CSC 252: Computer Organization
Spring 2022: Lecture 11

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Programming assignment 3 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment3.html
  - Due on **March 3**, 11:59 PM
  - You (may still) have 3 slip days

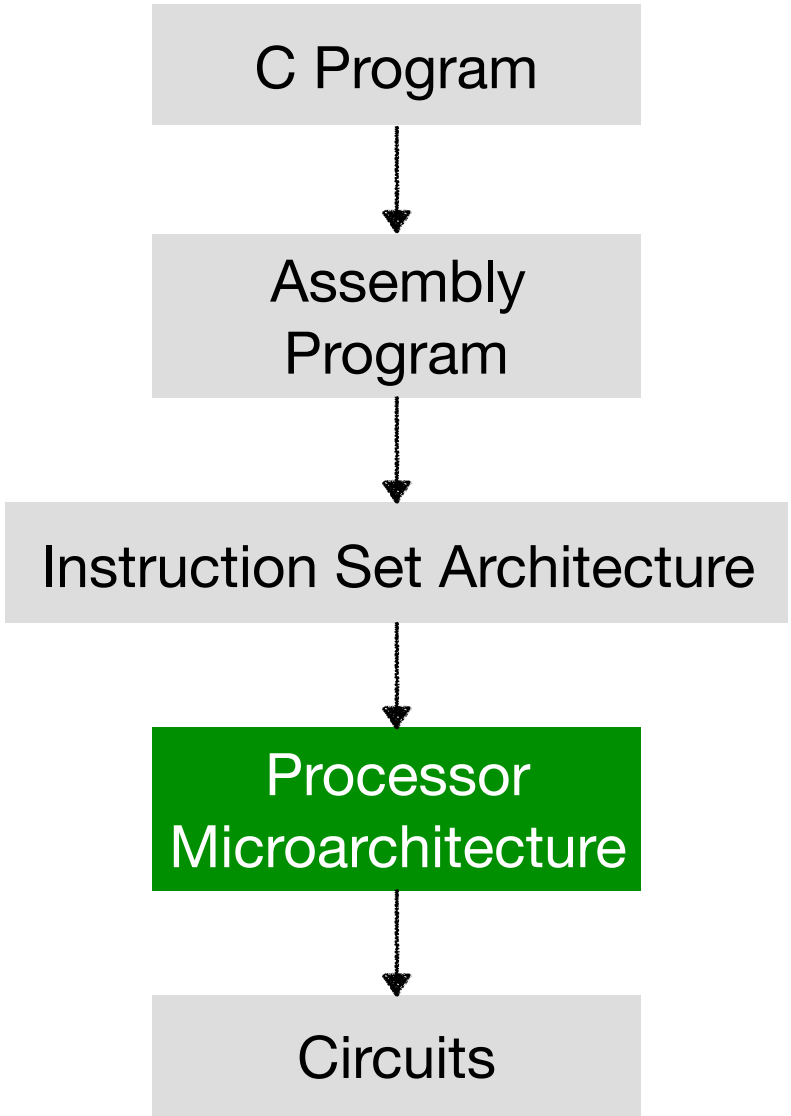
13	14	15	16	17 **Today**	18	19
20	21	22	23	24	25	26
27	28	Mar 1	2	3 **Due** **Mid-term**	4	5

Announcements

- Grades for Lab 1 are posted.
- We are processing regrading requests
- Will grade Lab 2 soon.

- Programming assignment 3 is in x86 assembly language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

So far in 252…

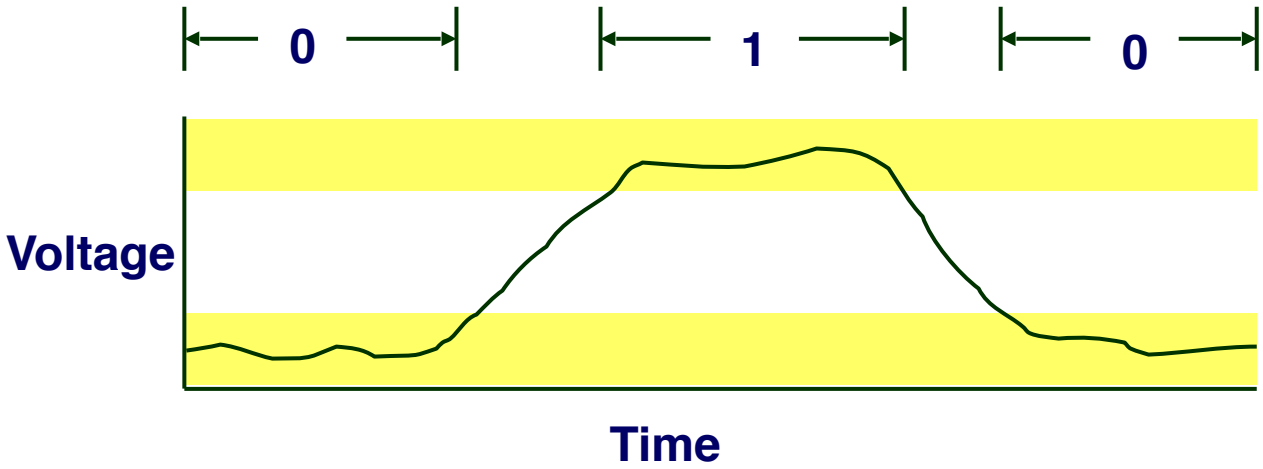

C Program
Assembly Program
Instruction Set Architecture
Processor Microarchitecture
Circuits

Today: Circuits Basics

- Basics
- Circuits for computations
- Circuits for storing data

Overview of Circuit-Level Design

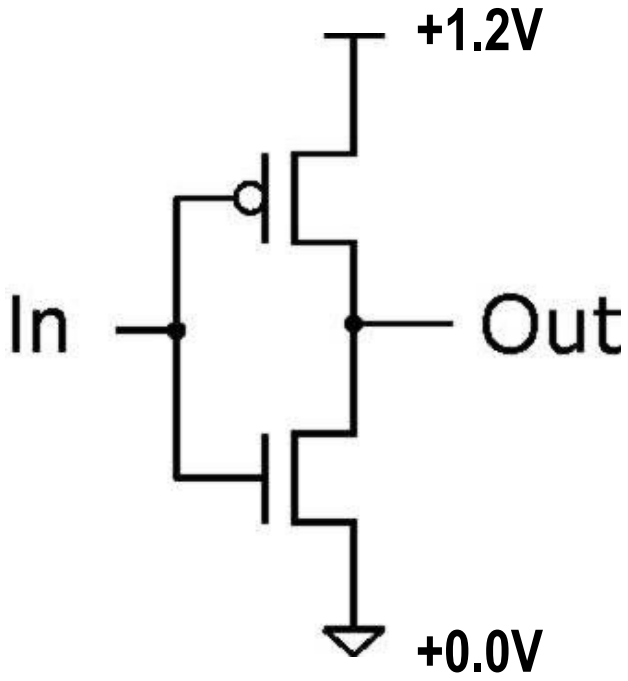
- Fundamental Hardware Requirements
  - Communication: How to get values from one place to another. Mainly three electrical **wires**.
  - Computation: **transistors**. Combinational logic.
  - Storage: **transistors**. Sequential logic.
- Circuit design is often abstracted as logic design



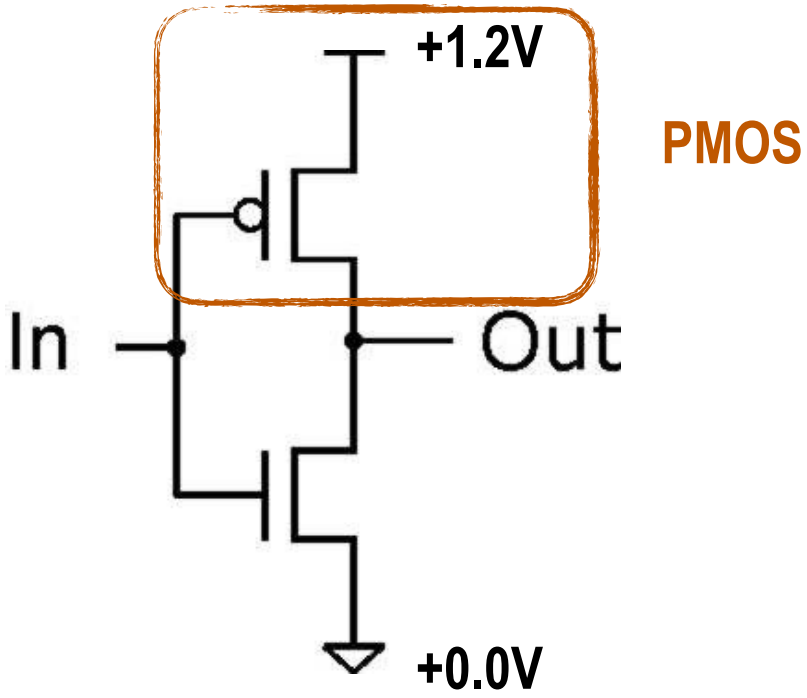
Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

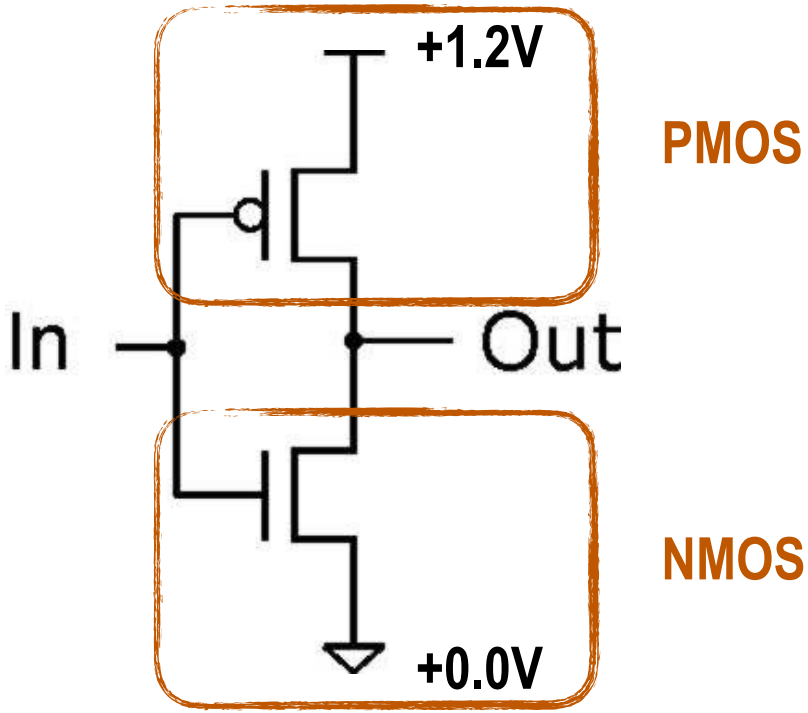
Inverter (NOT Gate)



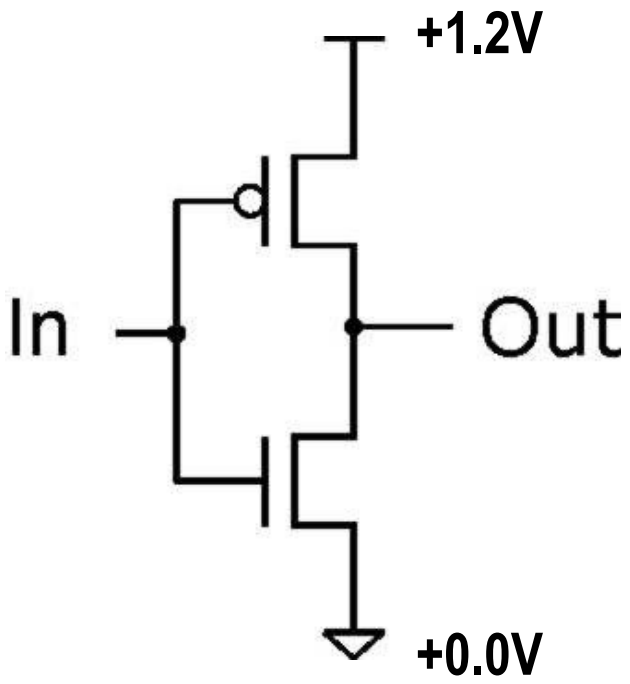
Inverter (NOT Gate)

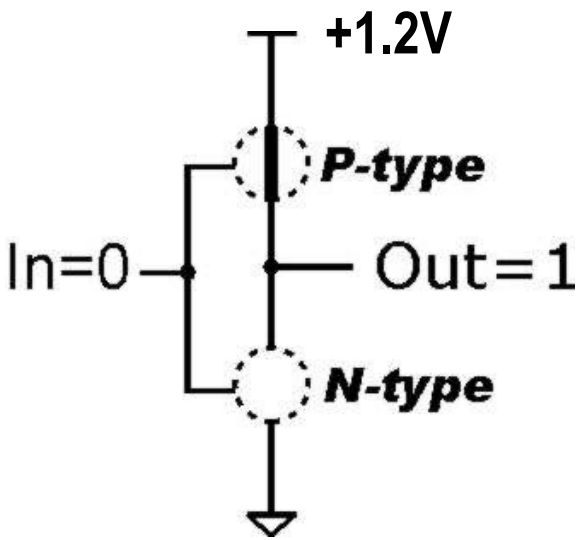


Inverter (NOT Gate)

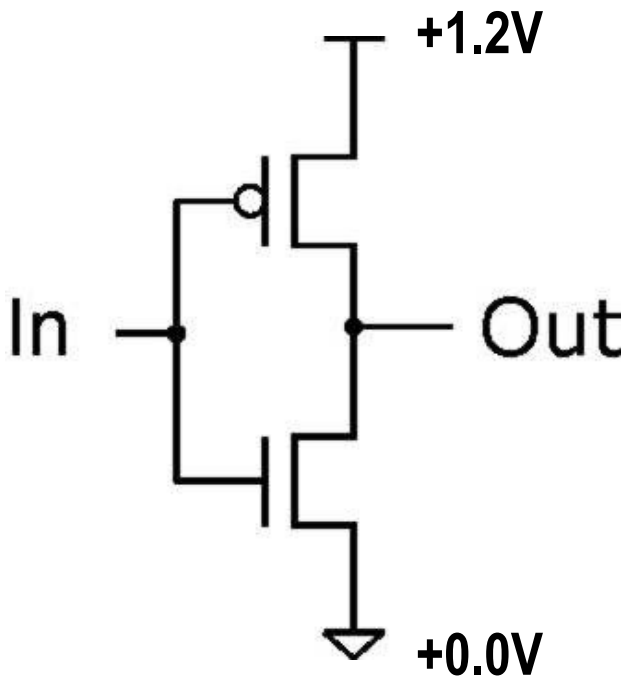


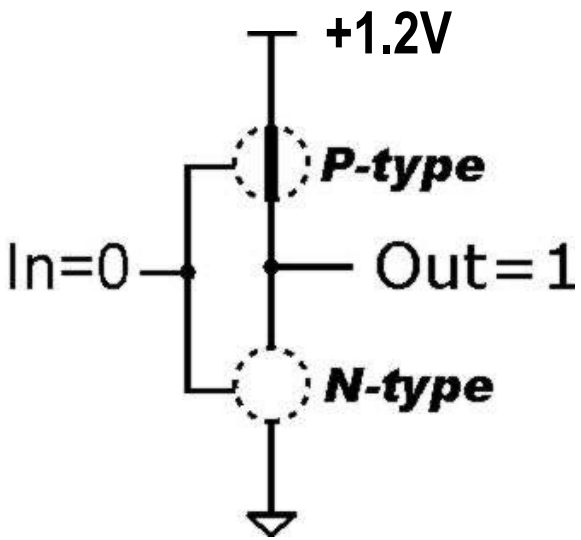
Inverter (NOT Gate)

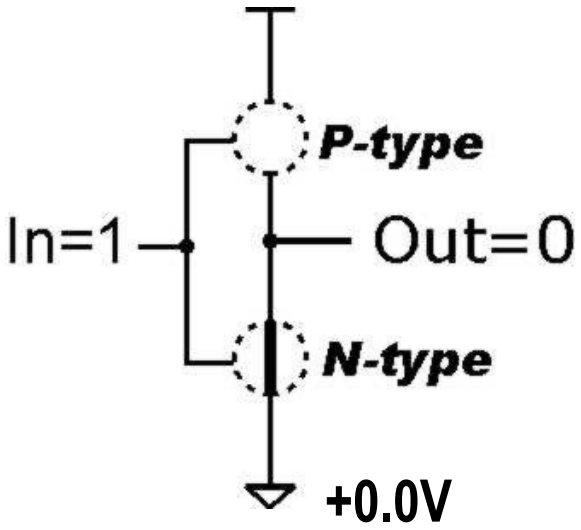




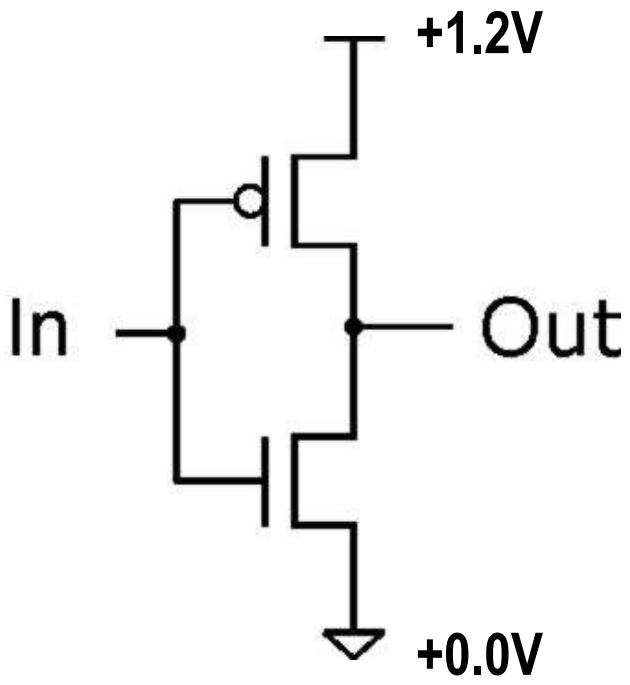
Inverter (NOT Gate)



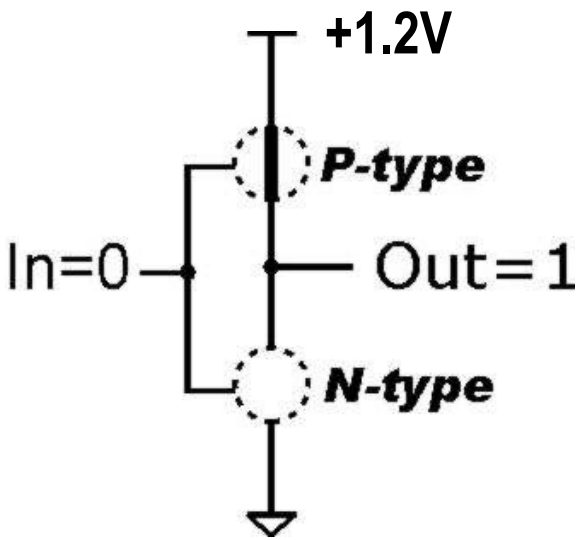


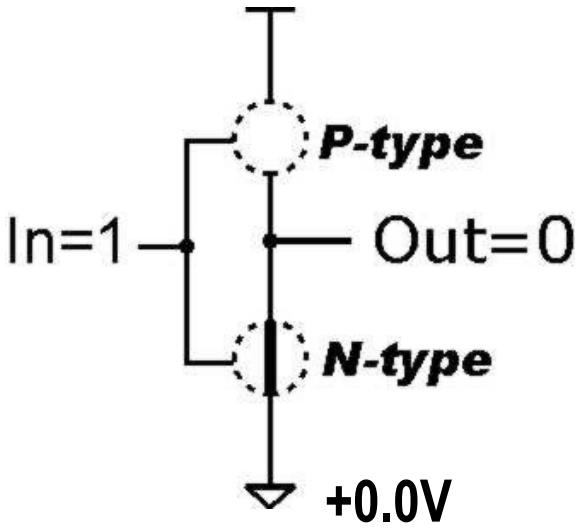


Inverter (NOT Gate)

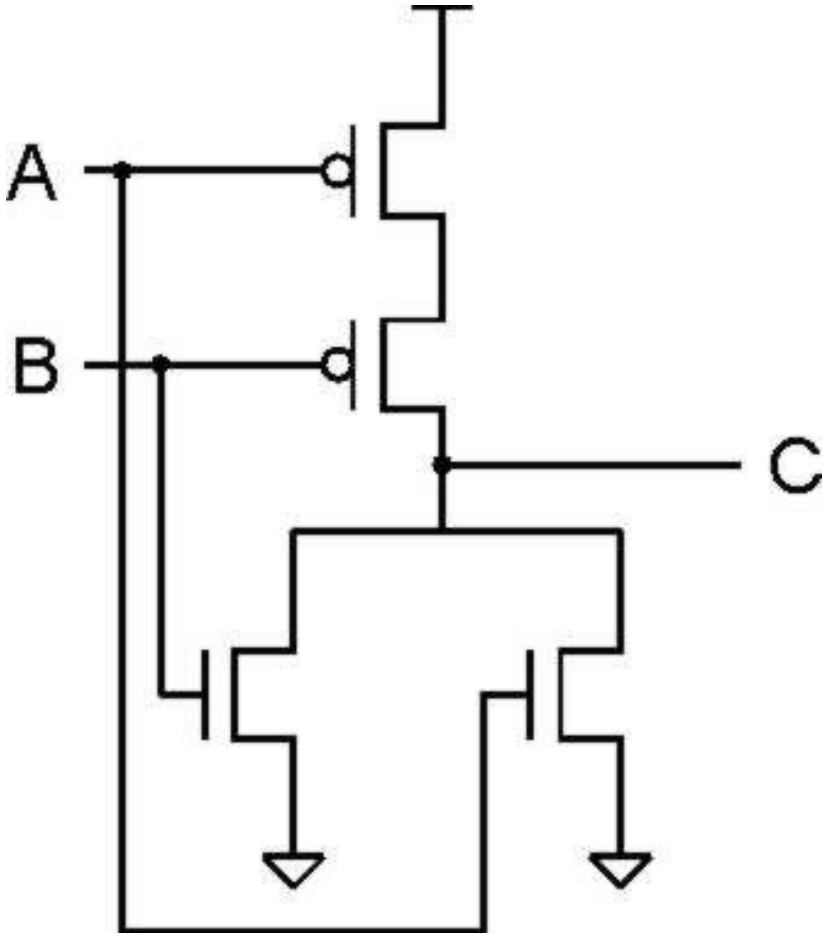


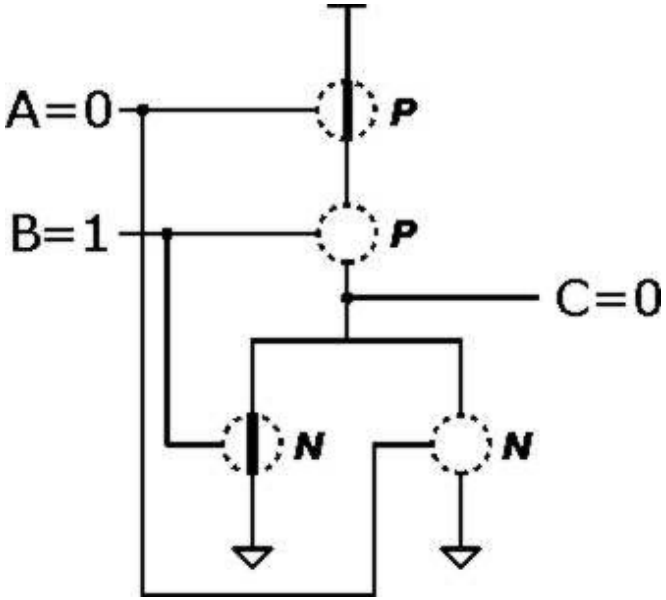
In	Out
0	1
1	0





NOR Gate (NOT + OR)

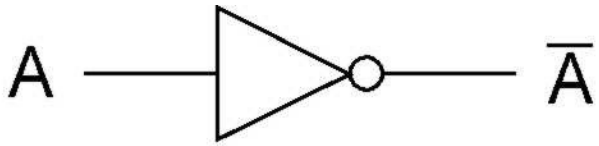




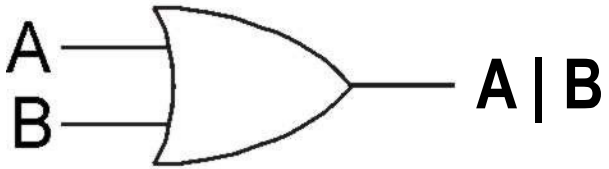
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

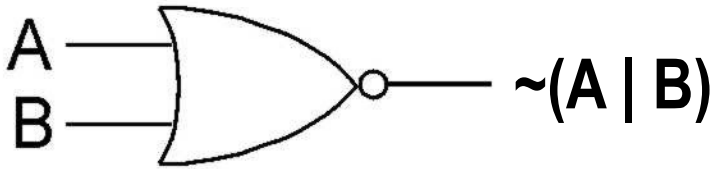
Basic Logic Gates



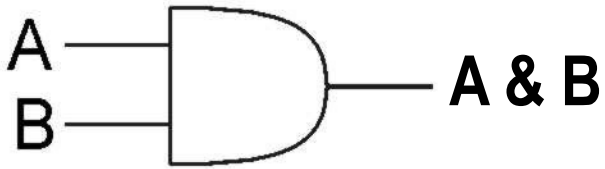
NOT



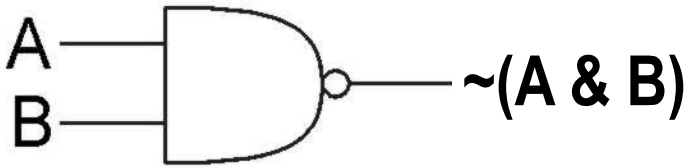
OR



NOR

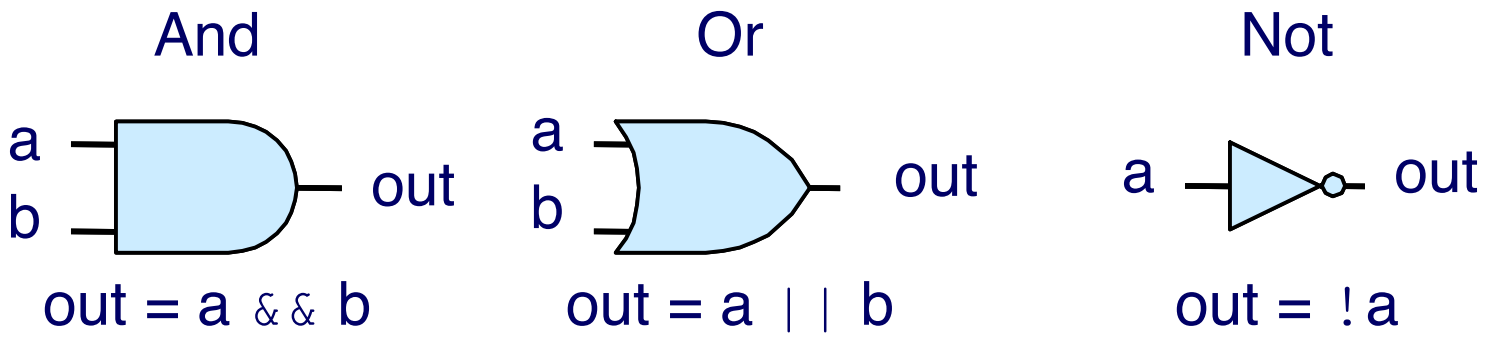


AND

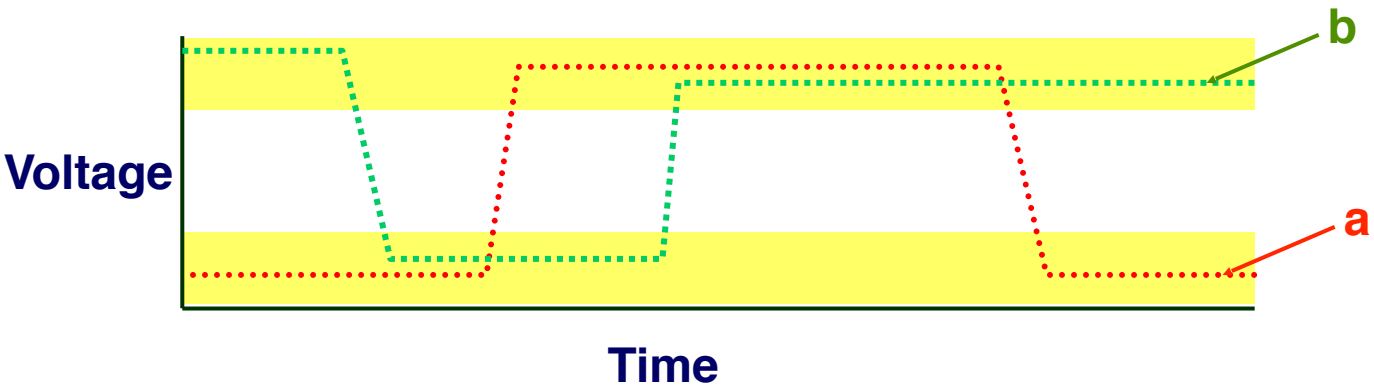


NAND

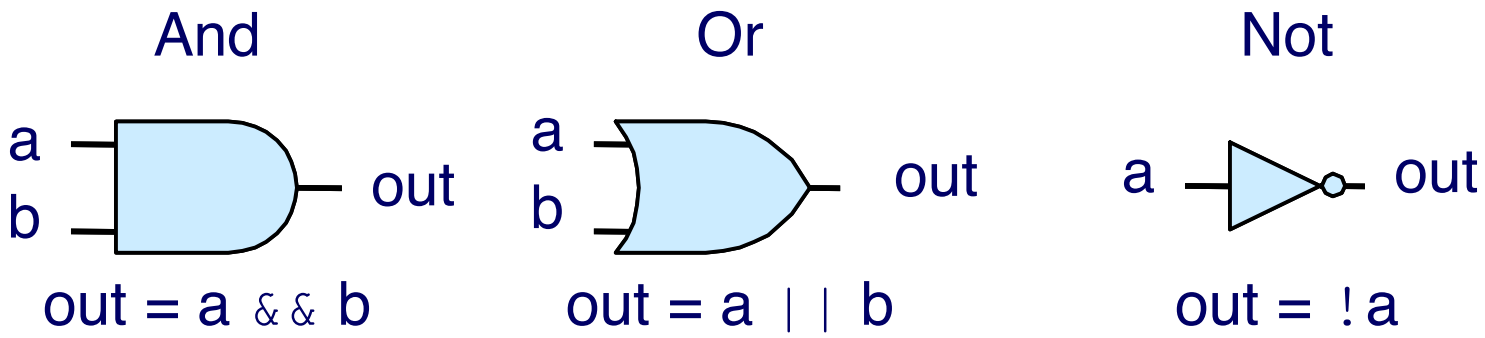
Computing with Logic Gates



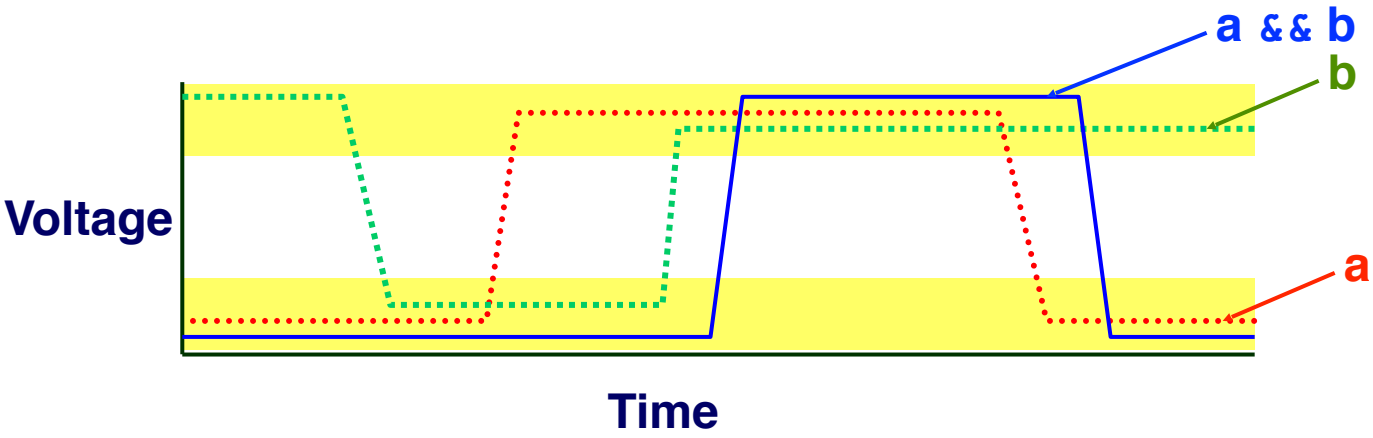
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



