

# CSE1003-Digital Logic Design

**Dr.Penchalaiah Palla**

Dept. of Micro and Nanoelectronics  
School of Electronics Engineering,VIT,  
Vellore

CSE1003	DIGITAL LOGIC AND DESIGN	L 3	T 0	P 2	J 0	C 4
Pre-requisite	NIL	Syllabus version v1.0				

### Course Objectives:

1. Introduce the concept of digital and binary systems.
2. Analyze and Design combinational and sequential logic circuits.
3. Reinforce theory and techniques taught in the classroom through experiments in the laboratory.

### Expected Course Outcome:

1. Comprehend the different types of number system.
2. Evaluate and simplify logic functions using Boolean Algebra and K-map.
3. Design minimal combinational logic circuits.
4. Analyze the operation of medium complexity standard combinational circuits like the encoder, decoder, multiplexer, demultiplexer.
5. Analyze and Design the Basic Sequential Logic Circuits
6. Outline the construction of Basic Arithmetic and Logic Circuits
7. Acquire design thinking capability, ability to design a component with realistic constraints, to solve real world engineering problems and analyze the results.

<b>Module:1</b>	<b>INTRODUCTION</b>	<b>3 hours</b>
Number System - Base Conversion - Binary Codes - Complements(Binary and Decimal)		
<b>Module:2</b>	<b>BOOLEAN ALGEBRA</b>	<b>8 hours</b>
Boolean algebra - Properties of Boolean algebra - Boolean functions - Canonical and Standard forms - Logic gates - Universal gates – Karnaugh map - Don't care conditions - Tabulation Method		
<b>Module:3</b>	<b>COMBINATIONAL CIRCUIT - I</b>	<b>4 hours</b>
Adder - Subtractor - Code Converter - Analyzing a Combinational Circuit		
<b>Module:4</b>	<b>COMBINATIONAL CIRCUIT -II</b>	<b>6 hours</b>
Binary Parallel Adder- Look ahead carry - Magnitude Comparator - Decoders – Encoders - Multiplexers –Demultiplexers.		

<b>Module:5</b>	<b>SEQUENTIAL CIRCUITS – I</b>	<b>6 hours</b>
Flip Flops - Sequential Circuit: Design and Analysis - Finite State Machine: Moore and Mealy model - Sequence Detector.		
<b>Module:6</b>	<b>SEQUENTIAL CIRCUITS – II</b>	<b>7 hours</b>
Registers - Shift Registers - Counters - Ripple and Synchronous Counters - Modulo counters - Ring and Johnson counters		
<b>Module:7</b>	<b>ARITHMETIC LOGIC UNIT</b>	<b>9 hours</b>
Bus Organization - ALU - Design of ALU - Status Register - Design of Shifter - Processor Unit - Design of specific Arithmetic Circuits Accumulator - Design of Accumulator.		
<b>Module:8</b>	<b>Contemporary Issues: RECENT TRENDS</b>	<b>2 hours</b>
	<b>Total Lecture hours:</b>	<b>45 hours</b>

**Text Book(s)**

1. M. Morris Mano and Michael D.Ciletti— Digital Design: With an introduction to Verilog HDL, Pearson Education – 5th Edition- 2014. ISBN:9789332535763.

**Reference Books**

1. Peterson, L.L. and Davie, B.S., 2007. Computer networks: a systems approach. Elsevier.
2. Thomas L Floyd. 2015. Digital Fundamentals. Pearson Education. ISBN: 9780132737968
3. Malvino, A.P. and Leach, D.P. and Goutam Saha. 2014. Digital Principles and Applications (SIE). Tata McGraw Hill. ISBN: 9789339203405.
4. Morris Mano, M. and Michael D.Ciletti. 2014. Digital Design: With an introduction to Verilog HDL. Pearson Education. ISBN:9789332535763

Mode of Evaluation: CAT / Assignment / Quiz / FAT / Project / Seminar

<b>List of Challenging Experiments (Indicative)</b>		
1.	Realization of Logic gates using discrete components, verification of truth table for logic gates, realization of basic gates using NAND and NOR gates	4.5 hours
	Implementation of Logic Circuits by verification of Boolean laws and verification of De Morgans law	3 hours
	Adder and Subtractor circuit realization by implementation of Half-Adder and Full-Adder, and by implementation of Half-Subtractor and Full-Subtractor	4.5 hours
	Combinational circuit design i. Design of Decoder and Encoder ii. Design of Multiplexer and De multiplexer iii. Design of Magnitude Comparator iv. Design of Code Converter	4.5 hours
	Sequential circuit design i. Design of Mealy and Moore circuit ii. Implementation of Shift registers iii. Design of 4-bit Counter iv. Design of Ring Counter	4.5 hours
	Implementation of different circuits to solve real world problems: A digitally controlled locker works based on a control switch and two keys which are entered by the user. Each key has a 2-bit binary representation. If the control switch is pressed, the locking system will pass the difference of two keys into the controller unit. Otherwise, the locking system will pass the sum of the two numbers to the controller unit. Design a circuit to determine the input to the controller unit.	4.5 hours
	Implementation of different circuits to solve real world problems: A bank queuing system has a capacity of 5 customers which serves on first come first served basis. A display unit is used to display the number of customers waiting in the queue. Whenever a customer leaves the queue, the count is reduced by one and the count is increased by one if a customer joins a queue. Two sensors (control signals) are used to sense customers leaving and joining the queue respectively. Design a circuit that displays the number of customers waiting in the queue in binary format using LEDs. Binary 1 is represented by LED glow and 0 otherwise.	4.5 hours
<b>Total Laboratory Hours</b>		<b>30 hours</b>
Mode of assessment: Project/Activity		
Mod-1-Introduction-Number Systems-Conversion-Binary codes	Recommended by Board of Studies 28-02-2017	
Approved by Academic Council	No. 46	Date 24-08-2017

# 1. Number Systems

Module:1	INTRODUCTION	3 hours
Number System - Base Conversion - Binary Codes - Complements(Binary and Decimal)		

# Common Number Systems

System	Base	Symbols	Used by humans?	Used in computers?
Decimal	10	0, 1, ... 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, ... 7	No	No
Hexa-decimal	16	0, 1, ... 9, A, B, ... F	No	No

# Quantities/Counting (1 of 3)

Decimal	Binary	Octal	Hexa-decimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7

# Quantities/Counting (2 of 3)

Decimal	Binary	Octal	Hexa-decimal
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

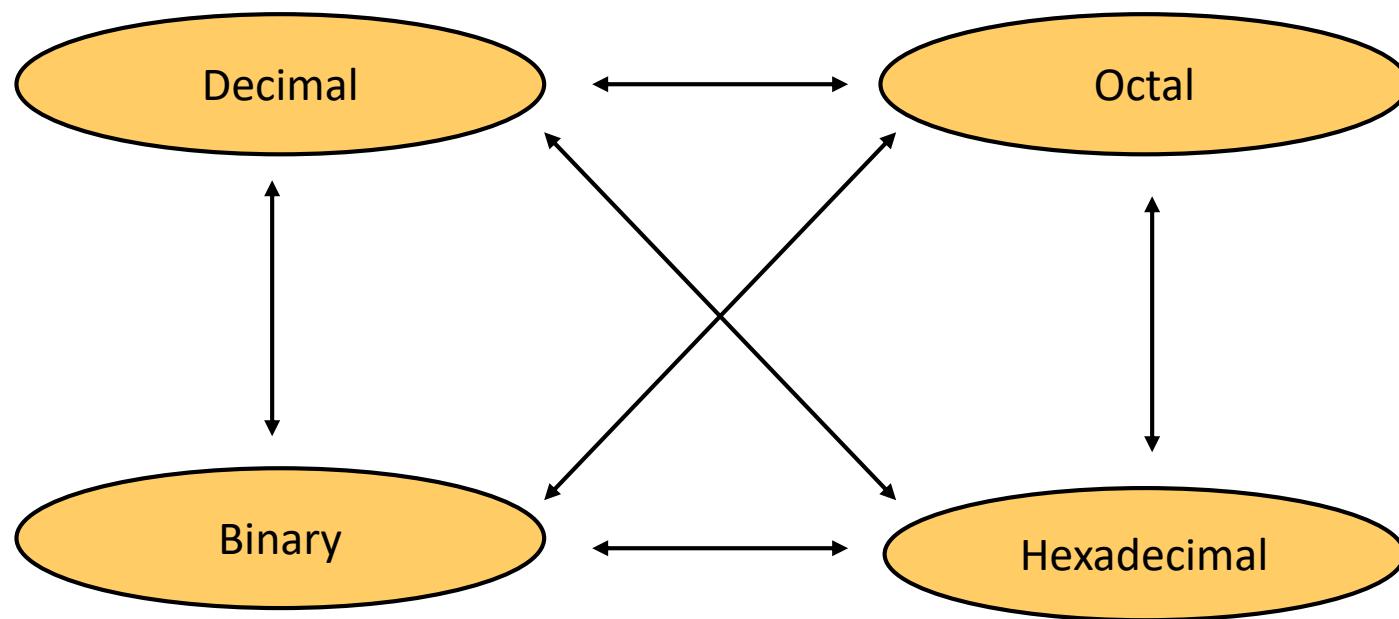
# Quantities/Counting (3 of 3)

Decimal	Binary	Octal	Hexa-decimal
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17

Etc.

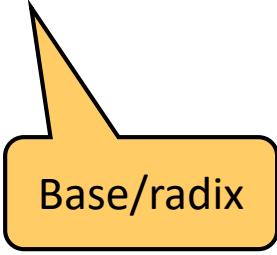
# Conversion Among Bases

- The possibilities:



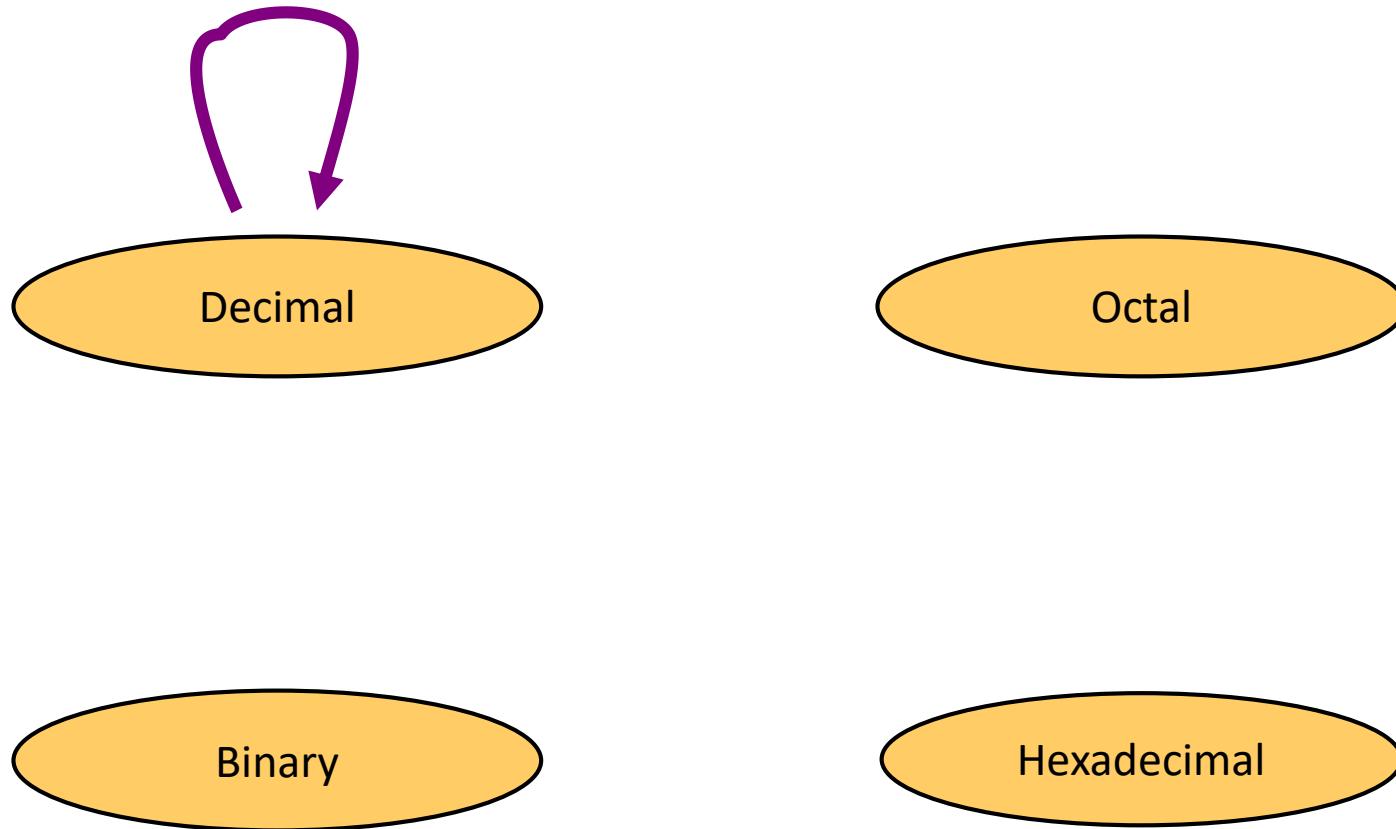
# Quick Example

$$25_{10} = 11001_2 = 31_8 = 19_{16}$$



Base/radix

# Decimal to Decimal (just for fun)

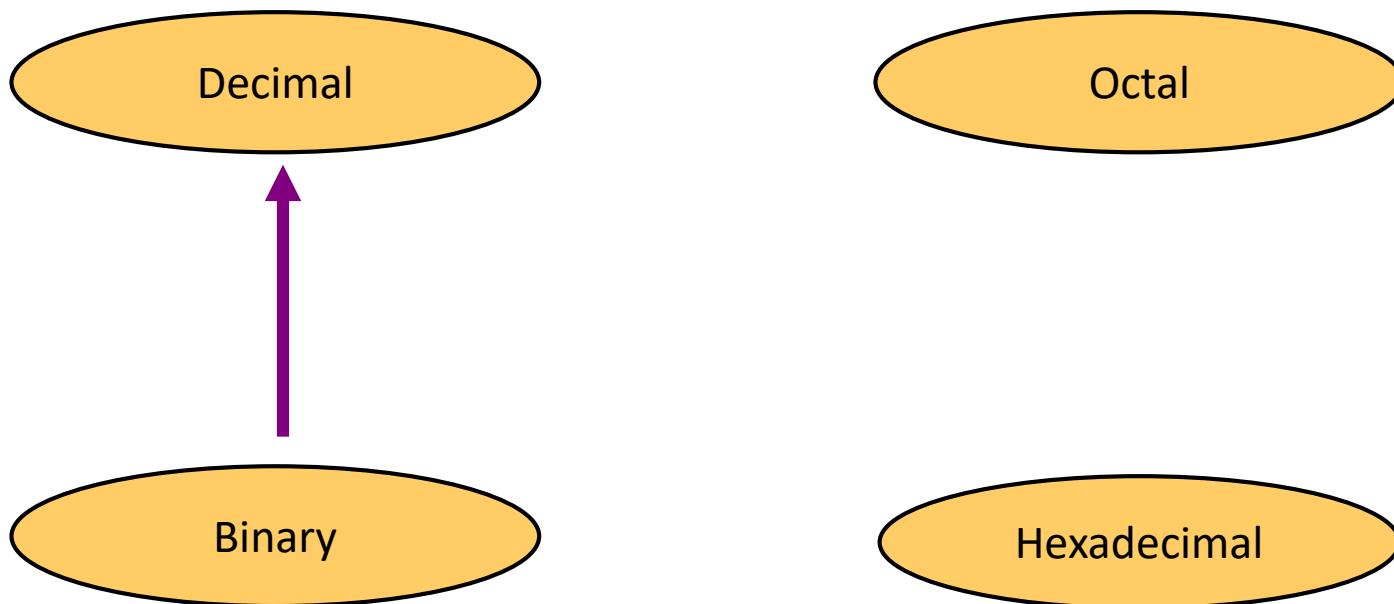


Weight

$$\begin{array}{rcl} 125_{10} \Rightarrow & 5 \times 10^0 & = 5 \\ & 2 \times 10^1 & = 20 \\ & 1 \times 10^2 & = \underline{100} \\ & & 125 \end{array}$$

Base

# Binary to Decimal



# Binary to Decimal

- Technique
  - Multiply each bit by  $2^n$ , where  $n$  is the “weight” of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

# Example

Bit “0”

$101011_2 \Rightarrow$

$$1 \times 2^0 = 1$$

$$1 \times 2^1 = 2$$

$$0 \times 2^2 = 0$$

$$1 \times 2^3 = 8$$

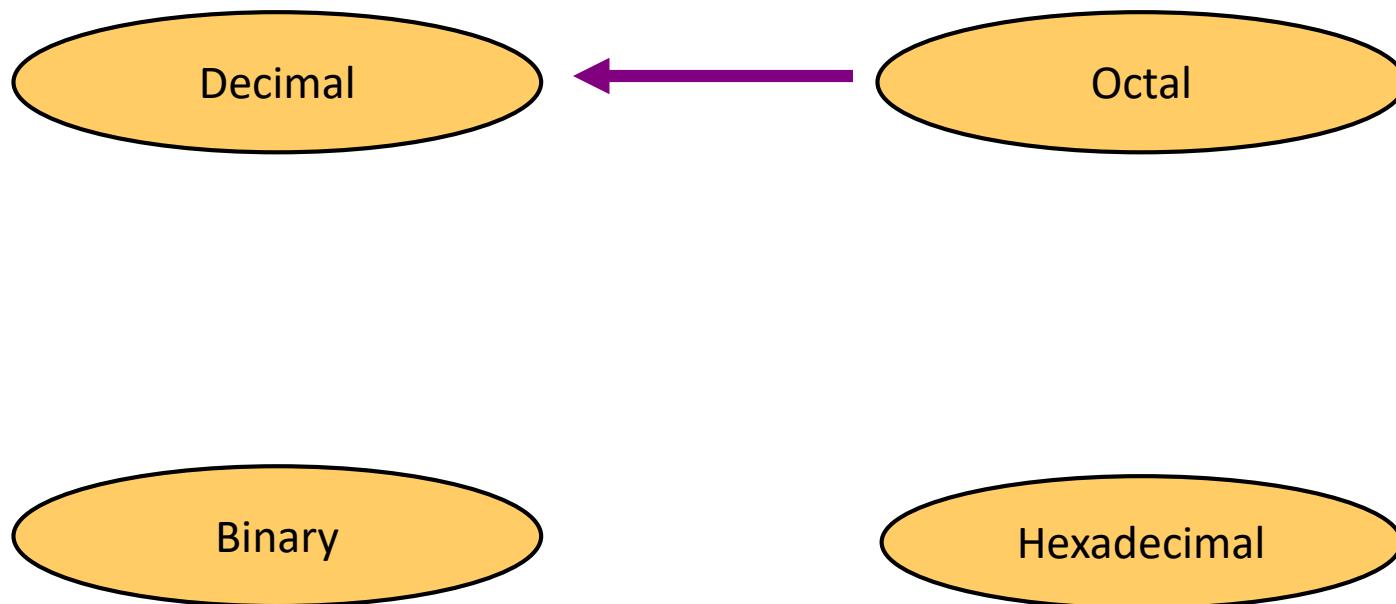
$$0 \times 2^4 = 0$$

$$1 \times 2^5 = 32$$

---

$$43_{10}$$

# Octal to Decimal



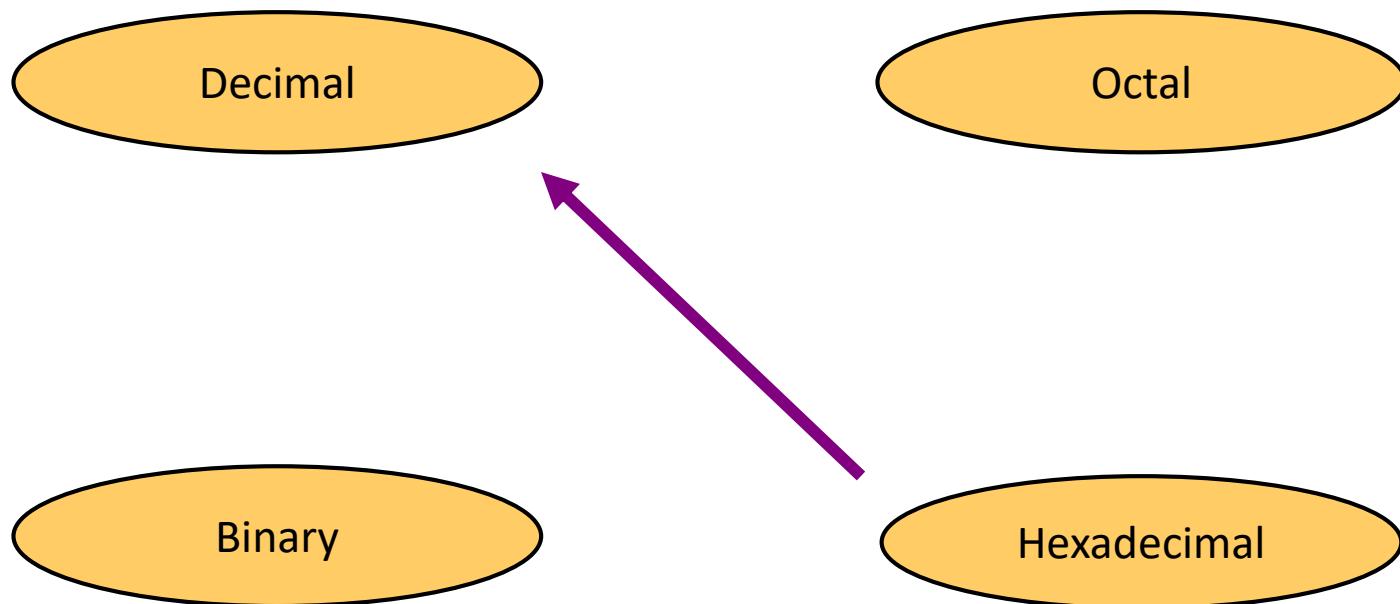
# Octal to Decimal

- Technique
  - Multiply each bit by  $8^n$ , where  $n$  is the “weight” of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

# Example

$$\begin{array}{rcl} 724_8 &=>& 4 \times 8^0 = 4 \\ && 2 \times 8^1 = 16 \\ && 7 \times 8^2 = \underline{\quad 448} \\ && \hline && 468_{10} \end{array}$$

# Hexadecimal to Decimal



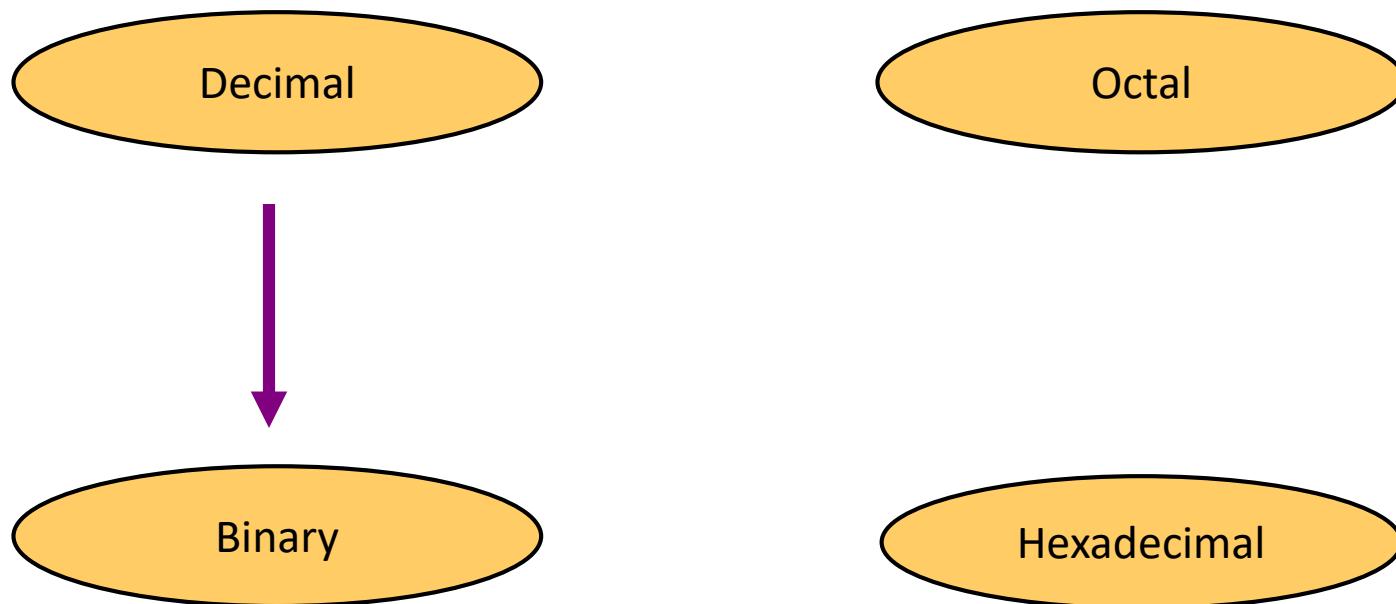
# Hexadecimal to Decimal

- Technique
  - Multiply each bit by  $16^n$ , where  $n$  is the “weight” of the bit
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

# Example

$$\begin{array}{rcl} \text{ABC}_{16} \Rightarrow & C \times 16^0 = 12 \times 1 = 12 \\ & B \times 16^1 = 11 \times 16 = 176 \\ & A \times 16^2 = 10 \times 256 = \underline{2560} \\ & & 2748_{10} \end{array}$$

# Decimal to Binary



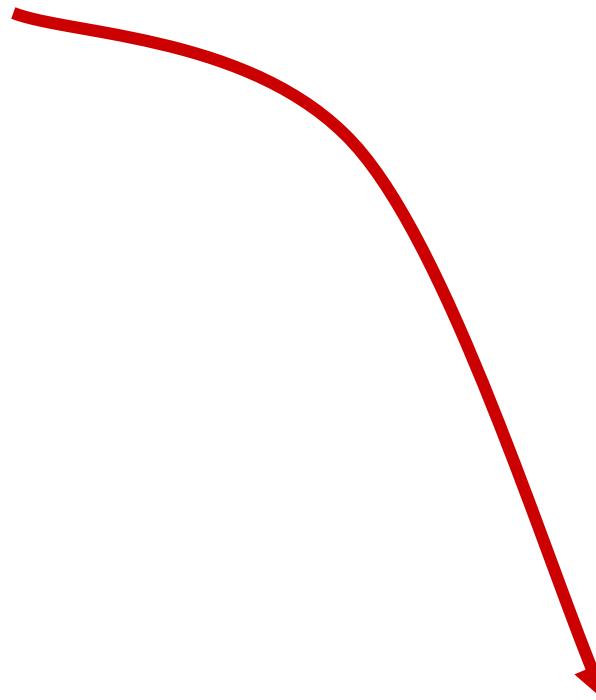
# Decimal to Binary

- Technique
  - Divide by two, keep track of the remainder
  - First remainder is bit 0 (LSB, least-significant bit)
  - Second remainder is bit 1
  - Etc.

# Example

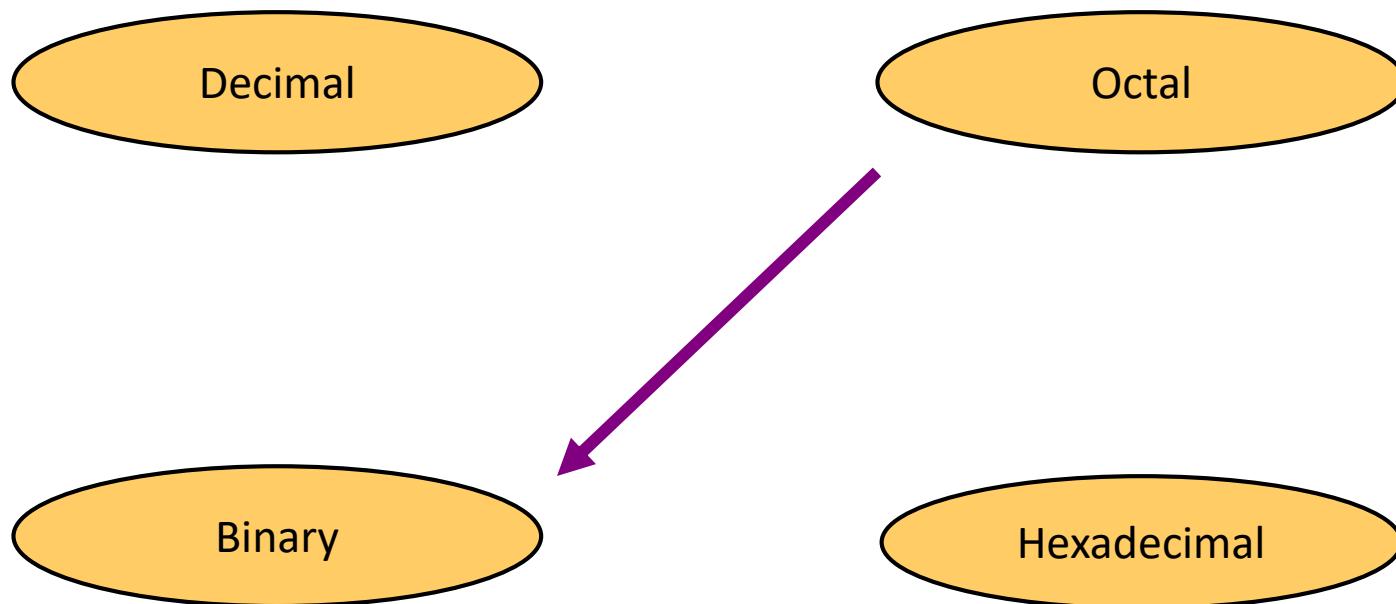
$$125_{10} = ?_2$$

2		125	
2		62	1
2		31	0
2		15	1
2		7	1
2		3	1
2		1	1
		0	



$$125_{10} = 1111101_2$$

# Octal to Binary



# Octal to Binary

- Technique
  - Convert each octal digit to a 3-bit equivalent binary representation

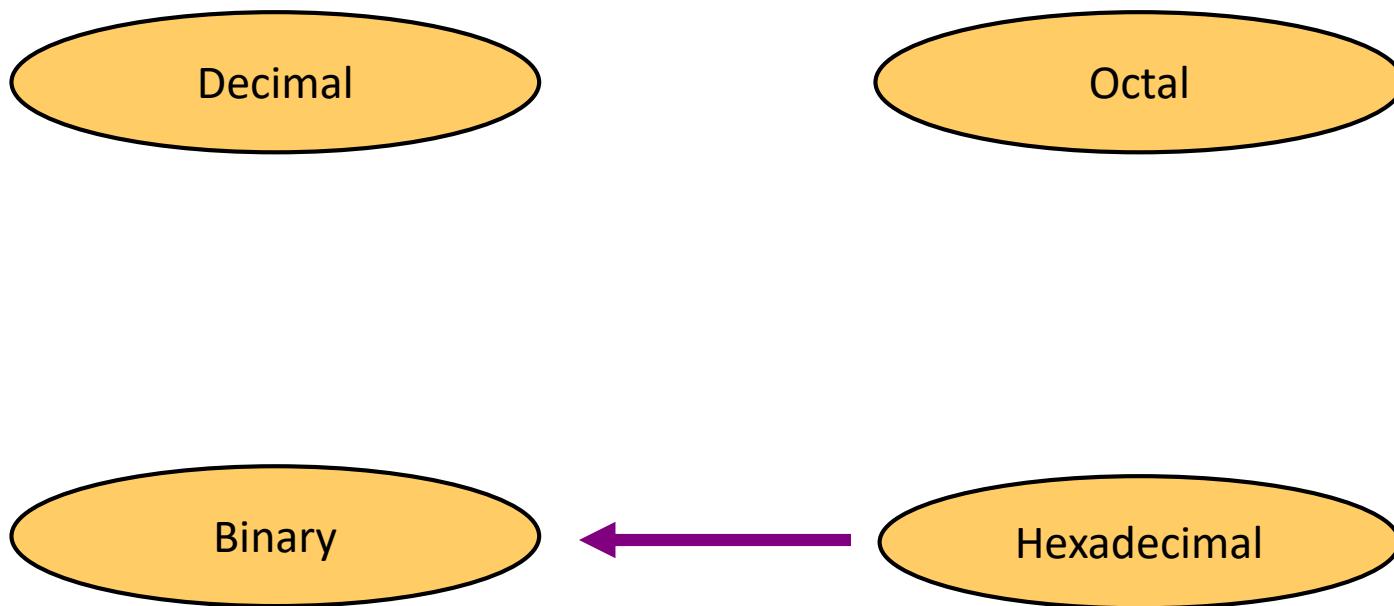
# Example

$$705_8 = ?_2$$

7	0	5
↓	↓	↓
111	000	101

$$705_8 = 111000101_2$$

# Hexadecimal to Binary



# Hexadecimal to Binary

- Technique
  - Convert each hexadecimal digit to a 4-bit equivalent binary representation

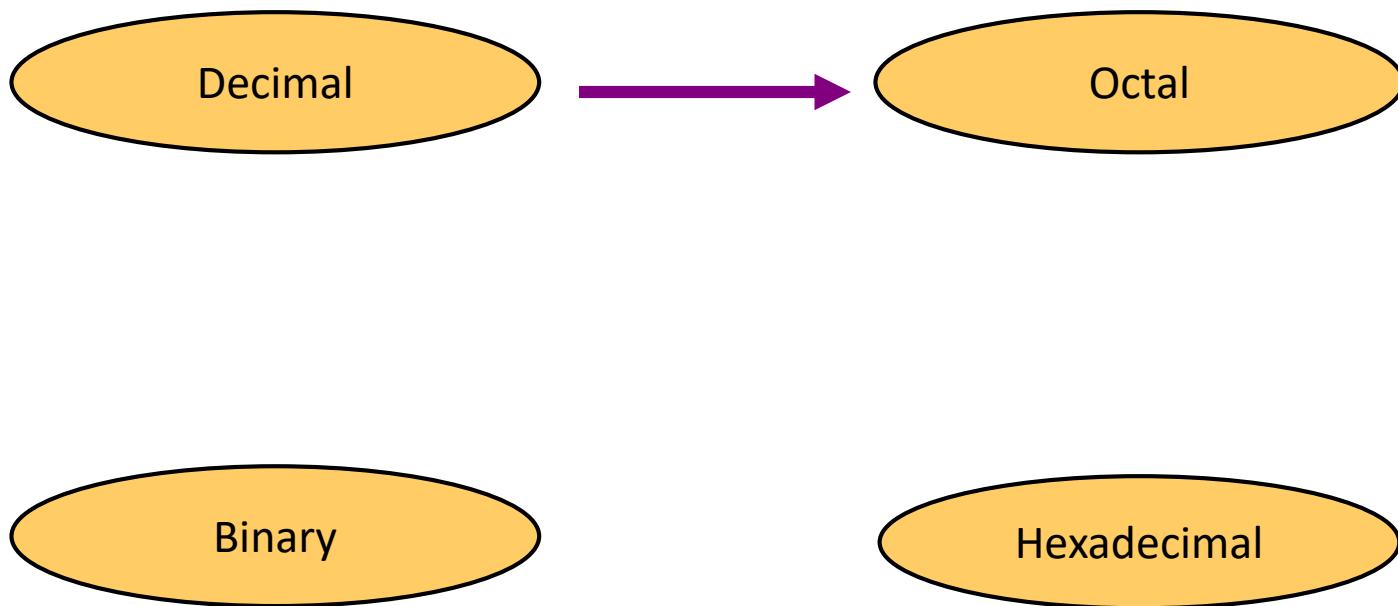
# Example

$$10AF_{16} = ?_2$$

1	0	A	F
↓	↓	↓	↓
0001	0000	1010	1111

$$10AF_{16} = 0001000010101111_2$$

# Decimal to Octal



# Decimal to Octal

- Technique
  - Divide by 8
  - Keep track of the remainder

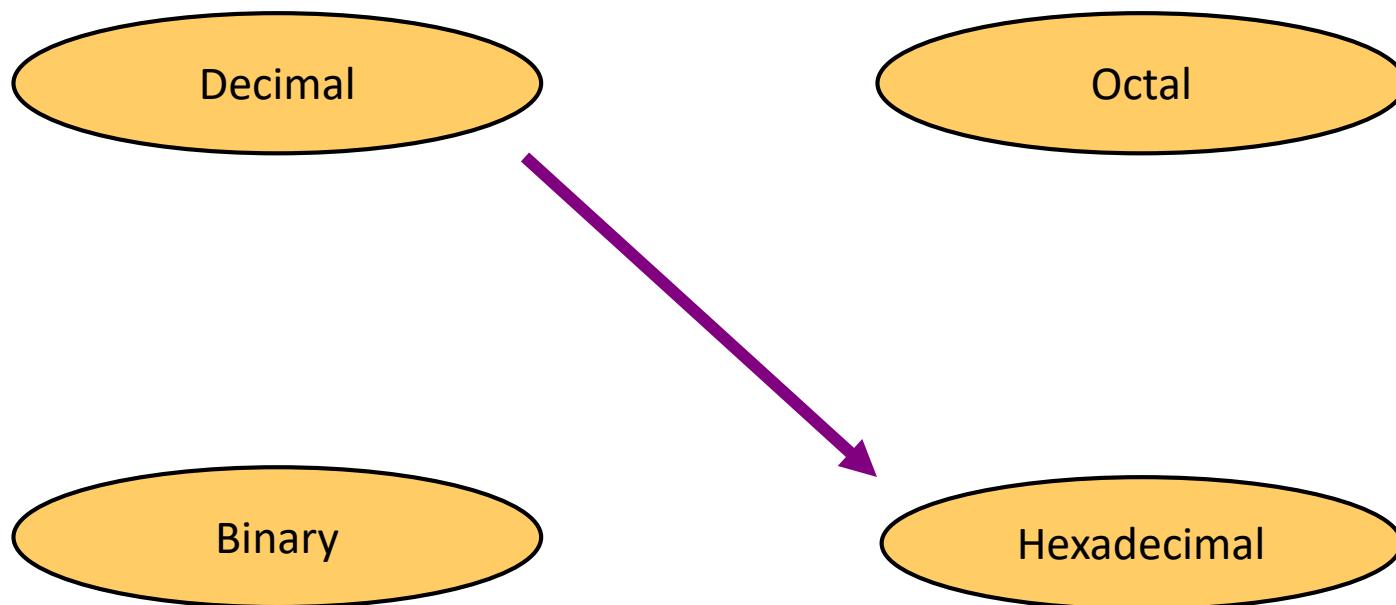
# Example

$$1234_{10} = ?_8$$

$$\begin{array}{r} 1234 \\ \hline 8 | 154 \\ \hline 8 | 19 \\ \hline 8 | 2 \\ \hline 0 & 2 & 3 & 2 \end{array}$$

$$1234_{10} = 2322_8$$

# Decimal to Hexadecimal



# Decimal to Hexadecimal

- Technique
  - Divide by 16
  - Keep track of the remainder

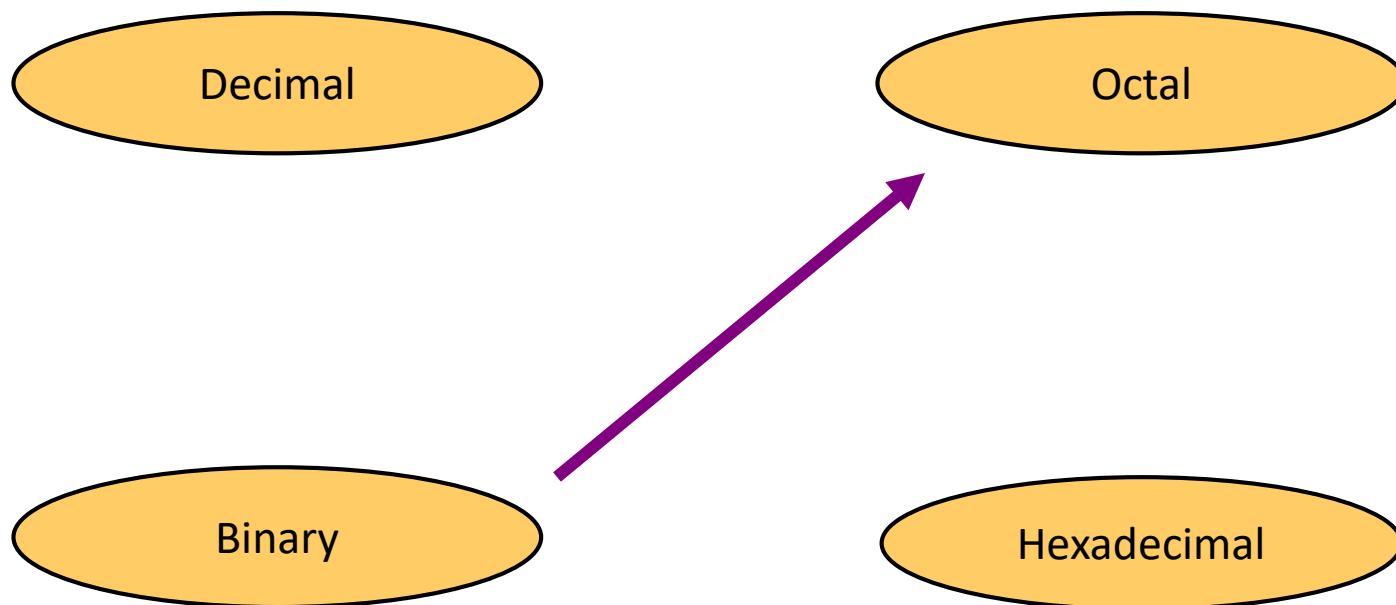
# Example

$$1234_{10} = ?_{16}$$

$$\begin{array}{r} 16 \Big| 1234 \\ 16 \Big| 77 \\ 16 \Big| 4 \\ \hline 0 \end{array} \quad \begin{matrix} 2 \\ 13 = D \\ 4 \end{matrix}$$

$$1234_{10} = 4D2_{16}$$

# Binary to Octal



# Binary to Octal

- Technique
  - Group bits in threes, starting on right
  - Convert to octal digits

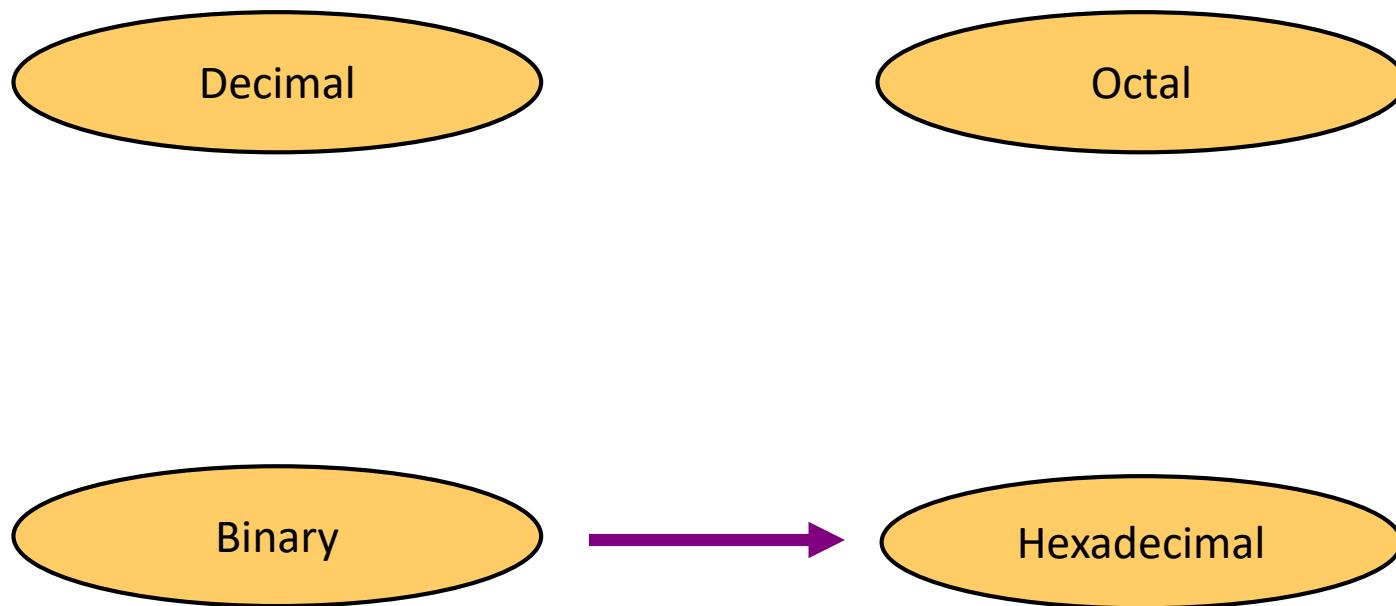
# Example

$$1011010111_2 = ?_8$$

1	011	010	111
↓	↓	↓	↓
1	3	2	7

$$1011010111_2 = 1327_8$$

# Binary to Hexadecimal



# Binary to Hexadecimal

- Technique
  - Group bits in fours, starting on right
  - Convert to hexadecimal digits

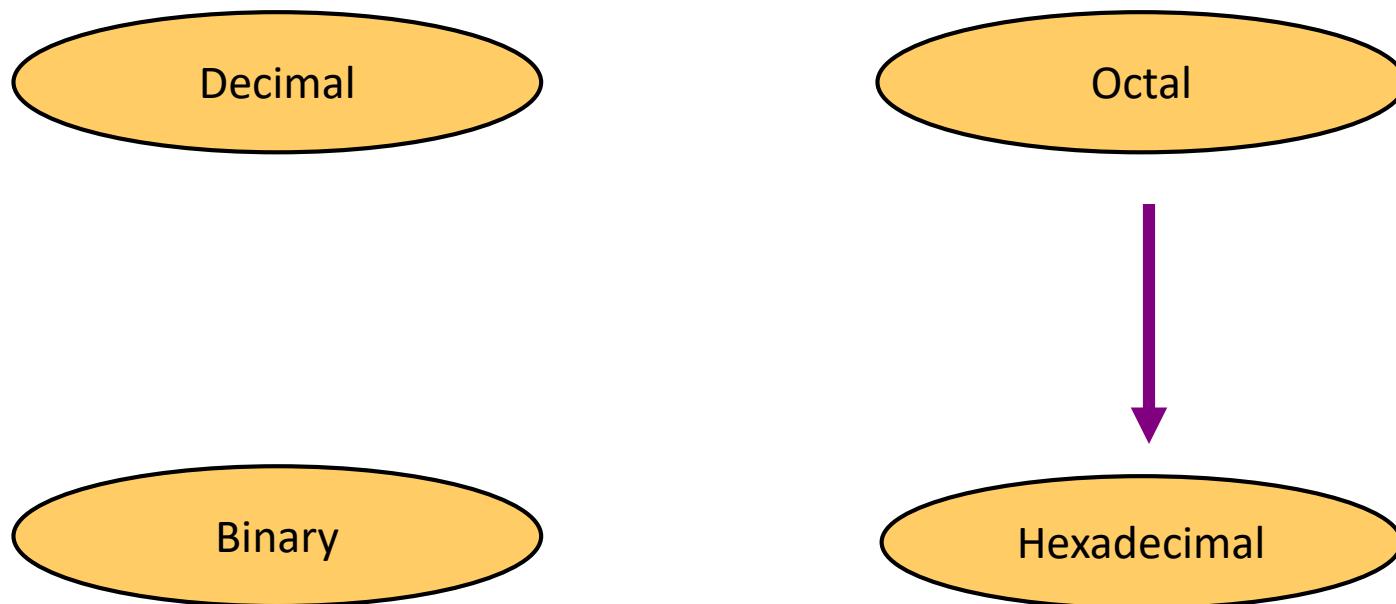
# Example

$$1010111011_2 = ?_{16}$$

10 1011 1011  
↓ ↓ ↓  
2 B B

$$1010111011_2 = 2BB_{16}$$

# Octal to Hexadecimal

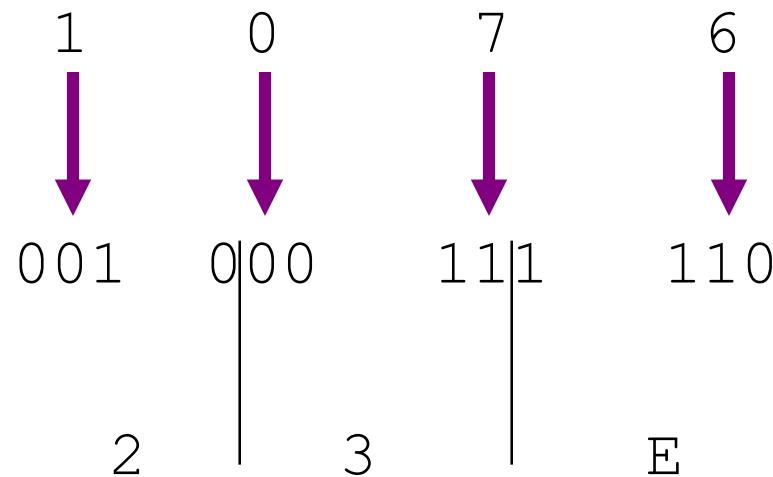


# Octal to Hexadecimal

- Technique
  - Use binary as an intermediary

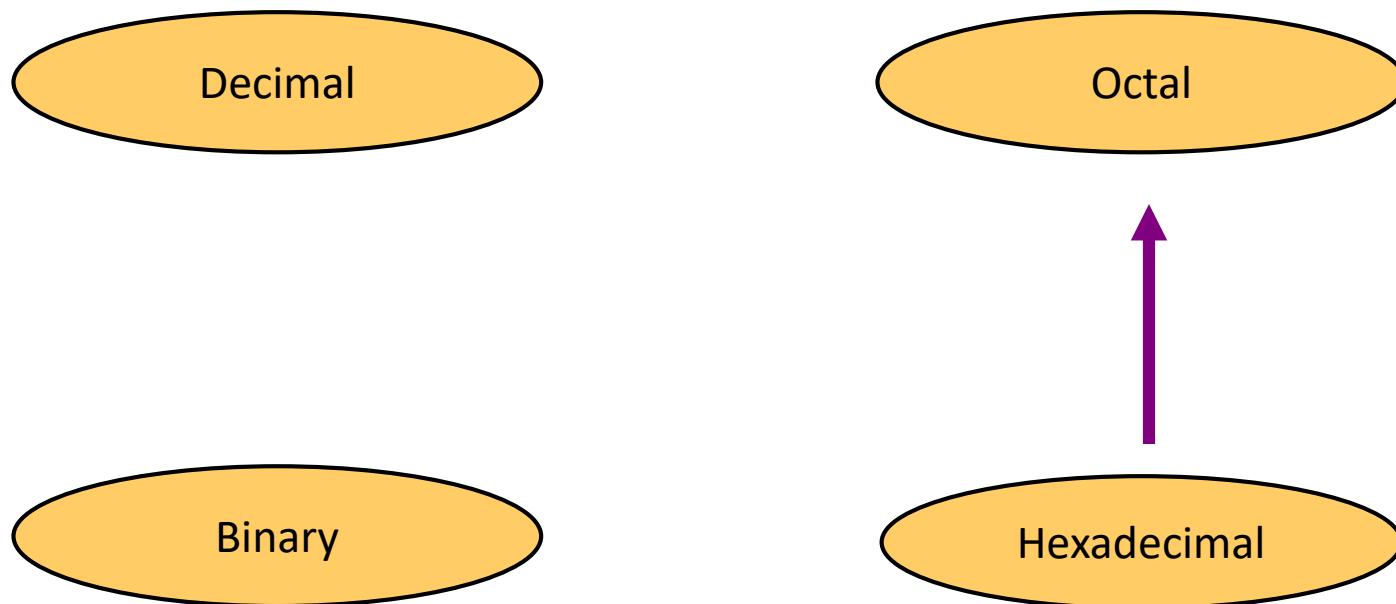
# Example

$$1076_8 = ?_{16}$$



$$1076_8 = 23E_{16}$$

# Hexadecimal to Octal

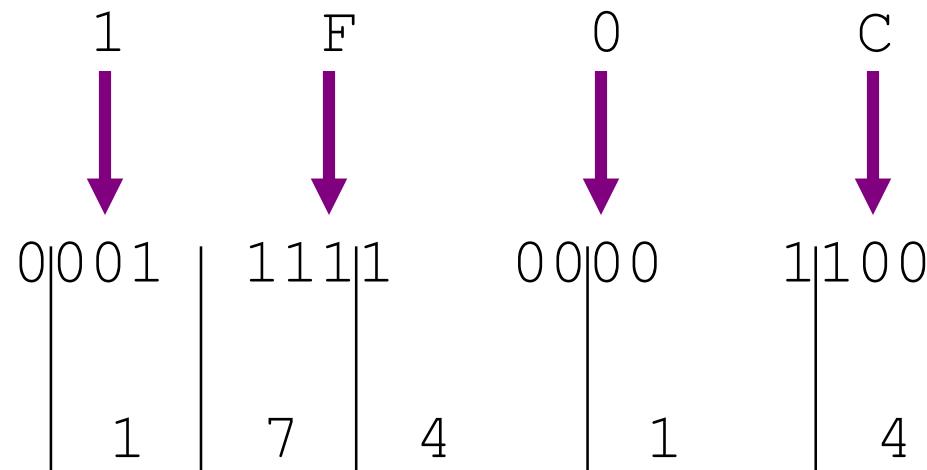


# Hexadecimal to Octal

- Technique
  - Use binary as an intermediary

# Example

$$1F0C_{16} = ?_8$$



$$1F0C_{16} = 17414_8$$

# Exercise – Convert ...

Decimal	Binary	Octal	Hexa-decimal
33			
	1110101		
		703	
			1AF

Don't use a calculator!

Skip answer

Answer

# Exercise – Convert

Answer

Decimal	Binary	Octal	Hexa-decimal
33	100001	41	21
117	1110101	165	75
451	111000011	703	1C3
431	110101111	657	1AF



# Common Powers (1 of 2)

- Base 10

Power	Preface	Symbol	Value
$10^{-12}$	pico	p	.000000000001
$10^{-9}$	nano	n	.000000001
$10^{-6}$	micro	$\mu$	.000001
$10^{-3}$	milli	m	.001
$10^3$	kilo	k	1000
$10^6$	mega	M	1000000
$10^9$	giga	G	1000000000
$10^{12}$	tera	T	1000000000000

# Common Powers (2 of 2)

- Base 2

Power	Preface	Symbol	Value
$2^{10}$	kilo	k	1024
$2^{20}$	mega	M	1048576
$2^{30}$	Giga	G	1073741824

- What is the value of “k”, “M”, and “G”?
- In computing, particularly w.r.t. memory, the base-2 interpretation generally applies

# Fractions

- Decimal to decimal (just for fun)

$$\begin{array}{rcl} 3.14 & \Rightarrow & 4 \times 10^{-2} = 0.04 \\ & & 1 \times 10^{-1} = 0.1 \\ & & 3 \times 10^0 = \underline{\underline{3}} \\ & & 3.14 \end{array}$$

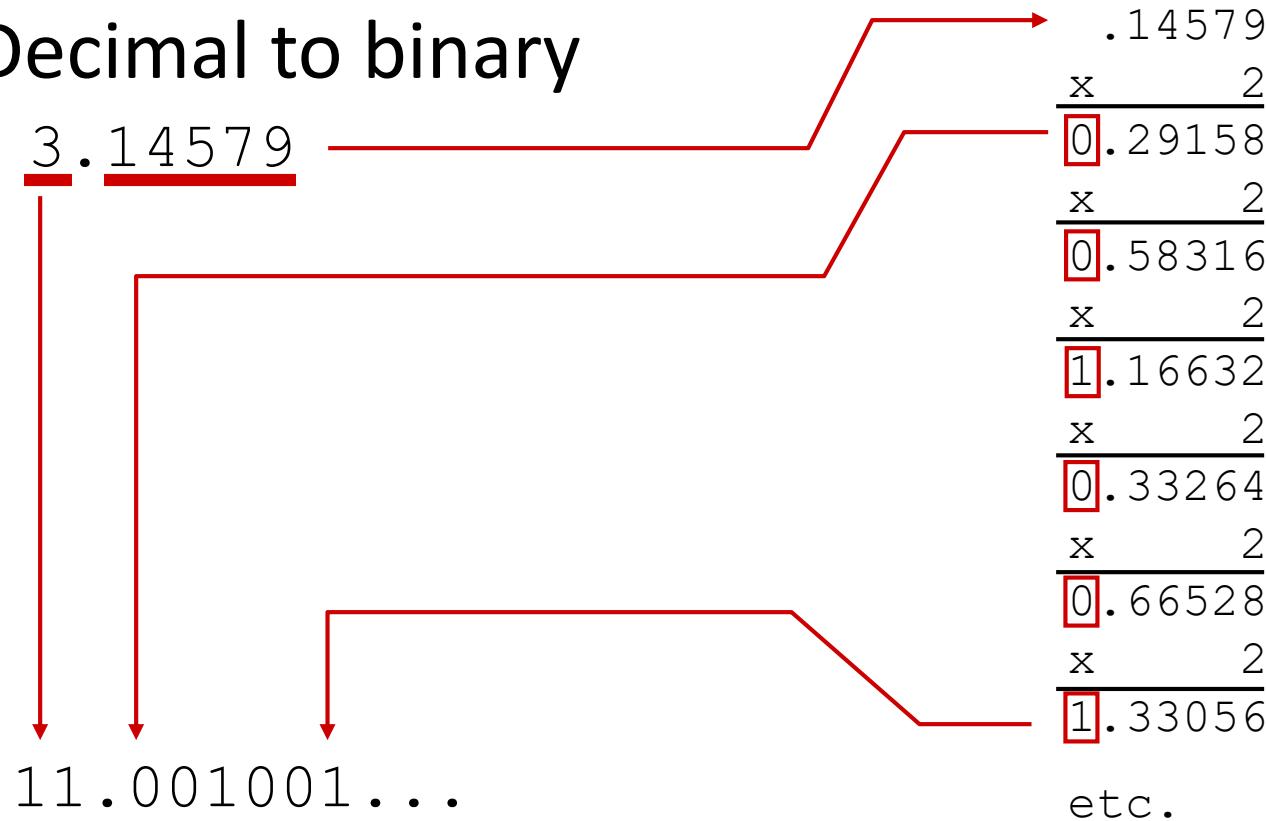
# Fractions

- Binary to decimal

$$\begin{array}{rcl} 10.1011 \Rightarrow & & \\ & 1 \times 2^{-4} = 0.0625 & \\ & 1 \times 2^{-3} = 0.125 & \\ & 0 \times 2^{-2} = 0.0 & \\ & 1 \times 2^{-1} = 0.5 & \\ & 0 \times 2^0 = 0.0 & \\ & 1 \times 2^1 = \underline{\underline{2.0}} & \\ & & 2.6875 \end{array}$$

# Fractions

- Decimal to binary



# Exercise – Convert ...

Decimal	Binary	Octal	Hexa-decimal
29.8			
	101.1101		
		3.07	
			C.82

Don't use a calculator!

Skip answer

Answer

# Exercise – Convert

Answer

Decimal	Binary	Octal	Hexa-decimal
29.8	11101.110011...	35.63...	1D.CC...
5.8125	101.1101	5.64	5.D
3.109375	11.000111	3.07	3.1C
12.5078125	1100.10000010	14.404	C.82



# Complements

- They are used to simplify the subtraction operation
- Two types (for each *base-r* system)
  - Diminishing radix complement ( $r-1$  complement)
  - Radix complement ( $r$  complement)

For *n*-digit number  $N$

$$(r^n - 1) - N \longrightarrow \text{r-1 complement}$$

$$r^n - N \longrightarrow \text{r complement}$$

# 9's and 10's Complements

- 9's complement of 674653
  - $999999-674653 = 325346$
- 9's complement of 023421
  - $999999-023421 = 976578$
- 10's complement of 674653
  - $325346+1 = 325347$
- 10's complement of 023421
  - $976578+1=976579$

# 1's and 2's Complements

- 1's complement of 10111001
  - $11111111 - 10111001 = 01000110$
  - Simply replace 1's and 0's
- 1's complement of 10100010
  - 01011101
- 2's complement of 10111001
  - $01000110 + 1 = 01000111$
  - Add 1 to 1's complement
- 2's complement of 10100010
  - $01011101 + 1 = 01011110$

# Subtraction with Complements of Unsigned

- $M - N$ 
  - Add  $M$ (minuend) to  $r$ 's complement of  $N$  (subtrahend)
    - $Sum = M + (r^n - N) = M - N + r^n$
  - If  $M > N$ ,  $Sum$  will have an end carry  $r^n$ , can be discarded
  - If  $M < N$ ,  $Sum$  will not have an end carry and
    - $Sum = r^n - (N - M)$  ( which is  $r$ 's complement of  $N - M$ )
    - So  $M - N = -(r$ 's complement of  $Sum$ )

# Subtraction with Complements of Unsigned

- $65438 - 5623$  (using 10's complement)

**65438**

**10's complement of 05623**      **+94377**

**159815**

**Discard end carry  $10^5$**       **-100000**

**Answer**      **59815**

# Subtraction with Complements of Unsigned

- $5623 - 65438$  (using 10's complement)

$$\begin{array}{r} 05623 \\ \underline{+34562} \\ 40185 \end{array}$$

**There is no end carry =>**

**-(10's complement of 40185)**

**-59815**

# Subtraction with Complements of Unsigned

- $10110010 - 10011111$  (using 2's complement)

10110010

2's complement of 10011111      +01100001

100010011  
-100000000  
000010011

Discard end carry  $2^8$

Answer

# Subtraction with Complements of Unsigned

- $10011111 - 10110010$  (using 2's complement)

10011111

2's complement of 10110010      +01001110

11101101

There is no end carry =>

-(2's complement of 11101101)

Answer = -00010011

**1010.11 – 1001.01**

**Solution:**

2's complement of 1001.01 is 0110.11. Hence

Minued -                    1 0 1 0 . 1 1

2's complement of subtrahend -                    0 1 1 0 . 1 1

Carry over    1    0 0 0 1 . 1 0

After dropping the carry over we get the result of subtraction as 1.10.

**10100.01 – 11011.10**

**Solution:**

2's complement of 11011.10 is 00100.10. Hence

Minued -                    1 0 1 0 0 . 0 1

2's complement of subtrahend -                    0 1 1 0 0 . 1 0

Result of addition -                    1 1 0 0 0 . 1 1

As there is no carry over the result of subtraction is negative and is obtained by writing the 2's complement of 11000.11.

Hence the required result is – 00111.01

# Subtraction with Complements of Unsigned

- $10110010 - 10011111$  (using 1's complement)

# Subtraction with Complements of Unsigned

- 10011111 -10110010 (using 1's complement)

# Signed Binary Numbers

- Unsigned representation can be used for positive integers
- How about negative integers?
  - Everything must be represented in binary numbers
  - Computers cannot use – or + signs

# Negative Binary Numbers

- Three different systems have been used
  1. Signed magnitude (used in ordinary arithmetic)
  2. Signed compliment (used in computer)
    - (i) One's complement
    - (ii) Two's complement (most commonly used)

**NOTE: For negative numbers the sign bit is always 1, and for positive numbers it is 0 in these three systems**

# Signed Magnitude

- The leftmost bit is the sign bit (0 is + and 1 is - ) and the remaining bits hold the absolute magnitude of the number
- Examples
  - $-47 = \textcolor{red}{1} 0101111$
  - $47 = \textcolor{red}{0} 0101111$

**For 8 bits, we can represent the signed integers  
–128 to +127**

**How about for N bits?**

# One's complement

- Replace each 1 by 0 and each 0 by 1
- Example (-6)
  - First represent 6 in binary format (00000110)
  - Then replace (11111001)

# Two's complement

- Find one's complement
- Add 1
- Example (-6)
  - First represent 6 in binary format (00000110)
  - One's complement (11111001)
  - Two's complement (11111010)

## *Signed Binary Numbers*

<b>Decimal</b>	<b>Signed-2's Complement</b>	<b>Signed-1's Complement</b>	<b>Signed Magnitude</b>
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	—	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	—	—

# Arithmetic Addition

- Usually represented by 2's complement

+ 5    00000101

+11    00001011

+16    00010000

Discard

- 5    11111011

+11    00001011

+6    **100000110**

+ 5    00000101

-11    11110101

-6    11111010

- 5    11111011

-11    11110101

-16    **111110000**

→ Discard

# Arithmetic subtraction

$$(+ \text{ or } - A) - (+B) = (+ \text{ or } - A) + (-B)$$

$$(+ \text{ or } - A) - (-B) = (+ \text{ or } - A) + (+B)$$

Example:  $(-6) - (-13)$  ?

# Binary Code

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as **binary code**. The binary code is represented by the number as well as alphanumeric letter.

# Advantages of Binary Code

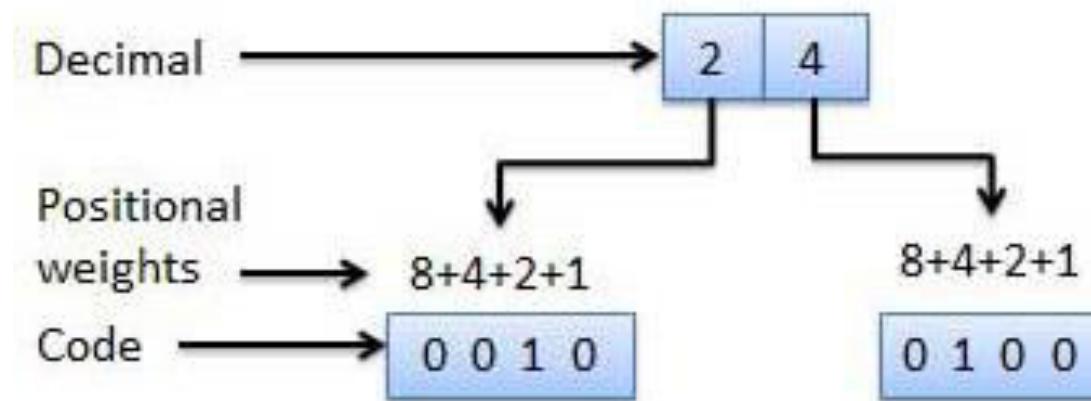
Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

# Classification of codes

**Weighted codes:** Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

**Example:** Binary, BCD, 8421, 2421



**Non-weighted codes:** In this type of binary codes, the positional weights are not assigned.

**Example:** ex-3, gray (unit distance code)

**Reflective code (self complementing):** 2421, ex-3 , 8 4 -2 -1

**Alpha numeric codes:** ASCII

**Error detection and correction codes:** Hamming, parity

## *Four Different Binary Codes for the Decimal Digits*

<b>Decimal Digit</b>	<b>BCD 8421</b>	<b>2421</b>	<b>Excess-3</b>	<b>8, 4, -2, -1</b>
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
<hr/>				
Unused bit combi- nations	1010 1011 1100 1101 1110 1111	0101 0110 0111 1000 1001 1010	0000 0001 0010 0010 1110 1111	0001 0010 0011 1100 1101 1110
<hr/>				

<b>Decimal</b>	<b>BCD</b>	<b>Gray</b>
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

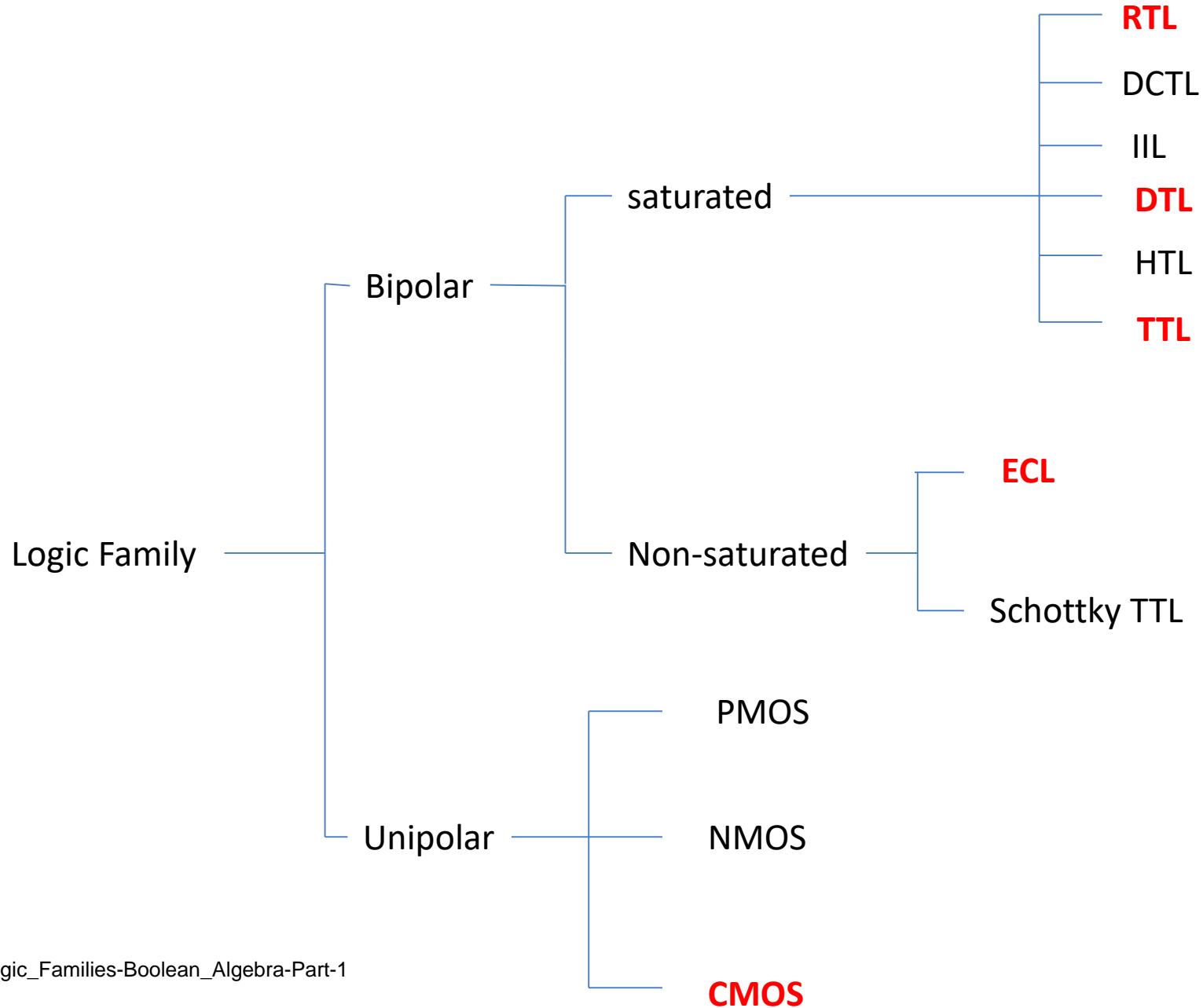
# CSE1003-Digital Logic Design

**Dr.Penchalaiah Palla**

Dept. of Micro and Nanoelectronics  
School of Electronics Engineering,VIT,  
Vellore

<b>Module:2</b>	<b>BOOLEAN ALGEBRA</b>	<b>8 hours</b>
Boolean algebra - Properties of Boolean algebra - Boolean functions - Canonical and Standard forms - Logic gates - Universal gates – Karnaugh map - Don't care conditions - Tabulation Method		

# Logic Families

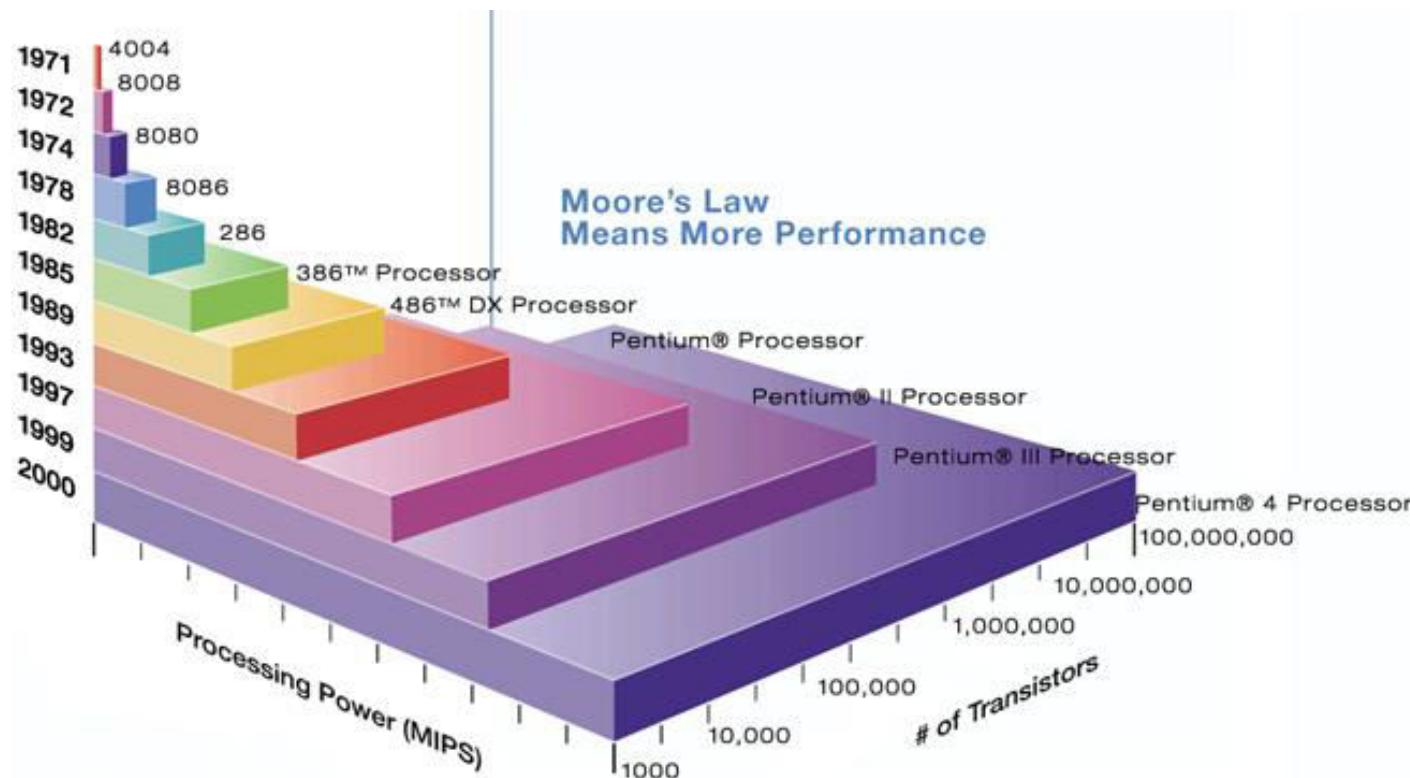


# Integration Levels

- Gate/transistor ratio is roughly 1/10
  - SSI < 12 gates/chip
  - MSI < 100 gates/chip
  - LSI ...1K gates/chip
  - VLSI ...10K gates/chip
  - ULSI ...100K gates/chip
  - GSI ...1Meg gates/chip

# Moore's law

- A prediction made by Moore (a co-founder of Intel) in 1965: "... a number of transistors to double every 2 years."

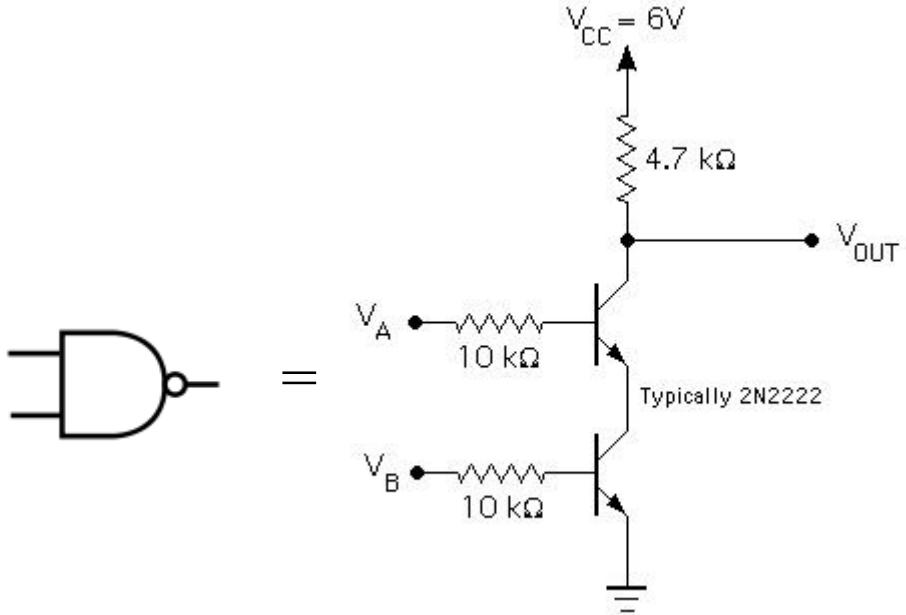


# Characteristics of Logic Families

1. Speed
2. Fan-out
3. Fan-in
4. Power dissipation
5. Propagation delay
6. Noise Margin
7. Figure of Merit = propagation delay X Power dissipation
8. Logic Swing ( $V_{OH} - V_{OL}$ )
9. Breadth : No. of functions the we can take from the circuit

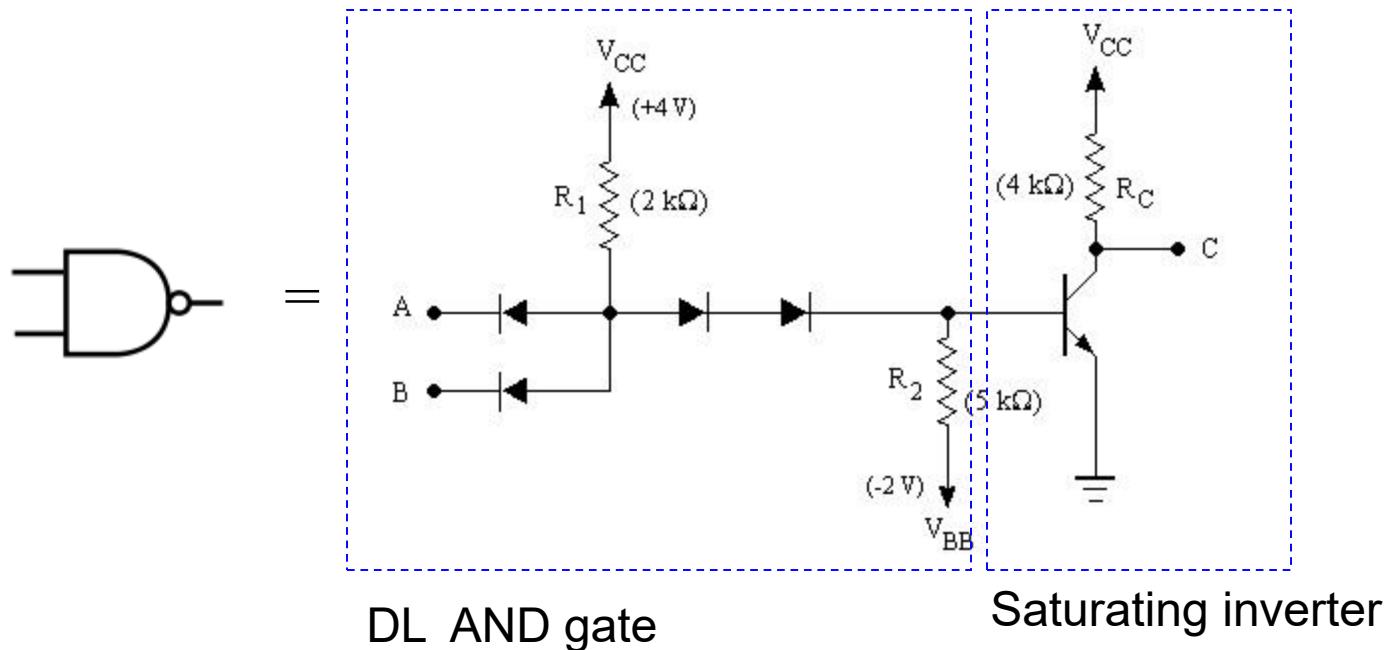
## Resistor-Transistor Logic (RTL)

- replace diode switch with a transistor switch
- can be cascaded
- large power draw



# Diode-Transistor Logic (DTL)

- essentially diode logic with transistor amplification
- reduced power consumption
- faster than RTL



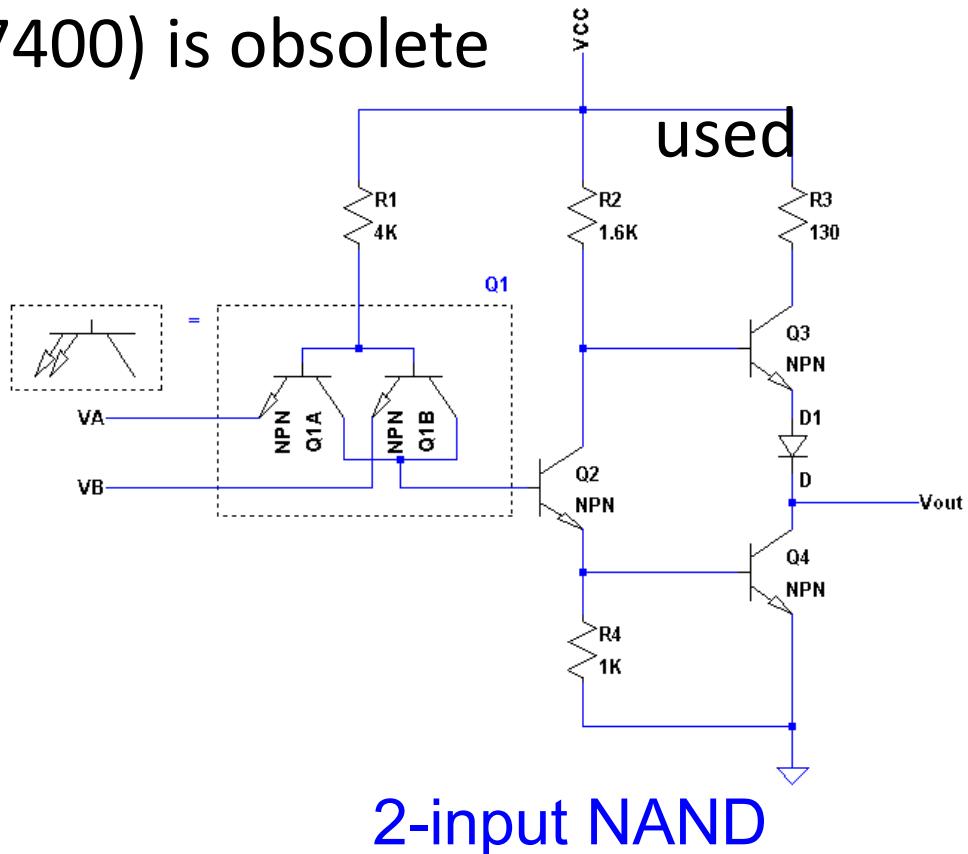
# TTL

## Bipolar Transistor-Transistor Logic (TTL)

- first introduced by in 1964 (Texas Instruments)
- TTL has shaped digital technology in many ways
- Standard TTL family (e.g. 7400) is obsolete
- Newer TTL families still  
(e.g. 74ALS00)

## Distinct features

- Multi-emitter transistors
- Totem-pole transistor arrangement



# Emitter-Coupled Logic (ECL)

- PROS: Fastest logic family available ( $\sim 1\text{ns}$ )
- CONS: low noise margin and high power dissipation
- Operated in emitter coupled geometry (recall differential amplifier or emitter-follower), transistors are biased and operate near their Q-point (never near saturation!)
- Logic levels. “0”:  $-1.7\text{V}$ . “1”:  $-0.8\text{V}$
- Such strange logic levels require extra effort when interfacing to TTL/CMOS logic families.

# Boolean Algebra

In ordinary algebra, the letter symbols take any number of values. In Boolean algebra, they take two values, i.e. 0 and 1. ... The values assigned to a variable have a numerical significance in ordinary algebra, whereas in Boolean algebra they have a logical significance.

# The “WHY” slide

- Boolean Algebra
  - When we learned numbers like 1, 2, 3, we also then learned how to add, multiply, etc. with them. Boolean Algebra covers operations that we can do with 0's and 1's. Computers do these operations ALL THE TIME and they are basic building blocks of computation inside your computer program.
- Axioms, laws, theorems
  - We need to know some rules about how those 0's and 1's can be operated on together. There are similar axioms to decimal number algebra, and there are some laws and theorems that are good for you to use to simplify your operation.

a statement or proposition which is regarded as being established, accepted,  
or self-evidently true."the axiom that sport builds character"

# How does Boolean Algebra fit into the big picture?

- It is part of the Combinational Logic topics (memoryless)
  - Different from the Sequential logic topics (can store information)
- Learning Axioms and theorems of Boolean algebra
  - Allows you to design logic functions
  - Allows you to know how to combine different logic gates
  - Allows you to simplify or optimize on the complex operations

# Basic Definitions

- Binary Operators

- AND

$$z = x \bullet y = x \cdot y$$

$z=1$  if  $x=1$  AND  $y=1$

- OR

$$z = x + y$$

$z=1$  if  $x=1$  OR  $y=1$

- NOT

$$z = \overline{x} = x'$$

$z=1$  if  $x=0$

- Boolean Algebra

- Binary Variables: only ‘0’ and ‘1’ values
  - Algebraic Manipulation

# Boolean Algebra Postulates

- Commutative Law

$$x \bullet y = y \bullet x$$

$$x + y = y + x$$

- Identity Element

$$x \bullet 1 = x$$

$$x + 0 = x$$

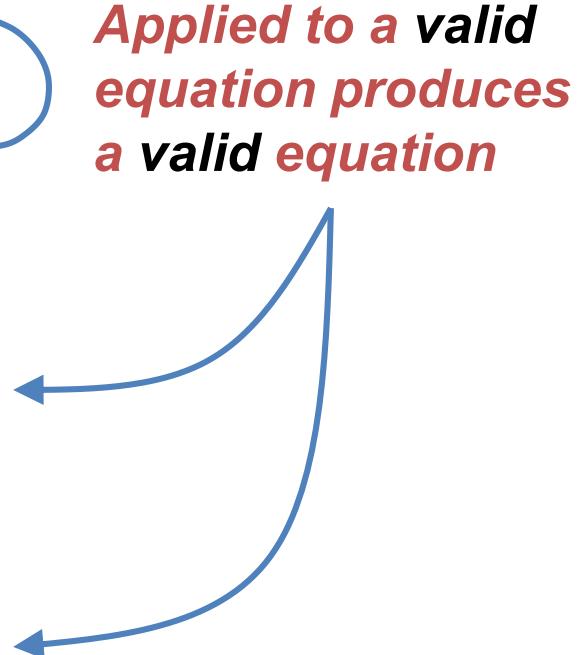
- Complement

$$x \bullet x' = 0$$

$$x + x' = 1$$

# Boolean Algebra Theorems

- Duality
  - The *dual* of a Boolean algebraic expression is obtained by interchanging the **AND** & **OR** operators and replacing the **1**'s by **0**'s and the **0**'s by **1**'s.
  - $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$
  - $x + (y \bullet z) = (x + y) \bullet (x + z)$
- Theorem 1
  - $x \bullet x = x$
  - $x + x = x$
- Theorem 2
  - $x \bullet 0 = 0$
  - $x + 1 = 1$



# Boolean Algebra Theorems

- Theorem 3: *Involution*
  - $(x')' = x$                                   $(\overline{x}) = x$
- Theorem 4: *Associative & Distributive*
  - $(x \bullet y) \bullet z = x \bullet (y \bullet z)$     $(x + y) + z = x + (y + z)$
  - $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$   
 $x + (y \bullet z) = (x + y) \bullet (x + z)$
- Theorem 5: *DeMorgan*
  - $(x \bullet y)' = x' + y'$                                   $(x + y)' = x' \bullet y'$
  - $(\overline{x \bullet y}) = \overline{x} + \overline{y}$                                   $(\overline{x + y}) = \overline{x} \bullet \overline{y}$
- Theorem 6: *Absorption*
  - $x \bullet (x + y) = x$                                   $x + (x \bullet y) = x$

# Operator Precedence

- Parentheses

$$(\dots) \bullet (\dots)$$

$$x [y + z \overline{(w+x)}]$$

- NOT

$$x' + y$$

$$(w+x)$$

$$\overline{(w+x)}$$

- AND

$$x + x \bullet y$$

$$z \overline{(w+x)}$$

$$y + z \overline{(w+x)}$$

- OR

$$x [y + z \overline{(w+x)}]$$

# DeMorgan's Theorem

$$\overline{a [b + c (\bar{d} + \bar{e})]}$$

$$\overline{\bar{a} + [\bar{b} + \bar{c} (\bar{d} + \bar{e})]}$$

$$\overline{\bar{a} + \bar{b} (\bar{c} (\bar{d} + \bar{e}))}$$

$$\overline{\bar{a} + \bar{b} (\bar{c} + (\bar{d} + \bar{e}))}$$

$$\overline{\bar{a} + \bar{b} (\bar{c} + (\bar{d} \bar{e}))} =$$

$$\overline{\bar{a} + \bar{b} (\bar{c} + \bar{d} e)}$$



# Useful laws and theorems

Identity:	$X + 0 = X$	Dual: $X \bullet 1 = X$
Null:	$X + 1 = 1$	Dual: $X \bullet 0 = 0$
Idempotent:	$X + X = X$	Dual: $X \bullet X = X$
Involution:	$(X')' = X$	
Complementarity:	$X + X' = 1$	Dual: $X \bullet X' = 0$
Commutative:	$X + Y = Y + X$	Dual: $X \bullet Y = Y \bullet X$
Associative:	$(X+Y)+Z=X+(Y+Z)$	Dual: $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$
Distributive:	$X \bullet (Y+Z) = (X \bullet Y) + (X \bullet Z)$	Dual: $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$
Uniting:	$X \bullet Y + X \bullet Y' = X$	Dual: $(X+Y) \bullet (X+Y')=X$

# Useful laws and theorems (con't)

Absorption:  $X+X \bullet Y = X$

Dual:  $X \bullet (X+Y) = X$

Absorption (#2):  $(X+Y') \bullet Y = X \bullet Y$

Dual:  $(X \bullet Y') + Y = X + Y$

de Morgan's:  $(X+Y+\dots)' = X' \bullet Y' \bullet \dots$

Dual:  $(X \bullet Y \bullet \dots)' = X' + Y' + \dots$

Duality:  $(X+Y+\dots)^D = X \bullet Y \bullet \dots$

Dual:  $(X \bullet Y \bullet \dots)^D = X + Y + \dots$

Multiplying & factoring:  $(X+Y) \bullet (X'+Z) = X \bullet Z + X' \bullet Y$

Dual:  $X \bullet Y + X' \bullet Z = (X+Z) \bullet (X'+Y)$

Consensus:  $(X \bullet Y) + (Y \bullet Z) + (X' \bullet Z) = X \bullet Y + X' \bullet Z$

Dual:

$(X+Y) \bullet (Y+Z) \bullet (X'+Z) = (X+Y) \bullet (X'+Z)$

# Proving theorems

- Example 1: Prove the uniting theorem--  
 $X \bullet Y + X \bullet Y' = X$

Distributive

$$X \bullet Y + X \bullet Y' = X \bullet (Y + Y')$$

Complementarity

$$= X \bullet (1)$$

Identity

$$= X$$

- Example 2: Prove the absorption theorem--  
 $X + X \bullet Y = X$

Identity

$$X + X \bullet Y = (X \bullet 1) + (X \bullet Y)$$

Distributive

$$= X \bullet (1 + Y)$$

Null

$$= X \bullet (1)$$

Identity

$$= X$$

# Proving theorems

- Example 3: Prove the consensus theorem--

$$(XY) + (YZ) + (X'Z) = XY + X'Z$$

Complementarity  $XY + YZ + X'Z = XY + (X + X')YZ + X'Z$

Distributive  $= XYZ + XY + X'YZ + X'Z$

- Use absorption  $\{AB + A = A\}$  with  $A = XY$  and  $B = Z$

$$= XY + X'YZ + X'Z$$

Rearrange terms  $= XY + X'ZY + X'Z$

- Use absorption  $\{AB + A = A\}$  with  $A = X'Z$  and  $B = Y$

$$XY + YZ + X'Z = XY + X'Z$$

# Boolean vs. Ordinary

- Boolean Algebra
  - Associate law not included (but still valid)
  - Distributive law is valid
  - No additive or multiplicative inverses
  - Define complement
  - No. of elements is not clearly defined
    - 2 for two-valued Boolean algebra
- Ordinary Algebra
  - Associate law included
  - Distributive law may not valid
  - Have additive and multiplicative inverses
  - No complement
  - Operator Deal with real numbers
    - Infinite set of elements

# Boolean Functions

- Boolean Expression

Example:

$$F = x + y' z$$

- Truth Table

All possible combinations  
of input variables

- Logic Circuit



$x$	$y$	$z$	$F$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

# Algebraic Manipulation

- *Literal:*  
A single variable within a term that may be complemented or not.
- Use Boolean Algebra to simplify Boolean functions to produce simpler circuits

*Example:* Simplify to a minimum number of literals

$$F = x + x' y \quad (\text{3 Literals})$$

$$= x + (x' y)$$

$$= (x + x') (x + y) \quad \text{Distributive law (+ over •)}$$

$$= (1) (x + y) = x + y \quad (\text{2 Literals})$$

# Complement of a Function

- DeMorgan's Theorem

$$F = A + B + C$$

$$\overline{F} = \overline{\overline{A} + \overline{B} + \overline{C}}$$

$$\overline{F} = \overline{A} \cdot \overline{B} \cdot \overline{C}$$

- Duality & Literal Complement

$$F = A + B + C$$

$$F = \overline{A} \cdot \overline{B} \cdot \overline{C}$$

$$\overline{F} = \overline{A} \cdot \overline{B} \cdot \overline{C}$$

# Canonical Forms

- Minterm
  - Product (AND function)
  - Contains all variables
  - Evaluates to ‘1’ for a specific combination

*Example*

$$\begin{aligned}
 A = 0 & \quad \overline{A} \quad \overline{B} \quad \overline{C} \\
 B = 0 & \quad (\overline{0}) \bullet (\overline{0}) \bullet (\overline{0}) \\
 C = 0 & \quad \downarrow \quad \downarrow \quad \downarrow \\
 & \quad 1 \cdot 1 \cdot 1 = 1
 \end{aligned}$$

	A	B	C	Minterm
0	0	0	0	$m_0$ $\overline{A}\overline{B}\overline{C}$
1	0	0	1	$m_1$ $\overline{A}\overline{B}C$
2	0	1	0	$m_2$ $\overline{A}BC$
3	0	1	1	$m_3$ $\overline{A}B\overline{C}$
4	1	0	0	$m_4$ $A\overline{B}\overline{C}$
5	1	0	1	$m_5$ $A\overline{B}C$
6	1	1	0	$m_6$ $ABC$
7	1	1	1	$m_7$ $A\overline{B}\overline{C}$

# Canonical Forms

- Maxterm
  - Sum (*OR* function)
  - Contains all variables
  - Evaluates to ‘0’ for a specific combination

*Example*

$$\begin{aligned}
 A &= 1 & \bar{A} & \bar{B} & \bar{C} \\
 B &= 1 & (\bar{1}) & + & (\bar{1}) & + & (\bar{1}) \\
 C &= 1 & \downarrow & \downarrow & \downarrow \\
 & & 0 & + & 0 & + & 0 = 0
 \end{aligned}$$

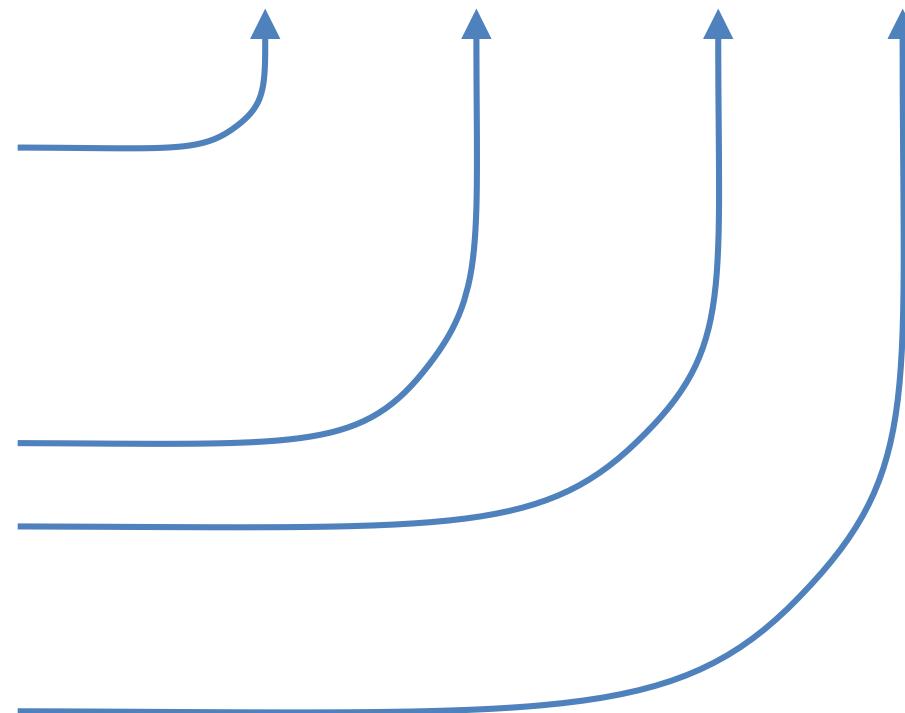
	A	B	C	Maxterm
0	0	0	0	$M_0$ $A + B + C$
1	0	0	1	$M_1$ $A + B + \bar{C}$
2	0	1	0	$M_2$ $A + \bar{B} + C$
3	0	1	1	$M_3$ $A + \bar{B} + \bar{C}$
4	1	0	0	$M_4$ $\bar{A} + B + C$
5	1	0	1	$M_5$ $\bar{A} + B + \bar{C}$
6	1	1	0	$M_6$ $\bar{A} + \bar{B} + C$
7	1	1	1	$M_7$ $\bar{A} + \bar{B} + \bar{C}$

# Canonical Forms

- Truth Table *to* Boolean Function

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

$$F = \overline{A}\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C + ABC$$



# Standard Forms

- Sum of Products (SOP)

$$F = \overline{A}\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C + ABC$$
$$\begin{aligned} &\rightarrow A\overline{B}(\overline{C} + C) \\ &= A\overline{B}(1) \\ &= A\overline{B} \\ &\rightarrow AC(\overline{B} + B) \\ &= AC \\ &\rightarrow \overline{B}C(\overline{A} + A) \\ &= \overline{B}C \end{aligned}$$

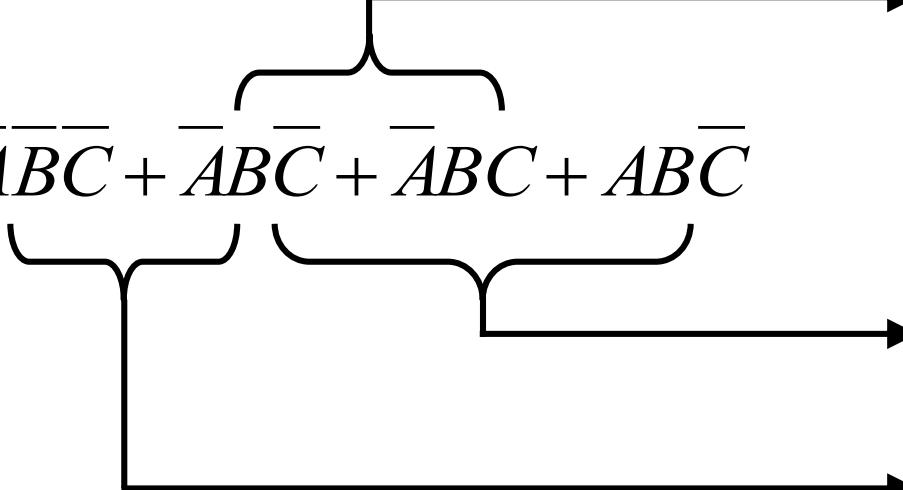
$$F = \overline{B}C(\overline{A} + A) + A\overline{B}(\overline{C} + C) + AC(\overline{B} + B)$$

$$F = \overline{B}C + A\overline{B} + AC$$

# Standard Forms

- Product of Sums (POS)

$$\overline{F} = \overline{\overline{ABC}} + \overline{\overline{ABC}} + \overline{\overline{ABC}} + \overline{\overline{ABC}}$$



$$\overline{F} \rightarrow \overline{BC}(\overline{A} + A)$$
$$\overline{F} \rightarrow \overline{AC}(\overline{B} + B)$$

$$\overline{F} = \cancel{\overline{AC}(\overline{B} + B)} + \cancel{\overline{AB}(\overline{C} + C)} + \cancel{B\overline{C}(\overline{A} + A)}$$

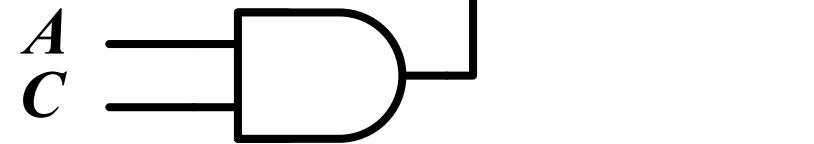
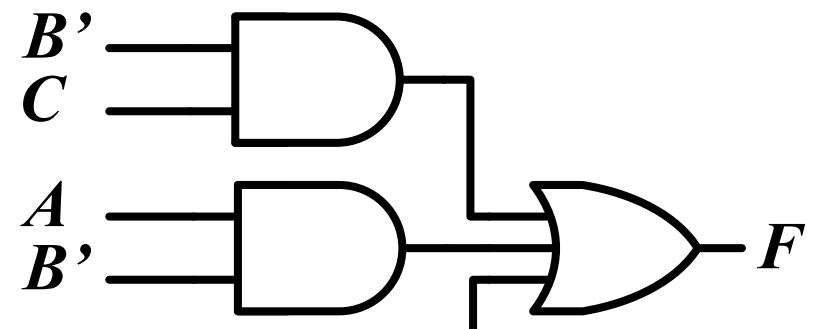
$$\overline{F} = \overline{\overline{AC} + \overline{AB} + B\overline{C}}$$

$$F = (A + C)(A + \overline{B})(\overline{B} + C)$$

# Two-Level Implementations

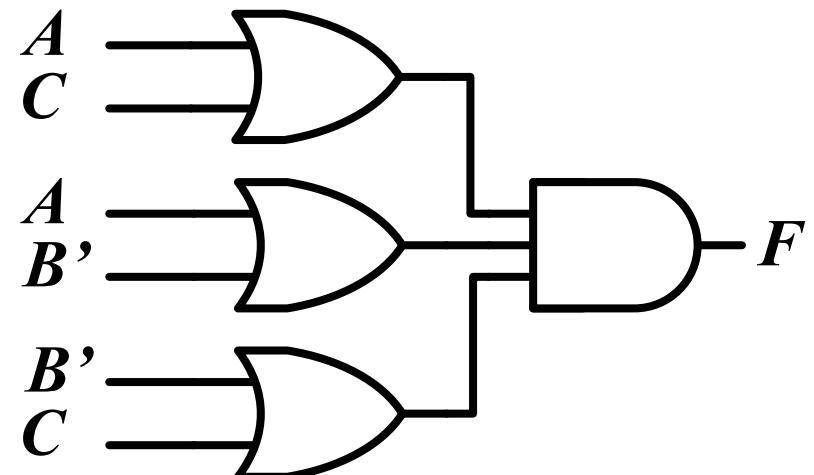
- Sum of Products (SOP)

$$F = \overline{B}C + A\overline{B} + AC$$



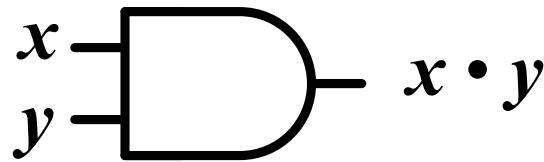
- Product of Sums (POS)

$$F = (A+C)(A+\overline{B})(\overline{B}+C)$$



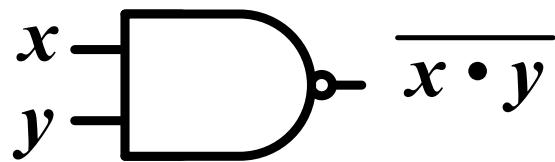
# Logic Operators

- AND



$x$	$y$	<i>AND</i>
0	0	0
0	1	0
1	0	0
1	1	1

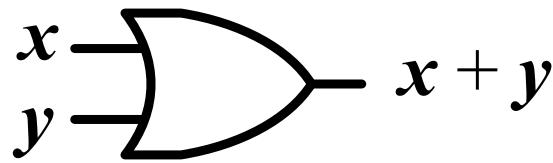
- NAND (Not AND)



$x$	$y$	<i>NAND</i>
0	0	1
0	1	1
1	0	1
1	1	0

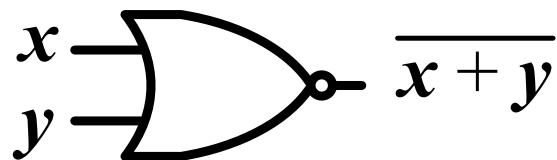
# Logic Operators

- OR



$x$	$y$	$OR$
0	0	0
0	1	1
1	0	1
1	1	1

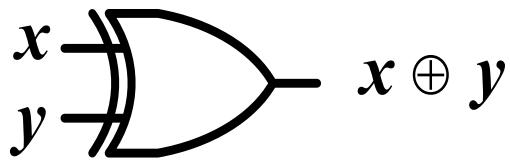
- NOR (Not OR)



$x$	$y$	$NOR$
0	0	1
0	1	0
1	0	0
1	1	0

# Logic Operators

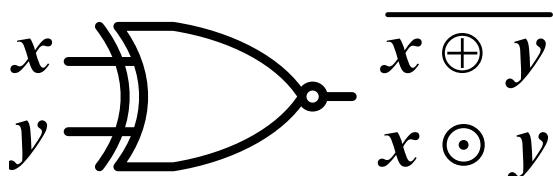
- XOR (Exclusive-OR)



$$\bar{x}y + x\bar{y}$$

$x$	$y$	$XOR$
0	0	0
0	1	1
1	0	1
1	1	0

- XNOR (Exclusive-NOR)  
(Equivalence)

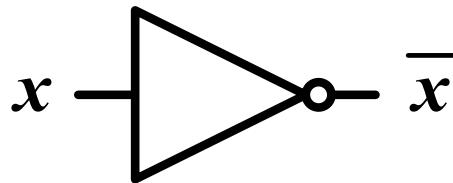


$$\bar{x}\bar{y} + xy$$

$x$	$y$	$XNOR$
0	0	1
0	1	0
1	0	0
1	1	1

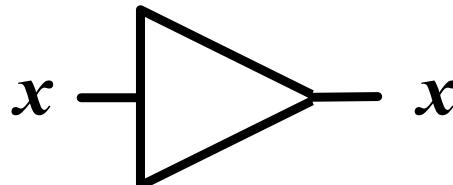
# Logic Operators

- NOT (Inverter)



$x$	<i>NOT</i>
0	1
1	0

- Buffer

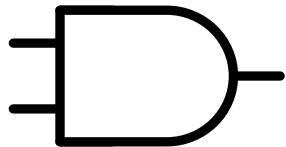


$x$	<i>Buffer</i>
0	0
1	1

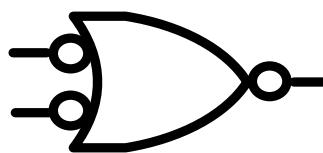
# DeMorgan's Theorem on Gates

- AND Gate

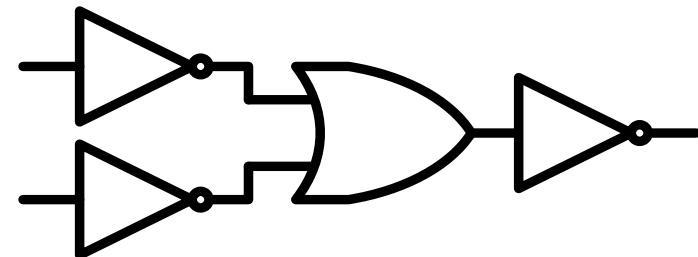
$$F = x \cdot y$$



$$\bar{F} = \overline{(x \cdot y)}$$

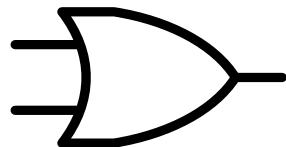


$$\bar{F} = \overline{x} + \overline{y}$$

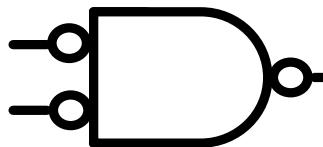


- OR Gate

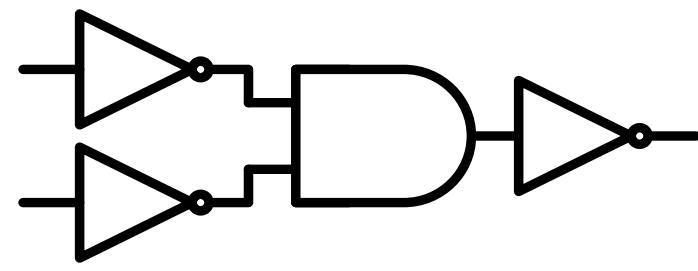
$$F = x + y$$



$$F = \overline{(x + y)}$$



$$F = \overline{x} \cdot \overline{y}$$



# Homework

- Mano

**2-4** Reduce the following Boolean expressions to the indicated number of literals:

(a)  $A'C' + ABC + AC'$  to three literals

(b)  $(x'y' + z)' + z + xy + wz$  to three literals

(c)  $A'B(D' + C'D) + B(A + A'CD)$  to one literal

(d)  $(A' + C)(A' + C')(A + B + C'D)$  to four literals

**2-5** Find the complement of  $F = x + yz$ ; then show that  $FF' = 0$  and  $F + F' = 1$

# Homework

**2-6** Find the complement of the following expressions:

(a)  $xy' + x'y$

(b)  $(AB' + C)D' + E$

(c)  $(x + y' + z)(x' + z')(x + y)$

**2-8** List the truth table of the function:

$$F = xy + xy' + y'z$$

**2-9** Logical operations can be performed on strings of bits by considering each pair of corresponding bits separately (this is called bitwise operation). Given two 8-bit strings  $A = 10101101$  and  $B = 10001110$ , evaluate the 8-bit result after the following logical operations: (a) AND, (b) OR, (c) XOR, (d) NOT  $A$ , (e) NOT  $B$ .

