

# **CSC 252: Computer Organization**

## **Spring 2018: Lecture 10**

Instructor: Yuhao Zhu


Department of Computer Science  
University of Rochester

### **Action Items:**

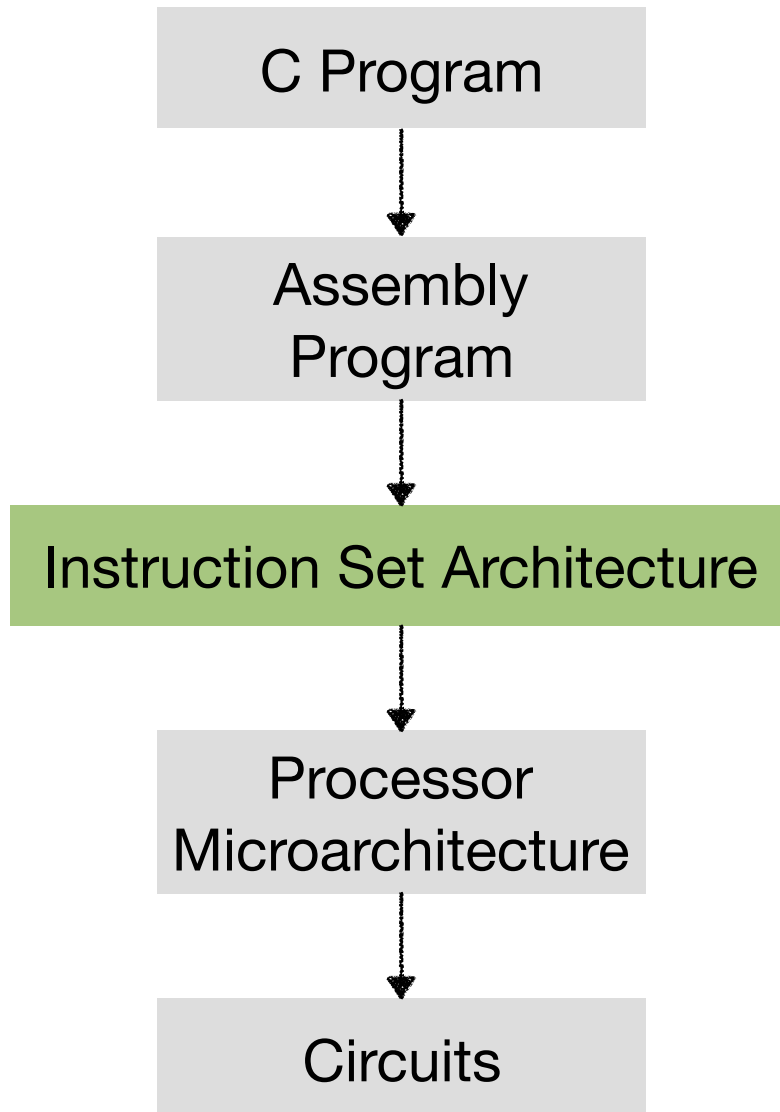
- **Trivia 3 was just due**
- **Assignment 3 is due March 2, midnight**

# Announcement

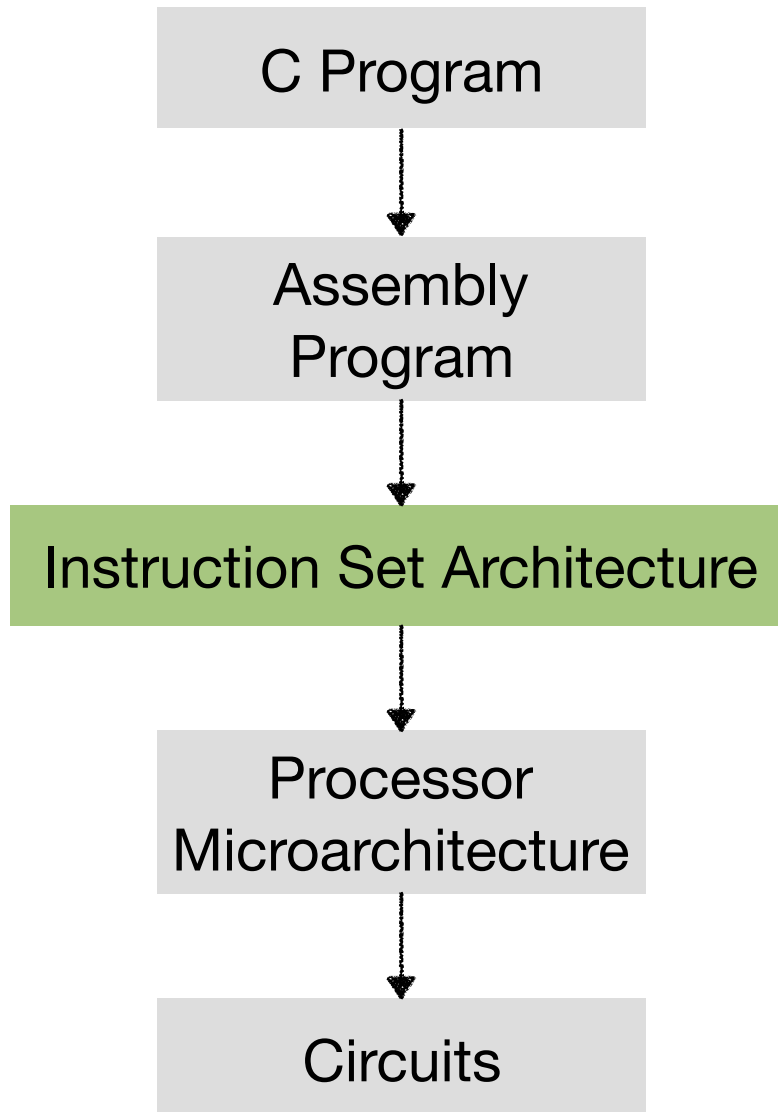
- Programming Assignment 3 is out
  - Due on **March 2, midnight**

18	19	20	21	22	23	24
25	26	27	28	Mar 1	2	3
					due	

# So far in 252...

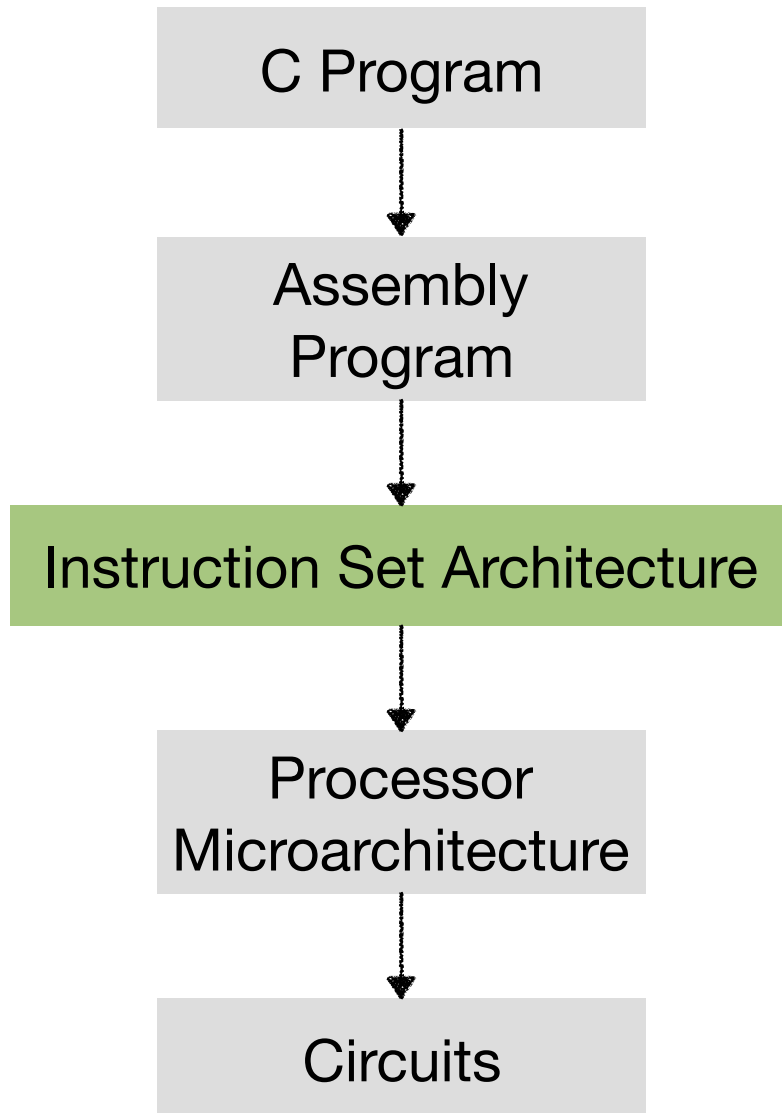


# So far in 252...



```
ret, call  
movq, addq  
jmp, jne
```

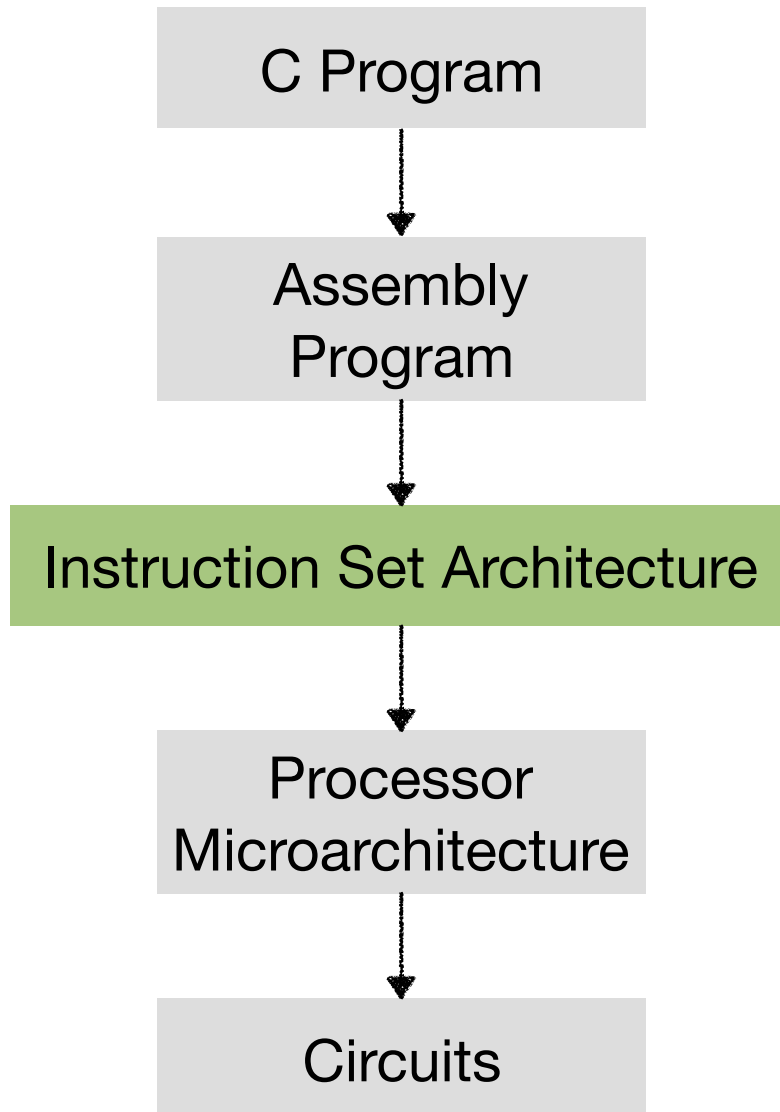
# So far in 252...



```
movq    %rsi, %rax
imulq   %rdx, %rax
jmp     .done
```

```
ret, call
movq, addq
jmp, jne
```

# So far in 252...

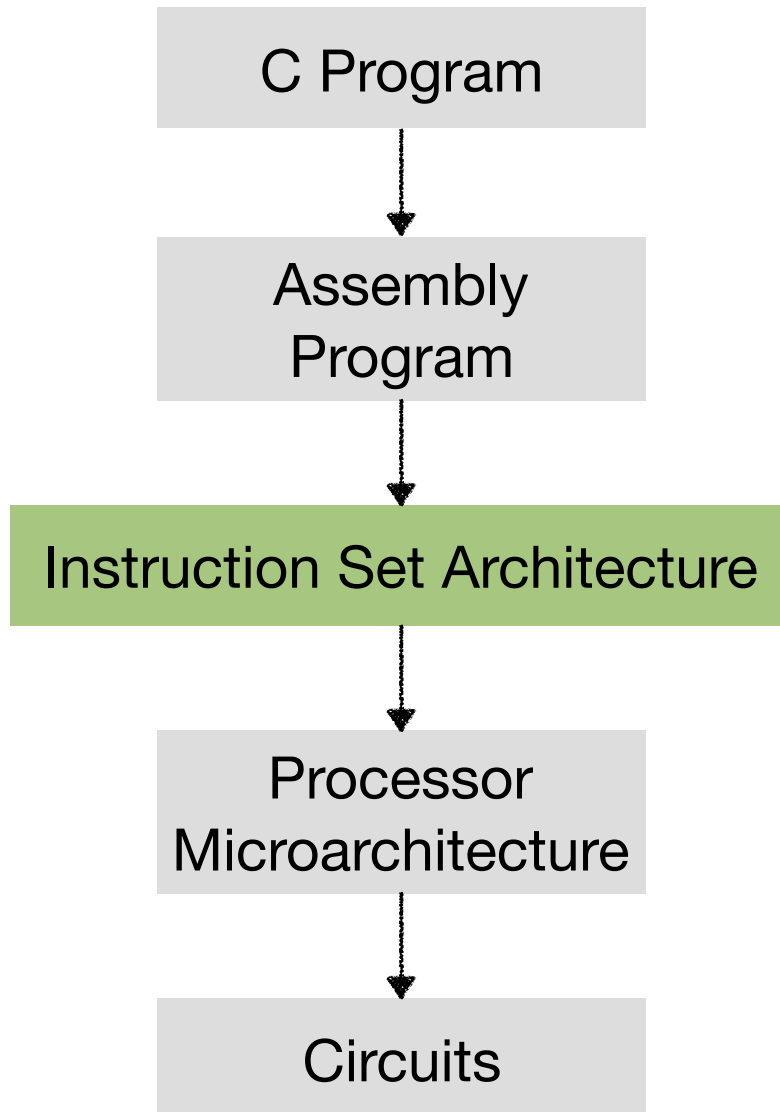


```
int, float  
if, else  
+, -, >>
```

```
movq    %rsi, %rax  
imulq   %rdx, %rax  
jmp     .done
```

```
ret, call  
movq, addq  
jmp, jne
```

# So far in 252...



int, float  
if, else  
+, -, >>

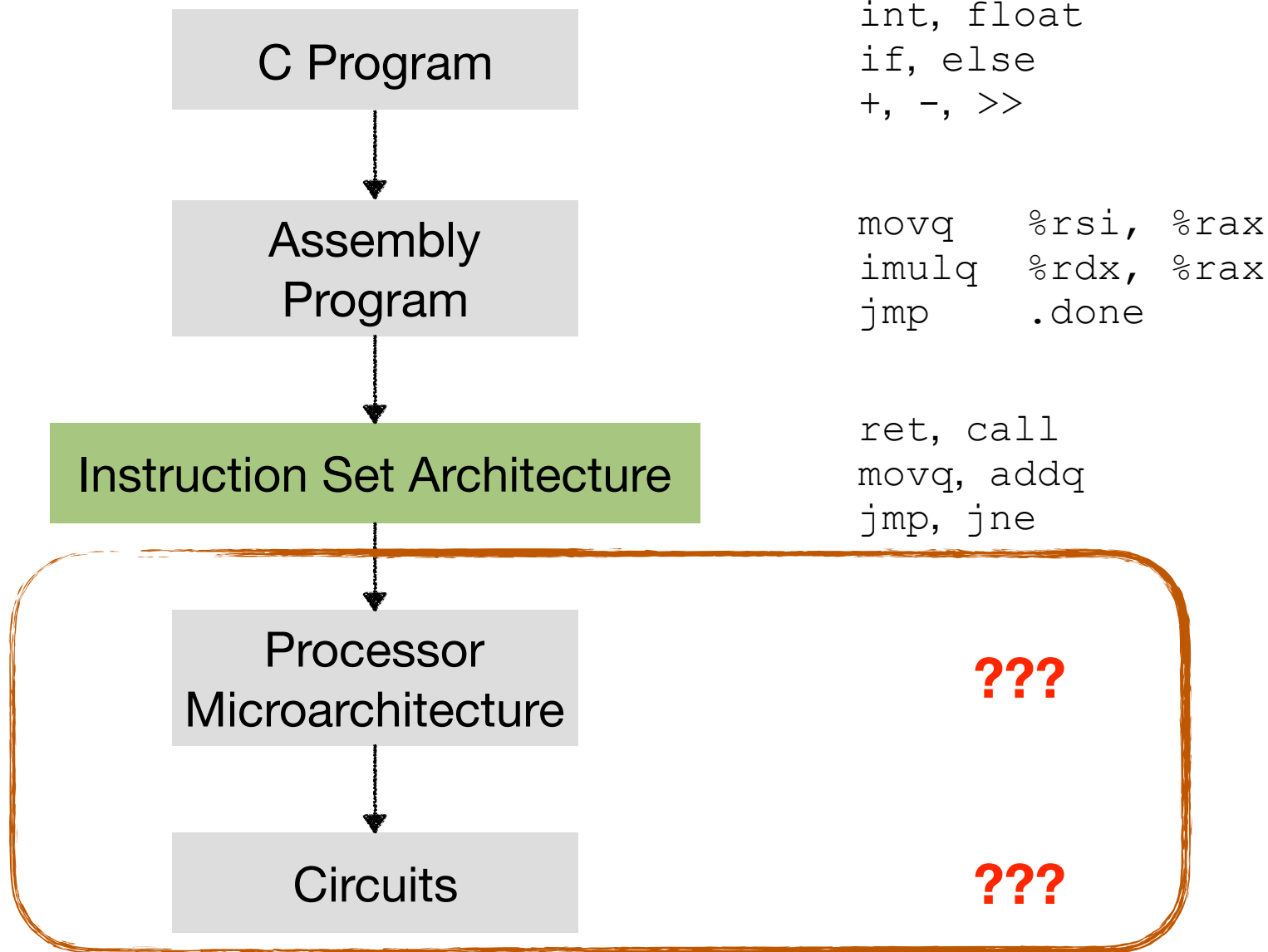
```
movq    %rsi, %rax  
imulq   %rdx, %rax  
jmp     .done
```

```
ret, call  
movq, addq  
jmp, jne
```

???

???

# So far in 252...





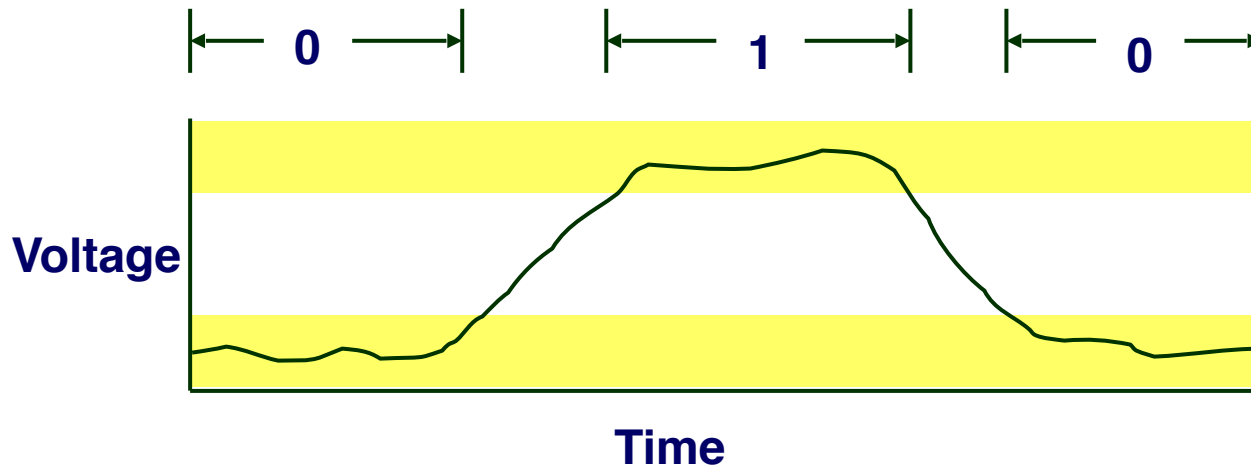
# Today: Circuits Basics

- Transistors
- 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.
- Bits are Our Friends: Everything expressed in 0s and 1s
  - Communication: Low or high voltage on wire
  - Computation: Compute Boolean functions
  - Storage: Store bits of information
- Circuit design is often abstracted as **logic design**

# Digital Signals



- Extract discrete values from continuous voltage signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

# Basic Building Block: Transistors

# Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

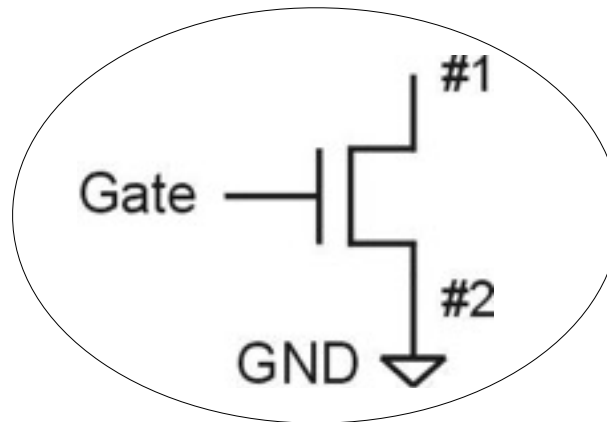
- two types: n-type and p-type

# Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)



Terminal #2 must be connected to GND (0V).

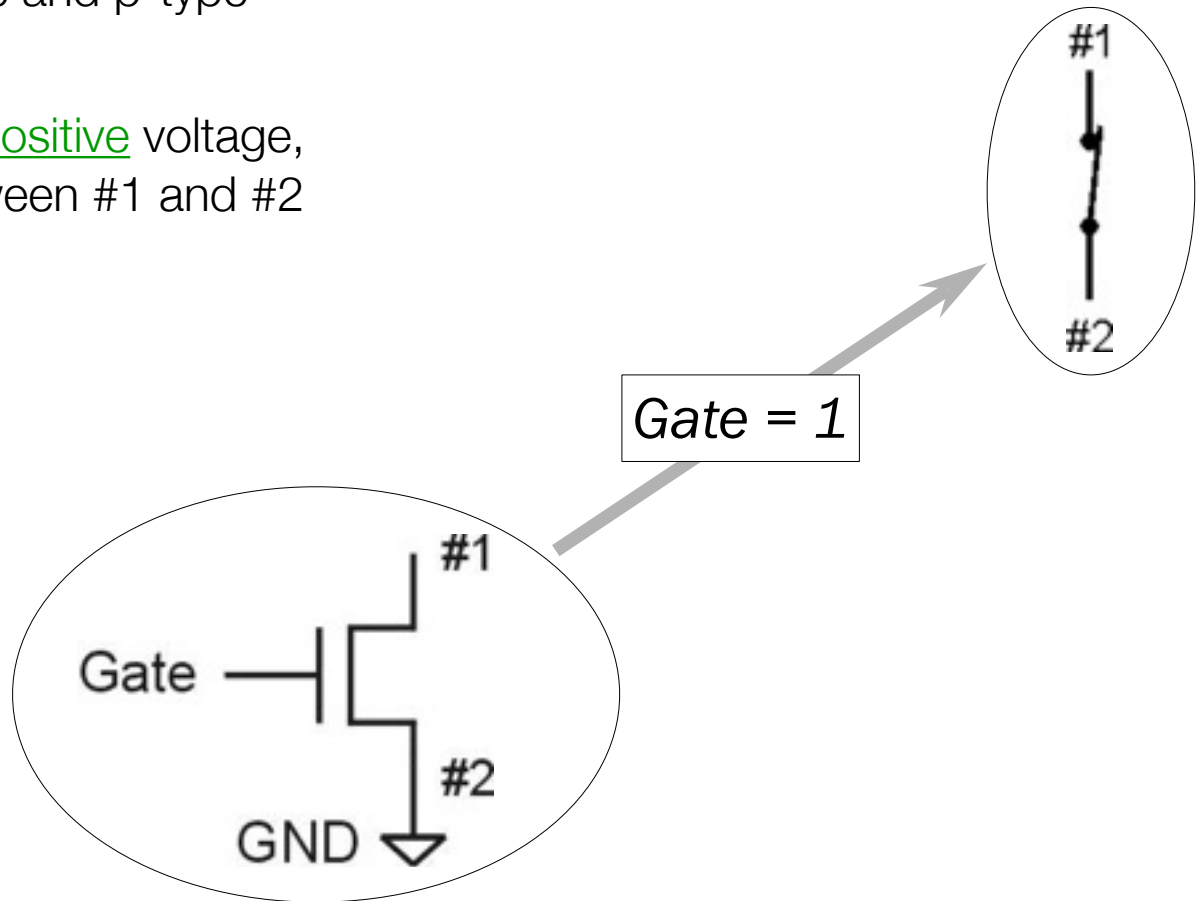
# Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

## n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)



Terminal #2 must be connected to GND (0V).