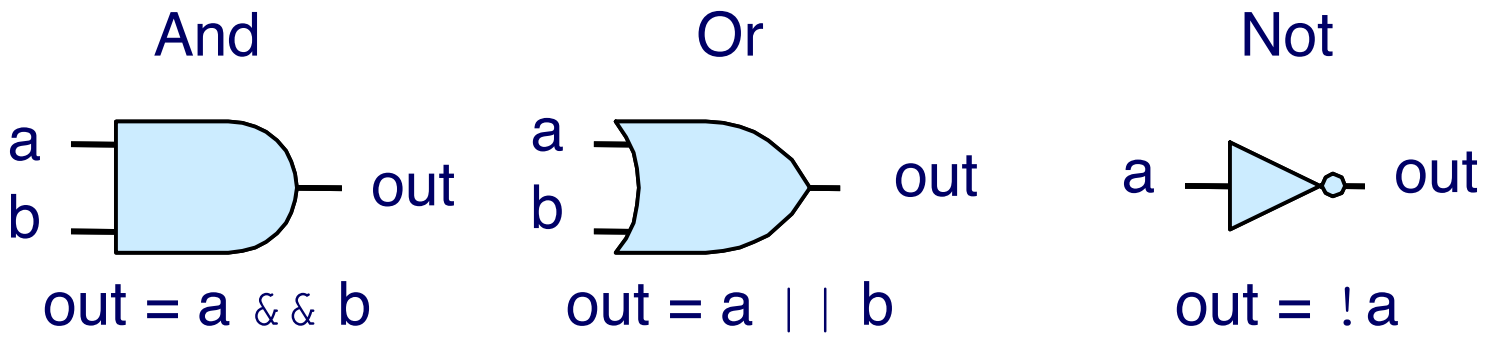
Computing with Logic Gates



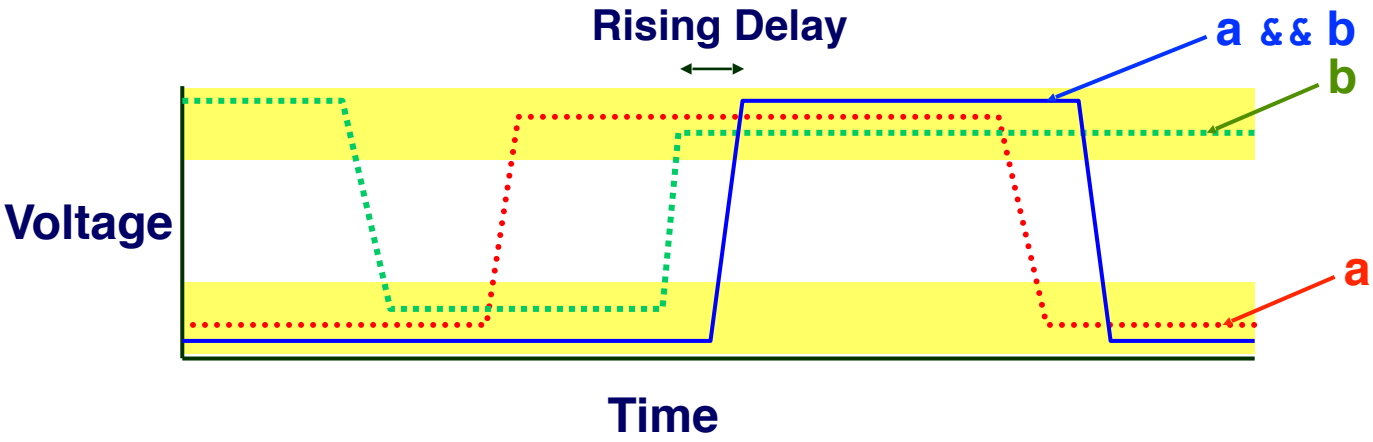
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



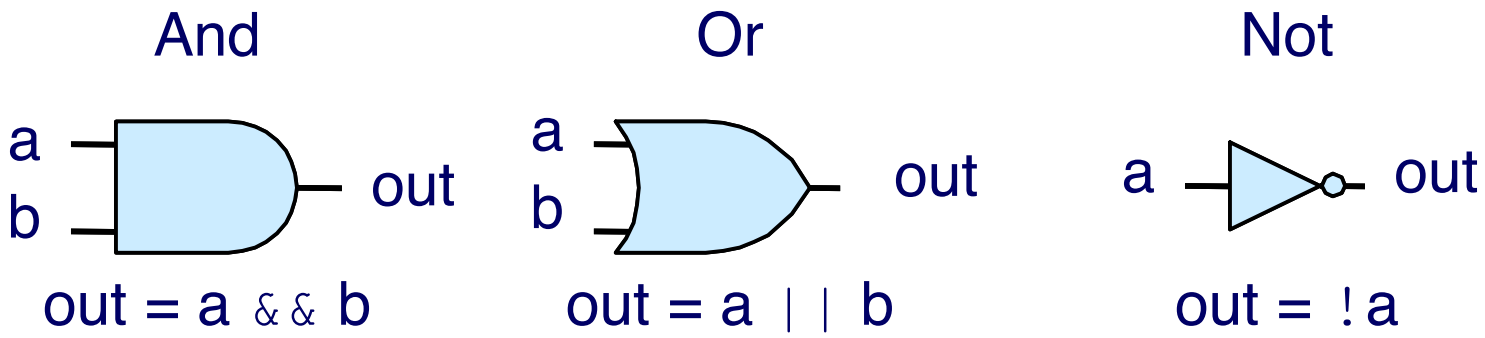
Computing with Logic Gates



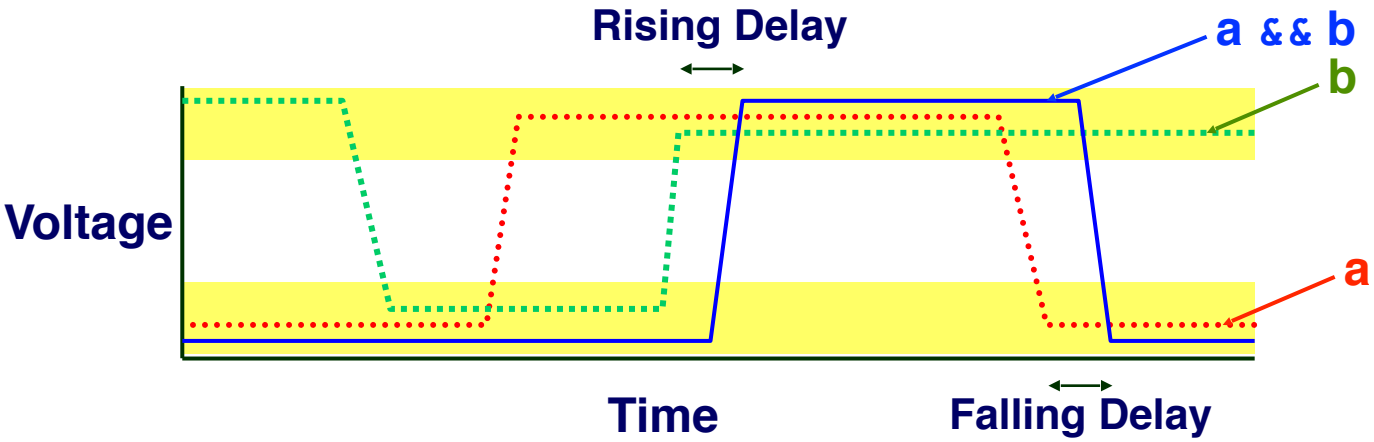
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



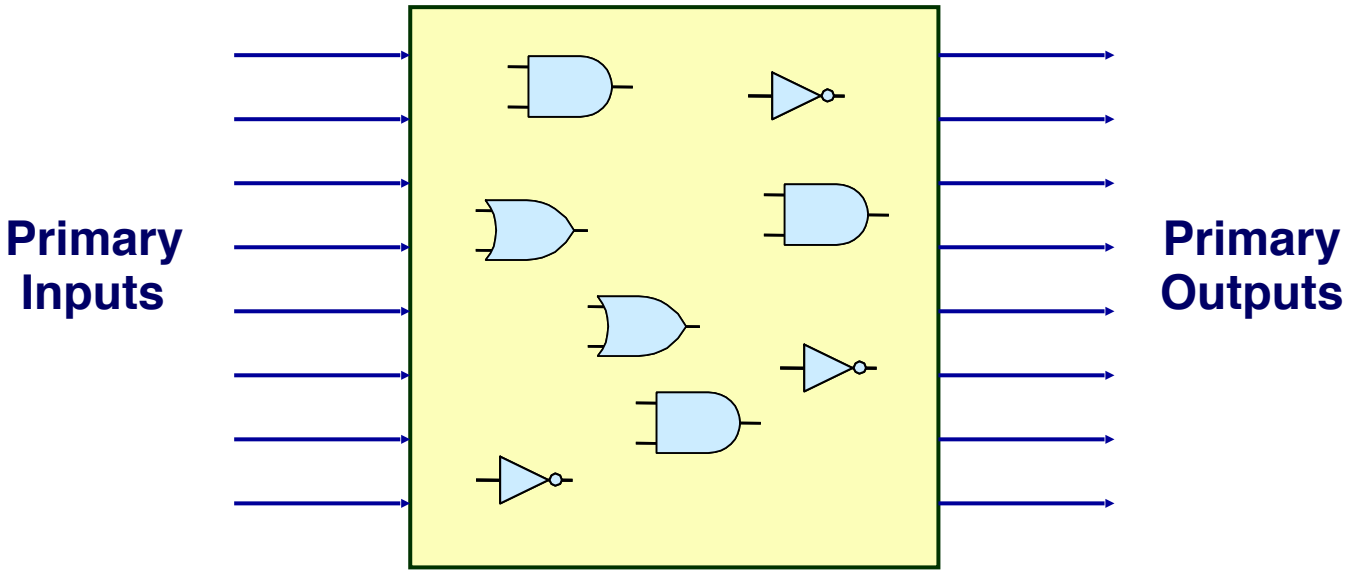
Computing with Logic Gates



- 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)



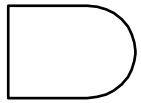
Combinational Circuits



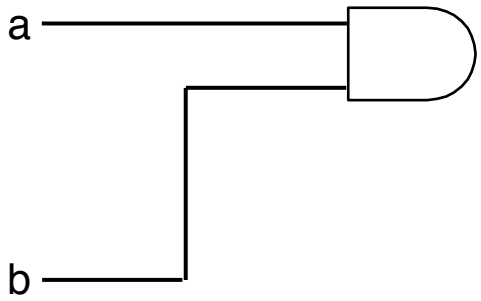
- A Network of Logic Gates
  - Continuously responds to changes on primary inputs
  - Primary outputs become (**after some delay**) Boolean functions of primary inputs

Bit Equality

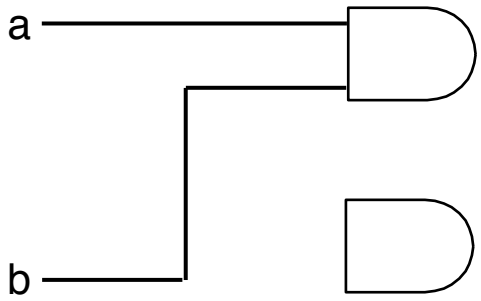
Bit Equality



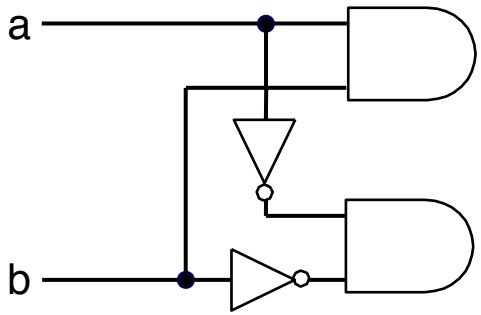
Bit Equality



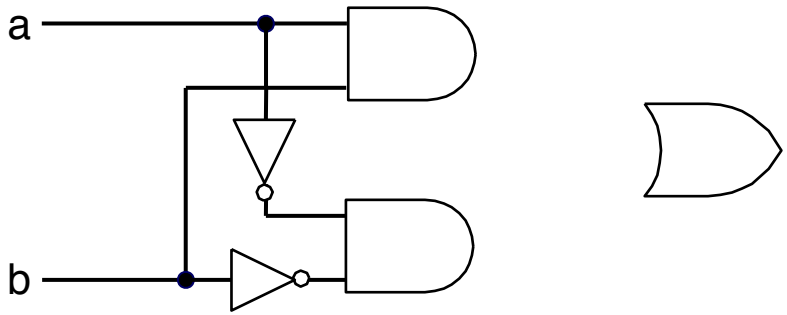
Bit Equality



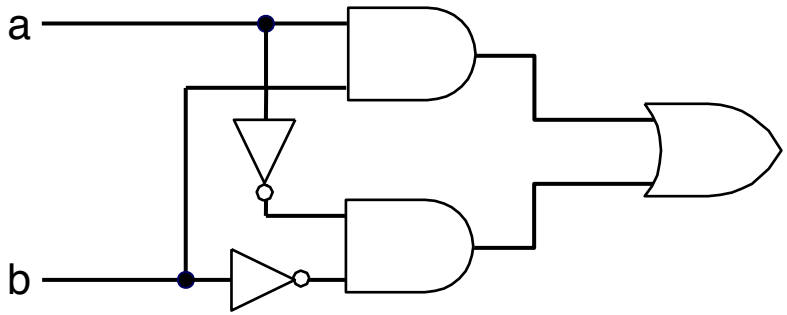
Bit Equality



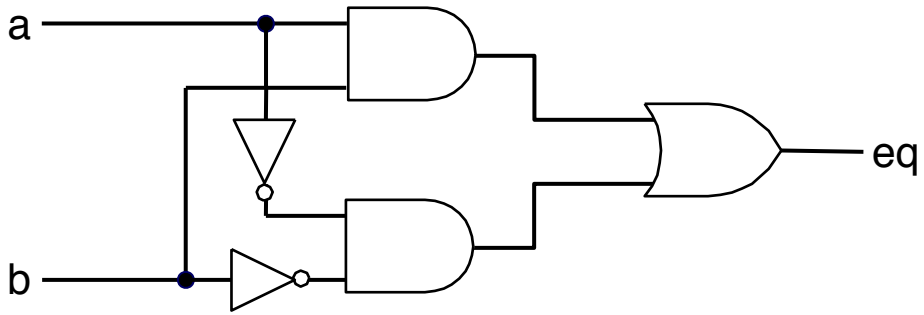
Bit Equality



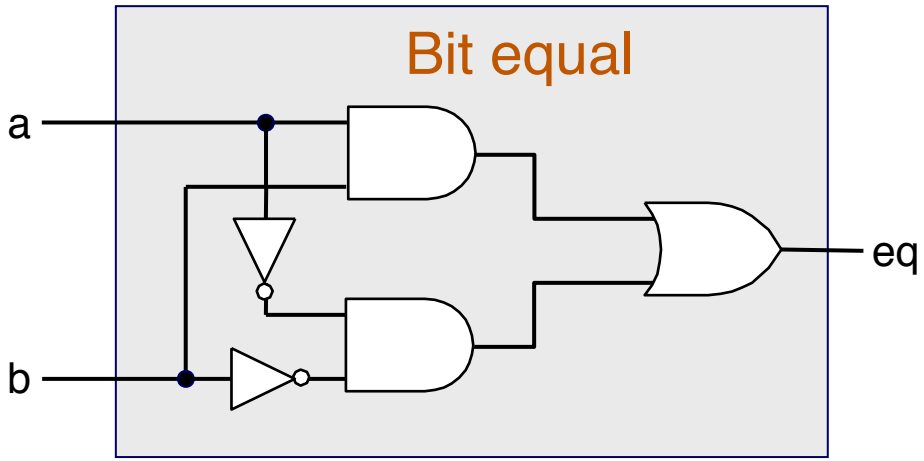
Bit Equality



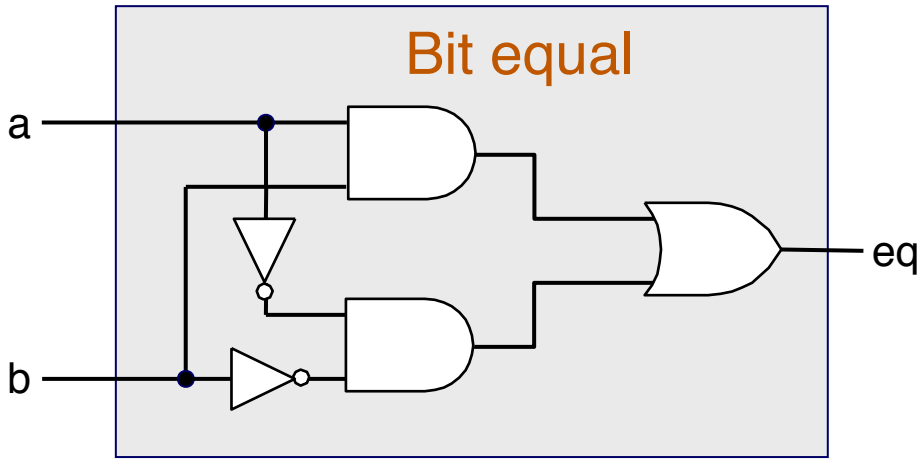
Bit Equality



Bit Equality

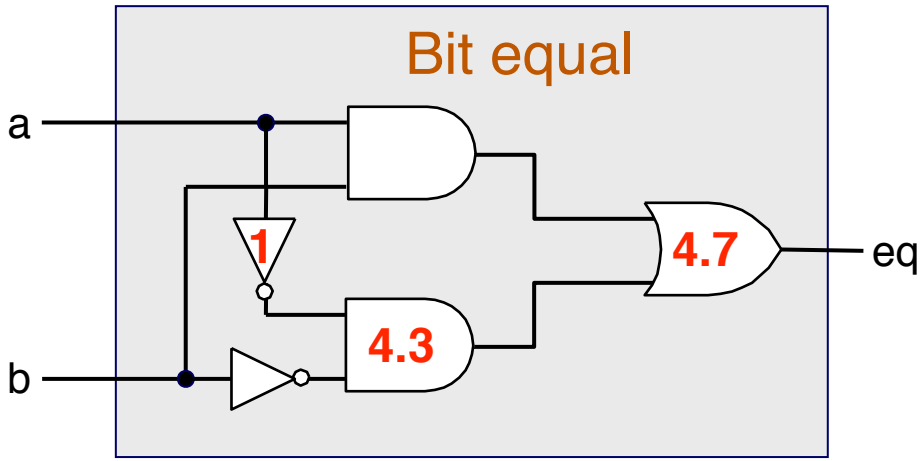


Delay of Bit Equal Circuit



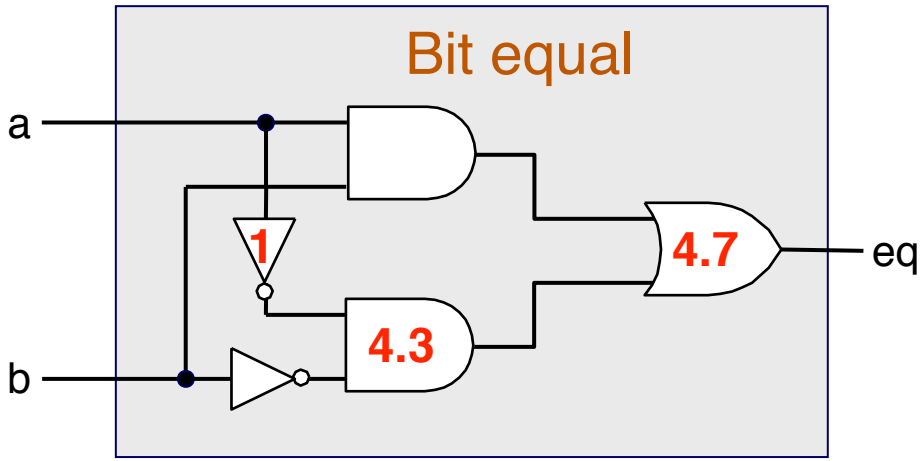
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



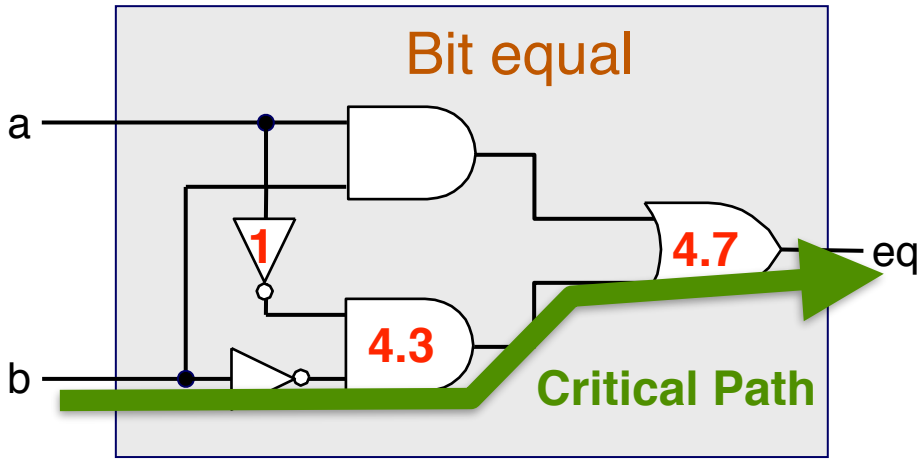
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



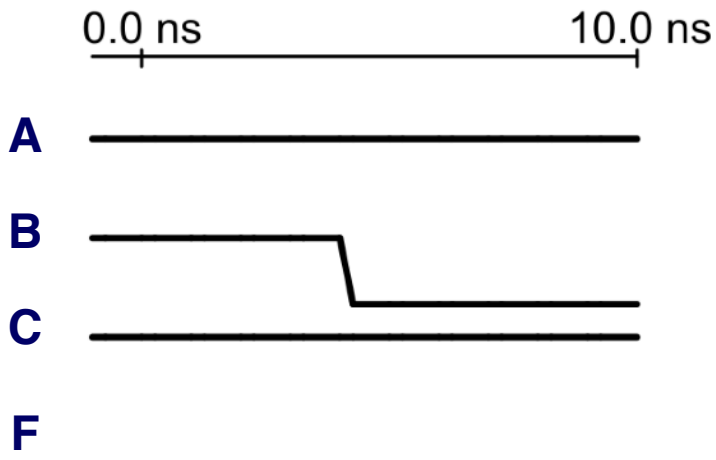
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its "critical path"
  - The path between an input and the output that the maximum delay
  - Estimating the critical path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

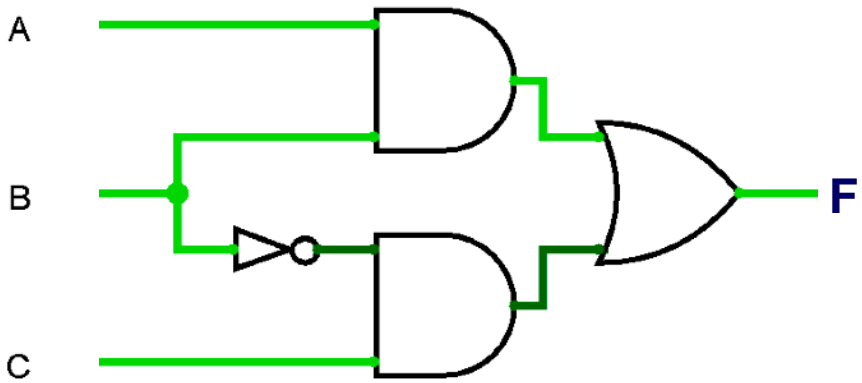
Delay of Bit Equal Circuit



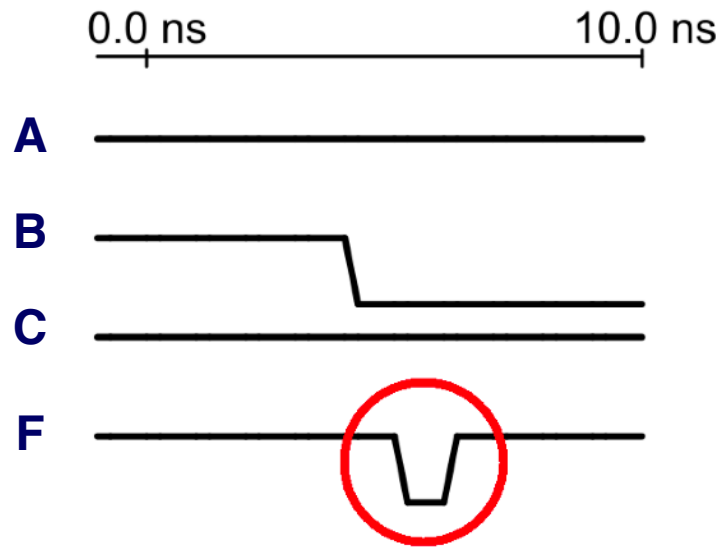
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its "critical path"
  - The path between an input and the output that the maximum delay
  - Estimating the critical path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

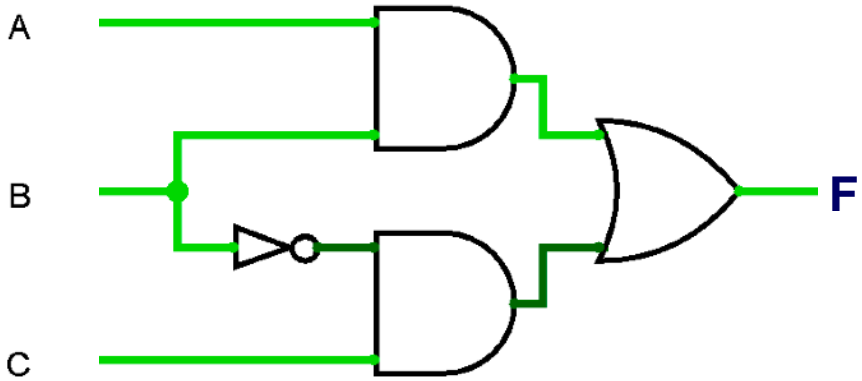
Glitch/Hazard



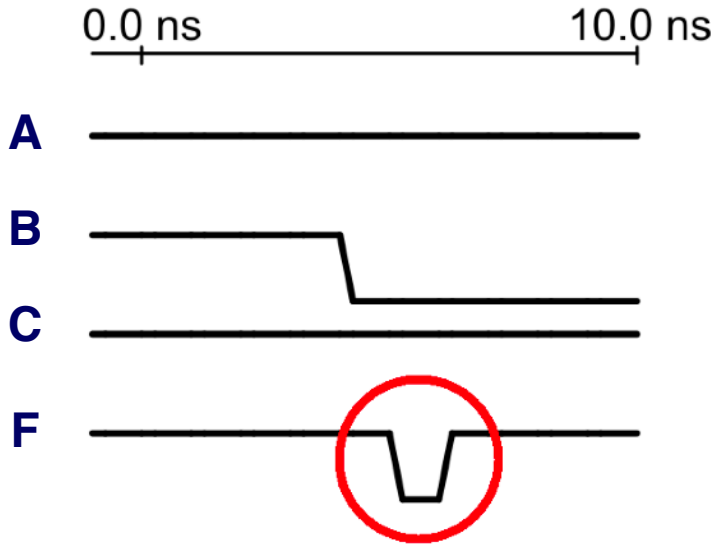


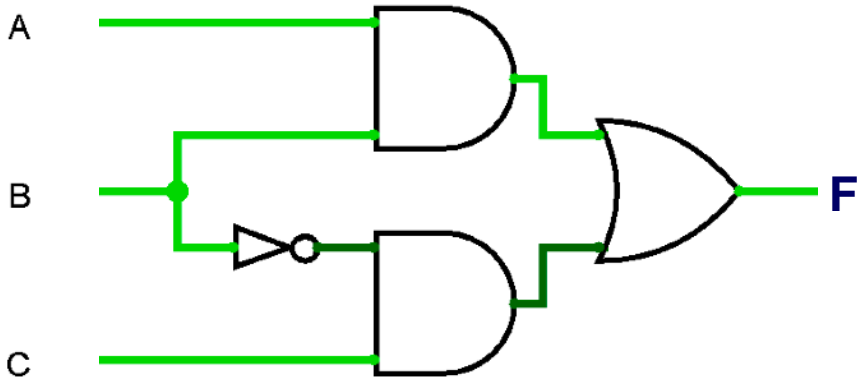
Glitch/Hazard





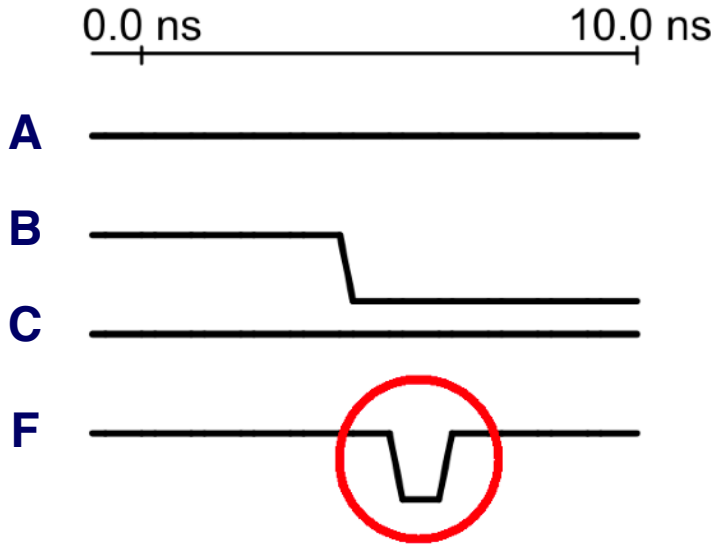
Glitch/Hazard

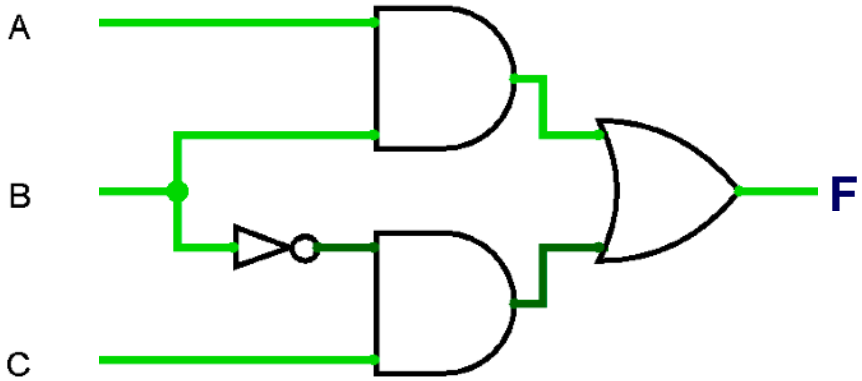




- A glitch is an unnecessary signal transition without functionality.

