University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

HW Constructs

References:

1) Textbook
2) Mark Redekopp's slide series

Shahin Nazarian

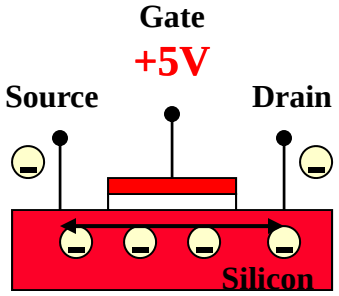
Spring 2010

Logic Circuits

- Combinational logic
  - Perform a specific function (mapping of 2^n input combinations to desired output combinations)
  - No internal state or feedback
    - Given a set of inputs, we will always get the same output after some time (propagation) delay
- Sequential logic ("Storage" devices)
  - Registers made up of flip-flops/latches are the fundamental building blocks
    - Controlled by a "clock" signal
    - Sample data on a "clock" edge and remember that value until the next edge

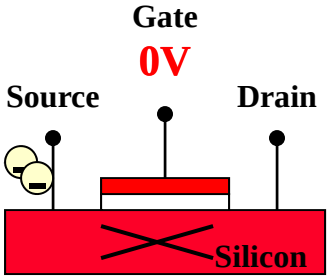
Transistor Review

- 3-terminal device
  - Gate input: the control input; it's voltage determines whether current can flow
  - Source & Drain: terminals that current flows from/to



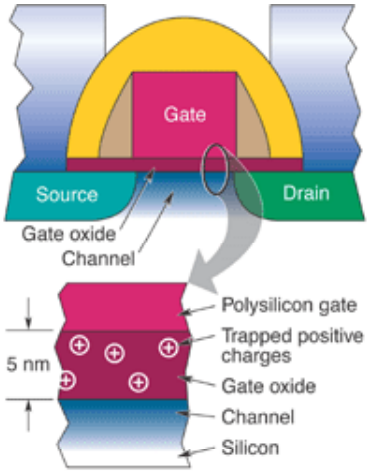
Transistor is 'on'

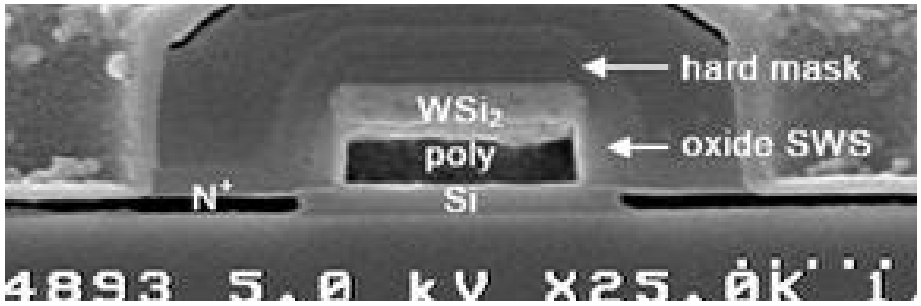
High voltage at gate allows current to flow from source to drain



Transistor is 'off'

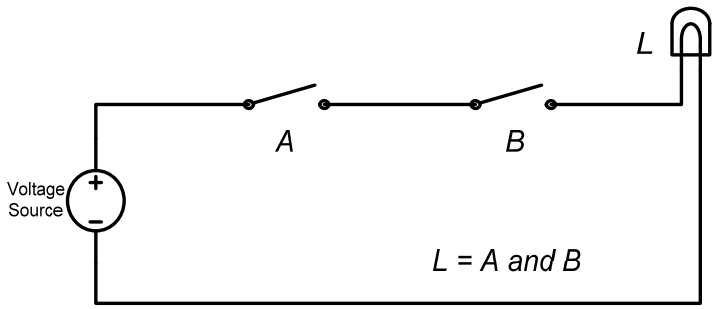
Low voltage at gate prevents current from flowing from source to drain



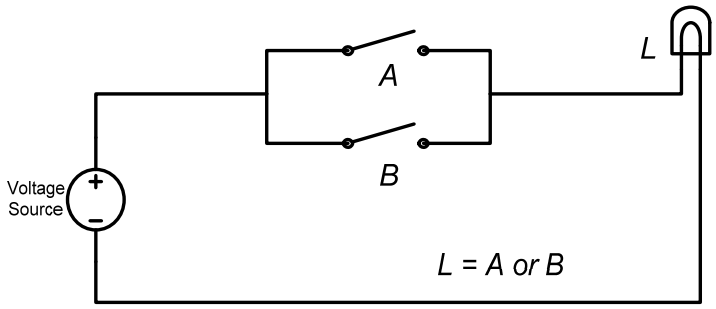


Fundamental Blocks: Logic Gates

- All digital circuits are built from transistors (voltage controlled switches) which can be on or off
- The arrangement of transistors leads to a few basic functions that express logical operations
  - These constructs are known as gates



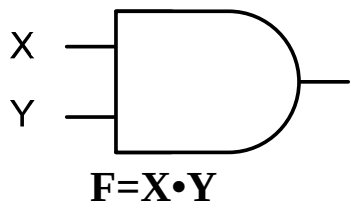
Transistors in SERIES = AND



Transistors in PARALLEL = OR

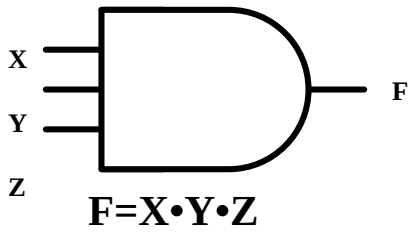
AND Gates

- An AND gate outputs a '1' (true) if ALL inputs are '1' (true)
- Gates can have several inputs
- Behavior can be shown in a truth table (listing all possible input combinations and the corresponding output)



X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

2-input AND

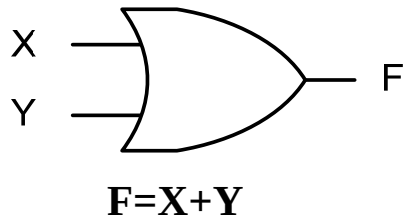


X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3-input AND

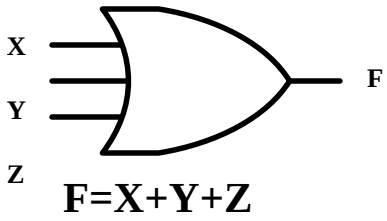
OR Gates

- An OR gate outputs a '1' (true) if ANY input is '1' (true)
- Gates can also have several inputs



X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

2-input OR

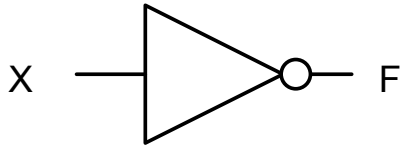


X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

3-input OR

NOT Gate

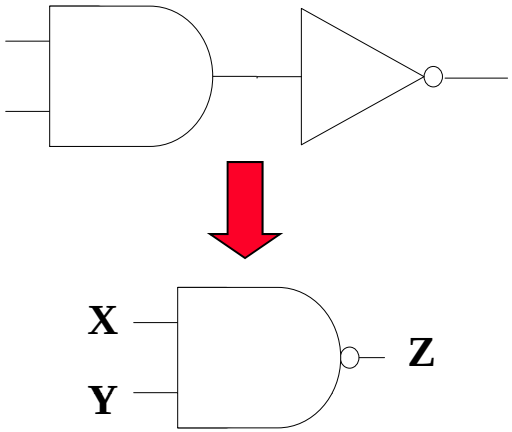
- A NOT gate outputs a '1' (true) if the input is '0' (false)



X	F
1	0
0	1

$F = \overline{X}$

NAND and NOR Gates



NAND

$$Z = \overline{X \cdot Y}$$

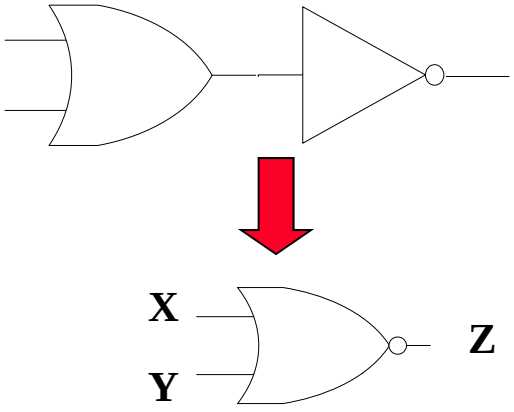
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

AND

X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

NAND

True if NOT ALL inputs are true



NOR

$$Z = \overline{X + Y}$$

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

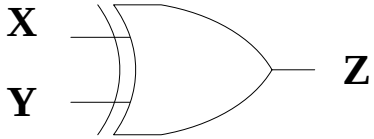
OR

X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

NOR

True if NOT ANY input is true

XOR and XNOR Gates

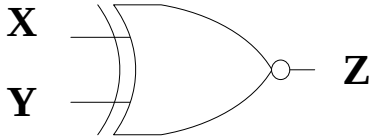


XOR

$$Z = X \oplus Y$$

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

True if an odd # of inputs are true
= True if inputs are different



XNOR

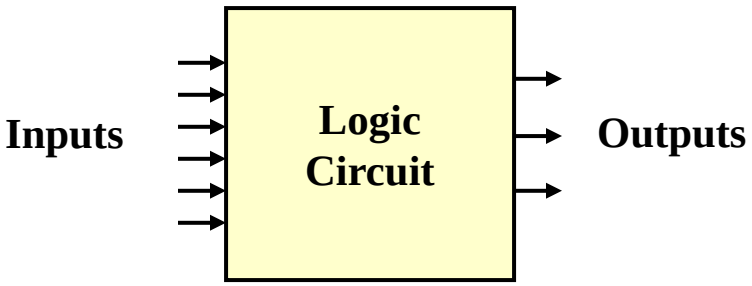
$$Z = \overline{X \oplus Y}$$

X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

True if an even # of inputs are true
= True if inputs are same

Logic Functions

- Map input combinations of n-bits to desired m-bit output
- Can describe function with a truth table and then find its circuit implementation

