

# **CSC 252: Computer Organization**

## **Spring 2020: Lecture 11**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

# Announcement

- Programming assignment 3 is out
  - Details: <https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment3.html>
  - Due on **Feb. 28**, 11:59 PM
  - You (may still) have 3 slip days

17	18	19	20	21	22
			Today		
24	25	26	27	28	29
				Due	

# Announcement

- Grades for lab2 are posted.
- If you think there are some problems
  - Take a deep breath
  - Tell yourself that the teaching staff like you, not the opposite
  - Email/go to Shuang or Sudhanshu's office hours and explain to them why you should get more points, and they will fix it for you

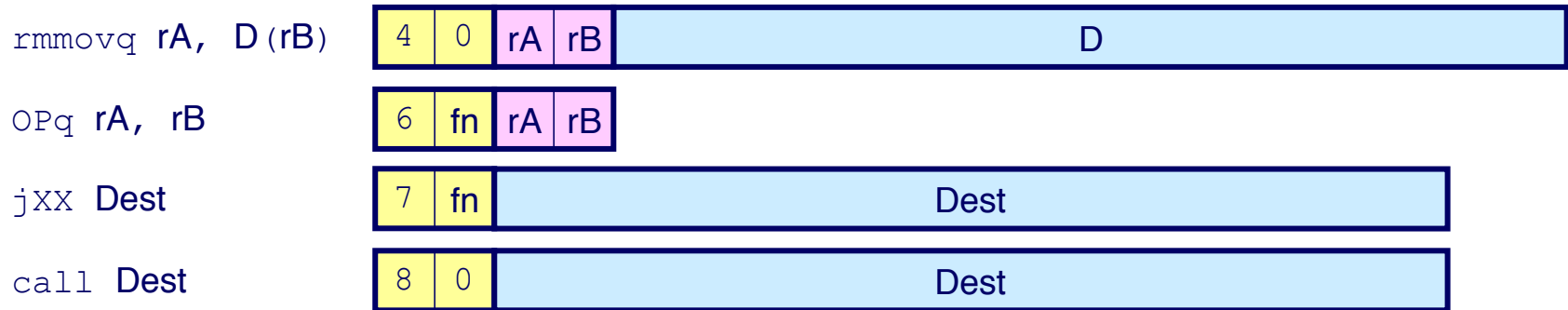
# Announcement

- 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.

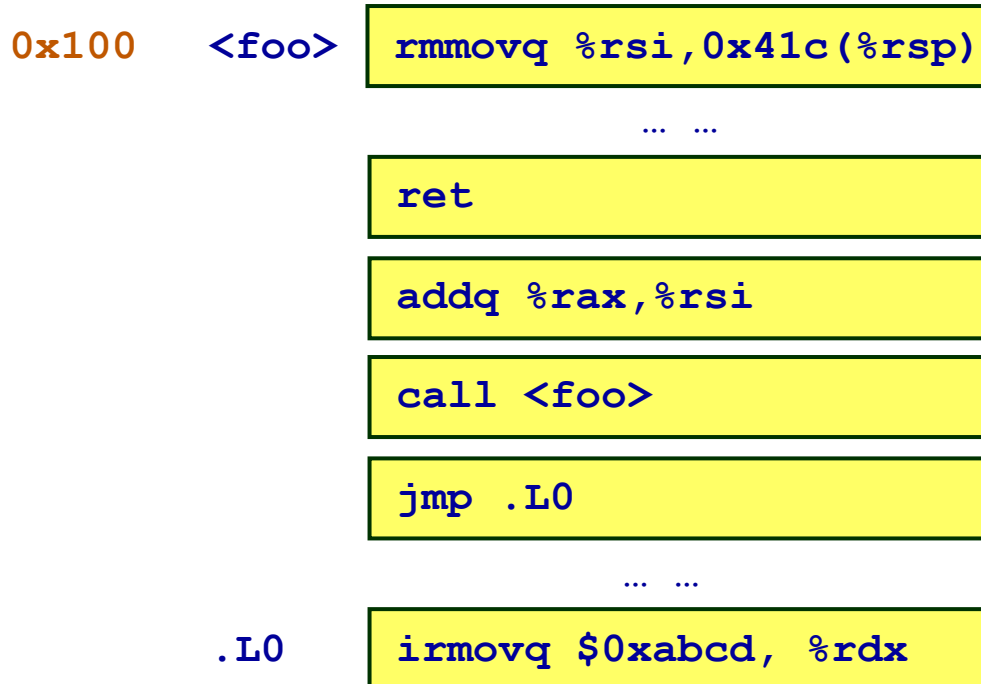
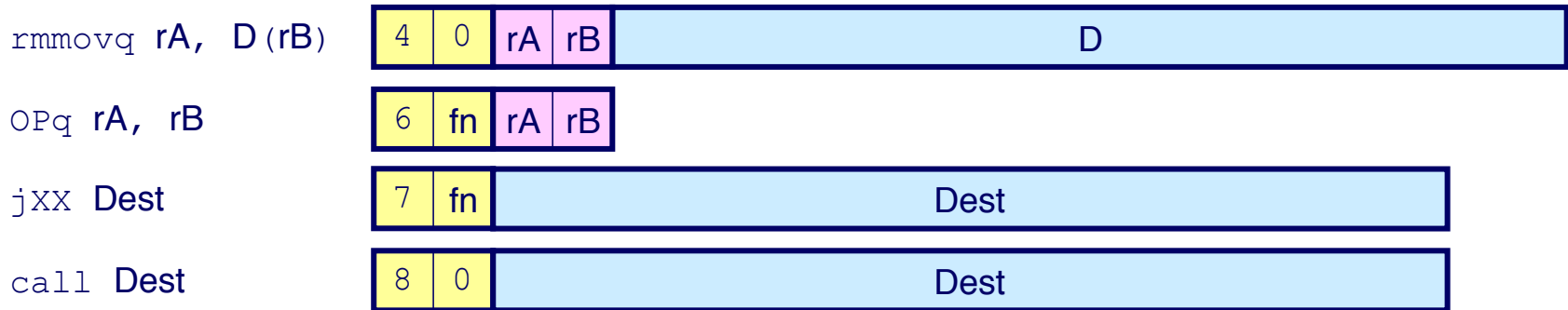
# Y86 Instruction Encoding

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
rmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

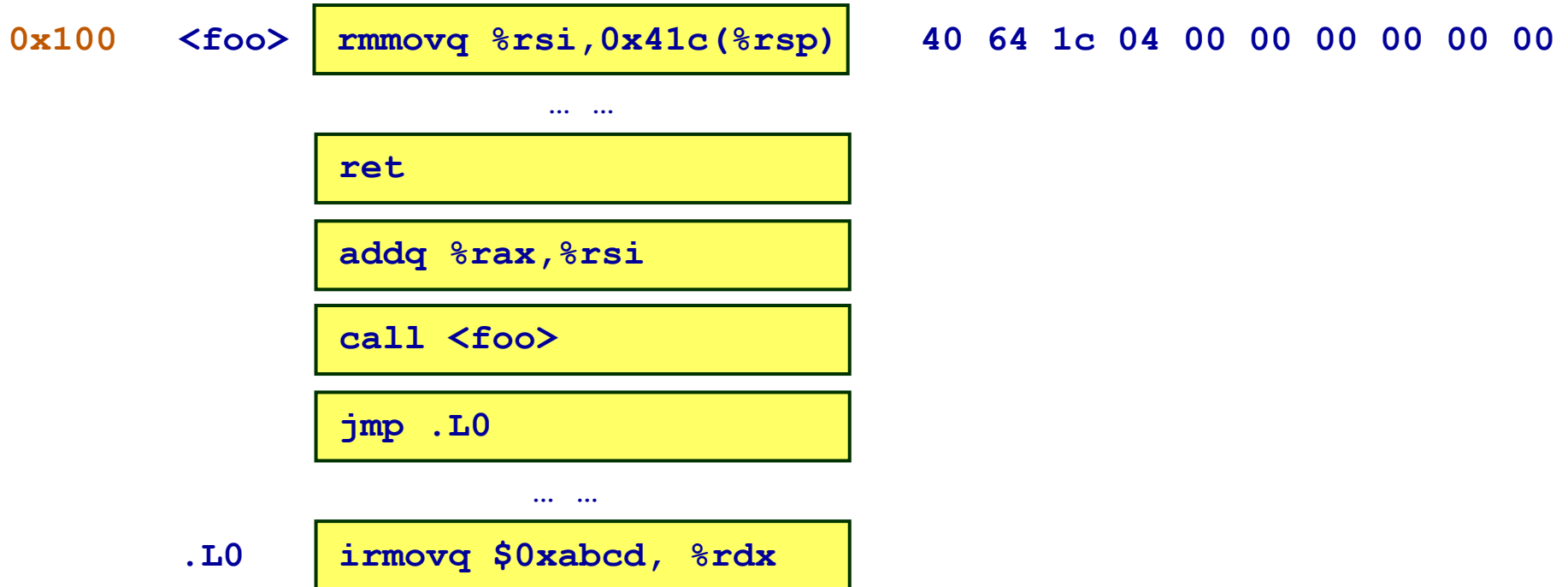
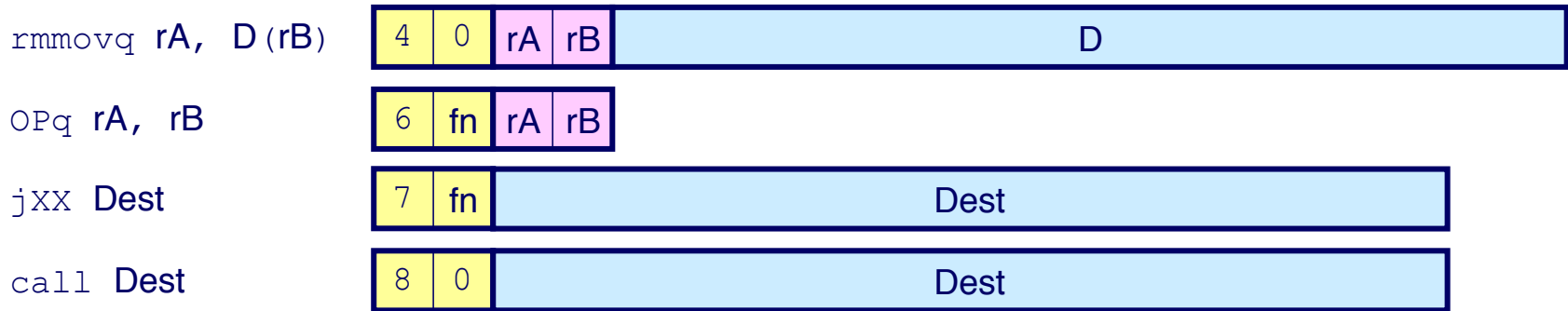
# How Does An Assembler Work?



# How Does An Assembler Work?

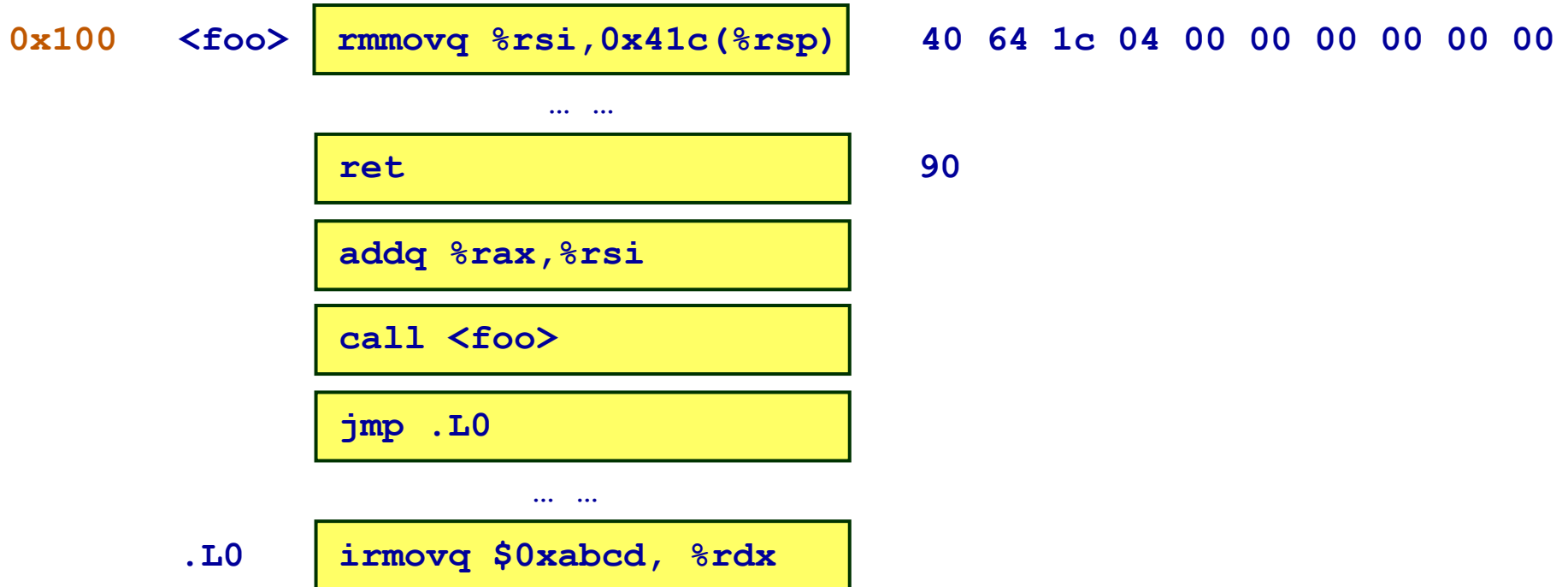
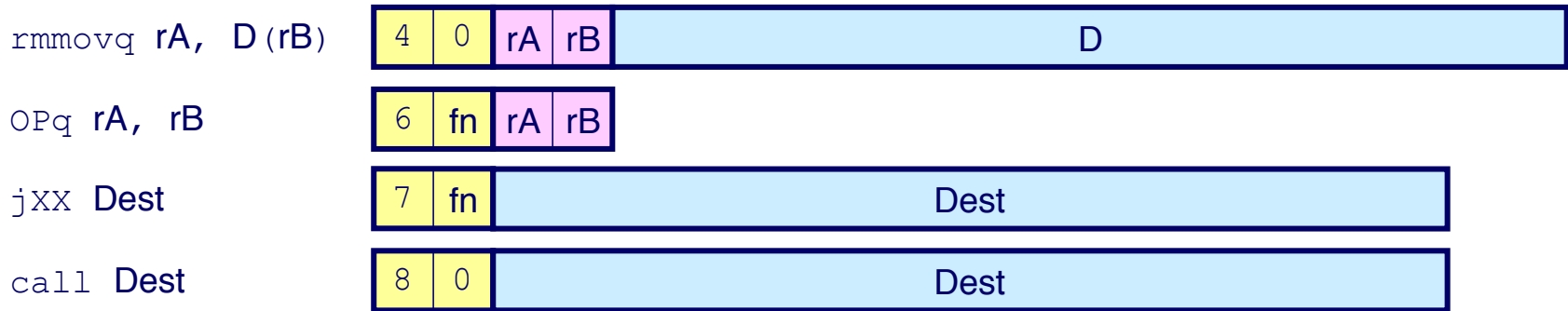


# How Does An Assembler Work?

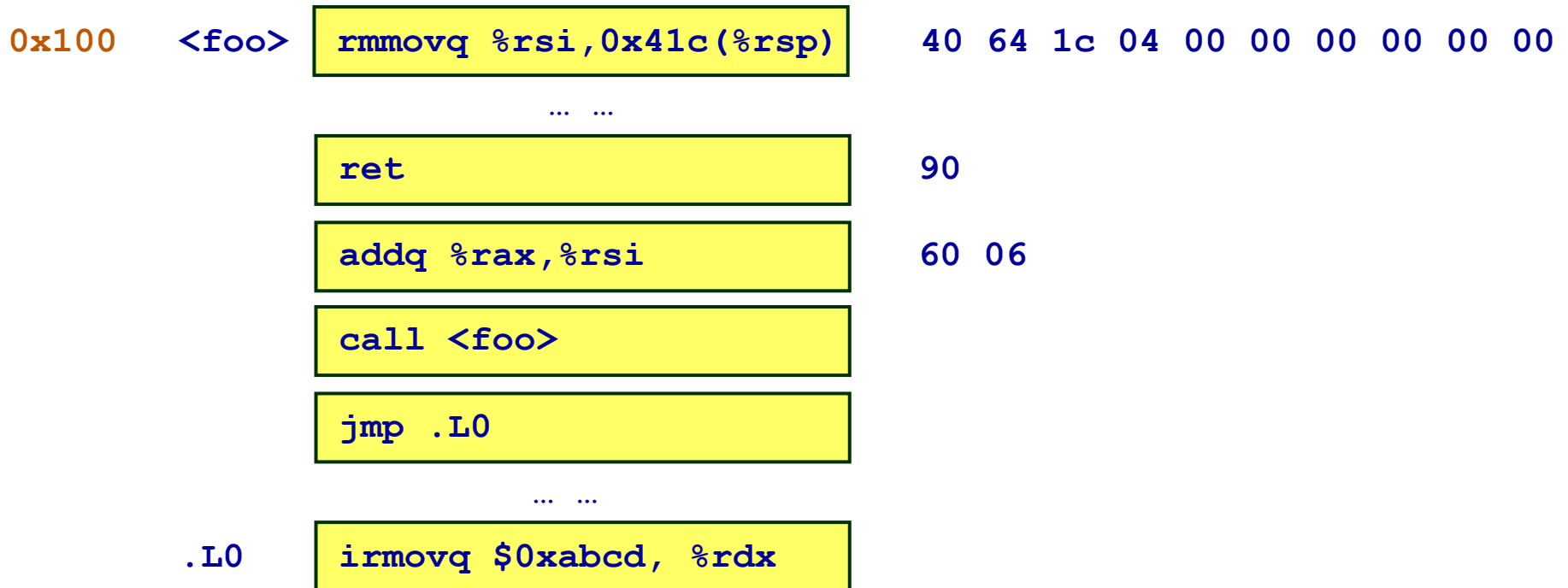
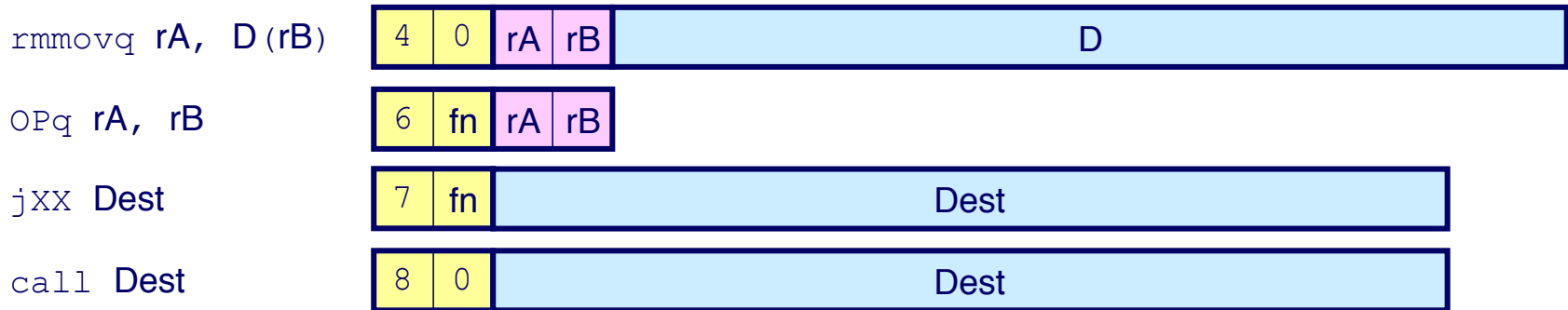




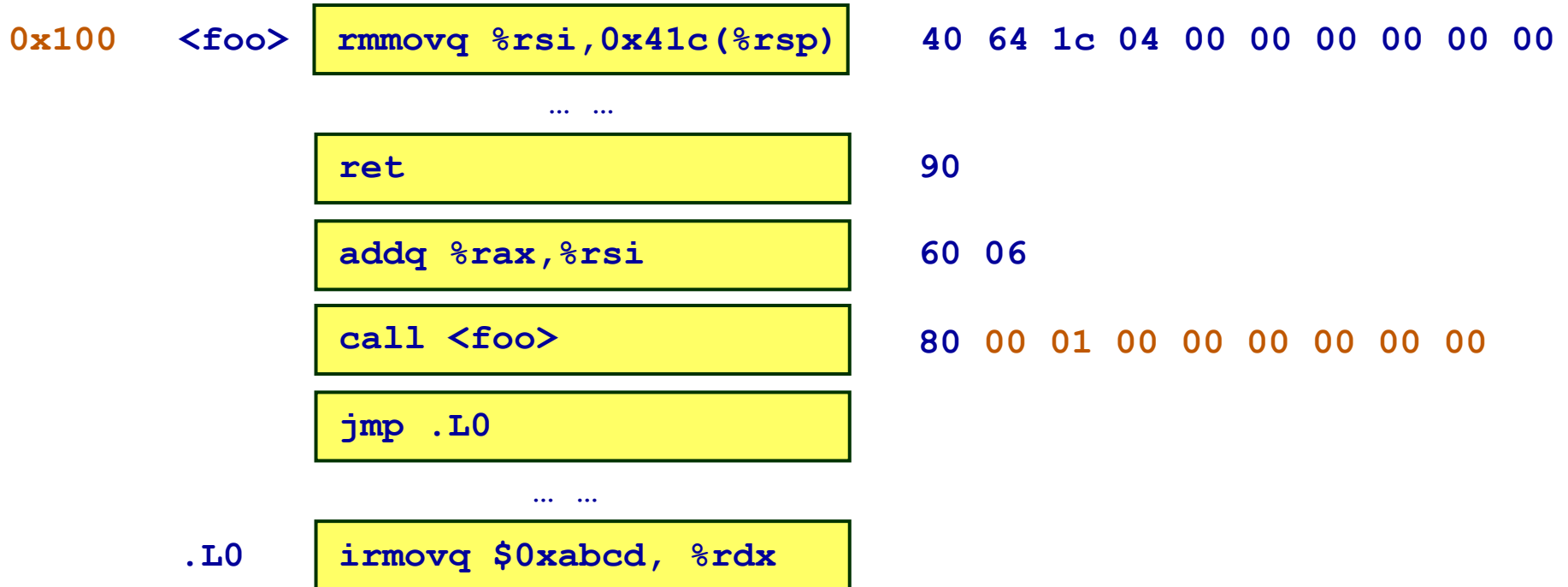
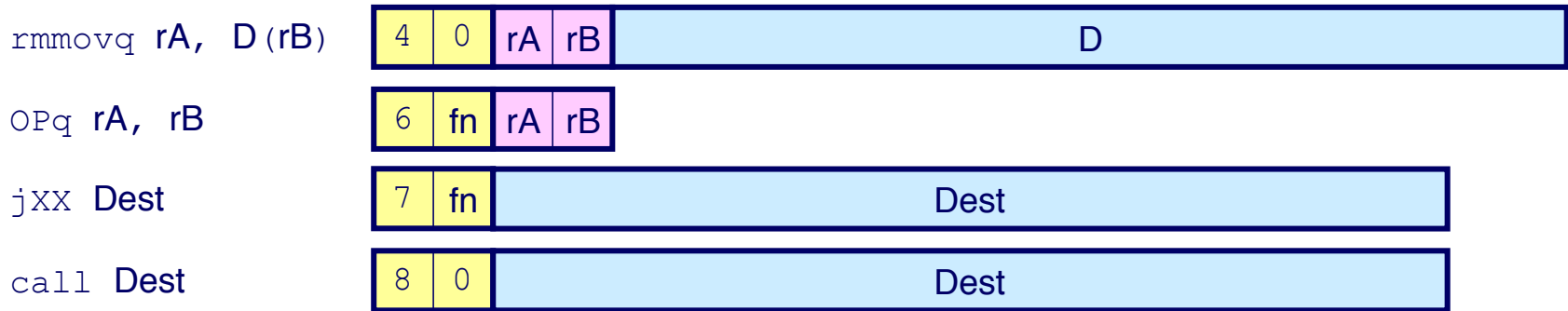
# How Does An Assembler Work?



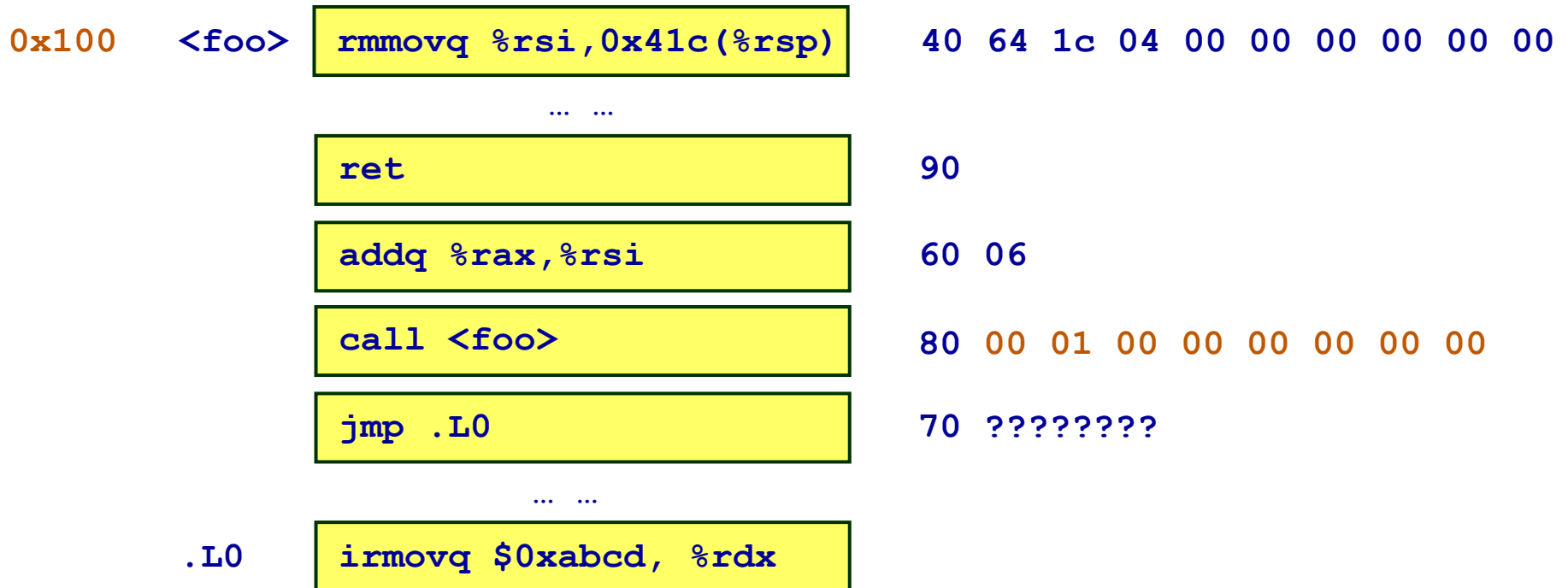
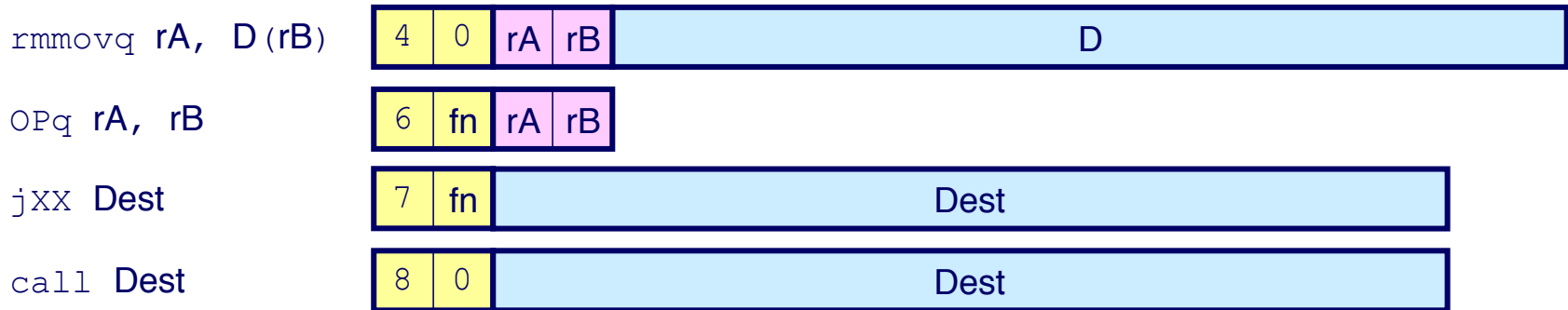
# How Does An Assembler Work?



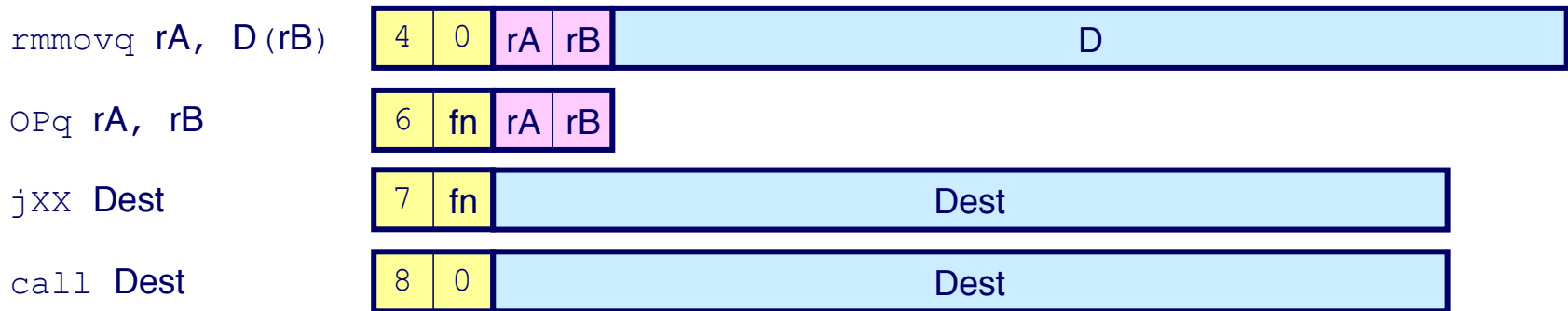
# How Does An Assembler Work?



