Unit 4

Digital Electronics Fundamentals

CHAPTER 1

FUNDAMENTAL CONCEPTS

1.1 INTRODUCTION

All of us are familiar with the impact of modern digital computers, communication systems, digital display systems, internet, email etc. on society. One of the main causes of this revolution is the advent of *integrated circuits* (*ICs*), which became possible because of the tremendous progress in semiconductor technology in recent years. Most of us may not be familiar with the principles of working of computers, communication systems, internet, email, etc. even though these have become an important part of our daily life. The operation of these systems, and many other systems, is based on the principles of digital techniques and these systems are referred to as *digital systems*.

Some of us are familiar with electronic amplifiers. These are used to amplify electrical signals. This type of signals are continuous signals and can have any value in a limited range and are known as *analog signals*. The electronic circuits used to process (amplify) these signals are known as *analog circuits* and the systems built around this kind of operation are known as *analog systems*.

On the other hand, in an electronic calculator, the input is given with the help of switches. This is converted into electrical signals which have two discrete values or levels. One of these may be called as LOW level and the other one as HIGH level. The signal will always be of one of the two levels. Here, the actual value of the signal is immaterial as long as it is within the specified range of LOW or HIGH level. This type of signal is known as a *digital signal* and the circuits inside the calculator used to process these signals are known as *digital circuits*. A calculator is an example of a *digital system*.

There has been an unprecedented growth in digital techniques since Claude Shannon systematised and adapted George Boole's theoretical work in 1938. Developments in semiconductor technology, together with the progress in digital techniques, brought about a revolution in digital electronics when a single chip device known as *microprocessor* was introduced by Intel Corporation of America in 1971.

Since the introduction of microprocessors, the digital systems have gained tremendous power and importance. There is no field of knowledge which has affected our lives as much as the digital theory and applications, in such a short span of time. It has, infact, created a culture which nobody could have imagined till a few years ago. The rate of growth in this field has been unprecedented and the digital technology has become the most powerful technology for all future innovations in every walk of human endeavour whether it is computers, communication systems, information systems, entertainment and consumer products, business,

banking and finance, office machines, homes, cars, education, industrial control systems, scientific and medical instruments, and defence equipment, etc.

Some of the principal reasons for the widespread use of digital techniques and systems are:

- The devices used in digital circuits generally operate in one of the two *states*, known as ON and OFF resulting in a very simple operation.
- There are only a few basic operations in digital circuits which are very easy to understand.
- Digital techniques require Boolean algebra which is very simple and can easily be learnt even in schools.
- Digital circuits require basic concepts of electric network analysis which can easily be learnt at the junior level in colleges. The principal electrical characteristics required are *switching speed* and *loading* considerations. On the other hand, analog circuits and systems involve frequency and time domain concepts, complicated circuit analysis techniques, etc. which make the understanding of these circuits much more difficult than the digital circuits.
- A large number of ICs are available for performing various operations. These are highly reliable, accurate, small in size and the speed of operation is very high. A number of programmable ICs are also available.
- Various ICs are available in a *logic family* with similar electrical characteristics which make the design and development of digital systems very simple and also reduces interfacing problems. Also, a number of logic families based on different technologies are available which help in optimising the system design from the point of view of power requirement and speed of operation.
- The effect of fluctuations in the characteristics of the components, ageing of components, temperature, and noise, etc. is very small in digital circuits.
- Digital circuits have capability of memory which makes these circuits highly suitable for computers, calculators, watches, telephones, etc.
- The display of data and other information is very convenient, accurate and elegant using digital techniques.
- Many students have an opportunity to learn programming of digital computers, hence they have a strong motivation to study the way the digital hardware works.
- It is a very fascinating and challenging field of study because most of the latest electronic systems are digital in nature.

1.2 DIGITAL SIGNALS

As mentioned above, a digital signal has two discrete levels or values. Two different representations of digital signals are shown in Fig. 1.1. In each case there are two discrete levels. These levels can be represented using the terms LOW and HIGH. In Fig. 1.1*a*, lower of the two levels has been designated as LOW level and the higher as HIGH level. In contrast to this, in Fig. 1.1*b*, higher of the two levels has been designated as LOW level and the lower as HIGH level. Digital systems using the representation of signal shown in Fig. 1.1*a* are said to employ *positive logic system* and those using the other representation of the signal shown in Fig. 1.1*b* are said to employ *negative logic system*. The genesis of the term *logic* is given later. In each of the two signals we observe that the voltage corresponding to a given level is not fixed; rather voltages in a limited range are designated as a level. As long as the voltage belongs to a level it will be taken as that level and the exact value of the voltage is immaterial. For example, any voltage in the range of 3.5 to 5 V will be considered as HIGH level in the positive logic system and LOW level in the negative logic system. Similarly, any voltage in the range of 0 to 1 V will be considered as LOW level in the positive logic system and HIGH level in the negative logic system. The actual voltage ranges corresponding to LOW and HIGH level are not

same for all types of circuits and are different for different logic families (see Chapter 4). Unless otherwise specified, we shall be dealing with positive logic system.

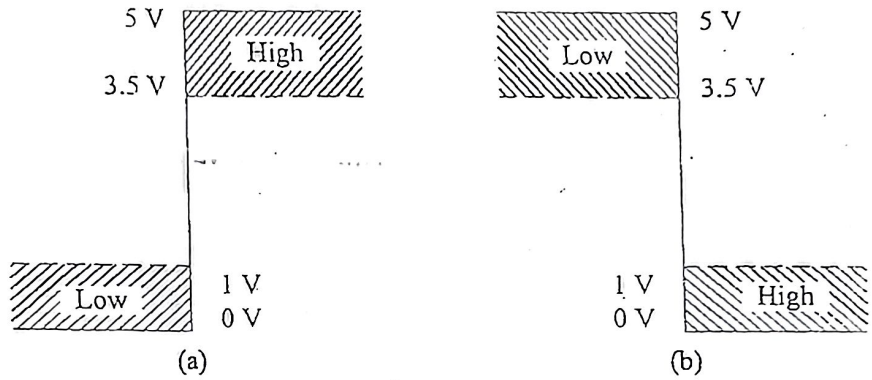


Fig. 1.1 *Digital Signal Representation (a) Positive Logic (b) Negative Logic*

The above discussion brings out one of the main advantages of digital systems, viz. they are less susceptible to noise, fluctuations in the characteristics of components, etc.

The two discrete signal levels HIGH and LOW can also be represented by the *binary digits* 1 and 0 respectively. A *binary digit* (0 or 1) is referred to as a *bit*. Since a digital signal can have only one of the two possible levels 1 or 0, the *binary number system* (see Chapter 2) can be used for the analysis and design of digital systems. The two levels (or states) can also be designated as ON and OFF or TRUE and FALSE. George Boole introduced the concept of binary number system in the studies of the mathematical theory of LOGIC in the work entitled *An Investigation of the Laws of Thought* in 1854 and developed its algebra known as *Boolean algebra*. These logic concepts have been adapted for the design of digital hardware since 1938 when Claude Shannon-organised and systematised Boole's work in *Symbolic Analysis of Relay and Switching Circuits*.

1.3 BASIC DIGITAL CIRCUITS

In a digital system there are only a few basic operations performed, irrespective of the complexities of the system. These operations may be required to be performed a number of times in a large digital system like digital computer or a digital control system, etc. The basic operations are AND, OR, NOT, and FLIP-FLOP. The AND, OR, and NOT operations are discussed here and the FLIP-FLOP, which is a basic memory element used to store binary information (one bit is stored in one FLIP-FLOP), will be introduced in Chapter 7.

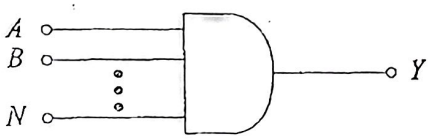


Fig. 1.2 *The Standard Symbol for an* AND *Gate*

1.3.1 The AND Operation

A circuit which performs an AND operation is shown in Fig. 1.2. It has N inputs ($N \geq 2$) and one output. Digital signals are applied at the input terminals marked $A, B, \ldots, N$, the other terminal being ground, which is not shown in the diagram. The output is obtained at the output terminal marked Y (the other terminal being ground) and it is also a digital signal. The AND operation is defined as: the output of an AND gate is 1 if and only if all the inputs are 1. Mathematically, it is written as

$$
\begin{aligned}
Y &= A \text{ AND } B \text{ AND } C \ldots \text{ AND } N \\
&= A \cdot B \cdot C \cdot \ldots \cdot N \\
&= ABC \ldots N
\end{aligned} \tag{1.1}
$$

where A, B, C,... N are the input variables and Y is the output variable. The variables are binary, i.e. each variable can assume only one of the two possible values, 0 or 1. The *binary variables* are also referred to as *logical variables*.

Equation (1.1) is known as the *Boolean equation* or the *logical equation* of the AND *gate*. The term gate is used because of the similarity between the operation of a digital circuit and a gate. For example, for an AND operation the gate opens ($Y = 1$) only when all the inputs are present, i.e. at logic 1 level.

Truth Table Since a logical variable can assume only two possible values (0 and 1), therefore, any logical operation can also be defined in the form of a table containing all possible input combinations (2^N combinations for N inputs) and their corresponding outputs. This is known as a *truth table* and it contains one row for each one of the input combinations.

For an AND gate with two inputs A, B and the output Y, the truth table is given in Table 1.1. Its logical equation is $Y = AB$ and is read as "Y equals A AND B".

Table 1.1 *Truth Table of a 2-Input* AND *Gate*

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Since, there are only two inputs, A and B, therefore, the possible number of input combinations is four. It may be observed from the truth table that the input–output relationship for a digital circuit is completely specified by this table in contrast to the input–output relationship for an analog circuit. The pattern in which the inputs are entered in the truth table may also be observed carefully, which is in the ascending order of binary numbers formed by the input variables. (See Chapter 2).

Logical Multiplication The AND operation is also referred to as logical multiplication and therefore, it is symbolised algebraically by a multiplication dot (·) as illustrated in Eq. (1.1).

Example 1.1

You have rented a locker in a bank. Express the process of opening the locker in terms of a digital operation.

Solution

The locker door (Y) can be opened by using one key (A) which is with you and the other key (B) which is with the bank executive. When both the keys are used, the locker door opens, i.e., the locker door can be opened ($Y = 1$) only when both the keys are applied ($A = B = 1$). Thus, this process can be expressed as an AND operation

$$Y = A \cdot B.$$

Example 1.2

The voltage waveforms shown in Fig 1.3 are applied at the inputs of a 2-input AND gate. Determine the output waveform.

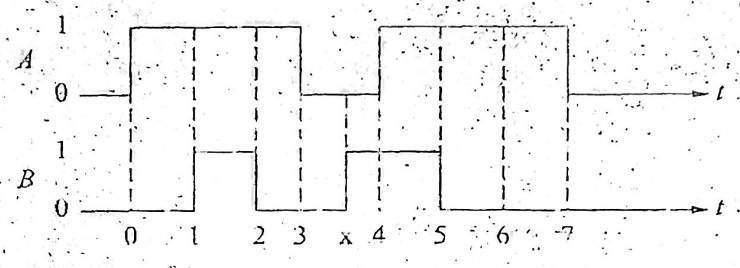


Fig 1.3

Solution

Using Table 1.1, we find.

From t = 0 to t = 1
$A = 1, B = 0$
Therefore, $Y = 0$

From t = 1 to t = 2
$A = B = 1$
Therefore, $Y = 1$

From t = 2 to t = 3
$A = 1, B = 0$
Therefore, $Y = 0$

From t = 3 to t = x
$A = 0, B = 0$
Therefore, $Y = 0$

From t = x to t = 4
$A = 0, B = 1$
Therefore, $Y = 0$

From t = 4 to t = 5
$A = 1, B = 1$
Therefore, $Y = 1$

From t = 5 to t = 7
$A = 1, B = 0$
Therefore, $Y = 0$. It will be 0 for $t > 7$

The output waveform with reference to the input waveforms are shown in Fig. 1.4.

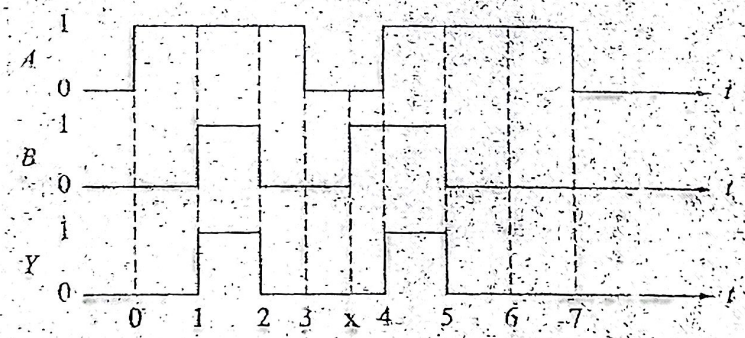


Fig. 1.4

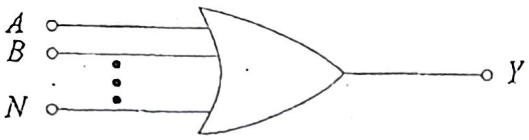


Fig. 1.5 *The Standard Symbol for an* OR *Gate*

1.3.2 The OR Operation

Figure 1.5 shows an OR gate with N inputs ($N \geq 2$) and one output. The OR operation is defined as: the output of an OR gate is 1 if and only if one or more inputs are 1. Its logical equation is given by

$$Y = A \text{ OR } B \text{ OR } C \ldots \text{ OR } N = A + B + C + \ldots + N \qquad (1.2)$$

The truth table of a 2-input OR gate is given in Table 1.2. Its logic equation is $Y = A + B$ and is read as "Y equals A OR B".

Table 1.2 *Truth Table of a 2-Input* OR *Gate*

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Example 1.3

In a chemical process an ALARM is required to be activated if either temperature or pressure or both exceed certain limits. Is it possible to express this operation in terms of a digital operation? If yes, find the operation.

Solution

Let the temperature and pressure be converted into electrical signals and $T = 1$ if temperature exceeds the specified limit and $P = 1$ if pressure exceeds the specified limit. If $T = 1$ or $P = 1$ or both T and P are 1 then the ALARM is required to be activated, i.e., the signal applied to the ALARM $Y = 1$. This operation can be expressed as

$$Y = T \text{ OR } P$$
$$= T + P$$

Which is an OR operation

Example 1.4

If the waveforms of Fig. 1.3 are applied at the inputs of a 2-input OR gate, determine the output waveform.

Solution

Using Table 1.2, we find

From $t = 0$ to $t = 1$
$A = 1, B = 0$
Therefore, $Y = 1$

From $t = 1$ to $t = 2$
$A = 1, B = 1$
Therefore, $Y = 1$

From $t = 2$ to $t = 3$
$A = 1, B = 0$
Therefore, $Y = 1$

From $t = 3$ to $t = x$
$A = 0, B = 0$
Therefore, $Y = 0$

From t = x to t = 4
$A = 0, B = 1$
Therefore, $Y = 1$

From t = 4 to t = 5
$A = 1, B = 1$
Therefore, $Y = 1$

From t = 5 to t = 7
$A = 1, B = 0$
Therefore, $Y = 1$. It is 0 for $t > 7$.

The output waveform with reference to the input waveforms are shown in Fig. 1.6

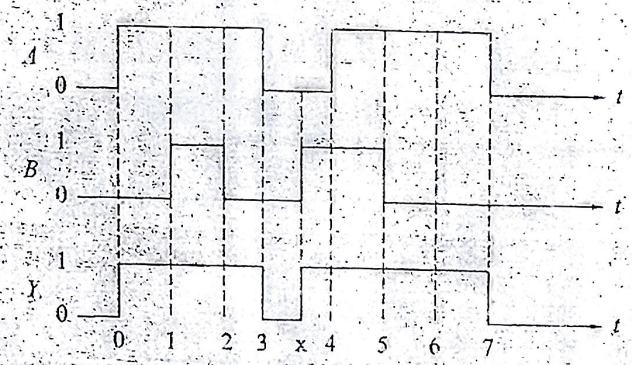


Fig. 1.6

1.3.3 The NOT Operation

Figure 1.7 shows a NOT gate, which is also known as an *inverter*. It has one input (A) and one output (Y). Its logic equation is written as

$$Y = \text{NOT } A = \overline{A} \tag{1.3}$$

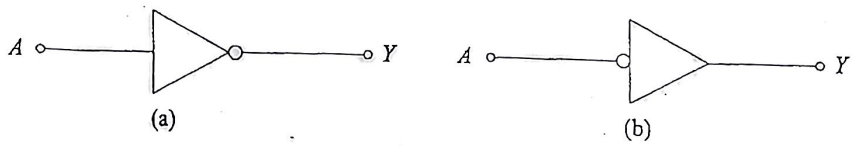


Fig. 1.7 *The Standard Symbols for a* NOT *Gate*

and is read as "Y equals NOT A" or "Y equals complement of A". The truth table of a NOT gate is given in Table 1.3.

The NOT operation is also referred to as an *inversion* or *complementation*. The presence of a small circle, known as the *bubble*, always denotes inversion in digital circuits.

Table 1.3 *Truth Table of a* NOT *Gate*

Input A	Output Y
0	1
1	0

Example 1.5

If the waveform shown in Fig. 1.8 is applied at the input of an inverter, find its output waveform

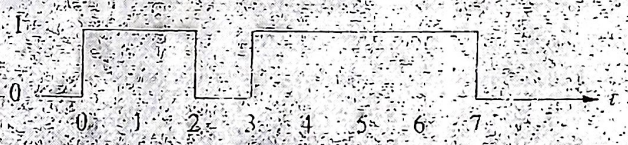


Fig. 1.8

Solution

Using Table 1.3, we find

From $t = 0$ *to* $t = 2$
$A = 1$
Therefore, $Y = \text{NOT } A = \text{NOT } 1 = 0$

From $t = 2$ *to* $t = 3$
$A = 0$
Therefore, $Y = \text{NOT } A = \text{NOT } 0 = 1$

From $t = 3$ *to* $t = 7$
$A = 1$
Therefore, $Y = \text{NOT } A = \text{NOT } 1 = 0$
Y will be 1 for $t > 7$

The output waveform with reference to the input waveform is shown in Fig. 1.9

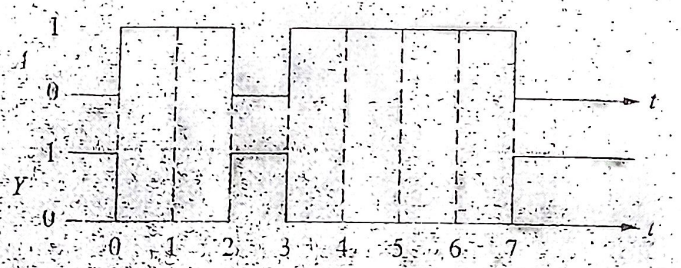


Fig. 1.9

1.4 NAND AND NOR OPERATIONS

Any Boolean (or logic) expression can be realised by using the AND, OR and NOT gates discussed above. From these three operations, two more operations have been derived: the NAND operation and NOR operation.

These operations have become very popular and are widely used, the reason being the only one type of gates, either NAND or NOR are sufficient for the realisation of any logical expression. Because of this reason, NAND and NOR gates are known as *universal gates*.

1.4.1 The NAND Operation

The NOT–AND operation is known as the NAND operation. Figure 1.10*a* shows an N input ($N \geq 2$) AND gate followed by a NOT gate. The operation of this circuit can be described in the following way:

The output of the AND gate (Y') can be written using Eq. (1.1)

$$Y' = AB \ldots N \tag{1.4}$$

Now, the output of the NOT gate (Y) can be written using Eq. (1.3)

$$Y = \overline{Y'} = \overline{AB \ldots N} \tag{1.5}$$

The logical operation represented by Eq. (1.5) is known as the NAND operation. The standard symbol of the NAND gate is shown in Fig. 1.10*b*. Here, a bubble on the output side of the NAND gate represents NOT operation, inversion or complementation.

The truth table of a 2-input NAND gate is given in Table 1.4. Its logic equation is $Y = \overline{A\ B}$ and, is read as "Y equals NOT (A AND B)".

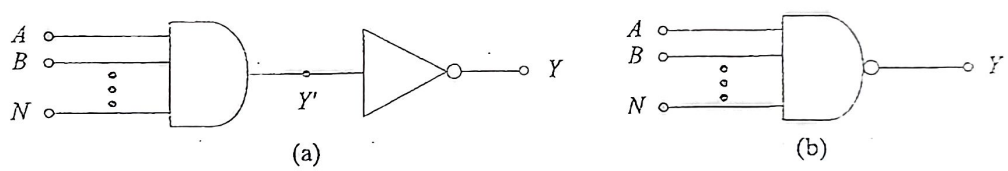


Fig. 1.10 (*a*) NAND *Operation as* NOT–AND *Operation*,
(*b*) *Standard Symbol for the* NAND *Gate*

Table 1.4 *Truth Table of a 2-Input* NAND *Gate*

Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

The three basic logic operations, AND, OR and NOT can be performed by using only NAND gates. These are given in Fig. 1.11.