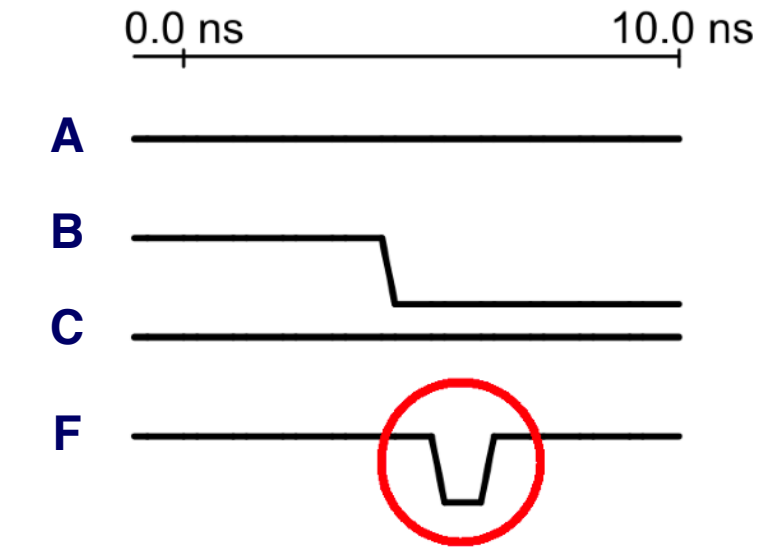
Glitch/Hazard

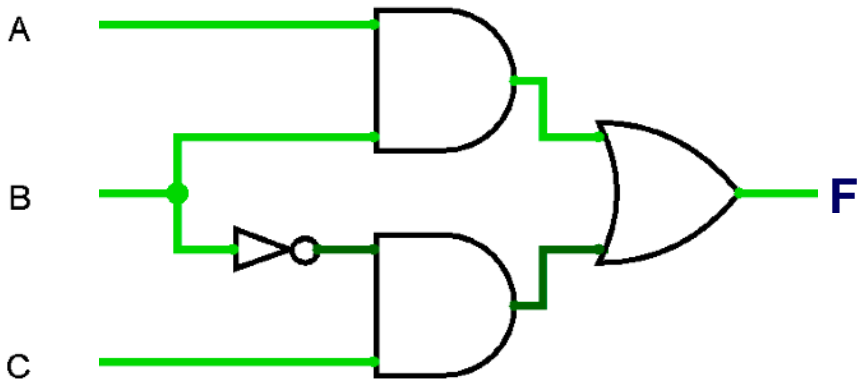




- A glitch is an unnecessary signal transition without functionality.
- Why is it bad? When transistors switch they consume power, but the power consumed during a glitch is a waste.

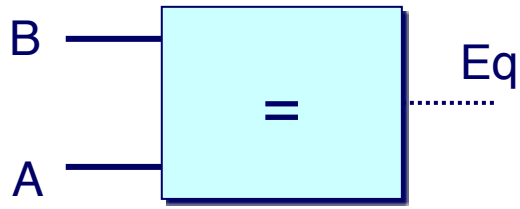
Glitch/Hazard



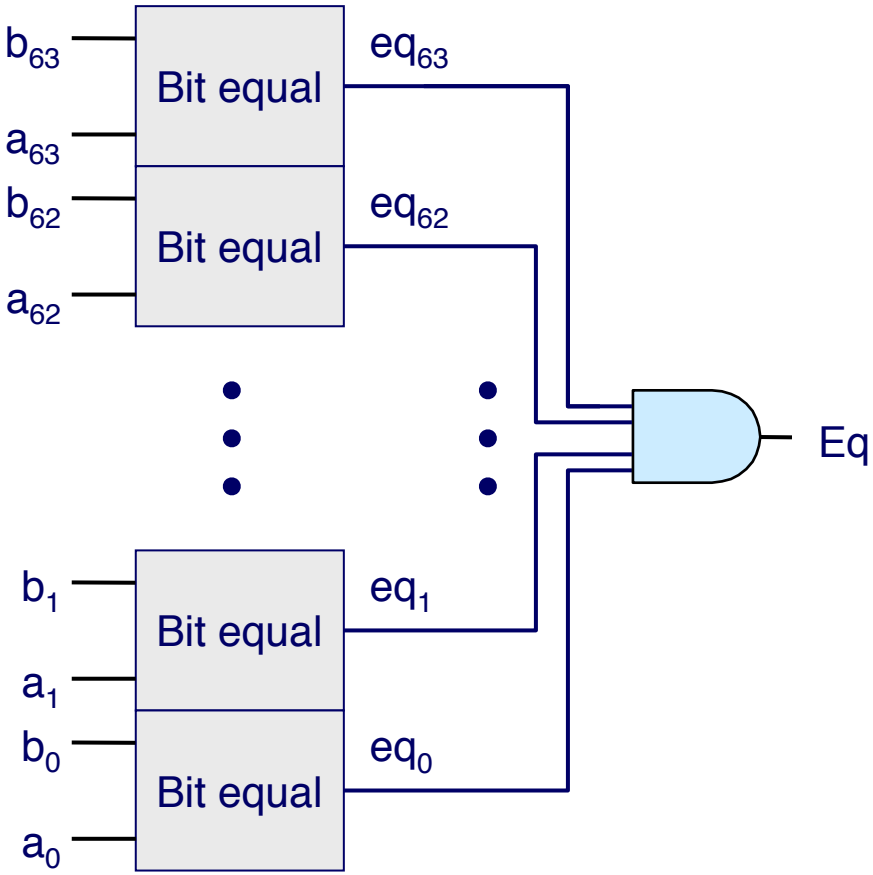


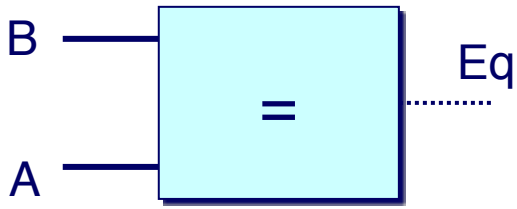
- A glitch is an unnecessary signal transition without functionality.
- Why is it bad? When transistors switch they consume power, but the power consumed during a glitch is a waste.
- Without care, glitch power dissipation is 20%-70% of total power dissipation.

64-bit Equality



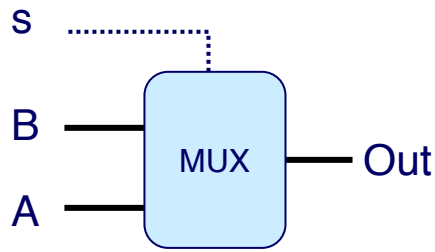
64-bit Equality





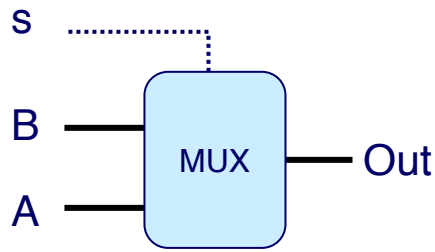
Bit-Level Multiplexor (MUX)

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



Bit-Level Multiplexor (MUX)

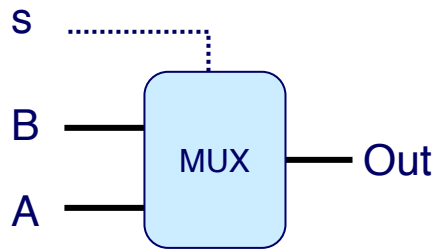
- Control signal s
- Data signals A and B
- Output A when s=1, B when s=0



