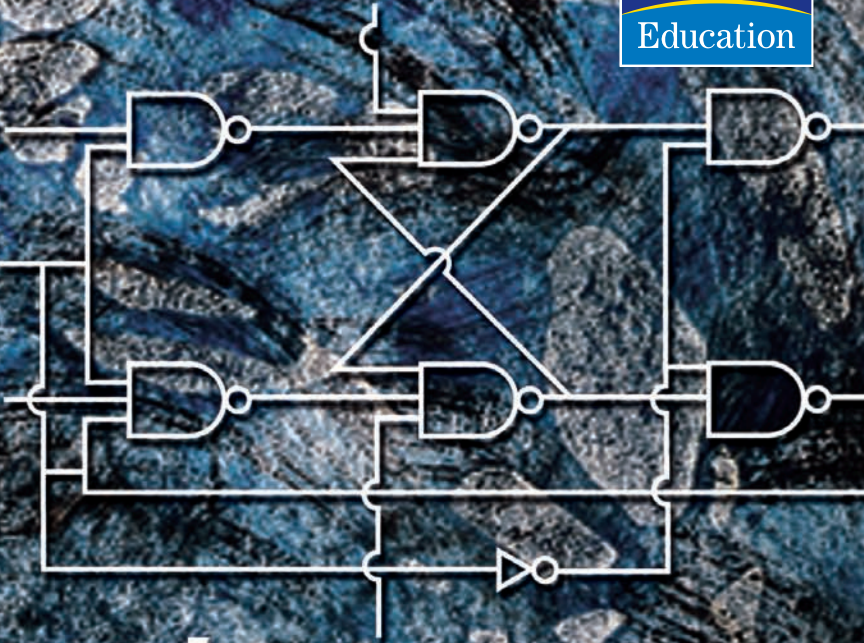
PEARSON
Education



Switching Theory & Logic Design

C V S Rao

Switching Theory
and
Logic Design

Switching Theory and Logic Design

C V S Rao

Formerly Principal and Professor of Electronics
University College of Engineering
Osmania University
Hyderabad, India

Copyright © 2006 Dorling Kindersley (India) Pvt. Ltd.

Copyright © 2005 Pearson Education (Singapore) Pte. Ltd.
This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and the above-mentioned publisher of this book.

ISBN 978-81-317-0183-6

First Impression, 2006
Second Impression, 2007
Third Impression, 2009

Published by Dorling Kindersley (India) Pvt. Ltd., licensees of Pearson Education in South Asia.

Head Office: 482, F.I.E., Patparganj, Delhi 110 092, India.
Registered Office: 14 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India.

Typeset by Bharati Composers, Delhi.

Printed in India at Baba Barkha Nath Printers.

To

The Great Achiever Par Excellence
Bhagvan Sri Satya Sai Baba
and
Ancestral Godfather Kanwarshy

This page is intentionally left blank

Preface

Switching Theory and Logic Design has found wide acceptance as a foundation course which is a prerequisite to many other courses like Digital Integrated Circuits, Computer Organisation, Digital Instrumentation, Digital Control, Digital Communications, Hardware Description Languages and so on. The principal objective of *Switching Theory and Logic Design* is to impart clarity to the concepts and problem-solving skills of the learner rather than merely stating information.

The book comprises 10 chapters, which can be used for a one semester course in the curriculum of a first degree in engineering. The coverage is more than what is required at the under-graduate level. Broadly, the first five chapters cover the combinational logic circuit design while chapters 6 to 10 cover the sequential logic circuits. Chapters 1 to 4 build the foundation for combinational logic circuit design. Chapter 5 deals with threshold logic and symmetric functions. Chapter 6 deals with different types of flip-flops and conversions from one to the other. Chapter 7 deals with synchronous sequential circuits. Depending on the time available, sections 7.6, 7.7 and 7.8 are optional in the first course. Chapter 8 deals with asynchronous sequential circuits which may be omitted in the first course. Chapter 9 deals with state reduction in sequential machines where section 9.1 through 9.5 are preferably covered in the first course in order to expose the student to some challenges and potentialities of the subject leaving the remaining sections for a more advanced course. It is in this chapter that a great deal of formalism becomes unavoidable. Chapter 10 enables the student to appreciate the elegance and usefulness of the ASM charts in designing the control unit of any digital system. In this chapter, all the knowledge acquired in combinational as well as sequential circuits is put to use in controlling the timing in a digital system.

Acknowledgement

I am grateful to Mr. K T Mahi, founder secretary of Sreenidhi Institute of Science and Technology for his support during the period of writing this book. I am also thankful to Dr P Narasimha Reddy, principal of the college, for providing free access to secretarial help and other amenities.

I appreciate the enthusiastic support of Mr G Bangar Reddy in meticulously preparing the manuscript. I also owe my thanks to Mr K Ramesh who typed the script in the initial stages.

Last, but not the least, is my wife Anjali Devi whose patience and forbearance has made me indebted to her for life. I appreciate the encouragement and cooperation extended to me by my children, Pratap, Karun and Savitri, and my son-in-law, Dr G V Ramana Rao.

C V S RAO

Contents

1

Introduction and Number Systems

Learning Objectives

After studying this chapter, you will know about:

- Number systems.
- Binary, octal and hexadecimal notation.
- Representation of negative numbers using 1's and 2's complement arithmetic.
- Codes, error detection, error correction, hamming code.

1.1 INTRODUCTION

Logic circuits, also referred to as **switching circuits** or digital circuits, perform logical functions involving mental processes. One of the simplest examples of a logic circuit is the staircase lighting circuit, which consists of a bulb controlled from two switches, situated at two ends of the staircase, in accordance with a certain logic. It is an EXCLUSIVE OR logic gate. Traffic signalling and control of elevators are examples of sequential logic circuits. A knowledge of switching theory is useful in designing digital computers, digital control systems, digital communication systems, instrumentation, telemetry and in a host of other applications.

Broadly, switching circuits can be classified into two categories:

1. Combinational switching circuits
2. Sequential switching circuits

While the outputs of a combinational switching circuit are functions of only the circuit inputs at the present moment, the outputs of a sequential switching circuit depend not only on the present inputs but also on the past history of the inputs. The objective of this book is to present some of the standard methods of designing combinational switching circuits first and subsequently sequential circuits in later chapters. Familiarity with the **binary number system** is a prerequisite to learning **logic circuits.**

Word statements of two simple problems are given below. It is useful to analyse the problems in an intuitive manner and obtain the solutions. As formal procedures are presented in due course, it will become evident that the solutions can be obtained in a methodical yet simple manner. Solutions to these examples will be discussed after the reader is exposed to the basics of Boolean Algebra.

Example 1.1 Admission to the Sreenidhi Engineering College can be granted provided the applicant qualifies in the Engineering Entrance Test (EET) and fulfills any one or more of the following eligibility criteria

1. The applicant should be in the top 1000 rankers in EET, over 17 years old and has not been involved in an accident resulting in a handicap.
2. The applicant is over 17 years old and has been involved in an accident resulting in a handicap.
3. The applicant is over 17 years old and has secured a rank in EET 1000 or better, and has been involved in an accident resulting in handicap.
4. The applicant is a male with a rank in EET better than 1000 and is over 17 years old.
5. The applicant is a female with a rank in EET better than 1000, is over 17 years old and has been involved in an accident resulting in a handicap.

Example 1.2 In a certain bank, the room with the safe deposit lockers has five locks—V, W, X, Y, Z—all of which must be unlocked for the room to open. The keys are distributed among five officers as follows

Mr. A has keys for V and Y
Mr. B has keys for W and V
Mr. C has keys for X and Z
Mr. D has keys for V and Z
Mr. E has keys for V and X

(a) Find all the combinations of officers who can open the locker room.

(b) Determine the minimum number of officers required to open the locker room.

(c) Who are the essential officers?

1.2 THE BINARY NUMBER SYSTEM

In most present-day digital machines, the devices employed make it necessary to use the binary number system. In the decimal system the **base**, also called **radix**, is 10 and ten symbols 0, 1, 2, …, 9 are required to represent any number. A decimal number actually represents a polynomial in powers of 10. For example,

$$(2357.468) = 2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 + 4 \times 10^{-1} + 6 \times 10^{-2} + 8 \times 10^{-3}$$

Thus, the individual digits are coefficients of powers of 10. The digit at the extreme right of the integral part, that is, to the immediate left of the decimal point is the coefficient of the 0th power of 10, the next digit is the coefficient of the first power of 10, and so on. The first digit to the right of the decimal point (radix point) is the coefficient of 10 to the power of –1 (minus one) and the next is the coefficient of 10 to the power of –2 (minus 2), and so on.

In the binary system, the base is 2 and only two symbols—0 and 1—are used to represent any number. Individual digits in the binary system are called 'bits'. The bits in a binary number represent coefficients of the power of 2. The base is usually indicated as a subscript while the number itself is enclosed in parentheses. For example,

$$\text{Decimal number } (23.625)_{10} = (10111.101)_2$$

Table 1.1 *Equivalent Numbers in Decimal and Binary Systems*

Decimal System	Binary System	Decimal System	Binary System
0	00000	16	10000
1	00001	17	10001
2	00010	18	10010
3	00011	19	10011
4	00100	20	10100
5	00101	21	10101
6	00110	22	10110
7	00111	23	10111
8	01000	21	10101
9	01001	25	11001
10	01010	26	11010
11	01011	27	11011
12	01100	28	11100
13	01101	29	11101
14	01110	30	11110
15	01111	31	11111

Table 1.2 *Powers of 2, Decimal Equivalents and Commonly Used Names*

N	2^n	Decimal Value	Commonly used names
8	2^8	256	
9	2^9	512	
10	2^{10}	1024	1 Kilo actually greater than 10^3
11	2^{11}	2048	2 K
12	2^{12}	4096	4 K
16	2^{16}	65536	64 K
20	2^{20}		1 Mega actually greater than 10^6
24	2^{24}		16 M
30	2^{30}		1 Giga actually greater than 10^9
32	2^{32}		4 Giga
40	2^{40}		1 Tera actually greater than 10^{12}

The decimal value of the binary number shown on the right-hand side of the above expression is easily computed as a polynomial in the power of 2 as given below.

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = (23.625)_{10}$$

A short list of equivalent integral numbers in decimal and binary systems is given in Table 1.1. Powers of 2, equivalent decimal number and commonly used names are given in Table 1.2.

Decimal to Binary Conversion

Conversion of a decimal number to binary form is achieved by treating the integer part and fractional part separately. In binary form, the integer part is obtained by successively dividing the integral part of the decimal number by 2 and listing the remainders from right to left. The fractional part is converted into binary form by successively multiplying by 2 and listing the integral part from left to right. This is illustrated below.

Conversion of integral part Suppose the number $(25)_{10}$ has to be converted to binary number. First, form two rows—top row for quotient and the bottom row for remainder-as shown in Table 1.3. Write the given number $(25)_{10}$ to the extreme right in the quotient row. Division by 2 yields quotient as 12 and remainder 1. These are entered as shown. Continue the process to its logical end, that is, until the quotient becomes 0 and remainder 1.

Table 1.3 *Illustrating D-B Conversion of Integers*

Quotient on division by 2	0	1	3	6	12	$(25)_{10}$
Remainder	1	1	0	0	1	Binary form

Conversion of fractional part If it is required to convert the fraction $(0.6875)_{10}$ into a binary fraction, two rows must be made-one for the decimal part on multiplication by 2 and the other for the integral part, as shown in **Table 1.4.** Write the given fraction to the extreme left in the top row. Multiplication by 2 yields the number 1.375. Write the integer part 1 in the bottom row and the fraction in the top row. Continue the process until the resulting fractional part is zero or until the desired number of fractional bits are obtained. In the end, place the radix point to the extreme left of the number written in the bottom row.

Table 1.4 *Illustrating D-B Conversion of Fractions*

$(0.6875)_{10}$	·375	·75	·5	·0	Conversion on multiplication by 2
·	1	0	1	1	Integral part on multiplication by 2

Thus $(25.68575)_{10} = (11001.1011)_2$
Check the result by computing the right-hand side as a sum of the power of 2.

1.3 OCTAL AND HEXADECIMAL NUMBERS

In digital systems, octal and hexadecimal numbers are frequently used to abbreviate long binary numbers. For octal numbers, the base is 8 and as many symbols denoted by 0, 1, 2, 3, 4, 5, 6, 7 are used. A number in the octal-based system will have a place positional value read from right to left as 8^0, 8^1, 8^2, and so on, starting from the digit next to the radix point towards the left. For the fractional part, the place positional values starting from the digit immediately to the right of the radix point will be 8^{-1}, 8^{-2}, 8^{-3} and so on, proceeding from left to right. For example,

$$(347.205)_8 = 3 \times 8^2 + 4 \times 8^1 + 7 \times 8^0 + 2 \times 8^{-1} + 0 \times 8^{-2} + 5 \times 8^{-3} = (231.26)_{10}$$

Decimal to octal conversion is obtained in a manner similar to the methods adopted for the binary number system. The only difference is that multiplication or division has to be by 8 instead of 2.

As the base 8 equals 2^3, which is an integral power of 2, it is easily seen that a binary number can be converted into an octal number by grouping the bits in triplets starting from the radix point and proceeding towards the left for the integral part and right for the fractional part. The integer portion and the fractional portion have to be treated separately. Remember, adding zeros to the left of the integral part and to the right of the fractional part does not change the value. For example,

$$(1, 010, 110.011, 101, 1)_2 = (126.354)_8$$

For hexadecimal numbers, the base is 16. Hence 16 symbols are needed. By convention, the 10 digits 0, 1, 2, 3, …, 9 and the alphabet A, B, C, D, E, F, totalling 16 symbols in all are used. The symbol ‘A’ will have a decimal value of 10, B will have 11, and so on upto F which will have a value $(15)_{10}$.

Hexadecimal numbers are indicated with a suffix Hex or simply H. For example,

$$\begin{aligned} & (23A4.EC)_H \\ = \ & 2 \times 16^3 + 3 \times 16^2 + 10 \times 16^1 + 4 \times 16^0 + 14 \times 16^{-1} + 12 \times 16^{-2} \\ = \ & (9124.9218)_{10} \end{aligned}$$

The reader is advised to check the above decimal value. Notice that the fraction part works out to 59/64 for which the first four decimal places are shown. Note that a fraction may keep on recurring in one number system but may terminate in another system with a different base. Binary to hexadecimal conversion is easily done by arranging the bits in groups of 4 starting from the radix point towards the left for the integral part and towards the right for obtaining the fractional part. For example,

$$101, 1101, 0111, 1010.1011, 01 = (5D7A.B4)_H$$

1.4 BINARY ADDITION AND SUBTRACTION

The two operands in the process of arithmetic addition are **Augend ‘A’** and **Addend ‘B’** which is added to the augend. The process produces the outputs **Sum ‘S’** and **carry C**. Table 1.5 depicts binary addition. Likewise, Minuned ‘X’ and Subtrahend ‘Y’ are the operands in the process of arithmetic subtraction which produces **Difference ‘D’** and **Borrow ‘B’**. Table 1.6 depicts binary subtraction.

Table 1.5 *Binary Addition*

Inputs		Outputs	
Augend	**Addend**	**Sum**	**Carry**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.6 *Binary Subtraction*

Inputs		Outputs	
Minuend	**Subtrahend**	**Difference**	**Borrow**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

1.5 REPRESENTATION OF NEGATIVE NUMBERS

In digital systems, negative numbers are usually represented in three different forms given below together with illustrations. Notice that ‘r’ represents the radix or base of the number system. ‘n’ is the word length in number of digits of the system under discussion. ‘N’ is used to represent numbers. The following discussion relates to decimal machines and binary machines.

Data Representation in Computers

1. Sign–Magnitude Form The digit (bit) in the most significant place is reserved to indicate the sign. Positive numbers are indicated by 0 (zero) Negative numbers are indicated by 9 for decimal numbers and 1 for binary numbers.

   Example: Word length = *n*

Dec. No	*Dec. M/c with n = 3*	*Bin M/c with n = 5*
+ 6	$\underline{0}$ 0 6	0 0 1 1 0
– 6	$\underline{9}$ 0 6	$\underline{1}$ 0 1 1 0

2. *(r –1)’s complement representation, that is, 9’s (1’s) complement representation*

   Def: *(r – 1)’s Complement of N =* $r^n - 1 - N$

   In decimal arithmetic, it is called 9’s complement which is obtained by subtracting each digit from 9. **In binary system, it is called 1’s complement and is obtained by subtracting each bit from 1 which is the same as replacing 0s with 1s and 1s with 0s—a process known as complemention or inversion.**

Dec. No	*9’s Comp*	*1’s Comp*
+ 6	$\underline{0}$ 0 6	0 0 1 1 0
– 6	$\underline{9}$ 9 3	$\underline{1}$ 1 0 0 1

3. *r’s complement representation, that is, 10’s (2’s) complement representation*

   Def: *r’s Complement of N =* $r^n - N$

   This is referred to as 10’s complement in decimal arithmetic which is obtained by subtracting the given number from 10^n where n is the word length. In binary system it is known as the 2’s complement which is obtained by adding ‘1’ to 1’s complement. An alternative method to obtain 2’s complement of a binary number is to scan the bits from right to left, retain the bits as they are until and including the first ‘1’ encountered and complement all the remaining bits.

Dec. No	*10's Comp*	*2's Comp*
+ 6	006	00110
– 6	994	11010

1.6 1'S AND 2'S COMPLEMENT ARITHMETIC

In digital machines, subtraction is usually formed by adding the complement of the subtrahend to the minuend. How exactly the arithmetic is performed by the digital machine is illustrated in this section using decimal numbers 6 and 5. 1's complement addition and 2's complement addition are illustrated in Tables 1.7 and 1.8. Notice that the word length of the decimal machine is 3 digits and that of the binary machine is 5 bits. Whenever the result of addition exceeds the capacity of the machine, an overflow is indicated by a carry. Notice that the most significant bit in the result indicates the sign of the number. In a decimal machine, negative numbers have 9 in the most significant place and in a binary machine, negative numbers have a '1' as the most significant bit. In case (ii) of 1's complement addition (Table 1.7), notice that there is an overflow carry which has to be added at the least significant position in order to get the correct sum. This is called '**End-**

Table 1.7 *Addition in Computers*

Using 1's Complement Arithmetic

Example: We will use the decimal numbers 6 and 5.

	Dec.No.	*9's Comp. Rep.*	*1's Comp. Rep.*
Case i	+ 6	006	00110
	+ 5	005	00101
	+ 11	011	01011
Case ii	+ 6	006	00110
	– 5	994	11010
	+ 1	1000	100000
Add End around Carry		↳1	↳1
		001	00001
Case iii	– 6	993	11001
	+ 5	005	00101
	– 01	998	11110
		Complement and assign –ve sign	
		– 001	– 00001
Case iv	– 6	993	11001
	– 5	994	11010
	– 11	1987	110011
		↳1	↳1
		988	10100
		Comp and assign –ve sign	
		– 011	– 01011

around-Carry' addition. In case (iii), the result indicates a negative number indicated by 1 in the binary machine (9 in the decimal machine) You have to complement the result and assign a negative sign to obtain the correct result. Case (iv) contains both the features of cases (ii) and (iii)

Table 1.8 illustrates the 2's (10's) complement addition of the same example. Notice, in particular, case (ii) in which overflow is simply ignored to get the correct result.

Table 1.8 *Addition in Computers*

Using 2's Complement Arithmetic

Case i	*Dec. No*	*10's Comp. Rep*	*2's Comp. Rep*
	+ 6	0 0 6	0 0 1 1 0
	+ 5	0 0 5	0 0 1 0 1
	+ 11	0 1 1	0 1 0 1 1
Case ii	+ 6	0 0 6	0 0 1 1 0
	– 5	9 9 5	1 1 0 1 1
	+ 1	1 0 0 1	1 0 0 0 0 1
			Ignore overflow
		0 0 1	0 0 0 0 1
Case iii	– 6	9 9 4	1 1 0 1 0
	+ 5	0 0 5	0 0 1 0 1
	– 1	9 9 9	1 1 1 1 1
			Complement & assign –ve sign
		– 0 0 1	– 0 0 0 0 1
Case iv	– 6	9 9 4	1 1 0 1 0
	– 5	9 9 5	1 1 0 1 1
	– 1 1	1 9 8 9	1 1 0 1 0 1
			Ignore overflow
		9 8 9	1 0 1 0 1
			Complement & Label –ve
		– 0 1 1	– 0 1 0 1 1

1.7 BINARY CODES

Hitherto, we have seen how numbers can be represented in binary form using place-positional values of 2^0, 2^1, 2^2, …, 2^i… same as 1, 2, 4, 8, 16, … which are called weights. Table 1.9 shows such a weighted binary code, also referred to as 8-4-2-1 code for 4-bit numbers. The corresponding decimal number is also indicated. There are some situations when it is preferable to have only one bit change between successive counts to ensure smooth changeover and avoid ambiguity. Such a code is called a **Gray Code** which is also indicated on the right hand side in the same Table 1.9. Notice that the code for 1 and the code for 2 differ only in one-bit position; $g_3 = g_2 = 0$, $g_0 = 1$ in both but $g_1 = 0$ for 1 and $g_1 = 1$ for 2. Similarly, the Gray Code for 11 is 1110 and the Gray Code for 12 is 1010, which differ only in one bit, g_2. Successive code words which differ in one bit only are said to be adjacent. The distance between two code words is the number of bits in which they differ. The distance is called the **Hamming Distance**. It is clear that no weights can be attached to the bits of gray

Table 1.9 *4-Bit Binary Code and Gray Code*

4 Bit Codes

Dec. No.	Binary Code				Gray Code			
	b_3	b_2	b_1	b_0	g_3	g_2	g_1	g_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

code. How then, do we obtain Gray Code from Binary Code? This is fairly simple. A little reflection will show that each Gray Code bit is obtained by applying EXCLUSIVE OR operation on the corresponding Binary Code bit and the next higher bit. This logical operation is discussed in greater detail elsewhere but for now the reader has to be content with the definition that

$$\begin{aligned} X \oplus Y &= 1 \text{ if } X \neq Y \\ &= 0 \text{ if } X = Y \end{aligned}$$

where the symbol ⊕ denotes the logical operation called **EXCLUSIVE OR**.

In other words, $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. This operation is called **Modulo-2 Addition**.

Thus, Gray Code bits are computed as

$$\begin{aligned} g_i &= b_i \oplus b_{i+1} \text{ for } 0 \leq i < n - 1 \\ g_{n-1} &= b_{n-1} \end{aligned}$$

Two examples are worked out below.

Example 1.3

B_1 = 1 1 0 1 0 1 1 1 0

Work from right to left to generate Gray Code bits ⊕

Gray G_1 = 1 0 1 1 1 1 0 0 1

Example 1.4

B_2 = 1 0 1 1 1 1 1 0 1

⊕

Gray G_1 = 1 1 1 0 0 0 0 1 1

An inquisitive reader might be interested in knowing how a pure binary number can be derived from a given Gray Code number. To convert Gray Code to pure binary, **start with the left-most bit and proceed to the right-most bit,** making $b_i = g_i$, if the number of 1's preceding and including g_i is odd and making $b_i = g_i^1$ if the number of 1's upto g_i is even where g_i^1 denotes the complement of g_i. This is equivalent to the formal expressions

$$b_{n-1} = g_{n-1}$$
$$b_i = g_i \oplus b_{i+1}.$$

The reader is advised to check the correctness of the above expressions with reference to the two examples learnt.

Reflected Codes

The Gray Code belongs to a class called **'reflected codes'.** These codes are characterised by the property that a *K*-bit code can be generated by **reflecting** (K-1)-bit code and prefixing 0 to the top code words and 1 to the bottom half. Table 1.10 illustrates the generation of 4-bit code words starting from the 1-bit code. Reflected codes are characterised by the property that successive code words differ in 1-bit position only. Further, the two terminal words namely, '0000' and the last word '1000' also differ in 1 bit only. This property of adjacensy of first and last words, makes it a 'Cyclic Code'.

Table 1.10 *Generation of Reflected Codes*

0	0 0	0 0 0	0 0 0 0
1	0 1	0 0 1	0 0 0 1
	Mirror axis—		
	1 1	0 1 1	0 0 1 1
	1 0	0 1 0	0 0 1 0
		Mirror axis—	
		1 1 0	0 1 1 0
		1 1 1	0 1 1 1
		1 0 1	0 1 0 1
		1 0 0	0 1 0 0
			Mirror axis—
			1 1 0 0
			1 1 0 1
			1 1 1 1
			1 1 1 0
			1 0 1 0
			1 0 1 1
			1 0 0 1
			1 0 0 0

1.8 BINARY CODES FOR DECIMAL DIGITS

In order to represent the decimal digits 0, 1, 2, …, 9, we need at least four bits with which we may form $2^4 = 16$ distinct 4-bit words. Ten of these will be used and the remaining six ignored. Some weighted codes and non-weighted codes are given in Table 1.11 and Table 1.12, respectively.

Table 1.11 *Weighted Binary Codes*

Decimal Digit	(BCD)	
	8 – 4 – 2 – 1	2 – 4 – 2 – 1
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 1
4	0 1 0 0	0 1 0 0
5	0 1 0 1	1 0 1 1
6	0 1 1 0	1 1 0 0
7	0 1 1 1	1 1 0 1
8	1 0 0 0	1 1 1 0
9	1 0 0 1	1 1 1 1

Table 1.12 *Non-weighted Binary Codes*

Decimal Digit	Excess–3	Cyclic
0	0 0 1 1	0 0 0 0
1	0 1 0 0	0 0 0 1
2	0 1 0 1	0 0 1 1
3	0 1 1 0	0 0 1 0
4	0 1 1 1	0 1 1 0
5	1 0 0 0	1 1 1 0
6	1 0 0 1	1 0 1 0
7	1 0 1 0	1 0 1 1
8	1 0 1 1	1 0 0 1
9	1 1 0 0	1 0 0 0

The 8-4-2-1 weighted code is the natural binary number code and is called the Binary-Coded-Decimal (BCD) code. The 2-4-2-1 code assigned in Table 1.11 has a special property that makes it a **self-complementing code**. If a code word is complemented bit by bit, it would result in 9's complement of the original digit. The code for the digit 4 is 0100. On complementation bit-by-bit, one gets 1011, which represents the digit 5. Notice that the assignment is not unique. The particular assignment given has the property of self-complementation. Notice also that the BCD code is not self-complementing. Self-complementing codes have some advantage in implementing arithmetic in digital computers.

It is possible to assign negative weights too. For instance, one can use the weights 8, 4, (–2), (–1) and construct a unique code for decimal digits, which is self-complementing. A little reflection reveals that the sum of the weights should be 9.

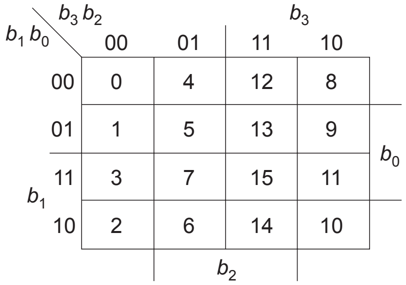
Likewise, one may use the weights 6, 4, 2, (–3) and construct a code for decimal digits. The reader is advised to construct the code to know that it is possible to make it a self-complementing code by choosing the appropriate code words.

Two non-weighted codes, Excess-3 and cyclic code, are given in Table 1.12. The code for 3 in BCD code becomes the code word for 0 in Excess-3 code. In other words, by adding 0011 to each word of BCD, you get the corresponding Excess-3 code word. Excess-3 code has the property of self-complementation just like the 2-4-2-1 weighted code.

The cyclic code given in Table 1.12 has the property that the successive code words differ only in one bit and this property of **adjacency** holds good for the terminal digits 0 and 9 too. The feature of adjacency can be advantageous in logic design, to be covered in subsequent chapters.

The cyclic code for the decimal digits is merely a subset of the 4-bit Gray Code, selecting a sequence of 10 code words with **adjacency ordering** in such a way that the first and last words are adjacent, that is, they differ only in one bit. There can be many such cyclic codes. One commonly used cyclic code is the sequence of binary codes for the digits 0, 1, 3, 2, 6, 14, 10, 11, 9, 8. This is easily remembered with the help of a map as in Fig. 1.1.

In Fig. 1.1, notice in particular the way in which the columns are labelled with the higher significant bits b_3 b_2 and rows with the lower significant bits b_1 b_0. Notice specially the ordering of labels. After 00, 01, note that 11 follows 01 and 10 follows 11. This is called **adjacency ordering** and ensures that the successive labels differ only in one bit. Observe that the left-most column ‘00’ is adjacent to the right-most column labelled ‘10’. Likewise the top row ‘00’ is adjacent to the bottom row labelled as ‘10’. Such an ordered map is called a **Karnaugh** map. Notice that the regions where each variable assumes the value ‘1’ are also marked by extending some lines in drawing the map. For instance, the variable b_3 is ‘1’ to the right of the extended line on top and b_2 is ‘1’ in the region bounded by the lines extended below. Similarly, b_1 is ‘1’ below the line on the left and b_0 is ‘1’ between the lines on the right.



$b_1 b_0$ \ $b_3 b_2$	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Fig. 1.1 Illustrating 4-bit binary codes for decimal numbers on a map with adjacency labelling called a Karnaugh map.

Each cell in the Karnaugh map of Fig. 1.1 is marked with the corresponding decimal number. For example, the cell labelled b_3b_2 b_1 b_0 = 1110 is marked with 14. It is now easy to locate the codes selected above by travelling along in a U shape starting from cell ‘0’ and proceeding to cell ‘8’ and assigning codes for the digits 0, 1, 2, …, 9 as in Table 1.12. The reader should now realise that the Gray Code of Table 1.9 can be derived by scanning the columns of Fig. 1.1 in the order 0 to 2, 6 to 4, 12 to 14, 10 to 8. Notice that cells 0 and 8 are adjacent in their codes.

In order to get a feel for the different codes for decimal digits learnt so far, let us refer to Example 1.5.

Example 1.5 Represent the number $(3952)_{10}$ in the following codes.

(a) BCD code (b) 2-4-2-1 code

(c) Excess-3 code (d) Cyclic code of Table 1.12.

The required representations are:

(a) 3952 = 0011, 1001, 0101, 0010 (BCD)

(b) 3952 = 0011, 1111, 1011, 0010 (2-4-2-1)

(c) 3952 = 0110, 1100, 1000, 0101 (Excess-3)

(d) 3952 = 0010, 1000, 1110, 0011

1.9 ERROR DETECTION AND CORRECTION

Although four bits are adequate to represent decimal digits, binary signals may get corrupted during transmission because of noise in the medium or equipment malfunction. It is safe to assume that the probability of occurrence of a single error has a finite value. Experience has proved that the probability of occurrence of two or more errors is relatively much much smaller. Two punctures occurring at the same time for a vehicle tyre are rare. For such reasons, fault detection and single fault correction assumed importance. These are discussed in this and the next section.

Table 1.13 shows two schemes. One uses transmission of an extra bit along with the original code word of four bits, called message in this context. This extra bit is called a ‘parity bit’ which can be 0 or 1 depending on the designer’s wish to make the number of 1’s in each coded message even or odd. An even-parity BCD code is shown in the table. At the receiving end, it is easy to discard the message which does not show up the prescribed parity.

In the code shown on the right hand side in Table 1.13, each message contains exactly two 1s and three 0s. Hence the name. Only 10 such distinct combinations are possible. It is easy to detect the culprit at the receiver by simply counting the 1s in each message. The code is not unique. For a given digit, the code word depends on the designer’s assignment. Even so, one such assignment is interesting in that all the digits except ‘0’ can be derived by assigning weights 0, 1, 2, 4, 7. The digits 1, 2, 4, 7 are realised by assigning 1’ in the corresponding place and another ‘1’ in the 0th position too. The number 3 is expressed as 1 + 2, 5 as 1 + 4, 6 as 2 + 4, 8 as

Table 1.13 *Error-Detecting Codes*

Decimal Digit	Even-parity BCD					2-out of – 5				
	8	**4**	**2**	**1**	**p**	**0**	**1**	**2**	**4**	**7**
0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	1	1	1	0	0	0
2	0	0	1	0	1	1	0	1	0	0
3	0	0	1	1	0	0	1	1	0	0
4	0	1	0	0	1	1	0	0	1	0
5	0	1	0	1	0	0	1	0	1	0
6	0	1	1	0	0	0	0	1	1	0
7	0	1	1	1	1	1	0	0	0	1
8	1	0	0	0	1	0	1	0	0	1
9	1	0	0	1	0	0	0	1	0	1

Table 1.14 *Error Codes for Different Positions*

Position		Error Code		
		c_2	c_1	c_0
(No Error)	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

1 + 7, 9 as 2 + 7. Nine combinations are thus used. The only remaining assignment with two 1's is used to represent 0.

1.10 SINGLE ERROR CORRECTING HAMMING CODE

Four bits are needed to represent the digits 0, 1, …, 9. Suppose we augment each code word with three extra bits called parity bits and generate a 7-bit code $b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7$. It is then possible that a single error can be corrected. In order to develop such a code, let us assume a 3-bit error code $c_2\, c_1\, c_0$, which indicates the position of the error. If the error code is 0 0 0 it would imply that there is no error in the transmission of the code $b_1 b_2 \ldots b_7$. If the error code is 001, it would imply that the bit b_1 is in error and hence the received message can be corrected by simply complementing b_1. The problem is now reduced to generating the error code from the received 7-bit message.

Table 1.14 shows the error position and the corresponding error code. Notice that $c_0 = 1$ implies error in any one of the positions 1, 3, 5, 7. If we assign the code in such a way that the bits b_1, b_3, b_5, b_7 have even parity and if the corresponding transmitted bits have odd parity, c_0 has to become 1 indicating error in any one of these positions. The required logic is simple. $c_0 = b_1 \oplus b_3 \oplus b_5 \oplus b_7$.

Likewise $c_1 = 1$ implies an error in one of the positions 2, 3, 6, 7, that is, b_2, b_3, b_6, b_7. We originally assign all the code words in such a way that the bits b_2, b_3, b_6, b_7 have even parity in each code word. If the parity changes during transmission, an error will be detected making $c_1 = 1$. Thus c_1 is synthesised as

$$c_1 = b_2 \oplus b_3 \oplus b_6 \oplus b_7.$$

Similarly, c_2 depends on the parity of b_4, b_5, b_6, and b_7.

$$c_2 = b_4 \oplus b_5 \oplus b_6 \oplus b_7.$$

From the above discussion, it is easy to conclude that code words must be assigned to ensure parity in the following groups of bits.

$$b_1,\ b_3,\ b_5,\ b_7$$

$$b_2,\ b_3,\ b_6,\ b_7$$

$$b_4,\ b_5,\ b_6,\ b_7$$

In order to assign the bits of the code, a little reflection reveals that b_1, b_2, b_4 are to be chosen as parity bits as each of these bits occurs only in one group. As such each parity bit can be assigned independently of other parity bits. It follows that the remaining bits b_3, b_5, b_6, b_7 become the message bits. The positions of parity bits are indicated by P_i and those of message bits as m_i in the complete code word as given below.

$$\begin{array}{ccccccc} b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \\ P_1 & P_2 & m_3 & P_4 & m_5 & m_6 & m_7 \end{array}$$

Each word consists of four message bits m_3, m_5, m_6, m_7 and three parity bits P_1, P_2, P_4, a total of seven bits. The error may occur in any one of the seven positions or in none as follows.

- Notice that c_0 must become 1 if the error position is 1, 3, 5, or 7. Select P_1 to have even parity among these positions.
- Similarly, select P_2 to have even parity among the positions 2, 3, 6, 7
- Select P_4 to have even parity among the bits in positions 4, 5, 6, 7.

Generation of Hamming Code for decimal digits is illustrated in Table 1.15. Notice that the code word for decimal 8 is considered as an example. BCD m_3 m_5 m_6 m_7 = 1 0 0 0 is first written in the corresponding positions 3, 5, 6, 7. In the next row, P_1 is assigned $\underline{1}$ to ensure even parity among the bits 1, 3, 5, 7. Bits so assigned are **underscored** for focus. In the next step, P_2 is assigned 1 to assure even parity among the bits 2, 3, 6, 7. In the next row, P_4 is finally assigned 0 to have even parity among the bits 4, 5, 6, 7. This completes

Table 1.15 *Generation of Hamming Code for Decimal Digits*

Example Message word is 0 1 0 0.
Choose parity bits in positions 1, 2, 4 (one from each group)

	b_1 P_1	b_2 P_2	b_3 m_3	b_4 P_4	b_5 m_5	b_6 m_6	b_7 m_7
Positions:	1	2	3	4	5	6	7
Original BCD word:	–	–	1	–	0	0	0
P_1 for even parity of 1, 3, 5, 7:	$\underline{1}$	–	1	–	0	0	0
P_2 for even parity of 2, 3, 6, 7:	–	$\underline{1}$	1	–	0	0	0
P_4 for even parity of 4, 5, 6, 7:	1	1	$\underline{1}$	0	0	0	0

Hamming Code for BCD

Decimal Digit	**Position**	**1** p_1	**2** p_2	**3** m_3	**4** p_4	**5** m_5	**6** m_6	**7** m_7
0		0	0	0	0	0	0	0
1		1	1	0	1	0	0	1
2		0	1	0	1	0	1	0
3		1	0	0	0	0	1	1
4		1	0	0	1	1	0	0
5		0	1	0	0	1	0	1
6		1	1	0	0	1	1	0
7		0	0	0	1	1	1	1
8		1	1	1	0	0	0	0
9		0	0	1	1	0	0	1

the generation of the 7-bit code word for the decimal digit '8' as **1 1** 1 **0** 0 0 0, in which the generated parity bits are underscored while the remaining are the original message bits.

The single error-correcting Hamming Code for the BCD digits is shown below.

In order to complete the discussion, let us now demonstrate how an error can be detected and corrected with an example illustrated in Table 1.16.

Table 1.16 *Illustrating Correction of Received Code Word*

Example Let it be that the word 1 1 0 1 0 0 1 is transmitted and 1 1 0 1 1 0 1 is received.

Procedure to compute error code

Position:	1	2	3	4	5	6	7	
Message received:	1	1	0	1	1	0	1	
4-5-6-7 parity check:	–	–	–	1	1	0	1	c_2 = 1 as parity is odd
2-3-6-7 parity check:	–	1	0	–	–	0	1	c_1 = 0 as parity is even
1-3-5-7 parity check:	1	–	0	–	1	–	1	c_0 = 1 as parity is odd

Error code = $c_2\ c_1\ c_0$ = 101 which means b_5 is in error, complement b_5 to get the correct message.

Note that the error may occur even in the parity check bits, which will be detected by the above procedure.

A little reflection reveals that the Hamming Code constructed as above ensures a minimum distance of three between any pair of code words. One can easily visualise and furnish a proof for this statement. Consider any two original 4-bit distinct messages, M_i and M_j. As they are different, it follows that the distance $d \geq 1$. Suppose $d = 1$ which implies that they differ in one of the message bits, b_3, b_5, b_6 or b_7. Let us call this differing message bit b_k. Since each message bit participates in at least two parity checks, the check bits yielded by performing parity checks with b_k must be different for M_i and M_j. If the differing message bit is b_7, then the parity bits b_1 and b_2 will differ and b_4 has to be the same as one of them. This assures an additional distance of two in the code word, which adds to the inherent distance between the pair of messages. This argument holds true for any number of differing bits in a pair of messages. Hence, a minimum distance of three is assured in any pair of code words. This is illustrated in the following example.

Example 1.6

To demonstrate $d \geq 3$.

		b_1	b_2	b_3	b_4	b_5	b_6	b_7
		P_1	P_2	m_3	P_4	m_5	m_6	m_7
Message	M_i			0		1	1	1
	M_j			0		0	1	1
Coded word	W_i	$\underline{0}$	0	0	$\underline{1}$	1	1	1
	W_j	$\underline{1}$	0	0	$\underline{0}$	0	1	1

Consider two 4-bit messages $M_i\,(b_3, b_5, b_6, b_7) = 0111$ and $M_j\,(b_3, b_5, b_6, b_7) = 0011$, **which differ in b_5 only**. We already know b_5 participates in two parity checks given below.

For M_i parity check of bits 1, 3, 5, 7 yields $P_1 = 0$

and check of bits 4, 5, 6, 7 yields $P_4 = 1$

For M_j parity check of bits 1, 3, 5, 7 yields $P_1 = 1$

and check of bits 4, 5, 6, 7 yields $P_4 = 0$

The differing check bits are underscored to put in focus. The check bit P_2 does not depend on b_5. Instead it depends on the parity check of bits 2, 3, 6, 7 which yields '0' for both M_i and M_j.

By adopting the procedure used, the reader is advised to construct the single error correcting 7-bit Hamming Code for all 16-4-bit messages.

Alphanumeric Codes

Digital systems have attained supremacy over analog systems mainly because of the features of noise immunity and arbitrarily high accuracy attainable. Modularity, software flexibility, reliability and the ability to be tested are some additional advantageous features of digital circuits and systems. Digital calculators, digital computers, digital communication systems, digital control in Instrumentation are some of the application areas. Digital circuits rely heavily on an electronic circuit called 'Flip-Flop', also called 'Binary' as it remains in one of the two possible states at any given time. The states are usually called '0' and '1'. Owing to the availability of binary devices, binary number systems and binary codes have assumed importance. Hitherto, we discussed how to convert numbers with one base to an equivalent number with another base—decimal to binary, binary to octal, hexadecimal and vice-versa.

We will now introduce the concept of alphanumeric codes. A code is a string of bits (0s or 1s) assigned to a symbol (at times called a word or message), which may or may not have a numerical value. For instance, if we wish to develop a distinct code for the alphabet A to Z in lower case (26), upper case (26) and numerals 0 to 9 (10), and 20 special symbols and punctuation marks, we need to have 82 distinct strings of 0s and 1s. For this purpose, we need to assign 7-bit positions resulting in $2^7 = 128$ distinct code words. Such a code is called a 'alphanumeric code'. For man-machine communication, such a code exists in the American Standard Code for Information Interchange, abbreviated as ASCII (pronounced askee) which is widely used for computer key boards and printers. ASCII is a 7-bit code which provides $2^7 = 128$ distinct codes, which can be represented in hexadecimal notation as ranging from (0 0) H to (7 F) H. For example, the letter A in uppercase is represented by (4 1) H and a in lowercase is denoted by (6 1) H. ASCII code (3 9) H = 011, 1001 denotes the decimal digit 9. Appendix A gives details of ASCII.

In practice, one parity bit is used at the extreme left to make all code words have either an even number of 1s or odd numbers of 1s depending on the choice of the user. Parity bits help in error detection and, at times, correction also. That makes ASCII an 8-bit code with code words ranging from (0 0) H to (F F) H.

Another code called the Extended Binary Coded Decimal Interchange Code (EBCDC) is also used although less common. It also employs eight bits including one parity bit.

SUMMARY

Number systems with different basics are covered. The binary, octal and hexadecimal codes are given special emphasis. Representation of negative numbers using 1's and 2's complement techniques are explained. Error detection and correction methods are illustrated. Binary, gray, cyclic and Hamming codes for decimal numbers is covered.

KEY WORDS

- Binary number system
- Grey code
- Reflected code
- Adjacency ordering
- Cyclic code
- Error detection
- Error correction
- Hamming code

REVIEW QUESTIONS

1. Represent the number 1234 in

   a) Binary b) BCD c) Excess-3 code d) Gray Code

2. Find the binary number equivalent of 0.875.
3. A binary with n bits all of which are 1s has the value

   a) $n^2 – 1$ b) 2^n c) $2^{(n–1)}$ d) $2^n – 1$

4. The Gray Code for number 6 is

   a) 1100 b) 1001 c) 0101 d) 0110

5. Convert $(1BC.D)_H$ into a binary number.
6. Convert $(0.6875)_{10}$ into an octal number.
7. If $\sqrt{41} = 5$, find the radix (base) of the number system.
8. A class has 60 students. They need to be assigned binary roll numbers. How many bits are required?
9. Find (110010)/(101) by using binary division.
10. Convert $(.6875)_{10}$ to octal.
11. Convert $(306.D)_{16}$ to binary.
12. Find the radix of the number system if 302/20 = 12.1.
13. 1101 × 110 = (?)
14. 110010/101 = __________.
15. Determine the value of base r if $(121)_r = (144)_8$.
16. Given that $(192)_{10} = (196)_b$, determine the value of b.
17. The most important advantage of a digital system over an analog system is

   a) Higher speed b) Lower cost c) Modularity d) Noise immunity

18. In a digital system, performance accuracy depends on

   a) Nature of devices used b) Number of gates
   c) Word length d) Propagation delay

19. The accuracy obtainable in a fixed-point binary system with word length of n bits is

   a) ± 1/2 ($2^{–n}$) b) ± 1/2 (2^{+n}) c) ± 2^n d) ± $2^{–n}$

20. The operational accuracy of a digital systems is given by

   a) ± 1/2 (Weight of the least significant digit) b) ± 1/2 (Weight of the most significant digit)
   c) ± (Weight of MSD) d) ± (Weight of LSD)

21. The ‘resolution’ of a digital system is given by
    a) Weight of the MSB b) Weight of the LSB
    c) ½ (Weight of MSB) d) ½ (Weight of LSB)
22. In an analog system, performance accuracy depends on
    a) Noise generated in the system b) Power supply variations
    c) Non-linear operation d) All the above and others
23. How many bits are needed to encode all letters (26), 10 symbols, and all numerals (10)?
    a) 4 b) 5 c) 6 d) 7
24. How many binary numbers are created with 8 bits?
    a) 256 b) 128 c) 64 d) 16
25. Number 84 in BCD form is given by
    a) 1000 0100 b) 0100 0100 c) 1000 1010 d) 1000 1100
26. The highest decimal number that can be represented with 10 binary digits is
    a) 1023 b) 1024 c) 512 d) $2^{11} - 1$
27. How many fractional bits will 2^{-48} have?
    a) 24 b) 47 c) 49 d) 48
28. What is the hexadecimal equivalent of the binary number 1010, 1111?
    a) AF b) 9E c) 8C d) 1015
29. What is the hexadecimal equivalent of the decimal number 511?
    a) FF1 b) 1FF c) 3FF d) FF3
30. The octal equivalent of decimal 324.875 is
    a) 504.7 b) 540.7 c) 215.2 d) 40.9
31. Which of the following codes is known as the 8421 code?
    a) Gray Code b) Excess-3 Code c) ASCII Code d) BCD Code
32. The sum of weights in a self-complementing BCD code must be
    a) 7 b) 9 c) 10 d) 8
33. ASCII
    a) is a subset of 8-bit EBCDIC b) is used only in western countries
    c) is Version II of the ASC Standard
    d) has 128 characters, including control characters
34. Which number system is not a positional number system?
    a) Roman b) Binary c) Decimal d) Hexadecimal
35. Let $(A\ 2\ C)_{16} = (X)_8$. Then X is given by
    a) 7054 b) 6054 c) 5154 d) 5054
36. Consider *n*-bit one’s complement representation of integer numbers. The range of integer values, *N*, that can be represented is
    a) $-2^{n-1} \leq N \leq 2^{n-1}$ b) $1 - 2^{n-1} \leq N \leq 2^{n-1} - 1$
    c) $-1 - 2^{n-1} \leq N \leq 2^{n-1} - 1$ d) $-1 - 2^{n-1} \leq N \leq 2^{n-1}$

37. The Excess-3 Code is also known as

a) Cyclic Code b) Weighted Code
c) Self-complementing Code d) Error-correcting Code

38. In which code do the successive code characters differ in only one position?

a) Gray Code b) Excess-3 Code c) 8421 code d) Hamming Code

39. The number of 1s present in the binary representation of 3 × 512 + 7 × 64 + 5 × 8 + 3

a) 7 b) 8 c) 9 d) 12

40. How many 1s are present in the binary representation of
15 × 256 + 5 × 16 + 3?

a) 8 b) 9 c) 10 d) 11

41. Add the given numbers without conversion into another radix.

a) $(4\ 5\ 7)_8 + (7\ 4\ 6)_8$ b) $(1\ F\ G\ C)_H + (D\ B\ 6\ A)_H$

42. Obtain the 1s and 2s complements of the given number.
1 0 1 1, 0 0 1 1, 1 0 1 1

43. What would be the 9s and 10s complements of the given number?
1 0 9 2 3 4 0

44. a) Perform the binary subtraction
0 1 1 1, 0 0 0 0 – 0 1 0 1, 1 1 1 1
b) Check by adding 1s complement of the addend.
c) Check by adding 2s complement of the addend.

45. Make the indicated conversions for the following numbers.

a) $(432)_5 = (\qquad)_7$ b) $(1431)_8 = (\qquad)_{10}$

46. a) Find 10s complement of $(935)_{11}$.
b) Convert $(347)_8$ to $(\qquad)_{16.}$

47. Convert 384.6875 to binary, octal, and hexadecimal systems.

48. Using 2s complement addition method, find the result of 01100-00011.

49. Perform 1101, 1101 + 1011.10 = $(\qquad)_2$.

50. Convert the following numbers

a) $(1431)_8$ to base 10 b) $(53.1575)_{10}$ to base 2

51. Express the number $(10110.0101)_2$ in the decimal system.

52. Find the 16s complement of AF3B.

53. Find the 9s complement of 6027.

54. Determine the possible base of the following arithmetic operation.
302/20 = 12.1

55. In which of the following codes do the successive characters differ in only 1-bit position.

a) 8421 Code b) 2421 Code c) Cyclic Code d) Binary Code

56. Which of the following codes is a self-complementing code?

a) Excess-3 b) Gray Code c) Hamming Code d) Cyclic Code

57. What is the word length of the Hamming Code to represent decimal digits? How many of them are parity bits?

PROBLEMS

1. Find the possible base so that the given expression is valid.

   a) 52/4 = 12 b) $\sqrt{61} = 7$ c) $(25)_{10} = (100)_b$ d) $(128)_{10} = (1003)_b$

   e) $(292)_{10} = (204)_b$

2. In each case, complete the code sequence to make it a Cyclic Code.

   a) 0000, 0001, 0011, 0010, 1010 b) 0000, 0100, 1100, 1000, 1010

   c) 0000, 0100, 1100, 1101 d) 0000, 0001, 0011, 0111

3. Represent the decimal number $(479)_{10}$ in the following codes.

   a) As a binary number b) In BCD 8 – 4 – 2 – 1 Code

   c) In BCD Excess-3 Code d) In 2 – 4 – 2 – 1 Code

4. a) Find the Gray Code number for the given 12-bit binary number.

   1 0 0 1, 1 0 1 0, 0 1 1 1

   b) What is the binary representation if the given number is in Gray Code?

5. Generate the weighted codes for the digits using the weights.

   a) 3, 3, 2, 1 b) 4, 4, 3, -2

   State whether they are unique and whether they have the property of self-complementation.

6. Construct a 7-bit single-error correcting code to represent the decimal digits by using Excess-3 Code words.

   a) Using even parity checks b) Using odd parity checks

7. a) Illustrate how a digital computer performs 9s/1s complement addition using the number ± 8 and ± 7. Let the word length for the decimal machine be n = 3 digits and for the binary machine be n = 5 bits.

   b) Repeat for 10s/2s complement addition.

8. a) i) Express decimal digits 0-9 in BCD Code and 2-4-2-1 Code.

   ii) Determine which of the above codes are self-complementing.

   b) i) Convert the decimal number 96 into binary and convert it to Gray Code number.

   ii) Convert the given Gray code number to binary: 1001001011.

9. a) Show the weights of three different 4-bit self-complementing codes whose only negative weight is – 4.

   b) Encode each of the ten decimal digits 0, 1, …, 9 by means of the weighted binary codes obtained in (a)

10. a) Generate 4-bit Gray Code directly using the mirror image property.

    b) Generate a Hamming Code for the given 11-bit message word 100,0111,0101 and rewrite the entire message in Hamming Code.

11. a) A person on Saturn possessing 18 fingers has a property worth $(1,00,000)_{18}$. He has three daughters and two sons. He wants to distribute half the money equally between his sons and the remaining half equally among his daughters. How much will each of his children get in Indian currency?

    b) An Indian started on an expedition to Saturn with Rs. 1,00,000. The expenditure on Saturn will be in the ratio of 1:2:7 for food, clothing and travelling. How much he will be spending on each item in the currency of Saturn.

12. Consider the following four codes.

Code A	Code B	Code C	Code D
0001	000	01011	000000
0010	001	01100	001111
0100	011	10010	110011
1000	010	10101	
	110		
	111		
	101		
	100		

    a) Which of the following properties is satisfied by each of the above codes?

    i) Detects single errors
    ii) Detects double errors
    iii) Detects triple errors
    iv) Corrects single errors
    v) Corrects double errors
    v) Corrects single and detects double errors

    b) How many words can be added to Code D without changing its error-detection and correction capabilities? Give a possible set of such words. Is this set unique?

13. a) Convert the number $(0.875)_{10}$ into the following forms.

    i) Binary ii) Ternary
    iii) Quinary iv) Octal using multiplication method

    b) Convert 3-bit Gray Code to 3-bit Binary Code.

14. a) Convert the following numbers.

    i) $(0.4375)_{10}$ to binary ii) $(11011.11)_2$ to decimal
    iii) $(FACE)_{16}$ to binary

    b) Construct a 7-bit error-correcting code to represent the decimal digits by augmenting the Excess-3 code and by using an odd-1 parity check.

15. a) Convert the following numbers.

    i) $(1431)_8$ to base 10 ii) $(53.1575)_{10}$ to base 2

    b) Determine which bit, if any, is in error in the Hamming Coded character 1100111. Decode the message.

16. a) i) Briefly explain error-detecting and error-correcting codes with examples.

ii) Find the base 2 equivalent of the following base 10 numbers

1) 0.00625 2) 43.32

b) The message below has been coded in Hamming Code and transmitted through a noisy channel. Decode the message assuming that at most a single error has occurred in each word code.

1001001, 0111001, 1110110, 0011011

17. Convert the following numbers.

i) $(1431)_8$ to base 10 ii) $(53.1575)_{10}$ to base 2

18. Perform the following calculations showing the complete procedure.

a) $(245679)_{10} = (?)_2$ b) $(10011010)_2 = (?)_{10}$

c) $(468.34375)_{10} = (?)_2$ d) $1011.01011 = (?)_{10}$

e) $(479)_{10} = (?)_{13}$ f) $101011 + 11011 = ?$

g) $110001 – 1110=?$ h) $(1101)_2 \times (110)_2 = ?$

i) $110010/101 = ?$ j) $(110111100)_2 = (?)_8$

19. The first expedition to Mars found only ruins of a civilisation. The explorers were able to translate a Martian equation as follows.

$$5x^2 - 50x + 125 = 0, \quad x = 5, 8$$

This was strange mathematics. The value of x = 5 seemed legitimate enough but x = 8 required some explanation. If the Martian number system developed in the same manner as on Earth, how many fingers would you say the Martians had?

20. What are weighted and non-weighted codes? Explain with examples.

21. a) Each of the following arithmetic operations is correct in at least one number system. Determine the possible bases of the numbers in each operation

i) $1234 + 5432 = 6666 \quad b \geq 7$ ii) $\sqrt{41} = 5$

iii) $302/20 = 12.1$

b) Encode the information character 01101110101 according to the 15-bit Hamming Code.

22. a) Solve for X.

i) $(1256)_8 = (X)_2$ ii) $(19.125)_{10} = (X)_8$

i) $(AC.2)_{16} = (X)_8$ iv) $(10011.11)_2=(X)_{16}$

v) $(33)_{10} = (201)_X$

b) The message 1001001 has been coded in Hamming Code for BCD and transmitted through a noisy channel. Decode the message assuming that at most a single error has occurred in the code word.

23. Find the single-error correcting Hamming Code for the message 0101. (Message should be in the form $p_1\ p_2\ m_1\ p_3\ m_2\ m_3\ m_4$, where p_1 indicates the parity bits and m_1 indicates the message bits)

24. Using Hamming Code, the received message is 0001110. Find errors, if any, and give the correct message.

2

Boolean Algebra

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Switching algebra.
- Two-variable Boolean algebra.
- Boolean expressions.
- Logic circuits.
- Gate circuits.
- Representation of expressions in different forms.
- Functionally complete sets of logic operators.
- Properties of NAND, NOR and EXCLUSIVE OR gates.

In this chapter, **switching algebra** is introduced as the basic mathematical tool essential in the analysis and synthesis of switching circuits. Switching circuits are also called **logic circuits, gate circuits,** and **digital circuits.** Switching algebra is another name for a **two-valued Boolean algebra.** Hence the terms **Boolean Expressions** and **Switching Expressions** are used to mean the same thing.

2.1 FUNDAMENTAL POSTULATES

The basic postulates of switching algebra state that every switching variable can assume any one of two distinct values 0 and 1, referred to as the **truth values.**

Switching algebra is an algebraic system consisting of the set of elements (0,1), two **binary operators** called OR and AND and one **unary operator** called NOT. A binary operation requires two operands or variables while the unary operation is performed on a single operand. The symbols and the definitions of these operations are given in Table 2.1(a) and 2.1(b).

Table 2.1(a) *Symbols for Switching or Logical Operators*

Operator	Symbols
OR	+, V, U (Union)
AND	•, ∧, ∩ (Intersection)
NOT	Prime, Bar on top of the literal, ~ (Tilda), ⌉ (Negation)

The symbol '•' for the AND operation is usually omitted. Thus xy means x • y. The negation symbol ⌉ precedes the literal. The binary operations are defined in Table 2.1(b).

Table 2.1(b) *Definitions of the Binary Operators*

OR Operation	AND Operation
0 + 0 = 0	0 • 0 = 0
0 + 1 = 1	0 • 1 = 0
1 + 0 = 1	1 • 0 = 0
1 + 1 = 1	1 • 1 = 1

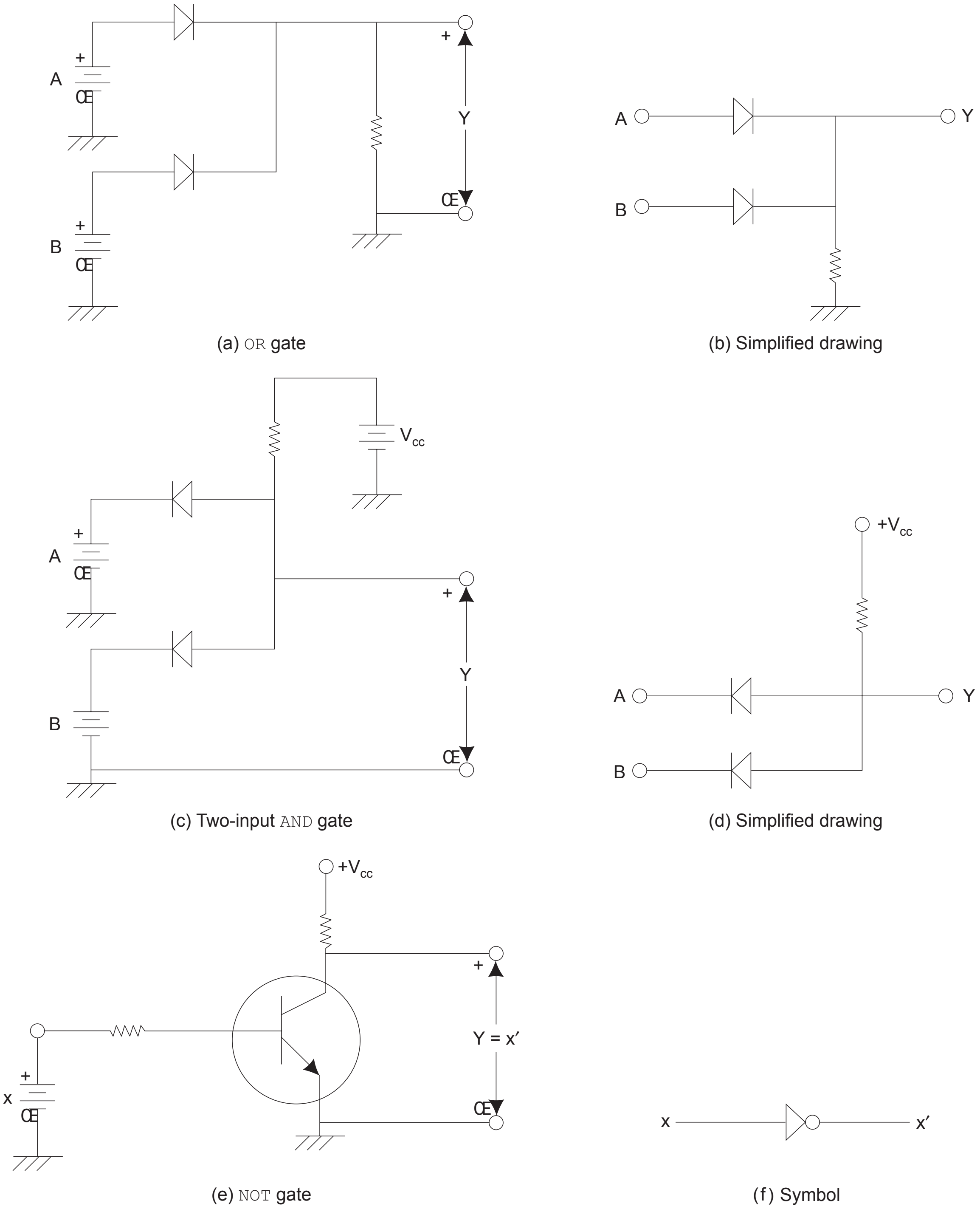
The OR operation is often called **logical sum** (or simply **sum**) or **union**. The AND operation is referred to as **logical product (multiplication)** or **intersection**. The truth values 1 and 0 are often referred to as True and False, Yes and No, High and Low, and so on. The reader should not confuse with the words 'sum' and 'product' used in conventional arithmetic. **Hierarchy** among the operators is as in the algebra of real numbers, that is, parentheses first, then AND and OR last.

The NOT operation, also known as **negation**, **inversion**, or **complementation**, is defined as follows. The complement of '0' is '1' and the complement of '1' is '0' symbolically expressed below.

$$0' = 1,\ 1' = 0$$

The above operators are often referred to as 'Gates'. As these simulate mental processes, they are also called logic circuits or logical operators.

Typical hardware realisation of these basic operators is shown in Fig. 2.1 using diodes, transistors, and resistors (DTL Logic). The main thrust in this book is on logic theory and design, not hardware involving transistors and semiconductor devices. V_{cc} is typically 5 volts representing the logic '1' level and normally, the common ground node corresponding to 0 volts represents the logic '0' level. In subsequent drawings, only single line drawings are used in which ground line is understood.



(a) OR gate

(b) Simplified drawing

(c) Two-input AND gate

(d) Simplified drawing

(e) NOT gate

(f) Symbol

Fig. 2.1 Basic gates

2.2 BASIC PROPERTIES [⊥]

1. Idempotency

$$x + x = x \tag{2.1a}$$

$$x \bullet x = x \tag{2.1b}$$

2. Commutativity

$$x + y = y + x \tag{2.2a}$$

$$x \bullet y = y \bullet x \tag{2.2b}$$

3. Associativity

$$(x + y) + z = x + (y + z) \tag{2.3a}$$

$$(x \bullet y) \bullet z = x \bullet (y \bullet z) \tag{2.3b}$$

4. Complementation

$$x + x' = 1 \tag{2.4a}$$

$$x \bullet x' = 0 \tag{2.4b}$$

5. Distributivity

$$x \bullet (y + z) = x \bullet y + x \bullet z \tag{2.5a}$$

$$x + y \bullet z = (x + y) \bullet (x + z) \tag{2.5b}$$

6. Identity elements—0 is the additive identity and 1 is the multiplicative identity

$$x + 0 = x \tag{2.6a}$$

$$x \bullet 1 = x \tag{2.6b}$$

$$x + 1 = 1 \tag{2.7a}$$

$$x \bullet 0 = 0 \tag{2.7b}$$

Observe in particular that the equations 2.1 relating to Idempotency and 2.5b relating to distributivity differ from the algebra of real numbers. According to Equation 2.5b Boolean algebra is distributive with respect to addition, which is not true in the algebra of real numbers. Boolean algebra is distributive, that is, multiplication distributes over addition and **addition distributes over multiplication in logical operations.**

In order to prove the above properties, 'perfect induction' is employed. 'Perfect induction' is a method of proof [Ψ] whereby a theorem or statement is verified for every possible combination of values that the variables may assume. Proof of Equation 2.5b by perfect induction is furnished in Table 2.2.

Tabulate all possible combinations of values that the three variables x, y and z can take. Since each of the variables can assume any one of the two possible values 0 and 1, it follows that there will be $2^3 = 8$ combinations as listed in Table 2.2. This is called the Truth Table. Treating each combination of values assumed by the variables as a binary number, the corresponding decimal number, also called code, is indicated in the first column of Table 2.2. The values of LHS and RHS expressions are then computed in the Truth Table and the identity is verified.

[⊥] also called Huntington postulates

Table 2.2 *Proof by Perfect Induction (Truth Table Method) for Equation 2.5b* $x + y \bullet z = (x + y) \bullet (x + z)$

Decimal Code	x	y	z	y • z	x + y	x + z	x + y •·z	(x + y) • (x + z)
0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0
2	0	1	0	0	1	0	0	0
3	0	1	1	1	1	1	1	1
4	1	0	0	0	1	1	1	1
5	1	0	1	0	1	1	1	1
6	1	1	0	0	1	1	1	1
7	1	1	1	1	1	1	1	1

2.3 DUALITY IN BOOLEAN ALGEBRA

Observe that all the properties of Section 2.2 are grouped in pairs. In any pair, one statement can be obtained from the other by merely replacing OR with AND, AND with OR, constants 0 by 1 and 1 by 0. Any two statements that have this property are said to be dual to each other. The implication of this concept is that once a theorem or statement is proved, its dual also thus stands proved. This is called the principle of duality, which follows from the symmetry of the postulates.

The dual of a general switching expression is symbolically defined by Equation 2.8 in which the suffix 'd' indicates the dual.

$$[f(x_1, x_2\ldots.. x_n, 0, 1, +, \bullet)]_d = f(x_1, x_2 \ldots x_n, 1, 0, \bullet , +)\ldots \qquad (2.8)$$

Example 2.1

$$F = A\,B + C$$

$$F_d = (A + B) \bullet C$$

Notice how the parentheses are used. AND is performed before OR in F. In order to ensure proper hierarchy of operations, parentheses will be required in F_d.

Example 2.2

$$F = (xy + abc) \bullet (a'x + bc') + PQ'R$$

$$F_d = ((x + y) \bullet (a + b + c) + (a' + x) \bullet (b + c')) \bullet (P + Q' + R)$$

Note in particular the way parentheses are used. Wherever AND is replaced by OR, it becomes necessary to use parentheses.

Example 2.3

$$F = P + Q \bullet R\,(A\,(B \bullet C + X') \bullet 1 + Y \bullet Z(D' \bullet E + 0))$$

$$F_d = P \bullet (Q + R + (A + (B + C) \bullet X' + 0) \bullet (Y + Z + (D' + E) \bullet 1))$$

Notice in F that there are four +s and eight •s including those implied by a left parenthesis following a variable. Hence there are four •s and eight +s in the dual indicated as F_d.

[Ψ] Only five different types of proof are generally accepted. They are (1) Proof by perfect induction, (2) Proof by logical deduction; (3) Proof by hypothesis; also called 'Reductio and Absurdum'; (4) Proof by constructing an example or a counter example or a procedure; and (5) Proof by mathematical induction. No other type of proof is acceptable, more so for a theorem. A given statement cannot be called a 'theorem' unless proof can be furnished on the above lines. Statements which are true but do not possess a formal proof may be called a 'principle', 'law', 'rule', 'conjecture', 'postulate', 'hypothesis' and so on but not a 'theorem'.

2.4 SIMPLIFICATION OF BOOLEAN EXPRESSIONS

The following properties and theorems are useful in the simplification and manipulation of switching expressions.

1. Absorption Law

$$x + xy = x \tag{2.9a}$$

$$x(x + y) = x \tag{2.9b}$$

Proof: $x + xy = x \bullet 1 + xy$

$= x(1 + y) = x \bullet 1 = x$

If a term appears in toto in another term, then the latter term becomes redundant and may be removed from the expression without changing its value. Removal of a term is equivalent to replacing the term by 0 if it is in a sum or by 1 if it is in a product.

2. Redundant Literal Rule

The following property is very often used. Let us for convenience, call it the Redundant Literal Rule (RLR).

$$x + x'y = x + y \tag{2.10a}$$

$$x(x' + y) = xy \tag{2.10b}$$

Proof: $x + x'y = (x + x')(x + y)$ by using distributivity

$= 1 \cdot (x + y)$

$= x + y$

Complement of a term appearing in another term is redundant.

3. Involution

$$(x')' = x \tag{2.11}$$

Notice that double complementation does not change the function.

4. Consensus Theorem

$$xy + x'z + yz = xy + x'z \tag{2.12a}$$

$$(x + y)(x' + z)(y + z) = (x + y)(x' + z) \tag{2.12b}$$

Proof: $xy + x'z + yz = xy + x'z + yz.1$

$= xy + x'z + yz(x + x')$

$= xy + x'z + xyz + x'yz$

$= xy(1 + z) + x'z(1 + y)$

$= xy + x'z$

Notice the manipulation using multiplication by $1 = x + x'$. If a sum of products comprises a term containing x and a term containing x′, and a third term containing the left-out literals of the fist two terms, then the third term is redundant, that is, the function remains the same with and without the third term removed or retained.

5. De Morgan's Theorems

$$(x + y)' = x' \bullet y' \tag{2.13a}$$

$$(x \bullet y)' = (x' + y') \tag{2.13b}$$

Proof: By perfect induction. Verify for all the four cases of

$$xy = 00, 01, 10, 11$$

These theorems lead to a general procedure to obtain the complement of a given function, stated next.

6. Complement of a Switching Expression

A generalisation of De Morgan's theorems of Equations (2.13) leads to the following rule

"The complement of a Boolean expression is obtained by replacing each OR with AND operator, each AND with OR, 1s with 0s, 0s with 1s and replacing each variable with its complement." This is symbolically expressed in Equation 2.14 where **f′** denotes complement of the function **f**.

$$f'(x_1, x_2 .. x_n, 0, 1, +, \bullet) = f(x_1', x_2' .. x_n', 1, 0, \bullet, +) \tag{2.14}$$

This can be proved by mathematical induction on the number of variables 'n' and remembering that the theorem is true for n = 2 and Boolean algebra obeys the property of associativity as in Equations 2.3(a) and (b).

7. Shannon's Expansion Theorem

This theorem states that any switching expression can be decomposed with respect to a variable x_i into two parts, one containing x_i and the other containing x_1'. This concept is useful in decomposing complex machines into an interconnection of smaller components.

$$f(x_1, x_2 \ldots x_n) = x_1 \bullet f(1, x_2 \ldots x_n) + x_1' f(0, x_2 \ldots x_n) \tag{2.15a}$$

$$f(x_1, x_2 \ldots x_n) = [x_1 + f(0, x_2 \ldots x_n) \bullet [x_1' + f(1, x_2 .. x_n) \tag{2.15b}$$

Proof: By perfect induction on x_1. For $x_1 = 0$, verify that LHS = RHS. Repeat for $x_1 = 1$

8. Relations between Complement and Dual

$$f'(x_1, x_2 .. x_n) = f_d(x_1', x_2' .. x_n') \tag{2.16}$$

$$f_d(x_1, x_2 .. x_n) = f'(x_1', x_2' .. x_n') \tag{2.17}$$

Equation 2.16 states that the complement of a function $f(x_1, x_2, \ldots, x_n)$ can be obtained by complementing all the variables (also called literals) in the dual function $f_d(x_1, x_2, \ldots, x_n)$. Likewise, Equation 2.17 states that the dual can be obtained by complementing all the literals in $f'(x_1, x_2, \ldots, x_n)$. These are easily deduced from Equations (2.8) and (2.14).

With this background, it is now possible to solve the examples of Chapter-1, by using Boolean algebra. Example 1.1 is reproduced here for reader's convenience.

Admission to the Sreenidhi Engineering College can be granted provided the applicant qualifies in the Engineering Entrance Test (EET) and fulfills any one or more of the following eligibility criteria

1. The applicant should be in the top 1000 rankers in EET, over 17 years old and has not been involved in an accident resulting in a handicap.
2. The applicant is over 17 years old and has been involved in an accident resulting in a handicap.

3. The applicant is over 17 years old and has secured a rank in EET 1000 or better, and has been involved in an accident resulting in handicap.
4. The applicant is a male with a rank in EET better than 1000 and is over 17 years old.
5. The applicant is a female with a rank in EET better than 1000, is over 17 years old and has been involved in an accident resulting in a handicap.

Let the variables A, B, C, D assume truth value ‘1’ in the following cases.

A = 1 if the applicant’s rank in EET ≤ 1000. If not, A = 0, that is, $A' = 1$.

B = 1 if the applicant is over 17 years old. If not, B = 0, that is, $B' = 1$.

C = 1 if the applicant was not involved in an accident resulting in handicap. If not, C = 0, that is, $C' = 1$.

D = 1 if the applicant is a male. If female D = 0 that is, $D' = 1$.

Finally let F assume a value ‘1’ if admission can be granted to the applicant; otherwise F becomes ‘0’. Obviously, F will be a function of the variables A, B, C, D. Further, the function F (A, B, C, D) is another binary variable of the algebra which can assume any one of the two values 0 and 1.

Each clause in the example can be represented by a conjunction (AND operation) of the variables. a disjunction (OR operation) of all the five terms thus obtained gives the function F as:

$$ABC + BC' + BAC' + DAB + D' ABC'$$

By rewriting the literals in the normal alphabetical order,

$$\begin{aligned} F(A, B, C, D) &= ABC + BC' + ABC' + ABD + ABC' D' \\ &= AB (C + C' + D + C' D') + BC' \\ &\text{by collecting the terms containing AB.} \\ &= AB \bullet 1 + BC' \\ &= AB + BC' \\ &= B (A + C') \end{aligned}$$

Thus, admission can be granted if the applicant is over 17 years old AND satisfies any one or both the conditions namely, the applicant is in the top 1000 rankers OR the applicant was involved in an accident resulting in a handicap. In simple words this would mean that the applicant should fulfil the age criteria and, in addition, he or she must possess a rank 1000 or better or else be considered under the category of physically challenged persons who enjoy relaxation of rank. The reader is now advised to solve Example 1.2.

In order to develop the skills of algebraic simplification of Boolean functions, some typical examples are presented below.

Example 2.4 Simplify the function

$$F_1(A, B, C) = (A + B) (A + BC') + A'B' + A'C'$$

Notice that $(A + B)' = A'B'$—De Morgan’s Theorem.

Now a apply the Redundant Literal Rule (RLR).

$$\begin{aligned} F_1 &= A + BC' + A'B' + A'C' \\ &= A + BC' + B' + C' \text{ by RLR} \\ &= A + C' + B' \text{ by RLR and Idempotency} \end{aligned}$$

Example 2.5 Using the Consensus Theorem prove the following identity with $F_2(A, B, C)$.

$$F_2(A, B, C) = AB' + BC' + CA' = A'B + B'C + C'A$$

Adding the redundant terms to LHS, we get

$$F_2 = AB' + BC' + CA' + B'C + AC' + A'B$$

Removing the first three redundant terms

$$F_2 = A'B + B'C + C'A = \text{RHS}$$

This is an example of $F(A, B, C) = F(A', B', C')$, that is, by complementing all the literals, the function remains the same.

Example 2.6 Simplify the function

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

After comparing the 2nd and 4th terms we may add the redundant term $x'z$ as the 5th term. Then comparing the 3rd and 5th terms, it is easily seen that the term $w'x'$ is redundant because of the Consensus Theorem. The working is shown below for clarity. By applying the Consensus Theorem to the 2nd and 4th terms and adding the redundant term we get

$$F = w'x' + x'y' + w'z' + yz + x'z$$

Now, comparing the 3rd and 5th terms and applying the same theorem we realise that $w'x'$ becomes redundant and hence may be removed. Then the term $x'z$ is still redundant and may be removed. Therefore, $F = x'y' + w'z' + yz$ is the simplified form.

Example 2.7 Using the Consensus Theorem, prove the identity given

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

Compare the terms containing A, A', B, B', C, C' and D, D'. Adding the redundant terms yielded on the application of Consensus Theorem to LHS we get

$$\begin{aligned}\text{LHS} = {} & (A + B')(B + C')(C + D')(D + A') \\ & (B' + D)(A + C')(B + D')(A' + C)\end{aligned}$$

Add further redundant terms by comparing the original set of four terms with the set of the redundant terms already introduced. We get four more redundant terms given below.

$$\begin{aligned}\text{LHS} = {} & (A + B')(B + C')(C + D')(A' + D) \\ & (B' + D)(A + C')(B + D')(A' + C) \\ & (B' + C)(C' + D)(A + D')(A' + B)\end{aligned}$$

Scanning the terms of the second set and third set we may remove the first set as redundant terms. Then by considering the terms of the 3rd set and applying the Consensus Theorem, we find that the 2nd set becomes redundant. The terms of the 3rd set are identical with the RHS of the given expression.

Hence LHS = RHS

The reader must have by now realised the difficulties in using algebraic manipulations for simplification of Boolean functions. In the next chapter, we will learn far more simpler and elegant methods for the purpose. Nevertheless, it is advantageous for the reader to develop the skills of algebraic manipulation before learning other methods.

2.5 BOOLEAN FUNCTIONS AND THEIR REPRESENTATION

A function of 'n' Boolean variables denoted by $f(x_1, x_2, \dots x_n)$ is another variable of the algebra and takes one of the two possible values, 0 and 1. The various ways of representing a given function are given in this section.

1. Sum-of-Products (SOP) Form This form is also called the disjunctive normal form (DNF). For example,

$$f(x_1, x_2, x_3) = x_1' x_2 + x_2' x_3 \tag{2.18}$$

2. Product-of-Sums (POS) Form This form is also called the conjunctive normal form (CNF). The function of Equation (2.18) may also be written in the form shown in Equation (2.19). By multiplying it out and using the Consensus Theorem, it is easily seen that it is the same function.

$$f(x_1, x_2, x_3) = (x_1' + x_2')(x_2 + x_3) \tag{2.19}$$

3. Truth Table Form One valid way of specifying a function is to list all possible combinations of values assumed by the variables and the corresponding values of the function. Then the function is said to be in the Truth Table form. Thus, Table (2.3) is the Truth Table form of the function.

Incidentally, observe that the column **f** in Table 2.3 can be filled in 2^8 ways and hence 2^8 different functions of 3 variables exist. In general with n variables there will be 2^{2^n} distinct functions of n variables.

Table 2.3 *Truth Table for $f(x_1, x_2, x_3) = x_1' x_2 + x_2' x_3$*

Decimal Code	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

4. Disjunctive Canonical Form (DCF) DCF is also referred to as the **standard sum of products (SSP) form**. It is also called the expanded sum of products. This is derived from the Truth Table by finding the sum of all the terms that correspond to those combinations (rows) for which 'f' assumes the value 1. Each term is a product of all the variables, with each variable appearing in either uncomplemented form or complemented form. A variable appears in an uncomplemented form in the product if its value is 1 in that combination and it appears in a complemented form if its value is 0.

For example, the product which corresponds to row 2 of Table 2.3 is $x_1' \bullet x_2 \bullet x_3'$. Thus, the function 'f' is given by

$$f(x_1, x_2, x_3) = x_1' \bullet x_2' \bullet x_3 + x_1' \bullet x_2 \bullet x_3' + x_1 \bullet x_2 \bullet x_3 + x_1 \bullet x_2' \bullet x_3 \tag{2.20}$$

A product term which contains all the n variables in either complemented or uncomplemented form is called a **'minterm'.** A minterm assumes the value 1 only for one combination of the variables.

There are 2^n minterms in all for an n-variable function. The sum of the minterms corresponding to those rows for which f equals 1, is the DCF (SSP) form of the function.

The minterms are often denoted as m_0, m_1, m_2 … where the suffixes are the decimal codes of the combinations. The minterm m_0 of Table 2.3 corresponds to the 0th row and is given by the product $x_1' \, x_2' \, x_3'$, m_1 corresponds to the 1st row and represents the product $x_1' \, x_2' \, x_3$, and so on. Another way of representing the function in DCF is by showing the sum of minterms for which the function equals 1.

Thus $$f(x_1, x_2, x_3) = m_1 + m_2 + m_3 + m_5 \tag{2.21}$$

Yet another form of representing the function in DCF is by listing the decimal codes of the minterms for which f = 1. Thus

$$f(x_1, x_2, x_3) = \sum (1, 2, 3, 5) \tag{2.22}$$

where $\sum$ represents the sum of all the minterms whose decimal codes are given in parentheses. The symbol 'V' is also used in place of sigma.

To sum up, the 1s of the function are called **true minterms** or simply **minterms**. The 0s are sometimes called **false minterms**.

An algebraic procedure to obtain the SSP form starting from the SOP form is illustrated below. It is based on suitably multiplying the terms of SOP of Equation 2.18 with terms like $x + x' = 1$ using the missing variables.

$$\begin{aligned} f(x_1, x_2, x_3) &= x_1' \, x_2 + x_2' \, x_3 \\ &= x_1' \, x_2 \, (x_3 + x_3') + (x_1 + x_1') \, x_2' \, x_3 \\ &= x_1' \, x_2 \, x_3 + x_1' \, x_2 \, x_3' + x_1 \, x_2' \, x_3 + x_1' \, x_2' \, x_3 \end{aligned}$$

This expression is the same as in Equation 2.20 but for the rearrangement of terms.

5. Conjunctive Canonical Form (CCF)

CCF is also referred to as the **standard product of sums (SPS)** form. This is derived by considering the combinations for which f = 0. Each term is a sum of all the variables. A variable appears in uncomplemented form, if it has a value of 0 in the combination and it appears in a complemented form if it has a value of 1. For example, the sum corresponding to row 4 is $(x_1' + x_2 + x_3)$. This means that wherever x_1' is 0, (same as x_1 is 1)

x_2 is 0 and x_3 is 0, this term becomes 0 and being a term in a product, the function assumes the value 0. Thus, the function f is given by the product of sums as

$$f(x_1, x_2, x_3) = (x_1 + x_2 + x_3)(x_1' + x_2 + x_3)$$
$$(x_1' + x_2' + x_3)(x_1' + x_2' + x_3') \qquad (2.23)$$

A sum term which contains each of the n variables in either complemented or uncomplemented form is called a **maxterm**. A maxterm assumes the value 0 only for one combination of the variables. For all other combinations it will be 1. There will be at the most 2^n maxterms. The product of maxterms corresponding to the rows for which f = 0, is the SPS form (CCF) of the function.

Just like minterms, maxterms also are often represented as M_0, M_1, M_2 … where the suffixes denote their decimal code. Thus the CCF of f may be written as

$$f(x_1, x_2, x_3) = M_0 \bullet M_4 \bullet M_6 \bullet M_7 \qquad (2.24)$$

or simply as

$$f(x_1, x_2, x_3) = \Pi\,(0, 4, 6, 7) \qquad (2.25)$$

where Π represents the product of all the maxterms whose decimal code is given within the parentheses. The symbol 'Λ' is also used in place of pi.

To sum up, the 0s of the function are called **maxterms** or **false minterms**.

The algebraic manipulation to get the SPS form starting from the SOP form is illustrated below. It is based on adding terms like $x \bullet x^1 = 0$ using the missing variables and remembering the distributive property. The POS form of the example function is given in Equation 2.19.

$$f(x_1, x_2, x_3) = (x_1' + x_2')(x_2 + x_3)$$
$$= (x_1' + x_2' + x_3 \bullet x_3')(x_1 \bullet x_1' + x_2 + x_3)$$
$$= (x_1' + x_2' + x_3')(x_1' + x_2' + x_3)(x_1' + x_2 + x_3)(x_1 + x_2 + x_3)$$

This expression is the same as in Equation 2.23 but for the rearrangement of terms.

6. Switching Cube An n-cube has 2^n vertices. Choose an axis for each variable, label each vertex in binary code, and then mark the vertices with either 'x' or '0' depending on whether f = 1 or f = 0. This representation is useful for functions of three variables. The function f is represented on the switching cube in Fig. 2.2.

