DIGITAL ELECTRONICS

Dr. Pradip Kumar Sahu
Department of Information Technology

Lecture of Module 1

Introduction to Digital Systems

Overview

- **Introduction**
- **Digital and Analog Signals**
- **Logic Levels and Digital Waveforms**
- **Positive and Negative Logics**
- **Combinational and Sequential logics**
- **Types of Logic Devices**

Introduction

Digital electronics is a field of electronics involving the study of digital signals and the engineering of devices that use or produce them.

This is in contrast to analog electronics and analog signals.

Digital electronic circuits are usually made from large assemblies of logic gates, often packaged in integrated circuits.

Complex devices may have simple electronic representations of Boolean logic functions.

Analog versus Digital

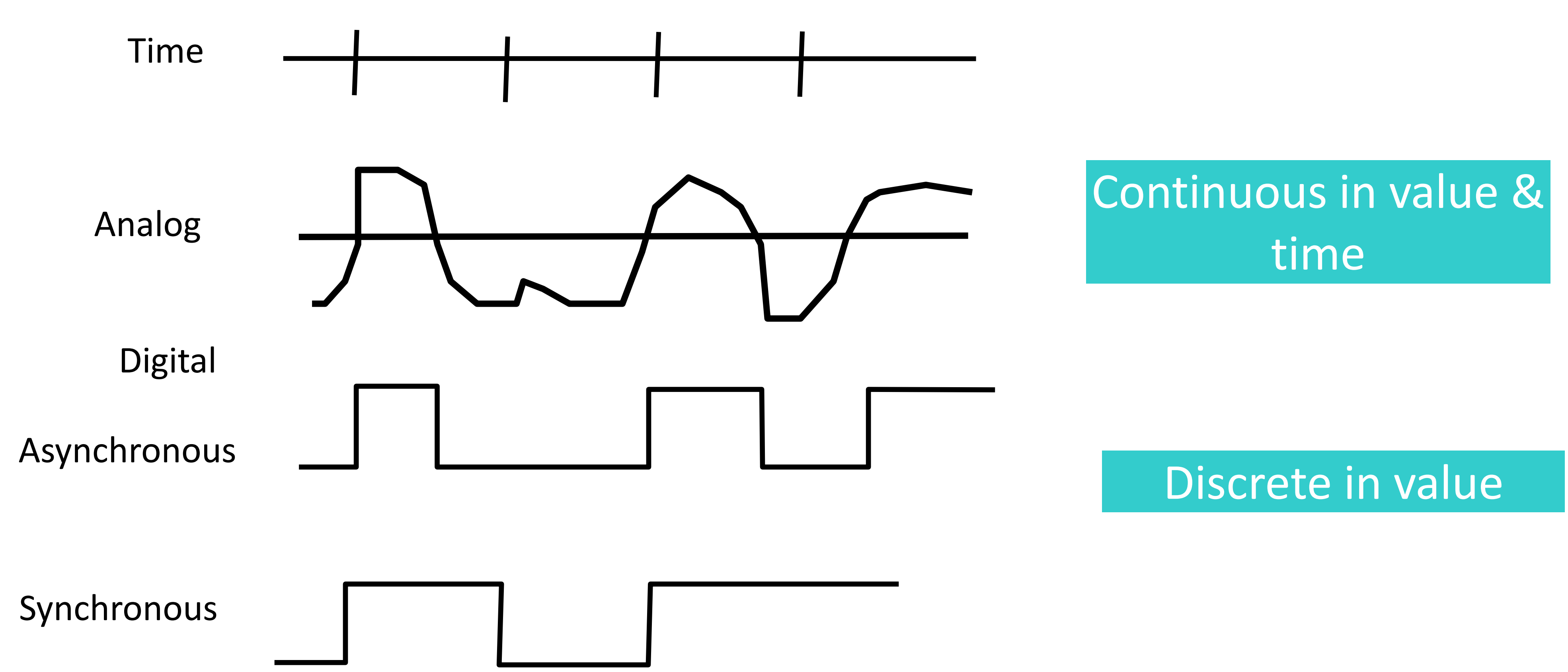
- Most observables are analog
- But the most convenient way to represent and transmit information electronically is digital
- Analog/digital and digital/analog conversion is essential

Analog Signals: The analog signals were used in many systems to produce signals to carry information. These signals are continuous in both values and time.
In short, analog signals – all signals that are natural or come naturally are analog signals.

Digital Signals: Unlike analog signals, digital signals are not continuous but signals are discrete in value and time. These signals are represented by binary numbers and consist of different voltage values.

Difference Between Analog And Digital Signal	
Analog Signals	**Digital Signals**
Continuous signals	Discrete signals
Represented by sine waves	Represented by square waves
Human voice, natural sound, analog electronic devices are few examples	Computers, optical drives, and other electronic devices
Continuous range of values	Discontinuous values
Records sound waves as they are	Converts into a binary waveform.
Only be used in analog devices.	Suited for digital electronics like computers, mobiles and more.

Signal Examples Over Time



Continuous in value & time

Discrete in value

Digital Signal

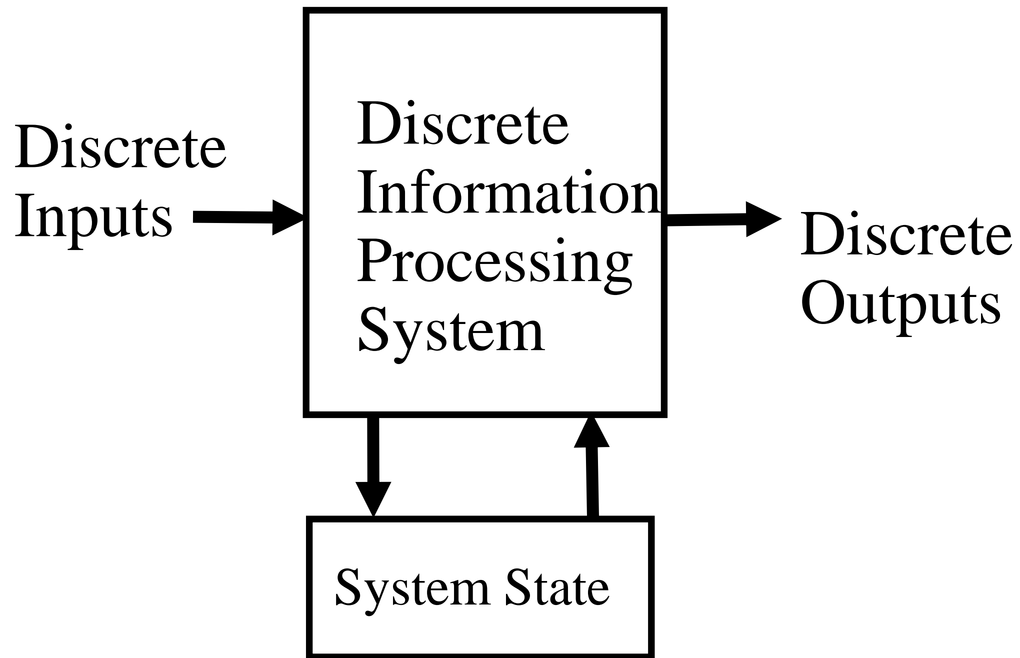
- An information variable represented by physical quantity.
- For digital systems, the variable takes on discrete values.
- Two level, or binary values are the most prevalent values in digital systems.
- Binary values are represented abstractly by:
  - digits 0 and 1
  - words (symbols) False (F) and True (T)
  - words (symbols) Low (L) and High (H)
  - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities

Binary Values: Other Physical Quantities

- What are other physical quantities represent 0 and 1?
  - CPU: Voltage Levels
  - Disk: Magnetic Field Direction
  - CD: Surface Pits/Light
  - Dynamic RAM: Electrical charge

Digital System

Takes a set of discrete information inputs and discrete internal information (system state) and generates a set of discrete information outputs.



Digital representations of logical functions

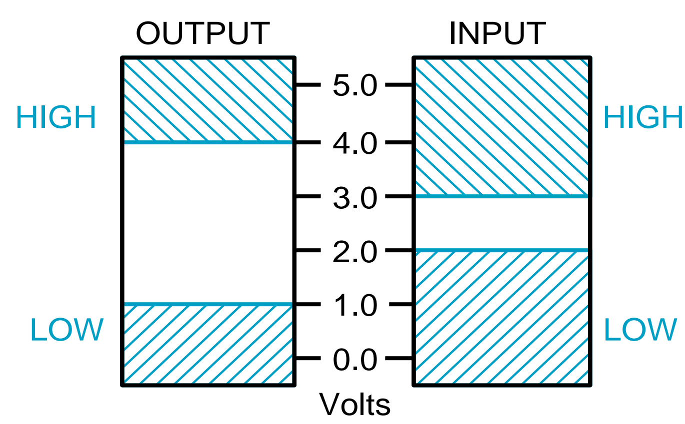
- Digital signals offer an effective way to execute logic. The formalism for performing logic with binary variables is called switching algebra or Boolean algebra.
- Digital electronics combines two important properties:
  - The ability to represent real functions by coding the information in digital form.
  - The ability to control a system by a process of manipulation and evaluation of digital variables using switching algebra.
- Digital signals can be transmitted, received, amplified, and retransmitted with no degradation.
- Binary numbers are a natural method of expressing logic variables.
- Complex logic functions are easily expressed as binary function.
- Digital information is easily and inexpensively stored

Logic Levels

In digital circuits, a **logic level** is one of a finite number of states that a digital signal can inhabit. Logic levels are usually represented by the voltage difference between the signal and ground, although other standards exist. The range of voltage levels that represent each state depends on the logic family being used.

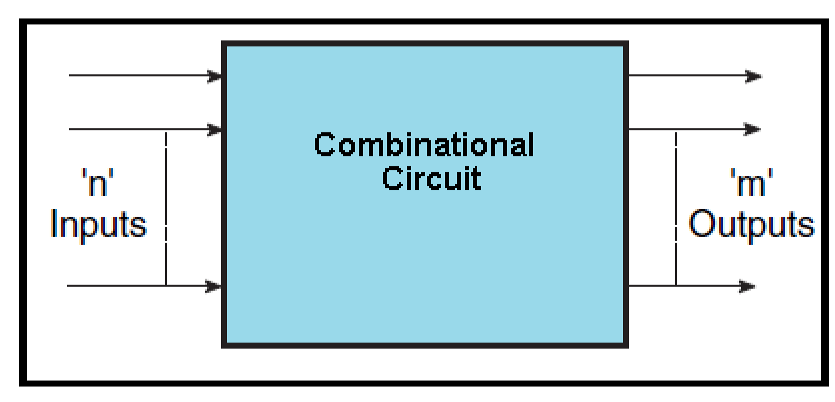
In binary logic the two levels are **logical high** and **logical low**, which generally correspond to binary numbers 1 and 0 respectively. Signals with one of these two levels can be used in Boolean algebra for digital circuit design or analysis.

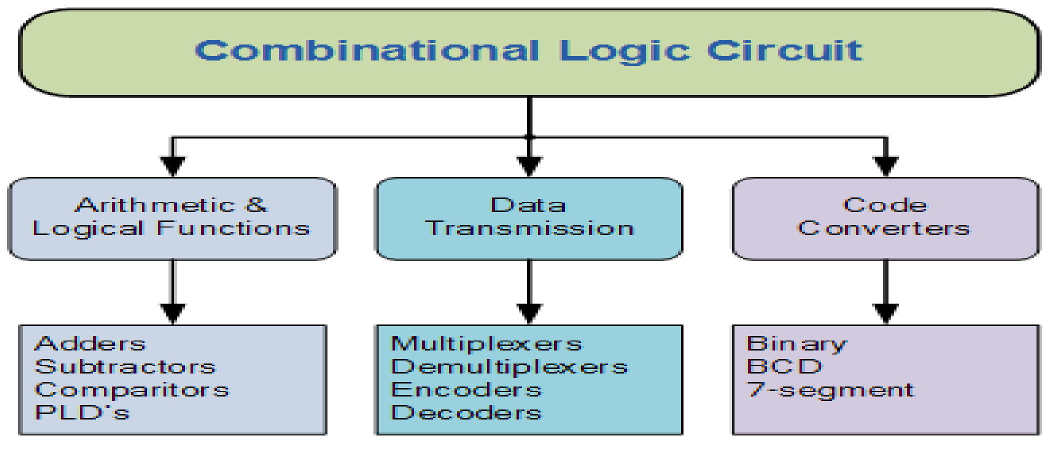
Logic level	Active-high signal	Active-low signal
Logical high	1	0
Logical low	0	1



Combinational Logic Circuit

The outputs of **Combinational Logic Circuits** are only determined by the logical function of their current input state, logic "0" or logic "1", at any given instant in time.

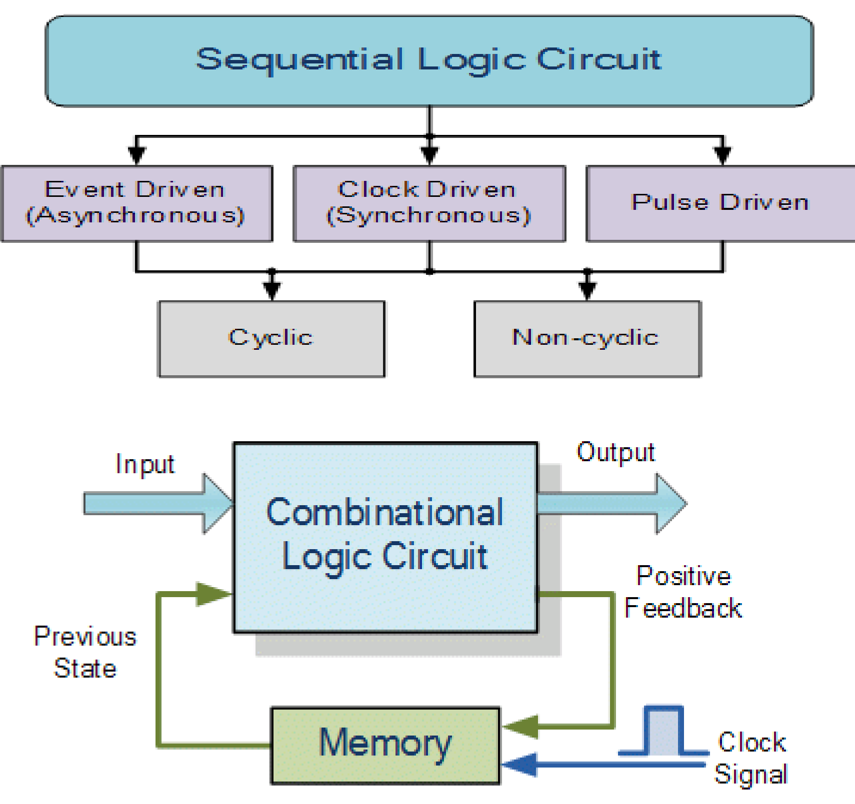




Sequential Logic Circuits

the output state of a “sequential logic circuit” is a function of the following three states, the “present input”, the “past input” and/or the “past output”. *Sequential Logic circuits* remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuits “Memory”.

Sequential logic circuits are generally termed as *two state* or Bistable devices which can have their output or outputs set in one of two basic states, a logic level “1” or a logic level “0” and will remain “latched” (hence the name latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.



Fixed function Logic devices

Fixed logic device such as a logic gate or a multiplexer or a flip-flop performs a given logic function that is known at the time of device manufacture

Complexity Classification for Fixed-Function ICs
SSI (Small-scale integration) – 10 gates–
MSI (Medium-scale integration) – 10—100 gates
LSI (Large-scale integration) – 100—10,000 gates
VLSI (Very large-scale integration) – 10,000—100,000 gates
ULSI (Ultra large-scale integration) -- >100,000 gates

Programmable Logic Devices

A programmable **logic device** can be configured by the user to perform a large variety of **logic functions**

A **programmable logic device** (**PLD**) is an electronic component used to build reconfigurable digital circuits

PLD has an undefined function at the time of manufacture

Before using PLD in a circuit it must be programmed (reconfigured) by using a specialized program

Purpose of PLD:

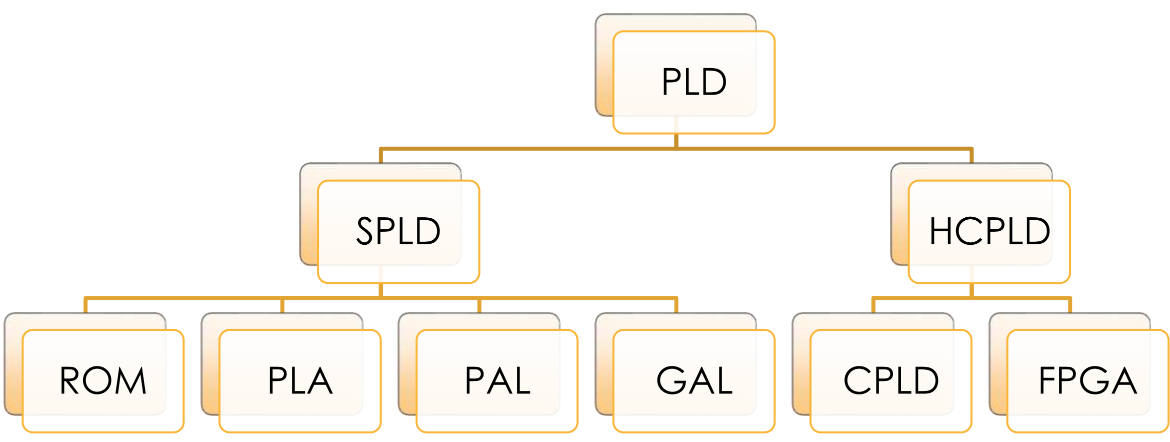
- Permits elaborate digital logic designs to be implemented by the user on a single device.
- Is capable of being erased and reprogrammed with a new design.

Advantages of PLDs

- Programmability
- Re-programmability
  - PLDs can be reprogrammed without being removed from the circuit board.
- Low cost of design
- Immediate hardware implementation
- less board space
- lower power requirements (i.e., smaller power supplies)
- Faster assembly processes
- higher reliability (fewer ICs and circuit connections => easier troubleshooting)
- availability of design software

Types of PLDs

- SPLDs (Simple Programmable Logic Devices)
  - ROM (Read-Only Memory)
  - PLA (Programmable Logic Array)
  - PAL (Programmable Array Logic)
  - GAL (Generic Array Logic)
- HCPLD (High Capacity Programmable Logic Device)
  - CPLD (Complex Programmable Logic Device)
  - FPGA (Field-Programmable Gate Array)

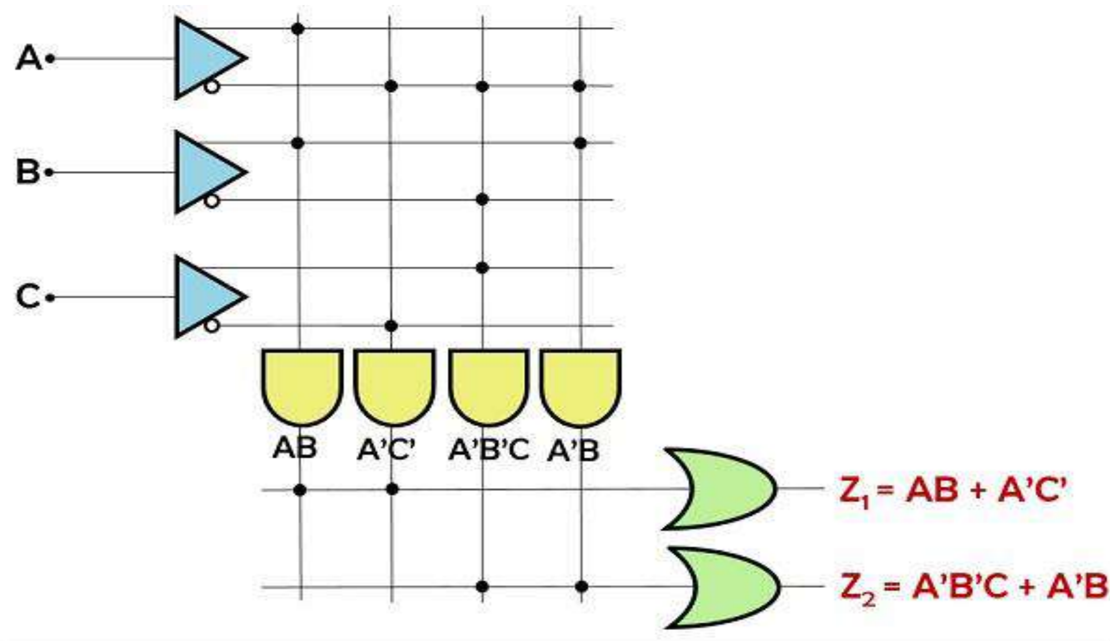


PLD Configuration

- Combination of a logic device and memory
- Memory stores the pattern the PLD was programmed with
  - EPROM
    - Non-volatile and reprogrammable
  - EEPROM
    - Non-volatile and reprogrammable
  - Static RAM (SRAM)
    - Volatile memory
  - Flash memory
    - Non-volatile memory
  - Antifuse
    - Non-volatile and no re-programmability

PLA

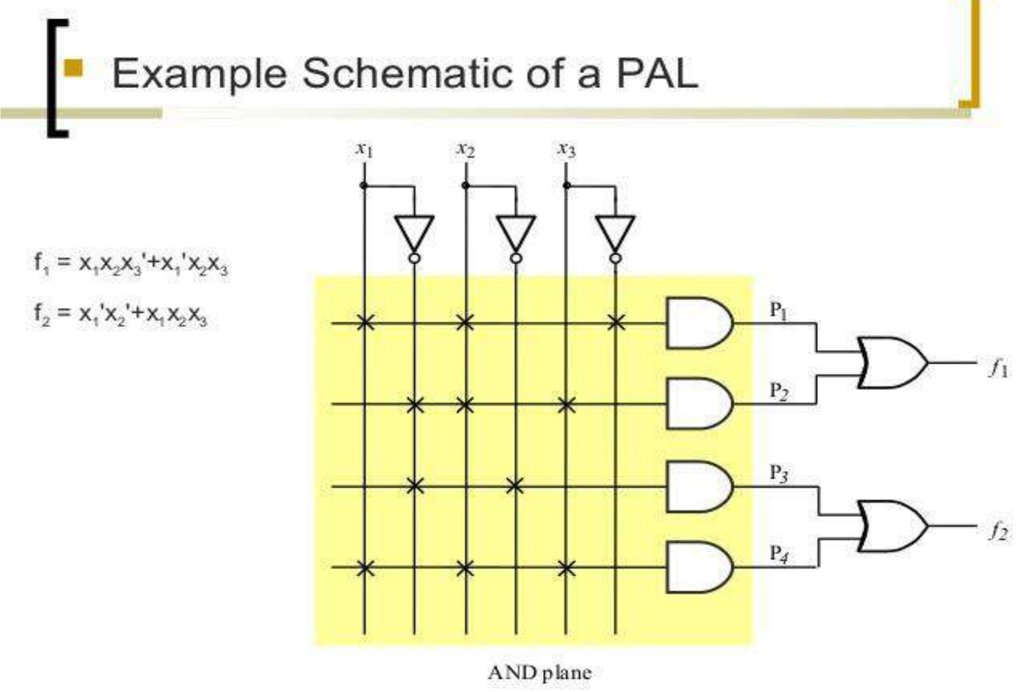
PLA: A programmable logic array (PLA) has a programmable AND gate array, which links to a programmable OR gate array



Implementation of Programmable Logic Array

PAL

PAL: PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane



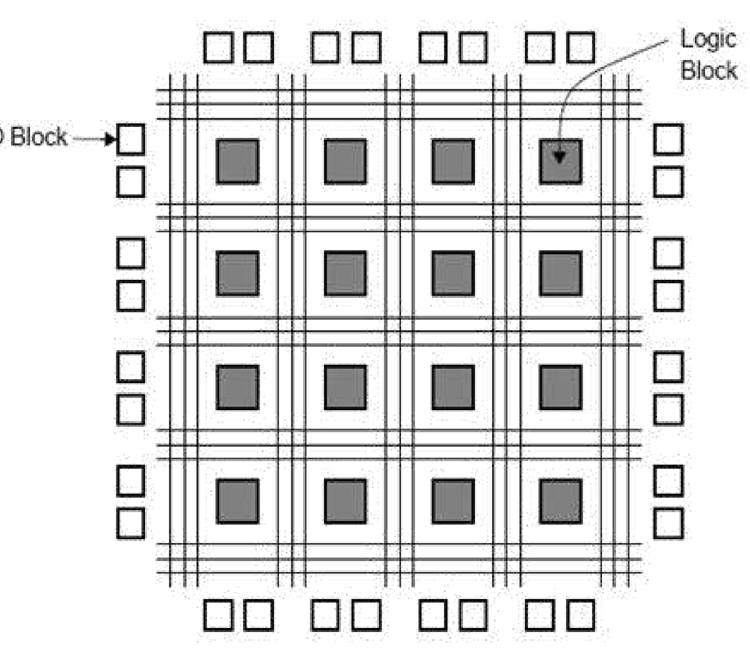
GAL

GAL: An improvement on the PAL was the Generic Array Logic device

- This device has the same logical properties as the PAL but can be erased and reprogrammed
- The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming
- GALs are programmed and reprogrammed using a PAL programmer

HCPLD

- CPLD (Complex Programmable Logic Device)
  - Lies between PALs and FPGAs in degree of complexity.
  - Inexpensive
- FPGA (Field-Programmable Gate Array)
  - Truly parallel design and operation
  - Fast turnaround design
  - Array of logic cells surrounded by programmable I/O blocks



FPGA

Number Systems

Overview

- **Introduction**
- **Number Systems** [binary, octal and hexadecimal]
- **Number System conversions**

Introduction

Number System

Code using symbols that refer to a number of items

Decimal Number System

Uses ten symbols (base 10 system)

Binary System

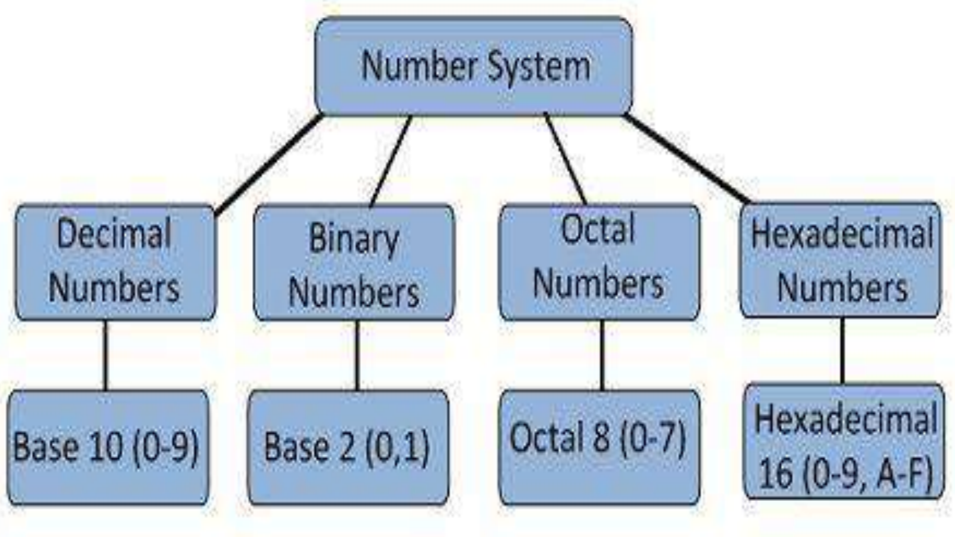
Uses two symbols (base 2 system)

Octal Number System

Uses eight symbols (base 8 system)

Hexadecimal Number System

Uses sixteen symbols (base 16 system)



Binary Number

- **Numeric value of symbols in different positions.**
- ***Example* - Place value in binary system:**

Place Value	8s	4s	2s	1s
Binary	Yes	Yes	No	No
Number	1	1	0	0

RESULT: Binary 1100 = decimal 8 + 4 + 0 + 0 = decimal 12

BINARY TO DECIMAL CONVERSION

Convert Binary Number 110011 to a Decimal Number:

Binary	1	1	0	0	1	1
	↓	↓	↓	↓	↓	↓
Decimal	32 +	16 +	0 +	0 +	2 +	1 = 51

TEST

Convert the following binary numbers into decimal numbers:

Binary 1001 =

Binary 1111 =

Binary 0010 =

TEST

Convert the following binary numbers into decimal numbers:

Binary 1001 = 9

Binary 1111 = 15

Binary 0010 = 2

DECIMAL TO BINARY CONVERSION

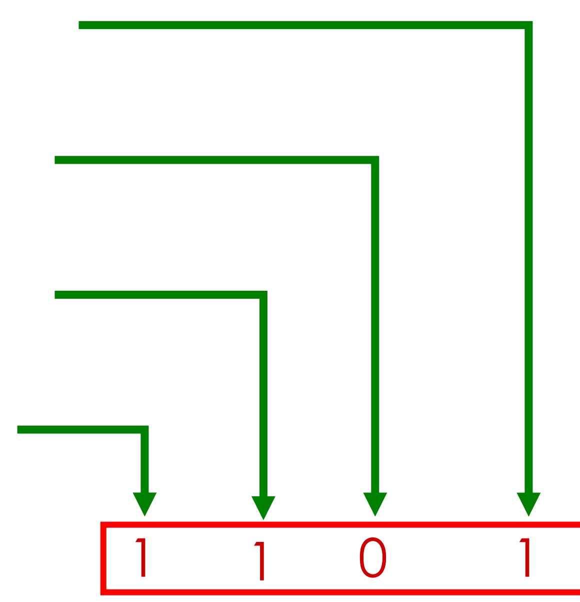
Divide by 2 Process

Decimal # 13 ÷ 2 = 6 remainder 1

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1



TEST

Convert the following decimal numbers into binary:

Decimal 11 =

Decimal 4 =

Decimal 17 =

TEST

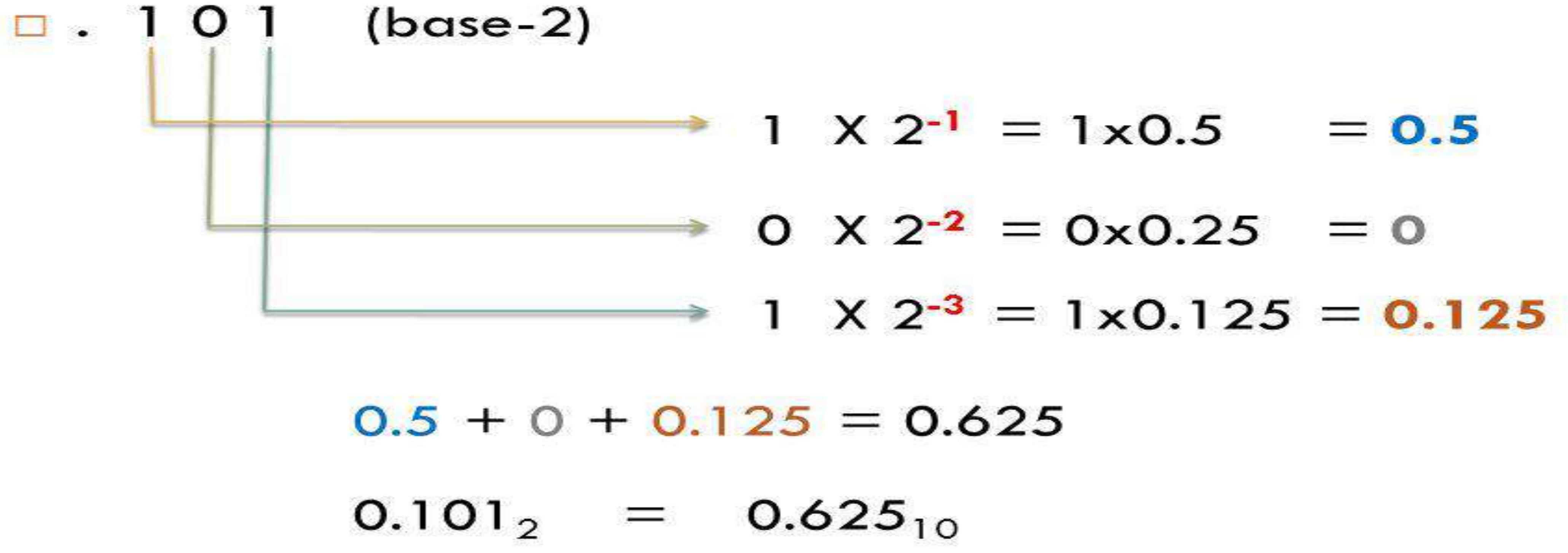
Convert the following decimal numbers into binary:

Decimal 11 = 1011

Decimal 4 = 0100

Decimal 17 = 10001

Binary-to-Decimal Conversion



Converting Decimal Fraction to Binary

- Convert $N = 0.6875$ to Radix 2
- Solution: Multiply N by 2 repeatedly & collect integer bits

Multiplication	New Fraction	Bit
0.6875 × 2 = 1.375	0.375	1
0.375 × 2 = 0.75	0.75	0
0.75 × 2 = 1.5	0.5	1
0.5 × 2 = 1.0	0.0	1

First fraction bit (first row, bit 1) … Last fraction bit (row 0.75 × 2, bit 1)

- Stop when new fraction = 0.0, or when enough fraction bits are obtained
- Therefore, $N = 0.6875 = (0.1011)_2$
- Check $(0.1011)_2 = 2^{-1} + 2^{-3} + 2^{-4} = 0.6875$

Digital Electronics by Parag P

HEXADECIMAL NUMBER SYSTEM

Uses 16 symbols -Base 16 System, 0-9, A, B, C, D, E, F

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10

HEXADECIMAL AND BINARY CONVERSIONS

- Hexadecimal to Binary Conversion

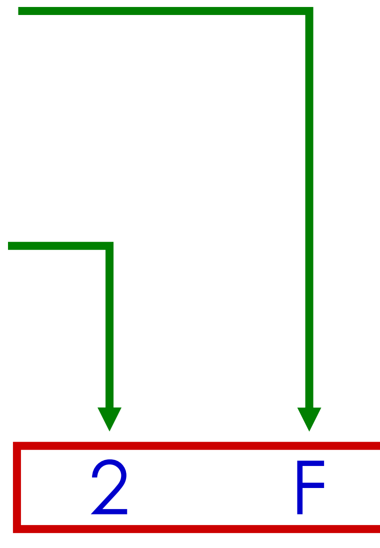
- Binary to Hexadecimal Conversion

DECIMAL TO HEXADECIMAL CONVERSION

Divide by 16 Process

Decimal # 47 ÷ 16 = 2 remainder 15

2 ÷ 16 = 0 remainder 2



Decimal Fraction To Hex

- To convert Decimal fraction into Hex, multiply fractional part with 16 till you get fractional part 0.
- Example : convert 0.03125_{10} to Hex

	Integer Part	
0. 03125 * 16 =0.5	0	Write From Up to Down
0. 5 * 16 = 8.0	8	

➔ $0.03125_{10} = 0.08_{16}$

HEXADECIMAL TO DECIMAL CONVERSION

Convert hexadecimal number 2DB to a decimal number

Place Value	256s	16s	1s
Hexadecimal	2	D	B
	(256 x 2)	(16 x 13)	(1 x 11)
Decimal	512 +	208 +	11 = **731**

Hexadecimal System

The weight associated with each symbol in the given hexadecimal number can be determined by raising 16 to a power equivalent to the position of the digit in the number.

Example 4A90.2BC

Digit	4	A	9	0	.	2	B	C
Weight	16^3	16^2	16^1	16^0	Hexadecimal Point	16^{-1}	16^{-2}	16^{-3}

Example

The following shows that the number (2AE)16 in hexadecimal is equivalent to 686 in decimal.

		16^2		16^1		16^0	Place values
		2		A		E	Number
N	=	2×16^2	+	10×16^1	+	14×16^0	Values

The equivalent decimal number is N = 512 + 160 + 14 = 686.

TEST

Convert Hexadecimal number A6 to Binary

A6 = 1010 0110 (Binary)

Convert Hexadecimal number 16 to Decimal

16 = 22 (Decimal)

Convert Decimal 63 to Hexadecimal

63 = 3F (Hexadecimal)

- Translate every hexadecimal digit into its 4-bit binary equivalent
- Examples:

$(3A5)_{16} = (0011\ 1010\ 0101)_2$

$(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$

$(1.8)_{16} = (0001\ .\ 1000)_2$

OCTAL NUMBERS

Uses 8 symbols -Base 8 System
0, 1, 2, 3, 4, 5, 6, 7

Decimal	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7
8	001 000	10
9	001 001	11

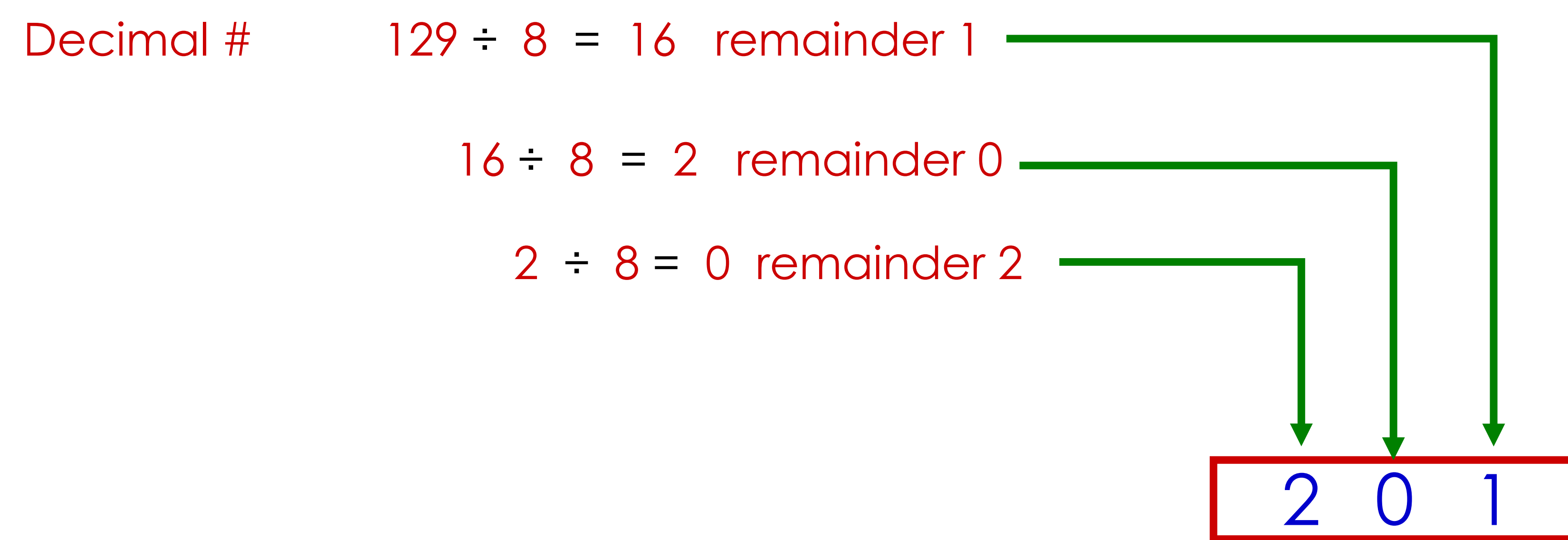
OCTAL AND BINARY CONVERSIONS

- Octal to Binary Conversion

- Binary to Octal Conversion

DECIMAL TO OCTAL CONVERSION

Divide by 8 Process



Fraction Decimal to Octal Conversion - Example

- Example: convert 0.356_{10} to octal.

0.356 * 8 = 2.848 → integer part = 2
0.848 * 8 = 6.784 → integer part = 6
0.784 * 8 = 6.272 → integer part = 6
0.272 * 8 = 2.176 → integer part = 2
0.176 * 8 = 1.408 → integer part = 1
0.408 * 8 = 3.264 → integer part = 3, etc.

Answer = $0.266213\ldots_8$

OCTAL TO DECIMAL CONVERSION

Convert octal number 201to a decimal number

Place Value	64s		8s		1s			
Octal	2		0		1			
	(64 x 2)		(8 x 0)		(1 x 1)			
Decimal	128	+	0	+	1	=	**129**	

- **Octal fraction to decimal**
- **Example**
- Convert $(23.25)8$ to decimal
- $8^1\ 8^0\ .\ 8^{-1}\ 8^{-2}$

  2 3 2 5

$= (2 \times 8^1) + (3 \times 8^0) + (2 \times 8^{-1}) + (5 \times 8^{-2})$

$= 16+3+0.25+0.07812$

$= (19.32812)10$

Binary, Octal, and Hexadecimal

- Binary, Octal, and Hexadecimal are related:

  Radix 16 = 2^4 and Radix 8 = 2^3

- Hexadecimal digit = 4 bits and Octal digit = 3 bits
- Starting from least-significant bit, group each 4 bits into a hex digit or each 3 bits into an octal digit
- Example: Convert 32-bit number into octal and hex

Octal	3	5	3	0	5	5	2	3	6	2	4

32-bit binary	1	1	1	0	1	0	1	1	0	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0

Hexadecimal	E	B	1	6	A	7	9	4

Digital Electronics by Parag P.

Convert 0.10111_2 to base 8: **0.101_110 = $\mathbf{0.56_8}$**

Convert 0.1110101 to base 16: **0.1110_1010 = $\mathbf{0.EA_{16}}$**

Arithmetic Operations

Overview

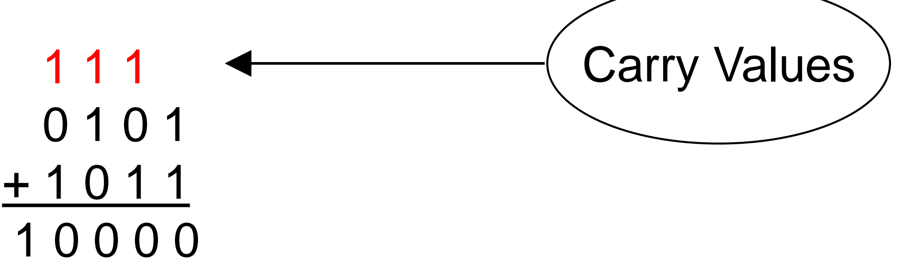
- **Arithmetic Operations**
- **Decimal Arithmetic**
- **Binary Arithmetic**
- **Signed Binary Numbers**

Arithmetic Operations

Addition

- Follow same rules as in decimal addition, with the difference that when sum is 2 indicates a carry (not a 10)
- Learn new carry rules
  - 0+0 = sum 0 carry 0
  - 0+1 = 1+0 = sum 1carry 0
  - 1+1 = sum 0 carry1
  - 1+1+1 = sum 1carry1

Carry	1	1	1	1	1	0
Augend	0	0	1	0	0	1
Addend	0	1	1	1	1	1
Result	1	0	1	0	0	0



Subtraction

- Learn new borrow rules
  - 0-0 = 1-1 = 0 borrow 0
  - 1-0 = 1 borrow 0
  - 0-1 = 1 borrow 1

Borrow	1	1	0	0	
Minuend	1	1	0	1	1
Subtrahend	0	1	1	0	1
Result	0	1	1	1	0

The rules of the decimal base applies to binary as well. To be able to calculate 0-1, we have to "borrow one" from the next left digit.

```
  1 2
 0 2 0 2
  1 0 1 0
- 0 1 1 1
  0 0 1 1
```

Decimal Subtraction

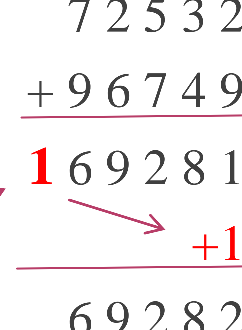
- 9's Complement Method
- 10's Complement Method

Example: 72532 – 3250

9's complement of 3250 is

9 9 9 9 9 – 0 3 2 5 0 = 9 6 7 4 9

If Carry, result is positive.
Add carry to the partial result

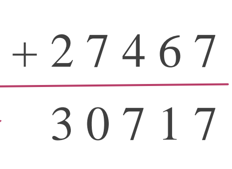


Example: 3250 – 72532

9's complement of 72532 is

9 9 9 9 9 – 7 2 5 3 2 = 2 7 4 6 7

If no Carry, result is negative.
Magnitude is 9's complement of the result



Decimal Subtraction

- 9's Complement Method
- 10's Complement Method

10's Complement Method

Example: 72532 – 3250

10's complement of 3250 is

1 0 0 0 0 0 – 0 3 2 5 0 = 9 6 7 5 0

```
  7 2 5 3 2
+ 9 6 7 5 0
-----------
1 6 9 2 8 2
```

Result is 6 9 2 8 2

If Carry, result is positive.
Discard the carry

Example: 3250 – 72532

10's complement of 72532 is

1 0 0 0 0 0 – 7 2 5 3 2 = 2 7 4 6 8

```
  0 3 2 5 0
+ 2 7 4 6 8
-----------
  3 0 7 1 8
= – 6 9 2 8 2
```

If no Carry, result is negative.
Magnitude is 10's complement of the result

Binary Subtraction

- 1's Complement Method
- 2's Complement Method

1's Complement Method

Example: 1010100 – 1000100

1's complement of 1000100 is 0111011

```
  1010100
+ 0111011
1 0001111
       +1
  0010000
```

If Carry, result is positive.
Add carry to the partial result

Example: 1000100 – 1010100

1's complement of 1010100 is 0101011

```
    1000100
  + 0101011
    1101111
= – 0010000
```

If no Carry, result is negative.
Magnitude is 1's complement of the result

Binary Subtraction

- 1's Complement Method
- 2's Complement Method

2's Complement Method

Example: 1010100 – 1000100

2's complement of 1000100 is 0111100

```
  1 0 1 0 1 0 0
+ 0 1 1 1 1 0 0
1 0 0 1 0 0 0 0
```

If Carry, result is positive.
Discard the carry

0 0 1 0 0 0 0

Example: 1000100 – 1010100

2's complement of 1010100 is 0101100

```
  1 0 0 0 1 0 0
+ 0 1 0 1 1 0 0
  1 1 1 0 0 0 0
= – 0 0 1 0 0 0 0
```

If no Carry, result is negative.
Magnitude is 2's complement of the result

Signed Binary Numbers

- When a signed binary number is positive
  - The MSB is ‘0’ which is the sign bit and rest bits represents the magnitude

- When a signed binary number is negative
  - The MSB is ‘1’ which is the sign bit and rest of the bits may be represented by three different ways
    - Signed magnitude representation
    - Signed 1’s complement representation
    - Signed 2’s complement representation

Signed Binary Numbers

	- 9	**+ 9**
Signed magnitude representation	1 1001	0 1001
Signed 1's complement representation	1 0110	0 1001
Signed 2's complement representation	1 0111	0 1001

	- 0	**+ 0**
Signed magnitude representation	1 0000	0 0000
Signed 1's complement representation	1 1111	0 0000
Signed 2's complement representation	-None-	0 0000

Range of Binary Number

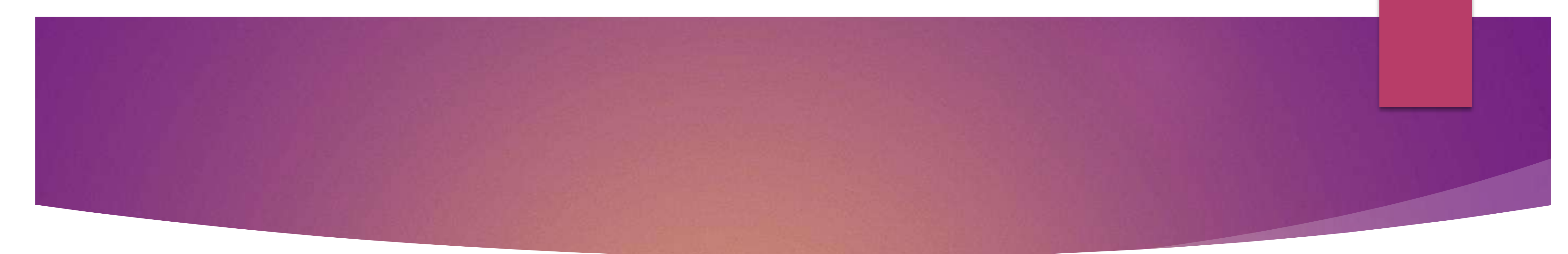
Binary Number of n bits

- General binary number: $(2^n - 1)$
- Signed magnitude binary number: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
- Signed 1's complement binary number: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
- Signed 2's complement binary number: $-(2^{n-1})$ to $+(2^{n-1} - 1)$

Signed Binary Number Arithmetic

- Add or Subtract two signed binary number including its sign bit either signed 1's complement method or signed 2's complement method
- The 1's complement and 2's complement rules of general binary number is applicable to this

- It is important to decide how many bits we will use to represent the number
- Example: Representing +5 and -5 on 8 bits:
  - +5: 00000101
  - -5: 10000101
- *So the very first step we have to decide on the number of bits to represent number*



Digital Codes

Overview

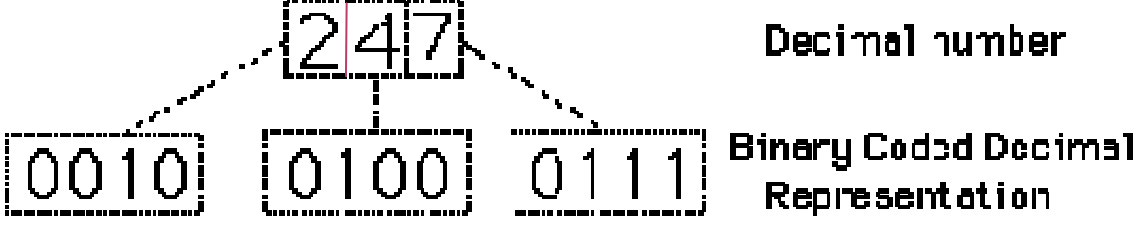
- **Introduction**
- **Binary Coded Decimal Code**
- **EBCDIC Code**
- **Excess-3 Code**
- **Gray Code**
- **ASCII Code**

Introduction

- Calculations or computations are not useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.
- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.
- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit BCD codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

Binary Coded Decimal (BCD)



Convert 0100 0010 1000 0010 BCD to decimal

0100	0010	1000	0110	BCD
4	**2**	**8**	**6**	Decimal

Decimal	BCD code Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

- Consider 5 + 5
- 5 0 1 0 1
- +5 0 1 0 1
- giving 1 0 1 0 which is binary 10 but not a BCD digit!
- What to do?
- Try adding 6??

- Had 1010 and want to add 6 or 0110
  - so 1 0 1 0
  - plus 6 0 1 1 0
  - Giving 1 0 0 0 0

- Add 7 + 6
  - have 7 0 1 1 1
  - plus 6 0 1 1 0
  - Giving 1 1 0 1 and again out of range
  - Adding 6 0 1 1 0
  - Giving 1 0 0 1 1 so a 1 carries out to the next BCD digit
  - FINAL BCD answer 0001 0011 or 13_{10}

6	0110	BCD for 6	42	0100 0010	BCD for 42
+3	0011	BCD for 3	+27	0010 0111	BCD for 27
9	1001	BCD for 9	69	0110 1001	BCD for 69

- Add the BCD for 417 to 195
- Would expect to get 612
  - BCD setup - start with Least Significant Digit
  - 0 1 0 0 0 0 0 1 0 1 1 1
  - 0 0 0 1 1 0 0 1 0 1 0 1
  - 1 1 0 0
  - Adding 6 0 1 1 0
  - Gives 1 0 0 1 0
- Had a carry to the 2nd BCD digit position
  - 1
  - 0 1 0 0 0 0 0 1 done
  - 0 0 0 1 1 0 0 1 0 0 1 0
  - 1 0 1 1
  - Again must add 6 0 1 1 0
  - Giving 1 0 0 0 1
  - And another carry

- Had a carry to the 3rd BCD digit position
  - 1
  - 0 1 0 0 done done
  - 0 0 0 1 0 0 0 1 0 0 1 0
  - 0 1 1 0
  - And answer is 0110 0001 0010 or the BCD for the base 10 number 612

EBCDIC Code

- The EBCDIC code is an 8-bit alphanumeric code that was developed by IBM to represent alphabets, decimal digits and special characters, including control characters.
- The EBCDIC codes are generally the decimal and the hexadecimal representation of different characters.
- This code is rarely used by non IBM-compatible computer systems.

The Excess-3- Code

DECIMAL	BCD	EXCESS-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

(a) 13 **(b) 430**

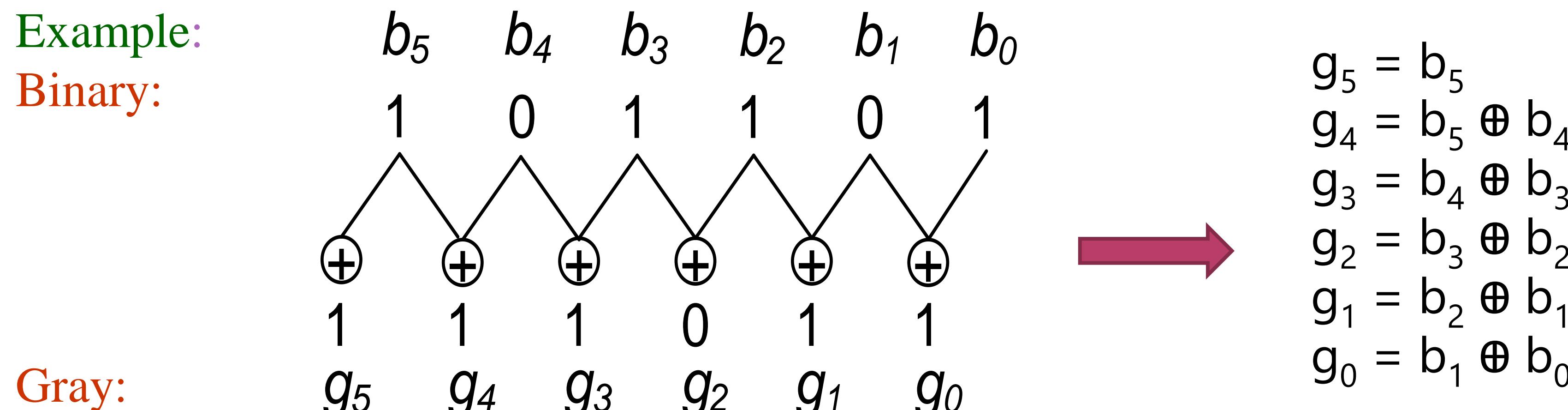
	1	3		(b)	4	3	0	
+	$\underline{3}$	$\underline{3}$			$\underline{3}$	$\underline{3}$	$\underline{3}$	
	4	6			7	6	3	
	0100	0110	Excess-3		0111	0110	0011	Excess-3

- Excess-3 code is self complementary code? Justify.

Gray Code

- Gray code is another important code that is also used to convert the decimal number into 8-bit binary sequence. However, this conversion is carried in a manner that the contiguous digits of the decimal number differ from each other by one bit only
- In pure binary coding or 8421 BCD then counting from 7 (0111) to 8 (1000) requires 4 bits to be changed simultaneously
- Gray coding avoids this since only one bit changes between subsequent numbers

Binary to Gray



$$g_5 = b_5$$
$$g_4 = b_5 \oplus b_4$$
$$g_3 = b_4 \oplus b_3$$
$$g_2 = b_3 \oplus b_2$$
$$g_1 = b_2 \oplus b_1$$
$$g_0 = b_1 \oplus b_0$$

Gray to Binary

Decimal number	Gray				Binary			
	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

$b_5 = g_5$

$b_4 = g_5 \oplus g_4$

$b_3 = g_5 \oplus g_4 \oplus g_3$

$b_2 = g_5 \oplus g_4 \oplus g_3 \oplus g_2$

$b_1 = g_5 \oplus g_4 \oplus g_3 \oplus g_2 \oplus g_1$

$b_0 = g_5 \oplus g_4 \oplus g_3 \oplus g_2 \oplus g_1 \oplus g_0$

Reflection of Gray Codes

2-bit	3-bit	4-bit
00	0 00	0 000
01	0 01	0 001
11	0 11	0 011
10	0 10	0 010
	1 10	0 110
	1 11	0 111
	1 01	0 101
	1 00	0 100
		1 100
		1 101
		1 111
		1 110
		1 010
		1 011
		1 001
		1 000

So, called reflected code

Alphanumeric Codes

- How do you handle alphanumeric data?
- Easy answer!
- Formulate a binary code to represent characters! ☺
- For the 26 letter of the alphabet would need 5 bit for representation.
- But what about the upper case and lower case, and the digits, and special characters

ASCII

- ASCII stands for American Standard Code for Information Interchange
- The code uses 7 bits to encode 128 unique characters
- Formally, work to create this code began in 1960. 1st standard in 1963. Last updated in 1986
- Represents the numbers
  - All start 011 xxxx and the xxxx is the BCD for the digit
- Represent the characters of the alphabet
  - Start with either 100, 101, 110, or 111
  - A few special characters are in this area
- Start with 010 – space and !"#$%&'()*+.-,/
- Start with 000 or 001 – control char like ESC

Table 1.7
American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Control characters

NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

ASCII Properties

ASCII has some interesting properties:

- Digits 0 to 9 span Hexadecimal values 30_{16} to 39_{16}
- Upper case A - Z span 41_{16} to $5A_{16}$
- Lower case a - z span 61_{16} to $7A_{16}$
  - Lower to upper case translation (and vice versa) occurs by flipping bit 6.
- Delete (DEL) is all bits set, a carryover from when punched paper tape was used to store messages.
- Punching all holes in a row erased a mistake!