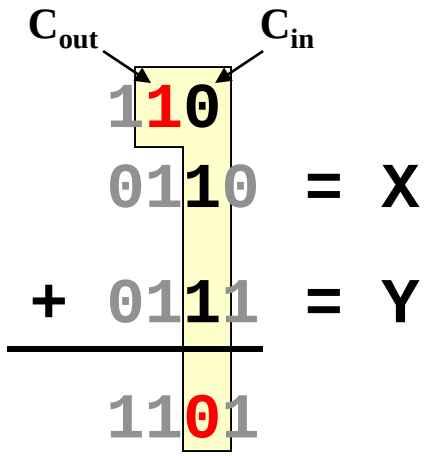


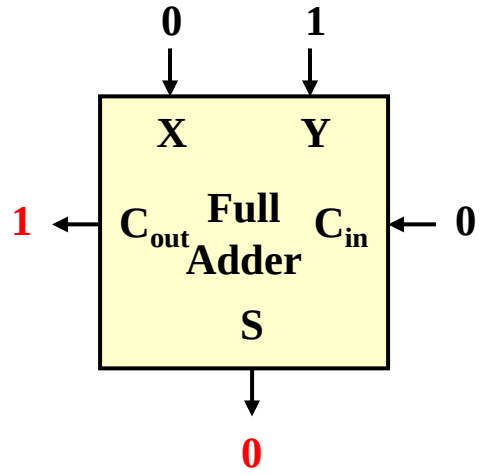
IN0	IN1	IN2	OUT0	OUT1
0	0	0	0	1
0	0	1	1	1
	…			
1	1	1	0	0

Functions & Logic Networks

- We can generate arbitrary logic functions using multiple levels of logic
- Ex: Full Adder performs one column of addition of two numbers
- Adds a bit from each number plus a carry-in from the previous column and generates a sum bit & carry-out

X	Y	C_{in}	C_{out}	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

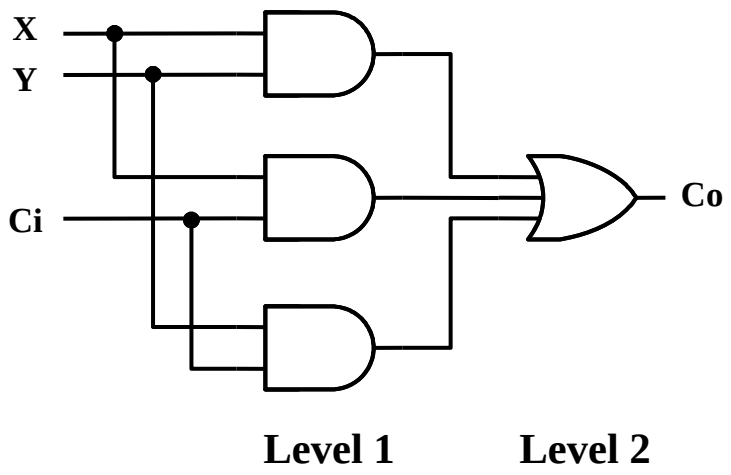




Functions & Logic Networks

- Find patterns in the input combinations that uniquely identify the rows where the output is 1.

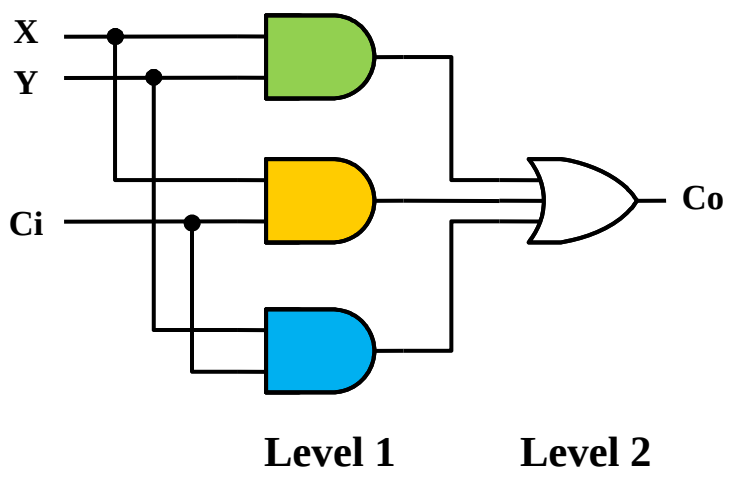
X	Y	C_i	C_o
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**



Functions & Logic Networks

- Decompose function into smaller sub-functions that can easily be implemented. Then recombine sub-functions to create overall function

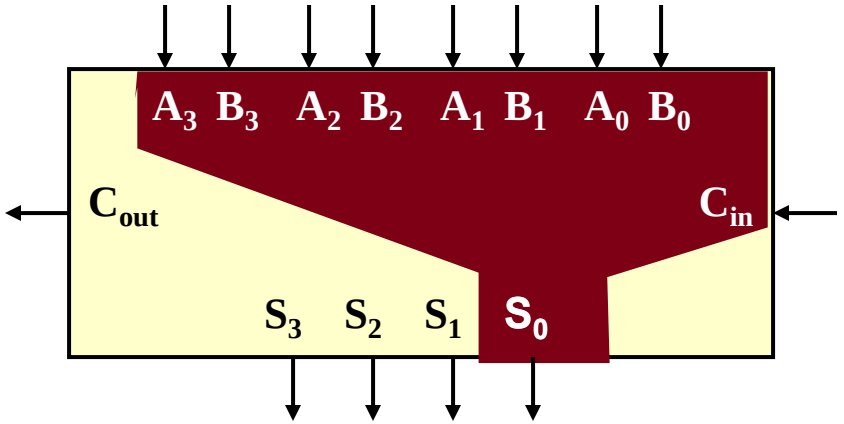
X	Y	C_i	C_o	X·Y	X·Ci	Y·Ci
0	0	0	**0**	**0**	**0**	**0**
0	0	1	**0**	**0**	**0**	**0**
0	1	0	**0**	**0**	**0**	**0**
0	1	1	**1**	**0**	**0**	**1**
1	0	0	**0**	**0**	**0**	**0**
1	0	1	**1**	**0**	**1**	**0**
1	1	0	**1**	**1**	**0**	**0**
1	1	1	**1**	**1**	**1**	**1**



Functions & Logic Networks

- Functions with multiple outputs can be generated by considering each output as a separate function of the inputs
- Example: A dedicated 4-bit adder

$$
\begin{array}{rl}
A_3A_2A_1A_0 & = A \\
+\ B_3B_2B_1B_0 & = B \\
\hline
S_4S_3S_2S_1S_0 & = S
\end{array}
$$

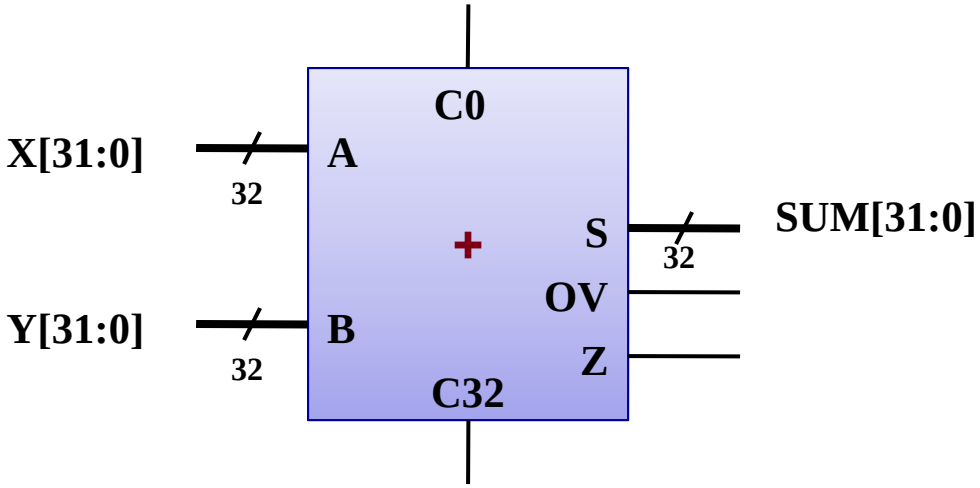


Combinational Functional Blocks

- We can take our logic gates and build up a set of commonly used functional blocks
- For this class, the common functional blocks include
  - Adders
  - Multiplexers
  - ALU (Arithmetic and Logic Units)

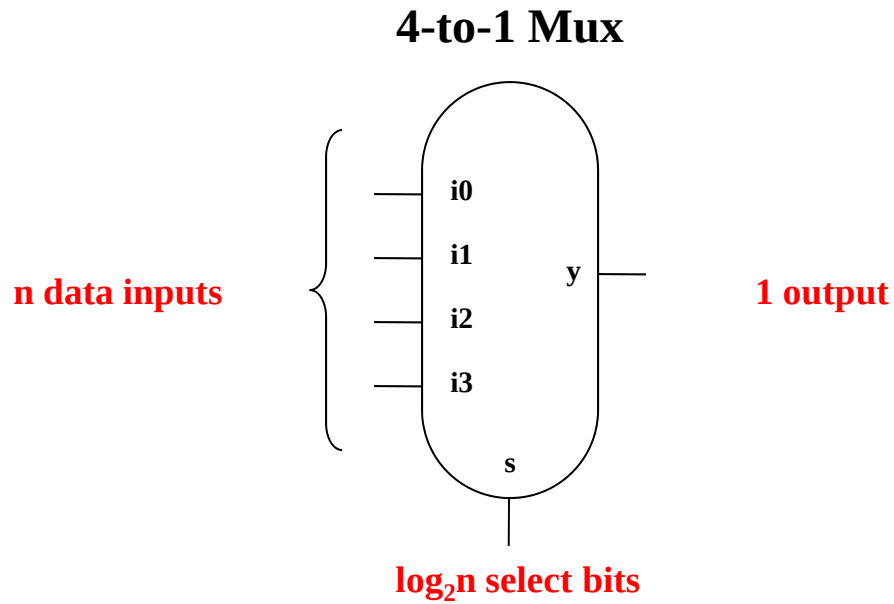
Adders

- Take two numbers as input and produce the binary sum

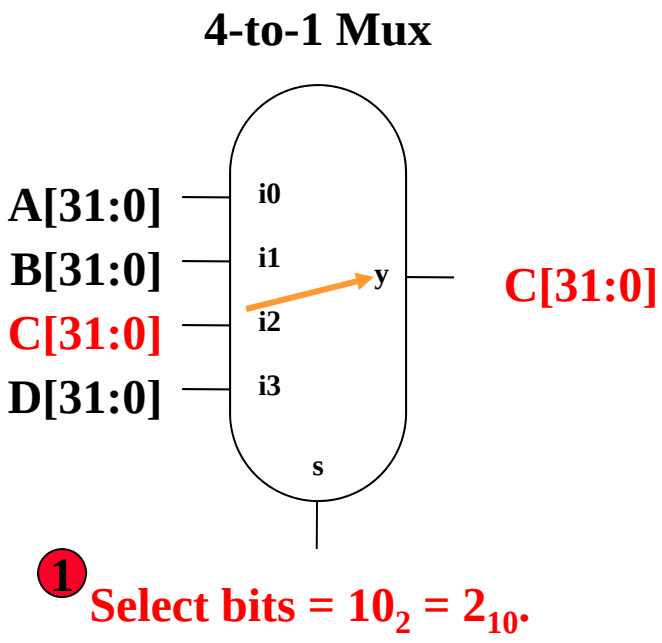


Multiplexers

- Along with adders, multiplexers are most used building block
- n data inputs, $\log_2 n$ select bits, 1 output
- A multiplexer ("mux" for short) selects one data input and passes it to the output



