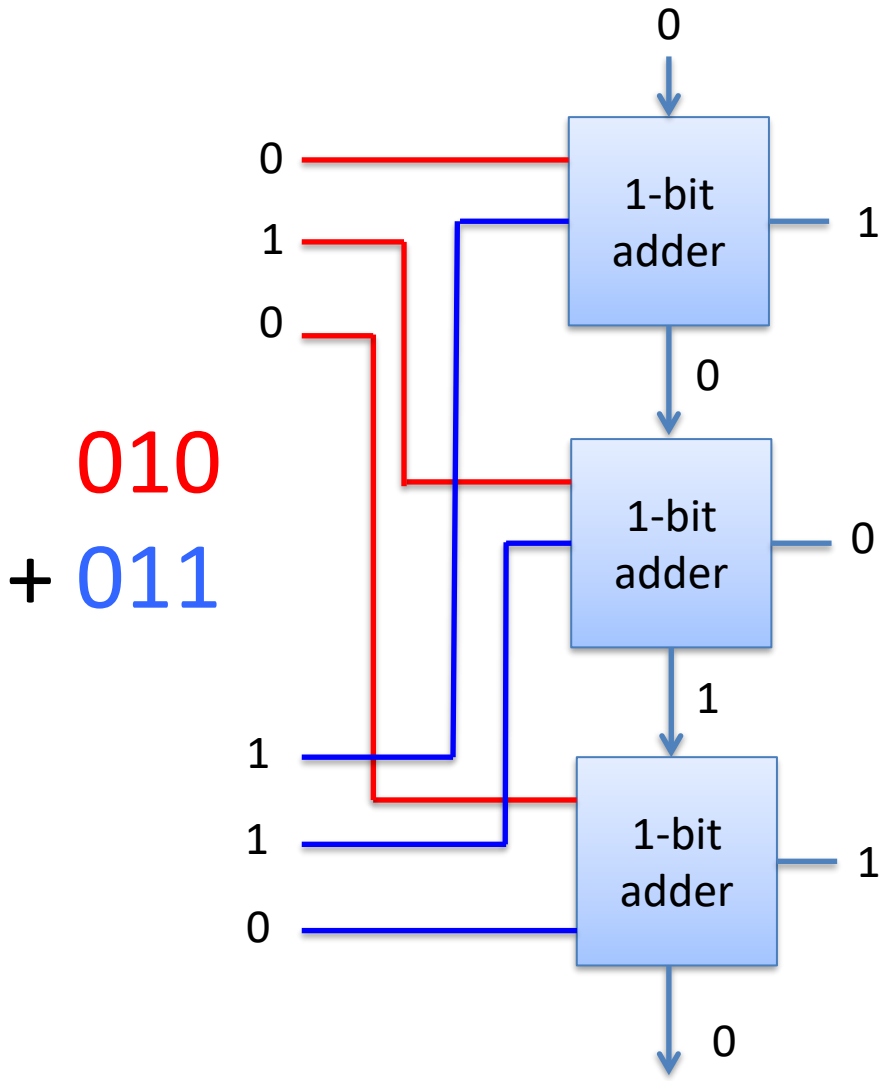


Slides by Tia Newhall, CS31, Swarthmore

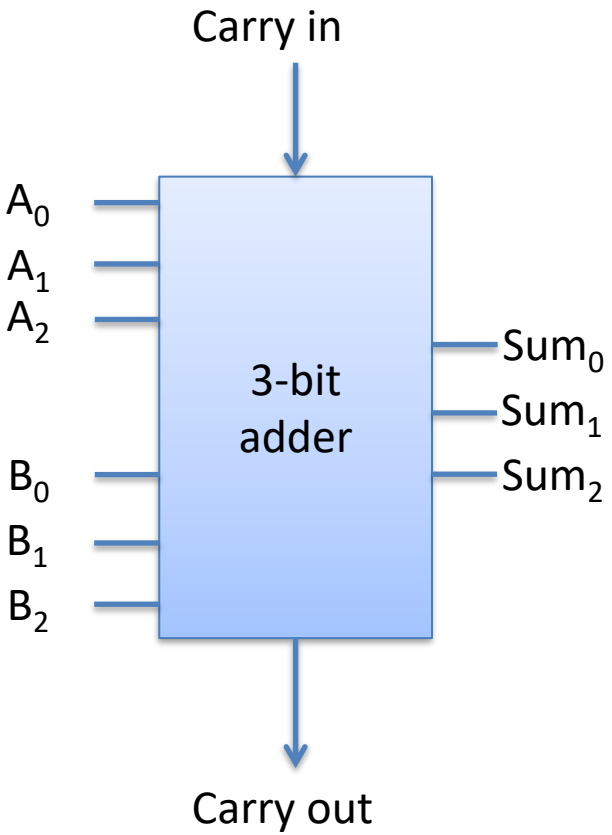
Multi-bit Adder (Ripple-carry Adder)



Three-bit Adder (Ripple-carry Adder)

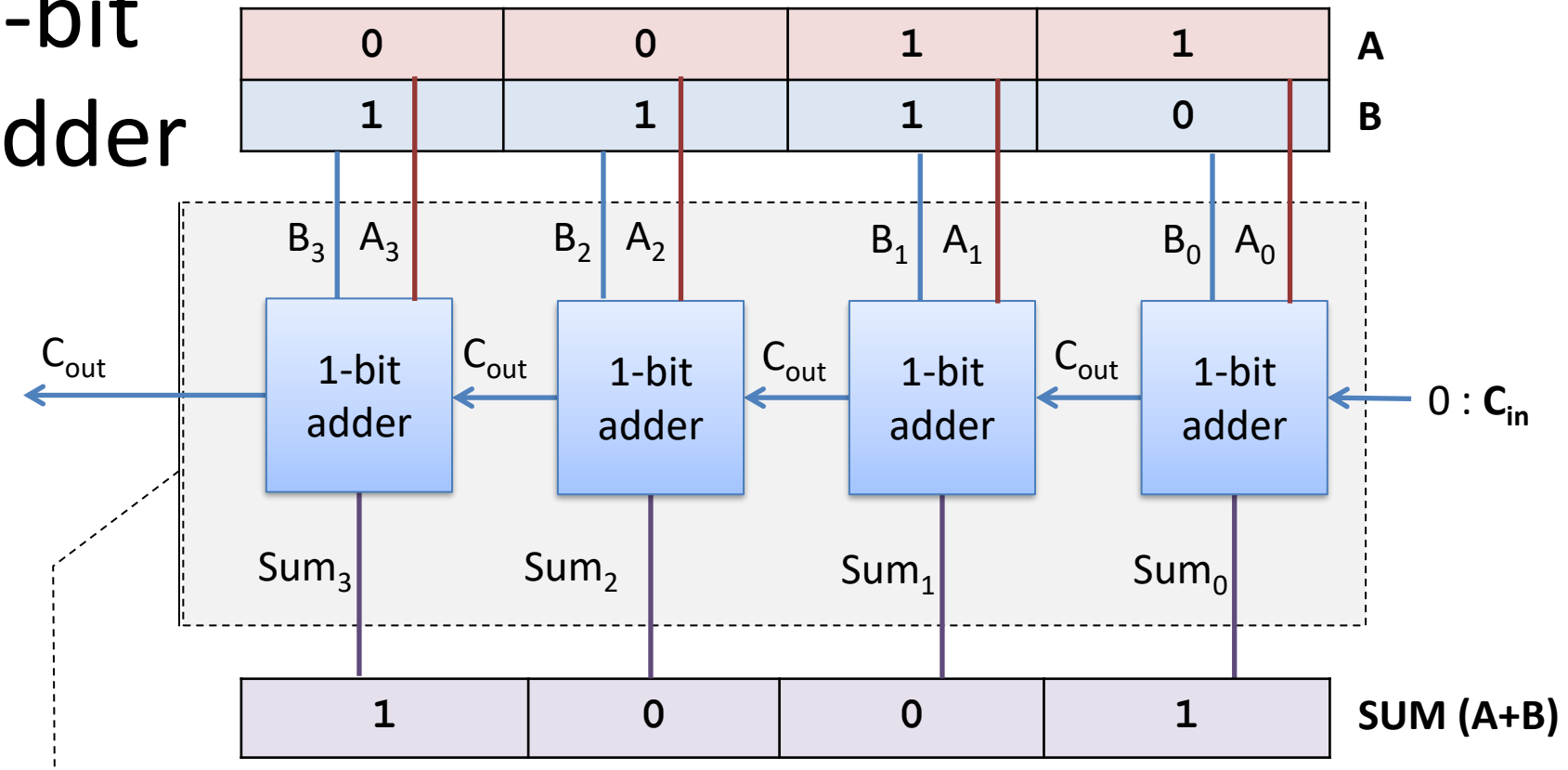


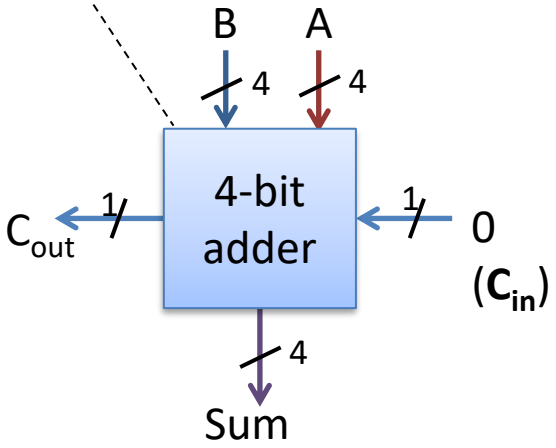
=



=101

4-bit adder

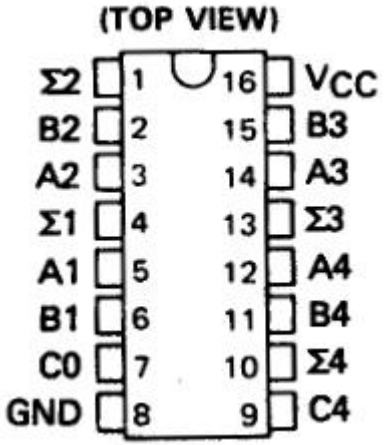




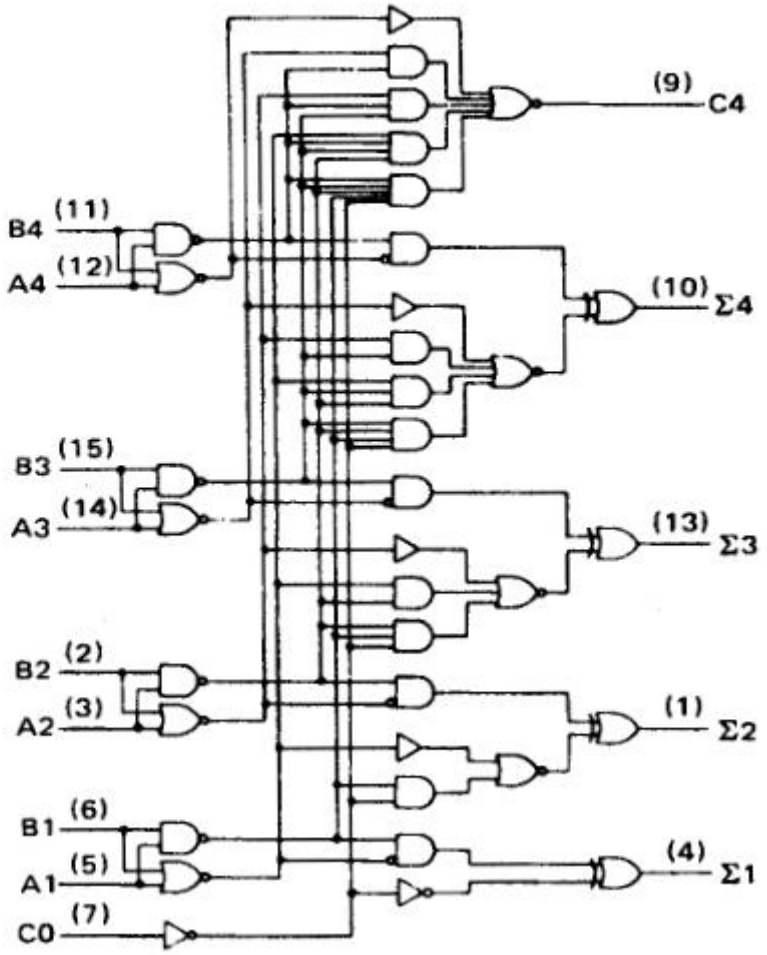
Abstraction!

Can treat Multi-bit adder as a unit: as a building block for larger, more complicated functionality.

4-bit adder real life example



logic diagram (positive logic)

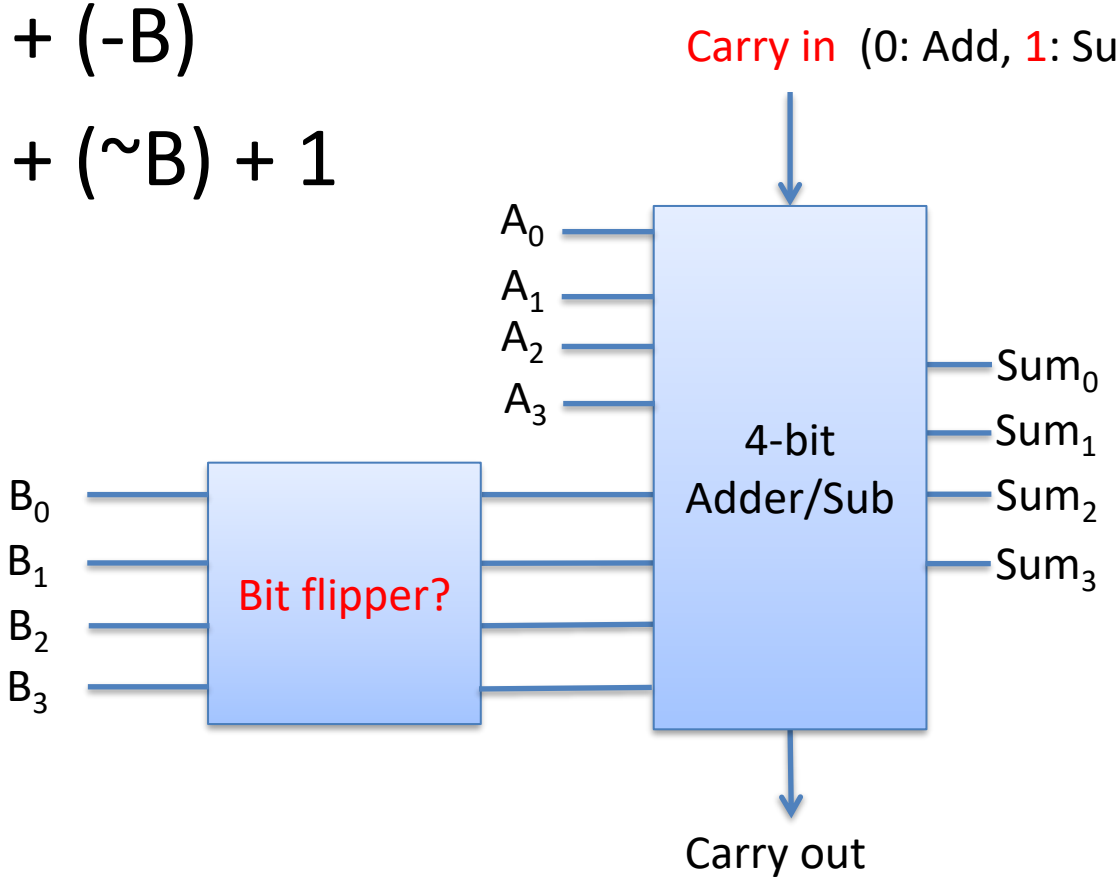


How to Implement Subtraction Circuit?

Q: what is relationship between Add & Sub?