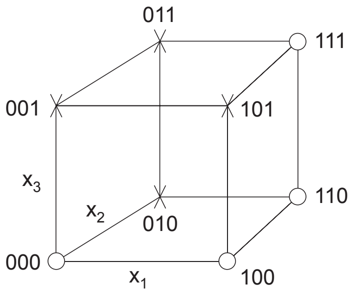


Fig. 2.2 Switching cube representation of $f = x' x_2 + x_2' x_3$

7. Venn Diagram Form As the algebra of sets and Boolean algebra are similar systems, Boolean expressions can be represented by a Venn diagram in which each variable is considered as a set, AND operation is considered as an intersection and OR operation is considered as a union. Complementation corresponds to outside of the set. Thus the function $f = x_1' \bullet x_2 + x_2' \bullet x_3$ is represented in Fig. 2.3.

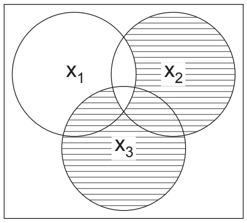


Fig. 2.3 Venn diagram representation of $f = x_1' \bullet x_2 + x_2' \bullet x_3$

8. Octal Designation Take a look at the Truth Table of the function, shown in Table 2.3 and the column in which the truth values of the function for each combination (minterm) are indicated. If these values are indicated in the order starting from m_7 at the extreme left and proceeding to m_0 at the extreme right**,** the resulting string of 0s and 1s is called the **'Characteristic Vector'** for the function. If written as an octal number, it becomes the octal designation for the function, which is illustrated below

m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0
0	0 ,	1	0	1 ,	1	1	0

Octal designation of $f(x_1, x_2, x_3) = (056)_8$

9. Karnaugh Map[Ψ] In this representation we put the Truth Table in a compact form by labelling the rows and columns of a map. This is extremely useful and extensively used in the minimisation of functions of 3, 4, 5 or 6 variables. The rows and columns are assigned a binary code such that two adjacent rows or columns differ in one bit only. Notice that the column on the extreme left is adjacent to the column on the extreme right. Likewise, the top row and bottom row are adjacent. Typical maps and their row and column designations are shown in Fig. 2.4. The numbers in the squares represent the decimal code of that cell. A function is represented by placing 1s in the cells corresponding to true minterms or by indicating 0s of the functions. Notice in particular how each map is drawn with some lines extended and the true regions indicated for each of the variables on the map itself. Notice also that the example function is mapped on a 3-variable map shown in Fig. 2.4(a) Annexed.

The 4-variable map is the same as the one used in Section 1.7 as Fig. 1.1, Fig. 2.4(a), (b), (c), and (d) are the maps commonly used for functions of 3, 4, 5, and 6 variables respectively. You will realise the elegance and usefulness of Karnaugh map in subsequent chapters.

When the number of variables exceeds 4, the Karnaugh Map takes the form akin to a 3-dimensional matrix. Five variable functions are represented on two 4-variable maps and for 6-variable functions, four such maps are used. For number of variables n exceeding six, it becomes incredibly cumbersome to use Karnaugh Maps.

[Ψ] It was Veitch who first used such maps. Karnaugh modified the map using adjacency ordering which improved its utility enormously

x_3 \ $x_1 x_2$	00	01	11 (x_1)	10 (x_1)
0	0	2	6	4
1	1	3 (x_2)	7 (x_2)	5

3-variable Karnaugh map

			x_1	x_1
		1		
x_3	1	1		1
		x_2	x_2	

Annexed map of
$f(x_1, x_2, x_3) = x_1^1 \bullet x_3 + x_2^1 \bullet x_3$

Fig. 2.4(a)

$x_3 x_4$ \ $x_1 x_2$	00	01	11 (x_1)	10 (x_1)	
00	0	4	12	8	
01	1	5	13	9	x_4
11 (x_3)	3	7	15	11	x_4
10 (x_3)	2	6	14	10	
		x_2	x_2		

Fig. 2.4(b) 4-variable Karnaugh map

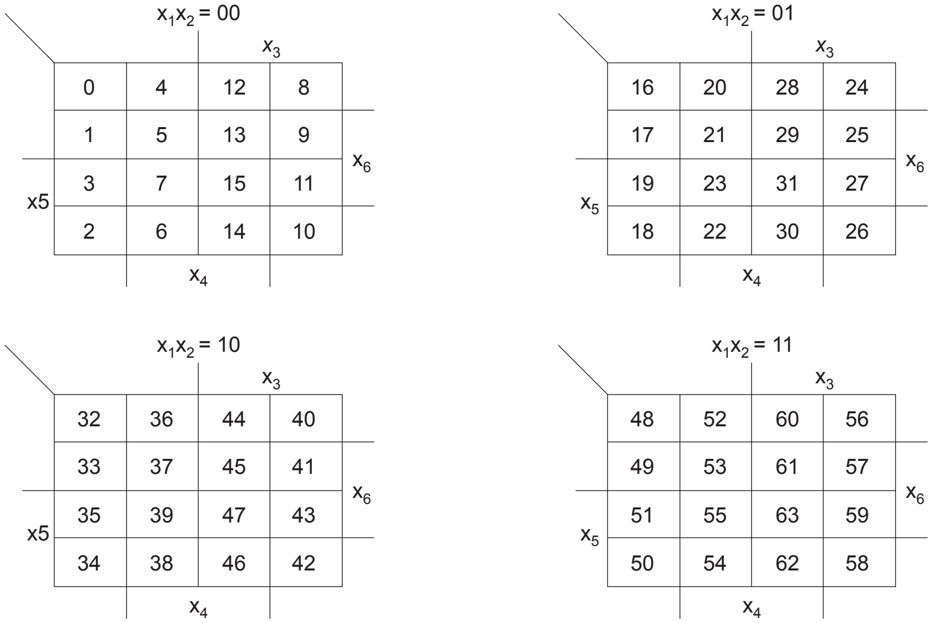
$x_1 = 0$

			x_2	x_2	
	0	4	12	8	
	1	5	13	9	x_5
x_4	3	7	15	11	x_5
x_4	2	6	14	10	
		x_3	x_3		

$x_1 = 1$

			x_2	x_2	
	16	20	28	24	
	17	21	29	25	x_5
x_4	19	23	31	27	x_5
x_4	18	22	30	26	
		x_3	x_3		

Fig. 2.4(c) 5-variable Karnaugh map



0	4	12	8
1	5	13	9
3	7	15	11
2	6	14	10

16	20	28	24
17	21	29	25
19	23	31	27
18	22	30	26

32	36	44	40
33	37	45	41
35	39	47	43
34	38	46	42

48	52	60	56
49	53	61	57
51	55	63	59
50	54	62	58

Fig. 2.4(d) 6-variable Karnaugh map

2.6 FUNCTIONALLY COMPLETE SETS OF OPERATIONS

The most common 2-variable functions are listed in Table 2.4. Recall that there will be in all (2^4 = 16 2-variable functions as we need to fill 2^2 = 4 rows in the Truth Table with 0 or 1 to define a function.

Table 2.4 *Common 2-Variable Functions*

A	B	A • B	A + B	(A • B)′	(A + B)′	A ⊕ B = A′B + AB′	A ⊙ B = A′B′ + AB
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

The function (A • B)′ is often written as A (NAND) B. ‘NAND’ is an abbreviation for ‘NOT AND’. It is also called the sheffer stroke function and denoted in this book by a slash as A/B——read as A slash B or A sheffer B.

The function (A + B)′ is often written as A (NOR) B. ‘NOR’ is the short form for ‘NOT OR’. It is also called the pierce arrow function and is written using a dagger symbol as A ↓ B—read as A pierce B.

The NAND and NOR functions are important and used extensively.

The function A ⊕ B is read as A EXCLUSIVE OR B. Notice that it has a value 1 when A ≠ B and a value of 0 when A = B. The complement of this function is called the **‘Equivalence Function’** and is denoted with a dot inside a circle as ⊙. It assumes logical value 1 when A = B and 0 when A ≠ B. In addition to these six commonly used functions, ten more functions of two variables exist including 0 (NULL), 1 (Tautology), A′, B′, A, B, A′B′, A′B, AB′, A′ + B′.

It is obvious that OR, AND and complement (+, •, prime) form a **functionally complete set** in the sense that any function can be realised using these operators in SOP or POS form. The question arises as to whether a smaller number of operators can perform this job and form a **functionally complete set**. Is it possible to dispense with one of the three basic operators? The answer is yes.

With the help of De Morgan’s Theorems, it is possible to produce A • B using only the set of operators (+, ~). Likewise A + B can be realised using only the operations (•, ~). These sets, each comprising two operators only (•, ~) or (+, ~) are said to be functionally complete sets. Further, using only the NAND operator, it is possible to produce all the Boolean operations. Hence NAND alone constitutes a functionally complete single element set. Therefore, it is said to be an **Universal Operator**. Likewise, NOR is another universal operator. EXCLUSIVE OR is not universal. The statements of this paragraph are implemented in algebraic and Gate Diagram forms in Table 2.5.

For the present, each operator is shown by a circle within which the operation is indicated. Notice, for instance, A + B is realised using 3 NAND gates as deduced below.

$$
\begin{aligned}
A + B &= \overline{\overline{A + B}} \text{ (By double complementation)}\\
&= \overline{\overline{A} \bullet \overline{B}} \text{ (By removing the inner bar)}\\
&= \overline{A} \; \text{NAND} \; \overline{B}\\
&= (A/A)/(B/B)
\end{aligned}
$$

Similar working holds good in other cases. The reader is advised to acquire a thorough knowledge of realisations shown in Table 2.5 and the associated algebraic working.

The implication of the concept of **universal operators** is as follows. The cost of a large digital system such as a computer will be less if the variety of logic gates is reduced. Experience has shown that the cost of a digital system depends on the variety and not so much on multiplicity of components. Most digital systems use only one type of gate–NAND–to produce any Boolean function and consequently secure economic advantages by mass producing one universal gate.

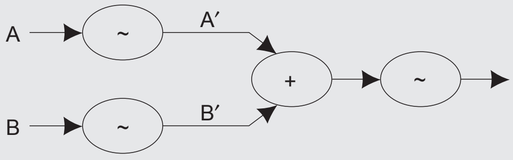
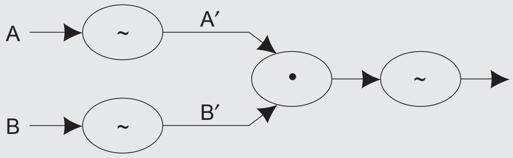
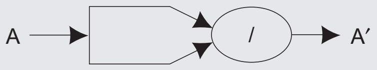
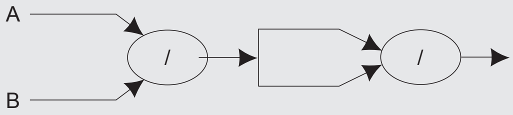
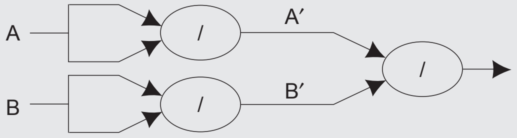
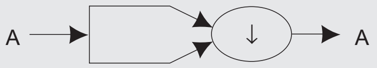
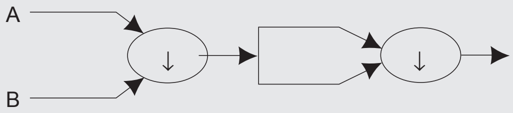
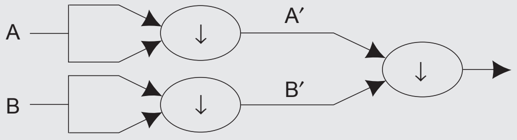
2.7 PROPERTIES OF EXCLUSIVE OR

Readers are advised to furnish proofs for the following identities involving the operator EXCLUSIVE OR **(sometimes called Modulo 2 adder)**.

1. $A \oplus 1 = A'$
2. $A \oplus 0 = A$
3. $A \oplus A = 0$
4. $A \oplus A' = 1$
5. $(AB) \oplus (AC) = A (B \oplus C)$
6. If $A \oplus B = C$, then $A \oplus C = B$

$$
\begin{aligned}
B \oplus C &= A\\
A \oplus B \oplus C &= 0
\end{aligned}
$$

Table 2.5 *Functionally Complete Sets of Operators*

Sl. No.	Functionally Complete Set	Algebraic Realisations of Other Operations	Gate Circuit Realisation
1.	{+, ~}	A • B = (A′ + B′)′	
2.	{•, ~}	A + B = (A′ • B′)′	
3.	NAND (/)	A/A = (A • A)′ = A′	
		$A \bullet B = \overline{\overline{A \bullet B}}$ = (A/B)/(A/B)	
		$A + B = \overline{\overline{A + B}}$ = A′ • B′ = (A/A)/(B/B)	
4.	NOR (↓)	A ↓ A = (A + A)′ = A′	
		$A + B = \overline{\overline{A + B}}$ = (A ↓ B) ↓ (A ↓ B)	
		$A \bullet B = \overline{\overline{A \bullet B}}$ = A′ + B′ = (A ↓ A) ↓ (B ↓ B)	

2.8 STANDARD SYMBOLS FOR LOGIC GATES

In this book, every logic gate is indicated, for convenience, by a circle within which the Boolean operators such as +, •, ~, ⊕, /, ↓ are indicated.

Standard symbols for different logic gates are indicated in Fig. 2.5. **Note that a small circle, sometimes called a bubble, indicates NOT operation.**

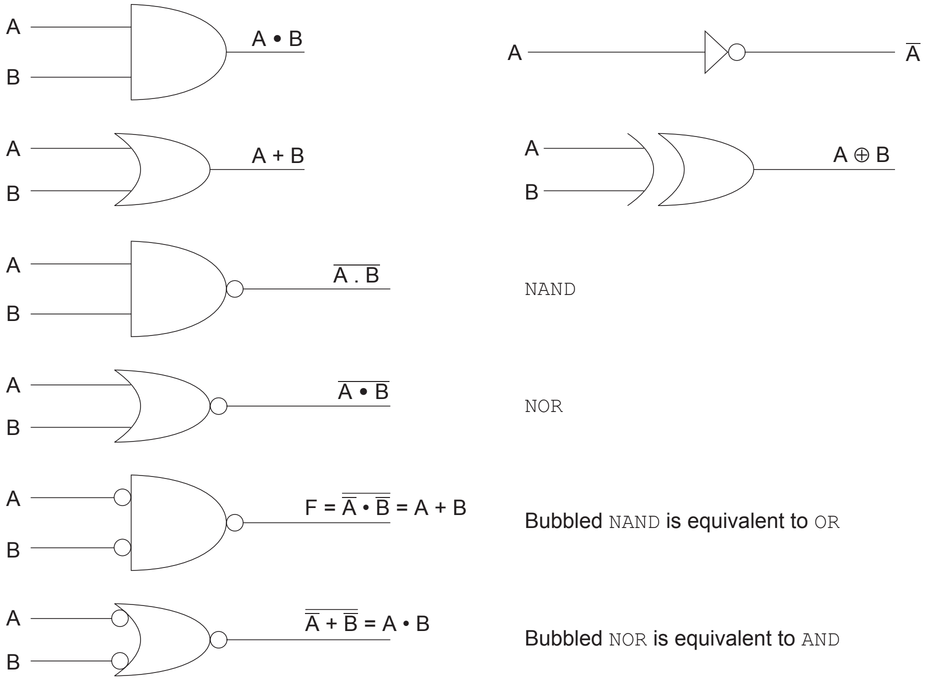
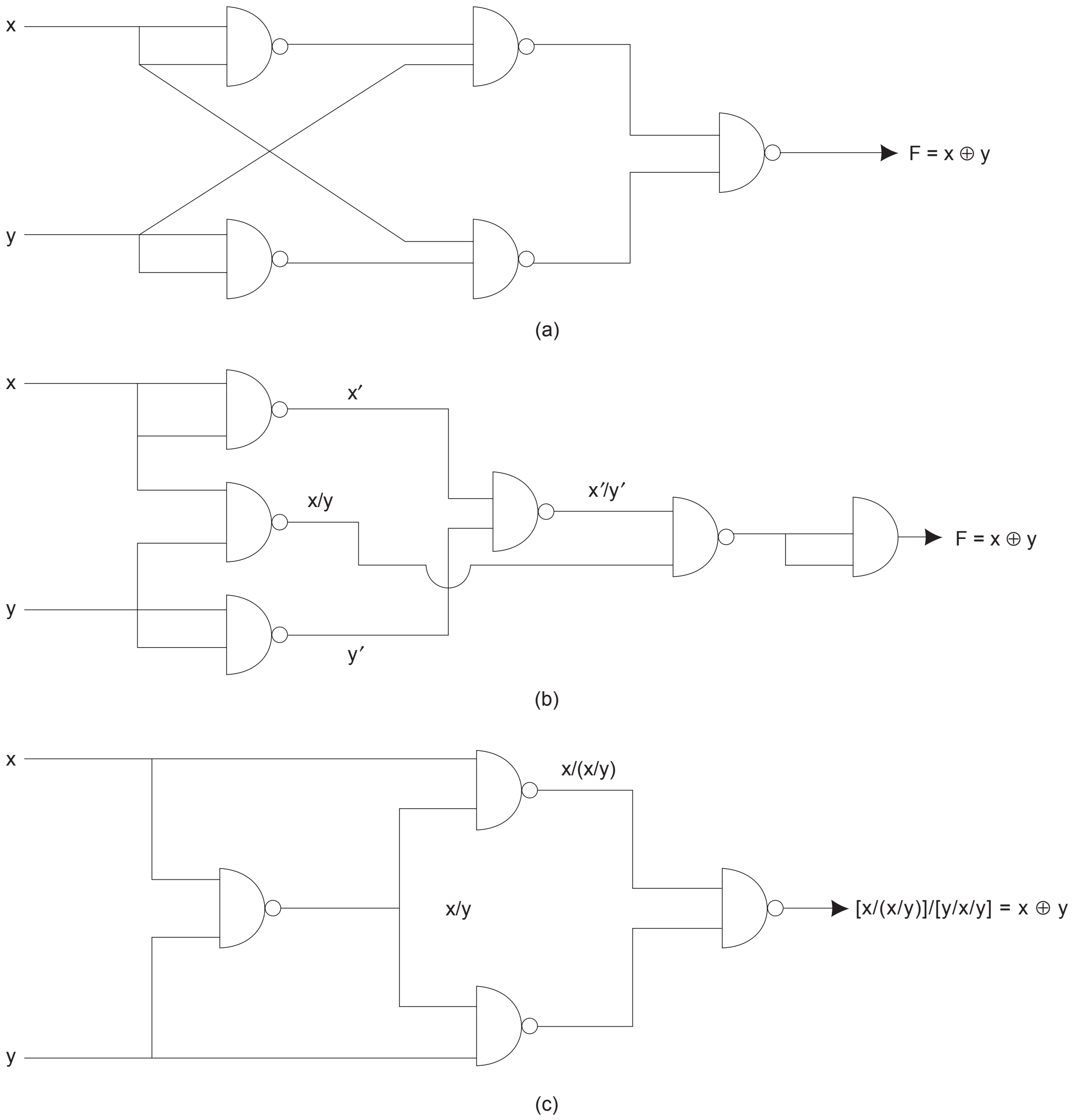

A
B
A • B
A
$\overline{A}$
A
B
A + B
A
B
A ⊕ B
A
B
$\overline{A . B}$
NAND
A
B
$\overline{A • B}$
NOR
A
B
F = $\overline{\overline{A} • \overline{B}}$ = A + B
Bubbled NAND is equivalent to OR
A
B
$\overline{\overline{A} + \overline{B}}$ = A • B
Bubbled NOR is equivalent to AND

Fig. 2.5 Standard symbols for logic gates

Notice that a NAND gate with its inputs complemented (bubbled) is equivalent to OR gate. Similarly a NOR gate with its inputs bubbled is equivalent to an AND gate.

2.9 REALISATION WITH UNIVERSAL GATES

In the preceding section, it was shown that NAND is a single element, functionally complete set. Any switching function can be realised using NAND gates alone. Likewise, NOR is also a universal gate. As a result of some technological features relating to fabrication and mass production, NAND has become more popular. Most digital systems use NAND gate as a building block. Even sequential circuits, which are covered in later chapters, make use of Flip-Flops which in turn, are built using NAND as the basic building block. In what follows, realisation of the EXCLUSIVE OR function using NAND gate is illustrated. The working and the resulting final realisation is presented for each version and shown in Fig. 2.6.



(a)

(b)

(c)

Fig. 2.6 NAND gate realisations of EXCLUSIVE OR

Multi Level Realisation of EXCLUSIVE OR with NAND Gates

For brevity, the symbol A/B (read as A slash B or A sheffer B) is used to indicate A NAND B. Both the prime and bar are used for complementation depending on the space available and clarity desired.

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

$$= x'y + xy' \quad \text{after double complementation}$$

Now, remove the inner bar by using De Morgan's Theorems. While performing this step, do not change the expressions, which are already in the NAND form of (A • B)′. The exercise should be to convert all other patterns to the NAND form. Thus, the next step in the working would be

$$\begin{aligned} F(x, y) &= \overline{\overline{x'y} \bullet \overline{xy'}} \\ &= \overline{x'y}/\overline{xy'} \\ &= (x'/y)/(x/y') \\ &= ((x/x)/y)/(x/(y/y) \end{aligned}$$

It is a five-NAND gate realisation that is shown in Fig. 2.6(a). Notice that it is a 3-level network, which means that the input signals have to pass through at the most three gates to reach the output level.

If the EXCLUSIVE OR function is expressed in product-of-sums form to start with, the working would be as given below towards NAND realisation.

$$\begin{aligned} F(x, y) &= (x + y)\,(x' + y') \\ &= \overline{\overline{(x + y)\,(x' + y')}} \quad \text{after double complementation} \\ &= \overline{x' \bullet y' + x \bullet y} \quad \text{after implementing the inner bar} \\ &= \overline{x' \bullet\ y'} \bullet \overline{x \bullet\ y} \\ &= (x'/y') \bullet (x/y) \end{aligned}$$

Using the identity A • B = (A/B)/(A/B) we get

$$\begin{aligned} F(x, y) &= [(x'/y')/(x/y)]/[(x'/y')/(x/y)] \\ &= [((x/x)/(y/y))/(x/y)]/[((x/x)/(y/y))/(x/y)] \end{aligned}$$

This leads to a six-NAND gate realisation, shown in Fig. 2.6 (b). This is a 4-level network.

A Clever Realisation of EXCLUSIVE OR with Four NAND Gates

The following working leads to a four-NAND gate realisation

$$\begin{aligned} F = x \oplus y &= (x + y)\,(\bar{x} + \bar{y}) \text{ in POS form.} \\ &= (x + y)\,\overline{xy} \\ &= x \bullet \overline{xy} + y \bullet \overline{xy} \end{aligned}$$

Double complement and remove the inner bar to get.

$$\begin{aligned} F &= \overline{\overline{(x \bullet \overline{xy})} \bullet \overline{(y \bullet \overline{xy})}} \\ &= (x/\overline{xy})/(y/\overline{xy}) \\ &= [x/(x/y)]/[y/(x/y)] \end{aligned}$$

Notice the manipulation involved in the working. It is evident that the number of gates depends on the ingenuity of the designer. In the past researchers used to struggle very hard to minimise the number of gates. Although this approach is basically sound, minimisation with the sole objective of reducing the cost, is no longer relevant today as the cost of a single gate is insignificant. Instead, minimisation of the number of gates plays a role in special situations involving fault diagnosability, testability and such other considerations.

2.10 PROCEDURE FOR MULTI LEVEL NAND REALISATION

1. Obtain the circuit using AND, OR, NOT gates. Assume that the variable and its complement are available for this purpose.
2. Replace each of the above gates by the equivalent NAND circuit.
3. Simplify the circuit by substituting short circuit for every pair of inverters in cascade, that is, in line. For Example,

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

The NAND gate realisation of this function is shown in Fig. 2.7. Observe in particular that gates 2 and 5 of Fig. 2.7(b) are replaced by a short circuit as required in step 3 above. So also is the case with gates 4 and 6 which are in line.

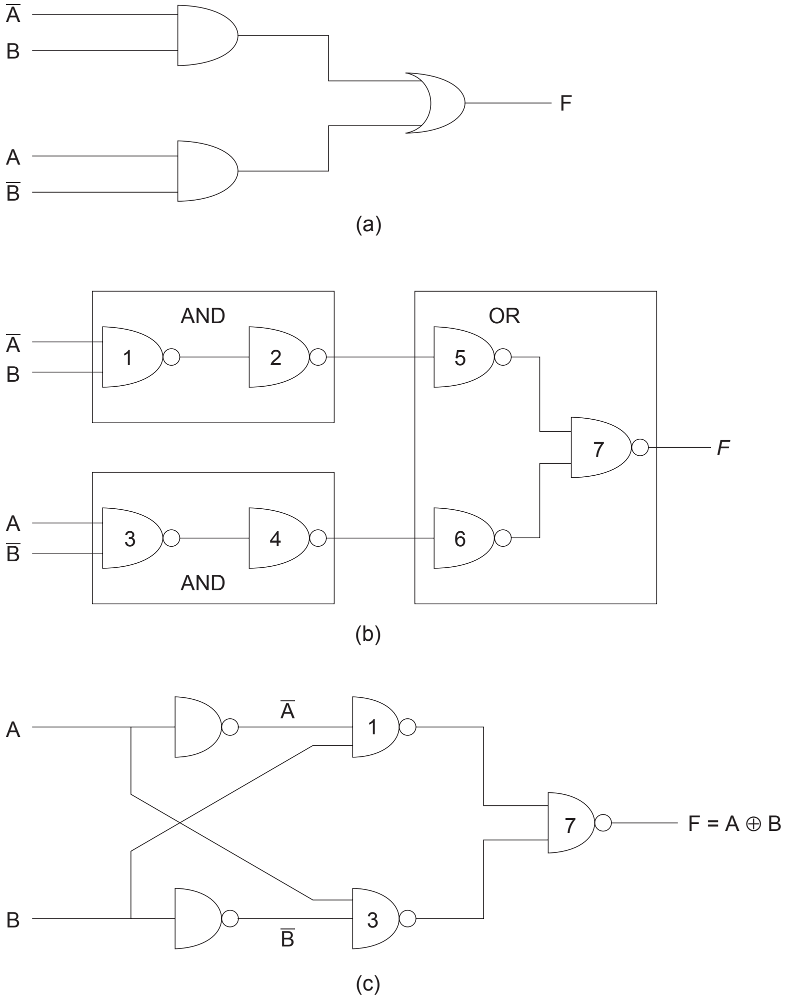


Fig. 2.7 Illustrating NAND gate realisations of F(A, B) = AB* + AB*

SUMMARY

Representation of switching functions in different forms has been illustrated. Truth table forms a vital link between the function and different ways of expressing the function. Venn diagrams and switching cubes are useful for three variable functions. Realization of function using universal operators is discussed in detail. Methods to arrive at the final form using only one type of universal gates has been illustrated. That ingenuinity of the designer is important in reducing the number of gates, has been emphasized.

KEY WORDS

- AND, OR, NOT, operators
- NAND gate, NOR gate
- Switching algebra
- SSP form
- SPS for
- MSP form
- MPS form
- Characteristic vector
- Distributivity of Boolean algebra
- Universal operators
- Equivalence
- Exclusive OR
- Binary operators
- Unary operator

REVIEW QUESTIONS

1. A combinational circuit can be designed using only
   a) AND gates
   b) OR gates
   c) AND and XOR gates
   d) NAND gates
2. Implement a two-input OR gate using NAND gates only.
3. Implement the function $f(A, B, C) = AC + BC'$ by using NAND operators only.
4. Realise XOR gate using only NOR gates. Show the working.
5. $wxy + x'z' + w'xz$. Find the complement of the given function.
6. Express the given function in product of sums form $F = x'z' + y'z' + xy'$
7. Simplify $A + AB + ABC + ABCD + ABCDE + \ldots$
8. What is the decimal code for the minterm of a four-variable function given by $w'x'yz$?
9. What is the logic which controls a staircase light associated with two switches A and B located at the bottom and top of the staircase respectively?
10. Find the complement of the functions.

$$F_1 = x'yz' + x'y'z$$

$$F_2 = x(y'z' + yz)$$

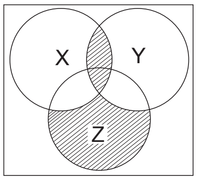
11. Realise Ex-OR gate using only NOR gates.
12. The Boolean expression for the shaded area in the Venn diagram is

a) $X' + Y' + Z$

b) $XY'Z + X'YZ$

c) $X'Y'Z + XY$

d) $X + Y + Z$



13. Find the complement of the following expression $(AB' + C)D' + E$.
14. Simplify $a + a'b + a'b'c + a'b'c'd \ldots$
15. What is the decimal code of the maxterm $(A' + B + C' + D)$?
16. $F(x, y, z) = \Sigma(0, 2, 3, 5, 6)$. Find the standard (canonical) product of sums for this function.
17. State the absorption laws.
18. Find the dual of $x + yz$.
19. $f(a, b, c) = \Sigma(1, 3, 6, 7)$. Write the standard sum of products.
20. Simplify $x' + y' + xyz'$.
21. $F = x \bullet y + x^1 z$. Convert this into standard SOP form.
22. Find the complement of $T(a, b, c) = a'b + ac'$.
23. State the Idempotency law.
24. In switching algebra, if $A + B = A + C \Longrightarrow B = C$. True/False?
25. Determine the canonical sum of products form for $T(x, y, z) = x'y + z' + xyz$.
26. Simplify the algebraic expression $x' + y' + xyz'$.
27. State the Consensus theorem.
28. Dual of $x + yz$ = __________.
29. What are the decimal codes of the minterm ABCD and maxterm $(A + B + C + D)$?
30. $F(x, y, z) = \Pi(0, 2, 4, 5)$. Represent the function in a sum of minterms.
31. $F = (AB + CD)(A'B' + C'D')$ is in

    a) SP form b) PS form c) SSP form d) SPS form

32. Express the Boolean function $F = xy + x'z$ as a product of maxterms.
33. The NAND can function as a NOT gate if

    a) inputs are connected together b) inputs are left open

    c) one input is set to 0 d) one input is set to 1

34. How many Boolean functions of 2 variables exist?

    a) 4 b) 8 c) 16 d) 5

35. How many of 4-variable functions are self-dual functions?

    a) 256 b) 8 c) 16 d) 4096

36. An XOR gate gives a high output

    a) if there are odd number of 1's b) if it has even number of 0's

    c) if the decimal value of digital word is even d) for odd decimal value

37. An exclusive NOR gate is logically equivalent to

a) Inverter followed by an XOR gate b) NOT gate followed by NOR gate
c) Exclusive OR gate followed by an Inverter d) Complement of a NOR gate

38. De Morgan's theorems state that

a) $\overline{A + B} = \overline{A} \bullet \overline{B}$ and $\overline{A \bullet B} = \overline{A} \bullet \overline{B}$ b) $\overline{A + B} = A + B$ and $\overline{A \bullet B} = \overline{A} \bullet \overline{B}$
c) $\overline{A + B} = \overline{A} \bullet \overline{B}$ and $\overline{A \bullet B} = \overline{A} + \overline{B}$ d) $\overline{A + B} = \overline{A} + \overline{B}$ and $\overline{A \bullet B} = \overline{A} \bullet \overline{B}$

39. The logic expression $(A + B)\,(A' + B')$ can be implemented by giving the inputs A and B to a two-input

a) NOR gate b) Exclusive NOR gate
c) Exclusive OR gate d) NAND gate

40. The logic expression $AB + A'B'$ can be implemented by giving inputs A and B to a two-input

a) NOR gate b) Exclusive NOR gate
c) Exclusive OR gate d) NAND gate

41. A switching function $f(A, B, C, D) = AB'CD + AB'C'D + A'BC'D + A'B'CD$ can also be written as

a) $\Sigma(1, 3, 5, 7, 9)$ b) $\Sigma(3, 5, 7, 9, 11)$ c) $\Sigma(3, 5, 9, 11)$ d) $\Sigma(5, 7, 9, 11, 13)$

42. If a three-variable switching function is expressed as the product of maxterms by $f(A, B, C) =$ II (0, 3, 5, 6), then it can also be expressed as the sum of minterms by

a) II(1, 2, 4, 7) b) $\Sigma(0, 3, 5, 6)$ c) $\Sigma(1, 2, 4, 7)$ d) $\Sigma(1, 2, 3, 7)$

43. Realise a three-input NAND gate using only two-input-NAND gates.

44. Which of the following is known as Mod-2 adder?

a) XOR gate b) XNOR gate c) NAND gate d) NOR gate

45. What is the minimum number of two-input NAND gates used to perform the function of two-input OR gate?

a) One b) Two c) Three d) Four

46. Which of the following gates are added to the inputs of the OR gate to convert it to a NAND gate?

a) NOT b) AND c) OR d) XOR

47. What logic function is produced by adding inverters to the inputs of an AND gate?

a) NAND b) NOR c) NAND d) OR

48. What logic function is produced by adding an inverter to each input and output of an AND gate?

a) NAND b) NOR c) XOR d) OR

49. Which of the following gates is known as a "coincidence detector"?

a) AND gate b) OR gate c) NOT gate d) NAND gate

50. An AND gate can be imagined as

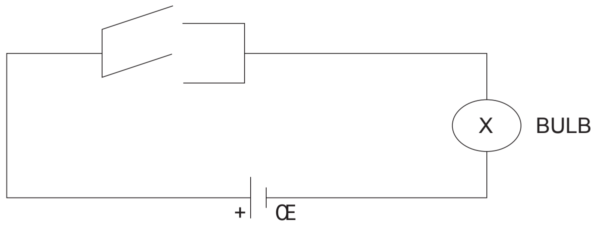
a) switches connected in series b) switches connected in parallel
c) transistors connected in series d) transistors in parallel

51. An OR gate can be imagined as

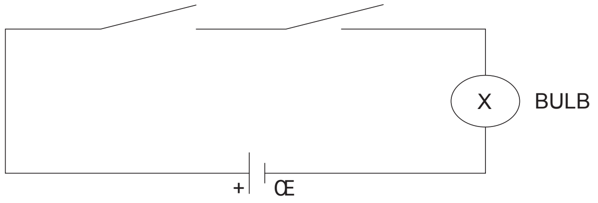
a) switches connected in series b) switches connected in parallel
c) MOS transistors connected in series d) transistors in parallel

52. Which of the following statements is correct?
    a) The output of a NOR gate is high if all its inputs are high.
    b) The output of a NOR gate is low if all its inputs are low.
    c) The output of a NOR gate is high if all its inputs are low.
    d) The output of a NOR gate is high if only one of its inputs is low.

53. Which logic is represented by the switching diagram shown in the figure below?
    a) AND b) OR c) NAND d) NOR



54. Which logic is represented by the circuit shown in the figure below?
    a) NAND b) NOR c) AND d) EQUIVALENCE



55. Which gate is known as the universal gate?
    a) NOT gate b) AND gate c) NAND gate d) XOR

56. How many NAND gates are needed to perform the logic function X • Y?
    a) 1 b) 2 c) 4 d) 3

57. Which of the following is a functionally complete set?
    a) AND, OR b) AND, XOR c) NOT, OR d) XOR, OR

58. Which of the following Boolean algebra expressions is incorrect?
    a) $A + A'B = A + B$ b) $A + AB = B$
    c) $(A + B)(A + C) = A + BC$ d) $(A + B')(A + B) = A$

59. The simplified form of the Boolean expression $(X + Y + XY)(X + Z)$ is
    a) $X + Y + Z$ b) $XY + YZ$ c) $X + YZ$ d) $XZ + Y$

60. The simplified form of the Boolean expression $(X + Y' + Z)(X + Y' + Z')(X + Y + Z)$ is
    a) $X'Y + Z'$ b) $X+Y'Z$ c) $X'Y + Z'$ d) $XY + Z'$

61. How many lines would the truth table for a four-input NOR gate contain to cover all possible input combinations?

a) 4 b) 8 c) 12 d) 16

PROBLEMS

1. Prove that if $w'x + yz' = 0$, then $wx + y'(w' + z') = wx + xz + x'z' + w'y'z$.
2. Obtain the simplified expression for $F(A, B, C, D) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9,12, 13, 14)$.
3. Obtain the expression for F′ in product of sums for

$$F(W, X, Y, Z) = \Pi(1, 3, 7, 11, 15) \quad d(W, X, Y, Z) = \Sigma(0, 2, 5)$$

4. Simplify by tabulation method. $F(W, X, Y, Z) = \Sigma(1, 3, 5, 7, 13, 15)$
5. Prove or give a counter-example

a) AB = BC ==> A = C b) A + B = A + C ==> B = C

6. Simplify the following switching functions stating the theorem at each step.

a) $F(a, b, c, d) = abcd + a(b.c) + abd'$

b) $T(A, B, C) = (A + B + C)(A + D + C')(A + B + D)$

c) $G(x, y, z) = xyz + xyz' + xy'z + xy'z' + x + z$

d) $H(x_1, x_2, x_3) = (x_1 + x_2)(x_1+x_2 x_3) + x_1^1 x_2^1 + x_1^1 x_3^1$

7. Let $F(x, y) = C_{00}x'y' + C_{01}x'y + C_{10}xy' + C_{11}xy$

where Cij = 0 or 1. Express F(x, y) in the following forms by determining k′s and R′s in terms of c′s′

a) $F(x, y) = K_{00} \oplus K_{01}y \oplus K_{10}x \oplus K_{11}xy$

b) $F(x, y) = R_{00}x'y' \oplus R_{01}x'y \oplus R_{10}xy' \oplus R_{11}xy$

8. Express the function $f(x_1, x_2, x_3) = x_1 + x_2^1 \bullet x_3$ in all the **nine** forms presented.
9. An irascible husband carried his new bride across the threshold and remarked, “we’ll get along fine, sweety, provided you observe the following rules”

a) If you serve cheese (C) at a meal, better serve bread (B) too.

b) Ice cream (I) is never served at a meal in my house unless you have included cheese in the menu.

c) I don’t like mango pickle (M) with bread during my meals.

Show with the help of Boolean algebra (not using a Venn diagram) that one of the things that the bride has to watch out for is not to serve mango pickles and ice cream at the same meal.

10. a) Minimize the following Boolean expressions to the required number of literals.

a) $BC + AC' + AB + BCD$ to four literals

b) $ABC + A'B'C + A'BC + A'B'C'$ to five literals

b) Obtain complement and dual for the given expression (AB+BC+AC)(EF)

11. Expand the given expression into canonical SOP form, then simplify.

$$F(A, B, C) = AC' + AB'C' + ABC + AB'C$$

12. a) Show that $(B + D)(A + D)(B + C)(A + C) = AB + CD$.

b) Obtain the Truth Table for the POS expression $F = (A + B + C)(A' + B + C')$

13. a) Simplify the logic function $f = AB + AC' + C + AD + AB'C + ABC$

    b) Simplify Boolean function $F = A'C + A'B + AB'C + BC$

14. Find the complement of the function $Y = (A + BC)(B + CA)$ in its POS form.

15. Realise $Y = D'B + D'C' + DB'$ using NAND gates.

16. a) Convert the following expression to a sum-of-products form. $(A + BC)(A'B' + A'B)$

    b) Minimise the following expression to the least number of literals. $BC' + A'B + BCD' + AB'C'D$

17. Simplify the following Boolean expression using Boolean theorems.

    $(A + B + C)(B + C) + (A + D)(A + C)$.

18. Verify the following by Boolean algebraic manipulation. Justify each step with reference to a postulate or theorem.

    a) $(X + Y' + XY)(X + Y')X'Y = 0$

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

19. Simplify using postulates and theorems of Boolean algebra.

    a) $(X + Y' + XY')(XY + X'Z + YZ)$

    b) $(A + B)(A' + C)(B + C)$

20. Simplify the following by manipulation of Boolean algebra.

    a) $(X + Y'X')[XZ + XZ'(Y + Y')]$

    b) $x + xyz + yzx' + wx + w'x + x'y$

21. Express the Boolean function $F = xy + x'z$ as a product of maxterms.

22. Find the complement of the functions $F_1 = x'yz' + x'y'z$

    $F_2 = x(y'z' + yz)$

23. Find the canonical PS for the functions

    a) $T(x, y, z) = x'y'z' + x'y'z + z'yz + xyz + xy'z + xy'z'$.

    b) Simply the expression

    $$T(A, B, C, D) = A'C' + ABD + BC'D + AB'D' + ABC + ACD'$$

    (Hint: Use Consensus Theorem)

24. Realise the function $F = A(B + CD) + BC'$ using minimum number of NAND gates only.

3

Minimisation of Switching Functions

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Algebraic minimisation using the theorems and rules of the algebra.
- Veitch–Karnaugh map method.
- Quine–McCluskey Tabular method.
- Covering tables and simplification rules.

3.1 INTRODUCTION

Minimising of switching functions is required in order to reduce the cost of circuitry (using contact or gate networks discussed later) according to some pre-specified cost criteria. This takes the form of minimising the number of terms in the Sum of Products (SOP) or Product of Sums (POS) form, minimising the number of literals in the expression or minimising the number of contacts or gates or gate inputs in the final logical circuit realisation. A literal is any variable appearing in the expression in complemented or uncomplemented form. This chapter presents the methods to obtain the **minimal sum of products (MSP)** form or the **minimal product of sums (MPS)** form of the given switching function.

There are essentially two methods widely used besides the algebraic method based on the simplification techniques discussed in Chapter II. These methods are **(a) the Veitch–Karnaugh Map method and (b) the Quine–McLuskey Tabular method.**

It was Veitch who first used maps for simplification of Boolean functions but it was Karnaugh who suggested adjacency ordering for columns and rows, which made the method far more elegant and powerful. Likewise, Quine used tables for listing minterms but McLuskey ordered them in accordance to their weights, which enormously reduced the number of comparisons.

The map method takes advantage of the human pattern recognition capabilities and is extremely useful for functions of not more than four variables. It becomes somewhat unwieldy as the number of variables increases beyond six. The tabular method is fully algorithmic and hence programmable. When the number of variables is large, the tabular method is invariably used even for hand computation.

3.2 THE VEITCH–KARNAUGH MAP METHOD

The basic philosophy of minimisation is to recognise terms of the type $x + x'$ and replace it by 1 and replace the terms of the type $x \bullet x'$ by 0. These situations are easily noticed on a K' map illustrated in this section. For example, consider

$$xy + xy' = x(y + y') = x \bullet 1 = x \tag{3.1a}$$

Dual expression is given by

$$(x + y)(x + y') = x + yy' \quad \text{using the distributivity property}$$
$$= x + 0 = x \tag{3.1b}$$

In Equation 3.1(a), observe that the two terms on the LHS contain a common factor *x* and the variable *y* appears in both complemented and uncomplemented forms. Regardless of whether y is 0 or 1, the function remains the same. Hence the expression becomes independent of y. On a two-variable map (see Fig. 3.1a) the 1s of the function correspond to the cells xy = 11 and 10 which are adjacent and bunched together and marked by an ellipse. The bunch (also called a cluster) of 1s of the function shows that the function assumes a logical value of 1 when x is 1. We may also show the bunch of 0s (not marked in the map) and interpret it as the function assuming a value 0 when x is 0. This is a dual approach, which will become clearer in due course.

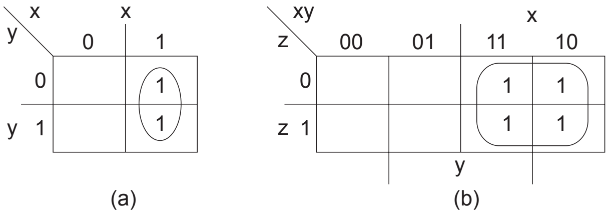


Fig. 3.1 Illustrating use of a a K–map

1. $F(x, y) = xy + xy' = x$
2. $F(x, y, z) = xy'z' + xy'z + xyz + xyz' = x$

Note: Clusters of adjacent 1s have to be marked. The number of 1s in a cluster must be an integral power of 2, that is, 2, 4, 8, 16 and so on. The bigger the bunch, the simpler would be the expression.

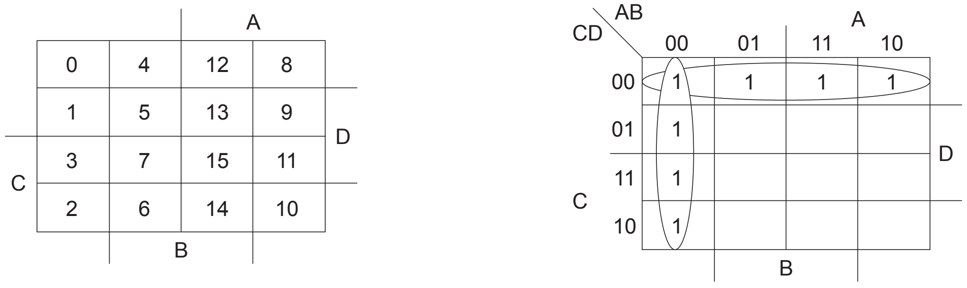
Now, consider a 3-variable function given by

$$F(x, y, z) = xy'z' + xy'z + xyz + xyz'$$

If we take out *x* as a common factor, the function reduces to only *x* as the remaining variables *y* and *z* appear in all the possible four forms. Take a look at the 3-variable K–map of Fig. 3.1(b). Notice specially that *xy* are labelled as columns and *z* as rows. Note in particular that xy = 11 follows 01 and 10 follows 11. This is called 'adjacency ordering', innovatively adopted by Karnaugh in contrast with the normal ordering of 00, 01, 10, 11 used by Veitch. On the map, the four minterms of the function form one bunch marked with an ellipse or a rectangle, referred to as a **subcube**. A subcube of two 1s makes the corresponding function independent of one variable; a subcube of four 1s makes the corresponding function independent of two variables, and so on. The number of 1s in a subcube is an integral power of 2, that is, 2, 4, 8, 16, …, 2^n. A subcube of 2^n 1s represents a function independent of n variables. The sum of all the 16 minterms of a 4-variable function is 1 and independent of all the 4-variables.

Figs. 3.2 and 3.3 show typical bunching of 1s in 4-variable maps. Notice that the true regions of each variable are marked on the body of the map by extending appropriate lines. For the convenience of the reader, Fig. 1.1 giving the decimal designation of the cells, discussed in Section 1.8, is reproduced as Fig. 3.2(a) for ready reference. In Fig. 3.2(b), a function F(A, B, C, D) having seven 1s is mapped and the corresponding Boolean expression is given below. Notice that two clusters of adjacent 1s are marked with ellipses. Recall that a minterm can be covered any number of times without changing the function. This follows from the property of Idempotency that x + x = x and x • x = x.

$$
\begin{aligned}
F(A, B, C, D) &= \Sigma(0, 1, 2, 3, 4, 8, 12) \\
&= A'B'(C'D' + C'D + CD + CD') + C'D'(A'B' + A'B + AB + AB') \\
&= A'B' + C'D'
\end{aligned}
$$



(a) Decimal designation of cells

(b) Illustration minimisation using K' map

Fig. 3.2

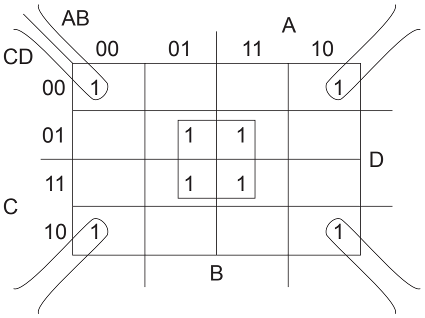


Fig. 3.3 Illustrating minimisation of a 4-variable function
G(A, B, C, D) = Σ(0, 2, 5, 7, 8, 10, 13, 15)

The bunch of four adjacent minterms 0, 1, 3, 2 marked by an ellipse in Fig. 3.2(b) sum up to A′B′ which means that F is 1 when A is 0 and B is 0, that is, in the column AB = 00, which is equivalent to saying A′ = 1 and B′ = 1. This result is easily seen from the K–map. Likewise, the other cluster of true minterms 0, 4, 12, 8 convey the information that the function F must be 1 when C′ is 1 and D′ is 1 which means the row labelled as CD = 00. To arrive at the terms of the MSP form, the question one should ask oneself is which variables should be 1 to make the function 1.

Consider now, another function G(A, B, C, D) with eight minterms mapped in Fig. 3.3 and given below.

$$G(A, B, C, D) = \Sigma(0, 2, 5, 7, 8, 10, 13, 15)$$

Collecting the group of adjacent minterms (0, 2, 10, 8) at one place and another group (5, 7, 15, 13) at another place, we may write the Boolean function as given below. Notice in particular that the four corners of the map form a cluster of adjacent 1s.

$$\begin{aligned} G &= m_0 + m_2 + m_{10} + m_8 + m_5 + m_7 + m_{15} + m_{13} \\ &= A'B'C'D' + A'B'CD' + AB'CD' + AB'C'D' \\ &\quad + A'BC'D + A'BCD + ABCD + ABC'D \end{aligned}$$

Notice that B′D′ is a common factor in the first four terms and BD is a common factor in the remaining four terms. The function is therefore reduced to the following form.

$$\begin{aligned} G = {} & B'D'\,(A'C' + A'C + AC + AC') \\ & + BD\,(A'C' + A'C + AC + AC') \end{aligned}$$

The terms in parentheses add up to logic 1.

Hence, G = B′D′ + BD

This result can as well be written down by observing the pattern of the clusters of 1s and noting which variables assume the logic value 1 in the region and which variables keep changing.

3.3 MINIMAL SUM OF PRODUCTS (MSP) FORM

It is clear that the minimisation process involves the selection of the largest subcubes in such a way that all the 1s of the function are covered. **Each of these subcubes is called a prime implicant (PI)**. The objective now reduces to finding the minimal set of PIs which covers all the minterms of the function.

Definition

$f(x_1 \ldots x_n)$ INCLUDES $g(x_1 \ldots x_n)$ if

$g(x_1 \ldots x_n) = 1 \Rightarrow f(x_1 \ldots x_n) = 1$ where the symbol '⇒' means 'implies'.

It follows that each true minterm of a function is an **'implicant'** of the function.

Definition

i A prime implicant of a function ***f*** is a product of literals.

ii The function ***f*** includes this product.

iii If any literal is removed from the product, ***f*** does not include the resultant product.

It is possible that some of the prime implicants of a function may be redundant. For example,

$$F_1(A, B, C, D) = \Sigma(1, 5, 6, 7, 11, 12, 13, 15) \text{ mapped in Fig. 3.4.}$$

In Fig. 3.4 instead of showing all the row headings, each variable is marked in the region of the map where it assumes the value 1. For the function F_1(A, B, C, D) mapped in Fig. 3.4, the prime implicants ABC′, ACD, A′BC and A′C′D are all **Essential Prime Implicants (EPI)** since each of them contains at least one '1' which cannot be covered by any other prime implicant. The Prime Implicant BD, marked by the dotted ellipse, is obviously a **Redundant Prime Implicant (RPI)** since each '1' of this PI is covered by at least one EPI. The function mapped in Fig. 3.4 has a unique MSP comprising EPIs only given by

$$F_1(A, B, C, D) = ABC' + ACD + A'BC + A'C'D$$

The RPI 'BD' may be included without changing the function but the resulting expression would not be in minimal sum of products (MSP) form.

A PI which is neither an EPI nor an RPI is called a **Selective Prime Implicant (SPI).** For example,

$$F_2(A, B, C, D) = \Sigma(0, 1, 5, 7, 10, 14, 15).$$

Look at the function F_2(A, B, C, D) mapped in Fig. 3.5. SPIs are marked by dotted ellipses. This shows that the MSP form of a function need not be unique. For the function mapped in Fig. 3.5, the MSP form is obtained by including two EPIs and selecting a set of SPIs to cover the remaining uncovered minterms 5, 7, 15. These can be covered in the following ways.

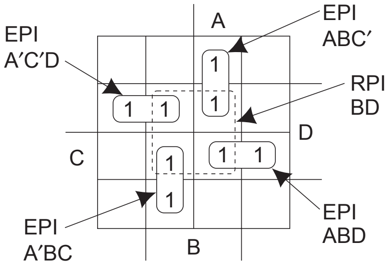


Fig. 3.4 Essential and redundant prime implicants

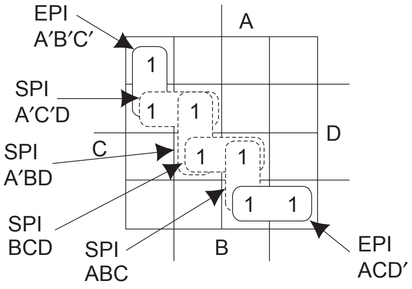


Fig. 3.5 Selective prime implicants

i) (1, 5) and (7, 15) ….. A′C′D + BCD

ii) (5, 7) and (7, 15) ….. A′BD + BCD

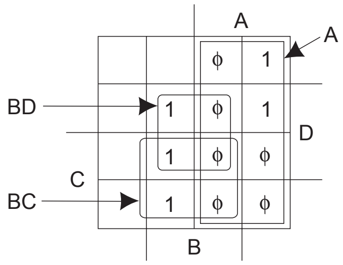
iii) (5, 7) and (14, 15) ….. A′BD + ABC

$$
\begin{aligned}
F_2(A, B, C, D) = {} & A'B'C' + ACD' \text{ ….. EPI's} \\
+ {} & (A'C'D + BCD) \\
\text{OR} \quad & (A'BD + BCD) \\
\text{OR} \quad & (A'BD + ABC)
\end{aligned}
$$

Thus this function has three different MSP forms.

3.4 DON'T CARE COMBINATIONS

Often we come across situations where the value of the function is not specified for certain combinations of variables. This arises due to some interdependence among the variables. In systems dealing with very high voltages, opening the door of any cubicle switches off the power supply by an interlocked arrangement. The combination of 'door open' AND 'power supply on' will never occur. In decade counters using four binaries, the combinations corresponding to 10, 11, 12, 13, 14, 15 will never occur. Thus, certain combinations of variables may never occur; even if they occur, the value the function assumes for those combinations does not matter. Such combinations are called **'Don't care combinations'** or **'Optional combinations'**. The ϕs (or dash) in Fig. 3.6 show such combinations. While minimising, we may consider them as 1s or 0s whichever gives a simpler expression for the function. PIs are marked on the map of Fig. 3.6 and the corresponding function has a unique MSP form given by A + BC + BD. Notice that all the don't care cells are considered as 1s as the realisation leads to a simpler Boolean function with few literals.



F(A, B, C, D) = $\Sigma_V(5 - 9) + \Sigma_\phi(10 - 15)$ Where 'v' denotes the sum of minterms and
= A + BC + BD ϕ denotes don't care combinations.

Fig. 3.6 Don't care (optional) combinations

3.5 MINIMAL PRODUCT OF SUMS (MPS) FORM

Hitherto, the sum of products (SOP) form was being used for minimising the given functions. We used the 1s (true minterms) of the function and obtained the MSP form by finding a minimal set of PIs, which are essentially products. Each product is written down by posing a question on which variables should be '1' to make the function 1.

Instead, we may as well use the 0s (False minterms, also called maxterms) of the function. This is a dual approach. We deal with the product of sums (POS) form and obtain a minimal set of false prime implicants (clusters of 0s) to represent the function. In order to get a feel for the procedures involved, let us solve the same examples with this dual approach. Consider the function mapped in Fig. 3.2(b) which is shown again in Fig. 3.7 with its 0s mapped and algebraically expressed below.

$$
\begin{aligned}
F(A, B, C, D) &= \Sigma(0\text{-}4, 8, 12) \\
&= \Pi(5, 6, 7, 9, 10, 11, 13, 14, 15) \\
&= (A' + D')\,(A' + C')\,(B' + C') + (B' + D')
\end{aligned}
$$

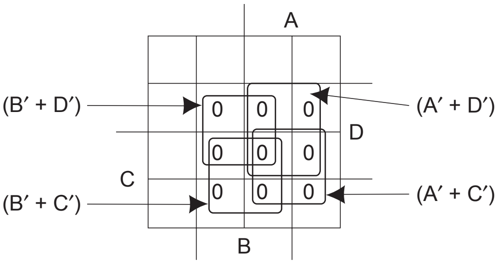


Fig. 3.7 Illustrating minimisation using 0s of the function
F(A, B, C, D) = Π(5-7, 9-11, 13-15)

Notice that each of the largest bunch (cluster) of 0s is a false prime implicant (FPI). For example, consider the bunch of zeros (9, 11, 13, 15) marked by ellipse, labelled as (A′ + D′). If you ask "Which variables should be zero to make the function zero?" the answer is A′ and D′ because A and D are at '1' and the other variables B and C are changing in this region. Hence the cluster represents the **FPI,** which is a sum of literals given by $(\overline{A} + \overline{D})$. This function has four such FPIs.

The problem now is to select a minimal set of FPIs to cover all the false minterms, that is, 0s of the function. In this example, all the FPIs are essential as each of them contains at least one zero which cannot be covered by any other FPI. Wherever clarity is required we use the terms 'True PI' and 'False PI'. If only P1 were used, it would mean True PI. Notice that the MSP form given in Section 3.2 contains four literals and the MPS form now contains eight literals but they represent the same function and one form can be derived from the other.

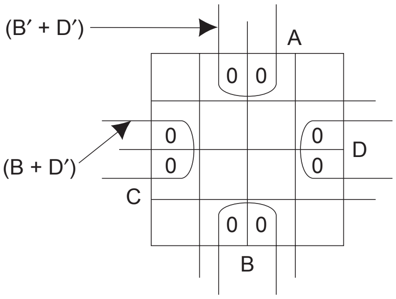


Fig. 3.8 MPS form of

G(A, B, C, D) = Π(1, 3, 4, 6, 9, 11, 12, 14)

For another example, let us now turn to Fig. 3.3 in which the 1s of a Boolean function G (A, B, C, D) are mapped and MSP form derived. The 0s of the same function are now mapped in Fig. 3.8 and the false prime implicants marked. This function has two FPIs and both of them are essential to cover all the 0s of the function. The MPS form is thus given by

$$\begin{aligned} G(A, B, C, D) &= \Sigma(0, 2, 5, 7, 8, 10, 13, 15) \\ &= \Pi(1, 3, 4, 6, 9, 11, 12, 14) \\ &= (B + D')\,(B' + D) \end{aligned}$$

Notice that the MSP form obtained earlier in Section 3.2 and the present MPS form contain the same number of literals—four. Remember, they represent the same function and one form can be derived from the other.

In order to consolidate the concepts, let us now focus attention on functions F_1(A, B, C, D) and F_2(A, B, C, D) for which the 1s are mapped in Fig. 3.4 and Fig. 3.5. These functions are shown once again in Fig. 3.9 and Fig. 3.10, respectively. This time, however only the 0s are mapped to illustrate the dual approach of finding the MPS form.

$$\begin{aligned} F_1(A, B, C, D) &= \Sigma(1, 5\text{-}7, 11\text{-}13, 15) \\ &= \Pi(0, 2–4, 8–10, 14) \end{aligned}$$

Take a look at Fig. 3.9. Notice that the four FPIs, (A + C + D), (A′ + B + C), (A + B + C′), (A′ + C′ + D), are all essential FPIs as each of them contains at least one zero which cannot be covered by any other FPI. The four corner zeroes form the largest cluster of adjacent zeroes, which is an FPI whose zeroes are covered by

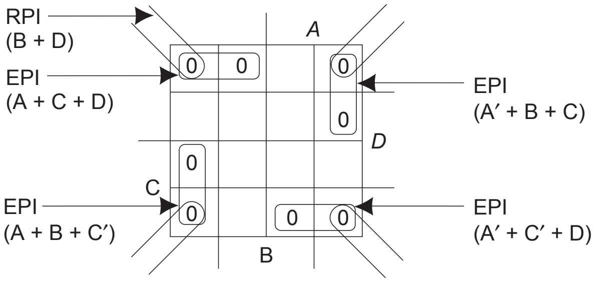


Fig. 3.9 MPS form of F_1(A, B, C, D)

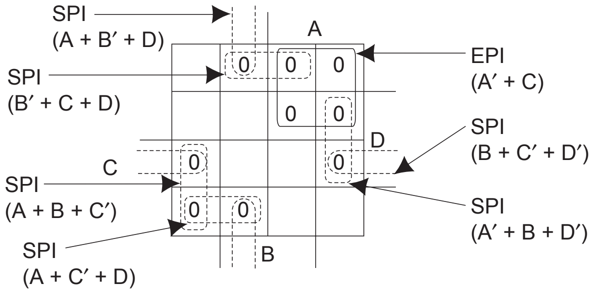


Fig. 3.10 MPS form of F_2(A, B, C, D)

essential PIs and hence is a redundant false prime implicant. Now compare with the MSP form discussed in Section 3.3. Each of them contains 12 literals.

Hence the MPS is given by

$$F_1(A, B, C, D) = (A + C + D)(A' + B + C)(A + B + C')(A' + C' + D)$$

Focus attention on Fig. 3.10 wherein the 0s of $F_2(A, B, C, D)$ of Fig. 3.5 are mapped. The function is expressed as

$$F_2(A, B, C, D) = \Sigma(0, 1, 5, 7, 10, 14, 15)$$
$$= \Pi(2, 3, 4, 6, 8, 9, 11, 12, 13)$$

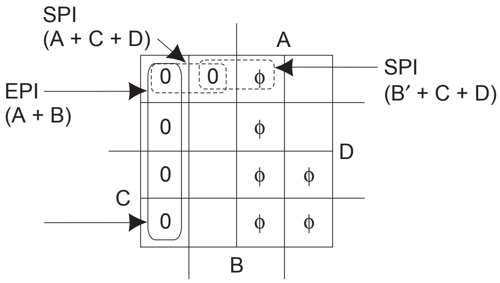
Note that the function has in all seven FPIs marked on the body of Fig. 3.10. The FPI $(A' + C)$ is an essential FPI as it contains 0s at locations 8 and 13, which cannot be covered by any other FPI. The remaining six FPIs are all SPIs. As the EPI covers the 0s at locations 8, 9, 12, 13, we must now select a minimal set of SPIs to cover the remaining five 0s at locations 2, 3, 4, 6, 11. The answer is not unique. One possible solution is given below comprising EPI and three SPIs.

$$F_2(A, B, C, D) = (A' + C)(A + B + C')(A + B' + D)(B + C' + D').$$

3.6 MPS FORM FOR FUNCTIONS WITH OPTIONAL COMBINATIONS

Consider the function for which 1s and ϕs are mapped in Fig. 3.6. The same function is shown again in Fig. 3.11 with its 0s and ϕs mapped and the FPIs marked with ellipses.

This function has two possible MPS forms as it has one EPI which covers the 0s at cells 0, 1, 2, 3. To cover the remaining 0 in cell 4, we need to have one of the SPIs. If you select the SPI $(A + C + D)$, then all the ϕs would become 1s. If, however, you select the SPI $(B' + C + D)$ then ϕ in cell 12 above would become a zero of the function in which case the MSP and MPS forms will not be equivalent. Hence it is clear that for functions containing optional entries, MSP and MPS forms may or may not be equivalent.



$$F(A, B, C, D) = \Pi_{\wedge}(0\text{-}4) + \Pi_{\phi}(10\text{-}15)$$
$$= (A + B)(A + C + D)$$
OR
$$= (A + B)(B' + C + D)$$

Where ‘∧’ denotes false minterms and ϕ denotes don’t care entries.

Fig. 3.11 MPS for a function with don’t care entries

3.7 QUINE–McCLUSKEY TABULAR METHOD OF MINIMISATION

The tabular method makes use of the binary representation of the minterms. An uncomplemented variable is shown by 1 in the corresponding position and a complemented variable by 0. For example, the minterm wxy′z is denoted by 1101 and wxyz is represented as 1111. Observe that these two binary forms differ only in one place. Hence they combine to form a term wxz independent of y, which is represented in binary form as 11-1. The missing literal is represented by a dash.

The number of 1s in the binary form of a minterm is called its weight. **Two minterms may combine only if their weights differ by one.** For instance, m_1 of weight 1 and m_5 of weight 2 combine as they differ only in one-bit position and hence are adjacent. In contrast, m_1 of weight 1 and m_{10} of weight 2 **do not** combine as 0001 and 1010 differ in three places and hence are not adjacent. Thus, weights differing by one, is a necessary but not a sufficient condition for two minterms to combine. The first step is to arrange all the true minterms (and also don't care combinations, if any) into groups such that each minterm of a group has the same number of 1s (weight) and then put down the groups of minterms in ascending order of weight.

Look at Table T_0 of the example worked out below. Every minterm of weight i (starting from i = 0 and going upto i = n – 1) is now compared with every minterm of weight i + 1 to form combinations. Whenever two minterms combine, the resulting binary form containing a dash is noted in a separate table and the two minterms are checked in the present table to indicate that they are included in the subsequent table. Minterms which do not combine with any other minterms are prime implicants and are assigned some labels and set apart for later consideration. They are dropped from further consideration at this stage. We will then have a list of terms, each with a single dash. The same process is repeated to get a list of terms with two dashes, three dashes, and so on until no further combinations are possible. **Two terms with dashes in different positions cannot combine.** This gives a great facility in comparison of terms. The working is shown in Tables T_0, T_1, T_2 for the example given below.

Example: Minimise the function

$$f(w, x, y, z) = \Sigma(0, 1, 5, 7, 8, 10, 13) + \Sigma_\phi(2, 9, 14, 15)$$

In Table T_0, first compare m_0 of weight 0 with m_1, m_2, and m_8 of weight 1. Between m_0 = 0000 and m_1 = 0001, only one variable-Z-differs. These two minterms (0, 1) combine to give a 3-variable term w′x′y′

Table T_0

Weight	Decimal Code	w	X	y	z	
0	0	0	0	0	0	√
1	1	0	0	0	1	√
	2	0	0	1	0	√
	8	1	0	0	0	√
2	5	0	1	0	1	√
	9	1	0	0	1	√
	10	1	0	1	0	√
3	7	0	1	1	1	√
	13	1	1	0	1	√
	14	1	1	1	0	√
4	15	1	1	1	1	√

Table T_1

Combined minterms	w	X	Y	z	
0, 1	0	0	0	–	√
0, 2	0	0	–	0	√
0, 8	–	0	0	0	√
1, 5	0	–	0	1	√
1, 9	–	0	0	1	√
2, 10	–	0	1	0	√
8, 9	1	0	0	–	√
8, 10	1	0	–	0	√
5, 7	0	1	–	1	√
5, 13	–	1	0	1	√
9, 13	1	–	0	1	√
10, 14	1	–	1	0	PI : P
7, 15	–	1	1	1	√
13, 15	1	1	–	1	√
14, 15	1	1	1	–	PI : Q

Table T_2

Combined minterms	W	X	y	z	PI
0, 1, 8, 9	–	0	0	–	A
0, 2, 8, 10	–	0	–	0	B
1, 5, 9, 13	–	–	0	1	C
5, 7, 13, 15	–	1	–	1	D

which is shown as 000-in the next table T_1 and the minterms m_0 and m_1 are checked. Similarly, the combinations (0, 2) and (0, 8), each with one dash in place of y and w respectively, are entered in T_1 with the corresponding minterms checked. Notice in particular that the place positional values for w, x, y, z are respectively 8, 4, 2 and 1. **Whenever two minterms combine, the difference in the decimal designations must be a power of 2 namely 1, 2, 4, 8, and so on.** The dash needs to be marked in the place corresponding to the difference.

Now, compare the minterms of weight 1 with the minterms of weight 2. Each of the minterms m_1, m_2 or m_8 is to be compared with every minterm m_5, m_9, and $m_{10.}$ This yields the combinations (1, 5), (1, 9) which are noted in T_1. Notice that m_1 does not combine with m_{10} as the difference in the decimal codes is 9, which is not a power of 2. Repeat the process with m_2, which combines with m_{10} only giving a dash in place 8, that is, w. Continue the process. T_1 will contain only four groups as against five groups of minterms in T_0 to start with. Notice that all the rows in T_1 will have a dash in one place. Note that all the minterms have been checked and there is no prime implicant so far.

Continue the process with T_1 by comparing the terms of one group with those of the next group and enter the combined terms in T_2. **Terms with dashes in different places do not combine;** this property makes the comparison much easier. You will find that each row in T_2 will have two dashes, which means that the corresponding term is independent of two variables. The process naturally ceases when no more combinations are possible. In this example, T_2 is the terminal step. The rows, which are not checked, are put in focus as they constitute the PIs of the given function. These PIs are labelled as A, B, C, D, P, Q. Remember that A = x′y′ corresponding to the term – 0 0 -. Similarly, B = x′z′, C = y′z, D = xz, P = wyz′ and Q = wxy.

3.8 PRIME IMPLICANT CHART

Having obtained a list of all the prime implicants, we now form the **prime implicant chart** as in Table 3.1. The row labels are the PIs and the column headings are the true minterms which are to be covered. The Don't care combinations need not be covered. The minterms covered by a PI are marked with a cross in the corresponding row. **We need to find a minimal set of PIs which covers all the true minterms.** The following rule helps in simplifying the PI Chart.

Difinition of domination of row (column): A row (column) is said to dominate another row (column) if for every cross in the latter, there is a cross in the same column (row) of the former.

Rule Dominated rows and dominating columns may be eliminated to simplify the PI chart.

Now look at the PI chart of Table 3.1. Row B dominates row P. The dominated row P is removed from the table as all the minterms covered by P can be covered by B and some more. Removal is indicated by a circle. Row Q does not cover any minterms; in fact, it is dominated by every other row. After removing the dominated rows P and Q, examine column dominations. Columns 0 and 8 dominate column 10. This means that the PI which covers minterm 10 has to necessarily cover minterms 0 and 8 too. Hence the dominated column is important and has to be retained. The dominating columns may be ignored as the corresponding minterms are covered by the PI which covers the minterm corresponding to the dominated column. Hence, columns 0, 8 are

Table 3.1 *Prime Implicant Chart*

	True Minterms						
PI	0	1	5	7	8	10	13
A	x	x			x		
B	x				x	x	
C		x	x				x
D			x	x			x
P						x	
Q							

Table 3.2 *Simplified PI Chart*

Tabular Method of Minimising Switching Functions

	1	7	10
A	x		
B			x
C	x		
D		x	

deleted from the table, marked by circling. For a similar reason, columns 5 and 13 are deleted as they dominate column 7. Table 3.2 shows the simplified PI Chart.

Looking at the rows of the simplified PI chart, we find that rows A and C dominate each other. Hence we need to retain only one of them. Normally, the dominated row can be ignored for the simple reason that the PI which covers all the minterms of a dominating row will automatically cover its subsets (dominated row's). The PIs B and D are essential as B alone covers the minterm 10 and D alone covers minterm 7. To cover the remaining minterm m_1, one of the PIs, either A or C, would suffice. If both are included, the function does not change, but it would not be a minimal realisation. Thus there are two solutions given by the sets of prime implicants (A, B, D) or (B, C, D). The MSP form is given by

$$f(w, x, y, z) = (x'y' + x'z' + xz) \text{ or } (xvz' + y'z + xz)$$

The reader is advised to verify using the map method already learnt. Selection of the minimal set of PIs covering all the true minterms may also be obtained by an algebraic procedure illustrated next.

3.9 ALGEBRAIC METHOD OF FINDING MINIMAL SETS OF PRIME IMPLICANTS

In this method, the labels given to the PIs are treated as Boolean variables and a function is written in SOP form such that there is a term for each column of the PI covering table. For the example, under discussion, the expression is given by

$$M = (A + C)(D)(B) = A\,B\,D + B\,C\,D$$

This means there are two minimal sets of PIs. Choose the term in M, which has the least number of literals, each of which indicates a prime implicant.

3.10 TABULAR METHOD USING DECIMAL CODES

The reader must have realised that in the tabular method, it is rather cumbersome, tedious and time-consuming to write the minterms in binary form using strings of 0s and 1s. Instead, it is quicker and elegant to work with decimal codes directly, without wasting time and energy in listing the strings of 0s and 1s. This is illustrated next using a function of five variables. For example,

$$F(A, B, C, D, E) = \Sigma_V(0, 2, 5, 8, 14, 16, 18, 25, 26, 30) + \Sigma_\phi(10, 12, 24)$$

First, the true minterms together with optional minterms are listed in the order of their weights 0, 1, 2, 3 or 4 in T0. Notice that only decimal designations are used, not the strings of 0s and 1s to save time and tedium. Two minterms combine if the difference is an integral power of 2 namely, 1, 2, 4, 8 or 16. Such pairs of minterms are listed in Table T1, with the difference noted in parentheses. The figure in parentheses indicates the position of dash, that is, the missing variable. Note in particular that the minterm 5 did not combine with any other minterm and hence constitutes a prime implicant given by $A'B'CD'E$ corresponding to 0 0 1 0 1. This has been marked as the prime implicant labelled P.

Now, compare the pairs in Table T1 from one group with those of the next group. Combinations are possible only if the figures in parentheses are the same. This gives a great facility in comparison. While comparing the pair 16, 18 (2) with the pair 12, 14 (2) of the next group, notice that there is, in fact, **no need to compare with lower numbered minterms.** The former pair denotes 100 – 0 and the latter pair stands for 011 – 0, which are not adjacent. Notice, finally, that all the pairs have combined with others except the pair indicated by 24, 25 (1) which is marked as another PI labelled ‘Q’. This stands for the term 1 1 0 0 - which is expressed as ABC′D′.

Table T2 contains quadruplets. The process of comparison resulted in repeated terms, which are noted for the benefit of the reader. All such repeated terms are to be ignored.

Table T0

Weight		
0	0 √	
1	2 √	
	8 √	
	16 √	
2	5	PI:P
	10 √	
	12 √	
	18 √	
	24 √	
3	14 √	
	25 √	
	26 √	
4	30 √	

Table T1

Dec. Codes (Diff.)	
0, 2 (2) √	
0, 8 (8) √	
0,16 (16) √	
2, 10 (8) √	
2, 18 (16) √	
8, 10 (2) √	
8, 12 (4) √	
8, 24 (16) √	
16, 18 (2) √	
16, 24 (8) √	
10, 14 (4) √	
10, 26 (16) √	
12,14 (2) √	
18, 26 (8) √	
24, 25 (1)	PI:Q
24, 26 (2) √	
14, 30 (16) √	
26, 30 (4) √	

Table T2

Dec. Codes (Diff.)	
0, 2, 8, 10 (2, 8) √	
0, 2, 16, 18 (2, 16) √	
0, 8, 2, 10 (8, 2)	Repeated
0, 8, 16, 24 (8, 16) √	
0, 16, 2, 18 (16, 2)	Repeated
0, 16, 8, 24 (16, 8)	Repeated
2, 10, 18, 26 (8, 16) √	
2, 18, 10, 26 (16, 8)	Repeated
8, 10, 12, 14 (2, 4)	PI : R
8, 10, 24, 26 (2, 16) √	
8, 12, 10, 14 (4, 2)	Repeated
8, 24, 10, 26 (16, 2)	Repeated
16, 18, 24, 26 (2, 8) √	
16, 24, 18, 26 (8, 2)	Repeated
10, 14, 26, 30 (4, 16)	PI : S
10, 26, 14, 30 (16, 4)	Repeated

Table T3

Dec. Codes (Diff.)	
0, 2, 8, 10, 16, 18, 24, 26 (2, 8, 16)	PI : T
0, 2, 16, 18, 8, 10, 24, 26 (2,16, 8)	Repeated
0, 8, 16, 24, 2, 10, 18, 26 (8,16, 2)	Repeated

In T2, notice that the term 8, 10, 12, 14 (2, 4) does not combine and hence is marked as PI and labelled R. This stands for the term 0 1 – – 0, which represents the product A′BE′. Likewise, the term 10, 14, 26, 30 (4, 16) is marked as PI : S. This denotes the term – 1 – 1 0 which gives the product BDE′. Continuing the process of comparisons in T2 yields T3 which has only one term comprising of eight minterms 0, 2, 8, 10, 16, 18, 24, 26 (2, 8, 16) which obviously is a prime implicant labelled as T. The PI T has to be represented by – – 0 – 0 with three dashes corresponding to the places 2, 8, 16 and 0s in the remaining places as it includes the minterm m_0. Thus, we have now five PIs labelled P, Q, R, S and T. We need to find a minimal set of PIs which covers all the true minterms. Now, look at the Prime Implicant chart given in Table 3.3.

Table 3.3 *Prime Implicant Chart*

Minterms				X	X				X	
PI	0	2	5	8	14	16	18	25	26	30
P			√							
Q								√		
R				√	√					
S					√				√	√
T	√	√		√		√	√		√	

Simplified PI Chart

Minterms PI	0	2	5	16	18	25	30
P			√				
Q						√	
R							
S							√
T	√	√		√	√		

Using the column domination rule, notice column 8 dominates column 0; columns 14 and 26 dominate column 30. Hence columns 8, 14, 26 are removed, shown by a cross. In the simplified chart, notice that row R does not cover any minterms at all which means that it is a redundant PI. It is easy to see that all the remaining PIs happen to be essential PIs. Hence the MSP form of the function is given by

$$\begin{aligned} F(A, B, C, D, E) = {} & P + Q + S + T = 5 + 24, 25\ (1) + 10, 14, 26, 30\ (4, 16) + \\ & 0, 2, 8, 10, 16, 18\ 24, 26\ (2, 8, 16) \\ = {} & 0\,0\,1\,0\,1 + 1\,1\,0\,0 - + - 1 - 1\,0 + - - 0 - 0 \\ = {} & A'B'CD'E + ABC'D' + BDE' + C'E' \end{aligned}$$

The reader is advised to verify the working by using the map method illustrated in Fig. 3.12.

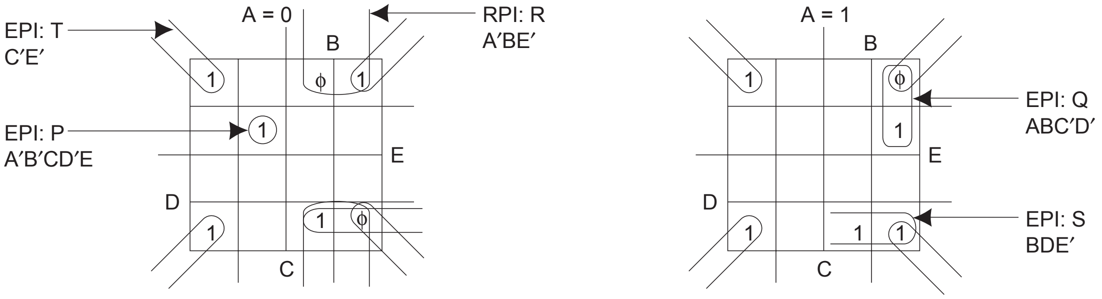


Fig. 3.12 Verification by K–map

SUMMARY

Various techniques of minimising switching functions, namely, the algebraic method, Karnaugh map method and Tabular method are illustrated. While the map method uses the human ability to recognise patterns which may fail at times, the Tabular method is algorithmic and computerised. The Map method is used for hand computation for functions of at most six variables and the Tabular method is invariably used for more than six variables. Covering tables and simplification rules are also presented.

KEY WORDS

- Binary number system
- True minterms
- False minterms
- True prime implicant
- False prime implicant
- Sub-cube
- Essential prime implicant
- Selective prime implicant
- Redundant prime implicant
- Cyclic code
- Don’t care combinations
- Optional combinations
- Weight of a minterm
- Adjacency
- Hamming distance
- Karnaugh map
- Covering table

REVIEW QUESTIONS

1. The minimal expression for the Boolean function mapped blew is

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

   a) C′D + A′B′C′ + AB′C′ + A′B′C′D + AB′CD′

   b) B′CD′ + B′C′D + C′D

   c) B′D′ + C′D

   d) A′B′CD′ + AB′CD′ + A′B′C′ + AB′C′ + BC′D

2. The code used for labelling the cells of the K–map is

   a) 8-4-2-1 Binary b) Hexadecimal c) Gray d) Octal

3. Don’t care combinations are marked by Xs. The minimal expression for the function shown on the K–map is

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

   a) (C + D) (C′ + D′) (A + B′)

   b) (B′ + C + D) (A + B′ + C)

   (A + B + C′ + D′) (A′ + B′ + C′ + D′)

   c) (B′ + C + D) (A + B′ + C + D)

   (A + B + C′ + D′) (A′ + B′ + C′ + D′)

d) (A + B′ + C) (A′ + B′ + C + D)
(A + B + C′ + D′) (A′ + B′ + C′ + D′)

4. The number of cells in a 6-variable K–map is

a) 6 b) 12 c) 36 d) 64

5. What is the maximum number of prime implicants that a 4-variable function can have?

a) 16 b) 8 c) 4 d) 2

6. How many Boolean functions can be formed with n variables?

a) 2^{2n} b) 2^{2n} c) 2^{n} d) 2^{2n-1}

7. The Quine–McCluskey method of minimisation of a logic expression is a

a) graphical method b) algebraic method
c) tabular method d) a computer-oriented algorithm

The correct answers are

a) 3 and 4 b) 2 and 4 c) 1 and 3 d) 1 and 2

8. In the Quine–McCluskey method of minimisation of the function f(A, B, C, D), the prime implicant corresponding to – 11 – is

a) A′BCD′ b) BC c) B′C′ d) AB′C′D

9. Prime implicants of a function f(w, x, y, z) = Σm(0, 1, 3, 7, 8, 9, 11, 15) are given by the following code groups. Which of these are EPIs?

A = (0, 1, 8, 9)

B = (1, 3, 9, 11)

C = (3, 7, 11, 15)

10. In simplification of a Boolean function of n variables, a group of 2^m adjacent 1s leads to a term with

a) m – 1 literals less than the total number of variables
b) m + 1 literals less than the total number of variables
c) n + m literals
d) n – m literals

11. The minimised expression for the function

$$f(A, B, C, D) = \Sigma_m(2, 3, 7, 8) + \Sigma_d(10, 11, 12, 13, 14, 15)$$

a) A′BC′D′ + A′CD + A′B′CD′ b) AD′ + CD + BC
c) BC′D′ + CD + A′B′C d) BC′D′ + CD + B′C

15. Find the minimal expression for

$$f(A, B, C, D) = \Pi M(0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14)$$

16. Define prime implicant.

17. Obtain the minimal sum of products form from the function given by

$$F(A, B, C, D) = \Sigma(1, 4, 5, 6, 7) + \Sigma_\phi(8, 9, 10, 11, 13, 14, 15).$$

18. Find the prime implicants of the function given by

$$G(x1, x2, x3, x4) = \Sigma(3, 7, 11, 12, 13, 14, 15)$$

19. Consider a 4-variable Boolean function. List the decimal codes of the cells adjacent to cell 13.
20. List the maxterms of the function $G(x, y, z) = xyz + x^1z$.
21. On an n-variable K–map a certain prime implicant comprises 2^m cells. How many variables will be present in the term indicating the prime implicant ?
22. Find the minimal sum of products form for the function given by

$$R(w, x, y) = wxy' + w'xy + x'y'$$

23. F(w, x, y, z) = Σ(0, 6, 8, 13, 14). Simplify in product of sums form.
24. $w'z + xz + x'y + wx'z$. Simplify using a 4-variable K–map.
25. $T(w, x, y, z) = \Sigma(1, 2, 3, 5, 13) + \Sigma_\phi(6, 7, 8, 9, 11, 15)$. Find a minimal sum of products expression.
26. Obtain the minimal SP form for the function given by

$$F(A, B, C,D) = \Sigma(1, 4 - 7)\ \Sigma_\phi(8 - 11, 13 - 15).$$

27. A certain switching function is given by f(a, b, c) = Σ(0, 2 – 5, 7)
    a) How many true prime implicants does this function have?
    b) How many prime implicants are required for the minimal SP form?
    c) Find the false prime implicants of the function.
28. Consider a 5-variable Boolean function. List the decimal codes of the cells adjacent to cell 13.
29. What do you understand by don't care conditions?
30. Simplify using a 4-variable K–map.

$$F(A, B, C, D) = \Sigma(0, 3, 4, 7, 8) + \Sigma_\phi(10, 11, 12, 13, 14, 15).$$

31. One of the prime implicants obtained from the tabulation procedure is 10, 11, 14, 15 (1, 4) of the function f(A, B, C, D). Obtain the expression for the given prime implicant.
32. F(W, X, Y, Z) = Σ(1, 2, 3, 4, 5) + Σd(6, 7, 8, 9, 11, 15). Simplify using a K–map.
33. F(W, X, Y, Z) = Σm(0, 6, 8, 13, 14). Simplify in POS form.
34. In the tabulation method of simplification, if the function f(w, x, y, z) has one of the prime implicants marked by 0, 2, 8, 10 (2, 8), obtain the corresponding prime implicant in terms of wxyz.
35. Define essential prime implicant.
36. Identify the prime implicants and essential prime implicants of

    Σm(1, 6, 8, 14, 15, 16).
37. Minimise the Σm(3, 5, 6, 7) using the tabular method.
38. Simplify using a 4-variable K–map.

$$f(A, B, C, D) = \Sigma m(0, 1, 4, 8, 10, 11, 12) + \Sigma_\phi(2, 3, 6, 7, 9)$$

39. Minimise using a K–map.

$$f(A, B, C, D) = \Pi(0, 2, 4, 6, 8)$$

40. In an m-variable Karnaugh map, each cell will have "m + 2" adjacent minterms. State True/False.
41. Map the function $A \oplus B \oplus C \oplus D$.