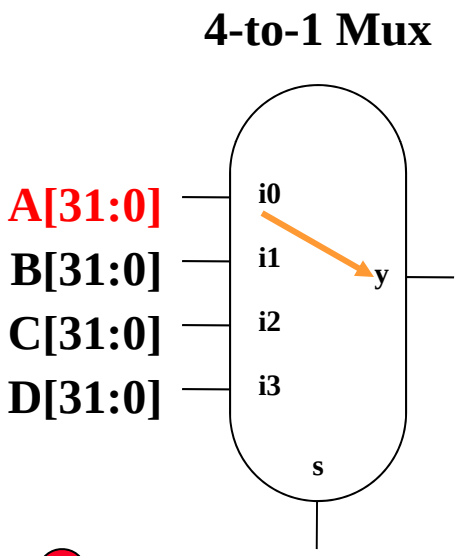
Multiplexers



2 Thus, input 2 = C[31:0] is selected and passed to the output

1 Select bits = $10_2 = 2_{10}$.

Multiplexers



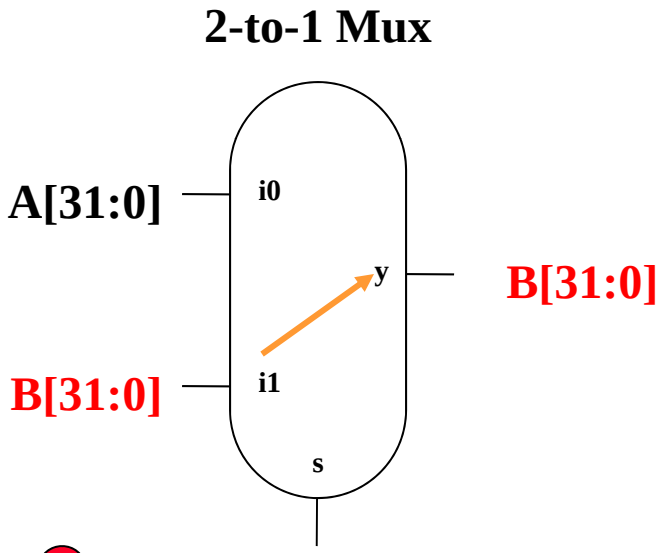
2

Thus, input 0 = A[31:0] is selected and passed to the output

1

Select bits = $00_2 = 0_{10}$.

Multiplexers

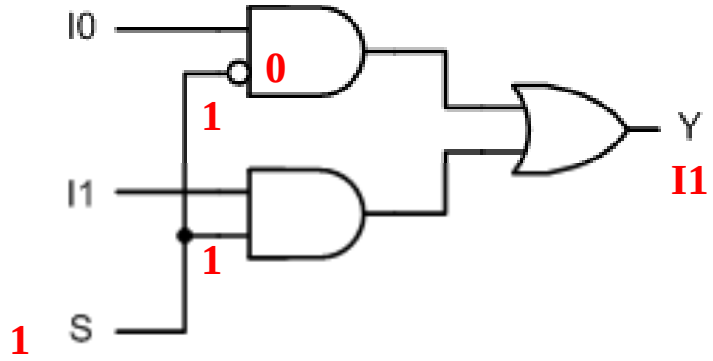


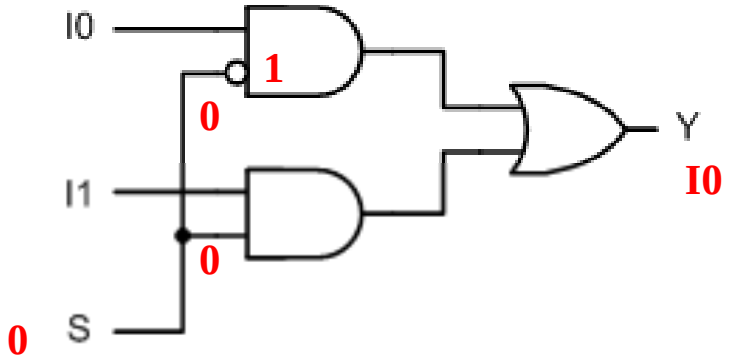
2 Thus, input 1 = B[31:0] is selected and passed to the output

1 Select bits = $1_2 = 1_{10}$.

Building a Mux

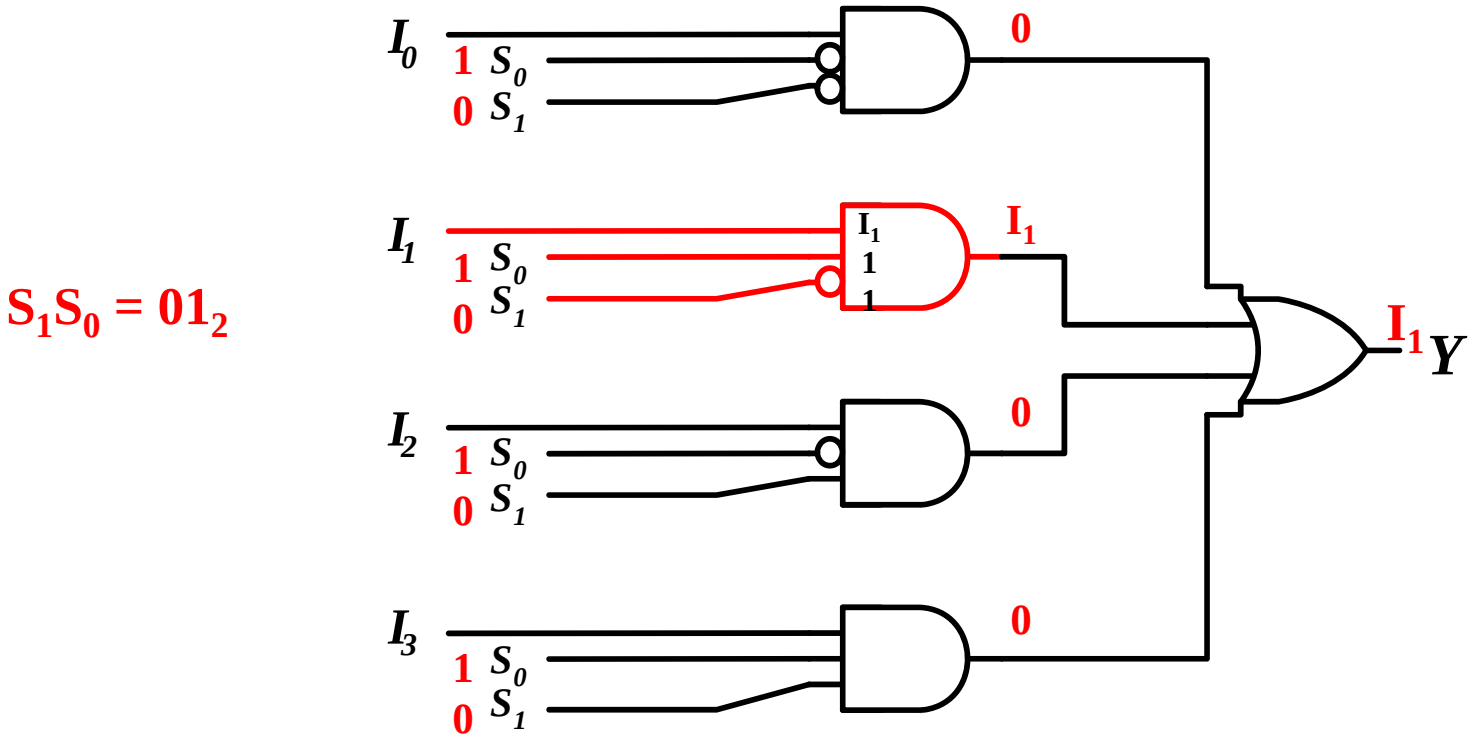
- To build a mux
  - Decode the select bits and include the corresponding data input
  - Finally OR all the first level outputs together





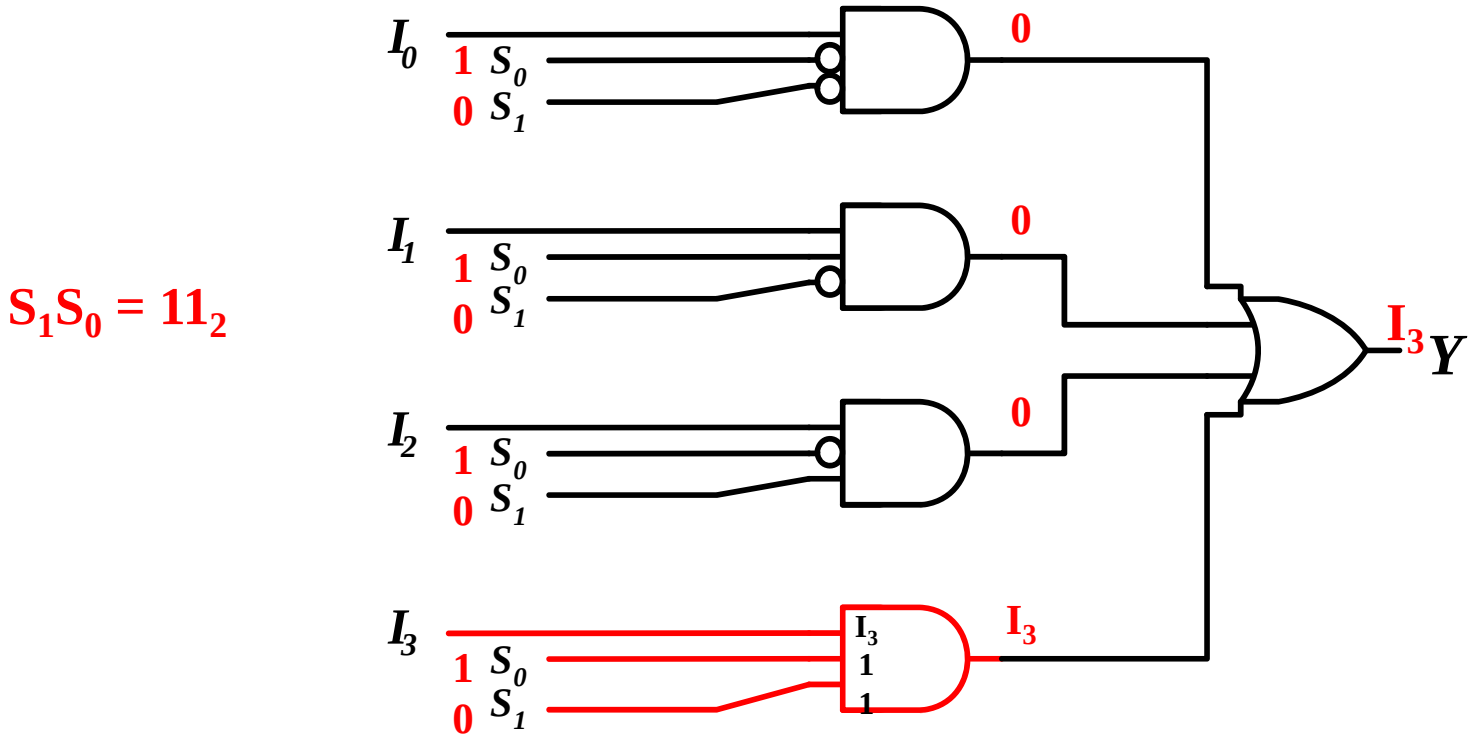
Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input
  - Finally OR all the first level outputs together



