

CSC 252: Computer Organization Fall 2021: Lecture 11

Processor Architecture:
Circuits

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

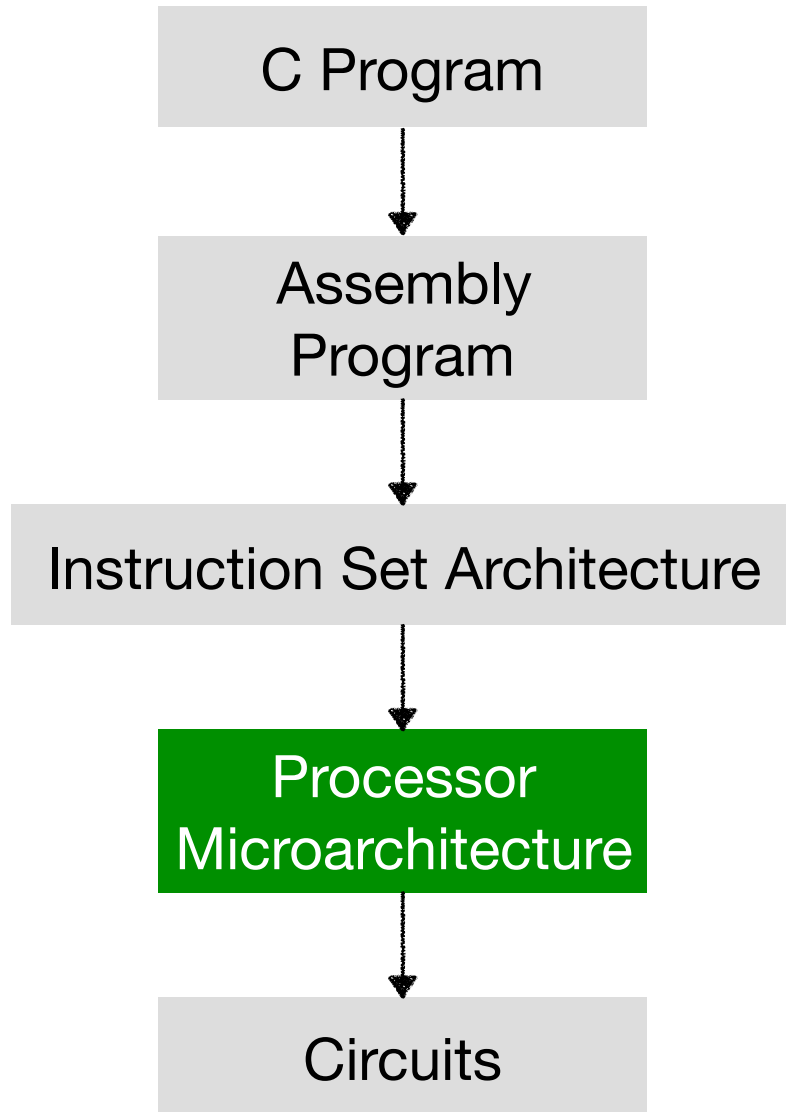
Announcements

Assignment 3 out!

Midterm will be 3 weeks from today (Monday 10/25)

- Can include things discussed on Monday 10/18
- Format based on previous exams, new questions
- Open book, unlimited notes, no electronics
- Recommend making your own condensed note sheet (helps with learning, and faster to find things)

So far in 252...

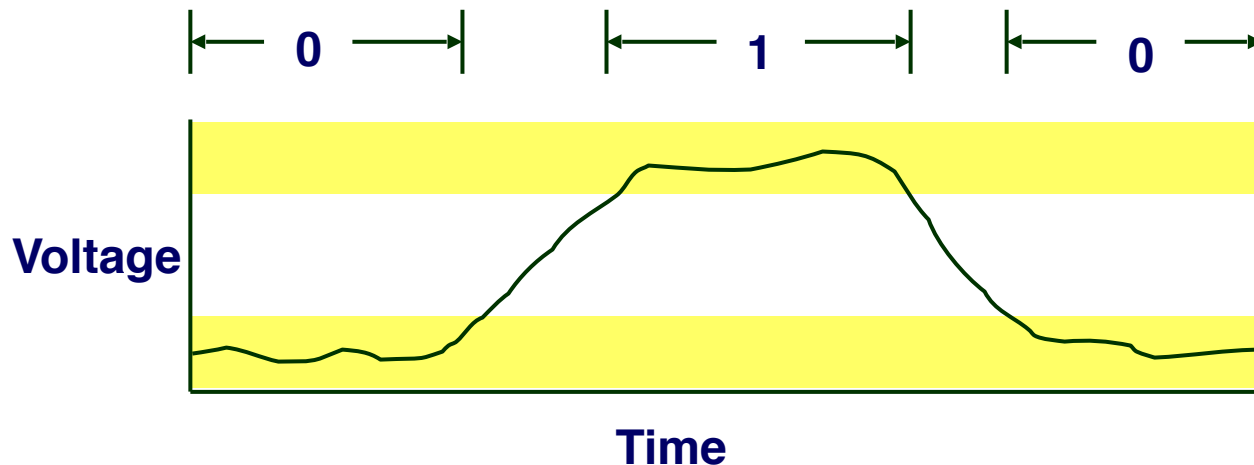


Today: Circuits Basics

- Basics
- Circuits for computations
- Circuits for storing data

Overview of Circuit-Level Design

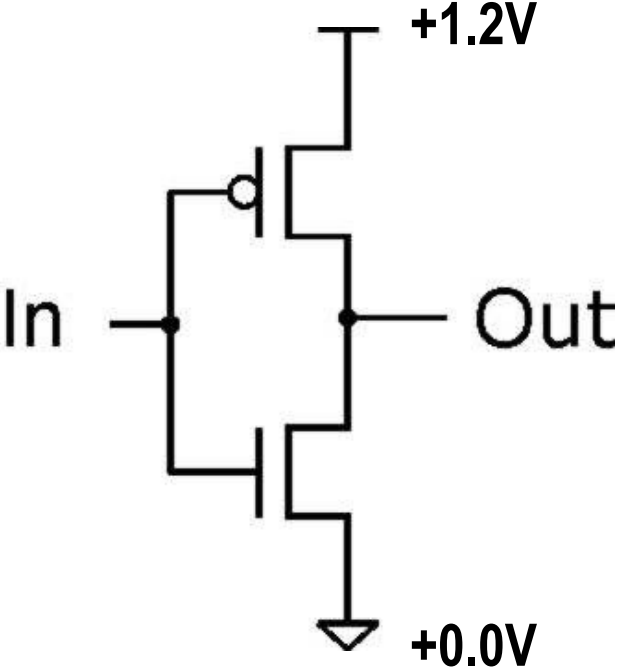
- Fundamental Hardware Requirements
 - Communication: How to get values from one place to another. Mainly three electrical **wires**.
 - Computation: **transistors**. Combinational logic.
 - Storage: **transistors**. Sequential logic.
- Circuit design is often abstracted as **logic design**



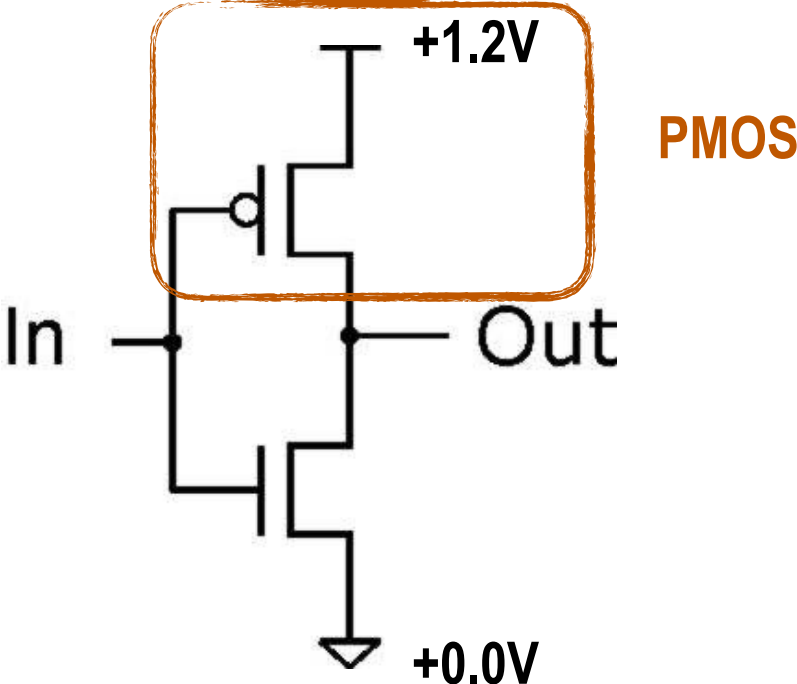
Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

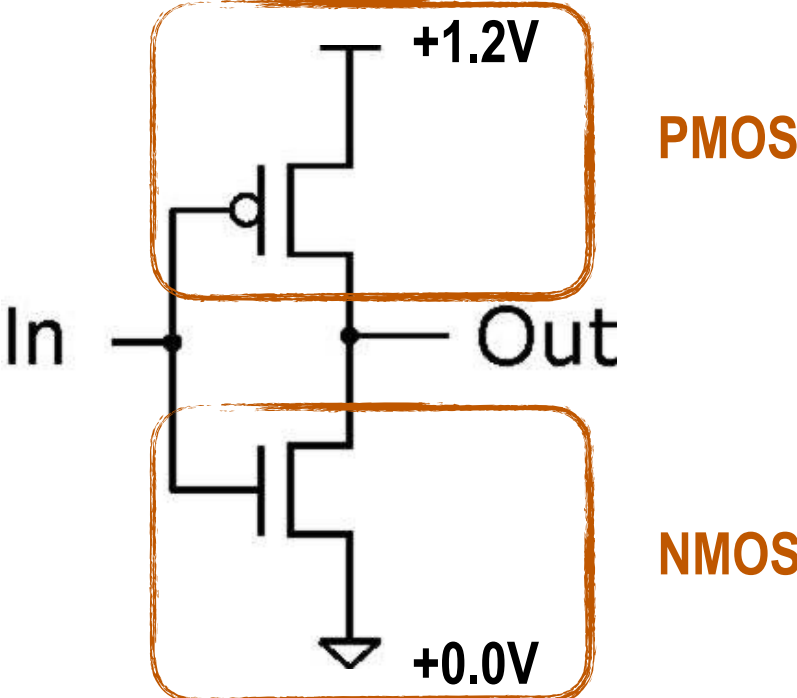
Inverter (NOT Gate)



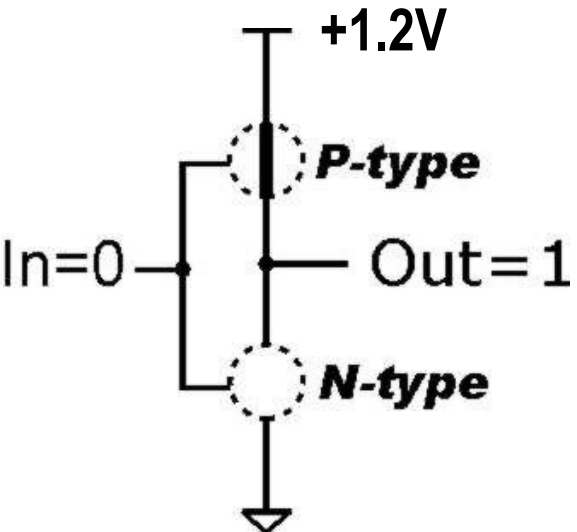
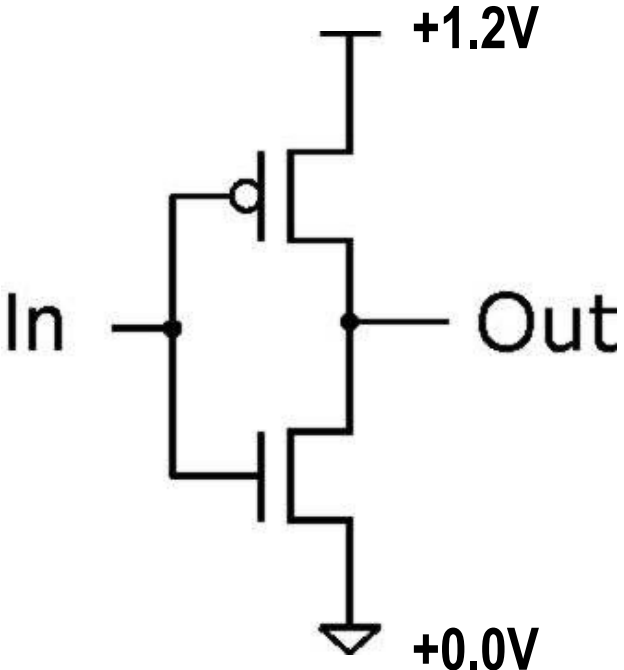
Inverter (NOT Gate)



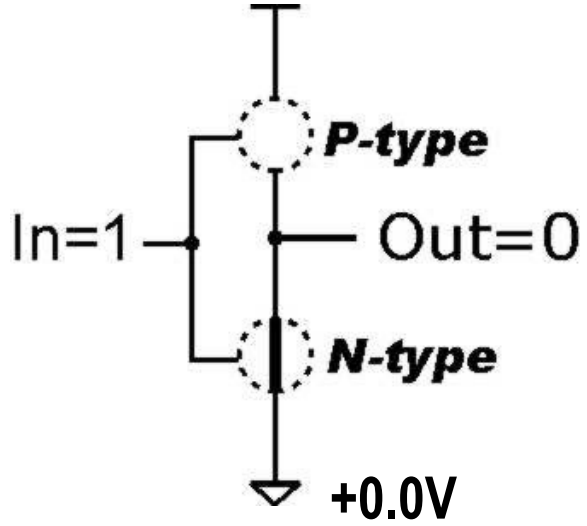
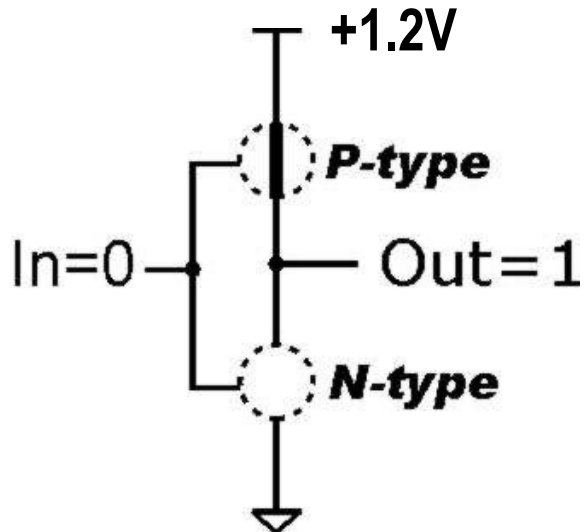
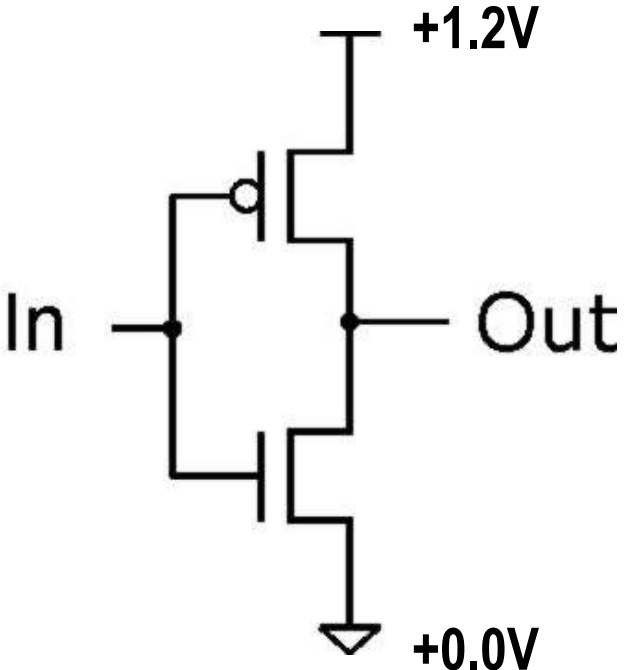
Inverter (NOT Gate)



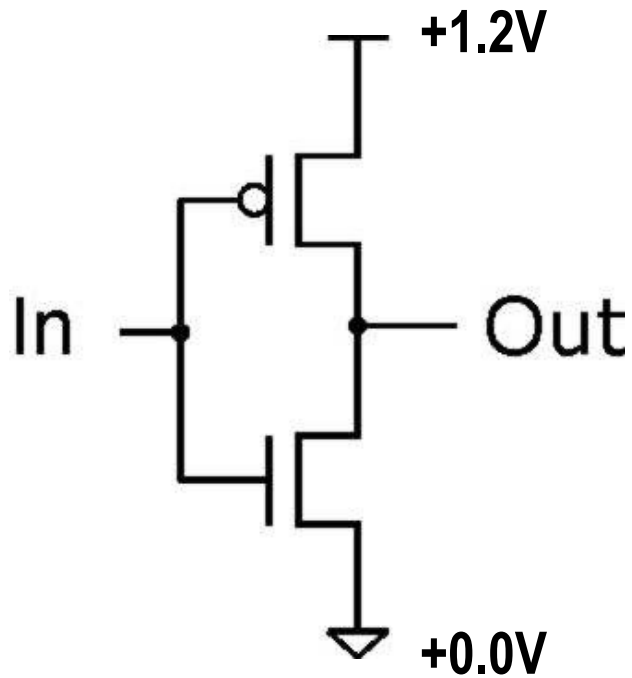
Inverter (NOT Gate)



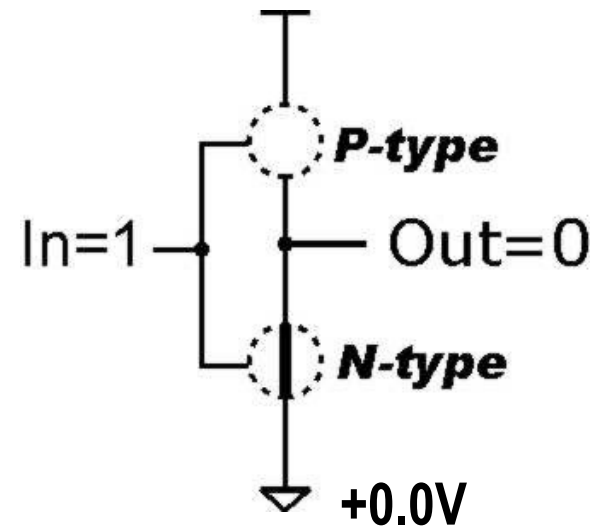
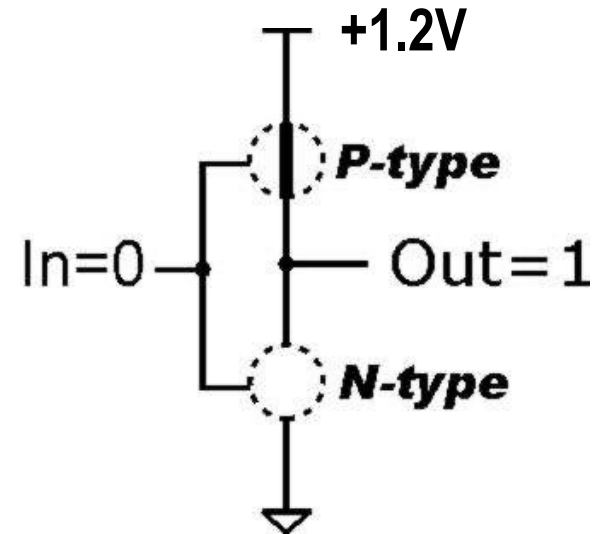
Inverter (NOT Gate)



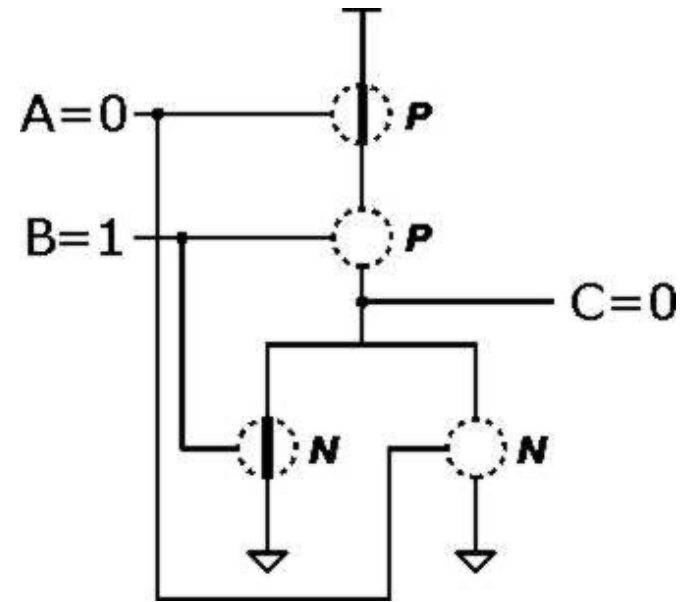
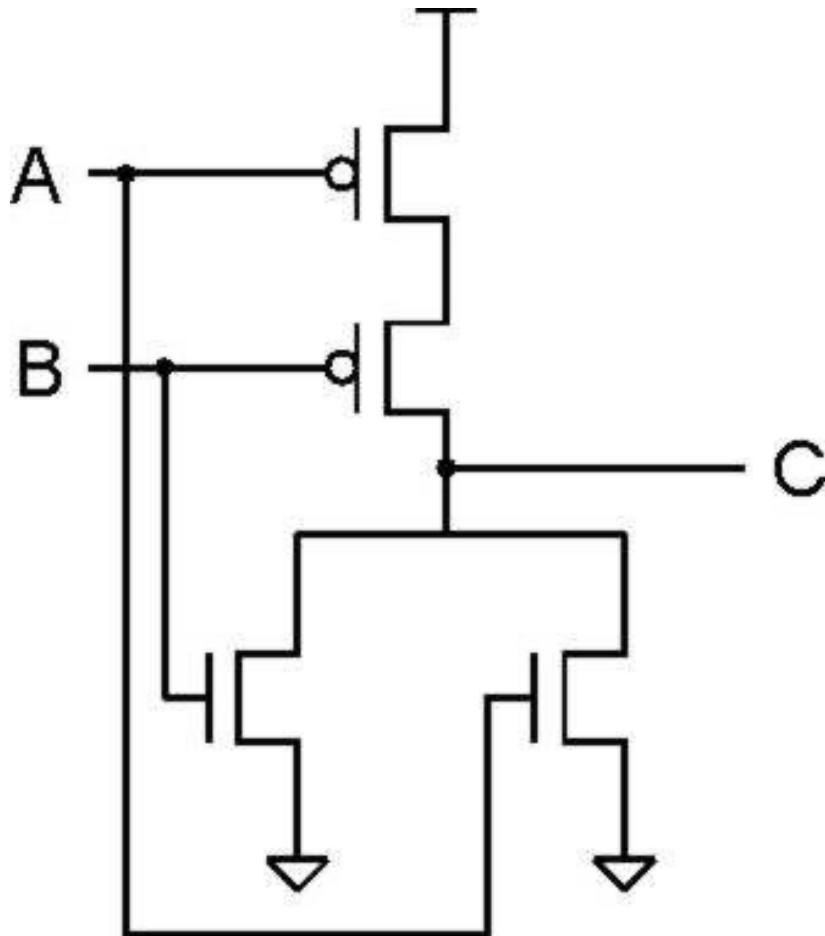
Inverter (NOT Gate)



In	Out
0	1
1	0



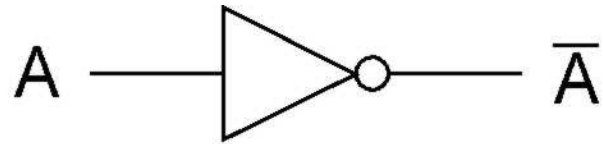
NOR Gate (NOT + OR)



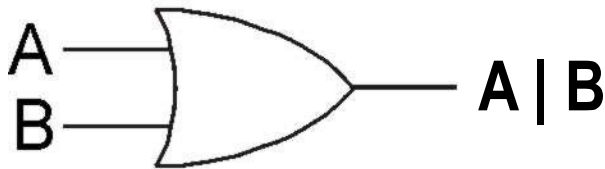
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