Lecture of Module 2

Logic Gates

Overview

- **Introduction**
- **Logical Operators**
- **Basic Gates**
- **Universal Gates**
- **Realization of Basic Gates using Universal Gates**
- **Other Logic Gates**

Introduction

- Binary variables take on one of two values
- Logical operators operate on binary values and binary variables
- Basic logical operators are the logic functions AND, OR and NOT
- Logic gates implement logic functions
- Boolean Algebra: a useful mathematical system for specifying and transforming logic functions
- We study Boolean algebra as a foundation for designing and analyzing digital systems

Binary Variables

- **Recall that the two binary values have different names:**
  - **True/False**
  - **On/Off**
  - **Yes/No**
  - **1/0**
- **We use 1 and 0 to denote the two values.**
- **Variable identifier examples:**
  - **A, B, x, y, z, or X_1 , X_2 etc. for now**

Logical Operations

- The three basic logical operations are:
  - AND
  - OR
  - NOT
- AND is denoted by a dot (·)
- OR is denoted by a plus (+)
- NOT is denoted by an over bar (¯), a single quote mark (') after, or (~) before the variable

Operator

- Operators operate on binary values and binary variables
- Operations are defined on the values "0" and "1" for each operator:

AND	OR	NOT
$0 \cdot 0 = 0$	$0 + 0 = 0$	$\overline{0} = 1$
$0 \cdot 1 = 0$	$0 + 1 = 1$	$\overline{1} = 0$
$1 \cdot 0 = 0$	$1 + 0 = 1$	
$1 \cdot 1 = 1$	$1 + 1 = 1$	

Truth Tables

- ***Truth table*** - a tabular listing of the values of a function for all possible combinations of values on its arguments
- Example: Truth tables for the basic logic operations:

AND		
X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

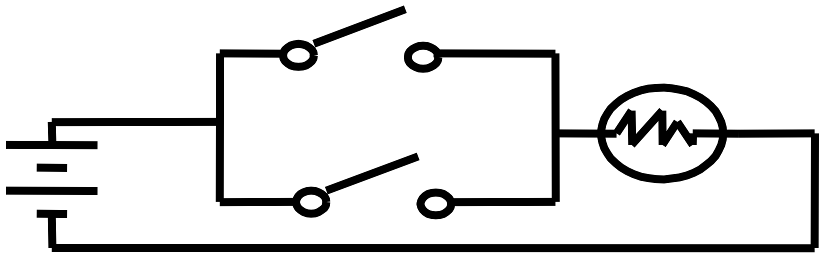
OR		
X	Y	$Z = X+Y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT	
X	$Z = \overline{X}$
0	1
1	0

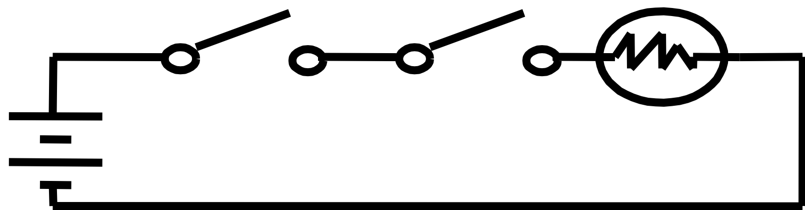
Logic Function Implementation

- Using Switches
  - For inputs:
    - logic 1 is switch closed
    - logic 0 is switch open
  - For outputs:
    - logic 1 is light on
    - logic 0 is light off.

Switches in parallel => OR



Switches in series => AND

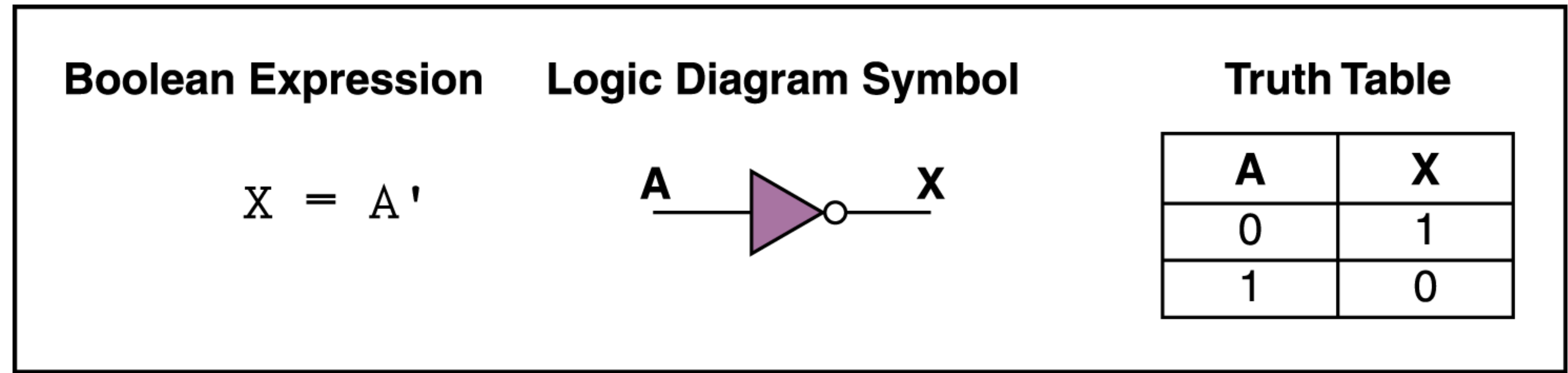


Logic Gates

- In the earliest computers, switches were opened and closed by magnetic fields produced by energizing coils in *relays*. The switches in turn opened and closed the current paths.
- Later, *vacuum tubes* that open and close current paths electronically replaced relays.
- Today, *transistors* are used as electronic switches that open and close current paths.
- NOT, AND and OR Gates (Basic gates)
- NAND and NOR Gates (Universal logic gates)

NOT Gate

A NOT gate accepts one input signal (0 or 1) and returns the opposite signal as output

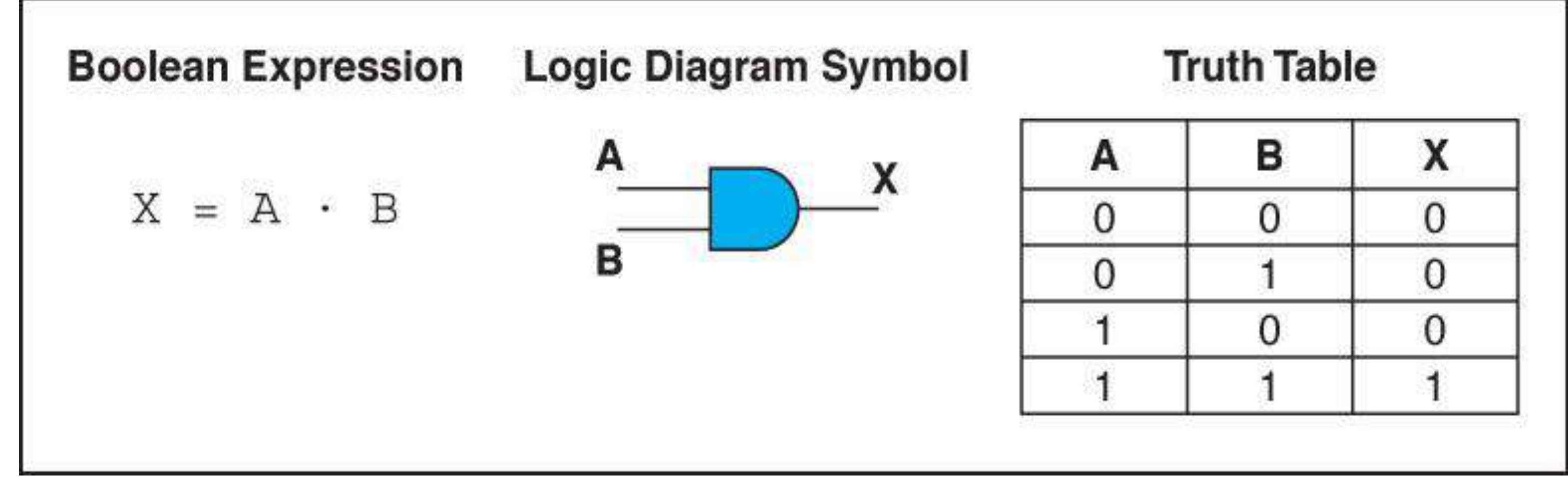


A	X
0	1
1	0

AND Gate

If all inputs are 1, the output is 1; otherwise, the output is 0

Or if any input is 0, output is 0

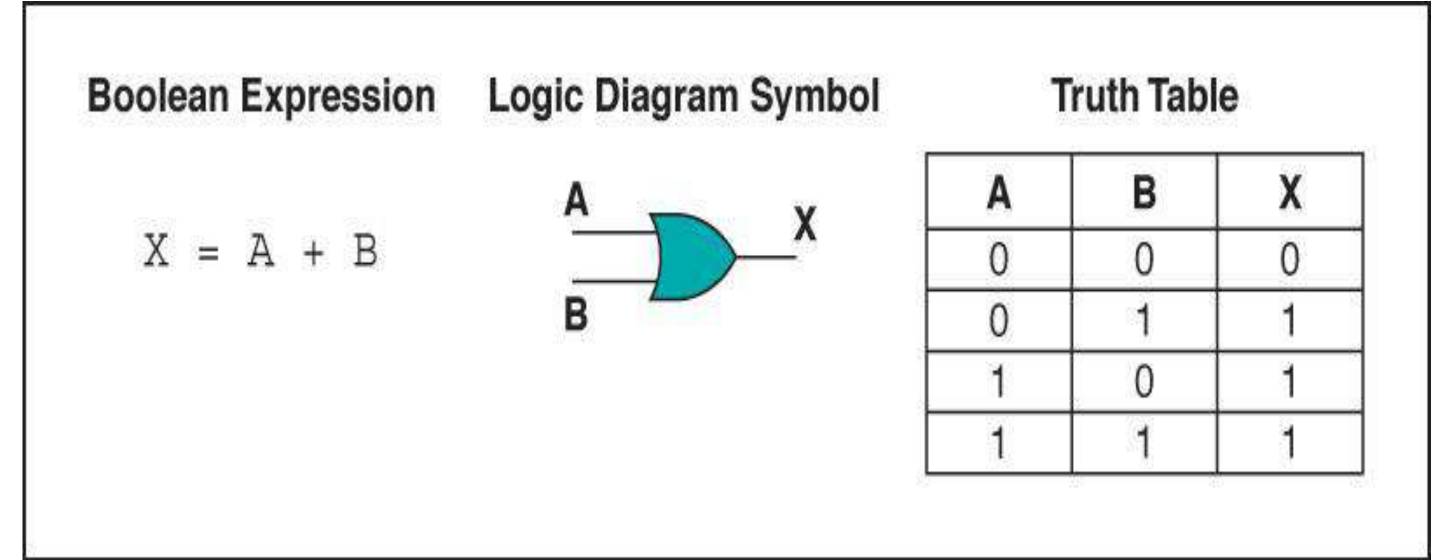


A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

If all inputs are 0, the output is 0; otherwise, the output is 1

Or if any input is 1, output will be 1



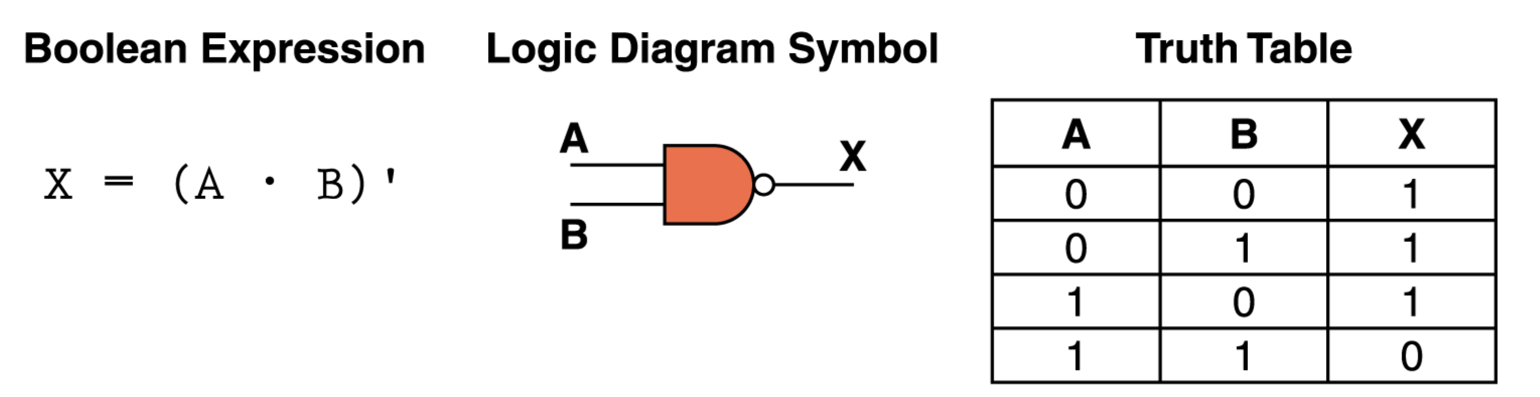
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Universal Gates

- Universal Logic Gate: Any basic gate or logic function can be realized using this gate
- Two universal logic gates
  - NAND
  - NOR

NAND Gate

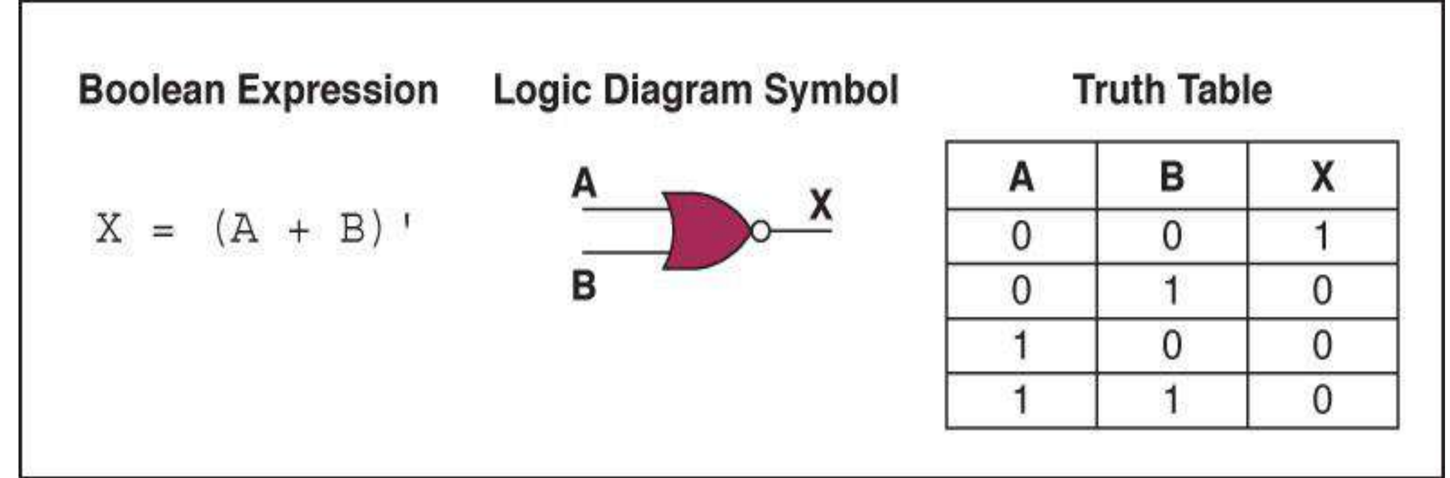
If all inputs are 1, the output is 0; otherwise, the output is 1



A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

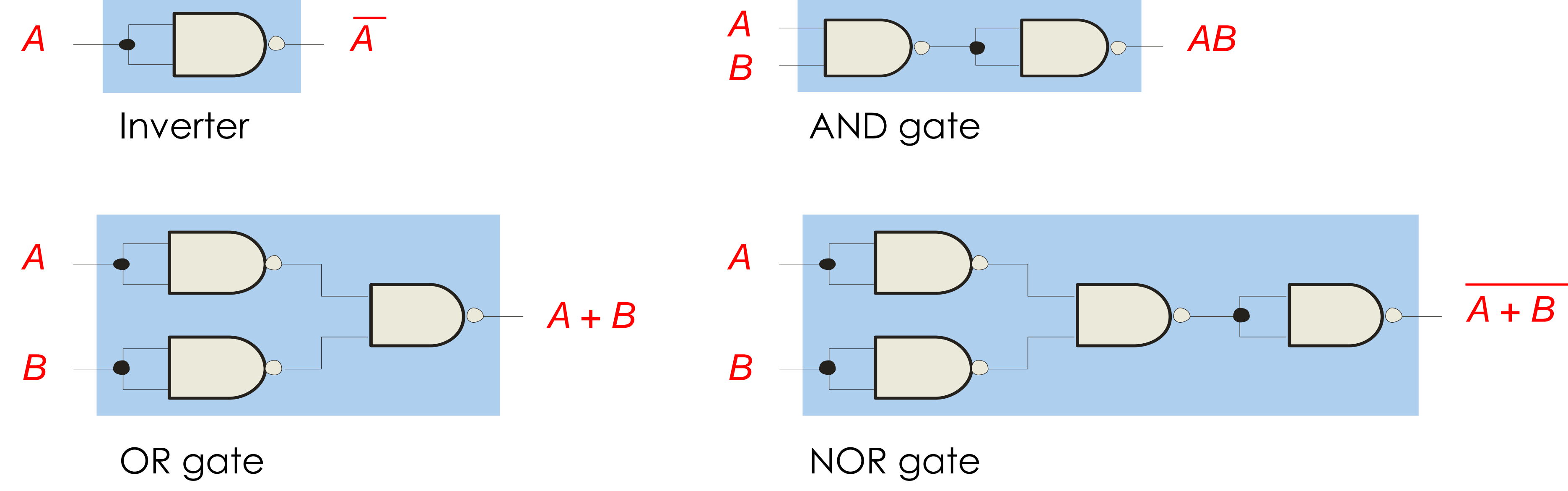
If all inputs are 0, the output is 1; otherwise, the output is 0



A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

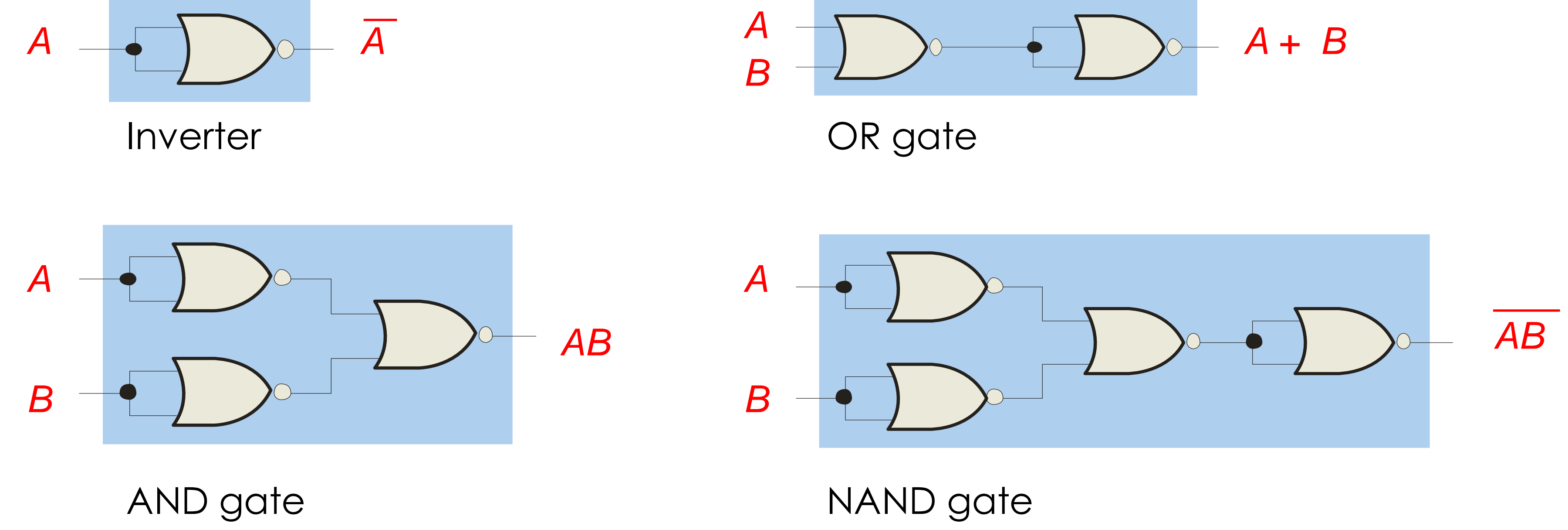
Realization

NAND gates are sometimes called **universal** gates because they can be used to produce the other basic Boolean functions.



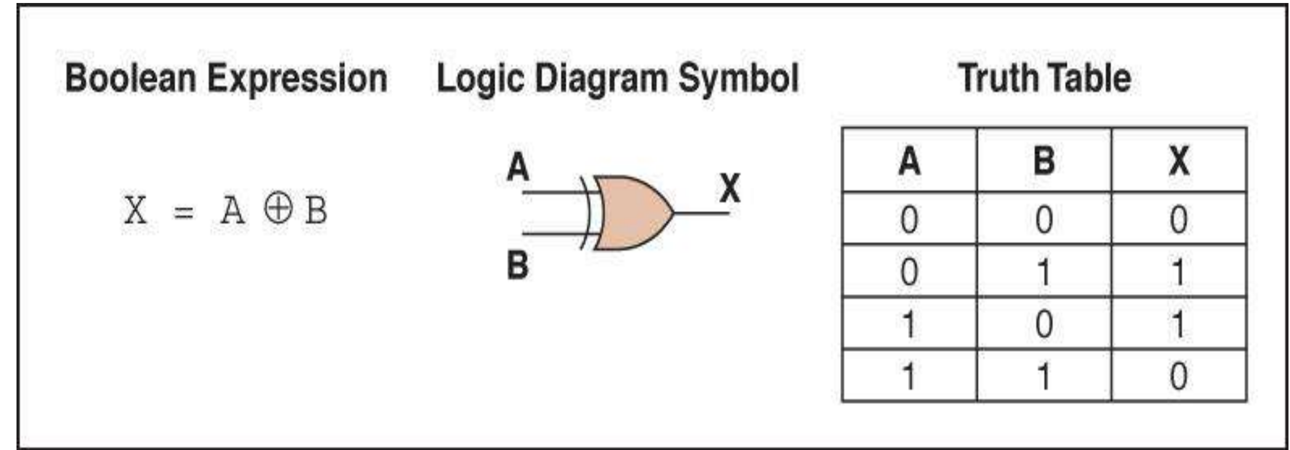
Realization

NOR gates are also **universal** gates and can form all of the basic gates.



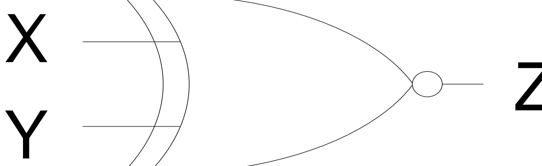
XOR Gate

If odd numbers of inputs are 1, the output is 1; otherwise, the output is 0



A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

X-NOR Gate



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

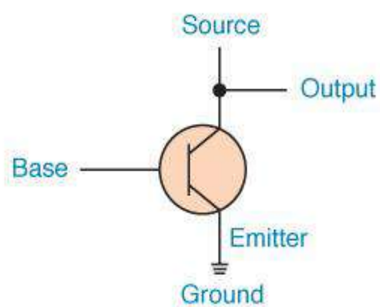
Constructing Gates

Transistor

A device that acts either as a wire that conducts electricity or as a resistor that blocks the flow of electricity, depending on the voltage level of an input signal

A transistor has no moving parts, yet acts like a switch

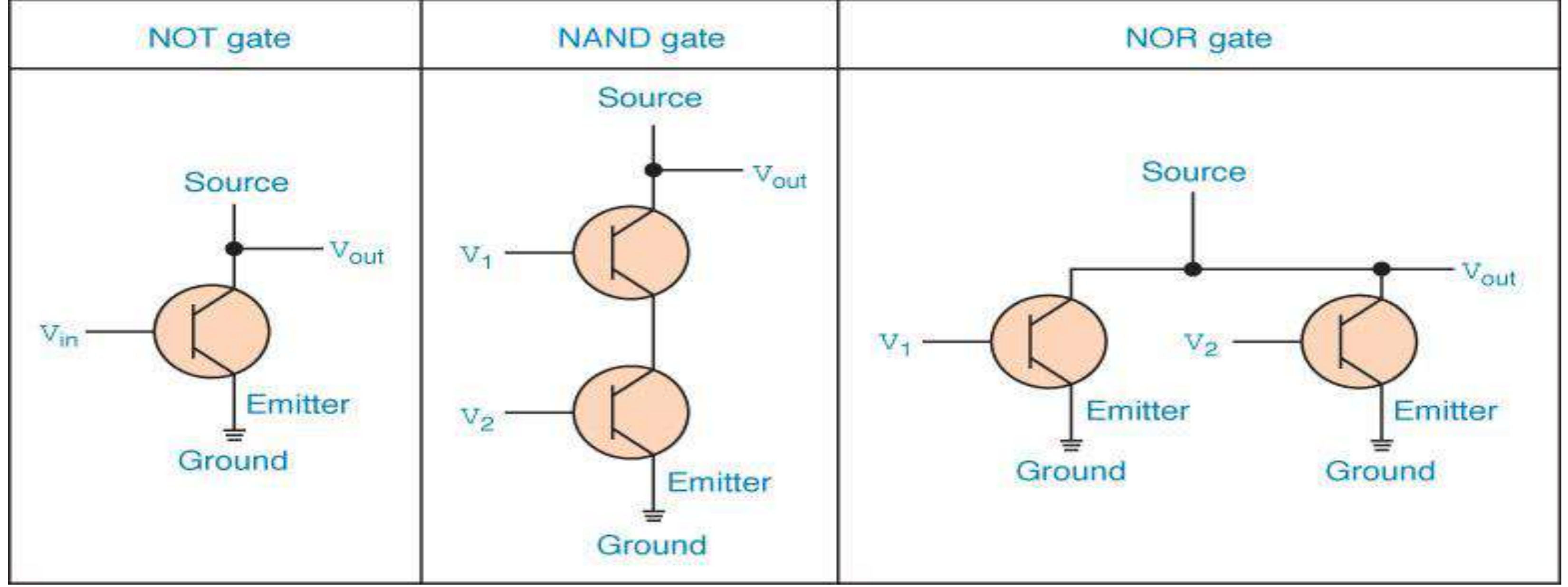
It is made of a semiconductor material, which is neither a particularly good conductor of electricity nor a particularly good insulator



A transistor has three terminals

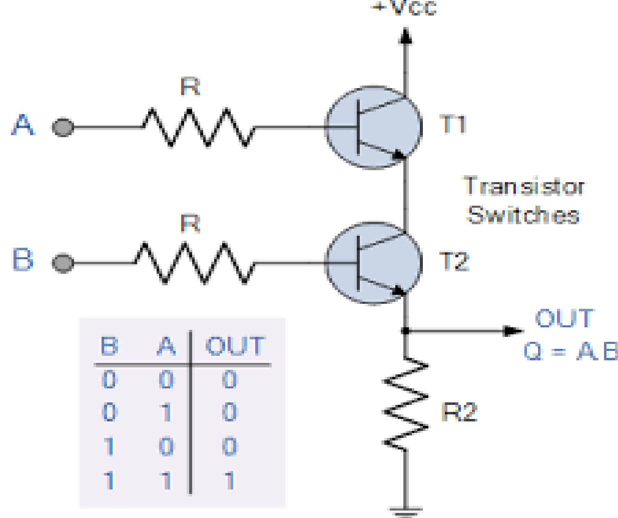
- A source
- A base
- An emitter, typically connected to a ground wire

If the electrical signal is grounded, it is allowed to flow through an alternative route to the ground (literally) where it can do no harm

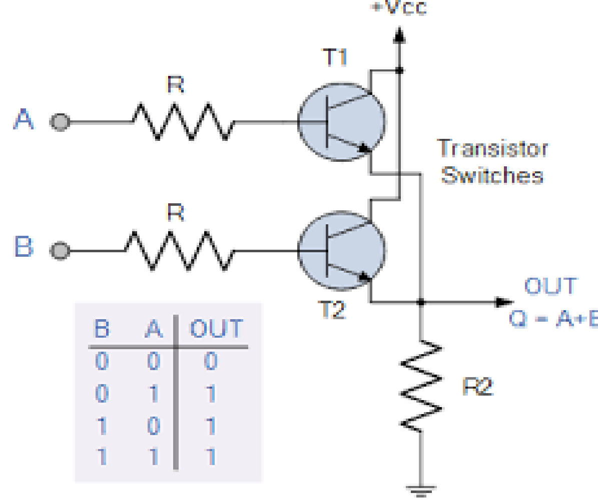


NOT gate
NAND gate
NOR gate
Source
Vout
Vin
Emitter
Ground
Source
Vout
V1
V2
Emitter
Ground
Source
Vout
V1
V2
Emitter
Emitter
Ground
Ground

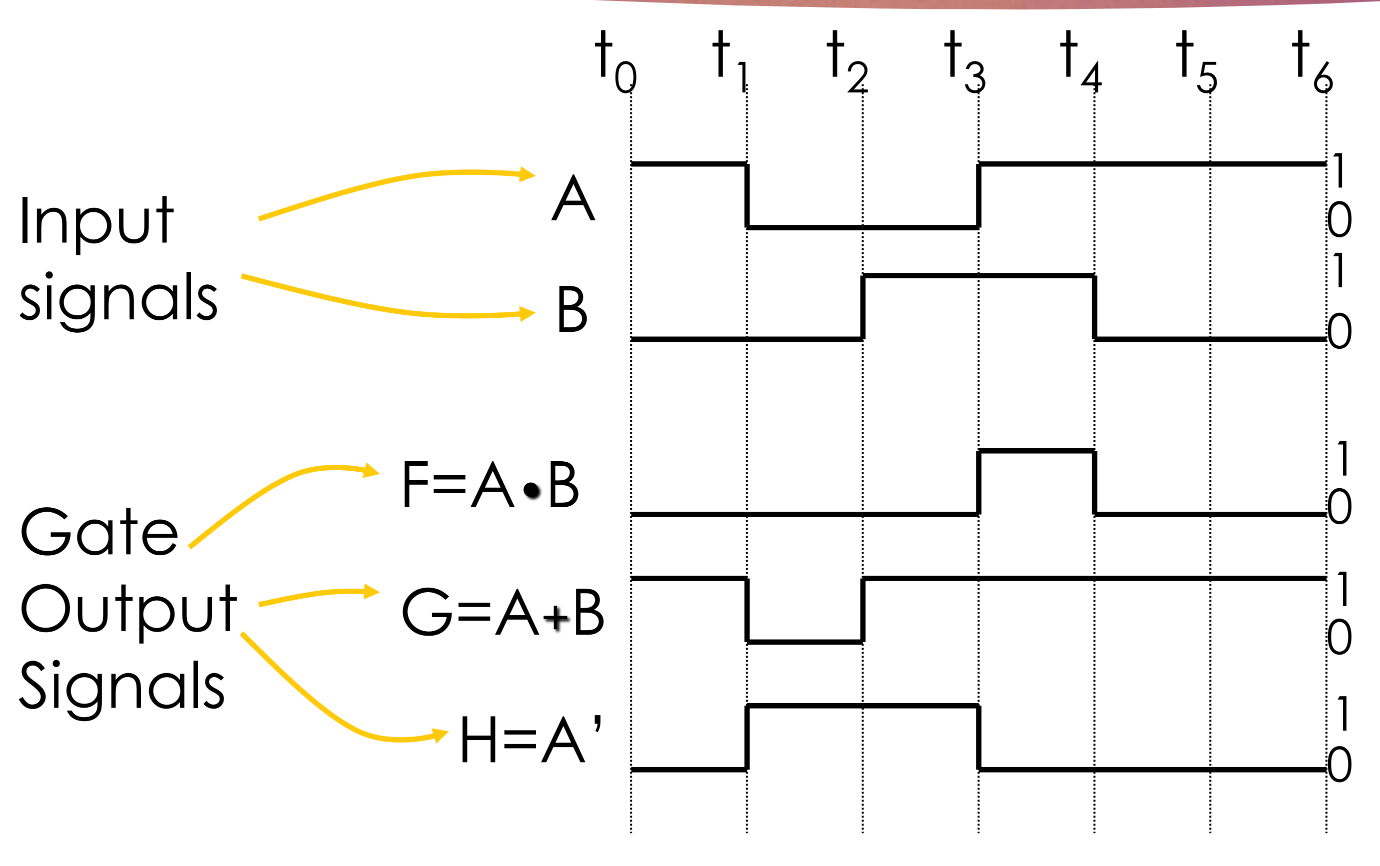
AND Gate



OR Gate



Timing Diagram

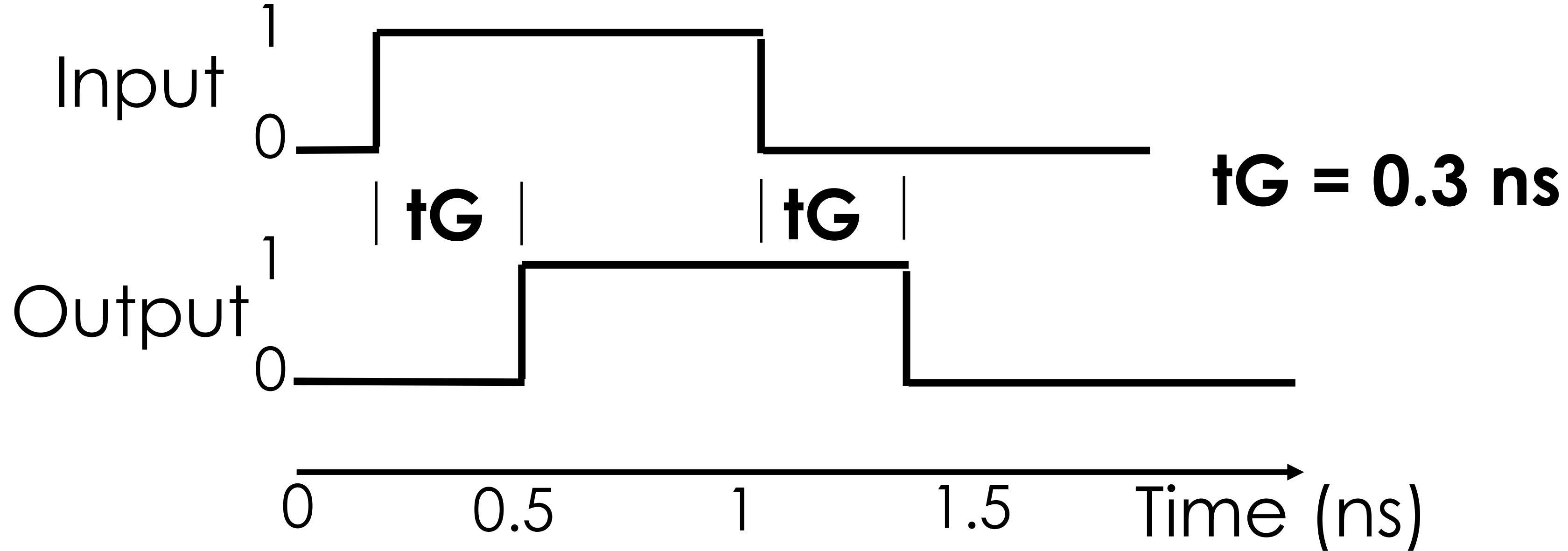


Transitions

<u>Basic Assumption:</u>
Zero time for signals to propagate Through gates

Gate Delay

- **In actual physical gates, if one or more input changes causes the output to change, the output change does not occur instantaneously.**
- **The delay between an input change(s) and the resulting output change is the *gate delay* denoted by t_G:**



Boolean Algebra

Overview

- Introduction
- Boolean Algebra
- Properties
- Algebraic Manipulation
- De-Morgan Theorem
- Complementation
- Truth Table

Introduction

- Understand the relationship between Boolean logic and digital computer circuits.
- Learn how to design simple logic circuits.
- Understand how digital circuits work together to form complex computer systems.
- In the latter part of the nineteenth century, **George Boole** suggested that logical thought could be represented through mathematical equations.
- Computers, as we know them today, are implementations of Boole's *Laws of Thought*.
- In this chapter, you will learn the simplicity that constitutes the essence of the machine (Boolean Algebra).

Boolean algebra

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
  - In formal logic, these values are "true" and "false."
  - In digital systems, these values are "on" and "off," 1 and 0, or "high" and "low."
- Boolean expressions are created by performing operations on Boolean variables.
  - Common Boolean operators include AND, OR, NOT, XOR, NAND and NOR

- A Boolean operator can be completely described using a truth table.
- The truth table for the Boolean operators AND, OR and NOT are shown at the right.
- The AND operator is also known as a Boolean product.
- The OR operator is the Boolean sum.
- The NOT operation is most often designated by an over-bar. It is sometimes indicated by a prime mark (‘) or an “elbow” (¬).

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

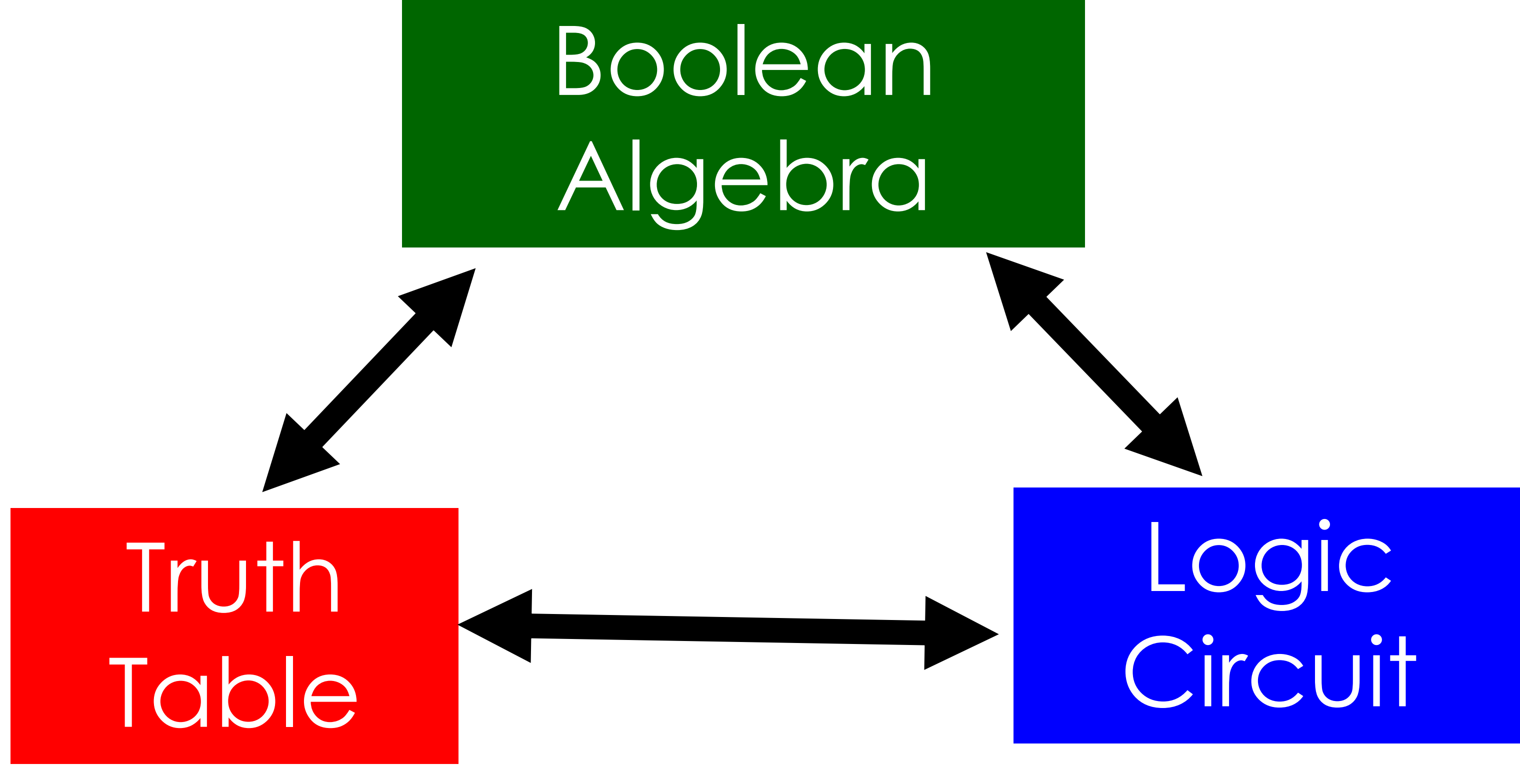
NOT X

X	$\overline{X}$
0	1
1	0

- A Boolean function has:
  - At least one Boolean variable,
  - At least one Boolean operator, and
  - At least one input from the set {0,1}
- It produces an output that is also a member of the set {0,1}

Now you know why the binary numbering system is so handy in digital systems

Conceptually



- Digital computers contain circuits that implement Boolean functions.
- The **simpler** that we can make a Boolean function, the **smaller** the circuit that will result.
  - Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
- With this in mind, we always want to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to do this.

Properties of Boolean Algebra

- Most Boolean identities have an AND (product) form as well as an OR (sum) form.

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\overline{x} = 0$	$x + \overline{x} = 1$

- Our second group of Boolean identities should be familiar to you from your study of algebra:

Identity Name	AND Form	OR Form
Commutative Law	xy = yx	x+y = y+x
Associative Law	(xy)z = x(yz)	(x+y)+z = x + (y+z)
Distributive Law	x+yz = (x+y)(x+z)	x(y+z) = xy+xz

- Our last group of Boolean identities are perhaps the most useful.
- If you have studied set theory or formal logic, these laws are also familiar to you.

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x+y)} = \bar{x}\bar{y}$
Double Complement Law	$\overline{(\bar{x})} = x$	

- We can use Boolean identities to simplify the function:

$$F(X,Y,Z) = (X + Y)(X + \overline{Y})(\overline{X\overline{Z}})$$

as follows:

$(X + Y)(X + \overline{Y})(\overline{X\overline{Z}})$	Idempotent Law (Rewriting)
$(X + Y)(X + \overline{Y})(\overline{X} + Z)$	DeMorgan's Law
$(XX + X\overline{Y} + XY + Y\overline{Y})(\overline{X} + Z)$	Distributive Law
$((X + Y\overline{Y}) + X(Y + \overline{Y}))(\overline{X} + Z)$	Commutative & Distributive Laws
$((X + 0) + X(1))(\overline{X} + Z)$	Inverse Law
$X(\overline{X} + Z)$	Idempotent Law
$X\overline{X} + XZ$	Distributive Law
$0 + XZ$	Inverse Law
XZ	Idempotent Law

With respect to duality, Identities 1 – 8 have the following relationship:

1. $X + 0 = X$ 2. $X \bullet 1 = X$ (dual of 1)
3. $X + 1 = 1$ 4. $X \bullet 0 = 0$ (dual of 3)
5. $X + X = X$ 6. $X \bullet X = X$ (dual of 5)
7. $X + X' = 1$ 8. $X \bullet X' = 0$ (dual of 8)

Algebraic Manipulation

- **Boolean algebra is a useful tool for simplifying digital circuits.**
- **Why do it? Simpler can mean cheaper, smaller, faster.**
- **Example: Simplify F = x'yz + x'yz' + xz.**
  F= x'yz + x'yz' + xz
  = x'y(z+z') + xz
  = x'y•1 + xz
  = x'y + xz
- **Example: Prove x'y'z' + x'yz' + xyz' = x'z' + yz'**
- **Proof: x'y'z'+ x'yz'+ xyz'**
  = x'y'z' + x'yz' + x'yz' + xyz'
  = x'z'(y'+y) + yz'(x'+x)
  = x'z'•1 + yz'•1
  = x'z' + yz'

Complementation

- Sometimes it is more economical to build a circuit using the complement of a function (and complementing its result) than it is to implement the function directly.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- DeMorgan's law states:

$$\overline{(xy)} = \overline{x} + \overline{y} \quad \text{and} \quad \overline{(x+y)} = \overline{x}\,\overline{y}$$

- **Find the complement of $F(x, y, z) = x\, y'\, z' + x'\, y\, z$**

- **$G = F' = (xy'z' + x'yz)'$**
  $= (xy'z')' \bullet (x'yz)'$ ***DeMorgan***
  $= (x'+y+z) \bullet (x+y'+z')$ ***DeMorgan*** **again**

- **Note: The complement of a function can also be derived by finding the function's *dual,* and then complementing all of the literals**

Truth Table

- Enumerates all possible combinations of variable values and the corresponding function value
- Truth tables for some arbitrary functions $F_1(x,y,z)$, $F_2(x,y,z)$, and $F_3(x,y,z)$ are shown to the right.
- Truth table: a <u>unique</u> representation of a Boolean function
- If two functions have identical truth tables, the functions are equivalent (and vice-versa).
- Truth tables can be used to prove equality theorems.
- However, the size of a truth table grows <u>exponentially</u> with the number of variables involved. This motivates the use of Boolean Algebra.

x	y	z		F_1	F_2	F_3
0	0	0		0	1	1
0	0	1		0	0	1
0	1	0		0	0	1
0	1	1		0	1	1
1	0	0		0	1	0
1	0	1		0	1	0
1	1	0		0	0	0
1	1	1		1	0	1

Standard SOP and POS

Overview

- Introduction
- SOP and POS
- Minterms and Maxterms
- Canonical Forms
- Conversion Between Canonical Forms
- Standard Forms

Introduction

- Through our exercises in simplifying Boolean expressions, we see that there are numerous ways of stating the same Boolean expression.
  - These "synonymous" forms are *logically equivalent.*
  - Logically equivalent expressions have identical truth tables.
- In order to eliminate as much confusion as possible, designers express Boolean functions in *standardized* or *canonical* form.

SOP and POS

- There are two canonical forms for Boolean expressions: Sum-Of-Products (SOP) and Product-Of-Sums (POS).
  - Recall the Boolean product is the AND operation and the Boolean sum is the OR operation.
- In the Sum-Of-Products form, ANDed variables are ORed together.
  - For example: `F(x,y,z) = xy + xz + yz`
- In the Product-Of-Sums form, ORed variables are ANDed together:
  - For example: `F(x,y,z) = (x+y)(x+z)(y+z)`

Definitions

- *Literal:* A variable or its complement
- *Product term:* literals connected by •
- *Sum term:* literals connected by +
- *Minterm:* a product term in which all the variables appear exactly once, either complemented or un-complemented
- *Maxterm:* a sum term in which all the variables appear exactly once, either complemented or un-complemented

Truth Table notation for Minterms and Maxterms

- Minterms and Maxterms are easy to denote using a truth table.
- Example:
  Assume 3 variables x,y,z (order is fixed)
- Any Boolean function F( ) can be expressed as a *unique* **sum** of **min**terms and a unique **product** of **max**terms (under a fixed variable ordering).
- In other words, every function F() has two canonical forms:
  - Canonical Sum-Of-Products (sum of minterms)
  - Canonical Product-Of-Sums (product of maxterms)

x	y	z		Minterm	Maxterm
0	0	0		$x'y'z' = m_0$	$x+y+z = M_0$
0	0	1		$x'y'z = m_1$	$x+y+z' = M_1$
0	1	0		$x'yz' = m_2$	$x+y'+z = M_2$
0	1	1		$x'yz = m_3$	$x+y'+z'= M_3$
1	0	0		$xy'z' = m_4$	$x'+y+z = M_4$
1	0	1		$xy'z = m_5$	$x'+y+z' = M_5$
1	1	0		$xyz' = m_6$	$x'+y'+z = M_6$
1	1	1		$xyz = m_7$	$x'+y'+z' = M_7$

Canonical Forms

- Canonical Sum-Of-Products:
  The minterms included are those m_j such that F( ) = 1 in row *j* of the truth table for F( ).
- Canonical Product-Of-Sums:
  The maxterms included are those M_j such that F( ) = 0 in row *j* of the truth table for F( ).

- $f_1(a,b,c) = \sum m(1,2,4,6)$, where $\sum$ indicates that this is a sum-of-products form, and $m(1,2,4,6)$ indicates that the minterms to be included are m_1, m_2, m_4, and m_6.
- $f_1(a,b,c) = \prod M(0,3,5,7)$, where $\prod$ indicates that this is a product-of-sums form, and $M(0,3,5,7)$ indicates that the maxterms to be included are M_0, M_3, M_5, and M_7.
- Since $m_j = M_j'$ for any *j*,
  $\sum m(1,2,4,6) = \prod M(0,3,5,7) = f_1(a,b,c)$

Conversion Between Canonical Forms

- Replace $\sum$ with $\prod$ (or *vice versa*) and replace those j's that appeared in the original form with those that do not.
- Example:

$$f_1(a,b,c) = a'b'c + a'bc' + ab'c' + abc'$$
$$= m_1 + m_2 + m_4 + m_6$$
$$= \sum(1,2,4,6)$$
$$= \prod(0,3,5,7)$$
$$= (a+b+c) \cdot (a+b'+c') \cdot (a'+b+c') \cdot (a'+b'+c')$$

$$F = \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7 = \sum m(0, 2, 5, 7)$$

$$\overline{F} = \overline{X}\,\overline{Y}Z + \overline{X}YZ + X\overline{Y}\,\overline{Z} + XY\overline{Z} = m_1 + m_3 + m_4 + m_6 = \sum m(1, 3, 4, 6)$$

$$\overline{F} = m_1 + m_3 + m_4 + m_6$$

$$\Rightarrow F = \overline{m_1 + m_3 + m_4 + m_6} = \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_6}$$

$$\Rightarrow F = M_1 \cdot M_3 \cdot M_4 \cdot M_6 = (X + Y + \overline{Z})(X + \overline{Y} + \overline{Z})(\overline{X} + Y + Z)(\overline{X} + \vec{Y} + Z)$$

$$= \prod M(1, 3, 4, 6)$$

Standard Forms

- Standard forms are "*like*" canonical forms, except that not all variables need appear in the individual product (SOP) or sum (POS) terms.
- Example:
  $f_1(a,b,c) = a'b'c + bc' + ac'$
  is a *standard* sum-of-products form
- $f_1(a,b,c) = (a+b+c) \bullet (b'+c') \bullet (a'+c')$
  is a *standard* product-of-sums form.

Conversion of SOP from standard to canonical form

- Expand *non-canonical* terms by inserting equivalent of 1 in each missing variable x:
  $(x + x') = 1$
- Remove duplicate minterms
- $f_1(a,b,c) = a'b'c + bc' + ac'$
  $= a'b'c + (a+a')bc' + a(b+b')c'$
  $= a'b'c + abc' + a'bc' + abc' + ab'c'$
  $= a'b'c + abc' + a'bc' + ab'c'$

Conversion of POS from standard to canonical form

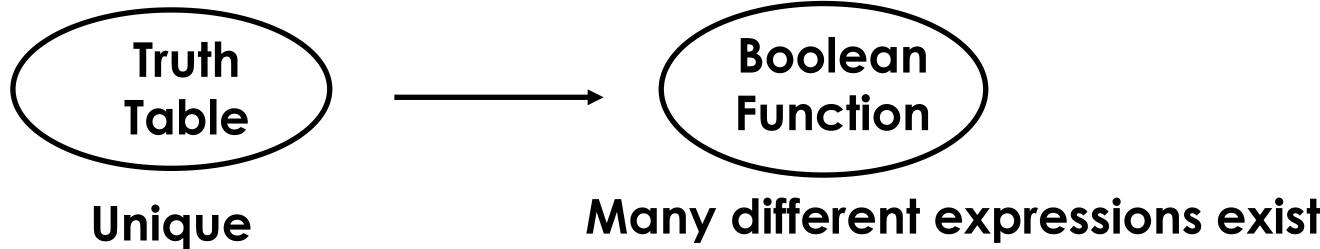
- Expand non-canonical terms by adding 0 in terms of missing variables (*e.g.*, $xx' = 0$) and using the distributive law
- Remove duplicate maxterms
- $f_1(a,b,c) = (a+b+c)\bullet(b'+c')\bullet(a'+c')$
  $= (a+b+c)\bullet(aa'+b'+c')\bullet(a'+bb'+c')$
  $= (a+b+c)\bullet(a+b'+c')\bullet(a'+b'+c')\bullet(a'+b+c')\bullet(a'+b'+c')$
  $= (a+b+c)\bullet(a+b'+c')\bullet(a'+b'+c')\bullet(a'+b+c')$

Minimization Techniques

Overview

- Introduction
- Karnaugh Map (K-Map)
- Simplification Rules
- K-Map Simplification for Two Variables
- K-Map Simplification for Three Variables
- K-Map Simplification for Four Variables
- Don’t Care Conditions
- Redundancy
- Design of Combinational Circuits

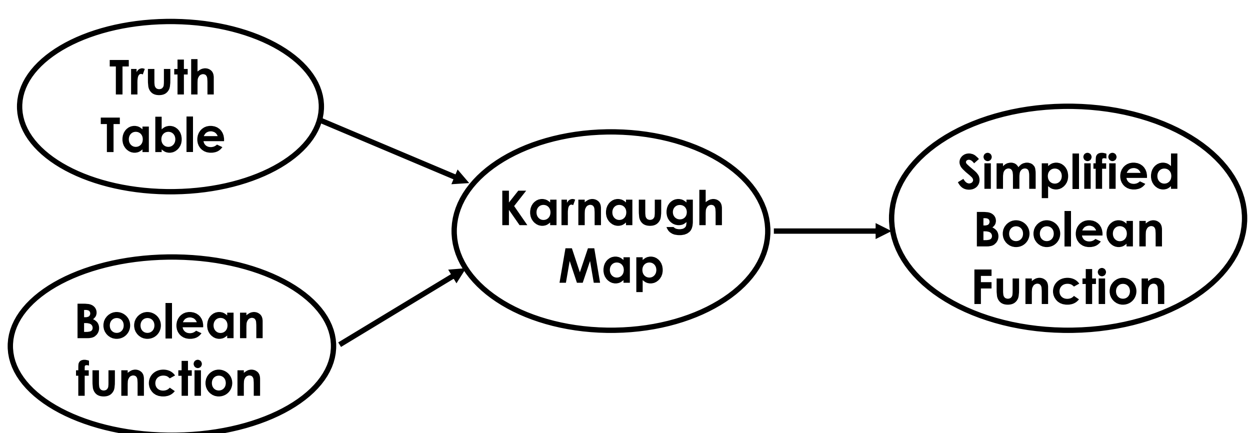
Introduction



Simplification from Boolean function

- Finding an equivalent expression that is least expensive to implement
- For a simple function, it is possible to obtain a simple expression for low cost implementation
- But, with complex functions, it is a very difficult for implementation

Karnaugh Map (K-map) is a simple procedure for simplification of Boolean expressions.



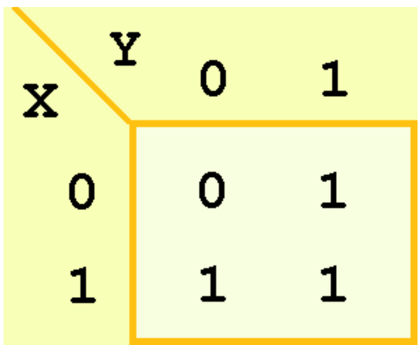
Karnaugh Map (K-Map)

- Karnaugh maps (K-maps) are *graphical* representations of Boolean functions.
- One ***map cell*** corresponds to a row in the truth table.
- Also, one map cell corresponds to a minterm or a maxterm in the Boolean expression
- Each term is identified by a decimal number whose binary representation is identical to the binary interpretation of the input values of the term.

	C'D'	C'D	CD	CD'
A'B'	0	1	3	2
A'B	4	5	7	6
AB	12	13	15	14
AB'	8	9	11	10

K-Map Simplification for Two Variables

- Of course, the Minterm function that we derived from our Truth Table was not in simplest terms.
  - That's what we started with in this example.
- We can, however, reduce our complicated expression to its simplest terms by finding adjacent 1s in the K-map that can be collected into groups that are powers of two.

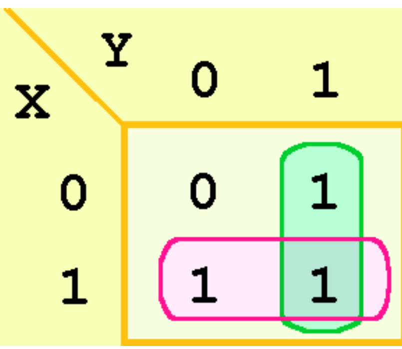


- In our example, we have two such groups.
  - Can you find them?

K-Map Rules

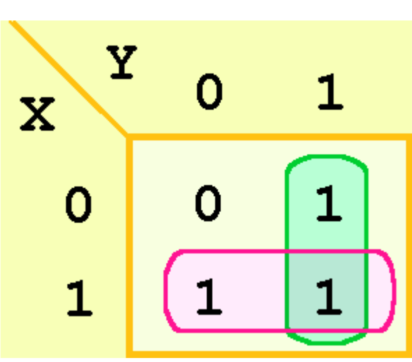
The rules of K-map simplification are:

- Groupings can contain only 1s; no 0s.
- **The number of 1s in a group must be a power of 2 – even if it contains a single 1.**
- Nearby 1s are to be grouped.
- Corner 1s are to be grouped.
- Group that wraps around the sides of a K-map.
- Diagonal groups are not allowed.
- The groups must be made as large as possible.
- Groups can overlap.



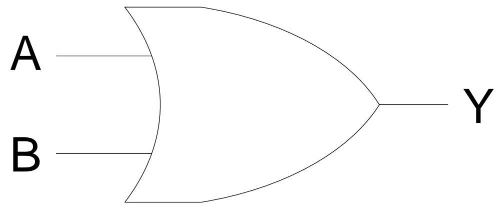
K-Map Rules

- The best way of selecting two groups of 1s form our simple K-map is shown.
- We see that both groups are powers of two and that the groups overlap.

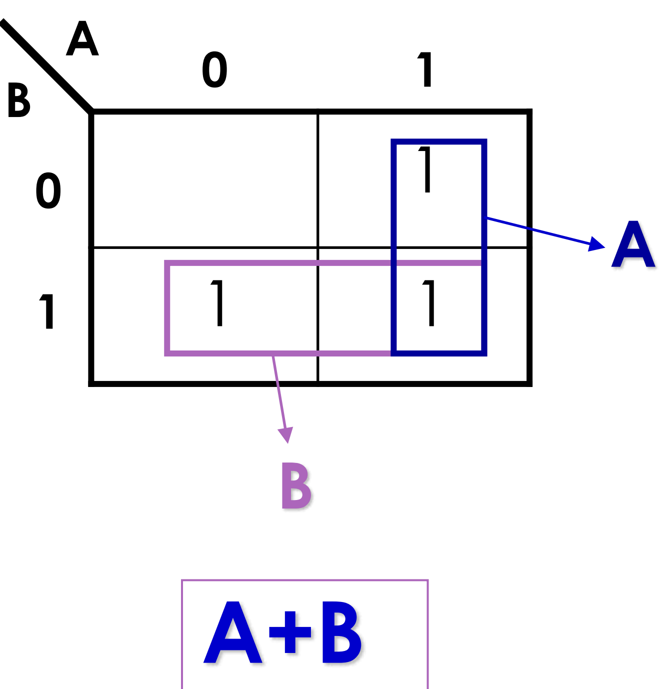


K-Map Simplification for Two Variables

2-variable Karnaugh maps are trivial but can be used to introduce the methods you need to learn. The map for a 2-input OR gate looks like this:

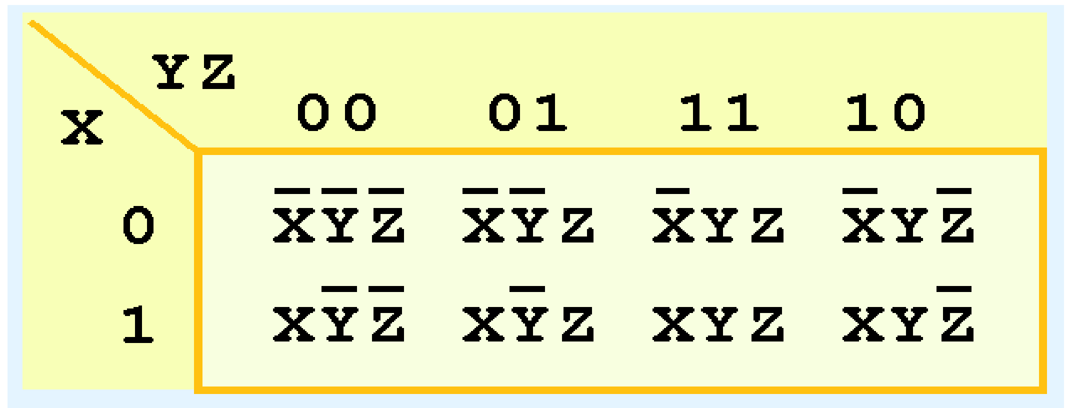


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



K-Map Simplification for Three Variables

- A K-map for three variables is constructed as shown in the diagram below.
- We have placed each Minterm in the cell that will hold its value.
  - Notice that the values for the *yz* combination at the top of the matrix form a pattern that is not a normal binary sequence.