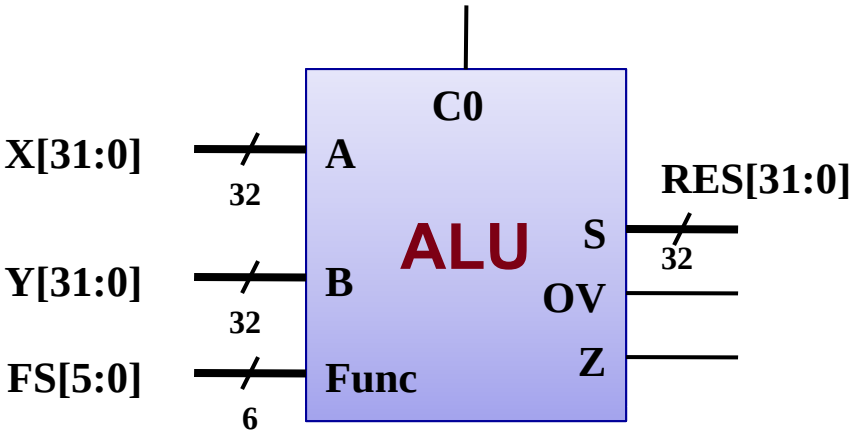
Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input
  - Finally OR all the first level outputs together



ALU's

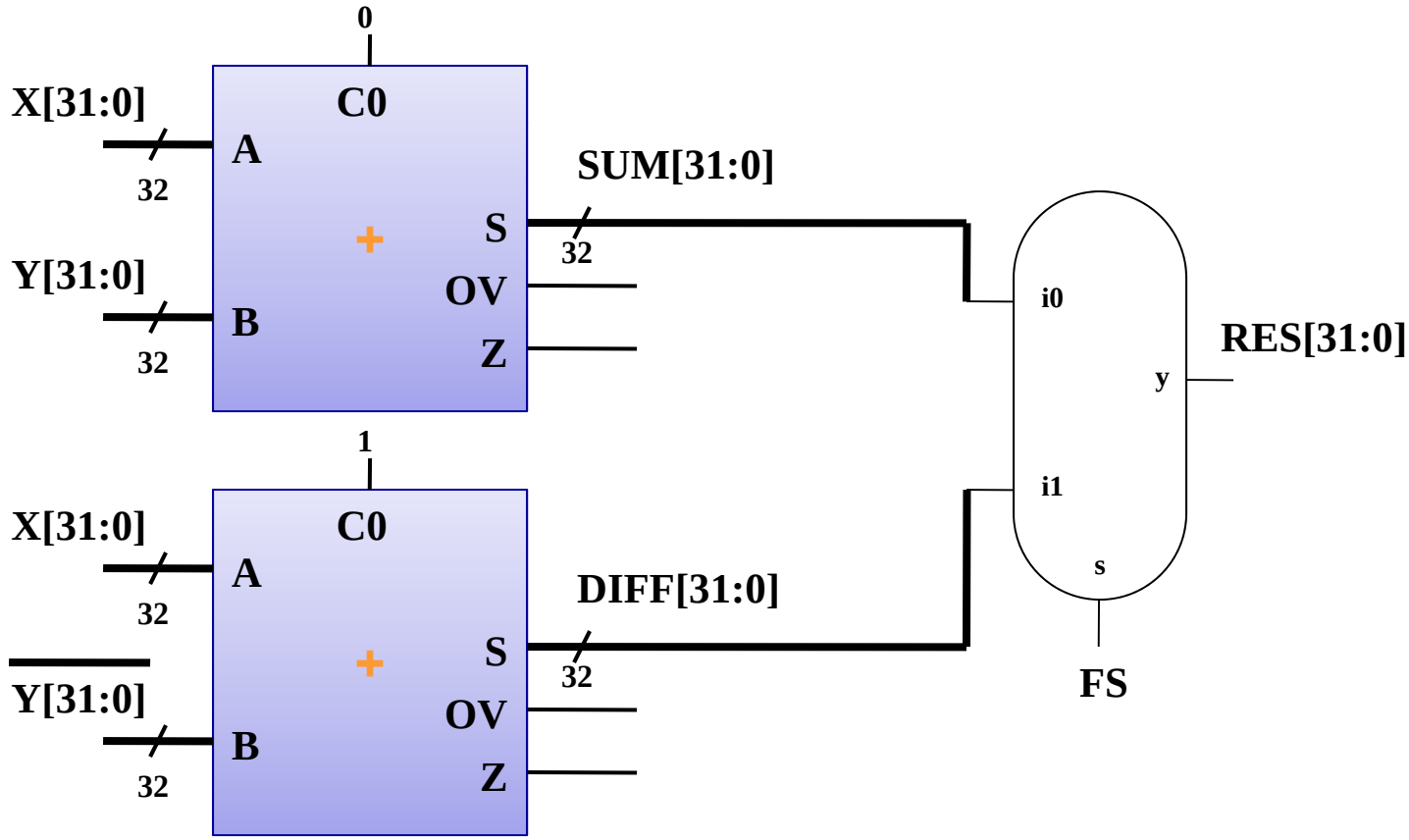
- Perform a selected operation on two input numbers
  - FS[5:0] select the desired operation



Func. Code	Op.	Func. Code	Op.
00_0000	A SLL B	10_0000	A+B
00_0010	A SRL B	10_0010	A-B
00_0011	A SRA B	…	…
…	…	10_0100	A AND B
01_1000	A * B	10_0101	A OR B
01_1001	A * B (uns.)	10_0110	A XOR B
01_1010	A / B	10_0111	A NOR B
01_1011	A / B (uns.)	…	…
…	…	10_1010	A SLT B

Simple ALU Design

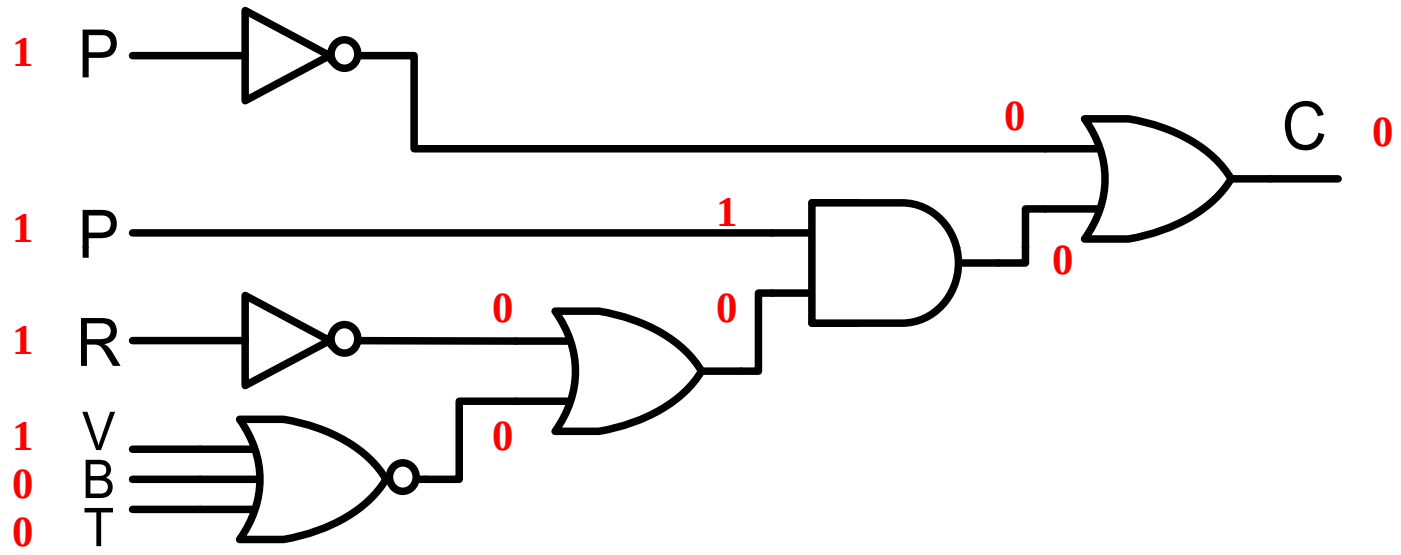
- Build an ALU to either ADD or SUB X and Y, based on an input FS (0=Add, 1=Sub)