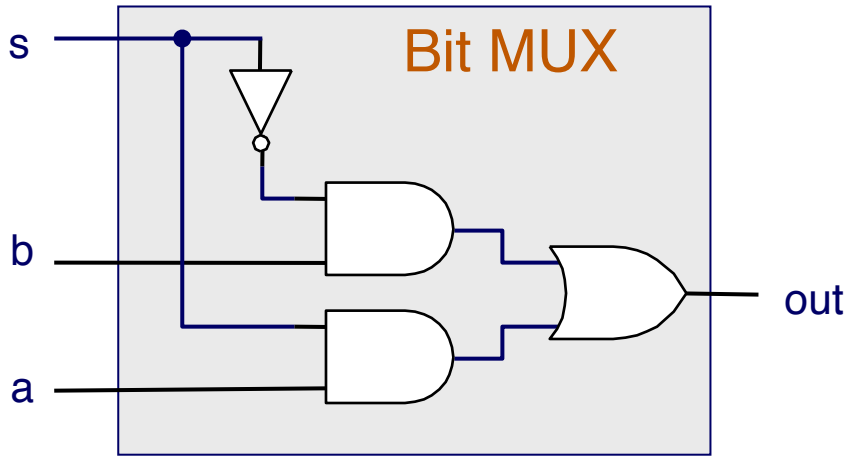

```
bool out = (s&&a)||(!s&&b)
```

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)
```

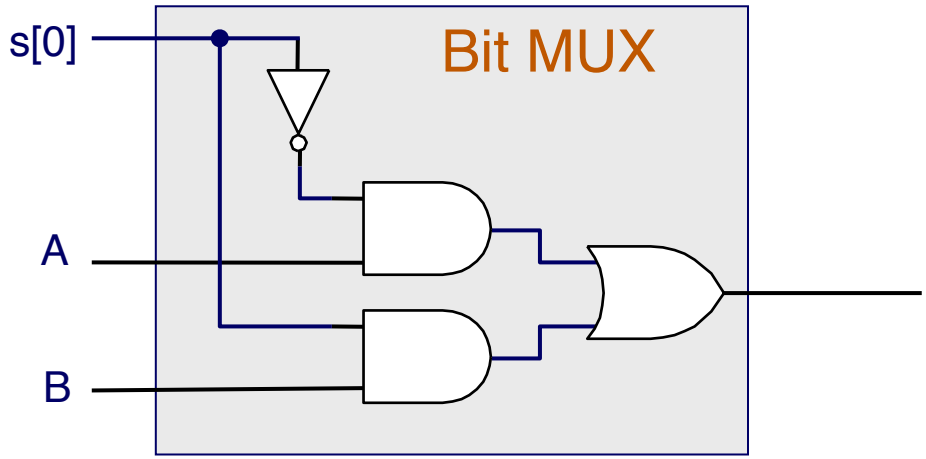


4-Input Multiplexor

- 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

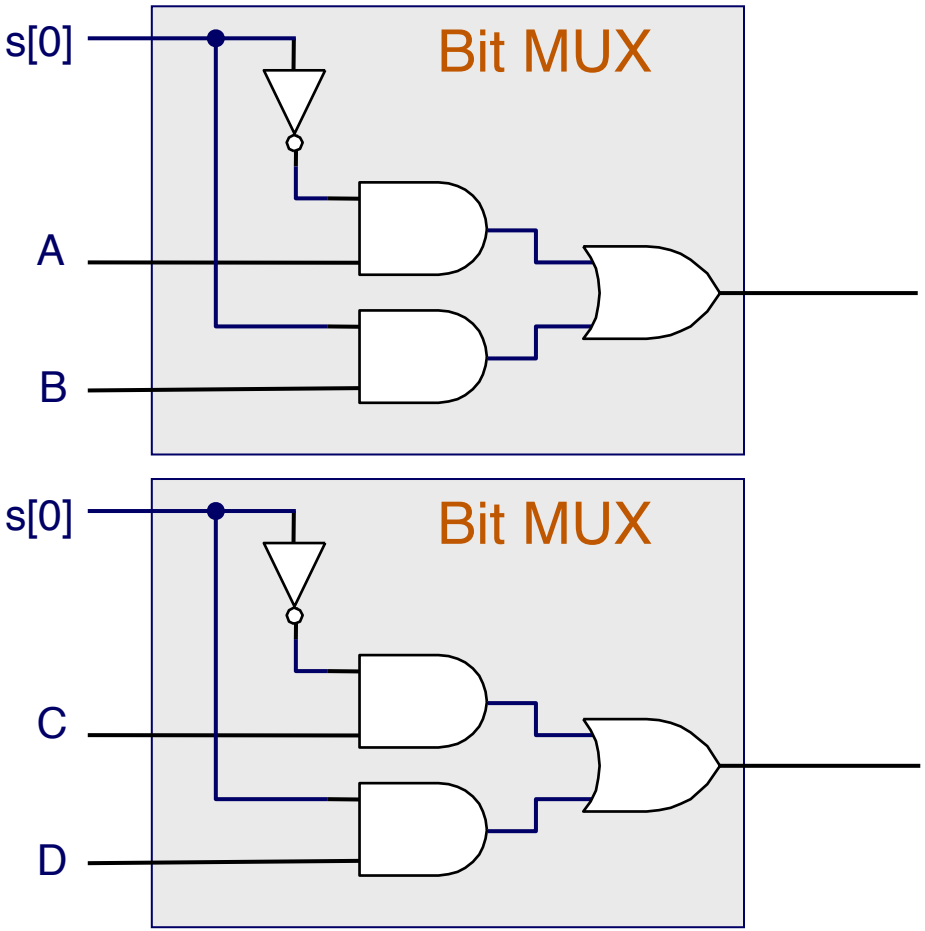
4-Input Multiplexor

- 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



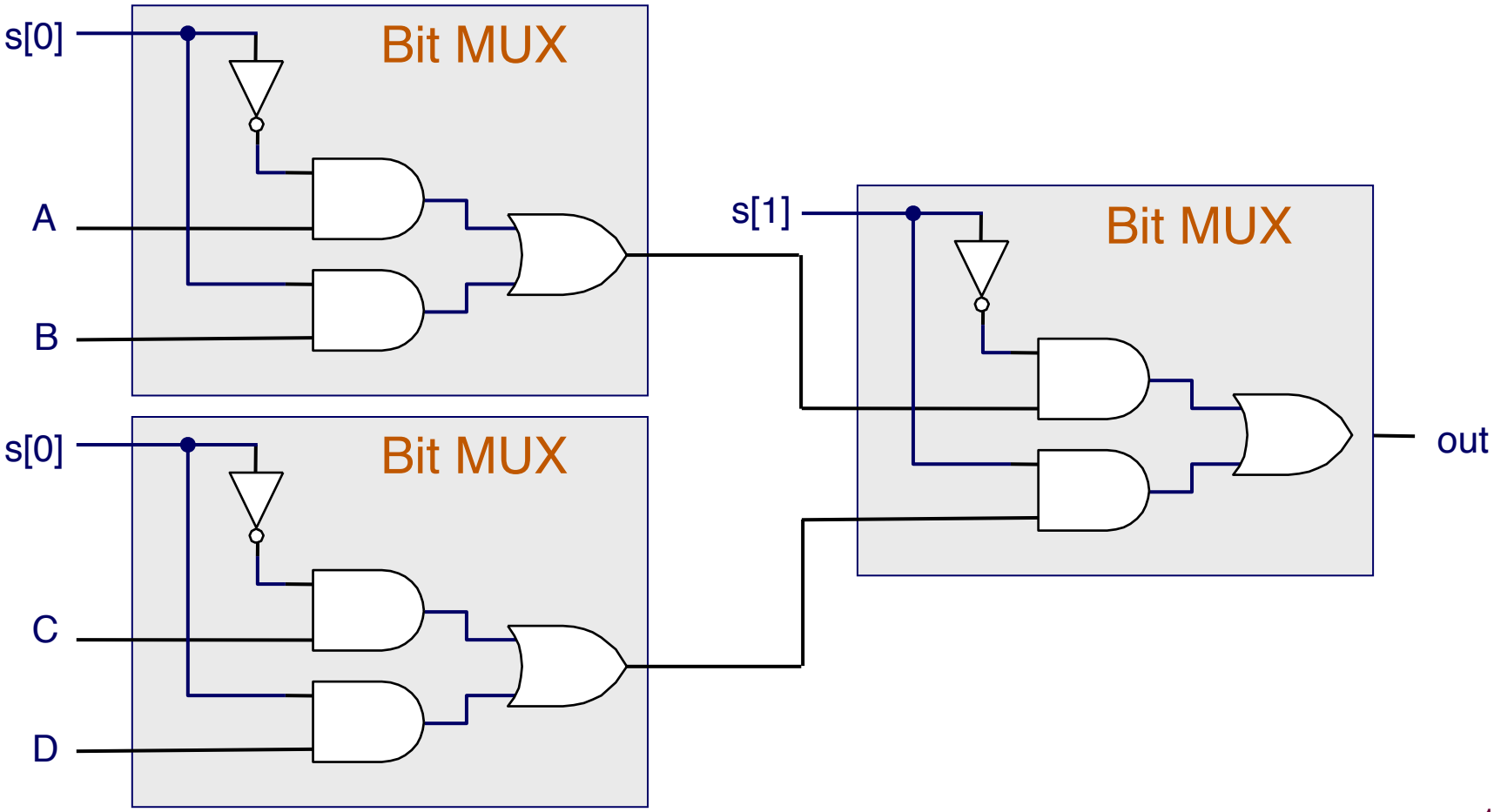
4-Input Multiplexor

- 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



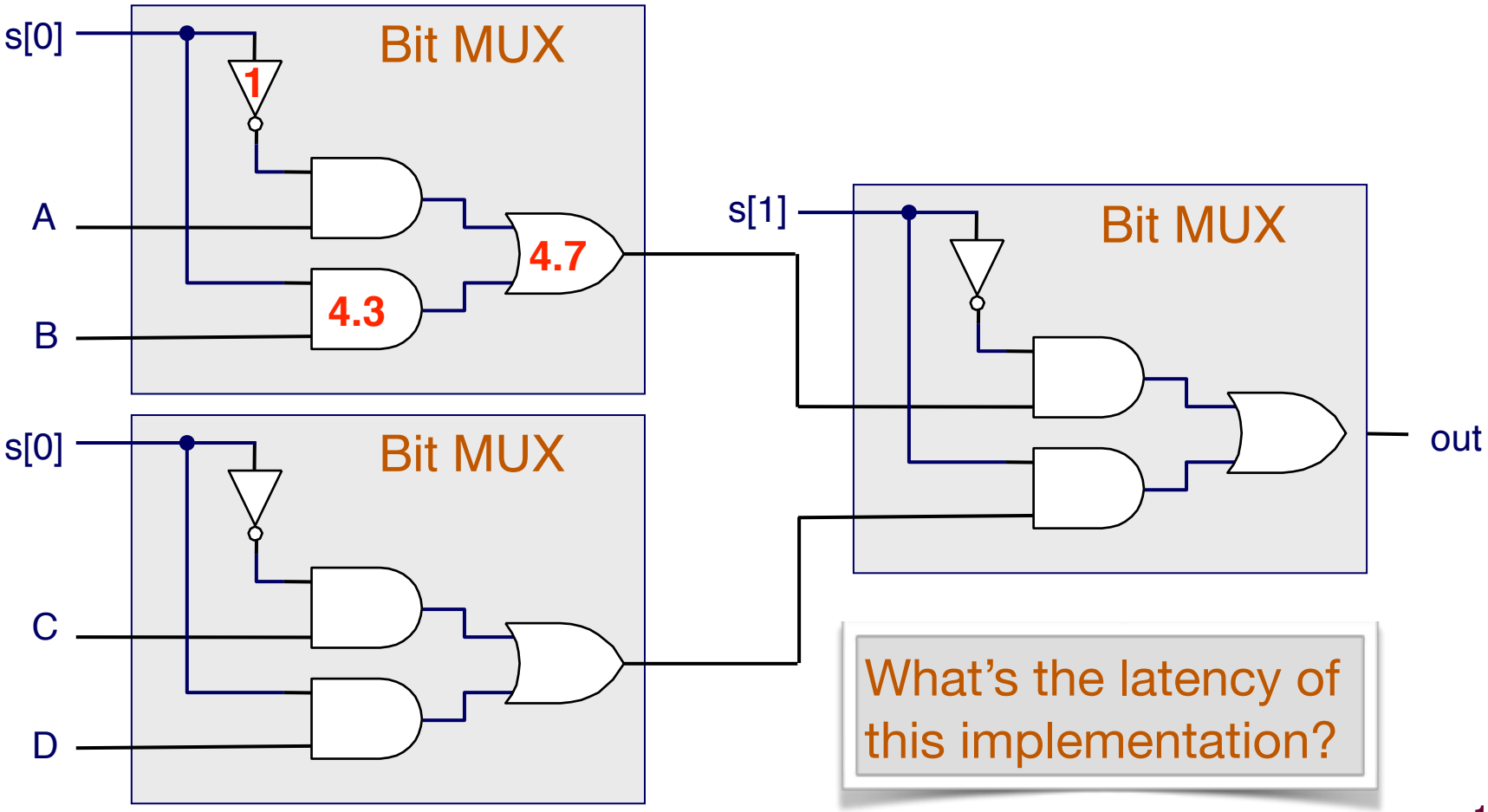
4-Input Multiplexor

- 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

Recall: Full (1-bit) Adder

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

A	B	C_{in}	S	C_{out}
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: Full (1-bit) Adder

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

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

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

A	B	C_{in}	S	C_{out}
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

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

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

Recall: 1-bit Full Adder

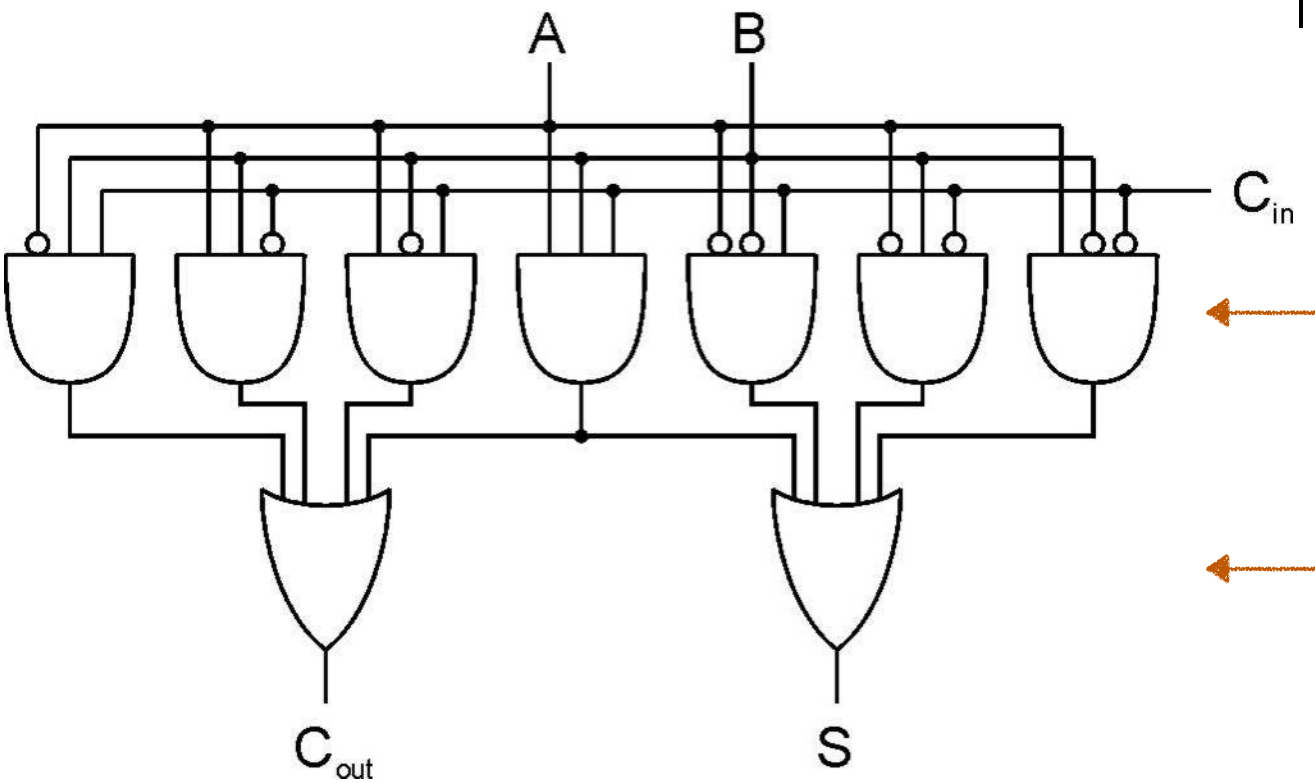
$C_{ou} = (\sim A \,\&\, B \,\&\, C_{in})$

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

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

$\mid (A \,\&\, B \,\&\, C_{in})$

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



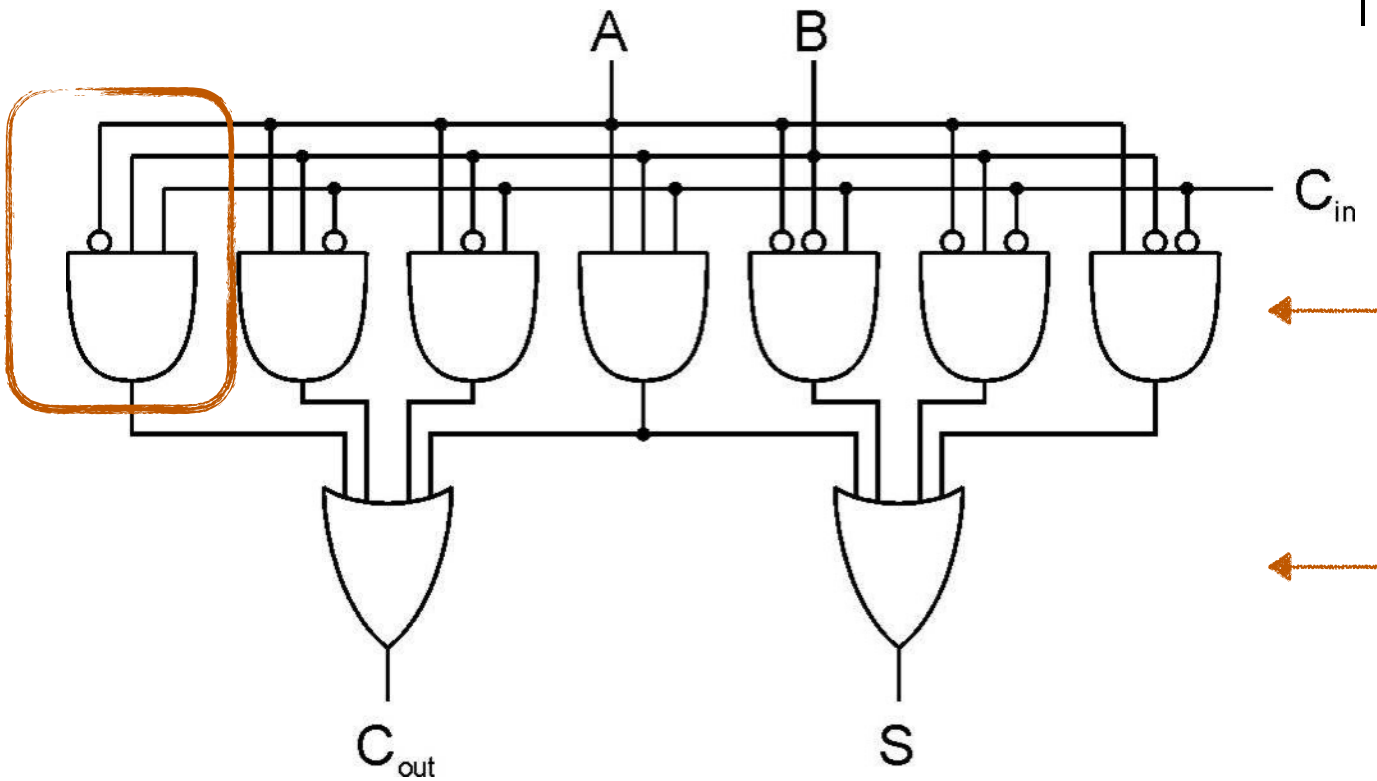
AND Gates

OR Gates

Recall: 1-bit Full Adder

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

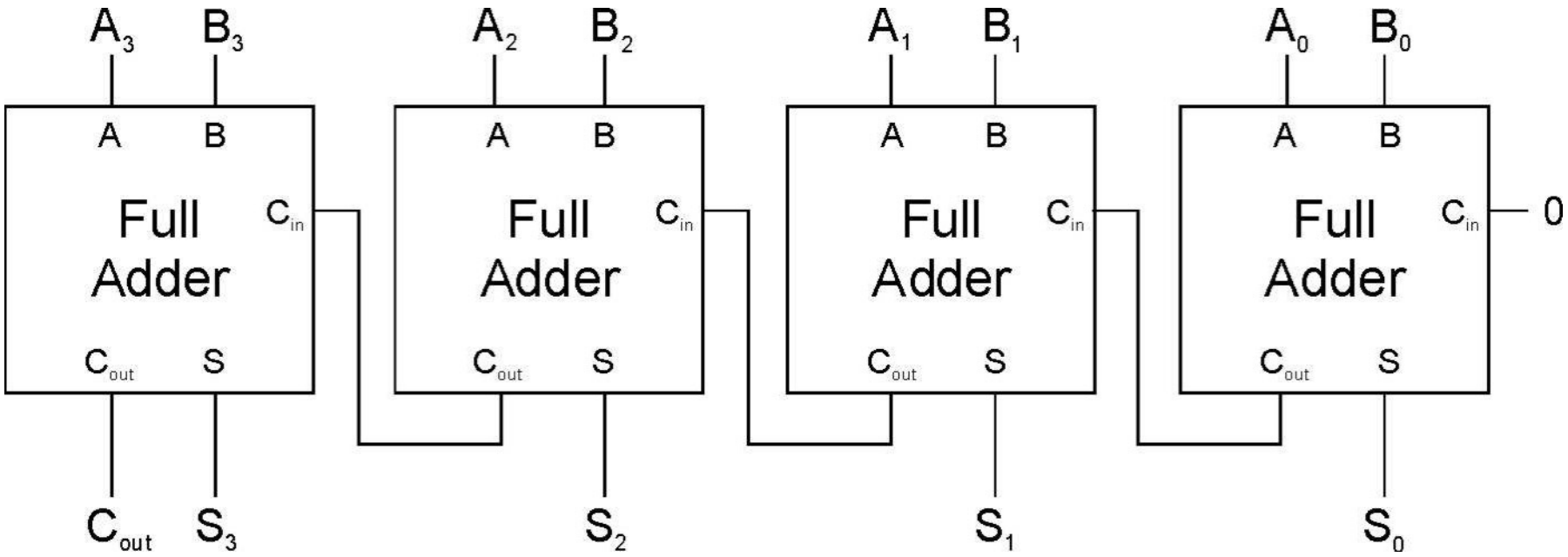
Add two bits and carry-in,
produce one-bit sum and carry-out.



AND Gates

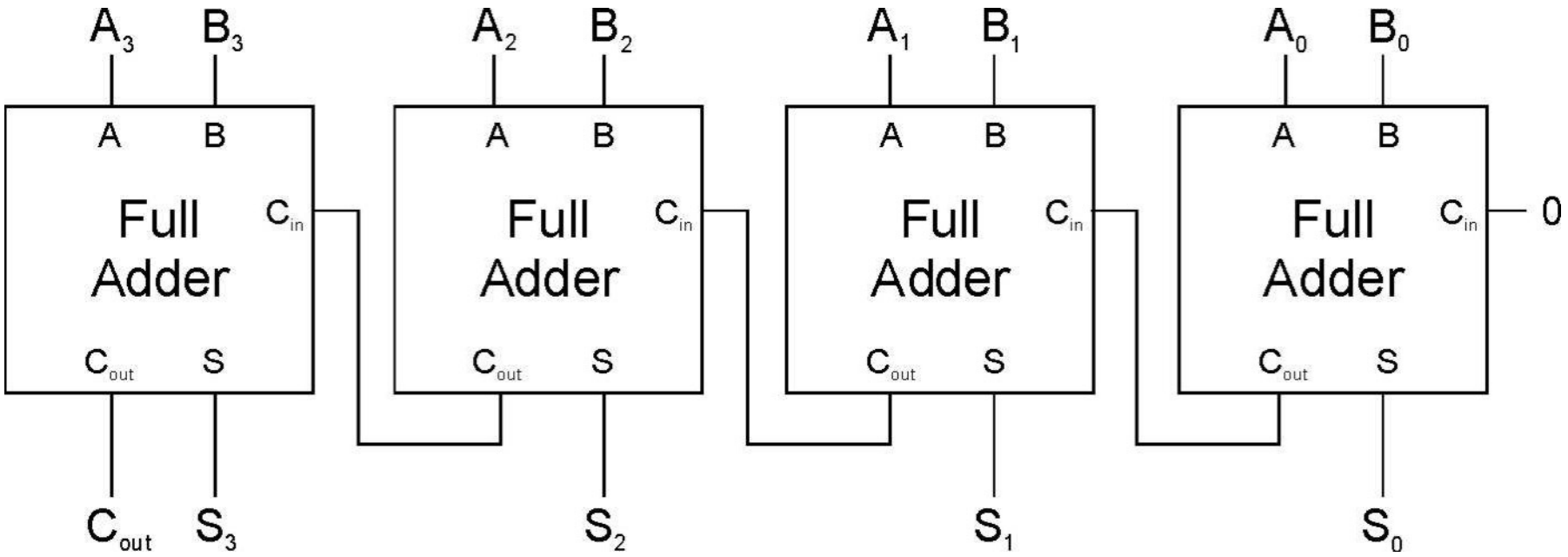
OR Gates

Recall: Four-bit Adder



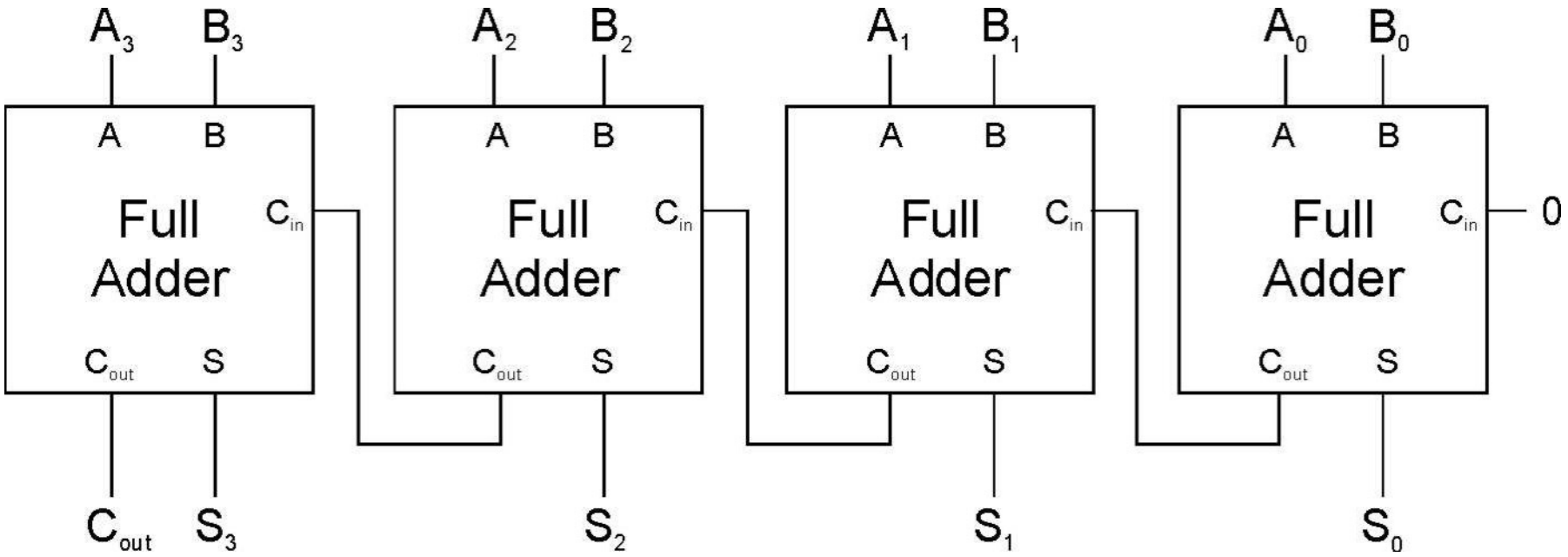
Recall: Four-bit Adder

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

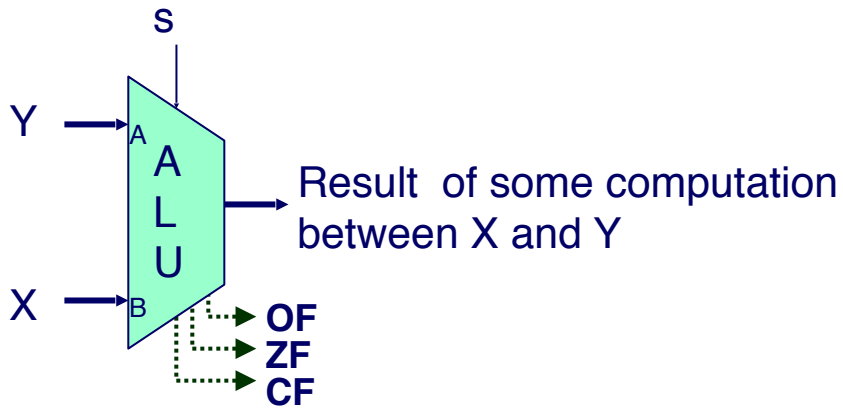


Recall: Four-bit Adder

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



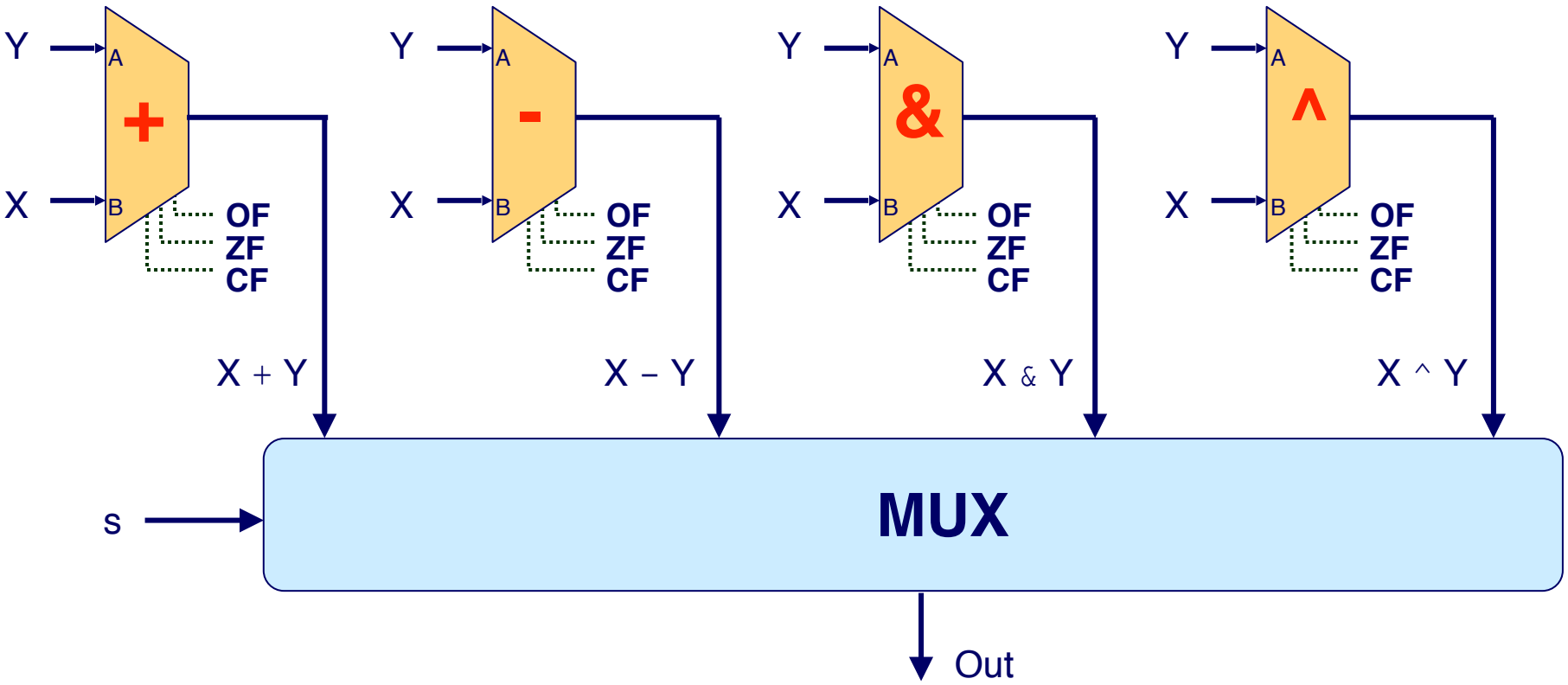
Arithmetic Logic Unit



- 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 ^ 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.

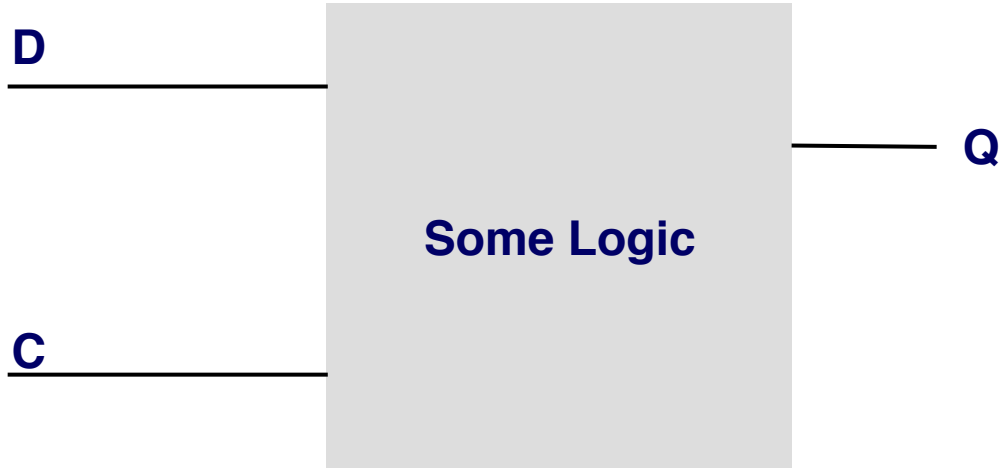
The Need for Storing Bits

- 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

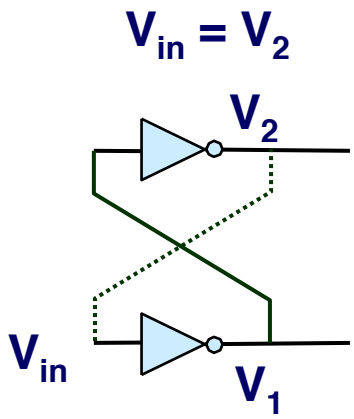
- Assembly programs set architecture (processor) states.
  - Register File
  - Status Flags
  - Memory
  - Program Counter
- Every state is essentially some bits that are stored/loaded.
- Think of the program execution as an FSM.
- The hardware must provide mechanisms to load and store bits.
- There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

Build a 1-Bit Storage

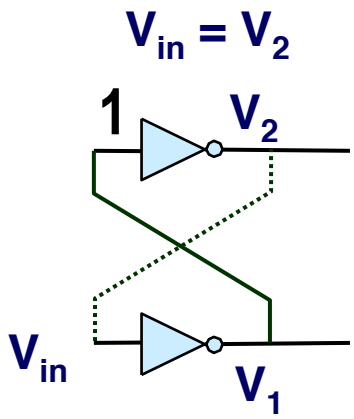


- What we would like:
  - D is the data we want to store (0 or 1)