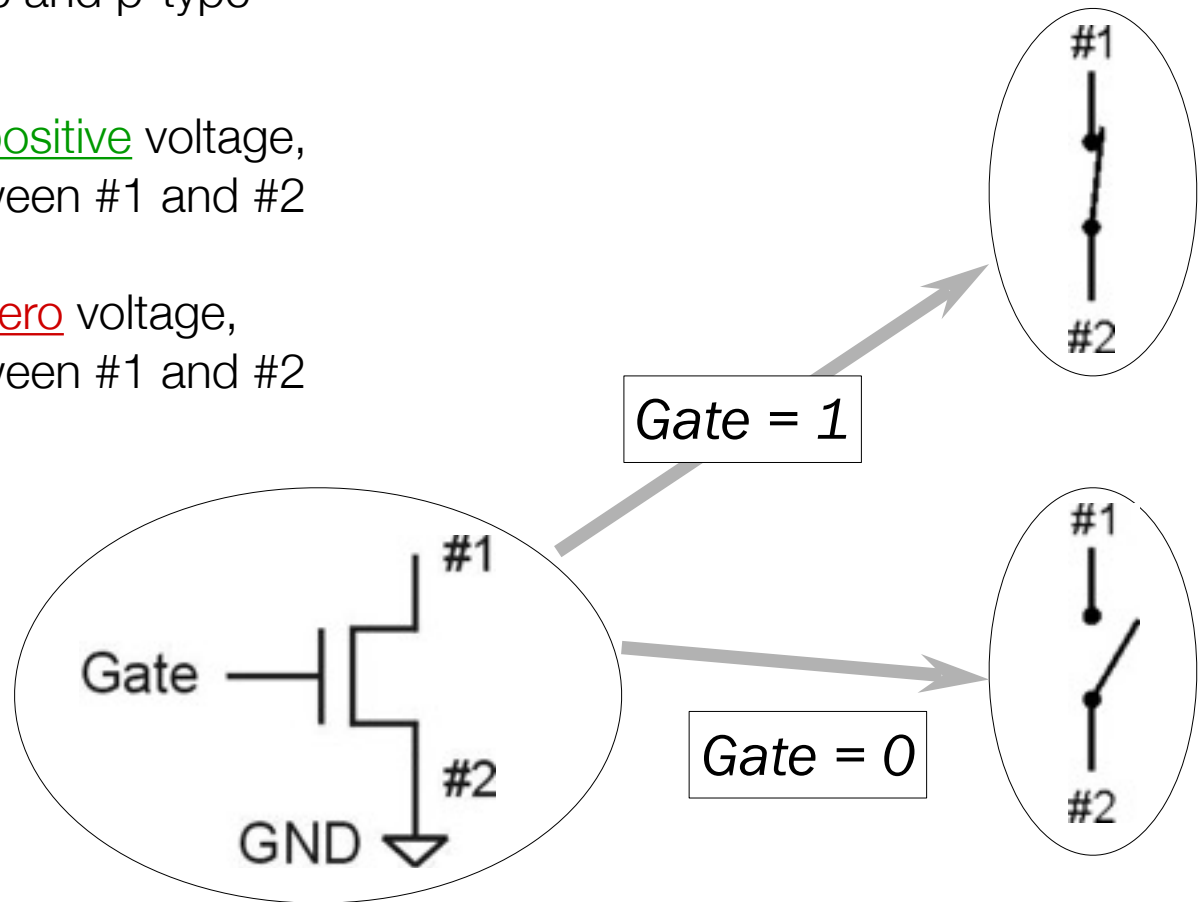
# Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

## n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)



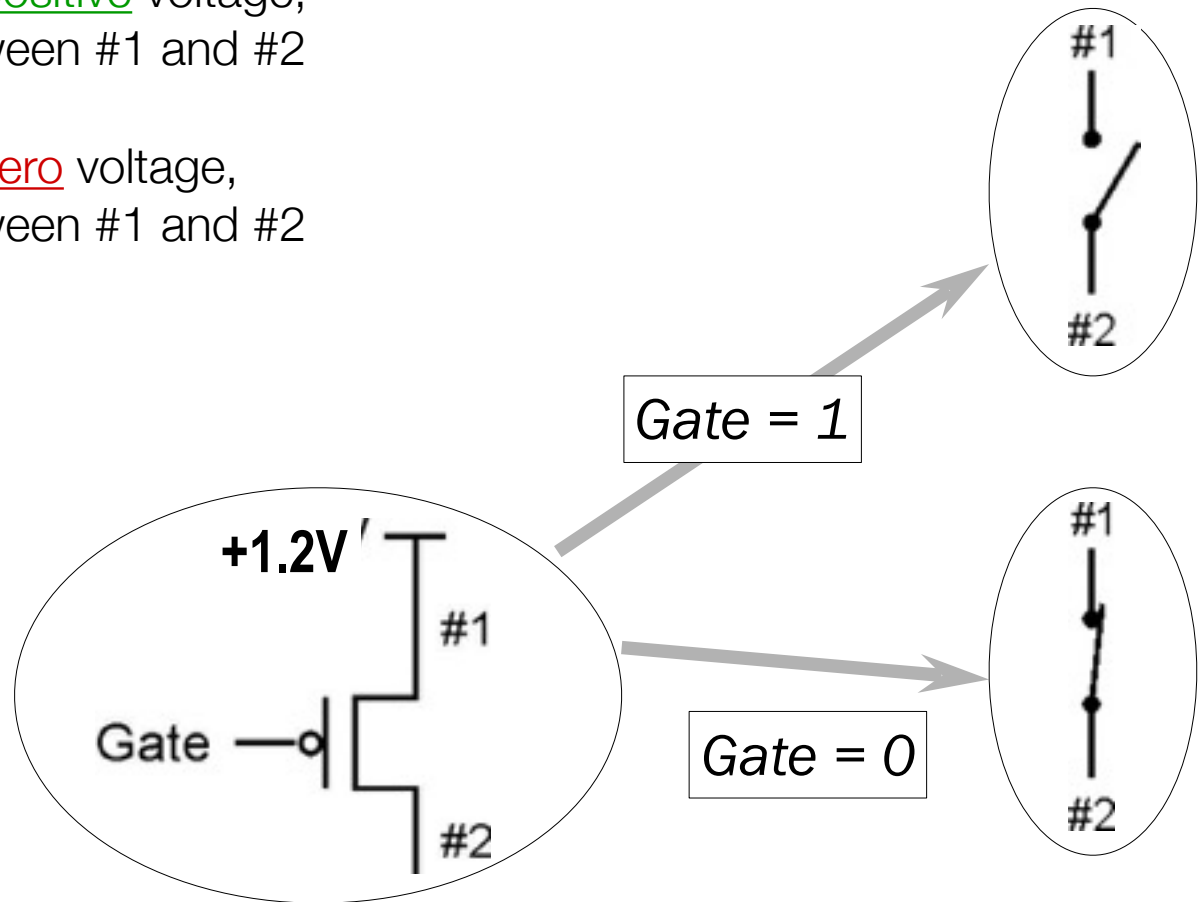
Terminal #2 must be connected to GND (0V).



# Basic Building Block: Transistors

**p-type** is *complementary* to n-type (**PMOS**)

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

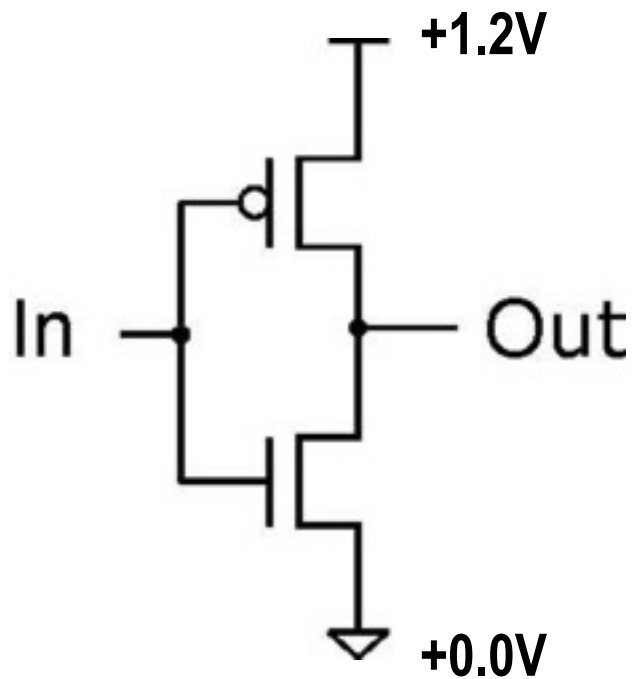


Terminal #1 must be connected to +1.2V

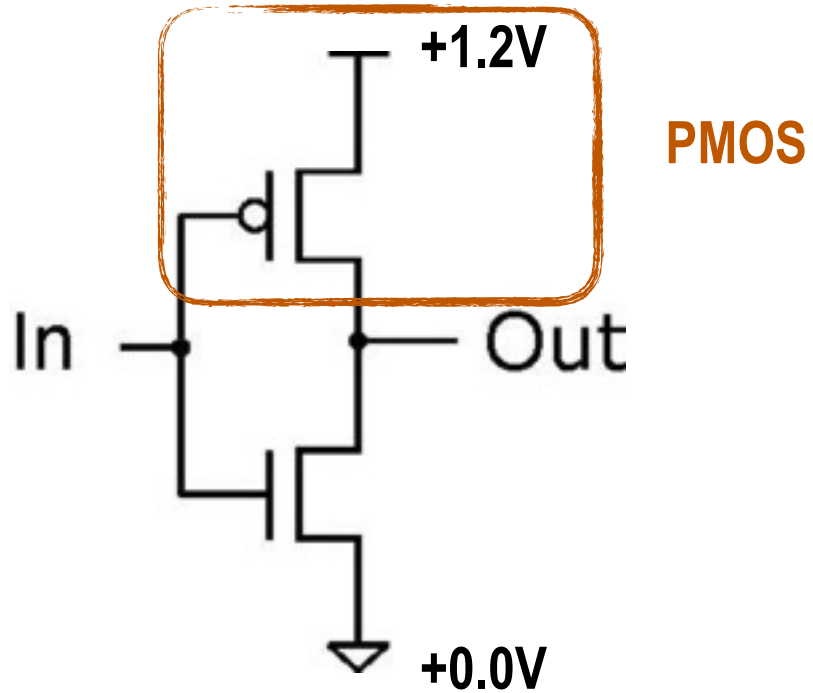
# CMOS Circuit

- Complementary MOS
- Uses both n-type and p-type MOS transistors
  - p-type
    - Attached to + voltage
    - Pulls output voltage UP when input is zero
  - n-type
    - Attached to GND
    - Pulls output voltage DOWN when input is one

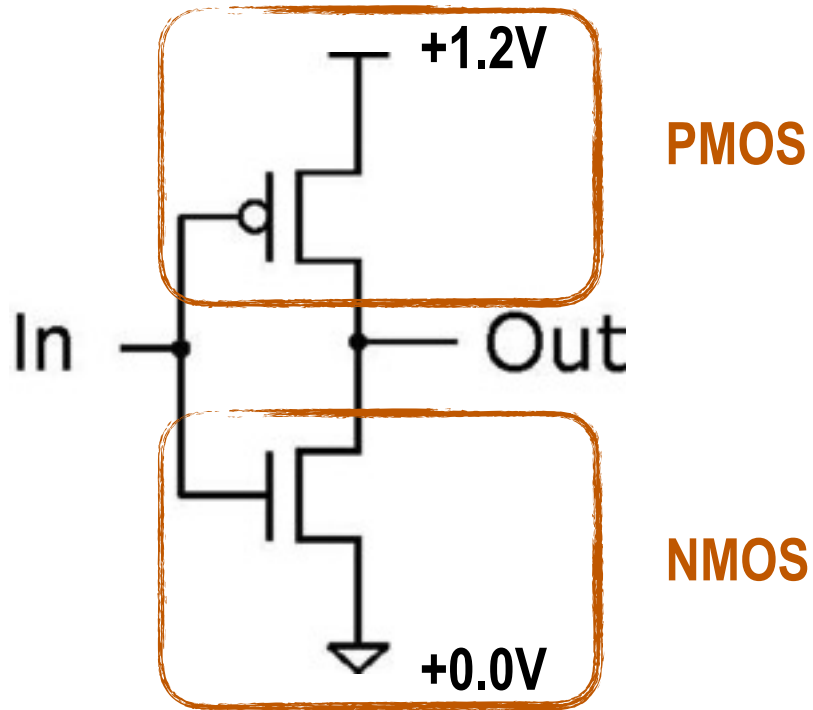
# Inverter (NOT Gate)



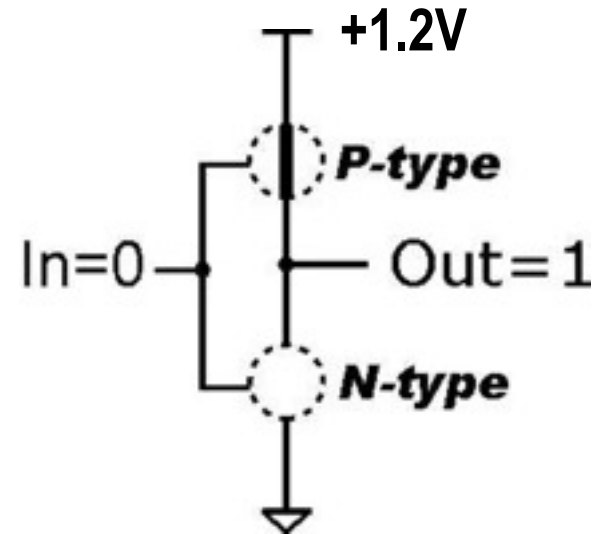
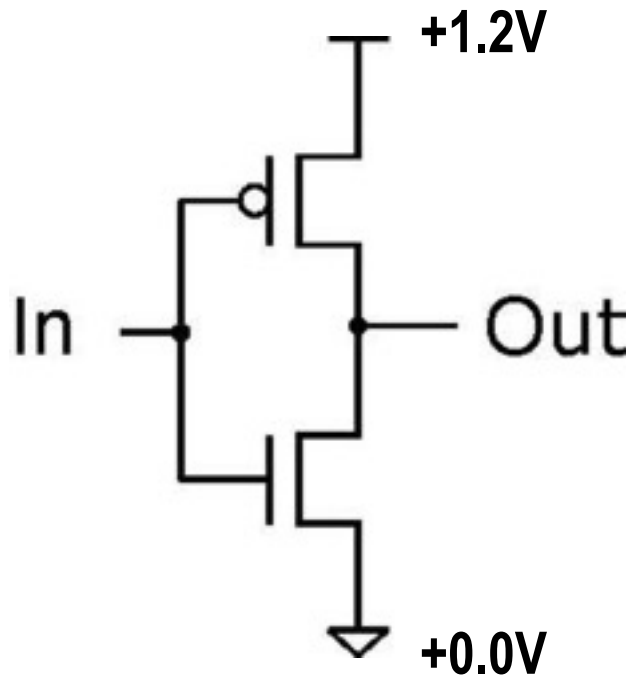
# Inverter (NOT Gate)



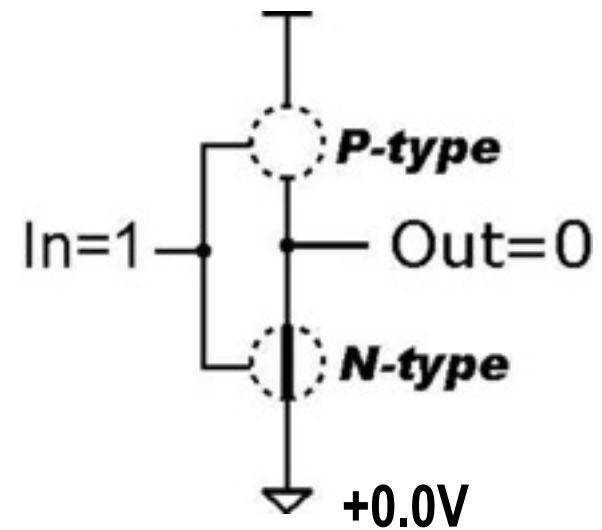
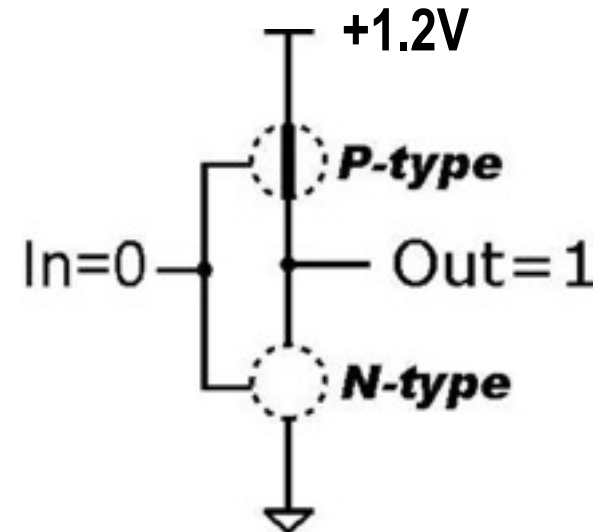
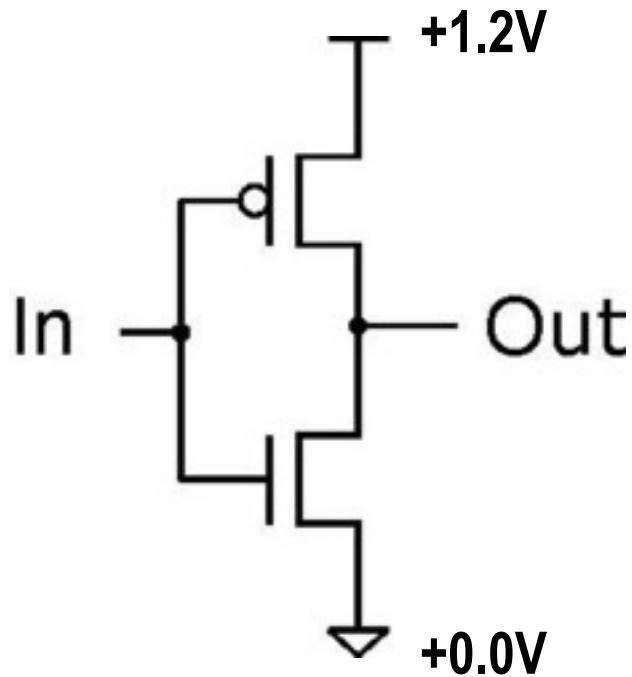
# Inverter (NOT Gate)



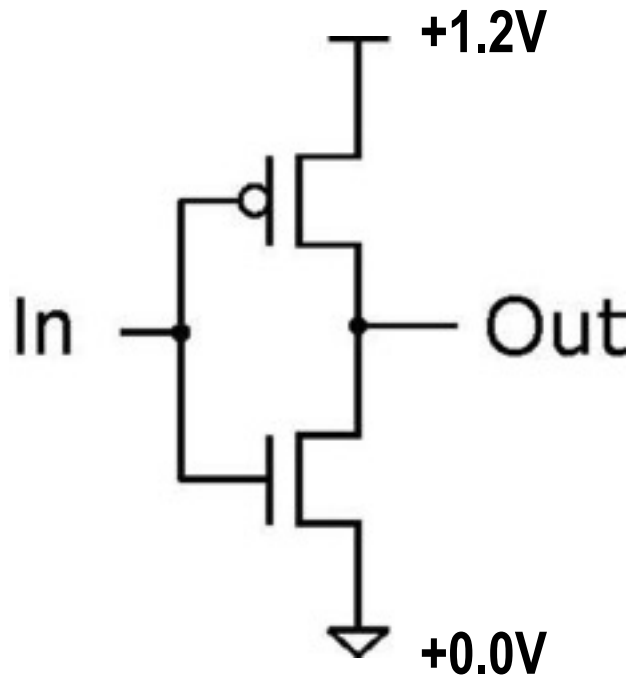
# Inverter (NOT Gate)



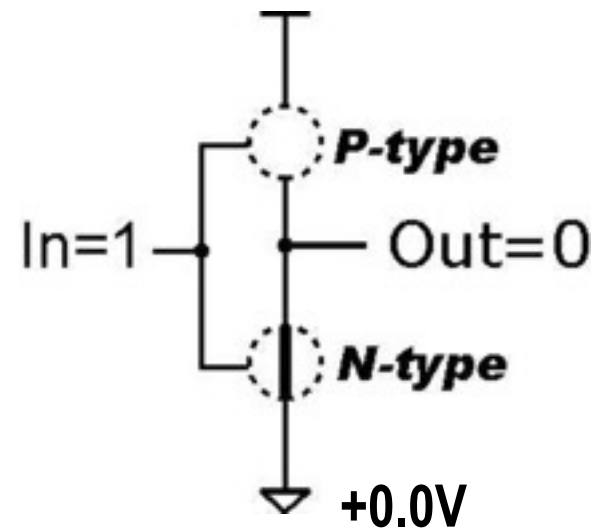
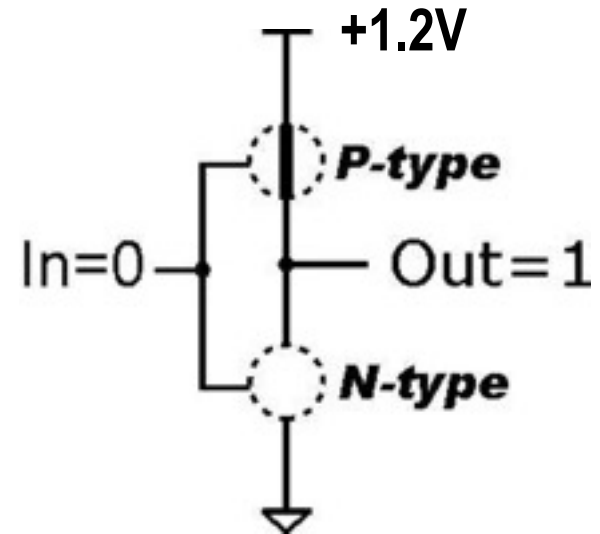
# Inverter (NOT Gate)