- Consider the function:

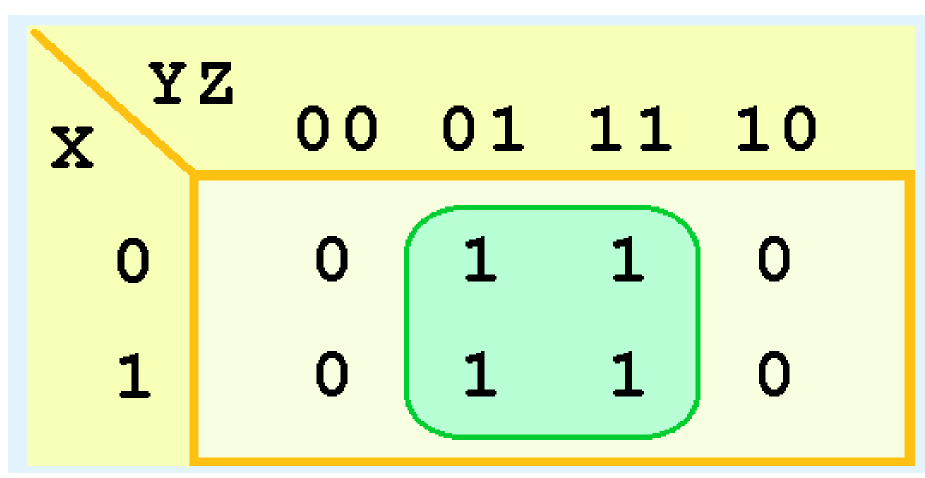
  F (X, Y, Z) = X'Y'Z + X'YZ + XY'Z + XYZ

- Its K-map is given below.
  - What is the largest group of 1s that is a power of 2?

X \ YZ	00	01	11	10
0	0	1	1	0
1	0	1	1	0

- This grouping tells us that changes in the variables *x* and *y* have no influence upon the value of the function: They are irrelevant.
- This means that the function, **F (X, Y, Z) = X'Y'Z + X'YZ + XY'Z + XYZ** reduces to ***F = Z***.

You could verify this reduction with Boolean Algebra



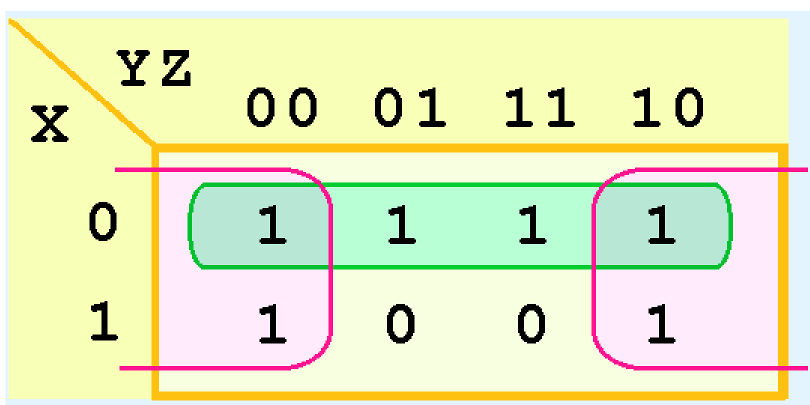
- Now for a more complicated K-map. Consider the function:

$$F(X,Y,Z) = \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}\,\overline{Y}Z + \overline{X}YZ + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XY\overline{Z}$$

- Its K-map is shown below. There are (only) two groupings of 1s.
  - Can you find them?

X \ YZ	00	01	11	10
0	1	1	1	1
1	1	0	0	1

- In this K-map, we see an example of a group that wraps around the sides of a K-map.



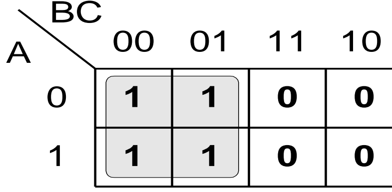
$f = \sum(0,4) = \bar{B}\,\bar{C}$

A \ BC	00	01	11	10
0	1	0	0	0
1	1	0	0	0

$f = \sum(4,5) = A\,\bar{B}$

A \ BC	00	01	11	10
0	0	0	0	0
1	1	1	0	0

$f = \sum(0,1,4,5) = \bar{B}$



A \ BC	00	01	11	10
0	1	1	0	0
1	1	1	0	0

$f = \sum(0,1,2,3) = \bar{A}$

A \ BC	00	01	11	10
0	1	1	1	1
1	0	0	0	0

$f = \sum(1,3) = A'C$

~~$f = \sum(0,4) = \bar{A}\,C$~~

A \ BC	00	01	11	10
0	0	1	1	0
1	0	0	0	0

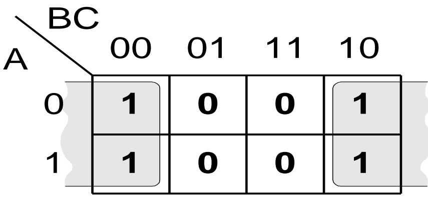
$f = \sum(4,6) = A\,\bar{C}$

A \ BC	00	01	11	10
0	0	0	0	0
1	1	0	0	1

$f = \sum(0,2) = \bar{A}\,\bar{C}$

A \ BC	00	01	11	10
0	1	0	0	1
1	0	0	0	0

$f = \sum(0,2,4,6) = \bar{C}$



A \ BC	00	01	11	10
0	1	0	0	1
1	1	0	0	1

K-Map Simplification for Four Variables

- The K-map can be extended to accommodate the 16 Minterms that are produced by a four-input function.
- This is the format for a 16-minterm K-map.

WX \ YZ	00	01	11	10
00	$\overline{W}\overline{X}\overline{Y}\overline{Z}$	$\overline{W}\overline{X}\overline{Y}Z$	$\overline{W}\overline{X}YZ$	$\overline{W}\overline{X}Y\overline{Z}$
01	$\overline{W}X\overline{Y}\overline{Z}$	$\overline{W}X\overline{Y}Z$	$\overline{W}XYZ$	$\overline{W}XY\overline{Z}$
11	$WX\overline{Y}\overline{Z}$	$WX\overline{Y}Z$	$WXYZ$	$WXY\overline{Z}$
10	$W\overline{X}\overline{Y}\overline{Z}$	$W\overline{X}\overline{Y}Z$	$W\overline{X}YZ$	$W\overline{X}Y\overline{Z}$

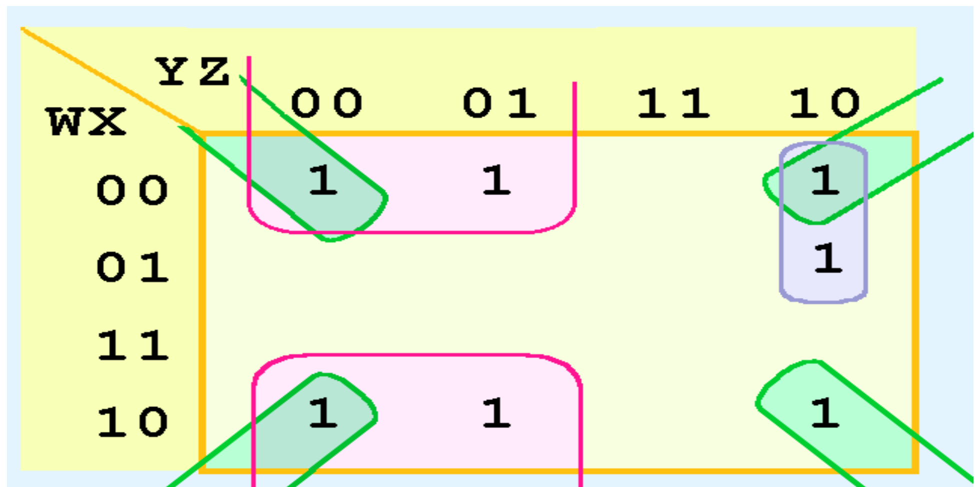
- We have populated the K-map shown below with the nonzero minterms from the function:

$$F(W,X,Y,Z) = \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z + W\overline{X}Y\overline{Z}$$

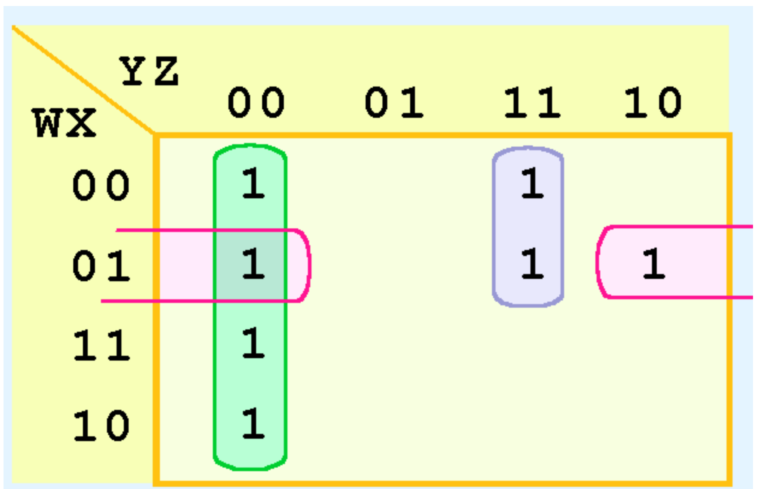
  - Can you identify (only) three groups in this K-map?

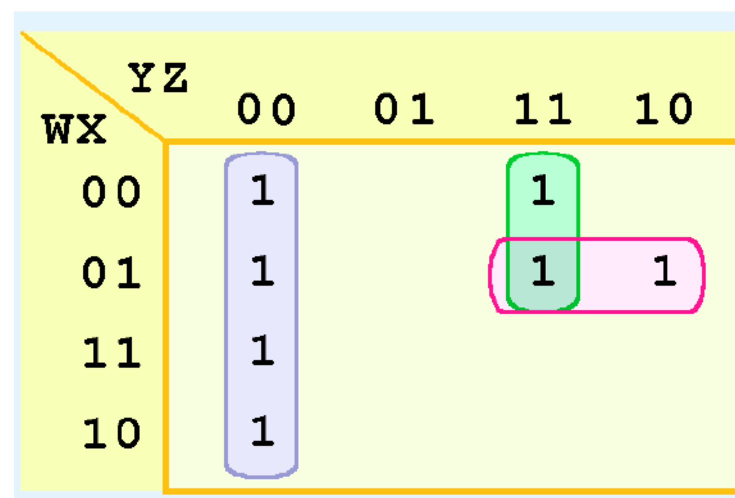
WX \ YZ	00	01	11	10
00	1	1		1
01				1
11				
10	1	1		1

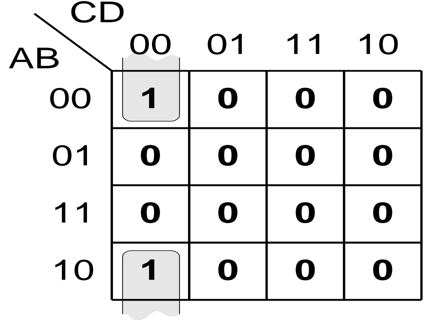
- Our three groups consist of:
  - A purple group entirely within the K-map at the right.
  - A pink group that wraps the top and bottom.
  - A green group that spans the corners.
- Thus we have three terms in our final function:



- It is possible to have a choice as to how to pick groups within a K-map, while keeping the groups as large as possible.
- The (different) functions that result from the groupings below are logically equivalent.







AB \ CD	00	01	11	10
00	1	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	0	0	0

$f = \sum(0,8) = \overline{B} \bullet \overline{C} \bullet \overline{D}$

AB \ CD	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	0	1	0	0
10	0	0	0	0

$f = \sum(5,13) = B \bullet \overline{C} \bullet D$

AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	1	1	0
10	0	0	0	0

$f = \sum(13,15) = A \bullet B \bullet D$

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	0	0	1
11	0	0	0	0
10	0	0	0	0

$f = \sum(4,6) = \overline{A} \bullet B \bullet \overline{D}$

AB \ CD	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	0	0

$f = \sum(2,3,6,7) = \overline{A} \bullet C$

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	0	0	1
11	1	0	0	1
10	0	0	0	0

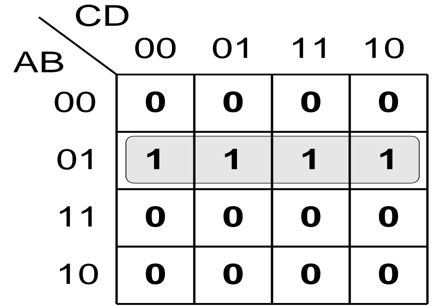
$f = \sum(4,6,12,14) = B \bullet \overline{D}$

AB \ CD	00	01	11	10
00	0	0	1	1
01	0	0	0	0
11	0	0	0	0
10	0	0	1	1

$f = \sum(2,3,10,11) = \overline{B} \bullet C$

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	0	0
10	1	0	0	1

$f = \sum(0,2,8,10) = \overline{B} \bullet \overline{D}$



AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

$f = \sum(4,5,6,7) = \overline{A} \bullet B$

AB \ CD	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	0	1	0
10	0	0	1	0

$f = \sum(3,7,11,15) = C \bullet D$

AB \ CD	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

$f = \sum(0,3,5,6,9,10,12,15)$

$f = A \otimes B \otimes C \otimes D$

AB \ CD	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$f = \sum(1,2,4,7,8,11,13,14)$

$f = A \oplus B \oplus C \oplus D$

AB \ CD	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$f = \sum(1,3,5,7,9,11,13,15)$

$f = D$

AB \ CD	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	1	0	0	1
10	1	0	0	1

$f = \sum(0,2,4,6,8,10,12,14)$

$f = \overline{D}$

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	0	0	0	0

$f = \sum(4,5,6,7,12,13,14,15)$

$f = B$

AB \ CD	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	0	0
10	1	1	1	1

$f = \sum(0,1,2,3,8,9,10,11)$

$f = \overline{B}$

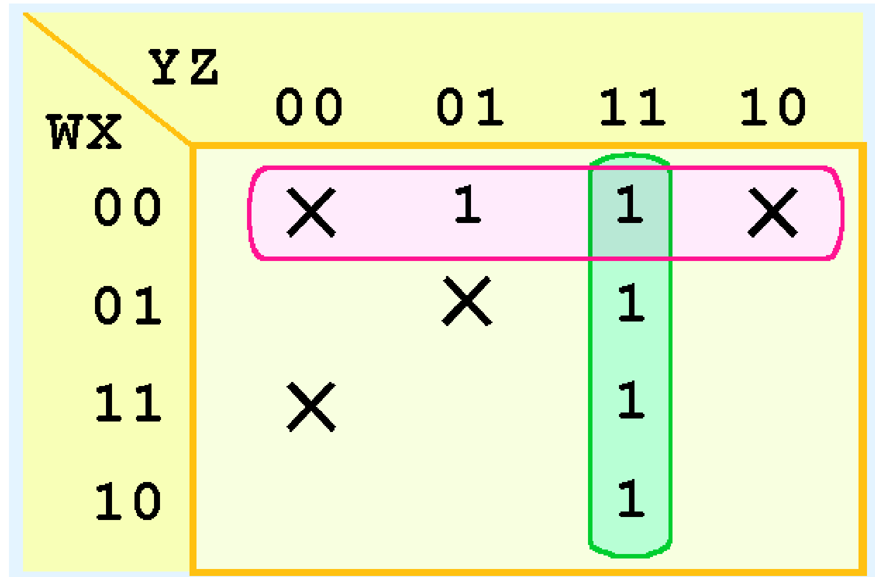
Don’t Care Conditions

- Real circuits don’t always need to have an output defined for every possible input.
  - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display 2^7 patterns but all patterns are not used.
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a *don’t care* condition.
- They are very helpful to us in K-map circuit simplification.

- In a K-map, a don't care condition is identified by an *X* in the cell of the minterm(s) for the don't care inputs, as shown below.
- In performing the simplification, we are free to include or ignore the *X*'s when creating our groups.

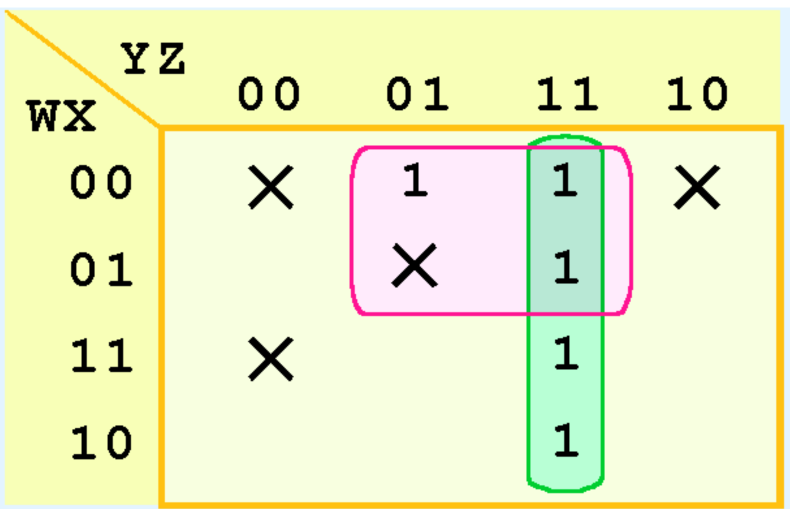
WX \ YZ	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

- In one grouping in the K-map below, we have the function:
- **F = W'X' + YZ**



- A different grouping gives us the function:

$$F(W,X,Y,Z) = \overline{W}Z + YZ$$



- The truth table of:

$$F(W, X, Y, Z) = W'X' + YZ$$

differs from the truth table of:

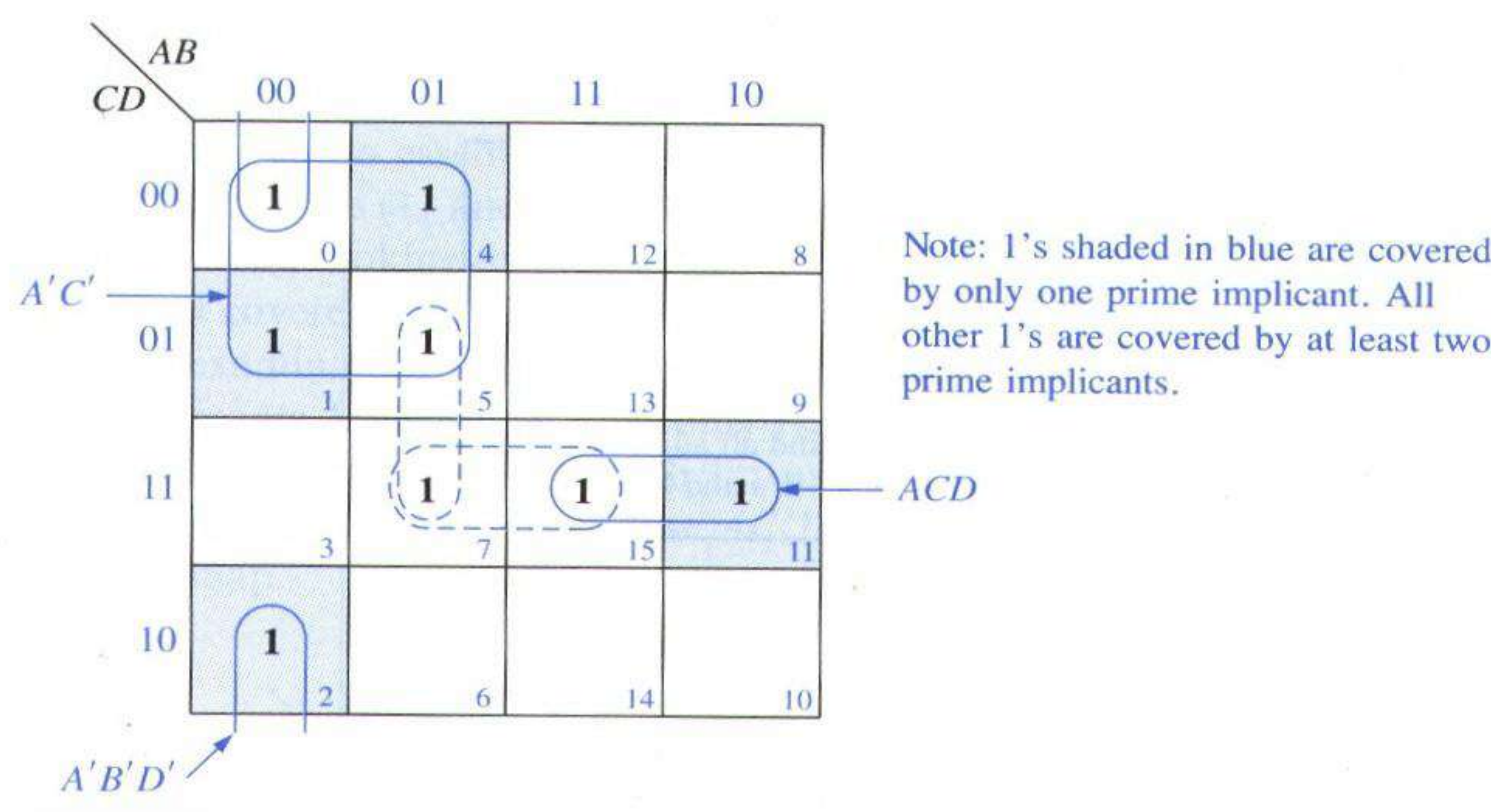
$$F(W,X,Y,Z) = \overline{W}Z + YZ$$

- However, the values for which they differ, are the inputs for which we have don't care conditions.

WX \ YZ	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

WX \ YZ	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

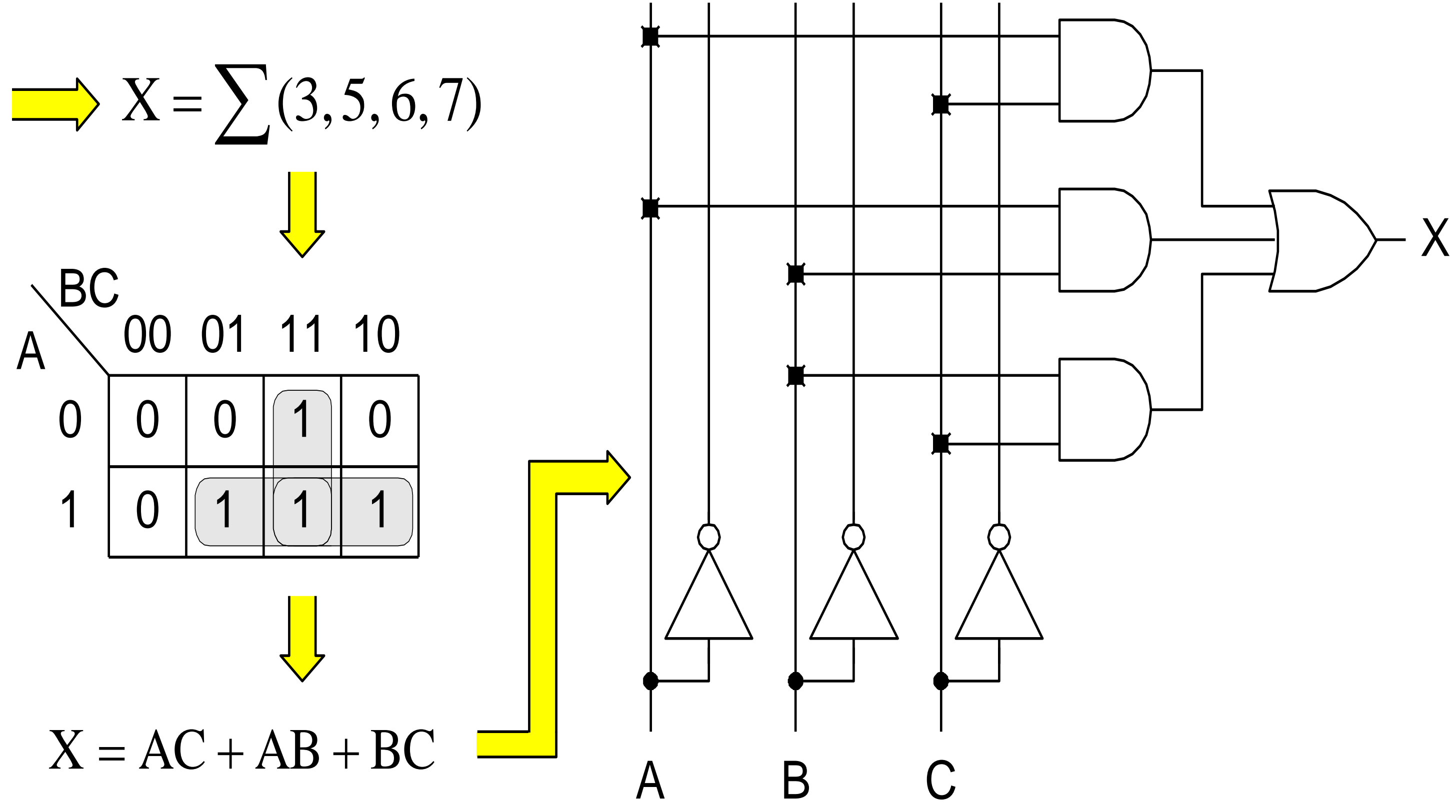
Redundancy



Design of combinational digital circuits

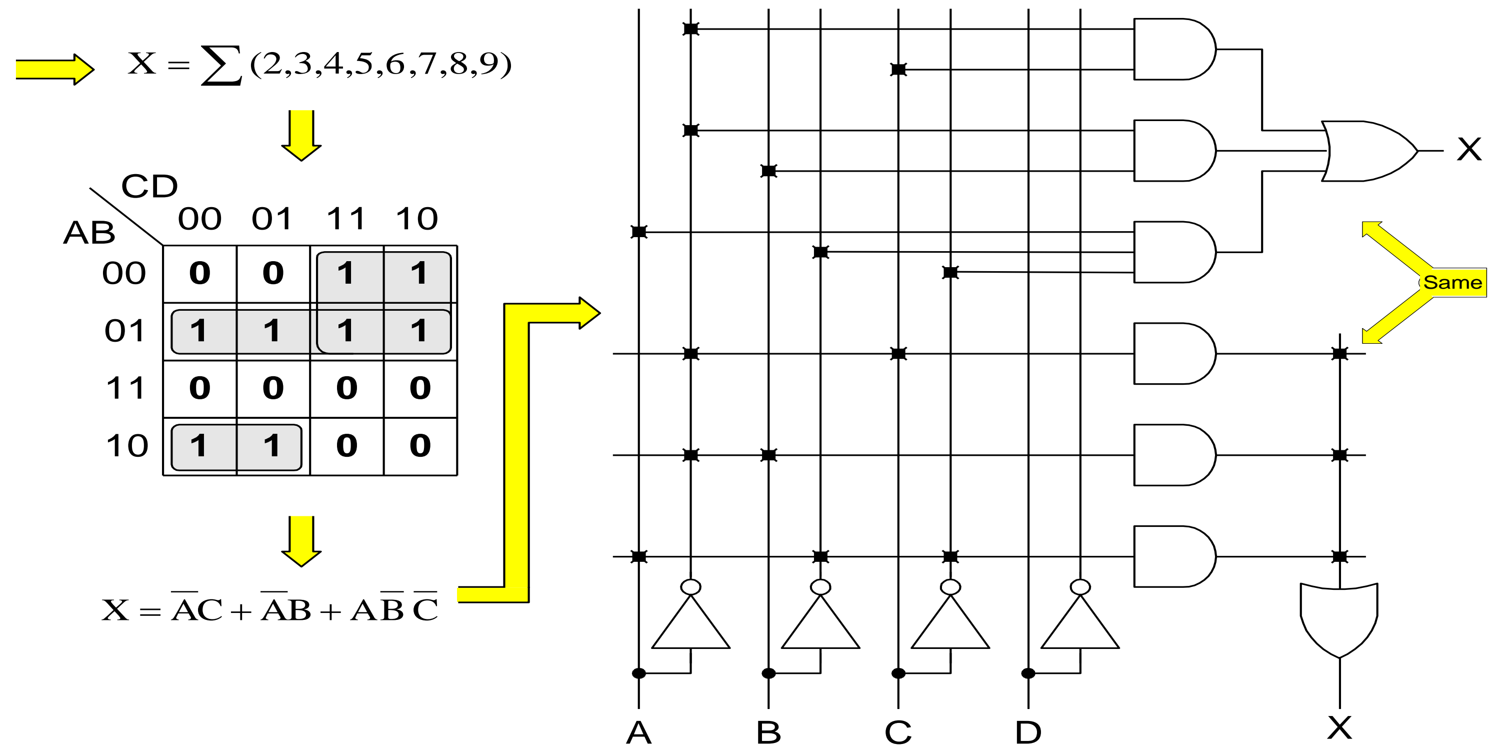
- Steps to design a combinational digital circuit:
  - From the problem statement derive the truth table
  - From the truth table derive the unsimplified logic expression
  - Simplify the logic expression
  - From the simplified expression draw the logic circuit
- Example: Design a 3-input (A,B,C) digital circuit that will give at its output (X) a logic 1 only if the binary number formed at the input has more ones than zeros.

	Inputs			Output
	A	B	C	X
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



- Example: Design a 4-input (A,B,C,D) digital circuit that will give at its output (X) a logic 1 only if the binary number formed at the input is between 2 and 9 (including).

	Inputs				Output
	A	B	C	D	X
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0



Conclusion

- K-maps provide an easy graphical method of simplifying Boolean expressions.
- A K-map is a matrix consisting of the outputs of the minterms of a Boolean function.
- In this section, we have discussed 2- 3- and 4-input K-maps. This method can be extended to any number of inputs through the use of multiple tables.

Recapping the rules of K-map simplification:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.
- Groups can overlap and wrap around the sides of the K-map.
- Use don’t care conditions when you can.
- Redundancy must be reduced

Lecture of Module 3

Combinational Circuits

Overview

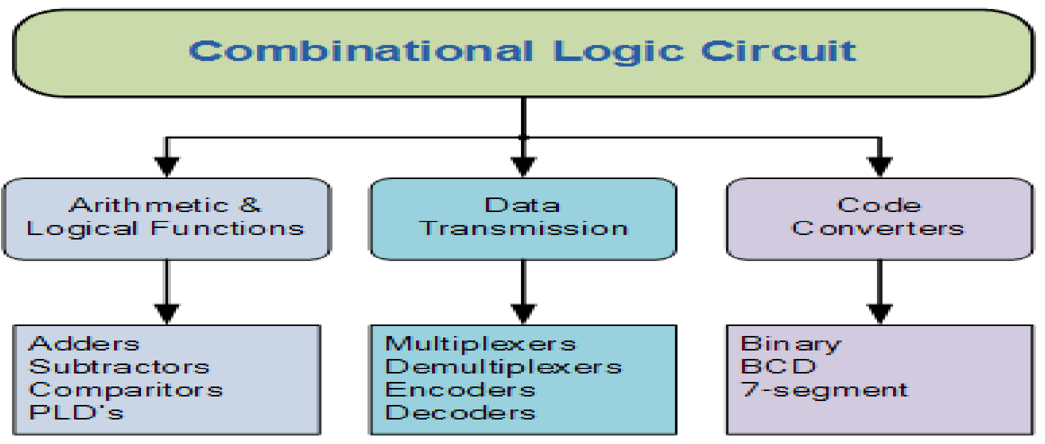
- **Introduction**
- **Half Adder**
- **Full Adder**
- **Half Subtractor**
- **Full Subtractor**
- **Ripple/Parallel Adder**
- **Adder-Subtractor**
- **Look-ahead carry Adder**

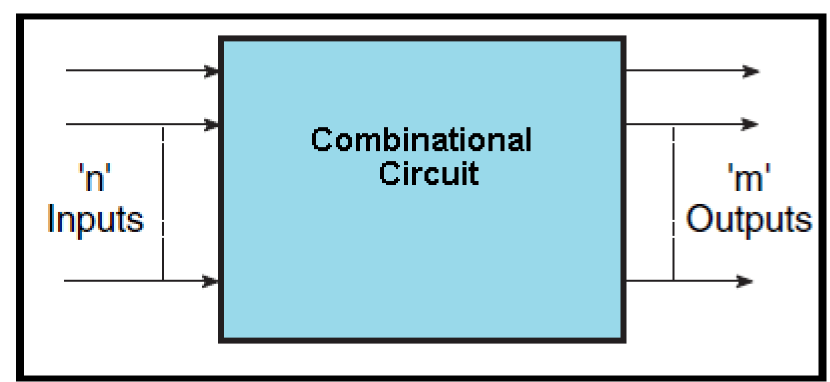
Introduction

The outputs of **Combinational Logic Circuits** are only determined by the logical function of their current input state(s), logic “0” or(and) logic “1”, at any given instant.

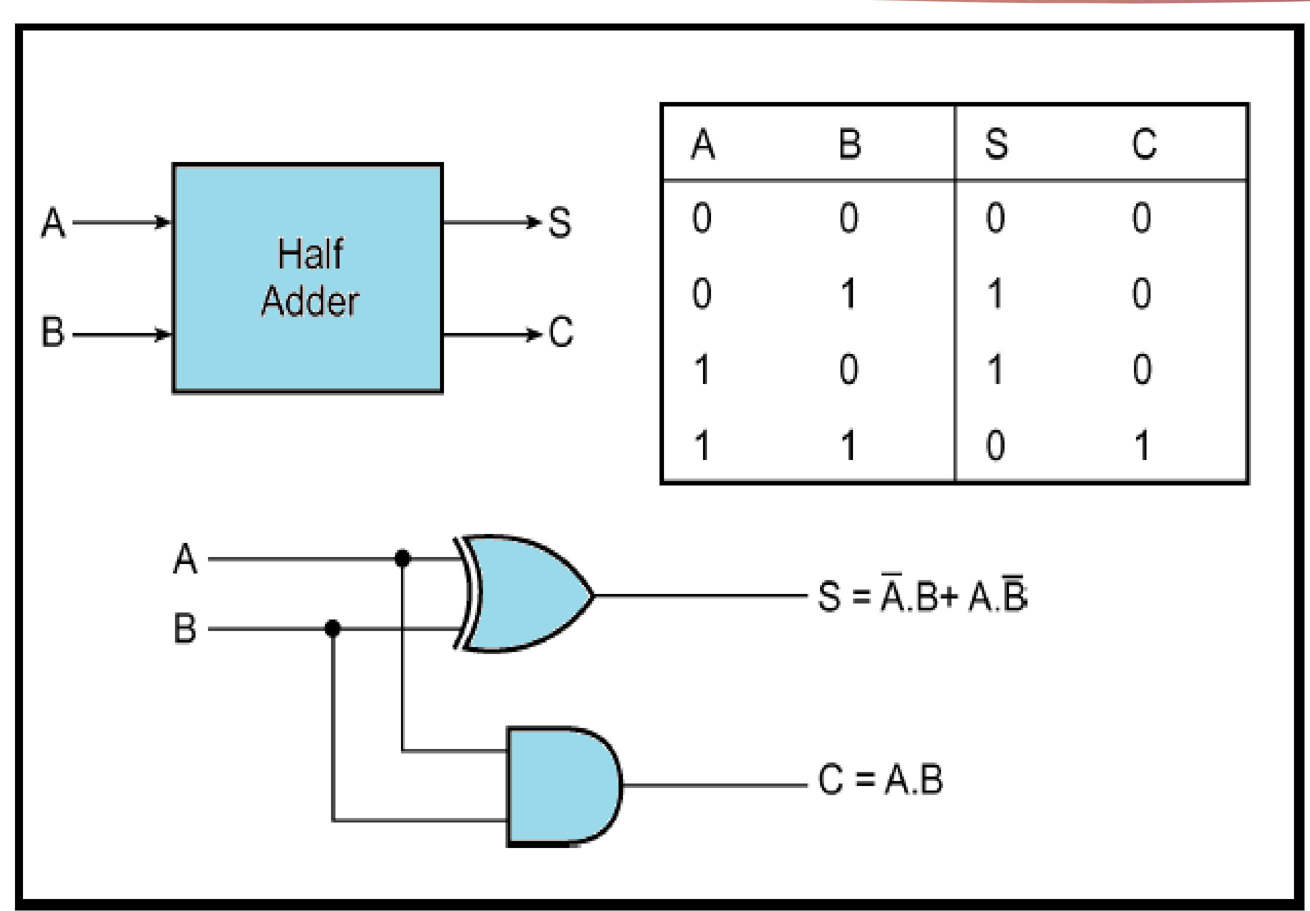
Combinational logic circuits give us many useful devices.

One of the simplest is the *half adder*, which finds the sum of two bits.

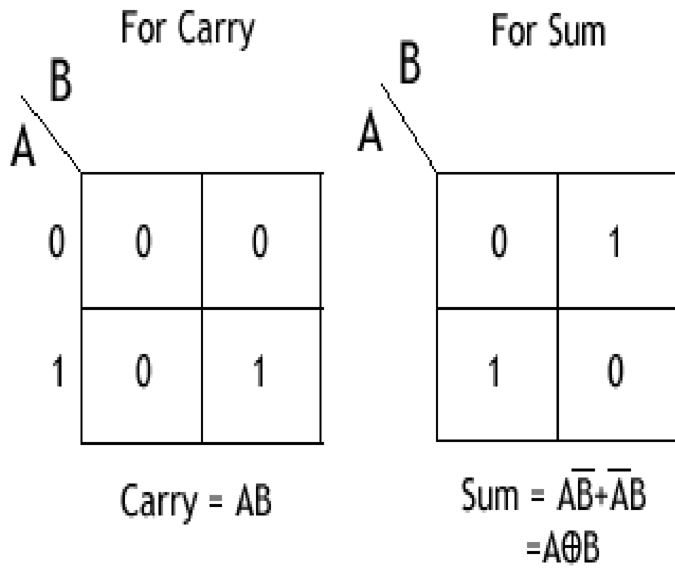


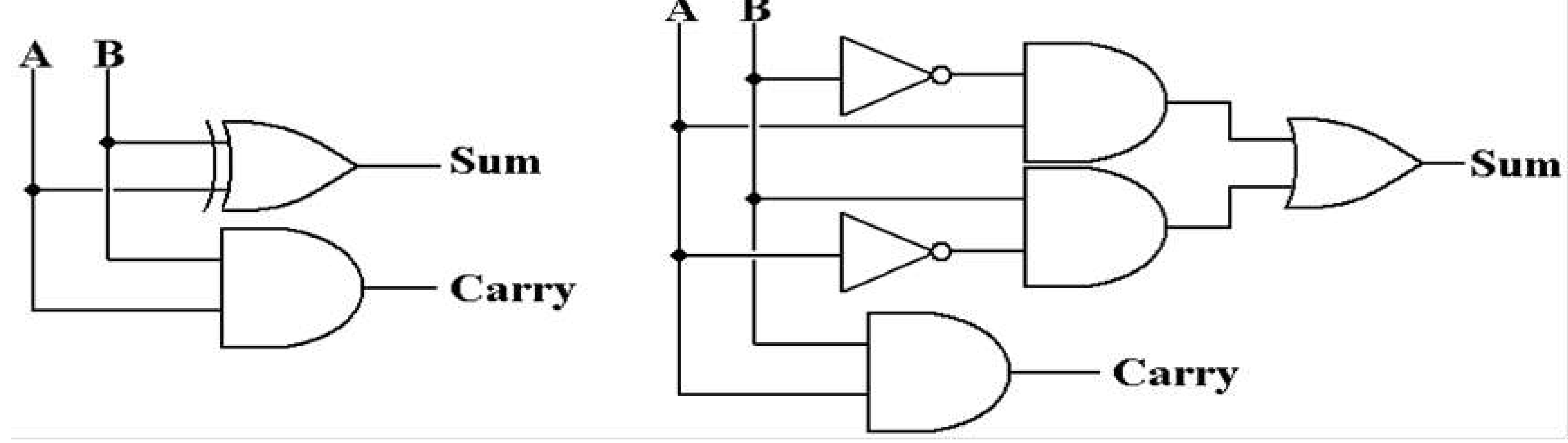


Half Adder

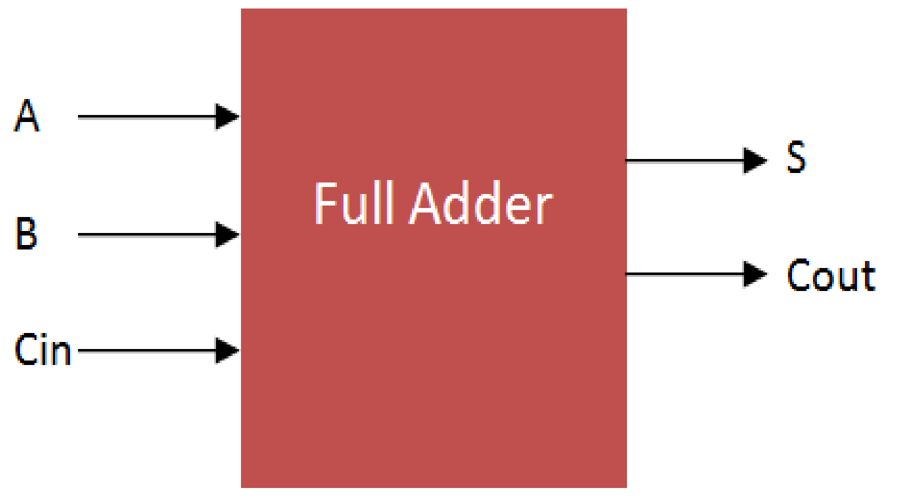


A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

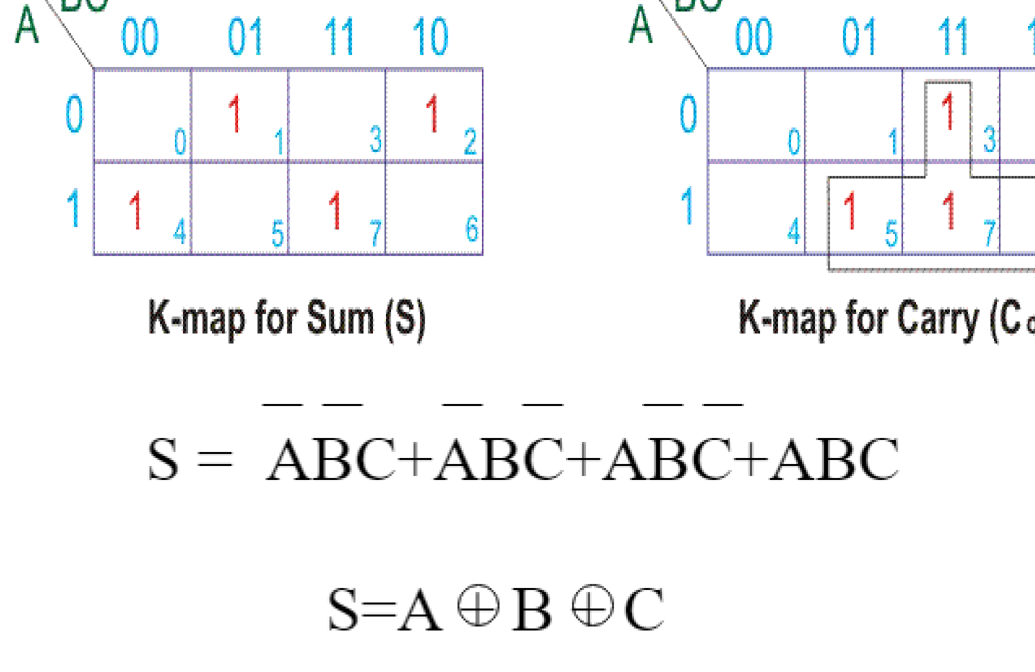



A
B
Sum
Carry
A
B
Sum
Carry

Full Adder



Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$S = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

$$S = A \oplus B \oplus C$$

$$C = AB + BC + CA$$

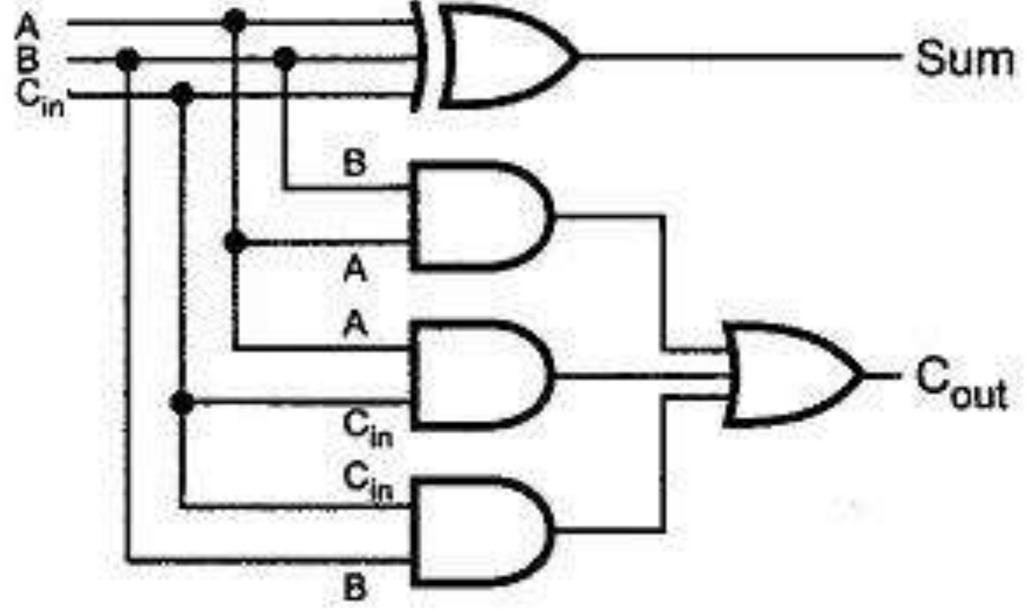
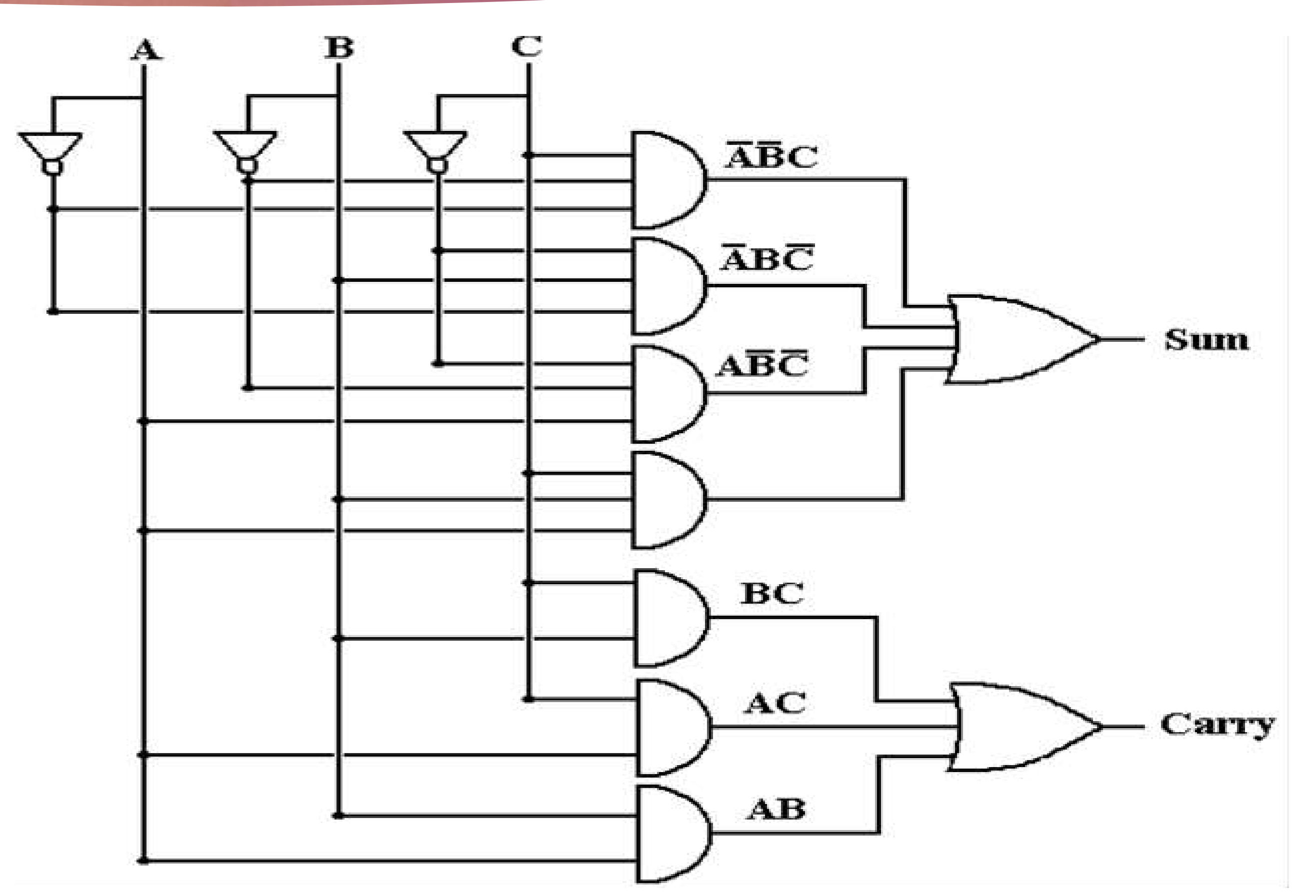


Fig. 3.17 Implementation of full-adder

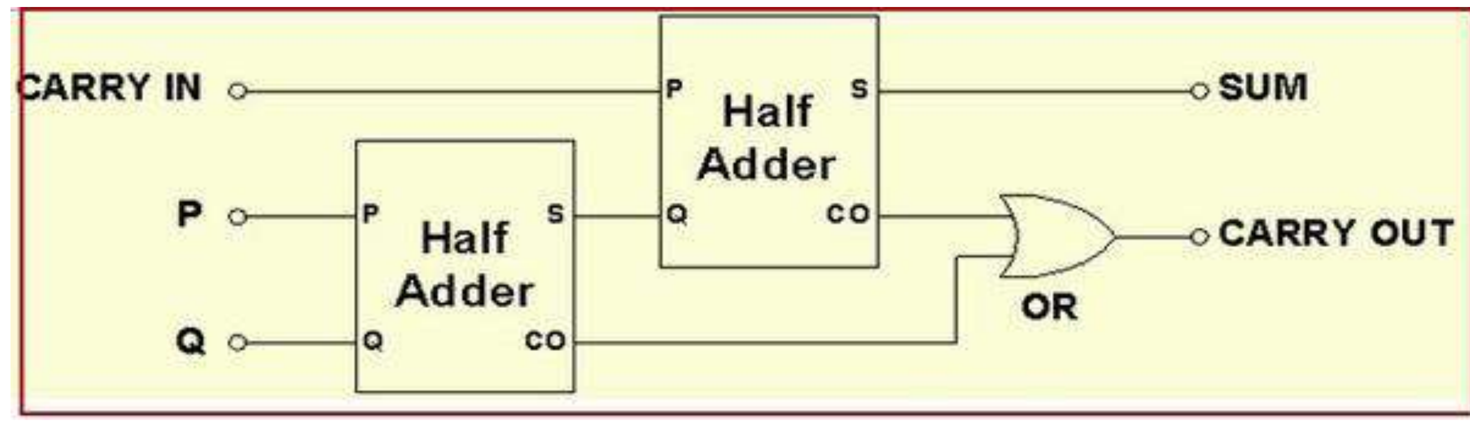
$$S = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