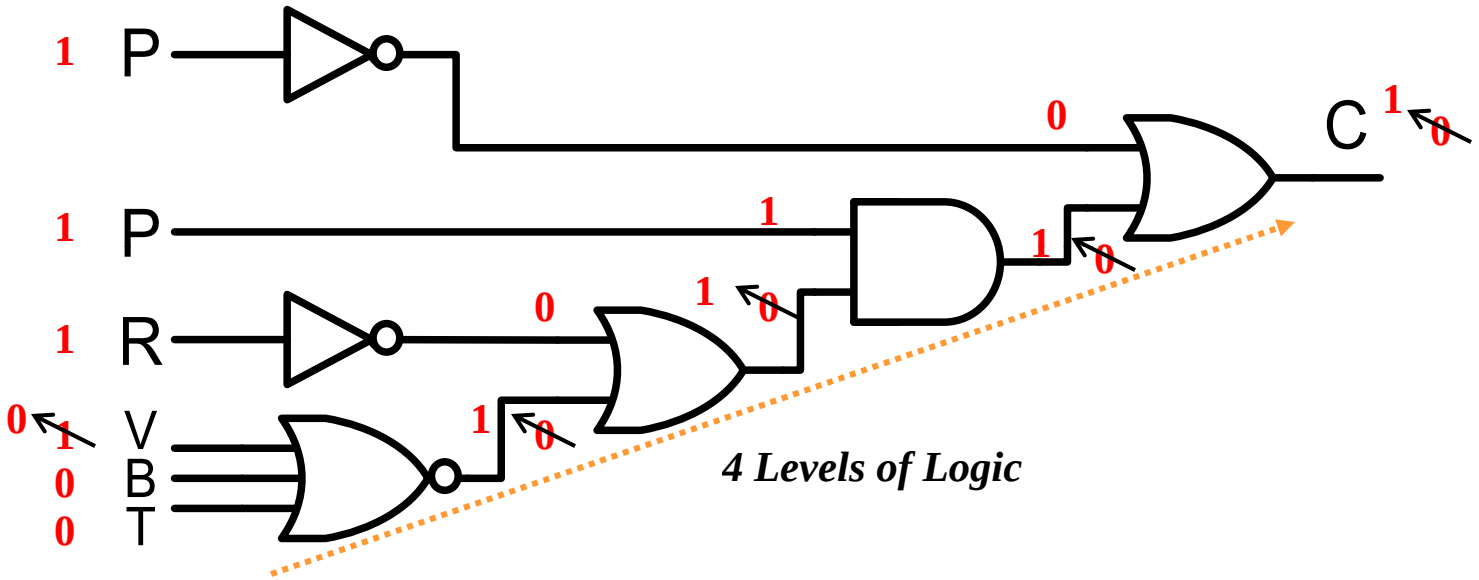
Delay in Combinational Logic

- Beyond the functionality, it is important to understand the timing of logic circuits
- Each gate (transistor) has inherent propagation delay
  - Propagation Delay = time from the instant the inputs change until the output becomes stable and correct
- Delay of a circuit is proportional to the longest path (chain) of gates from any input to the output

Delay Example

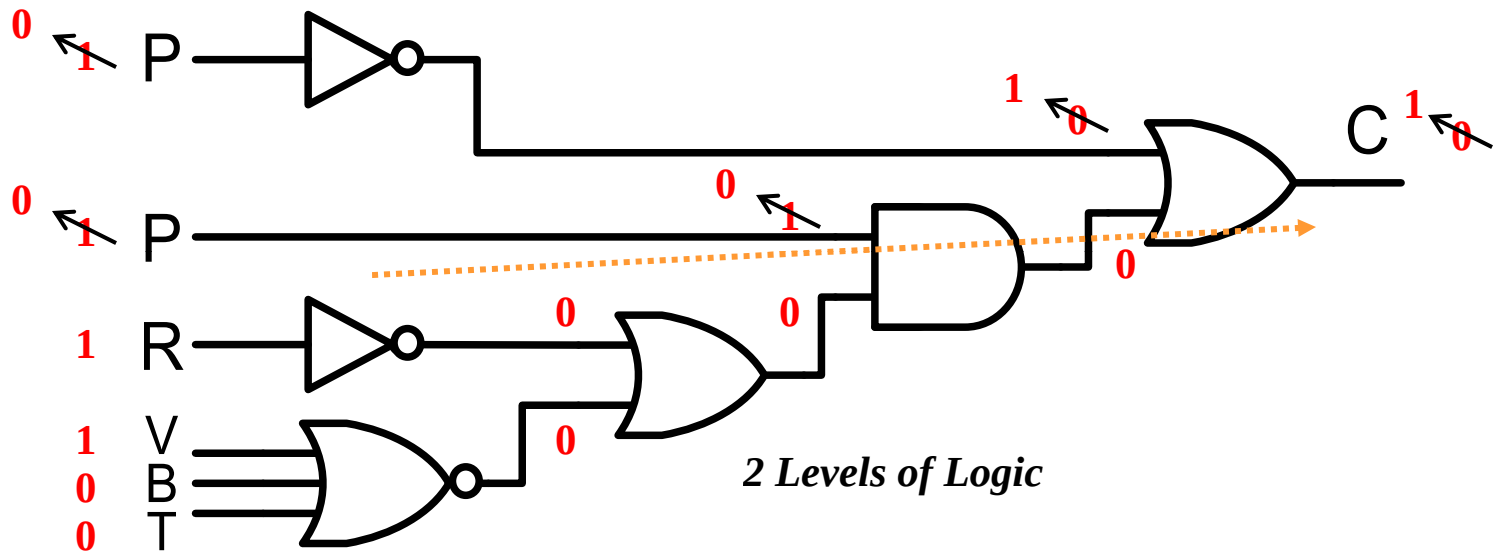


Delay Example



Change in V, B, or T must propagate through the 4 levels of logic

Delay Example



Change P must propagate through only 2 levels of logic

Timing Example

- Assume that we were adding one set of inputs and then change to a new set of inputs:

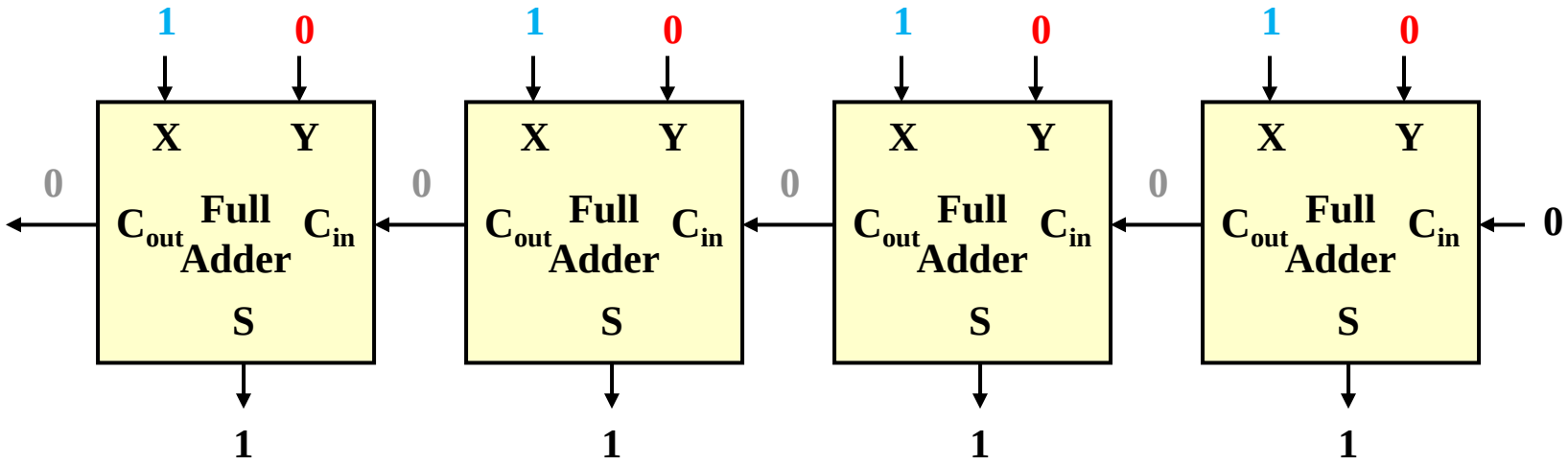
Old inputs:

```
  0000
  1111  = X
+ 0000  = Y
------
  1111
```

New inputs:

```
  1
  1111  = X
+ 0001  = Y
------
  0000
```

Old inputs:



Timing

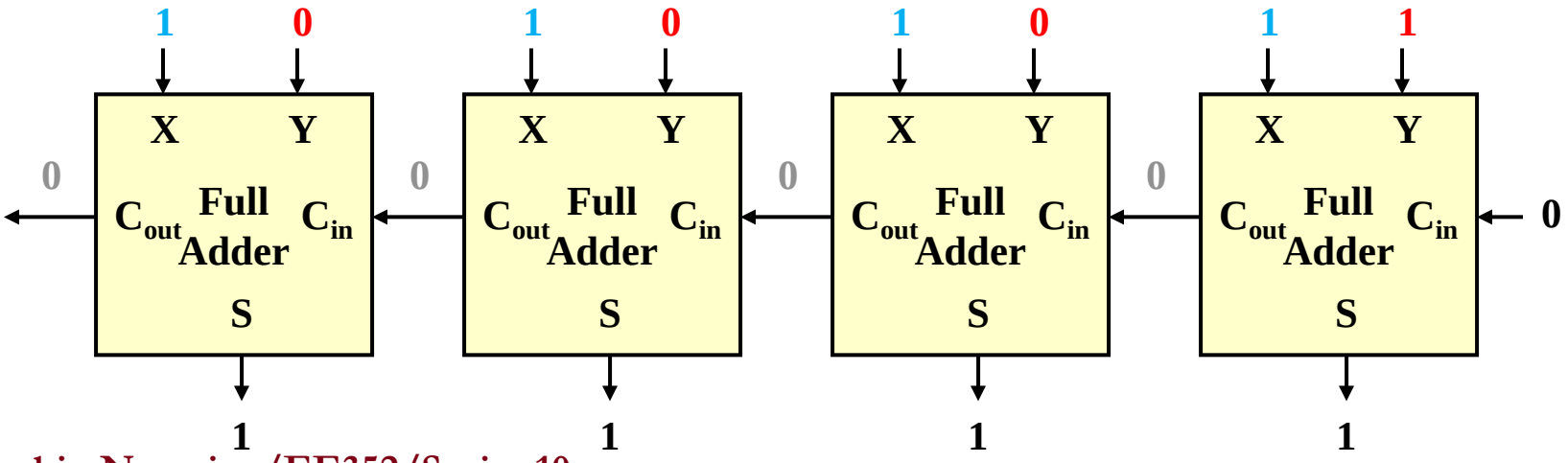
- At the time just before we enter the new input values, all carries are 0's

New inputs:

```
  1
  1111   = X
+ 0001   = Y
------
  0000
```

Time
0

Old inputs:



Timing

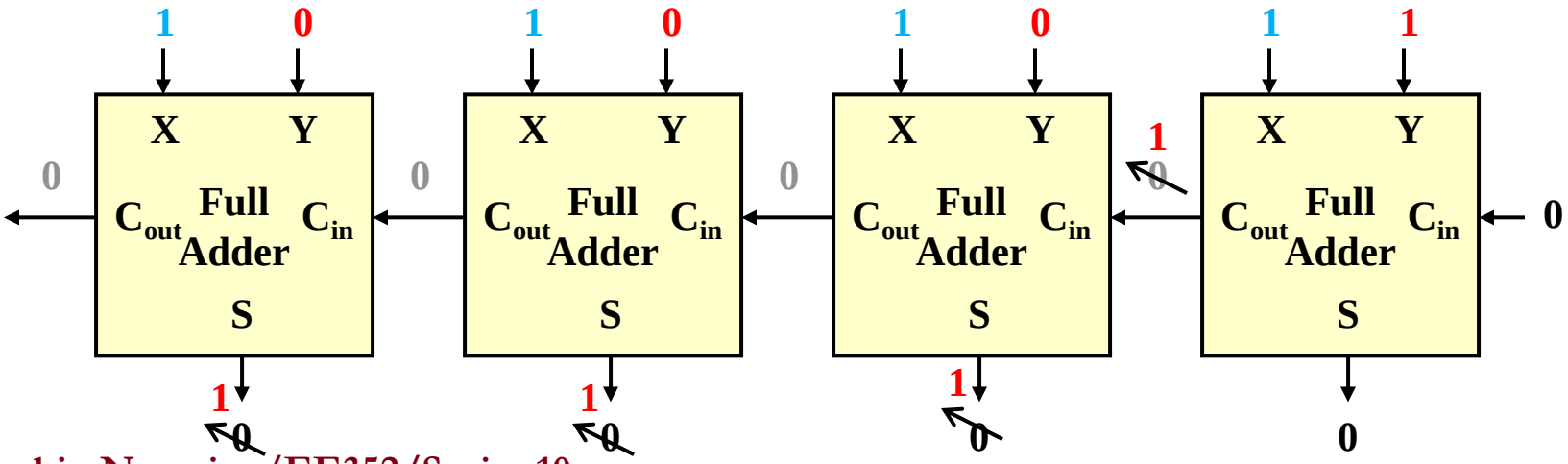
- The first adder updates his carry and sum but meanwhile other adders output "incorrect" values due to the lack of correct carries

New inputs:

```
  1
  1111   = X
+ 0001   = Y
------
  0000
```

Time
1

Old inputs:



Timing

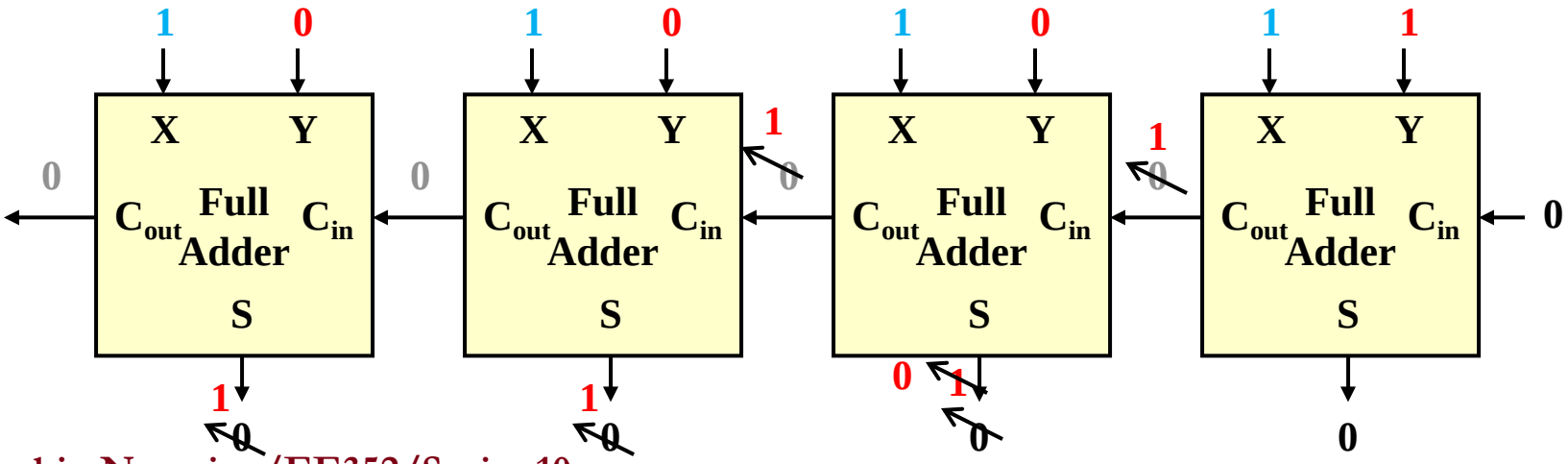
- The second adder updates his carry and sum but later adders are still incorrect

New inputs:

```
  1
    1111   = X
+   0001   = Y
-------
   0000
```

Time
2

Old inputs:



Timing

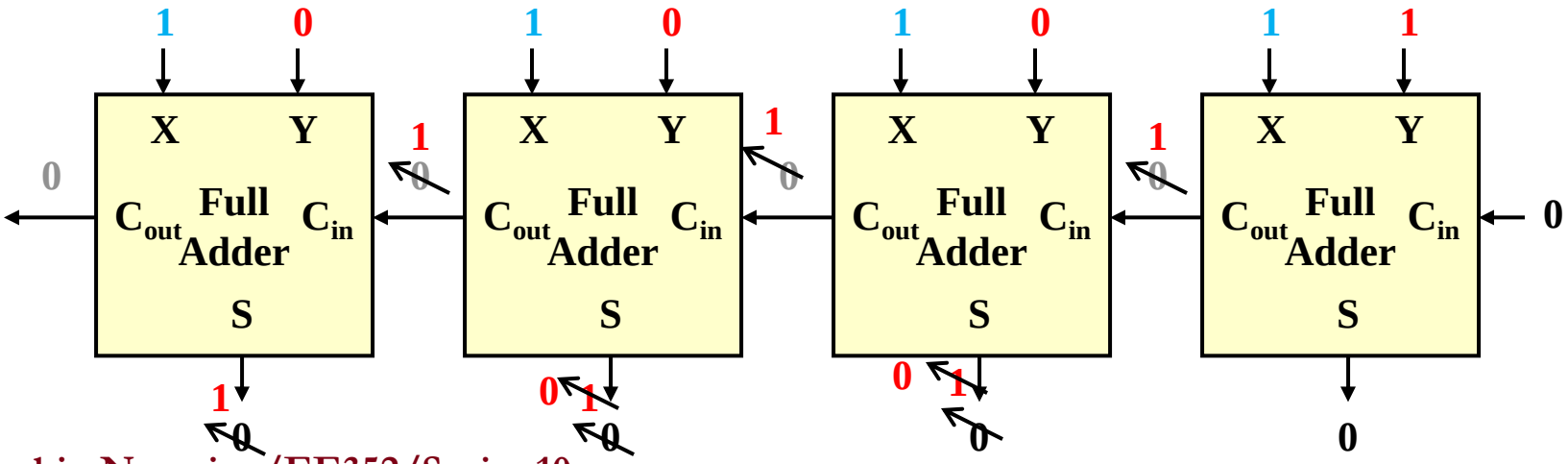
- The third adder is now correct but the last adder is still incorrect

New inputs:

```
  1
  1111   = X
+ 0001   = Y
------
  0000
```

Time
3

Old inputs:



Timing

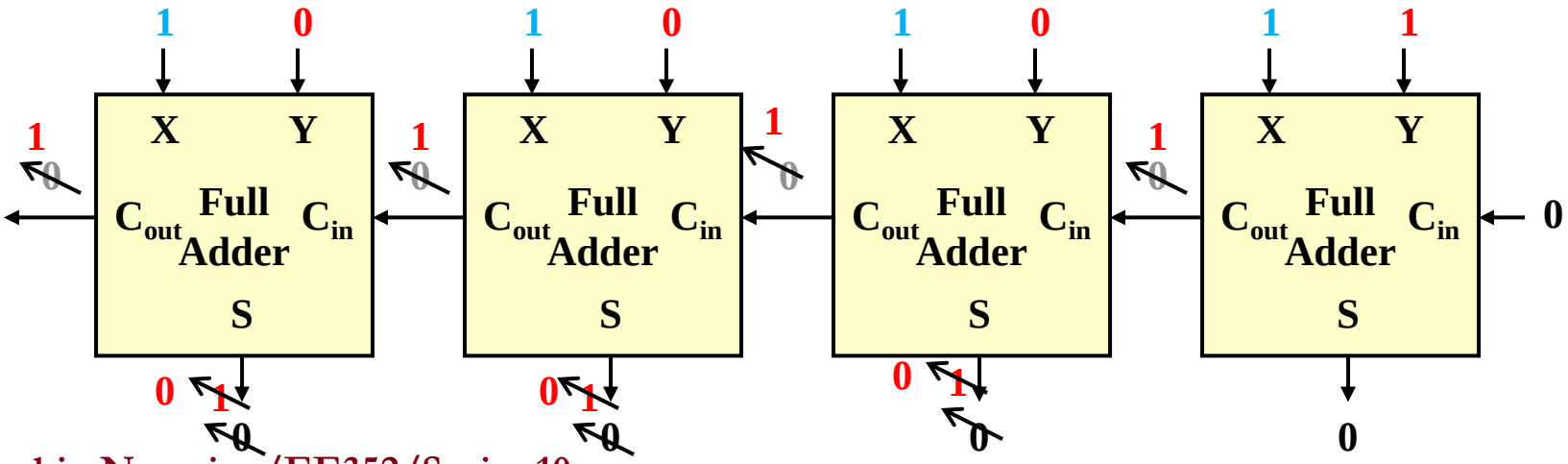
- Finally, all the adders are correct

New inputs:

```
  1
  1111   = X
+ 0001   = Y
------
  0000
```

Time
4

Old inputs:

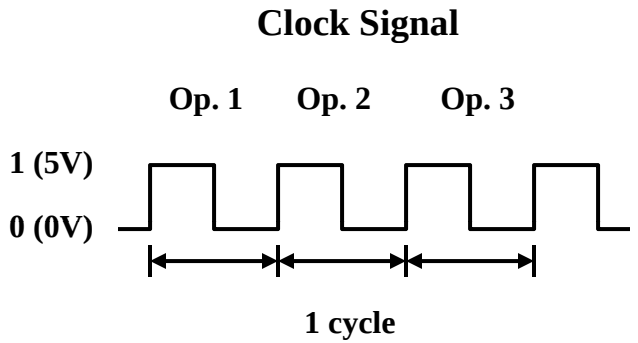


Sequential Devices (Registers)