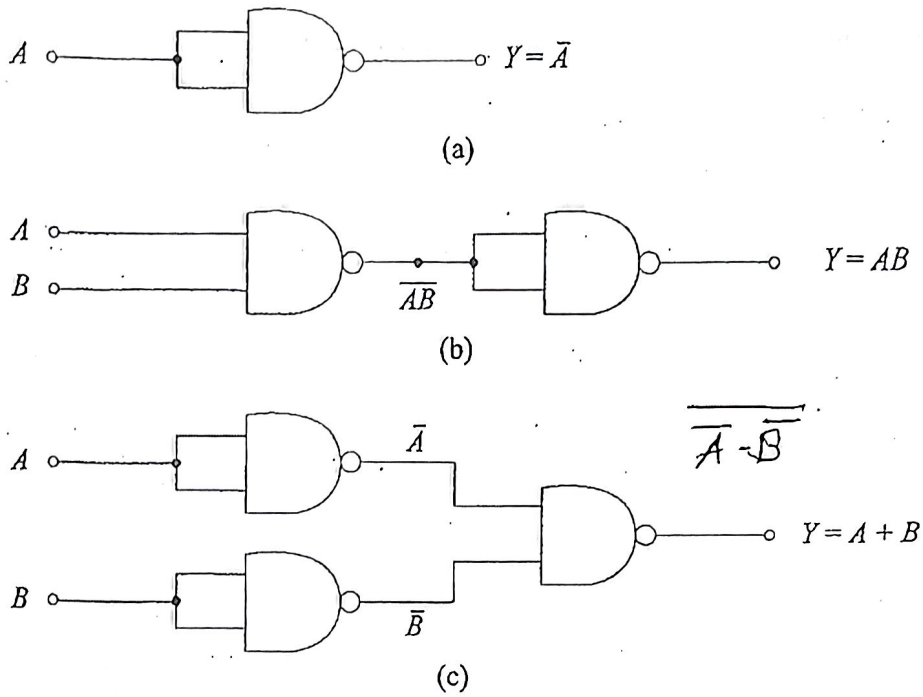


Fig. 1.11 *Realisation of Basic Logic Operations Using* NAND *Gates (a)* NOT *(b)* AND *(c)* OR

Example 1.6

If the voltage waveforms of Fig. 1.3 are applied at the inputs of a 2-input NAND gate, find the output waveform.

Solution

The output waveform will be inverse of the output waveform Y of Fig. 1.4.

Example 1.7

What will be the output of a 2-input NAND gate, if one of its inputs is permanently connected to

(a) logic 0 voltage

(b) logic 1 voltage

Solution

(a) Figure 1.12*a* shows a 2-input NAND gate with one of its inputs connected to logic 0 voltage.

In this circuit if $A = 0$, $Y = 1$. Similarly, if $A = 1$, then also the output $Y = 1$.

This shows that the output Y is 1, irrespective of the other input. In fact, a NAND gate is *disabled* or *inhibited* if one of its inputs is connected to logic 0.

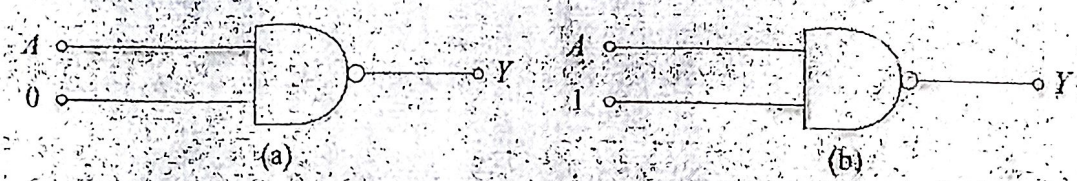


Fig. 1.12 *Circuits for Example 1.7*

(b) Figure 1.12*b* shows a 2-input NAND gate with one of its inputs connected to logic 1 voltage.
Here, if $A = 1$, $Y = 0$,
and if $A = 0$, $Y = 1$
This means $Y = \overline{A}$

This condition *enables* a NAND gate

1.4.2 The NOR Operation

The NOT–OR operation is known as the NOR operation. Figure 1.13*a* shows an N input ($N \geq 2$) OR gate followed by a NOT gate. The operation of this circuit can be described in the following way:

The output of the OR gate Y' can be written using Eq. (1.2) as

$$Y' = A + B + \ldots + N \tag{1.6}$$

and the output of the NOT gate (Y) can be written using Eq. (1.3)

$$Y = \overline{Y'} = \overline{A + B + \ldots + N} \tag{1.7}$$

The logic operation represented by Eq. (1.7) is known as the NOR operation.

The standard symbol of the NOR gate is shown in Fig. 1.13*b*. Similar to the NAND gate, a bubble on the output side of the NOR gate represents the NOT operation.

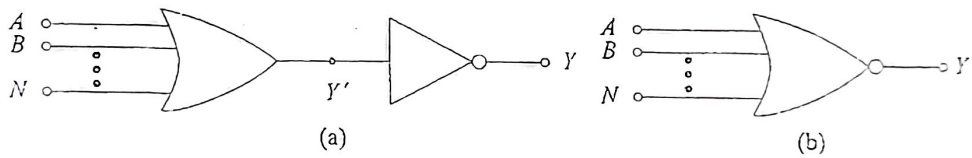


Fig. 1.13 *(a)* NOR *Operation as* NOT–OR *Operation*
(b) Standard Symbol for the NOR *Gate*

Table 1.5 gives the truth table of a 2-input NOR gate. Its logic equation is $Y = \overline{A + B}$ and is read as "Y equals NOT (A OR B)".

Table 1.5 *Truth Table of a 2-Input* NOR *Gate*

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

The three basic logic operations, AND, OR, and NOT can be performed by using only the NOR gates. These are given in Fig. 1.14.

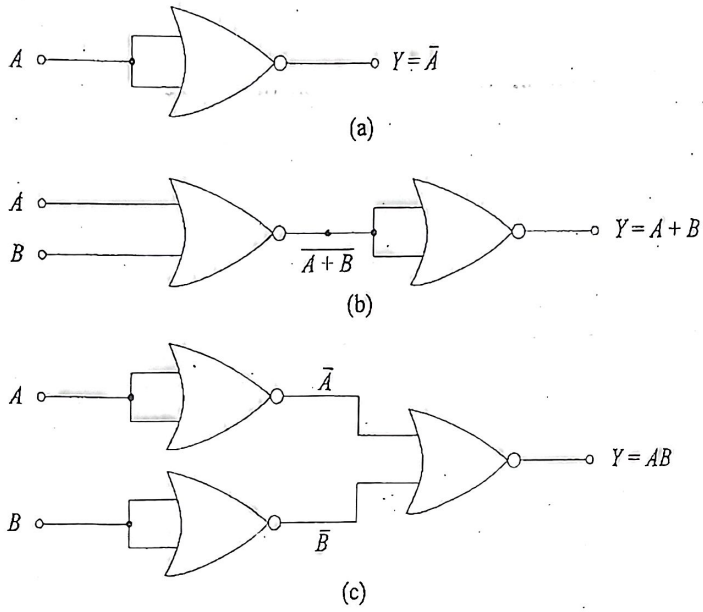


Fig. 1.14 *Realisation of Basic Logic Operations Using* NOR *Gates (a)* NOT *(b)* OR *(c)* AND

Example 1.8

If the voltage waveforms of Fig. 1.3 are applied at the inputs of a 2-input NOR gate, determine the output waveform.

Solution

The output waveform will be inverse of the output waveform of Fig. 1.6.

Example 1.9

(a) If one of the inputs of a NOR gate is connected to logic 0 voltage, find the output voltage in terms of the other input.
(b) Repeat (a) if one of the inputs is connected to logic 1 voltage.

Solution

(a) From Table 1.5, we observe that if $A = 0$ then $Y = \bar{B}$. This operation is used for enabling a NOR gate.
(b) Similarly from Table 1.5, we observe that if $A = 1$, then $Y = 0$. This operation inhibits or disables a NOR gate. Here, the output is 0 irrespective of the B input.

1.5 EXCLUSIVE-OR AND EXCLUSIVE-NOR OPERATIONS

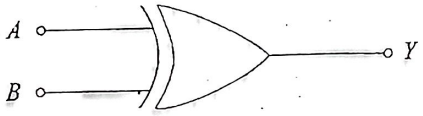


Fig. 1.15 *Standard Symbol for* EX-OR *Gate*

1.5.1 The EXCLUSIVE-OR Operation

The EXCLUSIVE-OR (EX-OR) operation is widely used in digital circuits. It is not a basic operation and can be performed using the basic gates—AND, OR and NOT or universal gates NAND or NOR. Because of its importance, the standard symbol shown in Fig. 1.15 is used for this operation and it is treated as basic logic element.

The truth table of an EX–OR gate is given in Table 1.6 and its logic equation is written as

$$Y = A \text{ EX–OR } B = A \oplus B \tag{1.8}$$

Table 1.6 *Truth Table of* EX–OR *Gate*

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

If we compare the truth table of an EX–OR gate with that of an OR gate given in Table 1.2, we find that the first three rows are same in both. Only the fourth row is different. This circuit finds application where two digital signals are to be compared. From the truth table we observe that when both the inputs are same (0 or 1) the output is 0, whereas when the inputs are not same (one of them is 0 and the other one is 1) the output is 1.

Example 1.10

The voltage waveforms, shown in Fig 1.3, are applied at the inputs of an EX–OR gate. Determine the output waveform

Solution

Using Table 1.6, we find

From $t = 0$ *to* $t = 1$
$A = 1, B = 0$
Therefore, $Y = 1$

From $t = 1$ *to* $t = 2$
$A = 1, B = 1$
Therefore, $Y = 0$

From $t = 2$ *to* $t = 3$
$A = 1, B = 0$
Therefore, $Y = 1$

From $t = 3$ *to* $t = x$
$A = 0, B = 0$
Therefore, $Y = 0$

From $t = x$ *to* $t = 4$
$A = 0, B = 1$
Therefore, $Y = 1$

From $t = 4$ *to* $t = 5$
$A = 1, B = 1$
Therefore, $Y = 0$

From $t = 5$ *to* $t = 7$
$A = 1, B = 0$
Therefore, $Y = 1$

For $t > 7$
$A = 0, B = 0$
Therefore, $Y = 0$

The output waveform with reference to the input waveforms are shown in Fig. 1.16.

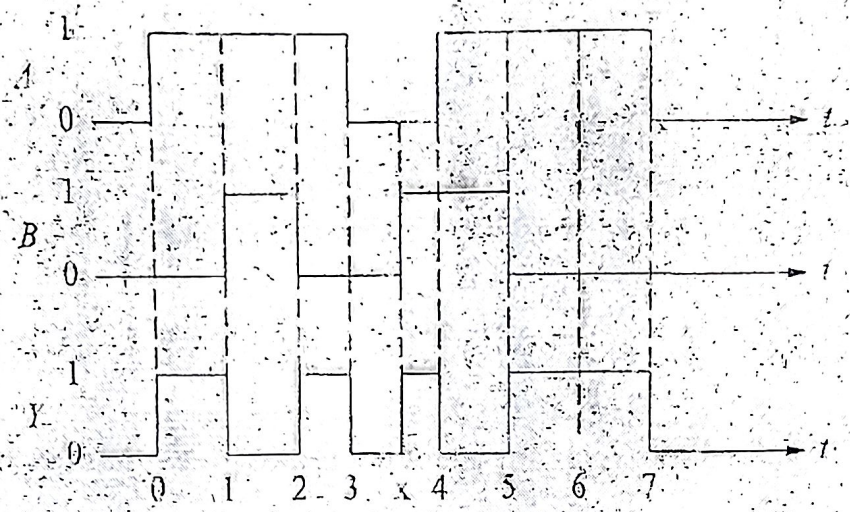


Fig. 1.16

Example 1.11

Determine the relation between the output Y and one of its inputs, if its other input is connected to

(a) logic 0

(b) logic 1

Solution

(a) Let us connect input A to 0; from the first two rows of the Table 1.6, we observe that $Y = B$.

(b) Similarly if input A is connected to logic 1 voltage, from the last two rows of the truth table, we observe that

$Y = \overline{B}$

1.5.2 The EXCLUSIVE-NOR Operation

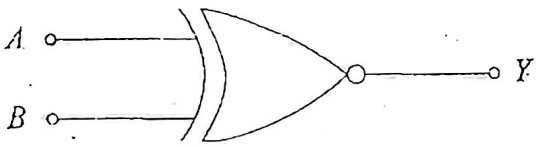


Fig. 1.17 *Standard Symbol for* EX-NOR *Gate*

Figure 1.17 shows the standard symbol of EXCLUSIVE-NOR (EX-NOR) gate. Its truth table is given in Table 1.7.

Its logic operation is specified as

$$Y = A \text{ EX-NOR } B = \overline{A \text{ EX-OR } B} = \overline{A \oplus B} = A \odot B \qquad (1.9)$$

Similar to EX-OR gate, EX-NOR gate is not a basic operation and can be performed using the basic gates or universal gates, but because of its importance, it has been given a standard symbol. The EX-NOR operation is also referred to as the *coincidence* operation because it produces output of 1 when its inputs coincide in value, both 0 or both 1.

Table 1.7 *Truth Table of* EX-NOR *Gate*

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Example 1.12

Repeat Example 1.10 for EX-NOR gate.

Solution

The output waveform will be inverse of the output waveform, as shown in Fig. 1.16.

Example 1.13

Repeat Example 1.11 for EX-NOR gate.

Solution

(a) $Y = \overline{B}$

(b) $Y = B$

1.6 BOOLEAN ALGEBRA

As discussed above, the digital signals are discrete in nature and can only assume one of the two values 0 or 1. A number system based on these two digits is known as *binary number system*. In the middle of 19th century, an English mathematician George Boole developed rules for manipulations of binary variables, known as *Boolean algebra*. This is the basis of all digital systems like computers, calculators, etc.

Binary variables can be represented by a letter symbol such as $A, B, X, Y, \ldots$ The variable can have only one of the two possible values at any time, viz. 0 or 1. The Boolean algebraic theorems are given in Table 1.8.

From these theorems, we observe that the even numbered theorems can be obtained from their preceding odd numbered theorems by (*i*) interchanging + and · signs, and (*ii*) interchanging 0 and 1.

Theorems which are related in this way are called *duals*.

Table 1.8 *Boolean Algebraic Theorems*

Theorem No.	Theorem
1.1	$A + 0 = A$
1.2	$A \cdot 1 = A$
1.3	$A + 1 = 1$
1.4	$A \cdot 0 = 0$
1.5	$A + A = A$
1.6	$A \cdot A = A$
1.7	$A + \overline{A} = 1$
1.8	$A \cdot \overline{A} = 0$
1.9	$A \cdot (B + C) = AB + AC$
1.10	$A + BC = (A + B)(A + C)$
1.11	$A + AB = A$

(Continued)

Table 1.8 *(Continued)*

Theorem No.	Theorem
1.12	$A(A+B)=A$
1.13	$A+\bar{A}B=(A+B)$
1.14	$A(\bar{A}+B)=AB$
1.15	$AB+A\bar{B}=A$
1.16	$(A+B)(A+\bar{B})=A$
1.17	$AB+\bar{A}C=(A+C)(\bar{A}+B)$
1.18	$(A+B)(\bar{A}+C)=AC+\bar{A}B$
1.19	$AB+\bar{A}C+BC=AB+\bar{A}C$
1.20	$(A+B)(\bar{A}+C)(B+C)=(A+B)(A+C)$
1.21	$\overline{A\cdot B\cdot C\cdots}=\bar{A}+\bar{B}+\bar{C}+\cdots$
1.22	$\overline{A+B+C+\cdots}=\bar{A}\cdot\bar{B}\cdot\bar{C}\cdots$

Theorems 1.1 to 1.8 involve a single variable only. Each of these theorems can be proved by considering every possible value of the variable. For example, in Theorem 1.1,

if $A=0$ then $0+0=0=A$

and if $A=1$ then $1+0=1=A$

and hence the theorem is proved.

Theorems 1.9 to 1.20 involve more than one variable and can be proved by making a truth table. For example, Theorem 1.10 can be proved by making the truth table given in Table 1.9.

Table 1.9 *Truth Table to Prove Theorem 1.10*

A	B	C	BC	$A+BC$	$A+B$	$A+C$	$(A+B)\cdot(A+C)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

From the table we observe that there are 8 (= 2^3) possible combinations of the three variables A, B, and C. For each combination, the value of $A + BC$ is the same as that of $(A + B)(A + C)$, which proves the theorem. Theorems 1.21 and 1.22 are known as De Morgan's theorems. These theorems can be proved by first considering the two variable case and then extending this result. From the truth table given in Table 1.10 we get the relations

$$\overline{A \cdot B} = \overline{A} + \overline{B} \qquad (1.10)$$

73236

and

$$\overline{A + B} = \overline{A} \cdot \overline{B} \qquad (1.11)$$

Table 1.10 *Truth Table to Prove De Morgan's Theorems*

A	B	$\overline{A}$	$\overline{B}$	$\overline{AB}$	$\overline{A}+\overline{B}$	$\overline{A+B}$	$\overline{A}\ \overline{B}$
0	0	1	1	1	1	1	1
0	1	1	0	1	1	0	0
1	0	0	1	1	1	0	0
1	1	0	0	0	0	0	0

Now consider the NAND operation of three variables,

$$\begin{aligned}\overline{ABC} &= \overline{(AB) \cdot C} \\ &= \overline{(A \cdot B)} + \overline{C} \\ &= \overline{A} + \overline{B} + \overline{C} \quad \text{using Eq. (1.10)}\end{aligned} \qquad (1.12)$$

In a similar way, the NOR operation of three variables gives

$$\begin{aligned}\overline{A + B + C} &= \overline{(A + B) + C} \\ &= \overline{(A + B)} \cdot \overline{C} \\ &= \overline{A} \cdot \overline{B} \cdot \overline{C} \quad \text{using Eq. (1.11)}\end{aligned} \qquad (1.13)$$

The above results can be easily extended to any number of variables.

A logic problem can be specified in terms of a set of statements. This set of statements can be represented in terms of an equation called the logic equation or in terms of a truth table. A digital circuit using the gates discussed above can be designed to realise a logic equation. In general, it is possible to simplify (minimise) a logic equation. The minimised logic equation will probably need less number of gates and/or less number of inputs for the gates. The techniques used to minimise logic equations will be discussed in Chapter 5. An example of the realisation of a circuit for a given logic equation is given below:

Example 1.14

Realise (design) a digital circuit for the logic equation

$$Y = \bar{A} \cdot B + A \cdot \bar{B} \tag{1.14}$$

Solution

This equation consists of two input variables A and B and one output variable Y. The variable A appears as $\bar{A}$ in the term $\bar{A} \cdot B$ and as A in the term $A \cdot \bar{B}$. Similarly the variable B appears as $\bar{B}$ in the term $\bar{A} \cdot B$ and as B in the term $A \cdot \bar{B}$. The two terms $\bar{A} \cdot B$ and $A \cdot \bar{B}$ are obtained by AND operations and the output Y is the result of OR operation performed on $\bar{A} \cdot B$ and $A \cdot \bar{B}$ terms. The realisation of this equation is obtained using NOT, AND and OR gates as shown in Fig. 1.18.

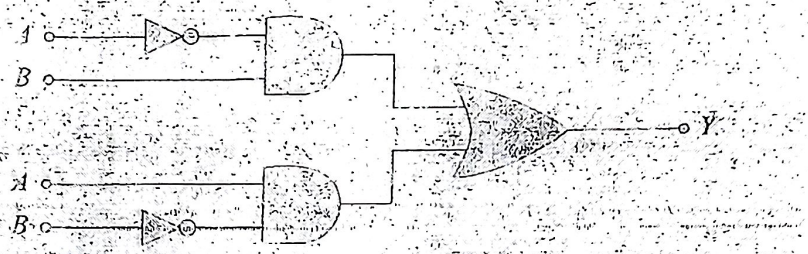


Fig. 1.18 *Realisation of Eq. (1.14)*

1.7 EXAMPLES OF IC GATES

All the logic functions introduced in this chapter are commercially available in integrated circuit (IC) form. For example, 7400 IC chip is a quadruple 2-input NAND gate available in 14-pin DIP. It has four identical, independent 2-input NAND gates arranged as shown in Fig. 1.19. It requires a +5 V d.c. supply (to be connected between V_{CC} and GND pins) for the operation of the gates. Table 1.11 gives some of the available gate ICs. The details of their pin connexions, electrical characteristics, etc. can be obtained from the manufacturers' data catalogues.

Some of the manufacturers are:

- Texas Instruments (www.ti.com)
- Philips (www.philips.com)
- Fairchild semiconductor (www.fairchildsemi.com)

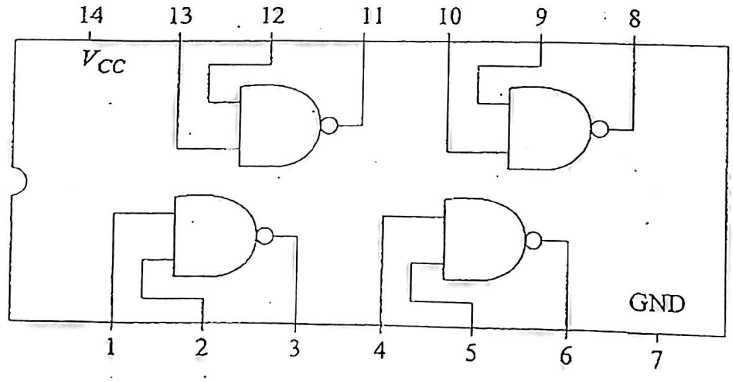


Fig. 1.19 *Block Diagram of 7400 IC*

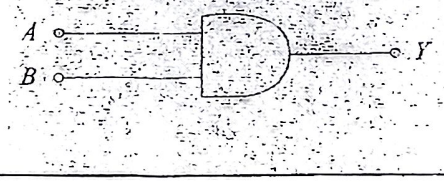
Table 1.11 *Some of the Available IC*

IC No.	Description
7400	Quad 2-input NAND gates
7402	Quad 2-input NOR gates
7404	Hex inverters
7408	Quad 2-input AND gates
7410	Triple 3-input NAND gates
7411	Triple 3-input AND gates
7420	Dual 4-input NAND gates
7421	Dual 4-input AND gates
7427	Triple 3-input NOR gates
7430	8-input NAND gate
7432	Quad 2-input OR gates
7486, 74386	Quad EX–OR gates
74133	13-input NAND gate
74135	Quad EX-OR/NOR gates
74260	Dual 5-input NOR gates

SUMMARY

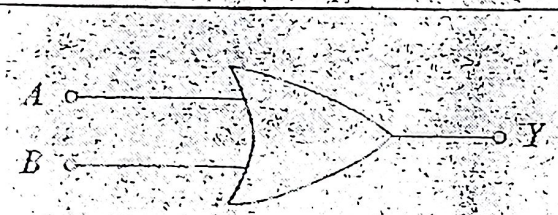
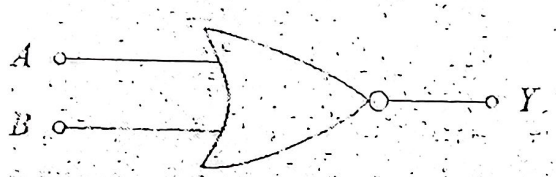
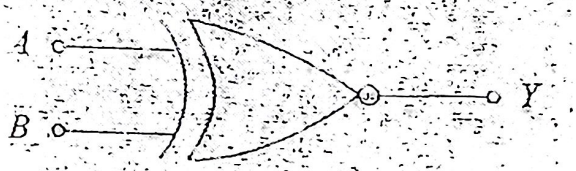
In this chapter, the basic concepts of the digital systems have been discussed. The basic features and advantages of these systems have been given briefly. The level of the treatment has been kept low to avoid any confusion. Table 1.12 summarises the operation of all the gates introduced in this chapter. For convenience, two input gates have been taken and the different symbols used for various operations are also given. A brief exposure to Boolean algebra has also been given. The techniques for the simplification of logic equations will be discussed in Chapter 5.