# Inverter (NOT Gate)

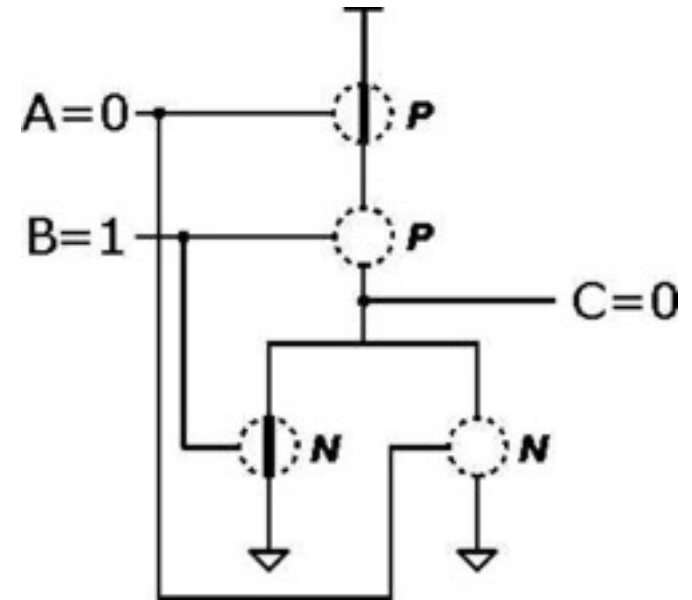
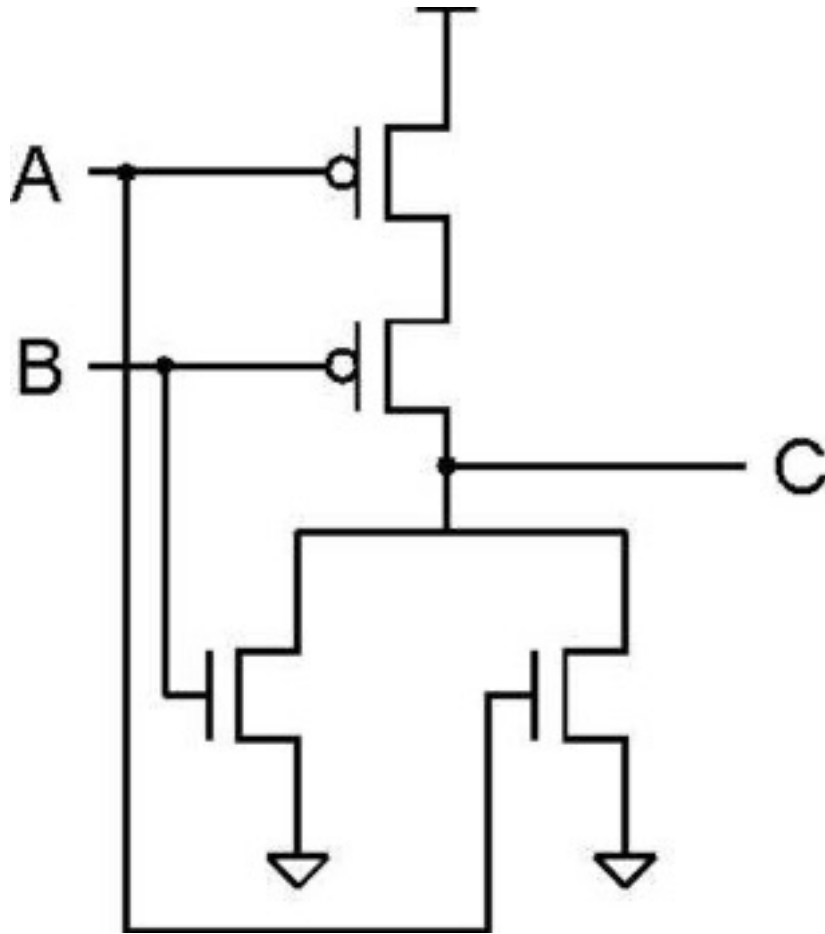


In	Out
0	1
1	0





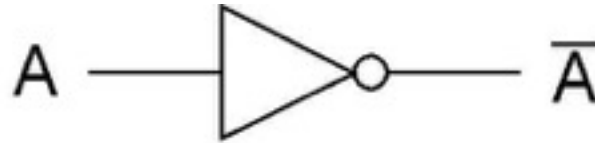
# NOR Gate (NOT + OR)



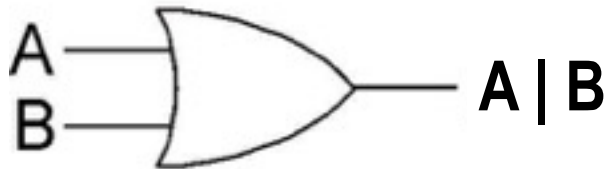
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

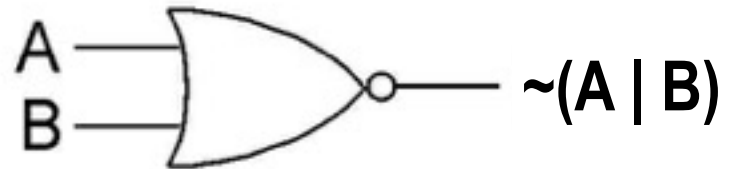
# Basic Logic Gates



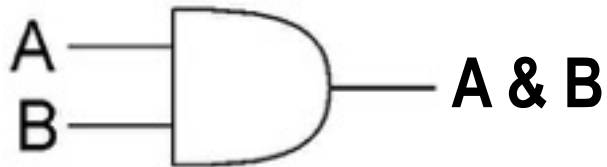
*NOT*



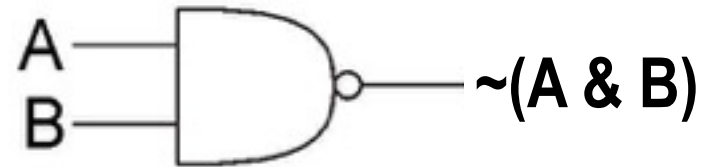
*OR*



*NOR*



*AND*



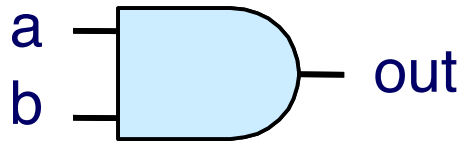
*NAND*

# Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

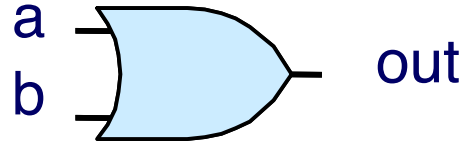
# Computing with Logic Gates

And



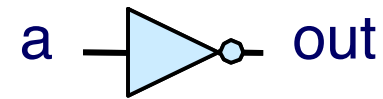
$$\text{out} = a \ \&\& \ b$$

Or



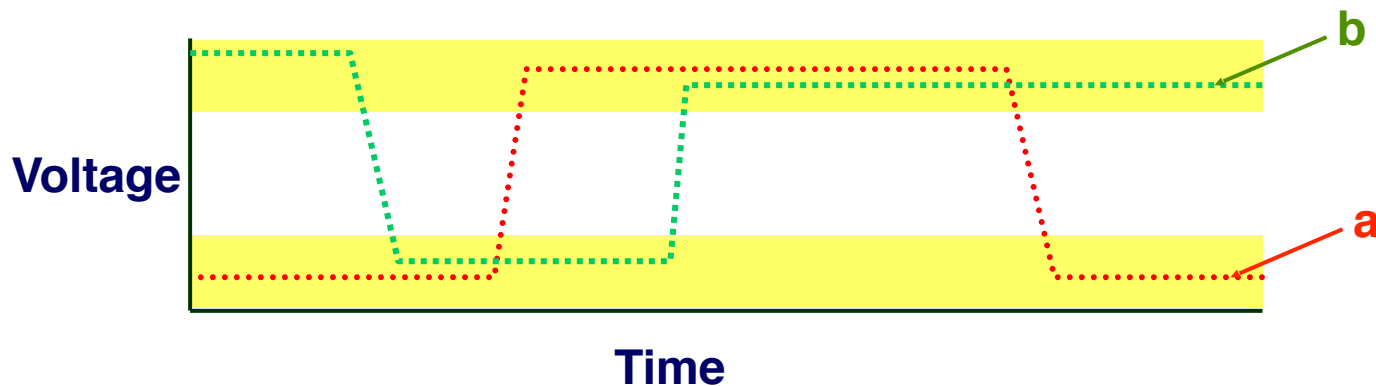
$$\text{out} = a \ || \ b$$

Not

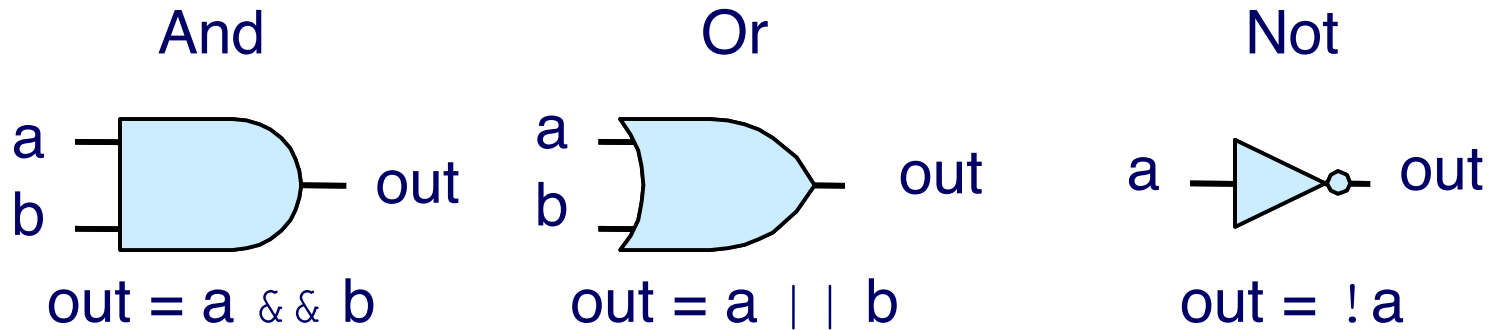


$$\text{out} = !a$$

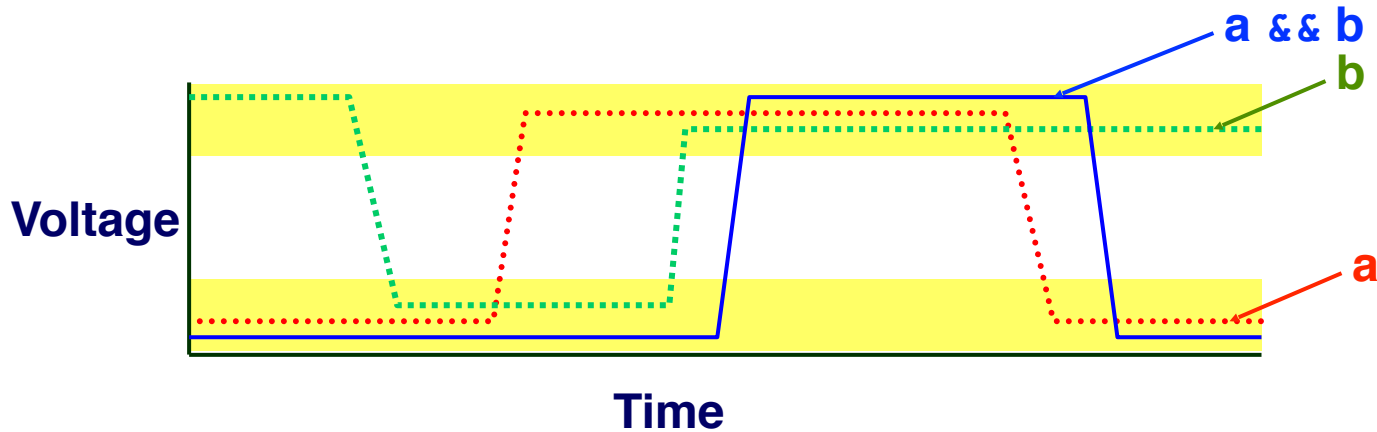
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay



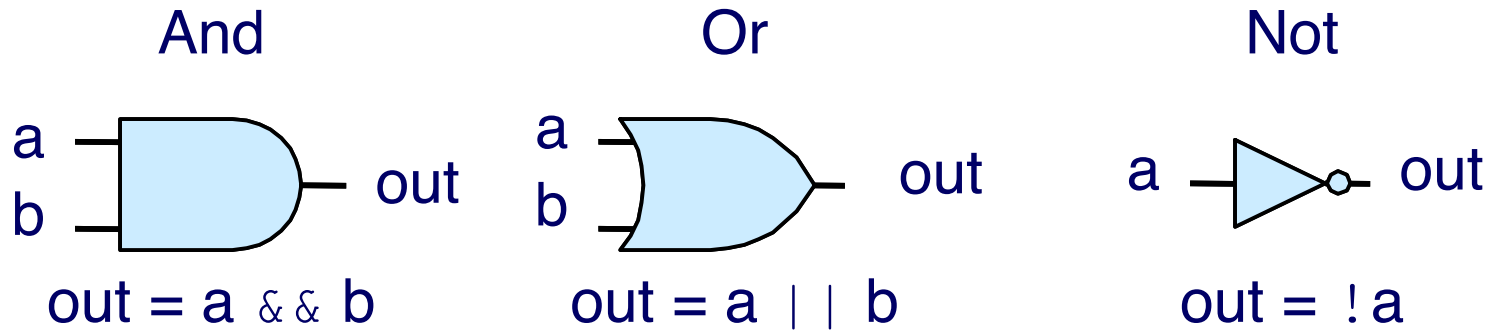
# Computing with Logic Gates



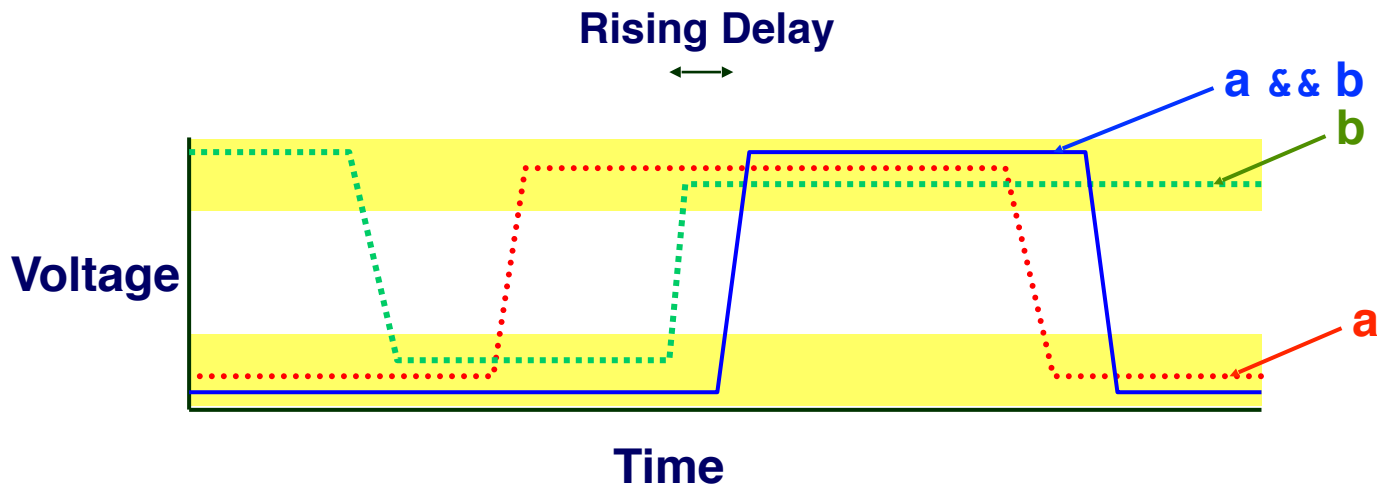
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay



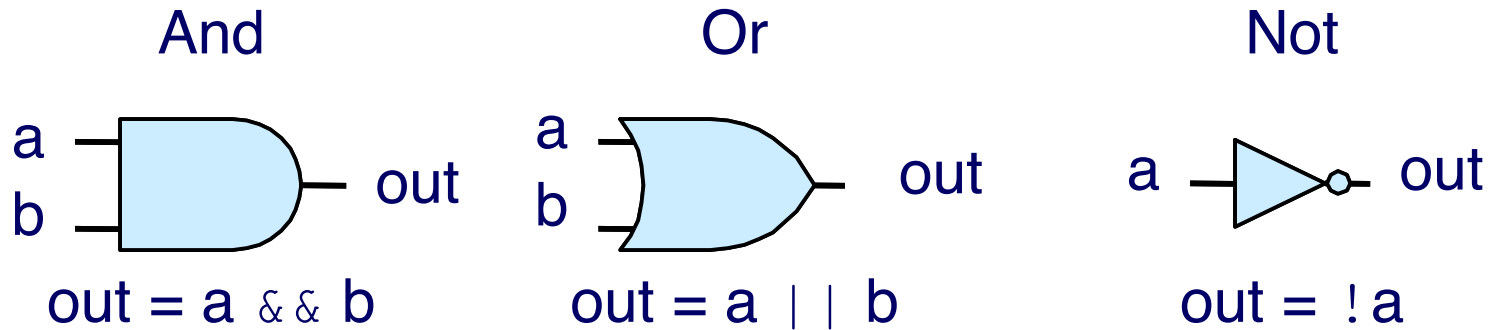
# Computing with Logic Gates



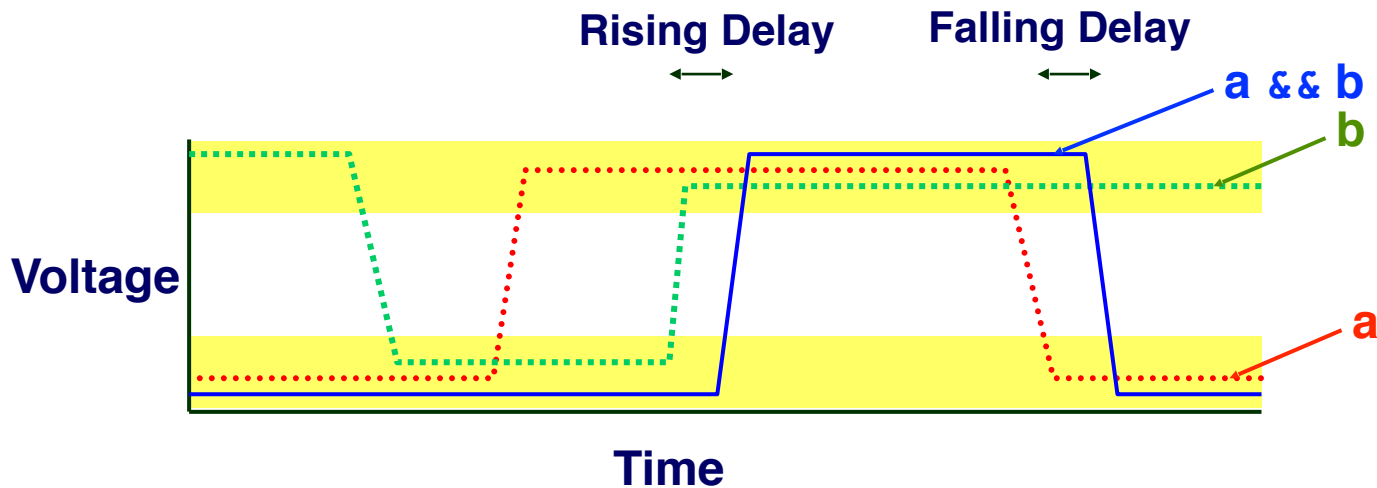
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay



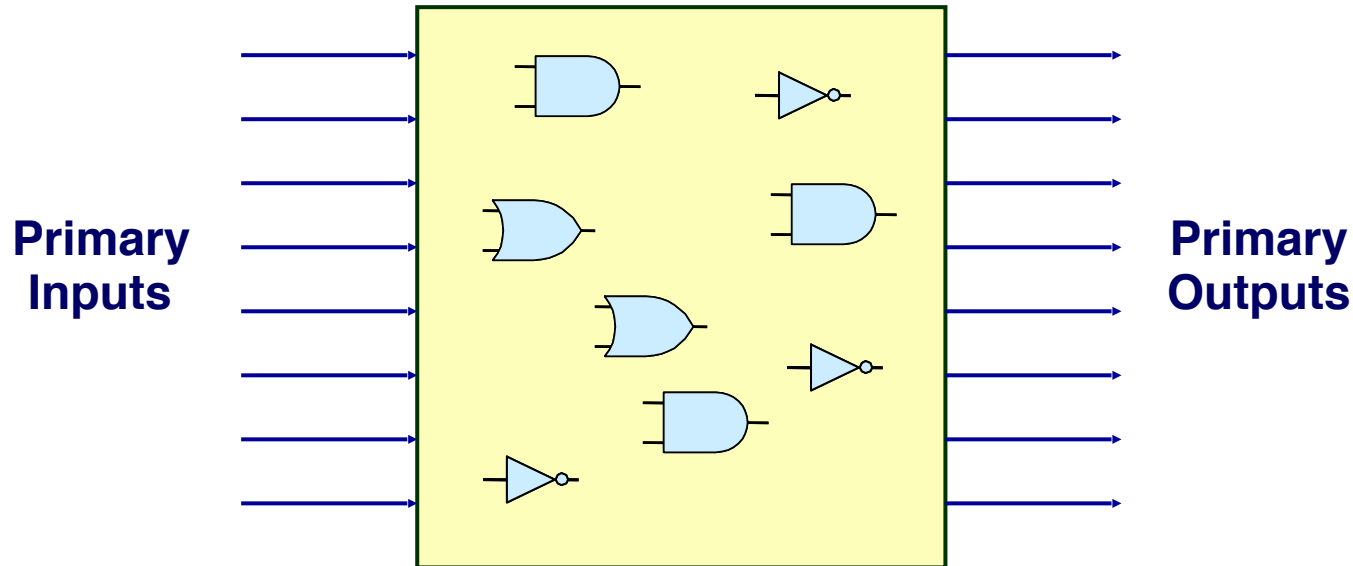
# Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay



# 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

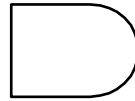


# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```

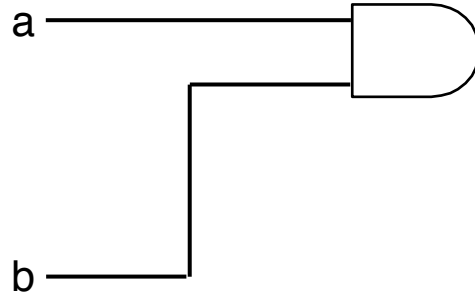
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



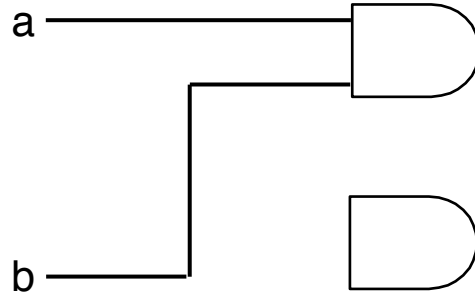
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



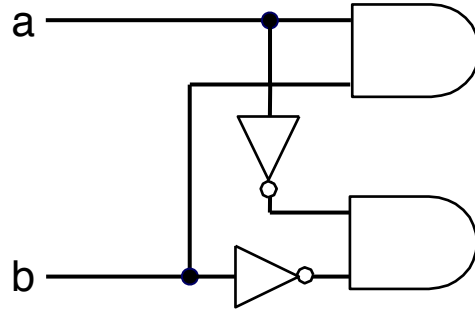
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



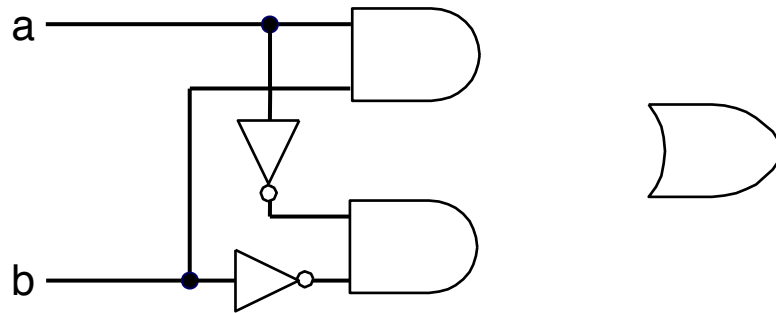
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



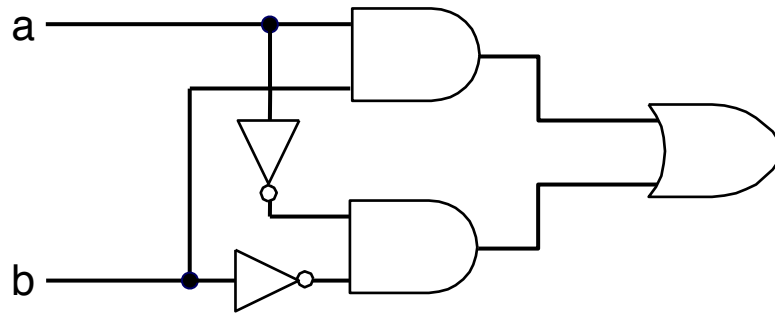
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



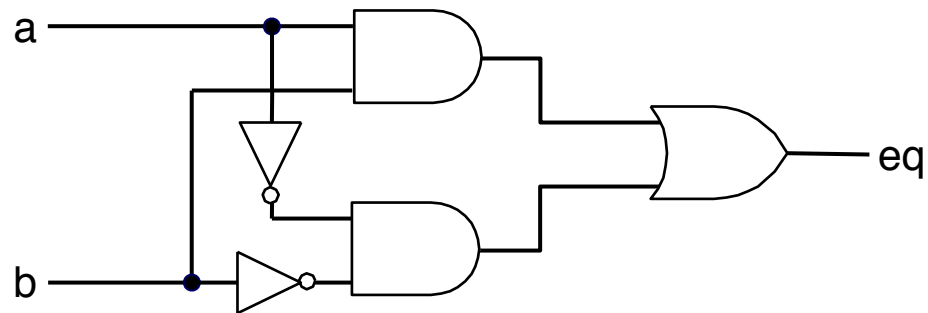
# Bit Equality

```
bool eq = (a&&b) || (!a&&!b)
```