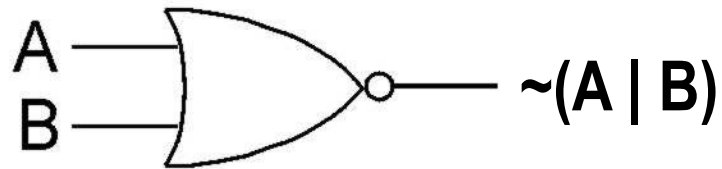
Basic Logic Gates



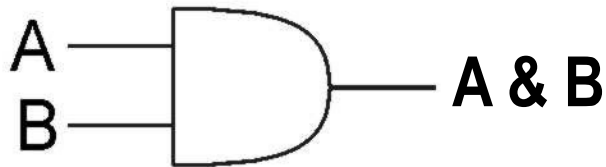
NOT



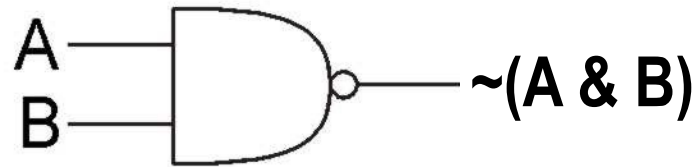
OR



NOR

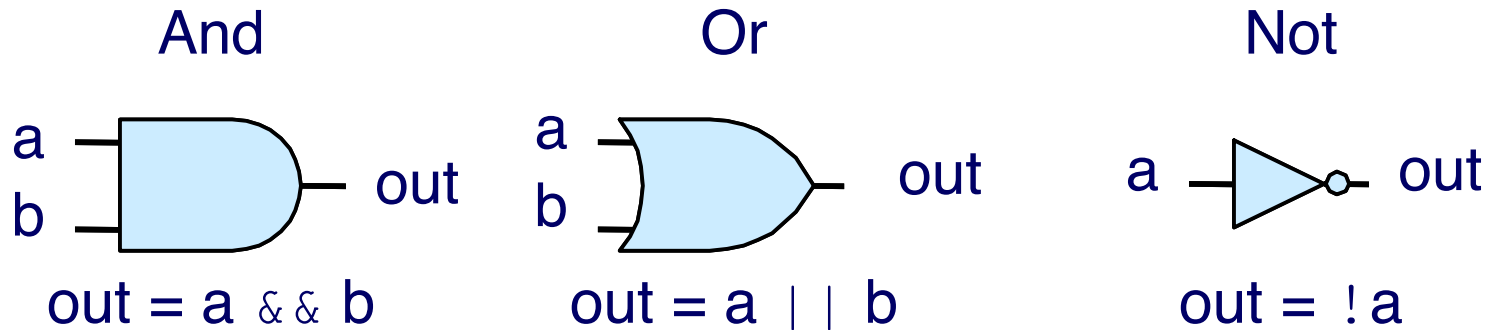


AND

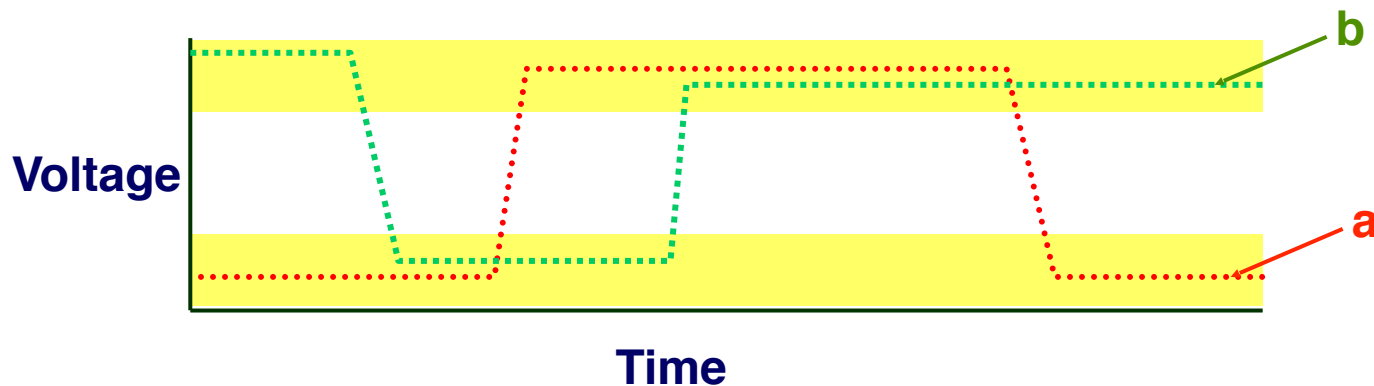


NAND

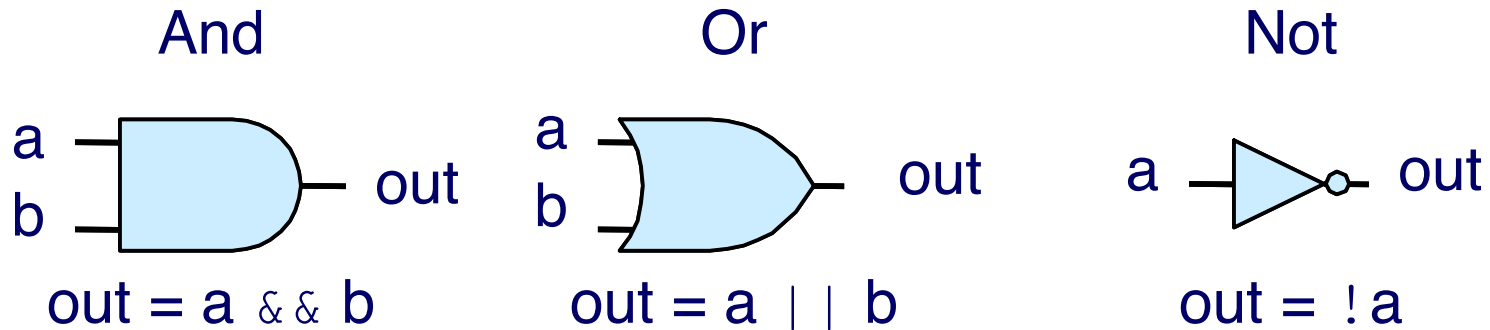
Computing with Logic Gates



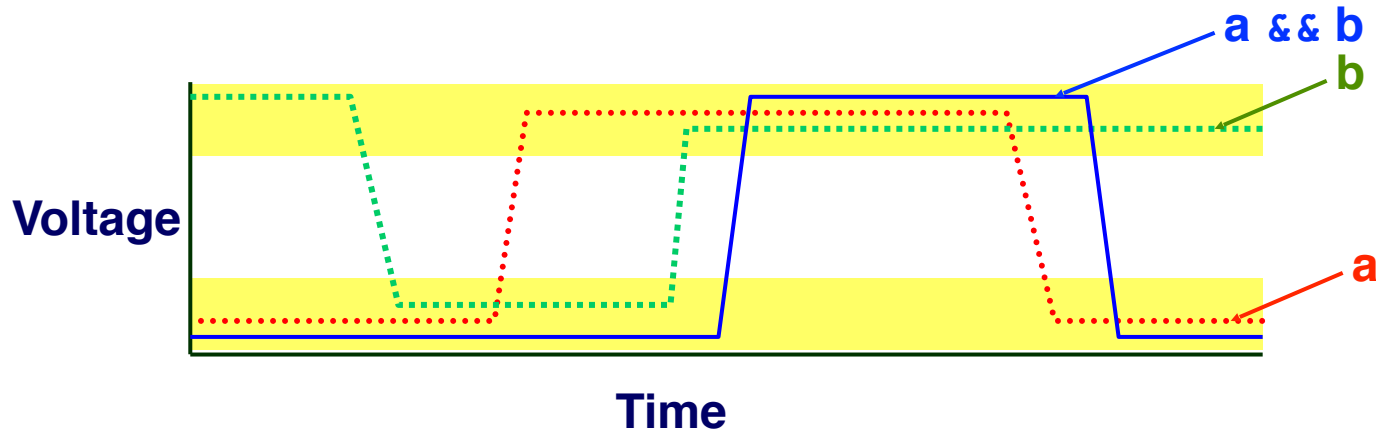
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



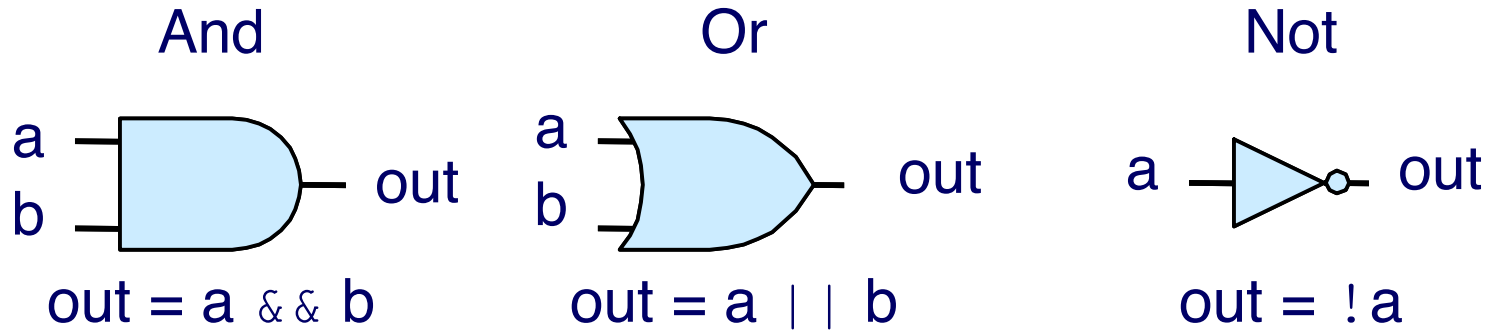
Computing with Logic Gates



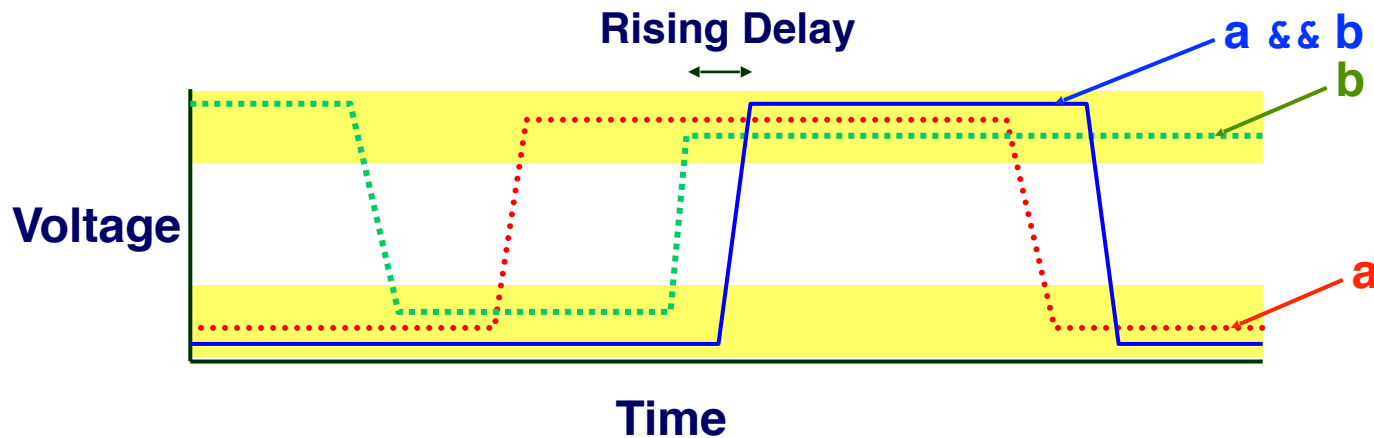
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



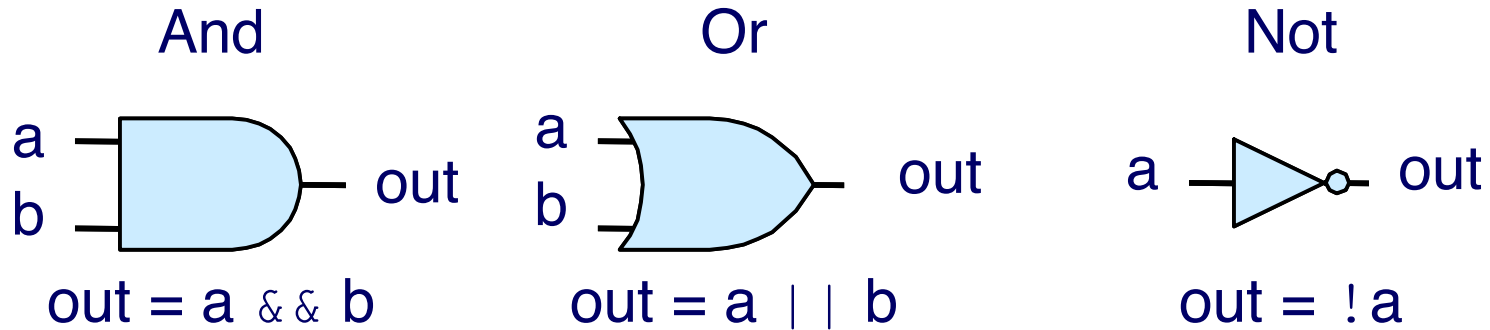
Computing with Logic Gates



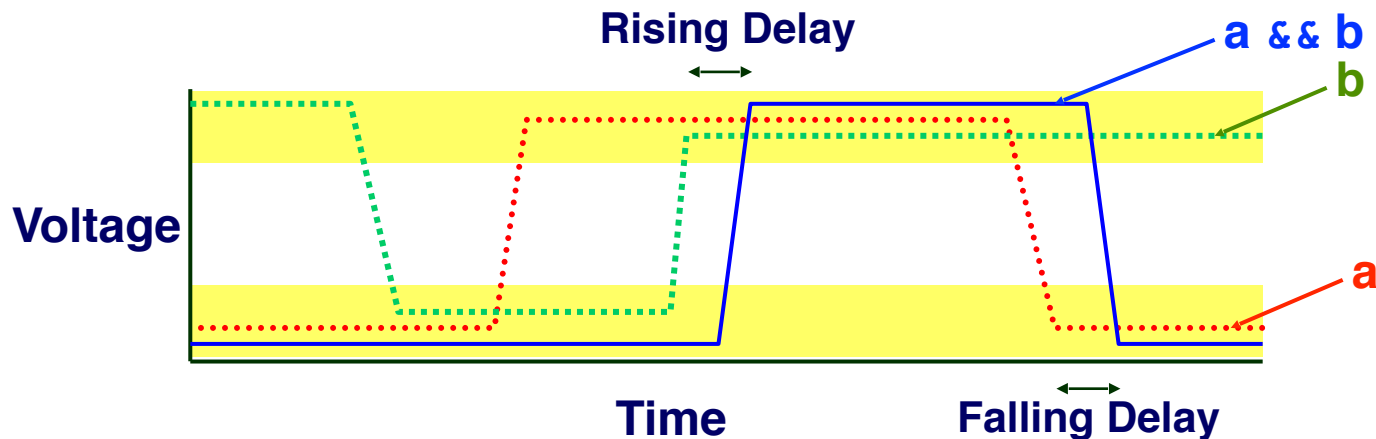
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



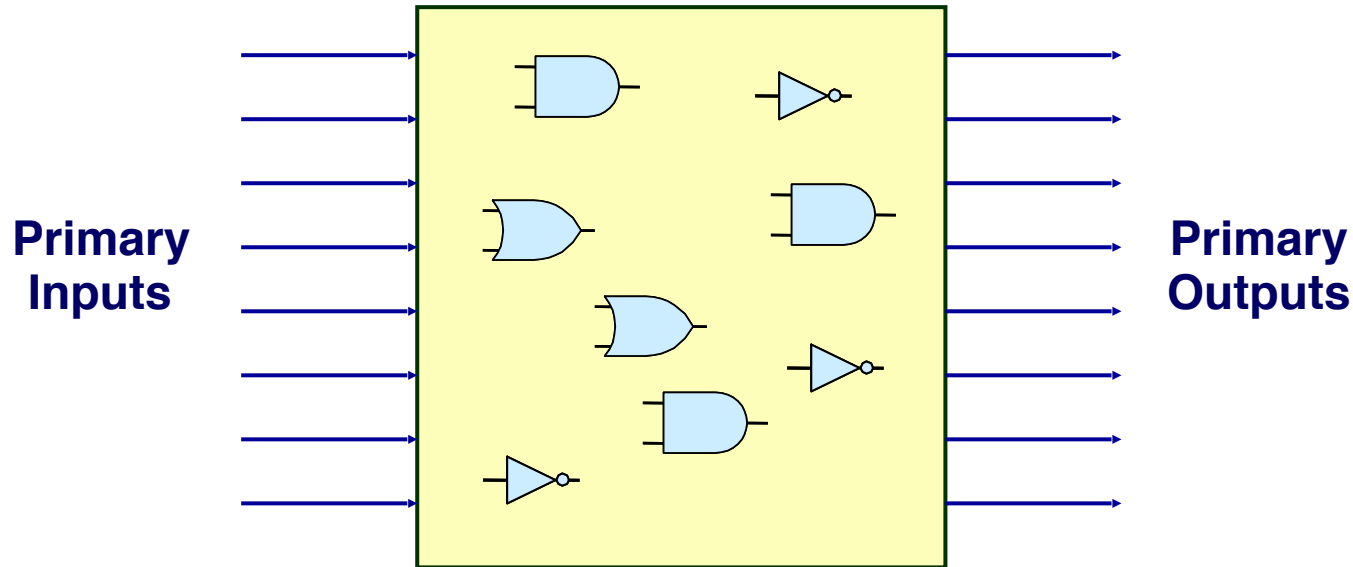
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



Combinational Circuits



- A Network of Logic Gates

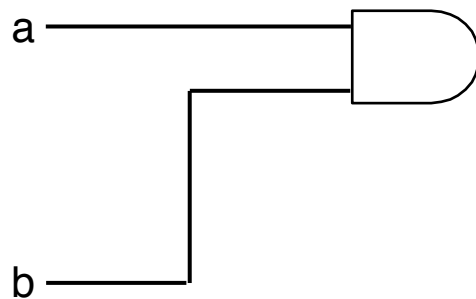
- Continuously responds to changes on primary inputs
- Primary outputs become (**after some delay**) Boolean functions of primary inputs

Bit Equality

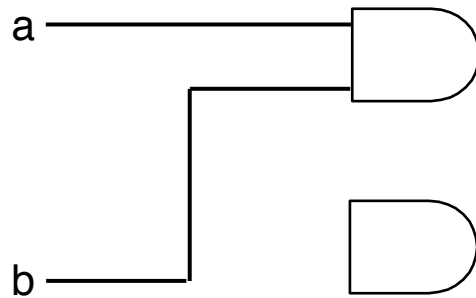
Bit Equality



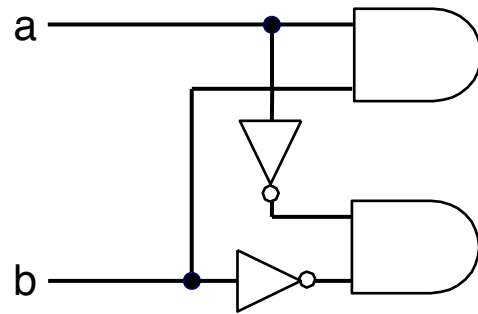
Bit Equality



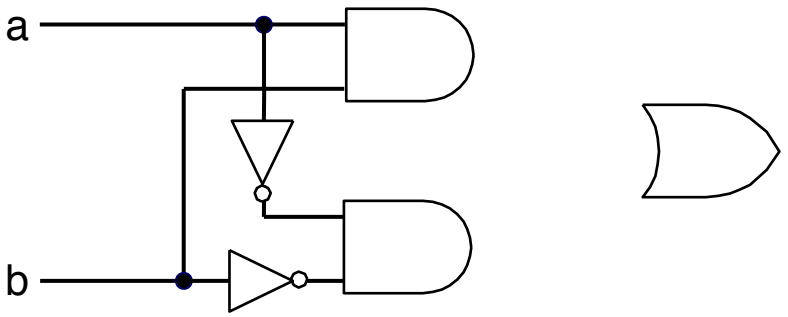
Bit Equality



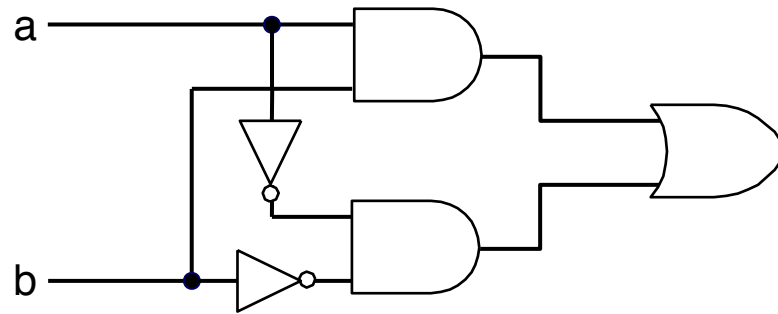
Bit Equality



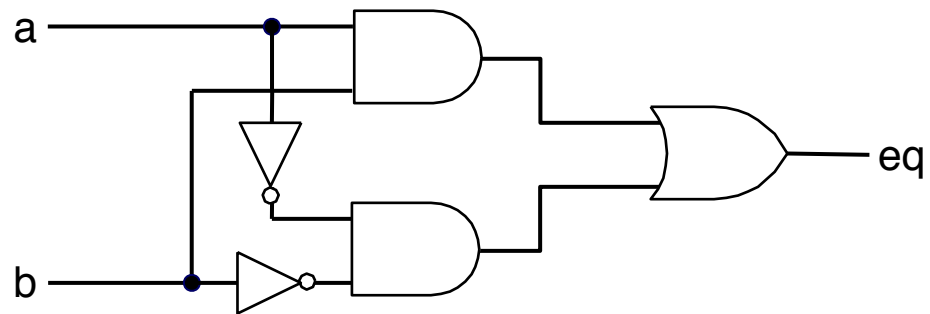
Bit Equality



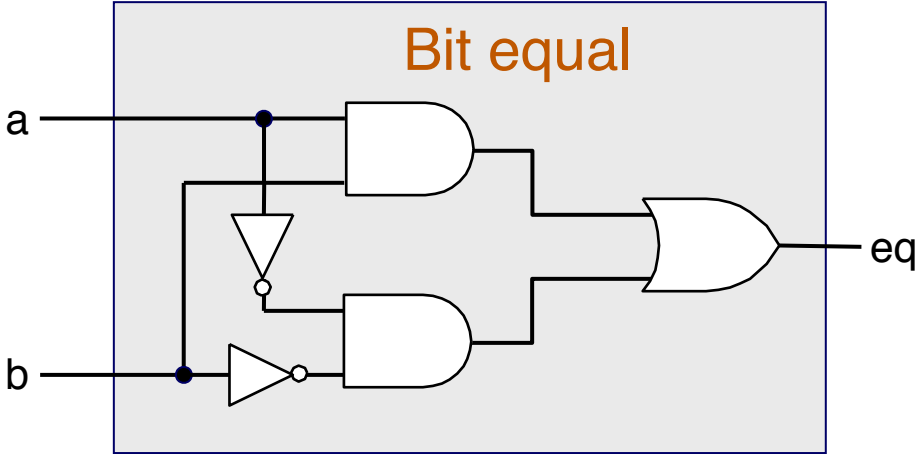
Bit Equality



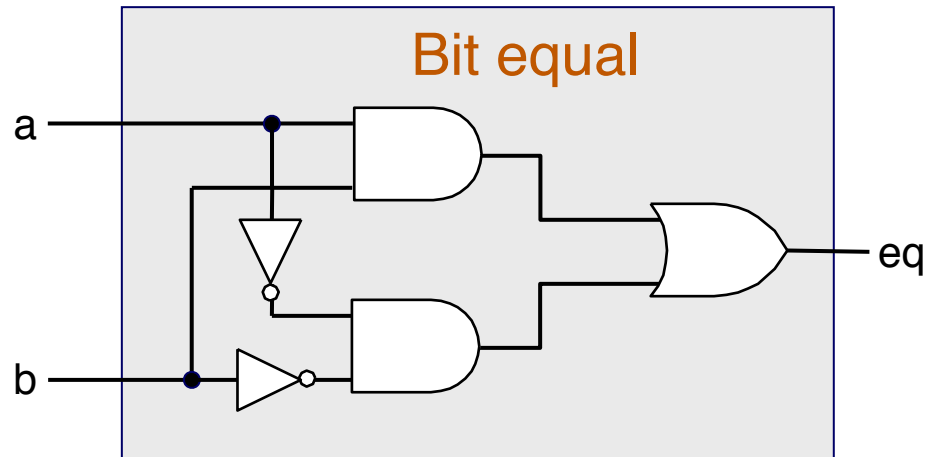
Bit Equality



Bit Equality

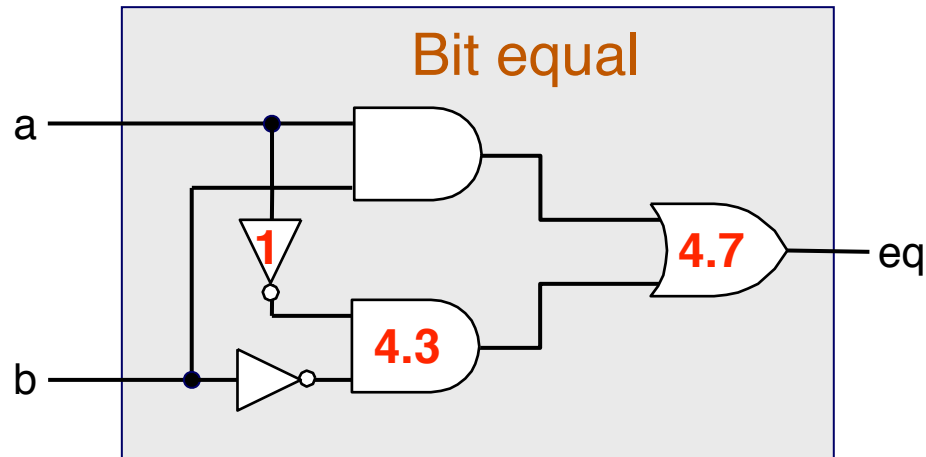


Delay of Bit Equal Circuit



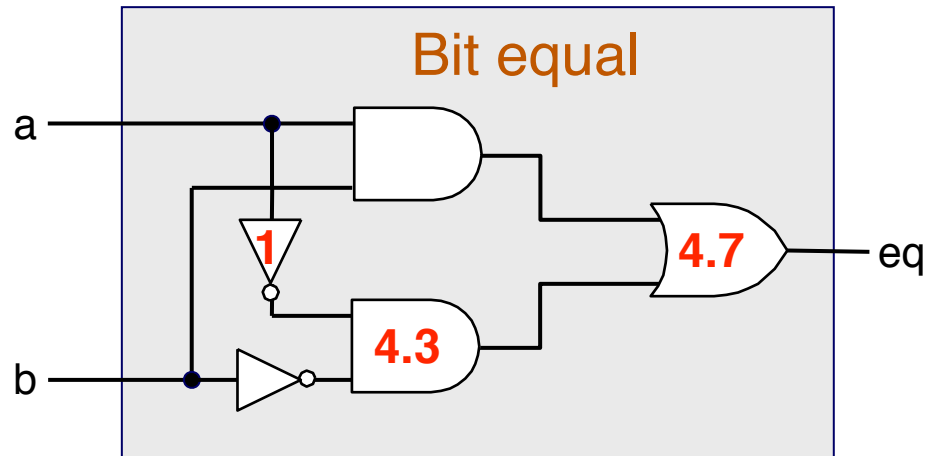
- What's the delay of this bit equal circuit?
 - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