Table 1.12 ***Summary of Logic Gates***

Gate	Logic diagram	Function	Truth table		
			Inputs		Output
			A	B	Y
AND		$Y = A$ AND B $= A \cdot B$ $= A \cap B$ $= A \wedge B$ $= AB$	0	0	0
			0	1	0
			1	0	0
			1	1	1

(*Continued*)

Table 1.12 *(Continued)*

Gate	Logic diagram	Function	Truth table
OR		$Y = A \text{ OR } B$ $= A + B$ $= A \cup B$ $= A \vee B$	Inputs: A, B; Output: Y 0 0 0 0 1 1 1 0 1 1 1 1
NOT (inverter)		$Y = \text{NOT } A$ $= \overline{A}$	Input: A; Output: Y 0 1 1 0
NAND		$Y = A \text{ NOT AND } B$ $= A \text{ NAND } B$ $= \overline{A \cdot B}$ $= \overline{A \cap B}$ $= \overline{A \wedge B}$ $= A \uparrow B$ $= \overline{AB}$	Inputs: A, B; Output: Y 0 0 1 0 1 1 1 0 1 1 1 0
NOR		$Y = A \text{ NOT OR } B$ $= A \text{ NOR } B$ $= \overline{A + B}$ $= \overline{A \cup B}$ $= \overline{A \vee B}$ $= A \downarrow B$	Inputs: A, B; Output: Y 0 0 1 0 1 0 1 0 0 1 1 0
EX-OR		$Y = A \text{ EX-OR } B$ $= A \oplus B$ $= A\overline{B} + \overline{A}B$	Inputs: A, B; Output: Y 0 0 0 0 1 1 1 0 1 1 1 0
EX-NOR		$Y = \overline{A \text{ EX-OR } B}$ $= A \text{ EX-NOR } B$ $= \overline{A \oplus B}$ $= \overline{A\overline{B} + \overline{A}B}$ $= \overline{A}\overline{B} + AB$	Inputs: A, B; Output: Y 0 0 1 0 1 0 1 0 0 1 1 1

CHAPTER 2

NUMBER SYSTEMS AND CODES

2.1 INTRODUCTION

We all are familiar with the number system in which an ordered set of ten symbols—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, known as *digits*—are used to specify any number. This number system is popularly known as the *decimal number system*. The *radix* or *base* of this number system is 10 (number of distinct digits). Any number is a collection of these digits. For example, 1982.365 signifies a number with an *integer* part equal to 1982 and a *fractional* part equal to 0.365, separated from the integer part with a *radix point* (.) also known as *decimal point*. There are some other systems also, used to represent numbers. Some of the other commonly used number systems are: *binary, octal* and *hexadecimal* number systems. These number systems are widely used in digital systems like microprocessors, logic circuits, computers, etc. and therefore, the knowledge of these number systems is very essential for understanding, analysing and designing digital systems.

As discussed in Chapter 1, computers and other digital circuits use binary signals but are required to handle data which may be numeric, alphabets or special characters. Therefore, the information available in any other form is required to be converted into suitable binary form before it can be processed by digital circuits. This means that the information available in the form of numerals, alphabets and special characters or in any combination of these must be converted into binary format. To achieve this, a process of coding is employed whereby each numeral, alphabet or special character is coded in a unique combination of 0s and 1s using a coding scheme, known as a *code*. The process of coding is known as *encoding*.

There can be a variety of coding schemes (codes) to serve different purposes, such as arithmetic operations, data entry, error detection and correction, etc. In digital systems, a large number of codes are in use. Selection of a particular code depends on its suitability for the purpose. In one digital system, different codes may be used for different operations and it may be necessary to convert data from one code to another code. For this purpose, code converter circuits are required which will be discussed later.

2.2 NUMBER SYSTEMS

In general, in any number system there is an ordered set of symbols known as digits with rules defined for performing arithmetic operations like addition, multiplication, etc. A collection of these digits makes a number which in general has two parts—integer and fractional, set apart by a radix point (.), that is

$$(N)_b = \underbrace{d_{n-1}\, d_{n-2} \ldots d_i \ldots d_1\, d_0}_{\text{Integer portion}} \underset{\substack{\uparrow \\ \text{Radix point}}}{.} \underbrace{d_{-1}\, d_{-2} \ldots d_{-f} \ldots d_{-m}}_{\text{Fractional portion}} \tag{2.1}$$

where N = a number
b = radix or base of the number system
n = number of digits in integer portion
m = number of digits in fractional portion
d_{n-1} = most significant digit (msd)
d_{-m} = least significant digit (lsd)

and

$$0 \le (d_i \text{ or } d_{-f}) \le b - 1$$

The digits in a number are placed side by side and each position in the number is assigned a *weight* or *index* of importance by some predesigned rule. Table 2.1 gives the details of commonly used number systems.

Table 2.1 *Characteristics of Commonly Used Number Systems*

Number system	Base or radix (b)	Symbols used (d_i or d_{-f})	Weight assigned to position i	Weight assigned to position $-f$	Example
Binary	2	0, 1	2^i	2^{-f}	1011.11
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	8^i	8^{-f}	3567.25
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	10^i	10^{-f}	3974.57
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E*, *F*	16^i	16^{-f}	3*FA*9.56

2.3 BINARY NUMBER SYSTEM

The number system with base (or radix) two is known as the *binary number system*. Only two symbols are used to represent numbers in this system and these are 0 and 1. These are known as bits. This system has the minimum base (0 is not possible and 1 is not useful). It is a positional system, that is every position is assigned a specific weight.

Table 2.2 illustrates counting in binary number system. The corresponding decimal numbers are given in the right-hand column. Similar to decimal number system, the left-most bit is known as the *most significant bit* (MSB) and the right-most bit is known as the *least significant bit* (LSB). Any number of 0s can be added to the left of the number without changing the value of the number. In the binary number system, a group of four bits is known as a *nibble*, and a group of eight bits is known as a *byte*.

Table 2.2 *4-bit Binary Numbers and Their Corresponding Decimal Numbers*

Binary number				Decimal number	
B_3	B_2	B_1	B_0	D_1	D_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	0	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	0	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	2
1	1	0	1	1	3
1	1	1	0	1	4
1	1	1	1	1	5

2.3.1 Binary-to-Decimal Conversion

Any binary number can be converted into its equivalent decimal number using the weights assigned to each bit position as given in Table 2.1.

Example 2.1

Find the decimal equivalent of the binary number (1 1 1 1 1).

Solution

The equivalent decimal number is

$$= 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 16 + 8 + 4 + 2 + 1$$
$$= (31)_{10}$$

To differentiate between numbers represented in different number systems, either the corresponding number system may be specified along with the number or a small subscript at the end of the number may

be added signifying the number system. For example, $(1000)_2$ represents a binary number and is not one thousand.

Example 2.2

Determine the decimal numbers represented by the following binary numbers:

(a) 110101 (b) 101101 (c) 11111111 (d) 00000000

Solution

(a) $(110101)_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 32 + 16 + 0 + 4 + 0 + 1$
$= (53)_{10}$

(b) $(101101)_2 = 32 + 0 + 8 + 4 + 0 + 1$
$= (45)_{10}$

(c) $(11111111)_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$
$= (255)_{10}$

(d) $(00000000)_2 = (0)_{10}$

Example 2.3

Determine the decimal numbers represented by the following binary numbers:

(a) 101101.10101 (b) 1100.1011 (c) 1001.0101 (d) 0.10101

Solution

(a) $(101101.10101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$
$= 32 + 0 + 8 + 4 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} + 0 + \frac{1}{32}$
$= (45.65625)_{10}$

(b) $(1100.1011)_2 = 8 + 4 + 0 + 0 + 0.5 + 0 + 0.125 + 0.0625$
$= (12.6875)_{10}$

(c) $(1001.0101)_2 = 8 + 0 + 0 + 1 + 0 + 0.25 + 0 + 0.0625$
$= (9.3125)_{10}$

(d) $(0.10101)_2 = 0.5 + 0 + 0.125 + 0 + 0.03125$
$= (0.65625)_{10}$

2.3.2 Decimal-to-Binary Conversion

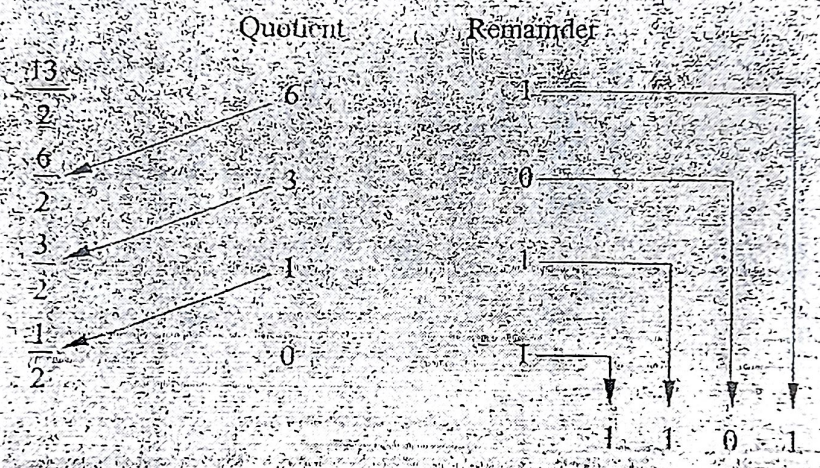
Any decimal number can be converted into its equivalent binary number. For integers, the conversion is obtained by continuous division by 2 and keeping track of the remainders, while for fractional parts, the

conversion is affected by continuous multiplication by 2 and keeping track of the integers generated. The conversion process is illustrated by the following examples.

Example 2.4

Convert $(13)_{10}$ to an equivalent base-2 number.

Solution

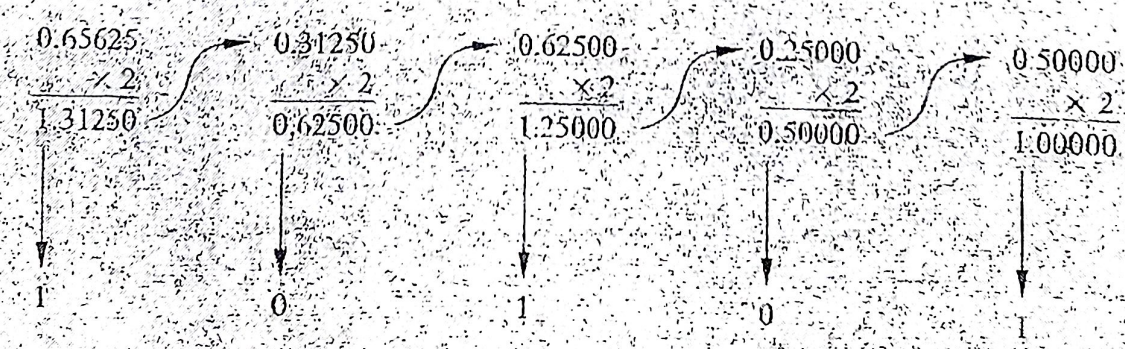


Thus, $(13)_{10} = (1101)_2$

Example 2.5

Convert $(0.65625)_{10}$ to an equivalent base-2 number.

Solution



Thus, $(0.65625)_{10} = (0.10101)_2$

Example 2.6

Express the following decimal numbers in the binary form:

(a) 25.5 (b) 10.625 (c) 0.6875

Solution

(a) *Integer part*

	Quotient	Remainder
$\frac{25}{2}$	12	1
$\frac{12}{2}$	6	0
$\frac{6}{2}$	3	0
$\frac{3}{2}$	1	1
$\frac{1}{2}$	0	1

1 1 0 0 1

Therefore, $(25)_{10} = (11001)_2$

Fractional part

$$\begin{array}{r} 0.5 \\ \times\ 2 \\ \hline 1.0 \end{array}$$

1

i.e., $(0.5)_{10} = (0.1)_2$

Therefore, $(25.5)_{10} = (11001.1)_2$

(b) *Integer part* $(10)_{10} = (1010)_2$

Fractional part

$$\begin{array}{r} 0.625 \\ \times\ 2 \\ \hline 1.250 \end{array} \qquad \begin{array}{r} 0.250 \\ \times\ 2 \\ \hline 0.500 \end{array} \qquad \begin{array}{r} 0.500 \\ \times\ 2 \\ \hline 1.000 \end{array}$$

1 0 1

i.e., $(0.625)_{10} = (0.101)_2$

Therefore, $(10.625)_{10} = (1010.101)_2$

2.5 BINARY ARITHMETIC

We all are familiar with the arithmetic operations such as addition, subtraction, multiplication, and division of decimal numbers. Similar operations can be performed on binary numbers; infact, binary arithmetic is much simpler than decimal arithmetic because here only two digits, 0 and 1 are involved.

2.5.1 Binary Addition

The rules of binary addition are given in Table 2.4.

Table 2.4 *Rules of Binary Addition*

Augend	Addend	Sum	Carry	Result
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	10

In the first three rows above, there is no *carry*, that is, carry = 0, whereas in the fourth row a carry is produced (since the largest digit possible is 1), that is, carry = 1, and similar to decimal addition it is added to the next higher binary position.

Example 2.13

Add the binary numbers:

(i) 1011 and 1100 (ii) 0101 and 1111

Solution

(i)

```
      1  0  1  1
(+)   1  1  0  0
   ------------
   1  0  1  1  1
   ↑
 carry
```

(ii)

```
        (1) (1) (1)  ← carry
         0   1   0   1
(+)      1   1   1   1
   ------------------
   1     0   1   0   0
   ↑
 carry
```

Example 2.14

Add the binary numbers.

```
01101010
00001000
10000001
11111111
```

Solution

				1				← Two pair of 1's in the previous column
	(1)	(1)	(1)	1	(1)	(1)	(1)	← one pair of 1's in the previous column
	0	1	1	0	1	0	1	0
	0	0	0	0	1	0	0	0
	1	0	0	0	0	0	0	1
	1	1	1	1	1	1	1	1
1	1	1	1	1	0	0	1	0
Carry					↑	↑		↑ Even number of 1's in column
	↑	↑	↑	↑			↑	odd number of 1's in column

∴ The sum = 1 1 1 1 1 0 0 1 0

From the above example, we observe the following:

(i) if the number of 1's to be added in a column is even then the sum bit is 0, and if the number of 1's to be added in a column is odd then the sum bit is 1.
(ii) Every pair of 1's in a column produces a carry (1) to be added to the next higher bit column.

2.5.2 Binary Subtraction

The rules of binary subtraction are given in Table 2.5.

Table 2.5 *Rules of Binary Subtraction*

Minuend	Subtrahend	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Except in the second row above, the *borrow* = 0. When the *borrow* = 1, as in the second row, this is to be subtracted from the next higher binary bit as it is done in decimal subtraction.

Example 2.15

Perform the following subtraction:

$$\begin{array}{r} 1011 \\ -\ 0110 \\ \hline \end{array}$$

Solution

	Column 4	3	2	1	
	1	0	1	1	Minuend
(–)	0	1	1	0	Subtrahend
	0	1	0	1	Difference

Here, in columns 1 and 2, borrow = 0 and in column 3 it is 1. Therefore, in column 4 first subtract 0 from 1 and from this result obtained subtract the borrow bit.

2.5.3 Binary Multiplication

Binary multiplication is similar to decimal multiplication. In binary, each partial product is either zero (multiplication by 0) or exactly same as the multiplicand (multiplication by 1). An example of binary multiplication is given below:

Example 2.16

Multiply 1001 by 1101.

Solution

$$\begin{array}{rl} 1001 & \text{Multiplicand} \\ \times\ 1101 & \text{Multiplier} \\ \hline 1001 & \text{I} \\ 0000 & \text{II} \quad \text{Partial Products} \\ 1001 & \text{III} \\ 1001 & \text{IV} \\ \hline 1110101 & \text{Final Product} \end{array}$$

In a digital circuit, the multiplication operation is performed by repeated additions of all partial products to obtain the final product.

entered as 376 using keys. Since digital circuits can process only zeros and ones, the octal numbers have to be converted into binary form using special circuits known as octal-to-binary converters before being processed by the digital circuits.

2.8 HEXADECIMAL NUMBER SYSTEM

Hexadecimal number system is very popular in computer uses. The base for hexadecimal number system is 16 which requires 16 distinct symbols to represent the numbers. These are numerals 0 through 9 and alphabets A through F. Since numeric digits and alphabets both are used to represent the digits in the hexadecimal number system, therefore, this is an alphanumeric number system. Table 2.7 gives hexadecimal numbers with their binary equivalents for decimal numbers 0 through 15. From the table, it is observed that there are 16 combinations of 4-bit binary numbers and sets of 4-bit binary numbers can be entered in the computer in the form of hexadecimal (hex.) digits. These numbers are required to be converted into binary representation, using hexadecimal-to-binary converter circuits before these can be processed by the digital circuits.

Table 2.7 *Binary and Decimal Equivalents of Hexadecimal Numbers*

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

2.8.1 Hexadecimal-to-Decimal Conversion

Using Eq. (2.1), hexadecimal numbers can be converted to their equivalent decimal numbers.

Example 2.28

Obtain decimal equivalent of hexadecimal number $(3A.2F)_{16}$

Solution

Using Eq. (2.1),

$$(3A.2F)_{16} = 3 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 15 \times 16^{-2}$$
$$= 48 + 10 + \frac{2}{16} + \frac{15}{16^2}$$
$$= (58.1836)_{10}$$

The fractional part may not be an exact equivalent and therefore, may give a small error.

2.8.2 Decimal-to-Hexadecimal Conversion

For conversion from decimal to hexadecimal, the procedure used in binary as well as octal systems is applicable, using 16 as the dividing (for integer part) and multiplying (for fractional part) factor.

Example 2.29

Convert the following decimal numbers into hexadecimal numbers.

(a) 95.5 (b) 675.625

Solution

(a) *Integer part*

	Quotient	Remainder
95/16	5	15
5/16	0	5

5 F

Thus, $(95)_{10} = (5F)_{16}$

Fractional part

$$\begin{array}{r} 0.5 \\ \times\ 16 \\ \hline 8.0 \end{array}$$
$$\downarrow$$
$$8$$

Thus, $(0.5)_{10} = (0.8)_{16}$

Therefore, $(95.5)_{10} = (5F.8)_{16}$

(b) *Integer part*

	Quotient	Remainder
675/16	42	3
42/16	2	10
2/16	0	2

2 A 3

Thus, $(675)_{10} = (2A3)_{16}$

Fractional part

$$\begin{array}{r} 0.625 \\ \times 16 \\ \hline 10.000 \end{array}$$

$\downarrow$

A

Thus, $(0.625)_{10} = (0.A)_{16}$

Therefore, $(675.625)_{10} = (2A3.A)_{16}$

2.8.3 Hexadecimal-to-Binary Conversion

Hexadecimal numbers can be converted into equivalent binary numbers by replacing each hex digit by its equivalent 4-bit binary number.

Example 2.30

Convert $(2F9A)_{16}$ to equivalent binary number.

Solution

Using Table 2.7, find the binary equivalent of each hex digit.

$$(2F9A)_{16} = (0010\ 1111\ 1001\ 1010)_2$$
$$= (0010111110011010)_2$$

2.8.4 Binary-to-Hexadecimal Conversion

Binary numbers can be converted into the equivalent hexadecimal numbers by making groups of four bits starting from LSB and moving towards MSB for integer part and then replacing each group of four bits by its hexadecimal representation.

CHAPTER 5

COMBINATIONAL LOGIC DESIGN

5.1 INTRODUCTION

Logic operations and Boolean algebra have already been discussed in Chapter 1. Boolean algebraic theorems are used for the manipulations of logic expressions. It has also been demonstrated that a logic expression can be realised using the logic gates. The number of gates and the number of input terminals for the gates required for the realisation of a logic expression, in general, get reduced considerably if the expression can be simplified. Therefore, the simplification of logic expression is very important as it saves the hardware required to design a specific system. A large number of functions are available in IC form and therefore, we should be able to make optimum use of these ICs in the design of digital systems. That is, our aim should be to minimise the number of IC packages. Some of the logic gates available in ICs have been discussed in Section 1.7. We will be discussing in Chapter 6, some of the MSI digital circuits available in IC form and design of digital systems using these ICs. Programmable logic devices (PLDs), field-programmable gate arrays (FPGAs) and the design using these devices have been discussed in Chapter 12.