A – B == A + (-B)

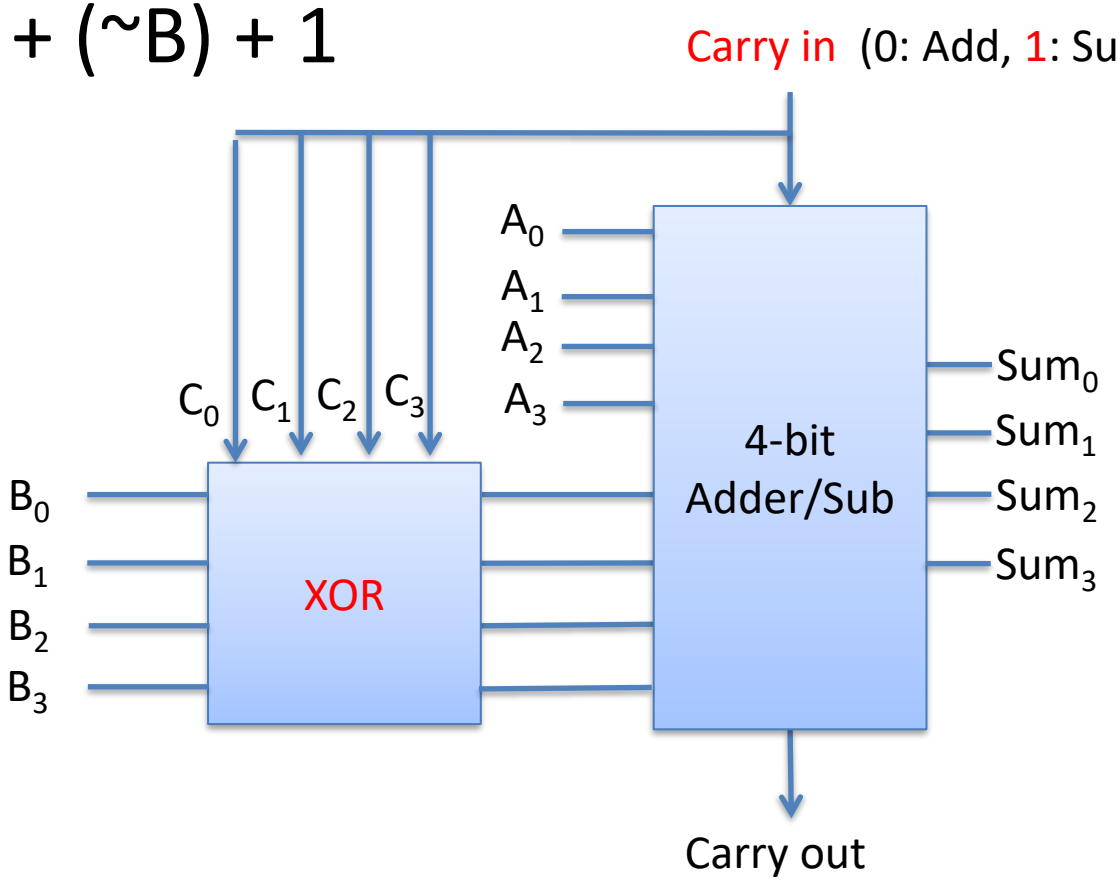
A – B == A + (~B) + 1



How to Implement Subtraction Circuit?

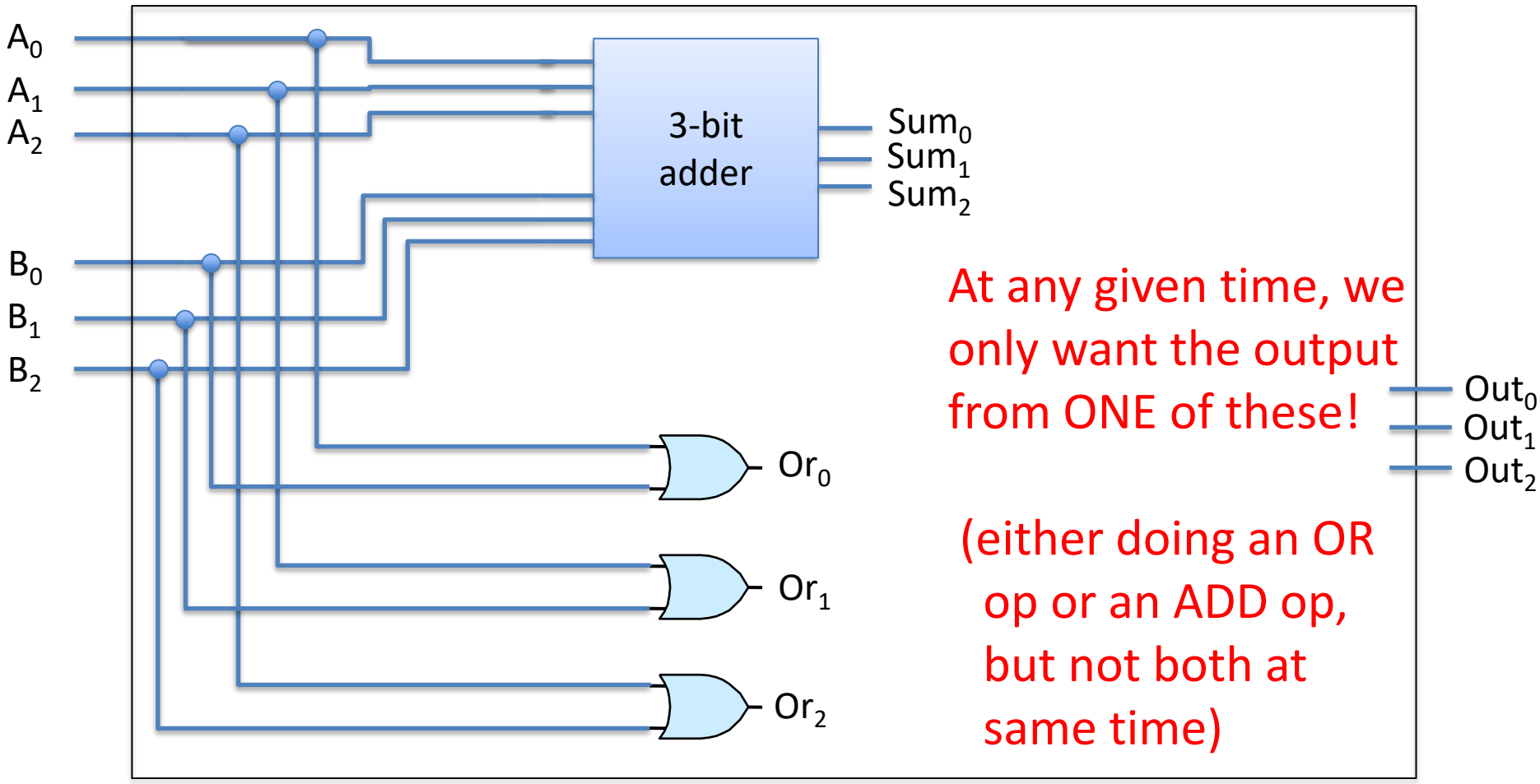
Q: what is relationship between Add & Sub?

A – B == A + (~B) + 1



Simple 3-bit ALU: Add and bitwise OR

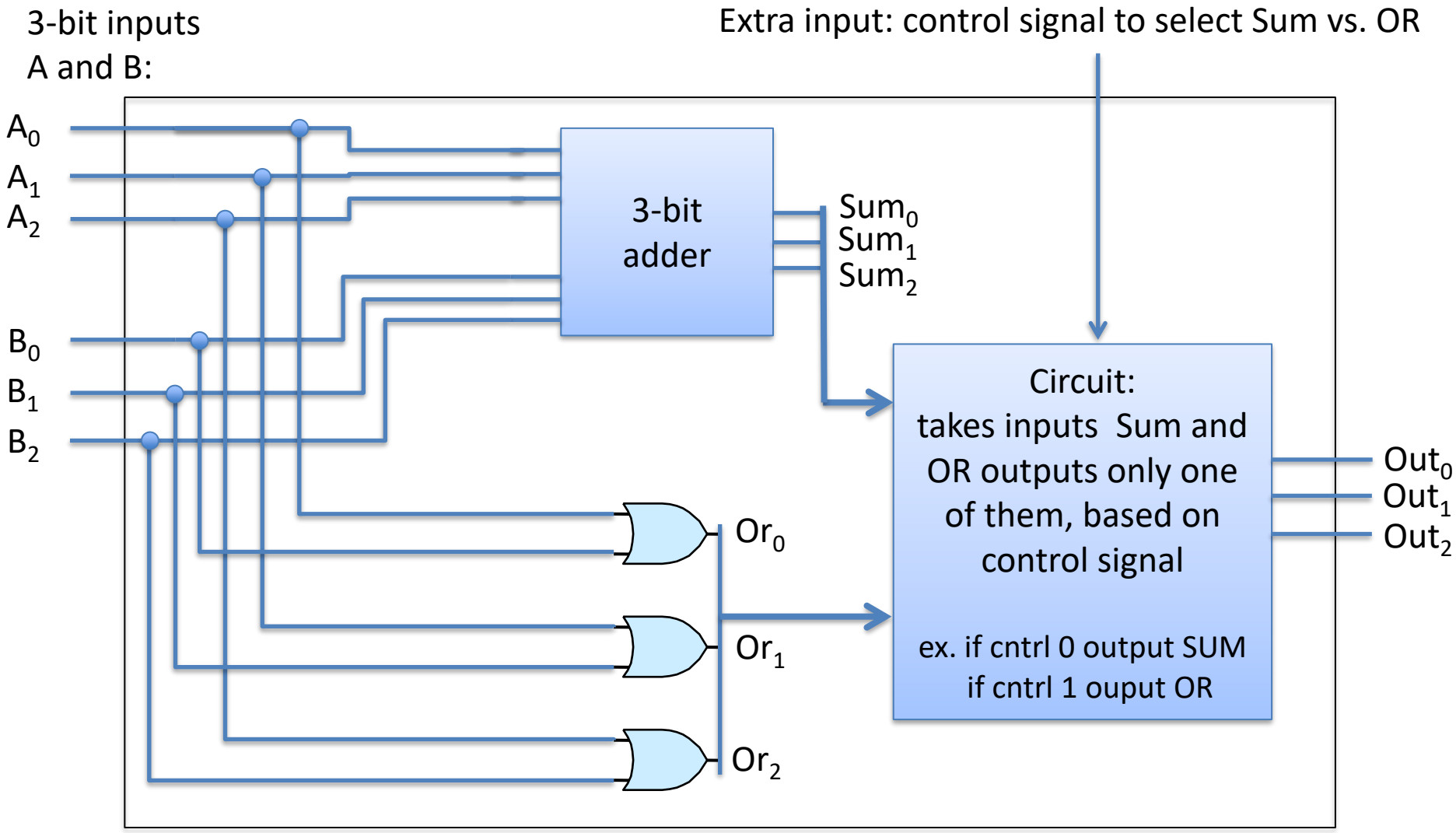
3-bit inputs
A and B:



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

(either doing an OR op or an ADD op, but not both at same time)

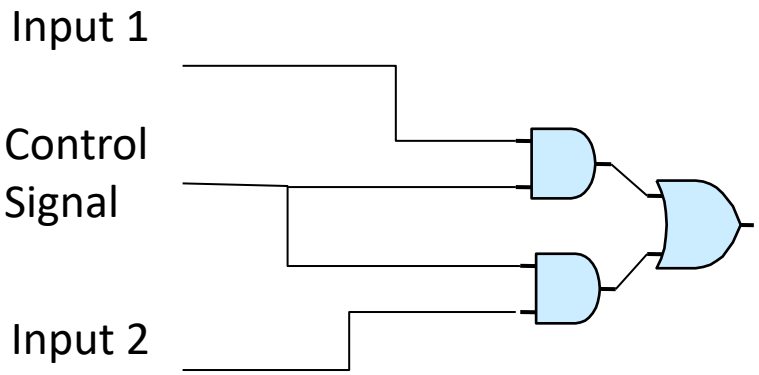
Simple 3-bit ALU: Add and bitwise OR

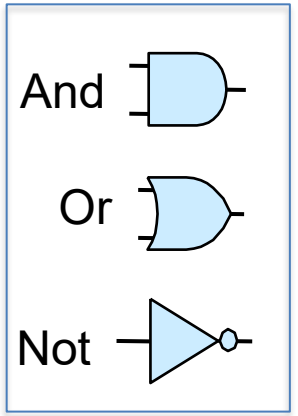


Slides by Tia Newhall, CS31, Swarthmore

Draw the truth tables for this circuit

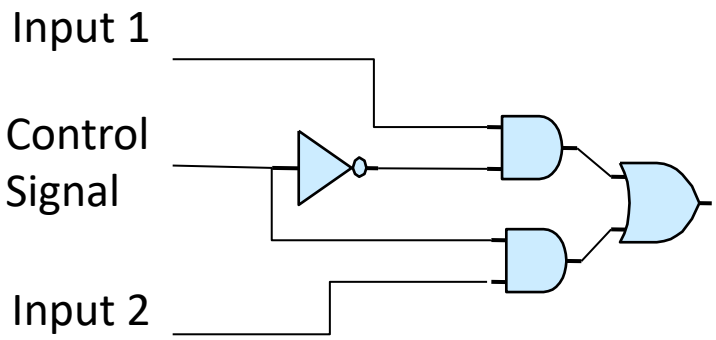
A:

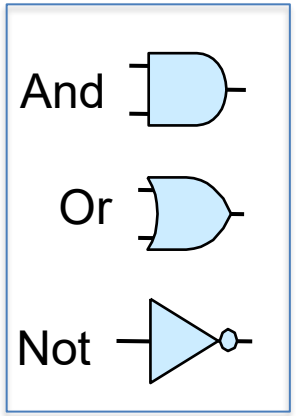




Draw the truth tables for this circuit

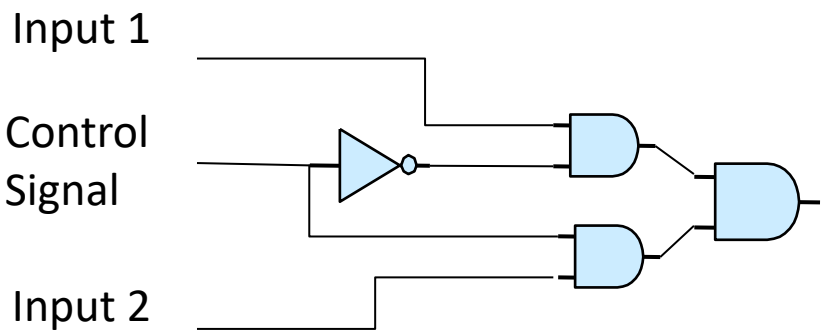
B:

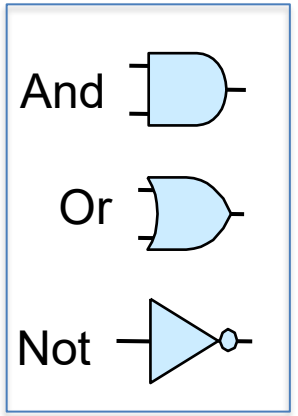




Draw the truth tables for this circuit

C:

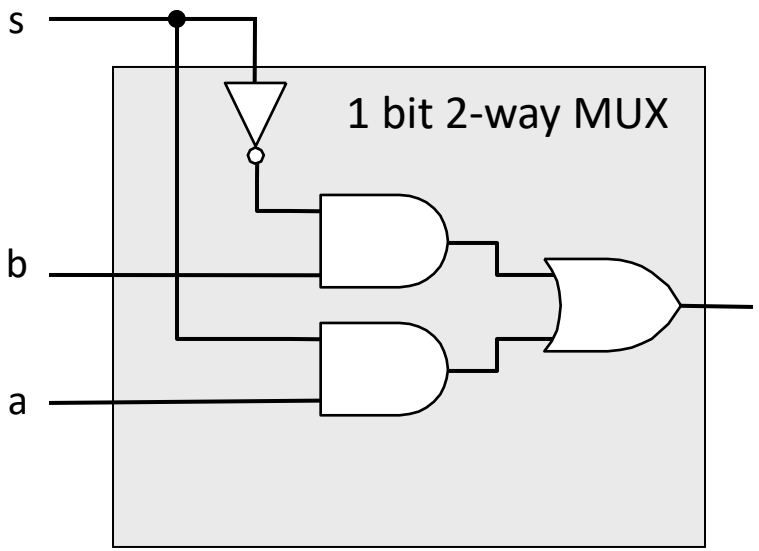




Multiplexor: Chooses an input value

Inputs: 2^N data inputs, N signal bits

Output: is one of the 2^N input values

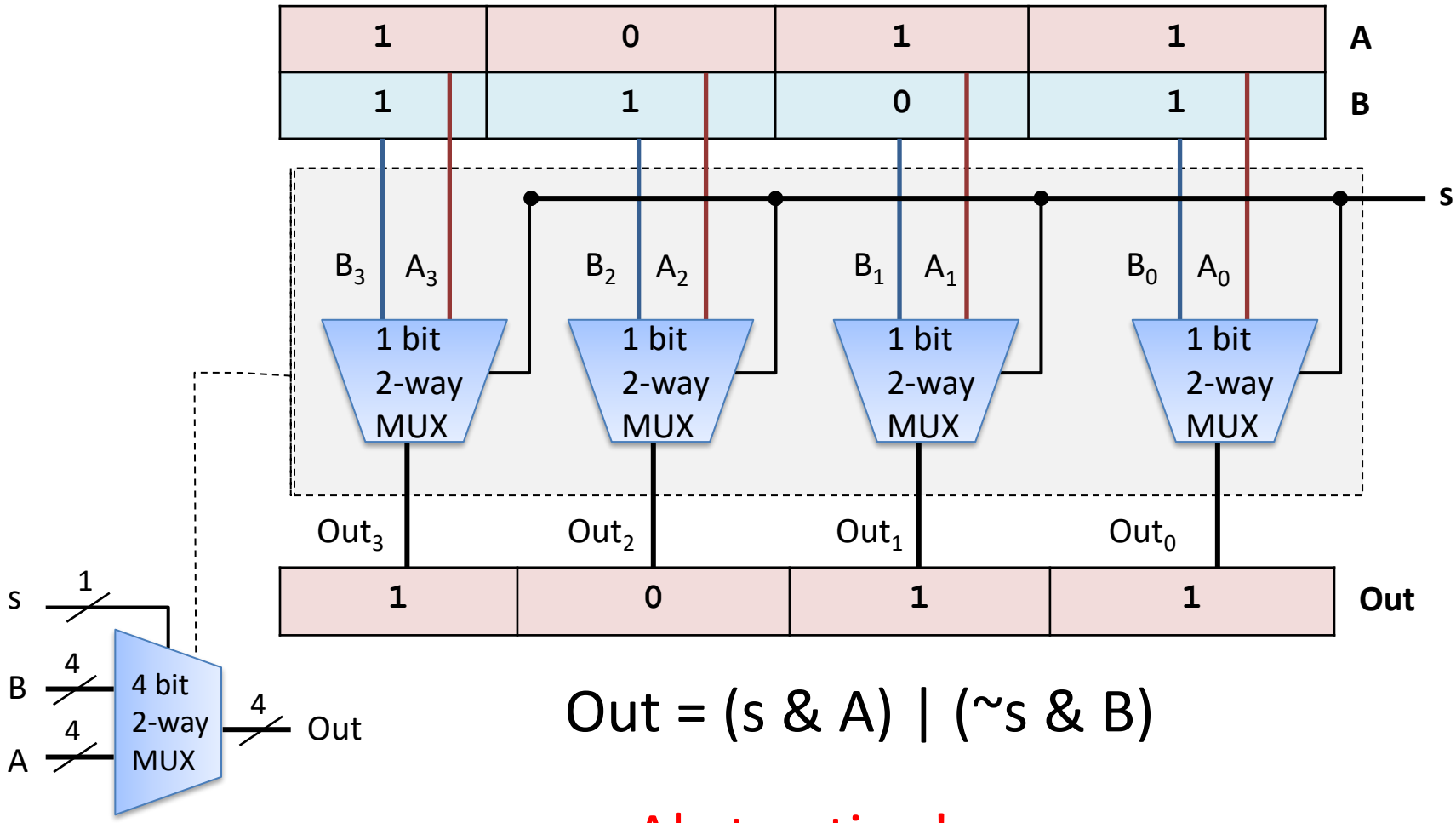



```
out = (s & a)|(~s & b)
```

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

4-bit 2-way MUX 1-bit 2-way Muxes

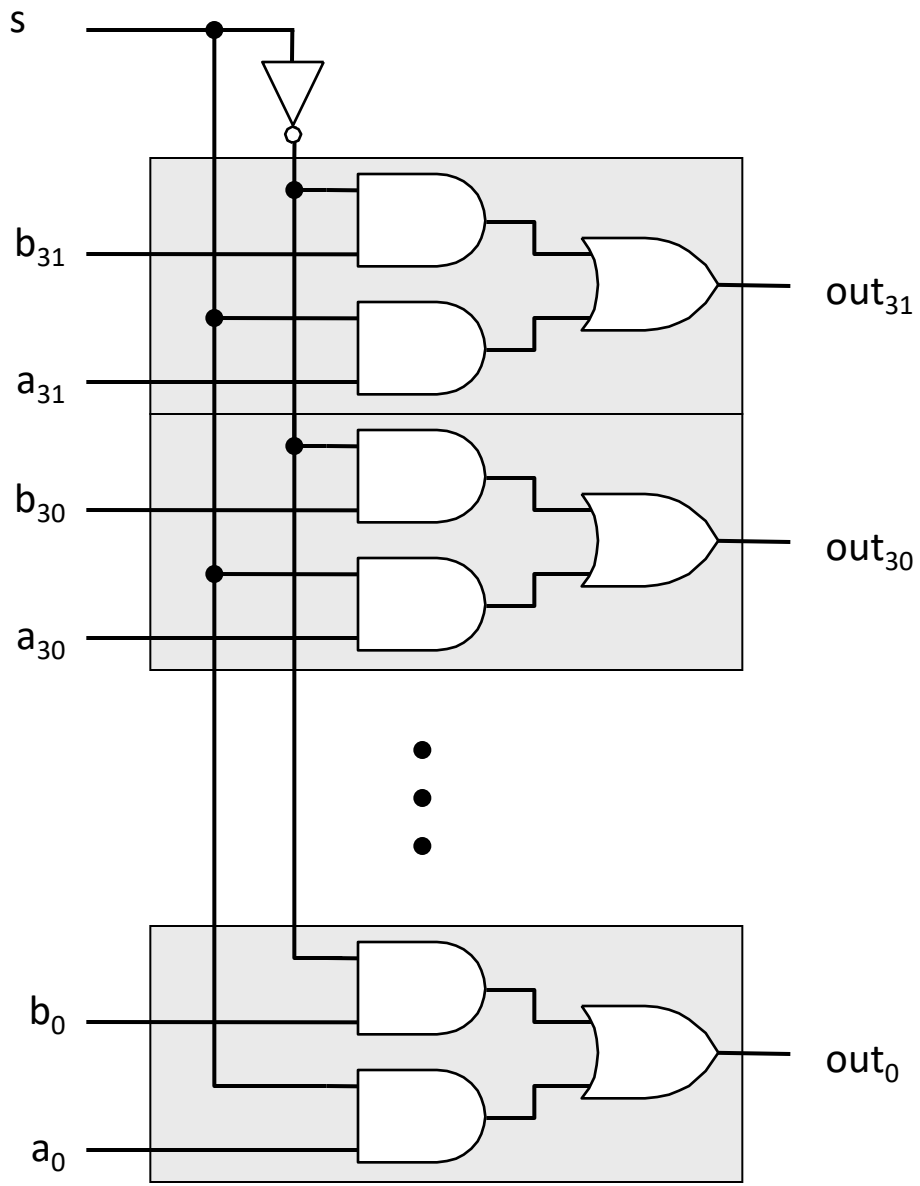
Corresponding bits of A and B fed through a 1-bit MUX

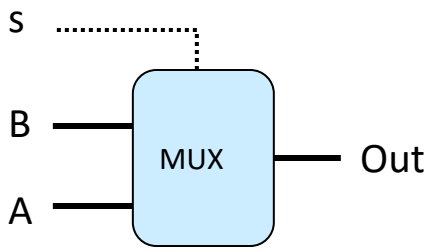


Out = (s & A) | (~s & B)

Abstraction!

32 bit 2-Way Multiplexor





Input:

two 32 bit values (a, b)

one 1bit signal s

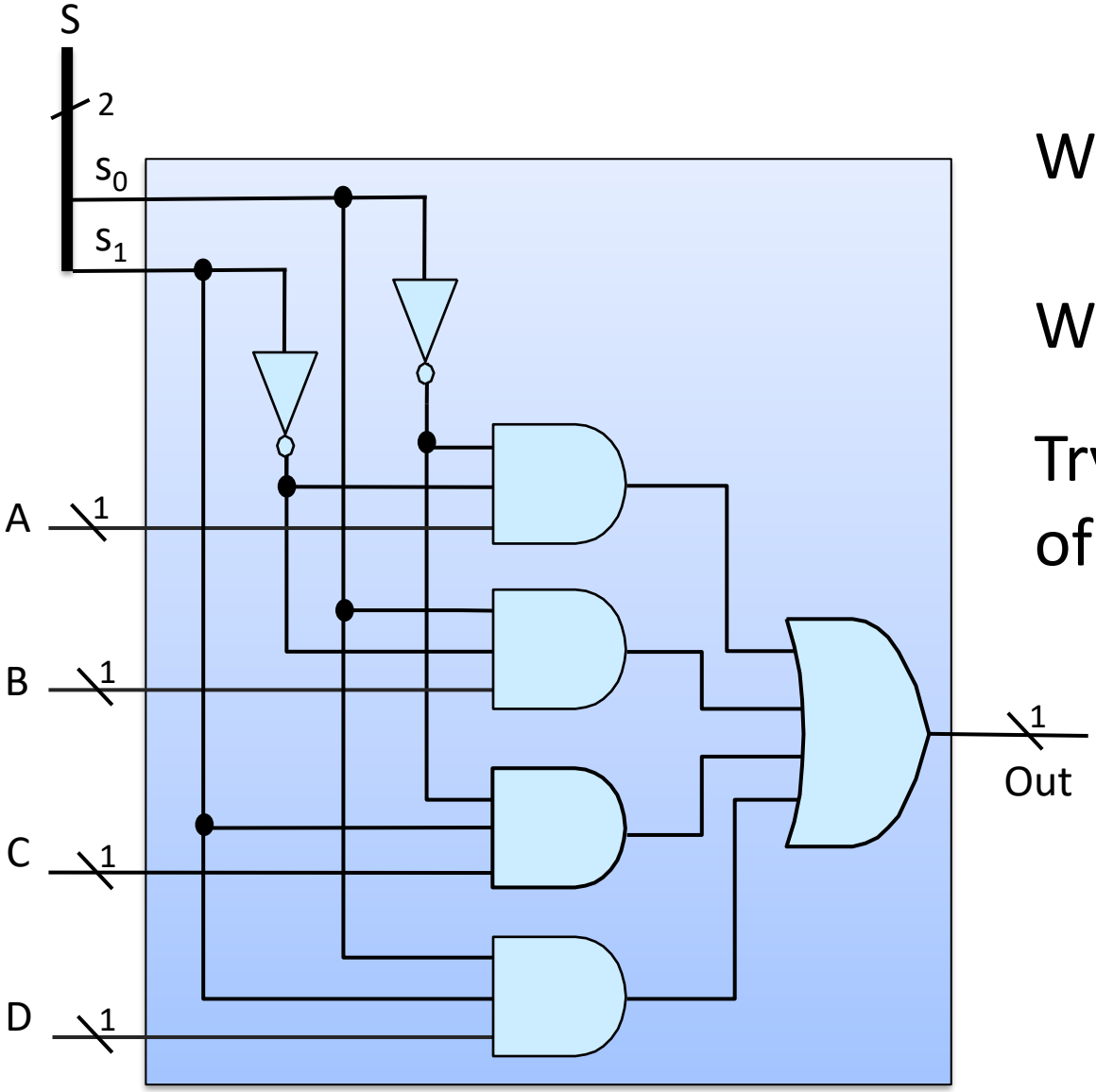
Each corresponding bit of 32 bit input values, fed through 1 bit mux

Output: 1 of the 2 inputs

One 32 bit value

(either a or b)

What does this Circuit do?



What are inputs?

What does it output?

Try for different values of S (S is 2 bits of input)

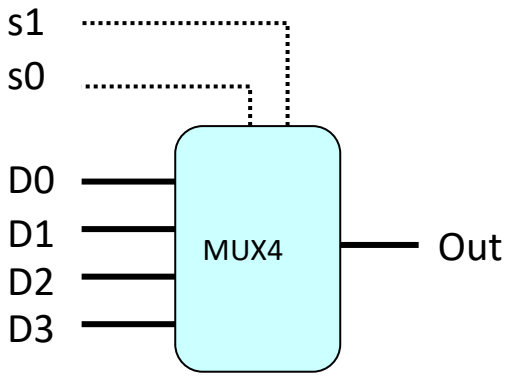
S (s_1s_0)	Out?
0 (00)	
1 (01)	
2 (10)	
3 (11)	

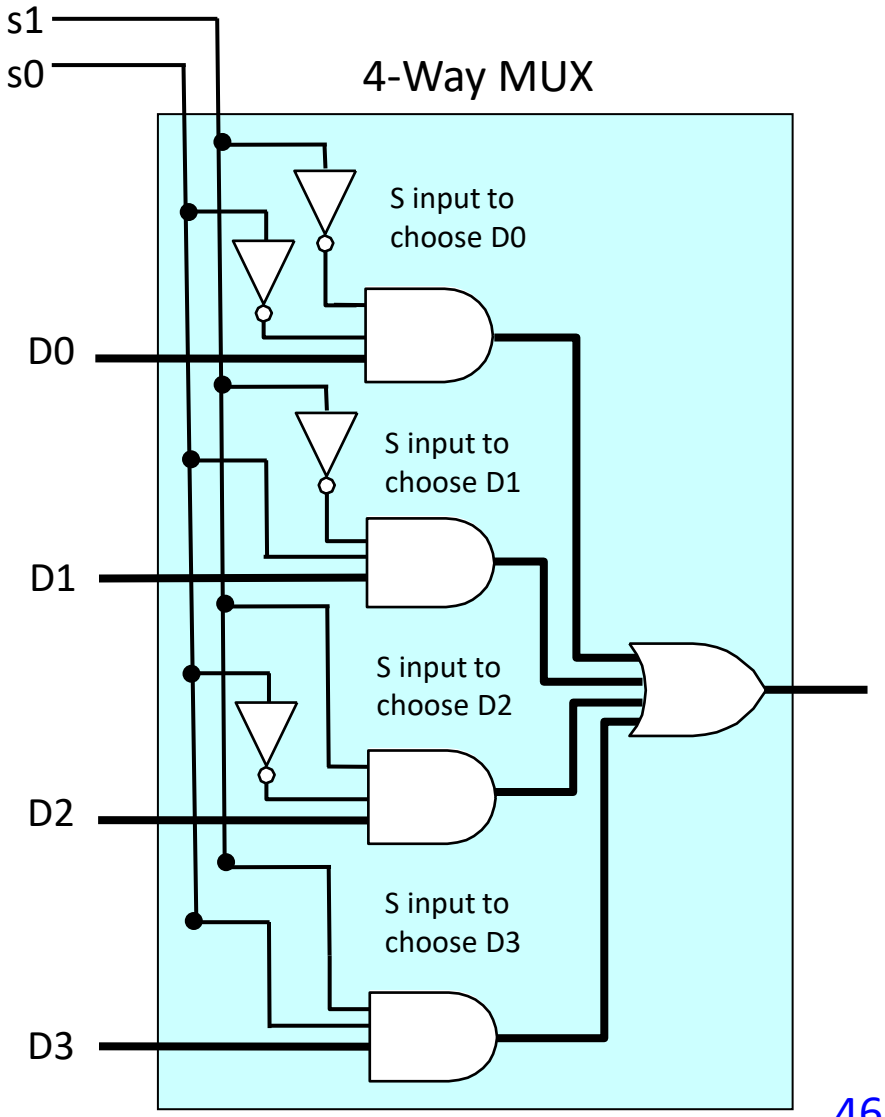
N-Way Multiplexor

Choose one of N inputs

need $\log_2 N$ select bits

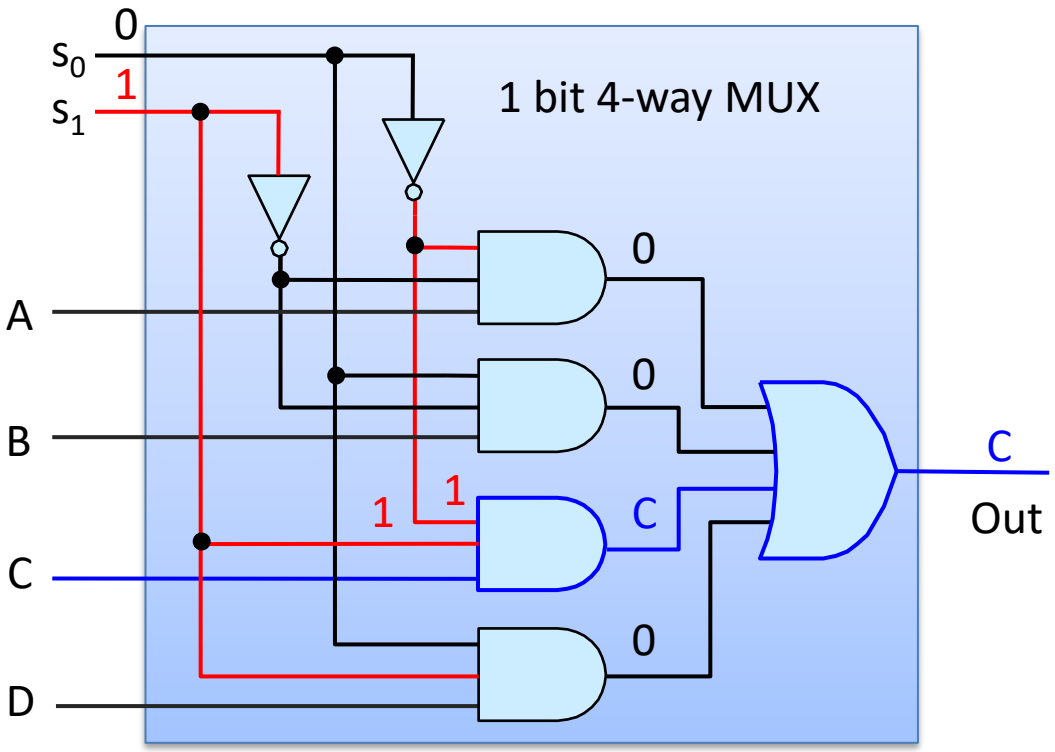
s1	s0	choose
0	0	D0
0	1	D1
1	0	D2
1	1	D3