# Homework

**2-10** Draw the logic diagrams for the following Boolean expressions:

(a)  $Y = A'B' + B(A + C)$

(b)  $Y = BC + AC'$

(c)  $Y = A + CD$

(d)  $Y = (A + B)(C' + D)$

**2-12** Simplify the Boolean function  $T_1$  and  $T_2$  to a minimum number of literals.

A	B	C	$T_1$	$T_2$
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

# Homework

**2-15 Given the Boolean function**

$$F = xy'z + x'y'z + w'xy + wx'y + wxy$$

- (a) Obtain the truth table of the function.**
- (b) Draw the logic diagram using the original Boolean expression.**
- (c) Simplify the function to a minimum number of literals using Boolean algebra.**
- (d) Obtain the truth table of the function from the simplified expression and show that it is the same as the one in part (a)**
- (e) Draw the logic diagram from the simplified expression and compare the total number of gates with the diagram of part (b).**

# Homework

**2-18** Convert the following to the other canonical form:

(a)  $F(x, y, z) = \sum (1, 3, 7)$

(b)  $F(A, B, C, D) = \prod (0, 1, 2, 3, 4, 6, 12)$

**2-19** Convert the following expressions into sum of products and product of sums:

(a)  $(AB + C)(B + C'D)$

(b)  $x' + x(x + y')(y + z')$

# Circuit Optimization

---

- ✓ Goal: To obtain the simplest implementation for a given function
- ✓ Optimization is a more formal approach to simplification that is performed using a specific procedure or algorithm
- ✓ Optimization requires a cost criterion to measure the simplicity of a circuit
- ✓ Distinct cost criteria we will use:
  - Literal cost ( $L$ )
  - Gate input cost ( $G$ )
  - Gate input cost with NOTs ( $GN$ )

# Literal Cost

---

- ✓ Literal - a variable or its complement
- ✓ Literal cost - the number of literal appearances in a Boolean expression corresponding to the logic circuit diagram
- ✓ Example, which solution is best?
  - $F = BD + A\bar{B}C + A\bar{C}\bar{D}$   $L = 8$
  - $F = BD + ABC + ABD + \underline{ABC}$   $L = 11$
  - $F = (A + B)(A + D)(B + C + D)(\bar{B} + \bar{C} + D)$   $L = 10$

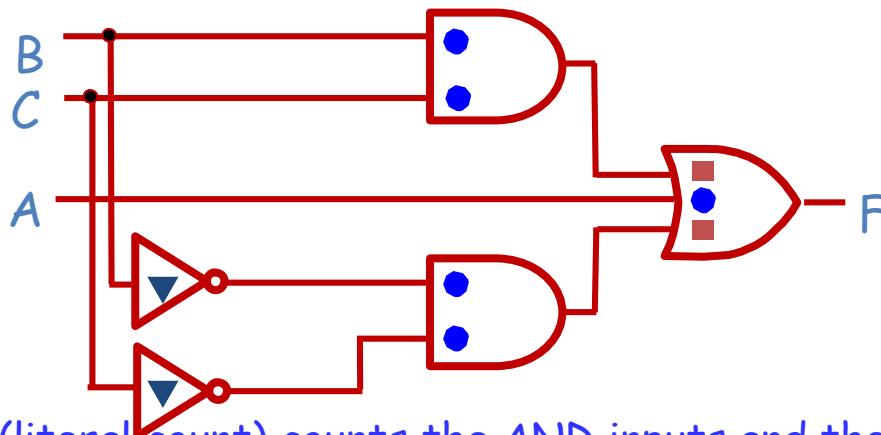
# Gate Input Cost

---

- ✓ Gate input costs - the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations. ( $G$  - inverters not counted,  $GN$  - inverters counted)
- ✓ For SOP and POS equations, it can be found from the equation(s) by finding the sum of:
  - all literal appearances
  - the number of terms, ( $G$ ) and
  - optionally, the number of distinct complemented single literals ( $GN$ ).
- ✓ Example, which solution is best?
  - $F = BD + \underline{A}\underline{\bar{B}}C + A\underline{\bar{C}}\underline{\bar{D}}$   $G = 11, GN = 14$
  - $F = BD + \underline{\bar{A}}\underline{\bar{B}}C + AB\bar{D} + A\underline{\bar{B}}\underline{\bar{C}}$   $G = 15, GN = 18$
  - $F = (A + B)(A + D)(B + C + D)(B + C + D)$   $G = 14, GN = 17$

# Cost Criteria (continued)

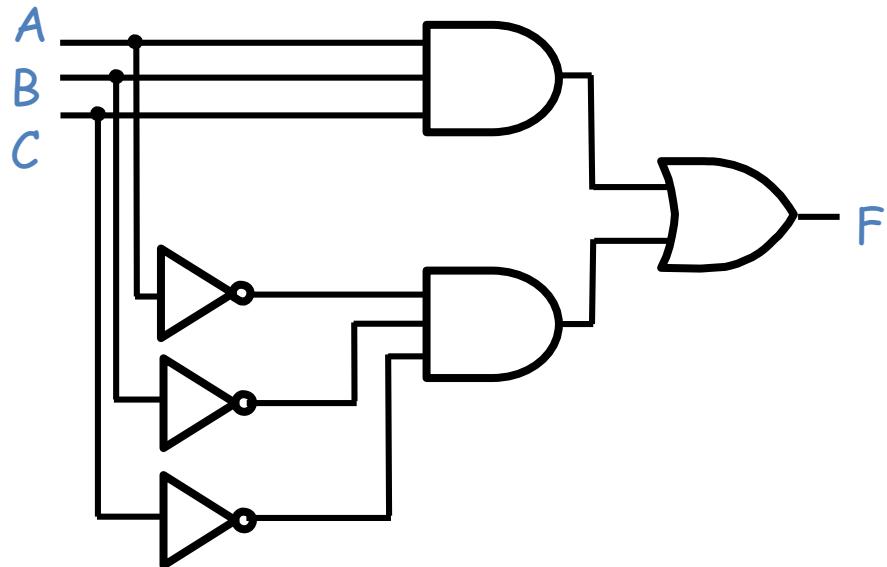
- ✓ Example 1:  
 $F = A + \overline{B}C + \overline{\overline{B}}\overline{C}$   $GN = G + 2 = 9$
- ✓  $L = 5$   
 $G = L + 2 = 7$



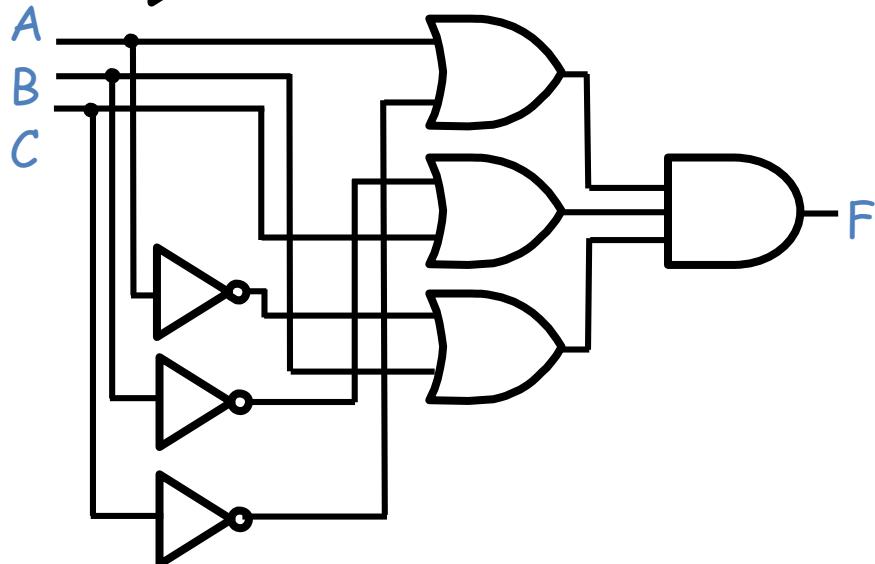
- L (literal count) counts the AND inputs and the single literal OR input.
- G (gate input count) adds the remaining OR gate inputs
- ▼ GN(gate input count with NOTs) adds the inverter inputs

# Cost Criteria (continued)

- $F = ABC + \bar{A}\bar{B}\bar{C}$
- $L = 6 \ G = 8 \ GN = 11$



- $F = (A + \bar{C})(\bar{B} + C)(\bar{A} + B)$
- $L = 6 \ G = 9 \ GN = 12$
- ✓ Same function and same literal cost
- ✓ But first circuit has better gate input count and better gate input count with NOTs



# Boolean Function Optimization

---

- ✓ Minimizing the gate input (or literal) cost of a (a set of) Boolean equation(s) reduces circuit cost.
- ✓ We choose gate input cost.
- ✓ Boolean Algebra and graphical techniques are tools to minimize cost criteria values.
- ✓ Some important questions:
  - When do we stop trying to reduce the cost?
  - Do we know when we have a minimum cost?
- ✓ Treat optimum or near-optimum cost functions for two-level (SOP and POS) circuits first.
- ✓ Introduce a graphical technique using Karnaugh maps (K-maps, for short)

# Karnaugh Maps (K-map)

---

- ✓ A K-map is a collection of squares
  - Each square represents a minterm
  - The collection of squares is a graphical representation of a Boolean function
  - Adjacent squares differ in the value of one variable
  - Alternative algebraic expressions for the same function are derived by recognizing patterns of squares (corresponding to cubes)
- ✓ The K-map can be viewed as

A reorganized version of the truth table or a particular cube representation

# Some Uses of K-Maps

---

✓ Provide a means for:

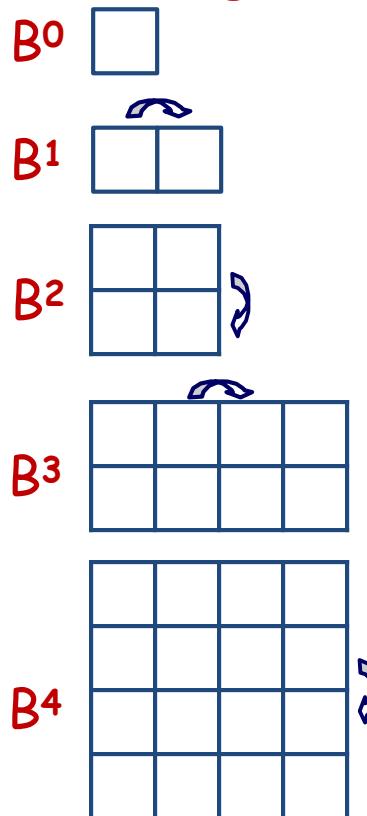
- Finding optimum
  - SOP and POS standard forms, and
  - two-level AND/OR and OR/AND circuit implementations
- Visualizing concepts related to manipulating Boolean expressions
- Demonstrating concepts used by computer-aided design programs to simplify large circuits

# The Boolean Space $B^n$

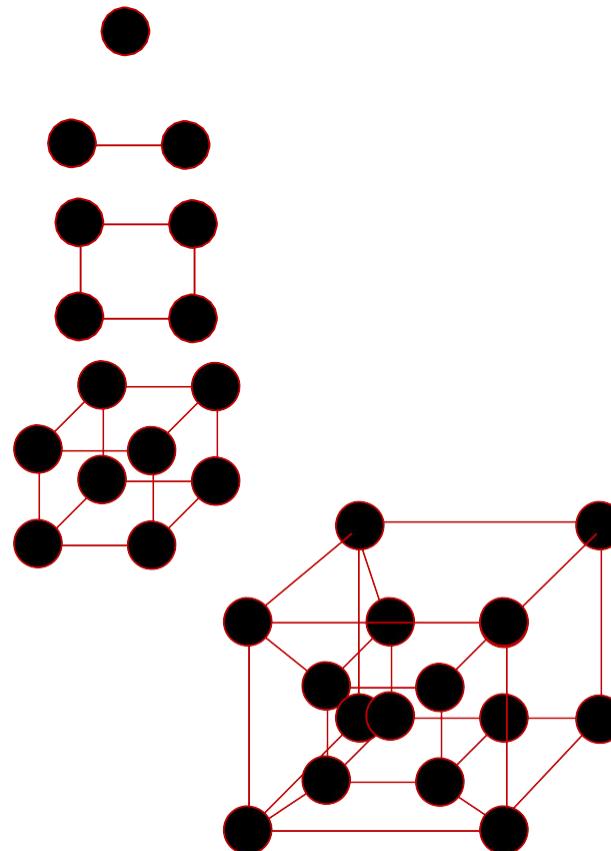
---

- ✓  $B = \{0, 1\}$
- ✓  $B^2 = \{0, 1\} \times \{0, 1\} = \{00, 01, 10, 11\}$

Karnaugh Maps:



Boolean Cubes:



# Two Variable Maps

## ✓ A 2-variable Karnaugh Map:

- Note that minterm  $m_0$  and minterm  $m_1$  are “adjacent” and differ in the value of the variable  $y$
- Similarly, minterm  $m_0$  and minterm  $m_2$  differ in the  $x$  variable.
- Also,  $m_1$  and  $m_3$  differ in the  $x$  variable as well.
- Finally,  $m_2$  and  $m_3$  differ in the value of the variable  $y$

$x \backslash y$	$y = 0$	$y = 1$
$x = 0$	$m_0$	$m_1$
$x = 1$	$m_2$	$m_3$

# K-Map and Truth Tables

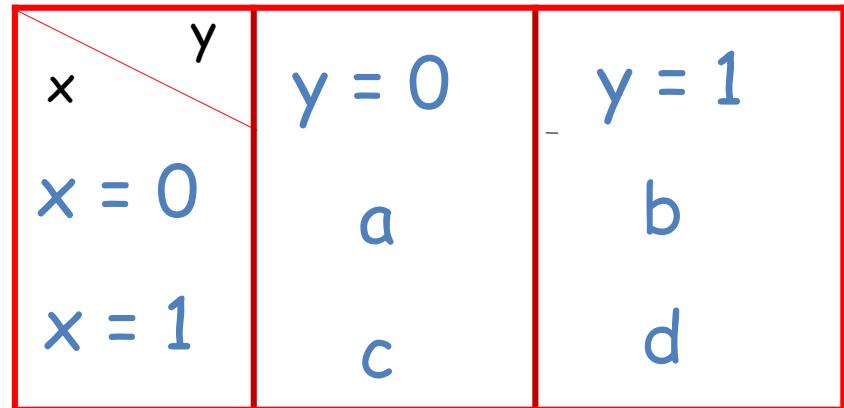
---

- ✓ The K-Map is just a different form of the truth table.
- ✓ Example - Two variable function:
  - We choose a,b,c and d from the set {0,1} to implement a particular function,  $F(x,y)$ .

Function Table

Input Values $(x,y)$	Function Value $F(x,y)$
0 0	a
0 1	b
1 0	c
1 1	d

K-Map



# K-Map Function Representation

---

- ✓ Example:  $F(x,y) = x$

$F = x$	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	1	1

- ✓ For function  $F(x,y)$ , the two adjacent cells containing 1's can be combined using the Minimization Theorem:

$$F(x,y) = x\bar{y} + xy = x$$

# K-Map Function Representation

- ✓ Example:  $G(x,y) = \bar{x}\bar{y} + x\bar{y} + xy$

$G=x+y$	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	1

- ✓ For  $G(x,y)$ , two pairs of adjacent cells containing 1's can be combined using the Minimization Theorem:

$$G(x,y) = (\bar{x}\bar{y} + x\bar{y}) + (\bar{x}y + xy) = x + y$$

Duplicate  $xy$

# Three Variable Maps

- ✓ A three-variable K-map:

<del>yz</del>	$yz=00$	$yz=01$	$yz=11$	$yz=10$
<del>x</del>	$m_0$	$m_1$	$m_3$	$m_2$
$x=0$				
$x=1$	$m_4$	$m_5$	$m_7$	$m_6$

- ✓ Where each minterm corresponds to the product terms:

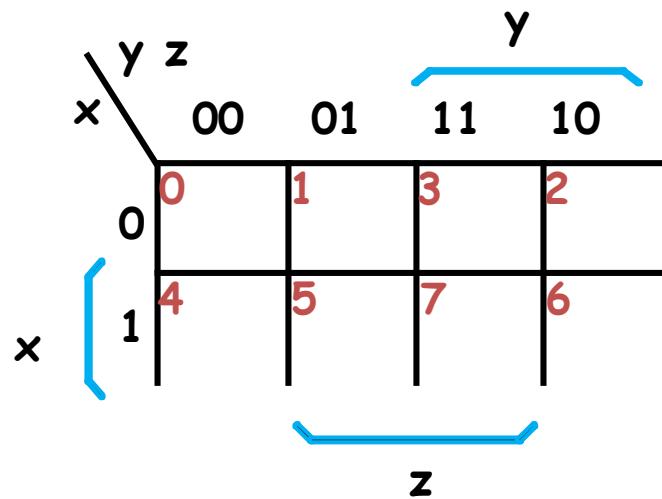
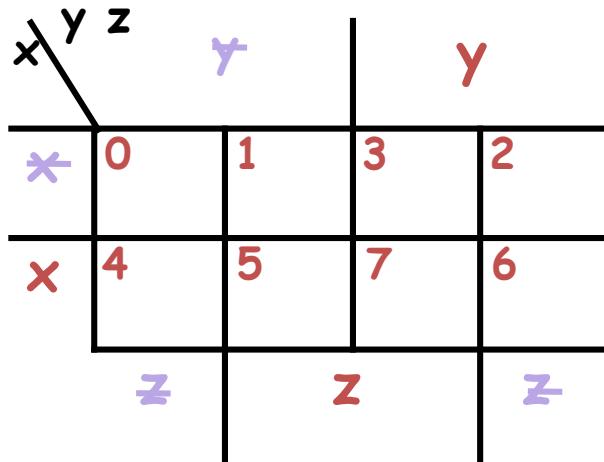
<del>yz</del>	$yz=00$	$yz=01$	$yz=11$	$yz=10$
<del>x</del>				
$x=0$	$\bar{x}\bar{y}z$	$\bar{x}yz$	$\bar{x}yz$	$\bar{x}yz$
$x=1$	$\bar{x}yz$	$\bar{x}yz$	$xyz$	$xyz$

- ✓ Note that if the binary value for an index differs in one bit position, the minterms are adjacent on the K-Map

# Alternative Map Labeling

---

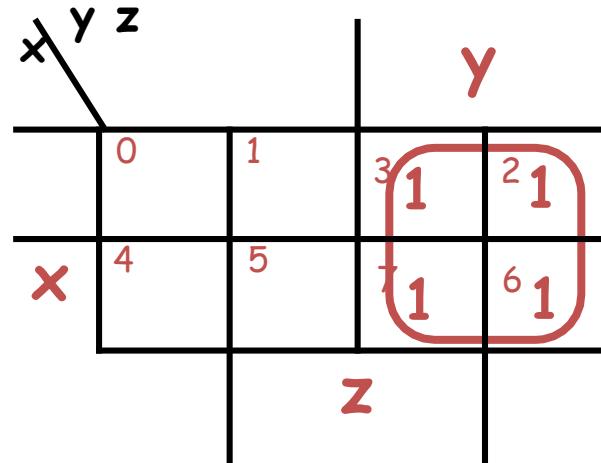
- ✓ Map use largely involves:
  - Entering values into the map, and
  - Reading off product terms from the map.
- ✓ Alternate labelings are useful:



# Example: Combining Squares

✓ Example: Let

✓  $F(x, y, z) = \sum_m (2, 3, 6, 7)$



✓ Applying the Minimization Theorem three times:

$$\begin{aligned} F(x, y, z) &= \bar{x}yz + xy\bar{z} + \bar{x}y\bar{z} + xy\bar{z} \\ &= yz + y\bar{z} \\ &= y \end{aligned}$$

✓ Thus the four terms that form a  $2 \times 2$  square correspond to the term "y".

# Combining Squares

---

- ✓ By combining squares, we reduce number of literals in a product term, reducing the literal cost, thereby reducing the other two cost criteria
- ✓ On a 3-variable K-Map:
  - One square represents a minterm with three variables
  - Two adjacent squares represent a cube that is product term with two variables
  - Four "adjacent" terms represent a cube that is product term with one variable
  - Eight "adjacent" terms is the function of all ones (no variables) is a tautology  $f^1=1$ .

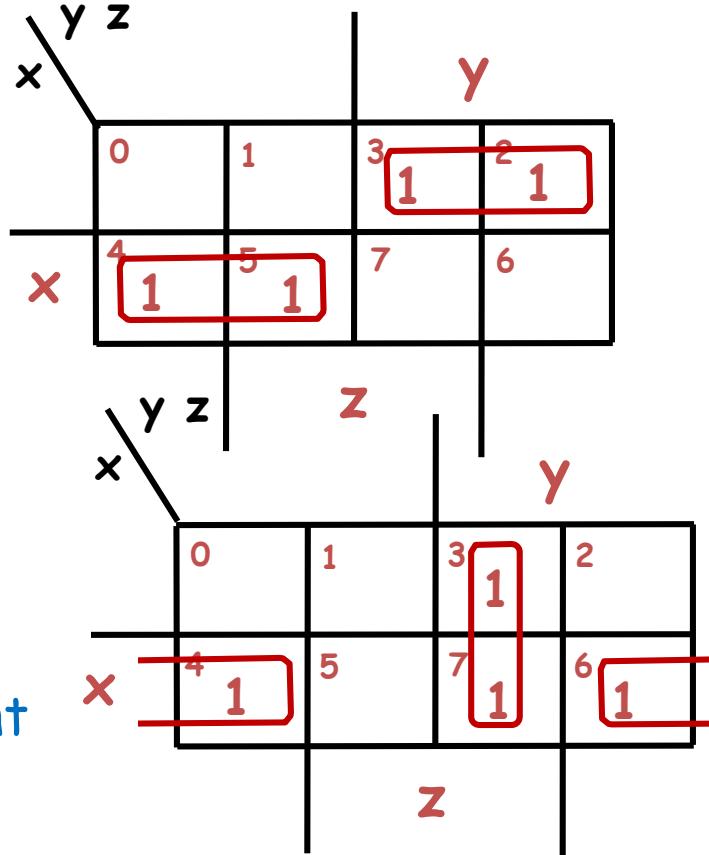
# Example Functions

- ✓ By convention, we represent the minterms of  $F$  by a "1" in the map and leave the minterms of blank  $\bar{F}$
- ✓ Example:

$$F(x, y, z) = \sum_m (2, 3, 4, 5)$$

$$F(x, y, z) = \bar{x}y + x\bar{y}$$

- ✓ Example:
- ✓  $F(x, y, z) = \sum_m (3, 4, 6, 7)$
- ✓  $F(x, y, z) = yz + x\bar{z}$
- ✓ Learn the locations of the 8 indices based on the variable order shown (x, most significant and z, least significant) on the map boundaries



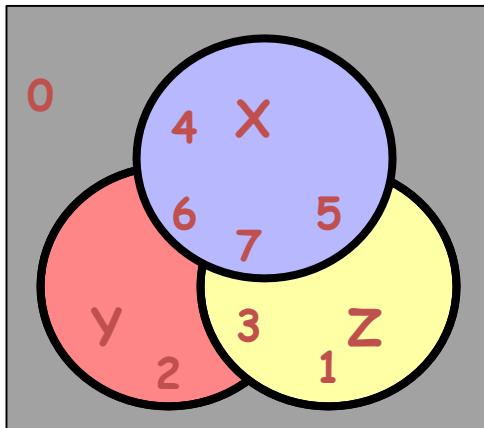
# Three-Variable Maps

---

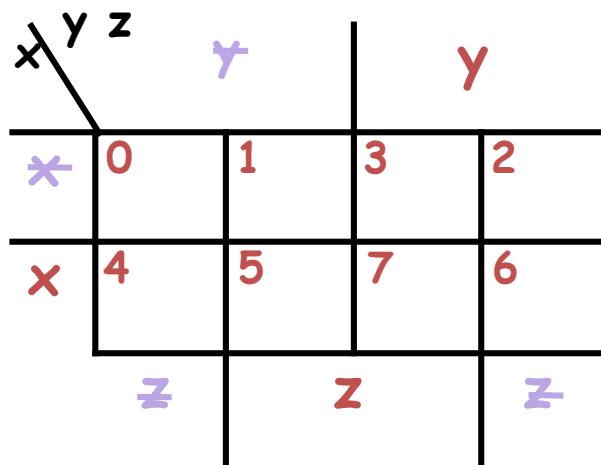
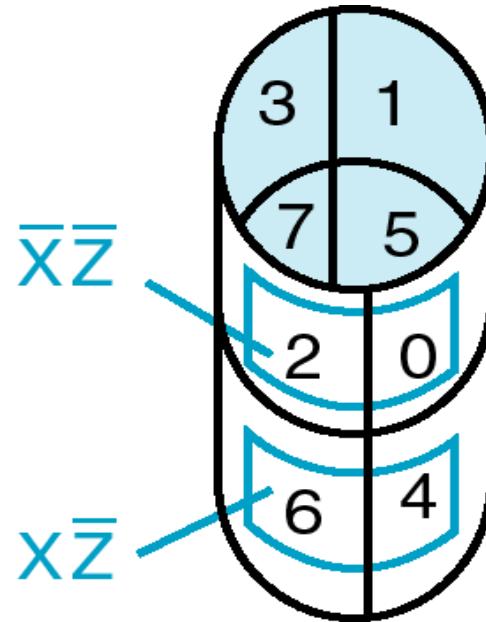
- ✓ Reduced literal product terms for SOP standard forms correspond to cubes i.e. to **rectangles** on the K-maps containing cell counts that are powers of 2.
- ✓ Rectangles of 2 cells represent 2 adjacent minterms; of 4 cells represent 4 minterms that form a “pairwise adjacent” ring.

# Three-Variable Maps

- ✓ Topological warps of 3-variable K-maps that show all adjacencies:
  - Venn Diagram

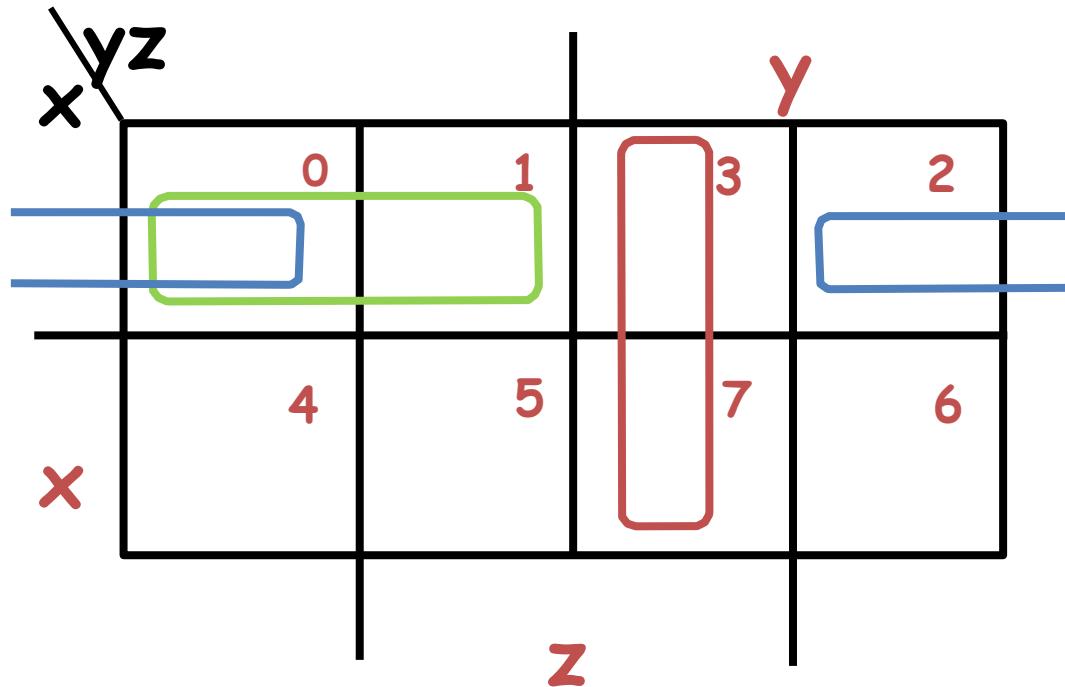


- Cylinder



# Three-Variable Maps

- ✓ Example Shapes of 2-cell Rectangles:



- ✓ Read off the product terms for the rectangles shown

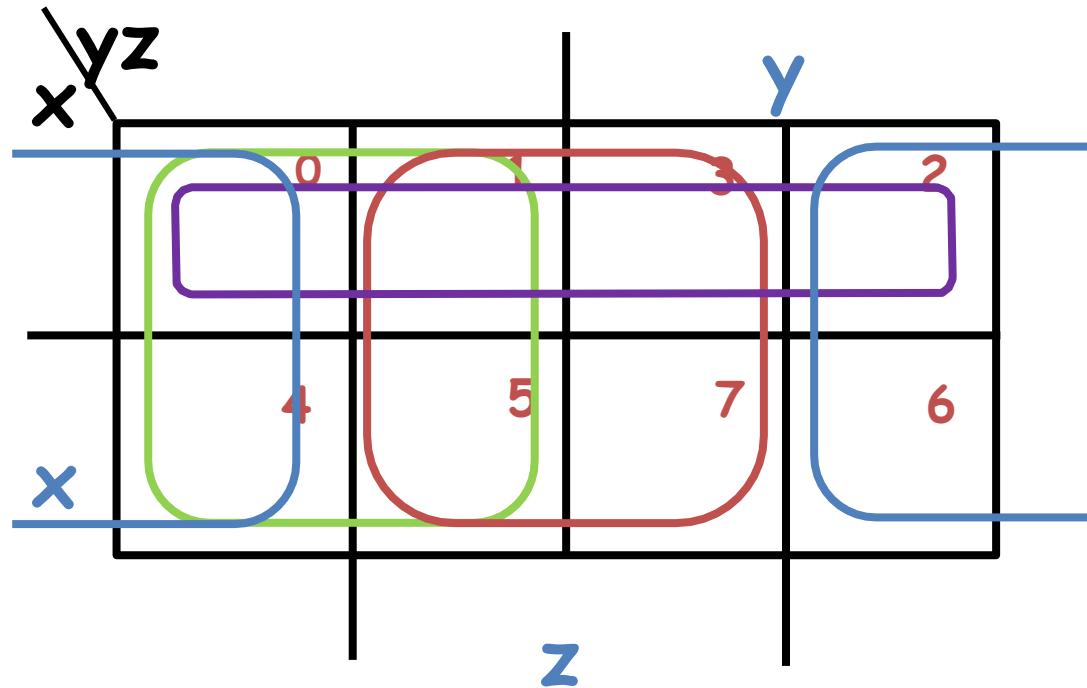
$x'z'$

$x'y'$

$yz$

# Three-Variable Maps

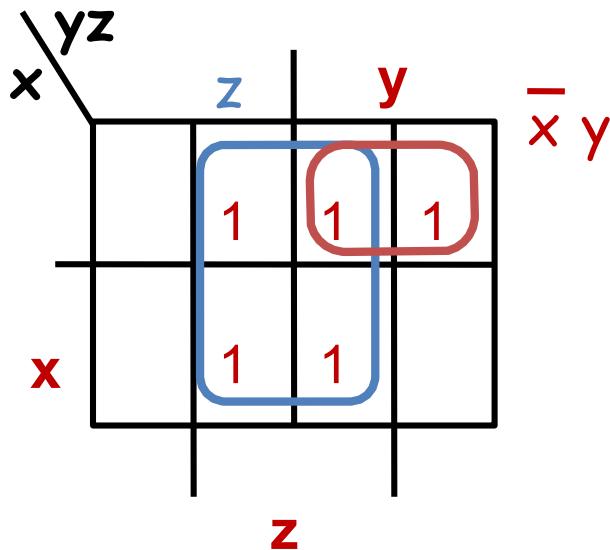
- ✓ Example Shapes of 4-cell Rectangles:



- ✓ Read off the product terms for the rectangles shown

# Three Variable Maps

- ✓ K-Maps can be used to simplify Boolean functions by a systematic methods. Terms are selected to cover the "1s" in the map.
- ✓ Example: Simplify  $F(x, y, z) = \sum_m (1, 2, 3, 5, 7)$

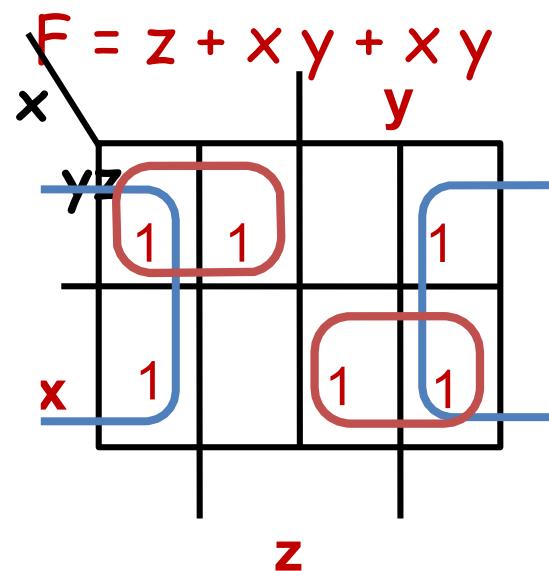


$$F(x, y, z) = z + \bar{x}y$$

# Three-Variable Map Simplification

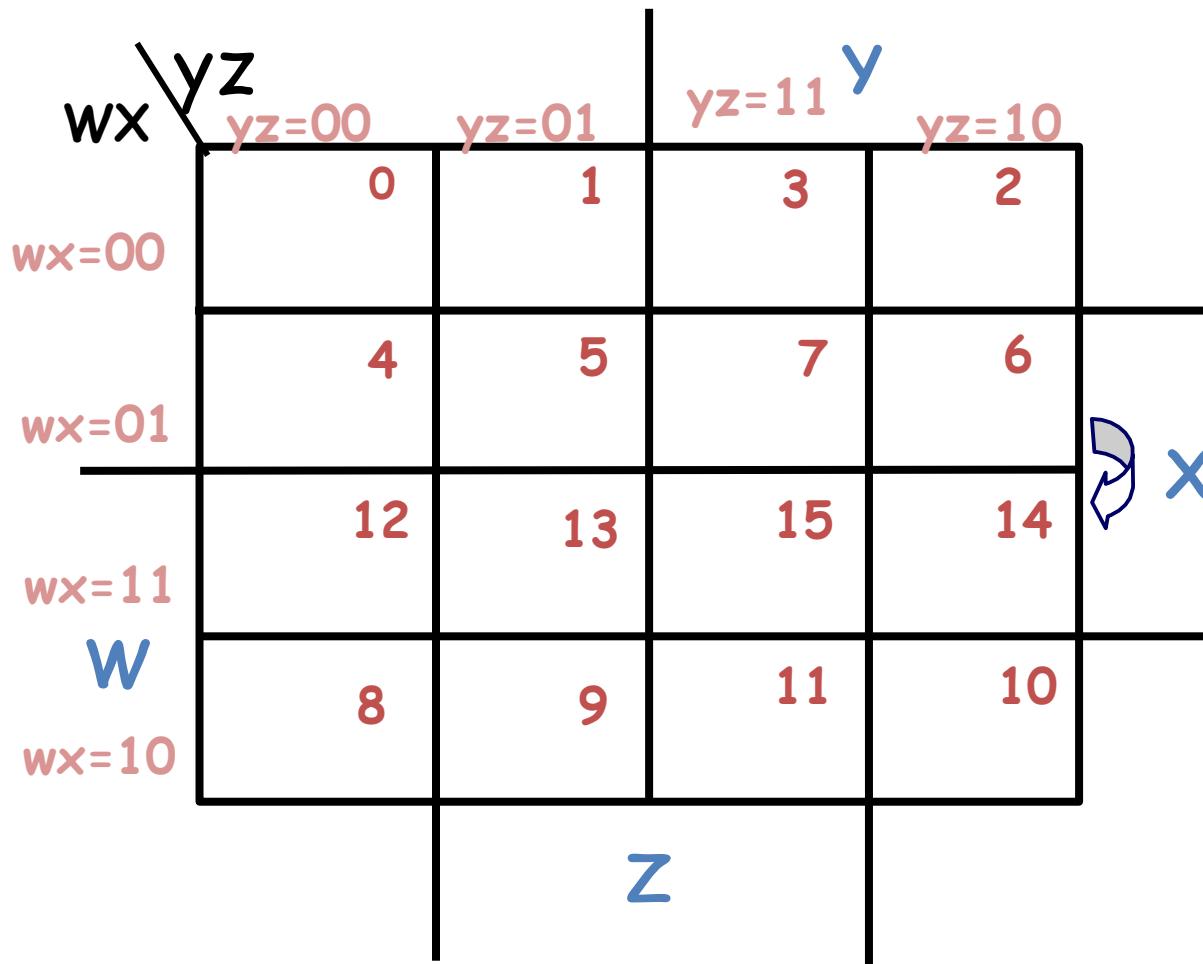
- ✓ Use a K-map to find an optimum SOP equation for  $F(x, y, z) = \sum_m (0, 1, 2, 4, 6, 7)$

— — —



# Four Variable Maps

✓ Map and location of minterms:



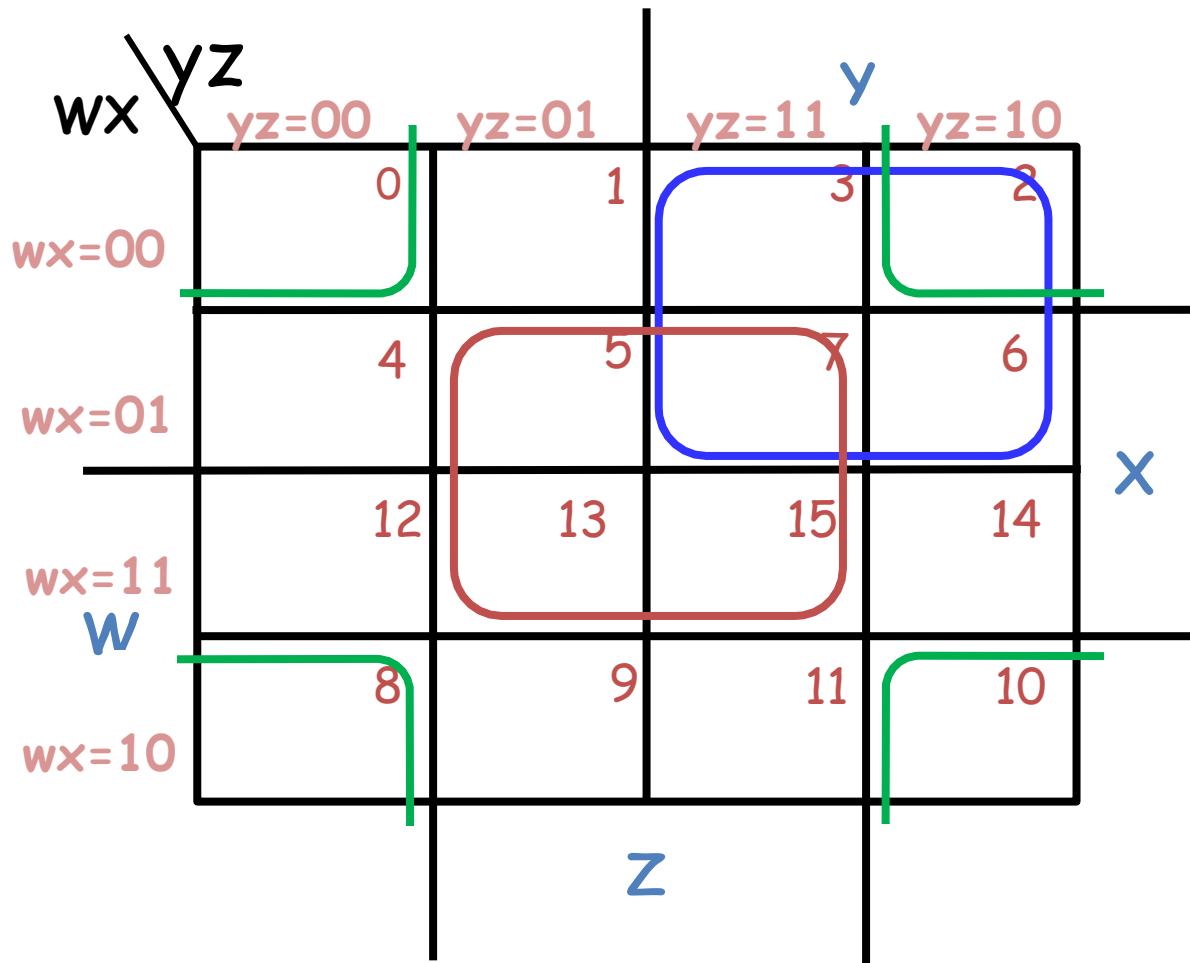
# Four Variable Terms

---

- ✓ Four variable maps can have rectangles corresponding to:
  - A single 1 = 4 variables, (i.e. Minterm)
  - Two 1s = 3 variables,
  - Four 1s = 2 variables
  - Eight 1s = 1 variable,
  - Sixteen 1s = zero variables (i.e. Constant "1")

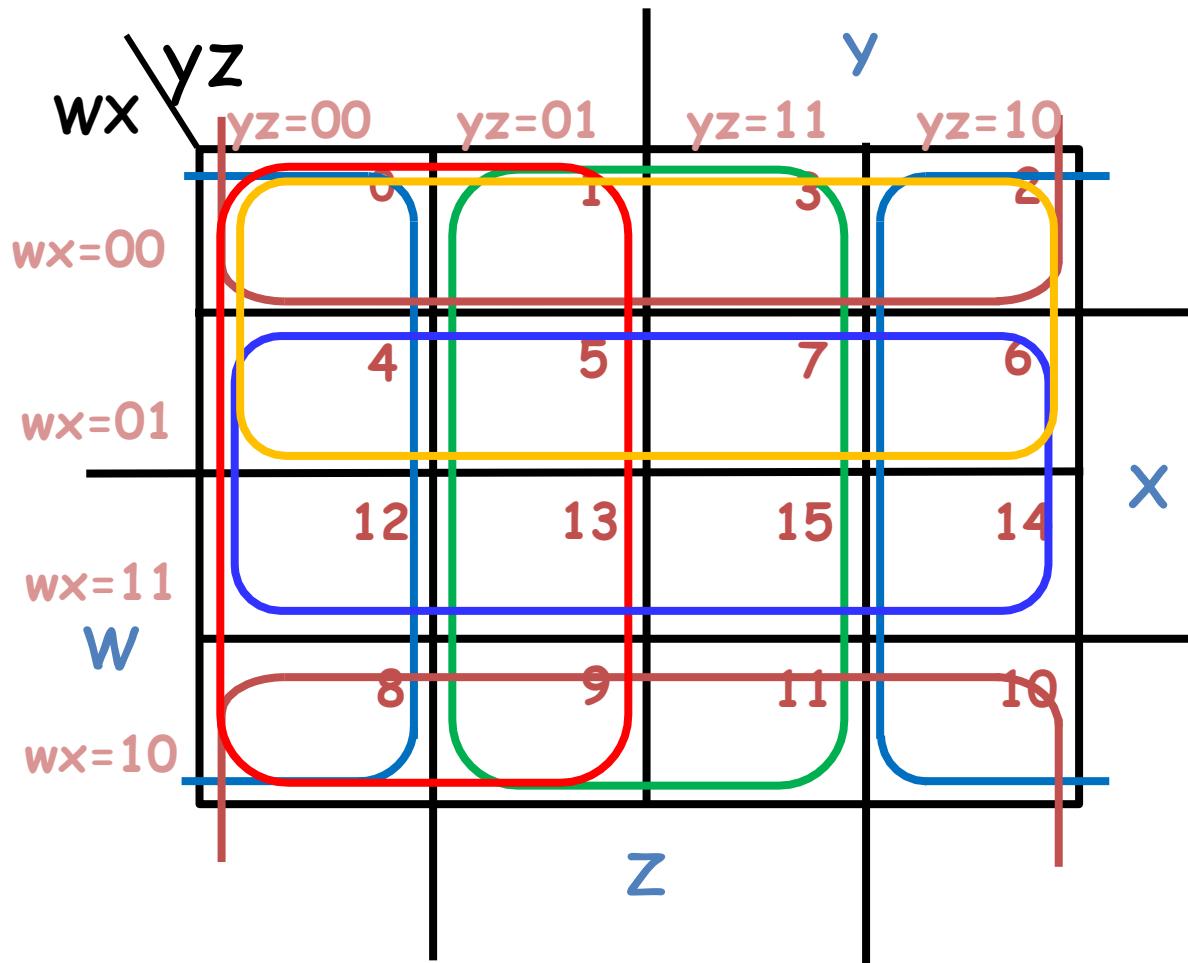
# Four-Variable Maps

✓ Example Shapes of Rectangles:



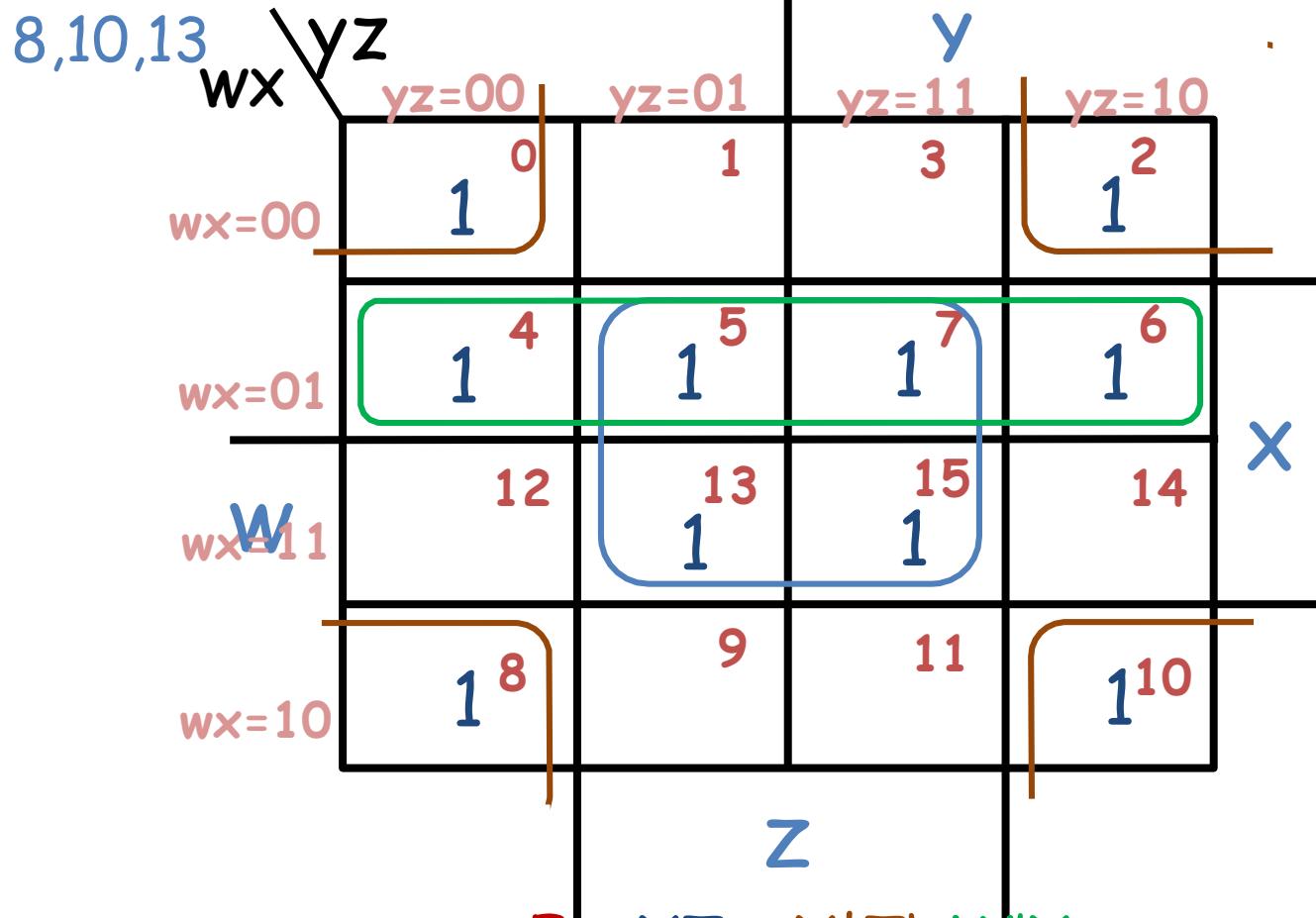
# Four-Variable Maps

✓ Example Shapes of Rectangles:



# Four-Variable Map Simplification

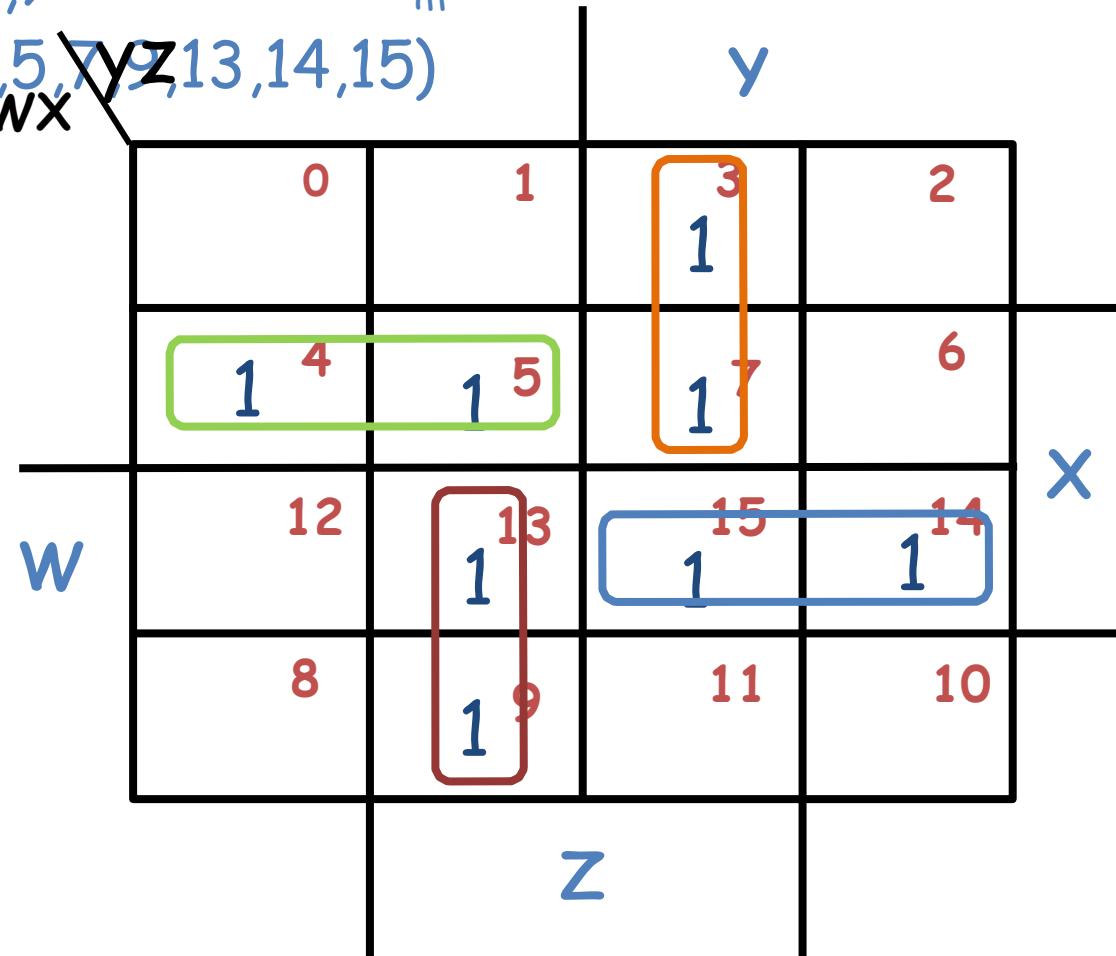
$$\checkmark F(W, X, Y, Z) = \sum_m (0, 2, 4, 5, 6, 7,$$



$$F = XZ + X'Z' + W'X$$

# Four-Variable Map Simplification

$$F(W, X \vee Z) = \sum_m (3, 4, 5, 7, 9, 13, 14, 15)$$



$$F = W'X'Y' + W'YZ + WXY + WY'Z$$

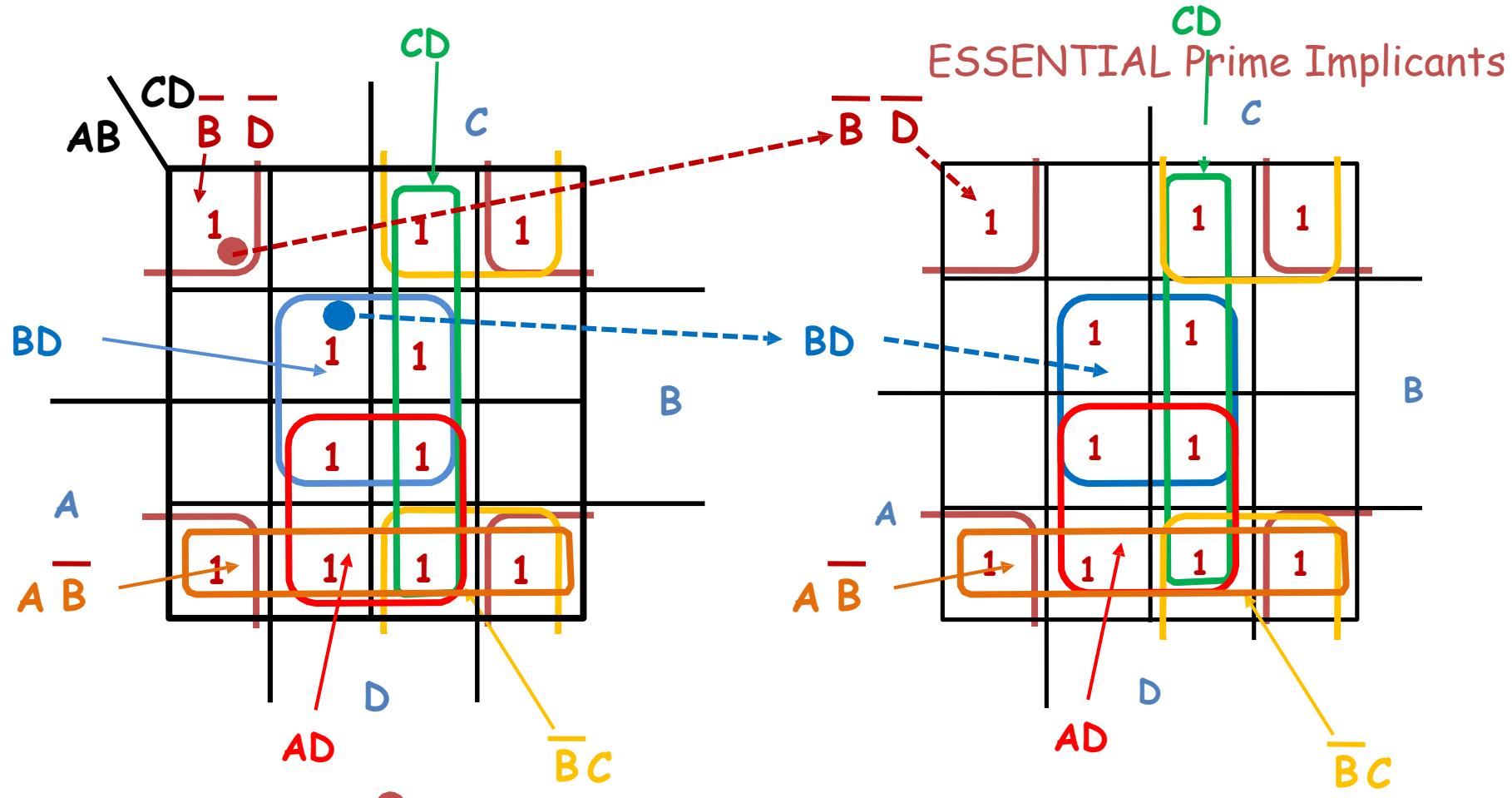
# Systematic Simplification

---

- ✓ A Prime Implicant is a cube i.e. a product term obtained by combining the maximum possible number of adjacent squares in the map into a rectangle with the number of squares a power of 2.
- ✓ A prime implicant is called an Essential Prime Implicant if it is the only prime implicant that covers (includes) one or more minterms.
- ✓ Prime Implicants and Essential Prime Implicants can be determined by inspection of a K-Map.
- ✓ A set of prime implicants "covers all minterms" if, for each minterm of the function, at least one prime implicant in the set of prime implicants includes the minterm.

# Example of Prime

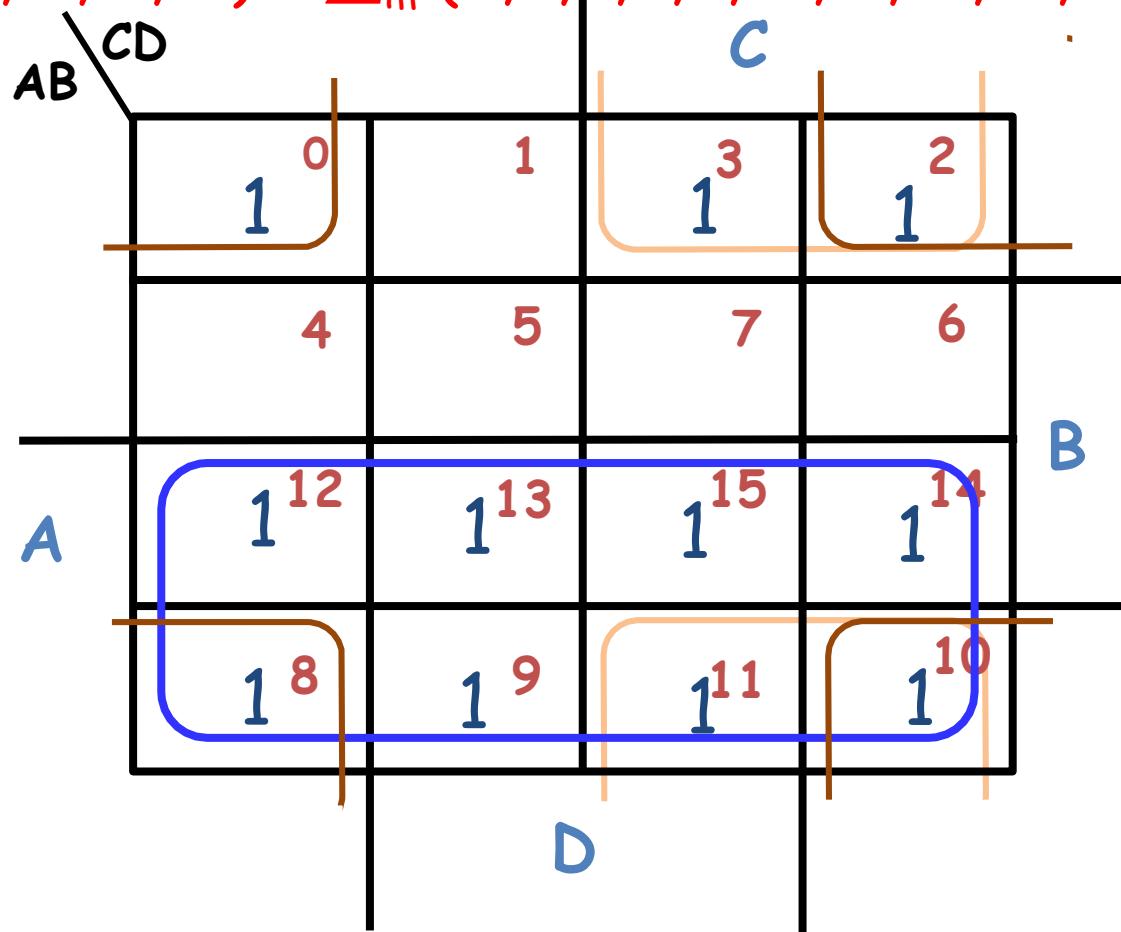
✓ Find ALL Prime Implicants



# Prime Implicant Practice

✓ Find all prime implicants for:

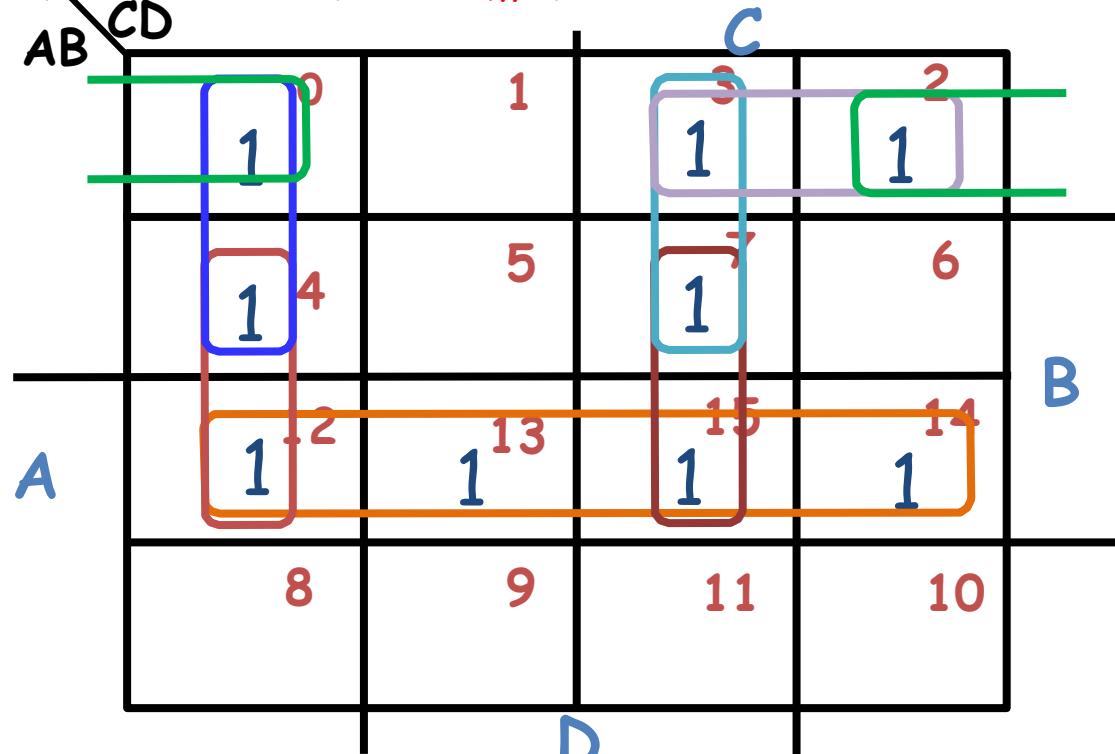
$$\checkmark F(A, B, C, D) = \sum_m (0, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15)$$



# Another Example

✓ Find all prime implicants for:

$$F(A, B, C, D) = \sum_m (0, 2, 3, 4, 7, 12, 13, 14, 15)$$

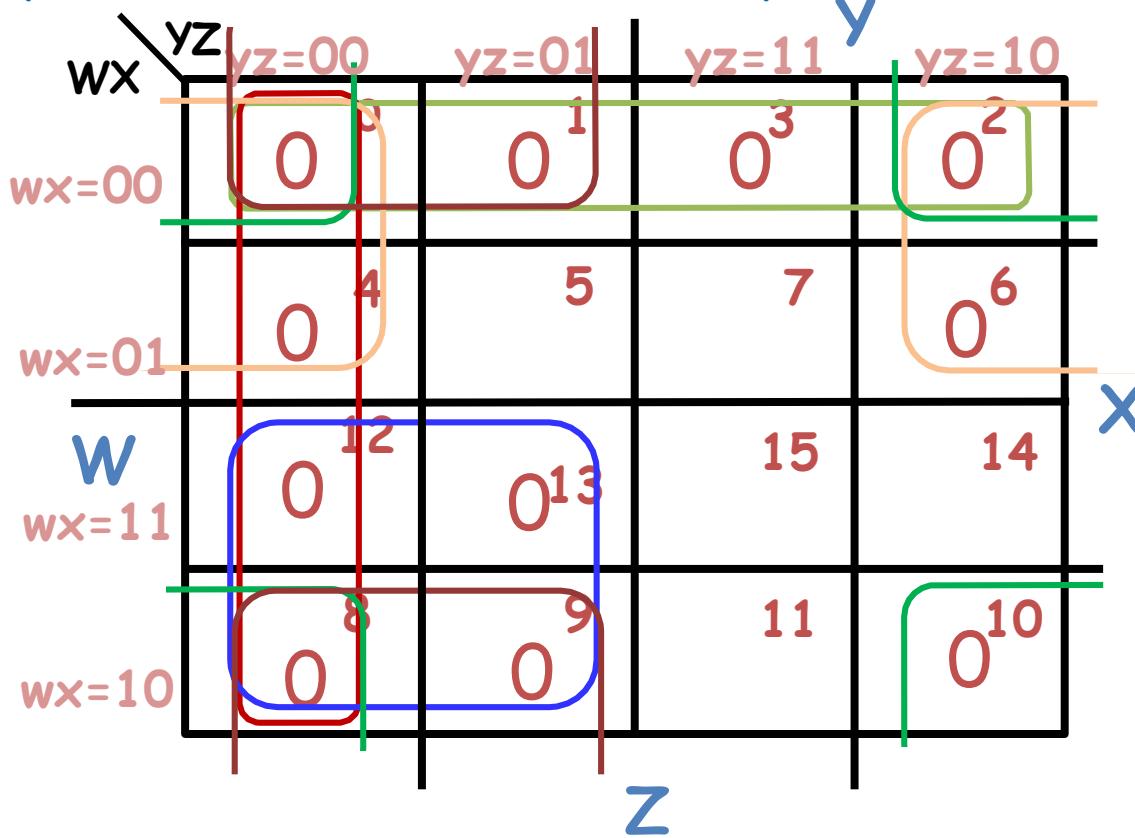


$AB, B\bar{C}D, \bar{A}CD, \bar{A}BD, \bar{A}BC, \bar{A}CD, BCD$

# K-Maps, implicants

✓ Find all prime implicants for:

$$F = (w+y+z) (w'+x'+y) (x+y) (w+x+y') (w+x'+z) (w'+x+z)$$



$$\prod_m = (w+x)(w'+y)(w+z)(x+z) \quad \prod_c = \prod_m (y+z) (y'+x')$$

# Don't Cares in K-Maps

---

- ✓ Sometimes a function table or map contains entries for which it is known that:
  - the input values for the minterm will **never occur** or
  - the output value for the minterm is **not used**
- ✓ In these cases, the output value **need not be defined**
- ✓ Instead, the output value is defined as a **don't care**
- ✓ By placing "don't cares" ( an "x" entry) in the function table or map, **the cost of the logic circuit may be lowered.**
- ✓ Example: A logic function having the binary codes for the BCD digits as its inputs. Only the codes for 0 through 9 are used. The six codes, 1010 through 1111 **never occur**, so the output values for these codes are **x** to represent "don't cares."

# Incompletely Specified Functions

✓  $F = (f, d, r) : B^n \rightarrow \{0, 1, \text{x}\}$

where  $\text{x}$  represents "don't care".

- $f$  = onset function -
- $r$  = offset function -
- $d$  = don't care function -

$$f(x)=1 \leftrightarrow F(x)=1$$

$$r(x)=1 \leftrightarrow F(x)=0$$

$$d(x)=1 \leftrightarrow F(x)=*$$

$(f, d, r)$  forms a partition of  $B^n$ . i.e.

- $f + d + r = B^n$
- $fd = fr = dr = \emptyset$  (pairwise disjoint)

---

# **CSE1003-Digital Logic Design**

## **Module-3 COMBINATIONAL CIRCUITS-I**

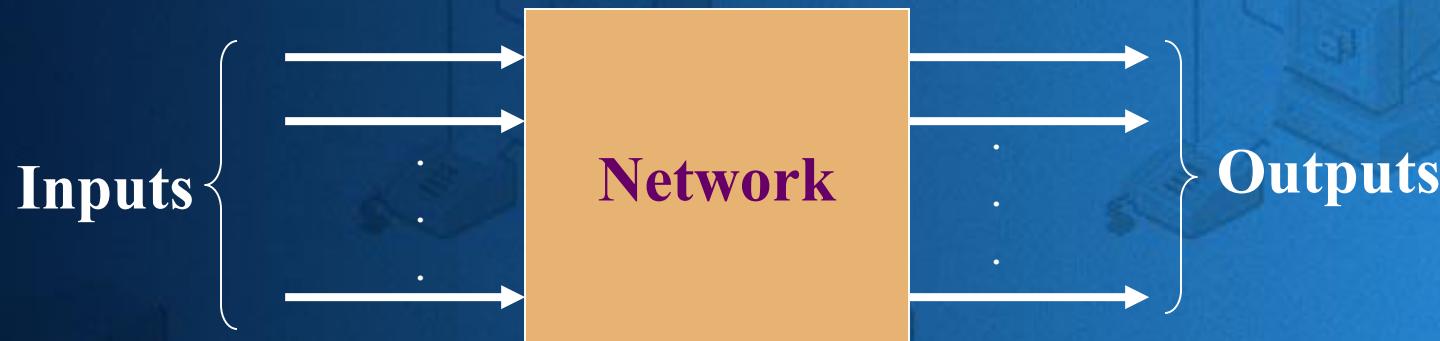
<b>Module:3</b>	<b>COMBINATIONAL CIRCUIT - I</b>	<b>4 hours</b>
Adder - Subtractor - Code Converter - Analyzing a Combinational Circuit		
<b>Module:4</b>	<b>COMBINATIONAL CIRCUIT -II</b>	<b>6 hours</b>
Binary Parallel Adder- Look ahead carry - Magnitude Comparator - Decoders – Encoders - Multiplexers –Demultiplexers.		

# Overview

---

- **Part 1 – Design Procedure**
- **Part 2 – Combinational Logic**
- **Part 3 – Arithmetic Functions**
  - Iterative combinational circuits
  - **Binary adders**
    - Half and full adders
    - Ripple carry and carry lookahead adders
  - **Binary subtraction**
  - **Binary adder-subtractors**
    - Signed binary numbers
    - Signed binary addition and subtraction
    - Overflow
  - **Binary Multiplication**

# Remember



## ◆ Combinational

- The outputs depend only on the current input values
- It uses only logic gates

## ◆ Sequential

- The outputs depend on the current and past input values

**It uses logic gates and storage elements**

# Notes

- ◆ If there are  $n$  input variables, there are  $2^n$  input combinations
- ◆ For each input combination, there is one output value
- ◆ Truth tables are used to list all possible combinations of inputs and corresponding output values

# Basic Combinational Circuits

- ◆ Adders
- ◆ Subtractors
- ◆ Multipliers
- ◆ Multiplexers
- ◆ Decoders
- ◆ Encoders
- ◆ Comparators

# Part-1-Design Procedure

- ◆ Determine the inputs and outputs
- ◆ Assign a symbol for each
- ◆ Derive the truth table
- ◆ Get the simplified boolean expression for each output
- ◆ Draw the network diagram

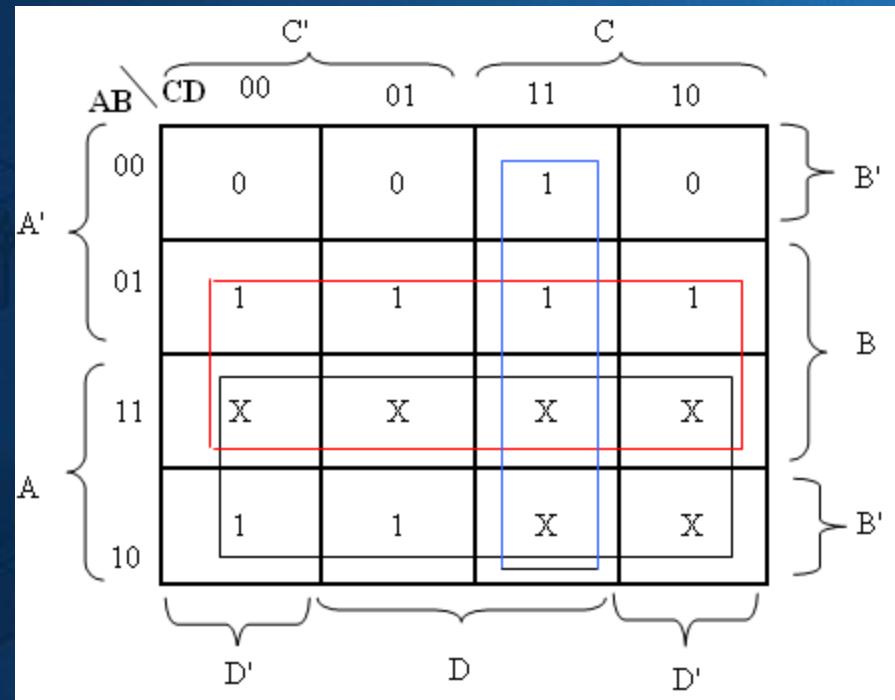
# Example

## ◆ Conversion from BCD to excess-5

INPUT				OUTPUT			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	1	0	1
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	1
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	1	1	1	0

# Example (Cont.)

$$W = A + B + CD$$



# Example (Cont.)

$$X = A + B'D' + B'C' + BCD$$

		CD		C'		C	
		00	01	11	10		
AB	00	1	1	0	1	B'	
	01	0	0	1	0		
A	11	X	X	X	X	B	
	10	1	1	X	X		
		D'		D		D'	

# Example (Cont.)

Find  $Y$  and  $Z$

Draw the network diagram

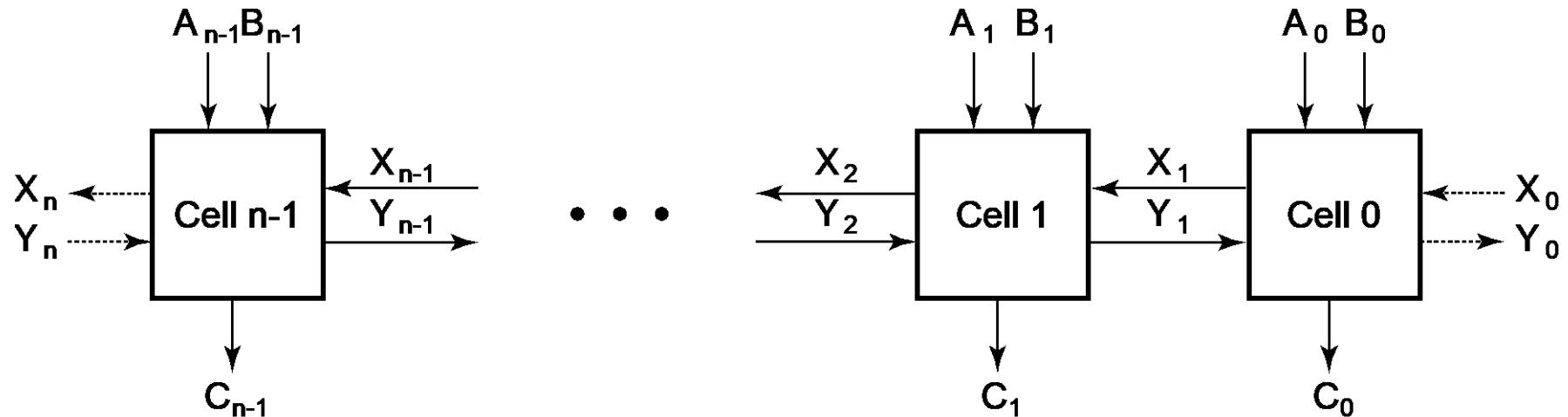
For more details read the text Book

# Iterative Combinational Circuits

---

- Arithmetic functions
  - Operate on binary vectors
  - Use the same subfunction in each bit position
- Can design functional block for subfunction and repeat to obtain functional block for overall function
- *Cell* - subfunction block
- *Iterative array* - a array of interconnected cells
- An iterative array can be in a single dimension (1D) or multiple dimensions

# Block Diagram of a 1D Iterative Array



- Example:  $n = 32$ 
  - Number of inputs = ?
  - Truth table rows = ?
  - Equations with up to ? input variables
  - Equations with huge number of terms
  - Design impractical!
- Iterative array takes advantage of the regularity to make design feasible

# Adders

- ◆ Essential part of every CPU
- ◆ Half adder (Ignore the carry-in bit)
  - It performs the addition of two bits
- ◆ Full adder
  - It performs the addition of three bits

# Functional Blocks: Addition

---

- **Binary addition used frequently**
- **Addition Development:**
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block,
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block,
  - *Ripple Carry Adder*, an iterative array to perform binary addition, and
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.

# Functional Block: Half-Adder

---

- A 2-input, 1-bit width binary adder that performs the following computations:

$$\begin{array}{r}
 & X & 0 & 0 & 1 & 1 \\
 & + Y & + 0 & + 1 & + 0 & + 1 \\
 \hline
 C S & 0 0 & 0 1 & 0 1 & 0 1 & 1 0
 \end{array}$$

- A half adder adds two bits to produce a two-bit sum
- The sum is expressed as a sum bit, S and a carry bit, C
- The half adder can be specified as a truth table for S and C  $\Rightarrow$

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

# Logic Simplification: Half-Adder

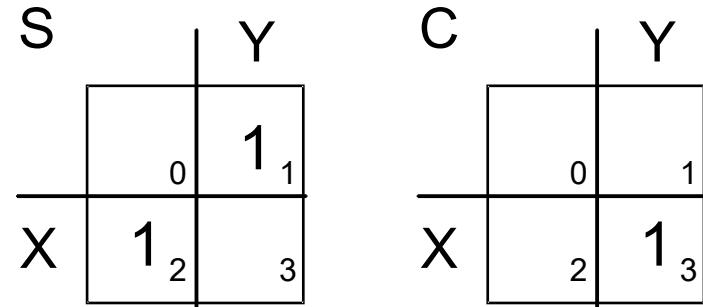
---

- The K-Map for S, C is:
- This is a pretty trivial map!  
By inspection:

$$S = X \times \bar{Y} + \bar{X} \times Y = X \oplus Y$$

$$S = (X + Y) \times (\bar{X} + \bar{Y})$$

- and
- $C = X \times Y$
- $C = \overline{(\overline{(X \times Y)})}$
- These equations lead to several implementations.



# Five Implementations: Half-Adder

---

- We can derive following sets of equations for a half-adder:

$$(a) S = X \times \bar{Y} + \bar{X} \times Y \\ C = X \times Y$$

$$(d) \underline{S} = (\underline{X} + \underline{Y}) \times \bar{\underline{C}} \\ \underline{C} = (\underline{X} + \underline{Y})$$

$$(b) S = (X + Y) \times (\bar{X} + \bar{Y}) \\ C = \underline{X \times Y}$$

$$(e) S = X \oplus Y \\ C = X \times Y$$

$$(c) S = \underline{(C + \bar{X} \times \bar{Y})} \\ C = X \times Y$$

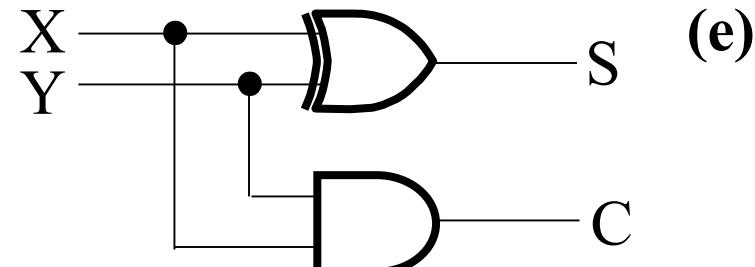
- (a), (b), and (e) are SOP, POS, and XOR implementations for S.
- In (c), the C function is used as a term in the AND-NOR implementation of S, and in (d), the  $\bar{C}$  function is used in a POS term for S.

# Implementations: Half-Adder

- The most common half adder implementation is:

$$S = X \oplus Y$$

$$C = X \times Y$$



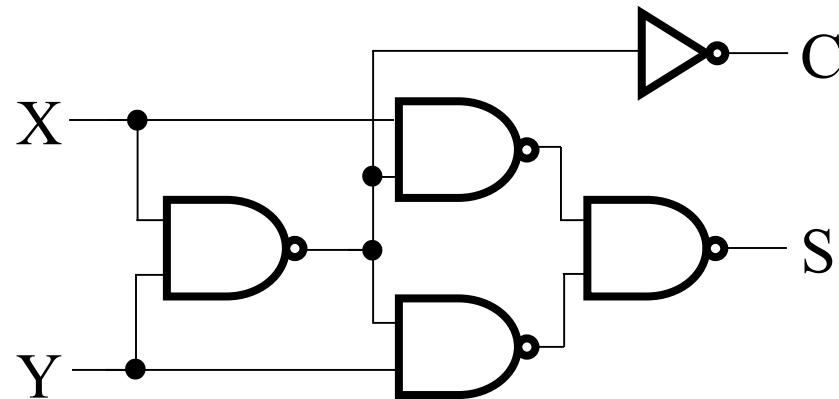
- A NAND only implementation is:

$$S = (X + Y) \times \overline{C}$$

$$= X \times \overline{C} + Y \times \overline{C}$$

$$= \overline{(X \times \overline{C})} \times \overline{(Y \times \overline{C})}$$

$$C = \overline{\overline{X} \times \overline{Y}}$$



# Functional Block: Full-Adder

---

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.

- For a carry-in (Z) of 0, it is the same as the half-adder:

$$\begin{array}{r}
 \begin{array}{ccccc} Z & 0 & 0 & 0 & 0 \\ X & 0 & 0 & 1 & 1 \\ + Y & + 0 & + 1 & + 0 & + 1 \\ \hline C & 0 & 0 & 0 & 1 \\ S & 1 & 1 & 1 & 0 \end{array}
 \end{array}$$

- For a carry- in (Z) of 1:

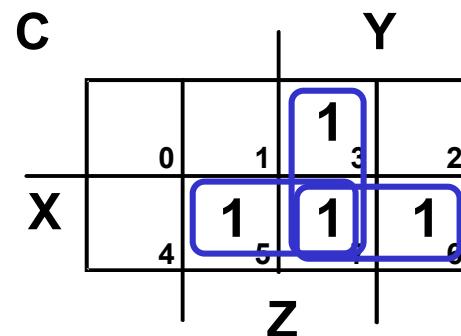
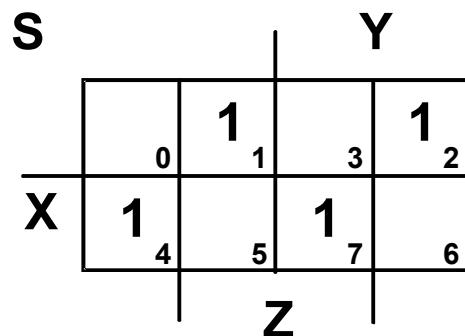
$$\begin{array}{r}
 \begin{array}{ccccc} Z & 1 & 1 & 1 & 1 \\ X & 0 & 0 & 1 & 1 \\ + Y & + 0 & + 1 & + 0 & + 1 \\ \hline C & 0 & 1 & 1 & 0 \\ S & 1 & 0 & 0 & 1 \end{array}
 \end{array}$$

# Logic Optimization: Full-Adder

- Full-Adder Truth Table:

X	Y	Z	C	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

- Full-Adder K-Map:



# Equations: Full-Adder

---

- From the K-Map, we get:

$$\begin{aligned}S &= X \bar{Y} \bar{Z} + \bar{X} Y \bar{Z} + \bar{X} \bar{Y} Z + X Y Z \\C &= X Y + X Z + Y Z\end{aligned}$$

- The S function is the three-bit XOR function (Odd Function):

$$S = X \oplus Y \oplus Z$$

- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

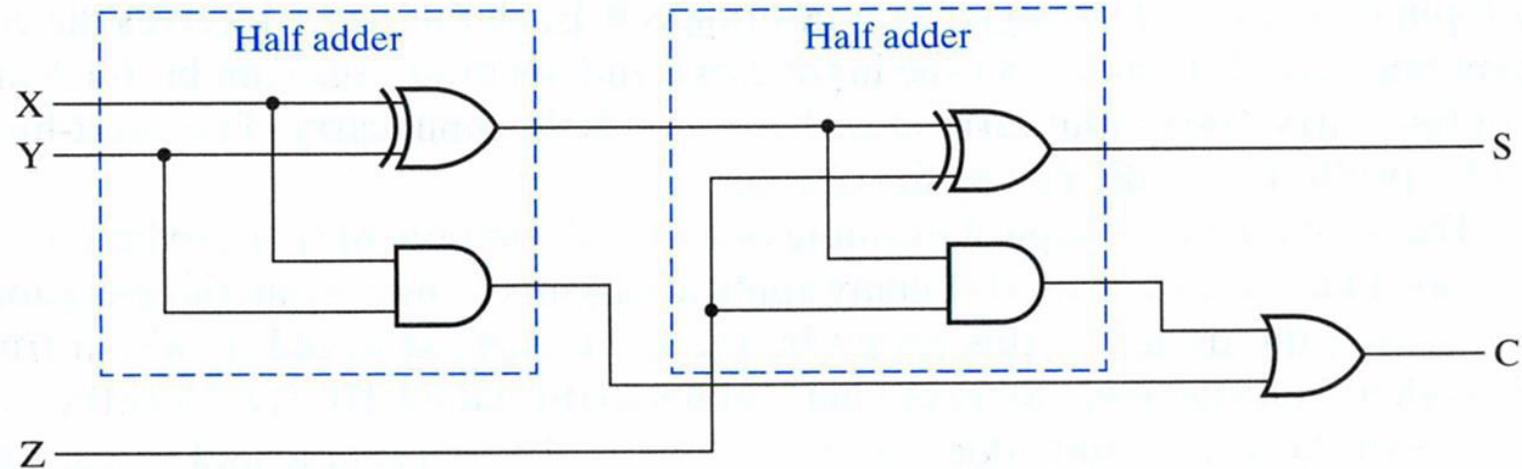
$$C = X Y + (X \oplus Y) Z$$

- The term  $X \cdot Y$  is *carry generate*.
- The term  $X \oplus Y$  is *carry propagate*.

# Logic diagram of full adder

$$S = X \oplus Y \oplus Z$$

$$C = XY + (X \oplus Y)Z$$



□ **FIGURE 3-42**  
Logic Diagram of Full Adder

# Another Implementation of full adder

- Full Adder Schematic
- Here X, Y, and Z, and C (from the previous pages) are  $A_i$ ,  $B_i$ ,  $C_i$  and  $C_{i+1}$ , respectively. Also,

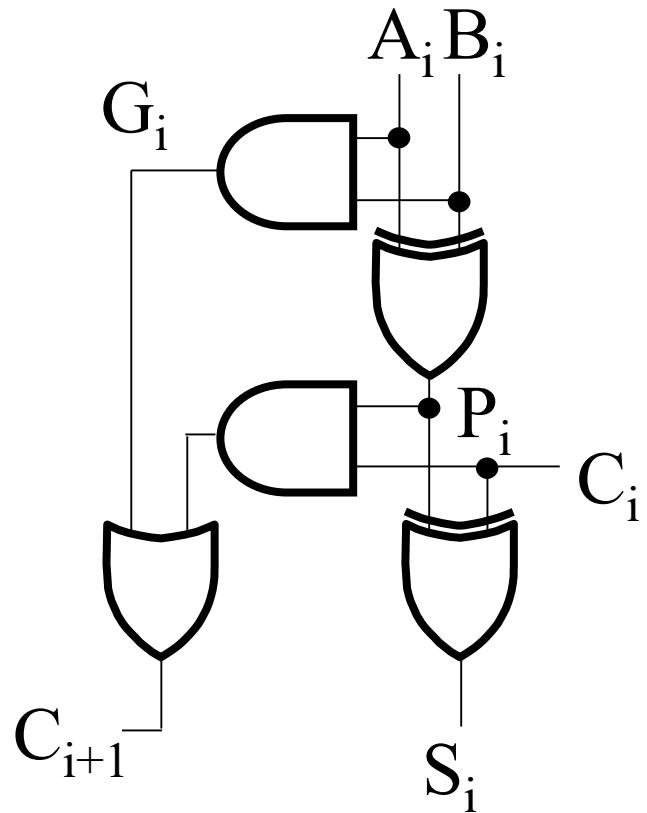
$G$  = generate and

$P$  = propagate.

- Note: This is really a combination of a 3-bit odd function (for S) and Carry logic (for  $C_{i+1}$ ):

( $G$  = Generate) OR ( $P$  = Propagate AND  $C_i$  = Carry In)

$$C_{i+1} = G + P \cdot C_i$$



# Binary Adders

---

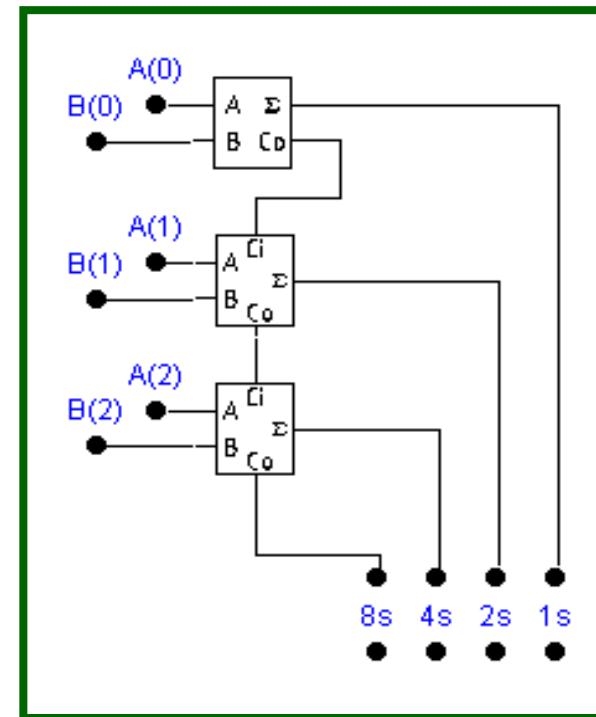
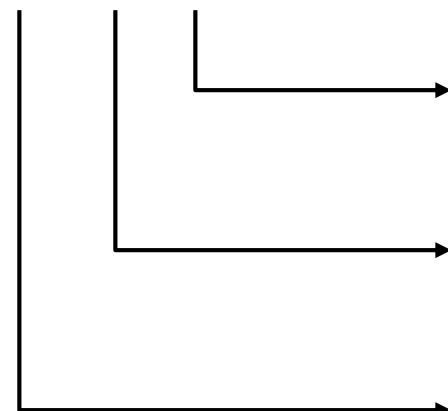
- To add multiple operands, we “bundle” logical signals together into vectors and use functional blocks that operate on the vectors
- Example: 4-bit ripple carry adder:** Adds input vectors **A(3:0)** and **B(3:0)** to get a sum vector **S(3:0)**
- Note:** carry out of cell **i** becomes carry in of cell **i + 1**

Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	$C_i$
Augend	1 0 1 1	$A_i$
Addend	<u>0 0 1 1</u>	$B_i$
Sum	1 1 1 0	$S_i$
Carry out	0 0 1 1	$C_{i+1}$

# BINARY PARALLEL ADDING

- Use half adder for LSD
- Use full adder for other digits

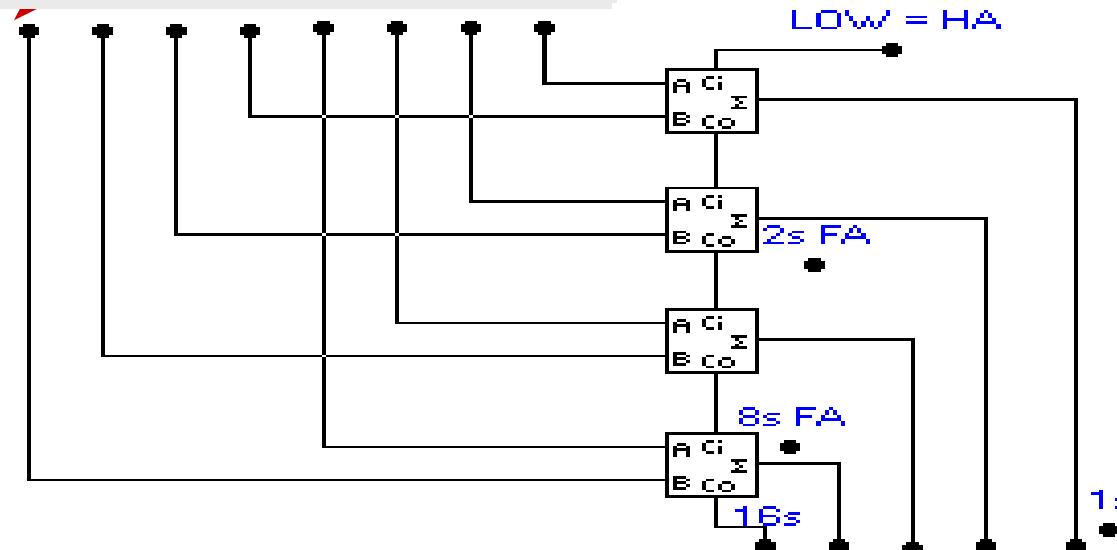
$$\begin{array}{r} \textcolor{blue}{A_2} \quad \textcolor{blue}{A_1} \quad \textcolor{blue}{A_0} \\ + \textcolor{red}{B_2} \quad \textcolor{red}{B_1} \quad \textcolor{red}{B_0} \\ \hline \end{array}$$



# 4-BIT PARALLEL ADDER

Enter binary numbers  
to be added

$$1\ 1\ 1\ 0 + 0\ 1\ 1\ 0$$



$$1\ 0\ 1\ 0\ 0$$

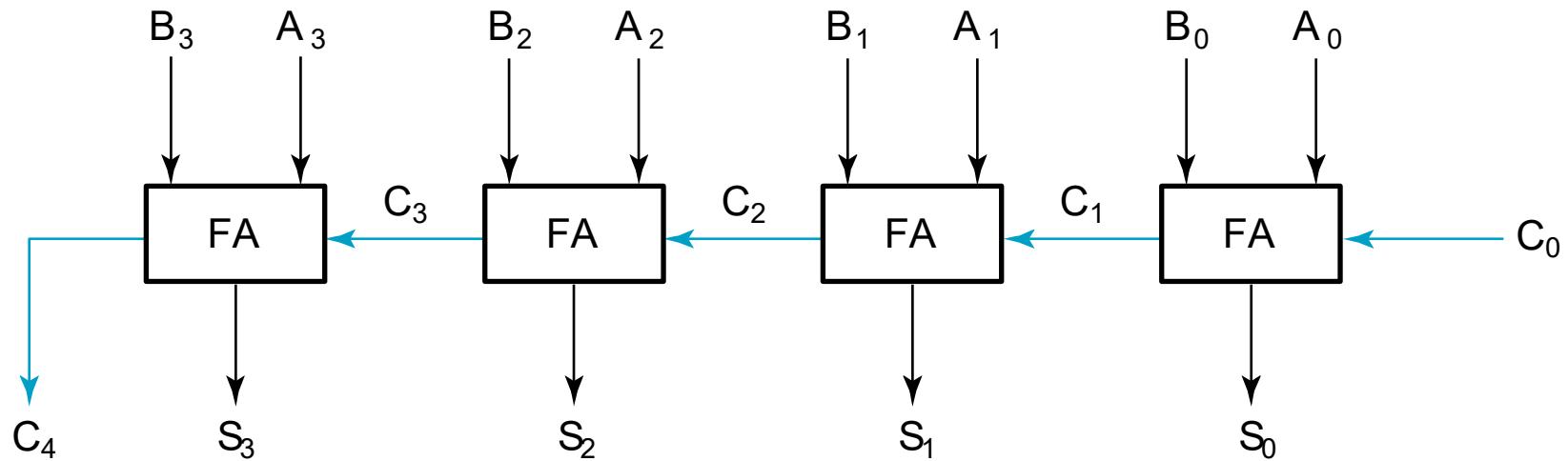
Parallel adders are available in IC form.

1s place uses half-adder

2s, 4s, 8s places use full adders

# 4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



# BINARY SUBTRACTION

*Example:* Subtract binary number 101 from 1011

(borrow)

$$\begin{array}{r} & \overset{0}{\cancel{1}} & 1 & 0 & 1 & 1 \\ - & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$



TEST

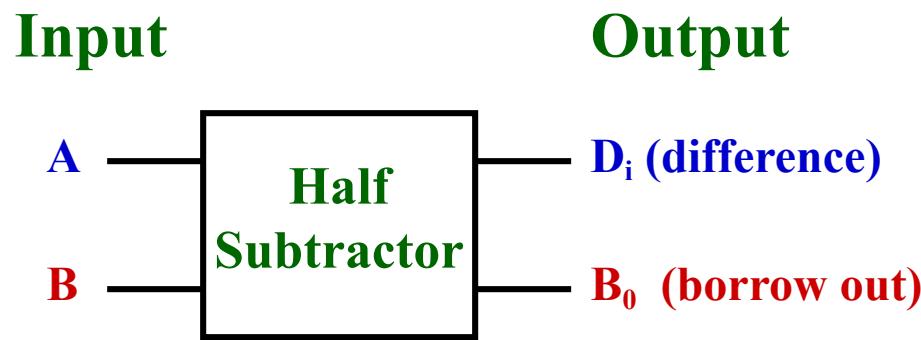
Subtract binary number 11 from 1010

$$\begin{array}{r} & & 01 \\ & \cancel{0} & \cancel{10} & 10 \\ 1 & \cancel{0} & \cancel{10} & 1 \\ - & & 1 & 1 \\ \hline 0 & 1 & 1 & 1 \end{array}$$

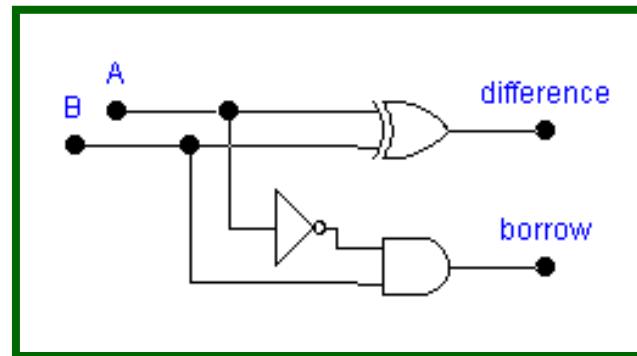
# HALF SUBTRACTOR

Subtracts LSD column in binary subtraction

Logic  
Symbol:



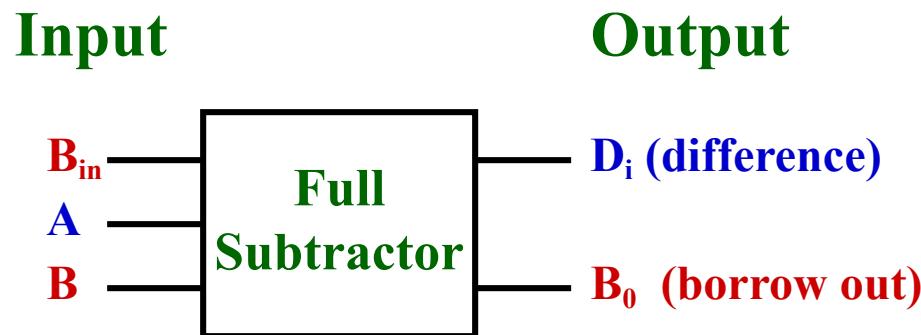
Logic  
Diagram:



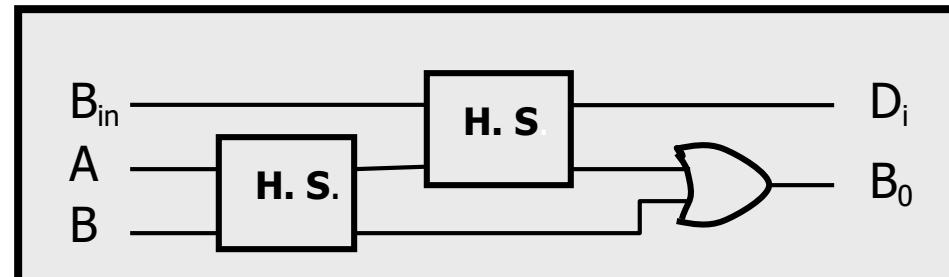
# FULL SUBTRACTOR

Used for subtracting binary place values other than the 1s place

Logic Symbol:

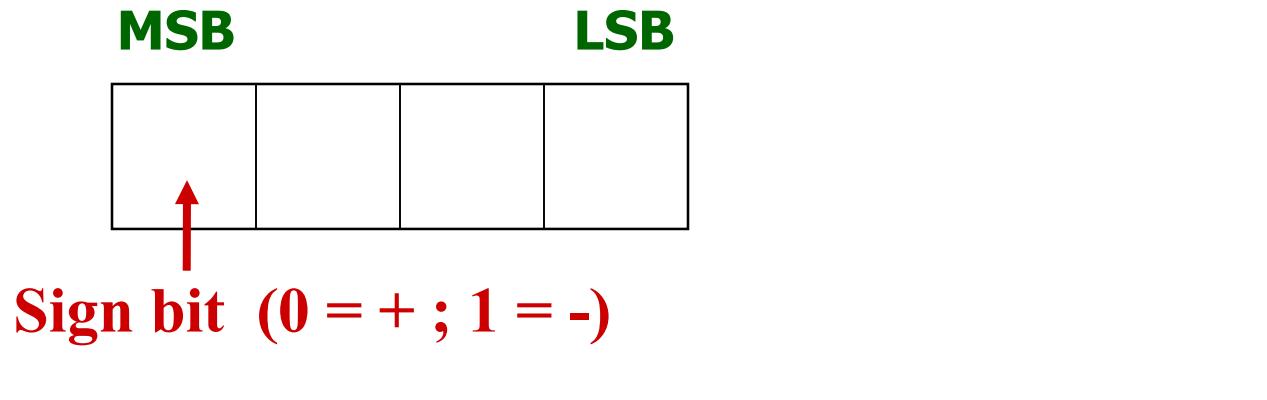


Logic Diagram:



# **2s COMPLEMENT NOTATION**

- **2s complement representation - widely used in microprocessors.**
- Represents *sign* and *magnitude*



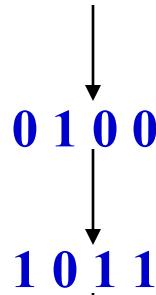
<b>Decimal:</b>	<b>+7</b>	<b>+4</b>	<b>+1</b>	<b>0</b>	<b>-1</b>	<b>-4</b>	<b>-7</b>
<b>2s Complement:</b>	<b>0111</b>	<b>0100</b>	<b>0001</b>	<b>0000</b>	<b>1111</b>	<b>1100</b>	<b>1001</b>

# 2s COMPLEMENT - CONVERSIONS

- Converting positive numbers to 2s complement:
  - Same as converting to binary
- Converting negative numbers to 2s complement:

## Decimal to 2s Complement

- 4 (decimal)



Convert decimal to binary

1s complement

Add 1

$$- 4 = 1100 \text{ (2s Complement)}$$

## 2s Complement to Binary

1 1 0 0 (2s C)

0 0 1 1

1s complement

Add 1

0 1 0 0 (Binary)

# ADDING/SUBTRACTING IN 2s COMPLEMENT

2s complement notation makes it possible  
to add and subtract signed numbers

(Decimal)

$$\begin{array}{r} (-1) \\ + (-2) \\ \hline (-3) \end{array}$$

2s

Complement

$$\begin{array}{r} 1111 \\ + 1110 \\ \hline \boxed{1}1101 \end{array}$$

Discard

2s complement

$$\begin{array}{r} (+1) \\ + (-3) \\ \hline (-2) \end{array}$$

$$\begin{array}{r} 0001 \\ + 1101 \\ \hline 1110 \end{array}$$

2s complement



# TEST

Add the following 2s complement numbers:

$$\begin{array}{r} (+5) \\ + (-4) \\ \hline \end{array} \quad \begin{array}{r} 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 0 \\ \hline \end{array}$$

**(+1)      1 0 0 0 1**

Discard

### EXAMPLE 3-20 Unsigned Binary Subtraction by 2s Complement Addition

Given the two binary numbers  $X = 1010100$  and  $Y = 1000011$ , perform the subtraction  $X - Y$  and  $Y - X$  using 2s complement operations. We have

$$X = \begin{array}{r} 1010100 \\ \hline \end{array}$$

$$\text{2s complement of } Y = \begin{array}{r} 0111101 \\ \hline \end{array}$$

$$\text{Sum} = \begin{array}{r} 10010001 \\ \hline \end{array}$$

$$\text{Discard end carry } 2^7 = -\underline{\hspace{2cm}10000000}$$

$$\text{Answer: } X - Y = \begin{array}{r} 0010001 \\ \hline \end{array}$$

$$Y = \begin{array}{r} 1000011 \\ \hline \end{array}$$

$$\text{2s complement of } X = \begin{array}{r} \underline{0101100} \\ \hline \end{array}$$

$$\text{Sum} = \begin{array}{r} 1101111 \\ \hline \end{array}$$

There is no end carry.

$$\text{Answer: } Y - X = -(\text{2s complement of } 1101111) = -0010001.$$

# Binary Subtractor

## ◆ Remember

- You need to take 2's complement to represent negative numbers
- A-B
  - ✓ Take 2's complement of B and add it to A
    - First take 1's complement and add 1

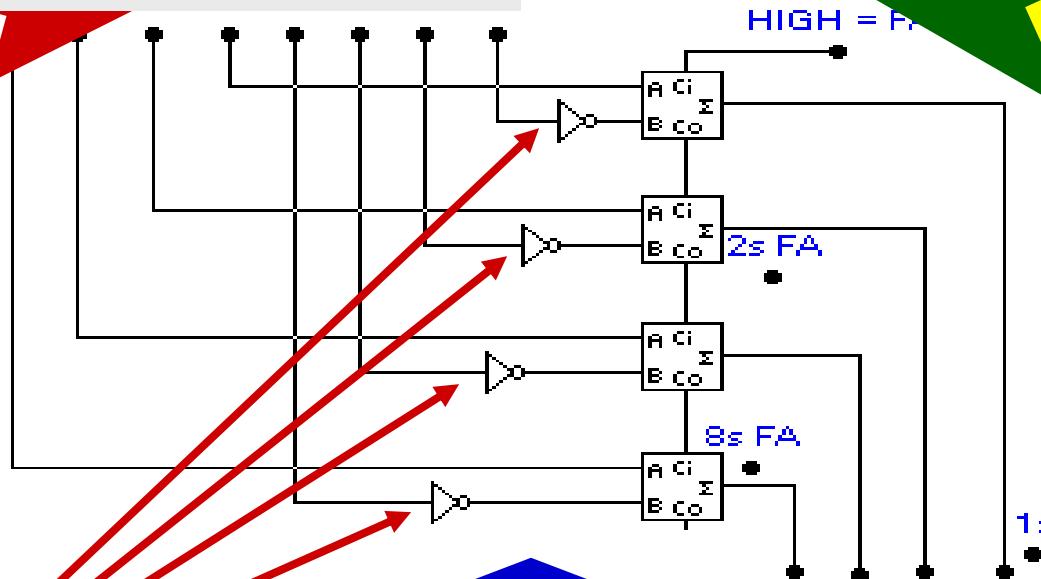
# PARALLEL SUBTRACTOR USING FULL ADDERS

Binary numbers to subtracted are input

1 0 0 1 - 0 1 1 1

HIGH at Carry in input acts like adding +1 to a 1s C number to form the 2s complement.  
1sC is formed by four inverters.

