Sukkur Institute of Business Administration University

Department of Electrical Engineering.

Digital Logic Design

Course Handouts

Instructor

Dr. Muhammad Asim Ali

Fall 2024

Contents

Chapter 1

Introduction

Outline

The objective of this chapter / course is to motivate you to appreciate the importance of digital logic to design digital systems using basic elements of logic and understand the design of digital systems through elementary logic components.

Ⓐ Motivation

Ⓑ Digital & Analog Systems

Ⓒ Modelling of Digital System

Ⓓ Advantages of Digital / Analog systems

Ⓔ Examples of Digital Systems.

1.1 Motivation

Digital systems (anything useful you can ever imagine) ranging from industry, research and development, Information Technology would not have been possible without digital systems. The most important aspect of Digital systems is that they are highly reconfigurable and can process complex information. They include systems such as Computer and Mobile Phones and other networking devices, application specification integrated circuits for medical, Defence, Instrumentation & Measurement, Robotics, Industrial Automation and Space applications just to name a few.

This elementary course is your journey to the fundamentals of Digital Circuit Design using the constructs of Digital Logic.

1.2 Digital vs. Analog Signals

Natural forces and signals are all analogue (continuous) signals which may be measured as a continuous signal. On the other hand our digital (non continuous binary) signals which can be processed and communicated more effectively. This leads us to translate analog signals into digital form.

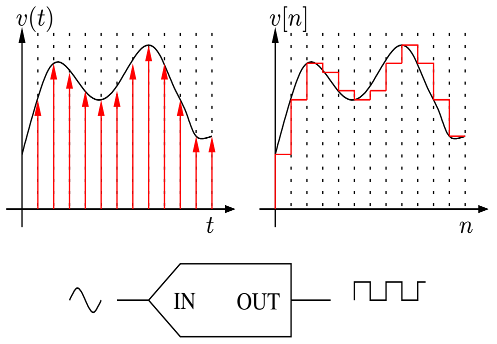


Figure 1.1: Analog to Digital Conversion via sampling process.

Characteristic of digital systems is its manipulation of discrete elements i.e. impulses, decimal digits, letters, alphabets arithmetic operations or any other meaningful symbols.

1.3 Mathematical Model of Logical System

Many physical systems can be described as mathematical functions whose solutions completely describe the behaviour of device.
A digital device is an interconnect of simpler digital devices (such as transistors). A highly complex system such as microprocessor may have billions of transistors packed in space as small as 1 cm^2.

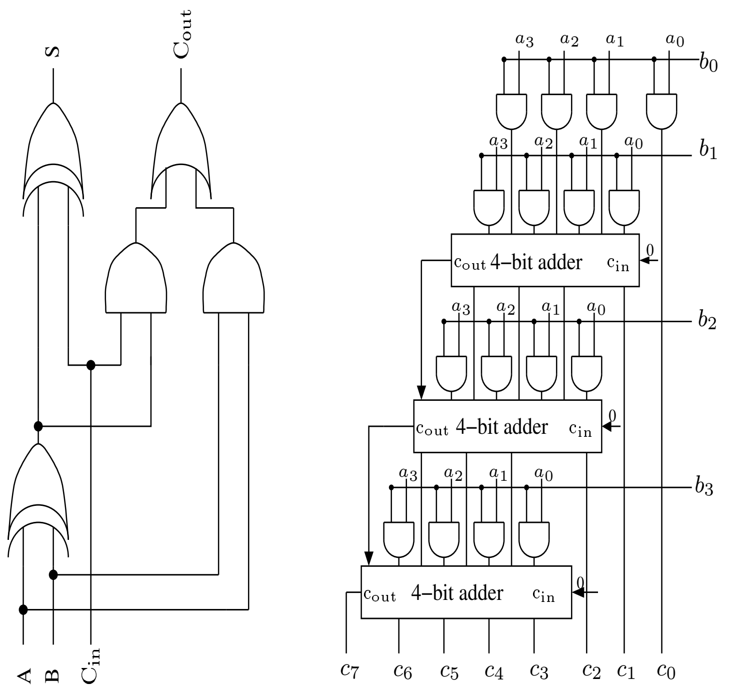


Figure 1.2: (a) Circuit of One-bit Full-adder (b) Circuit of 4-bit Multiplier with uses several FA's.

Fig. 1.2 illustrates that sophistication of device is directly proportional to the number of transistors. To understand/ design the behaviour of a digital devices, it is important the have the knowledge of the basic building blocks of digital system and to understand their behaviour.

Over the past 50 years design tools and manufacturing technology has become a well established industrial standard.

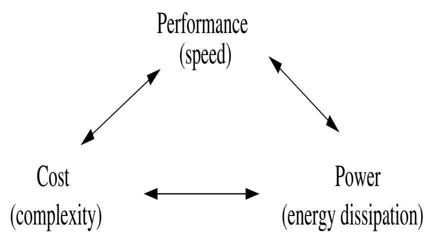


Figure 1.3: Key factors affecting system design.

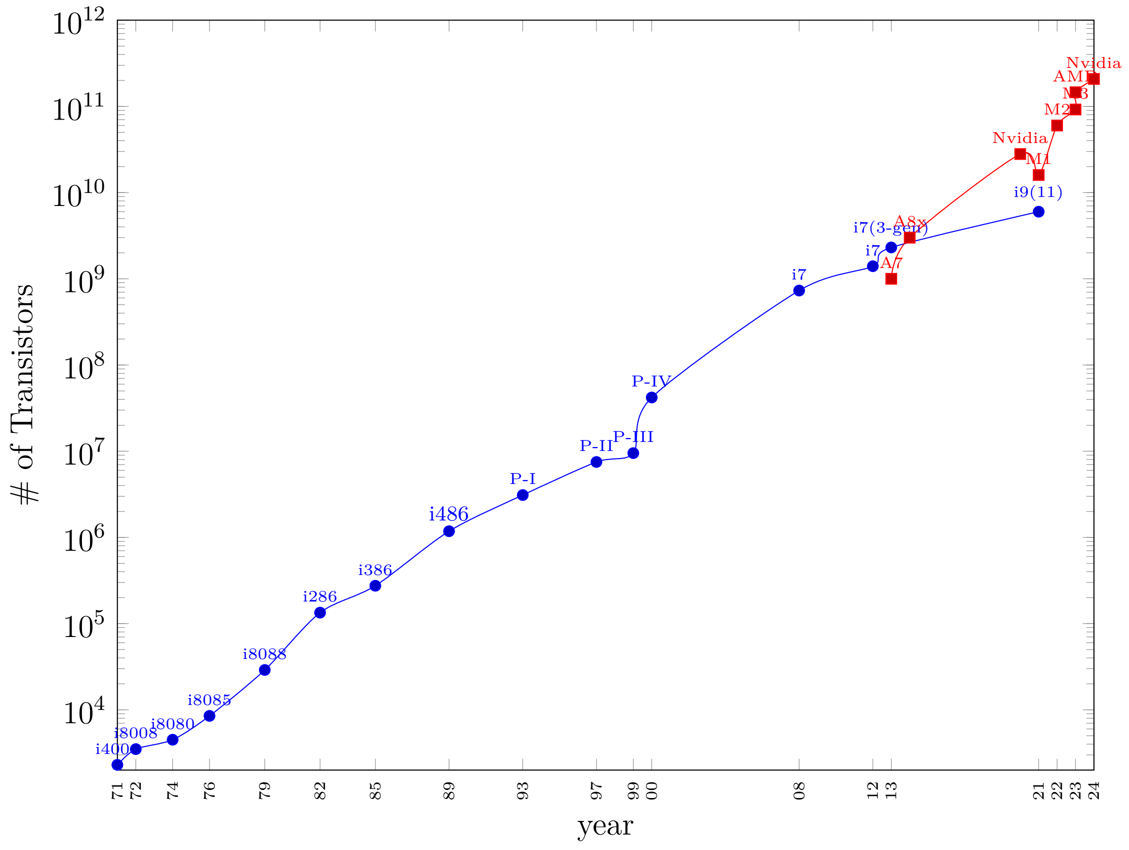
Why digital?

Property	Digital	Analog
Precision	Generally unlimited	Generally limited
	cost, complexity	increase in performance
	and precision	drastic increase in cost
Aging	Without problems	Problematic
Production costs	Low	Higher
Linear-phase	exactly realizable	approximate realization
frequency response		
Complex Algorithms	realizable	strong limitations

1.4 Examples

The application of digital systems are diverse. It is not an exaggeration to say that enhancement in daily life would have been impossible without application of digital circuits and systems.
A handful of applications and examples are illustrated in the picture below.

Figure 1.4: a. Calculator b. Microprocessor c. HDMI (Encoder / Decoder) d. DES (Encryptor / Decryptor)



Moore's Law

In 1965, Gordon Moore co-founder of the Intel corporation predicted that "The number of transistors and resistors on a single chip will double every 18 months", regarding the development of semiconductor gate technology. When Gordon Moore made his famous comment way back in 1965 there were approximately only 60 individual transistor gates on a single silicon chip or die.

The worlds first microprocessor developed in 1971 Intel 4004 had a 4-bit data bus and contained about 2,300 transistors on a single chip, operating at about 600kHz. Today, Apple Corporation have placed a staggering 116 Billion individual transistor gates onto its new M3 64-bit microprocessor chip operating at nearly 4GHz, and the on-chip transistor count is still rising, as newer faster microprocessors and microcontrollers are developed.

Chapter 2

Number Systems

Outline

The objective of this chapter is to take a quick review of the elementary concepts essential for this module. This chapter highlights the all important concepts of sampling and quantization. The fundamental concepts covered in this chapter are

Ⓐ Types of Numbers
Ⓑ Numbers Conversion
Ⓒ Octal and Hexa-Decimal Numbers
Ⓓ Complement Codes
Ⓔ Binary Codes
Ⓕ Cyclic Parity Check
Ⓖ Gray Codes
Ⓗ ASCII Codes

2.1 Types of Numbers

A decimal number such as 7392 represents a quantity equal to 7 thousands plus 3 hundreds, plus 9 tens and 2 units. The thousands, hundreds e.t.c. are powers of 10 implied by the position of coefficients. To be more exact, 7392 should be written as

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

However the convention is to write only the coefficients and from their position deduce the necessary power of 10. In general, a number with decimal point is represented by a series of coefficients as follows:

$$a_5a_4a_3a_2a_1a_0a_{-1}a_{-2}a_{-3}$$

For a_i coefficients are one of the ten digitals $(0, 1, \cdots, 9)$ and subscript i gives the place value hence the power of 10 by when the coefficients must be multiplied.

$$a_5{\cdot}10^5 + a_4{\cdot}10^4 + a_3{\cdot}10^3 + a_2{\cdot}10^2 + a_1{\cdot}10^1 + a_0{\cdot}10^0 + a_{-1}{\cdot}10^{-1} + a_{-2}{\cdot}10^{-2} + a_{-3}{\cdot}10^{-3}$$

The decimal number system is said to be <u>of base / of radix</u> <u>‘10’</u>. The binary system is different number system as it has only <u>two possible values 0 and 1</u>; Each coefficient a_i is multiplied by 2^i. For example, the decimal equivalent of the number 110010.11 is 50.75, as illustrated below:

$$1 \times 2^5 + 1 \times 2^4 + 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

In general, a number expressed in base-r system has coefficients multiplied by powers of r:

$$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_2r^2 + a_1r^1 + a_0 \quad + a_{-1} \cdot r^{-1} + a_{-2}r^{-2} + \cdots + a_{-m}r^{-m}$$

in compact notation this can be rewritten as:

$$x = \sum_{i:-m}^{n} a_ir^i$$

The coefficients a_i range in the values from 0 and $r - 1$. To distinguish between numbers of different bases, the bases is written as subscript. Specially so when dealing with number of different bases.

Note!
Write the general formula here.

Conversion of Number Basis

Dec. to Dec.	$392.25_{10} \rightarrow 3\times10^2+9\times10^1+2\times10^0+2\times10^{-1}+5\times10^{-2}$	392.25_{10}
Bin. to Dec.	$1010.011_2 \rightarrow 2^3+2^1+2^{-1}+2^{-2}+2^{-3}$	10.375_{10}
Oct. to Dec.	$630.4_8 \rightarrow 6\times8^2+3\times8+4\times8^{-1}$	408.5_{10}
Hex. to Dec.	$B65F_{16} \rightarrow 11\times16^3+6\times16^2+5\times16^1+15$	46687_{10}

Table 2.1: Numbers with Different Bases.

Decimal	Binary	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

2.2 Numbers Conversion

Hexa-decimal, Decimal and Octal number can be converted to binary format through the process of *long division*, Examples of the process are illustrated below:

Decimal to Binary Conversion

$$\begin{array}{r|l}
2 & 12 \rightarrow 0 \\ \cline{2-2}
2 & 6 \rightarrow 0 \\ \cline{2-2}
2 & 3 \rightarrow 1 \\ \cline{2-2}
 & 1 \rightarrow 1
\end{array}$$

$$1{\times}2^3{+}1{\times}2^2{+}0{\times}2^1{+}0{\times}2^0{=}12$$

$$\begin{array}{r|l}
2 & 13 \rightarrow 1 \\ \cline{2-2}
2 & 6 \rightarrow 0 \\ \cline{2-2}
2 & 3 \rightarrow 1 \\ \cline{2-2}
 & 1 \rightarrow 1
\end{array}$$

$$1{\times}2^3{+}1{\times}2^2{+}0{\times}2^1{+}1{\times}2^0{=}13$$

Some more Example

$$\begin{array}{r|l}
2 & 127 \rightarrow 1 \\ \cline{2-2}
2 & 63 \rightarrow 1 \\ \cline{2-2}
2 & 31 \rightarrow 1 \\ \cline{2-2}
2 & 15 \rightarrow 1 \\ \cline{2-2}
2 & 7 \rightarrow 1 \\ \cline{2-2}
2 & 3 \rightarrow 1 \\ \cline{2-2}
 & 1 \rightarrow 1 \\ \cline{2-2}
\end{array}$$

$$2^7{+}2^6{+}2^5{+}2^4 + 2^3{+}2^2{+}2^1{+}2^0{=}127$$

$$\begin{array}{r|l}
2 & 224 \rightarrow 0 \\ \cline{2-2}
2 & 112 \rightarrow 0 \\ \cline{2-2}
2 & 56 \rightarrow 0 \\ \cline{2-2}
2 & 28 \rightarrow 0 \\ \cline{2-2}
2 & 14 \rightarrow 0 \\ \cline{2-2}
2 & 7 \rightarrow 1 \\ \cline{2-2}
2 & 3 \rightarrow 1 \\ \cline{2-2}
 & 1 \rightarrow 1 \\ \cline{2-2}
\end{array}$$

$$2^7{+}2^6{+}2^5{=}224$$

2.3 Octal and Hexa-Decimal Numbers

Octal and Hexa-decimal numbers are important from computer implementation point of view; Hexa decimal numbers more so because two hexa-decimal numbers can fit in the basic data structure called Byte. Below we illustrate some examples of number conversion between binary, octal and hexadecimal numbers:

Number Conversion Examples

Convert the following binary Numbers into Octal and Hexadecimal Numbers

(a) 110101_2 (b) 101111001_2 (c) 11010000100_2 (d) 11010000100_2

Binary	Octal	Hexadecimal
110101_2	$\underset{6}{\underline{110}}\ \underset{5}{\underline{101}}$	$\underset{3}{\underline{11}}\ \underset{5}{\underline{0101}}$
101111001_2	$\underset{5}{\underline{101}}\ \underset{7}{\underline{111}}\ \underset{1}{\underline{001}}$	$\underset{1}{\underline{1}}\ \underset{7}{\underline{0111}}\ \underset{9}{\underline{1001}}$
100110011010_2	$\underset{4}{\underline{100}}\ \underset{6}{\underline{110}}\ \underset{3}{\underline{011}}\ \underset{2}{\underline{010}}$	$\underset{9}{\underline{1001}}\ \underset{9}{\underline{1001}}\ \underset{A}{\underline{1010}}$
11010000100_2	$\underset{3}{\underline{011}}\ \underset{2}{\underline{010}}\ \underset{0}{\underline{000}}\ \underset{4}{\underline{100}}$	$\underset{6}{\underline{0110}}\ \underset{8}{\underline{1000}}\ \underset{4}{\underline{0100}}$

2.3.1 Decimal to Octal Conversion

Decimal to octal conversion is also performed through long division; the decimal number is repeatedly divided by 8 until the integer part of the remainder is greater than **0**. The process is illustrated through examples:

$$\begin{array}{c|ll}
8 & 359 & 44.875 \longrightarrow 0.875 \times 8 = 7 \\ \cline{2-2}
8 & 44 & 5.5 \longrightarrow 0.5 \times 8 = 4 \\ \cline{2-2}
8 & \underset{\uparrow}{0}.625 & 0 \longrightarrow 0.625 \times 8 = 5 \\ \cline{2-2}
 & 0 & \underset{\text{MSB}}{\underline{5}}\ \ \underline{4}\ \ \underset{\text{LSB}}{\underline{7}}
\end{array}$$

$$5{\times}8^2{+}4{\times}8^1{+}7{\times}8^0{=}359_{10}$$

$$\begin{array}{c|ll}
8 & 1276 & 159.5 \longrightarrow 0.5 \times 8 = 4 \\ \cline{2-2}
8 & 159 & 19.875 \longrightarrow 0.875 \times 8 = 7 \\ \cline{2-2}
8 & 19 & 2.375 \longrightarrow 0.375 \times 8 = 3 \\ \cline{2-2}
8 & 2 & 0.25 \longrightarrow 0.25 \times 8 = 2 \\ \cline{2-2}
 & 0 & \underset{\text{MSB}}{\underline{2}}\ \ \underline{3}\ \ \underline{7}\ \ \underset{\text{LSB}}{\underline{4}}
\end{array}$$

$$2{\times}8^3{+}3{\times}8^2{+}7{\times}8^1{+}4{\times}8^0{=}1276_{10}$$

Note!

Conversion from Decimals to all other formats requires long division.

2.3.2 Decimal to Hexadecimal Conversion

Decimal numbers can be converted to Hexadecimal numbers through the process of long division as discussed above. Some more examples to illustrate the process.

$$
\begin{array}{r|ll}
16 & 3018 & 188.625 \longrightarrow 0.625 \times 16 = 10 \\ \cline{2-2}
16 & 188 & 11.75 \longrightarrow 0.75 \times 16 = 12 \\ \cline{2-2}
16 & 11 & \underset{\uparrow}{0}.6875 \longrightarrow 0.6875 \times 16 = 11 \\ \cline{2-2}
 & 0 & \underset{\text{MSB}}{\underline{B}} \; \underline{C} \; \underset{\text{LSB}}{\underline{A}}
\end{array}
\qquad
\begin{array}{r|ll}
16 & 2576 & 161 \longrightarrow 0 \times 16 = 0 \\ \cline{2-2}
16 & 161 & 10.0625 \longrightarrow 0.0625 \times 16 = 1 \\ \cline{2-2}
16 & 10 & 0.625 \longrightarrow 0.6255 \times 16 = 10 \\ \cline{2-2}
 & 0 & \underset{\text{MSB}}{\underline{A}} \quad \underline{1} \quad \underset{\text{LSB}}{\underline{0}}
\end{array}
$$

Note!
$CEBFA98$ Want to discuss how a byte constitutes of number.

$$10\times16^2+12\times16^1+10\times16^0=3018_{10}$$

$$10\times16^2+1\times16^1+0\times16^0=2576_{10}$$

Summary

Conversion of numbers from one format to another

	Binary	Octal	Decimal	HexaDecimal
Binary	–	pair 3 bits	$\sum_i a_i 2^i$	pair 4 bit
Octal	convert each dig.	–	$\sum_i a_i 8^i$	regroup
Decimal	Long div.	Long div.	–	Long div.
HexaDecimal	convert each dig.	regroup	$\sum_i a_i 16^i$	–

Note!

Plz talk about composition of byte.

2.4 Arithmetic Operations

Arithmetic operations with numbers in base r follow the same rules as for decimal numbers. When other than the familiar base 10 is used, be careful to use only the r allowable digits. Examples of addition, subtraction and multiplication of two binary numbers are as follows:

$$\begin{array}{lccccccc}
\text{augend:} & & 1 & 0 & 1 & 1 & 0 & 1 \\
\text{addend:} & + & 1 & 0 & 0 & 1 & 1 & 1 \\
\hline
\text{sum:} & 1 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
\qquad
\begin{array}{lccccccc}
\text{minuend:} & & 1 & 0 & 1 & 1 & 0 & 1 \\
\text{subtrahend:} & - & 1 & 0 & 0 & 1 & 1 & 1 \\
\hline
\text{difference:} & & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

$$\begin{array}{lccccccc}
\text{multiplicand:} & & & & 1 & 0 & 1 & 1 \\
\text{multiplier:} & & & \times & & 1 & 0 & 1 \\
\hline
 & & & & 1 & 0 & 1 & 1 \\
 & & & 0 & 0 & 0 & 0 & \\
 & + & 1 & 0 & 1 & 1 & & \\
\hline
\text{product:} & & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
\end{array}$$

2.5 Complement Codes

Binary numbers are positive numbers, to perform arithmetic operations such as subtraction, we need to process the numbers so that the system stays intact. The first way to represent positive and negative numbers is to use the most significant bit of a binary number as sign-bit; while using the rest of bits to represent magnitude.

However, Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r- system: the radix complement and the diminished radix complement. When the value of the base r is substituted in the name, the two types are referred to as the 2‘s complement and 1‘s complement for binary numbers, and the 10‘s complement and 9‘s complement for decimal numbers.

The ones and twos complement of a binary number are important because they permit representation of negative numbers. The method of 2’s complement arithmetic is commonly used in computers for handling negative numbers.
The 1’s complement of a binary number is found by changing all 1s to 0s and 0s to 1s as illustrated below

8-bit Binary Number	10110010
	↓↓↓↓↓↓↓↓
1s complement	01001101

The 2’s complement of a binary number is obtained by adding 1 to the LSB of the 1s complement.

$$2s\text{complement} = (\text{1s complement}) + 1$$

2.5.1 Signed Numbers

A digital system should be able to handle both positive and negative numbers. A signed binary number consists of both sign and magnitude information. The sign indicates whether a number is positive or negative and the magnitude is the value of the number. There are forms in which signed integer (whole) numbers can be represented in binary, sign-magnitude, 1s and 2s complement.

With sign-magnitude representation left most bit is the **sign**-bit, which tell whether a number is positive or negative.

25	00011001
-25	10011001

Subtraction with Complements

The size of a number which can be represented by an $n-$bit digit is 2^n. For 2‘s complement signed numbers, the range of values for an $n-$bit numbers is

$$\text{Range} = -(2^{n-1}) \text{ to} + (2^{n-1} - 1)$$

Note!
Two’s complement code provides the widest number representation.

The comparison between 8-bit binary numbers in 1‘s and 2‘s complement notation is illustrated in Fig. 2.1.

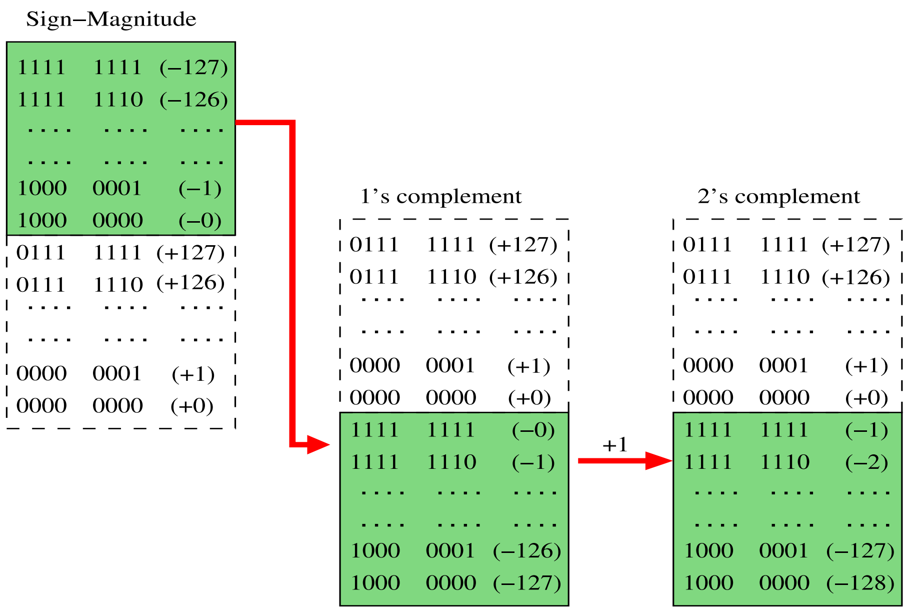


Figure 2.1: Geographical representation of 1‘s and 2‘s complement codes.

2.6 Binary Codes

Binary Coded Decimals

The Humans prefer decimal numbers; however computers and other electronic device operate with binary codes, Binary coded decimals is a suitable arrangement

Table 2.2: Table outlining different binary number formatting techniques.

number	sign magn	2's compl.	offset bin.	1's compl.
+127	01111111	01111111	11111111	01111111
+126	01111110	01111110	11111110	01111110
+125	01111101	01111101	11111101	01111101
$\vdots$	$\dots$	$\dots$	$\dots$	$\vdots$
$\vdots$	$\dots$	$\dots$	$\dots$	$\dots$
+2	00000010	00000010	10000010	00000010
+1	00000001	00000001	10000001	00000001
+0	00000000	00000000	10000000	00000000
-0	10000000	00000000	10000000	11111111
−1	10000001	11111111	01111111	11111110
−2	10000010	11111110	01111110	11111101
−3	10000011	11111101	01111101	11111100
$\vdots$	$\dots$	$\dots$	$\dots$	$\vdots$
$\vdots$	$\dots$	$\dots$	$\dots$	$\dots$
−126	11111110	10000010	00000010	10000001
−127	11111111	10000001	00000001	10000000
−128		10000000	00000000	

which allows for convenience of user interface as well as programmability. In BCD codes each number is represented as set of 4 binary digits. With 4 digits $2^4 = 16$ digits are possible however on 10 of them are valid i.e. BCD codes run from 0000_2 (0_{10}) to 1001_2 (9_{10}); numbers from 1010_2 to 1111_2 are classed as forbidden numbers. The main advantage of BCD codes is that it allows for efficient conversion between decimal and binary forms, however it is also wasteful as several states are not used.

Table 2.3: Different Numbering Formats.

Decimal	BCD	8421	2421	84-2-1	Excess-3	5211
00	0000	0000	0000	0000	0011	0000
01	0001	0001	0001	0111	0100	0001
02	0010	0010	0010	0110	0101	0011
03	0011	0011	0011	0101	0110	0101
04	0100	0100	0100	0100	0111	0111
05	0101	0101	1011	1011	1000	1000
06	0110	0110	1100	1010	1001	1010
07	0111	0111	1101	1001	1010	1100
08	1000	1000	1110	1000	1011	1110
09	1001	1001	1111	1111	1100	1111

BCD has special importance in digital displays.

Example: Addition of BCD formatted numbers

(a)	0011+0100	(b)	00100011+00010101
(c)	10000110+00010011	(d)	010001010000+010000010111

$$\begin{array}{r}0011\\+0100\\\hline 0111\end{array}\quad\begin{array}{r}3\\+4\\\hline 7\end{array}\qquad\qquad\begin{array}{rr}0010&0011\\+0001&0101\\\hline 0011&1000\end{array}\quad\begin{array}{r}23\\+15\\\hline 38\end{array}$$

$$\begin{array}{rr}1000&0110\\+0001&0011\\\hline 1001&1001\end{array}\quad\begin{array}{r}86\\+13\\\hline 99\end{array}\qquad\qquad\begin{array}{rrr}0100&0101&0000\\+0100&0001&0111\\\hline 1000&0110&0111\end{array}\quad\begin{array}{r}450\\+417\\\hline 867\end{array}$$

Example: 357_{10} in BCD would be represented as 0011 0101 0111_{BCD}. Key limitations also include that addition / subtraction of BCD number is not straight forward, BCD number should not be mixed with binary numbers, addition coding is required to format the numbers, Each byte contains two BCD digits.

84-2-1 Codes

As illustrated in Table 2.3 84-2-1 Codes are just another variant of number representation.

Gray Codes

In Gray code two consecutive codes differ by only one bit. Gray codes have very special place in digital electronics.

Table 2.4: Gray Coded Binary Numbers.

Decimal	Binary	Gray Code
00	0000	0000
01	0001	0001
02	0010	0011
03	0011	0010
04	0100	0110
05	0101	0111
06	0110	0101
07	0111	0100
08	1000	1100
09	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Excess-3 Codes

Excess-3 code is an example of it and it is an important 4 bit code. The excess – 3 code of a decimal number is achieved by adding the number 3 to the 8421 code.

Some important classifications

Weighted / Non-Weighted Codes: Some of the codes will not follow the weights of the sequence binary numbers these are called as non-weighted codes. ASCII code and Grey code are some of the examples where they are coded for some special purpose applications and they do not follow the weighted binary number calculations.
Sequential Codes: Sequential codes are the codes in which 2 subsequent numbers in binary representation differ by only one digit. The 8421 and Excess-3 codes are examples of sequential codes. 2421 and 5211 codes do not come under sequential codes.
Reflective Code:It can be observed that in the 2421 and 5211 codes, the code for decimal 9 is the complement of the code for decimal 0, the code for decimal 8 is the complement of the code for decimal 1, the code for decimal 7 is the complement of the code for decimal 2, the code for decimal 6 is the complement of the code for decimal 3, the code for decimal 5 is the complement of the code for decimal 4. These codes are called as Reflective Codes.

2.7 Parity Check

The Parity Check is widely used code used for detecting one-and two-bit transmission error in digital communication.

Table 2.5: Gray Coded Binary Numbers.

Even Parity		Odd Parity	
P	BCD	P	BCD
0	0000	1	0000
1	0001	0	0001
1	0010	1	0010
0	0011	0	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

The parity bits can be attached to the code at either the beginning or at the end, depending on the system design. Notice that the total number of 1s, including the parity bit, is always even for even parity and odd for odd parity.

2.8 ASCII Codes

Table 2.6: Table of ASCII Codes.

Decimal	Hex	Value	Decimal	Hex	Value	Decimal	Hex	Value	Decimal	Hex	Value
000	000	NUL	032	020	SP	065	041	A	097	061	a
001	001	SOH	033	021	!	066	042	B	098	062	b
002	002	STX	034	022	"	067	043	C	099	063	c
003	003	ETX	035	023	#	068	044	D	100	064	d
004	004	EOT	036	024	$	069	045	E	101	065	e
005	005	ENQ	037	025	%	070	046	F	102	066	f
006	006	ACK	038	026	&	071	047	G	103	067	g
007	007	BEL	039	027	’	072	048	H	104	068	h
008	008	BS	040	028	(	073	049	I	105	069	i
009	009	HT	041	029	)	074	04A	J	106	06A	j
010	00A	LF	042	02A	*	075	04B	K	107	06B	k
011	00B	VT	043	02B	+	076	04C	L	108	06C	l
012	00C	FF	044	02C	,	077	04D	M	109	06D	m
013	00D	CR	045	02D	-	078	04E	N	110	06E	n
014	00E	SO	046	02E	.	079	04F	O	111	06F	o
015	00F	SI	047	02F	/	080	050	P	112	070	p
016	010	DLE	048	030	0	081	051	Q	113	071	q
017	011	DC1	049	031	1	082	052	R	114	072	r
018	012	DC2	050	032	2	083	053	S	115	073	s
019	013	DC3	051	033	3	084	054	T	116	074	t
020	014	DC4	052	034	4	085	055	U	117	075	u
021	015	NAK	053	035	5	086	056	V	118	076	v
022	016	SYN	054	036	6	087	057	W	119	077	w
023	017	ETB	055	037	7	088	058	X	120	078	x
024	018	CAN	056	038	8	089	059	Y	121	079	y
025	019	EM	057	039	9	090	05A	Z	122	07A	z
026	01A	SUB	059	03B	;	091	05B	[	123	07B	{
027	01B	ESC	060	03C	<	092	05C	\	124	07C	|
028	01C	FS	061	03D	=	093	05D	]	125	07D	}
029	01D	GS	062	03E	>	094	05E	ˆ	126	07E	∼
030	01E	RS	063	03F	?	095	05F	_	127	07F	DEL
031	01F	US	064	040	@	096	060	‘			

2.9 Number Formats

The size of number which can be represented by a data structure primarily depends on its format. The numbers can be categorized into two formats namely fixed-point and floating-point numbers.

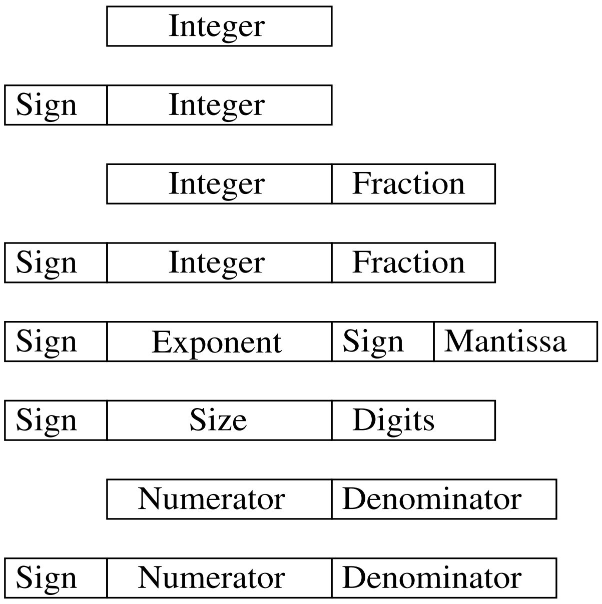


Figure 2.2: Geographical representation of Fixed and Floating point number formats .

Fixed Point Numbers

Fixed point numbers as apparent from their name have a fixed size of bits, usually the most significant bit is the sign bit while the remaining bits carry the information about the magnitude. Commonly used fixed-point variables include byte (1 byte / 8 bits), Integer (2-4 bytes/ 16-32 bits) and long type variables have (8 bytes / 64 bits).

Floating Point Numbers

To represent a very large integer (whole) number, many many bits are required. There is also a problem when number has both integer and fractional parts such as 23.5618 need to be represented. The floating-point number system, based on scientific notation, is capable of representing very large and very small numbers without an increase in number of bits and is also capable of representing numbers that have both integer and fractional components.

A Floating-point number consists of two parts and sign.. The **mantissa** is the part of the floating-point number that represents the magnitude of the number and is between **0** and **1**. The exponent is the part of a floating-point number that represents the number of places the decimal point needs to be moved.

Consider a number 241,506,800

In Floating Point Representation

mantissa 0.241506800 exponent is 9.

IEEE standard for floating point Format

Institute of Electrical & Electronics Engineers has produced a standard for floating point format arithmetic (i.e. ANSI / IEEE 754-1985). This standard specifies single precision (32bit), double precision (64bit) and quadruple precision (128bit) floating point numbers are to be represented and how arithmetic operation is to be performed.

Sign	Exponent	Fraction

In the fractional part, the binary point is understood to be on the left of 23 bits. Effectively, there are 24 bits because in any binary number the left most bit (most significant bit) is always one; therefore it is understood to be there although it does not occupy an actual bit position.

The eight bits in the exponent represent *biased exponent*, which is obtained by adding 127 to the actual exponent. The purpose of the bias is to allow very large and very small numbers without requiring another sign bit for exponent. The biased exponent allows a range of actual exponent values from -126+128.

Example: Consider 1011010010001

expressing number in fractional form $1.011010010001 \times 2^{12}$ Assuming a positive number $S = 0$, the exponent is $\underline{1}2$ and therefore the biased-exponent would be $127 + 12 = 139$ i.e. 10001011 in binary. The fractional part of the number .011010010001, because there is always a 1 to the left of binary point its not mentioned in the mantissa. The complete floating point number would be:

$$\boxed{\boxed{0}\ \boxed{10001011}\ \boxed{011010010001}}$$

Example: Consider

$$\boxed{\boxed{1}\ \boxed{10010001}\ \boxed{10001110001000000000000}}$$

$$\text{Number} = (-1)^S(1 + F) \times 2^{(E-127)}$$

inserting the details of the number above we have

$$\begin{aligned}\text{Number} &= (-1)^1(1.10001110001000000000000) \times 2^{(145-127)}\\ &-1 \times (1.10001110001000000000000)2^{18} = -1100011100010000000\end{aligned}$$

The floating point binary is equivalent of -407,688 in decimal.

A 32-point floating point number can replace a binary integer having size of 129 bits. Since the exponent determines the position of binary point, numbers containing both integer and fractional parts can be represented.

There are two exceptions to the format for floating point numbers, the number 0.0 is represented by all 0s and infinity is represented by all 1s in the exponents and all 0s in the mantissa.

Chapter 3

Boolean Algebra & Logic Gates

The objective of this chapter is to study Combinatorial Logic, Analysis and Synthesis of Boolean expressions to achieve simple circuits.

Ⓐ Basic Definitions	Ⓓ Boolean Functions
Ⓑ Axioms of Boolean Algebra	Ⓔ Canonical and Standard forms
Ⓒ Theorems & Properties	Ⓕ Simplification Techniques

This chapter concerns itself with the fundamental logic operator (i.e. the logic gates) and their combination to create complex logic circuits i.e. *combinatorial circuits* . The chapter briefly discusses the axioms of Boolean algebra, the techniques to simplify boolean expressions and physical limitations effecting the performance of logic gates.

3.1 Logic Gates

Inverter

The inverter (Not Circuit) performs the operation called *inversion* or *complementation.* The inverter changes one logic level to the opposite level. In terms of bits, it changes 1 to a 0 and 0 to a 1.
The standard logic symbol for inverter and its Truth Table is illustrated below:

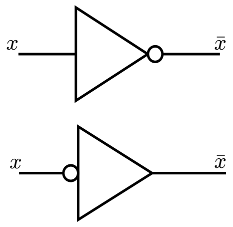


Figure 3.1: Logic symbol for NOT-gate.

Truth Table

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

Table 3.1: Truth Table for NOT-gate.

And-Gate

An And gate produces a HIGH output only when all the inputs are HIGH. When any of the inputs is LOW, the output is LOW. Therefore, the basic purpose of an AND gate is to determine when certain conditions are simultaneously true.
The operation of a 2-input AND gate can be expressed as:

$$X = AB \tag{3.1}$$

The standard logic symbol for AND-gate and its Truth-Table is illustrated below:

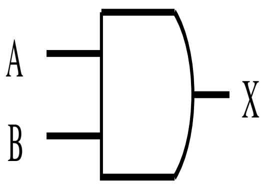


Figure 3.2: Logic symbol for AND-gate.

Truth Table

A	B	$X = AB$
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.2: Truth Table for AND-gate.

Or-Gate

An OR gate produces a HIGH output only when any one the inputs are HIGH. Therefore, the basic purpose of an OR gate is to determine when any of the conditions is true.
The operation of a 2-input OR gate can be expressed as:

$$X = A + B \tag{3.2}$$

The standard logic symbol for OR-gate and its truth-table is illustrated in figure below:

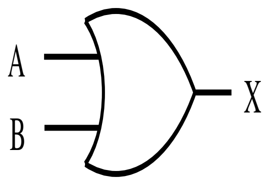


Figure 3.3: Logic symbol for OR-gate.

Truth Table

A	B	$X = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.3: Truth Table for OR-gate.

NAND-Gate

The NAND gate is popular logic element because it can be used as a universal gate; that is NAND gates can be used in combination to perform the AND, OR and inverter operations.
A NAND gate produces a LOW output only when all the inputs are HIGH. When any of the inputs is LOW, the output will be HIGH.

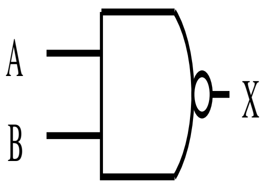


Figure 3.4: Logic symbol for NAND-gate.

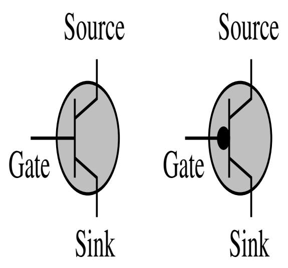
Truth Table

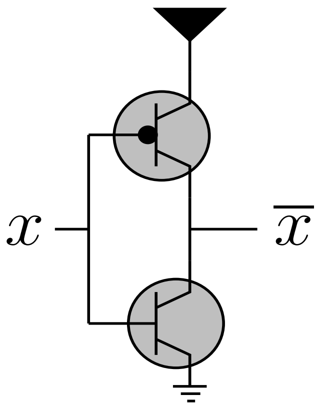
A	B	$X = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

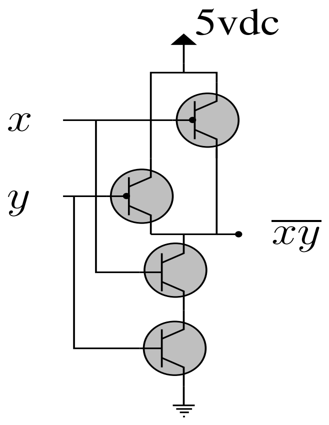
Table 3.4: Truth Table for NAND-gate.