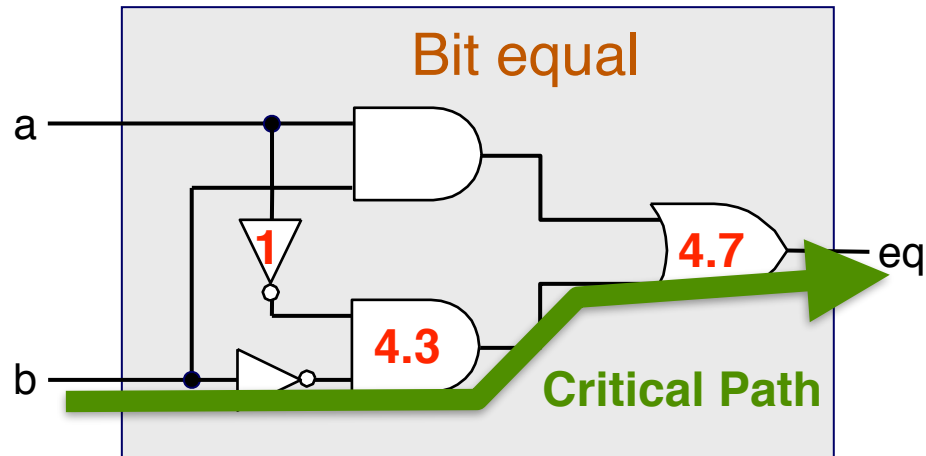
- What's the delay of this bit equal circuit?
 - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



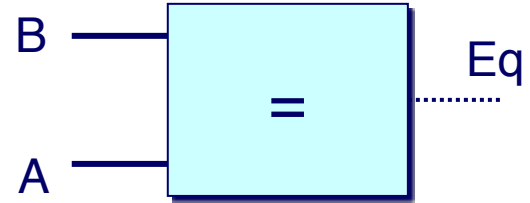
- What's the delay of this bit equal circuit?
 - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its “critical path”
 - The path between an input and the output that the maximum delay
 - Estimating the critical path delay is called static timing analysis

Delay of Bit Equal Circuit

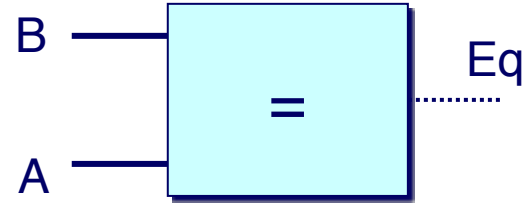
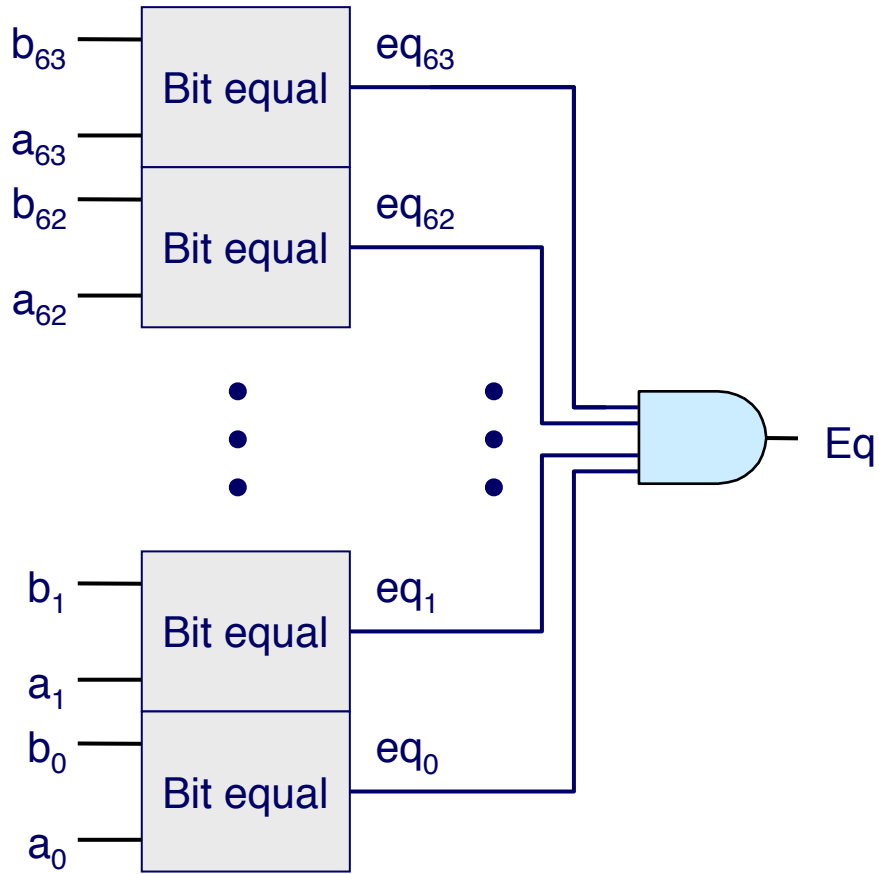


- What's the delay of this bit equal circuit?
 - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its “critical path”
 - The path between an input and the output that the maximum delay
 - Estimating the critical path delay is called static timing analysis

64-bit Equality

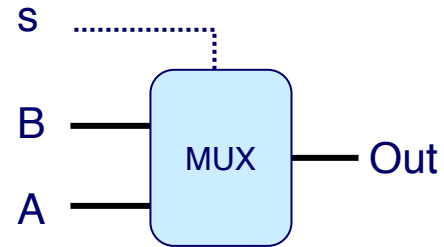


64-bit Equality



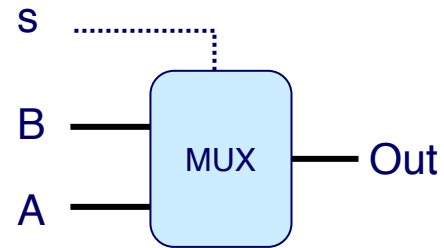
Bit-Level Multiplexor (MUX)

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



Bit-Level Multiplexor (MUX)

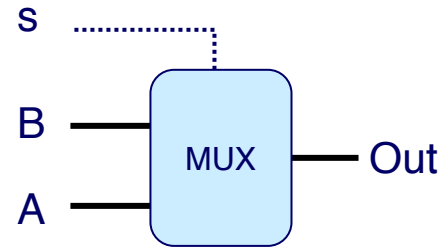
- Control signal s
- Data signals A and B
- Output A when $s=1$, B when $s=0$