  - C is the control signal
    - When C is 1, Q becomes D (i.e., storing the data)
    - When C is 0, Q doesn't change with D (data stored)

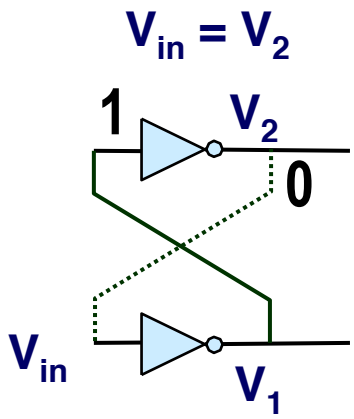
Bitstable Element



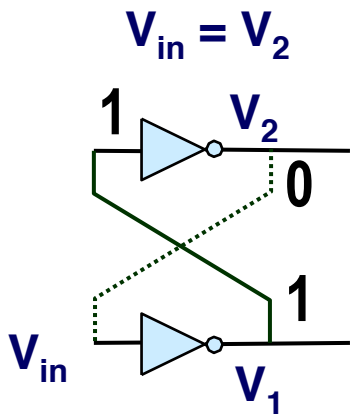
Bitstable Element



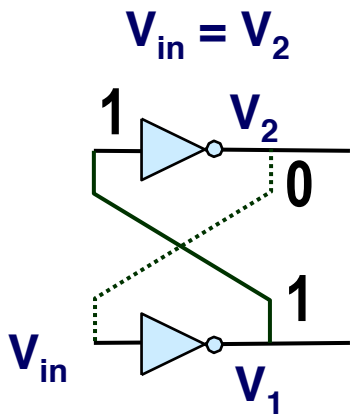
Bitstable Element



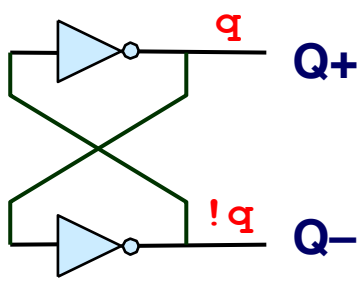
Bitstable Element



Bitstable Element

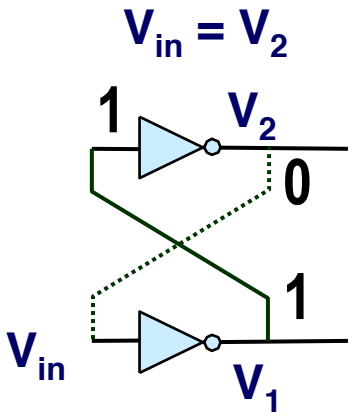


Bistable Element

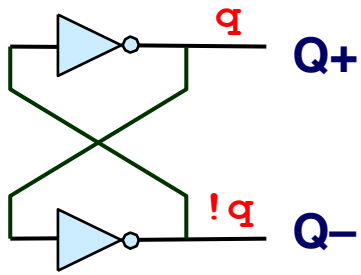


q = 0 or 1

Bitstable Element



Bistable Element

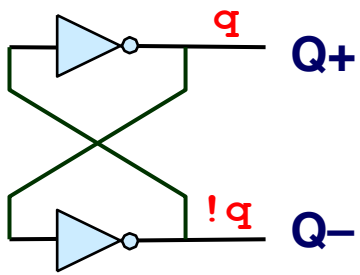


`q` = 0 or 1

Q+ ***continuously*** outputs q.

Storing and Accessing 1 Bit

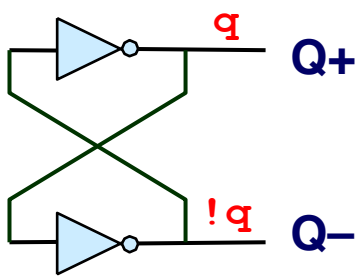
Bistable Element



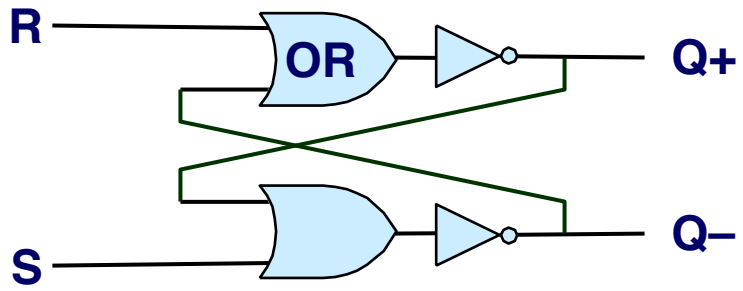
`q` = 0 or 1

Storing and Accessing 1 Bit

Bistable Element

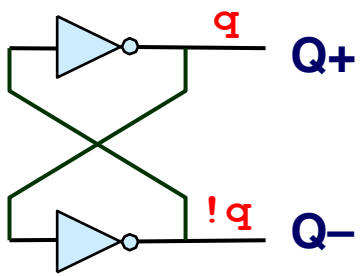


`q` = 0 or 1

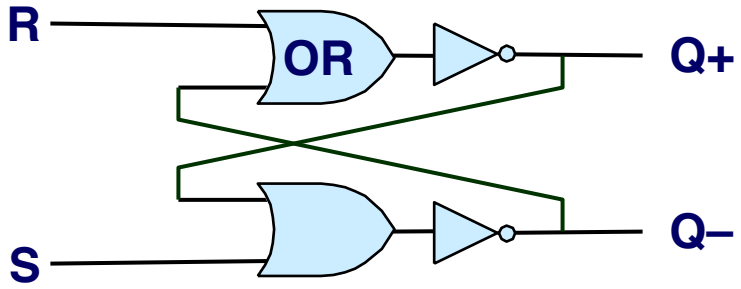


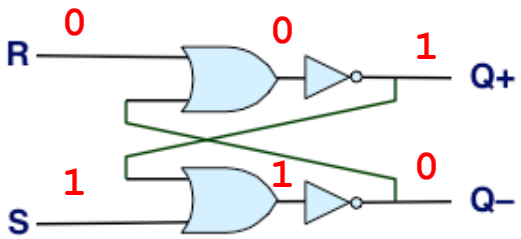
Storing and Accessing 1 Bit

Bistable Element



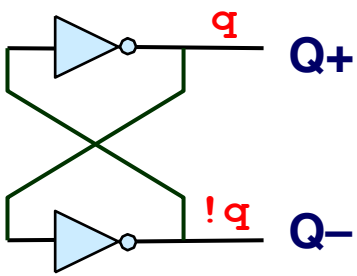
q = 0 or 1



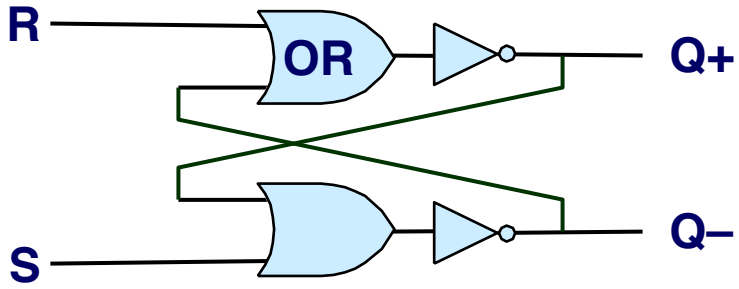


Storing and Accessing 1 Bit

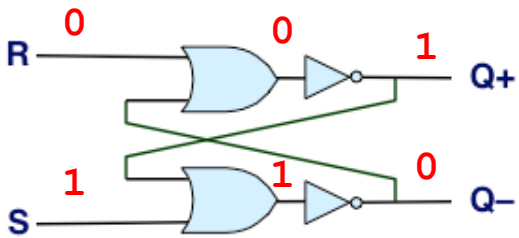
Bistable Element



q = 0 or 1

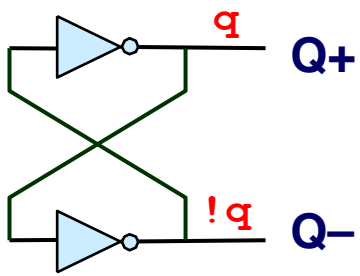


Setting Q+ to 1

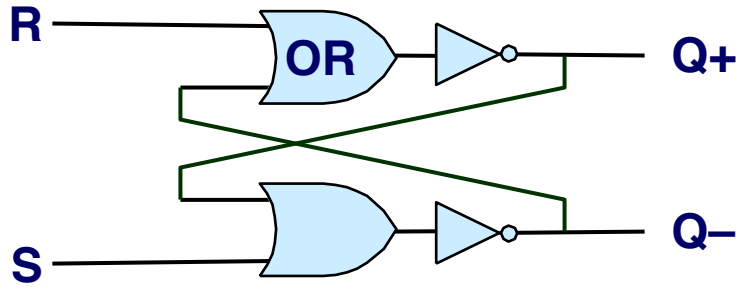


Storing and Accessing 1 Bit

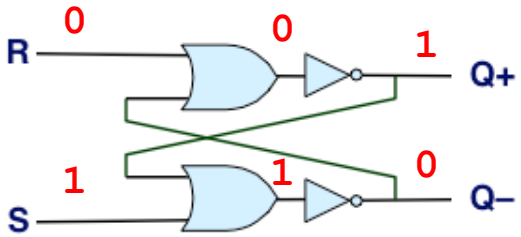
Bistable Element

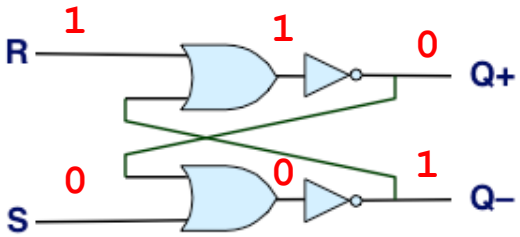


q = 0 or 1



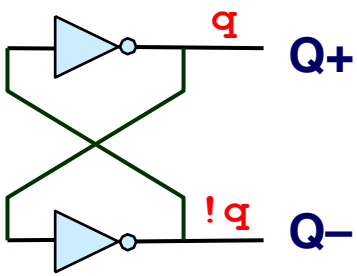
Setting Q+ to 1



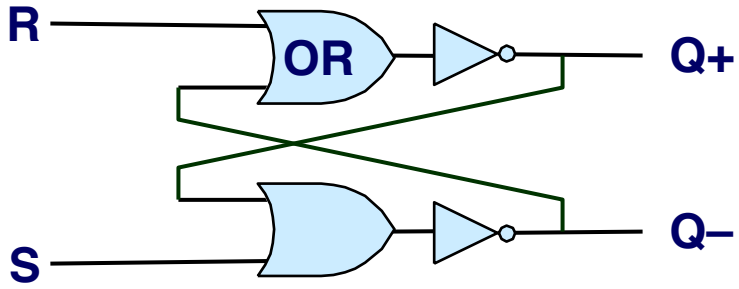


Storing and Accessing 1 Bit

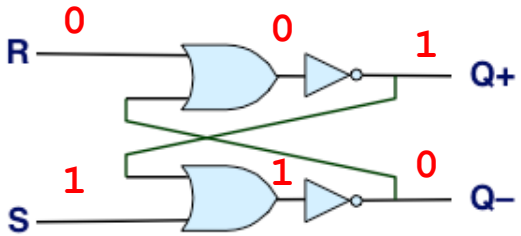
Bistable Element



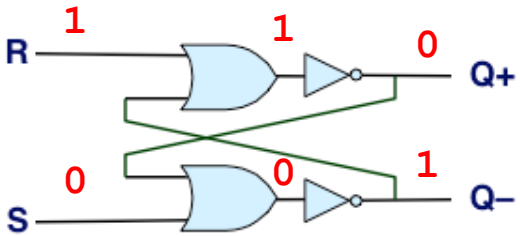
q = 0 or 1



Setting Q+ to 1

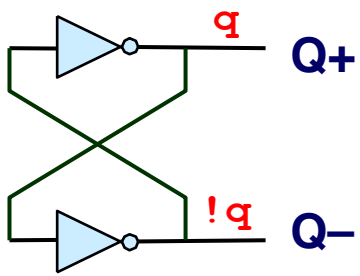


Setting Q+ to 0

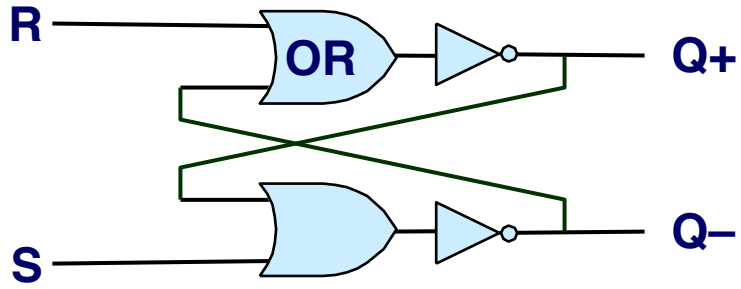


Storing and Accessing 1 Bit

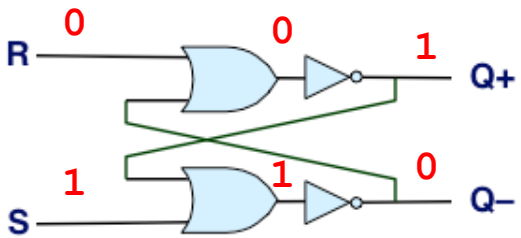
Bistable Element



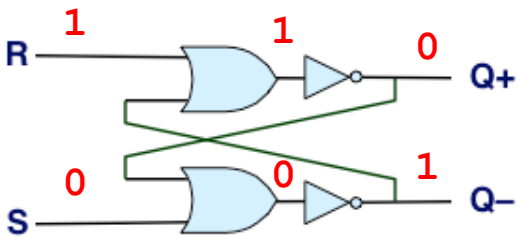
q = 0 or 1

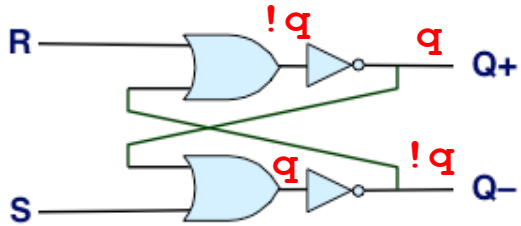


Setting Q+ to 1



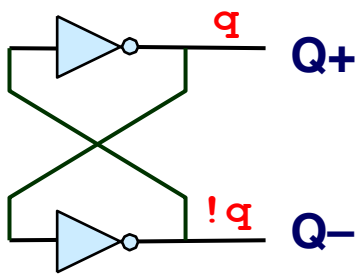
Setting Q+ to 0



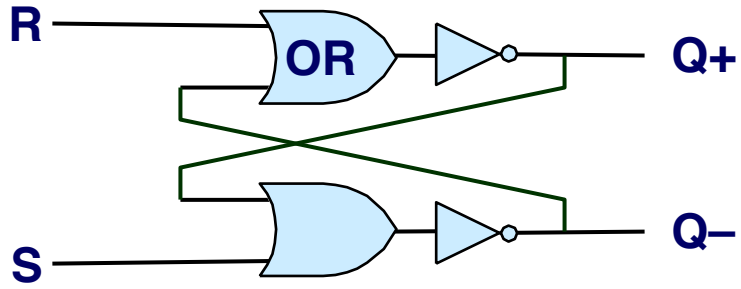


Storing and Accessing 1 Bit

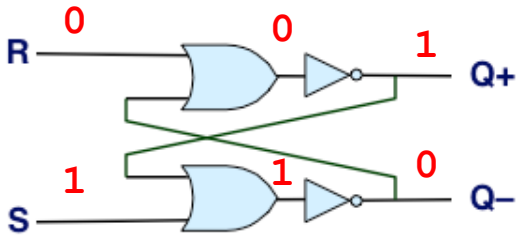
Bistable Element



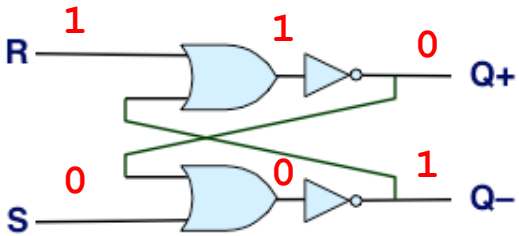
q = 0 or 1

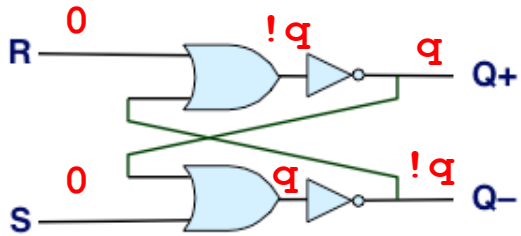


Setting Q+ to 1



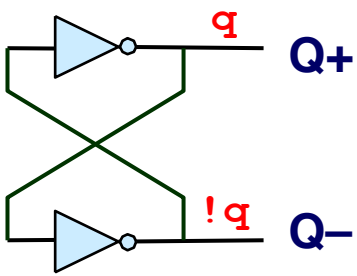
Setting Q+ to 0

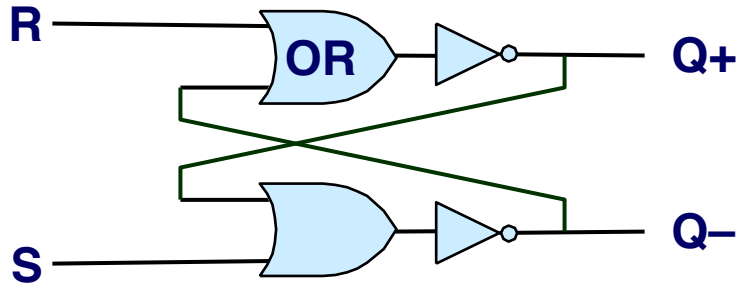




Storing and Accessing 1 Bit

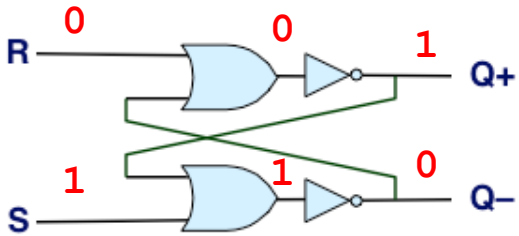
Bistable Element



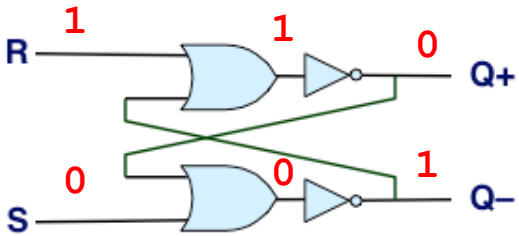


`q` = 0 or 1

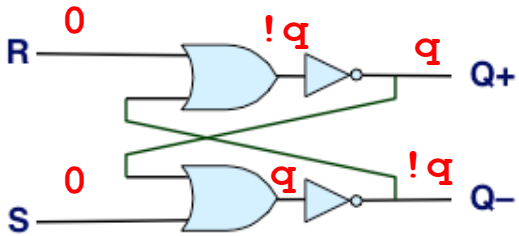
Setting Q+ to 1



Setting Q+ to 0

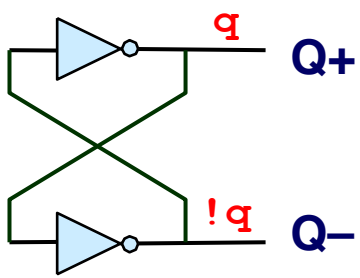


Q+ value unchanged i.e., stored!



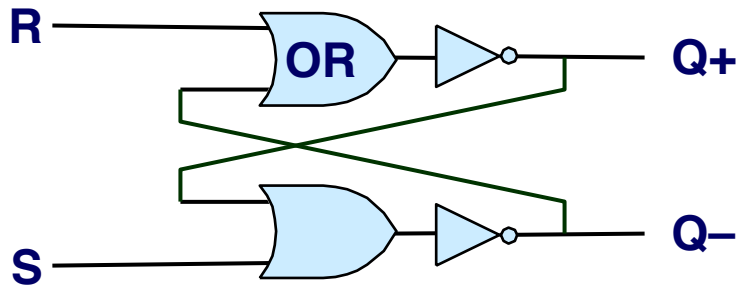
Storing and Accessing 1 Bit

Bistable Element

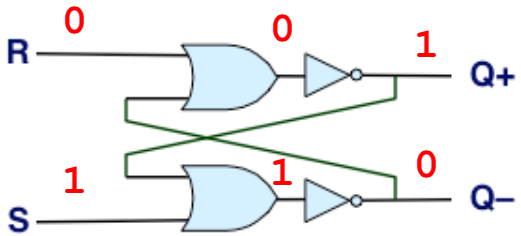