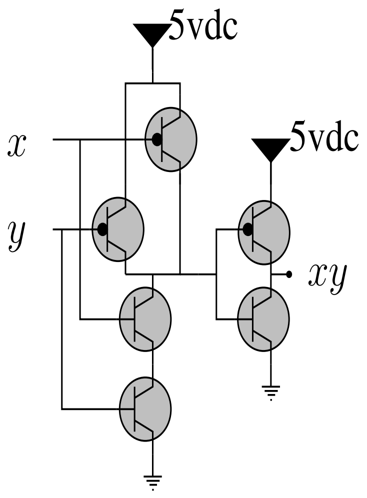
Design of Logic Gates at Transistor Level

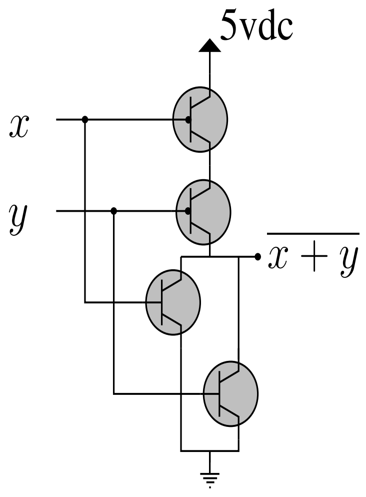
All logic gates are composed of transistors. The transistors operate with active low or active high input. From the pictures it is apparent that a gate of inverter logic requires two transistors.









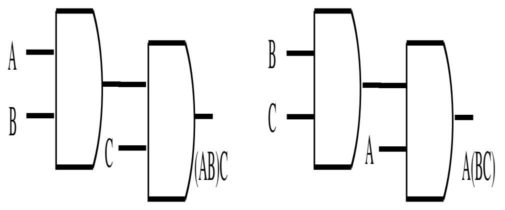


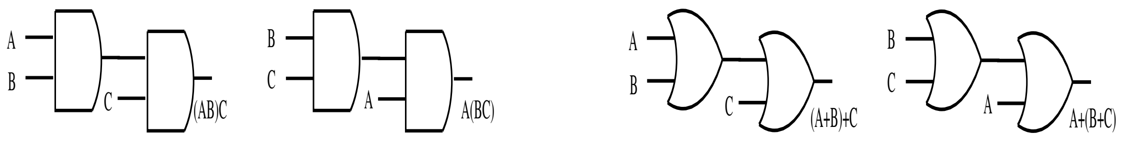
3.2 Axioms of Boolean Algebra

The basic laws of Boolean Algebra are enumerated below:

Commutative Laws: The commutative law of addition for two variables is written as

$$A + B = B + A$$





which means it makes no difference in which way variables are ORed. The commutative law of multiplication for two variables is written as

$$AB = BA$$

Associative Laws: The associate law of addition is written as follows for three variables

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

Similarly, the associate law of multiplication is written as follows for three variables

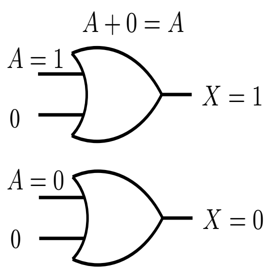
$$A(BC) = (AB)C$$

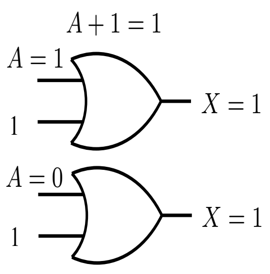
Distributive Law: The associate law of addition is written as follows for three variables

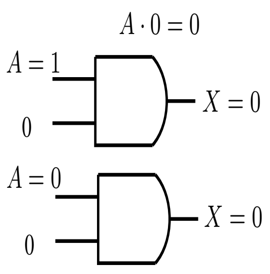
$$A(B + C) = AB + AC$$

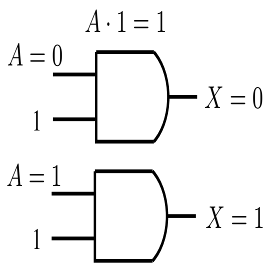
Key Properties of Boolean Algebra

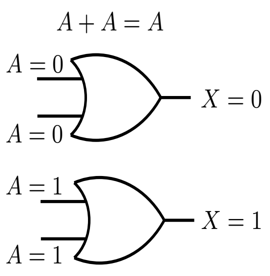
1. $A + 0 = A$
2. $A + 1 = 1$
3. $A \cdot 0 = 0$
4. $A \cdot 1 = A$
5. $A + A = A$
6. $A + \bar{A} = 1$
7. $A \cdot A = A$
8. $A \cdot \bar{A} = 0$
9. $\bar{\bar{A}} = A$
10. $A + AB = A$
11. $A + \bar{A}B = A + B$
12. $(A + B)(A + C) = A + BC$

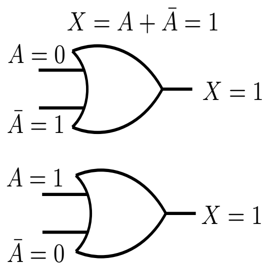


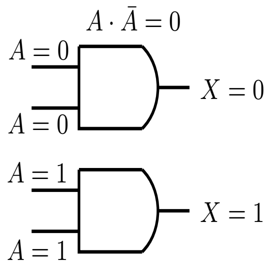


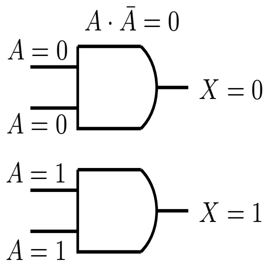


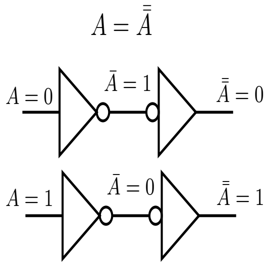


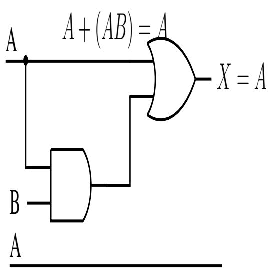


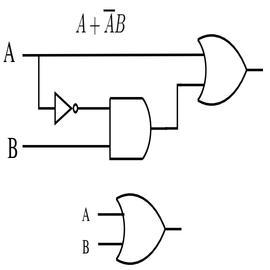


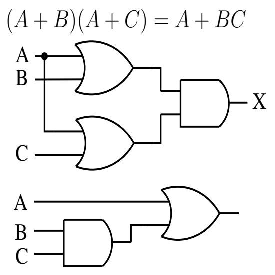










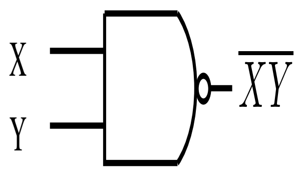


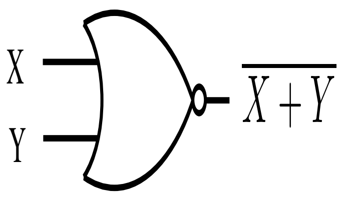
3.2.1 DeMorgan's Theorem

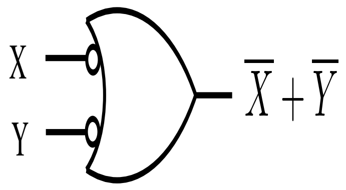
DeMorgan, a mathematician who knew Bool, proposed two theorems that are important part of the Boolean Algebra. The theorems are stated below:

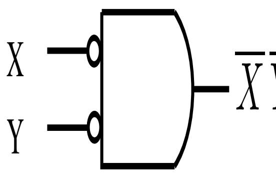
$$\overline{X\,Y} = \bar{X} + \bar{Y}$$

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









These statements can be easily verified with the help of Truth Tables.

Examples:

a. $\overline{(A+B+C)D}$

using the identity $\overline{X\,Y} = \overline{X} + \overline{Y}$, we may write

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

Again applying DeMorgan's theorem to $\overline{(A+B+C)} = \overline{A}\,\overline{B}\,\overline{C}$

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

b. $\overline{ABC + DEF}$

using the identity $\overline{X+Y} = \overline{X}\,\overline{Y}$, we may write,

$$\overline{ABC + DEF} = (\overline{ABC})(\overline{DEF})$$

using identity $\overline{XY} = \overline{X} + \overline{Y}$

$$(\overline{ABC})(\overline{DEF}) = (\overline{A} + \overline{B} + \overline{C})(\overline{D} + \overline{E} + \overline{F})$$

c. $\overline{A\overline{B} + \overline{C}D + EF}$

using the identity $\overline{X + Y + Z} = \overline{X}\,\overline{Y}\,\overline{Z}$, we may write

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

applying DeMorgan's theorem to individual components, we have

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

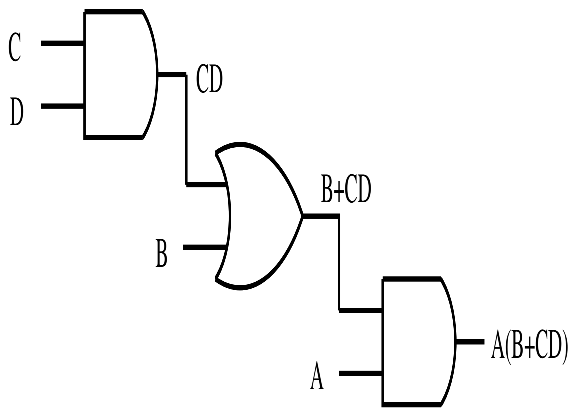
3.2.2 Simplification Using Boolean Algebra

A logical expression can be reduced to its simplest form or changed to more convenient form to implement the expression most efficiently using Boolean algebra. This section demonstrates the application of basic laws, rules and theorems of boolean algebra to manipulate and simplify an expression.

Analysis of Logic Circuit with Truth Table

Once the Boolean expression for a given logic circuit has been determined, truth table shows the output for all possible values of the input variables. The number of possible i input combination is 2^n, where n is the number of input variables.

A	B	C	D	A(B+CD)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



Example:Using Boolean Algebra techniques, simplify this expression:

$$AB + A(B + C) + B(B + C) \tag{3.3}$$

Step 1: Apply distributive law to expand the 2nd and 3rd terms of the expression

$$AB + AB + AC + BB + BC$$

Step 2: According to Rule 7 $A \cdot A = A$

$$AB + AB + AC + B + BC$$

Step 3: According to Rule 5 $AB + AB = AB$

$$AB + AC + B + BC$$

Step 4: According to Rule 10 $AB + B = B$

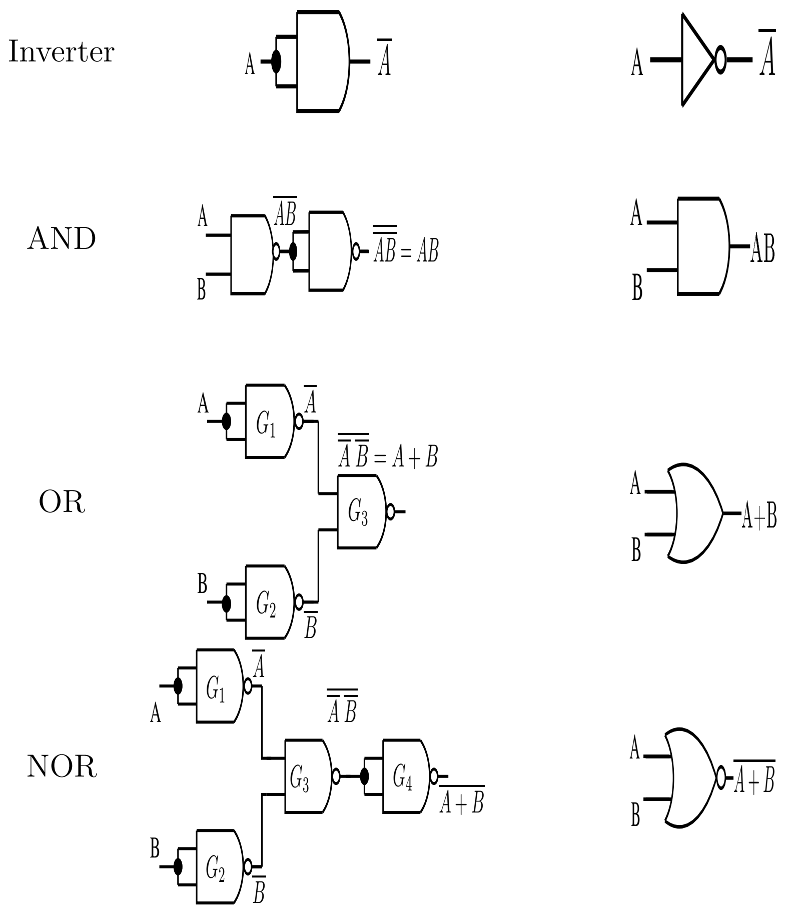
$$B + AC$$

3.3 Universal Gates

The universality of NAND and NOR gates means that they can be used as an inverter and their combinations can be used as AND / OR gates.

3.3.1 The NAND Gate as Universal Logic Element

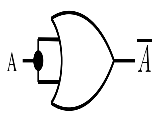
The NAND gate is a universal gate because it can be used to produce NOT, AND, OR and NOR functions; These implementations are illustrated below:

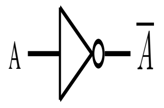


3.3.2 The NOR Gate as Universal Logic Element

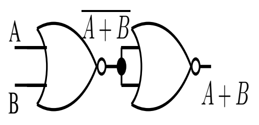
Like NAND gate NOR gate can be used to produce NOT, AND, OR and NAND functions; These implementations are illustrated below:

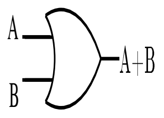
Inverter



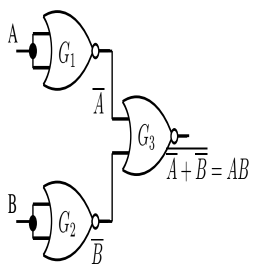


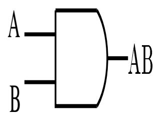
OR



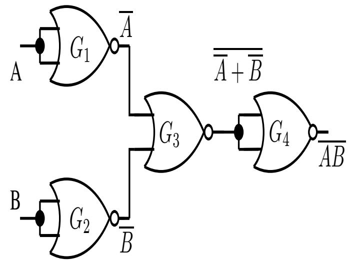


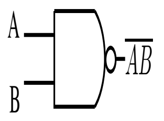
AND





NAND





3.4 Canonical and Standard Forms

All boolean expressions regardless of their form can be converted into either of two standard form: the sum-of-products and the product-of-sums. Standardization of boolean expression makes simplification much more systematic and straight forward.

3.4.1 Sum of Products Notation

When two of more products terms are summed by boolean addition, the resulting expression is a **sum-of-products** (SOP); Some examples are

$$A + ABC$$

$$ABC + CDE + \bar{B}C\bar{D}$$

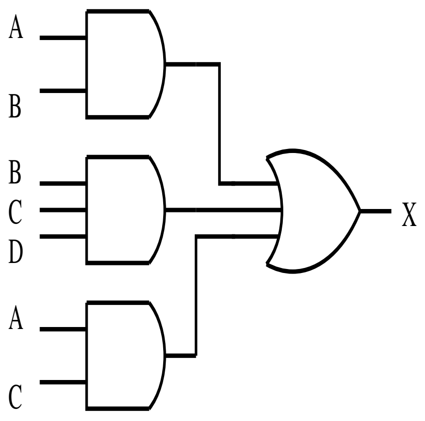
$$\bar{A}B + \bar{A}B\bar{C} + AC$$

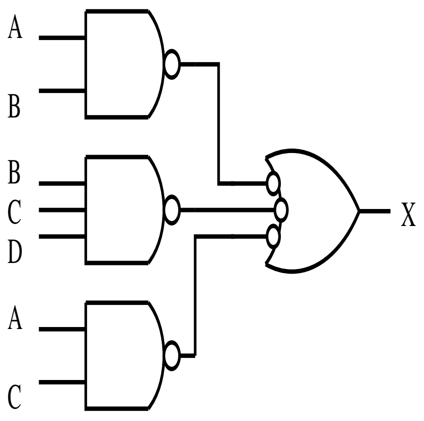
a. A SOP expression can have a single-variable term term.

b. After simplification the final expression would be a single expression or SOP.

c. Any of the variables can have bars e.g. $\bar{A}\bar{B}\bar{C}$, however $\overline{ABC}$ is not allowed

d. SOP expression can be implemented by ORing the outputs produced from ANDing of each term. This can be implemented as follows:





Standard SOP expression is in one which all the variables in the domain appear in each product term in expression. For example $A\bar{B}CD + \bar{A}\bar{B}C\bar{D} + AB\bar{C}\bar{D}$; Standard SOP expressions are important in constructing truth tables and Karnaugh map simplification method.

Example: $A\bar{B}C + \bar{A}\bar{B} + AB\bar{C}D$

i. First term $A\bar{B}C$ is missing D, therefore $A\bar{B}C(D + \bar{D})$ yields $A\bar{B}CD + A\bar{B}C\bar{D}$.

ii. Second term $\bar{A}\bar{B}$ is missing C and D there we will have $\bar{A}\bar{B}CD + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}C\bar{D}$

iii. The Standard SOP expression looks like:

$$A\bar{B}CD + A\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + AB\bar{C}D$$

3.4.2 Product of Sums Notation

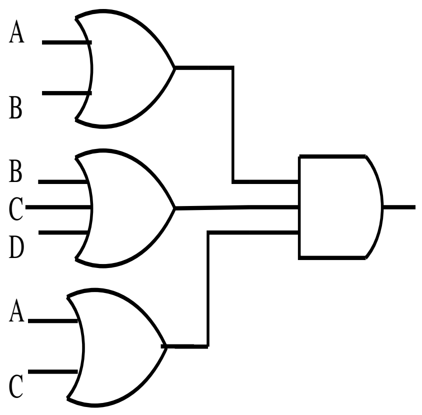
When two or more sum terms are multiplied, the resulting expression is a **Product-of-sums** (POS); Some examples are

$$(\bar{A} + B)(A + \bar{B} + C)$$
$$(\bar{A} + \bar{B} + \bar{C})(C + \bar{D} + E)(\bar{B} + C + D)$$
$$(A + B)(A + \bar{B} + C)(\bar{A} + C)$$

a. A POS expression can have a single-variable term.

b. In a POS expression bar cannot extend over more than one variable e.g. $\bar{A}+\bar{B}+\bar{C}$, however $\overline{A + B + C}$ is not allowed.

c. POS expression can be implemented by ANDing the outputs produced from ORing of each term. This can be implemented as follows:



3.5 Simplification Techniques

Karnaugh map provides a systematic method for simplifying Boolean expressions; through its applications one case determine the simplest SOP and POS expressions for a logical circuit. The expression is known as Canonical or Minimal expression.

Karnaugh map is similar to truth table as it presents all possible values of input variables and resulting output for each combination. Karnaugh map is a powerful way of graphically visualizing behaviour of logic circuits. The method can be used most conviniently for 2, 3 and 4 variable problems.
The K-maps for 2,3 and 4 variables are illustrated below:

$$\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{B} \\
 & & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \cline{3-4}
\multirow{2}{*}{A} & 0 & & \\ \cline{3-4}
 & 1 & & \\ \cline{3-4}
\end{array}
\qquad
\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{B} \\
 & & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \cline{3-4}
\multirow{2}{*}{A} & 0 & \bar{A}\bar{B} & \bar{A}B \\ \cline{3-4}
 & 1 & A\bar{B} & AB \\ \cline{3-4}
\end{array}$$

$$\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{C} \\
 & & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \cline{3-4}
 & 00 & & \\ \cline{3-4}
 & 01 & & \\ \cline{3-4}
AB & 11 & & \\ \cline{3-4}
 & 10 & & \\ \cline{3-4}
\end{array}
\qquad
\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{C} \\
 & & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} \\ \cline{3-4}
 & 00 & \bar{A}\bar{B}\bar{C} & \bar{A}\bar{B}C \\ \cline{3-4}
 & 01 & \bar{A}B\bar{C} & \bar{A}BC \\ \cline{3-4}
AB & 11 & AB\bar{C} & ABC \\ \cline{3-4}
 & 10 & A\bar{B}\bar{C} & A\bar{B}C \\ \cline{3-4}
\end{array}$$

CD

AB	00	01	11	10
00				
01				
11				
10				

CD

AB	00	01	11	10
00	$\bar{A}\bar{B}\bar{C}\bar{D}$	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}CD$	$\bar{A}\bar{B}C\bar{D}$
01	$\bar{A}B\bar{C}\bar{D}$	$\bar{A}B\bar{C}D$	$\bar{A}BCD$	$\bar{A}BC\bar{D}$
11	$AB\bar{C}\bar{D}$	$AB\bar{C}D$	$ABCD$	$ABC\bar{D}$
10	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}\bar{C}D$	$A\bar{B}CD$	$A\bar{B}C\bar{D}$

Process of using K-MAPs

a. Select a K-map according to the total number of variables.

b. Identify maxterms or minterms as given in the problem.

c. For SOP, put the 1‘s in the blocks of the K-map with respect to the minterms (elsewhere 0‘s).

d. For POS, putting 0‘s in the blocks of the K-map with respect to the maxterms (elsewhere 1‘s).

e. Making rectangular groups that contain the total terms in the power of two, such as 2,4,8 ..(except 1) and trying to cover as many numbers of elements as we can in a single group.

f. From the groups that have been created in step (e.), find the product terms and then sum them up for the SOP form.