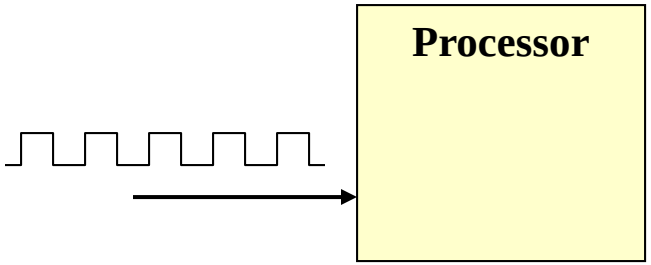
- Combinational outputs are only a function of the current inputs
  - Outputs only depend on what the inputs are right now, not one second ago
  - This implies they have no "memory" (can't remember a value)
- Sequential logic devices provide the ability to retain or "remember" a value by itself (even after the input is changed or removed)
  - Usually have a controlling signal that indicates when the device should update the value it is remembering vs. when it should simply remember that value
  - This controlling signal is usually the "clock" signal

Clock Signal

- Alternating high/low voltage pulse train
- Controls the ordering and timing of operations performed in the processor
- 1 cycle is usually measured from rising edge to rising edge
- Clock frequency = # of cycles per second (e.g. 2.8 GHz = 2.8 * 10^9 cycles per second)

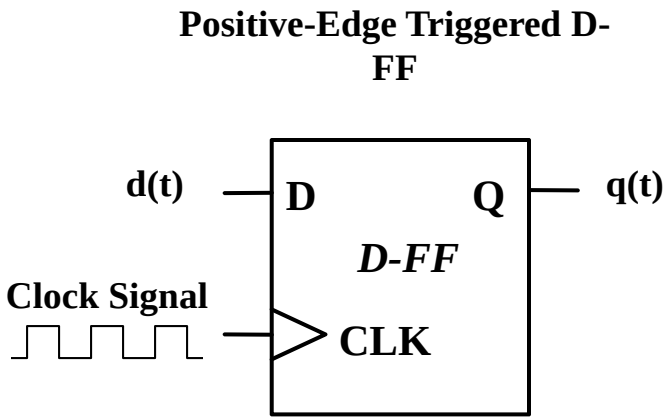


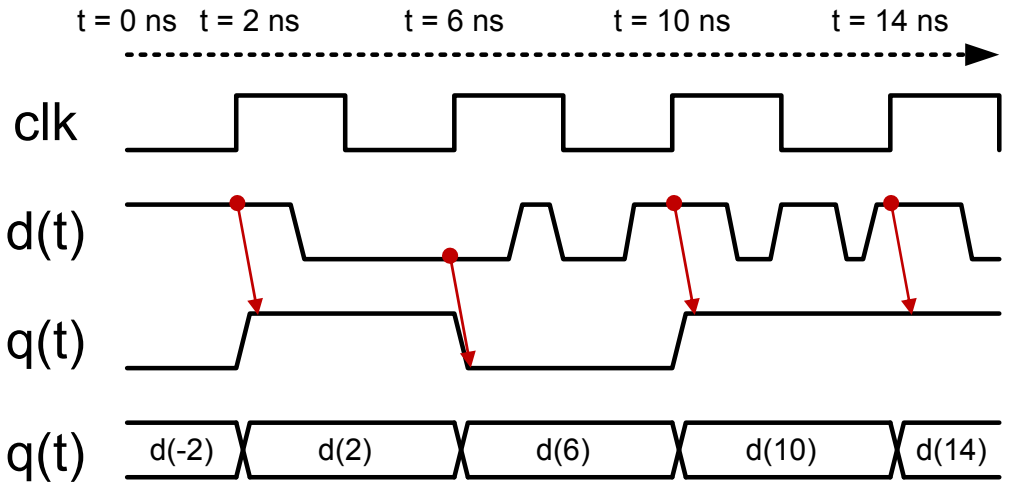
2.8 GHz
= 2.8*10^9 cycles per second
= 0.357 ns/cycle



D Flip-Flop

- Fundamental sequential building block
- q-output samples the d-input at the instant of the rising clock edge and then retains that value until the next clock edge

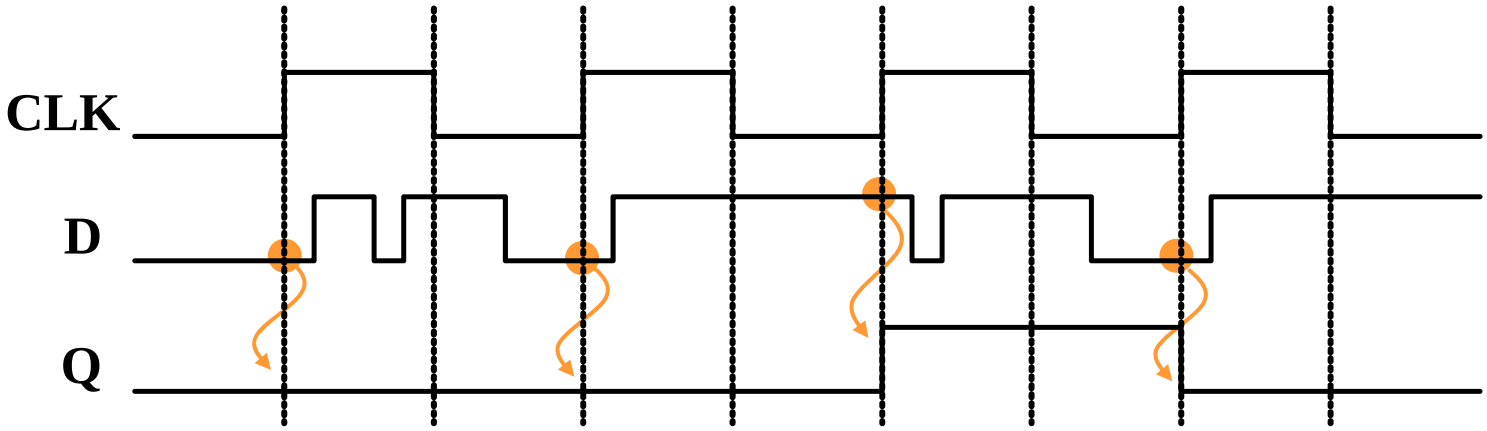




Positive-Edge Triggered D-FF

- Q looks at D only at the positive-edge

CLK	D	Q*
0	x	Q
1	x	Q
↑	0	0
↑	1	1

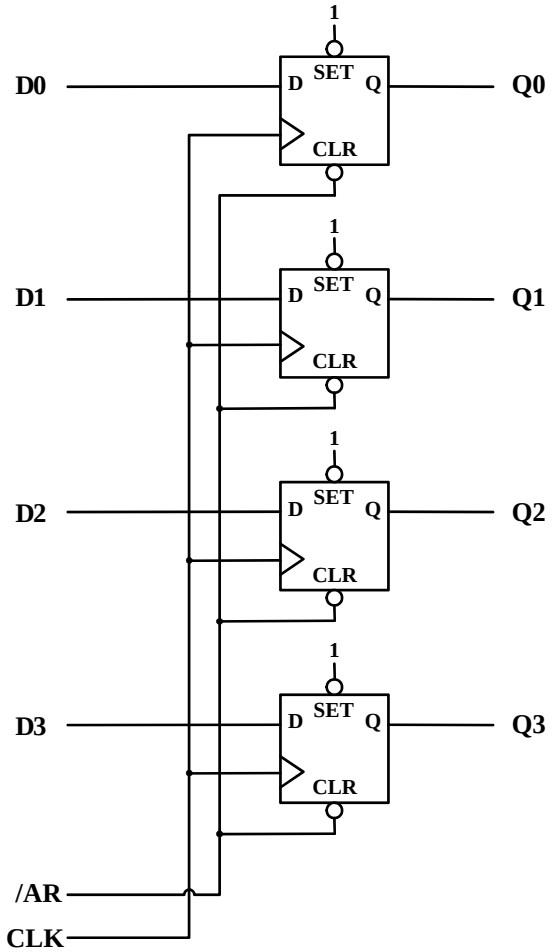


Q only samples D at the positive edges and then holds that value until the next edge

Registers

- A Register is a group of D-FFs tied to a common clock and clear (reset) input
  - Reset input allows register to be initialized to 0s
- Used to store multiple bit values on each clock cycle

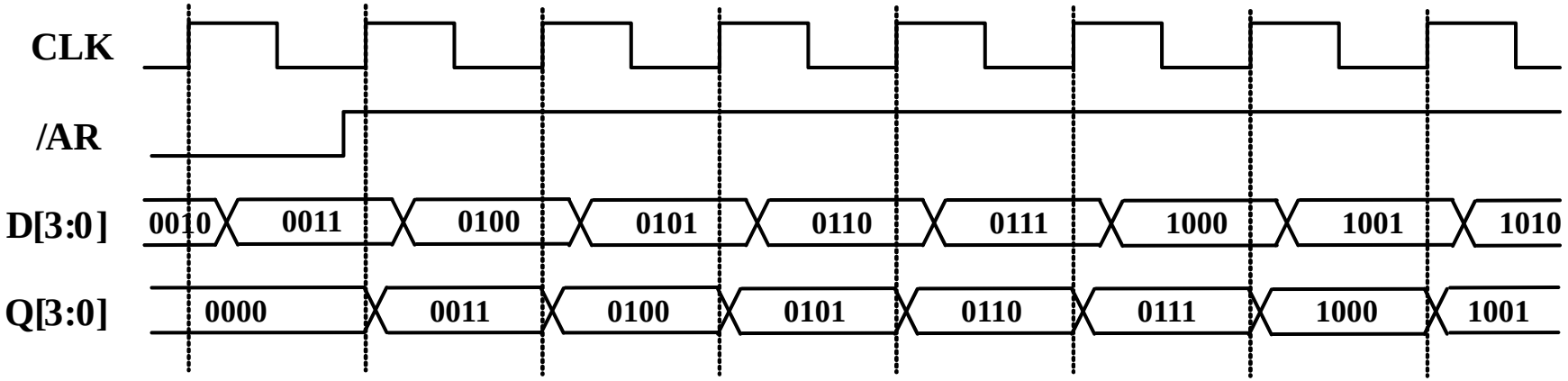
CLK	/AR	D_i	Q_i^*
X	0	X	0
1,0	1	X	Q_i
↑	1	0	0
↑	1	1	1