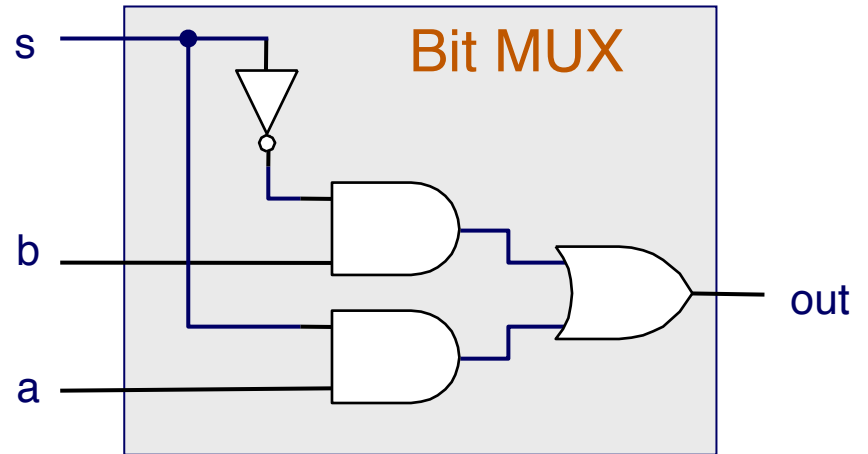
```
bool out = (s&&a) || (!s&&b)
```

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)
```

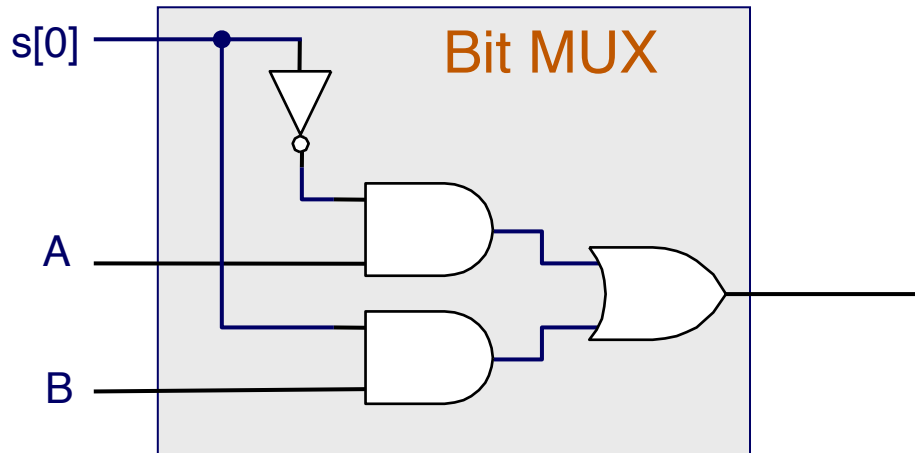


4-Input Multiplexor

- 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 $s = 11$

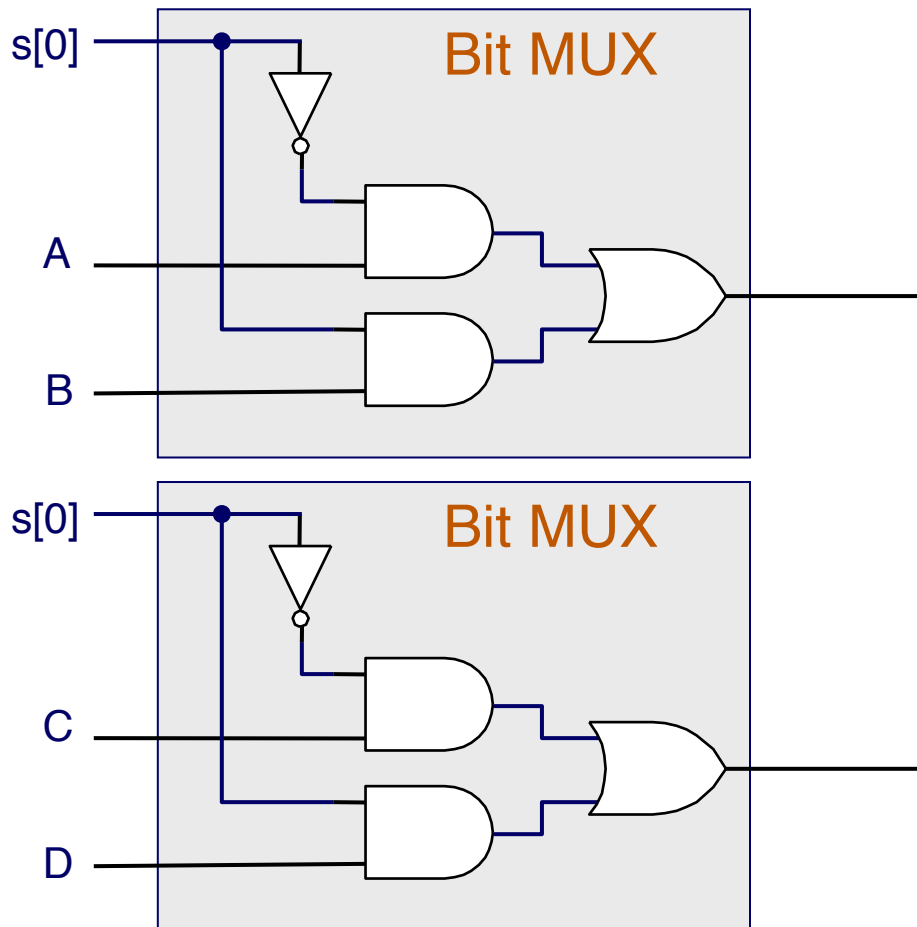
4-Input Multiplexor

- 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 $s = 11$



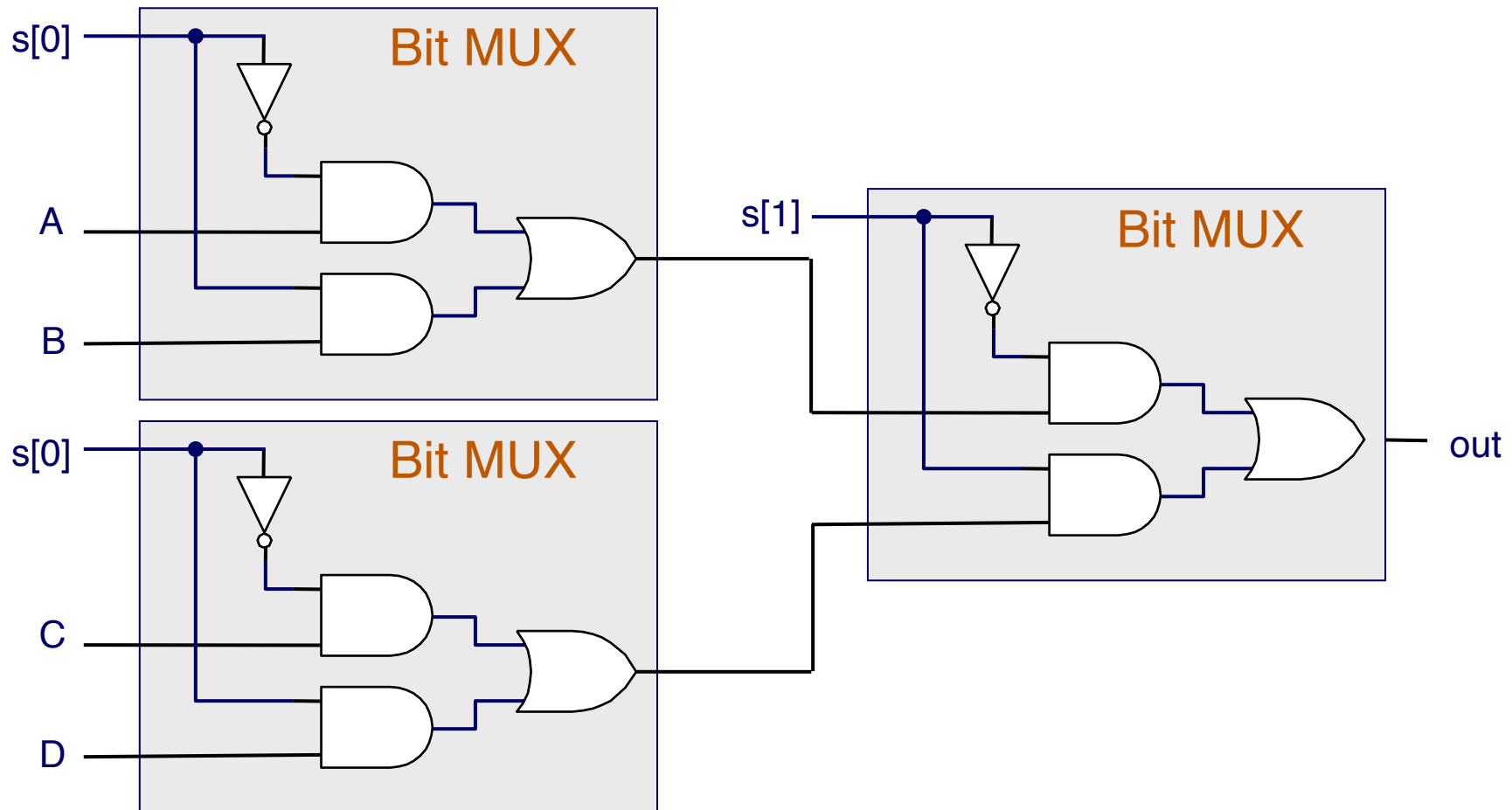
4-Input Multiplexor

- 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 $s = 11$



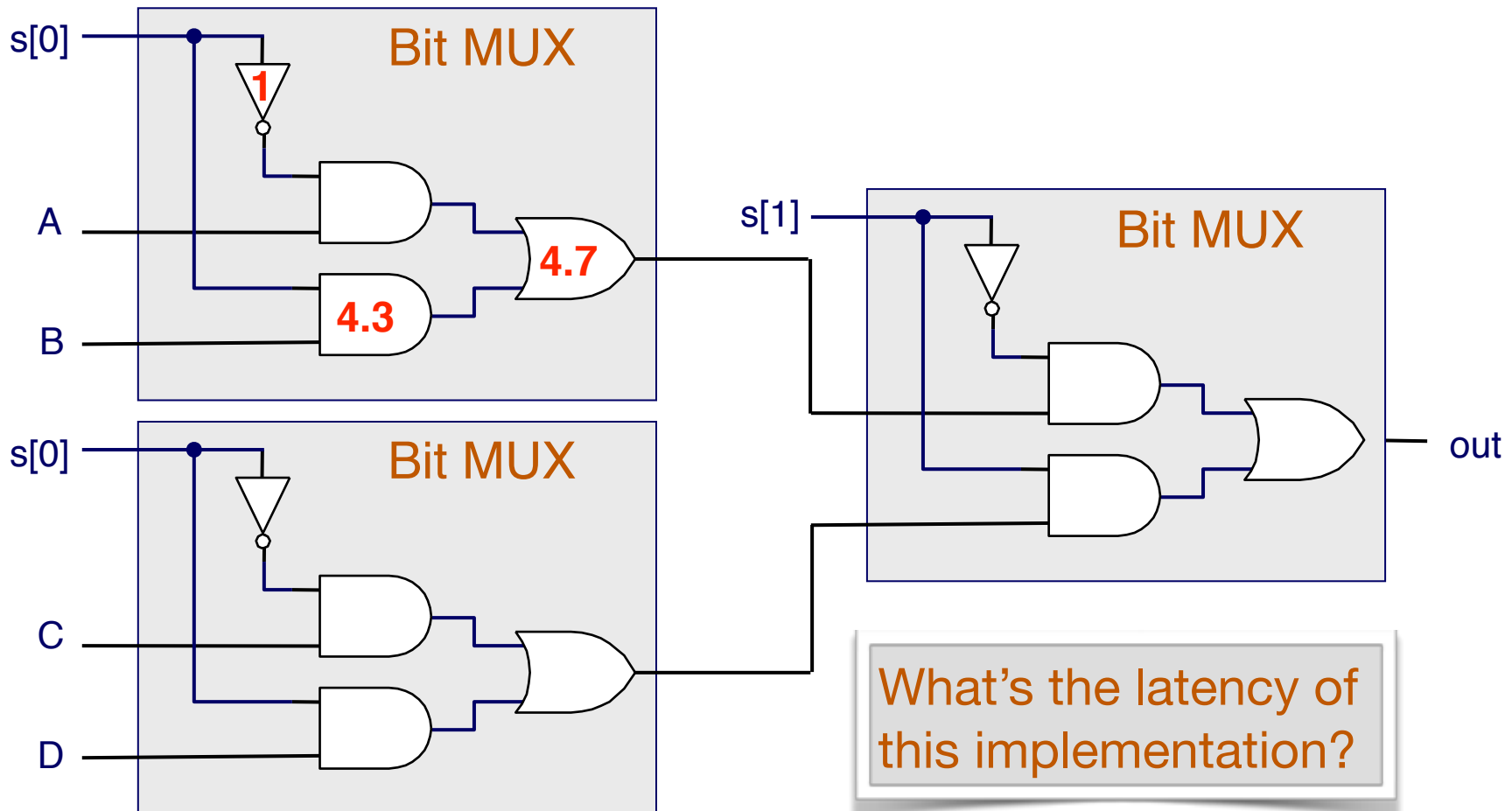
4-Input Multiplexor

- 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 $s = 11$



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.

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 LEGOs, using which you generate the gate level circuit design for complex functionalities.

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 LEGOs, 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.

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 LEGOs, 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.

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 LEGOs, 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

Full (1-bit) Adder

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

A	B	C_{in}	S	C_{ou} t
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

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

1-bit Full Adder

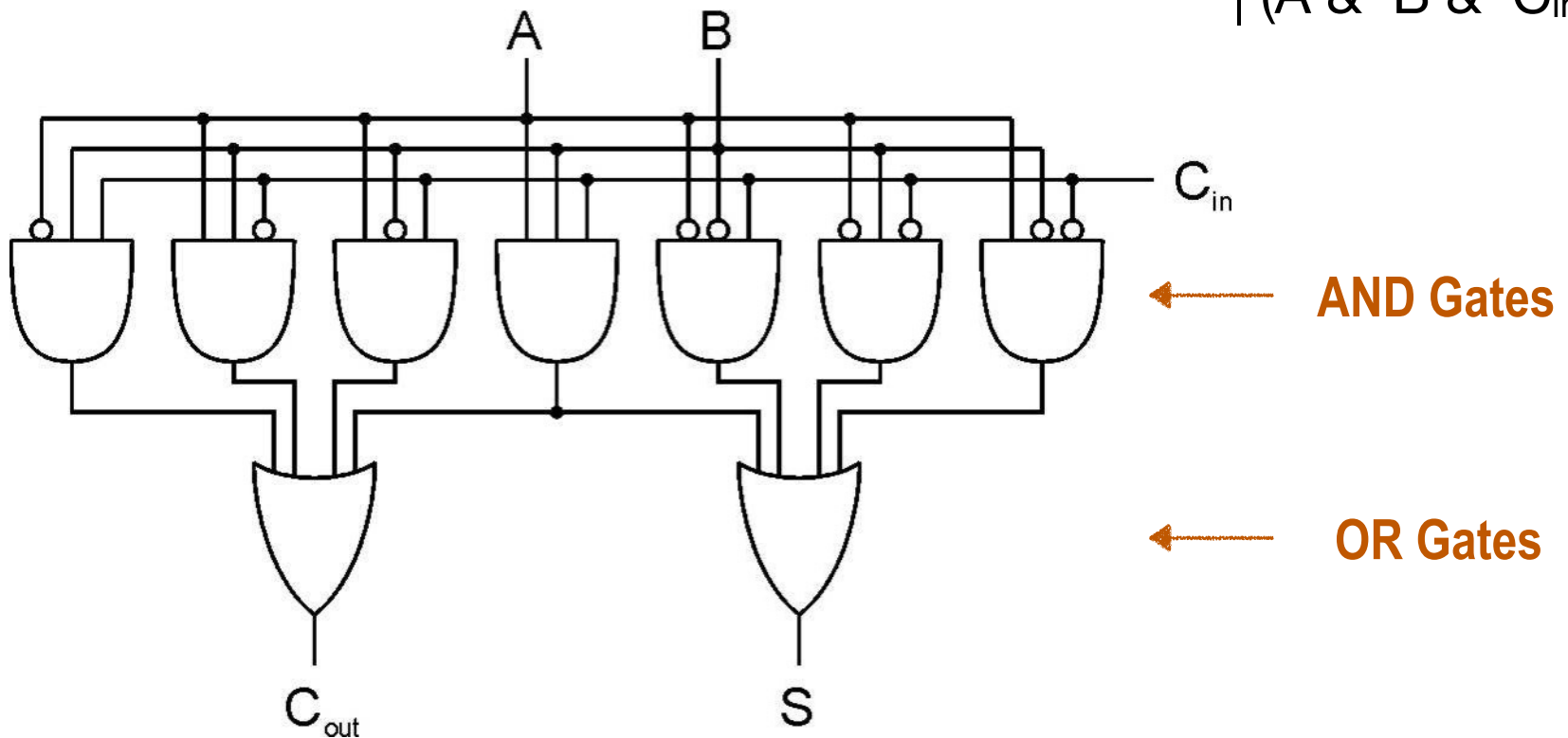
Add two bits and carry-in,
produce one-bit sum and carry-out.

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

1-bit Full Adder

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

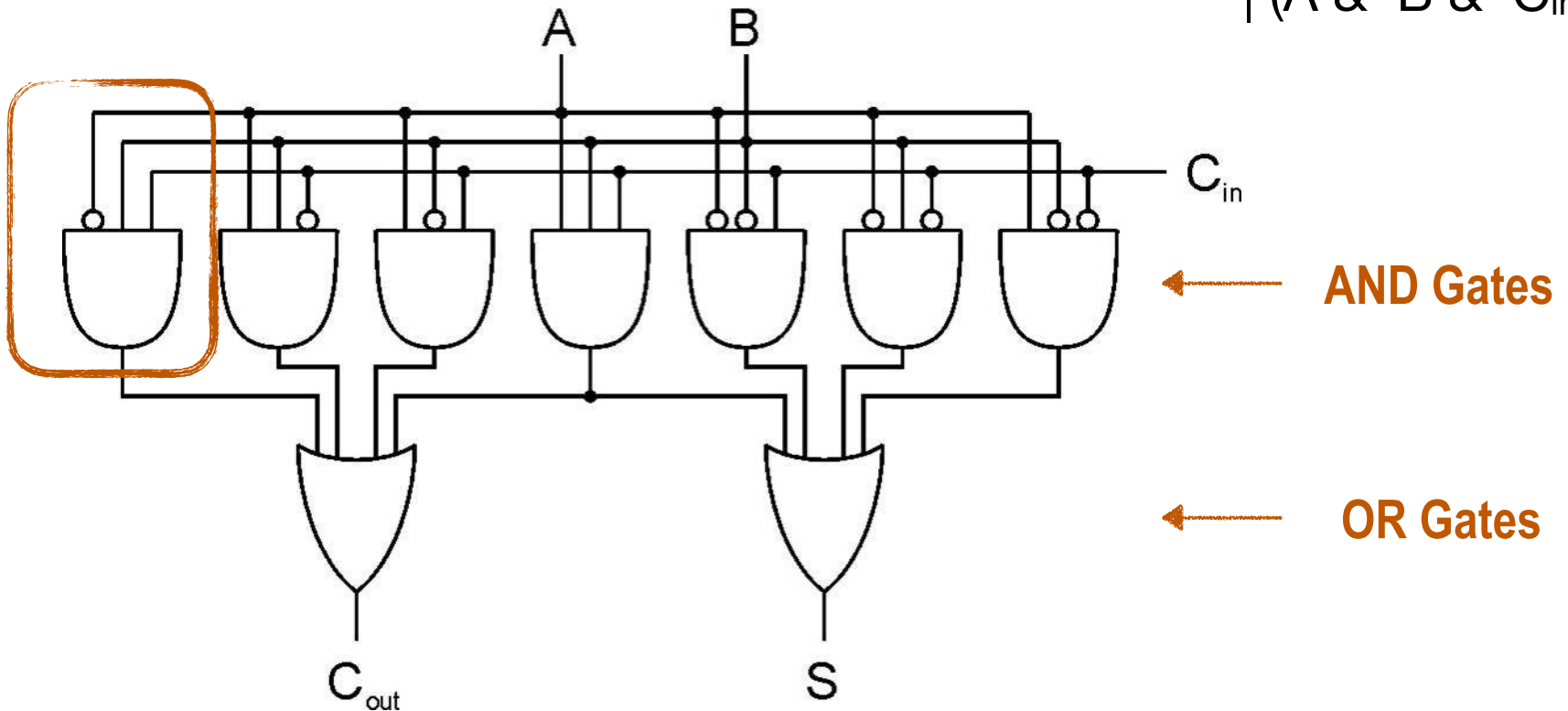
Add two bits and carry-in,
produce one-bit sum and carry-out.



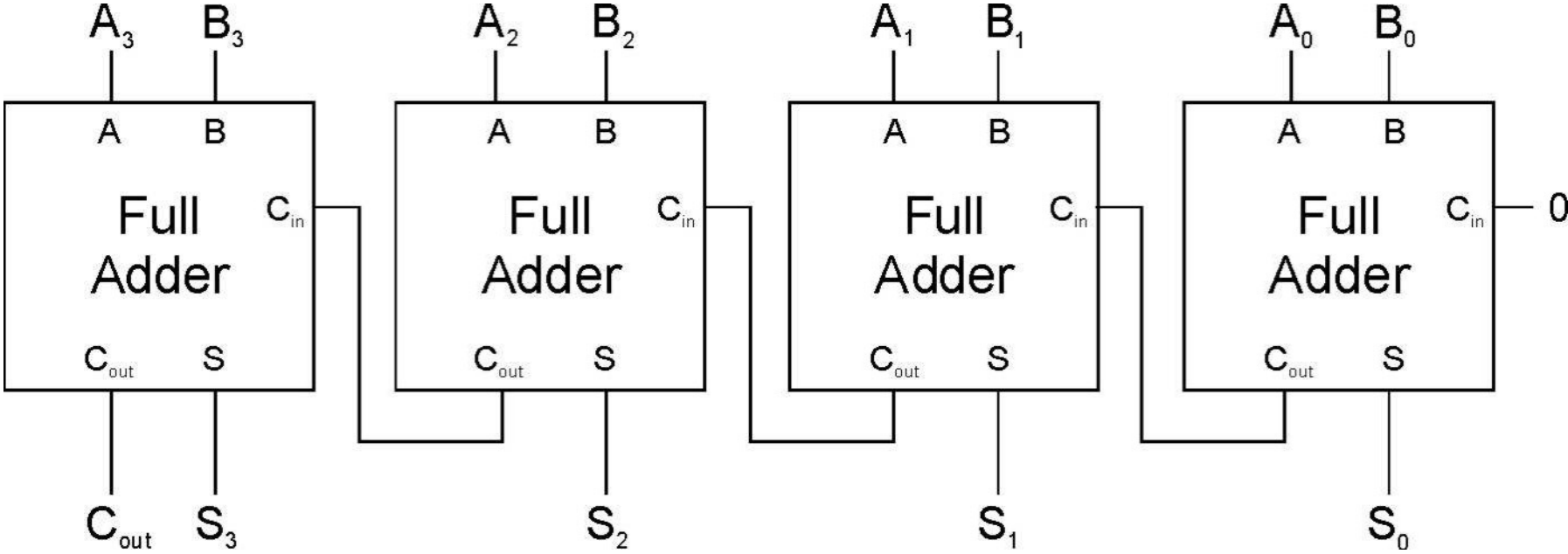
1-bit Full Adder

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

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

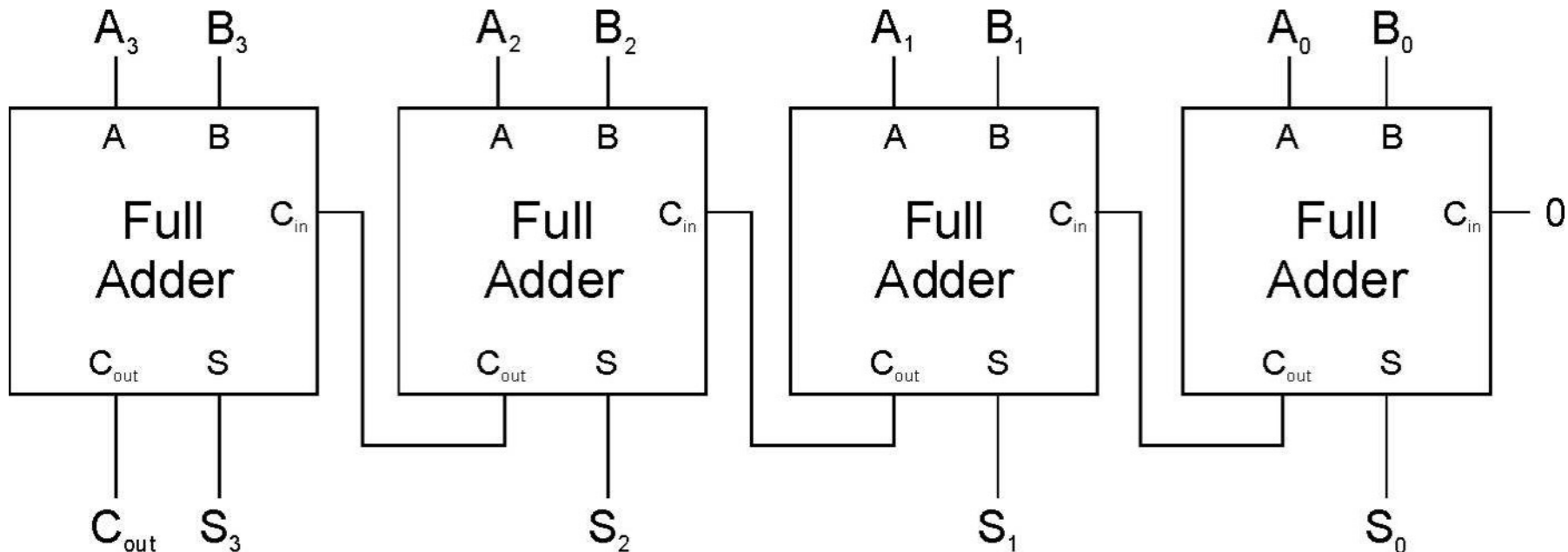


Four-bit Adder



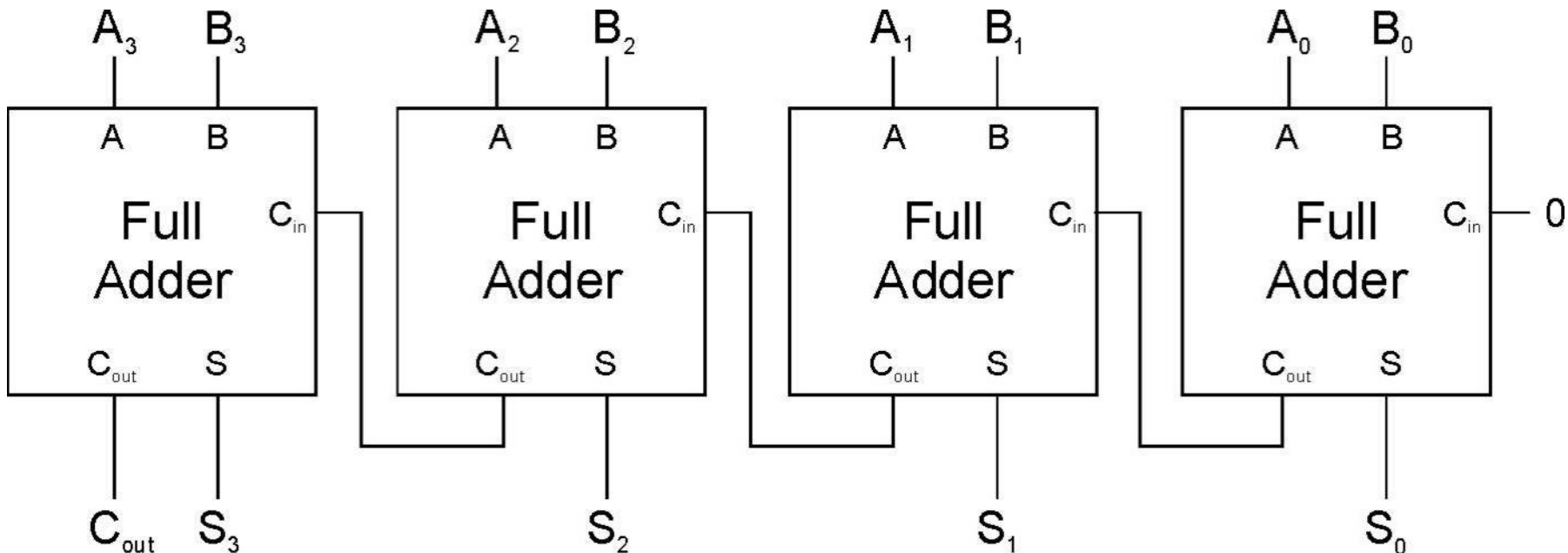
Four-bit Adder

- Ripple-carry Adder
 - Simple, but performance linear to bit width

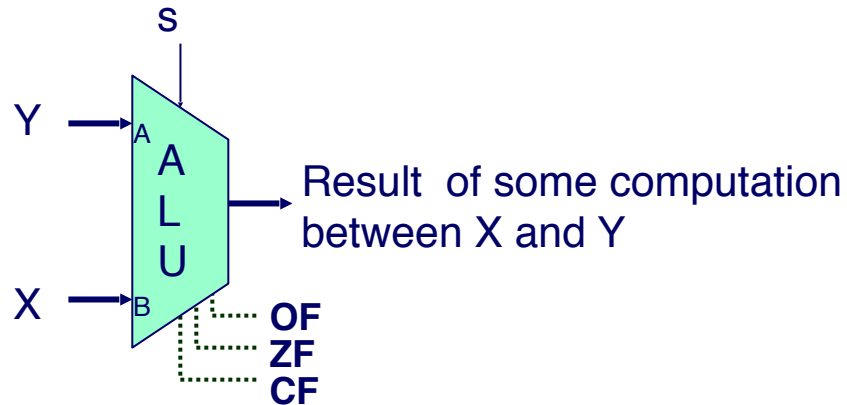


Four-bit Adder

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



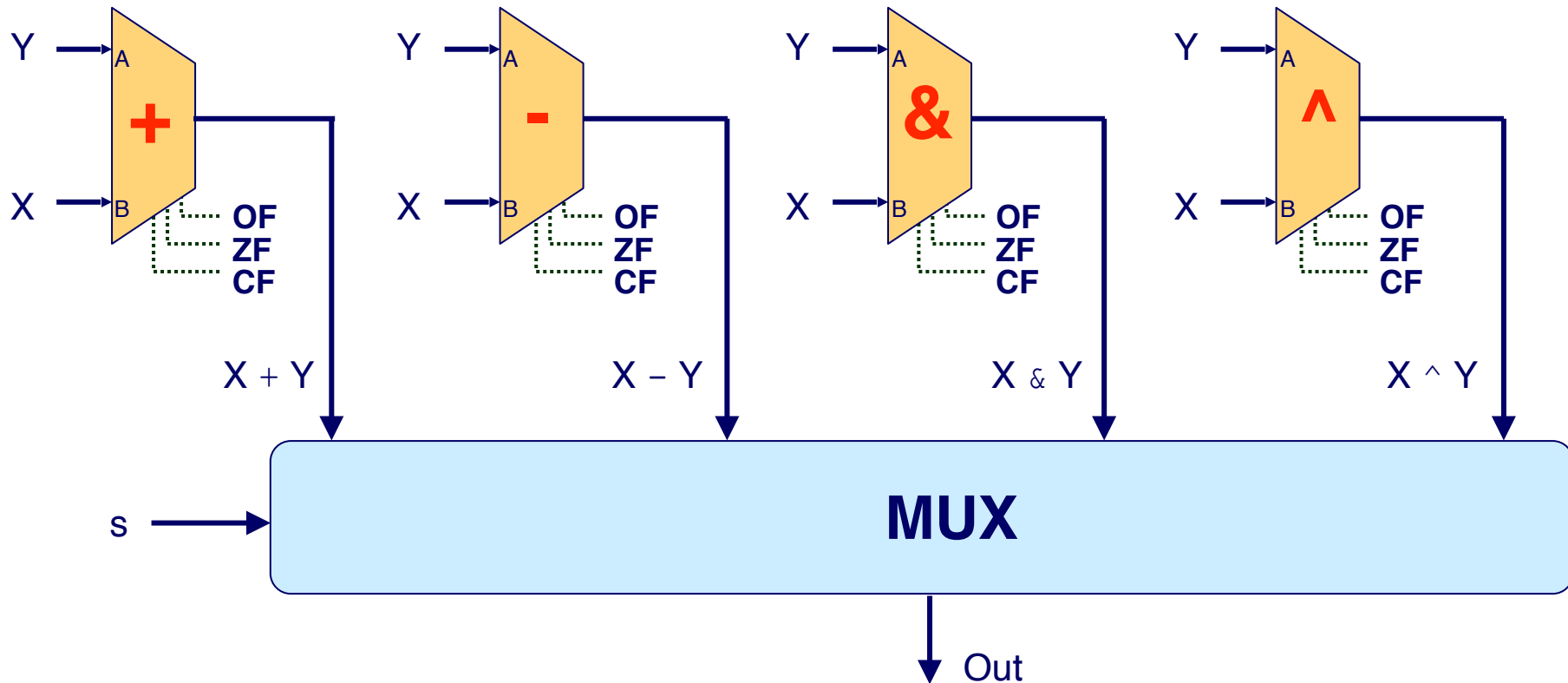
Arithmetic Logic Unit



- 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