# Bit Equality

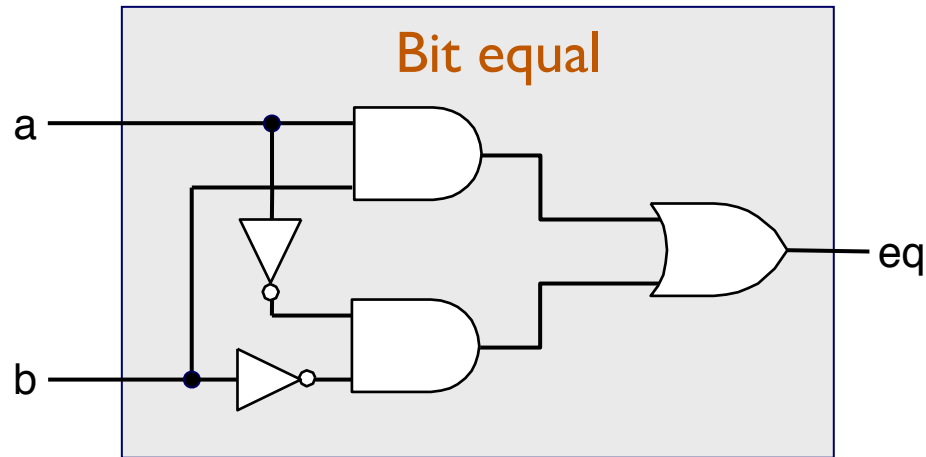
```
bool eq = (a&&b) || (!a&&!b)
```





# Bit Equality

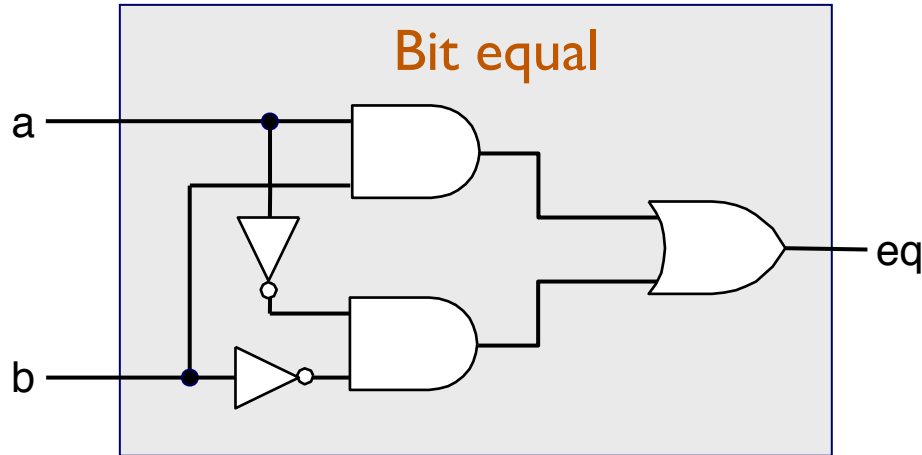
```
bool eq = (a&&b) || (!a&&!b)
```



# Bit Equality

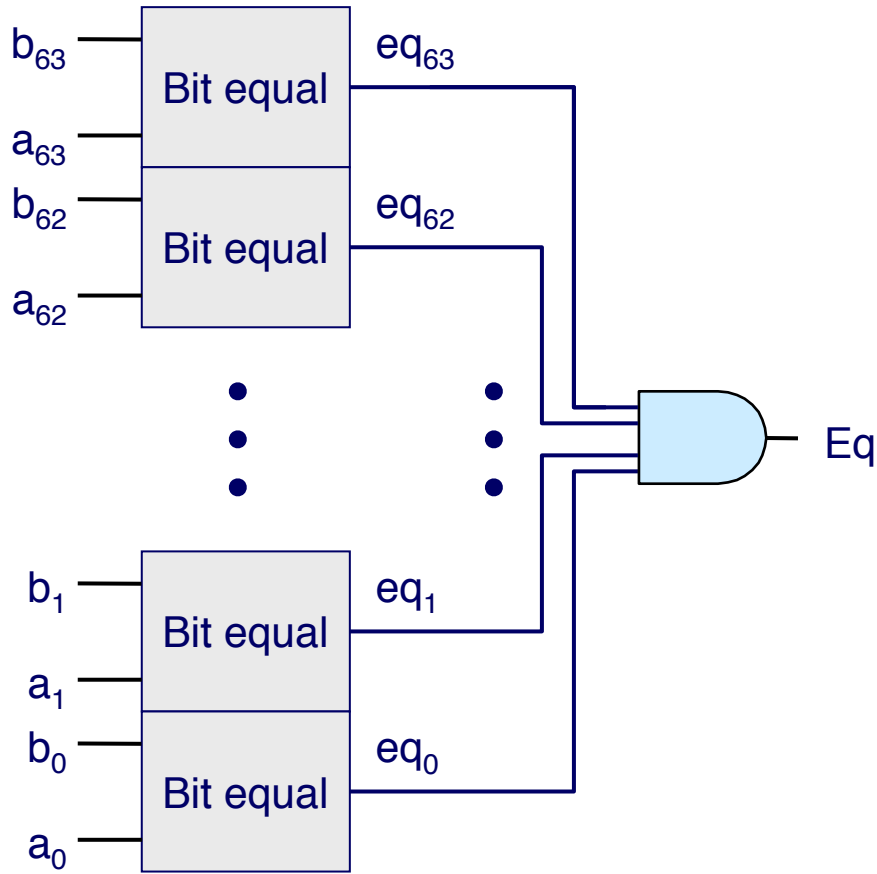
HCL Expression

```
bool eq = (a&&b) || (!a&&!b)
```

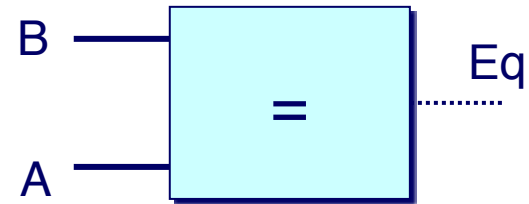


- Hardware Control Language (HCL)
  - Very simple hardware description language
    - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors

# Word Equality



## Word-Level Representation



## HCL Representation

```
bool Eq = (A == B)
```

# 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  $s=0$

## HCL Expression

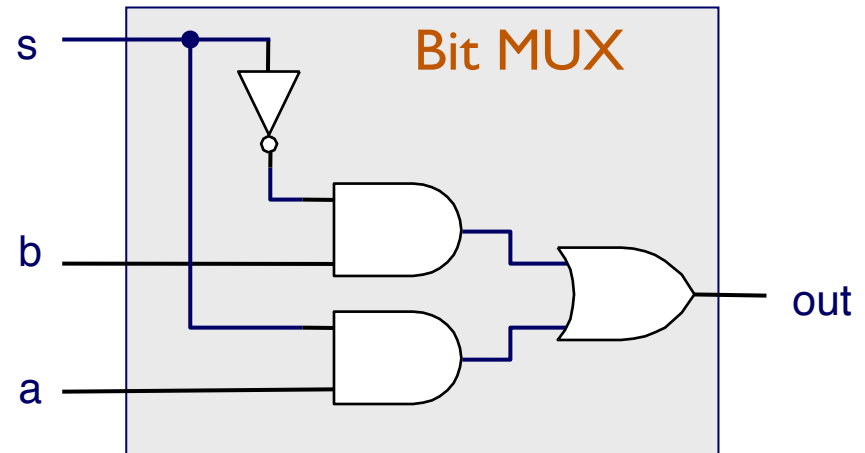
```
bool out = (s&&a) || (!s&&b)
```

# Bit-Level Multiplexor (MUX)

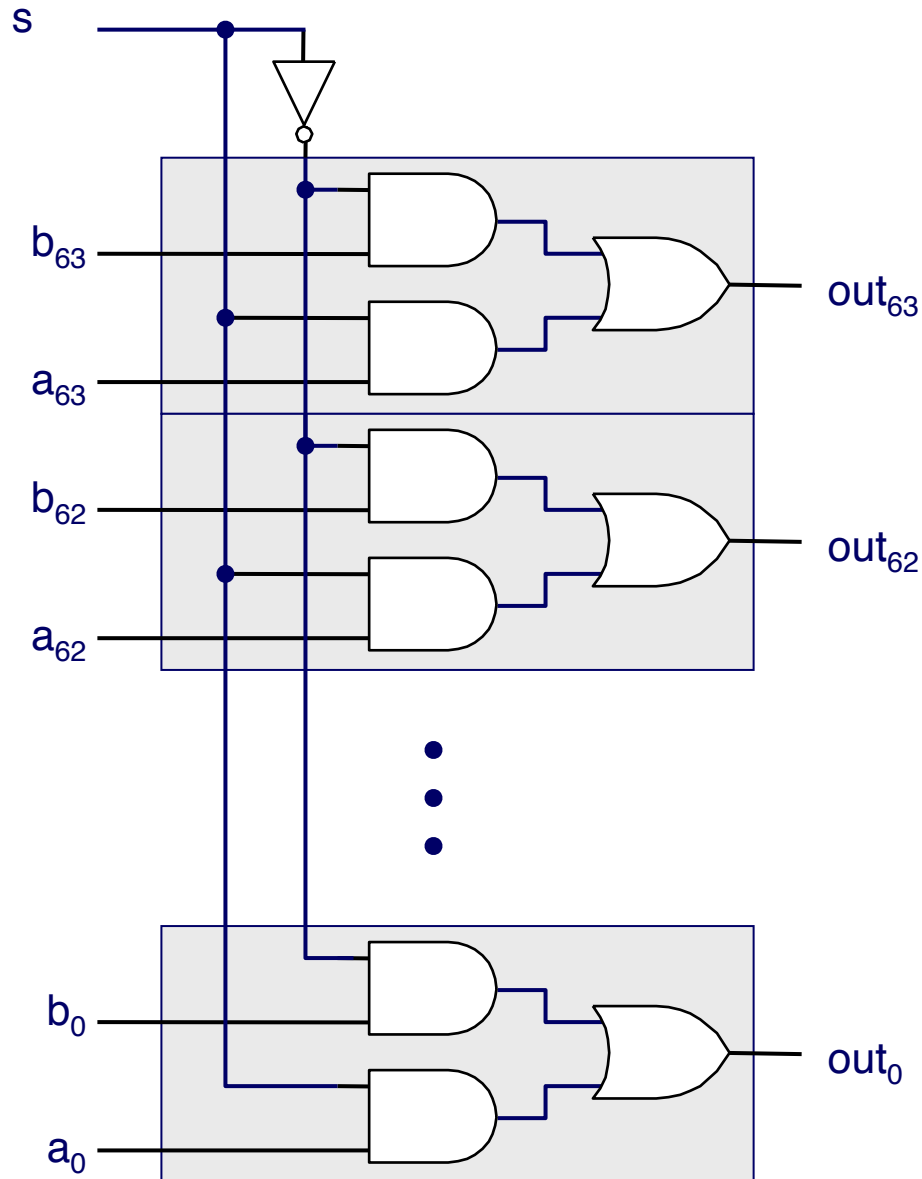
- Control signal  $s$
- Data signals  $a$  and  $b$
- Output  $a$  when  $s=1$ ,  $b$  when  $s=0$

## HCL Expression

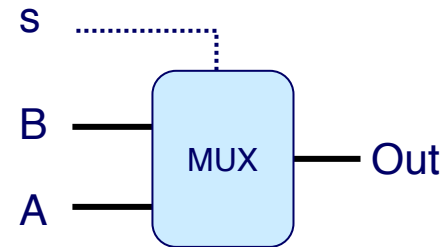
```
bool out = (s&&a) || (!s&&b)
```



# Word Multiplexor



## Word-Level Representation



## HCL Representation

```
int Out = [  
    s : A;  
    1 : B;  
];
```

- Select input word  $A$  or  $B$  depending on control signal  $s$
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

# 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

# Full (1-bit) Adder

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

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

# Full (1-bit) Adder

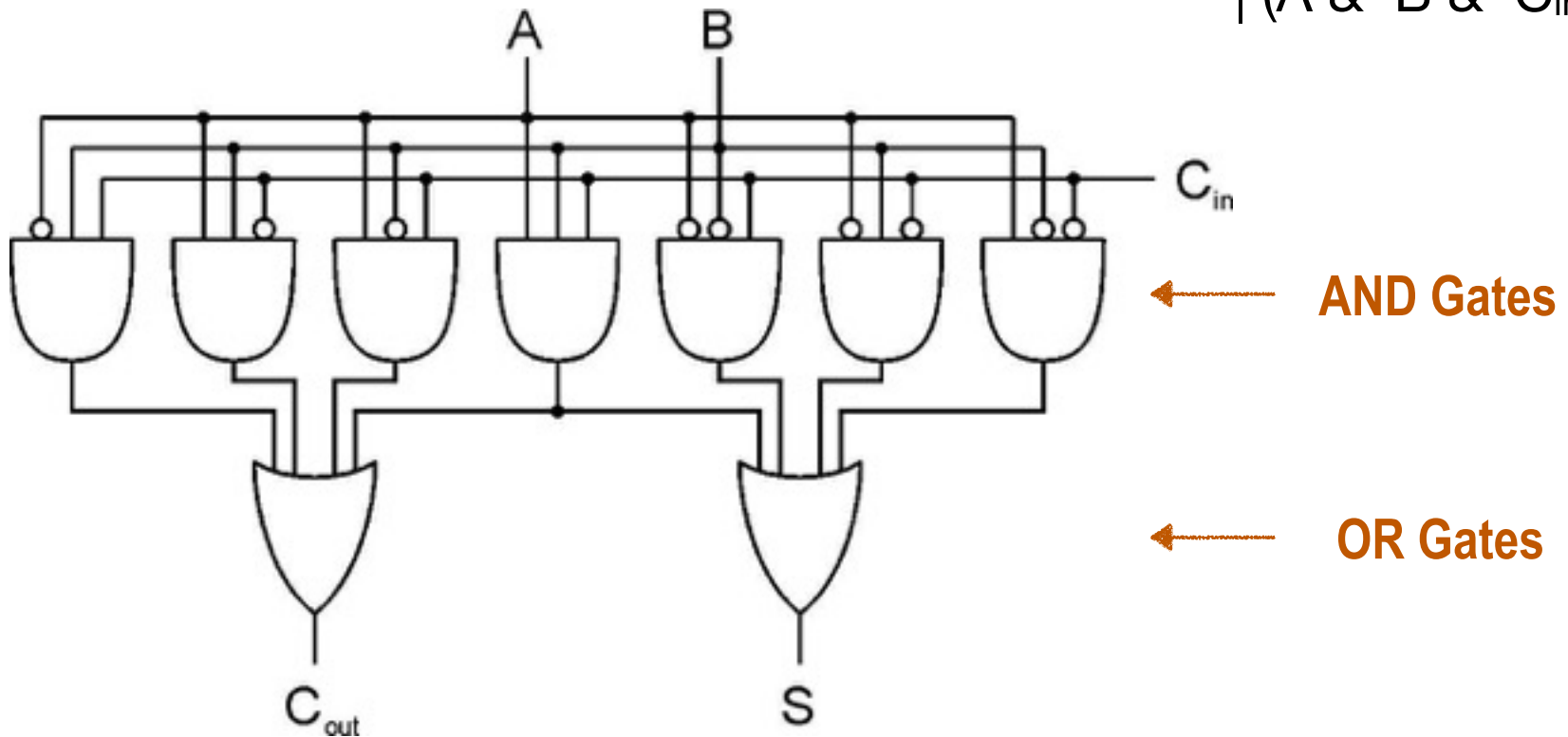
$$C_{ou} = (\sim A \& B \& C_{in})$$

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

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

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

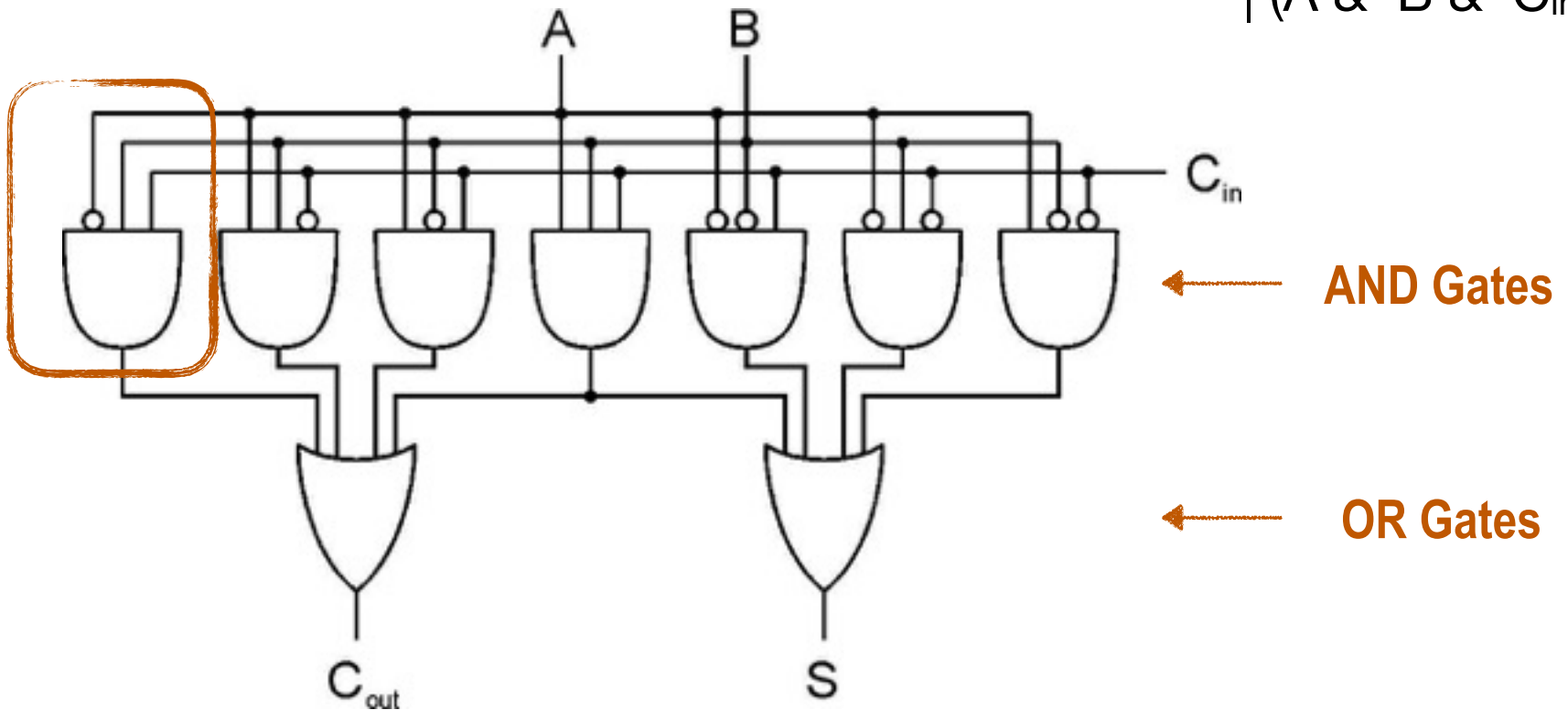
Add two bits and carry-in,  
produce one-bit sum and carry-out.



# Full (1-bit) Adder

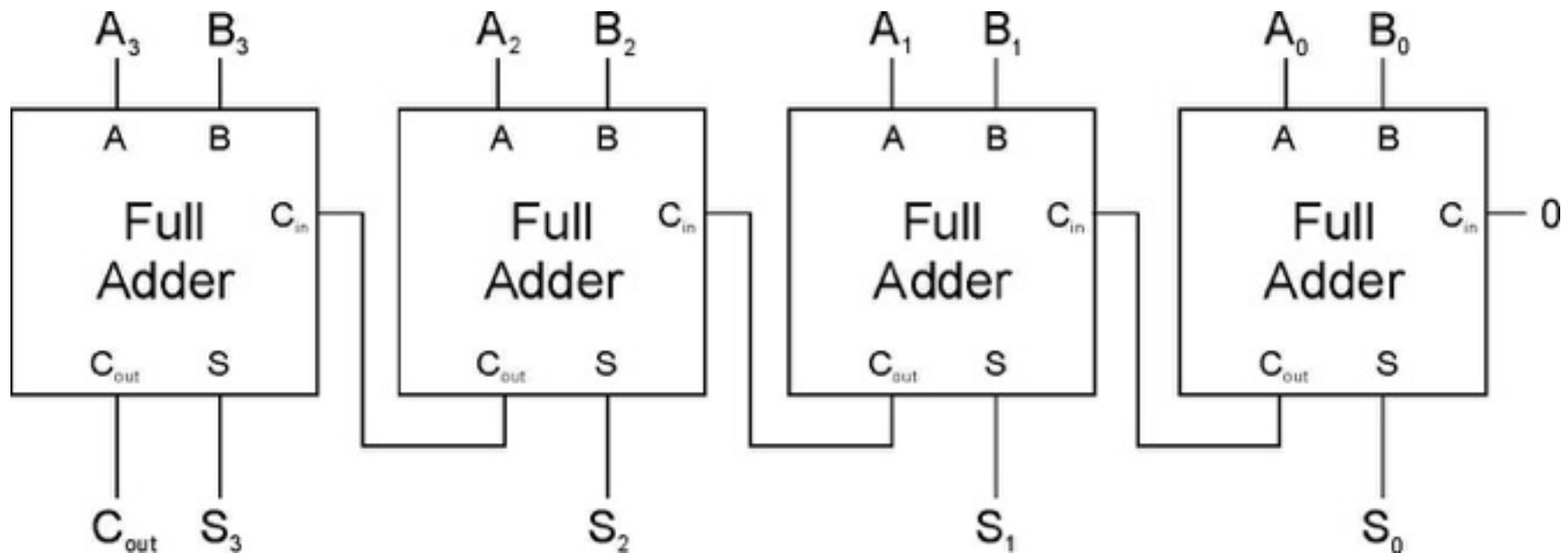
$$C_{ou} = (\sim A \& B \& C_{in}) \mid (A \& \sim B \& C_{in}) \mid (A \& B \& \sim C_{in}) \mid (A \& B \& C_{in})$$