3.6 Timing, Power & other Considerations

The logic IC of 74 family are implemented through various semi-conductor materials and manufacturing technologies. A brief description of these variants is presented in table below:

3.6.1 Logic IC Family 74xxabc

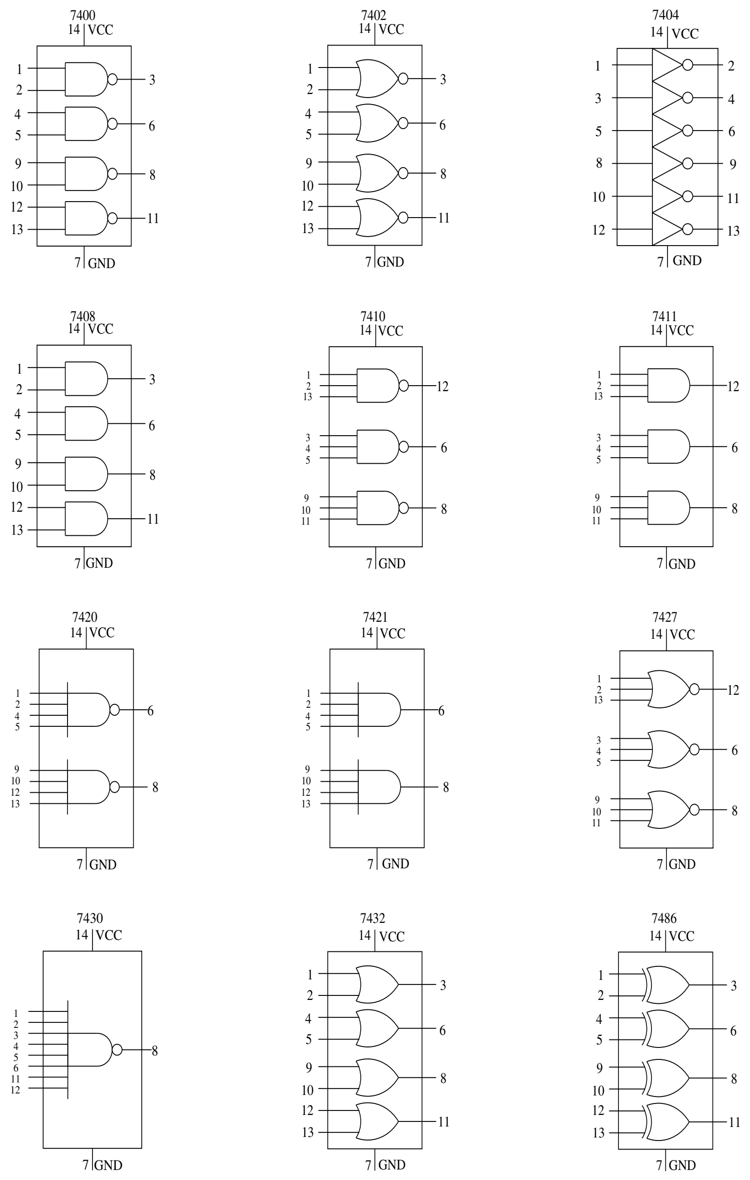
These ICs are pin compatible and come in a variety of specifications with the basic difference in semi-conductor technologies. The minimal list of the semi-conductor technologies is presented below:

Circuit Type	Description	Technology
LS	Low Power Schottky	Bipolar
ALS	Adv Low Power Schottky	Bipolar
HS	High Speed CMOS	CMOS
LVC	Low Voltage CMOS	CMOS
F	Fast	Bipolar

The 74xx series family of IC's has hundreds of unique logic circuits, the table below, only a select few circuits are enumerated: Pin diagrams of various popular components of IC's 74xx-series are tabulated below:

Table 3.5: The Selected list of circuits from 74xx ICs family.

7400	quad 2-input NAND gate	7481	16-bit random access memory
7402	quad 2-input NOR gate	7483	4-bit binary full adder
7404	hex inverter	7484	16-bit random access memory
7408	quad 2-input AND gate	7485	4-bit magnitude comparator
7410	triple 3-input NAND gate	7486	quad 2-input XOR gate
7420	dual 4-input NAND gate	7491	8-bit shift register, serial In/out
7427	triple 3-input NOR gate	7493	4-bit binary counter
7430	8-input NAND gate	7494	4-bit shift register
7432	quad 2-input OR gate	74104	J-K master-slave flip-flop
7442	BCD to dec. decoder	74121	Monostable multi-vibrator
7444	excess-3-Gray to decimal decoder	74138	3 to 8-line demux
7456	50:1 frequency divider	74145	BCD to decimal decoder/driver
7457	60:1 frequency divider	74148	8 to 3-line priority encoder
7468	dual 4 bit decade counters	74151	8 to 1-line data mux
7469	dual 4 bit binary counters	74154	BCD to decimal decoder/driver
7470	+edge trig J-K flip-flop	74166	||-Load 8-bit shift register
7472	J-K master-slave flip-flop	74171	quad D-type flip-flops
7473	dual J-K flip-flop with clear	74180	9-bit odd/even parity gen/chk
7474	dual D +ve edge trig flip-flop	74182	lookahead carry generator
7475	4-bit bistable latch	74194	4-bit bi-direct shift register
7479	dual D flip-flop	74195	4-bit parallel-access shift register
7480	gated full adder	74198	8-bit bi-direct shift register



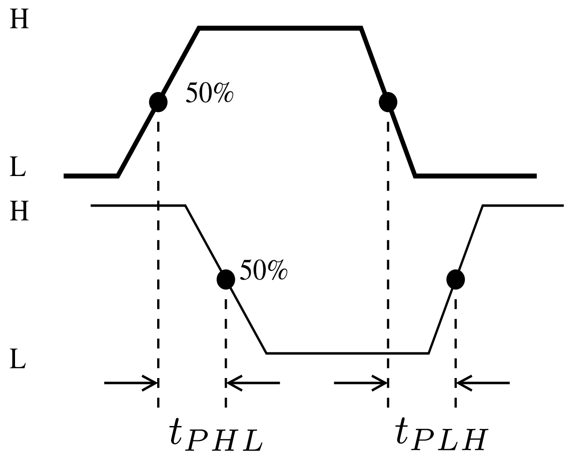
There are several parameters which define the performance of integrated circuits such as switching speed measured in terms of propagation delay time, the power dissipation, the fan out or drive capability, the speed power product and the input/output logic levels. All of these features are discussed further:

Propagation Delay Time

Note!
Propagation Delay: is a major factor limiting the processing speed of digital systems.

This parameter is a result of the limitation of switching speed or frequency at which a logic circuit can operate. The switching speed of a circuit is inversely proportional to its propagation delay. The propagation delay t_p of a logic gate is the time interval between transition of an input pulse and the occurrence of resulting transition of the output pulse. There are two different measurements of propagation delay time associated with a logic gate that apply to all the types of basic gates:

- t_{PHL}: the time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse with output changing from High Level to Low Level (HL).
- t_{PLH}: the time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse with output changing from Low Level to High Level (LH).



For HCT family CMOS, the propagation delay is 7 nS, for AC family it is 5 nS for the ALVC family it is 3 nS, for standard-family Bipolar (TTL) gates typical delay is 11 nS and for F family it is 3.3 nS.

DC Supply Voltage (V_{CC})

The typical DC supply voltage for CMOS logic is either 5V, 3.3V, 2.5V and 1.8V, depending on the category. An advantage of CMOS is that the supply voltage can vary over a wider range than for a BJT (TTL) logic. The 5V CMOS can tolerate supply variation from 2V to 6V and still operate properly although propagation delay time and power dissipation are significantly affected.

Power Dissipation

The **Power Dissipation** P_D of a logic gate is the product of the DC supply voltage and the average supply current. Normally, the supply current when the gate output is LOW is greater than when gate supply is HIGH, the average power dissipation of logic gate is

$$P_D = V_{CC}\left(\frac{I_{CCH} + I_{CCL}}{2}\right)$$

Note!
Quiescent Power: is the minimal power that is drawn by the IC no matter what.

CMOS gates have very low power dissipations compared to the bipolar family; However, the power dissipation of CMOS is dependent on the frequency of operation; At zero frequency the quiescent power is typically in microwatt/gate range, and at the maximum operating frequency it can be in low milliwatt/gate range. Therefore power is specified at a given frequency range. The HC family, for example the power of 2.75 μW/gate at 0 Hz (quiescent) and 600 μW/gate at 1 MHZ. Power dissipation of bipolar gates is independent of frequency e.g. ALS family consumes 1.4 mW/gate regardless of frequency while F family consumers 6mW/gate.

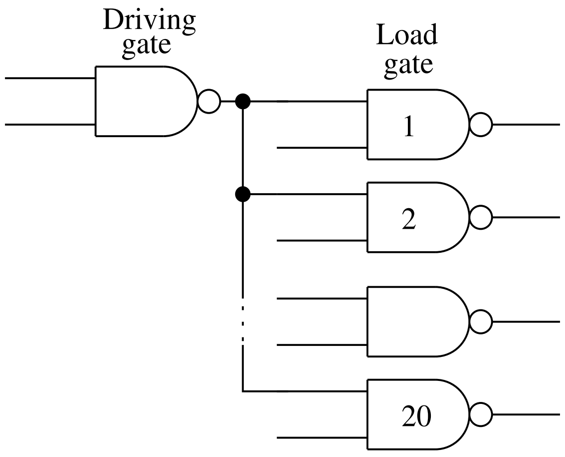
Speed-Power Product

The parameter speed-power product (SPP) can be used as a measure of a logic circuit taking into account the propagation delay time and the power dissipation. It is specially useful for comparing the various logic gate series within the CMOS and bipolar technology family.
The SPP of a logic circuit is the product of the propagation delay time and the power dissipation and is expressed in *joules*.

$$SPP = t_p P_D$$

Fan-Out and Loading

The fan-out of a logic gate is the maximum number of inputs of the same series in an IC family that can be connected to the gate's output and still maintain the output voltage.



Fan-out The number of equivalent gate inputs of the same family series that a logic gate can drive. This is special significant for Bipolar logic because of the technology. The Fan-out is specified in terms of **unit loads**. A unit load for a logic gate equals one input to a like circuit.

Chapter 4

Combinatorial Functions

This chapter presents several types of combinatorial logic functions such as adders, comparators, decoders, encoders, code converters multiplexors, demultiplexors etc. The designs are discussed through truth-tables and comparison of various possible implementations to determine the optimal implementations.

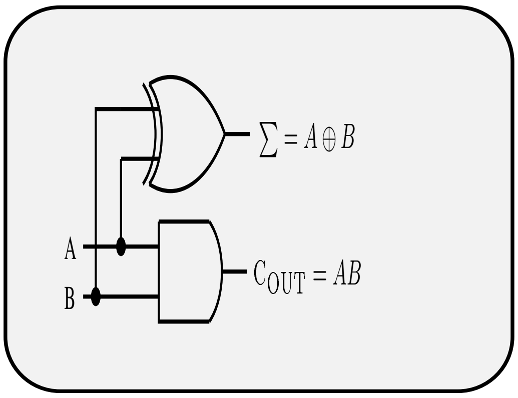
Ⓐ Types of Adders	Ⓓ Code Converters
Ⓑ Comparators	Ⓔ Multiplexors / Demultiplexors
Ⓒ Decoders / Encoders	Ⓕ Parity Generators / Checkers

4.1 Full / Half Adders

Adders are essential to manipulator and process data. The **half-adder** accepts two binary digits on its inputs and produces two binary digits on its outputs- a sum bit and a carry bit. The mathematical expression for the sum and carry bits are as follows:

$$C_{OUT} = AB$$
$$\sum = A \oplus B \qquad (4.1)$$

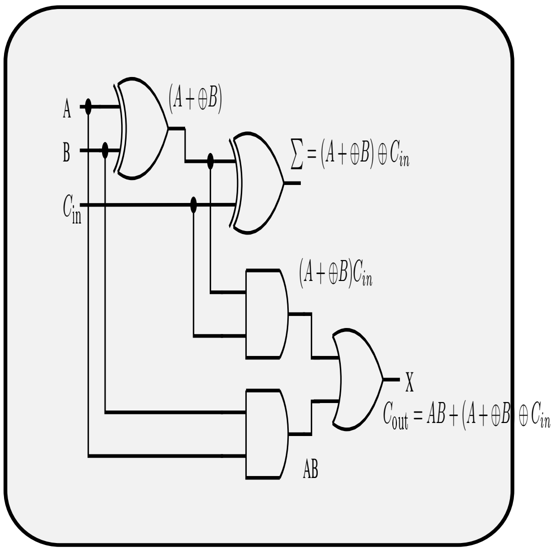
The output carry is produced as a AND-product of input A and B while the sum is generated with exclusive-OR gate as illustrated below



The **full-adder** accepts two binary digits and an input carry on goes on to generate two outputs- a sum bit and a carry bit. the mathematical expression for the sum and carry bits are as follows:

$$C_{OUT} = AB + (A \oplus B) \oplus C_{IN}$$
$$\sum = (A \oplus B) \oplus C_{IN} \qquad (4.2)$$

The output carry is produced as a AND-product of input A and B while the sum is generated with exclusive-OR gate as illustrated below



A
B
C_{in}
$(A+\oplus B)$
$\sum = (A+\oplus B) \oplus C_{in}$
$(A+\oplus B)C_{in}$
X
$C_{out} = AB + (A+\oplus B \oplus C_{in}$
AB

The truth-table for Half and Full adders are tabulated below:

A	B	C_{out}	$\sum$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

A	B	C_{in}	C_{out}	$\sum$
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

Two half-adders can be combined together to develop a full-adder as illustrated below:

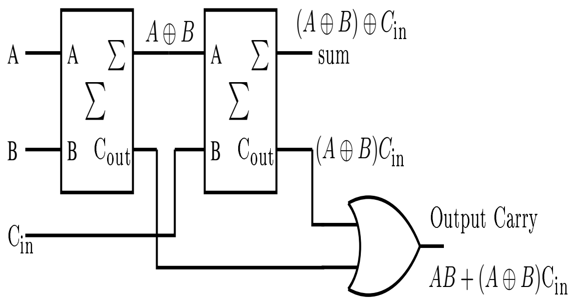


Figure 4.1: Two half adders can be combined to yield a full-adder.

4.2 Parallel Binary Adders

Two or more full-adders connected to form parallel binary adders. The carry output of each adder is connected to the carry input of next higher-order adder. The implementation of 2-bit and 4-bit parallel adders is illustrated below:

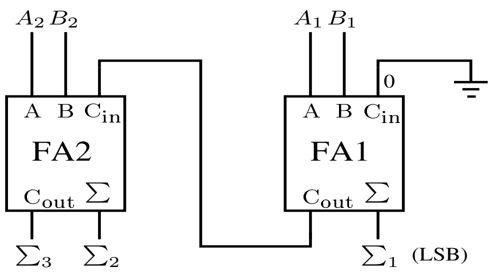


Figure 4.2: Ripple Carry Circuit with two full-adders.

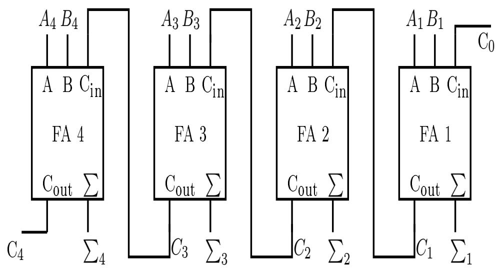


Figure 4.3: Ripple Carry Circuit with four full-adders.

4.3 Ripple Carry and Look-Ahead Carry Adders

Adders can be classified into two types, ripple carry and look-ahead carry, externally both adders are the same in-terms of inputs and outputs, however the difference is the speed at which they can add two numbers. Look-ahead adder is much faster than ripple carry adder.

The Ripple Carry Adder

The ripple carry adder is one which the carry output of each full-adder is connected to the carry input of the higher-order stage. The sum and the output carry of ay stage can not be produced until the input carry occurs; this causes a time delay in addition process illustrated in the 4.4. The carry delay propagation for each full-adder is the time from the application of the input of the carry until output carry occurs, assuming that the inputs are already present.

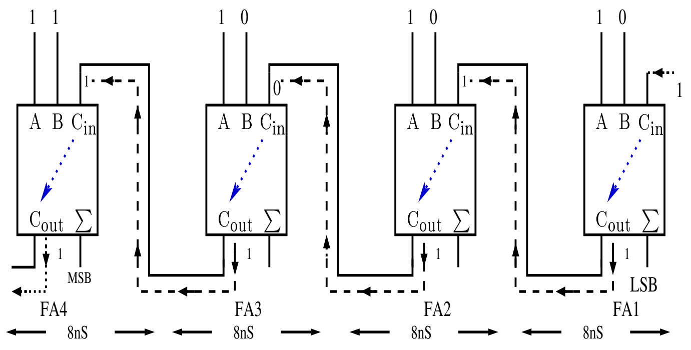


Figure 4.4: The Propagation Delay Effect of Ripple Carry Circuit.

The Look-Ahead Carry Adder

The speed with which an addition can be performed is limited by the time required for carries to propagate, or ripple through all the stages of a parallel adder. One method of speeding up the addition process by eliminating this ripple carry delay is called look-ahead carry addition. The look-ahead carry adder anticipates the output carry of each stage, and based on the inputs, produces the output carry by either carry generation or carry propagation.

Carry generation occurs when an output carry is produced (generated) internally by the full-adder. A carry is generated when both bits are 1s. The *generated carry* C_g is expressed as AND function of the two input bits, A and B.

$$C_g = AB$$

Carry Propagation occurs when the input carry is rippled to become the output carry. An input carry may be propagated by full-adder when either or both of the input bits are 1s. the *propagated carry* C_p is expressed as the OR function of the input bits.

$$C_p = A + B$$

The conditions for carry generation and carry propagation are illustrated in 4.5.

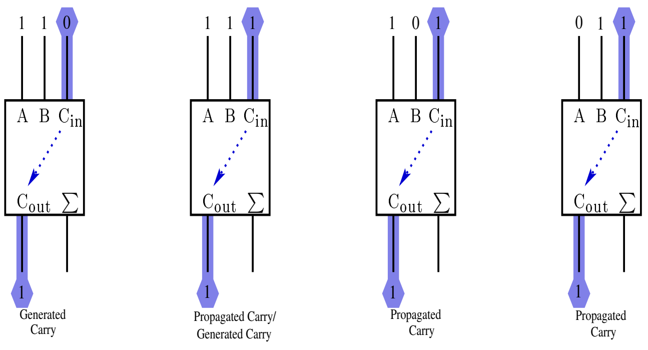


Figure 4.5: Conditions of Carry Generation and Carry Propagation.

The output carry of a full-adder can be expressed in terms of both the generated carry C_g and propagated carry C_p. The output carry C_{out} is a 1 if the generated carry is a 1 AND the input carry C_{in} is a 1.

Based on this analysis, we can now develop expression for the output carry $\mathrm{C_{out}}$, of each full-adder stage for the 4-bit adder.

Full Adder 1:

$$C_{\text{out}1} = C_{g1} + C_{p1}C_{in1} \tag{4.3}$$

Full Adder 2:

$$\begin{aligned}
C_{\text{in}2} &= C_{\text{out}1} \\
C_{\text{out}2} &= C_{g2} + C_{p2}C_{in2} \\
C_{\text{out}2} &= C_{g2} + C_{p2}(C_{g1} + C_{p1}C_{in1}) \\
&= C_{g2} + C_{p2}C_{g1} + C_{p2}C_{p1}C_{in1} \qquad (4.4)
\end{aligned}$$

Full Adder 3:

$$\begin{aligned}
C_{\text{in3}} &= C_{\text{out2}} \\
C_{\text{out3}} &= C_{g3} + C_{p3}C_{in3} \\
C_{\text{out3}} &= C_{g3} + C_{p3}(C_{g2} + C_{p2}C_{g1} + C_{p2}C_{p1}C_{in1}) \\
&= C_{g3} + C_{p3}C_{g2} + C_{p3}C_{p2}C_{p1}C_{in1} \qquad (4.5)
\end{aligned}$$

Full Adder 4:

$$\begin{aligned}
C_{\text{in4}} &= C_{\text{out3}} \\
C_{\text{out4}} &= C_{g4} + C_{p4}C_{in4} \\
C_{\text{out4}} &= C_{g4} + C_{p4}(C_{g3} + C_{p3}C_{g2} + C_{p3}C_{p2}C_{g1} + C_{p3}C_{p2}C_{p1}C_{in1}) \\
&= C_{g4} + C_{p4}C_{g3} + C_{p4}C_{p3}C_{g2} + C_{p4}C_{p3}C_{p2}C_{g1} + C_{p4}C_{p3}C_{p2}C_{p1}C_{in1} \qquad (4.6)
\end{aligned}$$

Note that in each of these expressions, the output carry of each full-adder stage is dependent only on the initial carry C_{in1}, the C_g and C_p functions of that stage and the C_g and C_p functions of the preceding stage; Since the C_g and C_p can be expressed in terms of A and B inputs to the full-adders, all the output carries are immediately available except for gate delays.