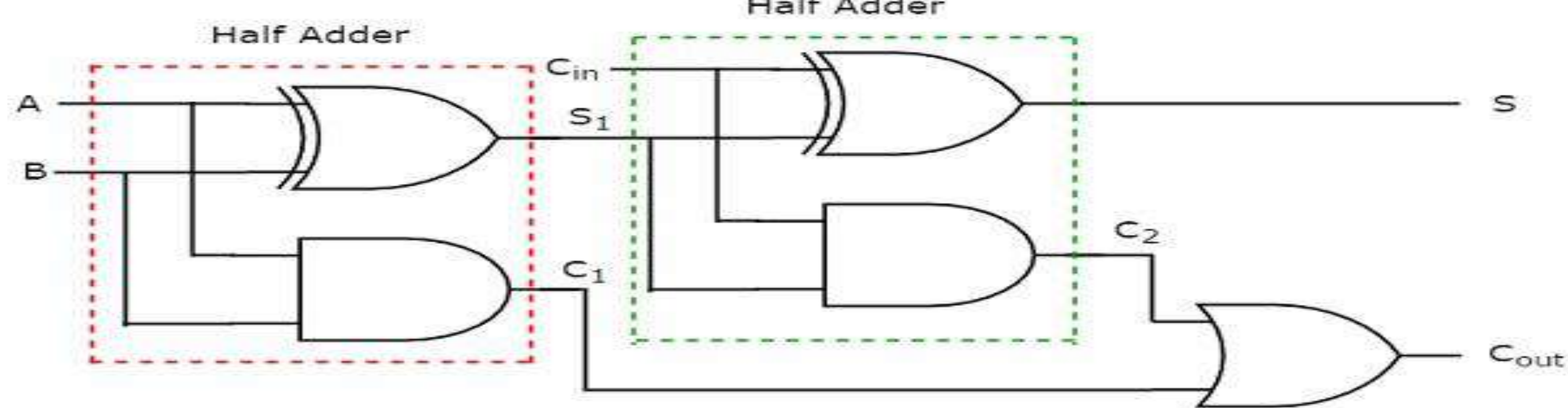
$$S = A \oplus B \oplus C$$

$$C = AB + BC + CA$$

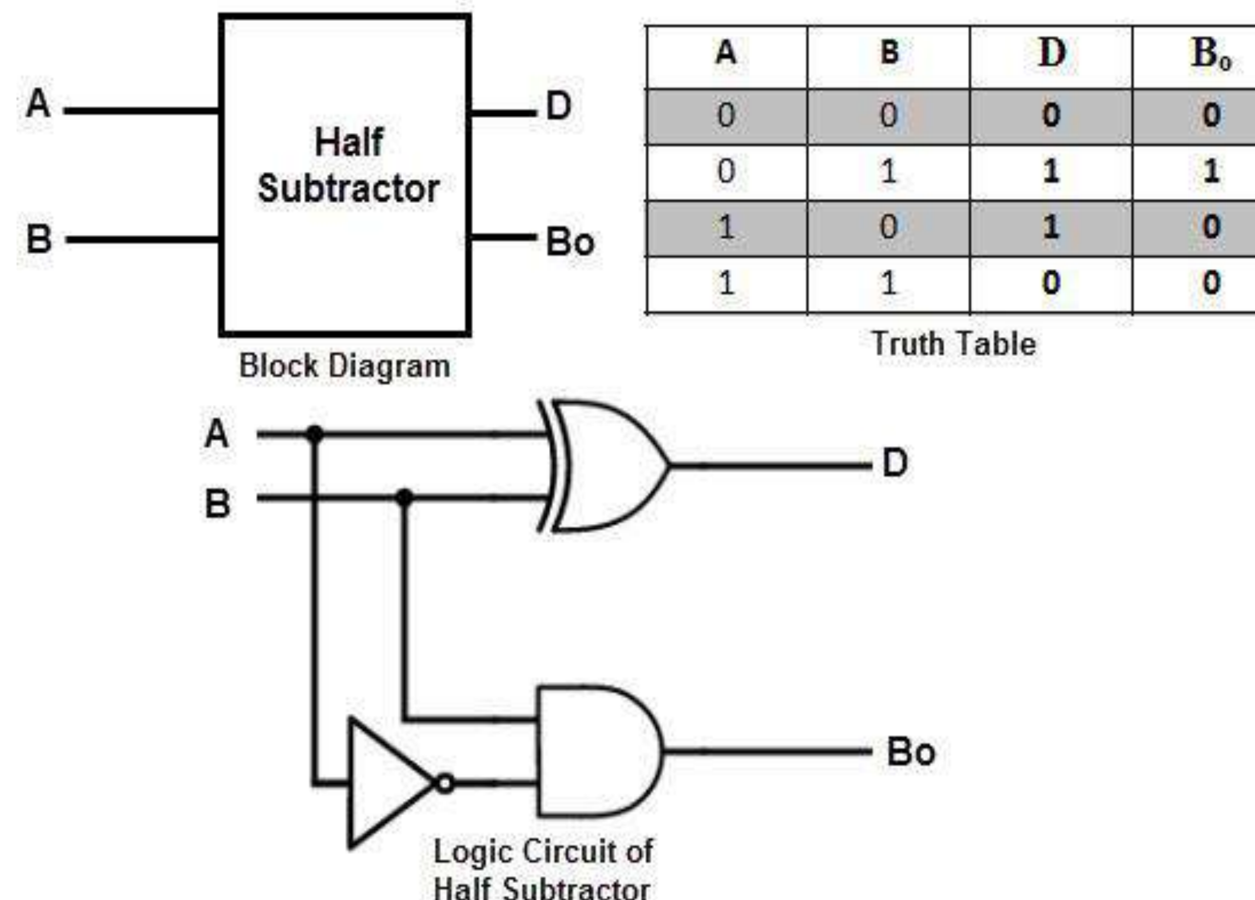


Full Adder using Half Adders



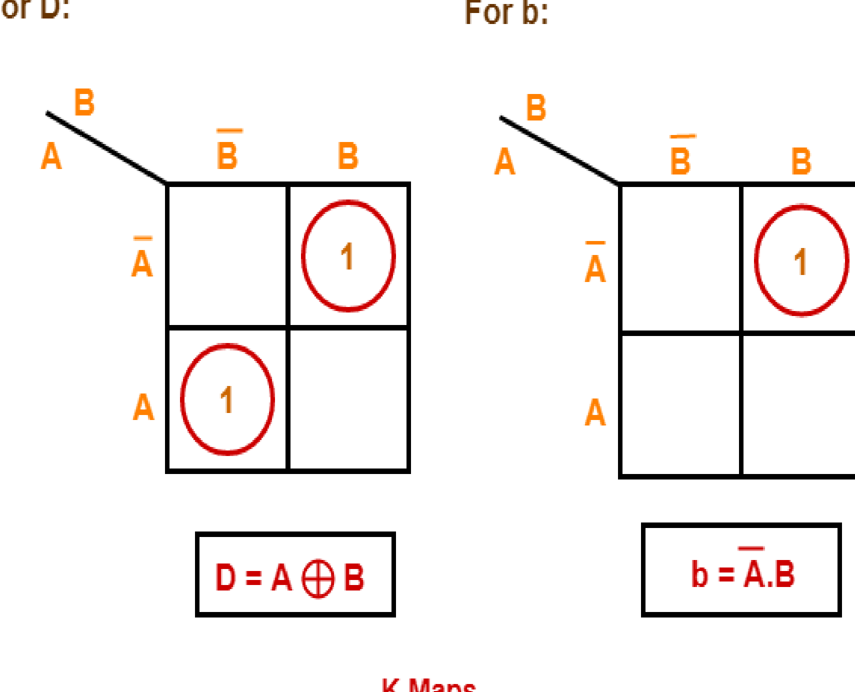


Half Subtractor



A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

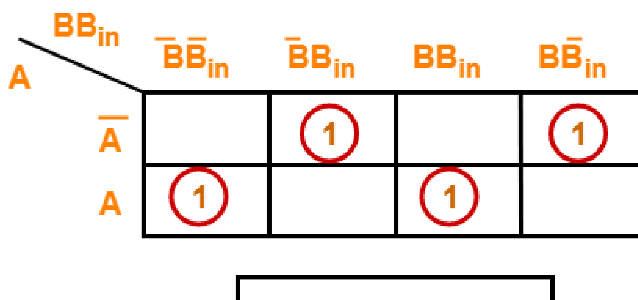
Truth Table



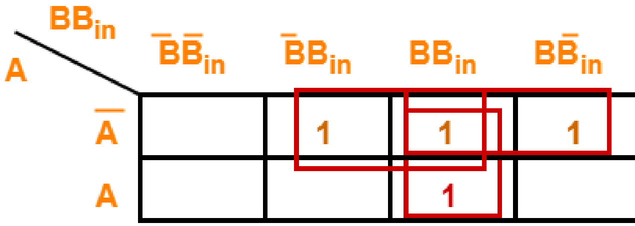
Full Subtractor

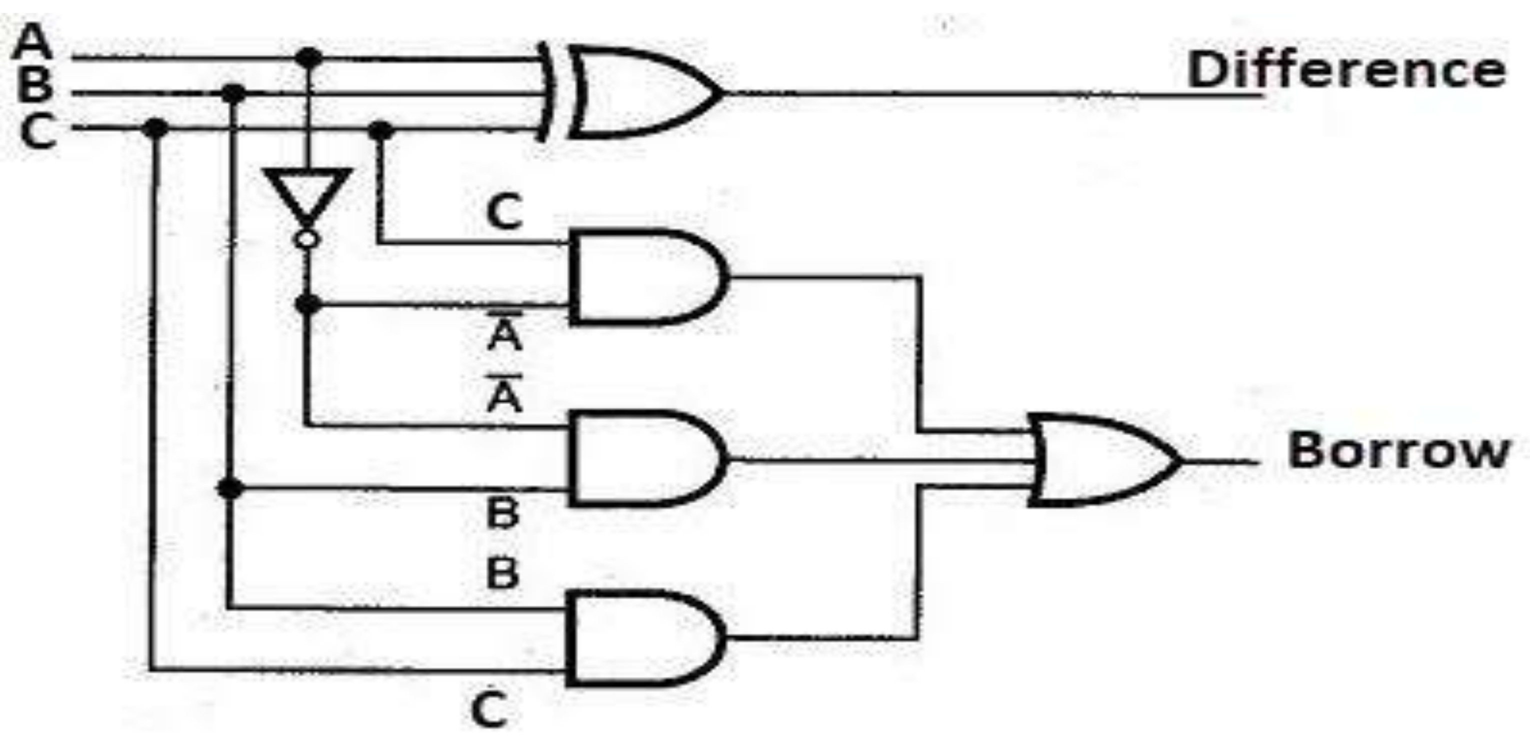
INPUT			OUTPUT	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

For D:

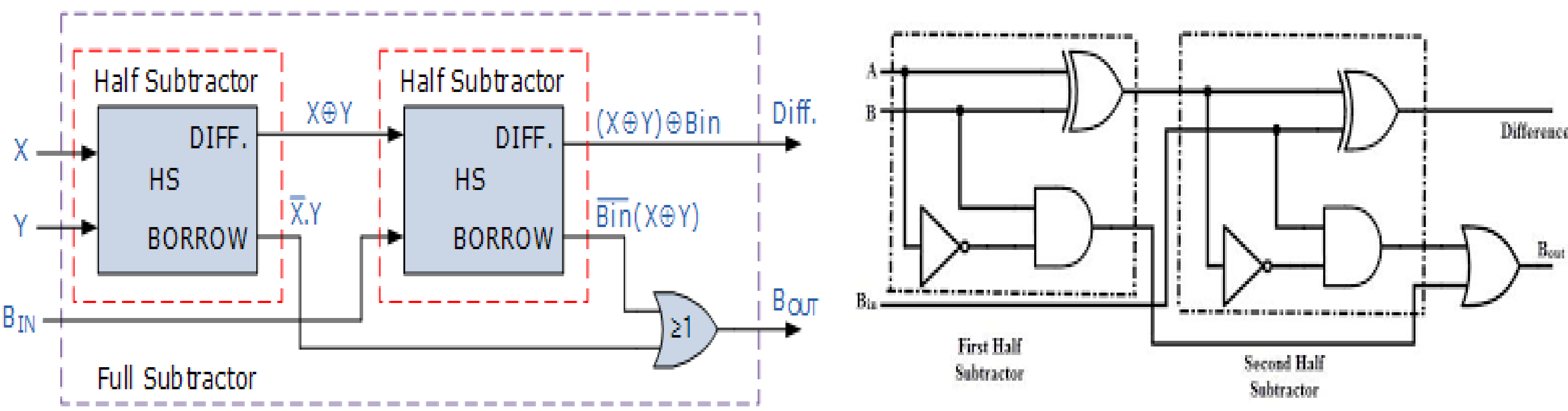


For B_{in}:



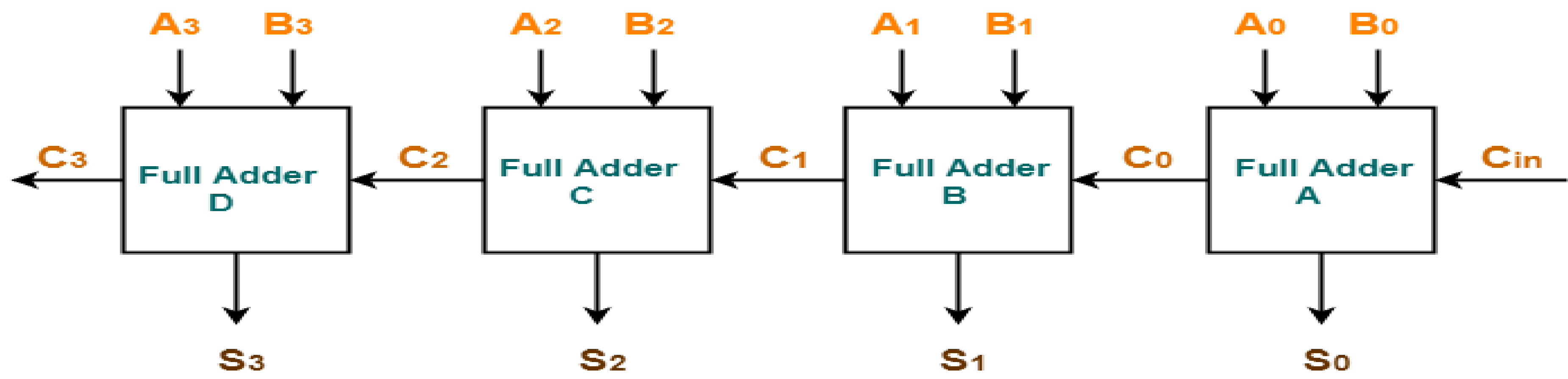

A
B
C
Difference
C
$\overline{A}$
$\overline{A}$
B
B
C
Borrow

Full Subtractor using Half Subtractor



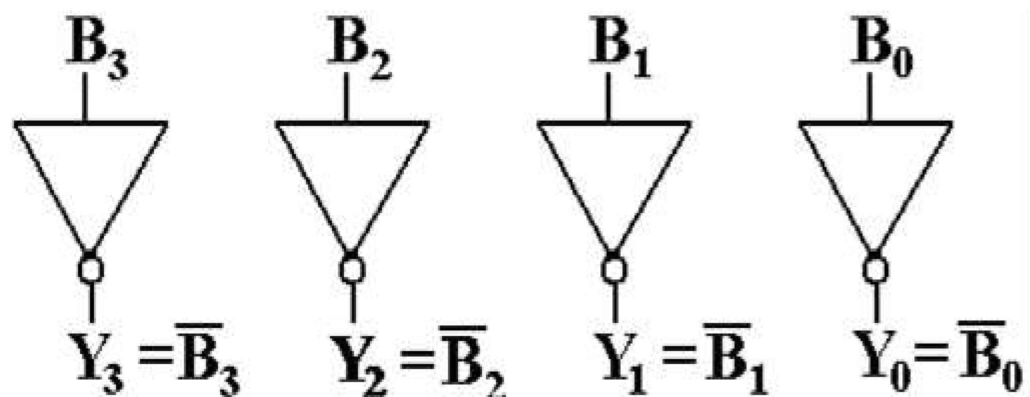
Ripple/ Parallel Adder

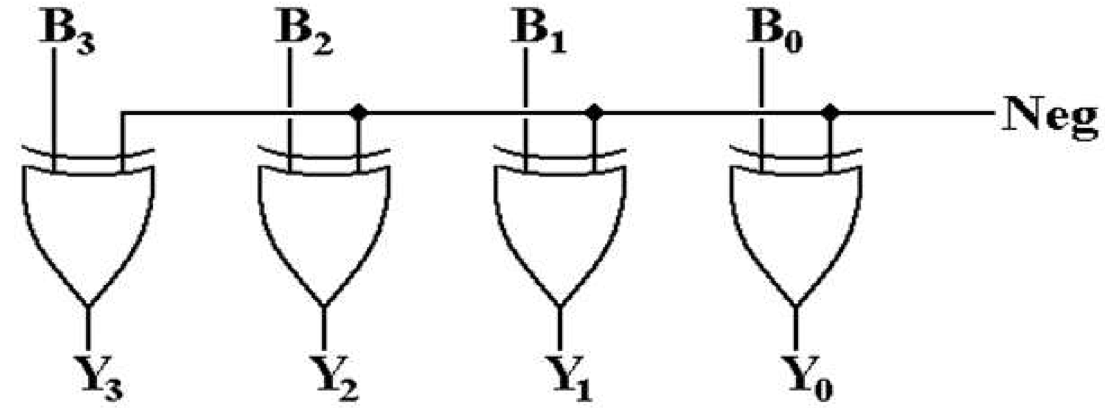
- Just as we combined half adders to make a full adder, full adders can connected in series.
- The carry bit "ripples" from one adder to the next; hence, this configuration is called a *ripple-carry adder.*



4-bit Ripple Carry Adder

One's Complement Circuit





In order to make an adder/subtractor, it is necessary to use a gate that can either pass the value through or generate its one's–complement.

The exclusive OR gate, XOR, is exactly what we need.

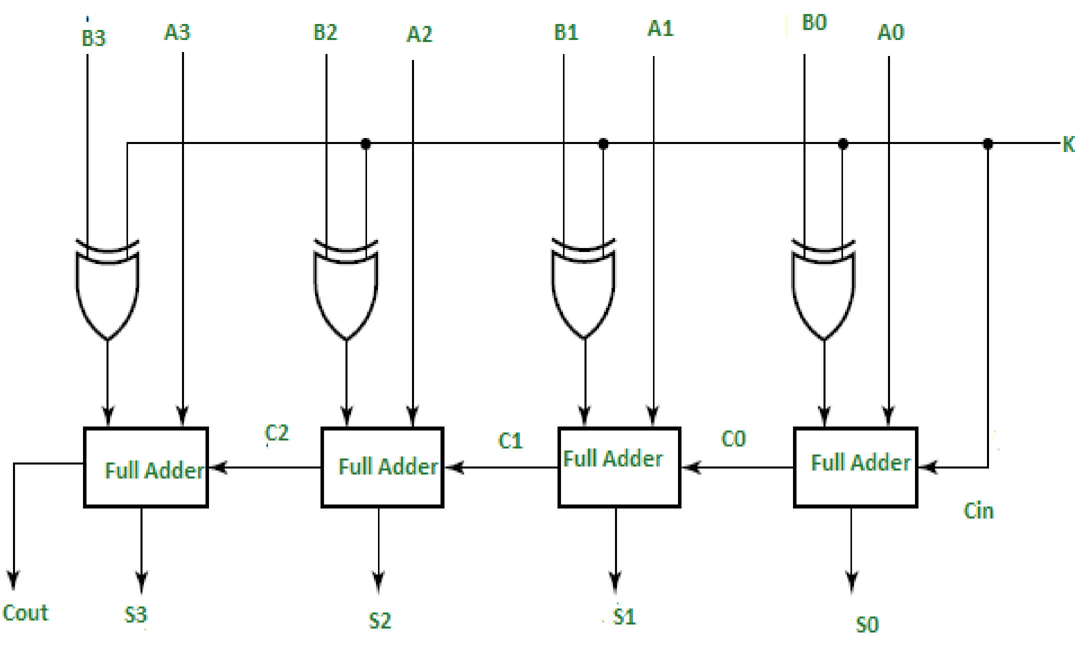
This is controlled by a binary signal: **Neg**.

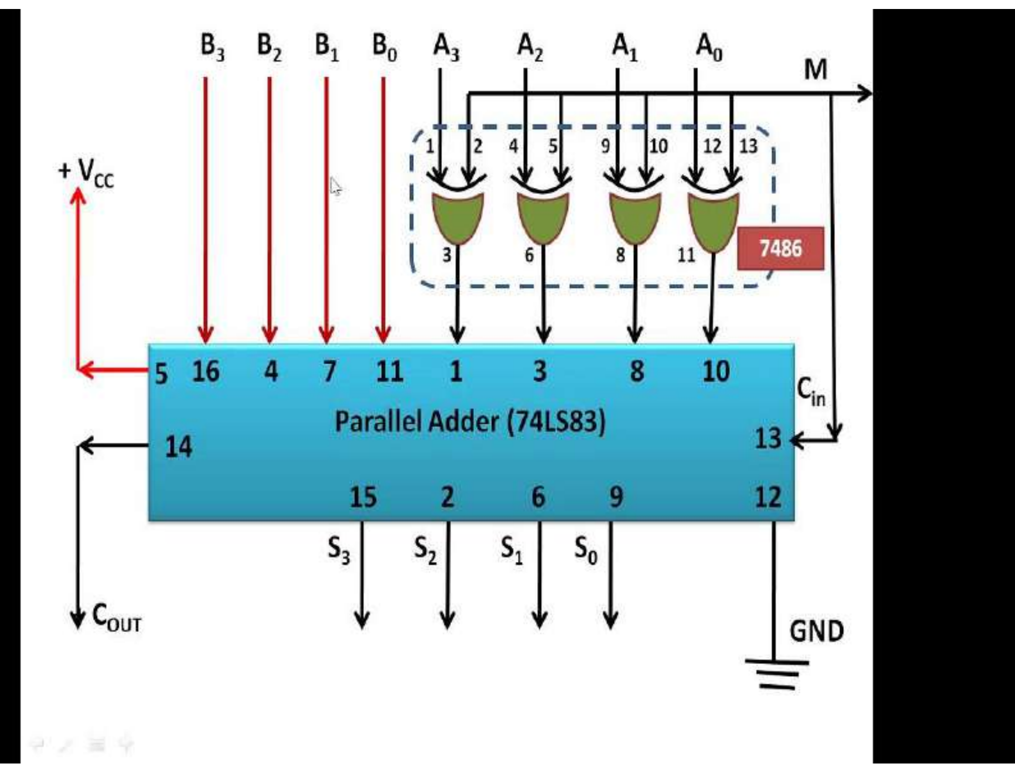
Let B = 1011.

If Neg = 0, then Y = 1011.

If Neg = 1, then Y = 0100.

Adder-Subtractor





- In any combinational circuit, the signal must propagate through the gates before the correct output is available in the output terminal.
- The total propagation time equal to the propagation delay of a typical gate times multiplied with the gate levels in the circuit.
- The propagation delay time in a parallel adder is the time it takes the carry to propagate through the full adder.
- In each full adder the carry out from the carry in passes through two gate levels.
- For n-bit parallel adder the total gate delay will be 2n.

- So, the carry propagation time is a limiting factor on the speed with which two numbers are added in parallel.
- To avoid that another adder is widely used which employs the principle of Look-ahead carry.
- The adder designed using the principle of Look-ahead carry is called as Look-ahead carry adder or Carry look-ahead adder.

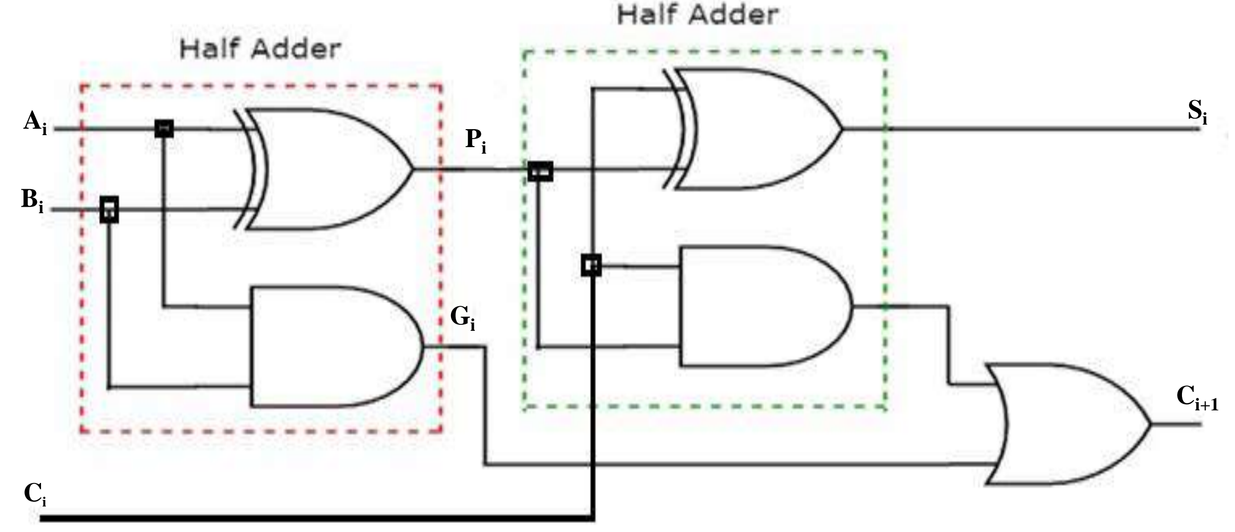
Look-Ahead Carry Adder

$P_i = A_i \oplus B_i$
$G_i = A_i B_i$

The output Sum and Carry can be expressed as:

$S_i = P_i \oplus Ci$
$C_{i+1} = G_i + P_i C_i$

G_i is called as carry generator and P_i is called as carry propagator.



These equations show that a carry signal will be generated in two cases:

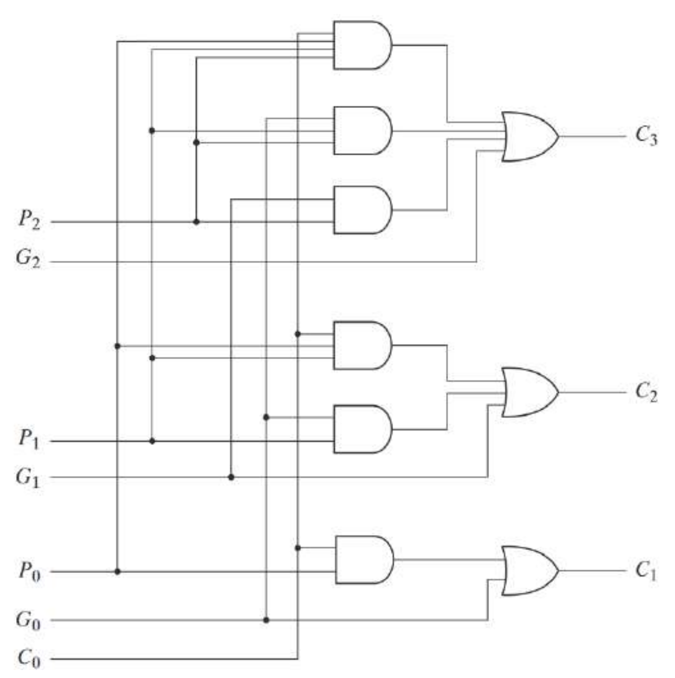
1) if both bits A_i and B_i are **1**
2) if either A_i or B_i is **1** and the carry-in C_i is **1**.

Let's apply these equations for a 4-bit adder:

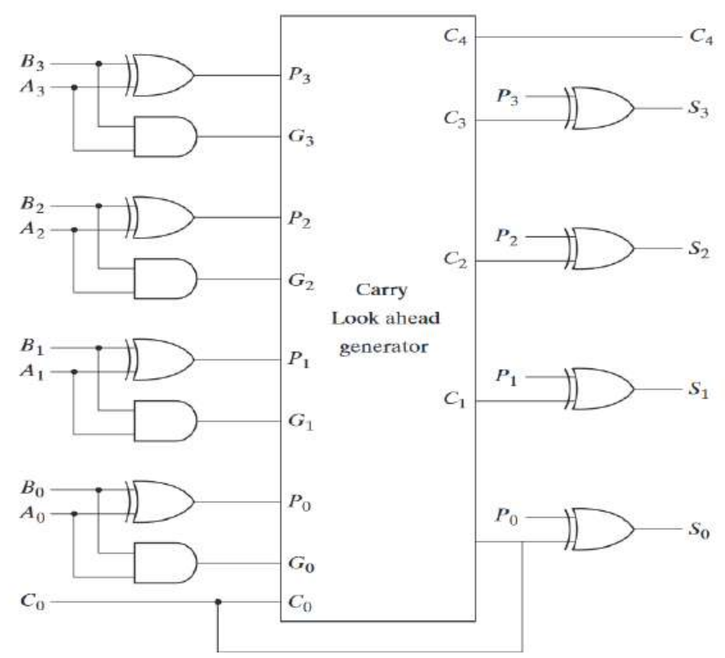
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- These expressions show that C_2, C_3 and C_4 do not depend on its previous carry-in.
- Therefore C_4 does not need to wait for C_3 to propagate.
- As soon as C_0 is computed, C_4 can reach steady state.
- The same is also true for C_2 and C_3.
- The general expression is

  $$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots\ldots. P_iP_{i-1} \ldots. P_2P_1G_0 + P_iP_{i-1} \ldots. P_1P_0C_0.$$
- This is a two level circuit



Carry Look-Ahead Generator



Full Adder with Look-Ahead Carry

Total 4 gate delay: One gate delay for P_i and G_i generator, two gate delay for Carry generator and one gate delay for Sum generator.

Advantages:

- CLA Adders generate the carry-in for each full adder simultaneously, by using simplified equations involving P_i, G_i, and C_{in}.
- This system reduces the propagation delay.
- This is because the output carry at any stage is dependent only on the first Carry signal given at the input.
- It is the fastest adder when compared to other addition mechanisms.

Disadvantages:

- The carry look-ahead adder circuit gets more complicated as the number of variables increase.
- The circuit for a carry look-ahead adder is expensive as it involves more hardware.
- As the number of variables increases, the circuit implements more hardware.
- Thus, when the carry look-ahead adder is implemented as an IC, the area is bound to increase.

Ripple Carry Adder vs. Carry Look Ahead Adder

Ripple Carry Adder	Carry Look Ahead Adder
The Carry bit passes through a long logic chain through the entire circuit.	The Carry bit enters in the system only at the input.
As the full adder blocks are dependent on their predecessor blocks' carry value, the entire system works a little slow.	Since the entire system depends on the first carry input, the computations are very quick, making it the fastest adder.
It has a simple repetitive design.	Has a slightly complicated design with many logic gates
The system design is cheap to manufacture.	The manufacturing process is expensive as compared to other systems.
The ripple carry adder chips have a considerable size and area.	The chip area increases, as there are many components in the circuit.

Combinational Circuits

Overview

- **BCD Adder**
- **BCD Subtractor**
- **Comparator**
- **Error detection and correction codes**

BCD

Decimal Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

- Inputs: $A_3A_2A_1A_0$, $B_3B_2B_1B_0, C_{in}$ from previous decade.
- Output: C_{out} (carry to next decade), $Z_3Z_2Z_1Z_0$.
- Idea: Perform regular binary addition and then apply a corrective procedure.

BCD Addition Rules

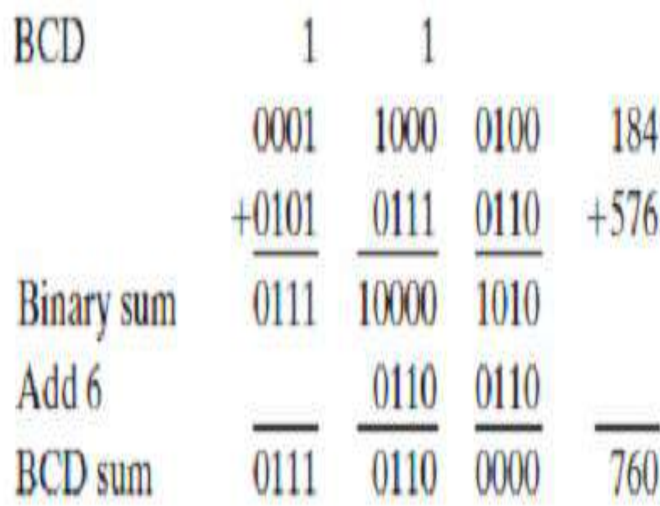
BCD addition

Add two numbers as same as binary addition

Case 1: If the result is less than or equals to 9 and carry is zero then it is valid BCD.

Case 2: If result is greater than 9 and carry is zero then add 6 in four bit combination.

Case 3: If result is less than or equals to 9 but carry is 1 then add 6 in four bit combination.



Comparing Binary and BCD Sums

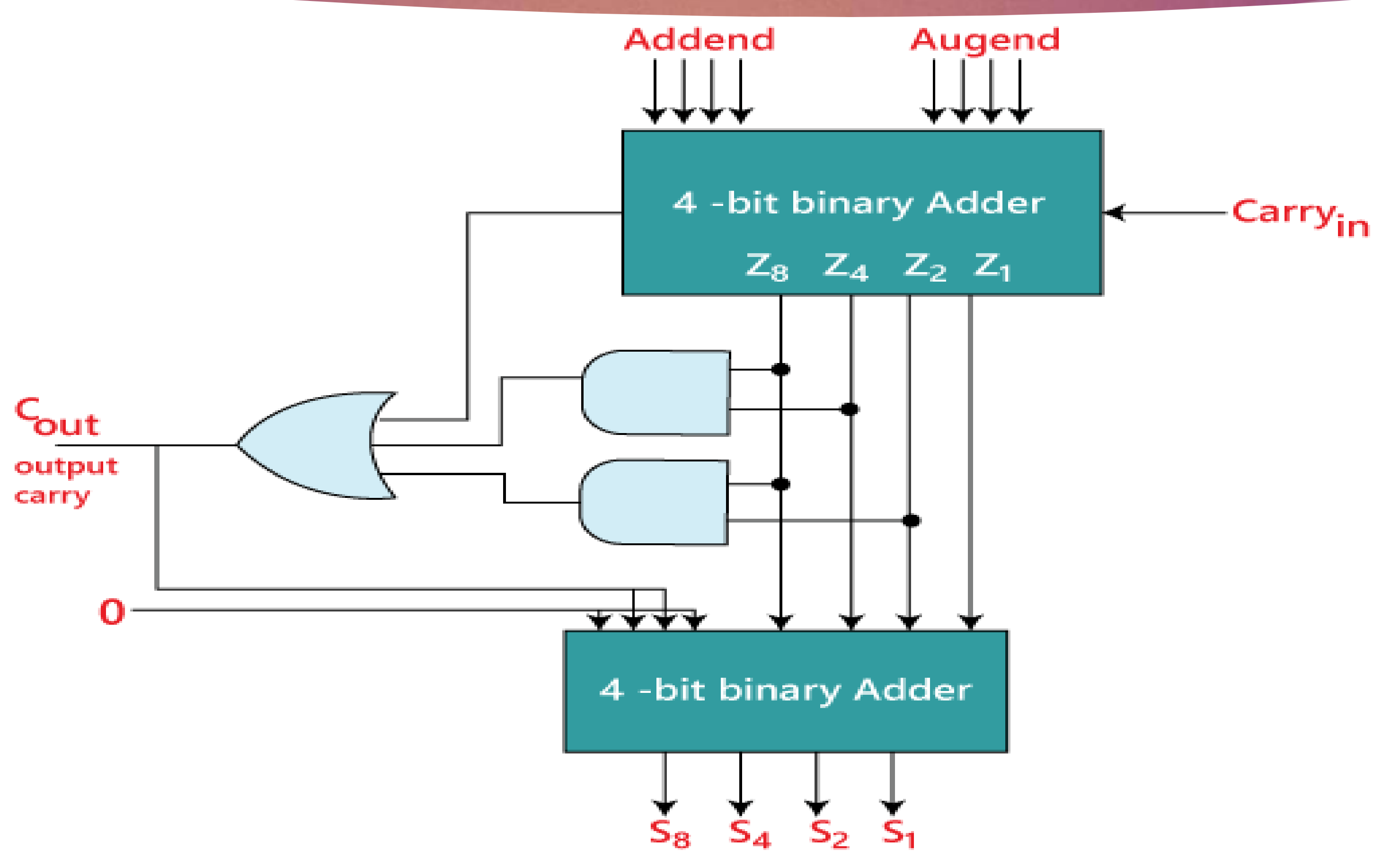
Binary Sum						BCD Sum						Decimal
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1		
0	0	0	0	0	S	0	0	0	0	0		0
0	0	0	0	1	A	0	0	0	0	1		1
0	0	0	1	0	M	0	0	0	1	0		2
. . . .	. . . .	. . . .	. . . .	. . . .	E C O	. . . .	. . . .	. . . .	. . . .	. . . .		. . . .
0	1	0	0	0	D	0	1	0	0	0		8
0	1	0	0	1	E	0	1	0	0	1		9
10 to 19 Binary and BCD codes are not the same												
0	1	0	1	0		1	0	0	0	0		10
0	1	0	1	1		1	0	0	0	1		11
0	1	1	0	0		1	0	0	1	0		12
0	1	1	0	1		1	0	0	1	1		13
0	1	1	1	0		1	0	1	0	0		14
0	1	1	1	1		1	0	1	0	1		15
1	0	0	0	0		1	0	1	1	0		16
1	0	0	0	1		1	0	1	1	1		17
1	0	0	1	0		1	1	0	0	0		18
1	0	0	1	1		1	1	0	0	1		19

BCD Adder

- In the previous table Decimal sum from **0 to 9**, the Binary sum same as BCD sum. So, no conversion is needed.
- Apply correction if the Decimal sum is between **10-19**.
  - The correction is needed (Decimal sum **16-19**)when the binary sum has an output carry $K = 1$
  - The correction is needed (Decimal sum **10-15**)when $Z_8 = 1$ and either $Z_4 = 1$ or $Z_2 = 1$.
- So, the condition for a correction and an output carry can be expressed by the Boolean function:

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

- When $C = 1$, it is necessary to add 0110 to the binary sum to get BCD sum and provide an output carry for the next stage.



Black diagram of a BCD adder

Cascading of BCD Adders

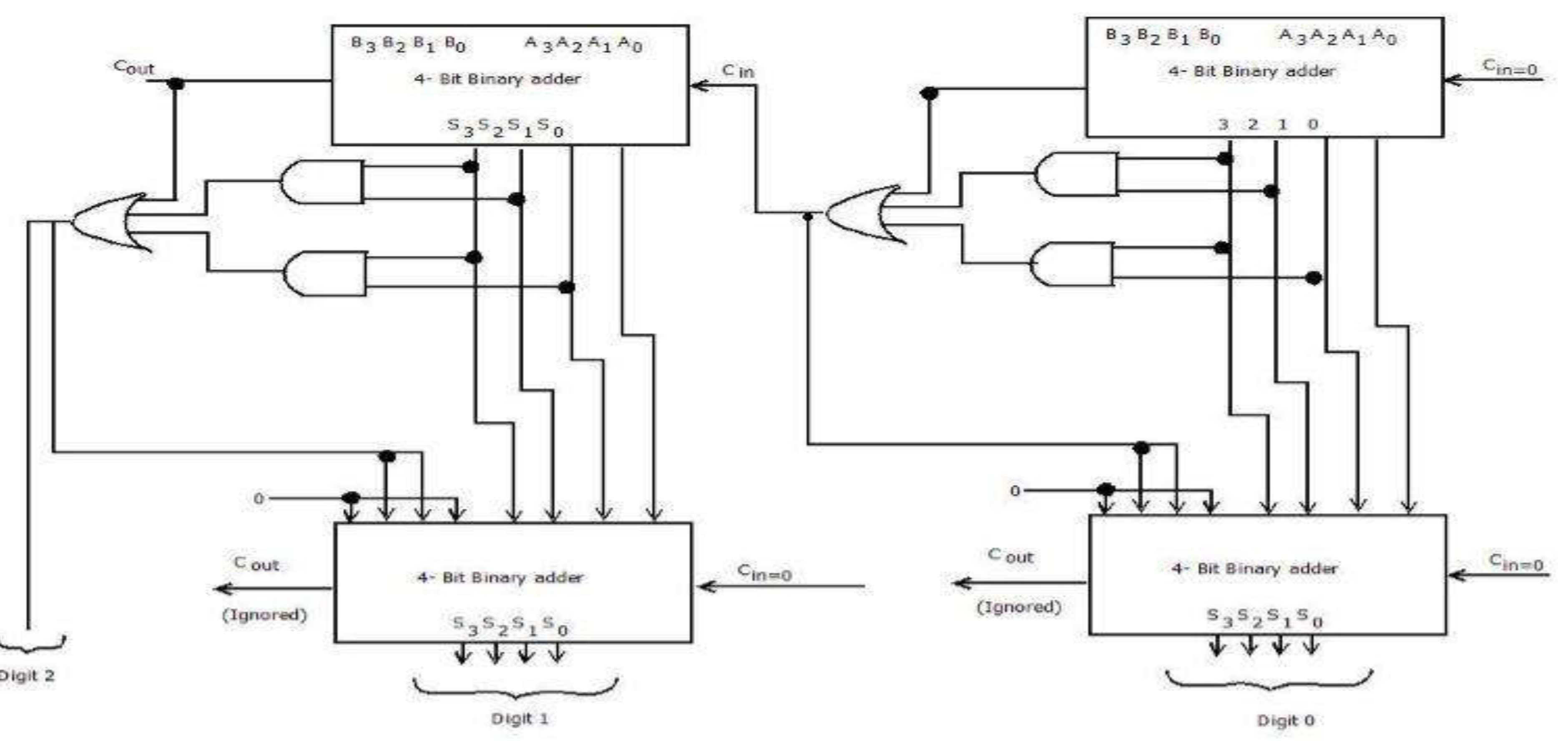


Fig : 8 - Bit BCD Adder

BCD Subtraction Rules

Let two BCD numbers are A and B.
B to be subtracted from A.

RULES:

- Add 9's Complement of B to A
- If result > 9, Correct by adding 0110
- If carry is generated at most significant position then the result is positive and the End around carry must be added
- If carry is not generated at most significant position then the result is negative and the result is 9's complement of original result

BCD number (d)	Binary equivalent of BCD number				9's complement of BCD (9 – d)	Binary equivalent of 9's complement number			
	w	x	y	z		C_3	C_2	C_1	C_0
0	0	0	0	0	9	1	0	0	1
1	0	0	0	1	8	1	0	0	0
2	0	0	1	0	7	0	1	1	1
3	0	0	1	1	6	0	1	1	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	4	0	1	0	0
6	0	1	1	0	3	0	0	1	1
7	0	1	1	1	2	0	0	1	0
8	1	0	0	0	1	0	0	0	1
9	1	0	0	1	0	0	0	0	0

Example

Regular Subtraction | **9's Complement Subtraction**

(a)

```
  8
- 2
---
  6
```

```
     8
  +  7     9's complement of 2
  ----
(1)  5
 └→+ 1     Add carry to result
  ----
     6
```

(b)

```
  28
- 13
----
  15
```

```
    28
  + 86     9' complement of 13
  ----
(1) 14
 └→+ 1     Add carry to the result
  ----
    15
```

(c)

```
  18
- 24
----
  -6
```

```
    18
  + 75     9' complement of 24
  ----
    93     9's complement of result
     ↓     (No carry indicates that the
           answer is negative and in
   -06     complement form)
```

E.G. 8 – 3 = 8 + [9's COMP. OF 3]
= 8 + 6

```
     1000
     0110
     1110   ← INVALID (>9)
     0110   ← CORRECTION
 (1) 0100
  └────→ 1  ← END AROUND CARRY
     0101 = 5
```

(b) 3 – 8 = -5

```
0011
0001
0100
```

NO CARRY >>> NEGATIVE
9's COMP. OF 0100 = **0101** = **-5**

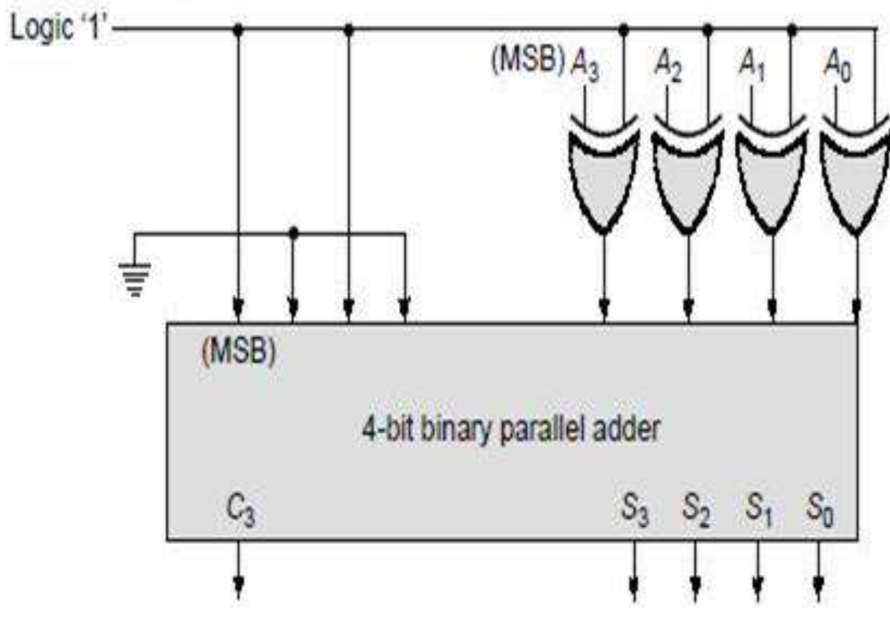
(c) 87 –39 >>> 87 + [9's COMP OF 39]

```
  8 7       1000    0111
  6 0       0110    0000
           ------  ------
 INVALID →  1110    0111
            0110
       (1)  0100    0111
        └──────────→   1
           ------  ------
            0100    1000
  =          4       8
```

9's Complement Circuit

- 9'complement of 2 is 7
- Binary equivalent of 2 is 0010
- 1's complement of 0010 is 1101
- Then, 1101
  + 1010
  = 0111 which is Binary equivalent of 7
- If carry discard it.

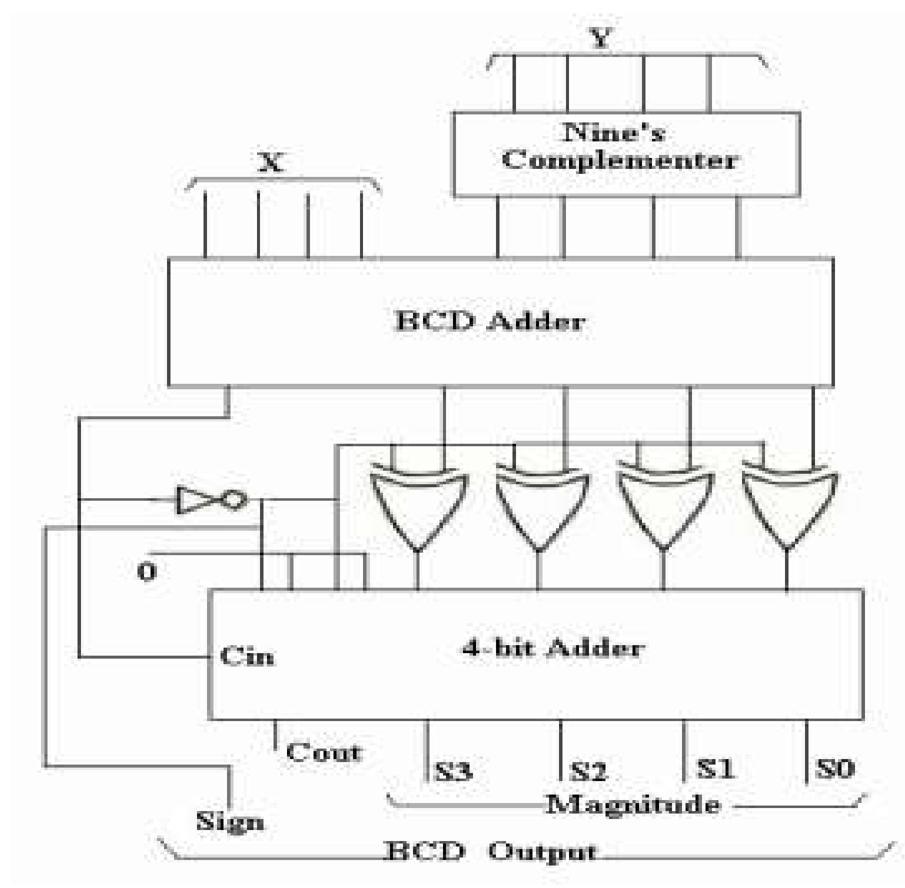
- 9'complement of 3 is 6
- Binary equivalent of 3 is 0011
- 1's complement of 0011 is 1100
- Then, 1100
  + 1010
  = 0110 which is Binary equivalent of 6
- If carry discard it.



BCD Subtractor Circuit

RULES:

- Add 9's Complement of B to A
- If result > 9, Correct by adding 0110
- If carry is generated at most significant position then the result is positive and the End around carry must be added
- If carry is not generated at most significant position then the result is negative and the result is 9's complement of original result



Comparator

- A magnitude digital Comparator is a combinational circuit that **compares two digital or binary numbers** in order to find out whether one binary number is equal, less than or greater than the other binary number.
- We logically design a circuit for which we will have two inputs one for A and other for B and have three output terminals, one for $A > B$ condition, one for $A = B$ condition and one for $A < B$ condition.
- A comparator makes use of a cascade connection of identical sub networks similar to the case of the parallel adder.

1-Bit Magnitude Comparator

- A comparator used to compare two bits is called a single bit comparator.
- It consists of two inputs each for two single bit numbers and three outputs to generate less than, equal to and greater than between two binary numbers.

From the above truth table logical expressions for each output can be expressed as follows:

A	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

A>B: AB' **A<B: A'B** **A=B: A'B' + AB**

Logic Diagram

From the above expressions we can derive the following formula:

$$(A<B)+(A>B) = A'B+AB'$$

Taking complement both sides

$$((A<B) + (A>B))' = (A'B + AB')'$$

$$((A<B) + (A>B))' = (A'B)'(AB')'$$

$$((A<B) + (A>B))' = (A + B')(A' + B)$$

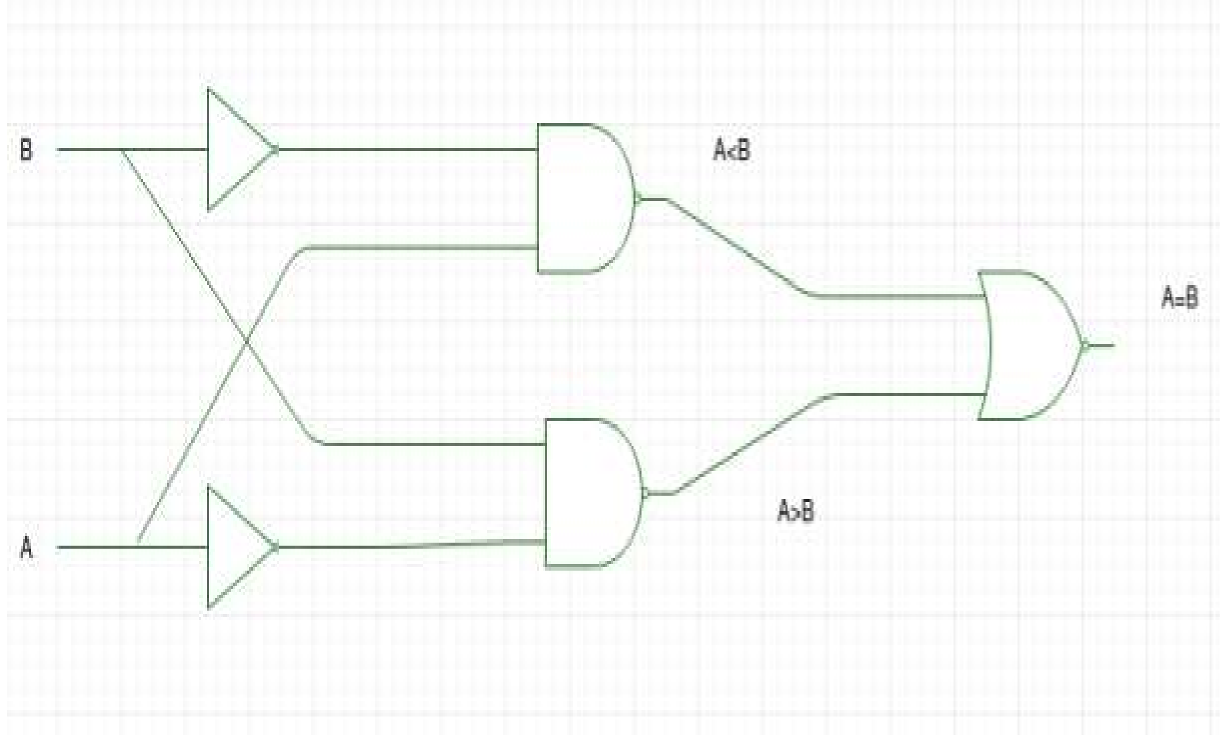
$$((A<B) + (A>B))' = (AA' + AB + A'B' + BB')$$

$$\text{"} \qquad \text{"} \qquad = (AB + A'B')$$

Thus,

$$((A<B) + (A > B))' = (A = B)$$

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below:

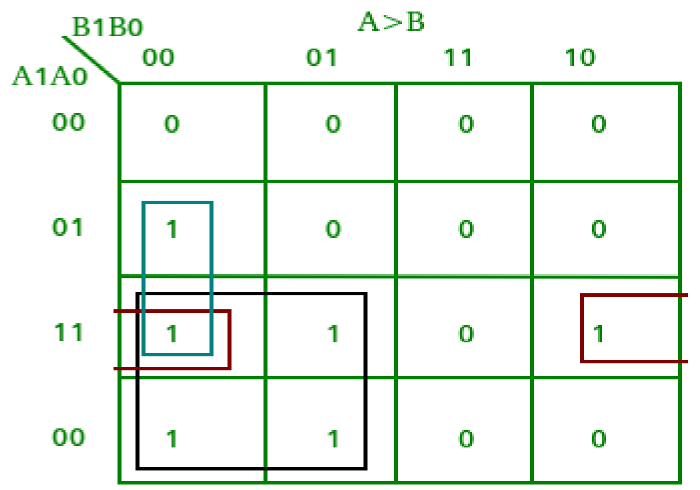


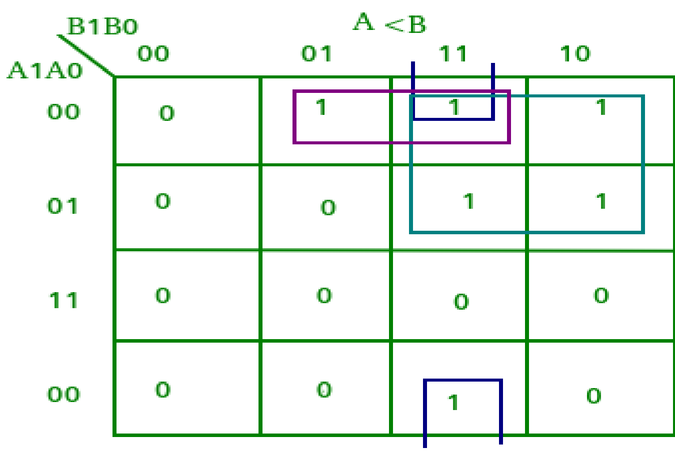
2-Bit Magnitude Comparator

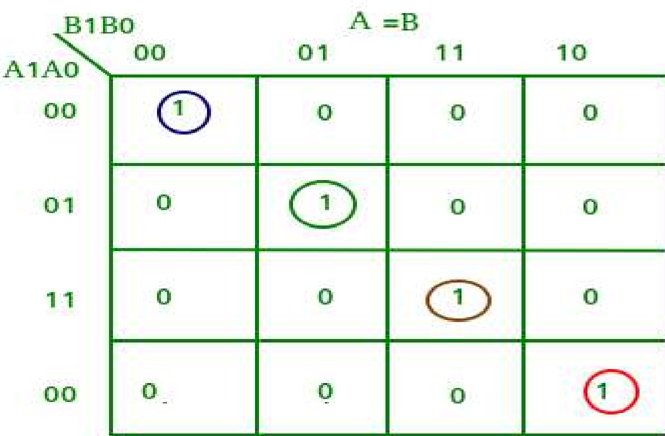
A comparator used to compare two binary numbers each of two bits is called a 2-bit Magnitude comparator. It consists of four inputs and three outputs to generate less than, equal to and greater than between two binary numbers.

INPUT				OUTPUT		
A1	A0	B1	B0	A<B	A=B	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

From the Truth Table K-map for each output can be drawn as follows:







A>B: A1B1’ + A0B1’B0’ + A1A0B0’ **A<B**: A1’B1 + A0’B1B0 + A1’A0’B0

A=B: A1’A0’B1’B0’ + A1’A0B1’B0 + A1A0B1B0 + A1A0’B1B0’

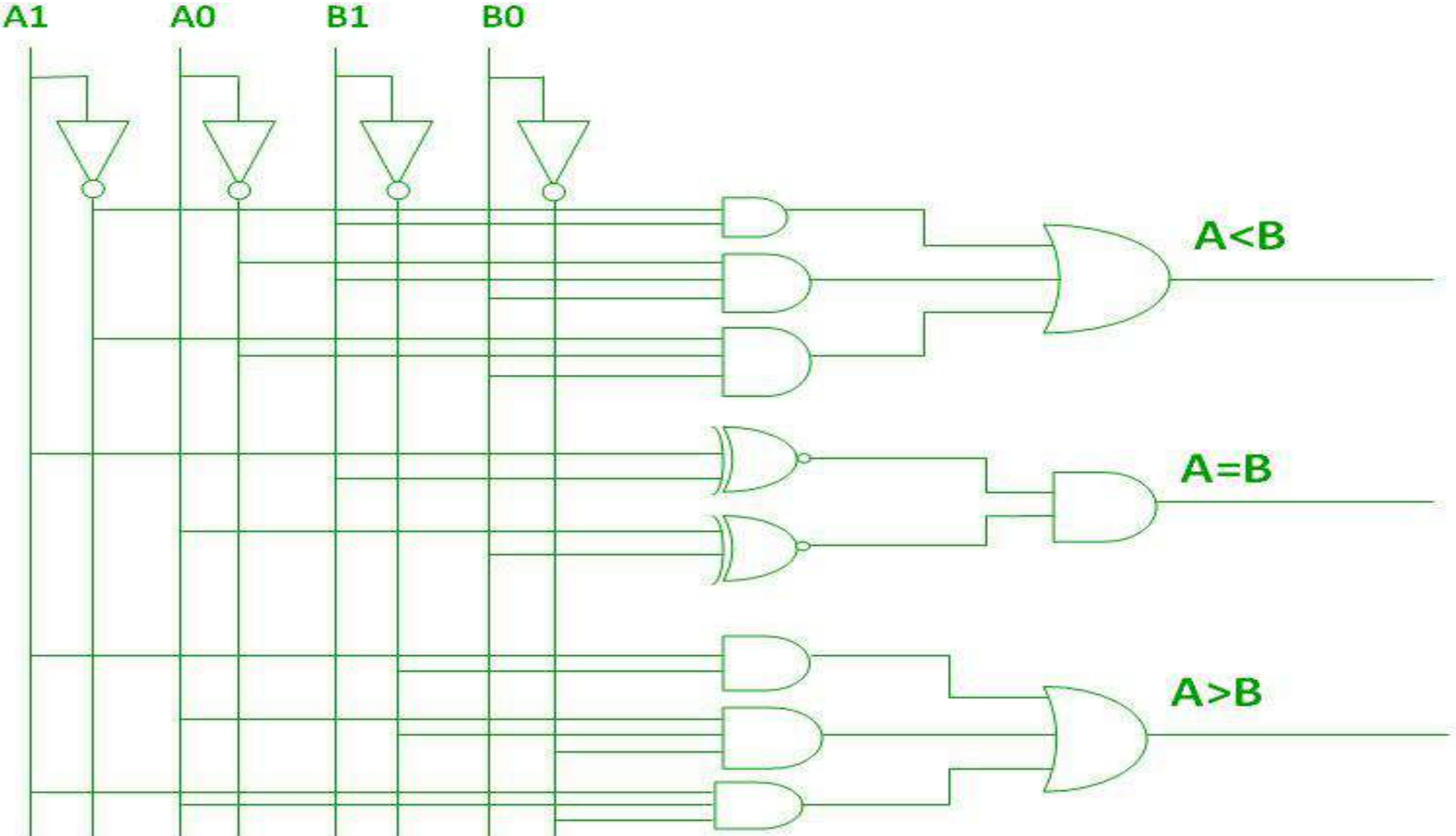
A1’B1’ (A0’B0’ + A0B0) + A1B1 (A0B0 + A0’B0’)

(A0B0 + A0’B0’) (A1B1 + A1’B1’)

(A0 Ex-Nor B0) (A1 Ex-Nor B1)

Logic Diagram

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below:



4-Bit Magnitude Comparator

- A comparator used to compare two binary numbers each of four bits is called a 4-bit magnitude comparator.
- It consists of eight inputs each for two four bit numbers.
- Three outputs to generate less than, equal to and greater than between two binary numbers.

In a 4-bit comparator the condition of A = B can be possible in the following four cases:

A = B is possible only when all the individual bits of one number exactly coincide with corresponding bits of another number.

If $A3 = B3$ and $A2 = B2$ and $A1 = B1$ and $A0 = B0$
As the numbers are binary, the digits are either 0 or 1.
The equality relation of each pair of bits can be expressed logically with an equivalence function.

$x_i = A_iB_i + A_i'B_i'$ $\quad i = 0, 1, 2, 3 \quad$ where $x_i = 1$ if the pair of bits in position i are equal.

