

CSC 252: Computer Organization

Spring 2026: Lecture 11

Instructor: Yuhao Zhu

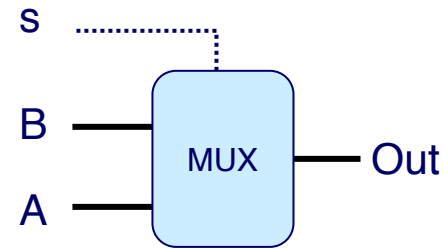
Department of Computer Science
University of Rochester

Today: Circuits Basics

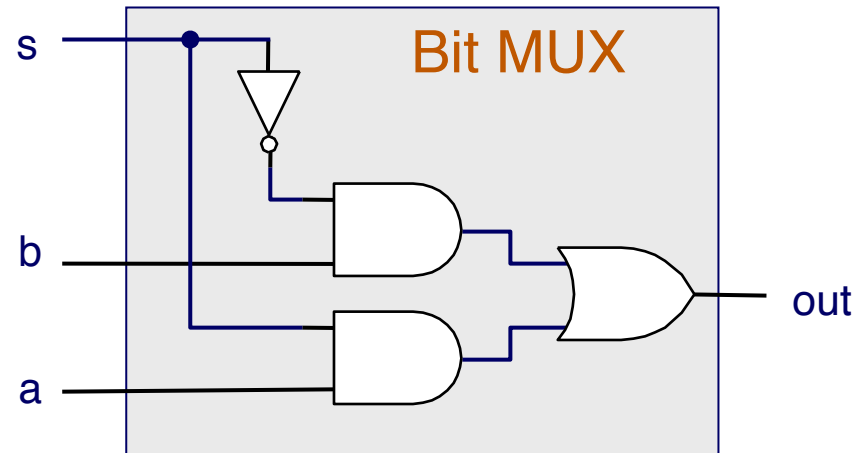
- Basics
- Circuits for computations
- Circuits for storing data

Bit-Level Multiplexor (MUX)

- Control signal s
- Data signals A and B
- Output A when $s=1$, B when $s=0$

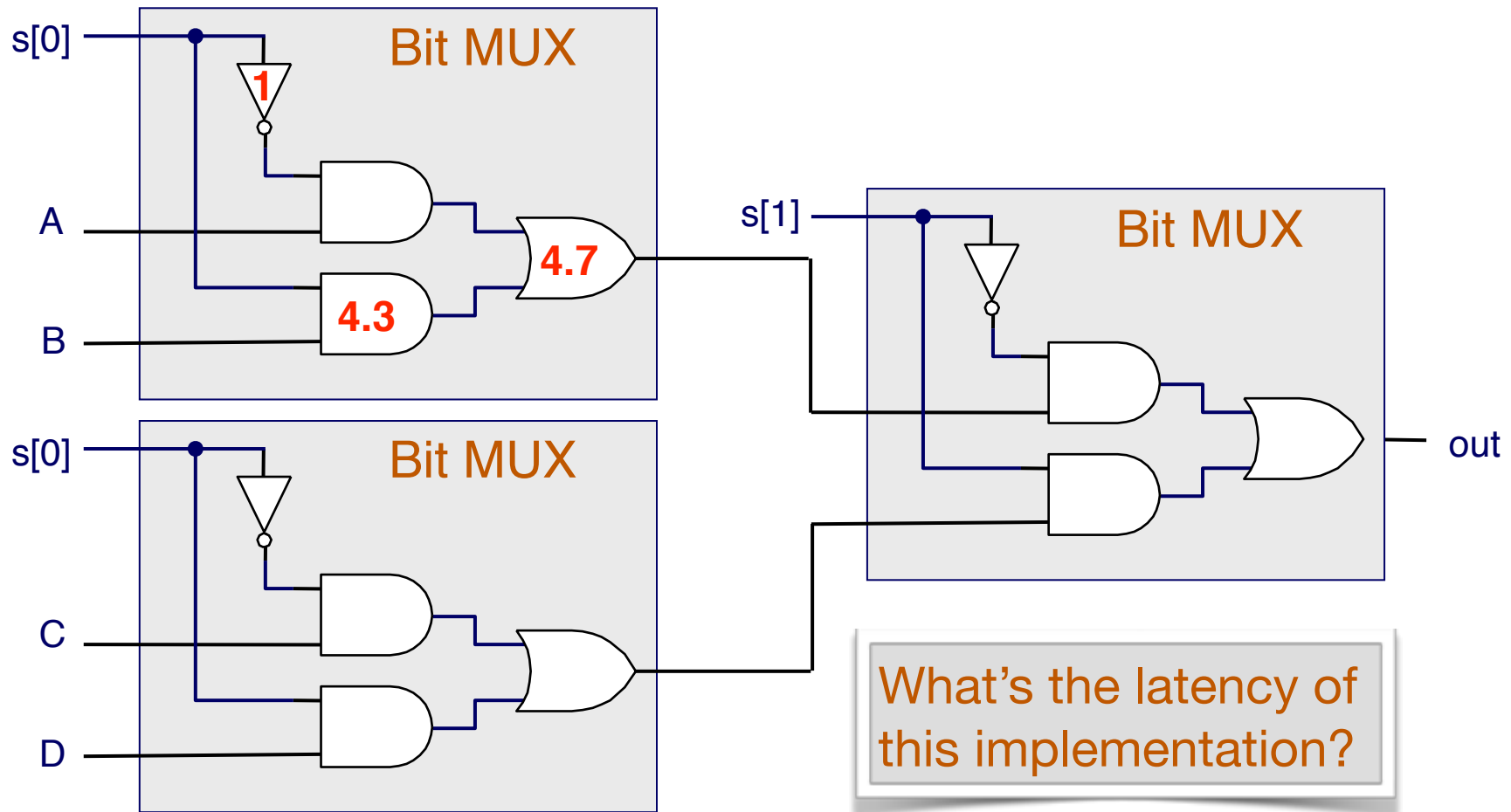


```
bool out = (s&&a) || (!s&&b)
```



4-Input Multiplexor

- Control signal s ; Data signals A, B, C, and D
- Output: A when $s = 00$, B when $s = 01$, C when $s = 10$, D when $s = 11$



Logic Design and VLSI

- The number of inputs of a gate (fan-in) and the number of outputs of a gate (fan-out) will affect the gate delay.
- Think of logic gates as LEGO chips, using which you generate the gate level circuit design for complex functionalities.
- A standard cell library is a collection of well defined and appropriately characterized logic gates (delay, operating voltage, etc.) that can be used to implement a digital design.
- The *logic synthesis tool* will automatically generate the “best” gate-level implementation of a piece of logic.
- Take a Logic Design or Very Large Scale Integrated-Circuit (VLSI) course if you want to know more about circuit design.
 - Logic design uses the gate-level abstractions
 - VLSI tells you how the gates are implemented at transistor-level

Recall: Full (1-bit) Adder

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$\begin{aligned} S = & (\sim A \ \& \ \sim B \ \& \ C_{in}) \\ & | (\sim A \ \& \ B \ \& \ \sim C_{in}) \\ & | (A \ \& \ \sim B \ \& \ \sim C_{in}) \\ & | (A \ \& \ B \ \& \ C_{in}) \end{aligned}$$