# How Does An Assembler Work?

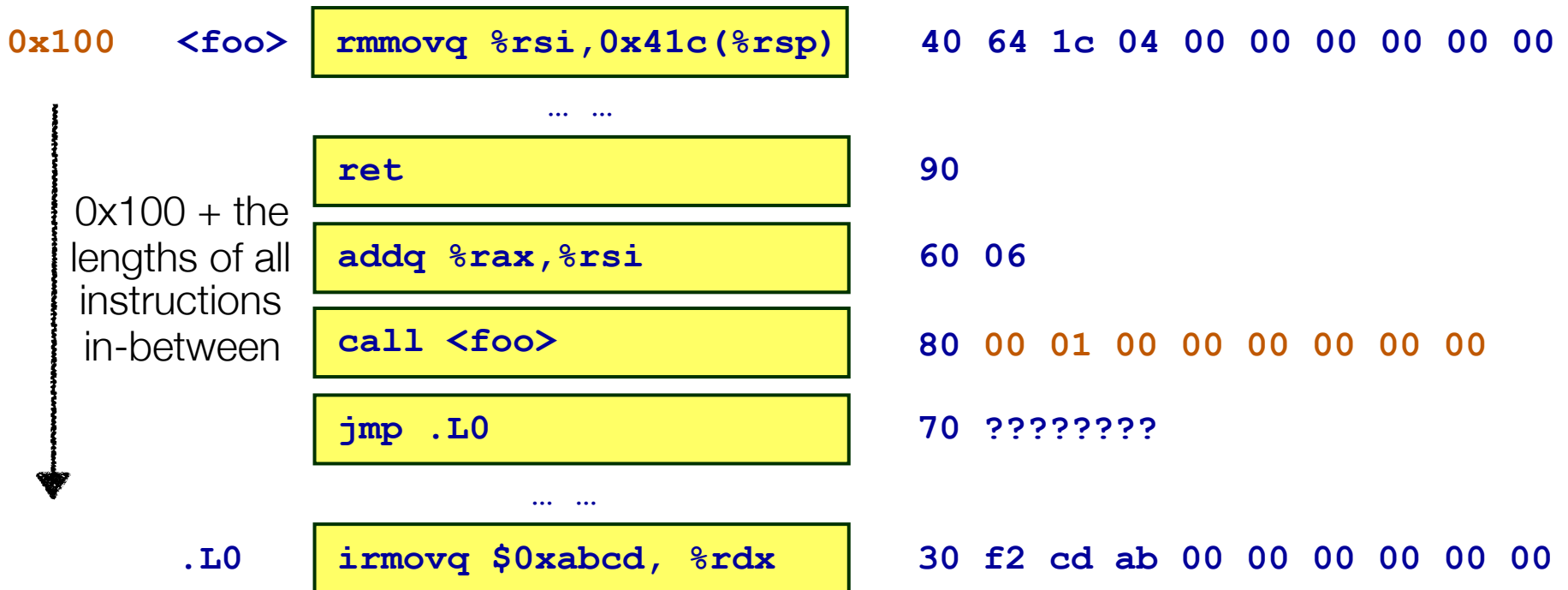
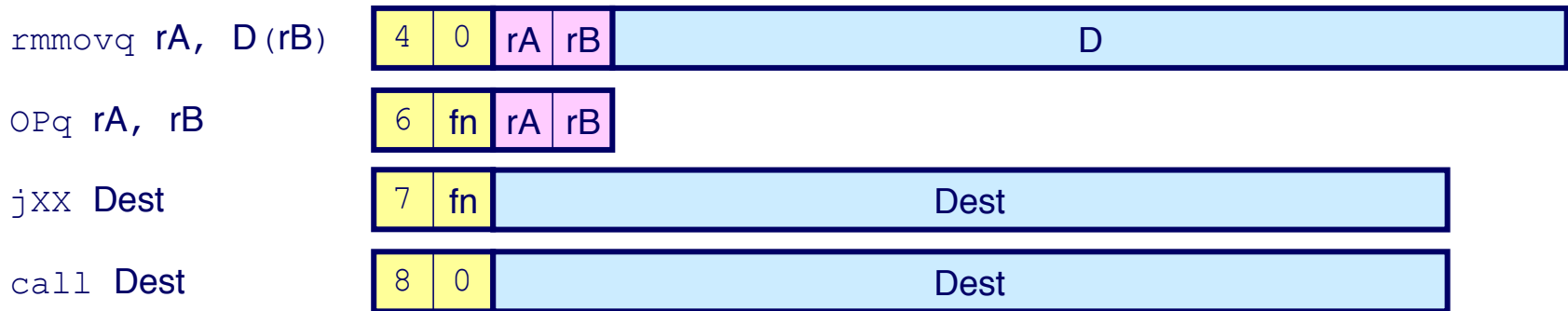


# How Does An Assembler Work?

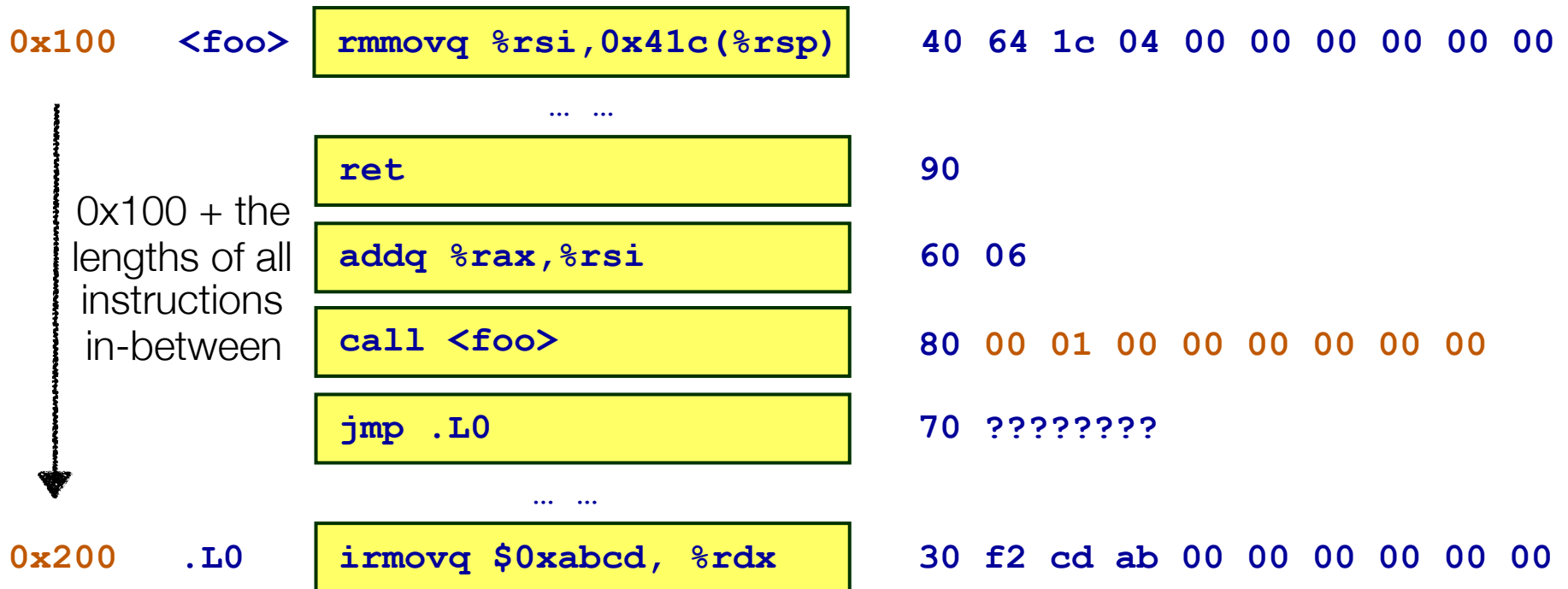
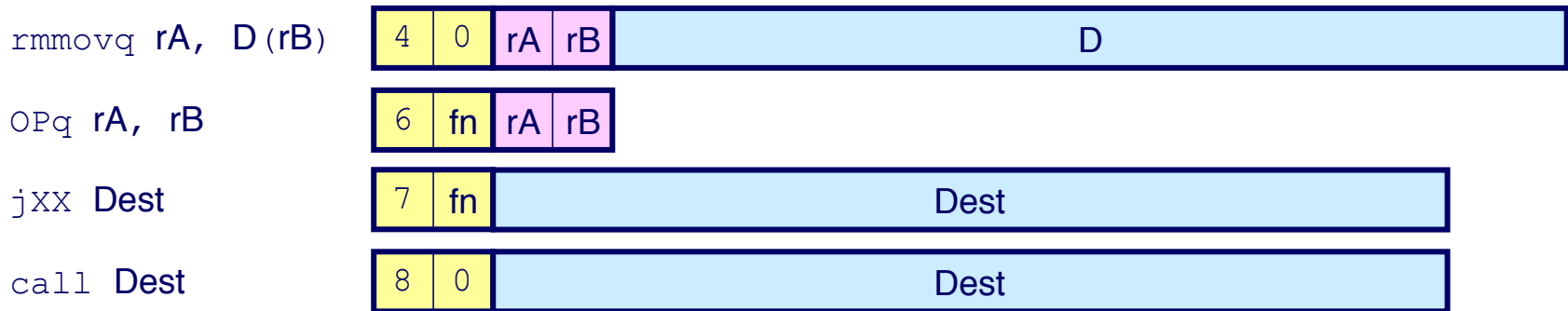


0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40	64	1c	04	00	00	00	00	00	00
		...										
		ret	90									
		addq %rax,%rsi	60	06								
		call <foo>	80	00	01	00	00	00	00	00	00	00
		jmp .L0	70	????????								
		...										
.L0		irmovq \$0xabcd, %rdx	30	f2	cd	ab	00	00	00	00	00	00

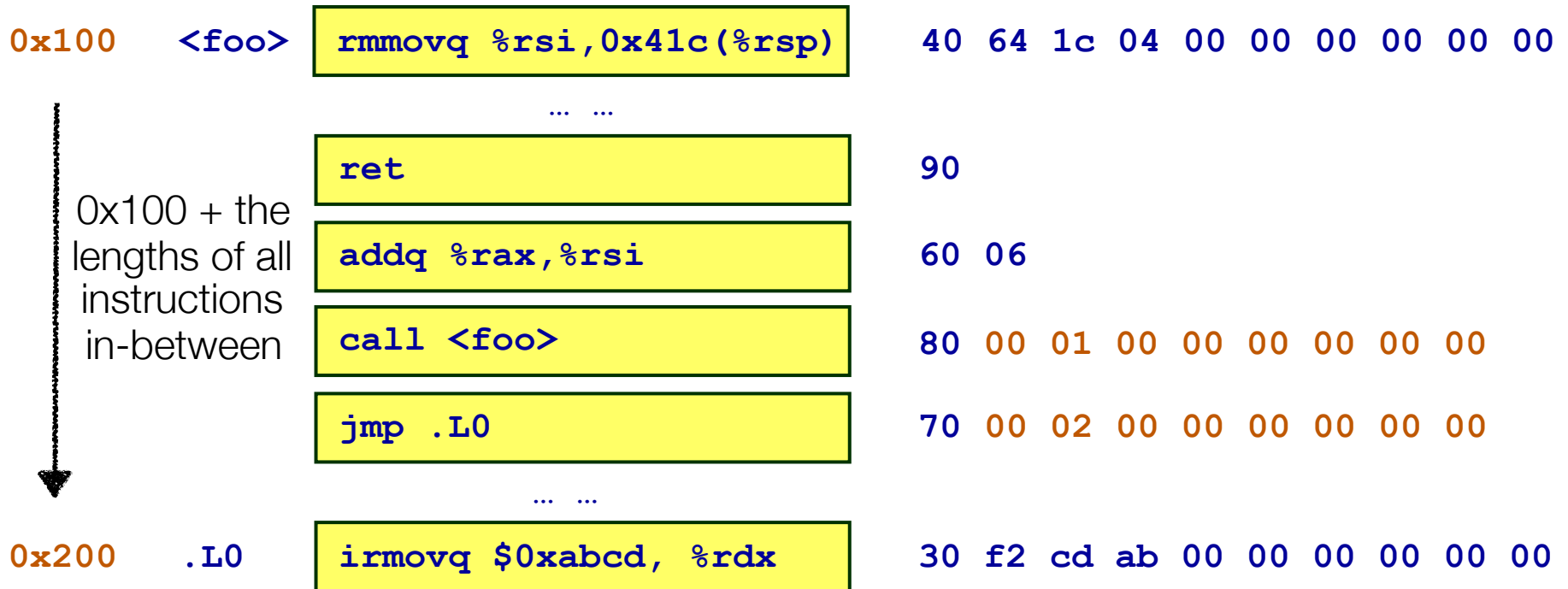
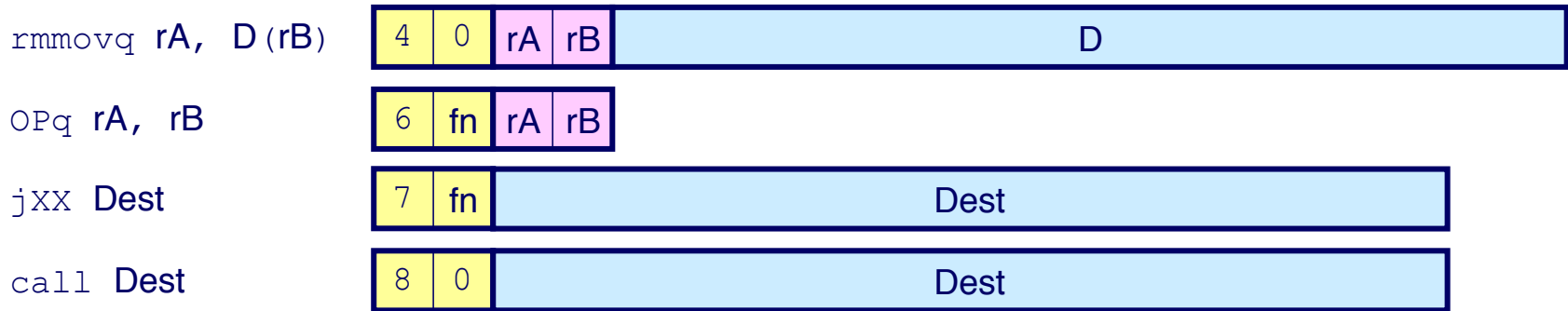
# How Does An Assembler Work?



# How Does An Assembler Work?



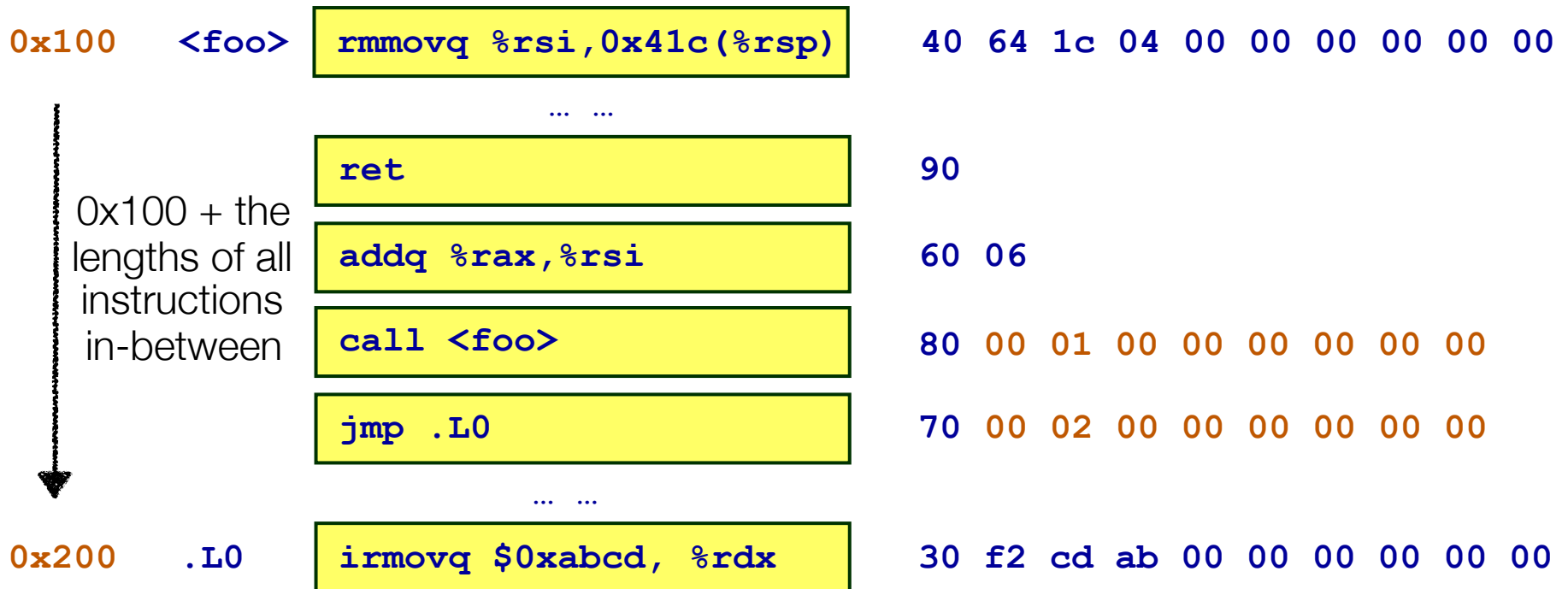
# How Does An Assembler Work?





# How Does An Assembler Work?

- The assembler is a program that translates assembly code to binary code
- The OS tells the assembler the start address of the code (sort of...)
- Translate the assembly program line by line
- Need to build a “label map” that maps each label to its address



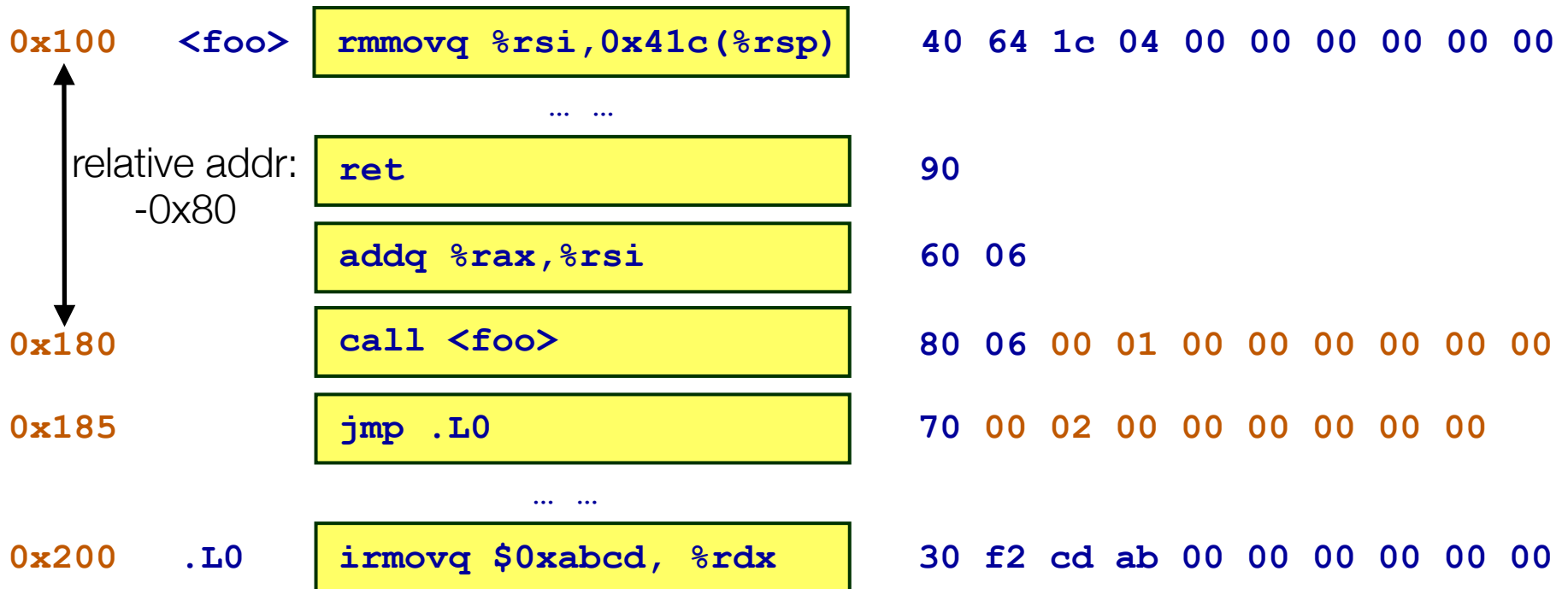
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?

0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
... ..			
		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 06 00 01 00 00 00 00 00 00
0x185		jmp .L0	70 00 02 00 00 00 00 00 00 00
... ..			
0x200	.L0	irmovq \$0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

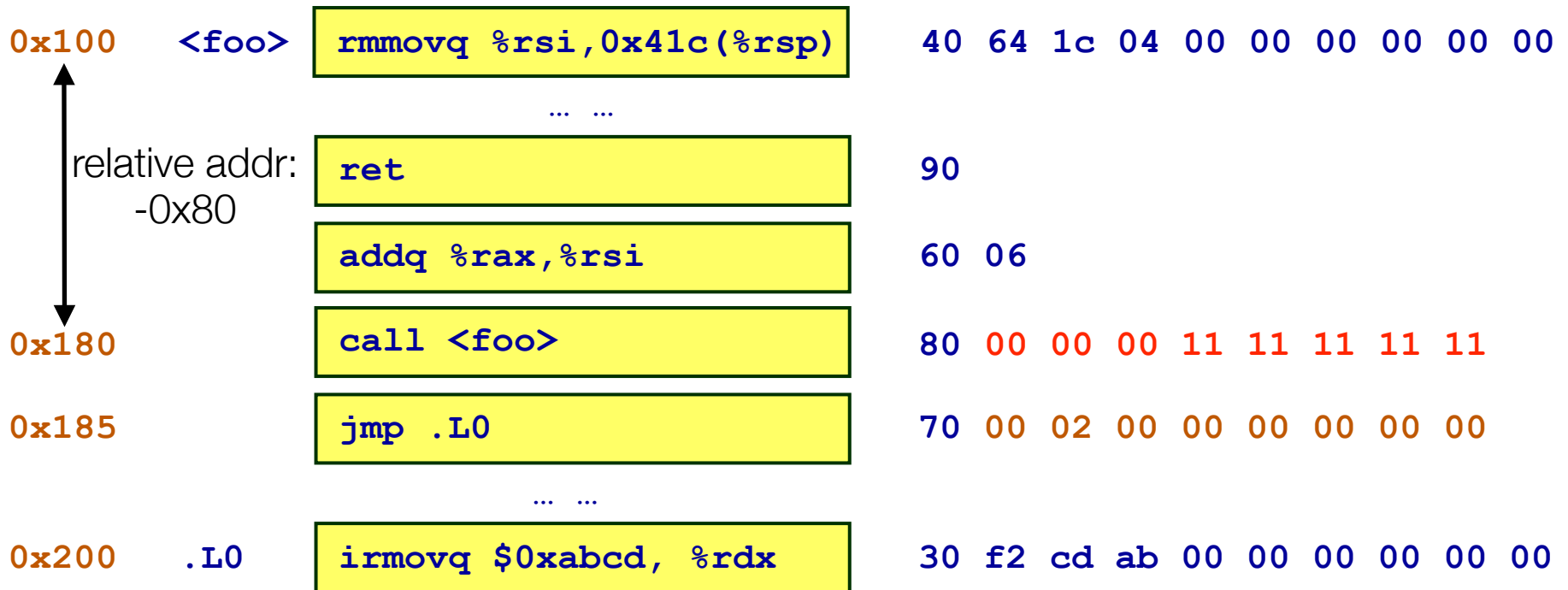
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?



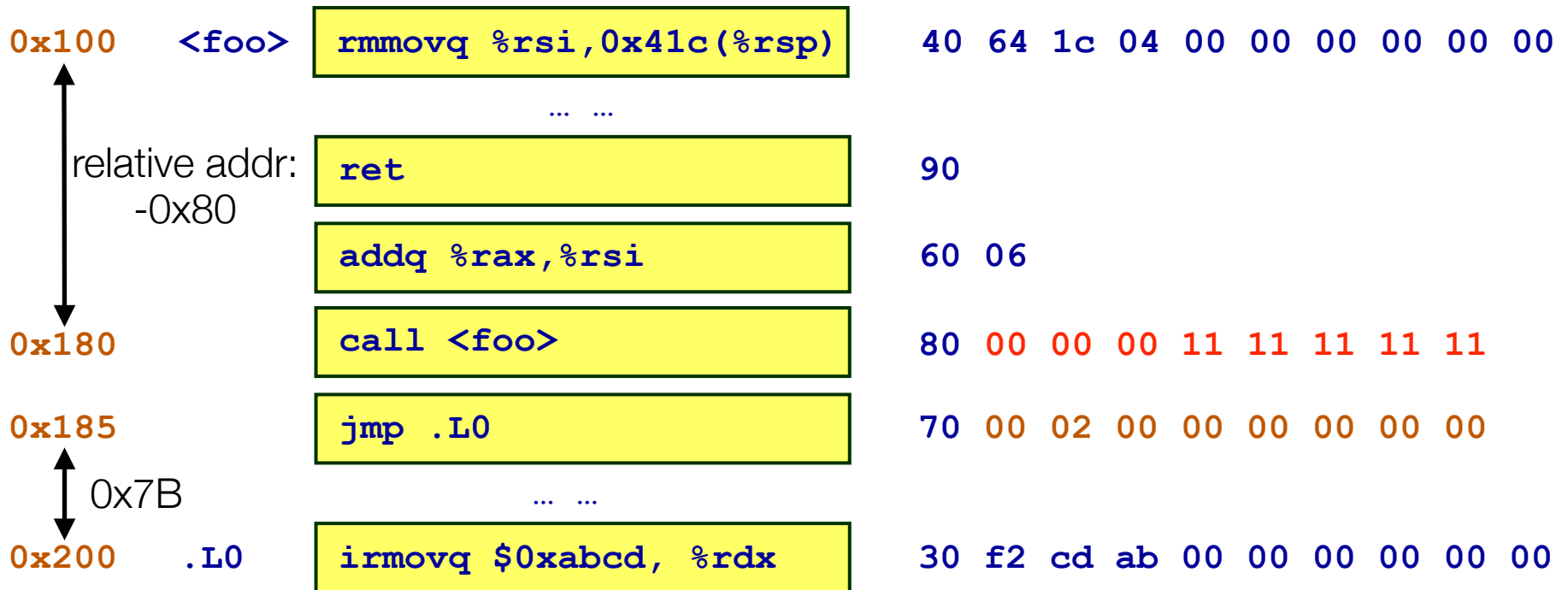
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?



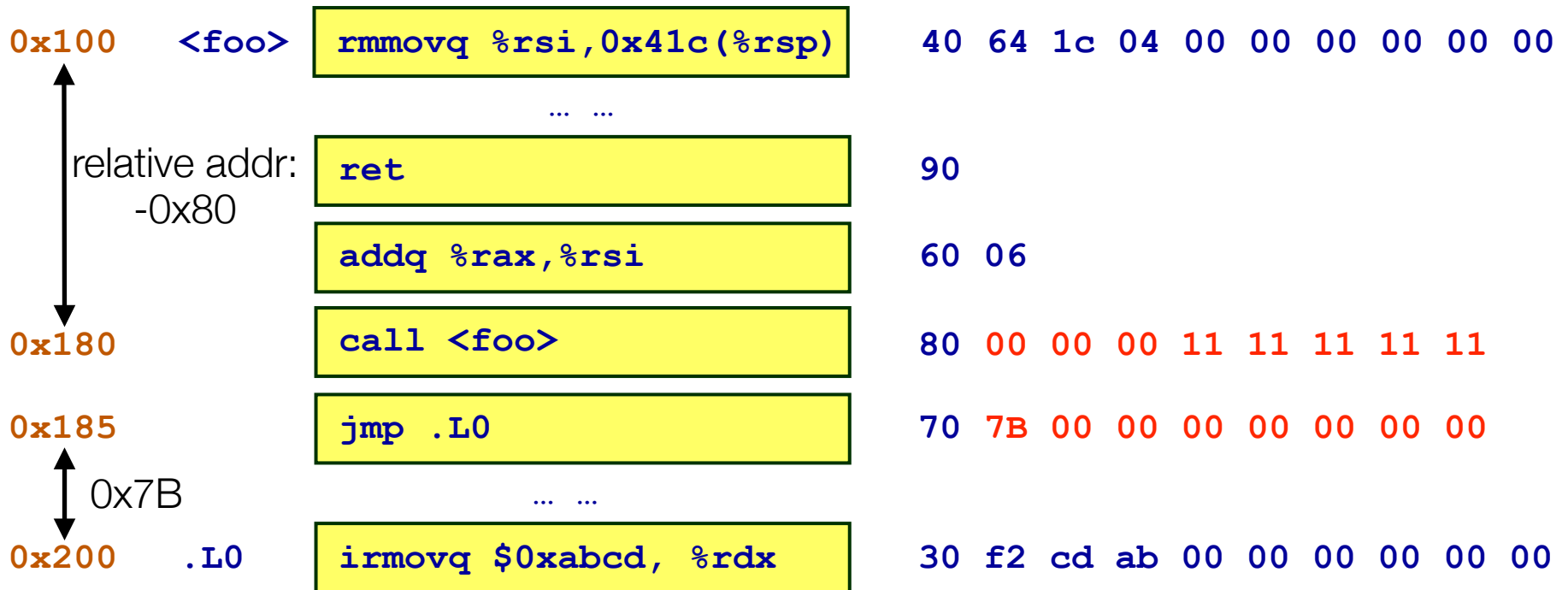
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?



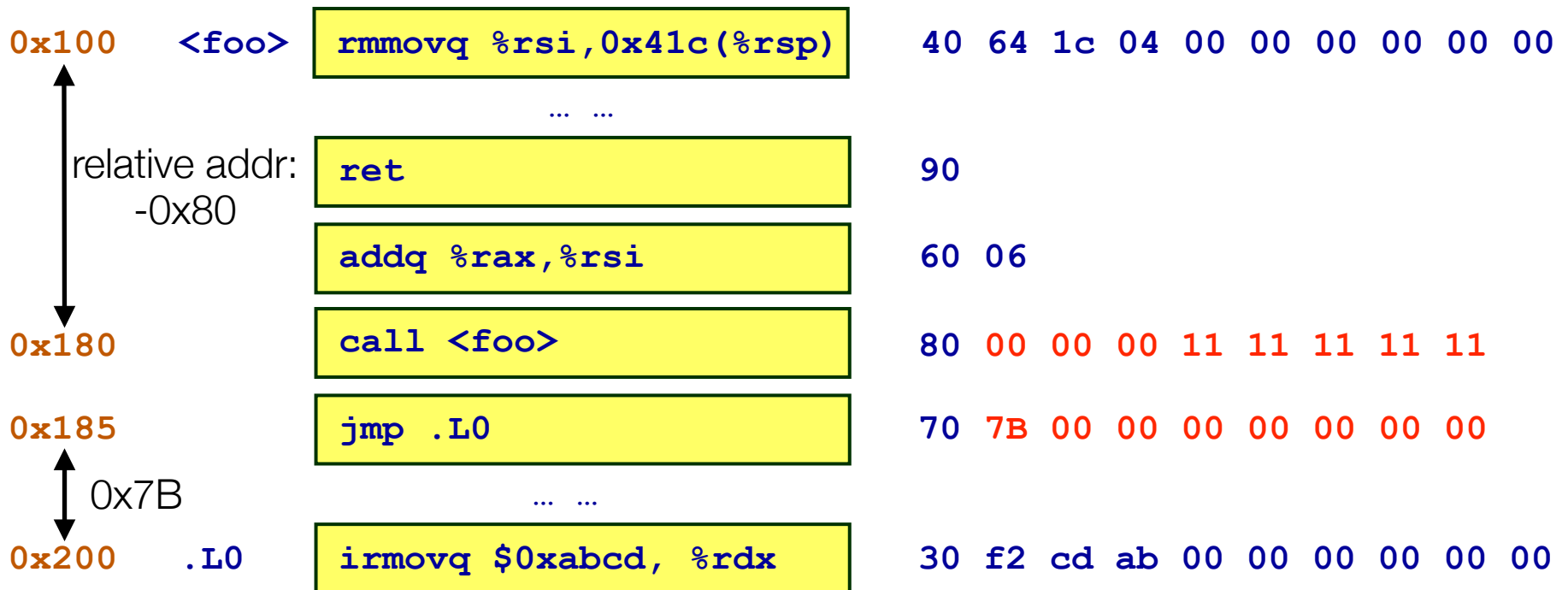
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?



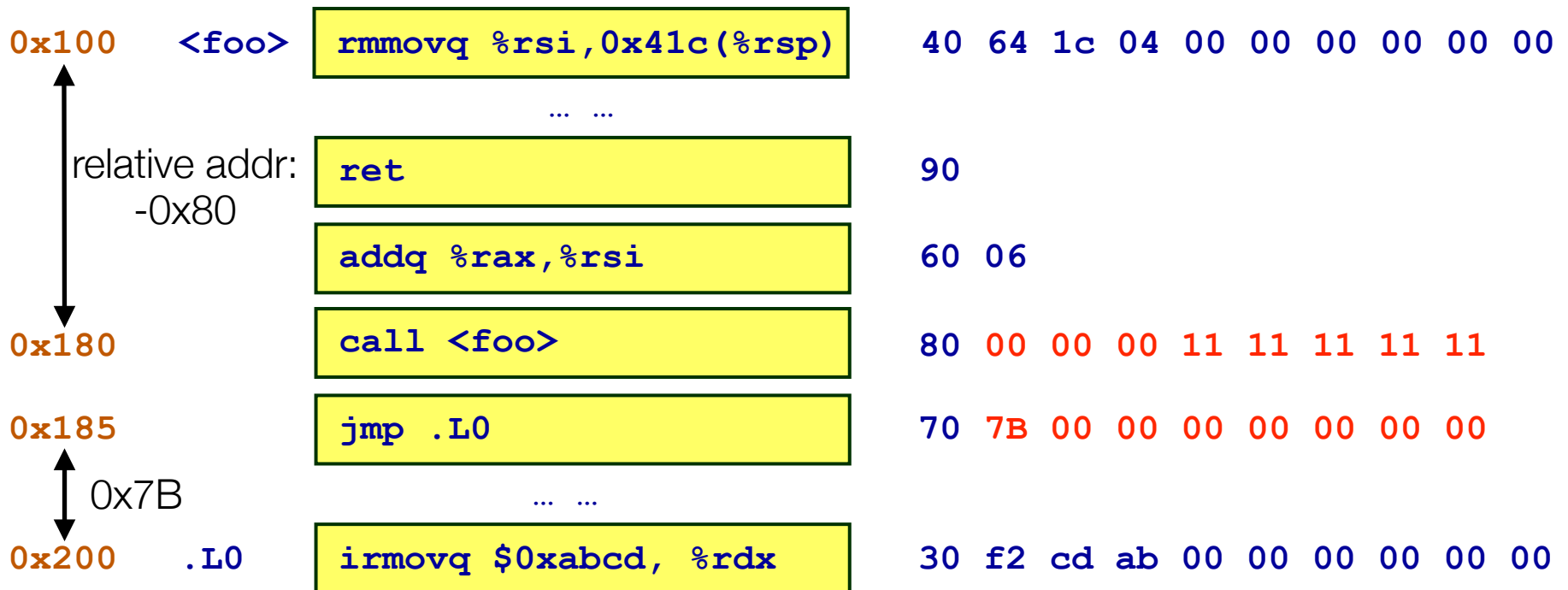
# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?
- If we use relative address, the exact start address of the code doesn't matter. Why?