Inverters



Note the us

The result (difference) of the subtraction problem will appear here.

Also notice the addition of four inverters on the B inputs to the FAs

# 4-Bit Adder and Subtractor

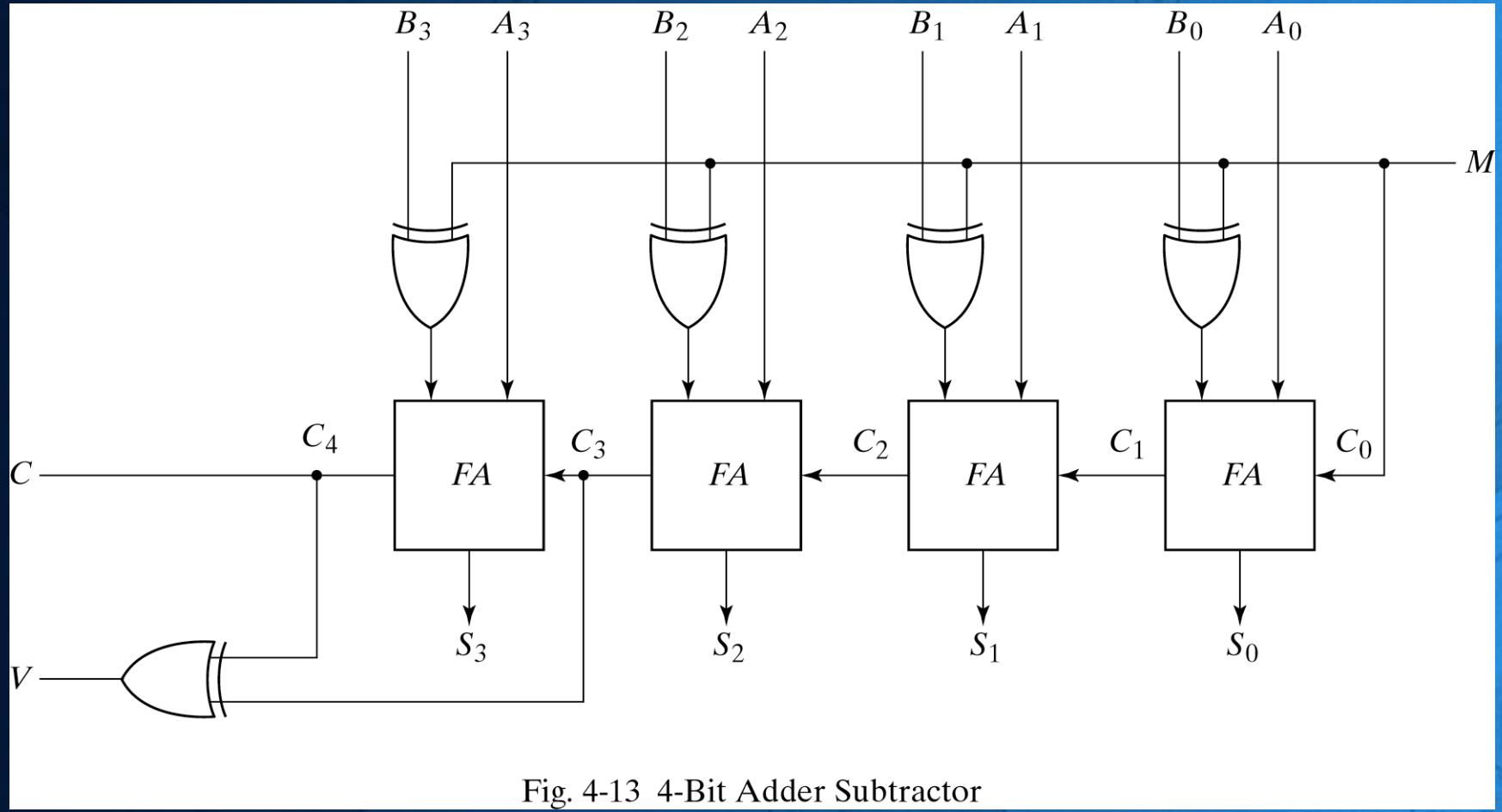


Fig. 4-13 4-Bit Adder Subtractor

$$M = 0(\text{Adder})$$

$$M = 1(\text{Subtractor})$$

V(Overflow)

# Signed Integers

---

- Positive numbers and zero can be represented by unsigned n-digit, radix  $r$  numbers. We need a representation for negative numbers.
- To represent a sign (+ or -) we need exactly one more bit of information (1 binary digit gives  $2^1 = 2$  elements which is exactly what is needed).
- Since computers use binary numbers, by convention, the most significant bit is interpreted as a sign bit:

$$s \ a_{n-2} \dots a_2 a_1 a_0$$

where:

**s = 0 for Positive numbers**

**s = 1 for Negative numbers**

**and  $a_i = 0$  or  $1$  represent the magnitude in some form.**

# Signed Integer Representations

---

- *Signed-Magnitude* – here the  $n - 1$  digits are interpreted as a positive magnitude.
- *Signed-Complement* – here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:
  - *Signed 1's Complement*
    - Uses 1's Complement Arithmetic
  - *Signed 2's Complement*
    - Uses 2's Complement Arithmetic

# Signed Integer Representation Example

---

- $r = 2, n = 3$

Number	Sign -Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	—
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	—	—	100

# Signed Integer Representation

□ TABLE 3-13  
Signed Binary Numbers

Decimal	Signed 2s Complement	Signed Magnitude
+ 7	0111	0111
+ 6	0110	0110
+ 5	0101	0101
+ 4	0100	0100
+ 3	0011	0011
+ 2	0010	0010
+ 1	0001	0001
+ 0	0000	0000
- 0	—	1000
- 1	1111	1001
- 2	1110	1010
- 3	1101	1011
- 4	1100	1100
- 5	1011	1101
- 6	1010	1110
- 7	1001	1111
- 8	1000	—

# Signed-Magnitude Arithmetic

---

- If the parity of the two signs is 0:
  1. Add the magnitudes.
  2. Check for overflow (a carry out of the MSB)
  3. The sign of the result is the same as the sign of the first operand.
- If the parity of the two signs is 1:
  1. Subtract the second magnitude from the first.
  2. If a borrow occurs:
    - take the two's complement of result
    - and make the result sign the complement of the sign of the first operand.
  3. Overflow will never occur.

# Sign-Magnitude Arithmetic Examples

---

- Example 1:    **0010**

$$+ \underline{0101}$$

- Example 2:    **0010**

$$+ \underline{1101}$$

- Example 3:    **1010**

$$- \underline{0101}$$

# Signed-Complement Arithmetic

---

- **Addition:**

1. Add the numbers including the sign bits, discarding a carry out of the sign bits (2's Complement), or using an end-around carry (1's Complement).
2. If the sign bits were the same for both numbers and the sign of the result is different, an overflow has occurred.
3. The sign of the result is computed in step 1.

- **Subtraction:**

Form the complement of the number you are subtracting and follow the rules for addition.

# Signed 2's Complement Examples

---

- Example 1: 1101

$$+ \underline{\underline{0011}}$$

- Example 2: 1101

$$- \underline{\underline{0011}}$$

# Signed Binary Addition

## EXAMPLE 3-21 Signed Binary Addition Using 2s Complement

$$\begin{array}{r} +6 \quad 00000110 \quad -6 \quad 11111010 \quad +6 \quad 00000110 \quad -6 \quad 11111010 \\ +13 \quad \underline{00001101} \quad +13 \quad \underline{00001101} \quad -13 \quad \underline{11110011} \quad -13 \quad \underline{11110011} \\ +19 \quad 00010011 \quad +7 \quad 00000111 \quad -7 \quad 11111001 \quad -19 \quad 11101101 \end{array}$$

In each of the four cases, the operation performed is addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2s complement form. ■

# Signed Binary Subtraction

## EXAMPLE 3-22 Signed Binary Subtraction Using 2s Complement

$$\begin{array}{r} -6 \\ -(-13) \\ \hline +7 \end{array} \quad \begin{array}{r} 11111010 \\ -11110011 \\ \hline 00000111 \end{array} \quad \begin{array}{r} 11111010 \\ +00001101 \\ \hline 00000111 \end{array} \quad \begin{array}{r} +6 \\ -(-13) \\ \hline +19 \end{array} \quad \begin{array}{r} 00000110 \\ -11110011 \\ \hline 00010011 \end{array} \quad \begin{array}{r} 00000110 \\ +00001101 \\ \hline 00010011 \end{array}$$

The end carry is discarded. ■

# Overflow Detection

---

- *Overflow occurs if  $n + 1$  bits are required to contain the result from an n-bit addition or subtraction*
- **Overflow can occur for:**
  - Addition of two operands with the same sign
  - Subtraction of operands with different signs
- **Signed number overflow cases with correct result sign**

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 1 \\ + 0 \quad -1 \quad -0 \quad +1 \\ \hline 0 \quad 0 \quad 1 \quad 1 \end{array}$$

- **Detection can be performed by examining the result signs which should match the signs of the top operand**

# Overflow Detection

---

- Signed number cases with carries  $C_n$  and  $C_{n-1}$  shown for correct result signs:

$$\begin{array}{ccccccccc}
 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & & 0 & & 1 & & & 1 \\
 +\frac{0}{0} & -\frac{1}{1} & -\frac{0}{0} & +\frac{1}{1} \\
 0 & 0 & 1 & 1
 \end{array}$$

- Signed number cases with carries shown for erroneous result signs (indicating overflow):

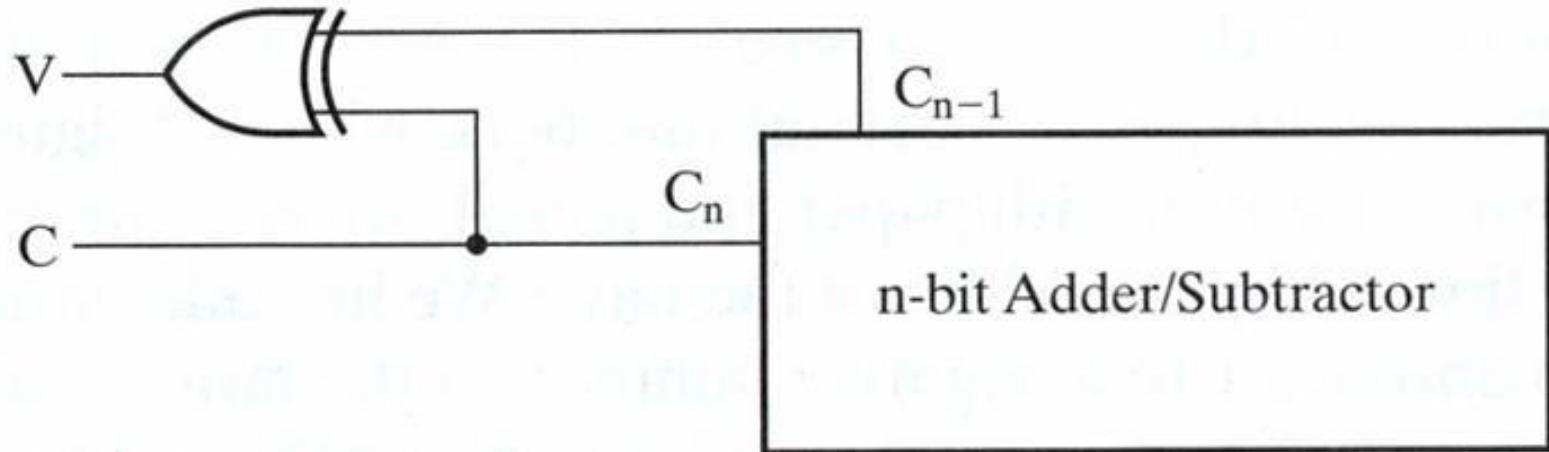
$$\begin{array}{ccccccccc}
 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
 0 & & 0 & & 1 & & & 1 \\
 +\frac{0}{1} & -\frac{1}{1} & -\frac{0}{0} & +\frac{1}{0} \\
 1 & 1 & 0 & 0
 \end{array}$$

○

- Simplest way to implement overflow  $V = C_n + C_{n-1}$

# Overflow Detection

---



□ **FIGURE 3-46**  
Overflow Detection Logic for Addition and Subtraction

# BINARY MULTIPLICATION

*Example:* Multiply the binary numbers 111 and 101.

$$\begin{array}{r} & 1 & 1 & 1 & \text{Multiplicand} \\ \times & 1 & 0 & 1 & \text{Multiplier} \\ \hline & 1 & 1 & 1 & \text{1st partial product} \\ & 0 & 0 & 0 & \text{2nd partial product} \\ & 1 & 1 & 1 & \text{3rd partial product} \\ \hline & 1 & 0 & 0 & 0 & 1 & 1 & \text{Product} \end{array}$$

111 x 101 can also be calculated: 111 + 111 + 111 + 111 + 111



# TEST

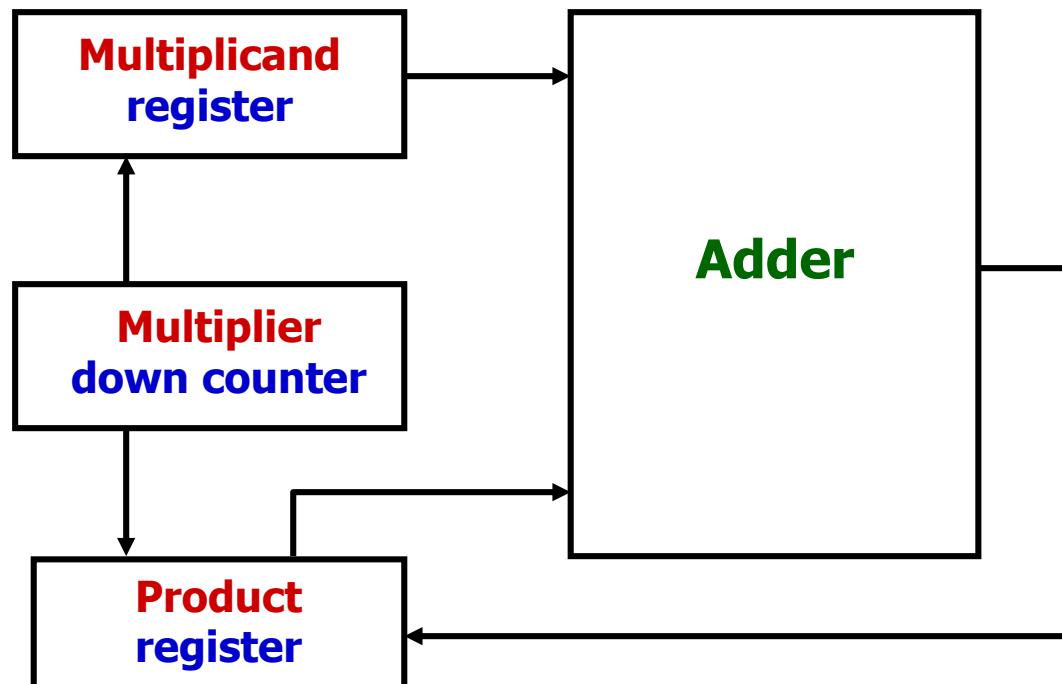
Multiply the binary numbers 101 and 100.

$$\begin{array}{r} 1 & 0 & 1 \\ \times & 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 \end{array}$$

# BINARY MULTIPLIERS

Binary multiplier circuits - utilize repeated addition.

Block  
Diagram:



# Binary Multiplier

$$\begin{array}{r} B_1 \quad B_0 \\ A_1 \quad A_0 \\ \hline A_0B_1 \quad A_0B_0 \\ A_1B_1 \quad A_1B_0 \\ \hline C_3 \quad C_2 \quad C_1 \quad C_0 \end{array}$$

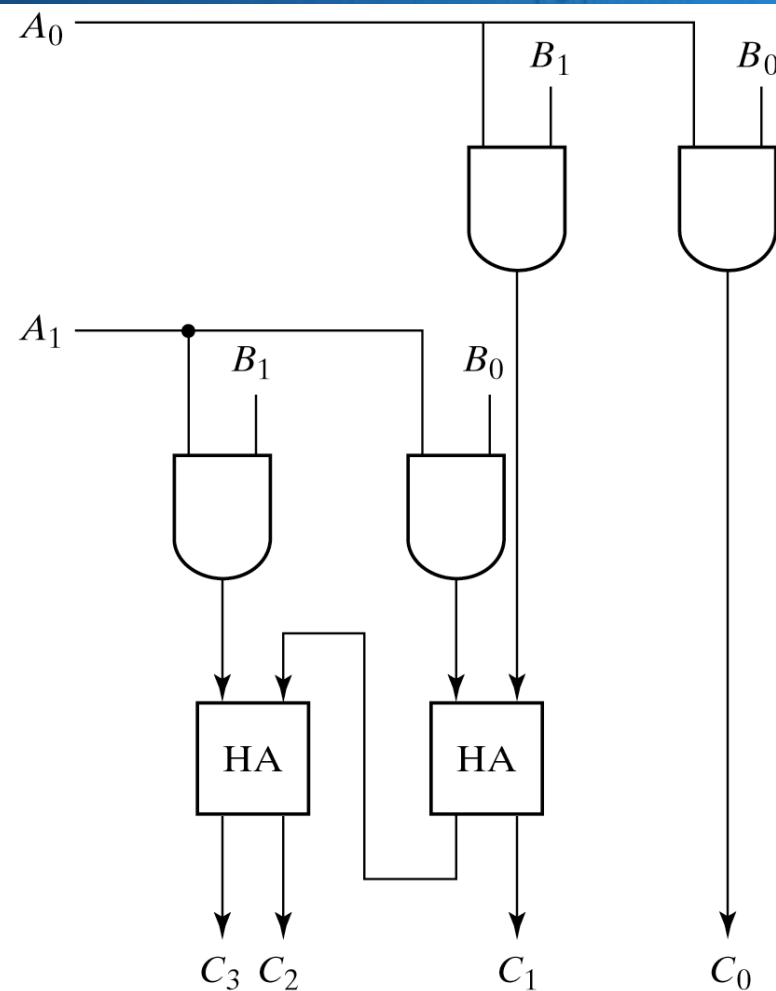


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

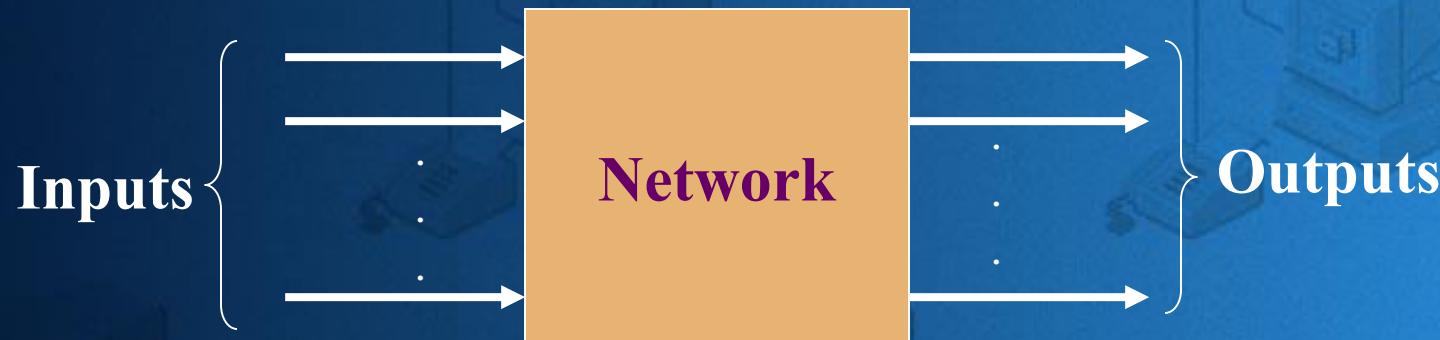
---

# **CSE1003-Digital Logic Design**

## **Module-4 COMBINATIONAL CIRCUITS-II**

<b>Module:3</b>	<b>COMBINATIONAL CIRCUIT - I</b>	<b>4 hours</b>
Adder - Subtractor - Code Converter - Analyzing a Combinational Circuit		
<b>Module:4</b>	<b>COMBINATIONAL CIRCUIT -II</b>	<b>6 hours</b>
Binary Parallel Adder- Look ahead carry - Magnitude Comparator - Decoders – Encoders - Multiplexers –Demultiplexers.		

# Remember



## ◆ Combinational

- The outputs depend only on the current input values
- It uses only logic gates

## ◆ Sequential

- The outputs depend on the current and past input values

**It uses logic gates and storage elements**

# Notes

- ◆ If there are  $n$  input variables, there are  $2^n$  input combinations
- ◆ For each input combination, there is one output value
- ◆ Truth tables are used to list all possible combinations of inputs and corresponding output values

# Basic Combinational Circuits

- ◆ Adders
- ◆ Subtractors
- ◆ Multipliers
- ◆ Comparators
- ◆ Decoders
- ◆ Encoders
- ◆ Multiplexers
- ◆ Demultiplexers

# Part-1-Design Procedure

- ◆ Determine the inputs and outputs
- ◆ Assign a symbol for each
- ◆ Derive the truth table
- ◆ Get the simplified boolean expression for each output
- ◆ Draw the network diagram

# Magnitude Comparator

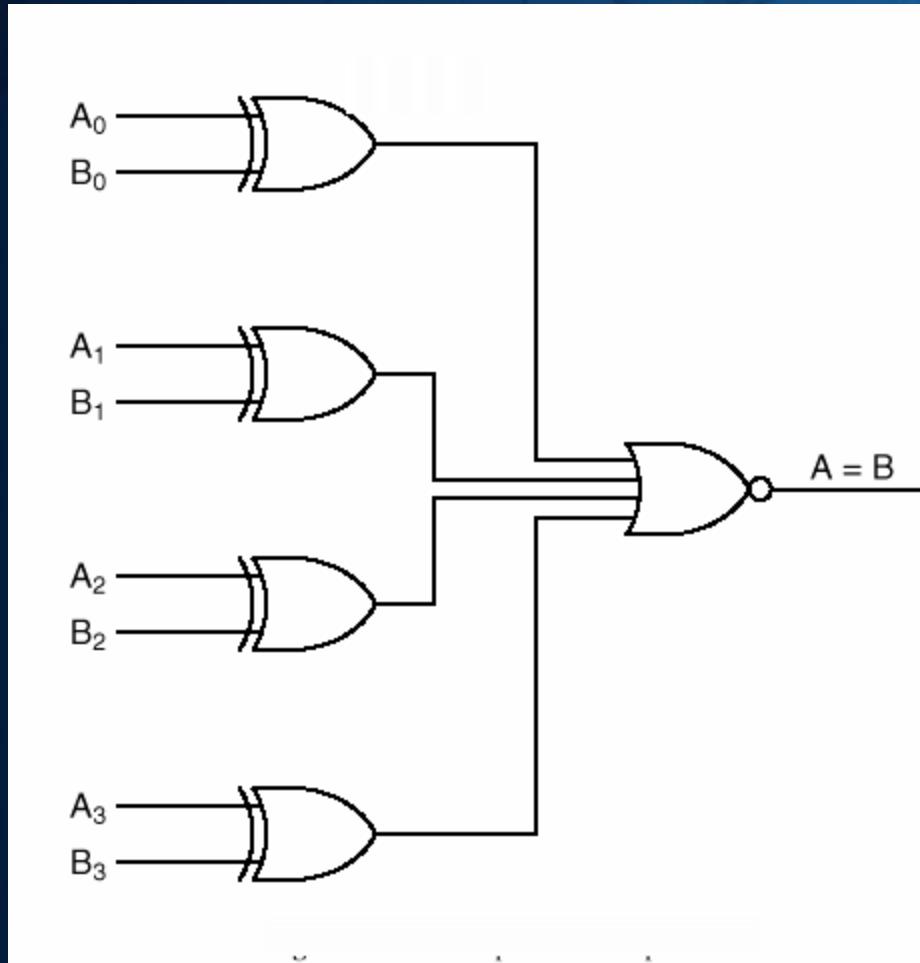
- A magnitude digital comparator is a combinational circuit that compares two digital or binary numbers (consider A and B) and determines their relative magnitudes in order to find out whether one number is equal, less than or greater than the other digital number.
- Three binary variables are used to indicate the outcome of the comparison as  $A > B$ ,  $A < B$ , or  $A = B$ . The below figure shows the block diagram of a n-bit comparator which compares the two numbers of n-bit length and generates their relation between themselves.

# MAGNITUDE COMPARATOR: DIGITAL COMPARATOR

- It is a combinational logic circuit.
- Digital Comparator is used to compare the value of two binary digits.
- There are two types of digital comparator (i) Identity Comparator  
(ii) Magnitude Comparator.
- IDENTITY COMPARATOR: This comparator has only one output terminal for when  $A=B$ , either  $A=B=1$  (High) or  $A=B=0$  (Low)
- MAGNITUDE COMPARATOR: This Comparator has three output terminals namely  $A>B$ ,  $A=B$ ,  $A<B$ . Depending on the result of comparison, one of these output will be high (1)
- Block Diagram of Magnitude Comparator is shown in Fig. 1

# Identity Comparator

- ◆ Compare two input words



- ◆ Returns 1 if  $A=B$ , 0 otherwise

# BLOCK DIAGRAM OF MAGNITUDE COMPARATOR

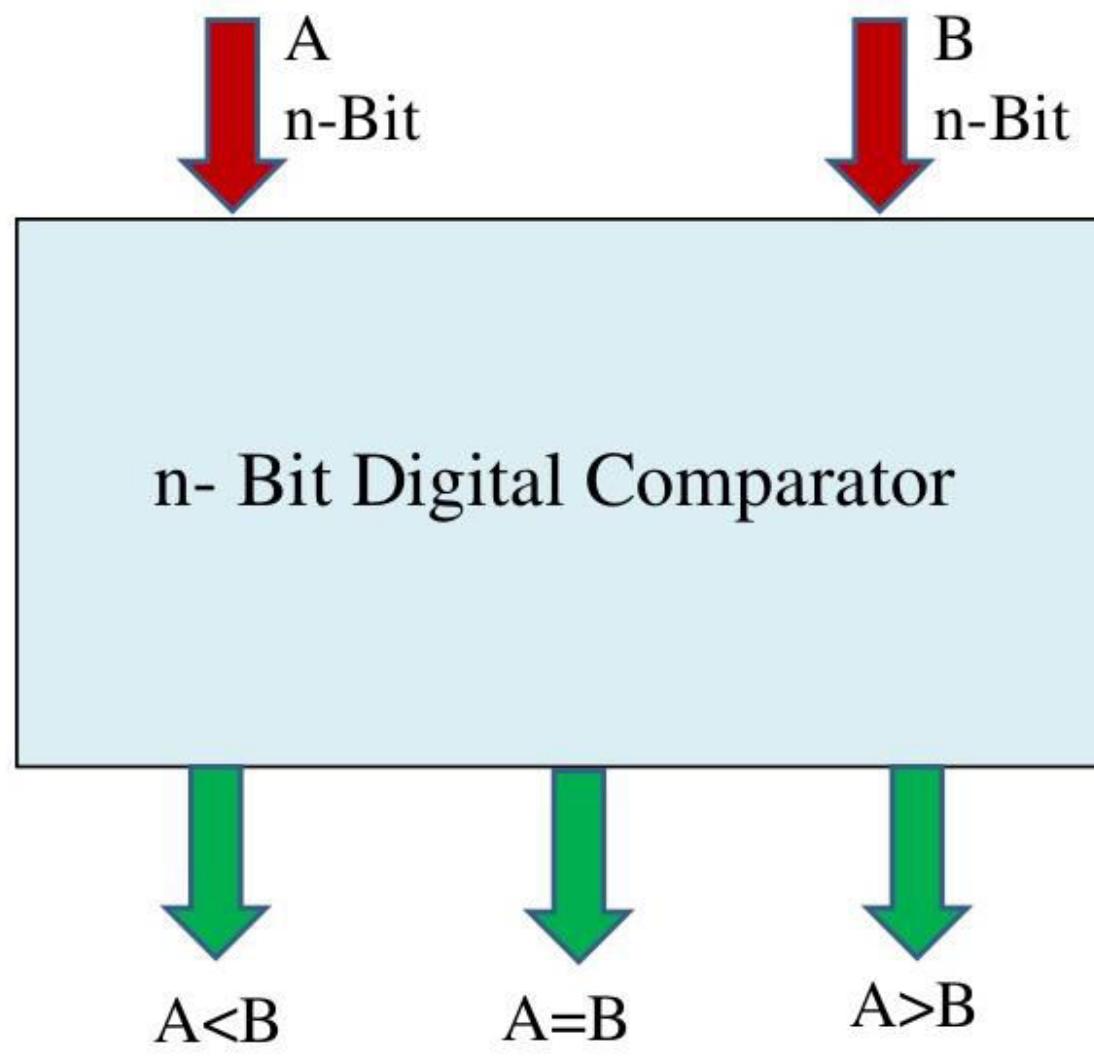


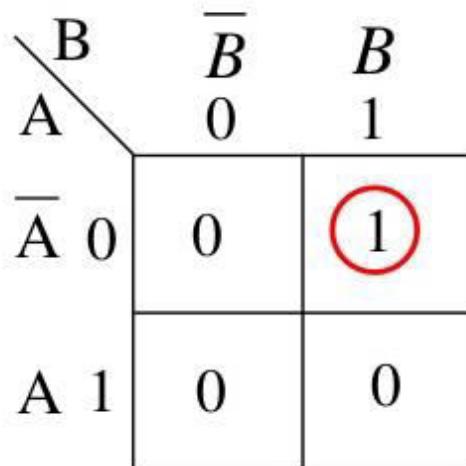
Fig. 1

## 1- Bit Magnitude Comparator:

- This magnitude comparator has two inputs A and B and three outputs  $A < B$ ,  $A = B$  and  $A > B$ .
- This magnitude comparator compares the two numbers of single bits.
- Truth Table of 1-Bit Comparator

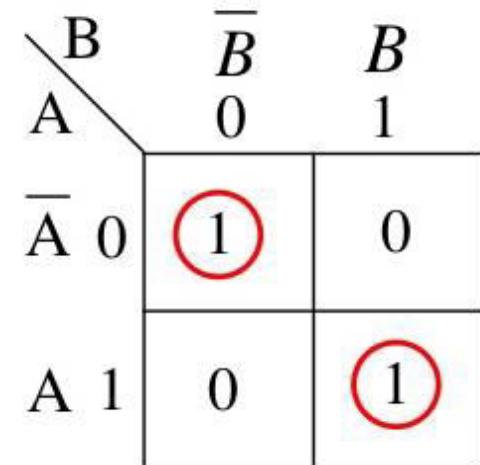
INPUTS		OUTPUTS		
A	B	$Y_1$ ( $A < B$ )	$Y_2$ ( $A = B$ )	$Y_3$ ( $A > B$ )
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

## K-Maps For All Three Outputs :



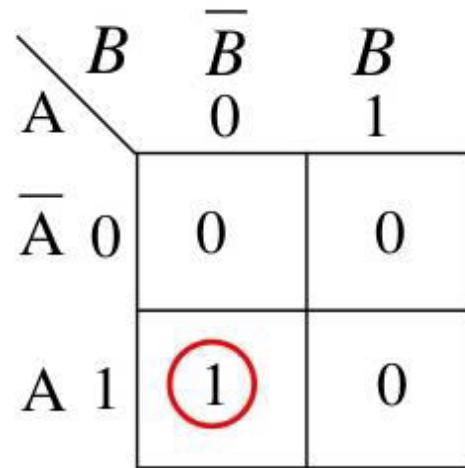
K-Map for  $Y_1$  :  $A < B$

$$Y_1 = \bar{A}\bar{B}$$



K-Map for  $Y_2$  :  $A = B$

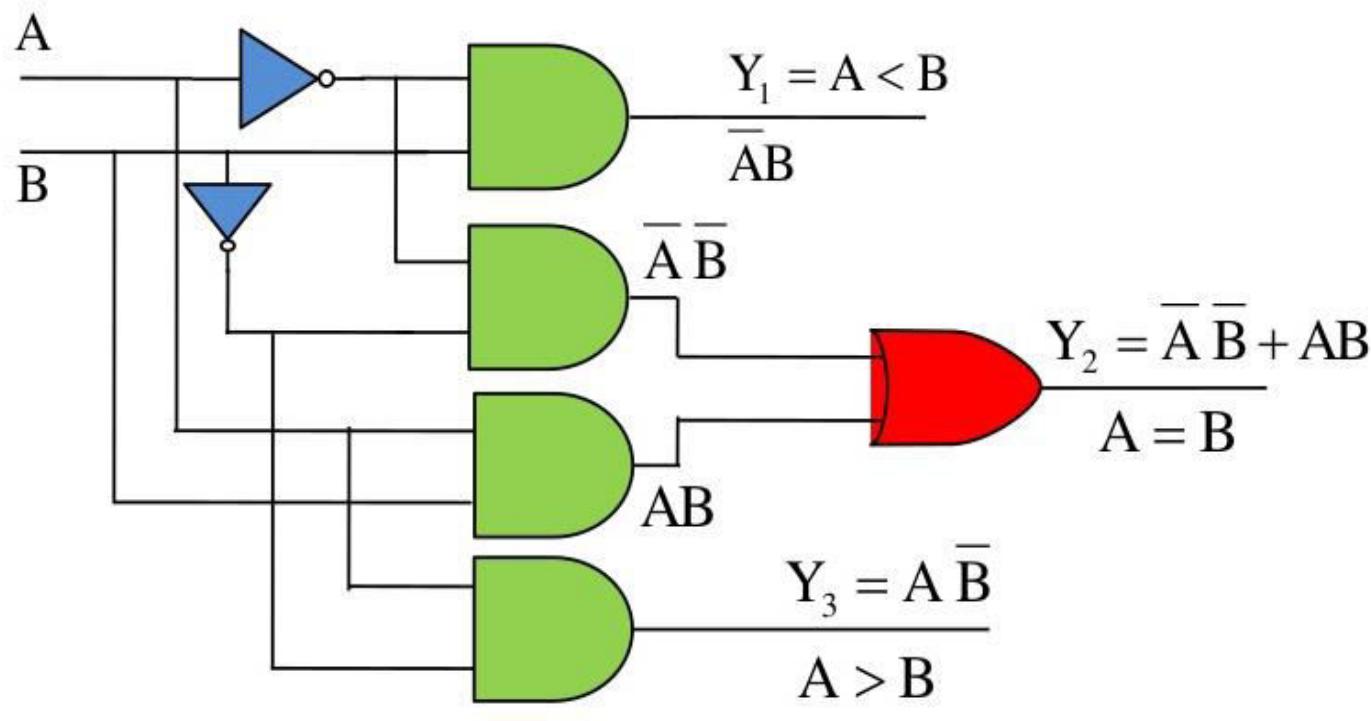
$$Y_2 = \bar{A}\bar{B} + AB$$



K-Map for  $Y_2$  :  $A > B$

$$Y_3 = A\bar{B}$$

## Realization of One Bit Comparator



$$Y_1 = \overline{AB}$$

$$Y_2 = \overline{\overline{A}}\overline{\overline{B}} + AB$$

$$Y_3 = A\overline{B}$$

## 2-Bit Comparator:

- A comparator which is used to compare two binary numbers each of two bits is called a 2-bit magnitude comparator.
- Fig. 2 shows the block diagram of 2-Bit magnitude comparator.
- It has four inputs and three outputs.
- Inputs are  $A_0, A_1, B_0$  and  $B_1$  and Outputs are  $Y_1, Y_2$  and  $Y_3$

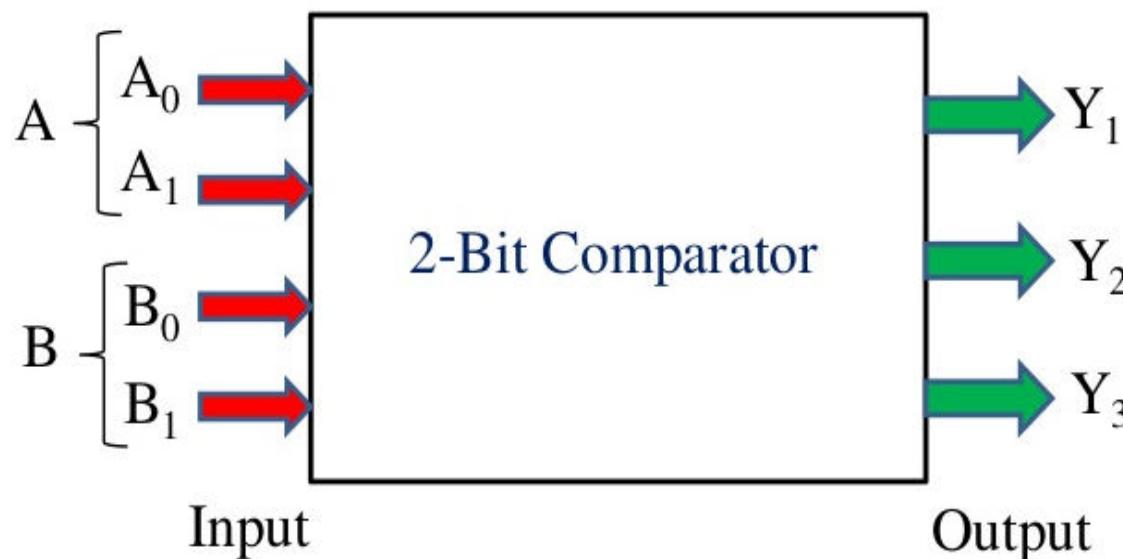


Fig. 2

## GREATER THAN (A>B)

$A_1$	$A_0$	$B_1$	$B_0$
1	0	0	1
1	1	1	0
0	1	0	0

1. If  $A_1 = 1$  and  $B_1 = 0$  then  $A > B$
2. If  $A_1$  and  $B_1$  are same, i.e  $A_1 = B_1 = 1$  or  $A_1 = B_1 = 0$  and  $A_0 = 1$ ,  $B_0 = 0$  then  $A > B$

## LESS THAN (A<B)

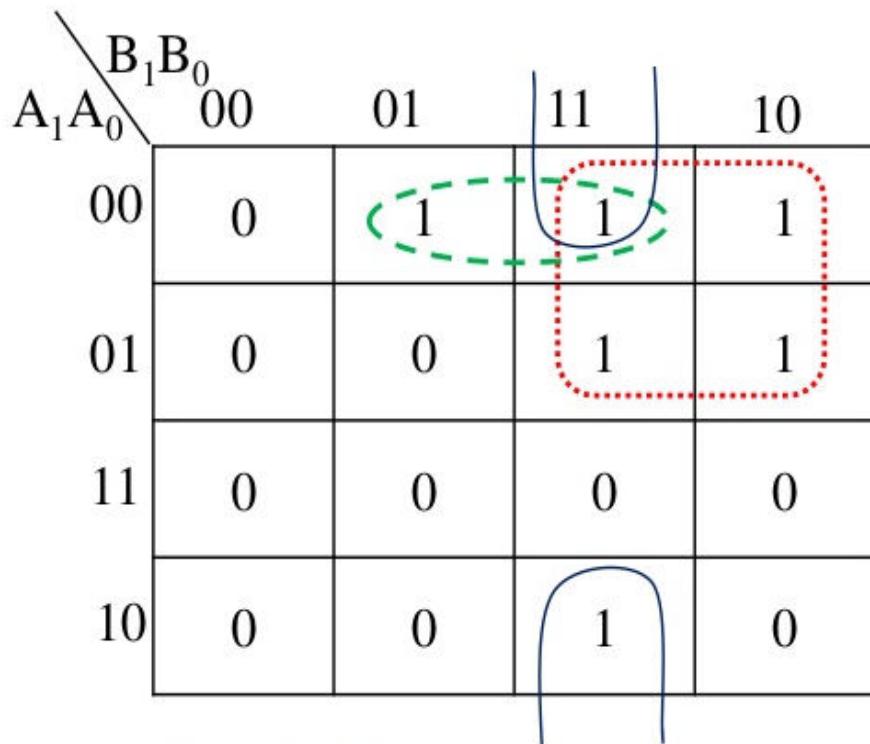
Similarly,

1. If  $A_1 = B_1 = 1$  and  $A_0 = 0$ ,  $B_0 = 1$ , then  $A < B$
2. If  $A_1 = B_1 = 0$  and  $A_0 = 0$ ,  $B_0 = 1$  then  $A < B$

## TRUTH TABLE

INPUT				OUTPUT		
$A_1$	$A_0$	$B_1$	$B_0$	$Y_1 = A < B$	$Y_2 = (A = B)$	$Y_3 = A > B$
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

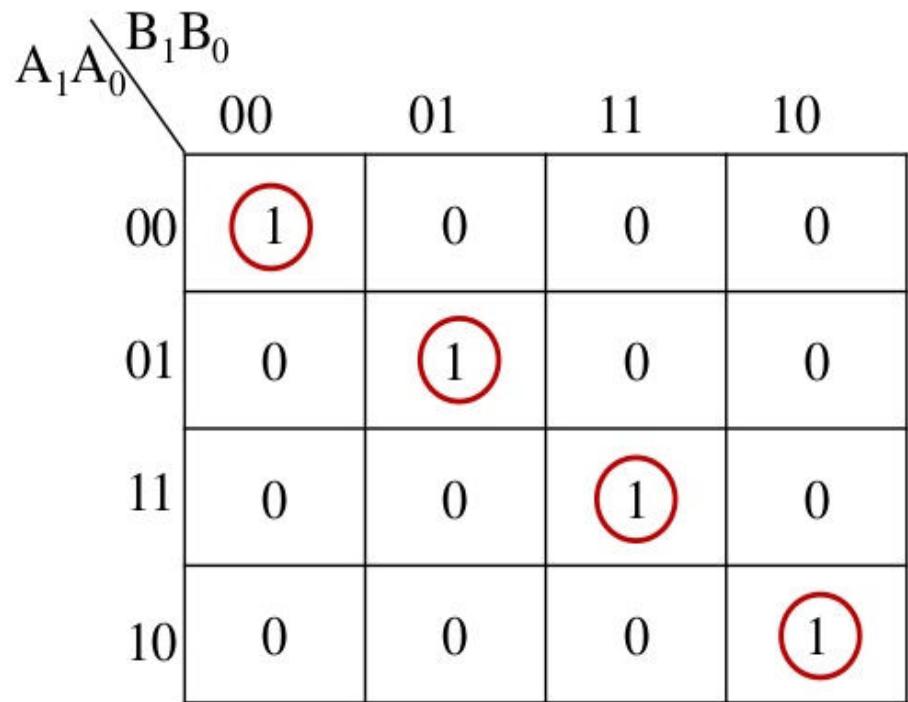
K-Map for A<B:



For A<B

$$Y_1 = \overline{A}_1 \overline{A}_0 B_0 + \overline{A}_1 B_1 + \overline{A}_0 B_1 B_0$$

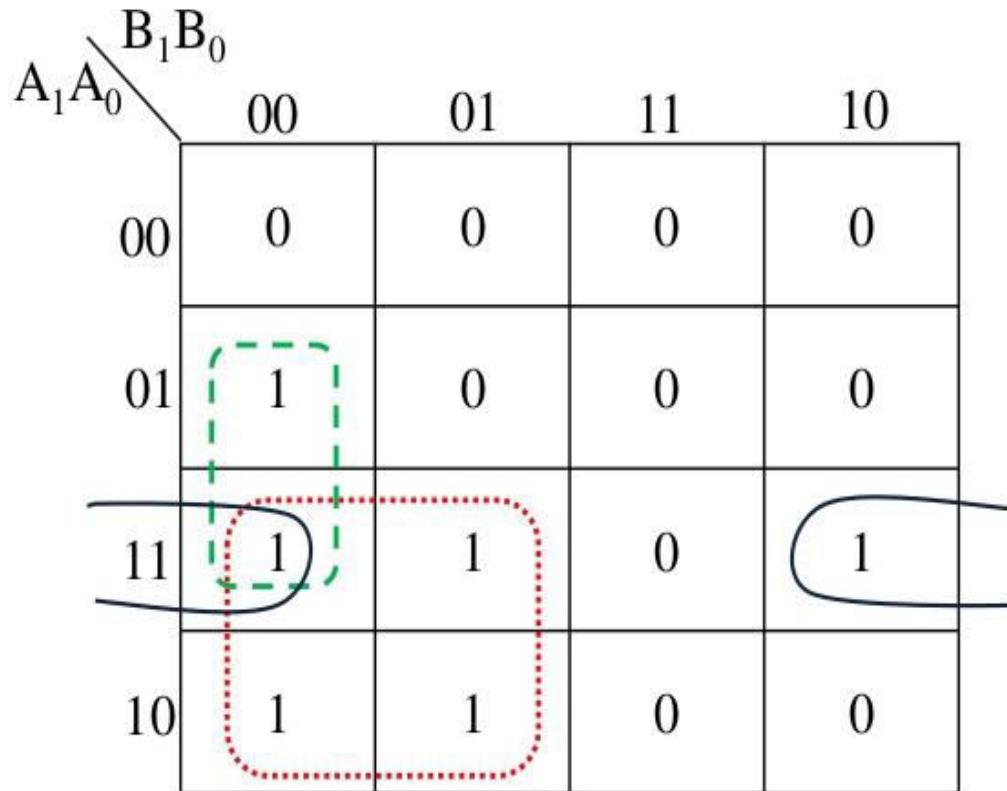
K-Map for A=B:



For A=B

$$Y_2 = \overline{A}_1 \overline{A}_0 \overline{B}_1 \overline{B}_0 + \overline{A}_1 A_0 \overline{B}_1 B_0 + A_1 A_0 B_1 B_0 + A_1 \overline{A}_0 B_1 \overline{B}_0$$

## K-Map For A>B



$$Y_3 = A_0 \overline{B}_1 \overline{B}_0 + A_1 \overline{B}_1 + A_1 A_0 \overline{B}_0$$

For A=B From K-Map

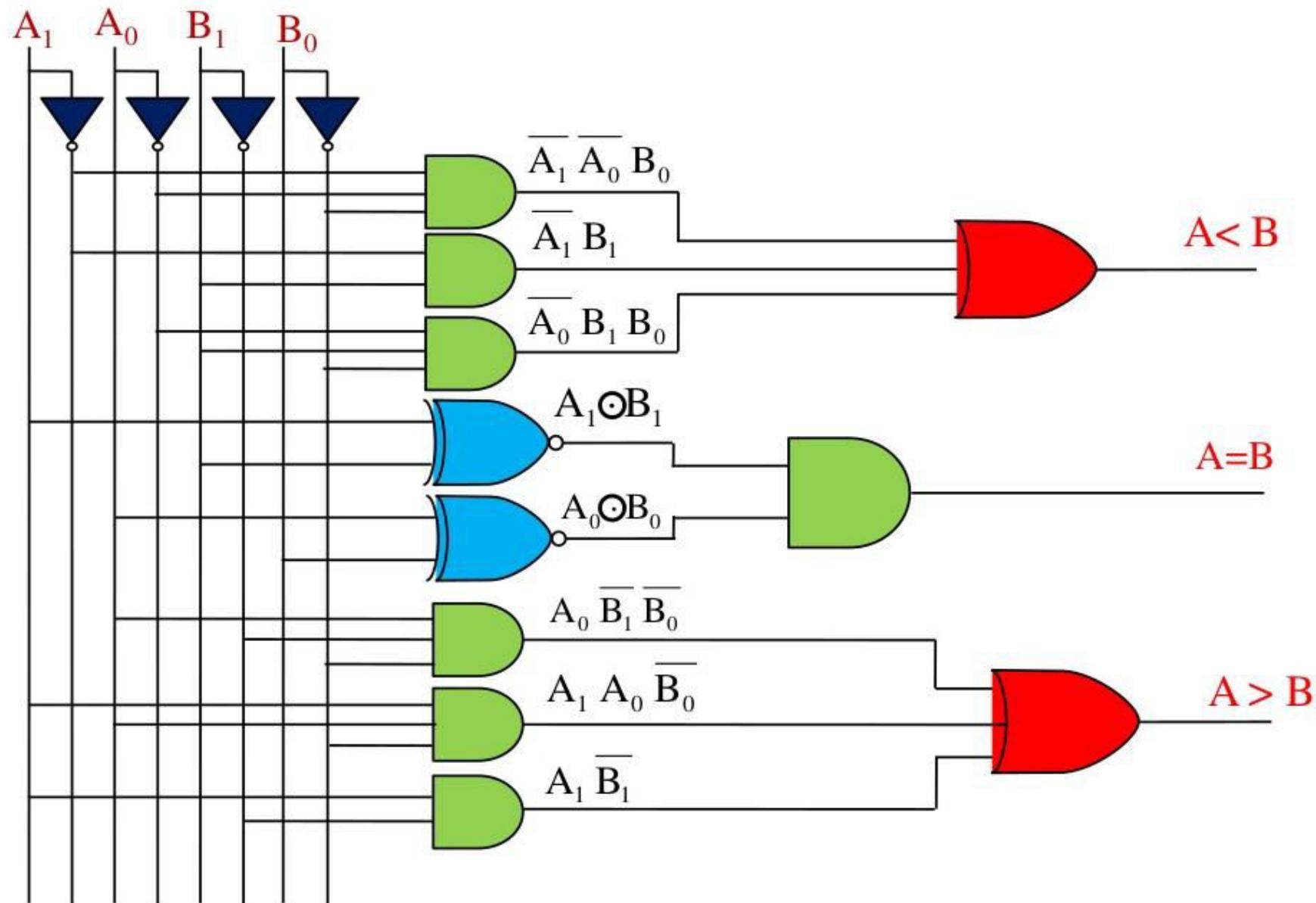
$$Y_2 = \overline{A_1} \overline{A_0} \overline{B_1} \overline{B_0} + \overline{A_1} A_0 \overline{B_1} B_0 + A_1 A_0 B_1 B_0 + A_1 \overline{A_0} B_1 \overline{B_0}$$

$$Y_2 = \overline{A_0} B_0 (A_1 \overline{B_1} + A_1 B_1) + A_0 B_0 (\overline{A_1} \overline{B_1} + A_1 B_1)$$

$$Y_2 = (\overline{A_1} \overline{B_1} + A_1 B_1) (\overline{A_0} \overline{B_0} + A_0 B_0)$$

$$Y_2 = (A_1 \odot B_1) (A_0 \odot B_0)$$

## LOGIC DIAGRAM OF 2-BIT COMPARATOR:



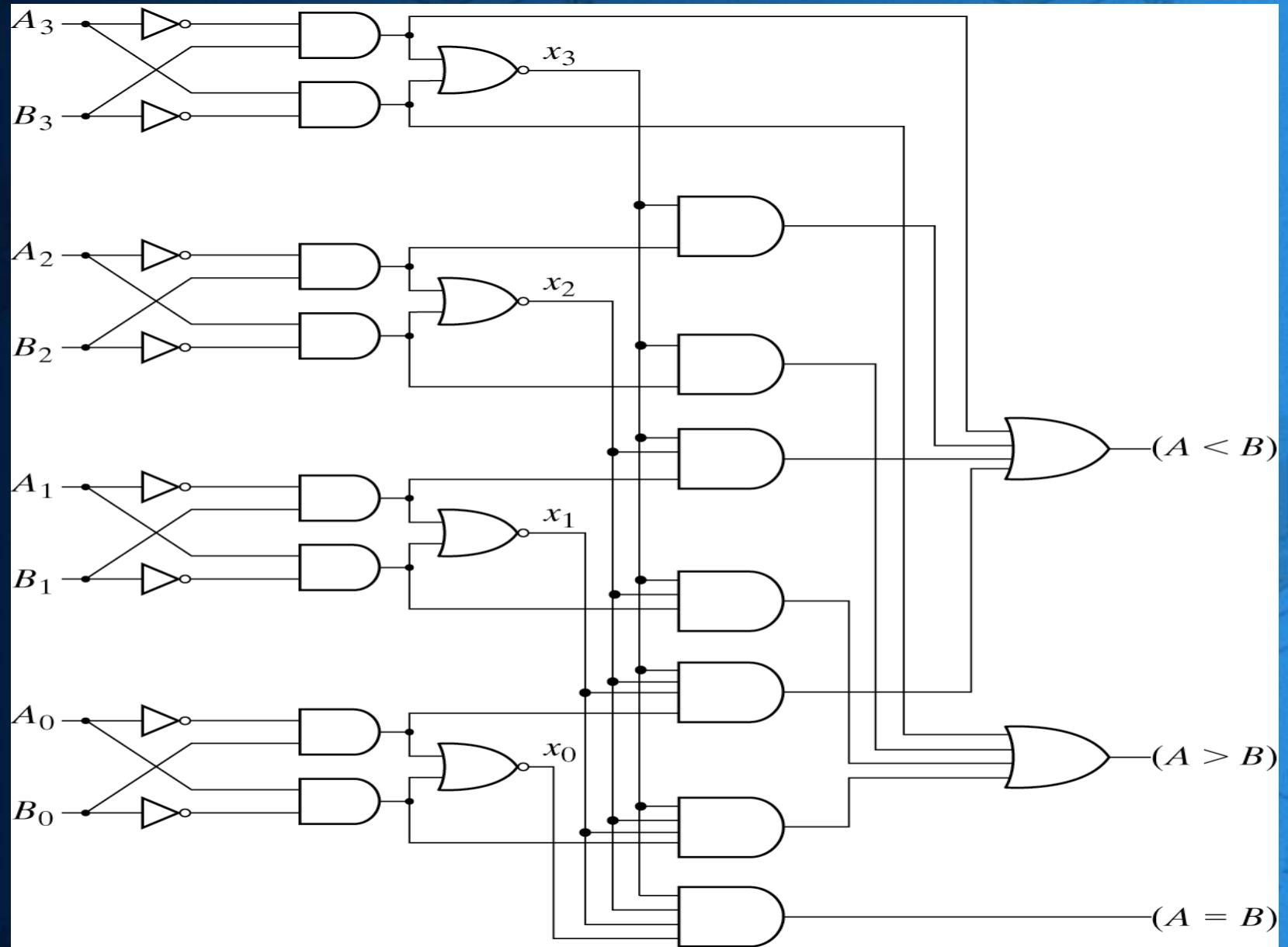
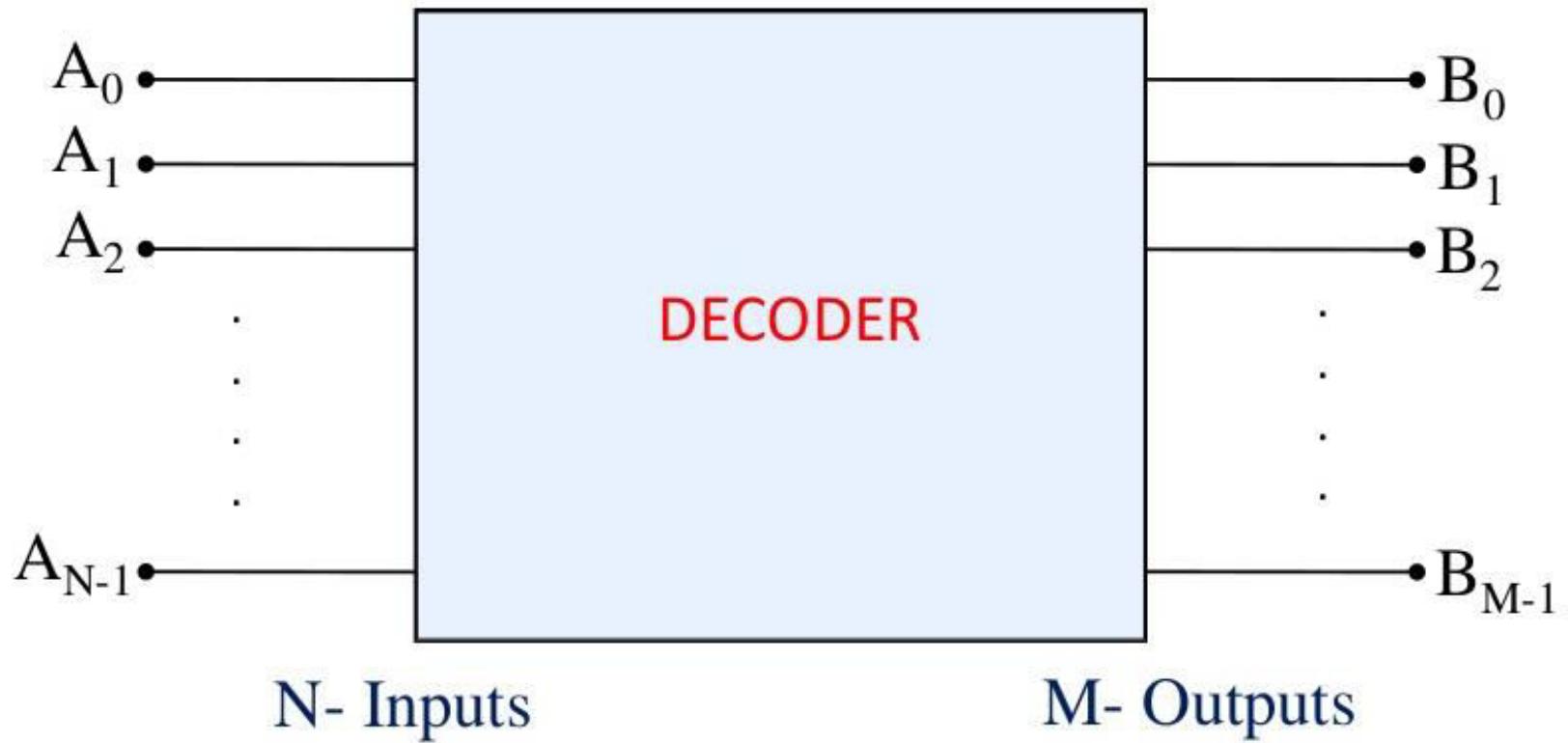


Fig. 4-17 4-Bit Magnitude Comparator

## DECODER

- A decoder is a combinational circuit.
- A decoder accepts a set of inputs that represents a binary number and activates only that output corresponding to the input number. All other outputs remain inactive.
- Fig. 1 shows the block diagram of decoder with ‘N’ inputs and ‘M’ outputs.
- There are  $2^N$  possible input combinations, for each of these input combination only one output will be HIGH (active) all other outputs are LOW
- Some decoder have one or more ENABLE (E) inputs that are used to control the operation of decoder.

## BLOCK DIAGRAM OF DECODER



*Only one output is High for each input*

Fig. 1

## 2 to 4 Line Decoder:

- Block diagram of 2 to 4 decoder is shown in fig. 2
- A and B are the inputs. ( No. of inputs =2)
- No. of possible input combinations:  $2^2=4$
- No. of Outputs :  $2^2=4$ , they are indicated by  $D_0$ ,  $D_1$ ,  $D_2$  and  $D_3$
- From the Truth Table it is clear that each output is “1” for only specific combination of inputs.

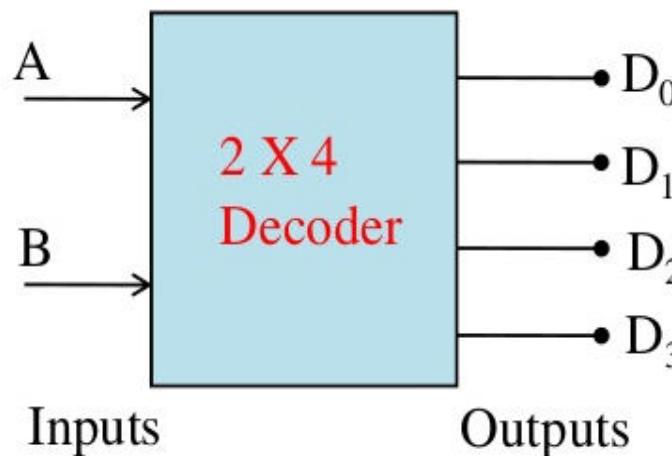


Fig. 2

TRUTH TABLE

INPUTS		OUTPUTS			
A	B	$D_0$	$D_1$	$D_2$	$D_3$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

## BOOLEAN EXPRESSION:

From Truth Table

$$D_0 = \overline{A} \overline{B}$$

$$D_1 = A \overline{B}$$

$$D_2 = \overline{A} B$$

$$D_3 = AB$$

## LOGIC DIAGRAM:

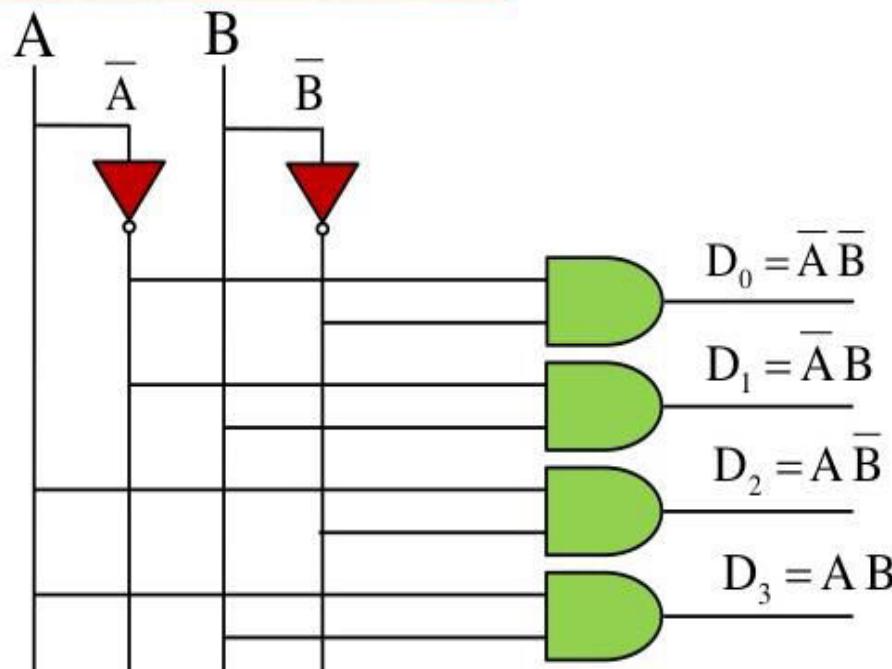
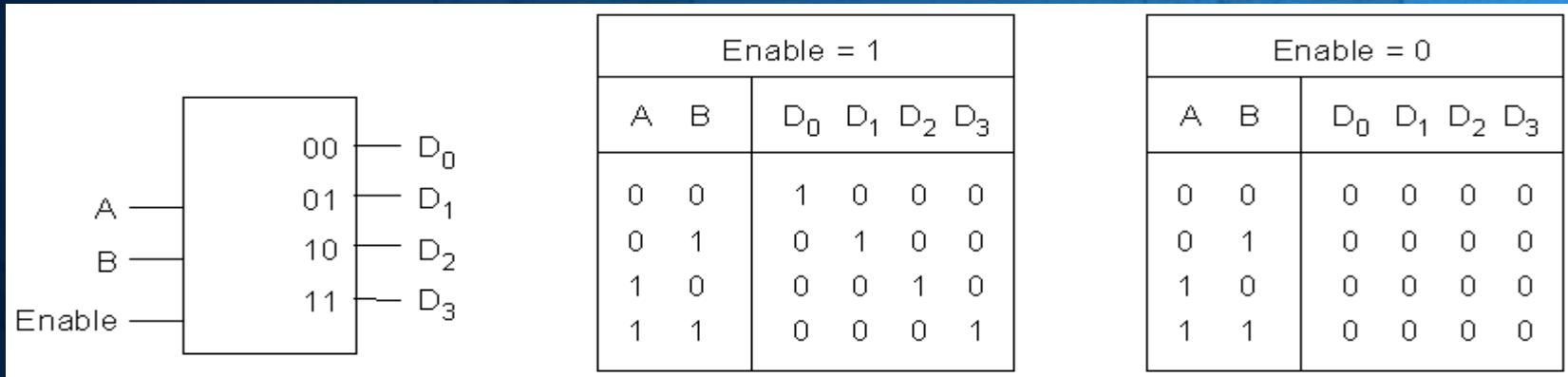


Fig. 3

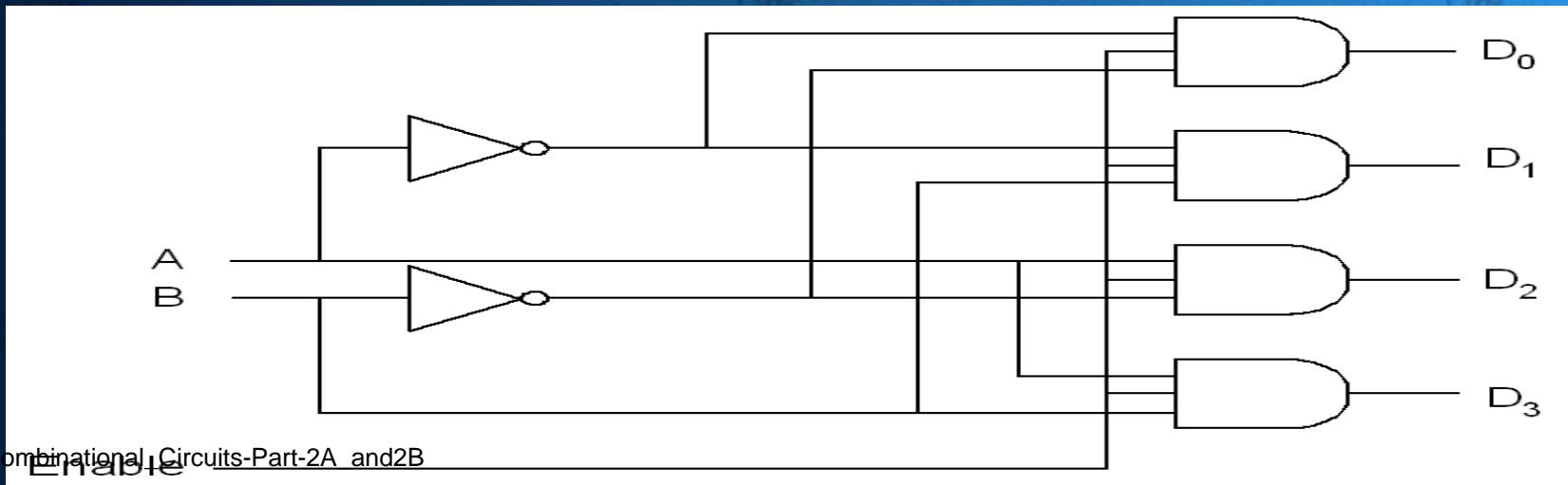
# Decoder

- ◆ n by  $2^n$  decoder
  - Converts information from n input lines into  $2^n$  output lines
- ◆ Example - 2x4 Decoder, 3x8 Decoder, 4x16 Decoder

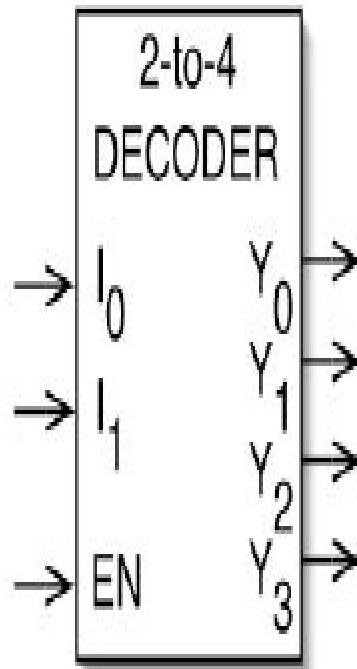
## 2x4 Decoder



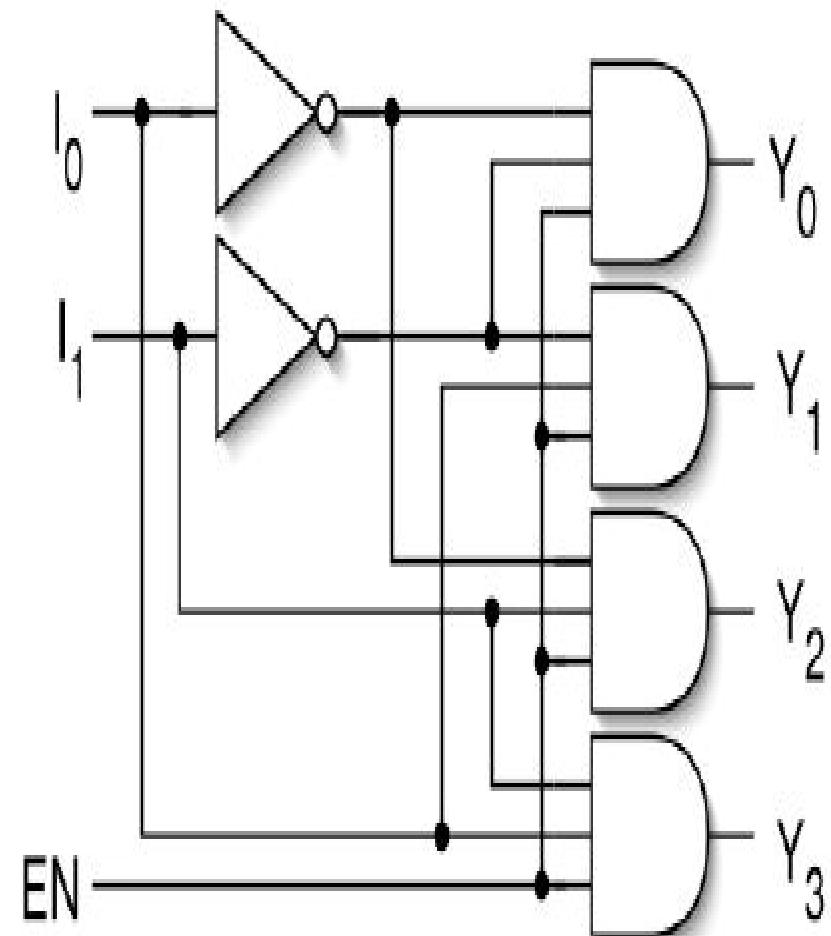
## Internal Structure of 2x4 Decoder



## Another View



EN	$I_1$	$I_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	x	x	0	0	0	0
1	0	0	0	0	0	1
	1	0	1	0	0	0
	1	1	0	0	1	0
	1	1	1	1	0	0



## 3 to 8 Line Decoder:

- Block diagram of 3 to 8 decoder is shown in fig. 4
- A , B and C are the inputs. ( No. of inputs =3)
- No. of possible input combinations:  $2^3=8$
- No. of Outputs :  $2^3=8$ , they are indicated by  $D_0$  to  $D_7$
- From the Truth Table it is clear that each output is “1” for only specific combination of inputs.

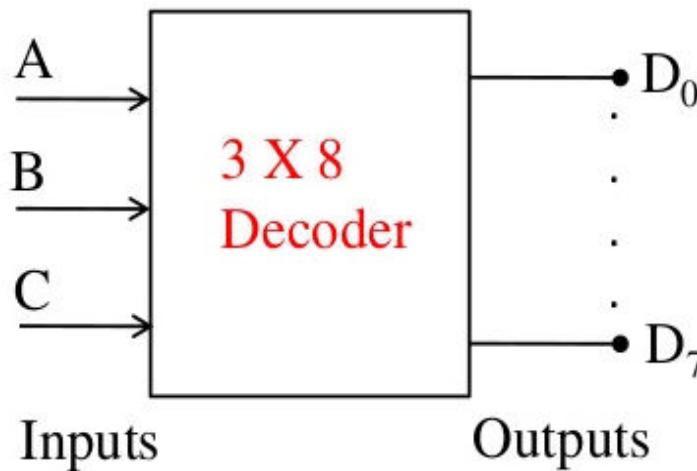
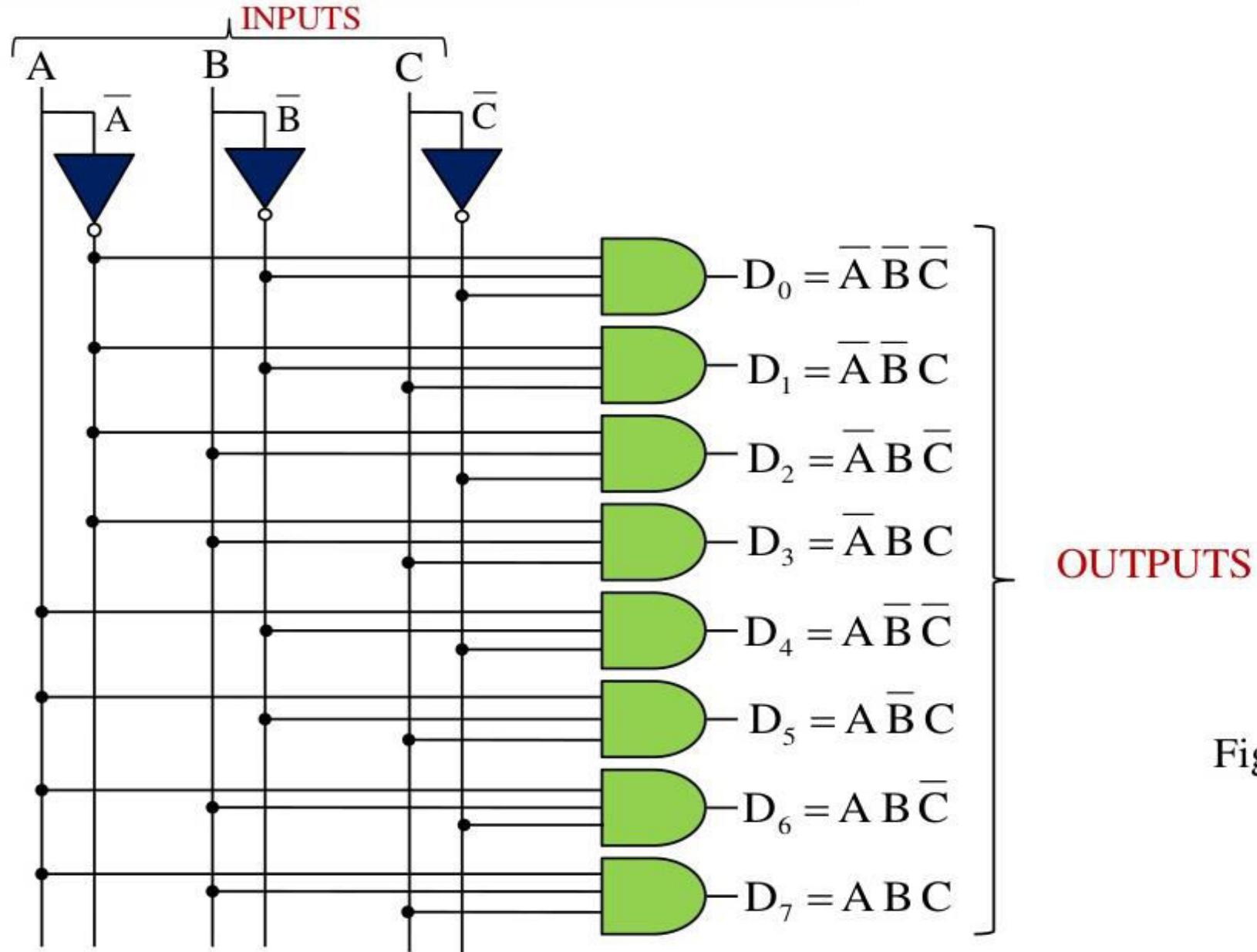


Fig. 4

## TRUTH TABLE FOR 3 X 8 DECODER:

INPUTS			OUTPUTS								
A	B	C	D0	D1	D2	D3	D4	D5	D6	D7	
0	0	0	1	0	0	0	0	0	0	0	$D_0 = \overline{A} \overline{B} \overline{C}$
0	0	1	0	1	0	0	0	0	0	0	$D_1 = \overline{A} \overline{B} C$
0	1	0	0	0	1	0	0	0	0	0	$D_2 = \overline{A} B \overline{C}$
0	1	1	0	0	0	1	0	0	0	0	$D_3 = \overline{A} B C$
1	0	0	0	0	0	0	1	0	0	0	$D_4 = A \overline{B} \overline{C}$
1	0	1	0	0	0	0	0	1	0	0	$D_5 = A \overline{B} C$
1	1	0	0	0	0	0	0	0	1	0	$D_6 = A B \overline{C}$
1	1	1	0	0	0	0	0	0	0	1	$D_7 = A B C$

## LOGIC DIAGRAM OF 3 X 8 DECODER:



## EXPANSION OF DECODERS:

The number of lower order Decoder for implementing higher order Decoder can be find as

$$\text{No. of lower order required} = m_2/m_1$$

Where,  $m_1$ =No. of Outputs of lower order Decoder

$m_2$ =No. of Outputs of higher order Decoder

## 3 x 8 Decoder From 2 x 4 Decoder:

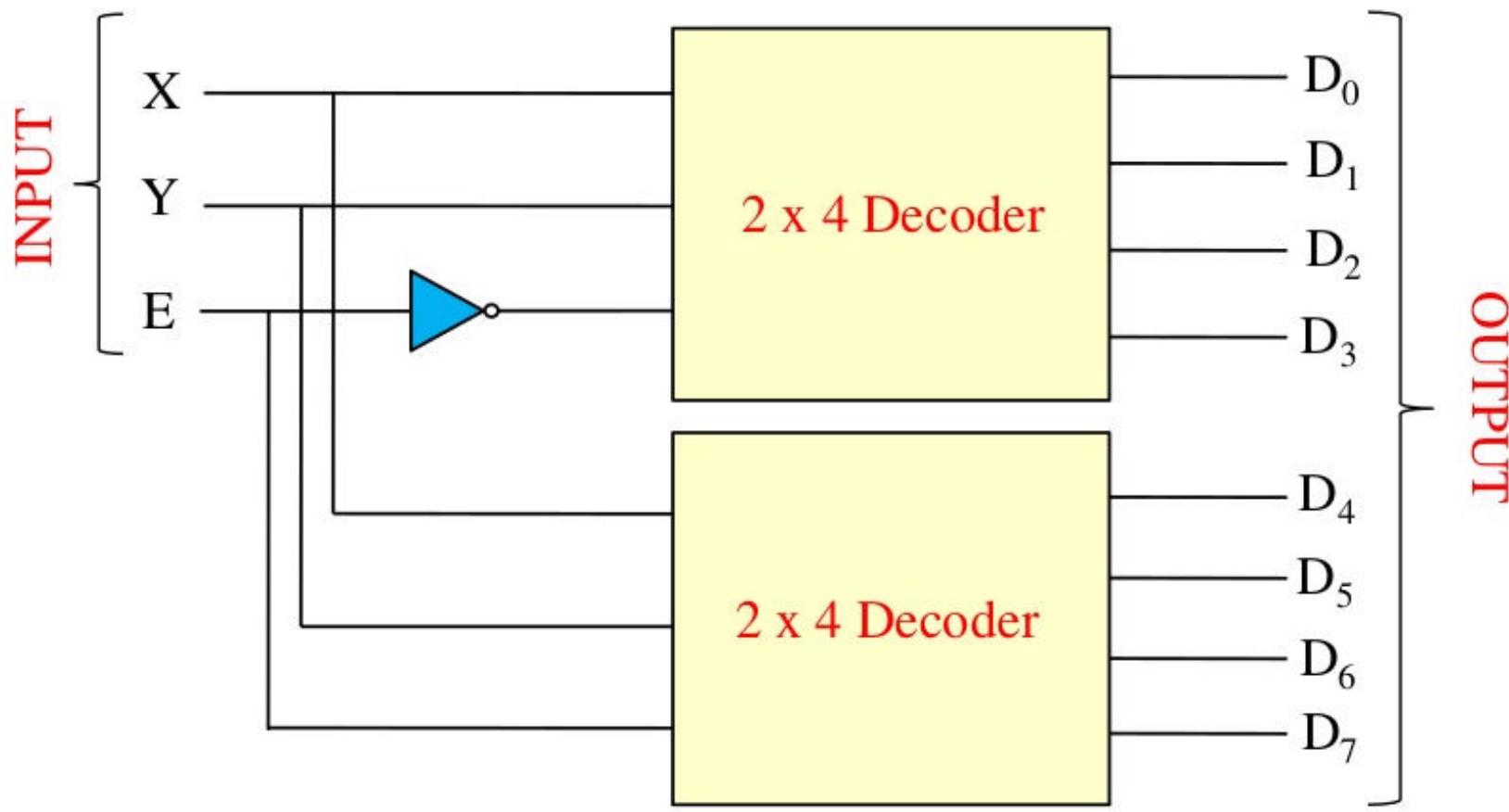


Fig. 6

# 4x16 Decoder

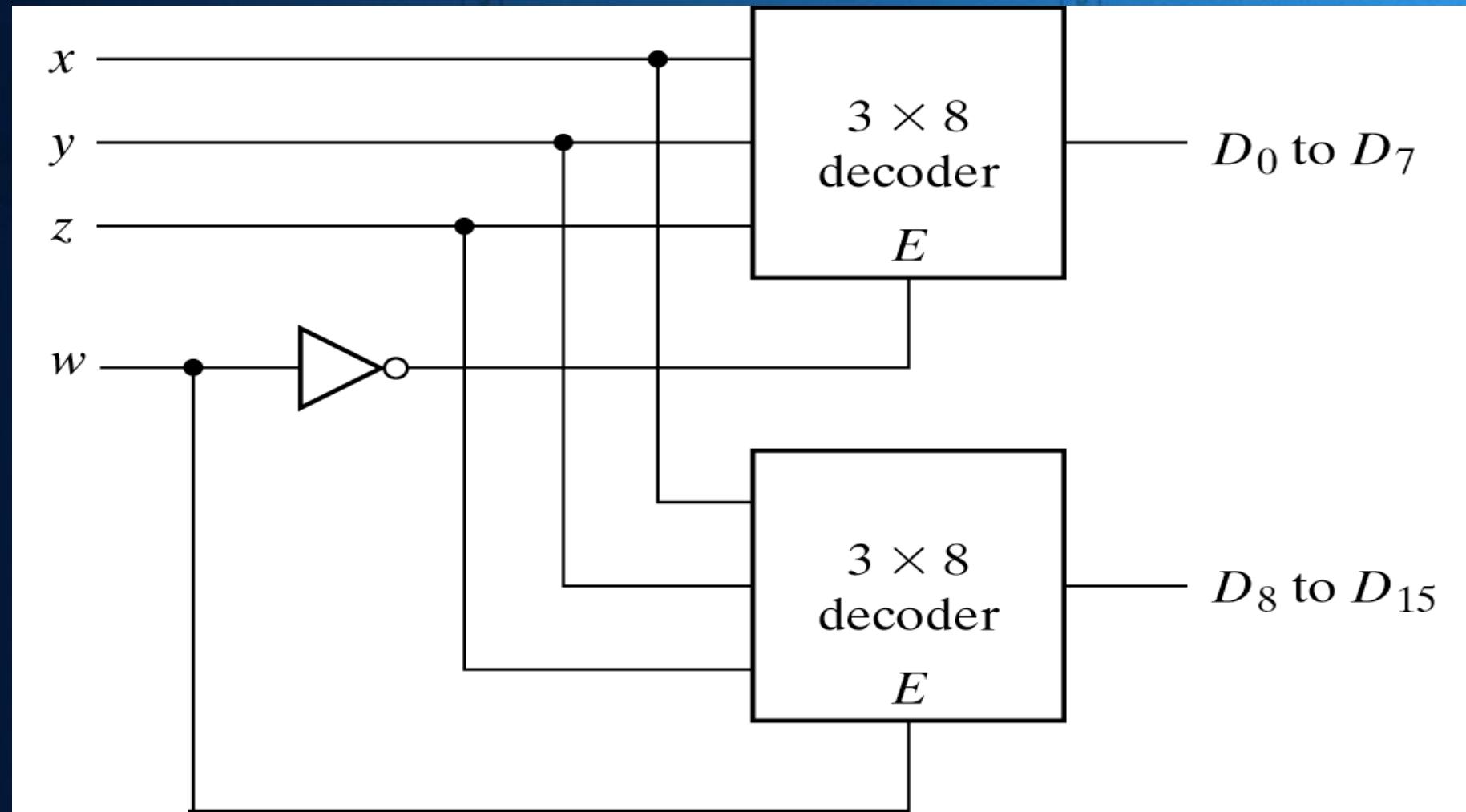


Fig. 4-20  $4 \times 16$  Decoder Constructed with Two  $3 \times 8$  Decoders

**Example:** Implement the following multiple output function using a suitable Decoder.

$$f_1(A, B, C) = \sum m(0, 4, 7) + d(2, 3)$$

$$f_2(A, B, C) = \sum m(1, 5, 6)$$

$$f_3(A, B, C) = \sum m(0, 2, 4, 6)$$

**Solution:**  $f_1$  consists of don't care conditions. So we consider them to be logic 1.

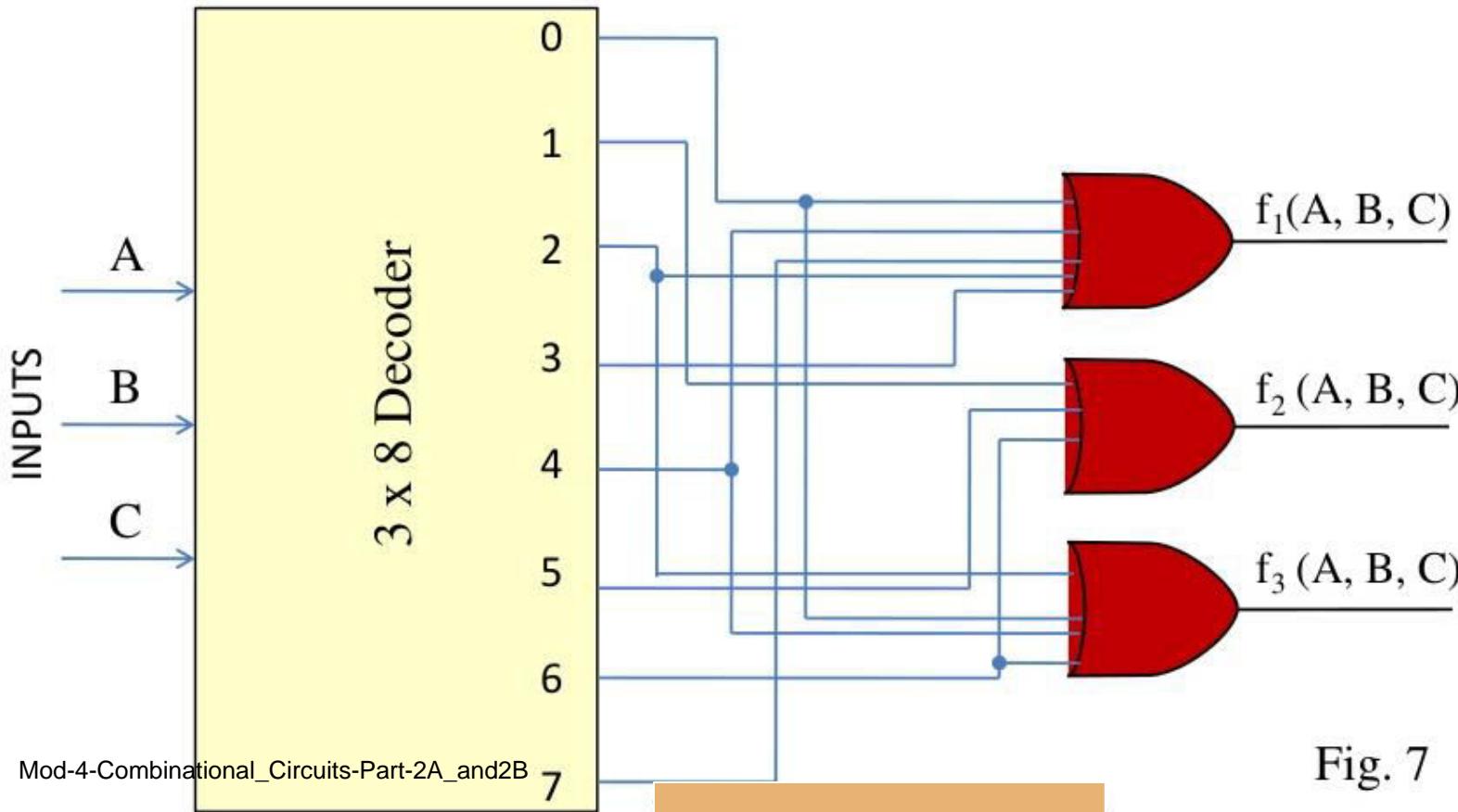


Fig. 7

**EXAMPLE:** Implement the following Boolean function using suitable Decoder.

$$f_1(x,y,z) = \sum m(1,5,7)$$

$$f_2(x,y,z) = \sum m(0,3)$$

$$f_3(x,y,z) = \sum m(2,4,5)$$

**Solution:**

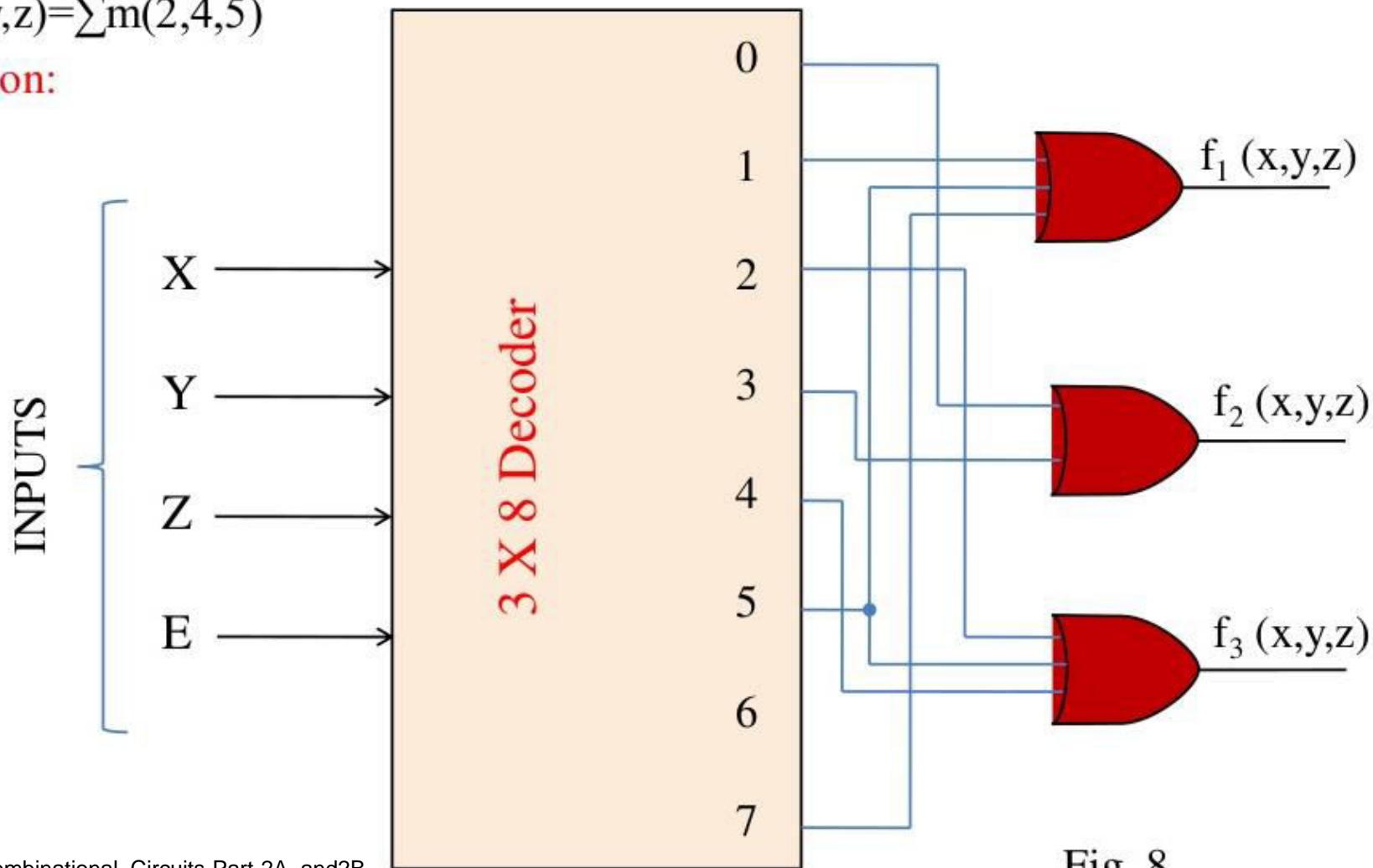


Fig. 8

**EXAMPLE:** A combinational circuit is defined by the following Boolean function. Design circuit with a Decoder and external gate.

$$F_1(x, y, z) = \overline{x} \overline{y} \overline{z} + x z$$

$$F_2(x, y, z) = x \overline{y} \overline{z} + \overline{x} z$$

**SOLUTION:** STEP 1: Write the given function  $F_1$  in SOP form

$$F_1(x, y, z) = \overline{x} \overline{y} \overline{z} + (y + \overline{y}) x z$$

$$F_1(x, y, z) = \overline{x} \overline{y} \overline{z} + x y z + x \overline{y} z$$

$$F_1(x, y, z) = \Sigma m(0, 5, 7)$$

$$F_2(x, y, z) = x \overline{y} \overline{z} + x z$$

$$F_2(x, y, z) = x \overline{y} \overline{z} + (y + \overline{y}) \overline{x} z$$

$$F_2(x, y, z) = x \overline{y} \overline{z} + \overline{x} y z + \overline{x} \overline{y} z$$

$$F_2(x, y, z) = \Sigma m(1, 3, 6)$$

## Boolean Function using Decoder:

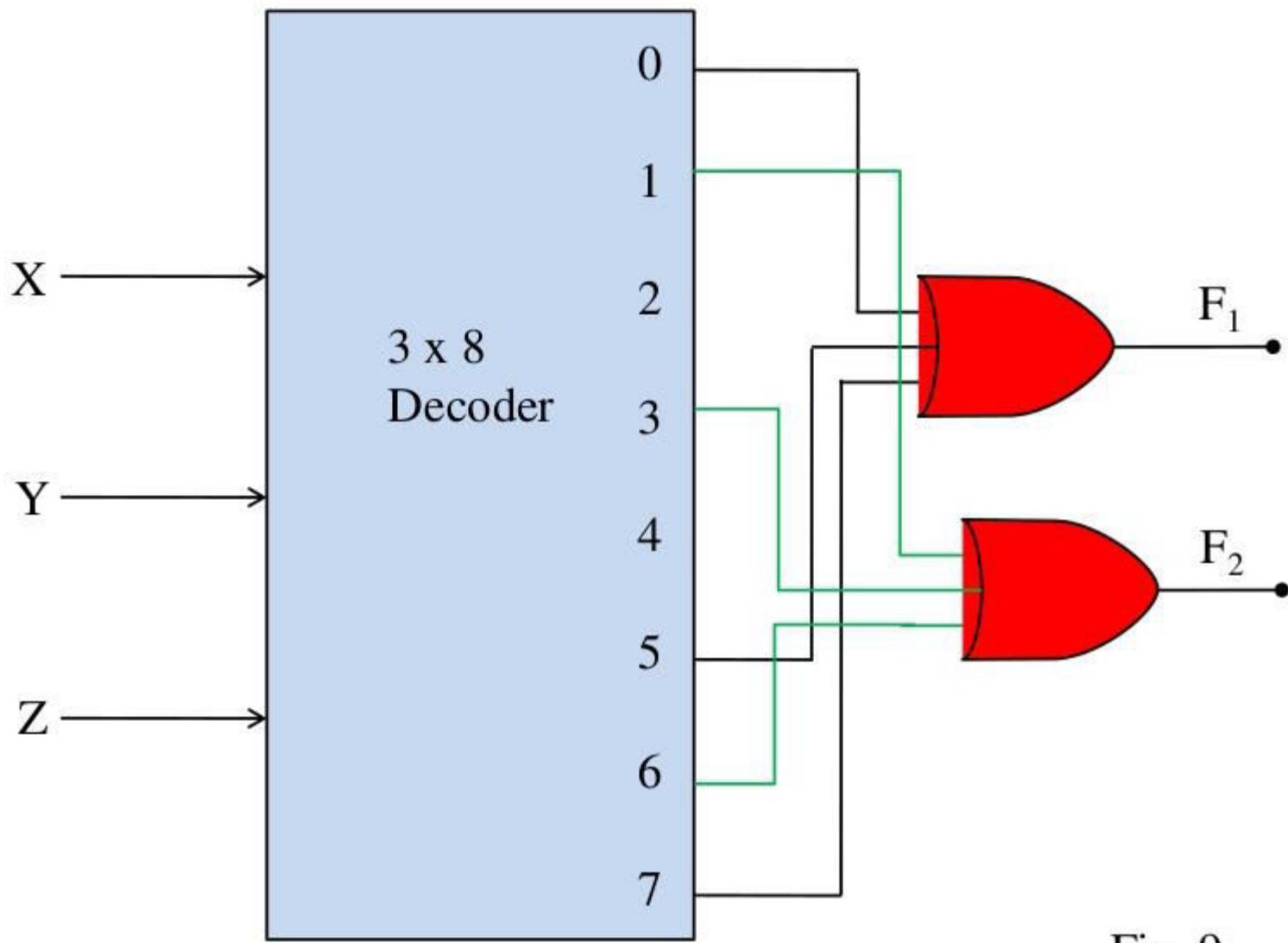
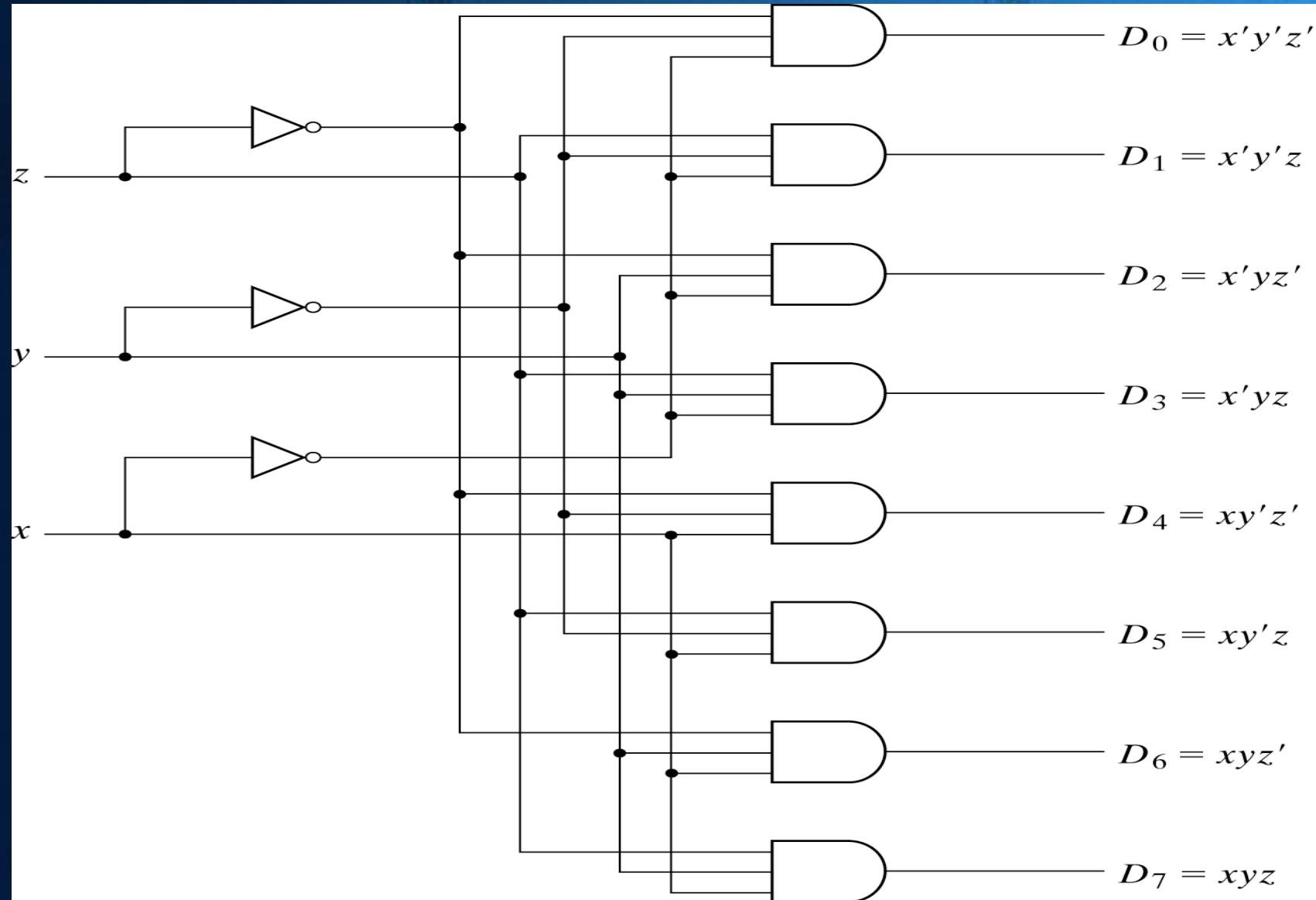


Fig. 9

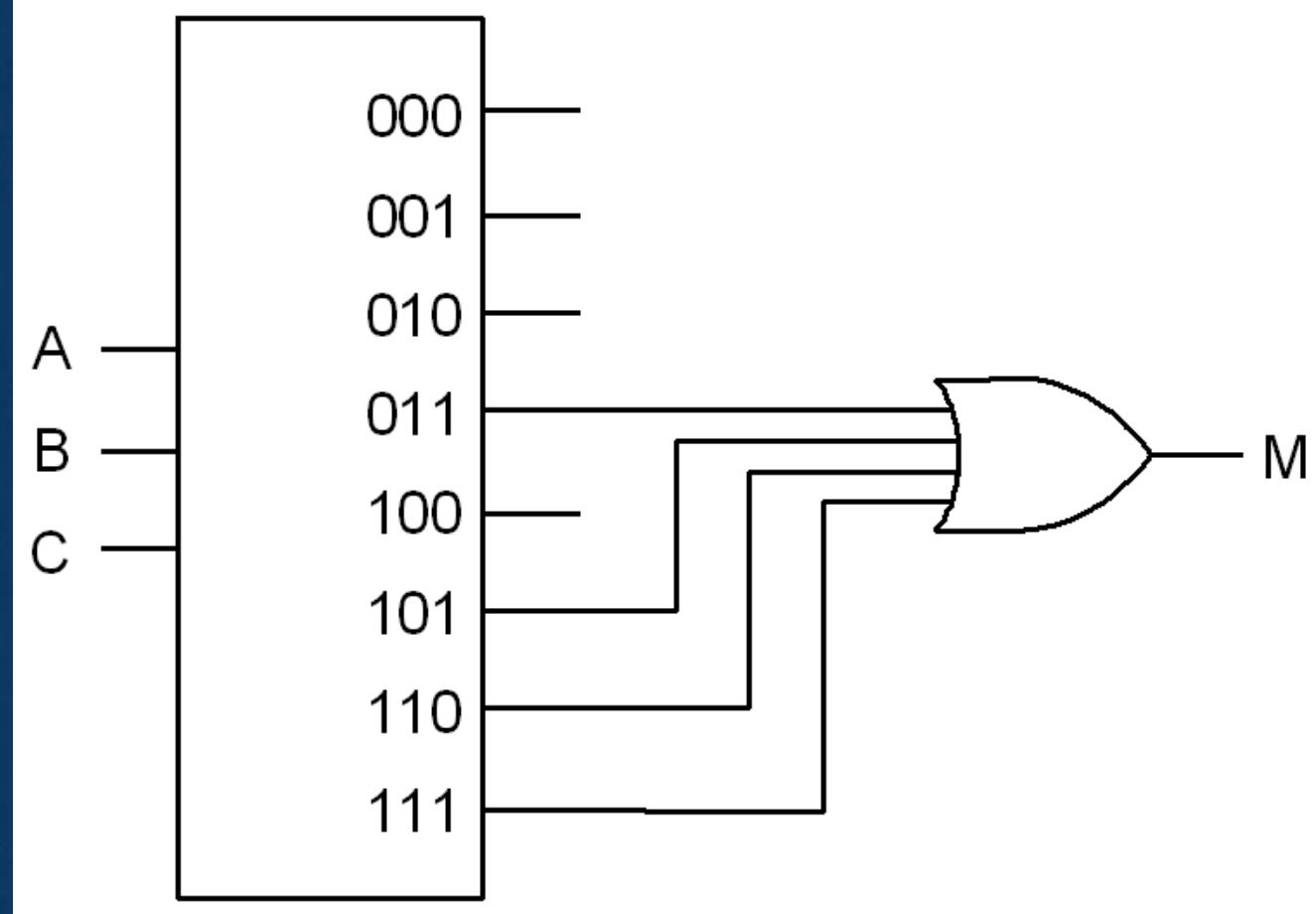
## 3x8 Decoder

From course text book



# Example

A	B	C	M
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



# Full Adder with Decoder

A <sub>i</sub>	B <sub>i</sub>	C <sub>i</sub>	S <sub>i</sub>	C <sub>i+1</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

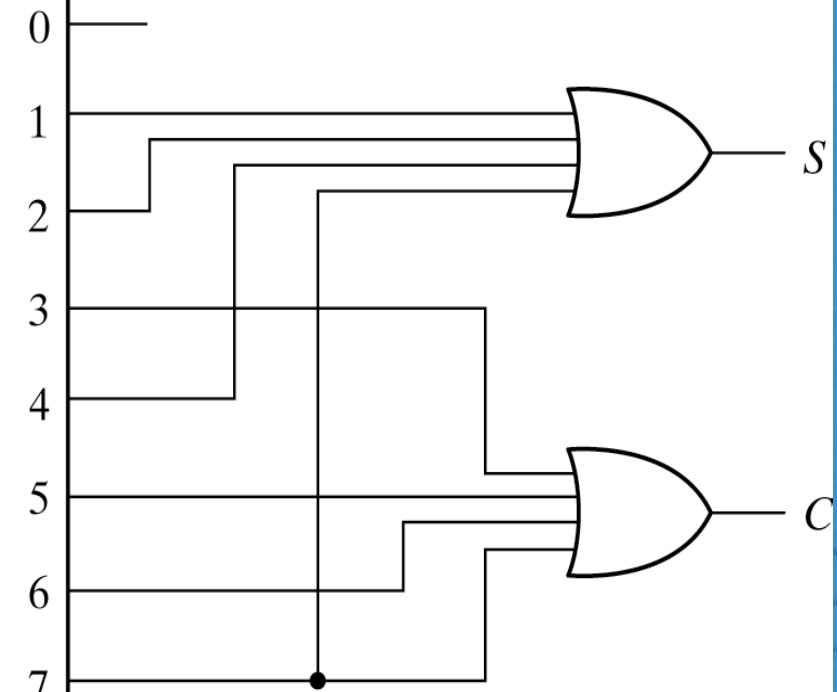
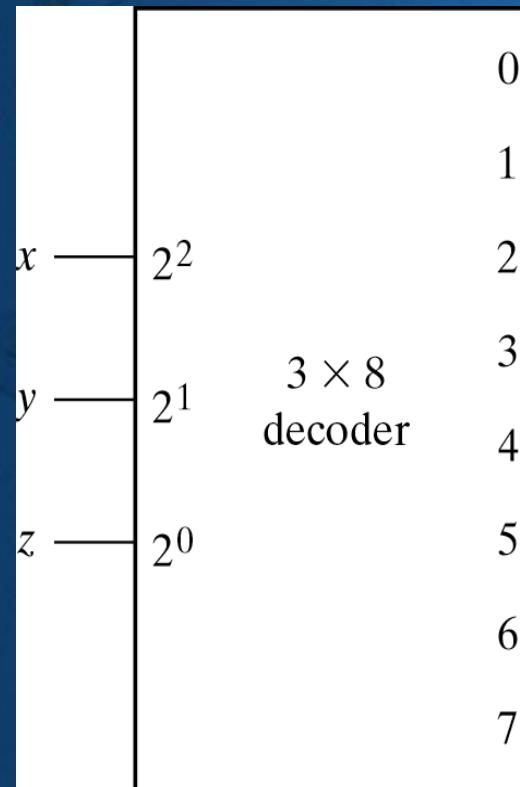


Fig. 4-21 Implementation of a Full Adder with a Decoder

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i' C_i + A_i' B_i C_i + A_i B_i$$

## ENCODER

- An Encoder is a combinational logic circuit.
- It performs the inverse operation of Decoder.
- The opposite process of decoding is known as Encoding.
- An Encoder converts an active input signal into a coded output signal.
- Block diagram of Encoder is shown in Fig.10. It has ‘M’ inputs and ‘N’ outputs.
- An Encoder has ‘M’ input lines, only one of which is activated at a given time, and produces an N-bit output code, depending on which input is activated.