42. Map the equivalence function $A \odot B \odot C \odot D$.
43. List the true minterms of a 3-variable EXCLUSIVE OR function.
44. List the true minterms of a 3-variable EQUIVALENCE function.
45. Give an example of F(A, B, C) such that it is a self-dual function.
46. What is the maximum number of self-dual functions of n variables?
47. What is the maximum number of functions of n variables?
48. Give an example such that F(A, B, C, D) = F(A′, B′, C′, D′).
49. What is the maximum number of functions of n variables which obey the property that the function remains invariant on complimenting all the variables, that is,

$$F(x_1, x_2, \ldots. x_n) = F(\overline{x_1}, \overline{x_2}, \ldots. \overline{x_n}).$$

50. If F(A, B, C, D) = F(A′, B′, C′, D′), then show that $\overline{F(A, B, C, D)} = F_d(A, B, C, D)$ where the suffix ‘d’ denotes the dual function.
51. What are the disadvantages of a K–map.
52. Find the minimal product of sums for f(x, y, z) = x′z′ + y′.
53. For the function f(a, b, c) = Π(2, 4, 6), write the canonical product of sums.

PROBLEMS

1. Find the minimal cost sum of products expression for the following Boolean functions.

$$F(A, B, C, D, E) = \Sigma(5, 11, 13, 15, 16, 24, 25, 28, 29, 31) + \Sigma_\phi(0, 4, 7, 8, 14, 27)$$

2. Represent the function F(A, B, C, D) = Σ(1, 2, 3, 6, 7, 8, 9, 10, 13) on a Veitch–Karnaugh map. Indicate all prime implicants and write a minimum cost sum of products expression.
3. Minimise using the map method and get the PS form.

$$F(A, B, C, D, E) = \Pi(0, 3, 13, 14, 15, 17, 18, 28, 29, 30, 31)$$
$$+ \Pi_\phi(1, 2, 9, 11, 16, 19, 21, 23, 25, 27)$$

4. A circuit receives a 3-bit long binary numbers A and B. ‘A’ comprises bits a2a1a0 and B comprises the bits b2b1b0. Find the minimal SP expression which will be 1 whenever A is greater than B. Use a 6-variable K–Map.
5. Minimise using the tabular method.
   a) F(a, b, c, d, e) = V(0, 4, 12, 16, 18, 24, 27, 28, 29, 31)
   b) $G(x_1, x_2, x_3, x_4, x_5) = \Sigma(0, 1, 3, 8, 9, 13, 14, 15, 16, 17, 19, 24, 25) + \Sigma_\phi(27, 31)$
6. Obtain simplified expressions in sum of products for the following Boolean functions using the Karnaugh Map.
   a) F(A, B, C, D) = Σ(7, 13, 14, 15)   b) F(w, x, y, z) = Σ(2, 3, 12, 13, 14, 15)
7. Using the Quine–McCluskey method of tabular reduction, minimise the given combinational single-output function.

$$f(w, x, y, z) = \Sigma m(0, 1, 5, 8, 10, 14, 15)$$

8. a) Obtain the simplified expression in sum of products for the following Boolean functions using the Karnaugh map.

i) $F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15) + \Sigma_d(0, 2, 5)$

ii) $F(A, B, C, D) = ABD + A'C'D + A'B + A'CD' + AB'D$

b) Show the Truth Table for each of the following functions and find the simplest product of sums form.(POS)

i) $f(x, y, z) = xy + xz$

ii) $f(x, y, z) = x^1 + yz^1$

9. Using the tabular method, obtain the prime implicants of a four-input single-output function $f(w, x, y, z) = \Sigma m(0, 2, 4, 5, 6, 7, 8, 9, 10, 11, 13)$. Reduce the prime implicant table and find the minimal cover of f.

10. Use the tabulation procedure to generate the set of prime implicants and to obtain all the minimal expressions for the following function.

$$F(w, x, y, z) = \Sigma m(1, 5, 6, 12, 13, 14) + \Sigma_d(2, 4)$$

11. a) Minimise the following expression using the Karnaugh map.

$$f(A, B, C, D) = \Sigma m(1, 4, 7, 10, 13) + \Sigma_d(5, 14, 15)$$

b) Find the complement and dual of the function below and then reduce it to a minimum number of literals in each case. $F = [(ab)'a]\,[(ab)'b]$

12. Derive the implicant chart for the given expression.

$$f(v, w, x, y, z) = \Sigma m(0, 4, 12, 16, 19, 24, 27, 28, 29, 31).$$

Find a minimal expression.

13. a) Use the Karnaugh map to minimise

$$F = AB' + A'B'C'D' + ABC'D + A'BCD + ABD + B'CD' + A'BC'D$$

b) i) Convert (A98B)12 to ( )3.

ii) Determine Gray Code for $(97)_{10}$.

iii) Perform the subtraction on the following decimal numbers by using 9s and 10s complements.

1) 6521 – 433 2) 3674 – 598

14. Simplify the function using the Karnaugh map method.

$$F(A, B, C, D) = \Sigma(1, 4, 6, 8, 11, 13, 14)$$

15. Minimise the function using Karnaugh map method.

$$F(A, B, C, D) = \Sigma_m(1, 2, 3, 8, 9, 10, 11, 14) + \Sigma d(7, 15)$$

16. Draw a K–map and minimise.

$$Y = D'B'A' + D'C'B' + D'CB'A + CBA + CBA' + DC'B$$

17. Simplify the following function by the K–map method.

$$f(A, B, C, D) = m(0, 5, 6, 8, 9, 10, 11, 13) + d(1, 2, 15)$$

18. Obtain a minimal product of sums realisation of

$$f(w, x, y, z) = \Sigma m(3, 4, 5, 7, 9, 13, 14, 15)$$

19. Determine the minimal SOP realisation for the incompletely specified function $F(W, X, Y, Z,) = \Sigma m(4, 5, 7, 9, 13, 14, 15) + d(3, 8, 10)$ using the Karnaugh map.

20. Determine the minimal SOP realisation for the incompletely specified function f(A, B, C, D) = Σm(1, 3, 5, 8, 9, 11, 15) + d(2, 13) using the Karnaugh map.

21. Define prime implicant and essential prime implicant. Minimise the given function using the tabular method.

$$f(W, X, Y, Z) = \Sigma m(1, 4, 8, 9, 13, 14, 15) + d(2, 3, 11, 12)$$

22. Define prime implicant and essential prime implicant. Minimise the given function using the tabular method

$$f(A, B, C, D) = \Sigma m(0, 4, 6, 8, 9, 10, 12) + d(5, 7, 14)$$

23. Define prime implicant and essential prime implicant. Minimise the following function by the tabular method.

$$f(A, B, C, D, E) = \Sigma m(0, 1, 2, 3, 4, 6, 9, 10, 15, 16, 17, 18, 19, 20, 23, 25, 26, 31)$$

24. With the aid of a Karnaugh map, determine the minimal SOP expression for the given function.

$$f(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 8, 10, 12)$$

25. Using the Karnaugh map, determine the sum-of-products realisation of the following expression.

$$f(w, x, y, z) = \Sigma(1, 4, 8, 9, 13, 14, 15) + \Sigma_{\phi}(2, 3, 11, 12).$$

26. Use the QM method to determine the set of prime implicants and obtain the minimal expression for the following function.

$$F(A, B, C, D, E) = \Sigma m(8, 12, 13, 18, 19, 21, 22, 24, 25, 28, 30, 31) + \Sigma_{\phi}(9, 20, 29).$$

27. Use the QM (tabular) method using only decimal codes and obtain all possible minimal expressions for the following function.

$$F(A, B, C, D, E) = \Sigma_m(8, 12, 13, 18, 19, 21, 22, 24, 25, 28, 30, 31) + \Sigma_{\phi}(2, 6, 9, 20, 26, 29)$$

28. For the Boolean function f(w, x, y, z) = Σ(1, 4, 6, 7, 8, 9, 10, 11, 15), the prime implicants are

a) 1, 9 (8) – A
b) 4, 6 (2) – B
c) 6, 7 (1) – C
d) 7, 15 (8) – D
e) 11, 15 (4) – E
f) 8, 9, 10, 11, (1, 2) – F.

Find the essential prime implicants.

29. For the Boolean function f(w, x, y, z) = Σ(1, 4, 6, 7, 8, 9, 10, 11, 15), the prime implicants are

a) $x'y'z$ (1, 9)
b) $w'xz'$ (4, 6)
c) $w'xy$ (6, 7)
d) xyz (7, 15)
e) wyz (11, 15)
f) wx' (8, 9, 10, 11)

Find the essential prime implicants.

30. Minimise the following function by the tabular method. Use only decimal codes for minterms.

$$f(A, B, C, D, E) = \Sigma(5, 11, 13, 15, 16, 24, 25, 28, 29, 31) + \Sigma_{\phi}(0, 4, 7, 8, 14, 27)$$

31. Obtain the simplified expression for F(A, B, C, D) = Σ(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) in product of sums.

32. Obtain the expression for F′ in product of sums for

$$F(W, X, Y, Z) = \Pi(1, 3, 7, 11, 15)$$
$$d(W, X, Y, Z) = \Sigma(0, 2, 5)$$

33. Simplify by tabulation method.

$$F(W, X, Y, Z) = \Sigma(1, 3, 5, 7, 13, 15)$$

34. If the dual of a function is equal to the function, then it is called self-dual function.

Let $F = A'BC + D(AC + B) + G(A, B, C, D)$

Find a $G(A, B, C, D)$ which makes $F = F_d$ where F_d is the dual of the function.

35. Four persons are members of a TV panel game. Each have an ON/OFF button, that is used to record their opinion of a certain pop record. Instead of recording individual scores, data processing is required such that the scoreboard shows a HIT when the majority vote is in favour and a MISS if it is against. Provision must also be made for a TIE. From this statement

a) Derive the Truth Tables separately for HIT, MISS, and TIE.

b) Extract SOP and POS for each of the three outputs.

c) Simplify the equation in SOP form.

36. a) Obtain the simplified expression in sum of products for the following Boolean functions using a Karnaugh map.

i) $F(A, B, C, D) = \Sigma(7, 13, 14, 15)$

ii) $F(w, x, y, z) = \Sigma(2, 3, 12, 13, 14, 15)$

37. a) Minimise the given multiple output function x, y, z are inputs and F, G are outputs.

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

b) Give a two-level NAND-NAND circuit for the given expression.

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

c) Obtain the two-level OR–AND circuit for the given expression.

$F = (x + y') \bullet .z + (x' \bullet y \bullet z')$

38. Using the Karnaugh map method express the function in standard SOP form

$$f = AB + AC' + C + AD + AB'C + ABC$$

39. Find the prime implicants, essential prime implicants and number of minimal expressions for the given function.

$$f(A, B, C, D) = \Sigma m(1, 3, 5, 7, 8, 10, 12, 13)$$

4

Design of Combinational Circuits

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Encoders, decoder, multiplexer, demultiplexer.
- Modular expansion.
- Parity bit generator.
- Priority encoder.
- Code converters.
- Magnitude comparators.
- Adders and high speed adders.
- Subtractors.
- Logic of programmable logic devices such as ROM, PROM, PLA and PAL and their applications.
- Static and dynamic hazards that can be identified and the techniques of eliminating static hazards by design.

The proceeding chapters demonstrated how switching functions are obtained from a given word statement of a problem and the different techniques of simplifying such functions. This chapter is devoted to familiarising the learner with the various combinational logic circuits used in digital systems such as digital computers, digital communication systems, digital instrumentation, and so on. In all logic circuits, the ground line, usually common to the input port and the output port, is omitted. Such diagrams are referred to as single line diagrams, which show only the inputs and outputs but not the common ground line.

4.1 DESIGN USING AND, OR, NOT, NAND AND NOR GATES

Switching functions expressed in sum of products form are easily realised in AND-OR form illustrated by the following example.

Example 4.1 Realise the function given in SOP form as

$$F(A, B, C) = ABC' + A'BC + A'B'C'$$

Assuming that the variables and their complements are available, the AND-OR logic circuit shown in Fig. 4.1(a) produces the given function. Even if the complemented variables are not available, it would only mean inserting inverters where necessary.

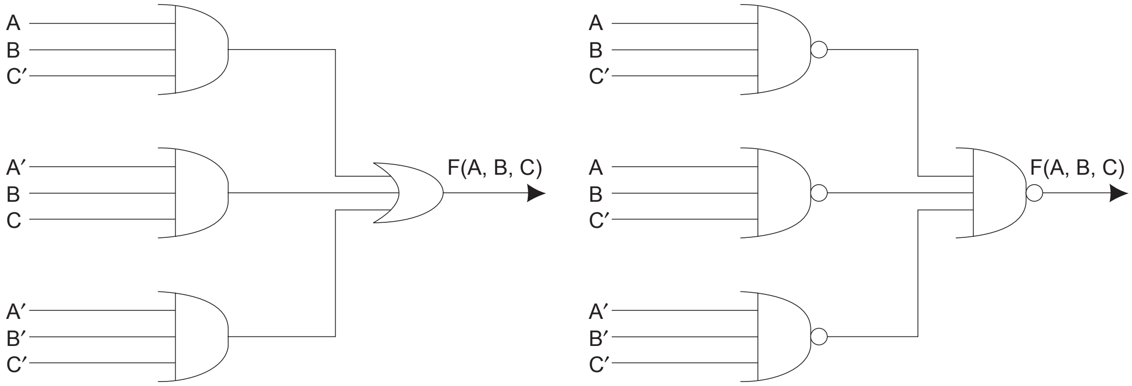
Now suppose that all the AND gates at the input level and the OR gate at the output level are replaced by NAND gates. It is interesting to note that the resulting circuit shown in Fig. 4.1(b) still realises the same function as illustrated below

$$F(A, B, C) = \overline{\overline{ABC'} \bullet \overline{A'BC} \bullet \overline{A'B'C'}}$$

$$= ABC' + A'BC + A'B'C'$$

by applying De Morgan's Theorem for removing the full bar on top of the expression.

Notice that the symbols both prime and bar are used for complementation depending on the space available and clarity required.



(a) AND-OR realisation of the function in SOP form

(b) NAND realisation of the function

Fig. 4.1 Realisation of the function F(A, B, C) = ABC′ + A′BC + A′B′C′

Notice also that **NAND realisation can easily be obtained by replacing all gates in AND-OR realisation by NAND's.** Additional NAND gates may be required for complementing the input variables if complemented variables are not available.

Switching functions expressed as a product of sums are easily realised in OR-AND form illustrated by the following example.

Example 4.2 Realise the function given in POS form as

$$F(x, y, z) = (x + y' + z)\,(x' + y + z')\,(x' + y')$$

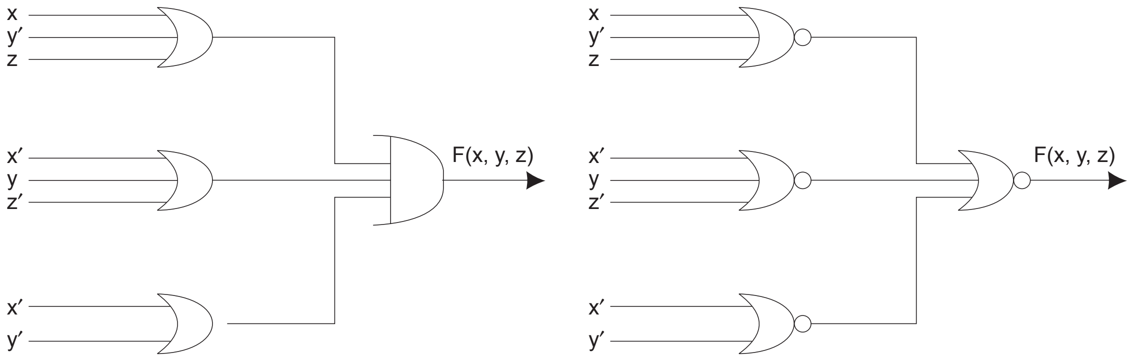
The OR-AND circuit shown in Fig. 4.2(a) realises the above function. It is interesting to note that if all the gates, namely the OR gates at the input level and the AND gate at the output level, are replaced by NOR gates, then the resulting circuit shown in Fig. 4.2(b) produces the same function. The algebraic working is illustrated below

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

$$= (x + y' + z) \bullet (x' + y + z') \bullet (x' + y')$$

by applying De Morgan's Theorem for removing the full bar on top of the expression.

The reader must have noticed that the expressions and circuits of Examples 4.1 and 4.2 are duals of each other. Further, they are referred to as **"two level realisations"** meaning that the input signals have to pass through at most two logic gates to reach the output. To produce a given function, it is possible to reduce the total number of gates if the designer chooses to use a multi-level gate network, which implicitly carries with it differential propagation delays which might cause problems in operation. Minimising the number of gates is never resorted to only for the purpose of reducing the cost but for other reasons of reliability and *testability*.



(a) OR-AND realisation of the function in POS form

(b) NOR realisation of the function F(x, y, z)

Fig. 4.2 Realisation of the function F(x, y, z) = (x + y′ + z) (x′ + y + z′) (x′ + y′)

4.2 ENCODER, DECODER, MULTIPLEXER, DE-MULTIPLEXER

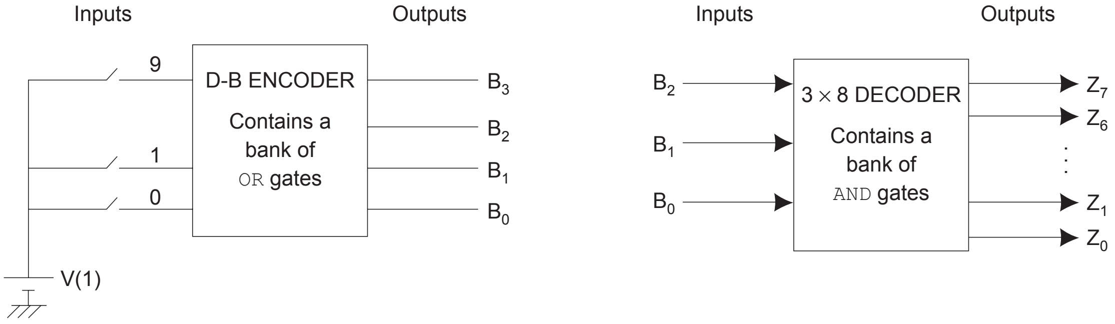
In Chapter 1, a number of codes including binary codes for decimal digits, alphanumeric codes such as ASCII and EBCDIC codes used in digital systems and computers were discussed. The logic circuit which produces such codes is called a **Encoder** and is discussed in this section. The related circuits namely, **decoder**, **multiplexer**, and **demultiplexer** circuits are also discussed. A digital system designer uses these circuits extensively. A special circuit called a “Priority Encoder” which finds application when preferring an input over other inputs as per the user’s specifications is also described.

Encoder

Fig. 4.3(a) shows an encoder which produces a binary code for each of the decimal digits 0, 1, …, 9. On the input side, only one of the inputs is kept at logic ‘1’ level by pressing the corresponding switch. To have 10 distinct binary combinations, we need to have at least four bit positions numbered B_0, B_1, B_2, B_3 as the outputs. With four bits, it is possible to have $2^4 = 16$ distinct binary combinations but in this case, we use only ten of them and discard the remaining six combinations, which are normally referred to as **Optional**, or **don’t care** combinations. The outputs, B_3, B_2, B_1, B_0 will assume the values as assigned by the code. For example, if one wants to produce a binary number corresponding to the input digit ‘5’, the outputs will be $B_3\,B_2\,B_1\,B_0 = 0101$. The digits and the corresponding 4-bit words are listed in Table 4.1.

Table 4.1 *Binary Words for Decimal Digits*

Digit	B_3	B_2	B_1	B_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1



(a) Decimal to binary encoder

(b) Decoder for 3 binary inputs

Fig. 4.3

Notice that

B_0 whose weight is $2^0 = 1$ has to be 1 for the input digits 1, 3, 5, 7, 9.

B_1 whose weight is $2^1 = 2$ has to be 1 for the input digits 2, 3, 6, 7.

B_2 whose weight is $2^2 = 4$ has to be 1 for the input digits 4, 5, 6, 7.

B_3 whose weight is $2^3 = 8$ has to be 1 for the input digits 8, 9.

Thus, the outputs are realised simply by using OR gates inside the D-B encoder block of Fig. 4.3(a) shown explicitly in Fig. 4.3(a) Annexure below.

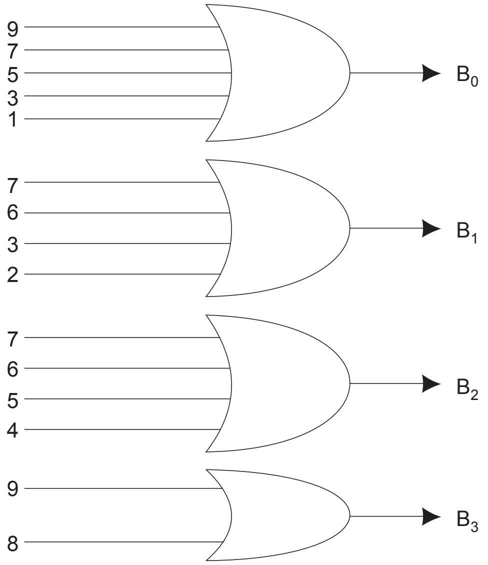


Fig. 4.3(a) Annexe: bank of OR gates forming encoder

Decoder

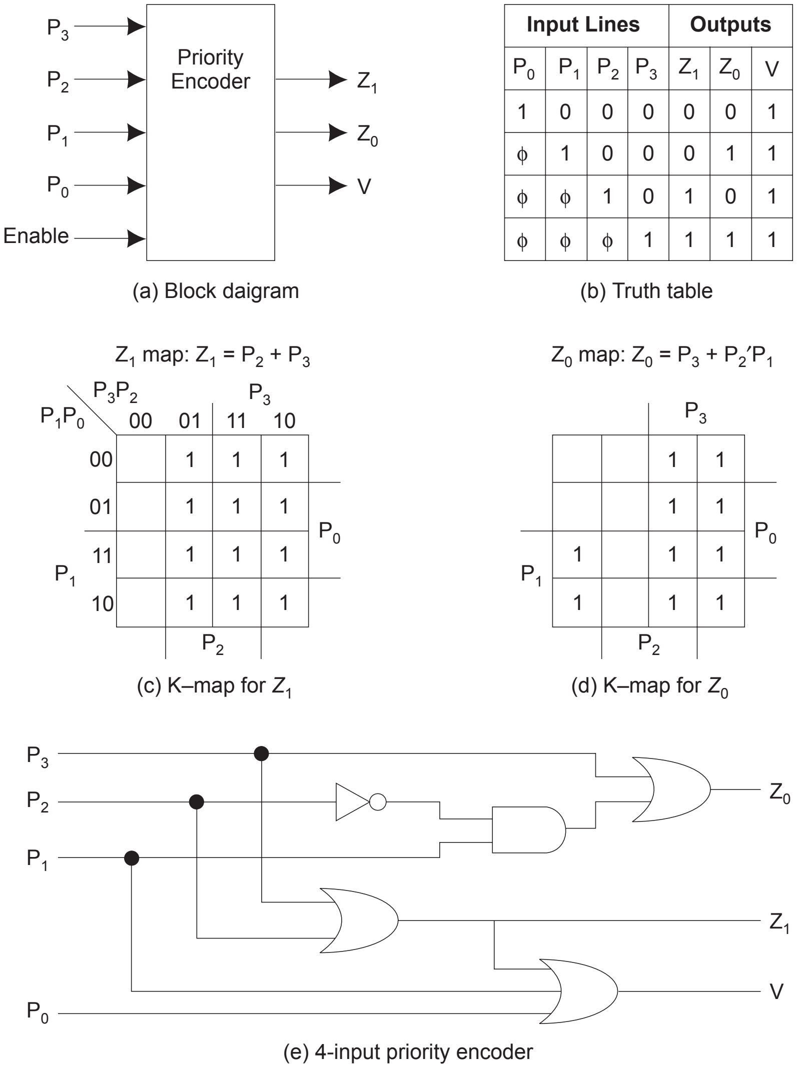
An example of a decoder for a 3-bit code word is shown in Fig. 4.3(b) With three bits, one can form $2^3 = 8$ distinct combinations; let us call them 0, 1, …, 6, 7. For each set of three binary inputs, there will be one and only one of the eight outputs assuming logic 1 level; all the remaining outputs should be at '0' level. This is precisely the property of a 3 to 8 decoder. It is easily seen that the block contains eight AND gates whose outputs are $Z_0 \ldots Z_7$ which are essentially minterms of a function of three variables. Each of the AND gates will have three inputs either in primal (un-complemented) form or complemented form, that is, B_2 or B_2', B_1 or B_1', and B_0 or B_0'. It is possible to have a modular approach and get expansion capabilities (discussed in the next section)

Priority Encoder

A priority encoder is a useful device in many situations; For instance, there may be many subscribers to an intercom in an office. Among the subscribers, there needs to be some priority scheme, so that the Chief Executive is given some preference while other Executives too will have their own hierarchy in the priority scheme. It is possible to ensure this kind of prioritisation using a priority encoder.

Digital computers are generally connected to many peripheral devices like input data entry devices or output printers, memory devices and so on. At times, a priority scheme is needed among them. In process control systems, there may be situations when several component machines may require the attention of a central computing facility. For instance, a fuming boiler may have to be attended to on a priority basis compared to routine load transfer applications. Thus, there must be a strategy to evolve a priority hierarchy in many situations.

The requirement reduces to choosing one with the highest priority among the inputs, which becomes logic 1. Figure 4.4(a) shows the block diagram of the priority encoder for four inputs. The reader is advised to have a keen look at the Truth Table given in Fig. 4.4(b) Notice that diagonal entries are marked ‘1’ among the input



(a) Block daigram

Input Lines				Outputs		
P_0	P_1	P_2	P_3	Z_1	Z_0	V
1	0	0	0	0	0	1
ϕ	1	0	0	0	1	1
ϕ	ϕ	1	0	1	0	1
ϕ	ϕ	ϕ	1	1	1	1

(b) Truth table

Z_1 map: $Z_1 = P_2 + P_3$

P_1P_0 \ P_3P_2	00	01	11	10
00		1	1	1
01		1	1	1
11		1	1	1
10		1	1	1

(c) K–map for Z_1

Z_0 map: $Z_0 = P_3 + P_2'P_1$

		1	1
		1	1
1		1	1
1		1	1

(d) K–map for Z_0

(e) 4-input priority encoder

Fig. 4.4 Priority encoder

columns. It is assumed that the higher numbered inputs will have higher priority in an hierarchical manner. To enable output indication for the corresponding input, it follows that the lower numbered columns will be filled with don't care entries, while the higher numbered columns must all be at 0. In this particular case of four inputs, the reader may actually like to verify all the 16 combinations, yielded by assigning all possible truth values to the optional entries. In brief, the output Z_1 should become 1 for $P_2 P_3' + P_3 = 1$ which simplifies to $P_2 + P_3 = 1$. Likewise, the output Z_0 should become 1 for $P_1 P_2' P_3' + P_3 = P_1 P_2' + P_3 = 1$.

Figures 4.4(c) and (d) depict the Karnaugh maps for the outputs Z_1 and Z_0, which are easily synthesised as given below.

$$Z_1 = p_2 + p_3$$

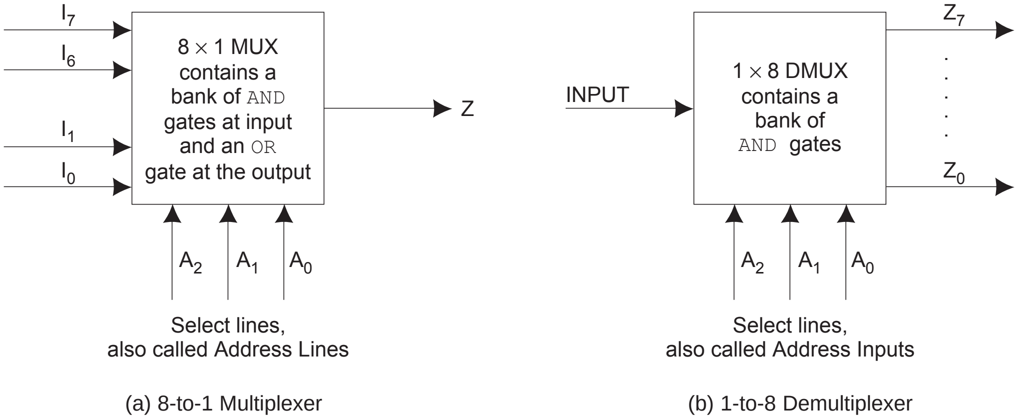
$Z_0 = p_3 + p_2'.p_1$ Note that p_3' is a redundant literal in the expression for Z_0.

$$V = p_0 + p_1 + p_2 + p_3$$

The logic circuit of a 4-input priority encoder is shown in Fig. 4.4(e). In integrated versions of this circuit, the "Chip Enable" lead helps the designer to achieve expansion capability. With two such modules and some external logic circuits, one can build an 8-input priority encoder. The extra output 'V' becomes 1, if the unit is active, thereby meaning that the outputs are valid and that at least one of the inputs is at '1' level. This requires one OR gate. If V = 0, outputs are to be ignored as the unit is idle, which means that it is not being used at present. The output lead, V, can be used intelligently to inhibit the module of the lower order in hierarchy.

Multiplexer (MUX)

A multiplexer is a device that helps to choose one out of many. It is also called a **Data Selector.** As an illustration, an 8-to-1 multiplexer is shown in Fig. 4.5(a) It has eight input lines (also called channels) and only one of them is allowed to be transmitted to the output, depending on the select lines (also called address lines or control lines) Since we have to select one out of eight, we need to have three binary select inputs to an AND gate to which the corresponding input channel will also be connected as a fourth input. Thus, the block contains eight AND gates, each with four inputs. The outputs of the AND gates become the inputs to an OR gate whose output is the final output of the multiplexer. For instance, if A_2 A_1 A_0 are set to 101, then the input I_5 alone will pass through the corresponding AND gate which has inputs A_2, A_1', $A_{0,}$ I_5 and the following OR gate


I7
I6
I1
I0
8 × 1 MUX contains a bank of AND gates at input and an OR gate at the output
Z
A2
A1
A0
Select lines, also called Address Lines
INPUT
1 × 8 DMUX contains a bank of AND gates
Z7
Z0
A2
A1
A0
Select lines, also called Address Inputs

(a) 8-to-1 Multiplexer

(b) 1-to-8 Demultiplexer

Fig. 4.5

and finally appear on the output Z line. The outputs of all other AND gates will be at 0 level as they are disabled because at least one of the inputs is at level 0. It is possible to have expansion capabilities by providing an "Enable" terminal (discussed in the next section). Suppose we have a counter with three binary outputs connected to A_2 A_1 A_0 and if the counter is driven by a pulse generator (pulser), the eight inputs I_0 to I_7 will appear **successively** in time on the output line Z and the whole sequence will be repeated depending on the speed of the pulser. This process is called **Time Division Multiplexing**, which is extensively used in communication systems, control systems, computers, and so on.

Demultiplexer (DMUX)

The principle of a demultiplexer is illustrated by the block diagram shown in Fig. 4.5(b), which indicates a 1-to-8 DMUX. The property of a DMUX is to steer (direct) the input to the correct output line depending on the select lines. If there are eight output lines, we need to have three select lines and eight AND gates, each with 4

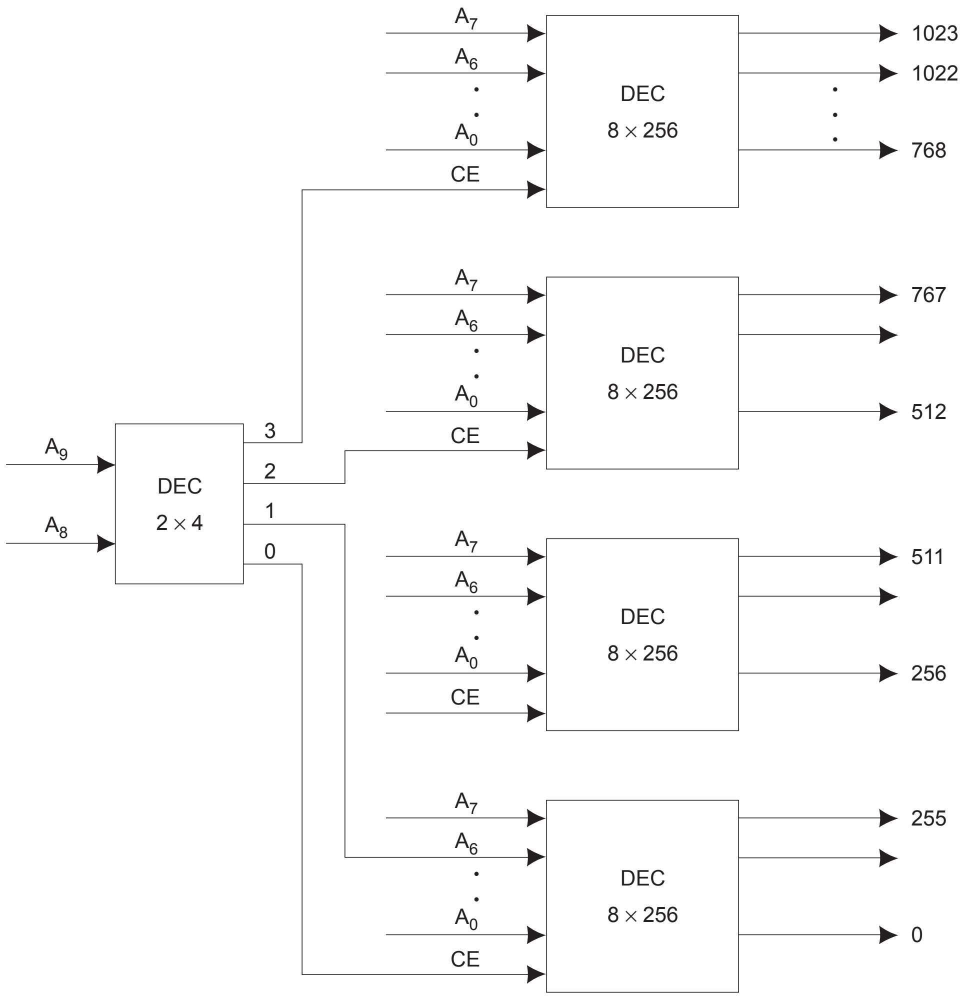


Fig. 4.6 Modular expansion of decoding

inputs-3 select inputs and the original input. The output of each AND gate goes to the corresponding output line Z_0, or Z_1, … or Z_7.

If the select lines A_2 A_1 A_0 are set to 110, the input, I, is passed through the corresponding AND gate (whose inputs are $A_2 = 1$, $A_1 = 1$, $A_0' = 1$) to the output line Z_6. All other AND gates will have zero output as they are disabled because of at least one zero at their inputs.

Suppose eight different signals are fed successively, one after another, on the input line. Further suppose that each signal remains for, say, one microsecond, on the input line in rotation. This is called **Time Division Multiplexing (TDM)**. As in the case of an MUX, if a counter is used in conjunction with a DMUX, we may receive the **Time Division Multiplexed (TDM)** signal separated on each of the outputs. Synchronisation of the pulsers (also called clock) is, of course, necessary to ensure correct sequence and correspondence among the received signal and the output signals.

4.3 MODULAR DESIGN USING INTEGRATED CIRCUIT (IC) CHIPS

Many decades ago, the design of digital circuits was based on discrete components like transistors, diodes, resistors, and so on. From the 1960s, logic circuit design relies heavily on the availability of modules called **“Integrated Circuits”** which are fabricated in solid state form and are referred to as **IC chips**. There are many levels of integration depending on the number (10s, 100s, 1000s, or larger still) of transistors and other components provided on chip. Small scale Integration (SSI), Medium Scale Integration (MSI), Large Scale Integration (LSI), and Very Large Scale Integration (VLSI) are all in use for various applications ranging from realisation of simple logic functions to large digital systems like computers. Most of the modules presented in this chapter belong to SSI or MSI category. The designer will have the choice of modular expansion by using the chip enabled lead illustrated by the following examples.

Example 4.3 Design a 10 × 1 K decoder using chips of 8 × 256 decoder and additional logic.

For each of the given decoder chips, there are eight address inputs and $2^8 = 256$ outputs. In addition, there is a **Chip Enable (CE) input**, which feeds all the AND gates in the chip. The decoded output becomes 1 only if CE is at 1. The enable input is also called **Chip Select (CS)** or **strobe** input. Four such modules are necessary to decode 10 address inputs and produce $2^{10} = 1024$ outputs. This requires a simple strategy to connect the least significant eight bits of the 10-bit address to all the four decoders and use the remaining two most significant bits to enable only one of the four chips by using discrete AND gates to produce $A_9'A_8'$, $A_9'A_8$, A_9A_8' and A_9A_8. It would be more elegant to use a small decoder of size 2 X 4 instead of discrete gates, as shown in Fig. 4.6.

Example 4.4 Design 2K × 1 MUX using 1K modules.

As another example, take a look at Fig. 4.7 where two modules each of 1K MUX are used to build a 2K MUX to select one out of 2K inputs. In this case, the designer must provide 11 address bits to produce 2^{11} = 2K different addresses. Call them A_{10} … A_0. The scheme is to connect the 10 bits, A_9 … A_0, to both the 1K chips and use the most significant bit to enable one chip with A'_{10} and to enable the other chip with A'. Finally, the two outputs have to be mixed with an output OR gate. The additional hardware in this case is thus one inverter and one OR gate. A similar scheme is commonly used in expanding the size of memory by connecting additional modules. Such simple schemes may not be useful straightaway where the design involves hierarchical levels, in which case the designer is called upon to demonstrate ingenuity and effect modifications.

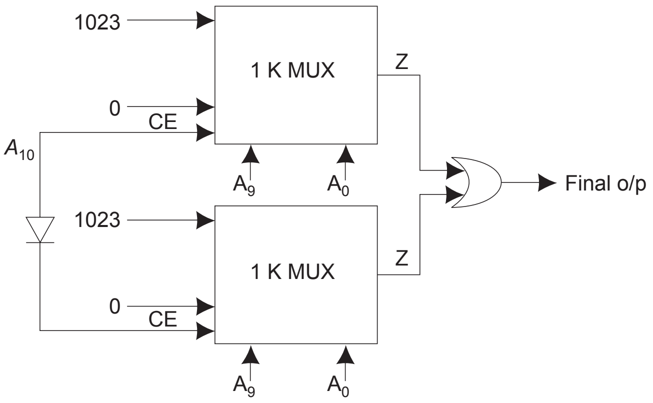


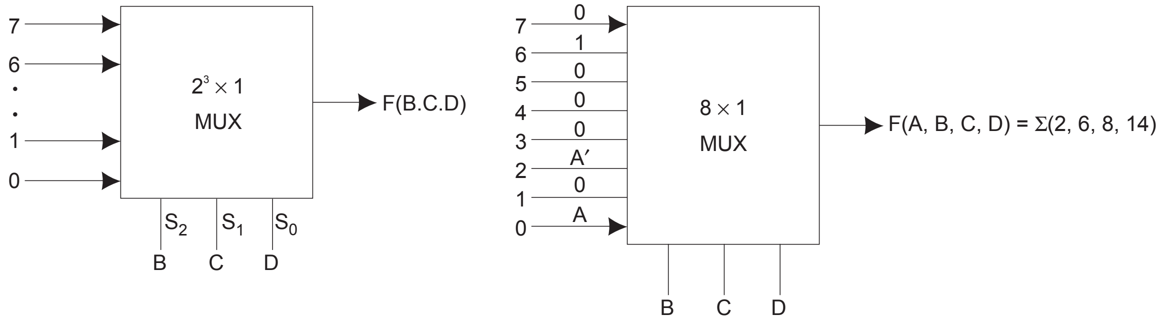
Fig. 4.7 Modular expansion of a data selector

4.4 REALISATION OF BOOLEAN FUNCTION USING A MULTIPLEXER

In Section 4.2, we learnt that the multiplexer selects one out of 2^n inputs and transmits it to the output lead. Take a look at Fig. 4.8, which depicts logic realisation using a multiplexer. As an example consider a 8 × 1 MUX shown in the Fig. 4.8(a). It has eight inputs marked 0–7 and one output f(A, B, C, D). It has three select inputs S_2, S_1, S_0. Suppose we connect three variables B, C, D to the three select inputs $S_{2,}$ S_1, S_0, respectively and decide to apply either 0 or 1 to each of the primary inputs which are now labelled as m_0, m_1 … m_7, which correspond to the minterms of the function F(B, C, D). The output of the MUX will then be the sum of the minterms m_i ($0 \leq i \leq 7$) which are at 1. This follows from the fact that the variables B, C, D are internally decoded using eight AND gates and one of the inputs m_0 … m_7 is also connected to each of the AND gates. For example, m_5 is an input to the AND gate to which other inputs are B, C′ and D corresponding to B = 1, C = 0 and D = 1. We know that the outputs of the AND gates are finally OR ed to give the output of the MUX. It follows that the output of the multiplexer will be the logical sum of minterms m_i which are at 1 level. Thus, 8 × 3 MUX can be used to realise any function of three variables B, C, D in the standard sum of products (SSP) form.” For example F(B, C, D) = Σ(2, 3, 4, 7). A little reflection shows that we need to feed logic one level at the corresponding inputs namely 2, 3, 4, 7 and 0s to the remaining inputs. The output will then contain the above Boolean function of three variables. In general, a three-variable function can be realised with MUX of size $2^3 \times 1$.

It is possible to realise a four-variable function F(A, B, C, D) using a $2^3 \times 1$ MUX with the help of a simple strategy. The procedure is indicated below.

The given MUX will have $2^3 = 8$ inputs and three select (Address) lines S_2, S_1, S_0. Feed the input variables B, C, D to the select lines in that order. Remember that the MUX contains 8 AND gates to decode S_2, S_1, S_0



(a) Logic realisation using MUX

(b) Realisation of four-variable function using 8 × 1 MUX

Fig. 4.8 Realisation of Boolean function using a MUX

Sand an OR gate at the output level. We know that the minterms of the three-variable function F(B, C, D) will be contained in the output if the corresponding primary input lines are kept at 1 level. Thus, any function of three variables can be obtained with this MUX. This logic is easily extended to a function of four variables by exploring the possibility of feeding A′, A, 0 or 1 to the input lines.

Case (i) If A′ is fed to the input line 6, it can easily be seen that the minterm 6 corresponding to the term A′ BCD′ will be contained in F and not the minterm 14 corresponding to ABCD′.

Case (ii) If A is fed to the input line 6, a little reflection reveals that the minterm 14 corresponding to ABCD′ will be contained in F and not the minterm 6.

Case (iii) If 0 is fed to the input line 6, this ensures false minterms 6 and 14 in F.

Case (iv) If 1 is fed to the input line 6, F will contain both the true minterms 6 and 14 corresponding to A′BCD′ + ABCD′ = BC D′.

The aforementioned logic is summarised in the following Table 4.2 for the purpose of synthesising the MUX. This is followed by an illustration in Table 4.3 using the minterms 5 and 13. **Generalising this, it is possible to realise a function of n variables using a MUX of size $2^{n-1} \times 1$.**

Table 4.2

Case	ith minterm	(i + 8)th minterm	ith I/P
III	0	0	0
II	0	1	A
I	1	0	A′
IV	1	1	1

Table 4.3

Minterms		5th I/P	*Minterms realised* whether False or True
5	13		
0	0	0	0 1 0 1 } F = 0 1 1 0 1
0	1	A	0 1 0 1 → F = 0 1 1 0 1 → F = 1
1	0	A′	0 1 0 1 → F = 1 1 1 0 1 → F = 0
1	1	1	0 1 0 1 } F = 1 1 1 0 1

To sum up, all the 2^n minterms (0 to $2^n - 1$) are partitioned into two sets 0 to $2^{n-1} - 1$ and 2^{n-1} to $2^n - 1$. If n = 4, then the minterms 0 to 15 are partitioned into two sets. One set contains the minterms 0 to 7, and the other set contains the minterms 8 to 15. The first set contains minterms m_i, with i varying from 0 to 7. The other set contains minterms m_{i+8}.

Case (i) Corresponds to minterm m_i but not m_{i+8} contained in F.

Case (ii) Corresponds to minterm i + 8 but not i.

Case (iii) Corresponds to both not contained in F.

Case (iv) Corresponds to both contained in F.

The following example gives a feel of the above technique to the reader.

Example 4.5 Realise F(A, B, C, D) = Σ(2, 6, 8, 14) using an 8 × 1 MUX.

Consider the true minterms m_2, m_6, m_8 and m_{14}.

$$m_2 = A' B' CD'$$

Remember that the MUX contains AND gates at the input level and an OR gate at the output level. I/P line 2 is one of the inputs to the AND gate to which the other inputs are B′ CD′.

The minterm m_{10} is a false minterm and hence the term AB′ CD′ should not be contained in SOP expression for F but it should contain A′ B′ CD′. This is achieved by feeding A′ to the input line 2. The corresponding AND gate inside MUX will then have the inputs A′ B′ CD′ and the product appears in the output of the final OR gate realising m_2 but not m_{10}. Now focus on

$$m_6 = A' BCD'$$

and $$m_{14} = ABCD'$$

Note that the true minterms m_6 and m_{14} add up to give B, C, $\overline{D}$.

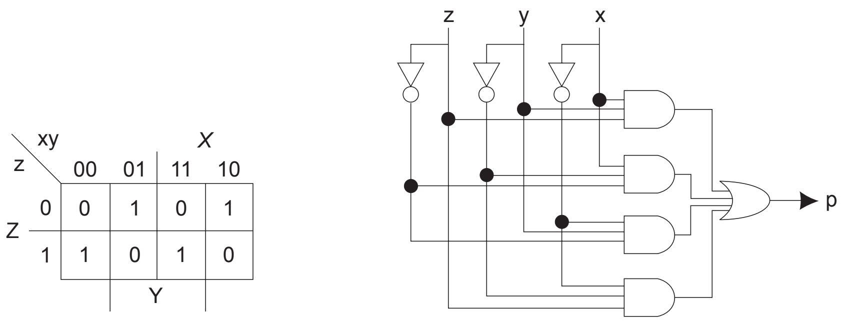
It is easily seen that both these true minterms can be realised by feeding ‘1’ to the input line 6 as the output of the corresponding AND gate in MUX will then be BCD′.

Finally, m_8 is realised by feeding A to I/P line 0 to realise the term AB′ C′ D′. Fig. 4.8(b) shows the realisation of the given function of four variables using an 8 × 1 MUX.

4.5 PARITY BIT GENERATOR

In Sections 1.9 and 1.10, we discussed error detecting codes and a single error correcting code wherein parity bits are used. In the case of the even parity BCD code shown in Table 1.13, the parity bit ‘P’ is an additional bit appended as the 5th bit to the message bits. It is the designer’s choice to choose ‘P’ for even parity or odd parity. In either case, one has to produce the exclusive OR operation of all the message bits which gives an output 1, if the number of 1s in the original message word is odd and 0 if even. The output may be appended to the message as it is, or in a complement form. Parity bit P equals the output of the exclusive OR operation or its complement, as the case may be, depending on the choice of the designer. For establishing even parity, P becomes simply equal to the exclusive OR output of the message bits.

Focus now on the single error correcting code presented in Section 1.10. Each word of the code contained seven bits numbered 1–7. The bits 1, 2 and 4 are labelled as parity bits and called P_1, P_2, and P_4. The message bits in the word are labelled M_3, M_5, M_6 and M_7. Each parity bit is chosen in such a way that together with three other message bits it forms a 4-bit word with even parity. Thus, we would like to generate a parity bit with the three message bits given. A circuit to generate such a parity bit is shown in Fig. 4.9. If the three message bits are considered x, y, z, we need to generate the parity bit P. The Karnaugh map for P is shown in Fig. 4.9(a) and the circuit to generate P is shown in Fig. 4.9(b). Notice that output P has to be 1 whenever the number of 1s in the 3-message bits is odd, that is, a single one or three ones. It is easily synthesised in SOP form from the map.



z \ xy	00	01	11	10
0	0	1	0	1
1	1	0	1	0

(a) P-map

(b) Implementation

Fig. 4.9 Parallel parity-bit generator

4.6 CODE CONVERTERS

We come across many situations when two different sub-systems use different codes. In order to establish compatibility, we need to transform or convert one code word into an equivalent code word of the other sub-system. This job is performed by a circuit called a code converter. We will demonstrate two examples of code converters in this section. These are generally **multiple output circuits.**

Example 4.6 Design a circuit to transform 8-4-2-1 BCD code into Excess-3-code.

Let us denote the BCD inputs as a, b, c, d and the Excess 3 outputs as w, x, y, z. The correspondence between the input and output code words together with the decimal numbers is shown in Table 4.4. Outputs are easily synthesised using maps shown in Fig. 4.10. The minimised Boolean expressions for the output functions are given in Table 4.4. The logic gate circuit is shown in Fig. 4.11.

$$z = \Sigma(0, 2, 4, 6, 8) + \Sigma_{\phi}(10, 11, 12, 13, 14, 15) = d'$$

$$y = \Sigma(0, 3, 4, 7, 8) + \Sigma_{\phi}(10, 11, 12, 13, 14, 15) = y = c'\,d' + cd$$

$$x = \Sigma(1, 2, 3, 4, 9) + \Sigma_{\phi}(10, 11, 12, 13, 14, 15) = x = b'\,c + b'\,d + bc'\,d'$$

$$w = \Sigma(5, 6, 7, 8, 9) + \Sigma_{\phi}(10, 11, 12, 13, 14, 15) = w = a + bc + bd$$

Table 4.4 *Truth Table for BCD and Excess-3 Codes*

Decimal number	BCD – inputs	Excess-3-outputs
	a b c d	w x y z
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

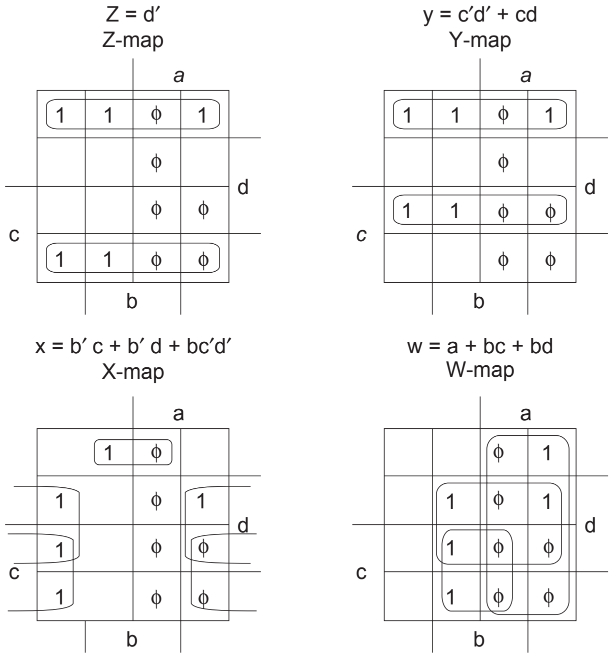


Fig. 4.10 Code conversion maps

Example 4.7 Design a circuit to transform a BCD 4-bit code into a 7-segment display code and hence derive the expressions for decoding.

For displaying decimal digits, seven segments are conventionally used as shown in Fig. 4.12. Depending on the digit, some segments are lighted and others are not. The scheme for illumination is shown in Table 4.5 by marking 1 at the corresponding position. The pattern of illumination and the corresponding lighted

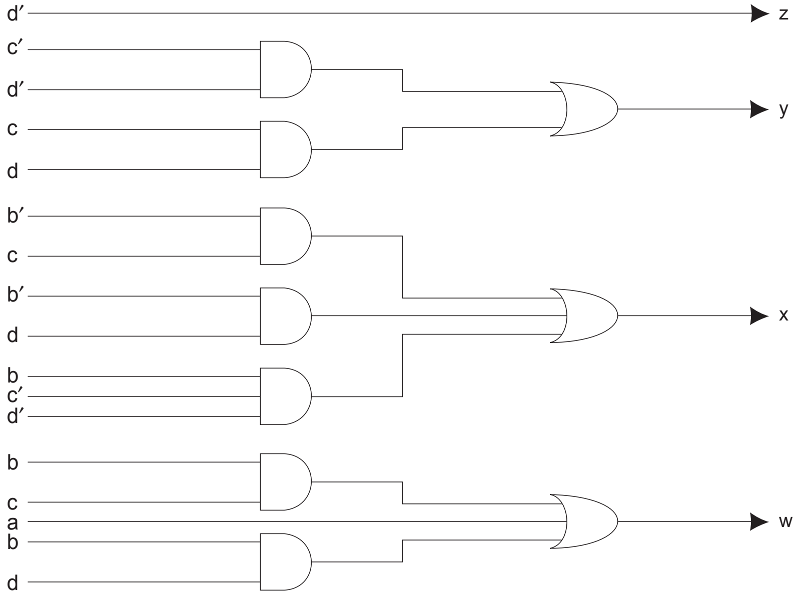


Fig. 4.11 BCD-to-Excess-3 code converter

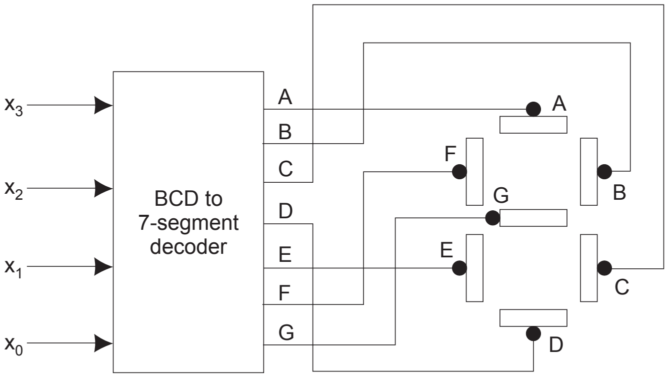
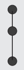
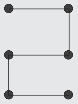
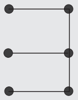
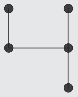
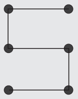
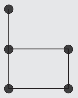
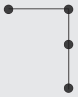
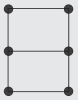
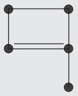
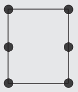


Fig. 4.12 Seven-segment display

(1)/ dark (0) segments are also shown in the table. The reader is advised to form seven maps, one for each of the segments, and synthesise A, B, C, D, E, F, G as functions of four BCD code variables x3, x2, x1 and x0. Notice that the segment excitation functions are easily obtained from the maps remembering that six cells denote don't care conditions. Alternatively, a decoder of size 4 × 16, followed by seven OR gates may be used for synthesising the multiple outputs.

Table 4.5 *Seven-segment Pattern and Code*

Digital Digit	Pattern	BCD code				Seven-segment code						
		x_3	x_2	x_1	x_3	A	B	C	D	E	F	G
1		0	0	0	1	0	1	1	0	0	0	0
2		0	0	1	0	1	1	0	1	1	0	1
3		0	0	1	1	1	1	1	1	0	0	1
4		0	1	0	1	0	1	1	0	0	1	1
5		0	1	0	1	1	0	1	1	0	1	1
6		0	1	1	0	0	0	1	1	1	1	1
7		0	1	1	1	1	1	1	0	0	0	0
8		1	0	0	0	1	1	1	1	1	1	1
9		1	0	0	1	1	1	1	0	0	1	1
0		0	0	0	0	1	1	1	1	1	1	0

4.7 MAGNITUDE COMPARATORS

If two binary numbers A and B are to be compared, it is necessary that we compare the Most Significant Bits (MSBs) first. If these MSBs are equal, then only do we need to compare the next significant bits. If the MSBs are not equal, it would then be clear that either A is less than B or greater than B, and the process of comparison ceases. Consider, for instance, two numbers each of two bits. Let A = a_1a_0 and B = b_1b_0. If $a_1 \neq b_1$, we conclude that A is less than B if $a_1 = 0$ and $b_1 = 1$ or else we conclude that A is greater than B if A = 1 and B = 0. The process of comparison ceases at this stage. If the MSBs are equal, that is, $a_1 = b_1$, only then do we need to proceed further in comparing the next significant bits a_0 and b_0 and then decide whether the numbers are equal or unequal and produce three outputs L, E, and G for less than (A < B), equal to (A = B) and greater than (A > B) outputs. Notice that it is basically an ordered process starting from the comparison of MSBs and proceeding towards LSBs. The process stops whenever the bits exhibit inequality.

The comparator for two bits, namely two numbers each of a single bit, is shown in Fig. 4.13. The logic is simple. The Truth Table and the functions are given below.

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

The output functions L, E, and G are given below and implemented in the circuit shown in Fig. 4.13(a)

$$L = A'_0, \quad E = A'_0B'_0 + A_0B_0, \; G = A_0B'_0$$

The block diagram of a 1-bit comparator is shown in Fig. 4.13(b) which will be used as a module for comparing larger numbers.

Using two modules of 1-bit comparators and some additional logic gates, one can build a comparator for two binary numbers, each of two bits. The logic equations are given below. Notice in particular that the terms in the functions L and G depend on the equality of higher significant digits. In the case of a 2-bit comparator, for instance, L becomes 1, if L_1 is 1 or if the bits in the first position are equal and bits in the 0th position differ,

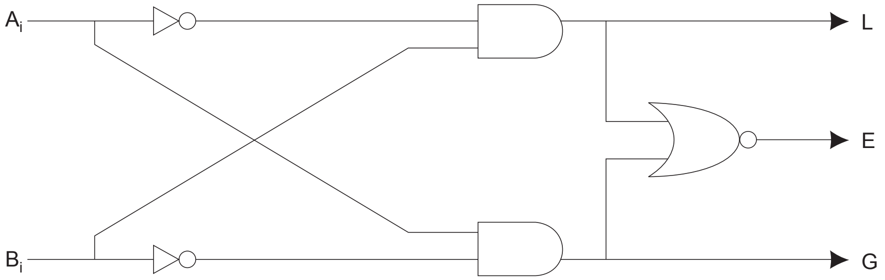
$A_0 < B'_0$: $L = A'_0.B_0$

$A_0 = B_0$: $E = A'_0.B'_0 + A_0B_0$

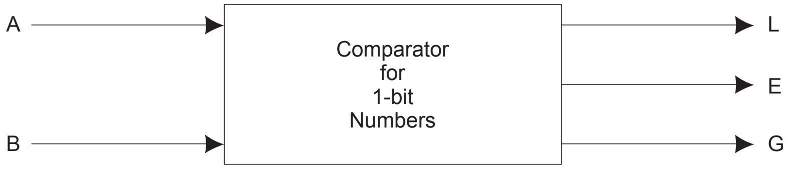
$A_0 > B_0$: $G = A_0.B'_0$

Note: E can be realised as $(L + G)'$

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



(a) Logic circuit for comparing 1-bit-numbers



(b) Block diagram of 1-bit-comparator

Fig. 4.13 Comparator for 1-bit numbers

such that a_0 is 0 and b_0 is 1. A similar argument holds for G, which becomes 1 if G_1 or $E_1 \cdot G_0$ is 1. The realisation of the logic circuit is shown in Fig. 4.14. The logic can easily be extended to a comparator for 4-bit numbers shown in Fig. 4.15. Such 4-bit comparators are produced in an IC chip form. The reader should remember that such comparators of 4-bit numbers can be conveniently used for comparison of BCD digits. The reader should also remember that it is essentially an ordered iterative process. Some clever schemes, not discussed in this book, exist for cascading the 4-bit comparator modules to achieve comparisons of longer numbers.

Let $A = A_1A_0$ and $B = B_1B_0$.

$A < B: L = A_1'B_1 + E_1 \cdot A_0'.B_0 = L_1 + E_1 \cdot L_0$

$A = B: E = E_1E_0$

$A > B: G = A_1.B_1' + E_1 \cdot A_0B_0'. = G_1 + E_1 \cdot G_0$

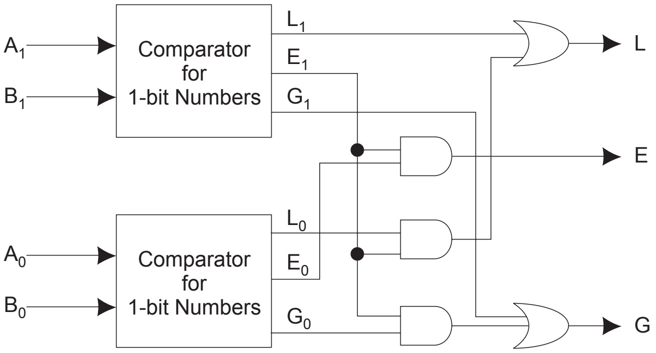


Fig. 4.14 Comparator for 2-bit numbers

Let $A < B: L = L_3 + E_3 \bullet L_2 + E_3 \bullet E_2 \bullet L_1 + E_3 \bullet E_2 \bullet E_1 \bullet L_0$

$A = B: E = E_3 \bullet E_2 \bullet E_1 \bullet E_1$

$A > B: G = G_3 + E_3 \bullet G_2 + E_3 \bullet E_2 \bullet G_1 + E_3 \bullet E_2 \bullet E_1\ G_0$

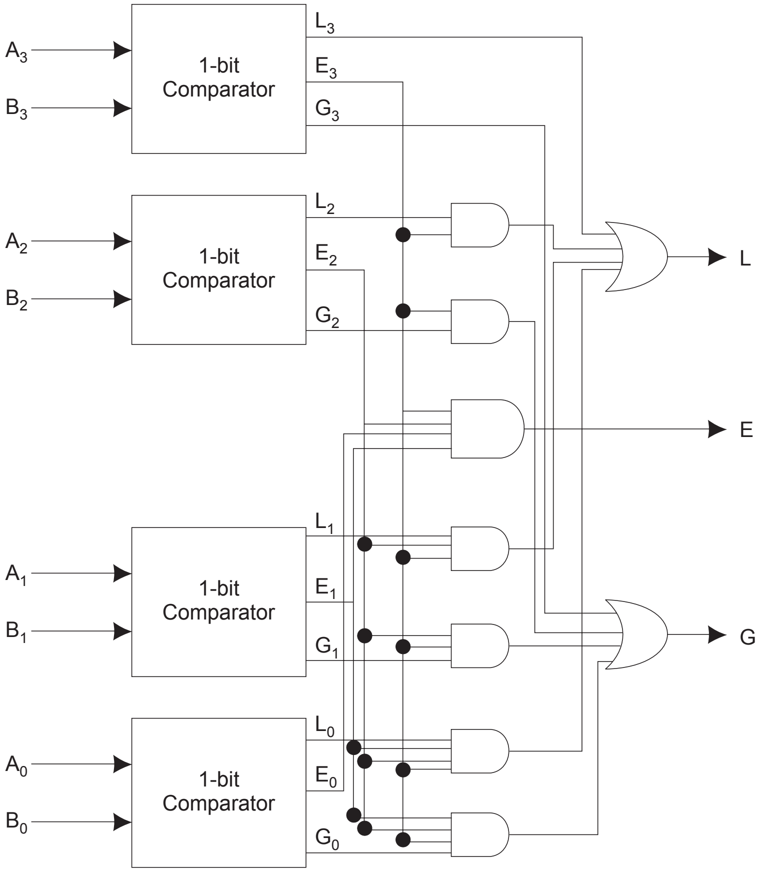


Fig. 4.15 Comparator for 4-bit numbers

4.8 ADDERS AND SUBTRACTORS

Arithmetic circuits, such as adders and subtractors, multipliers, dividers and so on find application in digital computers. Generally, only adders are used to obtain other arithmetic operations also if complementing facility is available. Subtraction is achieved by complementing the subtrahend and adding to the minuend. Magnitude comparators discussed in Section 4.7 are also a kind of arithmetic circuit. In this section, circuits for binary half adder, full adder, high- speed carry-look-ahead adder, and BCD adder for decimal digits are discussed. These are examples of arithmetic circuits which are available in MSI/LSI chip form.

The **half adder** (HA) has two inputs called **Augend** bit A and **Addend** bit B. It has to produce two outputs called **sum** S and **carry** C. The truth Table for half adder is given in table 4.6 and the logic circuit realising the functions S and C is shown in Fig. 4.16. Notice that the output S becomes EXCLUSIVE OR of the inputs A, B while the carry is produced by an AND gate. Likewise, the Truth Table of a half subtractor with two inputs called **minuend** A and **subtrahend** B producing two outputs **difference** D and **borrow** B is shown in Table 4.7. The logic is shown by the side of the Truth Table. The reader is advised to draw the corresponding logic circuit.

Table 4.6 *Truth Table of Half Adder*

Inputs		Outputs	
A	B	S	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Augend is A

Addend is B

Sum S = $A' \cdot B + A \cdot B'$

Carry c = $A \cdot B$

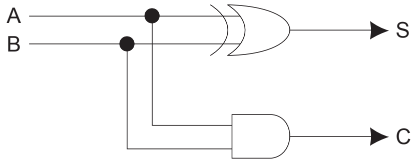


Fig. 4.16 Realisation of a Half Adder

A full adder will have three inputs including an input carry c_i, which is normally the output carry generated in the previous (lower significant) stage. In the stage corresponding to the least significant bit position, the input carry has to be provided by the designer. For all the other stages, input carry is the same as the output carry produced by the previous stage. The Truth Table for a full adder is given in Table 4.8. Output sum S and carry c_0 are mapped alongside and the logic functions also indicated. Two-level realisation of S and c_0 is shown in Fig. 4.17.

Table 4.7 *Truth Table of Half Substractor*

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

Minuend is A

Subtrahend is B

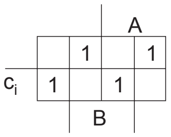
Difference D = $A' \cdot B + A \cdot B'$

Barrow B = $A' \cdot B$

Table 4.8 *Truth Table of Full Adder*

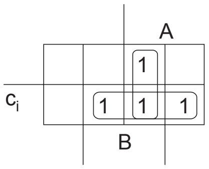
Inputs			Outputs	
A	**B**	**c_i**	**S**	**c_0**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S(A, B, c_i) = \Sigma(1, 2, 4, 7)$ **S Map**



$S(A, B, c_i) = A'(B'c_i + Bc'_i) + A(B'c_i' + Bc_i)$

$c_0(A, B, c_i) = \Sigma(3, 5, 6, 7)$ **c_0 Map**



$c_0 = A(B + c_i) + Bc_i$

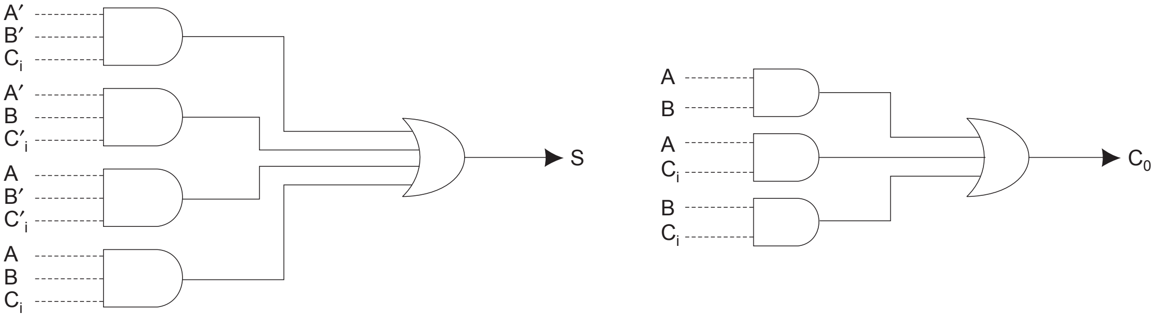


Fig. 4.17 AND-OR realisation of full adder

It is possible to realise a full adder using two half adders. The circuit is shown in Fig. 4.18. Output functions S and c_0 obtained in Fig. 4.18 are easily checked by the analysis given below the figure and verified with the Truth Table of Table 4.8.

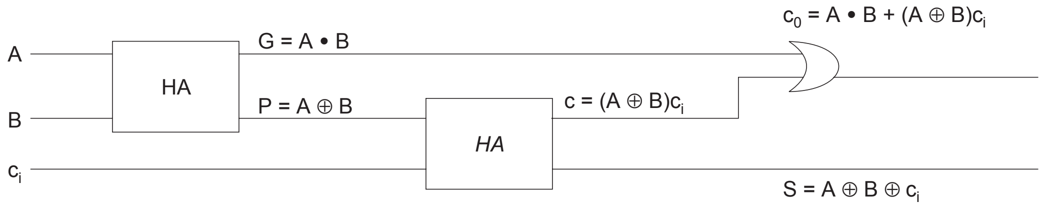


Fig. 4.18 Full adder using two half adders

Check by Analysis

$$\begin{aligned}
S &= A \oplus B \oplus c_i \\
&= A \oplus (B' \, c_i + Bc'_i) \\
&= A' \, B' \, c_i + A' \, Bc'_i + A\,[(B + c'_i)\,(B' + c_i)] \\
&= A' \, B' \, c_i + A' \, Bc'_i + A(Bc_i + B' \, c'_i) \\
&= A' \, B' \, c_i + A' \, Bc'_i + AB' \, c'_i + AB \, c_i \\
&= \Sigma(1, 2, 4, 7)
\end{aligned}$$

$$\begin{aligned}
c_0 &= (A \oplus B) \bullet c_i + A \bullet B \\
&= (\, A' \, B + AB')c_i + A \bullet B \\
&= A' \, Bc_i + AB' \, c_i + AB \\
&= \Sigma(3, 5, 6, 7)
\end{aligned}$$

Full Subtractor

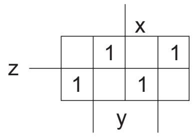
Although **subtraction** is usually achieved by adding the complement of subtrahend to the minuend, it is of academic interest to work out the Truth Table and logic realisation of a **full subtractor.** The reader is advised to get a clear understanding of the Truth Table given in Table 4.9; x is the minuend; y is the subtrahend; z is the input borrow; D is the difference; and B denotes the output borrow. The corresponding maps for logic functions for outputs of the full subtractor namely **difference** and **borrow,** are shown alongside the Truth Table. This would be a good intellectual exercise for readers; it would be educative to work out independently.

Table 4.9 *Truth Table of Full Subtractor*

Inputs			Outputs	
x	***y***	**z**	**D**	**B**
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

D = Σ(1, 2, 4, 7)

D Map

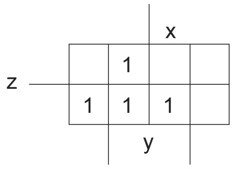


	1		1
1		1	

D = x′y′z + x′yz′ + xy′z′ + xyz

B = Σ(1, 2, 3, 7)

B Map



	1		
1	1	1	

B = x′y′ + x′z + yz

Note: It is common to realise subtraction by adding the complement of the subtrahend to the minuend.

Ripple Carry Adder

Broadly, there are two classes of adder, namely **Serial Adder** and **Parallel Adder**. The **Serial Adder** comprises a single full adder in which the output carry c_{i+1} is fed back to the input as c_i. The augend and addend are fed serially bit by bit to the A and B inputs. The output sum and carry also appear bit by bit serially at the output terminals. This is essentially a slow and sequential process, which is used only in special purpose circuits and instrumentation.

In a parallel Adder of word length ‘n’, all the bits of the Augend A = $a_{n-1}\ a_{n-2} \ldots a_1 a_0$ and the Addend B = $b_{n-1}\ b_{n-2} \ldots b_1 b_0$ together with the input carry c_0 of the least significant stage are simultaneously available in registers and applied as inputs to the respective full adders provided in the system. Obviously, this is a faster process compared to serial addition. The speed is limited by the time required for the carries generated at intermediary stages to propagate to subsequent stages upto the most significant stage. In other words, carries ripple through the intermediary stages. Therefore, this is also called a **Ripple Carry Adder.** A 4-bit binary parallel adder is shown in Fig. 4.19. Notice that it consists of four full adders. The output carry of one stage is wired as the input carry of the next higher significant stage. For the least significant stage, the input carry is to be assigned as 0 or 1 by the designer. The circuit produces SUM outputs for each of the four stages S_0, S_1, S_2, S_3 and the final carry output $c_{4,}$ which feeds the next higher stage. Such a circuit is available in an integrated circuit (IC) chip form.

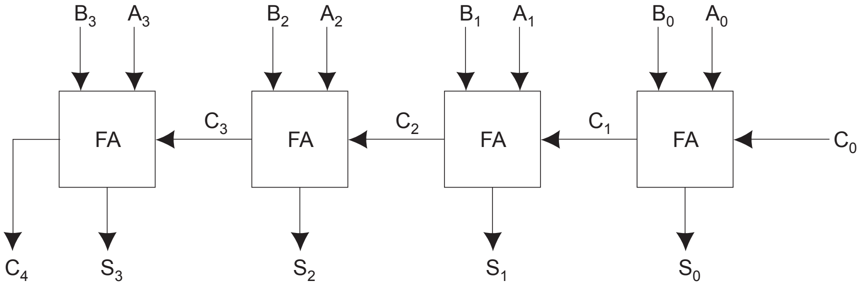


Fig. 4.19 4-bit parallel adder

It is worthwhile to learn at this stage that the adder of Fig. 4.19 may be used as a subtractor too by providing additional logic and making use of c_0 input to specify the mode M. Such an adder-subtractor is shown in Fig. 4.20. Notice that the B inputs are fed through an EXCLUSIVE OR gate to each full adder. If M is assigned 0, the circuit produces the addition A + B. If M is set to 1, the circuit produces A + B′ + 1 which is the same as addition of 2’s complement of B which means A-B.

The speed of a parallel adder is limited by the time required for the carries to propagate through the intermediary stages. That much time must be allowed for the results to settle down before the inputs change. The output carry in a particular stage of full adder clearly depends on all the augend and addend inputs to all the previous stages. Extending the argument to the final (n – 1)th stage, generation of output carry has to consider all the n augend bits, n’ addend bits, and the input carry to the 0th stage. That means a total of (2n + 1) inputs would decide the output carry function. Even if we consider a practical case of adding 16 bit numbers, we need to handle 33 inputs, which means handling 2^{33} combinations. This is a huge figure of more than 8 Billion. computing of output carry for so many possible input combinations and is clearly not practicable. Such a conventional straightforward approach is not feasible.

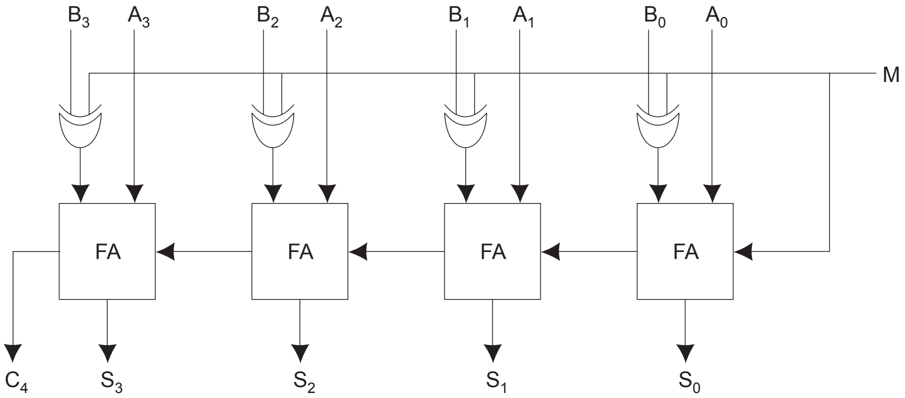


Fig. 4.20 4-bit adder-subtractor

In order to overcome the limitation in speed due to carry propagation delay, it is possible to produce the final output carry directly as a function of the input augend and addend bits. This requires looking ahead of the carries that may be generated or propagated through the intermediary stages, from the least significant stage to the most significant stage. Such a **carry-look-ahead adder** is discussed next. As it is not practical to work out functions of a huge number of variables, the carry-look-ahead scheme is limited to a manageable number of stages, usually four. Such a carry-look-ahead scheme involving four stages of full adders is presented in the following section.

Carry-Look-Ahead Adder

Consider one full adder stage, say the ith stage, of a parallel adder. For the present discussion, look at the full adder of Fig. 4.18. Remember that the Half Adder contains an XOR gate to produce sum and an AND gate to produce carry. If both the bits A_i and B_i are 1s, a carry has to be generated in this stage regardless of whether the input carry c_i is 0 or 1. Let us call this the **generated carry**, expressed as $G_i = A_i \bullet B_i$, which has to appear at the output through the OR gate as shown in Fig. 4.18.

There is another possibility of producing a carry output. EXCLUSIVE OR gate inside the half adder at the input produces an intermediary sum bit—call it P_i—which is expressed as $P_i = A_i \oplus B_i$. Next, P_i and c_i are added to produce the final sum bit S_i and output carry c_{i+1} (shown as c_0 in the figure), which becomes input carry for the (i + 1)th stage.

Consider the case of both P_i and c_i being 1. The input carry c_i has to be propagated to the output only if P_i is 1. If P_i is 0 even if c_i is 1, the AND gate in the half adder will inhibit c_i. We may thus call P_i as the propagated carry as this is associated with enabling propagation of c_i to the carry output of the ith stage which is denoted as c_{i+1} or c_{0i}. For the final sum and carry outputs of the ith stage, we get the following Boolean expressions.

$$S_i = P_i \oplus c_i \qquad \text{where } P_i = A_i \oplus B_i$$

$$C_{i+1} = G_i \oplus P_i c_i \qquad \text{where } G_i = A_i B_i$$

Notice the recursive nature of the expression for the output carry at the ith stage which becomes the input carry for the (i + 1)st stage. By successive substitution, it is possible to express the output carry of a higher significant stage in terms of the applied input variables, namely augend bits and addend bits of the lower stages. This is demonstrated below for a four-stage parallel adder. Remember that the input carry $c_0 = c_f$ for the least significant stage has to be fixed by the designer; $c_f = 0$ or 1 as assigned. The output carry for the 0th stage is denoted by c_1.

$$c_1 = G_0 + P_0\, c_0$$

$$c_2 = G_1 + P_1 c_1 = G_1 + P_1\, G_0 + P_1\, P_0\, c_0$$

$$c_3 = G_2 + P_2\, c_2 = G_2 + P_2\, G_1 + P_2\, P_1\, G_0 + P_2\, P_1\, P_0\, c_0$$

$$c_4 = G_3 + P_3\, c_3 = G_3 + P_3\, G_2 + P_3\, P_2\, G_1 + P_3\, P_2\, P_1\, G_0 + P_3\, P_2\, P_1\, P_0\, c_0$$

The general expression for n stages designated as 0 through (n – 1) would be

$$c_n = G_{n-1} + P_{n-1}\, c_{n-1} = G_{n-1} + P_{n-1}\, G_{n-2} + P_{n-1}\, P_{n-2}\, G_{n-3} + \ldots + P_{n-1} \ldots.\, P_0\, c_0$$

Notice that the final output carry is expressed as a function of the input variables in SOP form, which is a two-level AND-OR or equivalent NAND-NAND function. To produce the output carry for any particular stage, it is clear that it requires only that much time required for the signals to pass through two levels only. In effect, we examined the inputs at all the n stages to produce the output carry for the most significant (n – 1)th stage. Hence, the circuit for look-ahead-carry introduces a delay corresponding to two gate levels, as against the delay introduced in propagating the ripple carry through n stages each contributing a delay of two levels. Notice that the full look ahead scheme requires the use of OR gate with (n + 1) inputs and AND gates with number of inputs varying from 1 through n + 1. For increasing word lengths, it becomes unwieldy. Four-stage carry-look-ahead parallel adders are commercially available in integrated chip form represented as a block diagram in Fig. 4.21. It is possible to have hierarchical levels of look-ahead-group carry scheme to further reduce the addition time and make it faster. Such schemes involve large number of gates.

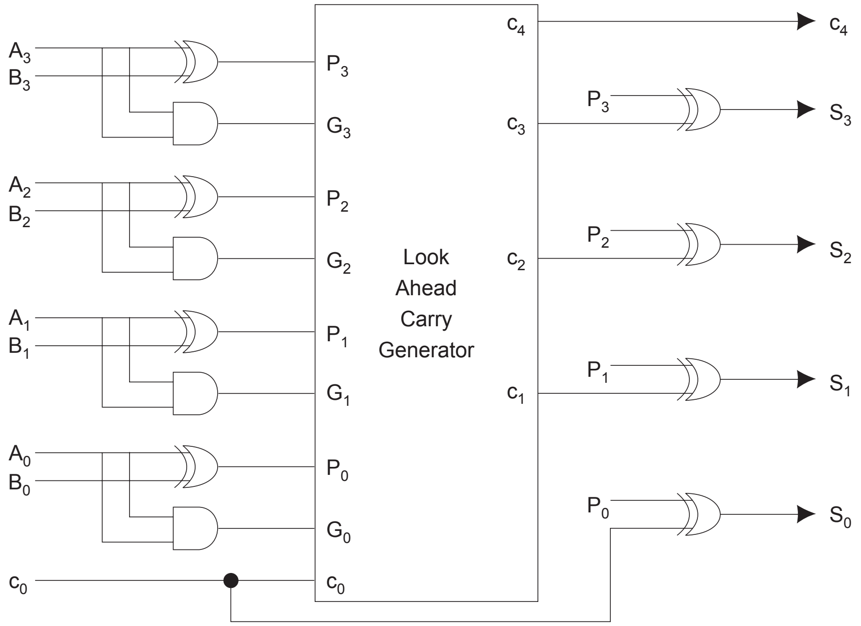


Fig. 4.21 4-bit full adder with look-ahead-carry

Remember that all the sums are produced at the same time. The XOR gate in the first Half Adder produces the P_i signal and the AND gate produces the G_i signal. Thus all P_i and G_i signals are generated in two gate levels. Generation of all output carries in the look-ahead circuit takes two more levels after the P_i and G_i signals settle into their final values. Two more levels produce the sums (see Fig. 4.21).

BCD Adder for Decimal Digits

Calculators, input/output (I/O) devices like keyboards, printers, and so on use decimal digits for their operations. There are instances when arithmetic operations are performed directly in the decimal number system using the BCD code for decimal digits. It is necessary to learn how BCD digits are added. Actually, addition has to be achieved by using only the binary adders; in this case, four full adders. In order to get a feel for addition of BCD digits, a few examples are given below for the purpose of understanding and appreciating the need for modifying the result. The reader should remember that the result of adding two digits and an input carry lies between 0 and 19 (9 + 9 + 1)

Case (i) Result of addition does not exceed 9.

In this case there is no need of modifying the sum bits as they form a valid code and represent the true value of the sum.

Example 4.8 4 + 5 = 9

The machine representation is given below.

$$\begin{array}{rl}
4\ldots 0\,1\,0\,0 & \\
+5\ldots \underline{0\,1\,0\,1} & \\
9\ldots \underline{1\,0\,0\,1} & \text{Sum bits}
\end{array}$$

Case (ii) Result exceeds 9 but ≤ 15.

Consider the following addition in which the sum exceeds 9 but no output carry is generated.

$$\begin{array}{rl}
5\ldots.\,0\,1\,0\,1 & \\
+\,7\ldots.\,\underline{0\,1\,1\,1} & \\
\underline{1\,1\,0\,0} & \text{Sum bits}
\end{array}$$

It is clear that the sum bits do not represent a valid BCD code. There is no carry either. This situation has arisen because we used only 10 out of the 16 possible combinations with four bits in forming the BCD code by skipping six combinations beyond the code for 9. If the modifier 0110 is added, then the result would be a valid sum and an output carry will be generated corresponding to BCD 12 illustrated below.

$$\begin{array}{rrl}
 & 1\,1\,0\,0 & \text{Invalid sum bits} \\
+\text{ Modifier} & \underline{0\,1\,1\,0} & \\
\text{Carry } \boxed{1} & \underline{0\,0\,1\,0} & \text{Valid sum bits}
\end{array}$$

Case (iii) Result ≥ 16, that is, 16, 17, 18 or 19. Consider addition of digits 9 + 9. The machine (abbreviated as M/c) has to produce sum digit of 8 together with an output carry, which goes to the next higher significant stage of addition. Notice that the sum bits produced in performing binary addition represent 2 but not 8. Carry, of course, is produced as required.

$$\begin{array}{rl} 9\;.\;.\;.\;. & 1\;0\;0\;1 \\ +\,9\;.\;.\;.\;. & \underline{1\;0\;0\;1} \\ \text{Carry } 1\;.\;.\;.\;. & \underline{0\;0\;1\;0} \end{array}$$

The sum bits being 0010 is obviously erroneous. They ought to be 1000, representing the digit 8. Carry output is generated as it should be which is sent as input to the next more significant stage. Although the latter part of generating a carry is satisfied, we find that the sum bits are incorrect.