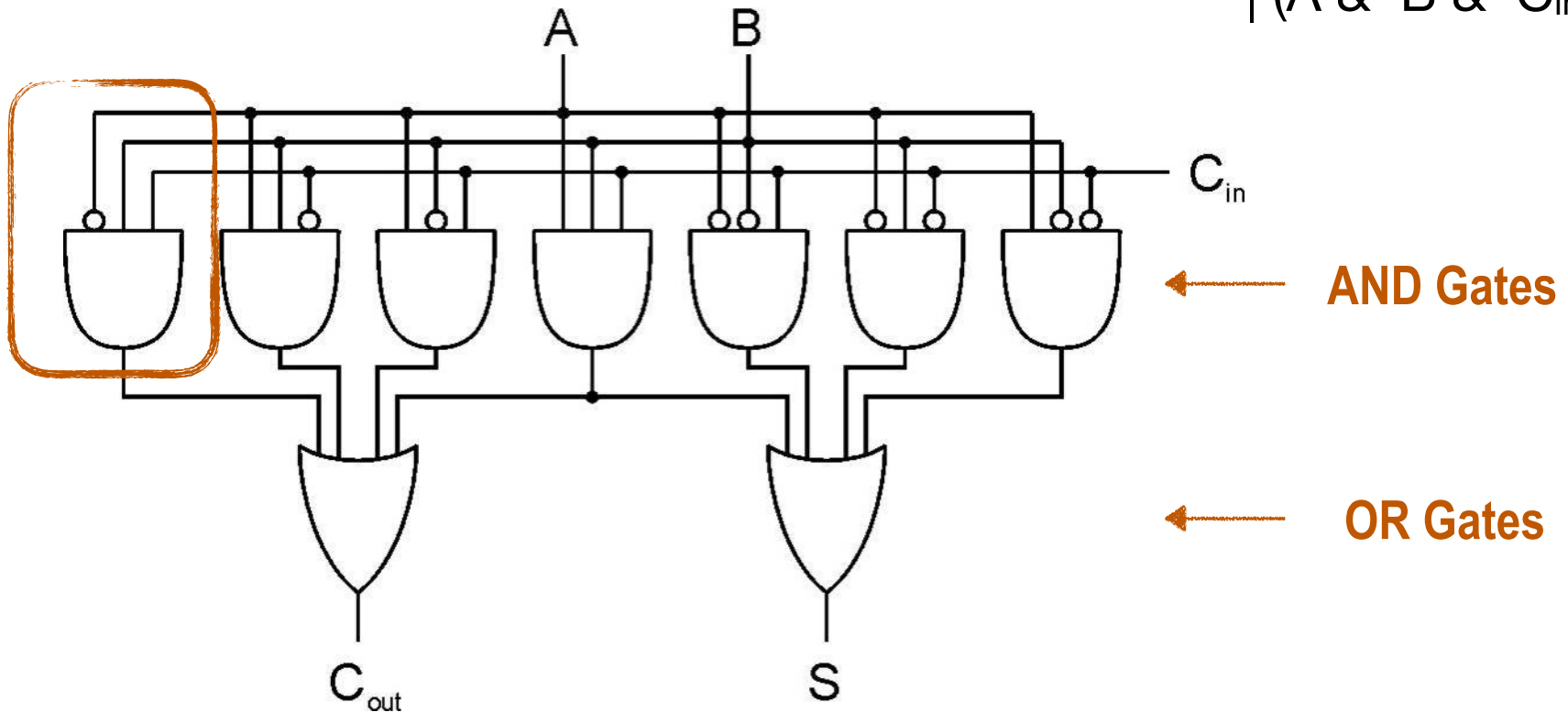
$$\begin{aligned} C_{ou} = & (\sim A \ \& \ B \ \& \ C_{in}) \\ & | (A \ \& \ \sim B \ \& \ C_{in}) \\ & | (A \ \& \ B \ \& \ \sim C_{in}) \\ & | (A \ \& \ B \ \& \ C_{in}) \end{aligned}$$

A	B	C _{in}	S	C _{ou}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Recall: 1-bit Full Adder

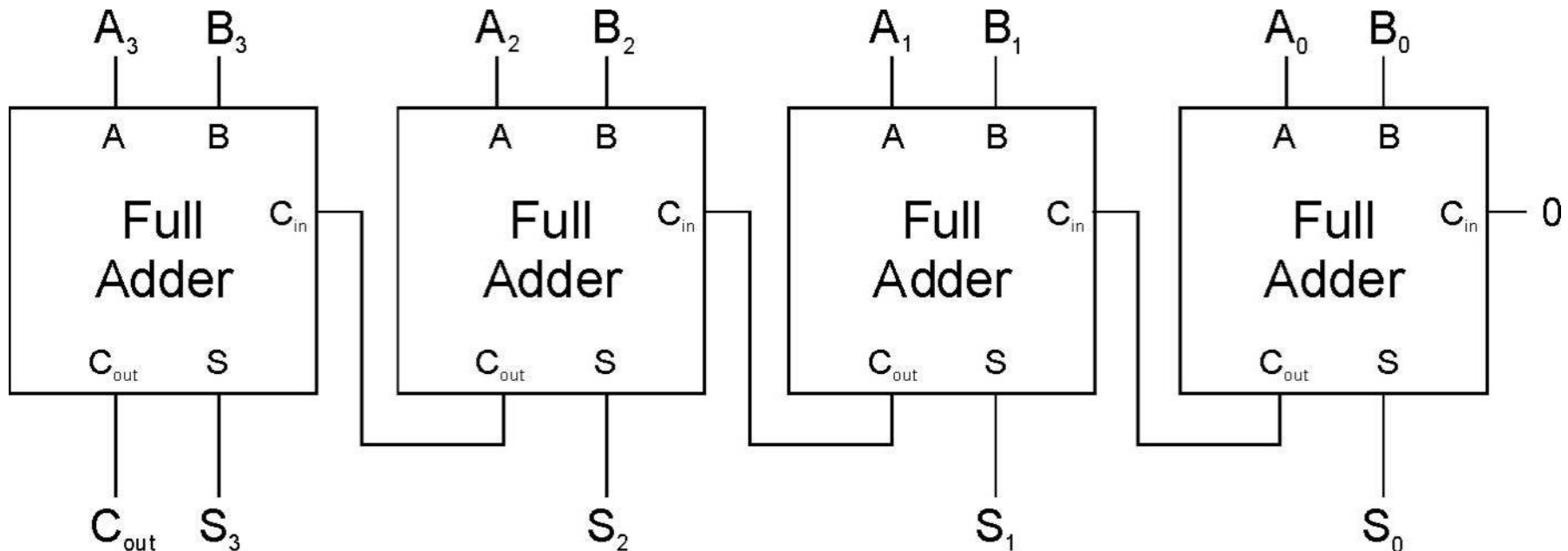
$$C_{ou} = (\sim A \& B \& C_{in}) \mid (A \& \sim B \& C_{in}) \mid (A \& B \& \sim C_{in}) \mid (A \& B \& C_{in})$$

Add two bits and carry-in,
produce one-bit sum and carry-out.