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



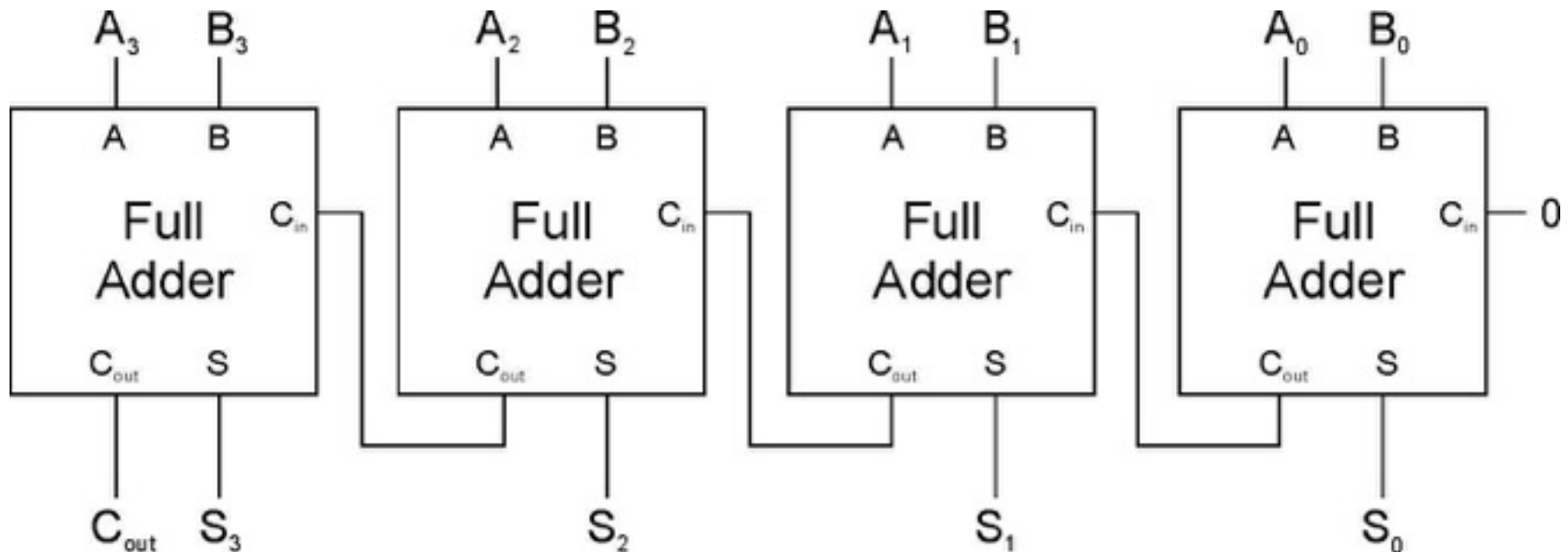


# Four-bit Adder



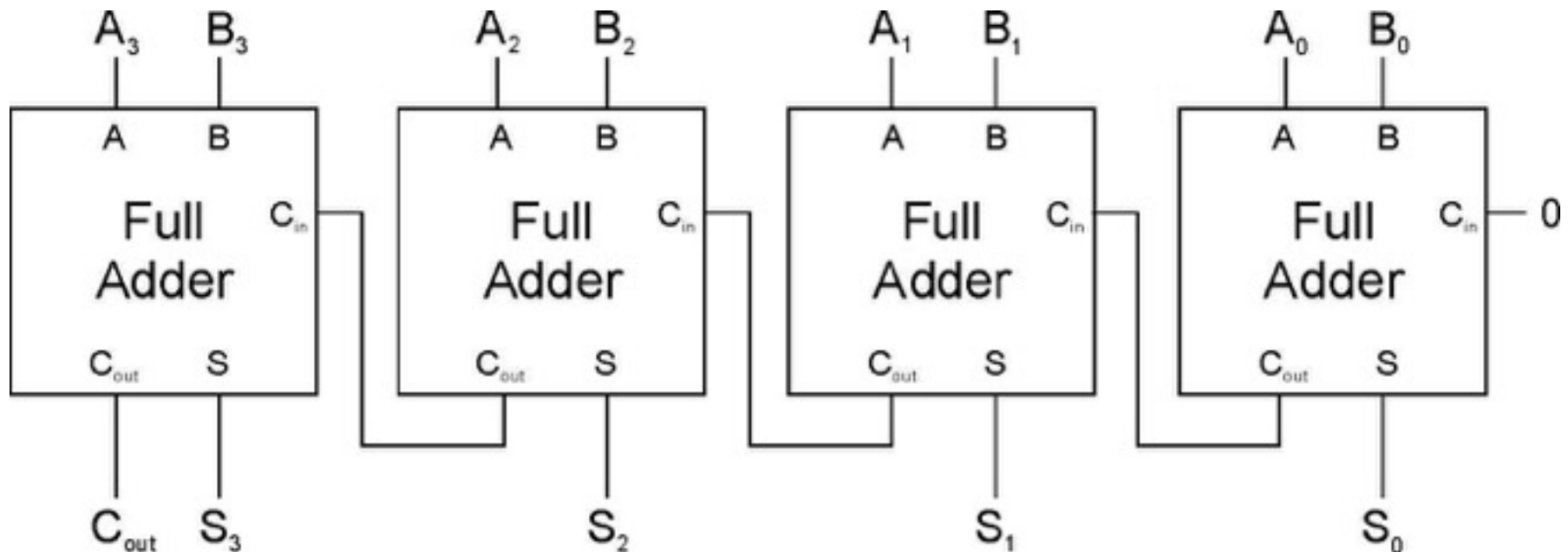
# Four-bit Adder

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

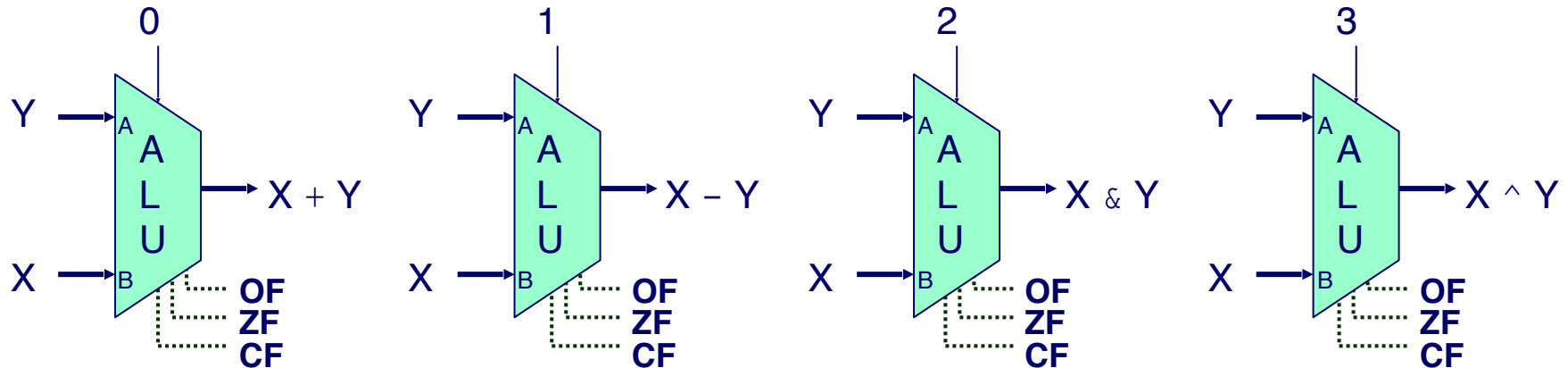


# Four-bit Adder

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



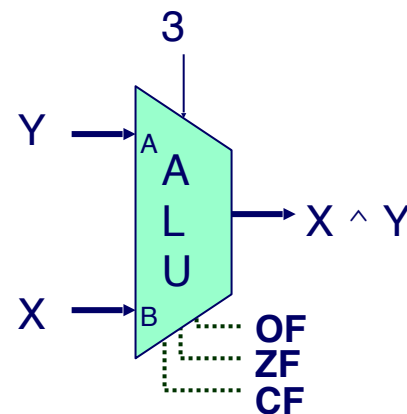
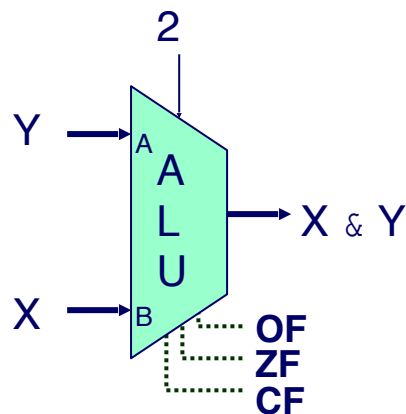
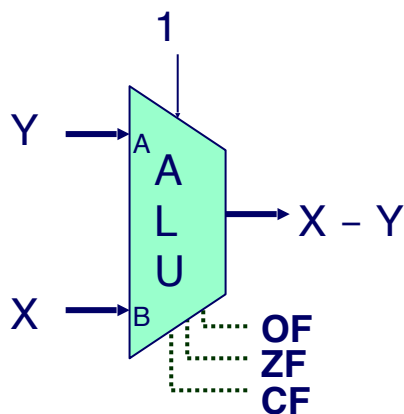
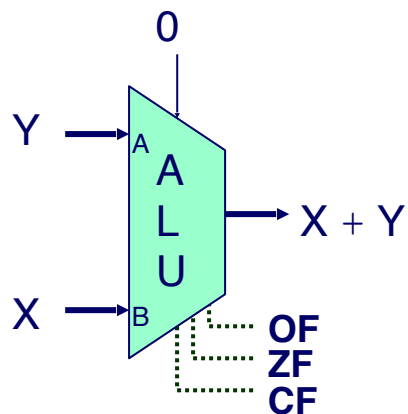
# Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - add, subtract, and, or
- Also computes values for condition codes

# Arithmetic Logic Unit

# Questions?

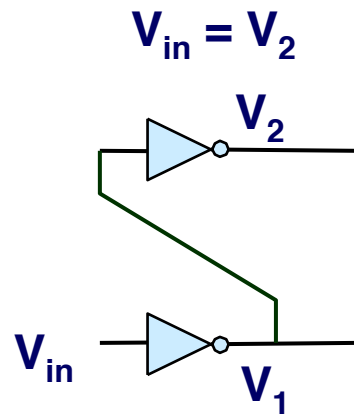


- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - add, subtract, and, or
- Also computes values for condition codes

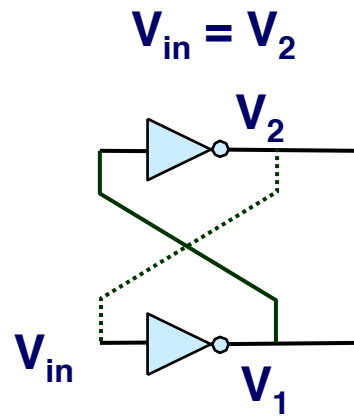
# Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