# How Does An Assembler Work?

- What if the ISA encoding uses relative address for jump and call?
- If we use relative address, the exact start address of the code doesn't matter. Why?
- This code is called Position-Independent Code (PIC)

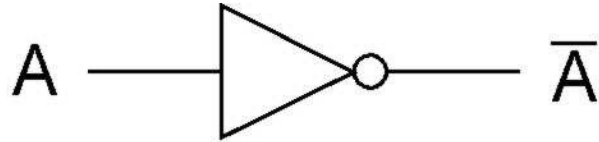




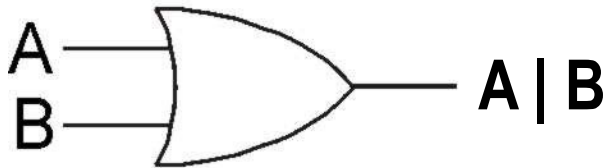
# Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

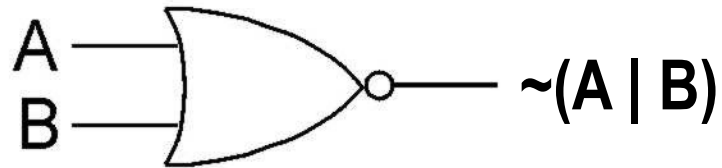
# Basic Logic Gates



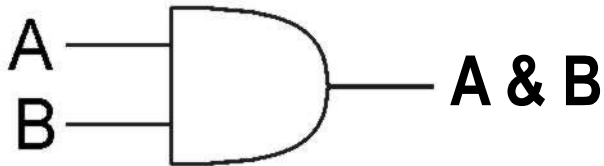
*NOT*



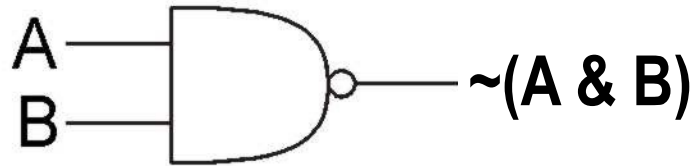
*OR*



*NOR*

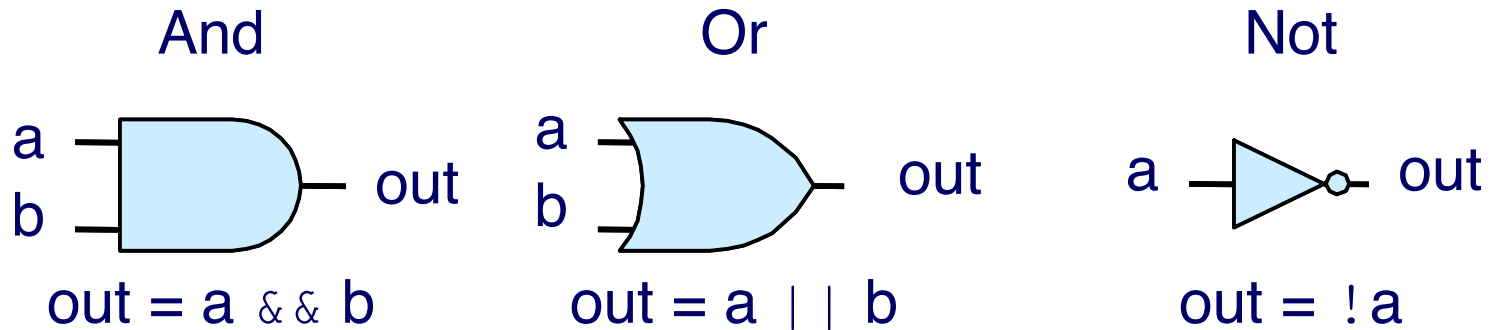


*AND*

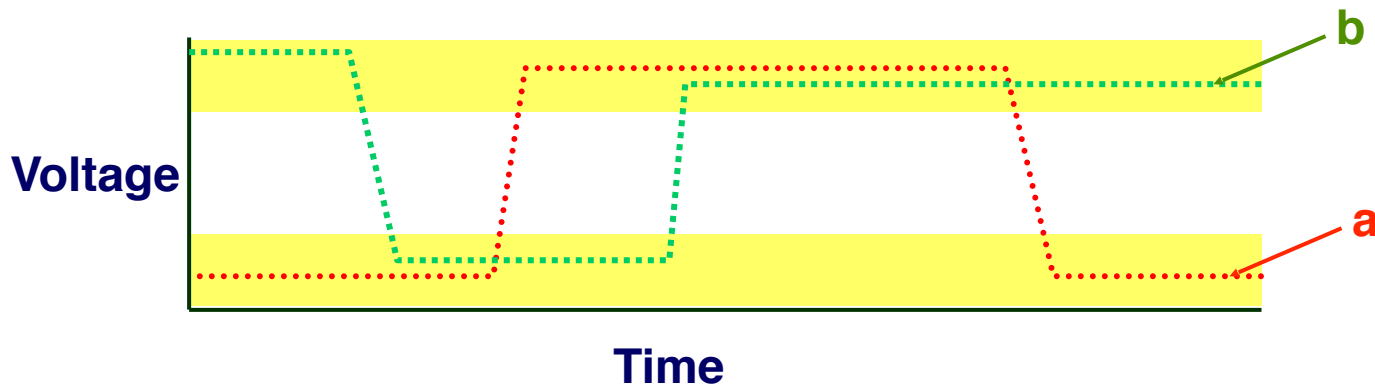


*NAND*

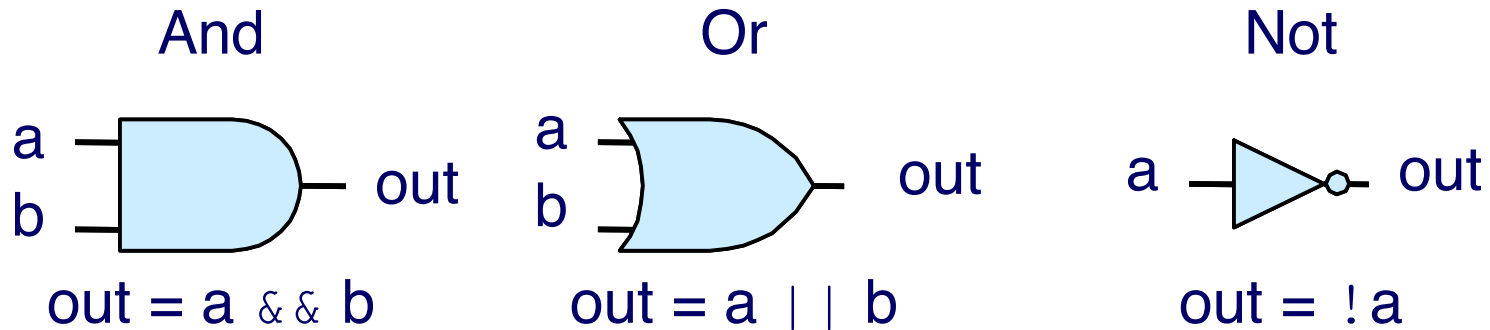
# Computing with Logic Gates



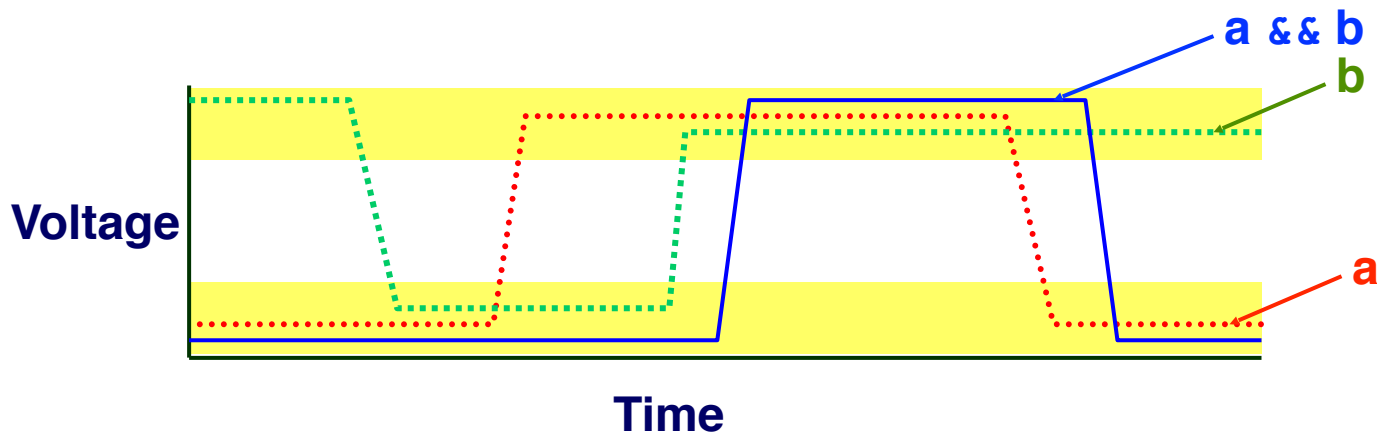
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



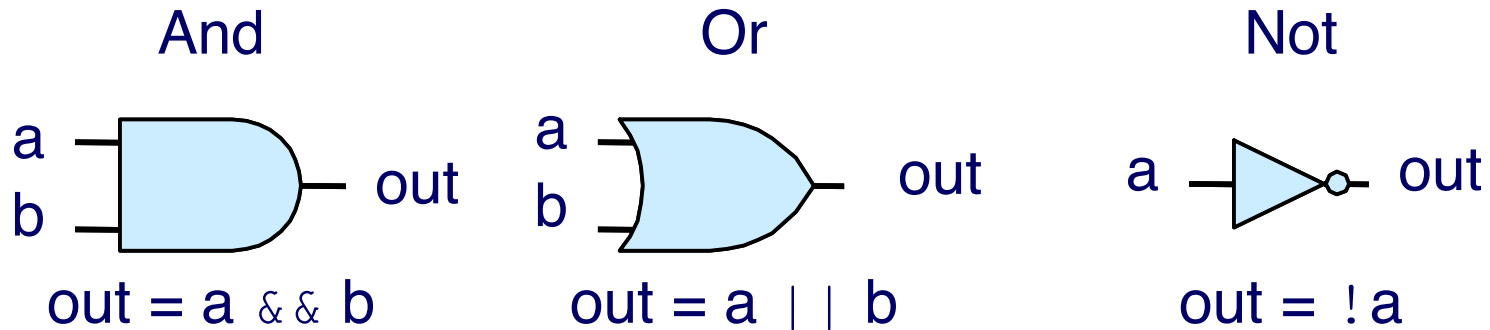
# Computing with Logic Gates



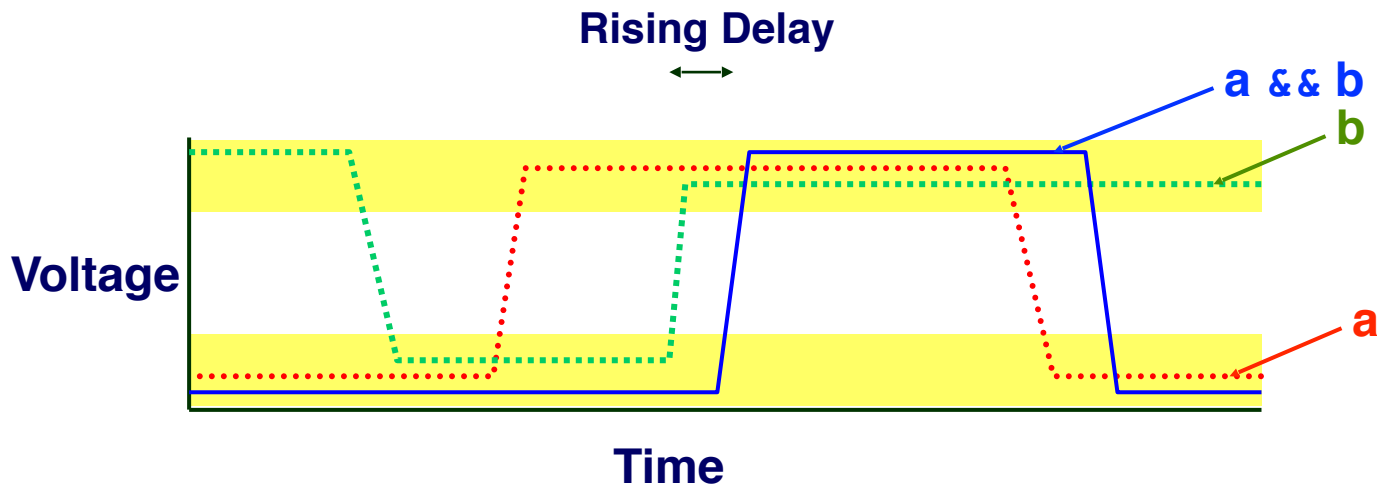
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



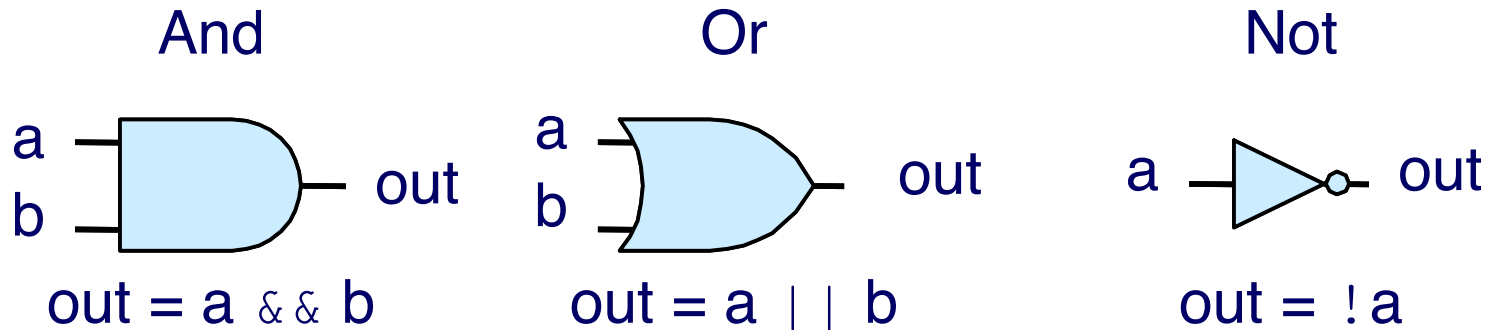
# Computing with Logic Gates



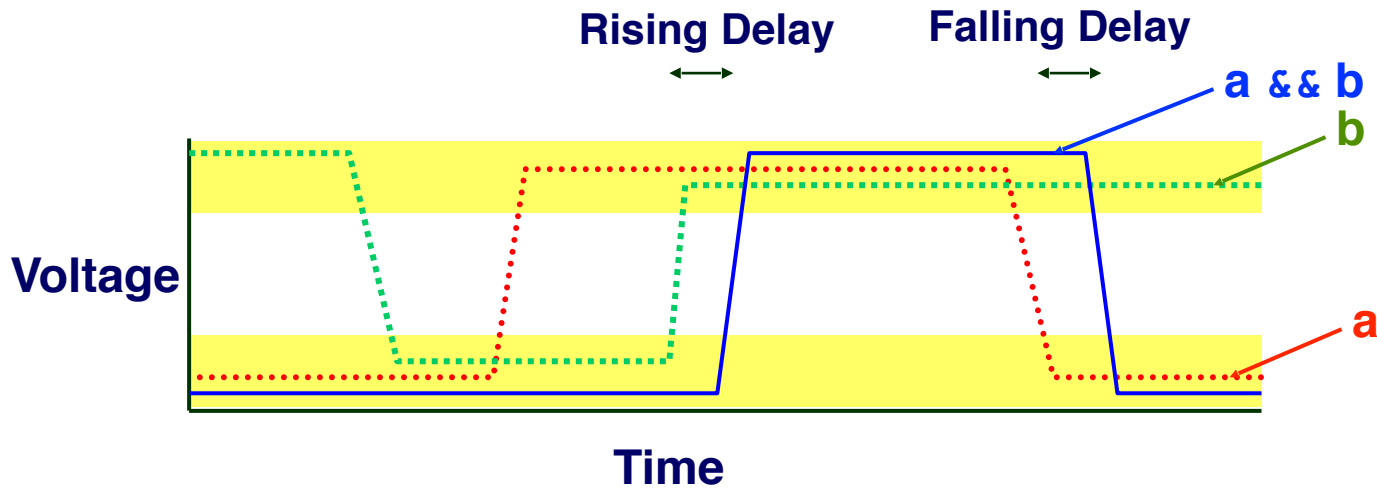
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs **with some small delay**
- **Different gates have different delays (b/c different transistor combinations)**



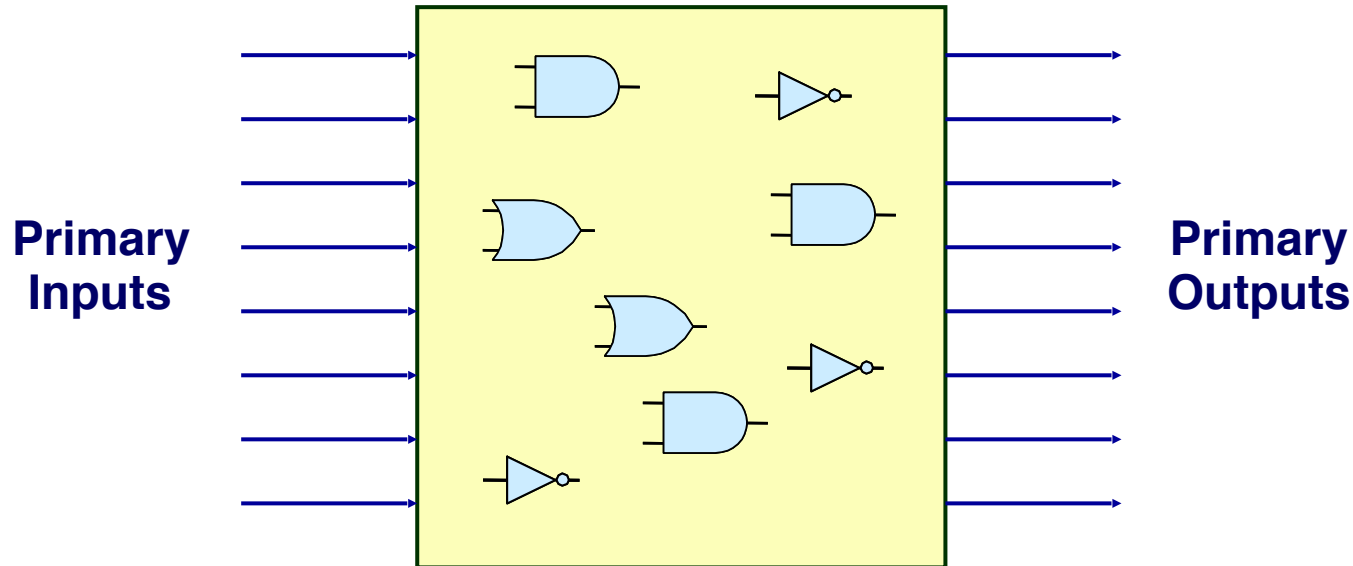
# Computing with Logic Gates



- 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



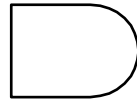
- A Network of Logic Gates

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

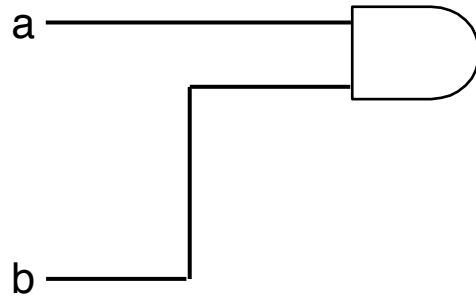
# Bit Equality



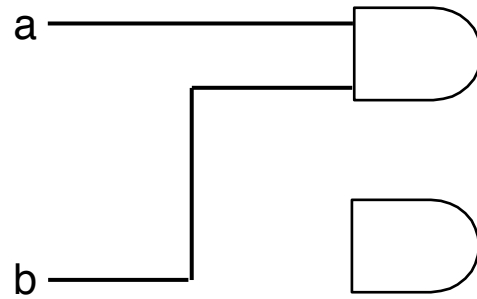
# Bit Equality



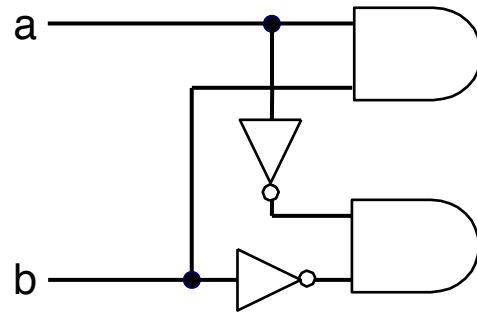
# Bit Equality



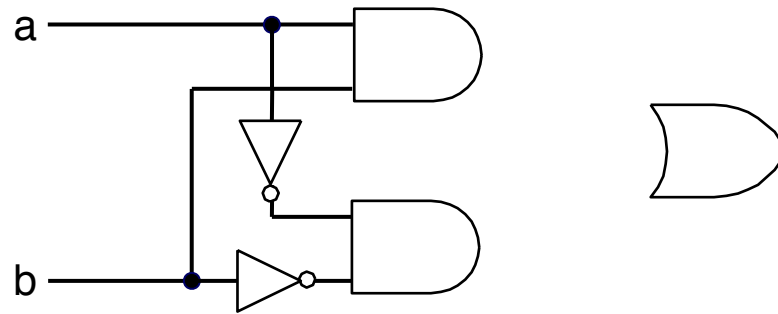
# Bit Equality



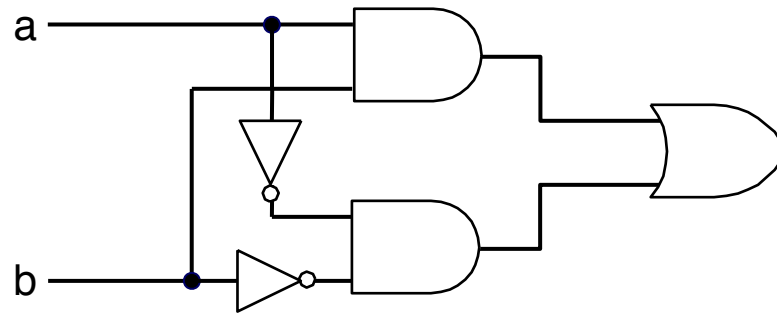
# Bit Equality



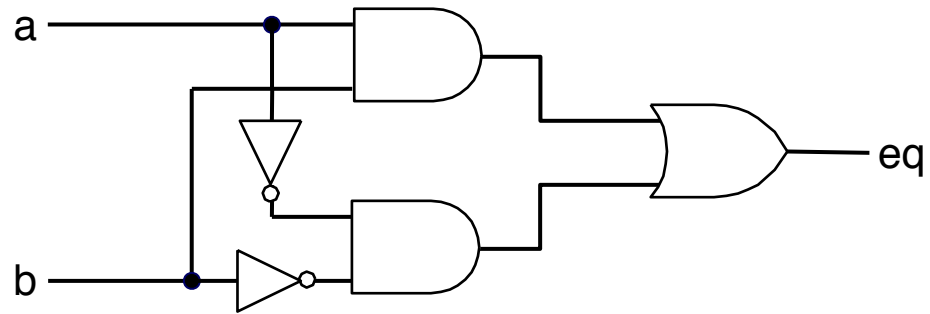
# Bit Equality



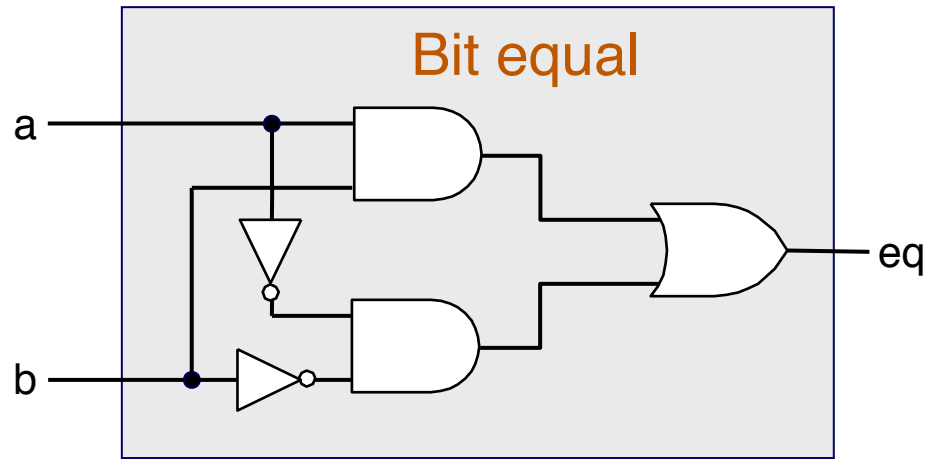
# Bit Equality



# Bit Equality

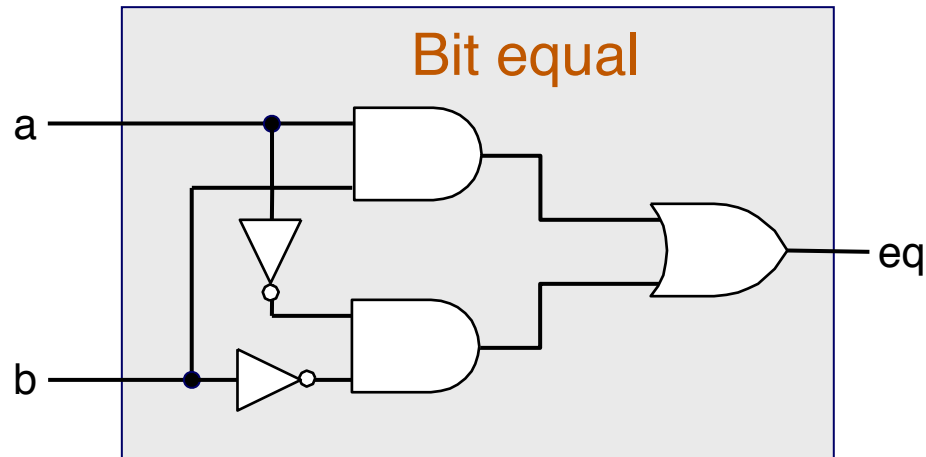


# Bit Equality



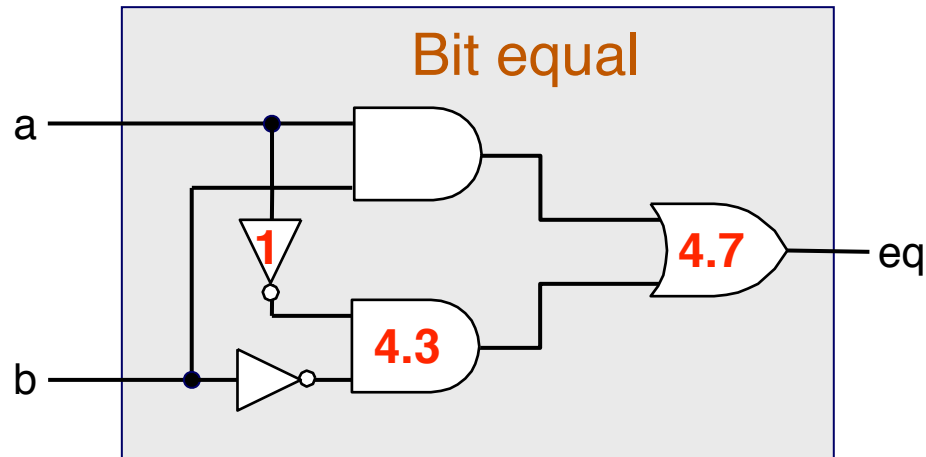


# Delay of Bit Equal Circuit



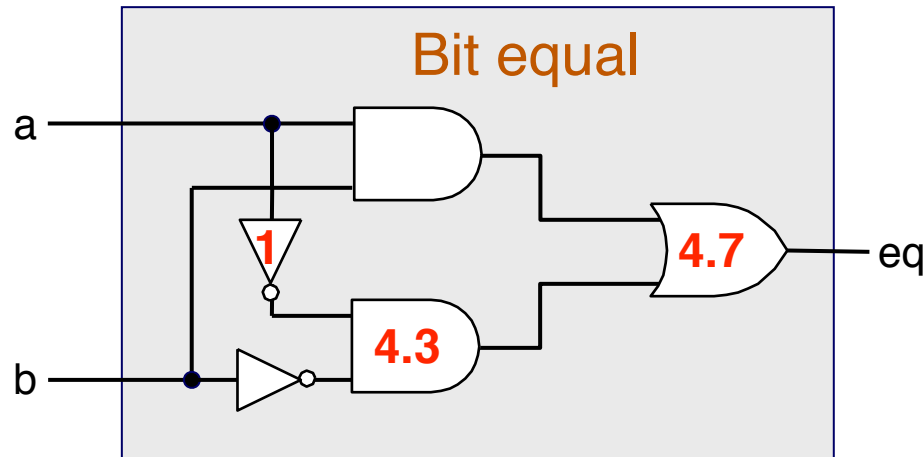
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

# Delay of Bit Equal Circuit



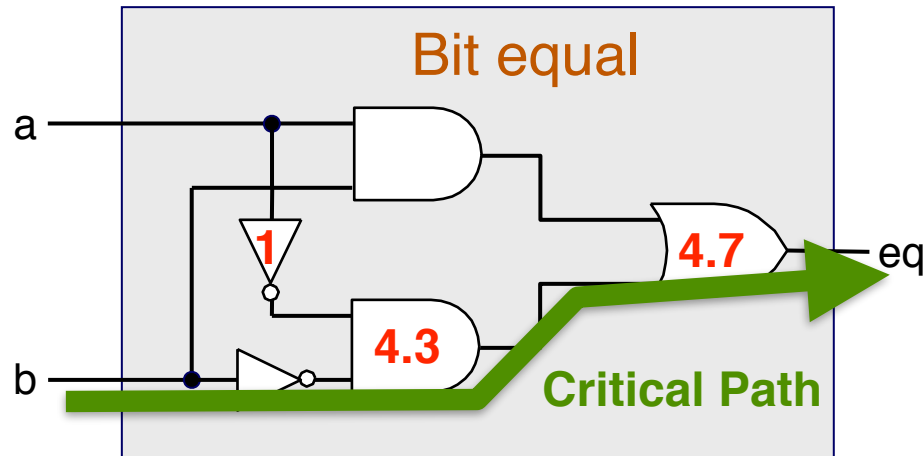
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

# Delay of Bit Equal Circuit



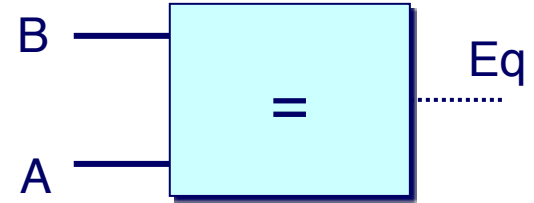
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its “critical path”
  - The path between an input and the output that the maximum delay
  - Estimating the critical path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

# Delay of Bit Equal Circuit

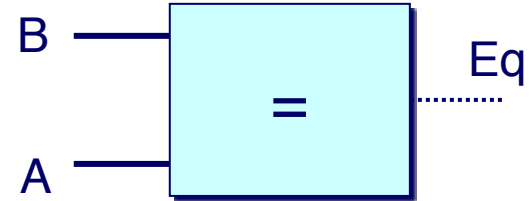
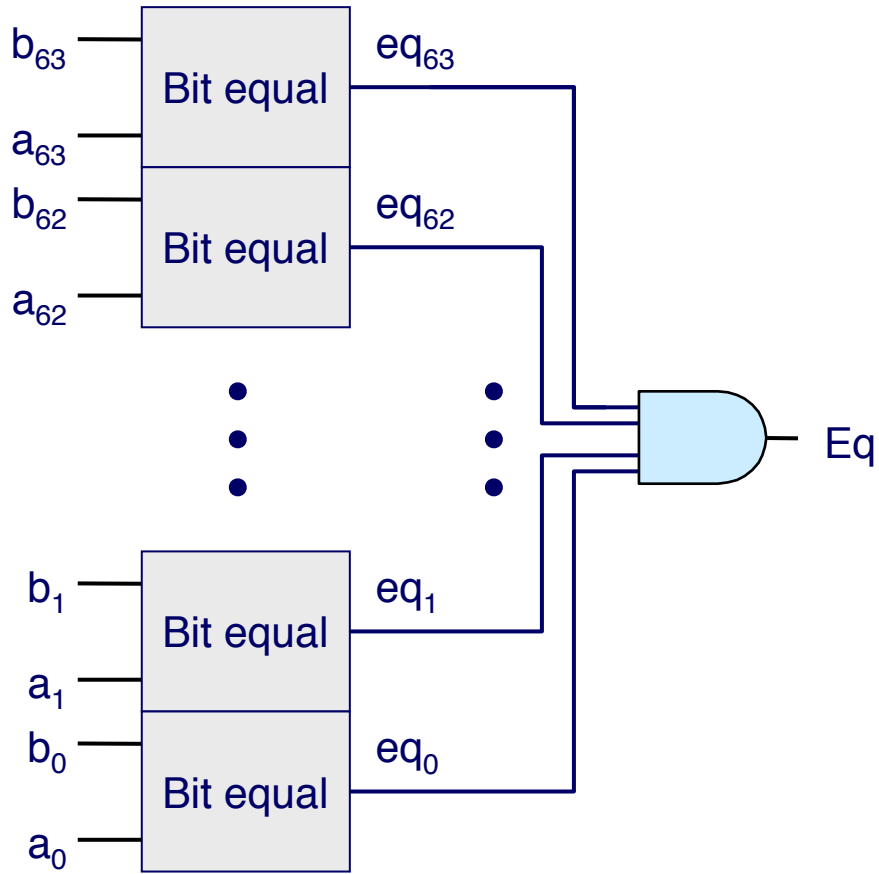


- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its “critical path”
  - The path between an input and the output that the maximum delay
  - Estimating the critical path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

# 64-bit Equality

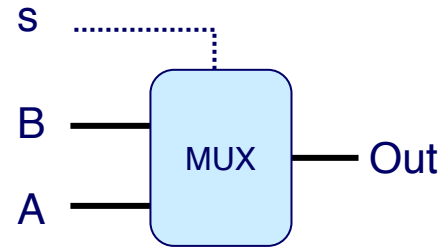


# 64-bit Equality



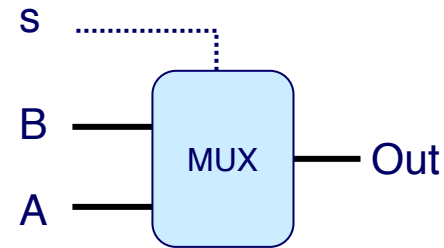
# Bit-Level Multiplexor (MUX)