It is therefore logical to modify the result by adding 6 = 0110 to the sum to get the correct result whenever an output carry is generated.

$$\begin{array}{rll} \text{Carry } \boxed{1}\;.\;.\;.\;. & 0\;0\;1\;0 & \text{Erroneous sum} \\ +\text{ Modifier}\;.\;.\;.\;. & \underline{0\;1\;1\;0} & \\ & \underline{1\;0\;0\;0} & \text{Correct sum} \end{array}$$

The reader is advised to work out other examples and appreciate the following conclusion.

Modifier 0110 is to be added to the resulting binary sum whenever an output carry is generated and/or the sum bits represent an invalid code beyond 1001 which in turn implies that the bits Z_4 or Z_2 or both are 1 while $Z_8 = 1$. The corresponding logic expression is $Z_8\ (Z_4 + Z_2) = Z_8\ Z_4 + Z_8\ Z_2$.

Circuit realisation is shown in Fig. 4.22. Notice that one extra adder is used to add the modifier. Also notice that the sum bits before the addition of modifier are denoted as Z_8, Z_4, Z_2, Z_1 while the final sum bits are denoted as S_8, S_4, S_2, S_1 as usual. The output carry generated by the extra adder has to appear at the output but this is ignored as it is taken care of by feeding the modifier logic directly to the OR gate producing the output carry which in turn feeds the B_4B_2 inputs of the extra adder.

4.9 PROGRAMMABLE LOGIC DEVICES

Read-only Memory (ROM)

Read-only memory is essentially a combinational circuit, which realises m output functions of the given n input variables. For the given assignment of input variables, there is a corresponding assignment of the multiple output functions. In this sense, we may call the input variables forming an address of n bits and the corresponding output values forming m-bit word, one bit from each of the m output functions. After going through the examples, the reader will be able to appreciate the power of ROM and get a feel of the elegance of this concept. For now, suffice to say that for every input address of n bits, there will be one (not necessarily unique) output word of m bits fixed by design and hardware wiring. Once designed and wired, the output words cannot be changed. By giving the address inputs, one can read the corresponding output word and hence the name **Read-only Memory (ROM)** in contrast with **Random Access Memory (RAM)** which has **Read & Write (R/W)** facility. At this stage, the reader should realise that ROM is a combinational circuit while RAM is a system comprising sequential circuit elements called **Flip-Flops**, discussed in subsequent chapters.

ROM is basically a decoder with n inputs and 2^n output lines followed by a bank of OR gates. For each assignment of inputs, only one of the output lines of the decoder will be at 1 and the rest will be at 0 level. Each output of a decoder represents a minterm of the input variables. The structure of a typical ROM with three inputs and eight output lines of decoder is shown in Fig. 4.23(a). Notice that all the $2^3 = 8$ minterms, marked 0

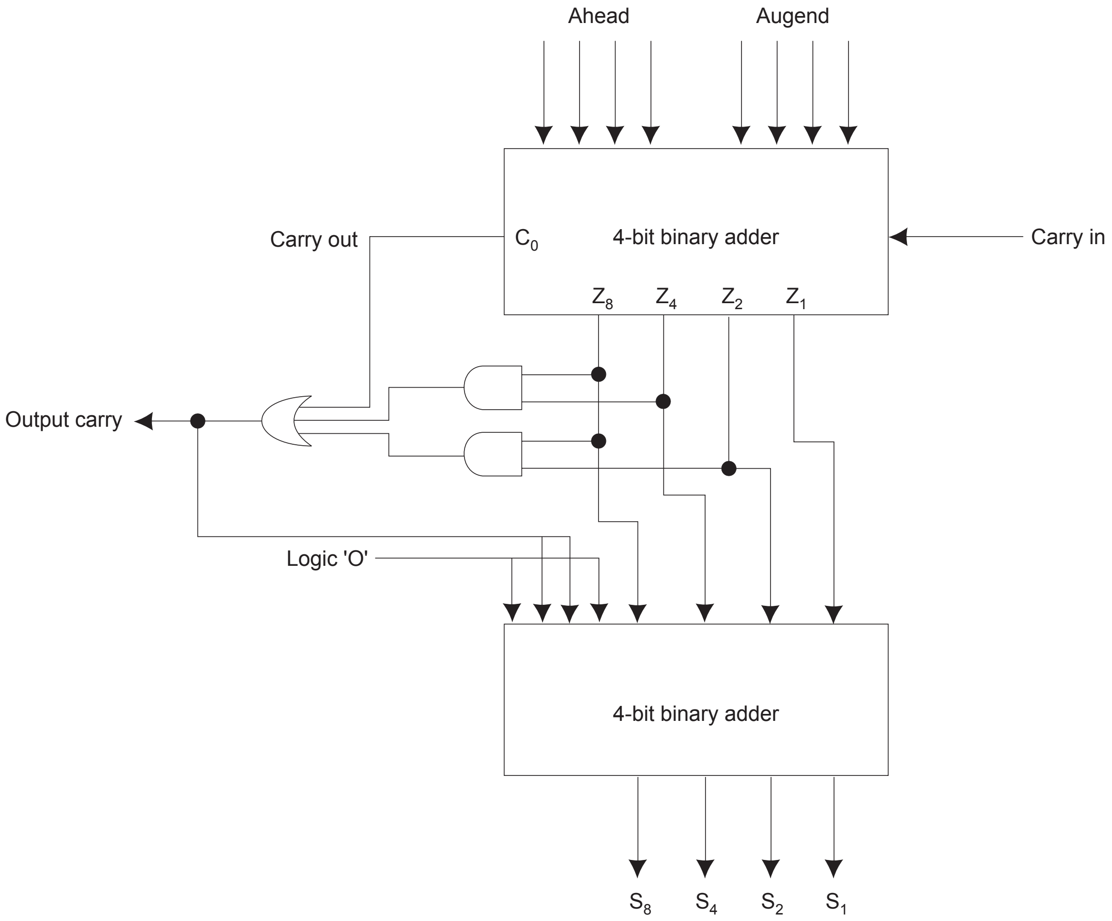
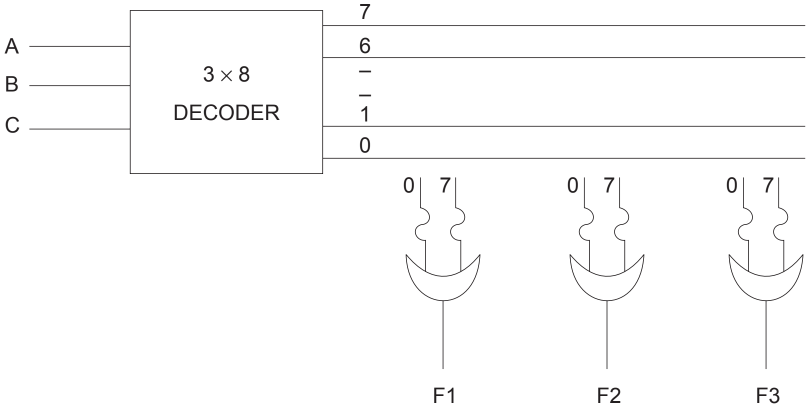


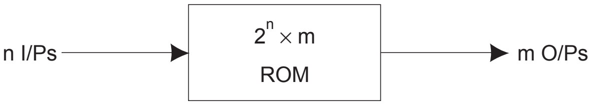
Fig. 4.22 Block diagram of a BCD adder

through 7, are available on the output lines of the decoder. By simply connecting the outputs selectively to OR gates we may realise any function of three variables among the 2^{2^3} = 256 possible 3-variable functions. Hence a ROM can be viewed as a **Universal Logic Module.** Notice that all the minterms are connected to each OR gate through links (also called fuses) Depending on the desired function, the links connecting the desired minterms are kept intact and the remaining links are broken (burnt or blown) The process is called **programming the ROM** for which automated instruments exist.

Block diagram representation of a ROM is indicated in Fig 4.23(b) The bank of OR gates of Fig. 4.23(a) may be realised by using a diode array shown in Fig. 4.24. The links (not shown in the figure) are connected in series with the diodes. The unwanted fuses are blown, breaking the links. Such broken links and the corresponding series diodes are also not shown in the figure. Vertical lines represent the output functions F1, F2, F3 and horizontal lines represent the minterms of a 3-variable function derived as outputs of a 3 × 8 Decoder.



(a) ROM for producing several functions



(b) Block diagram representation of $2^n \times m$ ROM

Fig. 4.23 Structure and block diagram representation of ROM

Example 4.9 Realise the following functions using a ROM of size 8 × 3.

$$F1 = \Sigma(0, 5, 7)$$

$$F2 = \Sigma(1, 3, 6)$$

$$F3 = \Sigma(1, 2, 4, 5)$$

The output functions realised are shown in Fig. 4.24. Notice that the required minterms (horizontal lines) are connected through diodes to the vertical line representing the function. Other diodes at the junctions are disconnected by blowing off the fuses originally connected in series with the diodes. In specifying the size of the ROM, as 8 × 3, note in particular that $8 = 2^n$ where $n = 3$ is the number of inputs and $m = 3$ is the number of output functions. The variables n and m are not related. While n is the number of words stored, m is the word length of each word. This is further elaborated below.

Hitherto, we looked upon ROM as a circuit which can realise any Boolean function. There is an alternative way of looking at it. Look at the horizontal lines in Fig 4.24. They represent the eight decoded addresses marked 0 through 7. Let us now list the binary values of the functions F1, F2, F3, which appear at the crossover points of the grid. Columns can be filled straightway. For instance, F_1 will be 1 for 0th, 5th, 7th rows and 0 in other rows. The blanks in the table are 0s.

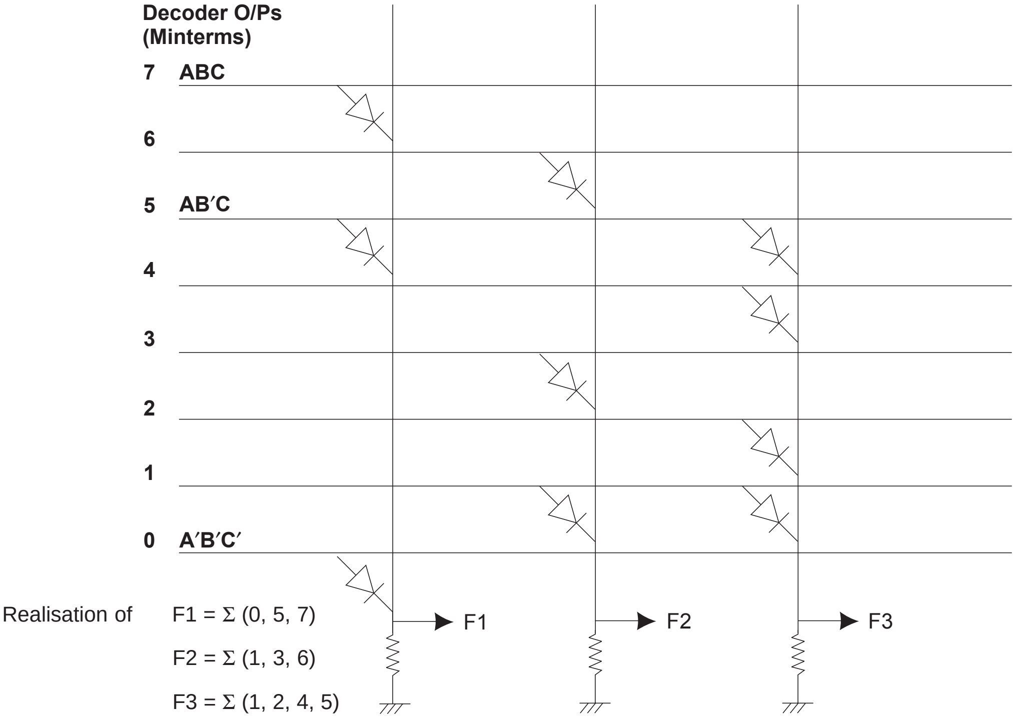


Fig. 4.24 Use of ROM illustrated

Address	F1	F2	F3
7(111)	1		
6(110)		1	
5(101)	1		1
4(100)			1
3(011)		1	
2(010)			1
1(001)		1	1
0(000)	1		

Now read the output words. The 7th word is 100, 5th word is 101, 4th word is 001, 1st word is 011, and so on. Thus, it is easy to see that ROM can be used to store arbitrary words but not necessarily functions. Depending on the location of 1s in the words, the fuses together with the diodes will be retained and others blown or burnt to open. Thus, a ROM is specified as $2^n \times m$ where n is the number of address bits and m is the number of bits in each word, called the “word length” of the stored words.

In order to consolidate the concepts learnt, let us work out some more examples.

Example 4.10

Realise two outputs F_1 and F_2 using a 4 × 2 ROM

$F_1(A_1, A_0) = \Sigma(0, 2)$

$F_2(A_1, A_0) = \Sigma(0, 1, 3)$

In this example, a small ROM of size 4 × 2 is used for the purpose of realising two functions F_1 and F_2 of variables A_1, A_0. Notice that F_1 has two 1s and two 0s, while F_2 has three 1s and a single 0. Notice the broken links in Fig. 4.25(a) To realise F_1, the links 1 and 3 are broken and the links connecting the minterms 0, 2 are left in tact. To realise F_2 only the link 2 is broken to realise 3 minterms designated by 0, 1, 3.

It is advantageous in practice to burn or break as few fuses as necessary. The function may be realised in terms of 1s or 0s, whichever is less, and then inverted if necessary. Such a realisation using AND-OR-Invert gates is shown in Fig. 4.25(b)

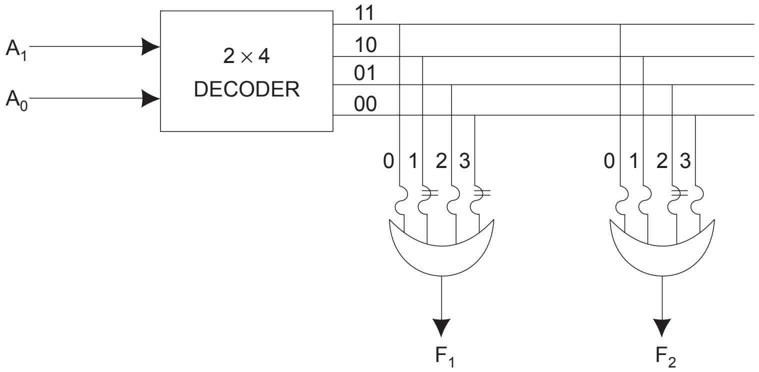
Example 4.11

This example illustrates how a ROM can be used to obtain square of a given input number. The 3-input bits of Fig. 4.26 form a number varying from 0 through 7. It is required to produce the arithmetic square of the given input number. It will range from 0 to 49. Clearly, the output word should contain six bits. The inputs and the corresponding outputs are shown in the Truth Table of Fig. 4.26. It is easily seen that the three binary inputs A_2, A_1, A_0 and the corresponding outputs F_5, F_4, F_3, F_2, F_1 and F_0 need a ROM of size $2^3 \times 6$. The logic functions are indicated by the side of the Truth Table.

The reader must have realised that ROM comprises a decoder followed by an encoder (a bank of OR gates). It is a **Universal Logic Module,** which can be used to realise a number of combinational logic functions. It may also be used to store permanently arbitrarily long binary words. Although small simple examples are used for the purpose of illustration, ROMs are used for the number of inputs typically ranging from 4 to 16. Breaking the links is usually done by the manufacturer, depending on the specifications given by the customer in a prescribed computer-compatible format. This is called **“mask programming”** used by the vendor for producing large numbers. Breaking the links is an irreversible process; if the program has to be altered, the ROM has to be discarded.

Programmable Read-only Memory (PROM)

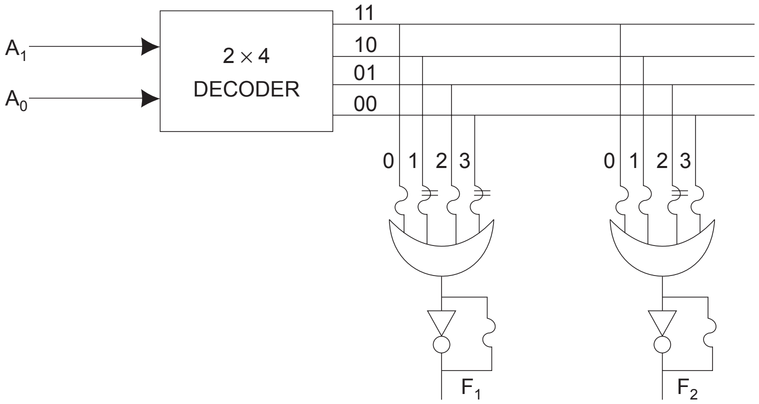
The reader must have realised that a ROM contains a decoder followed by a bank of OR gates, which may be interpreted as an encoder. In a ROM, the inputs are fully decoded to produce 2^n outputs. Each of these 2^n outputs of the decoder is connected through links (fuses) to all the inputs of OR gates provided at the output. Each OR gate at the output realises a function of n variables. Notice that the structure is basically an AND-array followed by an OR-array. In a ROM, the AND-array is fixed while the OR-array has to be programmed to realise the desired output functions. This calls for breaking of unwanted links, which is done by the manufacturer according to specifications given by the customer. This procedure is used for producing large numbers of the same ROM. The program of a ROM is fixed and cannot be altered.



(a) Direct realisation of minterms with broken links

ROM with AND-OR-INVERT Gates

This is an alternative realisation of functions.



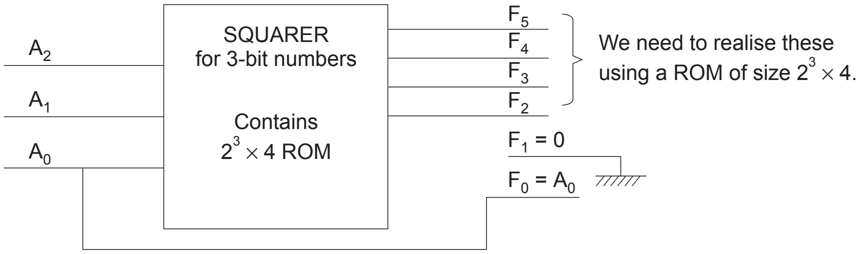
(b) ROM with AND-OR-INVERT structure

Fig. 4.25 Alternative uses of ROM in realising boolean functions

TRUTH TABLE

Dec. Code	Input			Outputs					
	A_2	A_1	A_0	F_5	F_4	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	1
2	0	1	0	0	0	0	1	0	0
3	0	1	1	0	0	1	0	0	1
4	1	0	0	0	1	0	0	0	0
5	1	0	1	0	1	1	0	0	1
6	1	1	0	1	0	0	1	0	0
7	1	1	1	1	1	0	0	0	1

$F5 = \Sigma(6, 7)$
$F4 = \Sigma(4, 5, 7)$
$F3 = \Sigma(3, 5)$
$F2 = \Sigma(2, 6)$
$F1 = 0$
$F0 = \Sigma(1, 3, 5, 7) = A_0$



Note: Input can be treated as address and the output may be treated as the words stored.

Fig. 4.26 ROM used to produce squares

In order to introduce flexibility for the designer, **Programmable Read-Only Memory (PROM)** has become commercially available. It generally uses MOSFET (Metal oxide Semi Conductor Field Effect Transistor) devices in which the logic states 1, 0 are represented by a electrical charge, retained or otherwise. While programming the OR-array, unwanted fuses are burnt by passing electric current through the specified fuses only. For this job, special instruments called PROM-programmers are used. PROMs are convenient for testing in the development stages and the final design is implemented by using ROM.

Over a period of time, flexibility has increased and some more types of ROMs have evolved. **Erasable and Programmable Read-Only Memory (EPROM)** uses a source of ultraviolet light passed through a built-in window on the chip. When exposed to ultraviolet light, all the charges are discharged and the EPROM can be used again for programming to realise different functions. The only drawback of this is that it requires an ultraviolet light source.

Electrically Erasable and Programmable Read-Only Memory (EEPROM) is the next development. EEPROMs are commercially available in the form of integrated circuit chips. EEPROMs are erased electrically. They are extensively used for experimentation in the development stage. After final testing in the laboratory, the ROM is built and integrated into the equipment.

ROM uses all possible 2^n fundamental products (minterms) of the n inputs. Owing to the many don't care combinations in many situations, a good number of products in ROM are not used. Thus, the full capacity of a ROM is hardly ever utilised.

Programmable-Logic-Array (PLA)

In the quest for improving flexibility and efficiency, other **Programmable Logic Devices (PLDs)** have evolved. **Programmable-Logic-Array (PLA)** and **Programmable-Array-Logic (PAL)** have found extensive use in logic design. These are discussed below.

PLA produces only the required products. The structure of a PLA is shown in Fig. 4.27. Notice that both the AND-array and the OR-array are programmable by burning the unwanted fuses shown in the figure. The size of the PLA is specified as $n \times k \times m$, where n is the number of inputs, k is the number of products produced by the AND-array and m is the number of OR gates at the output. There is provision to realise the functions in MSP (Minimal Sum of Products) or complemented MSP form, whichever contains the minimum number of product terms. The following example demonstrates the relative power of the PLA versus ROM.

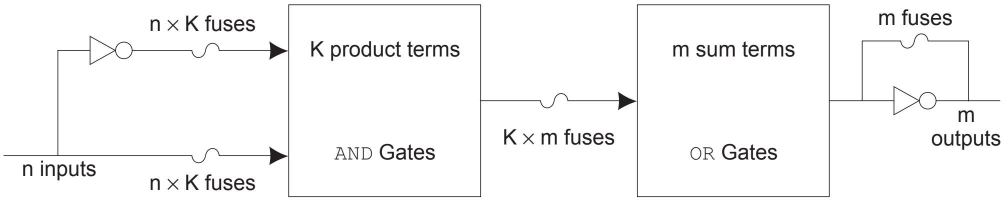


Fig. 4.27 Structure of PLA

Example 4.12

No. of inputs n = 16

No. of AND gates k = 40 products

No. of OR gates m = 100 functions

No. of fuses to be programmed for PLA $= 2\,nk + km + m$ (see Fig. 4.27)

$= 2 \times 16 \times 40 + 40 \times 100 + 100$

$= 1280 + 4000 + 100$

$= 5380$

Note: m different functions of n variables can be realised if each function has not more than k minterms. If $k = 2^{n-1}$, then all functions can be realised.

$$\text{No. of fuses to be programmed for ROM} = \underset{\substack{\uparrow \\ \text{Full decoding}}}{2^{16}} \times 100 + 100 = 64\text{ k} \times 100 \text{ (Huge)} + \underset{\substack{\uparrow \\ \text{for complement realisation}}}{100}$$

PLA Program Table

Programming the PLA means specifying the paths indicated in Fig. 4.27, starting from inputs to AND gates, then to OR gates and finally to inverters to reach the outputs. This is illustrated with the following example.

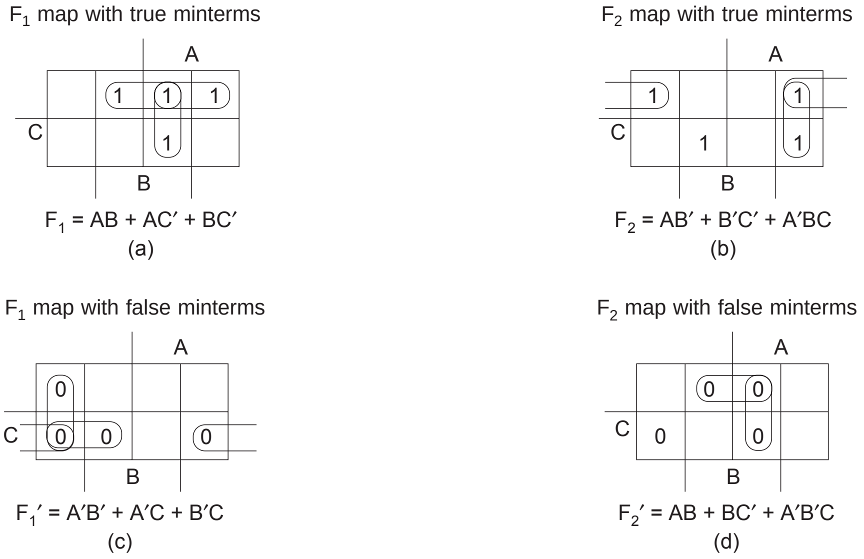
Example 4.13

Obtain the PLA program table to realise the following functions.

$F_1(A, B, C) = \Sigma(2, 4, 6, 7)$

$F_2(A, B, C) = \Sigma(0, 3, 4, 5)$

The first step is to get the MSP form of the functions and list the product terms required considering only the true and false minterms. Notice in Fig. 4.28(c) and (d) the 0s are temporarily considered as 1s and the minimal sum of products form is found and the corresponding function is complemented. Look at the maps and the corresponding minimal expressions given in Fig. 4.28. A little reflection reveals that it would be economical to realise F_1 with true minterms and F_2 in its complement form as two of the products are common. Any other choice would require a larger number of products. Minimising the number of products in the PLA for a large number of functions is a special topic which requires special techniques. The leaner should realise that this simple



F_1 map with true minterms

$F_1 = AB + AC' + BC'$

(a)

F_2 map with true minterms

$F_2 = AB' + B'C' + A'BC$

(b)

F_1 map with false minterms

$F_1' = A'B' + A'C + B'C$

(c)

F_2 map with false minterms

$F_2' = AB + BC' + A'B'C$

(d)

PLA Program Table

Product Terms	**Inputs**			**Outputs**	
	A	**B**	**C**	**F_1**	**F_2**
AB … 1	1	1	-	1	1
AC′… 2	1	-	0	1	-
BC′ … 3	-	1	0	1	1
A′ B′ C ... 4	0	0	1	-	1
	True or Complement Form?			T	C

(e) PLA Program Table

Fig. 4.28 Illustrating realisations F_1 and F_2 using PLA

example is chosen only for the purpose of illustration. More than 10 inputs and 100 product terms are common with commercially available PLAs. In the PLA program table shown in Fig. 4.28(e), uncomplemented variables are shown by 1s and complemented variables indicated by 0s in the input columns. Notice that the column giving products is no longer necessary but retained only for the purpose of reference. Similarly, the columns of F_1 and F_2 are filled with only 1s corresponding to the products and whether they represent true form (T) or complement form (c) is indicated in a separate row. This completes the PLA program table of Fig. 4.28(e)

Programmable-Array-Logic (PAL)

While ROM uses a fixed AND-array followed by a fixed OR-array, PROM contains a fixed AND-array followed by programmable OR-array, giving flexibility to the designer. In PLA, further flexibility is offered to the designer by making both the AND-array, and OR-array programmable. Designers found advantage in having flexibility in programming AND-array while fixing the OR-array. Such a device is called Programmable-Array-Logic (PAL) and is presented below.

Note that ROM realises combinational circuits in the form of minterms. PLA/PAL implements the combinational functions in the form of products (not necessarily minterms) contained in the MSP form.

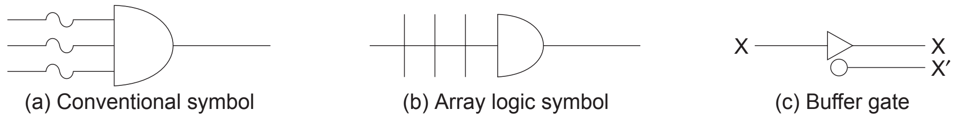
For convenience in drawing the circuits for programmable logic devices, designers use the notation shown in Fig. 4.29. Vertical lines denote the inputs and horizontal lines feed the AND gates shown in Fig. 4.30, 4.31 and 4.32. Crosses (Xs) are used to indicate inputs connected to the AND/OR gates through fusible links, whether fixed or programmable. PAL with 4-inputs and 3-wide AND-OR structure is indicated in Fig. 4.30. In this example, each function can have three minterms or product terms. To get a feel for designing with a PAL, consider the example of the code converter for BCD to Excess-3 code discussed in Section 4.6. The functions to be realised are reproduced below.

$$w = a + bc + bd, \qquad x = b' c + b' d + bc' d'\text{'}$$

$$y = c' d' + cd, \qquad z = d'$$

Notice that each of the output functions requires at most three products of four input variables. At most three AND gates may be required for each of the output functions. Notice that the horizontal lines feeding each of the AND gates represent the inputs marked by Xs at the crossover points. The vertical lines represent the input variables and their compliments. Vertical line 1 represents input a, vertical line 2 represents a′. Similarly 3 and 4 represent b and b′, 5 and 6 represent c and c′, and finally 7 and 8 represent d and d′ respectively. Xs represent the fuses left intact at their places. Where there are no Xs marked, the corresponding fuses have to be blown off. Fig. 4.30 shows programming the PAL for realising the above functions w, x, y, z.

Notation



(a) Conventional symbol (b) Array logic symbol (c) Buffer gate

Fig. 4.29 Symbols used for drawing programmable logic devices

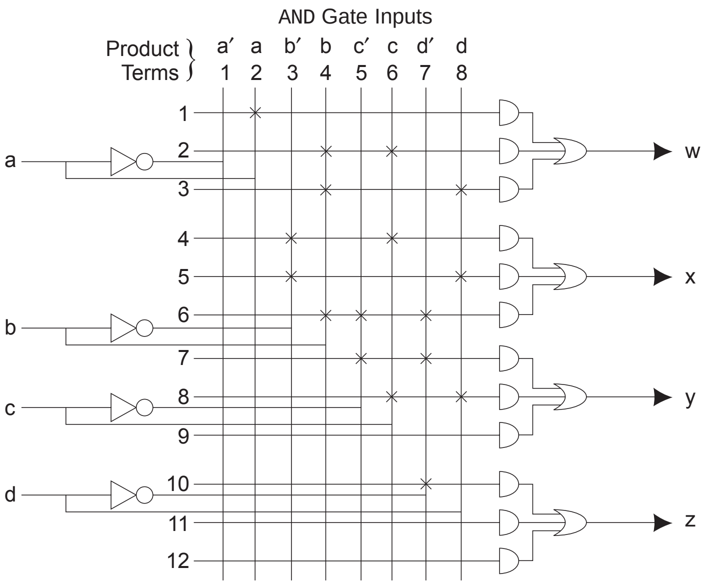


Fig. 4.30 PAL with four inputs and four outputs

Programming a PAL In Fig. 4.31, Xs indicate fusible links to the AND gate on the input side, while the Xs on the output side are fixed connections to the OR gates.

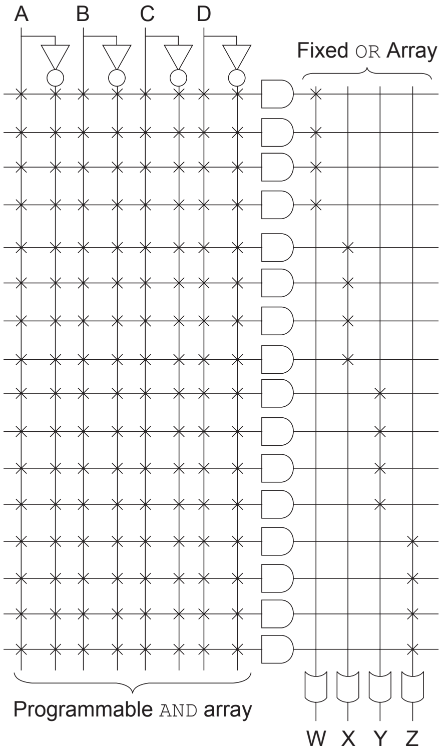


Fig. 4.31 Structure of PAL

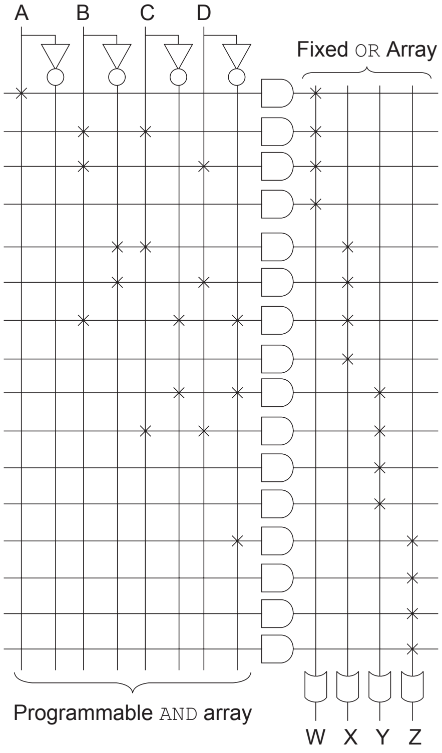


Fig. 4.32 Illustration for programming of PAL

The general structure of a PAL with four inputs and four outputs is shown in Fig. 4.31. Notice that there are 16 AND gates, which implies only 16 chosen products of not more than four variables A, B, C, D. Inputs to the OR gates at the output are fixed as shown by Xs marked on the vertical lines. The inputs to the AND gates are marked on the corresponding line by the Xs. Removing the X implies blowing off the corresponding fuse which, in turn, implies that the corresponding input variable is not applied to the particular AND gate. Removal of Xs to program the input AND array, while retaining all the Xs in the output fixed OR array must be noted as illustrated in Fig. 4.32. The PAL shown in Fig. 4.32 realises the functions given in the example. Notice how the inputs and outputs are marked differently in Fig. 4.30 as against Figs. 4.31 and 4.32. They are equivalent. Remember that Xs on both A and A′ vertical lines imply 0 output at the corresponding AND gate.

PAL programming table The PAL program table is similar to that of PLA except that only the inputs of the AND gates have to be programmed while the inputs to the OR gates are fixed. Further, PAL programming is simpler than PLA as there is no need to minimise the number of products by considering whether to take the true form or the complement form of the output functions. Unlike PLA, a product term cannot be shared by two or more OR gates at the output. Hence, we go straight indicating the inputs to the AND gates. One finer detail is that PAL may have a provision to use some of the outputs as inputs and feed them to other OR gates.

To brush up the concepts learnt so far, consider the following example.

Example 4.13 Realise the following functions using a PAL with four inputs and 3-wide AND-OR structure.

F_1 (A, B, C, D) = Σ(1, 3, 7, 9, 11, 12–15)

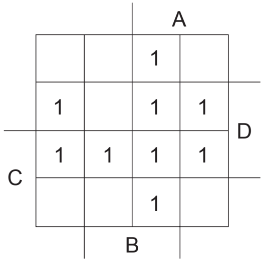
F_2 (A, B, C, D) = Σ(0–3, 7, 10, 12, 13)

F_3 (A, B, C, D) = Σ(0–3, 12, 13)

F_4 (A, B, C, D) = Σ(4–7, 9, 12–15)

The first step is to obtain the MSP form of all the given functions as shown below the corresponding maps in Fig 4.33. Note that each section comprises three AND gates feeding the OR gate in the given PAL. Notice that F_2 has four product terms but the given PAL device has provision for three products only as inputs to OR gates. Some manipulation becomes necessary. We need to use one extra unused section. We add three products in one section and then add the result to the fourth product. This, however, requires an extra input line.

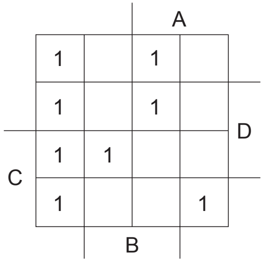
F_1 map



		1	
1		1	1
1	1	1	1
		1	

F_1 = (A, B, C, D) = AB + CD + B′D

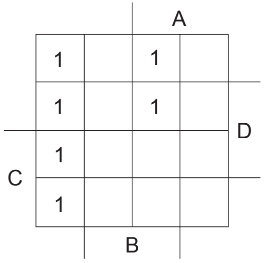
F_2 map



1		1	
1		1	
1	1		
1			1

F_2 = (A, B, C, D) = A′B′ + ABC′ + A′CD + B′CD′
= F_3 + A′CD + B′CD′

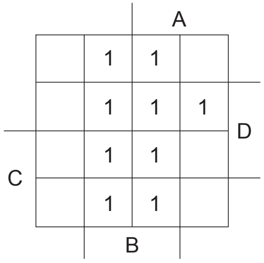
F_3 map



1		1	
1		1	
1			
1			

F_3 = (A, B, C, D) = A′B′ + ABC′

F_4 map



	1	1	
	1	1	1
	1	1	
	1	1	

F_4 = (A, B, C, D) = B + C′D

PAL Programming Table

Product term	AND inputs					Outputs
	A	B	C	D	F_3	
1	1	1	-	-	-	F_1 (A, B, C, D) = AB +
2	-	-	1	1	-	CD+
3	-	0	-	1	-	B′ D
4	-	-	-	-	1	F_2 (A, B, C, D, F_3) = F_3 +
5	0	-	1	1	-	A′ CD+
6	-	0	1	0	-	B′ CD′
7	0	0	-	-	-	F_3 (A, B, C, D,) = A′ B′ +
8	1	1	0	-	-	ABC′
9	-	-	-	-	-	
10	-	1	-	-	-	F_4 (A, B, C, D,) = B + C′ D
11	-	-	0	1	-	
12	-	-	-	-	-	

Fig. 4.33 Illustrating PAL program table

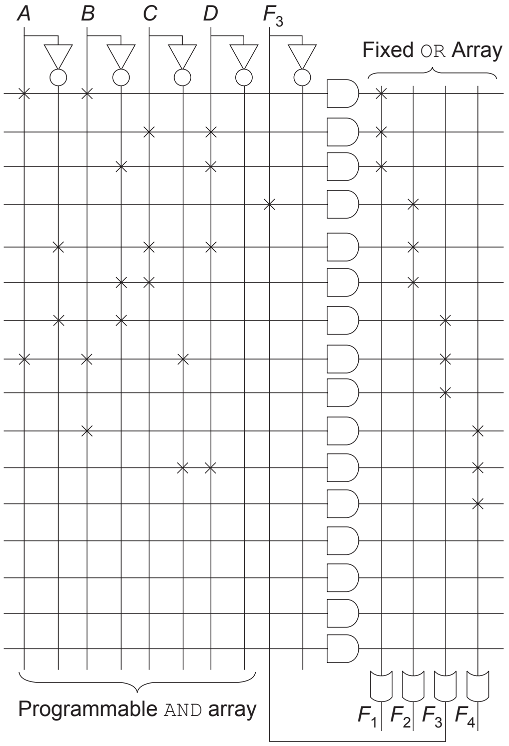


Fig. 4.34 Realisation of the example functions using PAL

In the present example, notice that F_3 is included in F_2 and it covers two products. We take advantage of this fact by connecting F_3 to a spare input line and realise the function F_2. The PAL program table is shown in Fig. 4.33. The actual realisation is shown in Fig. 4.34. In the AND array, Xs indicate links left intact, while their absence indicates fuses blown off at the crossover points. The Xs in the OR array indicate fixed connections.

Programmable Logic Devices–A Comparison

ROM, PROM, PLA, and PAL have been discussed. These devices are used for synthesising multiple-output functions. All these basically comprise of an AND array followed by an OR array. The programming flexibility for the user in each case is summarised below.

PLD	Programming flexibility—fixed or programmable	
	AND Array	OR Array
ROM	Fixed	Fixed
PROM	Fixed	Programmable
PLA (FPLA)	Programmable	Programmable
PAL	Programmable	Fixed

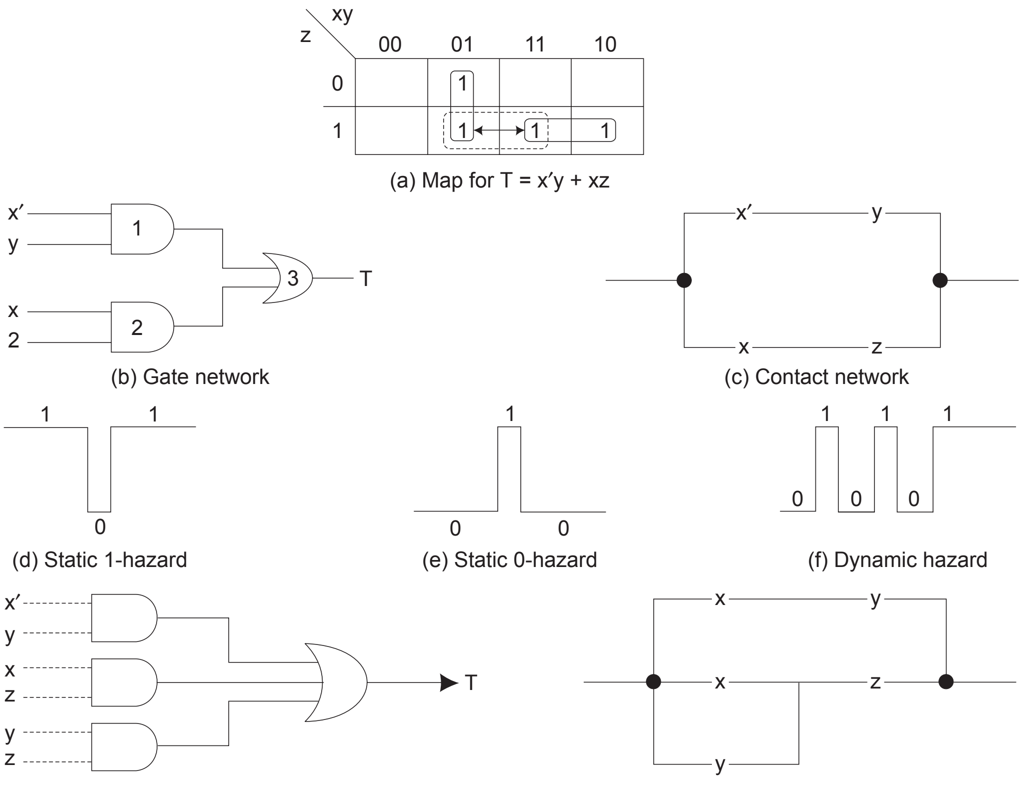
4.10 HAZARDS AND HAZARD–FREE REALISATIONS

Hazards in combinational logic circuits mean the possibility of a malfunction caused by delay in propagation of signals. There are two types of hazards namely **“Static Hazard”** and **“Dynamic Hazard”.**

Suppose all the inputs to a logic circuit except one remain at their assigned levels and only one input, say x, changes either from 0 to 1 or from 1 to 0. Look at the gate circuit of Fig. 4.35(b) Consider the situation when y = 1, z = 1; x is changing from 0 → 1. The output function has to remain at 1 regardless of whether x is 0 or 1. In such a situation, the output may become momentarily 0 for a short interval. Such a transient is called a **“glitch”** or a spurious **“spike”**, which is caused by the hazardous behaviour of the logic circuit. If the output is expected to be at 1 regardless of the changing variable, it is called a **“Static 1-hazard”** shown in Fig. 4.35(d) If the output is designed to be at 0 regardless of the changing variable, the spurious 1 level for a short interval is referred to as a **“Static 0-hazard”** shown in Fig. 4.35(e) Static 1-hazards and 0-hazards refer to improper behaviour of the circuit for adjacent input combinations because of delays in gates, which are not considered at the time of designing the switching function.

“Dynamic hazards” occur when the output changes for two adjacent input combinations. While changing, the output should change only once but it may change three or more times at short intervals because of differential delays in several paths. Dynamic hazards occur only in multilevel circuits. A typical output change is shown in Fig. 4.35(f) We consider only single input changes as in other cases it becomes almost impossible to prevent hazards.

In asynchronous sequential machines, delays in logic gates might cause improper operation by causing the machine to land in a wrong state. Such situations are referred to as **“Essential Hazards”** discussed in a subsequent chapter. In synchronous sequential machines, hazards caused by transient behaviour are of no consequence as the clock speed is determined to allow all signals to settle in their steady static values before the next change of inputs.



(a) Map for T = x′y + xz

(b) Gate network

(c) Contact network

(d) Static 1-hazard

(e) Static 0-hazard

(f) Dynamic hazard

(g) Gate network

(h) Contact network

Fig. 4.35 Networks containing static hazards

Static Hazards

Consider the function F(x, y, z) = $\Sigma(2, 3, 5, 7)$ mapped in Fig. 4.35(a). The minimal sum of products (SOP) realisation with logic gates is shown in Fig. 4.35(b), while the implementation with contacts is shown in Fig. 4.35(c).

Consider the situation when y = 1, z = 1 and only x is changing from 0 to 1. Look at the map. The output F has to remain at 1 by design. Now look at the gate network. When x is 0, the output of gate 1 is 1; hence F is 1. When x changes to 1, the output of gate 1 becomes 0 but the output of gate 2 becomes 1 to maintain F at 1. For the change in x from 0 to 1, if gate 2 responds faster than gate 1, it is luck that F will be 1 as expected. If we are unlucky, gate 1 will be faster and its output becomes 0 earlier than gate 2, and the output of gate 2 continues to be 0 resulting in F becoming 0 momentarily; a little later, the sluggish gate 2 gives the output 1 and F will attain its correct value 1. This erratic behaviour is shown in Fig. 4.35(d) and is called a **static 1-hazard** marked by an arrow in the map. Clearly we should not allow the behaviour of circuits to depend on chance; we should ensure that the functions do not contain hazards. Hazard-free realisation is, therefore, discussed next. If you consider a sum of products realisation, you would end up in a **static 0-hazard** shown in Fig. 4.35(e). With contact networks, the former is called a **"tieset hazard"** and the latter a **"cutset hazard".**

Hazard-free Realisation

Notice in the map of Fig. 4.35(a) that the 1-hazard arises because two adjacent 1s are covered by different **sub-cubes (a cluster of 2^m adjacent cells, each adjacent to m cells).** Minimal realisations are most vulnerable to hazards. If we want to ensure that this pair of adjacent 1s is covered by the same sub-cube shown dotted on the map, we need to add one more AND gate shown in Fig. 4.35(g). The corresponding contact network will have one more path shown in Fig. 4.35(h). This realisation will now have no 1-hazard but the function contains a redundant term, yz.

The removal of 1-hazards in a function by the addition of sub-cubes does not imply the removal of 0-hazards in the function. This concept is illustrated in the following example.

Example 4.14

Realise the following switching function by a hazard-free logic gate network.

$T(x, y, z) = \Sigma(0, 2, 4, 5)$

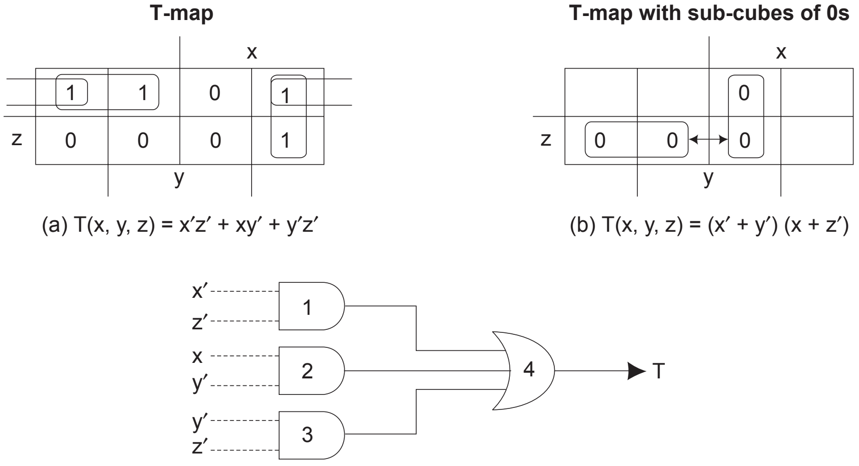
This function is mapped in Fig. 4.36(a). If you consider realisation in SOP form, the function is expressed as

$$T = xy' + x'z' + y'z'$$

This SOP form is hazard-free because every pair of adjacent 1s is covered by the same sub-cube and in order to satisfy this constraint, a redundant sub-cube y′z′ had to be added.

Suppose we wish to realise the same function in POS form by choosing the sub-cubes marked in Fig. 4.36(b). There will be a hazard marked by arrows, if you realise it as

$$T = (x' + y')(x + z')$$



(a) T(x, y, z) = x′z′ + xy′ + y′z′

(b) T(x, y, z) = (x′ + y′) (x + z′)

(c) Hazard-free realisation of the function

Fig. 4.36 A function with no 1-hazards but having a 0-hazard

Notice that merely expanding the POS form and ignoring the term $xx' = 0$ results in the same SOP form as above, which is hazard-free. The hazard in the POS form must be attributed to ignoring the terms like $xx' = 0$.

Naturally, we conclude that the hazard-free function has to be realised in its original form without resorting to simplification or factoring. The realisation is shown in Fig. 4.36(c)

Let us now examine whether a static 0-hazard marked in Fig. 4.36(b) exists in the gate circuit realised based on hazard-free SOP expression.

Notice that $y = 1$, $z = 1$ and x is changing in this situation. Take a look at the gate circuit of Fig. 4.36 (c). The function remains at 0 regardless of the changing variable x because $y = 1$ ensures 0 at the outputs of gates 2 and 3, and $z = 1$ ensures that the output of gate 1 is 0. We therefore conclude that a gate circuit realising hazard-free function with no 1-hazards does not have 0-hazards too at the circuit level. This is valid for 2-level realisations in general. The proof is left to the reader as an exercise. **A 2-level circuit with no static 1 (0)-hazards will not have static 0 (1)-hazards too. But this is not true with the functions** as in the case of T mapped in Fig. 4.36(a) and (b).

Another important observation is that the dual of a hazard-free circuit is also completely hazard-free.

SUMMARY

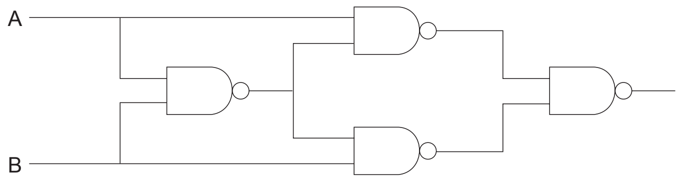
Various techniques of designing combinational logic circuits to suit different situations are presented and illustrated. The elegance of using multiplexers in realising Boolean functions has been highlighted. Use of programmable logic devices such as ROM, PROM, PLA, PAL has been illustrated. Finally, hazards and hazard-free design has been covered.

KEY WORDS

- AND-OR realisation
- NAND–NAND realisation
- Encoder
- Multiplexer
- Demultiplexer
- Priority encoder
- Modular expansion
- Code converter
- Magnitude comparator
- Chip enable
- Chip select
- Data selector
- Time division multiplexing
- Strobe
- Multiple output circuits
- Adder
- Subtractor
- ROM
- RAM
- PROM
- PLA
- PAL
- Static hazard
- Glitch
- Hazard-free design

REVIEW QUESTIONS

1. The logic network shown in the figure below realises the function represented by



a) A $\boxed{\text{XNOR}}$ B   b) A $\boxed{\text{XOR}}$ B

c) A $\boxed{\text{NOR}}$ B   d) A $\boxed{\text{OR}}$ B

2. Neatly draw the combinational circuit for realising $f(x, y, z) = \Sigma(0, 1, 5, 7)$ using AND, OR, Inverter gates.
3. Given $f_1(w_1 x_1 y_1 z) = \Sigma(0, 1, 2, 3, 4, 5, 6, 7)$

   $f_2(w_1 x_1 y_1 z) = \Sigma(8, 9, 10, 11, 12, 13, 14, 15)$

   Find $G = f_1 + f_2$

   Find $H = f_1 . f_2$
4. Which logic gate is used in parity checkers?
5. A decoder with an Enable I/P (E) can function as a demultiplexer, if data is fed to E line. (True/False)
6. Decoder and OR gate can produce any Boolean function. (True/False)
7. For the following two Boolean functions design a circuit with a decoder and external gates.

   $F1 = \Sigma(1, 4, 6)$; $F2 = \Sigma(2, 3, 6)$
8. A logic circuit, which is used to change a BCD number into an equivalent decimal number is

   a) Decoder   b) Encoder   c) Multiplexer   d) Code converter
9. MUX of 2^n to 1 can produce any Boolean function of n variables by inputting 1s on the I/P lines corresponding to minterms formed by select lines (n variables). (True/False)
10. Enable input for MSIs is used for modular expansion. (True/False).
11. Using an 8 × 1 MUX realise the function $f(a, b, c, d) = \Sigma(1, 3, 11, 13)$.
12. Realise NAND function using 2 × 1 multiplexer.
13. A multiplexer is also known as

    a) Encoder   b) Decoder   c) Data selector   d) Multiplier
14. A demultiplexer is used to

    a) Perform arithmetic division

    b) Select data from several inputs and route it to a single output

    c) Steer the data from a single input to one of the many outputs

    d) Perform parity checking.
15. How many select lines are contained in a multiplexer with 1024 inputs and one output ?

    a) 512   b) 258   c) 64   d) 10

16. A combinational logic circuit which is used to send data coming from a single source to two or more separate destinations is called

a) Decoder b) Encoder c) Multiplexer d) Demultiplexer

17. A multiplexer is also known as a

a) Coder b) Decoder c) Data selector d) Multivibrator

18. What is the largest number of data inputs which a data selector (multiplexer) with two control inputs can handle?

19. Which of the following logic circuits takes data from a single source and distribute it to one of the several output lines ?

a) Multiplexer b) Data multiplexer c) Demultiplexer d) Parallel counter

20. A demultiplexer is also known as a

a) Data selector b) Data distributor c) Multiplier d) Encoder

21. A multiplexer with four select bits is a

a) 4:1 multiplexer b) 8:1 multiplexer c) 16:1 multiplexer d) 32:1 multiplexer

22. Which of the following devices selects one of several inputs and transmits it to a single output

a) Decoder b) Multiplexer c) Demultiplexer d) Counter

23. The number of select lines, m, required to select one out of n input lines is

b) $m = \log_2 n$ b) $m = \log n$ c) $m = \ln.n$ d) $m = 2^n$

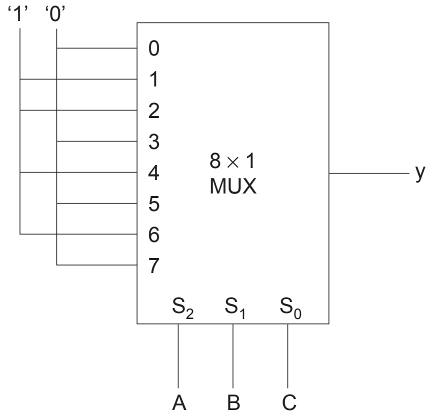
24. A combinational logic circuit which is used when data has to be sent from two or more sources through a single transmission line is known as a

a) Encoder b) Decoder c) Multiplexer d) Demultiplexer

25. What is the largest number of data inputs which a data selector with two control inputs can have?

a) Two b) Four c) Eight d) Sixteen

26. For the multiplexer circuit shown in the figure below, y is given by



a) $\Sigma m(0, 3, 5, 7)$ b) $\Pi M(1, 2, 4, 6)$ c) $\Sigma m(1, 2, 4, 6)$ d) $\Pi M(1, 3, 5, 6)$

27. The difference output in a full subtractor is the same as______ .
    a) Difference output of a half subtractor
    b) Sum output of a half adder
    c) Sum output of a full adder
    d) Carry output of a full adder
28. Give two differences between a serial adder and a parallel adder.
29. Draw the block diagram of a full adder using two half adders and one
    OR gate.
30. Draw the logic diagram of a half subtractor using NOR gates only.
31. Which of the following logic circuits accepts two binary digits on inputs and produces two binary digits, a sum bit and a carry bit, on its outputs?
    a) Full adder b) Half adder c) Decoder d) Multiplexer
32. Draw the circuit diagram of a 2-bit adder-subtractor.
33. A certain 4-bit binary adder produces output sum denoted by S_8, S_4, S_2, S_1 and output carry C_0. This needs to be converted into a BCD adder. What is the logic function which decides whether or not to add the modifier to the output sums?
    a) $C_0S_8 + S_4 + S_2$ b) $C_0 + S_8 (S_4 + S_2)$ c) $C_0 + S_8 . S_4 .S_2$ d) $C_0 + S_8 (S_4 +S_1)$
34. What is the propagation delay of a 4-bit binary parallel adder in terms of single gate delay 'τ' in the case of (a) ripple-carry-adder and (b) carry-look-ahead-adder?
35. For the addition of two 4-bit numbers?
    i) How many full adders are required?
    ii) How many outputs are there?
    iii) How many inputs are there?
36. How many full adders are required to construct an m-bit parallel adder?
    a) m/2 b) m – 1 c) m d) m + 1
37. Draw a 4-bit binary adder-subtracter.
38. What is the advantage of look-ahead-carry generator?
39. Parallel adders are
    a) Combinational logic circuits b) Sequential logic circuits
    c) Both of the above d) None of the above
40. A parallel adder requires more hardware than a serial adder. (True/False)
41. In which of the following adder circuits is the carry-ripple delay eliminated?
    a) Half adder b) Full adder
    c) Parallel adder d) Carry-look-ahead adder
42. To secure a higher speed of addition, which of the following is the preferred solution?
    a) Serial adder b) Parallel adder
    c) Adder with a look-ahead-carry
    d) Adder with rippling group-carry with carry-look-ahead in each group

43. How many inputs and outputs does a full adder have?

a) Two inputs, two outputs b) Two inputs, one output

c) Three inputs, two outputs d) Two inputs, three outputs

44. How many inputs and outputs does a full subtractor circuit have?

a) Two inputs, one output b) Two inputs, two output

c) Two inputs, three outputs d) Three inputs, two outputs

45. A full adder can be realised using

a) One half adder, two OR gates b) two half adders, one OR gate

c) two half adders, two OR gates d) two half adders, one AND gate

46. A ROM of size $2^n \times m$ can store

a) 2^n words each of m bits b) 2^m words each of n bits

c) $m \times n$ bits d) m + n bits

47. It is desired to have a 64 × 8 ROM. The ROMs available are of 16 × 4 size. The number of ROMs required will be

a) 16 b) 8 c) 32 d) 64

48. A programmable logic array has n = 8 inputs, k = 20 product terms, and m = 100 outputs. What is the number of fuses to be programmed?

a) 2420 b) 16000 c) 128 d) 2260

49. A certain programmable array logic has n = 32 inputs. It has to be programmed with k = 50 product terms and m = 20 outputs. What is the number of fuses to be programmed?

a) $2^{32} \times 50$ b) 3200 c) $2^{50} \times 20$ d) 4220

50. Draw the schematic to realise the following functions using 4 × 2 ROM.

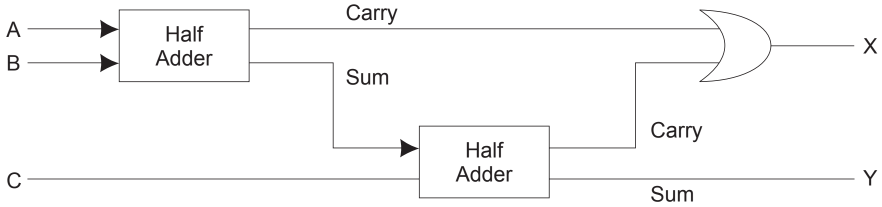
$F_1(A_1, A_0) = \Sigma(1, 2, 3)$

$F_2(A_1, A_0) = \Sigma(0, 2)$

51. The number of programmable links in a $n \times k \times m$ PLA with K product terms is ______.

52. State, in one sentence, the difference between combinational circuits and sequential circuits.

53. For the logic network shown in the figure below, the outputs X and Y are given by



a) $X = A \oplus B + (AB) \oplus C$ $Y = ABC$

b) $X = ABC$ $Y = A \oplus B + (AB) \oplus C$

c) $X = AB + (A \oplus B)C$ $Y = A \oplus B \oplus C$

d) $X = A \oplus B \oplus C$ $Y = AB + (A \oplus B)C$

54. Match the following by marking (1) or (2) against each.

1. Combinational circuits	a. Encoder, decoder	…( )
2. Sequential circuits	b. Multiplexer, demultiplexer.	…( )
	c. Full adder, comparator	…( )
	d. ROM, PROM, EPROM, EEPROM	…( )
	e. PLA, PAL	…( )
	f. Counter	…( )
	g. Sequence detector	…( )
	h. Add-shift multiplier	…( )

55. A MUX with its address bits generated by a counter operates as
    a) Parallel to serial converter b) Serial to parallel converter
    c) Modified counter d) Modified multiplexer

56. A DMUX with its address bits derived from a counter performs as
    a) Modified DMUX b) Modified counter
    c) Serial to parallel converter d) Parallel to serial converter

PROBLEMS

1. a) Using a 2 to 1 MUX, realise all possible 16 functions of two variables. Use B as the select input and assign 0, 1, A or $\overline{A}$ to the inputs.
   b) Design a 4-to-1 MUX using a 2-to-4 decoder and basic logic gates.
2. a) $F_1(x, y) = \Sigma m(0, 3)$
   $F_2(x, y) = \Sigma m(1, 2, 3)$
   Implement the combinational circuit by means of the 2-to-4 line decoder with enable input and external NAND gates.
   b) Draw the logic diagram of a 2-line to 4-line decoder-DEMUX using NOR gates.
3. Design a BCD-to-decimal decoder which gives an output of all 0s when any invalid input combination occurs.
4. a) Give the logic circuit schematic to realise a BCD to decimal decoder.
   b) Distinguish between
   i) Multiplexers – demultiplexers ii) Parity checker – comparators
   iii) Encoders – decoders
5. a) What are multi-output systems? Explain the operation of a decoder circuit.
   b) Design a combinational circuit that will accomplish the multiplication of the 2-bit binary number $X_1 X_0$ by the 2-bit binary number $Y_1 Y_0$. Is a two level circuit the most economical?
6. For the following two Boolean functions design a circuit with a decoder and external gates.
   $F_1 = x' y' z' + xz$
   $F_2 = xy' z' + x' z$
7. Realise 2K × 1 MUX using two modules of 1K × 1 MUX and additional logic.

8. Construct a 4 × 16 decoder using five 2 × 4 decoder modules. Show the schematic diagram neatly.
9. Construct a 4 × 16 decoder using two 3 × 8 decoder modules and additional logic. Show the schematic diagram neatly.
10. a) Give the schematic circuit of a 2-to-4 binary decoder with an active-low enable input. Show the Truth Table.
    b) Give the gate-level realisation for 8:1 MUX with active-low enable input. Show how several 8:1 MUXs can be combined to make a 32-to-1 MUX.
11. a) Show how a 16-to-1 MUX can be realised using a tree-type network of 4-to-1 MUXs.
    b) Implement the function f(a, b, c) = a.b + b.c using the 2-to-1 MUX.
12. Show that four modules of 2-to-4 decoder plus a gate switching matrix can be connected to form a 4-to-16 decoder.
13. Implement the following function using a multiplexer of proper size.

    $F(w, x, y, z) = \Sigma m(0, 1, 2, 3, 4, 9, 13, 14, 15)$
14. a) Realise $F(a, b, c, d) = \Sigma(0, 1, 5, 7, 6, 10, 14)$ using a multiplexer.
    b) Design a decimal to BCD converter. Draw its realisation.
15. a) Implement the following logic function with $2^{n-1} \times 1$ multiplexer, where n is the number of variables in the function.

       $F(A, B, C, D) = \Sigma(4, 5, 6, 7, 8, 13, 14, 15)$

    b) Implement the following using 4-to-16 line decoder.

       $F(A, B, C, D) = \Sigma(0, 1, 4, 7, 9, 12, 14)$
16. a) Design a full adder with two half adders and other logic gates if required.
    b) Convert Excess-3 code to BCD using 2-level circuits.
17. a) Briefly explain the operation of a multiplexer.
    b) Design a combinational logic circuit which converts a BCD number to a seven-segment display.
18. Realise a 3-bit comparator.
19. In a comparator of 2-bit numbers given by $A = A_1A_0$ and $B = B_1B_0$, what are the logical expressions for the following?

    L =

    E =

    G =

    L becomes 1 only if A is less than B. E becomes 1 if A = B. G becomes 1 if A > B.
20. Two single-bit numbers A_0 and B_0 are to be compared and the circuit should produce three outputs L, E, G, for $A_0 < B_0$, $A_0 = B_0$, and $A_0 > B_0$ respectively. Show the logic and give the Boolean expression for L, E, G.
21. a) Construct a full adder using half adders. Differentiate between a serial adder and a parallel adder.
    b) Define a multiplexer. Show how a multiplexer may be used as a sequential data selector.
22. Design a combinational circuit whose input is a 4-bit number and where output is the 2’s complement of the input number.

23. Use a 64 × 8 ROM to convert a 6-bit binary number to its corresponding 2-digit BCD representation.

    $(a_5\ a_4\ a_3\ a_2\ a_1\ a_0)_2 = ((x_3, x_2, x_1, x_0)_{BCD}\ (y_3, y_2, y_1, y_0)_{BCD})_{10}$

    Show the ROM contents in a truth table format.

24. Illustrate the carry-look-ahead phenomenon in a parallel adder giving an example.

25. Design a combinational circuit that accepts a 3-bit number and generates an output binary number equal to the square of the input number.

26. a) Draw the organisational schematic of a PLA and explain its operation. What are the advantages of PLAs?

    b) Give the diode matrix configuration for a ROM and explain the principle of data storage. How can PROMs and EEPROMs be realised?

27. Realise the functions given using PLA with 6 inputs, 4 outputs and 10 AND gates.

    $f_1(A, B, C, D, E, F) = \Sigma m(0, 1, 2, 7, 8, 9, 10, 11, 15, 19, 23, 26, 27, 31, 32, 33, 35, 39, 40, 41, 47, 63)$

    $f_2(A, B, C, D, E, F) = \Sigma m(8, 9, 10, 11, 12, 14, 21, 25, 27, 40, 41, 42, 43, 44, 46, 57, 59)$

28. Realise the given functions using PAL.

    $f_1(A, B, C, D, E, F) = ABD'F + CD'EF' + ACE'F + AB'CDF'$

    $f_2(G, H, K, M, N) = GH'KMN' + GHK'M'N$

29. Write the PLA program table for $F_1 = \Sigma(4, 5, 7)$, $F_2 = \Sigma(3, 5, 7)$.

30. a) It is required to design a multiplier for 2-bit numbers with the multiplicand A denoted by $A_1\ A_0$ and multiplier B denoted by $B_1\ B_0$ and the product denoted by $P_3\ P_2\ P_1\ P_0$ using a ROM. Write the ROM Truth Table.

    b) List the minterms for Ps.

    c) Using AND-OR-INVERT structure, what is the minimum number of links (fuses) to be burnt?

31. a) Show the general block schematic of a PLA.

    b) Given no. of inputs = 16

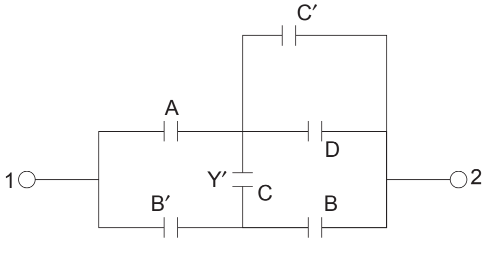
    No. of product terms = 40

    No. of output terms = 100

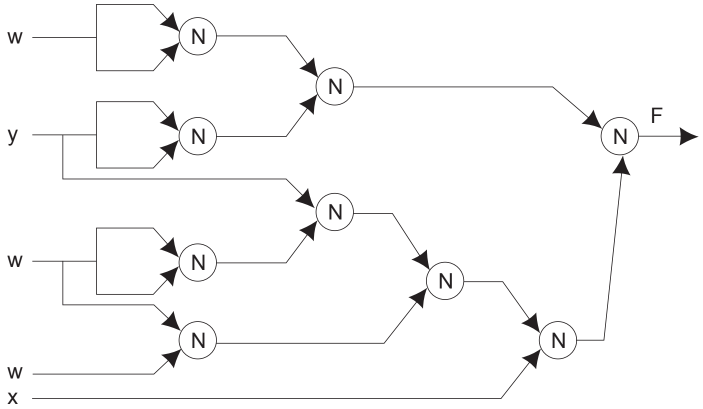
    What is the total number of fuses to be programmed (burnt/not burnt) in the PLA?

    c) If realised with ROM and AND-OR-NOT structure, how many fuses are to be programmed?

32. Find all cut-sets and tiesets of the following network. Express the transmission T_{12} in SP and PS forms and map the function. Draw the dual and complement of the network.



33. a) Map the function realised by the following NAND gate network.



b) If all NAND gates are replaced by NOR gates, find the function F and map it.

c) Indicate all static hazards of the network on a map of F.

d) Leave the original network as it is and add +, •3, and negation gates to make F hazard free.

34. Show how the following functions can be realised by an AND-OR array.

$F_1(a, b, c) = ab + c'$ $\qquad F_2(a, b, c) = ab + b'c$

35. Realise $F = x_1 s_2 x_3 + x'_2 x_4 + x'_1 x_2$ using

a) NAND gates b) NOR gates

36. Design a combinational network, which will realise the following pair of functions with +, · , and negation gates. Assume that the variable and its complement are available.

$$T1(A, B, C, D) = V(2, 8, 10, 11)$$

$$T2(A, B, C, D) = V(6, 12, 14, 15)$$

37. Construct a 5 × 32 decoder with four 3 × 8 decoders with enable and one 2 × 4 decoder.

38. Design a 4-input priority encoder with inputs as in the table given below, but with input D_0 having the highest priority and input D_3 the lowest priority.

Inputs				Outputs		
D_0	D_1	D_2	D_3	X	Y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

39. Implement a full adder with two 4 × 1 multiplexers.

40. Given a 32 × 8 ROM chip with an enable input, show the external connections necessary to construct a 128 × 8 ROM with four chips and a decoder.

41. A ROM chip of 4096 × 8 bits has two enable inputs and operates from a 5-volt power supply. How many pins are needed for the integrated circuit package? Draw a block diagram and label all input and output terminals in the ROM.

42. Obtain the PLA programming table for the BCD-to-Excess-3-code converter.
43. Work out the PAL programming table for the BCD-to-Excess-3-code converter.
44. The following is a Truth Table of a 3-input, 4-output combinational circuit. Obtain the PAL programming table for the circuit and mark the fuses to be blown in a PAL diagram.

Inputs			Outputs			
X	Y	Z	A	B	C	D
0	0	0	0	1	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	0	0	0	0	1	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1

5

Threshold Logic and Symmetric Functions

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Logic of threshold gate and its impact on researchers.
- Incredible power of the T-gate in realising different functions in contrast with conventional logic gates.
- Usefulness of the concept of threshold gate in other areas like neural networks.
- Limitations with regard to sensitivity to parameter variations and difficulty in achieving solid state versions in IC chip form.
- Symmetric functions and their identification.

5.1 INTRODUCTORY CONCEPTS

The earlier chapters covered the techniques of designing various logic circuits using AND, OR, NOT gates. We also discussed implementation using the universal operators/universal gates such as NAND, NOR gates. An entirely new concept is being presented in this chapter. The **threshold element,** also called **threshold gate (T-gate),** is a much more powerful device than any of the conventional logic gates such as NAND, NOR and others. The power of the threshold gate lies in realising complex, large Boolean functions using much fewer gates. Frequently, a single gate can realise a very complex function which otherwise might require a large number of conventional gates. In spite of the fact that the T-gate offers incomparably economical realisation, it has not found extensive use with the digital system designers; the main reasons lie in its limitations with regard to sensitivity to parameter variations and difficulty in fabricating it in integrated circuit form. However, it is a fascinating concept which has attracted the attention of innumerable researchers in the 1960s and 1970s. The underlying concepts are also useful in areas like neural networks. The basic principles of **threshold logic** are presented in this chapter.

5.2 THRESHOLD GATE—DEFINITION AND HARDWARE

A pictorial definition of a T-gate is given in Fig. 5.1. Whereas the inputs x_i and the output F are binary (two-valued 0 or 1), the w_is, called weights, and T, called threshold, can be any real number. The output function F assumes the value 1 if the weighted sum of the inputs equals or exceeds the value of T.

A threshold element is constructed in many ways; one method is shown in Fig. 5.2. When all the inputs are at zero level (positive logic), the base-emitter junction is reverse biased and the output will be at level 1. The base to emitter voltage depends on the weighted sum of the inputs determined by R_0 and the input resistors. When this weighted sum equals or exceeds a certain value, the transistor goes into saturation to make the output logical zero. Thus, this is a **complement threshold gate.** The threshold is determined by R_0. Since all resistances have positive values, this gate is capable of providing only positive weights. Using magnetic coils wound round a core, it is possible to realise negative weights too. The purpose of this chapter is to present the fundamental concepts about the T-gate and its capabilities in realising logic functions.

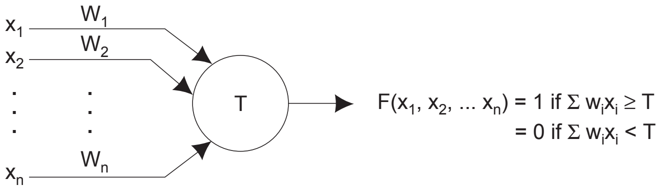

x_1
x_2
x_n
W_1
W_2
W_n
T
$F(x_1, x_2, \ldots x_n) = 1$ if $\Sigma w_i x_i \geq T$
$= 0$ if $\Sigma w_i x_i < T$

Fig. 5.1 Symbol and definition of threshold gate

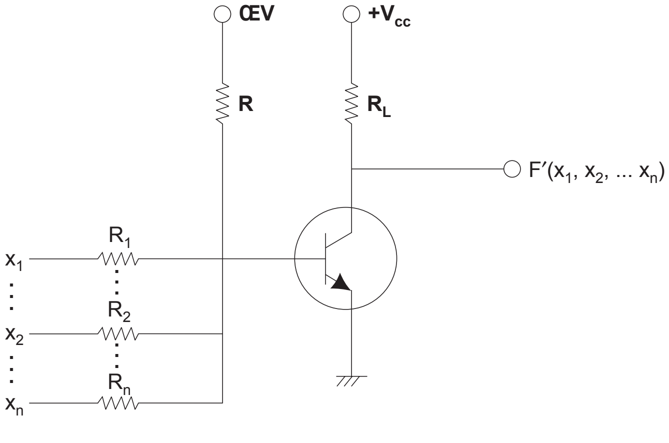
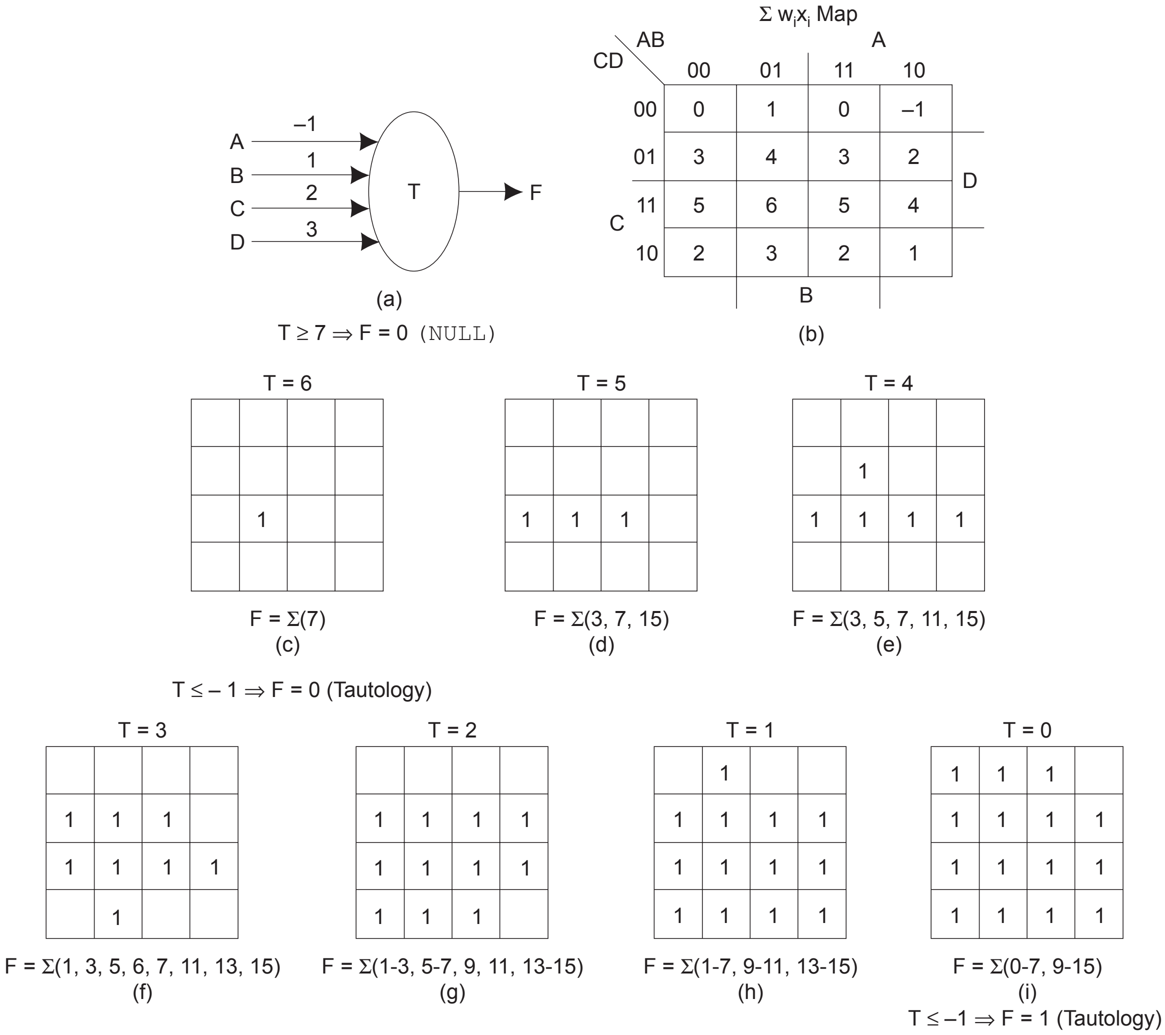


Fig. 5.2 Typical hardware required to construct a complement T-gate

5.3 CAPABILITIES OF T-GATE

The capability of a single threshold gate in realising various functions is demonstrated by means of a numerical example in Fig. 5.3. Note that Fig. 5.3(a) shows a T-gate with four inputs A, B, C, D and F is a function of these input variables. The weights assigned are –1, 1, 2 and 3, as shown. By altering the value of T, the same gate produces different functions. Fig. 5.3(b) shows the weighted sum figures in each cell. For cell 6 in the map corresponding to A = 0, B = 1, C = 1, D = 0. The weighted sum $\Sigma w_i x_i$ is given by $(-1) \times 0 + 1 \times 1 + 2 \times 1 + 3 \times 0 = 3$. While writing the weighted sum in each cell, it is convenient and quicker if one remembers the adjacency relationships in the map. For instance, the row labelled 01 is adjacent to the row labelled 00 and note that the only variable which changes is D which goes from 0 to 1. Hence by adding the weight of D to the corresponding entry in the earlier row, one gets the values in the row of 01. Similarly, one gets the entries in the



(a)

T ≥ 7 ⇒ F = 0 (NULL)

$\Sigma w_i x_i$ Map

CD \ AB	00	01	11	10
00	0	1	0	–1
01	3	4	3	2
11	5	6	5	4
10	2	3	2	1

(b)

T = 6

	1		

F = Σ(7)
(c)

T = 5

1	1	1	

F = Σ(3, 7, 15)
(d)

T = 4

	1		
1	1	1	1

F = Σ(3, 5, 7, 11, 15)
(e)

T ≤ – 1 ⇒ F = 0 (Tautology)

T = 3

1	1	1	
1	1	1	1
	1		

F = Σ(1, 3, 5, 6, 7, 11, 13, 15)
(f)

T = 2

1	1	1	1
1	1	1	1
1	1	1	

F = Σ(1-3, 5-7, 9, 11, 13-15)
(g)

T = 1

	1		
1	1	1	1
1	1	1	1
1	1	1	1

F = Σ(1-7, 9-11, 13-15)
(h)

T = 0

1	1	1	
1	1	1	1
1	1	1	1
1	1	1	1

F = Σ(0-7, 9-15)
(i)

T ≤ –1 ⇒ F = 1 (Tautology)

Fig. 5.3 Illustration of the power of T-gate

row of 11 by simply adding the weight of C. Finally, the row labelled 10 is filled by simply subtracting the weight of D from the corresponding entry of row 11 since D is changing from 1 to 0.

Notice that this gate produces the NULL function F = 0 if T ≥ 7 because the weighted sum is less than 7 for every combination of input variables. As the threshold T is gradually decreased to 6, 5, 4, 3, 2, 1 and 0, the function realised will contain the minterms as indicated. Eventually if T is further decreased to (– 1) or less, the function will be 1 (called Tautology), regardless of the input variables.

By merely changing the values of resistance, we can change the value of the threshold and realise different Boolean functions. The weights can also be changed in a similar fashion. We, therefore, conclude that we may realize a large number of functions using a single T-gate. This capability has to be attributed to the hybrid nature of the T-gate having analog weights and threshold but digital inputs and output.

Universality of T-Gate

We have learnt that a single T-gate can realise a large number of functions by merely changing either the weights of the input variables or the threshold or both, which can be done by altering the value of the corresponding resistors.

Several questions arise. Is the threshold gate a universal gate? Can a single threshold gate with n inputs realise all the 2^{2^n} possible functions of n variables? What are the conditions that a given function must satisfy so that it can be realised by a single threshold gate? If the given function cannot be realised by a single T-gate, how do we decompose the functions into several smaller functions so that each one can be realised by a single threshold gate, and how do we interconnect all such realizations?

The answer to the first question is yes. The universality of the threshold gate is easily established by realising negation (inversion or complement) and one of the operators OR or AND. Realisation implies giving a set of weights and fixing the threshold. This is shown in Fig. 5.4(a) and (b). The reader should prepare Truth Tables in each case and get a feel of the concept. Figures 5.5(a), (b), (c), (d), and (e) indicate realisation of other frequently encountered logic functions. The reader is advised to verify with the Truth Table of each function.

The answer to the second question is "no". If a single threshold gate could realise all possible functions, then that would be the ultimate and no further development would be possible. Such situations do not arise in nature. EXCLUSIVE OR function, for example, cannot be realised by a single threshold gate. This statement can be proved by a simple counter example. Suppose a T-gate with weights w_1 for x_1 and w_2 for x_2 realises the XOR function. Remember that F becomes 1 whenever $\Sigma w_i x_i = w_1 x_1 + w_2 x_2 \geq T$.