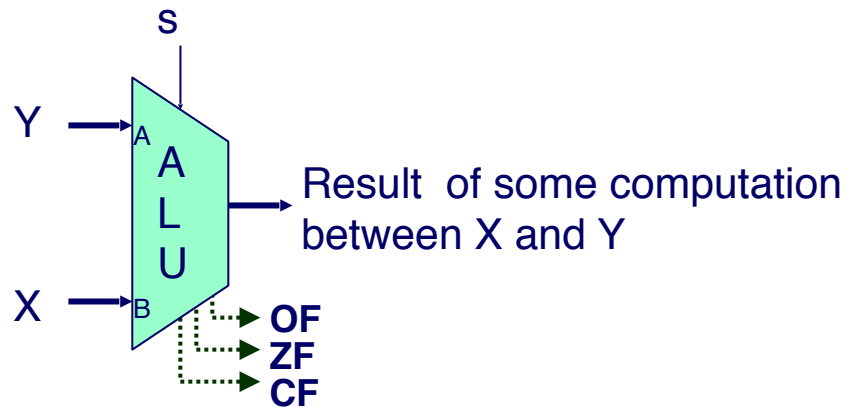


Recall: Four-bit Adder

- Ripple-carry Adder
 - Simple, but performance linear to bit width
- Carry look-ahead adder (CLA)
 - Generate all carriers simultaneously



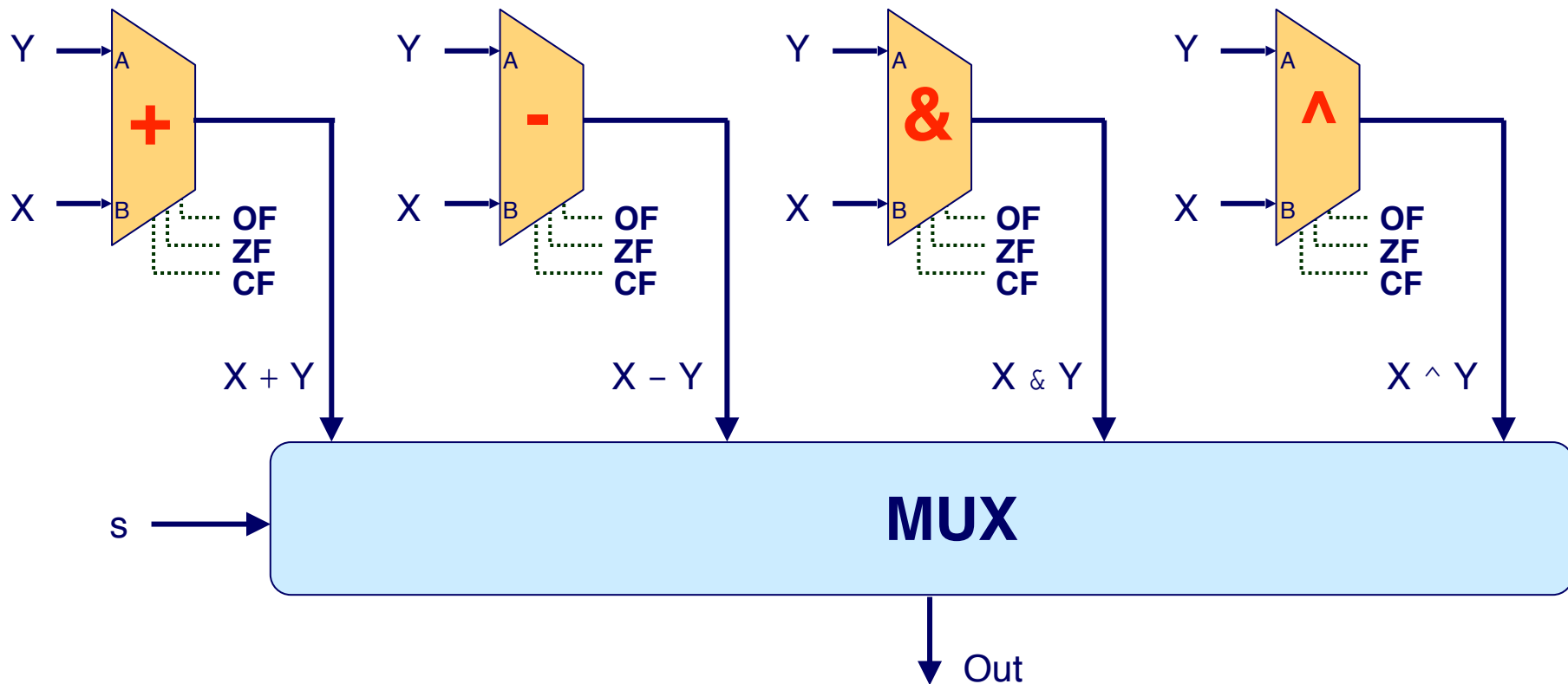
Arithmetic Logic Unit



- An ALU performs multiple kinds of computations.
- The actual computation depends on the selection signal s .
- Also sets the condition codes (status flags)
- For instance:
 - $X + Y$ when $s == 00$
 - $X - Y$ when $s == 01$
 - $X \& Y$ when $s == 10$
 - $X \wedge Y$ when $s == 11$
- How can this ALU be implemented?

Arithmetic Logic Unit

- Implement 4 different circuits, one for each operation.
- Then use a MUX to select the results



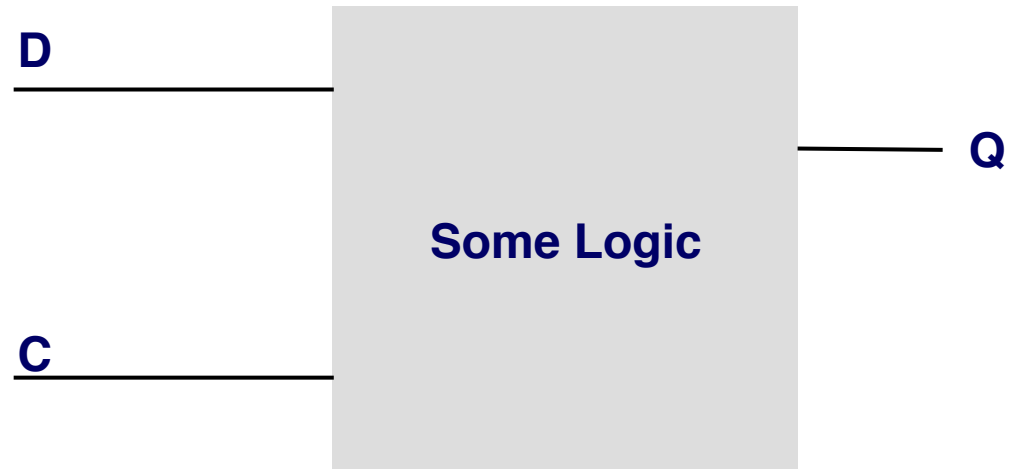
Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

The Need for Storing Bits

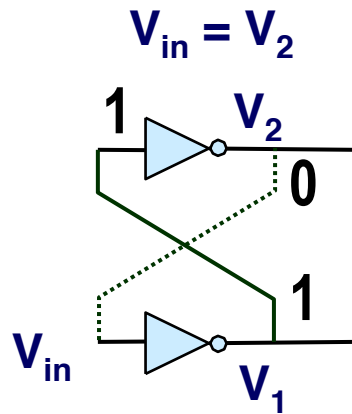
- Assembly programs set architecture (processor) states.
 - Register File
 - Status Flags
 - Memory
 - Program Counter
- Every state is essentially some bits that are stored/loaded.
- Think of the program execution as an FSM.
- The hardware must provide mechanisms to load and store bits.
- There are many different ways to store bits. They have trade-offs.

Build a 1-Bit Storage

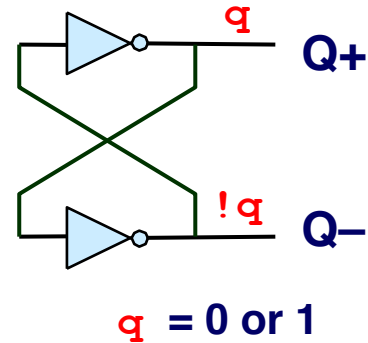


- What we would like:
 - D is the data we want to store (0 or 1)
 - C is the control signal
 - When C is 1, Q becomes D (i.e., storing the data)
 - When C is 0, Q doesn't change with D (data stored)

Bitstable Element



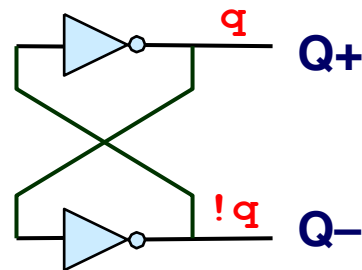
Bistable Element



$Q+$ *continuously* outputs q .