q = 0 or 1

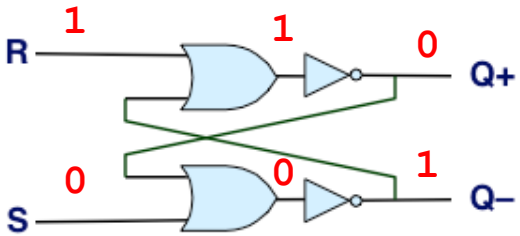
R-S Latch



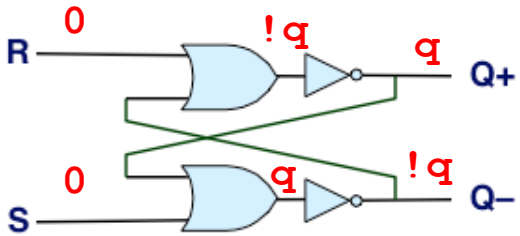
Setting Q+ to 1



Setting Q+ to 0

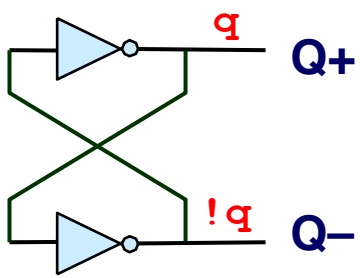


Q+ value unchanged i.e., stored!



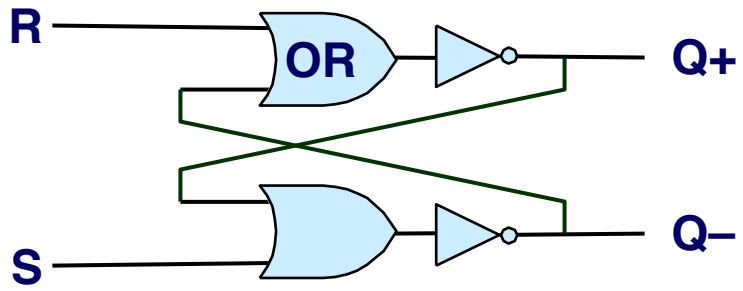
Storing and Accessing 1 Bit

Bistable Element

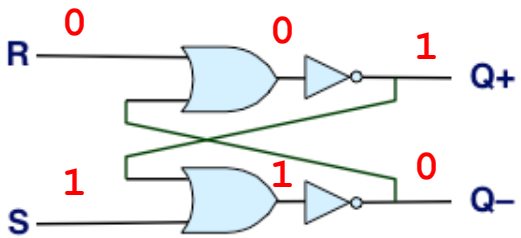


q = 0 or 1

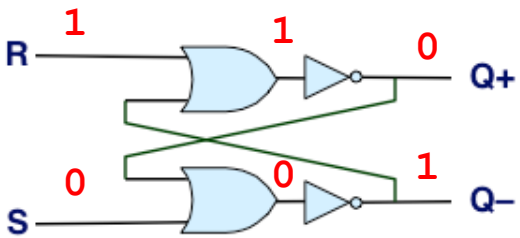
R-S Latch



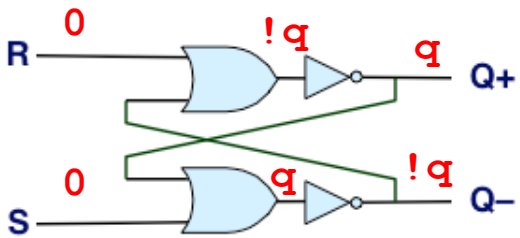
Setting Q+ to 1



Setting Q+ to 0

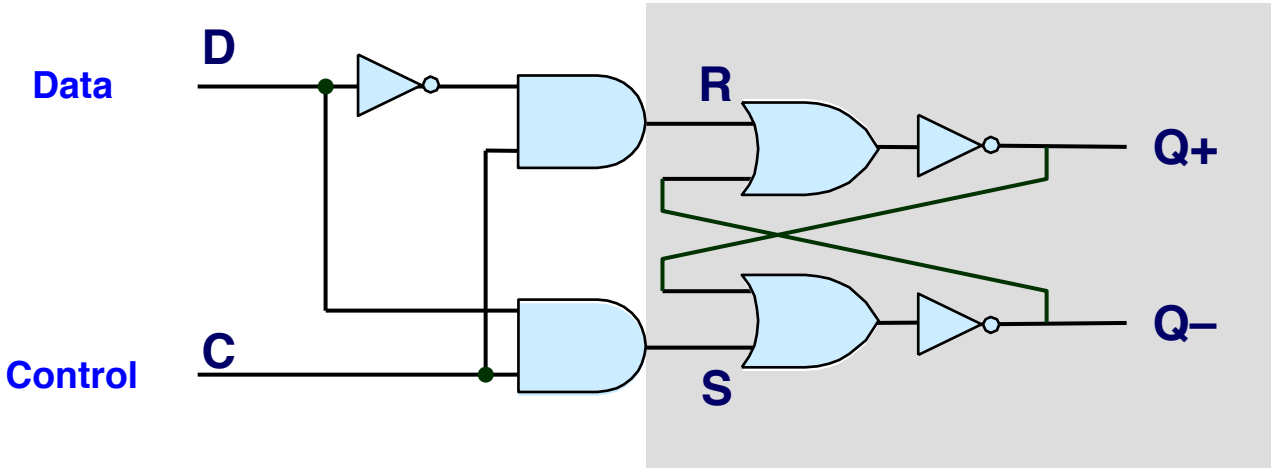


Q+ value unchanged i.e., stored!



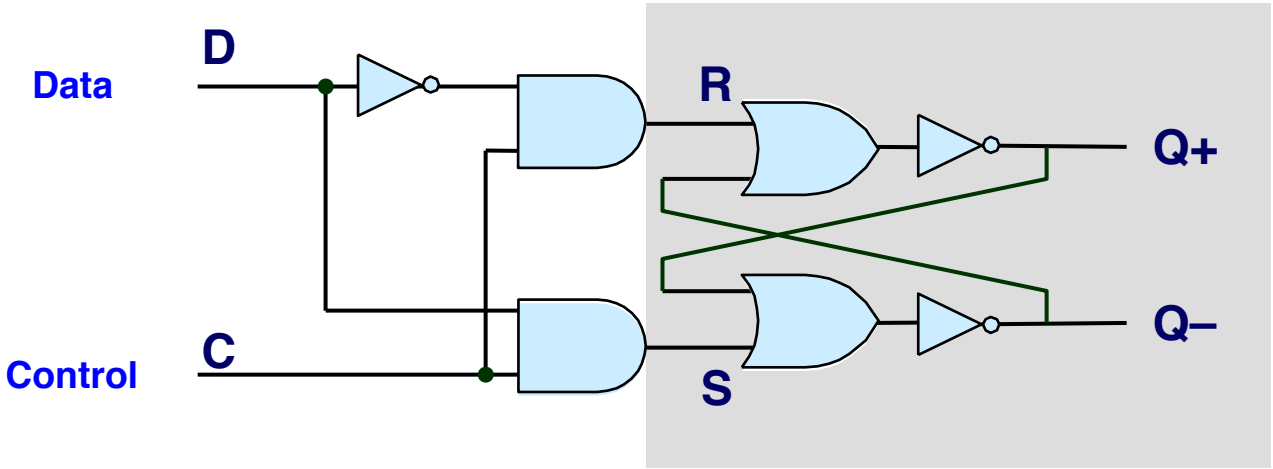
If R and S are different, Q+ is the same as S

Building on top of R-S Latch

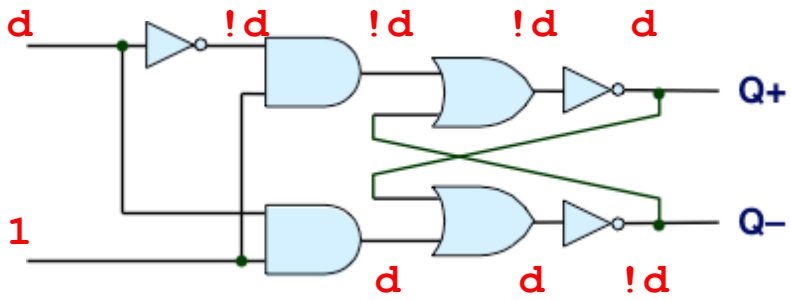


If R and S are different, Q+ is the same as S

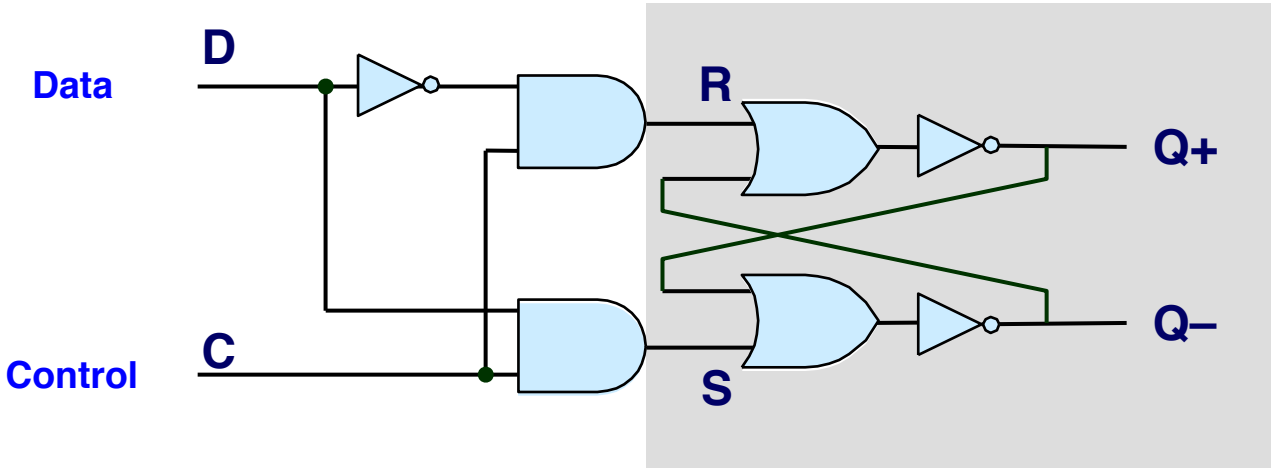
Building on top of R-S Latch



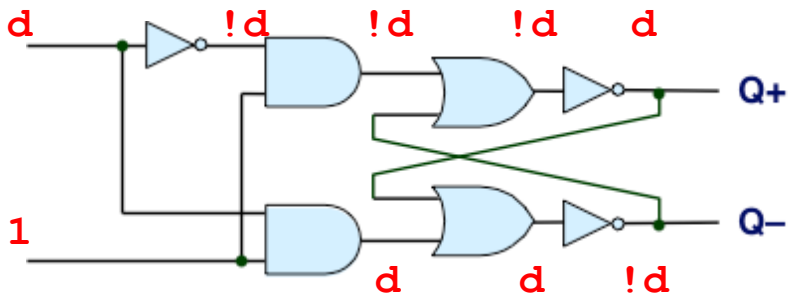
If R and S are different, Q+ is the same as S



Building on top of R-S Latch

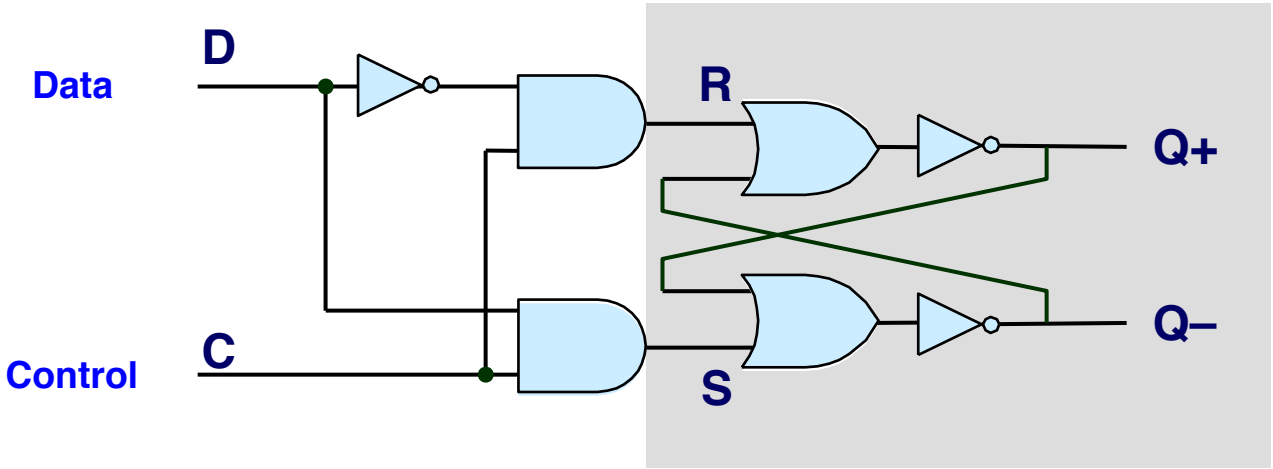


If R and S are different, Q+ is the same as S



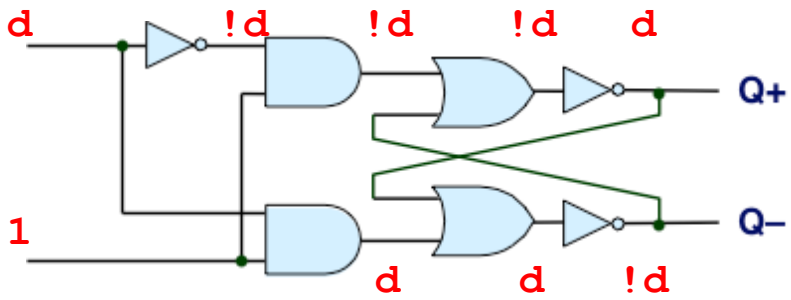
Q+ will continuously change as d changes

Building on top of R-S Latch



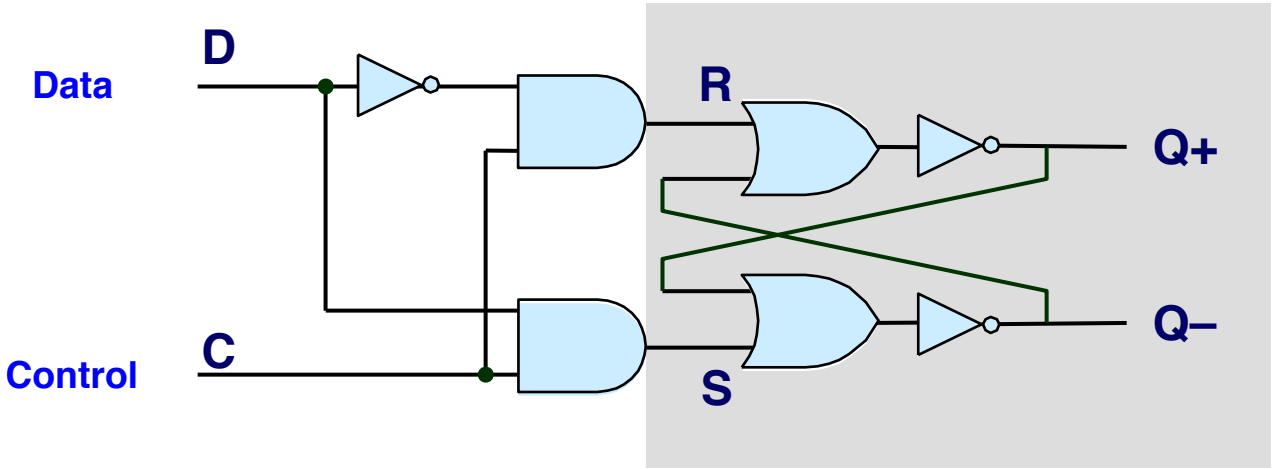
If R and S are different, Q+ is the same as S

Storing Data (Latching)



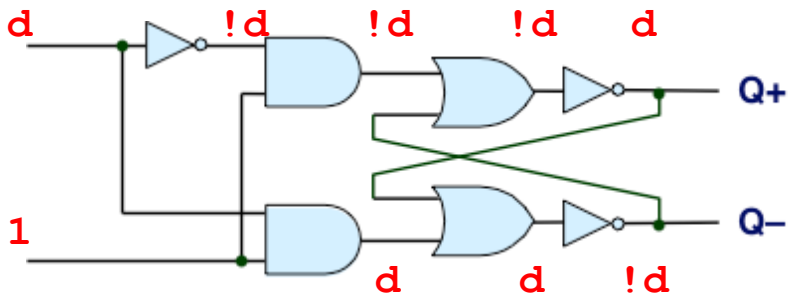
Q+ will continuously change as d changes

Building on top of R-S Latch

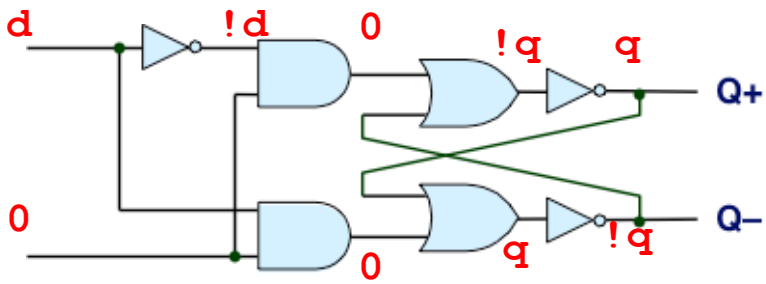


If R and S are different, Q+ is the same as S

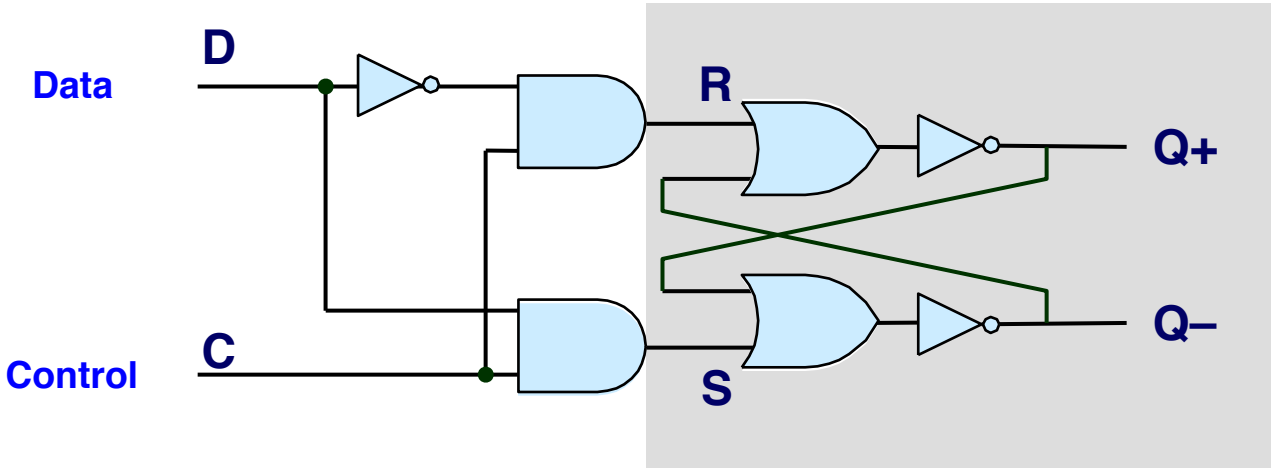
Storing Data (Latching)



Q+ will continuously change as d changes

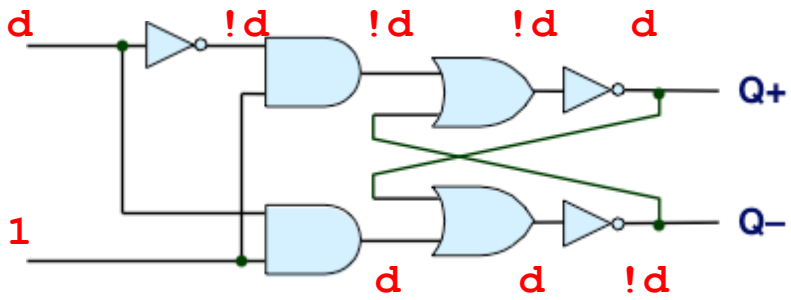


Building on top of R-S Latch

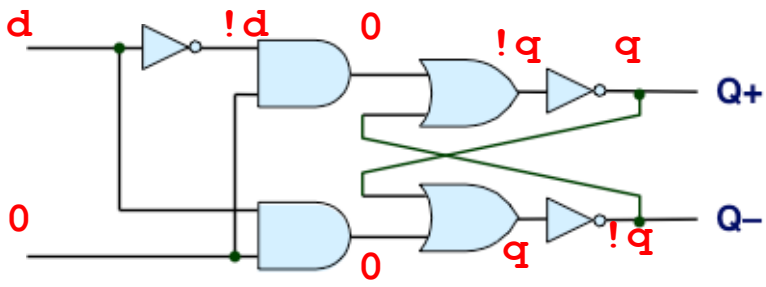


If R and S are different, Q+ is the same as S

Storing Data (Latching)

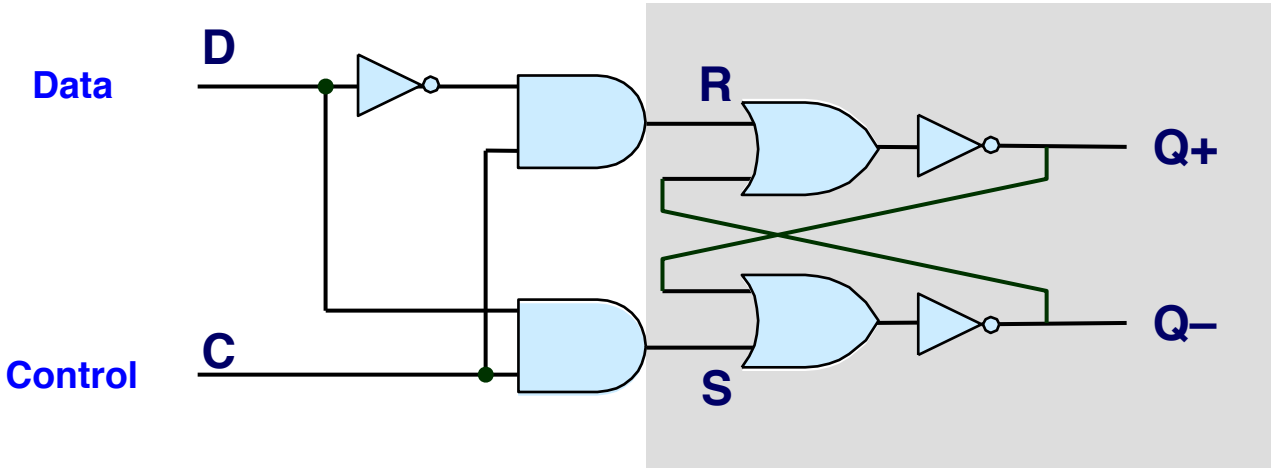


Q+ will continuously change as d changes



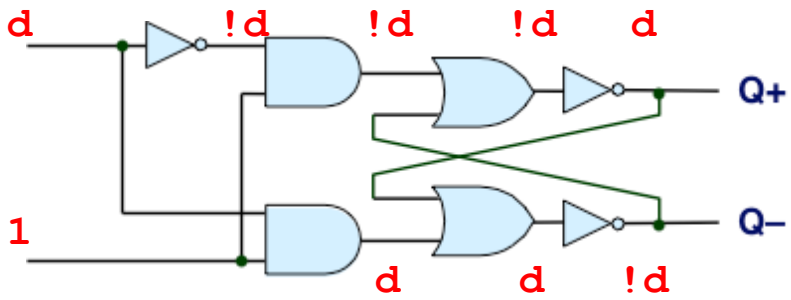
Q+ doesn't change with d

Building on top of R-S Latch



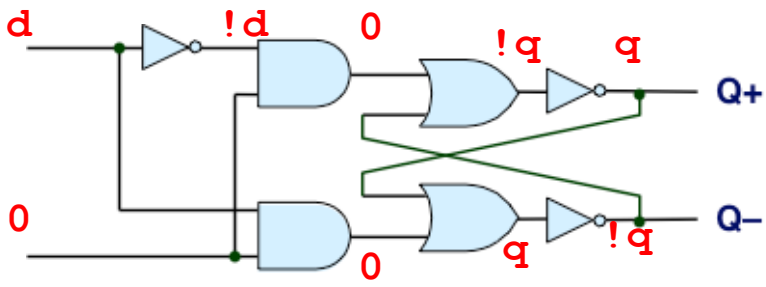
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

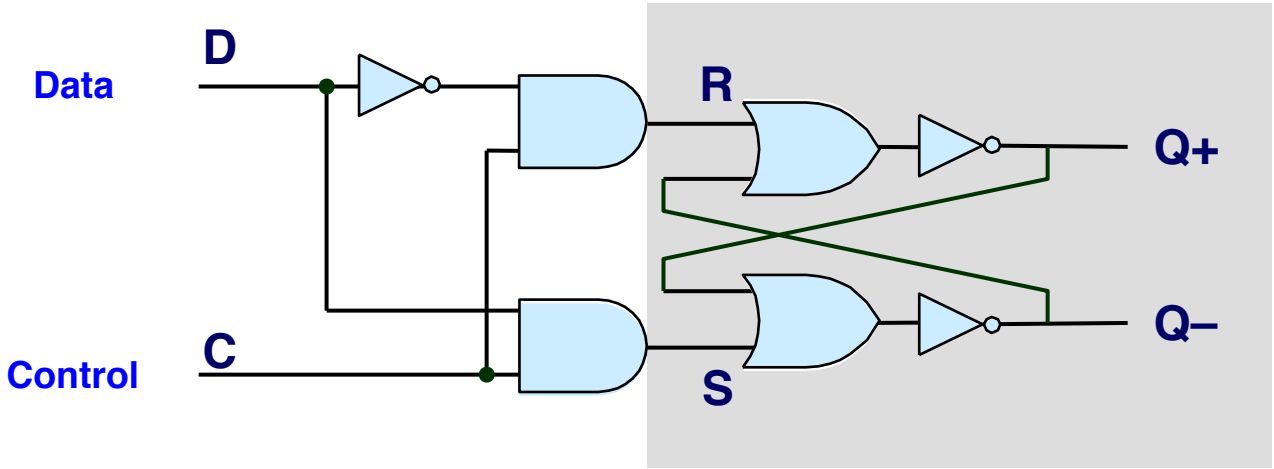
Holding Data



Q+ doesn't change with d

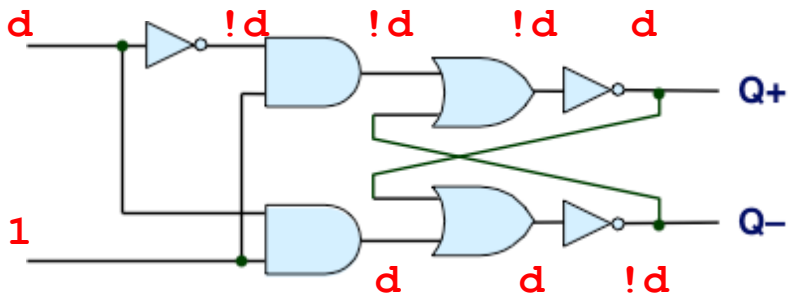
Building on top of R-S Latch

D Latch



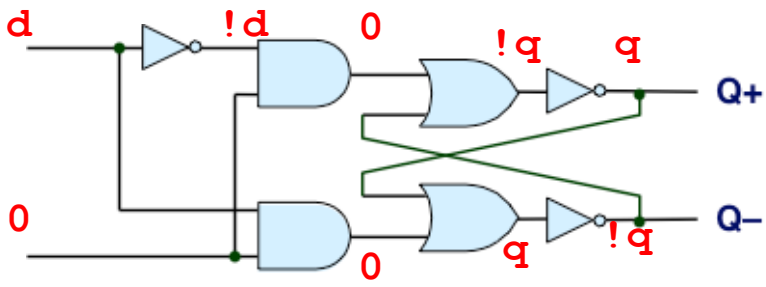
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

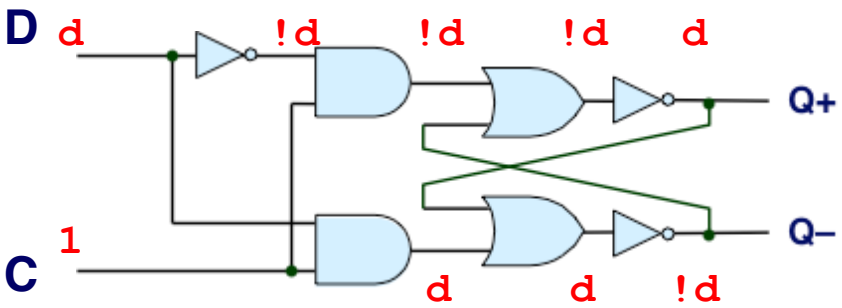
Holding Data



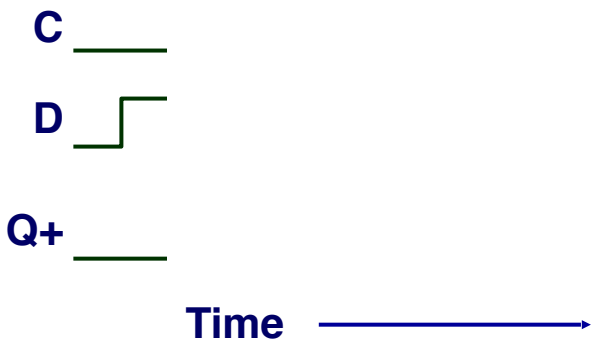
Q+ doesn't change with d

D-Latch is “Transparent”

Latching

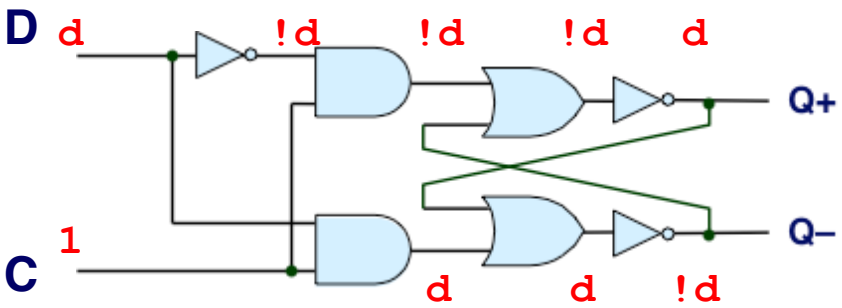


Changing D

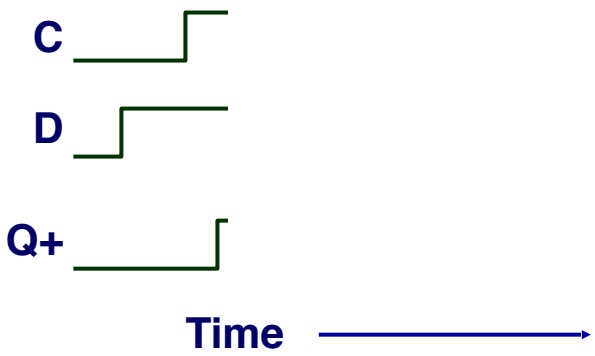


D-Latch is “Transparent”

Latching

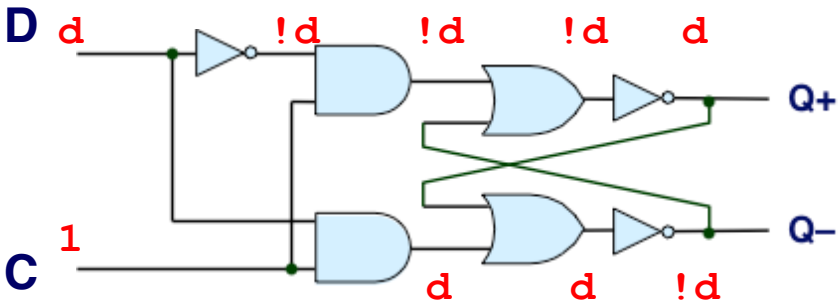


Changing D

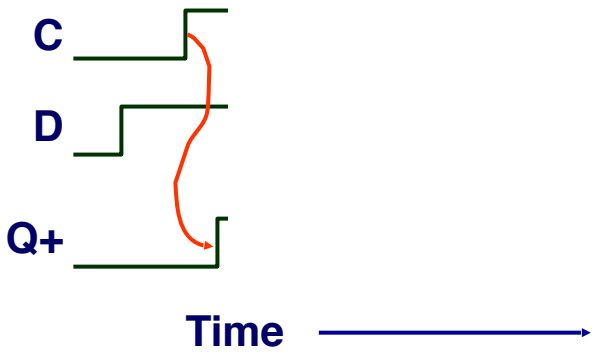


D-Latch is “Transparent”

Latching

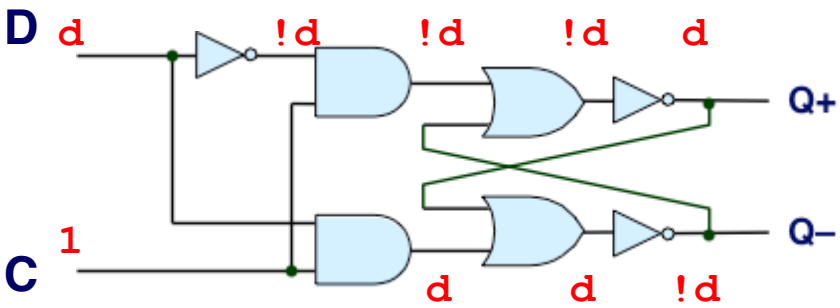


Changing D

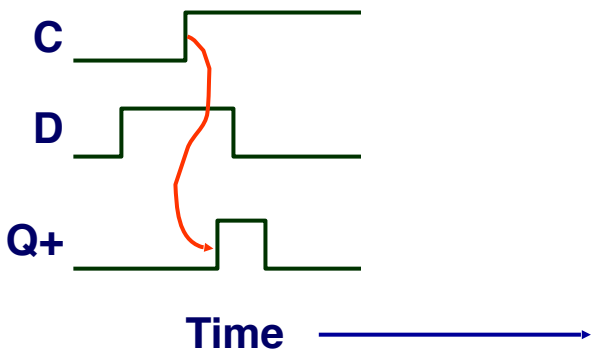