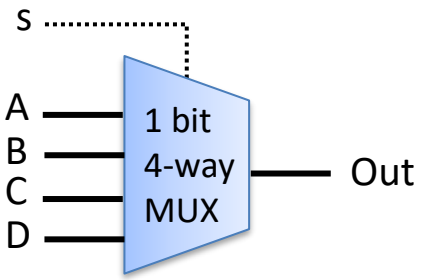


Example, 1-bit 4-way MUX

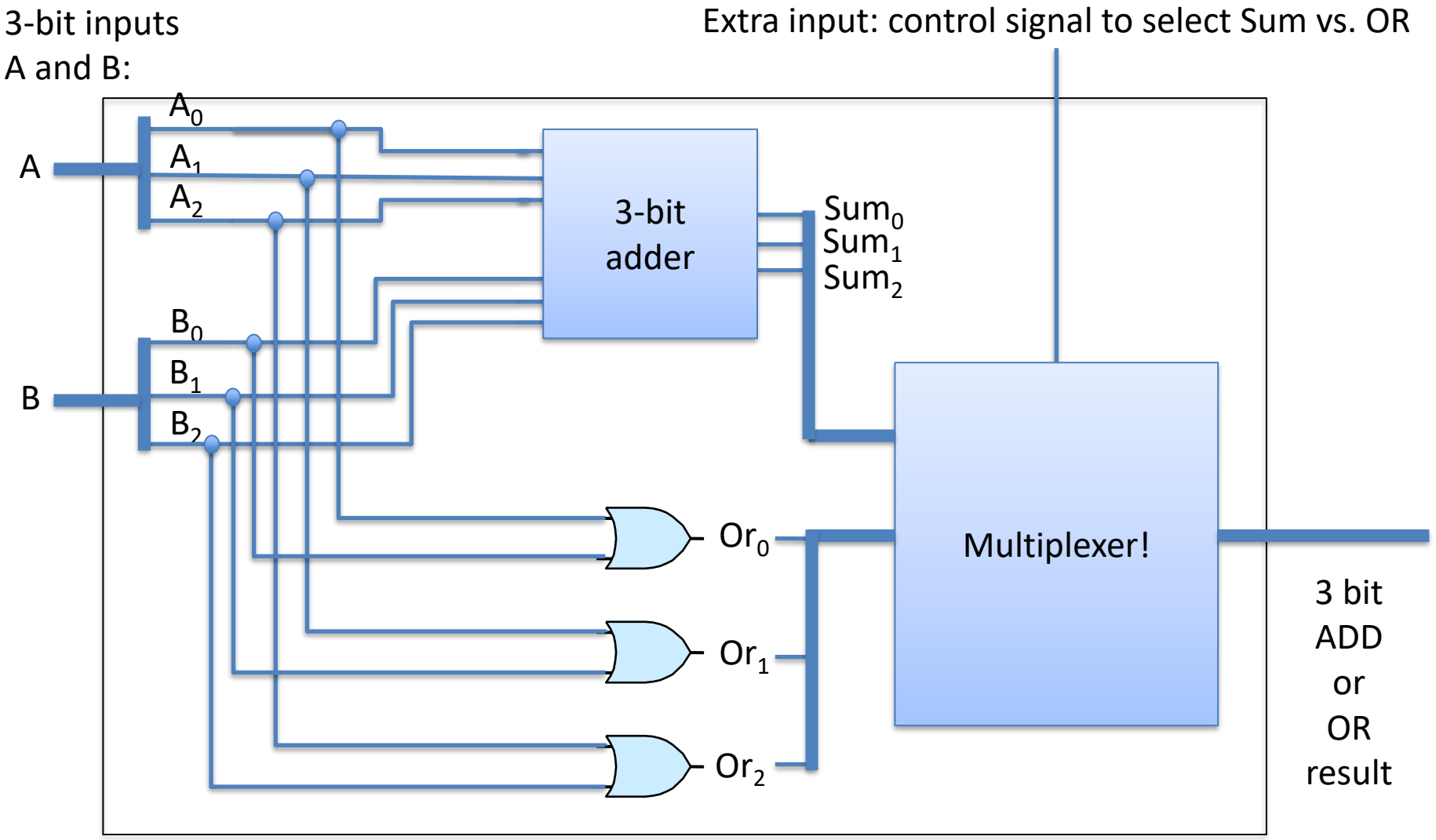
When select input is 2 (0b10): C chosen as output



S	Out
0	A
1	B
2	C
3	D



Simple 3-bit ALU: Add and bitwise OR



Slides by Tia Newhall, CS31, Swarthmore

Where do Select bits come from?

- Encoded in the bits of the CPU instruction to execute in IR:

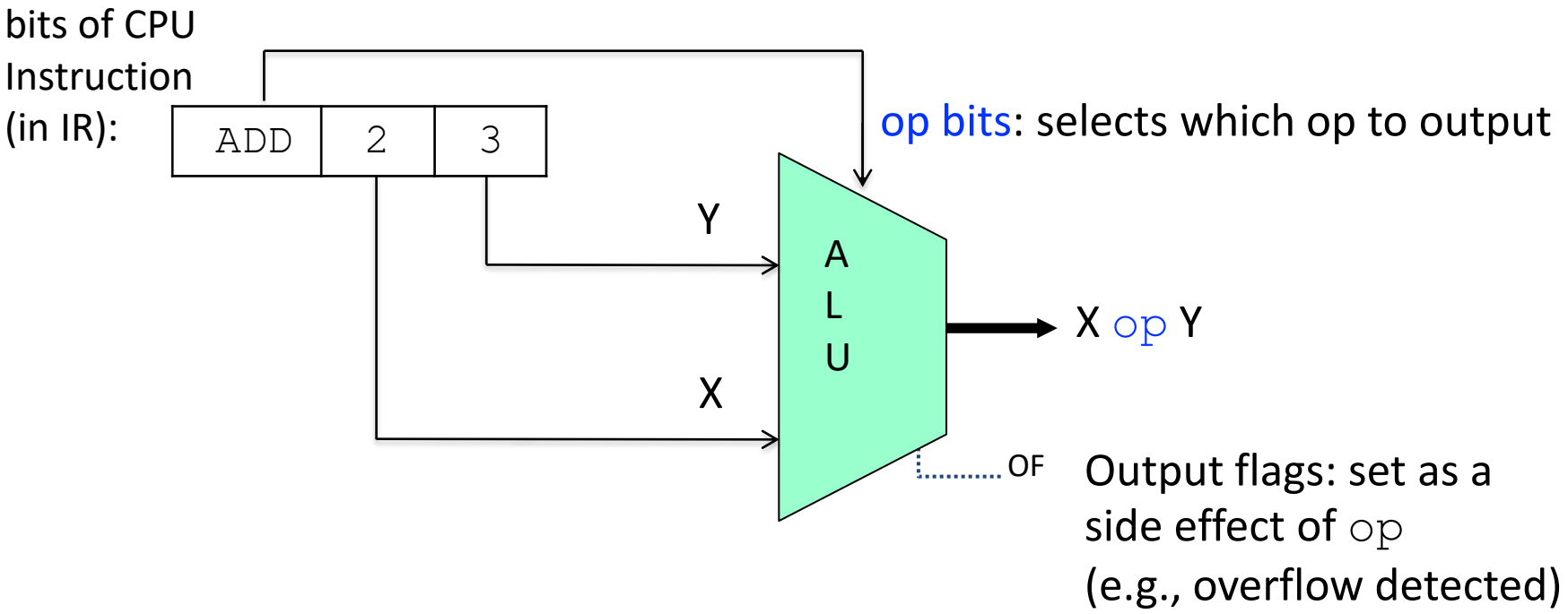
IR:	10010011

- Instruction encodes information about
  - The operation to perform (OP): selects the operation
  - Sometimes operands for the operation (ex. val1, val2)
  - Sometimes destination for result

IR:	**10**	010	011
encodes:	OP	val1	val2
e.g.:	**ADD**	2	3

compute: 2 + 3

Summary: CPU so far



- Arithmetic and logic circuits: ADD, SUB, NOT, …
- Control circuits: MUX 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

Summary: 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

Building a CPU: Storage

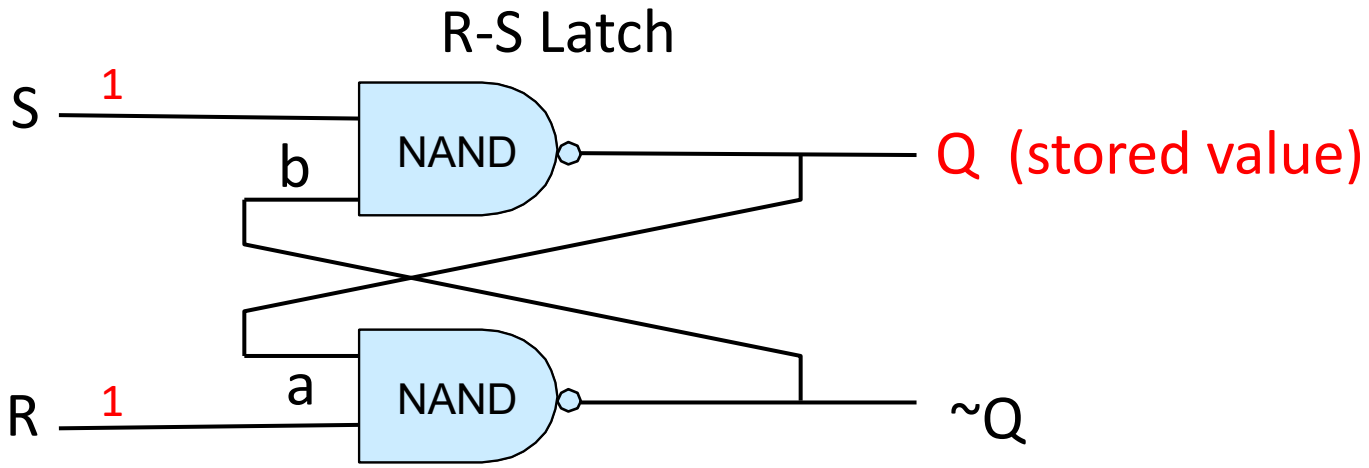
Goal: Give the CPU a “scratch space” to perform calculations and keep track of the state its in.

Baby steps:

- 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

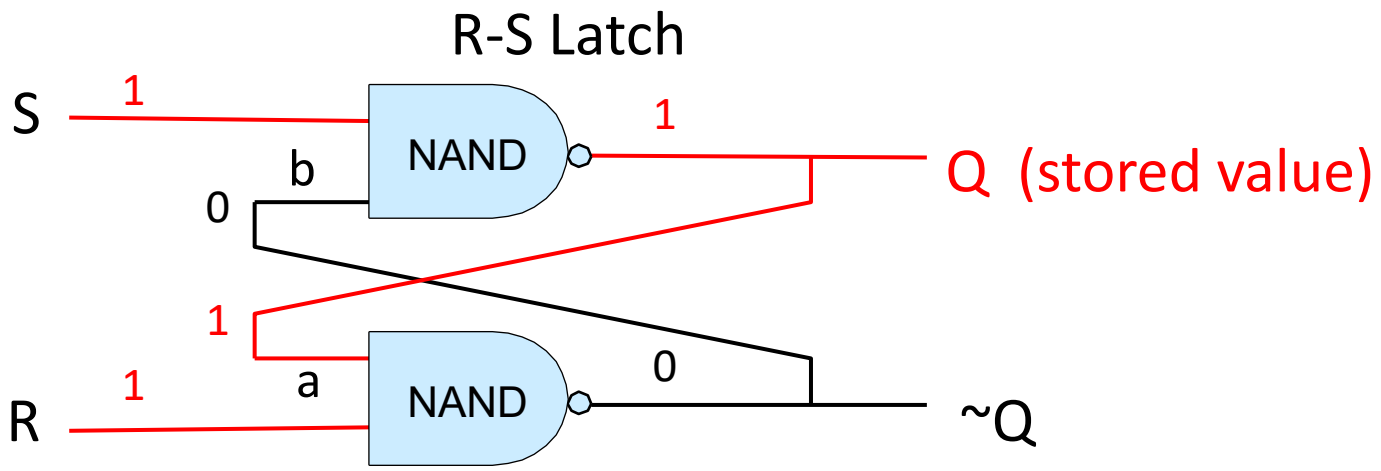
- To store a value need a circuit with a Cycle
- Value is Stored when R and S are both 1
  - Stored value, Q, is stable in this circuit
  - External control circuitry ensures R and S are 1



ex. latch stores 0 if a is 0: R =1 --> b is 1, S =1 and b = 1 --> Q is 0

Latch Stores value when R and S both 1

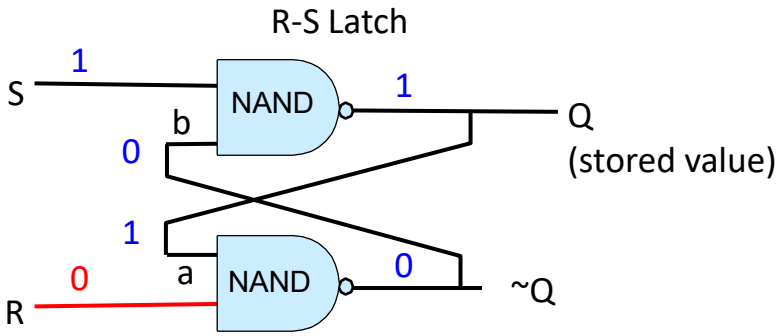
(ex) if a is 1: R= 1 --> b is 0, S= 1 and b= 0 --> Q is 1, and a is 1 …



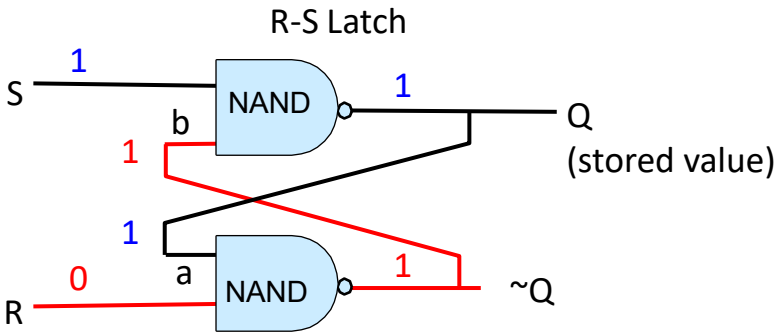
The latch stores 1

Ex. to write (and store) 0 into Latch

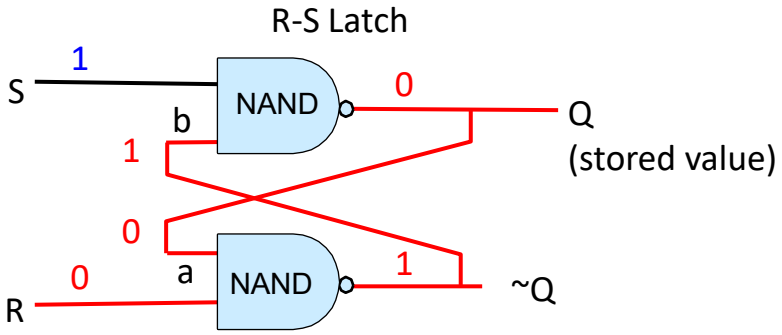
- Set R to 0 momentarily (S stays at 1)