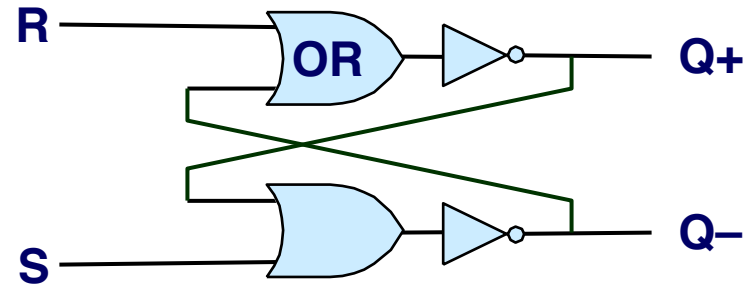
Storing and Accessing 1 Bit

Bistable Element

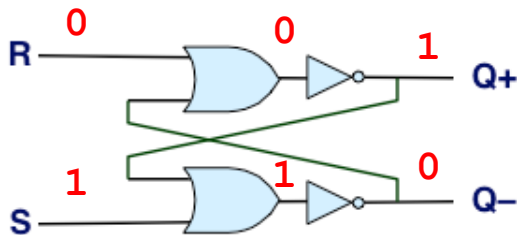


$q = 0 \text{ or } 1$

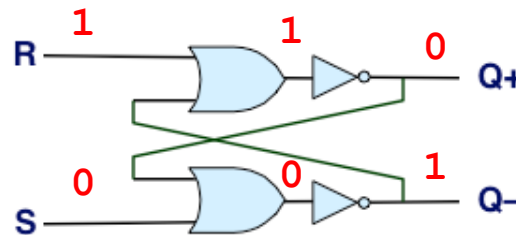
R-S Latch



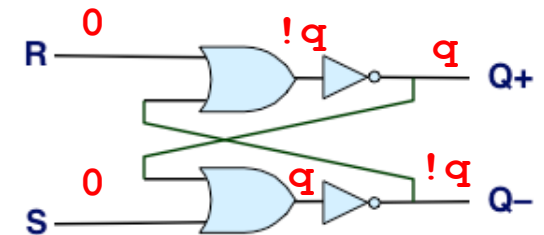
Setting $Q+$ to 1



Setting $Q+$ to 0



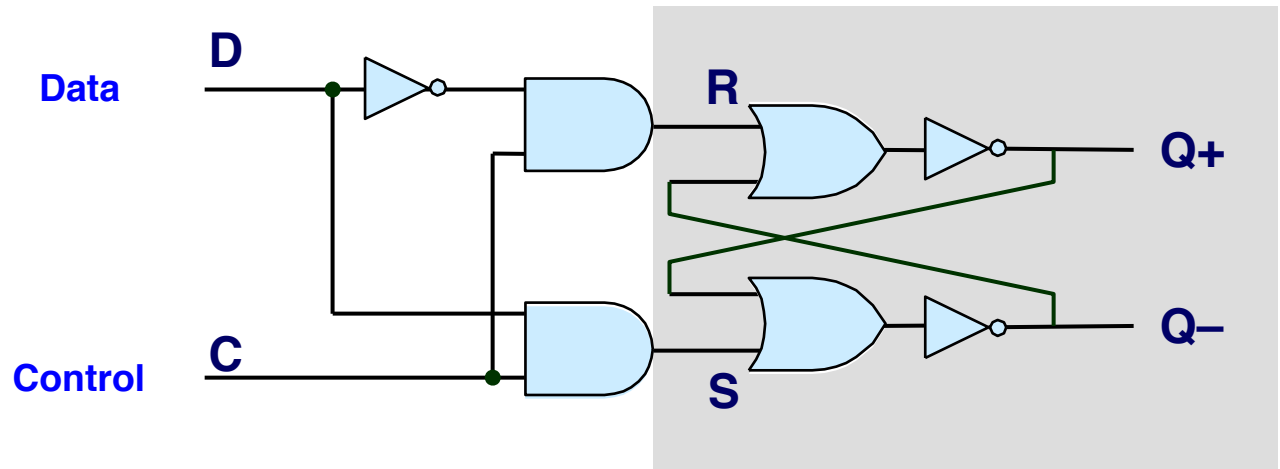
$Q+$ value unchanged
i.e., stored!



If R and S are different, $Q+$ is the same as S

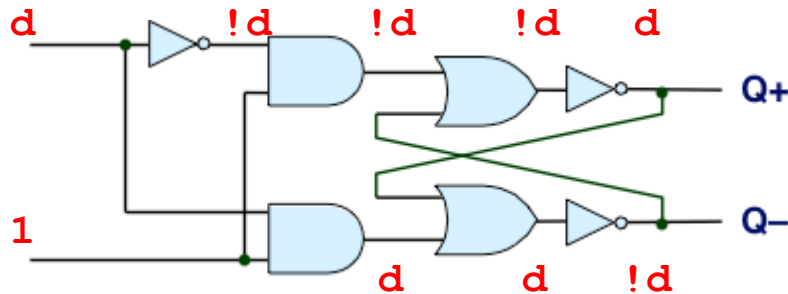
Building on top of R-S Latch

D Latch



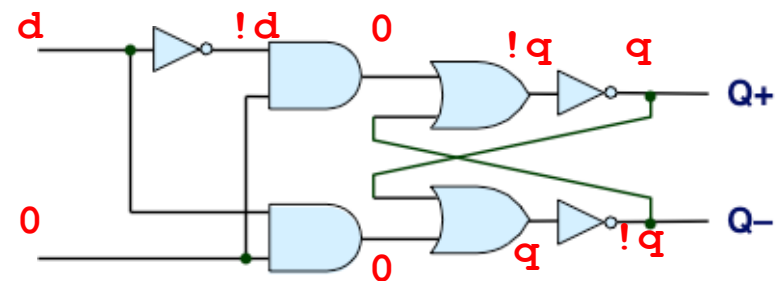
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

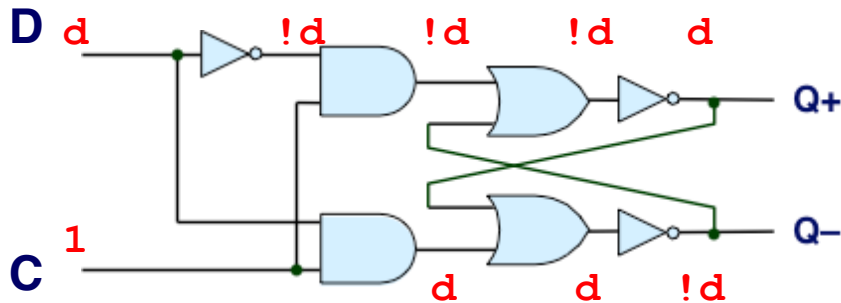
Holding Data



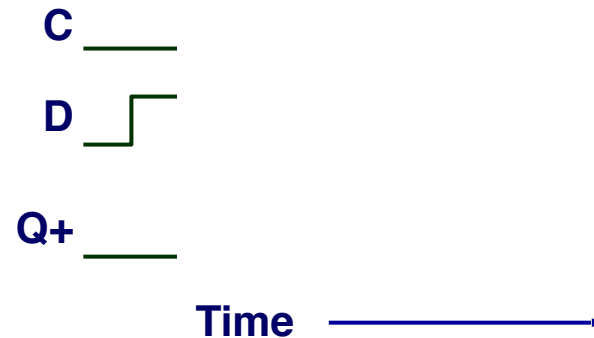
Q+ doesn't change with d

D-Latch is “Transparent”