D-Latch is “Transparent”

Latching

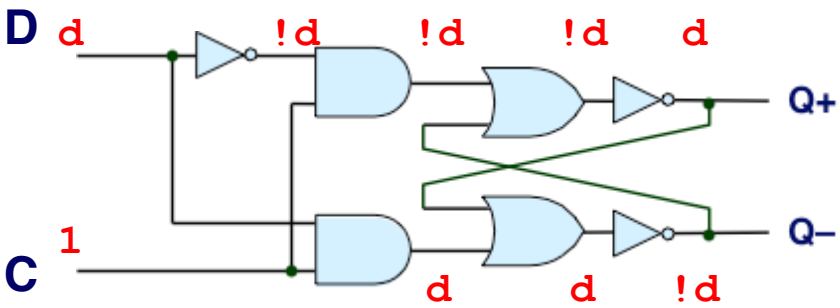


Changing D

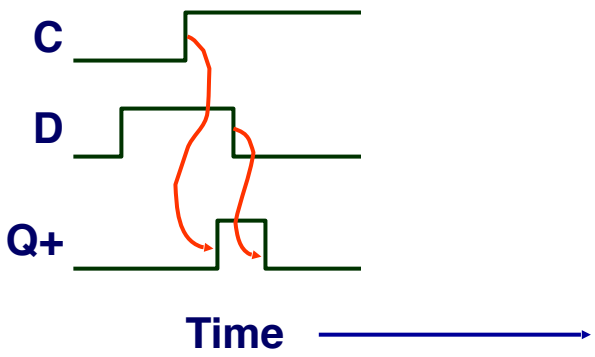


D-Latch is “Transparent”

Latching

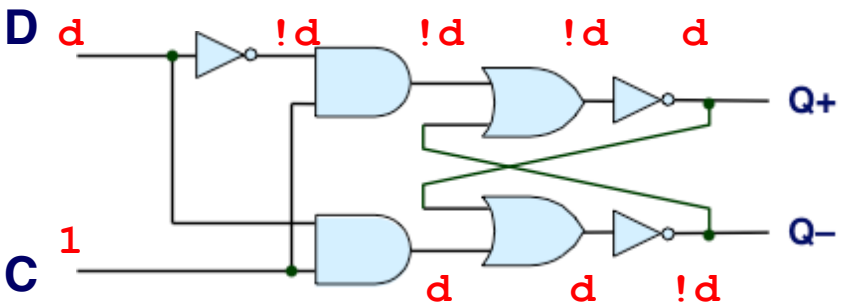


Changing D

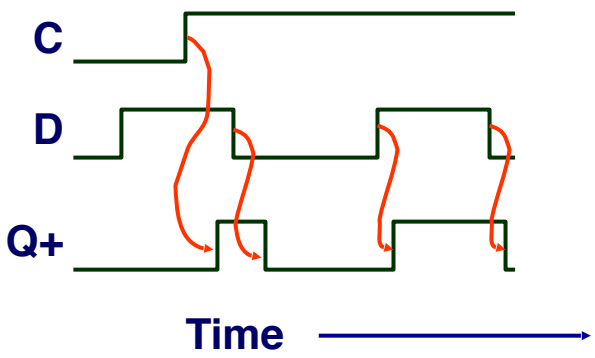


D-Latch is “Transparent”

Latching

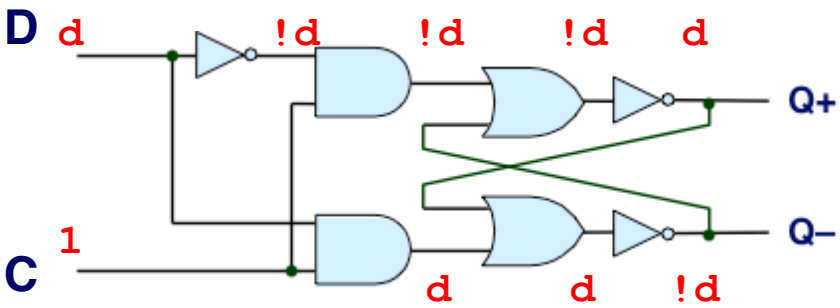


Changing D

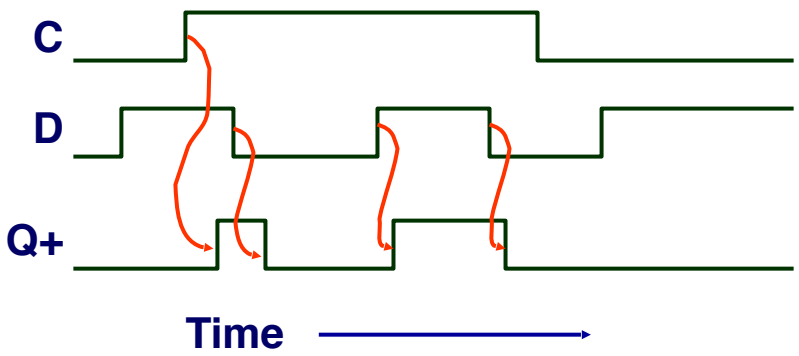


D-Latch is “Transparent”

Latching

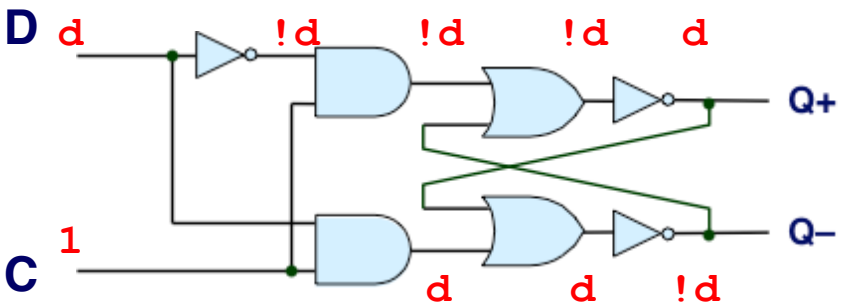


Changing D

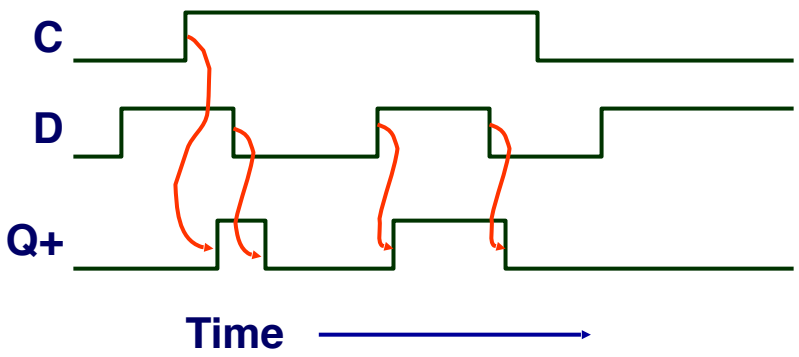


D-Latch is “Transparent”

Latching



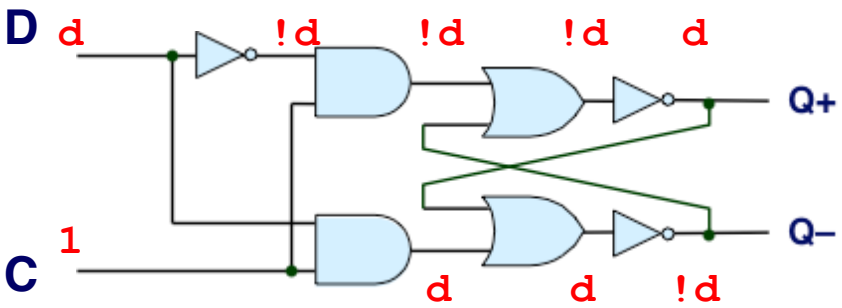
Changing D



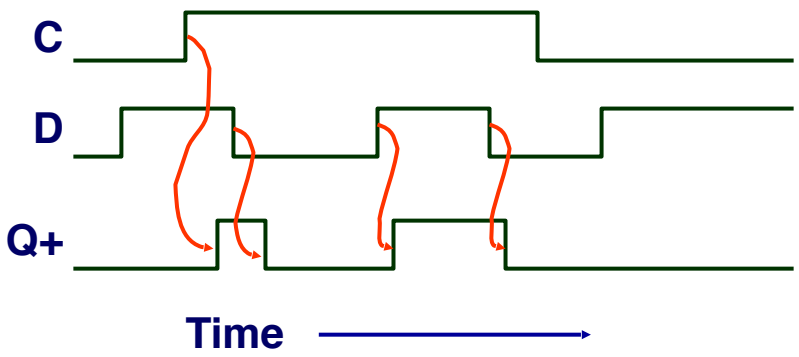
- When you want to store **d**, you have to first set **C** to 1, and then set **d**

D-Latch is “Transparent”

Latching



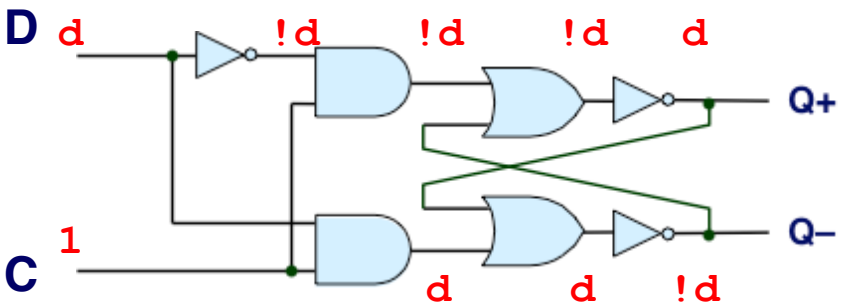
Changing D



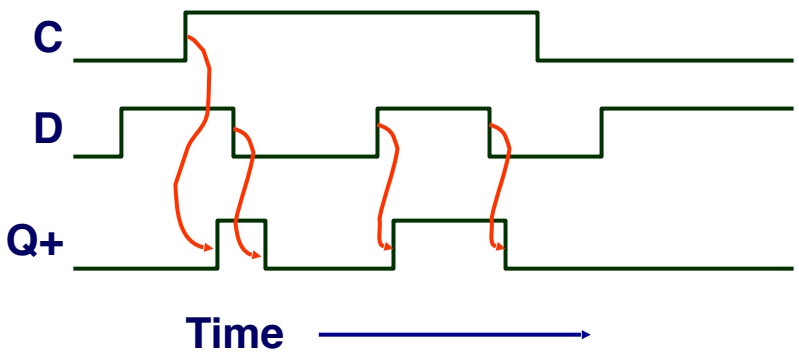
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+**. So hold C for a while until the signal is fully propagated

D-Latch is “Transparent”

Latching



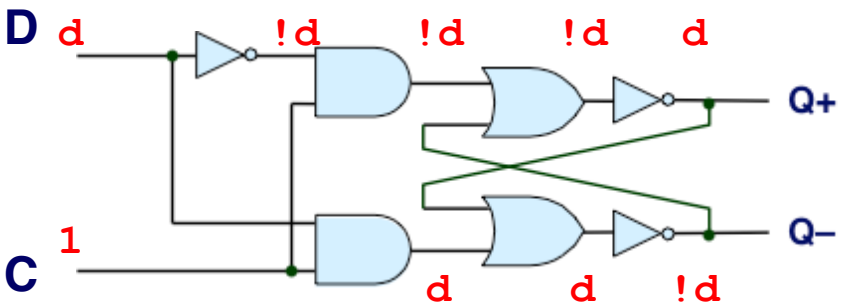
Changing D



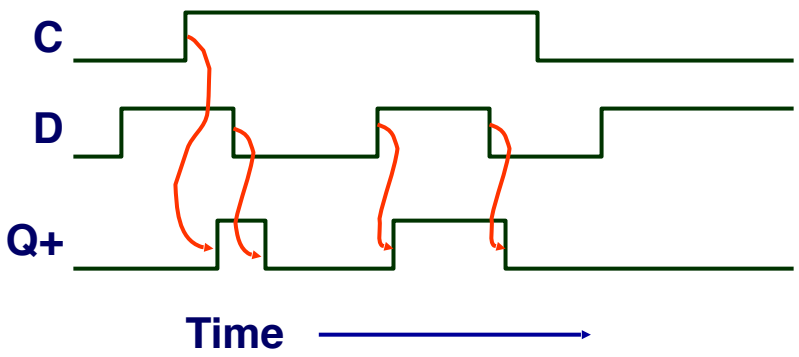
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+**. So hold C for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **C** goes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

D-Latch is “Transparent”

Latching



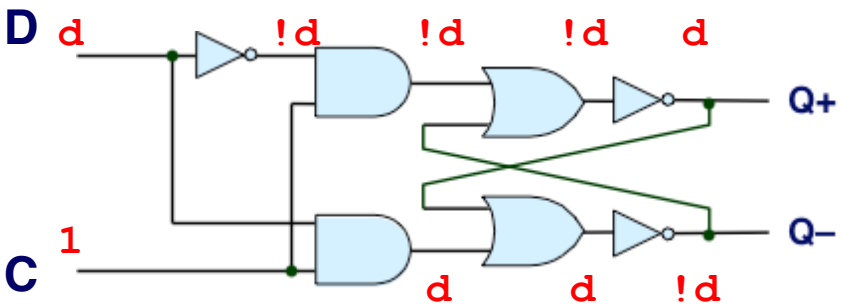
Changing D



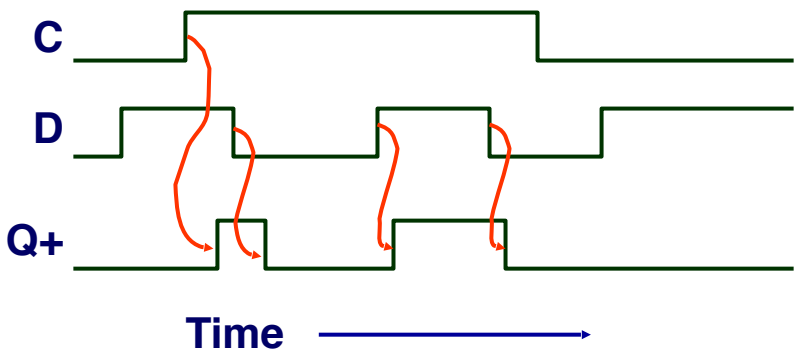
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+**. So hold C for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **C** goes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

D-Latch is "Transparent"

Latching

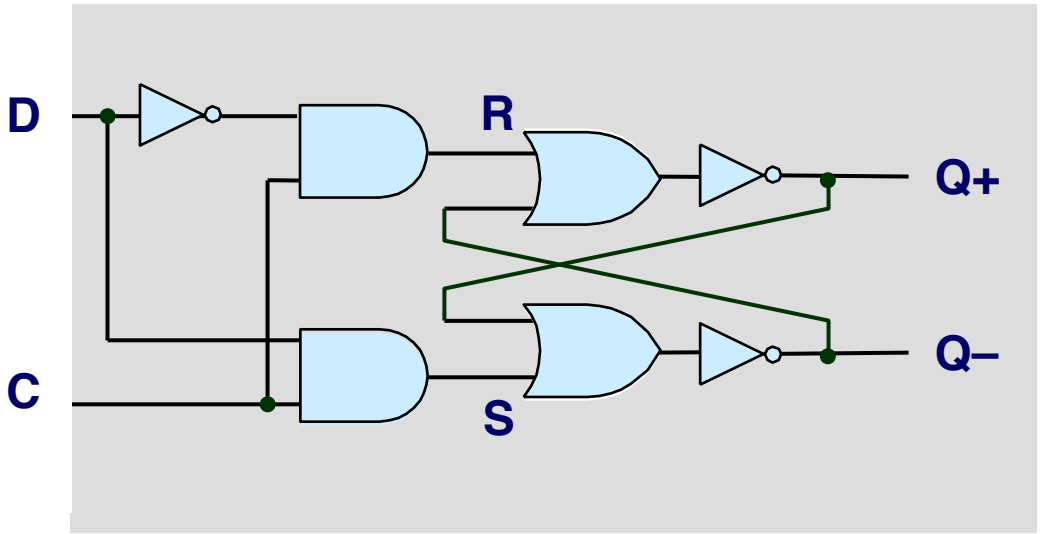


Changing D

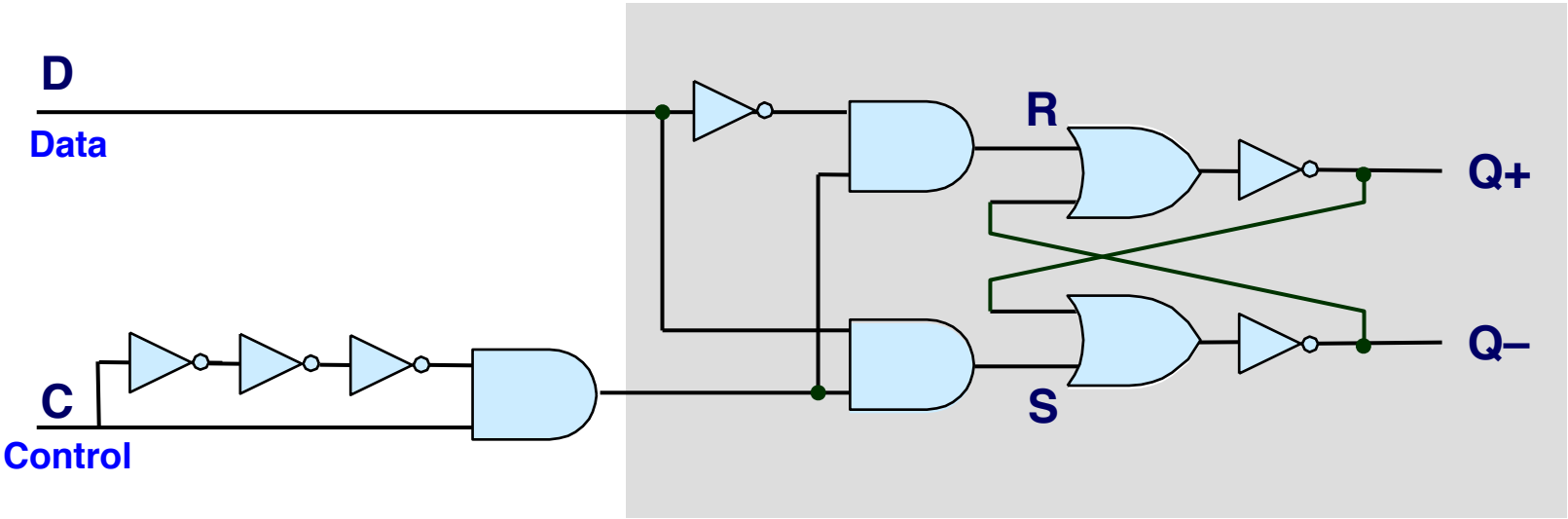


- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+**. So hold C for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D** as **C** goes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

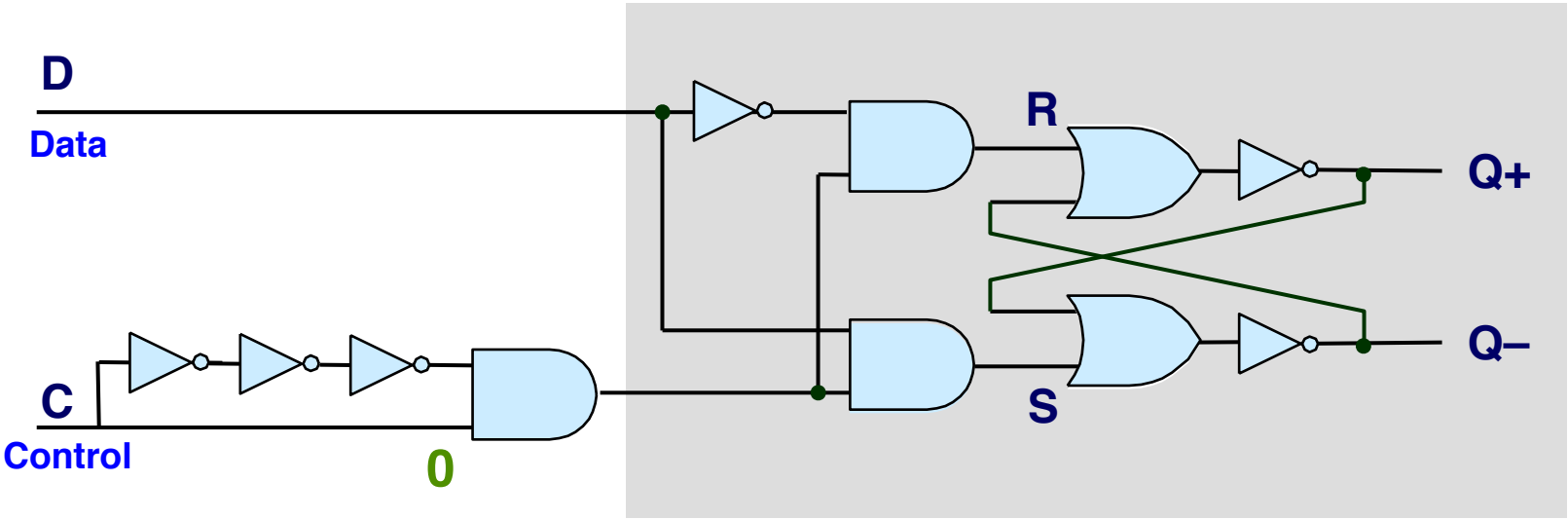
Edge-Triggered Latch (Flip-Flop)



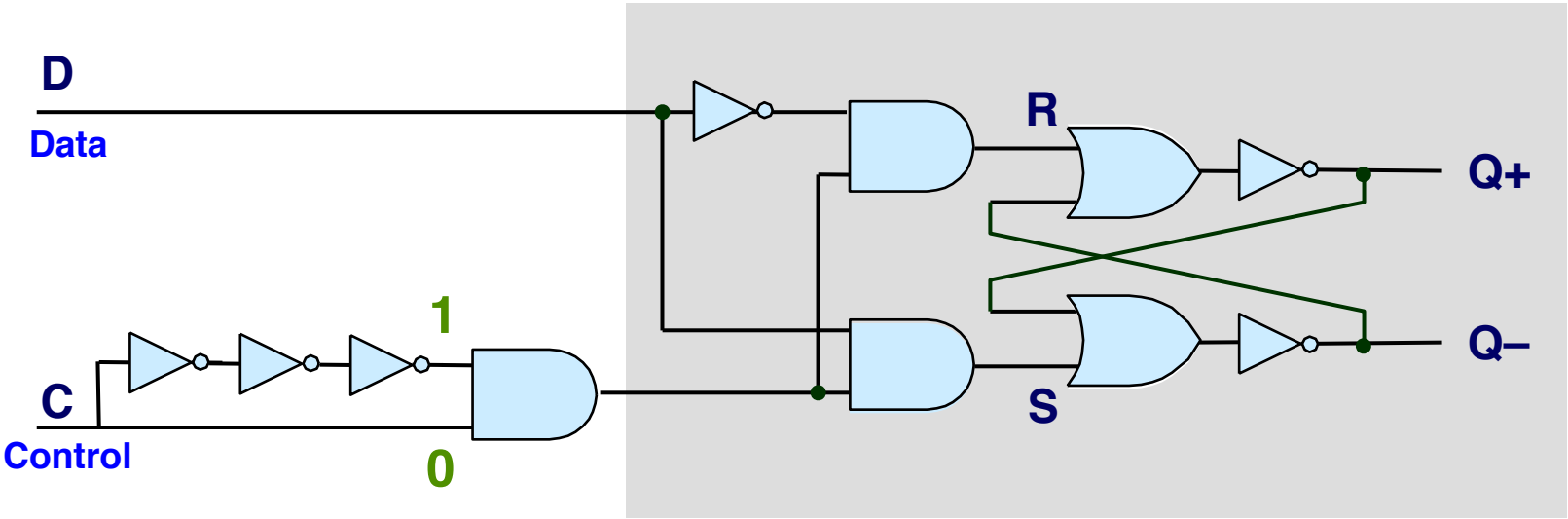
Edge-Triggered Latch (Flip-Flop)



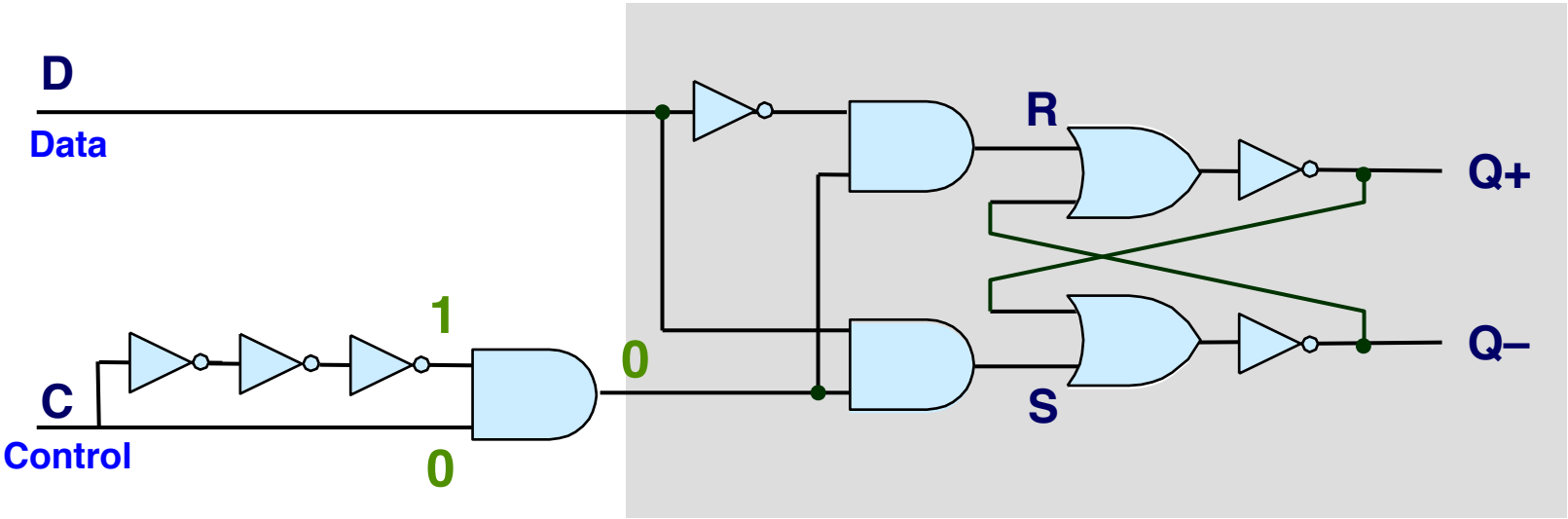
Edge-Triggered Latch (Flip-Flop)



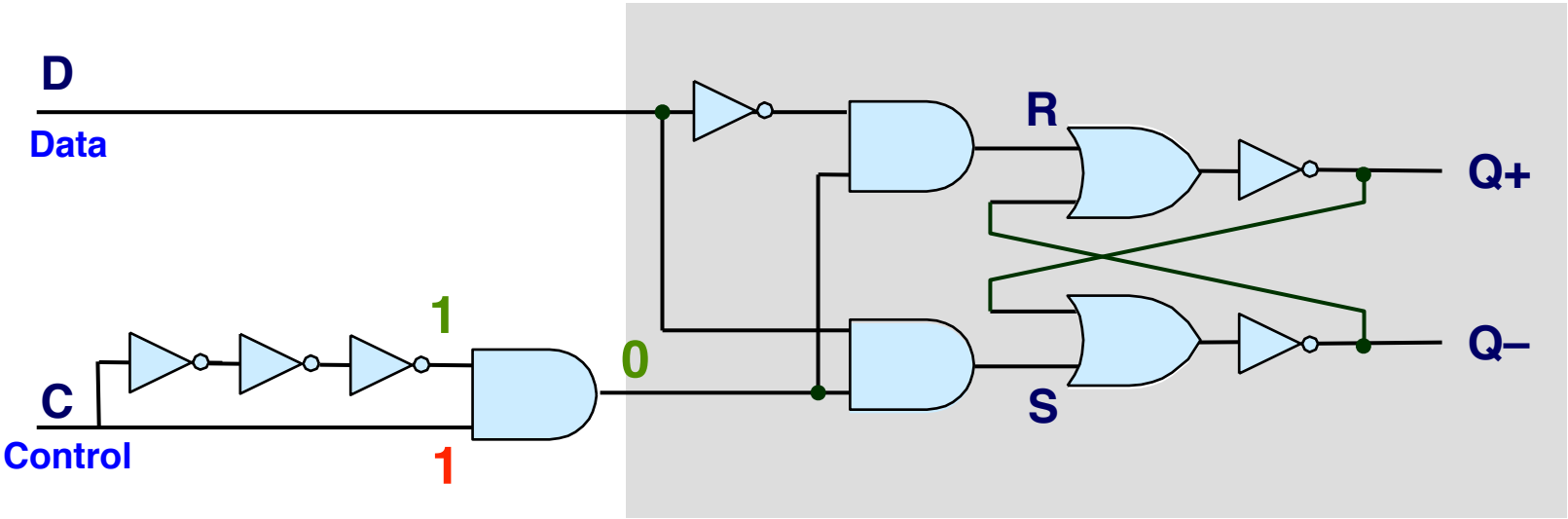
Edge-Triggered Latch (Flip-Flop)



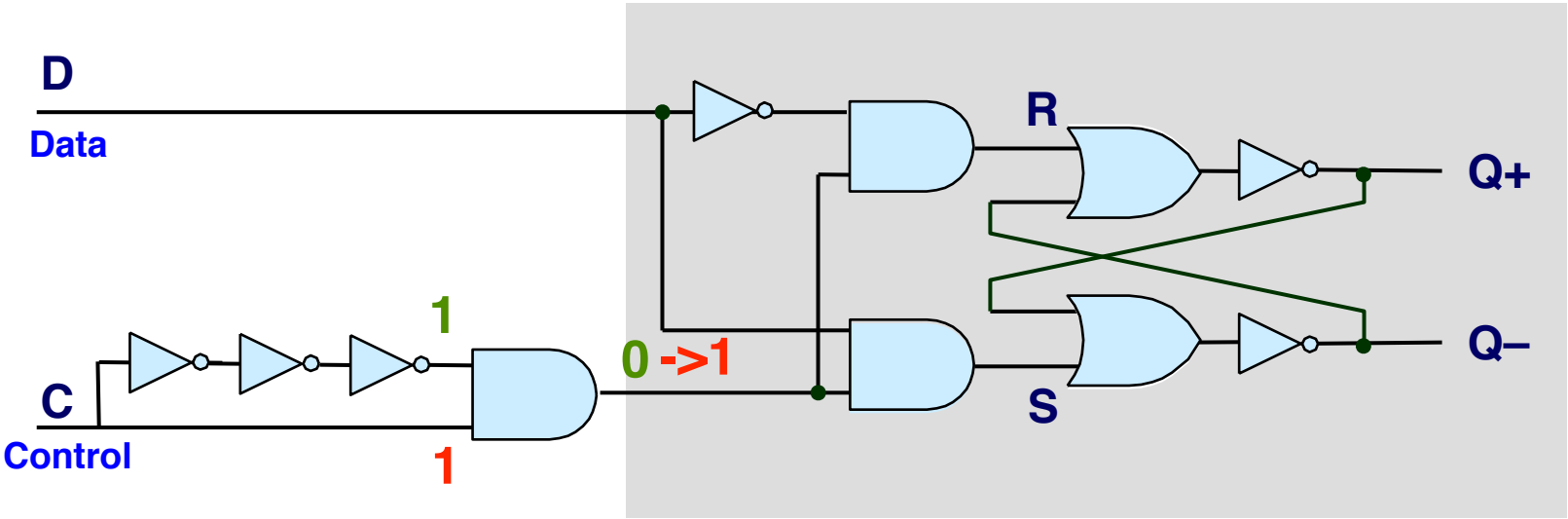
Edge-Triggered Latch (Flip-Flop)



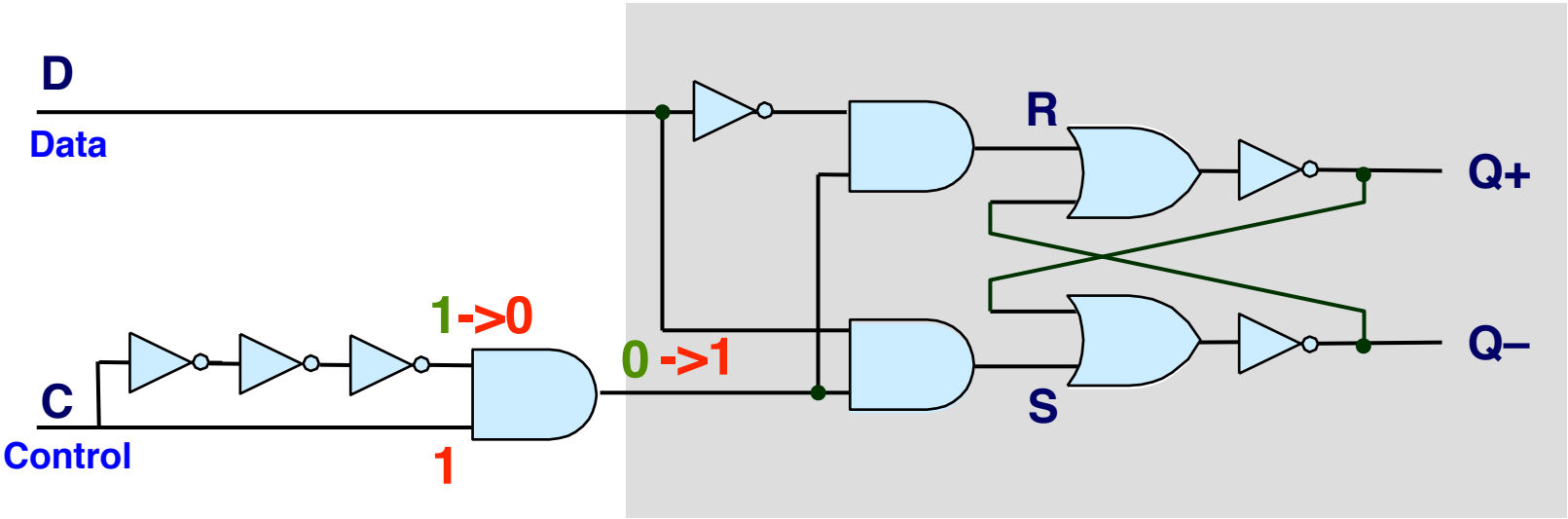
Edge-Triggered Latch (Flip-Flop)



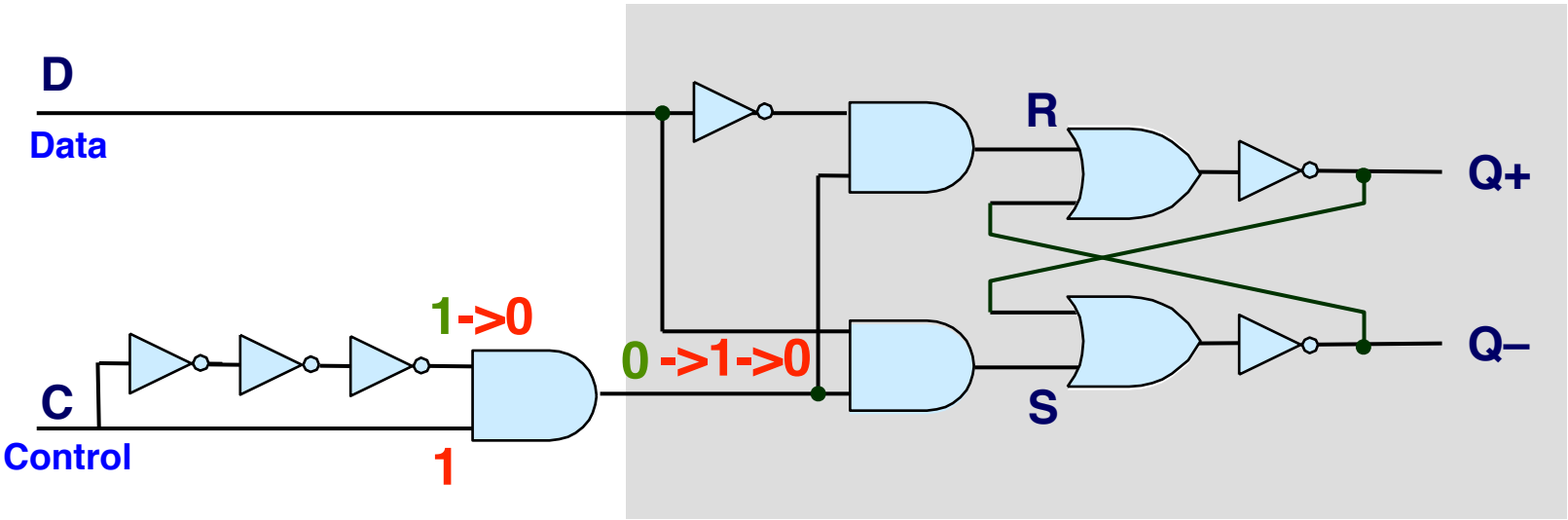
Edge-Triggered Latch (Flip-Flop)



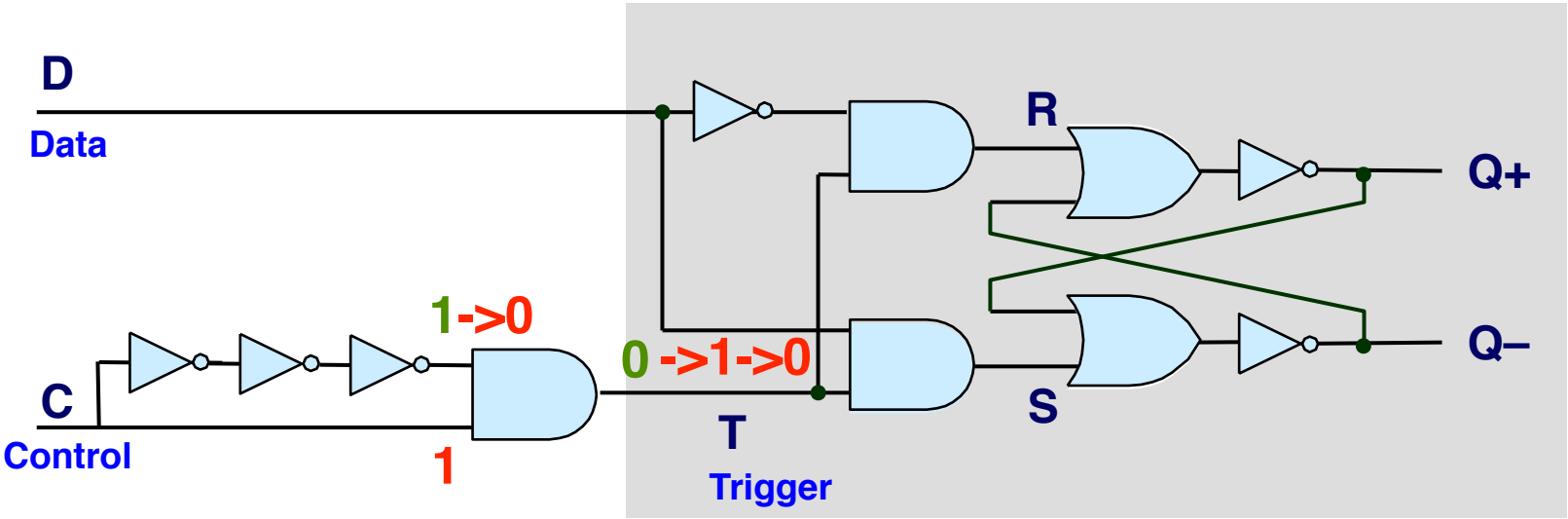
Edge-Triggered Latch (Flip-Flop)

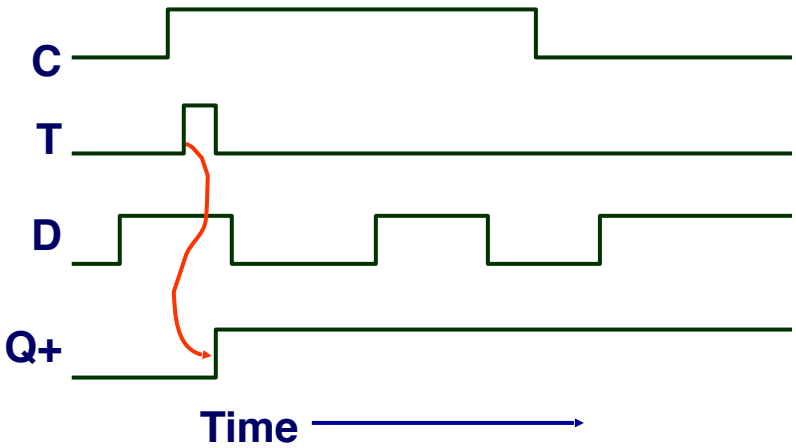


Edge-Triggered Latch (Flip-Flop)

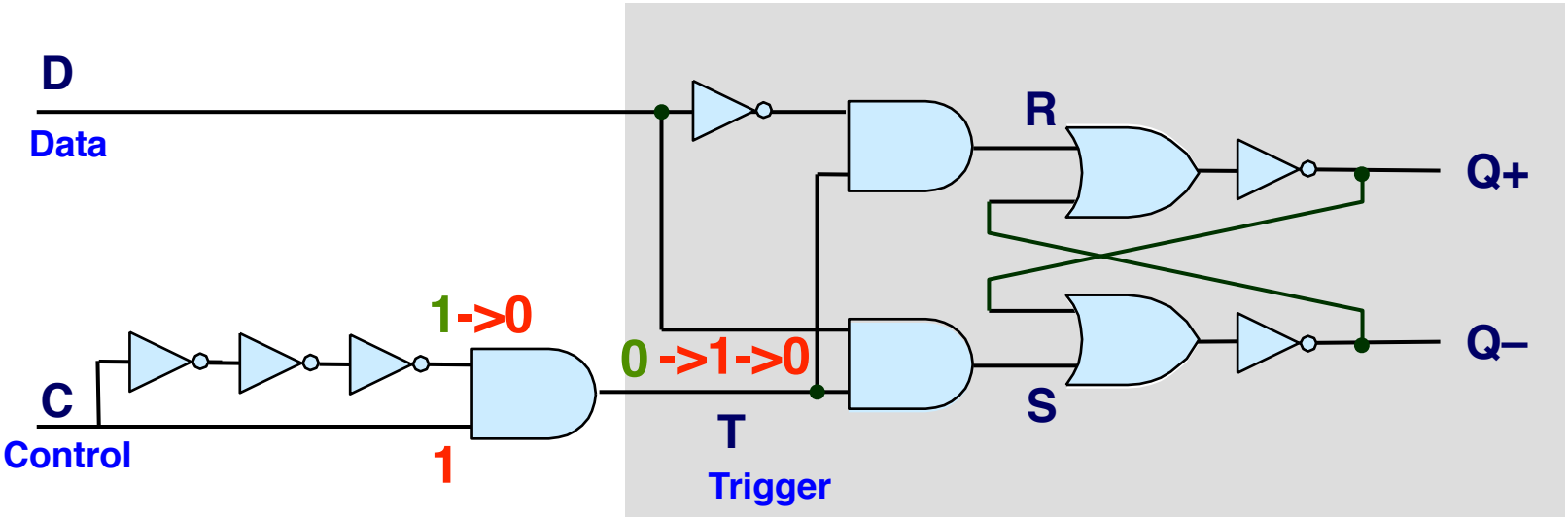


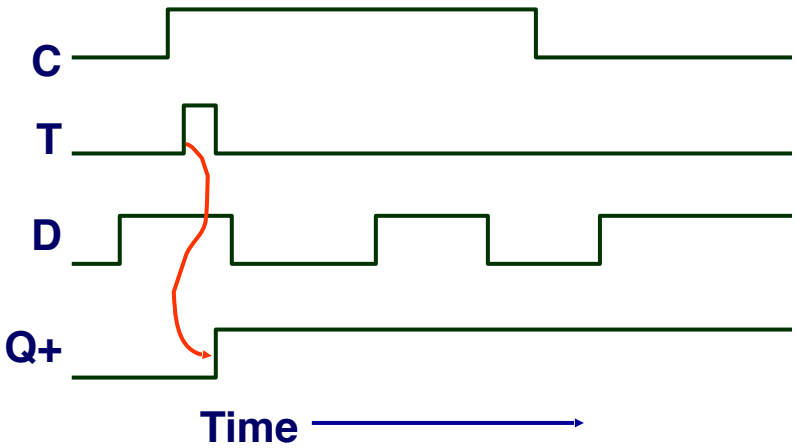
Edge-Triggered Latch (Flip-Flop)





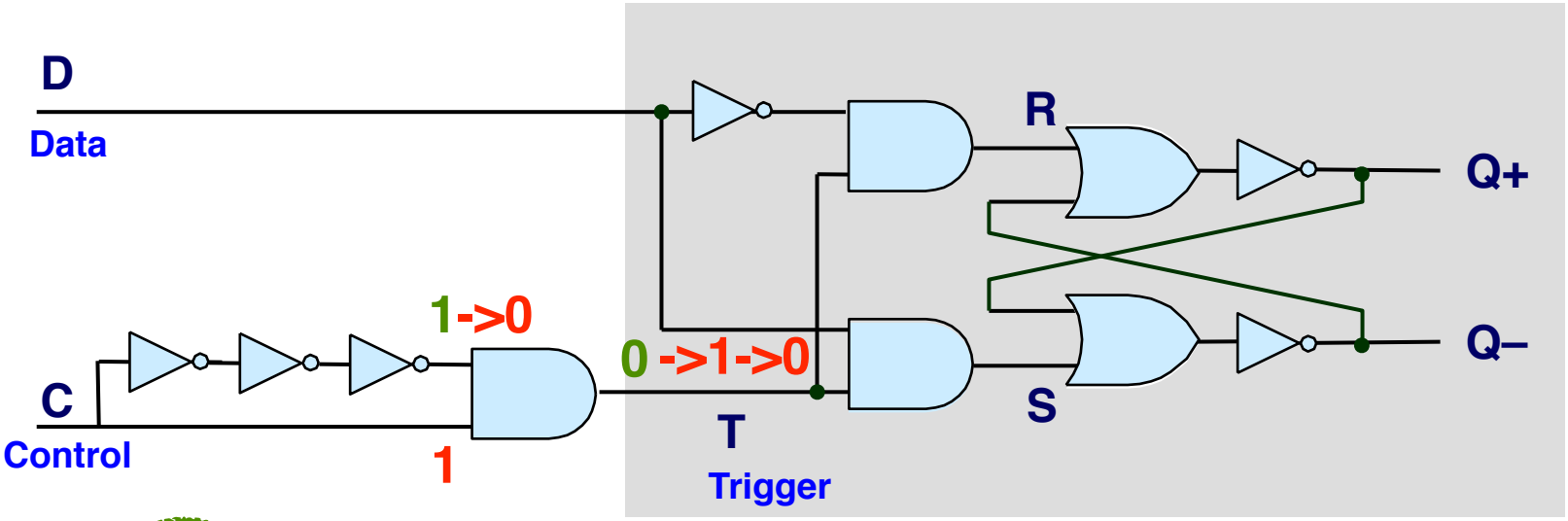
Edge-Triggered Latch (Flip-Flop)