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.

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.

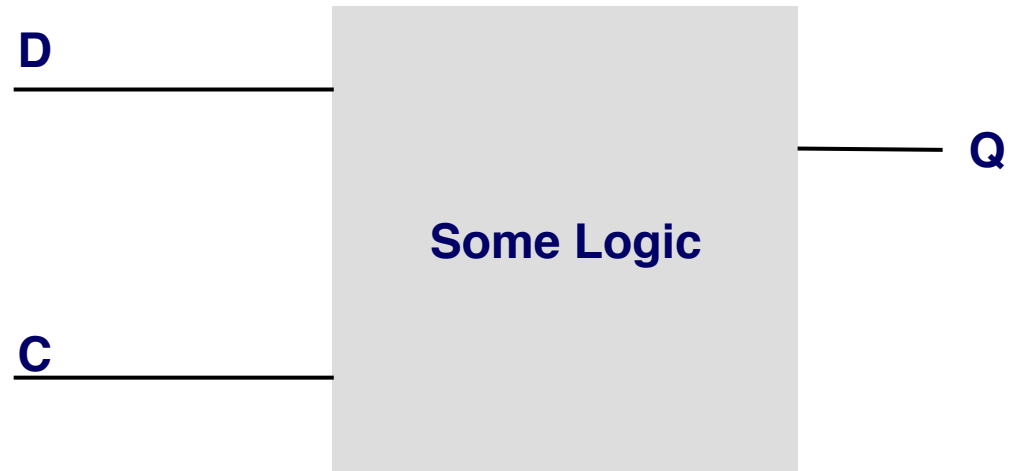
The Need for Storing Bits

- 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.

The Need for Storing Bits

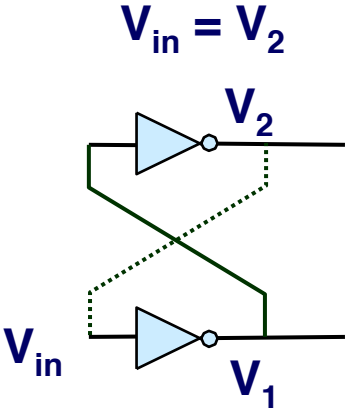
- Assembly programs set architecture (processor) states.
 - Register File
 - Status Flags
 - Memory
 - Program Counter
- Every state is essentially some bits that are stored/loaded.
- Think of the program execution as an FSM.
- The hardware must provide mechanisms to load and store bits.
- There are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

Build a 1-Bit Storage

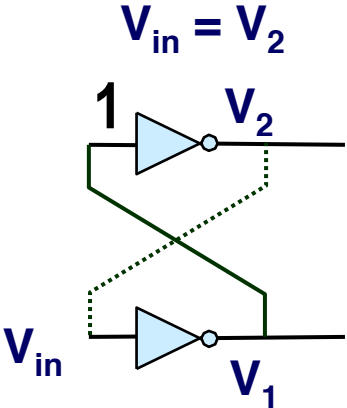


- What I would like:
 - D is the data I want to store (0 or 1)
 - C is the control signal
 - When C is 1, Q becomes D (i.e., storing the data)
 - When C is 0, Q doesn't change with D (data stored)

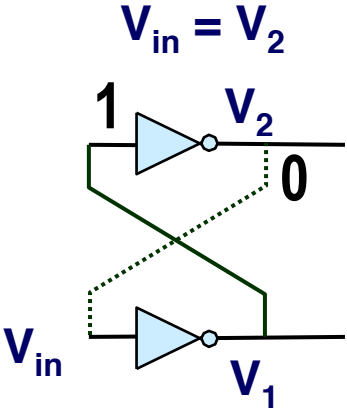
Bitstable Element



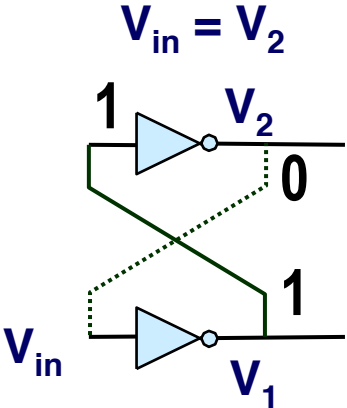
Bitstable Element



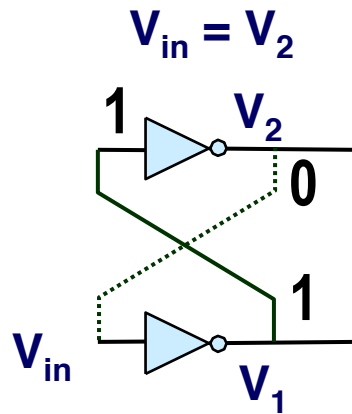
Bitstable Element



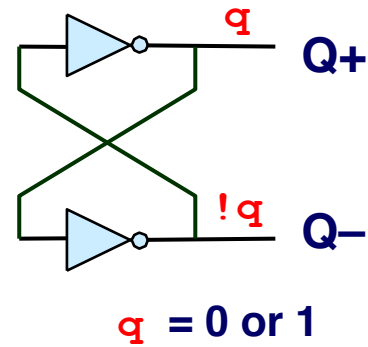
Bitstable Element



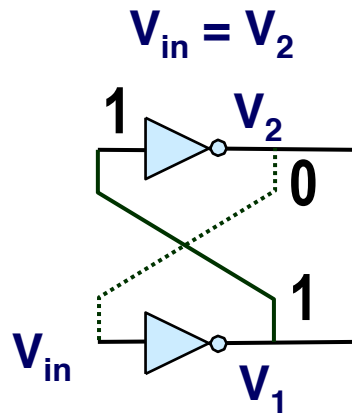
Bitstable Element



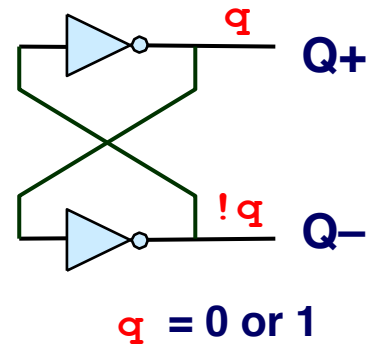
Bistable Element



Bitstable Element



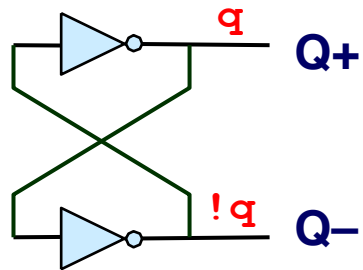
Bistable Element



$Q+$ *continuously* outputs q .

Storing and Accessing 1 Bit

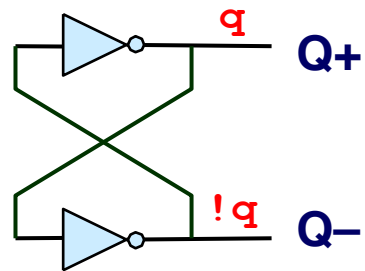
Bistable Element



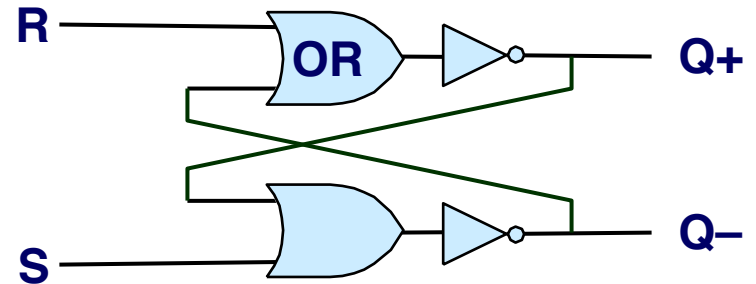
$q = 0$ or 1

Storing and Accessing 1 Bit

Bistable Element

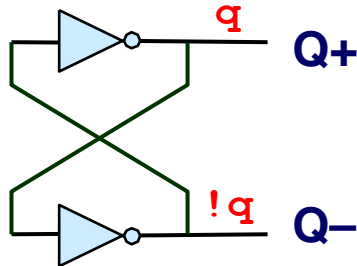


$q = 0$ or 1

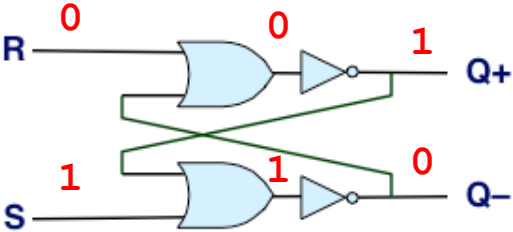
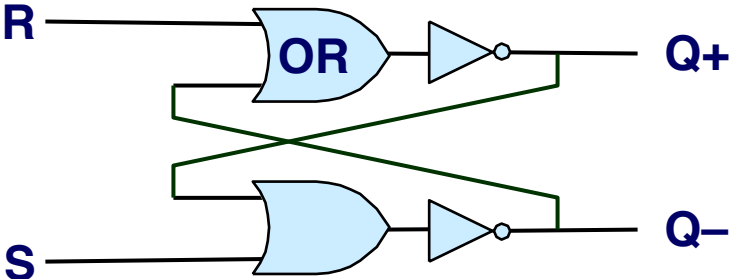


Storing and Accessing 1 Bit

Bistable Element

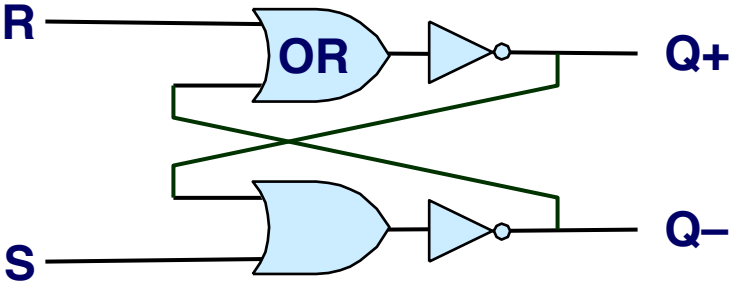
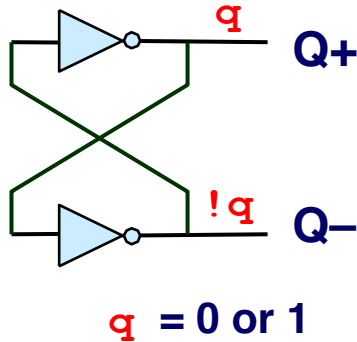


$q = 0$ or 1

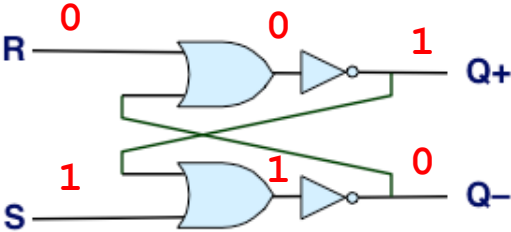


Storing and Accessing 1 Bit

Bistable Element

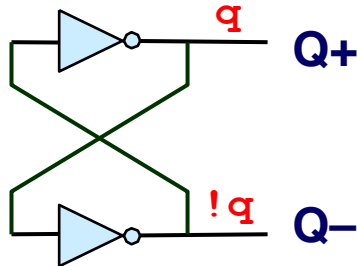


Setting $Q+$ to 1

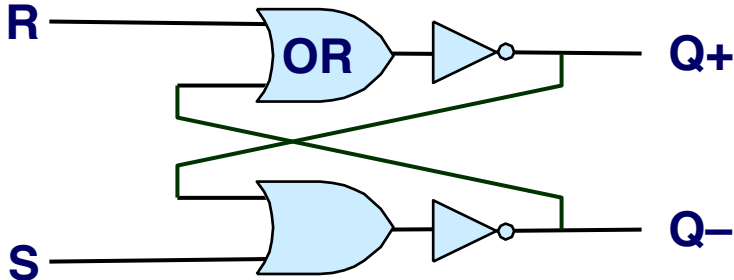


Storing and Accessing 1 Bit

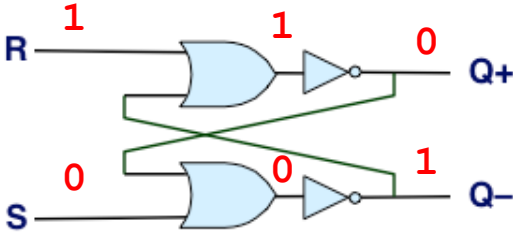
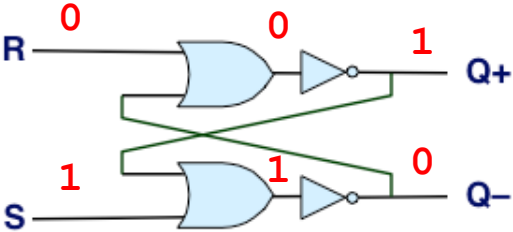
Bistable Element



$q = 0$ or 1

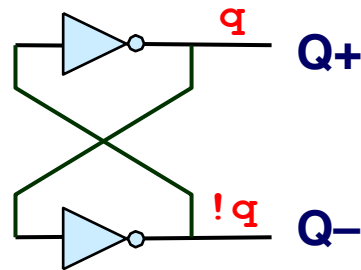


Setting $Q+$ to 1

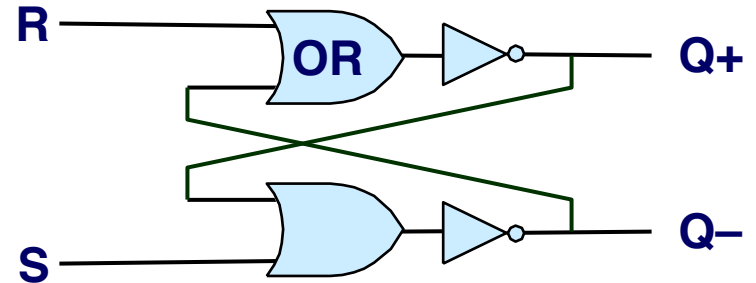