# Storing 1 Bit

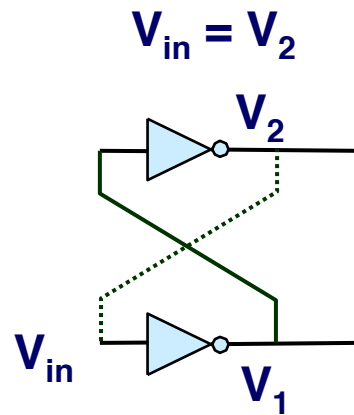


# Storing 1 Bit

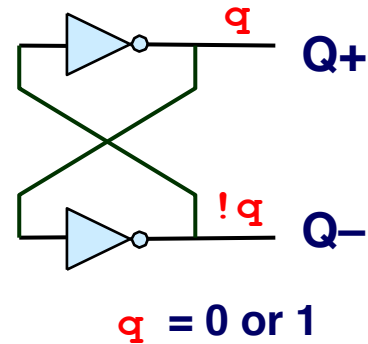




# Storing 1 Bit

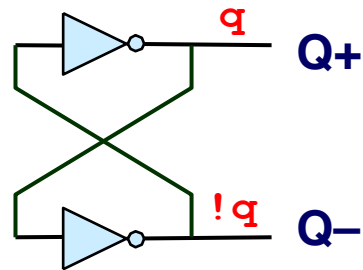


## Bistable Element



# Storing and Accessing 1 Bit

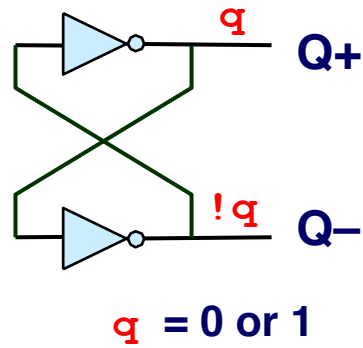
## Bistable Element



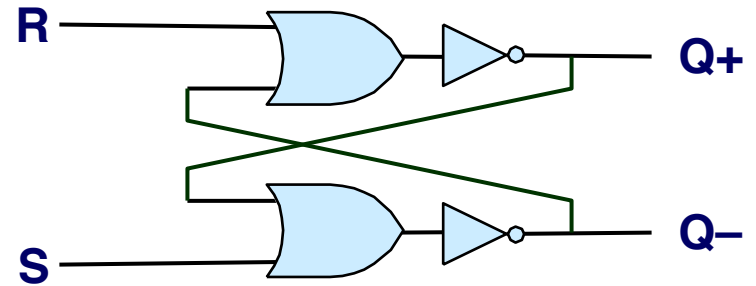
$q = 0 \text{ or } 1$

# Storing and Accessing 1 Bit

Bistable Element

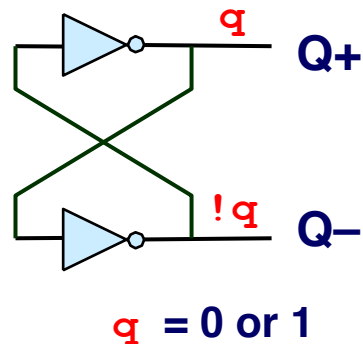


R-S Latch

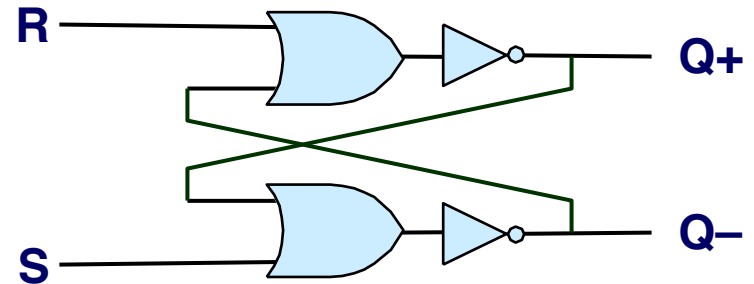


# Storing and Accessing 1 Bit

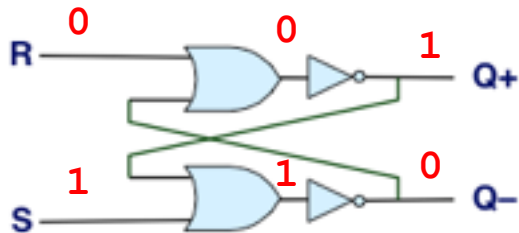
Bistable Element



R-S Latch

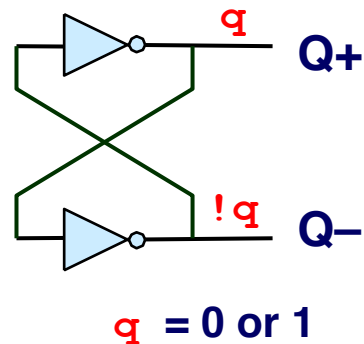


Setting

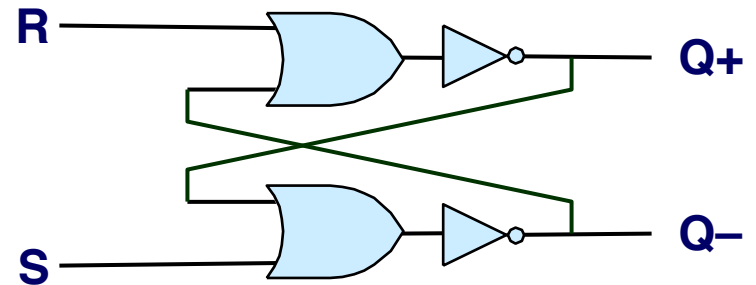


# Storing and Accessing 1 Bit

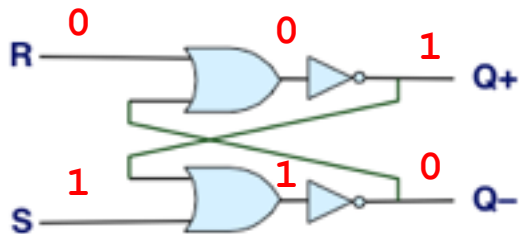
Bistable Element



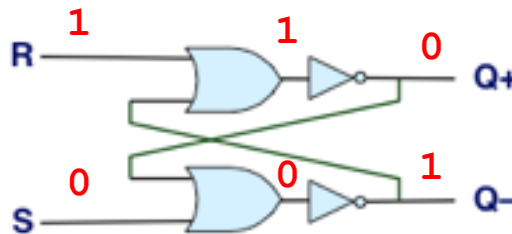
R-S Latch



Setting

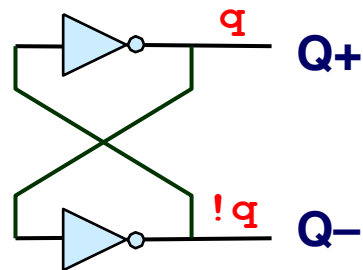


Resetting



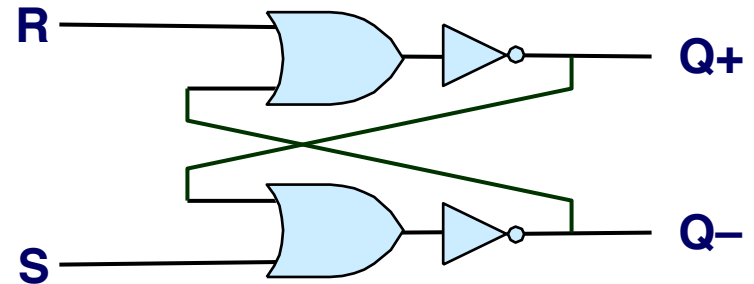
# Storing and Accessing 1 Bit

Bistable Element

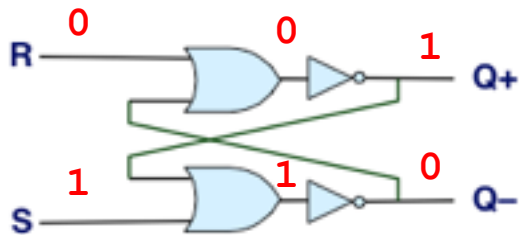


$q = 0 \text{ or } 1$

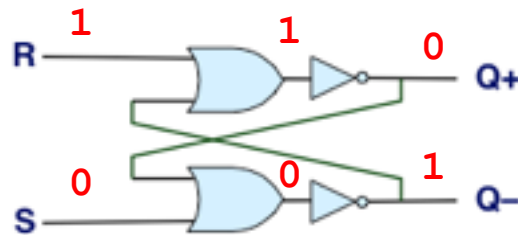
R-S Latch



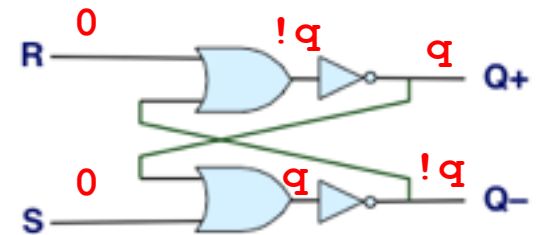
Setting



Resetting

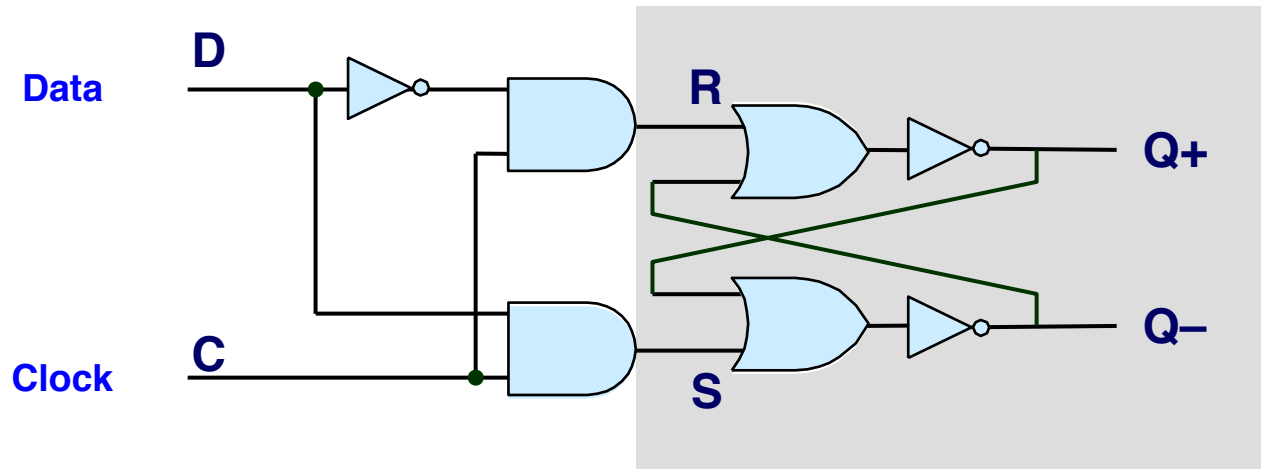


Storing



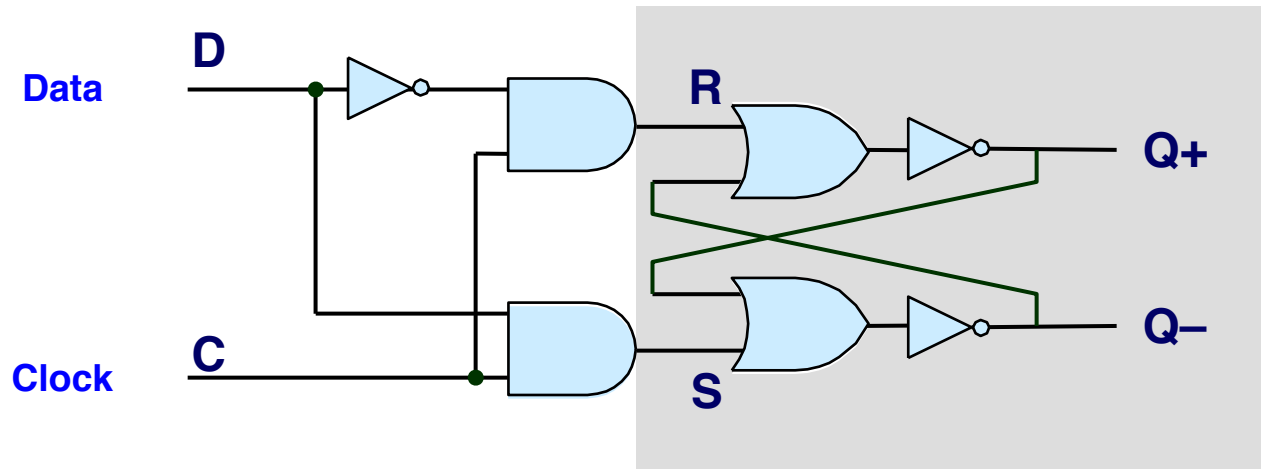
# 1-Bit D Latch

D Latch

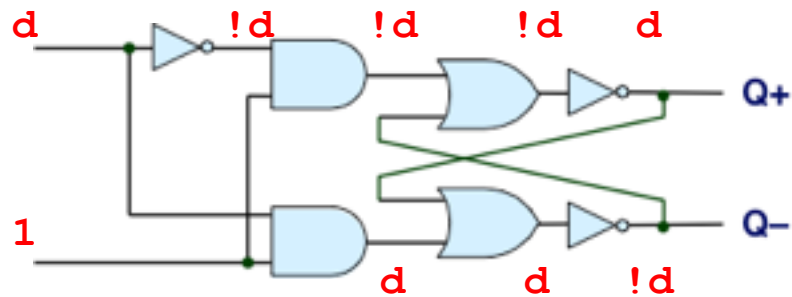


# 1-Bit D Latch

D Latch



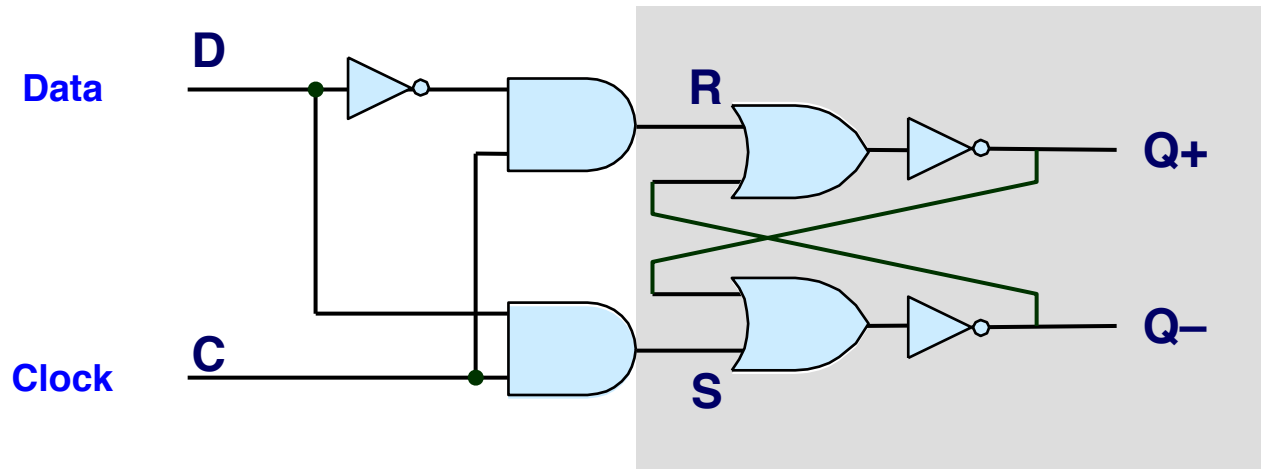
Latching



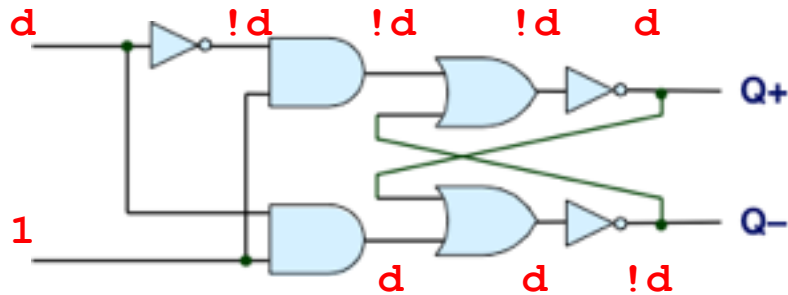


# 1-Bit D Latch

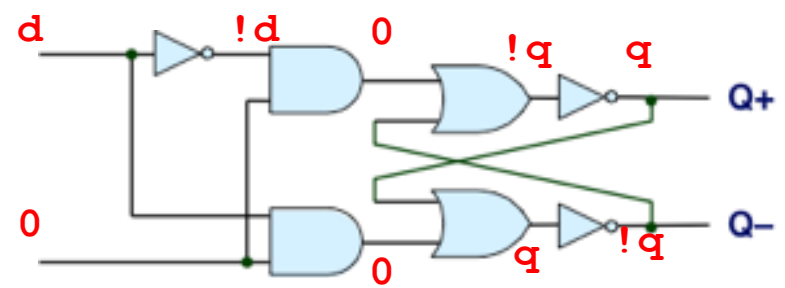
D Latch



Latching

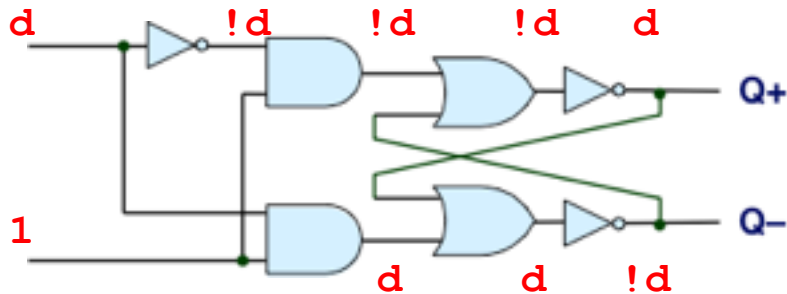


Storing

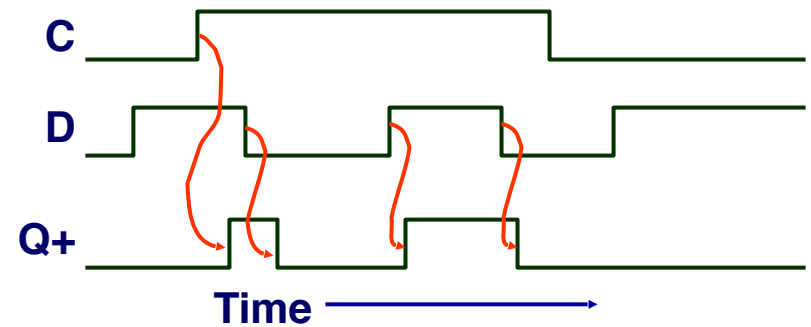


# D-Latch is Transparent (Level-Triggered)

## Latching

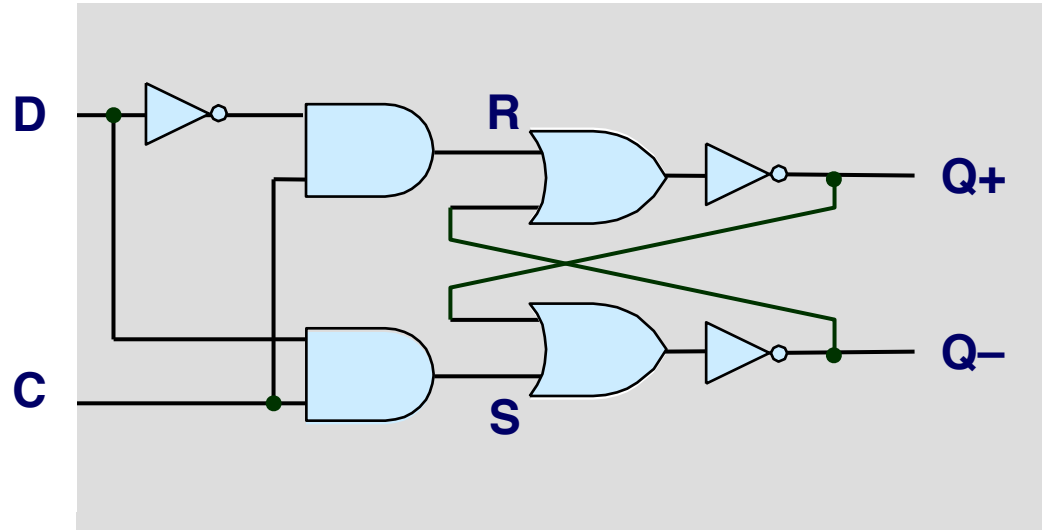


## Changing D

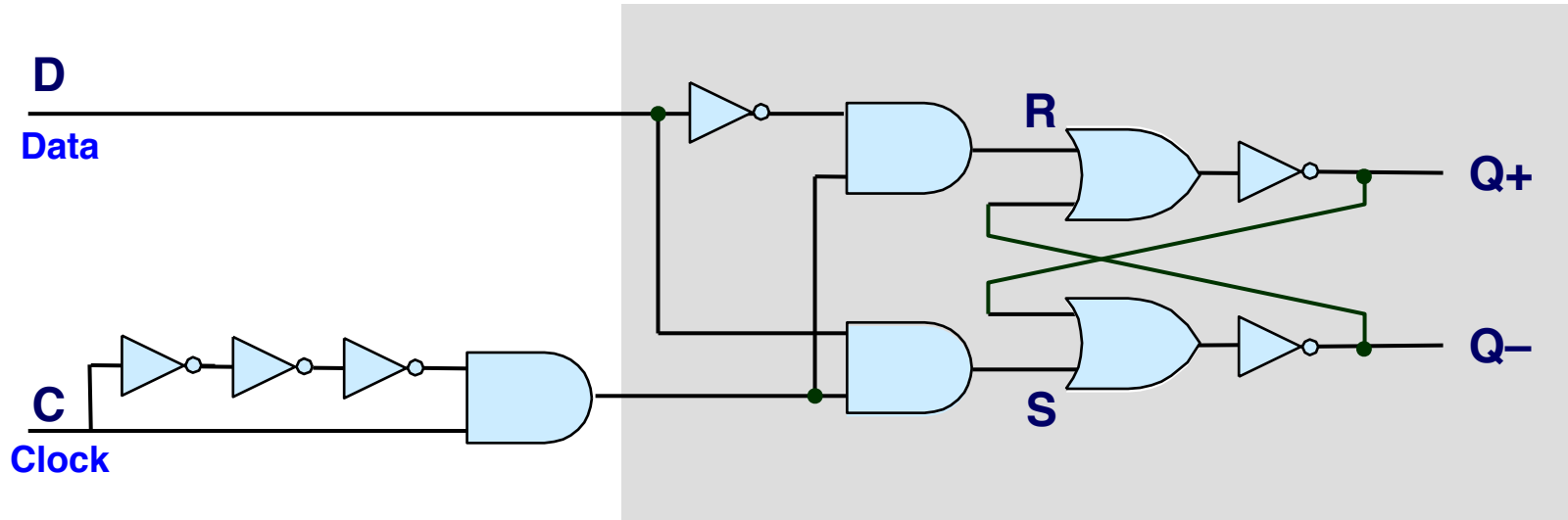


- When in latching mode, combinational propagation from D to Q+ and Q-
- Value latched depends on value of D as C falls

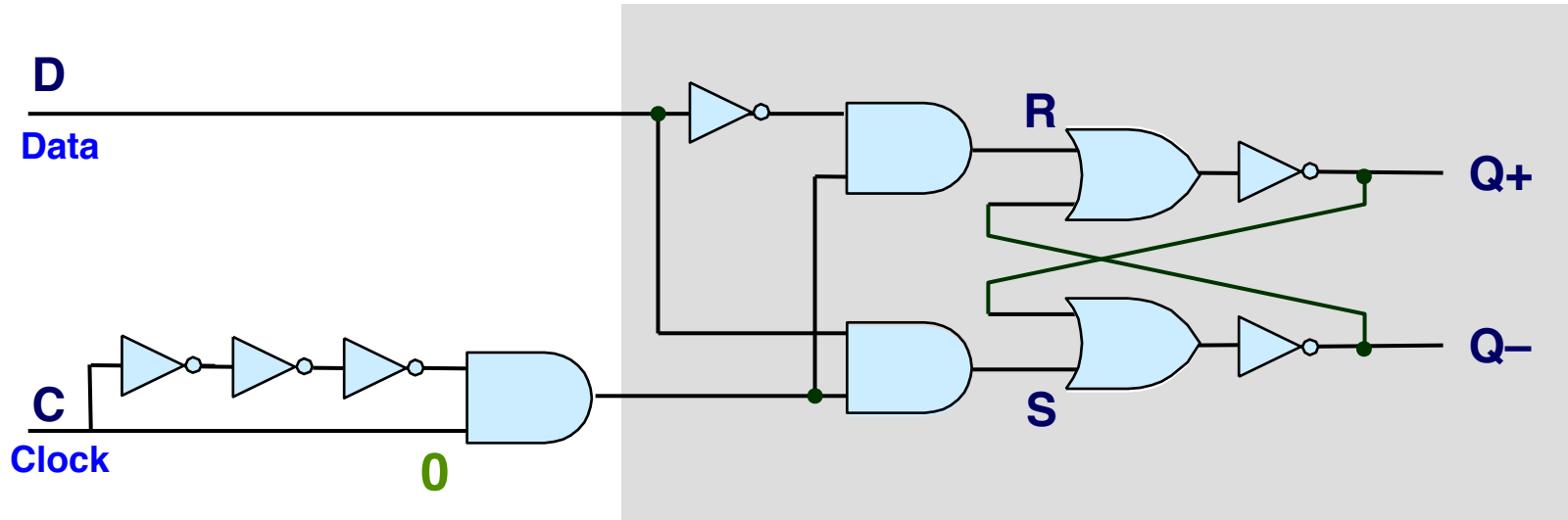
# Edge-Triggered Latch (Flip-Flop)



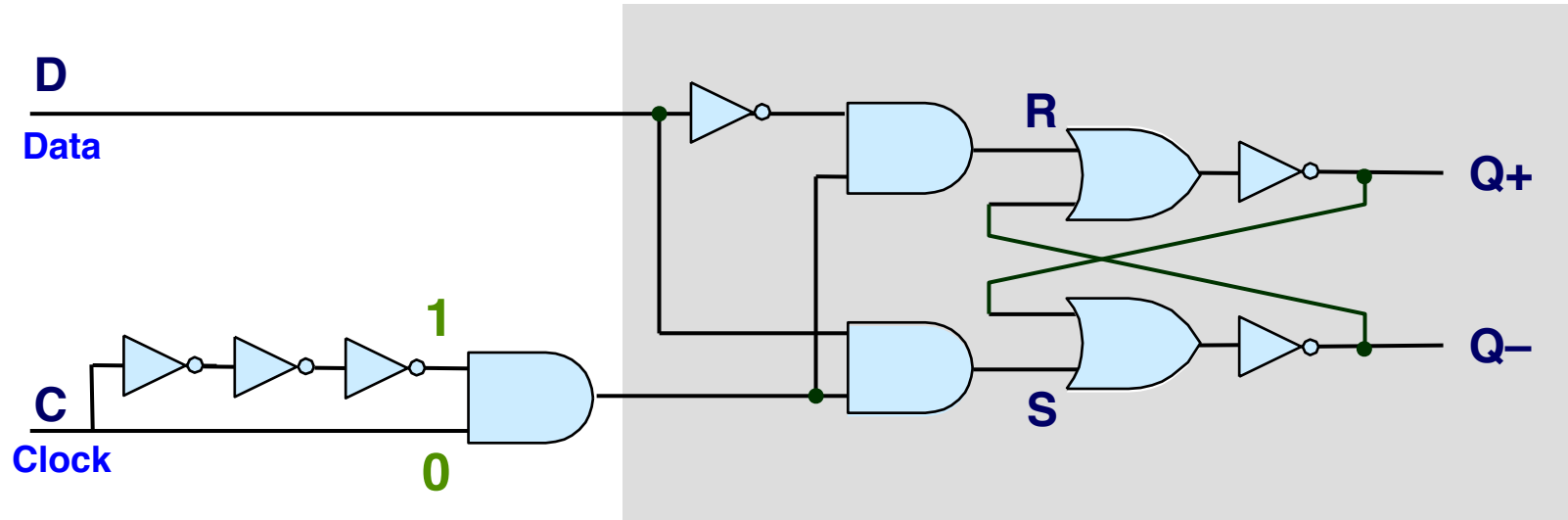
# Edge-Triggered Latch (Flip-Flop)



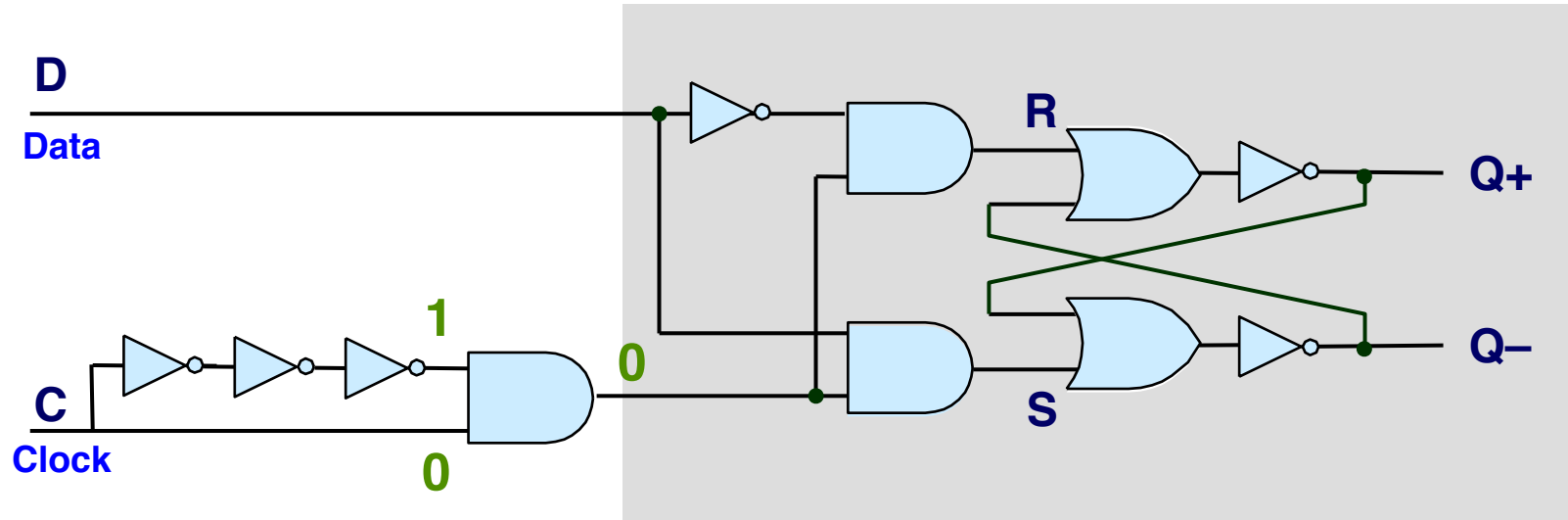
# Edge-Triggered Latch (Flip-Flop)



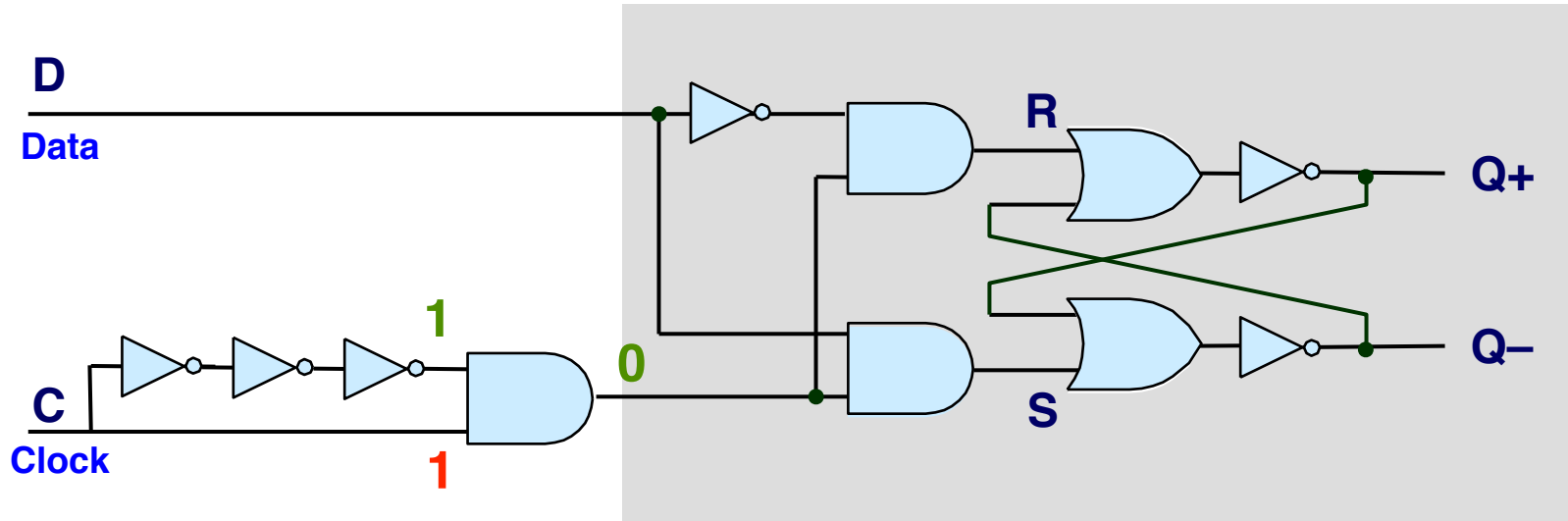
# Edge-Triggered Latch (Flip-Flop)



# Edge-Triggered Latch (Flip-Flop)

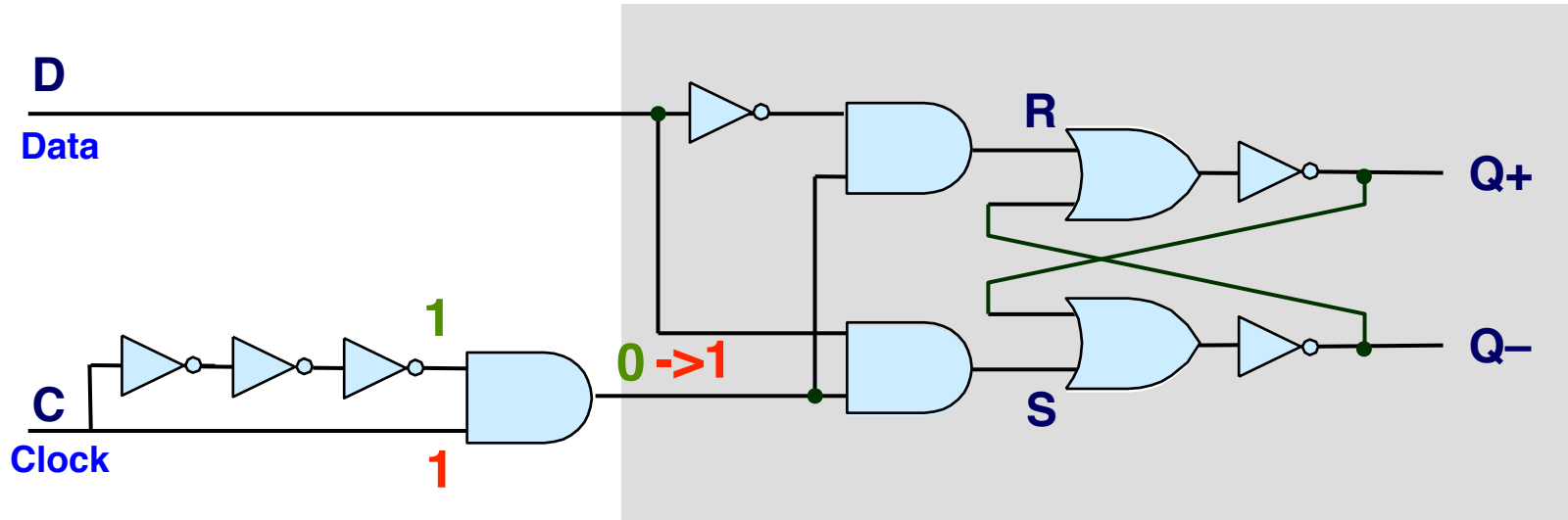


# Edge-Triggered Latch (Flip-Flop)

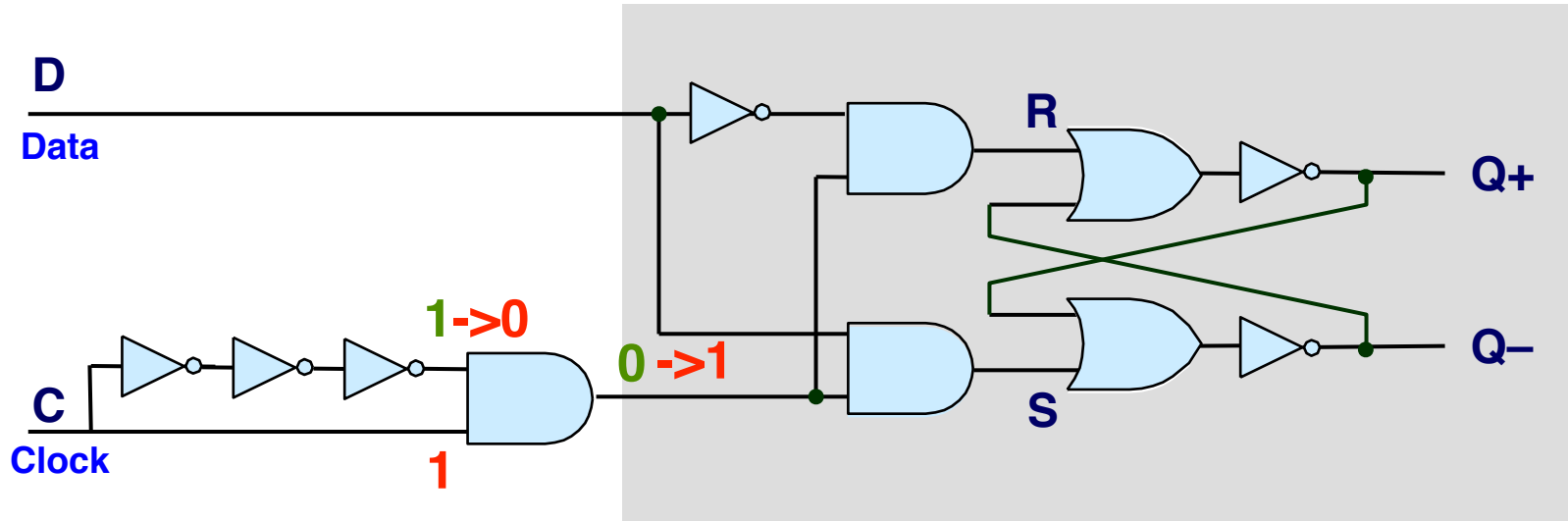




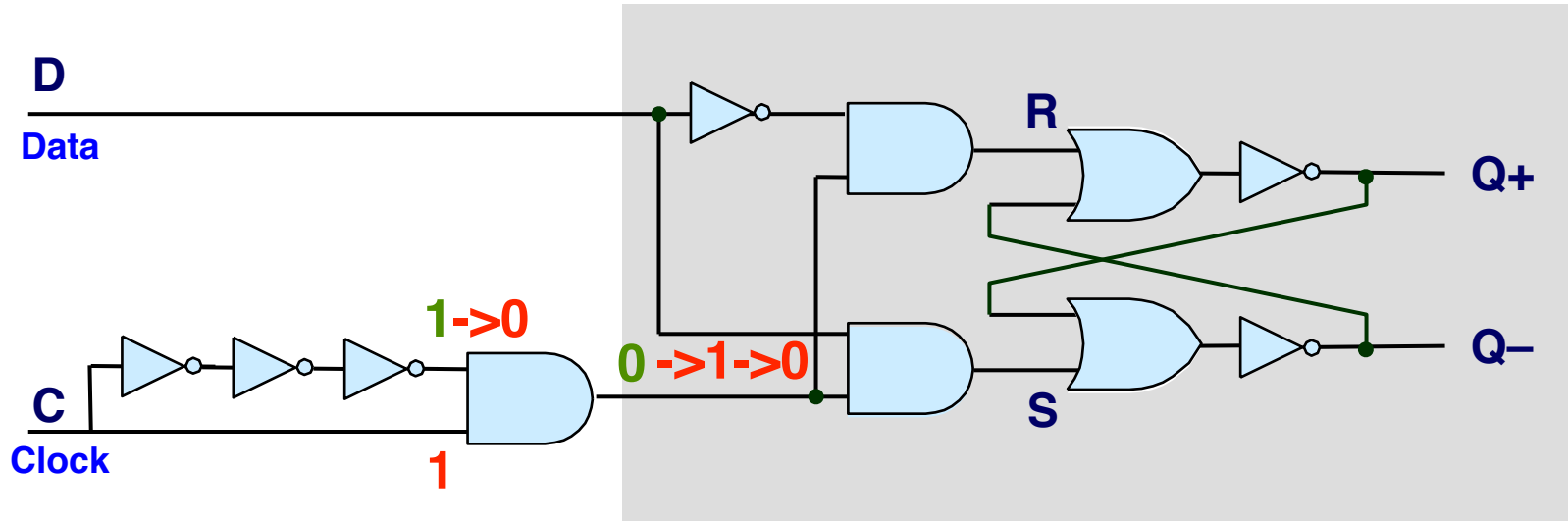
# Edge-Triggered Latch (Flip-Flop)