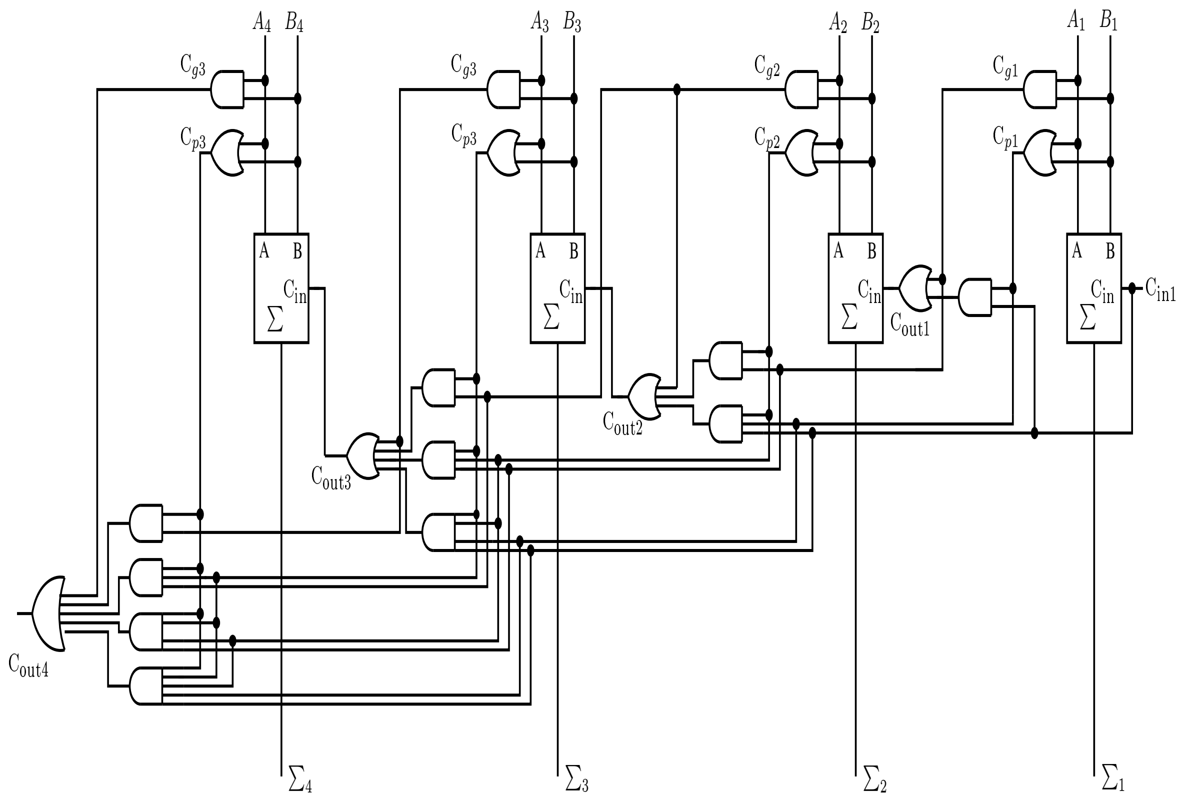


Figure 4.6: Implementation of 4-bit Look Ahead Adder using expressions (4.3)-(4.6).

4.4 Comparators

The basic function of a comparator is to compare the magnitudes of two binary quantities to determine the relationship between those quantities.

Equality

In order to compare two binary numbers containing two bits each an additional exclusive NOR gate is necessary. The two LSBs of the two numbers are compared by gate G_1 and the two MSBs are compared by gate G_2 are illustrated in the figure.

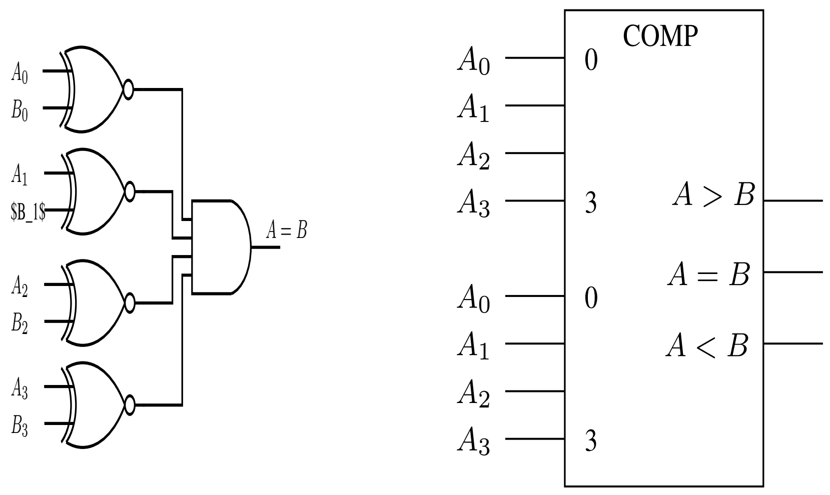


Figure 4.7: Circuit and Functional diagram of a 4-bit Comparator

Inequality

In addition to the equality output, fixed-function comparators can provide additional output that indicate which of the two binary numbers being compared is larger. That is, there is an output that indicates when number A is larger. That is, there is an output that indicates when number A is greater than number B ($A > B$) and output that indicates when number A is less than number B ($A < B$). These three operations are valid of each bit position. The general procedure used in a comparator is to check for inequality in a bit position, starting with the highest-order bits, When such inequality is found in a bit position, the relationship of the two numbers is established.

4.5 Decoder / Encoder

Basic Binary Decoder

A decoder is a digital circuit that detects the presence of a specified combination of bits (codes) on its input and indicates presence of that code by a specified output level. In general a decoder has n input lines to handle $n-$bits and one to 2^n output lines to indicate presence of one or more $n-$bit combinations. The operation of 3-line-to-8-line (1-of-8) decoder is presented in table below:

Decimal	Binary Inputs			Decoding Function	Outputs							
Digit	A_2	A_1	A_0		1	2	3	4	5	6	7	8
0	0	0	0	$\bar{A}_2\bar{A}_1\bar{A}_0$	1	0	0	0	0	0	0	0
1	0	0	1	$\bar{A}_2\bar{A}_1A_0$	0	1	0	0	0	0	0	0
2	0	1	0	$\bar{A}_2A_1\bar{A}_0$	0	0	1	0	0	0	0	0
3	0	1	1	$\bar{A}_2A_1A_0$	0	0	0	1	0	0	0	0
4	1	0	0	$A_2\bar{A}_1\bar{A}_0$	0	0	0	0	1	0	0	0
5	1	0	1	$A_2\bar{A}_1A_0$	0	0	0	0	0	1	0	0
6	1	1	0	$A_2A_1\bar{A}_0$	0	0	0	0	0	0	1	0
7	1	1	1	$A_2A_1A_0$	0	0	0	0	0	0	0	1

4.5.1 Encoder

Encoder is a combinatorial logic circuit that essentially performs the reverse decoder function. An encoder accepts an active level on one of its inputs representing a decimal or octal digit and converts it into coded output such as BCD or binary.

Implementation of BCD Code using Encoder

This type of encoder would has ten inputs-one for each decimal digits and four outputs corresponding to the BCD code, as illustrated in Fig. 4.8. Table 4.1 represents truth table of the BCD encoder. The logic circuitry required for encoding each decimal digit to a BCD code by using logic expression is illustrated as follows:

Table 4.1: Truth Table of a BCD Encoder.

Decimal Input	A_3	A_2	A_1	A_0
0	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

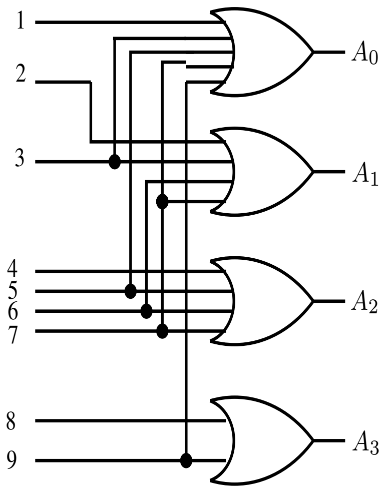


Figure 4.8: Circuit diagram of a BCD Encoder.

The Decimal-to-BCD Priority Encoder

This type of encoder performs the same basic encoding function as previously discussed. A priority encoder also offers additional flexibility in that it can used to in applications that require priority detection. This priority function means that the encoder will produce a BCD output corresponding to the highest-order decimal digit.students may be asked to design 3-bit priority encoder.

4.6 Multiplexers and Demultiplexers

A multiplexer (MUX) is a device that allows digital information from several sources to be routed onto a single line for transmission over to a common destination. The logical symbol of multiplexer is illustrated in fig 4.9. The behaviour of function

Inputs				Outputs	
D_3	D_2	D_1	D_0	B	A
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

Table 4.2: The truth-table of a 4 input/2 output priority encoder.

Data-Select Inputs		Selected
S_1	S_0	Input
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Table 4.3: Truth Table of a 4:1 Multiplexer.

of 1-of-4 multiplexer is illustrated in table 4.3 and circuit diagram is illustrated in fig.4.9

The Boolean Expression for data output can be described as:

$$Y = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1S_0 + D_2S_1\bar{S}_0 + D_3S_1S_0$$

The implementation of this expression requires 3-input AND gate and a 4-input OR gate and two inverter. The circuit is also referred to as $\underline{\text{D}}$ata Selectors.

Multiplexer Use Cases

Multiplexor is a versatile device, can be configured to work in a host of different applications, a select few applications are presented below:

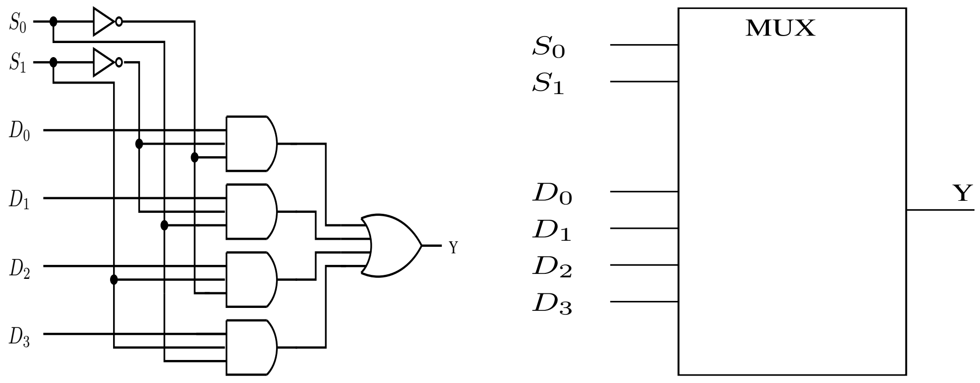
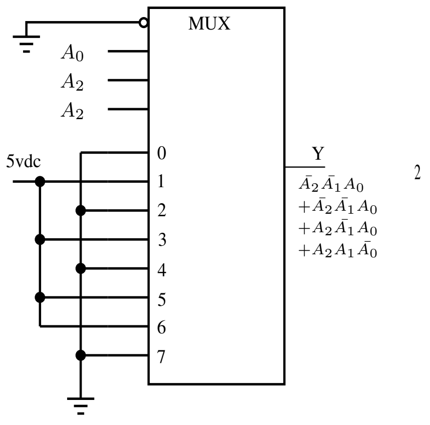
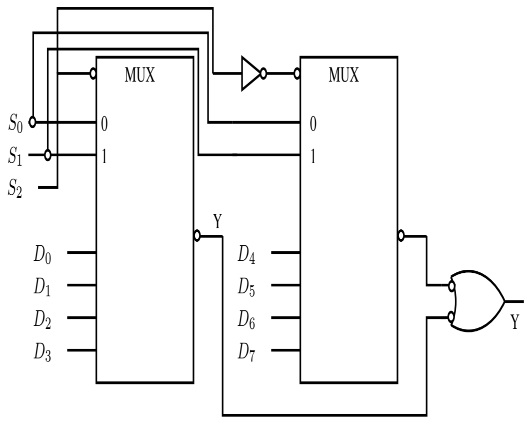
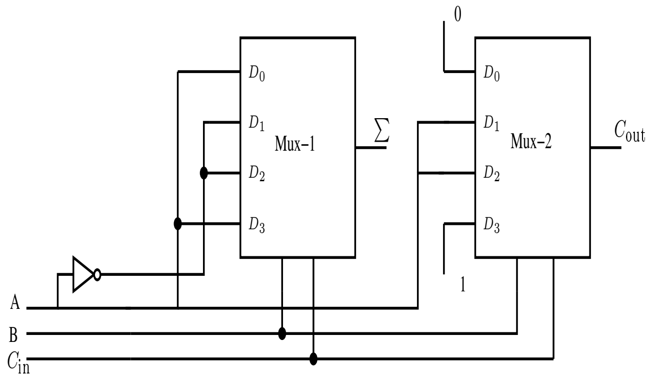


Figure 4.9: Circuit diagram and Functional Block diagram of a Multiplexer.







Demultiplexer

A demultiplexer basically reverses the multiplexing function. It takes digital information from one line and distributes it to a given number of output lines. for this reason a demultiplexer is also known as a data distributor. The circuit diagram of a 1-line-to-4 demultiplexer circuit is illustrated in fig 4.10.

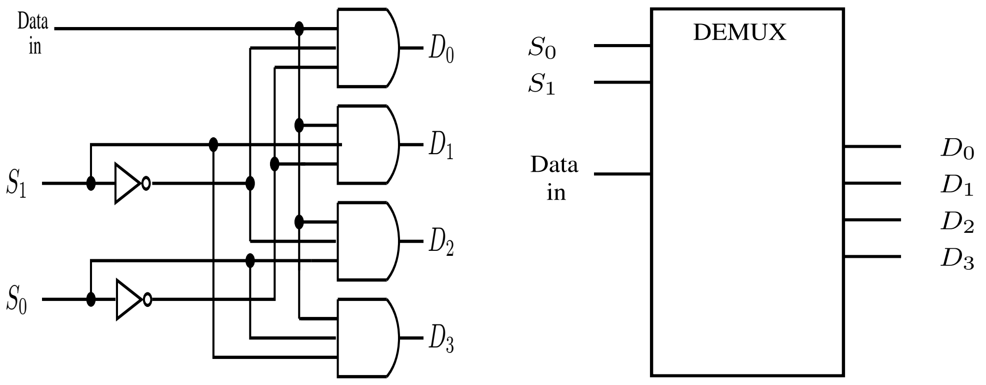


Figure 4.10: Circuit diagram and Functional Block diagram of a Multiplexer.

4.7 Parity Generator / Checkers

Errors can occur as digital codes are being transferred from one point to another within a digital system. These errors can create undesirable effects and must be prevented from happening, detected and corrected as required.

In parity method of error detection, a Parity Bit is attached to a group of information bits in order to make total number of 1s bits either even or odd.

> The sum (disregarding the carries) of an even number of 1s is always 0, and the sum of an odd number of 1s is always 1.

To determine the parity of a given code the bits are added modulo-2 using exclusive-or gates.

Please discuss implementation of 1's and 2's complement with logic gates.

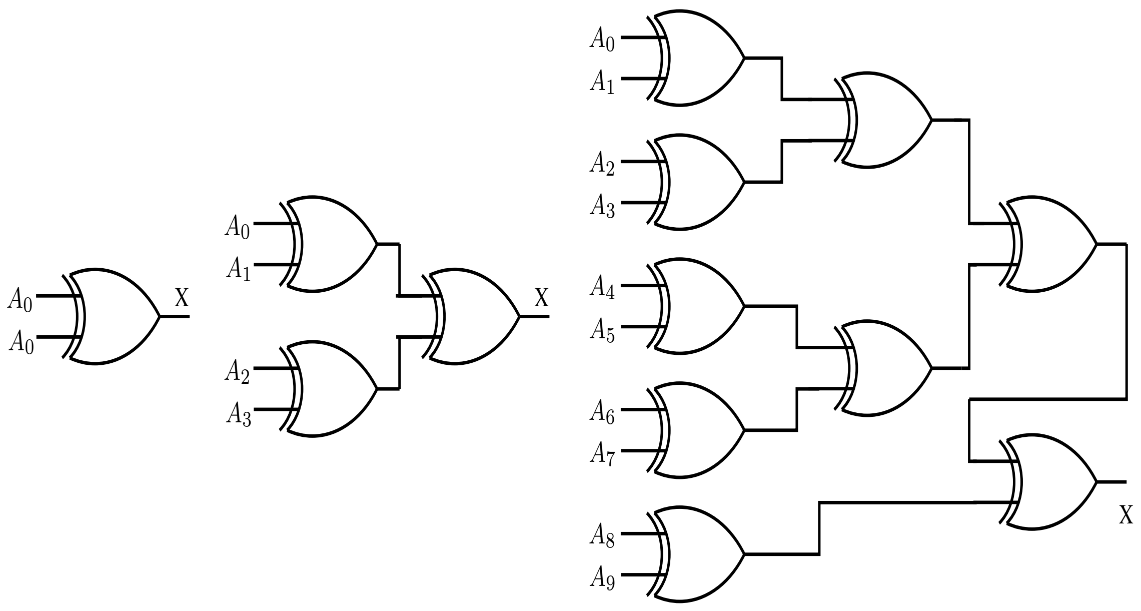


Figure 4.11: Circuit diagram and 2 / 4 and 9 bit words.

Chapter 5

Behavioural Model of Digital Circuits

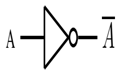
The objective of this chapter is to introduce to the behavioural modelling of digital circuits. This is very important because this powerful tool can help students develop complicated circuits and validate their performance without even implementing them on the hardware.

Ⓐ Modeling of Gates	Ⓓ Adder Circuit
Ⓑ Modeling of Sequential Circuit	Ⓔ Multiplexer / De Multiplexer
Ⓒ Structural Design	Ⓕ Traffic Circuit Implementation.

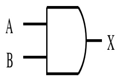
Introduction

5.1 Behavioural Modelling of Gates

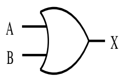
Hardware description language differs from software programming languages because HDL include ways of describing logic connections and characteristics. A HDL implements a logic design in hardware i.e. Programmable Logic Devices (PLD), where are software programming language, such as C or Basic instruct existing hardware what to do. Two standard HDLs are used for programming namely Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) and Verification Logic (Verilog). Table below illustrates basic implementation of Logic gates using VHDL. A VHDL implementation has entity/architecture structure. The **entity** describes the logic structure the input/output or ports, whereas the architecture describes the logic operations.



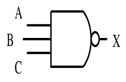
```
entity inverter is
port(A in bit; X out bit);
end entity interter;
architecture NOTfunction of inverter
     is
begin
   X<= not A;
end architecture NOTfunction
```



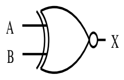
```
entity ANDgate is
port(A,B in bit; X out bit);
end entity ANDgate;
architecture ANDfunction of ANDgate is
begin
   X<= A and B
end architecture ANDfunction
```



```
entity ORgate is
port(A,B in bit; X out bit);
end entity ORgate;
architecture ORfunction of ORgate is
begin
   X<= A or B;
end architecture ORfunction
```



```
entity NANDgate is
port(A,B,C in bit; X out bit);
end entity NANDgate;
architecture NANDfunction of NANDgate
     is
begin
   X<= A nand B nand C;
end architecture NANDfunction
```



```
inverter symbol
entity XNORgate is
port(A,B in bit; X out bit);
end entity XNOR;
architecture XNORfunction of XNORgate
     is
begin
   X<= A xnor B;
end architecture XNORfunction
```

The ability to create simple and compact code is important in a VHDL program. Now we consider several examples illustrating the implementing Boolean expressions in VHDL.