Basically, digital circuits are divided into two broad categories:

1. Combinational circuits, and
2. Sequential circuits.

In *combinational circuits*, the outputs at any instant of time depend upon the inputs present at that instant of time. This means there is no memory in these circuits. There are other types of circuits in which the outputs at any instant of time depend upon the present inputs as well as past inputs/outputs. This means that there are elements used to store past information. These elements are known as *memory*. Such circuits are known as *sequential circuits*. A sequential logic system may have combinational logic sub-systems. The design of combinational circuits will be discussed here. Sequential circuits design will be discussed later.

The design requirements of combinational circuits may be specified in one of the following ways:

1. A set of statements,
2. Boolean expression, and
3. Truth table.

The aim is to design a circuit using the gates already discussed or some other circuits which are in fact derived from the basic gates. As is usual in any engineering design, the number of components used should be minimum to ensure low cost, saving in space, power requirements, etc. There can be two different approaches to the design of combinational circuits. One of these is the traditional method, wherein the given Boolean expression or the truth table is simplified by using standard methods and the simplified expression is realised using the gates. The other method normally does not require any simplification of the logic expression or truth table, instead the complex logic functions available in medium scale integrated circuits (MSI) can be directly used. Computer-aided design (CAD) tools are used for the design using PLDs and FPGAs. Combinational circuit design using the traditional design methods have been discussed below, however, the concepts used in these methods will help in the understanding of design using MSIs, PLDs, and FPGAs, etc.

The following methods can be used to simplify the Boolean functions:

1. Algebraic method,
2. Karnaugh-map technique,
3. Quine–McCluskey method, and
4. Variable entered mapping (VEM) technique.

The algebraic method, the Karnaugh map ~~(K-map) technique, and the~~ Quine-McCluskey method have been discussed here. The K-map is the simplest and most commonly used method. It is a manual method and depends to a great extent upon human intuition. This method can be used conveniently up to six variables beyond which it is very cumbersome.

The Quine–McCluskey method is suitable for computer mechanisation and is seldom used by logic designers manually. The variable entered mapping (VEM) has been discussed in Fletcher[2] and the interested reader can refer to it.

5.2 STANDARD REPRESENTATIONS FOR LOGIC FUNCTIONS

Logic functions are expressed in terms of logical variables. The values assumed by the logic functions as well as the logic variables are in the binary form. Any arbitrary logic function can be expressed in the following forms:

1. Sum-of-products form (SOP), and
2. Product-of-sums form (POS)

This does not mean that the logic function cannot be written in any other form. It can be written in various forms but the above two forms are conveniently suited in arriving at the standard methods for designing the circuits which will become clear from the following discussions.

Example 5.1

Given the logic equation

$$Y = (A + BC)(\overline{B} + \overline{C}A) \tag{5.1}$$

(a) Design a circuit using gates to realise this function.

[2] Fletcher, W.L., *An Engineering Approach to Digital Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

(b) Find out whether it is possible to design the circuit with only one type of gates (NAND or NOR). If yes, design the circuits.

(c) Find out whether it is possible to simplify this equation. If yes, simplify it.

(d) Now design the circuit using the simplified expression obtained in part (c).

(e) Compare the circuits obtained in parts (a), (b), and (d) from the point of view of number of gates, number of inputs for the gates, types of gates, and propagation delay.

Solution

In Eq. (5.1), there are three input logic variables A, B, and C, and Y is the output. The variable C appears as C in one term and as $\overline{C}$ in the other term. A variable in *uncomplemented* or *complemented* form is known as a *literal*.

(a) A circuit using gates can simply be designed by looking at the expression and finding out the basic gates which can be used to realise the various terms and then connect these gates appropriately. We assume that the signals corresponding to each literal are available, that is the variables are available in uncomplemented as well as the complemented form.

(i) The first term (A) has only one literal A and the second term (BC) has two literals B and C. The second term is recognised as an AND operation and can be realised by using a 2-input AND gate. The combination of these two terms is realised by using a 2-input OR gate. The complete realisation of first two terms is shown in Fig. 5.1*a*.

(ii) The third term (B) is again a single literal term and the fourth term ($\overline{C}A$) has two literals. The combination of these two terms is similar to the combination of the first two terms and is realised in a similar way. This realisation is shown in Fig. 5.1*b*.

(iii) Now, the complete realisation is obtained by using a 2-input AND gate with Y_1 and Y_2 as the inputs and the output of this gate will be the required output Y. This realisation is given in Fig. 5.1*c*. The above design requires three 2-input AND gates and two 2-input OR gates.

(b) (i) Sum-of-Products Form

Equation (5.1) can be written as

$$Y = A(B + \overline{C}A) + (BC)(B + \overline{C}A) \qquad \text{(Theorem 1.9)}$$

$$= AB + A\overline{C}A + BCB + BC\overline{C}A \qquad \text{(Theorem 1.9)} \qquad (5.2)$$

$$= AB + A\overline{C} + BC \qquad (5.3)$$

Equation (5.3) is obtained from Eq. (5.2) in the following way.

$$A\overline{C}A = (A \cdot A) \cdot \overline{C} = A\overline{C} \qquad \text{(Theorem 1.6)}$$

$$BCB = (B \cdot B) \cdot C = BC \qquad \text{(Theorem 1.6)}$$

$$BC\overline{C}A = B \cdot (C \cdot \overline{C}) \cdot A = B \cdot 0 \cdot A \qquad \text{(Theorem 1.8)}$$

$$= 0 \qquad \text{(Theorem 1.4)}$$

This term does not appear in Eq. (5.3), because of Theorem 1.1.

The representation of Eq. (5.3) is known as *sum-of-products* (SOP) form. This can be realised using AND–OR configuration as shown in Fig. 5.2*a*. This realisation is known as *two-level realisation*. The first level consists of AND gates and the second level consists of the OR gate. By making use of the De-Morgan's theorem (Theorem 1.22) we can write Eq. (5.3) as

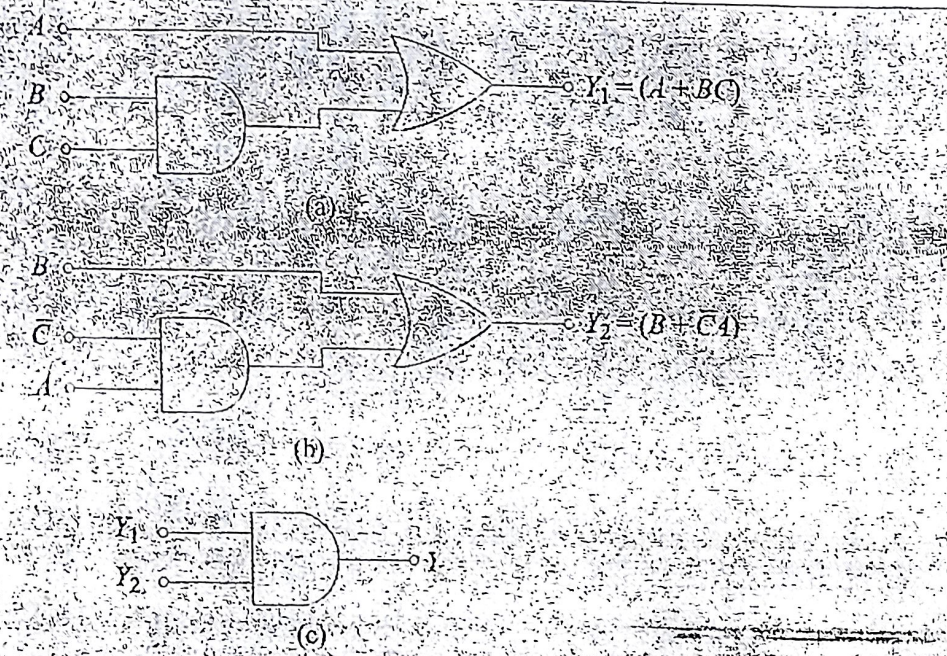


Fig. 5.1 *Logic Circuits for the Realisation of Eq. (5.1)*

$$\overline{Y} = \overline{AB + A\overline{C} + BC}$$

$$= \overline{AB} \cdot \overline{A\overline{C}} \cdot \overline{BC}$$

Or

$$Y = \overline{Y_1 \cdot Y_2 \cdot Y_3} \tag{5.4}$$

Where

$$Y_1 = \overline{AB}$$

$$Y_2 = \overline{A\overline{C}}$$

$$Y_3 = \overline{BC}$$

Equation (5.4) can be realised using NAND gates only. The realisation is given in Fig. 5.2*b*. This is also a two-level realisation and in this only NAND gates are used. Therefore, if we express the equation in the SOP form we can always design the circuit using only one type of gates (NAND).

(ii) Product-of-Sums Form

It is possible to represent Eq. (5.1) in another form.

$$Y = (A + B)(A + C)(B + \overline{C})(B + A) \quad \text{(Theorem 1.10)}$$

$$= (A + B)(A + C)(B + \overline{C}) \quad \text{(Theorem 1.6)} \tag{5.5}$$

The representation of Eq. (5.5) is known as *Product-of-sums* (POS) form. This can be realised using OR-AND gates as shown in Fig. 5.3*a*. This is also a two-level realisation

Using De-Morgan's theorem (Theorem 1.21), we can write Eq. (5.5) as

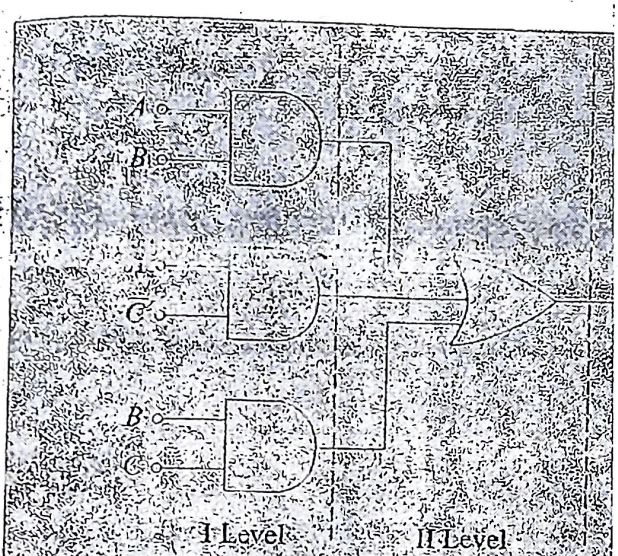


Fig. 5.2 *Realisation of Eq.*

$$Y = (A +$$

$$= (A +$$

Or

$$Y = Y_A +$$

Where

$$Y_A = A + B$$

$$Y_b = \overline{A + C}$$

$$Y_c = \overline{B + C}$$

Equation (5.6) can be realised using NOR g level realisation. Hence, if we express the e one type of gates (NOR).

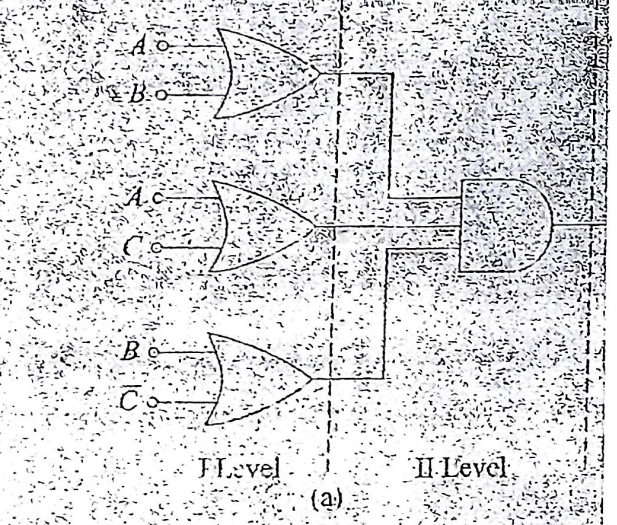


Fig. 5.3 *Realisation of Eq*

We notice that in the SOP form of Eq. (5.3) and the involve all the three literals. If each term in SOP and PO as *canonical* SOP and POS, respectively. Each ind and in *canonical* POS form as *maxterm*. The usefulnes become clear from the discussion which follows. S the terms in the expression with terms formed by OR present in that term. For example for a three-variable ex A, where B and C variables are missing, then we fo Therefore, we get $A \cdot (B + \bar{B}) \cdot (C + \bar{C}) = ABC + A\bar{B}C +$

Example 5.2

Convert Eq. (5.3) into canonical SOP form.

Solution

$$Y = AB(C + \bar{C}) + A\bar{C}(B + \bar{B}) + B$$

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

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

Similarly, POS form can be converted to canonic formed by ANDing the variable and its complement w three-variable expression with variables A, B, and C, if then we form two terms $B \times \bar{B}$ and $C \times \bar{C}$ and OR A

$$A + B\bar{B} + C\bar{C} = (A + B\bar{B} + C)(A$$

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

Example 5.3

Convert Eq. (5.5) into canonical POS form.

Solution

$$Y = (A + B + C\bar{C})(A + B\bar{B} + C)(A$$

$$= (A + B + C)(A + B + \bar{C})(A +$$

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

$$= (A + B + C)(A + B + \bar{C})(A +$$

The concept of minterm and maxterm introduced above allows us to introduce a very convenient shorthand notation to express logical functions. Table 5.2 gives the minterms and maxterms for a four variable logical function where the number of minterms as well as maxterms is $2^4 = 16$. In general, for an n-variable logical function there are 2^n minterms and an equal number of maxterms. While writing a particular minterm or maxterm, we shall always write it with the variables in an orderly way as can be seen from Table 5.2. Each minterm is represented by m_i where the subscript i is the decimal equivalent of the natural binary number corresponding to the minterm with normal (uncomplemented) variables taken as 1's and the complemented variables taken as 0's.

Table 5.2 *Minterms/Maxterms for Four Variables*

Variable				Minterm	Maxterm
A	B	C	D	m_i	M_i
0	0	0	0	$\bar{A}\bar{B}\bar{C}\bar{D} = m_0$	$A+B+C+D = M_0$
0	0	0	1	$\bar{A}\bar{B}\bar{C}D = m_1$	$A+B+C+\bar{D} = M_1$
0	0	1	0	$\bar{A}\bar{B}C\bar{D} = m_2$	$A+B+\bar{C}+D = M_2$
0	0	1	1	$\bar{A}\bar{B}CD = m_3$	$A+B+\bar{C}+\bar{D} = M_3$
0	1	0	0	$\bar{A}B\bar{C}\bar{D} = m_4$	$A+\bar{B}+C+D = M_4$
0	1	0	1	$\bar{A}B\bar{C}D = m_5$	$A+\bar{B}+C+\bar{D} = M_5$
0	1	1	0	$\bar{A}BC\bar{D} = m_6$	$A+\bar{B}+\bar{C}+D = M_6$
0	1	1	1	$\bar{A}BCD = m_7$	$A+\bar{B}+\bar{C}+\bar{D} = M_7$
1	0	0	0	$A\bar{B}\bar{C}\bar{D} = m_8$	$\bar{A}+B+C+D = M_8$
1	0	0	1	$A\bar{B}\bar{C}D = m_9$	$\bar{A}+B+C+\bar{D} = M_9$
1	0	1	0	$A\bar{B}C\bar{D} = m_{10}$	$\bar{A}+B+\bar{C}+D = M_{10}$
1	0	1	1	$A\bar{B}CD = m_{11}$	$\bar{A}+B+\bar{C}+\bar{D} = M_{11}$
1	1	0	0	$AB\bar{C}\bar{D} = m_{12}$	$\bar{A}+\bar{B}+C+D = M_{12}$
1	1	0	1	$AB\bar{C}D = m_{13}$	$\bar{A}+\bar{B}+C+\bar{D} = M_{13}$
1	1	1	0	$ABC\bar{D} = m_{14}$	$\bar{A}+\bar{B}+\bar{C}+D = M_{14}$
1	1	1	1	$ABCD = m_{15}$	$\bar{A}+\bar{B}+\bar{C}+\bar{D} = M_{15}$

Similar to minterms, each maxterm is represented by M_i, where the subscript i is the decimal equivalent of the natural binary number corresponding to the maxterm with uncomplemented variables taken as 0's and complemented variables taken as 1's. Using these notations the canonical SOP (Eq. (5.9)) can be written as:

$$\begin{aligned} Y &= m_3 + m_4 + m_6 + m_7 \\ &= \sum m(3, 4, 6, 7) \end{aligned} \tag{5.12}$$

Where $\bar{A}BC = m_3$

$A\bar{B}\bar{C} = m_4$

$AB\bar{C} = m_6$

and $ABC = m_7$

Similarly, the canonical POS (Eq. (5.11)) can be written as

$$Y = M_0 \cdot M_1 \cdot M_2 \cdot M_5$$
$$= \Pi M(0, 1, 2, 5) \tag{5.13}$$

Where $A + B + C = M_0$

$A + B + \bar{C} = M_1$

$A + \bar{B} + C = M_2$

and $\bar{A} + B + \bar{C} = M_5$

Equations (5.12) and (5.13) are the shorthand forms of canonical SOP and POS respectively. Since these two equations represent the same logical function (Eq. (5.1)), therefore we notice that there is a complementary type of relationship between a function expressed in terms of minterms and in terms of maxterms. Here, we are dealing with a three-variable function where the number of minterms and maxterms are $2^3 = 8$ and the corresponding decimal numbers are 0 through 7. Out of this the terms corresponding to decimal numbers 3, 4, 6, and 7 are minterms, and the terms corresponding to decimal numbers 0, 1, 2, and 5 are maxterms. Hence, if a logic function is specified in terms of minterm/maxterm, its maxterm/minterm representation can be determined by using this complementary property. For example, for a four-variable case if

$$Y = \Sigma m(0, 3, 6, 7, 10, 12, 15)$$

then $Y = \Pi M(1, 2, 4, 5, 8, 9, 11, 13, 14)$.

5.3 KARNAUGH MAP REPRESENTATION OF LOGIC FUNCTIONS

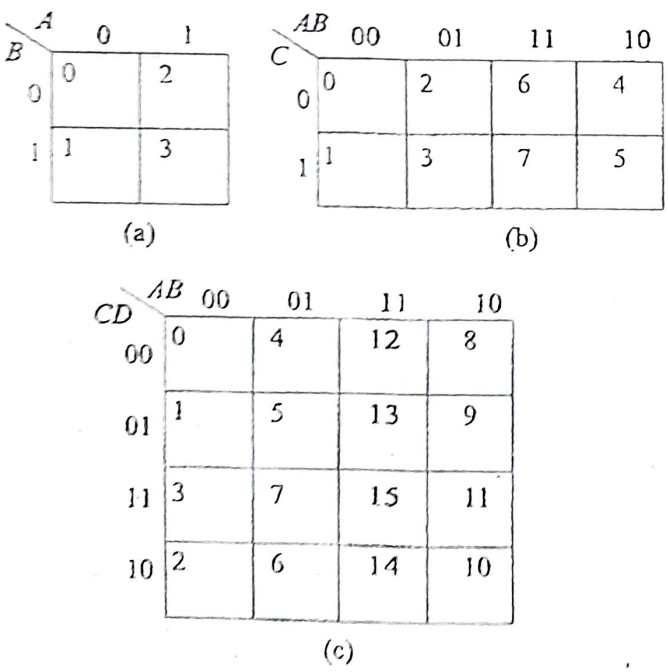


Fig. 5.5 *Karnaugh-maps (a) Two-variable (b) Three-variable (c) Four-variable*

We have discussed the two standard forms of logic functions and their realisations using gates. We have also established the need for the simplification of the Boolean expressions and introduced the algebraic method of simplification using Boolean algebraic theorems. Sometimes it is difficult to be sure that a logic expression can be simplified. There is another technique, which is graphical, known as the Karnaugh map technique which provides a systematic method for simplifying and manipulating Boolean expressions. In this technique, the information contained in a truth table or available in POS or SOP form is represented on Karnaugh map (K-map). This is perhaps the most extensively used tool for simplification of Boolean functions. Although the technique may be used for any number of variables, it is generally used up to six variables beyond which it becomes very cumbersome.

Figure 5.5 shows the K-maps for two, three, and four variables. In an n-variable K-map there are 2^n cells. Each cell corresponds to one of the combinations of n variables, since there are 2^n combinations of n variables. Therefore, we see that for each

row of the truth table, for each minterm and for each maxterm there is one specific cell in the K-map. The variables have been designated as A, B, C, and D, and the binary numbers formed by them are taken as AB, ABC, and $ABCD$ for two, three, and four variables respectively. In each map the variables and all possible values of the variables are indicated (the first bit corresponds to the first variable and the second bit corresponds to the second variable) to identify the cells. Gray code has been used for the identification of cells. The reason for using Gray code will become clear when we discuss the application of K-map. You can verify the decimal number corresponding to each cell which is written in the top left corner of the cell as shown in Fig. 5.5.

Figure 5.6 shows the minterm/maxterm corresponding to each cell and the term is written inside the cell for clear understanding.