Remember also that $F = x_1 \oplus x_2 = x_1' x_2 + x_1 x_2'$

Set $x_1 = 0$, $x_2 = 0$, which implies F = 0

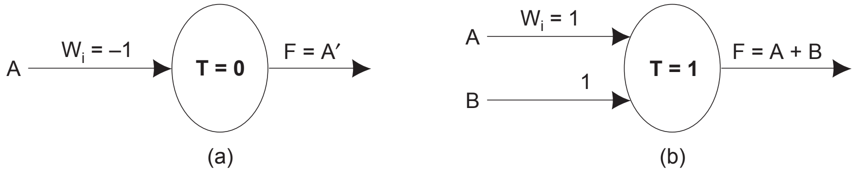
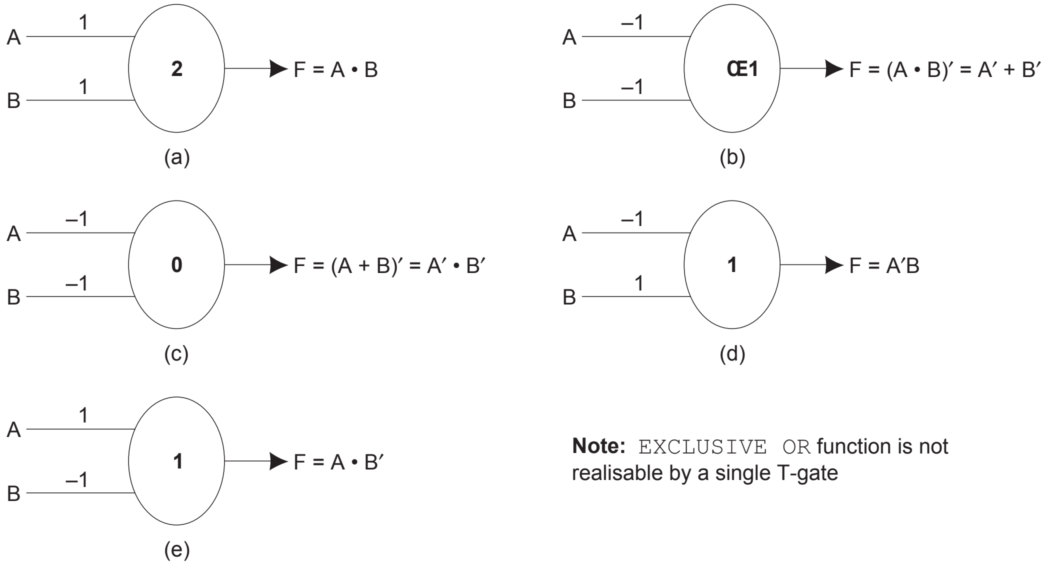
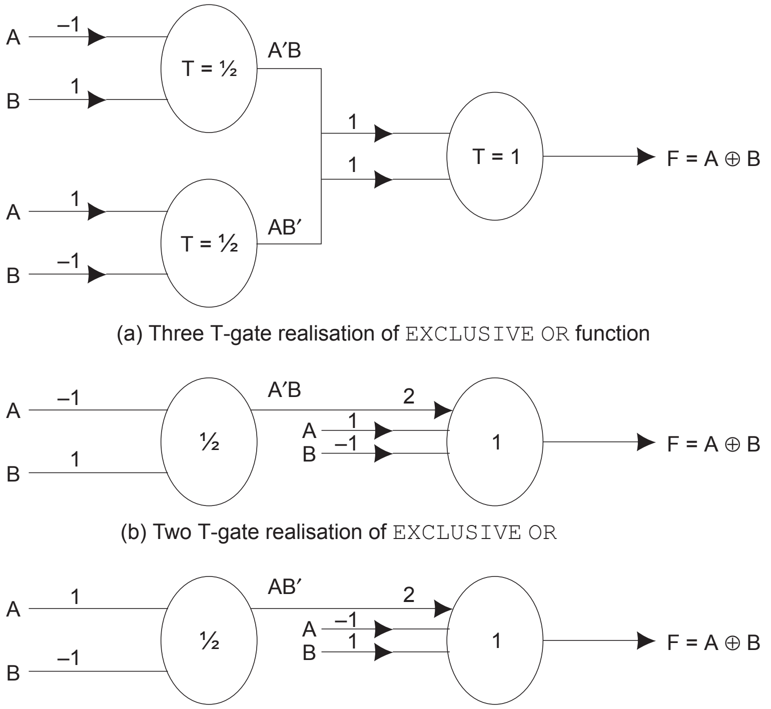


Fig. 5.4 Demonstrating the universality of T-gate



Note: EXCLUSIVE OR function is not realisable by a single T-gate

Fig. 5.5 Realisation of common 2-variable Boolean logic functions using a single T-gate



(a) Three T-gate realisation of EXCLUSIVE OR function

(b) Two T-gate realisation of EXCLUSIVE OR

(c) Two T-gate realisation of EXCLUSIVE OR

Fig. 5.6 Realisation of EXCLUSIVE OR function using threshold gates

If x_1 changes from 0 to 1 while $x_2 = 0$, then F should change from 0 to 1, which implies that w_1 should be positive. Consider another situation where $x_1 x_2 = 11 \Rightarrow F = 0$. Let x_1 change from 1 to 0; then F should change from 0 to 1. As x_1 decreases, F increases, which implies that w_1 is negative. Since the weight of a variable cannot be simultaneously positive and negative, we conclude that it is not possible to assign weights and therefore a single T-gate realisation for XOR does not exist. Multiple T-gate realisations of XOR function are given in Fig. 5.6(a), (b), and (c). Notice that in all the realisations shown in Fig. 5.5, the variables associated with positive weights appear in uncomplemented form in the function and the variables with negative weights appear in complemented form in the output function. This aspect is further discussed in a subsequent section.

The third question is answered in the subsequent paragraphs of this chapter. It is shown that the functions should obey the property of **linear separability** in order that a single T-gate will be able to realise the given function. Such functions are called **“Threshold Functions”** or **“Linearly Separable (LS) functions”.** It is also shown that **“Unateness”** is a necessary but not sufficient condition for a given function to be realised by a single T-gate.

The fourth question poses a challenging problem. The decomposition of a given function into smaller threshold functions is a topic beyond the scope of this book.

5.4 PROPERTIES OF THRESHOLD FUNCTIONS

1. Linear Separability Recall that the output of a T-gate, also called threshold function is dependent on whether the weighted sum is less than a certain constant or not. Consider an n-variable threshold function, which can be represented by an n-cube. The true vertices are separated from the false vertices by a linear equality.

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = T$$

represents an (n – 1) dimensional hyperplane which cuts across the n-cube. Owing to this property of linear separability of the true and false vertices, threshold functions are called **linearly separable or linear input functions.** Obviously, any LS function can be realised by a single T-gate.

2. Unateness If the minimal sum of products (MSP) form of a switching function contains each variable in only one form, either complemented or uncomplemented, then the function is called a unate function. If all the variables appear in uncomplemented (complemented) form only, then the function is called a positive unate (negative unate) or simply positive (negative) function. If a variable x_i appears in only complemented (uncomplemented) form, then the function is said to be negative (positive) unate in that variable x_i.

Example 5.1

$F_1 = AB' + BC'$ is not unate.

$F_2 = AB' + CD$ is unate. It is positive unate in A, C, D and negative unate in B.

$F_3 = AB + CD$ is a positive function.

$F_4 = A'B' + C'D'$ is a negative function.

Every LS function is unate. (The converse is not true). This can be proved by a simple argument. Consider a function $F(x_1, x_2, \ldots x_n)$ which is LS and hence is realisable by a set of weights $w_1, w_2, \ldots w_n$ and threshold T. Without loss of generality, consider a variable x_1 and its weight w_1.

Two cases arise.

Case (i) w_1 is positive. Let the function F have a true minterm containing x_1', say $x_1' x_2 \ldots x_n$. This would mean that F = 1 for $x_1 = 0$, $x_2 = 1$, $x_3 = 1 \ldots x_n = 1$ which, in turn, means

$$\sum_{i=2}^{n} w_i x_i \geq T \quad \text{as } w_1 x_1 \text{ is } 0.$$

Now, consider the minterm obtained by replacing x_1' by x_1 namely, the minterm $x_1 x_2 \ldots x_n$. This must also be a true minterm since w_1 is positive and

$$\sum_{i=1}^{n} w_i x_i \geq T$$

Thus, the two minterms combine to make F independent of x_1. We, therefore, conclude that the variable x_1 cannot appear in a complemented form in the MSP form of F. It must appear only in uncomplemented form if its weight is positive.

Case (ii) w_1 is negative. A similar argument shows that the variable x_1 must appear only in a complemented form in the MSP form of F.

Note that **unateness** is a necessary but not a sufficient condition for linear separability.

3. Isobaric Functions Two threshold functions are said to be isobaric if they can be realised by the same set of weights but with different threshold values. The functions mapped in Fig. 5.3 are all isobaric. Two isobaric functions are comparable in the sense that one includes the other.

5.5 SEPARATING FUNCTION REPRESENTATION

The function realised by the T-gate of Fig. 5.3 for T = 4 may also be written as

$$F = \langle -x_1 + x_2 + 2x_3 + 3x_3 \rangle 4$$

The expression on the RHS is called a separating function. The general form is

$$F = \left\langle \sum w_i x_i \right\rangle T$$

There is yet another form used in literature which explicitly shows the minimum value (U) of the weighted sum for F = l and the maximum value (L) of the weighted sum for F = 0. For example, in Fig. 5.3, for T = 3, looking at the map of the weighted sum, we find that U = 3 and L = 2. The difference u-l is called the gap ‘g’. This leads to the representation of the function as

$$F = (-x_1 + x_2 + 2x_3 + 3x_3) \quad 3:2$$

Or, in general

$$F = \left(\sum w_i x_i\right) U : L$$

The threshold T has to be chosen such that it is greater than L and does not exceed U and is generally chosen to be midway between U and L in order to allow the greatest tolerance.

5.6 COMPLEMENT AND DUAL FUNCTIONS

The following statements can be proved easily by starting from the basic definition of the threshold gate and are left as an exercise to the reader.

$$\text{If } F(x_1, x_2 \ldots x_i, \ldots x_n) = \left\langle \sum_1^n w_i x_i \right\rangle U : L$$

Then $F(x_1, x_2 \ldots x_i' \ldots x_n)$ is given by

$$\langle w_1 x_1 + w_2 x_2 + \ldots w_i x_i + \ldots w_n x_n \rangle (U - w_i) : (L - w_i)$$

$$F'(x_1, x_2 \ldots x_n) = \left\langle -\sum w_i x_i \right\rangle - L : - U$$

$$F_d(x_1, x_2 \ldots x_n) = \left\langle \sum w_i x_i \right\rangle \left(\sum w_i - L\right) : \left(\sum w_i - U\right)$$

5.7 SYNTHESIS OF THRESHOLD FUNCTIONS

In this section, synthesis of threshold functions is illustrated with an example.

Example 5.2 Realise the given function using a single T-gate.

$$F = CD + A'D + B'D + A'B'C$$

Step I First we check for unateness. The given function is unate, negative unate in A and B and positive unate in C and D.

Step II Transform the function by changing the negative variables A and B and write as

$$R = CD + AD + BD + ABC$$

If we can find a realisation for R, then by simple manipulation, we can find the realisation for F.

Step III Find the true and false prime implicants of the function R. (Note that it can be proved that a unate function does not have any redundant prime implicants.) The true prime implicants of R are given by (see Fig. 5.7)

$$R = CD + AD + BD + ABC$$

These are marked in Fig. 5.7.

The false prime implicants are given by

$$R = (A + D)\ (B + D)\ (C + D)\ (A + B + C)$$

Step IV Form inequalities as follows.

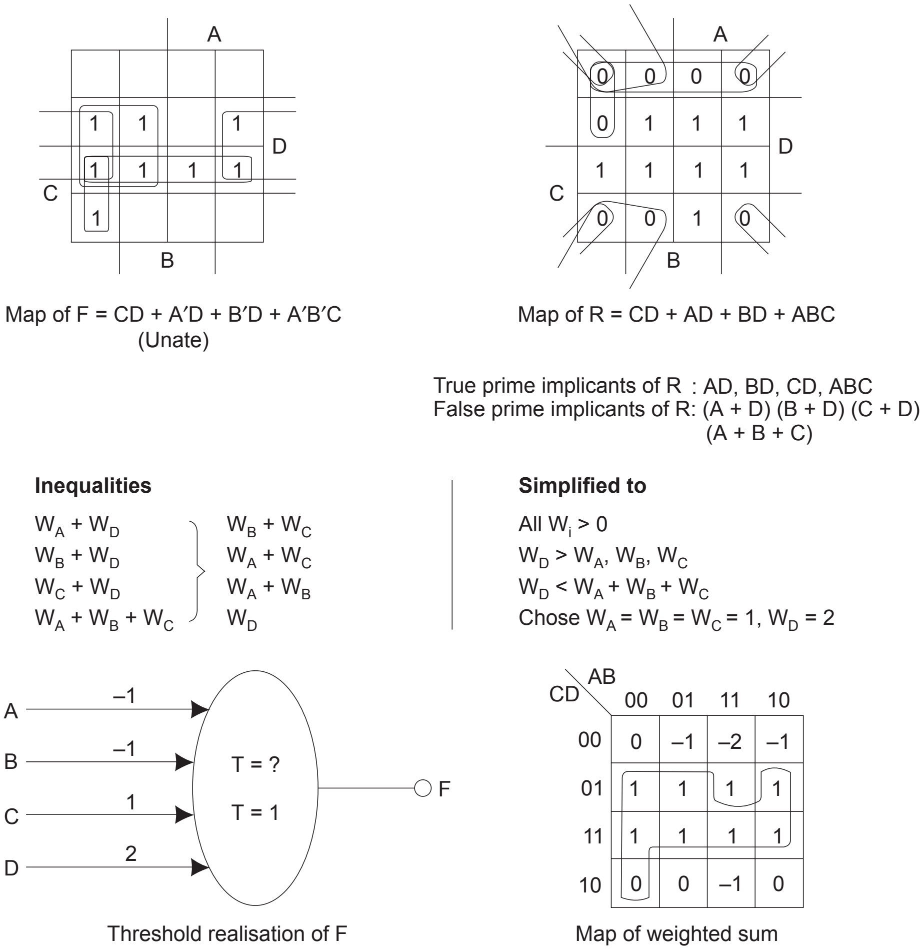
The minimal weighted sum corresponding to each true prime implicant must be greater than the maximal weighted sum of any false prime implicant.

The true prime implicant CD says that the function must be 1 when C = D = 1 regardless of the value of other variables. This would mean $W_C + W_D \geq T$.

The false prime implicant A + D, for instance, says that the function must be 0 when A = D = 0, irrespective of the values of other variables. The most adverse case occurs when both B and C assume the value 1, in which case the weighted sum corresponding to the false prime implicant (A + D) will be maximum. Even this value should be less than T, that is, $W_B + W_C < T$.

Following the above arguments, the inequalities are written as follows.

$$\left.\begin{array}{l} W_C + W_D \\ W_A + W_D \\ W_B + W_D \\ W_A + W_B + W_C \end{array}\right\} \geq T > \left\{\begin{array}{l} W_B + W_C \\ W_A + W_C \\ W_A + W_B \\ W_D \end{array}\right.$$



1	1		1
1	1	1	1
1			

Map of F = CD + A′D + B′D + A′B′C
(Unate)

0	0	0	0
0	1	1	1
1	1	1	1
0	0	1	0

Map of R = CD + AD + BD + ABC

True prime implicants of R : AD, BD, CD, ABC
False prime implicants of R: (A + D) (B + D) (C + D)
(A + B + C)

Inequalities

$$\left.\begin{array}{l} W_A + W_D \\ W_B + W_D \\ W_C + W_D \\ W_A + W_B + W_C \end{array}\right\} \begin{array}{l} W_B + W_C \\ W_A + W_C \\ W_A + W_B \\ W_D \end{array}$$

Simplified to

All $W_i > 0$
$W_D > W_A, W_B, W_C$
$W_D < W_A + W_B + W_C$
Chose $W_A = W_B = W_C = 1$, $W_D = 2$

Threshold realisation of F

CD \ AB	00	01	11	10
00	0	–1	–2	–1
01	1	1	1	1
11	1	1	1	1
10	0	0	–1	0

Map of weighted sum

Fig. 5.7 Illustrating single threshold gate synthesis

Simplifying the above inequalities we get

$W_A > 0$, $W_B > 0$, $W_C > 0$, $W_D > W_A$, or W_B, or $W_{C,}$ and $W_D < W_A + W_B + W_C$. If we now choose $W_A = W_B = W_C = l$ and $W_D = 2$, all the inequalities are satisfied and the given function is linearly separable and hence is realisable by a single T-gate. If, however, we encounter any inconsistency in the inequalities, then we simply declare the function as not linearly separable.

Step V Now, we go back to the original function F and as it is negative unate in A and B, we negate the weights of the variables A and B. Then we obtain a map of the weighted sum and by comparison with the map of the function F, fix the threshold T. The expression given in Section 5.6 relating to complementing literals can be used. The entire procedure is shown in Fig. 5.7.

Various other methods for the synthesis of threshold functions exist in the literature.

5.8 MULTI-GATE SYNTHESIS

Given an arbitrary Boolean function, there are as yet no general methods to realise the function with a minimal number of T-gates. This deficiency is not peculiar to T-gate only but is applicable to conventional gates such as NAND and, NOR also. It is interesting to note that symmetric functions, presented next, lend themselves to simple, interesting realisations with T-gates. Some of the problems given at the end cover these aspects.

5.9 LIMITATIONS OF THRESHOLD GATE

One major limitation of the T-gate is its sensitivity to parameter variations and supply voltage variations. The weighted sum may deviate from the designed value, especially with a large number of inputs. This limits the **fan-in** (number of inputs permissible) capacity. Another problem is the lack of simple, efficient procedures for identification and synthesis of threshold-gate networks. In spite of the large amount of research that has gone into this area, satisfactory procedures for synthesis of an arbitrary Boolean function using only T-gates are yet to be found.

5.10 SYMMETRIC FUNCTIONS AND NETWORKS

Symmetric functions fascinated logicians and design engineers who used to revel in learning all their aspects till the 1970s. Symmetric functions yield directly bridge and non-planar networks which are much more economical than the best series-parallel circuits obtainable. A knowledge of symmetric functions is useful in tackling problems like “Draw a circuit which will be closed if and only if exactly four out of a total of seven relays are operated”. At that time, most telephone and telegraph networks used contacts and the associated electro-mechanical relays which were replaced gradually in time by electronic gates. Thus, the interest shifted. Nevertheless, the subject is still absorbing and holds a great deal of academic interest. In this section some fundamental concepts are presented.

Symmetric functions lend themselves to interestingly simple realisations using T-gates. A switching function $f(x_1, x_2, \dots x_n)$ is called symmetric (or totally symmetric) if it is invariant under any permutation of its variables. It is called partially symmetric in the variables x_i, x_j if the interchange of the variables x_i, x_j leaves the function unchanged. In this section we shall concern ourselves only with totally symmetric functions.

Example 5.3

$F(x, y, z) = x'y'z + xy'z' + x'yz'$

is symmetric with respect to the variables x, y, z. The variables of symmetry may be complemented or uncomplemented.

Example 5.4

$F (a, b, c) = a'b'c' + ab'c + a'bc$ is not symmetric with respect to the variables a, b, c but it is symmetric with respect to the variables a, b, c′.

A symmetric function is denoted by

$$S^n_{a_1, a_2, \ldots a_k} (x_1, x_2 \ldots x_n)$$

where S designates the property of symmetry, the subscripts a_1, a_2, … a_k are called alpha numbers, the superscript n denotes the number of variables, and $(x_1, x_2, \ldots, x_n)$ designates the variables of symmetry. The function assumes the value 1 when exactly a_1 or a_2 or … or a_n variables assume the value 1.

The following Boolean identities may be proved easily.

(1) $S^n_a + S^n_b = S^n_{a, b}$

(2) $S^n_{a, b} \cdot S^n_{b, c} = S^n_b$

(3) $(S^n_{a, b})' = S_{i1, i2} \ldots, i \neq a, b.$

(4) $S^n_{a, b} \ (x_1, x_2, \ldots x_n) = S_{\ n-a,\ n-b} (x_1', x_2', \ldots x_n')$

Example 5.5

Satisfy yourself that all the four symmetric functions given below are only different forms of the same Boolean function.

$$S^5_{1, 3, 4, 5} (a, b, c, d, e), (S^5_{0, 2} (a, b, c, d, e))'$$

$$S^5_{0, 1, 2, 4} (a', b', c', d', e'), (S^5_{3, 5} (a', b', c', d', e'))'$$

By extension of the notation

$$S^{n\ 3} j = S^n j, j + 1, j + 2, \ldots n$$

Applying the Shannon's Expansion Theorem, we may write

$$S_a (x_1, x_2 \ldots x_n) = x_1' S_a (0, x_2 \ldots x_n) + x_1 S_a (1, x_2 \ldots x_n)$$

$$= x_1' \bullet S_a (x_2 \ldots x_n) + x_1 S_{a-1} (x_2 \ldots x_n)$$

Or, in general, we get

$$S_{a1,\ a2\ \ldots\ ak} (x_1, x_2, x_n) = x_1' S_{a1,\ a2\ \ldots\ ak} (x_2 \ldots x_n) + x_1 \ S\ a_1 - 1, a_2 - 1, \ldots a_k - 1 \ (x_2 \ldots x_n)$$

where alpha number n (if it exists) is eliminated from the first term since it is a function of n – 1 variables. Similarly, the alpha number 0 (if it exists) is eliminated from the second term.

5.11 CIRCUIT REALISATION OF SYMMETRIC FUNCTIONS

Contacts are of different types: (1) normally open, (2) normally closed, (3) transfer contact, (4) make before break, and so on. When the associated relay coil is energised, the contacts rigidly attached to the yoke of the coil, change state.

A symmetric structure, which realises all symmetric functions of five variables, is shown in Fig. 5.8. First draw a grid of n + 1 horizontal lines and n + 1 vertical lines. Insert the normally closed contacts (complemented literals) in the horizontal segments and the normally open contacts (uncomplemented literals) in the diagonal segments as shown in the figure. Each output line gives a symmetric function with exactly one alpha number. If you want to realise, for instance, $S_{2,4,5}$, you need only short-circuit the output leads corresponding to S_2, S_4 and S_5.

The circuits can be simplified by identifying the paths from input to output which correspond to x + x′ or x + x′y. For example, the top E and E′ contacts may be shorted and thus eliminated in the realisation of $S_{2,4,5}$. For further simplification, the reader is advised to further probe into the grid structure of Fig. 5.8.

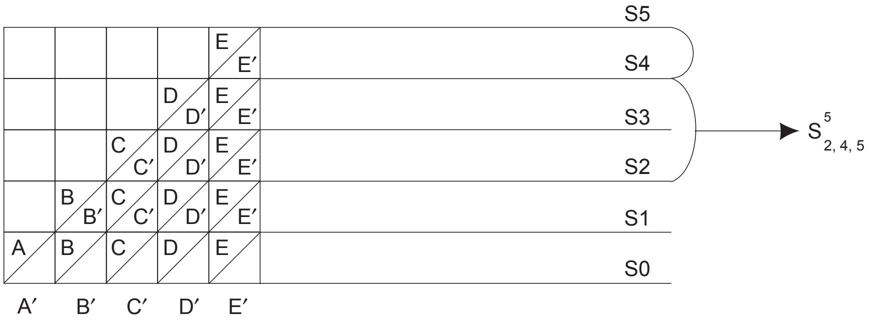


Fig. 5.8 Circuit realisation of symmetric function $S^5_{2,4,5}$

5.12 IDENTIFICATION OF SYMMETRIC FUNCTIONS

An algorithmic procedure for identifying symmetric functions and the variables of symmetry is illustrated below by means of an example.

Example 5.6 Find whether the function

$$F(w, x, y, z) = \Sigma(0, 1, 3, 5, 8, 10, 11, 12, 13, 15)$$

is symmetric and if so express the function in symmetric notation.

The true minterms of the function are listed in binary code in Table 5.1(a) and the column sums are noted below. A little reflection reveals that for any totally symmetric function, all column sums must be equal since the permutation of variables of symmetry does not change the function. In Table 5.1(a), there are two different column sums 6 and 4. Suppose now we complement the columns x and y, that is, we consider w, x′, y′ and z as the variables of the function, then the list of minterms will be as in Table 5.1(b). This process does not alter the function since x assuming the value 0 is the same as x′ assuming the value 1 and vice versa. The process is sometimes referred to as **double complementation** as both the variable and the entries in the corresponding column are complemented. In Table 5.1(b), we find that all column sums are identical. If there were to be more than two different column sums or if the sum of the two column sums were to be not equal to the number of rows, complementing any number of columns would not result in equal column sums. Thus, we obtain a necessary condition. **For a function to be symmetric, either all column sums must be equal or else there should be only two different column sums.**

Now, focus attention on the row sums indicated in the r_i column of Table 5.1 column (b). **If a row sum r occurs, it should occur $\binom{n}{r}$ times in order that the function be symmetric.** This follows from the fact that any r variable may assume the value 1 to make the function equal to 1. This leads to another necessary condition.

The two necessary conditions namely, equal column sums and occurrence of each row sum $\binom{n}{r}$ times, together constitute sufficiency to declare the function as a symmetric function. The row sums become the alpha numbers.

In Table 5.1(b), observe that the row sum 2 occurs $\binom{4}{2} = \frac{4 \times 3}{2 \times 1} = 6$ times and the row sum 3 occurs $\binom{4}{3} = 4$ times.

Hence the function is symmetric and can be written as

$$F(w, x, y, z) = S^{4}_{2,\,3}(w, x', y', z).$$

Table 5.1 *Test for Symmetry of the Function F = Σ (0,1, 3, 5, 8, 10, 11, 12, 13, 15)*

w	x	y	z	w	x′	y′	z	r_i	w′	x	y	Z′	r_i
0	0	0	0	0	1	1	0	2	1	0	0	1	2
0	0	0	1	0	1	1	1	3	1	0	0	0	1
0	0	1	1	0	1	0	1	2	1	0	1	0	2
0	1	0	1	0	0	1	1	2	1	1	0	0	2
1	0	0	0	1	1	1	0	3	0	0	0	1	1
1	0	1	0	1	1	0	0	2	0	0	1	1	2
1	0	1	1	1	1	0	1	3	0	0	1	0	1
1	1	0	0	1	0	1	0	2	0	1	0	1	2
1	1	0	1	1	0	1	1	3	0	1	0	0	2
1	1	1	1	1	0	0	1	2	0	1	1	0	2
6	4	4	6	6	6	6	6		4	4	4	4	
	(a)				(b)					(c)			

In Table 5.1(a), instead of complementing x, y columns, if we choose to complement w, z columns, we obtain Table 5.1(c), which gives another symmetric form of the same function as

$$F = (w, x, y, z) = S^{4}_{1,\,2}\,(w', x, y, z')$$

A little more difficult situation is encountered in testing the following function.

Example 5.7 Test for symmetry

$$T\,(A, B, C, D) = \Sigma(0, 3, 5, 10, 12, 15)$$

The true minterms are listed in Table 5.2(a). We find that all the column sums are identical satisfying the first necessary condition. But the second condition is not satisfied. The row sum 2 does not occur $\begin{pmatrix}4\\2\end{pmatrix}$ = 6 times as required for the function to be symmetric. The column sums do not give us any clue about which columns could be complemented to satisfy the sufficiency of row sum occurrence.

In this situation, the column sum should be equal to one half the number of rows; otherwise any complementation of column(s) upsets the equality of column sums. This condition being satisfied in Table 5.2(a), we now proceed to expand the function about any one of its variables say A. This can be achieved in the tabular form as in Table 5.2(b). The partial functions of B, C, D are easily recognised as symmetric in B′, C′, D. This would mean

$$F\,(A, B, C, D) = A' \bullet S_2\,(B', C', D\,) + A\,S_1\,(B', C', D)$$

which may be written as S_2(A, B′, C′, D) using the expansion theorem. Alternatively we, once again, write out the tabular form of the entire function with B and C columns complemented, as in Table 5.2(c). This table satisfies the sufficiency of occurrence of row sums. Hence the function will be declared as symmetric and written as

Table 5.2 *Test for Symmetry Involving an Expansion of the Function F = Σ(0, 3, 5, 10, 12, 15)*

A	B	C	D	r_i
0	0	0	0	0
0	0	1	1	2
0	1	0	1	2
1	0	1	0	2
1	1	0	0	2
1	1	1	1	4
3	3	3	3	

(a)

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
	1	1	2
1	0	1	0
1	1	0	0
1	1	1	1
	2	2	1

(b)

A	B′	C′	D	r_i
0	1	1	0	2
0	1	0	1	2
0	0	1	1	2
1	1	0	0	2
1	0	1	0	2
1	0	0	1	2
3	3	3	3	

(c)

$$F(A, B, C, D) = S^{4}_{2}(A \,.\, B', C', D)$$

$$= S^{4}_{2}(A', BC, D')$$

The entire procedure is summarised below

I Obtain column sums

(a) If more than two different sums occur, the function is not symmetric.

(b) If two different sums occur, compare the total of these two sums with the number of rows in the table. If they are not equal, the function is not symmetric. If they are equal, complement the columns corresponding to either one of the column sums and continue to step II.

(c) If all column sums are identical, compare the sum with half the number of rows in the table. If they are not equal, go to step II. If they are equal, go to step III.

II Obtain row sums and check for sufficient occurrence; that is, if r is a row sum and n is the number of variables, then that row sum must occur $\binom{n}{r}$ times.

(a) If any row sum does not occur the required number of times, the function is not symmetric.

(b) If all row sums occur the required number of times, the function is symmetric in the variables given at the top of the current table and the alpha numbers are given by the different row sums.

III Obtain row sums and check each for sufficient occurrence.

(a) If all row sums occur the required number of times, the function is symmetric.

(b) If any row sum does not occur the required number of times, expand the function about any one of its variables; that is, find functions g and h such that $f = x'g + xh$.

Determine all variable complementations required for the identification of symmetries in g and h. Test f under the same variable complementations. If all row sums occur the required number of times, f is symmetric; otherwise, it is not.

SUMMARY

The capabilities and limitations of a threshold gate are discussed. Realisation of a Boolean function using a single threshold gate is illustrated with an example. Necessary and sufficient conditions for a logic function to be symmetric are illustrated. How column sums and row sums play a crucial role in deciding symmetry has been demonstrated.

KEY WORDS

- ❖ Threshold logic
- ❖ Threshold gate
- ❖ Unateness
- ❖ Linear separatability
- ❖ True and false bodies
- ❖ Isobaric functions
- ❖ Totally symmetric function
- ❖ Different forms
- ❖ Partially symmetric function
- ❖ Identification of symmetric functions
- ❖ Alpha numbers

REVIEW QUESTIONS

1. What are the digital parameters and analog parameters of a T-gate?
2. Is T-gate a uiversal gate?
3. State the minimal integer weights and threshold for a 2-input T-gate to realise the following basic operators.

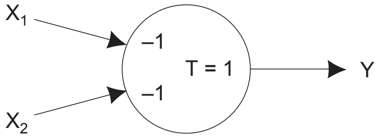
   a) Inverter b) OR c) AND
4. Assign the weights and threshold for a single T-gate to realise the following operators.

   a) 2-input NAND b) 2-input NOR
5. What do you understand by isobaric functions?
6. What is the equality which separates the true body from the false body of a n-variable threshold function?
7. Two isobaric functions must obey the property that one is included in the other. (True/False)
8. What is the condition for a function to be realised by a single T-gate?
9. All LS functions are unate. Is the converse true?
10. Define unateness.
11. In a given threshold function, the variables appearing in complemented form must have negative weights. Why?
12. Express the given symmetric function in three other forms.

    $S_{2,\,3}$ (a, b, c, d, e)
13. Should the dual of a symmetric function be also symmetric?
14. Does a symmetric function, on decomposition about one of its variables, yield symmetric functions as components?
15. State the conditions that every symmetric function should obey.
16. What is meant by double complementation used in testing symmetric functions?
17. What are the advantages in identifying symmetric functions?
18. How do we arrive at the variables of symmetry?
19. Distinguish totally symmetric functions from partially symmetric functions.

PROBLEMS

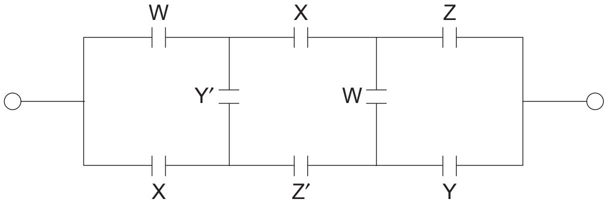
1. For the threshold gate given below, find the output function Y.



2. Prove the functional completeness (universality) of threshold gate by, realizing NOT, AND operations each using single threshold gate.
3. Find the functions F (A, B, C, D) realised by the threshold gate with weights 1, –1, 2, 3 and threshold 3. Realise f (A, B, C′, D), F′(A, B, C, D), and F_d (A, B, C, D).
4. Determine whether the following functions are linearly separable. If so, find a minimal integer weight realisation. If not, explain why.

   a) $F_1 = A'B + CD$,   b) $F_2 = D + AB' + AC$
5. Realise the symmetric function $S_{0,1}(x_1, x_2', x_3)$ with a single threshold element.
6. a) Show that $S_{1,2}(x_1, x_2, x_3)$ is not a linearly separable function.

   b) Find a two-T-gate realisation of $S_{1,2}(x_1, x_2, x_3)$.
7. Prove that unate function has no RPIs.
8. For the network given below, find an equivalent series parallel network with the fewest number of contacts.



9. a) Let F be a function for which the product of all true prime implicants is 0. Prove that F cannot be linearly separable.

   b) Let G be a linearly separable function. Prove that $H = x_0 G + x_0' G_d$ is also linearly separable.

   Derive a weak upper bound on the number of threshold gates required to realise a function of n variables.
10. Design a threshold gate circuit with 2n binary inputs ($x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n$) such that F ($x_1, x_2, \ldots x_n, y_1, y_2, \ldots y_n$) = 1 if $x_i = y_i$ for all i. (Three T-gates are sufficient and two gate circuit exits.)
11. Using the 3-variable tree, design a 2-output contact network, which realises the functions.

$$T_1(x, y, z) = \Sigma\, 0, 1, 2, 3, 5$$

$$T_2(x, y, z) = \Sigma\, 0, 1, 4, 6, 7$$

12. With the help of the expansion theorem, express the following as symmetric functions.

    a) A', $S_{0,1,4}$ (B, C, D, E) + A $S'_{0,3,4}$ (B, C, D, E)

    b) A', $S_{0,1,4}$ (B, C, D, E) + A $S_{0,3,4}$ (B, C, D, E)

    c) A', $S_{0,1,4}$ (B, C, D, E) + A $S_{0,3,4}$ (B′, C′, D′, E′)

13. Design a switching circuit which can turn a lamp on or off from three different locations independently. (Four transfer contacts are sufficient.) Generalise this to n switches, any one of which can turn the lamp on or off.

14. PUZZLER You are in a well stocked electronics lab but have only four relays, each with only one normally open contact. It is necessary to light a bulb for the condition AB + AC + CD + no others. What can you do?

6

Flip-Flops as Memory Elements

Learning Objectives

After studying this chapter, you will know about:

- Flip-flops as the basic memory element in sequential circuits.
- Various types of flip-flops and how they evolved in the latter half of 20th century.
  - S-R flip-flop
  - Clocked S-R flip-flop
  - J-K flip-flop
  - Master slave J-K flip-flop
- Reason for race-around-condition in J-K flip-flop and how it is overcome.
- Conversion from one type of flip-flop to the other type and the required interface logic.
- Triggering of flip-flops.
- D-Latch Vs D-flip-flop.

In combinational circuits presented so far, no memory was required since the output depended solely on the present values of the inputs, that is, on the present input combination only. In contrast, sequential circuits have to produce outputs depending not only on the present inputs but also on a finite number of past inputs. Thus, there is need to have memory associated with logic circuits. A **flip-flop** is a simple logic circuit which has the capability of storing (memorising) one bit of information. The output of a flip-flop can remain in any one of two possible states called 0 or 1 levels. A low voltage of 0 volts may be assigned to logic 0 level and a voltage of + 5 volts may typically represent logic 1 state. There are several types of flip-flops. The subject has evolved during the second half of the 20th century. It is the purpose of this chapter to expose the learner to the principles of operation of various types and understand their evolution. We will concern ourselves with the structure and behaviour in this chapter and we will put them into action by using them in sequential circuits in later chapters.

The reader should know the following types of flip-flops thoroughly to be able to design sequential machines.

1. S-R (Set-Reset or Clear) flip-flop
2. Clocked S-R flip-flop
3. J-K flip-flop
4. J-K master-slave flip-flop
5. T (Toggle) flip-flop
6. D (data or delay) flip-flop

A flip-flop has the property to remain in one state indefinitely until it is commanded to change by input signals called **excitations.** For this reason, it is referred to as a **latch.** A latch should preserve or store the last input received. A **D-latch,** which is commonly used, is described subsequently.

6.1 BASIC STRUCTURE OF A FLIP-FLOP

Two NAND gates cross-coupled so that the output of one becomes the input of the other gate is shown in Fig. 6.1. The outputs are usually marked Q, Q′. For the present, assume that they are 2-input gates with both the inputs shorted. So they behave as single-input gates. Note that Q and Q′ are connected through a NAND gate.

If Q is at 0 level, it forces Q′ to assume logic 1 level which confirms the 0 level of Q. If Q happens to be at logic 1 level, Q′ will be forced to be at 0 level which in turn reiterates the 1 level at Q. We conclude that the outputs of this circuit will be stable at either one of two levels 0 or 1 called **state** until and unless perturbed by some external excitation. This circuit is called a **flip-flop, binary** or **one-bit-memory,** which has two mutually complementary outputs.

Replace the NAND gates in Fig. 6.1 with NOR gates and the same behaviour results. But, due to some technological constraints, NAND is preferred.

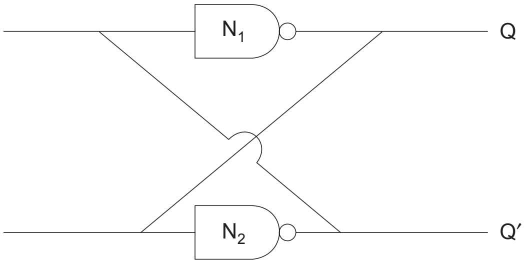
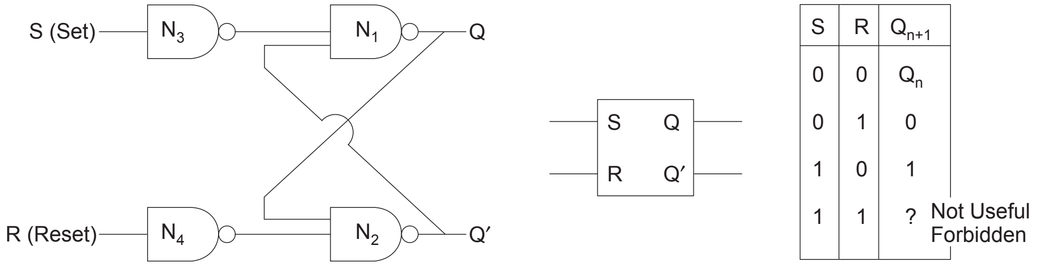


Fig. 6.1 Two NAND's cross-coupled form binary, flip-flop, one–bit–memory

6.2 S-R FLIP-FLOP

Look at the circuit of S-R flip-flop shown in Fig. 6.2(a). Notice that it consists of four 2-input NAND gates. N_1 and N_2 form the flip-flop and N_3 and N_4 with short-circuited inputs give us the facility of having excitation signals called set (S) and reset (R), also called Clear (c). Fig. 6.2(b) denotes the symbol and (c) shows the behaviour of the S-R flip-flop in the form of a Truth Table (also called **Characteristic Table**).



(a) The logic circuit (b) Symbol (c) Truth Table

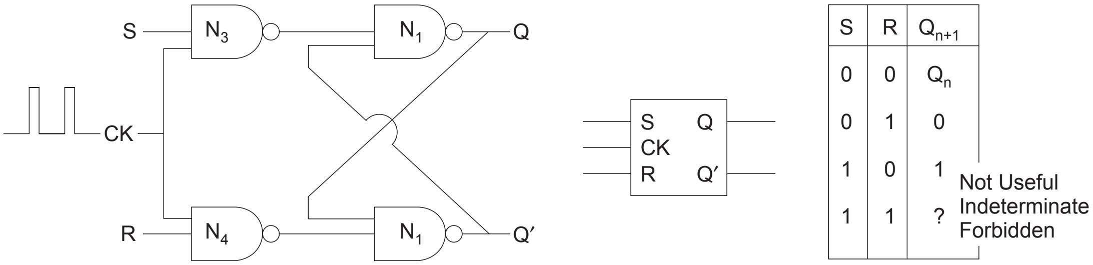
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

Fig. 6.2 S-R flip-flop

The reader should develop familiarity with the notation. Q_n denotes the state of flip-flop before the application of inputs and Q_{n+1} refers to the state attained by the flip-flop after the application of inputs. For the input combination SR = 00, observe that the outputs of N_3 and N_4 will be 1s and hence Q and Q′ will remain where they were, which is shown in the Truth Table by $Q_{n+1} = Q_n$. If we set SR = 01, the output of N_3 becomes 1 and the output of N_4 becomes 0 which forces Q′ to become 1. Both the inputs of N_1 will then be 1 and hence Q becomes 0. This is referred to as resetting or clearing the flip-flop to 0. In a similar manner, if SR = 10, Q is forced to become 1 and Q′ becomes 0 called setting the flip-flop to 1. Consider the last input combination namely, SR = 11. The input excitation commands the flip-flop to set and reset at the same time, which is impossible. The behaviour of the flip-flop will then be erratic and unpredictable. This combination of SR = 11 is forbidden and hence is not to be used by the designer as it is not useful.

6.3 CLOCKED S-R FLIP-FLOP

In most practical situations, the sequence and timing of the various steps has to be fixed by the designer. Generation of partial products one after another and adding them in a multiplier is only one of the many processes encountered in digital systems. For this purpose we use a pulse generator, which produces periodically recurring pulses at a pre-designed frequency. This is referred to as a **clock** by digital designers. Only when the clock pulse occurs, do the desired operations take place as they are enabled using AND gates wherever needed. All flow of information occurs only when the clock pulse occurs. Such machines (circuits) controlled by a clock are called **synchronous sequential machines (circuits).** There are yet other machines (circuits) which do not require a clock—for example, counting the member of visitors in a library, counting



(a) The logic circuit (b) Symbol (c) Truth Table

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

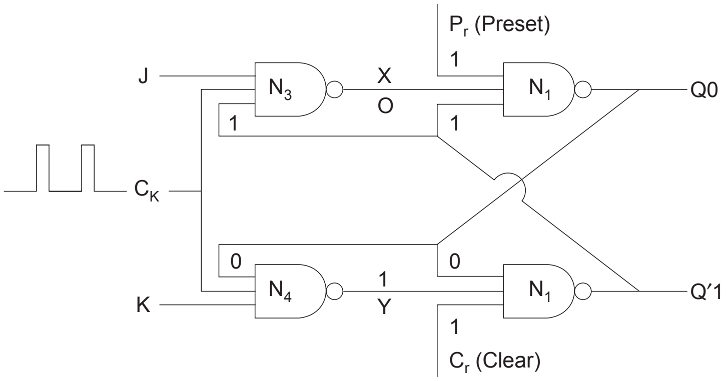
Fig. 6.3 Clocked S-R flif-flop

cosmic particles impinging on the earth's atmosphere, and so on. Machines without a clock are called asynchronous sequential machines, discussed in a later chapter.

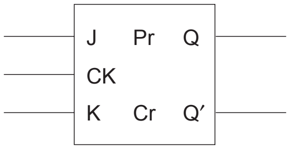
The circuit of a **clocked S-R flip-flop** is shown in Fig. 6.3(a), the symbol in (b), and the Truth Table in (c). The structure and behaviour is identical with the S-R flip-flop of Fig. 6.2 except that the second inputs of N_3 and N_4 are connected to the system clock. It is easy to realise that only during the clock pulse are gates N_3 and N_4 enabled and only then do the signals at S and R flow to the outputs and cause change of state as mentioned in the Truth Table. Assigning the values 11 to SR inputs is still forbidden whether it is clocked or asynchronous.

6.4 J-K FLIP-FLOP

In order to overcome the constraint that we cannot use the assignment of SR = 11, a clever strategy is adopted in J-K flip-flop, shown in Fig. 6.4, in which the designer exploits the '11' assignment for making the flip-flop to toggle, that is, to change state at the occurrence of the clock pulse. The letters J and K are chosen by convention; they do not represent any words like set for S and R for reset. Now, take a keen look at the circuit given in Fig. 6.4(a). All the four NAND gates are 3-input gates and there is global feedback path from Q′ to the 3rd input of N_3 and from Q to the 3rd input of N_4. Because of this arrangement, the flip-flop toggles when J = 1,



(a) Circuit



(b) Symbol

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

(c) Truth Table

For normal operation: Set P_r = 1 and C_r = 1.

DEFECT: Race-Around-Condition makes the final sate unpredictable. To overcome this defect a Master-Slave arrangement is used.

Fig. 6.4 J-K flip-flop

K = 1, and clock pulse occurs. If Q = 0 and Q′ = 1, then all the inputs of N_3 will be at 1 during the clock pulse, which forces a 0 at x which, in turn makes Q = 1; hence Q′ becomes 0. A similar argument holds when Q = 1 and Q′ = 0. This is depicted in the Truth Table of Fig. 6.4(c).

Notice that there are two more leads—**preset (P_r)** and **clear (C_r)**—provided in the solid state integrated circuit chip version of a J-K flip-flop. For normal operation, they are kept at $P_r = C_r = 1$. So they do not effect the NAND gates N_1 and N_2. P_r and C_r are used only for initialising purpose such as in counters and registers. This flip-flop has an inherent defect, which is discussed next.

6.5 RACE AROUND CONDITION

Consider the assignment of excitations J = K = 1. If the width of the clock pulse t_p is too long, the state of the flip-flop will keep on changing from 0 to 1, 1 to 0, 0 to1 and so on and, at the end of the clock pulse, its state will be uncertain. This phenomenon is called the **race around condition.** The outputs Q and Q′ will change on their own if the clock pulse width t_p is too long compared with the **propagation delay** of each NAND gate τ. Table 6.1 shows how Q and Q′ keep on changing with time if the clock pulse width t_p is greater than τ. Assuming that the clock pulse occurs at t = 0 and $t_p >> \tau$, the following expressions hold before and during t_p in the J-K flip-flop of Fig. 6.4. Note that f(t-τ) is f(t) delayed by τ.

During no pulse

$$\left.\begin{array}{l} X(t) = 1 \\ Y(t) = 1 \end{array}\right\} \Rightarrow \begin{array}{l} \text{No change} \\ \text{in Q and } \overline{Q} \end{array}$$

During the clock pulse of width t_p with J = k = 1

$$X(t) = \overline{J \bullet C_K \bullet \overline{Q(t-\tau)}} = \overline{\overline{Q(t-\tau)}},\ Q(t) = \overline{x(t-\tau) \bullet \overline{Q(t-\tau)}}$$

$$Y(t) = \overline{K \bullet C_K \bullet Q(t-\tau)} = \overline{Q(t-\tau)},\ \overline{Q}(t) = \overline{y(t-\tau) \bullet Q(t-\tau)}$$

The reader is advised to work out the transitions assuming τ as the propagate delay through each NAND gate and that the clock pulse width t_p is greater than τ. Notice in Table 6.1 that the change of state takes at least 2τ. If the flip-flop is initially at QQ′ = 01, then it is easily seen that the transition will follow the sequence of logic levels for each τ as given below.

$$\begin{tikzcd} QQ' = 01 \arrow[r] & 01 \arrow[r] \arrow[rr, bend right] & 11 \arrow[r] & 10 \arrow[r] \arrow[rr, bend right] & 11 \arrow[r] & 01 \ldots \end{tikzcd}$$

If initially the flip-flop were in QQ′ = 10, the transitions will take the following path.

$$\begin{tikzcd} QQ' = 10 \arrow[r] & 10 \arrow[r] \arrow[rr, bend right] & 11 \arrow[r] & 01 \arrow[r] \arrow[rr, bend right] & 11 \arrow[r] & 10 \ldots \end{tikzcd}$$

We thus conclude that the state of the flip-flop keeps on complementing itself for every 2τ. The clock pulse width should be such as to allow only one change to complement the state and not too long to allow many changes resulting in uncertainty about the final state. This is a stringent restriction which cannot be ensured in practice. In order to relax this restriction about the time delays, the **master-slave** configuration emerged, which is discussed next.

Let τ be the propagation delay of the NAND gate. Follow the transitions indicated.

X becomes the complement of previous Q′

Y becomes the complement of previous Q

Q is the complement of the product of X AND Q′ of the previous row.

Q′ is the complement of the product of Y AND Q of the previous row.

Table 6.1 *Flow of Signals in Race around Condition*

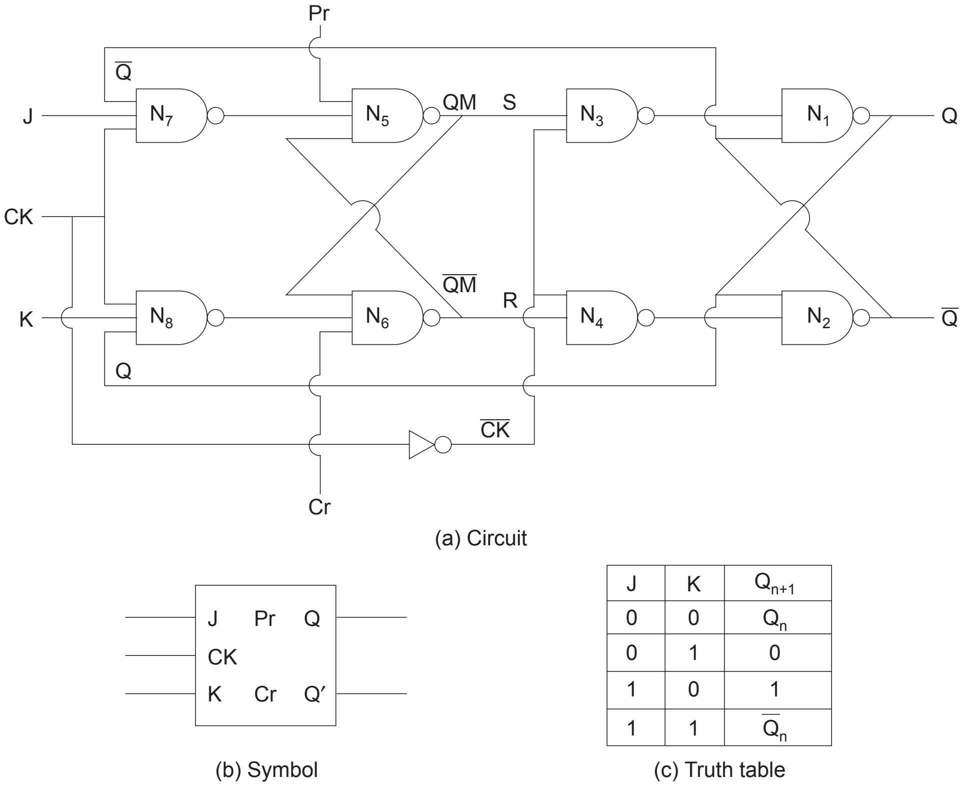
Time		X	Y	Q	Q′	
Initial	$t < 0$:	1	1	0	1	Initial state assumed
$t \geq 0$	$t = \tau$:	0	1	0	1	Pulse is present for $t_p > \tau$
	$t = 2\tau$:	0	1	1	1	
	$t = 3\tau$:	0	0	1	0	←
	$t = 4\tau$:	1	0	1	1	
	$t = 5\tau$:	0	0	0	1	←
	$t = 6\tau$:	0	1	1	1	
	$t = 7\tau$:	0	0	1	0	←
	$t = 8\tau$:	1	0	1	1	

6.6 MASTER-SLAVE J-K FLIP-FLOP

Look at Fig. 6.5(a) wherein the circuit of the J-K master-slave flip-flop is shown. Notice that it comprises two flip-flops in cascade; the first one is a J-K flip-flop followed by a clocked S-R flip-flop. The NAND gates N_5, N_6, N_7, N_8 form a J-K flip-flop with the outputs marked as Q_M and Q'_M called master. The gates N_1, N_2, N_3, N_4 form a clocked S-R flip-flop whose outputs are marked Q and Q′ and referred to as the slave flip-flop. Notice, in particular, two aspects. Firstly, the clock pulse is inverted and fed to the slave flip-flop. Secondly, the global feedback path is from the outputs of the slave to the inputs of the master. During the clock pulse, the master is active and Q_M and Q'_M change as commanded by the excitations J, K. After the clock pulse, that is, on the lagging edge of the pulse, the slave flip-flop becomes active and the values of the master Q_M and Q'_M are transferred to the slave as Q and Q′ respectively. Global feedback cannot cause the race around condition as the loop is broken by inverting the clock pulse before enabling the slave. Finally, it must be borne in mind that the slave stores the data until disturbed by future command through the J-K leads. All the synchronous sequential machines invariably employ master-slave J-K flip-flops only. Other types of flip-flops can be obtained by modification using additional logic gates.

Excitation Table for J-K Flip-Flop

For each of the possible transitions 0 → 0, 0 → 1, 1 → 0 and 1 → 1, what should be the values of the leads J and K of the flip-flop? Achieving the given transitions is a synthesis problem. For simplicity, let us develop a simple notation, which will be used in this book. In this notation, a change of state is put in focus by having a cap on the value such as ↑ (read as capped one) or θ (read as capped zero). Look at Table 6.2. Remember that



(a) Circuit

(b) Symbol

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

(c) Truth table

Fig. 6.5 J-K master-slave flip-flop

assigning JK = 11, changes the state of the flip-flop on the occurrence of clock pulse. For the transition of state from 0 to 0 marked as 0 → 0, we may use JK = 00 or JK = 01 which means that J = 0 and K = ϕ, a don't care condition. Likewise, for the transition 0 → 1, indicated by capped one as ↑, the excitations JK should be assigned 10 or 11, which is indicated as 1ϕ. The other rows in Table 6.2 are similarly filled.

Table 6.2 *Excitation Table of J-K Flip-Flop*

Transition Required	Notation	Excitation	
PS NS		J	K
0 → 0	0	0	ϕ
0 → 1	↑	1	ϕ
1 → 1	1	ϕ	0
1 → 0	θ	ϕ	1

6.7 CONVERSION OF J-K FLIP-FLOP INTO T FLIP-FLOP

The toggle (T) flip-flop has the property that if T is at 1, it changes state on the occurrence of a clock pulse and if T is at 0, the flip-flop remains in the same state regardless of the clock pulse. The Truth Table is given below. For the sake of comparison, the truth table of J-K flip-flop is also given alongside.

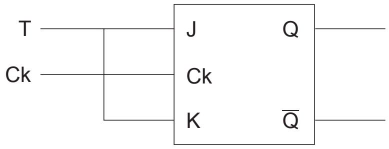
Truth Table of J-K flip-flop

J	K	Q_{n+1}
0	0	Qn
0	1	0
1	1	$\overline{Q}_n$
1	0	1

Truth Table of T flip-flop

T	Qn + 1
0	Qn
1	$\overline{Q}_n$

Comparing the truth tables of J-K and T flip-flops, it is easily realised that if J and K are short circuited and shown as the only input, it behaves as a T-flip-flop.



Excitation Table for a T flip-flop

Transition required	T
For all 0's & 1's	0
For all θ's & ↑'s	1

Fig. 6.6 Conversion of J-K to T flip-flop

Note: If T flip-flop is used in a sequential circuit, T should be assigned to 1 whenever a change of state is required and 0 if no change is desired. This is summarised in the excitation table.

Conversion of J-K Flip-Flop into D Flip-Flop

The Delay (D) flip-flop, also called data flip-flop, has the property that the next state of the flip-flop will become equal to D on the occurrence of the clock pulse. The Truth Table is given below. As the D flip-flop is obtained by a simple modification of J-K flip-flop, the Truth Table of the latter is also given alongside for the sake of comparison.

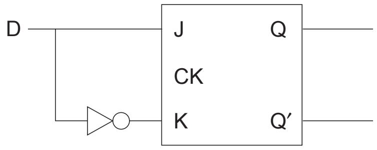
Truth Table of J-K Flip-Flop

J	K	Q_{n+1}
0	0	Qn
0	1	0
1	1	$\overline{Q}_n$
1	0	1

Truth Table of D Flip-Flop

D	Q_{n+1}
0	0
1	1

Comparing the Truth Tables of the two flip-flops, one can realise that if D goes directly to J lead and D′ is fed to K lead, the set-up then works as a D flip-flop as shown below.



Excitation table for a D flip-flop

Transition required	D
0 Or θ	0
1 Or ↑	1

Fig. 6.7 Conversion of J-K into D flip-flop

Note: If D flip-flop is used, the excitation D must be assigned as equal to the next state required. If the next state of the flip-flop is called Y (same as Q_{n+1}) then, the excitation table is filled by assigning D = Y.

6.8 CONVERSION OF T TO D-FF

You have seen that the D flip-flop is the simplest and the most convenient to use as no special effort is required to synthesise the Boolean expression for the excitation D as it is given simply by D = Y where Y indicates the next state. We have already learnt how to convert J-K flip-flop into D flip-flop. Sometimes, we may come across situations when T flip-flops are available but we want a D flip-flop. Let us agree that y (lower case letter) represents the present state and Y (Capital) denotes the next state of the flip-flop.

Look at Fig. 6.8 below. The T flip-flop is given but we want to use it as a D flip-flop. We should consider T as the output and D and y as the inputs to the additional logic circuit interfacing between D and T and synthesise T as a function of D and y. We use D and an interface to excite T. Now, look at the Truth Table. For the combination Dy = 00, the next state Y has to be same as D, that is, 0, which means that the required transition is 0 → 0, which implies T should be assigned 0. For Dy = 01, the next state Y has to be 0; the transition has to be 1 → 0, which implies that T has to become 1. Other entries in the Truth Table are filled in a similar manner. Notice that the column Y is used for the purpose of illustration. Notice further that we used our notation of ↑ and θ to focus attention on the change of state. It is now easy to synthesise T and construct the interface circuit.

Truth Table

D	y	Y ⇒ T	
0	0	0	0
0	1	θ	1
1	1	1	0
1	0	↑	1

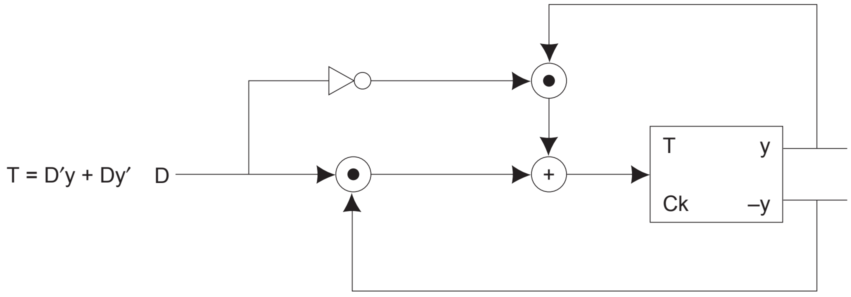


Fig. 6.8 Interface to convert T to D flip-flop

Truth Table

T	y	Y ⇒ D	
0	0	0	0
0	1	1	1
1	1	θ	0
1	0	↑	1

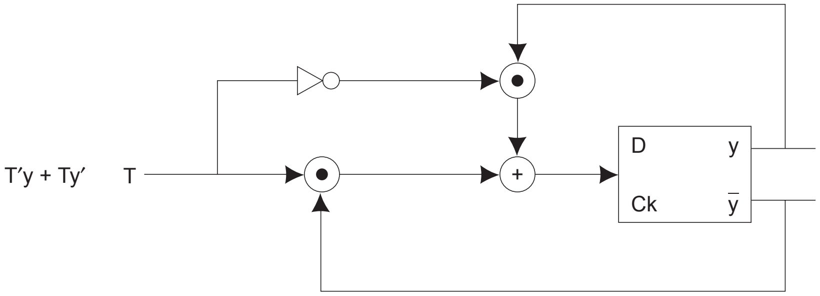


Fig. 6.9 Interfacing for conversion of D to T flip-flop

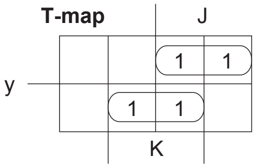
Conversion of D to T FF

Sometimes we may require a T flip-flop but only D flip-flops are available at hand. Then, we may develop a simple interface logic circuit to convert the given D flip-flop into a T flip-flop. This time, T and y have to be taken as inputs and produce D as the output. In other words, we have to use T signals and an interface to excite D flip-flop. Remember that T = 1 produces a change of state and the excitation D = Y, the next state variable. For the combination T = 1, y = 1, for example, the next state Y has to be 0 which implies that we should ensure that excitation D = 0. The Truth Table and the circuit are given below.

6.9 CONVERSION OF T TO J-K FF AND D TO J-K FF

Normally, we do not expect a situation when we need to convert a given T flip-flop into a J-K flip-flop. Even so, it is an interesting academic exercise to work out an interface logic to use J-K signals and excite a T flip-flop for the required transitions. The Truth Table shows the inputs as J, K, y and the output as T. For each combination of inputs, the next state Y is also indicated for reference purpose and the value of T deduced. The conversion interface is shown in the circuit given below (see Fig. 6.10).

J	K	y	Y ⇒ T
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	θ ⇒ 1
1	0	0	↑ ⇒ 1
1	0	1	1
1	1	0	↑ ⇒ 1
1	1	1	↑ ⇒ 1



T = Σ(3, 4, 6, 7)
= Jy′ + Ky

Does this expression cause hazardous behaviour? Think it over.

T to J-K

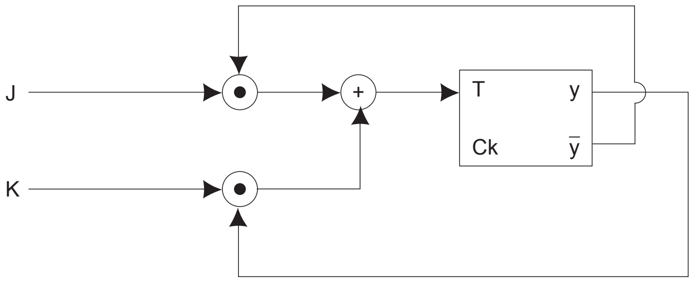
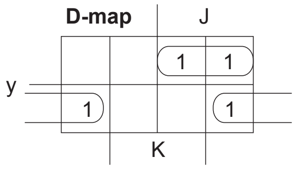


Fig. 6.10 Interface to convert T to J-K flip-flop

Conversion of D FF to J-K FF

A similar working is shown below for using J-K to excite D (refer to Fig. 6.11).

J	K	y	Y ⇒ D
0	0	0	0
0	0	1	1 ⇒ 1
0	1	0	0
0	1	1	0
1	0	0	1 ⇒ 1
1	0	1	1 ⇒ 1
1	1	0	1 ⇒ 1
1	1	1	0



$T = \Sigma(1, 4, 5, 6)$

$= Jy' + K'y$

Does this cause a hazard?
Think it over.

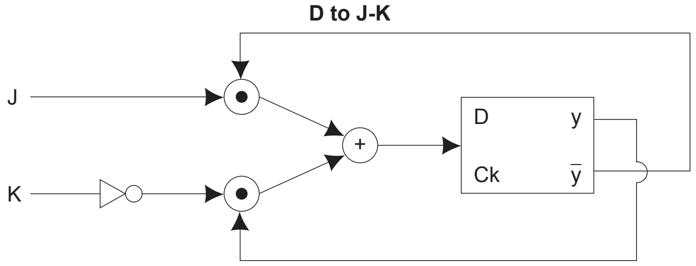


Fig. 6.11 Interface to convert D to J-K flip-flop

6.10 TRIGGERING OF FLIP-FLOPS

A momentary change of signal level and return to the initial level is referred to as a “trigger”. Flip-flops do need some kind of triggering to change their state. Clocked flip-flops are triggered by pulses. A pulse is a signal with two levels at different times, indicated in Fig. 6.12. Pulse width is usually indicated as t_p. The logic levels 0 and 1 may be assigned to voltages, typically 0 volts and 5 volts respectively. The figure shows both the positive going pulse and negative going pulse with the leading (first in time) edge and the lagging edge (later in time) marked as positive edge and negative edge.

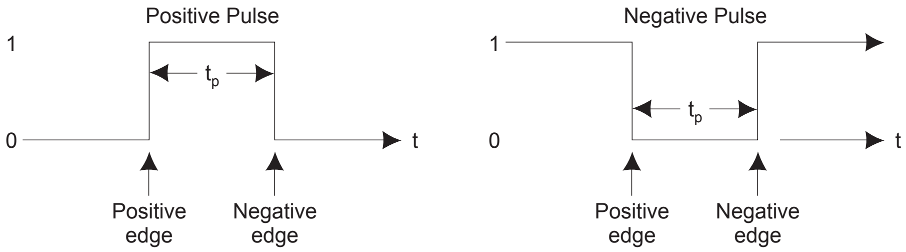


Fig. 6.12 Pulse shape

Edge triggered FFs are used for storage in digital computers and systems and in electronic instruments and counters. Fig. 6.13(a) shows an RC differentiated clock pulse and the output spikes used in triggering an S-R flip-flop. Notice in Fig. 6.12 that the narrow positive spikes enable the AND gates momentarily and trigger the S and R inputs. The negative spikes do nothing. Sometimes, triggering on the negative edge is better suited to the application. An inverter can complement the pulse before it reaches the AND gates.

The **preset** (P_r) and **clear** (C_r) are asynchronous inputs to the flip-flop. These are used for setting all the flip-flops in a register to 1s or clearing them by resetting to 0s for the purpose of initialisation. These operations are done independently of the clock pulses. Fig. 6.13(b) shows these functions for an S-R flip-flop.

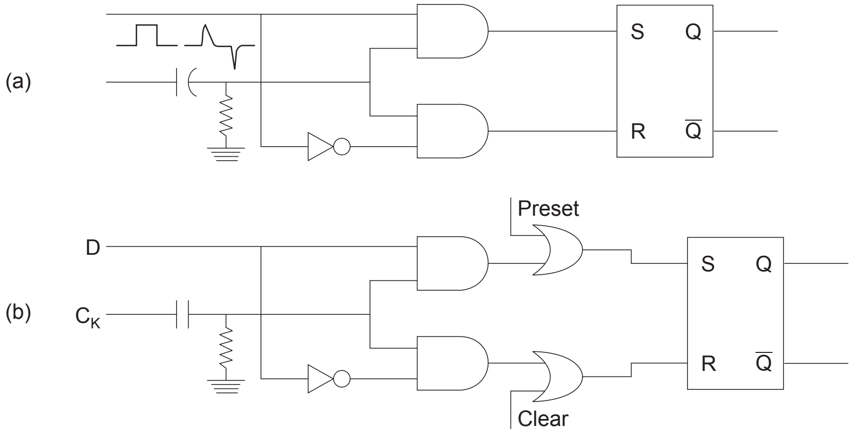


Fig. 6.13 Edge triggering of flip-flops

The reader should note that the RC differentiating circuits are used here only for the purpose of illustrating the concept of **edge-triggering,** but the actual flip-flops in the form of integrated circuit chips do not use RC circuits to obtain narrow spikes. Instead, a variety of direct coupled designs are used. Fig. 6.14 shows the symbols used for a D flip-flop with clock input together with **preset** and **clear** leads. Notice that positive edge triggering is indicated by a small triangle on the clock input shown in (a) and the negative edge triggering is indicated by a small bubble before the triangle on the clock input shown in (b). Sometimes, logic level 0 may be used for **preset** and **clear** leads. Such active low inputs are normally indicated by a small bubble shown in (c).

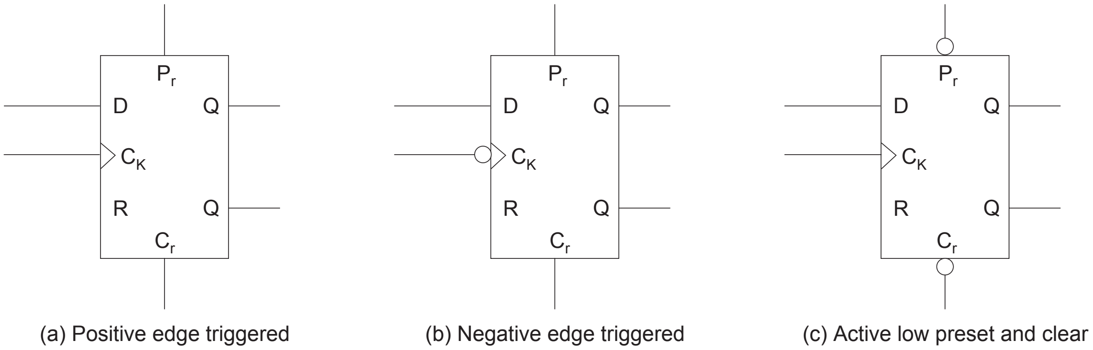
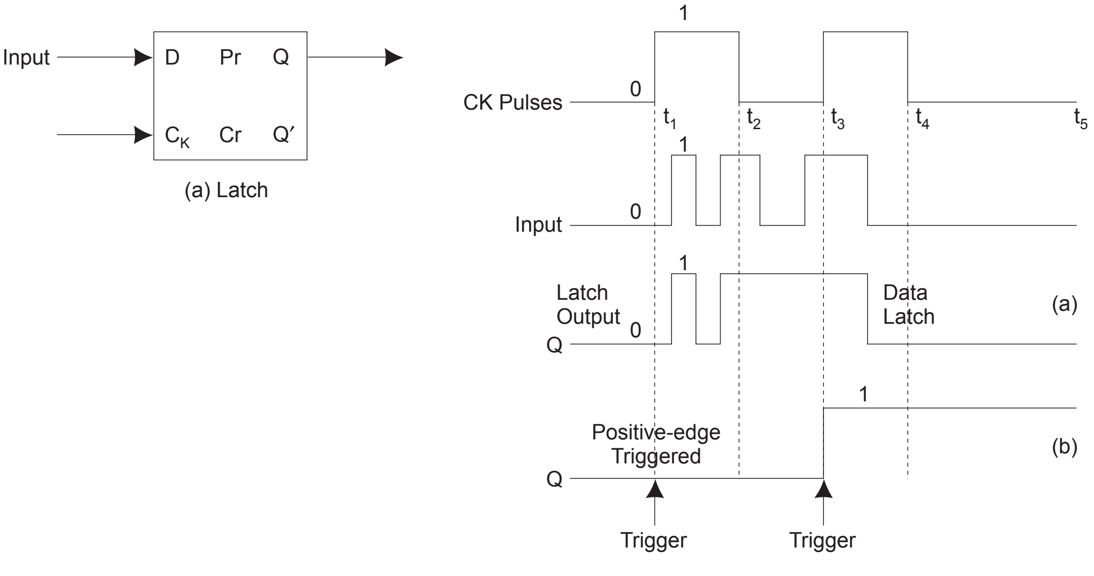

P_r
D
Q
C_K
R
Q
C_r
(a) Positive edge triggered
P_r
D
Q
C_K
R
Q
C_r
(b) Negative edge triggered
P_r
D
Q
C_K
R
Q
C_r
(c) Active low preset and clear

Fig. 6.14 D flip-flop symbols

6.11 THE D–LATCH

The characteristic feature of a **latch** is to follow the input during the clock pulse and preserve, or store, the last value received after the clock pulse is gone, for future use or reference. The excitations (inputs) of flip-flops are sometimes referred to as triggering inputs. Triggering normally means a single change of state. Some flip-flops respond to the leading edge (positive going edge) of the clock and others respond to the lagging edge (negative going edge) of the clock pulse. The M-S J-K flip-flop clearly goes to the desired state on the lagging edge of the input clock pulse.

The distinction between a D-latch and a positive edge triggered flip-flop is shown in Fig. 6.15. Latch output follows the input during the clock pulse and retains the last value when the pulse is gone. Positive edge-triggered flip-flop responds to the value of the input at the positive edge of the clock pulse.



(a) Latch

(b) Positive-edge-triggered flip-flop

Fig. 6.15 Output waveforms

SUMMARY

Flip-flops constitute the basic memory elements used in sequential circuits. Some are particularly suited for specific applications. Starting from cross-coupled NAND gates forming a flip-flop, evolution through S-R, clocked S-R, J-K and M-S-J-K flip-flops has been covered. The race-around condition arising in J-K flip-flop and the detailed timing chart has been discussed. How the race-around condition is eliminated using M-S configuration has been explained. Conversion from one type of flip-flop to another type and the associated logic is discussed. The distinction between a D-latch and D-flip-flop is illustrated.

KEY WORDS

- Flip-flop
- Excitation
- Latch
- D-Latch
- Binary
- Characteristic table
- Clock
- Synchronous sequential machines/circuits
- Clocked S-R flip-flop
- Preset (P_r)
- Clear (C_r)
- RACE AROUND CONDITION
- Propagation delay
- Master-slave flip-flop
- State
- Edge-triggering

REVIEW QUESTIONS

1. Convert a given J-K flip-flop into a D flip-flop using additional logic if necessary.
2. In a master-slave J-K flip-flop, J = K = 1. What will be the state Q_{n+1} after the clock pulse?
3. A J-K flip-flop is invariably used in a master-slave configuration. Why?
4. The output Q_n of a J-K flip-flop is 1. It has to change to 0 after the clock pulse. What should be the excitations J and K?
5. Which of the following conditions must be met to avoid race-around problem in a J-K (not M-S) flip-flop if τ is the propagation delay of a NAND gate and t_p is the clock pulse width?

   a) $3\tau \le t_p < 4\tau$ b) $\tau \le t_p \le 2\tau$ c) $2\tau \le t_p < 3\tau$ d) $t_p \le 2\tau$
6. The following flip-flop is used as a latch.

   a) J-K flip-flop b) Master-slave J-K flip-flop
   c) T flip-flop d) D flip-flop
7. Preset and clear inputs in a flip-flop are used for making Q = _______ and _______ respectively.
8. Which of the following input combinations is not allowed in an SR flip-flop?

   a) S = 0, R = 0 b) S = 0, R = 1 c) S = 1, R = 0 d) S = 1, R = 1
9. When an inverter is placed between the inputs of an SR flip-flop, the resulting flip-flop is

   a) JK flip-flop b) D flip-flop
   c) T flip-flop d) Master-slave JK flip-flop
10. The functional difference between an SR flip-flop and a JK flip-flop is that

    a) JK flip-flop is faster than SR flip-flop b) JK flip-flop has a feedback path
    c) JK flip-flop accepts both inputs 1 d) JK flip-flop does not require external clock
11. A flip-flop can store

    a) One bit of data b) Two bits of data
    c) Three bits of data d) Any number of bits of data
12. For an input pulse train of clock period T, the delay produced by n stage shift register is

    a) (n – 1) T b) NT c) (n + 1) T d) 2nT
13. An n stage ripple counter can count up to

    a) 2^n b) $2^n - 1$ c) n d) 2^{n-1}

14. Distinguish between latch and edge-triggered flip-flop.
15. Convert a J-K flip-flop into a T flip-flop.
16. Convert a J-K flip-flop into a D flip-flop.
17. Convert a T flip-flop into a D flip-flop.
18. Convert a D flip-flop into a T flip-flop.
19. What is a register?
20. What is the cause for the race around phenomenon in a J-K flip-flop?
    a) Closed loop feedback
    b) Regenerative feedback
    c) High voltage gain
    d) Thermal runaway of the devices
21. What is a trigger?
22. What is meant by triggering of a flip-flop?
23. J-K M-S flip-flop comprises the following configuration.
    a) S-R flip-flop followed by an S-R flip-flop
    b) S-R flip-flop followed by a J-K flip-flop
    c) J-K flip-flop followed by a J-K flip-flop
    d) J-K flip-flop followed by an S-R flip-flop
24. In J-K M-S flip-flop, race-around is eliminated because of the following reason:
    a) Output of slave is fed back to input of master
    b) Output of master is fed back to input of slave
    c) While the clock drives the master, inverted clock drives the slave
    d) The clock drives the slave and the inverted clock drives the Master
25. The Boolean expression for the next state (Y_i) as a function of the present state (y_i) and the inputs (excitations in the case of a flip-flop) is referred to as **Characteristic equation** for the flip-flop which will be valid only when the appropriate restrictions are observed. For an S-R flip-flop, the characteristic equation is given by

$$Y = S + R \bullet y \text{ if } S \bullet R = 0$$

Find the characteristic equations for J-K, T and D flip-flops.

PROBLEMS

1. Draw the circuit diagram of J-K flip flop with NAND gates with positive edge triggering and explain its operation with the help of a Truth Table. How is the race around condition eliminated?
2. Realise D-latch using S-R latch. How is it different from D flip-flop? Draw the circuit using NAND gates and explain.
3. a) Draw the circuit diagram of master-slave J-K flip-flop and explain its operation with the help of a Truth Table. How is it different from edge triggering? Explain.
   b) Give the transition table for the following flip-flops.
      i) SR flip-flop
      ii) J-K flip-flop
      iii) T flip-flop
      iv) D flip-flop

   Briefly state the salient features of each flip-flop.

4. a) Compare combinational Vs sequential logic circuits.
   b) Define the following terms of a flip flop.
      i) Hold time ii) Set up time iii) Propagation delay time
   c) Draw the circuit diagram of a master-slave J-K flip-flop and explain its operation with the help of a Truth Table.
5. Convert an S-R flip-flop into J-K flip-flop with the feature of preventing race around condition.
6. a) Draw a J-K master-slave flip-flop configuration and explain its operation. Show that the race around condition is eliminated.
   b) Construct a 4-bit shift register using flip-flops and explain its operation.
7. a) Distinguish between synchronous and asynchronous counters giving examples. Mention applications of each type.
   b) Draw the Truth Table and logic diagrams of J-K, R-S, D and T type of flip-flops.
8. What is a race-around condition? How is it overcome in a J-K master-slave flip-flop?
9. Convert a T flip-flop to a D flip-flop using additional logic if required.
10. Draw the circuit of a clocked S-R flip-flop using only NAND gates and indicate the Truth Table.
11. Write the excitation requirements for a J-K flip-flop for state transitions 0 → 0, 0 → 1, 1 → 0, 1 → 1.
12. Convert D flip-flop into a T flip-flop using additional logic gates.
13. An equation expressing the next state Q_{n+1} in terms of the present state Q_n and the excitations of the flip-flop is called a **characteristic equation**. For a J-K flip-flop show that it is given by

    $$Q_{n+1} = JQ_n' + K'Q_n$$

    Find the characteristic equations for

    a) T flip-flop b) D flip-flop
14. Show that the characteristic equation of an S-R flip-flop is given by
    $Q_{n+1} = S + R'Q_n$ with the condition that $S \bullet R = 0$.

7

Synchronous Sequential Circuits

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Design steps for synchronous sequential machine.
- Counters, sequence detection, serial adders etc.
- Model of a finite state sequential machine also called finite automation.
- Ring counter and their applications.
- Multiple registrar operations using multiplexer control.
- Elegance of using shift counters in control applications.

In the previous chapter, some essential components of sequential machines namely, flip-flops used as memory elements, were presented. This chapter deals with the design of synchronous sequential circuits. When represented by a block or a model with inputs and outputs indicated, a sequential circuit is referred to as a **"sequential machine".** Our approach will be to develop the skills of designing sequential machines through examples and finally discuss the formal definition and structure at the end.

Digital computers and digital communication systems are examples of synchronous sequential machines. Such machines are associated with a "clock", or a periodic pulse generator. Signals are transmitted within the system only when the clock pulse occurs. When there is no clock pulse, the machine will be idle and remain in the same state. All flow of information takes place only during the clock pulse.

7.1 DESIGN STEPS

Design of synchronous sequential machines consists of the following steps.

Step I **Word statement** This is the starting point. The purpose of the machine has to be stated in simple, unambiguous words: Care must be taken to ensure that it does not contain any internal conflicts, contradictions, inconsistencies or paradoxes. It should be realisable with a finite number of memory elements.

Step II **State diagram (also called state graph)** This is a pictorial model of the machine and its behaviour. Care should be taken to ensure that the transitions never contradict the word statement. Further, for all possible applicable inputs, state transition and outputs should be specified unless they never occur in which case they become don't care entries denoted by a dash (-) or phi (ϕ).

Step III **State table** Here, the entire information about the machine contained in the state diagram is posted in a tabular form. It is sometimes convenient to work with tables rather than pictures.

Step IV **Reduced Standard Form State Table (RSFST)** In the second step, we may sometimes unwittingly introduce redundant states while the machine can be sent to one of the states already defined. All such redundancies will be eliminated in this step and a minimal state table emerges. Minimization of the number of states of a given machine is important for various reasons and is discussed as a special topic in a subsequent chapter. For the purpose of continuity in the present discussion, one simple technique will be presented.

If the reduced machine is put in a standard form, it would be convenient to compare two machines and decide whether they are identical machines with permuted state names or whether they are different machines.

Step V **State assignment and transitions table** In this step, we assign binary names for the states of the machine. If the machine has eight different states, we need to have three binary values so that each combination refers to a unique state. With n binary variables, 2^n states can be assigned binary names. These binary variables are called **"state variables"** or **"secondary input variables"** to distinguish them from the external primary inputs. Note that the state variables are generated inside the machine. A **transition table** refers to the state table with all the states replaced by their assigned binary names. Thus far, we used the word "machine (M/c)" to describe the behaviour. In the following steps we proceed to realise a logic circuit, which we will call a sequential circuit, comprising flip-flops and other logic gates.

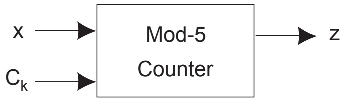
Step VI **Choose the type of flip-flops and form the excitation table** Normally, three types of flip-flops namely J-K, T, and D are used in synchronous sequential machines. Truth Tables, transitions and excitations were discussed in an earlier chapter. Depending on the entries in the transition table, the required excitations are entered in the excitation tables.

Step VII **Excitations maps and the logic functions** Notice that the excitation table will be in a form similar to a Karnaugh Map, with each column denoting one variable. Draw the maps and synthesise the logic functions for each of the excitations as functions of input variables and state variables.

Step VIII **Draw the circuit for the machine** All the above steps are illustrated in the following design examples.

7.2 DESIGN EXAMPLE 1—MODULO-5 COUNTER

Step I **Word statement** A synchronous sequential machine has one input x besides the clock and one output Z. Whenever the clock pulse occurs, the input x has to be sensed. If x is 1, the machine has to act as a counter and increase the count; if x is 0, the count is not altered. For every five 1s sensed, the machine should produce a pulse on the output lead Z and reset the count to 0. The block diagram of the machine and the counting sequence are shown below.



Counting Sequence

Dec count of clock pulses	Binary Code of count		
	y	y	y
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	0	0	0

Step II **State diagram** The machine has to remember the number of times x was at logic 1 level. See Fig. 7.1. Let the initial state of the machine be S_0. While in S_0, if x is 1 when the clock pulse happens, let the machine go to another state S_1. This is indicated by a directed arc (edge) from S_0 to S_1 with 1/0 (input/output) indicated on the arc, which means x = 1 and z = 0 for this transition. Note in particular that the machine being in state S_1 implies that the input x was 1 once, or 6 times, or 11 times, and so on, that is, (5k + 1) times. The reader is advised to follow the state graph closely. Notice that the fifth clock pulse while x is 1 takes the machine from S_4 to S_0 and gives an output pulse Z = 1. Notice that if x is at 0 level, the state of the machine is not altered; this fact is indicated by self-loops at each state. The important aspect is that the machine has to be directed (told) as to what it should do for every applicable input; it should not be left unsaid in the state diagram.

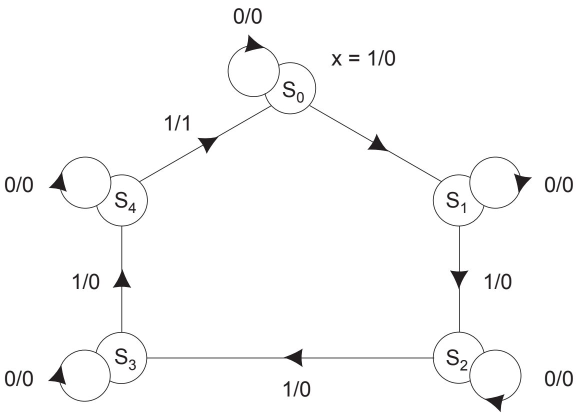


Fig. 7.1 State diagram of a Mod-5 counter

Step III **State table** The information contained in the state graph is posted in tabular form with present state (PS), next state (NS) and output (Z) for each possible input. This results in the state table for the machine.

PS	NS, Z Inputs	
	x = 0	x = 1
S_0	S_0, 0	S_1, 0
S_1	S_1, 0	S_2, 0
S_2	S_2, 0	S_3, 0
S_3	S_3, 0	S_4, 0
S_4	S_4, 0	S_0, 1

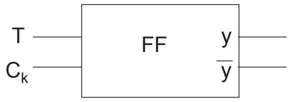
Step IV **Obtain reduced standard form state table (RSFST)** This machine is already in this form. A simple technique will be illustrated in the next example.

Step V **State assignment and transition table** As the machine has five states, we need three binary variables y_2, y_1, y_0. Only five combinations corresponding to the decimal codes 0 to 4 are used and the remaining become don't cares. The letter designations of the states are now replaced by binary codes. In the columns under NS, the change of state is put in focus by using a capped zero (θ) or a capped one (↑).

Step VI **Choose the type of flip-flip and write the excitation matrix** For this example, let us choose T flip-flops, which are preferred for counting applications. The block diagram and the truth table are given below. The transitions and excitations are also given alongside. Notice that T should be made 1 only for capped θs and capped ↑s in the transition table.