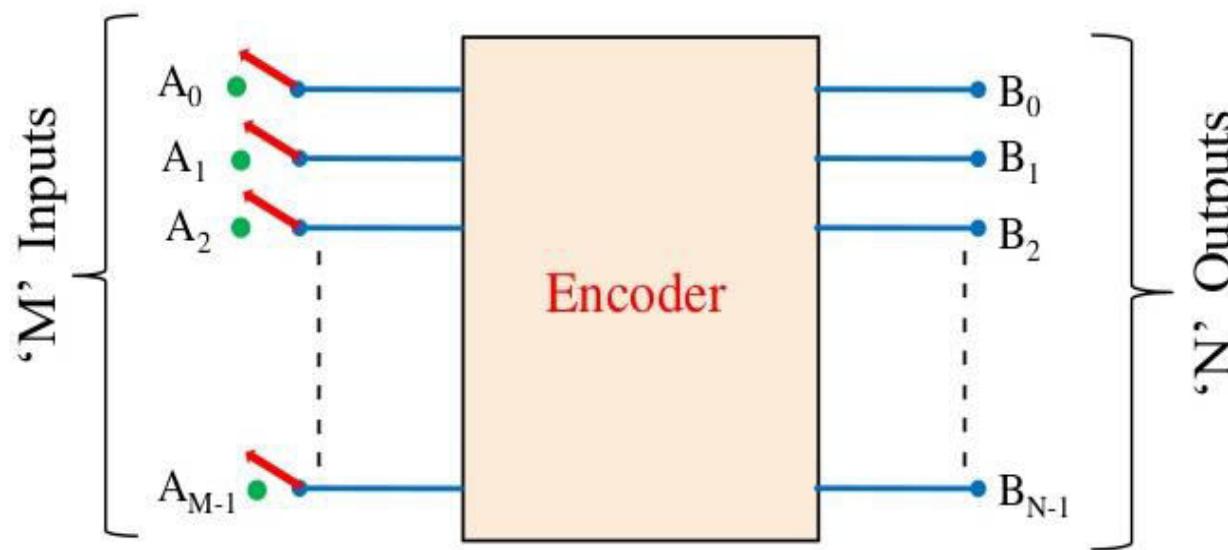


Fig. 10

- Encoders are used to translate the rotary or linear motion into a digital signal.
- The difference between Decoder and Encoder is that Decoder has Binary Code as an input while Encoder has Binary Code as an output.
- Encoder is an Electronics device that converts the analog signal to digital signal such as BCD Code.
- **Types of Encoders**
  - i. Priority Encoder
  - ii. Decimal to BCD Encoder
  - iii. Octal to Binary Encoder
  - iv. Hexadecimal to Binary Encoder

## ENCODER

$M=4$

$M=2^2$

$M=2^N$

'M' is the input and  
'N' is the output

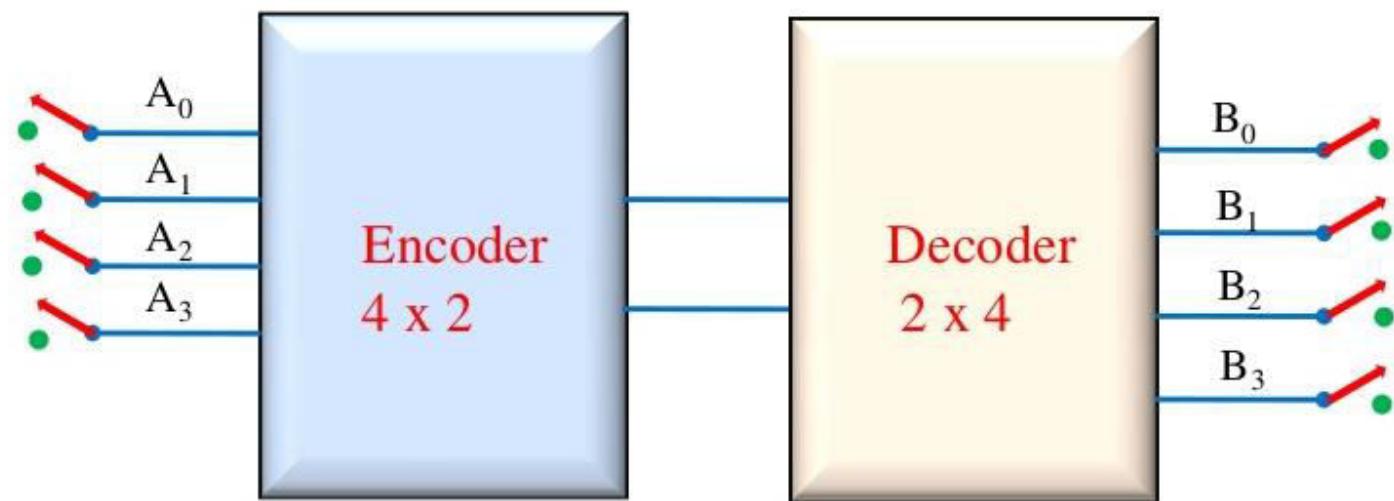


Fig. 11

## ENCODER

$M=4$

$M=2^2$

$M=2^N$

‘M’ is the input and  
‘N’ is the output

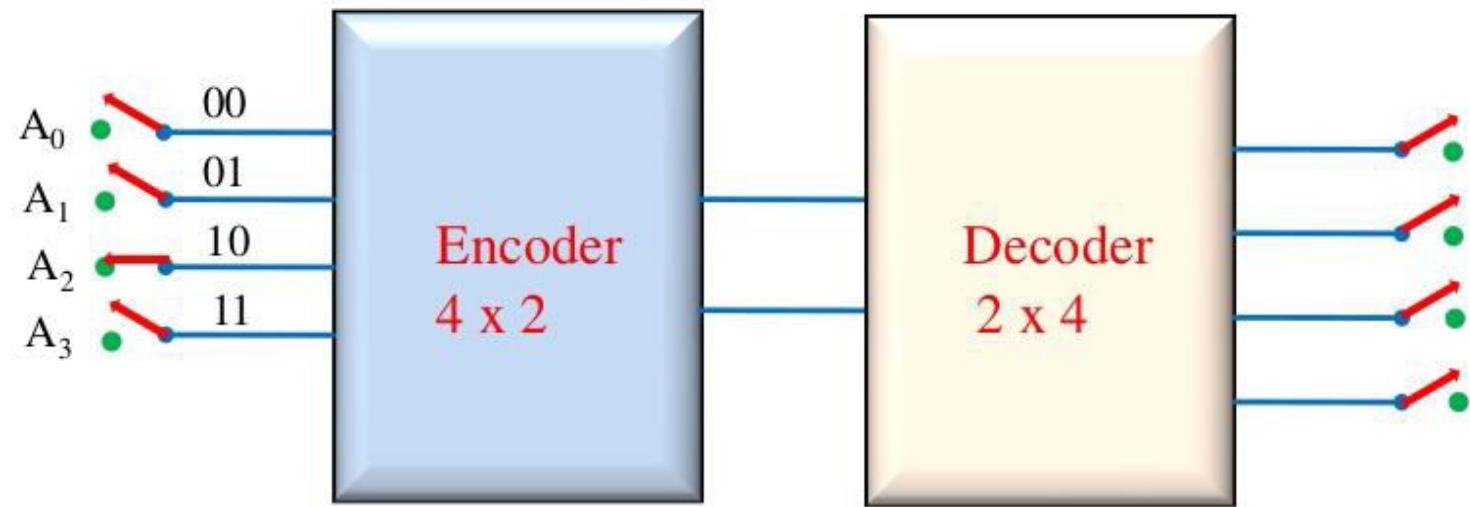


Fig. 12

## ENCODER

$M=4$

$M=2^2$

$M=2^N$

'M' is the input and  
'N' is the output

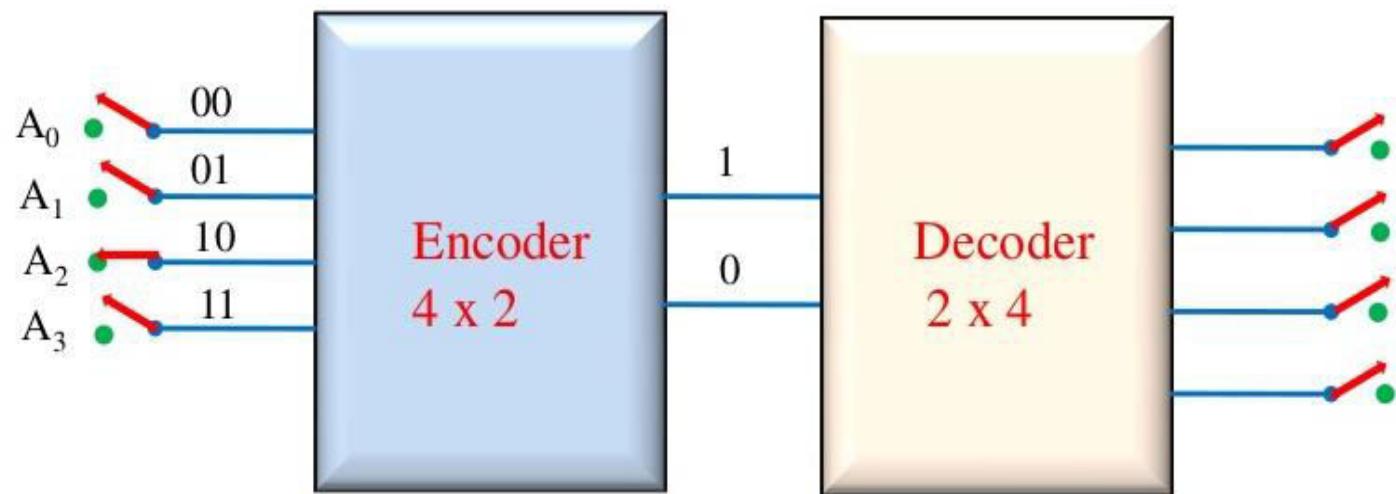


Fig. 13

# ENCODER

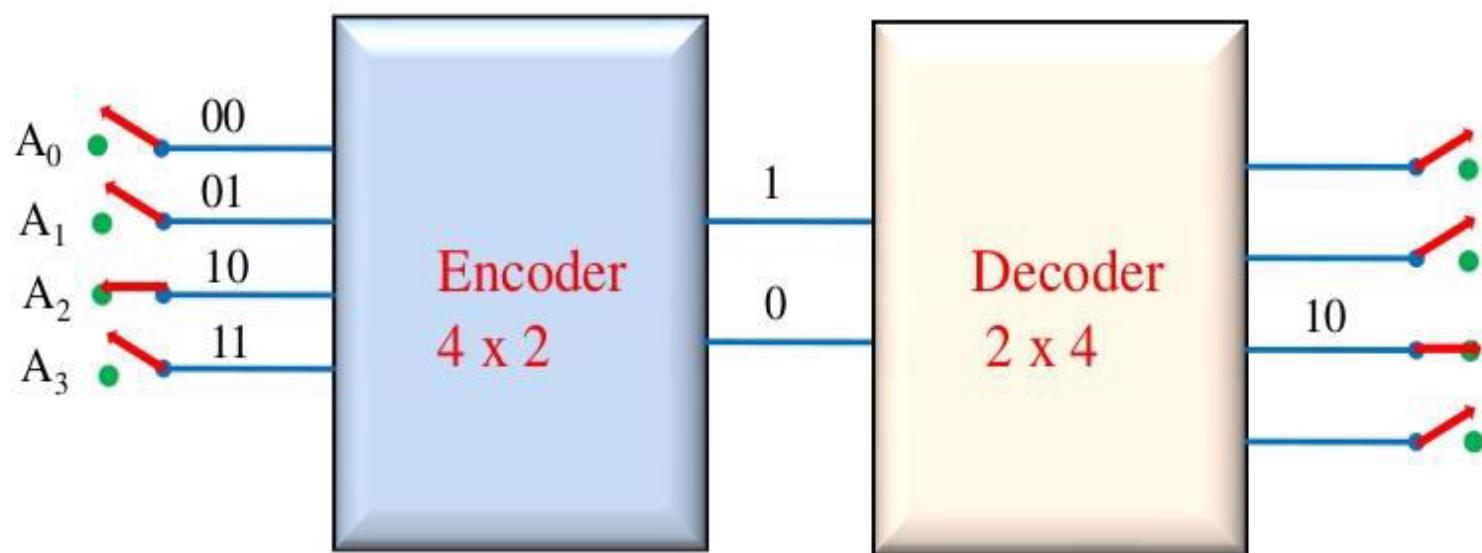
$M=4$

$M=2^2$

$M=2^N$

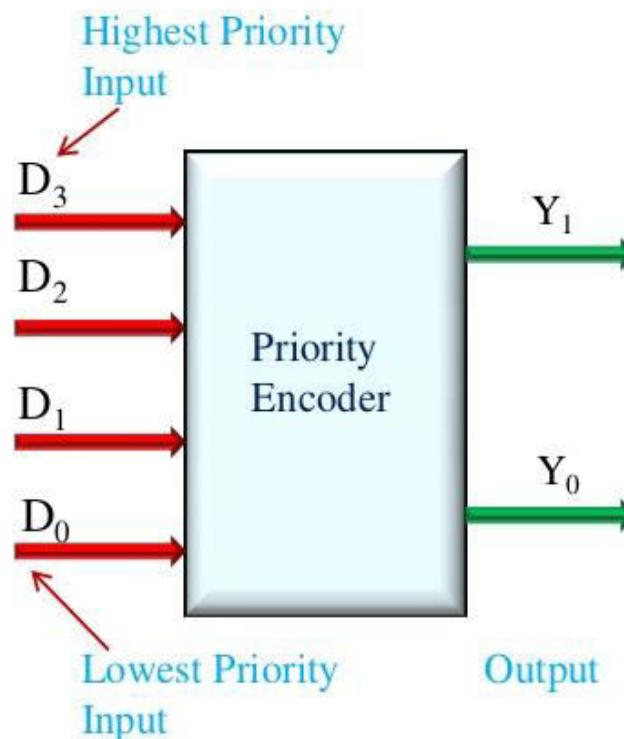
'M' is the input and

'N' is the output



## PRIORITY ENCODER:

- As the name indicates, the priority is given to inputs line.
- If two or more input lines are high at the same time i.e 1 at the same time, then the input line with high priority shall be considered.
- Block diagram and Truth table of Priority Encoder are shown in fig.15



*Block Diagram of Priority Encoder*

## TRUTH TABLE:

D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	INPUTS		OUTPUTS		V
						Y <sub>1</sub>	Y <sub>0</sub>	
0	0	0	0	x	x	0	0	
0	0	0	1	0	0	1	1	
0	0	1	x	0	1	1	1	
0	1	x	x	1	0	1	1	
1	x	x	x	1	1	1	1	

*Fig.15*

- There are four inputs  $D_0, D_1, D_2, D_3$  and two outputs  $Y_1$  and  $Y_0$ .
- $D_3$  has highest priority and  $D_0$  is at lowest priority.
- If  $D_3=1$  irrespective of other inputs then output  $Y_1 Y_0=11$ .
- $D_3$  is at highest priority so other inputs are considered as don't care.

**K-map for  $Y_1$  and  $Y_0$**

		$D_1 D_0$	00	01	11	10	
		$D_3 D_2$	00	X	0	0	0
		00	1	1	1	1	
		01	1	1	1	1	
		11	1	1	1	1	
		10	1	1	1	1	

$$Y_1 = D_2 + D_3$$

		$D_1 D_0$	00	01	11	10	
		$D_3 D_2$	00	X	0	1	1
		00	0	0	0	0	
		01	1	1	1	1	
		11	1	1	1	1	
		10	1	1	1	1	

$$Y_0 = D_3 + \overline{D}_2 D_1$$

**Fig. 16**

## LOGIC DIAGRAM OF PRIORITY ENCODER:

$$Y_1 = D_2 + D_3$$

$$Y_0 = D_3 + \overline{D}_2 D_1$$

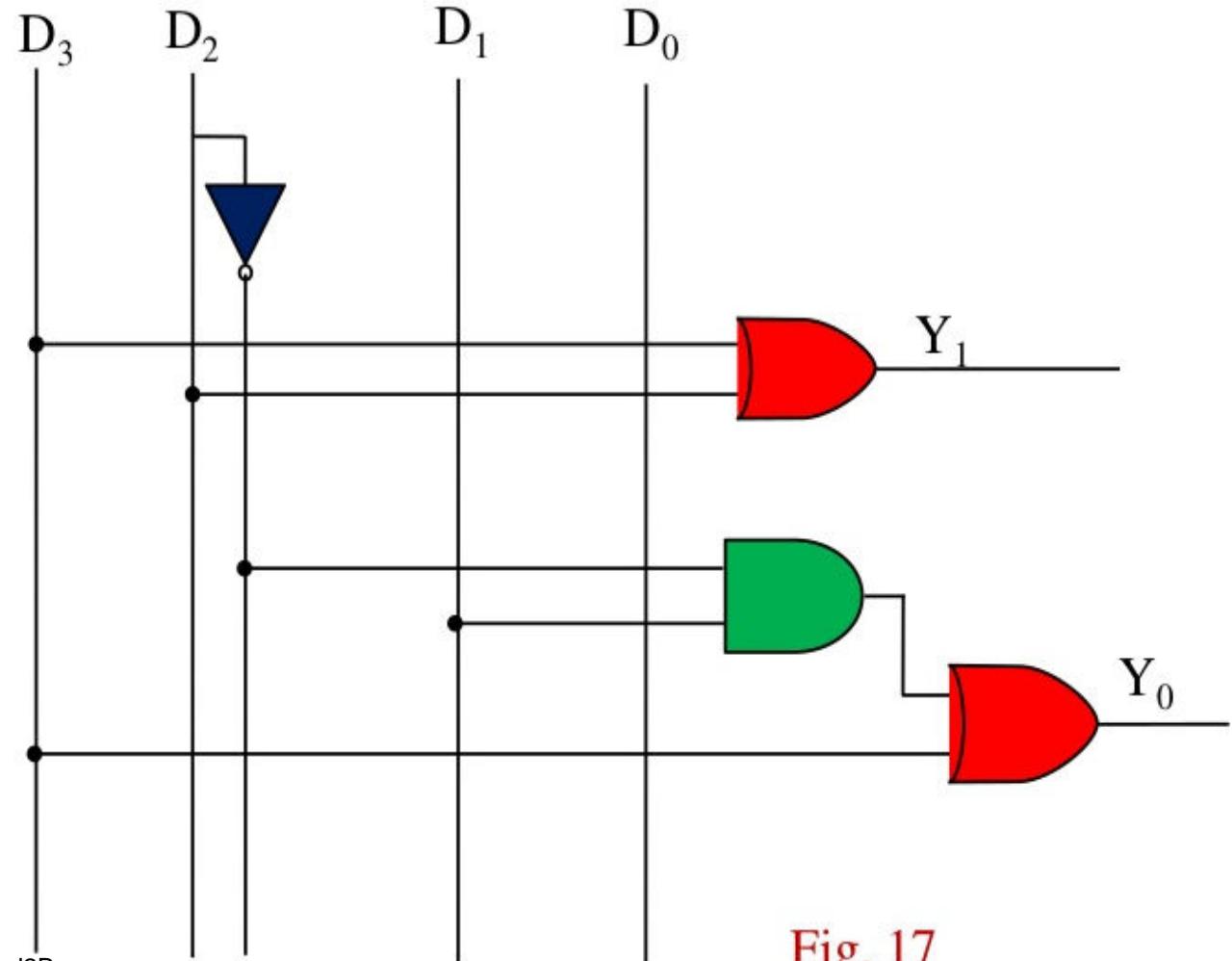


Fig. 17

## DECIMAL TO BCD ENCODER:

- It has ten inputs corresponding to ten decimal digits (from 0 to 9) and four outputs (A,B,C,D) representing the BCD.
- The block diagram is shown in fig.18 and Truth table in fig.19

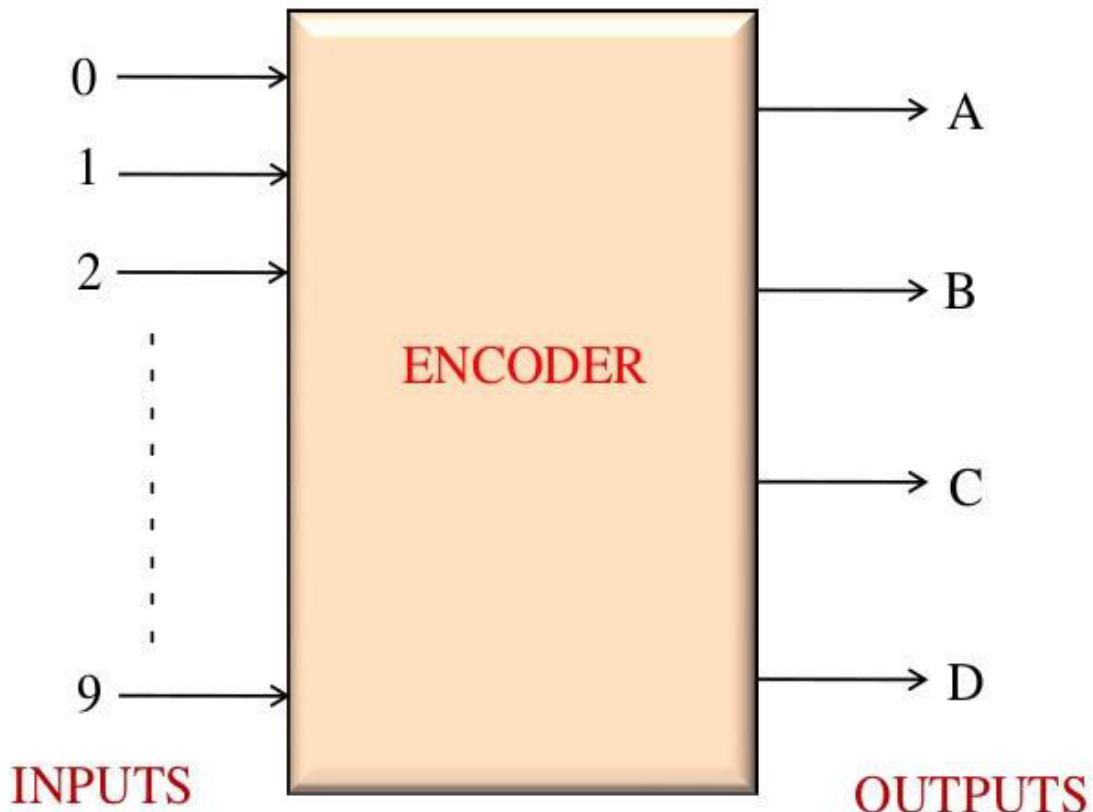


Fig. 18

## Truth table:

INPUTS										BCD OUTPUTS			
0	1	2	3	4	5	6	7	8	9	A	B	C	D
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

Fig. 10

- From Truth Table it is clear that the output A is HIGH when input is 8 OR 9 is HIGH

Therefore  $A=8+9$

- The output B is HIGH when 4 OR 5 OR 6 OR 7 is HIGH

Therefore  $B=4+5+6+7$

- The output C is HIGH when 2 OR 3 OR 6 OR 7 is HIGH

Therefore  $C=2+3+6+7$

- Similarly  $D=1+3+5+7+9$

Logic Diagram is shown in fig.20

## DECIMAL TO BCD ENCODER

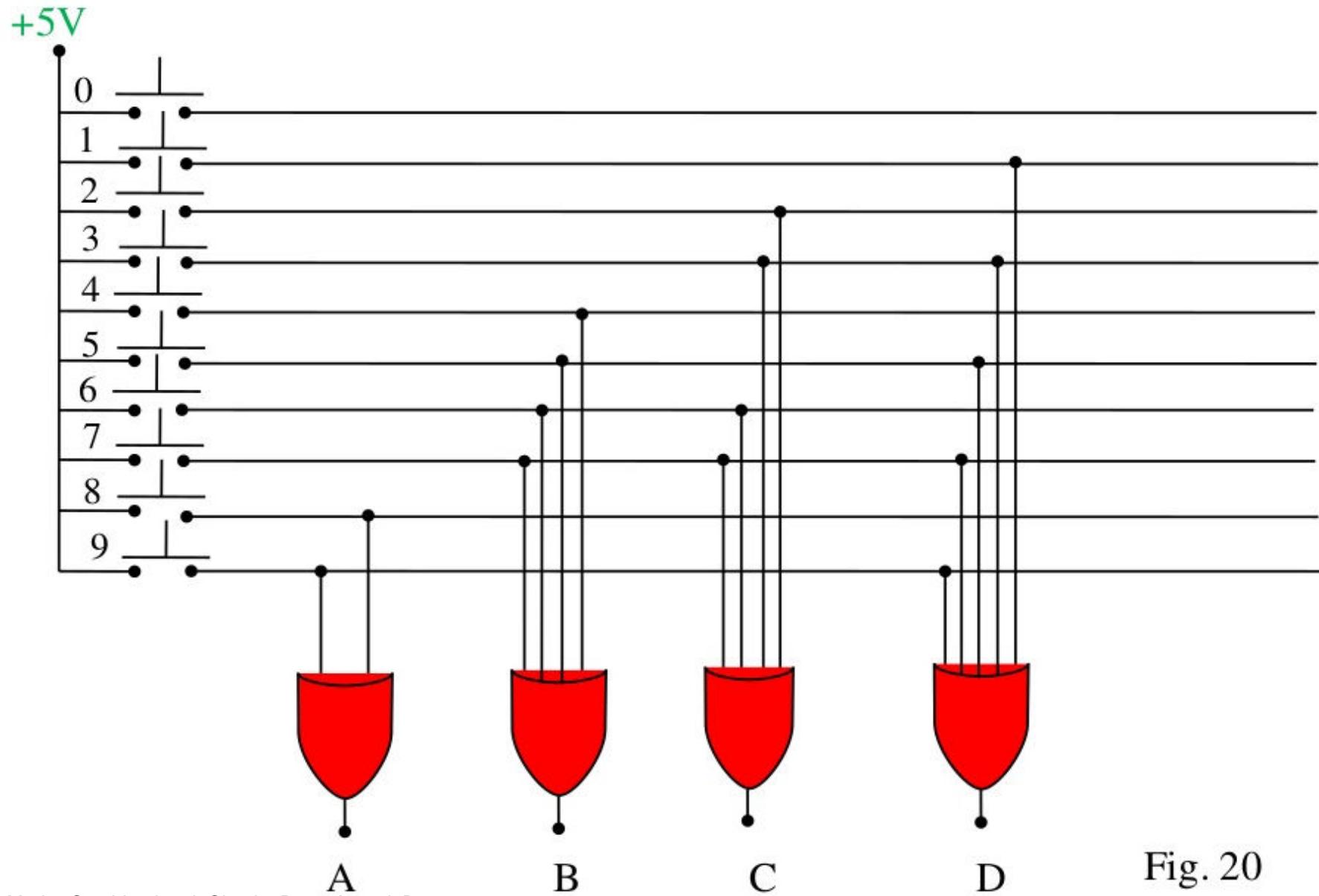


Fig. 20

## OCTAL TO BINARY ENCODER:

- Block Diagram of Octal to Binary Encoder is shown in Fig. 21
- It has eight inputs and three outputs.
- Only one input has one value at any given time.
- Each input corresponds to each octal digit and output generates corresponding Binary Code.

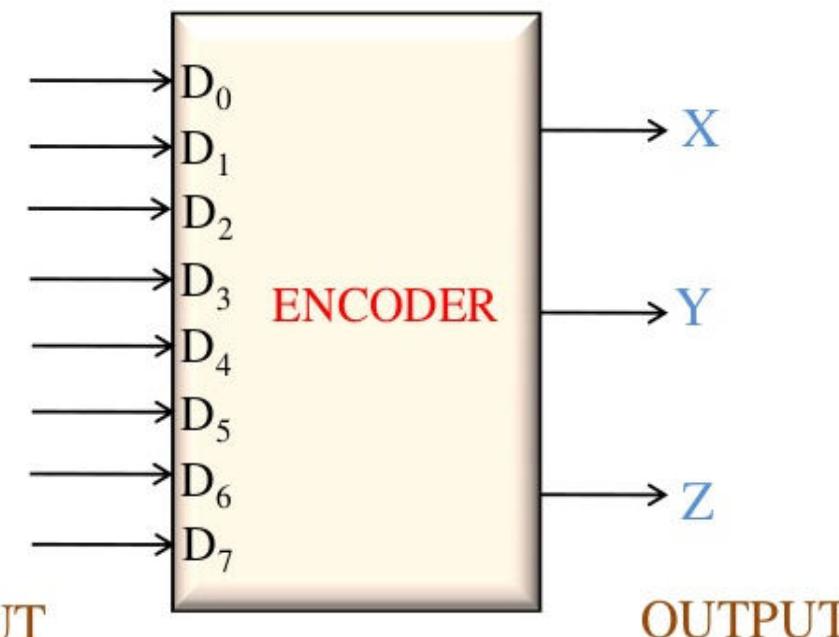


Fig. 21

## TRUTH TABLE:

INPUT								OUTPUT		
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	X	Y	Z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Fig. 22

## From Truth table:

$$X = D_4 + D_5 + D_6 + D_7$$

$$Y = D_2 + D_3 + D_6 + D_7$$

$$Z = D_1 + D_3 + D_5 + D_7$$

- It is assumed that only one input is HIGH at any given time. If two outputs are HIGH then undefined output will be produced. For example  $D_3$  and  $D_6$  are HIGH, then output of Encoder will be 111. This output is neither equivalent code corresponding to  $D_3$  nor to  $D_6$ .
- To overcome this problem, priorities should be assigned to each input.
- From the truth table it is clear that the output X becomes 1 if any of the digit  $D_4$  or  $D_5$  or  $D_6$  or  $D_7$  is 1.
- $D_0$  is considered as don't care because it is not shown in expression.
- If inputs are zero then output will be zero. Similarly if  $D_0$  is one, the output will be zero.
-

$$X = D_4 + D_5 + D_6 + D_7$$

$$Y = D_2 + D_3 + D_6 + D_7$$

$$Z = D_1 + D_3 + D_5 + D_7$$

### LOGIC DIAGRAM:

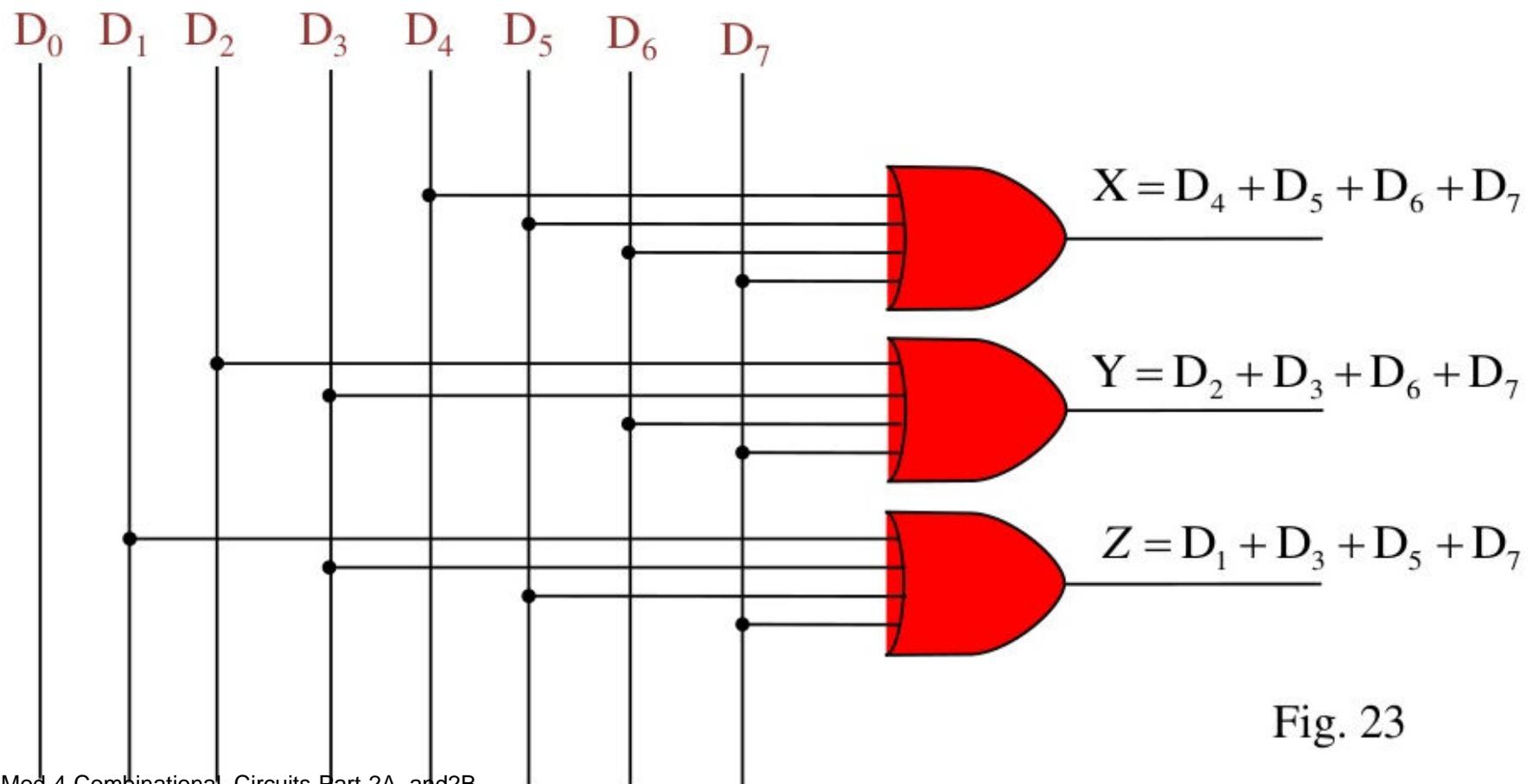
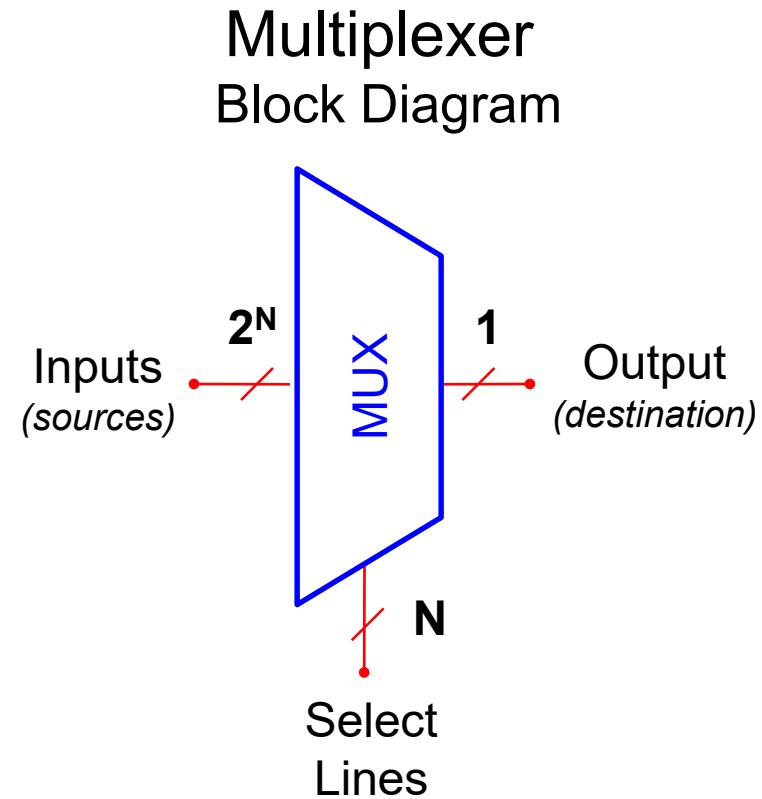


Fig. 23

# What is a Multiplexer (MUX)?

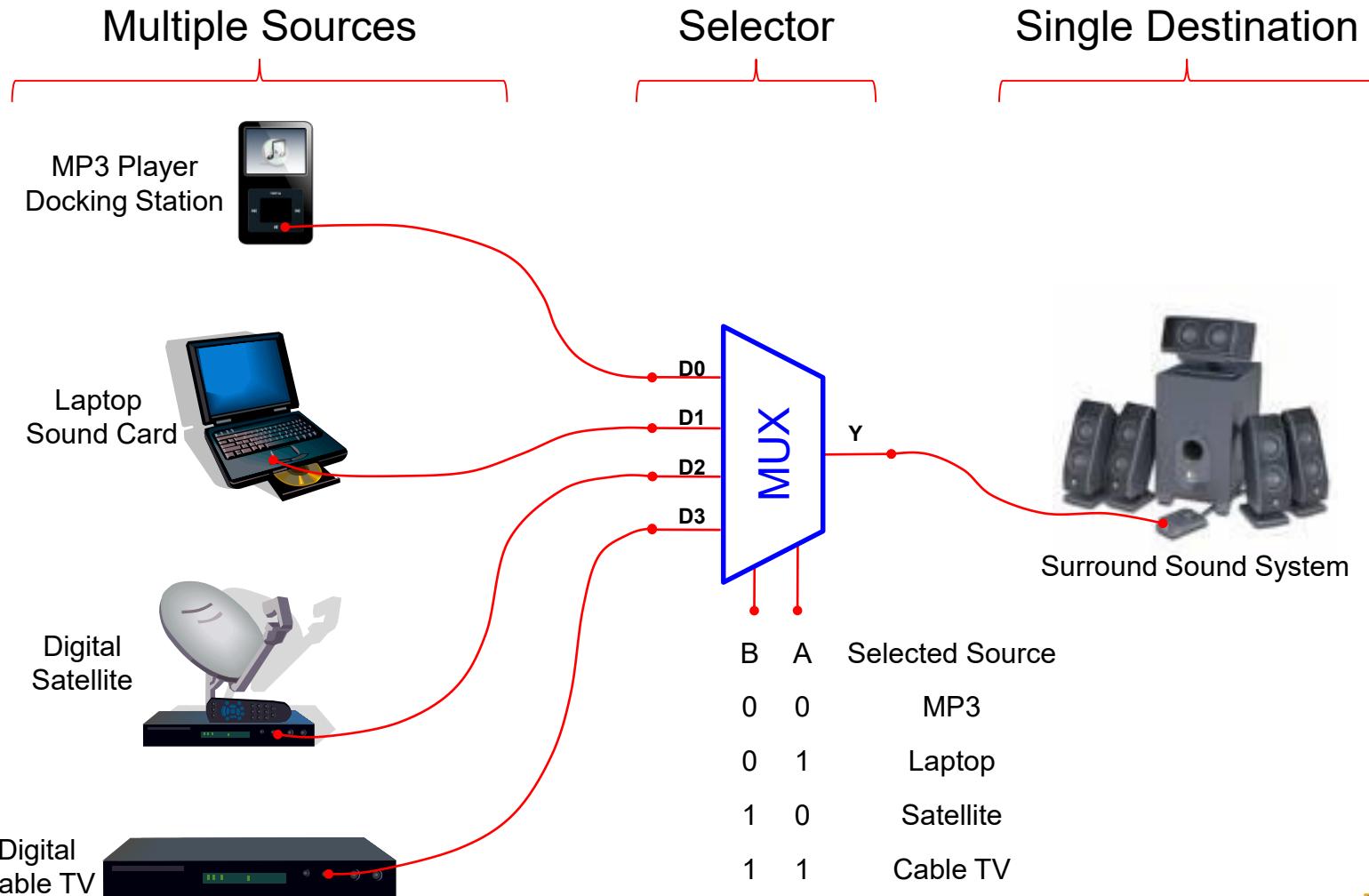
- A MUX is a digital switch that has multiple inputs (sources) and a single output (destination).
- The select lines determine which input is connected to the output.
- MUX Types
  - 2-to-1 (1 select line)
  - 4-to-1 (2 select lines)
  - 8-to-1 (3 select lines)
  - 16-to-1 (4 select lines)



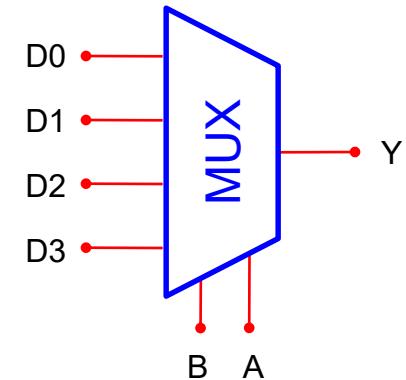
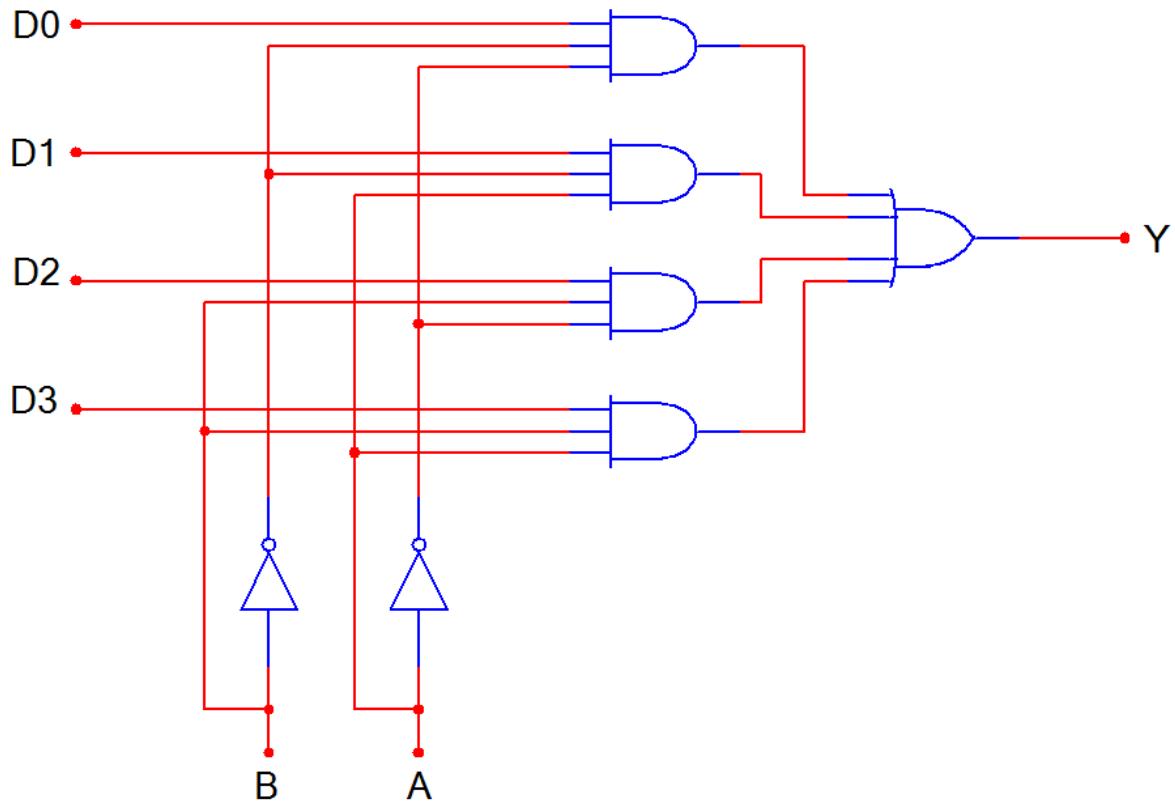
# Multiplexer(contd..)

- Also called data selectors.
- Basic function: select one of its  $2^n$  data input lines and place the corresponding information onto a single output line.
- $n$  input bits needed to specify which input line is to be selected.
  - Place binary code for a desired data input line onto its  $n$  select input lines.

# Typical Application of a MUX

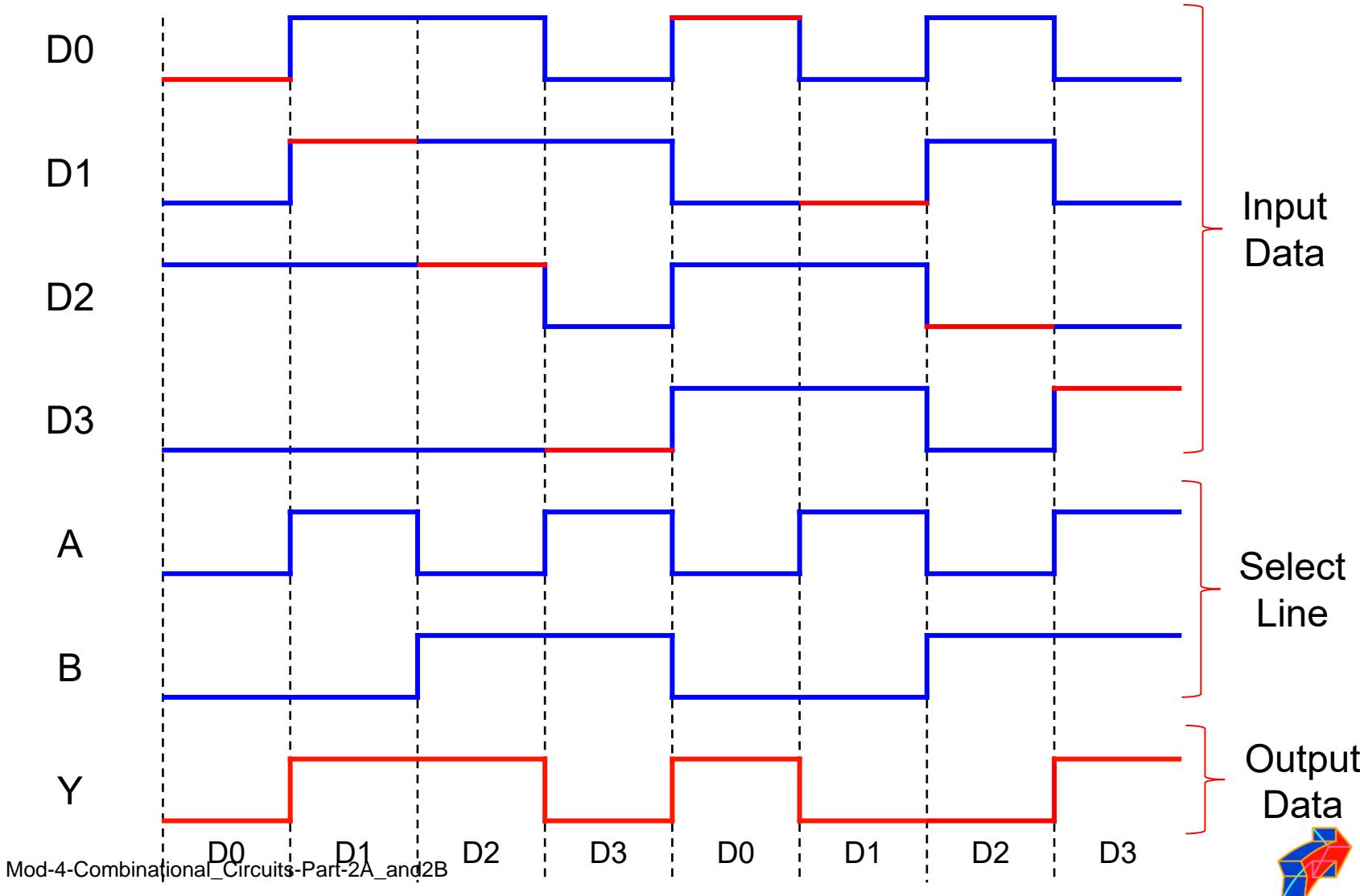


# 4-to-1 Multiplexer (MUX)



B	A	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

# 4-to-1 Multiplexer Waveforms



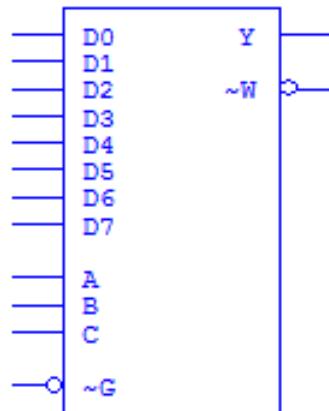
# Medium Scale Integration MUX

4-to-1 MUX

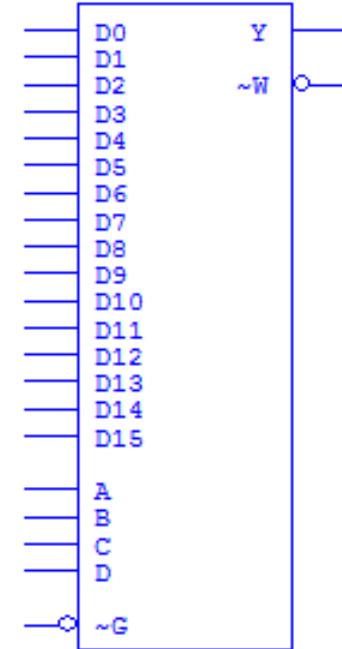
Inputs {  
Select {  
Enable ↗

} Output (Y)  
(and inverted output)

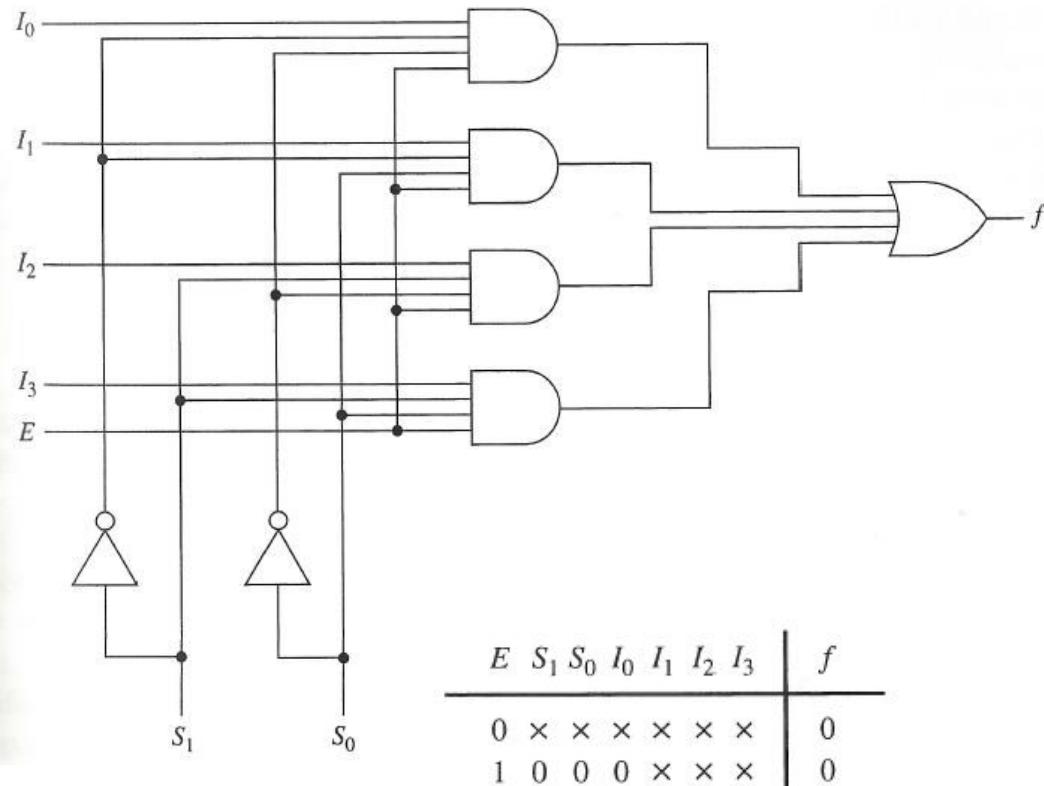
8-to-1 MUX



16-to-1 MUX



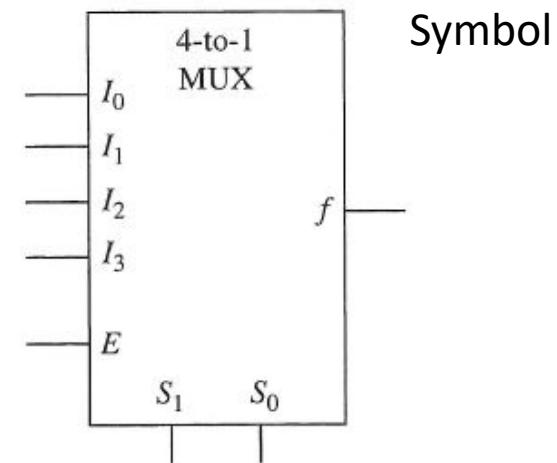
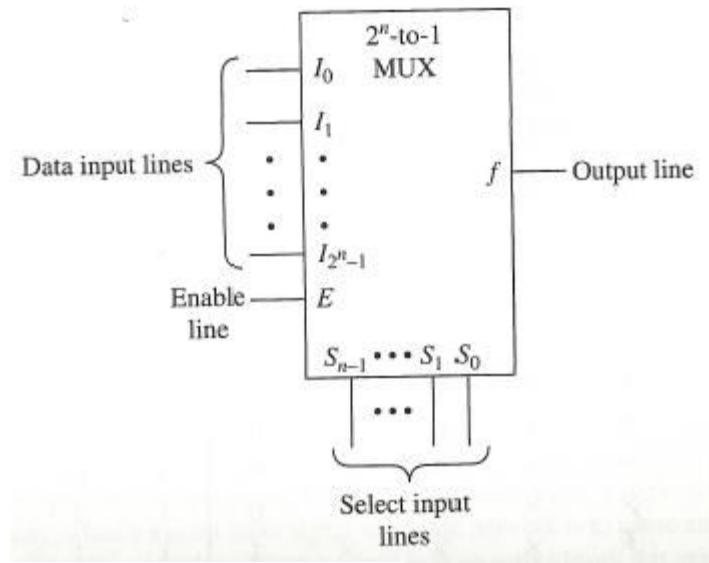
# Realization of 4-to-1 line multiplexer



Logic Diagram

E	$S_1$	$S_0$	$I_0$	$I_1$	$I_2$	$I_3$	f
0	x	x	x	x	x	x	0
1	0	0	0	x	x	x	0
1	0	0	1	x	x	x	1
1	0	1	x	0	x	x	0
1	0	1	x	1	x	x	1
1	1	0	x	x	0	x	0
1	1	0	x	x	1	x	1
1	1	1	x	x	x	0	0
1	1	1	x	x	x	1	1

Truth Table



Symbol

# Realization of 4-to-1 line multiplexer

- Alternate description:

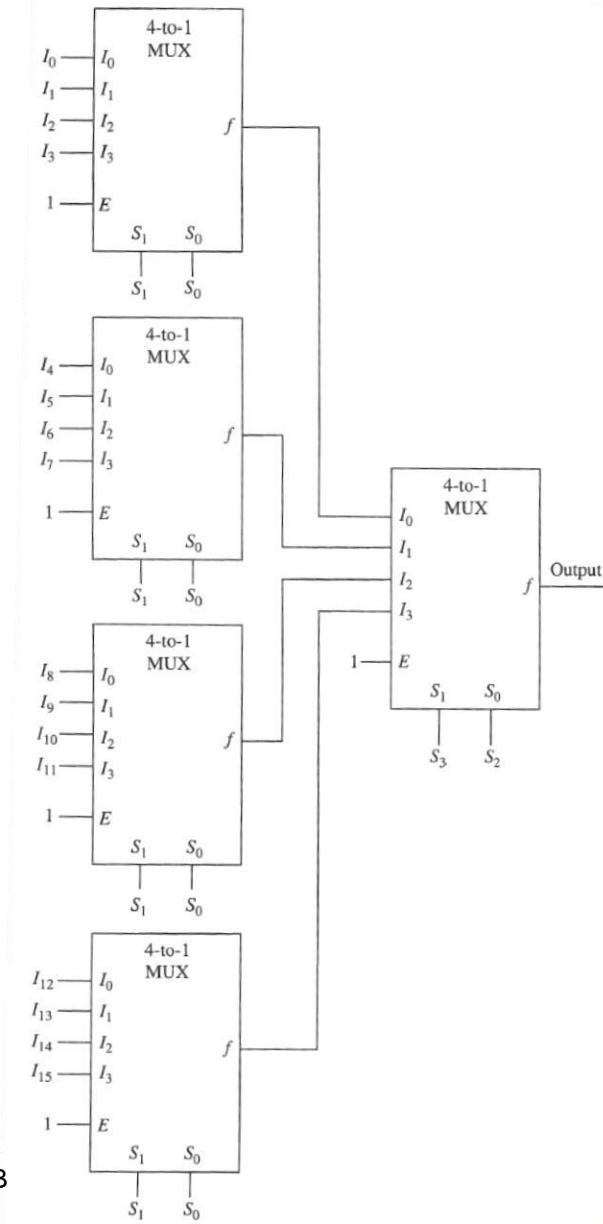
**Table 5.6** Function table for a 4-to-1-line multiplexer

<i>E</i>	<i>S<sub>1</sub></i>	<i>S<sub>0</sub></i>	<i>f</i>
0	×	×	0
1	0	0	<i>I<sub>0</sub></i>
1	0	1	<i>I<sub>1</sub></i>
1	1	0	<i>I<sub>2</sub></i>
1	1	1	<i>I<sub>3</sub></i>

- Algebraic description of multiplexer:

$$f = (I_0 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_1 S_0 + I_2 S_1 \bar{S}_0 + I_3 S_1 S_2)E$$

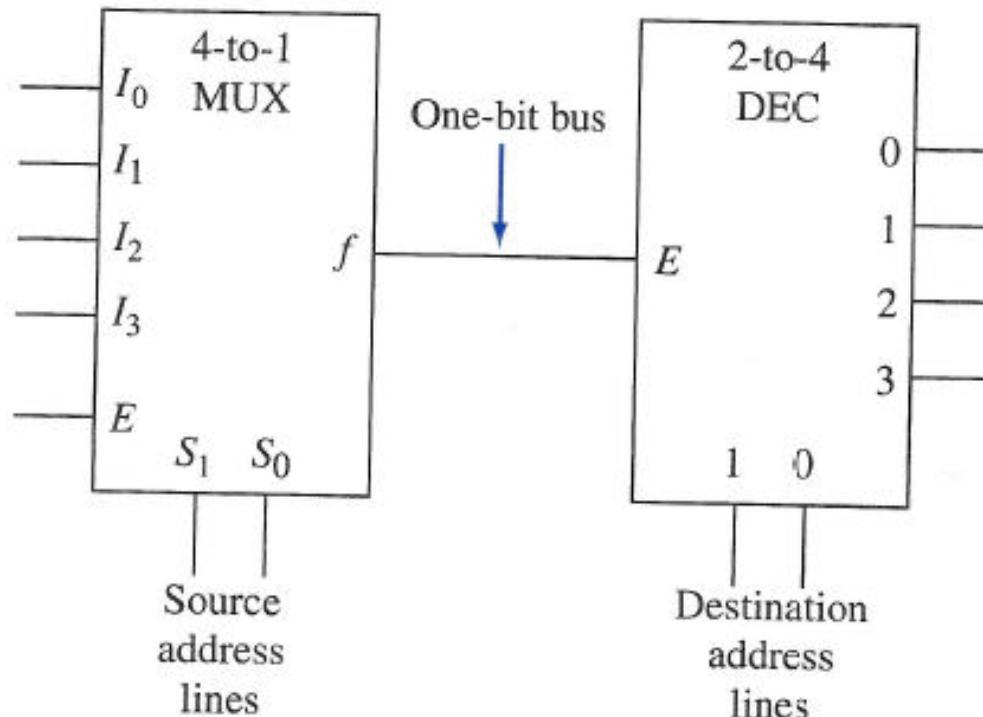
# Building a Large Multiplexer



# Multiplexers

- One of the primary applications of multiplexers is to provide for the transmission of information from several sources over a single path.
- This process is known as multiplexing.
- Demultiplexer = decoder with an enable input.

# Multiplexer/Demultiplexer for information transmission



**Figure 5.35** A multiplexer/demultiplexer arrangement for information transmission.

# Logic Design with Multiplexers

$x$	$y$	$z$	$f$
0	0	0	$f_0$
0	0	1	$f_1$
0	1	0	$f_2$
0	1	1	$f_3$
1	0	0	$f_4$
1	0	1	$f_5$
1	1	0	$f_6$
1	1	1	$f_7$

The Boolean expression corresponding to this truth table can be written as:

$$\begin{aligned}f(x, y, z) \\= f_0 \cdot \bar{x} \bar{y} \bar{z} + f_1 \cdot \bar{x} \bar{y} z + f_2 \cdot \bar{x} y \bar{z} + f_3 \cdot \bar{x} y z + f_4 \cdot x \bar{y} \bar{z} + f_5 \cdot x \bar{y} z \\+ f_6 x y \bar{z} + f_7 \cdot x y z.\end{aligned}$$

# Logic Design with Multiplexers

- The Boolean expression corresponding to this truth table can be written as:

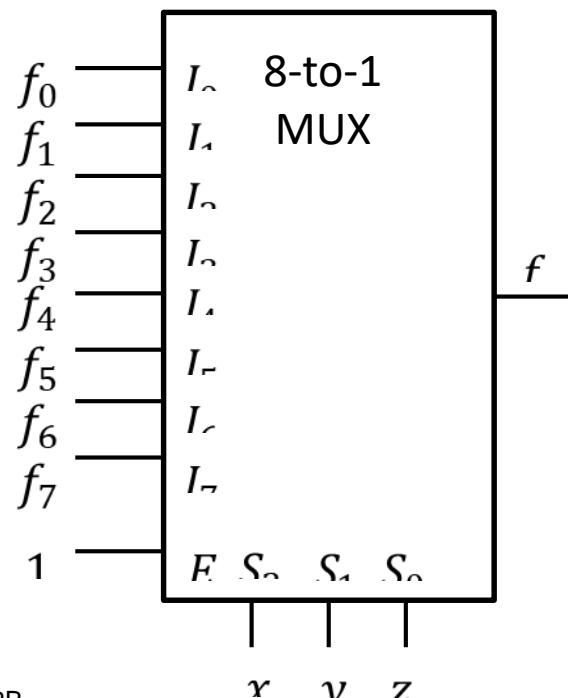
$$f(x, y, z) = f_0 \cdot \bar{x} \bar{y} \bar{z} + f_1 \cdot \bar{x} \bar{y} z + f_2 \cdot \bar{x} y \bar{z} + f_3 \cdot \bar{x} y z + f_4 \cdot x \bar{y} \bar{z} + f_5 \cdot x \bar{y} z + f_6 \cdot x y \bar{z} + f_7 \cdot x y z.$$

- The Boolean expression for an 8-to-1-line multiplexer is:

$$\begin{aligned} f &= (I_0 \bar{S}_2 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_2 \bar{S}_1 S_0 + I_2 \bar{S}_2 S_1 \bar{S}_0 + I_3 \bar{S}_2 S_1 S_0 \\ &\quad + I_4 S_2 \bar{S}_1 \bar{S}_0 + I_5 S_2 \bar{S}_1 S_0 + I_6 S_2 S_1 \bar{S}_0 + I_7 S_2 S_1 S_0). \end{aligned}$$

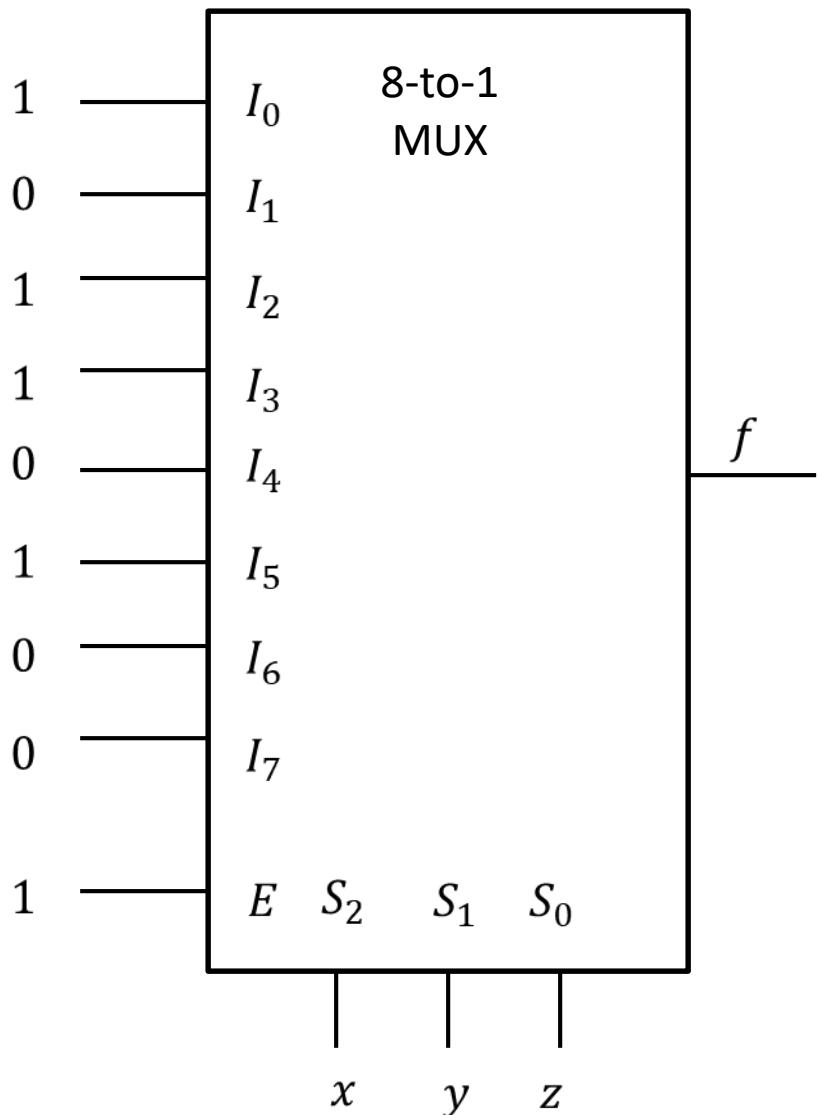
# Logic Design with Multiplexers

- If E is logic-1 then the latter is transformed into the former by replacing  $I_i$  with  $f_i$ ,  $S_2$  with  $x$ ,  $S_1$  with  $y$ , and  $S_0$  with  $z$ .
- Placing  $x, y, z$  on the select lines  $S_2, S_1, S_0$ , respectively and placing the functional values  $f_i$  on data input lines  $I_i$ .



# Example:

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



# Logic Design with Multiplexers

- If at least one input variable of a Boolean function is available in both its complemented and uncomplemented form, any  $n$ -variable function is realizable with a  $2^{n-1}$ -to-1-line multiplexer.
- For the case of a 3-variable function, only a 4-to-1 multiplexer is needed.
- $$\begin{aligned} f(x, y, z) &= f_0 \cdot \bar{x} \bar{y} \bar{z} + f_1 \cdot \bar{x} \bar{y} z + f_2 \cdot \bar{x} y \bar{z} + f_3 \cdot \bar{x} y z + \\ &\quad f_4 \cdot x \bar{y} \bar{z} + f_5 \cdot x \bar{y} z + f_6 x y \bar{z} + f_7 \cdot x y z \\ &= (f_0 \cdot \bar{z} + f_1 \cdot z) \bar{x} \bar{y} + (f_2 \cdot \bar{z} + f_3 \cdot z) \bar{x} y \\ &\quad + (f_4 \cdot \bar{z} + f_5 \cdot z) x \bar{y} + (f_6 \cdot \bar{z} + f_7 \cdot z) x y \end{aligned}$$
- When  $E = 1$ , 4-to-1 Multiplexer has the form

$$I_0 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_1 S_0 + I_2 S_1 \bar{S}_0 + I_3 S_1 S_2$$

# Logic Design with Multiplexers

$$\begin{aligned}f(x, y, z) &= (f_0 \cdot \bar{z} + f_1 \cdot z)\bar{x}\bar{y} + (f_2 \cdot \bar{z} + f_3 \cdot z)\bar{x}y \\&\quad + (f_4 \cdot \bar{z} + f_5 \cdot z)x\bar{y} + (f_6 \cdot \bar{z} + f_7 \cdot z)xy\end{aligned}$$

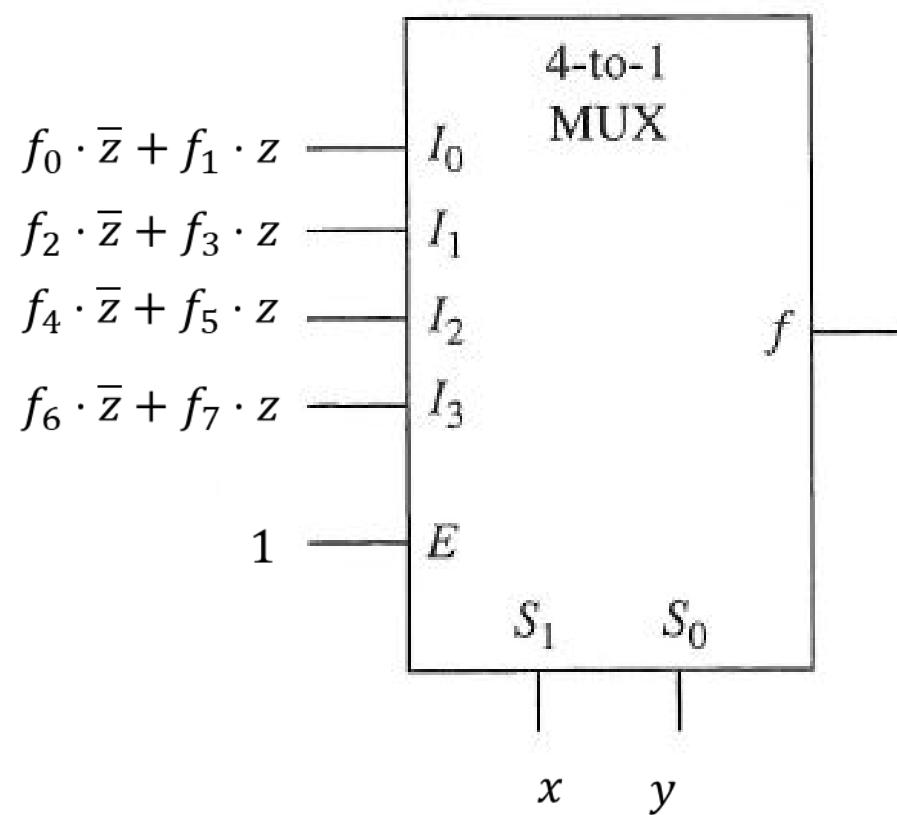
4-to-1 Multiplexer has the form

$$f = I_0 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_1 S_0 + I_2 S_1 \bar{S}_0 + I_3 S_1 S_2$$

- Realization of  $f(x, y, z)$  is obtained by placing the  $x$  and  $y$  variables on the  $S_1, S_0$  select lines, the single variable functions  $f_i \cdot \bar{z} + f_j \cdot z$  on the data input lines and let  $E = 1$ .
- Note:  $f_i \cdot \bar{z} + f_j \cdot z$  reduce to 0, 1,  $z$  or  $\bar{z}$ .

# Example

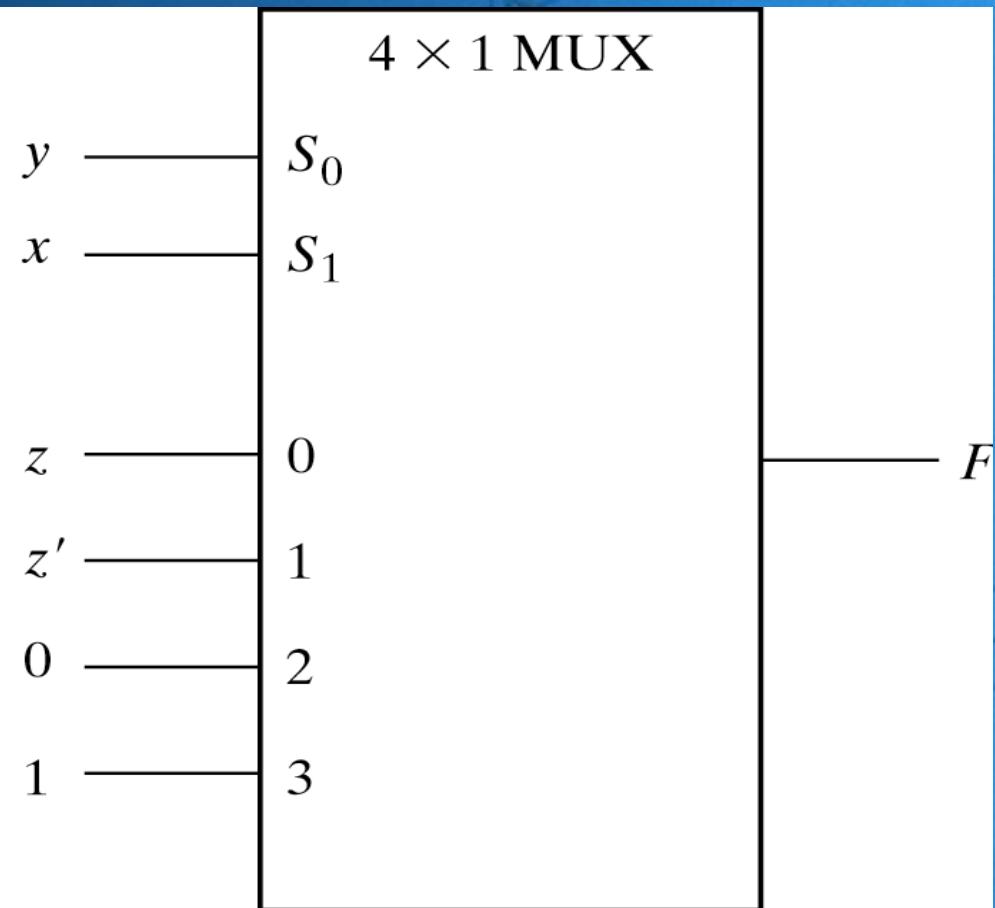
$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



# Boolean Function Implementation

$x$	$y$	$z$	$F$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a) Truth table



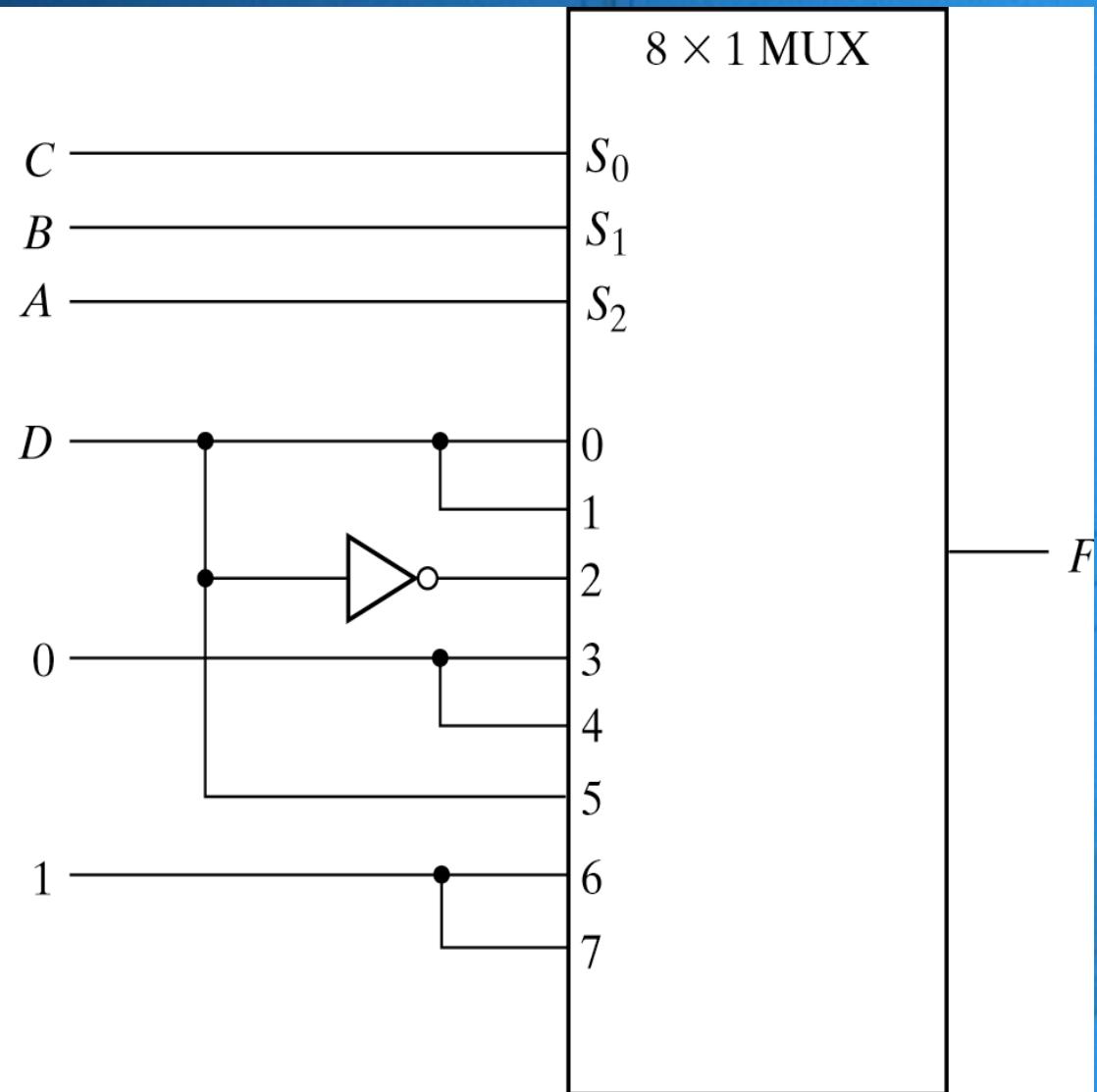
(b) Multiplexer implementation

Fig. 4-27 Implementing a Boolean Function with a Multiplexer

# Example

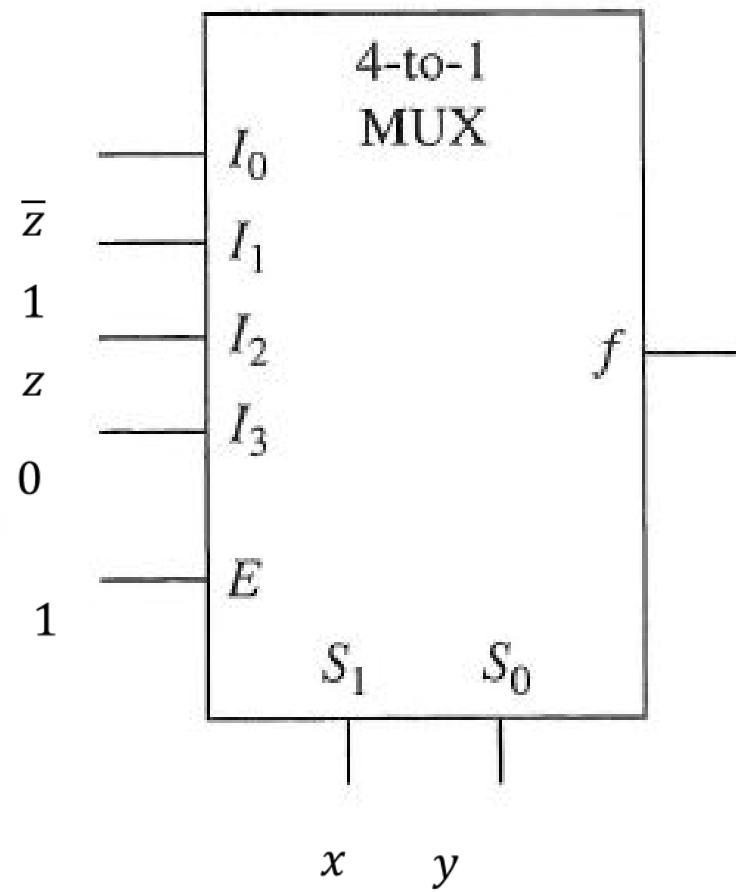
How do you implement it with 8x1 MUX?

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1 $F = D$
0	0	1	0	0
0	0	1	1	1 $F = D$
0	1	0	0	1
0	1	0	1	0 $F = D'$
0	1	1	0	0
0	1	1	1	0 $F = 0$
1	0	0	0	0
1	0	0	1	0 $F = 0$
1	0	1	0	0
1	0	1	1	1 $F = D$
1	1	0	0	1
1	1	0	1	1 $F = 1$
1	1	1	0	1
1	1	1	1	1 $F = 1$



# Example How do you implement it with 4x1 MUX?

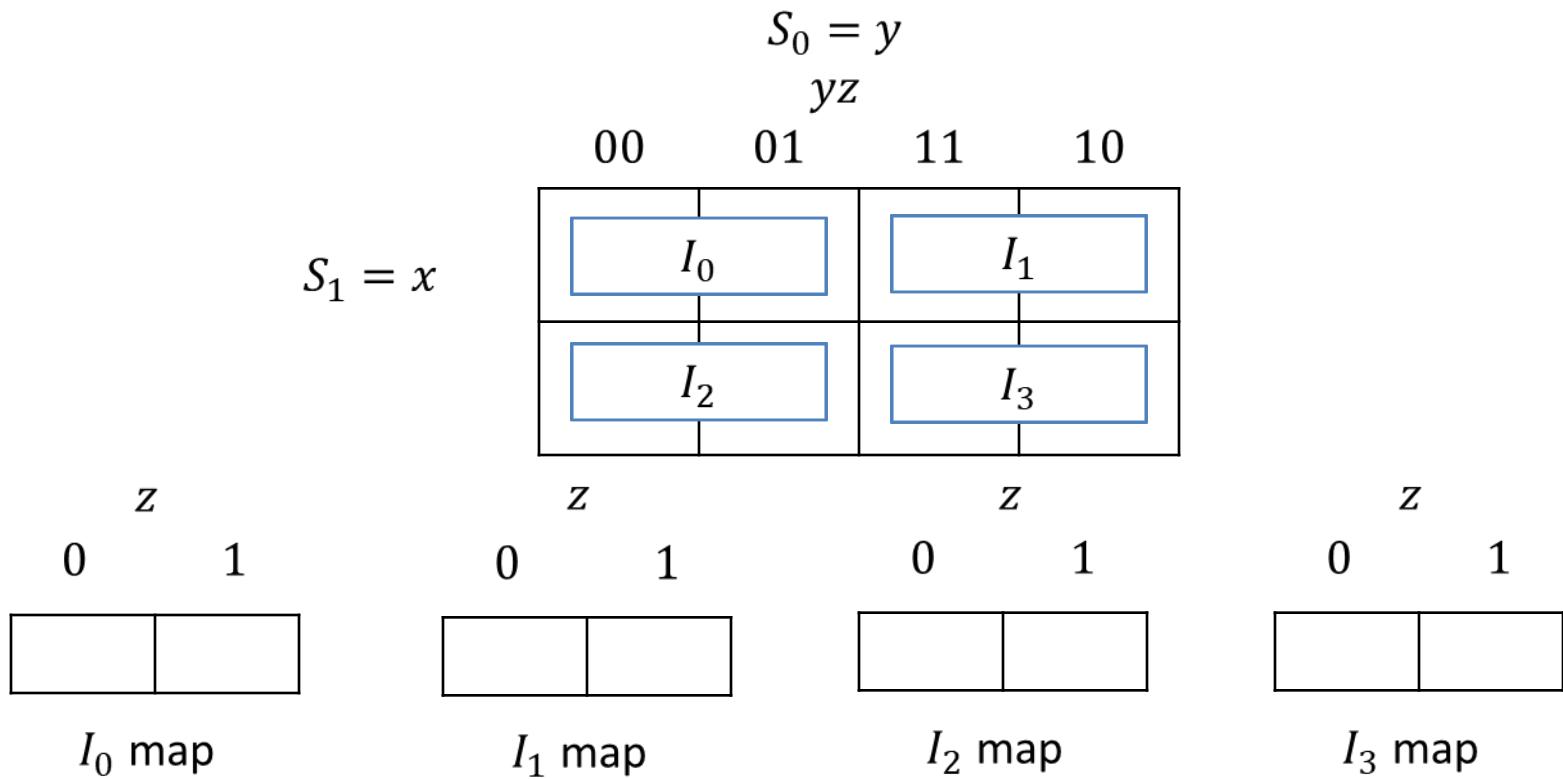
$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



# Logic Design with Multiplexers and K-maps

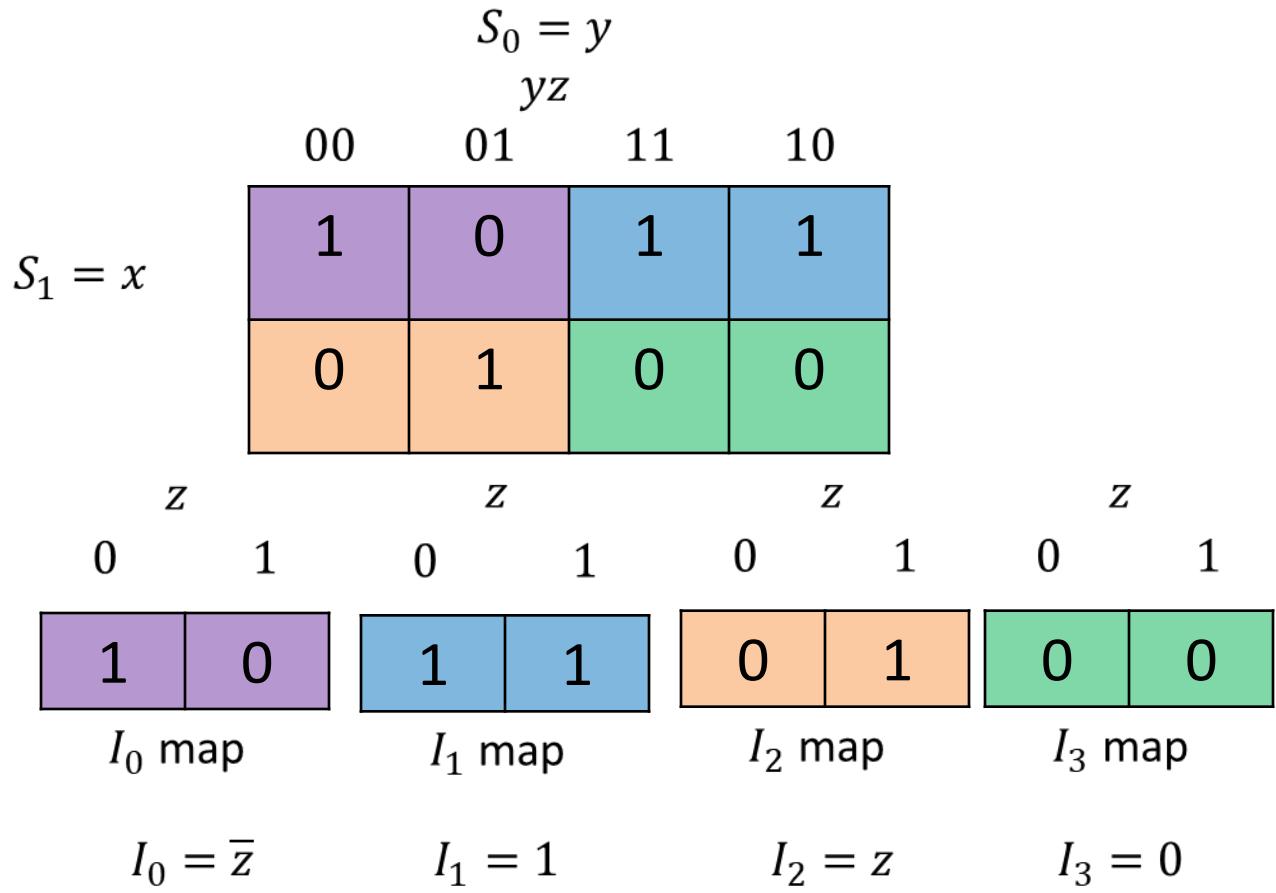
- Consider 3-variable Karnaugh map. Assume  $x$  is placed on the  $S_1$  line and  $y$  is placed on the  $S_0$  line.
- We get that the output is:  $I_0\bar{x}\bar{y} + I_1\bar{x}y + I_2x\bar{y} + I_3xy$
- $I_0\bar{x}\bar{y}$  corresponds to those cells in which  $x = 0, y = 0$
- $I_1\bar{x}y$  corresponds to those cells in which  $x = 0, y = 1$
- $I_2x\bar{y}$  corresponds to those cells in which  $x = 1, y = 0$
- $I_3xy$  corresponds to those cells in which  $x = 1, y = 1$

# K-map representation



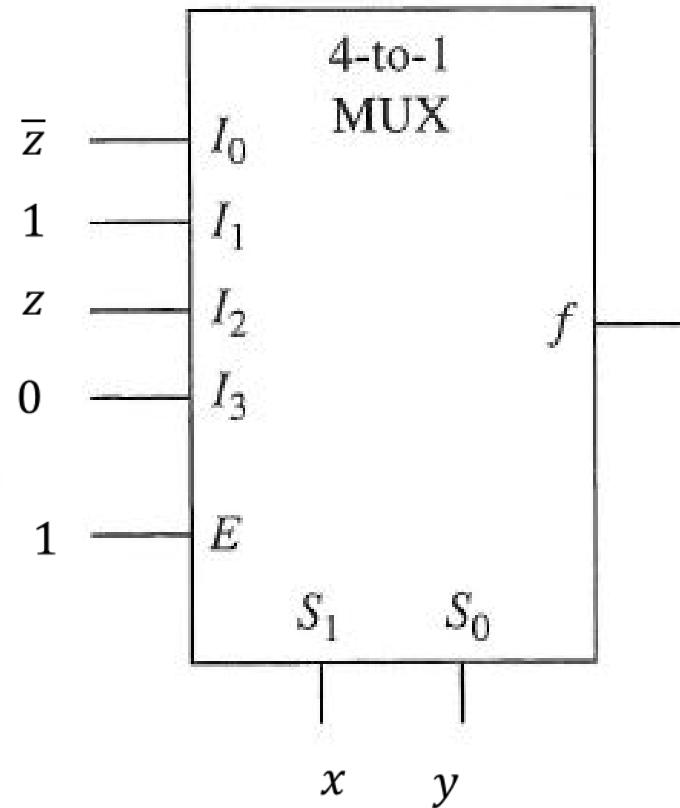
# Example

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



# Realization

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



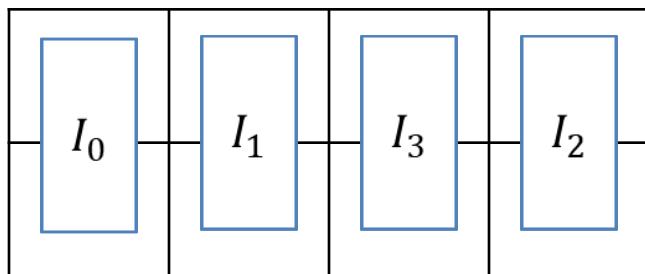
# Alternative Structures

$$S_1 = y, S_0 = z$$

$yz$

00      01      11      10

$x$

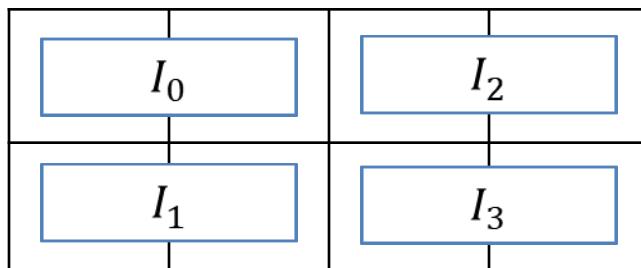


$$S_1 = y$$

$yz$

00      01      11      10

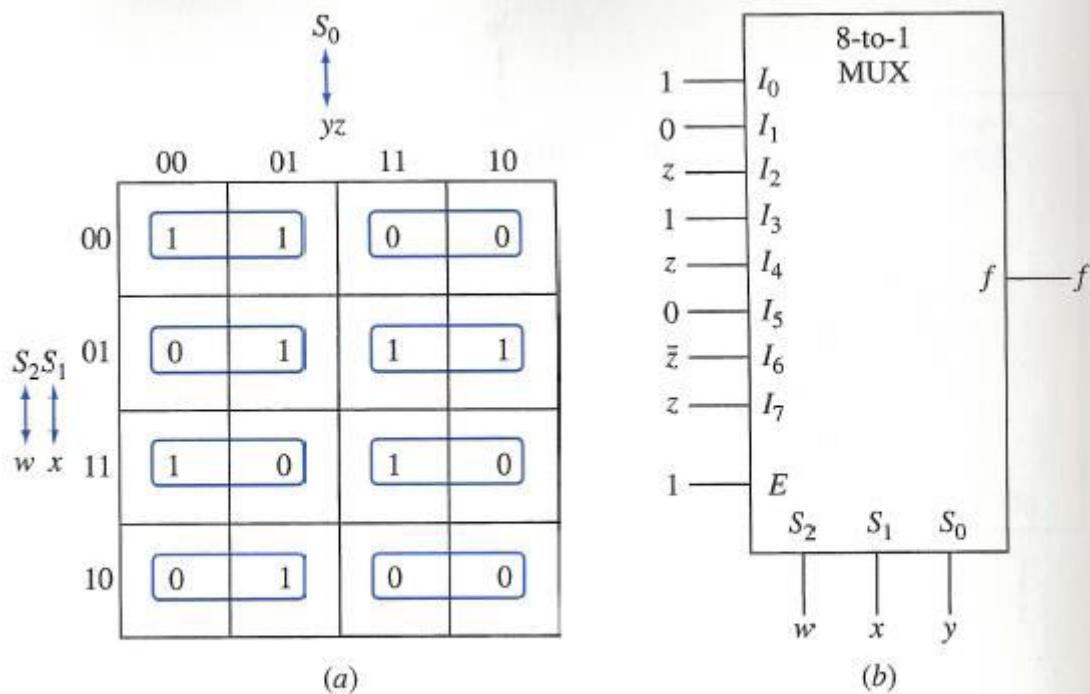
$S_0 = x$



Note that order of variables on input lines matters!

# 8-to-1-line multiplexers and 4-variable Boolean functions

- Can do the same thing, three variables are placed on select lines, inputs to the data lines are single-variable functions.
- Example:



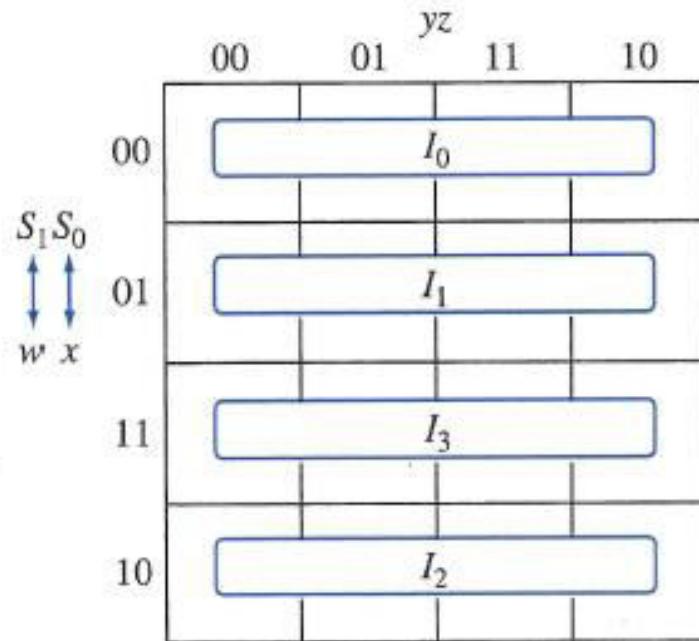
**Figure 5.45**

Realization of  $f(w,x,y,z) = \sum m(0,1,5,6,7,9,12,15)$ .  
(a) Karnaugh map. (b) Multiplexer realization.

# Can we do better?

- By allowing realizations of  $m$ -variable functions as inputs to the data input lines,  $2^n$ -to-1-line multiplexers can be used in the realization of  $(n + m)$ -variable functions.
- E.g.: input variables w and x are applied to the  $S_1, S_0$  select inputs. Functions of the y and z variables appear at the data input lines.

# K-map Structure



**Figure 5.46** Using a four-variable Karnaugh map to obtain a Boolean function realization with a 4-to-1-line multiplexer.