- 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  $s=0$

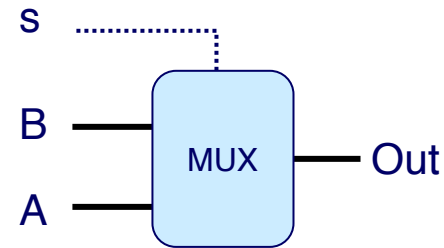


```
bool out = (s&&a) || (!s&&b)
```

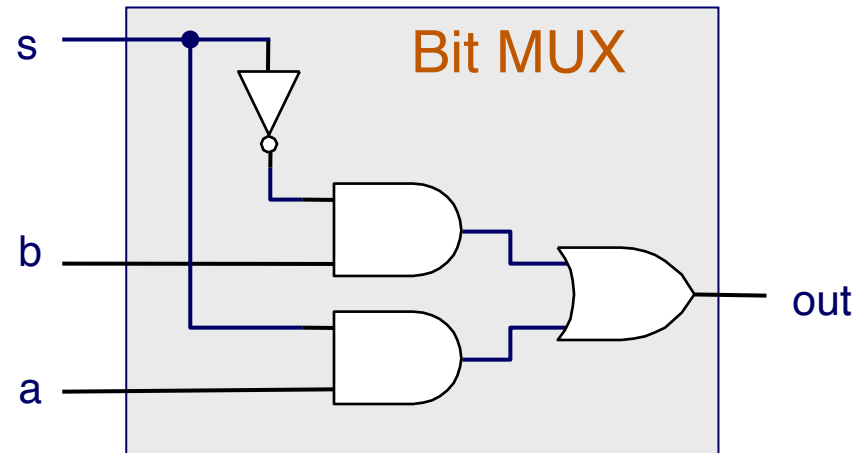


# Bit-Level Multiplexor (MUX)

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



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

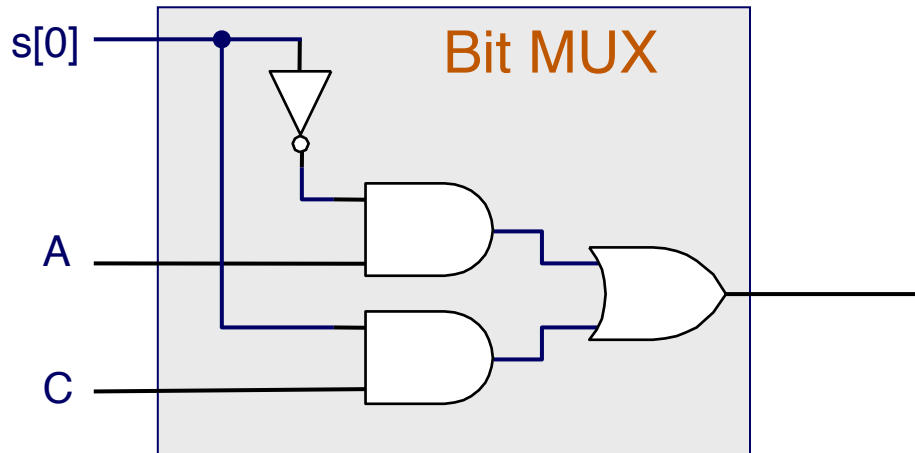


# 4-Input Multiplexor

- 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  $s = 11$

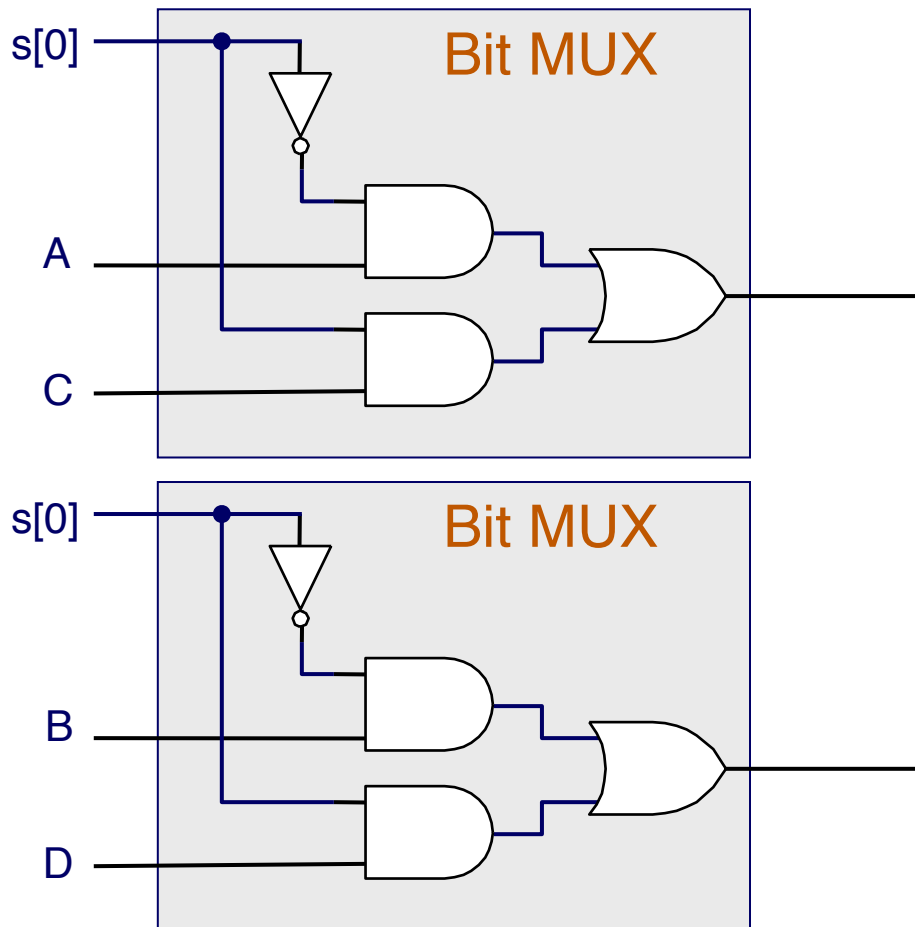
# 4-Input Multiplexor

- 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  $s = 11$



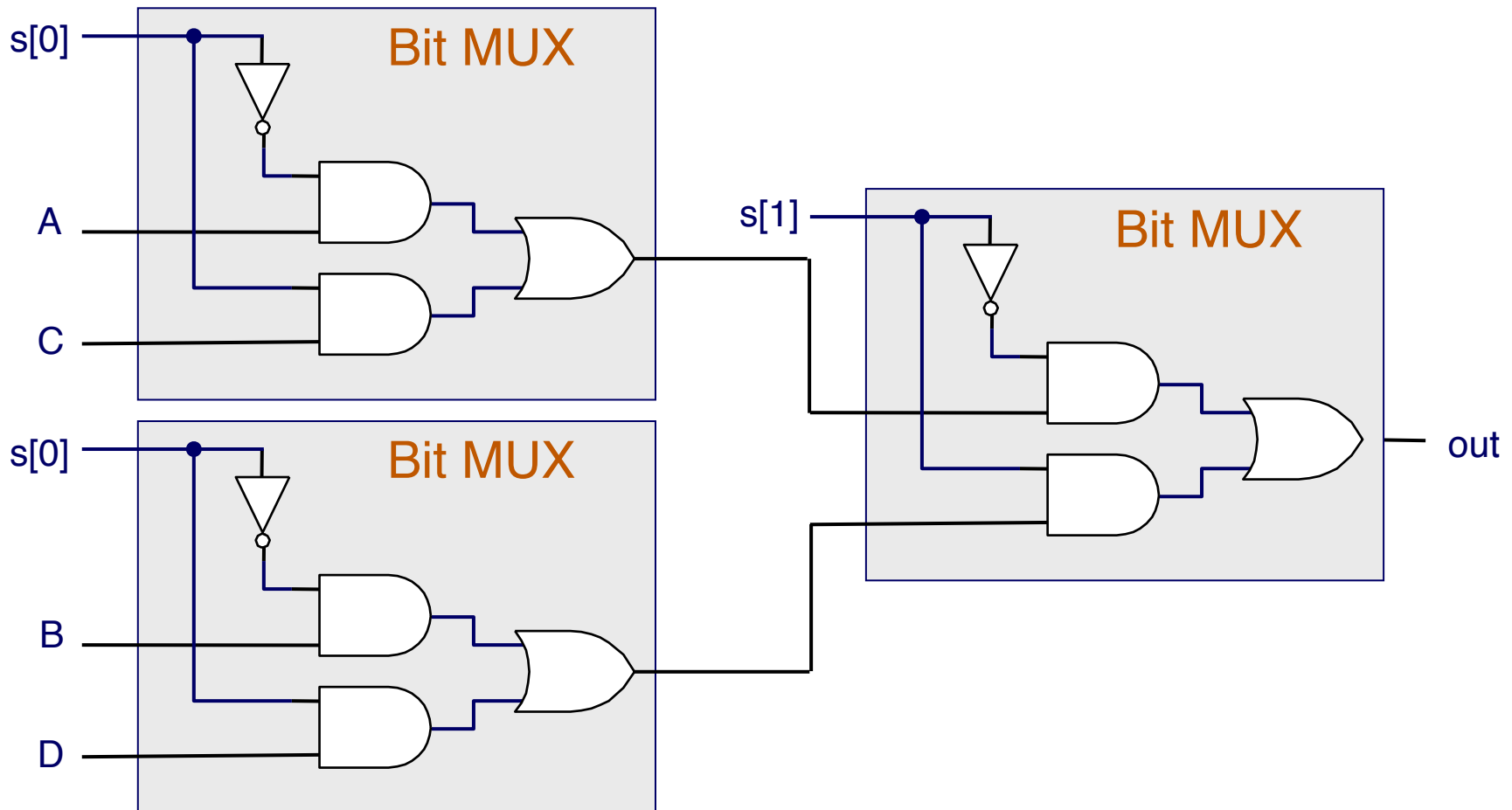
# 4-Input Multiplexor

- 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  $s = 11$



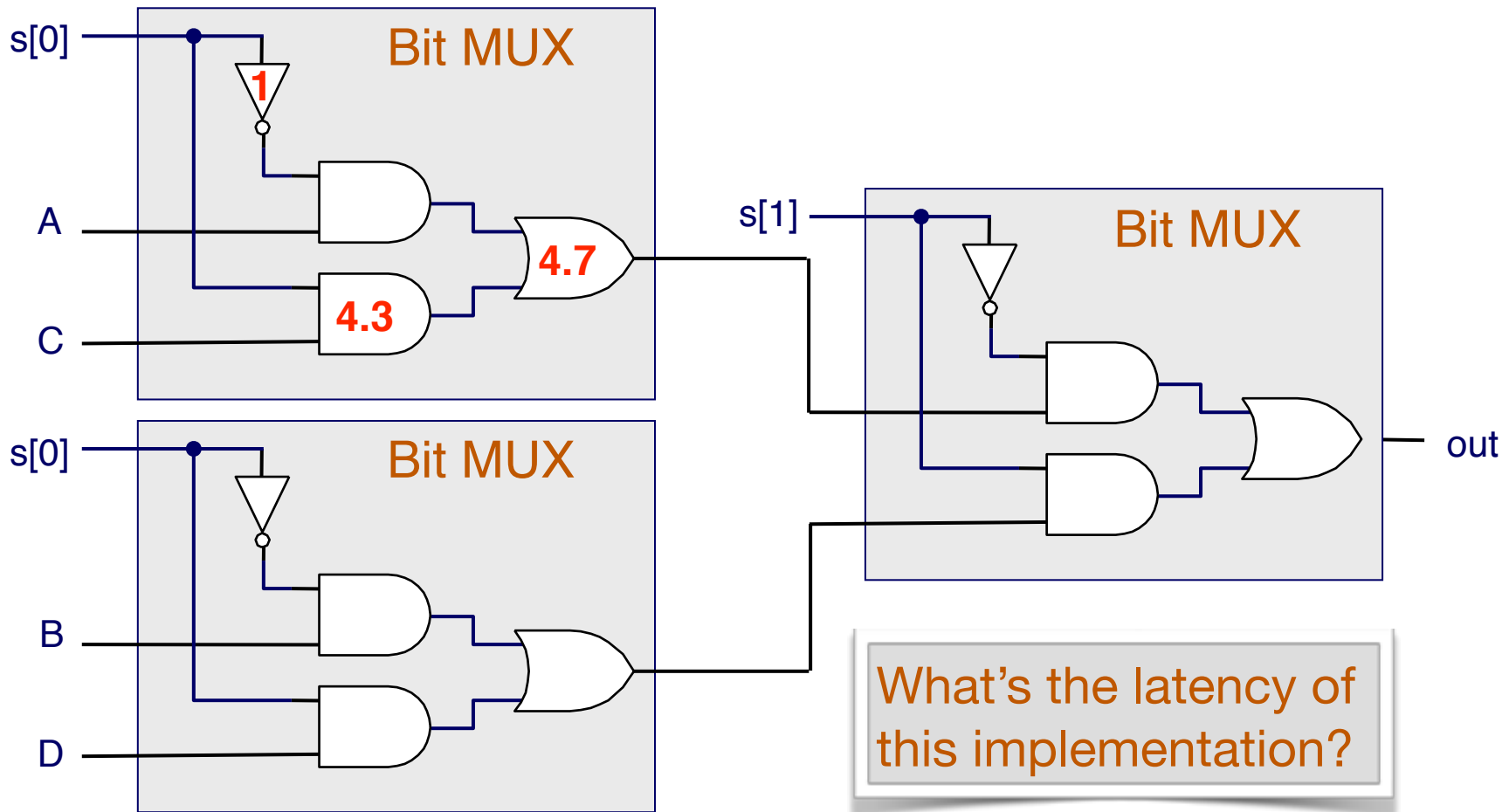
# 4-Input Multiplexor

- 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/synthesis tool 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/synthesis tool 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/synthesis tool 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/synthesis tool 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.

<b>A</b>	<b>B</b>	<b>C<sub>in</sub></b>	<b>S C<sub>ou</sub></b> <b>t</b>	
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.

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

A	B	C <sub>in</sub>	S	C <sub>ou</sub>
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.

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

A	B	C <sub>in</sub>	S	C <sub>ou</sub>
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.

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

A	B	C <sub>in</sub>	S	C <sub>ou</sub> 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.

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

A	B	C <sub>in</sub>	S	C <sub>ou</sub> 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 <sub>in</sub>	S	C <sub>ou</sub>
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.

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

# 1-bit Full Adder

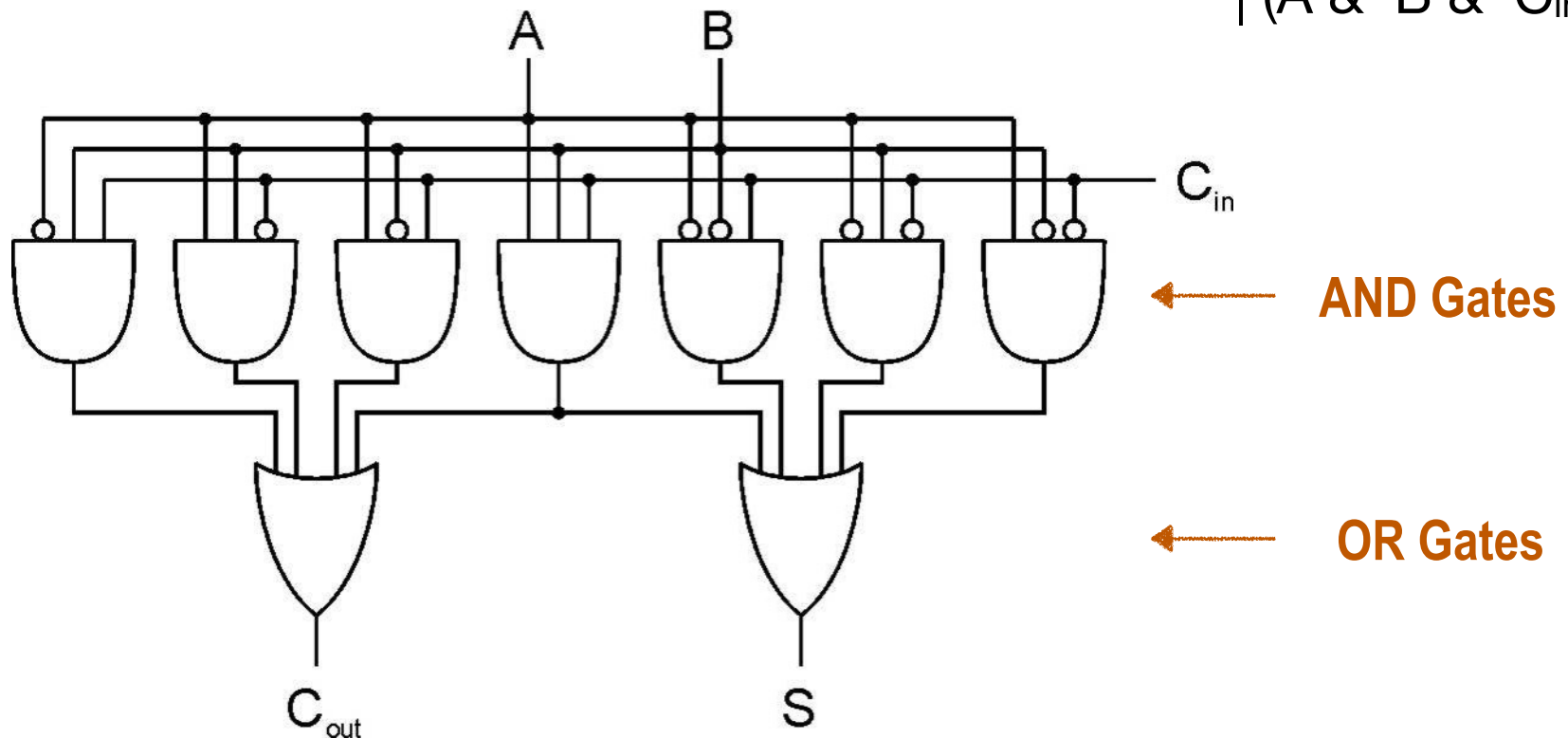
$$C_{ou} = (\sim A \& B \& C_{in})$$

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

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

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

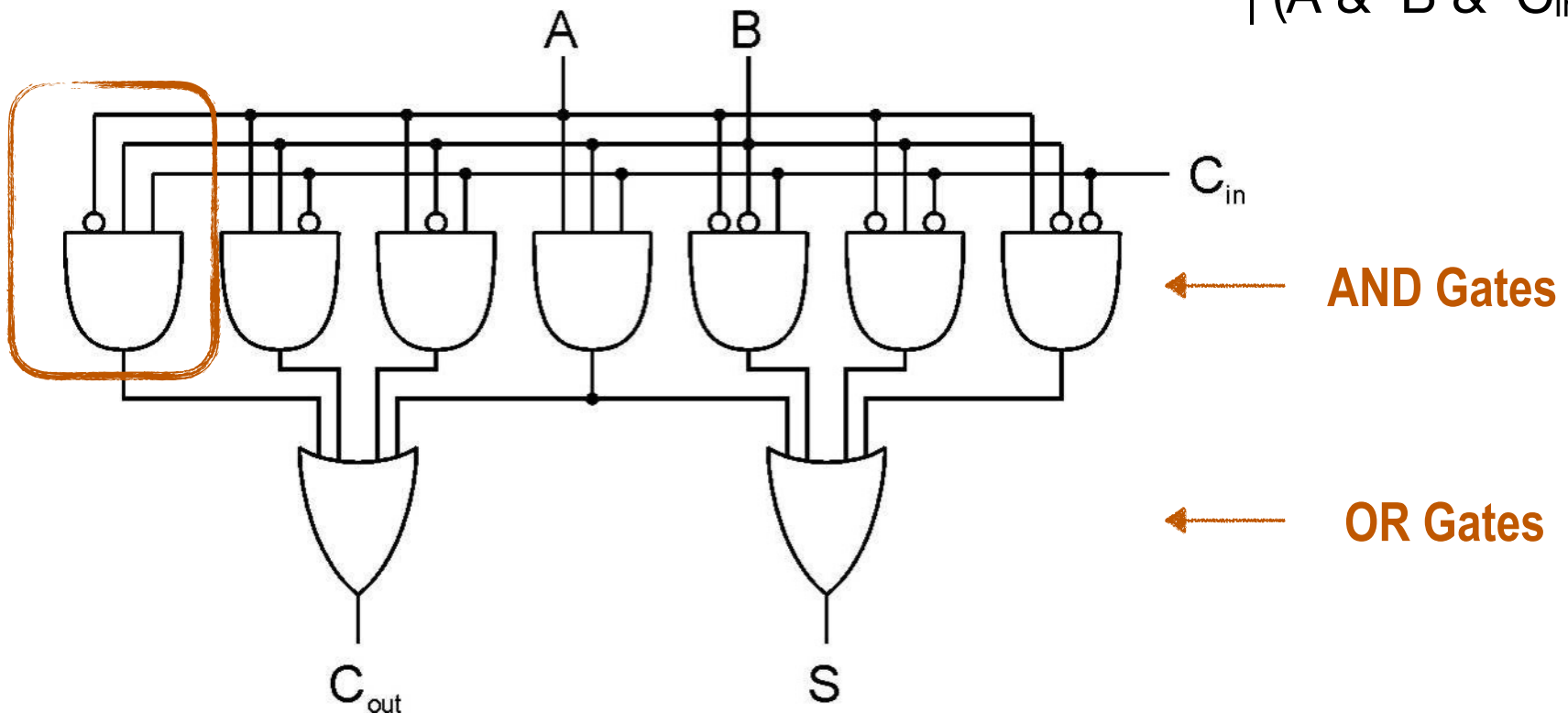
Add two bits and carry-in,  
produce one-bit sum and carry-out.



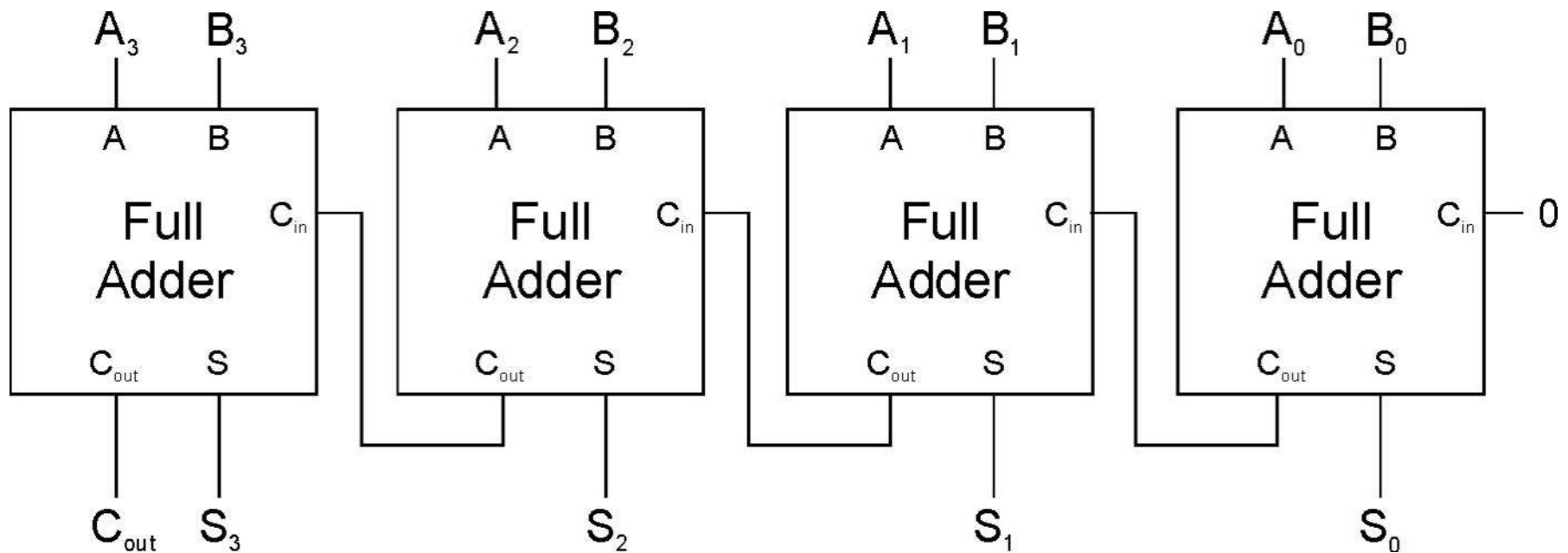
# 1-bit Full Adder

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

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

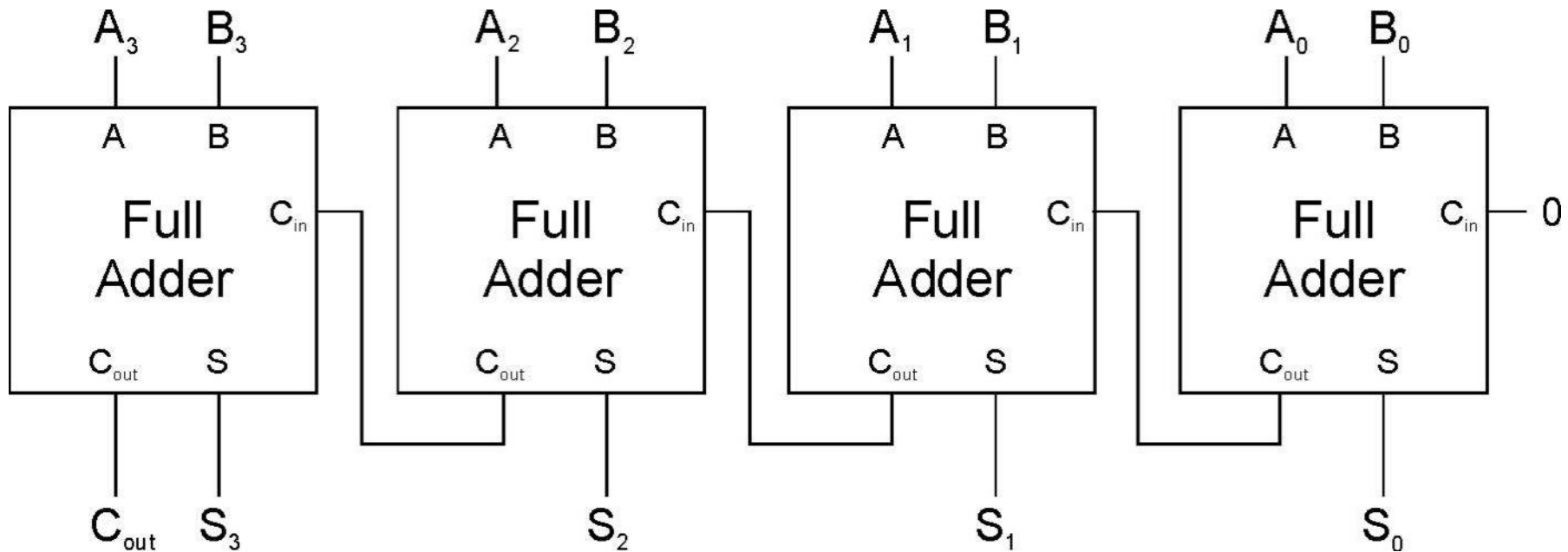


# Four-bit Adder



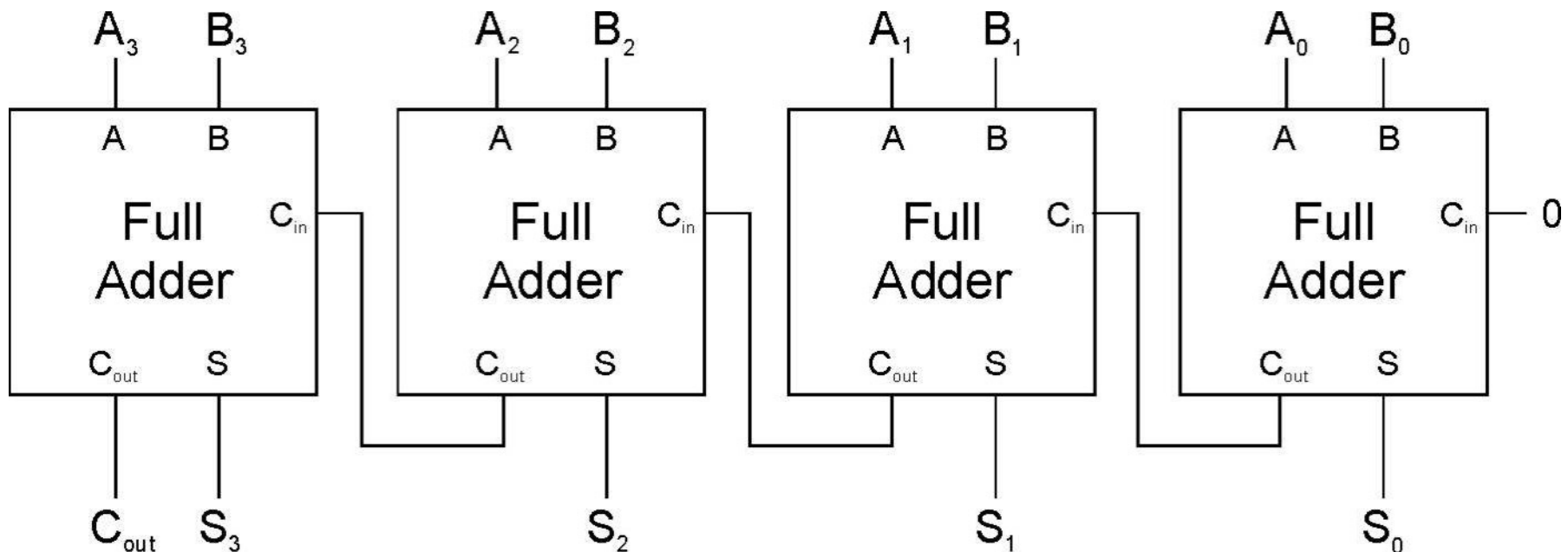
# Four-bit Adder

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

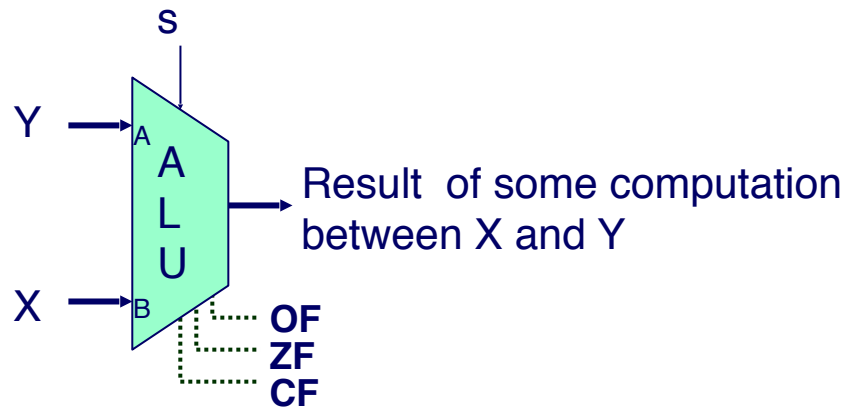


# Four-bit Adder

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



# Arithmetic Logic Unit

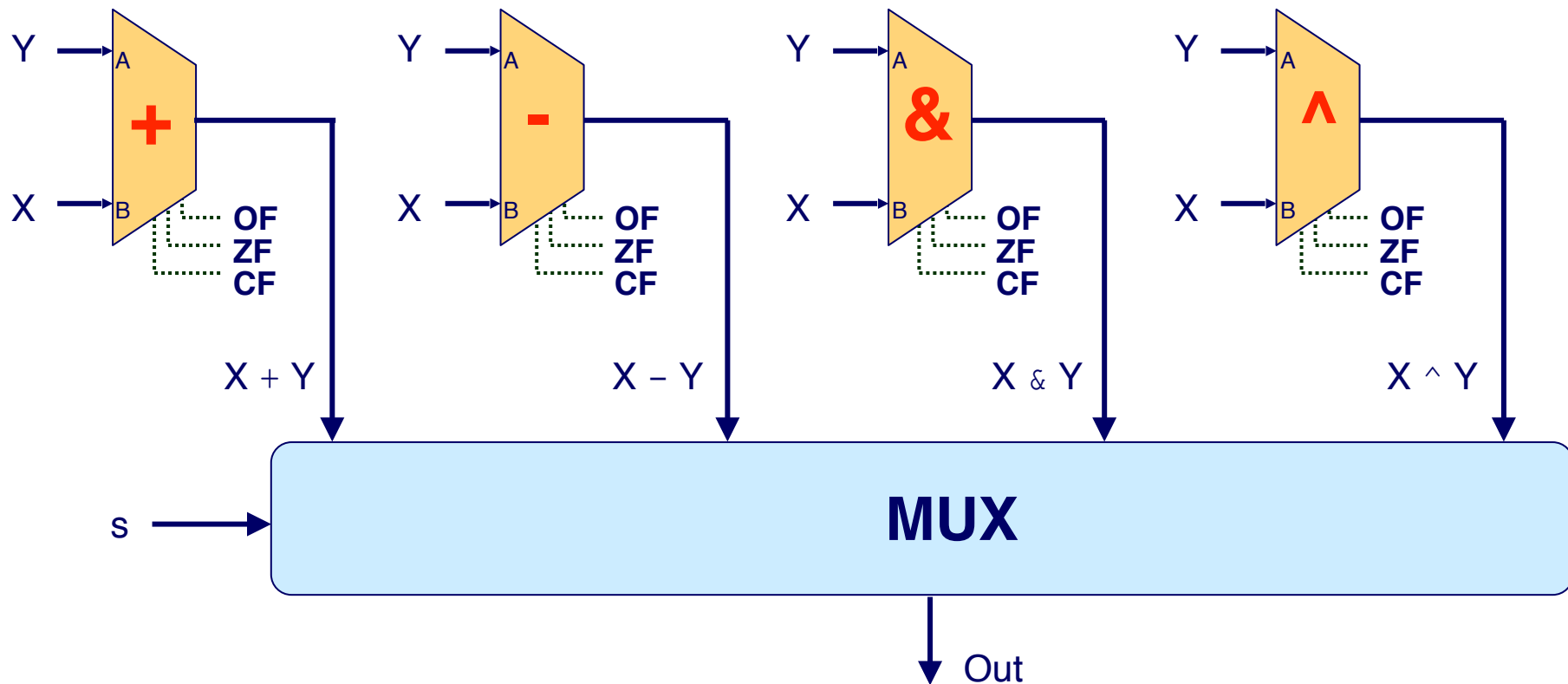


- An ALU performs multiple kinds of computations.
- The actual computation depends on the selection signal  $s$ .
- Also sets the condition codes (status flags)
- For instance:
  - $X + Y$  when  $s == 00$
  - $X - Y$  when  $s == 01$
  - $X \& Y$  when  $s == 10$
  - $X \wedge Y$  when  $s == 11$
-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

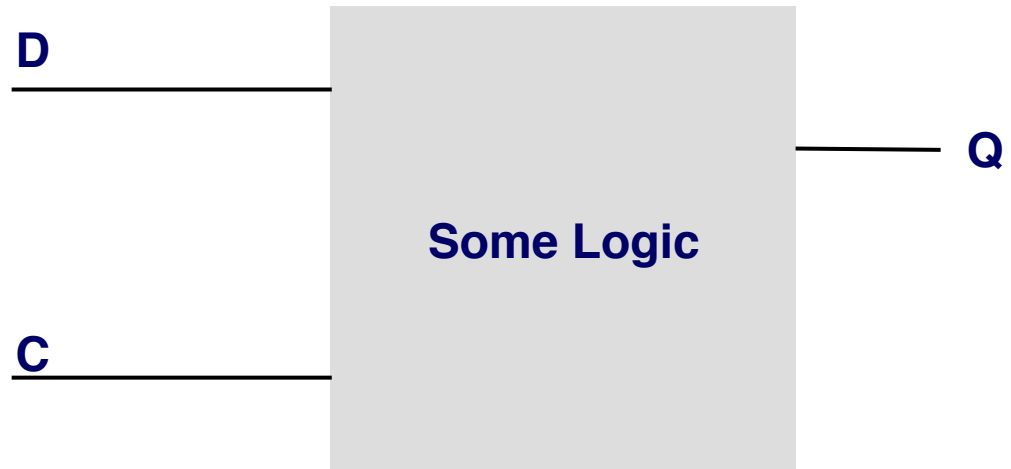
# The Need for Storing Bits

- 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.