Example 1:

$$X = (\overline{AC + \overline{B\overline{C}} + D}) + \overline{\overline{BC}}$$

The straight forward implementation of this expression in VHDL would look like

```
entity OriginalLogic is
port(A,B,C,D: in bit; X: out bit);
end entity OriginalLogic
architecture Expression1 of OriginalLogic is
begin
X<=not((A and C) or not(B and not C) or D) or not(not(B and C));
end architecture Expression1;
```

however, by selectively applying DeMorgan's theorem and axioms of Boolean Algebra, you can reduce the Boolean expression to the canonical form.

$$\begin{aligned}
(\overline{AC + \overline{B\overline{C}} + D}) + \overline{\overline{BC}} &= (\overline{AC})(\overline{\overline{B\overline{C}}})\overline{D} + \overline{\overline{BC}} \\
&= (\overline{AC})(B\overline{C})\overline{D} + BC \\
&= (\overline{A} + \overline{C})B\overline{CD} + BC \\
&= \overline{A}B\overline{CD} + B\overline{CD} + BC \\
&= B\overline{CD}(1 + \overline{A}) + BC \\
&= B\overline{CD} + BC
\end{aligned}$$

After application of axioms of Boolean algebra the implementation of this simplified expression in VHDL would look like

```
entity ReducedLogic is
port(A,B,C,D: in bit; X: out bit);
end entity ReducedLogic
architecture Expression2 of ReducedLogic is
```

```
begin
X<=(B and not C and not D) or (B and C);
end architecture Expression2;
```

Example 2: Implement the following SOP expression in VHDL without and with simplification

$$X = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}\bar{C}\bar{D}$$
$$+ A\bar{B}C\bar{D} + ABC\bar{D} + AB\bar{C}\bar{D} + A\bar{B}\bar{C}D + \bar{A}B\bar{C}D + AB\bar{C}D$$

The straight forward implementation of this expression in VHDL would look like

```
entity OriginalSOP is
port(A,B,C,D: in bit; X: out bit);
end entity OriginalSOP
architecture Equation1 of OriginalSOP is
begin
X<=(not A and not B and not C and not D) or
   (not A and not B and not C and     D) or
  (not A and     B and not C and not D) or
  (not A and     B and     C and not D) or
  (not A and not B and     C and not D) or
  (     A and not B and not C and not D) or
  (     A and not B and     C and not D) or
  (     A and     B and     C and not D) or
  (     A and     B and not C and not D) or
  (     A and not B and not C and     D) or
  (not A and     B and not C and     D) or
  (     A and     B and not C and     D)
end architecture Equation1;
```

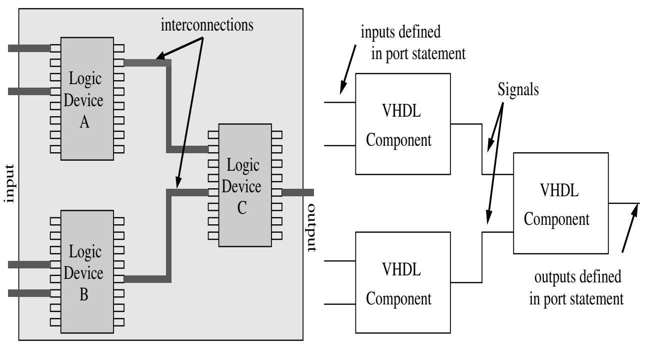


Figure 5.1: Simplified comparison between Hardware Implementation and VHDL Structural Implementation.

5.2 Combinational Logic with VHDL

The purpose of describing logic using VHDL is so that it can be programmed into PLD. This section provides insight into data flow approach using Boolean expressions and the structure approach are used to develop VHDL code for describing logic circuit.

The structural approach to writing a VHDL description of a logic function can be compared to installing IC devices of a circuit board and interconnecting them with wires. With the structural approach, you describe logic functions and specify how they are connected together. The VHDL component is away to predefine a logic function for repeated use in a program or other programs. The component can be used to describe anything from a simple logic gate to a complex logic function. The VHDL signal can be thought of as a way to specific wire connection between components. The simplified comparison of the structural approach to a hardware implementation on a circuit board.

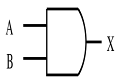
VHDL Components

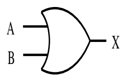
Note!
VHDL Component of AND gate can be defined once and used many times.

A VHDL component describes predefined logic that can be stored as a package declaration in a VHDL library and called as many times as necessary in a program. This is analoguous to having a storage bin of ICs and using them as needed.
VHDL program for any logic function can become a component and used whenever necessary in a larger program with use of component declaration of the following general form. **Component** is a VHDL keyword.

```
component name_of_component is
port(port_definitions);
end component name_of_component;
```

We have already discussed declaration of two input logic (and / or) gates with entity names ANDgate / ORgate





```
entity ANDgate is
port(A,B in bit; X out bit);
end entity ANDgate;
architecture ANDfunction of ANDgate is
begin
  X<= A and B
end architecture ANDfunction
```

```
entity ORgate is
port(A,B in bit; X out bit);
end entity ORgate;
architecture ORfunction of ORgate is
begin
  X<= A or B;
end architecture ORfunction
```

Now assume we are writing a program that has several AND gates so instead of writing the above declaration over and over again; use a component declaration to specify AND gate. The port statement in the component declaration must correspond to the port statement in the entity declaration of the AND gate.

```
component ANDgate is
port(A,B in bit; X out bit);
end component ANDgate;
```

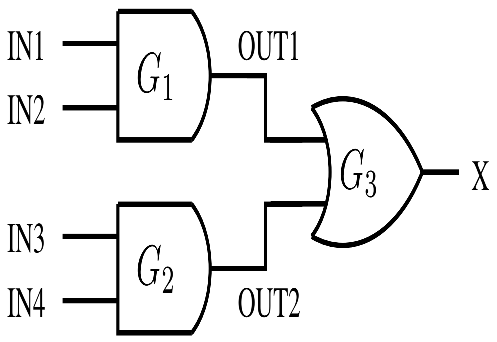


Figure 5.2: An example of implementation of VHDL structural design.

To use this component in the program, instantiate each component as a request or call for component to be used in the main program. The process of designing a simple SOP circuit is illustrated below:

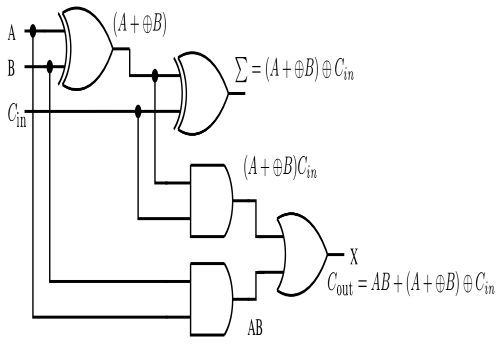
```
entity ANR_OR_LOGIC is
port(IN1,IN2,IN3,IN4:in bit OUT3: out bit);
end entity AND_OR_Logic;
```

The architecture declaration contains the components declarations for the AND gate and the ORgate, the signal definitions and the component instantiations.

```
architecture LogicOperation of AND_OR_Logic is

component AND_gate is
port (A,B: in bit; X:out bit);
end component AND_gate;

component OR_gate is
port (A,B: in bit; X:out bit);
end component OR_gate;

Signal OUT1, OUT2 bit;
begin
G1: AND_gate port map(A=> IN1,B=>IN2, X=>OUT1);
G2: AND_gate port map(A=> IN3,B=>IN4, X=>OUT2);
G3: OR_gate port map(A=> OUT1, B=>OUT3, X=>OUT3);
end architecture LogicOperation;
```

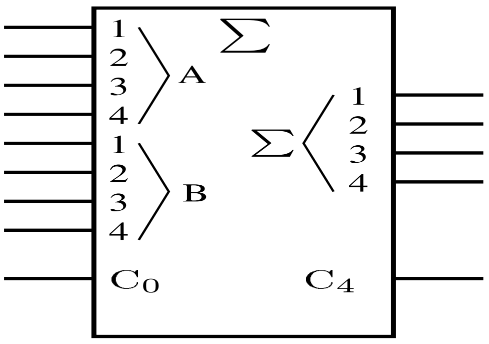
It can be noted that component instantiations appear between the keywords **begin** and **end architecture** statement for each instantiastion an identifier is defined such as G1, G2 and G3, the component name is specified. The keyword **port map** essentially makes all connections for the logic function using the operator =>. The next example

Full Adder


A
B
C_{in}
$(A+\oplus B)$
$\sum = (A+\oplus B)\oplus C_{in}$
$(A+\oplus B)C_{in}$
X
$C_{out} = AB+(A+\oplus B)\oplus C_{in}$
AB

```
entity FullAdder is
port (A, B, CIN: in bit; SUM, COUT: out bit);
end entity FullAdder;
architecture LogicOperation of FullAdder is
begin
SUM <= (A xor B) xor CIN;
COUT<= ((A xor B) and CIN) or (A and B);
end architecture LogicOperation;
```

4-bit Full Adder



This implementation is ripple carry adder.

```
entity 4BitFullAdder is
port (A1, A2, A3, A4, B1, B2, B3, B4, C0: in bit; S1, S2, S3, S4, C4: out bit);
end entity 4BitFullAdder;
architecture LogicOperation of 4BitFullAdder is
```

```
component FullAdder is
port (A, B, CIN: in bit; SUM, COUT: out bit);
end component FullAdder;
signal Cl, C2, C3: bit;
begin
FA1: FullAdder port map (A => A1, B => B1, CIN => C0, SUM => S1, COUT => Cl);
FA2: FullAdder port map (A => A2, B => B2, CIN => C1, SUM => S2, COUT => C2);
FA3: FullAdder port map (A => A3, B => B3, CIN => C2, SUM => S3, COUT => C3);
FA4: FullAdder port map (A => A4, B => B4, CIN => C3, SUM => S4, COUT => C4);
end architecture LogicOperation;
```

4-bit Comparator

```
entity 4BitComparator is
port (A0, A1, A2, A3, B0, B1, B2, B3: in bit; AequalB: out bit);
end entity 4BitComparator;
architecture LogicOperation of 4BitComparator is
begin
AequalB <= (A0 xnor B0) and (A1 xnor B1) and
(A2 xnor B2) and (A3 xnor B);
end architecture LogicOperation;
```

BCD to Decimal Encoder

```
entity BCDdecoder is
port (A0, A1, A2, A3: in bit; OUT0, OUT1, OUT2, OUT3,
OUT4, OUT5, OUT6, OUT7, OUT8, OUT9: out bit);
end entity BCDdecoder;
architecture LogicOperation of BCDdecoder is
begin
OUT0 <= not(not A0 and not A1 and not A2 and not A3);
OUT1 <= not(A0 and not A1 and not A2 and not A3);
OUT2 <= not(not A0 and A1 and not A2 and not A3);
OUT3 <= not(A0 and A1 and not A2 and not A3);
OUT4 <= not(not A0 and not A1 and A2 and not A3);
OUT5 <= not(A0 and not A1 and A2 and not A3);
OUT6 <= not(not A0 and A1 and A2 and not A3);
OUT7 <= not(A0 and A1 and A2 and not A3);
OUT8 <= not(not A0 and not A1 and not A2 and A3);
OUT9 <= not(A0 and not A1 and not A2 and A3);
end architecture LogicOperation;
```

Decimal to BCD Encoder

```
entity DecBCDencoder is
port (D1, D2, D3, D4, D5, D6, D7, D8, D9:
in bit; A0, A1, A2, A3: out bit);
end entity DecBCDencoder;
architecture LogicFunction of DecBCDencoder is
begin
A0 6= (D1 or D3 or D5 or D7 or D9);
A1 6= (D2 or D3 or D6 or D7);
A2 6= (D4 or D5 or D6 or D7);
A3 6= (D8 or D9);
end architecture LogicFunction;
```

8 to 1 Encoder

```
entity EightInputMUX is
port (S0, S1, S2, D0, D1, D2, D3, D4, D5, D6, D7,
EN: in bit; Y: inout bit; YI: out bit);
end entity EightInputMUX;
architecture LogicOperation of EightInputMUX is
signal AND0, AND1, AND2, AND3, AND4, AND5, AND6, AND7: bit;
begin
AND0 <= not S0 and not S1 and not S2 and D0 and not EN;
AND1 <= S0 and not S1 and not S2 and D1 and not EN;
AND2 <= not S0 and S1 and not S2 and D2 and not EN;
AND3 <= S0 and S1 and not S2 and D3 and not EN;
AND4 <= not S0 and not S1 and S2 and D4 and not EN;
AND5 <= S0 and not S1 and S2 and D5 and not EN;
AND6 <= not S0 and S1 and S2 and D6 and not EN;
AND7 <= S0 and S1 and S2 and D7 and not EN;
Y <= AND0 or AND1 or AND2 or AND3 or AND4 or AND5 or AND6 or AND7;
YI <= not Y;
end architecture LogicOperation;
```

VHDL implemention of two half adders make a full adder

VHDL implemention of multiplexer / demultiplexers econders and decoders

Chapter 6

Register / Counters & Memory

Outline

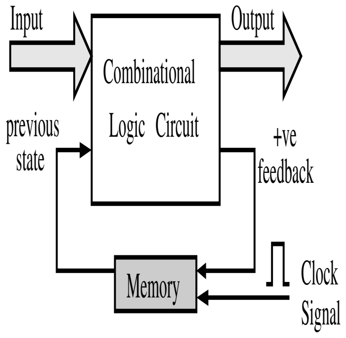
The chapter concerns itself with sequential logic circuits namely Bistable, monostable and Astable logic devices.

Ⓐ SR Latches	Ⓓ Flip-Flop Operation
Ⓑ Gate Enabled SR Latch	Ⓔ Flip-Flop Applications
Ⓒ D Latch	Ⓕ

Note!
Basic Difference between FlipFlops and Latches is the way in which their state changes from one stage to another

Unlike the Combinatorial logic where as values for boolean expressions is applied to the circuit the output is evaluated almost immediately; sequential circuits produce output as different signals travel through the circuit as a function of some reference signal ‘clock‘.
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 on 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 the two basic states, a logic level ‘1‘ or a logic level ‘0‘ and will remain latched 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.

The word sequential means that things happen in sequence, one after another and in sequential logic circuit, the actual clock signal determines when things will happen next. Simple sequential logic circuits can be constructed from standard Bistable circuits such as flip-flops, latches and counters and which themselves can be made by simply connecting together universal NAND and/or NOR gates in a particular combinational way to produce the required sequential circuit.

Unlike combinatorial logic circuit that change state depending upon the actual signals being applied to their inputs at that time. Sequential logic circuits have some form of inherent ‘Memory‘ built-in.

Sequential logic circuits use flip-flops and Latches as memory elements and in which their output is dependent on the input state.

Types of Sequential Circuits

Bi-Stable device has two stable states called SET and RESET.

A Monostable multivibrator (a.k.a One Shot) has only one stable state, A one-shot produces a single controlled-width pulse when activated or triggered.

Astable multivibrator, has no stable state and is used primarily as an oscillator, which is a self-sustained waveform generator.

6.1 Latches

The latch is type of temporary storage device that has two stable (bistable) and is normally placed in a category separate from that of flip-flops. Latches are similar to flip-flops because they are bistable devices that can reside in either of two states using a feedback arrangement, in which the outputs are connected back to the opposite inputs.

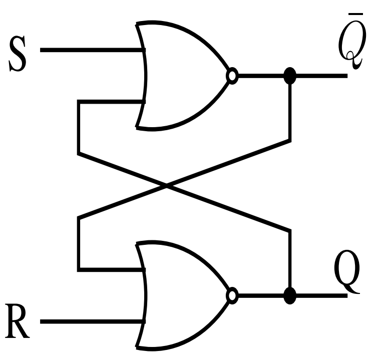
6.1.1 Set-Reset Latches

Note!
Latches and flip-flops differ in the method used for changing their state.

A latch is a type of bistable logic device or multivibrator. An active-High input S-R latch is formed with two cross-coupled NOR gates as in Fig. 6.1, an active-LOW input SR latch is formed with two cross-coupled NAND gates, as in Fig. 6.2. This produces a regenerative feedback that is characteristic of all latches and flip-flops. The characteristic equation for SR latch with NOR gate implementation is

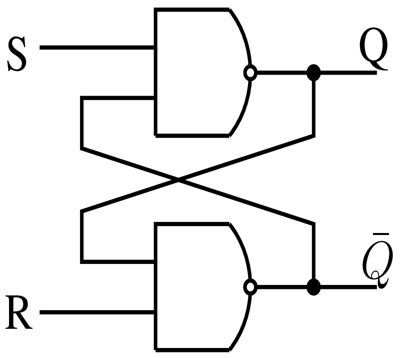
$$Q^+ = S + \bar{R}Q$$

Note that SR-Latch with NOR gate implementation is active HIGH input (at any given time one of the two inputs must be HIGH), while NAND gate implementation is active LOW input (at any given time one of the two inputs must be LOW).



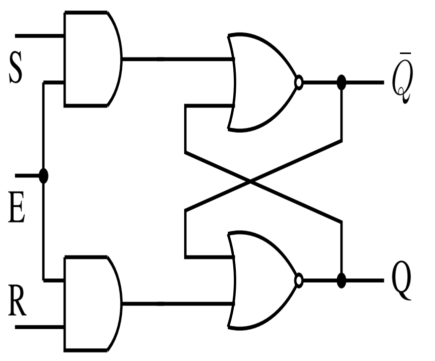
S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	?

Figure 6.1: Circuit diagram and Truth Table of SR latch with NOR Gate implementation.

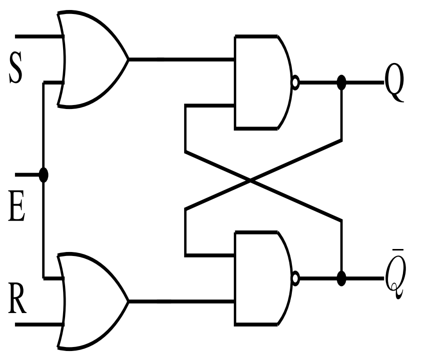


S	R	Q^+
0	0	?
0	1	1
1	0	0
1	1	Q

Figure 6.2: Circuit diagram and Truth Table of SR latch with NAND Gate implementation.



E	S	R	Q^+
0	X	X	Q
0	0	0	Q
0	0	1	0
1	1	0	1
1	1	1	?



E	S	R	Q^+
0	0	0	?
0	0	1	1
0	1	0	0
0	1	1	Q
1	X	X	Q

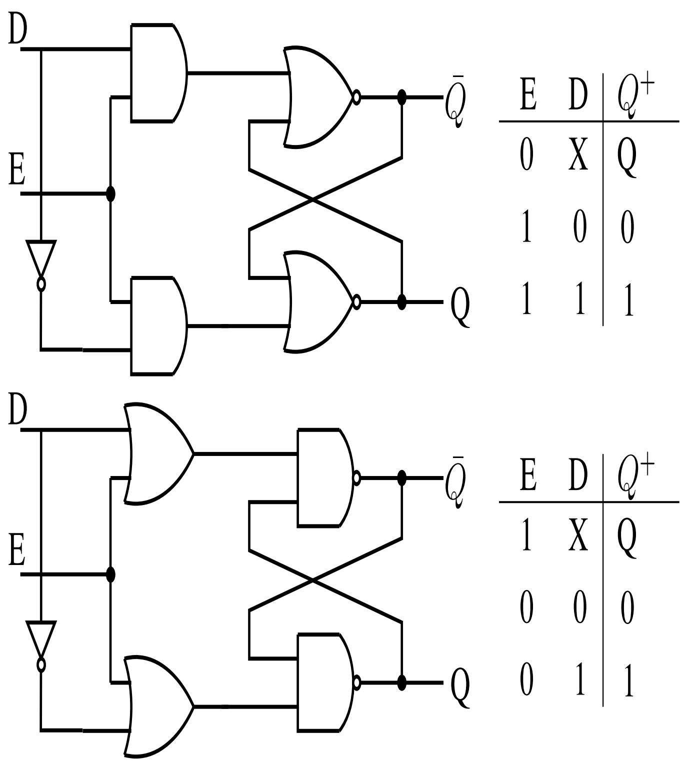
Figure 6.3: Circuit diagram and Truth Table of Gated SR latch with NOR and NAND Gate implementation.

6.1.2 Gated Set-Reset Latches

The gated SR-Latch requires an enable input (EN), the logic diagram and truth-table (for both NOR and NAND gate) are illustrated in figure 6.3. The S and R inputs control the state to which the latch will go as the EN signal is applied. The latch will not change until EN is HIGH, but as long as it remains HIGH the output will be controlled by the state of S and R input. The gated-latch is a level-sensitive device.

6.1.3 DATA Latches

This is another type of gated latch; it differs from the S-R latch because it has only one input in addition to EN. This input is called Data input; the logic diagram and the symbol are illustrated in figure 6.4. When D input is high, the latch will set;



E	D	Q^+
0	X	Q
1	0	0
1	1	1

E	D	Q^+
1	X	Q
0	0	0
0	1	1

Figure 6.4: Circuit diagram and Truth Table of D latch with NOR and NAND Gate implementation.

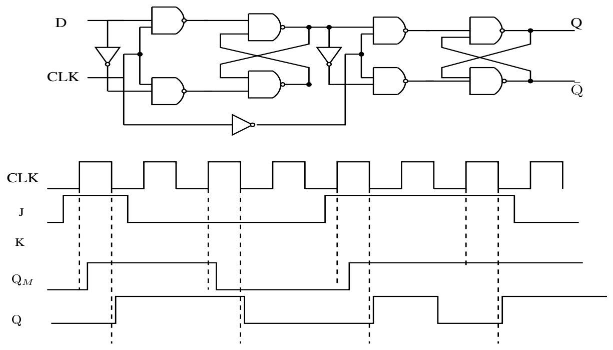
The output Q will follow the D input when EN is HIGH.