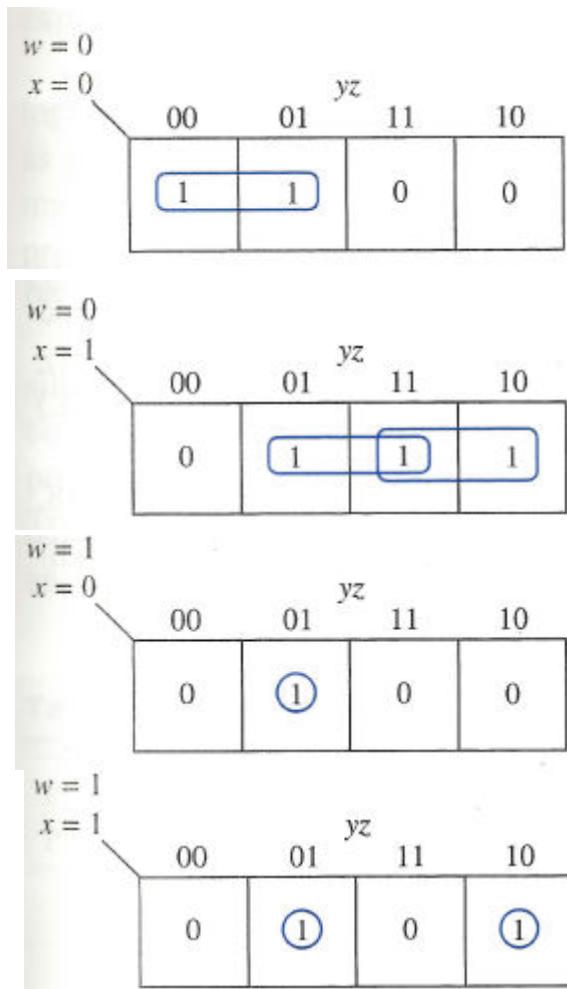
# Example:

$$f(x,y,z) = \sum m(0,1,5,6,7,9,13,14)$$

		yz				
		00	01	11	10	
		00	1	1	0	0
		01	0	1	1	1
		11	0	1	0	1
		10	0	1	0	0

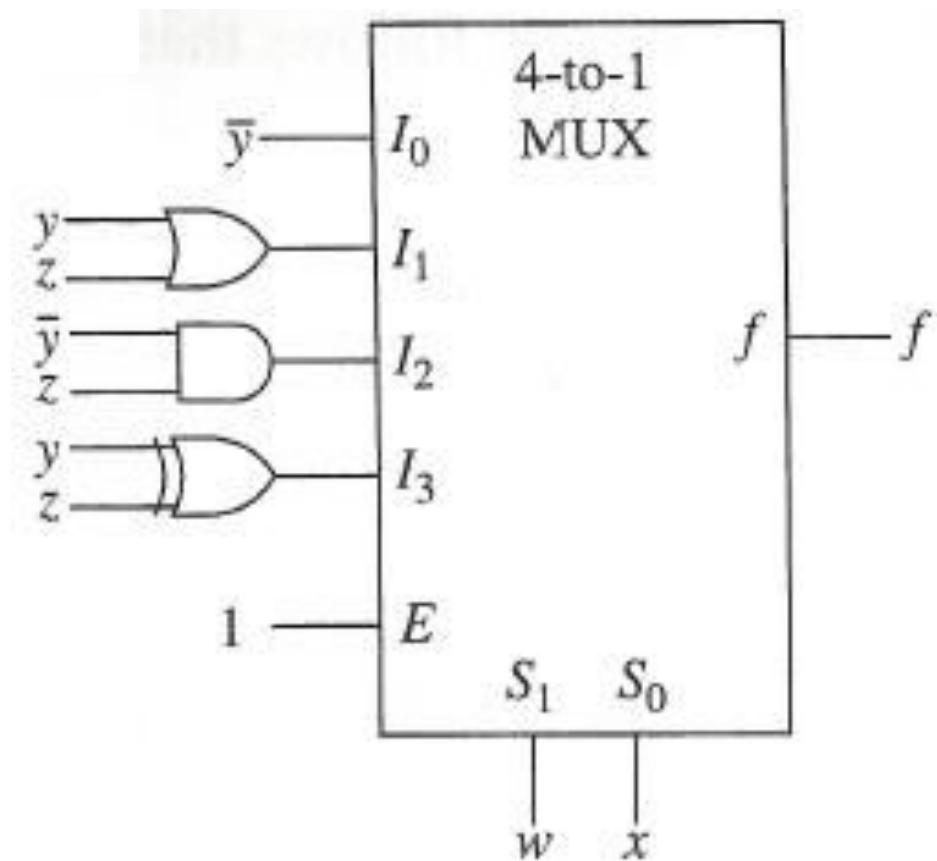
Inputs:  $w = S_1 S_0$ ,  $x = w \cdot x$

Outputs:  $I_0, I_1, I_2, I_3$

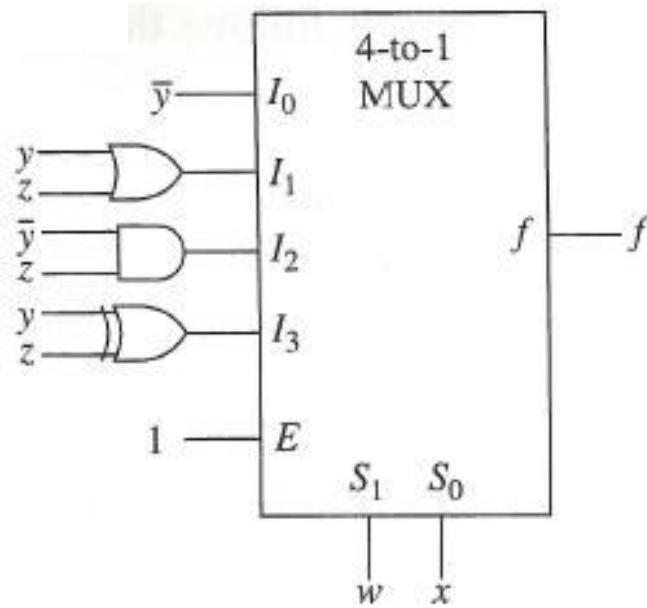


# Example

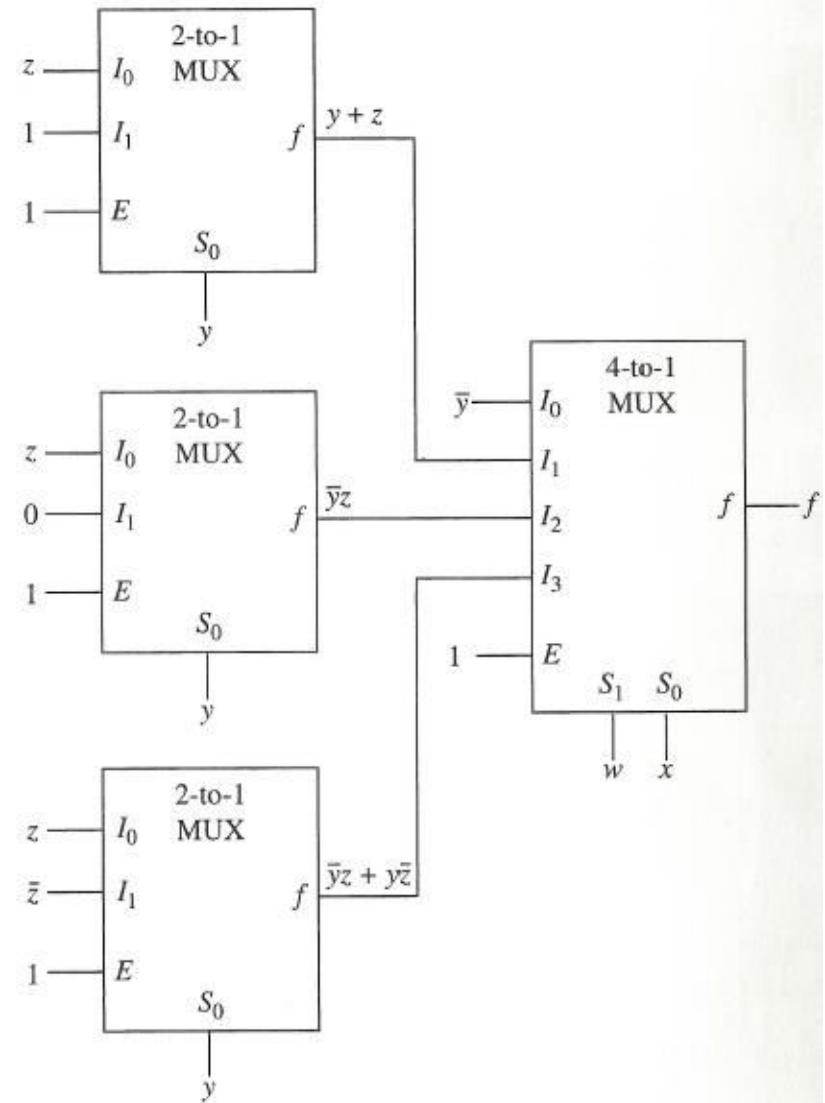
$w = 0$	$x = 0$	<table border="1"> <thead> <tr> <th>00</th><th>01</th><th><math>yz</math></th><th>11</th><th>10</th></tr> </thead> <tbody> <tr> <td>1</td><td>1</td><td></td><td>0</td><td>0</td></tr> </tbody> </table>	00	01	$yz$	11	10	1	1		0	0
00	01	$yz$	11	10								
1	1		0	0								
$w = 0$	$x = 1$	<table border="1"> <thead> <tr> <th>00</th><th>01</th><th><math>yz</math></th><th>11</th><th>10</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	00	01	$yz$	11	10	0	1	1	1	1
00	01	$yz$	11	10								
0	1	1	1	1								
$w = 1$	$x = 0$	<table border="1"> <thead> <tr> <th>00</th><th>01</th><th><math>yz</math></th><th>11</th><th>10</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td><td></td><td>0</td><td>0</td></tr> </tbody> </table>	00	01	$yz$	11	10	0	1		0	0
00	01	$yz$	11	10								
0	1		0	0								
$w = 1$	$x = 1$	<table border="1"> <thead> <tr> <th>00</th><th>01</th><th><math>yz</math></th><th>11</th><th>10</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td><td></td><td>0</td><td>1</td></tr> </tbody> </table>	00	01	$yz$	11	10	0	1		0	1
00	01	$yz$	11	10								
0	1		0	1								



# Example

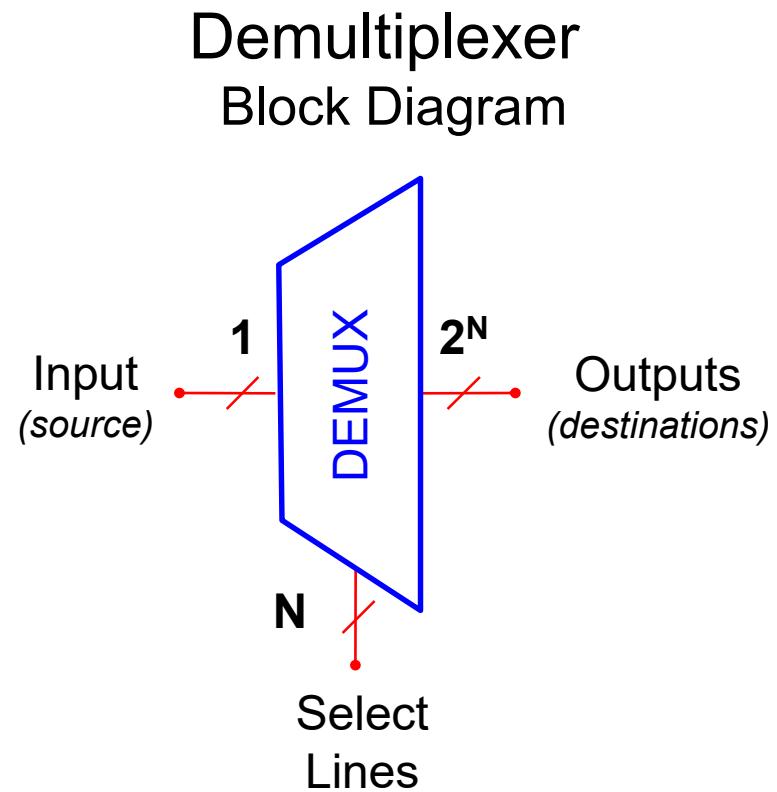


Multiplexer Tree

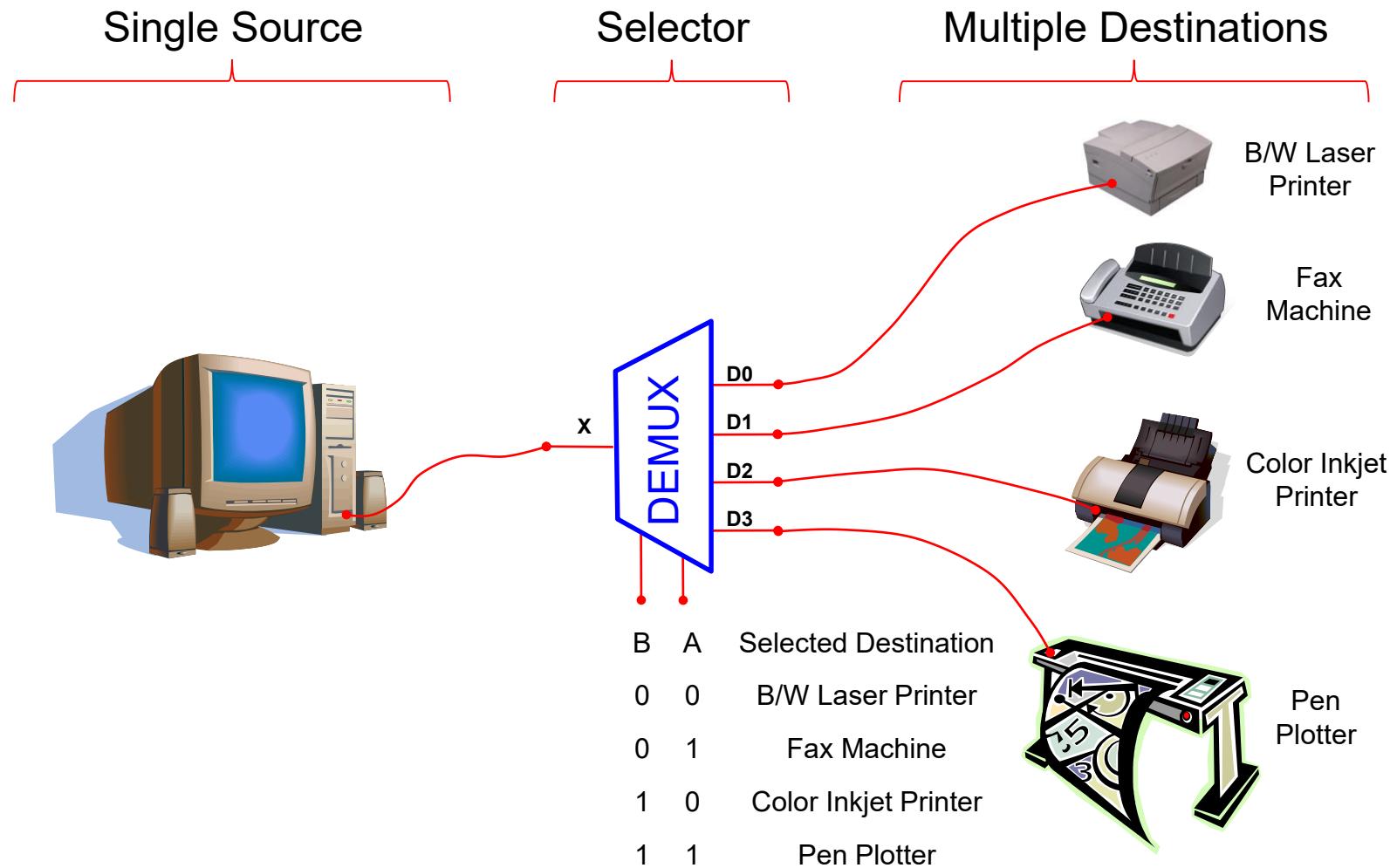


# What is a Demultiplexer (DEMUX)?

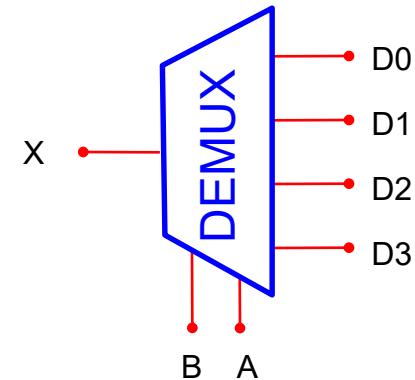
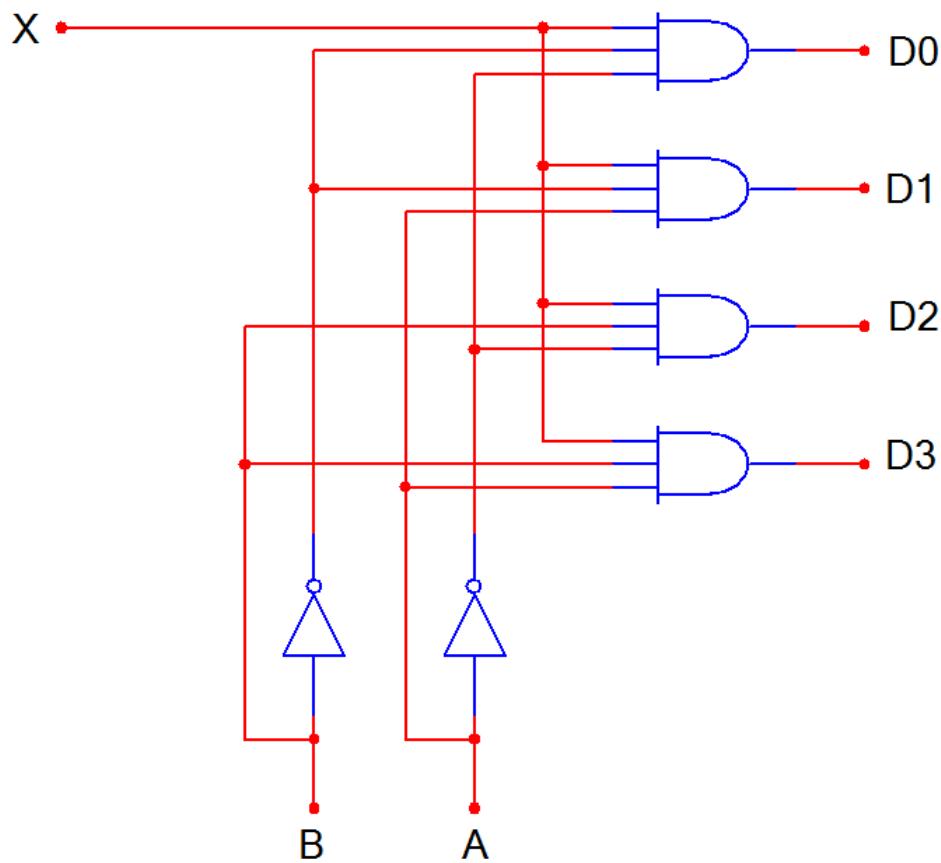
- A DEMUX is a digital switch with a single input (source) and a multiple outputs (destinations).
- The select lines determine which output the input is connected to.
- DEMUX Types
  - 1-to-2 (1 select line)
  - 1-to-4 (2 select lines)
  - 1-to-8 (3 select lines)
  - 1-to-16 (4 select lines)



# Typical Application of a DEMUX



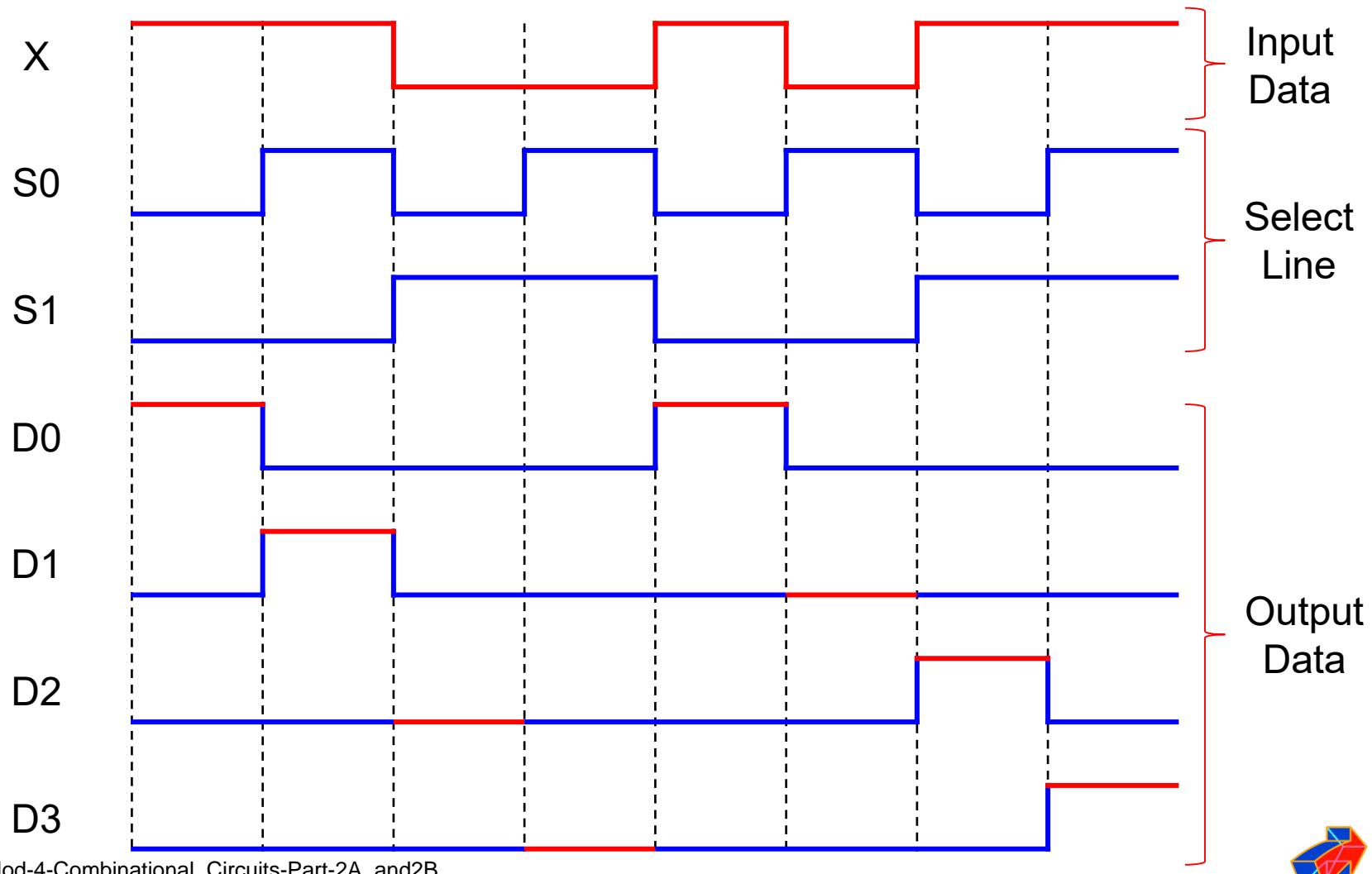
# 1-to-4 De-Multiplexer (DEMUX)



B	A	D0	D1	D2	D3
0	0	X	0	0	0
0	1	0	X	0	0
1	0	0	0	X	0
1	1	0	0	0	X

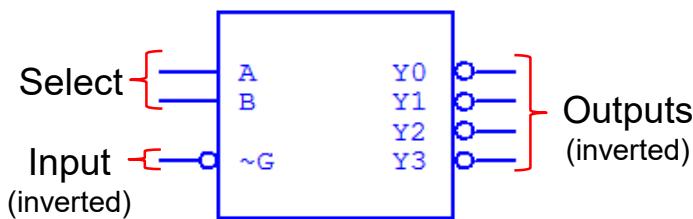


# 1-to-4 De-Multiplexer Waveforms

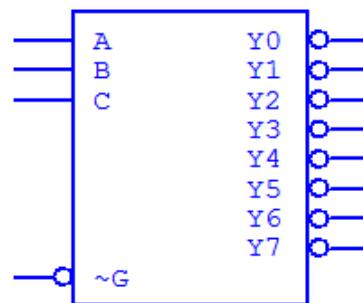


# Medium Scale Integration DEMUX

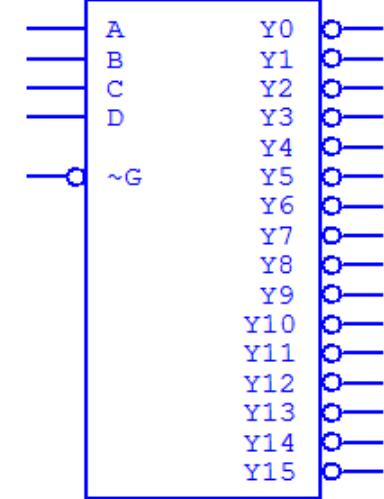
1-to-4 DEMUX



1-to-8 DEMUX



1-to-16 DEMUX



Note : Most Medium Scale Integrated (MSI) DEMUXs , like the three shown, have outputs that are inverted. This is done because it requires few logic gates to implement DEMUXs with inverted outputs rather than no-inverted outputs.

# Study Problem

- ◆ Course Book Chapter – 4 Problems
  - 4 – 31
    - ✓ Construct a  $16 \times 1$  multiplexer with two  $8 \times 1$  and one  $2 \times 1$  multiplexer. Use block diagrams

# Study Problem

- ◆ Course Book Chapter – 4 Problems

- 4 – 34

An 8x1 multiplexer has inputs A, B, and C connected to the selection inputs  $S_2$ ,  $S_1$ , and  $S_0$  respectively.

The data inputs

$$I_1 = I_2 = I_7 = 0;$$

$$I_3 = I_5 = 1;$$

$$I_0 = I_4 = D;$$

$$I_6 = D'$$

Determine the Boolean function that the multiplexer implements

# Study Problems

## ◆ Course Book Chapter – 4 Problems

- 4 – 1
- 4 – 4
- 4 – 6
- 4 – 11
- 4 – 20
- 4 – 21
- 4 – 25
- 4 – 32
- 4 – 33
- 4 – 35

# CSE1003-Digital Logic Design

## Module:5 Sequential Circuits-I

### Part-2

**Dr.Penchalaiah Palla**

Dept. of Micro and Nanoelectronics  
School of Electronics Engineering,VIT,  
Vellore

# **Module:5 Sequential Circuits-I**

<b>Module:5</b>	<b>SEQUENTIAL CIRCUITS – I</b>	<b>6 hours</b>
Flip Flops - Sequential Circuit: Design and Analysis - Finite State Machine: Moore and Mealy model - Sequence Detector.		
<b>Module:6</b>	<b>SEQUENTIAL CIRCUITS – II</b>	<b>7 hours</b>
Registers - Shift Registers - Counters - Ripple and Synchronous Counters - Modulo counters - Ring and Johnson counters		

# Introduction: Sequential Circuits

## ◆ Combinational

- The outputs depend only on the current input values
- It uses only logic gates

## ◆ Sequential

- The outputs depend on the current and past input values
- It uses logic gates and storage elements
- Example
  - ✓ Vending machine
- They are referred as finite state machines since they have a finite number of states

# Sequential Logic Design with Flip-flops

- Introduction
- Flip-flop Characteristic Tables
- Sequential Circuit Analysis
- Flip-flop Input Functions
- Analysis: Example #2
- Analysis: Example #3
- Flip-flop Excitation Tables



# Sequential Logic Design with Flip-flops

- Sequential Circuit Design
- Design: Example #1
- Design: Example #2
- Design: Example #3
- Design of Synchronous Counters



# Introduction

- Sequential circuits has an extra dimension – *time*.
- Combinational circuit output depends only on the present inputs
- Sequential circuit output depends on the history of past inputs as well
- More powerful than combinational circuit, able to model situations that cannot be modeled by combinational circuits
- Building blocks of **synchronous sequential logic** circuits: *gates* and *flip-flops*.
- Flip-flops make up the *memory M* while the gates form one or more combinational subcircuits  $C_1, C_2, \dots, C_q$ .



# Difference Between Analysis and Design

- *Analysis*: Starting from a circuit diagram, derive the state table or state diagram.
- *Characteristic tables* are used in analysis.
- *Design*: Starting from a set of specifications (in the form of state equations, state table, or state diagram), derive the logic circuit.
- *Excitation tables* are used in design.

# Flip-flop Characteristic Tables

- Each type of flip-flop has its own behavior. The **characteristic tables** for the various types of flip-flops are shown below:

$J$	$K$	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q(t)'$	Toggle

JK Flip-flop

$S$	$R$	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

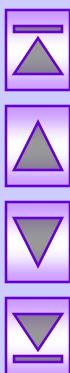
SR Flip-flop

$D$	$Q(t+1)$	
0	0	Reset
1	1	Set

D Flip-flop

$T$	$Q(t+1)$
0	$Q(t)$
1	$Q(t)'$

T Flip-flop



# Sequential Circuit Analysis

- Given a sequential circuit diagram, analyze its behaviour by deriving its *state table* and hence its *state diagram*.
- Requires *state equations* to be derived for the flip-flop inputs, as well as *output functions* for the circuit outputs other than the flip-flops (if any).
- We use  $A(t)$  and  $A(t+1)$  to represent the present state and next state, respectively, of a flip-flop represented by A.
- Alternatively, we could simply use  $A$  and  $A^+$  for the present state and next state respectively.



# Sequential Circuit Analysis

- Example #1 (using D flip-flops):

State equations:

$$A^+ = A.x + B.x$$

$$B^+ = A'.x$$

Output function:

$$y = (A + B).x'$$

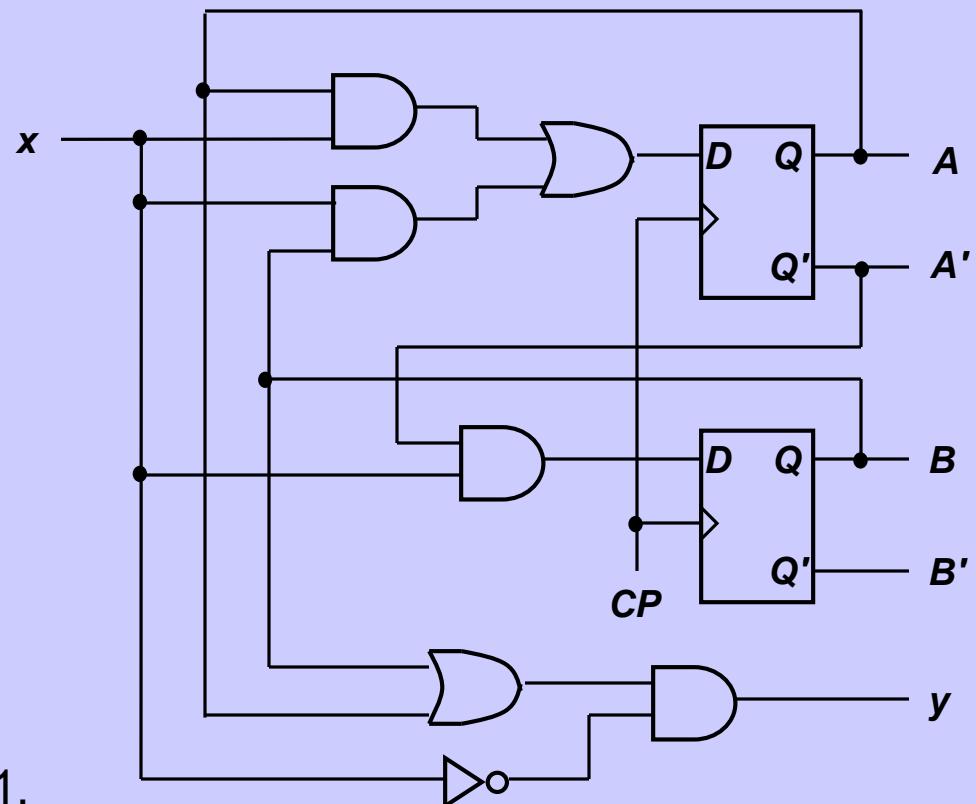
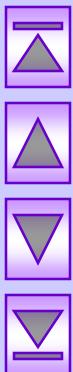


Figure 1.

# Sequential Circuit Analysis

- From the *state equations* and *output function*, we derive the **state table**, consisting of all possible binary combinations of present states and inputs.
- State table
  - ❖ Similar to truth table.
  - ❖ Inputs and present state on the left side.
  - ❖ Outputs and next state on the right side.
- $m$  flip-flops and  $n$  inputs  $\rightarrow 2^{m+n}$  rows.



# Sequential Circuit Analysis

- State table for the circuit of Figure 1:

State equations:

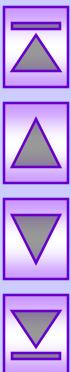
$$A^+ = A \cdot x + B \cdot x$$

$$B^+ = A' \cdot x$$

Output function:

$$y = (A + B) \cdot x'$$

Present State		Input <i>x</i>	Next State		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i> <sup>+</sup>	<i>B</i> <sup>+</sup>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0



# Sequential Circuit Analysis

- Alternate form of state table:

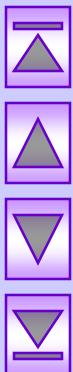
Present State		Input $x$	Next State		Output $y$
$A$	$B$		$A^+$	$B^+$	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Present State	Next State		Output	
	$x=0$	$x=1$	$x=0$	$x=1$
$AB$	$A^+B^+$	$A^+B^+$	$y$	$y$
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0



# Sequential Circuit Analysis

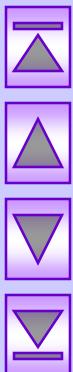
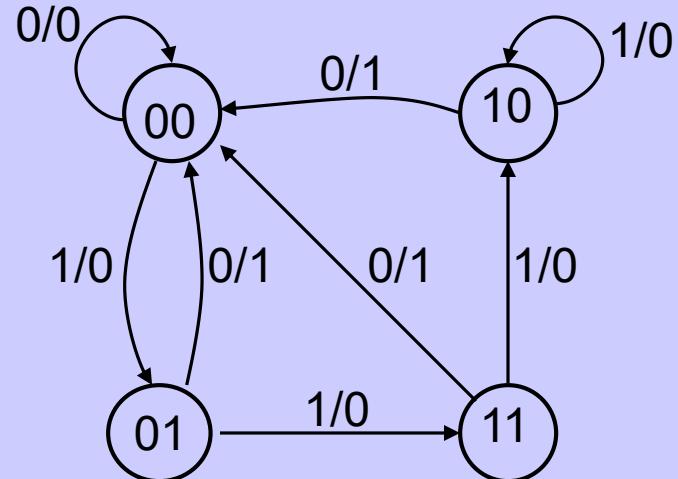
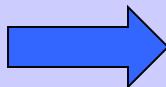
- From the *state table*, we can draw the *state diagram*.
- State diagram
  - ❖ Each state is denoted by a circle.
  - ❖ Each arrow (between two circles) denotes a transition of the sequential circuit (a row in state table).
  - ❖ A label of the form  $a/b$  is attached to each arrow where  $a$  denotes the inputs while  $b$  denotes the outputs of the circuit in that transition.
- Each combination of the flip-flop values represents a state. Hence,  $m$  flip-flops  $\rightarrow$  up to  $2^m$  states.



# Sequential Circuit Analysis

- State diagram of the circuit of Figure 1:

Present State $AB$	Next State		Output	
	$x=0$ $A^+B^+$	$x=1$ $A^+B^+$	$x=0$ $y$	$x=1$ $y$
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0



# Flip-flop Input Functions

- The outputs of a sequential circuit are functions of the present states of the flip-flops and the inputs. These are described algebraically by the *circuit output functions*.
  - ❖ In Figure 1:  $y = (A + B).x'$
- The part of the circuit that generates inputs to the flip-flops are described algebraically by the *flip-flop input functions* (or *flip-flop input equations*).
- The flip-flop input functions determine the next state generation.
- From the flip-flop input functions and the characteristic tables of the flip-flops, we obtain the next states of the flip-flops.



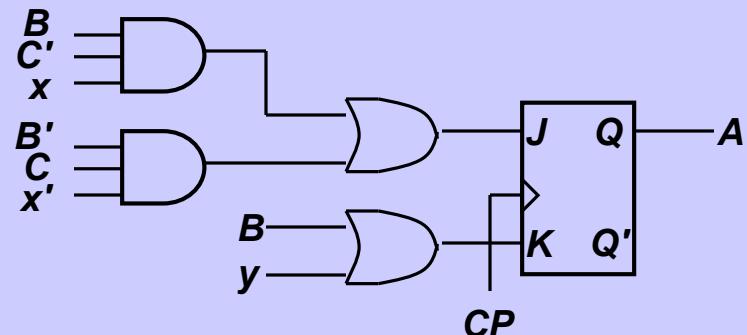
# Flip-flop Input Functions

- Example: circuit with a JK flip-flop.
- We use 2 letters to denote each flip-flop input: the first letter denotes the input of the flip-flop (*J* or *K* for JK flip-flop, *S* or *R* for SR flip-flop, *D* for D flip-flop, *T* for T flip-flop) and the second letter denotes the name of the flip-flop.



$$JA = B.C'.x + B'.C.x'$$

$$KA = B + y$$



# Flip-flop Input Functions

- In Figure 1, we obtain the following state equations by observing that  $Q^+ = DQ$  for a D flip-flop:

$$A^+ = A \cdot x + B \cdot x \quad (\text{since } DA = A \cdot x + B \cdot x)$$

$$B^+ = A' \cdot x \quad (\text{since } DB = A' \cdot x)$$

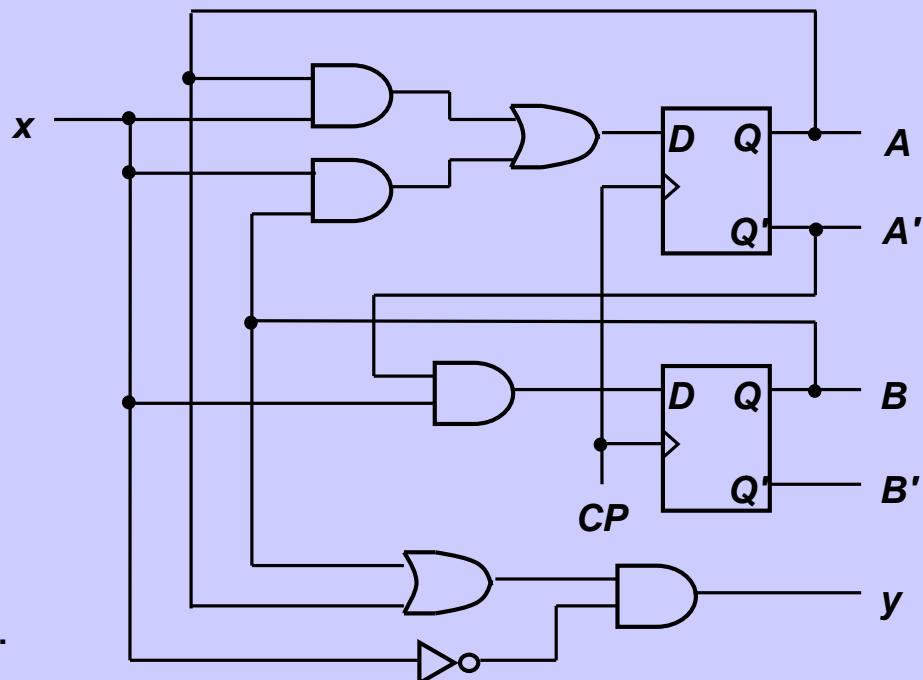


Figure 1.

# Analysis: Example #2

- Given Figure 2, a sequential circuit with two JK flip-flops  $A$  and  $B$ , and one input  $x$ .

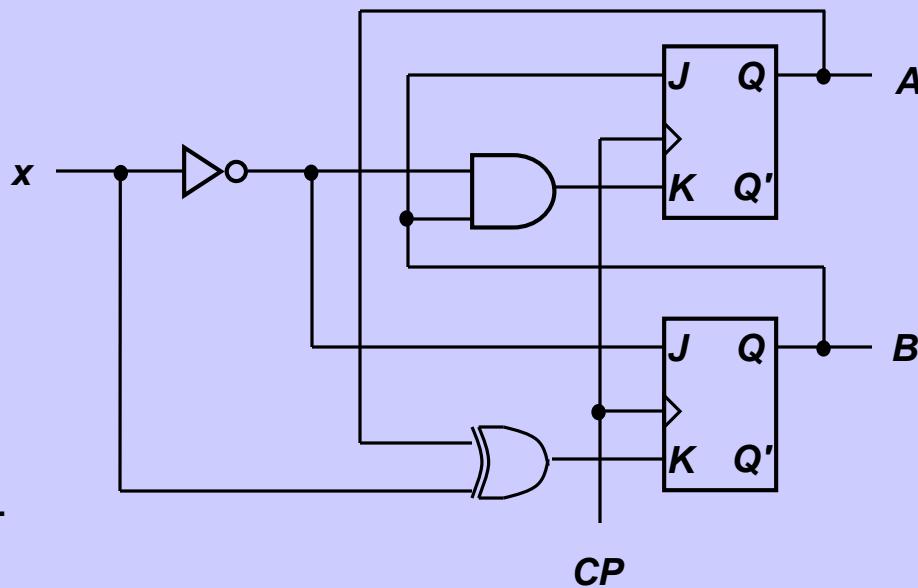


Figure 2.

- Obtain the flip-flop input functions from the circuit:

$$JA = B$$

$$JB = x'$$

$$KA = B \cdot x'$$

$$KB = A' \cdot x + A \cdot x' = A \oplus x$$

# Analysis: Example #2

- Flip-flop input functions:

$$JA = B$$

$$JB = x'$$

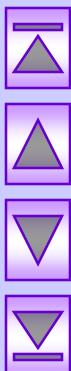
$$KA = B \cdot x'$$

$$JB = A' \cdot x + A \cdot x' = A \oplus x$$

- Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

J	K	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q(t)'$	Toggle

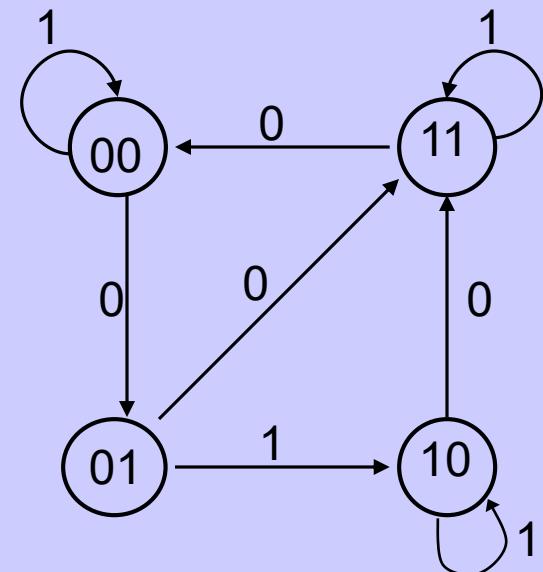
Present state		Input x	Next state		Flip-flop inputs			
A	B		$A^+$	$B^+$	JA	KA	JB	KB
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0



# Analysis: Example #2

- Draw the state diagram from the state table.

Present state		Input $x$	Next state		Flip-flop inputs			
$A$	$B$		$A^+$	$B^+$	$JA$	$KA$	$JB$	$KB$
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0



# Analysis: Example #3

- Derive the state table and state diagram of the following circuit.

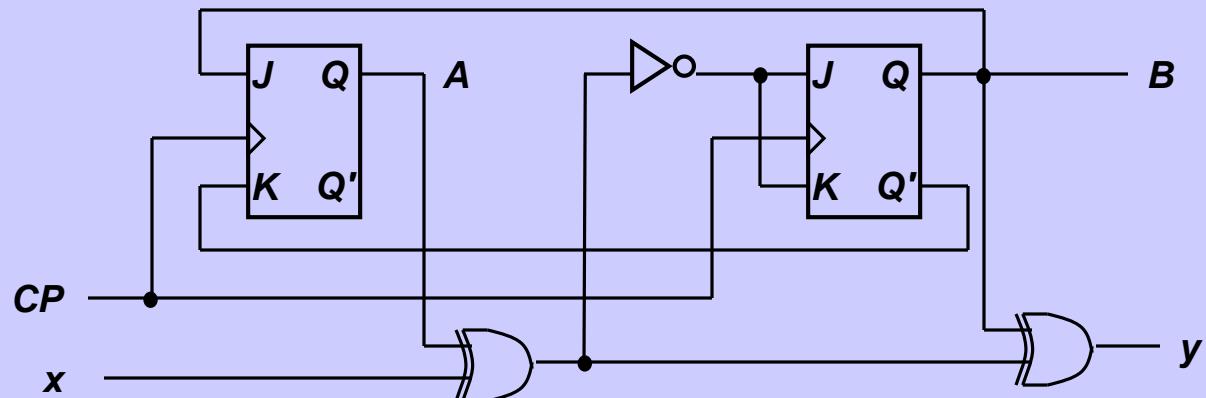


Figure 3.

- Flip-flop input functions:

$$JA = B$$

$$JB = KB = (A \oplus x)' = A \cdot x + A' \cdot x'$$

$$KA = B'$$

# Analysis: Example #3

- Flip-flop input functions:

$$JA = B$$

$$JB = KB = (A \oplus x)' = A.x + A'.x'$$

$$KA = B'$$

- State table:

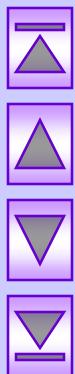
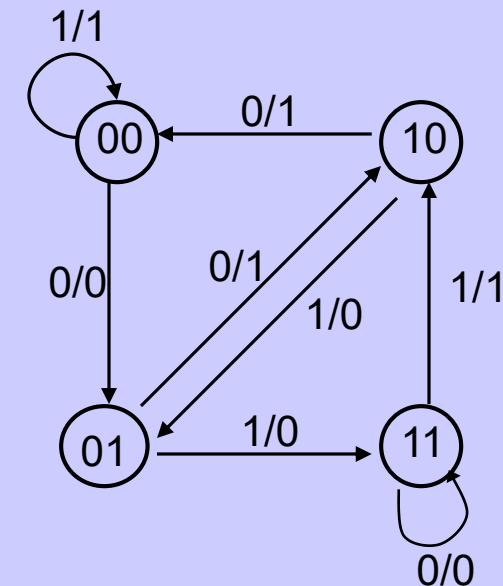
Present state		Input $x$	Next state		Output $y$	Flip-flop inputs			
$A$	$B$		$A^+$	$B^+$		$JA$	$KA$	$JB$	$KB$
0	0	0	0	1	0	0	1	1	1
0	0	1	0	0	1	0	1	0	0
0	1	0	1	0	1	1	0	1	1
0	1	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	1	0	0
1	0	1	0	1	0	0	1	1	1
1	1	0	1	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1	1



# Analysis: Example #3

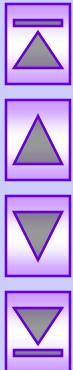
- State diagram:

Present state		Input $x$	Next state		Output $y$	Flip-flop inputs			
$A$	$B$		$A^+$	$B^+$		$JA$	$KA$	$JB$	$KB$
0	0	0	0	1	0	0	1	1	1
0	0	1	0	0	1	0	1	0	0
0	1	0	1	0	1	1	0	1	1
0	1	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	1	0	0
1	0	1	0	1	0	0	1	1	1
1	1	0	1	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1	1



# Flip-flop Excitation Tables

- *Design:* Starting from a set of specifications (in the form of state equations, state table, or state diagram), derive the logic circuit.
- *Excitation tables* are used in design.



# Flip-flop Excitation Tables

- *Excitation tables*: given the required transition from present state to next state, determine the flip-flop input(s).

$Q$	$Q^+$	$J$	$K$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

$Q$	$Q^+$	$S$	$R$
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

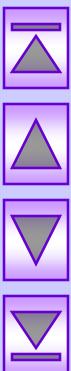
SR Flip-flop

$Q$	$Q^+$	$D$
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

$Q$	$Q^+$	$T$
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop



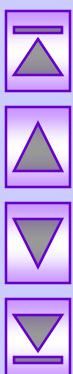
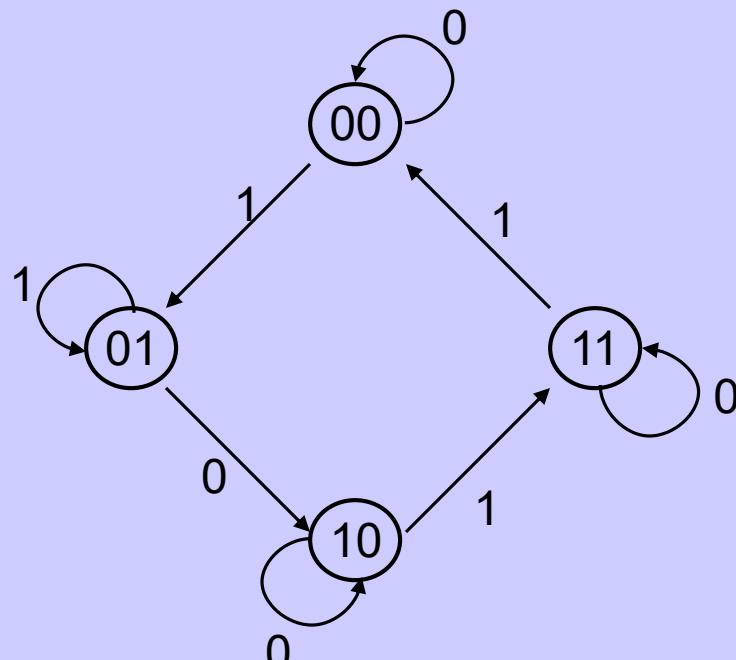
# Sequential Circuit Design

- Design procedure:
  - ❖ Start with circuit specifications – description of circuit behaviour.
  - ❖ Derive the state table.
  - ❖ Perform state reduction if necessary.
  - ❖ Perform state assignment.
  - ❖ Determine number of flip-flops and label them.
  - ❖ Choose the type of flip-flop to be used.
  - ❖ Derive circuit excitation and output tables from the state table.
  - ❖ Derive circuit output functions and flip-flop input functions.
  - ❖ Draw the logic diagram.



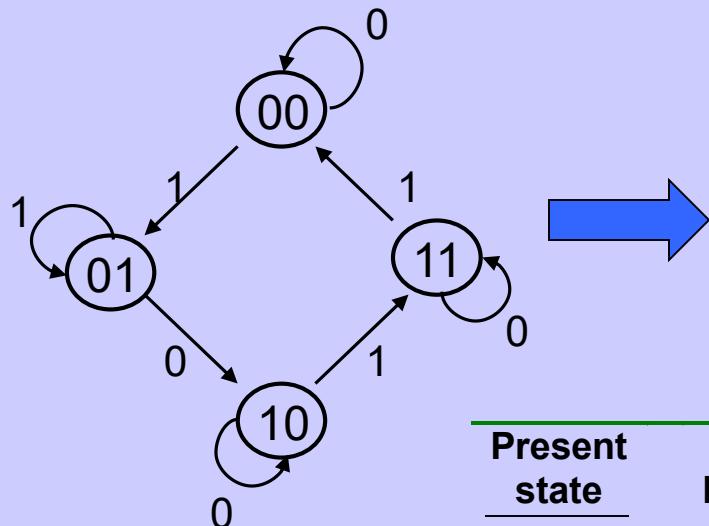
# Design: Example #1

- Given the following state diagram, design the sequential circuit using JK flip-flops.



# Design: Example #1

- Circuit state/excitation table, using JK flip-flops.



Present State	Next State	
	$x=0$	$x=1$
$AB$	$A^+B^+$	$A^+B^+$
00	00	01
01	10	01
10	10	11
11	11	00

$Q$	$Q^+$	$J$	$K$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

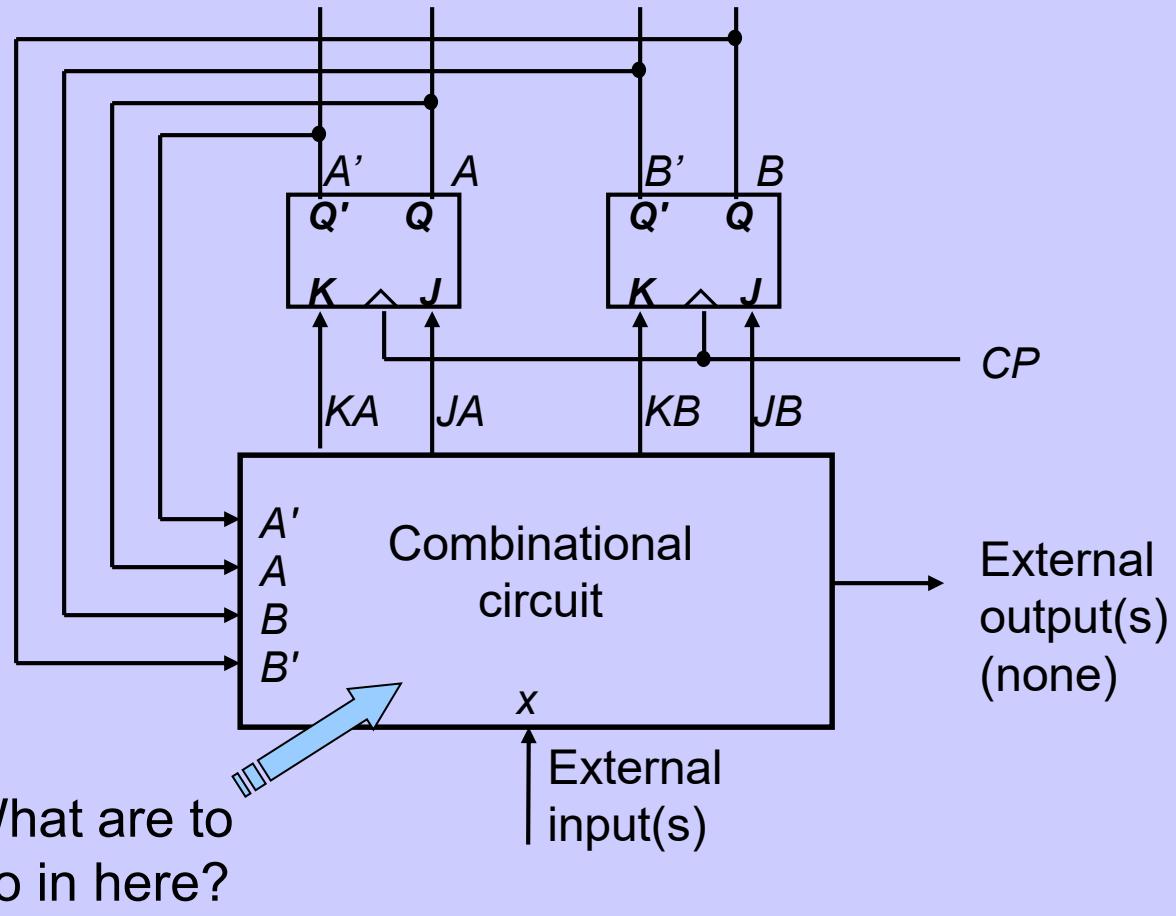
JK Flip-flop's  
excitation table.

Present state	Input	Next state		Flip-flop inputs						
		$A$	$B$	$x$	$A^+$	$B^+$	$JA$	$KA$	$JB$	$KB$
0 0		0	0	0	0	0	0	X	0	X
0 0		0	0	1	0	1	0	X	1	X
0 1		0	1	0	1	0	1	X	X	1
0 1		0	1	1	0	1	0	X	X	0
1 0		1	0	0	1	0	X	0	0	X
1 0		1	0	1	1	1	X	0	1	X
1 1		X	0	0	1	0	X	0	0	X
1 1		X	0	1	1	1	X	1	X	1



# Design: Example #1

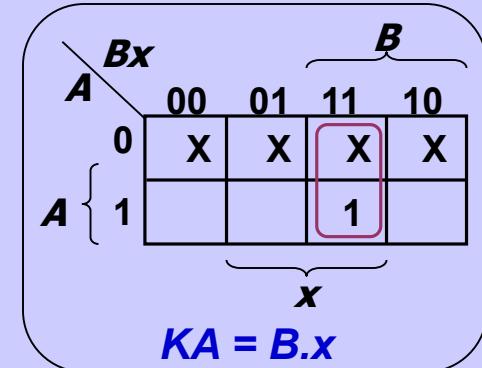
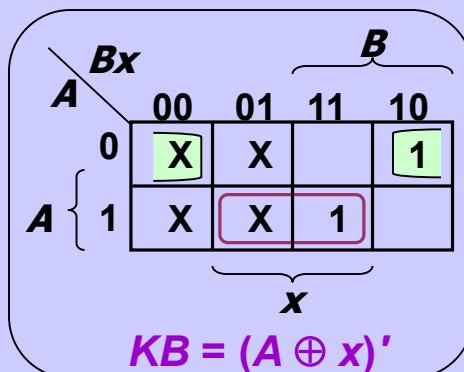
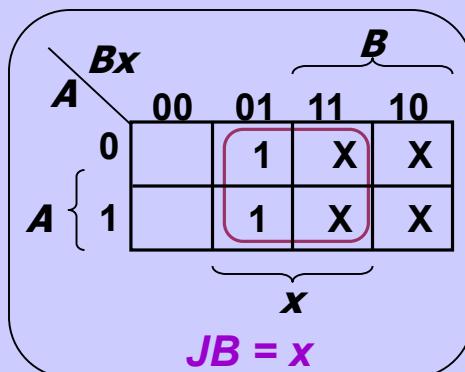
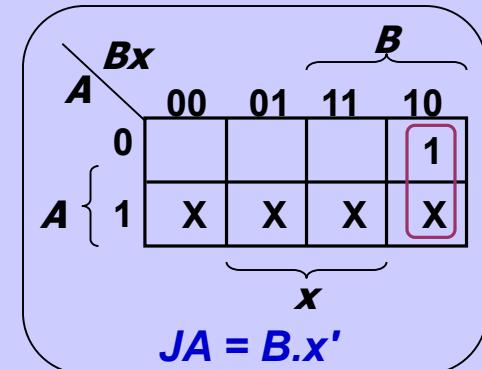
- Block diagram.



# Design: Example #1

- From state table, get flip-flop input functions.

Present state		Input $x$	Next state		Flip-flop inputs			
$A$	$B$		$A^+$	$B^+$	$JA$	$KA$	$JB$	$KB$
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1



# Design: Example #1

- Flip-flop input functions.

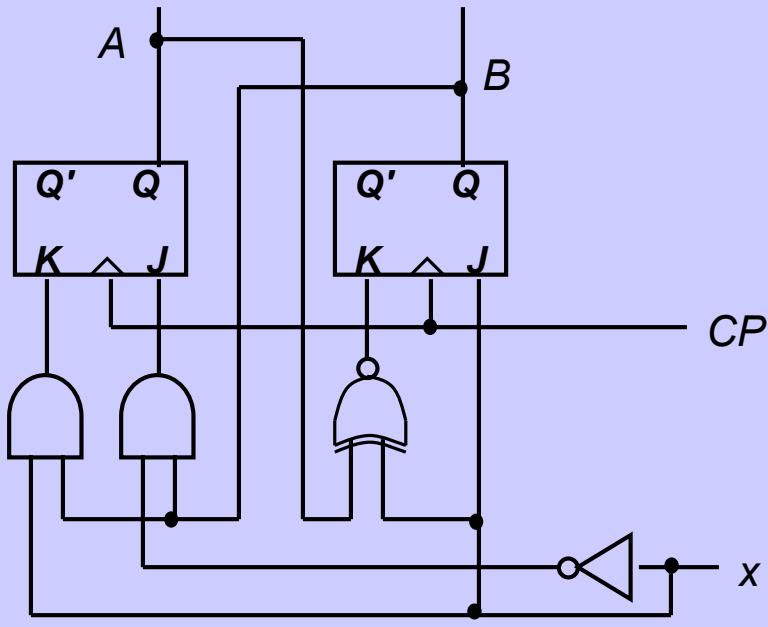
$$JA = B \cdot x'$$

$$JB = x$$

$$KA = B \cdot x$$

$$KB = (A \oplus x)'$$

- Logic diagram:



# Design: Example #2

- Design, using D flip-flops, the circuit based on the state table below. (Exercise: Design it using JK flip-flops.)

Present state		Input <i>x</i>	Next state		Output <i>y</i>
<i>A</i>	<i>B</i>		<i>A</i> <sup>+</sup>	<i>B</i> <sup>+</sup>	
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0



# Design: Example #2

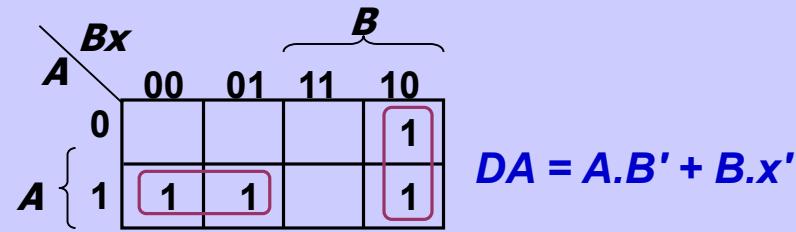
- Determine expressions for flip-flop inputs and the circuit output  $y$ .

Present state		Input $x$	Next state		Output $y$
$A$	$B$		$A^+$	$B^+$	
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

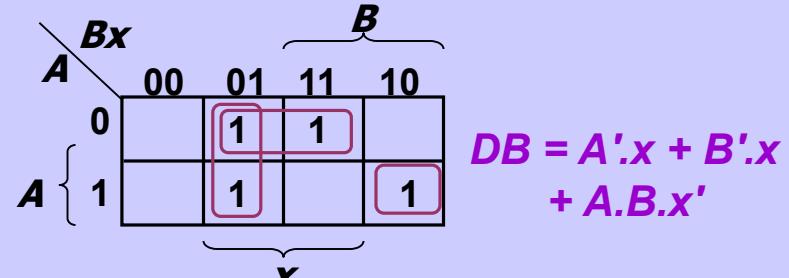
$$DA(A, B, x) = \Sigma m(2,4,5,6)$$

$$DB(A, B, x) = \Sigma m(1,3,5,6)$$

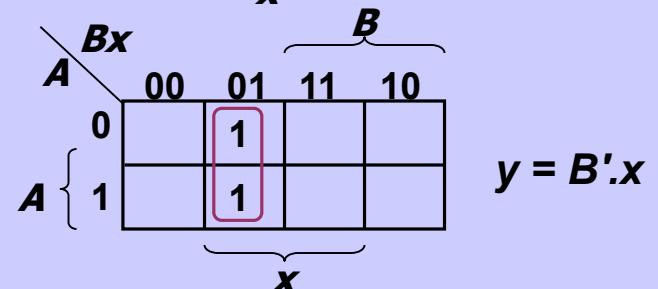
$$y(A, B, x) = \Sigma m(1,5)$$



$$DA = A \cdot B' + B \cdot x'$$



$$DB = A' \cdot x + B' \cdot x + A \cdot B \cdot x'$$



$$y = B' \cdot x$$



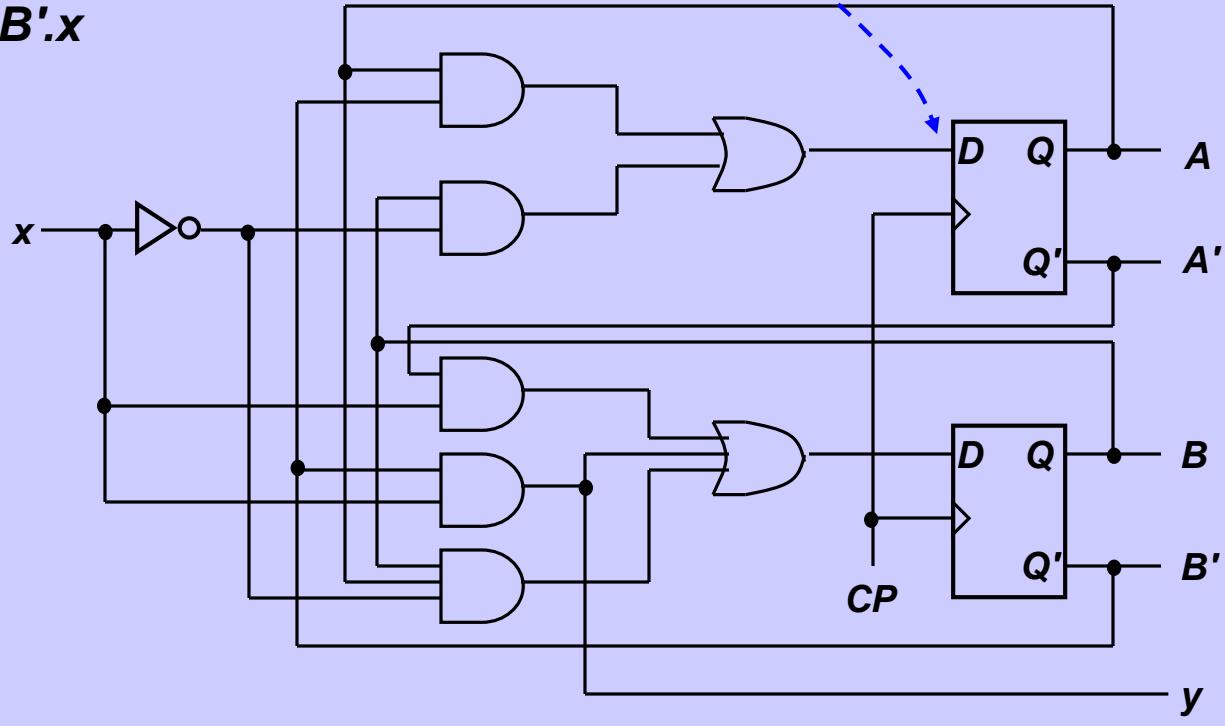
# Design: Example #2

- From derived expressions, draw logic diagram:

$$DA = A \cdot B' + B \cdot x'$$

$$DB = A' \cdot x + B' \cdot x + A \cdot B \cdot x'$$

$$y = B' \cdot x$$



# Design: Example #3

- Design with unused states.

Present state			Input <i>x</i>	Next state			Flip-flop inputs						Output	
<i>A</i>	<i>B</i>	<i>C</i>		<i>A</i> <sup>+</sup>	<i>B</i> <sup>+</sup>	<i>C</i> <sup>+</sup>	<i>SA</i>	<i>RA</i>	<i>SB</i>	<i>RB</i>	<i>SC</i>	<i>RC</i>	<i>y</i>	
0	0	1	0	0	0	1	0	X	0	X	X	0	0	
0	0	1	1	0	1	0	0	X	1	0	0	1	0	
0	1	0	0	0	1	1	0	X	X	0	1	0	0	
0	1	0	1	1	0	0	1	0	0	1	0	X	0	
0	1	1	0	0	0	1	0	X	0	1	X	0	0	
0	1	1	1	1	0	0	1	0	0	1	0	1	0	
1	0	0	0	1	0	1	X	0	0	X	1	0	0	
1	0	0	1	1	0	0	X	0	0	X	0	X	1	
1	0	1	0	0	0	1	0	1	0	X	X	0	0	
1	0	1	1	1	0	0	X	0	0	X	0	1	1	

Given these

Derive these

Unused state 000:

0	0	0	0	X	X	X	X	X	X	X	X	X	X
0	0	0	1	X	X	X	X	X	X	X	X	X	X



# Design: Example #3

- From state table, obtain expressions for flip-flop inputs.

$$SA = B.x$$

		Cx				
		AB	00	01	11	10
		A	X	X		
00			X	X		
01			1	1		
11			X	X	X	X
10			X	X	X	

$x$

$$RA = C.x'$$

		Cx				
		AB	00	01	11	10
		B	X	X	X	X
00			X	X	X	X
01			X			X
11			X	X	X	X
10						1

$x$

$$SB = A'.B'.x$$

		Cx				
		AB	00	01	11	10
		B	X	X	1	
00			X	X	1	
01			X			
11			X	X	X	X
10						

$x$

$$RB = B.C + B.x'$$

# Design: Example #3

- From state table, obtain expressions for flip-flop inputs (cont'd).

$$SC = x'$$

$AB \backslash Cx$	00	01	11	10
00	X	X		X
01	1			X
11	X	X	X	X
10	1			X

$A \left\{ \begin{matrix} 11 \\ 10 \end{matrix} \right\} \quad x$

$AB \backslash Cx$	00	01	11	10
00	X	X	1	
01			X	1
11	X	X	X	X
10	X		1	

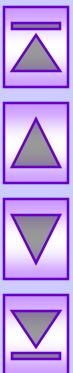
$B \left\{ \begin{matrix} 11 \\ 10 \end{matrix} \right\} \quad x$

$$RC = x$$

$AB \backslash Cx$	00	01	11	10
00	X	X		
01				
11	X	X	X	X
10		1	1	

$B \left\{ \begin{matrix} 11 \\ 10 \end{matrix} \right\} \quad x$

$$y = A.x$$



# Design: Example #3

- From derived expressions, draw logic diagram:

$$SA = B.x$$

$$RA = C.x'$$

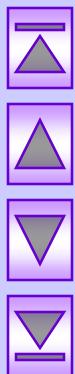
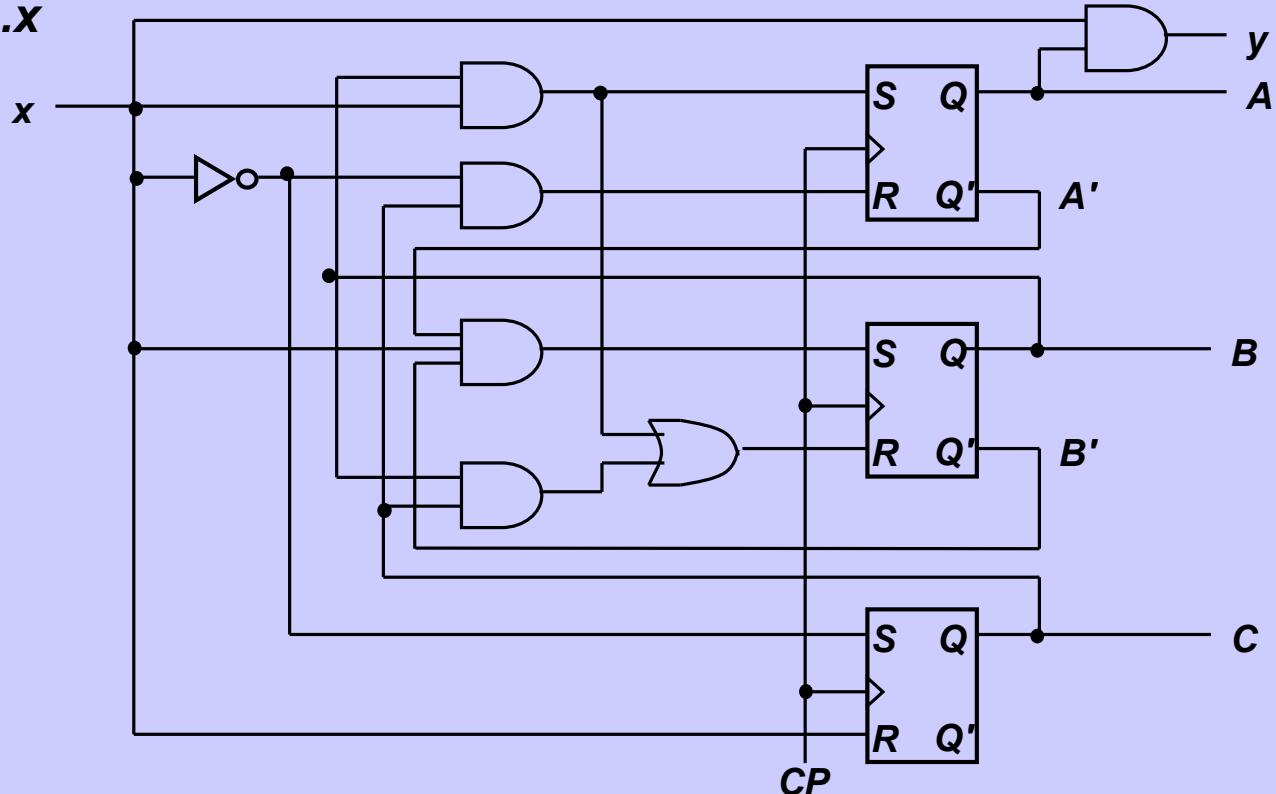
$$SB = A'B'x$$

$$RB = B.C + B.x'$$

$$SC = x'$$

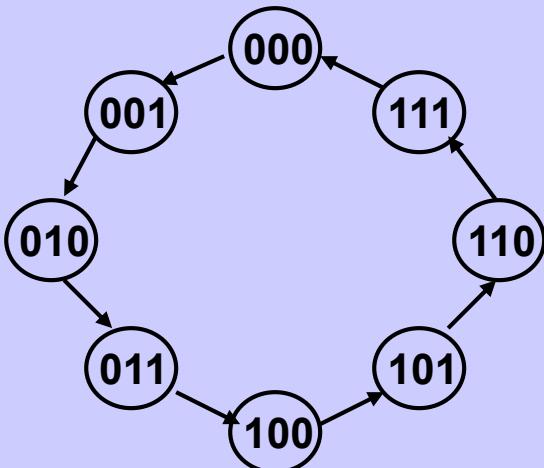
$$RC = x$$

$$y = A.x$$



# Design of Synchronous Counters

- Counter: a sequential circuit that cycles through a sequence of states.
- Binary counter: follows the *binary sequence*. An  $n$ -bit binary counter (with  $n$  flip-flops) counts from 0 to  $2^n-1$ .
- Example 1: A 3-bit binary counter (using T flip-flops).

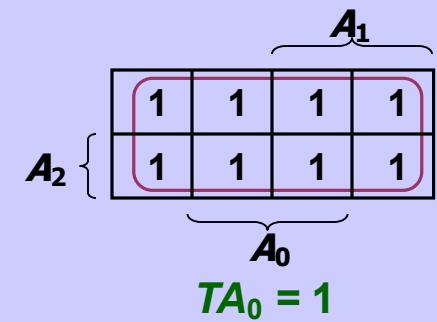
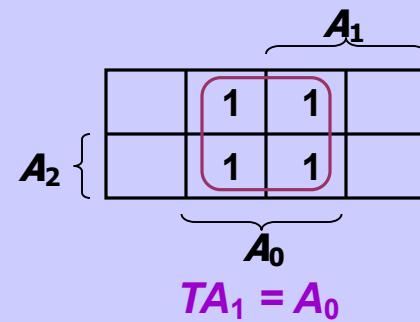
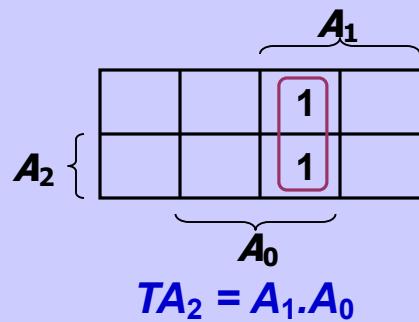


Present state			Next state			Flip-flop inputs		
$A_2$	$A_1$	$A_0$	$A_2^+$	$A_1^+$	$A_0^+$	$TA_2$	$TA_1$	$TA_0$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

# Design of Synchronous Counters

- 3-bit binary counter (cont'd).

Present state			Next state			Flip-flop inputs		
$A_2$	$A_1$	$A_0$	$A_2^+$	$A_1^+$	$A_0^+$	$TA_2$	$TA_1$	$TA_0$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



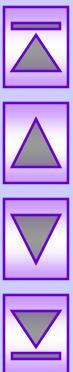
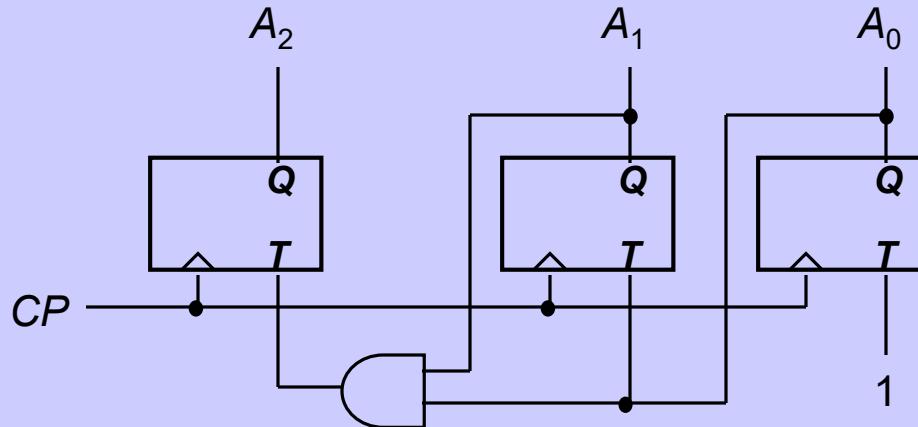
# Design of Synchronous Counters

- 3-bit binary counter (cont'd).

$$TA_2 = A_1 \cdot A_0$$

$$TA_1 = A_0$$

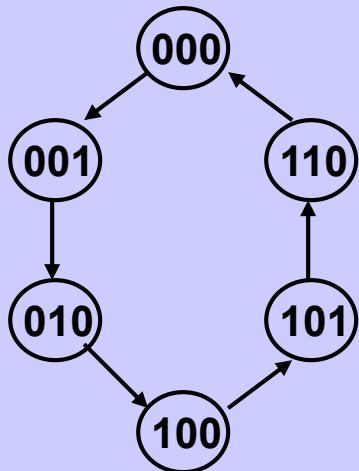
$$TA_0 = 1$$



# Design of Synchronous Counters

- Example 2: Counter with non-binary sequence:

000 → 001 → 010 → 100 → 101 → 110 and back to 000



Present state			Next state			Flip-flop inputs					
A	B	C	$A^+$	$B^+$	$C^+$	JA	KA	JB	KB	JC	KC
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

$$JA = B$$

$$KA = B$$

$$JB = C$$

$$KB = 1$$

$$JC = B'$$

$$KC = 1$$



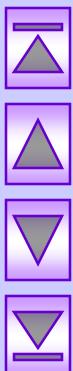
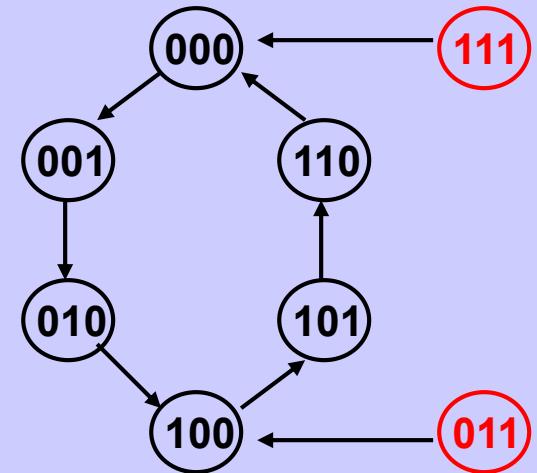
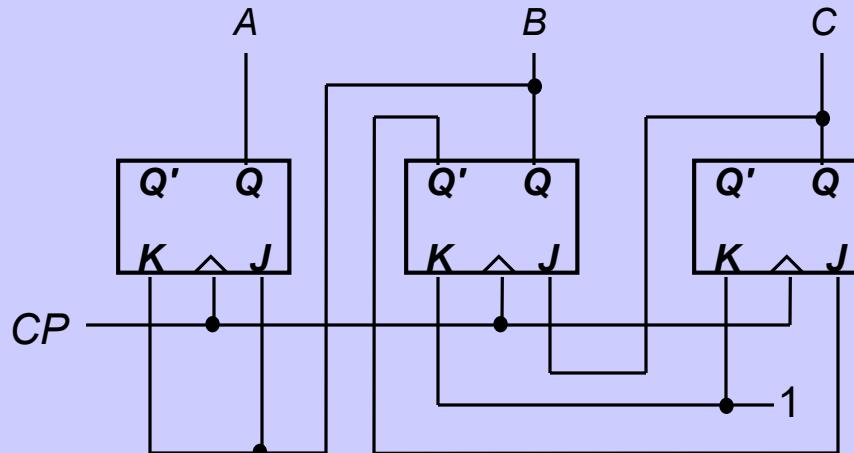
# Design of Synchronous Counters

- Counter with non-binary sequence (cont'd).

$$\begin{aligned} JA &= B \\ KA &= B \end{aligned}$$

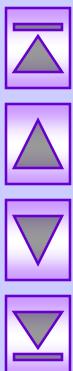
$$\begin{aligned} JB &= C \\ KB &= 1 \end{aligned}$$

$$\begin{aligned} JC &= B' \\ KC &= 1 \end{aligned}$$



# Summary

- Sequential circuits have memory and they are more powerful than combinational circuits.
- Analyzing sequential circuits
  - ❖ Flip-flop characteristic table
  - ❖ State Table
  - ❖ State diagram
- Designing sequential circuits
  - ❖ Flip-flop excitation table
  - ❖ State assignment
  - ❖ Circuit output function
  - ❖ Flip-flop input function



# ◆ Finite State Machine (FSM)

- This is what we have been waiting for in this class. Using combinational and sequential logics, now you can design a lot of clever digital logic circuits for functional products. We will learn different steps you take to go from word problems to logic circuits. We first talk about a simplified version of FSM which is a counter.

## ◆ Moore/Mealy machines

- There are two different ways to express the FSMs with respect to the output. Both have different advantages so it is good to know them.

# Finite state machines: more than counters

---

- ◆ FSM: A system that visits a finite number of logically distinct states
- ◆ Counters are simple FSMs
  - Outputs and states are identical
  - Visit states in a fixed sequence without inputs
- ◆ FSMs are typically more complex than counters
  - Outputs can depend on current state and on inputs
  - State sequencing depends on current state and on inputs

# FSM design

## ■ Counter-design procedure

- 1.State diagram
- 2.State-transition table
- 3.Next-state logic minimization
- 4.Implement the design

## ■ FSM-design procedure

- 1.State diagram
- 2.state-transition table
- 3.State minimization
- 4.State encoding
- 5.Next-state logic minimization
- 6.Implement the design

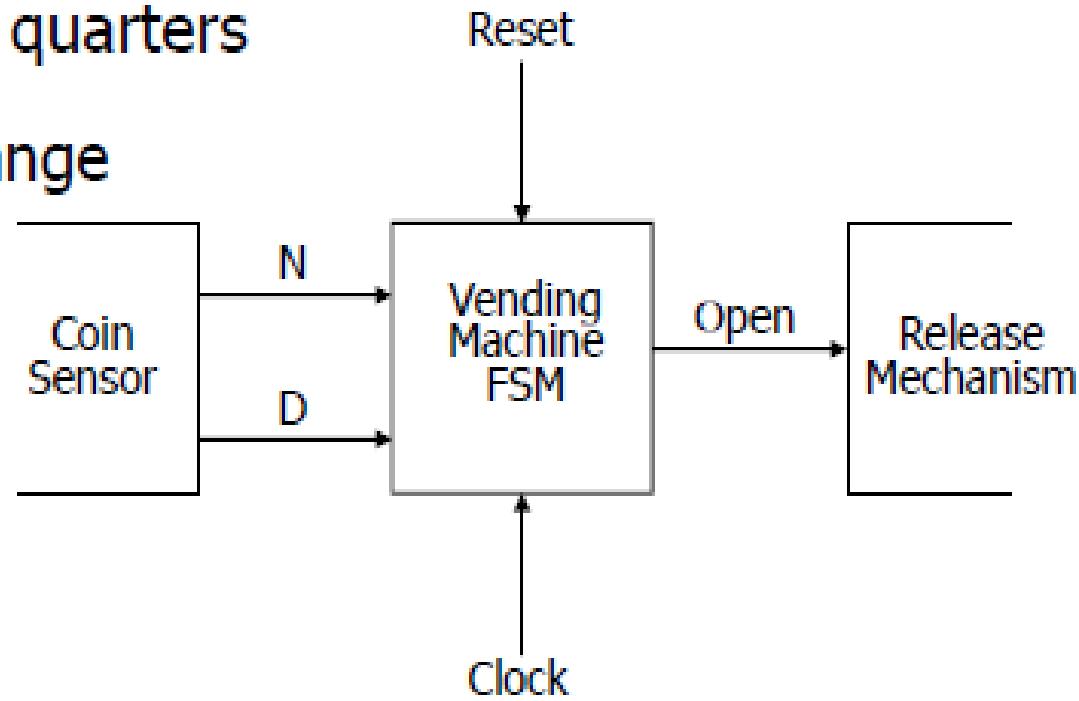
# Example: A vending machine

---

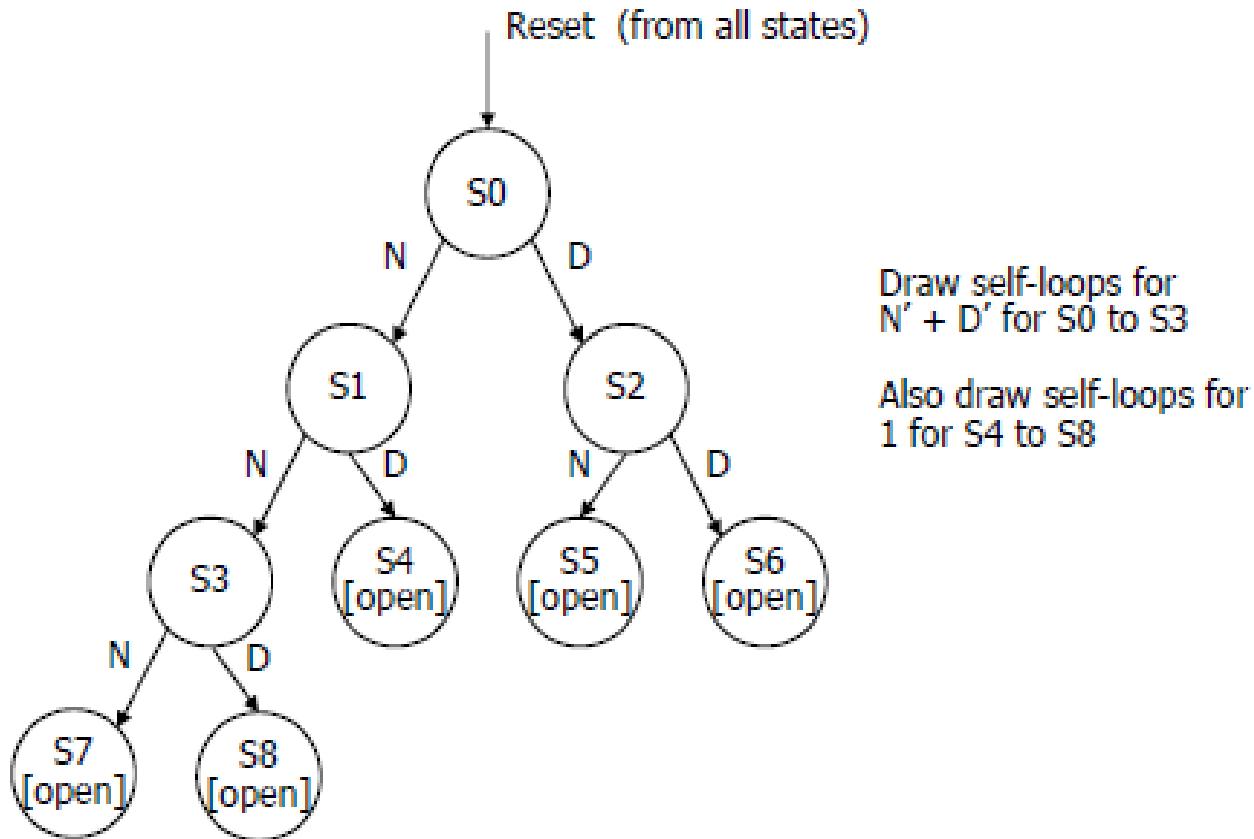
- ◆ 15 cents for a cup of coffee
- ◆ Doesn't take pennies or quarters
- ◆ Doesn't provide any change

## ■ FSM-design procedure

1. State diagram
2. state-transition table
3. State minimization
4. State encoding
5. Next-state logic minimization
6. Implement the design



# A vending machine: (conceptual) state diagram

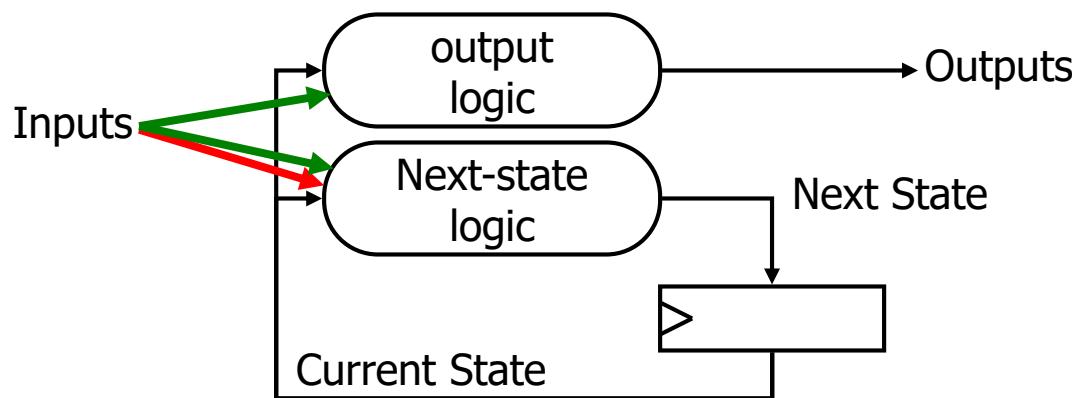


# A vending machine: State transition table

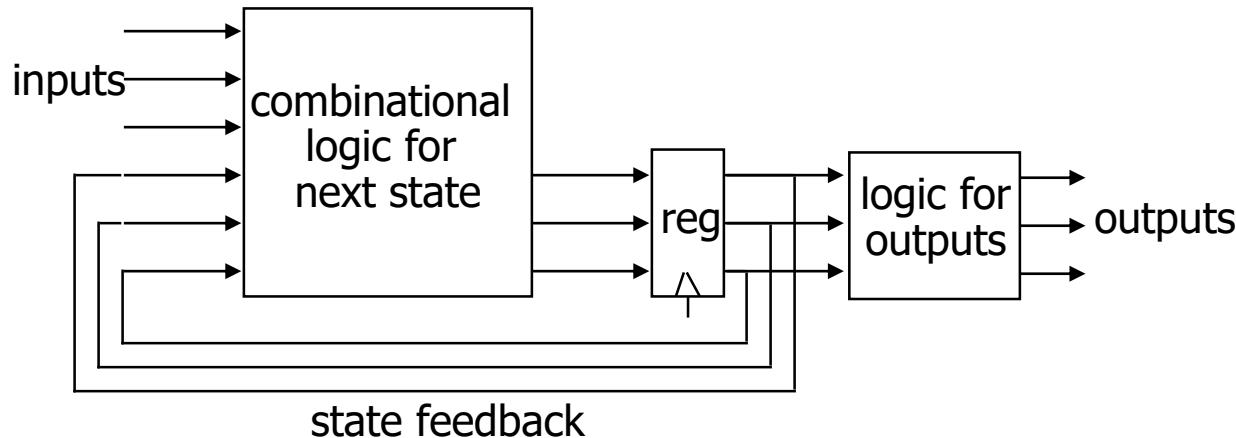
present state	inputs		next state	output open
	D	N		
S0	0	0	S0	0
	0	1	S1	0
	1	0	S2	0
	1	1	X	X
S1	0	0	S1	0
	0	1	S3	0
	1	0	S4	0
	1	1	X	X
S2	0	0	S2	0
	0	1	S5	0
	1	0	S6	0
	1	1	X	X
S3	0	0	S3	0
	0	1	S7	0
	1	0	S8	0
	1	1	X	X
S4	X	X	S4	1
S5	X	X	S5	1
S6	X	X	S6	1
S7	X	X	S7	1
S8	X	X	S8	1

# Generalized FSM model: Moore and Mealy

- ◆ Combinational logic computes next state and outputs
  - Next state is a function of current state and inputs
  - Outputs are functions of
    - ↳ Current state (**Moore** machine)
    - ↳ Current state and inputs (**Mealy** machine)



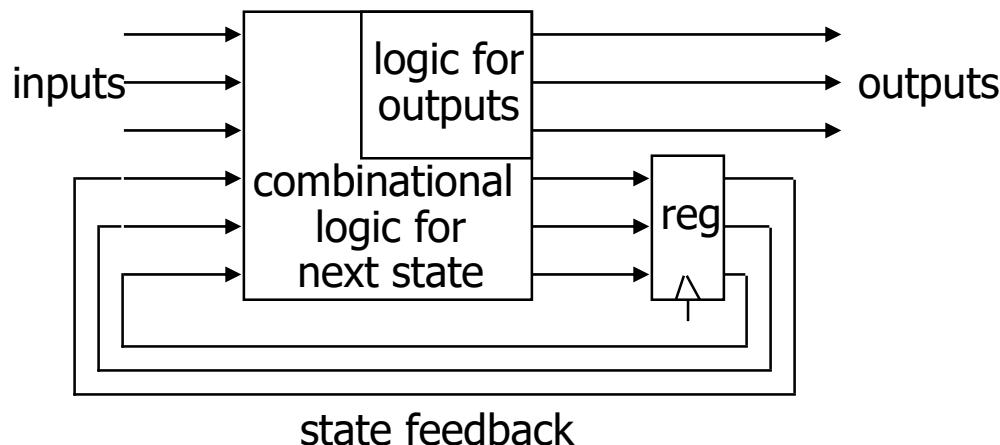
# Moore versus Mealy machines



## **Moore machine**

Outputs are a function of current state

Outputs change synchronously with state changes



## **Mealy machine**

Outputs depend on state and on inputs

Input changes can cause immediate output changes  
**(asynchronous)**

# Impacts start of the FSM design procedure

- Counter-design procedure
  1. State diagram
  2. State-transition table
  3. Next-state logic minimization
  4. Implement the design
- FSM-design procedure
  1. State diagram
  2. State-transition table
  3. State minimization
  4. State encoding
  5. Next-state logic minimization
  6. Implement the design

# State Diagrams

## ◆ Moore machine

- Each state is labeled by a pair:

state-name/output      or      state-name [output]

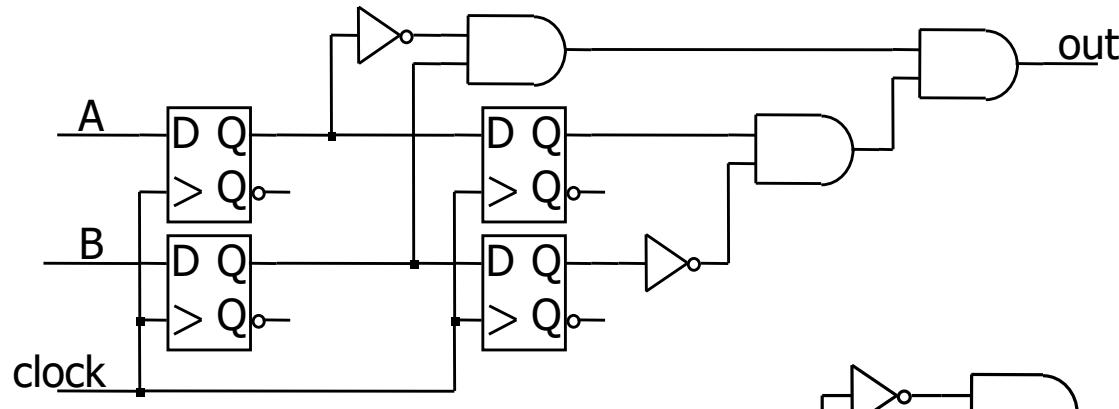
## ◆ Mealy machine

- Each transition arc is labeled by a pair:

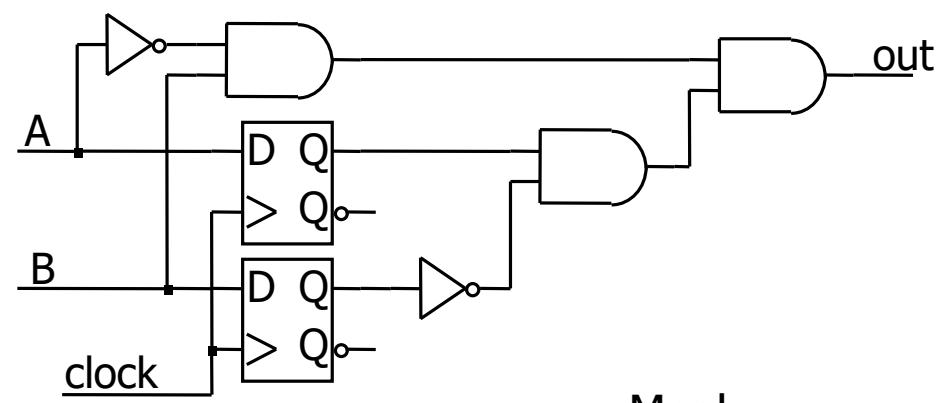
input-condition/output

# Example $10 \rightarrow 01$ : Moore or Mealy?

- ◆ Circuits recognize  $AB=10$  followed by  $AB=01$ 
  - What kinds of machines are they?



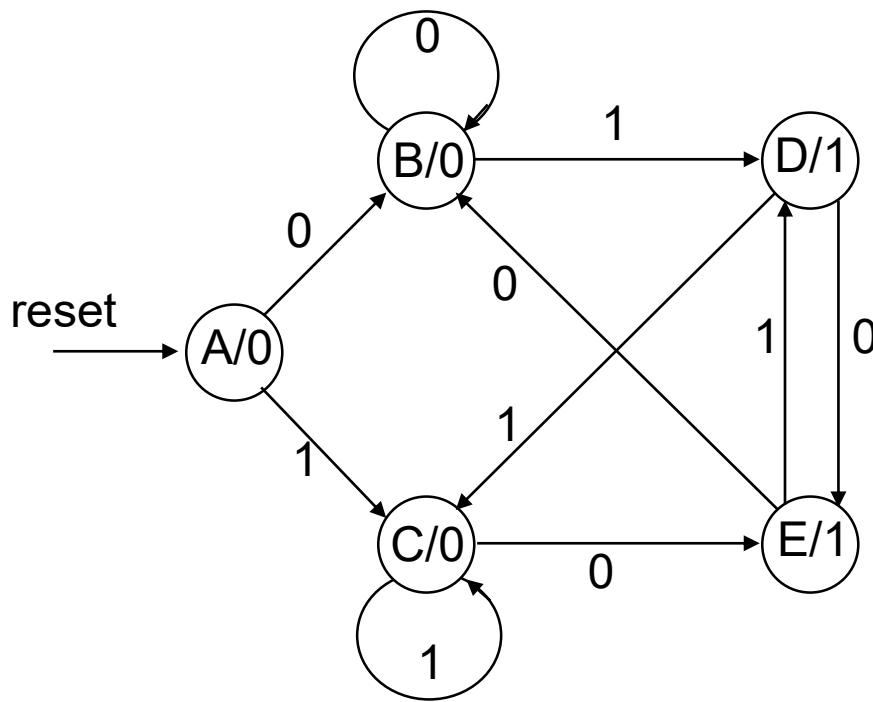
Moore



Mealy

# Example “01 or 10” detector: a Moore machine

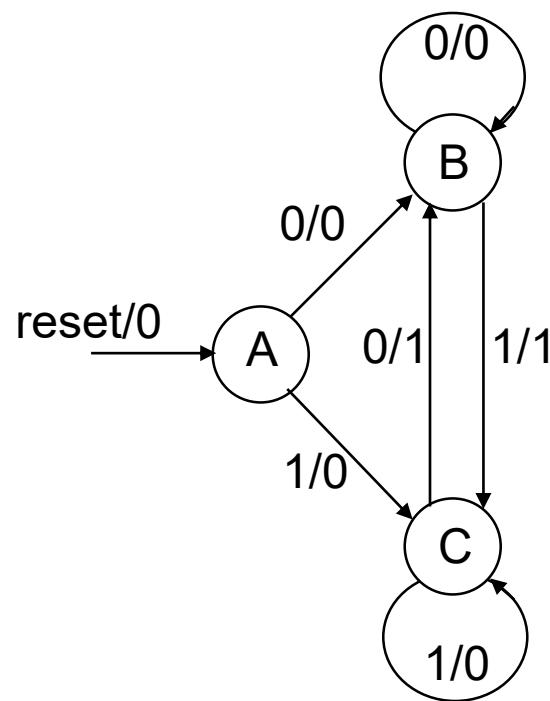
- ◆ Output is a function of state only
  - Specify output in the state bubble



reset	input	current state	next state	current output
1	-	-	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

# Example “01 or 10” detector: a Mealy machine

- ◆ Output is a function of state and inputs
  - Specify outputs on transition arcs



reset	input	current state	next state	current output
1	-	-	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

# Comparing Moore and Mealy machines

## ◆ Moore machines

- + Safer to use because outputs change at clock edge
- May take additional logic to decode state into outputs

## ◆ Mealy machines

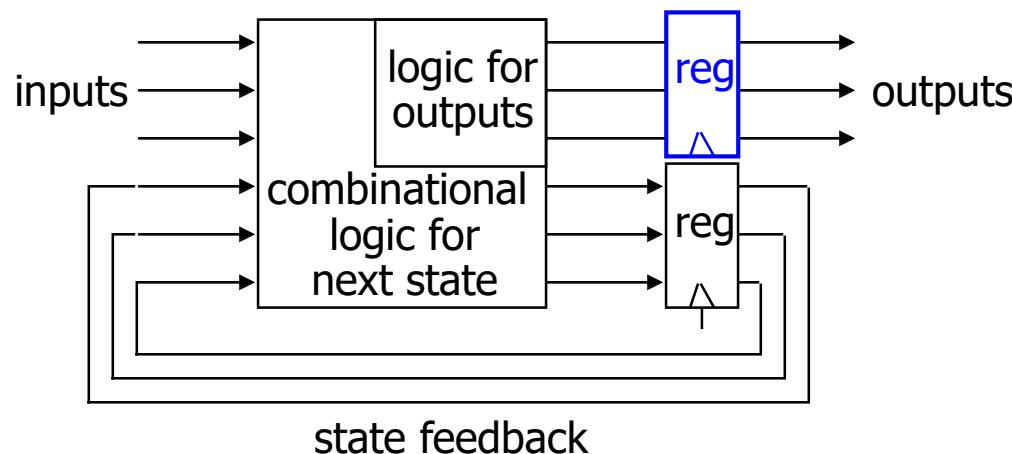
- + Typically have fewer states
- + React faster to inputs — don't wait for clock
- Asynchronous outputs can be dangerous

## ◆ We often design synchronous Mealy machines

- Design a Mealy machine
- Then register the outputs

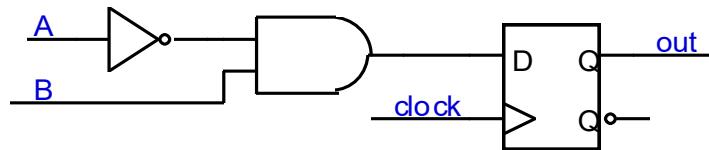
# Synchronous (registered) Mealy machine

- ◆ Registered state and registered outputs
  - No glitches on outputs
  - No race conditions between communicating machines

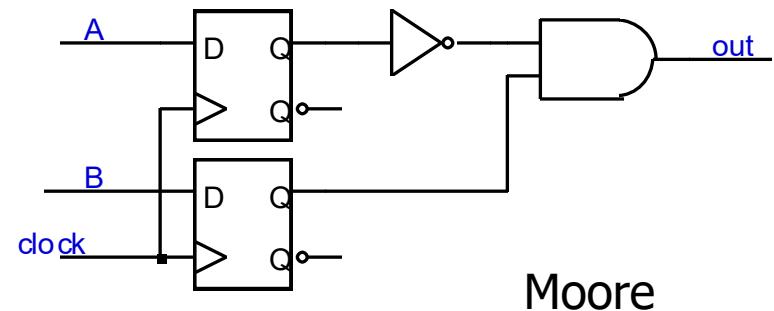


# Example “=01”: Moore or Mealy?

- ◆ Recognize  $AB = 01$ 
  - Mealy or Moore?



Registered Mealy  
(actually Moore)



Moore

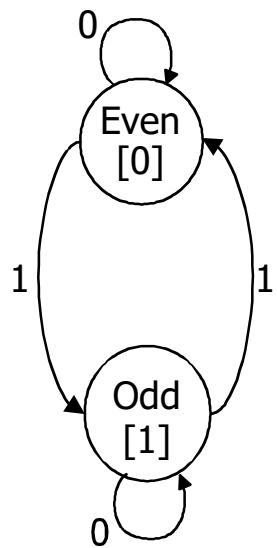
# Example: A parity checker

- ◆ Serial input string
  - OUT=1 if odd # of 1s in input
  - OUT=0 if even # of 1s in input
- ◆ Let's do this for Moore and Mealy

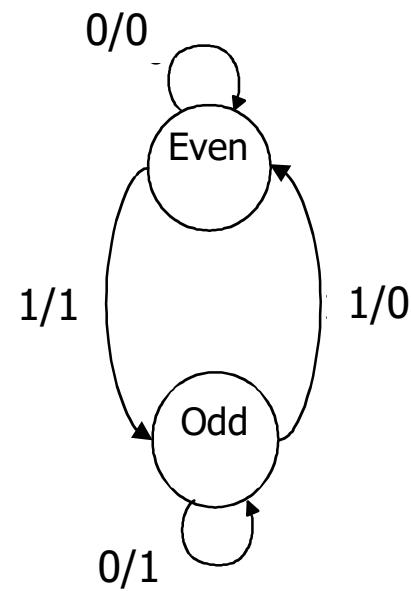
# Example: A parity checker

## 1. State diagram

Moore



Mealy



# Example: A parity checker

## 1. State-transition table

Moore

Present State	Input	Next State	Present Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Mealy

Present State	Input	Next State	Present Output
Even	0	Even	0
Even	1	Odd	1
Odd	0	Odd	1
Odd	1	Even	0

# Example: A parity checker

## 3. State minimization: Already minimized

- Need both states (even and odd)
- Use one flip-flop

# Example: A parity checker

## 4. State encoding

Moore

Assignment  
Even 0  
Odd 1

Present State	Input	Next State	Present Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Mealy

Present State	Input	Next State	Present Output
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

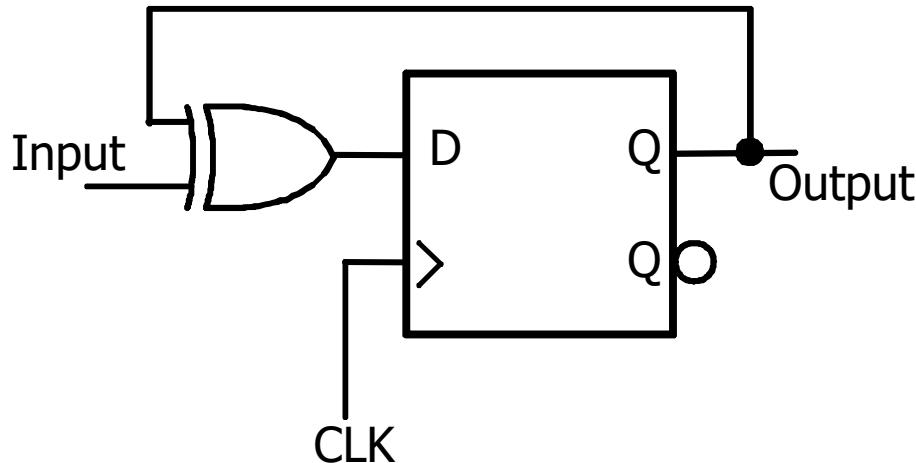
# Example: A parity checker

## 5. Next-state logic minimization

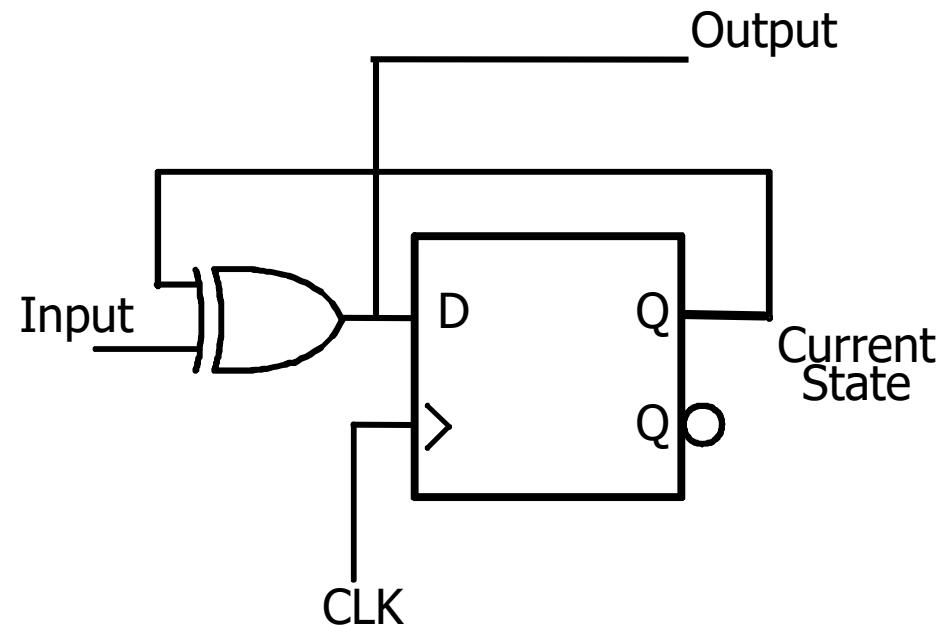
- Assume D flip-flops
- Next state = (present state) XOR (present input)

## 6. Implement the design

Moore



Mealy



# What was covered so far on FSM

- ◆ Finite state machines
  - FSM design procedure
    1. State diagram
    2. State-transition table
    3. State minimization
    4. State encoding
    5. Next-state logic minimization
    6. Implement the design
  - No Mealy machines

# CSE1003-Digital Logic Design

## Module:5 Sequential Circuits-I

**Dr.Penchalaiah Palla**

Dept. of Micro and Nanoelectronics  
School of Electronics Engineering,VIT,  
Vellore

# **Module:5 Sequential Circuits-I**

<b>Module:5</b>	<b>SEQUENTIAL CIRCUITS – I</b>	<b>6 hours</b>
Flip Flops - Sequential Circuit: Design and Analysis - Finite State Machine: Moore and Mealy model - Sequence Detector.		
<b>Module:6</b>	<b>SEQUENTIAL CIRCUITS – II</b>	<b>7 hours</b>
Registers - Shift Registers - Counters - Ripple and Synchronous Counters - Modulo counters - Ring and Johnson counters		

# Introduction: Sequential Circuits

## ◆ Combinational

- The outputs depend only on the current input values
- It uses only logic gates

## ◆ Sequential

- The outputs depend on the current and past input values
- It uses logic gates and storage elements
- Example
  - ✓ Vending machine
- They are referred as finite state machines since they have a finite number of states

# Block Diagram

- ◆ **Memory elements can store binary information**
  - **This information at any given time determines the state of the circuit at that time**

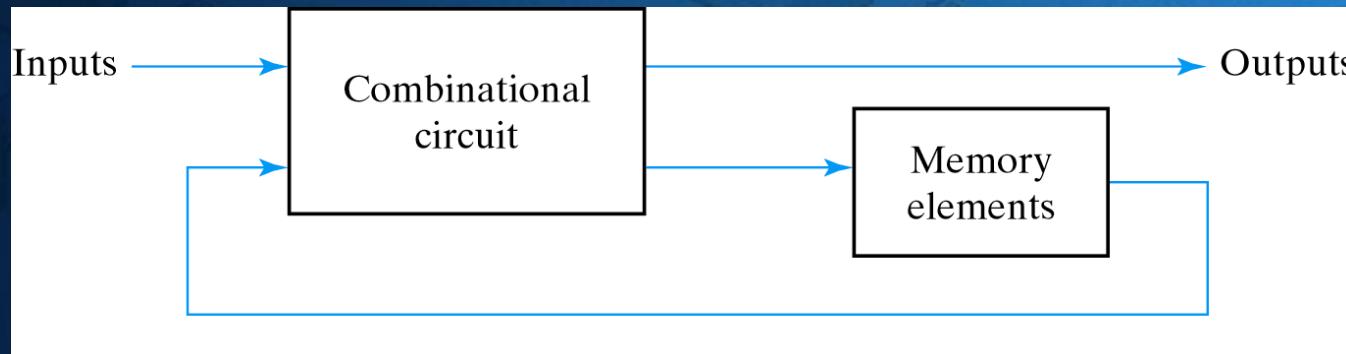
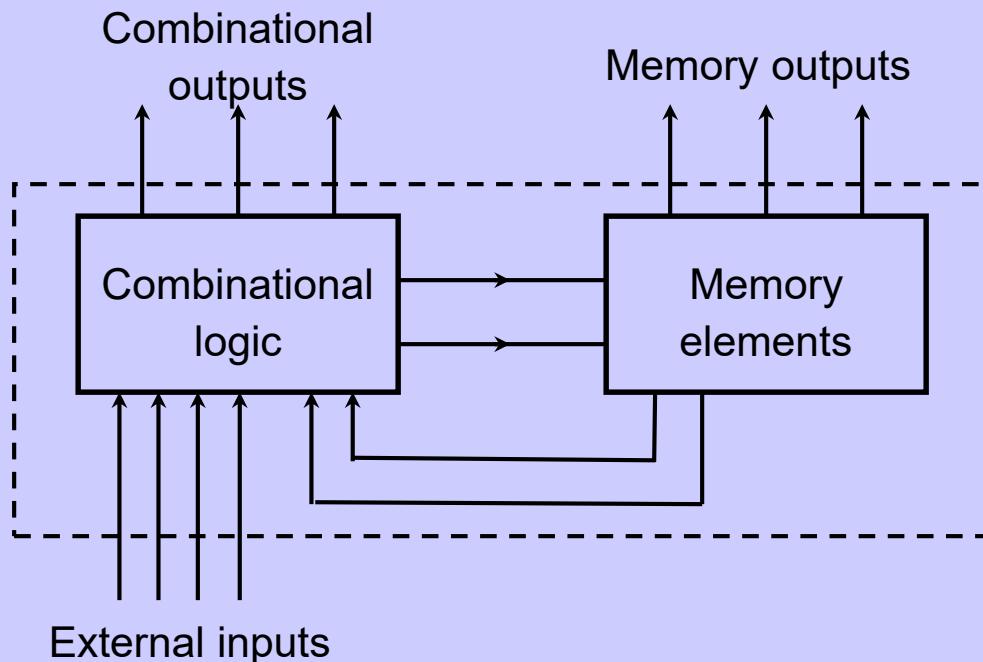


Fig. 5-1 Block Diagram of Sequential Circuit

- A **sequential circuit** consists of a *feedback path*, and employs some *memory elements*.