Latching

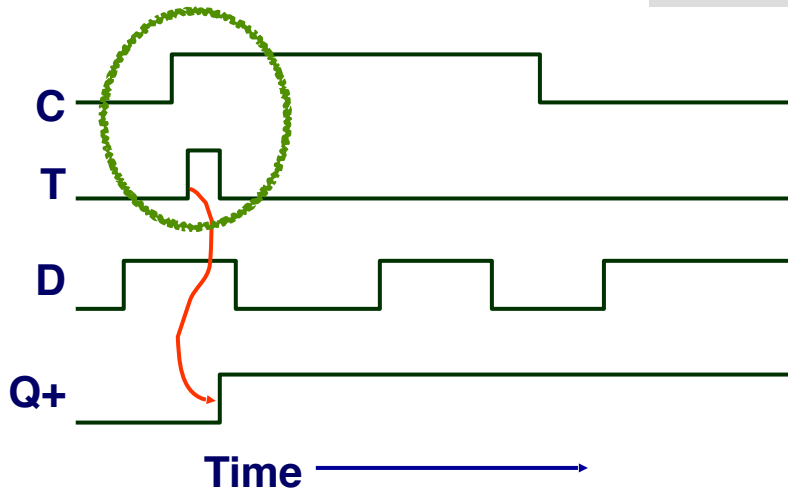
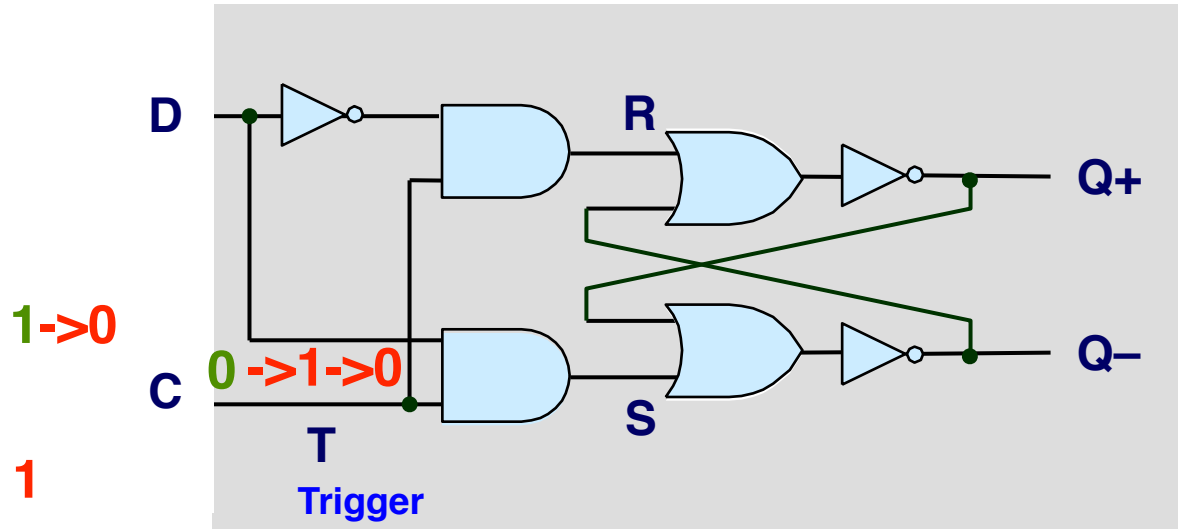


Changing D



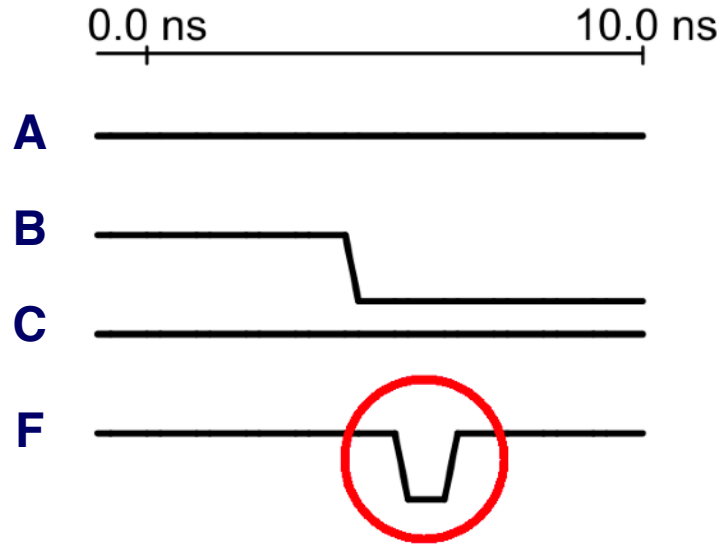
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+**. So hold **C** for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **C** goes to 0
- D-latch is *transparent* when **C** is 1
- D-latch is “*level-triggered*” b/c **Q** changes as the *voltage level* of **C** rises.

Edge-Triggered Latch (Flip-Flop)

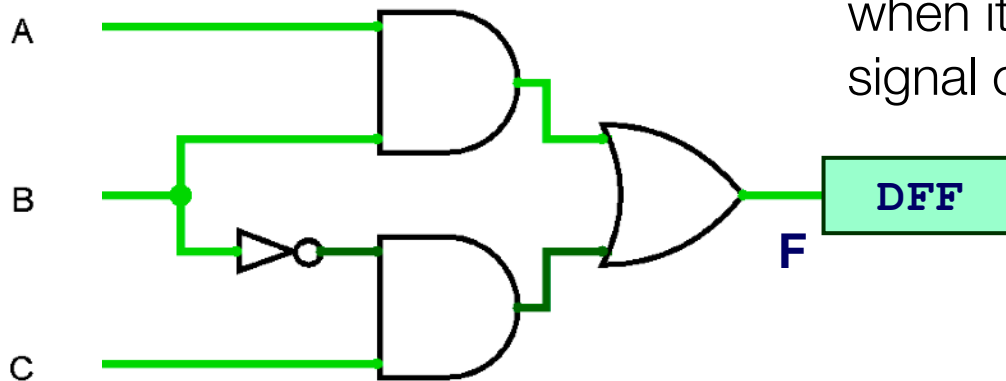


- Flip-flop: Only latches data for a brief period
- Value latched depends on data as C **rises** (i.e., 0→1); usually called at the **rising edge** of C
- Output remains stable at all other times

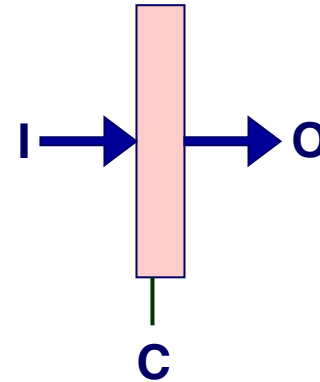
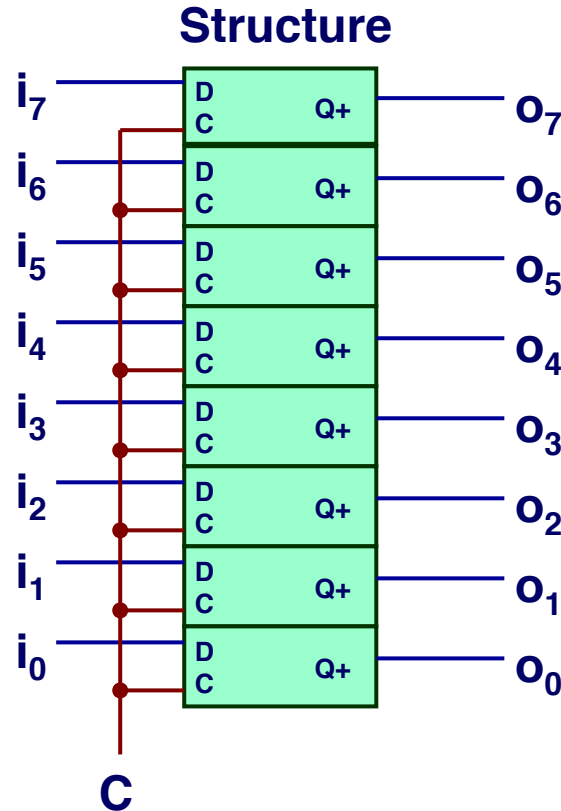
Why Use a Flip-Flop?