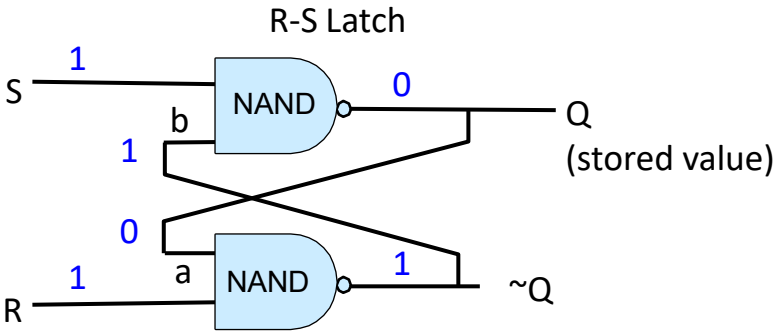
A. Set R to 0 to store 0



B. Changes lower NAND output to 1



C. Changes upper NAND output to 0

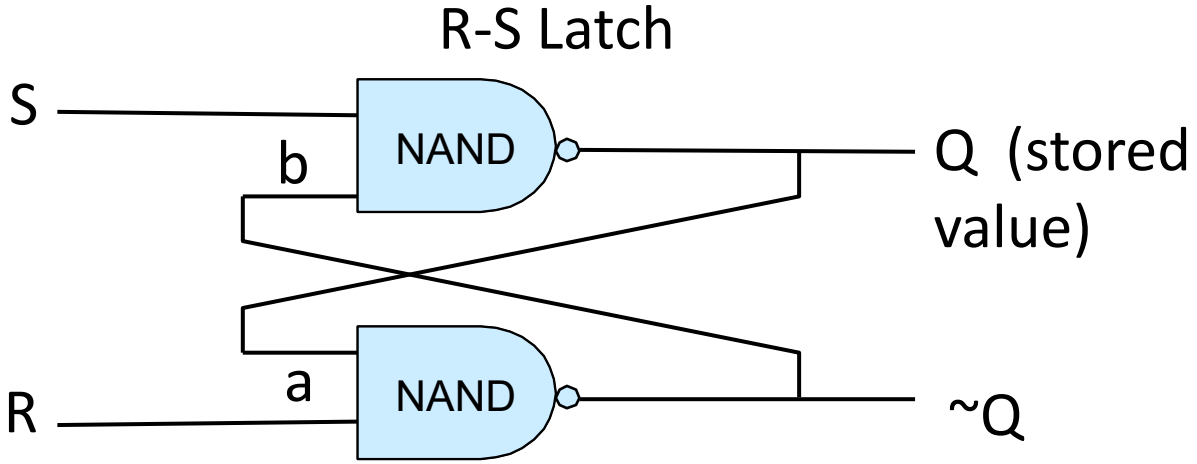


D. R-S Latch Now Stores 0
(R can be set back to 1 and still stores 0)

R-S Latch: Stores Value Q

- When R an S are both 1: Store a value

  R and S are never both simultaneously 0

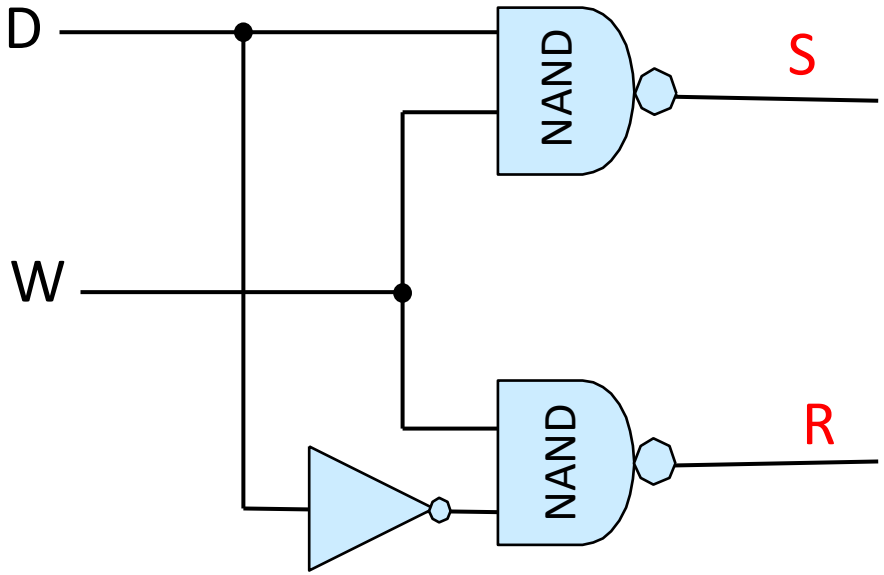


if a is 1: R= 1 --> b is 0, S= 1 and b= 0 --> Q is 1: latch stores 1

if a is 0: R =1 --> b is 1, S =1 and b = 1 --> Q is 0: latch stores 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

Given the circuit below, can outputs S and R ever simultaneously both be 0?

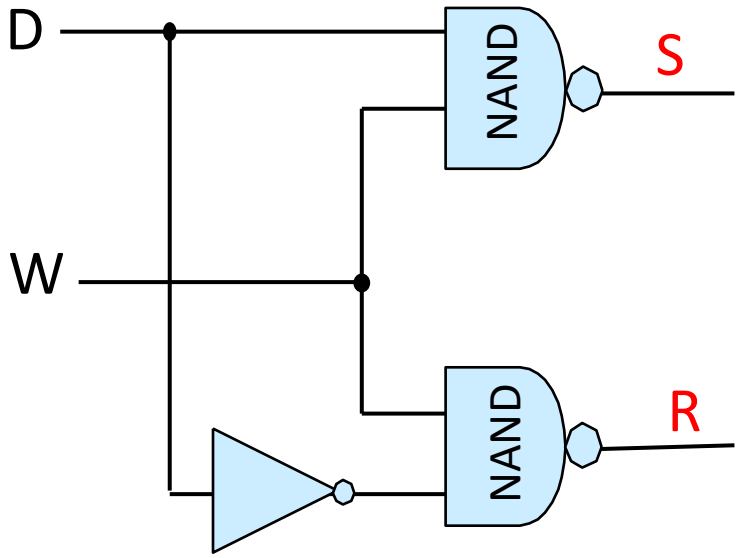


A: yes

B: no

C: maybe

Draw the truth table for this circuit



D	W	S D NAND W	R ~D NAND W

When are R and S 1?

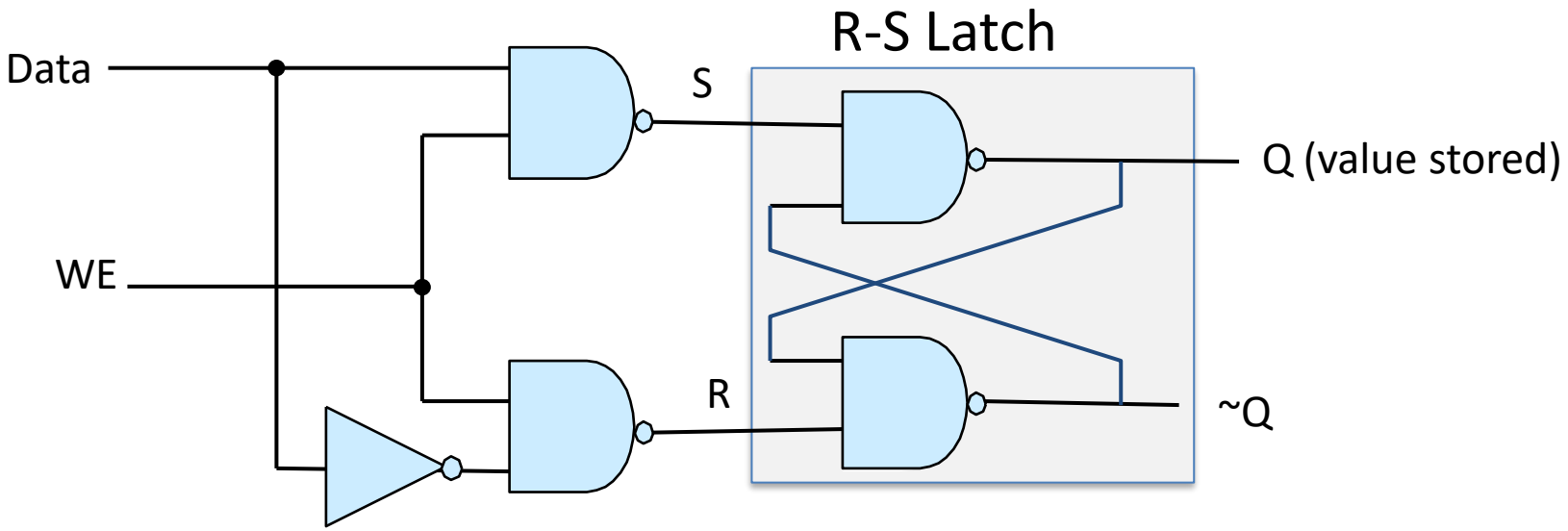
When are either R or S 0?

When is R 0?

When is S 0?

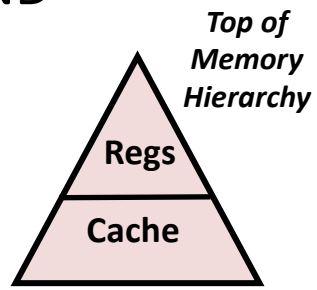
Gated D Latch

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



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

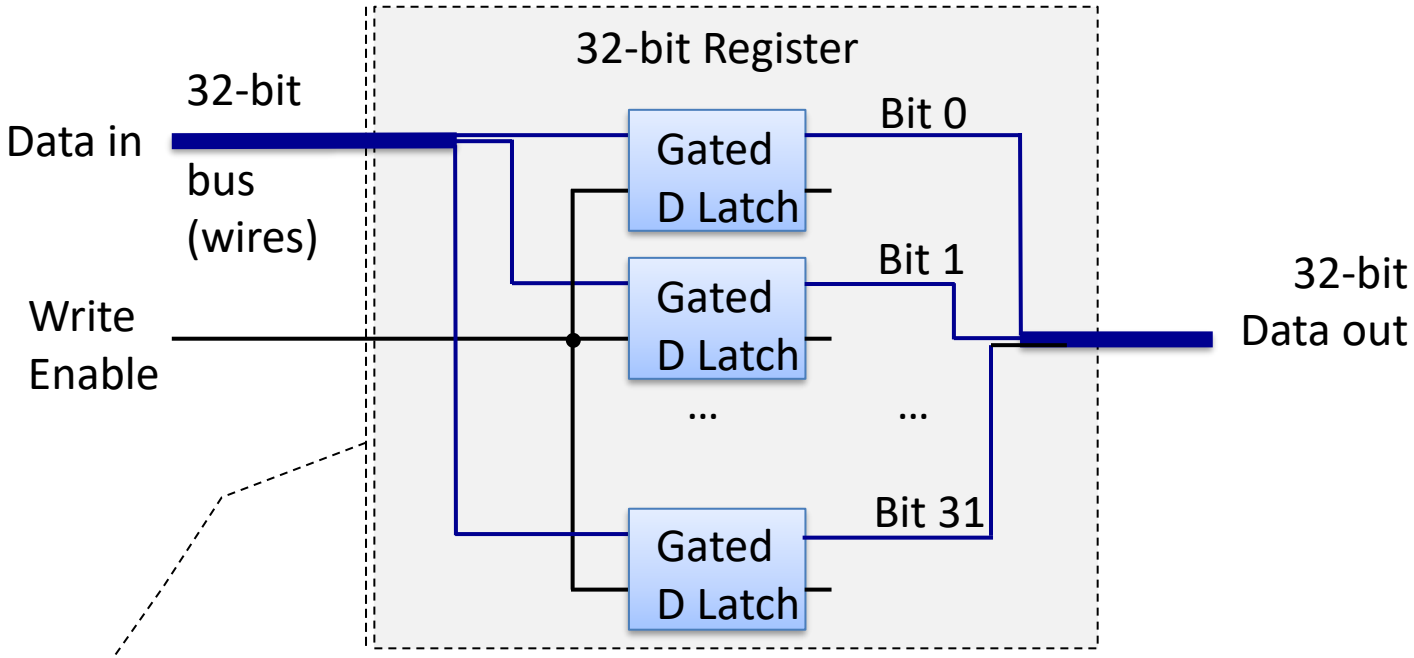
Latches are used for register and SRAM cache memory
Fast, not very dense, expensive



DRAM, used for RAM, is capacitor-based storage

An N-bit Register

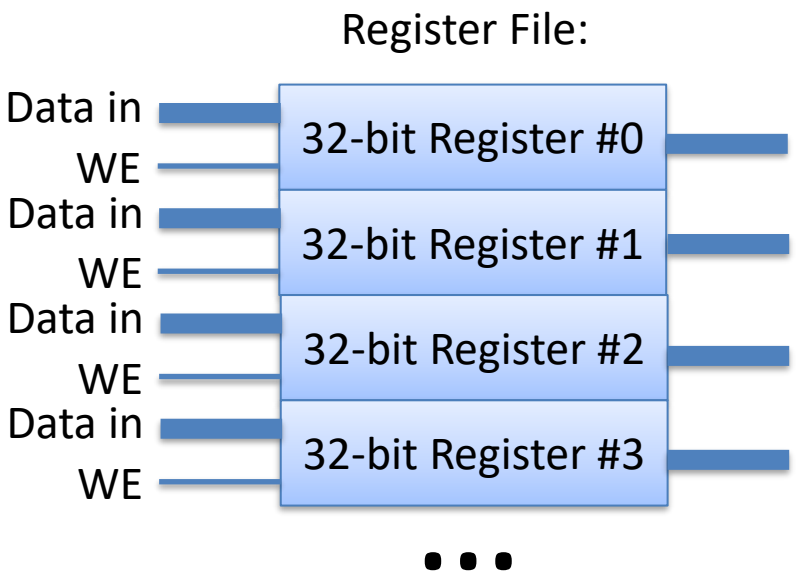
- Fixed-size storage (8-bit, 32-bit, etc.)
- Gated D latch stores one bit
  - Connect N of them to the same write-enable wire!



Abstraction!

Register "File"

- A set of general-purpose registers for the CPU to store temporary values.



- Instructions of form:
  - "add value in R1 and R2, store result in R3"

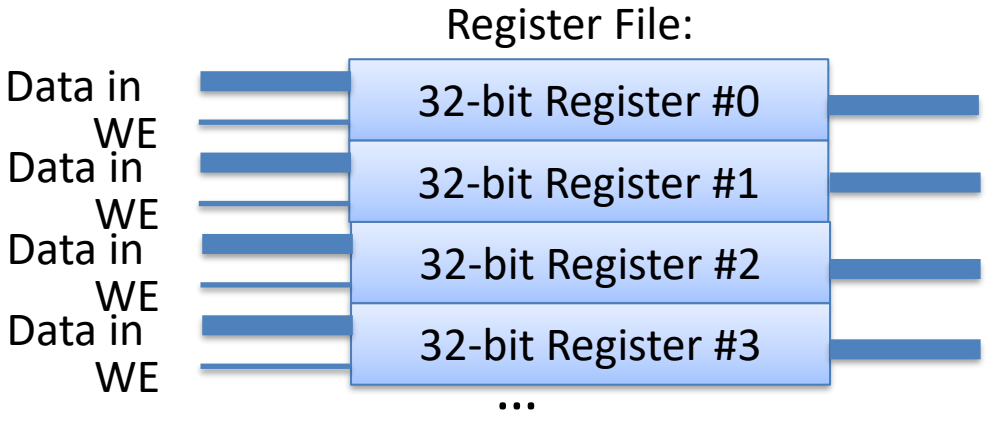
Register File Interface

For temporary value storage of operands and results:

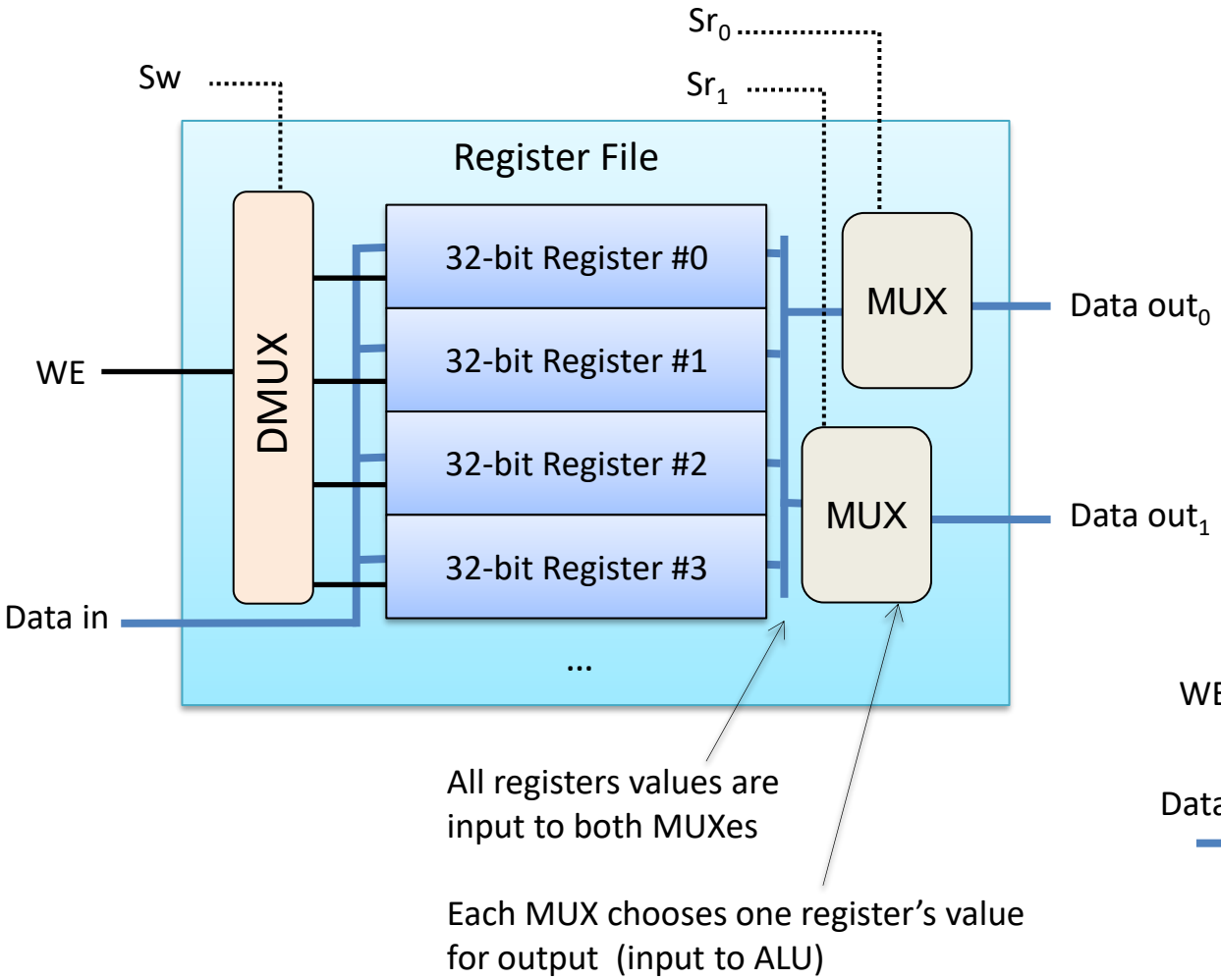
ADD Rx Ry Rz "add value in Rx and Ry, store result in Rz"