6.2 Flip-Flops

Flip Flops are synchronous bistable devices; also known as bistable multivibrators. In this case, the term synchronous means that the output changes only at a specified point (leading trailing edge) of the triggering input called clock (CLK). In synchronization with the clock the Flip-Flops are edge-triggered or edge-sensitive whereas gates latches are level sensitive.
An edge-triggered flip-flop changes state either at the positive edge (rising edge) or at negative edge (falling edge) of the clock pulse and is sensitive to inputs only at this transition of the clock.

6.2.1 D Flip-Flop

The D input of the D flip-flop is a synchronous input because data on the input are transferred to the flip-flop‘s output only at the triggering edge of the clock pulse. When D is HIGH, the Q output goes HIGH on the triggering edge of the clock pulse. and teh Flip-Flop is SET. When D is low, the Q output goes LOW on the triggering edge of the clock pulse and the flip-flop is RESET.



6.2.2 The JK Flip-Flop

The J and K inputs of the J-K flip-flop are synchronous devices. They are refinement of Gated RS Latches in that the indeterminate state of RS latches is defined in JK type. The inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. The input marked J is for set and K is for reset. When both J and K are HIGH, the flip-flop switches to its complement state. The circuit diagram, truth-table and symbol are illustrated in figure **??**.

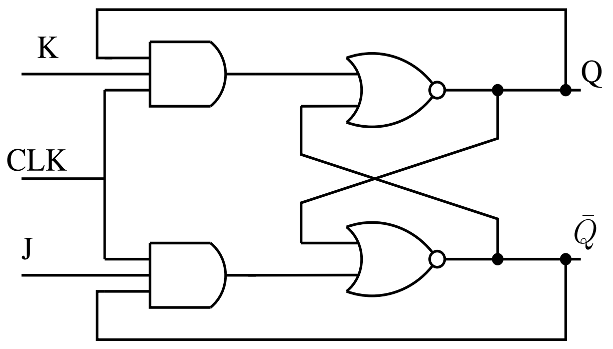


Figure 6.5: Basic logic circuit of JK Flip-Flop .

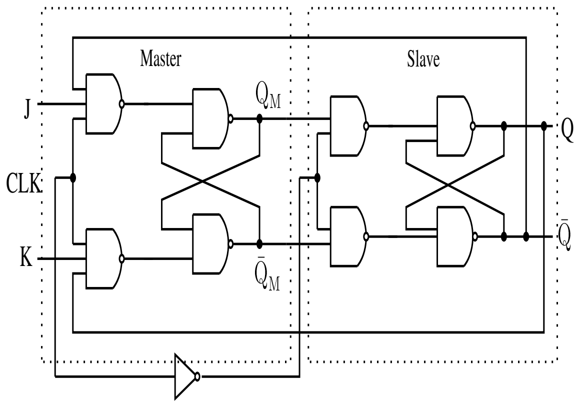
J	K	Q	Q^+	$\bar{Q}^+$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Table 6.1: Truth Table for JK Flip-Flop.

Feedback timing issue: it is very important to realize that because of the feedback connection in the JK flip-flop, a CLK pulse that remains in the 1 state while both J and K are equal to 1 will cause the output to complement again and again until the CLK pulse goes back to 0. To avoid this undesirable operation, the clock pulse must have a time duration that is shorter than the propagation delay time of the flip-flop. This is a restrictive requirement, since the operation of the circuit depends on the width of the pulse. for this reason JK flip-flops are never built in the way illustrated in figure 6.5. The restriction on the pulse width can be eliminated with a $\underline{\text{m}}$aster-slave or $\underline{\text{e}}$dge-triggered constructions.

6.2.3 The T Flip-Flop

The T flip-flop is a single-input version of the JK flip-flop. The T flip-flop is obtained from the JK flip-flop when both inputs are tied together. The designation T comes from the ability of the flip-flop to $\underline{\text{t}}$oggle its state regardless of the present state, the flip-flop complements its output when clock pulse occurs when in put T is 1. The characteristic table and circuit diagram are illustrated in the figure below.



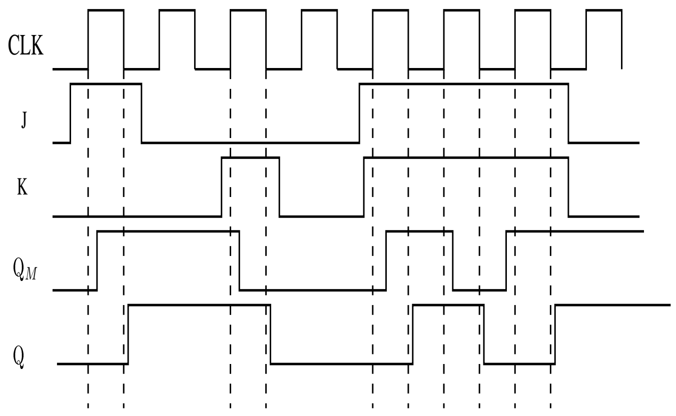


Figure 6.6: Implementation of Edge-triggered Master/Slave Logical Circuit of JK Flip-Flop.

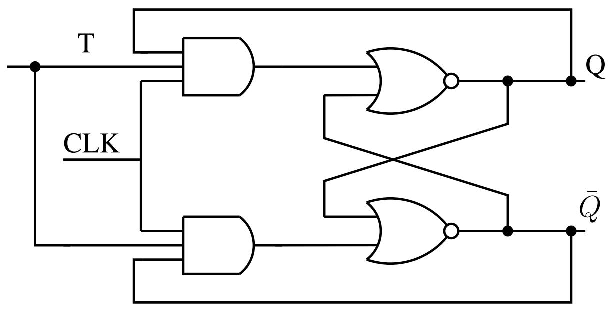


Figure 6.7: Basic logic circuit of Toggle Flip-Flop .

T	Q	Q^+	$\bar{Q}^+$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

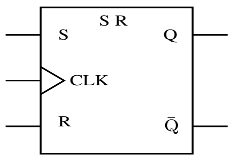
Table 6.2: Characteristic Table for T Flip-Flop.

6.2.4 Master-Slave Flip-FLop

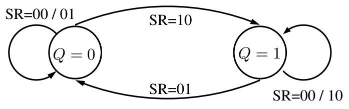
A master-slave flipflop is constructed from the two separate flip-flops. One circuit serves as a master and other as slave, and the overall circuit is referred to as a master-slave flip-flop. The logic diagram of an RS master-slave flip-flop is illustrated in figure. When clock pulse is 0, the output of the inverter is 1. Since the clock input of the slave is 1, the flip-flop is enabled and output **Q** is equal to $\bar{Q}$. The master flip-flop is disabled because clock pulse is 0. When the pulse becomes 1, the information then at external R and S inputs is transmitted to master flip-flop. The slave flip-flop however is isolated as long as the pulse is at its HIGH level, because the output of the inverter is 0. When the pulse returns to 0, the master flip-flop is isolated, which presents the external inputs from affecting it. The slave flip-flop then goes to the same state as the master flip-flop.

The timing relationships in figure illustrate the sequence in which the events occur in master slave flip-flop.

Name Symbol	Characteristic Truth (Table)	State Diagram Characteristic equation	Excitation Table



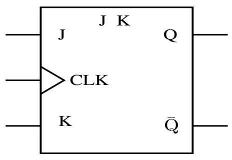
S	R	Q	Q$^+$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x



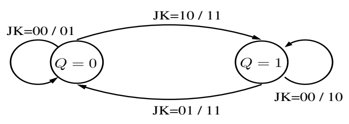
$Q^+ = S + \bar{R}Q$

\$SR=0\$

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



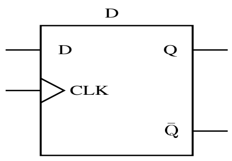
J	K	Q	Q$^+$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



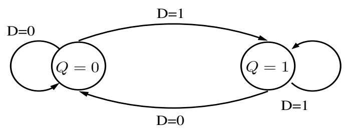
$Q^+ = \bar{J}\bar{K}Q + J\bar{K} + JK\bar{Q}$

$Q^+ = \bar{K}Q + J\bar{Q}$

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

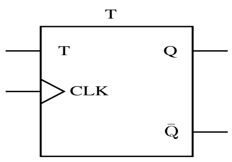


D	Q	Q$^+$
0	x	0
1	x	1

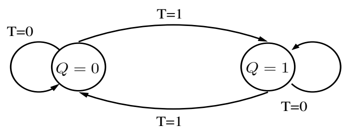


$Q^+ = D$

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



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



$Q^+ = T\bar{Q} + \bar{T}Q$

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

Difference in design and behaviour of latches and flip-flops are tabulated below in table 6.3:

6.3 Flip-Flop Applications

Flip-flops find applications in registers, frequency dividers , counters, these will be discussed in detail in the subsequent chapters, however brief introduction is provided here

Latches	Flip-Flops
▼ Latch do not require clock signal.	▼ Flip-Flops require clock signal.
▼ Latch is an asynchronous device.	▼ Flip-flop is a synchronous device.
▼ Latches are transparent i.e.	▼ A transition of clock from low to high
when they are enabled they change	and vice versa cause flip-flop to change
immediately when input changes.	or retain the state depending on input signal.
▼ Latch is a level sensitive device	▼ Flip-flop is an edge sensitive device
▼ Latches are simpler to design i.e.	▼ Compared to Latches Flip-Flops are more
no routing of clock signal is required.	complex to design
▼ The power requirement of a latch is less.	▼ Power requirement of a flip-flop is higher.
▼ A latch works on the enable signal.	▼ A flip-flop works with the clock signal.

Table 6.3: Comparison between design and operation of Latches and flip-flops.

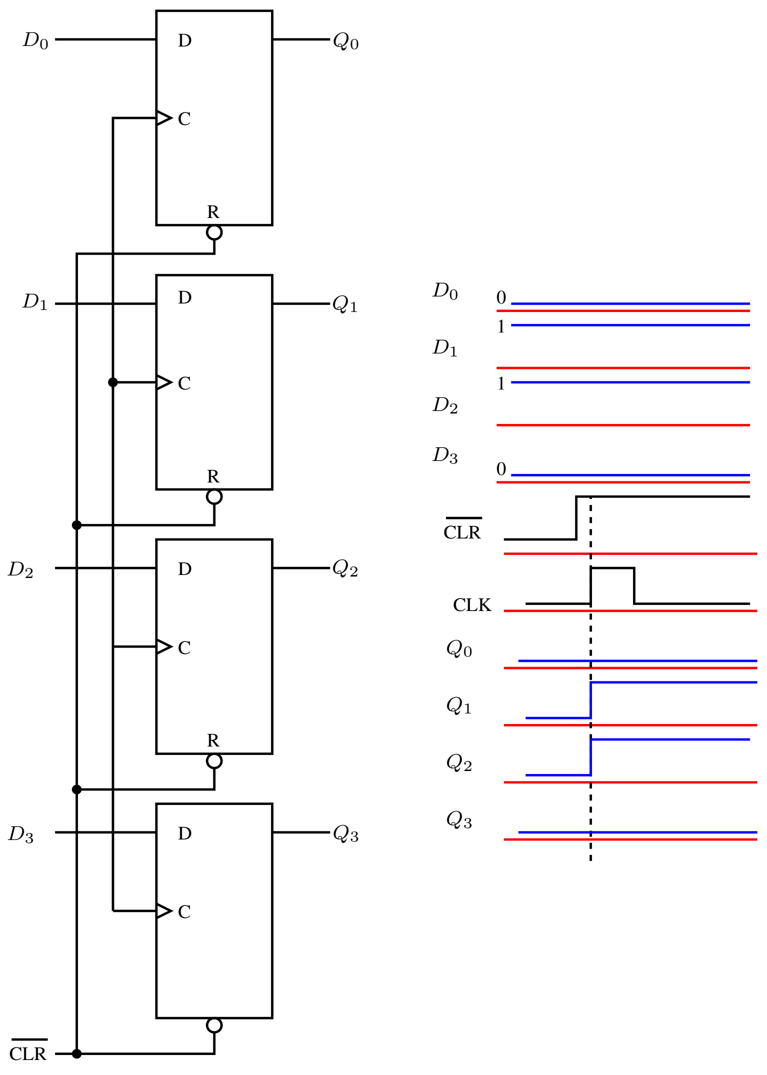


Figure 6.8: Application of Flip-Flops are Parallel Registers.

6.3.1 Parallel Data Storage

This is a very important requirement in a digital systems to store several bits of data from parallel lines simultaneously in a group of flip-flops. Each of the four parallel data lines is connected to the D input of a flip-flop. The clock inputs of the flip-flop are connected together, so that each flip-flop is triggered by the same clock pulse.

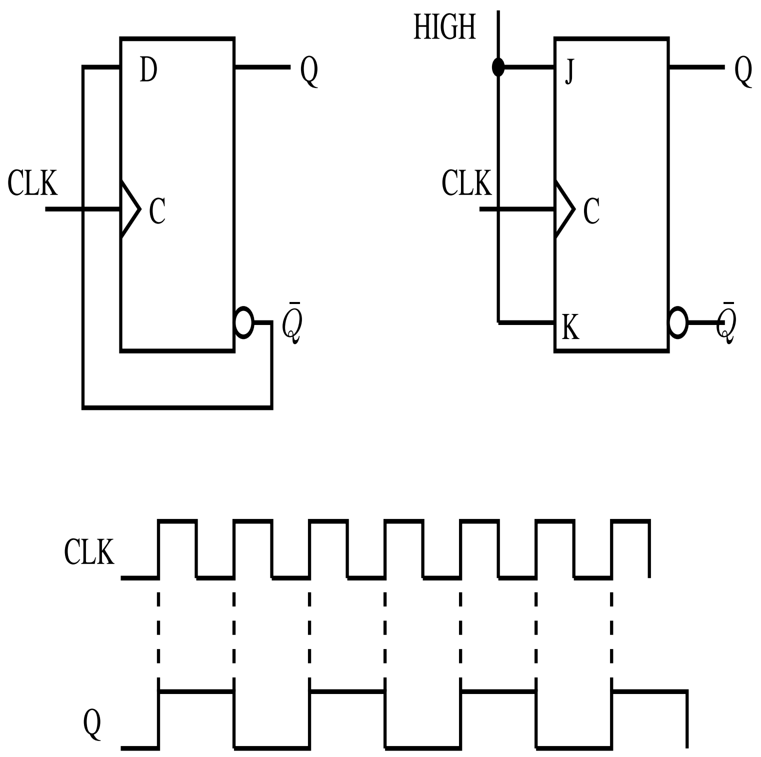


Figure 6.9: Application of Flip-Flops to create a frequency divider circuit.

6.3.2 Frequency Division

Another important application of a flip-flop is dividing (reducing) the frequency of periodic waveform. When a pulse waveform is applied tot he clock input of a D or JK flip-flop that is connected to toggle ($D = \bar{Q}$ or $J = K = 1$), the Q output is a square wave with one-half the frequency of the clock input. Thus, a single flip-flop can be applied as a divided-by-2 device as illustrated in figure 6.9 for both a D and J-K flip-flop.

Further division of a clock frequency can be achieved by using the output of one flip-flop as the clock input to a second flip-flop, as illustrated in figure 6.9. The frequency of the Q_A output is divided by 2 by flip-flop B. The Q_B output is, therefore, one-fourth the frequency of the original clock input. Propagation delay times are not shown on the timing diagrams.

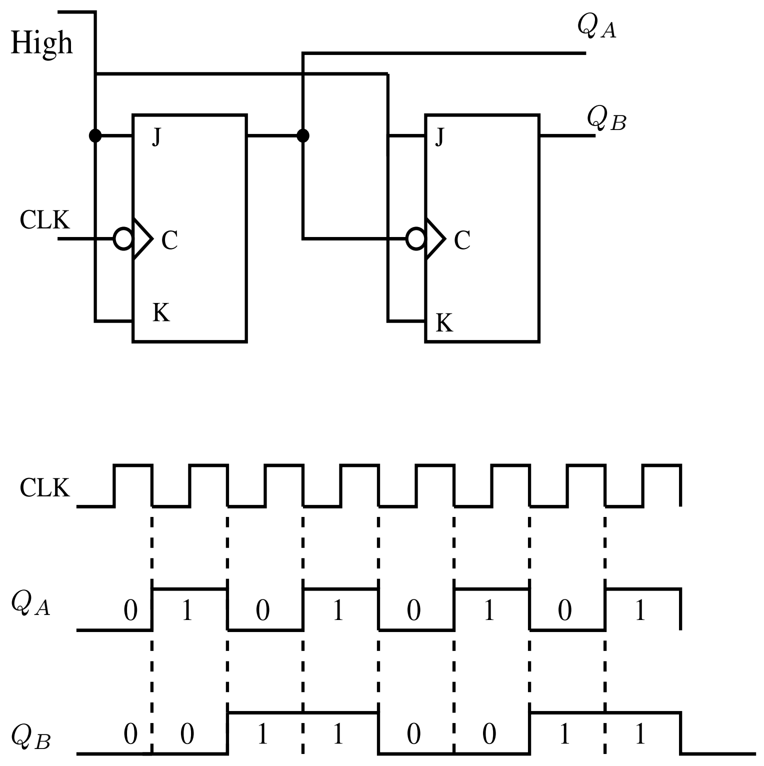


Figure 6.10: Application of Flip-Flops to create a binary counter circuit.

6.3.3 Counting

Another important application of flip-flops is in digital counters. The concept is illustrated in figure ??. Negative edge-triggered J-K flip-flops are used for illustration. both flip-flops are initially RESET. The flip-flop A toggles on each negative going transition of each clock pulse. The Q output of Flip-Flop Ac locks flip-flop B, so each time Q_A makes a HIGH-to-LOW transition, flip-flop B toggles. The resulting waveforms are illustrated in figure 6.10.

Chapter 7

Registers and Counters

Outline

The objective of this chapter is to review the design of digital circuits such are shift registers, counters and their applications; with special emphasis on the timing diagrams

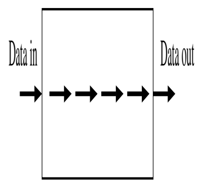
Ⓐ Shift Register Operation	Ⓔ Synchronous / Asynchronous Counters
Ⓑ Bidirectional Shift Register	Ⓕ Up/Down Synchronous Counters
Ⓒ Shift Register Applications	Ⓖ Counter Applications
Ⓓ Finite State Machines	Ⓗ

7.1 Shift Register Operations

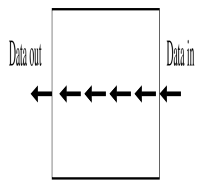
Shift registers consist of arrangements of flip-flops and are important in applications involving the storage and transfer of data in digital systems. A register has no specified sequence of states, except in certain very specialized applications. A register, in general is used solely for storing and shifting data (1s and 0s) entered into it from an external source and typically possesses no characteristic internal sequence of states.

A register is digital circuit with two basic function; data storage and data movement. This storage capability of a register makes it an important type of memory device.

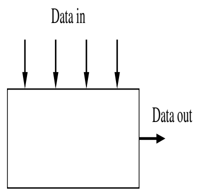
The storage capacity of a register is the total number of bits (1s and 0s) of a digital data it can retain. Each stage (flip-flop) in a shift register represents one bit of storage capacity; therefore, the number of stages in a register determines its storage capacity.



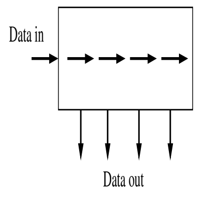
Left to Right Shift



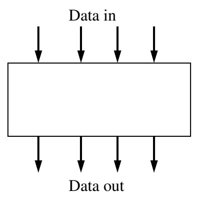
Right to Left Shift



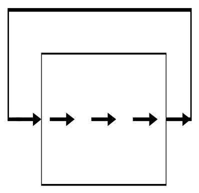
Parallel in Serial out



Serial in Parallel out



Parallel in Parallel out


Ring Shift Register

The shift capability of a register permits the movement of data from stage to stage within the register or into or out of the register upon application of clock pulses the types of data movement in shift registers. The block represents any arbitrary 4-bit register, and the arrows indicate the direction of data movement.