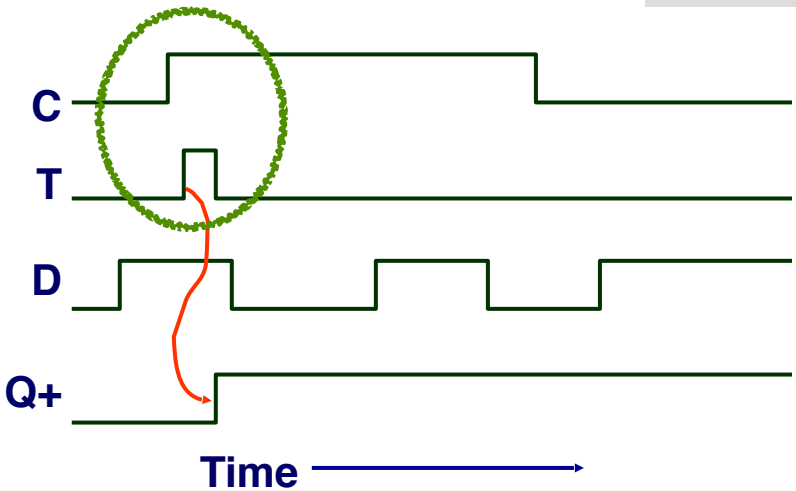




- Flip-flop: Only latches data for a brief period

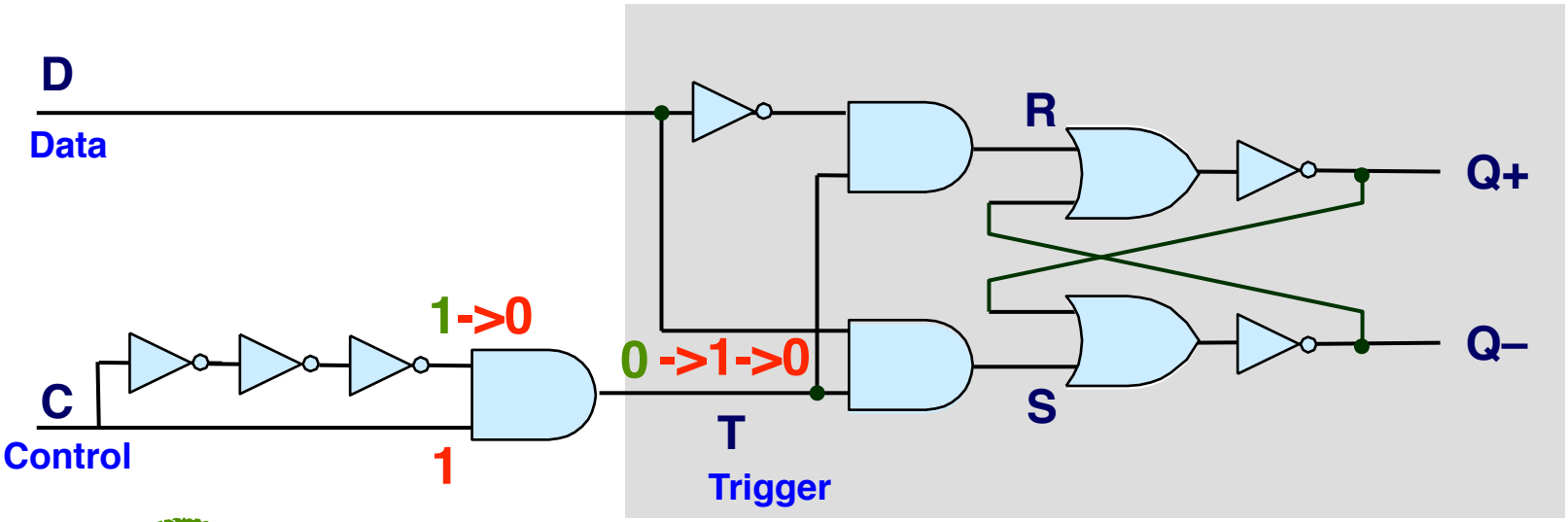
Edge-Triggered Latch (Flip-Flop)

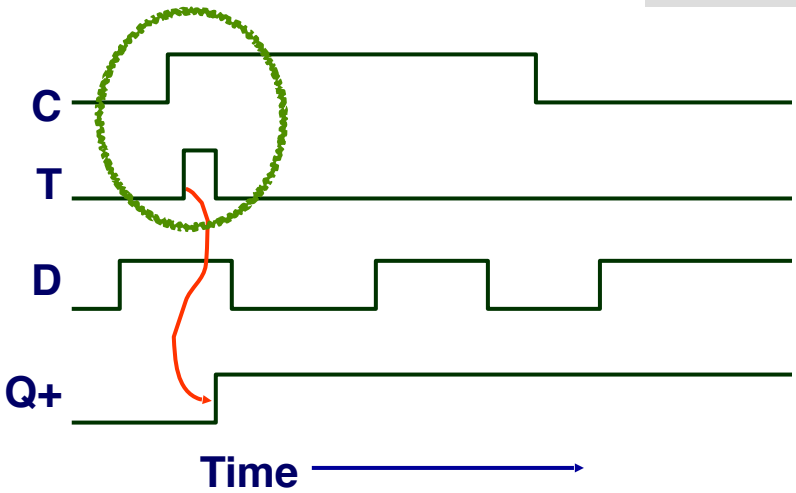




- Flip-flop: Only latches data for a brief period
- Value latched depends on data as C **rises** (i.e., 0–>1); usually called at the **rising edge** of C

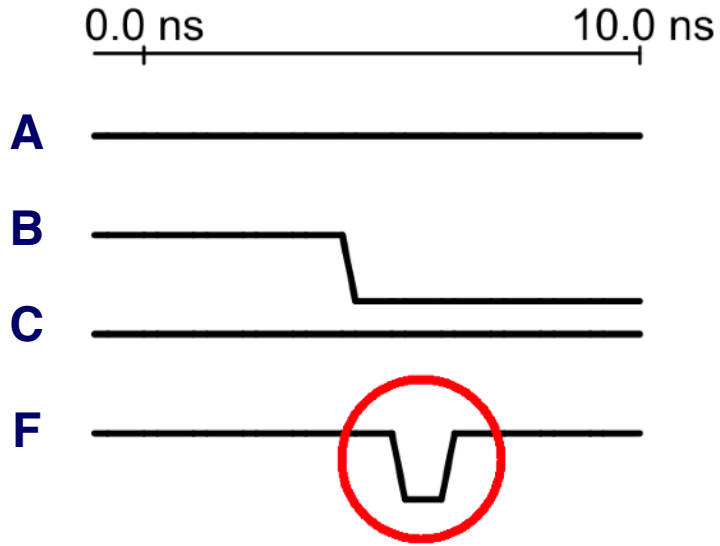
Edge-Triggered Latch (Flip-Flop)

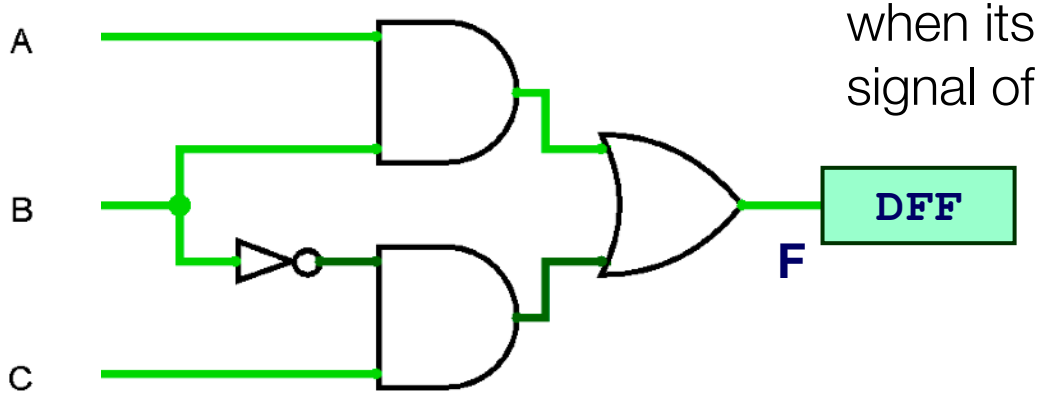




- Flip-flop: Only latches data for a brief period
- Value latched depends on data as C **rises** (i.e., 0–>1); usually called at the **rising edge** of C
- Output remains stable at all other times

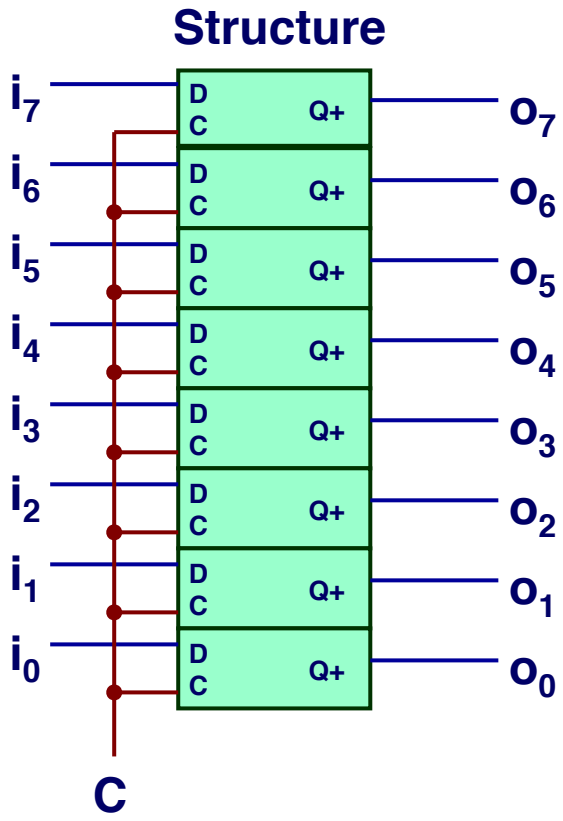
Why Use a Flip-Flop?





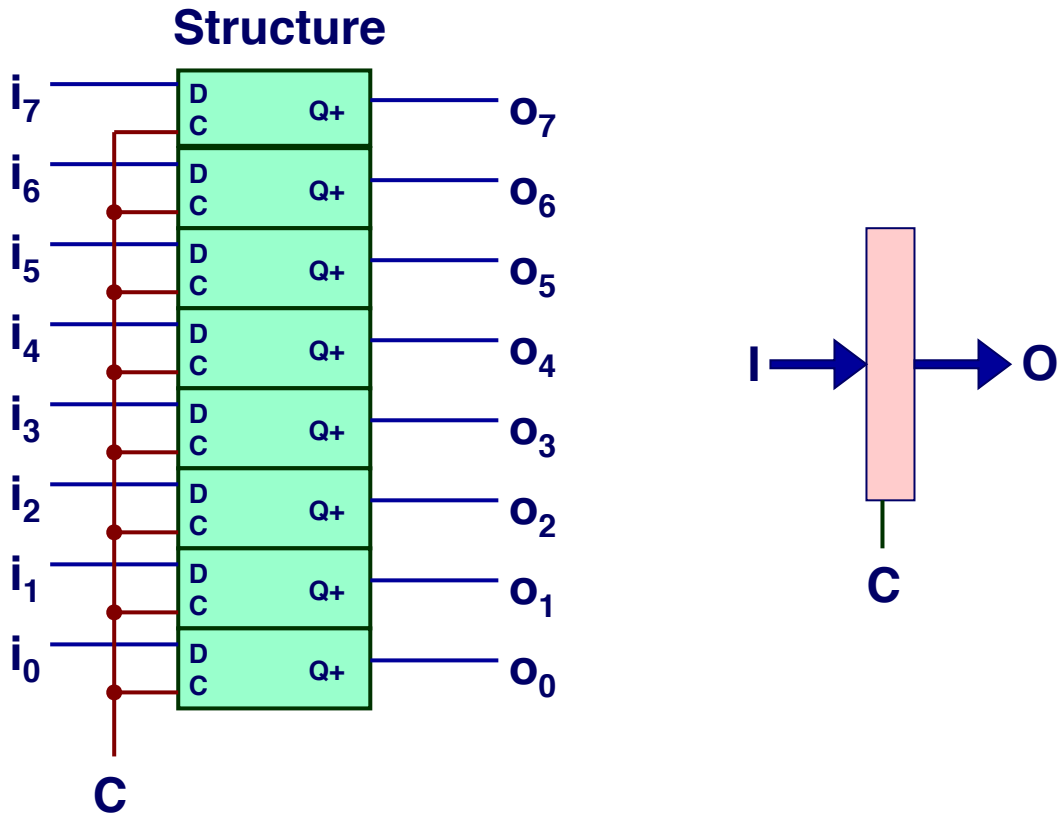
- Because the data we want to store might be temporarily changing before it settles down (due to glitch). We want to capture only the final value.
- If we had a transparent D latch, the latched value would change with F, i.e., temporal glitches will be temporarily stored as well.
- With a flip flop,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

Registers



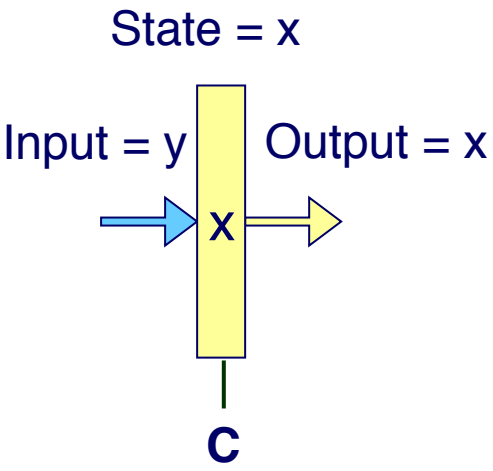
- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

Registers

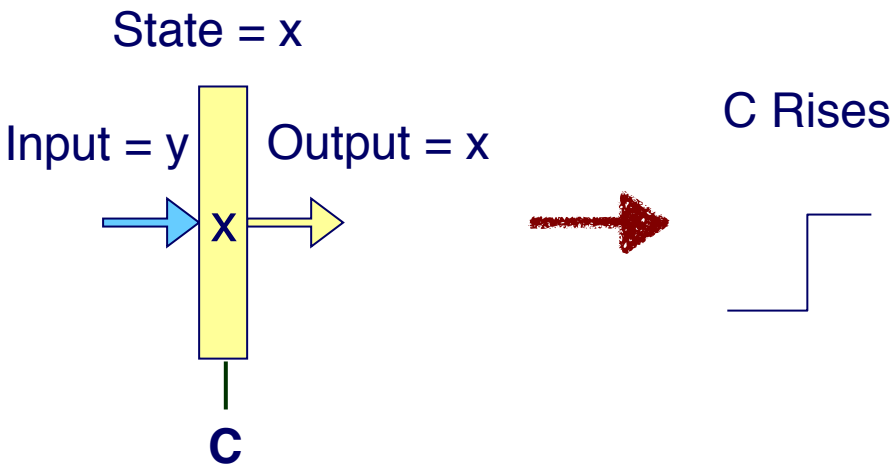


- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

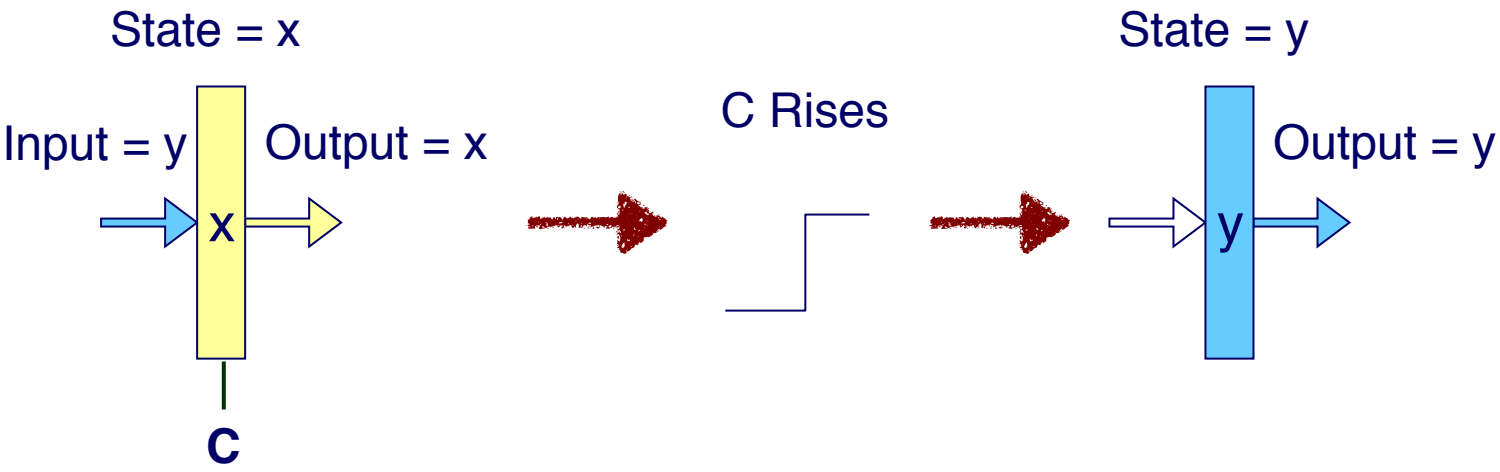
Register Operation



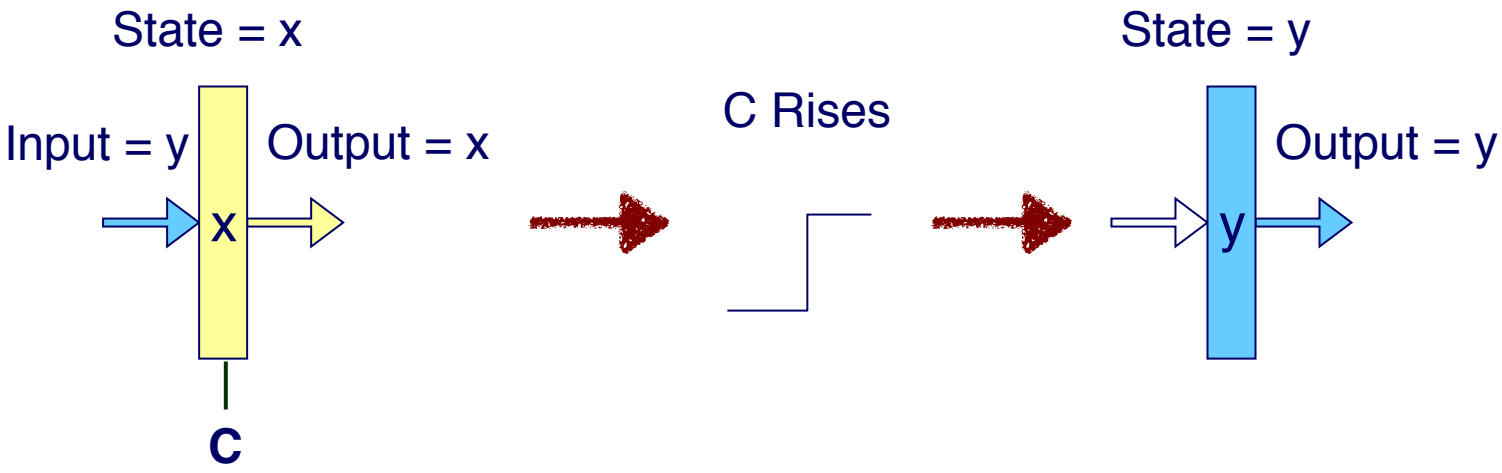
Register Operation



Register Operation

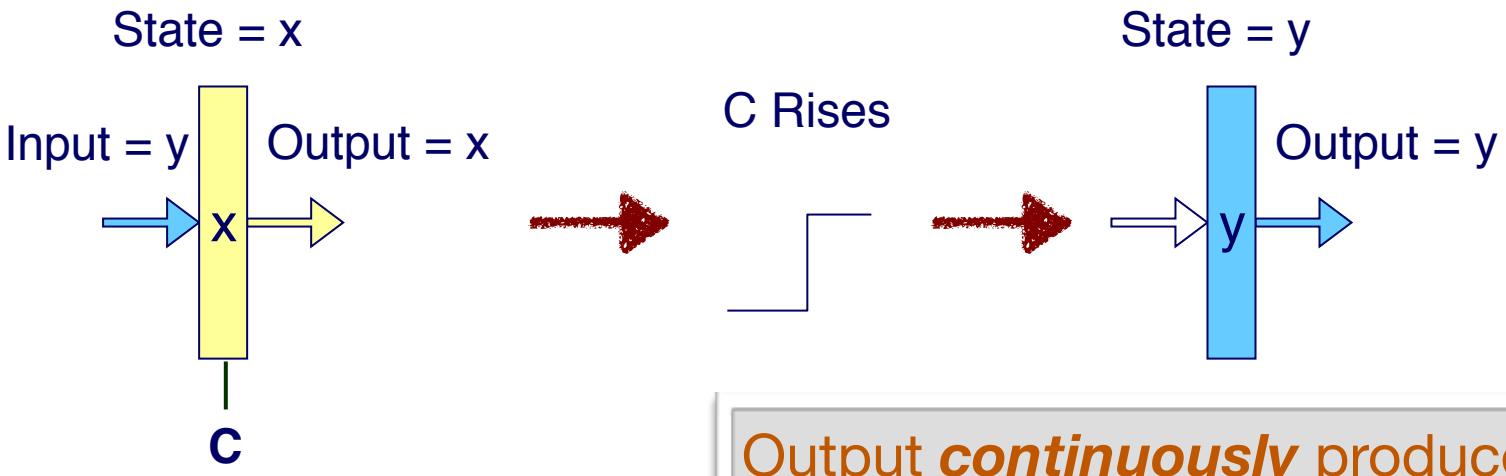


Register Operation



- 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

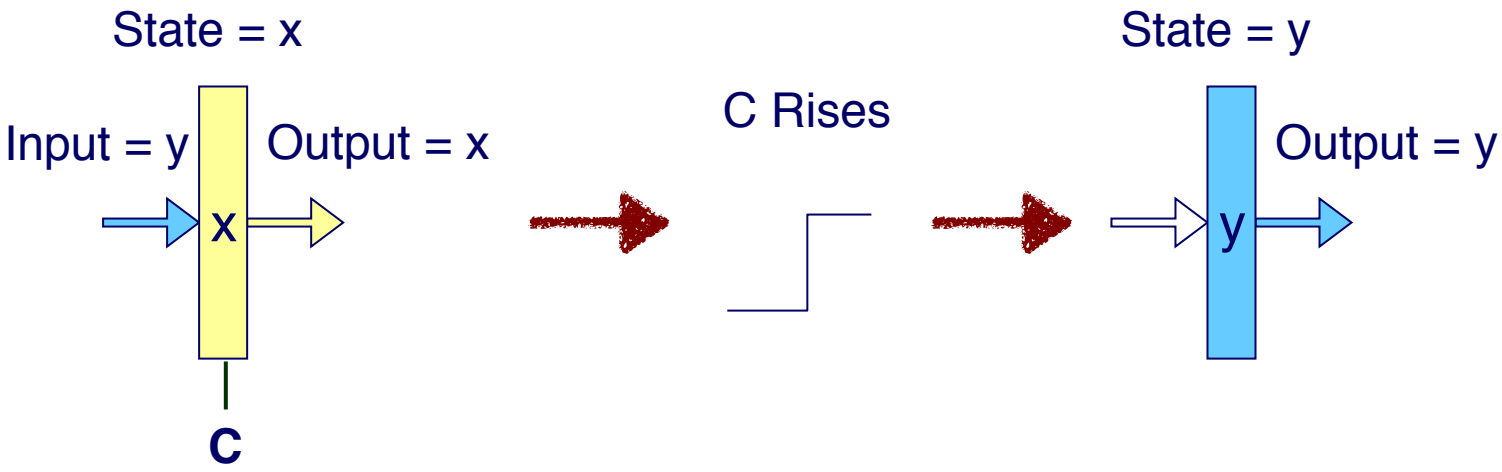
Register Operation



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

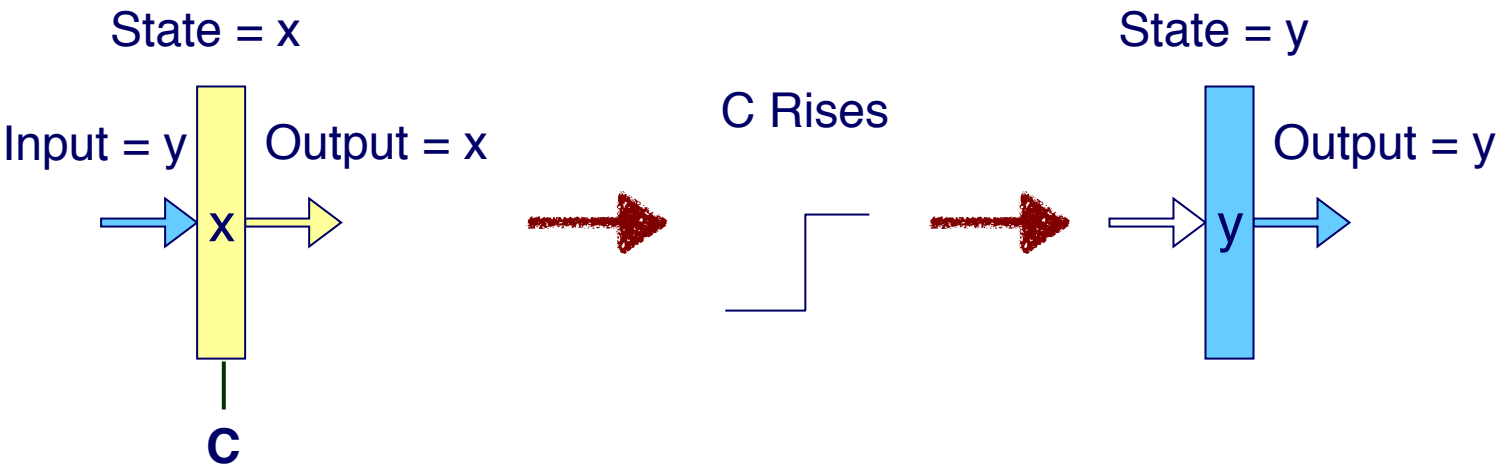
- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you'd better compute the input before the C signal rises if you want to store the input data to the register

Clock Signal



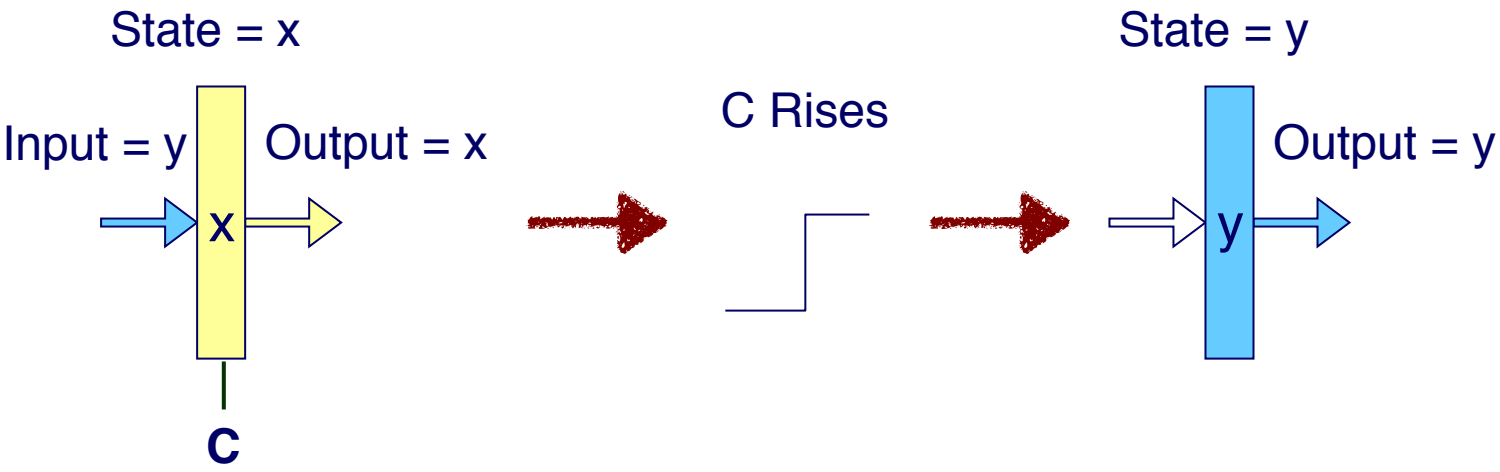
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

Clock Signal

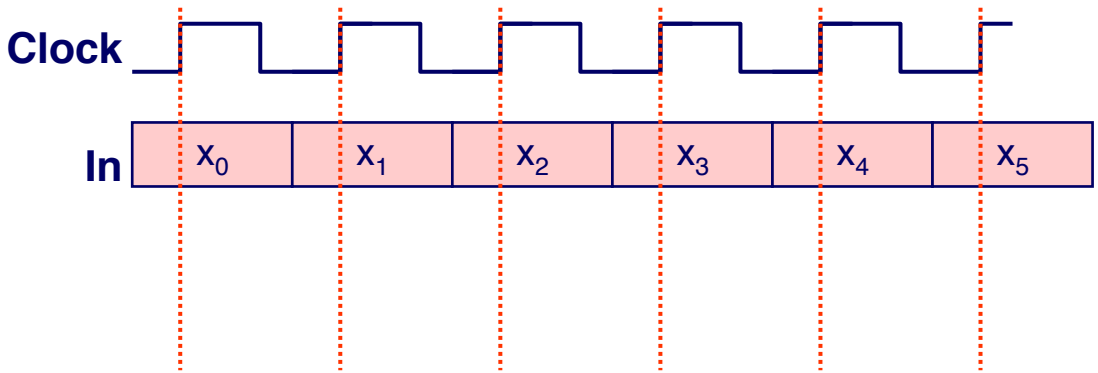


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

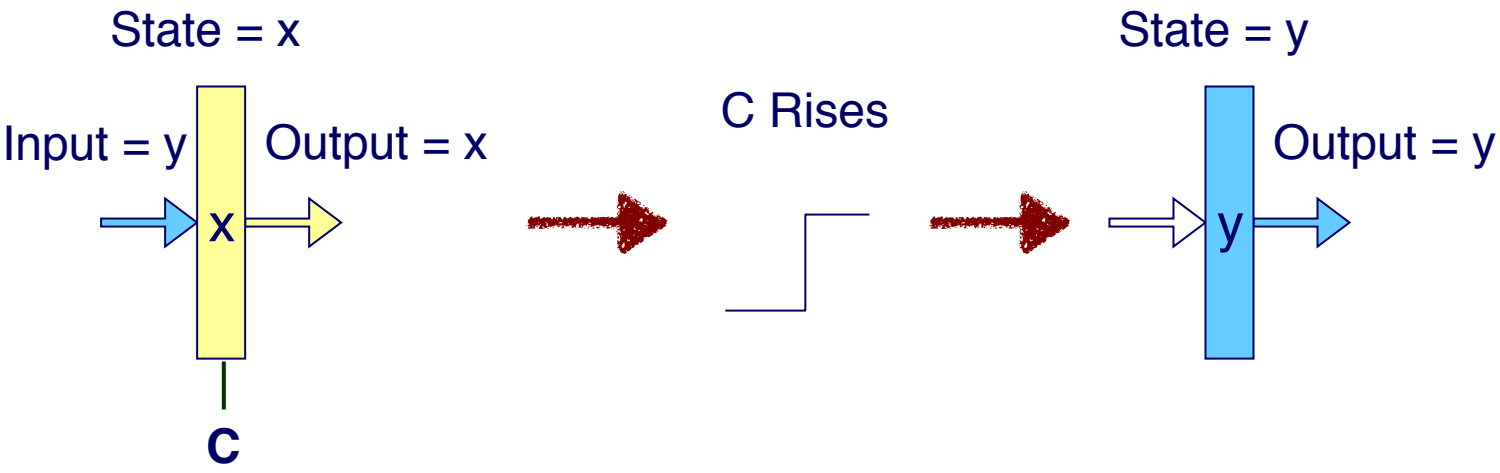
Clock Signal



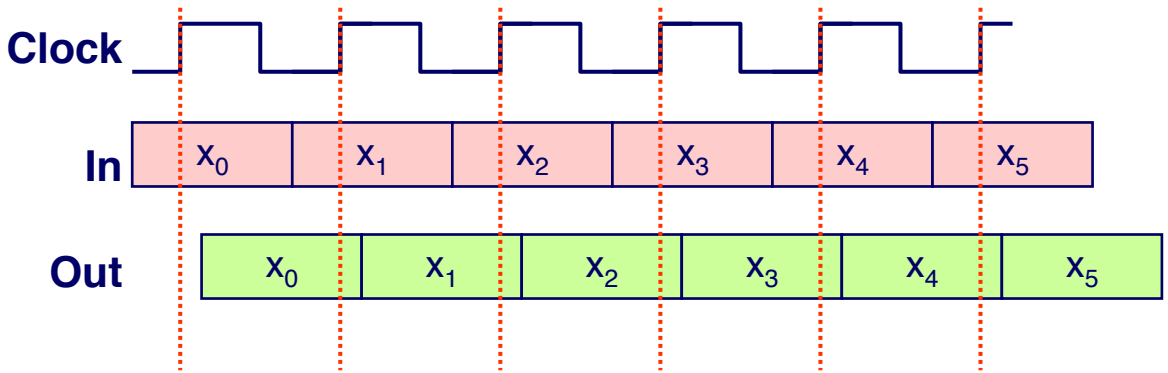
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



Clock Signal

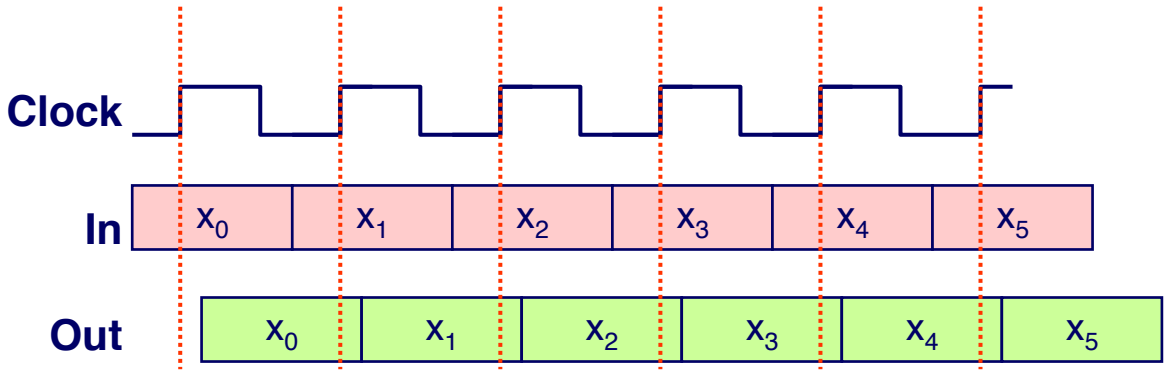


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



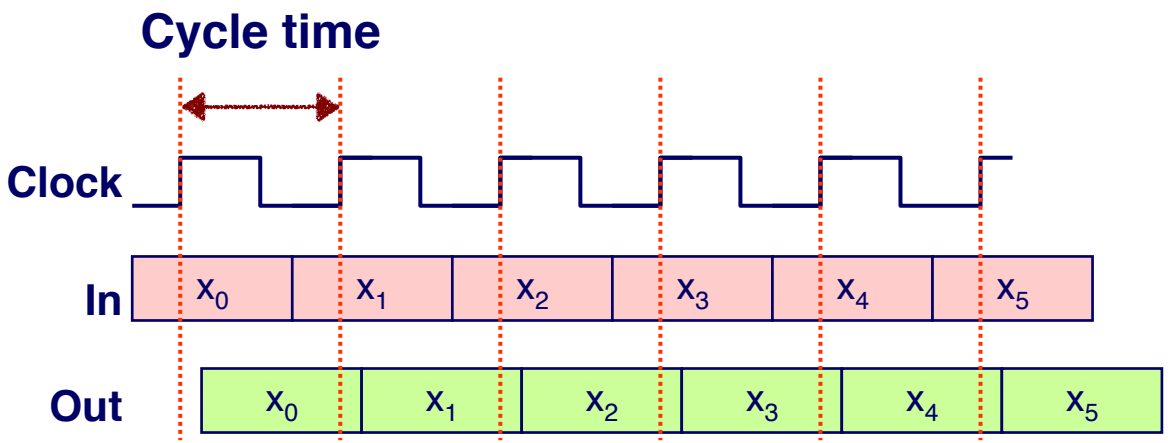
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.



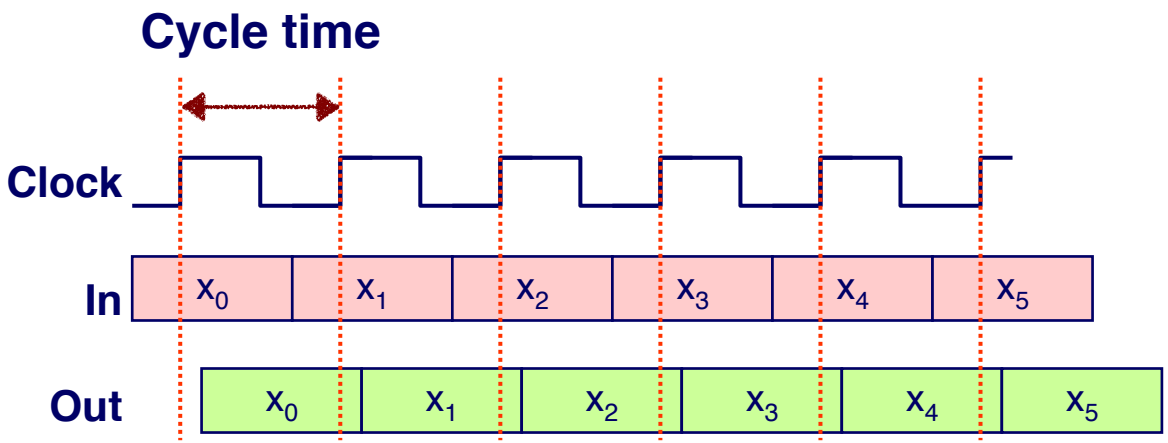
Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.



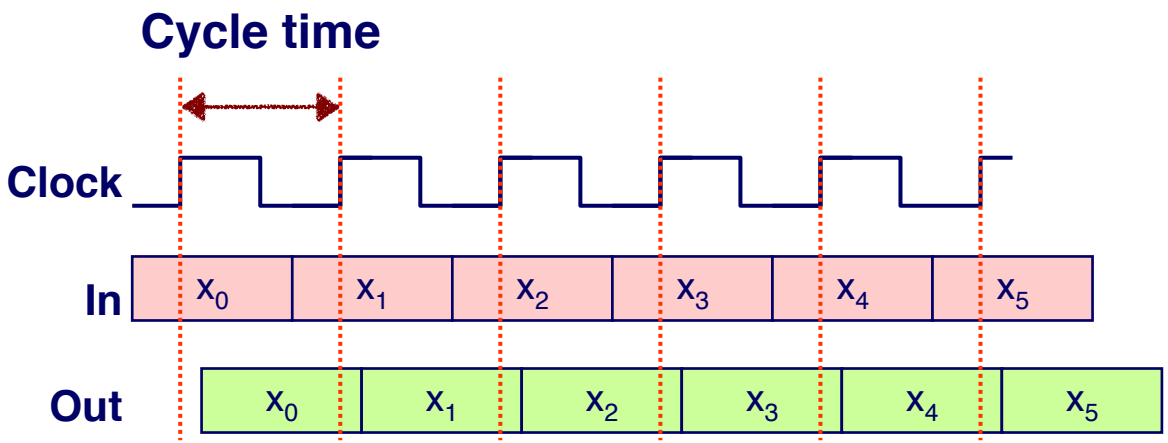
Clock Signal

- 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.



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



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 ns