Storing and Accessing 1 Bit

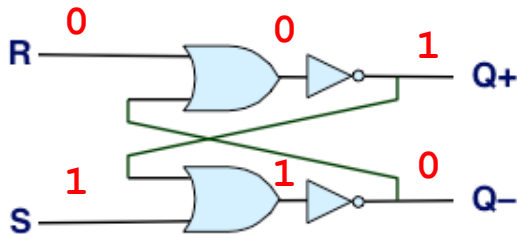
Bistable Element



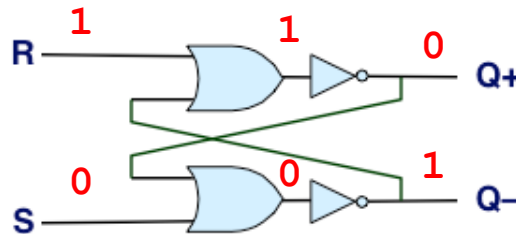
$q = 0$ or 1



Setting $Q+$ to 1

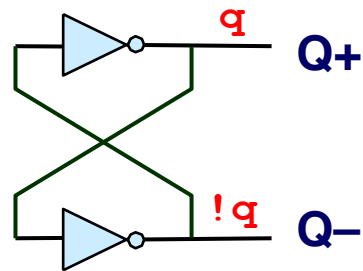


Setting $Q+$ to 0

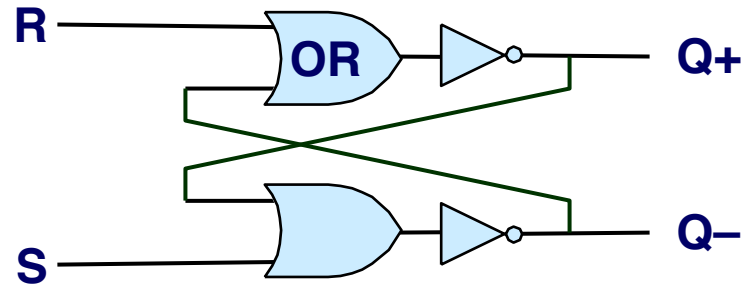


Storing and Accessing 1 Bit

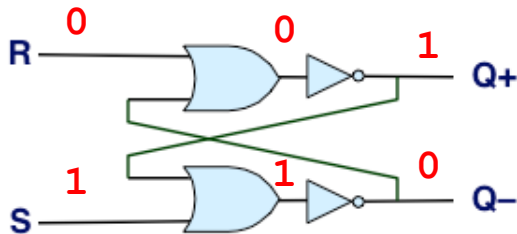
Bistable Element



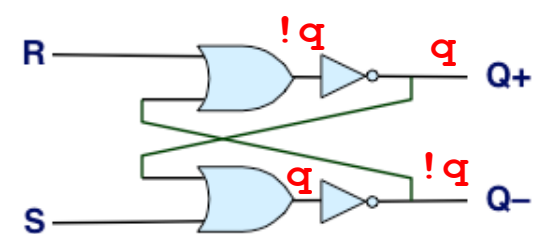
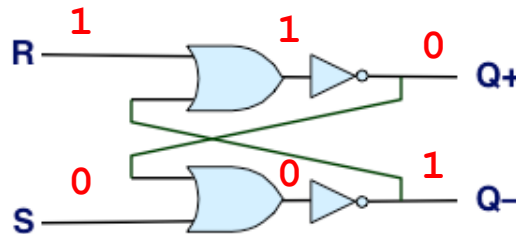
$q = 0$ or 1



Setting $Q+$ to 1

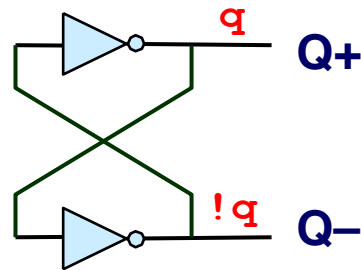


Setting $Q+$ to 0

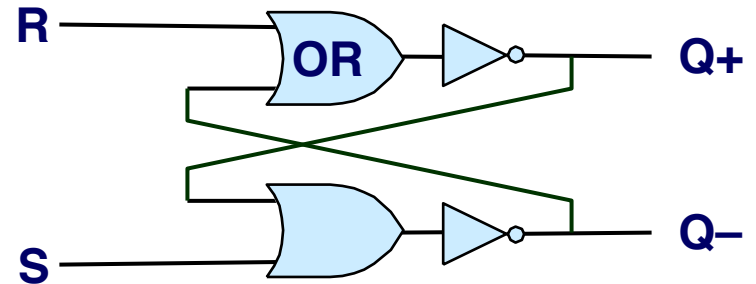


Storing and Accessing 1 Bit

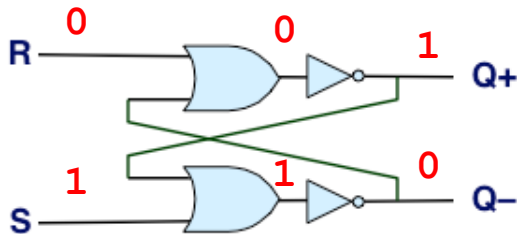
Bistable Element



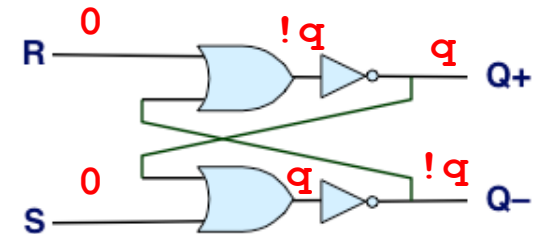
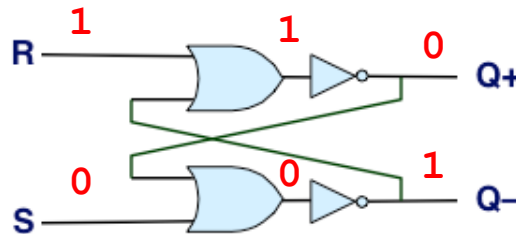
$q = 0$ or 1



Setting $Q+$ to 1

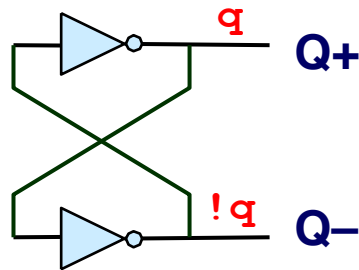


Setting $Q+$ to 0

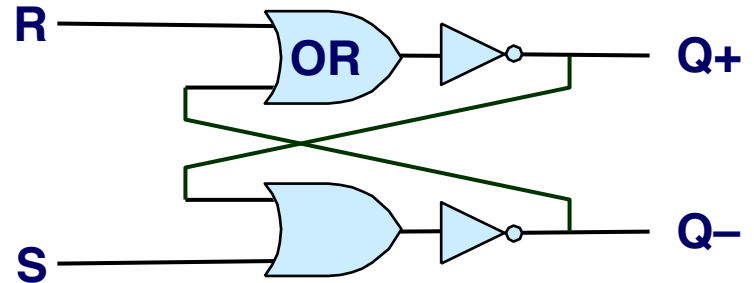


Storing and Accessing 1 Bit

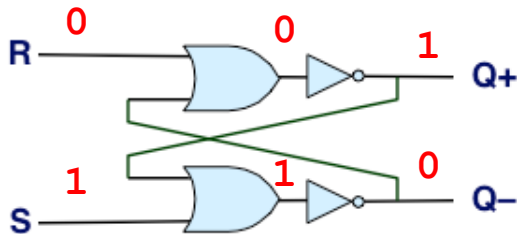
Bistable Element



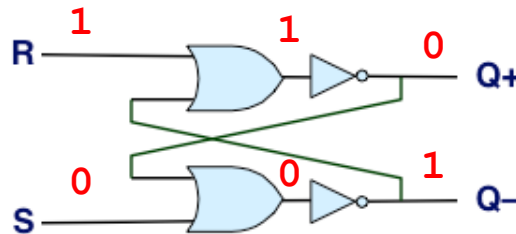
$q = 0$ or 1



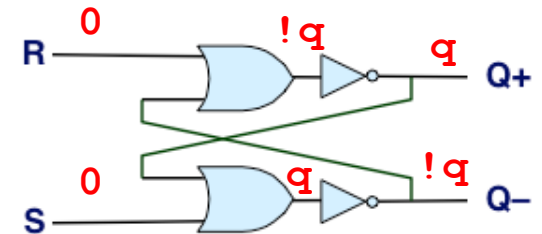
Setting $Q+$ to 1



Setting $Q+$ to 0

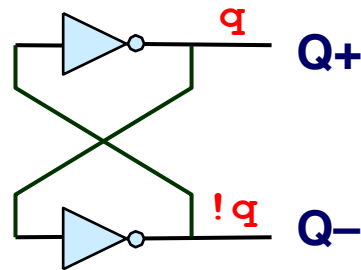


$Q+$ value unchanged
i.e., stored!



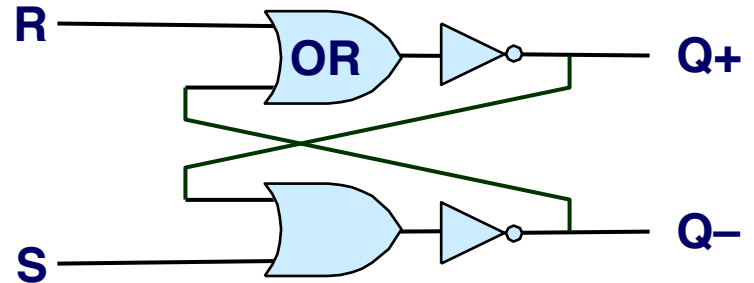
Storing and Accessing 1 Bit

Bistable Element

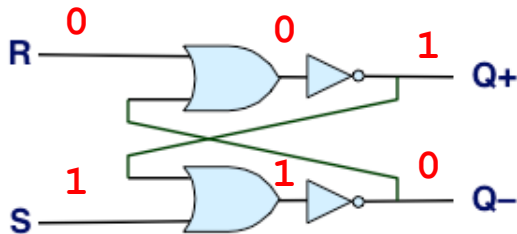


$q = 0 \text{ or } 1$

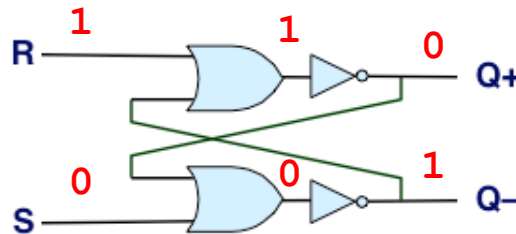
R-S Latch



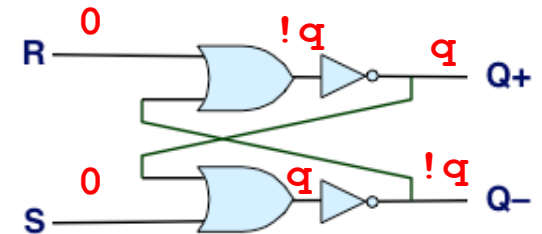
Setting $Q+$ to 1



Setting $Q+$ to 0

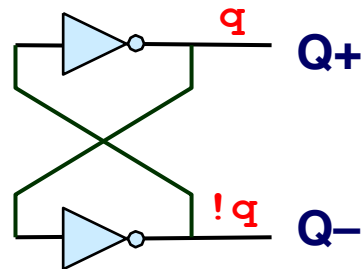


$Q+$ value unchanged
i.e., stored!



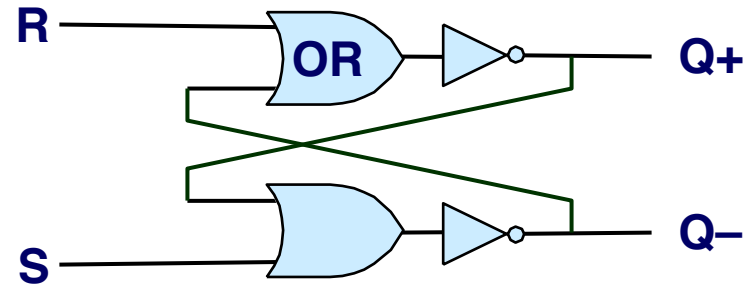
Storing and Accessing 1 Bit

Bistable Element

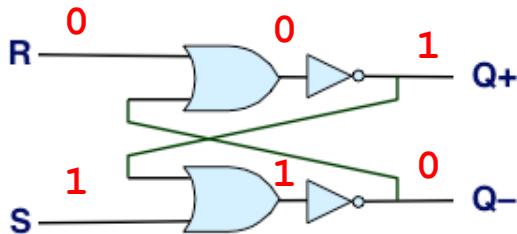


$q = 0 \text{ or } 1$

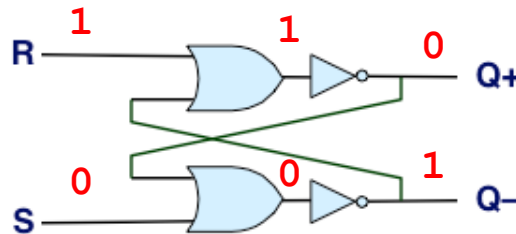
R-S Latch



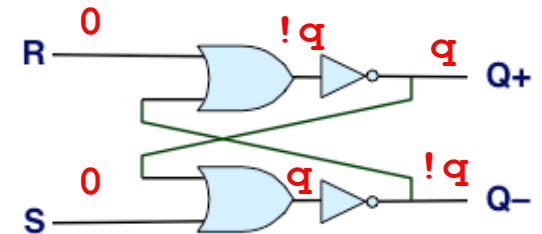
Setting $Q+$ to 1



Setting $Q+$ to 0

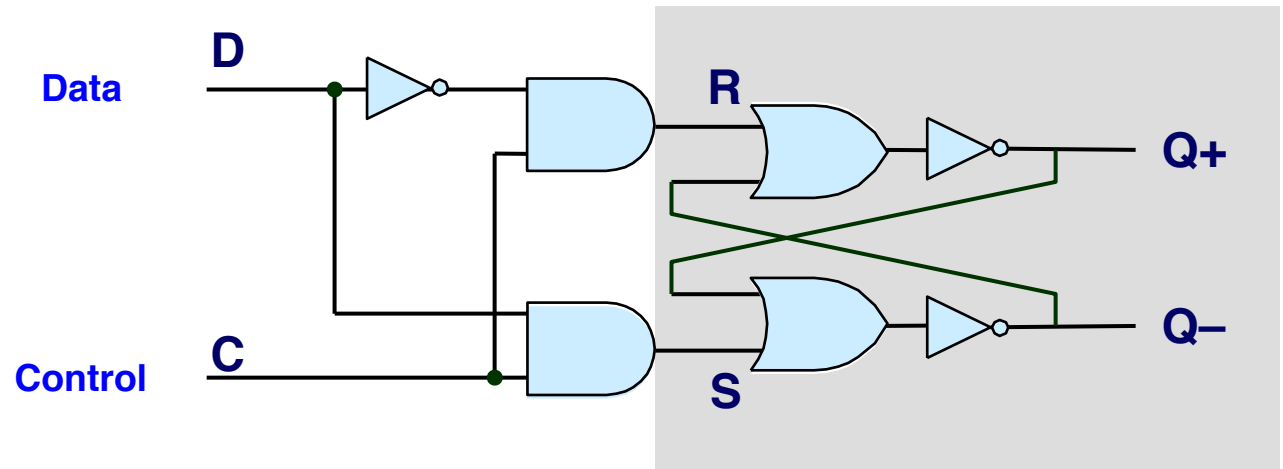


$Q+$ value unchanged
i.e., stored!



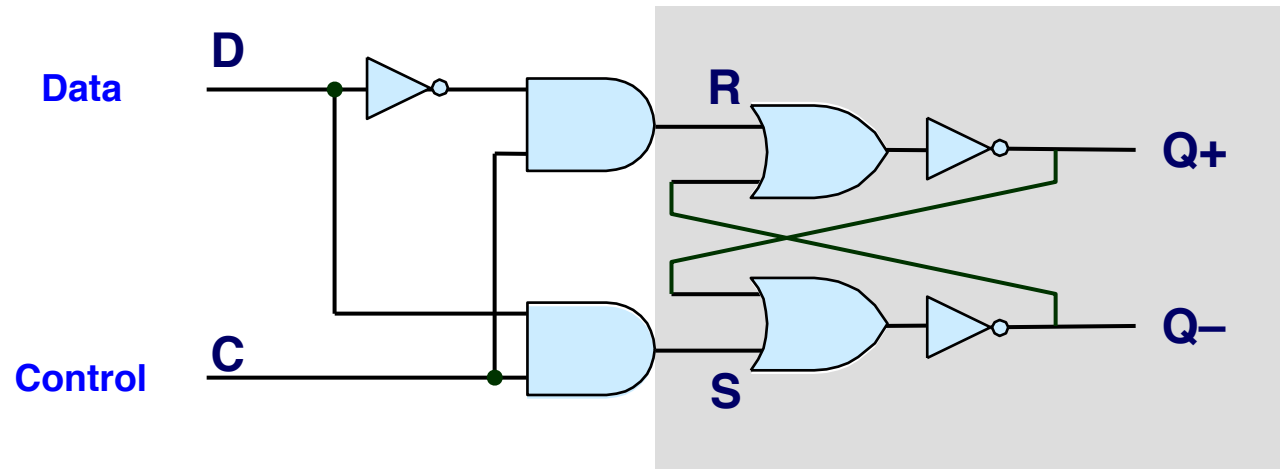
If R and S are different, $Q+$ is the same as S

Building on top of R-S Latch

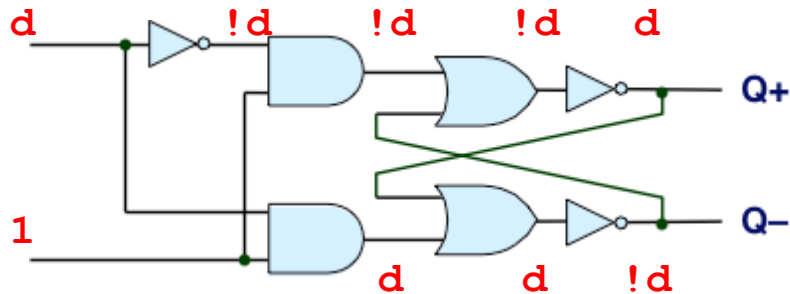


If R and S are different, $Q+$ is the same as S

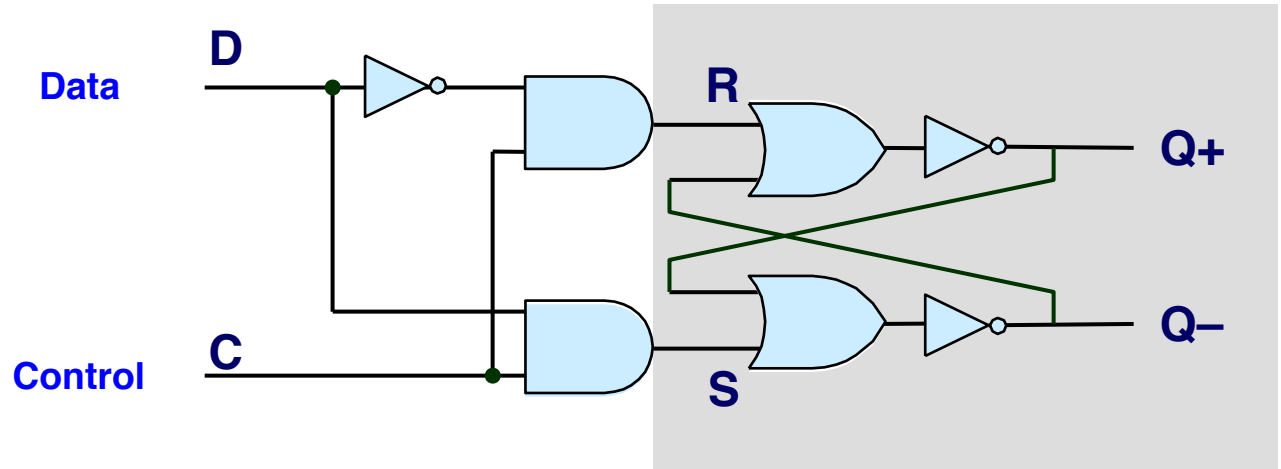
Building on top of R-S Latch



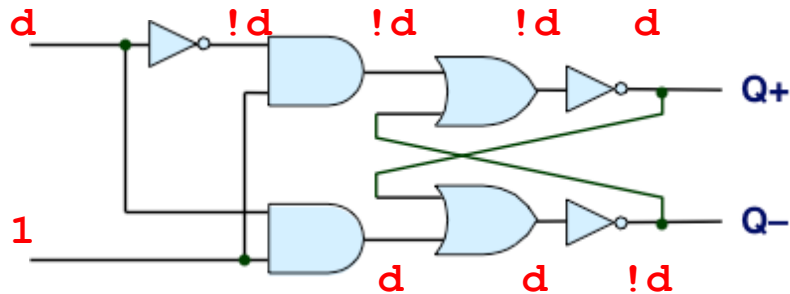
If R and S are different, Q+ is the same as S



Building on top of R-S Latch

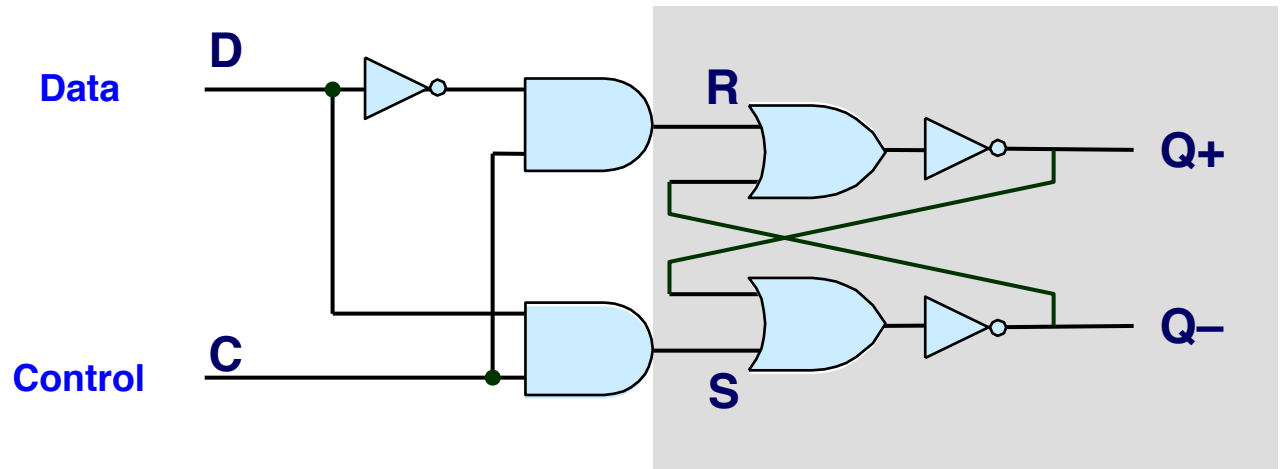