Sequential circuit = Combinational logic + Memory Elements

# Sequential Circuit Types

## ◆ Synchronous

- **The circuit behavior is determined by the signals at discrete instants of time**
- **The memory elements are affected only at discrete instants of time**
- **A clock is used for synchronization**
  - ✓ **Memory elements are affected only with the arrival of a clock pulse**
  - ✓ **If memory elements use clock pulses in their inputs, the circuit is called**
    - **Clocked sequential circuit**

# Sequential Circuit Types

## ◆ ASynchronous

- The circuit behavior is determined by the signals at any instant of time
- It is also affected by the order the inputs change

# IN short.....

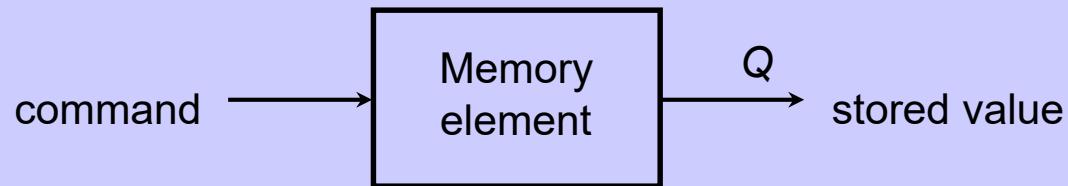
- There are two types of sequential circuits:
  - ❖ *synchronous*: outputs change only at specific time
  - ❖ *asynchronous*: outputs change at any time
- *Multivibrator*: a class of sequential circuits. They can be:
  - ❖ *bistable* (2 stable states)
  - ❖ *monostable* or *one-shot* (1 stable state)
  - ❖ *astable* (no stable state)
- Bistable logic devices: *latches* and *flip-flops*.
- Latches and flip-flops differ in the method used for changing their state.

# Latches & Flip-flops

- Memory Elements
- Pulse-Triggered Latch
  - ❖ S-R Latch
  - ❖ Gated S-R Latch
  - ❖ Gated D Latch
- Edge-Triggered Flip-flops
  - ❖ S-R Flip-flop
  - ❖ D Flip-flop
  - ❖ J-K Flip-flop
  - ❖ T Flip-flop
- Asynchronous Inputs

# Memory Elements

- **Memory element:** a device which can remember value indefinitely, or change value on command from its inputs.



- Characteristic table:

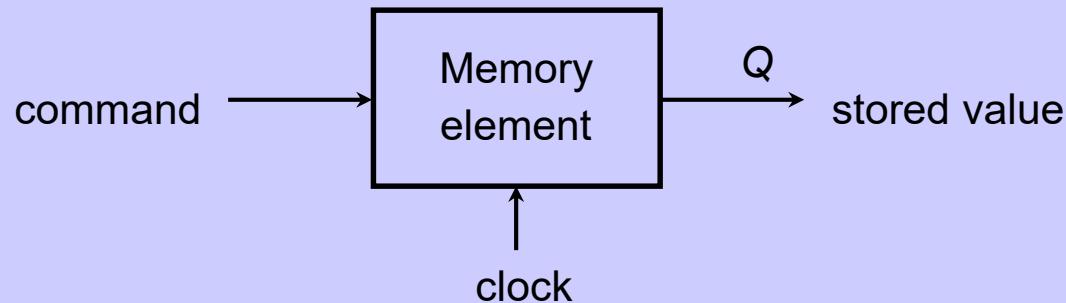
Command (at time $t$ )	$Q(t)$	$Q(t+1)$
Set	X	1
Reset	X	0
Memorise / No Change	0	0
	1	1

$Q(t)$ : current state

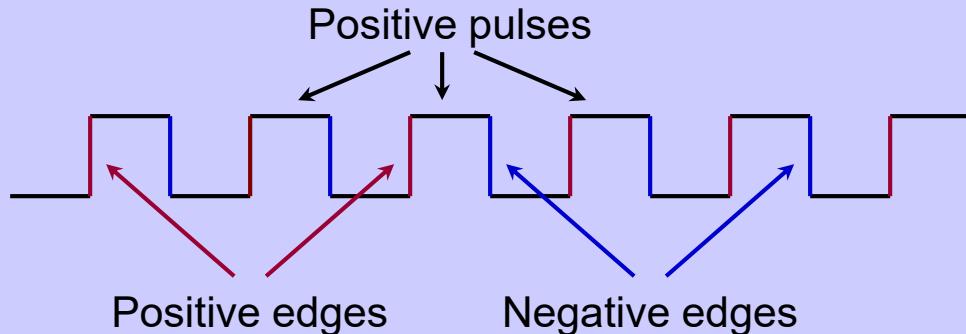
$Q(t+1)$  or  $Q^+$ : next state

# Memory Elements

- Memory element with clock. Flip-flops are memory elements that change state on clock signals.

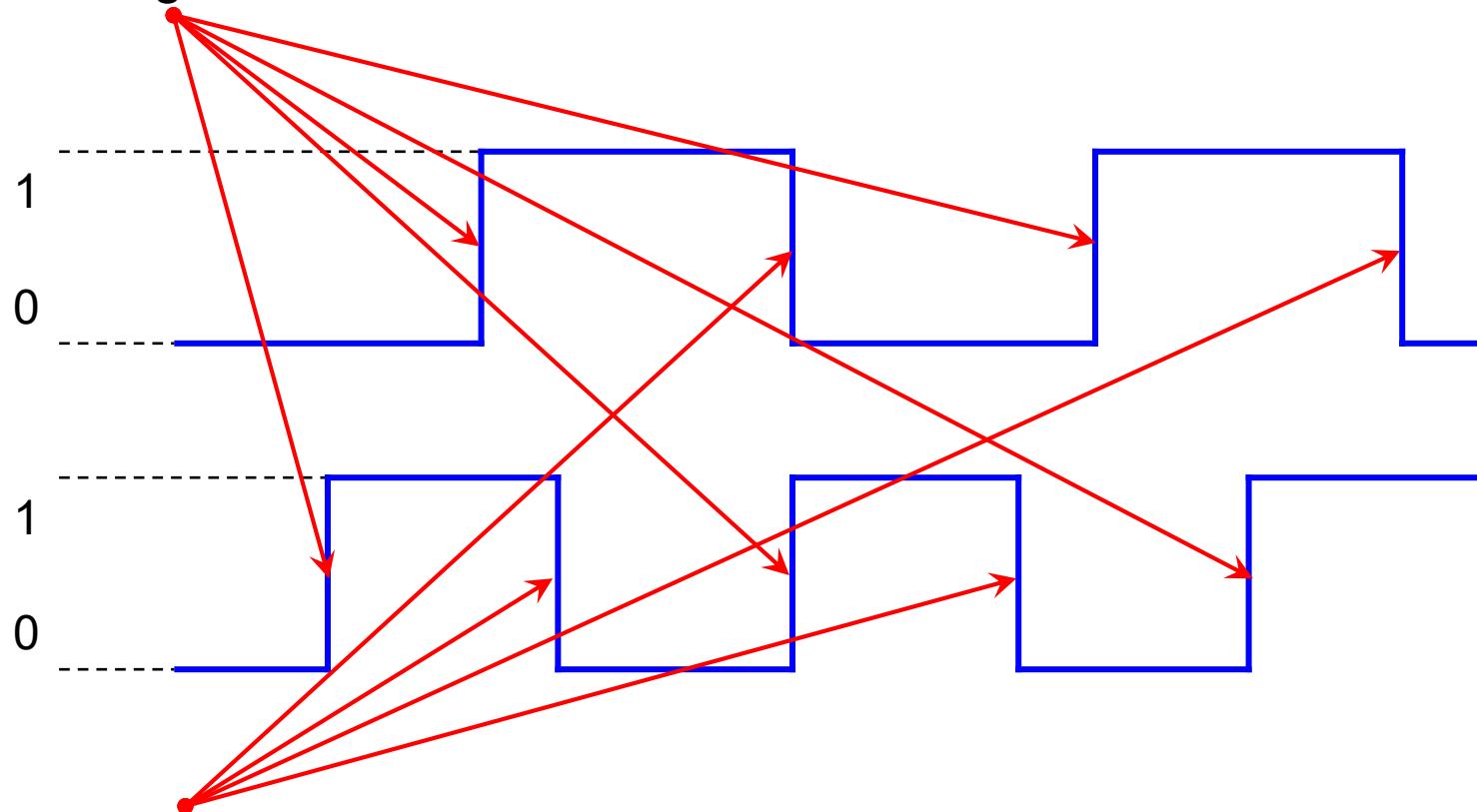


- Clock is usually a square wave.



# Clock Edges

Positive Edge Transition



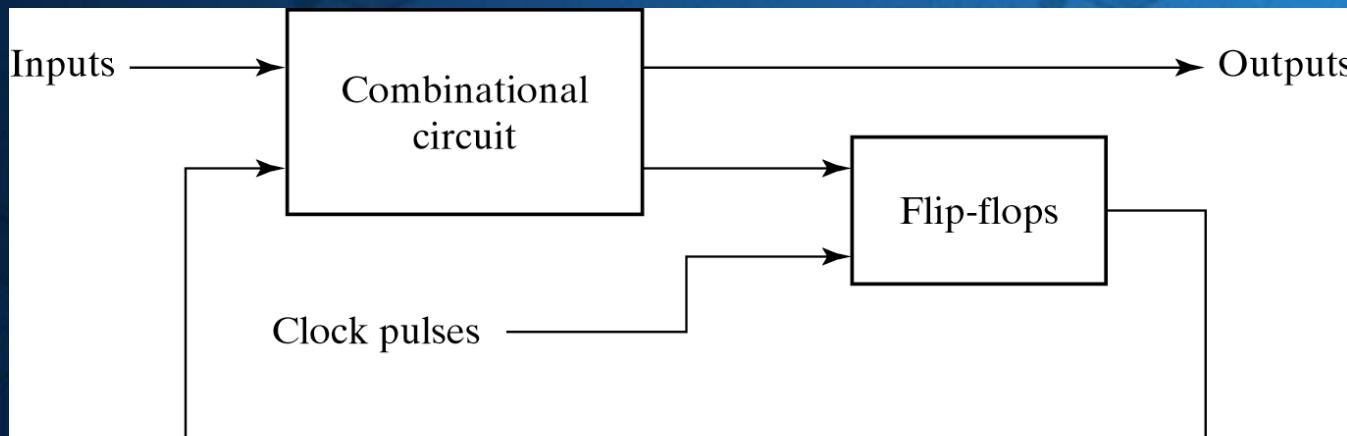
Negative Edge Transition

# Memory Elements

- Two types of triggering/activation:
  - ❖ pulse-triggered
  - ❖ edge-triggered
- Pulse-triggered
  - ❖ latches
  - ❖ ON = 1, OFF = 0
- Edge-triggered
  - ❖ flip-flops
  - ❖ positive edge-triggered (ON = from 0 to 1; OFF = other time)
  - ❖ negative edge-triggered (ON = from 1 to 0; OFF = other time)

# Flip-Flops

- ◆ They are memory elements
- ◆ They can store binary information



(a) Block diagram



(b) Timing diagram of clock pulses

# Clock

- ◆ It emits a series of pulses with a precise pulse width and precise interval between consecutive pulses
- ◆ Timing interval between the corresponding edges of two consecutive pulses is known as the clock cycle time, or period

# Flip-Flops

- ◆ Can keep a binary state until an input signal to switch the state is received
- ◆ There are different types of flip-flops depending on the number of inputs and how the inputs affect the binary state

# Latches

- ◆ The most basic flip-flops
  - They operate with signal levels
- ◆ The flip-flops are constructed from latches
- ◆ They are not useful for **synchronous** sequential circuits
- ◆ They are useful for **asynchronous** sequential circuits

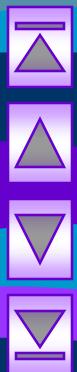
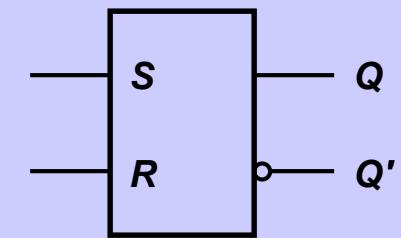
# S-R Latch

- Complementary outputs: Q and Q'.
- When Q is HIGH, the latch is in **SET** state.
- When Q is LOW, the latch is in **RESET** state.
- For *active-HIGH input S-R latch* (also known as NOR gate latch),
  - $R=\text{HIGH}$  (and  $S=\text{LOW}$ )  $\Rightarrow$  RESET state
  - $S=\text{HIGH}$  (and  $R=\text{LOW}$ )  $\Rightarrow$  SET state
  - both inputs LOW  $\Rightarrow$  no change
  - both inputs HIGH  $\Rightarrow$  Q and Q' both LOW (invalid)!

# S-R Latch

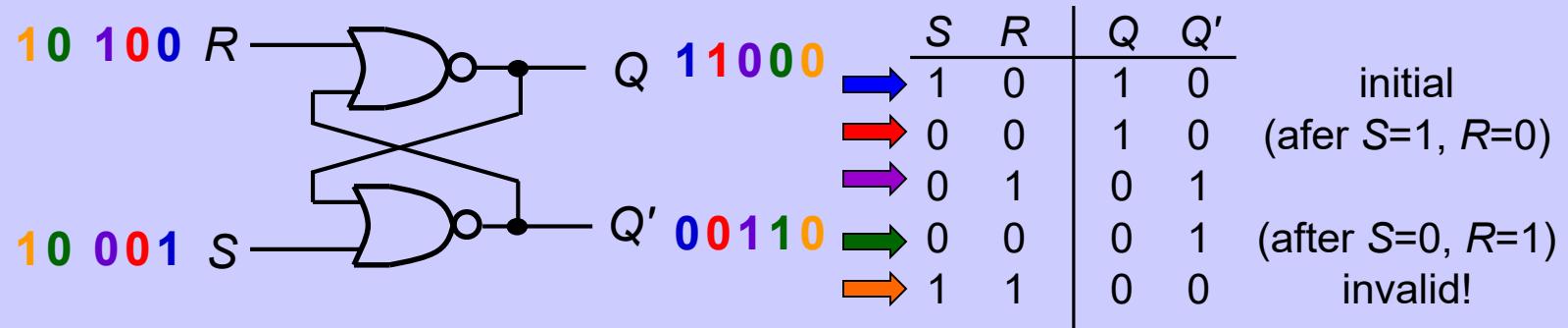
- Characteristics table for active-high input S-R latch:

S	R	Q	Q'	
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.



# S-R Latch with NOR

- Active-HIGH input S-R latch



# SR Latch with NOR

$S = set$

$R = reset$

$Q = 1, \quad Q' = 0 \Rightarrow \text{set state}$

$Q = 0, \quad Q' = 1 \Rightarrow \text{reset state}$

$S = 1, \quad R = 1 \Rightarrow \text{undefined, } Q \text{ and } Q' \text{ are set to 0}$

In normal conditions, avoid  $S = 1, R = 1$

# SR Latch with NAND

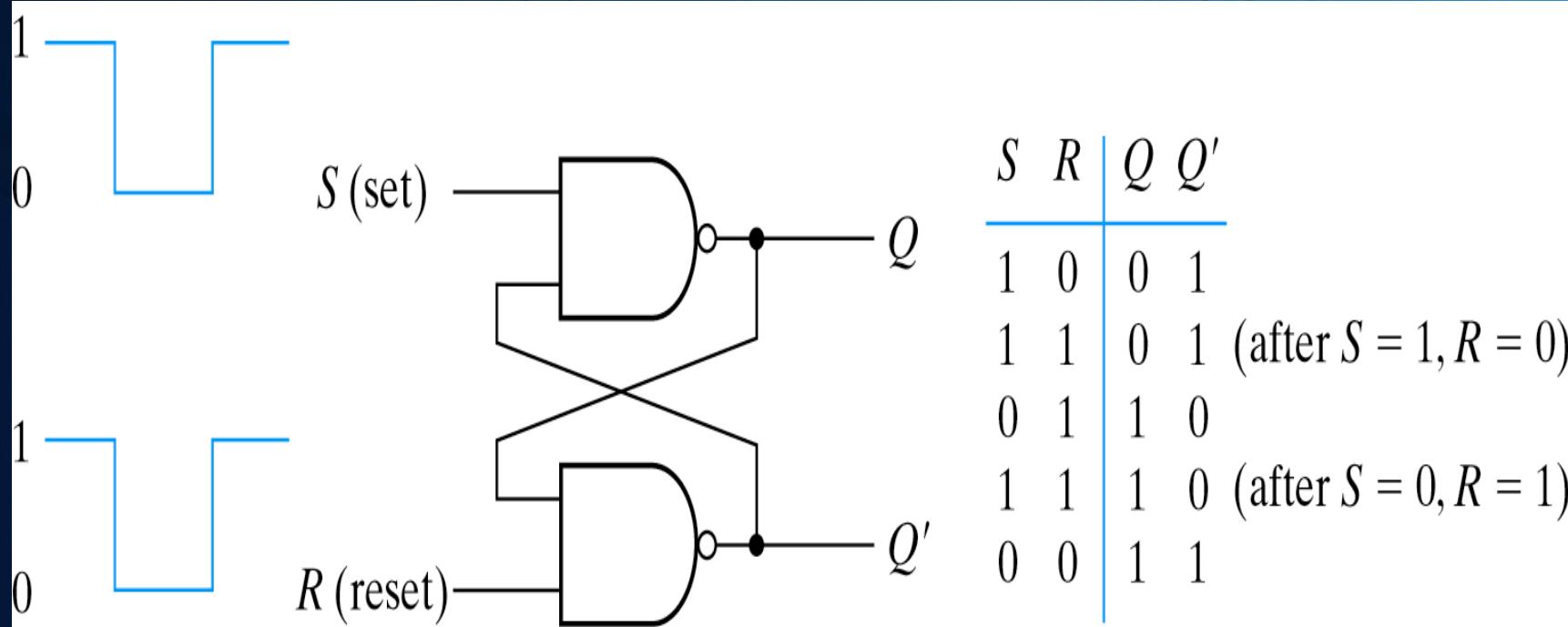


Fig. 5-4 SR Latch with NAND Gates

# SR Latch with NAND

$S = set$

$R = reset$

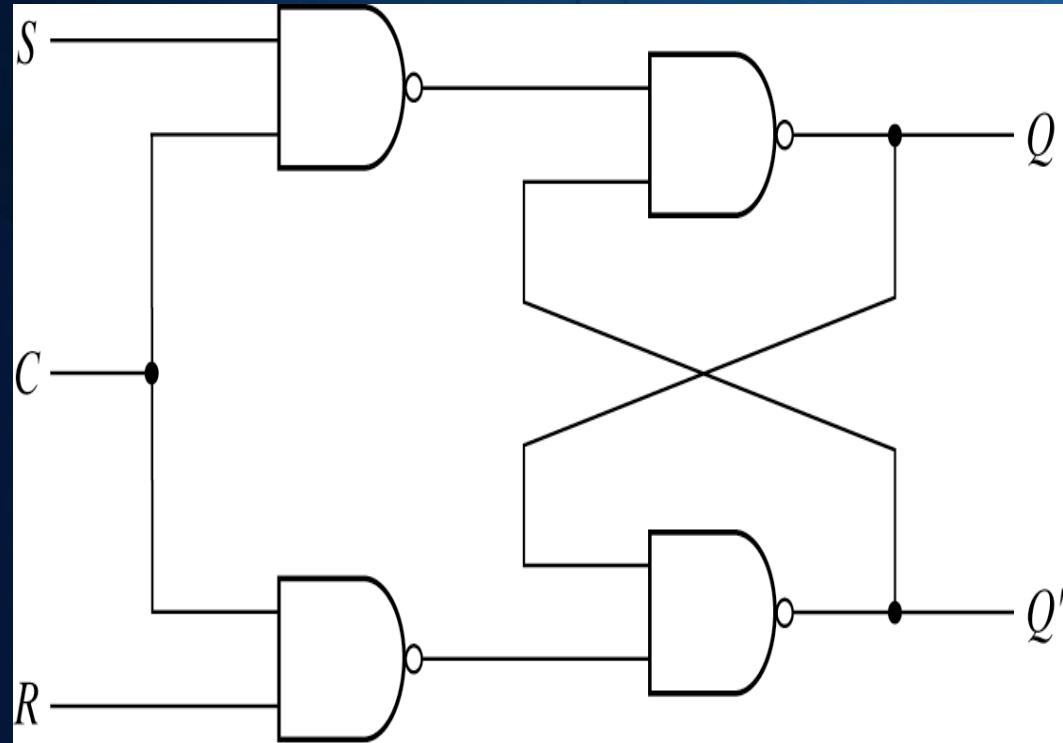
$Q = 0, \quad Q' = 1 \Rightarrow \text{set state}$

$Q = 1, \quad Q' = 0 \Rightarrow \text{reset state}$

$S = 0, \quad R = 0 \Rightarrow \text{undefined, } Q \text{ and } Q' \text{ are set to 1}$

In normal conditions, avoid  $S = 0, R = 0$

# SR Latch with Control Input



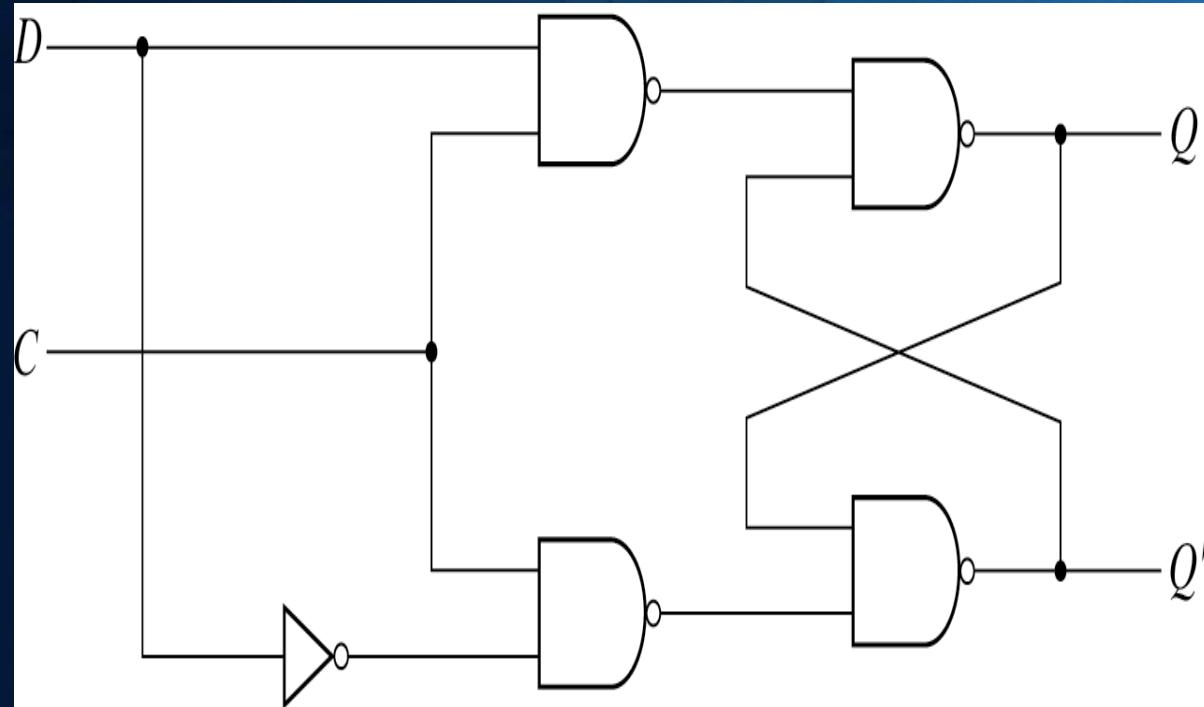
(a) Logic diagram

C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$ ; Reset state
1	1	0	$Q = 1$ ; set state
1	1	1	Indeterminate

(b) Function table

Fig. 5-5 SR Latch with Control Input

# D Latch



(a) Logic diagram

(b) Function table

Fig. 5-6 D Latch

$C$	$D$	Next state of $Q$
0	X	No change
1	0	$Q = 0$ ; Reset state
1	1	$Q = 1$ ; Set state

# Symbols for Latches

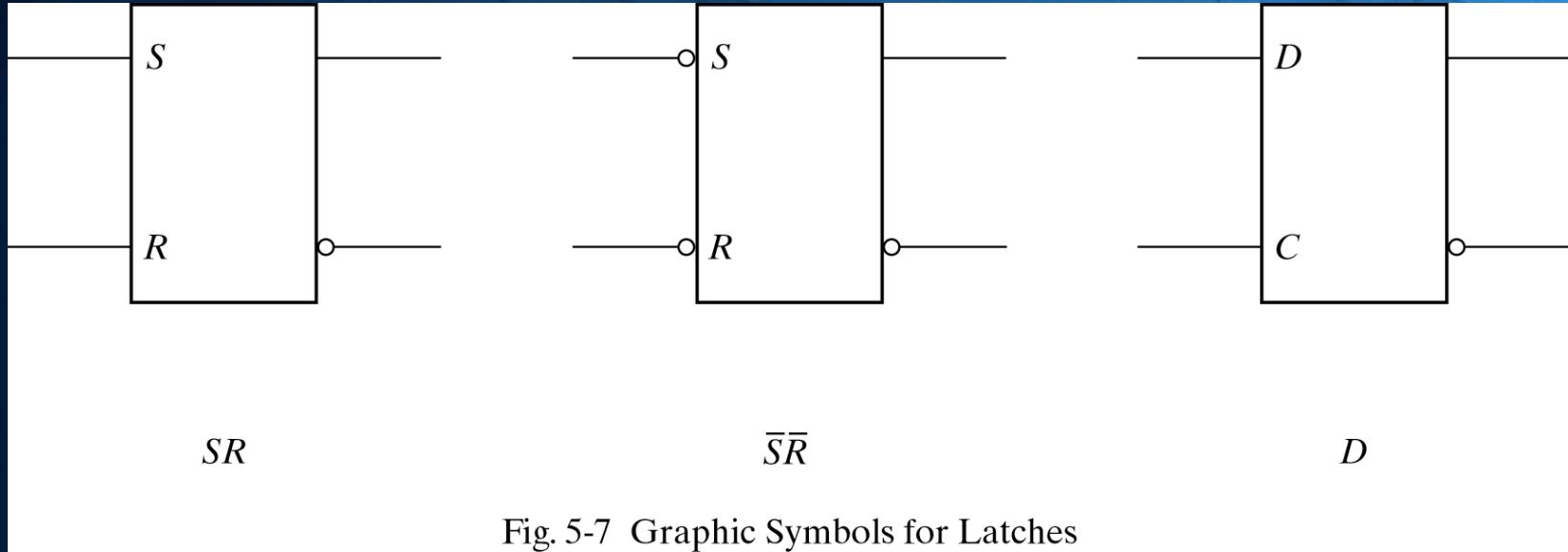


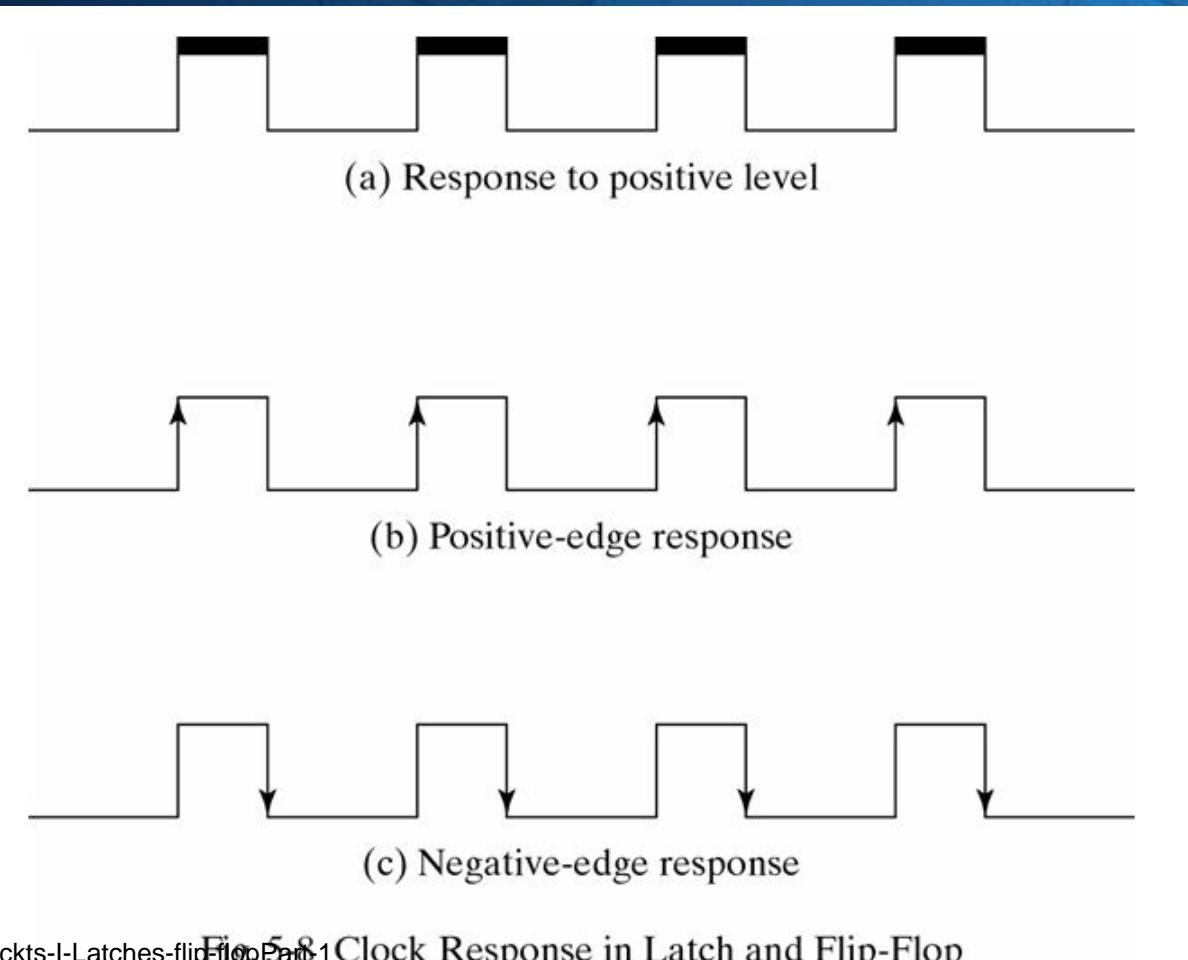
Fig. 5-7 Graphic Symbols for Latches

# Note

- ◆ The control input changes the state of a latch or flip-flop
- ◆ The momentary change is called a trigger
- ◆ Example: D Latch
  - It is triggered every time the pulse goes to the logic level 1
  - As long as the pulse remains at the logic level 1, the change in the data (D) directly affects the output (Q)
  - THIS MAY BE A BIG PROBLEM since the state of the latch may keep changing depending on the input (may be coming from a combinational logic network)

# How to Solve?

- ◆ Trigger the flip-flop only during a signal transition



Mod-5-\_Sequential\_ckt5-1-Latches-flipFlopPart1  
Fig 5.8 Clock Response in Latch and Flip-Flop

# Edge-Triggered D Flip-Flop

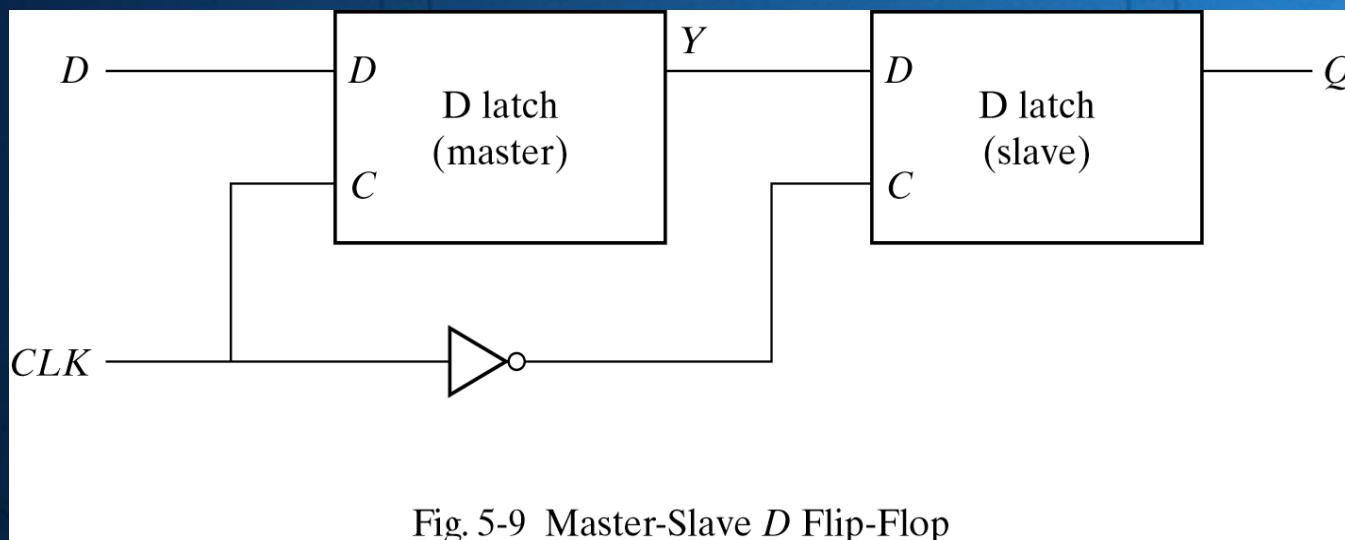
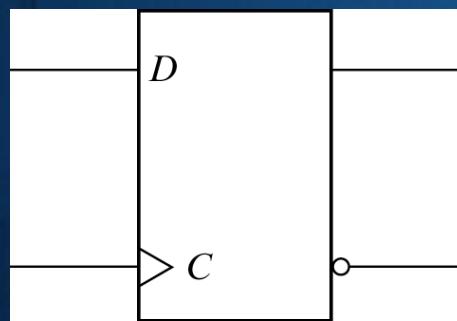
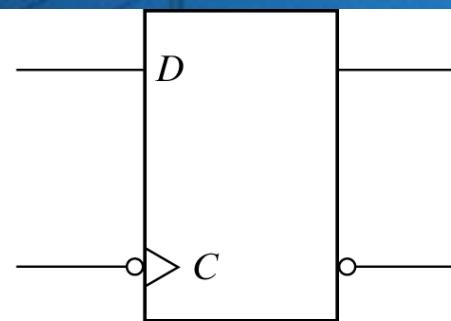


Fig. 5-9 Master-Slave *D* Flip-Flop



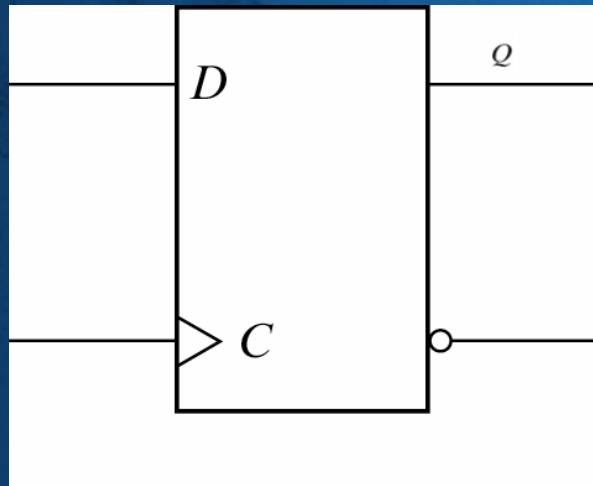
(a) Positive-edge



(a) Negative-edge

Fig. 5-11 Graphic Symbol for Edge-Triggered *D* Flip-Flop

# Characteristics of D Flip-Flop

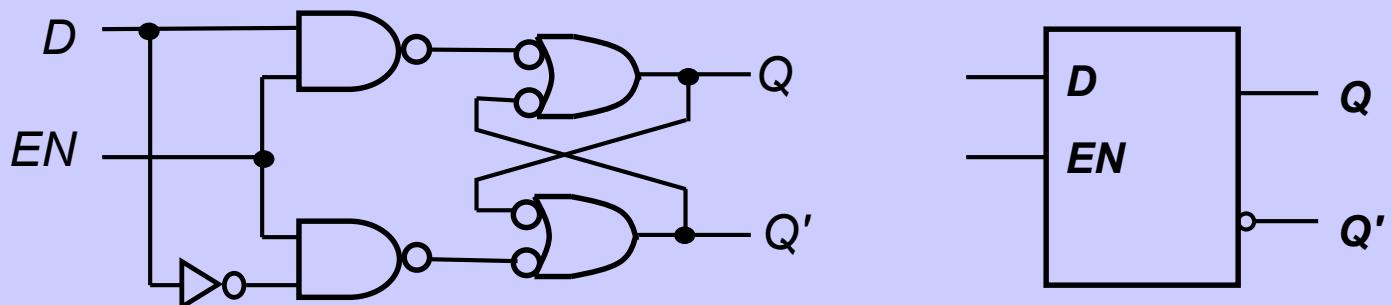


Inputs		Outputs		Comments
D	C	Q	Q'	
0	↑	0	1	RESET
1	↑	1	0	SET

$$Q(t+1) = D$$

# Gated D Latch

- Make  $R$  input equal to  $S'$   $\rightarrow$  *gated D latch*.
- $D$  latch eliminates the undesirable condition of invalid state in the  $S-R$  latch.



# Gated D Latch

- When  $EN$  is HIGH,
  - ❖  $D=HIGH \rightarrow$  latch is SET
  - ❖  $D=LOW \rightarrow$  latch is RESET
- Hence when  $EN$  is HIGH,  $Q$  ‘follows’ the  $D$  (data) input.
- Characteristic table:

$EN$	$D$	$Q(t+1)$	
1	0	0	Reset
1	1	1	Set
0	X	$Q(t)$	No change

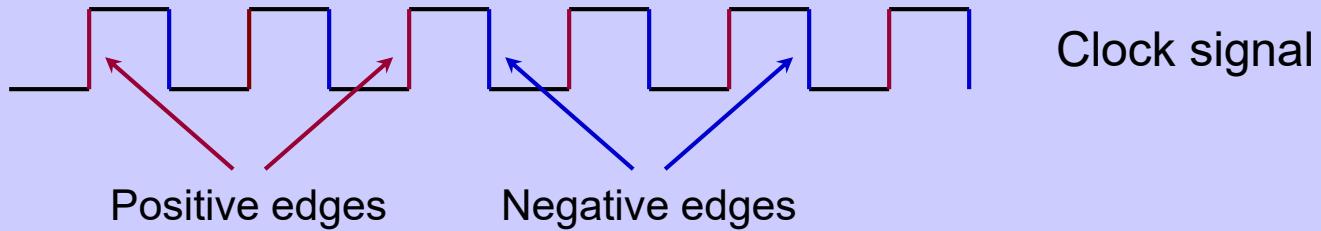
When  $EN=1$ ,  $Q(t+1) = D$

# Latch Circuits: Not Suitable

- Latch circuits are not suitable in synchronous logic circuits.
- When the enable signal is active, the excitation inputs are gated directly to the output Q. Thus, any change in the excitation input immediately causes a change in the latch output.
- The problem is solved by using a special timing control signal called a *clock* to restrict the times at which the states of the memory elements may change.
- This leads us to the edge-triggered memory elements called *flip-flops*.

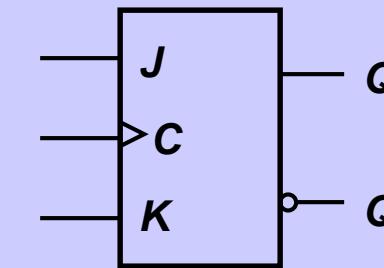
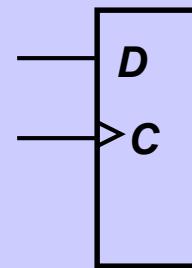
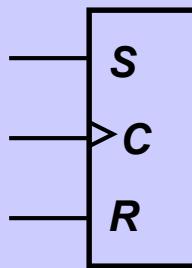
# Edge-Triggered Flip-flops

- *Flip-flops*: synchronous bistable devices
- Output changes state at a specified point on a triggering input called the *clock*.
- Change state either at the *positive edge* (rising edge) or at the *negative edge* (*falling edge*) of the clock signal.

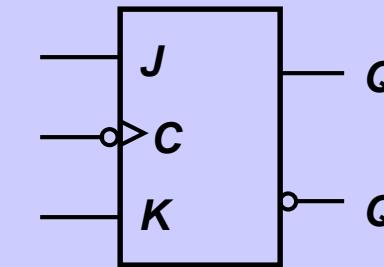
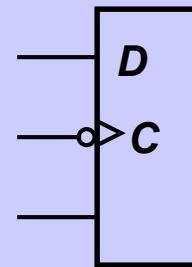
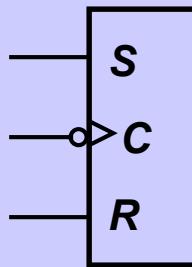


# Edge-Triggered Flip-flops

- S-R, D and J-K edge-triggered flip-flops. Note the “>” symbol at the clock input.



Positive edge-triggered flip-flops



Negative edge-triggered flip-flops

# S-R Flip-flop

- S-R flip-flop: on the triggering edge of the clock pulse,
  - ❖  $S=HIGH$  (and  $R=LOW$ )  $\Rightarrow$  SET state
  - ❖  $R=HIGH$  (and  $S=LOW$ )  $\Rightarrow$  RESET state
  - ❖ both inputs LOW  $\Rightarrow$  no change
  - ❖ both inputs HIGH  $\Rightarrow$  invalid
- Characteristic table of positive edge-triggered S-R flip-flop:

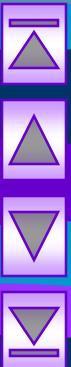
<b>S</b>	<b>R</b>	<b>CLK</b>	<b><math>Q(t+1)</math></b>	<b>Comments</b>
0	0	X	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

**X = irrelevant (“don’t care”)**

**↑ = clock transition LOW to HIGH**

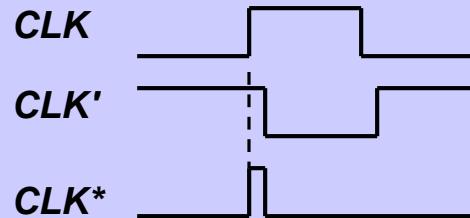
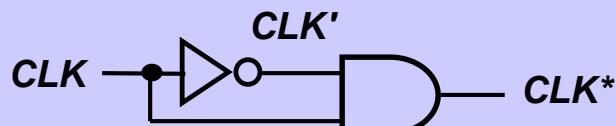
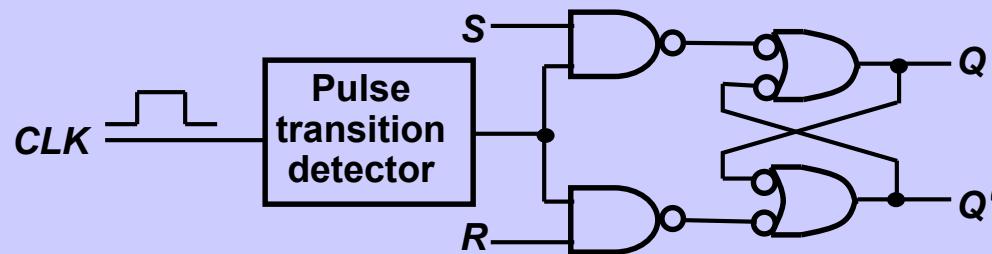
# S-R Flip-flop

- It comprises 3 parts:
  - ❖ a basic *NAND latch*
  - ❖ a *pulse-steering* circuit
  - ❖ a *pulse transition detector* (or *edge detector*) circuit
- The **pulse transition detector** detects a rising (or falling) edge and produces a very *short-duration spike*.

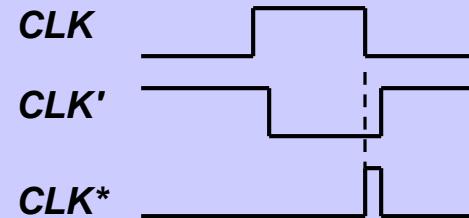
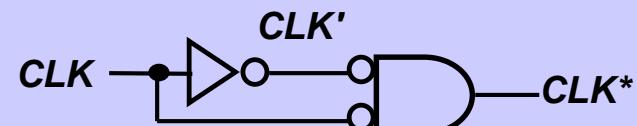


# S-R Flip-flop

The pulse transition detector.



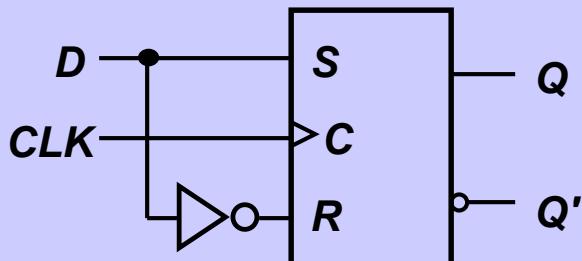
Positive-going transition  
(rising edge)



Negative-going transition  
(falling edge)

# Edge-Triggered D Flip-Flop

- D flip-flop: single input  $D$  (data)
  - ❖  $D=\text{HIGH} \Rightarrow \text{SET state}$
  - ❖  $D=\text{LOW} \Rightarrow \text{RESET state}$
- Q follows  $D$  at the clock edge.
- Convert S-R flip-flop into a D flip-flop: add an inverter.



$D$	$CLK$	$Q(t+1)$	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

A positive edge-triggered D flip-flop formed with an S-R flip-flop.

# Edge-Triggered D Flip-Flop

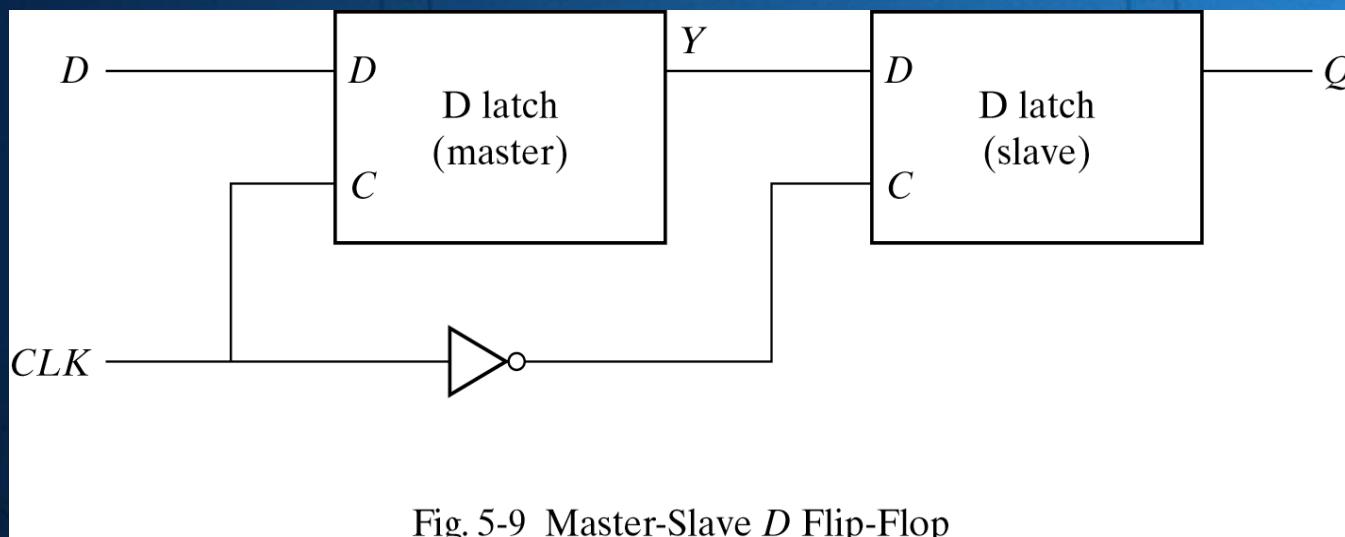
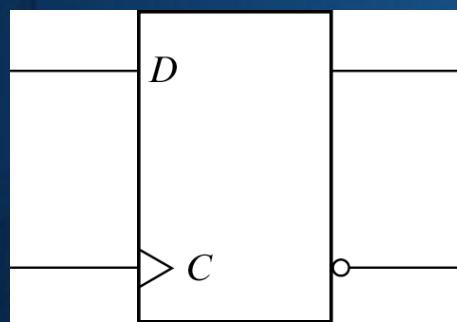
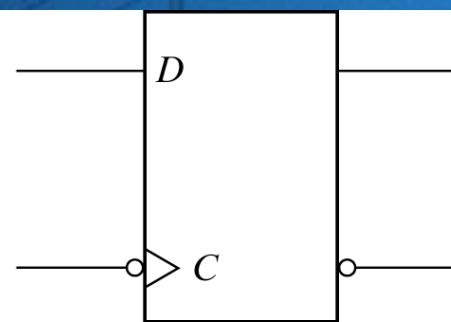


Fig. 5-9 Master-Slave D Flip-Flop



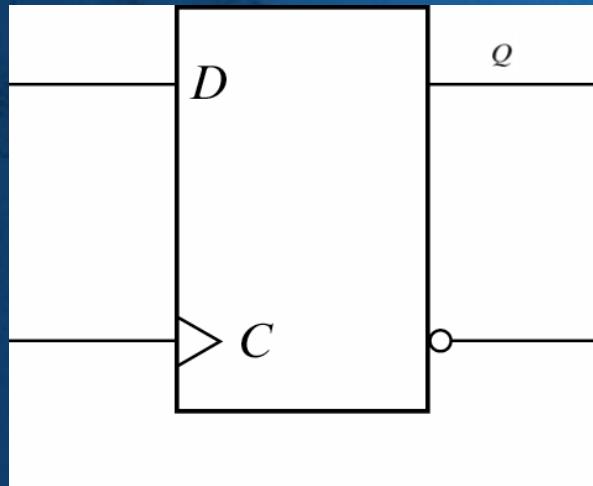
(a) Positive-edge



(a) Negative-edge

Fig. 5-11 Graphic Symbol for Edge-Triggered D Flip-Flop

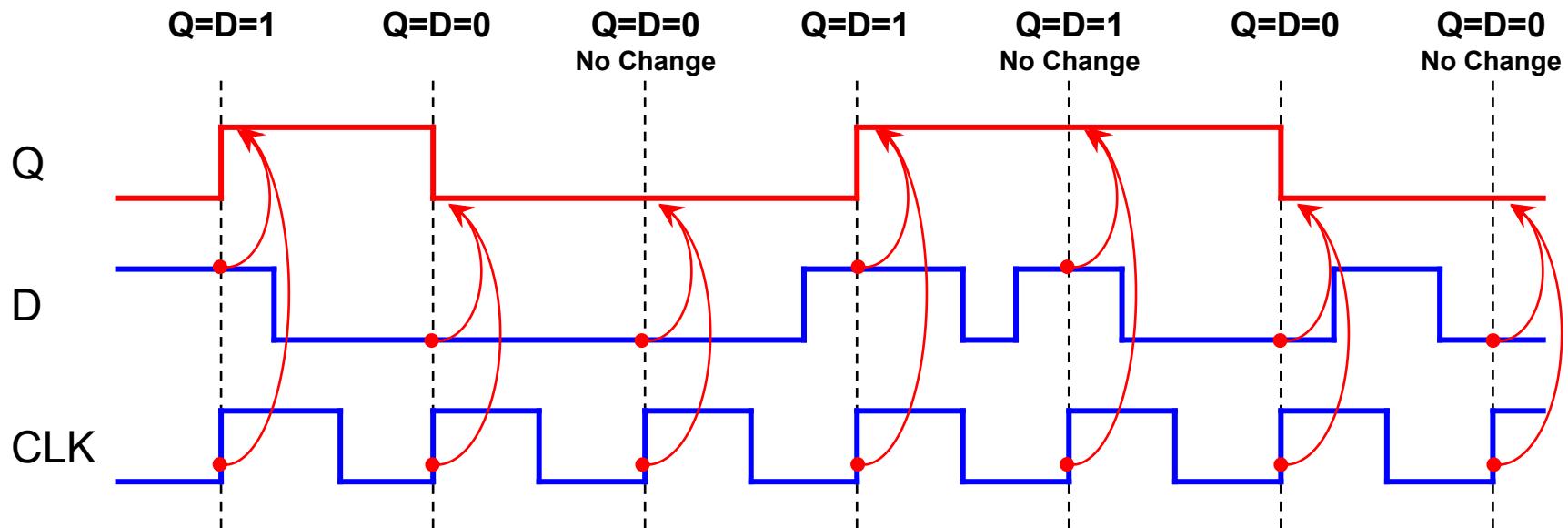
# Characteristics of D Flip-Flop



Inputs		Outputs		Comments
D	C	Q	Q'	
0	↑	0	1	RESET
1	↑	1	0	SET

$$Q(t+1) = D$$

# D Flip-Flop: Example Timing



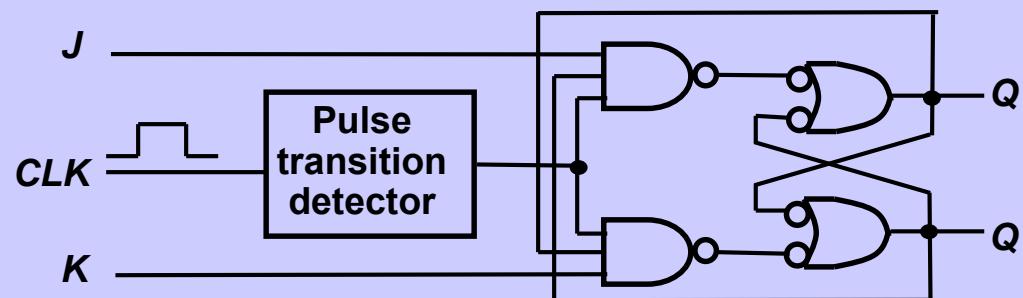
# Edge-Triggered J-K Flip-Flop

- J-K flip-flop: Q and Q' are fed back to the pulse-steering NAND gates.
- No invalid state.
- Include a *toggle* state.
  - ❖  $J=HIGH$  (and  $K=LOW$ )  $\Rightarrow$  SET state
  - ❖  $K=HIGH$  (and  $J=LOW$ )  $\Rightarrow$  RESET state
  - ❖ both inputs LOW  $\Rightarrow$  no change
  - ❖ both inputs HIGH  $\Rightarrow$  toggle



# Edge-Triggered J-K Flip-Flop

- J-K flip-flop.



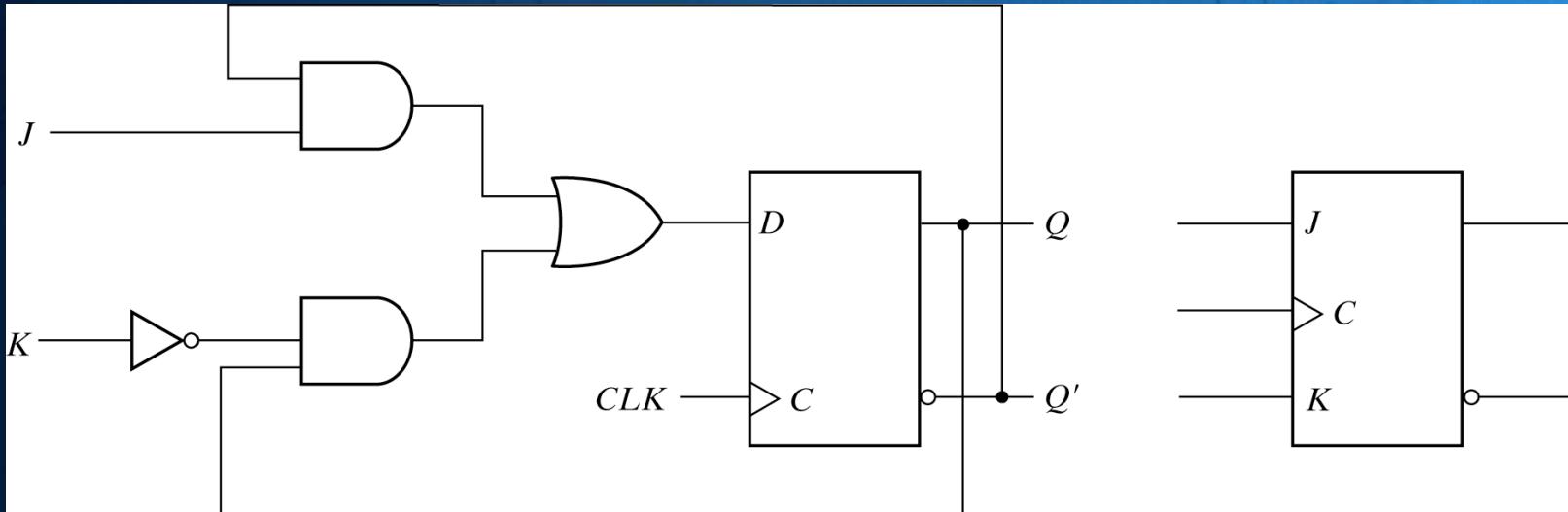
- Characteristic table.

<i>J</i>	<i>K</i>	<i>CLK</i>	<i>Q(t+1)</i>	Comments
0	0	↑	<i>Q(t)</i>	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	<i>Q(t)'</i>	Toggle

$$Q(t+1) = J \cdot Q' + K' \cdot Q$$

<i>Q</i>	<i>J</i>	<i>K</i>	<i>Q(t+1)</i>
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

# Edge-Triggered J-K Flip-Flop



(a) Circuit diagram

(b) Graphic symbol

Fig. 5-12 JK Flip-Flop

Inputs			Outputs		Comments
J	K	C	Q	Q'	
0	0	↑	Q	Q'	No change
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	Q'	Q	Toggle

$$Q(t+1) = JQ' + K'Q$$

How???????

# Excitation Table

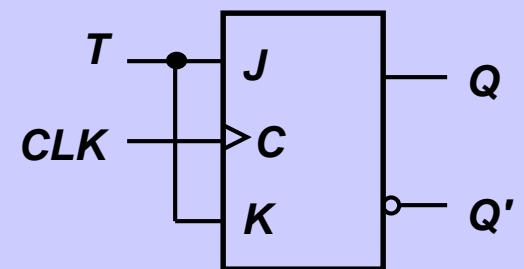
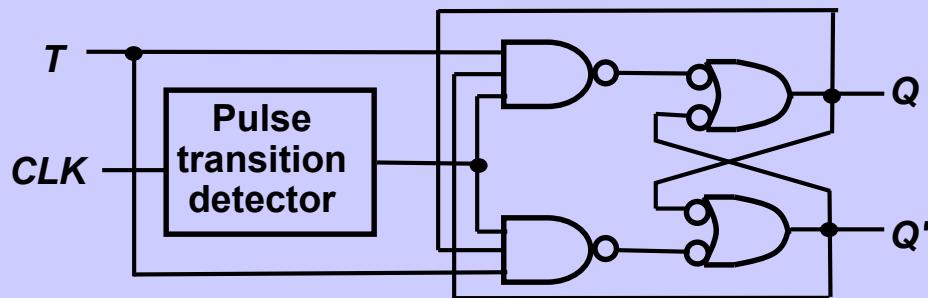
$Q(t)$	$Q(t+1)$	$J$	$K$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

**Excitation table** - shows the minimum inputs that are required to generate a particular next state or to "excite" it to the next state, when the current state is known.

They are similar to truth tables, except for the rearrangement of the data. Here, the current state and next state are next to each other on the left-hand side of the table, and the inputs needed to make that state change happen are shown on the right side of the table.

# Edge-Triggered T Flip-Flop

- T flip-flop: single-input version of the J-K flip flop, formed by tying both inputs together.



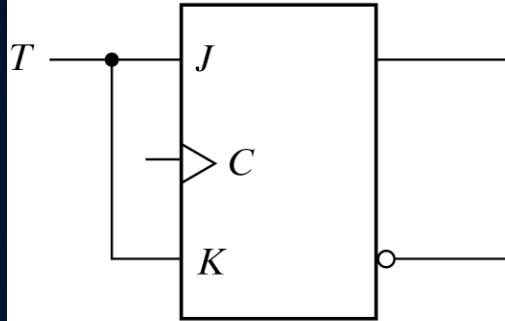
- Characteristic table.

T	CLK	Q(t+1)	Comments
0	↑	Q(t)	No change
1	↑	Q(t)'	Toggle

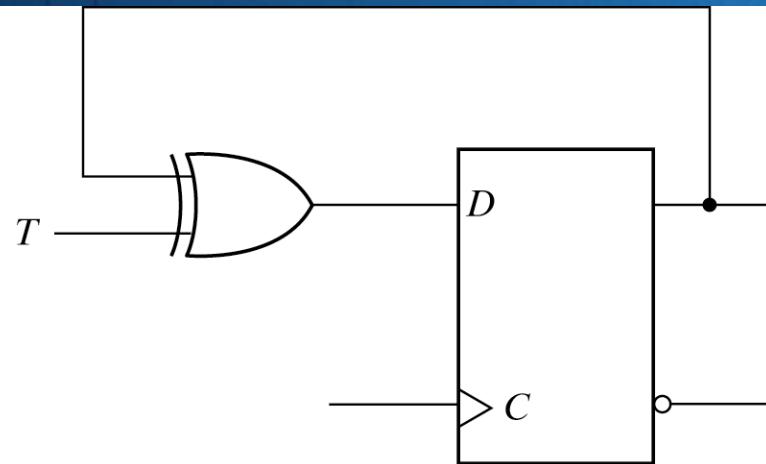
Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

$$Q(t+1) = T \cdot Q' + T' \cdot Q$$

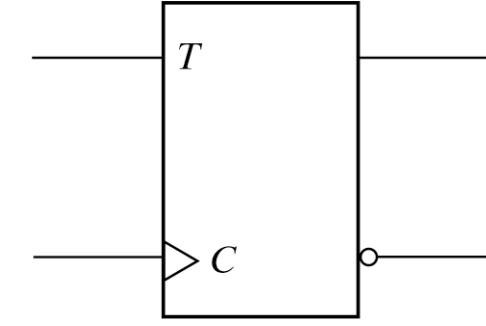
# Edge-Triggered T Flip-Flop



(a) From JK flip-flop



(b) From D flip-flop



(c) Graphic symbol

Fig. 5-13 T Flip-Flop

$$\begin{array}{ll} T & Q(t+1) \\ 0 & Q(t) \\ 1 & Q'(t) \end{array}$$

$$Q(t+1) = T \oplus Q = TQ' + T'Q$$

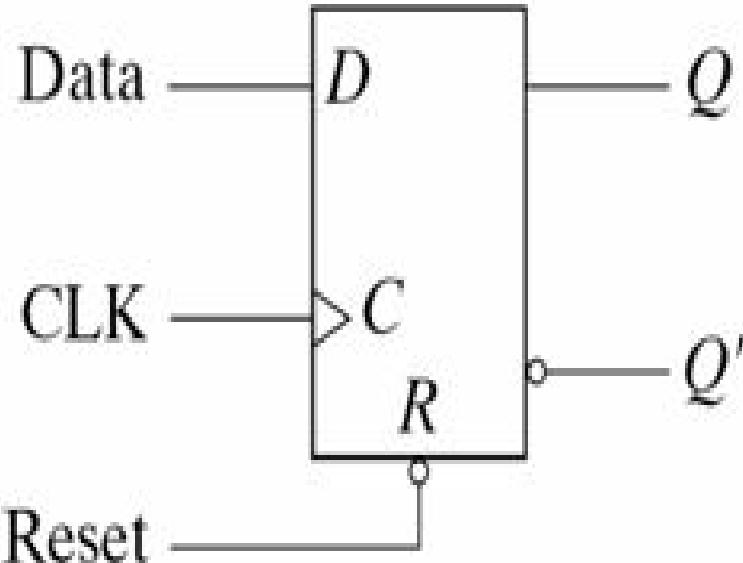
# Excitation Table

$Q(t)$	$Q(t+1)$	$T$
0	0	0
0	1	1
1	0	1
1	1	0

# Direct Inputs

- ◆ You can use asynchronous inputs to put a flip-flop to a specific state regardless of the clock
- ◆ You can clear the content of a flip-flop
  - The content is changed to zero (0)
  - This is called clear or direct reset
  - This is particularly useful when the power is off
    - ✓ The state of the flip-flop is set to unknown

# D Flip-Flop with Asynchronous Reset



(b) Graphic symbol

R	C	D	Q	$Q'$
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

(b) Function table

Fig. 5-14 D Flip-Flop with Asynchronous Reset

# Asynchronous Inputs(contd...)

Asynchronous inputs (Preset & Clear) are used to override the clock/data inputs and force the outputs to a predefined state.

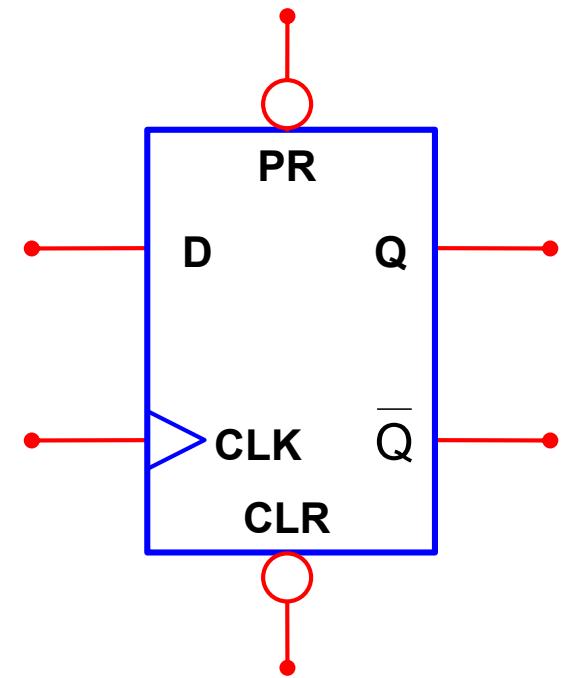
The Preset (PR) input forces the output to:

$$Q = 1 \text{ & } \bar{Q} = 0$$

The Clear (CLR) input forces the output to:

$$Q = 0 \text{ & } \bar{Q} = 1$$

PR PRESET	CLR CLEAR	CLK CLOCK	D DATA	Q	$\bar{Q}$
1	1	$\uparrow$	0	0	1
1	1	$\uparrow$	1	1	0
0	1	X	X	1	0
1	0	X	X	0	1
Mod-5_Sequential_ckts-I-Latches-flip-flopPart-1	0	X	X	1	1

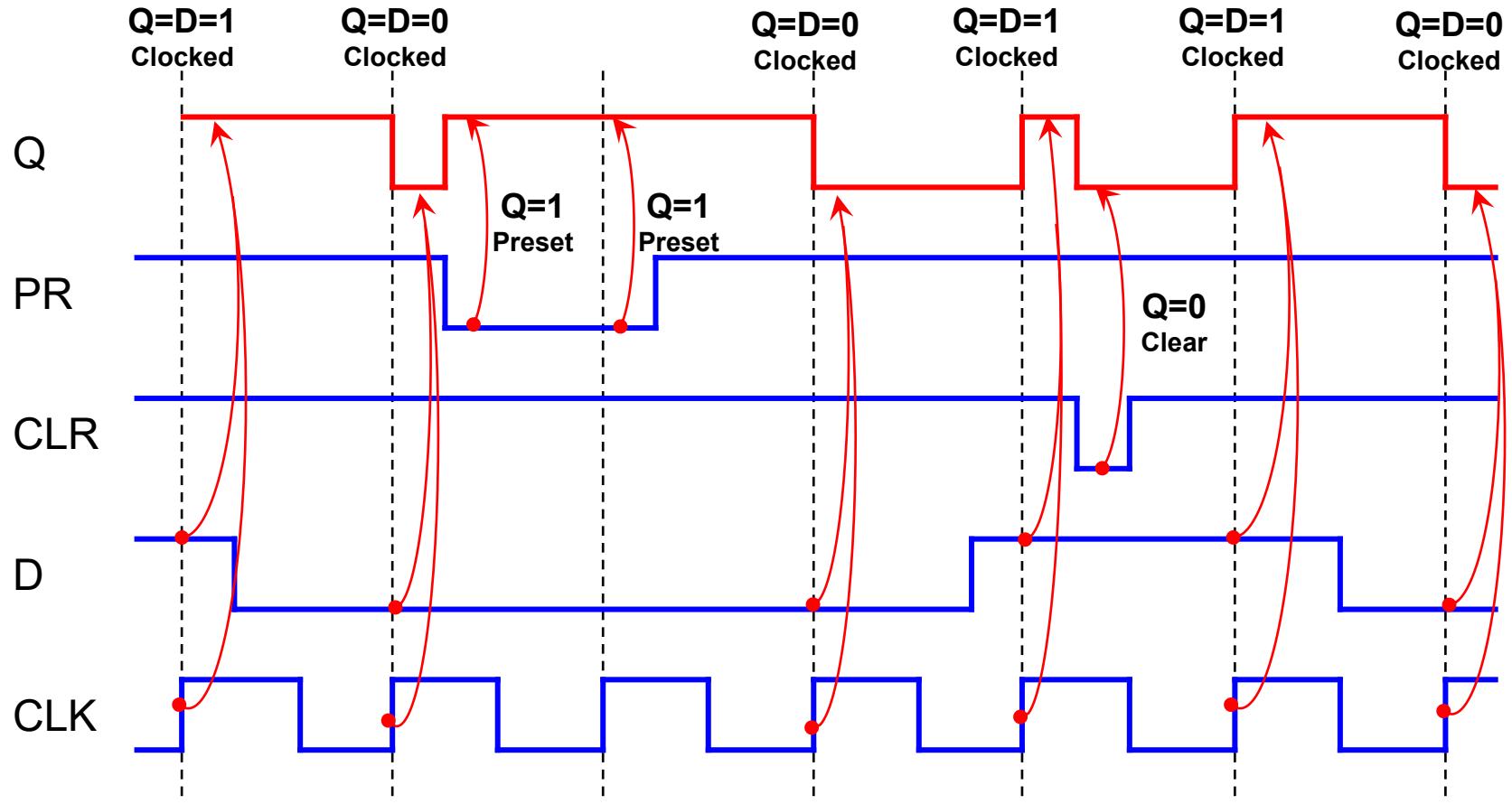


Asynchronous Preset

Asynchronous Clear

ILLEGAL CONDITION

# D Flip-Flop: PR & CLR Timing



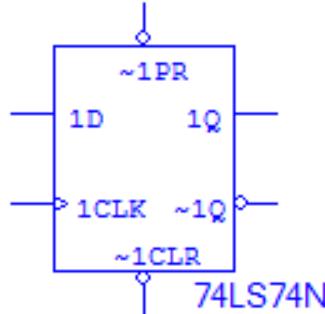
# Flip-Flop Vs. Latch

---

- The primary difference between a D flip-flop and D latch is the EN/CLOCK input.
- The flip-flop's CLOCK input is edge sensitive, meaning the flip-flop's output changes on the edge (rising or falling) of the CLOCK input.
- The latch's EN input is level sensitive, meaning the latch's output changes on the level (high or low) of the EN input.

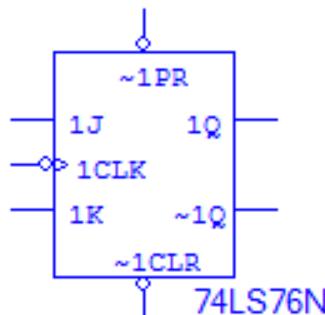
# Flip-Flops & Latches

---



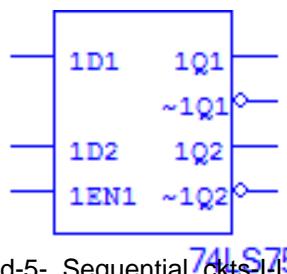
74LS74

Dual Positive-Edge-Triggered D Flip-Flops with Preset, Clear, and Complementary Outputs



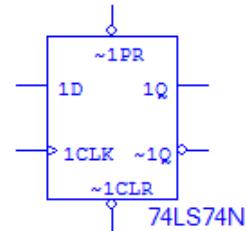
74LS76

Dual Negative-Edge-Triggered J-K Flip-Flops with Preset, Clear, and Complementary Outputs



74LS75  
Quad Latch

# 74LS74: D Flip-Flop



## Function Table

Inputs				Outputs	
PR	CLR	CLK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H (Note 1)	H (Note 1)
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	$Q_0$	$\bar{Q}_0$

H = HIGH Logic Level

X = Either LOW or HIGH Logic Level

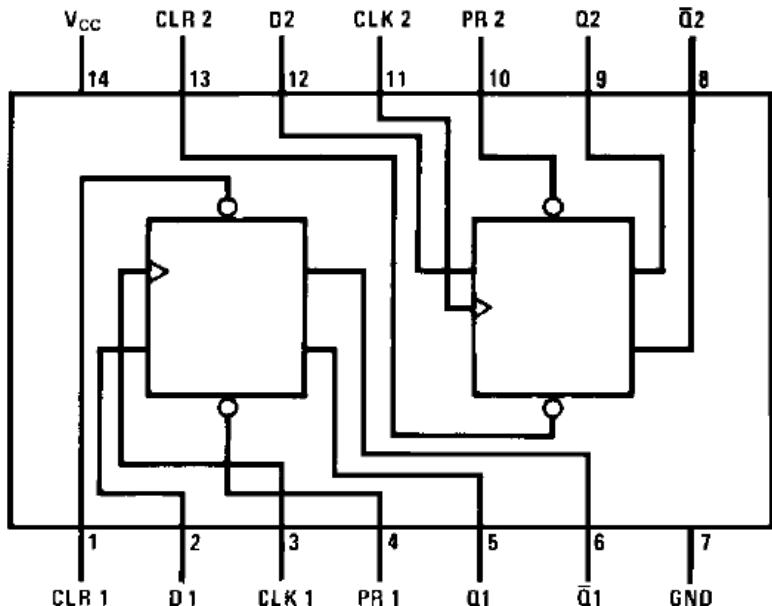
L = LOW Logic Level

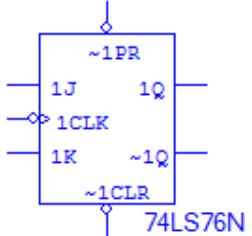
↑ = Positive-going Transition

$Q_0$  = The output logic level of Q before the indicated input conditions were established.

**Note 1:** This configuration is nonstable; that is, it will not persist when either the preset and/or clear inputs return to their inactive (HIGH) level.

## Connection Diagram





# 74LS76: J/K Flip-Flop

## Function Table

Inputs					Outputs	
PR	CLR	CLK	J	K	Q	$\bar{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H	H
					(Note 1)	(Note 1)
H	H	$\text{--}$	L	L	$Q_0$	$\bar{Q}_0$
H	H	$\text{--}$	H	L	H	L
H	H	$\text{--}$	L	H	L	H
H	H	$\text{--}$	H	H	Toggle	

H = High Logic Level

L = Low Logic Level

X = Either Low or High Logic Level

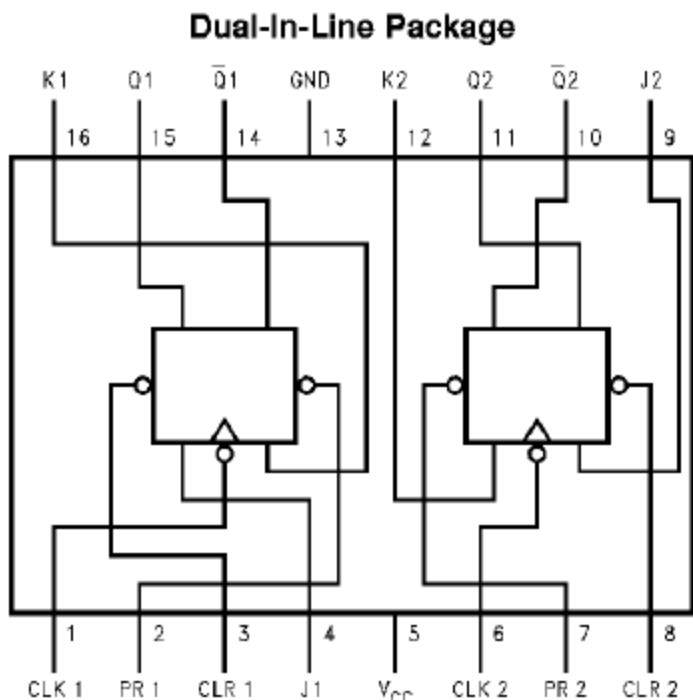
$\text{--}$  = Positive pulse data. The J and K inputs must be held constant while the clock is high. Data is transferred to the outputs on the falling edge of the clock pulse.

$Q_0$  = The output logic level before the indicated input conditions were established.

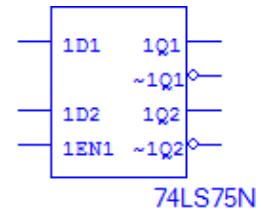
Toggle = Each output changes to the complement of its previous level on each complete active high level clock pulse.

**Note 1:** This configuration is nonstable; that is, it will not persist when the preset and/or clear inputs return to their inactive (high) level.

## Connection Diagram



# 74LS75: D Latch



**Function Table** (Each Latch)

Inputs		Outputs	
D	Enable	Q	$\bar{Q}$
L	H	L	H
H	H	H	L
X	L	$Q_0$	$\bar{Q}_0$

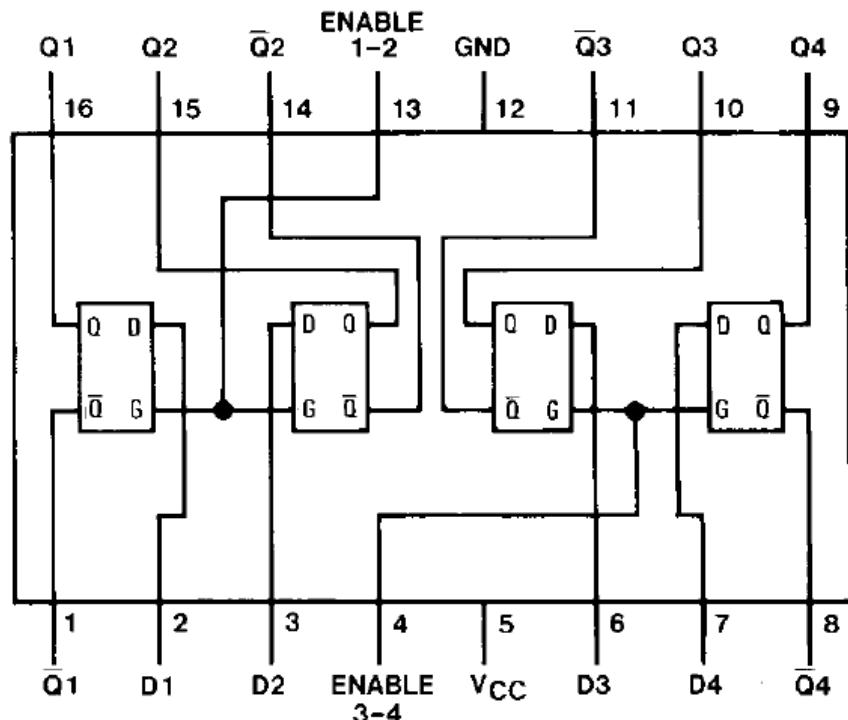
H = HIGH Level

L = LOW Level

X = Don't Care

$Q_0$  = The Level of Q Before the HIGH-to-LOW Transition of ENABLE

**Connection Diagram**



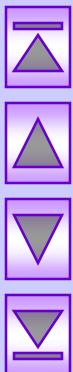
# **Module:6 Sequential Circuits-II**

<b>Module:5</b>	<b>SEQUENTIAL CIRCUITS – I</b>	<b>6 hours</b>
Flip Flops - Sequential Circuit: Design and Analysis - Finite State Machine: Moore and Mealy model - Sequence Detector.		
<b>Module:6</b>	<b>SEQUENTIAL CIRCUITS – II</b>	<b>7 hours</b>
Registers - Shift Registers - Counters - Ripple and Synchronous Counters - Modulo counters - Ring and Johnson counters		

# Sequential Logic Counters and Registers

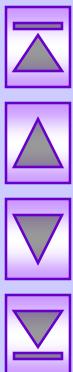
## Counters

- Introduction: Counters
- Asynchronous (Ripple) Counters
- Asynchronous Counters with MOD number <  $2^n$
- Asynchronous Down Counters
- Cascading Asynchronous Counters



# Sequential Logic Counters and Registers

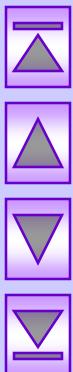
- Synchronous (Parallel) Counters
- Up/Down Synchronous Counters
- Designing Synchronous Counters
- Decoding A Counter
- Counters with Parallel Load



# Sequential Logic Counters and Registers

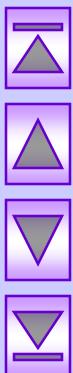
## Registers

- Introduction: Registers
  - ❖ Simple Registers
  - ❖ Registers with Parallel Load
- Using Registers to implement Sequential Circuits
- Shift Registers
  - ❖ Serial In/Serial Out Shift Registers
  - ❖ Serial In/Parallel Out Shift Registers
  - ❖ Parallel In/Serial Out Shift Registers
  - ❖ Parallel In/Parallel Out Shift Registers



# Sequential Logic Counters and Registers

- Bidirectional Shift Registers
- An Application – Serial Addition
- Shift Register Counters
  - ❖ Ring Counters
  - ❖ Johnson Counters
- Random-Access Memory (RAM)



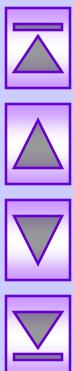
# Introduction: Counters

- Counters are circuits that cycle through a specified number of states.
- Two types of counters:
  - ❖ synchronous (parallel) counters
  - ❖ asynchronous (ripple) counters
- Ripple counters allow some flip-flop outputs to be used as a source of clock for other flip-flops.
- Synchronous counters apply the same clock to all flip-flops.



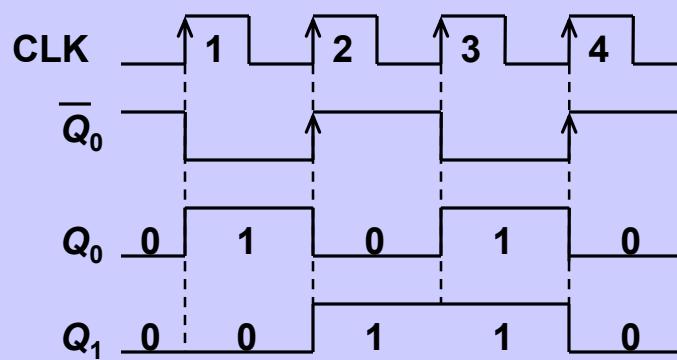
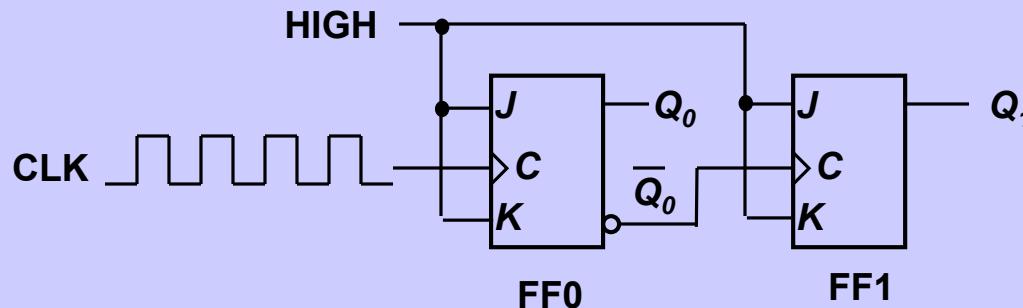
# Asynchronous (Ripple) Counters

- Asynchronous counters: the flip-flops do not change states at exactly the same time as they do not have a common clock pulse.
- Also known as ripple counters, as the input clock pulse “ripples” through the counter – cumulative delay is a drawback.
- $n$  flip-flops  $\rightarrow$  a MOD (modulus)  $2^n$  counter. (Note: A MOD- $x$  counter cycles through  $x$  states.)
- Output of the last flip-flop (MSB) divides the input clock frequency by the MOD number of the counter, hence a counter is also a *frequency divider*.



# Asynchronous (Ripple) Counters

- Example: 2-bit ripple binary counter.
- Output of one flip-flop is connected to the clock input of the next more-significant flip-flop.

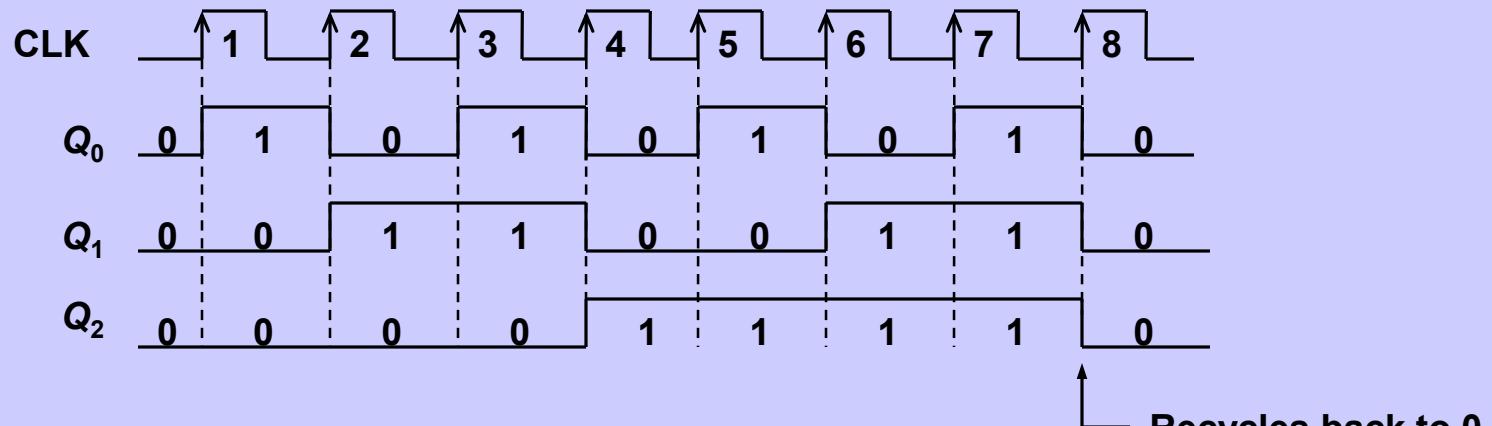
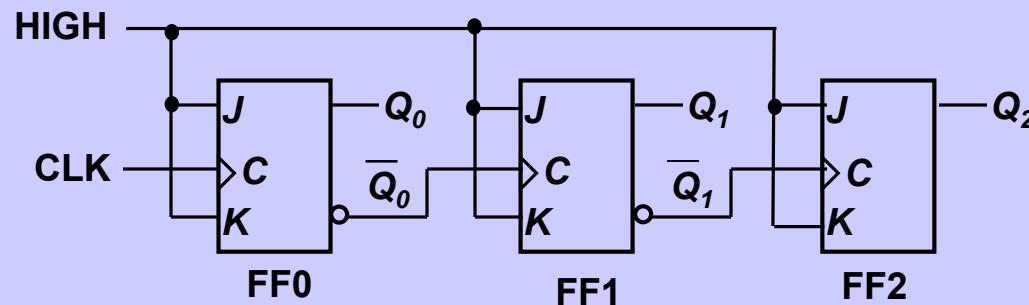


Timing diagram

$00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00 \dots$

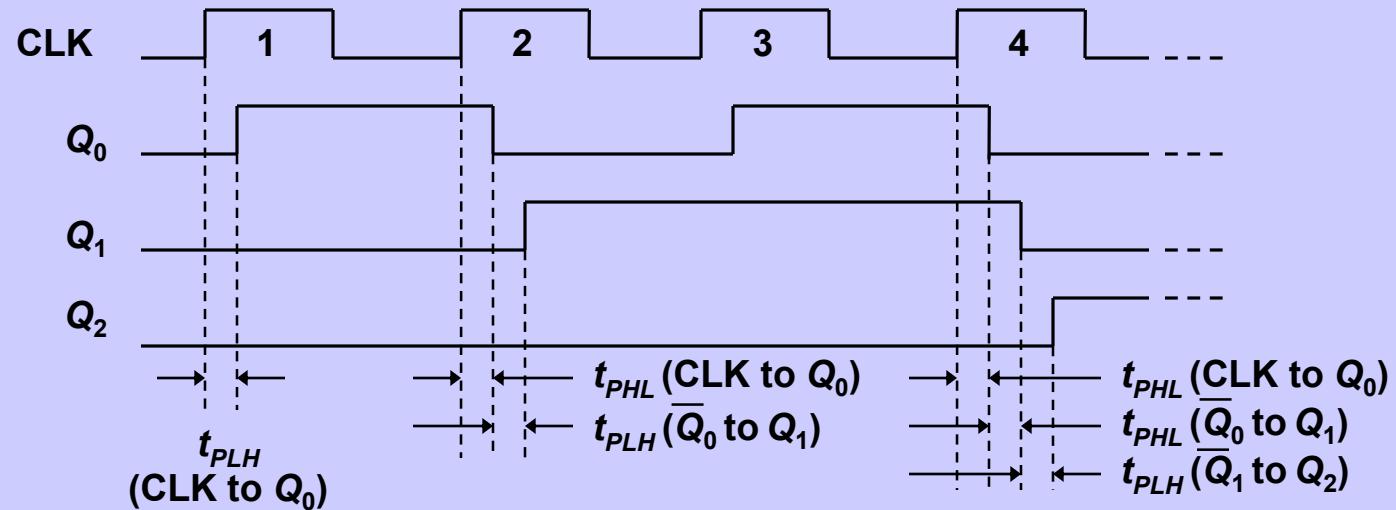
# Asynchronous (Ripple) Counters

- Example: 3-bit ripple binary counter.



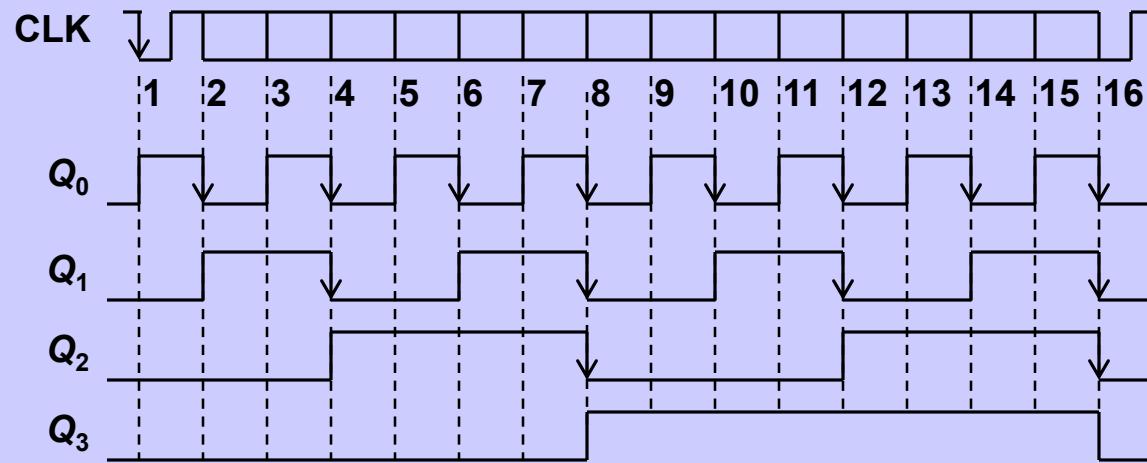
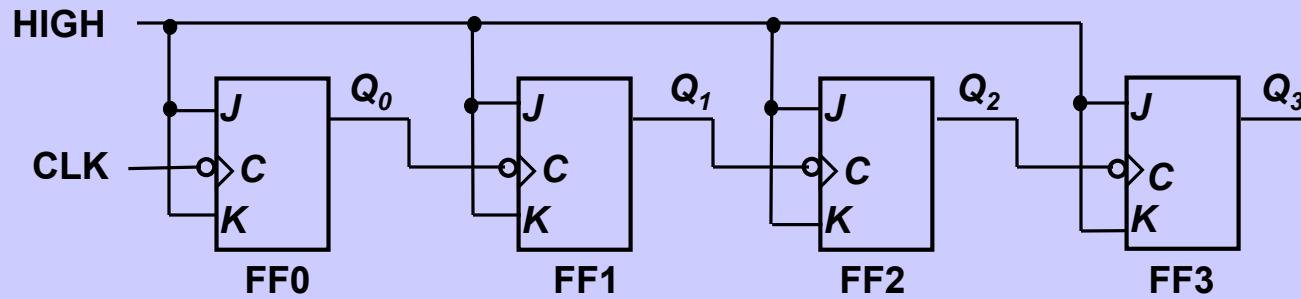
# Asynchronous (Ripple) Counters

- Propagation delays in an asynchronous (ripple-clocked) binary counter.
- If the accumulated delay is greater than the clock pulse, some counter states may be misrepresented!



# Asynchronous (Ripple) Counters

- Example: 4-bit ripple binary counter (negative-edge triggered).

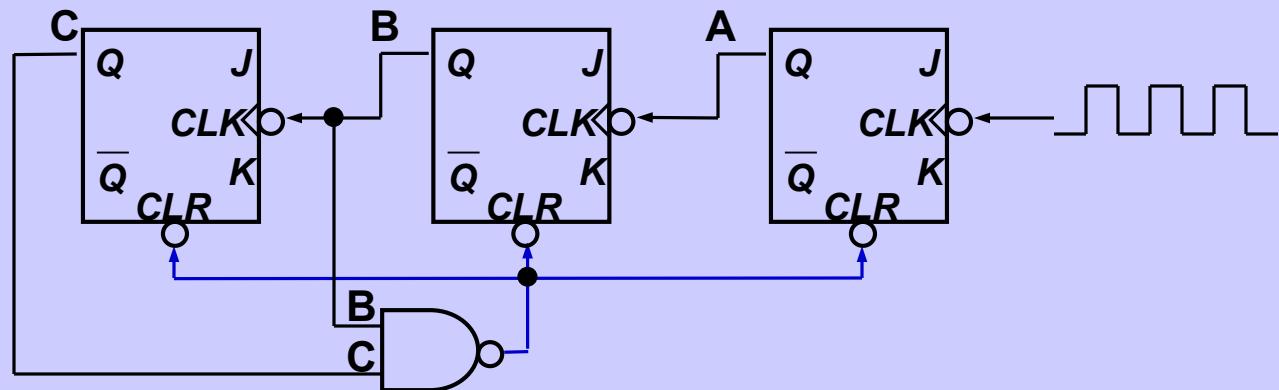


# Asyn. Counters with MOD no. $< 2^n$

- States may be skipped resulting in a **truncated sequence**.
- Technique: force counter to *recycle before going through all of the states* in the binary sequence.
- Example: Given the following circuit, determine the counting sequence (and hence the modulus no.)

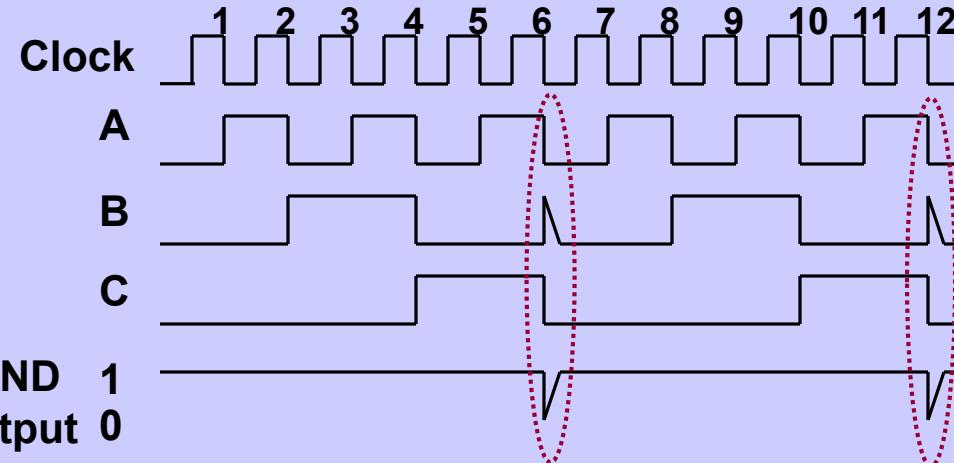
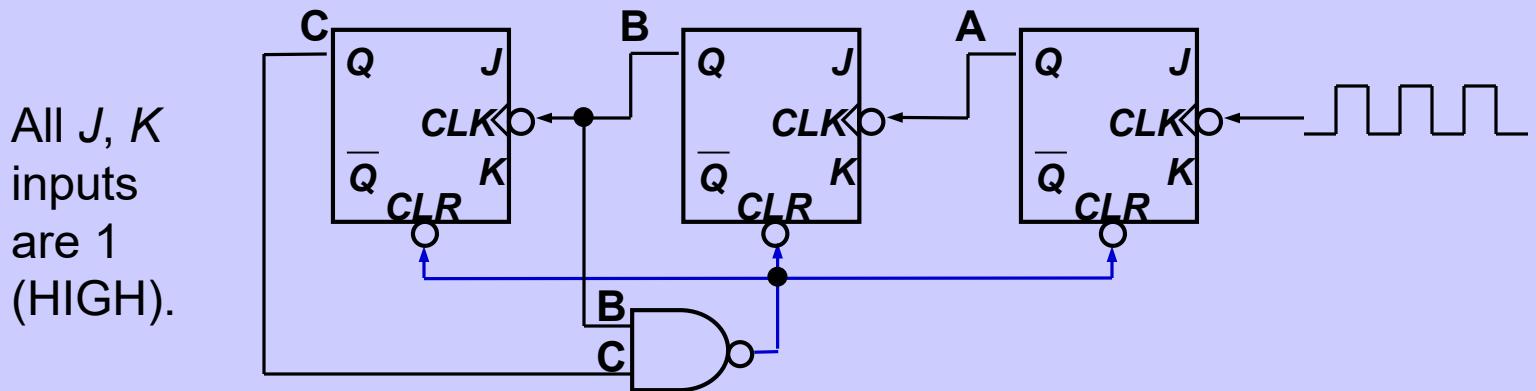


All  $J, K$   
inputs  
are 1  
(HIGH).



# Asyn. Counters with MOD no. $< 2^n$

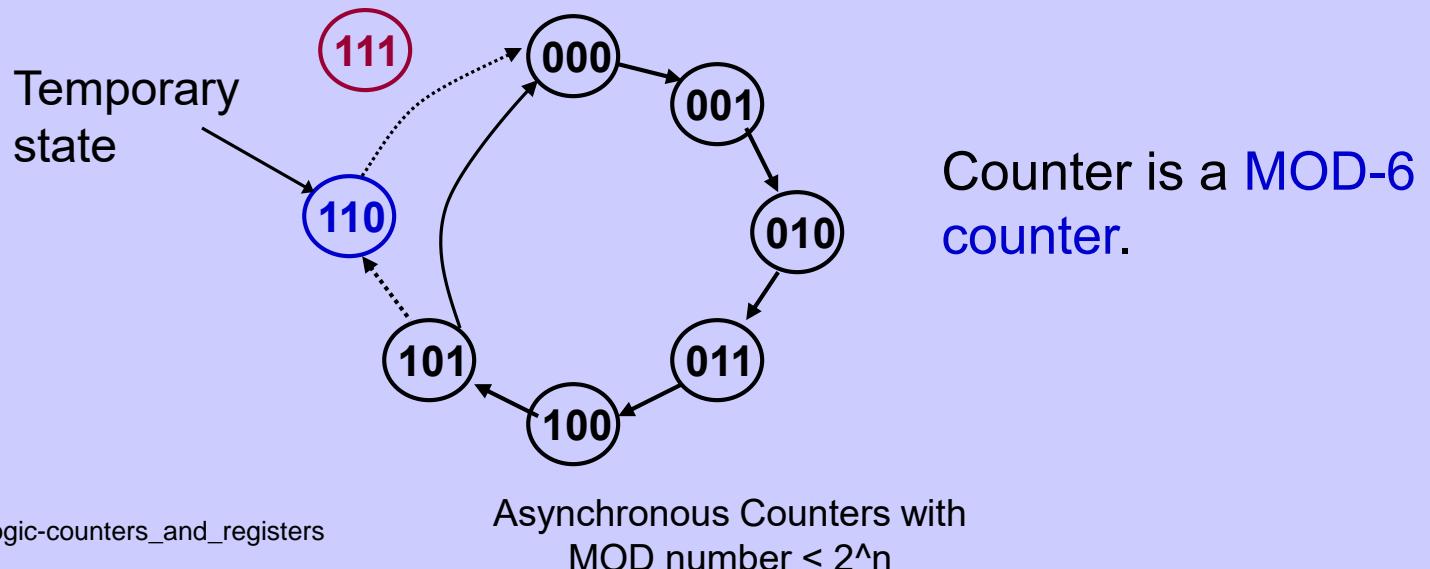
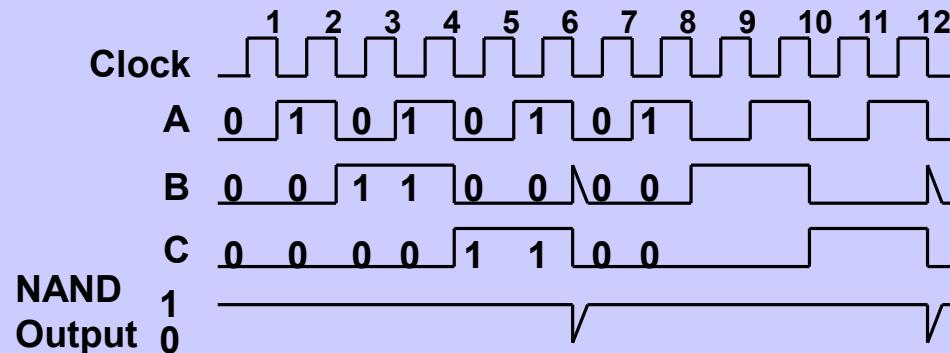
- Example (cont'd):



MOD-6 counter produced by clearing (a MOD-8 binary counter) when count of six (110) occurs.

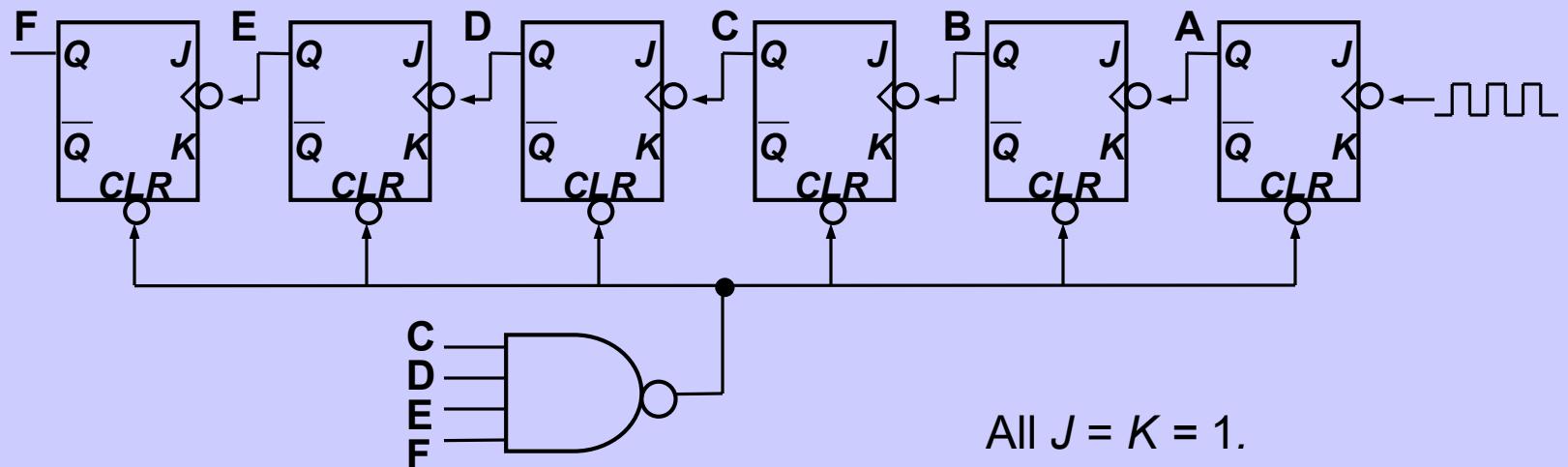
# Asyn. Counters with MOD no. < $2^n$

- Example (cont'd): Counting sequence of circuit (in CBA order).