- Because the data we want to store might be temporarily changing before it settles down (due to glitch). We want to capture only the final value.
- If we had a transparent D latch, the latched value would change with F, i.e., temporal glitches will be temporarily stored as well.
- With a flip flop, we can store data only when its value settles: raise the control signal of the flop when F settles.

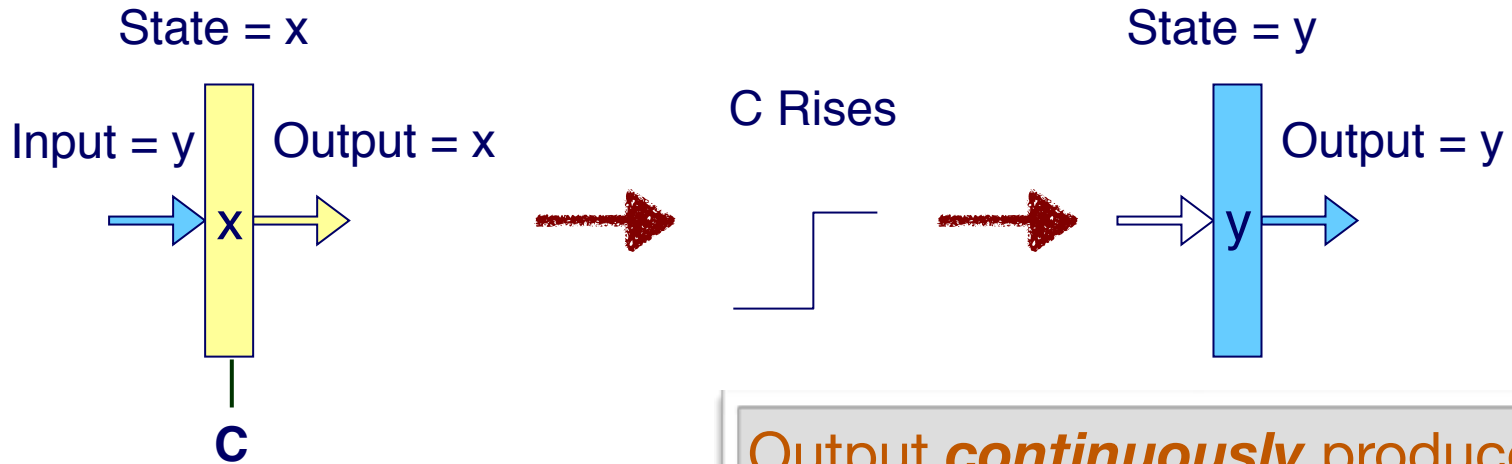


Registers



- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

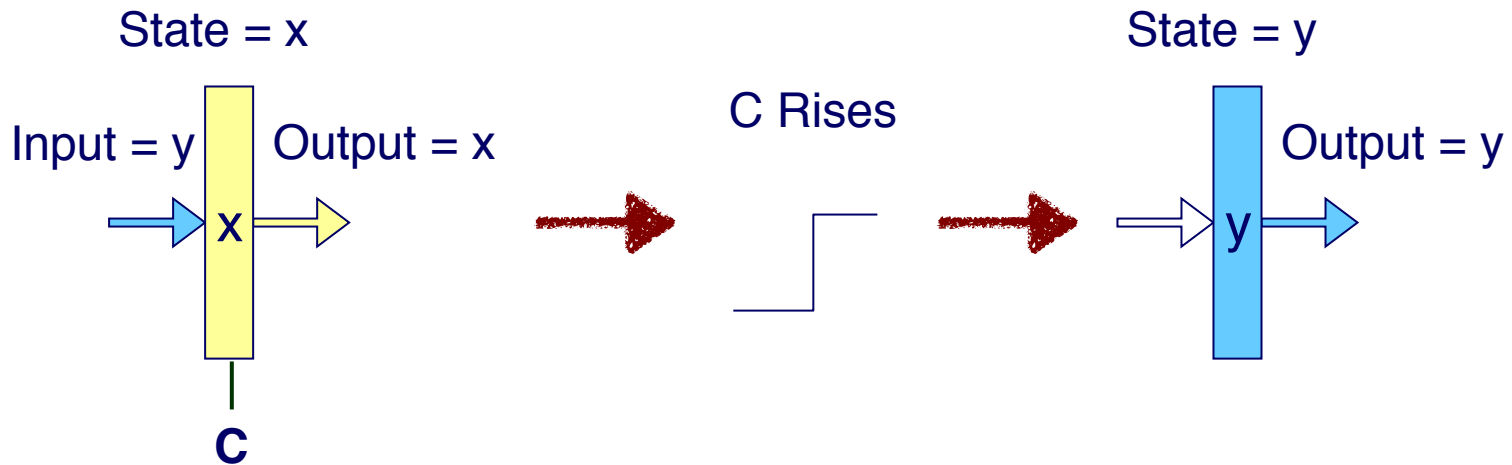
Register Operation



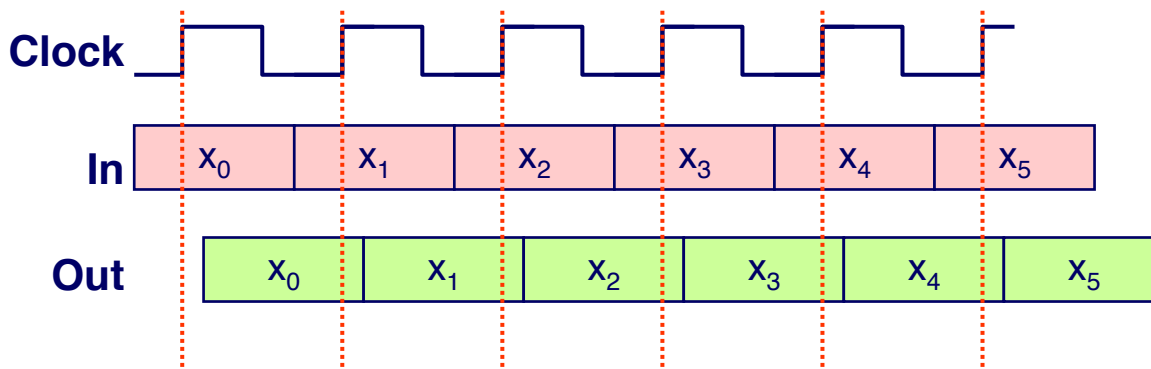
Output ***continuously*** produces y after the rising edge unless you cut off power.

- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you'd better compute the input before the C signal rises if you want to store the input data to the register

Clock Signal

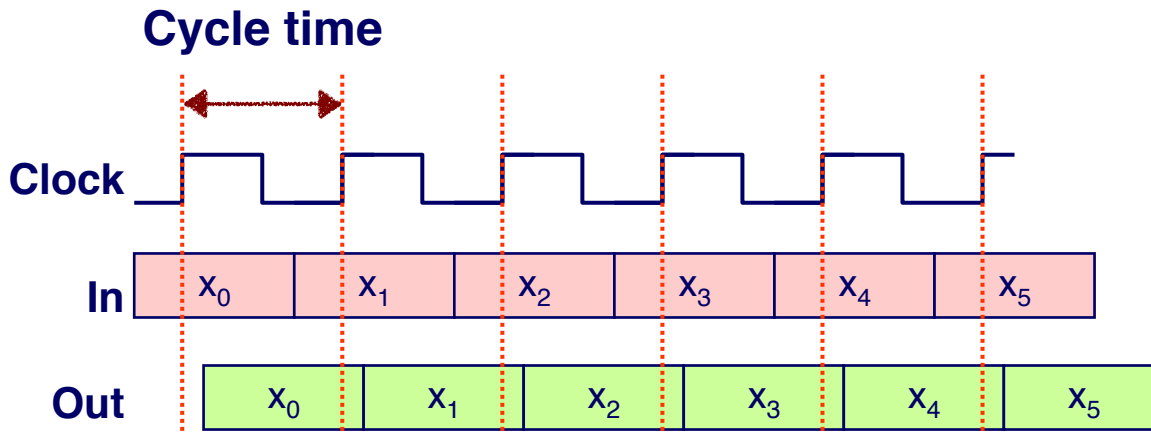


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



Clock Signal

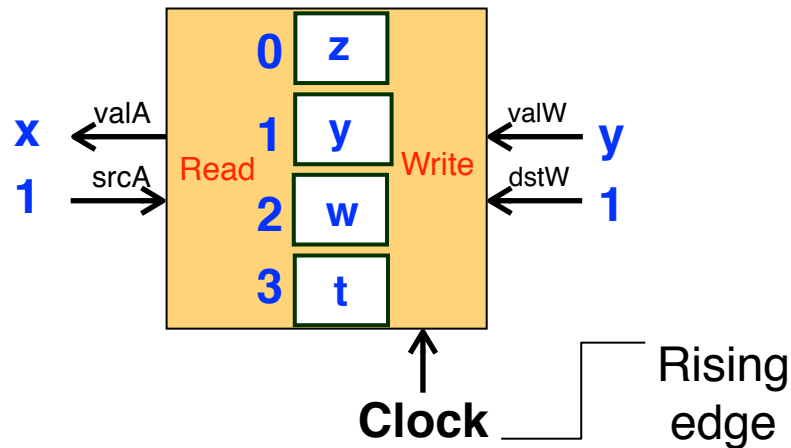
- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz
 - The cycle time is $1/10^9 = 1 \text{ ns}$



Register File

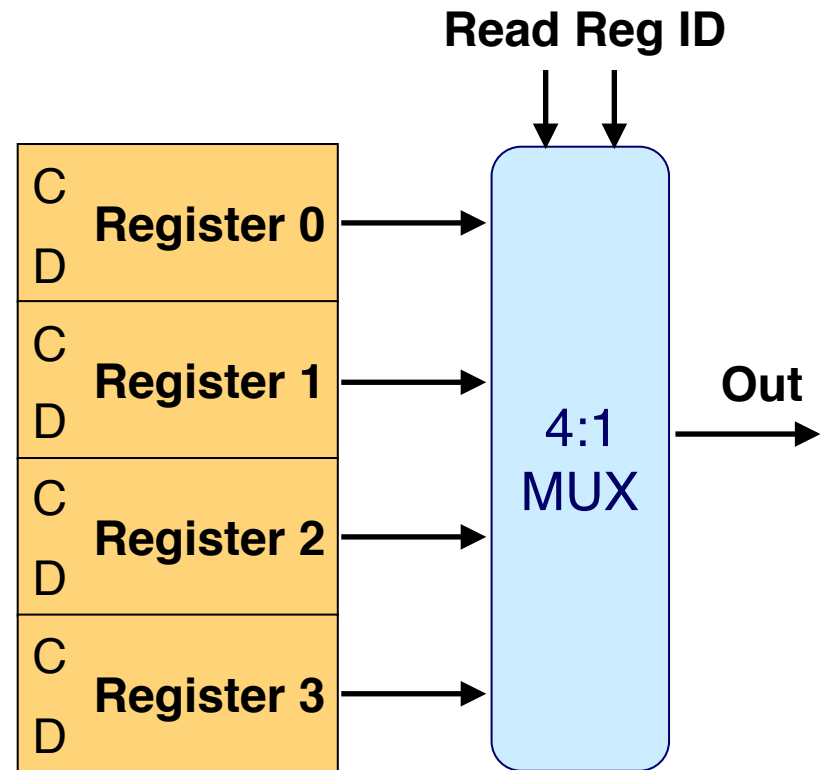
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value
- How do we build a register file out of individual registers??

Register File



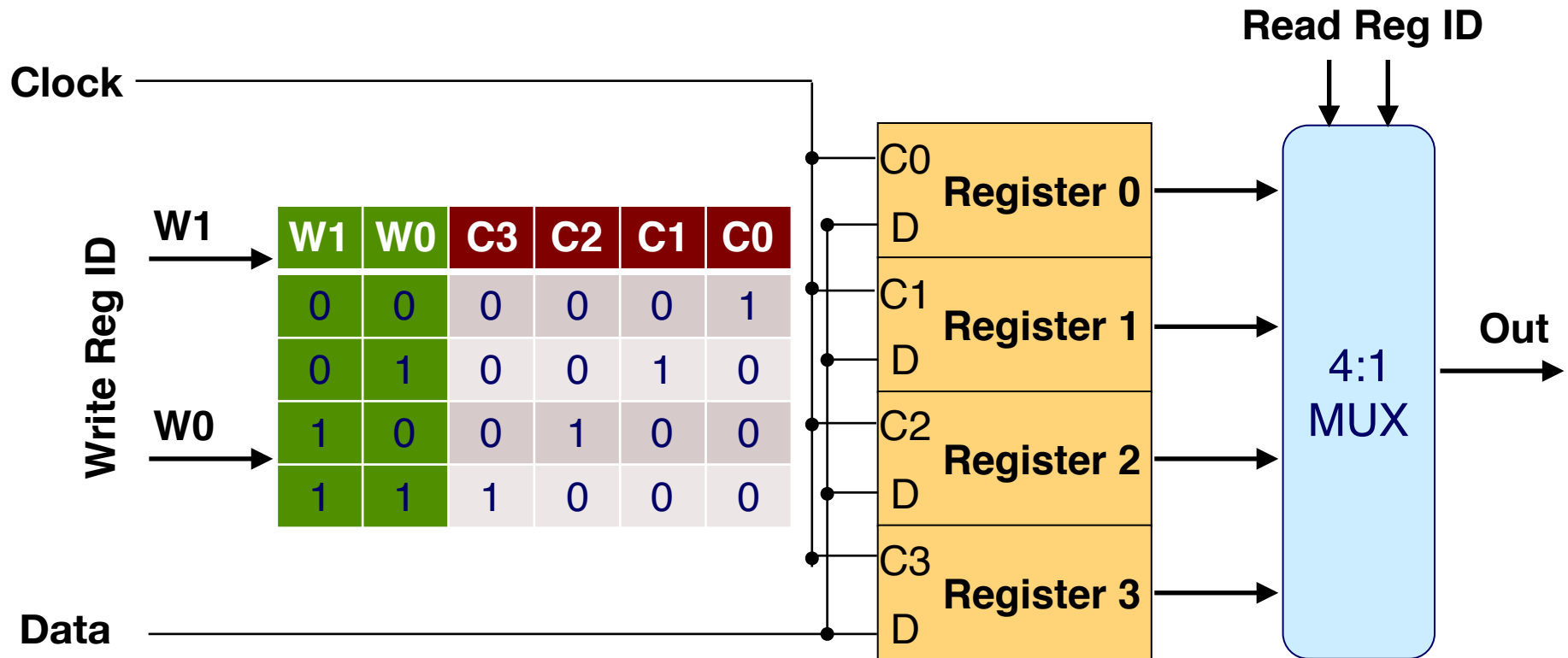
Register File Read

- Continuously read a register independent of the clock signal



Register File Write

- Only write to a specific register when the clock rises. How??