So,

$$(A = B) = x_3 \cdot x_2 \cdot x_1 \cdot x_0$$

In a 4-bit comparator the condition of A>B can be possible in the following four cases:

If **A3 = 1 and B3 = 0**
If **A3 = B3, A2 = 1 and B2 = 0**
If **A3 = B3, A2 = B2, A1 = 1 and B1 = 0**
If **A3 = B3, A2 = B2, A1 = B1, A0 = 1 and B0 = 0**
The sequential comparison can be expressed logically as:

$$(A>B) = A3B3' + x3\ A2B2' + x3x2\ A1B1' + x3x2x1\ A0B0'$$

In a 4-bit comparator the condition of A<B can be possible in the following four cases:

If **A3 = 0 and B3 = 1**
If **A3 = B3, A2 = 0 and B2 = 1**
If **A3 = B3, A2 = B2, A1 = 0 and B1 = 1**
If **A3 = B3, A2 = B2, A1 = B1, A0 = 0 and B0 =1**
The sequential comparison can be expressed logically as:

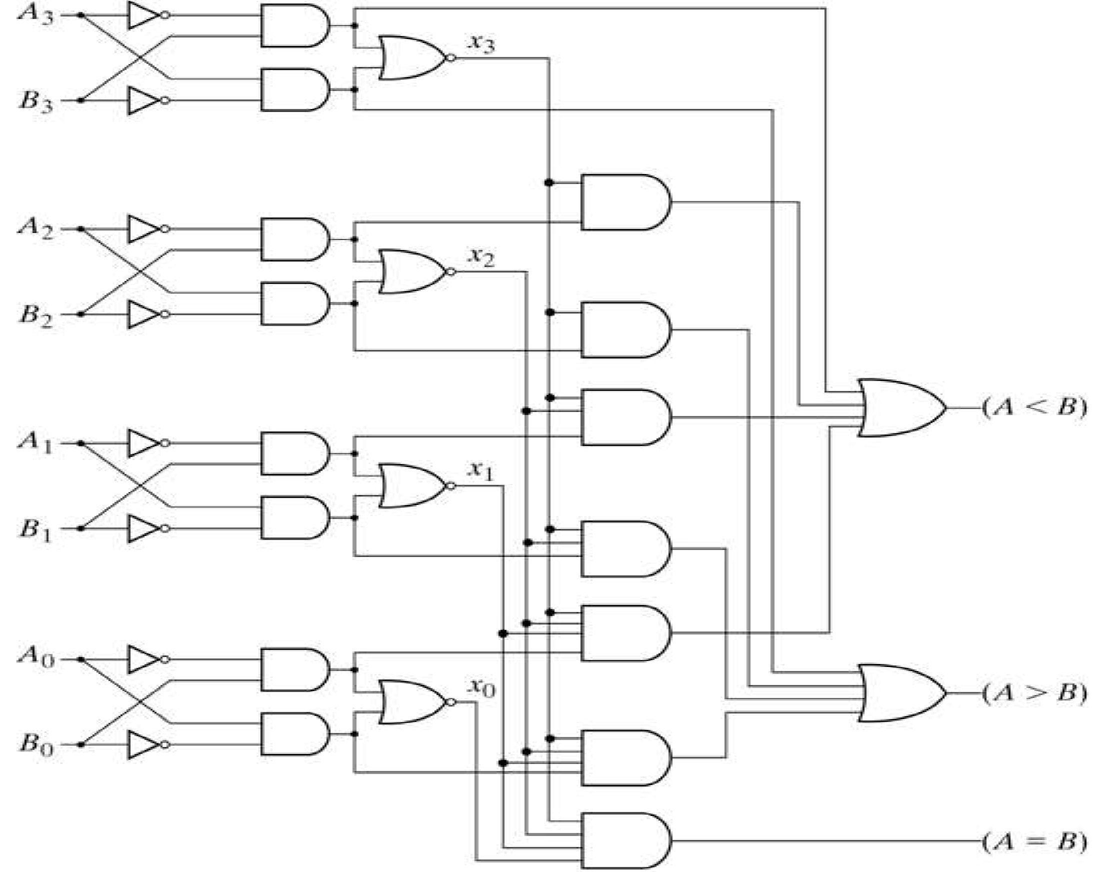
$$(A<B) = A3'B3 + x3\ A2'B2 + x3x2\ A1'B1 + x3x2x1\ A0'B0$$

Logic Diagram

$(A = B) = x3 \cdot x2 \cdot x1 \cdot x0$

$(A>B) = A3B3' + x3\,A2B2' + x3x2\,A1B1' + x3x2x1\,A0B0'$

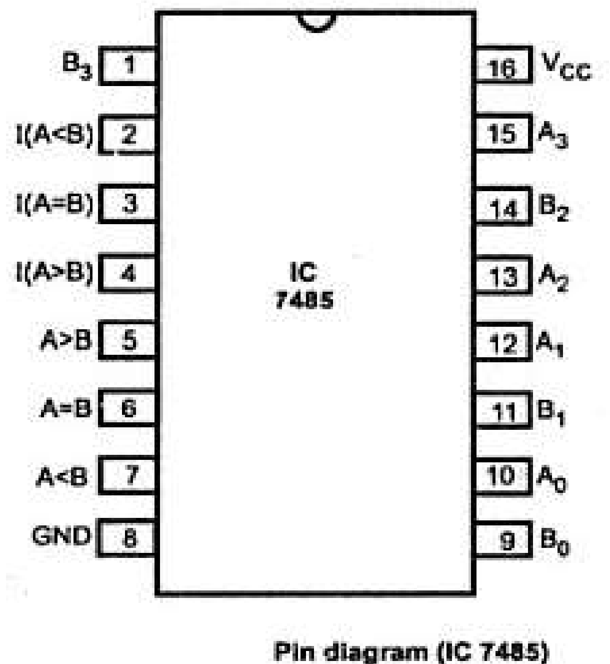
$(A<B) = A3'B3 + x3\,A2'B2 + x3x2\,A1'B1 + x3x2x1\,A0'B0$



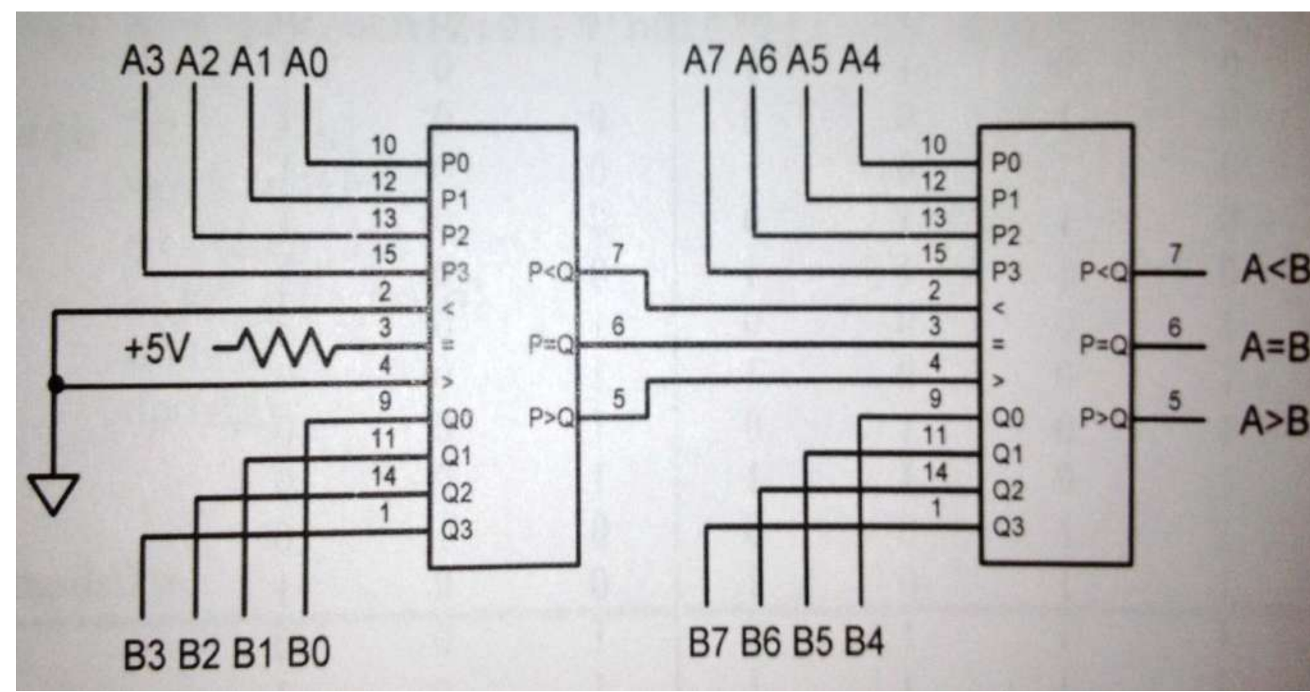
4-Bit Magnitude Comparator

Cascading Comparator

A comparator performing the comparison operation to more than four bits by cascading two or more 4-bit comparators is called cascading comparator.
When two comparators are to be cascaded, the outputs of the **lower-order comparator** are connected to corresponding inputs of the **higher-order comparator**.



Pin diagram (IC 7485)



Applications of Comparators

- Comparators are used in central processing units (CPUs) and microcontrollers (MCUs).
- These are used in control applications in which the binary numbers representing physical variables such as temperature, position, etc. are compared with a reference value.
- Comparators are also used as process controllers and for Servo motor control.
- Used in password verification and biometric applications.

Error Detection and Correction Codes

- Bits 0 and 1 corresponding to two different range of analog voltages. During transmission of binary data from one system to the other, the noise may also be added. Due to this, there may be errors in the received data at other system.
- That means a bit 0 may change to 1 or a bit 1 may change to 0. We can’t avoid the interference of noise. But, we can get back the original data first by detecting whether any errors present and then correcting those errors.
- For this purpose, we can use the following codes.
  - Error detection codes
  - Error correction codes

- **Error detection codes** − are used to detect the errors present in the received data. These codes contain some bits, which are included to the original bit stream. These codes detect the error, if it is occurred during transmission of the original data.

  Example − Parity code, Hamming code, CRC code etc.

- **Error correction codes** − are used to correct the errors present in the received data so that, we will get the original data. Error correction codes also use the similar strategy of error detection codes.

  It also detects the error.

  Example − Hamming code, CRC code etc.

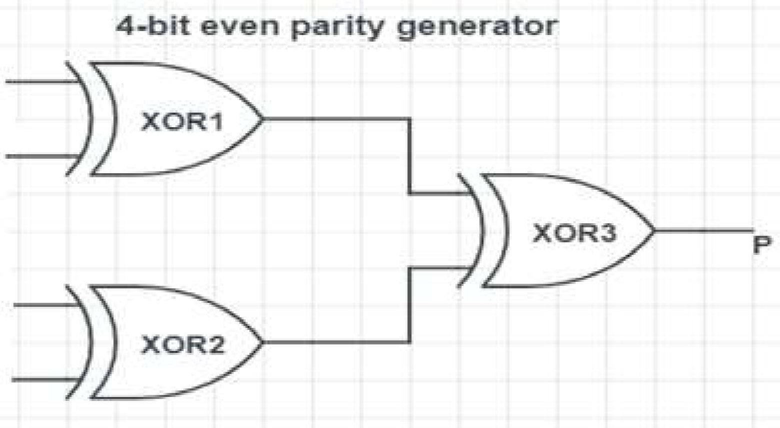
- Therefore, to detect and correct the errors, additional bits are appended to the data bits at the time of transmission.

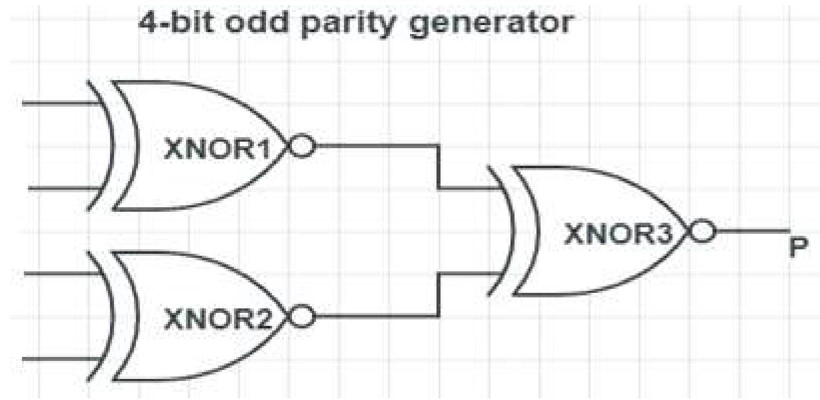
Parity Code Method

- A parity bit is an extra bit included in binary message to make total number of 1's either odd or even.
- Parity word denotes number of 1's in a binary string.
- There are two parity system-Even Parity and Odd Parity.
- **In even parity system** 1 is appended to binary string if there is an odd number of 1's in string otherwise 0 is appended to make total even number of 1's.
- **In odd parity system**, 1 is appended to binary string if there is even a number of 1's to make an odd number of 1's.
- The receiver knows that whether sender is an odd parity generator or even parity generator.
- Suppose if sender is an odd parity generator then there must be an odd number of 1's in received binary string.
- If an error occurs to a single bit that is either bit is changed to 1 to 0 or 0 to 1, received binary bit will have an even number of 1's which will indicate an error.

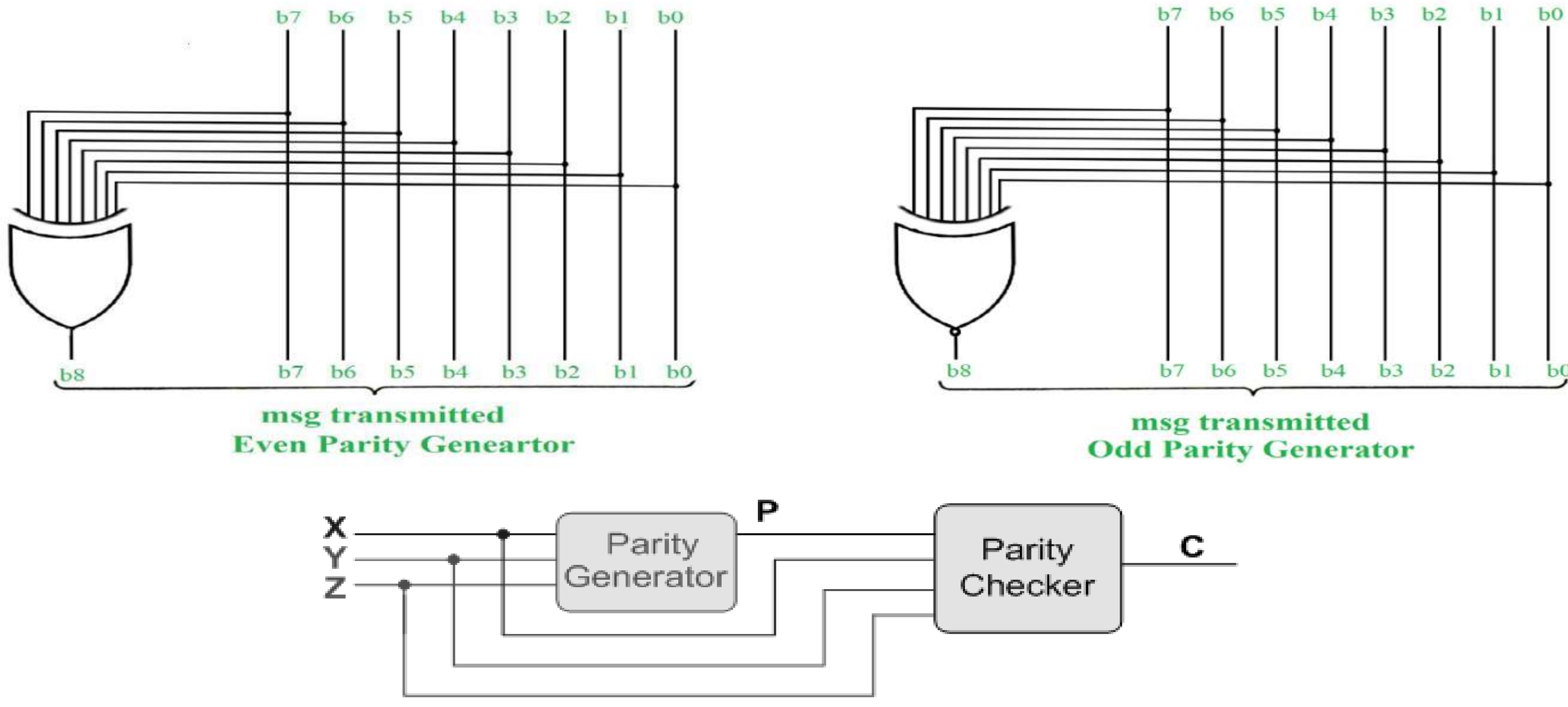
Parity Generator

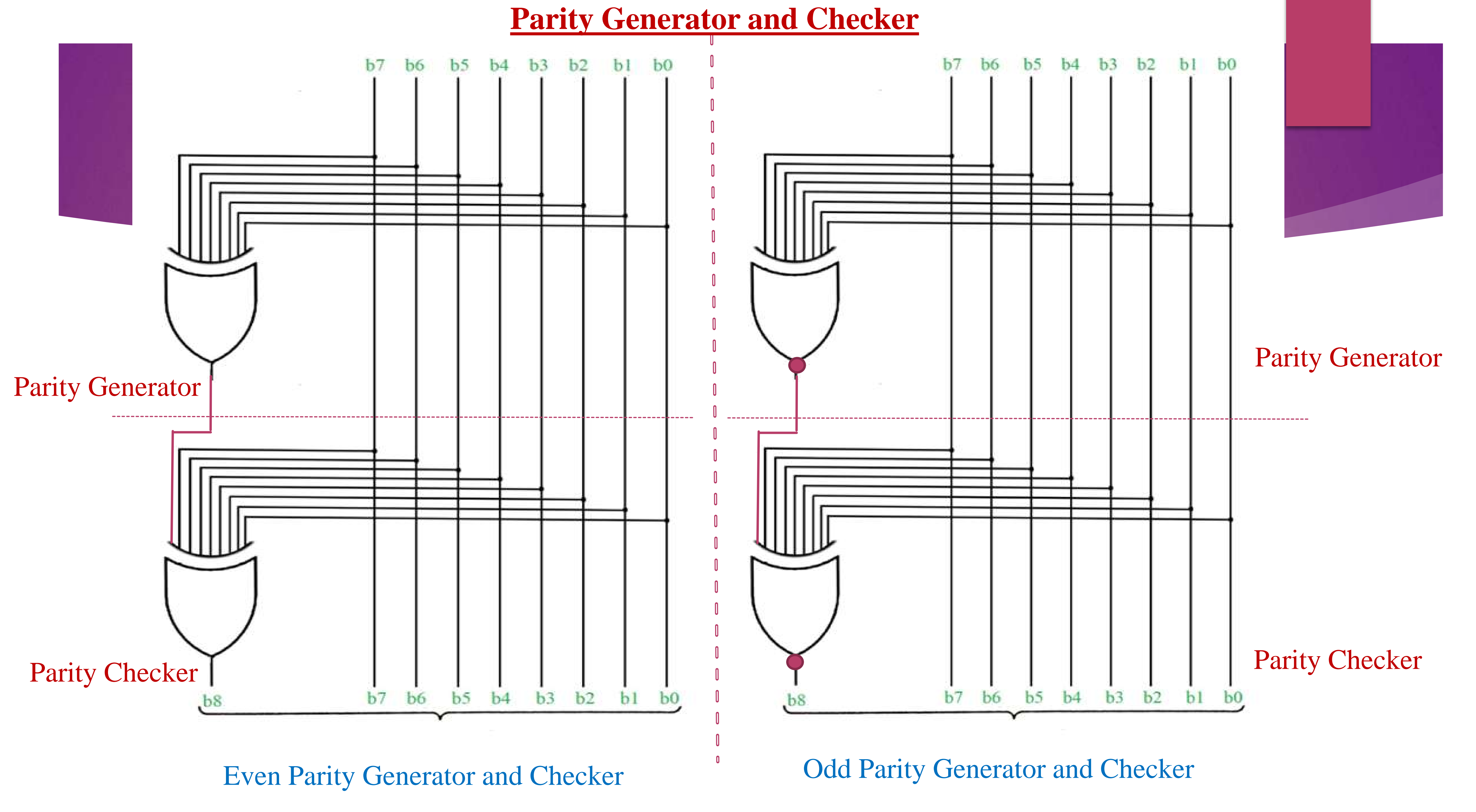
D_3	D_2	D_1	D_0	Even-parity P	Odd-parity P
0	0	0	0	0	1
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	0	1





Parity Generator and Checker




Parity Generator and Checker
b7 b6 b5 b4 b3 b2 b1 b0
Parity Generator
Parity Checker
b8 b7 b6 b5 b4 b3 b2 b1 b0
Even Parity Generator and Checker
b7 b6 b5 b4 b3 b2 b1 b0
Parity Generator
Parity Checker
b8 b7 b6 b5 b4 b3 b2 b1 b0
Odd Parity Generator and Checker

- The limitation of this method is that only error in a single bit would be identified.
- It does not tell which bit is incorrect .
- It also can not correct the incorrect bit.
- To overcome this another code called Hamming Code is used to detect an error.
- It indicates which bit is in error.
- It also correct that error.
- Because of this Hamming Code is called as self correcting code.

Hamming Code

- It was developed by R.W. Hamming for error correction.
- Hamming code is useful for both detection and correction of error present in the received data.
- This code uses multiple parity bits and we have to place these parity bits in the positions of powers of 2.
- The **minimum value of 'k'** for which the following relation is correct is nothing but the required number of parity bits.

  $2k \geq n + k + 1$ Where, ‘n’ is the number of bits in the binary code, ‘k’ is the number of parity bits

- Therefore, the number of bits in the Hamming code is equal to n + k.
- Based on requirement, we can use either even parity or odd parity while forming a Hamming code. But, the same parity technique should be used in order to find whether any error present in the received data.

- Let us find the Hamming code for 4-bit binary code
- We can find the required number of parity bits by using the following mathematical relation.
- $2k \geq n + k + 1$
- Substitute, $n = 4$ in the above mathematical relation.
- $\Rightarrow 2k \geq 4 + k + 1 \Rightarrow 2k \geq 5 + k$
- The minimum value of k that satisfied the above relation is 3. Hence, we require 3 parity bits.
- Therefore, the number of bits in Hamming code will be 7, since there are 4 bits in binary code and 3 parity bits.

- We have to place the parity bits and bits of binary code in the Hamming code as shown below.
- Now the Hamming code word format will be ***d7 d6 d5 p4 d3 p2 p1***, where **'*d*'** represents the data bit and **'*p*'** represents the parity bit.
- The parity bit *p1, p2* and *p4* are assigned values by the following three parity relations.
- ***p1 = d7 ⊕ d5 ⊕ d3*** ***p2 = d7 ⊕ d6 ⊕ d3*** ***p4 = d7 ⊕ d6 ⊕ d5***

Example: 1

Construct an even parity seven bit Hamming code for a word 1011.

d7	***d6***	***d5***	***p4***	***d3***	***p2***	***p1***
1	***0***	***1***	***?***	***1***	***?***	***?***

From first relation to have even parity ***p1*** should be 1. From second relation to have even parity ***p2*** should be 0. From third relation to have even parity ***p4*** should be 0. So, the final Hamming code is ***1 0 1 0 1 0 1.***

- For finding the position of error the following relations are to be followed.
- $x = d7 \oplus d5 \oplus d3 \oplus p1$ $y = d7 \oplus d6 \oplus d3 \oplus p2$ $z = d7 \oplus d6 \oplus d5 \oplus p4$
- The parity check may be even parity or odd parity
- If parity relation is satisfied then ***x*** or ***y*** or ***z*** equal to ***0***, otherwise ***1***.

Example: 2

The Hamming code is received 1010001. What was the correct code transmitted.

The code received ***d7 d6 d5 p4 d3 p2 p1***

1 0 1 0 0 0 1

Applying first parity relation $x = 1$. Applying second parity relation $y = 1$. Applying third parity relation $z = 0$.

So, $z\,y\,x = 011$, which is equal to *3*, that is, third data bit is erroneous one and should be corrected as *1* instead of *0*. Now, the correct code is ***1 0 1 0 1 0 1.***

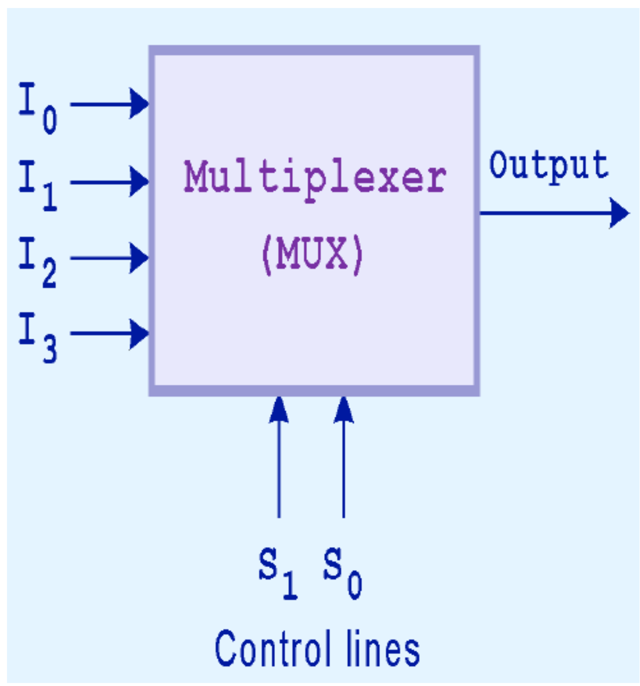
Combinational Circuits

Overview

- **Multiplexer**
- **De-Multiplexer**
- **Decoder**
- **Encoder**
- **Priority Encoder**
- **BCD to Seven Segment Display**

Multiplexer

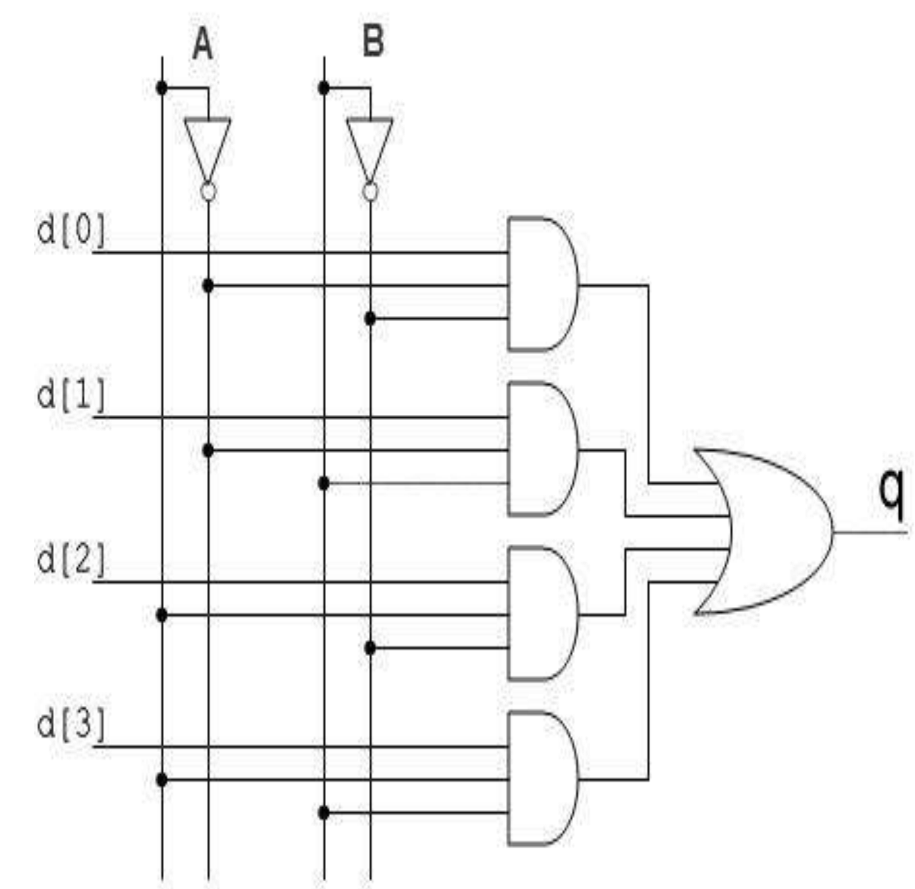
- A Multiplexer or Mux is a device that has many inputs and a single output.
- It selects a single input to the output from several inputs.
- The particular input chosen for output is determined by the value of the multiplexer's control lines.
- To be able to select among *n* inputs, $\log_2 n$ control lines are needed.
- A multiplexer is also called as a data selector.
- The main purpose of Mux is to perform high speed switching.
- In analog applications, these are made up of transistor switches and relays, whereas in digital applications, these are made up of logic gates.



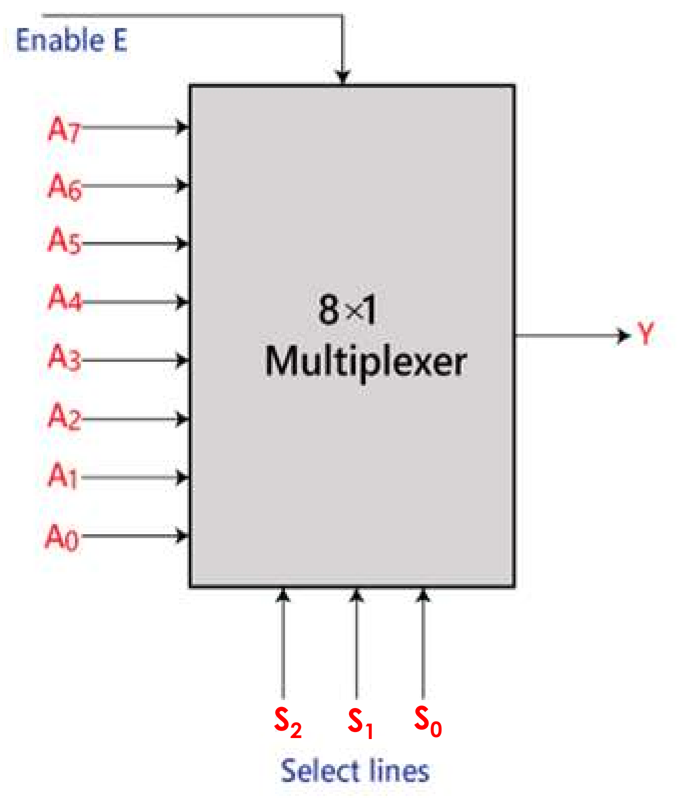
Block diagram of Multiplexer

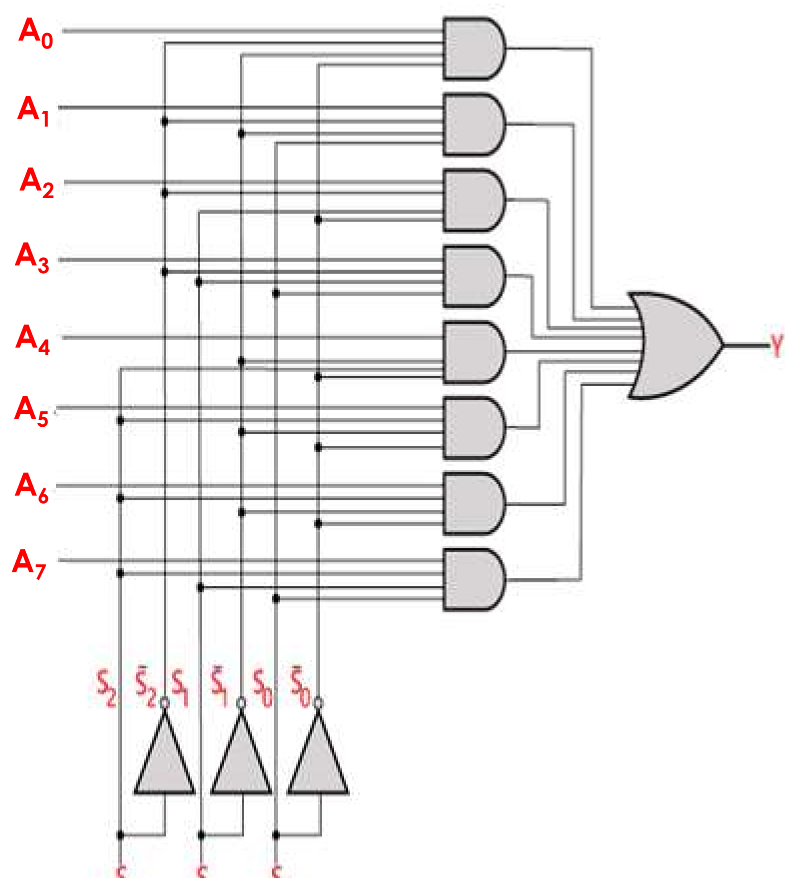
4-to-1 multiplexer

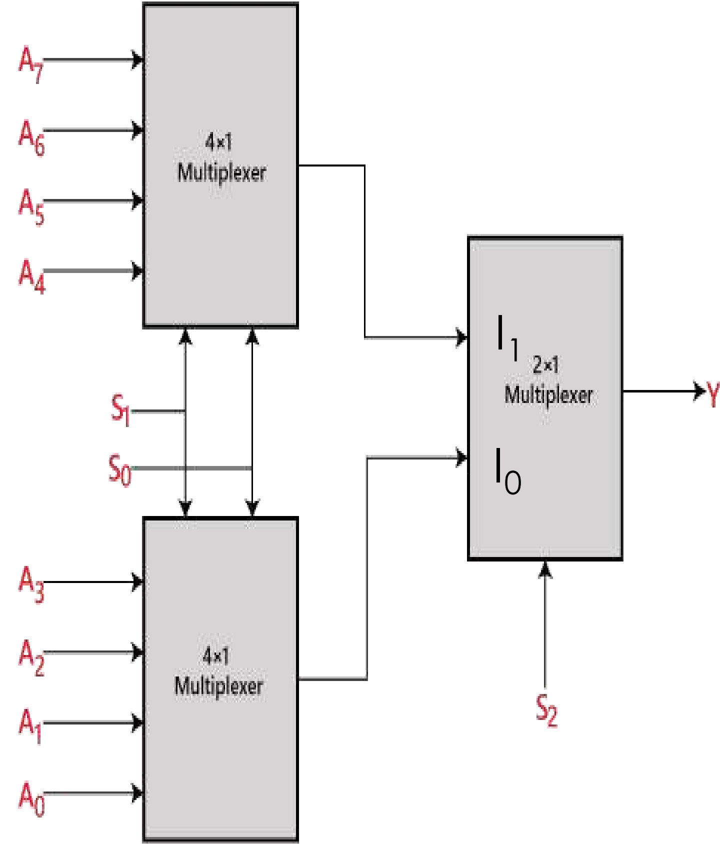
- This is what a 4-to-1 multiplexer looks like on the inside.
- The 4X1 multiplexer comprises 4-input bits, 1-output bit, and 2- control bits.
- The control bit AB decides which of the i/p data bit should transmit the output.
- For example, when the control bits AB =00, then the higher AND gate are allowed while remaining AND gates are restricted. Thus, data input d0 is transmitted to the output 'q"



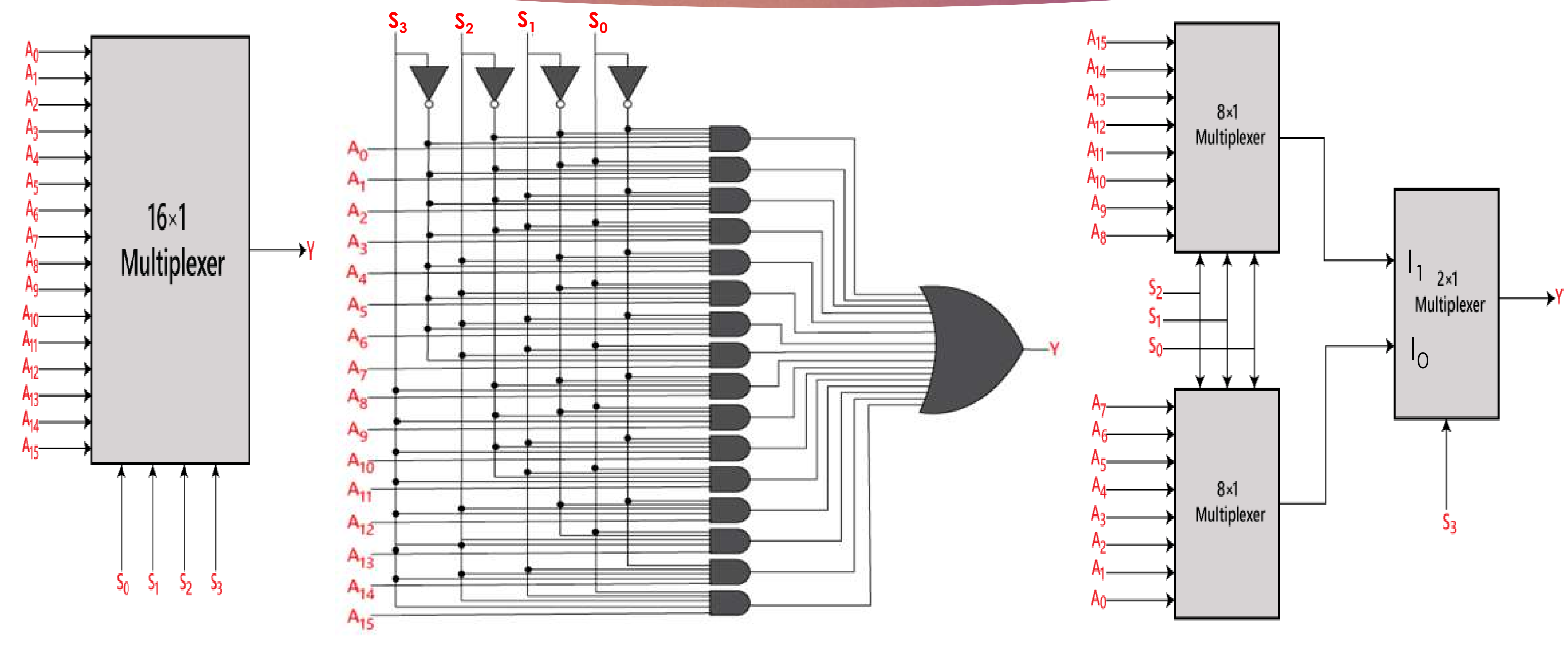
8-to-1 multiplexer







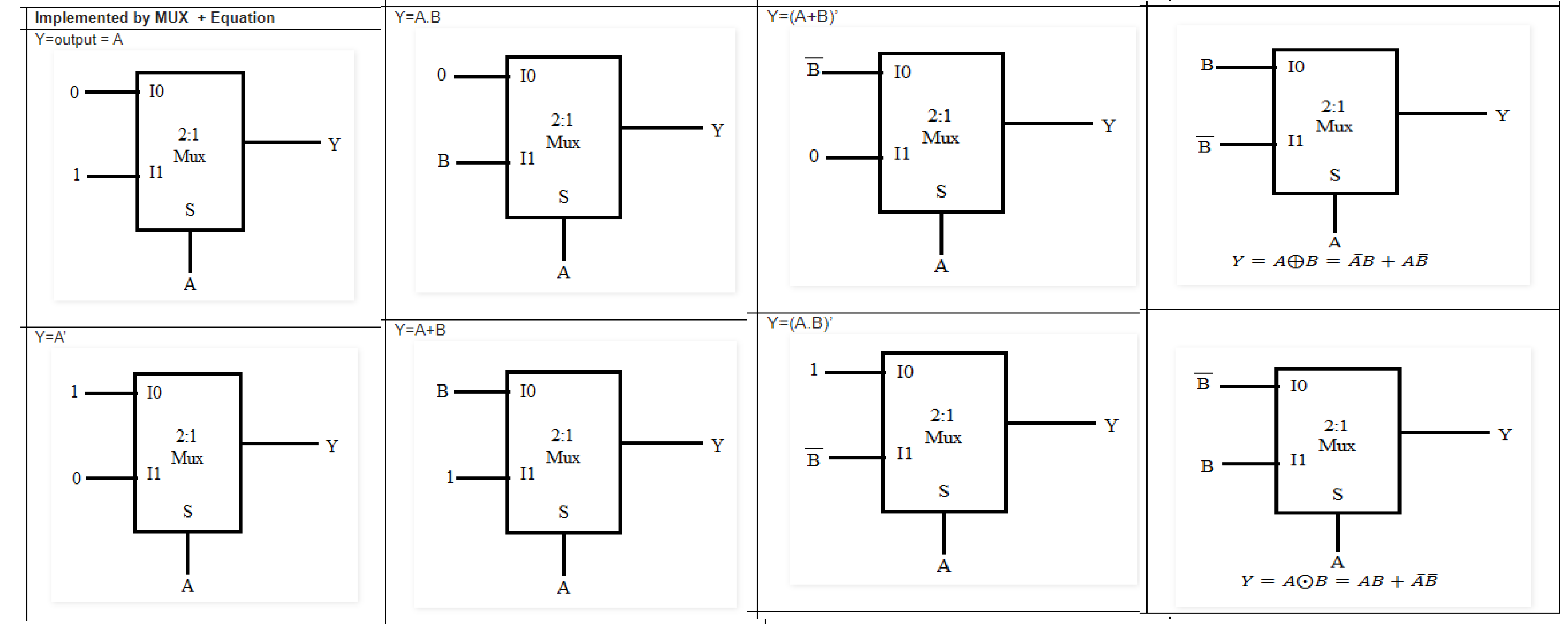
16-to-1 multiplexer



Applications

- A Multiplexer is used in various applications wherein multiple data can be transmitted using a single line.
- A Multiplexer is used to increase the efficiency of the communication system by allowing the transmission of data, such as audio & video data from different channels via cables and single lines.
- A Multiplexer is used in computer memory to decrease the number of copper lines necessary to connect the memory to other parts of the computer.
- A multiplexer is used in telephone networks to integrate the multiple audio signals on a single line of transmission.
- A Multiplexer is used to transmit the data signals from the computer system of a satellite to the ground system by using a GSM (Global System for Mobile communication) communication.

MUX as Universal Logic Circuit



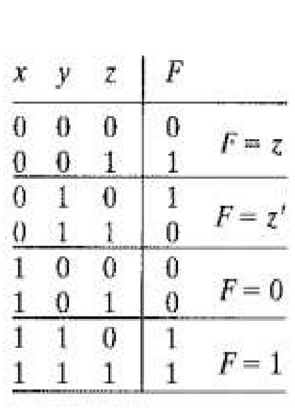
Boolean function implementation using Mux

Multiplexer Example

Implement the following Boolean function using a 4x1 Mux;

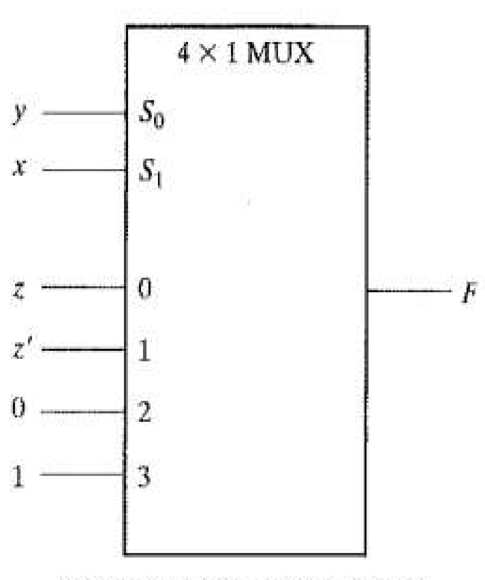
$F(x,y,z) = \Sigma\,(1,2,6,7)$

Solution



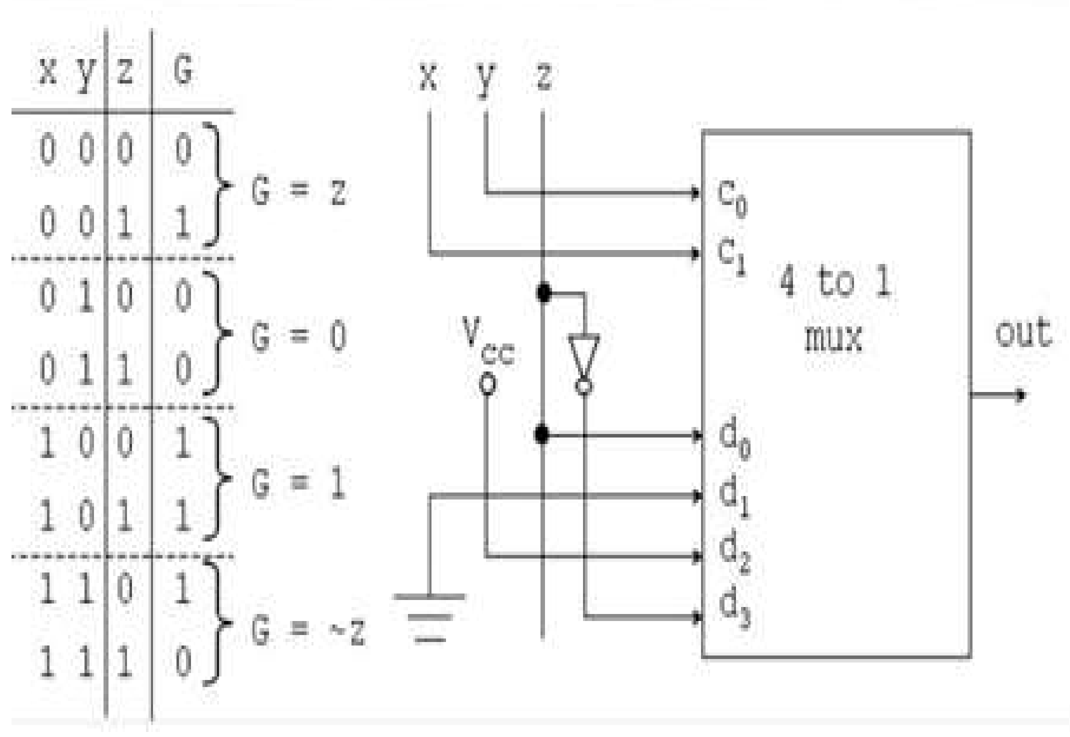
x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

$G(x,y,z) = m(\,1, 4, 5, 6\,)$



x	y	z	G	
0	0	0	0	G = z
0	0	1	1	
0	1	0	0	G = 0
0	1	1	0	
1	0	0	1	G = 1
1	0	1	1	
1	1	0	1	G = ~z
1	1	1	0	

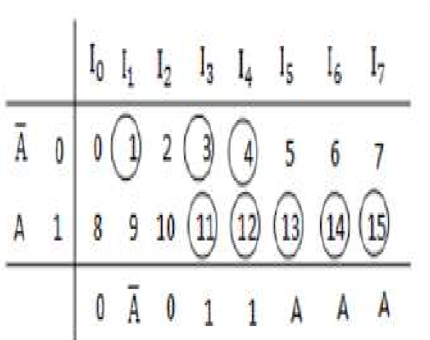
Example: Implementation of given function using 8 to 1 multiplexer

$F(A,B,C,D) = \Sigma(1,3,4,11,12,13,14,15)$

Solution.

- Total number of variable n = 4 (A,B,C,D)
- Number of select lines: n-1= 3 (B, C, D)
- The given function has 4 variable, so 16 possible minterms (0 – 15) are entered in the implementation table.
- All the minterms are divided into 2 groups
  - The first group (0-7) minterms are entered in the first row (Variable A =0)
  - The second group (8–15) minterms are entered in the second row (Variable A= 1)
- Circle the minterm number as per function, which you have to implement (in this case it's 1,3,4,11,12,13,14,15)
- Find out the multiplexer input as per above given steps.

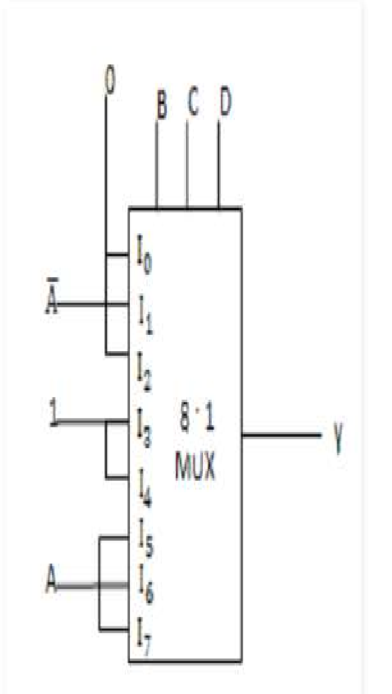
Implementation Table



		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
$\bar{A}$	0	0	(1)	2	(3)	(4)	5	6	7
A	1	8	9	10	(11)	(12)	(13)	(14)	(15)
		0	$\bar{A}$	0	1	1	A	A	A

Given multiplexer is 8:1

Logic diagram



Rules:

- If two min-terms are not circled in a coloumn, apply 0 to Mux input.
- If two min-terms are circled in a coloumn, apply 1 to Mux input.
- If bottom one is circled and top one is not circled in a column, apply A to Mux input.
- If bottom one is not circled and top one is circled in a column, apply A' to Mux input.

Example

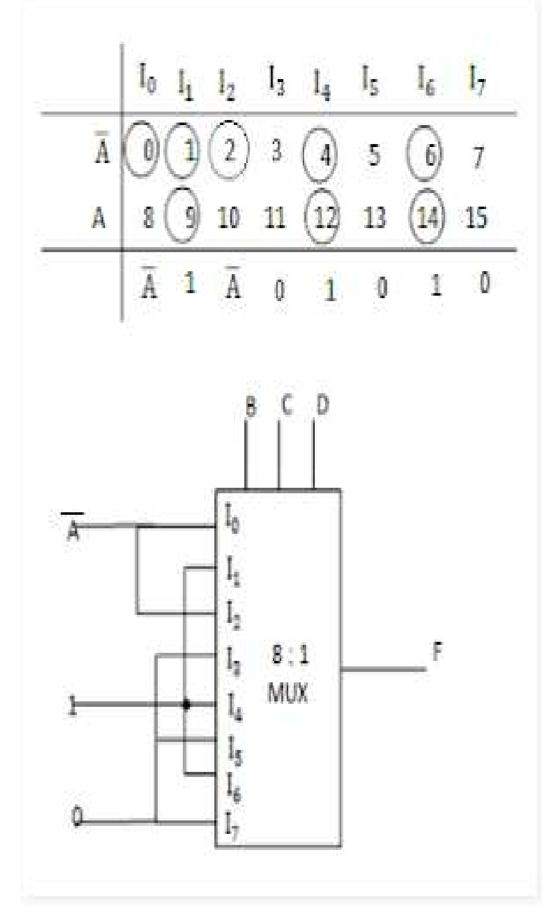
Implement the following Boolean function using 8 : 1 MUX

$F(A,B,C,D) = \Sigma m(0,1,2,4,6,9,12,14)$

Solution.

Select lines are B, C and D

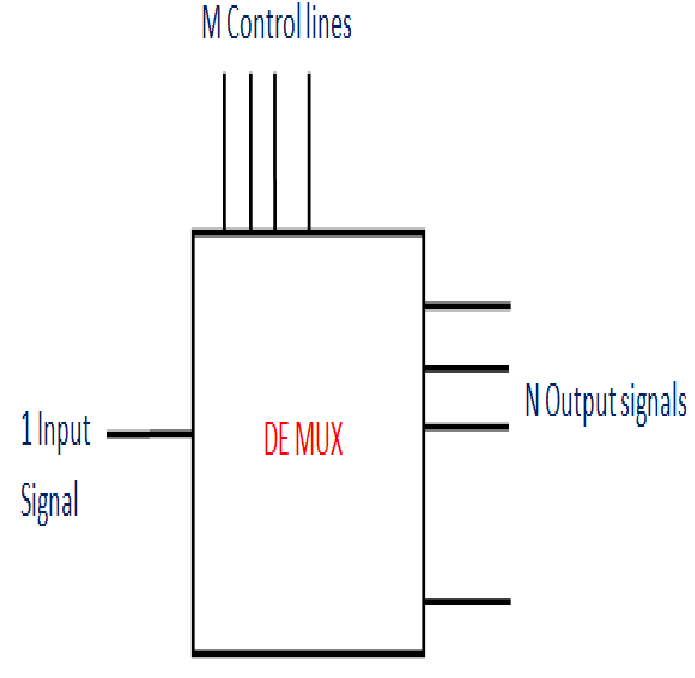
Follow all the steps as per above points.



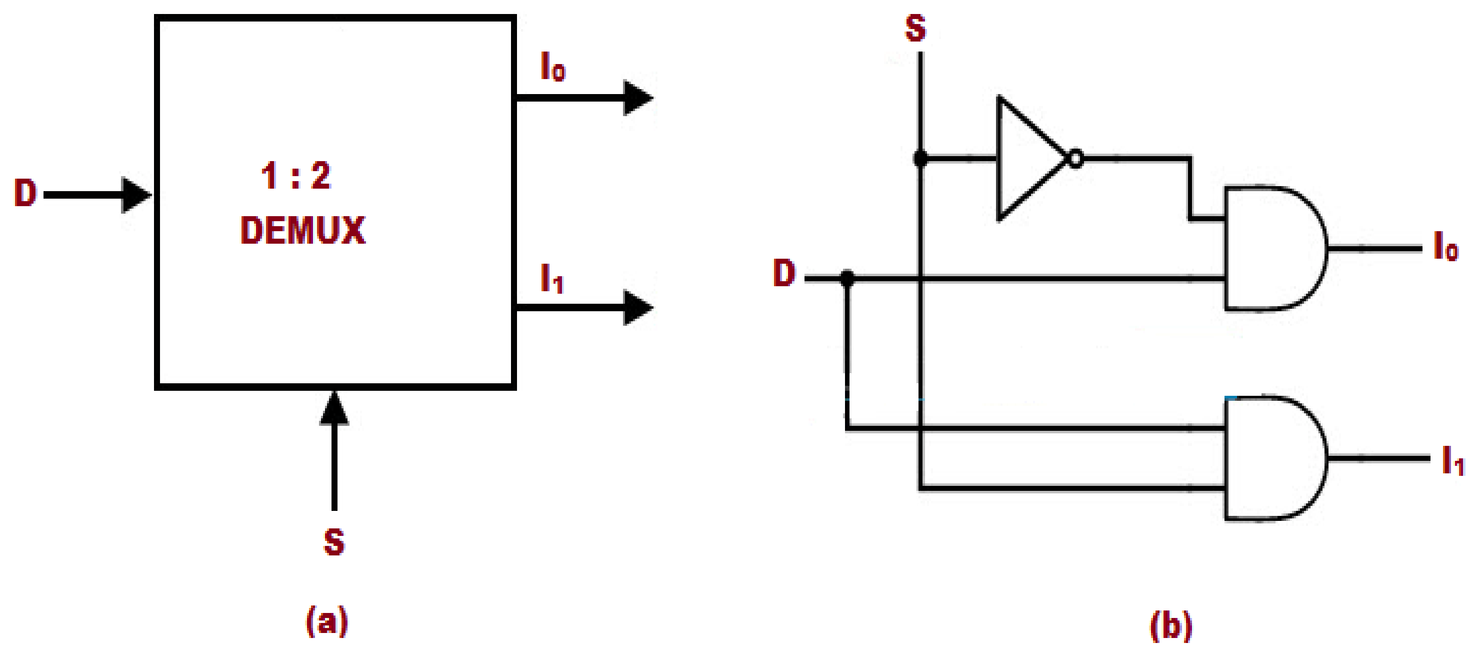
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
$\bar{A}$	(0)	(1)	(2)	3	(4)	5	(6)	7
A	8	(9)	10	11	(12)	13	(14)	15
	$\bar{A}$	1	$\bar{A}$	0	1	0	1	0

Demultiplexer

- **A Demultiplexer** or **Demux** is a circuit which can distribute or deliver multiple outputs from a single input.
- It can perform as single input many output switch.
- The output lines of demultiplexer are 'N' in number, select line number is 'M' and $N = 2^M$.
- The control signal or select input code decides the output line to which the input has to be transmitted.
- It is also called as **Data distributor.**
- There are several types of Demultiplexers
  - 1:2 Demultiplexer or 1-to-2 Demultiplexer
  - 1:4 Demultiplexer
  - 1:8 Demultiplexer
  - 1:16 Demultiplexer

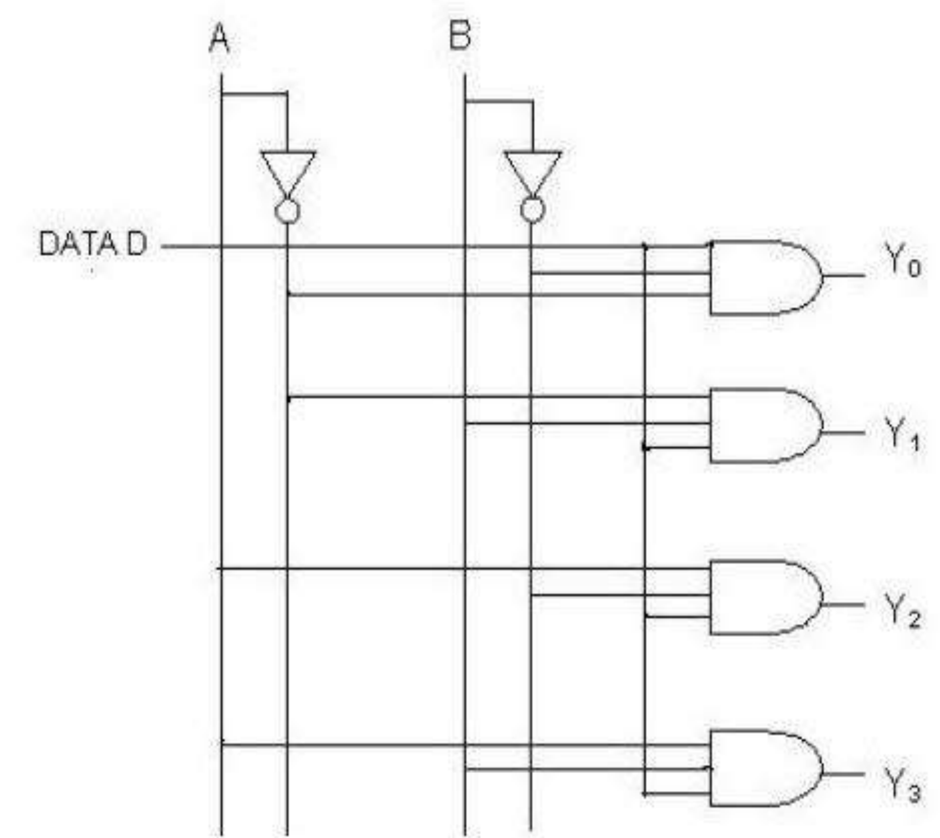


1:2 Demultiplexer

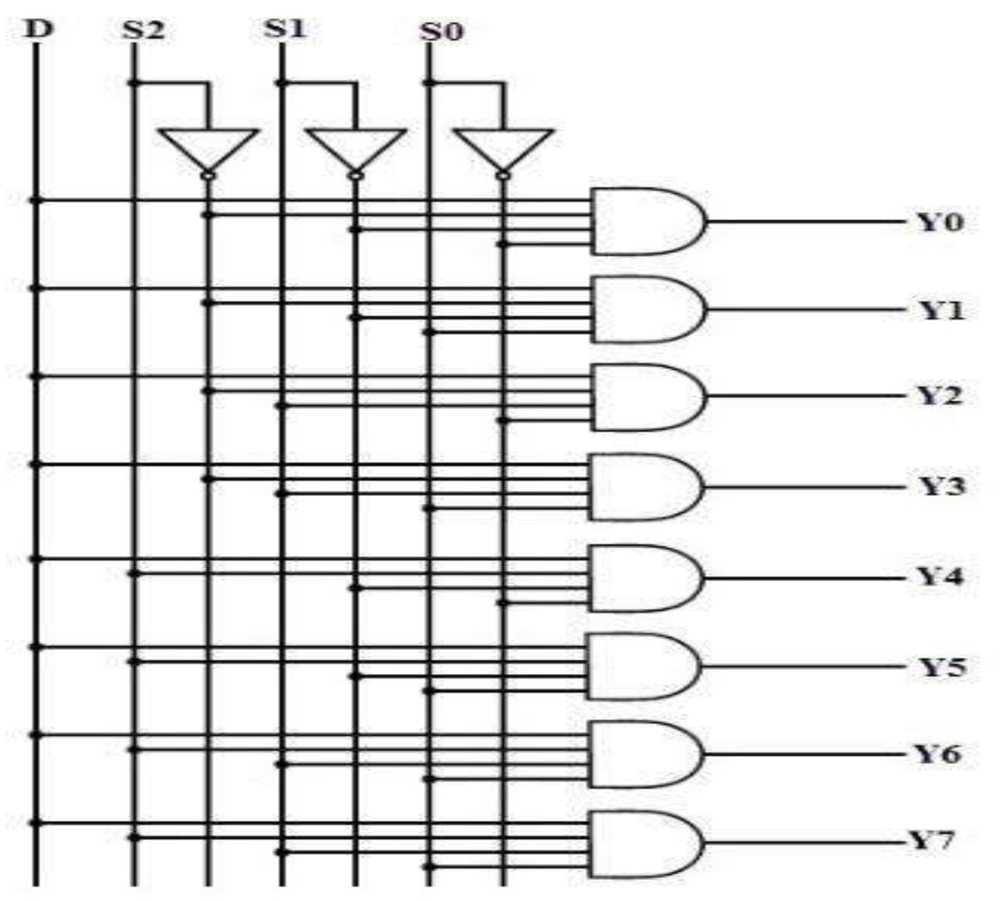


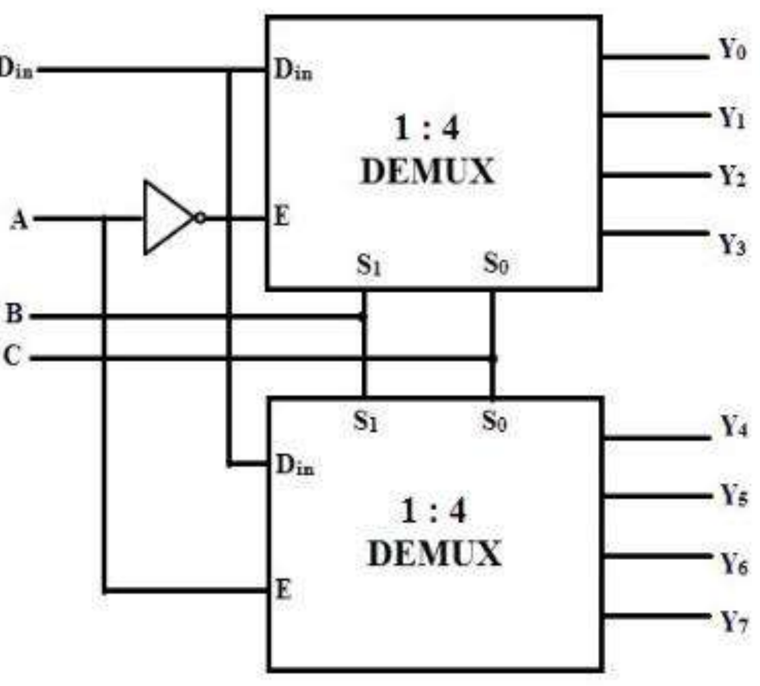
1:4 Demultiplexer

- The input bit is Data D with two select lines A and B.
- The input bit D is transmitted to four output bits Y0, Y1, Y2, and Y3.
- When AB is 00 the upper AND gate is enabled while the other AND gates are disabled. Thus, the data is transmitted to Y0.
- If D is low, then Y0 is low and if D is high, Y0 is high. The value of Y0 depends on the value of D.