B \ A	0	1
0	$\bar{A}\bar{B}$	$A\bar{B}$
1	$\bar{A}B$	AB

(a)

B \ A	0	1
0	$A+B$	$\bar{A}+B$
1	$A+\bar{B}$	$\bar{A}+\bar{B}$

(b)

C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

(c)

C \ AB	00	01	11	10
0	$A+B+C$	$A+\bar{B}+C$	$\bar{A}+\bar{B}+C$	$\bar{A}+B+C$
1	$A+B+\bar{C}$	$A+\bar{B}+\bar{C}$	$\bar{A}+\bar{B}+\bar{C}$	$\bar{A}+B+\bar{C}$

(d)

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

(e)

CD \ AB	00	01	11	10
00	$A+B+C+D$	$A+\bar{B}+C+D$	$\bar{A}+\bar{B}+C+D$	$\bar{A}+B+C+D$
01	$A+B+C+\bar{D}$	$A+\bar{B}+C+\bar{D}$	$\bar{A}+\bar{B}+C+\bar{D}$	$\bar{A}+B+C+\bar{D}$
11	$A+B+\bar{C}+\bar{D}$	$A+\bar{B}+\bar{C}+\bar{D}$	$\bar{A}+\bar{B}+\bar{C}+\bar{D}$	$\bar{A}+B+\bar{C}+\bar{D}$
10	$A+B+\bar{C}+D$	$A+\bar{B}+\bar{C}+D$	$\bar{A}+\bar{B}+\bar{C}+D$	$\bar{A}+B+\bar{C}+D$

(f)

Fig. 5.6 *Maxterm/Minterm Corresponding to Each Cell of K-maps*

5.3.1 Representation of Truth Table on K-Map

Consider the truth table of the 3-variable logic function given in Table 5.3. The output Y is logic 1 corresponding to the rows 1, 2, 4, and 7. Corresponding to this we can write the equation in terms of canonical SOP as given below:

$$Y = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \tag{5.14}$$

Equation (5.14) represents the complete truth table in canonical SOP form. Similarly, we note that the output Y is logic 0 corresponding to the rows 0, 3, 5, and 6 and the output Y can be represented in terms of canonical POS form as given below:

$$Y = (A + B + C)(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C) \tag{5.15}$$

Equation (5.15) also represents the complete truth table, and Eqs (5.14) and (5.15) are equivalent. We shall make use of the 3-variable K-map of Fig. 5.5*b* and enter the value of the output variable Y (0 or 1) in each cell corresponding to its decimal or minterm or maxterm identification. Figure 5.7 gives the complete K-map of the truth table given in Table 5.3.

$C \backslash AB$	00	01	11	10
0	0 0	2 1	6 0	4 1
1	1 1	3 0	7 1	5 0

Fig. 5.7 *K-map for Table 5.3*

The procedure used above is general and is used to represent a truth table on the K-map. On the other hand, if a K-map is given we can make the truth table corresponding to this by following the reverse process. That is, the output Y is logic 1 corresponding to the decimal numbers/minterms represented by cells with entries 1. In all other rows, the output Y is logic 0.

Table 5.3 *Truth Table of a 3-variable Function*

Row No.	Inputs A	B	C	Output Y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

CD \ AB	00	01	11	10
00	0 1	4	12 1	8
01	1	5 1	13	9 1
11	3	7 1	15 1	11
10	2	6	14 1	10

Fig. 5.8 *K-map of Ex. 5.4*

Table 5.4 *Truth Table for K-map of Fig. 5.8*

Row No.	Inputs A	B	C	D	Output Y
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

5.3.2 Representation of Canonical SOP Form on K-Map

A logical equation in canonical SOP form can be represented on a K-map by simply entering 1's in the cells of the K-map corresponding to each minterm present in the equation.

Example 5.5

Represent Eq. (5.14) on K-map.

Solution

Corresponding to each minterm in the logic equation, there is a cell in the K-map and a 1 is entered in each one of these cells. The K-map will be as shown in Fig. 5.7.

Similarly, from the K-map, we can write the corresponding logic equation in canonical SOP form by ORing the minterms corresponding to each 1 entry in the K-map.

Example 5.6

Write the logic equation in the canonical SOP form for the K-map of Fig. 5.8.

Solution

$$Y = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}B\overline{C}\overline{D} + \overline{A}BC\overline{D} + A\overline{B}\overline{C}\overline{D} + A\overline{B}CD + ABCD$$
$$= \Sigma m(0, 5, 7, 9, 12, 14, 15) \qquad (5.16)$$

If the equation is in SOP form, it can be converted to canonical SOP form by using the method discussed earlier and then it can be represented on K-map. Another method of representing SOP form on K-map without converting it to canonical SOP form will become clear from the discussions of Sections 5.4 and 5.6.

5.3.3 Representation of Canonical POS Form on K-Map

Logic equation in canonical POS form can be represented on K-map by entering 0's in the cells of K-map corresponding to each maxterm present in the equation.

Example 5.7

Represent Eq. (5.15) on K-map.

Solution

Corresponding to each maxterm in the equation, there is a cell in the K-map and a 0 is entered in each one of these cells. The K-map will be as shown in Fig. 5.7.

From a given K-map, we can write the logic equation in the canonical POS form by ANDing the maxterms corresponding to each 0 entry in the K-map.

Example 5.8

Write the logic equation in the canonical POS form for the K-map of Fig. 5.8.

Solution

$$Y = (A+B+C+D)(A+B+\bar{C}+D)(A+B+C+\bar{D})$$
$$(A+\bar{B}+C+D)(A+\bar{B}+\bar{C}+D)(A+\bar{B}+C+\bar{D})$$
$$(\bar{A}+B+\bar{C}+D)(\bar{A}+B+C+D)(\bar{A}+\bar{B}+C+\bar{D})$$
$$= \Pi M(1, 2, 3, 4, 6, 8, 10, 11, 13) \quad (5.17)$$

If the equation is in POS form, it can be converted into canonical POS form by the method discussed earlier and then it can be represented on K-map. Another method of representing POS form on K-map without converting it to canonical POS form will become clear from the discussion of Sections 5.4 and 5.6.

Equation (5.16) represents the K-map of Fig. 5.8 in canonical SOP form and Eq. (5.17) represents the same K-map in standard POS form. Therefore, these two equations are equivalent. Alternatively stated, an equation in SOP form can be converted into an equivalent POS form and vice-versa.

5.4 SIMPLIFICATION OF LOGIC FUNCTIONS USING K-MAP

Simplification of logic functions with K-map is based on the principle of combining terms in adjacent cells. Two cells are said to be adjacent if they differ in only one variable. For example, in the two-variable K-maps of Figs 5.6*a* and *b*, the top two cells are adjacent and the bottom two cells are adjacent. Also, the left two cells and the right two cells are adjacent. It can be verified that in adjacent cells one of the literals is same, whereas the other literal appears in uncomplemented form in one and in the complemented form in the other cell.

Similarly, we observe adjacent cells in the 3-variable and 4-variable K-maps. Table 5.5 gives the adjacent cells of each cell in 2-, 3-, and 4-variable K-maps. From this it becomes clear that if the Gray code is used for the identification of cells in K-map, physically adjacent (horizontal and vertical but not diagonal) cells differ in only one variable. Also, the left-most cells are adjacent to their corresponding right-most cells and similarly the top cells are adjacent to their corresponding bottom cells. The simplification of logical function is achieved by grouping adjacent 1's or 0's in groups of 2^i, where $i = 1, 2, \ldots n$ and n is the number of variables.

The process of simplification involves grouping of minterms and identifying *prime-implicants* (PI) and *essential prime-implicants* (EPI).

A *prime-implicant* is a group of minterms that cannot be combined with any other minterm or groups. An *essential prime-implicant* is a *prime-implicant* in which one or more minterms are unique; i.e. it contains at least one minterm which is not contained in any other prime-implicant.

5.4.1 Grouping Two Adjacent Ones

If there are two adjacent ones on the map, these can be grouped together and the resulting term will have one less literal than the original two terms. It can be verified for each of the groupings of two ones as given in Table 5.5.

Table 5.5 *Adjacent Cells in K-maps*

Cell with decimal number	Decimal numbers of adjacent cells		
	2-variable	3-variable	4-variable
0	1, 2	1, 2, 4	1, 2, 4, 8
1	[illegible]	0, 3, 5	[illegible]
2	0, 3	0, 3, 6	0, 3, 6, 10
3	1, 2	1, 2, 7	1, 2, 7, 11
4		0, 5, 6	0, 5, 6, 12
5		1, 4, 7	1, 4, 7, 13
6		2, 4, 7	2, 4, 7, 14
7		3, 5, 6	3, 5, 6, 15
8			0, 9, 10, 12
9			1, 8, 11, 13
10			2, 8, 11, 14
11			3, 9, 10, 15
12			4, 8, 13, 14
13			5, 9, 12, 15
14			6, 10, 12, 15
15			7, 11, 13, 14

Example 5.9

Simplify the K-map of Fig. 5.9.

Solution

The canonical SOP form of equation can be written by inspection as

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}BC + ABC + A\bar{B}\bar{C} \tag{5.18}$$

If we combine the ones in adjacent cells (0, 4) and (3, 7), Eq. (5.18) can be written as

$$Y = (\bar{A} + A)\bar{B}\bar{C} + (\bar{A} + A)BC \tag{5.19}$$

$$= \bar{B}\bar{C} + BC \text{ (Theorems 1.7 and 1.2)} \tag{5.20}$$

Equation (5.20) can be directly obtained from the K-map by using the following procedure:

1. Identify adjacent ones, then see the values of the variables associated with these cells. Only one variable will be different and it gets eliminated. Other variables will appear in ANDed form in the term; it will be in the uncomplemented form if it is 1 and in the complemented form if it is 0.
2. Determine the term corresponding to each group of adjacent ones. These terms are ORed to get the simplified equation in SOP form.

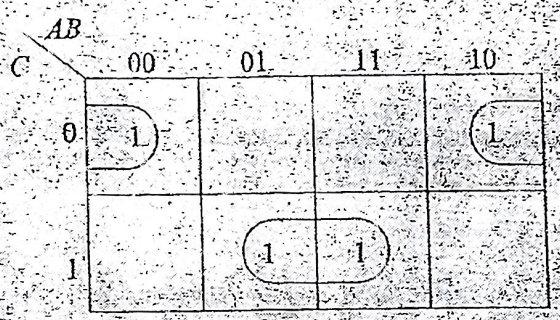


Fig. 5.9 *K-map of Ex. 5.9*

Here, $\overline{B}\overline{C}$ and BC are the two prime-implicants, since grouping of the two minterms m_0 and m_4 cannot be combined with anyone of the remaining minterms m_3 and m_7, or the combination of the minterms m_3 and m_7. Similarly, the grouping of m_3 and m_7 is a prime-implicant. We also observe that both the groups are also essential prime-implicants, since each one of them contains unique minterms which are not contained in the other group.

5.4.2 Grouping Four Adjacent Ones

Four cells form a group of four adjacent ones if two of the literals associated with the minterms/maxterms are not same and the other literals are same. Table 5.6 gives all possible groups of four adjacent ones for each cell in a 3-variable map. In case of 2-variable map, there is only one possibility corresponding to entry 1 in all the four cells, and the simplified expression will be $Y = 1$. That is, Y always equals 1 (independent of the variables).

On the basis of groupings of 4 adjacent ones given in Table 5.6, we can find the groupings in K-maps of four or more variables. In the case of a four-variable K-map, there are six possible groupings of 4-variables involving any cell. It is left to the reader to verify this fact.

Table 5.6 *Groups of Four Adjacent Ones in a 3-variable K-map*

Cell with decimal number	Decimal numbers of cells forming groups of adjacent fours		
0	(0, 2, 6, 4),	(0, 1, 2, 3),	(0, 1, 4, 5)
1	(1, 0, 2, 3),	(1, 3, 7, 5),	(1, 0, 4, 5)
2	(2, 0, 6, 4),	(2, 3, 1, 0),	(2, 3, 6, 7)
3	(3, 1, 7, 5),	(3, 2, 1, 0),	(3, 2, 6, 7)
4	(4, 6, 2, 0),	(4, 5, 6, 7),	(4, 5, 0, 1)
5	(5, 1, 3, 7),	(5, 4, 6, 7),	(5, 4, 0, 1)
6	(6, 0, 2, 4),	(6, 7, 4, 5),	(6, 7, 2, 3)
7	(7, 1, 3, 5),	(7, 6, 4, 5),	(7, 6, 2, 3)

Example 5.10

Simplify the K-map of Fig. 5.10.

CD \ AB	00	01	11	10
00	0 1	4	12	8 1
01	1 1	5	13	9 1
11	3 1	7 1	15 1	11 1
10	2	6	14	10

Fig. 5.10 *K-map of Ex. 5.10*

Solution

The canonical SOP form of equation can be written by inspection as

$$Y = m_0 + m_1 + m_3 + m_7 + m_8 + m_9 + m_{11} + m_{15}$$
$$= (m_0 + m_1 + m_8 + m_9) + (m_3 + m_7 + m_{15} + m_{11}) \quad (5.21)$$

In the K-map of Fig. 5.10, there are two groups of four adjacent ones. One corresponding to cells 0, 1, 8, and 9, and the other one corresponding to 3, 7, 15, and 11. In Eq (5.21), the minterms corresponding to each group are

combined. The first term can be written as

$$m_0 + m_1 + m_8 + m_9 = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D + A\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}D$$
$$= \overline{B}\overline{C}(\overline{A}\overline{D} + \overline{A}D + A\overline{D} + AD)$$
$$= \overline{B}\overline{C}[\overline{A}(\overline{D} + D) + A(\overline{D} + D)]$$
$$= \overline{B}\overline{C}[\overline{A}\cdot 1 + A\cdot 1]$$
$$= \overline{B}\overline{C}(\overline{A} + A)$$
$$= \overline{B}\overline{C}\cdot 1 = \overline{B}\overline{C}$$

In the first term of Eq. (5.21) we observe the following:

1. In this group of four minterms, two of the variables appear as $\overline{B}$ and $\overline{C}$ in all the four terms.
2. The variable A appears as A in two and as $\overline{A}$ in the other two minterms.
3. The variable D appears as D in two and as $\overline{D}$ in the other two minterms.
4. The combination of these four minterms results in one term with two literals which are present in all the four terms. Similarly, the second term of Eq. (5.21) is simplified to CD. Therefore, the K-map is simplified to

$$Y = \overline{B}\overline{C} + CD \tag{5.22}$$

Example 5.11

In Fig. 5.10, show that the following groups of minterms are not prime-implicants:

(a) m_0, m_1
(b) m_1, m_3
(c) m_3, m_7
(d) m_7, m_{15}
(e) m_{11}, m_{15}
(f) m_3, m_7
(g) m_0, m_{11}
(h) m_0, m_8
(i) m_1, m_9
(j) m_3, m_{11}

Solution

(a) m_0, m_1 group can be combined with m_8, m_9 group to form a group of four adjacent 1s. Hence, m_0, m_1 group is not a prime-implicant.
(b) m_1, m_3 group can be combined with m_9, m_{11} group, therefore, it is not a prime-implicant.
(c) m_3, m_7 group can be combined with m_{15}, m_{11} group.
(d) m_7, m_{15} group can be combined with m_3, m_{11} group.
(e) m_{11}, m_{15} group can be combined with m_3, m_7 group.
(f) See (a)
(g) See (b)
(h) m_0, m_8 group can be combined with m_1, m_9 group.
(i) See (h)
(j) See (d)

Example 5.12

Show that the following groups of minterms in Fig. 5.10 are essential prime-implicants:

(a) $m(0, 1, 8, 9)$
(b) $m(3, 7, 11, 15)$

Solution

(a) The group formed by the minterms m_0, m_4, m_8, m_6 is a prime-implicant, since it can not be combined with any other minterm or group. Also, it includes minterms which can not be included in the other group, therefore, it is an essential prime-implicant.

(b) The group formed by the minterms m_0, m_2, m_8, m_{10} have m_0 and m_{10} which can not be included in any other prime-implicant hence this is an essential prime-implicant.

5.4.3 Grouping Eight Adjacent Ones

Eight cells form a group of eight adjacent ones if three of the literals associated with the minterms/maxterms are not same and the other literals are same. In case of 3-variable K-map, there is only one possibility of eight ones appearing in the K-map and this corresponds to output equal to 1, irrespective of the values of the input variables. Table 5.7 gives all possible groups of eight adjacent ones in a 4-variable K-map. From an understanding of this, we can easily find out such combinations for 5- and 6-variable K-maps. When eight adjacent ones are combined, the resulting equation will have only one term with the number of literals three less than the number of literals in the original minterms. Similar to the groupings of adjacent two and four ones, the literals which are common in all the eight minterms will be present and the literals which are not same get eliminated in the resulting term.

Table 5.7 *Groups of Eight Adjacent Ones in 4-variable K-map*

Decimal numbers of cells forming groups of adjacent eights in a 4-variable K-map
0, 4, 12, 8, 1, 5, 13, 9,
0, 4, 12, 8, 2, 6, 14, 10
0, 1, 3, 2, 4, 5, 7, 6
0, 1, 3, 2, 8, 9, 11, 10
1, 5, 13, 9, 3, 7, 15, 11
4, 5, 7, 6, 12, 13, 15, 14
12, 13, 15, 14, 8, 9, 11, 10
3, 7, 15, 11, 2, 6, 14, 10

The reader is advised to verify the simplification of eight adjacent ones into a single term with three variables eliminated. For example, let us take the first group of eight adjacent ones in Table 5.7. For all these eight cells, the variable C appears as $\overline{C}$ in the minterms and the other three variables are not same. Therefore, the grouping of these eight cells results in a term $\overline{C}$. Figure 5.11 shows the simplified expression for each of the groupings of eight ones for a 4 variable K-map.

5.4.4 Grouping 2, 4, and 8 Adjacent Zeros

In the above discussion, we have considered groups of 2, 4, and 8 adjacent ones. Instead of making the groups of ones, we can also make groups of zeros. The procedure is similar to the one used above and is as follows:

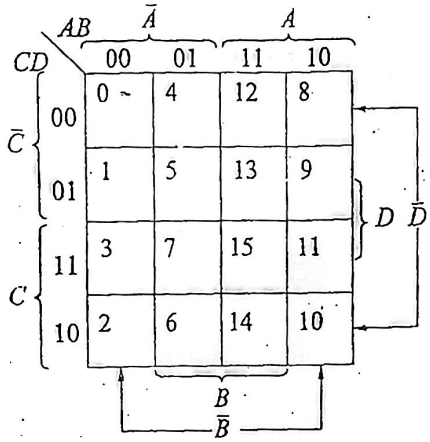


Fig. 5.11. *Four-variable K-map Illustrating the Groupings of Eight Adjacent Ones*

1. Group of two adjacent zeros result in a term with literal less than the number of variables. The literal w is not same in the two maxterms gets eliminated.
2. Group of four adjacent zeros result in a term two literals less than the number of variables. The literals which are not same in all the four maxterms eliminated.
3. Group of eight adjacent zeros result in a term three literals less than the number of variables. The tl literals which are not same in all the eight maxterms eliminated.

We have considered groups of 2, 4, and 8 adjacent c and zeros. The same logic can be extended to 16, 32, 64 adjacent ones and zeros which occur in K-maps more than 4 variables.

5.5 MINIMISATION OF LOGIC FUNCTIONS SPECIFIED IN MINTERMS/MAXTERMS OR TRUTH TABLE

5.5.1 Minimisation of SOP Form

We have seen the advantages of simplifying a logical expression. If the expression is simplified to a st beyond which it can not be further simplified, it will require minimum number of gates with minim number of inputs to the gates. Such an expression is referred to as the *minimised expression*.

For minimising a given expression in SOP form or for a given truth table, we have to prepare the K-n first and then look for combinations of ones on the K-map. We have to combine the ones in such a w that the resulting expression is minimum. To achieve this, the following algorithm can be used which w definitely lead to minimised expression:

1. Identify the ones which can not be combined with any other ones and encircle them. These are *essen prime-implicants*.
2. Identify the ones that can be combined in groups of two in only one way. Encircle such groups of ones
3. Identify the ones that can be combined with three other ones, to make a group of four adjacent ones, only one way. Encircle such groups of ones.
4. Identify the ones that can be combined with seven other ones, to make a group of eight adjacent ones, only one way. Encircle such groups of ones.
5. After identifying the essential groups of 2, 4, and 8 ones, if there still remains some ones which ha not been encircled then these are to be combined with each other or with other already encircled on Of course, however, we should combine the left-over ones in largest possible groups and in as f groupings as possible. In this, the groupings may not be unique and we should make the groupings in optimum manner. You can verify that any one can be included any number of times without affecting expression.

The logic function consisting of the essential prime-implicants obtained in steps 1 to 4 and the pri implicants obtained in step 5 will be the minimised function. The above algorithm will be used to minim the logic functions in the examples given.

Example 5.13

Minimise the four-variable logic function using K-map.

Solution

$$f(A, B, C, D) = \Sigma m(0, 1, 2, 3, 5, 7, 8, 9, 11, 14) \tag{5.23}$$

The K-map of Eq. (5.23) is shown in Fig. 5.12. The equation is minimised in the following steps:

1. Encircle 1 in cell 14 which can not be combined with any other 1. The term corresponding to this is $ABCD$.
2. There are at least two possible ways for every 1 forming groups of two adjacent ones. Therefore, we ignore it for the time being and go to the next step.
3. There is only one possible group of four adjacent ones involving each of the cells 8, 11, 5 or 7 and 2, and these are (8, 9, 0, 1), (11, 9, 1, 3), (5, 7, 3, 1) and (2, 3, 1, 0), respectively. Encircle these groups. The terms corresponding to these groups are $B\bar{C}$, $\bar{B}\bar{D}$, $\bar{A}\bar{D}$, and $\bar{A}\bar{B}$ respectively.

Since all the ones have been encircled, therefore, the minimised equation is

$$f(A, B, C, D) = ABCD + B\bar{C} + \bar{B}\bar{D} + \bar{A}\bar{D} + \bar{A}\bar{B} \tag{5.24}$$

Fig. 5.12 K-map for Eq. (5.23)

Example 5.14

Determine the minimised expression in SOP form for the truth table given in Table 5.8.