- 2 Outputs: for 2 operands of ALU operation
  - Read Register X's value
  - Read Register Y's value
- 1 Data input (+ WE): for writing result to Register Z

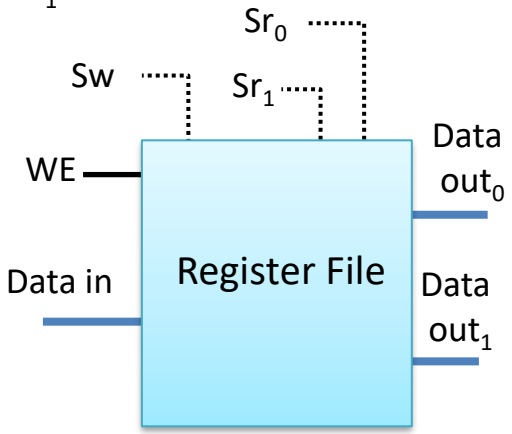
- Need to pick two Registers for outputs (Rx and Ry)
- Need to enable exactly one WE to Register (Rz)



- Need to pick two Registers for outputs (Rx and Ry)
- Need to enable exactly one WE to Register (Rz)



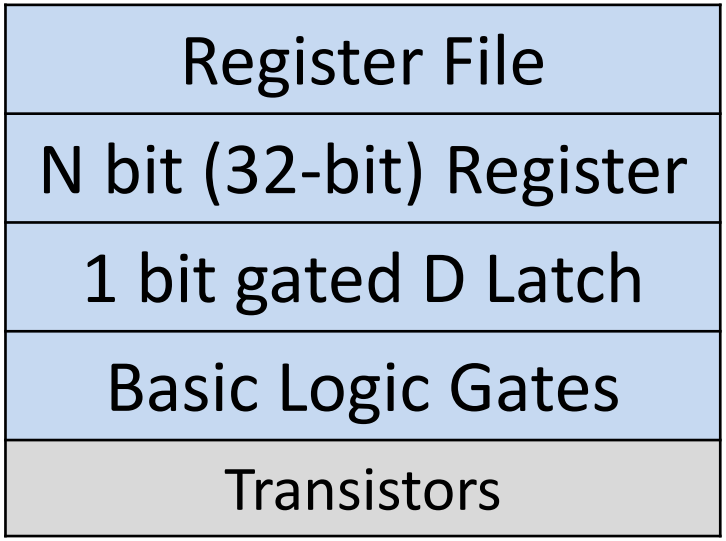
DMUX:
inverse of Multiplexer
1 input, N outputs
+ select input chooses one of N outputs to send input



Summary: Storage Circuits

Lots of abstraction going on here!

- Logic Gates hide the details of transistors (the building blocks of Logic Gates)
- Build 1 bit gated D latches from basic logic gates.
- Combine multiple latches to get one N-bit register.
- Grouping N-bit registers gives us register file.



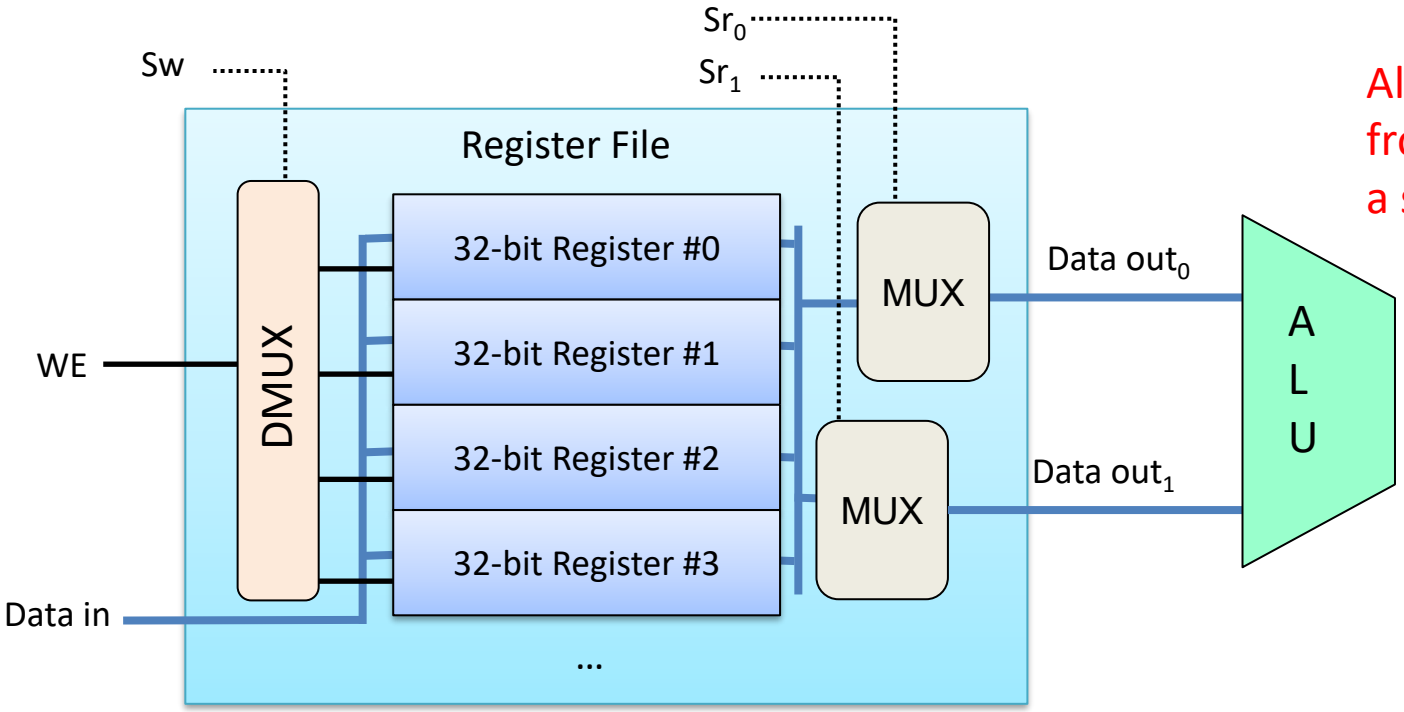
Summary CPU so far…

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.

Summary: 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

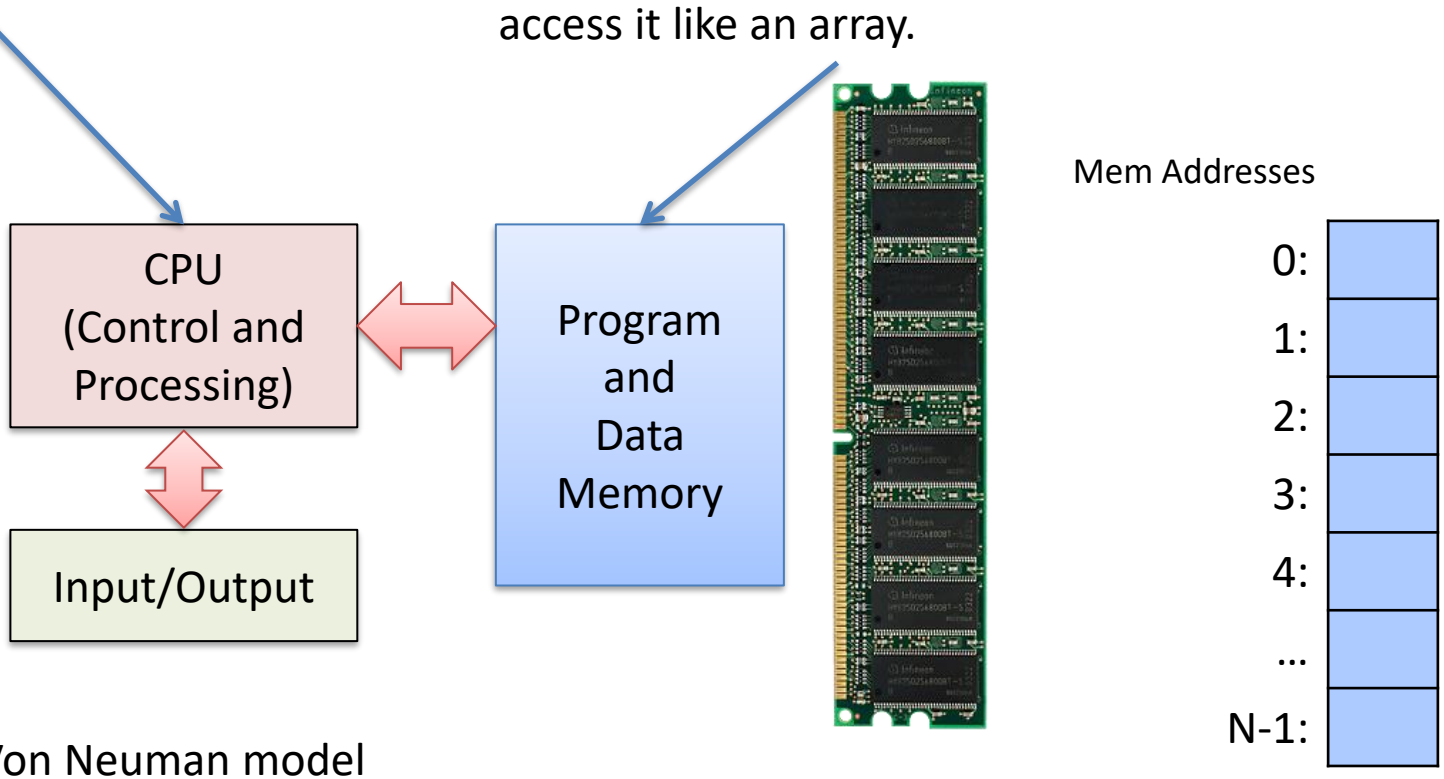
3. Control: support/coordinate instruction execution

   (ex) fetch the next instruction to execute

Building a CPU: Control

We're building this.

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



Recall the Von Neuman model

Recall: executing instructions

1. Fetch instruction from memory

2. Decode what the instruction is telling us to do
   - Tell the ALU what it should be doing
   - Find the correct operands

3. Execute the instruction (arithmetic, etc.)

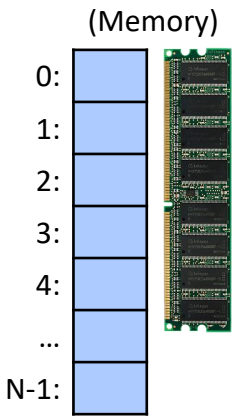
4. Store (Write Back) the result

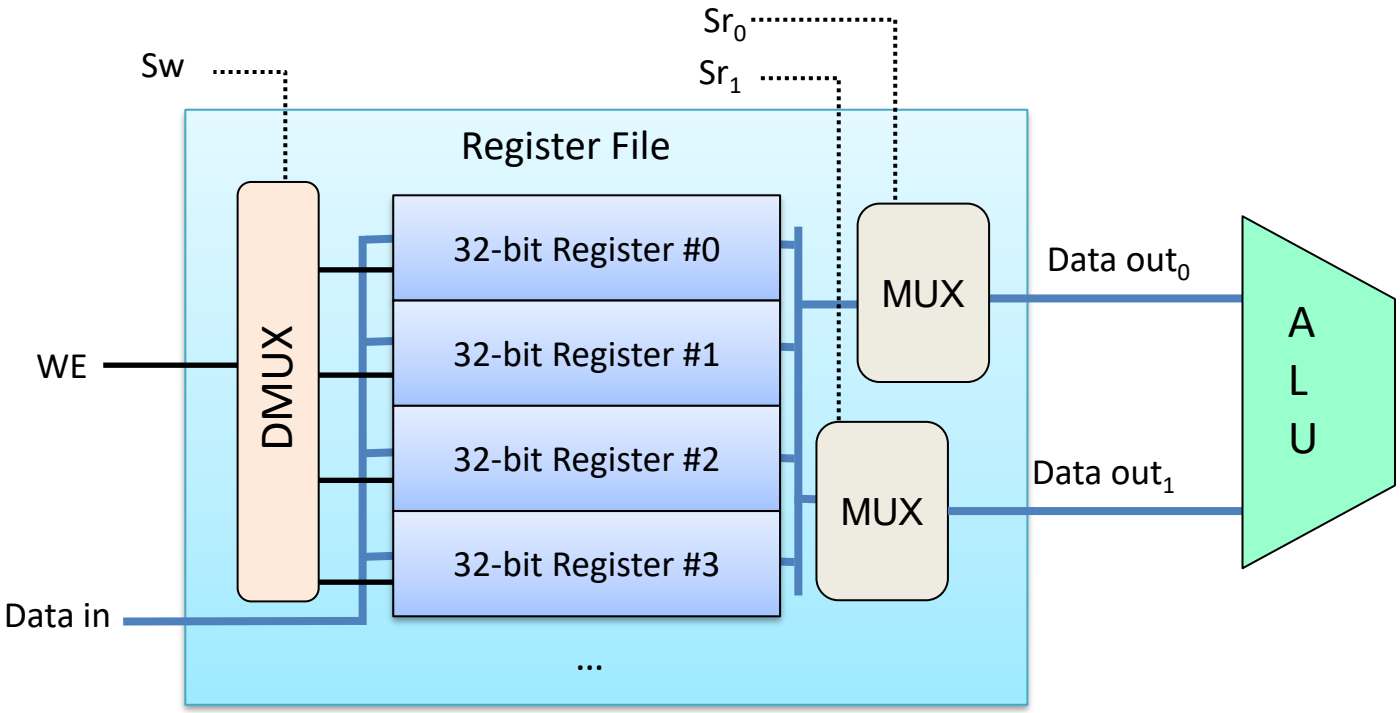
Program State

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)**





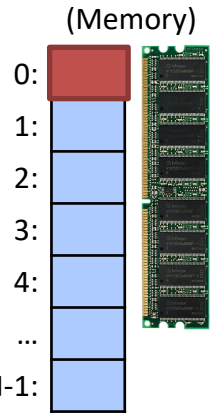
Fetching instructions.

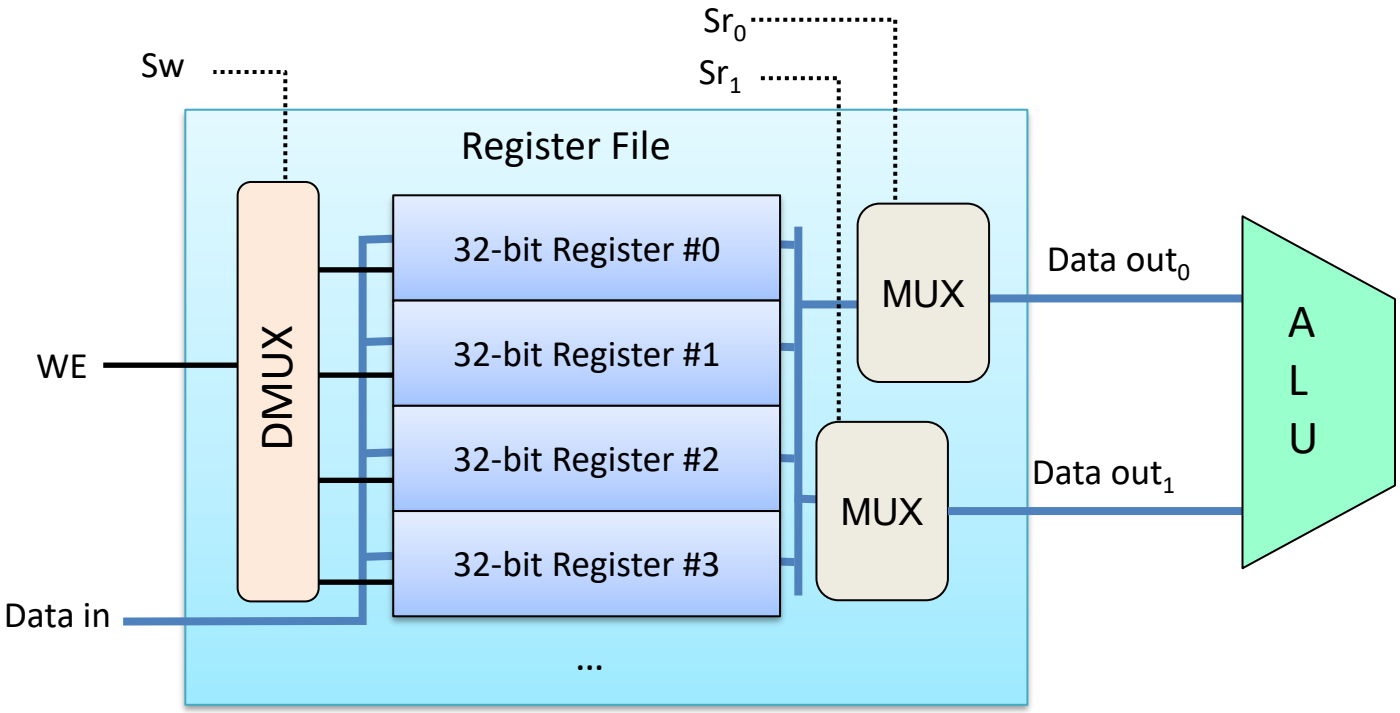
Load IR with the contents of memory at the address stored in the PC.