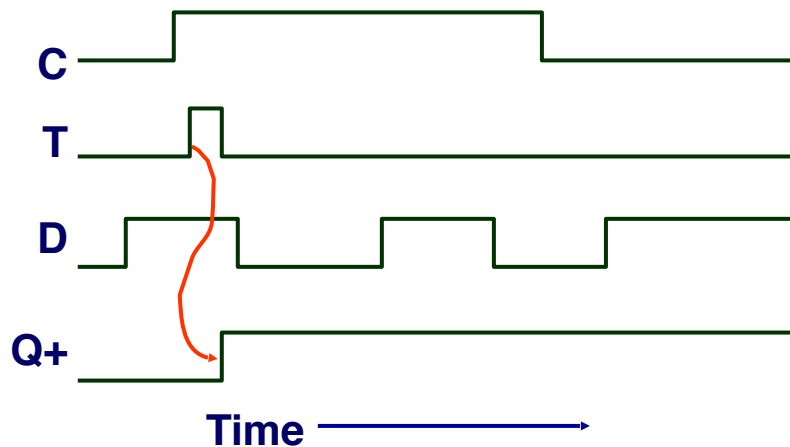
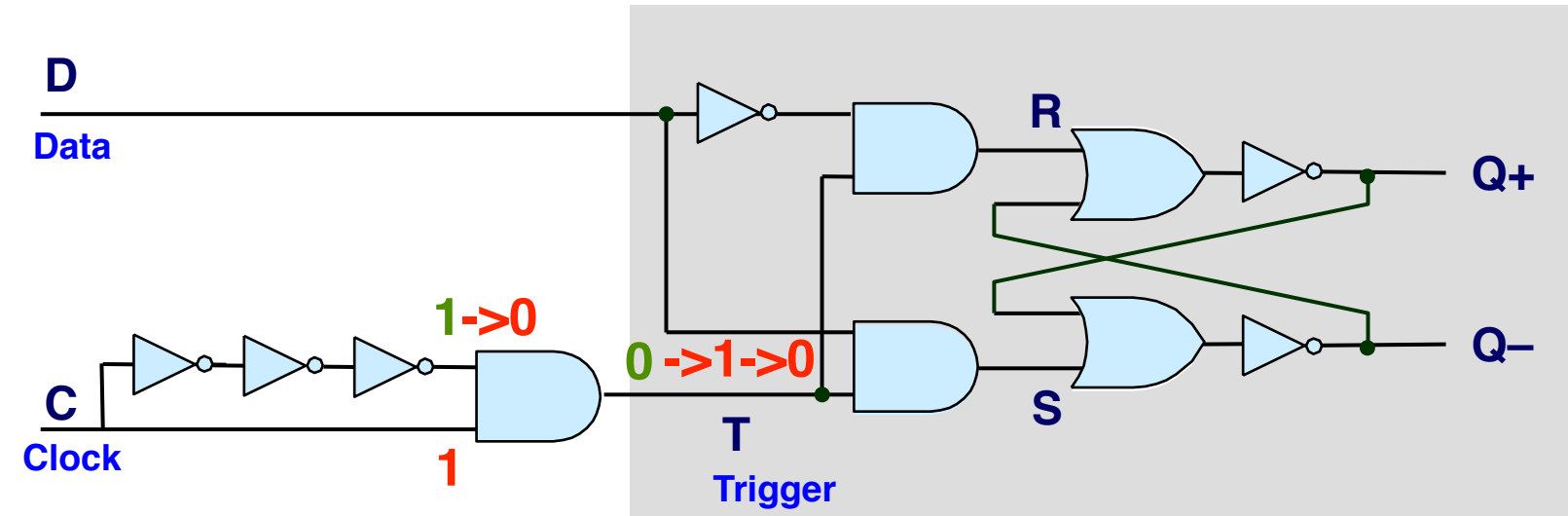
# Edge-Triggered Latch (Flip-Flop)



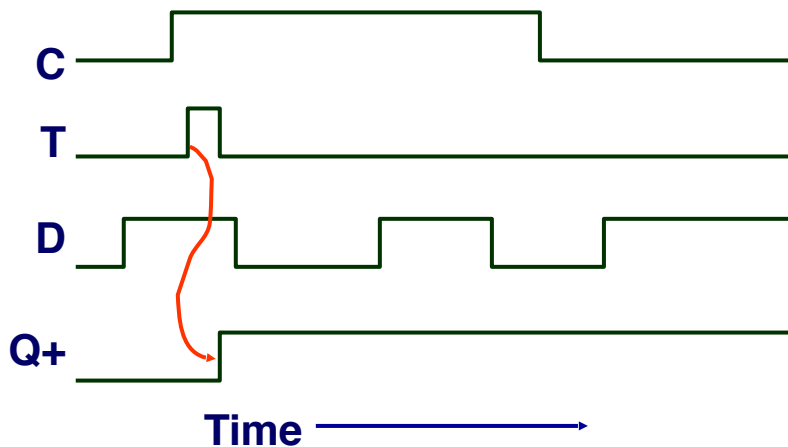
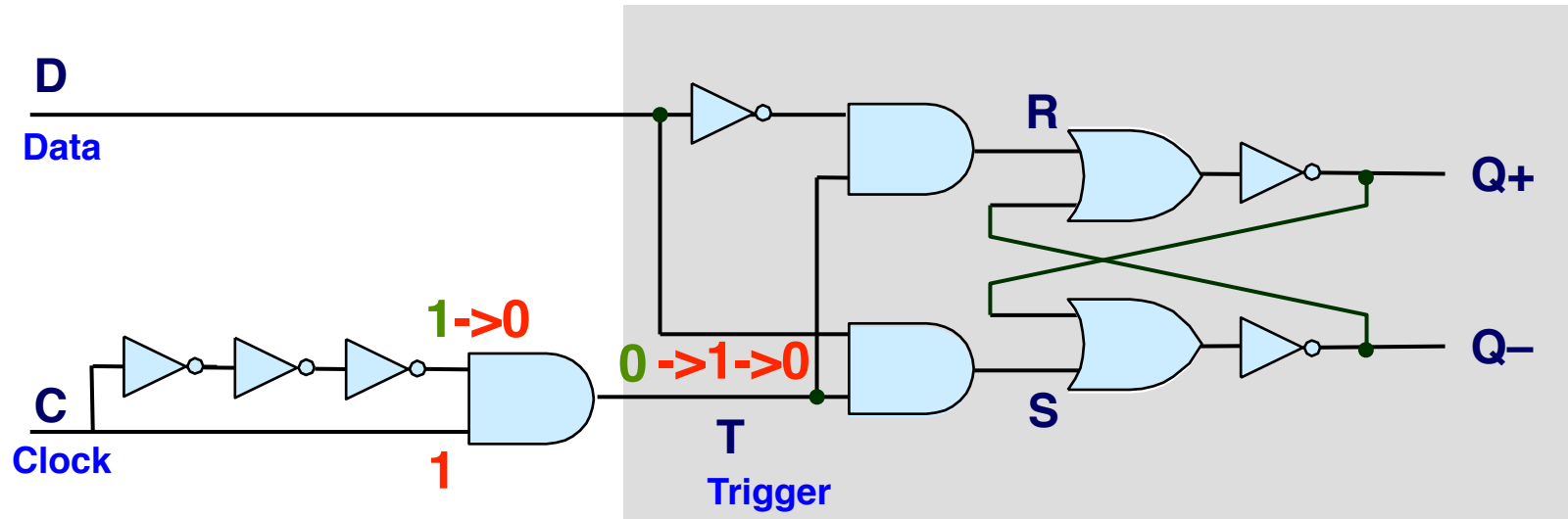
# Edge-Triggered Latch (Flip-Flop)



# Edge-Triggered Latch (Flip-Flop)

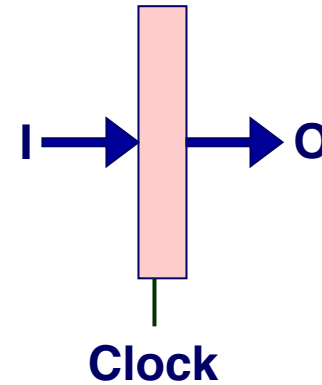
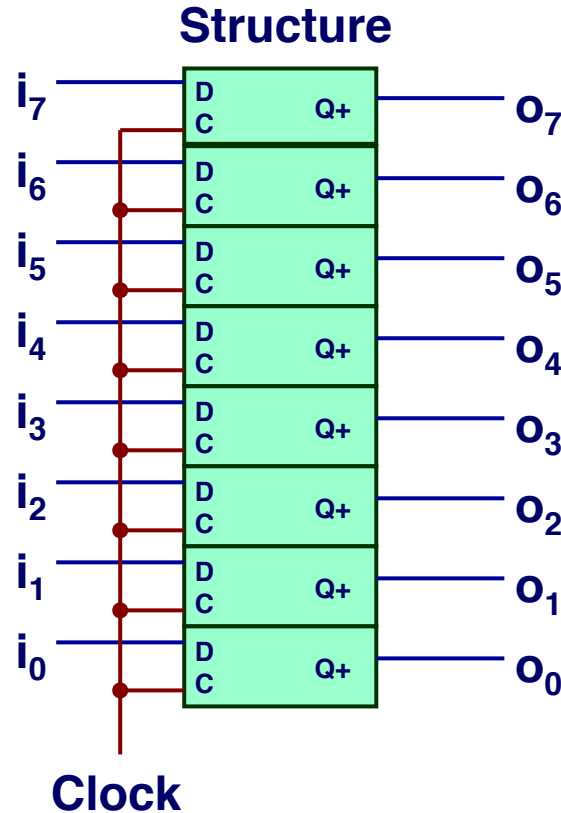


# Edge-Triggered Latch (Flip-Flop)



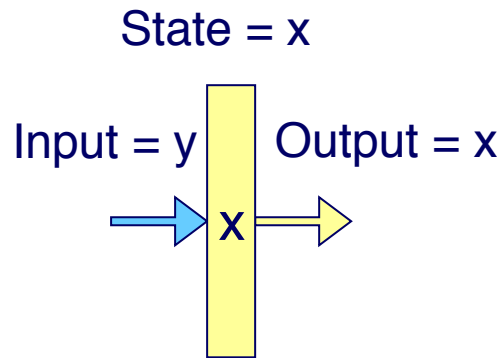
- Flip-flop: Only in latching mode for brief period
- Value latched depends on data as clock rises
- Output remains stable at all other times

# Registers

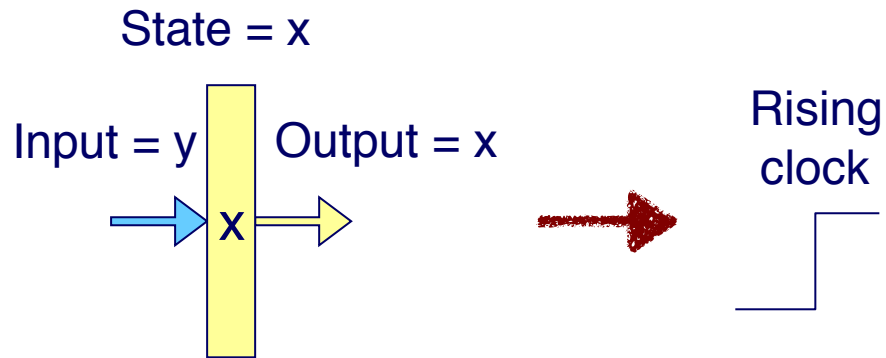


- Stores word of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of clock

# Register Operation

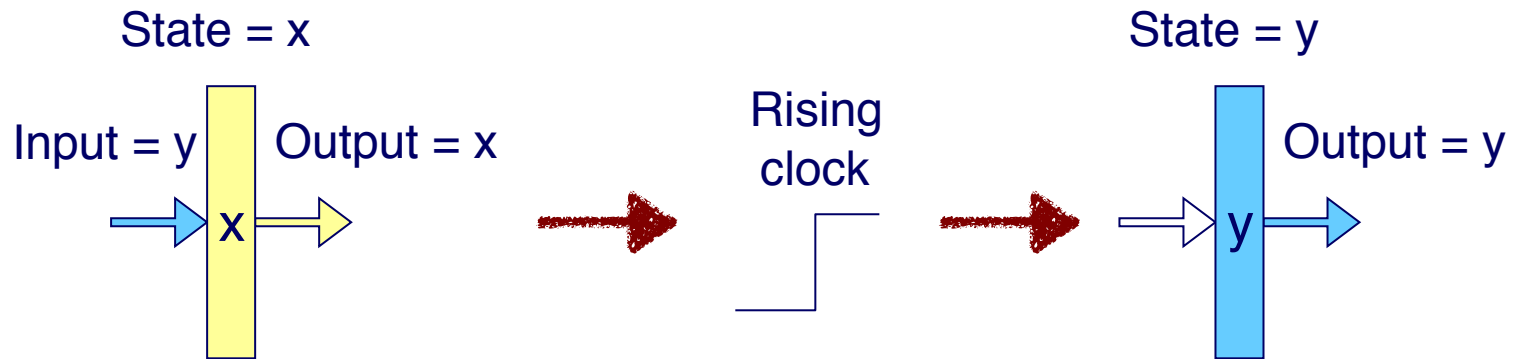


# Register Operation

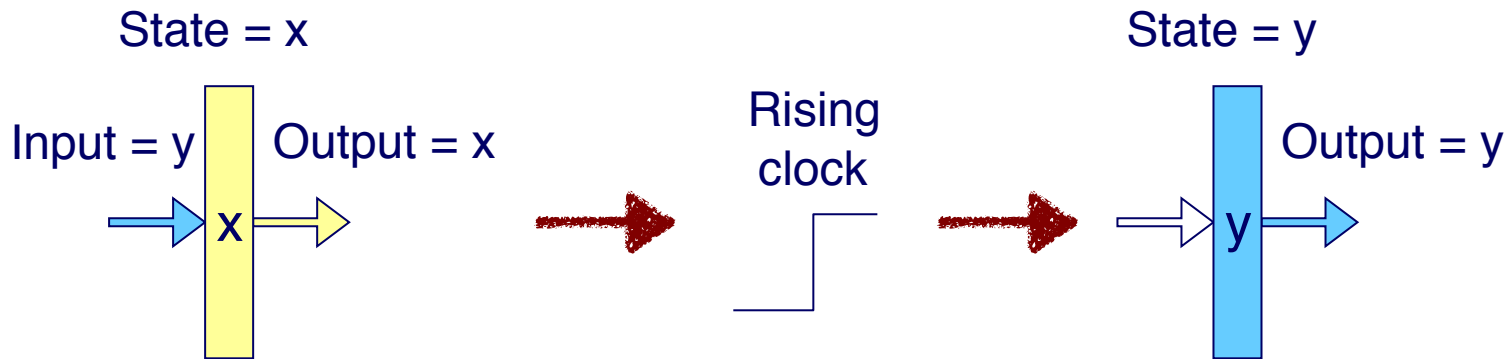




# Register Operation



# Register Operation



- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

# Decoder

A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$$D0 = \neg A1 \ \& \ \neg A0$$

$$D1 = \neg A1 \ \& \ A0$$

$$D2 = A1 \ \& \ \neg A0$$

$$D3 = A1 \ \& \ A0$$

# Decoder

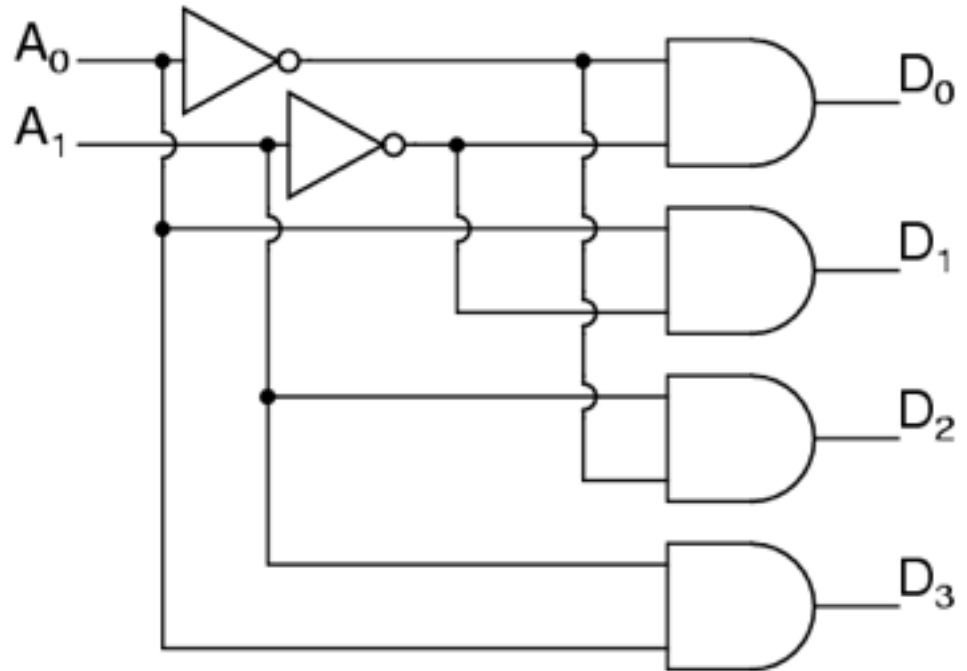
A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$$D0 = !A1 \ \& \ !A0$$

$$D1 = !A1 \ \& \ A0$$

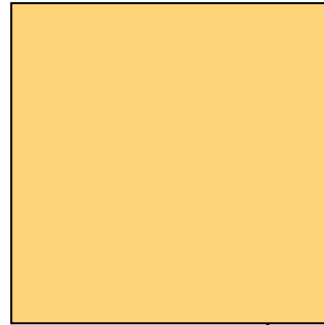
$$D2 = A1 \ \& \ !A0$$

$$D3 = A1 \ \& \ A0$$



# Register File

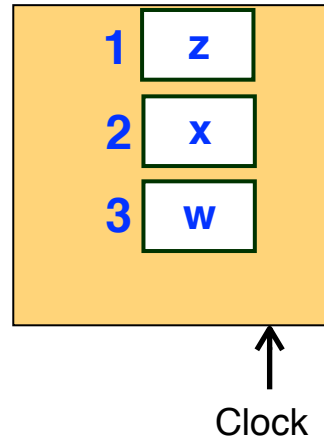
Register File



↑  
Clock

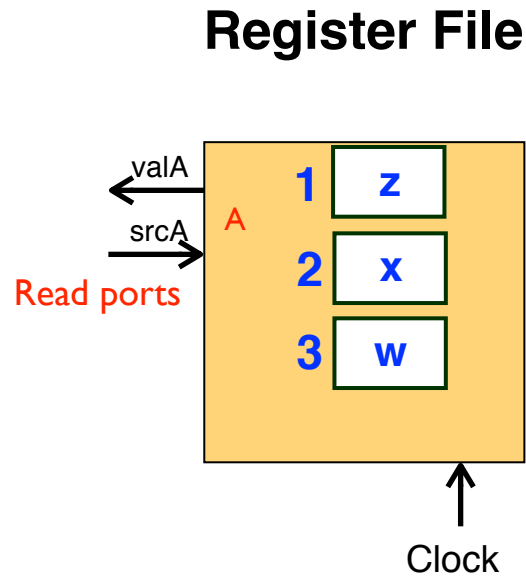
# Register File

## Register File



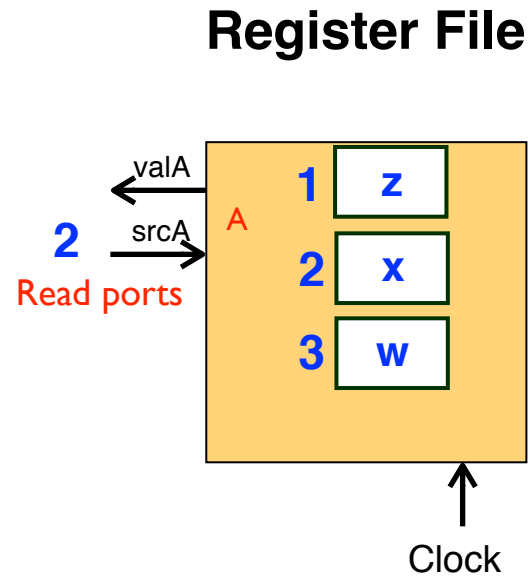
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



- Stores multiple registers of data
  - Address input specifies which register to read or write

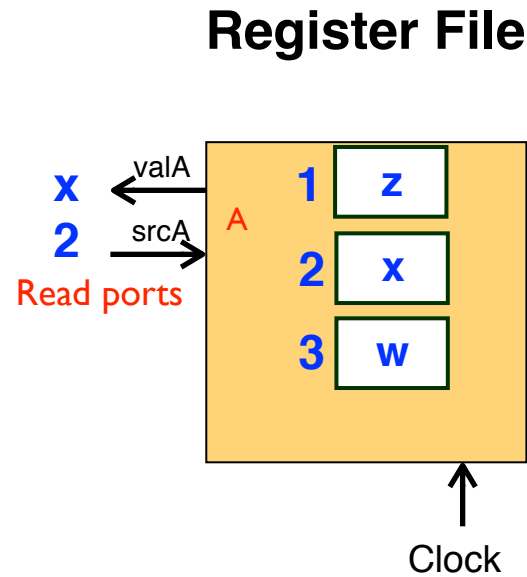
# Register File



- Stores multiple registers of data
  - Address input specifies which register to read or write

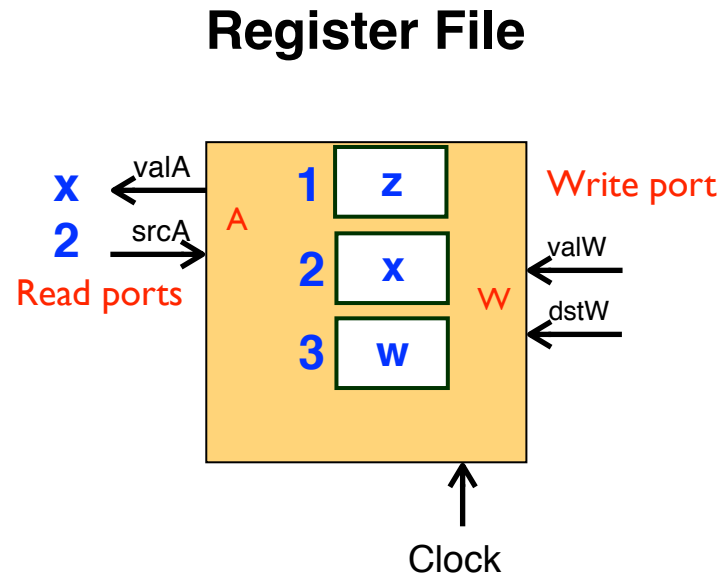


# Register File



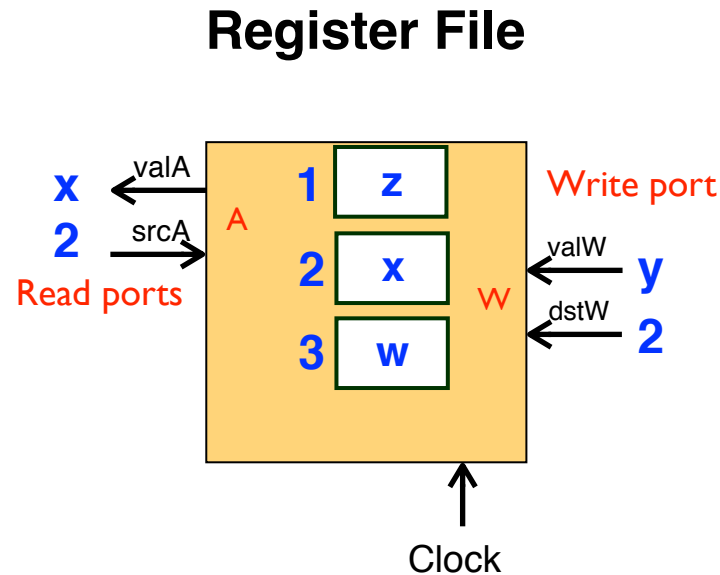
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