Decoder

W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

W0 _

W1 _

_C0

_C1

_C2

_C3

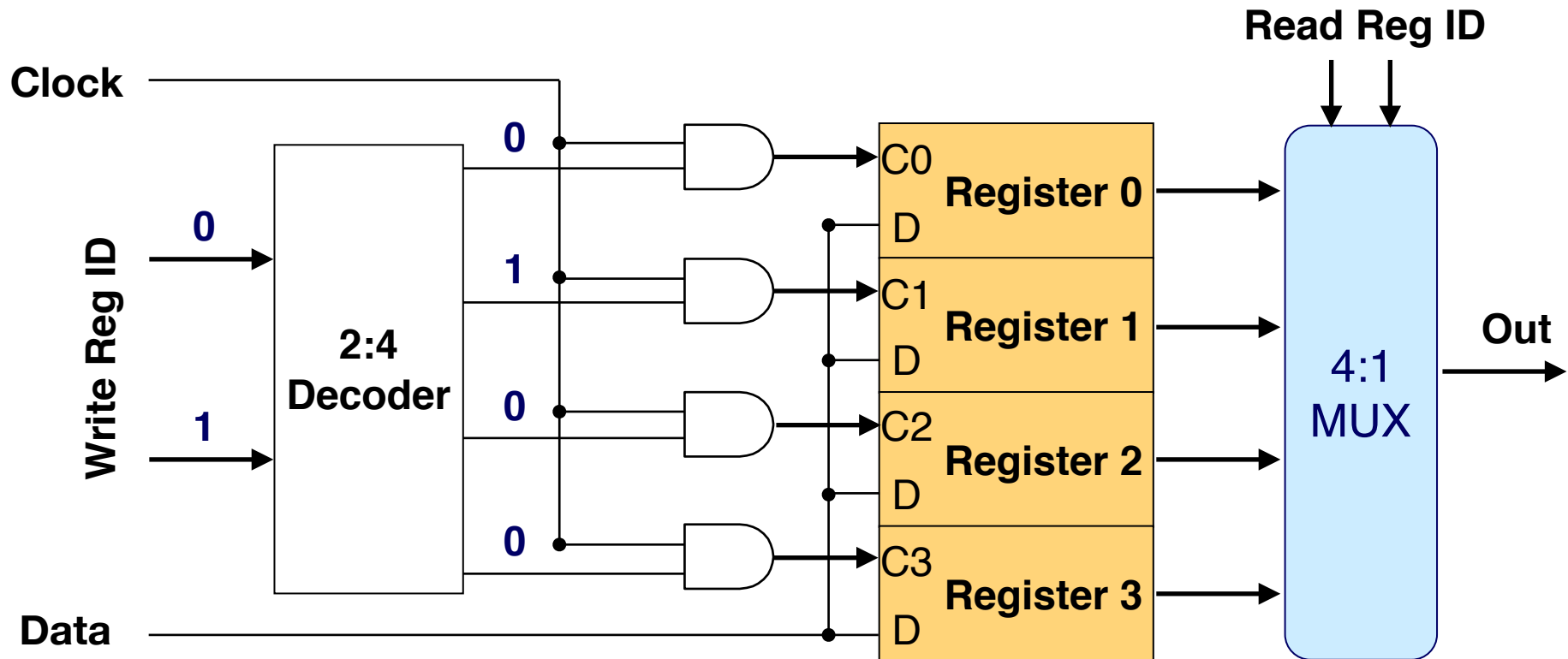
$$C0 = !W1 \ \& \ !W0$$

$$C1 = !W1 \ \& \ W0$$

$$C2 = W1 \ \& \ !W0$$

$$C3 = W1 \ \& \ W0$$

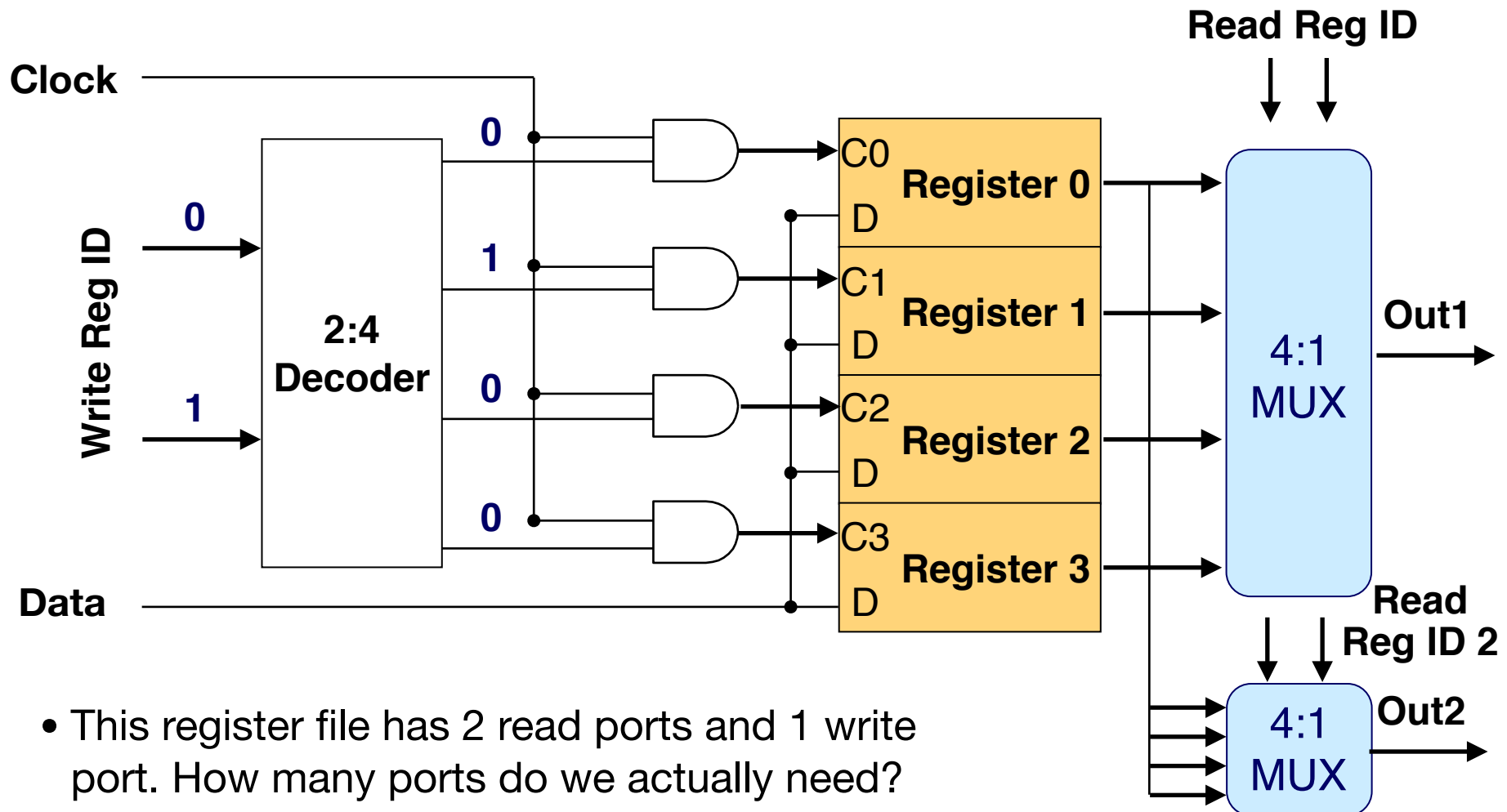
Register File Write



- This implementation can read 1 register and write 1 register at the same time: 1 read port and 1 write port

Multi-Port Register File

- What if we want to read multiple registers at the same time?



- This register file has 2 read ports and 1 write port. How many ports do we actually need?

Multi-Port Register File

- Is this correct? What if we don't want to write anything?