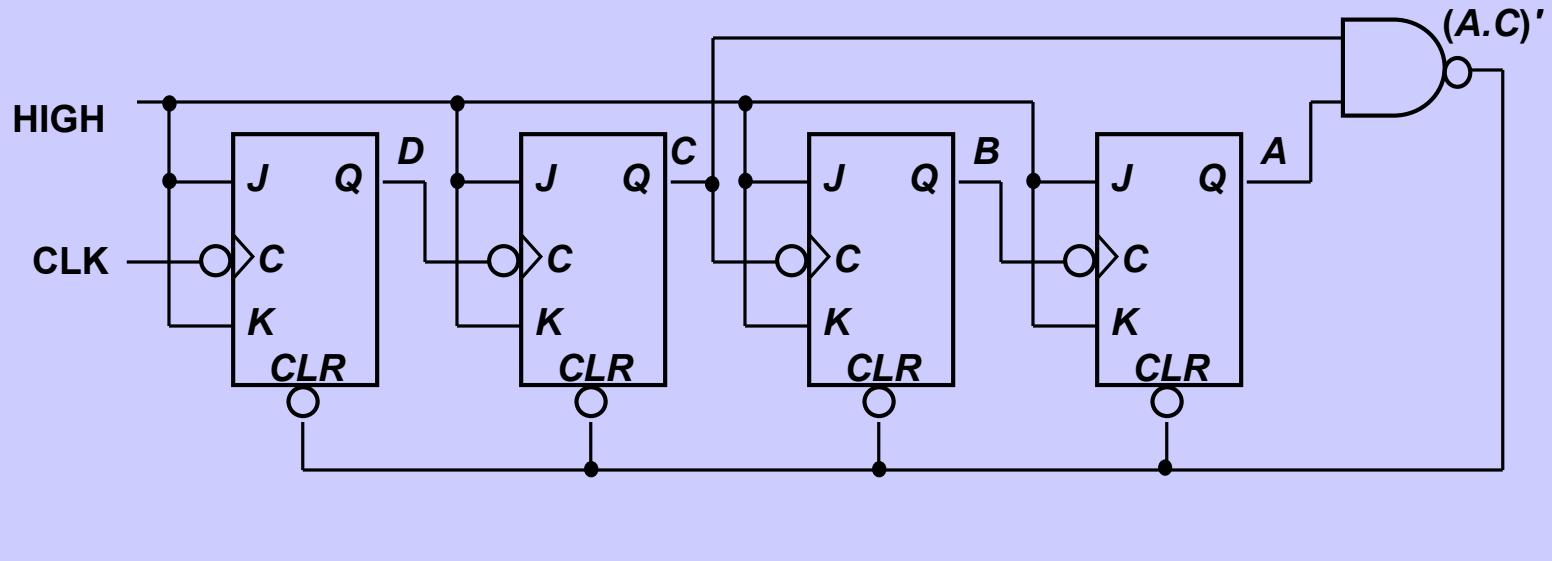
# Asyn. Counters with MOD no. < $2^n$

- *Exercise:* How to construct an asynchronous MOD-5 counter? MOD-7 counter? MOD-12 counter?
- *Question:* The following is a MOD-? counter?



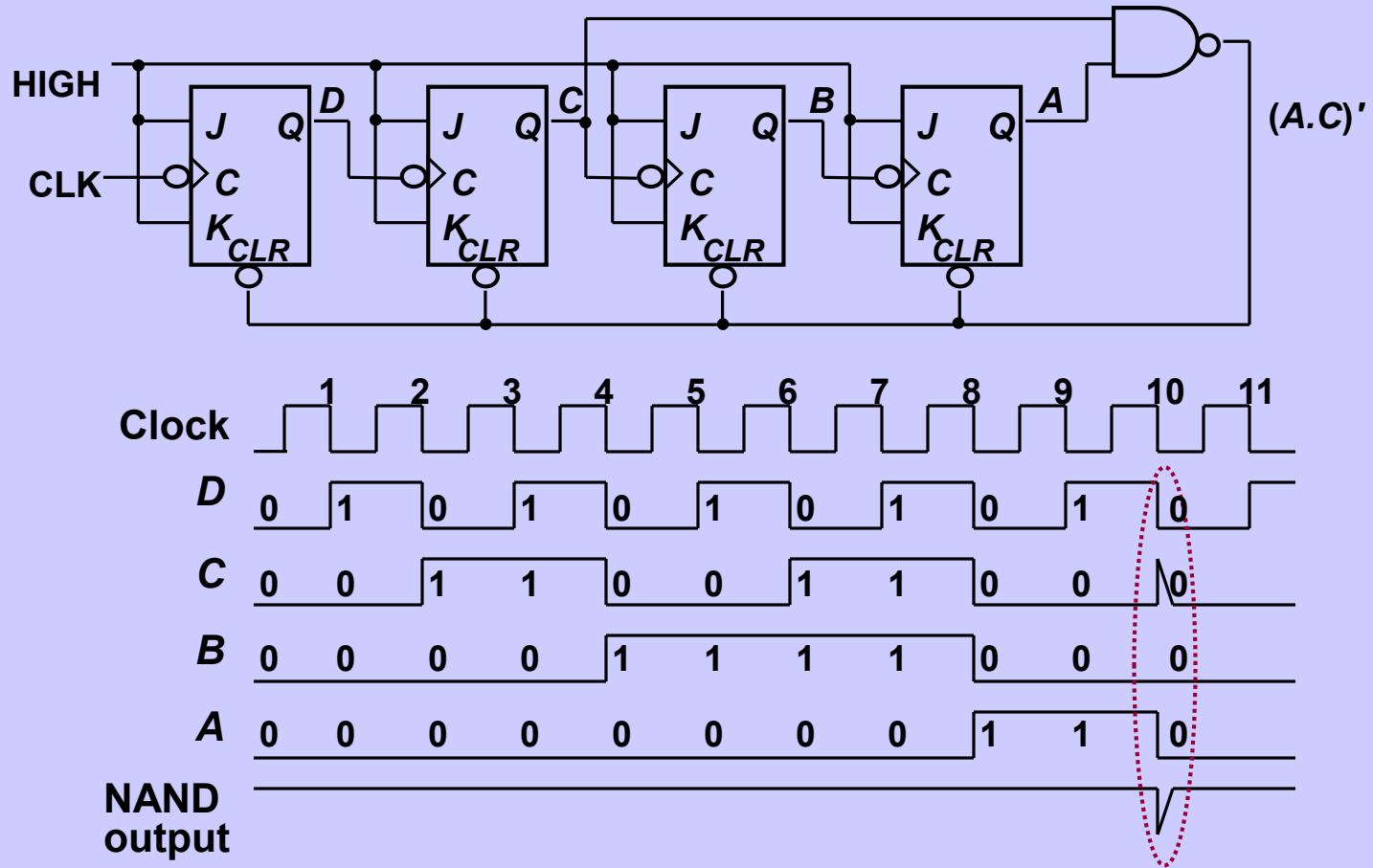
# Asyn. Counters with MOD no. $< 2^n$

- Decade counters (or BCD counters) are counters with 10 states (modulus-10) in their sequence. They are commonly used in daily life (e.g.: utility meters, odometers, etc.).
- Design an asynchronous decade counter.



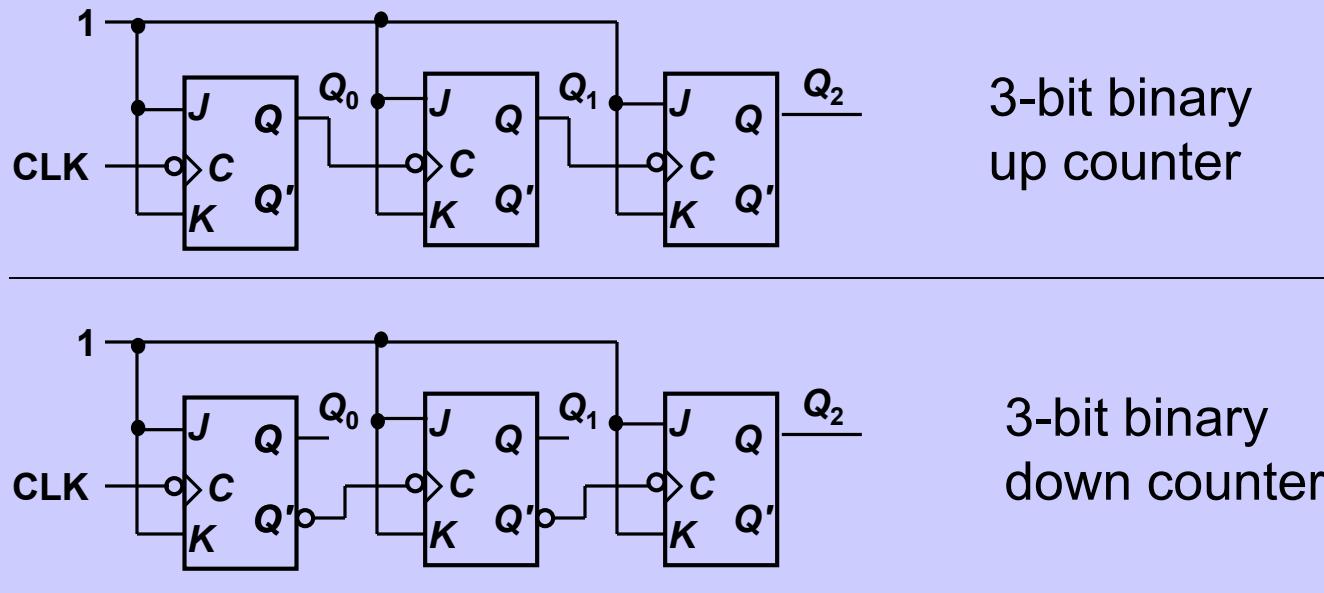
# Asyn. Counters with MOD no. $< 2^n$

- Asynchronous decade/BCD counter (cont'd).



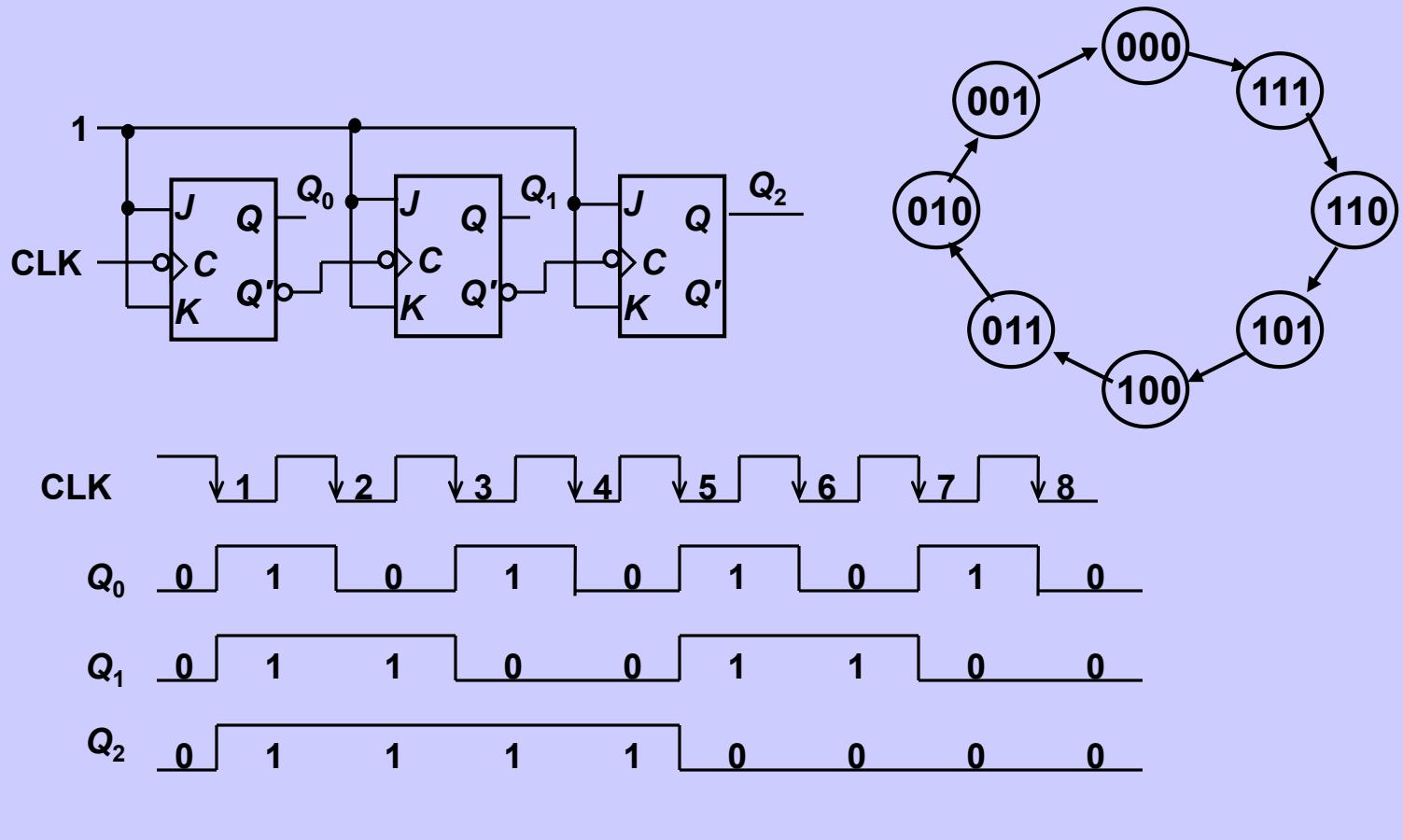
# Asynchronous Down Counters

- So far we are dealing with *up counters*. *Down counters*, on the other hand, count downward from a maximum value to zero, and repeat.
- Example: A 3-bit binary (MOD- $2^3$ ) down counter.



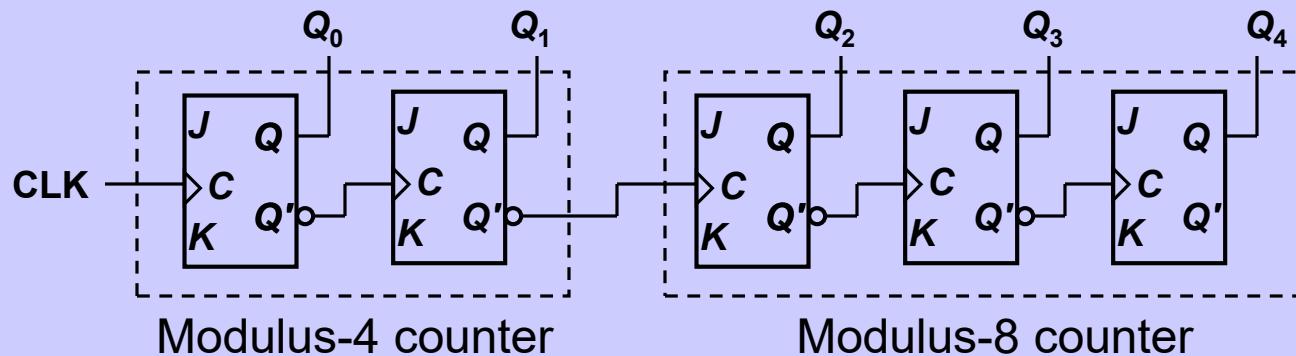
# Asynchronous Down Counters

- Example: A 3-bit binary (MOD-8) down counter.



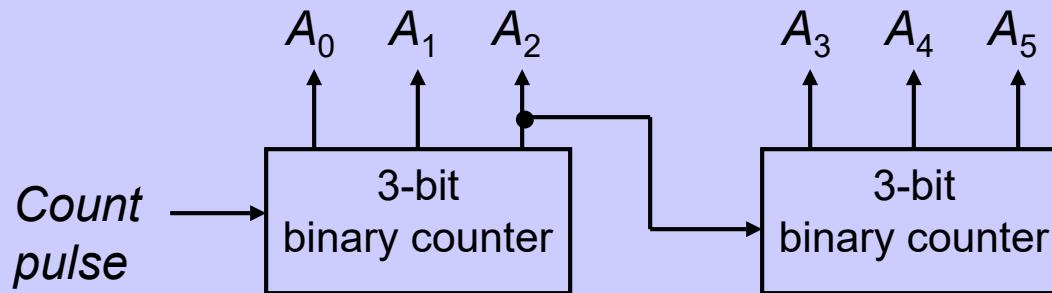
# Cascading Asynchronous Counters

- Larger asynchronous (ripple) counter can be constructed by cascading smaller ripple counters.
- Connect last-stage output of one counter to the clock input of next counter so as to achieve higher-modulus operation.
- Example: A modulus-32 ripple counter constructed from a modulus-4 counter and a modulus-8 counter.



# Cascading Asynchronous Counters

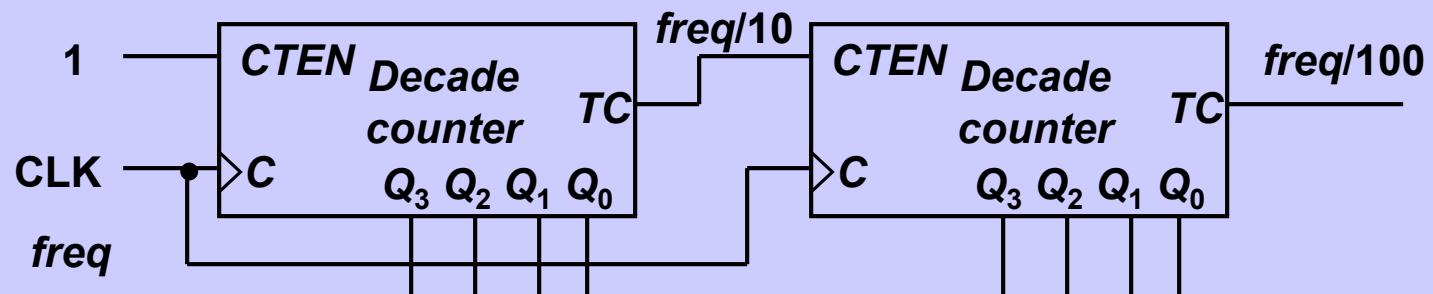
- Example: A 6-bit binary counter (counts from 0 to 63) constructed from two 3-bit counters.



$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	:	:	:
0	0	0	1	1	1
0	0	<b>1</b>	0	0	0
0	0	1	0	0	1
:	:	:	:	:	:

# Cascading Asynchronous Counters

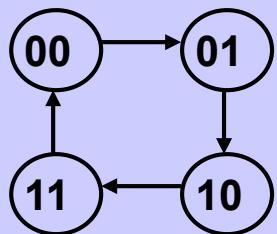
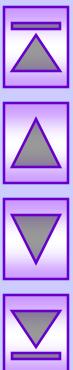
- If counter is not a binary counter, requires additional output.
- Example: A modulus-100 counter using two decade counters.



$TC = 1$  when counter recycles to 0000

# Synchronous (Parallel) Counters

- Synchronous (parallel) counters: the flip-flops are clocked at the same time by a common clock pulse.
- We can design these counters using the sequential logic design process (covered in Lecture #12).
- Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).



Present state		Next state		Flip-flop inputs	
$A_1$	$A_0$	$A_1^+$	$A_0^+$	$TA_1$	$TA_0$
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

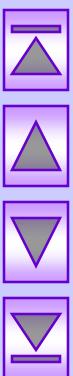
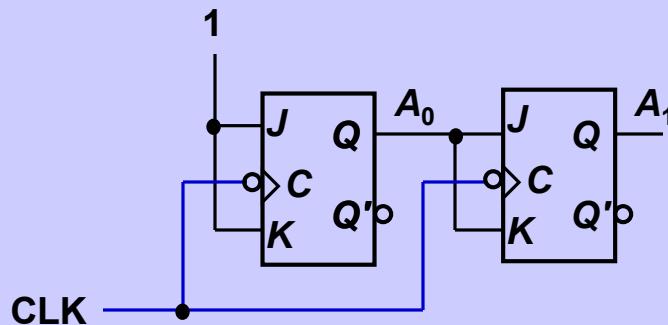
# Synchronous (Parallel) Counters

- Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).

Present state		Next state		Flip-flop inputs	
$A_1$	$A_0$	$A_1^+$	$A_0^+$	$TA_1$	$TA_0$
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

$$TA_1 = A_0$$

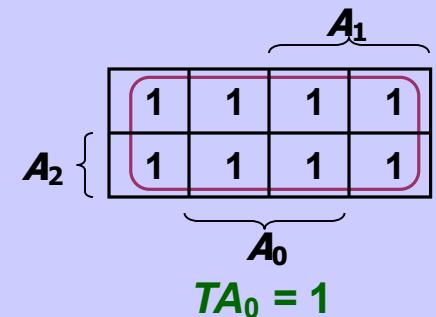
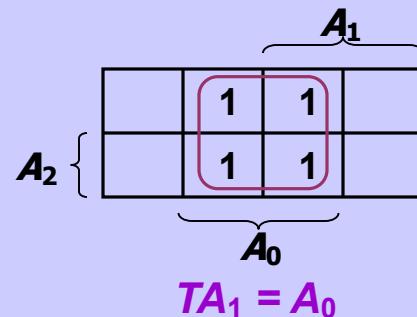
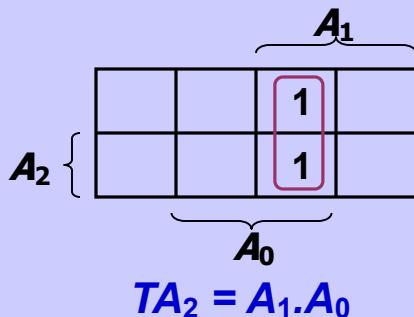
$$TA_0 = 1$$



# Synchronous (Parallel) Counters

- Example: 3-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J, K inputs).

Present state			Next state			Flip-flop inputs		
$A_2$	$A_1$	$A_0$	$A_2^+$	$A_1^+$	$A_0^+$	$TA_2$	$TA_1$	$TA_0$
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



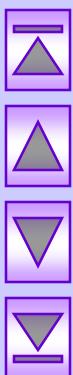
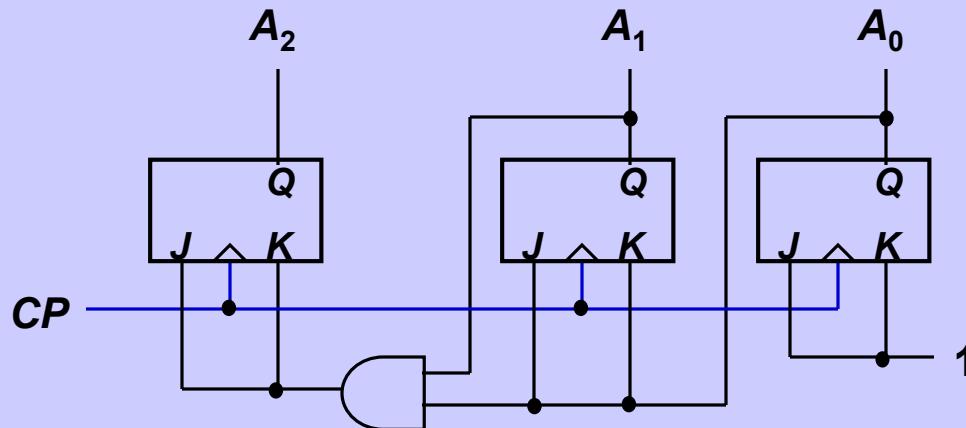
# Synchronous (Parallel) Counters

- Example: 3-bit synchronous binary counter (cont'd).

$$TA_2 = A_1 \cdot A_0$$

$$TA_1 = A_0$$

$$TA_0 = 1$$



# Synchronous (Parallel) Counters

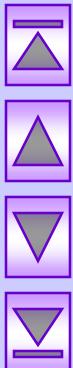
- Note that in a binary counter, the  $n^{\text{th}}$  bit (shown underlined) is always complemented whenever

$$\underline{0}11\dots11 \rightarrow \underline{1}00\dots00$$

or       $\underline{1}11\dots11 \rightarrow \underline{0}00\dots00$

- Hence,  $X_n$  is complemented whenever  
 $X_{n-1}X_{n-2}\dots X_1X_0 = 11\dots11.$
- As a result, if  $T$  flip-flops are used, then

$$TX_n = X_{n-1} \cdot X_{n-2} \cdot \dots \cdot X_1 \cdot X_0$$



# Synchronous (Parallel) Counters

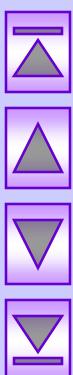
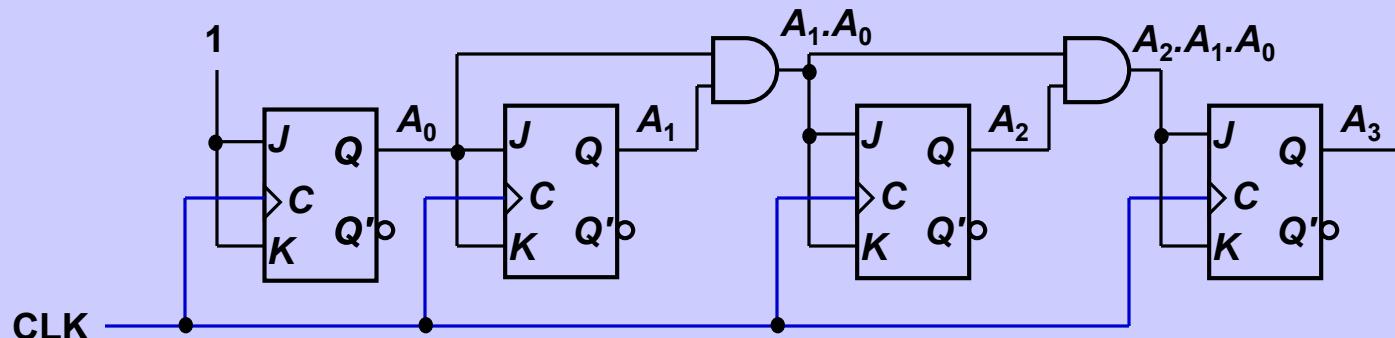
- Example: 4-bit synchronous binary counter.

$$TA_3 = A_2 \cdot A_1 \cdot A_0$$

$$TA_2 = A_1 \cdot A_0$$

$$TA_1 = A_0$$

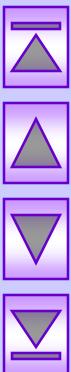
$$TA_0 = 1$$



# Synchronous (Parallel) Counters

- Example: Synchronous decade/BCD counter.

Clock pulse	$Q_3$	$Q_2$	$Q_1$	$Q_0$	
Initially	0	0	0	0	
1	0	0	0	1	$T_0 = 1$
2	0	0	1	0	
3	0	0	1	1	$T_1 = Q_3'.Q_0$
4	0	1	0	0	
5	0	1	0	1	$T_2 = Q_1.Q_0$
6	0	1	1	0	
7	0	1	1	1	$T_3 = Q_2.Q_1.Q_0 + Q_3.Q_0$
8	1	0	0	0	
9	1	0	0	1	
10 (recycle)	0	0	0	0	



# Synchronous (Parallel) Counters

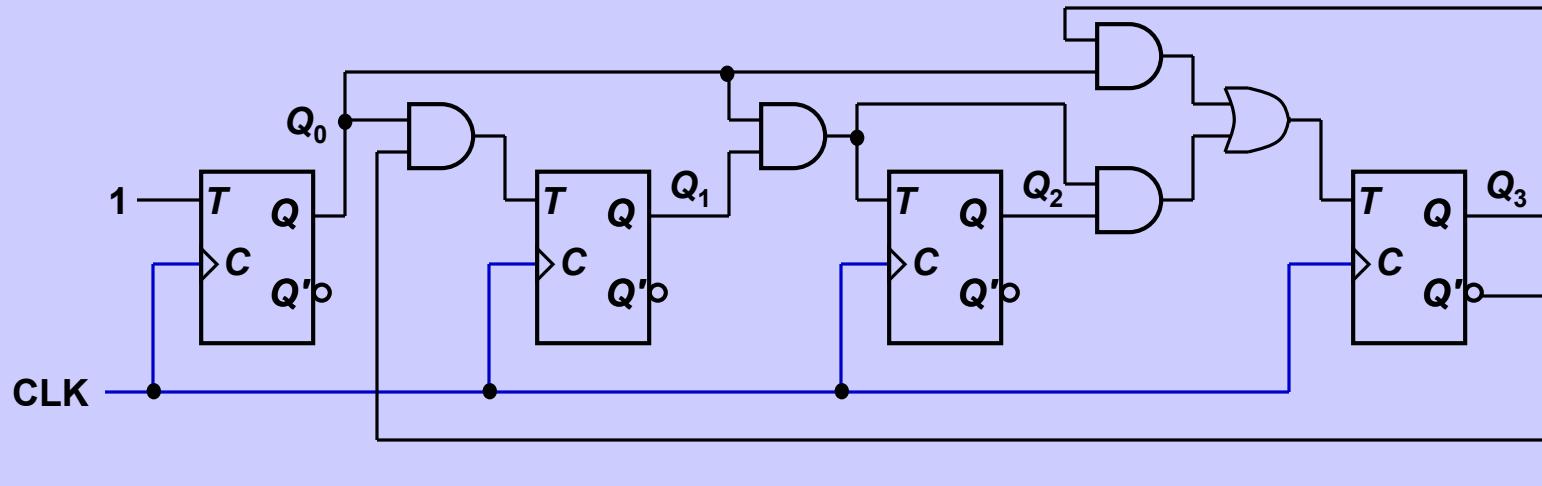
- Example: Synchronous decade/BCD counter (cont'd).

$$T_0 = 1$$

$$T_1 = Q_3'.Q_0$$

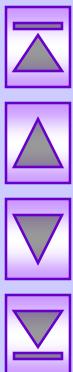
$$T_2 = Q_1.Q_0$$

$$T_3 = Q_2.Q_1.Q_0 + Q_3.Q_0$$



# Up/Down Synchronous Counters

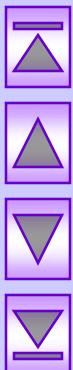
- Up/down synchronous counter: a *bidirectional* counter that is capable of counting either up or down.
- An input (control) line  $Up/\overline{Down}$  (or simply  $Up$ ) specifies the direction of counting.
  - ❖  $Up/\overline{Down} = 1 \rightarrow$  Count upward
  - ❖  $Up/\overline{Down} = 0 \rightarrow$  Count downward



# Up/Down Synchronous Counters

- Example: A 3-bit up/down synchronous binary counter.

Clock pulse	Up	$Q_2$	$Q_1$	$Q_0$	Down
0		0	0	0	
1		0	0	1	
2		0	1	0	
3		0	1	1	
4		1	0	0	
5		1	0	1	
6		1	1	0	
7		1	1	1	



$$TQ_0 = 1$$

$$TQ_1 = (Q_0 \cdot Up) + (Q_0' \cdot Up')$$

$$TQ_2 = (Q_0 \cdot Q_1 \cdot Up) + (Q_0' \cdot Q_1' \cdot Up')$$

Up counter

$$TQ_0 = 1$$

$$TQ_1 = Q_0$$

$$TQ_2 = Q_0 \cdot Q_1$$

Down counter

$$TQ_0 = 1$$

$$TQ_1 = Q_0'$$

$$TQ_2 = Q_0' \cdot Q_1'$$

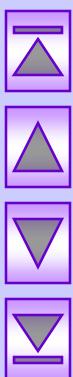
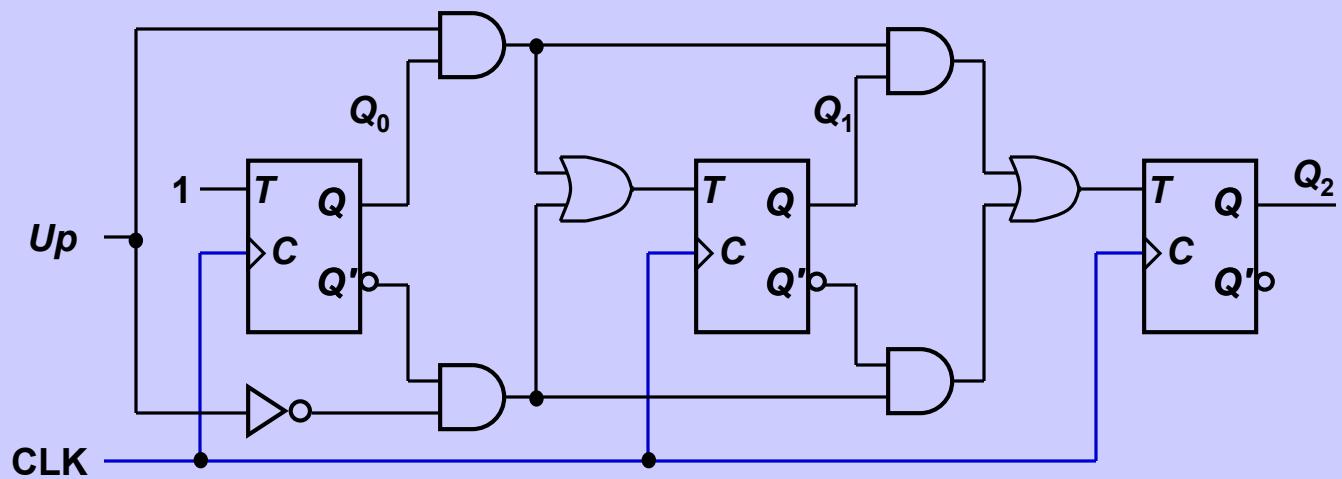
# Up/Down Synchronous Counters

- Example: A 3-bit up/down synchronous binary counter (cont'd).

$$TQ_0 = 1$$

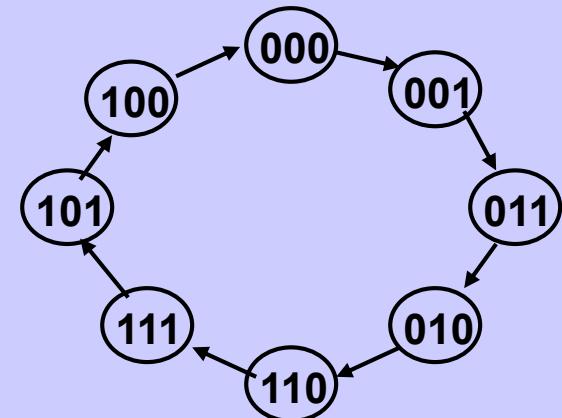
$$TQ_1 = (Q_0.Up) + (Q_0'.Up')$$

$$TQ_2 = (Q_0.Q_1.Up) + (Q_0'.Q_1'.Up')$$

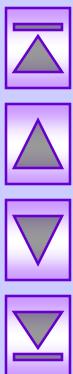


# Designing Synchronous Counters

- Covered in Lecture #12.
- Example: A 3-bit Gray code counter (using JK flip-flops).



Present state			Next state			Flip-flop inputs					
$Q_2$	$Q_1$	$Q_0$	$Q_2^+$	$Q_1^+$	$Q_0^+$	$JQ_2$	$KQ_2$	$JQ_1$	$KQ_1$	$JQ_0$	$KQ_0$
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	1	0	X	1	X	X	0
0	1	0	1	1	0	1	X	X	0	0	X
0	1	1	0	1	0	0	X	X	0	X	1
1	0	0	0	0	0	X	1	0	X	0	X
1	0	1	1	0	0	X	0	0	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	1	0	1	X	0	X	1	X	0



# Designing Synchronous Counters

- 3-bit Gray code counter: flip-flop inputs.

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0				1
		$Q_2$	1	X	X	X	X

$JQ_2 = Q_1 \cdot Q_0'$

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0	1	X	X	
		$Q_2$	1		X	X	

$JQ_1 = Q_2' \cdot Q_0$

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0	1	X	X	
		$Q_2$	1	X	X	1	1

$JQ_0 = Q_2 \cdot Q_1 + Q_2' \cdot Q_1'$   
 $= (Q_2 \oplus Q_1)'$

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0	X	X	X	X
		$Q_2$	1	1			

$KQ_2 = Q_1' \cdot Q_0'$

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0	X	X		
		$Q_2$	1	X	X	1	

$KQ_1 = Q_2 \cdot Q_0$

		$Q_1 Q_0$	00	01	11	10	
		$Q_2$	0	X		1	X
		$Q_2$	1	X	1		X

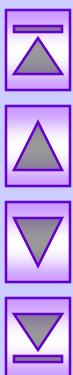
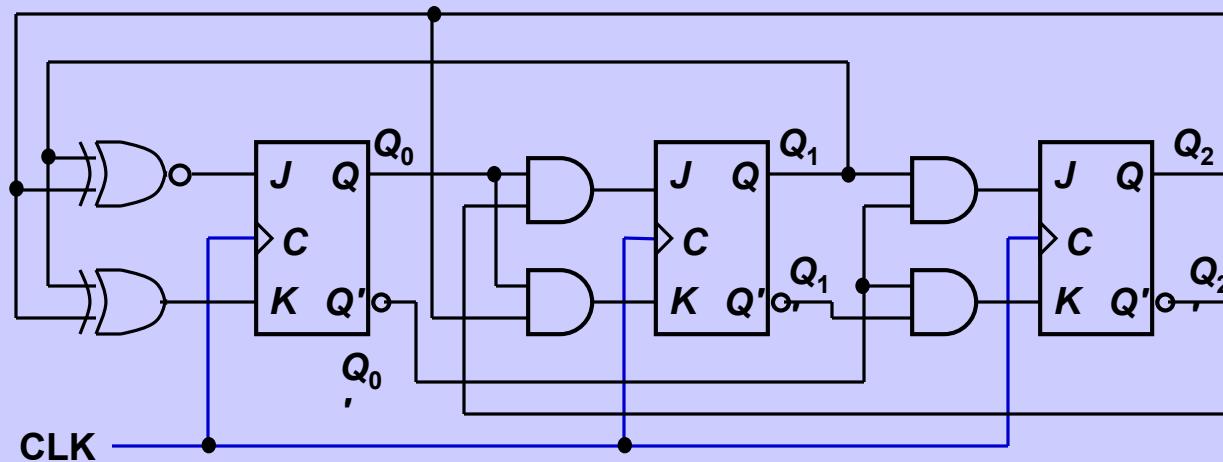
$KQ_0 = Q_2 \cdot Q_1' + Q_2' \cdot Q_1$   
 $= Q_2 \oplus Q_1$



# Designing Synchronous Counters

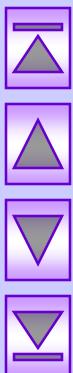
- 3-bit Gray code counter: logic diagram.

$$\begin{array}{lll} JQ_2 = Q_1 \cdot Q_0' & JQ_1 = Q_2' \cdot Q_0 & JQ_0 = (Q_2 \oplus Q_1)' \\ KQ_2 = Q_1' \cdot Q_0' & KQ_1 = Q_2 \cdot Q_0 & KQ_0 = Q_2 \oplus Q_1 \end{array}$$



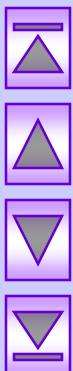
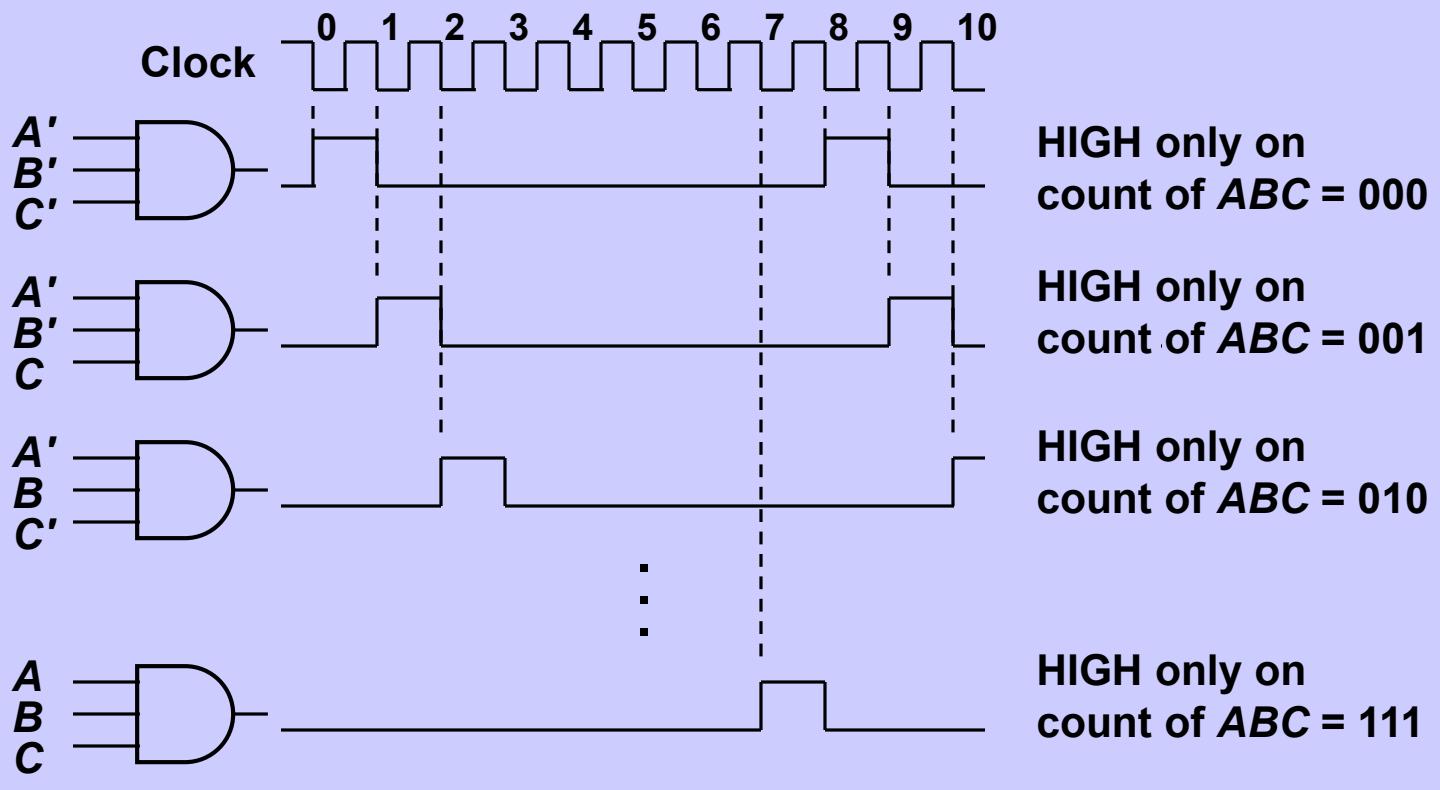
# Decoding A Counter

- Decoding a counter involves determining which state in the sequence the counter is in.
- Differentiate between *active-HIGH* and *active-LOW* decoding.
- Active-HIGH decoding: output HIGH if the counter is in the state concerned.
- Active-LOW decoding: output LOW if the counter is in the state concerned.



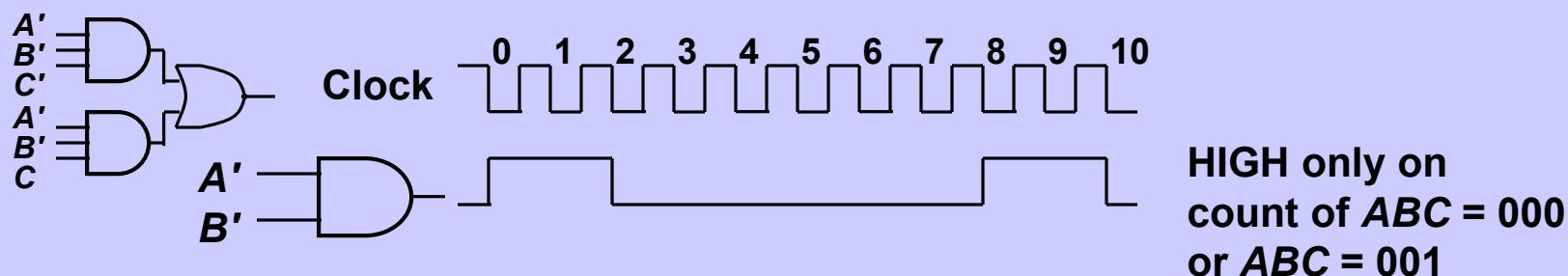
# Decoding A Counter

- Example: MOD-8 ripple counter (active-HIGH decoding).

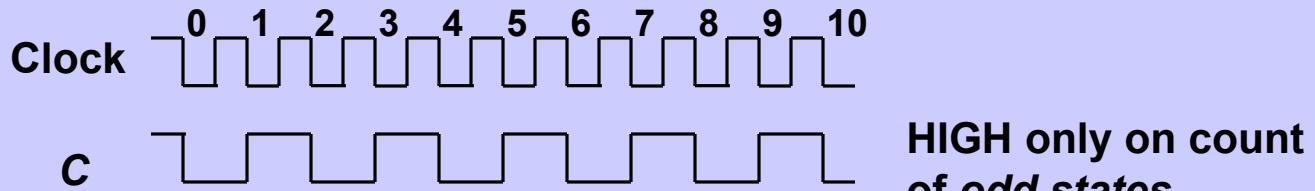


# Decoding A Counter

- Example: To detect that a MOD-8 counter is in state 0 (000) or state 1 (001).

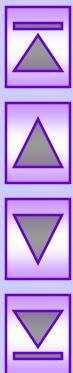


- Example: To detect that a MOD-8 counter is in the odd states (states 1, 3, 5 or 7), simply use  $C$ .



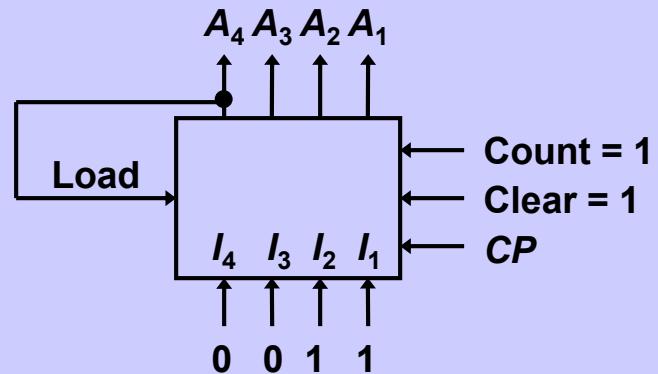
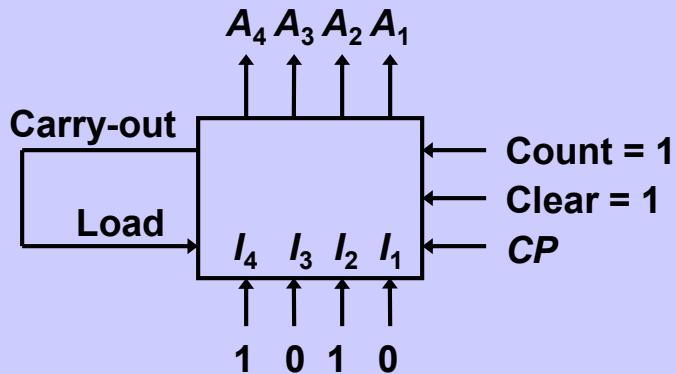
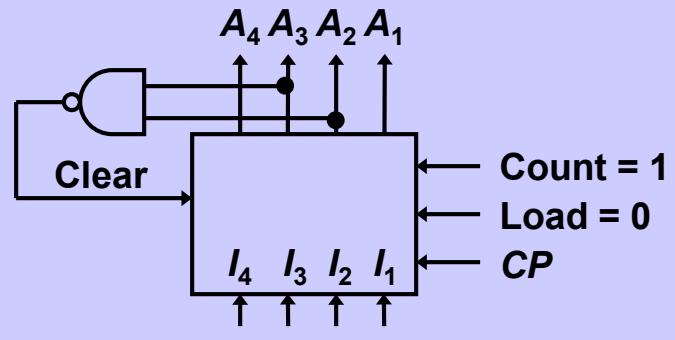
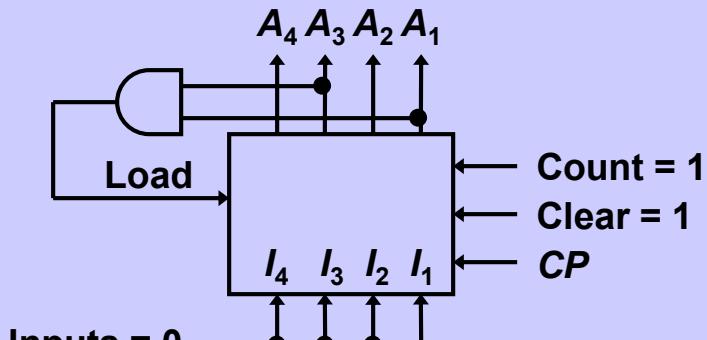
# Counters with Parallel Load

- Counters could be augmented with parallel load capability for the following purposes:
  - ❖ To start at a different state
  - ❖ To count a different sequence
  - ❖ As more sophisticated register with increment/decrement functionality.



# Counters with Parallel Load

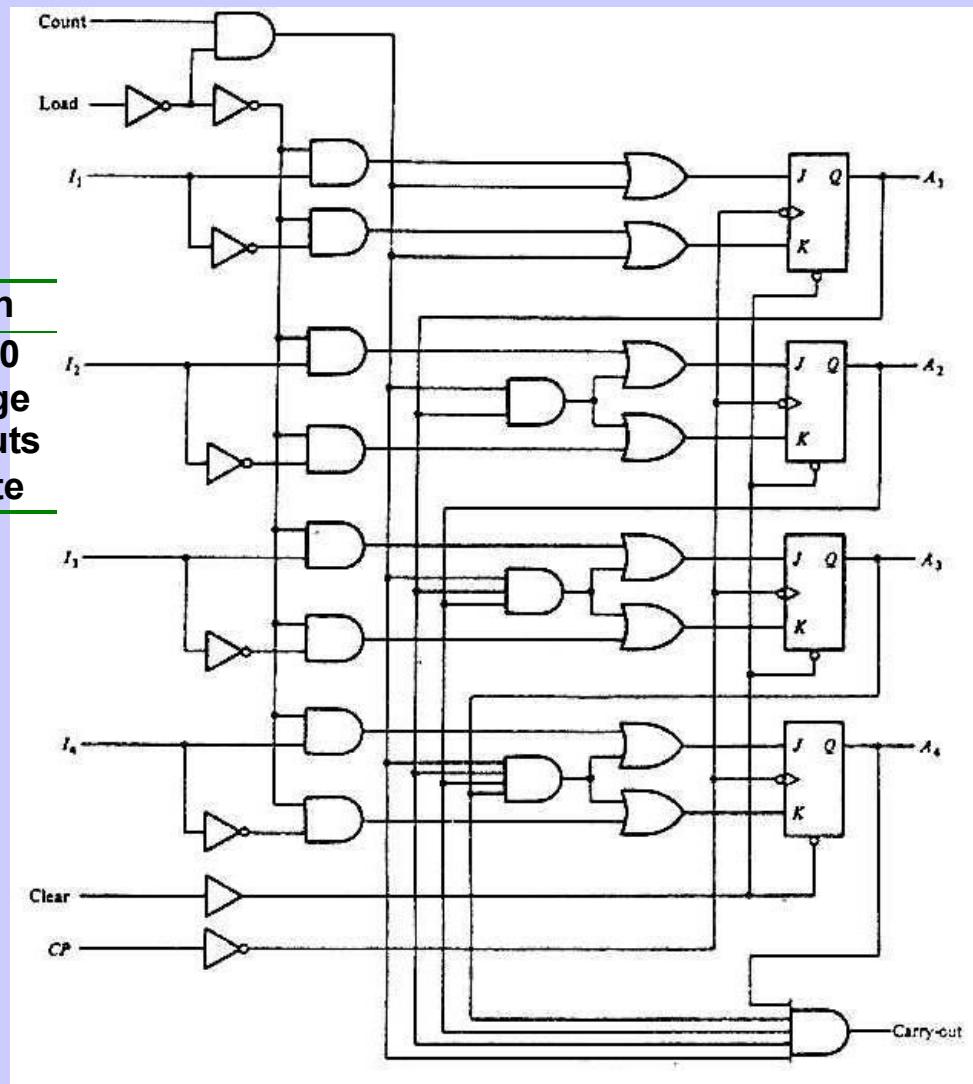
- Different ways of getting a MOD-6 counter:



# Counters with Parallel Load

- 4-bit counter with parallel load.

Clear	CP	Load	Count	Function
0	X	X	X	Clear to 0
1	X	0	0	No change
1	↑	1	X	Load inputs
1	↑	0	1	Next state



Counters with Parallel Load

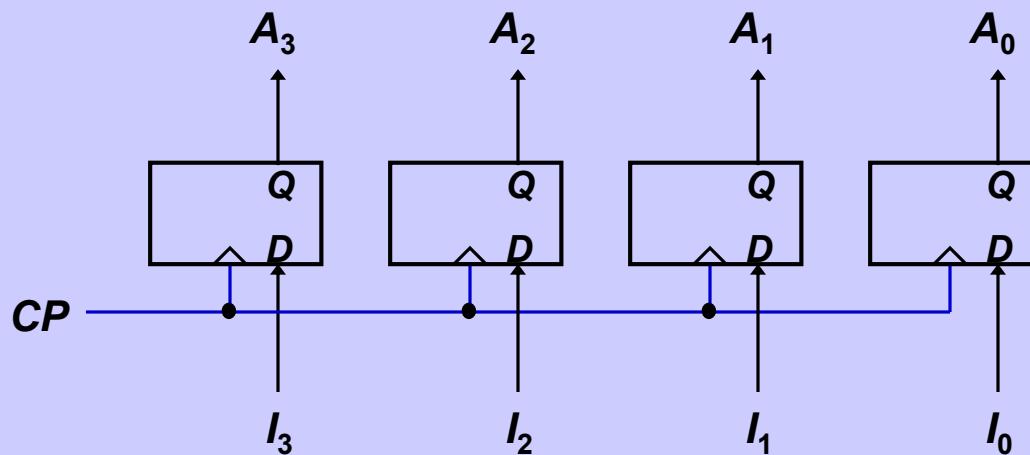
# Introduction: Registers

- An *n-bit register* has a group of  $n$  flip-flops and some logic gates and is capable of storing  $n$  bits of information.
- The flip-flops store the information while the gates control when and how new information is transferred into the register.
- Some functions of register:
  - ❖ retrieve data from register
  - ❖ store/load new data into register (serial or parallel)
  - ❖ shift the data within register (left or right)



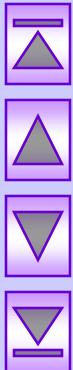
# Simple Registers

- No external gates.
- Example: A 4-bit register. A new 4-bit data is loaded every clock cycle.

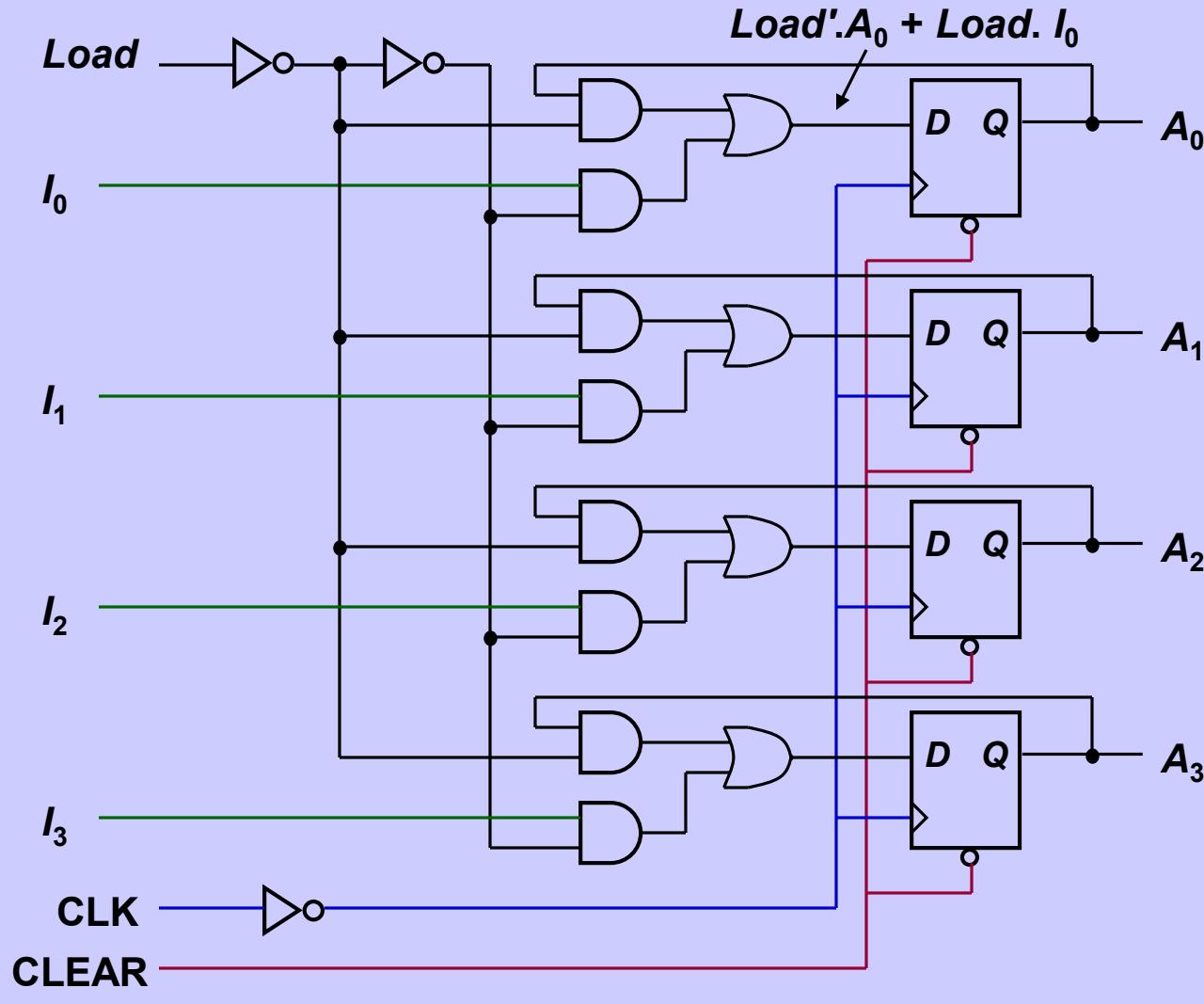


# Registers With Parallel Load

- Instead of loading the register at every clock pulse, we may want to control when to load.
- *Loading* a register: transfer new information into the register. Requires a *load* control input.
- *Parallel loading*: all bits are loaded simultaneously.

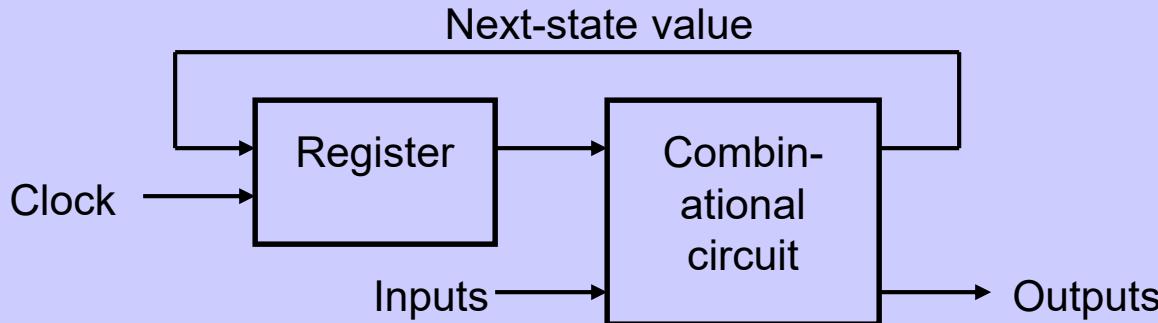


# Registers With Parallel Load



# Using Registers to implement Sequential Circuits

- A sequential circuit may consist of a *register* (memory) and a *combinational circuit*.



- The external inputs and present states of the register determine the next states of the register and the external outputs, through the combinational circuit.
- The combinational circuit may be implemented by any of the methods covered in *MSI components* and *Programmable Logic Devices*.

# Using Registers to implement Sequential Circuits

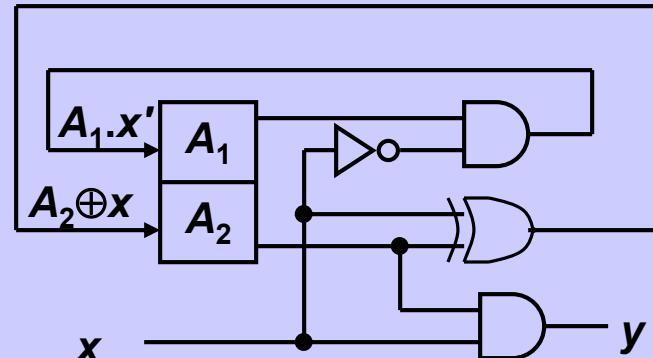
- Example 1:

$$A_1^+ = \sum m(4,6) = A_1 \cdot x'$$

$$A_2^+ = \sum m(1,2,5,6) = A_2 \cdot x' + A_2' \cdot x = A_2 \oplus x$$

$$y = \sum m(3,7) = A_2 \cdot x$$

Present state		Input $x$	Next State		Output $y$
$A_1$	$A_2$		$A_1^+$	$A_2^+$	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

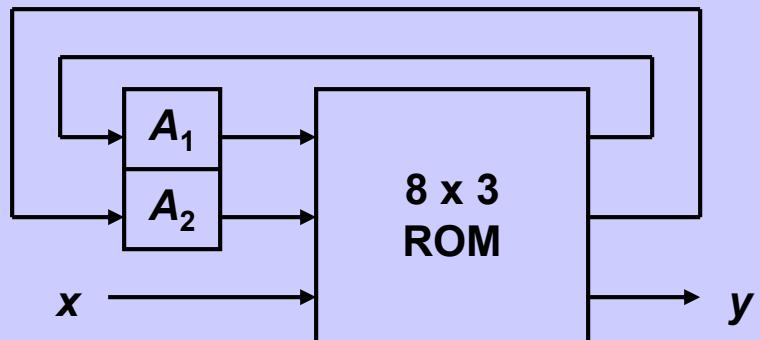


# Using Registers to implement Sequential Circuits

- Example 2: Repeat example 1, but use a ROM.

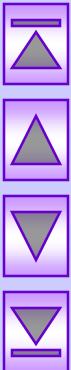
Address			Outputs		
1	2	3	1	2	3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

ROM truth table



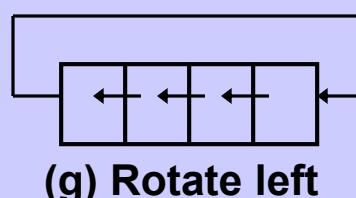
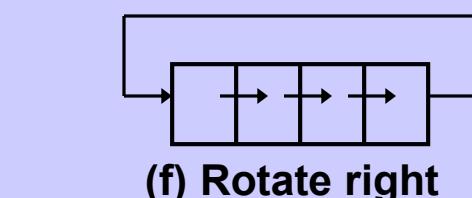
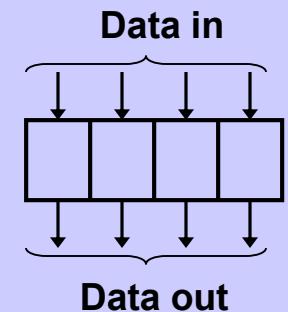
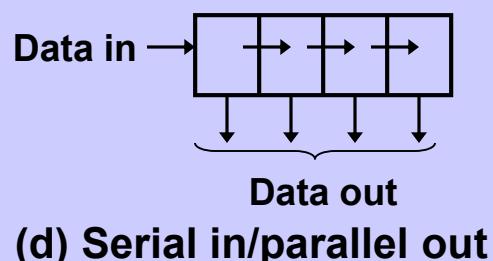
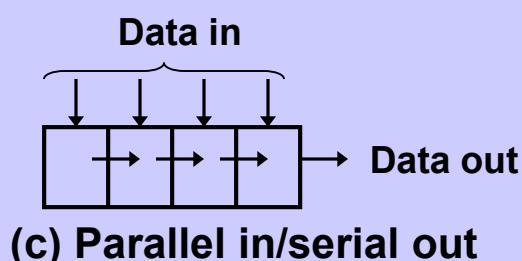
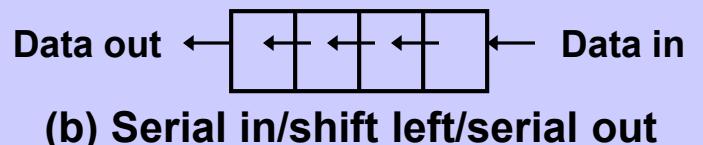
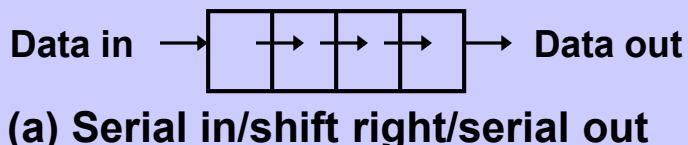
# Shift Registers

- Another function of a register, besides storage, is to provide for *data movements*.
- Each *stage* (flip-flop) in a shift register represents one bit of storage, and the shifting capability of a register permits the movement of data from stage to stage within the register, or into or out of the register upon application of clock pulses.



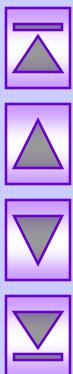
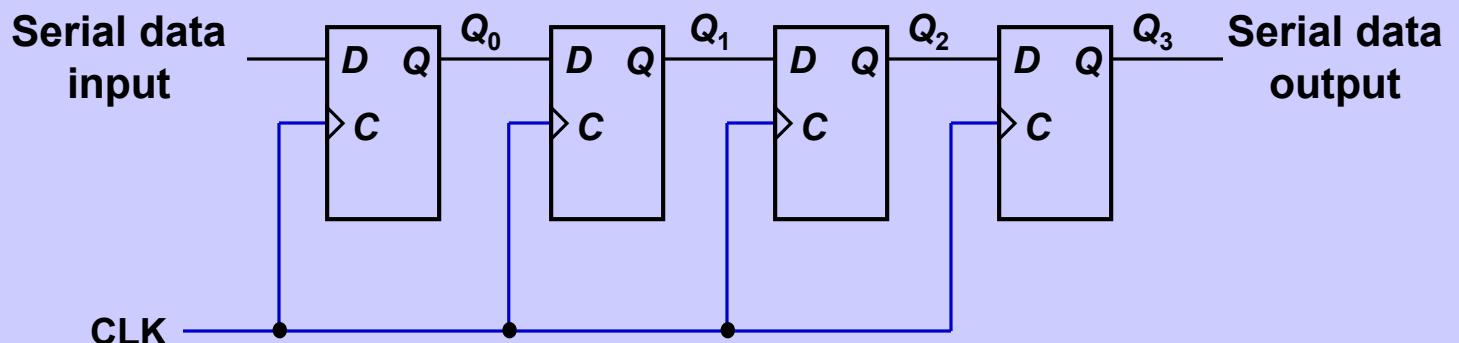
# Shift Registers

- Basic data movement in shift registers (four bits are used for illustration).



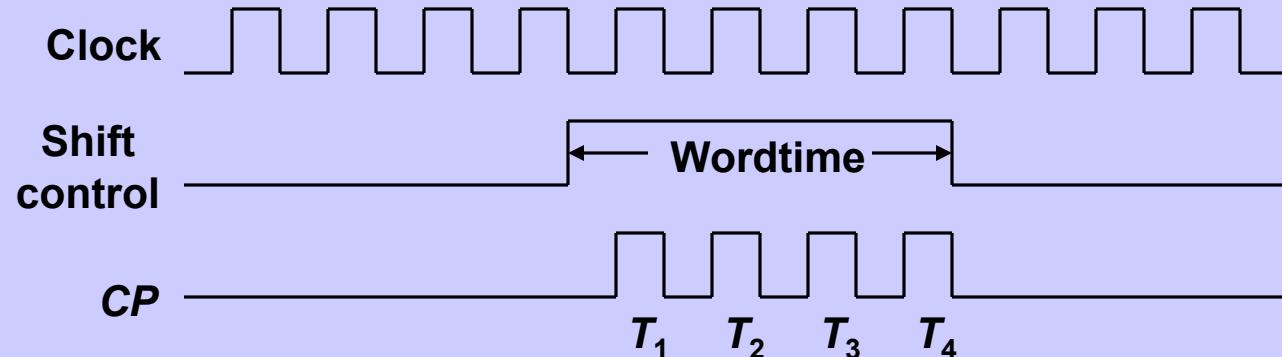
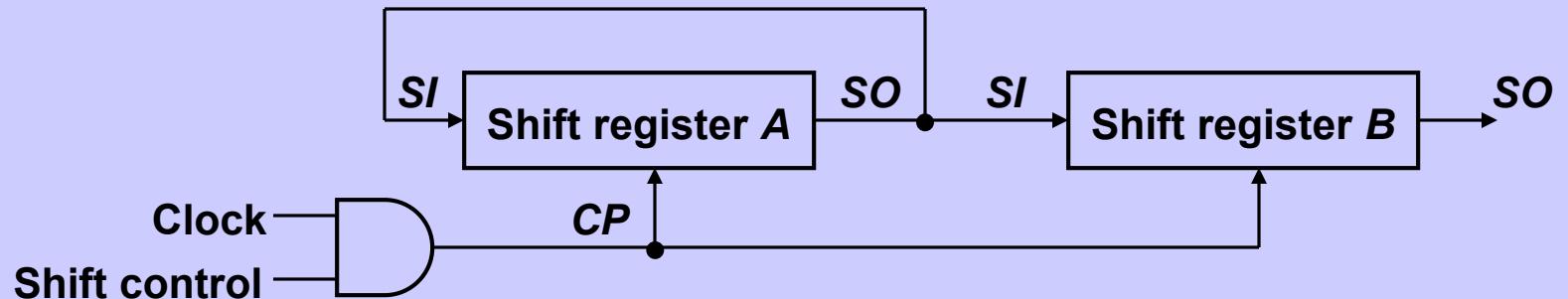
# Serial In/Serial Out Shift Registers

- Accepts data serially – one bit at a time – and also produces output serially.



# Serial In/Serial Out Shift Registers

- Application: Serial transfer of data from one register to another.



# Serial In/Serial Out Shift Registers

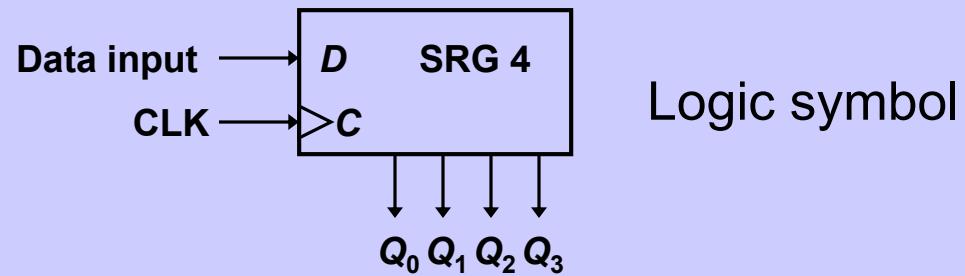
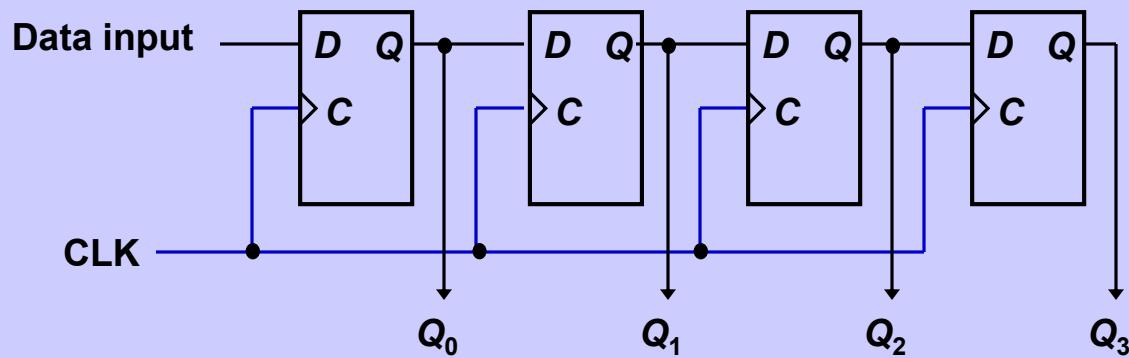
- Serial-transfer example.

Timing Pulse	Shift register A	Shift register B	Serial output of B
Initial value	1 0 1 1	0 0 1 0	0
After $T_1$	1 1 0 1	1 0 0 1	1
After $T_2$	1 1 1 0	1 1 0 0	0
After $T_3$	0 1 1 1	0 1 1 0	0
After $T_4$	1 0 1 1	1 0 1 1	1



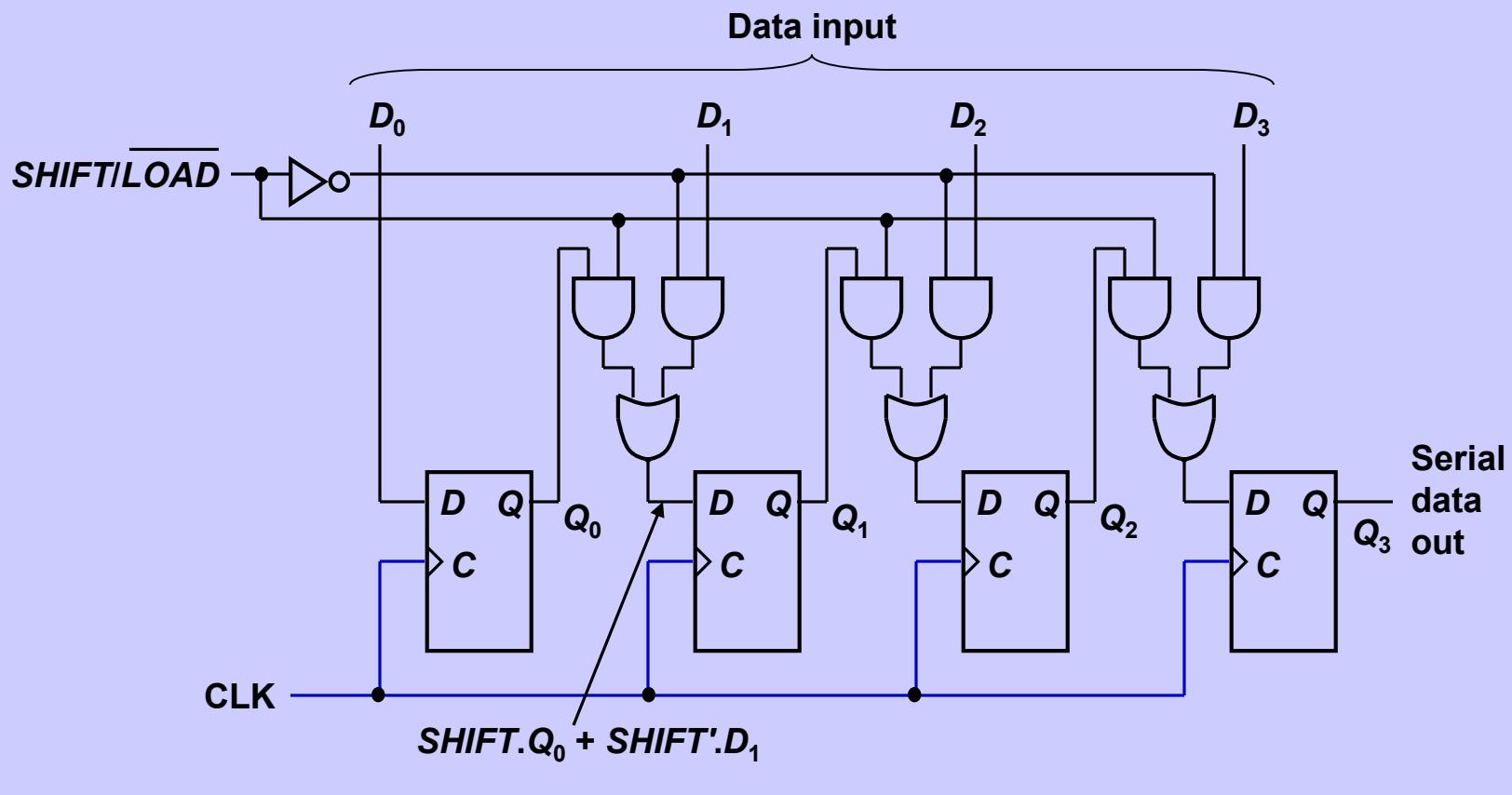
# Serial In/Parallel Out Shift Registers

- Accepts data serially.
- Outputs of all stages are available simultaneously.



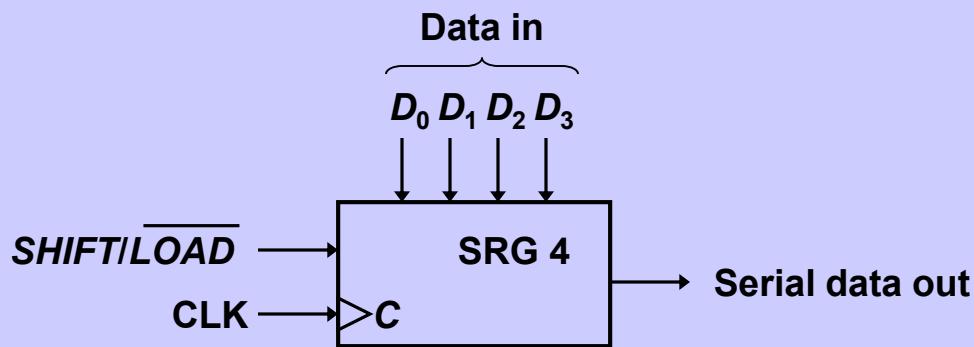
# Parallel In/Serial Out Shift Registers

- Bits are entered simultaneously, but output is serial.



# Parallel In/Serial Out Shift Registers

- Bits are entered simultaneously, but output is serial.

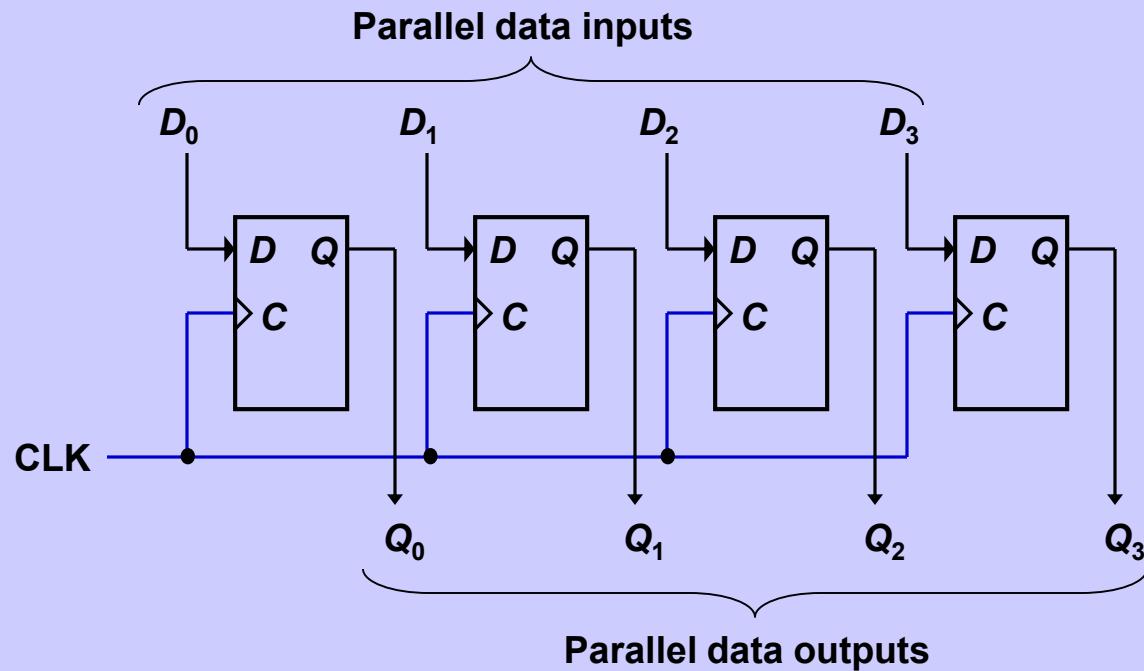


Logic symbol



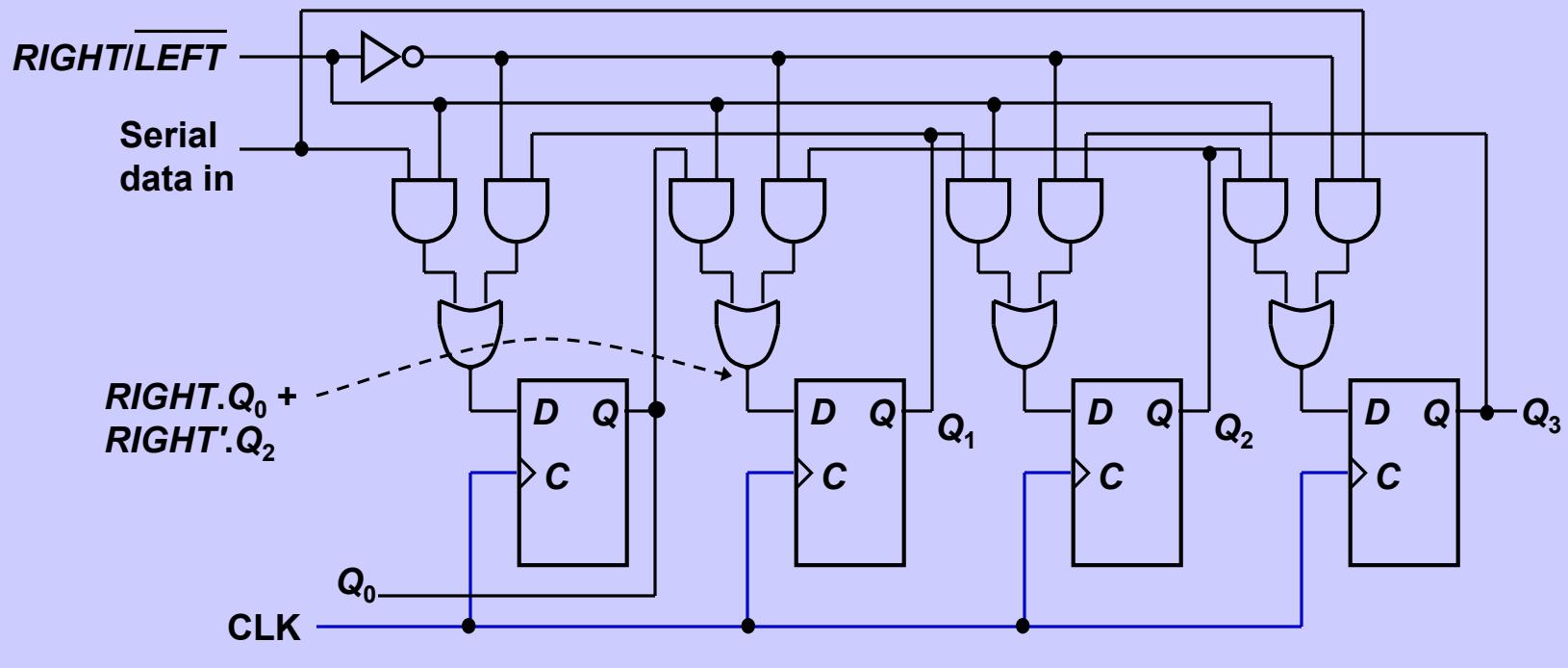
# Parallel In/Parallel Out Shift Registers

- Simultaneous input and output of all data bits.



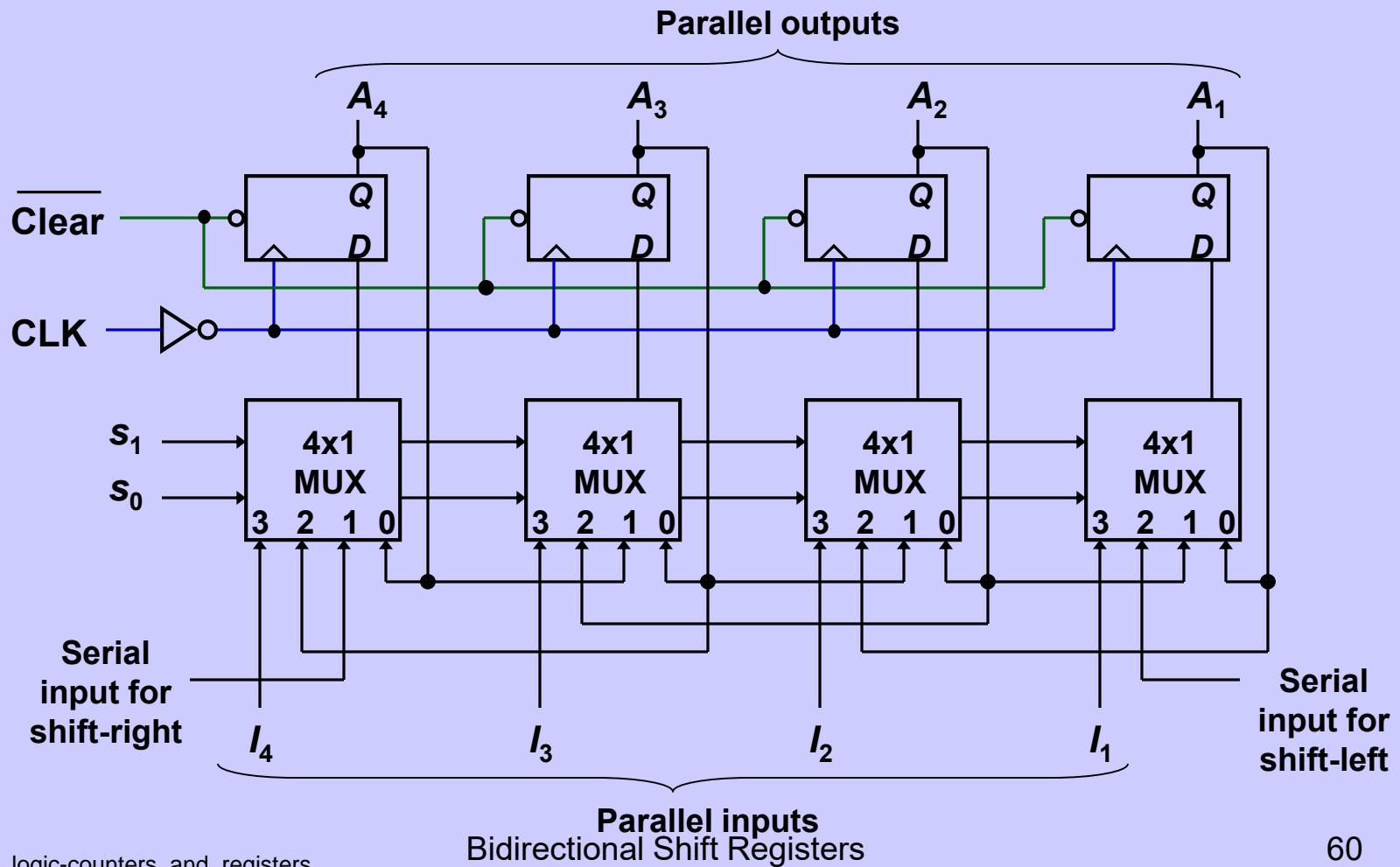
# Bidirectional Shift Registers

- Data can be shifted either left or right, using a control line *RIGHT/LEFT* (or simply *RIGHT*) to indicate the direction.



# Bidirectional Shift Registers

- 4-bit bidirectional shift register with parallel load.



# Bidirectional Shift Registers

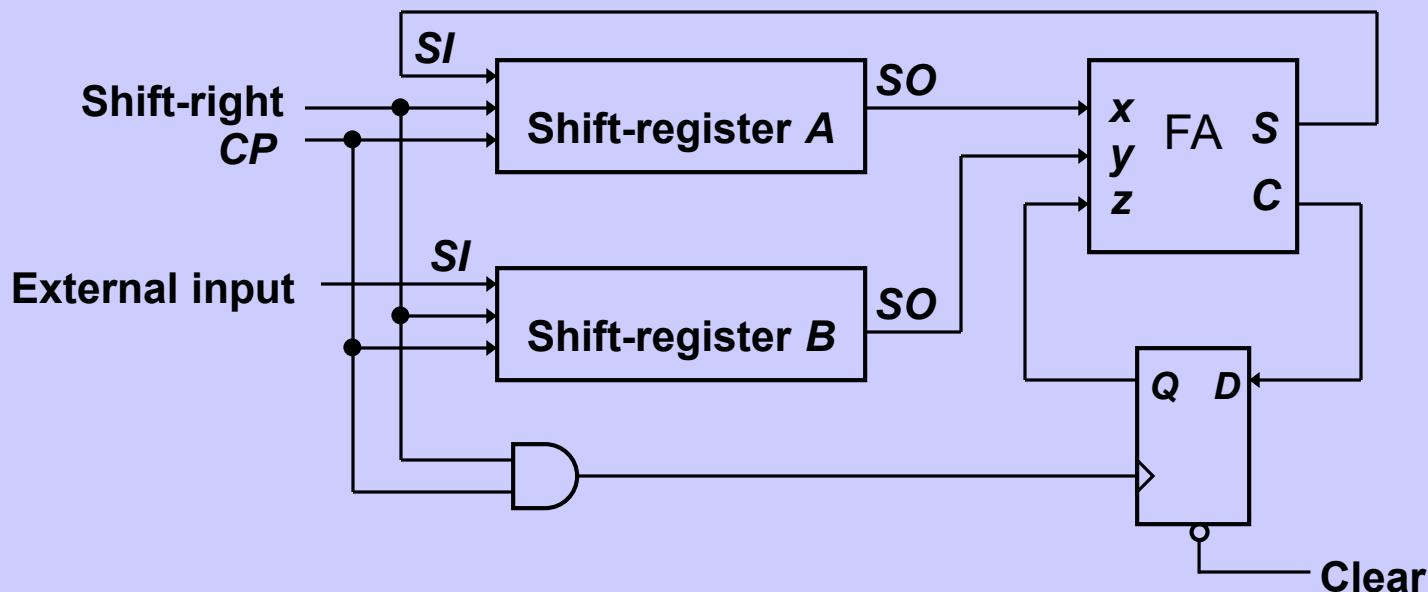
- 4-bit bidirectional shift register with parallel load.

<i>Mode Control</i>		<i>Register Operation</i>
$s_1$	$s_0$	
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load



# An Application – Serial Addition

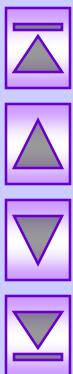
- Most operations in digital computers are done in parallel. Serial operations are slower but require less equipment.
- A serial adder is shown below.  $A \leftarrow A + B$ .



# An Application – Serial Addition

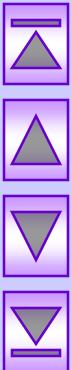
- $A = 0100; B = 0111$ .  $A + B = 1011$  is stored in  $A$  after 4 clock pulses.

Initial:	$A: 0\ 1\ 0\ 0$	$Q: \underline{0}$
	$B: 0\ 1\ 1\ 1$	
<hr/>		
Step 1: $0 + 1 + 0$	$A: 1\ 0\ 1\ 0$	$Q: \underline{0}$
$S = 1, C = 0$	$B: x\ 0\ 1\ 1$	
<hr/>		
Step 2: $0 + 1 + 0$	$A: 1\ 1\ 0\ 1$	$Q: \underline{0}$
$S = 1, C = 0$	$B: x\ x\ 0\ 1$	
<hr/>		
Step 3: $1 + 1 + 0$	$A: 0\ 1\ 1\ 0$	$Q: \underline{1}$
$S = 0, C = 1$	$B: x\ x\ x\ \underline{0}$	
<hr/>		
Step 4: $0 + 0 + 1$	$A: 1\ 0\ 1\ 1$	$Q: \underline{0}$
$S = 1, C = 0$	$B: x\ x\ x\ x$	



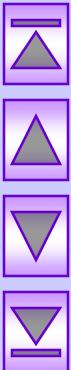
# Shift Register Counters

- Shift register counter: a shift register with the serial output connected back to the serial input.
- They are classified as counters because they give a specified sequence of states.
- Two common types: the *Johnson counter* and the *Ring counter*.



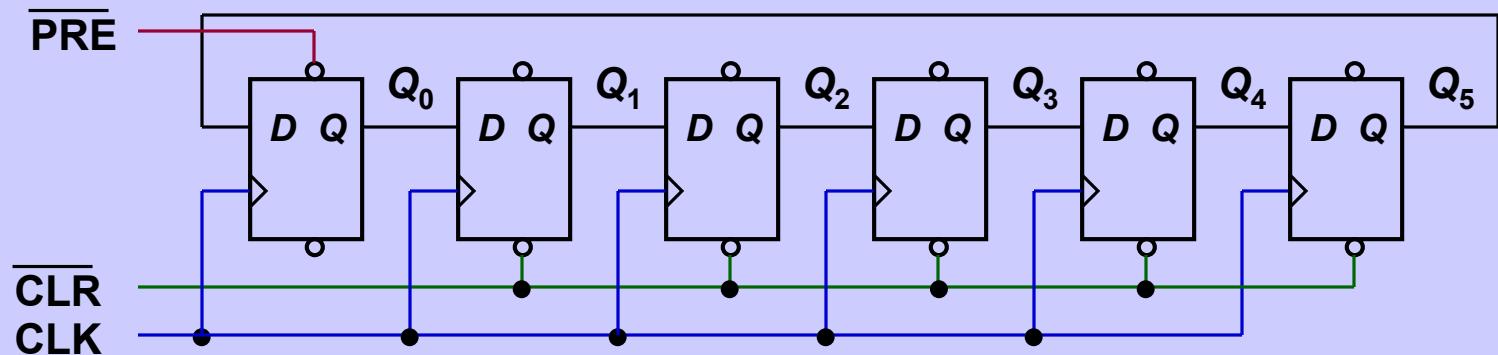
# Ring Counters

- One flip-flop (stage) for each state in the sequence.
- The output of the last stage is connected to the D input of the first stage.
- An  $n$ -bit ring counter cycles through  $n$  states.
- No decoding gates are required, as there is an output that corresponds to every state the counter is in.

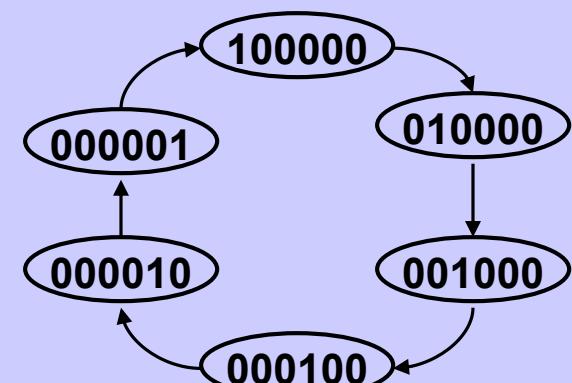


# Ring Counters

- Example: A 6-bit (MOD-6) ring counter.

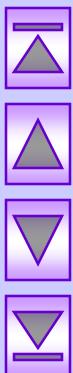


Clock	$Q_0$	$Q_1$	$Q_2$	$Q_3$	$Q_4$	$Q_5$
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	0	1



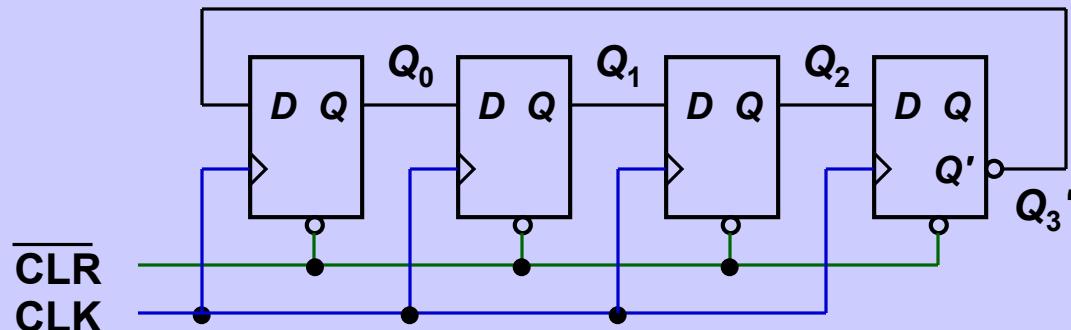
# Johnson Counters

- The complement of the output of the last stage is connected back to the D input of the first stage.
- Also called the *twisted-ring counter*.
- Require fewer flip-flops than ring counters but more flip-flops than binary counters.
- An  $n$ -bit Johnson counter cycles through  $2n$  states.
- Require more decoding circuitry than ring counter but less than binary counters.

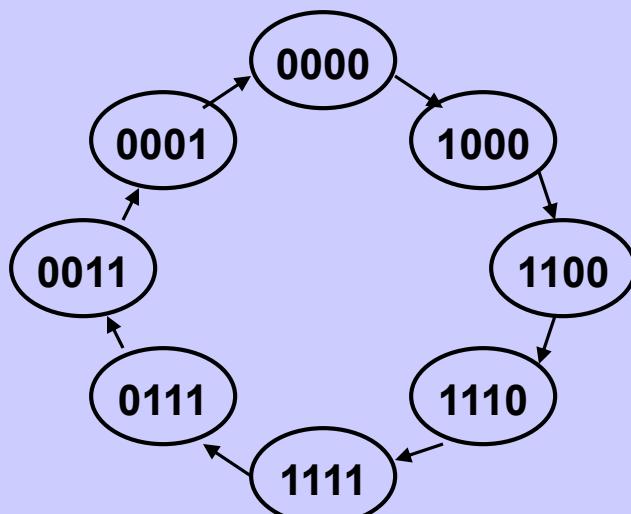


# Johnson Counters

- Example: A 4-bit (MOD-8) Johnson counter.



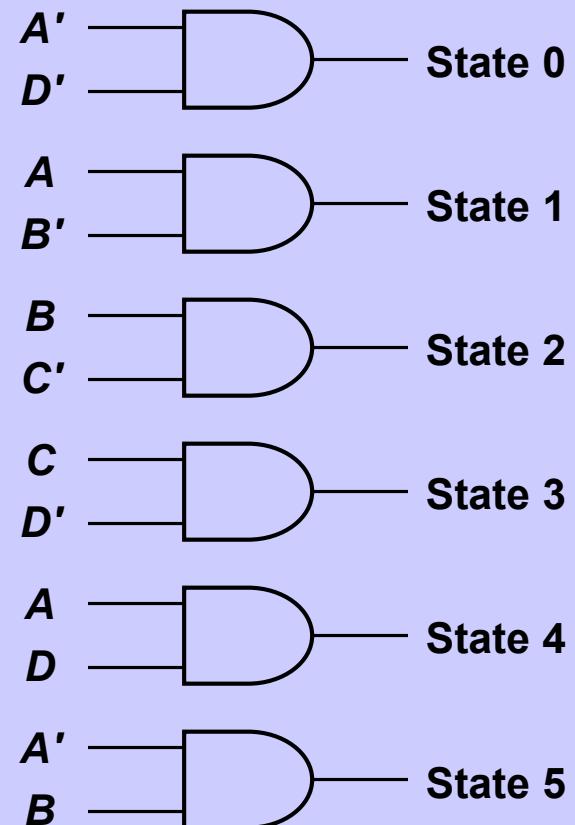
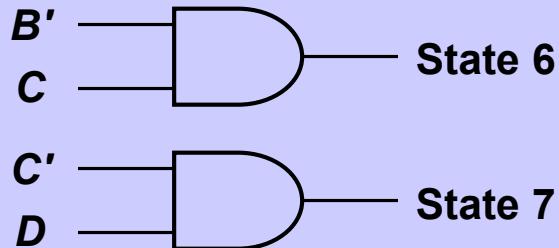
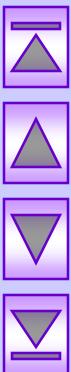
Clock	$Q_0$	$Q_1$	$Q_2$	$Q_3$
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1



# Johnson Counters

- Decoding logic for a 4-bit Johnson counter.

Clock	A	B	C	D	Decoding
0	0	0	0	0	$A'D'$
1	1	0	0	0	$A.B'$
2	1	1	0	0	$B.C'$
3	1	1	1	0	$C.D'$
4	1	1	1	1	$A.D$
5	0	1	1	1	$A'.B$
6	0	0	1	1	$B'.C$
7	0	0	0	1	$C'.D$

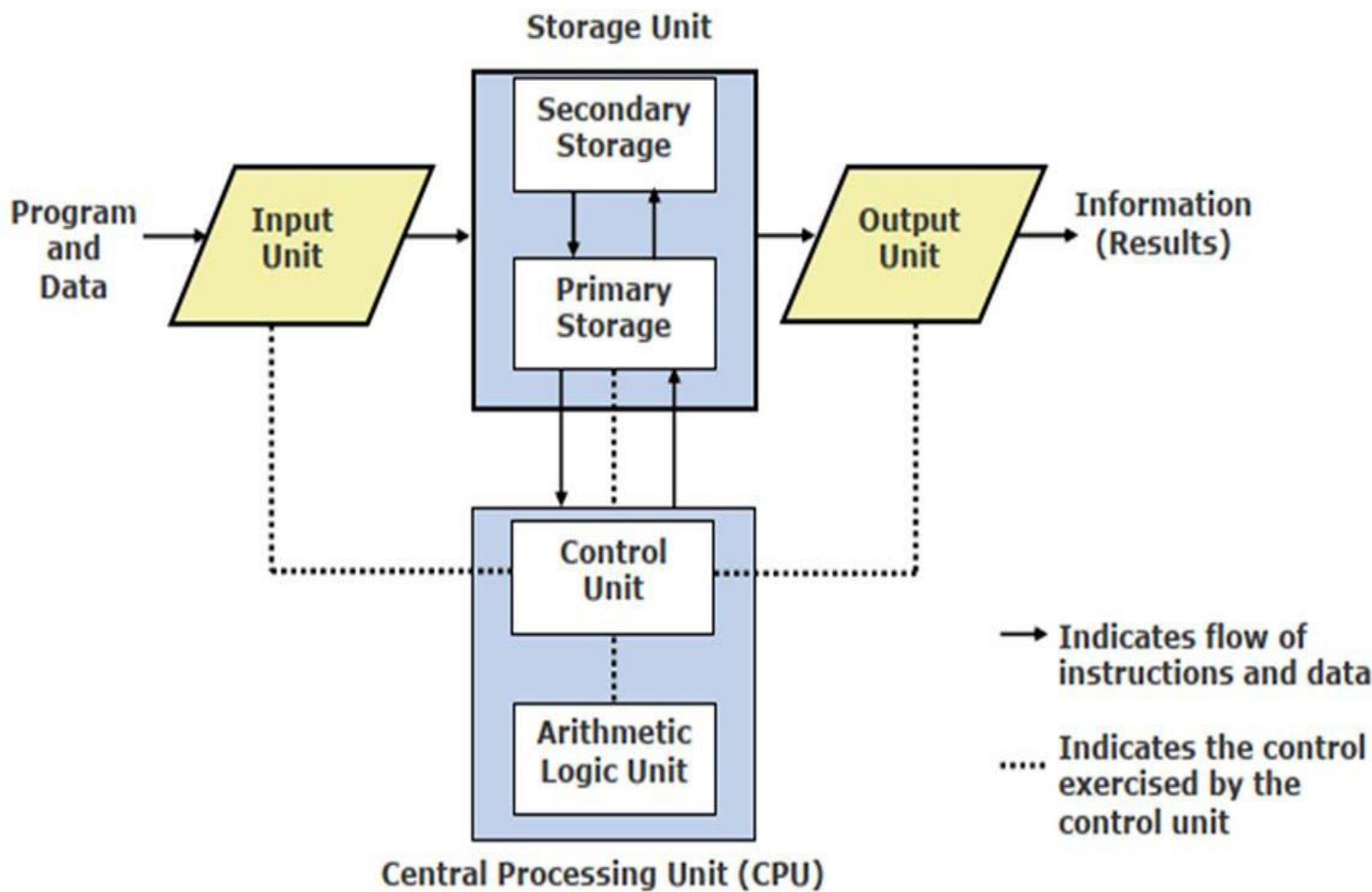


---

# Introduction

1. Digital circuits are frequently used for arithmetic operations
2. Fundamental arithmetic operations on binary numbers and digital circuits which perform arithmetic operations will be examined.
3. HDL will be used to describe arithmetic circuits.
4. An arithmetic/logic unit (ALU) accepts data stored in memory and executes arithmetic and logic operations as instructed by the control unit.

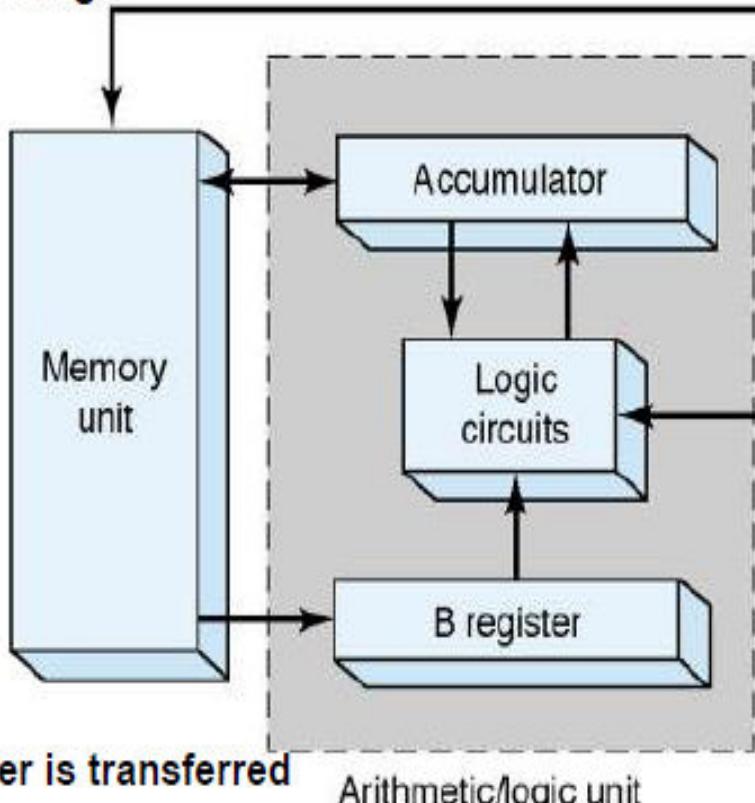
# Basic Organization of a Computer System



# Arithmetic Circuits

The new number remains in the accumulator for further operations or can be transferred to memory for storage

4



2

The number is transferred from memory to the B register

3

Number in B register and accumulator register are added in the logic circuit, with sum sent to accumulator for storage.

1

Control unit is instructed to add a specific number from a memory location to a number stored in the accumulator register.