Transition Table (also called Matrix)

		PS			NS (Y_2 Y_1 Y_0), Z						
		y_2	y_1	y_0	X = 0			x = 1			
S_0	=	0	0	0	0	0	0	0	0	↑	
S_1	=	0	0	1	0	0	1	0	↑	θ	
S_2	=	0	1	0	0	1	0	0	1	↑	
S_3	=	0	1	1	0	1	1	↑	θ	θ	
S_4	=	1	0	0	1	0	0		0	0,	1



T	Y_{n+1}
0	Y_n
1	$\overline{Y}_n$

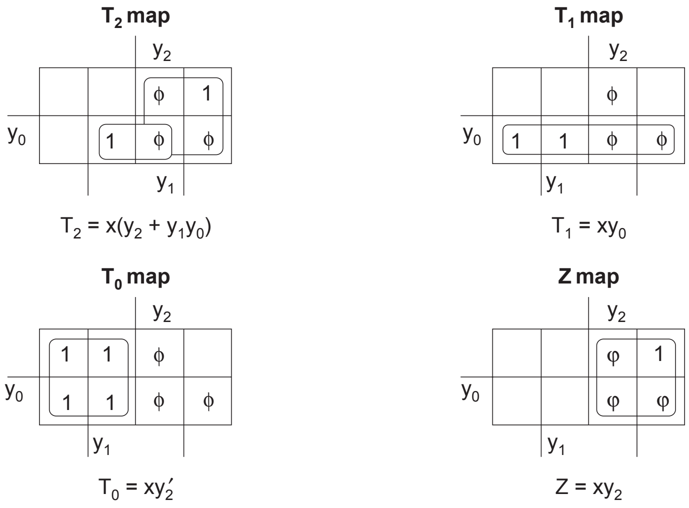
Excitation of a single T-flip-flop

Transition Required	Excitation to be applied
0 =>	T = 0
1 =>	T = 0
θ =>	T = 1
↑ =>	T = 1

Excitation Table (also called Matrix)

y_2	y_1	y_0	T_2 T_1 T_0, Z						
			x = 0			x = 1			
0	0	0	0	0	0	0	0	1	
0	0	1	0	0	0	0	1	1	
0	1	0	0	0	0	0	0	1	
0	1	1	0	0	0	1	1	1	
1	0	0	0	0	0	1	0	0,	1

Step VII **Find the Boolean functions** Find T_2, T_1, T_0 and Z as functions of x, y_2, y_1, y_0. Note that x is 1 in all the following maps. The state variable combinations 101 (decimal code 5), 110 (6), 111 (7) never occur in this machine and are, therefore, marked as ϕ indicating don't care conditions.



$T_2 = x(y_2 + y_1y_0)$

$T_1 = xy_0$

$T_0 = xy_2'$

$Z = xy_2$

Step VIII **Draw the circuit around the three T flip-flops to realise the functions** Notice in Fig. 7.2 that the flip-flops are drawn from right to left to show the count in the natural order with MSB at the extreme left and LSB at the extreme right. All the logic gates are denoted by a circle within which the operator is indicated. Wherever pulse output is desired, the corresponding level output is to be ANDed with the clock pulse which is not explicitly shown but is implicit with all synchronous circuits.

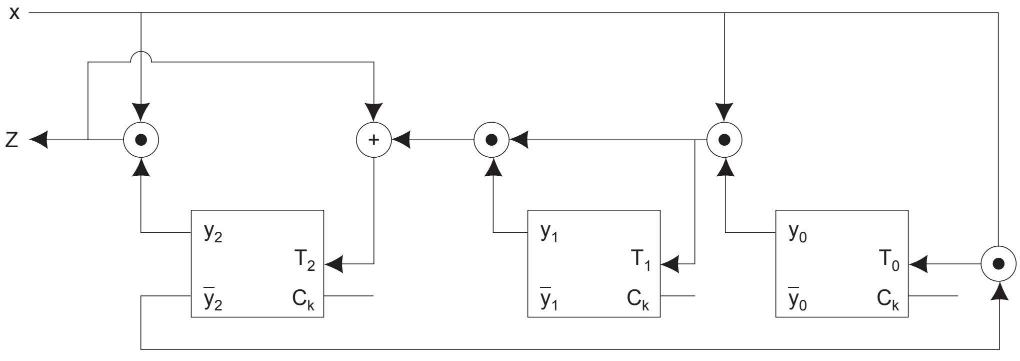


Fig. 7.2 Modulo-5 counter

7.3 DESIGN EXAMPLE 2—SEQUENCE DETECTOR

Step I **Word statement** Design a synchronous circuit with one I/P x and one O/p Z. The output Z should be a pulse coincident with the last 1 of the input sequence 0101. Overlapping sequences are accepted.

Ex. x: 0 1 0 1 0 11

Z: 0 0 0 1 0 1 0

Step II **State diagram** Visualise that the machine has to remember three past inputs. If the past inputs were 010 and the present input is 1, then a pulse should be produced on the output lead Z. Let the initial state be called A. As the sequence to be detected starts with 0, let us send the machine to another state B on input x = 0 with output Z = 0 (no pulse). State B remembers the first 0 occurring on input x lead. From state B, on x = 1, the machine goes to another state C. State C remembers two symbols 01 occurring on x lead. From state C, if x = 0 occurs following 01, let the machine go to state D. State D remembers three symbols 010 occurring on x lead. From D on input 1, let the machine go to another state E giving an output pulse indicated by 1/1 on the directed arrow.

Thus far, we strictly followed the word statement of the problem to produce an output pulse on Z. Now, we have to exhaust all other possibilities and tell the machine what it should do if other inputs occur in each of the states A, B, C, D and E. From initial state A, if a 1 occurs, it is clear that the desired sequence 0101 has not yet started and the machine has to remain in the same state A till the first 0 occurs. Hence we indicate the self loop at A by 1/0.

From state B, if a 0 occurs, this may be the first 0 of the desired sequence 0101 and the description of the state B is to remember the first 0 and hence we indicate it by a self-loop at B and mark 0/0.

Look at state C which remembers the past inputs 01. If a 1 occurs now, clearly the desired sequence is broken and we have to start all over again and send the machine to the initial state A.

Now look at state D which is reached by the symbols 010. If a 0 occurs when in D, it would be wrong to send the machine to the initial state as this may be the first 0 of the desired sequence. Hence the machine should be directed to go to state B characterised by the first 0.

From the state E, if input 0 occurs, it would be correct to send the machine to D as overlapping is acceptable and the next input may be 1. If overlap is not acceptable, where do you send the machine? Think it over. Finally from E on x = 1, the machine has to go to the initial state A and wait there till the first 0 occurs.

Notice that our job in this problem is to detect a 4-bit sequence and the machine has to remember only the past three inputs. Logically, the machine should have only four states including the initial state but we introduced five states without getting into a conflict with the given word statement. The redundant state will be removed in a later step in the design.

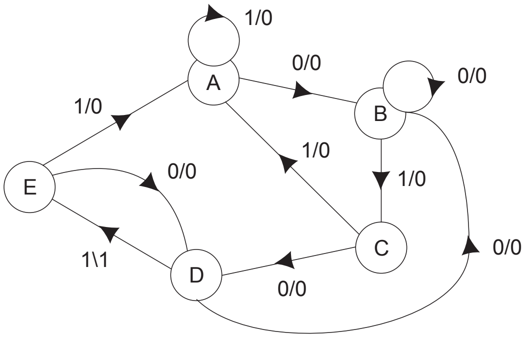


Fig. 7.3 State graph of a sequence detector for 0101

Step III **State table** The pictorial information contained in the state graph is posted in a tabular form below.

PS	NS, Z Input	
	X = 0	X = 1
A	B	A
B	B	C
C	D	A
D	B	E, 1
E	D	A

Step IV **Reduced Standard Form State Table (RSFST)**

Definition Two states P and Q will be said to be equivalent (indistinguishable) if the machine produces identical output sequences for all possible input sequences regardless of whether the starting state was P or Q.

We will now use a simple technique called the "partition method" to this machine to identify equivalent states and remove redundant states. Suppose we do not apply any input. Obviously, we cannot distinguish any one state from the total number of states. We, therefore, show all the states in one bracket and call it the "zero partition" as 0 number of inputs are applied to the machine. Now compare every pair of states and apply one input 0 or 1 and observe whether we get identical outputs. It is easy to note that state D cannot be equivalent to any other state as in this case alone the machine gives an output 1 on the application of input x = 1. Hence we split D from the rest and show it in a separate parentheses in partition P_1, observing outputs only in each column.

$$P_o = (ABCDE)$$

Looking at the o/p in each column, we obtain the partition P_1.

$$P_1 = (ABCE)\ (D)$$

$$P_2 = (AB)\ (CE)\ (D)$$

$$P_3 = (A)\ (B)\ (CE)\ (D)$$

$$P_4 = P_3 \text{ Hence } C = E.$$

Definition Two states will be equivalent if the successor states are also equivalent.

From now on, we compare each pair of states and successor states and try to establish inequivalence. For example, states A and C cannot be equivalent as the successor pair on input x = 0 is BD and we already established inequivalence between B and D and put them in different brackets.

Continuing this line of argument, we get the successive partitions P_2, P_3. We arrive at $P_4 \equiv P_3$ which means, no further equivalences are possible and that there does not exist any input sequence, which distinguishes the states in the same bracket. Therefore we take them as equivalent states.

In this example we conclude that $C \equiv E$ and hence we remove the redundant state E and substitute C for E wherever required. Thus, we obtain the reduced state table given below.

Reduced State Table

PS	NS, Z	
	X = 0	X = 1
A	B	A
B	B	C
C	D	A
D	B	C, 1

To put it in **standard form,** we first enumerate the states in each row and note the **first occurrence** of each state. If they are in natural order, either in alphabetical order or in ascending order–if the states are numbered–we say it is in standard form. Otherwise, by renaming the states, we leave it in **standard form**, which helps in comparing and distinguishing between two different machines.

Enumerate rows and mark first occurrences

A	B	A
B	B	C
C	D	A
D	B	C

It is in standard form as the first occurrences of the states are in natural order.

Step V **State assignment and transition matrix** As the reduced machine has four states, we need two binary variables y_1, y_2 which are also called present state variables. The state table is now rewritten with this assignment. The next state is shown by capital letters Y1, Y2 by notation. Notice that the changes of state variables are put in focus by using capped zeroes and ones.

y_1 y_2	Y_1 Y_2, Z	
	x = 0	x = 1
A = 0 0	0 ↑	0 0
B = 0 1	0 1	↑ 1
C = 1 1	1 θ	θ θ
D = 1 0	θ ↑	1 ↑, 1

Step VI **Choose the type of FF and write excitation matrix.** Let us choose J-K for a change, let us use J-K flip-flops for this example. We have learnt about this in an earlier chapter. The Truth Table and how we should assign the values of J and K are shown below for ready reference.

Truth Table

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	1	Q'_n
1	0	1

Excitation Table for a Single J-K Flip-Flop

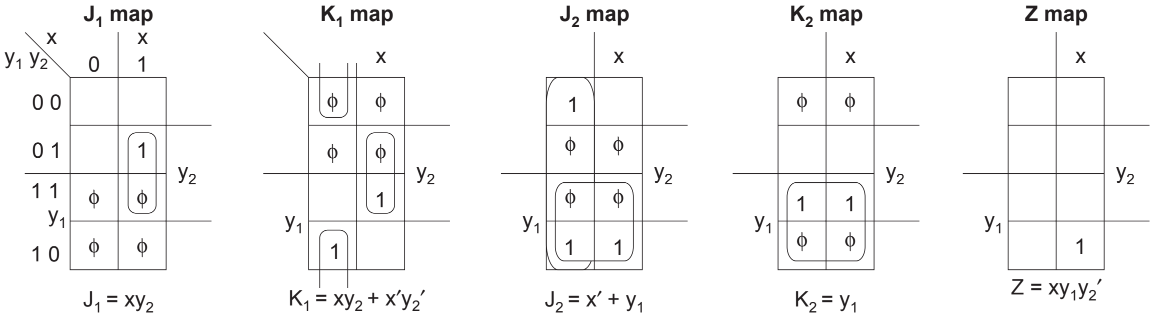
Transition	Symbol	Excitations	
		J	K
0 → 0	0	0	ϕ
0 → 1	↑	1	ϕ
1 → 1	1	ϕ	0
1 → 0	θ	ϕ	1

We should have two flip-flops in this circuit whose outputs are y_1, y_2 which represent the present state. Now substitute for the next state Y_1 Y_2 the values of J_1 K_1 J_2 K_2 depending as the transition required whether 0, ↑, 1 or θ to get the excitation table given. Note that the output Z is also shown after a comma in the table. We have to build the circuit around the two flip-flops. The reader is advised to verify the next two steps, which are routine. J_1, K_1 are determined by Y_1 and J_2, K_2 by Y_2. Each column represents one excitation.

Excitation Table

y_1	y_2	J_1	K_1	J_2	K_2,	Z				
		x = 0				x = 1				
0	0	0	ϕ	1	ϕ	0	ϕ	0	ϕ	
0	1	0	ϕ	ϕ	0	1	ϕ	ϕ	0	
1	1	ϕ	0	ϕ	1	ϕ	1	ϕ	1	
1	0	ϕ	1	1	ϕ	ϕ	0	1	ϕ,	1

Step VII **Write the Boolean expressions.**



$J_1 = xy_2$

$K_1 = xy_2 + x'y_2'$

$J_2 = x' + y_1$

$K_2 = y_1$

$Z = xy_1y_2'$

Step VIII **Draw the circuit neatly.**

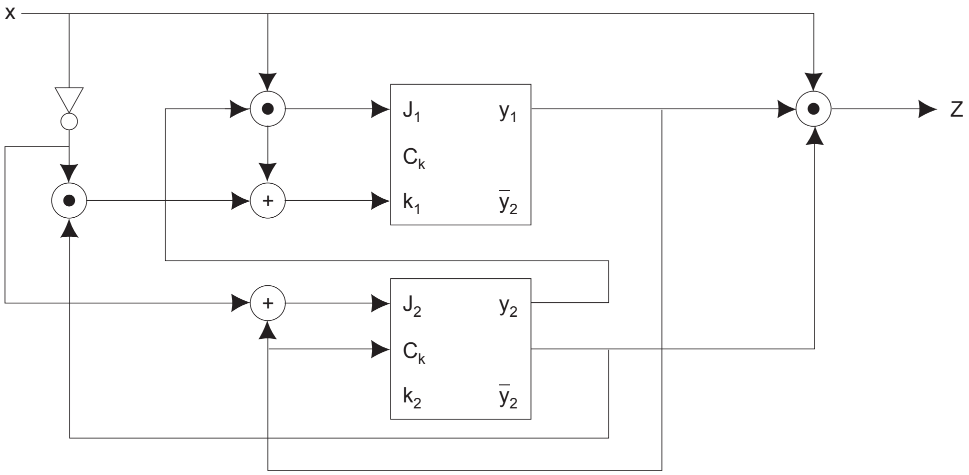


Fig. 7.4 Sequence detector for 0101

7.4 DESIGN OF A SERIAL BINARY ADDER

Step I **Word statement of the problem** A certain synchronous sequential machine has two inputs A and B and one output Z. The inputs and the output are finite length sequences of 0s and 1s. Addition is to be performed serially with the LSBs of A and B arriving first at the inputs in synchrony with the clock pulses. The reader may visualise this as three shift registers, one each for A, B and Z. For each clock pulse the bits are shifted right. The sum bits should appear serially on the output line Z.

The block diagram is shown in Fig. 7.5 along with typical strings of inputs and output.

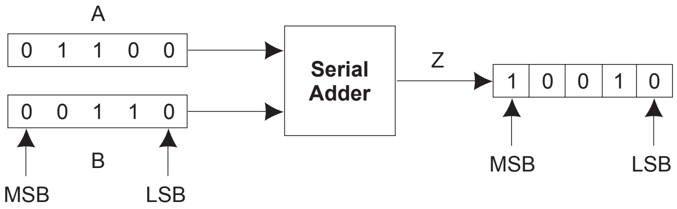


Fig. 7.5 Block diagram of a serial binary adder

Steps II and III **State graph and state table** It is convenient to directly form the state table in this example and then for the sake of completeness, we may draw the state graph, which gives a pictorial representation of the machine's behaviour.

Notice that the output sum bit Z depends on the present inputs and also on whether the immediate past inputs produced a carry 0 or 1. This information about the generation of carry during the previous clock pulse has to be remembered by the machine. For this purpose we define the state of the machine; we send the machine to state S_0 if no carry is generated or to state S_1 if a carry 1 is generated in the process of addition. We, therefore, say that the present inputs and the present state together decide the next state of the machine and the present output.

Let the clock pulses occur at t_0, t_1 … Addition of input bits, A and B, the output bit Z and the next state are indicated below.

Example 7.1

	t_4	t_3	t_2	t_1	t_0		
	0	1	1	0	0	=	A
+	0	0	1	1	0	=	B
	1	0	0	1	0	=	Z
	S_0	S_1	S_1	S_0	S_0	=	Next state

Remember that the LSBs appear at t_0. Working is shown from right to left. Notice that Z = 0 for AB = 00 at time t_0, as we assume that the machine at start is at the initial state S_0 with no carry. With the same inputs AB = 00 at t_4, the output Z is 1 because the machine landed in state S_1 at t_3 before the present inputs are applied at t_4. Notice that the next state at t_i becomes the present state at t_{i+1}. Thus, there is one unit time delay between the present state and next state.

Consider one more example to understand the concepts learnt.

Example 7.2

	t_4	t_3	t_2	t_1	t_0		
	1	0	0	1	1		
+	0	1	0	0	1		
	1	1	1	0	0	=	Z
	S_0	S_0	S_0	S_1	S_1	=	Next state

In this example, the inputs AB = 10 at t_1 as well as t_4 but the output is different. This is because the state of machine at the time of application of inputs was different. All this information is contained in the state table given in Table 7.1.

Table 7.1 *State Table of a Serial Adder*

PS	NS, Z Inputs A B			
	00	01	11	10
S_0	S_0, 0	S_0, 1	S_1, 0	S_0, 1
S_1	S_0, 1	S_1, 0	S_1, 1	S_1, 0

Observe that you fill the S_0 row of the table by simply adding the input bits A and B to 0 as the machine is in state $S_{0,}$ which means there was no carry generated in the previous addition. In the 11 input column of row S_0, the next state has to be S_1 because 1 + 1 produces a carry and Z has to be 0 which is shown in the corresponding cell as S_1, 0. Likewise, in the cell corresponding to input = 11 and S_1 row, the entry is S_1, 1 as 1 + 1 + 1 produces carry 1 and sum also 1.

The information contained in the state table is now put in the form of a state graph, also called a state diagram, shown in Fig. 7.6.

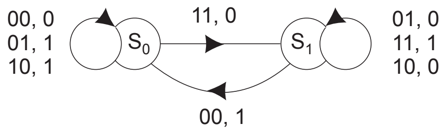


Fig. 7.6 State graph of serial adder

Notice that the machine in S_0 remains in S_0 itself for the inputs 00, 01, and 10 and the output Z is indicated after the comma along the self-loop. Likewise, the machine in state S_1 remains in S_1 itself for three different input combinations 01, 11, and 10 and with the output Z indicated after the comma along the self loop. Other directed arcs (edges) indicate state transitions and the corresponding outputs are indicated by the values of A B, Z as 11, 0 or 00, 1.

Step IV **Obtain the RSFST of the machine** The machine of Table 7.1 is already in this form. This is an extremely simple example chosen for illustration. Minimisation of state tables is discussed in detail elsewhere.

Step V **State assignment and transition table** As the machine has only two states, we need to have only one state variable y. The transition table is shown in Table 7.2.

Table 7.2 *Transition Table of Serial Adder*

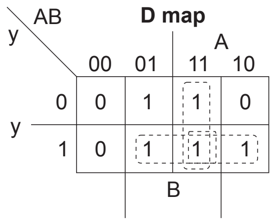
y(PS)	Y(NS), Z Inputs A B			
	00	01	11	10
$S_0 = 0$	0, 0	0, 1	1, 0	0, 1
$S_1 = 1$	0, 1	1, 0	1, 1	1, 0

Step VI **Choose a type of flip-flop and find the excitation table** Let us choose a D type flip-flop for this example. The excitation table is given in Table 7.3. Note that D = Y and the excitation table will be the same as the transition table, if you choose D type of flip-flops.

Table 7.3 *Excitation Table for Serial Adder*

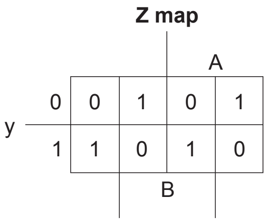
y(PS)	D, Z Inputs A B			
	00	01	11	10
0	0, 0	0, 1	1, 0	0, 1
1	0, 1	1, 0	1, 1	1, 0

Step VII **Synthesise the functions D and Z in terms of the inputs A, B and state variable y**



D map

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



Z map

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

$$D = AB + By + Ay \qquad Z = A'B'y + A'By' + ABy + AB'y'$$

Step VIII **Draw the circuit neatly** Notice that function D is identical with the output carry of full adder and Z is identical with that of the sum S of FA, while the state variable y takes the place of input carry C_i. The serial adder can be shown as indicated in Fig. 7.7.

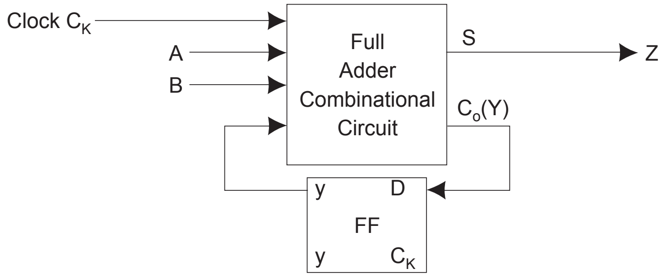


Fig. 7.7 Serial adder

Note that the memory element, the D flip-flop, is in the feedback loop, serving the purpose of merely delaying the carry signal by one clock period.

7.5 MODEL OF A FINITE STATE MACHINE (FSM)

After designing three finite state machines, it is expected that the reader developed a feel for what a synchronous sequential machine consists of. It is a **Finite State Machine (FSM)** also called **Finite Automaton** in the literature pertaining to **Automata Theory.** FSM comprises an input set (I), output set (Z), a set of states (S), state transition function (δ), and output function (λ). Thus, the finite state machine M is a quintuple given by M = (I, Z, S, δ, λ) where δ is a function of present inputs and present state resulting in the next state and λ is a function which enables us to compute the output depending on the present inputs and present state. The previous statement refers to what is generally called the **Mealy machines.** In the other type of machine, called the **Moore model,** each state is associated with a fixed output.

The structure of a synchronous sequential machine is shown in Fig. 7.8. Notice that the combinational circuit produces the next state variables as functions of the inputs and present state variables, and the flip-flops will act as memory. The clock pulses control all timing in the machine. If the clock is removed, the model represents an asynchronous sequential machine with mere delays replacing flip-flops, presented in a subsequent chapter.

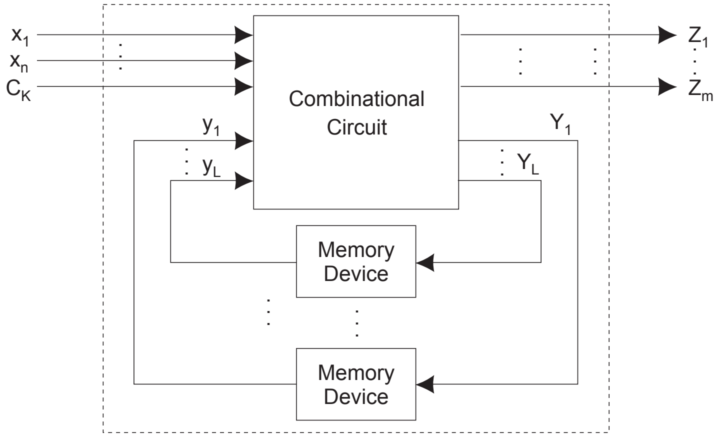


Fig. 7.8 Structure of a synchronous sequential machine

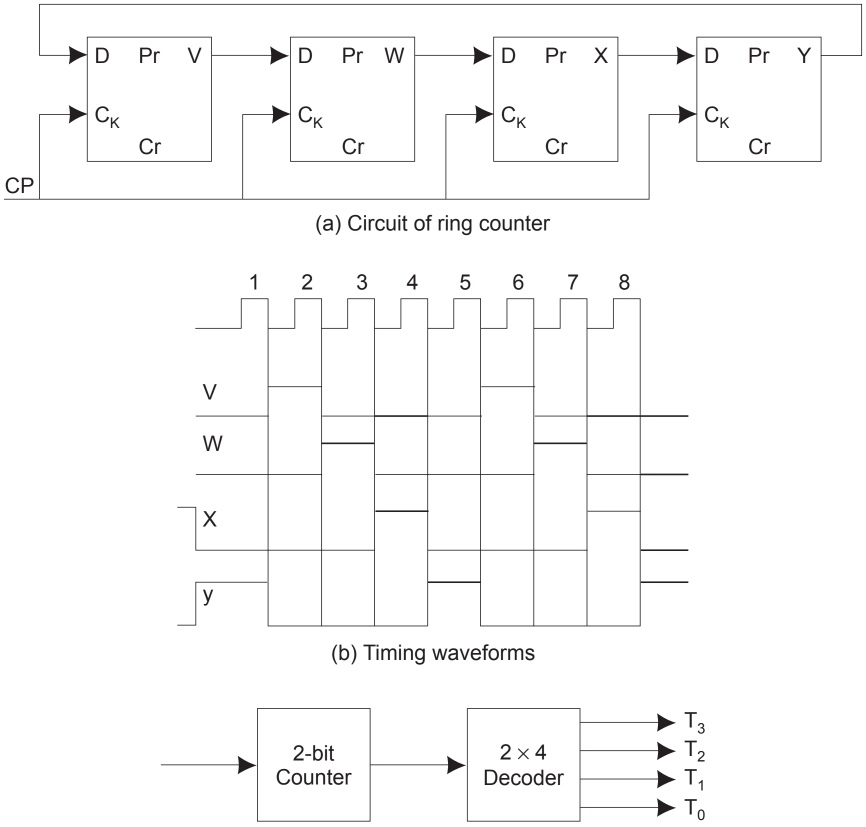
7.6 RING COUNTER AND SHIFT COUNTERS

A set of flip-flops form a register. Flip-flops connected in a cascade chain such that the output of one flip-flop is connected to the input of the next is called a shift register. A shift register in which the output of the last flip-flop is connected to the input of the first flip-flop is called a ring counter. D type of flip-flops are most suited to form a ring. A ring counter with its timing waveforms is shown Fig. 7.9. Initially, assume that one of the flip-flops, say V is preset to 1 and the rest cleared to 0. All the D flip-flops are driven by a clock. On the lagging edge of each clock pulse, the single 1 keeps circulating in the following pattern.

With K flip-flops in a ring, there will be K distinct states with the above configuration which works as a Mod K counter.

Ring counter is useful in generating timing waveforms used in control of digital systems such as the control unit of a digital computer. Take a look at the waveforms of V, W, X, Y in Fig. 7.9(b). Notice that the signals change state on the negative edge of the clock pulse. Only one of the signals is at 1 level. When V falls to 0, W rises to 1. When W falls to 0, X rises to 1 and so on in a cyclic order. Notice that the signal V is 1 during the 1st clock cycle, 5th, 9th and so on and signal W is at 1 level during the 2nd, 6th, 10th clock cycle and so on. These timing signals are used to enable sequential operations in digital systems.

Pulse	V	W	X	Y
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
1	1	0	0	0



(a) Circuit of ring counter

(b) Timing waveforms

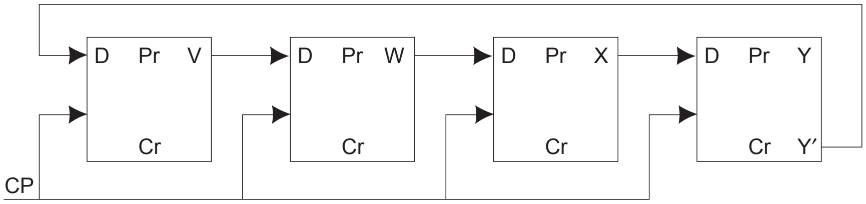
(c) Alternative circuit for generating timing signals

Fig. 7.9 Ring counter and timing waveforms

The above timing signals can also be generated by using a 2-bit counter and a 2-to-4 decoder shown in Fig. 7.9(c). The timing signals are called T_0, T_1, T_2, T_3. This subsystem is referred as a four-phase clock. Using the same principles, it is possible to have a multiphase clock. Shift counters, also called Johnson counters, discussed next, are also used for the purpose.

Shift Counter (Johnson Counter)

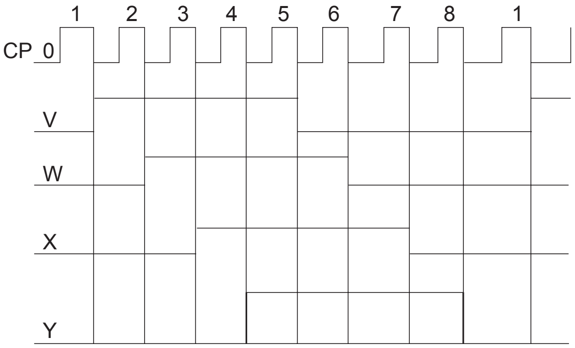
A slight modification of the ring counter is shown in Fig. 7.10(a). Notice in particular that y′ is connected to the excitation D of the flip-flop V. This is sometimes referred to as **"inverse feedback"** in contrast with "direct



(a) Circuit with D flip-flops

State after each CP	Y	X	W	V	Decimal code
0	0	0	0	0	0 initial
1	0	0	0	1	1
2	0	0	1	1	3
3	0	1	1	1	7
4	1	1	1	1	15
5	1	1	1	0	14
6	1	1	0	0	12
7	1	0	0	0	8
0	0	0	0	0	0

(b) Counting sequence



(c) Wave forms

Fig. 7.10 Shift (Johnson) Counter

feedback" used in a ring counter. This small modification results in special characteristics which are presented now.

If J-K flip-flops are used, Y′ is to be connected to J input of V and Y to K input of V. Initially all flip-flops are cleared to 0. Y′ = 1 excites the flip-flop V. On the occurrence of the first clock pulse, V becomes 1. On each occurrence of the clock pulse, Y′ moves to V, V goes to W, W goes to X, and X goes to Y. The counting sequence is listed in Fig. 7.10(b). On the negative edge of the 1st clock pulse, that is, at the beginning of the 2nd clock cycle V becomes 1. On the next pulse, W also becomes 1. The process keeps on going until all the flip-flops are filled with 1s. The next clock pulse puts a 0 in V and the subsequent pulses fill the counter with 0s and the process is repeated. The reader should observe that this has eight distinct states whose corresponding decimal codes are given in the table of Fig. 7.10(b). The output waveforms of the flip-flops V, W, X, Y are shown in Fig. 7.10(c). The period of each of these waveforms is equal to eight clock cycles and they are all square waves, shifted with respect to the previous one, by one clock period. This is commonly called **"shift counter"** or **"johnson counter".** Notice that n flip-flops can be used to divide the clock frequency by 2n. With n flip-flops connected as a shift counter, we get 2n distinct states. This concept is useful in building counters with even modulus. For instance, a Mod-6 counter with three flip-flops or a Mod-8 counter with four flip-flops and so on can be built using shift counters.

With four flip-flops, 16 distinct states are possible. What would happen to the unused states? The counting sequence of the Johnson counter with four flip-flops comprises eight valid states listed in Fig. 7.10(b) as 0, 1, 3, 7, 8, 12, 14, 15. Clearly the invalid states are 2, 4, 5, 6, 9, 10, 11, 13 listed in Fig. 7.11 together with the successor states attained at the occurrence of a clock pulse.

Remember y′ replaces V, V replaces W, W replaces X and X replaces Y on each clock pulse. Because of noise or some other malfunction, if the shift counter finds itself in an invalid state, it will continue to sequence itself among the invalid states and will not find a way to get into the correct sequencing of states (see the table of Fig 7.11).

Decimal code	Invalid state				Next state				Decimal code
	Y	X	W	V	Y	X	W	V	
2	0	0	1	0	0	1	0	1	5
4	0	1	0	0	1	0	0	1	9
5	0	1	0	1	1	0	1	1	11
6	0	1	1	0	1	1	0	1	13
9	1	0	0	1	0	0	1	0	2
10	1	0	1	0	0	1	0	0	4
11	1	0	1	1	0	1	1	0	6
13	1	1	0	1	1	0	1	0	10

Fig. 7.11 Invalid states in shift counter with four flip-flops

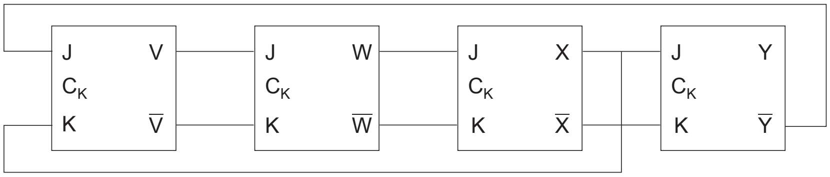
Even the cycle of invalid states divide the clock frequency by eight but decoding the states discussed next, to obtain the desired timing waveforms, will be different. Is there a way to force the counter into the intended correct sequencing? Yes. Notice that the first three flip-slops V, W, X assume the state X′ W V′ = 010 only twice in the invalid progression, corresponding to the decimal codes 2 and 10 (see Fig. 7.11). But this pattern is not at all contained in the valid desired progression of Fig. 7.10(b). Using this clue, our strategy is to use the function X′ W V′ to preset the flip-flops X and Y to 1 using the asynchronous preset lead of the J-K flip-flops. In that case Y X W V = 0010 (decimal 2) or 1010 (decimal 10) would be forced to become 1110 corresponding to 14 which is a member of the correct sequence of states. If this simple logic is embedded in the counter, the malfunction would get automatically corrected and the counter would be brought back onto the right track.

Thus far, we learnt how to construct even modulo counters with half the number of flip-flops. But how can odd modulo be constructed using shift counters? This requires a surprisingly simple modification which is illustrated now.

Consider the four flip-flop shift counter which has eight states. Suppose the first flip-flop V is cleared one clock cycle earlier. The other flip-flops W, X, Y will also be advanced by one clock period. Fig. 7.12(a) shows the modified sequence with seven states corresponding to the decimal codes 0, 1, 3, 7, 14, 12, 8. This would then behave as a Mod-7 counter. To advance clearing of flip-flop V by one clock period, we simply use J-K flip-flops directly, without converting them into D, and use the signal at X to excite the K input of V as shown in Fig. 7.12(b). In this case, at the end of 3rd clock pulse, J = 1, K = 1 for the flip-flop V. Hence V changes state on the 4th pulse itself instead of the 5th pulse in a Mod-8 counter.

Decimal code	Y	X	W	V	State after each clock pulse
0	0	0	0	0	Initial 0
1	0	0	0	1	After the 1st clock pulse V = Y′
3	0	0	1	1	2nd clock pulse
7	0	1	1	1	3rd clock pulse
14	1	1	1	0	4th clock pulse
12	1	1	0	0	5th clock pulse
8	1	0	0	0	6th clock pulse
0	0	0	0	0	Initial 0

(a) Counting sequence in Mod-7 shift counter



(b) Mod-7 shift counter circuit

Fig. 7.12 Mod-7 shift counter

The reader is advised to design, on similar lines, a Mod-6 shift counter with three flip-flops and a Mod-10 (decade) shift counter with five flip-flops. List the valid states and invalid states and find a way to force the counter onto the right track if for some reason it lands in an invalid state.

7.7 DECODING IN A SHIFT COUNTER AND CONTROL PULSES

Look at the states of the flip-flops V, W, X, Y and the corresponding waveforms shown in Fig. 7.10(c). Notice that the waveforms shift successively in time by one cock period in a cyclic order. Using this aspect, timing waveforms similar to that of Fig. 7.9(b) containing eight pulses may be produced by the following two-input AND gates. The 8-phase clock waveforms are shown in Fig. 7.13. Note the rhythmic order in the expressions for T_i. Notice specially that T_3 is the product of the 1st and last outputs of flip-flops and T_7 is the product of their complements.

$$T0 = V \bullet W', T1 = W \bullet X', T2 = X \bullet Y', T3 = Y \bullet V$$

$$T4 = V' \bullet W, T5 = W' \bullet X, T6 = X' \bullet Y, T7 = Y' \bullet V'$$

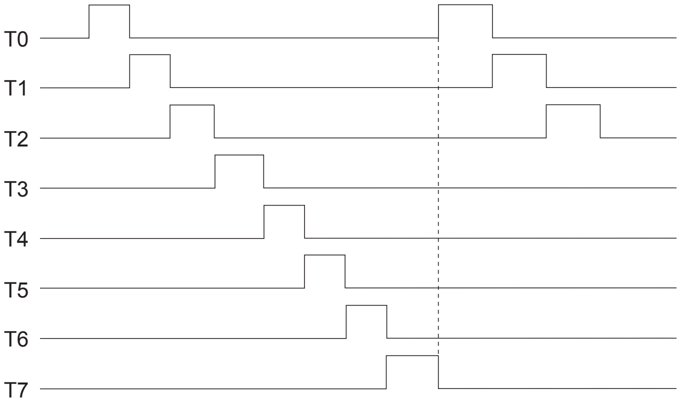


Fig. 7.13 Control pulses produced by a four flip-flop shift counter

To generate the control timing waveforms you have learnt three different techniques

1. Using a ring counter
2. Using a binary counter and decoder
3. Using a shift counter

If we wish to have an eight-phase clock, the first method uses eight flip-flops connected as a ring and one of them initially preset to 1. In the second method, we need a three flip-flop binary counter, eight AND gates each with three inputs for decoding. In the third method, four flip-flops are needed but decoding is simplified as the eight AND gates need to have only two inputs each. Depending on the situation, the designer can have a trade-off between elegance, cost, and complexity.

7.8 REGISTER OPERATIONS

There are some register operations which are extensively used in logic circuits in digital systems such as a digital computer. They are usually subsystems integrated with the overall system. The following are presented in this section.

1. Register with provision to shift right/shift left/parallel load.
2. Incrementer

3. Arithmetic operations add, subtract, increment, decrement
4. Word time generator

1. Shift Register Look at Fig. 7.14 which has three D flip-flops which form a 3-bit shift register. We use 4 × 1 MUXs to obtain four operations simultaneously. They are "no change", shift right, shift left, and parallel load. For each flip-flop, there is one MUX. The interconnections are highlighted below.

- The output of each MUX feeds the D input of the associated flip-flop through 0 input.
- S_1, S_0 are the select inputs connected to all the MUXs. S_1, S_2 = 00 for no change of outputs at flip-flops. Hence each Q is connected to 0 input of the associated MUX.
- S_1, S_0 = 01 provides right shift. Each Q is connected to 1 input of the next MUX to the right. Q_2 is connected to 1 of M_1, Q_1 is connected to 1 of M_0. Q_0 is the LSB. Note that the serial inputs feed the 1 of M_2.
- S_1 S_0 = 10 provides left shift. Each Q is connected to the 2 input of the MUX next to its left. Q_0 is connected to 2 of M_1 and Q_1 is connected to 2 of M_2. Q_2 is the MSB. Notice that the serial inputs feed the 2 of M_0. S_1 S_0 = 11 provides for loading the register in parallel using the external inputs I_2, I_1, I_0.
- How do you obtain a circular shift? Think it over.
- How do you transfer the content of one register to another register?

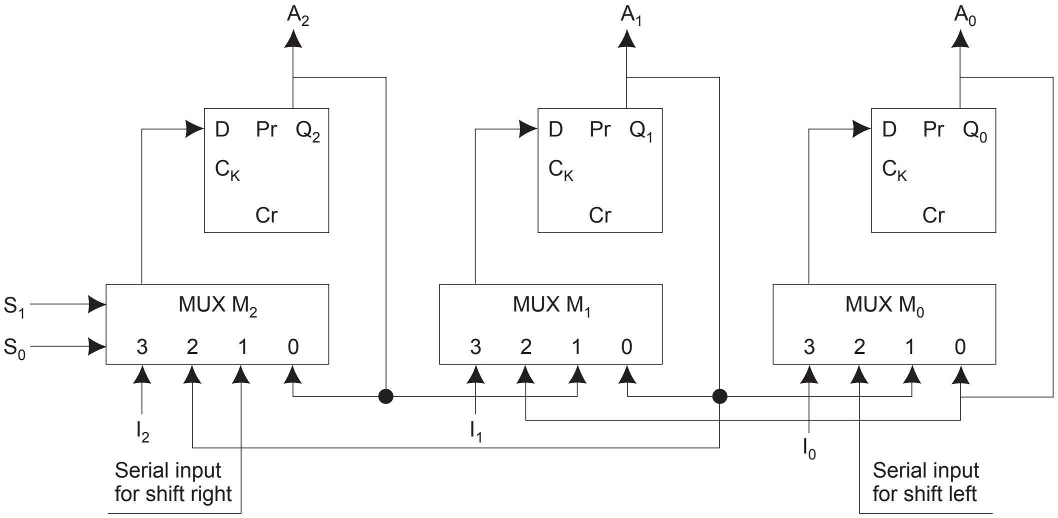


Fig. 7.14 Bidirectional shift register

2. Incrementer Incrementation of a register is required very often in digital systems. A conceptually simple circuit comprising half adders is shown in Fig. 7.15 for a word length of three bits. The binary word A_2, A_1, A_0 is fed to the circuit. Note that a 1 level feeds the 2nd input at the least significant place. The carry output of each Half Adder feeds the 2nd input of the next more significant stage. The incremented word is available as S_2 S_1 S_0 and the overflow carry is available at C_0.

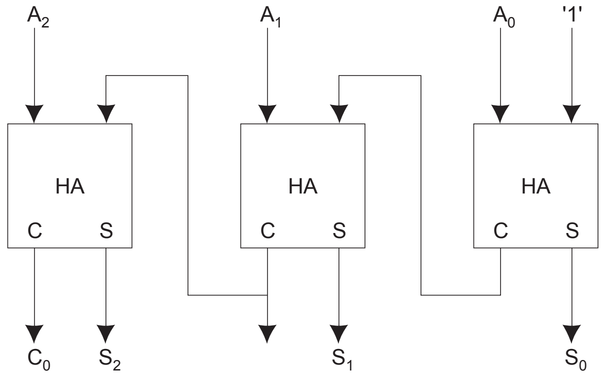


Fig. 7.15 Incrementer circuit

Select inputs	Operations performed and the Boolean expression
0 0	ADD: A + B + C_i
0 1	SUB: A + B′ + C_i represents 2's complement addition of B if C_i is fixed at 1. If C_i is at 0, then it becomes 1's complement addition.
1 0	INCREMENT: If C_i is fixed at 1. If C_i is 0, then transfer A to S.
1 1	DECREMENT by adding all 1s to A which is equivalent to subtracting 1 from A.

The reader should appreciate the elegance and capabilities of MUX learnt in an earlier chapter.

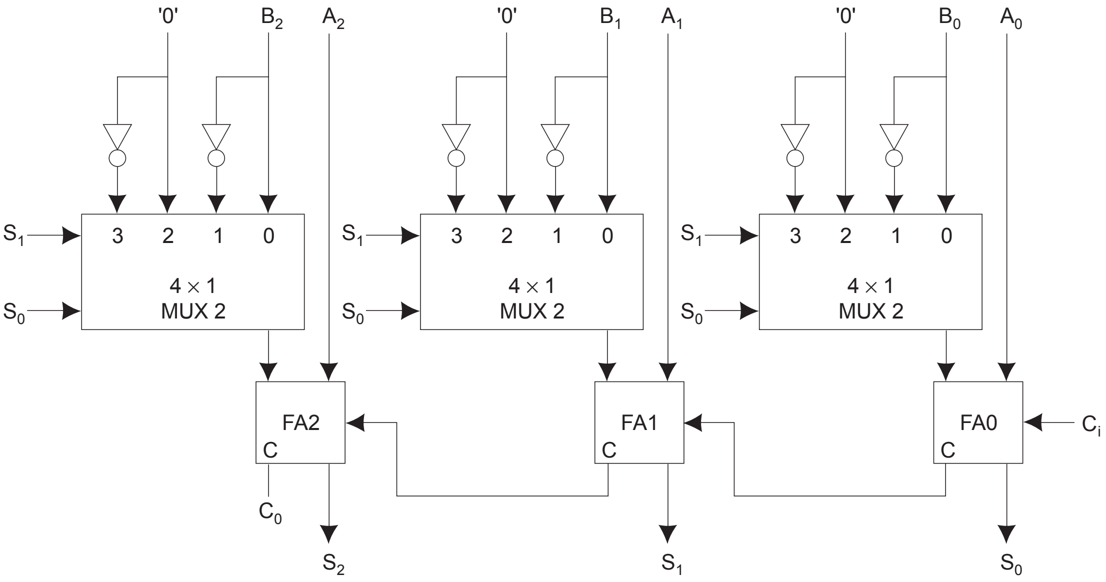
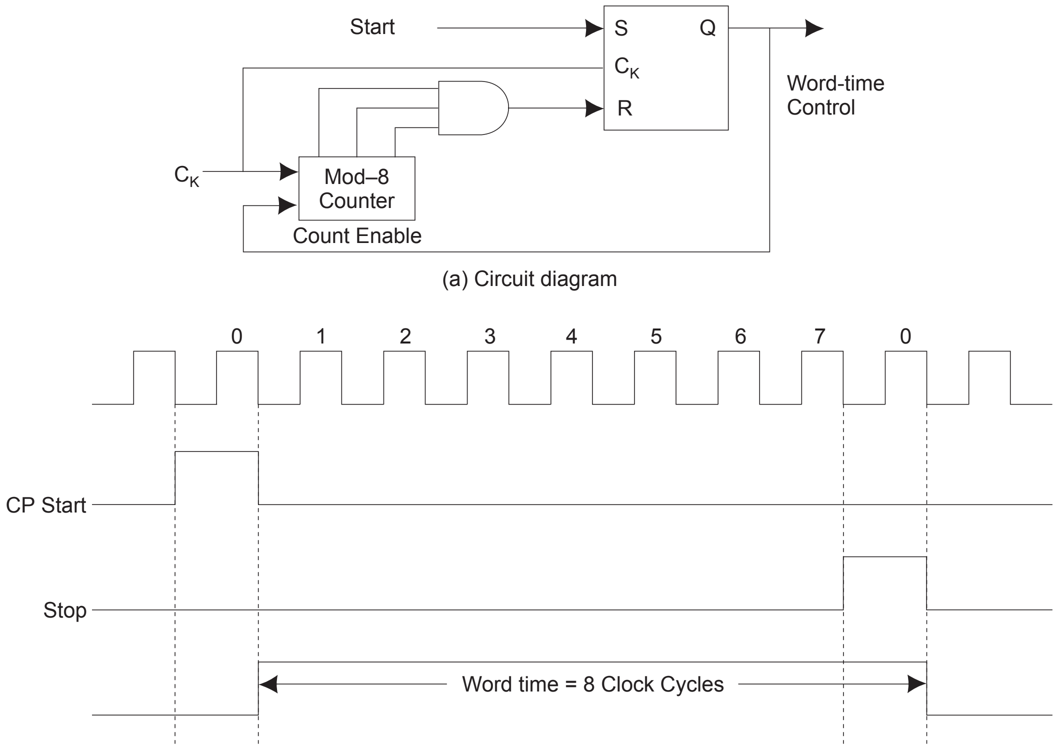


Fig. 7.16 A small arithmetic unit

3. Arithmetic operations A small arithmetic unit of word length three bits is shown in Fig. 7.16. It comprises three full adders and 4 × 1 MUXs associated with each full adder. The subsystem is capable of four operations depending on the select lines of MUXs S_1 S_0. Note that the binary number A is directly connected to the full adders. The second input B goes through a MUX via the 0 input of MUX. The 1 input of MUX gets B′, 2 input of MUX gets logic 0 and the 3 input of each MUX is fed with logic 1 level, as shown in Fig. 7.16. The operations performed are listed below.

4. Word time generator In serial transmission of binary words, it becomes necessary to mark the beginning and end of the word. For instance, the ASCII Code consists of eight bits for each character or digit. The starting bit and the last bit are to be marked by a timing pulse which is longer than the clock pulse. The pulse width of the timing pulse normally equals the period of the clock cycle. Timing pulses are obtained by decoding the states of the counter (see, for example, ring counter waveforms). The circuit and the relevant waveforms of word-time control are shown in Fig. 7.17. Notice that a start pulse sets the S-R flip-flop to 1 and the output Q enables a 3-bit counter by feeding the clock pulses through an AND gate to the counter. Exactly after eight pulses, counting from 0 through 7, the lagging edge of the 7th clock pulse resets the flip-flop to 0, producing a stop pulse with a width of one clock cycle. This pulse may be used to start another word-count control.



(a) Circuit diagram

(b) Waveforms

Fig. 7.17 Word-time generator

SUMMARY

Design of synchronous sequential machines and circuits covering counters, sequence detectors and adders have been presented through selected examples and various steps and their importance has been stated. An attempt is made to give a feel for the structure of a finite state machine. Use of ring counters and shift counters in control applications has been presented. Elegance of using multiplexers in realising multiple register operations has been brought to focus.

KEY WORDS

- ❖ Sequential machine
- ❖ State Variables
- ❖ Secondary input variables
- ❖ Transition table
- ❖ Standard form
- ❖ Finite state machine (FSM)
- ❖ Finite automaton (FA)
- ❖ Mealy machines
- ❖ Moore model
- ❖ Inverse feedback
- ❖ Shift Counterz
- ❖ Johnson Counter
- ❖ Ring counter
- ❖ Ripple

REVIEW QUESTIONS

1. In a T flip-flop, the ratio of the frequency of output pulses to the input pulse is

   a) ½ b) 1 c) 4 (d) 2

2. How many flip-flops are needed to divide the input frequency by 64?

   a) 4 b) 5 c) 6 d) 8

3. The number of flip-flops required in a decade counter is

   a) 3 b) 4 c) 5 d) 10

4. The number of flip-flops required in a modulo N counter is

   a) $\log_2(N) +1$ b) $[\log_2(N)]$ c) $[\log_2(N)] - 1$ d) $\log_2(N - 1)$

5. What is the maximum counting speed of a 4-bit counter which is composed of flip-flops with a propagation delay of 25 ns?

   a) 1 MHZ b) 10 MHZ c) 100 MHZ d) 4 MHZ

6. A shift register can be used for

   a) Parallel to serial conversion b) Serial to parallel conversion
   c) Digital delay line d) All of the above

7. The critical race hazard occurs in

   a) Combinational circuits only b) Asynchronous sequential circuits only
   c) Both combinational and sequential circuits d) Synchronous sequential circuits only

8. In sequential circuits, the present output depends on

   a) Past inputs only b) Present inputs only

c) Present as well as past inputs d) Past outputs

9. The following flip-flop is used as a latch.
   a) J-K flip-flop b) Master-slave J-K flip flop
   c) T flip-flop d) D flip-flop
10. D flip-flop is used as
   a) Differentiator b) Divider circuit c) Delay switch d) All of these
11. T flip-flop is used for
   a) Transfer b) Toggling c) Time delay d) Triggering
12. In T-flip flop the output frequency is
   a) Same as the input frequency b) One-half of its inputs frequency
   c) Double its input frequency d) Not related to input frequency
13. Shifting a binary data to the left by one bit position using shift left register, amounts to
   a) Addition of 2 b) Subtraction of 2 c) Multiplication by 2 d) Division by 2
14. If a counter is connected using six flip-flops, then the maximum number of states that the counter can count is
   a) 6 b) 256 c) 8 d) 64
15. The minimum number of flip-flops required for a Mod-10 ripple counter is
   a) 4 b) 2 c) 10 d) 5
16. The maximum number that can be obtained by a ripple counter using five flip-flops is
   a) 16 b) 32 c) 5 d) 31
17. A Mod-5 synchronous counter is designed using J-K flip-flops. The number of counts it will skip is
   a) 2 b) 3 c) 5 d) 0
18. The number of flip-flops required for a Mod-16 ring counter is
   a) 4 b) 8 c) 15 d) 16
19. The output frequency of a Mod-16 counter, clocked from a 10kHz clock input signal is
   a) 10 kHz b) 26 kHz c) 160 kHz d) 625 Hz
20. The type of register in which we have access only to the left most and right most flip-flops is
   a) Shift left and shift right registers b) Serial in serial out register
   c) Parallel in serial out register d) Serial in parallel out register
21. The maximum frequency to which a Mod-16 ripple counter using four J-K flip-flops with propagation delay time of 50ns is
   a) 4 MHz b) 5 MHz c) 3.2 MHz d) 320 MHz
22. The minimum number of flip-flops required to construct a Mod-64 (divide by 64) ripple counter is
   a) 4 b) 6 c) 16 d) 64
23. What is the basic difference between synchronous and asynchronous circuits?
24. List the steps in designing a synchronous sequential circuit.
25. Distinguish between state table and flow table.

26. What is RSFST?
27. Distinguish between primary inputs and secondary inputs
28. What is the importance of reduction of number of states?
29. What is the advantage of standard form for state tables?
30. Distinguish between internal variables and external variables?
31. What is the difference between internal state input state and total state?
32. What is the information contained in a transition table?
33. Distinguish between a transition table and excitation table?
34. What are the types of flip-flops used in synchronous sequential circuits?
35. When do we prefer T flip-flop?
36. When do we prefer S-R-flip-flop?
37. When do we prefer J-K-flip-flop?
38. When do we prefer D-flip-flop?
39. What is the characteristic feature of a latch?
40. What is the most commonly used flip-flop for counting?
41. Which flip-flop is preferred for control subsystems?
42. Define Mod-r count.
43. How many flip-flops are required for Mod-r count?
44. How do you highlight the change of state of a flip-flop from 0 to 1 or 1 to 0?
45. Indicate the excitations J and K required for 0, θ, 1, ↑.
46. What is the role of a flip-flop in a finite state machine?
47. What is the basic difference in the construction of a ring counter and a shift counter?
48. What is the number of distinct states in a ring counter versus a shift counter with the same number of flip-flops?
49. What is the characteristic feature in the timing waveforms of a ring counter which makes them directly suitable for use in control?
50. What are the various schemes used to produce control timing waveforms? What are the essential distinguishing features?
51. What are the valid states in a Mod-8 shift counter with four flip-flops?
52. What are the invalid states in a Mod-8 shift counter with four flip-flops?
53. How do you ensure a Mod-8 shift counter to keep its correct sequence?
54. What is the pattern common to all the invalid states of a Mod-8 shift counter?
55. How do you generate a signal to force the Mod-8 shift counter onto the correct track?
56. What is the principle used to convert an even modulo counter into an odd modulo counter?
57. What is the specific advantage in decoding the waveforms of a shift counter compared to a conventional counter?
58. Using a shift register, how do you obtain a circular shift?
59. How do you obtain transfer of data from one register to another?

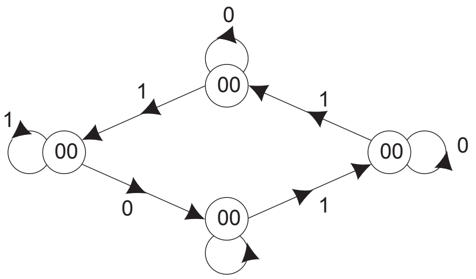
60. What kind of logic circuits are employed in an incrementer?
61. What type of logic modules are required to construct a decrementor?
62. Using 4x1 MUXs and full adders, suggest how to realise four ADD, SUB, INC, DEC operations?
63. What is the role of S-R flip-flop in the circuit of a word-time generator?
64. What is the purpose of a counter in word-time generator?
65. When do you use the terms sequential circuit and sequential machine?

PROBLEMS

1. Draw the circuit diagram of 4-bit **ring counter** using D flip-flops and explain its operation with the help of bit pattern.
2. Draw the circuit diagram of 4-bit **Johnson counter** using D flip-flops and explain its operation with the help of bit patterns.
3. Design a clocked sequential circuit to detect 1111 or 0000 and produce an output z = 1 at the end of sequence. Overlapping is allowed. Draw the circuit using D flip-flops.
4. a) Compare the merits and demerits of **ripple** and synchronous counters.
   b) Design a modulo-12 up synchronous counter using T flip-flops and draw the circuit diagram.
5. a) What is the maximum clock frequency required for a 10-bit ripple counter and synchronous counter, if the propagation delay time of flip-flop is 10ns?
   b) When are synchronous counters preferred?
   c) Draw the circuit diagram of a 4-bit **ring counter** using T flip-flops and draw the corresponding timing diagrams.
6. A sequential circuit has three D flip-flops, A, B, C and one input x. It is described by the following flip-flop input functions.

$$DA = (BC^1 + B^1 + C)X + (BC + B^1C^1)X^1,\ DB = A,\ DC = B$$

   a) Derive the state stable for the circuit.
   b) Draw two state diagrams one for x = 0, and the other for x = 1.
7. Design a sequential circuit with two D flip-flops A and B and one input x. When x = 0, the state of the circuit remains the same. When x = 1, the circuit goes through the state transitions from 00 to 01 to 11 to 10 back to 00 and repeats.
8. Design the sequential circuit specified by the state diagram using T flip-flops.



9. Design a sequential circuit with two J-K flip-flops A and B and two inputs E and W. If E = 0, the circuit remains in the same state regardless of the value of W. When E = 1 and W = 1, the circuit goes through the state transition from 00 to 01 to 11 to 10 and back to 00 and repeats. When E = 1 and W = 0, the circuit goes through the state transitions from 00 to 10 to 11 to 01 and back to 00 and repeats.
10. Design a circuit to check whether a 5-bit sequence satisfies an even parity.
11. Construct the state diagram and state table for a logic circuit that gives an output whenever the sum of the 1s in a repetitive 5-bit serial input sequence is 2.
12. a) Illustrate the various possible data entry methods of a shift register.
    b) Draw divide by 6 ripple counter using J-K flip-flops.
    c) Design a four-stage synchronous counter with parallel (look-ahead) carry.
13. When a certain serial binary communication channel is operating correctly, all blocks of 0s are of even length and all blocks of 1s of odd length. Show the state diagram or table of a machine which will produce an output pulse Z = 1, whenever a discrepancy from the above pattern is detected.

    Example

    X : 0 0 1 0 0 0 1 1 1 0 1 1 0 0

    Z : 0 0 0 0 0 0 1 0 0 0 1 0 1 0
14. A long sequence of pulses enters a two-input, two-output synchronous sequential circuit which is required to produce an output pulse z = 1 wherever the sequence 1111 occurs. Overlapping sequences are accepted; for example, if the input is 01011111 . . . , the required output is 00000011 . . .
    a) Draw a state diagram.
    b) Select assignment and show the excitation and output tables.
    c) Write down the excitation functions for S-R flip-flops, and draw the corresponding logic diagram.
15. Construct the state diagram for a two-input, eight-state machine which has to produce an output z = 1 whenever the last string of five inputs contains exactly three 1s and the string starts with two 1s. After each string which starts with two 1s, analysis of the next string will not start until the end of this string of five, whether it produces a 1 output or not. For example, if the input sequence is 11011010, the output sequence is 00000000, while an input sequence 10011010 produces an output sequence 00000001.
16. For each of the following cases, show the state table which describes a two-input, two-output machine having the following specifications.
    a) An output z = 1 is to be produced coincident with every occurrence of an input 1 following a string of two or three consecutive input 0s. At all other times the output is to be 0.
    b) Regardless of the inputs, the first two outputs are 0s. Thereafter the output z is a replica of the input x, but delayed two unit times, that is, $z(t) = x(t-2)$ for $t \geq 3$.
    c) $Z(t)$ is 1 if and only if $x(t) = x(t-2)$. At all other times Z is to be 0.
    d) Z is 1 whenever the last four inputs correspond to a BCD number which is a multiple of 3, that is, 0, 3, 6, 9.
17. Design a two-input, two-output synchronous sequential circuit which produces an output z = 1 whenever any of the following input sequences occur. 1100, 1010, or 1001. The circuit resets to its initial sate after a 1 output has been generated.

a) Form the state diagram or table. (Seven states are sufficient.)

b) Choose an assignment and show the excitation functions for J-K flip-flops.

18. Design a module-8 counter, which counts in the way specified below. Use flip-flops in your realisation.

Decimal	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

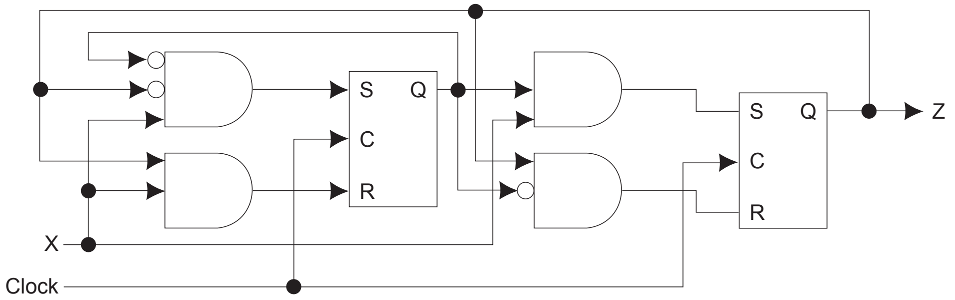
19. a) Convert a clocked S-R flip-flop into a J-K flip-flop using additional logic gates.

b) Convert a clocked J-K flip-flop into an S-R flip-flop using additional logic gates.

c) Convert a clocked J-K flip-flop into a D flip-flop and vice-versa.

d) Convert a clocked J-K flip-flop into a T flip-flop and vice-versa.

20. Analyse the sequential circuit shown below. Obtain first the flip-flop input equations and then the state table representation of the circuit. Does it have any unspecified entries?



21. Derive the state diagram and state table of the circuit featuring one input line, X_1, in addition to the clock input. The only output line must be 0 unless the input has been 1 for four consecutive clock pulses or 0 for four consecutive pulses. The output must be 1 at the time of the fourth consecutive identical input.

22. A sequential circuit is to be designed with two input lines X_1 and X_2 and a single output Z. If a clock pulse occurs when $X_2 X_1 = 00$, the circuit is to assume a reset state that may be labelled S_0. Suppose the four clock pulses following a resetting pulse coincide with the following sequence of input combinations— 00-10-11-01-11—the output Z is to become 1. The circuit cannot be reset to S_0 except by input 00. Define a special state S_1 to which the circuit will go once it becomes impossible for an output-producing sequence to occur. The circuit will thus wait in S_1 until it is reset. Determine a state diagram for the circuit.

23. Design a modulo-13 counter using T flip-flops.

24. Design a decade counter using 2-4-2-1 BCD code with J-K flip-flops.

25. Construct a state table for a synchronous sequential machine with one input x and one output z. The output z = 1 is to be produced coincident with every occurrence of an input 1 following a string of two or three consecutive input 0s. At all other times the output is to be 0. Realise the machine using J-K flip-flops.

26. a) Find the valid states of a Mod-8 shift counter with four flip-flops and give the counting sequence.
    b) Tabulate the invalid states.
    c) What is the strategy to correct an erring counter and force it onto the right track?

27. a) Build a Mod-6 shift counter and explain its working with a table of counting sequence.
    b) How do you convert this into a Mod-5 counter. Explain with the help of a circuit.

28. Discuss the following methods of producing control timing Waveforms. Illustrate with a 6-phase clock. Compare merits and demerits.
    a) Ring counter method
    b) Three-bit counter and decoder
    c) Shift counter method

29. Show a circuit using MUXs to obtain the following operations in a shift register.
    a) No change in the content
    b) Right shift
    c) Left shift
    d) Load the external inputs in parallel

8

Asynchronous Sequential Circuits

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Fundamental mode circuits and pulse mode circuits.
- Design steps.
- Pitfalls, remedies and essential hazards.
- Cycles, critical and non-critical races.
- Asynchronous pulse counters.

We encounter many situations when synchronising clock pulses are not available. For example, counting visitors to a library, counting cosmic particles or meteors impinging on earth's atmosphere are asynchronous phenomena. Furthermore, asynchronous circuits are generally faster than synchronous circuits as the speed of the latter is limited by the clock. At times, a large synchronous system may allow some of its subsystems to operate asynchronously in order to maintain the overall speed.

8.1 INTRODUCTORY CONCEPTS

Asynchronous sequential circuits are normally classified into **fundamental mode circuits** and **pulse mode circuits.** In **fundamental mode,** it is assumed that only one input can change at a time and all multiple input changes give rise to don't care entries. Further, it is assumed that the input changes occur only when the machine is in a stable state. (The concept of stable state and transitory states will be explained later.) The inputs and outputs assume logic levels in fundamental mode. In pulse mode, the inputs are pulses while the outputs may be pulses or levels. The model of an asynchronous sequential circuit is given in Fig. 8.1. Compare this with that of the synchronous machine of Fig. 7.8. There are essentially two differences. Firstly, there is no clock in asynchronous circuits. Secondly, the blocks marked 'Delay D' in the feedback path are essentially some kind of flip-flops, either J-K-M-S or T or D or others, in the case of synchronous circuits, while in asynchronous circuits they are mere path delays. Fundamental mode circuits will be presented first and pulse mode circuits later.

In synchronous sequential circuits, you have learnt that the present state (denoted by lower case y's) refers to the state of the machine before the clock pulse and the next state (denoted by capital Ys) is the state attained or reached by the machine after the clock pulse. Thus the clock pulses separate in time the present state and next state of synchronous sequential machines. The present state (indicated by row label) and the inputs (indicated by column label) together decide the output of the machine as well as the next state (row). A change of state involves change of row after the clock pulse.

In asynchronous sequential circuits, a slightly different orientation of thinking is required as there are no timing pulses. As usual, Ys are produced by a combinational circuit as functions of primary inputs and y's, which are called secondary inputs or secondary (internal) state variables. Ys are simply fed back to replace y's after a small delay in the path. For clarity in illustration, delays are shown in a feedback path. Actually, the normal path delay itself would serve the purpose of delays. After the delay, ys will assume the values of Ys. As a result of the closed loop, ys and Ys keep flowing (changing) until all $y_i \equiv Y_i$, which represents a **stable state.** It is assumed that any change of inputs occurs only after the ys and Ys settle down in a stable state.

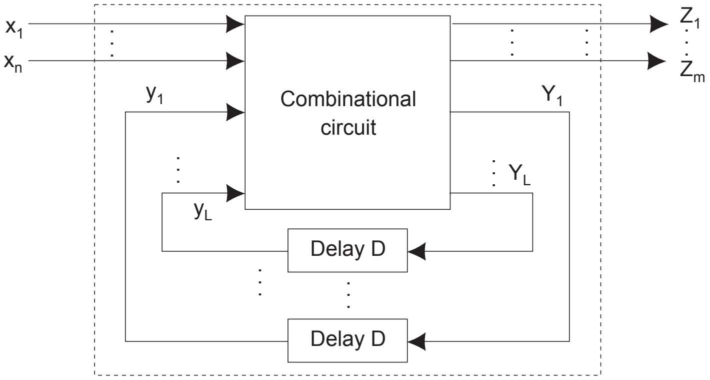


Fig. 8.1 Structure of an asynchronous sequential machine

In this case, we do not talk of a state diagram as in the case of synchronous sequential machines, but we start with a **flow table,** and as inputs change, the machine keeps flowing from one stable state to another stable state. There may be some transitory unstable states in this process.

8.2 FUNDAMENTAL MODE CIRCUITS—DESIGN STEPS

One goes through the following steps to design an asynchronous fundamental mode sequential machine.

Step I **Word statement** A problem's word statement has to be clearly understood. Important input sequences which cause changes in the output are listed for reference in the design process.

Step II **Primitive flow Table** Normally, we assume an initial stable state 1 of the machine with all inputs = 0. Then we make input changes following the sequences mentioned in the word statement, leading to changes in outputs. For every change in the inputs, introduce a new stable state by opening a new row and entering a circled number indicating a new stable state in the column corresponding to the changed inputs. Also indicate a transitory state in the same column of the previous row. We first follow the given sequences, which cause changes in outputs. Then we exhaust all possible transitions from the stable states without contradicting the word statement. This process may require introducing new stable states and hence new rows. As the input columns and the given sequences are finite, the process must terminate.

The following steps are better understood through the examples given next.

Step III Merger Diagram and minimum-row-merged flow table

Step IV Adjacency diagram and secondary state assignment, which does not cause **"Critical Races"**

Step V Transition (also called Excitation) table

Step VI Synthesise Ys and outputs as functions of inputs and ys. Draw a neat circuit.

8.3 DESIGN EXAMPLE 1

A certain asynchronous fundamental mode sequential circuit has two inputs x_1, and x_2 and one output Z. The output Z has to change from 0 to 1 only when x_2 changes from 0 to 1 while x_1 is already 1. The output Z has to change from 1 to 0 only when x_1 changes from 1 to 0 while x_2 is1.

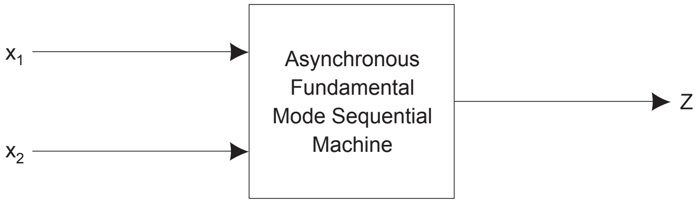
(a) Find a minimum row merged flow table.

(b) Find a valid secondary variable assignment.

(c) Make the output fast and flicker-free.

(d) Obtain hazard-free excitation and output functions.

Step I **Word statement** The problem's word statement has to be interpreted in unambignous terms. Important sequences, which cause changes in outputs, are listed for reference purpose.



x_1 = 0 1 1 0	**Important sequences**
x_2 = 0 0 1 1	x_1x_2 = 10 → 11 ==> Z = 0 → 1
Z = 0 0 1 0	x_1x_2 = 11 → 01 ==> Z = 1 → 0

Step II **Primitive flow table** For the benefit of the reader, various stages in the development of the primitive flow table are discussed below. We first assume an initial stable state with $x_1 = x_2 = 0$ indicated by a circled one–①–associated with output z = 0 marked adjacent to the circled number. By way of notation, all circled numbers indicate **stable states** and uncircled numbers indicate **transitory (unstable) states.** Let x_1 and x_2 be at 0 level in the initial state. Then we follow the input sequence x_1x_2 = 00 → 10 → 11 which changes Z from 0 → 1. We have to send the machine to a stable state ② on the change of inputs from 00 to 10. For this purpose, we enter an unstable 2 in 10 column and open a new row and have the stable state ②$_o$ in 10 column in the new row. We fill a dash in the 1st row of column 11 since the double-input change from 00 to 11 does not occur as assumed in the very beginning. Proceeding further with the input sequence, $x_1 x_2$ changes from 10 to 11 and, in accordance with the word statement, Z should change from 0 to 1. Hence we send the machine to another stable state via the transitory state 3 by opening a new row and have the stable state ③$_1$ under the input 11 column, as shown. We also indicate output Z = 1 by entering 1 by the side of the state.

Having become 1, Z goes to 0 only when the input sequence 11 → 01 occurs. Therefore, send the machine to another state ④$_o$ in a fresh row under column 01 via the transitory 4, as shown. With this, we have strictly followed the word statement. From every stable state, double-input changes are marked by dashes. See the partially filled primitive flow table shown in Fig. 8.2(a). For all other possible changes in inputs $x_1 x_{2,}$ we have to let the machine know what it should do.

Starting from ① if $x_1 x_2$ = 01 occurs, there is no need of changing the output Z which may continue to be at 0 level. We, therefore, send the machine to ④$_o$ via 4 marked in the 01 column. Now look at the 2nd row. From ②$_o$, if $x_1 x_2$ becomes 00, we may send the machine to ①$_o$ without contradicting the word statement and hence 1 is entered.

Now, look at the 3rd row. From ③$_1$, if $x_1 x_2$ becomes 10, what should the machine do? Z should continue to be at level 1and there is no stable state with Z = 1, in column 10. There is, thus, a need to introduce a new stable state with Z = 1 in column 10. Hence, open a new row and have the entry ⑤ as shown and direct the transition accordingly. From ④ if the inputs $x_1 x_2$ become 00, the machine has to be sent to a state with Z = 0 which is fulfilled by the stable state ①.

Following the same procedure, for the transition from ④ on $x_1 x_2$ = 11, we need to introduce a new state ⑥$_o$. Similarly, for the transition from ⑤$_1$ on $x_1 x_2$ = 00, a new state ⑦ has been introduced. Finally, from ⑦$_1$, on $x_1 x_2$ becoming 01, the machine has to be sent to a new state ⑧$_1$ with Z = 1 indicated in a new row. Notice that the procedure has to terminate as the machine has a finite number of inputs and outputs and the word statement has no ambiguities in the sense that it stipulates the conditions for all possible changes of output. The complete **primitive flow table** is shown in Fig. 8.2(b).

X_1X_2			
00	01	11	10
①$_o$		--	2
	--	3	②$_o$
--	4	③$_1$	
	④$_o$		--

(a) Partially filled Table

X_1X_2			
00	01	11	10
①$_o$	4	--	2
1	--	3	②$_o$
--	4	③$_1$	5
1	④$_o$	6	--
7	--	3	⑤$_1$
--	4	⑥$_o$	2
⑦$_1$	8	--	5
7	⑧$_1$	3	--

(b) Completed table

Fig. 8.2 Primitive flow table

Step III **Merger diagram** At this stage, we would like to explore the possibility of reducing the number of rows in the primitive flow table for the simple reason that we have to assign secondary state (also called internal) variables for each row. In each column, we introduced only two stable states, one with output 0 and the other with output 1. Clearly they cannot be equivalent. We now introduce another concept of **“merger of rows”**, which is analogous to the term **compatibility** used in minimising **incompletely specified sequential machines (ISSM’S)** discussed in a subsequent chapter. Consider, for instance, rows ① and ②. Using the unspecified entries and noting that the transitory unstable entries do not conflict, we might **“merge”** these two rows into one row shown below.

X_1X_2			
00	01	11	10
①o	4	--	2
1	--	3	②o

Merges into

X_1X_2			
00	01	11	10
①o	4	3	②o

In such a situation, stable states ① and ② will have the same internal secondary state assignment. Notice that the secondary states are identified by the rows, whereas the stable states require specification of both the row and input column (also called **input state**). Hence, the need to have the concept of **“total state”** in asynchronous machines. In synchronous machines, a change of state involves a change of row. There is no question of unstable transitory states in synchronous machines as it is the clock pulse which controls the transition. In the absence of the clock pulse, the circuit is always stable and even the inputs are not available to the machine. In asynchronous machines, a change in the input columns (horizontal movement) may result in a change of stable state with the internal state (row) unaltered. Both the row and column together identify the total state in asynchronous machines.

Notice in the primitive flow table of Fig. 8.2(b) that there is one row for each stable state. The rows 1 and 3 cannot merge as the entries in column 10 are different which direct the machine to go to ② in one case and ⑤ in the other case. Now, consider rows 1 and 4 which merge into one row, as shown below.

X_1X_2			
①o	01	11	10
	④o	--	2
1		6	--

Merges into

X_1X_2			
00	01	11	10
①o	④o	6	2

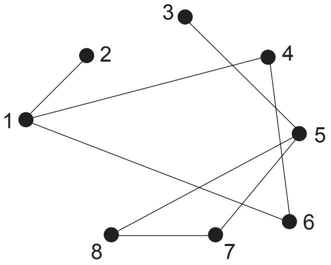
It is now clear that the following rules hold in effecting a merger of rows.

1. A stable or transitory (unstable) state and a dash (unspecified) merge into the corresponding state.
2. A stable state and a transitory state merge into the stable state.

Following the above rules, the rows are denoted by nodes and the possible mergers are indicated by edges connecting the nodes. See Fig. 8.3(a). Notice that we have to choose complete polygons for effecting mergers. A triangle formed by the nodes 1, 4, 6 is a complete polygon as every node is connected to every other node in the set. Likewise, nodes, 5, 7, 8 form another triangle. Hence the chosen mergers are given below. A rectangle (not encountered in this example) with both the diagonals is a complete polygon.

Mergers : (1, 4, 6) (2) (3) (5, 7, 8)

Mergers (1, 2), (3, 5) are not needed but rows 2 and 3 have to be left as singleton stable states in order to cover all the stable states of the machine. Thus, we get a 4-row flow table shown in Fig. 8.3(b). Notice further that the outputs are indicated only for the stable states. Outputs of unstable entries will be considered later. The rows in the minimum row-merged flow table are labelled a, b, c and d for further discussion next.



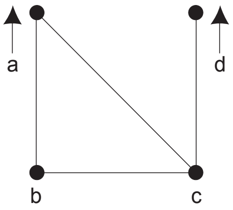
Mergers: (1, 4, 6) (2) (3) (5, 7, 8)

(a) Merger diagram

Rows	X_1X_2			
	00	01	11	10
a	$\textcircled{1}_o$	$\textcircled{4}_o$	$\textcircled{6}_o$	2
b	1	--	3	$\textcircled{2}_o$
c	--	4	$\textcircled{3}_o$	5
d	$\textcircled{7}_1$	$\textcircled{8}_1$	3	$\textcircled{5}_1$

(b) Merged flow table

Fig. 8.3 Merger diagram and minimum row-mergerd flow table



(a) Adjacency diagram

$PS(y_1y_2)$	X_1X_2 NS(Y_1Y_2), Z			
	00	01	11	10
a = 00	$\textcircled{00}$, 0	$\textcircled{00}$, 0	$\textcircled{00}$, 0	01
b = 01	00	--	11	$\textcircled{01}$, 0
c = 11	--	00	$\textcircled{11}$, 0	10
d = 10	$\textcircled{10}$, 1	$\textcircled{10}$, 1	11	$\textcircled{10}$, 1

(b) Invalid assignment

$PS(y_1y_2)$	X_1X_2 NS(Y_1Y_2), Z			
	00	01	11	10
a = 00	$\textcircled{00}$, 0	$\textcircled{00}$, 0	$\textcircled{00}$, 0	01, 0^f
b = 01	00, 0^f	00, 0^f	11, 0^f	$\textcircled{01}$, 0
c = 11	--	01, 0^f	$\textcircled{11}$, 0	10, 1^f
d = 10	$\textcircled{10}$, 1	$\textcircled{10}$, 1	11, 0^f	$\textcircled{10}$, 1

(c) Valid assignment and transition table

The superscript 'f' in Fig. 8.4 (c) indicates assignment of output same as that of the corresponding stable state.

Fig. 8.4 Secondary state variable assignment

Step IV **Adjacency diagram and secondary variable assignment**

Consider each transition from one stable state to another stable state through the unstable states. It is desirable that it involves change of a single secondary variable; Otherwise, we may encounter “races”, discussed next. Hence the need to examine adjacencies of rows indicated by an edge connecting the rows—see the adjacency diagram shown in Fig. 8.4(a). Row c has to be adjacent to three rows—a, b, d. This is not possible using two secondary variables (also called internal state variables). A node designated or assigned by n variables can have at the most n adjacent nodes. The unstable entry 4 in Fig. 8.3(b) required adjacency between rows a and c. If this adjacency condition could be relaxed, we would be able to meet the remaining adjacencies with two secondary (internal) state variables only. Otherwise, we may need three secondary variables, in which case there will be eight rows and many unused (optional or don’t care) entries. The importance of adjacency assignment would become clear when we discuss **“races”** next. The adjacency aspect will be discussed again after learning about races.

Suppose we assign two secondary state variables $y_1 y_2$ arbitrarily to the rows of 8.4(b). Remember that the stable states are identified by $y_i = Y_{i.}$ Therefore, in the cells representing stable states, we have to fill the same entries for $Y_1 Y_2$ as $y_1 y_2$ of that row. For the transitory states, we have to fill the assignment of the corresponding stable state to which the machine has to be directed. Now, observe the entry in row 11, column 01. The next state entry 00 therein causes a **race.** The machine is directed to change its secondary state variable $y_1 y_2$ from 11 to 00, involving a double change which is referred to as a **race.** If we are lucky, y_1 will be faster than y_2. Then $y_1 y_2$ becomes 01 first where there is a dash which is preferably filled by 00 in order to direct the machine to the 00 row. Even otherwise, a moment later, y_2 also would change and the state would become 00, which is the correct state intended by design. However, if y_1 is sluggish and y_2 is faster, then $y_1 y_2$ momentarily becomes 10, changing the row in the same input column 01. Clearly, the machine lands in a wrong stable state ⑩ and gets stuck there. Such a hazardous situation is called a **critical race.** The assignment involving a **critical race** shown in Fig. 8.4(b) is not valid. Note that the **critical race** is caused because of two stable states in the same input column 01 and that the assignment for a transitory state involved a double change of secondary variables.

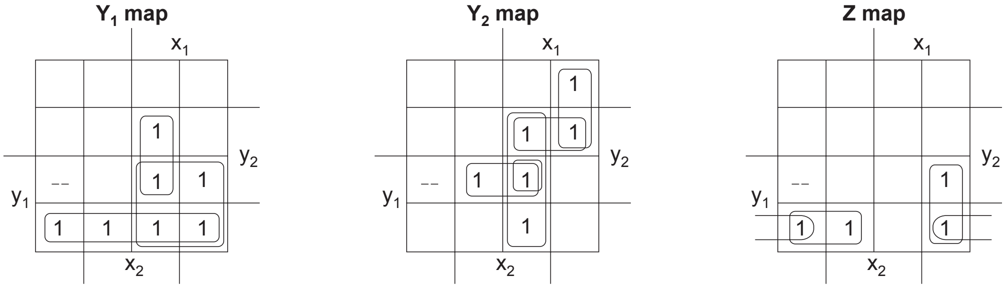
Observe that by making use of the dash in column 01, it is possible to avoid the critical race by directing the transition [see arrows marked in Fig. 8.4(c)] by assigning $Y_1 Y_2$ as shown. This is called a **cycle.** This does not involve double changes in secondary variables but needs one or more extra transitions and hence the corresponding delay in settling in a stable state. The unspecified entry came in handy to avoid the **critical race,** resulting in a valid assignment. Had there been only one stable state in the input column 01, it would not be a **critical race.** In that case, it would be a **non-critical race,** which is allowed. Note that each row represents only one internal (secondary) state but several stable states of the machine.

Having succeeded in obtaining a valid secondary state assignment, we now proceed to assign the outputs for the transitory states. If the output is to be fast, clearly, the output of the transitory state should be equal to the corresponding stable state. If the output has to be flicker-free, we have to examine all possible transitions between every pair of stable states with the same output and ensure that the intervening transitory states also will have the same output. Such a situation is not present in this example.

Step V **Transition table** The valid **flow table** is given below. Under the inputs $x_1 x_2$, the first column indicates Y_1, the second column denotes Y_2, and the entry after comma gives the output Z.

(y_1y_2)	(Y_1Y_2), Z X_1X_2			
	00	01	11	10
00	00, 0	00, 0	00, 0	01, 0
01	00, 0	00, 0	11, 0	01, 0
11	--	01, 0	11, 0	10, 1
10	10, 1	10, 1	11, 0	10, 1

Step VI **Synthesis** Map the functions and obtain a static hazard-free realisation of Y_1, Y_2 and Z. Draw a neat circuit.



$Y_1 = x_1y_1 + y_1y_2' + x_1x_2y_2$

$Y_2 = x_1x_2y_1 + x_1x_2y_2 + x_1x_2'y_1' + x_2y_1y_2 + xy_1'y_2$

$Z = x_1'y_1y_2' + x_1x_2'y_1 + x_2'y_2y_2'$

Fig. 8.5 Maps for excitations Y1, Y2 and output Z

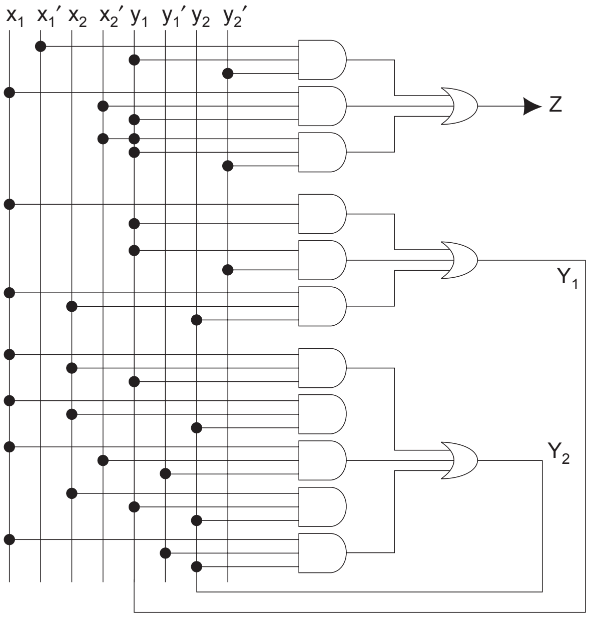


Fig. 8.6 Realisation of the asynchronous sequential circuit

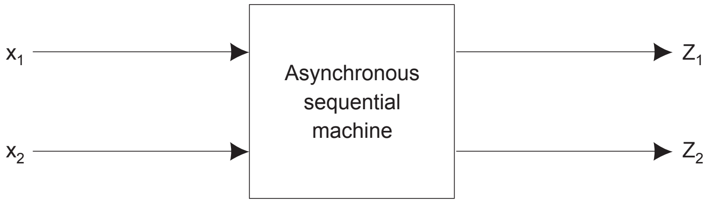
8.4 DESIGN EXAMPLE 2

In order to consolidate the concepts learnt, another example is presented in this section with all the various stages in the design.