1:8 Demultiplexer





Applications of Demultiplexer (Demux)

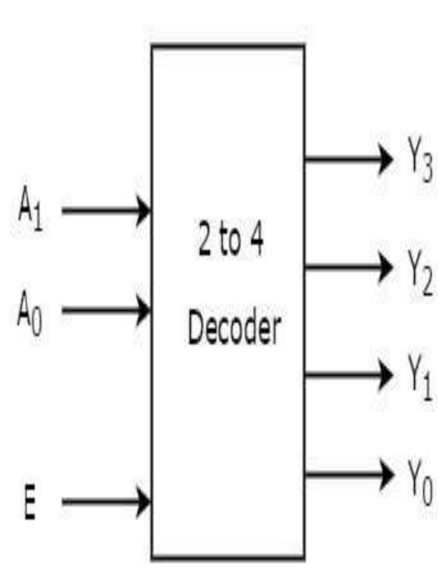
- Demux are widely used in microprocessor, computers and digital electronics.
- Demultiplexer and Multiplexer both are used in communication systems to carry multiple data signals (i.e. audio, video etc) using single line for transmission.
- In Arithmetic logic unit (ALU), the output of ALU can be stored in storage unit (multiple registers) by using Demultiplexer.

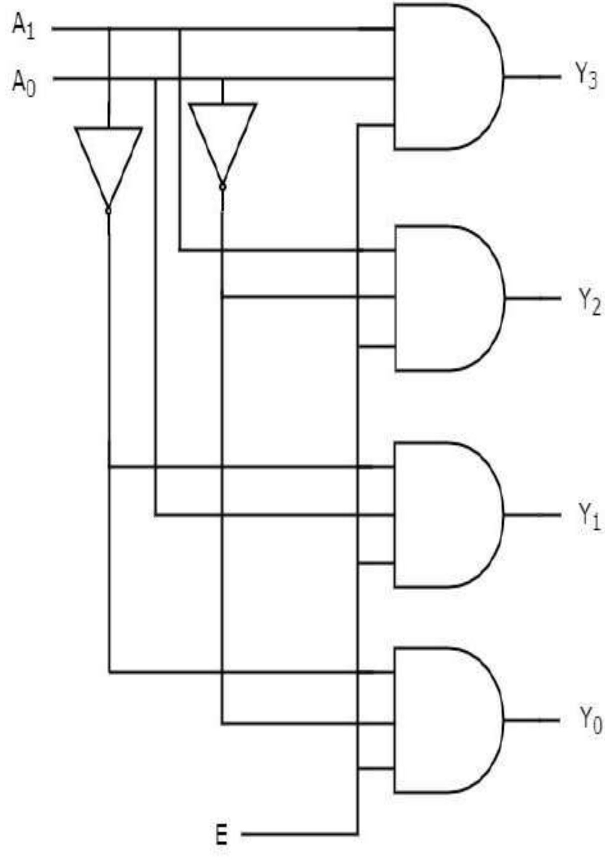
It is also used

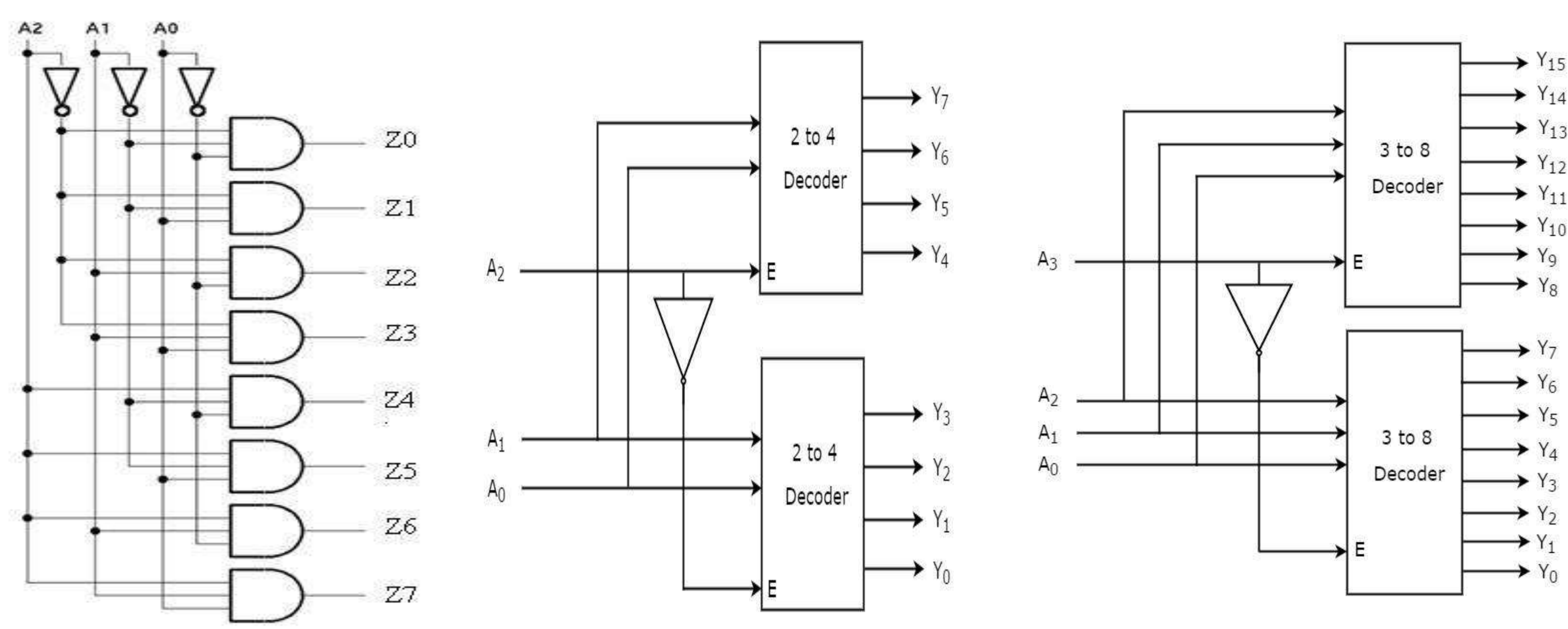
- To enable the different rows of memory chips depends on the address. Also to chose different banks of memory.
- To enable different functional unit in the system
- To select different IO devices for data transfer
- Data acquisition systems
- Automatic test equipment systems
- Security monitoring systems

Decoder

- **Decoder** is a combinational logic circuit whose purpose is to decode the information.
- It is comprised of n number of input lines and 2^n number of output lines.
- In every probable input condition, among the various output signals, only one output signal will produce the logic one.
- So, this is n-to-2^n decoder, where n input lines and 2^n output lines.
- Generally, there are 3 types of line decoders (2-to-4, 3-to-8 and 4-to-16).



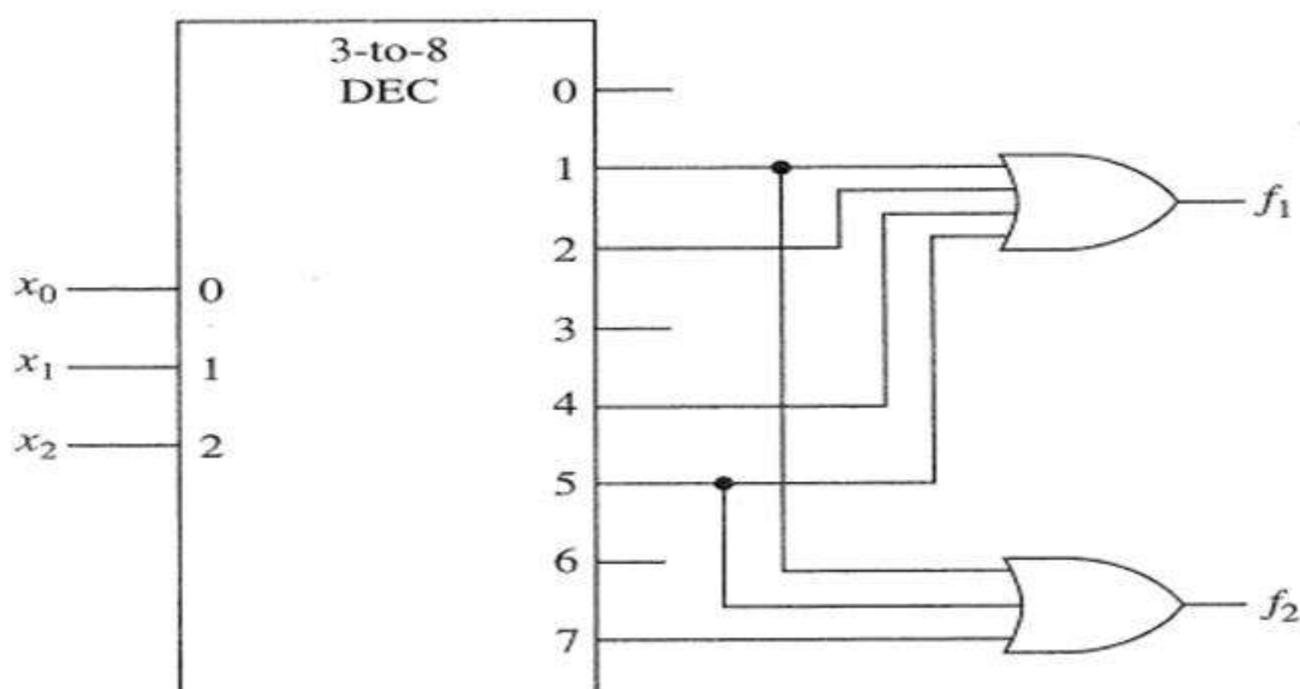



A2
A1
A0
Z0
Z1
Z2
Z3
Z4
Z5
Z6
Z7
2 to 4
Decoder
Y_7
Y_6
Y_5
Y_4
A_2
E
A_1
A_0
2 to 4
Decoder
Y_3
Y_2
Y_1
Y_0
E
3 to 8
Decoder
Y_{15}
Y_{14}
Y_{13}
Y_{12}
Y_{11}
Y_{10}
Y_9
Y_8
A_3
E
A_2
A_1
A_0
3 to 8
Decoder
Y_7
Y_6
Y_5
Y_4
Y_3
Y_2
Y_1
Y_0
E

Logic Design Using Decoders

- An n-to-2^n line decoder is a minterm generator.
- By using or-gates in conjunction with an n-to-2^n line decoder, realizations of Boolean functions are possible.
- Do not correspond to minimal sum-of-products.
- Are simple to produce. Particularly convenient when several functions of the same variable have to be realized.

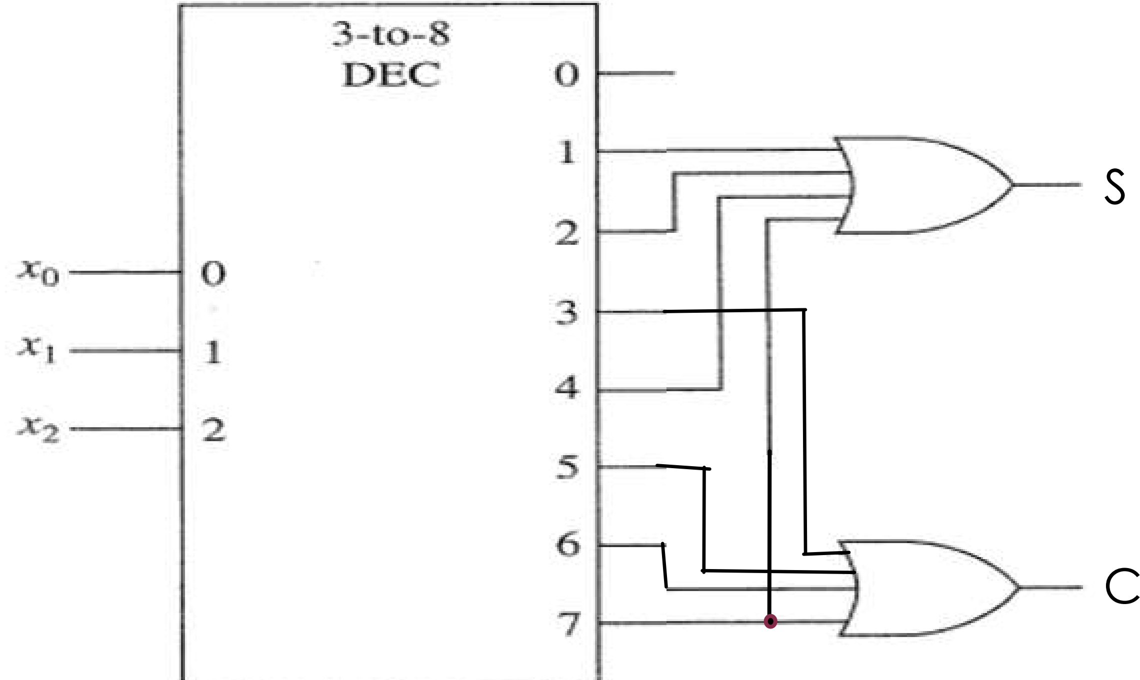
Realization of the Boolean expressions
$f_1(x_2, x_1, x_0) = \Sigma m(1,2,4,5)$ and
$f_2(x_2, x_1, x_0) = \Sigma m(1,5,7)$



Implementation of a Full Adder circuit using Decoder.

$S(x_0, x_1, x_2) = \sum(1, 2, 4, 7)$

$C(x_0, x_1, x_2) = \sum(3, 5, 6, 7)$

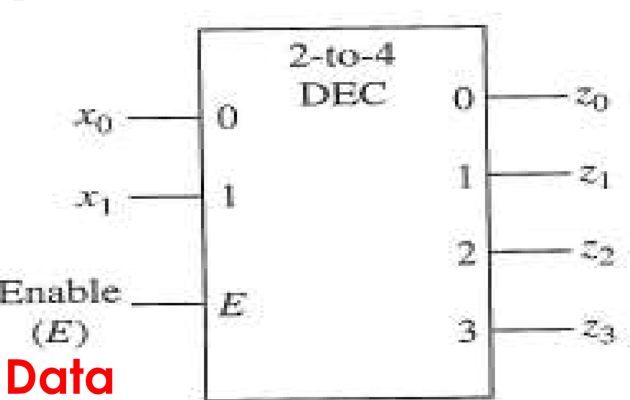


x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Decoders with enable inputs

- When disabled, all outputs of the decoder can either be at logic-0 or logic-1.
- Enable input provides the decoder with additional flexibility.
- Idea: if data is applied to the enable input.
- Process is known as demultiplexing.
- Now Decoder works as Demultiplexer.

$\overline{x}_0\overline{x}_1E$

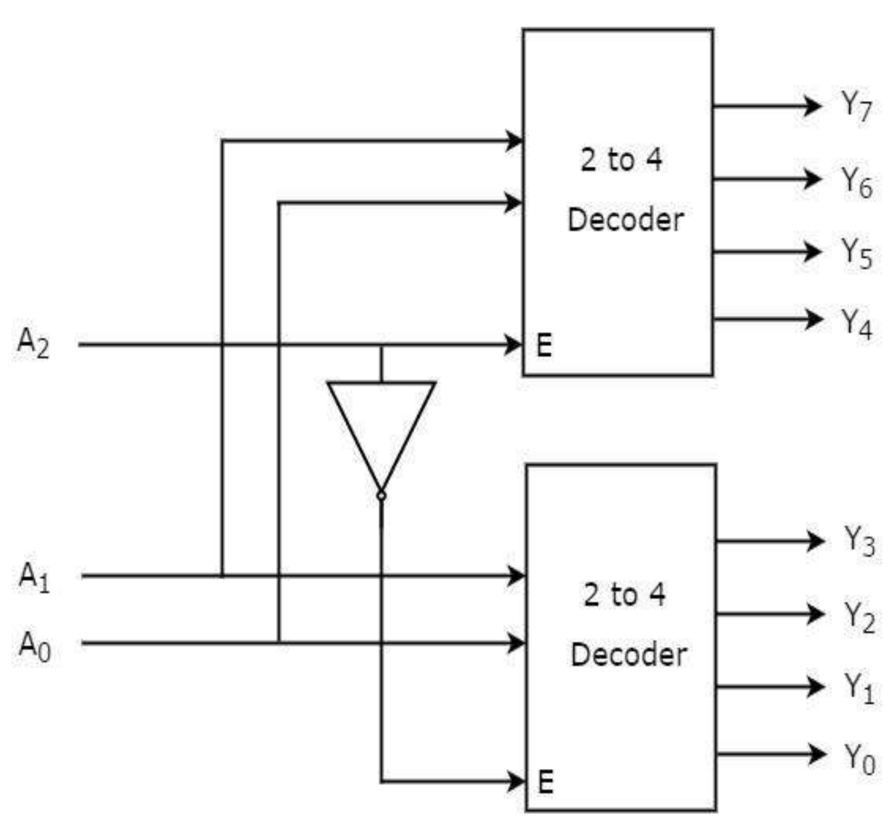


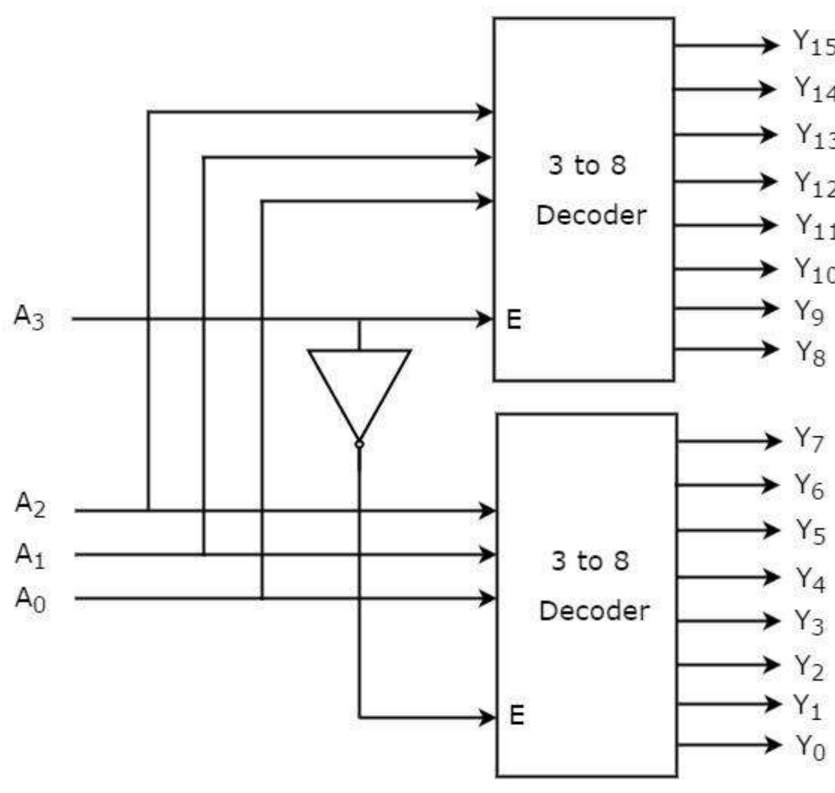
Data

If $x_0 = 0, x_1 = 0$ then data appears on line z_0.

- Enable inputs are useful when constructing larger decoders from smaller decoders.

Larger Decoders from smaller Decoder





Applications

- In digital electronic decoder play an important role. It is used to convert the data from one form to another form.
- Generally, these are frequently used in the communication systems like telecommunication, networking, and transfer the data from one end to the other end.
- In the same way it is also used in the digital domain for easy transmission of data.
- It is also used as

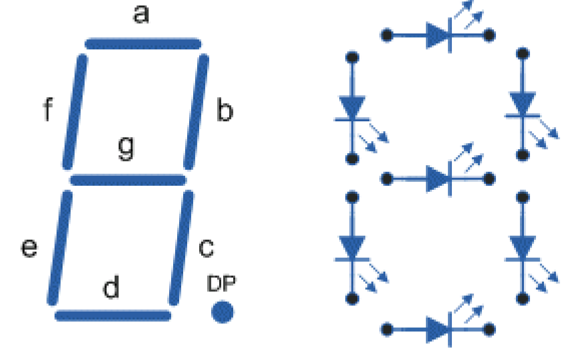
  Binary to Octal converter

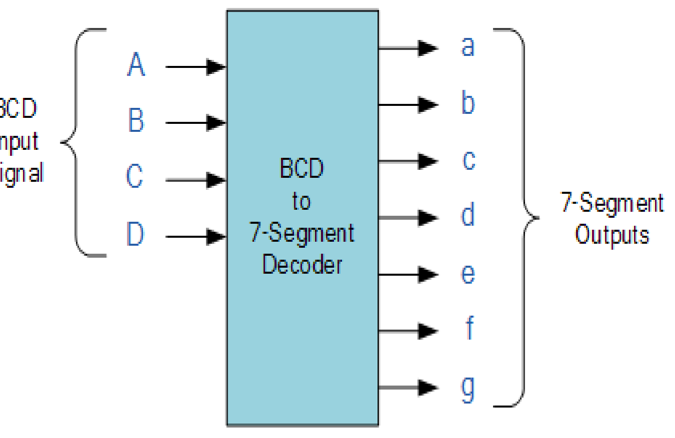
  BCD to Decimal converter

  BCD to Seven Segment Display
- Boolean functions can be implemented using decoder.

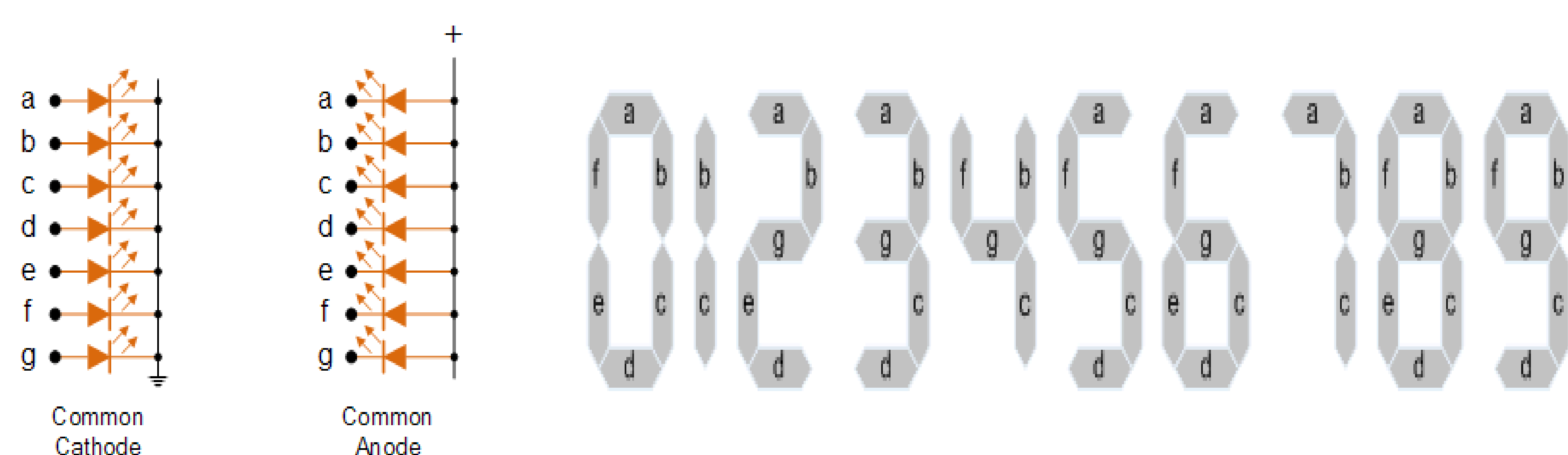
BCD to Seven segment display

- The Seven segment display is most frequently used the digital display in calculators, digital counters, digital clocks, measuring instruments, etc.
- Usually, the displays like LED's as well as LCD's are used to display the characters as well as numerical numbers.
- These displays are frequently driven by the output phases of digital integrated circuits like decade counters as well as latches.
- However, the outputs of these are in the type of 4-bit BCD (Binary Coded Decimal), so not appropriate for directly operating the seven segment display.
- For that, a display decoder can be employed for converting BCD code to seven segment code.
- Generally, it has four input lines as well as seven output lines.
- The Decoder is an essential component in BCD to seven segment display.



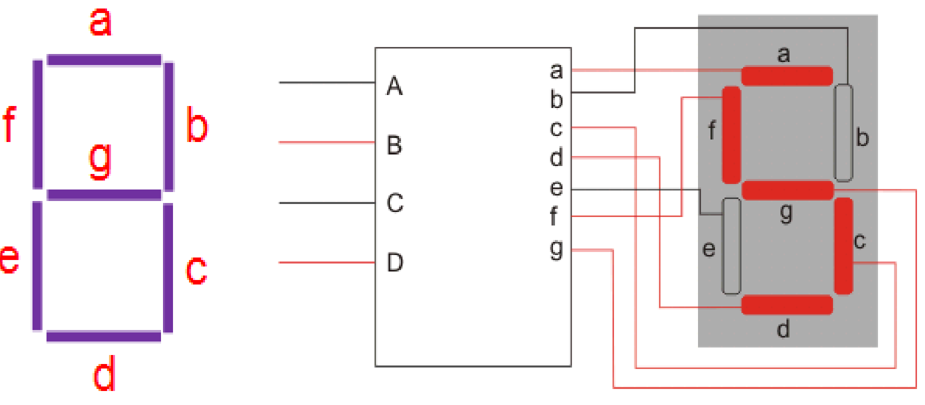


- The circuit design, as well as operation, mainly depends on the concepts of Boolean Algebra as well as logic gates.
- The common terminals are either anode or cathode. So, it may be common cathode type or common anode type.



Truth Table

Decimal Digit	Input lines				Output lines							Display pattern
	A	B	C	D	a	b	c	d	e	f	g	
0	0	0	0	0	1	1	1	1	1	1	0	0
1	0	0	0	1	0	1	1	0	0	0	0	1
2	0	0	1	0	1	1	0	1	1	0	1	2
3	0	0	1	1	1	1	1	1	0	0	1	3
4	0	1	0	0	0	1	1	0	0	1	1	4
5	0	1	0	1	1	0	1	1	0	1	1	5
6	0	1	1	0	1	0	1	1	1	1	1	6
7	0	1	1	1	1	1	1	0	0	0	0	7
8	1	0	0	0	1	1	1	1	1	1	1	8
9	1	0	0	1	1	1	1	1	0	1	1	9



$a = F1\ (A, B, C, D) = \sum m\ (0, 2, 3, 5, 6, 7, 8, 9)$

$b = F2\ (A, B, C, D) = \sum m\ (0, 1, 2, 3, 4, 7, 8, 9)$

$c = F3\ (A, B, C, D) = \sum m\ (0, 1, 3, 4, 5, 6, 7, 8, 9)$

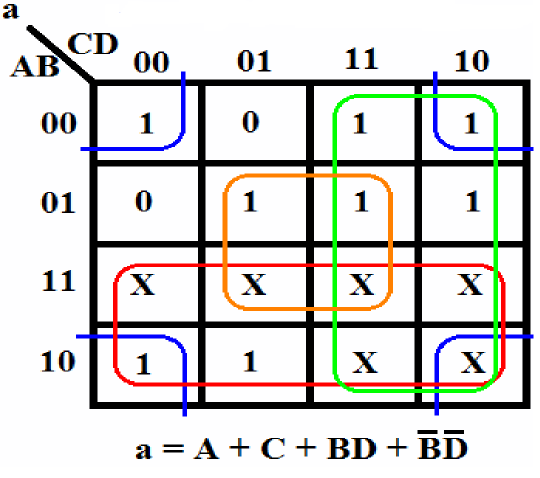
$d = F4\ (A, B, C, D) = \sum m\ (0, 2, 3, 5, 6, 8, 9)$

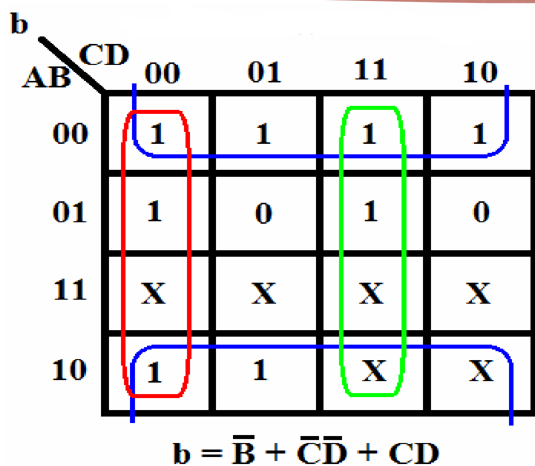
$e = F5\ (A, B, C, D) = \sum m\ (0, 2, 6, 8)$

$f = F6\ (A, B, C, D) = \sum m\ (0, 4, 5, 6, 8, 9)$

$g = F7\ (A, B, C, D) = \sum m\ (2, 3, 4, 5, 6, 8, 9)$

K-Map

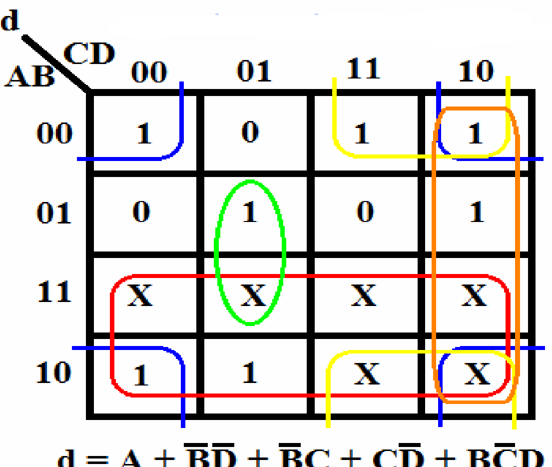


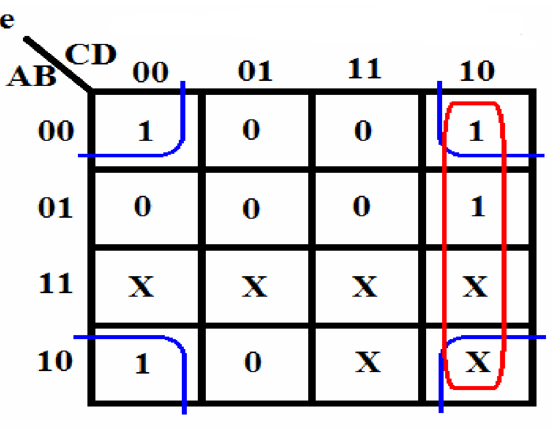


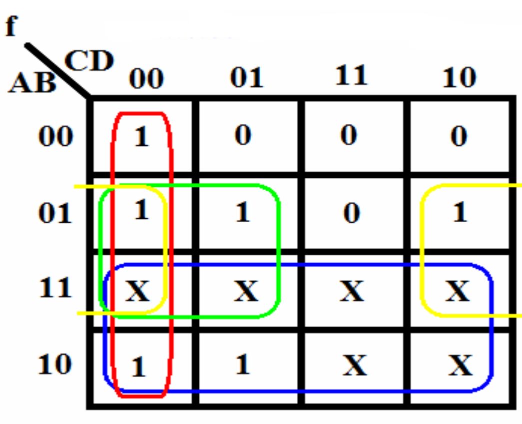
c

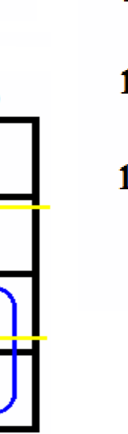
AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

$c = B + \overline{C} + D$







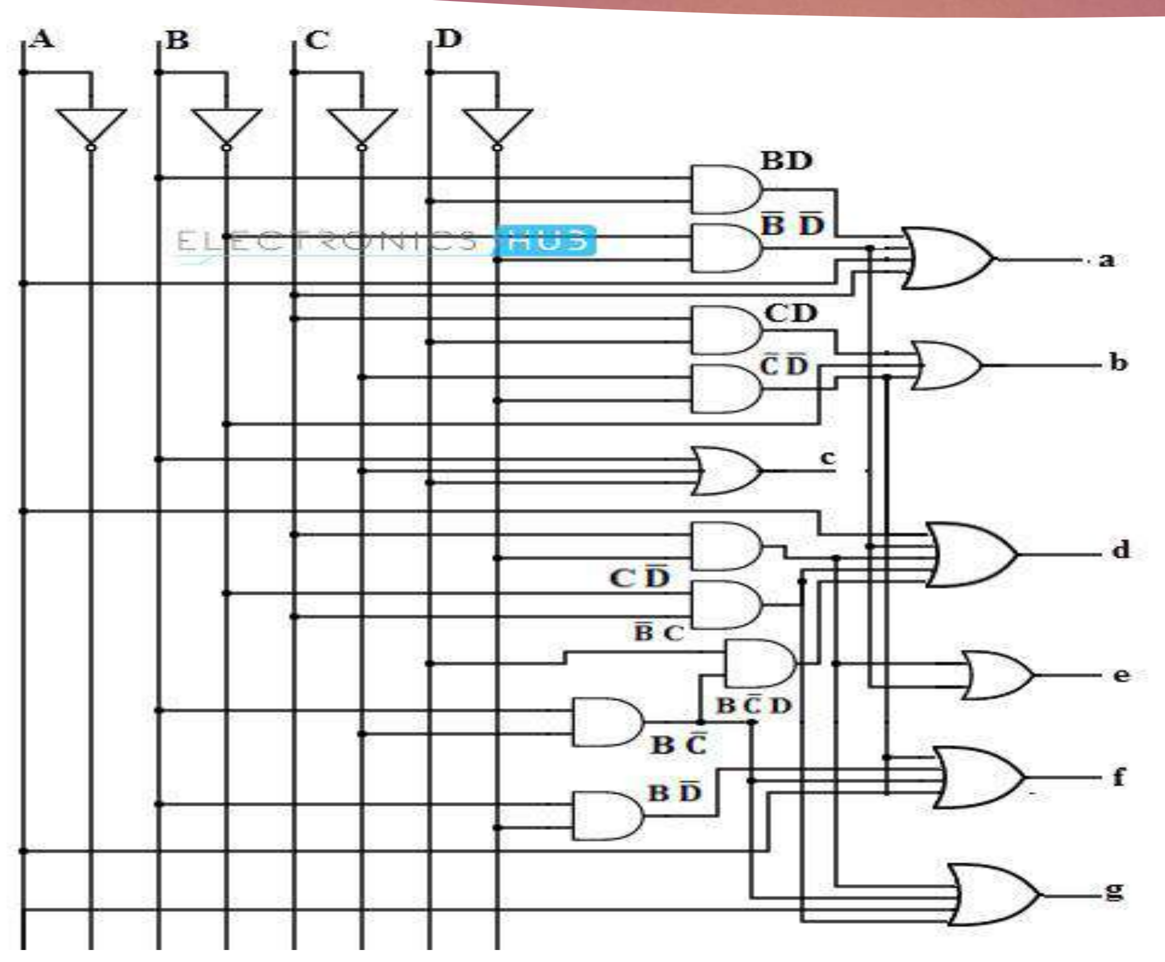


g

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

$g = A + B\overline{C} + \overline{B}C + C\overline{D}$

Logic Circuit



$$a = A + C + BD + \bar{B}\,\bar{D}$$

$$b = \bar{B} + \bar{C}\,\bar{D} + CD$$

$$c = B + \bar{C} + D$$

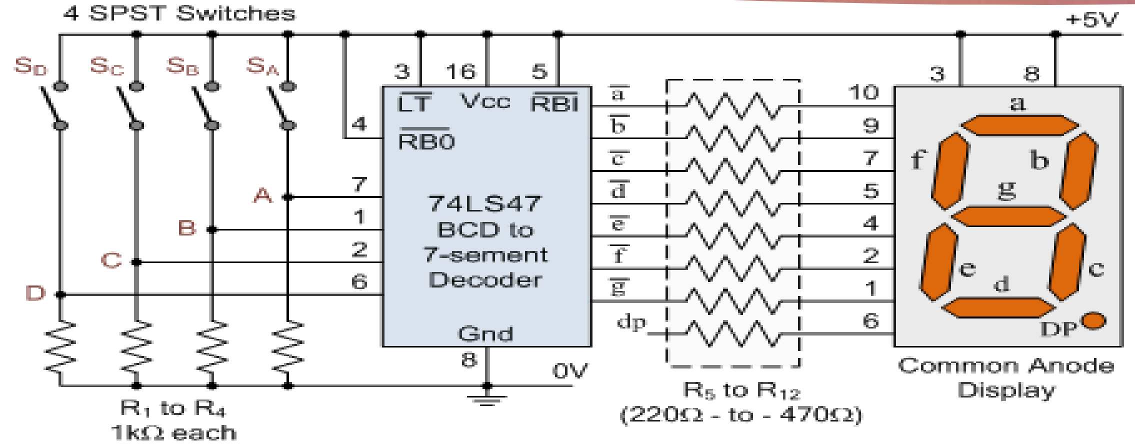
$$d = \bar{B}\,\bar{D} + C\,\bar{D} + B\,\bar{C}\,D + \bar{B}\,C + A$$

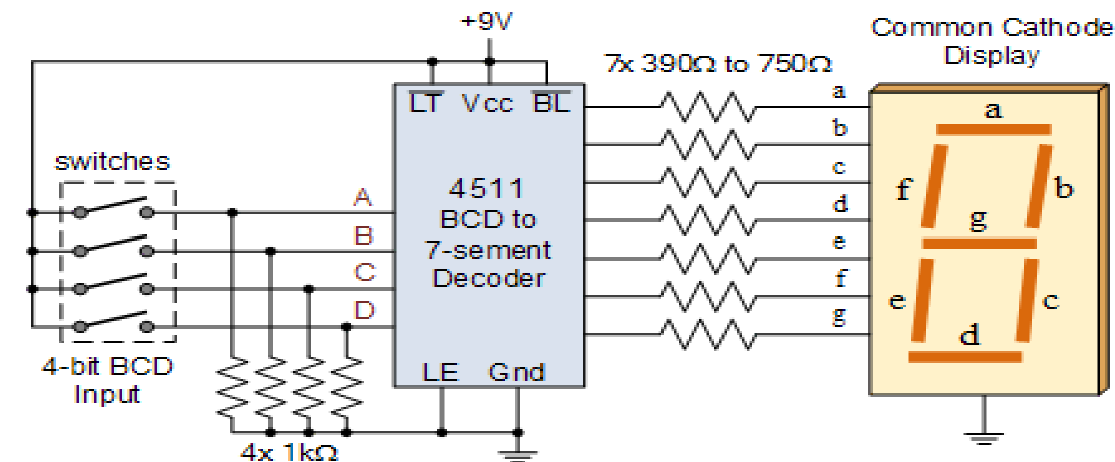
$$e = \bar{B}\,\bar{D} + C\,\bar{D}$$

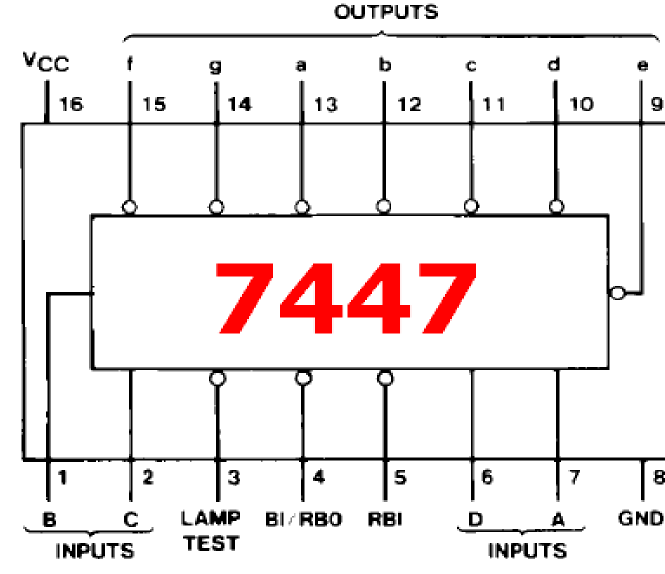
$$f = A + \bar{C}\,\bar{D} + B\,\bar{C} + B\,\bar{D}$$

$$g = A + B\,\bar{C} + \bar{B}\,C + C\,\bar{D}$$

IC and Connection Diagram

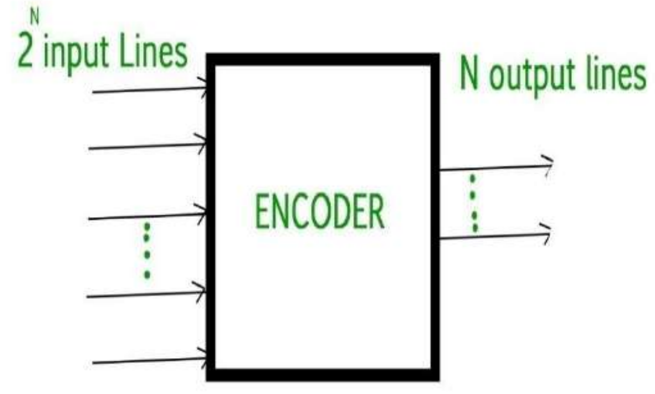




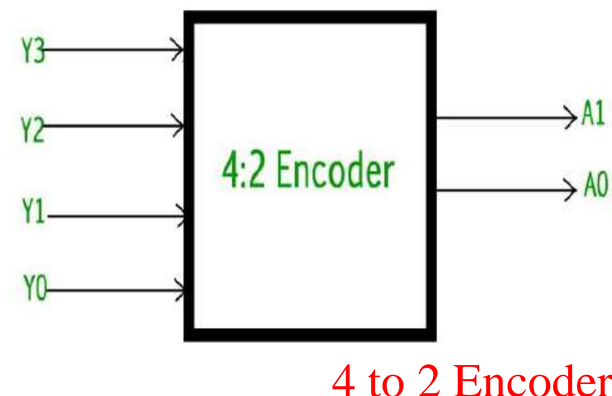


Encoder

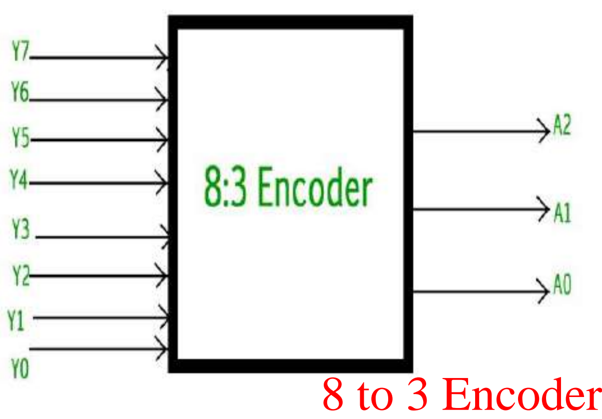
- An Encoder is a combinational circuit that performs the reverse operation of Decoder.
- It has maximum of 2^N **input lines** and **'N' output lines**, hence it encodes the information from 2^N inputs into an N-bit code.
- It will produce a binary code equivalent to the input.



- The 4 to 2 Encoder consists of **four inputs Y3, Y2, Y1, Y0 and two outputs A1 and A0**.
- At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output.
- The 8 to 3 Encoder or octal to Binary encoder consists of **8 inputs** : Y7 to Y0 and **3 outputs** : A2, A1 & A0.
- Each input line corresponds to each octal digit and three outputs generate corresponding binary code.



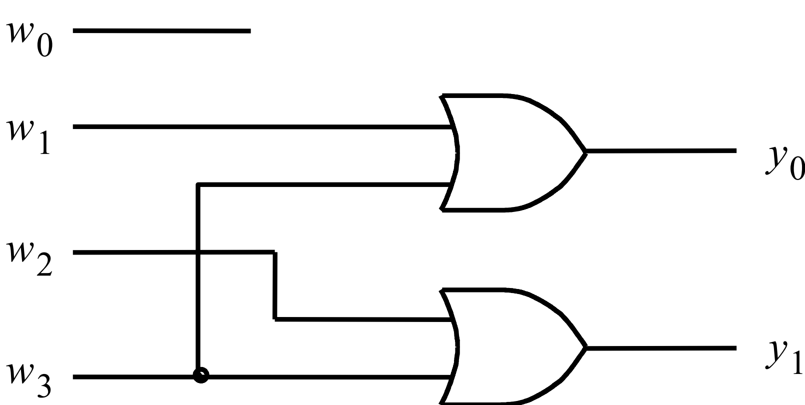
4 to 2 Encoder



8 to 3 Encoder

4-to-2 Binary Encoder

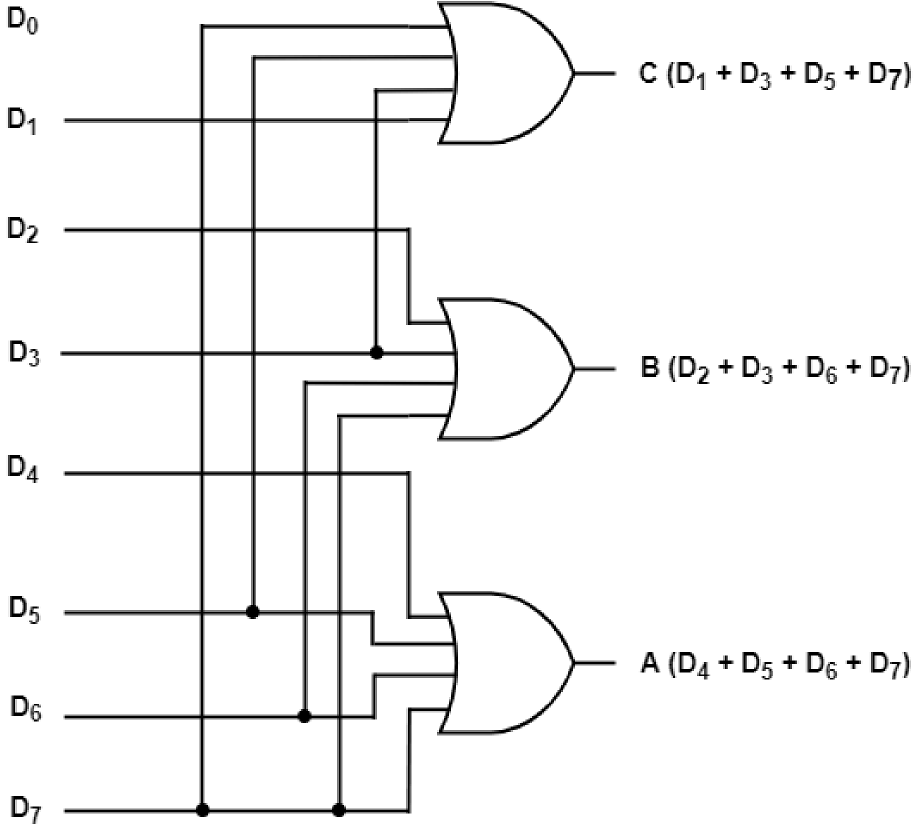
w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



8-to-3 Binary Encoder

At any one time, only
one input line has a value of 1.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
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



Priority Encoder

- One of the main disadvantages of standard digital encoder is that they can generate the wrong output code when there is more than one input present at logic level "1".
- One simple way to overcome this problem is to "Prioritize" the level of each input pin.
- If there is more than one input at logic level "1" at the same time, the actual output code would only correspond to the input with the highest designated priority.
- This type of digital encoder is known as **Priority Encoder** or **P-Encoder** for short.
- The **Priority Encoder** solves the problems by allocating a priority level to each input.
- The *priority encoders* output corresponds to the currently active input which has the highest priority.
- So, when an input with a higher priority is present, all other inputs with a lower priority will be ignored.

4-to-2 Priority Encoder

Truth Table

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

0 0 0 0

0 0 0 1

0 0 1 x

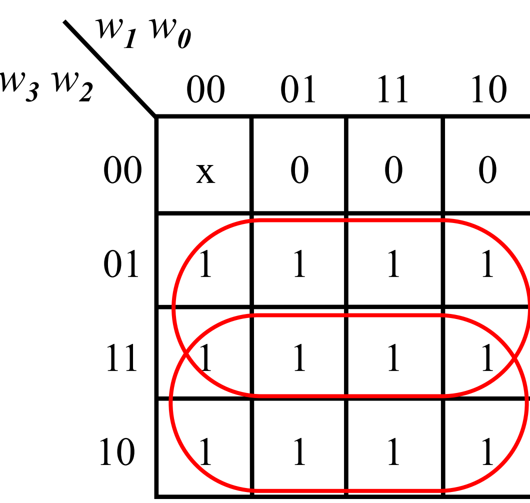
0 1 x x

1 x x x

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

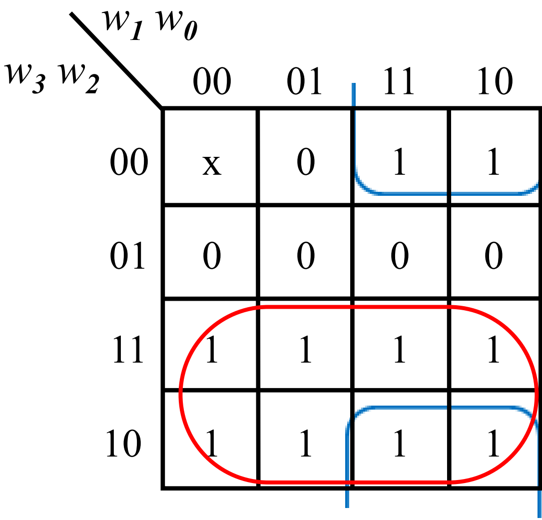
K-Map

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1



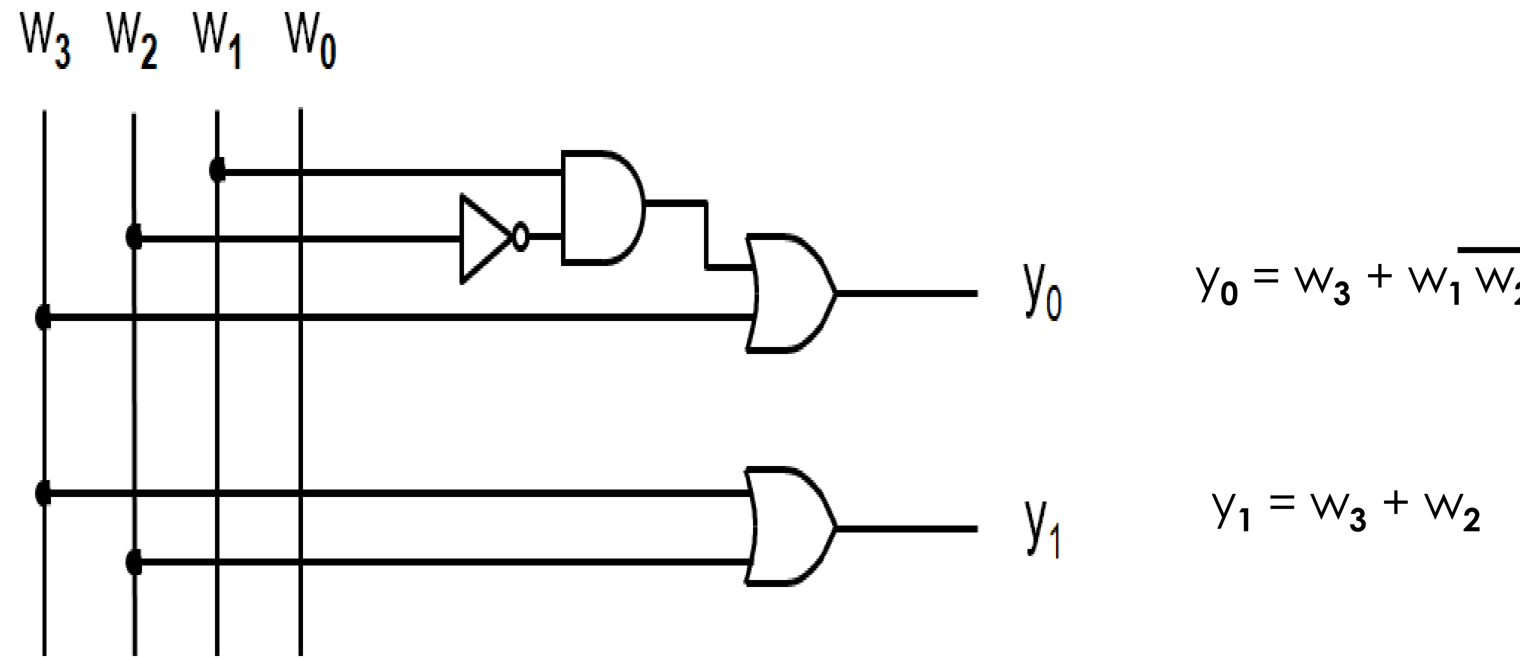
$y_1 = w_3 + w_2$

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1



$$y_0 = w_3 + w_1 \overline{w_2}$$

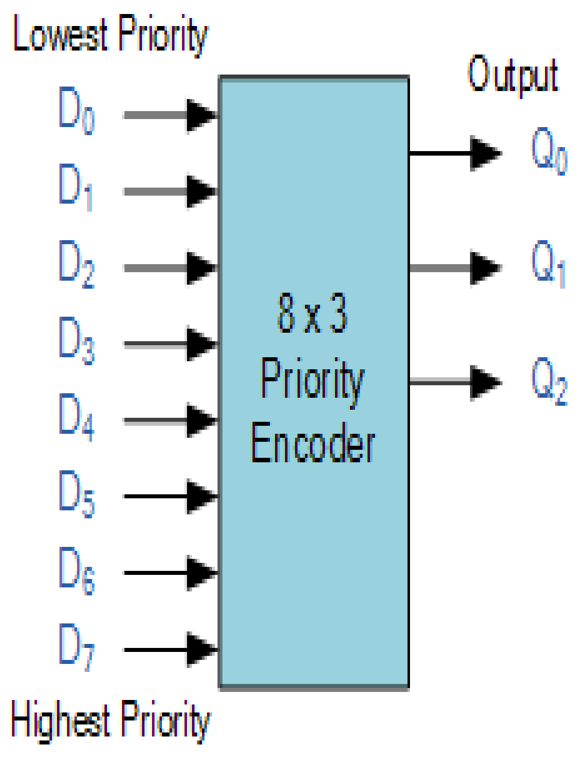
Circuit for the 4-to-2 priority encoder



$y_0 = w_3 + w_1 \overline{w_2}$

$y_1 = w_3 + w_2$

8-to-3 Priority Encoder



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

- From the truth table of the Priority Encoder, the Boolean expression with data inputs D_0 to D_7 and outputs Q_0, Q_1, Q_2 is given as:

$$Q_0 = \sum\left(\bar{D}_6\left(\bar{D}_4\bar{D}_2 D_1 + \bar{D}_4 D_3 + D_5\right) + D_7\right)$$

$$Q_1 = \sum\left(\bar{D}_5\bar{D}_4\left(D_2 + D_3\right) + D_6 + D_7\right)$$

$$Q_2 = \sum\left(D_4 + D_5 + D_6 + D_7\right)$$

Applications

- Keyboard Encoder
- Interrupt Requests
- Octal to Binary Encoder
- Decimal to Binary Encoder
- Decimal to BCD Encoder

Lecture of Module 4

Sequential Circuits
(Latch/Flip Flop)

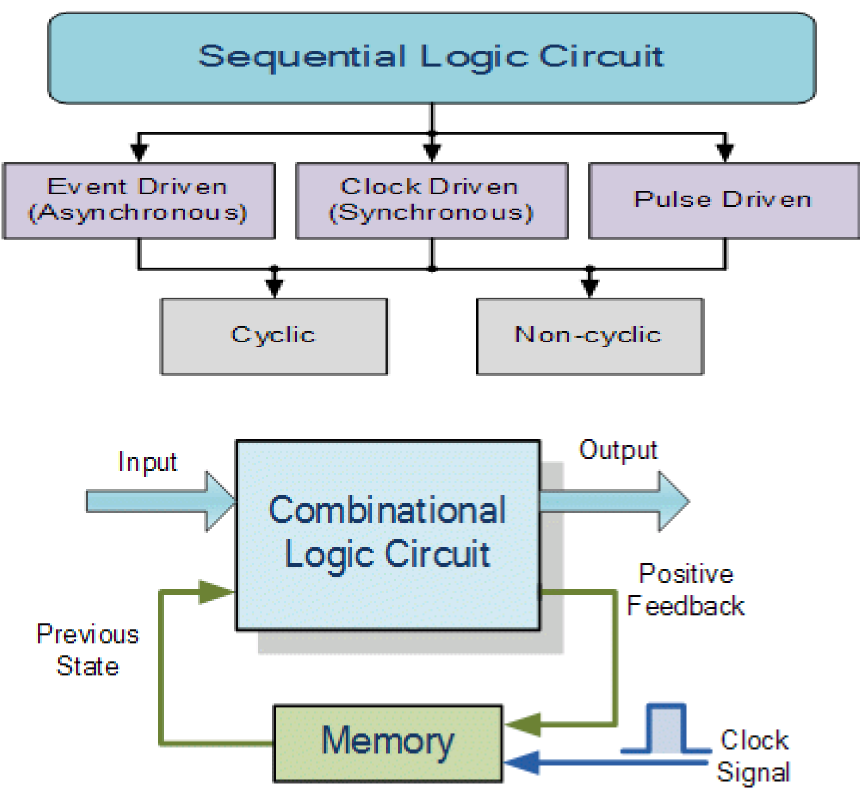
Overview

- **Introduction**
- **Latch**
- **Flip Flops**
- **Triggering of Flip Flop**
- **SR, D, JK, T Flip Flop**
- **Master Slave Flip Flop**
- **Characteristic Table and Characteristic Equation**
- **Excitation Table**

Sequential Logic Circuits

The output state of a "sequential logic circuit" is a function of the following three states, the "present input", the "past input" and/or the "past output". *Sequential Logic circuits* remember these conditions and stay fixed in their current state until the next clock signal changes.