# Datapaths

- We'll focus on **computer architecture**: how to assemble the combinational and sequential components we've studied so far into a complete computer.
- The **datapath** is the part of the central processing unit (CPU) that does the actual computations.



# Keeping it simple!

- Abstraction is very helpful in understanding processors.
  - Although we studied how devices like registers and muxes are built, we don't need that level of detail here.
  - You should focus more on what these component devices are doing, and less on how they work.
- Otherwise, it's easy to get bogged down in the details, and datapath and control units can be a little intimidating.

# An overview of CPU design

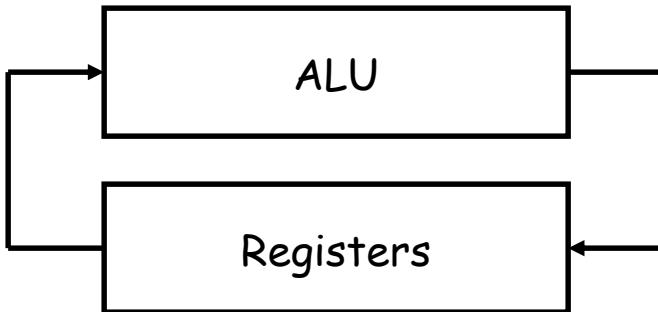
- We can divide the design of our CPU into three parts:
  - The **datapath** does all of the actual data processing.
  - An **instruction set** is the programmer's interface to CPU.
  - A **control unit** uses the programmer's instructions to tell the datapath what to do.
- We'll look in detail at a processor's datapath, which is responsible for doing all of the dirty work.
  - An **ALU** does computations, as we've seen before.
  - A limited set of registers serve as fast temporary storage.
  - A larger, but slower, random-access memory is also available.

# What's in a CPU?



- A processor is just one big sequential circuit.
  - Some registers are used to store values, which form the state.
  - An ALU performs various operations on the data stored in the registers.

# Register transfers



- Fundamentally, the processor is just moving data between registers, possibly with some ALU computations.
- To describe this data movement more precisely, we'll use a **register transfer language**.
  - The objects in the language are *registers*.
  - The basic operations are *transfers*, where data is copied from one register to another.
- We can also use the ALU to perform arithmetic operations on the data while we're transferring it.

# Register transfer language review (from Chapter 8)

- Two-character names denote **registers**, such as R0, R1, DR, or SA.
- Arrows indicate **data transfers**. To copy the contents of the **source register** R2 into the **destination register** R1 in one clock cycle:

R1 ← R2

- A **conditional transfer** is performed only if the Boolean condition in front of the colon is true. To transfer R3 to R2 when K = 1:

K: R2 ← R3

- **Multiple transfers** on the same clock cycle are separated by commas.

R1 ← R2, K: R2 ← R3

- Don't confuse this register transfer language with assembly language, which we'll discuss later.

# Register transfer operations (cont'd)

- We can apply **arithmetic operations** to registers.

$$R1 \leftarrow R2 + R3$$

$$R3 \leftarrow R1 - 1$$

- Logical operations** are applied bitwise. AND and OR are denoted with special symbols, to prevent confusion with arithmetic operations.

$$R2 \leftarrow R1 \wedge R2 \quad \text{bitwise AND}$$

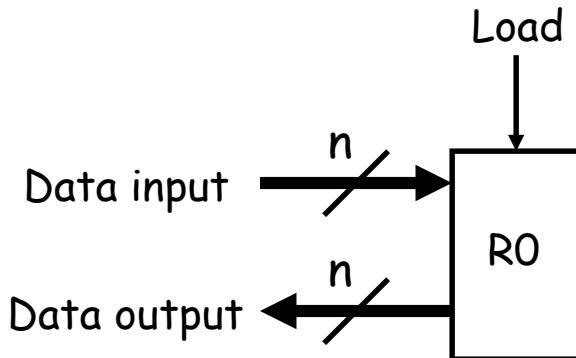
$$R3 \leftarrow R0 \vee R1 \quad \text{bitwise OR}$$

- Lastly, we can **shift** registers. Here, the source register R1 is not modified, and we assume that the shift input is just 0.

$$R2 \leftarrow sl\ R1 \quad \text{left shift}$$

$$R2 \leftarrow sr\ R1 \quad \text{right shift}$$

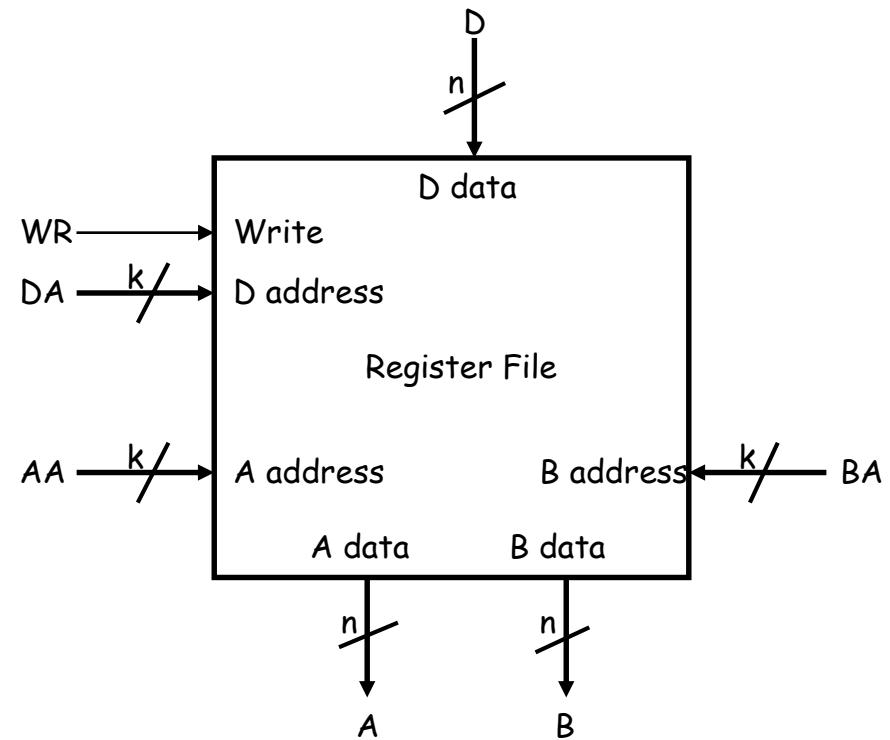
# Block symbols for registers



- We'll use this block diagram to represent an  $n$ -bit register.
- There is a data input and a load input.
  - When  $\text{Load} = 1$ , the data input is stored into the register.
  - When  $\text{Load} = 0$ , the register will keep its current value.
- The register's contents are always available on the output lines, regardless of the Load input.
- The clock signal is not shown because it would make the diagram messy.
- Remember that the input and output lines are actually  $n$  bits wide!

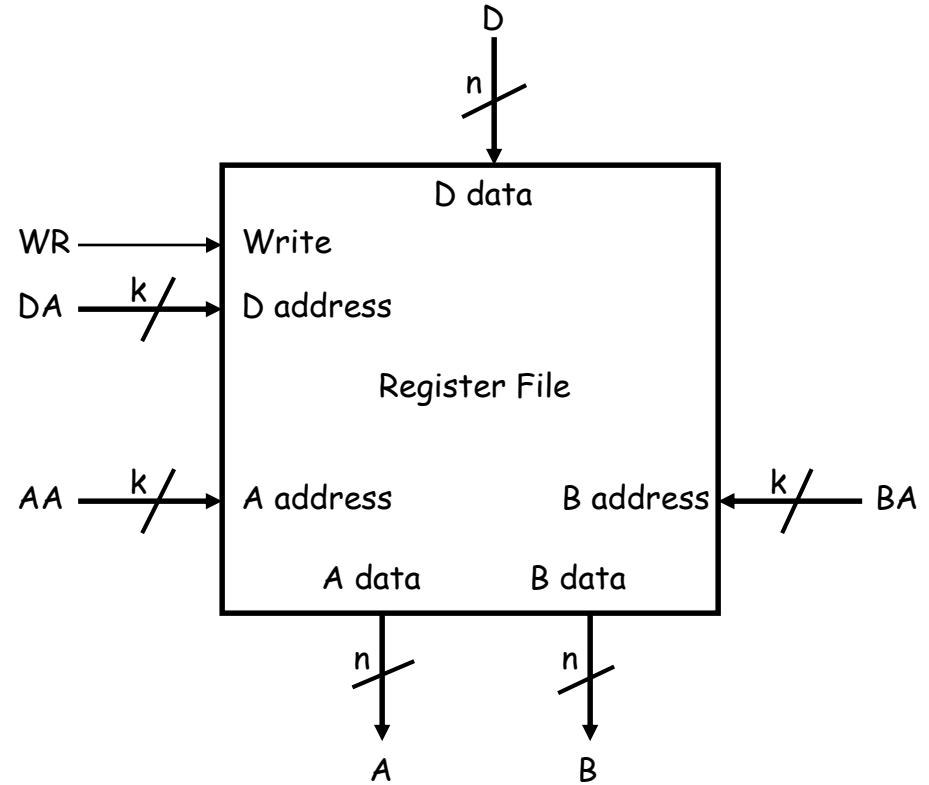
# Register file

- Modern processors have a number of registers grouped together in a **register file**.
- Much like words stored in a RAM, individual registers are identified by an address.
- Here is a block symbol for a  $2^k \times n$  register file.
  - There are  $2^k$  registers, so register addresses are  $k$  bits long.
  - Each register holds an  $n$ -bit word, so the data inputs and outputs are  $n$  bits wide.



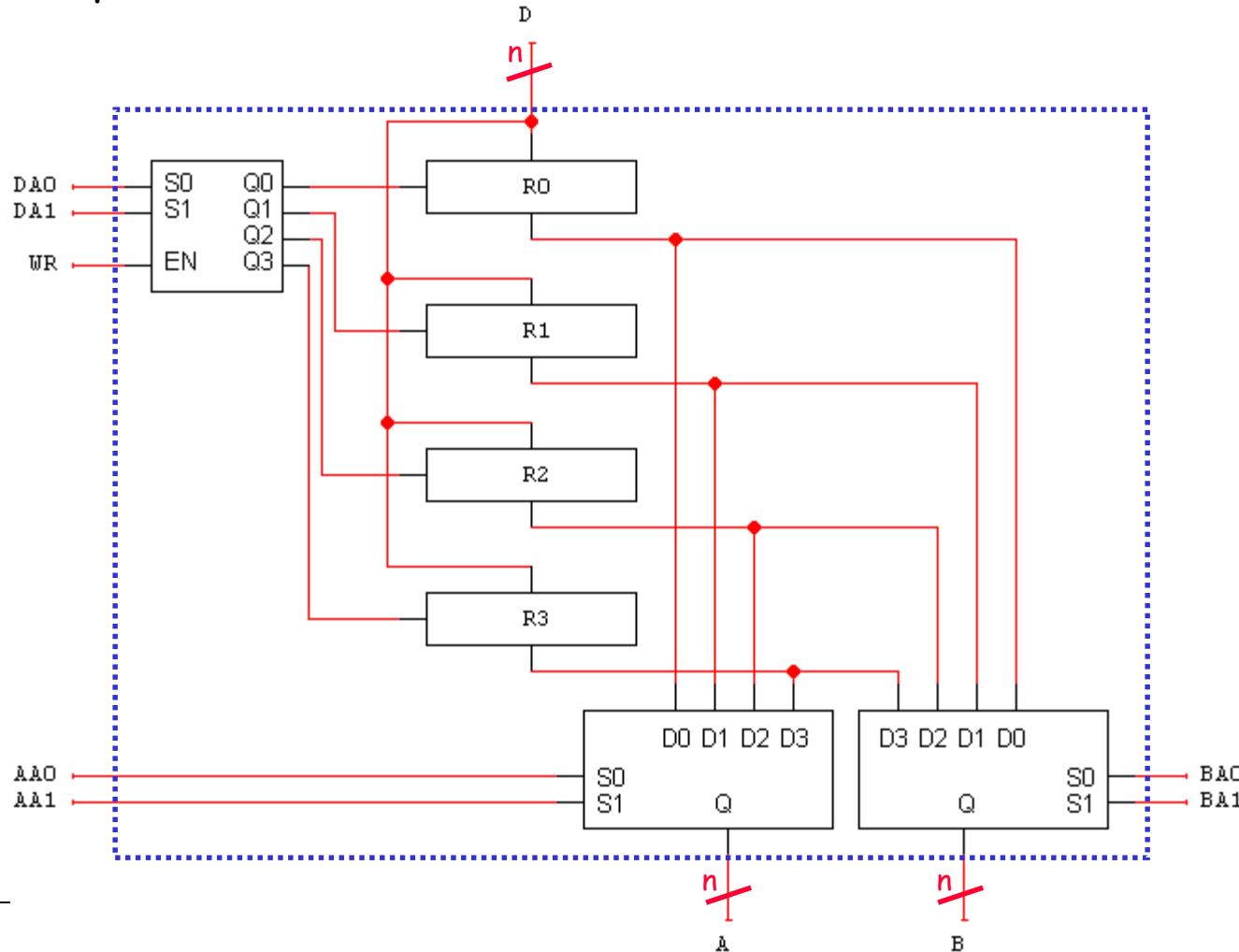
# Accessing the register file

- You can read two registers at once by supplying the AA and BA inputs. The data appears on the A and B outputs.
- You can write to a register by using the DA and D inputs, and setting WR = 1.
- These are registers so there must be a clock signal, even though we usually don't show it in diagrams.
  - We can read from the register file at any time.
  - Data is written only on the positive edge of the clock.



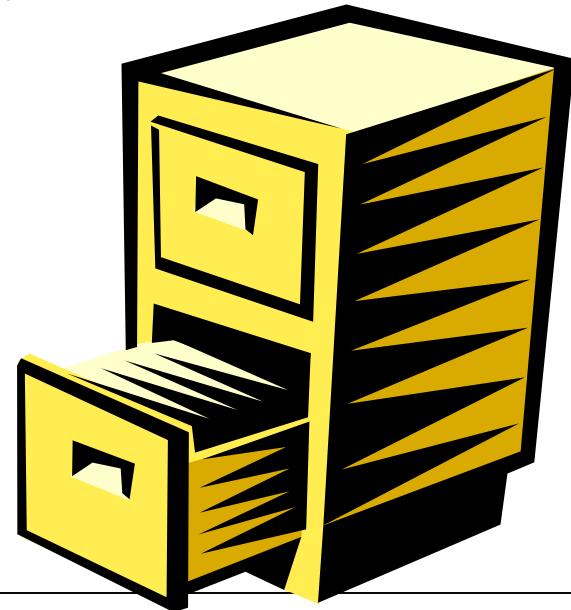
# What's inside the register file

- Here's a  $4 \times n$  register file. (We'll assume a  $4 \times n$  register file for all our examples.)



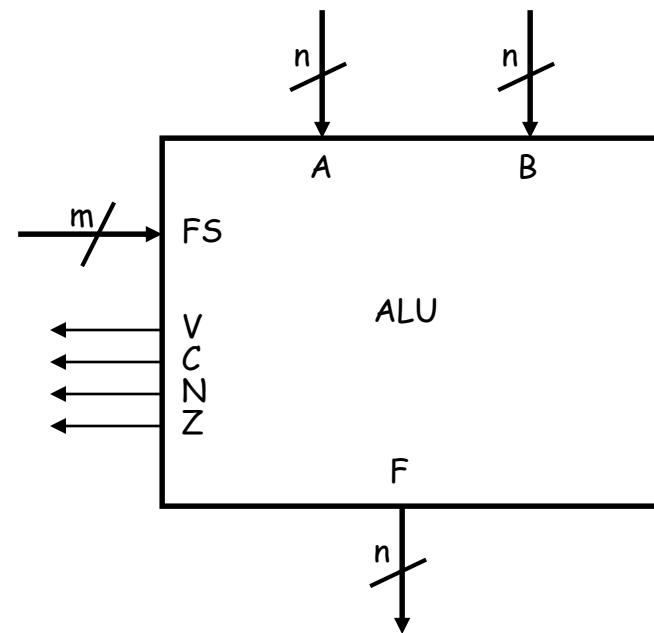
# Explaining the register file

- The 2-to-4 decoder selects one of the four registers for writing. If  $WR = 1$ , the decoder will be enabled and one of the Load signals will be active.
- The n-bit 4-to-1 muxes select the two register file outputs A and B, based on the inputs AA and BA.
- We need to be able to read two registers at once because most arithmetic operations require two operands.



# The all-important ALU

- The main job of a central processing unit is to “process,” or to perform computations....remember the ALU from way back when?
- We’ll use the following general block symbol for the ALU.
  - **A** and **B** are two n-bit numeric inputs.
  - **FS** is an m-bit function select code, which picks one of  $2^m$  functions.
  - The n-bit result is called **F**.
  - Several **status bits** provide more information about the output F:
    - **V = 1** in case of signed overflow.
    - **C** is the carry out.
    - **N = 1** if the result is negative.
    - **Z = 1** if the result is 0.



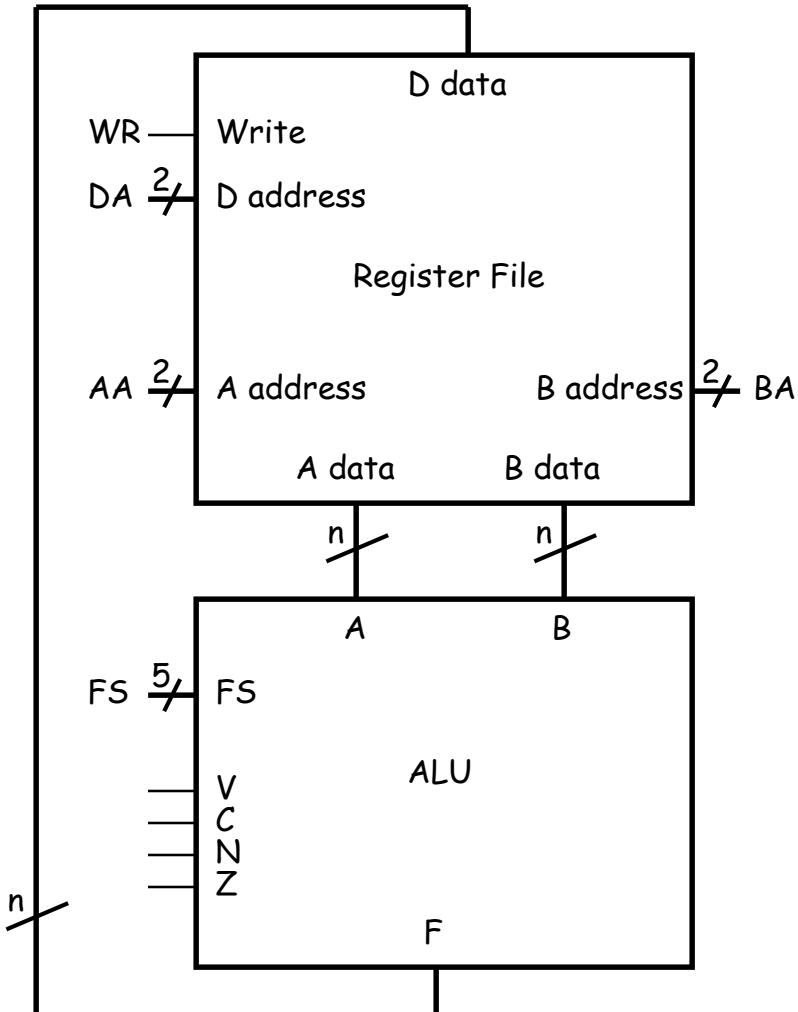
# ALU functions

- For concrete examples, we'll use the ALU as it's presented in the textbook.
- The table of operations on the right is taken from the book.
- The function select code FS is 5 bits long, but there are only 15 different functions here.
- We use an alternative notation for AND and OR to avoid confusion with arithmetic operations.

FS	Operation
00000	$F = A$
00001	$F = A + 1$
00010	$F = A + B$
00011	$F = A + B + 1$
00100	$F = A + B'$
00101	$F = A + B' + 1$
00110	$F = A - 1$
00111	$F = A$
01000	$F = A \wedge B$ (AND)
01010	$F = A \vee B$ (OR)
01100	$F = A \oplus B$
01110	$F = A'$
10000	$F = B$
10100	$F = sr\ B$ (shift right)
11000	$F = sl\ B$ (shift left)

# Our first datapath

- Here is the most basic datapath.
  - The ALU's two data inputs come from the register file.
  - The ALU computes a result, which is saved back to the registers.
- WR, DA, AA, BA and FS are control signals. Their values determine the exact actions taken by the datapath—which registers are used and for what operation.
- Remember that many of the signals here are actually multi-bit values.

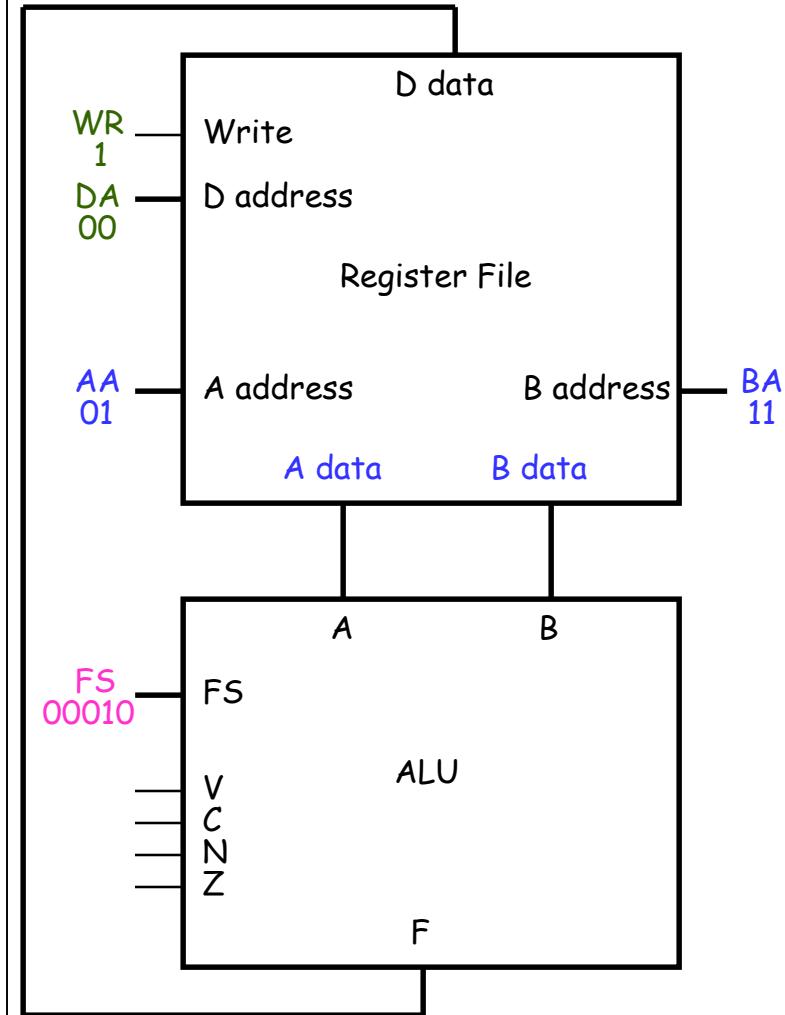


# An example computation

- Let's look at the proper control signals for the operation below:

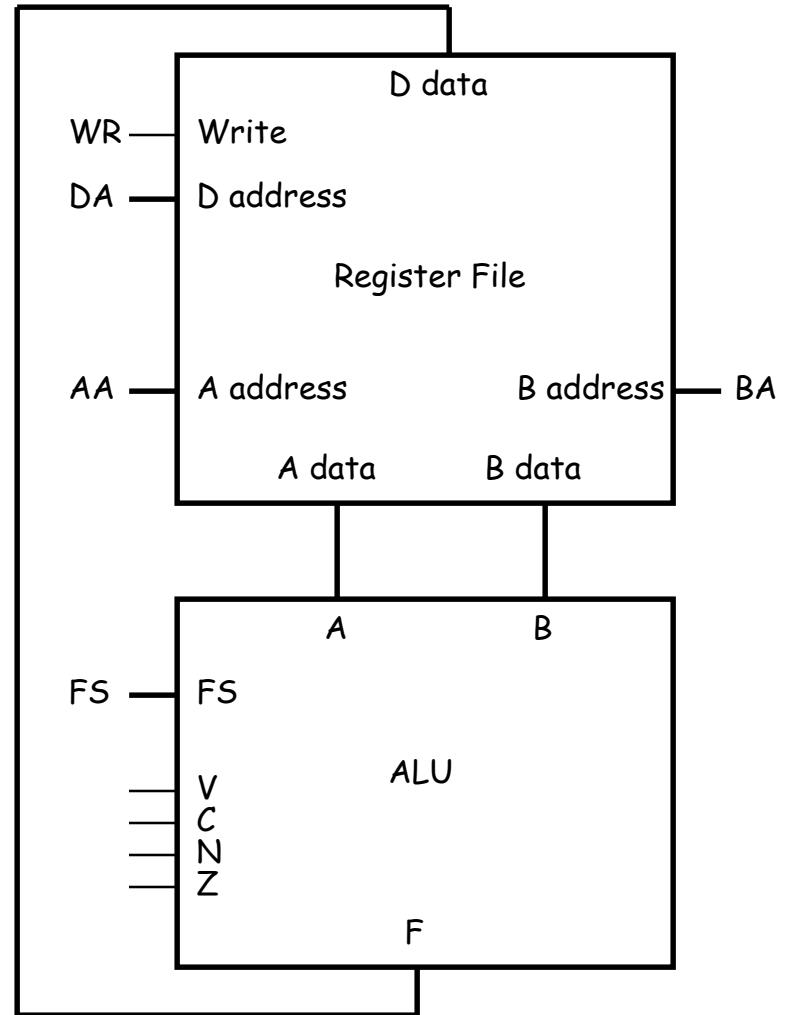
$$R0 \leftarrow R1 + R3$$

- Set **AA = 01** and **BA = 11**. This causes the contents of R1 to appear at **A data**, and the contents of R3 to appear at **B data**.
- Set the ALU's function select input **FS = 00010** ( $A + B$ ).
- Set **DA = 00** and **WR = 1**. On the next positive clock edge, the ALU result ( $R1 + R3$ ) will be stored in R0.



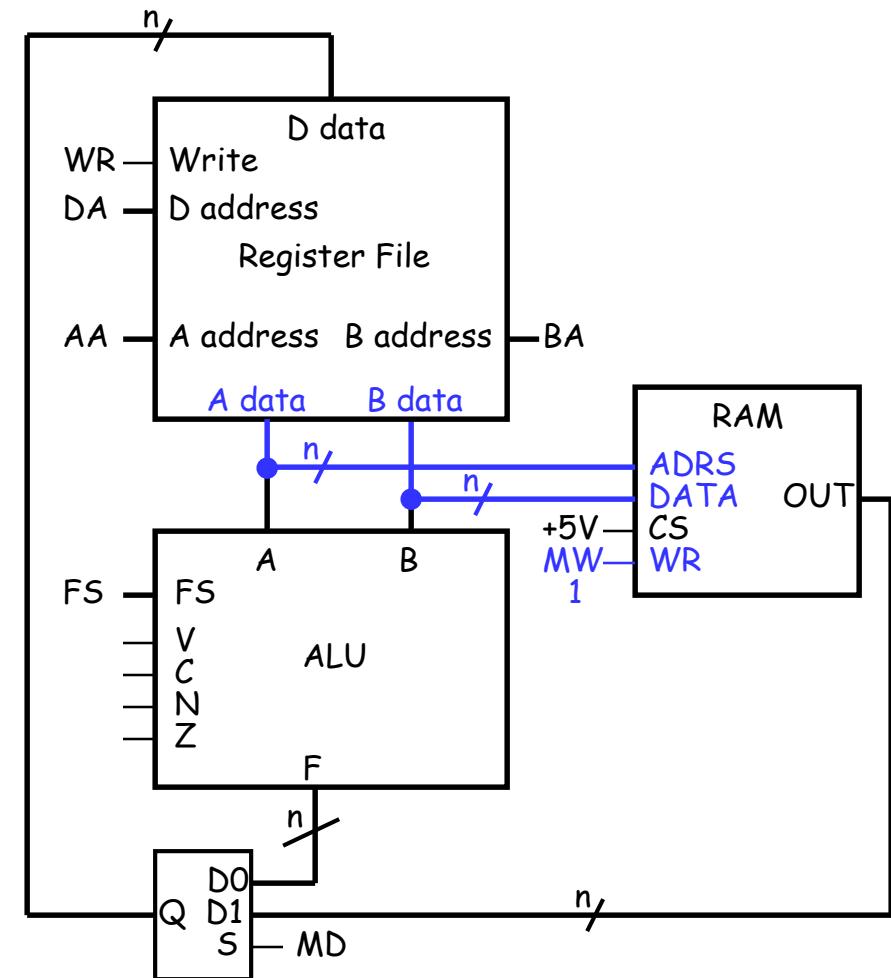
# Two questions

- Four registers isn't a lot. What if we need more storage?
- Who exactly decides which registers are read and written and which ALU function is executed?



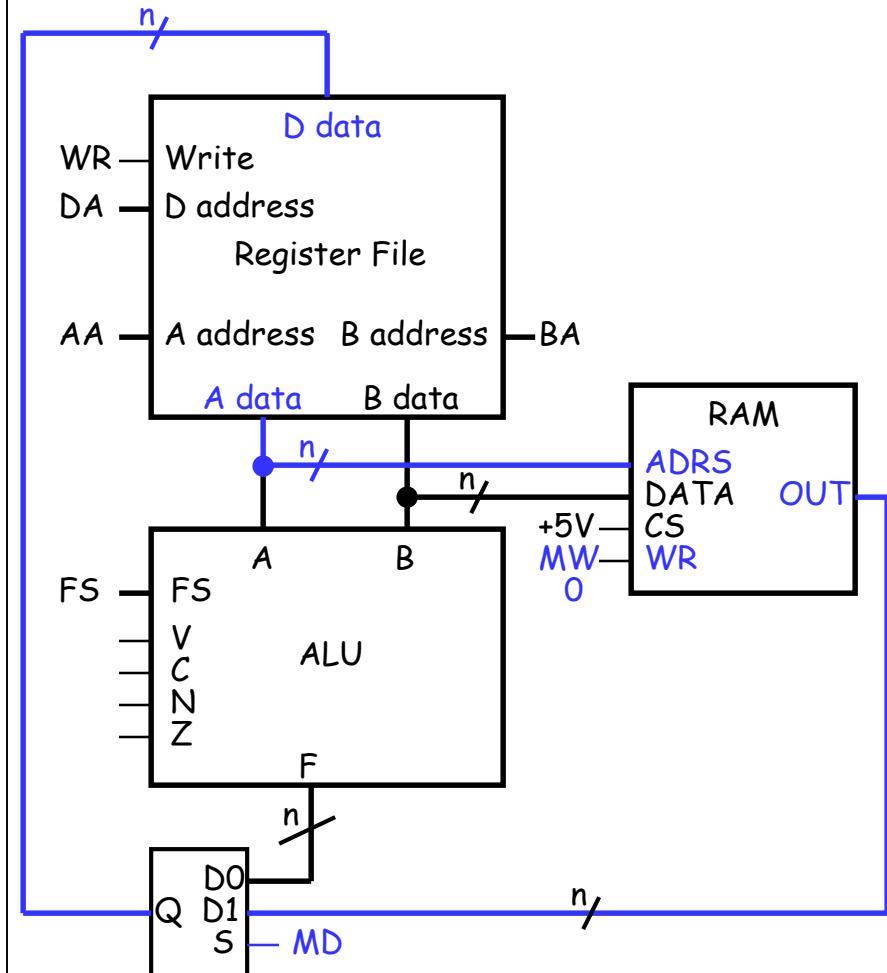
# We can access RAM also

- Here's a way to connect RAM into our existing datapath.
- To write to RAM, we must give an address and a data value.
- These will come from the registers. We connect **A data** to the memory's **ADRS** input, and **B data** to the memory's **DATA** input.
- Set **MW = 1** to write to the RAM. (It's called MW to distinguish it from the WR write signal on the register file.)



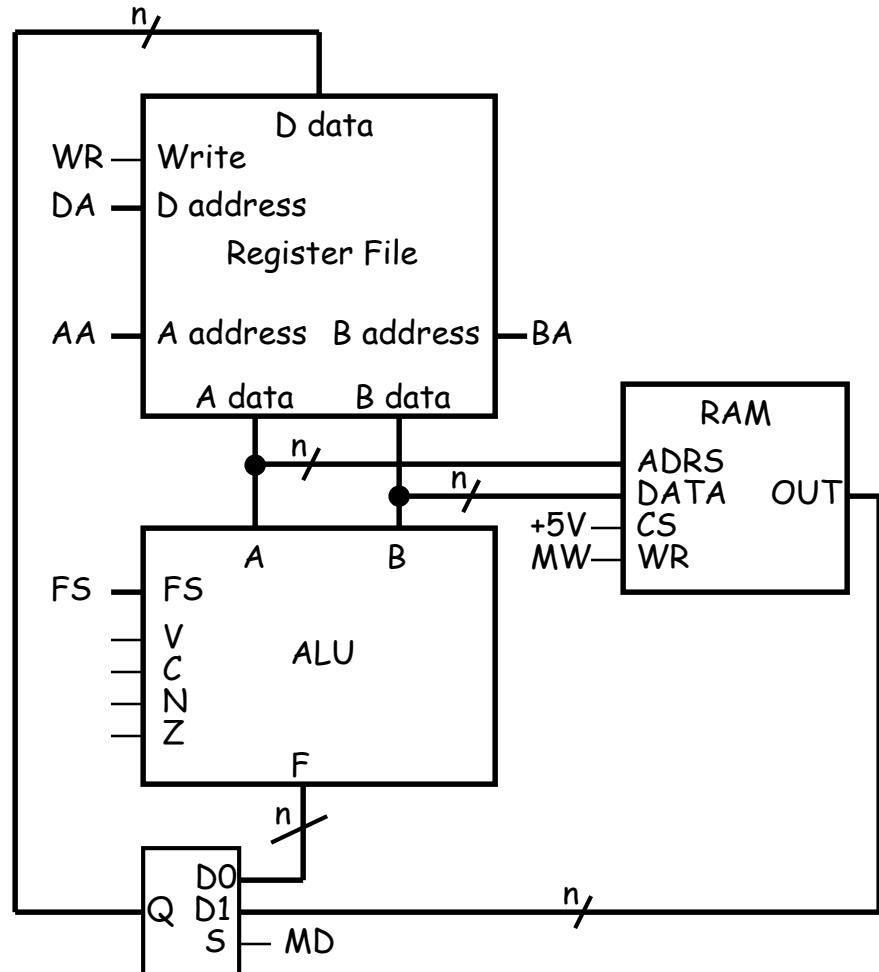
# Reading from RAM

- To read from RAM, **A data** must supply the address.
- Set **MW = 0** for reading.
- The incoming data will be sent to the register file for storage.
- This means that the register file's **D data** input could come from either the ALU output or the RAM.
- A mux **MD** selects the source for the register file.
  - When **MD = 0**, the ALU output can be stored in the register file.
  - When **MD = 1**, the RAM output is sent to the register file instead.



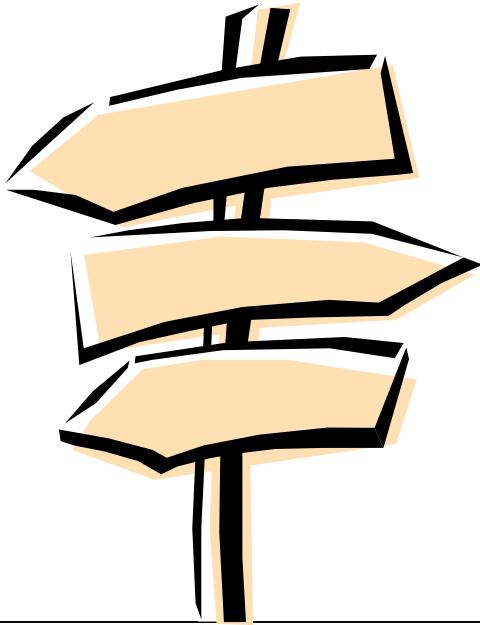
# Notes about this setup

- We now have a way to copy data between our register file and the RAM.
- Notice that there's no way for the ALU to directly access the memory—RAM contents must go through the register file first.
- Here the size of the memory is limited by the size of the registers; with  $n$ -bit registers, we can only use a  $2^n \times n$  RAM.
- For simplicity we'll assume the RAM is at least as fast as the CPU clock. (This is definitely not the case in real processors these days.)



# Memory transfer notation

- In our transfer language, the contents at random access memory address X are denoted  $M[X]$ . For example:
  - The first word in RAM is  $M[0]$ .
  - If register R1 contains an address, then  $M[R1]$  are the contents of that address.
- The  $M[ ]$  notation is like a pointer dereference operation in C or C++.



# Example sequence of operations

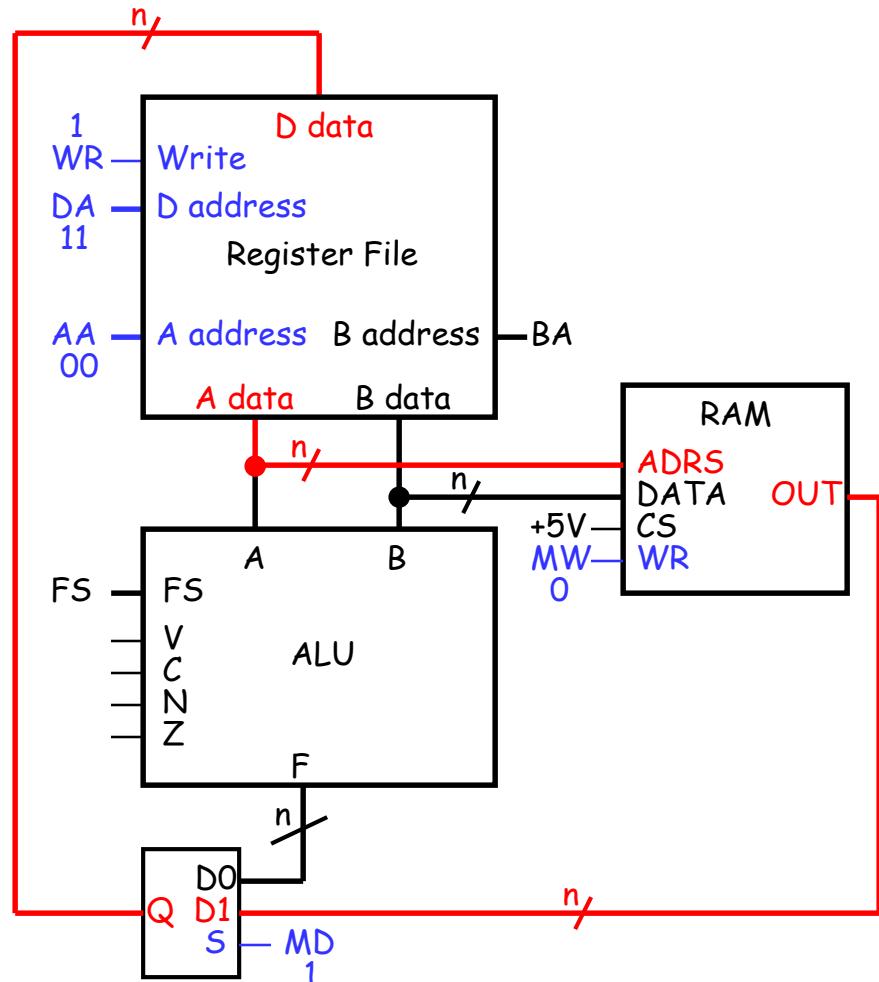
- Here is a simple series of register transfer instructions:

$$R3 \leftarrow M[R0]$$
$$R3 \leftarrow R3 + 1$$
$$M[R0] \leftarrow R3$$

- This just increments the contents at address R0 in RAM.
  - Again, our ALU only operates on registers, so the RAM contents must first be loaded into a register, and then saved back to RAM.
  - R0 is the first register in our register file. We'll assume it contains a valid memory address.
- How would these instructions execute in our datapath?

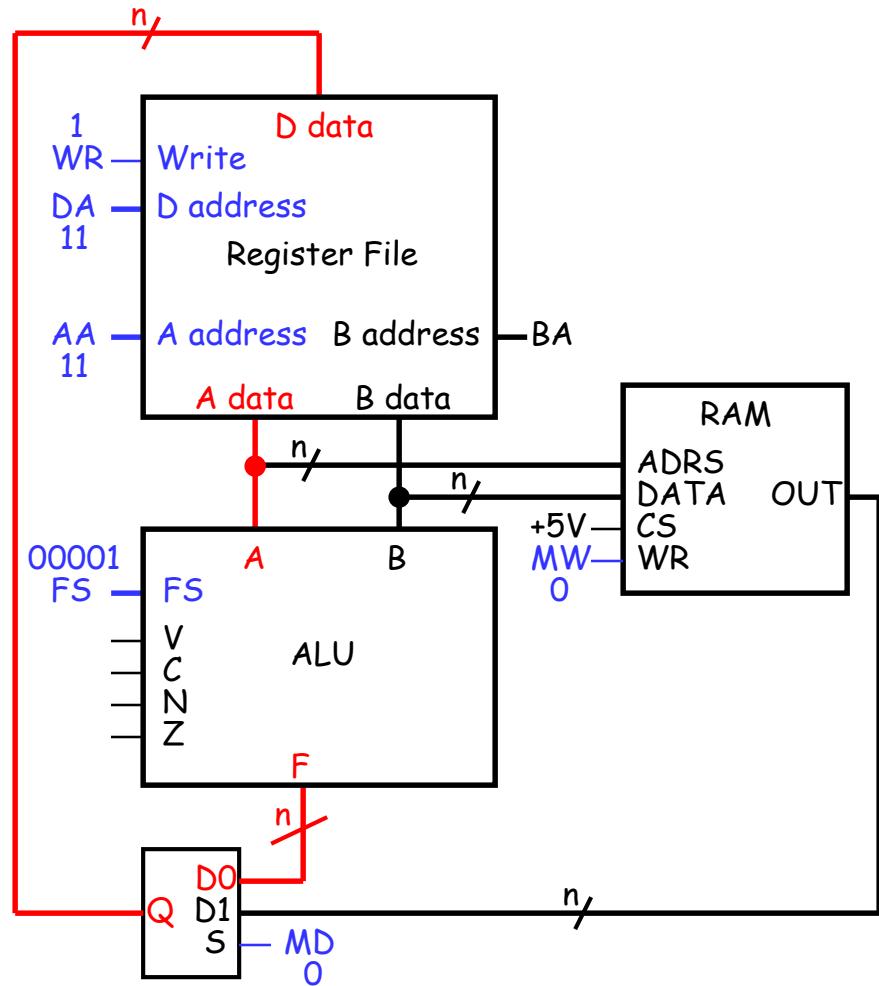
# $R3 \leftarrow M[R0]$

- $AA$  should be set to 00, to read register R0.
- The value in R0 will be sent to the RAM address input, so  $M[R0]$  appears as the RAM output OUT.
- $MD$  must be 1, so the RAM output goes to the register file.
- To store something into R3, we'll need to set  $DA = 11$  and  $WR = 1$ .
- $MW$  should be 0, so nothing is accidentally changed in RAM.
- Here, we did not use the ALU (FS) or the second register file output (BA).



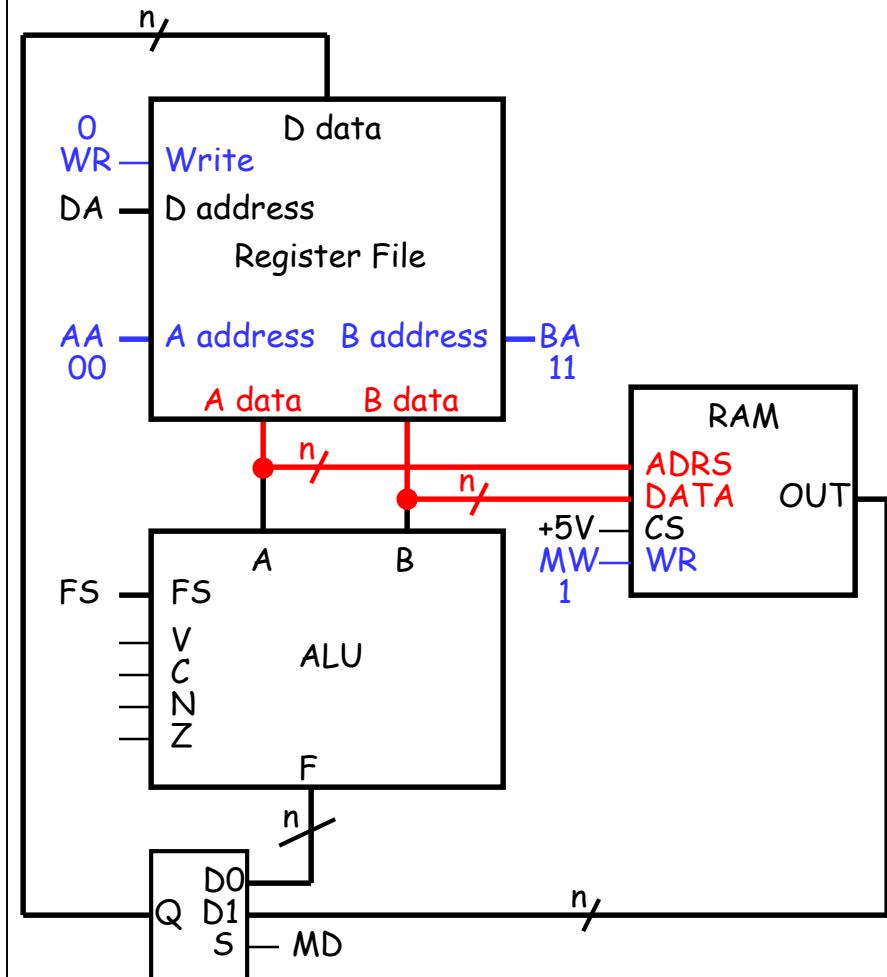
$$R3 \leftarrow R3 + 1$$

- $AA = 11$ , so R3 is read from the register file and sent to the ALU's A input.
- $FS$  needs to be 00001 for the operation  $A + 1$ . Then,  $R3 + 1$  appears as the ALU output F.
- If  $MD$  is set to 0, this output will go back to the register file.
- To write to R3, we need to make  $DA = 11$  and  $WR = 1$ .
- Again,  $MW$  should be 0 so the RAM isn't inadvertently changed.
- We didn't use BA.



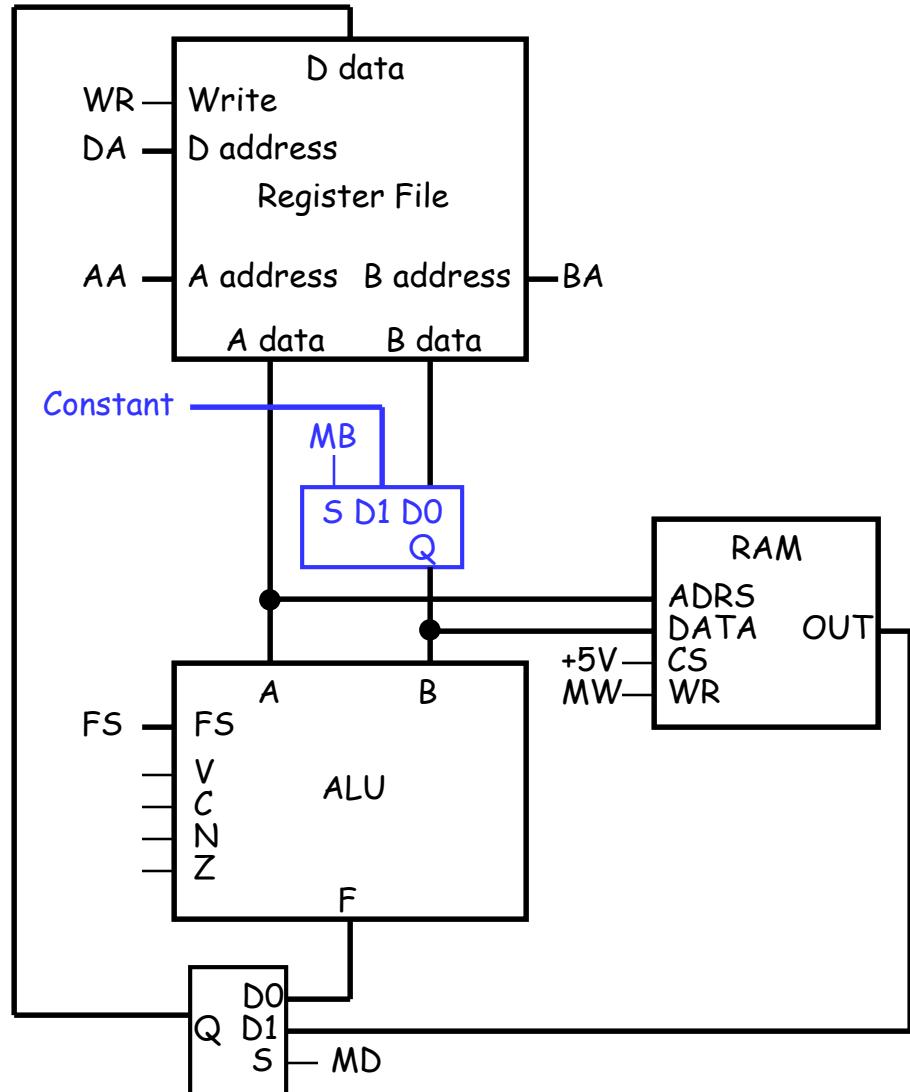
# $M[R0] \leftarrow R3$

- Finally, we want to store the contents of R3 into RAM address R0.
- Remember the RAM address comes from "A data," and the contents come from "B data."
- So we have to set  $AA = 00$  and  $BA = 11$ . This sends R0 to ADRS, and R3 to DATA.
- $MW$  must be 1 to write to memory.
- No register updates are needed, so  $WR$  should be 0, and MD and DA are unused.
- We also didn't use the ALU, so FS was ignored.



# Constant in

- One last refinement is the addition of a **Constant** input.
- The modified datapath is shown on the right, with one extra control signal **MB**.
- We'll see how this is used later. Intuitively, it provides an easy way to initialize a register or memory location with some arbitrary number.



# Control units

---

- From these examples, you can see that different actions are performed when we provide different inputs for the datapath control signals.
- The second question we had was "Who exactly decides which registers are read and written and which ALU function is executed?"
  - In real computers, the datapath actions are determined by the **program** that's loaded and running.
  - A **control unit** is responsible for generating the correct control signals for a datapath, based on the program code.
- We'll talk about programs and control units later.

# Summary

---

- The **datapath** is the part of a processor where computation is done.
  - The basic components are an **ALU**, a **register file** and some **RAM**.
  - The **ALU** does all of the computations, while the register file and **RAM** provide storage for the **ALU**'s operands and results.
- Various control signals in the datapath govern its behavior.
- Next, we'll see how programmers can give commands to the processor, and how those commands are translated in control signals for the datapath.

# HW

---

1. Design a 4 bit arithmetic circuit, with two selection variables  $S_1$  and  $S_0$ , that generates the arithmetic operations below. Draw the logic diagram for a single bit stage. (Q 10-4)
2. A computer has a 32 bit instruction word broken into fields as follows: opcode, 6 bits; two register fields, 6 bits each; and one immediate operand/register field, 14 bits.
  - (a) What is the maximum number of operations that can be specified?
  - (b) How many registers can be addressed?
  - (c) What is the range of unsigned immediate operands that can be provided? (Q 10-13)

$S_1 S_0$	$Cin = 0$	$Cin = 1$
00	$F = A + B$ (add)	$F = A + B + 1$
01	$F = A$ (transfer)	$F = A + 1$ (increment)
10	$F = B'$ (complement)	$F = B' + 1$ (negate)
11	$F = A + B'$	$F = A + B' + 1$ (subtract)

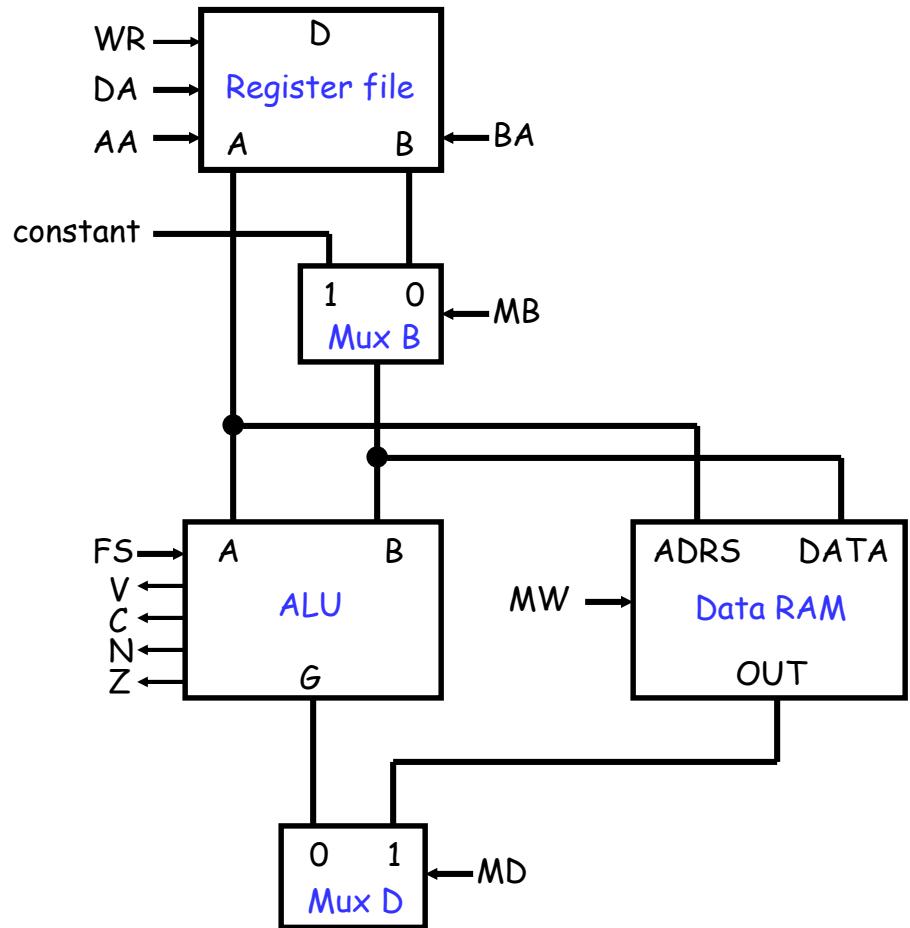
# Control units

---

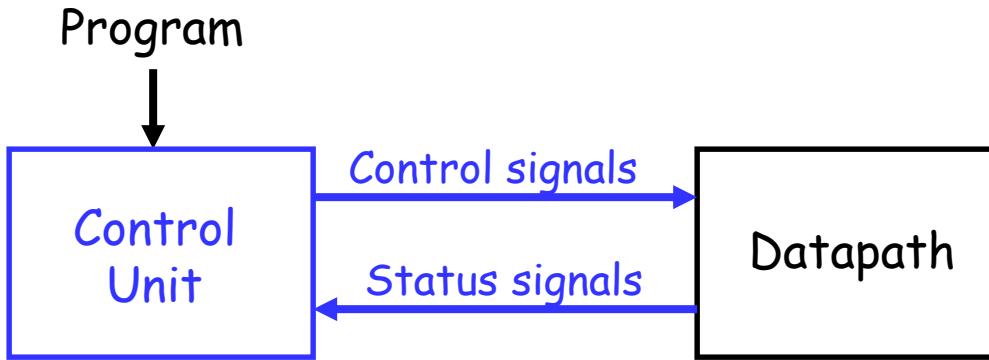
- We introduced the basic structure of a control unit, and translated assembly instructions into a binary representation.
- The last piece of the processor is a **control unit** to convert these binary instructions into datapath signals.
- At the end we'll have a complete example processor!

# Datapath review

- Set WR = 1 to write one of the registers.
- DA is the register to save to.
- AA and BA select the source registers.
- MB chooses a register or a constant operand.
- FS selects an ALU operation.
- MW = 1 to write to memory.
- MD selects between the ALU result and the RAM output.
- V, C, N and Z are status bits.



# Block diagram of a processor



- The **control unit** connects programs with the datapath.
  - It converts program instructions into **control words** for the datapath, including signals WR, DA, AA, BA, MB, FS, MW, MD.
  - It executes program instructions in the correct sequence.
  - It generates the “constant” input for the datapath.
- The datapath also sends information back to the control unit. For instance, the ALU status bits V, C, N, Z can be inspected by branch instructions to alter a program’s control flow.

# Where does the program go?

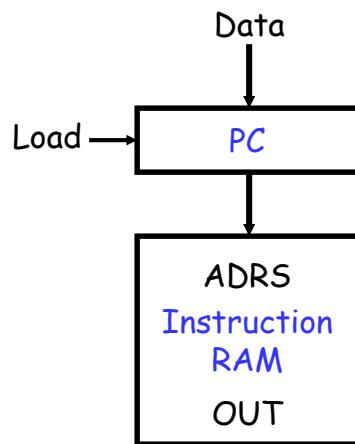
- We'll use a **Harvard architecture**, which includes two memory units.
  - An **instruction memory** holds the program.
  - A separate **data memory** is used for computations.
  - The advantage is that we can read an instruction and load or store data in the same clock cycle.
- For simplicity, our diagrams do not show any WR or DATA inputs to the instruction memory.



- Caches in modern CPUs often feature a Harvard architecture like this.
- However, there is usually a single main memory that holds both program instructions and data, in a **Von Neumann architecture**.

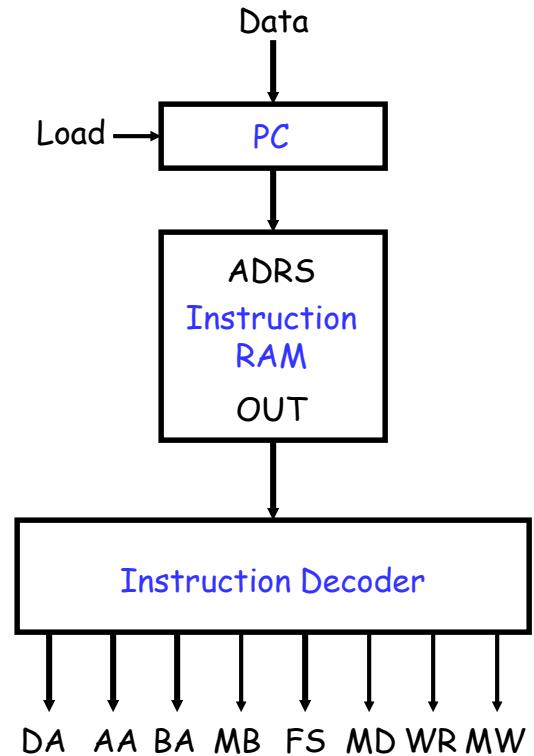
# Program counter

- A **program counter** or **PC** addresses the instruction memory, to keep track of the instruction currently being executed.
- On each clock cycle, the counter does one of two things.
  - If **Load = 0**, the PC increments, so the next instruction in memory will be executed.
  - If **Load = 1**, the PC is updated with **Data**, which represents some address specified in a jump or branch instruction.



# Instruction decoder

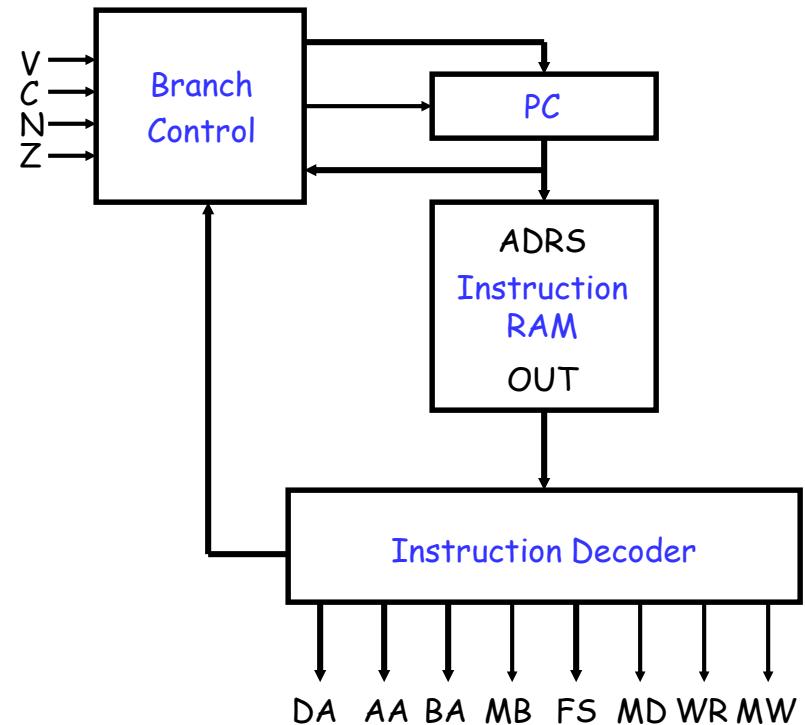
- The **instruction decoder** is a combinational circuit that takes a machine language instruction and produces the matching control signals for the datapath.
- These signals tell the datapath which registers or memory locations to access, and what ALU operations to perform.



(to the datapath)

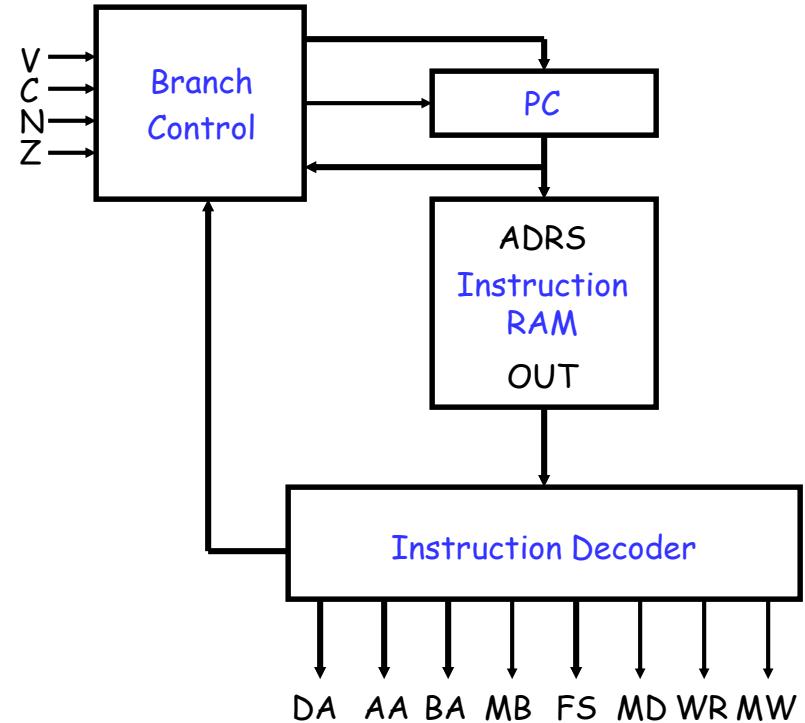
# Jumps and branches

- Finally, the **branch control unit** decides what the PC's next value should be.
  - For jumps, the PC should be loaded with the target address specified in the instruction.
  - For branch instructions, the PC should be loaded with the target address only if the corresponding status bit is true.
  - For all other instructions, the PC should just increment.

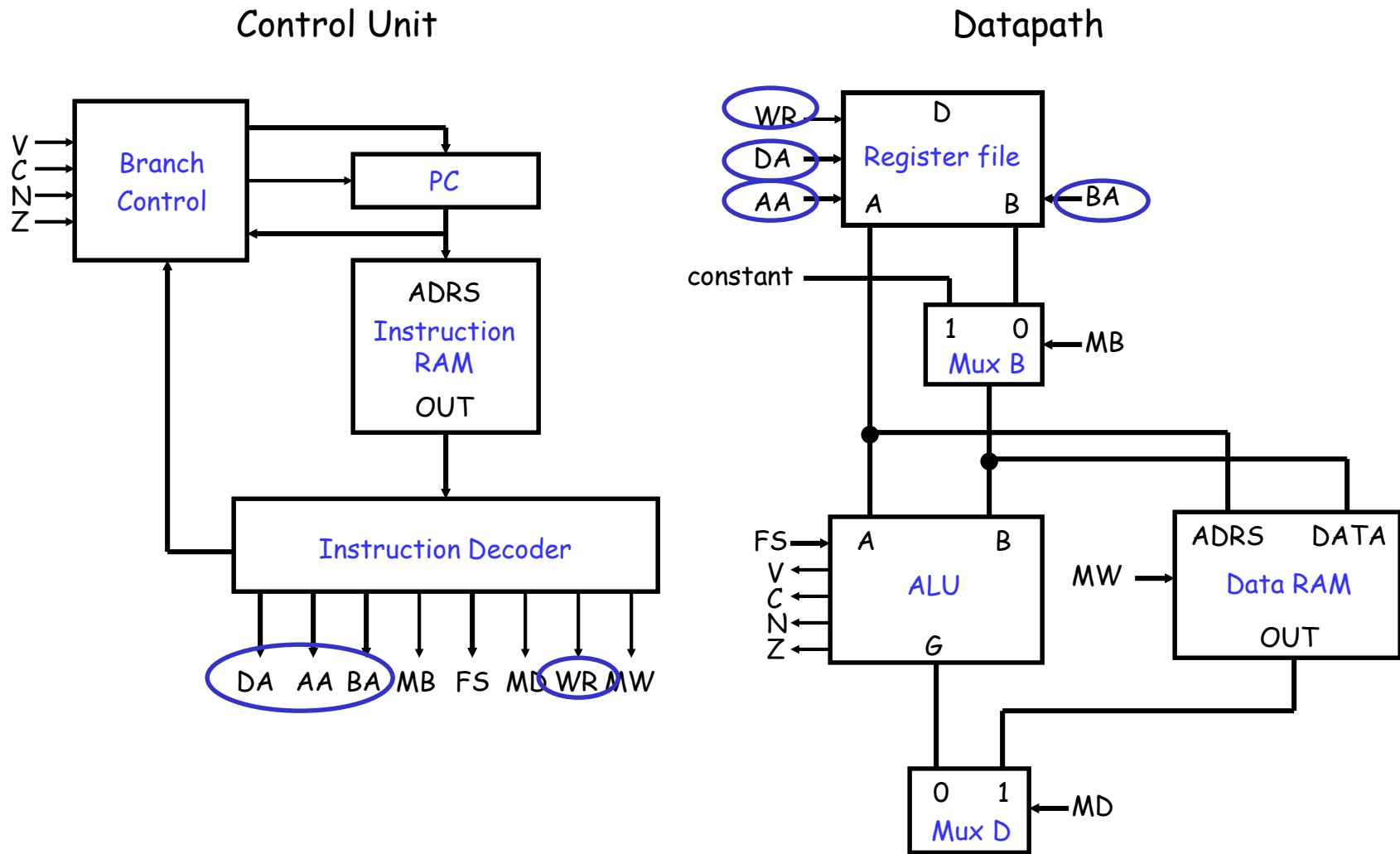


# That's it!

- This is the basic control unit. On each clock cycle:
  1. An instruction is read from the instruction memory.
  2. The instruction decoder generates the matching datapath control word.
  3. Datapath registers are read and sent to the ALU or the data memory.
  4. ALU or RAM outputs are written back to the register file.
  5. The PC is incremented, or reloaded for branches and jumps.



# The whole processor



# Instruction format

- We have three different instruction formats, each 16 bits long with a seven-bit opcode and nine bits for source registers or constants.
- The first three bits of the opcode determine the instruction category, while the other four bits indicate the exact instruction.
  - For ALU/shift instructions, the four bits choose an ALU operation.
  - For branches, the bits select one of eight branch conditions.
  - We only support one load, one store, and one jump instruction.

15	9 8	6 5	3 2	0
Opcode	Destination (DR) or Address 5-3 (AD)	Register A (SA)	Register B (SB), Operand (OP), or Address 2-0 (AD)	

# Instruction Formats

15	9 8	6 5	3 2	0
Opcode	Destination register (DR)	Source register A (SA)	Source register B (SB)	

(a) Register

15	9 8	6 5	3 2	0
Opcode	Destination register (DR)	Source register A (SA)	Operand (OP)	

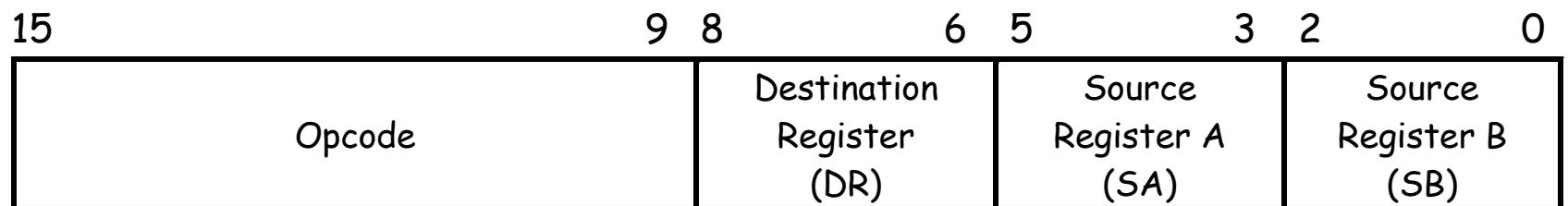
(b) Immediate

15	9 8	6 5	3 2	0
Opcode	Address (AD) (Left)	Source register A (SA)	Address (AD) (Right)	

(c) Jump and Branch

- The three formats are: Register, Immediate, and Jump and Branch
- All formats contain an Opcode field in bits 9 through 15.
- The Opcode specifies the operation to be performed

# Register format

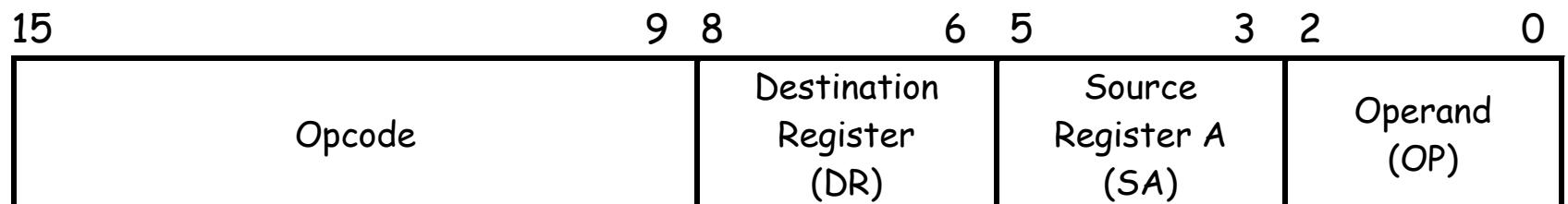


- An example register-format instruction:

ADD R1, R2, R3

- Our binary representation for these instructions will include:
  - A 7-bit **opcode** field, specifying the operation (e.g., ADD).
  - A 3-bit **destination register**, DR.
  - Two 3-bit **source registers**, SA and SB.

# Immediate format



- An example immediate-format instruction:

ADD R1, R2, #3

- Immediate-format instructions will consist of:
  - A 7-bit instruction opcode.
  - A 3-bit destination register, DR.
  - A 3-bit source register, SA.
  - A 3-bit **constant operand**, OP.

# Jump and branch format

15	9 8	6 5	3 2	0
Opcode	Address Bits 5-3 (AD)	Source Register A (SA)	Address Bits 2-0 (AD)	

- Two example jump and branch instructions:

BZ R3, -24  
JMP 18

- Jump and branch format instructions include:
  - A 7-bit instruction opcode.
  - A 3-bit source register SA for branch conditions.
  - A 6-bit **address field**, AD, for storing jump or branch offsets.
- Our branch instructions support only one source register. Other types of branches can be simulated from these basic ones.

# Assembly → machine language

- we defined a **machine language**, or a binary representation of the assembly instructions that our processor supports.
- Our CPU includes three types of instructions, which have different operands and will need different representations.
  - **Register format** instructions require two source registers.
  - **Immediate format** instructions have one source register and one constant operand.
  - **Jump and branch format** instructions need one source register and one constant address.
- Even though there are three different instruction formats, it is best to make their binary representations as similar as possible.
  - This will make the control unit hardware simpler.
  - For simplicity, all of our instructions are 16 bits long.

# Table 10-8

---

## Instruction Specifications for the Simple Computer - Part 1

---

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	$R[DR] \leftarrow R[SA]$	N, Z
Increment	0000001	INC	RD,RA	$R[DR] \leftarrow R[SA] + 1$	N, Z
Add	0000010	ADD	RD,RA,RB	$R[DR] \leftarrow R[SA] + R[SB]$	N, Z
Subtract	0000101	SUB	RD,RA,RB	$R[DR] \leftarrow R[SA] - R[SB]$	N, Z
Decrement	0000110	DEC	RD,RA	$R[DR] \leftarrow R[SA] - 1$	N, Z
AND	0001000	AND	RD,RA,RB	$R[DR] \leftarrow R[SA] \wedge R[SB]$	N, Z
OR	0001001	OR	RD,RA,RB	$R[DR] \leftarrow R[SA] \vee R[SB]$	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	$R[DR] \leftarrow R[SA] \oplus R[SB]$	N, Z
NOT	0001011	NOT	RD,RA	$R[DR] \leftarrow \overline{R[SA]}$	N, Z

---

# Summary

---

- We saw an outline of the control unit hardware.
  - The program counter points into a special instruction memory, which contains a machine language program.
  - An instruction decoder looks at each instruction and generates the correct control signals for the datapath and a branching unit.
  - The branch control unit handles instruction sequencing.
- The control unit implementation depends on both the instruction set architecture and the datapath.
  - Careful selection of opcodes and instruction formats can make the control unit simpler.
- We now have a whole processor! This is the culmination of everything we did this semester, starting from primitive gates.

# CSE1003-Digital Logic Design

---

<b>Module:7</b>	<b>ARITHMETIC LOGIC UNIT</b>	<b>9 hours</b>
Bus Organization - ALU - Design of ALU - Status Register - Design of Shifter - Processor Unit - Design of specific Arithmetic Circuits Accumulator - Design of Accumulator.		

# Arithmetic-logic units

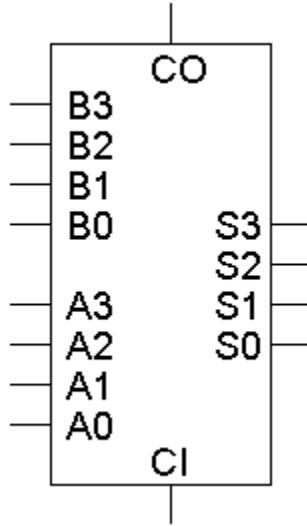
---

- An arithmetic-logic unit, or ALU, performs many different arithmetic and logic operations. The ALU is the “heart” of a processor—you could say that everything else in the CPU is there to support the ALU.
- Here’s the plan:
  - We’ll show an arithmetic unit first, by building off ideas from the adder-subtractor circuit.
  - Then we’ll talk about logic operations a bit, and build a logic unit.
  - Finally, we put these pieces together using multiplexers.
- We use some examples from the textbook, but things are re-labeled and treated a little differently.

# The four-bit adder

---

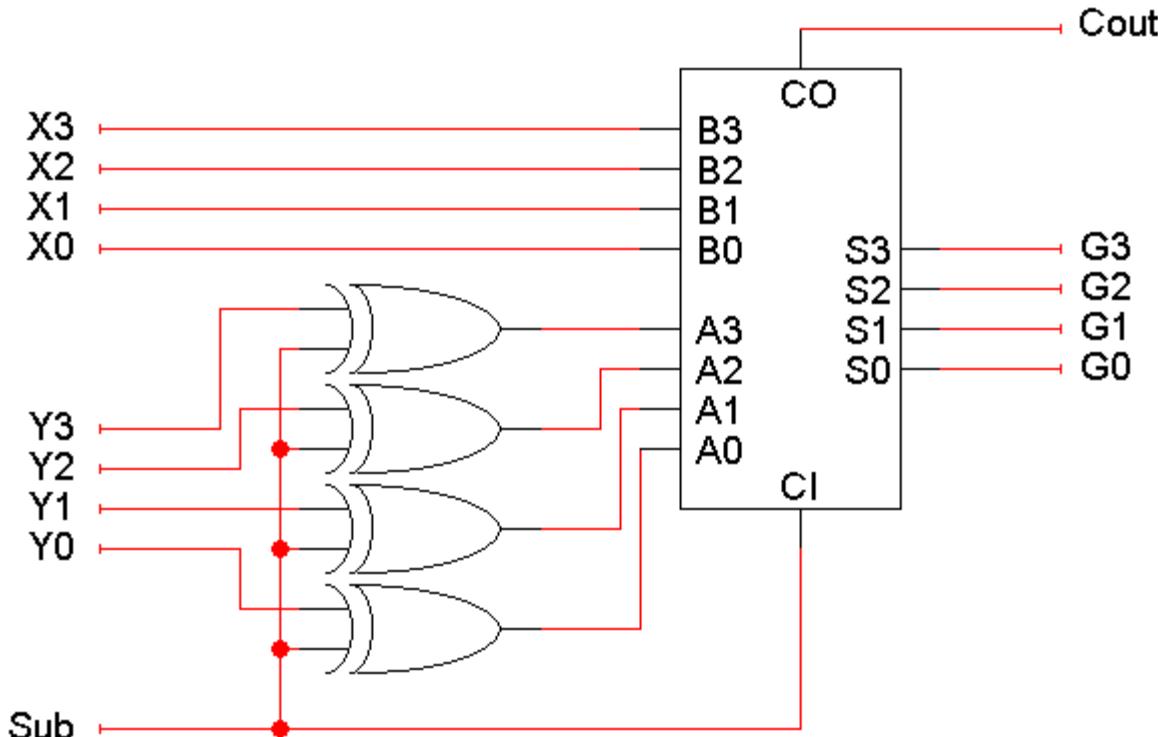
- The basic four-bit adder *always* computes  $S = A + B + CI$ .



- But by changing what goes into the adder inputs  $A$ ,  $B$  and  $CI$ , we can change the adder output  $S$ .
- This is also what we did to build the combined adder-subtractor circuit.

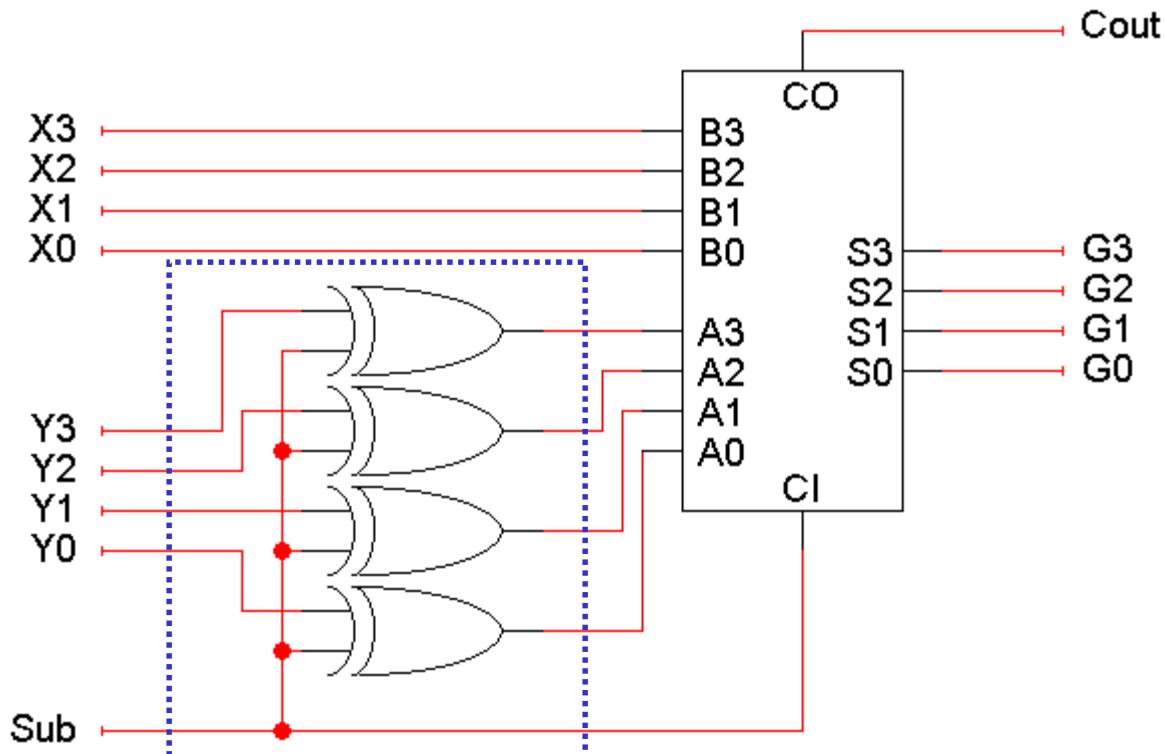
# It's the adder-subtractor again!

- Here the signal Sub and some XOR gates alter the adder inputs.
  - When **Sub = 0**, the adder inputs A, B, CI are Y, X, 0, so the adder produces  $G = X + Y + 0$ , or just  **$X + Y$** .
  - When **Sub = 1**, the adder inputs are  $Y'$ , X and 1, so the adder output is  $G = X + Y' + 1$ , or the two's complement operation  **$X - Y$** .



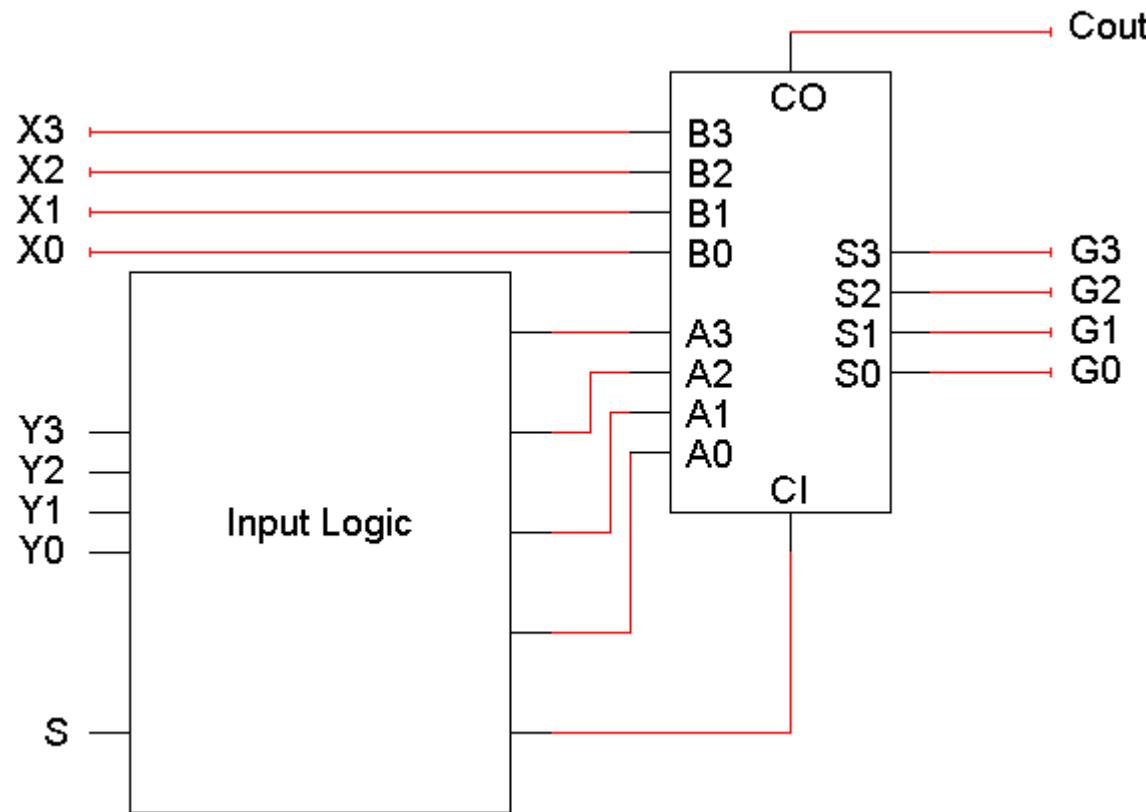
# The multi-talented adder

- So we have one adder performing two separate functions.
  - “Sub” acts like a function select input which determines whether the circuit performs addition or subtraction.
  - Circuit-wise, all “Sub” does is modify the adder’s inputs A and CI.



# Modifying the adder inputs

- By following the same approach, we can use an adder to compute other functions as well.
- We just have to figure out which functions we want, and then put the right circuitry into the "Input Logic" box .



# Some more possible functions

- We already saw how to set adder inputs A, B and CI to compute either  $X + Y$  or  $X - Y$ .
- How can we produce the increment function  $G = X + 1$ ?

One way: Set  $A = 0000$ ,  $B = X$ , and  $CI = 1$

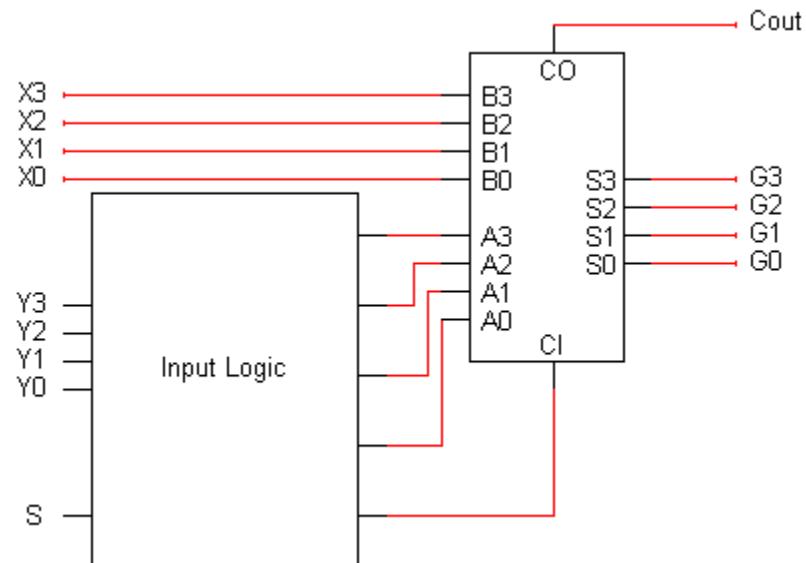
- How about decrement:  $G = X - 1$ ?

$A = 1111 (-1)$ ,  $B = X$ ,  $CI = 0$

- How about transfer:  $G = X$ ?  
(This can be useful.)

$A = 0000$ ,  $B = X$ ,  $CI = 0$

This is almost the same as the increment function!



# The role of CI

- The transfer and increment operations have the same A and B inputs, and differ only in the CI input.
- In general we can get additional functions (not all of them useful) by using both  $CI = 0$  and  $CI = 1$ .
- Another example:
  - Two's-complement subtraction is obtained by setting  $A = Y'$ ,  $B = X$ , and  $CI = 1$ , so  $G = X + Y' + 1$ .
  - If we keep  $A = Y'$  and  $B = X$ , but set  $CI$  to 0, we get  $G = X + Y'$ . This turns out to be a ones' complement subtraction operation.

# Table of arithmetic functions

---

- Here are some of the different possible arithmetic operations.
- We'll need some way to specify which function we're interested in, so we've randomly assigned a selection code to each operation.

$S_2$	$S_1$	$S_0$	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	$X + 1$	(increment)
0	1	0	$X + Y$	(add)
0	1	1	$X + Y + 1$	
1	0	0	$X + Y'$	(1C subtraction)
1	0	1	$X + Y' + 1$	(2C subtraction)
1	1	0	$X - 1$	(decrement)
1	1	1	X	(transfer)

# Mapping the table to an adder

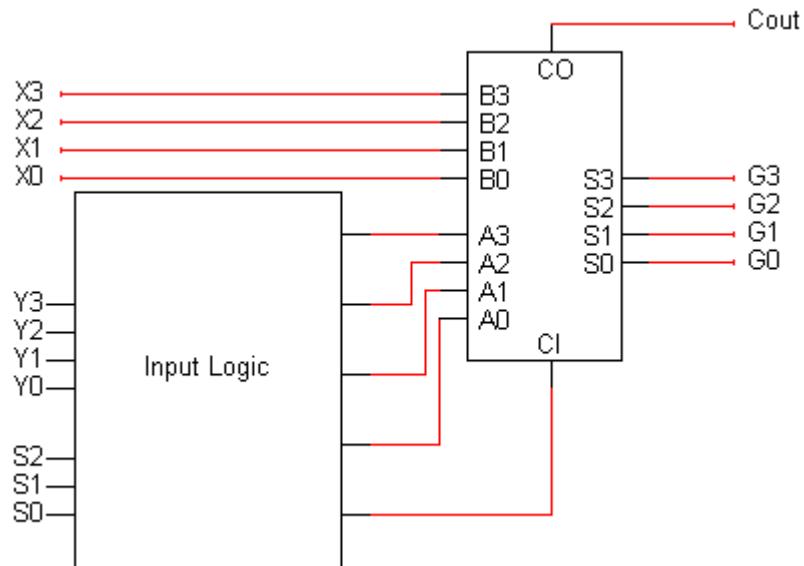
- This second table shows what the adder's inputs should be for each of our eight desired arithmetic operations.
  - Adder input CI is always the same as selection code bit  $S_0$ .
  - B is always set to X.
  - A depends only on  $S_2$  and  $S_1$ .
- These equations depend on both the desired operations and the assignment of selection codes.

Selection code $S_2$ $S_1$ $S_0$	Desired arithmetic operation $G (A + B + CI)$	Required adder inputs A      B      CI
0    0    0	X                 (transfer)	0000    X    0
0    0    1	$X + 1$ (increment)	0000    X    1
0    1    0	$X + Y$ (add)	Y          X    0
0    1    1	$X + Y + 1$	Y          X    1
1    0    0	$X + Y'$ (1C subtraction)	$Y'$ X    0
1    0    1	$X + Y' + 1$ (2C subtraction)	$Y'$ X    1
1    1    0	$X - 1$ (decrement)	1111        X    0
1    1    1	X                 (transfer)	1111        X    1

# Building the input logic

- All we need to do is compute the adder input A, given the arithmetic unit input Y and the function select code S (actually just  $S_2$  and  $S_1$ ).
- Here is an abbreviated truth table:

$S_2$	$S_1$	A
0	0	0000
0	1	Y
1	0	$Y'$
1	1	1111

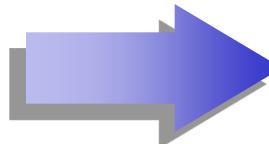


- We want to pick one of these four possible values for A, depending on  $S_2$  and  $S_1$ .

# Primitive gate-based input logic

- We could build this circuit using primitive gates.
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table.
  - The Y that appears in the output column ( $A$ ) is actually an input.
  - We make that explicit in the table on the right.
- Remember  $A$  and  $Y$  are each 4 bits long!

$S_2$	$S_1$	$A$
0	0	0000
0	1	$y$
1	0	$y'$
1	1	1111



$S_2$	$S_1$	$y_i$	$A_i$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

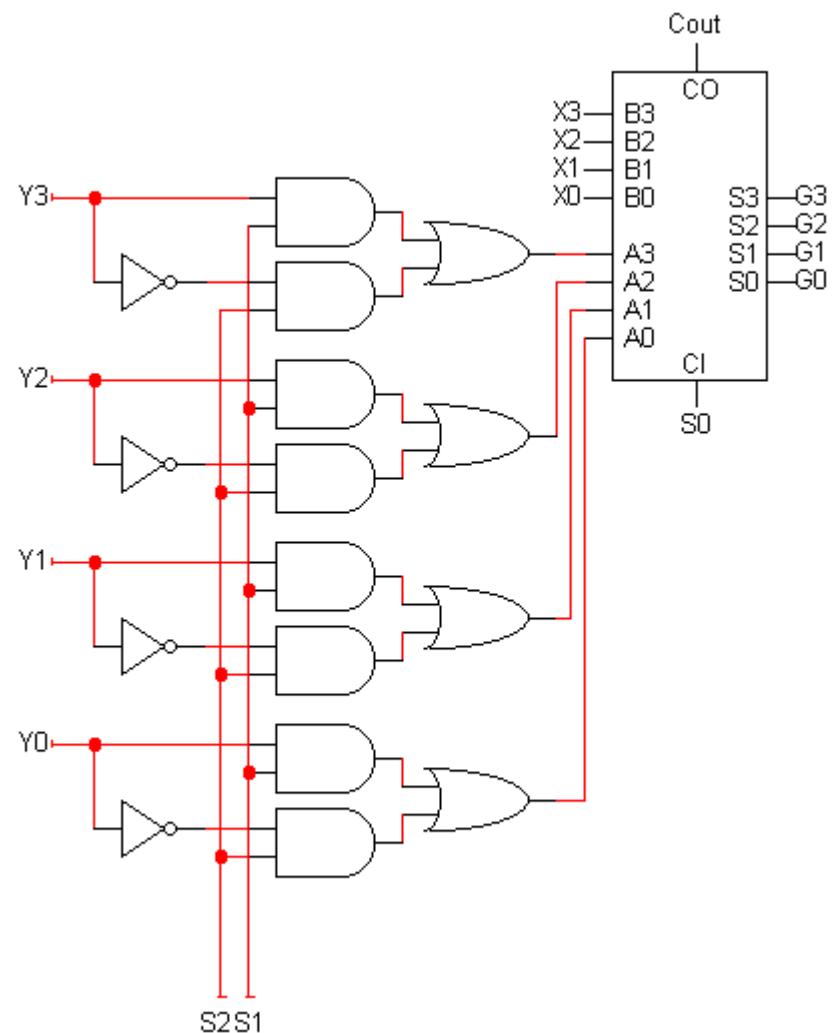
# Primitive gate implementation

- From the truth table, we can find an MSP:

		$S_1$	
		0	1
		1	0
$S_2$	0	0	1
	1	0	1
		$y_i$	

$$A_i = S_2 Y'_i + S_1 Y_i$$

- Again, we have to repeat this once for each bit  $Y_3-Y_0$ , connecting to the adder inputs  $A_3-A_0$ .
- This completes our arithmetic unit.



# Bitwise operations

- Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit **words** instead of just single bits.
- To apply a logical operation to two words  $X$  and  $Y$ , apply the operation on each pair of bits  $X_i$  and  $Y_i$ :

$$\begin{array}{r} \text{1 } \text{0 } \text{1 } \text{1} \\ \text{AND } \text{1 } \text{1 } \text{1 } \text{0} \\ \hline \text{1 } \text{0 } \text{1 } \text{0} \end{array}$$

$$\begin{array}{r} \text{1 } \text{0 } \text{1 } \text{1} \\ \text{OR } \text{1 } \text{1 } \text{1 } \text{0} \\ \hline \text{1 } \text{1 } \text{1 } \text{1} \end{array}$$

$$\begin{array}{r} \text{1 } \text{0 } \text{1 } \text{1} \\ \text{XOR } \text{1 } \text{1 } \text{1 } \text{0} \\ \hline \text{0 } \text{1 } \text{0 } \text{1} \end{array}$$

- We've already seen this informally in two's-complement arithmetic, when we talked about "complementing" all the bits in a number.

# Bitwise operations in programming

- Languages like C, C++ Java and HDLs provide bitwise logical operations:

& (AND)      | (OR)      ^ (XOR)      ~ (NOT)

- These operations treat each integer as a bunch of individual bits:

$13 \& 25 = 9$       because       $01101 \& 11001 = 01001$

- They are not the same as the operators `&&`, `||` and `!`, which treat each integer as a single logical value (0 is false, everything else is true):

$13 \&\& 25 = 1$       because      `true && true = true`

- Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.
- Easy to represent sets of fixed universe size with bits:
  - 1: is member, 0 not a member. Unions: OR, Intersections: AND

# Bitwise operations in networking

- IP addresses are actually 32-bit binary numbers, and bitwise operations can be used to find network information.
- For example, you can bitwise-AND an address 192.168.10.43 with a "subnet mask" to find the "network address," or which network the machine is connected to.

$$\begin{array}{r} 192.168.10.43 = 11000000.10101000.00001010.00101011 \\ \& 255.255.255.224 = 11111111.11111111.11111111.11100000 \\ \hline 192.168.10.32 = 11000000.10101000.00001010.00100000 \end{array}$$

- You can use bitwise-OR to generate a "broadcast address," for sending data to all machines on the local network.

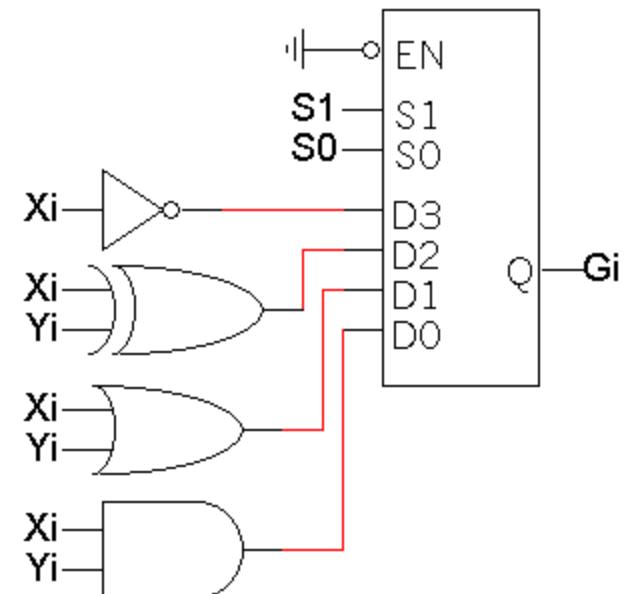
$$\begin{array}{r} 192.168.10.43 = 11000000.10101000.00001010.00101011 \\ | 0.0.0.31 = 00000000.00000000.00000000.00011111 \\ \hline 192.168.10.63 = 11000000.10101000.00001010.00111111 \end{array}$$

# Defining a logic unit

- A logic unit supports different logical functions on two multi-bit inputs  $X$  and  $Y$ , producing an output  $G$ .
- This abbreviated table shows four possible functions and assigns a selection code  $S$  to each.

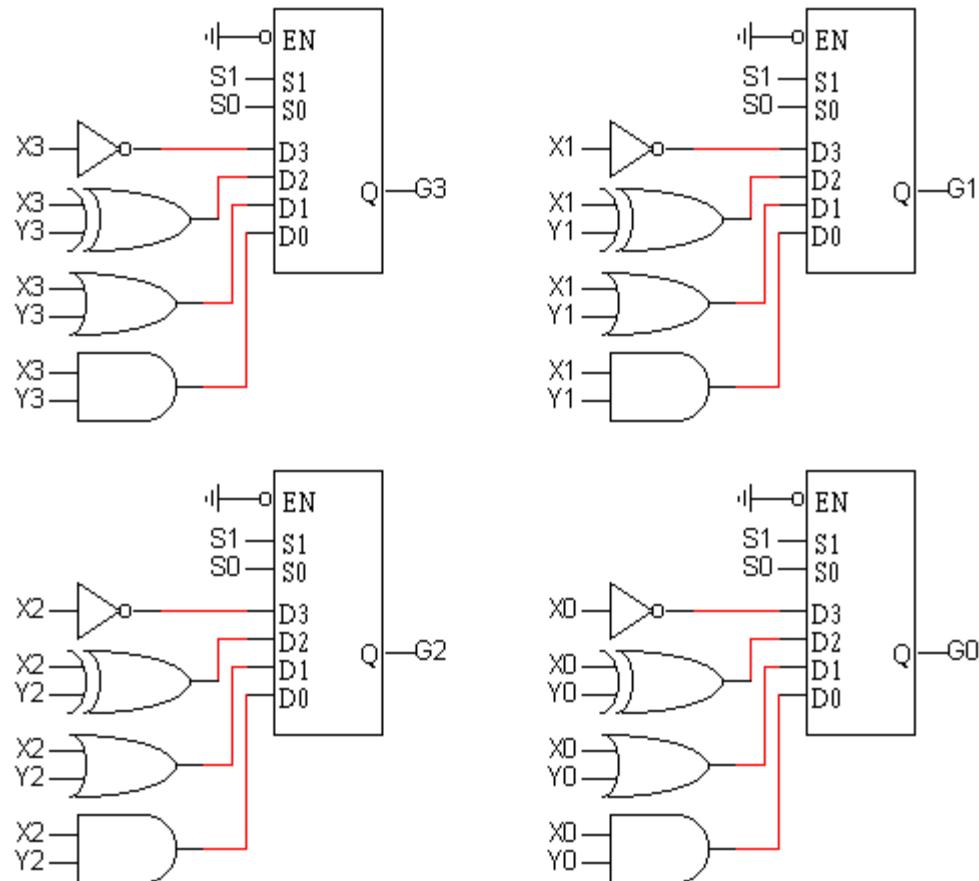
$S_1$	$S_0$	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$

- We'll just use multiplexers and some primitive gates to implement this.
- Again, we need one multiplexer for each bit of  $X$  and  $Y$ .



# Our simple logic unit

- **Inputs:**
  - X (4 bits)
  - Y (4 bits)
  - S (2 bits)
- **Outputs:**
  - G (4 bits)



# Combining the arithmetic and logic units

- Now we have two pieces of the puzzle:
  - An arithmetic unit that can compute eight functions on 4-bit inputs.
  - A logic unit that can perform four functions on 4-bit inputs.
- We can combine these together into a single circuit, an arithmetic-logic unit (ALU).

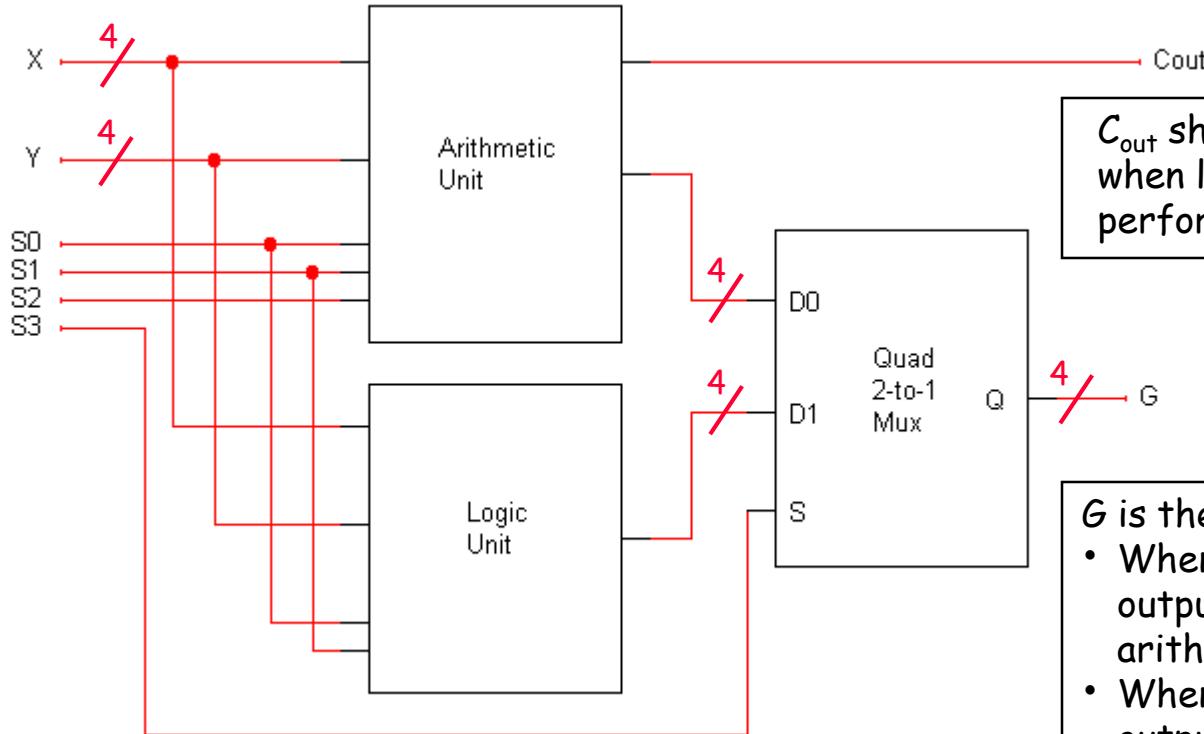
# Our ALU function table

- This table shows a sample function table for an ALU.
- All of the arithmetic operations have  $S_3=0$ , and all of the logical operations have  $S_3=1$ .
- These are the same functions we saw when we built our arithmetic and logic units a few minutes ago.
- Since our ALU only has 4 logical operations, we don't need  $S_2$ . The operation done by the logic unit depends only on  $S_1$  and  $S_0$ .

$S_3$	$S_2$	$S_1$	$S_0$	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X \text{ and } Y$
1	x	0	1	$G = X \text{ or } Y$
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

# A complete ALU circuit

The / and 4 on a line indicate that it's actually four lines.



C<sub>out</sub> should be ignored when logic operations are performed (when S3=1).

G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.

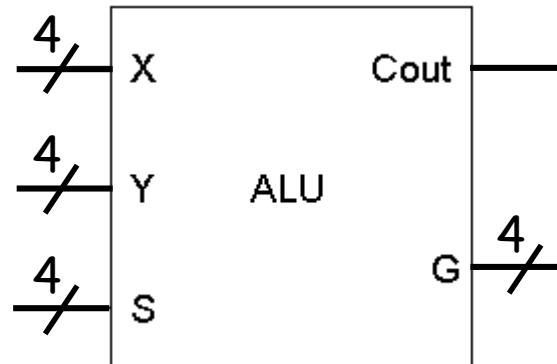
# Comments on the multiplexer

- Both the arithmetic unit and the logic unit are “active” and produce outputs.
  - The mux determines whether the final result comes from the arithmetic or logic unit.
  - The output of the other one is effectively ignored.
- Our hardware scheme may seem like wasted effort, but it’s not really.
  - “Deactivating” one or the other wouldn’t save that much time.
  - We have to build hardware for both units anyway, so we might as well run them together.
- This is a very common use of multiplexers in logic design.

# The completed ALU

---

- This ALU is a good example of hierarchical design.
  - With the 12 inputs, the truth table would have had  $2^{12} = 4096$  lines. That's an awful lot of paper.
  - Instead, we were able to use components that we've seen before to construct the entire circuit from a couple of easy-to-understand components.
- As always, we encapsulate the complete circuit in a "black box" so we can reuse it in fancier circuits.



# ALU summary

---

- We looked at:
  - Building adders hierarchically, starting with one-bit full adders.
  - Representations of negative numbers to simplify subtraction.
  - Using adders to implement a variety of arithmetic functions.
  - Logic functions applied to multi-bit quantities.
  - Combining all of these operations into one unit, the ALU.
- Where are we now?
  - We started at the very bottom, with primitive gates, and now we can understand a small but critical part of a CPU.
  - This all built upon our knowledge of Boolean algebra, Karnaugh maps, multiplexers, circuit analysis and design, and data representations.