Program Counter (PC): **Address 1**

Instruction Register (IR): Instruction at Address 0

Increment PC to
Addr of next instrunction





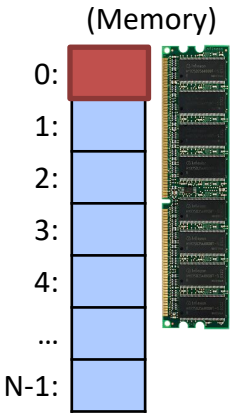
Slides by Tia Newhall, CS31, Swarthmore

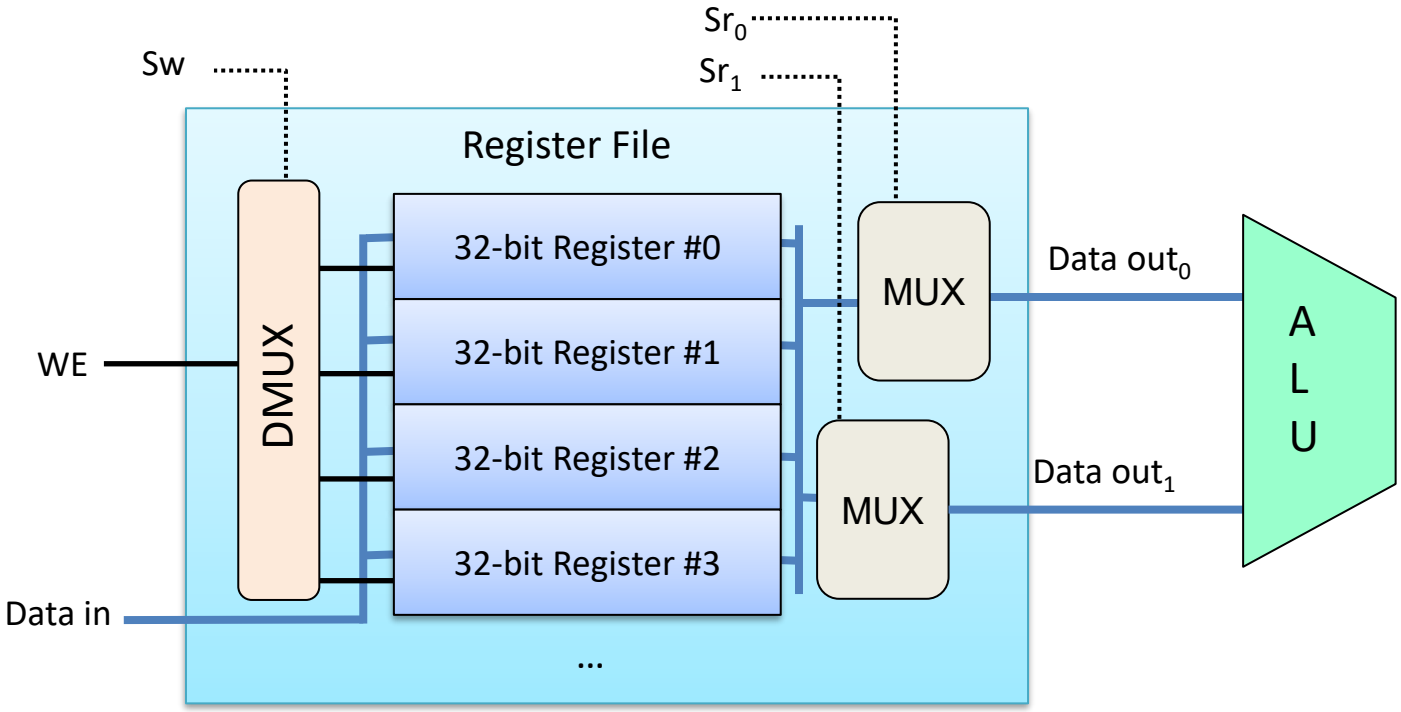
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): **Address 1**

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





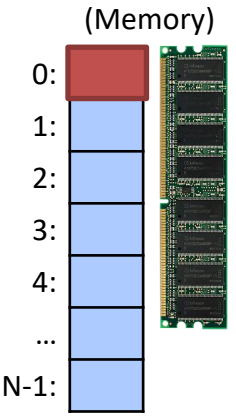
Slides by Tia Newhall, CS31, Swarthmore

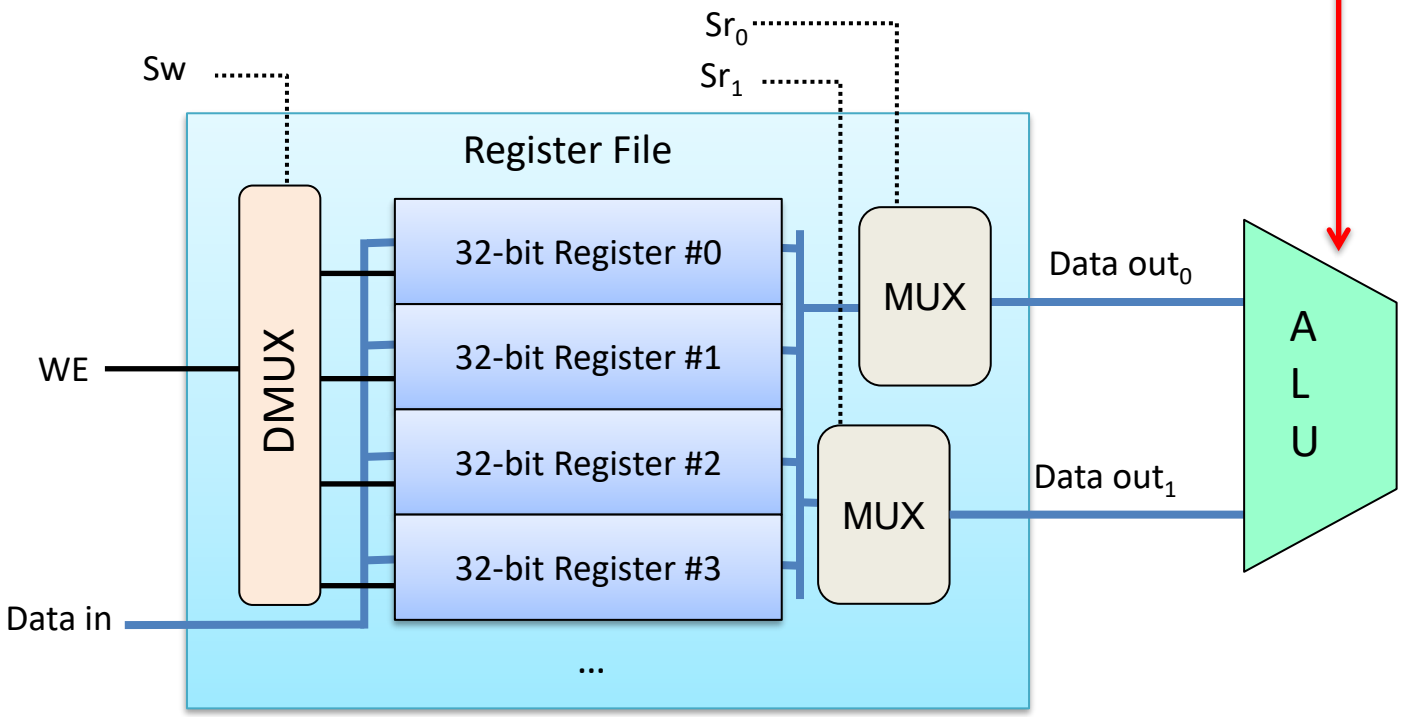
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 1

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

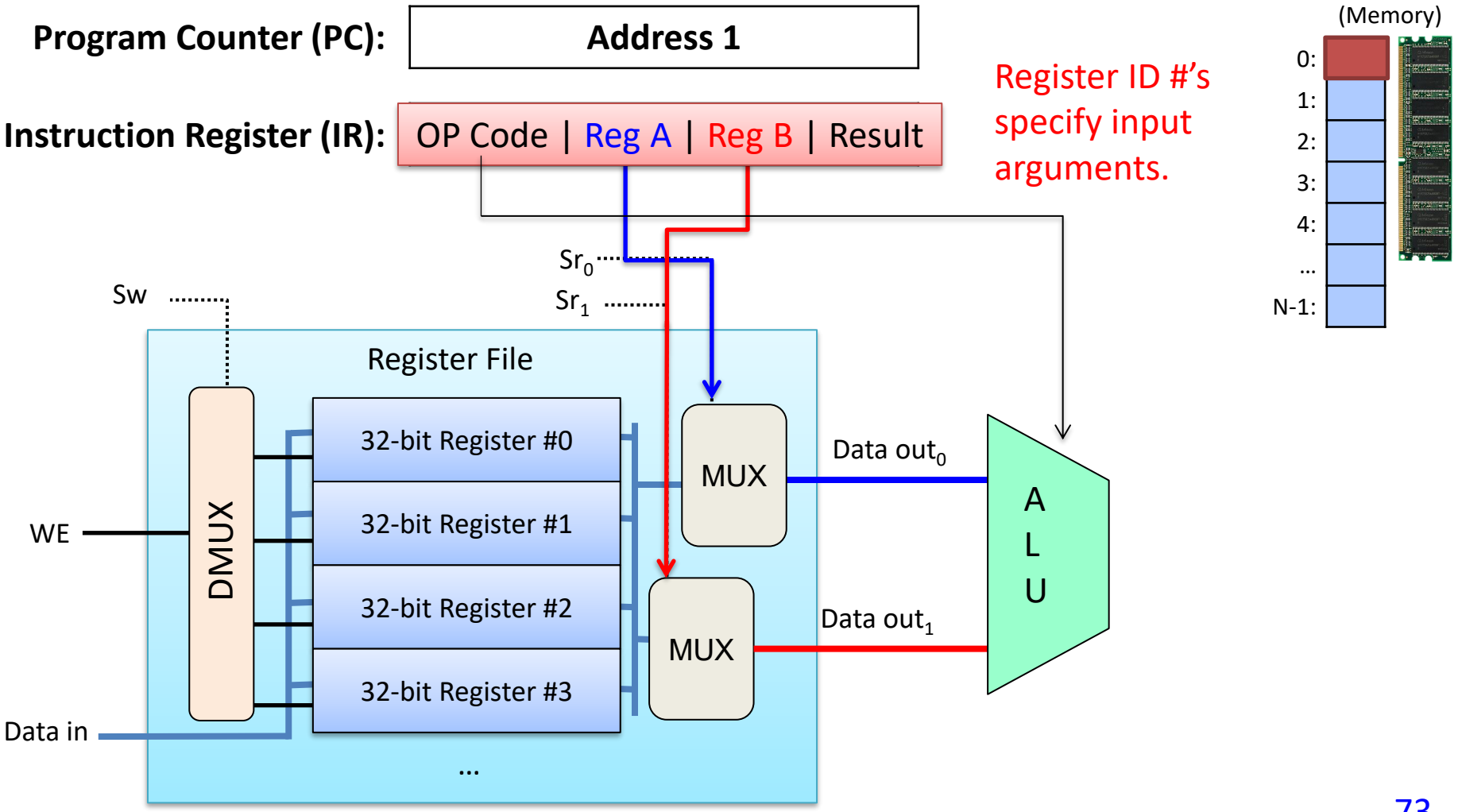




OP Code selects which ALU operation to perform.

Decoding instructions.

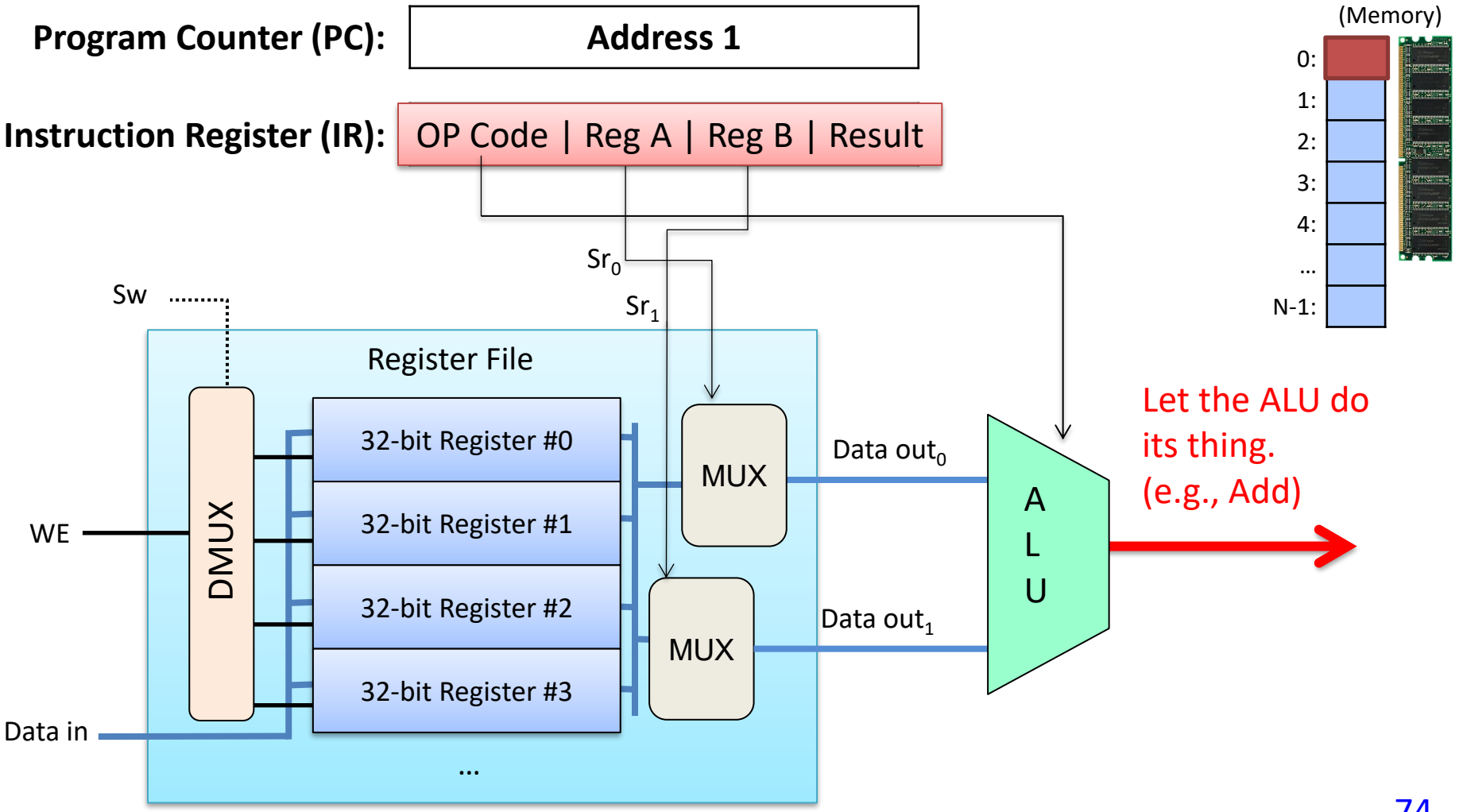
Interpret the instruction bits: What operation? Which arguments?



Slides by Tia Newhall, CS31, Swarthmore

Executing instruction.