Sequential logic circuits are generally termed as *two state* or Bistable devices. Outputs set in one of two basic states, a logic level "1" or a logic level "0" and will remain "latched" (hence the name is latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.



Sequential Logic: Concept

- Sequential Logic circuits remember past inputs and past circuit state.
- Outputs from the system are “fed back” as new inputs.
- The storage elements are circuits that are capable of storing binary information: Memory.

The basic sequential circuit elements can be divided in two categories:

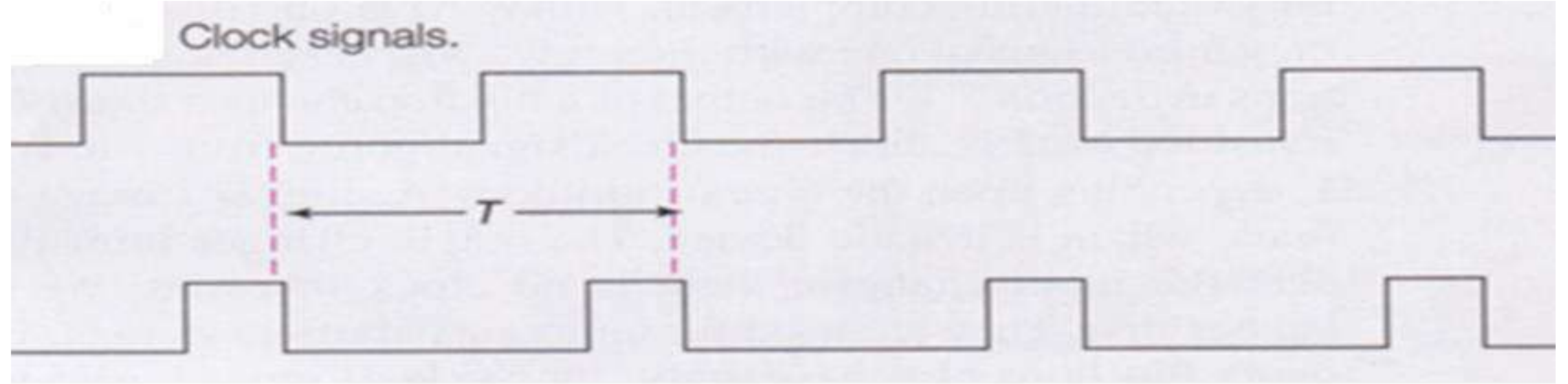
- Level-sensitive (Latches)
  - High-level sensitive
  - Low-level sensitive
- Edge-triggered (Flip-flops)
  - Rising (positive) edge triggered
  - Falling (negative) edge triggered
  - Dual-edge triggered

Clock

Sequential circuits can be Asynchronous or Synchronous.

Asynchronous sequential circuits change their states and output values whenever a change in input values occurs. Circuit output can change at **any** time (clock less).

Synchronous sequential circuits change their states and output values at fixed points of time. This type of circuits achieves synchronization by using a timing signal called the *clock*.



Clock generator: Periodic train of clock pulses

Memory Devices

Latches: A *latch* is a memory element whose excitation signals control the state of the device. A latch has two stages *set* and *reset*. *Set* stage sets the output to 1. *Reset* stage set the output to 0.

Latches are also called **level triggered** flip flops, because the change on the outputs will follow the changes of the inputs as long as the Enable input is set.

- This causes synchronization problems.

Solution: use latches to create flip-flops that can respond (update) only on specific times (instead of any time).

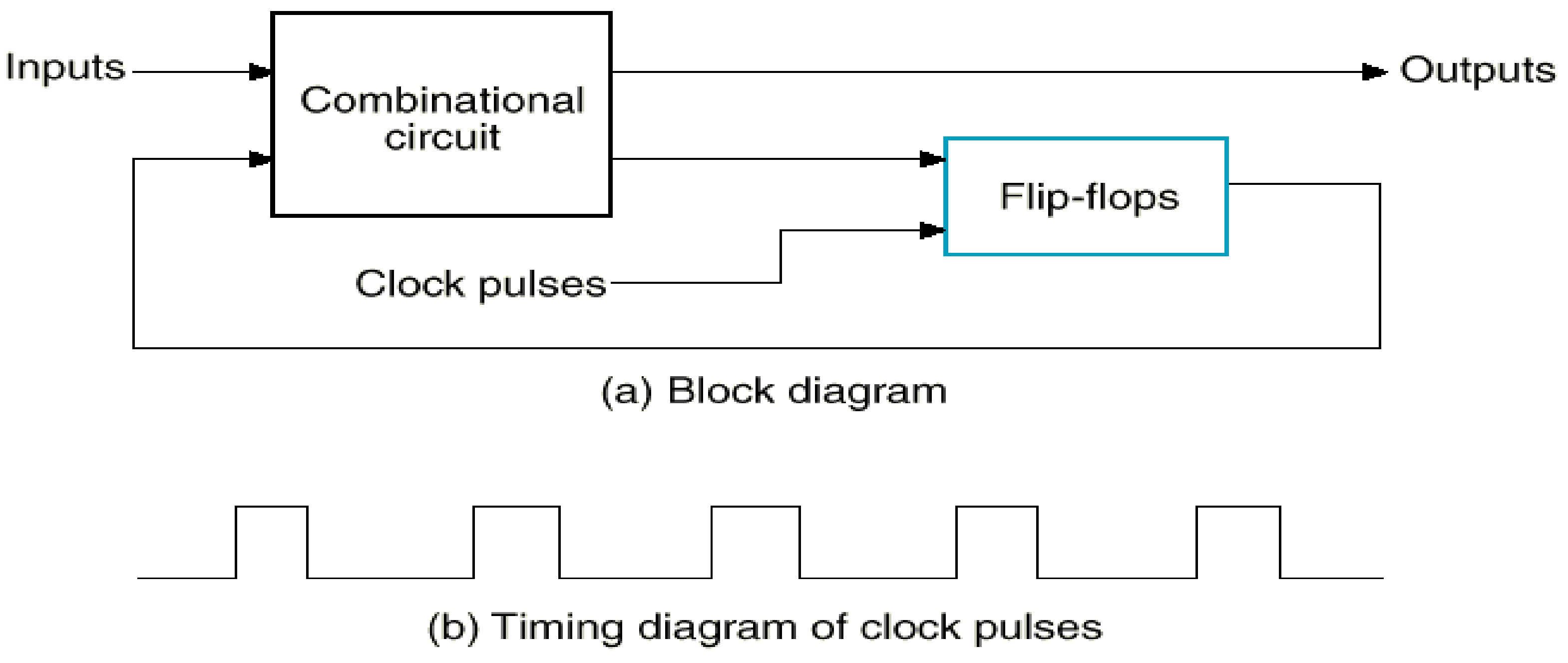
Flip-flops: A *flip-flop* is a memory device that has clock signals control the state of the device.

Flip Flops are **Edge triggered** that change there outputs only at the transition of the clock signal.

Latch Vs. Flip Flop

Latches	Flip Flops
Latches are building blocks of sequential circuits and these can be built from logic gates	Flip flops are also building blocks oof sequential circuits. But, these can be built from the latches.
Latch continuously checks its inputs and changes its output correspondingly.	Flip flop continuously checks its inputs and changes its output correspondingly only at times determined by clocking signal
The latch is sensitive to the duration of the pulse and can send or receive the data when the switch is on	Flipflop is sensitive to a signal change. They can transfer data only at the single instant and data cannot be changed until next signal change. Flip flops are used as a register.
It is based on the enable function input	It works on the basis of clock pulses
It is a level triggered, it means that the output of the present state and input of the next state depends on the level that is binary input 1 or 0.	It is an edge triggered, it means that the output and the next state input changes when there is a change in clock pulse whether it may a +ve or -ve clock pulse.

Synchronous Sequential Circuits: Flip flops as state memory

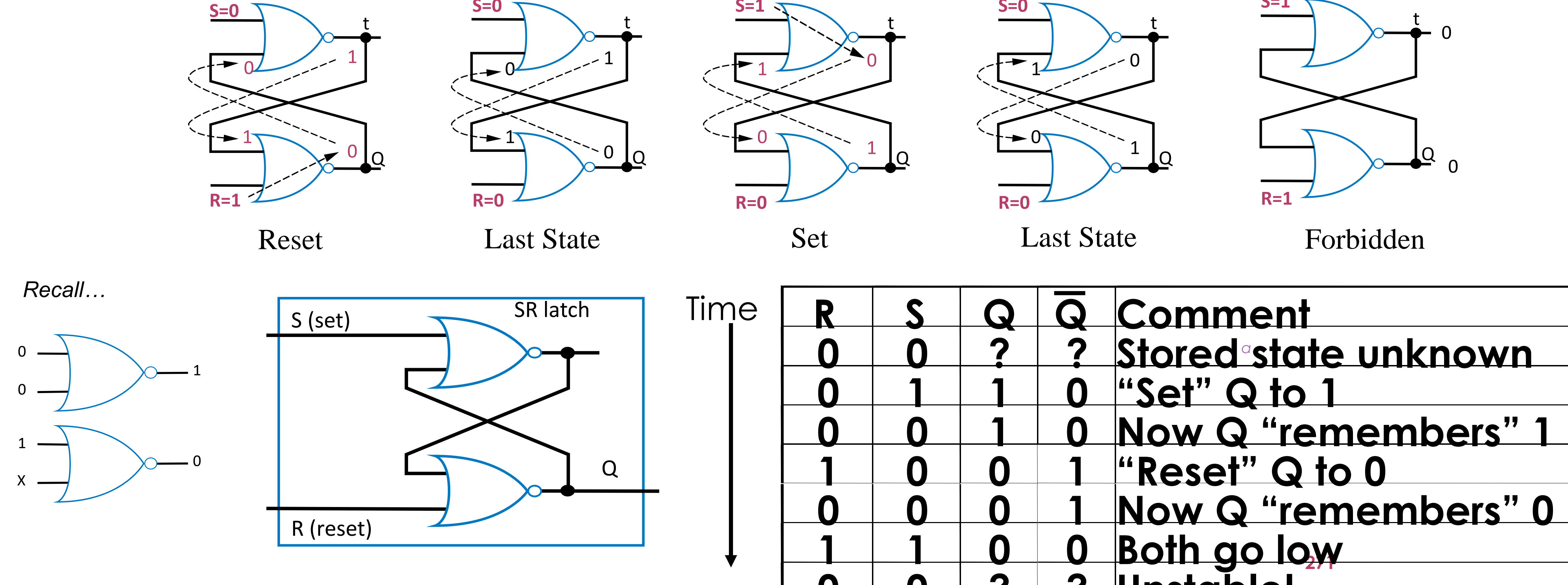


(a) Block diagram

(b) Timing diagram of clock pulses

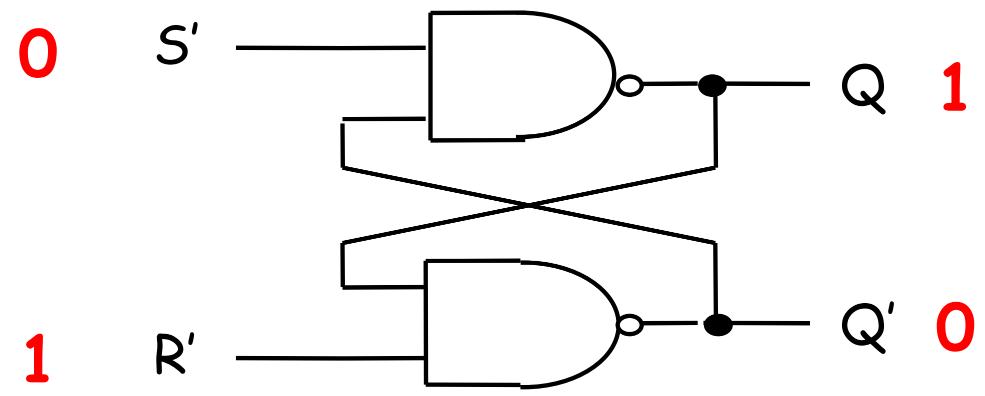
- The flip-flops receive their inputs from the combinational circuit and also from a clock signal with pulses that occur at fixed intervals of time, as shown in the timing diagram.

S-R Latch(NOR version)



R	S	Q	$\overline{Q}$	Comment
0	0	?	?	Stored state unknown
0	1	1	0	“Set” Q to 1
0	0	1	0	Now Q “remembers” 1
1	0	0	1	“Reset” Q to 0
0	0	0	1	Now Q “remembers” 0
1	1	0	0	Both go low
0	0	?	?	Unstable!

S-R Latch(NAND version)



S'	R'	Q	Q'	
0	0			
0	1	1	0	Set
1	0			
1	1			

Characteristic Table

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

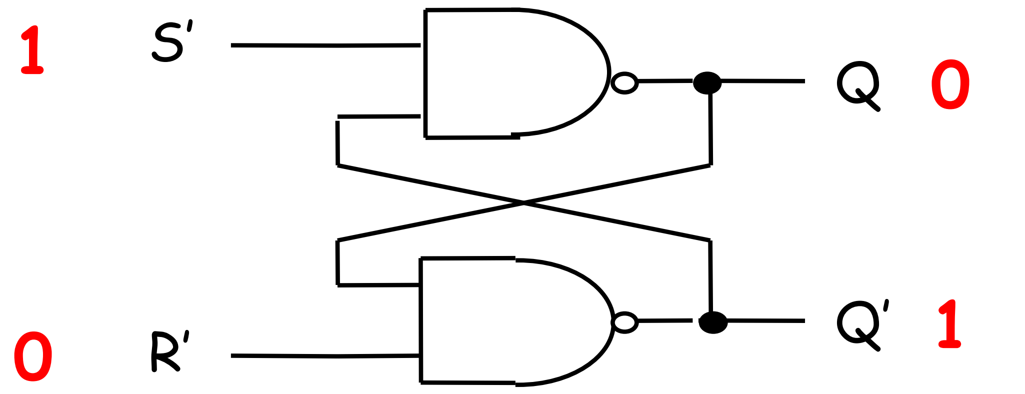
S-R Latch(NAND version)

S'	R'	Q	Q'	
0	0			
0	1	1	0	Set
1	0			
1	1	1	0	Last State

Characteristic Table

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

S-R Latch(NAND version)

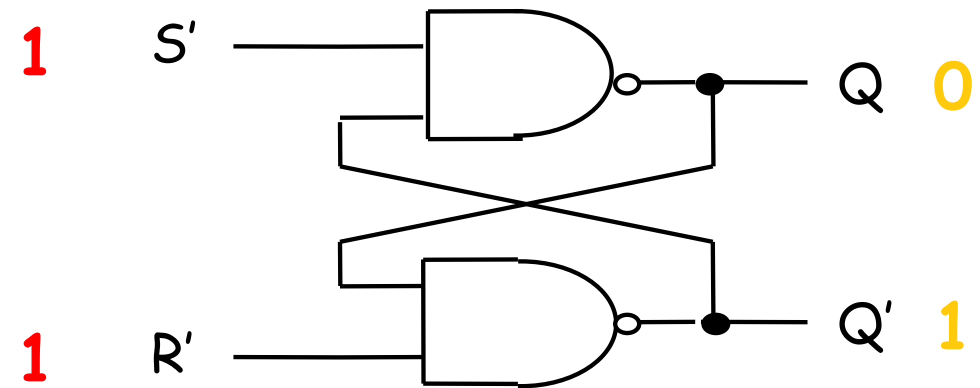


S'	R'	Q	Q'	
0	0			
0	1	1	0	Set
1	0	0	1	Reset
1	1	1	0	Last State

Characteristic Table

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

S-R Latch Flop(NAND version)

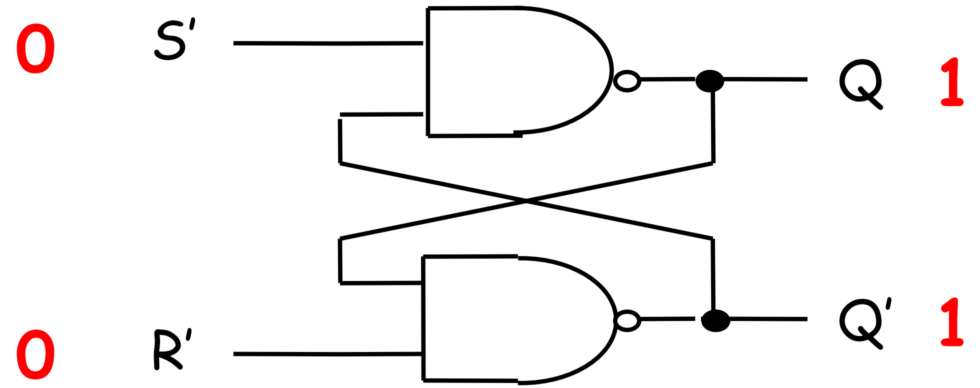


S′	R′	Q	Q′	
0	0			
0	1	1	0	Set
1	0	0	1	Reset
1	1	1	0	Last State
		0	1	Last State

Characteristic Table

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

S-R Latch(NAND version)



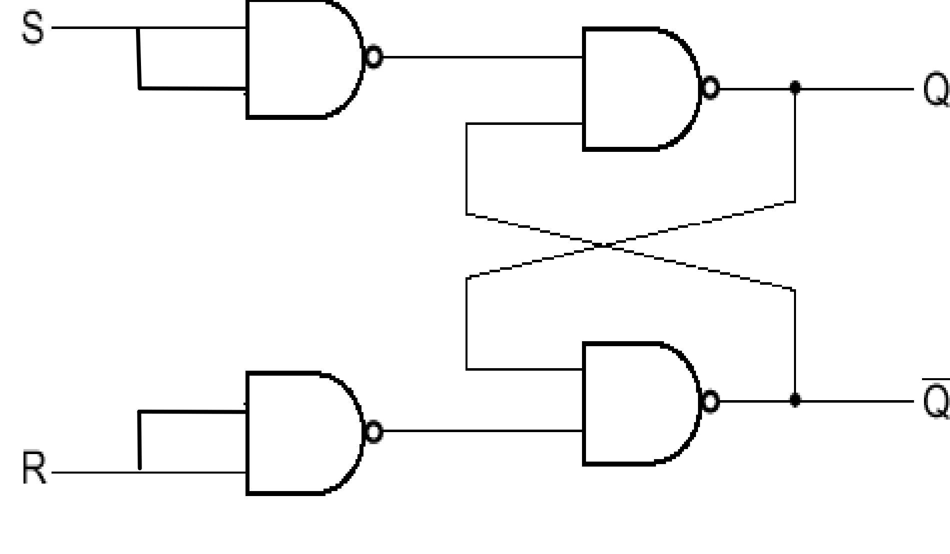
S'	R'	Q	Q'	
0	0	1	1	Forbidden
0	1	1	0	Set
1	0	0	1	Reset
1	1	1	0	Last State
		0	1	Last State

Characteristic Table

X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

S	R	Next state of Q
0	0	No change
0	1	Q = 0; Reset state
1	0	Q = 1; Set state
1	1	Undefined

S-R Latch(NAND version)

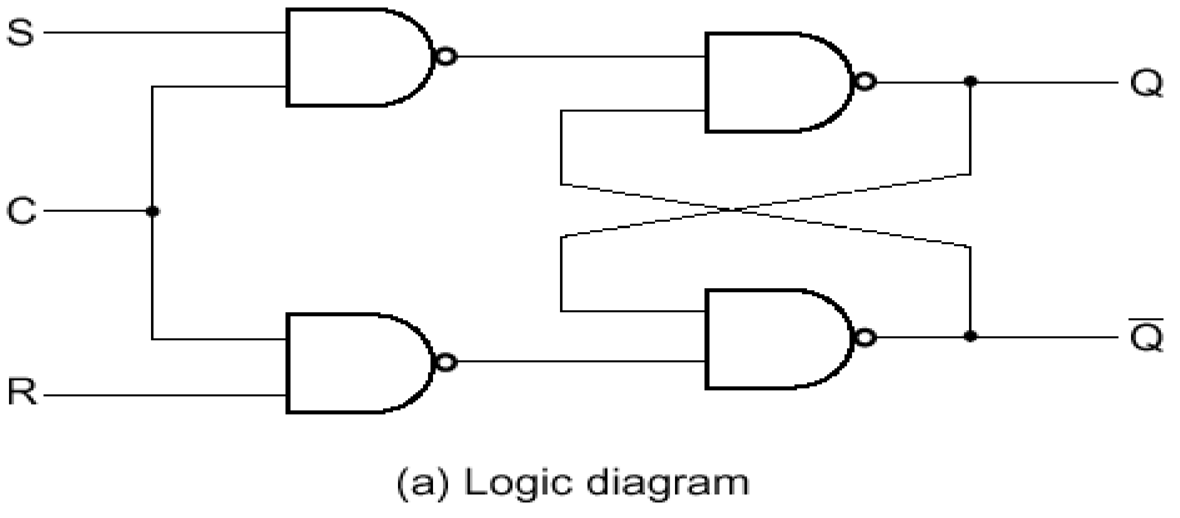


Logic diagram

S	R	Next state of Q
0	0	No change
0	1	Q = 0; Reset state
1	0	Q = 1; Set state
1	1	Undefined

Function table

S-R Flip Flop with Clock signal



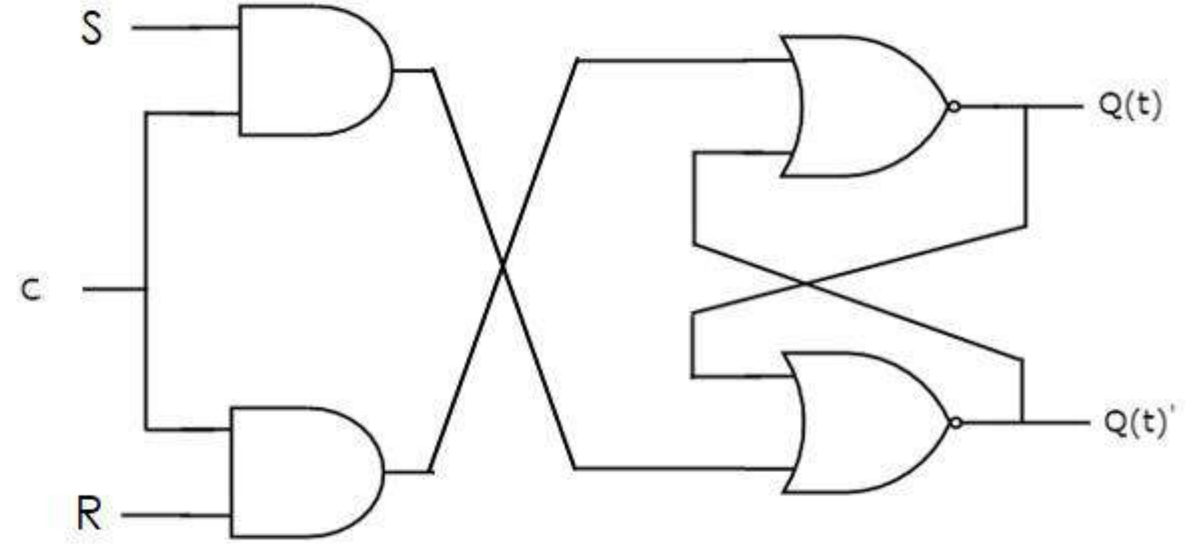
(a) Logic diagram

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	Undefined

(b) Function table

Latch is sensitive to input changes ONLY when C=1

S-R Flip Flop with Clock signal



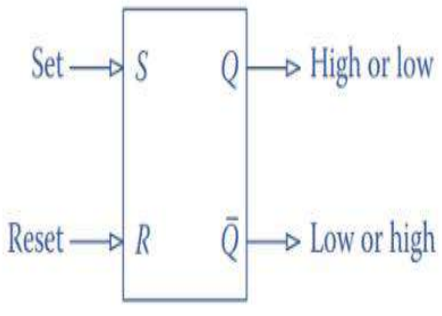
Logic diagram

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	Undefined

Function table

Latch is sensitive to input changes ONLY when C=1

Characteristic Equation of S-R Flip Flop

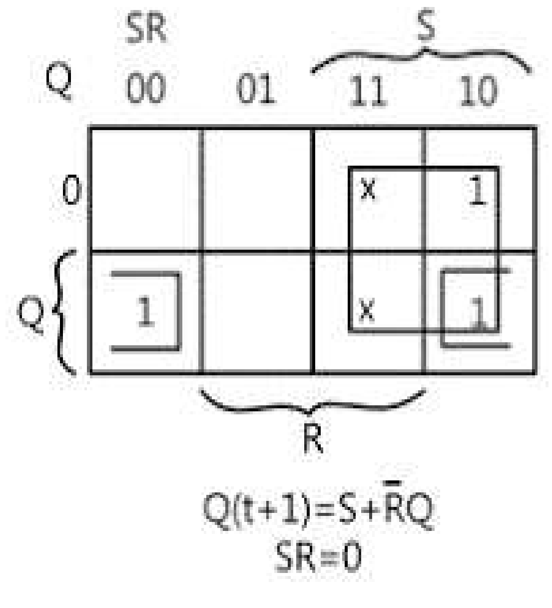


R	S	Q	$\bar{Q}$
0	0	Q_0	$\bar{Q}_0$
0	1	1	0
1	0	0	1
1	1	Not allowed	

Q_0 and $\bar{Q}_0$ are initial conditions (not defined)

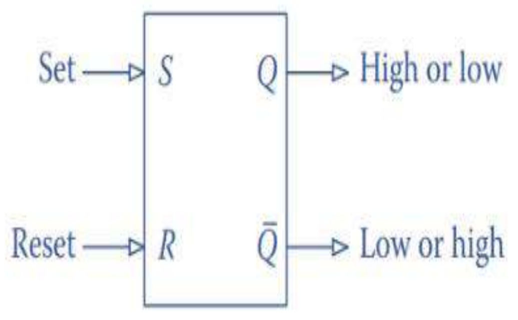
Q	S	R	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Negative Status
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Negative Status

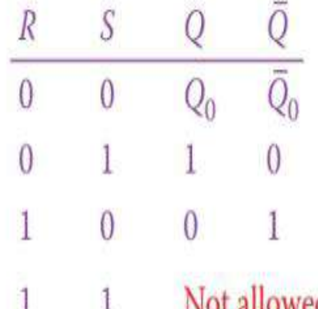
Characteristic Table



Characteristic Equation

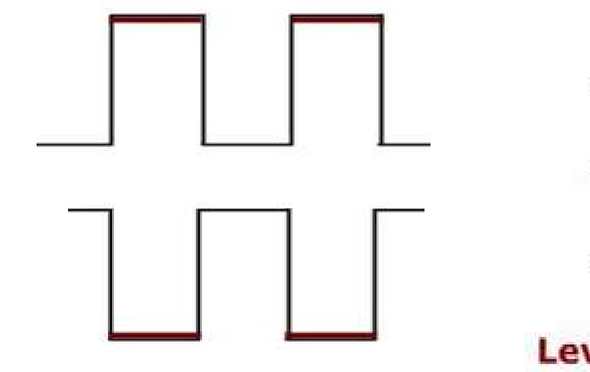
Triggering of Flip Flop

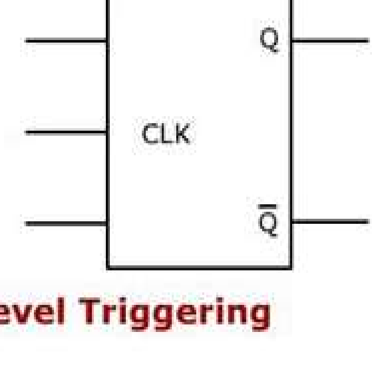


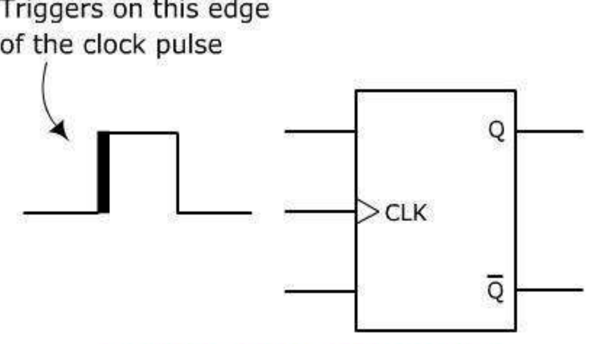


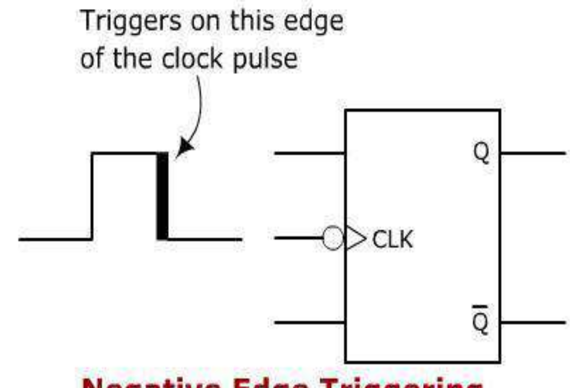
R	S	Q	$\bar{Q}$
0	0	Q_0	$\bar{Q}_0$
0	1	1	0
1	0	0	1
1	1	Not allowed	

Q_0 and $\bar{Q}_0$ are initial conditions (not defined)



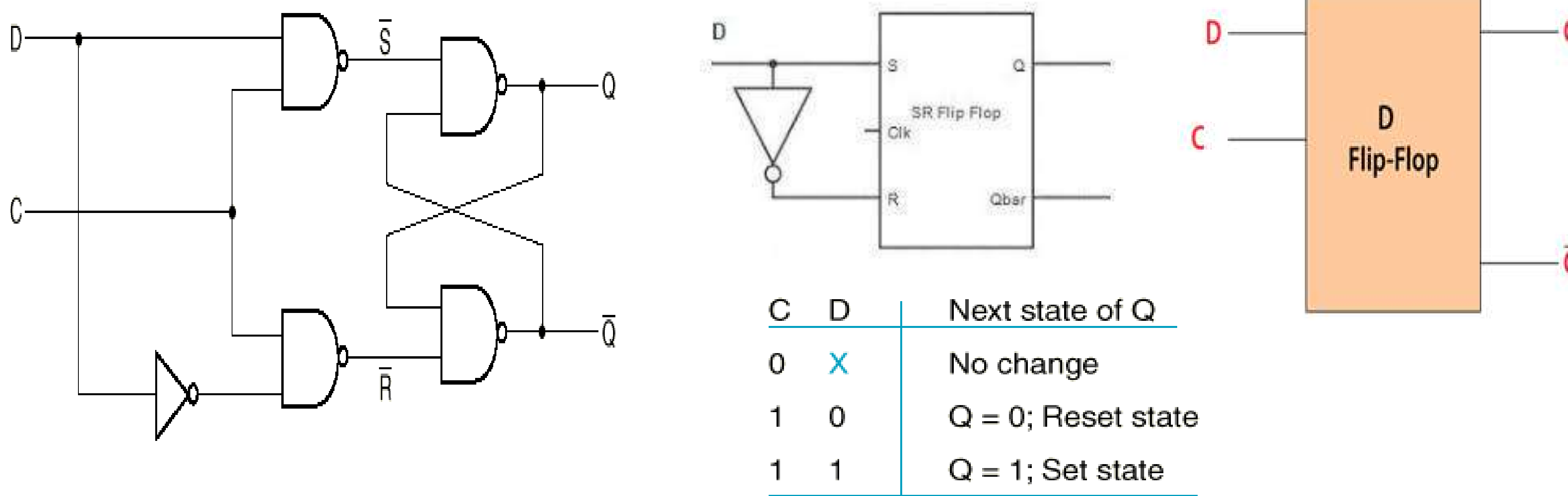






D Flip Flop

- One way to eliminate the undesirable indeterminate state in the RS flip flop is to ensure that inputs S and R are never 1 simultaneously.

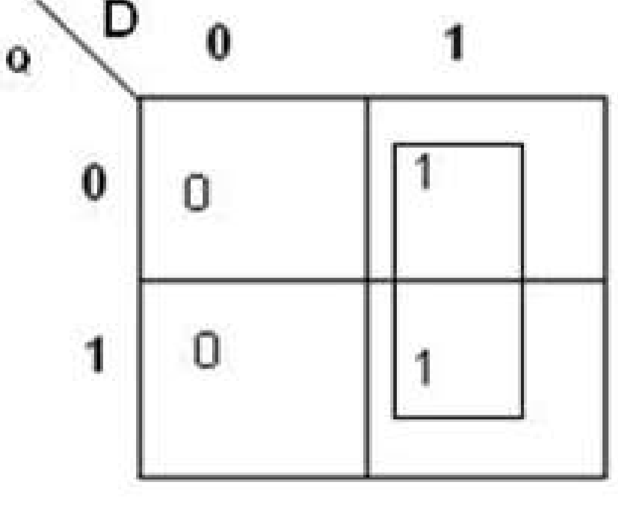


C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

Characteristic Equation of D Flip Flop

Characteristics table

Qn	D	Q(n+1)
0	0	0
0	1	1
1	0	0
1	1	1

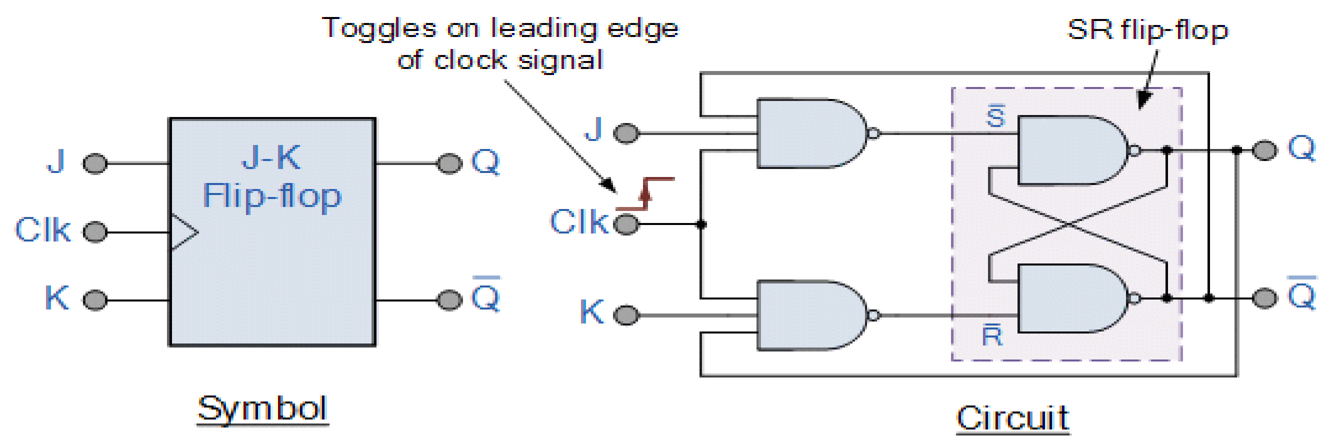


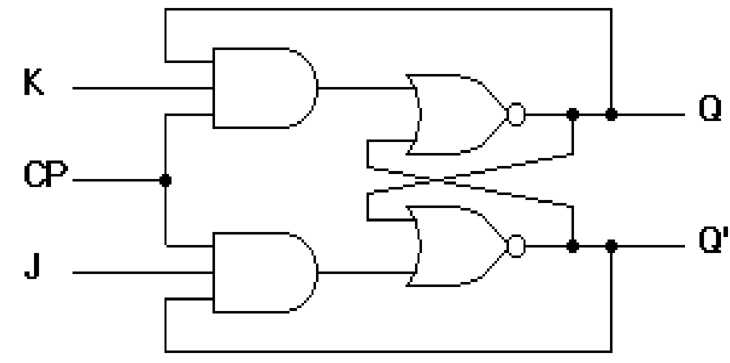
$$Q\,(n+1) = D$$

Characteristic Equation

J-K Flip Flop

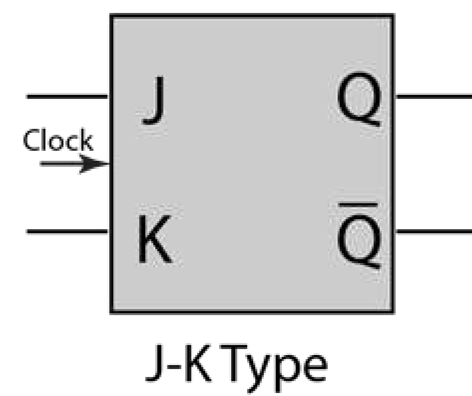
- In SR Flip Flop S=R=1 should be avoided.
- To overcome that JK Flip Flop developed.
- Both the S and the R inputs of the previous SR bistable have now been replaced by two inputs called the J and K inputs respectively after its inventor Jack Kilby. Then this may equates to: J = S and K = R.
- When J=0, K=0, no change in state.
- When J=0, K=1, Q will reset.
- When J=1, K=0, Q will set.
- When J=1, K=1, Toggle *i.e* Q'_n

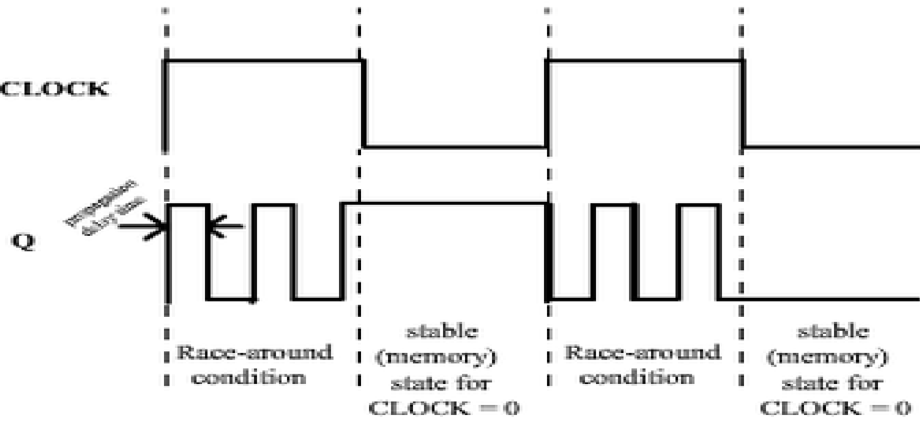




Clk	J	K	Q	Q'	State
1	0	0	Q	Q'	No change in state
1	0	1	0	1	Resets Q to 0
1	1	0	1	0	Sets Q to 1
1	1	1	-	-	Toggles

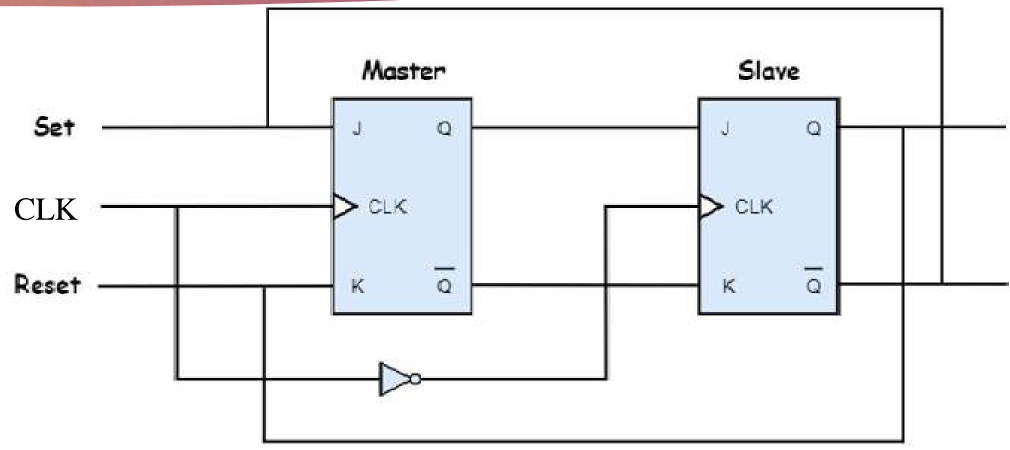
- When J=1, K=1, Toggle *i.e* Q'_n
- For JK flip-flop if J, K and Clock are equal to 1 the state of flip-flop keeps on toggling which leads to uncertainty in determining the output of the flip-flop. This problem is called **Race around** the **condition**.
- This can be avoided by
  - Using Edge triggering of JK Flip Flop
  - Enhancing the propagation delay
  - Using Master-Slave Flip Flop

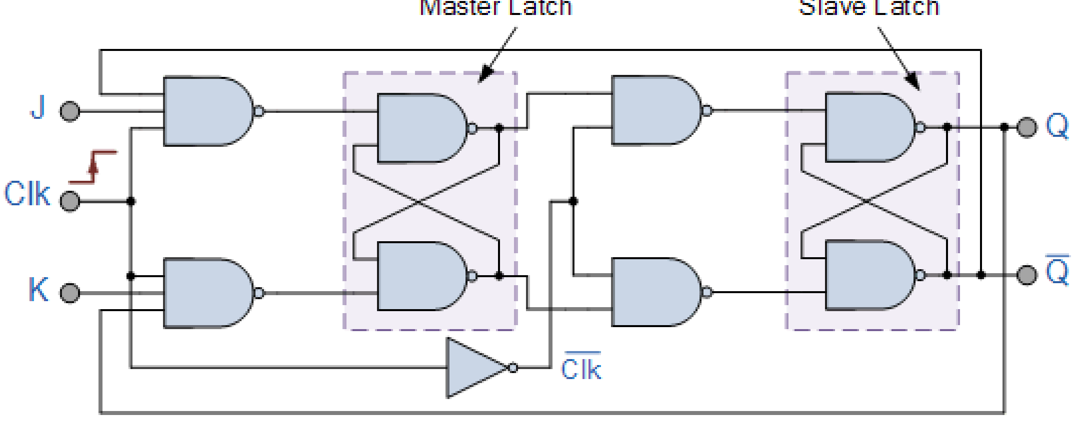




Master-Slave Flip Flop

- Master-slave flip flop is designed using two separate flip flops. Out of these, one acts as the master and the other as a slave.
- The J-K flip flops are presented in a series connection.
- The output of the master J-K flip flop is fed to the input of the slave J-K flip flop.
- The output of the slave J-K flip flop is given as a feedback to the input of the master J-K flip flop.
- The clock pulse [Clk] is given to the master J-K flip flop and it is sent through a NOT Gate and thus inverted before passing it to the slave J-K flip flop.
- It avoids the race around condition of J-K Flip Flop



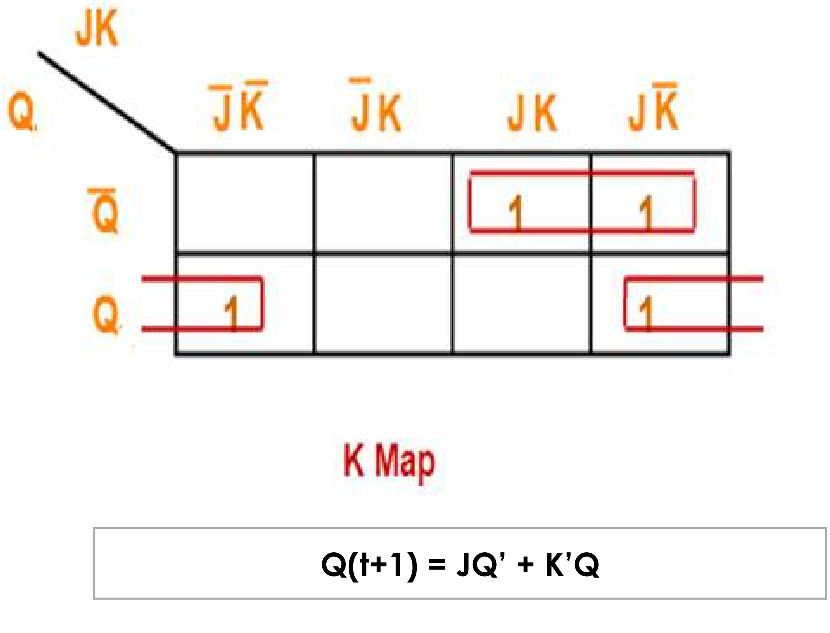


Characteristic Equation of J-K Flip Flop

Characteristic Table

Q	J	K	Q(t+1)
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

JK flip-flop



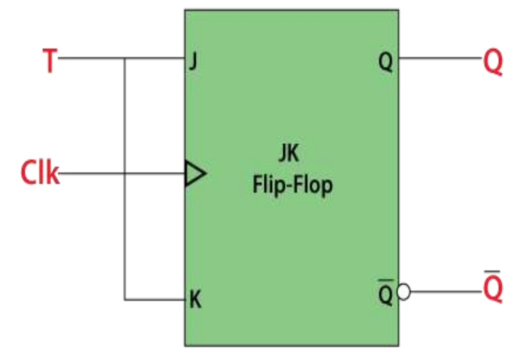
Q(t+1) = JQ' + K'Q

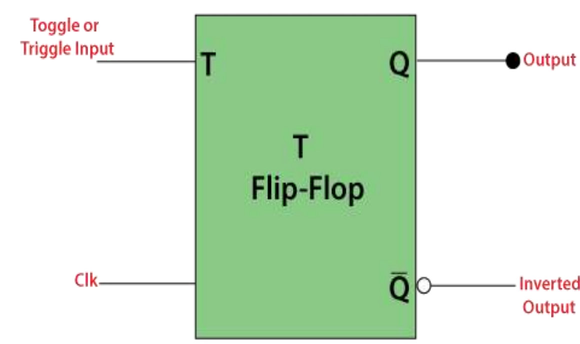
Characteristic Equation

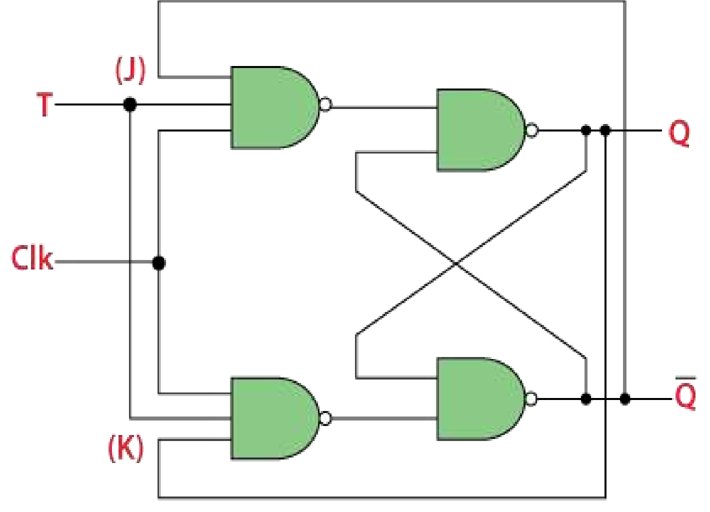
T Flip Flop

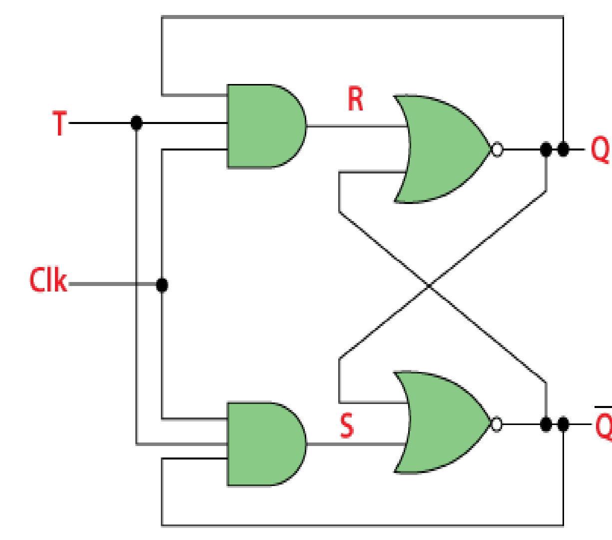
- We can construct the "T Flip Flop" by making changes in the "JK Flip Flop".
- The "T Flip Flop" has only one input, which is constructed by connecting the input of JK Flip Flop.
- This single input is called T.
- Sometimes the "T Flip Flop" is referred to as single input "JK Flip Flop".
- In T flip flop, "T" defines the term "Toggle"

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







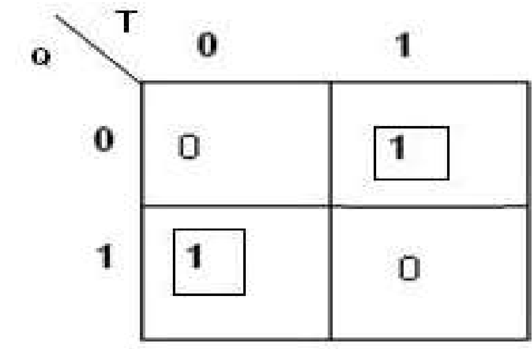


Characteristic Equation of T Flip Flop

Characteristic Table

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

T flip-flop



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

Characteristic Equation

Excitation Table

- The characteristic table is useful for analysis and for defining the operation of flip flop.
- It specifies the next state when the inputs and present state are known.
- During design process we usually know the transition from present state to next state.
- So, we want to know the flip flop input conditions that will cause the required transition.
- Therefore, we need a table that lists the required inputs for a given change of states.
- Such table is called as Excitation Table.

SR Flip-flop

Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

D Flip-flop

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

JK flip-flop

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

T flip-flop

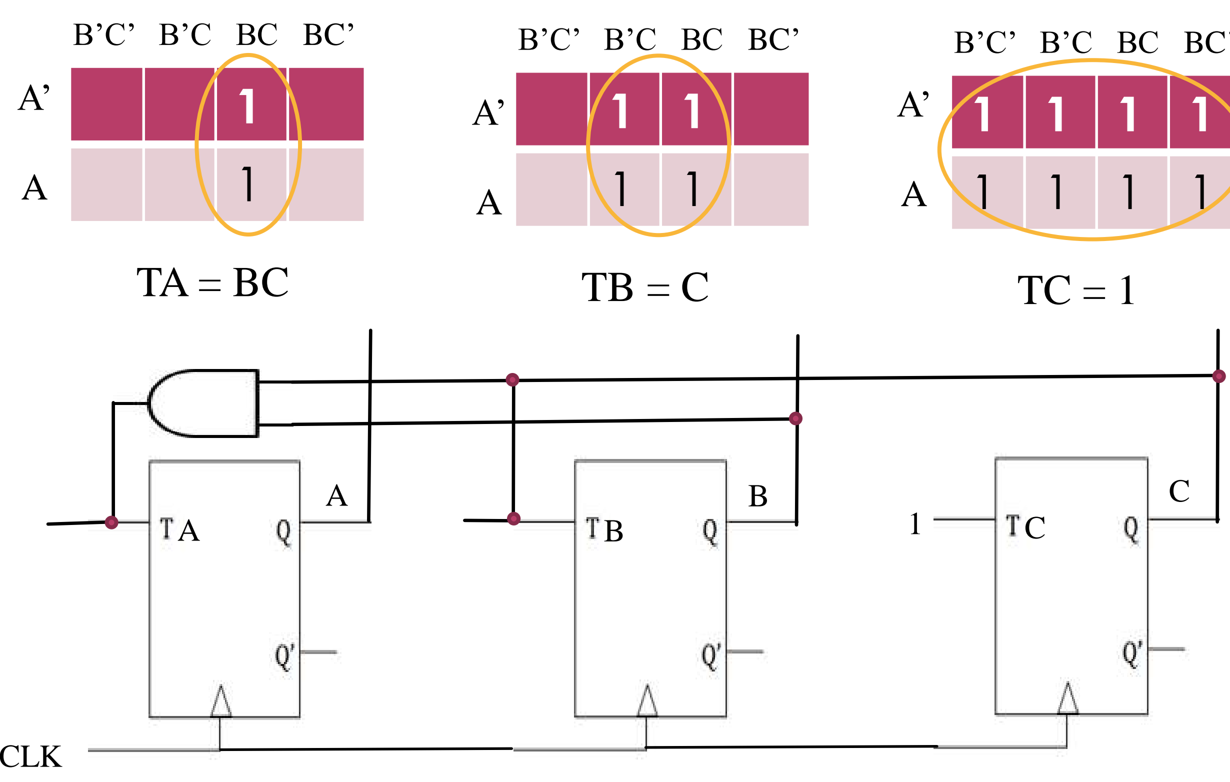
Q(t)	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

Excitation Table for different Flip Flops

Sequential Circuit Design

Example:

Count Sequences			Flip Flop Inputs		
A	B	C	TA	TB	TC
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	1	1	1



Example:

Let the state equations are:

$A(t+1) = A'B'CD + A'B'C + ACD + AC'D'$

$B(t+1) = A'C + CD' + A'BC'$

$C(t+1) = B$

$D(t+1) = D'$

The above equation can be rearranged in the form of characteristic equation of J-K flip flop.

Characteristic equation of J-K flip flop is

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

$A(t+1) = A'B'CD + A'B'C + ACD + AC'D'$

$= (B'CD + B'C)A' + (CD + C'D')A$

So, $J = B'CD + B'C = B'C$

$K = (CD + C'D')' = C'D + CD'$

$B(t+1) = A'C + CD' + A'BC'$

$= (A'C + CD')(B+B') + A'BC'$

$= (A'C + CD')B' + (A'C + CD')B + A'BC'$

$= (A'C + CD')B' + (A'C + CD' + A'C')B$

So, $J = A'C + CD'$

$K = (A'C + CD' + A'C')' = AC'+AD$

$C(t+1) = B = B(C+C') = BC' + BC$

So, $J = B$

$K = B'$

$D(t+1) = D' = (1)D' + (0)D$

So, $J = 1$

$K = 1$

So, finally

$JA = B'C$ | $KA = C'D + CD'$

$JB = A'C + CD'$ | $KB = AC'+AD$

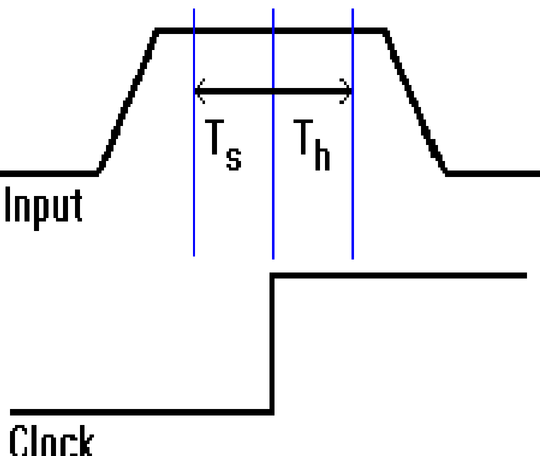
$JC = B$ | $KC = B'$

$JD = 1$ | $KD = 1$

Setup time and Hold time

The Clocking event can be either from low to high or from high to low. The input signal around clocking event must remain unchanged in order to have a correct effect on the outcome of the new state.

- T_s: the minimum time interval preceding the clocking event the input signal must remain available and unchanged.
- T_h: the minimum time interval after edge of the clocking event, the input signals must remain unchanged



Applications

Flip flops will find their use in many of the fields in digital electronics. Flip flops are the main components of sequential circuits. Particularly, edge triggered flip flops are very resourceful devices that can be used in wide range of applications like storing of binary data and transferring binary data from one location to other etc. Some of the most common applications of flip flops are

- Shift Register
- Counter
- Storage Registers
- Memory
- Data transfer
- Frequency Dividers etc.