A certain fundamental mode asynchronous sequential machine has two inputs x_1, x_2 and two outputs Z_1, Z_2. The output Z_i for i = 1 or 2 has to assume logic 1 level whenever the input that changed last was x_i and 0 at all other times.

1. Find a minimum row-reduced flow table.
2. Select a valid assignment and synthesise the circuit.
3. Does the machine have a power-on state with all $x_i = 0$ and all $Z_i = 0$?

Step I **Word statement**



This machine does not have a power-on state specifically defined.

$x_1x_2 = 00$ can be reached either from 01 or from 10. Hence, both the outputs being 00 will never occur except in the power-on state, which is only transitory. In steady state, the machine should have two stable states under each input column, one with $Z_1Z_2 = 01$ and the other with $Z_1Z_2 = 10$. If the power-on state is provided with $x_1x_2 = 00$, $Z_1Z_2 = 00$, the machine would never reach that state after leaving it.

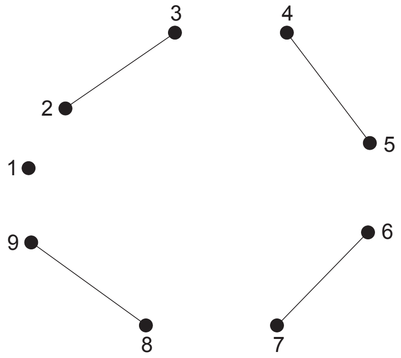
Step II **Primitive flow table**

X_1X_2			
00	01	11	10
①$_{00}$	2	--	4
3	②$_{01}$	6	--
③$_{01}$	2	--	4
5	--	9	④$_{10}$
⑤$_{10}$	2	--	4
--	7	⑥$_{10}$	8
3	⑦$_{10}$	6	--
5	--	9	⑧$_{01}$
--	7	⑨$_{01}$	8

Fig. 8.7 Primitive flow table

Using only single input changes, 00 can be reached from 01 or 10. Let us first exhaust these possibilities by introducing the stable states ①, ②, ③, ④, ⑤. Now, follow the transition starting from ② with input 01. Stable states ⑥, ⑦ exhaust the possibility of reaching inputs 01 from 11. Stable state ⑧ exhausts the transition to 10 and ⑨ covers the last possibility of reaching 11 from 10. All double-input changes in each row are marked by dashes and the remaining cells are filled with some transitory or stable states.

Step III **Merger diagram and merged flow table**



Notice that ① is merely a power-on state. Also note that the primitive flow table does not contain any transitory state labelled 1. Once the machine leaves stable state ①, it will never return to that total state. Hence, we consider this as a transient behaviour and ignore it in forming the minimum row-merged flow table.

Mergers: (2, 3), (4, 5), (6, 7), (8, 9)

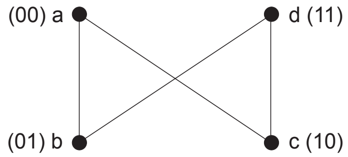
There will be four secondary internal states denoted by the rows a, b, c, d. The machine requires two binary state variables $y_{1,}\, y_2$.

Reduced (Merged) Flow Table

	X_1X_2			
Rows	00	01	11	10
a	③$_{01}$	②$_{01}$	6	4
b	⑤$_{10}$	2	9	④$_{10}$
c	3	⑦$_{10}$	⑥$_{10}$	8
d	5	7	⑨$_{01}$	⑧$_{01}$

The row labels a, b, c, d facilitate examining adjacencies next. Outputs Z_1, Z_2 are marked for the stable states.

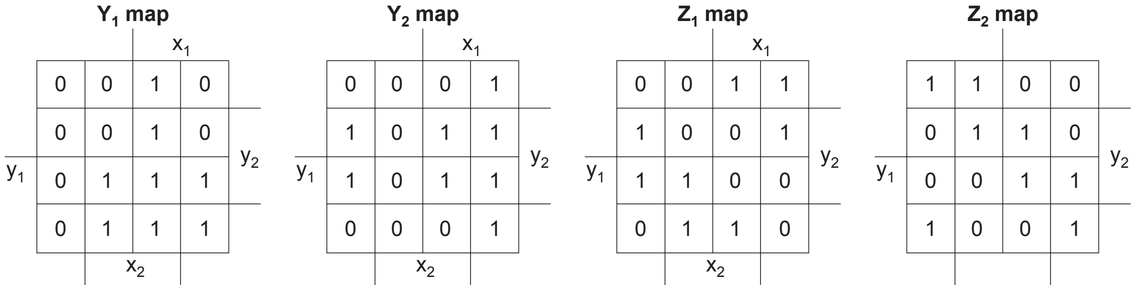
Step IV **Adjacency diagram and assignment** Noting the transitory states in each row of the merged flow table, adjacency requirements are indicated by connecting nodes by edges as shown below. Notice that it is possible to comply with all the adjacency requirements using only two secondary variables y_1, y_2. Adjacency assignment are marked on the vertices denoting rows.



Step V **Transition table with a valid assignment** We rewrite the flow table with the assigned values. As each transition requires only a single change in secondary variables, there will be no **races** unlike the previous example. Remember that the cells containing stable states are identified by $Y_i = y_i$, which are circled. Where there are transitory states, the row assigned to the corresponding stable state has to be entered. Special care should be taken in making the entries. Finally, the outputs are entered after a separating comma.

y_1y_2	Y_1Y_2, Z_1Z_2 X_1X_2			
	00	01	11	10
a = 00	00, 01	00, 01	10, 10	01, 10
b = 01	01, 10	00, 01	11, 01	01, 10
d = 11	01, 10	10, 10	11, 01	11, 01
c = 10	00, 01	10, 10	10, 10	11, 01

Step VI **Synthesis of the circuit** Notice that the columns of the transition table (assigned flow table) correspond to Y_1, Y_2, Z_1, Z_2 as functions of x_1, x_2, y_1, y_2. The maps are given below.



Y_1 map

0	0	1	0
0	0	1	0
0	1	1	1
0	1	1	1

Y_2 map

0	0	0	1
1	0	1	1
1	0	1	1
0	0	0	1

Z_1 map

0	0	1	1
1	0	0	1
1	1	0	0
0	1	1	0

Z_2 map

1	1	0	0
0	1	1	0
0	0	1	1
1	0	0	1

$Y_1 = x_1x_2 + x_1y_1 + x_2y_1$

$Y_2 = x_1x_2' + x_1y_2 + x_2'y_2$

$Z_1 = x_1y_1'y_2' + x_2'y_1'y_2' + x_1'y_1y_2 + x_2y_1y_2 +$
terms to be added for
hazard-free output
$x_1x_2'y_1' + x_1'x_2'y_2 + x_1'x_2y_1 + x_1x_2y_2$

$Z_2 = Z_1'$

Notice that all the terms are selective prime implicants. Drawing the logic circuit is left to the reader.

8.5 RACES, CYCLES AND VALID ASSIGNMENTS

Wherever a change of more than one secondary variable is required for the machine to reach a stable state, a race exists between the secondary variables. Consider the flow table of Fig 8.8.

y_1y_2	Y_1Y_2 x_1x_2			
	00	01	11	10
00	00	01	01 ?	11
01	10 ? 11 √	01	00 ?	01
11	10	01	11	11
10	10	01	11	01

Fig. 8.8 Illustration of races and cycles

Look at the entry in the row 01 column 00. It is an unstable 10 which means that both the variables $y_1 y_2$ have to change simultaneously from 01 to 10 to reach the stable state ⑩. It is most unlikely that two variables will change exactly at the same time. This is called a **race.** If y_2 is a little faster than y_1, then $y_1 y_2$ follows the transition 01 → 00 and then to 10 but on $y_1 y_2$ becoming 00, the machine would reach a stable state and does not move out until inputs change. Thus, there is a possibility of the machine landing in a wrong state; such a situation is called a **critical race.** If y_1 is faster than y_2, the transition would be 01 → 11 and then to 10. The intermediary entry is an unstable 10, which directs the machine to reach the desired state and there is no hazardous behaviour. In order to avoid the critical race, we change the entry in row 01 column 00, from 10 to 11 which directs the transition indicated by arrows. This is called a **"Cycle".** It is common to introduce **cycles** to avoid **critical races.**

Now look at the column 01 which has only one stable state in row 01. Notice that the 01 unstable entry in row 10 causes a **race** between y_1, y_2 but whichever is faster than the other, the machine has to finally land in the correct state and hence it is referred to as a **"non-critical race".** Such non-critical races are usually permitted. It is safe to direct the transition by proper assignment of the intervening cells instead of filling them with dashes.

Now focus on the entries in column 11. The two unstable entries 01 and 00 result in a never-ending cycle. The entry in row 00 directs the machine to go to row 01 and vice versa. Care must be taken that such cycles must not be contained in any secondary variable assignment.

Finally look at the column 10, which contains two stable states in rows 01 and 11. The unstable entries in that column cause critical races which cannot be avoided unless the assignment is changed or augmented with an extra secondary variable. With three secondary variables, it is possible to assign two combinations to each of the four rows of the flow table so that any transition involving any two rows can be directed using a cycle of adjacent rows. Fig. 8.9 shows such assignments. Take a look at Fig. 8.9(a). Every row is adjacent, in one of its assigned combinations, to every other row. This is equivalent to having two equivalent secondary states for every row. As such, it is possible to have **cycles** directing the transitions and avoid **races,** more particularly **critical races.** It has been shown in literature that at most $(2n - 1)$ secondary variables will be required to assign 2^n rows. Fig. 8.9(b) shows an assignment for 8-row tables with five secondary variables.

$y_3 \backslash y_1y_2$	00	01	11	10
0	a	c	c	a
1	b	b	d	d

(a) Assignment in 4-row flow tables

$y_3y_4y_5 \backslash y_1y_2$	00	01	11	10
000	a	b	c	c
001	b	b	d	d
011	a	a	c	d
010	b	b	c	d
110	e	f	e	e
111	e	f	f	f
101	e	g	g	h
100	h	h	g	h

(b) Assignment in 8-row flow tables

Fig. 8.9 Race-free assignments

8.6 ESSENTIAL HAZARDS IN ASYNCHRONOUS CIRCUITS

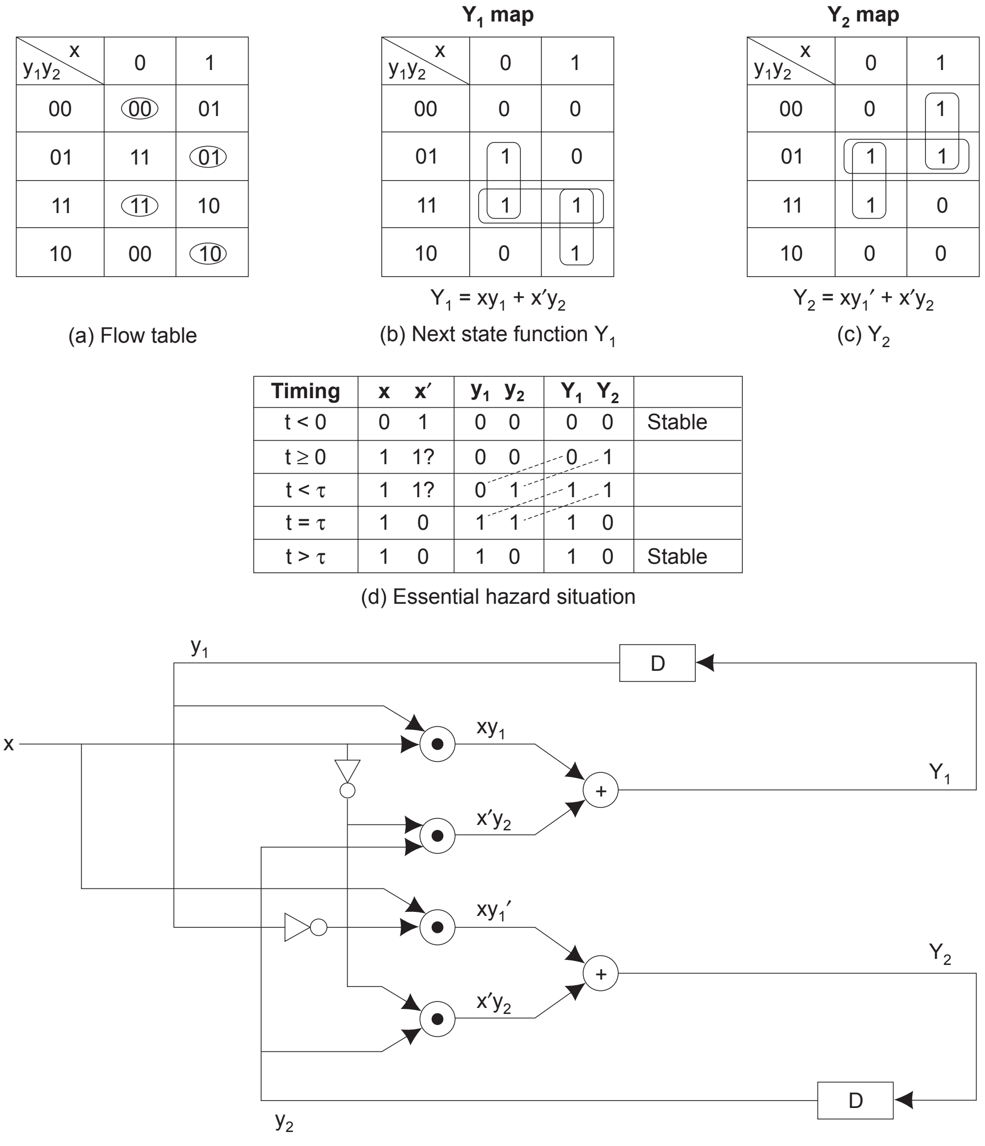
A type of hazard called **essential hazard** occurs in asynchronous sequential circuits when a change in input variable (columns) is slower than the consequential changes in secondary state variables (rows). Consider a typical flow table given in Fig. 8.10(a). Initially, let the input x be 0 and the secondary state variables $y_1 y_2 = 00$. Clearly the machine is in stable state $\textcircled{00}$. Let the input x change from $0 \to 1$. Notice that x′ appears in the transition functions Y_1, Y_2. Due to some delay in inversion or anywhere in the path, it is possible that both the nodes denoting x and x′ may be for a short time, at 1 level, a situation similar to **static hazards** in combinational circuits. Such **static hazards** caused spurious spikes called **glitches** in combinational circuits, which are easily avoided by including some redundant terms [not shown in the expressions of Fig. 8.10(b) and (c)].

In asynchronous sequential machines, such a situation might cause the machine to land in a ‘total state’, different from the intended stable state. Let us see how.

Look at Fig. 8.10(d) wherein the inputs x, x′, secondary state variables y_1, y_2 and the excitations Y_1, Y_2 (same as next state variables) are tabulated. Focus on the situation that x is changing from $0 \to 1$ at $t = 0$. Look at the hazardous situation obtaining for a short moment between 0 and τ. Input x has changed to 1 but x′, after the inverter, did not change yet. The nodes x and x′ are to be considered as distinct variables. Both x and x′ are 1 momentarily. Look at the expressions for Y_1 and Y_2 in (b) and (c). For this hazardous moment $Y_1 Y_2$ will be 01. After a little while, y_1 y_2 will also become 01. See the dotted line indicating the flow.

For $t < \tau$, the new values of $Y_1 Y_2$ become 11 because x′ is still at 1. It follows that $y_1 y_2$ becomes 11 as shown by the dots again. In the meantime, x′ assumes its correct value 0. Notice that the state variables have changed twice already from 00 to 01 and then to 11.

Look at the row for $t = \tau$. Compute $Y_1 Y_2$ which would become 10, which would flow to the next row as $y_1y_2 = 10$. Now look at the row for $t > \tau$ and satisfy yourself that $Y_i \equiv y_i \Rightarrow$ stable state. Now trace the transition from the initial stable state to the next stable state.



y_1y_2 \ x	0	1
00	00	01
01	11	01
11	11	10
10	00	10

(a) Flow table

Y_1 map

y_1y_2 \ x	0	1
00	0	0
01	1	0
11	1	1
10	0	1

$Y_1 = xy_1 + x'y_2$

(b) Next state function Y_1

Y_2 map

y_1y_2 \ x	0	1
00	0	1
01	1	1
11	1	0
10	0	0

$Y_2 = xy_1' + x'y_2$

(c) Y_2

Timing	x	x′	y_1	y_2	Y_1	Y_2	
$t < 0$	0	1	0	0	0	0	Stable
$t \geq 0$	1	1?	0	0	0	1	
$t < \tau$	1	1?	0	1	1	1	
$t = \tau$	1	0	1	1	1	0	
$t > \tau$	1	0	1	0	1	0	Stable

(d) Essential hazard situation

(e) Circuit realisation of the flow table

Fig. 8.10 Illustration of essential hazard

$$Y_1Y_2 = (00) \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow (10)$$

Notice that the intended stable state was (01) but the machine landed in a different state (10) after going through three transitory (unstable) secondary states. This is referred to as **essential hazard.**

In the literature on this subject, it has been proved that wherever a single change and three consecutive changes of an input lead the machine to different stable states, there will be an essential hazard. A typical flow table containing essential hazard is given below.

x = 0	x = 1
①	2
3	②
③	4
1	④

For the sake of completeness, notice that the above table corresponds to the flow chart of Fig. 8.10(a) which does not include output logic.

One final word. It is possible to avoid essential hazards by providing sufficiently long delays in the feedback paths at appropriately chosen points in the circuit. The idea is to allow adequate time for the input changes to settle down before the secondary state variables change.

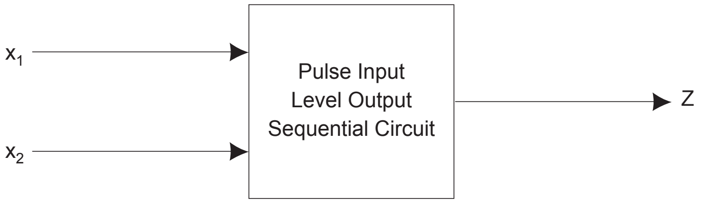
8.7 PULSE MODE ASYNCHRONOUS CIRCUITS

Pulse mode sequential circuits are useful for counting articles, events, or persons. Generally, no clock pulse generator is available to control the timing of events. Some kind of mechanism for producing pulses representing events has to be provided at the front end of the system. The outputs may be levels or pulses depending on the designer’s choice. The methods used for designing synchronous circuits are found to be suitable for **asynchronous pulse mode circuits** too. In this section, the objective is to present design techniques through an example.

8.8 DESIGN EXAMPLE 3

An office room is partitioned into two cabins to accommodate one officer in each cabin. In order to conserve electricity, the management decides to have an automatic system to switch off the power supply to the room when not in use. The room has two doors on either side used by the officers independently. Using photocells and relays, design a circuit with two pulse inputs and one level output to switch the power on or off.

Step I **Word statement** The circuit has two pulse inputs x_1, x_2 and one level output z shown in the block diagram below.



All the possibilities of the officers entering or leaving at their own random times are to be considered and the appropriate output is to be provided.

Let 0 indicate no pulse and 1 indicate a pulse either on x_1 lead or on x_2 lead, but not both at the same time. Let us use the same letters x_1, x_2 for indicating the entry and exit of officers for preparing the following table which leads to the next step.

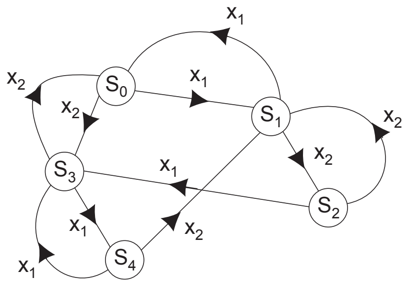
Event	x_1	x_2	Z	State
Initial	0	0	0	S_0
x_1 enters	1	0	1	S_1
x_1 leaves	1	0	0	S_0
x_1 enters	1	0	1	S_1
x_2 enters	0	1	1	S_2
x_2 leaves	0	1	1	S_1
x_1 leaves	1	0	0	S_0

Repeat, starting with S_2. It is more convenient to draw a state diagram, shown next.

Step II **State diagram** Note the following points which help in drawing the state diagram in Fig. 8.11(a) below.

- S_0 is the initial state when the room is vacant.
- S_1 is the state of the machine when only one officer x_1 is in the room.
- S_2 is the state of the machine when both x_1 and x_2 are in.
- S_3 represents the state when only x_2 is in.
- S_4 represents the state when x_1 enters the room and x_2 is already in.

Step III **State table** The information contained in the state diagram is posted in a tabular form yielding the state table shown in Fig. 8.11(b).



(a) State diagram

PS	NS		Z
	x_1	x_2	
S_0	S_1	S_3	0
S_1	S_0	S_2	1
S_2	S_3	S_1	1
S_3	S_4	S_0	1
S_4	S_3	S_1	1

(b) State table

Fig. 8.11 Design example of pulse input asynchronous sequential machine

Step IV **Reduction of state table and standard form** We reduce the state table using the partition technique. Associate the outputs to the states in the columns under NS. There are three different output configurations 11, 01, 10 which results in partition P_1.

$$P_0 = (S_0 S_1 S_2 S_3 S_4)$$
$$P_1 = (S_0 S_2 S_4)(S_1)(S_3)$$
$$P_2 = (S_0)(S_2 S_4)(S_1)(S_3)$$
$$P_3 \equiv P_2$$

Hence, $S_2 \equiv S_{4.}$ This result could have been observed at the beginning itself as these two states represent the state of the machine when both x_1 and x_2 are in the room.

Reduced State Table

PS	NS		Z
	x_1	x_2	
S_0	S_1	S_2	0
S_1	S_0	S_2	1
S_2	S_3	S_1	1
S_3	S_2	S_0	1

Step V **List the states of rows for checking the standard form**

$S_0\ S_1\ S_2$
$S_1\ S_0\ S_2$
$S_2\ S_3\ S_1$
$S_3\ S_2\ S_0$

The first occurrence is in order. Hence, it is in RSFST form.

Step VI There is no need to examine adjacencies in pulse mode circuits as the state transitions are triggered by pulses.

State Assignment and Transition Table

PS	NS ($Y_1 Y_2$)		Z
y_1y_2	x_1	x_2	
00	01	11	0
01	00	11	1
11	10	01	1
10	11	00	1

Step VII **Choose D flip-flops** Write the excitation functions by inspection.

$$D_1 = x_1 y_1 + x_2 y_1'$$
$$D_2 = x_1 y_2' + x_2 (y_1' + y_2)$$
$$Z = (y_1 + y_2)$$

Step VIII **Drawing the circuit is left to the reader.**

Note: 1. As we are dealing only with pulses, complement input variables do not arise. Even so, state variables may appear in both the forms.

2. The outputs are associated with the states. This is a Moore model.

8.9 ASYNCHRONOUS PULSE INPUT COUNTERS

In **asynchronous counters,** the count pulses drive only the first flip-flop in a cascade chain. The output of the first flip-flop drives the second flip-flop and the output of the second flip-flop drives the third flip-flop, and so on. For this reason, they are commonly called **ripple counters,** which cannot be described by Boolean equations used for clocked sequential circuits.

Contrast this with the synchronous counter wherein all the flip-flops are simultaneously driven by a clock pulse generator, in which case all the flip-flops ideally change state in parallel. In both types of counters, it is common to use J-K master-slave flip-flops connected as per design requirements. In the asynchronous case, the external pulses replace the clock for the first (LSB) flip-flop and the output of each flip-flop is connected to the clock input of the succeeding flip-flop. Remember that if both J and K are kept at 1 level (tied to the voltage V_{cc} in practice), the flip-flop changes state on every negative going transition at the output of the preceeding flip-flop.

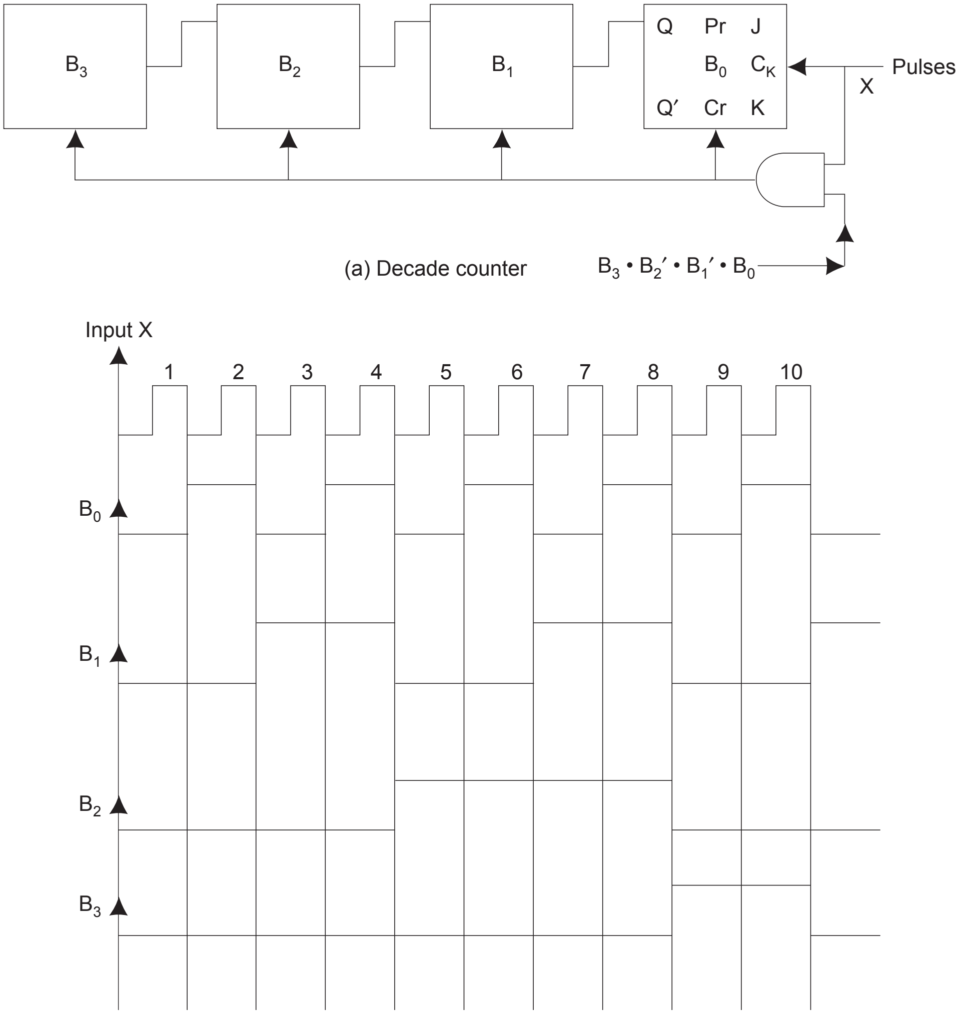
Counting Sequence for a decade counter

B_3	B_2	B_1	B_0	No. of Pulses on Input x
0	0	0	0	0 (zero)
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
0	0	0	0	10

A **decade counter** constructed with four flip-flops is shown in Fig. 8.12. Note that the binaries are arranged in the natural BCD order, with MSB to the left and LSB to the right. With four flip-flops it is possible to count from 0 to 15 but for the 10th input pulse, which ensures that the counter goes to the initial state. This is achieved in a number of ways. One simple method is to clear all the flip-flops at the prescribed time defined by the logic $X \bullet B_3 \bullet B_2' \bullet B_1' \bullet B_0$ where X is the input lead on which count pulses occur and B_i represents the corresponding outputs of the flip-flops.

The counting sequence for the **decade counter** is given above. Notice that B_0 changes state for every incoming pulse on the negative-going edge, B_1 for every two pulses, B_2 for four pulses and B_3 at the end of eight pulses. The waveforms at the outputs of flip-flops are shown in Fig. 8.12(b). All the four flip-flops are cleared to 0 on the 10th pulse which enables the AND gate to which other inputs are B_3, B_2', B_1', B_0 whose states represent the count 1001 = 9. Note that the negative-going transition of B_3 as the counter proceeds from 9 to 0, can be used to drive another decade counter block. Any method which does not involve abrupt clearing of all the four flip-flops is not simple. It requires a great deal of imagination and ingenuity on the part of the designer to build a decade counter which counts in a straight binary sequence from 0000 upto 1001 and back to 0000 using a four flip-flop chain. One such circuit is shown in Fig. 8.13. The counting sequence is the same as above.

Let us refer to the four binaries by their outputs—B_0, B_1, B_2 and B_3. Notice carefully the following in respect of two aspects namely, "driving the flip-flops" and "mode setting".



(a) Decade counter

(b) Waveforms

Fig. 8.12 Asynchronous decade counter

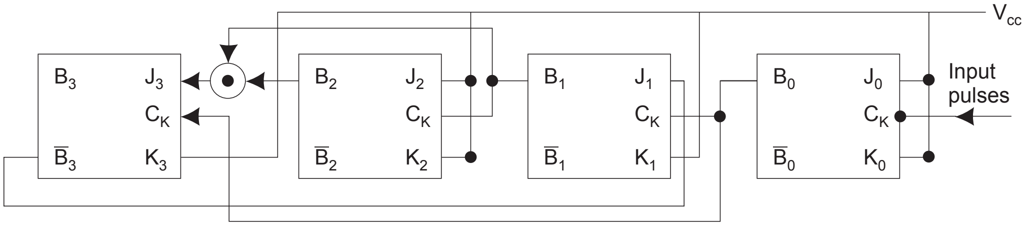


Fig. 8.13 Decade counter (alternative scheme)

1. Input pulses drive B_0 at its clock input.
2. B_0 drives B_1 and B_1 drives B_2
3. B_0 has to drive B_3 also for the simple reason that the 10th pulse causes 1 → 0 transition only at B_0 which has to cause similar transition at B_3 too. [Look at the counting sequence and the waveforms of Fig. 8.12(b)].
4. All $K_i = 1$ as they are tied to V_{cc}. Only $J_0 = J_2 = 1$ as they are tied to V_{cc}. Hence, B_0 and B_2 are set in toggle mode.

   $J_1 = B_3' = 1$ until the end of the 8th count pulse as there is need for keeping B_1 in toggling mode until then. (See the counting sequence.) After the 8th count, J_1 becomes 0 while $K_1 = 1$ which sets B_1 in a clearing mode.

   $J_3 = B_1 \bullet B_2$ which becomes 1 at the end of the 6th count pulse and remains at 1 till the 8th count pulse toggles B_0, B_1, B_2 and B_3 in that order by rippling through the binaries. Thereafter, B_3 would be in a clearing mode as $J_3 = 0$ and $K_3 = 1$.

   To sum up, the position on the 10th pulse is

   $J_0 = K_0 = 1 \Rightarrow B_0$ is in toggle mode $\Rightarrow 1 \rightarrow 0$

   $J_1 = 0$, $K_1 = 1 \Rightarrow B_1$ is in clearing mode $0 \rightarrow 0$

   $J_2 = K_2 = 1 \Rightarrow B_2$ is in toggle mode but there is no driving pulse.

   Hence $0 \rightarrow 0$

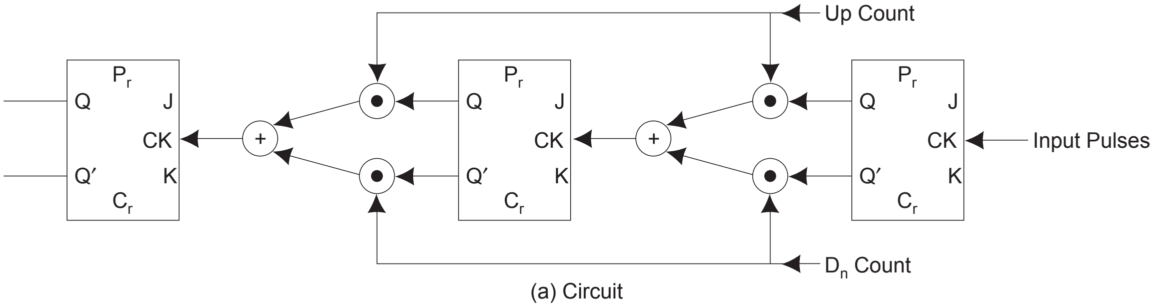
   $J_3 = 0$, $K_3 = 1 \Rightarrow B_3$ is in clearing mode $\Rightarrow 1 \rightarrow 0$

An inquisitive reader might wonder why not make $J_3 = B_2$ instead of $B_1 \cdot B_2$. In that case, B_3 would be in a toggling mode at the end of 4th count pulse and the 1 → 0 transition of B_0 at the 6th count pulse would cause the toggling of B_3 which is wrong and hence prevented as $B_1 = 0$ before the 6th count pulse. For this reason, B_1 and B_2 jointly control the excitation J_3 of B_3.

Up-down Counter

Hitherto we used only upward counting. Downward counting can be obtained by simply using the Q′ output of the flip-flop to drive the next flip-flop. In this case, the positive-going edges (at Q) cause a change of state in the succeeding flip-flop. The counter has to be initialised to all 1s by pulsing the preset leads.

Look at the typical up count and down count for a binary counter with three flip-flops shown in Fig. 8.14. This can count up from 0 through 7 or count down from 7 to 0. The logic circuit between each pair of flip-flops enables either up count or down count.



(a) Circuit

Up-count			Down-count			Pulses
B_2	B_1	B_0	B_2	B_1	B_0	
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	2
0	1	1	1	0	0	3
1	0	0	0	1	1	4
1	0	1	0	1	0	5
1	1	0	0	0	1	6
1	1	1	0	0	0	7
0	0	0	1	1	1	8

(b) Counting sequence

Fig. 8.14 3-bit binary up-down counter

It is important for the reader to particularly note that the facility of up-down counting is simply obtained by having the option of choosing Q or Q′ output to drive the next flip-flop. This kind of arrangement is used only if the count advances in a straight binary progression. Initialisation can be done easily by using either all preset or all clear inputs of the flip-flops.

Can you think of obtaining a down count for the decade counter? Does skipping counts cause problems? Think it over

SUMMARY

The reader is exposed to common life situations requiring asynchronous sequential machines. The subject of design of asynchronous sequential machines is introduced through design examples. Design steps starting from primitive flow table ending with the hardware circuit has been illustrated. Cycles, races, critical and non-critical races have been illustrated. Procedures are given to select a valid assignment of internal states. Essential hazards and their identification and elimination are presented. Design of asynchronous pulse-input-counters has received due emphasis.

KEY WORDS

- Fundamental mode circuits
- Pulse-mode circuits
- Fundamental mode
- Stable state
- Flow table
- Transitory (unstable) states
- Primitive flow table
- Merger of rows
- Incompletely specified sequential machines (ISSM'S)
- Compatibility
- Input state
- Total state
- Race
- Critical race
- Non-critical race
- Cycle
- Essential hazard
- Asynchronous counters
- Ripple counters
- Decade counter

REVIEW QUESTIONS

1. What are the basic differences in synchronous and asynchronous sequential circuits?
2. How are asynchronous machines classified? State the features of each type.
3. Which component acts as memory in fundamental mode asynchronous sequential circuits?
4. Distinguish between stable state and transitory state. How are they indicated in a flow table?
5. List the design steps for a fundamental mode asynchronous sequential machine.
6. What is the characteristic of a primitive flow table?
7. What do the dashes in a primitive flow table indicate?
8. How are the outputs indicated?
9. Distinguish between internal state, total state, secondary state, and input state.
10. What are the rules for merger of rows? Why do we choose mutually disjoint merger classes of rows of primitive flow table.
11. Why should we examine adjacency requirements between rows of a merged flow table?
12. Define race and distinguish between critical and non-critical races.
13. What is meant by a valid assignment?
14. What is meant by a cycle. Where is it used?
15. Give an example of a cycle which does not terminate in a stable state.
16. How do we ensure race-free assignment?
17. What is meant by essential hazard in asynchronous circuits?
18. How do you detect essential hazards in a flow table?
19. How do you avoid essential hazards?
20. State two examples where pulse mode asynchronous circuits are exployed.
21. Complemented input variables do not occur in pulse mode asynchronous circuits. Why?
22. There is no need to examine adjacencies in pulse mode asynchronous circuits. Why?
23. What is the advantage of choosing D flip-flops in sequential circuits?
24. What is the distinction between asynchronous counters and synchronous counters?
25. Why are asynchronous counters called ripple counters?
26. What is the least number of flip-flops required to build a decade counter?
27. What are the strategies adopted to convert a four flip-flop binary counter into a decade counter?

PROBLEMS

1. Determine the primitive-flow table of a fundamental mode sequential circuit with two inputs, x_1 and x_2. The single output, z, is to be 1 only when $x_2x_1 = 10$, provided that this is the fourth of a sequence of input combinations 00 01 11 10. Otherwise the output is to be 0. Ensure that no spurious 1 outputs occur during transitions between two states with 0 outputs. Both inputs will not change simultaneously.
2. Determine a minimal-row equivalent of the given flow table. The machine gives an output 1 in state ④ only.

X_1X_2			
00	01	11	10
①	2	--	5
1	②	3	--
--	4	③	9
1	④$_1$	3	--
1	--	6	⑤
--	7	⑥	8
1	⑦	6	--
1	--	6	⑧
1	--	3	⑨

3. Find a secondary state assignment to the minimal-flow table given. Construct a transition table that has no critical races. Obtain Boolean expressions for excitations.

X_1X_2				Z_1Z_2 X_1X_2			
00	01	11	10	00	01	11	10
①	2	①	3	00	--	01	--
1	②	②	②	--	00	00	00
1	③	2	③	--	10	--	

4. Obtain the minimum row merged flow table for the given primitive. Find a valid assignment and obtain the transition table.

X_1X_2				X_1X_2			
00	01	11	10	00	01	11	10
①	2	--	4	0			
1	②	3	--		1		
--	2	③	5			1	
1	--	3	④				
1	--	6	⑤		0		0
--	2	⑥	5			0	0

5. A circuit has a single-input line on which pulses of various widths will occur at random. Construct the primitive flow table of this circuit with a single output, z, on which a pulse will occur coinciding with every fourth input pulse. Since this circuit is part of a random-process generation scheme, the power on state is of no interest.
6. A sequential circuit has two level inputs and two level outputs. The outputs are coded to count in the straight binary code Modulo-4, as shown in the table below. The count is to increase by one for each 0-to-1 transition of either input occurring when the other input is at the 0 level. Both the inputs will not change at the same time. Determine a primitive flow table of a circuit to meet this specification.

$z_1\ z_0$	Count	Next Count
00	0	1
01	1	2
10	2	3
11	3	0

7. An asynchronous sequential network has two inputs X_1 and X_2 and one output Z. When the input $x_1\,x_2$ is 11, the output becomes 1 and stays 1 until the input that immediately preceded the $X_1X_2 = 11$ occurs again, at which time the output becomes 0. When the next 11 is detected, the network performs this operation again.

   Example

   $X_1X_2 =$ 00 01 10 11 00 01 11 10 11 10 01 00

   Z = 0 0 0 1 1 1 1 0 1 0 0 0

   The first input in the sequence is not allowed to be 11. Find a minimum-row flow table.
8. A sequential network has two inputs (X_1X_2) and one output (Z). If the input sequence 00, 01, 11 occurs, Z becomes 1 and remains 1 until the input sequence 11, 01, 00 occurs. In this case Z becomes 0 and remains 0 until the first sequence occurs again. (Note that the last input in one sequence may be the first input in the other sequence.) Find a minimum-row flow table.
9. At a junction of a railroad and a road, traffic lights are to be installed. The lights are controlled by switches, which are pressed or released by the trains. When a train approaches the junction from either direction and is within one kilometre from it, the lights should change from green to red and remain red until the train is one kilometre past the junction.
   a) Write a primitive flow table assuming that the length of the train is smaller than 2 kilometres.
   b) Obtain a merged flow table and show a logic gate network to control the lights.
10. A sequential circuit has two pulse inputs, x_1 and x_2. The output of the circuit becomes 1 when one or more consecutive x_1 pulses are followed by two x_2 pulses. The output then remains 1 for all subsequent x_2 pulses until an x_1 pulse occurs.
    a) Derive a minimal state table.
    b) Synthesise using S-R flip-flops.

11. In the 6-row primitive flow table given below, determine whether there are any redundant equivalent states. After removing any that occur, draw the merger graph. List all pairs (of row numbers) that cannot be enlarged to triple, and all triples that cannot be enlarged to quadruples. Determine the simplest merged flow table and the corresponding output or Z matrix, indicating don't cares with dashes.

00	01	11	10	Output Z
--	3	①	4	0
5	②	1	--	1
6	③	1	--	0
5	--	1	④	0
⑤	2	--	4	1
⑥	3	--	4	0

12. For the partially filled primitive, fill in the transitional entries so that the circuit will realise the input sequence $X_1X_2 = 00 – 01 – 11 – 10$ and no others. Derive the minimum row merged flow table. Choose a secondary assignment. Find the excitation matrix with no critical races and assign a flicker-free fast output.

x_1x_2 00	01	11	10	Z
①		--		0
--	②	3	--	0
--		③		0
	--		④	1
	--		⑤	0
	⑥		--	0
--		⑦		0

13. a) Find all the races in the flow table given below. Indicate those that are critical and those that are not.

b) Find another assignment which has no critical races.

PS y_1y_2	NS(Y_1Y_2)			
	x_1x_2			
	00	01	11	10
00	$\textcircled{00}\rightarrow$	11	$\leftarrow\textcircled{00}$	11
01	11	$\textcircled{01}$	11	11
11	$\textcircled{11}$	$\textcircled{11}\rightarrow$	00	$\leftarrow\textcircled{11}$
10	00	$\textcircled{10}$	11	11

14. An asynchronous fundamental mode sequential circuit is to have two inputs, x_1, x_2, and one output, z. When $x_2 = 1$, the value of z equals the value of x_1, when $x_2 = 0$, the output remains fixed at its last value prior to x_2 becoming 0. Obtain the minimum-row merged flow table, a valid assignment, and the excitation and output functions.
15. In a certain digital system, a major cycle consists of occurrence of three pulses, one on each of the leads A, B, C in that order. A sequence checker will receive these three pulses and also a check pulse K at the end of each major cycle. When the check pulse on the K input lead occurs, the sequence checker must reset and output an error pulse if the pulses A, B, C are not received in the order. Obtain a state table in RSFST form for this machine.
16. A lift (elevator) in a multistorey building can carry at most five persons. Work out a scheme to count the persons inside and give a signal to other callers.
17. At the junction of four roads going north, south, east, and west, it is proposed to install traffic lights of three colours—red (R), amber (A), and green (G)—with flexible time settings. Design an asynchronous system. Assume that an electronic timer is available.
18. Draw the circuit of a decade ripple counter using four flip-flops and explain its working with a table showing the counting sequence.
19. Draw the circuit of an asynchronous decade counter with four flip-flops that does not use the clear (C_r) leads for resetting the counter.
20. Where are UP-DN counters used? Discuss the operation of a 3-bit up-dn counter indicating the counting sequences.
21. Can you think of a strategy to design an up-dn decade counter?

9

Minimisation of Sequential Machines

LEARNING OBJECTIVES

After studying this chapter, you will know about:

- Limitations of finite state machines.
- Mealy and Moore Models.
- State minimisation of completely specified machines using Partition technique, and Merger chart.
- State reduction in incompletely specified sequential machines.
- Bounds on minimal state machines.

9.1 LIMITATIONS OF FINITE STATE MACHINES

There are some problems which can not be solved by any **finite state machine (FSM).** The objective of this section is to present some simple problems of this category.

We have discussed in a previous chapter how serial binary addition of two words of any length can be performed by a sequential machine with two states only. Such a sequential machine (circuit) is also called **finite state automaton (FA),** or simply **finite state machine.** A question that now arises is whether multiplication of two arbitrarily long binary numbers can also be performed by a finite state machine. The answer is "no". The following paragraphs elaborate on the capabilities and limitations of finite state machines.

Suppose we give a periodic input such as a continuous string of 1s (period is one bit time) to a sequential machine and that we want the machine to produce an output 1 for the 1st input, 3rd input, 6th input, 10th input, and so on, that is, for every input numbered $\frac{K(K+1)}{2}$ where K is an integer starting from 1, going upward. The typical input X and output Z are shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	. . .
Input X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	. . .
Output Z	1	0	1	0	0	1	0	0	0	1	0	0	0	0	1	0	. . .

Clearly, the output does not become periodic. No finite state machine can be designed to produce such a non-periodic infinite sequence for a periodic input. Suppose you keep repeating the same input more than n times, where n is the number of states of the FSM. A stage will come when the machine re-enters a state a second time. Thereafter, the machine has to visit its states in a periodic fashion with a period $\leq$ n.

As another example, consider the problem of multiplication of two binary numbers, which are arbitrarily long. We could design a serial adder as the pairs of inputs could be classified into two distinct classes, one producing a carry 1 and the other producing no carry. We needed only one-bit memory to store a single-bit carry. In the case of serial multiplication, it is necessary to produce partial products and add them to produce the final product. Suppose, for a moment, there exists an n state machine which can produce the product of two arbitrarily long numbers. Let us choose the two numbers for multiplication as $2^p \times 2^p = 2^{2p}$. Further, assume that p is greater than n. Each of the two numbers is represented by 1 followed by p 0s. The product would be 2^{2p} which is denoted by 1 followed by 2p 0s as shown below. The input numbers are serially fed to the machine with LSB first and MSB last. This is done during the first (p + 1) time-slots, that is, from t_1 till $t_{p+1.}$ In the duration between t_{p+1} and t_{2p+1}, the machine does not receive any inputs but it has yet to produce some more 0s followed by a 1 at t_{2p+1} which is not realisable for the following reason.

Time slot	t_{2p+1}	t_{2p}	. . . .	t_{p+1}	t_p	. . . .	t_2	t_1
Multiplicand				1	0	. . . .	0	0
Multiplier				1	0	. . . .	0	0
Product	1	0	. . . .	0	0	. . . .	0	0

Initially we assumed that p was greater than n. The machine has already received p 0s on the inputs. Therefore, the machine must have been twice in one of its states since p $\geq$ n, and the output must be periodic from that point onwards. Hence, the machine will never produce the required 1 at t_{2p+1}.

In spite of the above assertion that arbitrarily long numbers cannot be multiplied by any finite state machine, the reader should note that for any two finite numbers, we can find a FSM which can multiply them and produce the product.

Machine Identification

Engineers dealing with sequential circuits find analysis and identification much more challenging than synthesis. While there are systematic methods for synthesis, the general problem of identification is quite complex. Analysis calls for ingenuity on the part of engineers to figure out the sequences accepted by the given machine. Accepted sequences produce output 1 on the application of the last input symbol. In general, a sequential machine transforms one input sequence to another sequence at the output.

Testing of sequential machines is another challenging area. A machine is said to be **"strongly connected"** if there exists a sequence of inputs which takes the machine from any one of the states to every other state of the machine. Machines which are not strongly connected pose some problems in **"diagnosis"**, which is a distinct discipline. In the design of sequential circuits, we often come across a situation that the machine may contain a **"source state"** which has only outgoing edges but no incoming edges. Likewise, some machines may have **"sink states"** which have only incoming arrows but no outgoing edges.

9.2 MEALY AND MOORE MODELS

In the previous chapters you learnt two possible models of sequential machines. Let us examine these models further and learn how to convert one form to the other form. The basic definition is as follows.

Definition 9.1 A sequential machine M is a quintuple comprising a set of inputs I, a set of outputs Z, a set of states S, a transition function δ which enables finding the next state depending on the present state and present input and finally an output function λ. This is symbolically expressed as M = (I, Z, S, δ, λ).

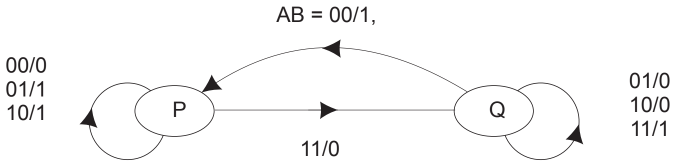
If the output function depends on the present state and present inputs, it is called the **Mealy model,** named after G.H. Mealy, a pioneer in this field. If the output is associated only with the present state, it is called the **Moore model,** named after another pioneer E.F. Moore. The counters are clearly **Moore machines** as the output depends only on the states of the flip-flops. Likewise, a **sequence detector** is also a **Moore machine. Serial adder,** discussed earlier, is an example of a **Mealy machine** as each one of its states is reached producing a 0 or 1 output depending on the starting state and the value of the inputs. Fundamental mode asynchronous sequential machines, discussed in an earlier chapter, are clearly Mealy machines as the outputs depend on the input column apart from the internal state. Is it possible to convert a Mealy machine into an equivalent Moore machine and vice versa?

Mealy to Moore conversion Let us learn how to convert a Mealy machine into a Moore machine. Let us use the familiar serial adder as an example to illustrate the procedure. The state diagram and the state table of the synchronous serial adder are given in Fig. 9.1(a) and (b) respectively. Notice that the state P is reached from the state Q on the application of the inputs AB = 00 and, in the process, the machine produces an output Z = 1 indicated on the arc as 00/1. Also notice that the machine produces output Z = 0 in another transition to P. This transition is indicated as a self loop around P on inputs AB = 00/0. For AB = 01 or 10 while in P, the machine produces an output 1 and remains in the same state P.

Since the transition to state P is associated with both the outputs, 0 and 1, we split the state into two equivalent states and call them P_0 and P_1. Every transition to P with an output 0 will be directed to P_0 and every transition to P with an output 1 will be directed to P_1. For similar reasons, state Q is also split into Q_0 and Q_1. The modified state table and the state diagram with four states are shown in Fig. 9.1(c) and (d) respectively. Clearly, the modified machine of Fig. 9.1(c), which is in the form of a Moore model, has to be equivalent in behaviour to the Mealy model of Fig. 9.1(b).

We make this statement as the two machines have to give the same output sequence for any arbitrary input sequence. The reader may make two observations.

1. If the Mealy machine has K states, the equivalent Moore machine will have at most 2K states if the output is a binary variable.
2. There is no power-on state, unless specifically defined. A special state may be introduced if the user wants one.



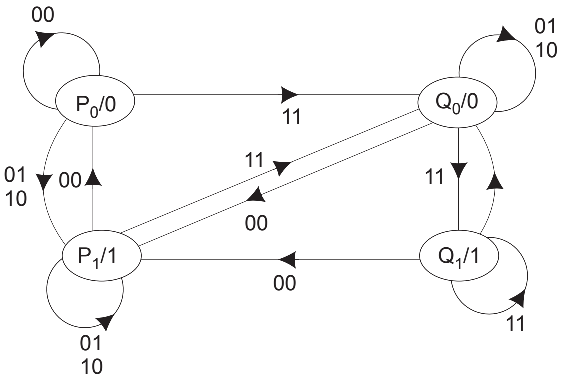
(a) State diagram of a serial adder as a Mealy machine

PS	NS, Z Inputs A B			
	00	01	11	10
P	P, 0	P, 1	Q, 0	P, 1
Q	P, 1	Q, 0	Q, 1	Q, 0

(b) Mealy state table

	NS, AB				Z
PS	**00**	**01**	**11**	**10**	
$\mathbf{P_0}$	P_0	P_1	Q_0	P_1	0
$\mathbf{P_1}$	P_0	P_1	Q_0	P_1	1
$\mathbf{Q_0}$	P_1	Q_0	Q_1	Q_0	0
$\mathbf{Q_1}$	P_1	Q_0	Q_1	Q_0	1

(c) Moore state table



(d) State diagram of a serial adder as a Moore machine

Fig. 9.1 Mealy to Moore conversion

3. In a Mealy machine, the next state and output are given at the same place separated by a comma while a Moore machine has an exclusive output column.

Moore to Mealy conversion It is amazingly simple to convert a Moore machine into a Mealy machine. Let some state S_i of a given Moore machine be associated with an output Z_i. What we need to do is simply associate the output Z_i wherever S_i occurs as the next state by scanning all the input columns. This is indicated by replacing all S_i by S_i, Z_i. We then call the state S_i as Z homogeneous. All states of a Moore machine are Z homogeneous.

The reader is advised to convert the Moore machine of Fig. 9.1(c) into Mealy form and reduce it and thereby understand the behavioural equivalence of the two models.

9.3 MINIMISATION OF COMPLETELY SPECIFIED SEQUENTIAL MACHINES

It often happens that the state diagram and the corresponding state table contain redundant states, that is, states whose functions can be performed by other states. We know that the number of flip-flops required is a direct function of the number of states. K flip-flops can provide 2^K distinct states. **State reduction** helps in reducing the number of flip-flops and the associated logic hardware. A more important aspect of **state minimisation** is that **diagnosis** of sequential machines becomes much simpler if the machine is in minimal form.

Synchronous sequential machines are usually completely specified in the sense that for every applicable input, we specify the next state and output. Yet, there are situations where certain combinations of inputs and states are not possible. For example, the decade counter has 10 states and 6 other states never occur. Corresponding states and outputs may be left unspecified which are marked usually by dashes. The objective of this section is to discuss **minimisation of completely specified machines** and the subsequent sections are devoted to **reduction of incompletely specified sequential machines.**

Definition 9.2 Two states, S_i and S_j, of a machine M are said to be **K-distinguishable** if there exists an input sequence of length K which produces different output sequences depending on whether S_i or S_j is the initial state.

Consider the machine M_1 of Table 9.1(a). If you apply a single input symbol, either x = 0 or 1 starting from state A or B, the output will be identically 0. Hence we say that A and B are 1-equivalent or 1-indistinguishable. Applying an input and observing the output is referred to as an experiment. Suppose we perform the same experiment with states A and C, we find that on the application of x = 1, the machine gives different outputs starting from A and C. The transitions are conventionally indicated as

$$A - 1 = B, 0 \text{ and } C - 1 = B, 1$$

We conclude that A and C are 1-distinguishable and therefore cannot be equivalent.

Extending the concept of distinguishability, we say that if no experiment exists which distinguishes between a pair of states, then we say that such states are equivalent. In other words, such states are K-equivalent for all values of K. This concept is now put in the form of a definition given below.

Definition 9.3 States S_i and S_j of a machine M are said to be **equivalent** if and only if for every possible input sequence, an identical output sequence will be produced regardless of whether S_i or S_j is the starting state.

Partition Technique

Take a look at the K-partitions shown in Table 9.1(b). Clearly, the 0 partition P_0 contains all the states of the machine in one group indicated by brackets, because by applying 0 inputs, that is, no inputs at all, it is not possible to distinguish between the states. If you apply any one input, either x = 0 or x = 1, observe that the outputs, A, B, F cause the corresponding output pattern to be 00 while the states C, D, E cause the outputs to be 01 in the two input columns of the corresponding rows. Thus, by merely observing the outputs, we may form the 1-partition P_1 as (ABF), (CDE). This means that no state in one equivalence class can be equivalent to any member of another class.

From now on, there is no need to observe the outputs to distinguish the states one from the other. Instead, we compare every pair of states within the same bracket and note the **successor pairs,** or **implied pairs**, in every input column. From the original definition, it follows that a pair of states will be equivalent if and only if the successor pairs are also equivalent. If each successor pair is within one bracket, we retain the pair intact; otherwise we split the pair. Suppose in P_2, we consider the transition from pair AB in the x = 0 column and x = 1 column. In one case, it is the same state repeated. In the other case it is the same pair under consideration. This means that “equivalence of A and B implies equivalence of A and B”–a strange proposition which is to be ignored! Thus, we take note of implied pairs and examine whether they are in one bracket of the same partition. If the states of the implied pair happen to be in different brackets, then split the pair under consideration. For example, AF ⇒ BC in column x = 1 and the states B and C are in different brackets. Hence, AF has to be split which is marked with an inverted 'V' in Table 9.1(b). For a similar reason, BF is also marked for splitting as BF ⇒ AC and the states A.,C are in different brackets. Now compare the pairs in the bracket (CDE). We find that no more splitting is possible at this stage and we obtain the next partition P_2 in Table 9.1(b).

Continuing the process, we find that neither C and E nor D and E can be equivalent. Hence E parts company from (CD) in the next partition P_3. Continuing further, we find that P_4 is identical to P_3 and hence we stop here and conclude that no experiment exists to distinguish between the states (AB) and (CD). Hence, we take them as equivalence classes. For each equivalence class, we need to retain only one state and substitute this for the other states of the same class wherever required. The reduced minimal state machine M_2 is given in Table 9.2(a).

Table 9.1 *Illustrative Example*

(a) Machine M_1

PS	NS, Z	
	X = 0	X = 1
A	F, 0	B, 0
B	F, 0	A, 0
C	E, 0	B, 1
D	E, 0	A, 1
E	C, 0	F, 1
F	B, 0	C, 0

(b) K Equivalence Partitions

P_0 = (ABCDEF)

P_1 = $(\widehat{ABF})$ (CDE)

P_2 = (AB) (F) $(\widehat{CDE})$

P_3 = (AB) (F) (CD) (E)

P_4 = (AB) (F) (CD) (E) $\equiv P_3$

Hence A ≡ B, C ≡ D.

Table 9.2 *Reduced Machine M_2*

(a) Machine M_2

PS	NS, Z	
	X = 0	X = 1
A	F, 0	A, 0
F	A, 0	C, 0
C	E, 0	A, 1
E	C, 0	F, 1

(b) Row-wise Occurrence of States

AFA
FAC
CEA
ECF

Set $A = S_{1,}$ $F = S_2$
$C = S_{3,}$ $E = S_4$
and rewrite the table.

(c) Transformed Table

PS	NS, Z	
	X = 0	X = 1
S_1	S_2, 0	S_1, 0
S_2	S_1, 0	S_3, 0
S_3	S_4, 0	S_1, 1
S_4	S_3, 0	S_2, 1

(d) RSFST

PS	NS, Z	
	X = 0	X = 1
A	B, 0	A, 0
B	A, 0	C, 0
C	D, 0	A, 1
D	C, 0	B, 1

Reduced Standard form State Table (RSFST)

Sometimes it becomes necessary to compare two machines with the same number of states to decide whether they represent the same machine. For this purpose, standard form (also called canonical form) helps in quickly arriving at the decision. All that we do is to enumerate the occurrence of states row-wise and check the first occurrence of the states. See Table 9.2 (b). Number the states in the order of their first occurrence and write a transformed state table which is clearly in standard form. Whether one wishes to have alphabets or numerals is the designer's choice. Table 9.1 (d) shows M_1 in RSFST form.

9.4 MERGER CHART METHOD

The partition technique can be implemented conveniently using a merger chart of M_1 shown in Table 9.3. A **merger chart** is also known as an **implication chart.** In the case of incompletely specified machines, it is referred to as a **compatible pair chart** also. Given that the machine M_1 has six states, we prepare a merger chart of Table 9.3(a) with five slots on the y-axis and five slots on the x-axis and draw the rectangles and label them with the states as shown. The first state A will have only column and the last state F will have only row. Each of the other states will be associated with a column and a row.

The first step in making the entries in the merger chart is to identify the pairs of states which cannot become equivalent and post this information with a cross in the corresponding block. Looking at the state table of M_1 in Table 9.1(a), we find that on the transition A – 1, the machine produces an output 0. In the same input column corresponding to x = 1, the rows C, D, E produce an output 1. We therefore conclude that state A can in no case become equivalent to any of the states C, D, E whose rows have 1 output in that column. Hence, we cross the blocks corresponding to the pairs AC, AD, AE.

By scanning every pair of rows in the state table and observing outputs only, the inequivalence is marked for the pairs BC, BD, BE, CF, DF, EF. Thus far, we marked inequivalence in the first round, observing only the outputs.

Table 9.3 *Merger Chart*

(a) Merger Chart of M_1

	A	B	C	D	E
B	√				
C	☒	☒			
D	☒	☒	AB		
E	☒	☒	~~BF~~	~~AF~~ ~~CE~~	
F	~~BF~~ ~~BC~~	~~AC~~	☒	☒	☒

(b) Equivalence Classes

(E) (F)
(D) (E) (F)
(CD) (E) (F)
(B) (CD) (E) (F)
(AB) (CD) (E) (F)

We then scan each pair of potentially equivalent (uncrossed so far) rows and mark the **"successor pairs"** in each input column. Such pairs are also called **"implied pairs",** noted in the corresponding cell of the merger chart. For example, AF ⇒ BF and BC. That is, AF will be equivalent if the states BF and BC are equivalent. The pair BC is already crossed and hence the pair AF also will now be crossed.

Continue this process of scanning the chart until no more cells can be crossed. Eventually, the cells remaining uncrossed in the merger chart are the **equivalent pairs of states.** In M_1, they are AB and CD.

We now proceed to generate the set of **"equivalence classes"** of the machine M_1 in a systematic manner, illustrated in Table 9.3(b). We start with the last column of the merger chart and note that the states E and F are not equivalent. Hence we put them in separate brackets. Next, we look at column D and note that D cannot merge with any existing groups. We update the list as shown. Then, we scan column C and observe that C can merge with D; and we enlarge the existing entry and write it as (CD) (E) (F). Next, we scan the B column and find that B cannot merge with any of the states in the current list. We update the list as (B) (CD) (E) (F). Lastly, we look at column A and the final list of equivalence classes is given by (AB) (CD) (E) (F). Based on these equivalence classes, the reduced machine M_2 is already contained in Table 9.2(a) and its standard form in Table 9.2(d).

Important Observations

The reader should take note of the following observations

1. Equivalence partition is unique
2. Equivalence partition contains all the implied classes. This property is called **"Closure"** which is discussed in detail is subsequent sections. In the present example, AB ⇒ CD which is in the set of **equivalence classes.**
3. In the partition technique, we started with all the states of the machine in one bracket and went on to establishing inequivalence and splitting the group, eventually yielding **"equivalence partition"**. In contrast, the merger chart method keeps an enlarging the states in brackets, eventually yielding the same **equivalence partition.**
4. Equivalence is a binary relation with the properties of **"reflexivity", "symmetry",** and "**transitivity".**
5. Equivalence classes are mutually disjoint sets.

9.5 STATE REDUCTION IN INCOMPLETELY SPECIFIED SEQUENTIAL MACHINES

The definition of a sequential machine is repeated below for the purpose of reinforcing the concept.

Definition 9.4 A **sequential machine** M is a quintuple

$$M = (I, Z, S, \delta, \lambda)$$

where I, Z and S are finite non empty sets of inputs, outputs, and states respectively.

δ : $I \times S \rightarrow S$ is the state transition function

λ : is the output function such that

: $I \times S \rightarrow Z$ for Mealy machines

: $S \rightarrow Z$ for Moore machines

The Cartesian product I × S is the set containing all pairs of elements ($\mathbf{I}_i$, s_j). The state transition function associates with each pair (I_i, s_j) an element s_k from S, called the "next state". In a **Mealy machine,** the output function λ associates with each pair (x_i, s_j) an element z_k from Z, while in a **Moore machine** a direct correspondence exists between the states and the outputs.

The treatment presented in this chapter is perfectly general and applies to both synchronous as well as asynchronous sequential machines whether in Mealy or Moore form. However, for our discussion and illustration, we will use only Mealy type of state tables.

The word statement of a sequential machine usually specifies the desired output sequences for certain input sequences and may also specify forbidden input sequences. A given sequential machine is specified by means of its **state table** (also called **flow table**). For any present state-input pair S_i, I_j, the state table displays the next state of the machine and the output. In constructing the state table of a finite-state machine, it often happens that the table contains redundant states, that is, states whose functions can be accomplished by other states. Definitions 9.2 and 9.3 are related to completely specified machines in the previous sections. The following additional definitions help in formulating the reduction process of incompletely specified sequential machines.

Definition 9.5 Two states, s_i and s_j, of machine M are **compatible** if and only if, for every input sequence applicable to both s_i and s_j, the same output sequence will be produced whenever both outputs are specified, regardless of whether s_i or s_j is the initial state.

It clearly follows that s_i and s_j are **compatible** if and only if their outputs are not conflicting (that is, identical when specified) and their I_i–successor states for every I_i for which both are specified, are either the same or also compatible. The **compatibility relationship** is reflexive and symmetric but not transitive.

Definition 9.6 A set of states C = {s_i, s_j, s_k, …} is said to be a **compatibility class** (or compatible) if and only if every pair of states in C is compatible.

The set parentheses are understood and are usually omitted. The compatibles are represented by simple blocks such as abc, def…and soon.

Definition 9.7 A **maximal compatibility class** (or **maximal compatible)** is a compatible which is not a subset of any other compatible.

Definition 9.8 A compatible C_i is said to **imply** a compatible C_j if C_j is the set of I_i–successor states of C_i for some I_i. It is important to notice the transitivity of this relationship.

Definition 9.9 The **closure class set** E_i of a compatible C_i is the set of all compatibles implied by C_i obtained by repeated use of the transitivity of implication such that the compatibles which are subsets of either C_i or any other member of E_i are removed from the set. The members of E_i are referred to as "closure conditions" of C_i.

Definition 9.10 A set of compatibles is said to be **closed** if and only if for every compatible contained in the set, each implied compatible is also contained in at least one member of the set. In other words, a closed set of compatibles is a set of compatibles which is closed with respect to the binary relation "implication".

A closed set of compatibles covers a given flow table F if every state of F is contained in at least one compatible of the set.

The problem of state minimisation for a given sequential machine M is equivalent to finding a closed set containing the minimum number of compatibles which covers all the states of M. This collection will be called the **minimal closed cover** or **minimal cover.**

Selecting a minimal closed cover from the entire set of compatibles requires, in general, a large amount of enumeration and searching. One technique consists of eliminating some compatibles before starting the selection process if it can be proved that such elimination does not preclude finding at least one minimal closed cover from the remaining compatibles. This is the basic approach and the tests should be conceptually simple and easy to implement.

Definition 9.11 State s_i of machine M_1 is said to **cover** state s_j of machine M_2 if and only if every input sequence applicable to s_j is also applicable to s_i and its application to both M_1 and M_2 when they are initially in s_i and s_j respectively, results in identical output sequences whenever the outputs of M_2 are specified. The covering concept can be extended to machines by the following definition..

Definition 9.12 Machine M_1 is said to **cover** machine M_2 if and only if for every state s_j in M_2, there is a corresponding state s_i in M_1 such that s_i covers s_j. Note that the covering relationship is reflexive and transitive but not symmetric.

Paull–Unger Method

In this method, we first enumerate all the compatible pairs (two-state compatibles) and their corresponding implied pairs in the form of the **implication chart**.

Paull–Unger called this a "pair chart". Kohavi called it "merger table". Biswas preferred to call it the "implication Chart". We will use this last name in the book.

Each pair of rows of the given state table is compared for compatibility relationship. For the machine M_1 of Table 9.4, the implication chart is given in Table 9.5. The compatible pairs which do not imply any other pairs, are indicated with a check mark in Table 9.5; the incompatible pairs are indicated by a cross in the corresponding position. Next, we systematically scan each entry containing implied pairs and check the cells corresponding to each of these implied pairs. If any one of the implied pairs is incompatible, then the pair implying it must also be incompatible and a cross is placed in the corresponding cell. If none of the cells corresponding to the implied pairs contains a cross, the entry is left unaltered. For instance, consider the entry corresponding to the pair ah that is, column a and row h. In order that this pair be compatible, the pair bg also must be compatible. Referring to the cell bg, we find that it is an incompatible. Hence we conclude that ah also is an incompatible and place a cross in the corresponding cell. In a similar manner, we place crosses in the ef and gh cells. In general, these crosses might generate further crosses in a recursive manner. The procedure naturally terminates when we cannot cross out any more cells. All the uncrossed cells correspond to compatible pairs. The states contained in each pair (say, s_i s_j) are indistinguishable in so far as the external behaviour of the machine is concerned. In other words, for any applicable input sequence, the two output sequences obtained, one with s_i as the starting state and the other with s_j as the initial state, will be identical whenever both are specified.

Table 9.4 *An Incomplete Sequential Machine M_1*

Present State	Next State, Output, Input			
	0	1	2	3
a	--	b, --	--	--, 1
b	a, --	--	c, 0	--
c	--	--	--	d, 1
d	b, --	a, --	b, --	f, 0
e	c, --	c, --	a, --	--, 0
f	--, 0	b, --	--	h, 1
g	--, 1	f, --	e, 1	d, 1
h	--, 1	g, --	--	e, 1

Table 9.5 *Implication Chart for the Machine of Table 9.4*

b	√						
c	√	√					
d	✗	ab bc	✗				
e	✗	ac	✗	ab ac bc			
f	√	√	~~dh~~	✗	✗		
g	bf	✗	√	✗	✗	✗	
h	~~bg~~	√	de	✗	✗	✗	~~f g b e~~
↑ States →	a	b	c	d	e	f	g
MCs	{abc,	abf,	acg,	bch,	bde }		

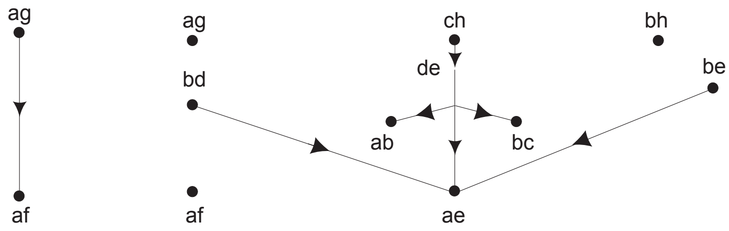


Fig. 9.2 Compatibility Graph for the machine of Table 9.4

The implication chart is a very useful aid in deriving the **maximal compatibles (MCs).** The procedure for obtaining MCs is given below.

1. Start in the rightmost column of the implication chart and proceed left until a column containing a compatible pair is encountered. List all the compatible pairs in that column. This step yields (de).
2. Proceed left to the next column containing at least one compatible pair. If the state to which this column corresponds is compatible with all members of some previously determined compatible, add this state to that compatible to form a larger compatible. If the state is not compatible with all members of a previously determined compatible, but is compatible with some members of such a compatible, then form a new compatible which includes those members and the state in question. Next, list all compatible pairs which are not included in any previously derived compatible.
3. Repeat step 2 until all columns have been considered. The final set of compatibles constitutes the set of maximal compatibles (MCs).

Applying the procedure to the machine M_1 of Table 9.4 whose implication chart is given in Table 9.5 yields the following sequence of compatibility classes.

Column	d	(de)
Column	c	(cg) (ch) (de)
Column	b	(bch) (bde) (bf) (cg)
Column	a	(abc) (abf) (acg) (bch) (bde)

From column b, it is evident that the state b is compatible with states c and h. Consequently, the compatible (ch) is enlarged to (bch). For a similar reason (de) is enlarged to (bde). The compatible (cg) is retained as it is, because b is not compatible with g and (bc) is already included in one other compatible. Since (bf) is not included in any other compatible, it is added to the list. After similar scanning of column a, the final list obtained is the set of **maximal compatibles.** Obtaining the MCs serves a useful purpose. The total number of MCs is an upper bound on the number of states in the **minimal covering machine.** This follows from the fact that every compatible must be a member of at least one MC. Thus, the minimal covering machine for our example cannot have more than five states. A lower bound on the number of states in the minimal covering machine is clearly the minimal number of MCs necessary to cover all the states of the machine. (This minimal set may or may not be closed.)

For our machine the set consisting of the MCs abf, acg, bch, bde is the minimal set covering all the states. Hence, the minimal covering machine must have at least four states. Let us explore, just for the sake of curiosity, whether we can construct a covering machine with this set of four MCs. Let us designate the states of the covering machine, each corresponding to an MC as follows.

$$\alpha = \text{abf} \quad ; \quad \beta = \text{acg}$$

$$\gamma = \text{bch} \quad ; \quad \varepsilon = \text{bde}$$

This would mean that the states a, b and f have to be **merged** to form a single state α; the states a, c and g have to be merged to form another state β and so on. The machine obtained after those mergers is shown in Table 9.6.

Table 9.6 *A Wrong Minimal Covering Machine for the Machine M_1 of Table 9.4*

Present State	Next State, Output, Input			
	0	1	2	3
abf = α	α/β, 0	$\alpha/\beta/$ε, --	β/γ, 0	γ, 1
acg = β	--, 1	α , --	ε, 1	ε, 1
bch = γ	α/β, 1	β, --	β/γ, 0	ε, 1
bde = ε	?, --	β, --	?, 0	α, 0

Consider the state α in Table 9.6 which covers the states a, b, and f of Table 9.4. On the application of the input symbol 0, the next states are --, a, and -- respectively. So, in the covering row, the next state should be one covering the state a. The state a can be covered by either of the states α, β. Thus we have an option here and the corresponding next-state entry in row α and column 0 of Table 9.6 may be filled as either α or β written as α/β. The outputs for this case are --, --,0; (see Table 9.4) we must enter a 0 to cover these outputs.

As another illustration consider the entry corresponding to row ε and input column 1 of Table 9.6. The state ε is supposed to cover the states b, d, and e of Table 9.4. On the application of the input symbol 1, the next states are --, a, and c. This would mean the next state entry has to be a state covering both the states a and c; this can be accomplished only by the state β in Table 9.6. Hence the corresponding entry is written as β. All the next outputs are optional in this case and hence the entry is optional in Table 9.6. Similarly, all the entries in Table 9.6 except those marked with a question mark, can be filled without any difficulty.

Let us now focus our attention on the entry corresponding to row ε, column 0 in Table 9.6. The state ε is supposed to cover the states b, d, and e. From Table 9.4 we observe that the corresponding next states are a, b, and c respectively. Hence, the next state entry has to be a state which covers all the states a, b, and c. Alas! we do not have such a state in our collection. What is to be done? An identical situation is encountered in trying to fill the next-state entry corresponding to row ε, column 2. To circumvent this predicament we now recall that there was an MC containing the states a, b, and c, which we did not select in our effort to keep the number of states low in Table 9.6. Looking back at Table 9.4 it is easy to realise that this situation arose because of the fact that the compatible bde implies the compatible abc. If we wish to select the compatible bde, we have to include the compatible abc as well in our collection; otherwise the collection will not satisfy the **closure property.** To be able to complete Table 9.6, it is now necessary to introduce one more row corresponding to a state (call it σ) which covers the states a, b, and c. This yields a five-state covering machine whose state table (not shown) may be prepared in a straightforward manner. This discussion illustrates the important point that the selected set of compatibles must not only cover all the states of the machine but be closed as well.

In the preceding paragraphs, we have seen that a five-state closed cover for the machine of Table 9.4, using the MCs, exists. Can we do any better? It may be possible to obtain a four-state closed cover (corresponding to the lower bound) if we consider the subsets of MCs also. But then the process requires a listing of all the subsets of MCs together with their closure class sets (Definition 9.8) and then selecting the smallest possible set of compatibles which is closed and covers all the states of the machine. This is a big task by any estimate even for machines of moderate size unless we have some systematic way of doing the job. It is here that the **compatibility graph** of Kohavi (same as Unger's **implication graph** but for some minor differences) plays a useful role. In the implication graph, an arc from a vertex (s_i $\mathbf{s_j}$) to a vertex ($\mathbf{s_p}$ $\mathbf{s_q}$) is removed if there is a longer path from the former vertex to the latter vertex. This implication can as well be effected in the compatibility graph also without changing its properties. For our discussion we will use the name **compatibility graph.**

The **compatibility graph** is a directed graph whose vertices correspond to all compatible pairs and an arc leads from vertex (s_i s_j) to vertex (s_p s_q) if and only if (s_i s_j) implies (s_p s_q). The **compatibility graph** is often used as a tool in minimising **incompletely specified sequential machines.** The compatibility graph for the machine of Table 9.4 is given in Fig. 9.2. This is no more than a pictorial representation of the information contained in the implication chart of Table 9.5. We now search on this graph for a minimal closed cover for the machine. Whenever we select a compatible, all the compatibles implied by it must also be selected. We may, of course, group some pairs to form a larger compatible. For example, the pairs bd, de, and be may be grouped to form a single compatible bde. Every set selected has to be checked for **closure** by reference to the flow table. Finally by a trial and error approach we have to arrive at the **minimal closed cover.**

The following notions are useful in the search for a minimal closed cover.

The order of a compatible is the number of states of the machine contained in it; the **rank of a compatible** is the order of the highest order compatible implied by it. The **image of a maximal compatible** (see Pager in References) is the set of the next states reached on the application of an input symbol starting from the states contained in the MC. An MC has one image per input column. Obviously, every image is a compatibility class.

Inadequacy of Compatibility Graph

The objective of this section is to focus attention on the pitfalls encountered in using the compatibility graph.

Let us now search for a minimal closed cover in Fig. 9.2. While searching in the compatibility graph for a minimal closed cover for the machine of Table 9.4, one may, unfortunately, end up by selecting the following set of compatibility classes as the solution.

{abf, acg, bch, bde} Set (1)

The above set as arrived at from the compatibility graph, apparently closed, is, in fact, not closed as can be verified by checking against the flow table of the machine. In the preceding paragraphs, recall that we had tried in vain to construct a minimal covering machine using these compatibles. This is so because the compatible bde (of rank 3) implies the compatible abc and unless abc is included as it is or is a subset of some other compatible of the set, it cannot be closed. It is not enough if ab, ac, and bc are contained each in a different compatibility class. Thus, it is seen that the compatibility graph is only a means but not the end to determine **closure.** This inadequacy is due to the fact that it shows the implications between compatible pairs but obscures the vital information about the implied compatibles containing more than two states. To list the implications of compatibles of rank greater than two, it is absolutely necessary to make a reference to the original state stable.

One may argue that if we bunch the compatibles bd, de and be to form a single compatible bde, this would imply the union of the implied compatibles, that is, abc. This is not in general true as is demonstrated by the segment of a flow table given in Table 9.7.

Table 9.7 *A Small Segment of a Flow Table*

Present State	Next State, Output Inputs		
	0	**1**	**2**
a	d	--	e
b	e	d	--
c	--	f	f
. .	. .	. .	. .
. .	. .	. .	. .
. .	. .	. .	. .

From Table 9.7 it is seen that ab implies de, bc implies df, and ac implies ef but the single bunched compatible abc does not imply def.

Further search on the compatibility graph of Fig. 9.2 yields another set comprising the compatibles de, abf, acg, and bch which constitutes the only minimal closed cover for the machine under consideration.

Bunching Graph

Let us now consider another machine for which the flow table is given in Table 9.8, the implication chart is given in Table 9.9, and Fig. 9.3 shows the compatibility graph. In Table 9.10 are shown only the images of MCs which are of order greater than two. Notice that the states contained in each image are written in the same order of their correspondence with the states of the MC. From this table, we find that the compatible acef implies the compatible efg and bdgh implies acef. The latter implication yields four more bunching conditions corresponding to the three-member subsets of bdgh. All these bunching constraints are pictorially represented by the bunching graph shown in Fig. 9.4.

Table 9.8 *Flow Table of an Incompletely Specified Sequential Machine M_2*

Present State	Next State, Output Input			
	0	1	2	3
a	--	g, 0	e, 1	d, --
b	a, --	d, --	--	--, 0
c	c, --	--, 0	--	g, 1
d	e, 0	--	a, --	--
e	--, 1	f, --	--, 1	--, 1
f	--, 1	e, --	a, 1	--, 1
g	f, --	--, 1	b, --	h, --
h	c, --	--	a, 0	--

Table 9.9 *Implication Chart for the Machine of Table 9.8*

b	dg						
c	dg	X					
d	ae	ae	ce				
e	fg	X	√	X			
f	eg ae	X	√	X	√		
g	X	af	X	ef ab	√	ab	
h	X	ac	√	ce	X	X	cf ab
	a	b	c	d	e	f	g
MCs {acef, abd, acd, bdgh, cdh, efg}							

The bunching graph and compatibility graph together contain all the information about the compatibles and their implications in a more elegant and easily recognisable manner than the original flow table. Once these two graphs are drawn, we do not need to refer to the flow table for checking the closure of the selected sets of compatibles. As a matter of fact, in many examples, the bunching graph helps us to converge quickly in the search for a **minimal closed cover.** For instance, we observe from Fig. 9.4 that the bunching of the

Table 9.10 *Images of MCs of the Machine of Table 9.8*

Maximal Compatibles	Images Input Columns			
	0	1	2	3
abd	--	--	--	--
acd	--	--	--	--
acef	--	g-fe	--	--
bdgh	aefc	--	--	--
cdh	--	--	--	--
efg	--	--	--	--

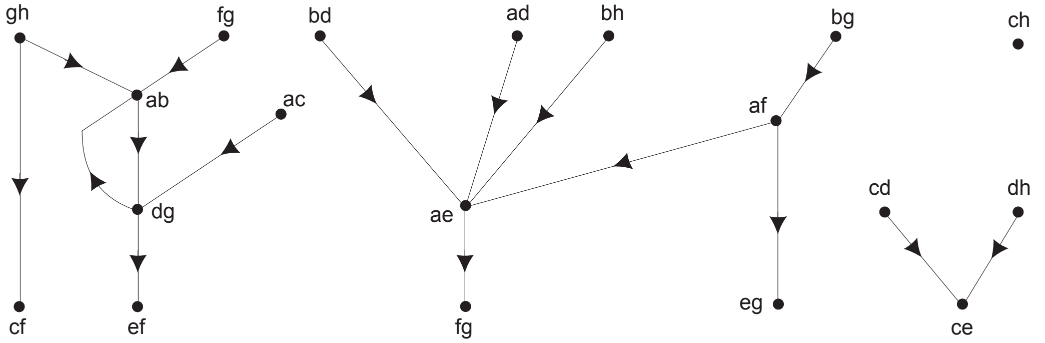


Fig. 9.3 Compatibility graph of the Machine M_2

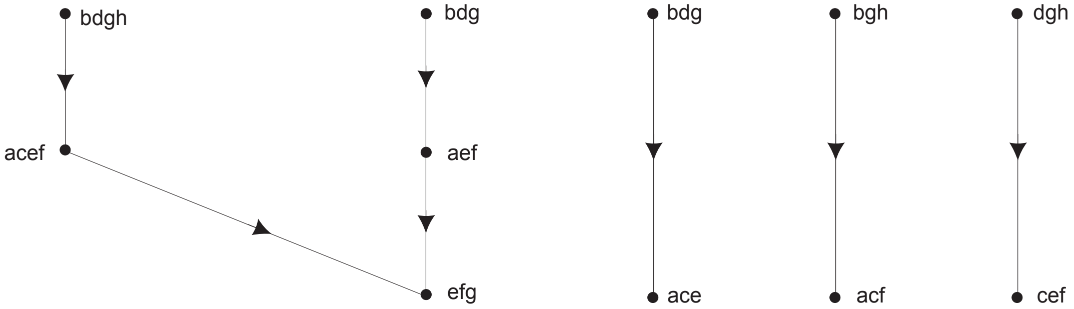


Fig. 9.4 Bunching graph of the machine M_2

compatibles dg, dh, and gh into a single compatible dgh implies the bunching of ce, cf, and ef into cef. Noting from the compatibility graph of Fig. 9.3 that ce, cf and ef are the terminal nodes and working further to cover all the states of the machine. We converge to the minimal closed cover given by

{ab, cef, dgh} Set (2)

The minimal covering machine with mergers represented by set (2) is given in Table 9.11.

Table 9.11 *Minimal Covering Machine for the Machine M_2*

Present State	Next State, Output Input			
	0	1	2	3
ab = α	α, --	β, 0	γ, 0	β, 0
cef = β	β, 1	β, 0	α, 1	γ, 1
dgh = γ	β, 0	--, 1	α, 0	γ, 1

The very fact that we are able to fill all the entries shows that the selected set is **closed.** This is always the case if we select the set by simultaneous reference to both the **compatibility** and **bunching graphs** which collectively determine **closure** with respect to **implication**.

From the above discussion, it is clear that the **compatibility graph** and the **bunching graph** together are adequate to determine the **minimal closed cover** without cumbersome reference to the flow table to check the closure of every selected set of compatibles. A certain amount of search is, of course, unavoidable.

9.6 GENERATION OF SYMBOLIC COMPATIBLES AND PRIME CLOSED SETS

There is a method to limit the search space by deleting and excluding some compatibles from the original list of compatibles without reducing our chances of finding a **minimal closed cover** from the remaining compatibles.

The conceptual framework is as follows. We first form a set of what are called "**primary compatibles"** which is simply a set of all subsets of **maximal compatibles.** We then investigate systematically whether we will be able to delete some compatibles from the original set without in any way affecting our chances of finding a minimal closed cover. By repeated application of Deletion Theorems 9.1 and 9.2, given next, we may delete a substantial number of primary compatibles. The **primary compatibles** which remain undeleted are called **"basic compatibles".** Thus, we obtain a smaller set of compatibles to deal with.