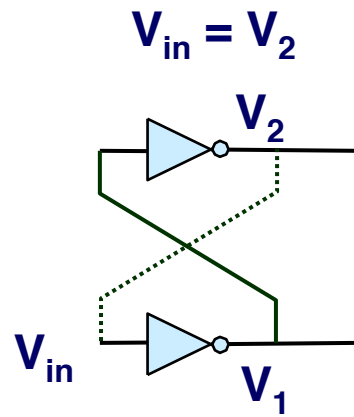
# Build a 1-Bit Storage



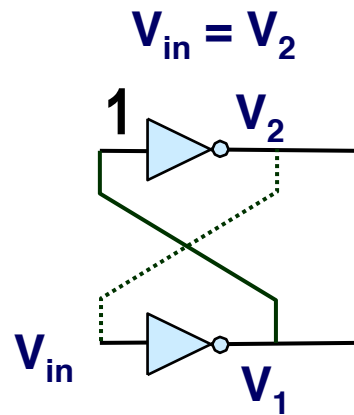
- What I would like:
  - D is the data I want to store (0 or 1)
  - C is the control signal
    - When C is 1, Q becomes D (i.e., storing the data)
    - When C is 0, Q doesn't change with D (data stored)



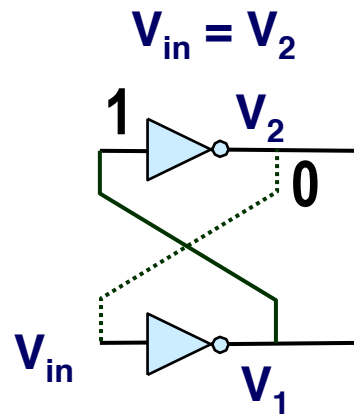
# Bitstable Element



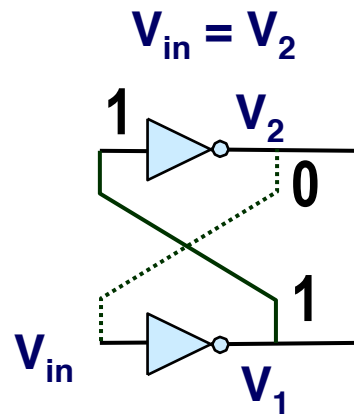
# Bitstable Element



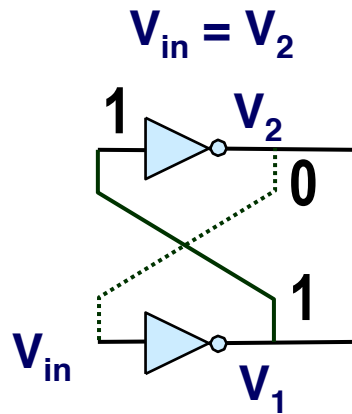
# Bitstable Element



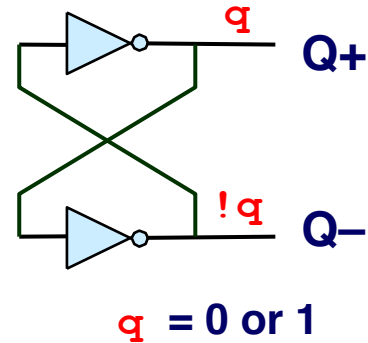
# Bitstable Element



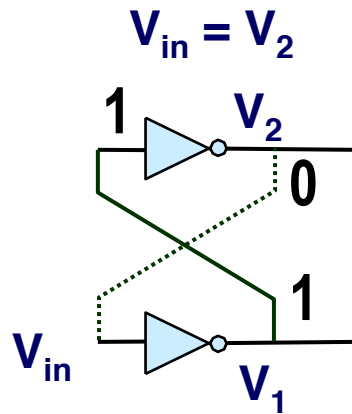
# Bitstable Element



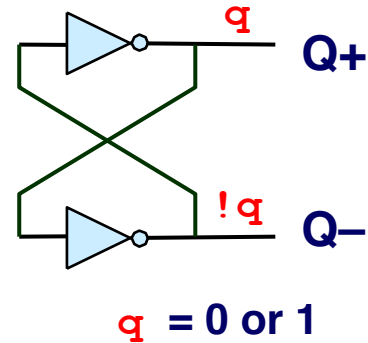
## Bistable Element



# Bitstable Element



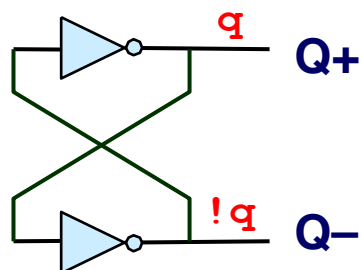
## Bistable Element



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

# Storing and Accessing 1 Bit

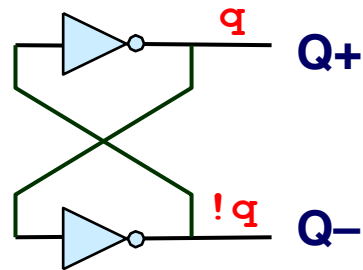
## Bistable Element



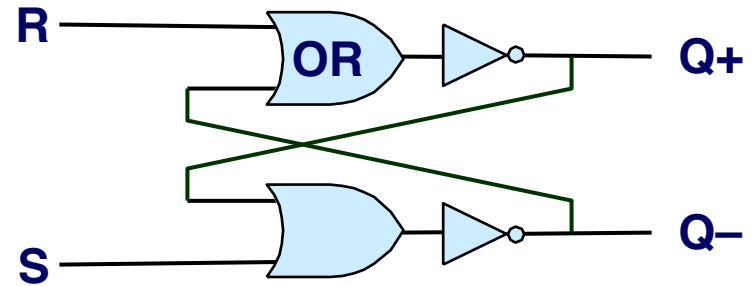
$q = 0 \text{ or } 1$

# Storing and Accessing 1 Bit

Bistable Element



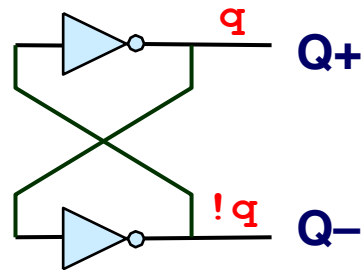
$q = 0 \text{ or } 1$



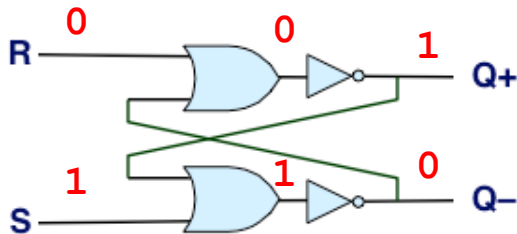
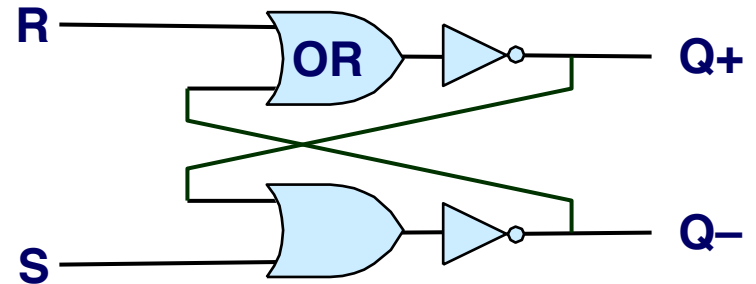


# Storing and Accessing 1 Bit

Bistable Element

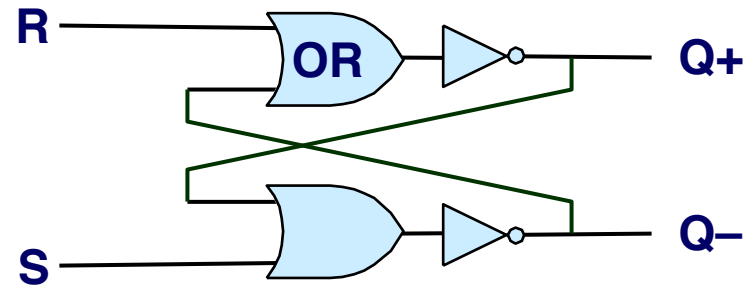
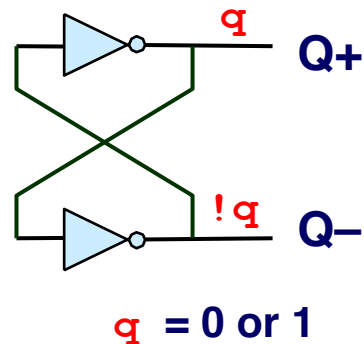


$q = 0 \text{ or } 1$

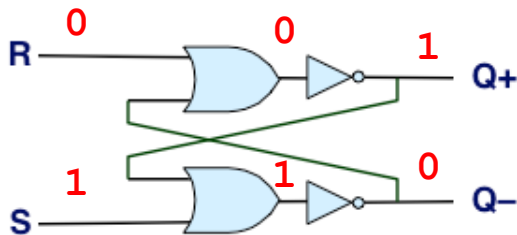


# Storing and Accessing 1 Bit

Bistable Element

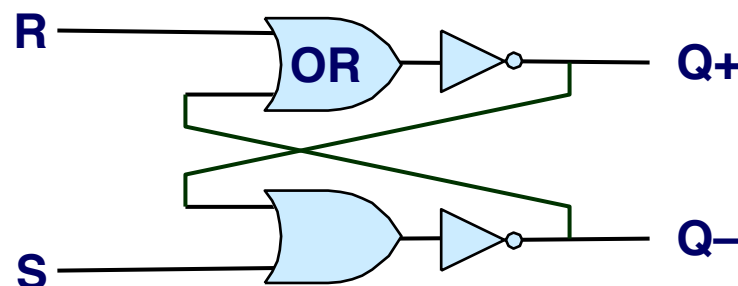
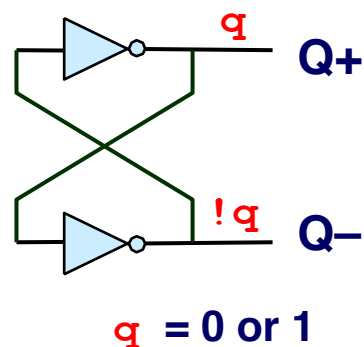


Setting  $Q+$  to 1

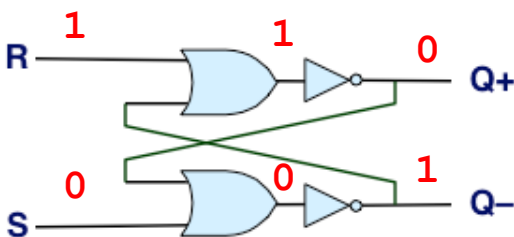
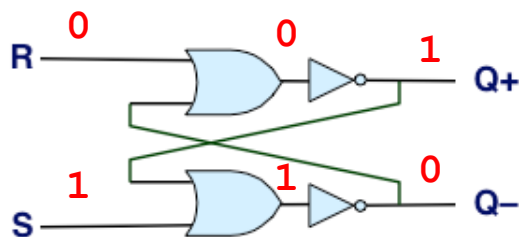


# Storing and Accessing 1 Bit

## Bistable Element

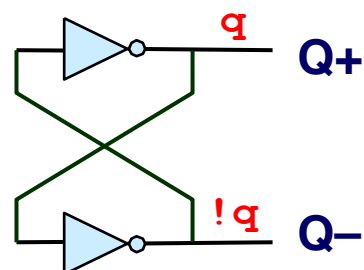


## Setting $Q+$ to 1

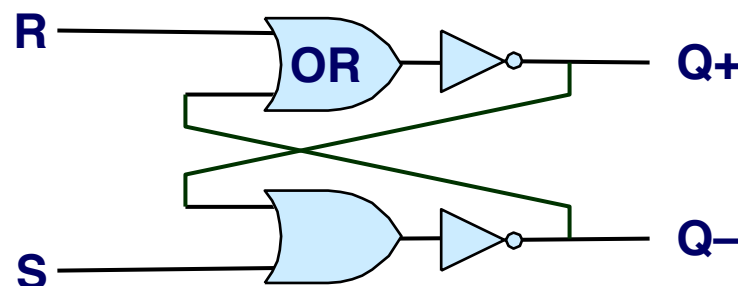


# Storing and Accessing 1 Bit

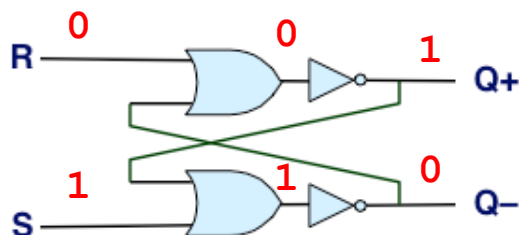
## Bistable Element



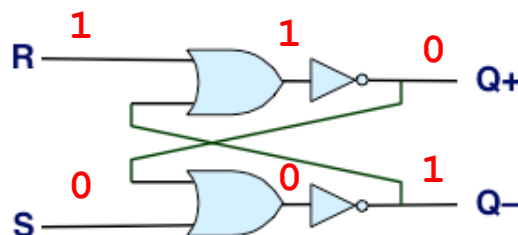
$q = 0 \text{ or } 1$



## Setting $Q+$ to 1

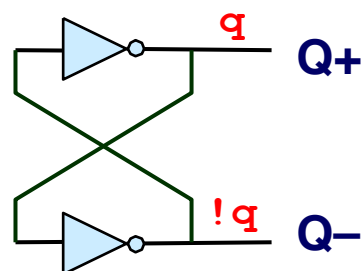


## Setting $Q+$ to 0

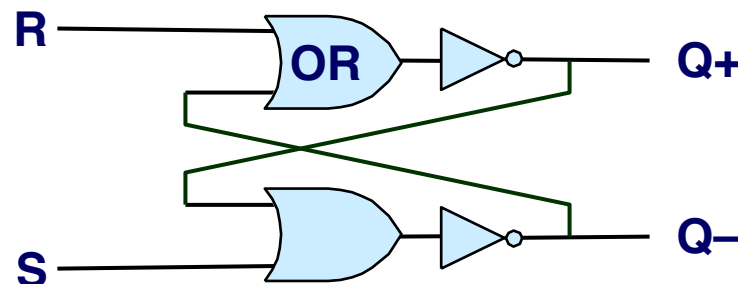


# Storing and Accessing 1 Bit

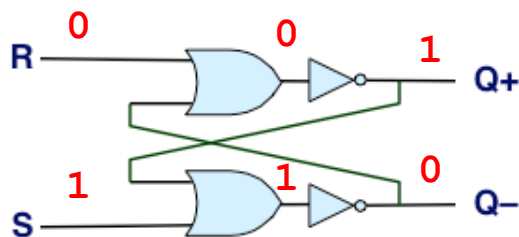
## Bistable Element



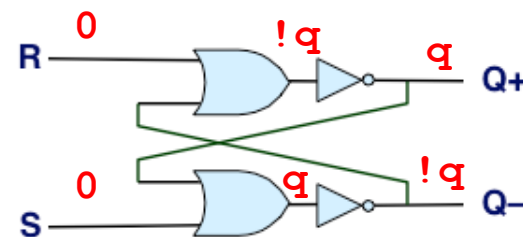
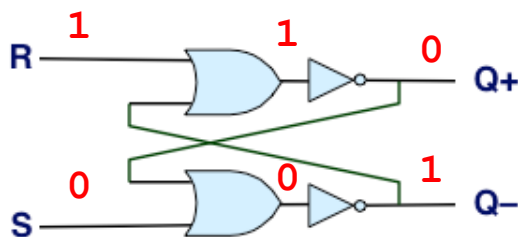
$q = 0 \text{ or } 1$



## Setting $Q+$ to 1

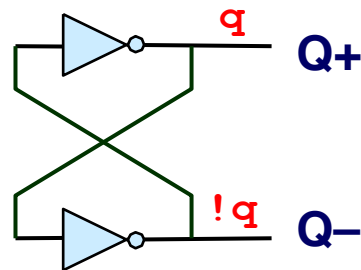


## Setting $Q+$ to 0

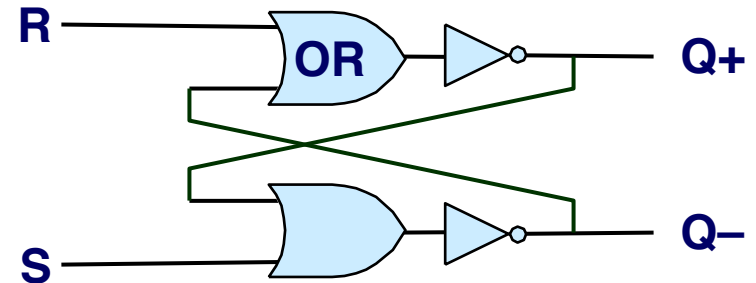


# Storing and Accessing 1 Bit

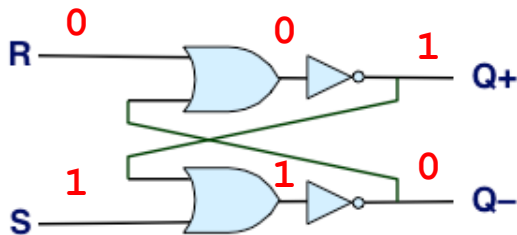
## Bistable Element



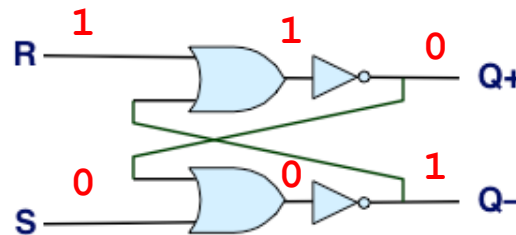
$q = 0 \text{ or } 1$



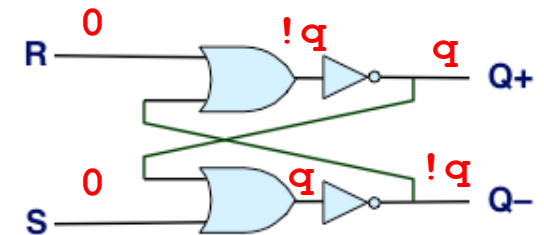
## Setting $Q+$ to 1



## Setting $Q+$ to 0

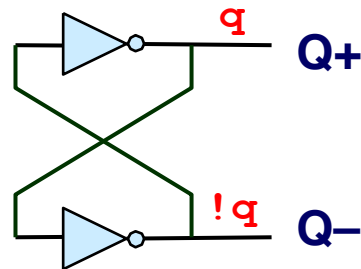


## $Q+$ value unchanged i.e., stored!



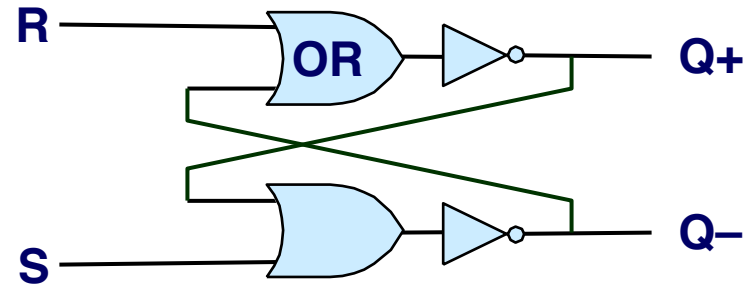
# Storing and Accessing 1 Bit

Bistable Element

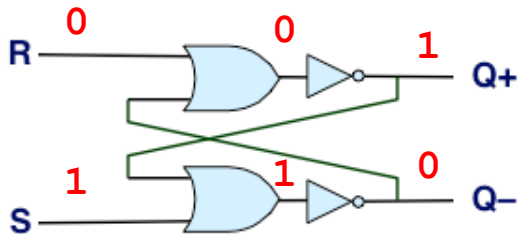


$q = 0 \text{ or } 1$

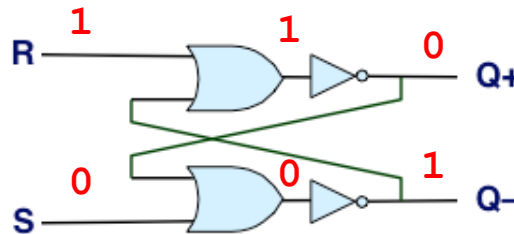
R-S Latch



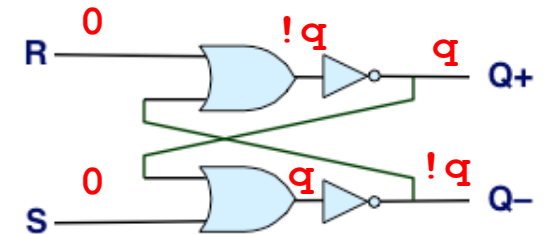
Setting  $Q+$  to 1



Setting  $Q+$  to 0

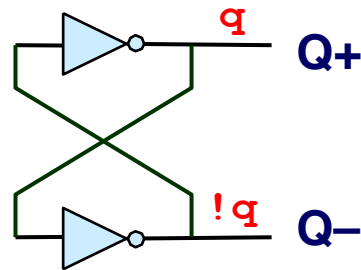


$Q+$  value unchanged  
i.e., stored!



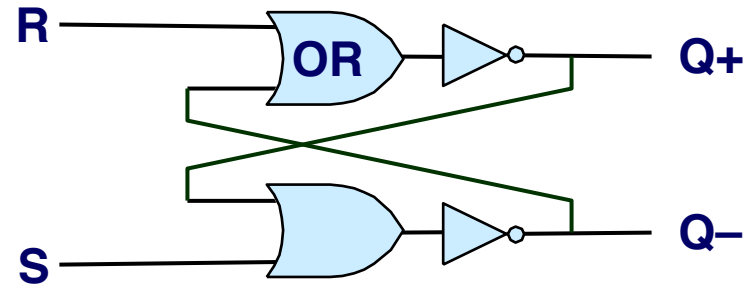
# Storing and Accessing 1 Bit

Bistable Element

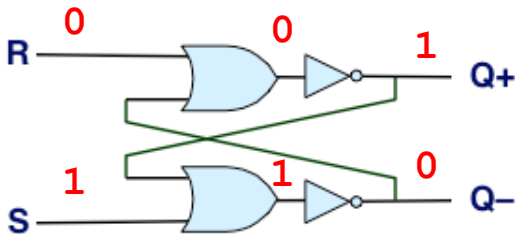


$q = 0 \text{ or } 1$

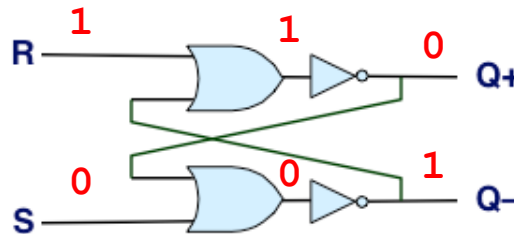
R-S Latch



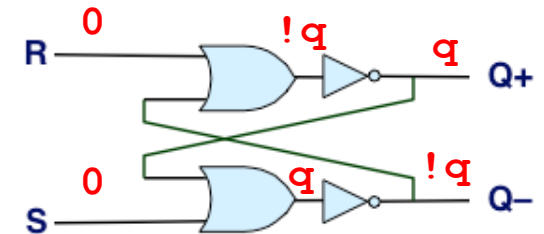
Setting  $Q+$  to 1



Setting  $Q+$  to 0



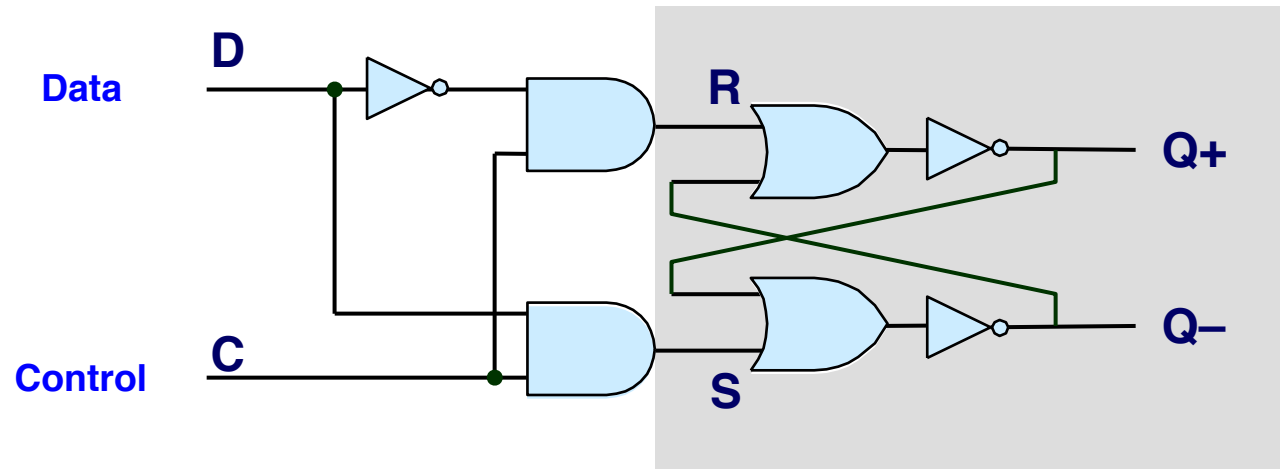
$Q+$  value unchanged  
i.e., stored!



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

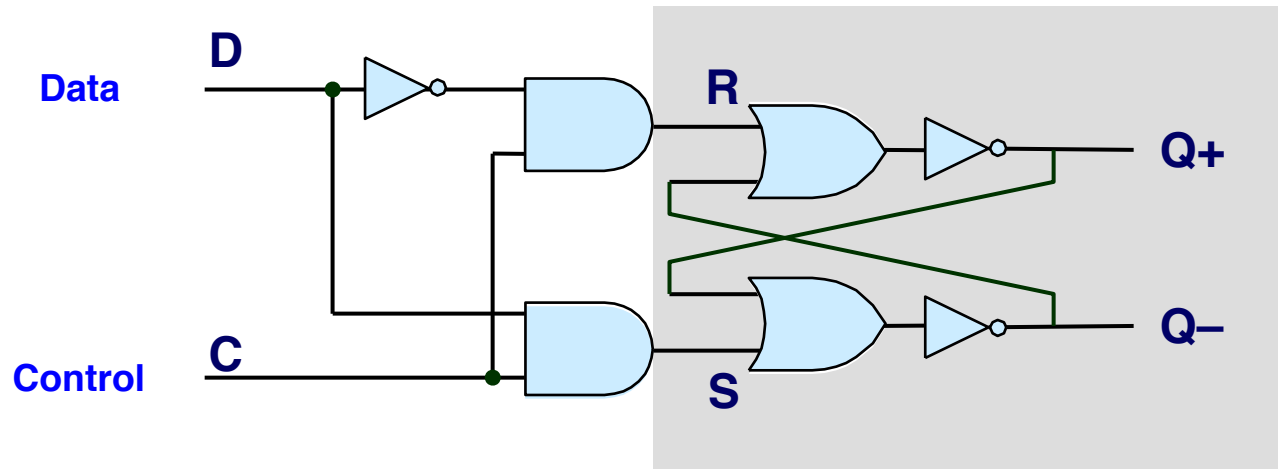


# A Better Way of Storing/Accessing 1 Bit

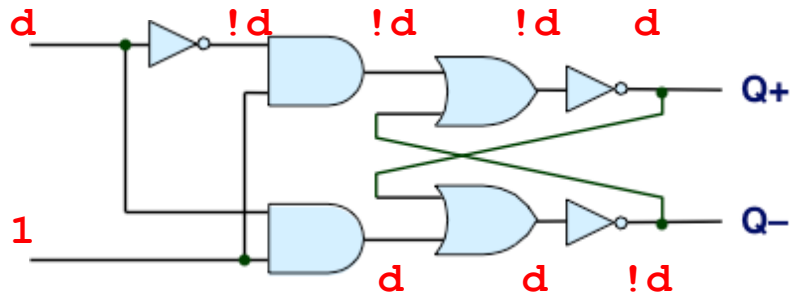


If R and S are different, Q+ is the same as S

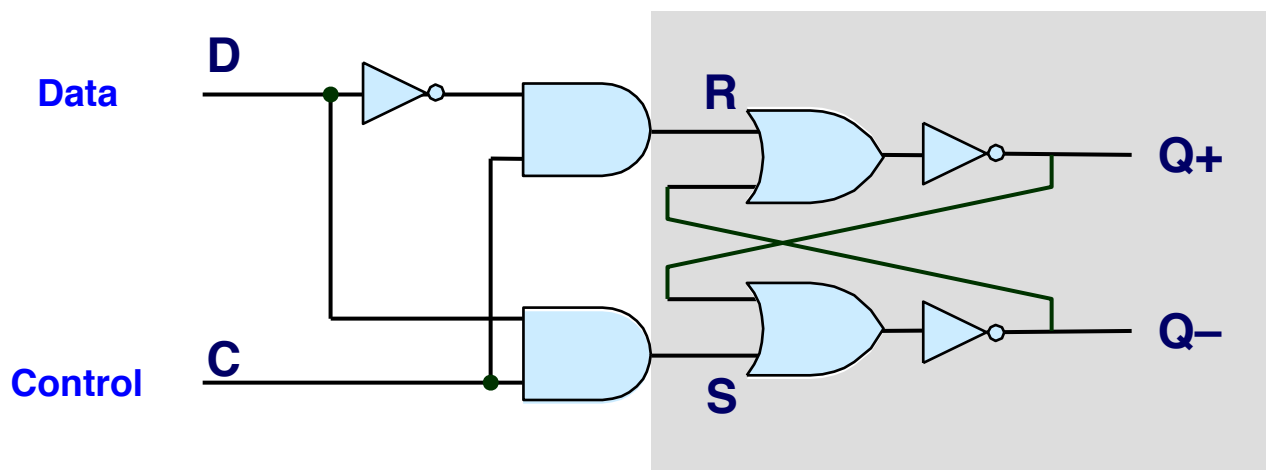
# A Better Way of Storing/Accessing 1 Bit



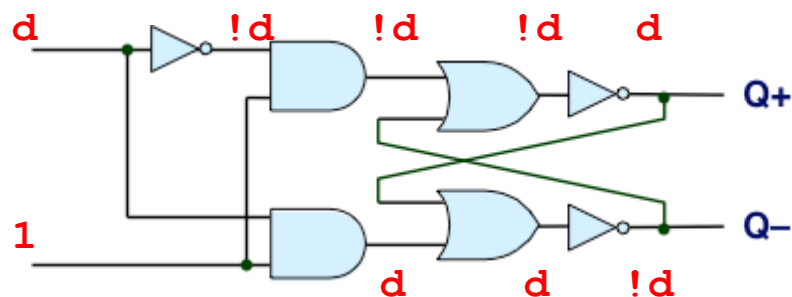
If R and S are different, Q+ is the same as S



# A Better Way of Storing/Accessing 1 Bit

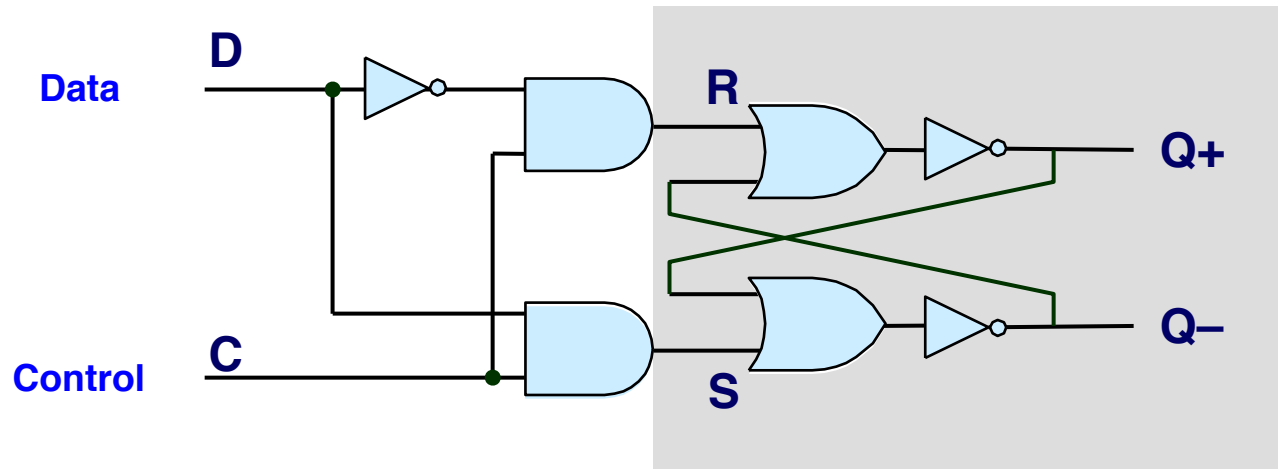


If R and S are different, Q+ is the same as S



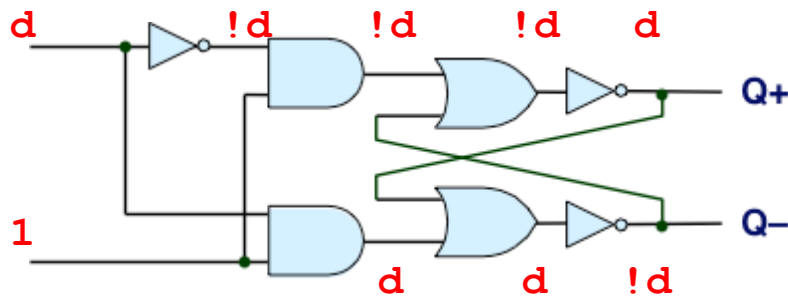
Q+ will continuously change as d changes

# A Better Way of Storing/Accessing 1 Bit



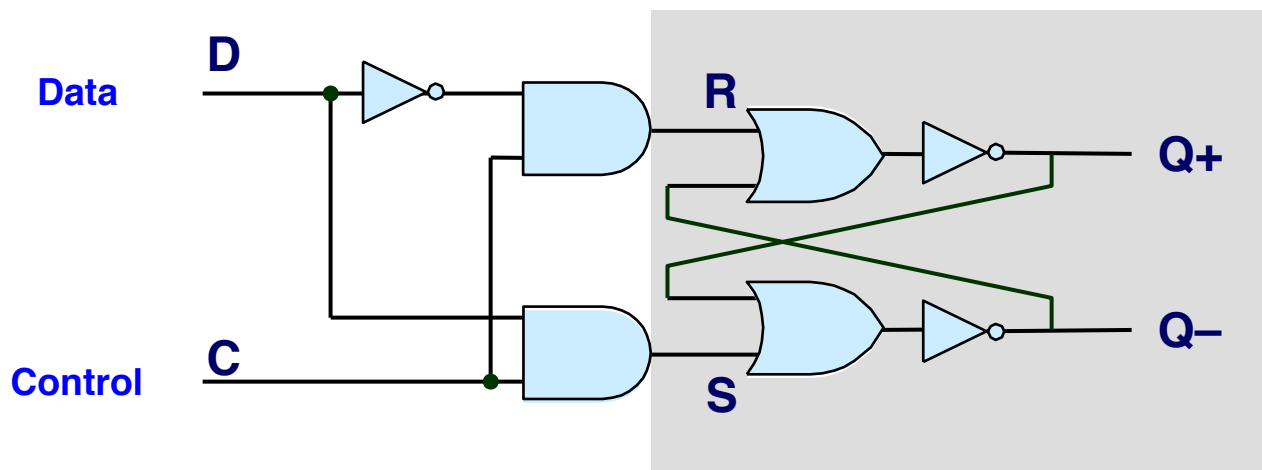
If R and S are different, Q+ is the same as S