Sequential Circuits
(Shift Register)

Overview

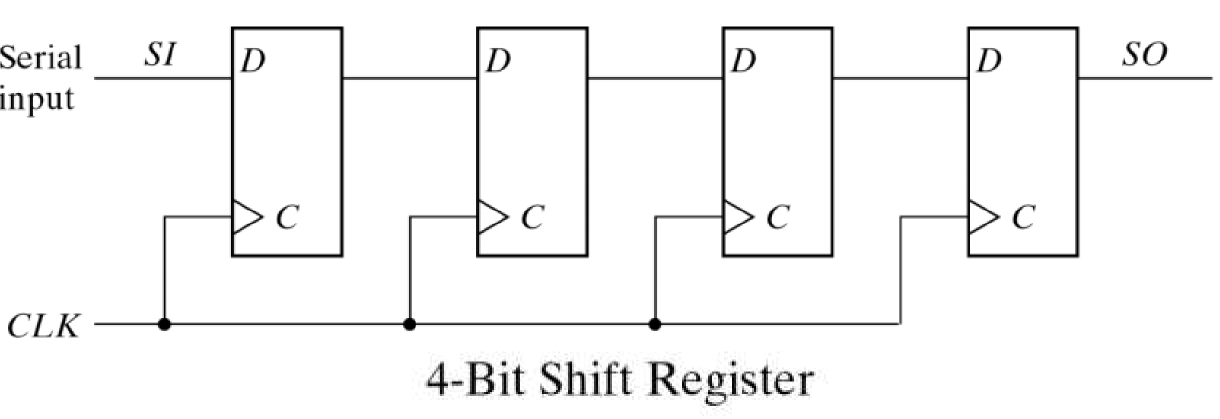
- **Register**
- **Shift Register**
- **Types of Shift Register**
- **Bidirectional Shift Register**
- **Universal Shift Register**
- **Typical ICs for Shift register**

Register

- The filp flops are essential component in clocked sequential circuits.
- Circuits that include filp flops are usually classified by the function they perform.
- Two such circuits are registers and counters.
- An n-bit register consists of a group of n flip flops capable of storing n bits of binary information.
- So, **Register** is a collection of flip flops.
- A flip flop is used to store single bit **digital** data. For storing a large number of bits, the storage capacity is increased by grouping more than one flip flops.
- It is used to perform simple data storage, movement, manipulation and processing operations (e.g. load, increment, shift, add, etc.)
- The computer processes data by performing operations on registers, e.g. ADD A, B where A and B are the registers.
- A register capable of shifting its binary information in one or both direction is called a **shift register**.
- All flip-flops receive common clock pulses, which activate the shift from one stage to the next.

Shift Register

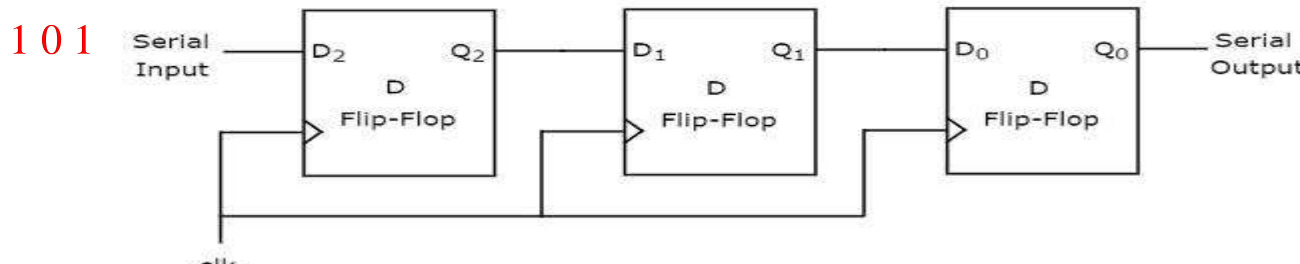
- The simplest possible shift register is one that uses only flip-flops, as shown in Fig.
- Each clock pulse shifts the contents of the register one bit position to the right.
- The serial input determines what goes into the leftmost flip-flop during the shift.
- The serial output is taken from the output of the rightmost flip-flop.
- Following are the four types of shift registers based on applying inputs and accessing of outputs.
- Serial In − Serial Out shift register
- Serial In − Parallel Out shift register
- Parallel In − Serial Out shift register
- Parallel In − Parallel Out shift register



4-Bit Shift Register

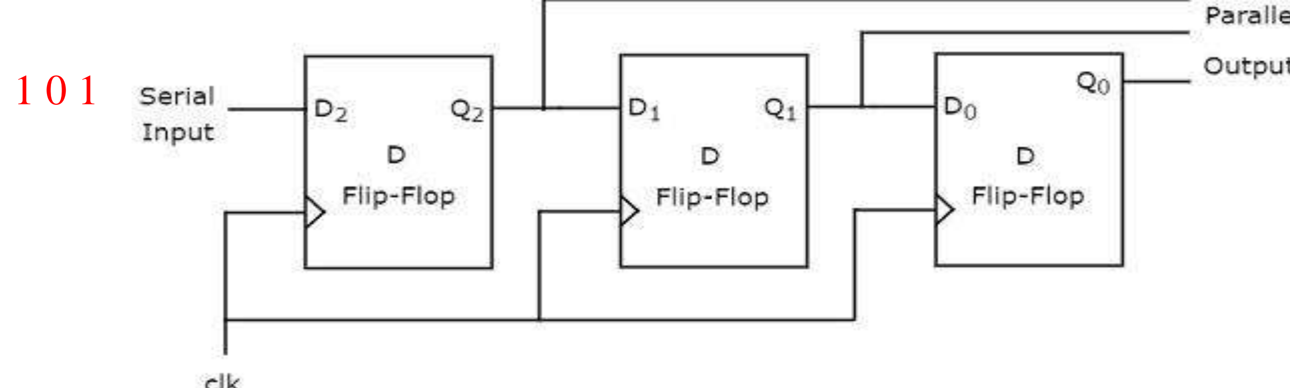
Serial In – Serial Out (SISO) shift register

- The shift register, which allows serial input and produces serial output is known as Serial In – Serial Out (SISO) shift register.
- This block diagram consists of three D flip-flops, which are **cascaded**. That means, output of one D flip-flop is connected as the input of next D flip-flop.
- All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
- In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**.
- For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we can receive the bits serially from the output of right most D flip-flop. Hence, this output is also called as **serial output**.



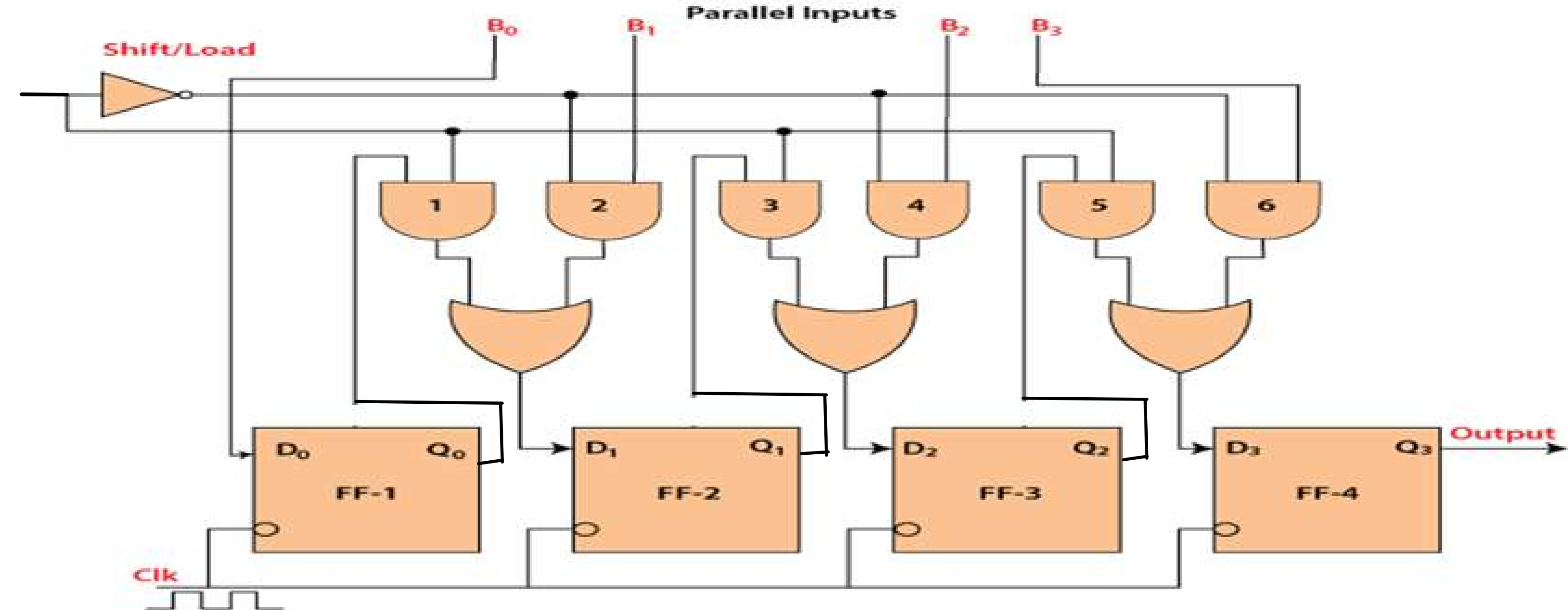
Serial In – Parallel Out (SIPO) shift register

- The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out (SIPO) shift register.
- In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**.
- For every positive edge triggering of clock signal, the data shifts from one stage to the next.
- In this case, we can access the outputs of each D flip-flop in parallel. So, we will get **parallel outputs** from this shift register.



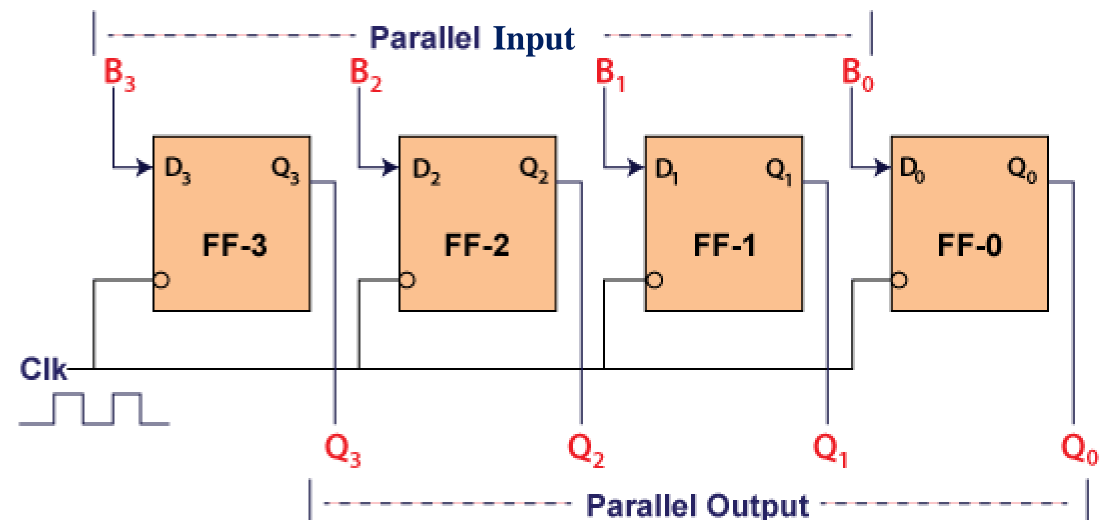
Parallel In – Serial Out (PISO) shift register

- In the **"Parallel In Serial Out"** register, the data is entered in a parallel way, and the outcome comes serially.
- The **shift mode** and the **load mode** are the two modes in which the **"PISO"** circuit works.



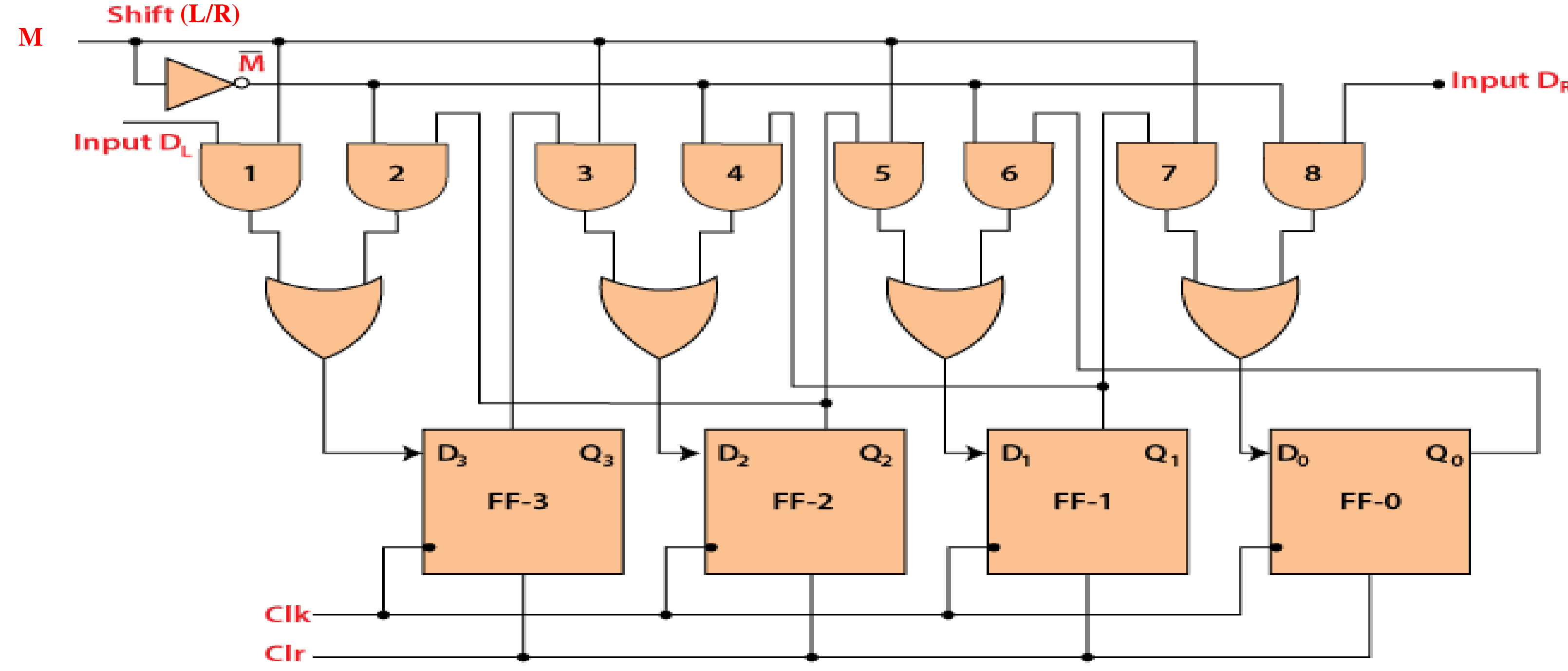
Parallel In – Parallel Out (PIPO) shift register

- In **"Parallel In Parallel Out"**, the inputs and the outputs come in a parallel way in the register.
- The inputs B_0, B_1, B_2, and B_3, are directly passed to the data inputs D_0, D_1, D_2, and D_3 of the respective flip flop.
- The bits of the binary input is loaded to the flip flops when the negative clock edge is applied. The clock pulse is required for loading all the bits. At the output side, the loaded bits appear.



Bidirectional Shift Register

- Below is the diagram of 4-bit **"bidirectional"** shift register where D_R is the **"serial right shift data input"**, D_L is the **"left shift data input"**, and M is the **"mode select input"**.



Universal Shift Register

- A shift-right control to enable the shift operation and the serial input and output lines associated with the shift right.
- A shift-left control to enable the shift operation and the serial input and output lines associated with the shift left.
- A parallel-load control to enable a parallel transfer and the ***n*** input lines associated with the parallel transfer.
- If the Shift register has the capability of

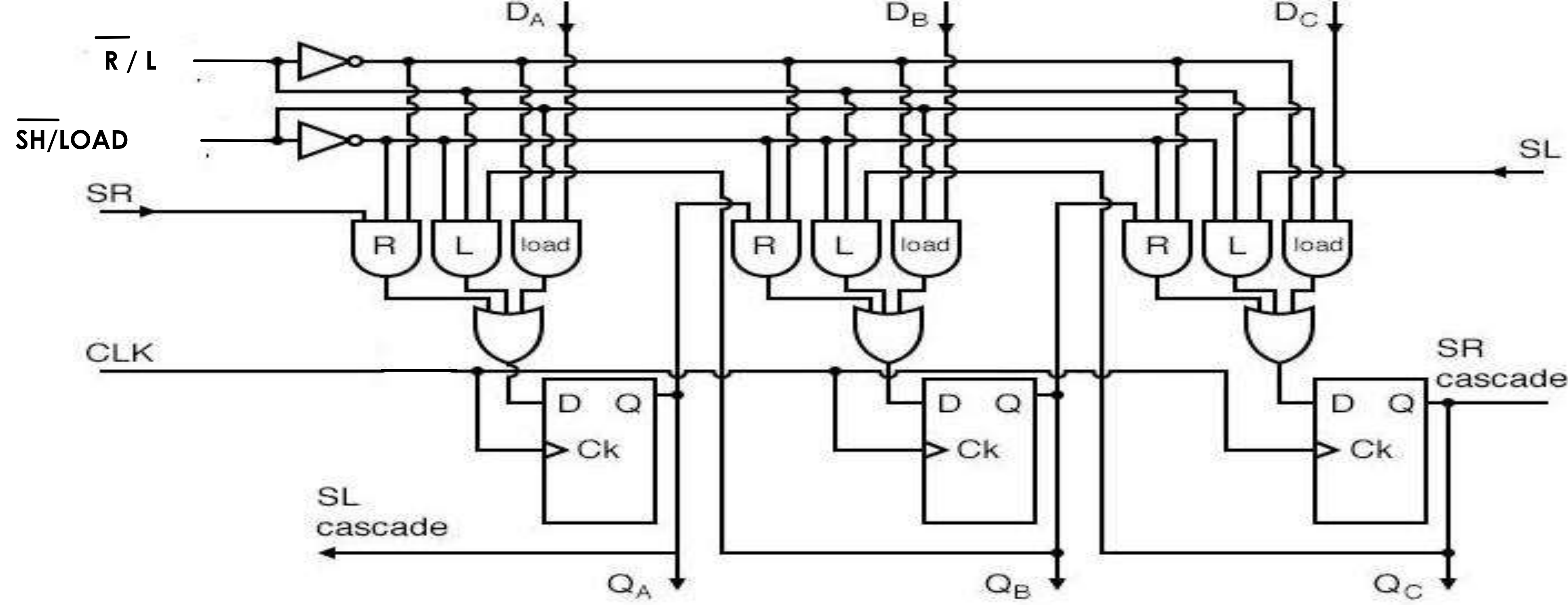
  Serial In – Serial Out

  Serial In – Parallel Out

  Parallel In – Serial Out

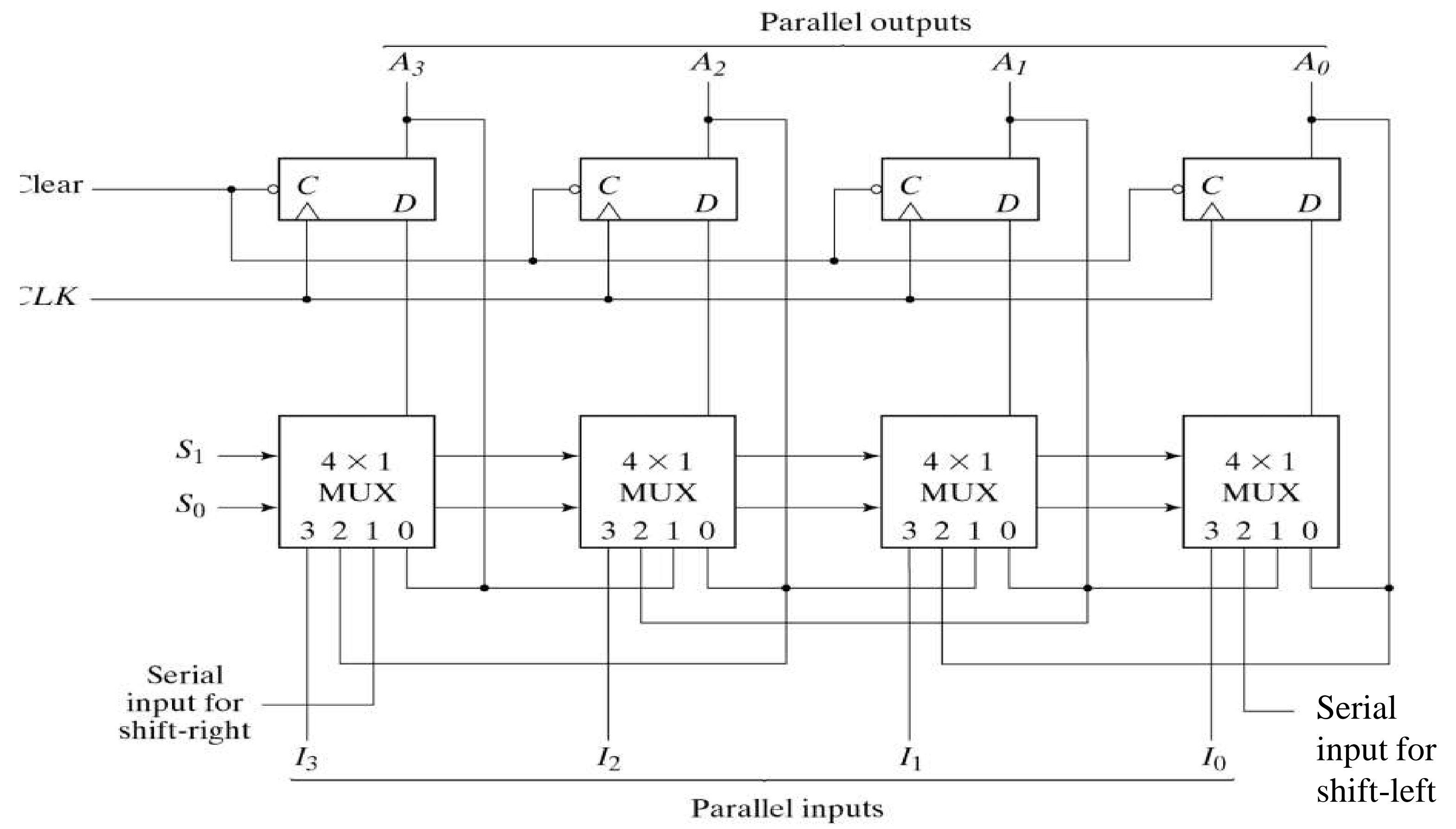
  Parallel In – Parallel Out

  and act as Bidirectional shift register is referred as a universal shift register.
- Shift registers are often used to interface digital system situated remotely from each other. If the distance is far, it will be expensive to use n lines to transmit the n bits in parallel.
- Transmitter performs a parallel-to-serial conversion of data and the receiver does a serial-to-parallel conversion.



Shift left/ right/ load

Universal Shift Register using MUX



Mode Control		Register Operation
S_1	S_0	
0	0	No Change
0	1	Shift right
1	0	Shift Left
1	1	Parallel load

Typical ICs for Shift register

- Commonly available SISO IC is 74HC595, which is 8-bit.
- Commonly available SIPO IC’s include the standard 8-bit 74LS164, 74LS594.
- Commonly available PISO IC is 74HC166, which is 8-bit.
- Commonly available PIPO IC’s include the standard 8-bit M54HC195, M74HC195.
- Today, there are many high speed bi-directional or “universal” type **Shift Registers** available such as the TTL 74LS194, 74LS195 or the CMOS 4035 which are available as 4-bit multi-function devices that can be used in either serial-to-serial, left shifting, right shifting, serial-to-parallel, parallel-to-serial, or as a parallel-to-parallel multifunction data register, hence their name “Universal”.
- The 74HC299 is an 8-bit Universal Shift register.
- The 74S299 is an 8-bit Universal Shift and Storage Register.

Lecture of Module 5

Sequential Circuits
(Counter)

Overview

- **Introduction**
- **Synchronous counter**
- **Asynchronous counter**
- **Up counter**
- **Down counter**
- **Decade counter**
- **Ring counter**
- **Johnson counter**

Introduction

- Counter essentially a register that goes through predetermined sequence of states upon the application of input pulses.
- A counter is a device which can count any particular event on the basis of how many times the particular event(s) is occurred.
- In a digital logic system or computers, this counter can count and store the number of time any particular event or process have occurred, depending on a clock signal.
- Most common type of counter is sequential digital logic circuit with a single clock input and multiple outputs.
- The outputs represent binary or binary coded decimal numbers.
- Each clock pulse either increase the number or decrease the number.
- Modulus of a counter is the total number of states through which a counter can progress.
- Two types of counters:
  - Synchronous (parallel) counters
  - Asynchronous (ripple) counters

Synchronous Counter

- Synchronous counter known as parallel counter.
- All flip flops of the counter changes their states at the same time in synchronous with the input clock signal.
- All flip-flops of the counter driven by same clock.
- Circuit delay is equal to the propagation delay of one flip flop.

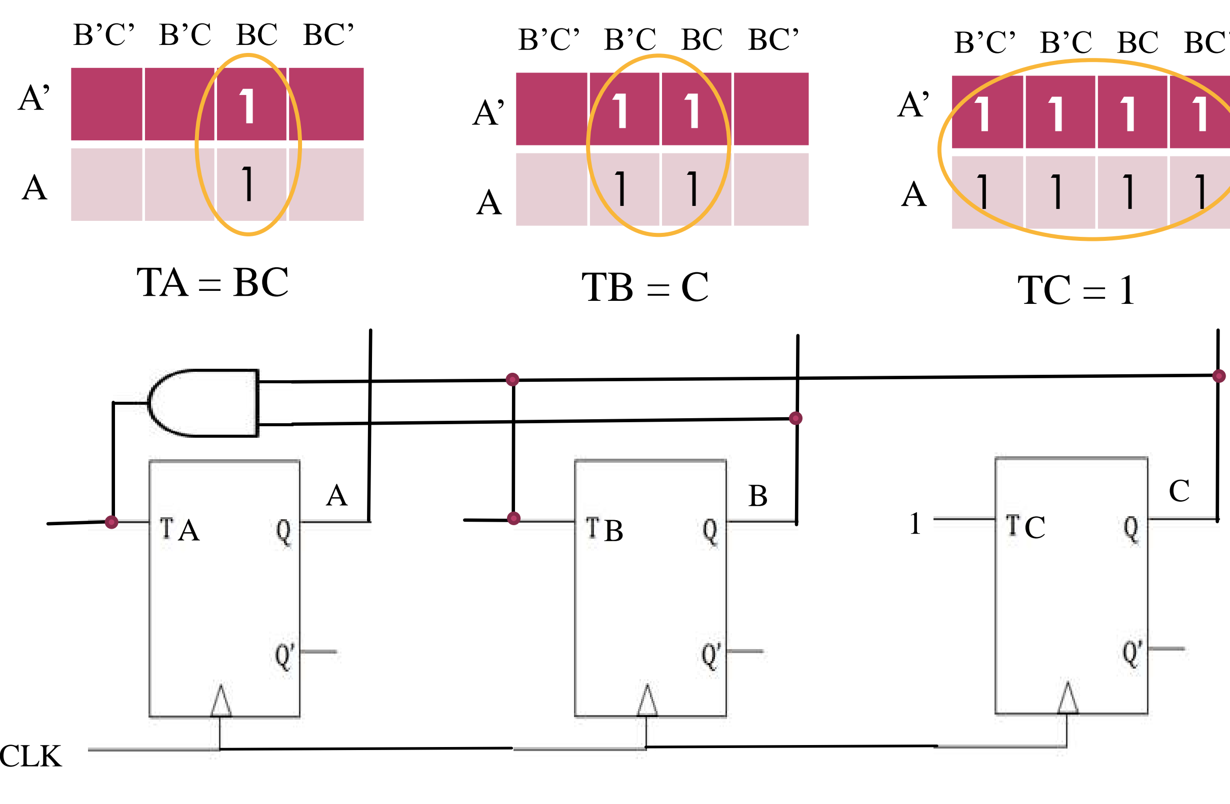
Asynchronous Counter

- Known as Ripple counter.
- Also known as Serial counter.
- Output of one flip flop is used as clock input of other flip flop.
- Circuit delay is equal to the sum of propagation delay of all flip flops.

Synchronous Counter

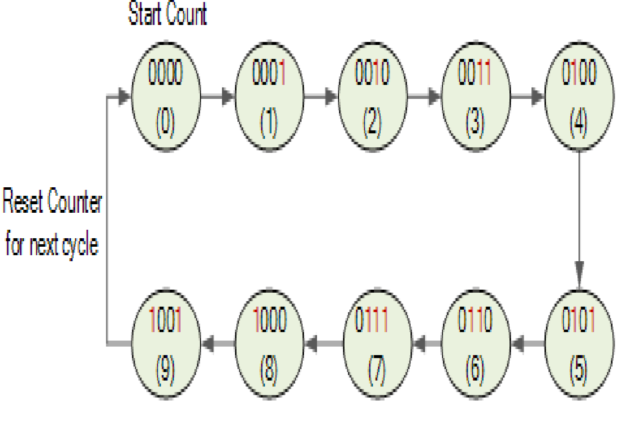
Binary Counter

Count Sequences			Flip Flop Inputs		
A	B	C	TA	TB	TC
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	1	1	1



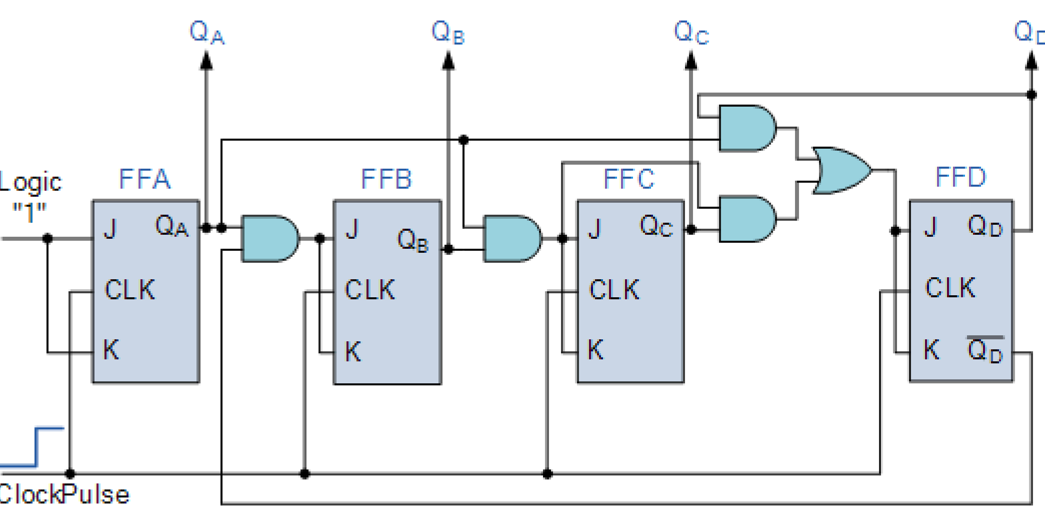
Decade or BCD synchronous counter

- A 4-bit decade synchronous counter can also be built using synchronous binary counters to produce a count sequence from 0 to 9.
- A standard binary counter can be converted to a decade (decimal 10) counter with the aid of some additional logic to implement the desired state sequence.
- After reaching the count of "1001", the counter recycles back to "0000". We now have a decade or **Modulo-10** counter or **MOD-10** counter.



Truth Table

count	Q_D	Q_C	Q_B	Q_A
0 [start]	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 [new cycle]	0	0	0	0



Asynchronous Counter

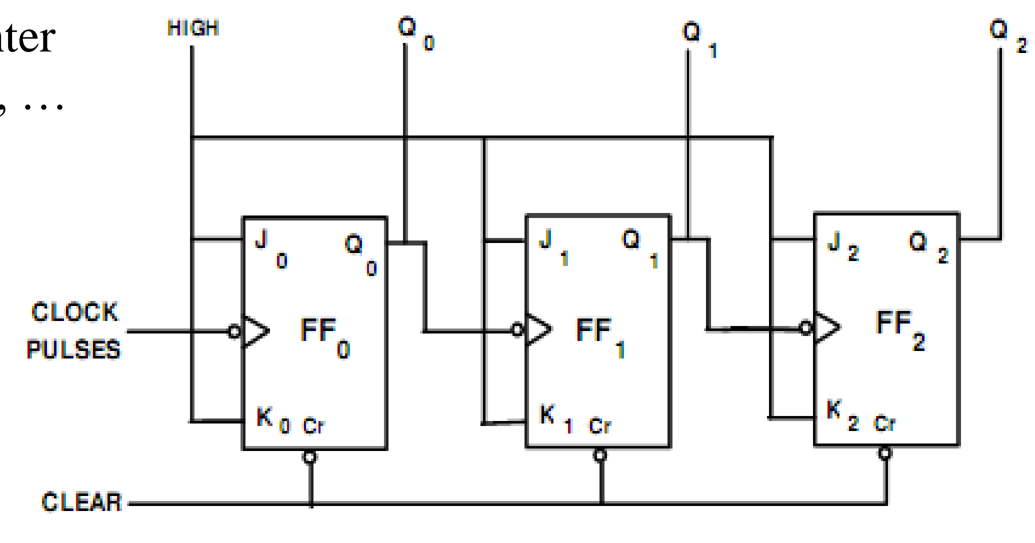
Binary Ripple Counter

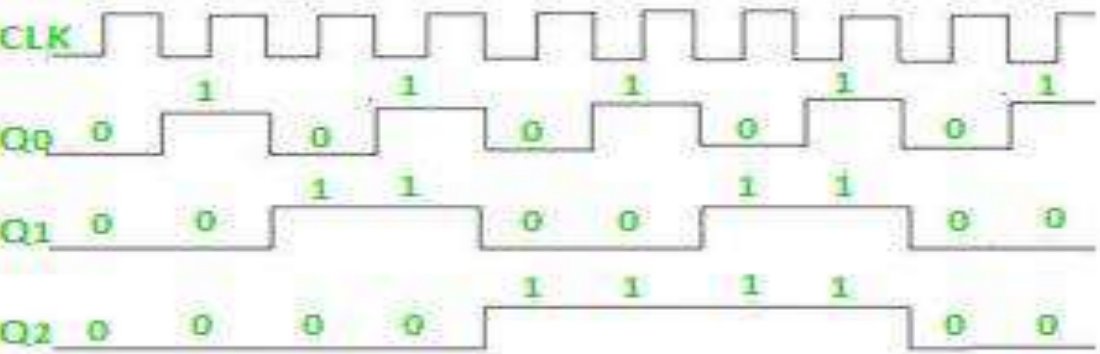
- Ripple counter is an Asynchronous counter. It got its name because the clock pulse ripples through the circuit.
- It is an asynchronous counter.
- Different flip-flops are used with a different clock pulse.
- All the flip-flops are used in toggle mode.
- Only one flip-flop is applied with an external clock pulse and another flip-flop clock is obtained from the output of the previous flip-flop.
- The flip-flop applied with external clock pulse act as LSB (Least Significant Bit) in the counting sequence.
- A counter may be an up counter that counts upwards or can be a down counter that counts downwards or can do both i.e. count up as well as count downwards depending on the input control.
- When counting up, for n-bit counter the count sequence goes from 000, 001, 010, … 110, 111, 000, 001, … etc.
- When counting down the count sequence goes in the opposite manner: 111, 110, … 010, 001, 000, 111, 110, … etc.

Ripple Counter

Count Up: When counting up, for n-bit counter the count sequence goes from 000, 001, 010, … 110, 111, 000, 001, … etc.

Counter State	Q_2	Q_1	Q_0
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

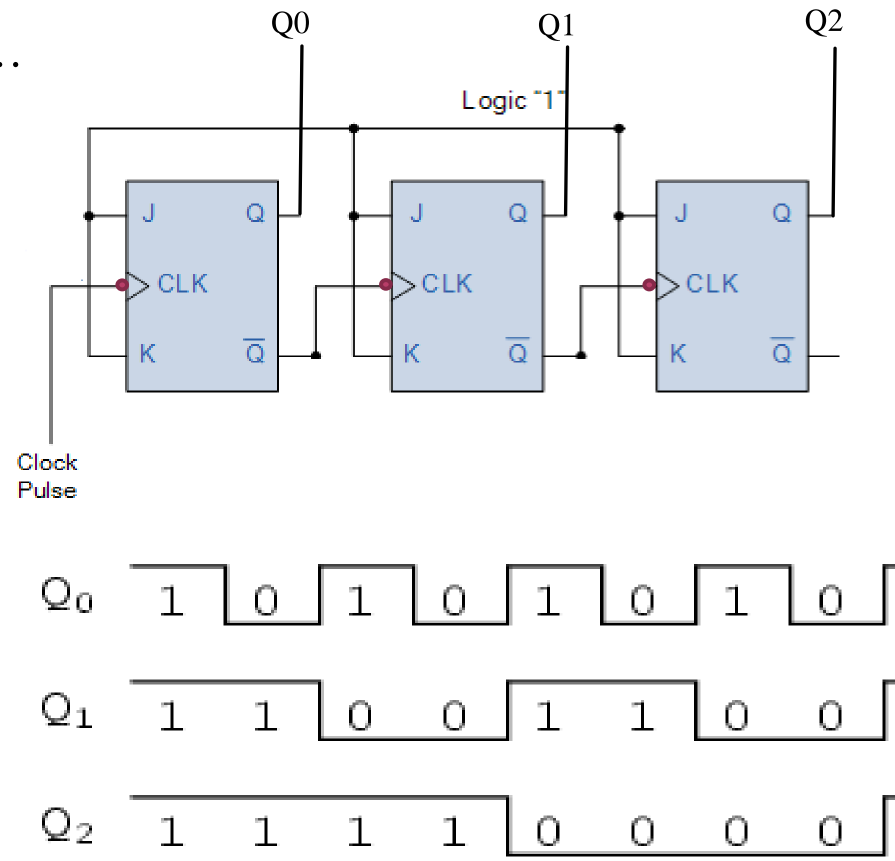




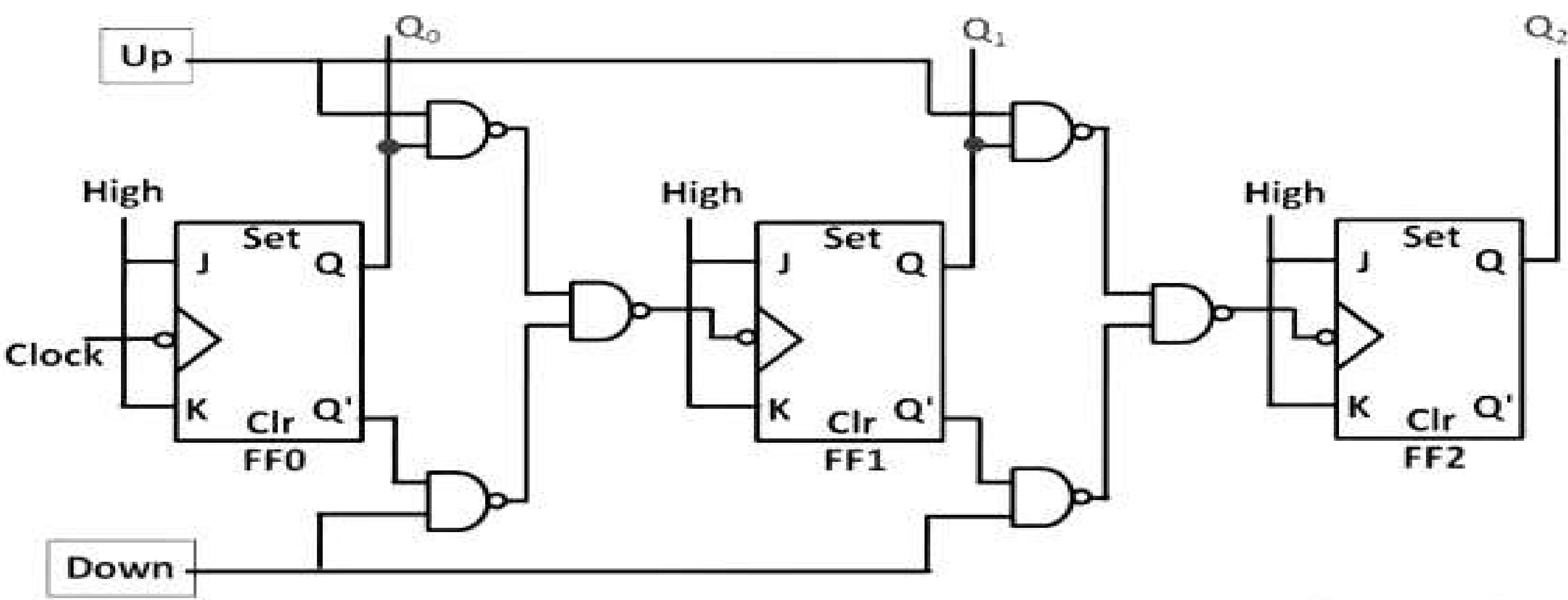
Ripple Counter

Count Down: When counting down the count sequence goes in the opposite manner: 111, 110, … 010, 001, 000, 111, 110, … etc.

Count States	Q2	Q1	Q0
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1
0	0	0	0

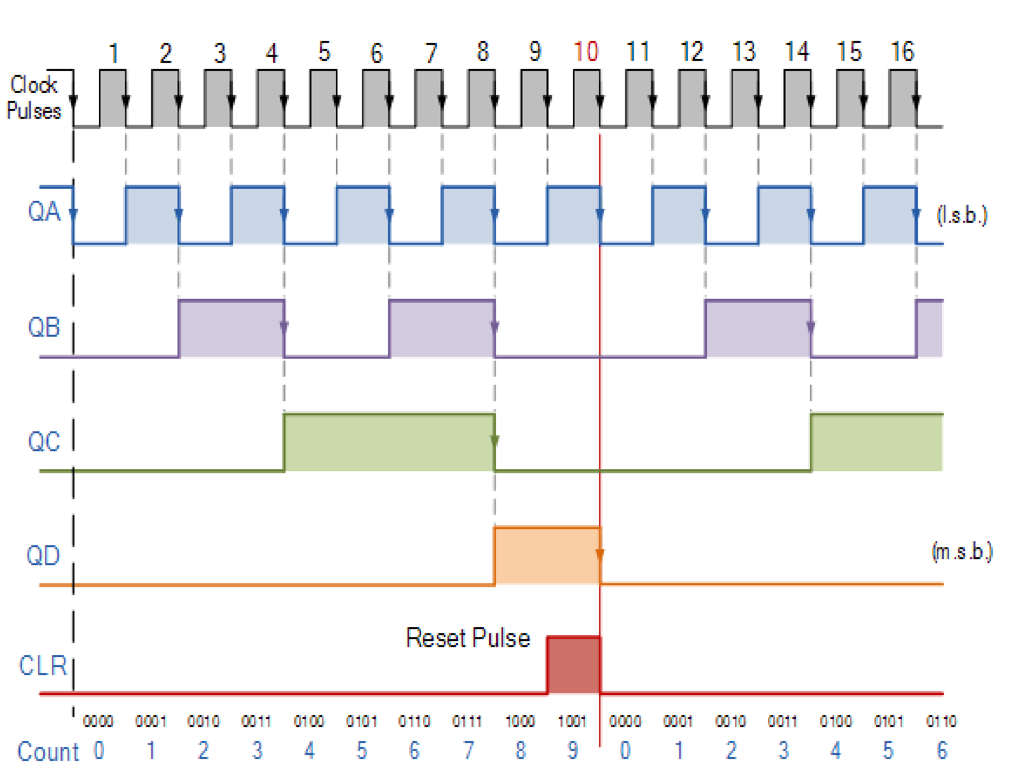


Up/Down Counter (Asynchronous)

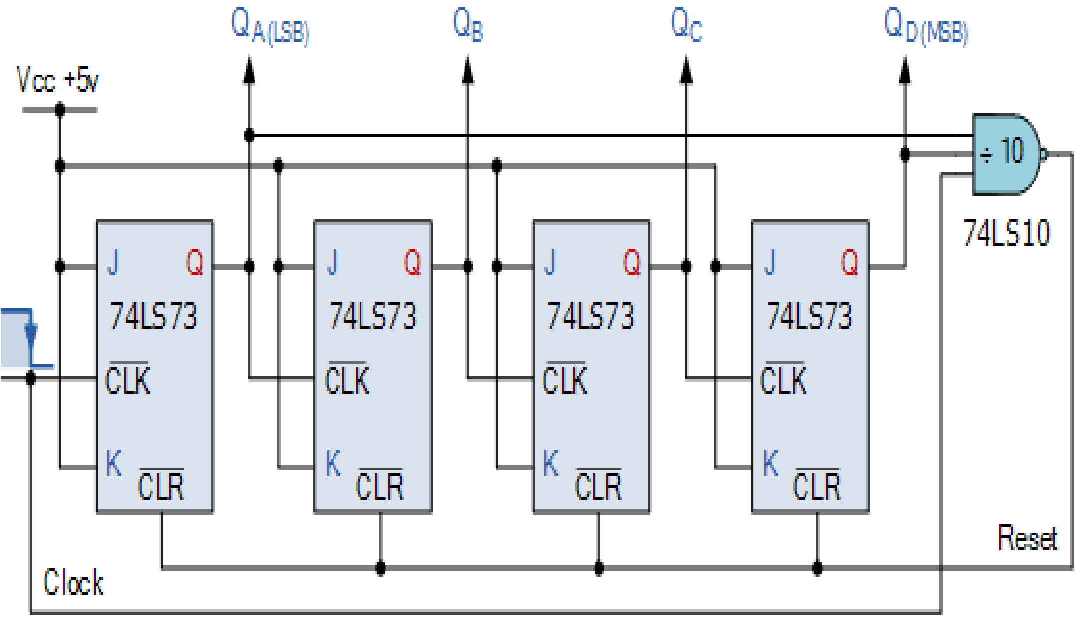


Decade or BCD asynchronous counter

- If we take the modulo-16 asynchronous counter and modify it with additional logic gates it can be made to give a Decade counter output for use in standard decimal counting and arithmetic circuits. Such counters are generally referred to as **Decade Counters or BCD Counters**.
- A decade counter requires resetting to zero after the output count reaches the decimal value of 9, i.e. when DCBA = 1001.
- This type of asynchronous counter counts upwards on each input clock signal starting from 0000 until it reaches an output 1001 (decimal 9).
- When it is 1001, both outputs QA and QD are now equal to logic "1". On the application of the next clock pulse, by connection NAND gate to QA and QD, the output from the NAND gate changes state from logic "1" to a logic "0" level.
- The output of NAND gate is connected to CLEAR inputs of flip flpos.
- As, the output of the NAND gate is connected to the CLEAR (CLR) inputs of all the flip-flops, this signal causes all of the Q outputs to be reset back to binary 0000 after the count 9.
- So, the counter restarts again from 0000. We now have a decade or Modulo-10 up-counter.

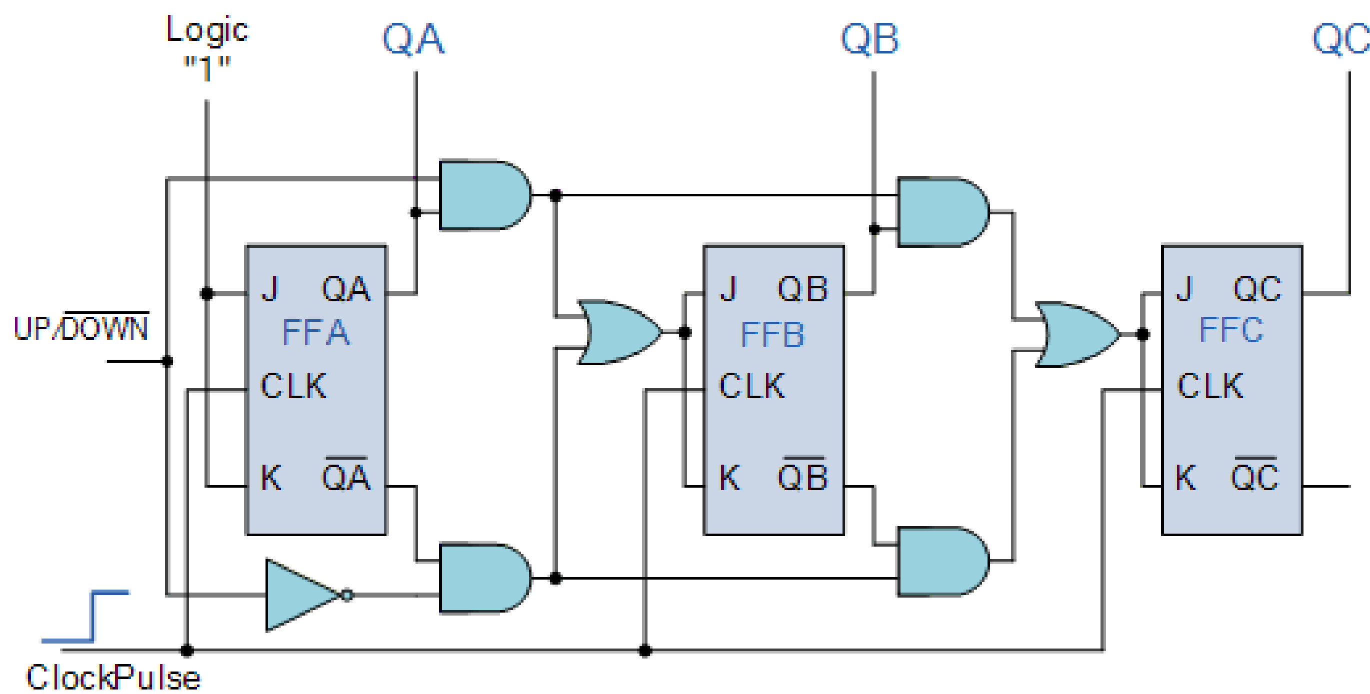


Clock Pulses
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
QA
(l.s.b.)
QB
QC
QD
(m.s.b.)
Reset Pulse
CLR
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 0000 0001 0010 0011 0100 0101 0110
Count 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6



Vcc +5v
QA(LSB)
QB
QC
QD(MSB)
J Q
74LS73
CLK
K CLR
÷10
74LS10
Reset
Clock

Up/Down Counter (Synchronous)



Ring Counter

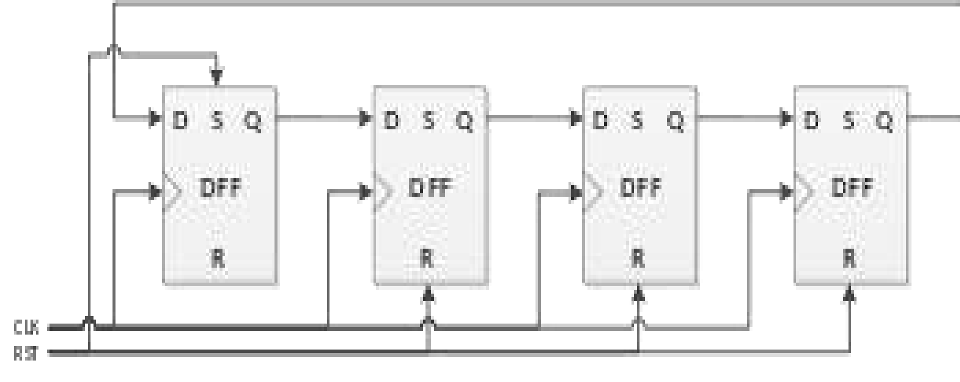
- A **ring counter** is a type of counter composed of flip flops working as shift register, with the output of the last flip-flop fed to the input of the first, making a "circular" or "ring" structure.
- There are two types of ring counters:
- A **straight ring counter**, connects the output of the last shift register to the first shift register input and circulates a single one bit around the ring.
- A **twisted ring counter**, also called **switch-tail ring counter**, **Johnson counter** connects the complement of the output of the last shift register to the input of the first register and circulates a stream of ones followed by zeros around the ring.
- The straight and twisted forms have different properties, and relative advantages and disadvantages.
- A binary counter can represent 2^N states, where N is the number of bits (flip flops).
- Whereas a straight ring counter can represent only N states.
- Johnson counter can represent 2N states.

- Johnson counters are sometimes favored, because they offer twice as many count states from the same number of flip flops in the shift registers, and because they are able to self-initialize from the all-zeros state, without requiring the first count bit to be injected externally at start-up.
- The Johnson counter generates a code in which adjacent states differ by only one bit (that is, have a Hamming distance of 1), as in a Gray code, which is advantageous in communication system.
- When a fully decoded representation of the counter state is needed, as in some sequence controllers, the straight ring counter is preferred.
- There are two types of ring counters:

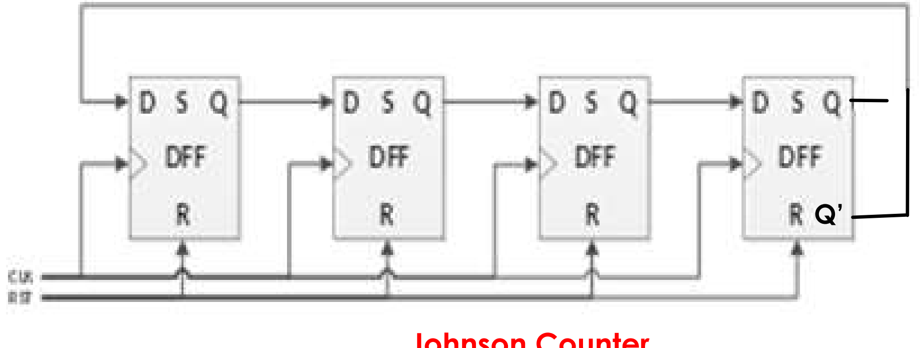
  Straight ring counter

  Twisted ring counter

Straight ring counter					Johnson counter				
State	Q0	Q1	Q2	Q3	State	Q0	Q1	Q2	Q3
0	1	0	0	0	**0**	0	0	0	0
1	0	1	0	0	**1**	1	0	0	0
2	0	0	1	0	**2**	1	1	0	0
3	0	0	0	1	**3**	1	1	1	0
0	1	0	0	0	**4**	1	1	1	1
1	0	1	0	0	**5**	0	1	1	1
2	0	0	1	0	**6**	0	0	1	1
3	0	0	0	1	**7**	0	0	0	1
0	1	0	0	0	***0***	0	0	0	0



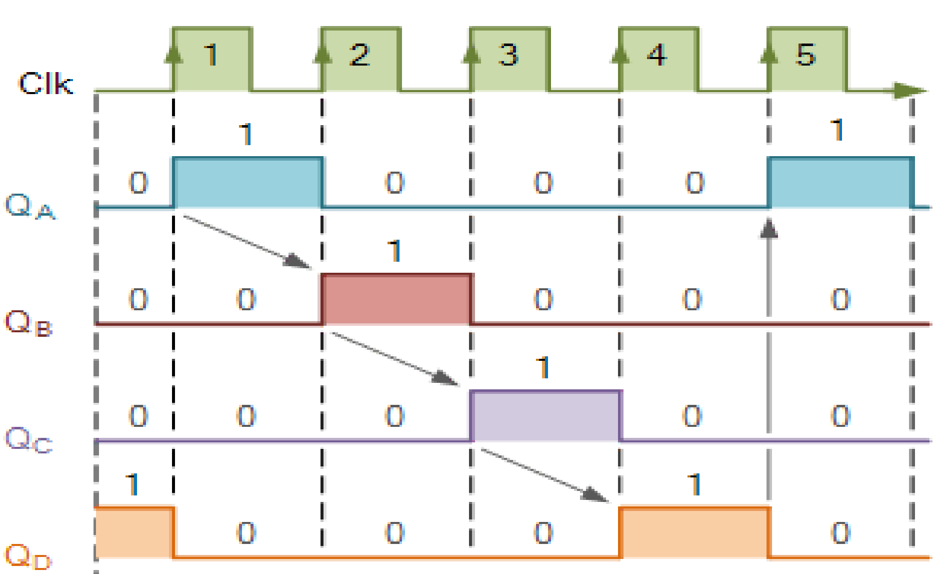
Ring Counter



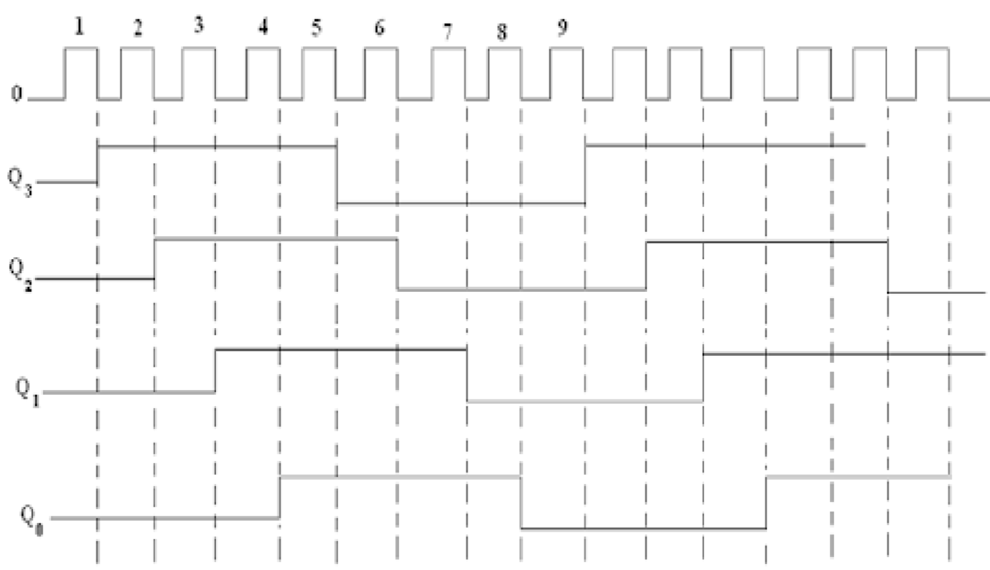
Johnson Counter

Ring counter has 4 sequences: 1000, 0100, 0010, 0001.

Johnson ring counter has sequences like “1000”, “1100”, “1110”, “1111”, “0111”, “0011”, “0001”, “0000”.



Timing Diagram of Ring Counter



Timing Diagram of Johnson Counter

Differences:

SYNCHRONOUS COUNTERS	ASYNCHRONOUS COUNTERS
The propagation delay is very low.	Propagation delay is higher than that of synchronous counters.
Its operational frequency is very high.	The maximum frequency of operation is very low.
These are faster than that of ripple counters.	These are slow in operation.
Large number of logic gates are required to design	Less number of logic gates required.
High cost.	Low cost.
Synchronous circuits are easy to design.	Complex to design.
Standard logic packages available for synchronous.	For asynchronous counters, Standard logic packages are not available.

Applications

Some of the counter applications are listed below.

- Frequency counters
- Digital clocks
- With some changes in their design, counters can be used as frequency divider circuits. The frequency divider circuit is that which divides the input frequency exactly by ‘2’.
- Counter used as a timer in electronic devices like ovens and washing machines
- Alarm Clock, AC Timer, timer in camera to take the picture, flashing light indicator in automobiles, car parking control etc.
- Counting the time allotted for special process or event by the scheduler.
- They are also used in machine moving control.