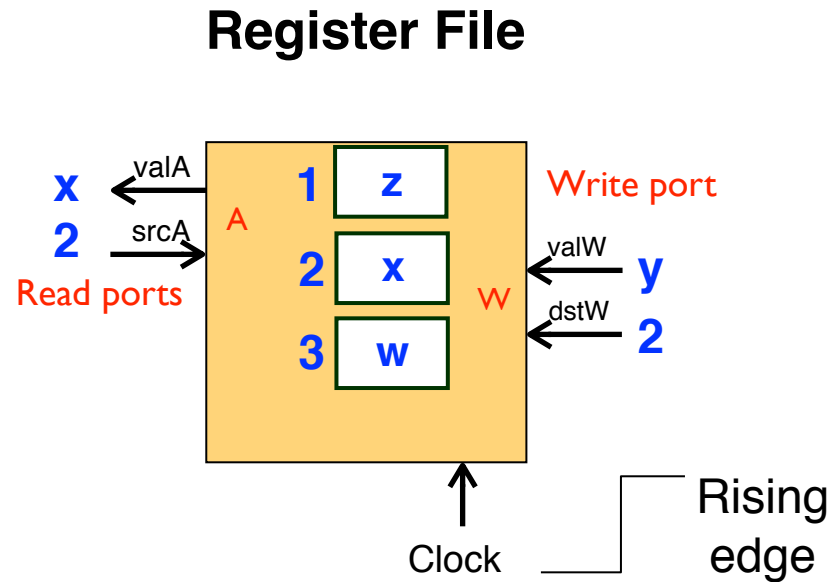
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



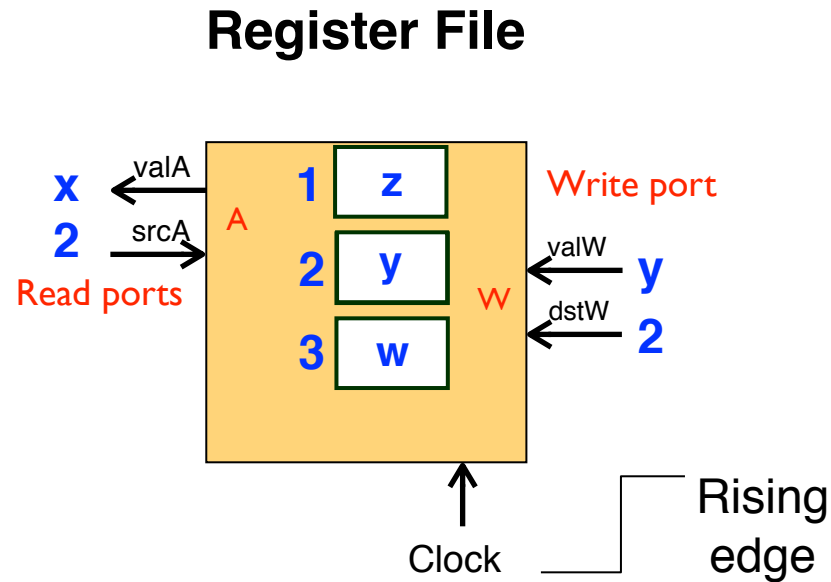
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



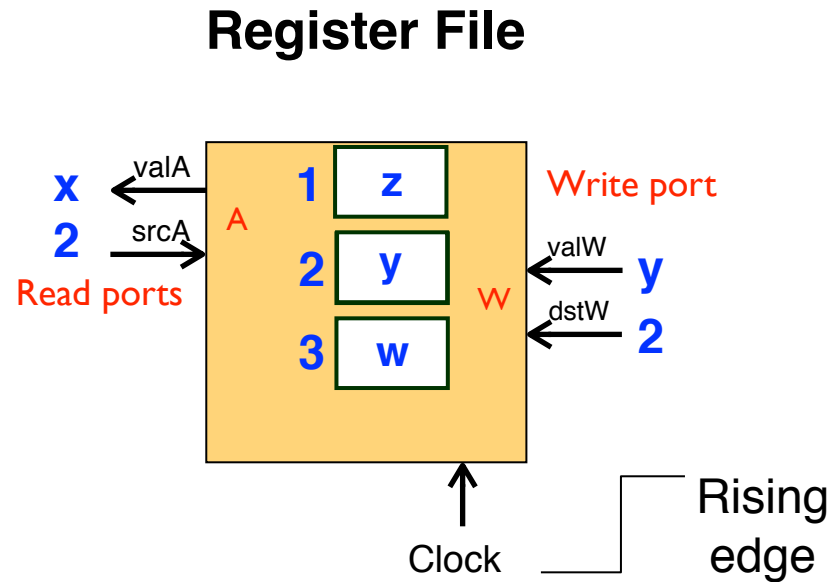
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



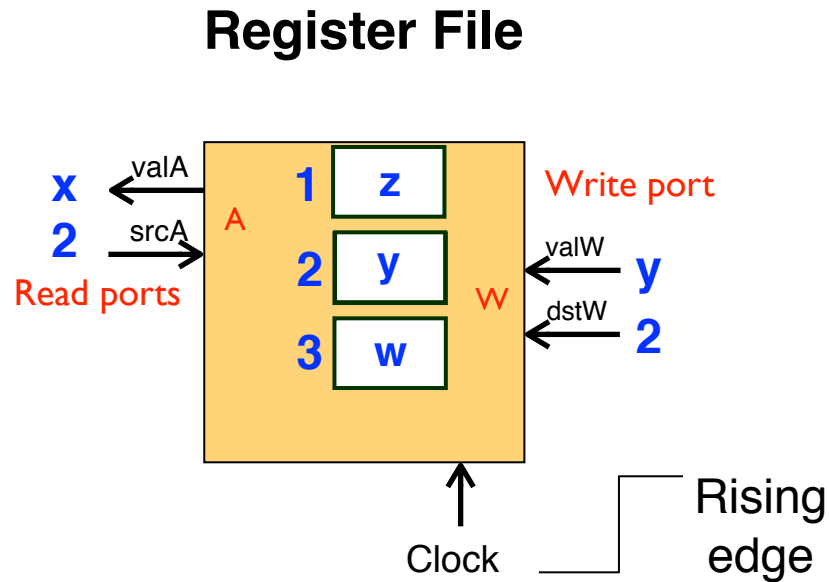
- Stores multiple registers of data
  - Address input specifies which register to read or write

# Register File



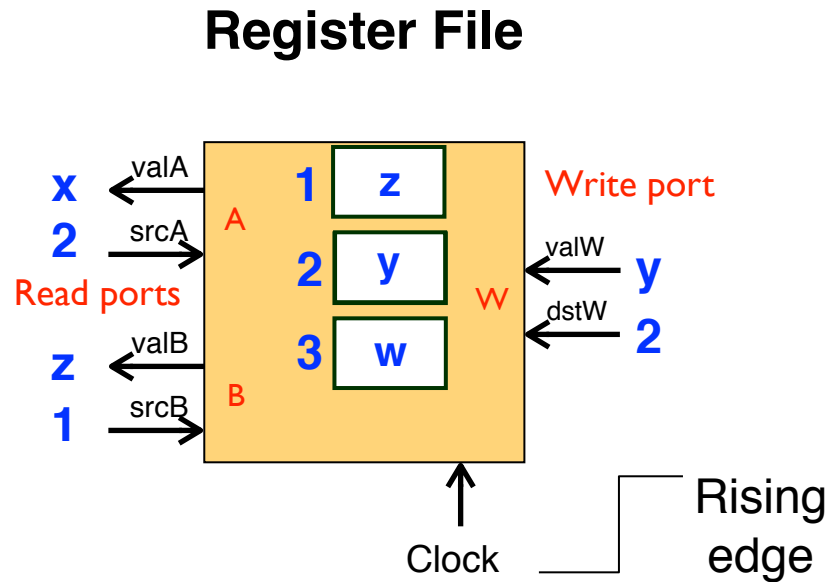
- Stores multiple registers of data
  - Address input specifies which register to read or write
- Register file is a form of **Random-Access Memory (RAM)**

# Register File



- Stores multiple registers of data
  - Address input specifies which register to read or write
- Register file is a form of **Random-Access Memory (RAM)**
- Multiple Ports: Can read and/or write multiple words in one cycle. Each port has separate address and data input/output

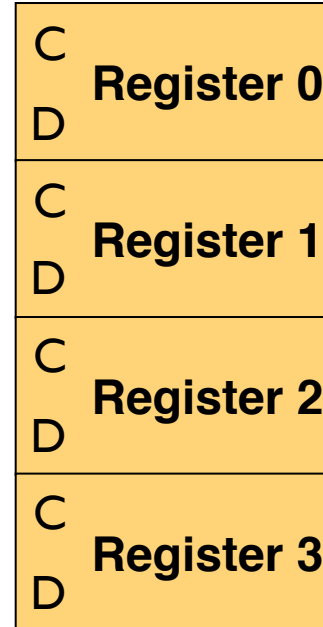
# Register File



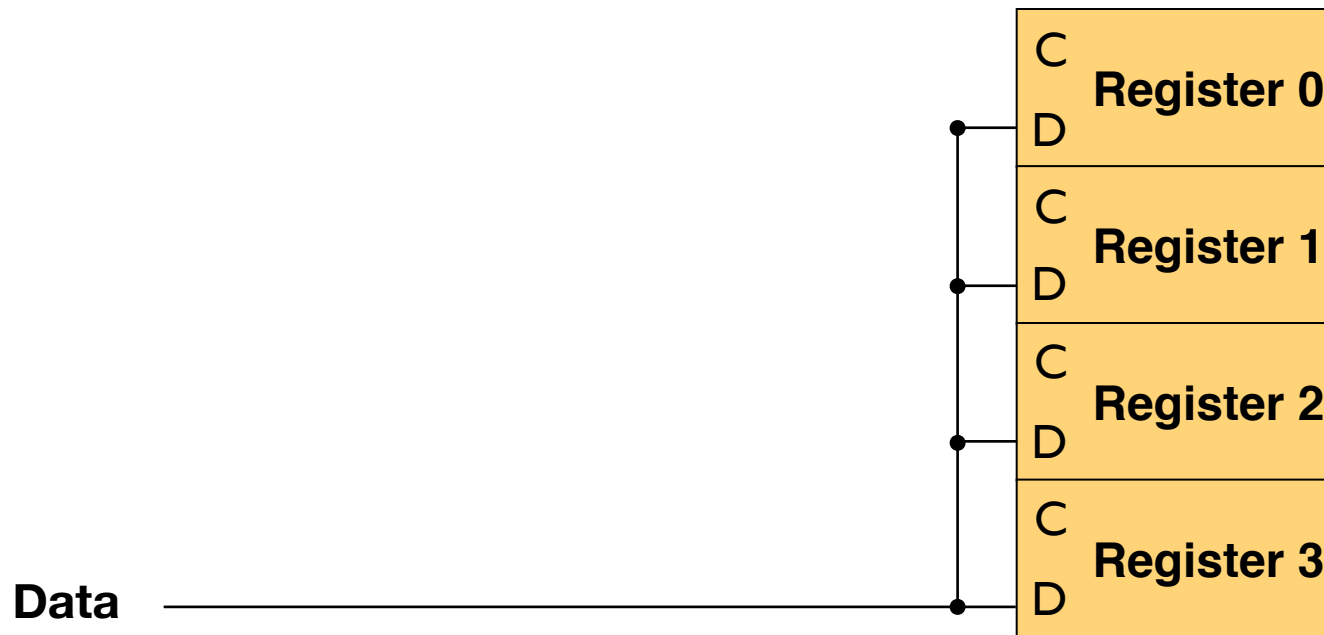
- Stores multiple registers of data
  - Address input specifies which register to read or write
- Register file is a form of **Random-Access Memory (RAM)**
- Multiple Ports: Can read and/or write multiple words in one cycle. Each port has separate address and data input/output



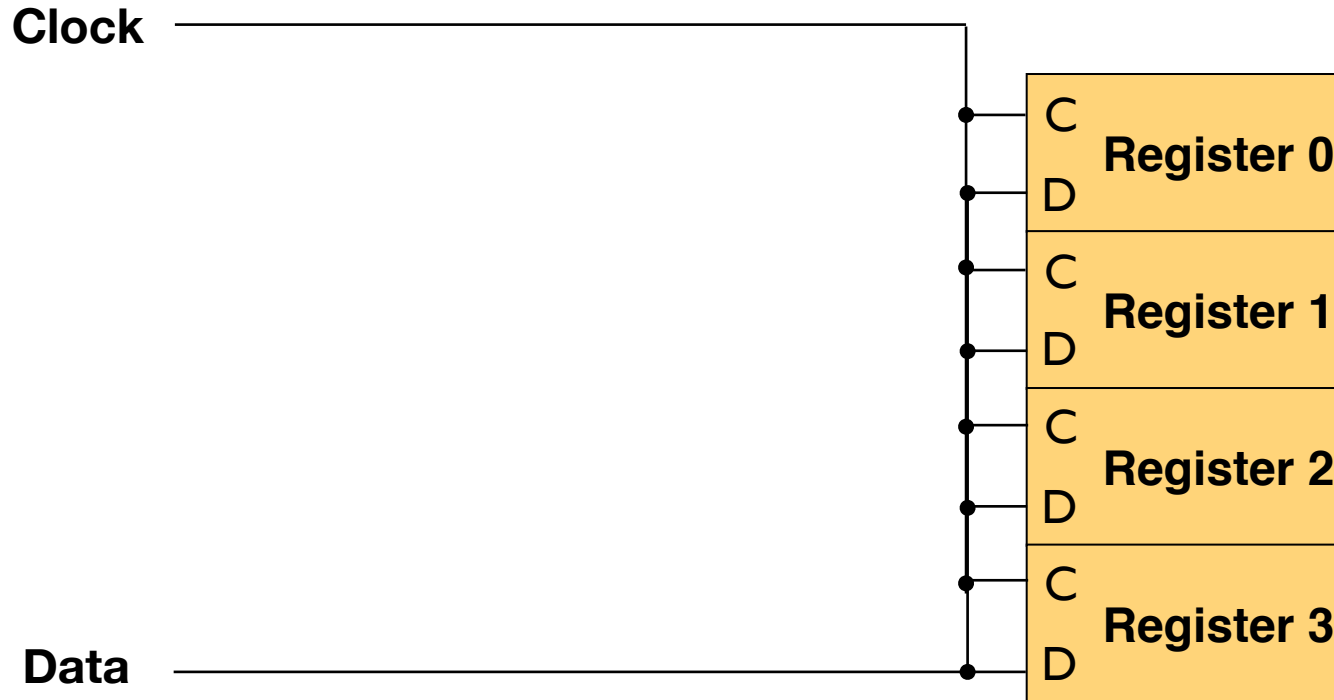
# Register File Implementation



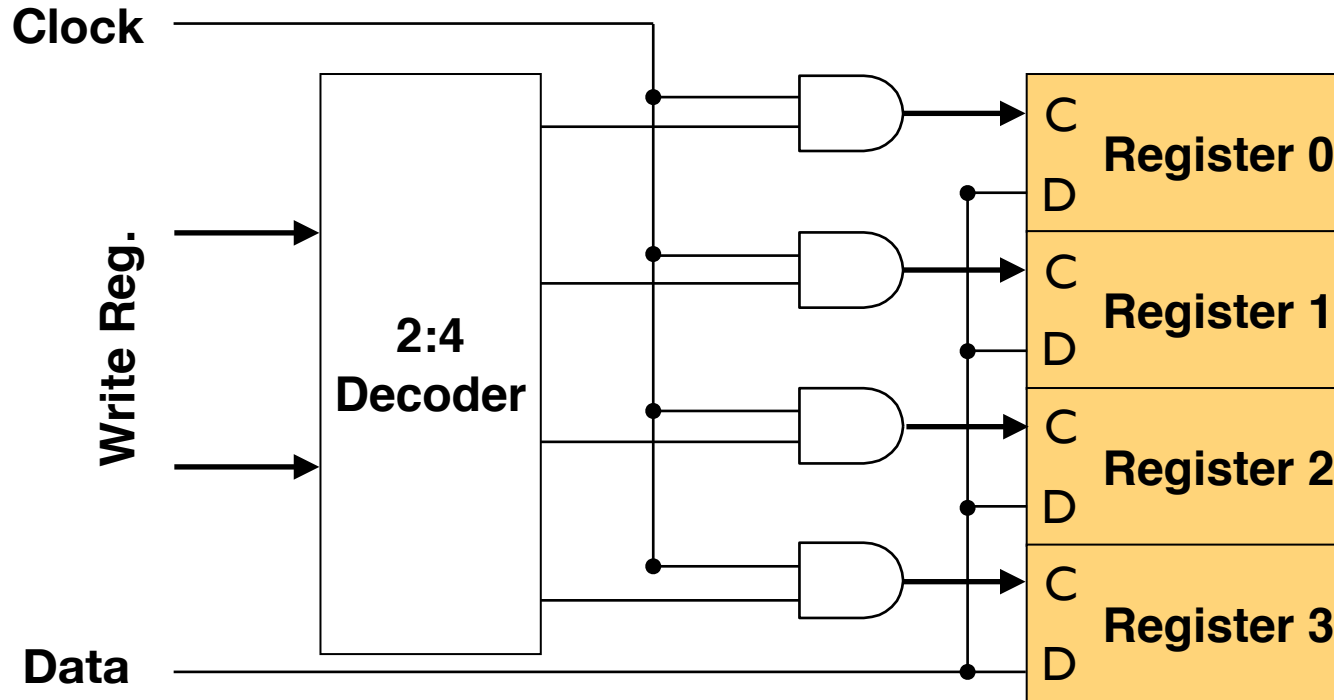
# Register File Implementation



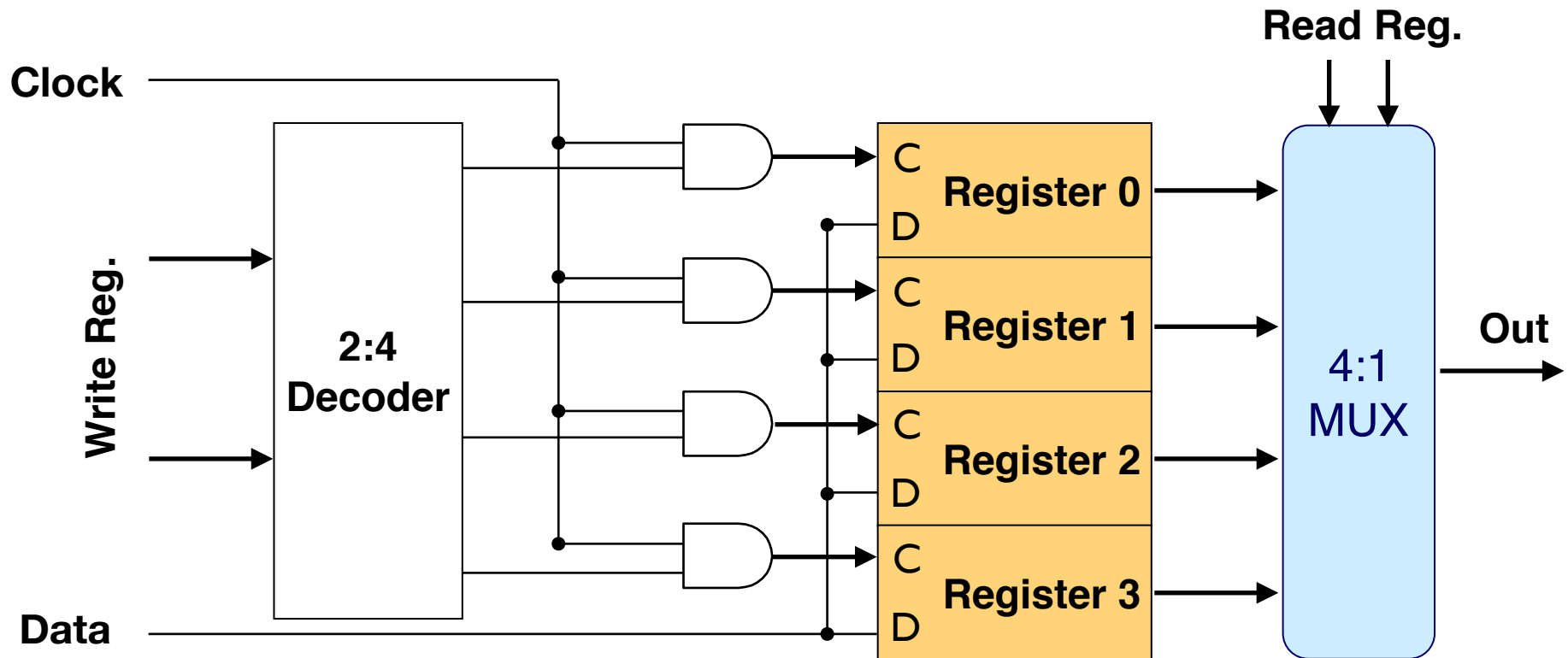
# Register File Implementation



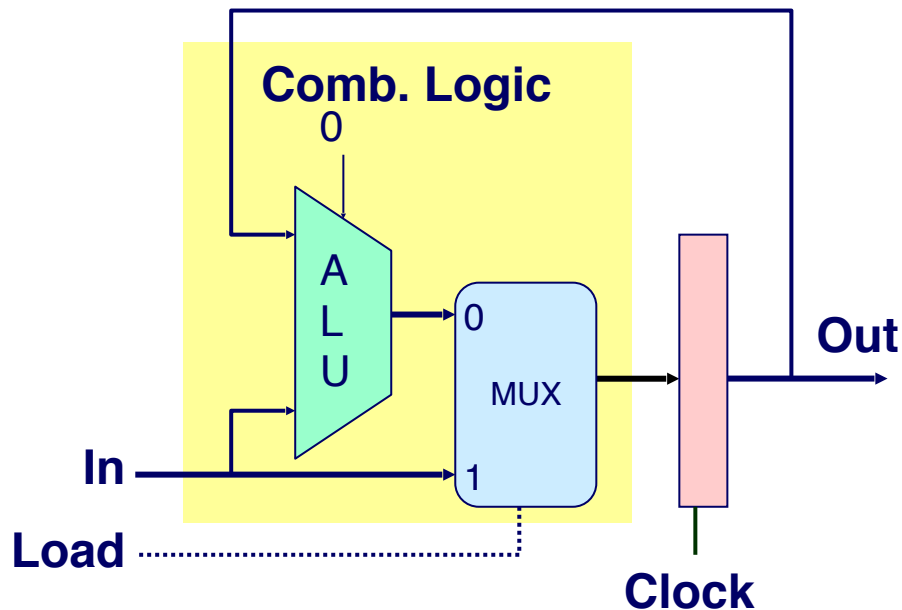
# Register File Implementation



# Register File Implementation

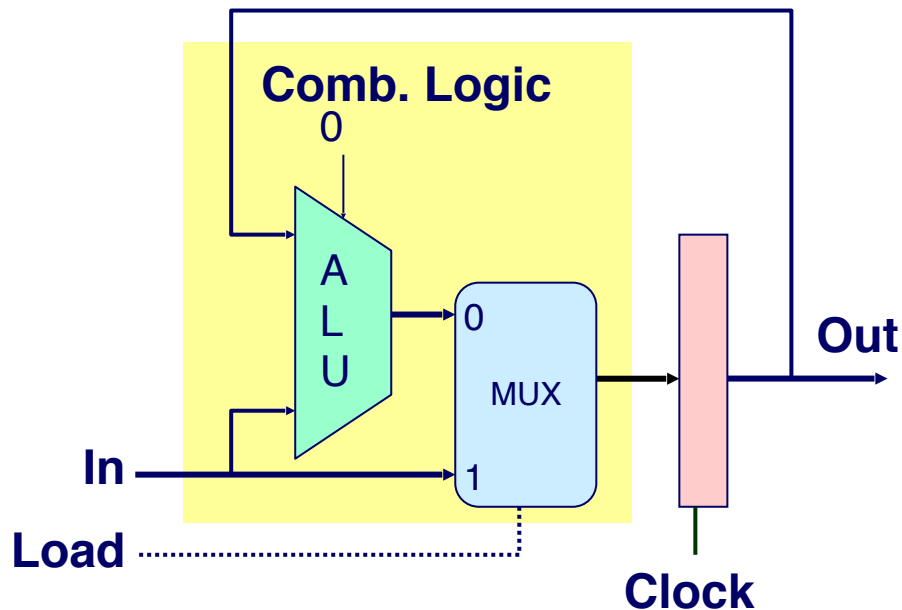


# Putting It Together: Accumulator Example



- Accumulator circuit
- Load or accumulate on each cycle

# Putting It Together: Accumulator Example



- Accumulator circuit
- Load or accumulate on each cycle