## Storing Data (Latching)



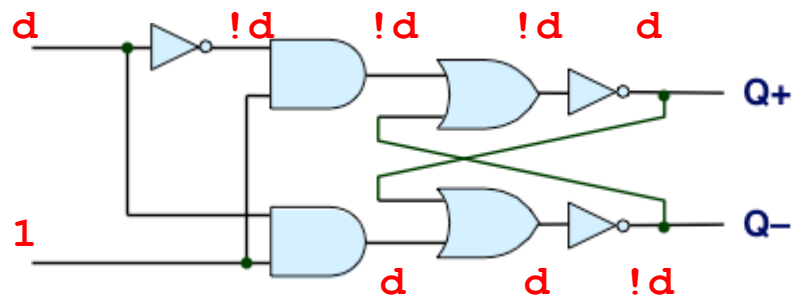
Q+ will continuously change as d changes

# A Better Way of Storing/Accessing 1 Bit

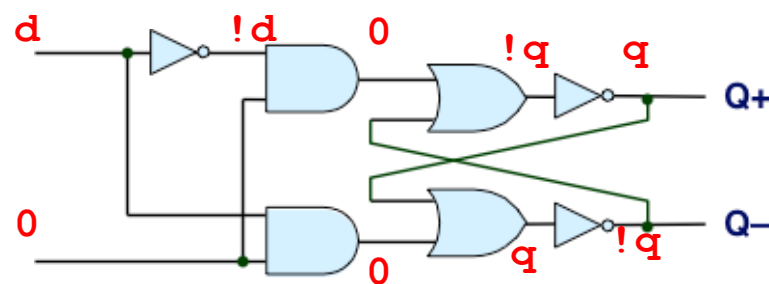


If R and S are different, Q+ is the same as S

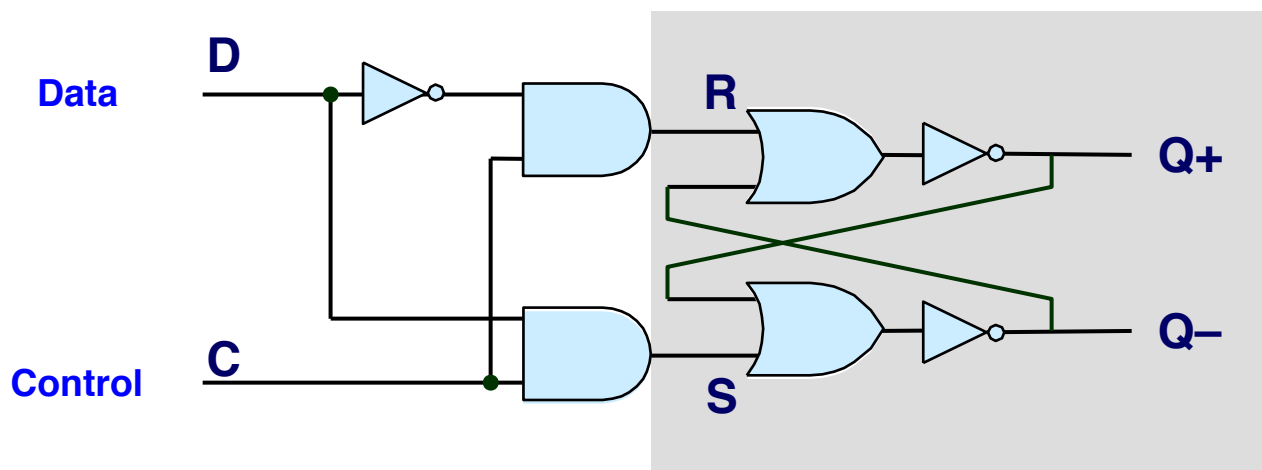
## Storing Data (Latching)



Q+ will continuously change as d changes

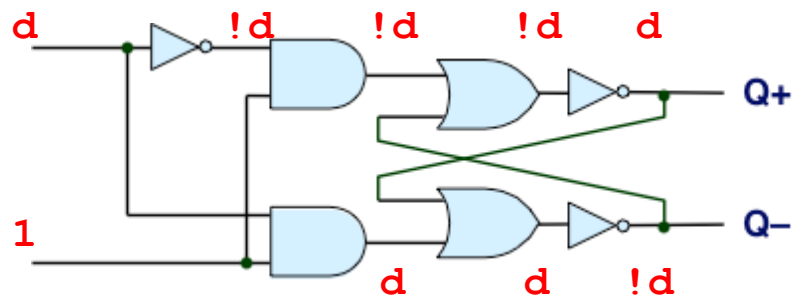


# A Better Way of Storing/Accessing 1 Bit

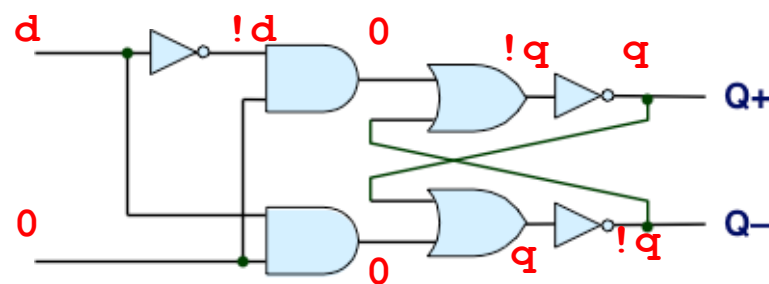


If R and S are different, Q+ is the same as S

## Storing Data (Latching)

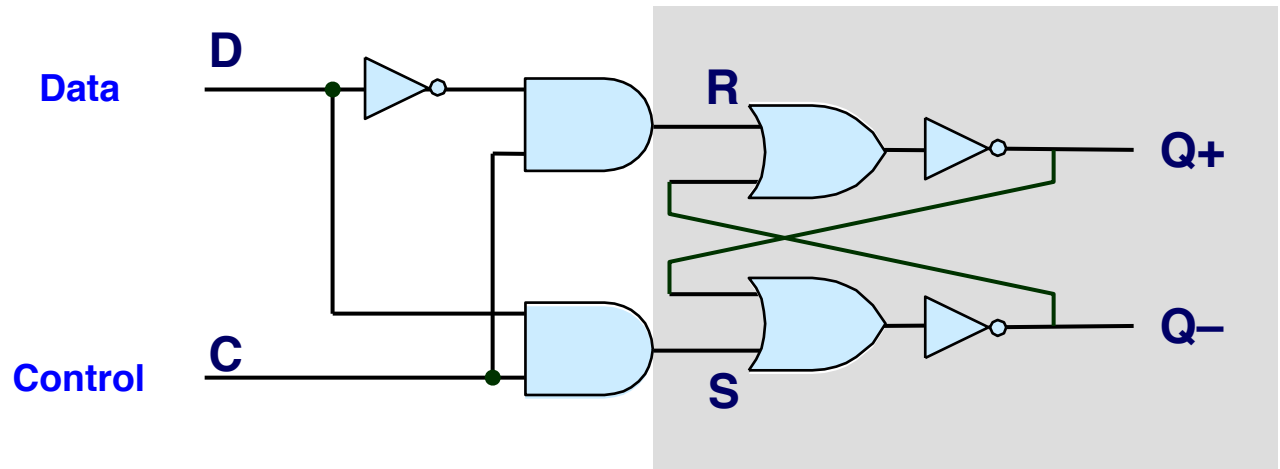


Q+ will continuously change as d changes



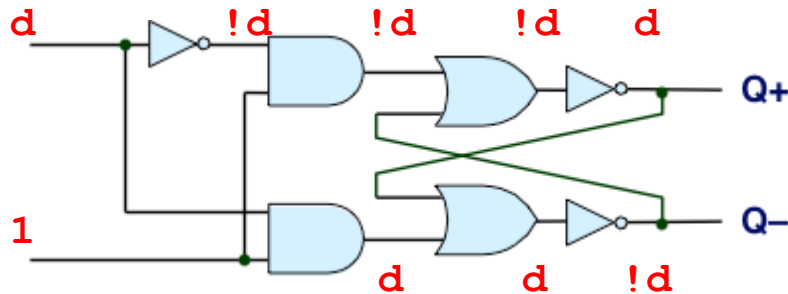
Q+ doesn't change with d

# A Better Way of Storing/Accessing 1 Bit



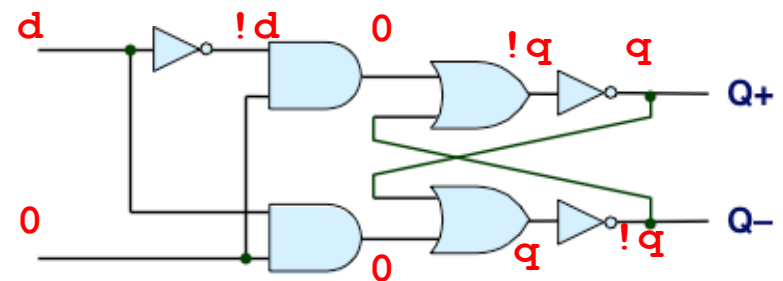
If R and S are different, Q+ is the same as S

**Storing Data (Latching)**



Q+ will continuously change as d changes

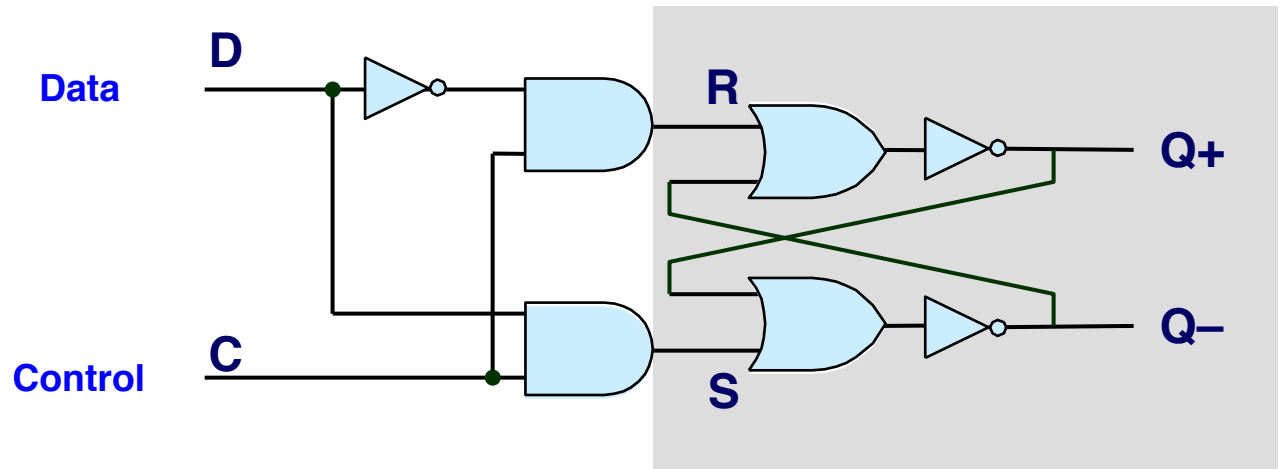
**Holding Data**



Q+ doesn't change with d

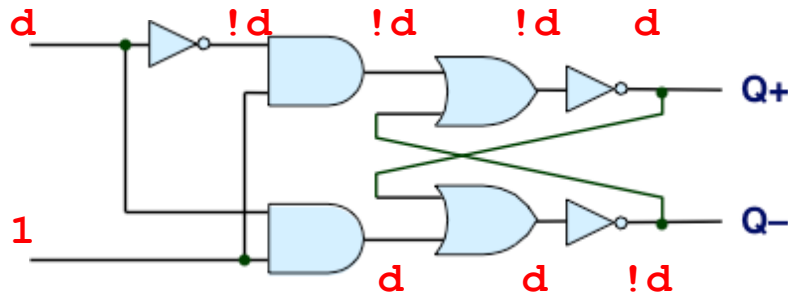
# A Better Way of Storing/Accessing 1 Bit

D Latch



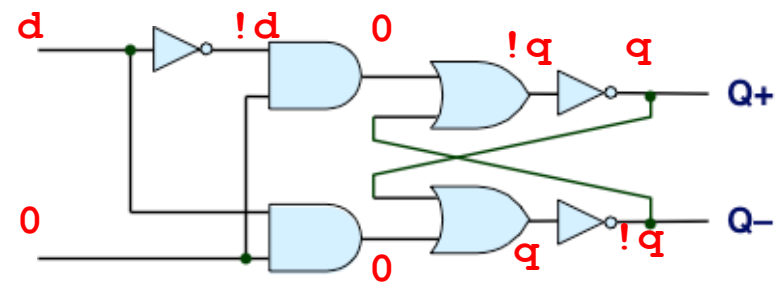
If R and S are different, Q+ is the same as S

Storing Data (Latching)



Q+ will continuously change as d changes

Holding Data

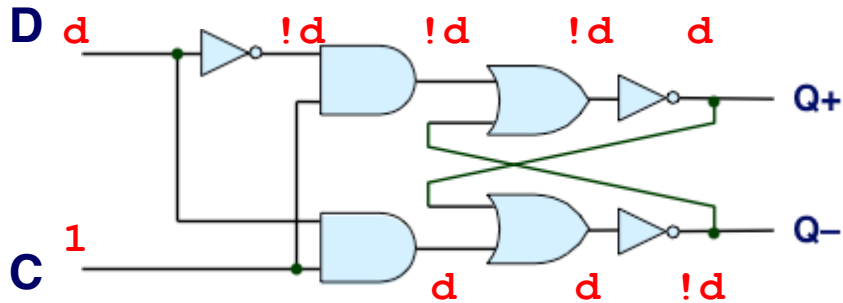


Q+ doesn't change with d

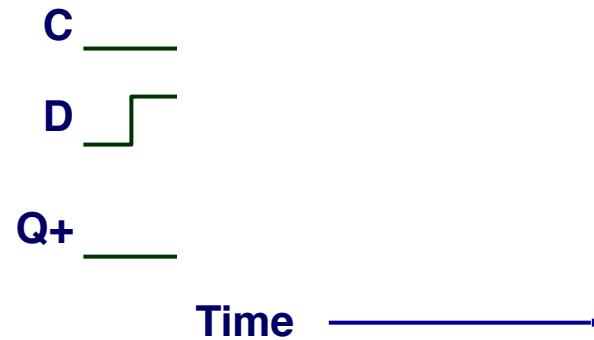


# D-Latch is “Transparent”

Latching

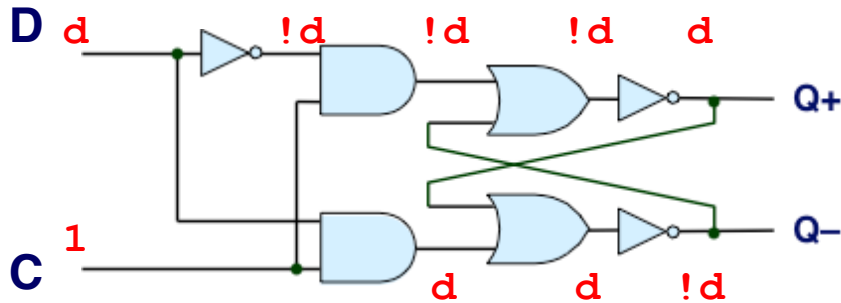


Changing D

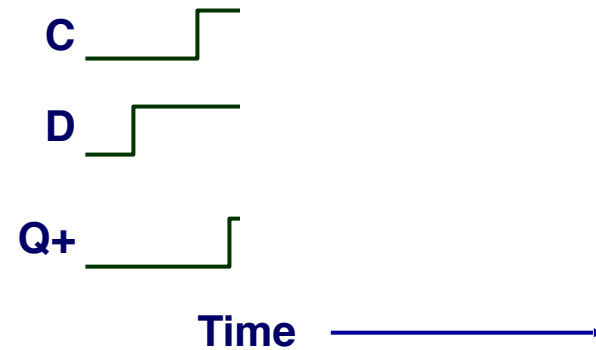


# D-Latch is “Transparent”

## Latching

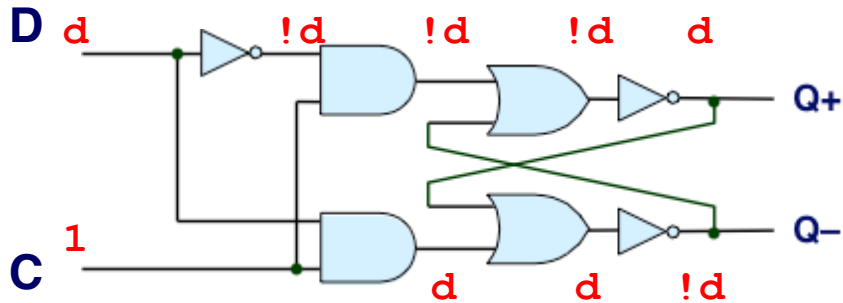


## Changing D

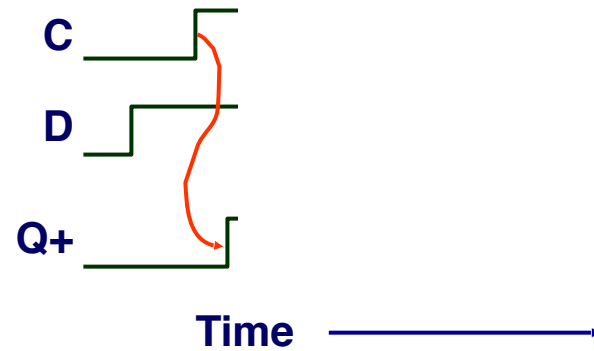


# D-Latch is “Transparent”

## Latching

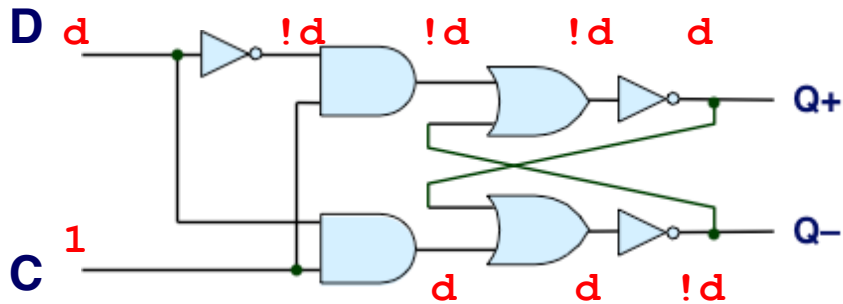


## Changing D

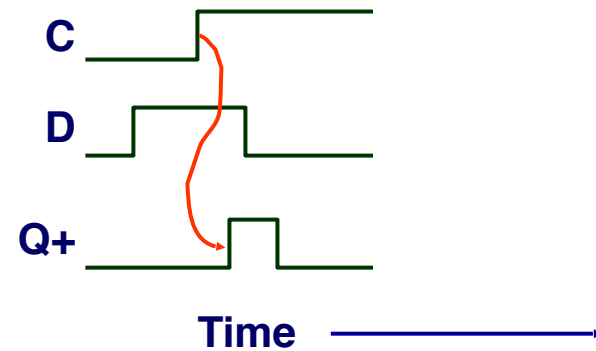


# D-Latch is “Transparent”

## Latching

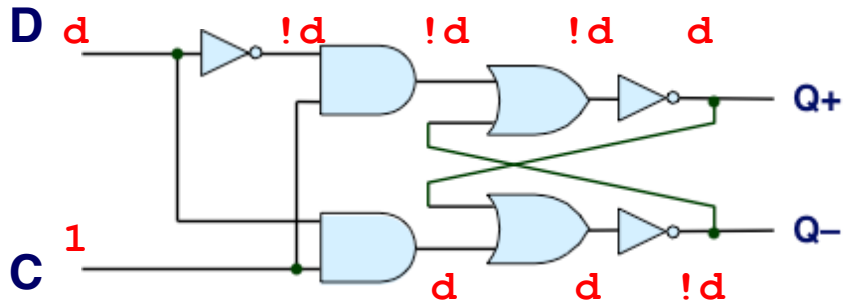


## Changing D

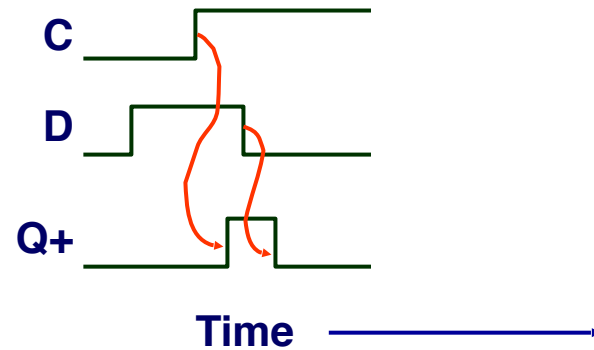


# D-Latch is “Transparent”

## Latching

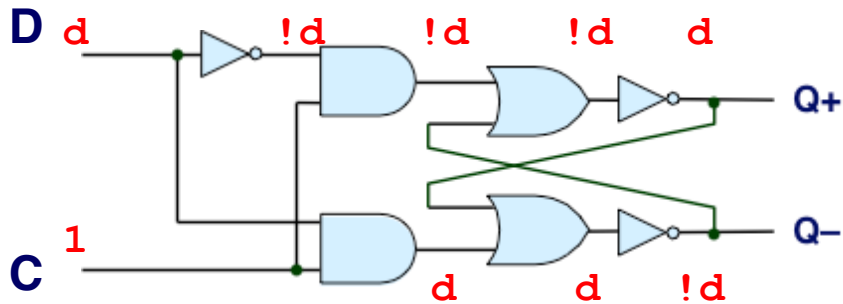


## Changing D

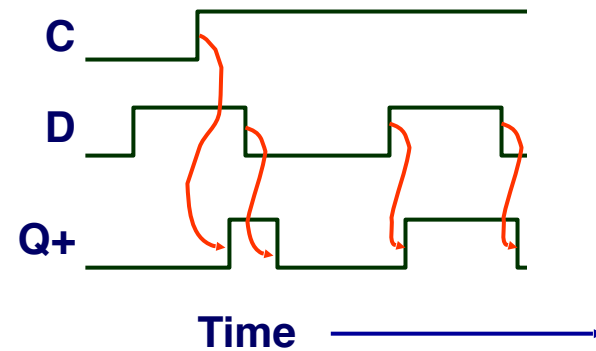


# D-Latch is “Transparent”

## Latching

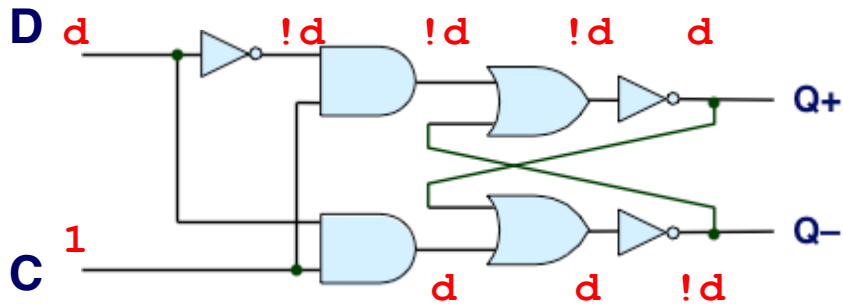


## Changing D

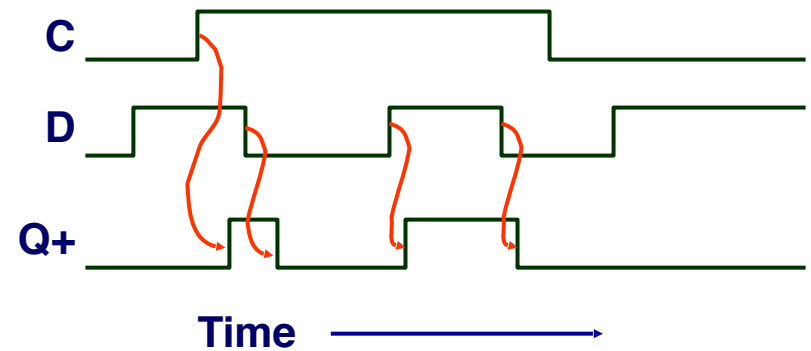


# D-Latch is “Transparent”

## Latching

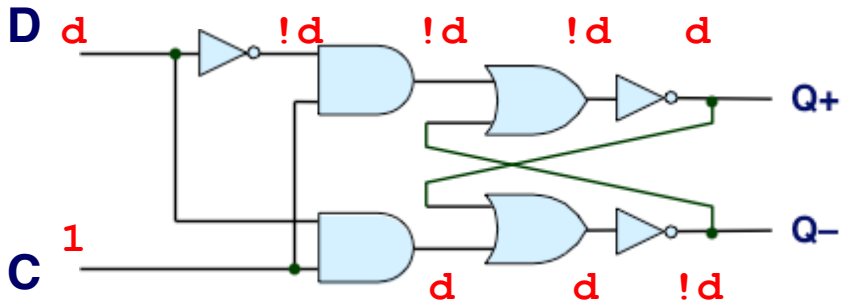


## Changing D

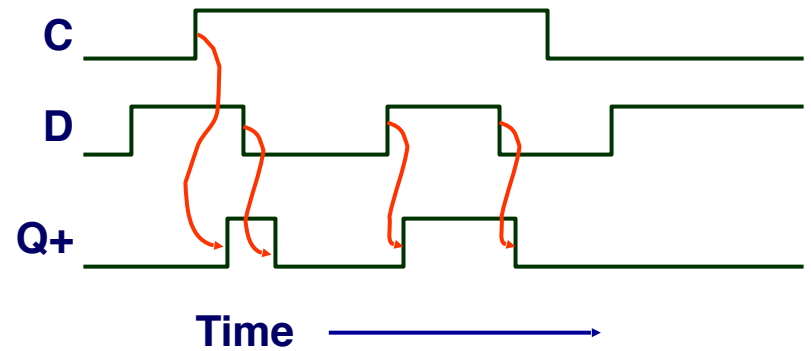


# D-Latch is “Transparent”

# Latching



## Changing D

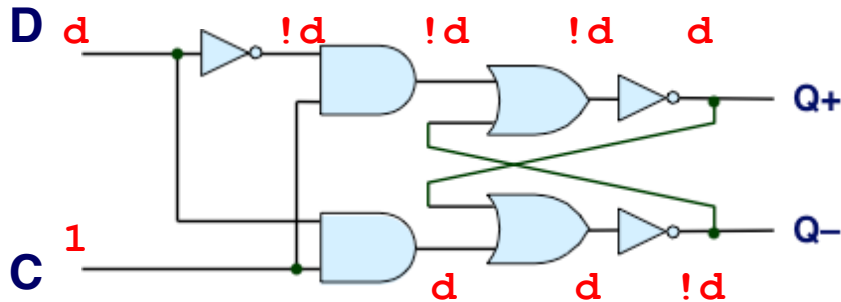


- When you want to store **d**, you have to first set **C** to 1, and then set **d**

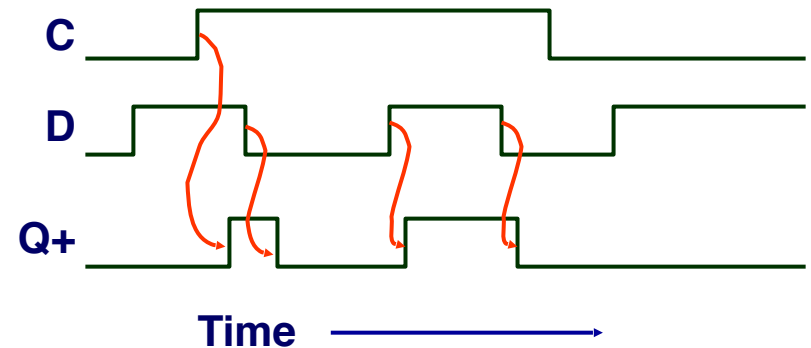


# D-Latch is “Transparent”

## Latching



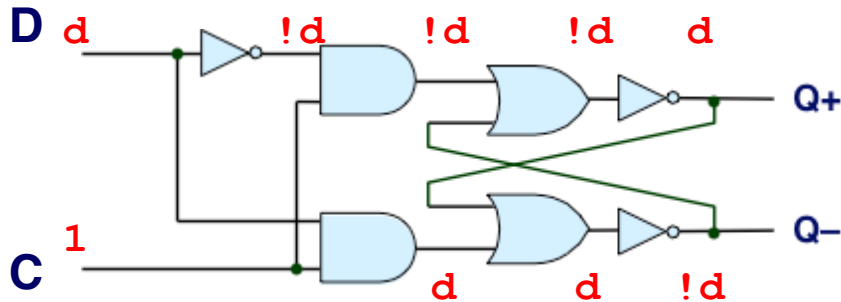
## Changing D



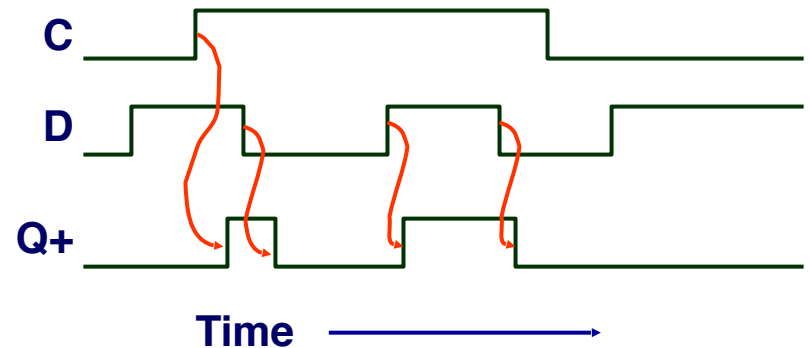
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold C for a while until the signal is fully propagated

# D-Latch is “Transparent”

## Latching



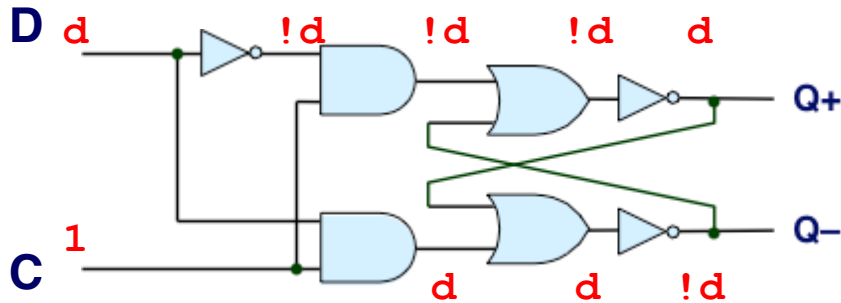
## Changing D



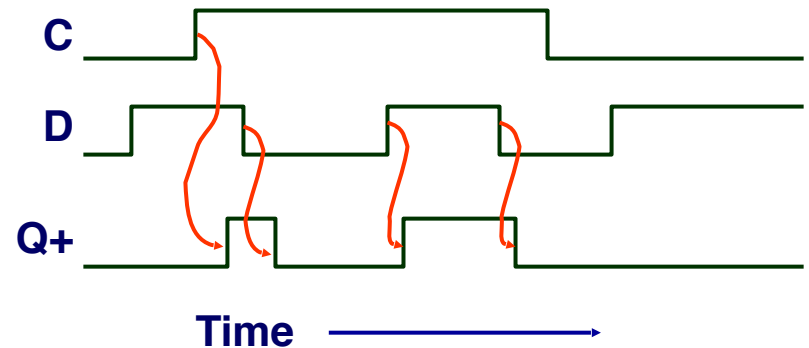
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold **C** for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D**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

# D-Latch is “Transparent”

## Latching



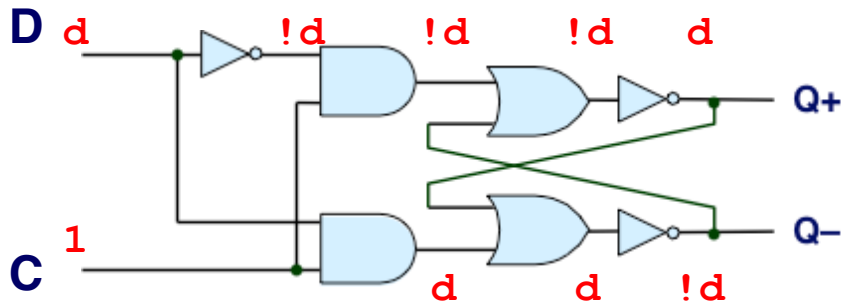
## Changing D



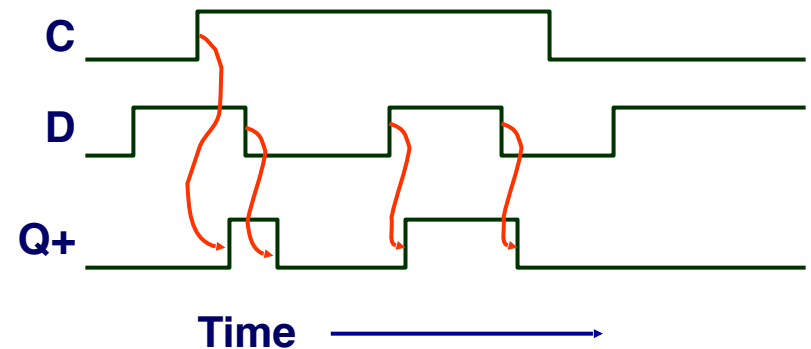
- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold **C** for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D**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