4-bit Register

Registers

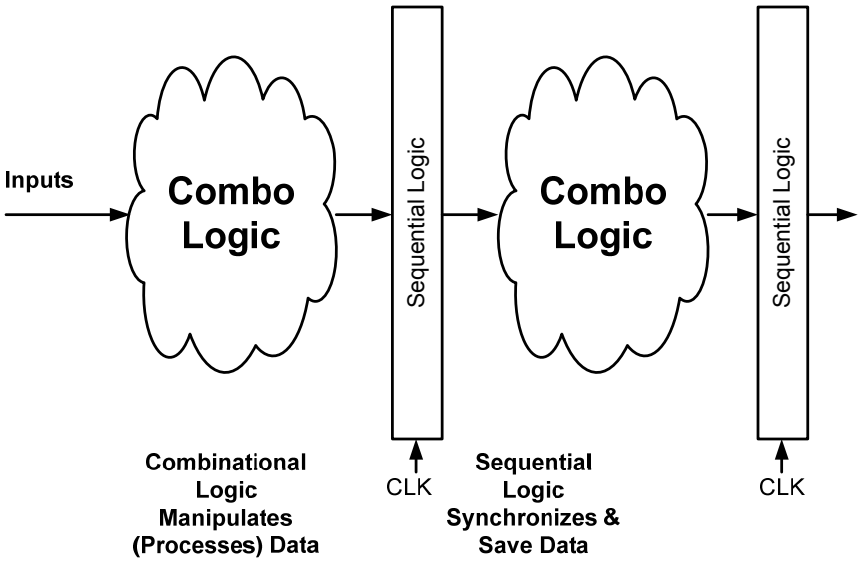
- Whatever the D value is at the clock edge, is sampled and passed to the Q output until the next clock edge



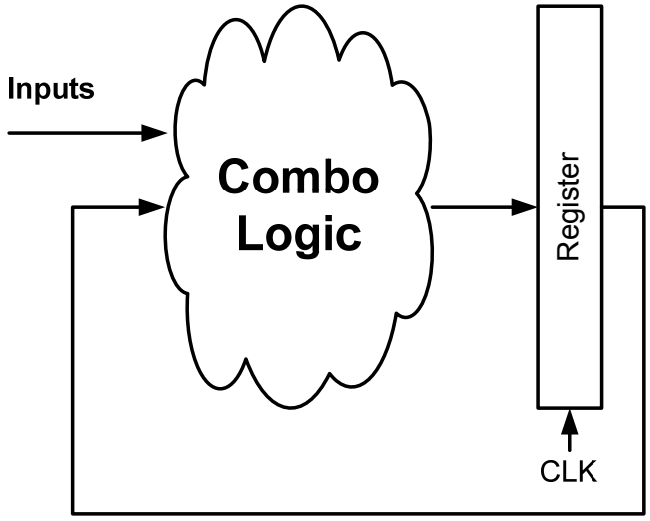
4-bit Register – On clock edge, D is passed to Q

Clocking Methodologies

- Typical designs use both combinational and sequential logic
  - Sequential logic: saves and synchronize data
  - Combinational logic: performs some operation on the data
- Can use feed-forward or feed-back methodology



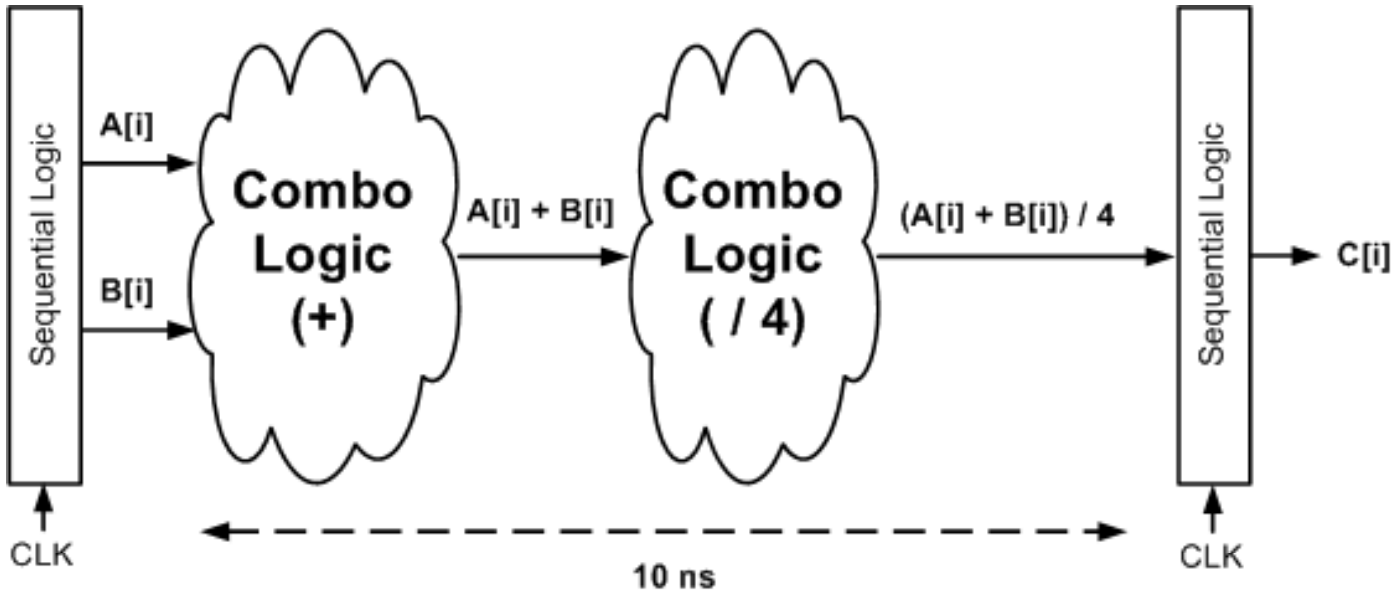
Feed-forward Style



Feed-back Style

Example

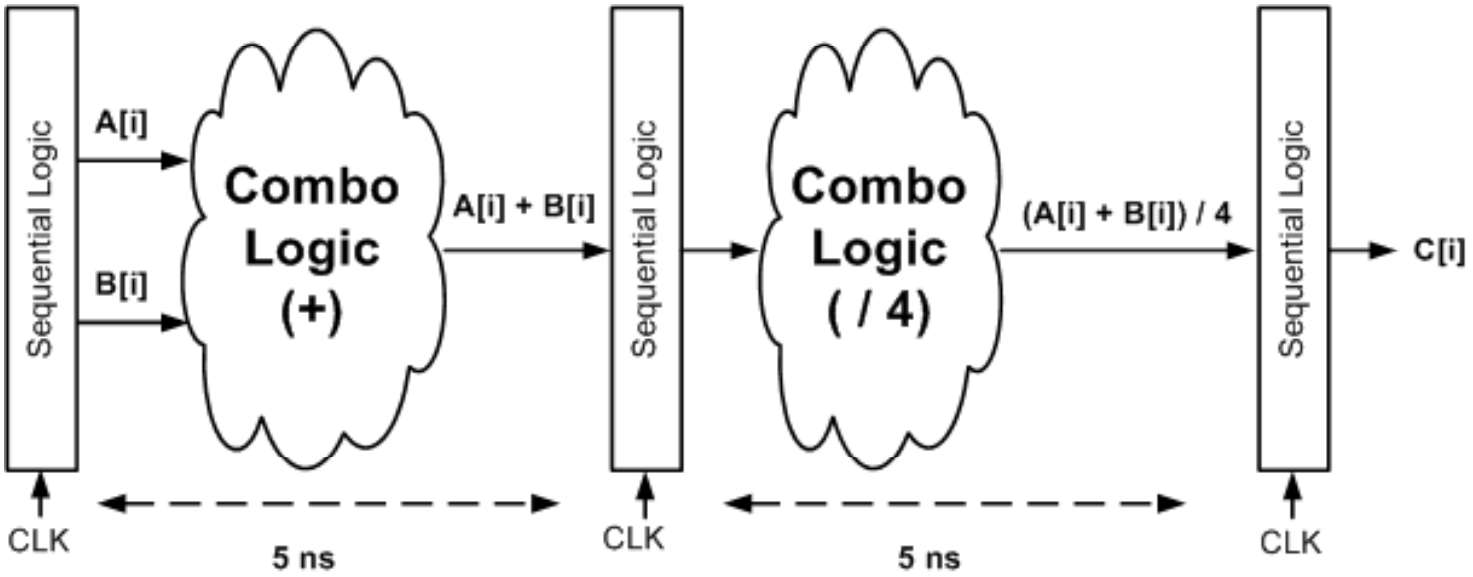
```
for(i=0; i < 100; i++)
  C[i] = (A[i] + B[i]) / 4;
```



10 ns per input set = 1000 ns total

Feed-forward (Pipelining) Example

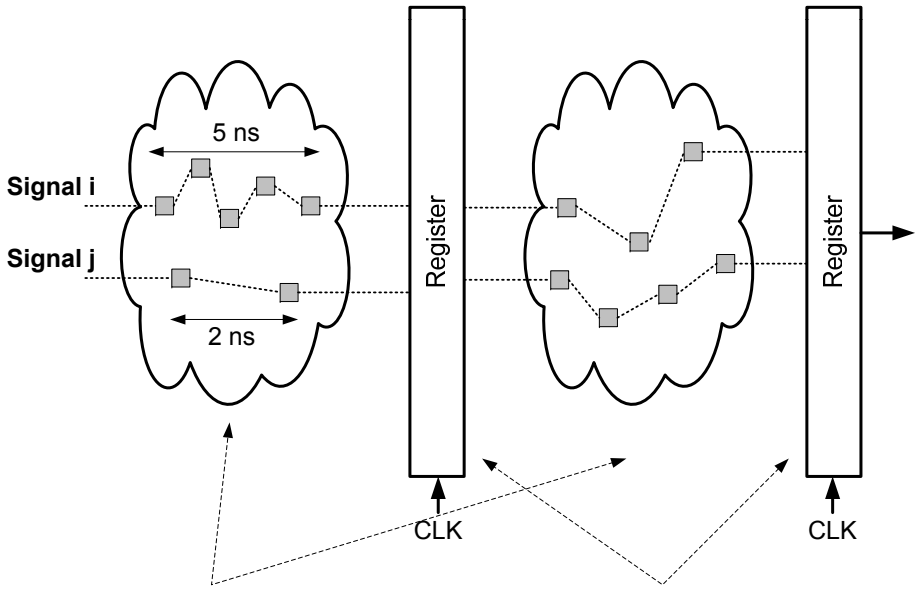
```
for(i=0; i < 100; i++)
  C[i] = (A[i] + B[i]) / 4;
```



	Stage 1	Stage 2
Clock 0	A[0] + B[0]	
Clock 1	A[1] + B[1]	(A[0] + B[0]) / 4
Clock 2	A[2] + B[2]	(A[1] + B[1]) / 4

Need for Registers

- Provides separation between combinational functions
  - Without registers, fast signals could "catch-up" to data values in the next operation stage



Counter Example