If R and S are different, Q+ is the same as S



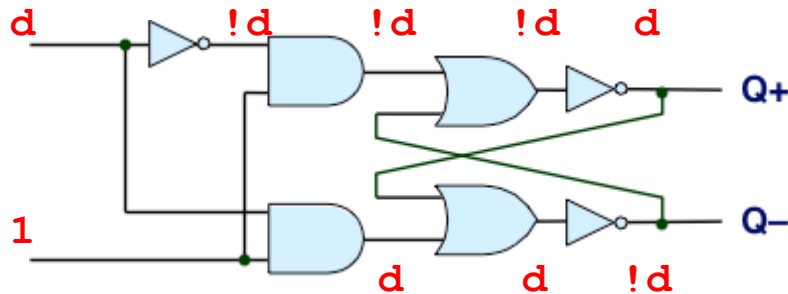
Q+ will continuously change as d changes

Building on top of R-S Latch



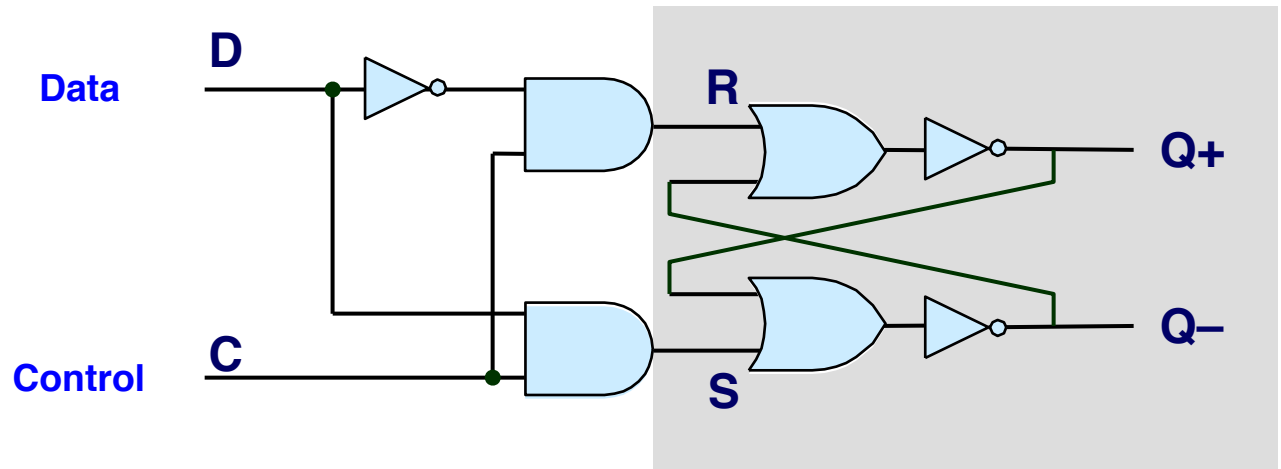
If R and S are different, Q+ is the same as S

Storing Data (Latching)



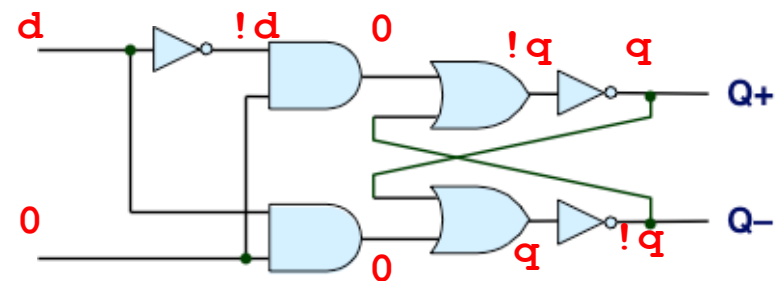
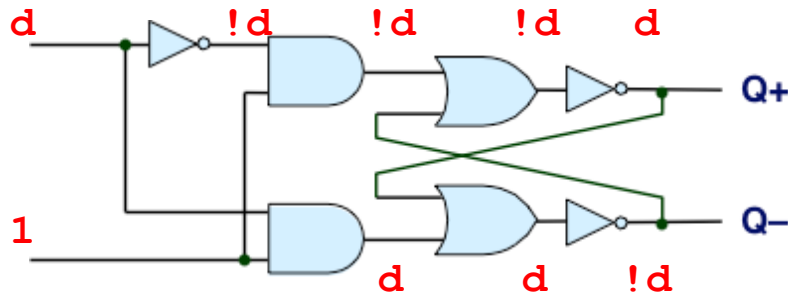
Q+ will continuously change as d changes

Building on top of R-S Latch



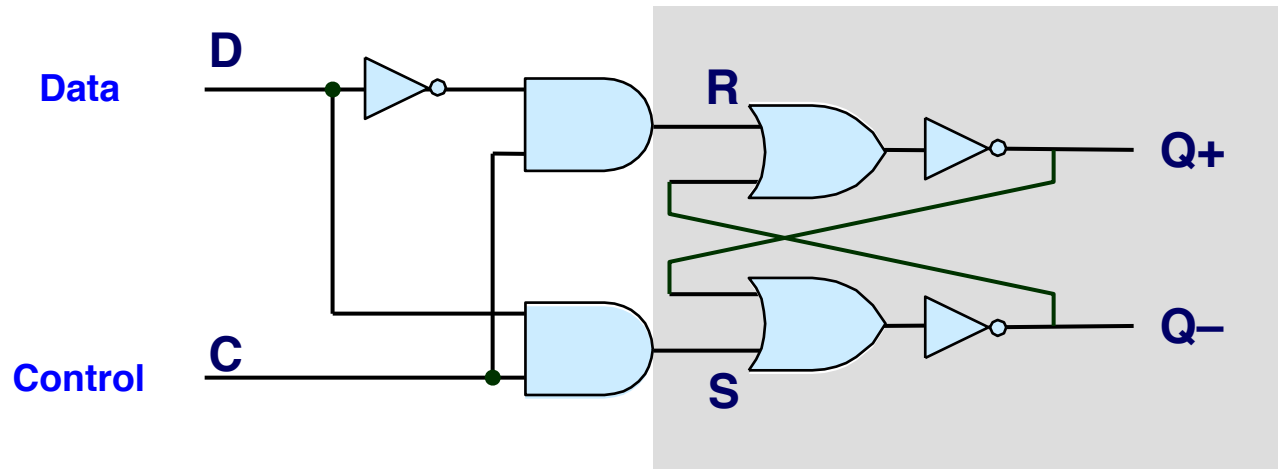
If R and S are different, Q+ is the same as S

Storing Data (Latching)



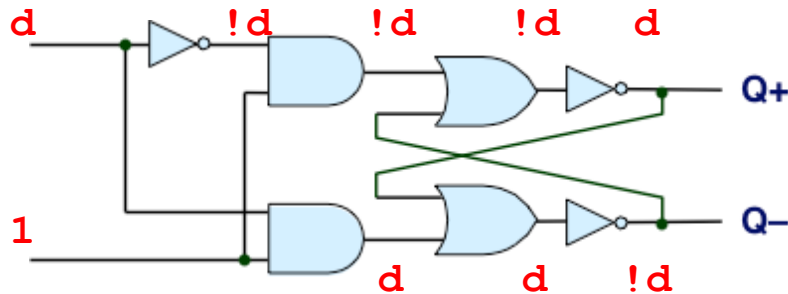
Q+ will continuously change as d changes

Building on top of R-S Latch

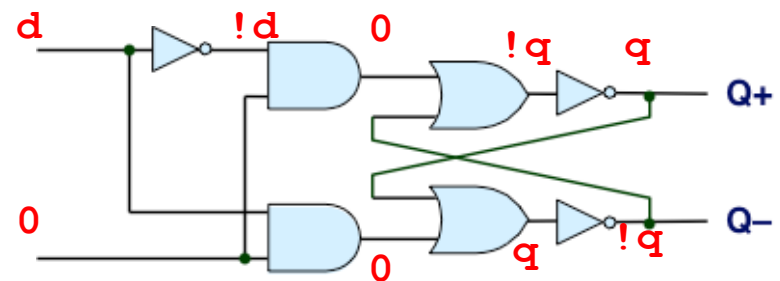


If R and S are different, Q+ is the same as S

Storing Data (Latching)

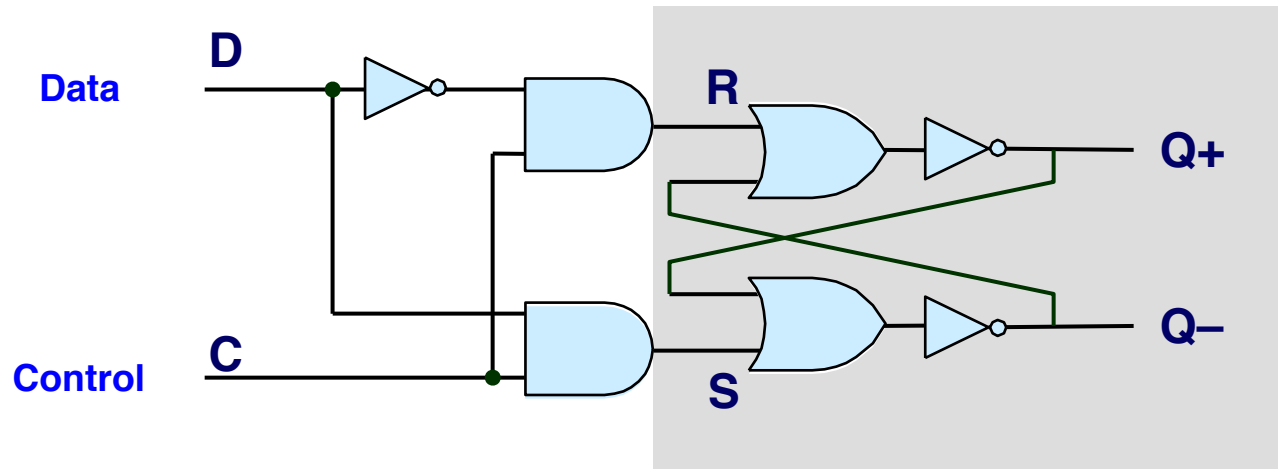


Q+ will continuously change as d changes



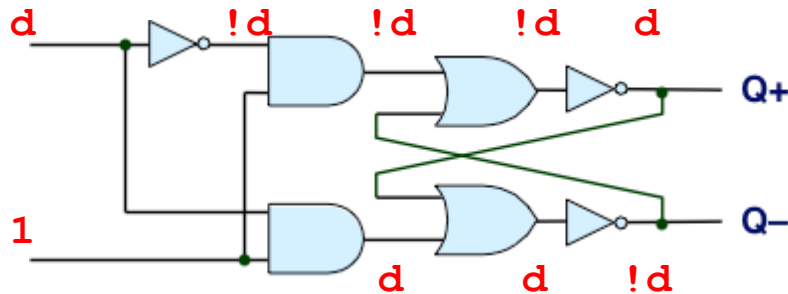
Q+ doesn't change with d

Building on top of R-S Latch



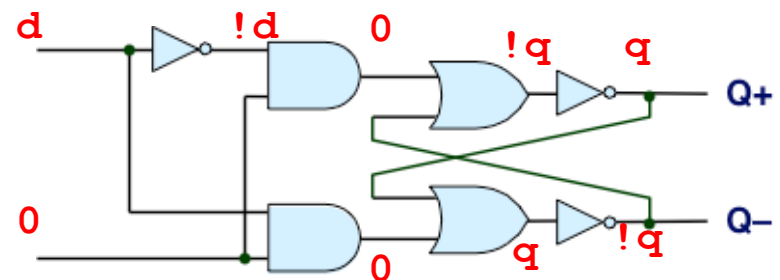
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

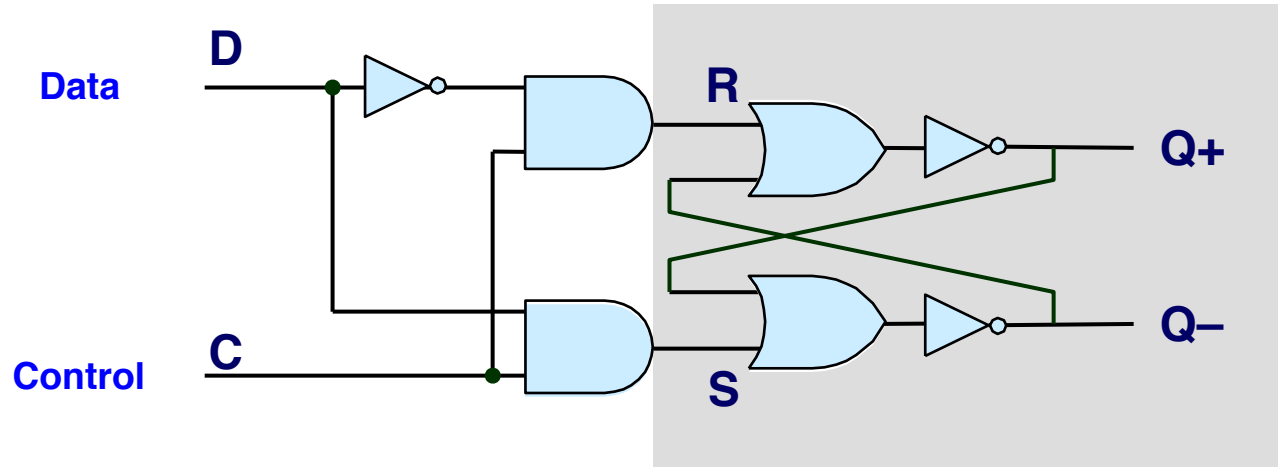
Holding Data



Q+ doesn't change with d

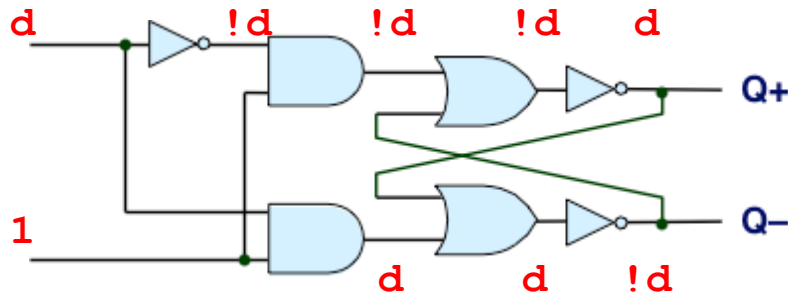
Building on top of R-S Latch

D Latch



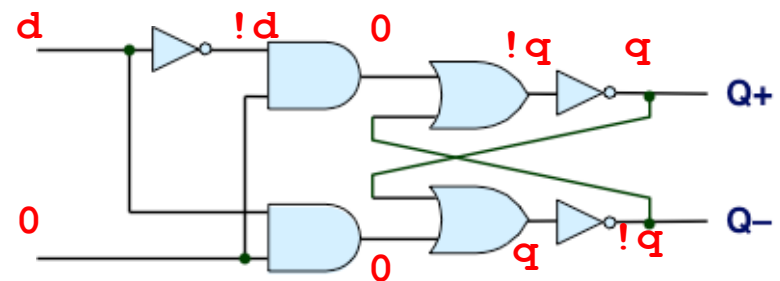
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

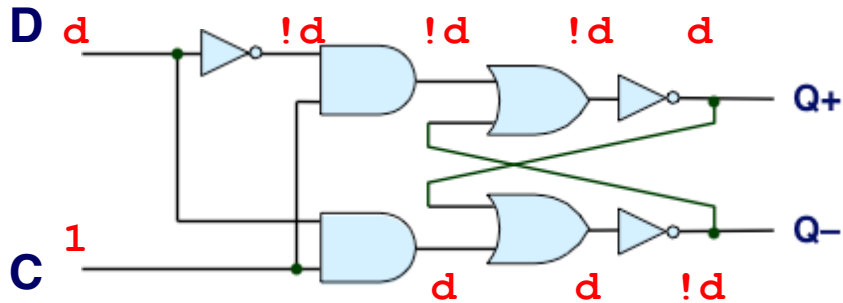
Holding Data



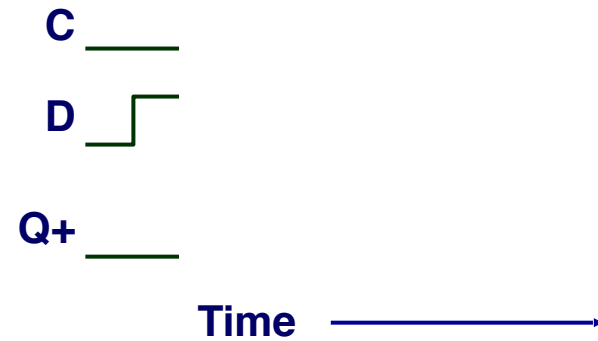
Q+ doesn't change with d

D-Latch is “Transparent”

Latching

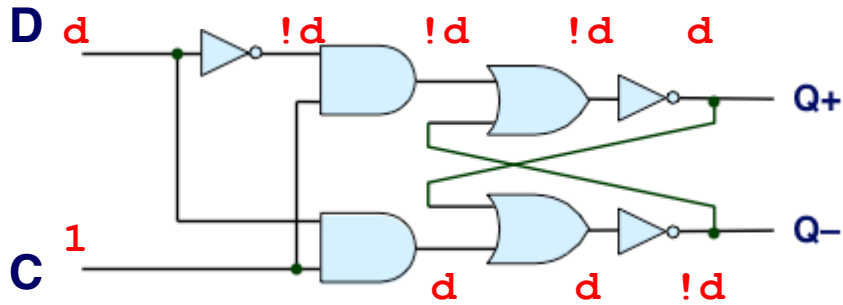


Changing D

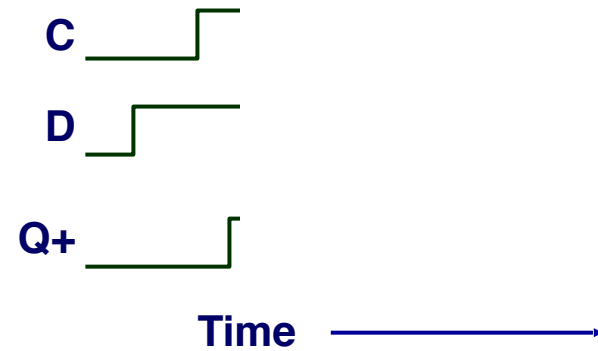


D-Latch is “Transparent”

Latching

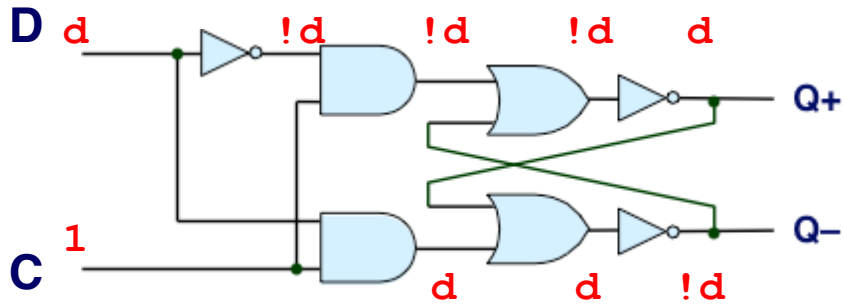


Changing D

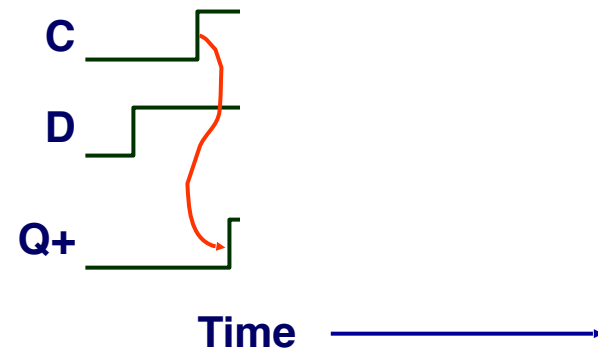


D-Latch is “Transparent”

Latching

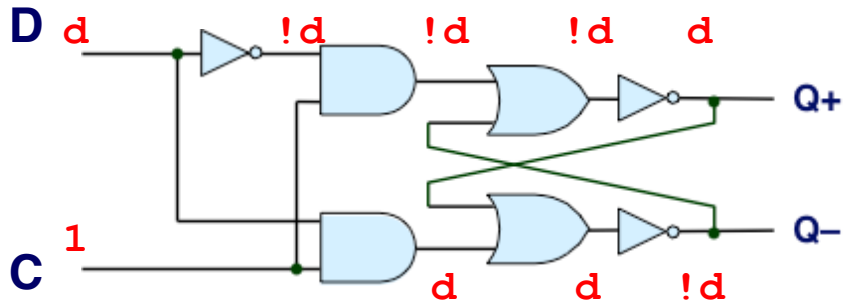


Changing D

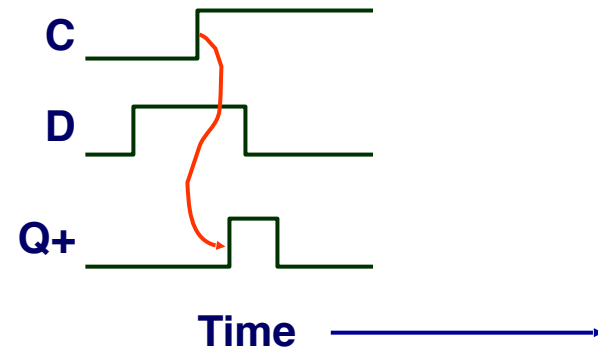


D-Latch is “Transparent”

Latching

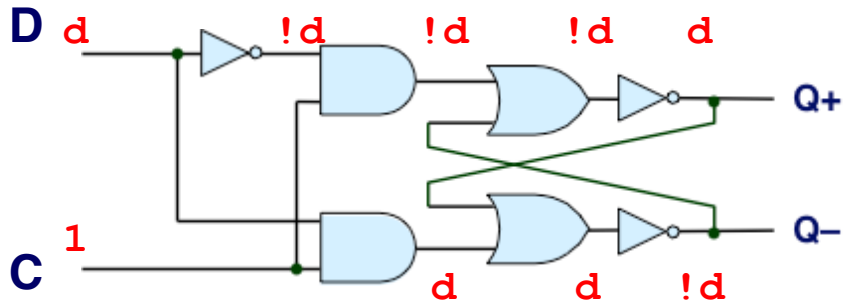


Changing D

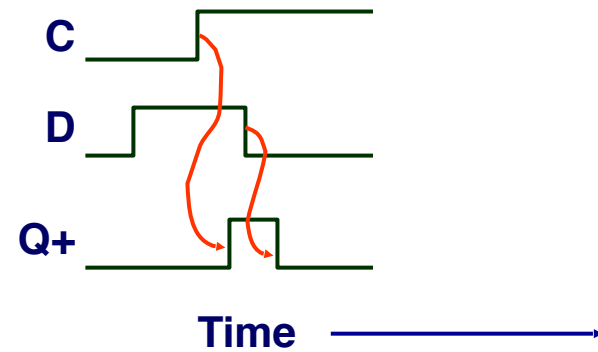


D-Latch is “Transparent”

Latching

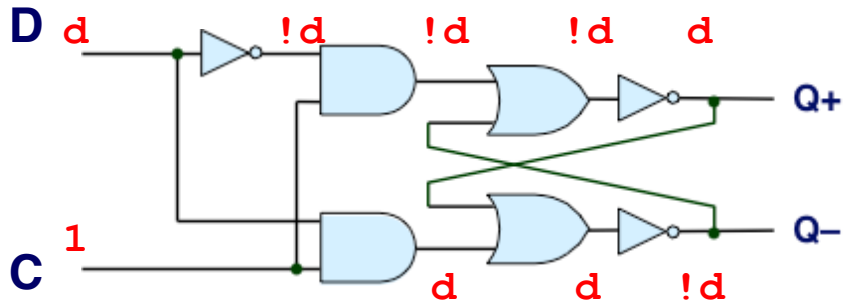


Changing D

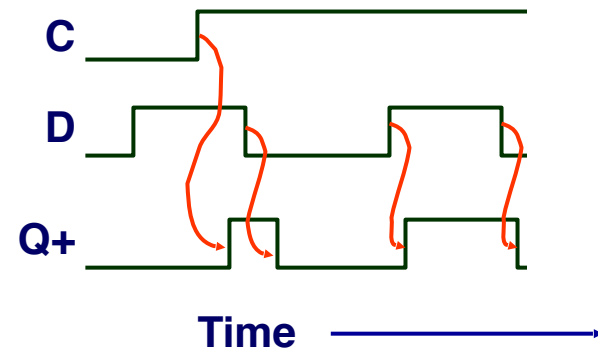


D-Latch is “Transparent”

Latching

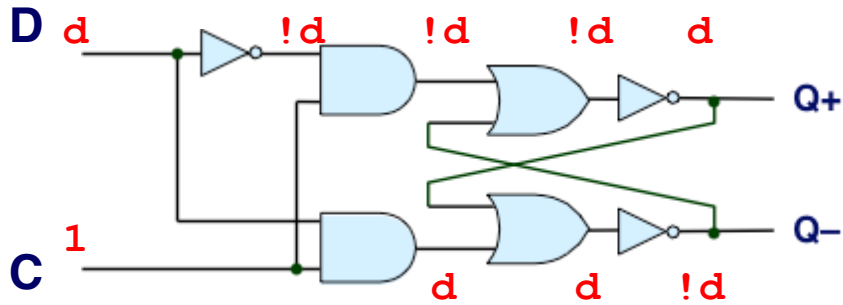


Changing D

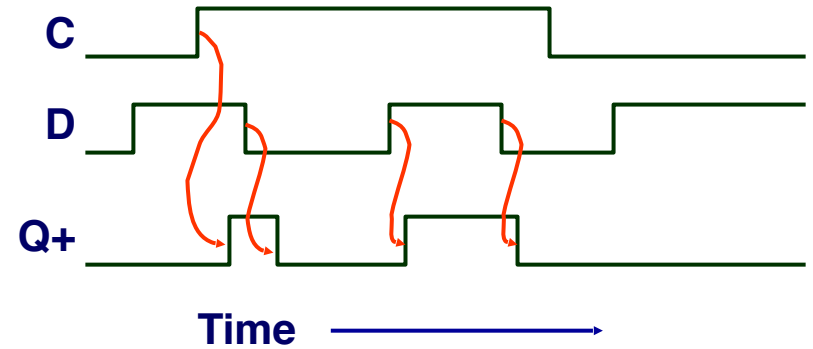


D-Latch is “Transparent”

Latching

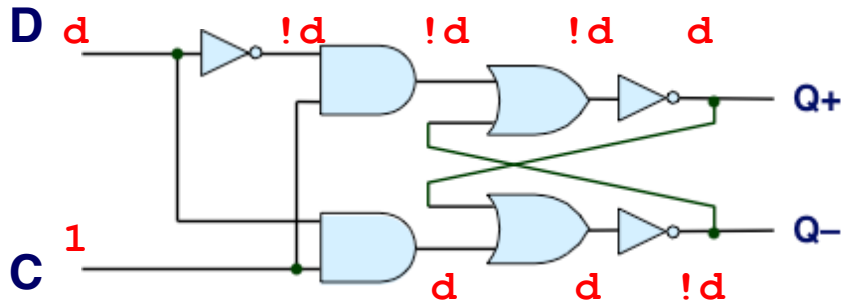


Changing D