Solution

The K-map for the truth table of Table 5.8 is shown in Fig. 5.13. Using the minimisation steps, we obtain the minimised expression

$$Y = \bar{B} + A\bar{C} + \bar{A}CD \tag{5.25}$$

Table 5.8

Inputs				Output
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

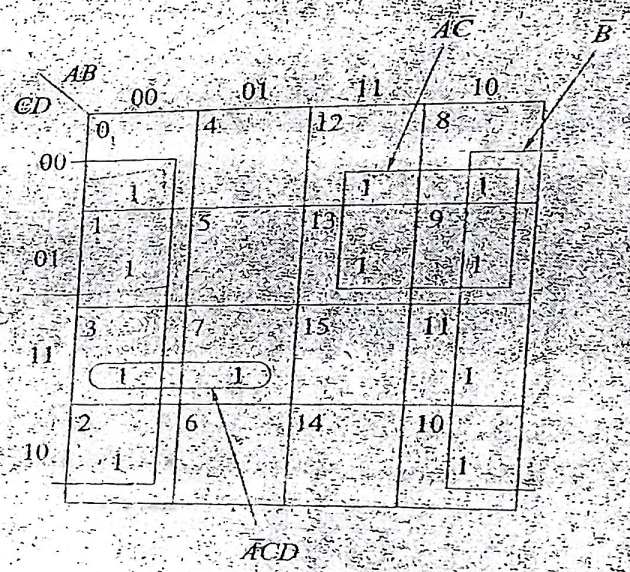


Fig. 5.13 *K-map of Table 5.8*

The NAND–NAND realisations are given in Fig. 5.21.

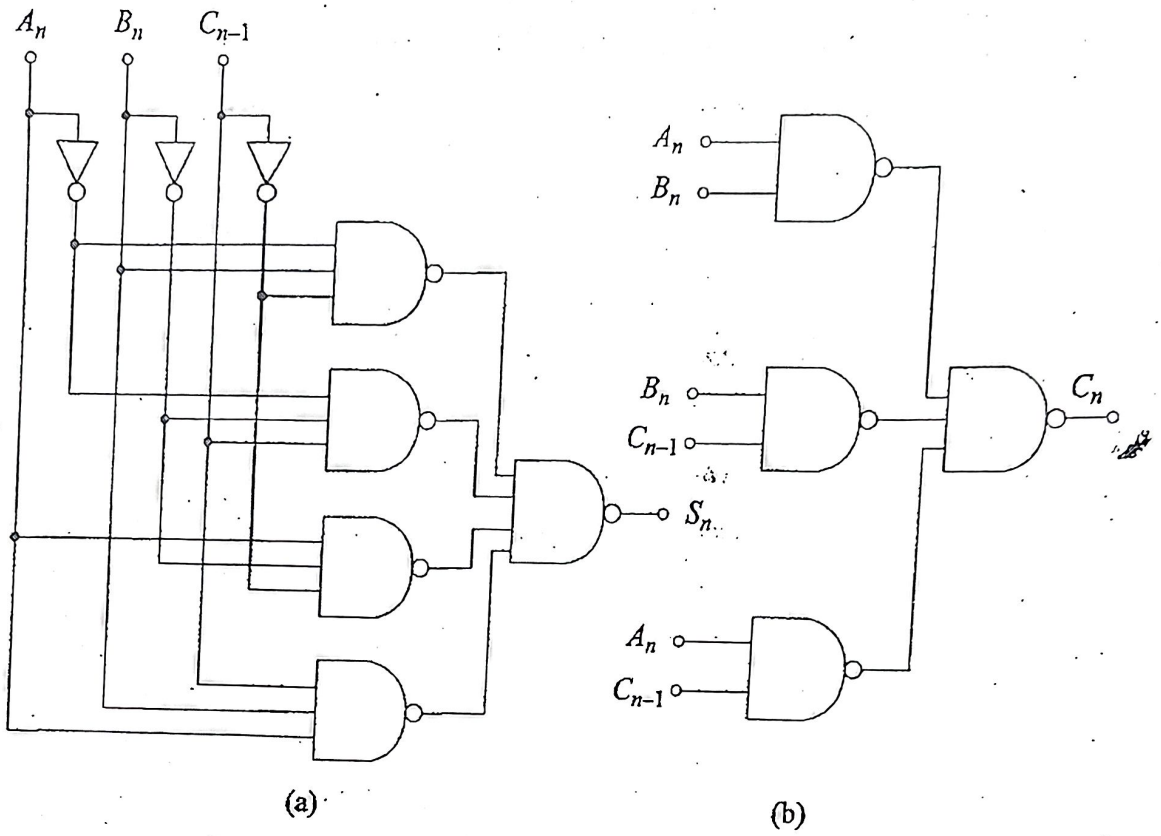


Fig. 5.21 NAND–NAND *Realisation of (a) S_n (b) C_n*

3. Half-subtractor A logic circuit for the subtraction of B (subtrahend) from A (minuend) where A and B are 1-bit numbers is referred to as a *half-subtractor*. The subtraction process is illustrated in Section 2.5 and is reproduced in truth table form in Table 5.13. Here, A and B are the two inputs and D (difference) and C (borrow) are the two outputs.

Table 5.13 *Truth Table of a Half-subtractor*

Inputs		Outputs	
A	B	D	C
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

From the truth table, the logical expressions for D and C are obtained as

$$D = \bar{A}B + A\bar{B} = A \oplus B \tag{5.36a}$$

$$C = \bar{A}B \tag{5.36b}$$

The realisation of a half-subtractor using gates is shown in Fig. 5.22.

4. Full-subtractor Just like a full-adder, we require a *full-subtractor* circuit for performing multibit subtraction wherein a borrow from the previous bit position may also be there. A full-subtractor will have three inputs, A_n (minuend), B_n (subtrahend) and C_{n-1} (borrow from the previous stage) and two outputs, D_n (difference) and C_n (borrow). Its truth table is given in Table 5.14. The K-map for the output D_n is exactly same as the K-map for S_n of the adder circuit and therefore, its realisation is same as given in Fig. 5.21*a*.

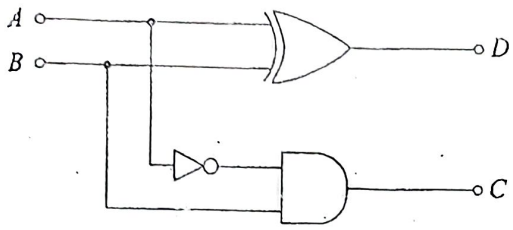


Fig. 5.22 *Realisation of a Half-subtractor*

Table 5.14 *Truth Table of a Full-subtractor*

Inputs			Outputs	
A_n	B_n	C_{n-1}	D_n	C_n
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The K-map for C_n is given in Fig. 5.23*a* and its realisation is given in Fig. 5.23*b*. The simplified expression for C_n is

$$C_n = \bar{A}_n B_n + \bar{A}_n C_{n-1} + B_n C_{n-1} \tag{5.37}$$

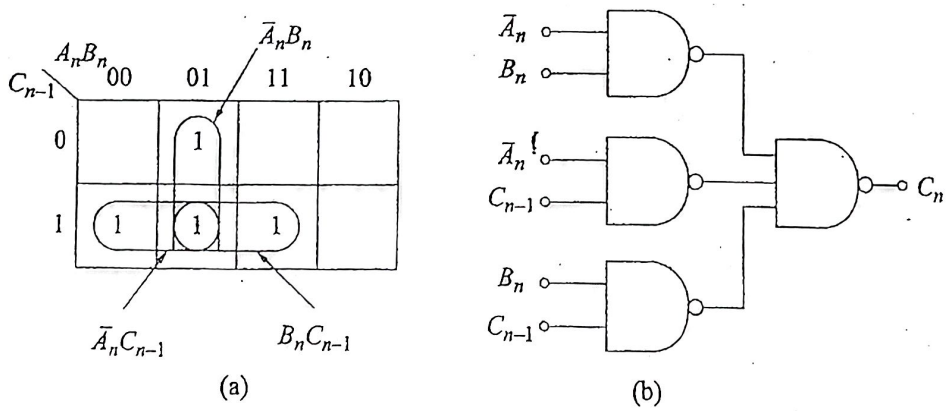


Fig. 5.23 *(a) K-map for C_n (b) Realisation of C_n*

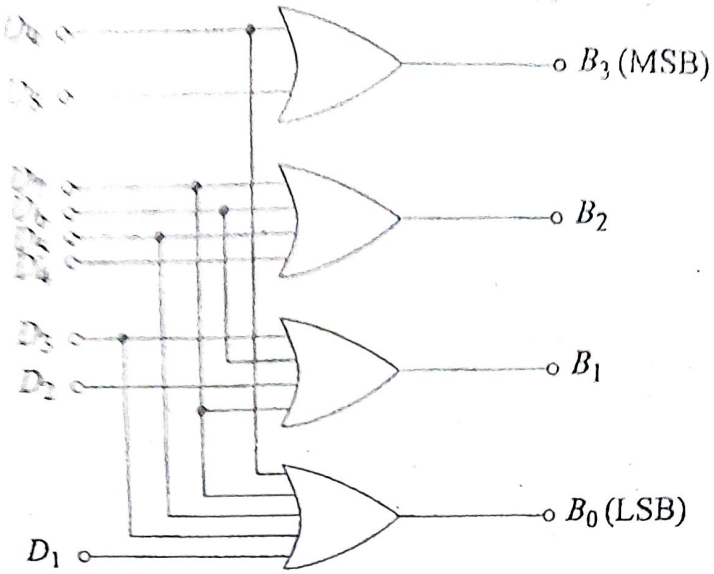


Fig. 5.28 *Implementation of Decimal-to-BCD Encoder*

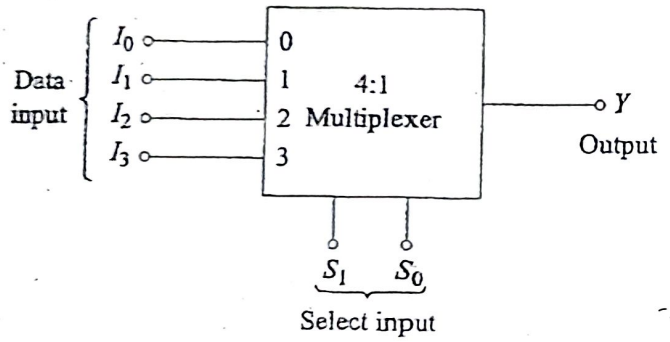


Fig. 5.29 *Block Diagram of a 4:1 Multiplexer*

5.8.4 Multiplexer

The multiplexer (or data selector) is a logic circuit that allows one of the n data inputs at the output. Figure 5.29 shows a 4:1 multiplexer. It has four data input lines (I_0–I_3), two select lines (S_1, S_0) and Y is the data output line. One of the four data inputs will appear at the output depending on the value of S_1 S_0. Its truth table is given in Table 5.17.

From Table 5.17 , the logic expression for the output is

Table 5.17 *Truth Table of a 4:1 Multiplexer*

Select input		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$Y = \bar{S}_0 \cdot \bar{S}_1 \cdot I_0 + S_0 \cdot \bar{S}_1 \cdot I_1 + \bar{S}_0 \cdot S_1 \cdot I_2 + S_0 \cdot S_1 \cdot I_3 \quad (5.49)$$

Its realisation using NAND gates and inverters is shown in Fig.5.30.

The circuit of Fig. 6.5 uses two 16:1 multiplexers (M_1 and M_2) for the realisation of a 32:1 multiplexer. The lower order 16 data input lines (I_0–I_{15}) are applied at the data input terminals of the multiplexer M_1 and the higher order 16 data input lines (I_{16}–I_{31}) are applied at the data input terminals of the multiplexer M_2. For a 32:1 multiplexer, the number of select input lines is required to be 5. The 5-bit select inputs are $ABCDE$. The most-significant select input bit A is applied at G_1 and $\bar{A}$ is applied at G_2. B, C, D, and E are connected to S_3, S_2, S_1, and S_0 inputs of both the multiplexers. The output F of the multiplexer is obtained by using an OR gate, where $F = Y_1 + Y_2$.

When $A = 0$, the multiplexer M_1 is enabled and M_2 is disabled, therby allowing one of the lower order 16 bits to Y_1, depending upon the value of $BCDE$. $Y_2 = 0$. Therefore, $F = Y_1$. Similarly, when $A = 1$, M_2 is enabled and M_1 is disabled and $F = Y_2$.

The circuit of Fig. 6.6 uses a 2:1 multiplexer instead of an OR gate. When $A = 0$, $F = Y_1$ and when $A = 1$, $F = Y_2$. Thus, both the circuits perform similar operation.

These two general techniques can be used to expand to an n input multiplexer without any difficulty.

6.3 DEMULTIPLEXERS/DECODERS AND THEIR USE IN COMBINATIONAL LOGIC DESIGN

6.3.1 Demultiplexer

The demultiplexer performs the reverse operation of a multiplexer. It accepts a single input and distributes it over several outputs. Figure 6.7 gives the block diagram of a demultiplexer. The select input code determines to which output the data input will be transmitted.

The number of output lines is n and the number of select lines is m, where $n = 2^m$. The data input D_i will appear on the output line selected by the select input. For example, if the decimal equivalent of the select input is 4, then the data will appear on D_4 output line. This circuit can also be used as binary-to-decimal decoder with binary inputs applied at the select input lines and the output will be obtained on the corresponding line. The data input line is to be connected to logic 1 level. This circuit can be designed using gates and it is left as an exercise for the reader. However, this device is available as an MSI IC and can conveniently be used for the design of combinational circuits. The device is very useful if multiple-output combinational circuit is to be designed, because this needs minimum package count. These devices are available (Table 6.5) as

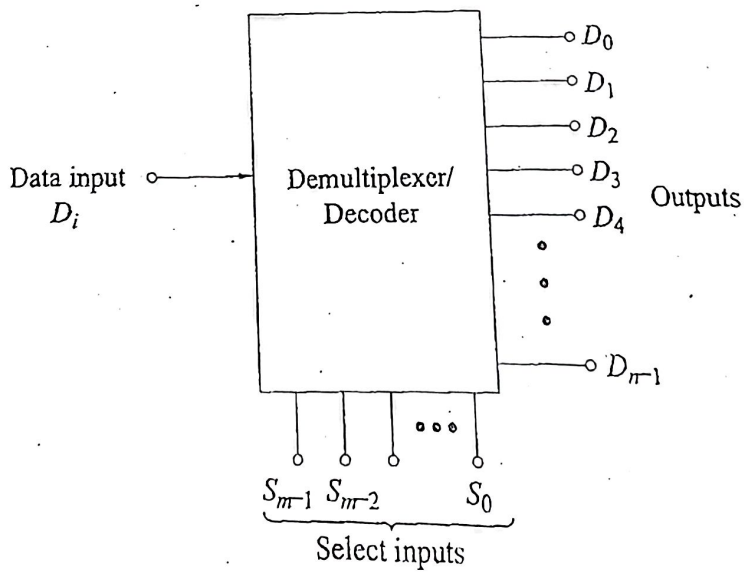


Fig. 6.7 *Block Diagram of a Demultiplexer.*

CHAPTER 7

FLIP-FLOPs

7.1 INTRODUCTION

So far we have directed our studies towards the analysis and design of combinational digital circuits. Though very important, it constitutes only a part of digital systems. The other major aspect of digital systems is analysis and design of sequential circuits. However, sequential circuit design depends, to a large extent, on the combinational circuit design discussed earlier.

There are many applications in which digital outputs are required to be generated in accordance with the sequence in which the input signals are received. This requirement can not be satisfied using a combinational logic system. These applications require outputs to be generated that are not only dependent on the present input conditions but they also depend upon the past history of these inputs. The past history is provided by feedback from the output back to the input.

A block diagram of a sequential circuit is shown in Fig. 7.1. It consists of combinational circuits which accept digital signals from external inputs and from outputs of memory elements and generates signals for external outputs and for inputs to memory elements referred to as excitation.

A memory element is some medium in which one bit of information (1 or 0) can be stored or retained until necessary, and thereafter its contents can be replaced by a new value. The contents of memory elements in Fig. 7.1 can be changed by the outputs of the combinational circuit which are connected to its input.

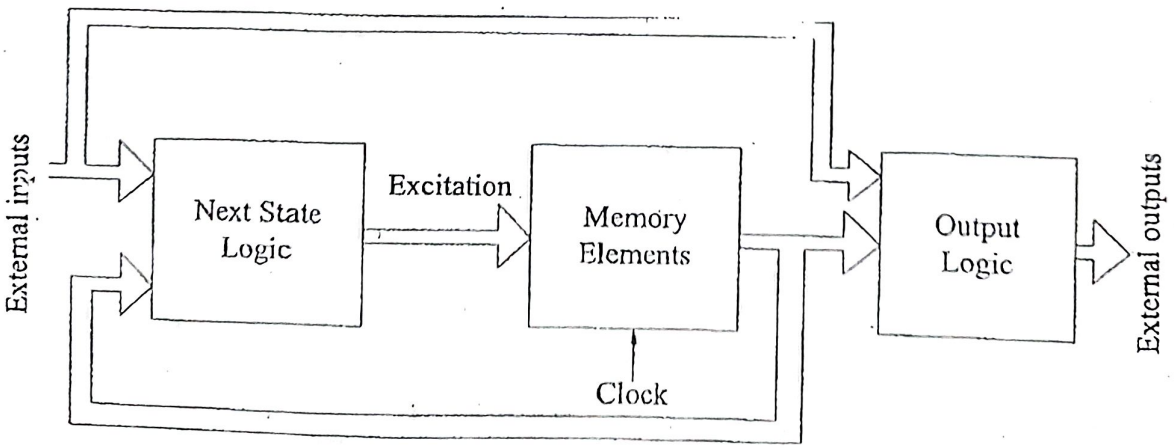


Fig. 7.1 *Block Diagram of a Sequential Circuit*

The combinational circuit performs certain operations, some of which are used to determine the digital signals to be stored in memory elements. The other operations are performed on external inputs and memory outputs to generate the external outputs.

The above process demonstrates the dependence of the external outputs of a sequential circuit on the external inputs and the present contents of the memory elements (referred to as the *present state* of memory elements). The new contents of the memory elements, referred to as the *next state*; depend on the external inputs and the present state. Hence, the output of a sequential circuit is a function of the time sequence of inputs and the *internal states*.

Sequential circuits are classified in two main categories, known as *asynchronous* and *synchronous* sequential circuits depending on timing of their signals.

A sequential circuit whose behaviour depends upon the sequence in which the input signals change is referred to as an asynchronous sequential circuit. The outputs will be affected whenever the inputs change. The commonly used memory elements in these circuits are time delay devices. These can be regarded as combinational circuits with feedback.

A sequential circuit whose behaviour can be defined from the knowledge of its signal at discrete instants of time is referred to as a synchronous sequential circuit. In these systems, the memory elements are affected only at discrete instants of time. The synchronization is achieved by a timing device known as a *system clock* which generates a periodic train of clock pulses as shown in Fig. 7.2. The outputs are affected only with the application of a clock pulse.

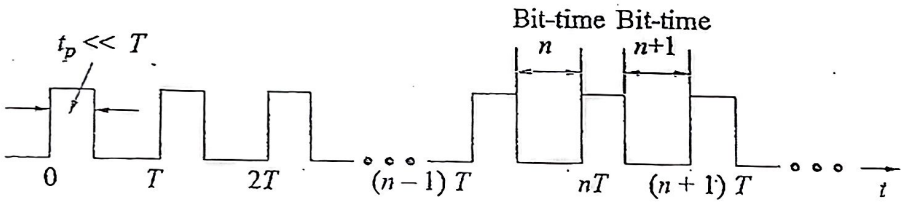


Fig. 7.2 *A Train of Pulses*

Since the design of asynchronous circuits is more-tedious and difficult, therefore their uses are rather limited.

Synchronous circuits have gained considerable domination and wide popularity and are also known as *clocked-sequential circuits*. The memory elements used are FLIP-FLOPs which are capable of storing binary information.

7.2 A 1-BIT MEMORY CELL

The basic digital memory circuit is known as FLIP-FLOP. It has two stable states which are known as the *1 state* and the *0 state*. It can be obtained by using NAND or NOR gates. We shall be systematically developing a FLIP-FLOP circuit starting from the fundamental circuit shown in Fig. 7.3. It consists of two inverters G_1 and G_2 (NAND gates used as inverters). The output of G_1 is connected to the input of G_2 (A_2) and the output of G_2 is connected to the input of G_1 (A_1).

Let us assume the output of G_1 to be $Q = 1$, which is also the input of $G_2(A_2 = 1)$. Therefore, the output of G_2 will be $\overline{Q} = 0$, which makes $A_1 = 0$ and consequently $Q = 1$ which confirms our assumption.

In a similar manner, it can be demonstrated that if $Q = 0$, then $\overline{Q} = 1$ and this is also consistent with the circuit connections.

From the above discussion we note the following:

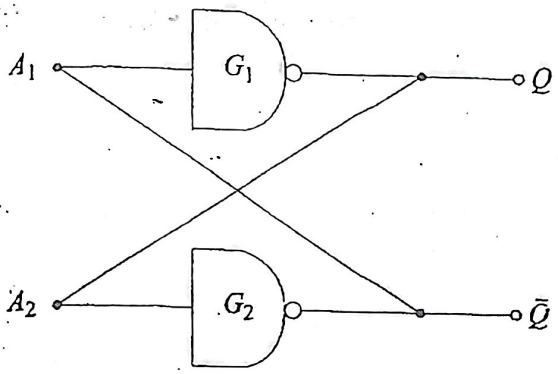


Fig. 7.3 *Cross-coupled Inverters as a Memory Element*

1. The outputs Q and $\bar{Q}$ are always complementary.
2. The circuit has two stable states; in one of the stable state $Q = 1$ which is referred to as the 1 state (*or set state*) whereas in the other stable state $Q = 0$ which is referred to as the 0 state (*or reset state*).
3. If the circuit is in 1 state, it continues to remain in this state and similarly if it is in 0 state, it continues to remain in this state. This property of the circuit is referred to as *memory*, i.e. it can store 1-bit of digital information.

Since this information is locked or latched in this circuit, therefore, this circuit is also referred to as a *latch*.