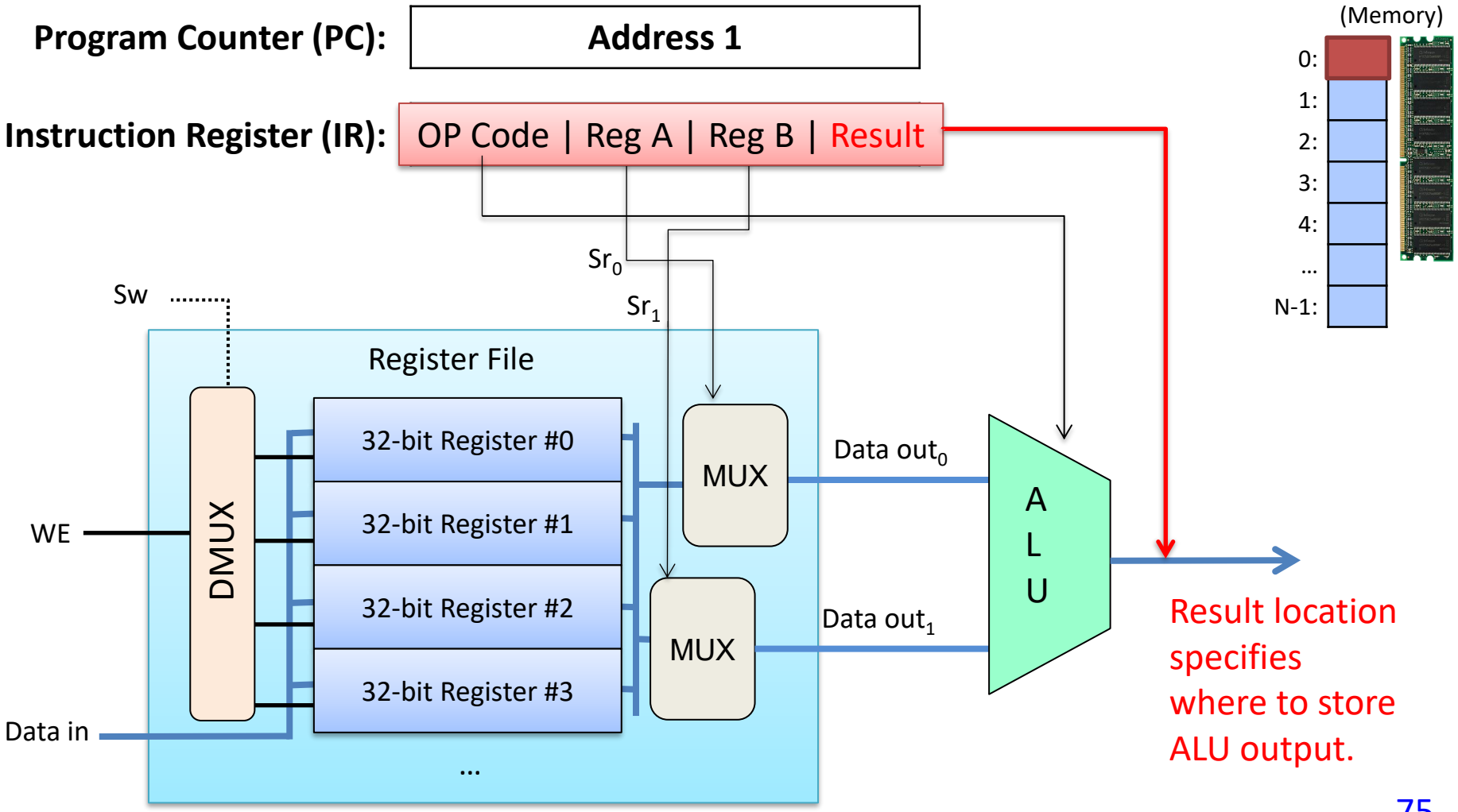
Let the ALU perform the operation on the selected operands



Slides by Tia Newhall, CS31, Swarthmore

Storing Result

ALU has just computed a result. Where to put ALU output?

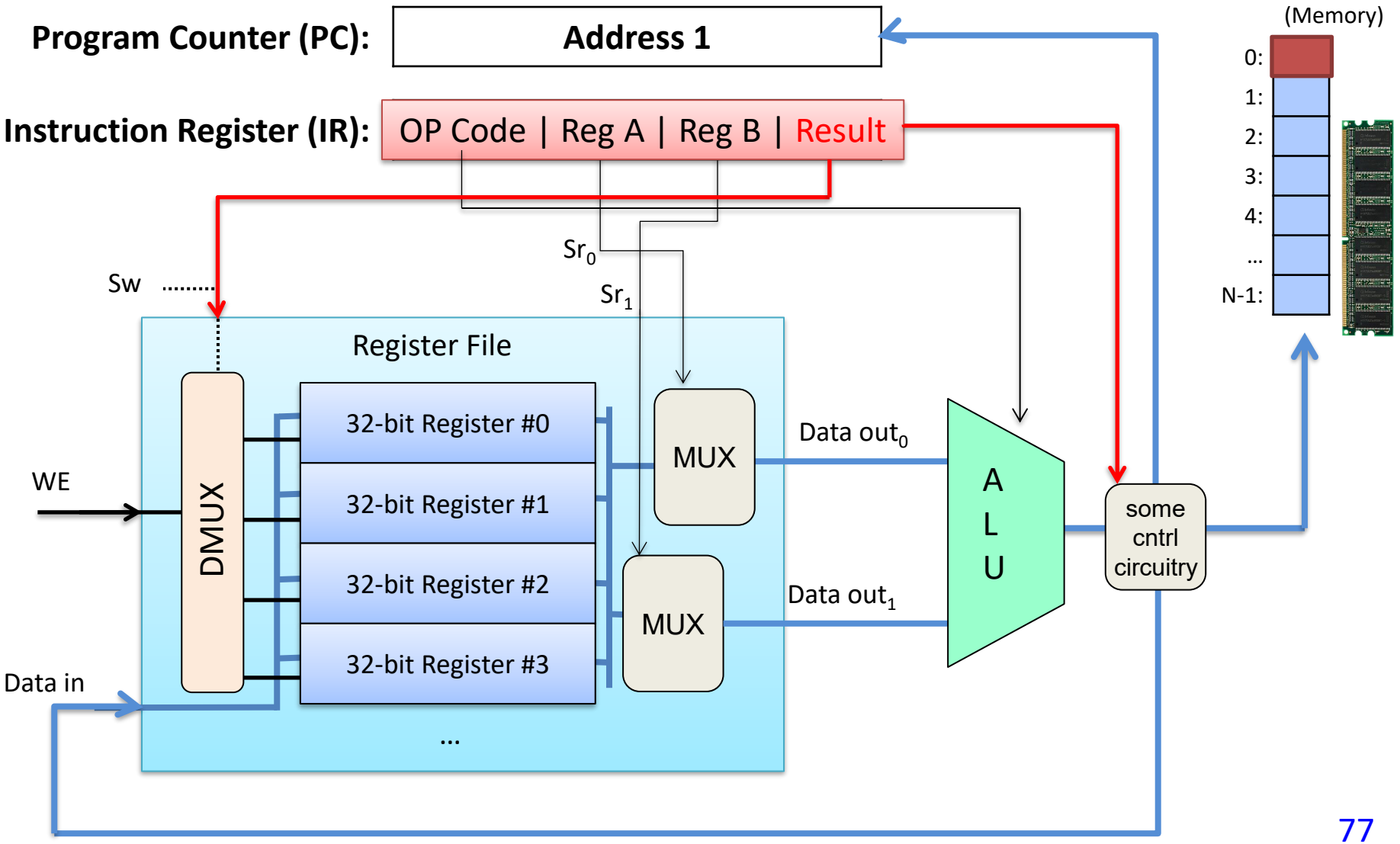


Slides by Tia Newhall, CS31, Swarthmore

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

Storing Result

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



Slides by Tia Newhall, CS31, Swarthmore

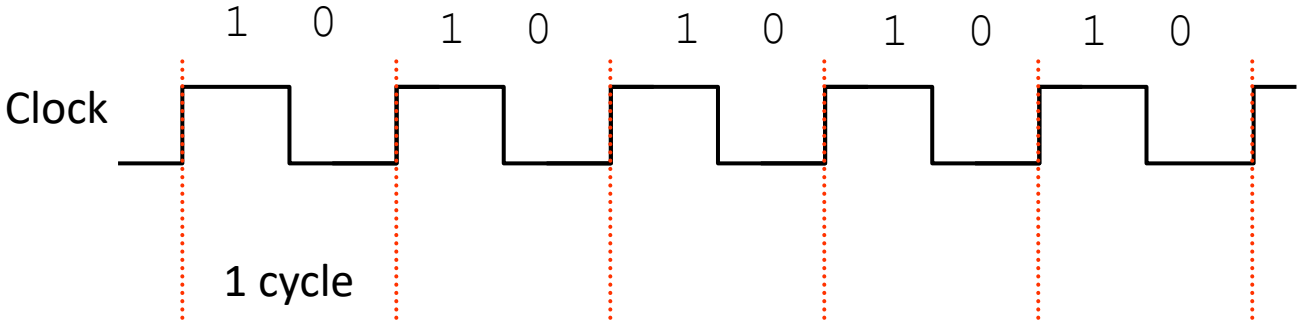
Clocking

- 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
    - Ripple carry adder: needs to get all the way to the end
  - 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:

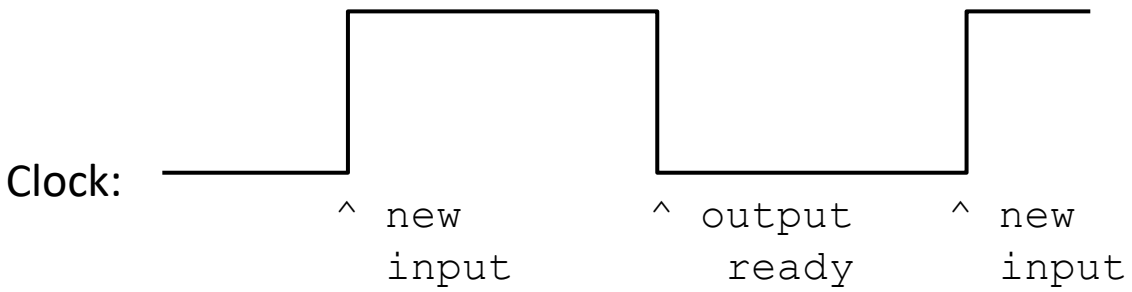
      1 billion cycles/second → 1 cycle every nanosecond (ns)

      (1ns = 10^{-9} secs)

Clock and Circuits

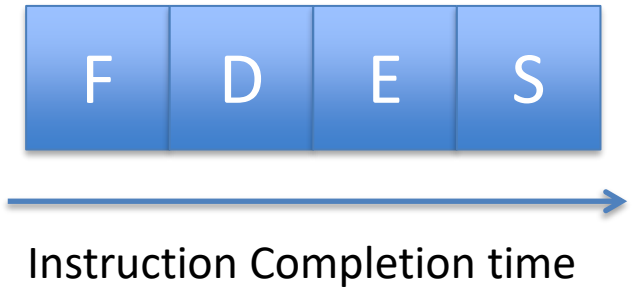
Clock Edges Triggers events

- 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



Clock cycle Time

- Execution stages: fetch, decode, execute, store

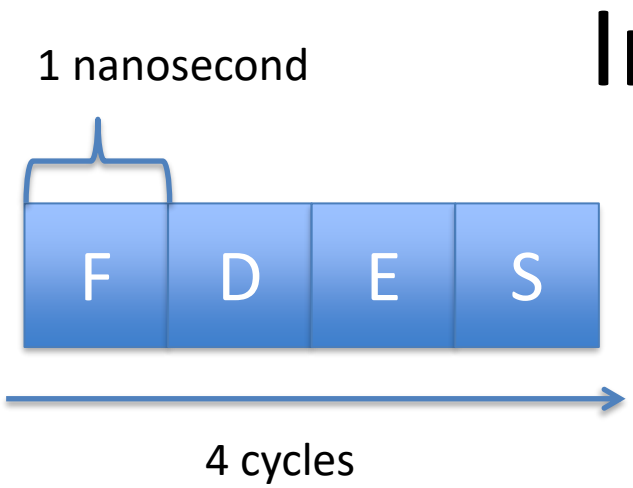


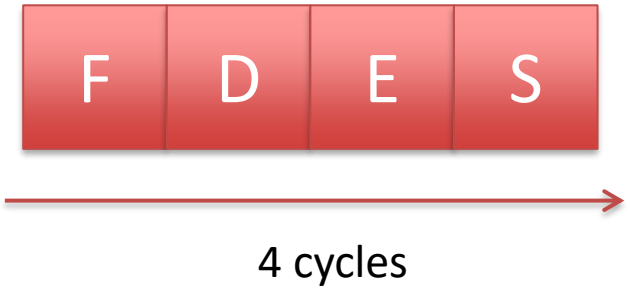
- If one stage per cycle:

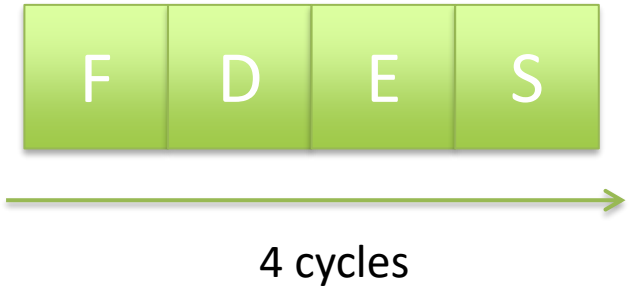
  longest stage to execute determines cycle time

  (ex) if **E**xecute stage is longest, and takes 1 nanosec (10^{-9} sec), then need clock cycle of 1 nanosecond (1 GHz clock)

Instruction Execution







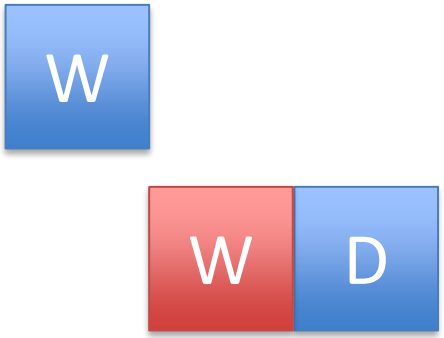
Complete 3 instruction in 12 cycles
(or 3 instructions in 12 nanoseconds)

Q: Can we do better?

Analogy

- Laundry
  - 4 Steps per load:
    Wash first, then **D**ry, then **F**old, then **P**ut Away

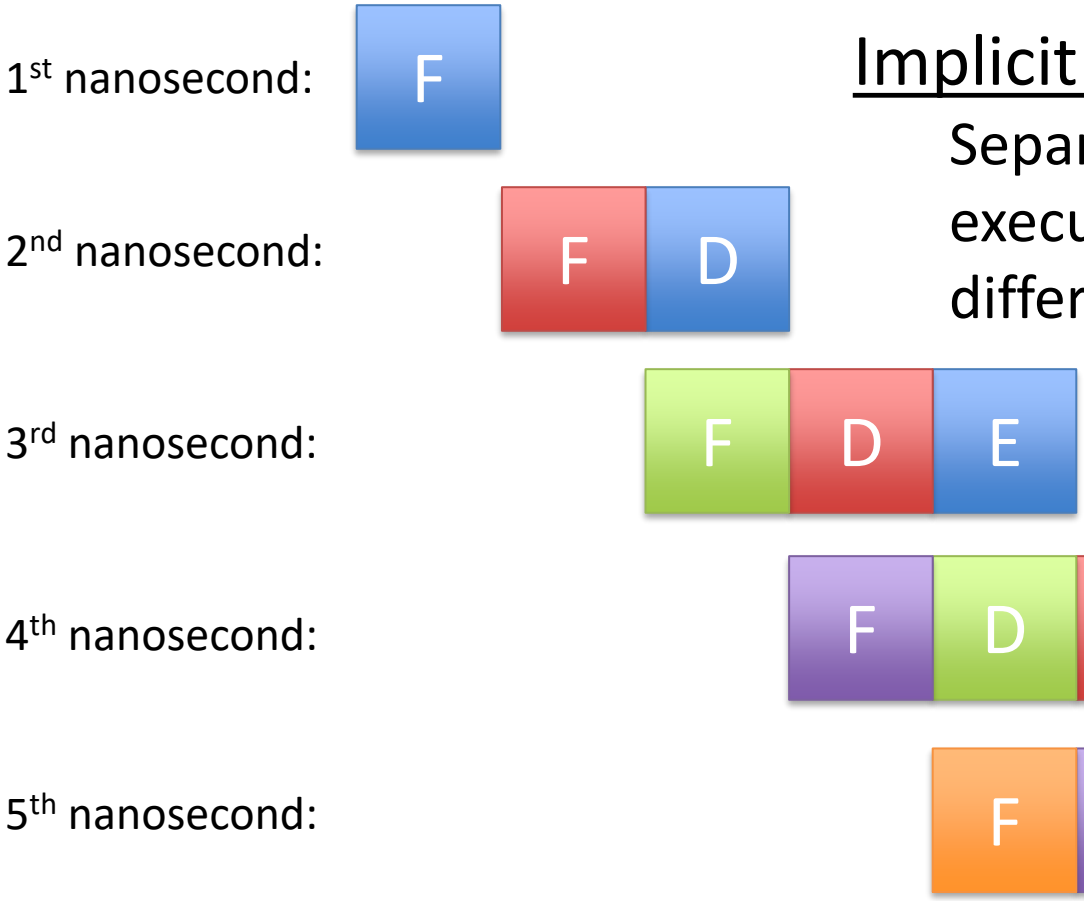
  - If have 6 loads, what do you do when you take first load out of the washer?



Start **W**ashing the second Load while first is in **D**ryer

Pipelining (CPU)

Idea: feed next instruction into previous stage of execution



Implicit Parallel Execution:

Separate instrs simultaneously executing on a single CPU (each in different phase of execution)

Steady state: One instruction finishes every nanosecond!

Summary: 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

Summary: Architecture & Circuits

Modern Computer based on Von Neumann

- Generic Compute Machine & Stored Program model

To run program, need different types of circuits:

- ALU: addition, subtraction, multiplexes, and, or, etc
- storage: R-S latches, registers
- control: (ex) fetch next instr from RAM
- abstractions allow us to build more complex functionality from simpler functionality

Clock-driven system (discrete time)

Pipelining: overlap instruction execution