# D-Latch is “Transparent”

## Latching

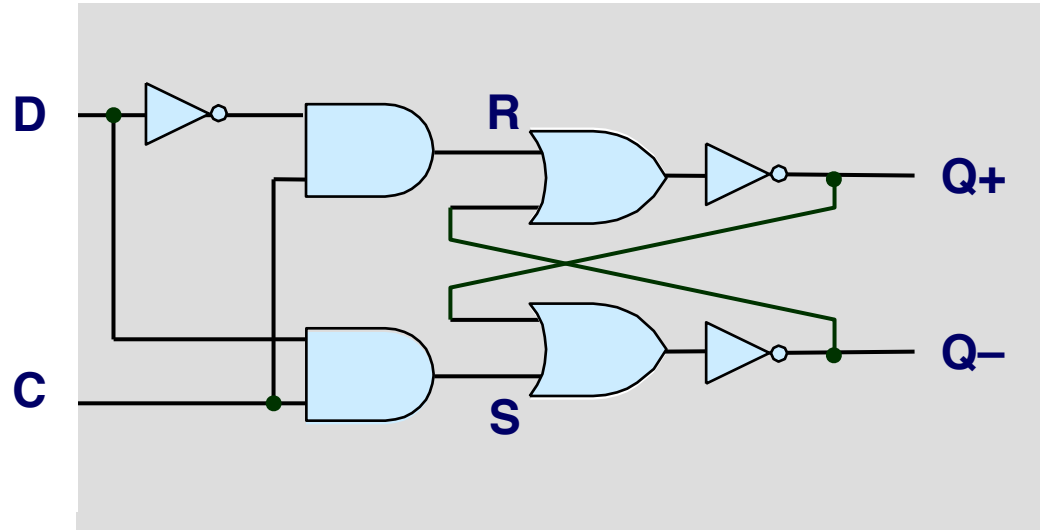


## Changing D

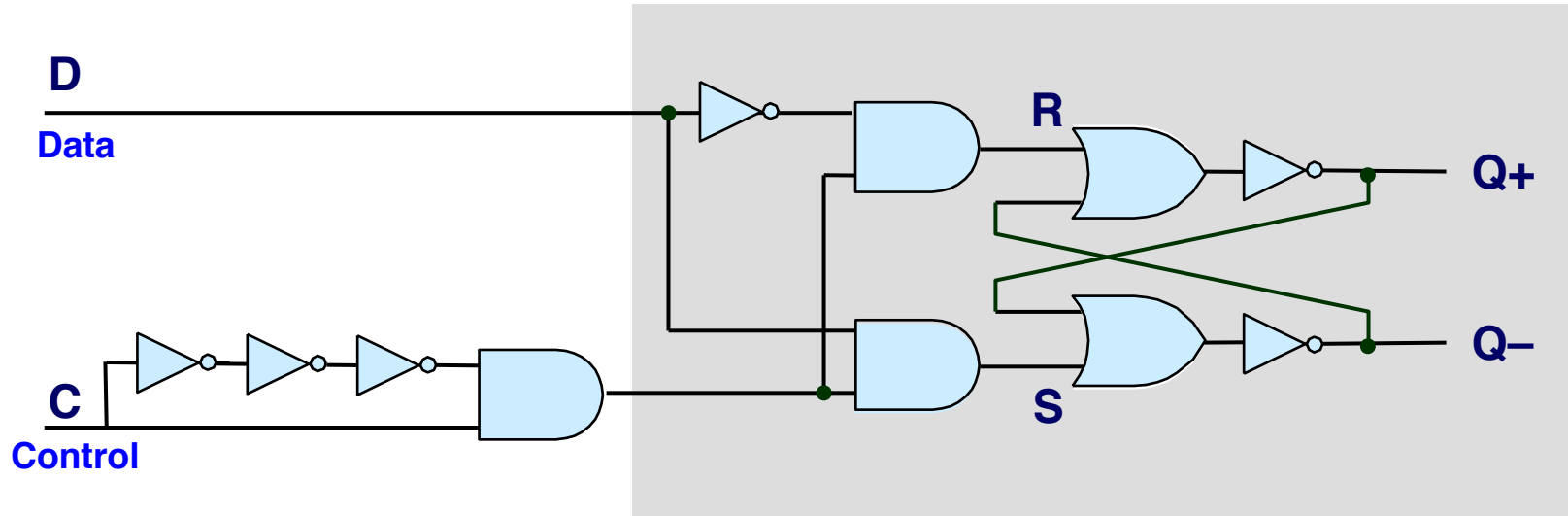


- When you want to store **d**, you have to first set **C** to 1, and then set **d**
- There is a propagation delay of the combinational circuit from **D** to **Q+** and **Q-**. So hold **C** for a while until the signal is fully propagated
- Then set **C** to 0. Value latched depends on value of **D**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

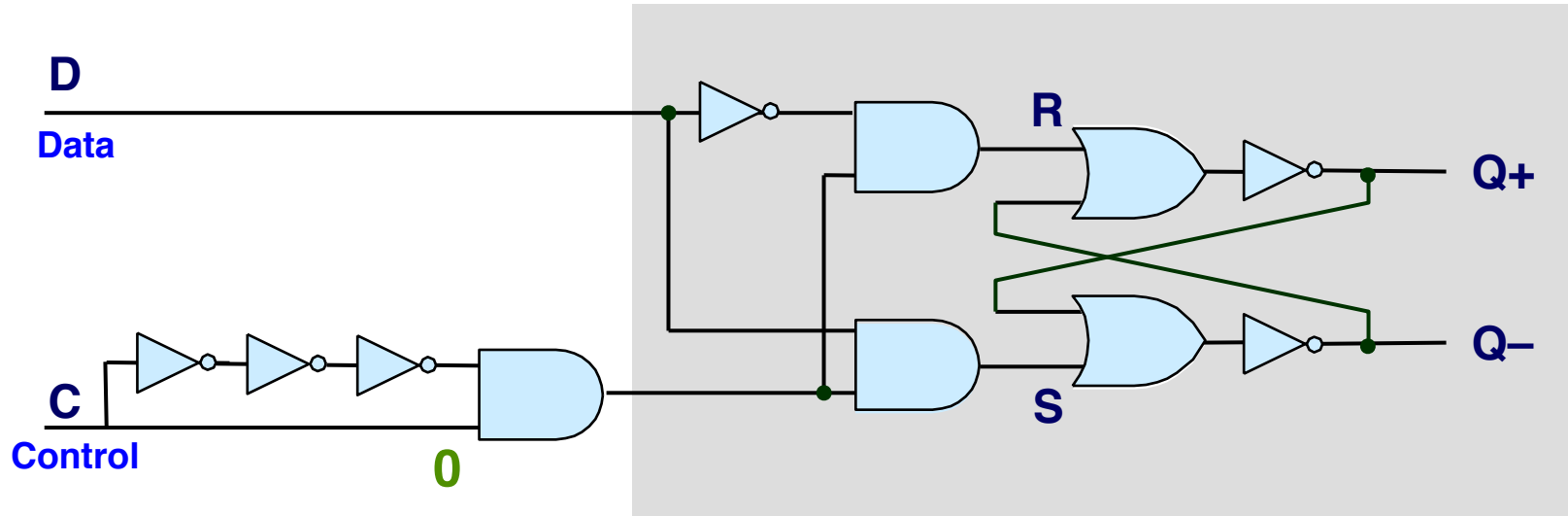
# Edge-Triggered Latch (Flip-Flop)



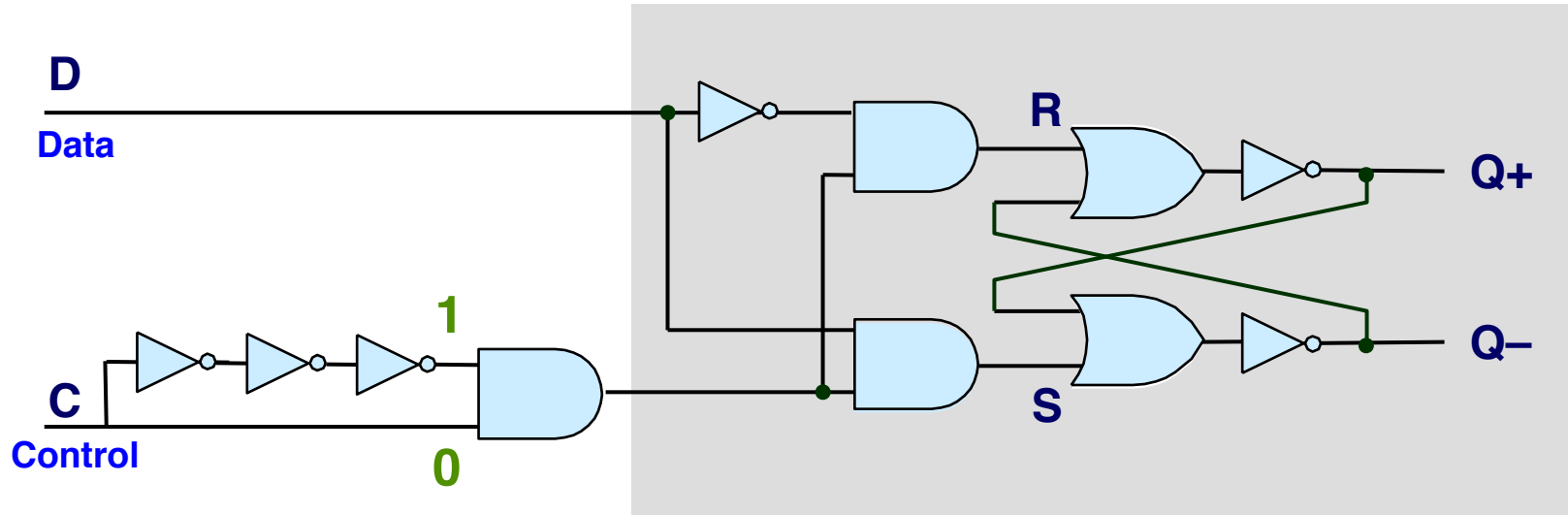
# Edge-Triggered Latch (Flip-Flop)



# Edge-Triggered Latch (Flip-Flop)

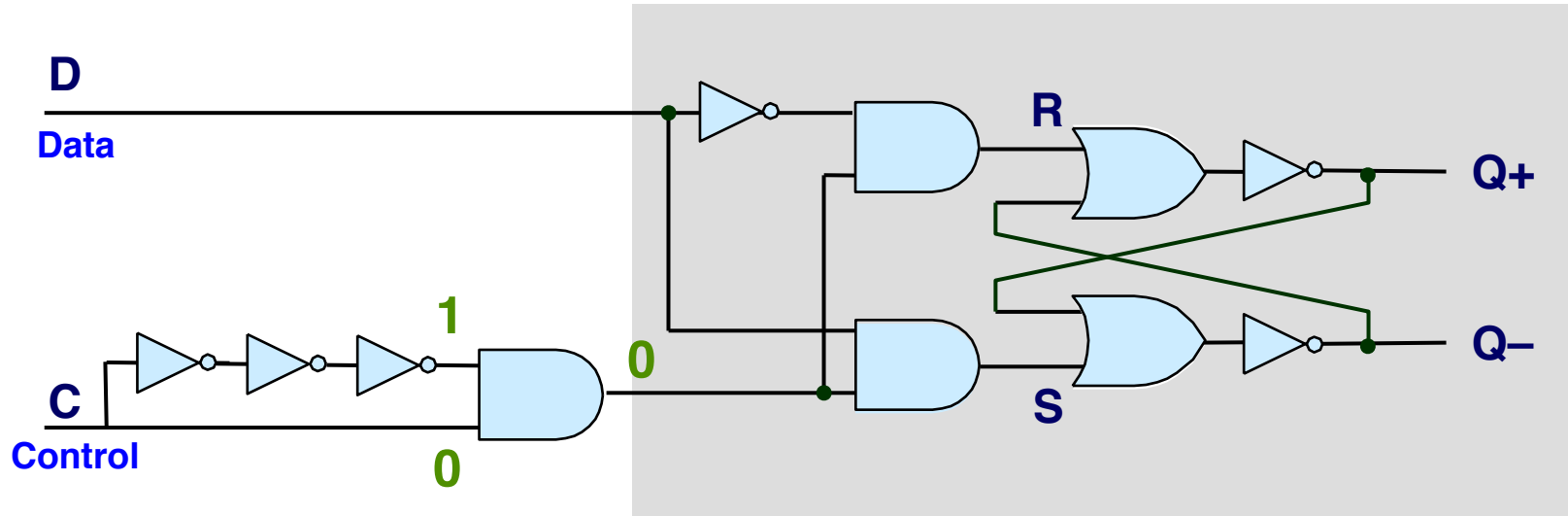


# Edge-Triggered Latch (Flip-Flop)

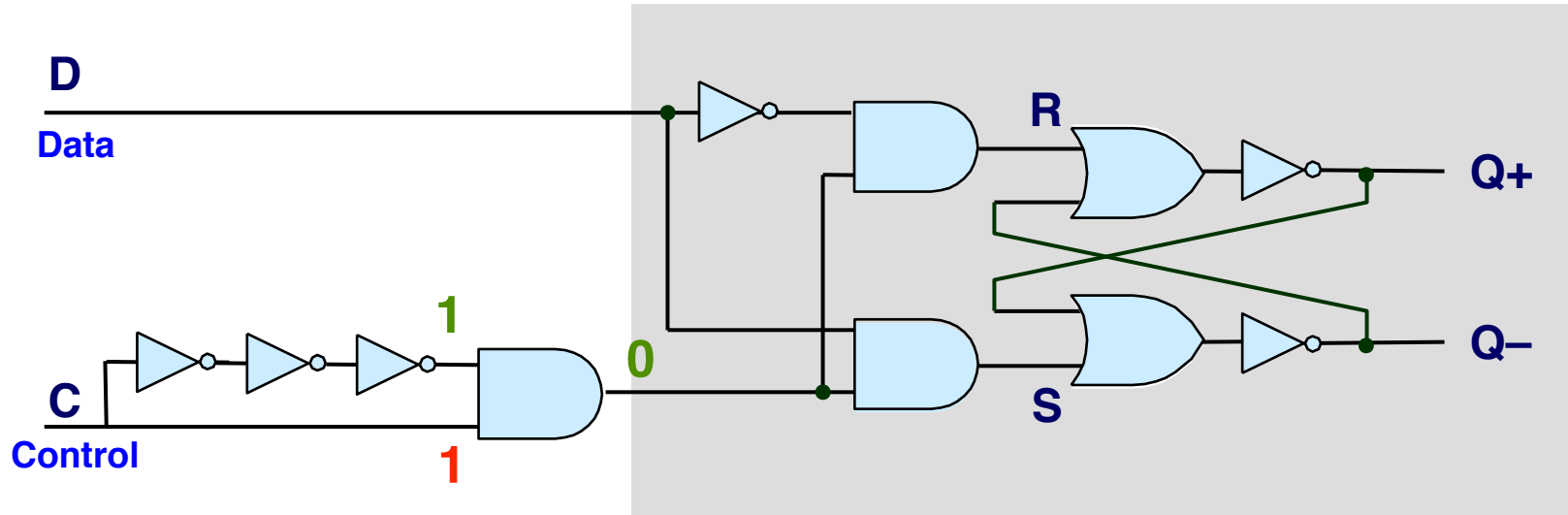




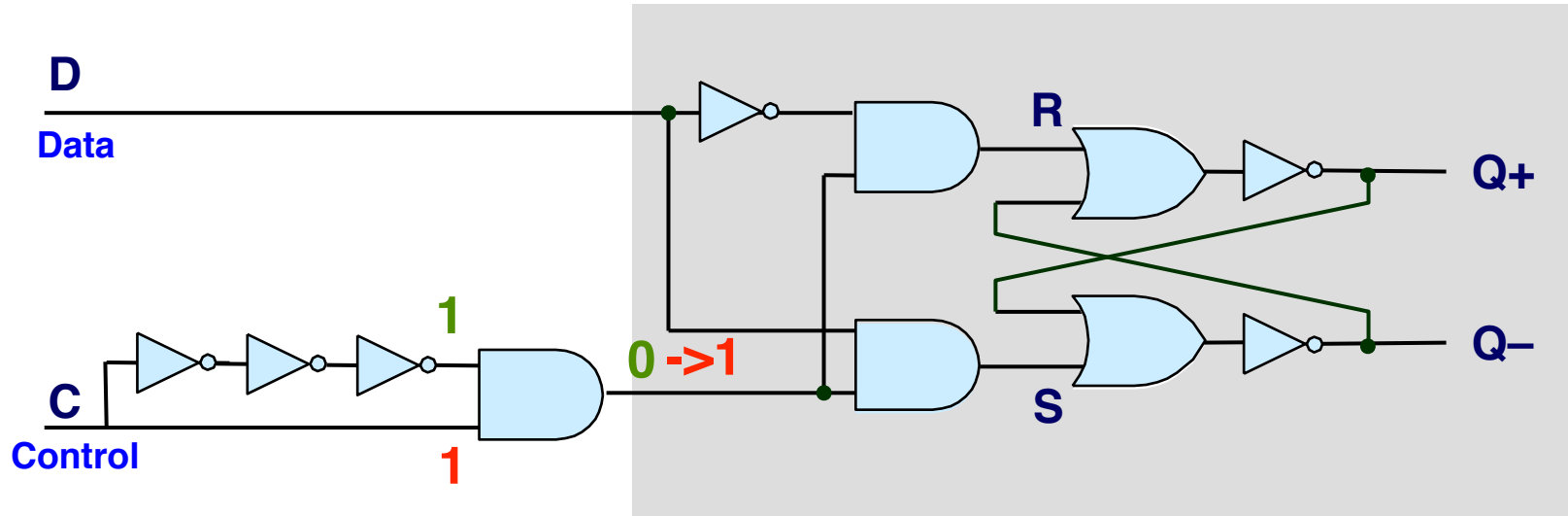
# Edge-Triggered Latch (Flip-Flop)



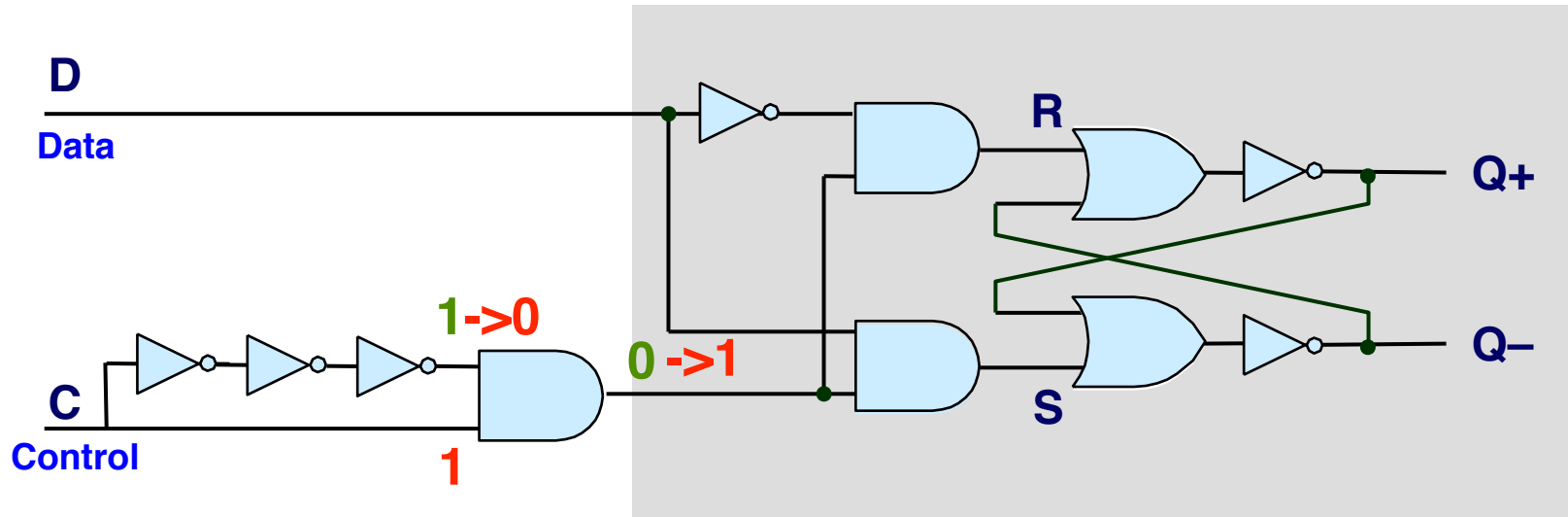
# Edge-Triggered Latch (Flip-Flop)



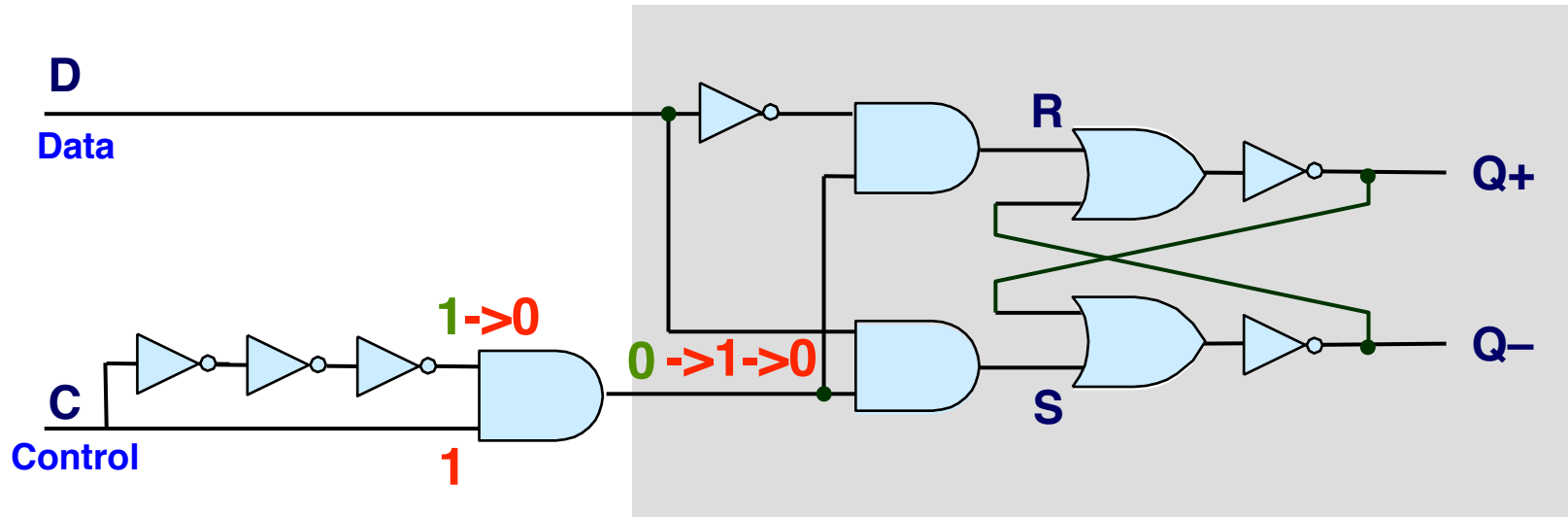
# Edge-Triggered Latch (Flip-Flop)



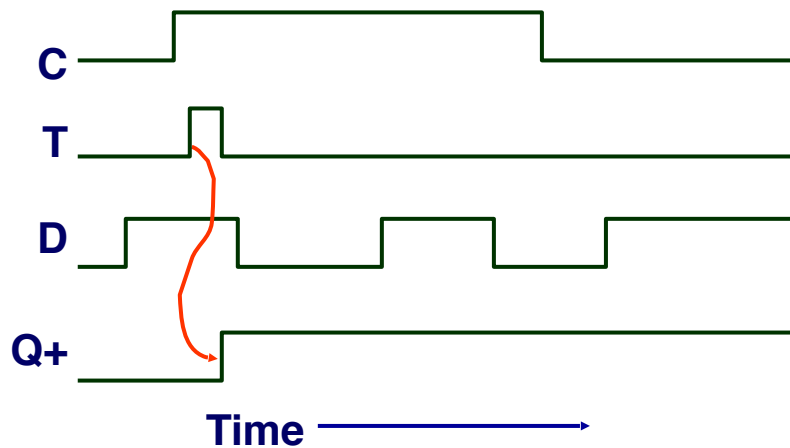
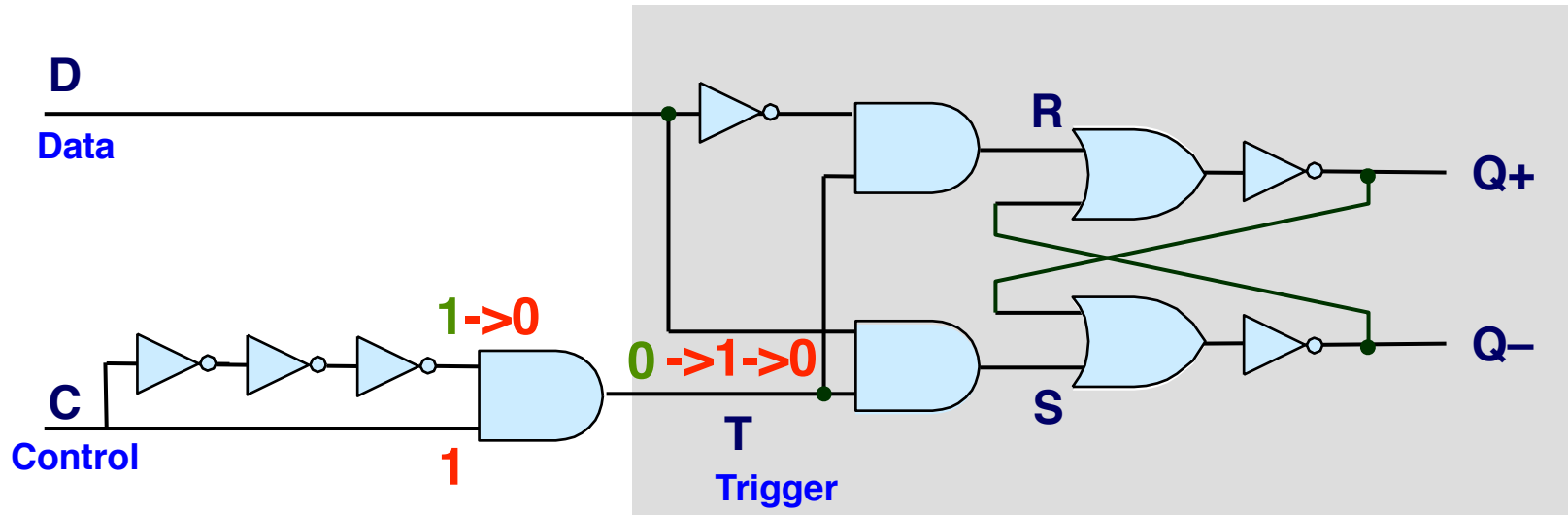
# Edge-Triggered Latch (Flip-Flop)



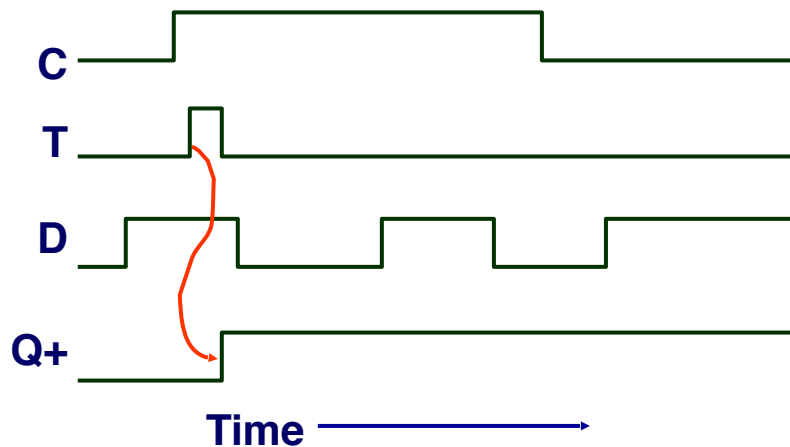
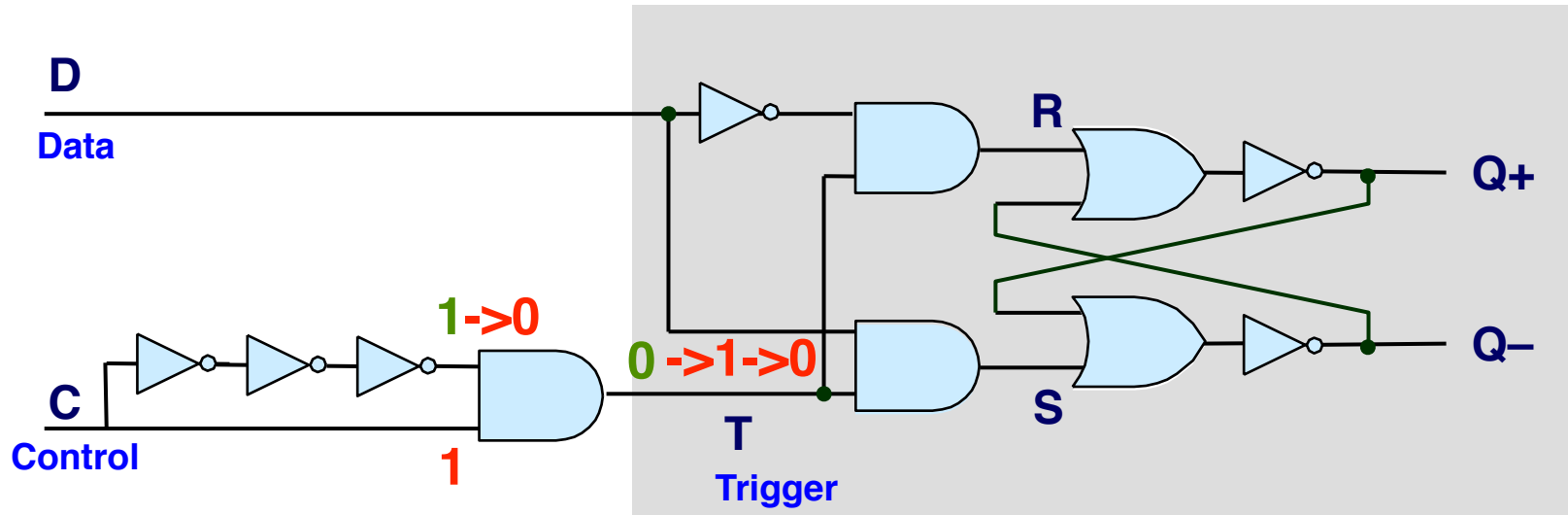
# Edge-Triggered Latch (Flip-Flop)



# Edge-Triggered Latch (Flip-Flop)

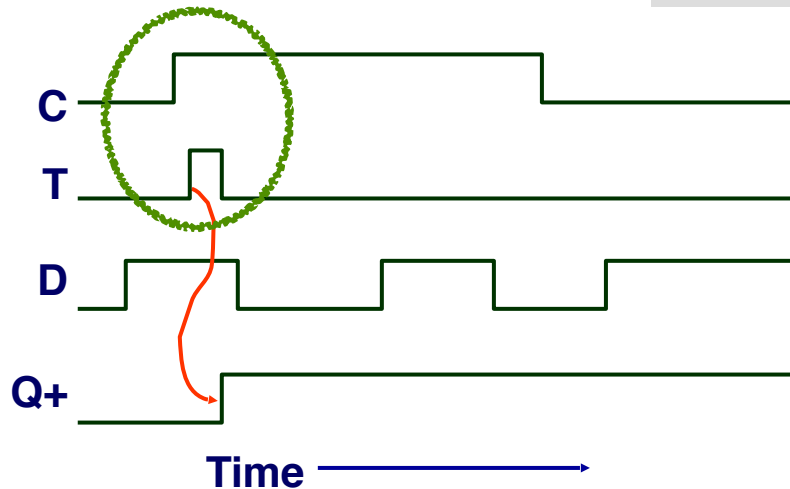
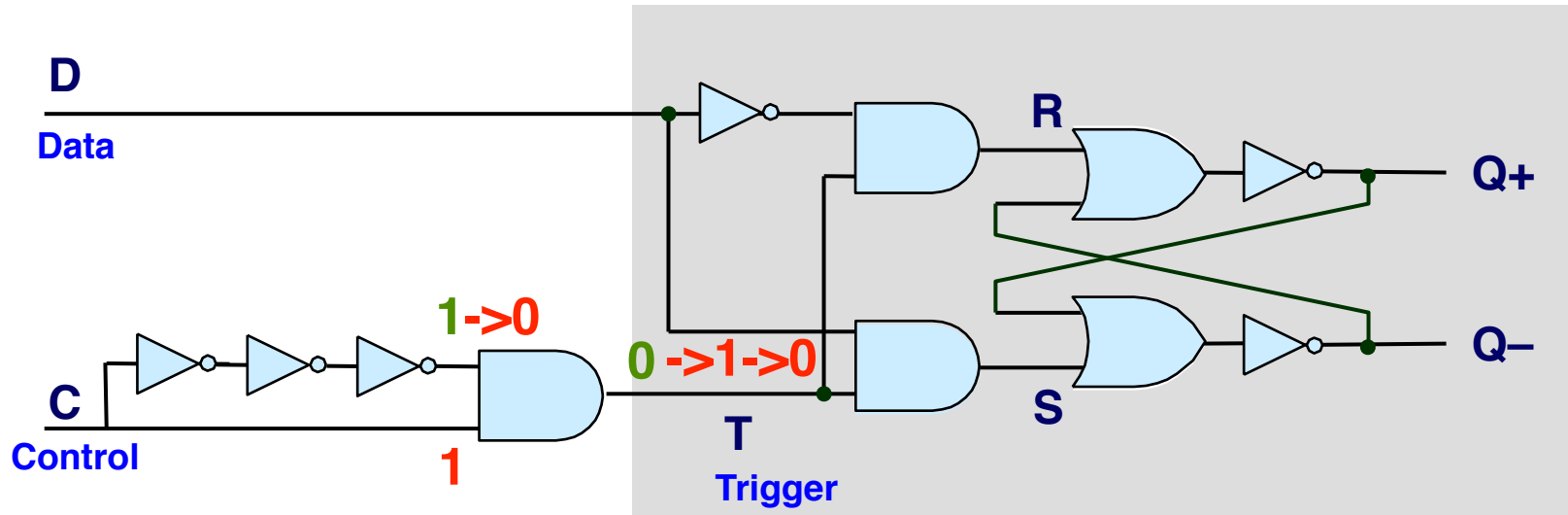