In the latch of Fig. 7.3, there is no way of entering the desired digital information to be stored in it. In fact, when the power is switched on, the circuit switches to one of the stable states ($Q = 1$ or 0) and it is not possible to predict the state. If we replace the inverters G_1 and G_2 with 2-input NAND gates, the other input terminals of the NAND gates can be used to enter the desired digital information. The modified circuit is shown in Fig. 7.4. Two additional inverters G_3 and G_4 have been added for reasons which will become clear from the following discussion.

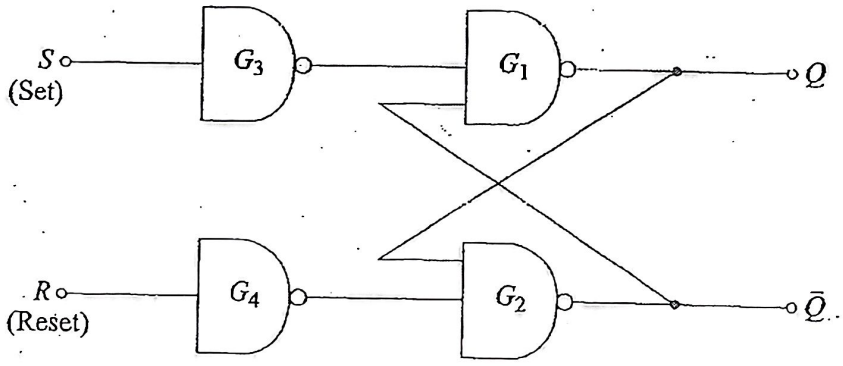


Fig. 7.4 *The Memory Cell with Provision for Entering Data*

If $S = R = 0$, the circuit is exactly the same as that of Fig. 7.3 (Prob. 7.1). If $S = 1$ and $R = 0$, the output of G_3 will be 0 and the output of G_4 will be 1. Since one of the inputs of G_1 is 0, its output will certainly be 1. Consequently, both the inputs of G_2 will be 1 giving an output $\bar{Q} = 0$. Hence, for this input condition, $Q = 1$ and $\bar{Q} = 0$. Similarly, if $S = 0$ and $R = 1$ then the outputs will be $Q = 0$ and $\bar{Q} = 1$. The first of these two input conditions ($S = 1, R = 0$) makes $Q = 1$ which is referred to as the *set state*, whereas the second input condition ($S = 0, R = 1$) makes $Q = 0$ which is referred to as the *reset state* or *clear state*. This gives us the means for entering the desired bit in the latch.

Now we see what happens if the input conditions are changed from $S = 1, R = 0$ to $S = R = 0$ or from $S = 0, R = 1$ to $S = R = 0$. The output remains unaltered (Prob. 7.2). This shows the basic difference between a combinational circuit and a sequential circuit, even though the sequential circuit is made up of combinational circuits.

The two input terminals are designated as set (S) and reset (R) because $S = 1$ brings the circuit in set state and $R = 1$ brings it to reset or clear state.

If $S = R = 1$, both the outputs Q and $\bar{Q}$ will try to become 1 which is not allowed and therefore, this input condition is prohibited.

7.3 CLOCKED *S–R* FLIP-FLOP

It is often required to set or reset the memory cell (Fig. 7.4) in synchronism with a train of pulses (Fig. 7.2) known as clock (abbreviated as *CK*). Such a circuit is shown in Fig. 7.5, and is referred to as a *clocked set-reset* (*S–R*) FLIP-FLOP.

In this circuit, if a clock pulse is present ($CK = 1$), its operation is exactly the same as that of Fig. 7.4. On the other hand, when the clock pulse is not present ($CK = 0$), the gates G_3 and G_4 are inhibited, i.e. their outputs are 1 irrespective of the values of S or R. In other words, the circuit responds to the inputs S and R only when the clock is present.

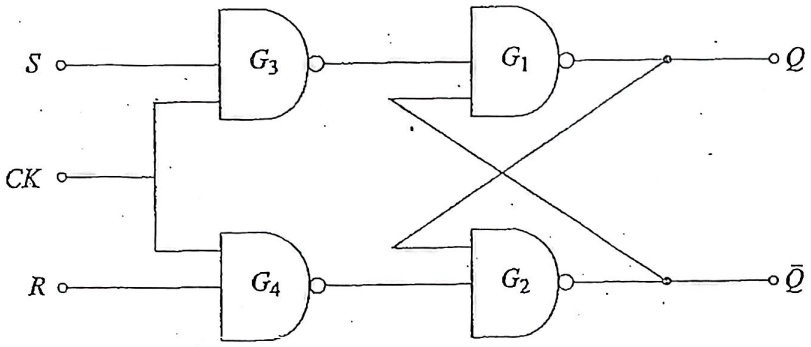


Fig. 7.5 *A Clocked S–R* FLIP-FLOP

Assuming that the inputs do not change during the presence of the clock pulse, we can express the operation of a FLIP-FLOP in the form of the truth table in Table 7.1 for the *S–R* FLIP-FLOP. Here S_n and R_n denote the inputs and Q_n the output during the bit time n (Fig. 7.2). Q_{n+1} denotes the output Q after the pulse passes, i.e. in the bit time $n + 1$.

Table 7.1 *Truth Table of S–R* FLIP-FLOP

Inputs		Output
S_n	R_n	Q_{n+1}
0	0	Q_n
1	0	1
0	1	0
1	1	?

If $S_n = R_n = 0$, and the clock pulse is applied, the output at the end of the clock pulse is same as the output before the clock pulse, i.e. $Q_{n+1} = Q_n$. This is indicated in the first row of the truth table.

If $S_n = 1$ and $R_n = 0$, the output at the end of the clock pulse will be 1, whereas if $S_n = 0$ and $R_n = 1$, then $Q_{n+1} = 0$. These are indicated in the second and third rows of the truth table respectively.

Que; 4 m

Explain J K flip flop

In the circuit of Fig. 7.4, it was mentioned that $S = R = 1$ is not allowed. Let us see what happens in the S–R FLIP–FLOP of Fig. 7.5 if $S_n = R_n = 1$. When the clock is present the outputs of gates G_3 and G_4 are both 0, making one of the inputs of G_1 and G_2 NAND gates 0. Consequently, Q and $\bar{Q}$ both will attain logic 1 which is inconsistent with our assumption of complementary outputs. Now, when the clock pulse has passed away ($CK = 0$), the outputs of G_3 and G_4 will rise from 0 to 1. Depending upon the propagation delays of the gates, either the stable state $Q_{n+1} = 1$ ($\bar{Q}_{n+1} = 0$) or $Q_{n+1} = 0$ ($\bar{Q}_{n+1} = 1$) will result. That means the state of the circuit is undefined, indeterminate or ambiguous and therefore is indicated by a question mark (fourth row of the truth table).

The condition $S_n = R_n = 1$ is forbidden and it must not be allowed to occur.

The logic symbol of clocked S–R FLIP–FLOP is given in Fig. 7.6.

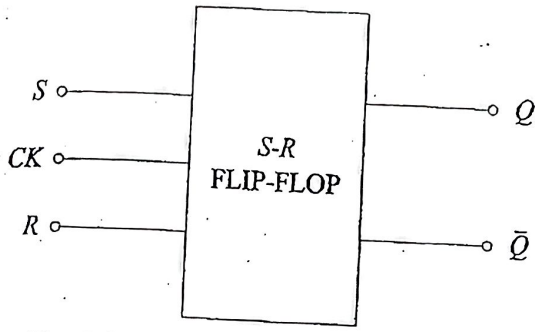


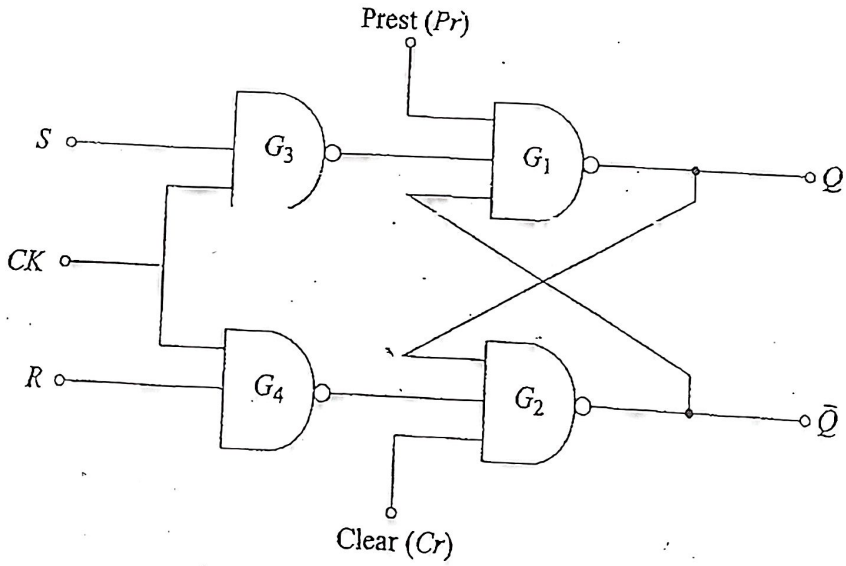
Fig. 7.6 *Logic Symbol of Clocked S–R* FLIP–FLOP

7.3.1 Preset and Clear

In the FLIP–FLOP of Fig. 7.5, when the power is switched on, the state of the circuit is uncertain. It may come to set ($Q = 1$) or reset ($Q = 0$) state. In many applications it is desired to initially set or reset the FLIP–FLOP, i.e. the initial state of the FLIP–FLOP is to be assigned. This is accomplished by using the direct, or *asynchronous inputs*, referred to as *preset* (*Pr*) and *clear* (*Cr*) inputs. These inputs may be applied at any time between clock pulses and are not in synchronism with the clock. An S–R FLIP–FLOP with preset and clear is shown in Fig. 7.7. If $Pr = Cr = 1$ the circuit operates in accordance with the truth table of S–R FLIP–FLOP given in Table 7.1.

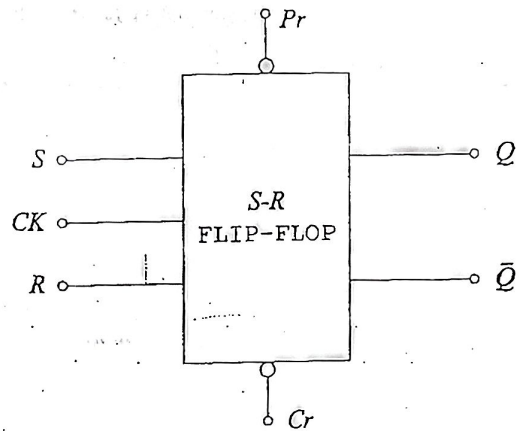
If $Pr = 0$ and $Cr = 1$, the output of G_1(Q) will certainly be 1. Consequently, all the three inputs to G_2 will be 1 which will make $\bar{Q} = 0$. Hence, making $Pr = 0$ sets the FLIP–FLOP.

Similarly, if $Pr = 1$ and $Cr = 0$, the FLIP–FLOP is reset. Once the state of the FLIP–FLOP is established asynchronously, the asynchronous inputs Pr and Cr must be connected to logic 1 before the next clock is applied.



(a)

Fig. 7.7 *(a) An S–R* FLIP–FLOP *with Preset and Clear, (b) Its Logic Symbol*



(b)

Fig. 7.7 *(Continued)*

The condition $Pr = Cr = 0$ must not be used, since this leads to an uncertain state.

In the logic symbol of Fig. 7.7*b*, bubbles are used for *Pr* and *Cr* inputs, which means these are active-low, i.e. the intended function is performed when the signal applied to *Pr* or *Cr* is LOW The operation of Fig. 7.7 is summarised in Table 7.2.

The circuit can be designed such that the asynchronous inputs override the clock, i.e. the circuit can be set or reset even in the presence of the clock pulse (Prob. 7.4).

Table 7.2 ***Summary of Operation of S–R*** FLIP–FLOP

Inputs			Output	Operation performed
CK	*Cr*	*Pr*	*Q*	
1	1	1	Q_{n+1} (Table 7.1)	Normal FLIP–FLOP
0	0	1	0	Clear
0	1	0	1	Preset

7.4 *J–K* FLIP–FLOP

ym que

The uncertainty in the state of an *S–R* FLIP–FLOP when $S_n = R_n = 1$ (fourth row of the truth table) can be eliminated by converting it into a *J–K* FLIP–FLOP. The data inputs are *J* and *K* which are ANDed with $\bar{Q}$ and *Q*, respectively, to obtain *S* and *R* inputs, i.e.

$$S = J \cdot \bar{Q} \tag{7.1a}$$

$$R = K \cdot Q \tag{7.1b}$$

A *J–K* FLIP–FLOP thus obtained is shown in Fig. 7.8. Its truth table is given in Table 7.3*a* which is reduced to Table 7.3*b* for convenience. Table 7.3*a* has been prepared for all the possible combinations of *J*

and K inputs, and for each combination both the states of the output have been considered. The reader can verify this (Prob. 7.5).

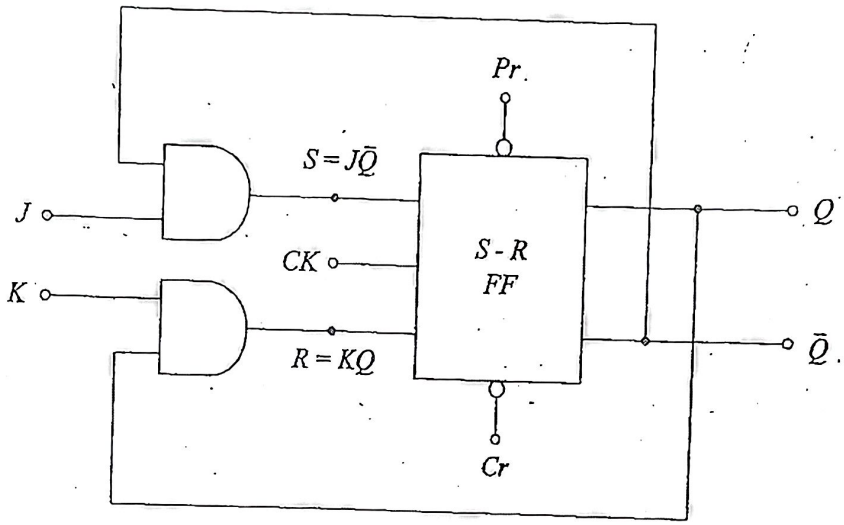


Fig. 7.8 *An S–R* FLIP-FLOP *Converted into J–K* FLIP-FLOP

Table 7.3a *Truth Table for Fig. 7.8*

S = JQ̄
R = KQ

Data inputs		Outputs		Inputs to S–R FF		Output
J_n	K_n	Q_n	$\bar{Q}_n$	S_n	R_n	Q_{n+1}
0	0	0	1	0	0	0
0	0	1	0	0	0	1 $= Q_n$
1	0	0	1	1	0	1
1	0	1	0	0	0	1 $= 1$
0	1	0	1	0	0	0
0	1	1	0	0	1	0 $= 0$
1	1	0	1	1	0	1
1	1	1	0	0	1	0 $= \bar{Q}_n$

Table 7.3b *Truth Table of J–K* FLIP–FLOP

Inputs		Output
J_n	K_n	Q_{n+1}
0	0	Q_n
1	0	1
0	1	0
1	1	$\bar{Q}_n$

It is not necessary to use the AND gates of Fig. 7.8, since the same function can be performed by adding an extra input terminal to each NAND gate G_3 and G_4 of Fig. 7.7 (Prob. 7.6). With this modification incorporated in Fig. 7.7, we obtain the *J–K* FLIP–FLOP using NAND gates as shown in Fig. 7.9. The logic symbol of *J–K* FLIP–FLOP is given in Fig. 7.10.

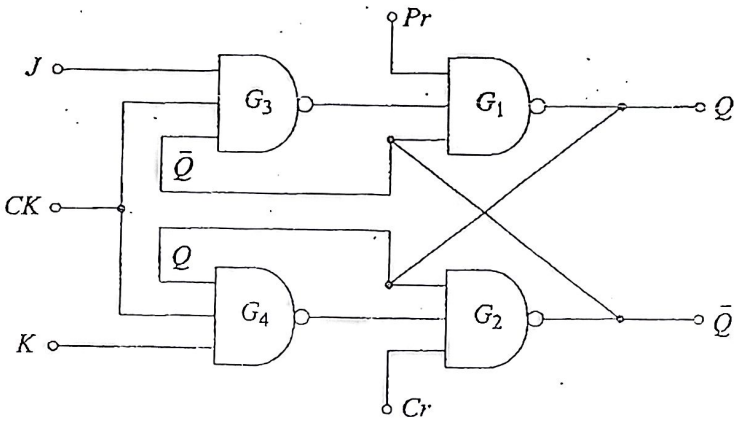


Fig. 7.9 *A J–K* FLIP–FLOP *Using* NAND *Gates*

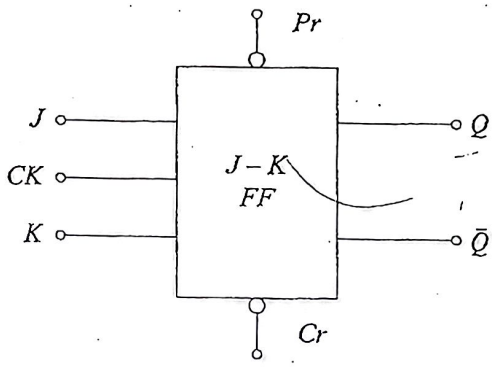


Fig. 7.10 *Logic Symbols of J–K* FLIP–FLOP

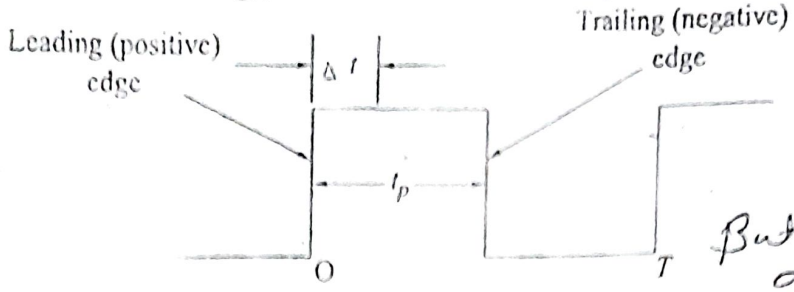
7.4.1 The Race-Around Condition

The difficulty of both inputs 1($S = R = 1$) being not allowed in an *S–R* FLIP–FLOP is eliminated in a *J–K* FLIP–FLOP by using the feedback connection from outputs to the inputs of the gates G_3 and G_4 (Fig. 7.9). Table 7.3 assumes that the inputs do not change during the clock pulse ($CK = 1$), which is not true because of the feedback connections. Consider, for example, that the inputs are $J = K = 1$ and $Q = 0$, and a pulse as shown in Fig. 7.11 is applied at the clock input. After a time interval Δt equal to the propagation delay through two NAND gates in series, the output will change to $Q = 1$ (see fourth row of Table 7.3b). Now we have $J = K = 1$ and $Q = 1$ and after another time interval of Δt the output will change back to $Q = 0$. Hence, we conclude that for the duration t_p of the clock pulse, the output will oscillate back and forth between 0 and 1. At the end of the clock pulse, the value of Q is uncertain. This situation is referred to as the *race-around condition.*

~~pulse~~ clk pulse time t_p > propagation delay Δt

To avoid Race-around.

clk pulse time t_p < prop. delay (Δt)



But diffi-lt to achieve as prop. delay is V. small.

Fig. 7.11 *A Clock Pulse*

The race-around condition can be avoided if $t_p < \Delta t < T$. However, it may be difficult to satisfy this inequality because of very small propagation delays in ICs. A more practical method for overcoming this difficulty is the use of the master–slave (*M–S*) configuration discussed below.

So Master Slave J-K.

7.4.2 The Master–Slave *J–K* FLIP-FLOP

A master–slave *J–K* FLIP-FLOP is a cascade of two *S–R* FLIP-FLOPs, with feedback from the outputs of the second to the inputs of the first as illustrated in Fig. 7.12. Positive clock pulses are applied to the first FLIP-FLOP and the clock pulses are inverted before these are applied to the second FLIP-FLOP.

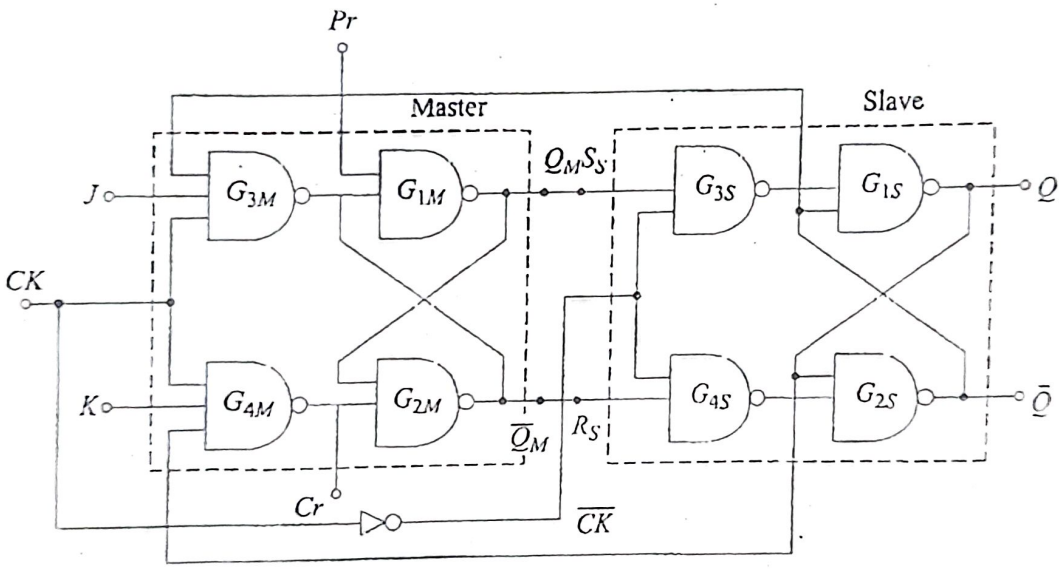


Fig. 7.12 *A Master–Slave J–K* FLIP-FLOP

When $CK = 1$, the first FLIP-FLOP is enabled and the outputs Q_M and $\overline{Q}_M$ respond to the inputs J and K according to Table 7.3. At this time, the second FLIP-FLOP is inhibited because its clock is LOW ($\overline{CK} = 0$). When CK goes LOW ($\overline{CK} = 1$), the first FLIP-FLOP is inhibited and the second FLIP-FLOP is enabled, because now its clock is HIGH ($\overline{CK} = 1$). Therefore, the outputs Q and $\overline{Q}$ follow the outputs Q_M and $\overline{Q}_M$, respectively (second and third rows of Table 7.3b). Since the second FLIP-FLOP simply follows the first one, it is referred to as the *slave* and the first one as the *master*. Hence, this configuration is referred to as *master—slave* (*M–S*) FLIP-FLOP.