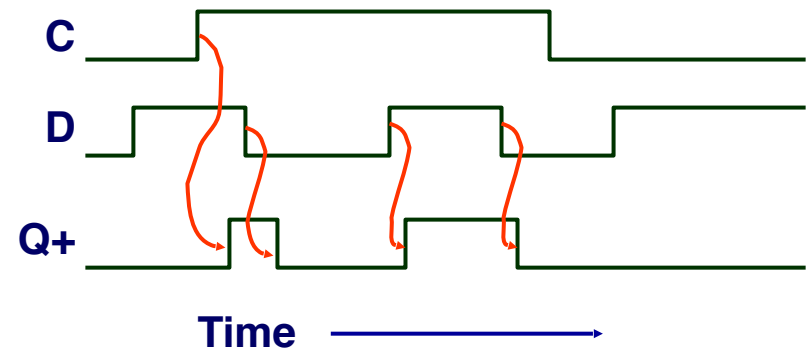


D-Latch is “Transparent”

Latching



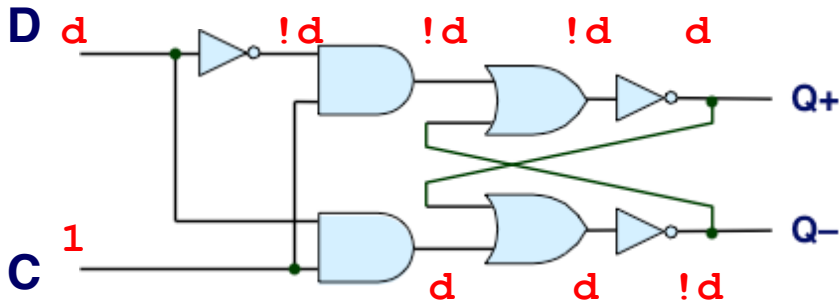
Changing D



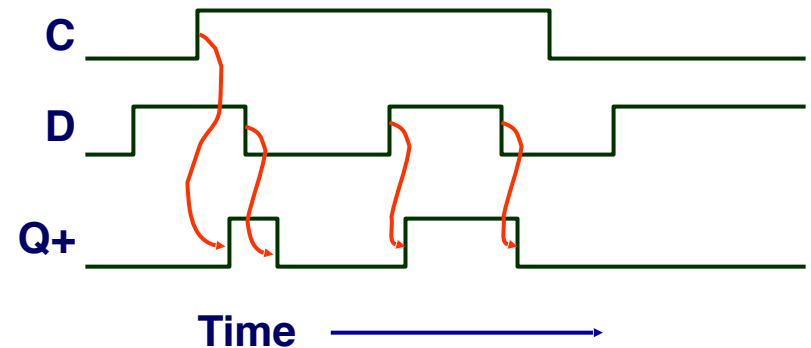
- When you want to store **d**, you have to first set **C** to 1, and then set **d**

D-Latch is “Transparent”

Latching



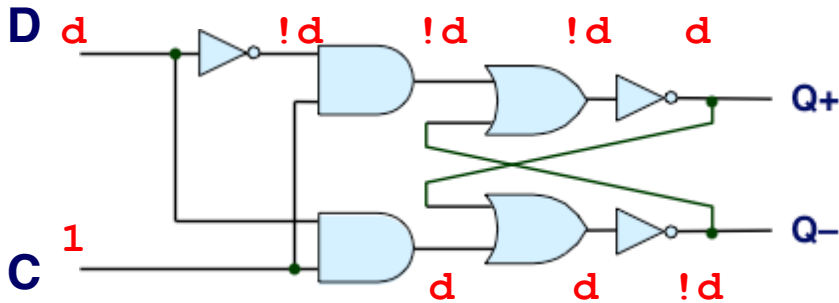
Changing D



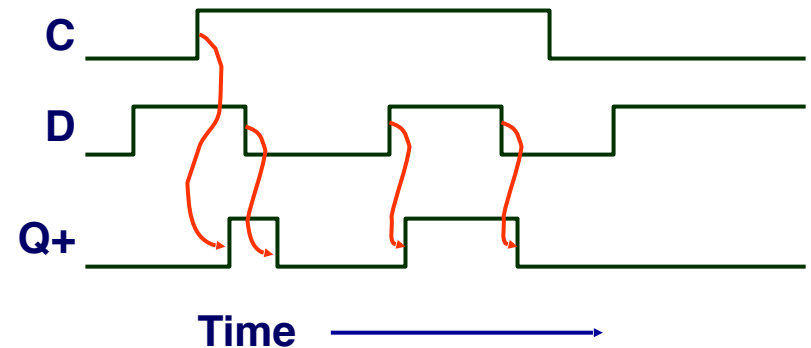
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold **C** for a while until the signal is fully propagated

D-Latch is “Transparent”

Latching



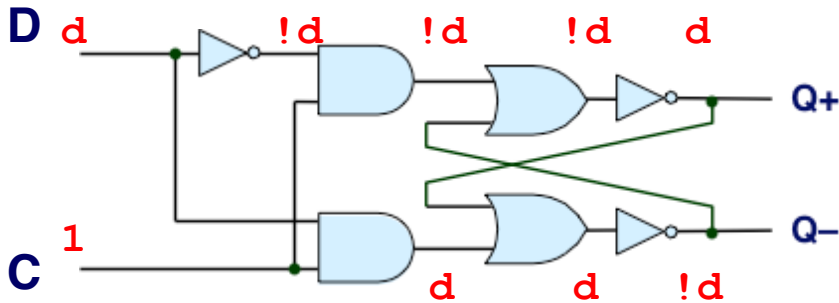
Changing D



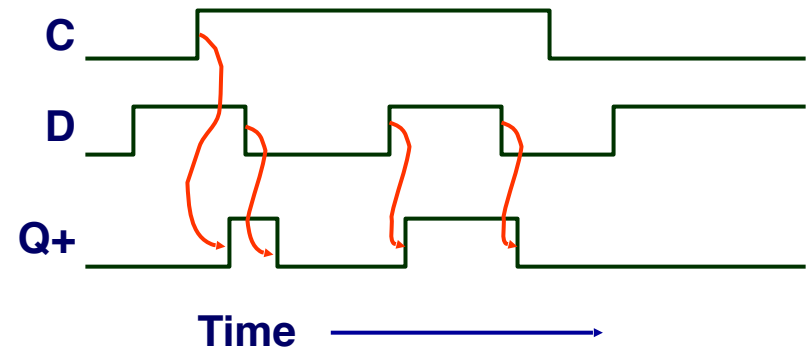
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold **C** for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **C** goes to 0

D-Latch is “Transparent”

Latching



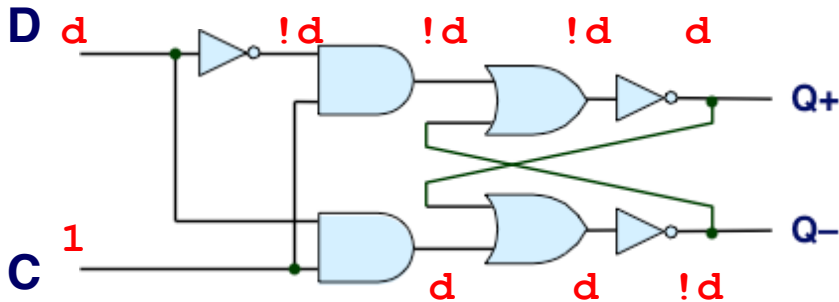
Changing D



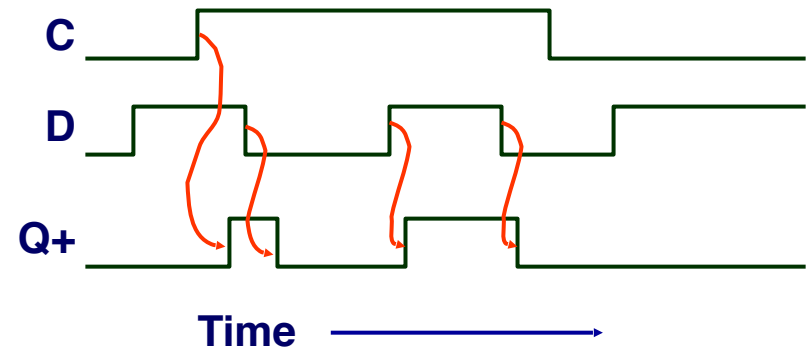
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold C for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as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

D-Latch is “Transparent”

Latching

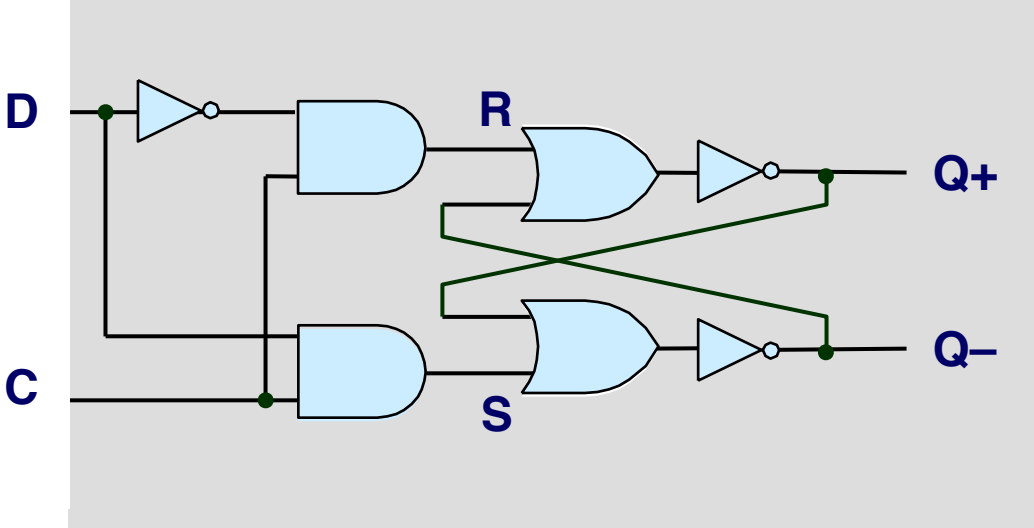


Changing D

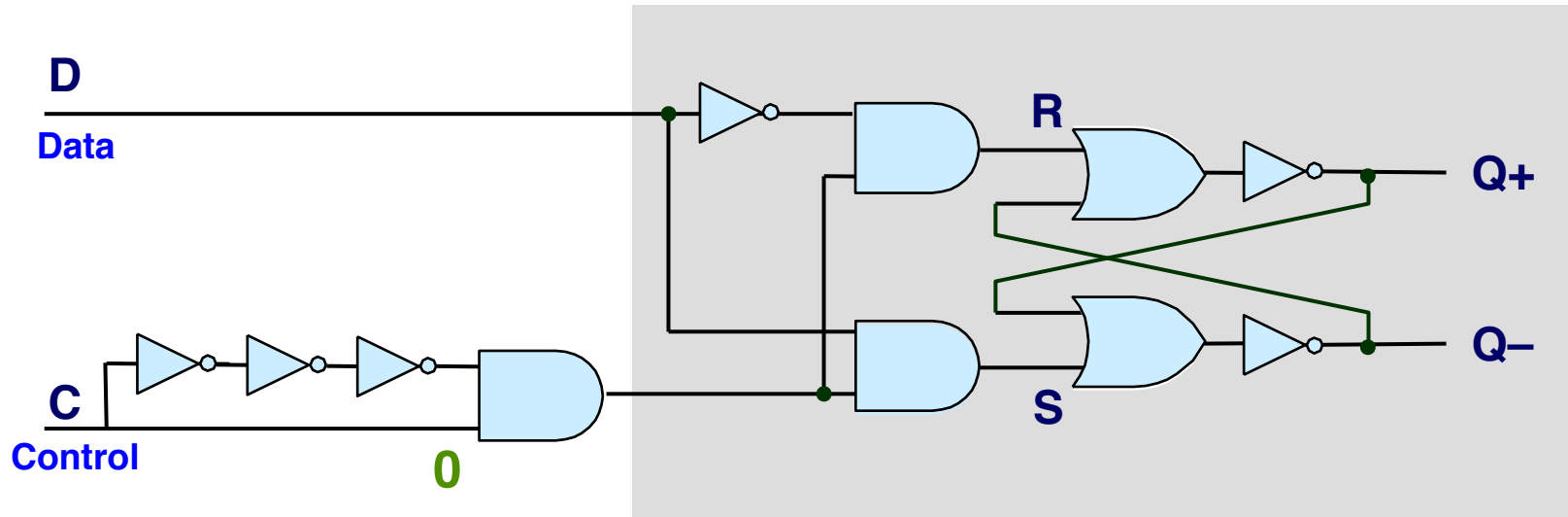


- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold C for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **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.

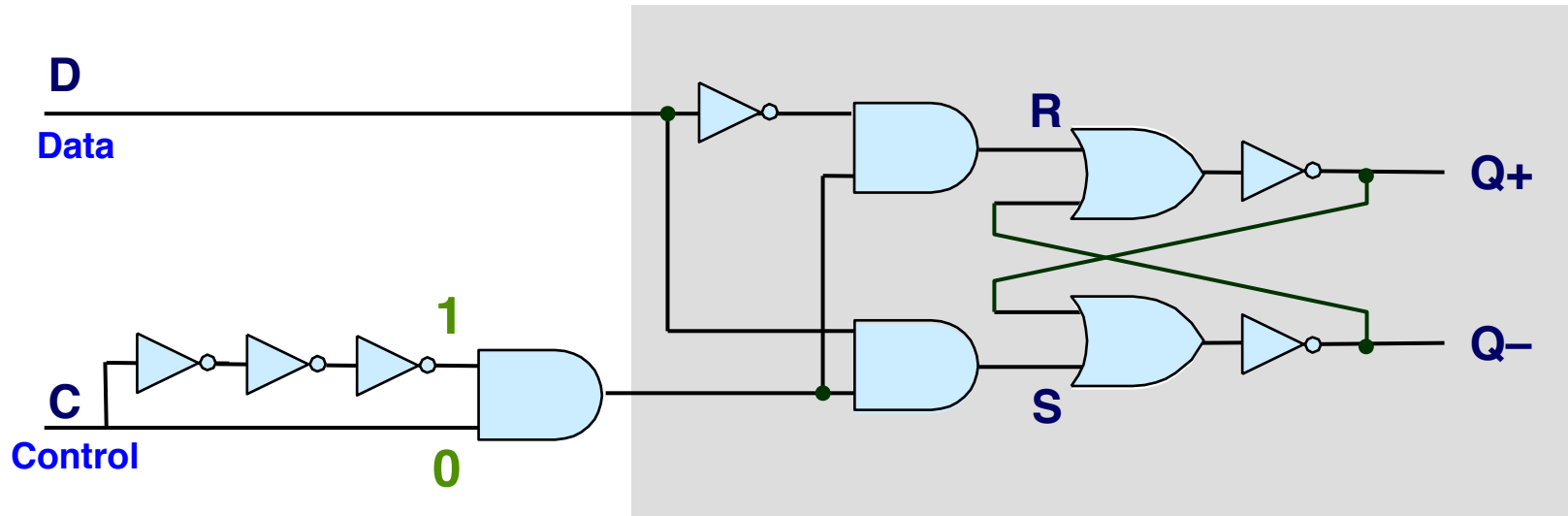
Edge-Triggered Latch (Flip-Flop)



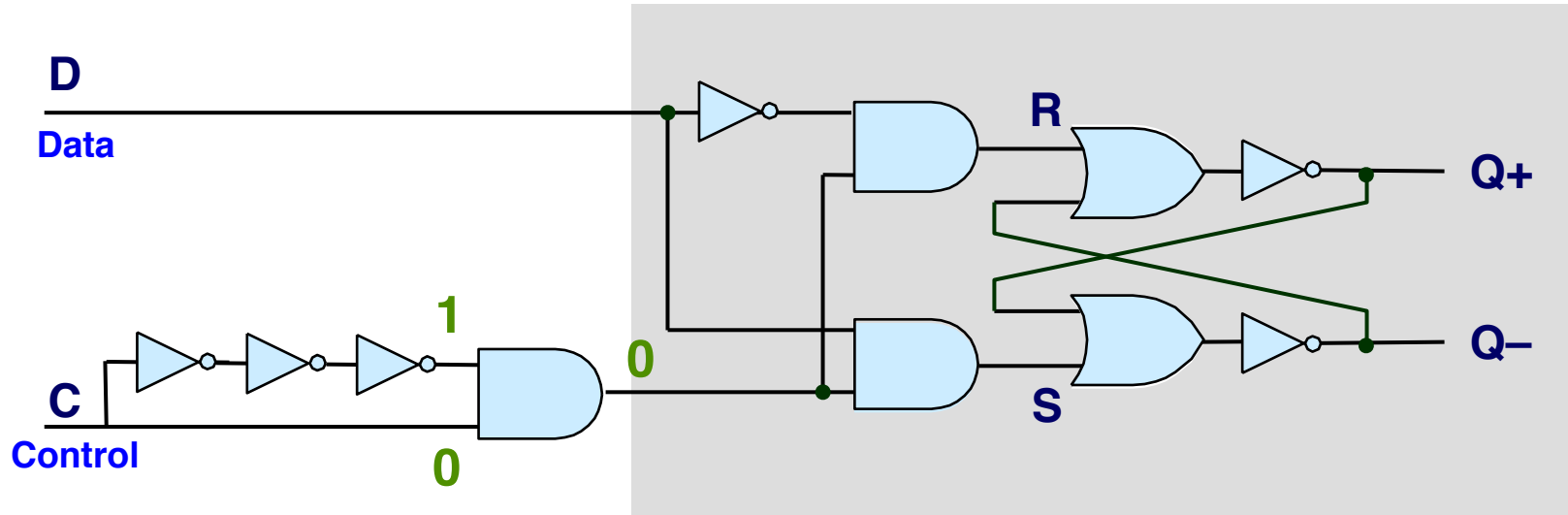
Edge-Triggered Latch (Flip-Flop)



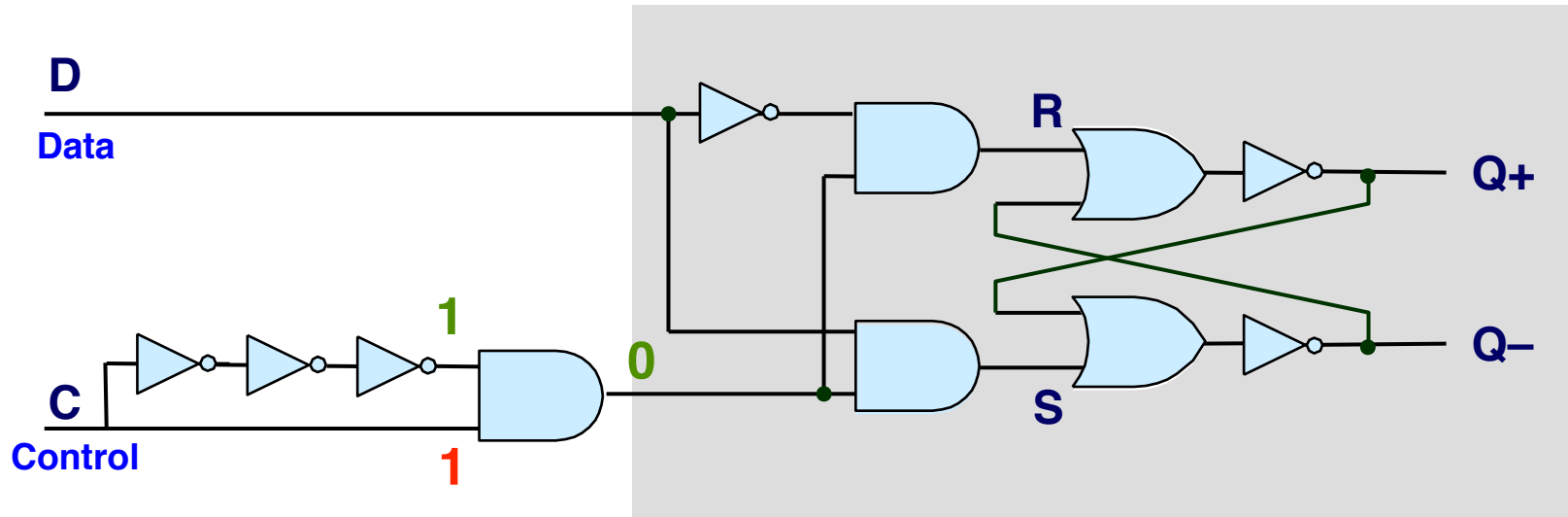
Edge-Triggered Latch (Flip-Flop)



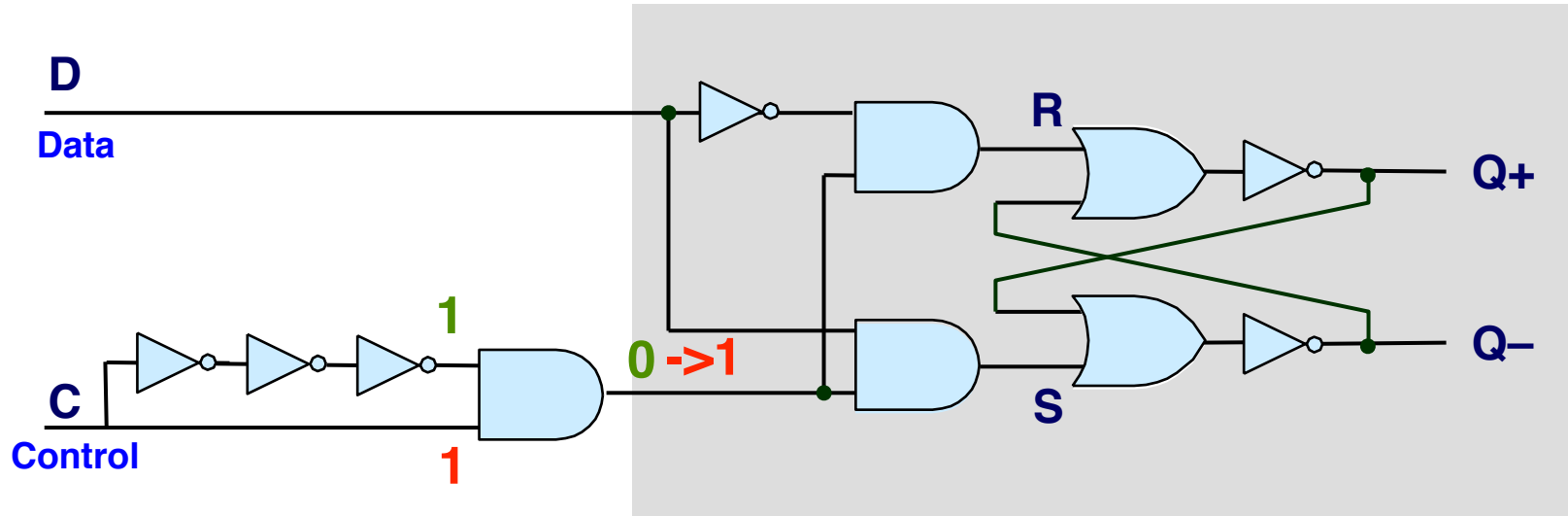
Edge-Triggered Latch (Flip-Flop)



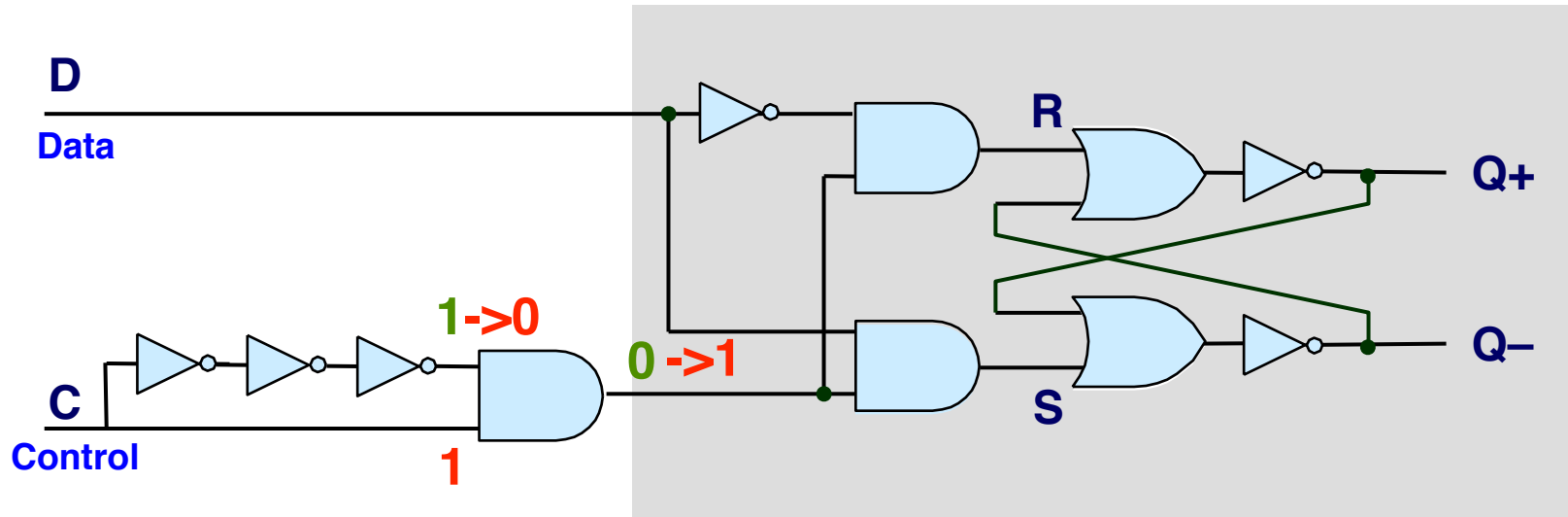
Edge-Triggered Latch (Flip-Flop)



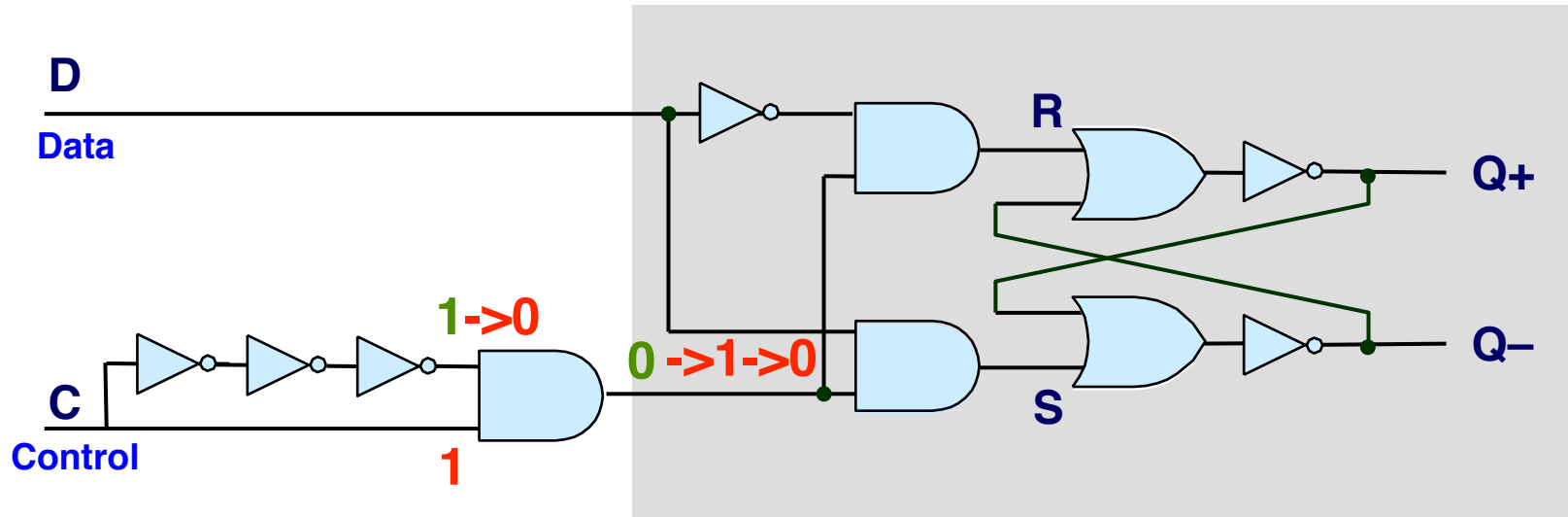
Edge-Triggered Latch (Flip-Flop)



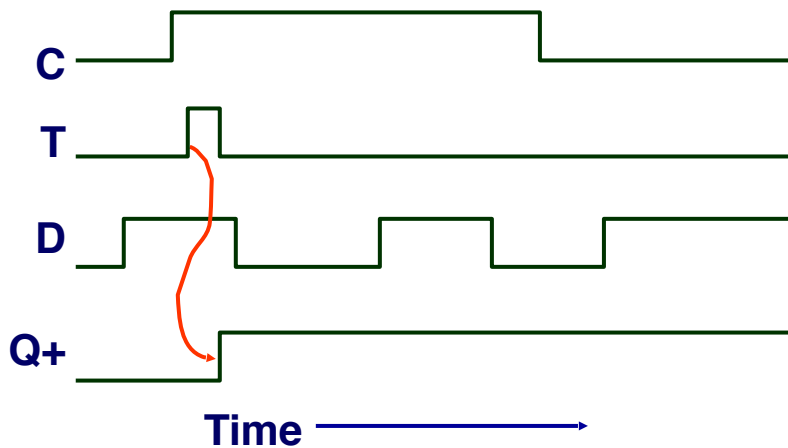
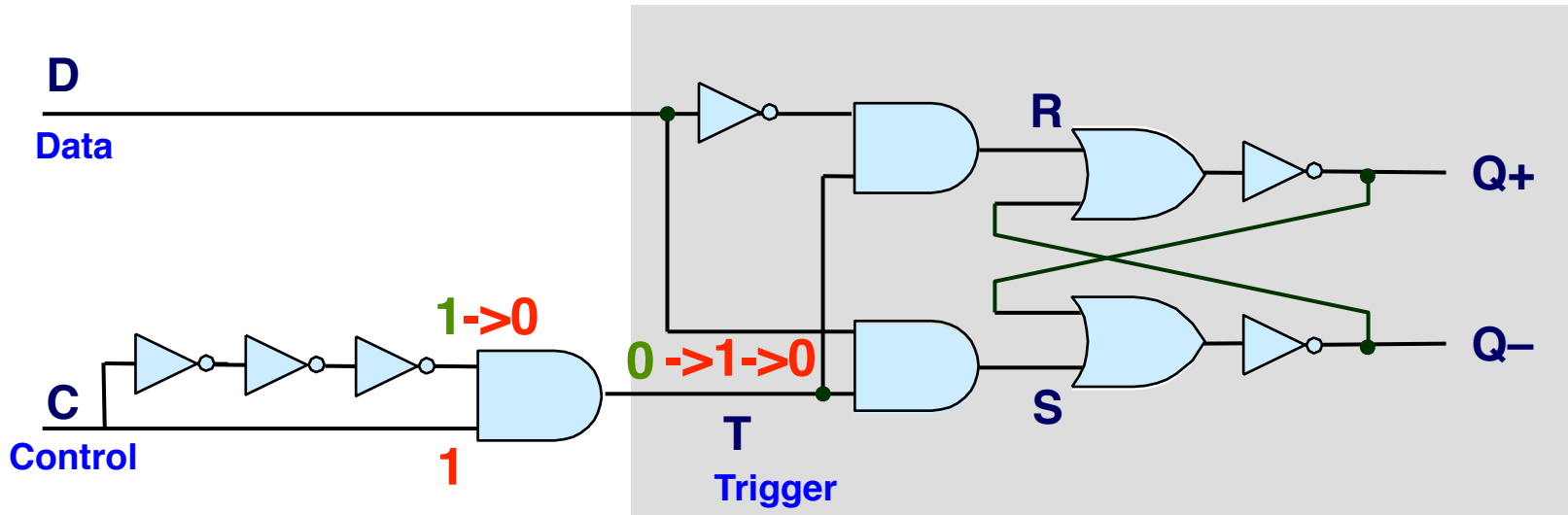
Edge-Triggered Latch (Flip-Flop)



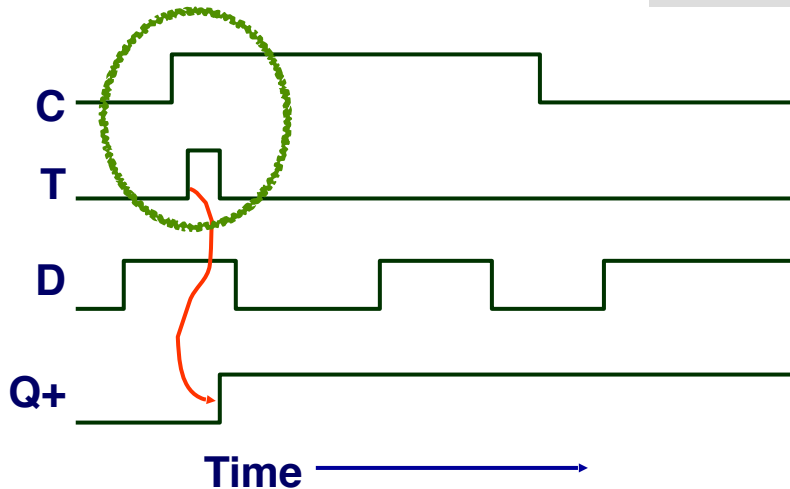
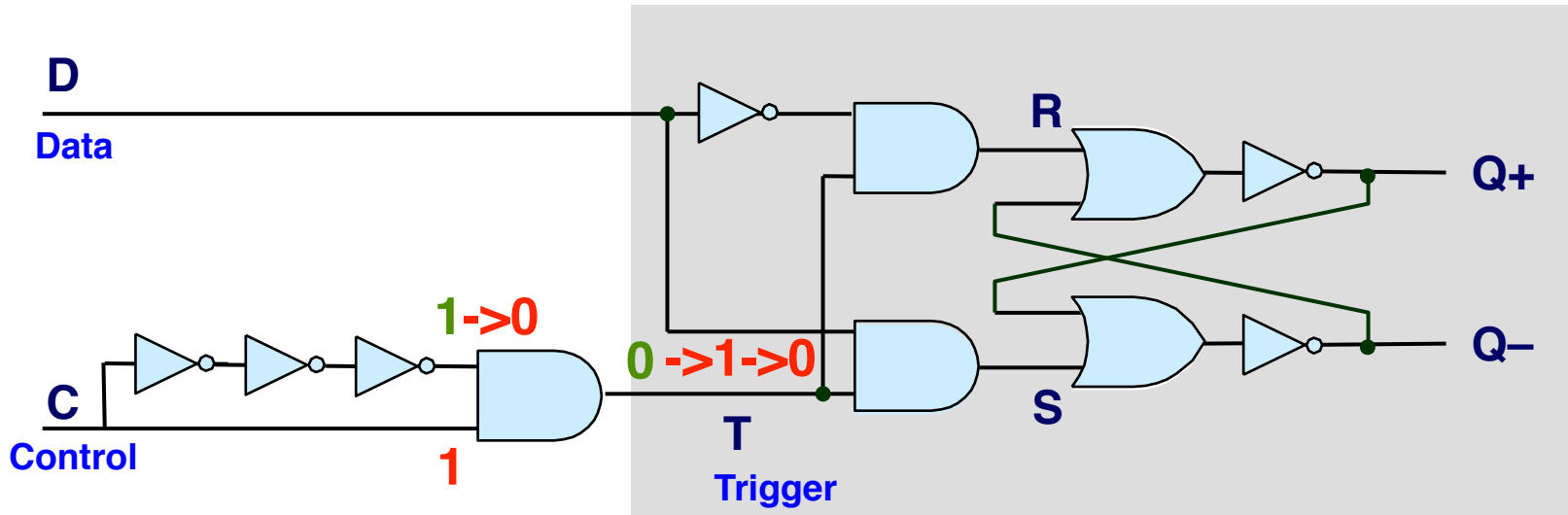
Edge-Triggered Latch (Flip-Flop)



Edge-Triggered Latch (Flip-Flop)

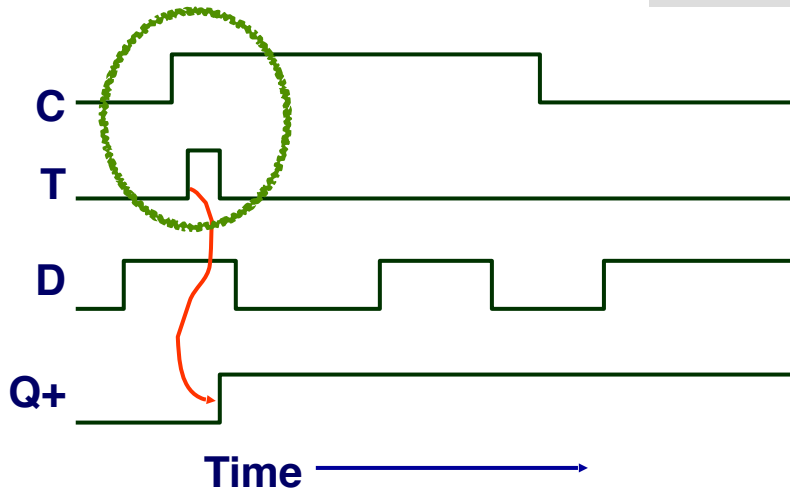
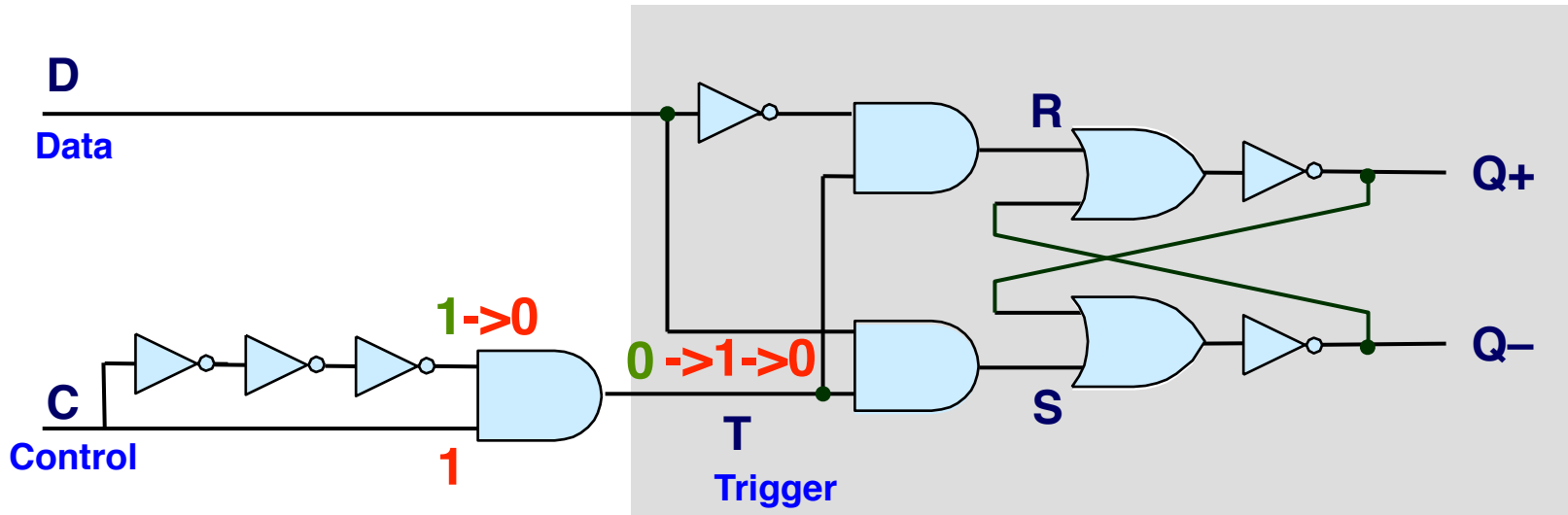


Edge-Triggered Latch (Flip-Flop)



- 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

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