# Edge-Triggered Latch (Flip-Flop)



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

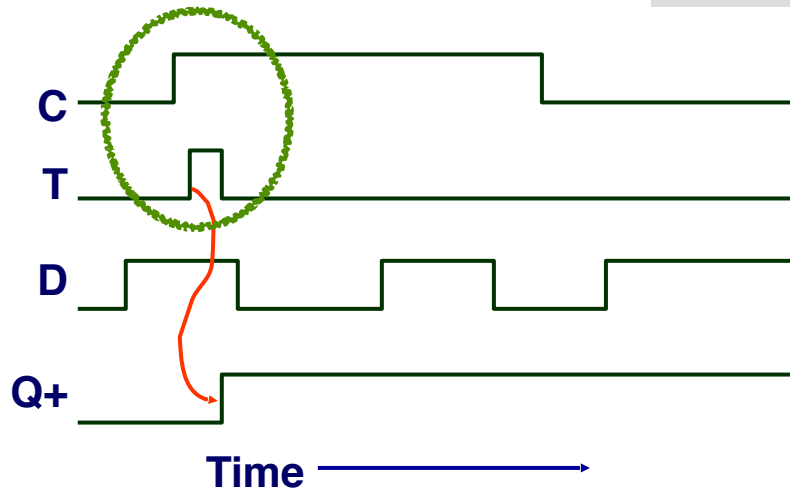
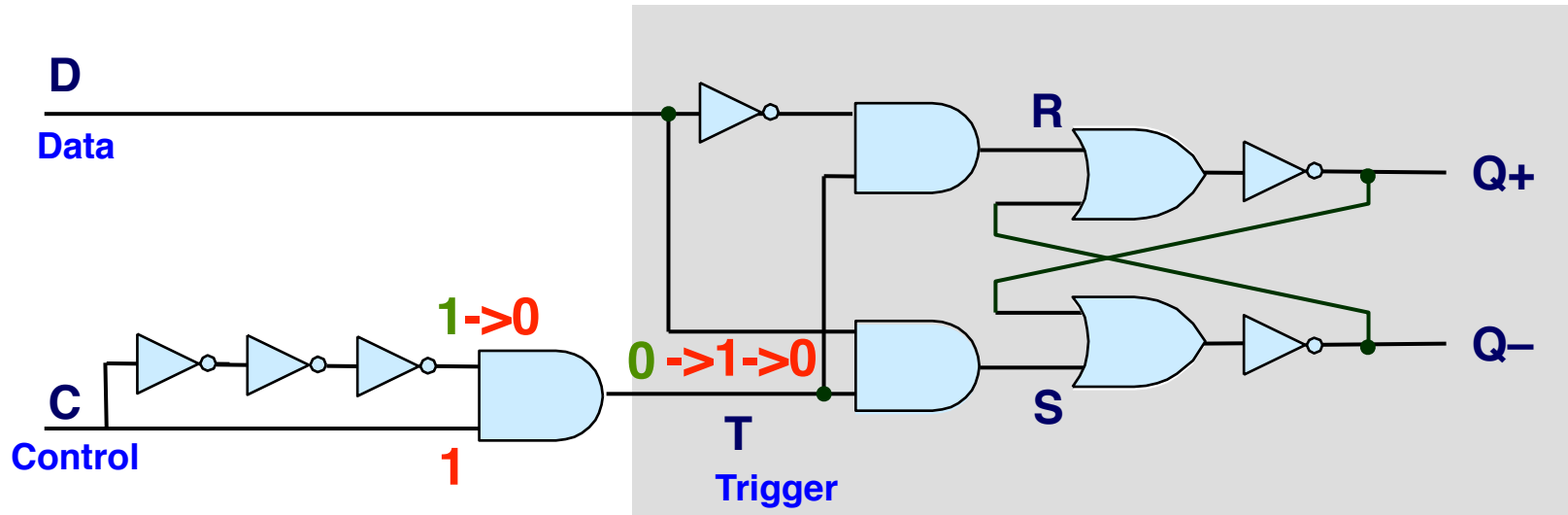
# Edge-Triggered Latch (Flip-Flop)



- 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

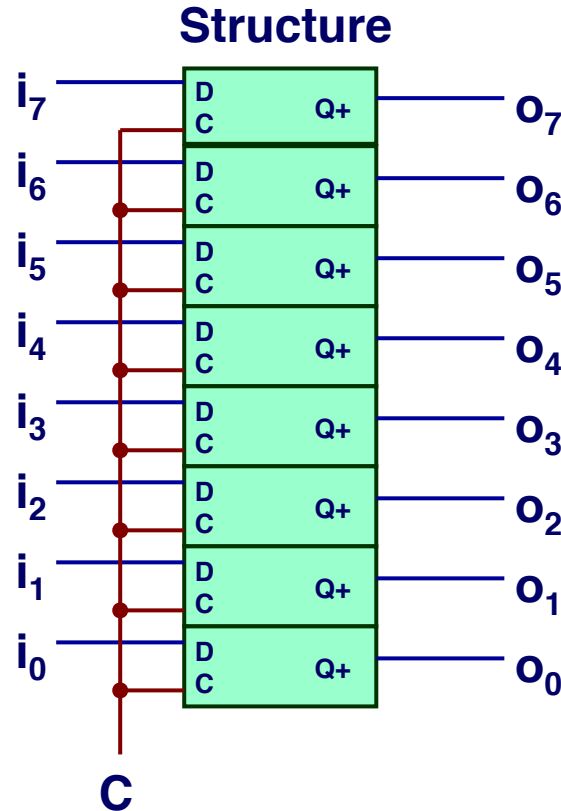


# Edge-Triggered Latch (Flip-Flop)



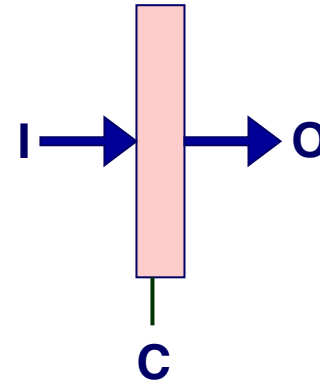
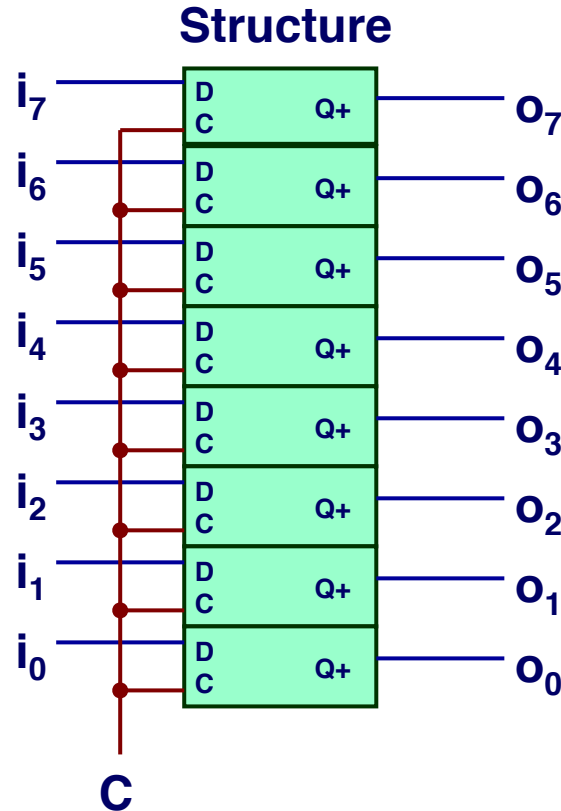
- Flip-flop: Only latches data for a brief period
- Value latched depends on data as **C rises** (i.e., 0→1); usually called at the **rising edge** of **C**
- Output remains stable at all other times

# Registers



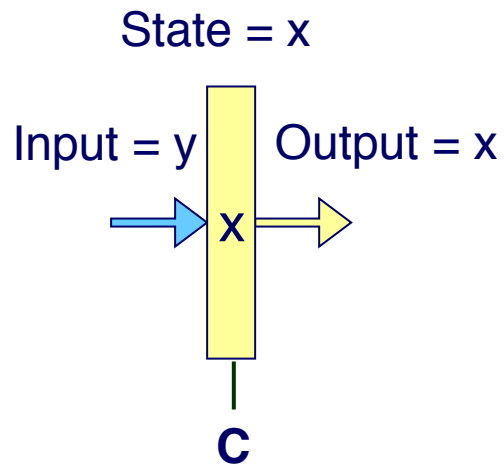
- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

# Registers

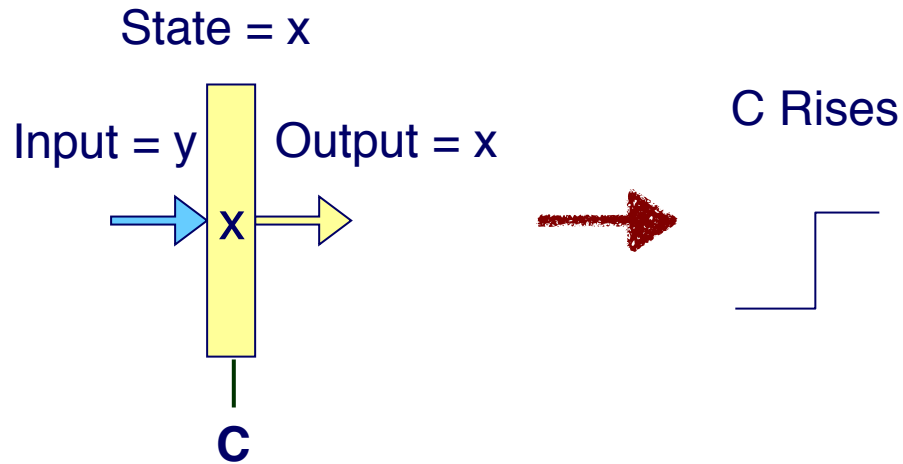


- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the  $C$  signal

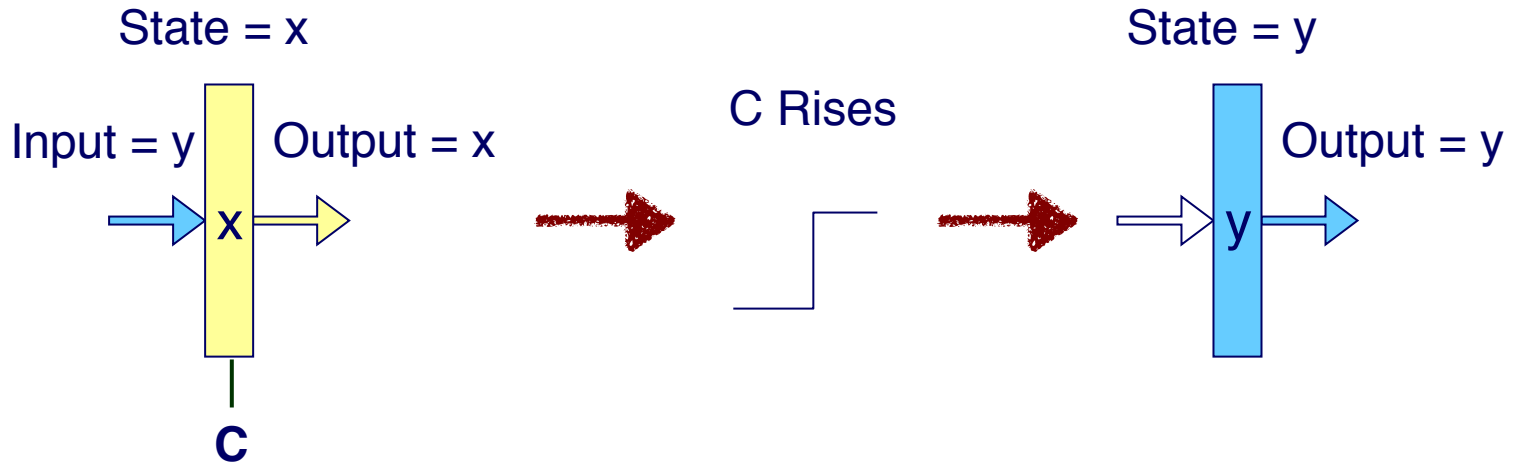
# Register Operation



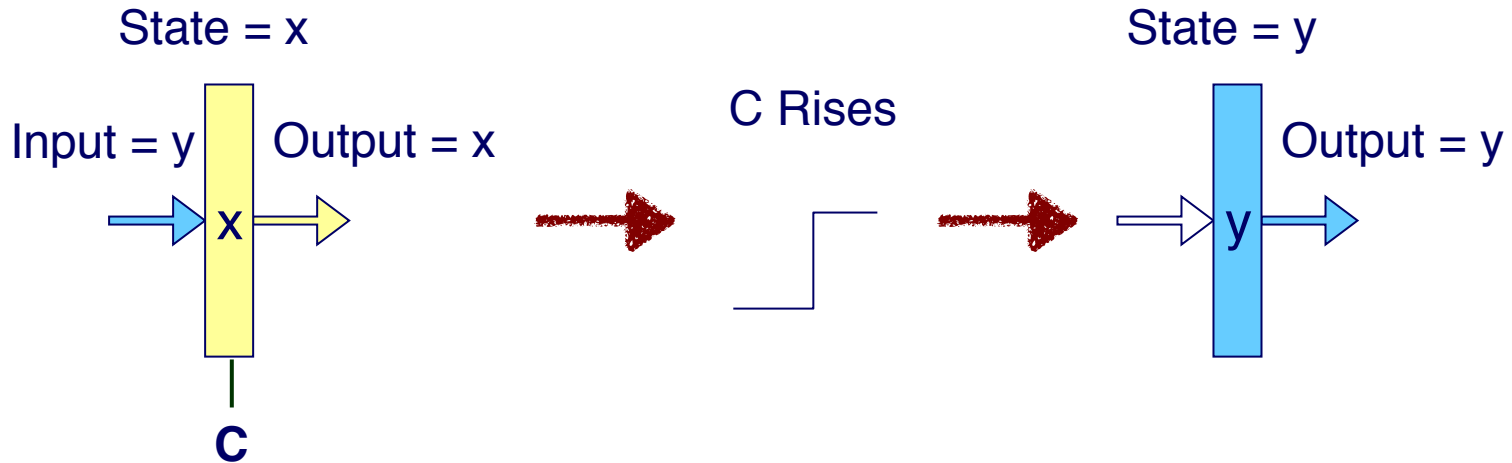
# Register Operation



# Register Operation

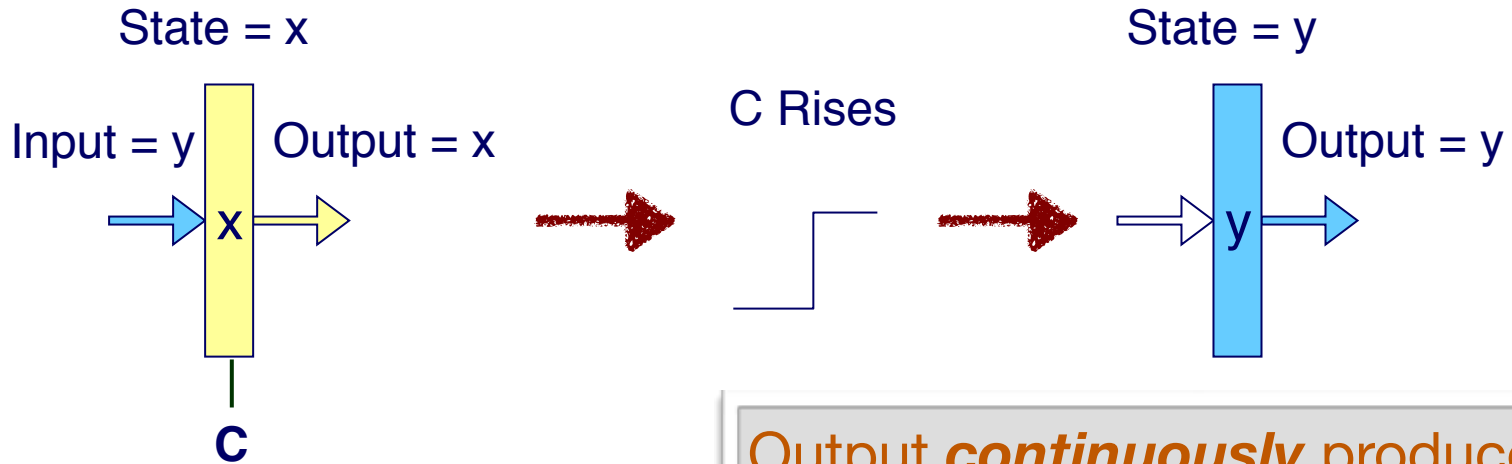


# Register Operation



- 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

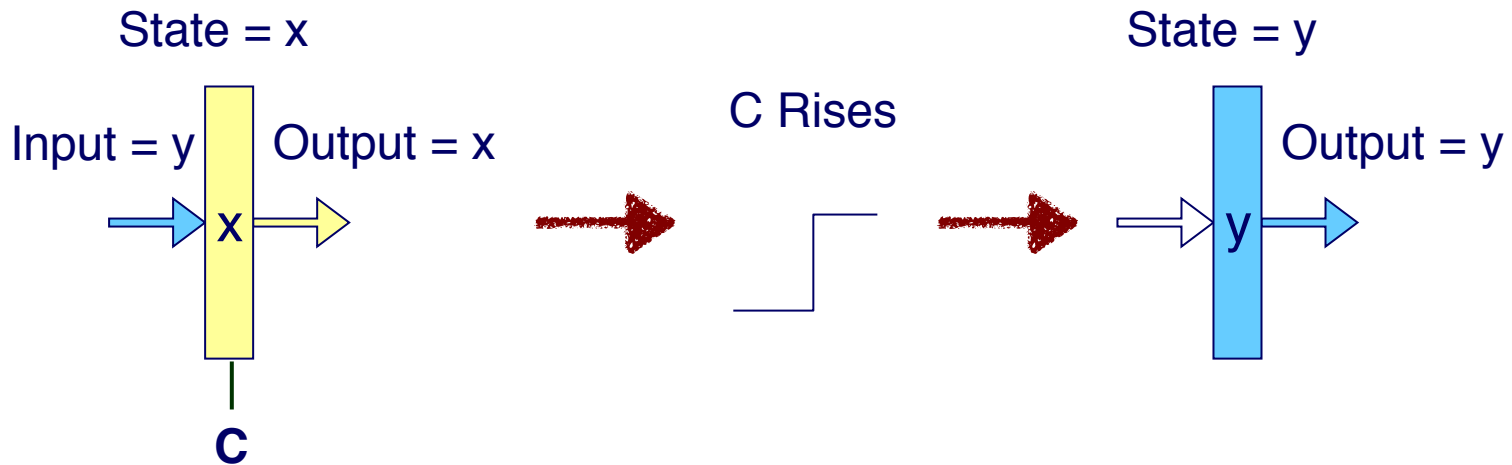


Output **continuously** produces y after the rising edge unless you cut off power.

- 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

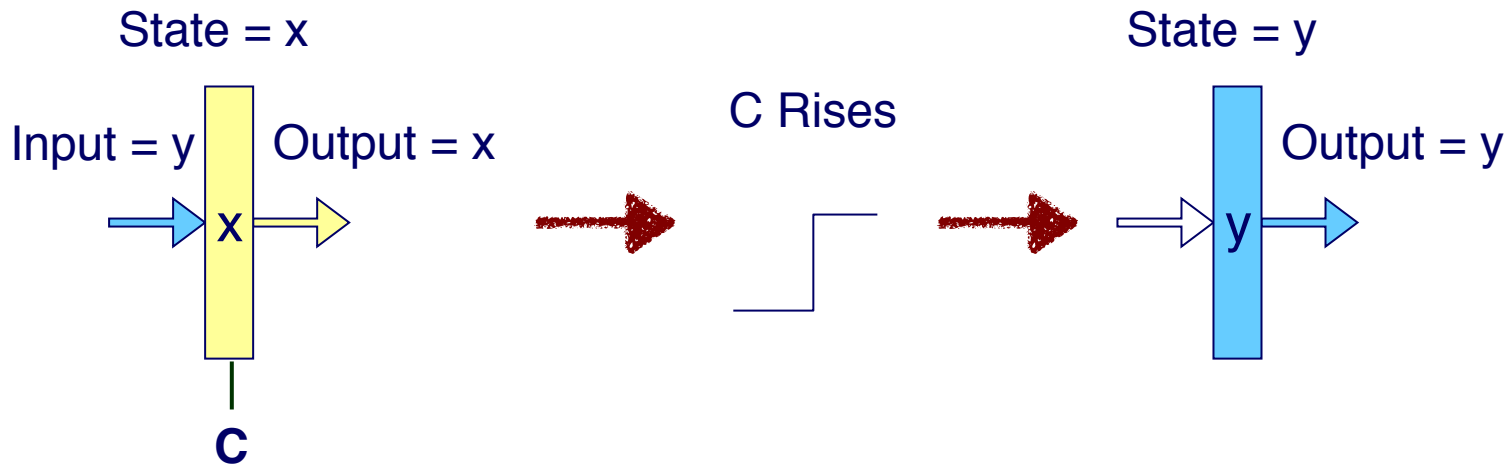


# Clock Signal



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

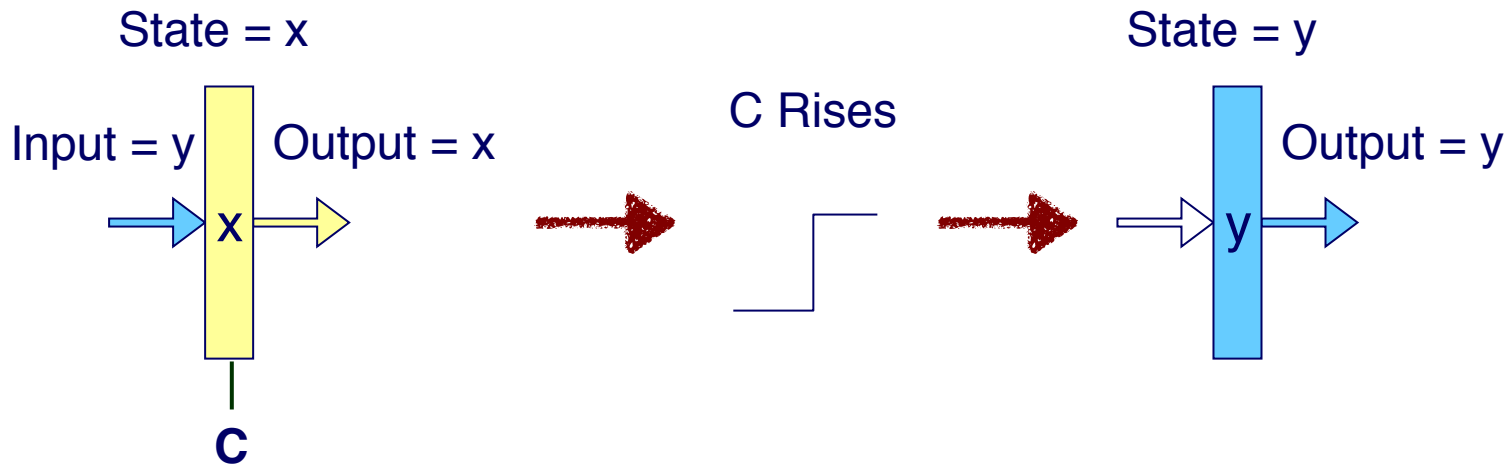
# Clock Signal



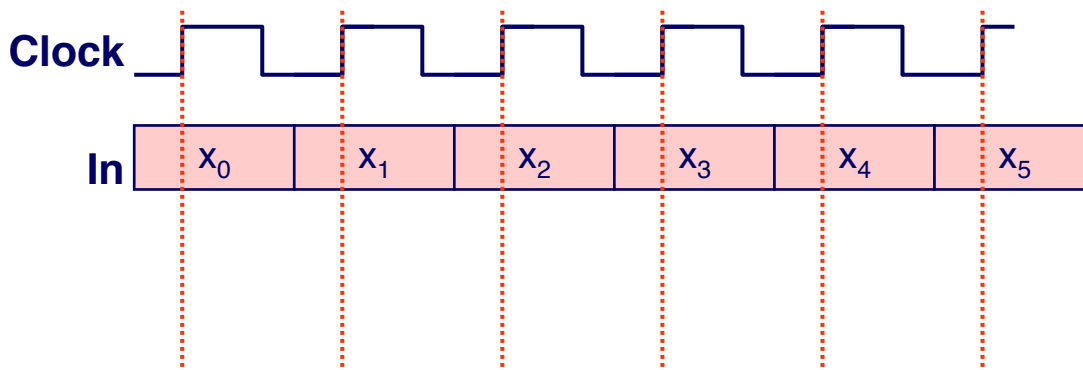
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer



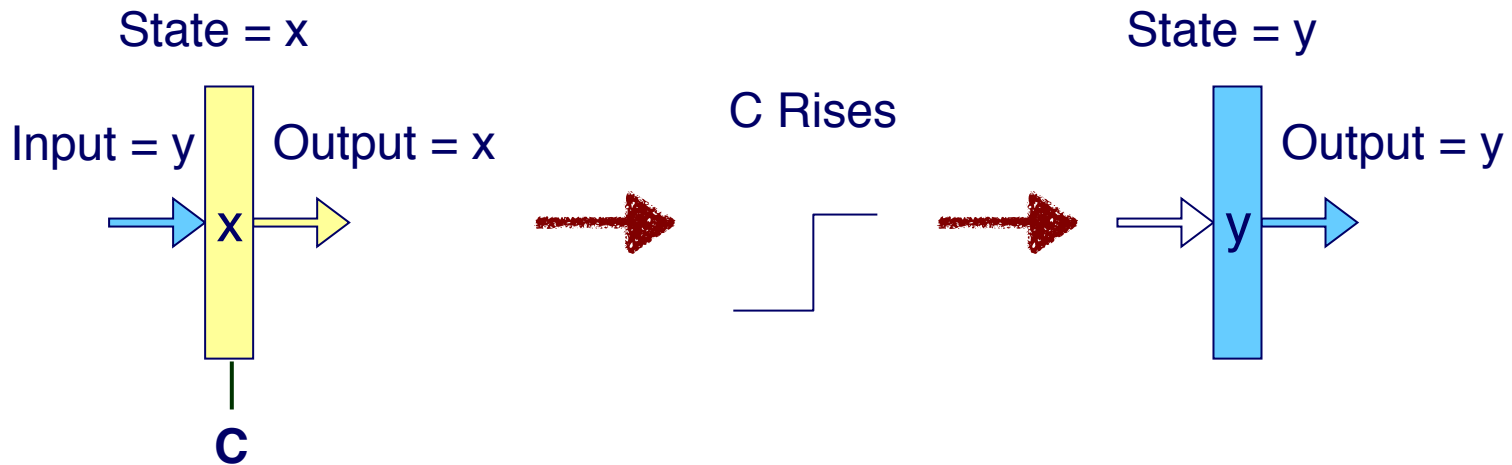
# Clock Signal



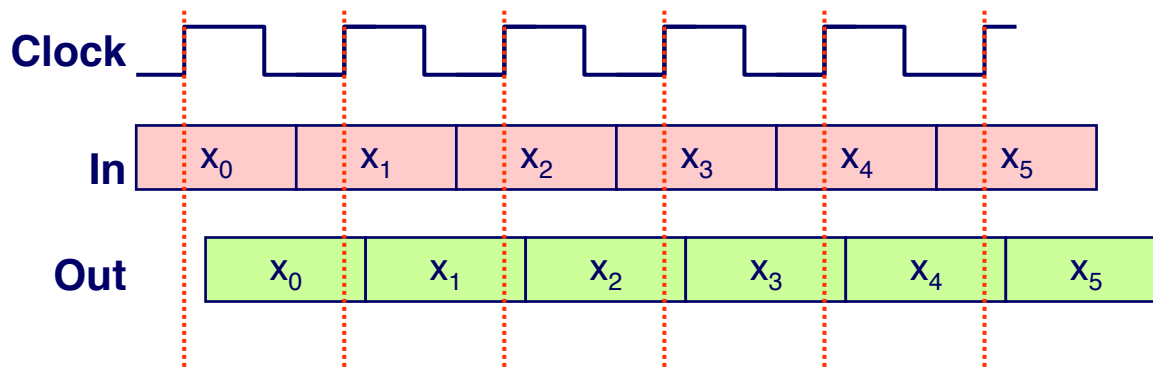
- A special  $C$ : periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer



# Clock Signal

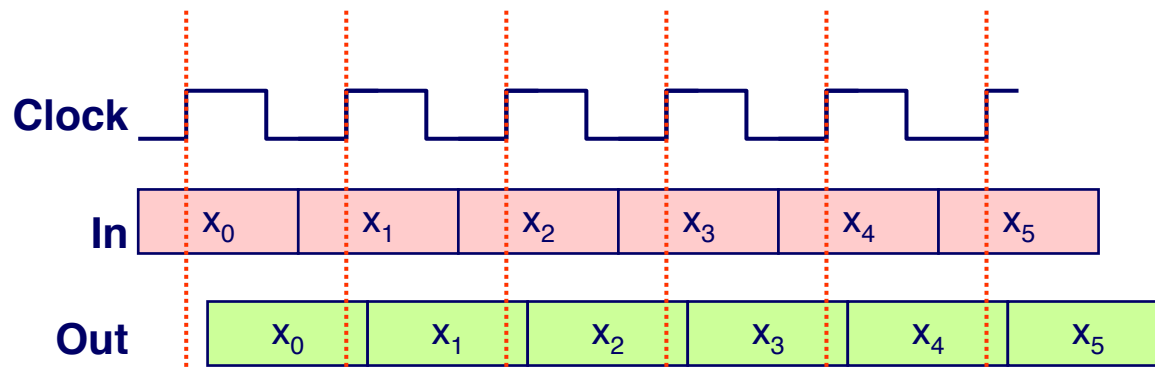


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer



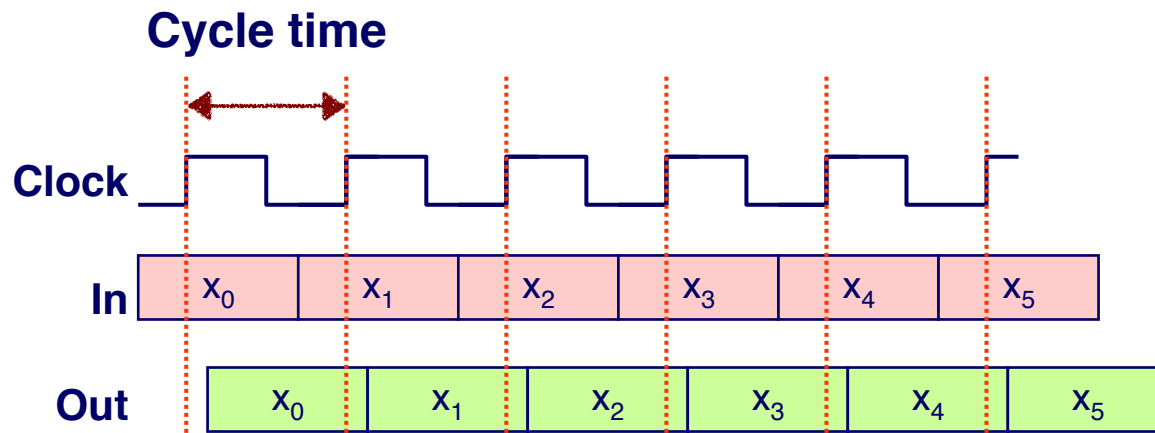
# Clock Signal

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



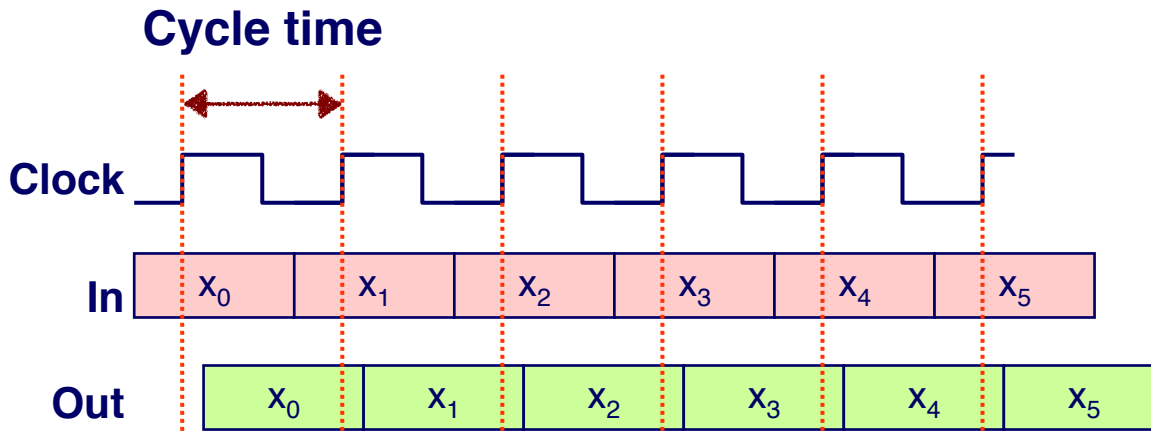
# Clock Signal

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



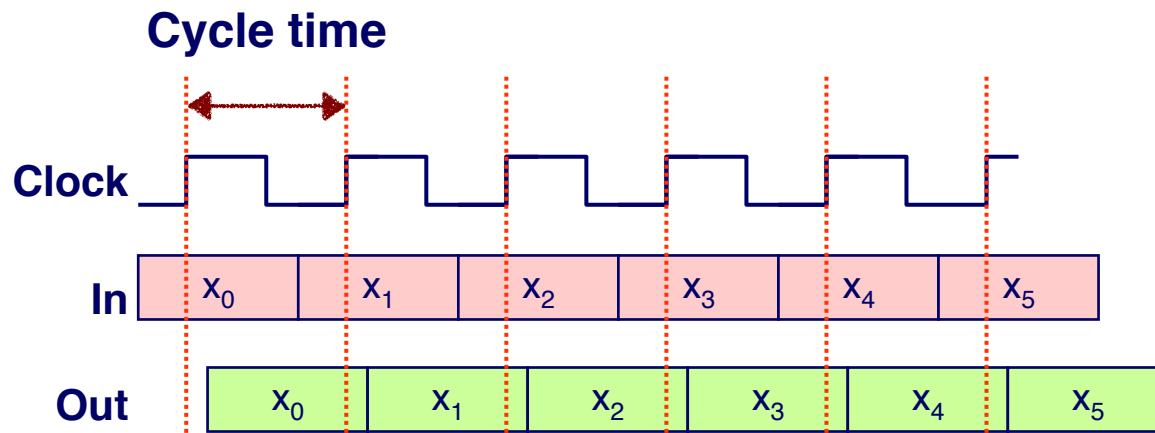
# Clock Signal

- 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  $1/10^9 = 1 \text{ ns}$