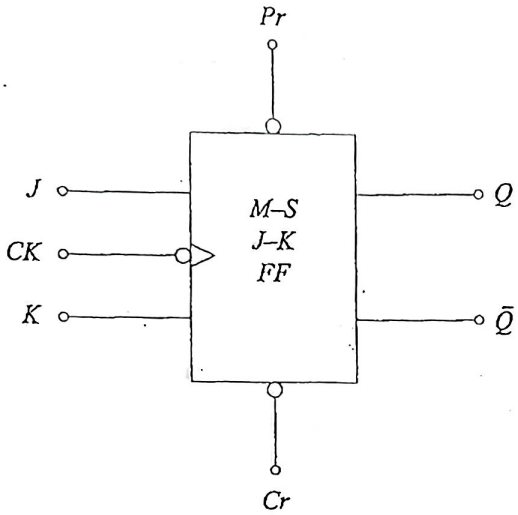


Fig. 7.13 *A Master–Slave J–K* FLIP–FLOP *Logic Symbol*

In this circuit, the inputs to the gates G_{3M} and G_{4M} do not change during the clock pulse, therefore the race-around condition does not exist. The state of the master–slave FLIP–FLOP changes at the negative transition (trailing edge) of the clock pulse. The logic symbol of a *M–S* FLIP–FLOP is given in Fig. 7.13. At the clock input terminal, the symbol > is used to illustrate that the output changes when the clock makes a transition and the accompanying bubble signifies negative transition (change in *CK* from 1 to 0).

7.5 *D*-TYPE FLIP–FLOP → Data input flipflop

↳ Delay flip-flop

If we use only the middle two rows of the truth table of the *S–R* (Table 7.1) or *J–K* (Table 7.3b) FLIP–FLOP, we obtain a *D*-type FLIP–FLOP as shown in Fig. 7.14. It has only one input referred to as *D*-input or data input. Its truth table is given in Table 7.4 from which it is clear that the output Q_{n+1} at the end of the clock pulse equals the input D_n before the clock pulse.

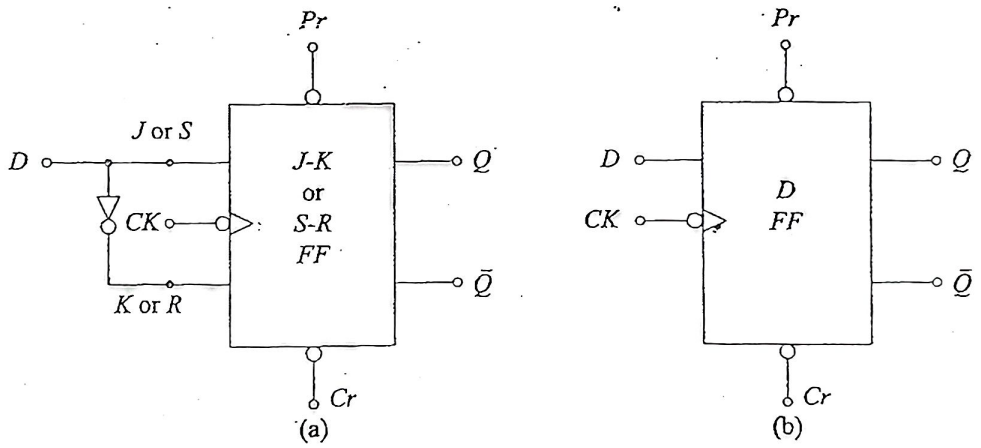


Fig. 7.14 *(a) A J–K or S–R* FLIP–FLOP *Converted into a D-type* FLIP–FLOP *(b) its Logic Symbol*

Table 7.4 *Truth Table of a D-type* FLIP–FLOP

Input D_n	Output Q_{n+1}
0	0
1	1

This is equivalent to saying that the input data appears at the output at the end of the clock pulse. Thus, the transfer of data from the input to the output is delayed and hence the name *delay* (*D*) FLIP-FLOP. The *D*-type FLIP-FLOP is either used as a delay device or as a latch to store 1-bit of binary information.

7.6 *T*-TYPE FLIP-FLOP

In a *J–K* FLIP-FLOP, if $J = K$, the resulting FLIP-FLOP is referred to as a *T*-type FLIP-FLOP and is shown in Fig. 7.15. It has only one input, referred to as *T*-input. Its truth table is given in Table 7.5 from which it is clear that if $T = 1$ it acts as a toggle switch. For every clock pulse, the output Q changes.

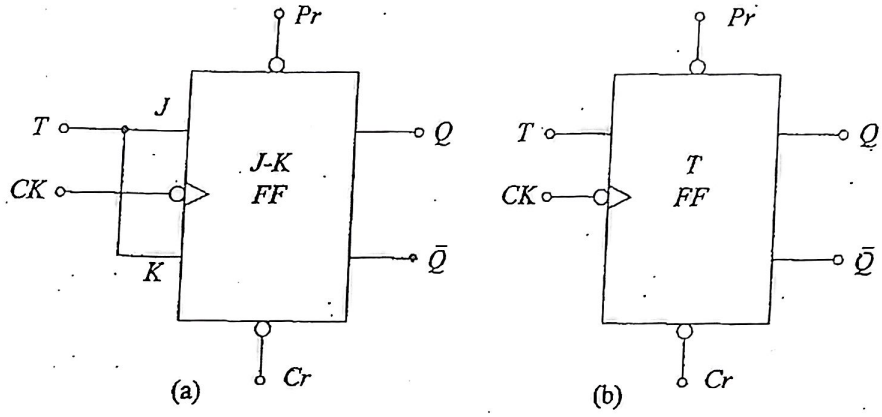


Fig. 7.15 *(a) A J–K* FLIP-FLOP *Converted into a T-type* FLIP-FLOP *(b) its Logic Symbol*

Table 7.5 *Truth Table of T-type* FLIP-FLOP

Input T_n	Output Q_{n+1}
0	Q_n
1	$\bar{Q}_n$

An *S–R* FLIP-FLOP cannot be converted into a *T*-type FLIP-FLOP since $S = R = 1$ is not allowed. However, the circuit of Fig. 7.16 acts as a toggle switch, i.e. the output Q changes with every clock pulse (Prob. 7.11).

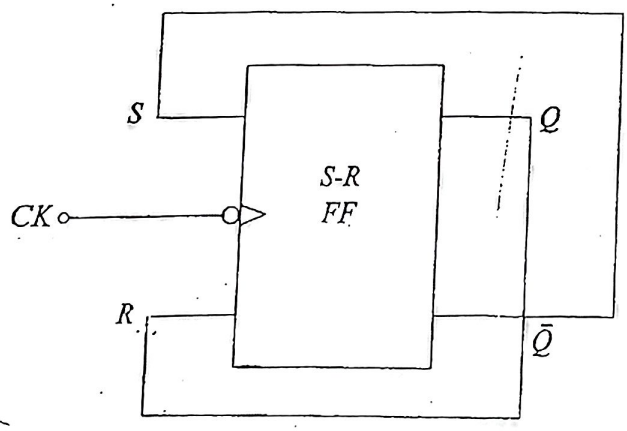


Fig. 7.16 *An S–R* FLIP-FLOP *as a Toggle Switch*

CHAPTER 8

SEQUENTIAL LOGIC DESIGN

8.1 INTRODUCTION

The FLIP-FLOP is a basic element of sequential logic system. Using FLIP-FLOPs and combinational logic circuits, any sequential logic circuit can be designed. The most important sequential circuits that are widely used in digital systems are registers and counters. These are discussed in detail in the following sections.

The design of registers and counters using FLIP-FLOPs has been introduced followed by standard MSI devices. Some of the common applications of registers and counters have also been discussed.

The design methods for general clocked sequential circuits have also been discussed. Basically, the system may be specified in terms of input–output relationship or/and the sequence of states to be followed when clock pulses are applied. The usual design steps are: reduction of states, state assignment and next-state decoder design.

The synchronous counter is a special case of the general clocked sequential circuit and can be designed using the design methods developed for clocked sequential circuits.

Asynchronous sequential circuits do not use clock pulses and their response depends upon the sequence in which the input signal changes. These circuits use delay circuits as memory elements. The delay is inherent in logic circuits and the feedback introduced in combinational circuits provides memory capability to these circuits. The S-R FLIP-FLOPs fall under this category of delay circuits which are extensively used in asynchronous sequential circuits. The analysis and design of asynchronous sequential circuits have been discussed in detail.

8.2 REGISTERS

As discussed in Chapter 7, a FLIP-FLOP can store (or remember) 1-bit of digital information (1 or 0). It is also referred to as a 1-bit *register*. An array of FLIP-FLOPs is required to store binary information, the number of FLIP-FLOPs required being equal to the number of bits in the binary word (one FLIP-FLOP for each bit) and is referred to as a register. Registers find application in a variety of digital systems including microprocessors. For example, Intel's 8085 microprocessor chip contains seven 8-bit registers, referred to as *general purpose registers* and five 1-bit registers, referred to as *flags*.

The data can be entered in *serial* (one-bit at a time) or in *parallel* form (all the bits simultaneously) and can be retrieved in the serial or parallel form. Data in serial form is also referred to as *temporal code* (a time arrangement of bits) and in parallel form as *spacial code*. A 4-bit data 1010 in serial form is shown in Fig. 8.1*a* and in parallel form in Fig. 8.1*b*. For serial input/output, only one line is required for data input and one line for data output, whereas the number of lines required is equal to the number of bits for parallel input/output.

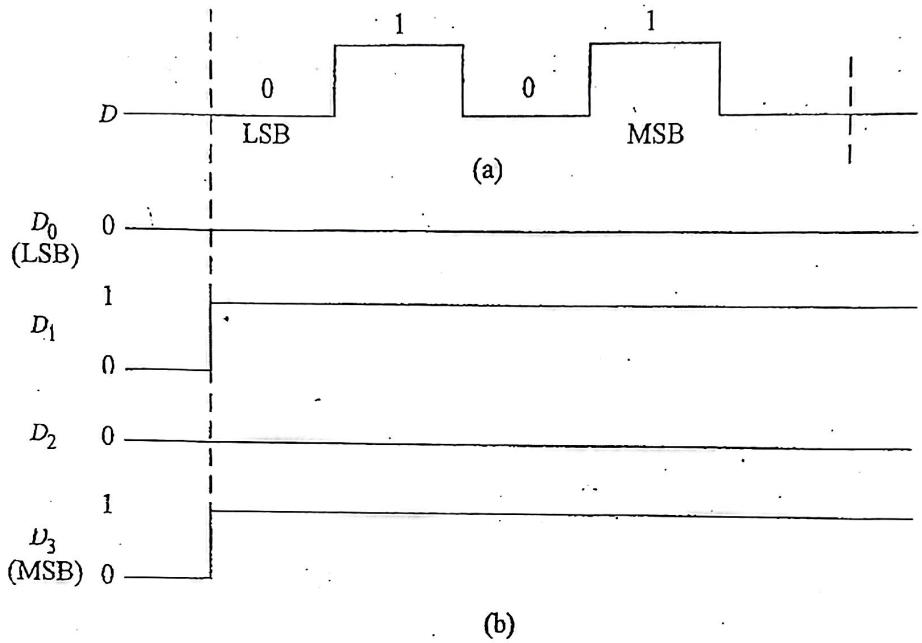


Fig. 8.1 ***Data Representation in (a) Serial Form (b) Parallel Form***

Registers are classified depending upon the way in which data are entered and retrieved. There are four possible modes of operation:

1. Serial-in, serial-out (SISO),
2. Serial-in, parallel-out (SIPO),
3. Parallel-in, serial-out (PISO), and
4. Parallel-in, parallel-out (PIPO).

Registers can be designed using discrete FLIP-FLOPs (*S-R* or *J-K* as *D*-type) and are also available as MSI devices. The registers available in 54/74-TTL and CMOS logic families are given in Table 8.1. The complete IC No. is different for different series. For example, 74AC164, 74ACT164, 74HC164, 74HCT164, 74ALS164A etc.

Registers in which data are entered or/and taken out in serial form are referred to as *shift registers*, since bits are shifted in the FLIP-FLOPs with the occurrence of clock pulses either in the right direction (*right-shift register*) or in the left direction (*left-shift register*). In the *bi-directional shift register*, data can be shifted from left to right as well as in the reverse direction, using the mode control. For example, IC 74295A is a bi-directional shift register.

A register is referred to as a *universal register* if it can be operated in all the four possible modes and also as a bi-directional register. For example, 74194 is a universal register.

Table 8.3 (*Continued*)

Number of clock pulses	FLIP-FLOP outputs Q_3	Q_2	Q_1	Q_0	Serial input Y
4	1	0	1	1	0
5	0	1	0	1	1
6	1	0	1	0	1
7	1	1	0	1	1
1	1	1	1	0	1
2	1	1	1	1	0
3	0	1	1	1	1
*	1	0	1	1	0
*	0	1	0	1	1
*	1	0	1	0	1

The sequence generator circuit can be designed using a 4-stage shift register using D-type FLIP-FLOPs and the decoder circuit given by Eq. (8.3).

8.4 RIPPLE OR ASYNCHRONOUS COUNTERS

A circuit used for counting the pulses is known as a *counter*. In Sec. 8.3, two types of counters have been discussed. The number of states in an N-stage ring counter is N, whereas it is $2N$ in the case of moebius counter. These counters are referred to as modulo N (or divide-by-N) and modulo $2N$ (or divide-by-$2N$) counters respectively, where modulo indicates the number of states in the counter. When the pulses to be counted are applied to a counter, it goes from state to state and the output of the FLIP-FLOPs in the counter is decoded to read the count. The circuit comes back to its starting state after counting N pulses in the case of modulo N counter.

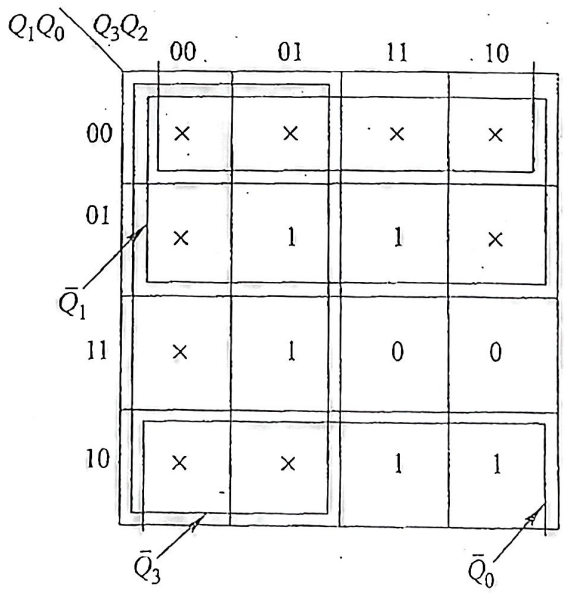


Fig. 8.9 *K-Map of Table 8.3*

The ring counter and the twisted-ring counter do not make efficient use of FLIP-FLOPs. A FLIP-FLOP has two states, therefore a group of N FLIP-FLOPs will have 2^N states. This means it is possible to make a modulo 2^N counter using N FLIP-FLOPs. Basically there are two types of such counters:

1. Asynchronous counter (ripple counter), and
2. Synchronous counter.

In the case of an asynchronous counter, all the FLIP-FLOPs are not clocked simultaneously, whereas in a synchronous counter all the FLIP-FLOPs are clocked simultaneously. The ring- and the twisted-ring counters are examples of synchronous counters.

Consider the count sequence shown in Table 8.4. The number of states in this sequence is 8 which requires 3 FLIP-FLOPs ($2^3 = 8$) and Q_2, Q_1, and Q_0 are the outputs of these FLIP-FLOPs. Assume master-slave FLIP-FLOPs.

Table 8.4 *Counting Sequence of a 3-bit Binary Counter*

Counter state	Count		
	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

The output Q_0 of the least-significant FLIP-FLOP, changes for every clock pulse. This can be achieved by using T-type FLIP-FLOP with $T_0 = 1$. The output Q_1 makes a transition (from 0 to 1 or 1 to 0) whenever Q_0 changes from 1 to 0. Therefore, if Q_0 is connected to the clock input of next T-type FLIP-FLOP, FF1 with $T_1 = 1$, Q_1 will change whenever Q_0 goes from 1 to 0 (falling edge of clock pulse). Similarly, Q_2 makes a transition whenever Q_1 goes from 1 to 0 and this can be achieved by connecting Q_1 to the clock input of the most-significant FLIP-FLOP, FF2 (and $T_2 = 1$). The resulting circuit is shown in Fig. 8.10. The waveforms of the outputs of the FLIP-FLOPs are shown in Fig. 8.11.

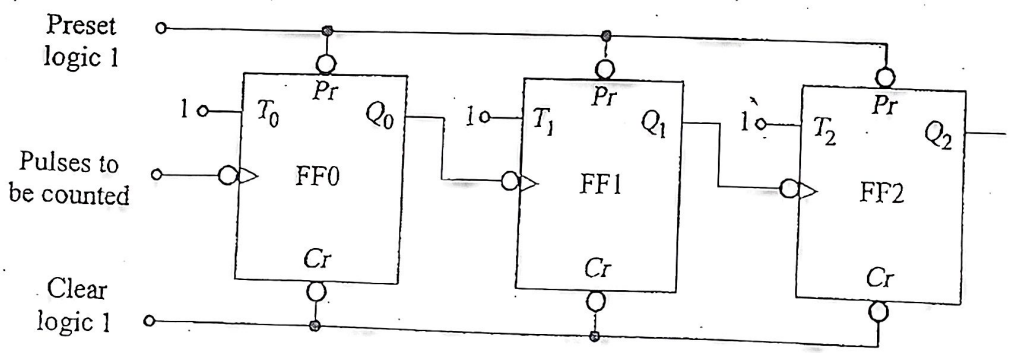


Fig. 8.10 *A 3-bit Binary Counter*

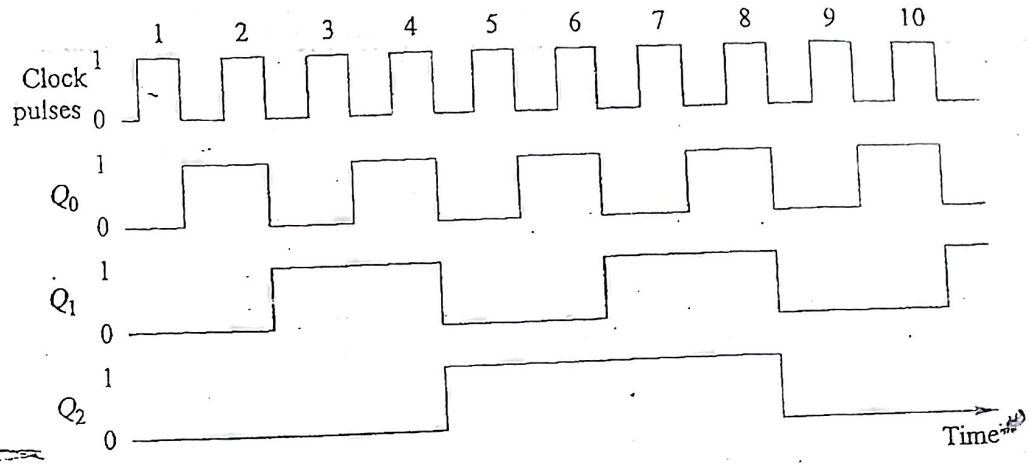


Fig. 8.11 *Output Waveforms of Counter of Fig. 8.10*

A decoder circuit for decoding the count is shown in Fig. 8.12. In this, the output corresponding to the number counted goes low (active-low).

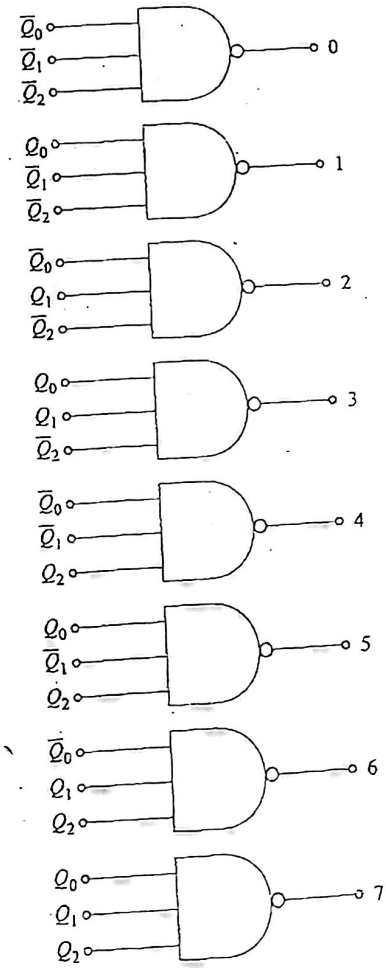


Fig. 8.12 *A Decoder Circuit for a 3-bit Binary Counter*