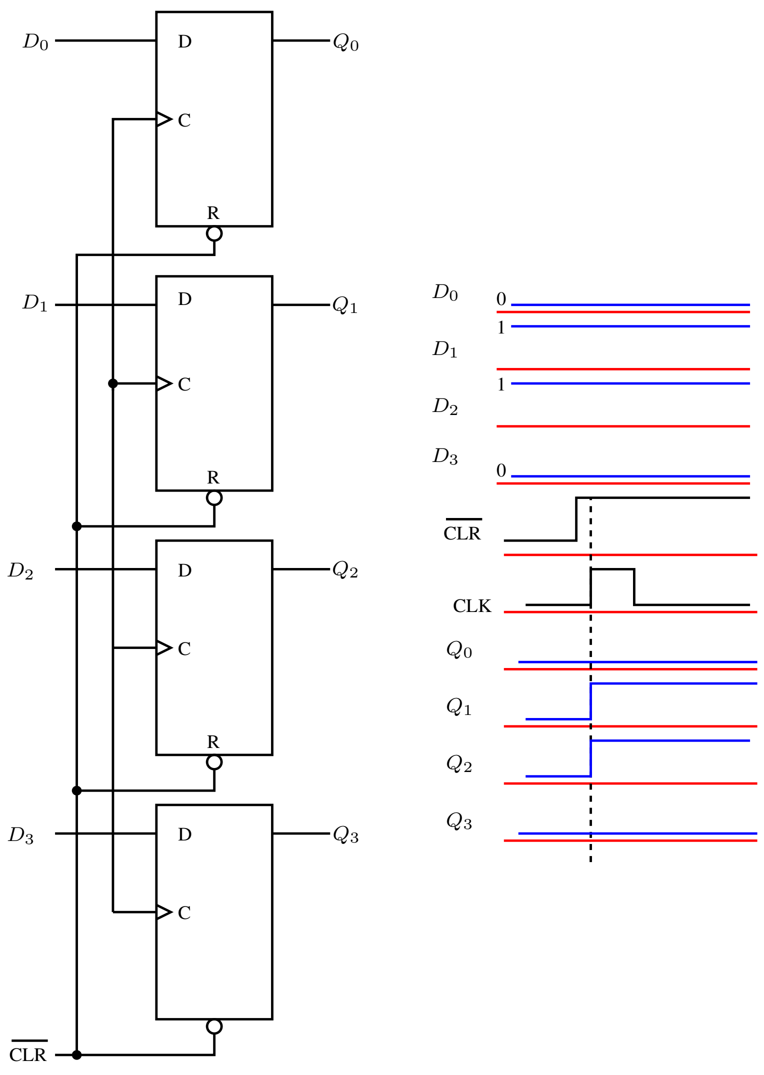


Figure 7.1: Basic data movement in shift registers.

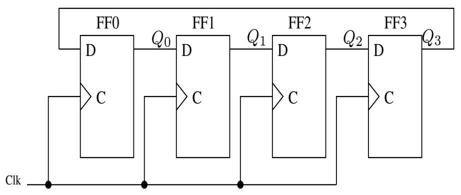


Figure 7.2: Circuit Diagram of Serial in Serial Out Shift Register.

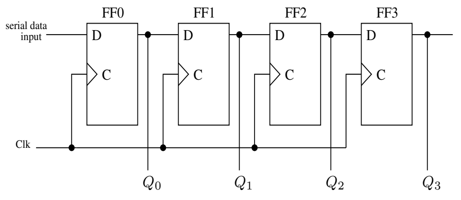


Figure 7.3: Circuit Diagram of Serial in Parallel Out Shift Register.

7.2 Types of Shift Register Data I/Os

There are four types of shift registers based on data input and output are considered; serial in/serial out, serial in / parallel out, parallel in / serial out and parallel in / parallel out.

7.2.1 Serial In / Serial Out Shift Registers

The serial in / serial out shift register accepts data serially - that is, one bit at a time on a single line. It produces the stored information on its output also in serial form.

7.2.2 Serial In / Parallel Out Shift Registers

Data bits are entered serially (least-significant bit first) into a serial in / parallel out shift register in the same manner as in serial in / serial out registers. The difference is the way in which the data bits are taken out of the register.

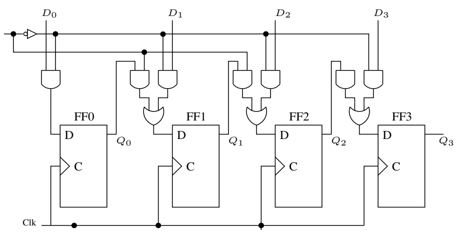


Figure 7.4: Circuit Diagram of Parallel in Serial Out Shift Register.

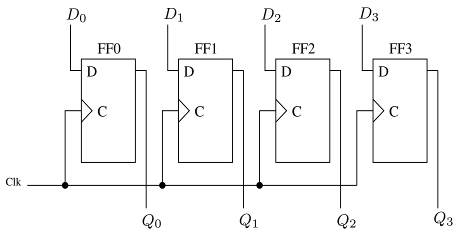


Figure 7.5: Circuit Diagram of Parallel in Parallel Out Shift Register.

7.2.3 Parallel In / Serial Out Shift Registers

For a register with parallel data inputs, the bits are entered simultaneously into their respective stages on parallel lines rather than on a bit-by-bit basis on one line as with serial data inputs. The serial output is the same as in serial in/ serial out shift registers, once the data are completely stored in the register. The $Shift/\overline{Load}$ input allows for loading and/or shifting the bits.

7.2.4 Parallel In / Parallel Out Shift Registers

Immediately following the simultaneous entry of data and the bits will appear on the parallel outputs.

7.3 Bi-Directional Shift Registers

A bi-directional shift register is one in which the data can be shifted either left or right. It can be implemented using gating logic that enables transfer of a data bit

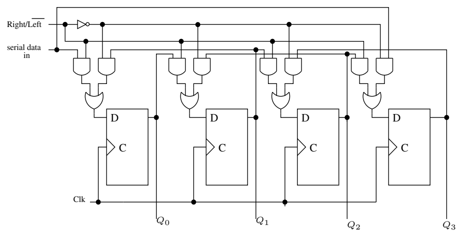


Figure 7.6: Circuit Diagram of Bi-Directional Shift Registers.

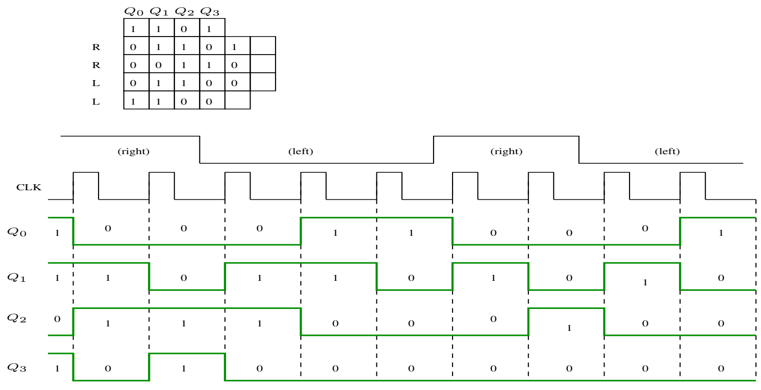


Figure 7.7: Timing Diagrams of Bi-Directional Shift Registers.

from one stage to next stage to the right or to the left, depending on the level of control line.

7.4 Shift Registers Counters

A shift register counter is basically a shift register with a serial output connected back to the serial input to produce a special sequences. These devices are classified as counters because they exhibit a specified sequence of states. Two of the most common shift registers counters are Johnson counter and ring counter, described in this section.

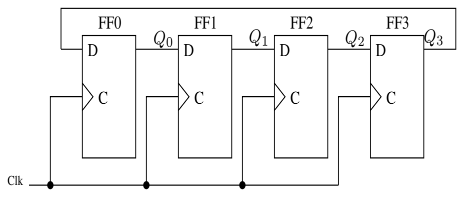


Figure 7.8: Circuit diagram of 4-bit Jhonson Counter.

Table 7.1: 4-bit Jhonson Sequence.

clk	Q_0	Q_1	Q_2	Q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

Johnson Counter

In a Johnson counter the complement of the output of the last flip-flop is connected back to the D input of the first flip-flop (obviously it can be implemented with other types of flip-flops as well). If the counter starts at 0, this feedback arrangement produces a characteristic sequence of states as illustrated in table. Please note that for n stages there are $2n$ number of counter stages.

Ring Counter

A ring counter utilizes one flip-flop for each state in its sequence. It has the advantage that decoding gates are not required. In this case of 10-bit ring counter, there

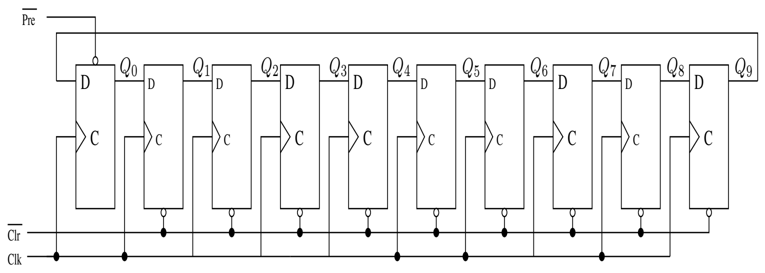


Figure 7.9: Circuit Diagram of Jhonson Shift Register Counter.

Table 7.2: Output sequence of Ring Counter.

clk	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	0

is a unique output for each decimal digit. The circuit diagram is illustrated in the Fig. ??. The counter output sequence is illustrated in the table.

7.5 Shift Register Applications

Shift registers are found in many types of applications, a few of which are presented here:

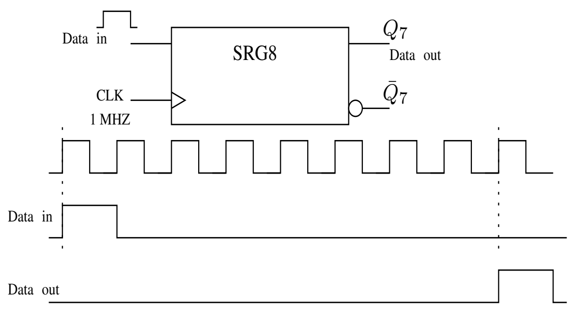


Figure 7.10: Shift Register as a delay Register.

Time Delay

A serial in/Serial out shift register can be used to provided a time delay from input to output that is a function of both the number of stages n in the register and the clock frequency. This time delay operation is illustrated in Fig. 7.10.

Serial-to-parallel Data Converter

Serial data transmission from one digital system to another is commonly used to reduce the number of wires in the transmission line. For example, eight bits can be sent serially over one wire, but it takes eight wires to send the same data in parallel. Serial data transmission is widely used by peripherals to pass data back and forth to a computer. For example, USB (universal serial bus) is used to connect keyboards, printers, scanners and more to the computer. All computer process data in parallel form, thus there is a requirement for serial-to-parallel conversion.
To illustrate the operation of this serial-to-parallel converter, the serial data format shown in Fig. 7.11. It consists of 11 bits. The first bit (start bit) is always 0 and always begins from HIGH-to-LOW transition. The next eight bits (D_7 through D_0) are the data bits one of the bits can be for parity, and the last one or two bits (stop bits) are always 1s. When no data are being sent there is a continuous HIGH on the serial data line.
The HIGH-to-LOW transition of the start bit sets the control flip-flop, which enables

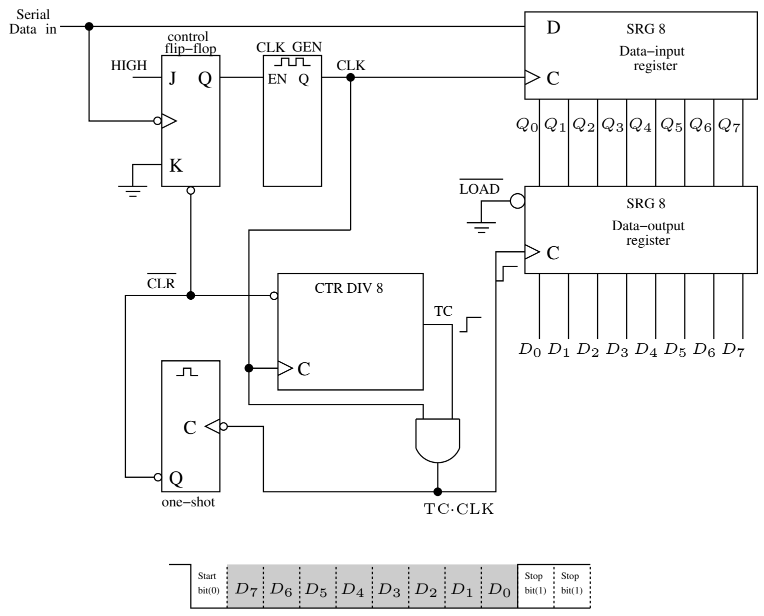


Figure 7.11: Circuit Diagram for Conversion of Serial Data to Parallel.

the clock generator. After a fixed delay time, the clock generator begins producing a pulse waveform, which is applied to the data-input register and to the divide-by-8 counter. The clock has a frequency precisely equal to that of the incoming serial data, and the first clock pulse after the start bit occurs during the first data bit. The timing diagram is illustrated in the figure. At the eighth clock pulse, the terminal count (TC) goes from LOW to HIGH indicating that counter is at the last state. The rising edge is ANDed with the clock pulse which is still HIGH, producing a rising edge at TC· CLK. This parallel loads the eight data bits from the data input shift register to the data-output register. A short time later the clock pulse goes LOW and HIGH-to-LOW transition triggers the one-shot, which produces a short pulse to clear the counter and rest control flip-flop and thus disable clock generator. The system is ready for next 11 bits.

7.6 Finite State Machines

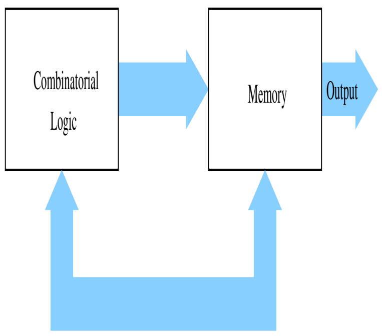
A state machine is a sequential circuit having limited (finite) number of states occuring in a prescribed order. A counter is an example of state machine; the number of states is called *modulus*. Two basic types of state machines are the **Moore** and **Mealy**.

Note!
Note:In Moore machine Output is dependant on State.

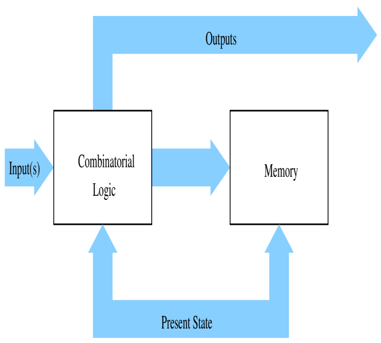
A **Moore State Machine** is one where outputs depend only on the internal present state.

Note!
Note:In Mealy machine Output is dependent on Transition.

A **Mealy State Machine** is one where outputs depend only on both the internal present state and on the input.



(a) Moore Finite State Machine

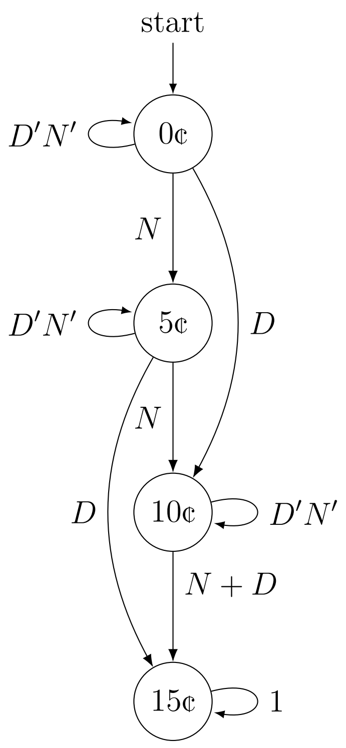


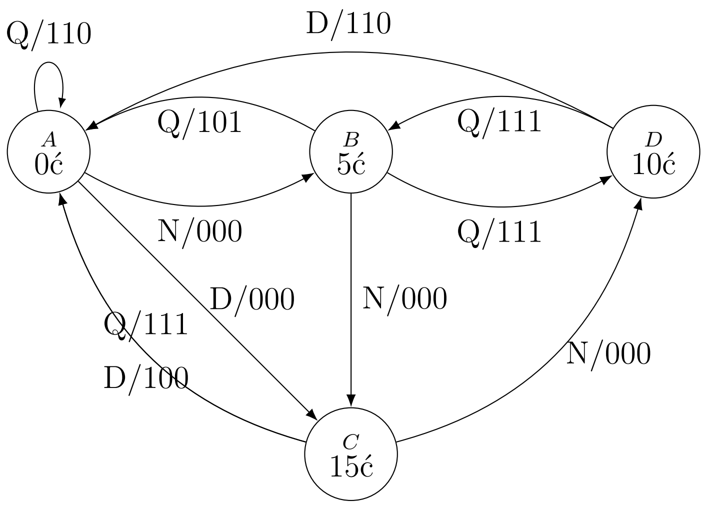
(b) Mealy Finite State Machine

In Moore's machine combinatorial logic gate array with outputs that determine the next state of flip-flops in the memory. There may or may not be inputs to the combinatorial logic.

7.7 Asynchronous Counters

The term asynchronous refers to events that do not have a fixed time relationship with each other and, generally, do not occur at the same time. An asynchronous counter is one in which the flip-flops within the counter do not change states at exactly the same time because they do not have a common clock phase.





A 2-Bit Asynchronous Binary Counter

figure shows a 2 bit counter connected for asynchronous operation. Notice that the clock is applied to the clock input (C) of only the first flip-flop FF0, which is always the least significant bit (LSB). The second flip-flop, FF1 is triggered by the $\bar{Q}_0$ output of FF0. FF0 changes state at the positive-going edge of each clock pulse, but FF1 changes only when triggered by a positive-going transition of the $\bar{Q}_0$ output of FF0. Because of the inherent propagation delay time through a flip-flop, a transition of the input clock pulse (CLK) and a transition of the $\bar{Q}_0$ output of FF0 can never occur at exactly same time. Therefore, the two flip-flops are never simultaneously triggered, so the counter operation is asynchronous.

7.8 Synchronous Counters

The synchronous counters refer to events that have a fixed time relationship with each othe. A **synchronous counter** is one in which flip-flops in the counter are

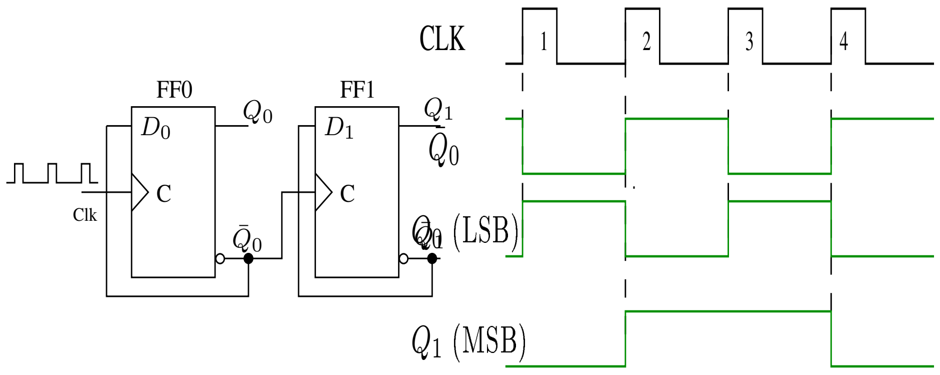


Figure 7.12: Circuit diagram and Functional Block diagram of a Multiplexer.

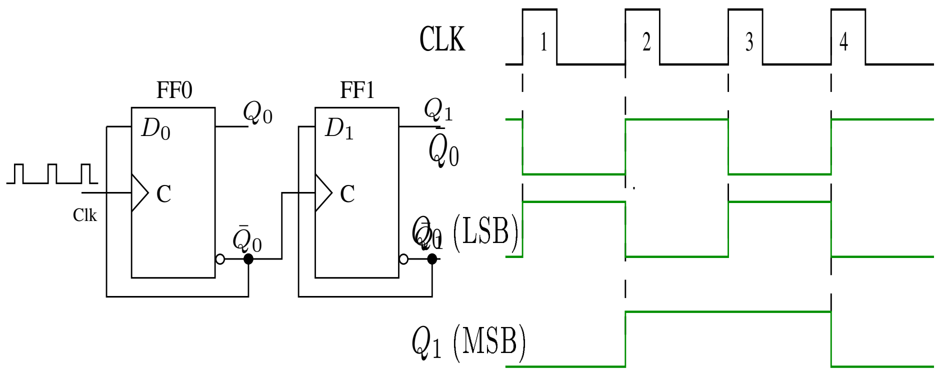


Figure 7.13: Circuit diagram 2-bit Synchronous Counter with JK and D flipflop implementations.

clocked at the same time by a common clock. D flip-flops can also be used but generally this implementation requires more logic as it does not have a *toggle* or *no-change* states. The basic structure of 2-bit synchronous counters with JK-flipflop or D- flipflop are illustrated below

Assume, initially counter is zero '0' binary state, i.e. both flipflops are in reset state. When positive edge of the first clock is applied, FF0 will toggle and Q_0 will go HIGH; however on FF1, at the positive edge of first clock inputs J_1 and K_1 would still be LOW, which is a no change condition and therefore FF1 does not change at first clock. One the second leading edge of the clock FF0 will toggle again and FF1 will set as HIGH.
Circuit diagram of 3 and 4 bit Synchronous counter are illustrated below

Insert the circuit diagram of 3 and 4 bit synchronous counters here.

7.9 Up/Down Synchronous Counters

An up/down counter is one that is capable of progressing in either direct through a certain sequence. An up/down counter, sometimes called bi-directional counter, can have a specified sequence of states.

Fig. shows a basic implementation of a 3-bit up/down binary counter. Notice that control input is HIGH for UP and LOW for DOWN.

Insert the circuit diagram of 3 bit UP/Down counters here.