From among the **basic compatibles,** some are **"excluded",** according to Definition 9.15 and 9.16, given next, making the set still smaller. The compatibles remaining in the set after deletion and exclusion effected in that sequence are called **"symbolic compatibles".** Each symbolic compatible serves as a representative for a subset of primary compatibles, hence this name. The set of symbolic compatibles is usually much smaller than the set of primary compatibles. Theorem 9.3, given in the text, guarantees the existence of at least one minimal closed cover which is a collection of symbolic compatibles only.

Deletion Theorems and Generation of Symbolic Compatibles

In this section, we will deliberately use a particularly simple machine to illustrate all the new concepts. The machine used is the same as the one given in Table 9.8 for which we have already obtained the solution using other methods. Some more simple definitions are required at this stage which help in precise formulation of the Deletion Theorems.

Definition 9.13 A set E_i **dominates** a set E_j denoted by $E_i \geq E_j$ if and only if every member of E_j is a subset of at least one member of E_i. The equality relation holds when E_i and E_j are identical sets, in which case they dominate each other. This definition is a generalisation of the definition of set-inclusion. For example, let E_i = {abc, def, eg, gh} and E_j = {ab, bc, de, df, gh}. The set E_i dominates the set E_j but E_i does not include E_j. It is easily seen that set-inclusion is a special case of set-domination.

Definition 9.14 A **prime closed set** S of compatibles is a closed set of compatibles in which no member of S is a subset of another member and S cannot be decomposed into two subsets (not necessarily disjoint) such that both the subsets are closed. (One of them may be closed.)

Definition 9.15 For a compatible C_i, the implied compatibles noted directly from the columns of the state table (first-level implications) are said to be **primary implications** and the set of primary implications is denoted by P_i.

Definition 9.16 A compatible C_i **excludes** a compatible Cj if $C_i \supset C_j$ and $P_i \le E_j$.

Definition 9.17 A **prime compatible** is a compatible that is not excluded by any other compatible.

The significance of Definitions 9.16 and 9.17 is as follows An excluding compatible can be substituted for the excluded compatible in any closed set of compatibles without affecting the closure or covering properties of the set. This process does not demand any extra closure conditions but might cover some extra states. Clearly there exists at least one minimal closed cover composed of **prime compatibles only.**

Definition 9.18 Every subset of a **maximal compatible** is a **primary compatible.**

Illustration Consider the machine represented by the flow table given in Table 9.8. The corresponding implication chart is given in Table 9.9. The set of maximal compatibles is easily obtained as mentioned in Table 9.10.

$$\{\text{acef, abd, acd, bdgh, cdh, efg}\} \qquad (1)$$

The primary compatibles and their closure class sets are listed in Table 9.12. There are 40 primary compatibles for this machine including eight single-state compatibles.

Definition 9.19 A compatible C_i is said to be an **"implied compatible" (IC)** if each compatible pair $C_j \subseteq C_i$ is a subset of some member of the closure class set of some compatible contained in the set of compatibles under consideration.

Definition 9.20 A compatible C_i is said to be an **"unimplied compatible" (UC)** if none of the subsets of C_i are implied by any compatible contained in the set of compatibles under consideration.

Illustration Observe in Table 9.12 that the compatible acef is implied by the compatible bdgh. We say that acef is an implied compatible and so are all the subsets of acef. It will be seen that each of the compatible pairs ab, ac, ae, af, ce, cf, dg, ef, eg, fg is an implied compatible. These implied compatible pairs are shown in Table 9.13 with an 'I' mark in the corresponding cell. The remaining compatible pairs ad, bd, bg, bh, cd, ch, dh, gh are unimplied compatibles indicated by a 'U' mark in the same table.

Definition 9.21 A **maximal implied/unimplied compatible (MIC/MUC)** is an implied/unimplied compatible which is not a proper subset of any other implied/unimplied compatible.

Illustration From Table 9.13 the maximal implied compatibles are easily obtained in the same manner as for obtaining maximal compatibles. The set of MICs is

$$\{\text{acef, ab, dg, efg}\} \qquad (2)$$

Similarly, the set of MUCs is obtained as

$$\{\text{ad, bdh, bgh, cdh}\} \qquad (3)$$

The MICs and MUCs serve the purpose of referencing in the deletion of primary compatibles, discussed later in this section.

Table 9.12 *Generation of Basic Compatibles*

Sl. No.	Primary Compatible	Closure Class Set		Deletions under Theorems 1 and 2			
		Primary Implications	Others	Iteration No.			
				1	2	3	4
1	a c e f	efg, dg	ab	√		D2	
2	b d g h	acef, ab	efg	D2			
3	a b d	ae, dg	fg, ef	D2			
4	a c d	ce, ae, dg	fg, ef, ab	D2			
5	a c e	fg, dg	ab, ef	√		D2	
6	a c f	eg, ae, dg	fg, ef, ab	√	D2		
7	a e f	efg	ab, dg	√		D2	
8	b d g	aef, ab	efg	D2			
9	b d h	ace	fg, dg, ab, ef	D1			
10	b g h	acf, ab	eg, ae, dg, fg, ef	D1			
11	c d h	ce		D1			
12	c e f	ϕ		*	*	*	*
13	d g h	cef, ab		*	*	*	*
14	e f g	ab	dg	√	√	√	D2
15	a b	dg	ef	*	*	*	*
16	a c	dg	ef, ab	√	D1		
17	a d	ae	fg, ab, dg, ef	D1			
18	a e	fg	ab, dg, ef	√	√	D1	
19	a f	eg, ae	fg, ab, dg, ef	√	D1		
20	b d	ae	fg, ab, dg, ef	D1			
21	b g	af	eg, ae, fg, ab, dg, ef	D1			
22	b h	ac	dg, ef, ab	D1			
23	c d	ce		D1			
24	c e	ϕ		*	*	*	*
25	c f	ϕ		*	*	*	*
26	c h	ϕ		*	*	*	*
27	d g	ef, ab		*	*	*	*
28	d h	ee		*	*	*	*
29	e f	ϕ		*	*	*	*
30	e g	ϕ		*	*	*	*
31	f g	ab	dg, ef	√	√	√	D1
32	g h	cf, ab	dg, ef	D1			
33 to 40	a - h	ϕ (single state compatibles)		*	*	*	*

Table 9.13 *Implied (I) and Unimplied (U) Compatible Pairs*

b	I						
c	I						
d	U	U	U				
e	I		I				
f	I		I		I		
g		U		I	I	I	
h		U	U	U			U
	a	b	c	d	e	f	g

Table 9.14 *Progressive Changes of MICs and MUCs*

Iteration No.	Maximal Implied Compatibles	Maximal Unimplied Compatibles
1	acef, ab, dg, efg	ad, bdh, bgh, cdh
2	ab, ae, cef, dg, efg	acd, af, bdh, bgh, cdh
3	ab, cef, dg, efg	acd, ae, af, bdh, bgh, cdh
4	ab, cef, dg	acd, ae, af, bdh, bgh, cdh eg, fg.

Lemma 9.1 If Ci and Cj are two unimplied compatibles such that $C_j \subset C_i$, then $E_i \geq E_j$.

Proof By virtue of the fact that C_j is a proper subset of C_i, it follows that every element of E_j, except those which are subsets of C_i, must be contained in some element of E_i. Since C_i is an unimplied compatible, E_j cannot contain any subset of C_i. Therefore, each element of E_j must be contained in at least one element of E_1.

Hence, $$E_i \geq E_{j.}$$

Illustration Let C_i = bdh and C_j = bd. Referring to Table 9.12, we note that E_i = {ace, fg, dg, ab, ef} and E_j = {ae, fg, ab, dg, ef}. Every member of E_j is a subset (proper or improper) of some member of E_i that is, $E_i \geq E_j$.

Lemma 9.2 Let C_i be an unimplied compatible and E_i be the closure class set of C_i. Let s be an arbitrary state of the machine not covered in E_i. Then the closure class set E_j of a compatible $C_j \subset C_i$ does not cover the state s.

Proof Since $C_i \supset C_j$ it follows from Definition 9.19 of unimplied compatibles and Lemma 9.1 that $E_i \geq E_j$. It is clear that all the states covered in E_j are covered in E_i also, but the converse is not true. Obviously, a state s not covered in E_i cannot be covered by E_j. Hence, the lemma.

Illustration Let C_i = bgh and C_j = bh. Comparing the closure class sets E_i and E_j from Table 9.12, we find that the state h not covered in E_i is not covered in E_j too.

As a consequence of Lemmas 9.1 and 9.2, Theorem 9.1 follows.

Theorem 9.1 (First Deletion Theorem) The deletion of all compatibles C_i satisfying the following conditions, from the set of primary compatibles, does not preclude finding at least one minimal closed cover from the remaining primary compatibles.

(a) C_i is an unimplied compatible.

(b) The closure class set E_i of C_i covers at least one state $s \in C_i$.

(This theorem is much more powerful than rule 1(a) of Luccio which requires that at least all but one states contained in C_i must be covered in E_i. Moreover, implementation of this theorem is very simple compared to that of Luccio's rule.)

Proof Suppose the compatible C_i is a member of a **minimal closed cover** M. Since C_i is an unimplied compatible, its presence is not required to satisfy any closure conditions. It only serves the purpose of covering the states of the machine. Further, each of the compatibles contained in E_i (closure class set of C_i) must be contained in at least one of the compatibles contained in the set (M-C_i). (M-C_i) denotes the subset yielded on removing the member C_i from the set M. Clearly, (M-C_i) ≥ E_i. Let the compatible C_i be given by $s_1\ s_2 \ldots s_k\ s_{k+1} \ldots s_r$ where s_1 through s_r are contained in the state set of the machine. Without loss of generality, assume that s_1, s_2, ... , s_k are the states not covered in E_i and the remaining states s_{k+1}, ..., s_r are covered in E_i. It is clear that the set (M-C_i) ≥ E_i covers the states $s_{k+1} \ldots s_r$. If C_i is replaced by a compatible C_j comprising the states $s_{1,}$ s_2, ... , s_k in M, it follows from Lemma 1 that the yielded set is still closed and covers the machine. In order to conclude that C_i may be deleted from the set of primary compatibles, it is to be shown that C_j remains undeleted. Recall that $C_j \subset C_i$ and none of the states comprising C_j namely $s_1\ s_2$, ..., s_k are covered in E_i. It follows from Lemma 2 that none of the states contained in C_j will be covered in E_j. Hence, C_j remains undeleted. Thus C_i can be replaced by C_j in any closed cover and hence can be deleted from the set of primary compatibles and ignored in the process of extracting a minimal closed cover. The only restriction on the number of states contained in C_i and covered in E_i is that it should be non-zero. As C_i is arbitrarily chosen, the theorem is proved.

Illustration For the machine of Table 9.8, the primary compatibles and their corresponding closure class sets are listed in Table 9.12. Let C_i be cdh which is an unimplied compatible. E_i = {ce}. E_i covers one state $c \in C_i$. The compatible cdh can be deleted from the list of primary compatibles. In this case, the compatible dh can represent the compatible cdh in any closed cover and the theorem guarantees that the compatible dh remains undeleted in the current list, which fact can be verified by noting the closure class set of the compatible dh.

An important and useful corollary to Theorem 9.1 follows

Corollary Let C_i be an unimplied compatible and E_i be its closure class set. Then C_i is not a member of any minimal closed cover M if E_i covers every state $s \in C_i$.

Proof Suppose C_i is a member of some minimal closed cover M. Since C_i is an unimplied compatible, it does not satisfy any closure conditions of (M-C_i). Therefore, the set (M-C_i) is closed. Further, (M-C_i) ≥ E_i; hence, all the states contained in C_i are covered in (M-C_i). Hence, (M-C_i) represents a closed cover for the machine—thus contradicting the hypothesis that M is minimal. In other words, C_i may be replaced by a "void compatibility class" in any closed cover and hence cannot participate in any minimal closed cover.

The above corollary naturally leads to the following definition.

Definition 9.22 A compatible which can be removed from any closed set of compatibles, without affecting the closure and covering properties of the set, is said to be a **"weak compatible".** Otherwise, it is a **"strong compatible"**.

Illustration Consider again the machine of Table 9.8 and compatible C_i = ad in Table 9.12. C_i is an unimplied compatible and E_i is the set {ae, fg, ab, dg, ef}. Since C_i is an unimplied compatible and E_i covers both the states a and d, the compatible C_i = ad cannot be a member of any minimal closed cover. Such **weak compatibles** can be weeded out and ignored even if, for some reason, we have to find all minimal solutions.

Let us, now, proceed to develop another theorem which is more powerful and, in fact, covers Theorem 9.1. The following definitions help a precise formulation of Theorem 9.2.

Definition 9.23 The **"implied part"** of a compatible C_i , denoted by I_i , is the set of states obtained by the union of all implied compatibles which are subsets of C_i .

Illustration Let C_i be bdgh of Table 9.12. Note also the implied compatibles marked 'I' in Table 9.13. There is only one implied compatible namely, dg which is a subset of C_i. We, therefore, say that I_i = dg.

Definition 9.24 The **"uncovered part"** of a compatible C_i, denoted by N_i, is the set of states contained in C_i and not covered in E_i.

Illustration For the compatible $C_i \equiv$ bdgh, the closure class set E_i = {acef, ab, efg} from Table 9.12. The states d and h contained in C_i are not covered in E_i. Therefore, we say that N_i = dh.

Theorem 9.2 (Second Deletion Theorem) The deletion of all compatibles C_i satisfying the condition $(I_i \cup N_i) \subset C_i$ from the set of primary compatibles does not preclude finding at least one minimal closed cover from the remaining primary compatibles. Notice that the union of the implied part and uncovered part should be a **proper** subset of C_i for its deletion.

Proof Suppose C_i is a member of a minimal closed cover M. Let C_j be a compatible given by $C_J = (I_i \cup N_i) \subset C_i$. If C_i is now replaced by C_j in M, then the closure conditions imposed by (M-C_i) are clearly satisfied as $C_J \supseteq I_i$. Since (M-C_i) $\geq E_i$, it follows that the set yielded on replacing C_i by C_j in M is closed. Further, the states contained in C_i and not covered in E_i are now contained in C_j. Hence, coverage also is satisfied. In order to conclude that C_i can be deleted from the list of primary compatibles, it is to be shown that C_j does remain undeleted.

We know that C_j is a proper subset of C_i. Normally, we expect $E_i \geq E_j$. But there is one situation when this is not obeyed. Remember that the closure class set of a compatible does not contain the subsets of the same compatible. Suppose E_j contains a compatible which is a subset of C_i but not of C_j. Then the above condition may be violated. In our case, all the implications of C_j are also in I_i. In other words, E_j is contained in I_i and hence in C_i and C_j too. The question of a compatible of E_j being contained in C_i but not in C_j, does not arise. Hence, E_i dominates E_j. A little reflection reveals that $I_j = I_i$. The states not covered in E_i cannot be covered in E_j since $E_i \geq E_j$ Hence, $N_j = N_i$. It clearly follows that for $C_{j,}$ the union $I_j \cup N_j = C_j$, which cannot be deleted.

The essence of Theorem 9.2 is that if at least one state contained in $(C_i - I_i)$ is covered in E_i, then C_i can be deleted from the list of primary compatibles.

As a matter of fact, Theorem 9.1 can be treated as a special case of Theorem 9.2 when I_i is a null set. However, in practice, it is much easier to implement Theorem 9.1 first and Theorem 9.2 next. This is the reason for stating them explicitly. This point is further discussed after the deletion procedure has been stated.

Illustration Referring again to Table 9.12, let C_i = bdgh. Then I_i = dg; N_i = dh and $I_i \cup N_i$ = dgh $\subset C_i$. Therefore, the compatible bdgh can be deleted. In fact, the compatible dgh can represent the compatible bdgh in any closed cover and the theorem guarantees that the compatible dgh remains undeleted in the current list.

We will now illustrate in detail the procedure to obtain a minimal closed cover for the machine of Table 9.8. Table 9.12 gives the list of primary compatibles with their corresponding closure class sets. First, it is noted that the compatibles cef, dgh, ab, ce, ef, ch, dg, dh, ef, eg cannot be deleted from the list as their corresponding closure class sets do not cover any of the states contained in the respective compatibles. This fact is indicated by stars in the corresponding rows. For example, for the compatible dg, the closure class set {ef, ab} does not cover the states, d, g. The states of a compatible covered in its closure class set are underscored to put them in focus.

The deletion of compatibles is achieved recursively as follows. Consider iteration 1 for which MICs and MUCs are given in Table 9.14. None of the MICs or their subsets can be deleted at this stage and this fact is indicated in Table 9.12 by the check mark. Now Theorem 9.1 is applied to all MUCs and their subsets; as a consequence, the compatibles bdh, bgh, cdh, ad, bd, bg, bh, cd, gh are deleted, indicated by D1. The remaining compatibles are now subjected to the test of Theorem 9.2 and consequently, the compatibles bdgh, abd, acd, bdg are deleted, indicated by D2. This completes the first iteration.

The above deletions of compatibles may result in some of the implied compatibles becoming unimplied compatibles. For example, after the deletions in the first iteration, the compatibles ac and af which were implied compatibles earlier, now become unimplied compatibles. The updated MICs and MUCs are given in Table 9.14. With the new list, the procedure is repeated. The procedure terminates when the set of MICs and hence MUCs remains unaltered. For the machine under consideration, the deletion procedure terminates after four iterations.

One point needs to be noted here. The frequency of iteration may be doubled by iterating alternately on the application of Theorem 9.1 and Theorem 9.2. This is the reason for stating the two theorems explicitly. The frequency of iteration may be increased still further by starting a fresh iteration whenever a primary compatible is deleted; however, the returns of such extra work may not be attractive in many examples.

The compatibles that remain after the deletions in Table 9.12 are called **basic compatibles.**

Definition 9.25 **A basic compatible** is a primary compatible which cannot be deleted by the application of Deletion Theorems 9.1 and 9.2.

For our machine, starting from 40 primary compatibles, we have so far **deleted** 22 of them, yielding 18 **basic compatibles** listed in Table 9.15, with their corresponding closure class sets. Among these, eight more compatibles are now **excluded** in accordance with Definition 9.15 indicated in Table 9.15.

Obviously, an excluded compatible can be represented by a compatible excluding it because this process does not impose any extra closure conditions but may cover some more states of the machine. The compatibles remaining after the elimination of **excluded basic compatibles** are called **symbolic compatibles.**

Definition 9.26 A **symbolic compatible** is a basic compatible which is not excluded by any other basic compatible.

For our machine, the symbolic compatibles, 10 in all, are listed in Table 9.16 together with their corresponding closure class sets. The compatibles represented by each symbolic compatible are also listed in Table 9.16.

It is interesting to note that a primary compatible may be represented by more than one symbolic compatible. For example, the compatible ac may be represented either by the compatible cef or ch in any closed cover. In this example, this situation arose due to the fact that the compatible ac can be represented by the single state compatible c which is excluded by two compatibles cef and ch. Another interesting point to be noted is that the compatibles acd, ad, ae, af, bd, bg, and fg are not represented by any symbolic compatible. These are weak compatibles, discussed earlier, and as such they cannot participate in any minimal closed cover in accordance with the corollary to Theorem 9.1. Hence they do not require any representation.

Table 9.15 *Generation of Symbolic Compatibles from Basic Compatibles*

Sl. No.	Basic Compatible	Closure Class Set		Excluding Compatibles
		Primary Implications	Others	
1	c e f	∅		
2	d g h	c e f, a b		
3	a b	d g	e f	
4	c e	∅		c e f
5	c f	∅		c e f
6	c h	∅		
7	d g	e f, a b		
8	d h	c e		
9	e f	∅		c e f
10	e g	∅		
11	a	∅		
12	b	∅		
13	c	∅		c e f, c h
14	d	∅		
15	e	∅		c e f
16	f	∅		c e f
17	g	∅		e g
18	h	∅		c h

Table 9.16 *Symbolic Compatibles and the Prime Closed Sets Generated by them*

Symbolic Compatible C_i	States Contained in C_i	Closure Class Set	Primary Compatibles Represented by Ci	Prime Closed Sets
C1	cef	∅	acef, ace, acf, aef, cef, efg, ac, ce, cf, ef, ef, c, e, f	C1
C2	dgh	cef, ab	bdgh, dgh	C1 C2 C3
C3	ab	dg, ef	abd, ab	C1 C2 C3; C1 C3 C5
C4	ch	∅	bdh, bgh, ac, bh, ch, gh, c, h	C4
C5	dg	ef, ab	bdg, dg	C1 C3 C5
C6	dh	c e	cdh, dh	C1 C6
C7	eg	∅	eg, e, g	C7
C8	a	∅	a	C8
C9	b	∅	b	C9
C10	d	∅	cd, d	C10

Now we need consider only the relatively small set of symbolic compatibles to search for a minimal closed cover for the machine. The following theorem guarantees that we do not miss the solution.

Theorem 9.3 There exists a minimal closed cover which is a collection of **symbolic compatibles** only.

Proof Let M be a minimal closed cover and suppose that a compatible C_i, which is not a symbolic compatible, is a member of M. Then C_i must be either a basic compatible or not a basic compatible. If C_i is a basic compatible, then there exists a symbolic compatible C_j which excludes C_i and C_i can be replaced by C_j in M. If C_i is not a basic compatible, then it must be a primary compatible which can be deleted under Theorem 9.1 or Theorem 9.2, in which case there exists a symbolic compatible C_k which can represent C_i in M. Hence, the theorem.

Following the methods presented next, prime closed sets of symbolic compatibles are easily obtained. They are listed in Table 9.16. Now the problem is only that of coverage and the minimal closed cover is easily obtained as the set of symbolic compatibles C1, C2, C3 given by

$$\{cef, dgh, ab\} \tag{4}$$

The results of this section are summarised in the procedure given below.

Procedure Generation of symbolic compatibles

1. From the flow table, obtain the implication chart and find all **maximal compatibles.**
2. Prepare a table listing the **primary compatibles** (all proper and improper subsets of maximal compatibles) and their corresponding **primary implications.**
3. In the table obtained in step 2, for each primary compatible, augment the list of primary implications with all implied compatibles obtained by repeated use of the transitivity of implication. The primary implications together with the additional compatibles thus obtained constitute the **closure class set** for the corresponding primary compatible. For this step, no reference needs be made to the flow table.
4. List all the **implied compatible pairs** and find **MICs.** Similarly, find **MUCs.**
5. Delete from the list each primary compatible C_i which is a subset of some MUC if at least one state contained in C_i is covered in E_i.
6. For each primary compatible C_j which is not a subset of any MIC or MUC, find $I_j \cup N_j$ and delete C_j from the list if $(I_j \cup N_j) \subset C_{j.}$
7. From the reduced list of primary compatibles, form a fresh list of MICs and MUCs. If these are identical with the earlier lists, go to step 8; otherwise replace the earlier lists by the current lists and go to step 5.
8. For each pair of compatibles C_i and C_j such that $C_i \supset C_{j,}$ compare P_i and E_j. Exclude (remove) C_j if $P_i \le E_j$.
9. The compatibles now remaining in the list are all symbolic compatibles.

9.7 REDUCED SYMBOLIC COMPATIBLES

A slightly different mechanism is now introduced to further reduce the number of compatibles we need consider. This is based on the fact that the removal of the excluded basic compatibles from the list may result in some of the implied compatibles becoming unimplied compatibles. It is then quite possible that for some compatibles C_i , $C_j = (I_i \cup N_i)$ may become a proper subset of C_i. If the compatible C_j is also contained in the current list, C_i may well be deleted and C_j can represent C_i in any closed cover. If, however, C_j has already been excluded earlier, we cannot delete C_i but we may substitute C_j for C_i in any closed cover provided $E_i \geq E_j$. This condition arises because such substitution should not demand any extra closure conditions. This concept is stated in the form of Theorem 9.4 below.

Theorem 9.4 (Substitution Theorem) Let S be the current set of compatibles under consideration and let $C_i \in$ S. Further, let $C_j = (I_i \cup N_i) \subset C_{i.}$ Then substitution of C_j for C_i in any closed cover M comprising the compatibles in S does not affect the closure and covering properties of M if $E_i \geq E_j$.

Proof Similar to that of Theorem 9.2. If $C_j \in$ S then C_i may be deleted and C_j can represent C_i in M. If C_j is not in S, then it is necessary to ensure that C_j does not impose any closure conditions not already imposed by C_i and hence the condition $E_i \geq E_j$.

Illustration In order to illustrate the concepts of this section, a second example is presented. Consider the machine represented by Table 9.17. The machine has six maximal compatibles indicated at the bottom of the table. The machine has in all 46 primary compatibles listed in different groups in Tables 9.18, 9.19, and 9.20. The implied compatibles and unimplied compatibles (pairs only) as noted by observing the closure class sets of the primary compatibles are indicated in Table 9.21. Table 9.22 shows the progressive changes of MICs and MUCs with different iterations.

Table 9.18 lists 16 primary compatibles which are deleted. The states of a compatible C_i, which are covered in E_i are underscored. The symbol 'D' in the remarks columns stands for deletion and the numbers 1 and 2 indicates the theorem that led to the deletion. Finally, the compatible written in parentheses represents $(I_i \cup N_i)$ of C_i. In fact, the compatible $C_j = (I_i \cup N_i)$ can represent C_i in any closed cover M without affecting the properties of closure and coverage of M. The compatibles remaining undeleted, are basic compatibles (BCs).

From among the basic compatibles, 13 excluded compatibles are listed in Table 9.19 with the excluding compatibles indicated in parentheses in the remarks column; E stands for 'excluded by'.

Starting with 46 primary compatibles, we have so far deleted 16 and excluded 13. The remaining 17 compatibles are symbolic compatibles listed in Table 9.20.

Now we will explore the possibility of further reducing the number of **closed cover compatible candidates.** This is achieved by subjecting the symbolic compatibles to the test of Theorem 9.4 and making the

Table 9.17 *An Incompletely Specified Sequential Machine*

Present	Next States, Output, Inputs							
State	0	1	2	3	4	5	6	7
a	c, -	d, -	b, 0	c, -	f, 1	--	--	--
b	e, -	c, -	--	--	--	f, 0	h, 0	e, -
c	-, 1	--	c, -	a, -	d, 1	--	--	--
d	--	a, 0	--	--	f, -	f, -	--	--
e	--	-, 1	--	e, 1	f, -	--	g, 1	--
f	a, 0	--	g, 1	c, 0	-, 1	d, -	-	e, 1
g	a, 0	--	-, 1	--	g, -	--	f, -	h, 1
h	e, 0	c, 1	f, 1	--	g, -	--	--	--
	MCs abcd, abdf, ace, bdfg, bfgh, egh							

Table 9.18 *Deleted Primary Compatibles*

PC (C_i)	Closure Class Set (E_i)	Remarks
a b d f	ace, acd, bg; bc, fh, eh, fg	D2 (adf)
b d f g	ae, ac, fh, eh; ce, bc	D2 (dfg)**
a b d	ce, acd; ae, df, bc	D2(ad)
a b f	ace, cd, bg, df, bc, fh, eh, fg	D1(∅)
a d f	ac, bg, bc, ae, fh, eh, ce, fg	D2(df)*
b d f	ae, ac; ce, bc	D2(df)
b d g	ae, ac, fg, fh, eh; ce, df, bc	D2(bg)
b g h	ae, fh, eh; ce, df, fg	D2(bg)
e g h	ae, fg; ce, df	D2(eh)
a b	ce, cd; ae, df	D1(b)
a f	ac, bg; bc, df, ae, fh, eh, ce, fg	D1(∅)
b d	ac; bc, df	D1(Ø)
b f	ae, df; ce	D1(b)
b g	ae, fh, eh; ce, df, fg	D1(b)**
d g	fg; eh	D1(d)
e g	fg; eh	D1(∅)
*Deleted in 2nd iteration **Deleted in 3rd iteration		

Table 9.19 *Excluded Basic Compatibles*

PC (C_i)	Closure Class Set (E_i)	Remarks
a b c	ce, cd, df; ae	E(abcd)
b f g	ae, df, fh, eh; ce	E(bfgh)
b f h	ae, fg, df; ce, eh	E(bfgh)
f g h	ae, eh; ce, df	E(bfgh)
a c	bc, df	E(acd, ace)
f g	eh	E(dfg)
f h	ae, fg; ce, df, eh	E(bfgh)
a	∅	E(ad)
b	∅	E(bc, bh)
c	∅	E(bc)
d	∅	E(ad, df)
f	∅	E(df)
h	∅	E(bf)

substitutions wherever possible. Referring to Table 9.20 we find that the compatible acd may be replaced (substituted) by ac which is in turn excluded by ace. Hence, the symbolic compatible acd may simply be ignored in the process of extracting a minimal closed cover. The compatible ace can represent the compatible acd in any closed cover comprising symbolic compatibles. For a similar reason, cd also may be ignored. The

Table 9.20 *Symbolic Compatibles*

SC (C_i)	Closure Class Set (E_i)	Remarks
$\underline{a}$ b $\underline{c}$ $\underline{d}$	ce, df; ae ...	S(bc)*
b $\underline{f}$ g $\underline{h}$	ae, df, eh; ce	
a $\underline{c}$ $\underline{d}$	bc, df ...	S(ac) E (ace)
$\underline{a}$ $\underline{c}$ e	bc, df	
b $\underline{c}$ $\underline{d}$	ac, df ...	S(bc)
d f g	eh	
a d	∅	
a $\underline{e}$	ce; df	
b c	∅	
b h	∅	
c $\underline{d}$	df ...	S(c) E(bc)
c $\underline{e}$	ae, df	
d f	∅	
e h	fg	
g h	ae; ce, df	
e	∅	
g	∅	
*Substituted in 2nd iteration		

Table 9.21 *Implied (I) and Unimplied (U) Compatibles*

b	U						
c	I	I					
d	I	U	I				
e	I		I				
f	U	U		I			
g		I		U	U	I	
h		U			I	I	U
	a	b	c	d	e	f	g

Table 9.22 *Progressive Changes of MICs and MUCs*

S.No.	Iteration number	MICs	MUCs
1.	In Table 9-18	acd, ace, bc, bg, df eh, fg, fh	abf, bd, bh dg, eg, gh
2.	In Table 9.18 (ad → UC)	ace, bc, bg, cd, df eh, fg, fh	abd, abf, bh dg,eg, gh
3.	In Table 9.18 (ad → UC)	ace, bc, cd, df, eh fg, fh	abd, abf, bgh, bdg, eg
1.	In Table 9-20 (cd, fh → UC's)	ace, bc, df, eh, fg	abd, abf, bdg, bfh, bgh, cd, eg
2.	In Table 9-20 (ac → UC)	ae, bc, ce, df, fg eh	abd, abf, acd, bdg, bfh, bgh, eg

Table 9.23 *Reduced Symbolic Compatibles and Prime Closed Sets*

RSC (C_i)	*States Contained in C_i*	*Closure Class Set (E_i)*	*Prime Closure Sets*
C1	bfgh	ae, df, eh, ce	--
C2	ace	bc, df	C2C6C9,C2C3C6C10
C3	dfg	eh	C3C10
C4	ad	∅	C4
C5	ae	ce, df	C5C8C9
C6	bc	∅	C6
C7	bh	∅	C7
C8	ce	ae, df	C5C8C9
C9	df	∅	C9
C10	eh	fg	C3C10
C11	gh	ae, ce, df	C2C6C9C11
C12	e	∅	C12
C13	g	∅	C13

compatible bcd may be replaced by bc which is already in the current set and hence bcd may be ignored. The removal of the above compatibles enables us to remove abcd in the 2nd iteration of this procedure.

In practice, unlike Theorems 9.1 and 9.2, it is not easy to implement Theorem 9.4 which requires comparison of closure class sets. However, in a computer-aided environment, it is worthwhile using this theorem also as any reduction in the number of compatibles drastically reduces the subsequent computational effort required in extracting a minimal closed cover.

Definition 9.27 The compatibles resulting from the repeated application of substitution and exclusion to the set of symbolic compatibles are called "**reduced symbolic compatibles" (RSCs).**

Theorem 9.5 There exists a minimal closed cover comprising reduced symbolic compatibles only.

Proof Proof follows from Theorems 9.3 and 9.4.

The reduced symbolic compatibles, 13 in all, are listed in Table 9.23. Also listed are the prime closed sets of cardinality, four or less, generated by each RSC. We would already have on hand an upper bound solution, discussed subsequently for this machine, which contains five compatibles. So we should now look for a closed cover which is a collection of four or less compatibles. This is the reason for discarding the prime closed sets of cardinality greater than four and also those of cardinality four which do not cover all the states of the machine. It remains to find a union of prime closed sets which represents a minimal cover for the given machine. The following three solutions, obtained with the help of a covering table, are all minimal.

1. {ace (C2), dfg (c3), bc (C6), eh (C10)}
2. {ace (C2), be (C6), df (C9), gh (C11)}
3. {dfg (C3), ad (C4), bc (C6), eh (C10)}

Several other minimal solutions may exist and may be obtained by further search but the effort may not be worthwhile.

9.8 SYMBOLIC VS PRIME COMPATIBLES

It is useful to compare the cardinality of the set of symbolic compatibles with that of the set of prime compatibles; the effort in generating the former set would be meaningful only if the former does not exceed the latter. Consider, for instance, a compatible C_i which excludes several other compatibles (obviously subsets of C_i). If C_i were deleted prior to the removal of the compatibles excluded by C_i, we may have to consider a larger number of compatibles in the process of minimising the given incompletely specified sequential machine (ISSM) and the set of prime compatibles may be smaller than the set of symbolic compatibles. In such a case, it would be useless to generate symbolic compatibles. Fortunately, this is not the case and Theorem 9.8 of this section establishes that the set of symbolic compatibles is never larger than the set of prime compatibles; in fact, the former is usually much smaller than the latter. The following theorem and lemmas help in precise formulation of Theorem 9.8.

Theorem 9.6 Let C_i, C_j, C_k be three compatibles such that $C_i \supset C_j \supset C_k$.

(i) If C_i excludes C_k, then C_i excludes C_j also.

(ii) If C_i does not exclude C_j, then C_i cannot exclude C_k.

Proof Let E_i, $E_{j,}$ and E_k be the closure class sets of C_i, $C_{j,}$ and C_k, respectively. To prove the first part of the theorem, assume that C_i excludes C_k that is, $E_i \le E_k$. We have to show that $E_i \le E_{j.}$ Suppose $E_i \not\le E_j$. Then there exists a compatible, say C_x , such that C_x is in E_i and C_x is not in E_j. Obviously, $C_x \not\subset C_i$. Hence, $C_x \not\subset C_j$, C_k. Since $E_i \le E_k$, it follows that C_x is a member of E_k. Further, since $C_k \subset C_{j\,,}$ it is clear that C_x must be in E_j which is a contradiction. Therefore, $E_i \le E_{j.}$ Hence, C_i excludes C_j. The second part of the theorem can be proved in a similar way.

Illurtration Consider the ISSM of Table 9.24. The machine has, in all, 49 primary compatibles listed together with their closure class sets in Table 9.25 and Table 9.26. The deleted compatibles and excluded basic compatibles are indicated in Table 9.25 in the same notation used thus far. The symbolic compatibles together with their closure class sets are listed in Table 9.26. Table 9.27 gives the implied and unimplied compatible pairs while Table 9.28 shows the progressive changes in MICs and MUCs. Note that in Table 9.25 the compatible abcd excludes ad and all the subsets of abcd containing ad are henceforth excluded by abcd. By observing the closure class sets, it may be verified that abcd excludes abd and acd also. On the other hand, the compatible efgh (Table 9.25) does not exclude efg (Table 9.26) and hence cannot exclude the subsets of the latter, that is, eg and fg (Table 9.25).

Lemma 9.3 Let C_i be a compatible and let $C_j = (I_i \cup N_i) \subset C_i$ be another compatible in the set of compatibles under consideration. Then $E_i \ge E_j$.

Proof If two compatibles are such that one is a subset of another, say $C_j \subset C_{i\,,}$ then E_j may contain a compatible which may be a subset of C_i but not of C_j. This is the only way in which E_i cannot be greater than E_j. In all other cases, the closure class set of a bigger compatible dominates the closure class set of its subsets.

In this special case, $C_j = (I_i \cup N_i) \subset C_{i.}$ Every element of E_j must be contained in I_i and hence in C_j and C_i . If follows that E_j cannot contain an element which is a subset of C_i but not of C_j. As $C_j \subset C_{i,}$ it follows that $E_j \le E_i$.

Table 9.24 *An Incomplete Sequential Machine*

Present State	Next states, output Inputs							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
a	d, -	-, 0	--	--	c, 1	--	--	--
b	--	c, -	d, 0	--	b, -	--	e, 0	--
c	f, -	--	--	a, -	--	b, 1	--	g, -
d	f, 0	--	g, -	b, 0	--	--	--	-, 1
e	--	-	-	h, -	d, 1	g, -	a, -	--
f	-, 0	--	d, -	-, 0	h, -	--	--	c, -
g	--	h, -	--	--	--	f, 1	--	b, 1
h	e, -	d, 1	-, 1	--	--	-, 1	g, 1	--
MCs: abcd, abdg, adfg, cdh, dfgh, efgh								

Table 9.25 *Deleted Primary Compatibles and Excluded Basic Compatibles*

Compatible	*Closure Class Set*	*Remarks*
a b c $\underline{d}$	d f, d g	D2 (a b c)
e f $\underline{g}$ $\underline{h}$	d h, a g, b c	D2 (e f g)
a $\underline{b}$ $\underline{d}$	d f, d g, b c	D2 (a b)
$\underline{a}$ $\underline{c}$ $\underline{d}$	d f, a b; d g, b c	D1 (∅)
a $\underline{f}$ g	c h, b c; e f, d h	D2 (a g)*
$\underline{b}$ c $\underline{d}$	d g, a b	D2 (b c)
e f $\underline{h}$	d h, a g	D2 (e f)
$\underline{e}$ $\underline{g}$ $\underline{h}$	d h, f g, a g; b c, e f	D1 (∅)
$\underline{f}$ g $\underline{h}$	d h, b c; e f	D2 (f g)
a c	d f; d g	E (a b c)
a $\underline{d}$	d f; d g	D1 (a)
a $\underline{f}$	c h; e f, d h	D1 (a)
b $\underline{d}$	d g	D1 (b)
b g	c h; e f, d h	E (b d g)
$\underline{c}$ d	a b; b c	D1 (d)
d h	e f	E (dgh)
e $\underline{g}$	f g; b c	D1 (e)
f g	b c	E (d f g)
g $\underline{h}$	d h; ef	D1 (g)
a	∅	E (a g)
b	∅	E (b c)
c	∅	E (b c)
d	∅	E (d g)
f	∅	E (f h)
g	∅	E (a g, d g)
h	∅	E (f h)
*Deleted in 2nd iteration		

Table 9.26 *Symbolic Compatibles*

Compatible	Closure Class Set
a b d g	d f, c h, b c; e f, d h
a d f g	c h, b c; e f, d h
a f g h	e f, b c
a b c	d f; d g
a b g	c h, b c; e f, d h
a d f	d g, c h; e f, d h
a d g	d f
b d g	c h; e f, d h
c d h	e f, a b; b c
d f g	b c
d f h	e f, d g
d g h	e f
e f g	d h, b c
a b	b c
a g	Ø
b c	Ø
c h	e f; d h
d f	d g
d g	Ø
e f	d h
e h	a g
f h	Ø
e	Ø

Table 9.27 *Implied (I) and Unimplied (U) Compatible Pairs*

b	I						
c	U	I					
d	U	U	U				
e							
f	U			I	I		
g	I	U		I	U	I	
h			I	I	U	U	U
	a	b	c	d	e	f	g

Table 9.28 *Progressive Changes of MICs and MUCs*

S.No.	Iteration Number	MICs	MUCs
1.	In Table 9.25	a b, a g, b c, c h, d f g, d h, e f	a c d, a f, b d, b g, e g h, f h
2.	In Table 9.25 (fg → UC)	a b, a g, b c, c h, d f, d g, d h, e f	a c d, a f, b d, b g, f g h, e g h

Illustration Let C_i = efgh (Table 9.25). Then I_i = efg, N_i = ef, $C_j = I_i \cup N_i$ = efg (listed in Table 9.26). It may be verified that $E_i > E_j$.

Theorem 9.7 If a compatible C_i excludes the compatible $C_j = (I_i \cup N_i) \subset C_i$, then $E_i = E_j$.

Proof Since C_i excludes C_j, it follows from Definition 9.15 that $E_i \leq E_j$ but Lemma 9.3 asserts that $E_i \geq E_j$. Comparing the two inequalities, we conclude $E_i = E_j$.

Illustration Let C_i = abcd (Table 9.25), then I_i = abc, N_i = abc, $C_j = (I_i \cup N_i)$ = abc (Table 9.26). It may be seen that C_i excludes C_j and $E_i = E_j$.

Lemma 9.4 Let C_i be a compatible which can be deleted and represented by a compatible $C_j = (I_i \cup N_i) \subset C_i$. Let C_x be a compatible such that $C_x \subset C_i$ and $C_x \not\subseteq C_j$. If C_i excludes C_x, then C_x is a compatible which can be deleted.

Proof Let the compatible C_i be the set of states given by

$$C_i = \{s_1, s_2, ..., s_k, s_{k+1}, ..., s_r\}$$

Without loss of generality, assume that the compatible C_j is given by

$$C_j = I_i \cup N_i = \{s_1, s_2, ..., s_k\}$$

This would mean that the states $s_{k+1,}$ s_{k+2}, ..., s_r are covered in E_i and none of these states in conjunction with any of the states contained in C_i is an imphied compatible. Now, consider a compatible C_x which is a proper subset of C_i and not a subset of C_j; that is, C_x contains at least one of the states $s_{k+1,}$ s_{k+2}, ..., $s_{r.}$ Let us arbitrarily choose that the state $s_{k+1} \in C_x$. Obviously s_{k+1} cannot be contained in I_x as it is not contained in $I_{i.}$ If C_i excludes C_x, it follows that $E_x \geq E_i$. Therefore, the state s_{k+1} which is covered in E_i, is covered in E_x too. Hence s_{k+1} must not be contained in N_x. It clearly follows that $I_x \cup N_x \subset C_x$ and C_x can be deleted. Hence, the lemma.

Illustration Let C_i = efgh (Table 9.25). Then I_i = efg, N_i = ef, and $C_j = I_i \cup N_i$ = efg (Table 9.26). Let C_x = egh (Table 9.25). In Table 9.25, the compatible C_i = efgh is deleted and it can be represented by C_j = efg (Table 9.26) in any closed cover. The compatible C_i = efgh excludes C_x = egh as can be verified by comparing their closure class sets. $I_x = \varnothing$, $N_x = \varnothing$ and $I_x \cup N_x = \varnothing$. Therefore, the compatible C_x = egh is deleted and it may be replaced by a void compatibility class in any closed cover. Incidentally, observe that the compatible efh, although not excluded by efgh, is also deleted in Table 9.25.

As another example, observe, in Table 9.25, that the compatibles abd and acd which are excluded by the deleted compatible abcd (represented by abc) are themselves deleted. In addition, the compatible bcd, although not excluded by abcd, also stands deleted.

Lemma 9.5 Let C_i be a compatible which can be deleted and represented by a compatible $C_j = (I_i \cup N_i) \subset C_i$. Let C_x be a compatible such that $C_x \subset C_j$. If C_i excludes C_x then C_j also excludes $C_{x.}$

Proof Since $C_i \supset C_j \supset C_x$, it follows from the hypothesis and Theorem 9.6 that C_i excludes C_j. Furthermore, Theorem 9.7 asserts that $E_i = E_j$. As C_i excludes C_x, we have $E_i \leq E_x$ and so is $E_j \leq E_x$. Thus C_j excludes C_x and the lemma is proved.

Illustration Let C_i = abcd (Table 9.25), $C_j = (I_i \cup N_i)$ = abc (Table 9.26), and C_x = ac (Table 9.25). Note that C_i can be deleted and represented by C_j in any closed cover. The compatible $C_x \subset C_j$, which is excluded by the deleted compatible C_i, is now excluded by its representative C_j. This is easily verified by noting the corresponding closure class sets.

The essence of Lemmas 9.4 and 9.5 is as follows. If a compatible C_i is deleted and represented by a compatible $C_j = (I_i \cup N_i) \subset C_i$, then every compatible C_x excluded by C_i is either deleted or excluded by C_j; it is deleted if $C_x \not\subset C_j$, in accordance with Lemma 9.4 or else it is excluded by C_j, in accordance with Lemma 9.5.

Now, we are in a position to formulate the most important and useful Theorem 9.8.

Theorem 9.8 For any sequential machine, the cardinality of the set of symbolic compatibles does not exceed the cardinality of the set of prime compatibles. (It may be smaller and is, in fact, usually significantly smaller.)

Proof In order to prove the theorem, we have to show that each compatible excluded by a deleted compatible, is either deleted or else excluded by some symbolic compatible. Let C_i be a compatible which is deleted and let $C_j = (I_i \cup N_i) \subset C_i$ be a compatible which can represent C_i in any closed cover. We need consider only those compatibles which are proper subsets of C_i. Let $C_x \subset C_i$ be a compatible excluded by C_i. Two cases arise.

Case (i) $C_x \not\subset C_j$. From Lemma 9.4 it follows that C_x can be deleted.

Case (ii) $C_x \not\subset C_j$. Lemma 9.5 ensures that C_j excludes C_x. Clearly, either C_j is a symbolic compatible or else there exists a symbolic compatible which excludes C_j and hence excludes C_x.

$C_x = C_j$ is a trivial case. The theorem is hence proved.

Illustration Referring to Tables 9.25 and 9.26, we note that the compatibles excluded by abcd are abc, abd, acd, ac, and ad. The compatible abcd is deleted and is represented by the symbolic compatible abc. The compatibles abd, acd, and ad correspond to case (i) above and are deleted. The compatible ac corresponds to case (ii) and is excluded by the symbolic compatible abc. The compatible abc may be viewed as being represented by itself.

The machine of Table 9.24 has, in all, 23 symbolic compatibles, listed in Table 9.26, as against 31 prime compatibles, which may be listed if desired. This substantiates the important fact that the symbolic compatible set is never larger and is in fact, usually, appreciably smaller than the prime compatible set. For the machine of Table 9.24 no further reduction of symbolic compatibles is possible and all the compatibles in Table 9.26 are reduced symbolic compatibles. Theorem 9.8 assures us of at least one minimal closed cover comprising only these compatibles.

9.9 BOUNDS ON MINIMAL CLOSED COVER

Prior knowledge of the upper bound on the number of states in a minimal machine (that is, the initial cardinality of the minimal closed cover) which covers a given incompletely specified sequential machine, greatly helps in reducing the computational work involved in the generation of prime closed sets of compatibles. Likewise, the lower bound tells us that it is futile to try to obtain a covering machine with less number of states than the lower bound. Clearly, the smallest number of maximal compatibles required to cover all the states of the given machine is a lower bound on the number of states in the covering machine.

Existing Upper Bound

Let the upper bound be denoted by u, obtained by actually having on hand a covering machine with u states. Then, in the process of minimisation, we initialise the cardinality of the minimal closed cover to the value u. We need only search for a covering machine with less than u states. As such, all prime closed sets containing u or more compatibles need not be generated. The computational work, perhaps, exponentially increases with the value of u. Thus, if u can be lowered at an early stage in the process, whenever we obtain a closed cover with a cardinality less than u, the value of u will be updated to the lower number thus obtained.

Clearly the bound is given by

u = Minimum of (m, n, v)

where m = number of maximal compatibles

n = number of states of the given machine

v = cardinality of the minimal closed cover obtained using only compatibles with void closure class sets

Min-Max Cover

The essential idea is to find a minimal closed cover comprising only maximal compatibles. For brevity let us call this the "**min-max cover**". For many machines, particularly when the number of maximal compatibles is large, the cardinality of the **min-max cover** may be smaller than u. Let x be the cardinality of the min-max cover. The modified upper bound, denoted by u′ is given by

$$u' = \text{Minimum of } (u, x)$$

Thus, we need to search for a covering machine with u′ – 1 or less number of states. If, however, the number of internal state variables were to be the only criterion for minimising the given machine, we need search for a covering machine with its number of states being 2^k or smaller where k is the integral part of $\log_2(u' - 1)$.

Illustration To illustrate the ideas of this section, let us consider a fairly challenging example of an incomplete sequential machine represented by Table 9.29. This machine is the same as the one used by Unger, but for renaming of the states by letters of the alphabet in place of numerals. The implication chart is given in Table 9.30. The set of maximal compatibles, 10 in all, and their closure class sets are listed in Table 9.31. For this machine, m = 10, n = 9, and v = 6. Therefore, u = 6 and we need search for a covering machine with five or less states since we already have a covering machine with six states.

Now, consider the set of maximal compatibles. We first pick out the state which is contained in the least number of maximal compatibles. If there is a tie, the choice is arbitrary. Note that the state d is contained in two maximal compatibles, M_3 and M_{10}, while each of the remaining states is contained in more than two maximal compatibles. It follows that either M_3 or M_{10} or both must be contained in the min-max cover. So we arbitrarily choose M_3 and generate the prime closed sets of maximal compatibles initiated by M_3, taking full advantage of the fact that those containing six or more MCs may be ignored in the process as we already have a cover with u = 6 compatibles. The prime closed sets of MCs generated by M_3 are

$$\{M_1, M_3, M_4, M_5, M_6\}, \{M_1, M_3, M_4, M_6, M_7\}$$

$$\{M_1, M_3, M_5, M_6, M_8\}, \{M_2, M_3, M_5, M_6, M_8\}$$

Table 9.29 *An Example of an Incomplete Sequential Machine*

Present State	Next states, output Inputs			
	00	**01**	**11**	**10**
A	B, -	D, -	--	C, -
B	F, -	I, -	--	--
C	--	--	G, -	H, -
D	B, -	A, -	F, -	E, -
E	--	--	--	F, -
F	A, 0	--	B, -	-, 1
G	E, 1	B, -	--	--
H	E, -	--	--	A, 0
I	E, -	C, -	--	--

Table 9.30 *Implication Chart for the Machine of Table 9.29*

B	BF DI							
C	CH	√						
D	~~DE~~	~~BF~~ ~~AI~~	~~FG~~ ~~EH~~					
E	CF	√	~~FH~~	EF				
F	AB	AF	BG	AB BF	√			
G	~~BE~~ ~~BD~~	EF BI	√	BE AB	√			
H	BE AC	EF	AH	BE AE	AF		√	
I	~~BE~~ ~~CD~~	EF CI	√	BE AC	√	AE	BC	√
	A	B	C	D	E	F	G	H

MCs BCGHI, BEGHI, DEGHI, ABCF, ABCF, ABEF, ABEH, BCFI, BEFI, DEFI

Table 9.31 *Maximal Compatibles and Their Closure Class Sets for the Machine of Table 9.29*

Maximal Compatible M_i	States Contained in M_i	Closure Class Set of M_i
M_1	BCGHI	EF, AH; BE, AC
M_2	BEGHI	EF, BCI; AF, AB, BF, DI, AC, CH, AH
M_3	DEGHI	BE, ABC, AEF, BF, CH, EF, AH, BG, BI, CI
M_4	ABCF	DI, BG, CH, BE, EF, BI, CI, AH
M_5	ABCH	BEF, DI, AF
M_6	ABEF	DI, CF, AC, CH, AH, BG, BI, CI
M_7	ABEH	BEF, DI, ACF, BG, CH, BI, CI
M_8	BCFI	AEF, BG, AB, BE, DI, AC, CH, AH
M_9	BEFI	AEF, CI, BG, AB, CF, DI, AC, CH, AH
M_{10}	DEFI	ABE, AC, BF, CF, CH, AF, BG, AH, BI, CI

It is easily verified that each of the above sets covers all the states of the machine. So we have a covering machine with five states only. Next, we obtain prime closed sets generated by M_{10} containing less than five MCs. It will be seen that there is no such set. Hence, we conclude that x = 5 and the initialised number of states in the minimal closed cover for the machine is given by

$$u' = \text{Minimum of } (5, 6) = 5$$

Now that we have a covering machine with u′ = 5 states, we need search for one with u′ – 1 = 4 or less number of states. This enables us to ignore all prime closed sets of symbolic compatibles with a cardinality greater than four. The process obviously converges to the solution much faster with the use of the reduced upper bound and the cost we pay is a little extra computation restricted to MCs only, which is negligible when compared with the amount of work saved.

Upper Bound Using MICs

We have seen that the first stage in the generation of symbolic compatibles is to list all the compatibles together with their closure class sets and then find the initial set of maximal implied compatibles. Clearly, a minimal set of MCs which dominates (Definition 9.13) the initial set of MICs and covers all the states of the given ISSM has to represent a closed cover for the machine and hence yields an upper bound. This follows from the fact that such a set contains in it all possible closure conditions. To find such a set we may adopt the following procedure.

I Determine the set of MCs and the initial set of MICs.

II Augment the set of MICs obtained in Step I with additional singleton compatibles which represent the states of the machine not covered in the set of MICs.

III Find a minimal set of MCs which dominates the set obtained in Step II. This is a covering problem which we normally encounter in the minimisation of combinational logic functions; the only difference is that each MIC is to be considered as one inseparable unit.

The set of MCs obtained in Step III is a closed cover for the given machine and hence its cardinality is an upper bound on the number of compatibles in the minimal closed cover. We will now illustrate these ideas with the help of the machines of Tables 9.8 and 9.24.

Illustration Consider the machine of Table 9.8 whose MCs are given in Table 9.9 and the initial set of MICs is given in Table 9.14.

I The set of MCs: {acef, abd, acd, bdgh, cdh, efg}

Initial set of MICs: {acef, ab, dg, efg}

II We note that the state h is not covered in the set of MICs. Therefore, Augmented set of MICs: {acef, ab, dg, efg, h}

III We notice that each of the MICs acef and efg is itself an MC; the MIC ab is included only in one MC abd and the MIC dg is included only in one MC bdgh. Replacing the MICs by the corresponding MCs and striking out the singleton h as it is now contained in another member, we obtain a closed cover for the machine as

{acef, abd, bdgh, efg}

which contains four compatibles.

We may now construct a machine with four states by effecting mergers indicated by the set (4) which covers the machine of Table 9.8. Now that we have a covering machine with four states, we need to search for a covering machine with three or less states. Subsequently, we do not have to generate prime closed sets of symbolic compatibles whose cardinality is four or more.

As another illustration consider the machine of Table 9.24 whose MCs are given at the bottom of Table and its MICs are given in Table 9.28.

Set of MCs: {abcd, abdg, adfg, cdh, dfgh, efgh}

Initial set of MICs: {ab, ag, bc, ch, dfg, dh, ef}

We first note that the MIC bc is included in only one MC abcd; the MIC ch is included only in one MC cdh; and the MIC ef is included only in one MC efgh. After making these substitutions in the set of MICs we may strike off the MICs ab and dh as they are included in other members of the set. Only one more MC adfg is required to include both the remaining MICs ag and dfg. Thus we arrive at the minimal set of MCs.

{adfg, abcd, cdh, efgh}

which dominates over the initial set of MICs. For big machines, this set may be determined systematically with the aid of a covering table and remembering that each of the MIC's has to be treated as a single inseparable unit. The above set indicates that we can now construct a machine with four states which covers the machine of Table 9.24. Hence, we may ignore all prime closed sets of symbolic compatibles with cardinality four or more in the process of extracting a minimal closed cover.

9.10 PRIME CLOSED SET—AN ALTERNATIVE DEFINITION AND TESTING PROCEDURE

Prime closed set (PCS) is defined as follows. (Also see Definition 9.13 of Section 9.6.)

Definition 9.28 A set of compatibles S is said to be a prime closed set if it satisfies the following conditions.

(i) S is closed with respect to the binary relation "implication".

(ii) No member of S is a subset of another member of S.

(iii) There exists at least one member (called the generating compatible) $C_g \in S$ such that for every non-void compatible $C_j \in S$ and $C_j \neq C_g$, $(S - C_j)$ is not closed.

Definition 9.29 Every member C_g which satisfies condition (iii) of Definition 9.27 is called a "generating compatible" of the corresponding prime closed set S. There may be more than one generating compatible in a PCS. In other words, the same PCS may be generated by more than one compatible. Further, one compatible may generate more than one PCS.

Condition (iii) of Definition 9.28 says that we cannot remove any member other than the generating compatible C_g such that the yielded set is closed. To put it more precisely, all the compatibles other than C_g have to be necessarily retained in the set in order to satisfy the closure conditions of $C_{g.}$ Note that $(S - C_g)$ may or may not be closed. A little reflection reveals that if a PCS contains two or more generating compatibles, then no single member of the PCS can be removed without violating the closure property of the yielded subset.

***Illustration* 1** Let C_1, C_2, C_3, C_4 be four distinct compatibles of some ISSM such that no one of them is a subset of another.

Let the implications (closure conditions) be given by

C_1 implies C_2; C_2 implies C_3

C_3 implies Ø; C_4 implies C_2

The PCS generated by C_1 is given by

$$S_1 = C_1\ C_2\ C_3$$

Clearly, removal of any compatible other than C_1 from S_1 yields a subset which is not closed.

Let us consider another set of compatibles given by

$$S_2 = C_1\ C_2\ C_3\ C_4$$

From this set, remove C_1 to yield $(S_2 - C_1) = C_2\ C_3\ C_4$. Since $(S_2 - C_1)$ is closed, C_1 must be a generating compatible or else S_2 is not a PCS. (Note that C_1 is not implied by any other compatible in S_2 and hence no other compatible can be a generating compatible.) Now find the subsets yielded on removing each of the compatibles other than C_1 and test for closure. We observe that $(S_2 - C_4)$ is closed, thus violating condition (iii) of Definition 9.28. Hence, S_2 is not a PCS. Likewise, $C_2\ C_3$ and C_3 alone also constitute PCSs. Thus, unlike prime implicants, a PCS may be a subset of another PCS.

***Illustration* 2** Consider another example in which C_x, C_y, C_z are the compatibles under consideration. Let their implications be given by

C_x implies C_y AND C_z

C_y implies Ø (null set)

C_z implies C_x

It is easily seen that the PCS $C_x\ C_y\ C_z$ is generated by two compatibles C_x and C_z. In this PCS, no single compatible can be removed without upsetting the closure property of the yielded set.

***Illustration* 3** Consider yet another example given by

C_a implies C_b OR C_c, AND C_d

C_b implies Ø

C_c implies Ø

C_d implies Ø

The compatible C_a generates two PCSs $C_a\ C_b\ C_d$ and $C_a\ C_c\ C_d$. That the set $S = C_a\ C_b\ C_c\ C_d$ is not a PCS can be seen by following the argument similar to that of Illustration 1.

Before we depart from this section, let us take a second look at the condition (iii) of Definition 9.27 from a new angle. Suppose we define "Decomposition of a Set" as follows

Definition 9.30 A "decomposition" of a set S is a collection of subsets (not necessarily mutually disjoint) whose set union is S.

Then the condition (iii) of Definition 9.28 amounts to saying that the set S cannot be decomposed into two subsets such that both the subsets are closed. This leads to another form of Definition 9.28 given earlier as Definition 9.14 in Section 9.6, which is reproduced below.

"A prime closed set S is a closed set of compatibles in which no member of S is a subset of another member of S and the set S cannot be decomposed into two proper subsets such that both the subsets are closed (one of them may be closed)."

However, if we were to test whether a given set of compatibles is prime or not, it is too cumbersome to examine all possible decomposed subset pairs for closure. It is much easier and more systematic to use Definition 9.28, and the following procedure is proposed.

Procedure for Testing Prime Closed Sets

I Check conditions (i) and (ii) of Definition 9.28. This is trivial.

II From the given set S select arbitrarily one compatible, say $C_i \in S$, and test $(S - C_i)$ for closure. If $(S - C_i)$ is not closed for all i, then S is a PCS. If some $C_g \in S$ is found such that $(S - C_g)$ is closed, go to step III.

III Test $(S - C_i)$ for all the remaining $C_i \neq C_g$ for closure. If none of them are closed, S is a PCS; otherwise it is not a PCS.

Procedure for Generation of All Prime Closed Sets

I List all the compatibles to be considered for extracting the minimal closed cover, together with their closure class sets. (In our procedure these are symbolic compatibles.) With each compatible C_i, associate an uncomplemented two-valued (0,1) Boolean variable C_i.

II The rule of vanishing products. Identify every pair of compatibles C_i, C_j such that $C_i \subset C_j$ and set the logical product $C_i . C_j = 0$. (This is necessary to ensure condition (ii) of Definition 9.28 of PCS. Furthermore, this step significantly reduces the computational work involved.)

III Consider each compatible C_g as a generating compatible and form a Boolean product of sums as follows. C_g itself is one sum. For each member of the closure class set of C_g, there is a logical sum of all compatibles which contain this member. Underline C_g to indicate that its closure conditions have been included in the expression.

IV Obtain a minimal sum of products form for the Boolean expression on hand bearing in mind the vanishing products of step II and using an additional rule $\underline{x} \,.\, x = \underline{x}$.

V For each literal not underlined, take into account the corresponding closure conditions as in step III and underline such literals. Iterate on steps IV and V until the final sum of products form in which all literals are underlined is obtained. In this expression, each product corresponds to a prime closed set of compatibles generated by the compatible C_g.

VI Repeat steps III, IV, and V starting each time with a new compatible in the set under consideration until the PCSs generated by all the compatibles are obtained.

***Illustration* 4** Some ISSM has the following compatibles and implications.

I C_1 = abc implies de

C_2 = ab implies bc

C_3 = bc implies $\varnothing$

C_4 = def implies ab

C_5 = de implies ef

C_6 = ef implies $\varnothing$

II We set $C_1 C_2 = C_1 C_3 = C_4 C_5 = C_4 C_6 = 0$.

III Let us find the prime closed sets generated by C_1. As its closure class set contains de which is, in turn, a subset of C_4 and C_5, the initial Boolean expression is given by the product of sums as.

$$F_1 = C_1 (C_4 + C_5)$$

IV The minimal sum of products form for F_1 is given by

$$F_1 = C_1 C_4 + C_1 C_5$$

V We take into account the closure conditions of the literals not underlined in the above expression namely, C_4 in the first product and C_5 in the second product, and write the resulting expression for F_1 as

$$F_1 = C_1 C_4 (C_1 + C_2) + C_1 C_5 (C_4 + C_6)$$

VI By simple manipulation using Boolean algebra and remembering the additional rule $\underline{x} . x = \underline{x}$, we get

$$F_1 = C_1 C_4 + C_1 C_5 C_6$$

VII Taking into account the closure conditions of C_6 in the second product we get

$$F_1 = C_1 C_4 + C_1 C_5 C_6$$

Thus the PCSs generated by C_1 are.

$$\{ABC (C_1), DEF (C_4)\} \qquad (1)$$

$$\{ABC (C_1), DE (C_5), EF (C_6)\} \qquad (2)$$

Notice that we did not have to use the rule of vanishing products in generating the PCSs for C_1. Let us now generate the PCSs for C_2.

$$\begin{aligned} F_2 &= C_2 (C_1 + C_3) \\ &= C_1 C_2 + C_2 C_3 \\ &= C_2 C_3 \text{ using the rule of vanishing products} \\ &= C_2 C_3 \end{aligned}$$

Thus C_2 generates only one PCS given by

$$\{AB (C_2), BC (C_3)\}$$

Had we not used the rule of vanishing products, we would have got the following expressions for the compatible C_2.

$$\begin{aligned} F_2 &= C_2 (C_1 + C_3) \\ &= C_1 C_2 + C_2 C_3 \\ &= C_1 C_2 (C_4 + C_5) + C_2 C_3 \\ &= C_1 C_2 C_4 + C_1 C_2 C_5 + C_2 C_3 \\ &= C_1 C_2 C_4 (C_1 + C_2) + C_1 C_2 C_5 (C_4 + C_6) + C_2 C_3 \\ &= C_1 C_2 C_4 + C_1 C_2 C_5 C_6 + C_2 C_3 \\ &= C_1 C_2 C_4 + C_1 C_2 C_5 C_6 + C_2 C_3 \end{aligned}$$

The above expression corresponds to the following sets of compatibles.

$$\{ABC\ (C_1),\ AB\ (C_2),\ DEF\ (C_4)\} \tag{3}$$

$$\{ABC\ (C_1),\ AB\ (C_2),\ DE\ (C_5),\ EF\ (C_6)\} \tag{4}$$

$$\{AB\ (C_2),\ BC\ (C_3)\} \tag{5}$$

Only set (5) is prime. The sets (3) and (4) although closed, are not prime closed sets as each of them contains both C_1 and $C_2 \subset C_1$. In fact, it can be easily shown that for every non-prime closed set S there will be a PCS or a union of PCSs which covers S. For instance, PCS (1) covers set (3) in the sense that PCS (1) is a subset of set (3) and yet covers all the states covered in (3). Similarly, PCS (2) covers set (4). Thus it is clear that the rule of vanishing products comes in handy to prevent such a situation.

Another more powerful concept is introduced now. For this purpose we need to invoke Definition 9.12, which is reproduced below for ready reference.

"A set S_i dominates a set S_j denoted by $S_i \geq S_j$ if and only if every member of S_j is a subset of at least one member of S_i. The equality relation holds when S_i and S_j are identical sets in which case they dominate each other."

***Illustration* 5** Consider the sets of compatibles given by

$$S_1 = \{ABC, DEF\}$$

$$S_2 = \{AB, BC, DE, EF\}$$

The set S_1 dominates S_2 written as $S_1 \geq S_2$. If S_1 and S_2 are two PCSs, then it is clear that S_1 can represent S_2 in any closed cover for the given ISSM. Further, since the cardinality of S_1 is less than that of S_2, the latter may be declared as redundant and ignored in the process of extracting a minimal closed cover.

A definition of a **redundant PCS** which covers the above points is given below.

Definition 9.31 A **prime closed set** S_j is "redundant" and may be ignored in the process of extracting a minimal closed cover if any one of the following conditions is satisfied.

(i) $K(S_j) \geq u$ where $K(S_j)$ denotes the cardinality of the set S_j and u is the upper bound on the cardinality of the minimal closed cover found by actually constructing a solution.

(ii) $K(S_j) = u\text{-}1$ and S_j does not cover all the states of the given ISSM. This follows from the fact that we are now looking for a covering machine with u-1 or less states.

(iii) There exists another PCS $S_i \subset S_j$ such that S_i and S_j cover the same states of the machine.

(iv) There exists another PCS S_i such that $S_i > S_j$ and $K(S_i) \leq K(S_j)$.

The conditions of the above definition may be used even if, for some reason, we wish to find all minimal closed covers. Whereas conditions (i) and (ii) may be implemented while generating the prime closed sets, it is far more simpler to implement conditions (iii) and (iv) while working with the covering table discussed in subsequent paragraphs.

9.11 SIMPLIFICATION OF PRIME CLOSED SET COVERING TABLES

The last stage in the reduction of an incompletely specified sequential machine is to find a collection of prime closed sets which covers all the states of the machine and yet contains the minimal number of distinct compatibles. This step involves consideration of **Prime Closed Set (PCS)** covering tables. Although PCS covering tables appear similar to **Prime Implicant (PI)** covering tables, there are certain important structural differences between the two as a consequence of which the former do not obey the same set of rules as the latter. In this chapter the differences between the PCS and PI covering tables are brought to focus and a set of rules is proposed for the simplification of the PCS covering tables. Also included is a mechanism by which a few symbolic compatibles may be ignored in the process of extracting a minimal closed cover.

Prime Closed Set Covering Table

The PCS covering table will now be illustrated by means of an example. The flow table of an ISSM is given in Table 9.32. The symbolic compatibles and their prime closed sets as arrived at by using the procedures developed thus far, are listed in Table 9.33. Although a compatible is a set of states of the machine and a PCS is a set of compatibles, the set parentheses are understood and are usually omitted. For a listing of deleted and excluded compatibles, see Appendix C.

In order to find a minimal closed cover for the machine, we have to find a union of prime closed sets which covers all the states of the machine and yet contains the minimal number of symbolic compatibles. Table 9.34 shows the PCS covering table in which the row designations are the prime closed sets (ordered in accordance with their cardinalities) and the columns are labelled with the states of the machine. The states covered in a particular PCS are indicated by the letter X in the corresponding columns of the particular row. For example the PCS C_3 C_{10} C_{13} corresponds to the collection of compatibles ADF, BH and FH, which together cover the states A, B, D, F and H.

Prima facie, the differences between a PI covering table and PCS covering table may be listed as follows.

1. In the case of a PI covering table, each row is designated by a PI and no PI is a subset of another PI. As opposed to this, in a PCS covering table, a PCS may be a subset of another PCS. For example C_4 is a subset of C_1 C_4, C_4 C_6 and C_4 C_7; C_3 C_{10} is a subset of C_3 C_{10} C_{13}. The row domination rule (elimination of dominated rows) of PI tables is not applicable to PCS tables for the retention of the dominating PCS in place of a dominated PCS may lead to a solution with larger number of compatibles. One is tempted to conjecture that if a PCS of lower cardinality (number of compatibles in the PCS) dominates a PCS of greater cardinality, then the row corresponding to the latter may be eliminated. That such a conjecture is not valid may be seen by comparing the rows corresponding to PCSs C_1C_4 and C_8 C_9 C_{11} in Table 9.34; the former PCS of cardinality 2 dominates the latter of cardinality 3. If the row designated

Table 9.32 *An Incompletely Specified Sequential Machine*

Present State	Next States, Output Inputs							
	0	1	2	3	4	5	6	7
A	D, 0	--	B, -	--	--	--	--	F, -
B	E, -	A, 1	--	--	--	I, -	--	--
C	F, -	C, -	-, 0	B, -	D, -	--	--	--
D	--	-, 0	-, 1	E, 1	A, -	G, -	--	--
E	-, 1	--	H, 1	--	A, -	-, 0	G, -	--
F	--	-, 0	H, -	E, -	A, -	--	--	F, -
G	H, 1	-, 0	--	-, 0	--	G, 1	E, -	D, 0
H	-, 1	A, -	--	-, 0	--	I, 1	E, -	A, -
I	-, 1	-, 0	--	C, 0	--	H, 1	B, -	D, -
MCs ABC, ACF, ADF, BCH, BE, CFH, CGHI, DEF								

Table 9.33 *Symbolic Compatibles and Prime Closed Sets*

SC (C)	States Contained in C_i	Closure Class Set (E_i)	Prime Closed Sets
C_1	A B C	D E F	C_1 C_4*
C_2	A C F	D F, B H, B E, A D	--
C_3	A D F	B H	C_3 C_{10}
C_4	D E F	∅	C_4
C_5	G H I	A D, B E	C_5 C_8 C_9
C_6	A B	D E	C_4 C_6
C_7	A C	D F	C_4 C_7
C_8	A D	∅	C_8
C_9	B E	∅	C_9
C_{10}	B H	∅	C_{10}
C_{11}	C F	A D, B E	C_8 C_9 C_{11}
C_{12}	E G	F H; A F, B H	--
C_{13}	F H	A F; B H	C_3 C_{10} C_{13}
C_{14}	C	∅	C_{14}
C_{15}	G	∅	C_{15}
C_{16}	I	∅	C_{16}

*As this PCS gives a five-state closed cover, all PCSs of cardinality greater than four and also those of cardinality four which do not cover all the states of the machine are ignored. We should now look for a closed cover comprising four or less compatibles.

Table 9.34 *Prime Closed Set Covering Table*

	States								
PCS	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
C_4				X	X	X			
C_8	X			X					
C_9		X			X				
C_{10}		X						X	
C_{14}			X						
C_{15}							X		
C_{16}									X
C_1 C_4	X	X	X	X	X	X			
C_3 C_{10}	X	X		X		X		X	
C_4 C_6	X	X		X	X	X			
C_4 C_7	X		X	X	X	X			
C_3 C_{10} C_{13}	X	X		X		X		X	
C_5 C_8 C_9	X	X		X	X		X	X	X
C_8 C_9 C_{11}	X	X	X	X	X	X			

by the PCS C_8 C_9 C_{11} were eliminated, we would end up in a solution which is not minimal. That this is so will be subsequently seen by the fact that the machine has a unique minimal closed cover composed of symbolic compatibles C_5 C_8 C_9 $C_{11.}$ Nevertheless, if the states covered by two prime closed sets S_i, S_j are identical and if $S_i \subset S_j$, then the row designated by S_j may well be eliminated in accordance with condition (iii) of Definition 9.31. For example, PCSS C_3 C_{10} and C_3 C_{10} C_{13} cover the same states of the machine namely, a, b, d, f, and h; the latter PCS may be eliminated for it would require an extra compatible C_{13} with no additional coverage. Thus the row domination rule requires to be modified in the case of PCS covering tables.

2. In the case of a PI table, the problem is to find a minimal number of PIs to cover all the true minterms. In contrast to this, the problem with PCS tables is to minimise not the number of prime closed sets but rather the number of compatibles contained in the collection so as to cover all the states of the machine.

There are other important aspects which deserve mention. Firstly, it is obvious that the **column domination rule** (dominating columns may be eliminated) of the PI tables is applicable to PCS tables as well. Secondly, if there are two rows designated by the prime closed sets S_i and S_j such that $S_i \subset S_j$, then a cross in a column in S_j row is redundant if row S_i also has a cross in the same column. For example, consider column A in Table 9.34. The crosses in the rows C_5 C_8 C_9 and C_8 C_9 C_{11} are redundant since there is a cross in the C_8 row. The reason is that we do not choose a larger PCS when its subset can account for the state a. Such a situation does not arise in PI tables as no PI is a subset of another PI.

At this stage a small digression is worthwhile. The rows C4 C6 and C_4 C_7 of Table 9.34 could have been eliminated at the very beginning of writing this table. Observe in Table 9.33 that these prime close sets are generated by the compatibles AB (C6) and AC (C7). Note that the compatible AB (C6) implies the compatible DE. The compatible DE is not available in the set of symbolic compatibles and we have to necessarily account for it by taking the compatible DEF (C4). In other words, the implied compatible DE in the closure class set of AB may be augmented and written as DEF, whence it follows that the compatible ABC (C1) excludes the compatible AB (C6) and the latter may be ignored. For a similar reason, the compatible AC (C7) may also be ignored. We may formalise the above statements as follows. For every implied compatible I_i which is a member of some closure class set, we have to enumerate all symbolic compatibles $C_i \supset I_i$ and augment I_i to $\cap_i C_i$ where $\cap_i$ denotes intersection for all i, and then test for exclusions if any. The simplification resulting from this sophistication may not be commensurate with the extra work done in many examples.

The simplification discussed in the preceding paragraph may well be obtained in a conceptually simpler and more elegant manner from the PCS covering table itself. Let S_i and S_j be two prime closed sets of the same cardinality and let the row S_i dominate over the row S_j. Then, we compare the elements of S_i and S_j. If they differ in one compatible only, they should be of the following form.

$$S_i = \{C_1, C_2, C_3 \ldots C_i, \ldots C_n\}$$

$$S_j = \{ C_1, C_2, C_3 \ldots C_j, \ldots C_n\}$$

We simply compare C_i and C_j and if $C_i \supset C_{j,}$ then the row S_j may be deleted. As a matter of fact, in this situation, the prime closed set S_i dominates over S_j that is, $S_i > S_j$ (recall Definition 9.13). It is important to distinguish between row domination and set domination; while the former is merely concerned with the crosses in the rows (corresponding to the states covered), the latter is a more stringent requirement that every member of S_j should be a subset of some member of S_i. Set domination implies row domination but the converse is not true. Furthermore, since $K(S_i) \le K(S_j)$, condition (iv) of Definition 9.31 is satisfied and the prime closed set S_j becomes redundant and may be ignored in the process of extracting a minimal closed cover. It is easily verified by reference to Tables 9.33 and 9.34 that the prime closed set C1 C4 dominates over the prime closed sets C4 C6 and C4 C7 and further, the cardinalities being equal, both the latter PCSs are redundant and may be ignored.

Simplification Procedure and Solution of Prime Closed Set Covering Tables

The discussion of this Section leads to the following simplification rules and procedure. Rules 1 and 2 below correspond to the conditions (iii) and (iv) of Definition 9.31.

Simplification rules

Rule 1 If S_i and S_j denote two prime closed sets covering the same states and if $S_i \subset S_j$, then PCS S_j is redundant and the corresponding row may be ignored.

Rule 2 Let S_i, S_j designate two prime closed sets such that $S_i > S_j$, and $K(S_i) \leq K(S_j)$. Then S_j is redundant and the corresponding row may be ignored.

Rule 3 If a column x dominates a column y, then column x is redundant.

Rule 4 In a column, a cross in row S_j is redundant if there exists a row designated by $S_i \subset S_j$ which contains a cross in the same column.

Simplification procedure

I Delete the row S_j if a subset row $S_i \subset S_j$ includes all the states covered by S_j. Referring to Table 9.34 we find that the row $C_3\,C_{10}$ includes all the states covered by the row $C_3\,C_{10}\,C_{13}$ and hence the latter row is deleted.

II Compare every pair of rows. Let the rows be designated after the corresponding prime closed sets as S_i and S_j. If row S_i dominates over row S_j, then check the cardinalities K (S_i) and K (S_j) of the PCSs. If $K(S_i) \leq K(S_j)$, then check the set domination relationship. If $S_i > S_{j,}$ then delete the row S_j. In Table 9.34 the row $C_1\,C_4$ dominates the rows $C_4\,C_6$ and row $C_4\,C_7$. Since the cardinality of the PCS corresponding to the dominating row does not exceed that of the PCSs corresponding to the dominated rows, we now check for the set-domination relationship. Referring to Table 9.33, we find that PCS $C_1\,C_4$ dominates PCSs $C_4\,C_6$ and $C_4\,C_7$. Hence, the rows in Table 9.34 corresponding to PCSs $C_4\,C_6$ and $C_4\,C_7$ are deleted.

III Delete from the PCS covering table each column which dominates another column. In Table 9.34, column B dominates column H and column D dominates columns A and F. Hence, columns B and D are deleted.

IV In each column, delete a cross in a row if a subset row contains a cross in the same column. In column A the crosses in rows $C_5\,C_8\,C_9$ and $C_8\,C_9\,C_{11}$ are deleted as there is a cross in row C_8 in the same column. Likewise, in column E, all the crosses except those in rows C_4 and C_9 are deleted; in column F, the crosses in rows $C_1\,C_4$, $C_4\,C_6$, and $C_4\,C_7$ are deleted, and in column H, the cross in $C_3\,C_{10}$ row is deleted.

The simplified PCS covering table after the above deletions is given in Table 9.35. To find the minimal closed cover, a Boolean expression is now formed as in PI tables. The compatibles are treated as Boolean variables and a product of sums expression in which each sum accounts for covering a state is written down as follows. Notice that the last sum is the product of three sums corresponding to states G, H, I.

$$F = (C_8 + C_1\,C_4 + C_3\,C_{10})\,(C_{14} + C_1\,C_4 + C_8\,C_9\,C_{11})\,(C_4 + C_9)$$
$$(C_4 + C_3\,C_{10} + C_8\,C_9\,C_{11})\,(\,C_{10}\,C_{15}\,C_{16} + C_5\,C_8\,C_9) \quad (1)$$

After some simple manipulation and keeping in mind the upper bound on the cardinality of the minimal closed cover, Eqn. (1) simplifies to $F = C_5\,C_8\,C_9\,C_{11}$ + (Terms containing more than four literals)

Hence, the set of compatibles C_5, C_8, C_9, C_{11} constitutes the minimal closed cover for the machine of Table 9.32, given by

$$\{GHI, AD, BE, EF\}$$

Table 9.35 *Simplified PCS Covering Table*

	States						
PCS	**A**	**C**	**E**	**F**	**G**	**H**	**I**
C_4			X	X			
C_8	X						
C_9			X				
C_{10}						X	
C_{14}		X					
C_{15}					X		
C_{16}							X
C_1 C_4	X	X					
C_3 C_{10}	X			X			
C_5 C_8 C_9					X	X	X
C_8 C_9 C_{11}		X		X			

Summary

It is shown that the prime closed set (PCS) covering tables are structurally different from the prime implicant (PI) covering tables. The differences between PCS and PI covering tables are brought to focus and discussed. A set of rules is proposed for the simplification of the PCS covering tables. A new mechanism by which a few symbolic compatibles may be ignored in the process of extracting a minimal closed cover is also mentioned.

9.12 MINIMAL CLOSED COVER

In order to extract a minimal closed cover for a given ISSM, we first obtain the set of symbolic compatibles following the procedure elaborated in Section 9.6. The last step requires exclusion of some basic compatibles. For this purpose, we have to compare each pair of compatibles C_i, C_j such that $C_i \supset C_j$ and their implications. The number of comparisons to be made can be appreciably reduced if we classify the compatibles according to their **"order"** and **"rank"** defined below.

Definition 9.32 The **order** of a compatible denoted by n is the number of states contained in it. For example, the order of the compatible dgh in Table 9.15 is 3.

Definition 9.33 The **rank** of a compatible, denoted by r, is the order of the highest order compatible (s) contained in its closure class set. For example, the rank of the compatible dgh in Table 9.15 is 3 as it implies the compatible cef of order 3.

The following theorems follow as a consequence of the above definition.

Theorem 9.9 The rank of a compatible cannot exceed its order.

Theorem 9.10 A compatible C_i of rank r_i cannot exclude a compatible C_j of lower rank $r_j < r_i$.

Illustration Referring to Table 9.15, let C_i = dgh and C_j = dg. We have P_i = {cef, ab}, and E_j = {ef, ab}.

Therefore, $r_i = 3$ and $r_j = 2$. The compatible C_i does not exclude the compatible C_j. Even if E_j were to be {ce, cf, ef, ab}, the compatible C_i would not exclude C_j as E_j does not contain the third order compatible cef. We, therefore, conclude that it is futile to make further comparisons once we recognise the fact that $r_i > r_j$. In order to exclude a compatible C_j we need compare E_j with P_i of another compatible C_i if and only if $C_i \supset C_j$ and $r_i \leq r_j$. Thus, listing the basic compatibles according to their order as well as rank will reduce the number of comparisons to be made.

Another technique using Theorem 9.6 of Section 9.8 comes in handy to constrain the process of exclusion. We first try to exclude some lowest order compatible, say C_k, with the highest order compatible, say C_i, such that $C_i \supset C_k$. If C_i excludes C_k, then all compatibles C_j satisfying the property $C_i \supset C_j \supset C_k$ are henceforth excluded in accordance with Theorem 9.6; no further comparisons of closure conditions are necessary (see the illustration under Theorem 9.6 of Section 9.8).

Generation of Prime Closed Sets of Symbolic Compatibles and Obtaining a Minimal Closed Cover

Consider the machine of Table 9.29 (also copied as Table B.1 of Appendix B). The implication chart and the MCs are given in Table 9.30 (also copied as Table B.2). The machine has a total of 97 primary compatibles listed in Table B.3 together with their closure class sets. The learner is advised to solve this example completely in order to appreciate and get a feel for the techniques learnt in this chapter. Following the procedures for generation of symbolic compatibles given in Section 9.6, 12 compatibles are deleted in step 5, 13 are deleted in step 6, and 28 compatibles are excluded in step 8. Thus, 53 compatibles in all may be ignored and we have to deal with the remaining 44 symbolic compatibles only in the search for a minimal closed cover. Even this is a formidable task for hand computation unless we have a fairly efficient technique which converges fast to a solution.

For the machine of Table B.1, the set of symbolic compatibles denoted in parentheses by C_i, $1 \leq i \leq 44$ is given below.

{BCGHI (C_1), ABCF (C_2), ABCH (C_3), ABEF(C_4), ABEH (C_5), BCFI (C_6),
BCGI (C_7), BEFI (C_8), BEGI (C_9), BGHI (C_{10}), CGHI (C_{11}), DEFI (C_{12}),
ABC (C_{13}), ABF (C_{14}), ACH (C_{15}), AEH (C_{16}), BEF (C_{17}), BEG (C_{18}),
BEI (C_{19}), BGH (C_{20}), BHI (C_{21}), CFI (C_{22}), CGH (C_{23}), CGI (C_{24}),
CHI (C_{25}), DGI (C_{26}), EGI (C_{27}), GHI (C_{28}), AE (C_{29}), BC (C_{30}), BE (C_{31}),
BH (C_{32}), CF (C_{33}), CG (C_{34}), CI (C_{35}), DH (C_{36}), DI (C_{37}), EF (C_{38}),
EG (C_{39}), EI (C_{40}), GH (C_{41}), HI (C_{42}), A (C_{43}), D (C_{44})} (1)

Observing the implications of the above symbolic compatibles from Table B.3, we get a set of implied compatibles listed in Table 9.36. In Table 9.36, for each implied compatible, all the symbolic compatibles in which the implied compatible is contained are also listed. For example, the implied compatible ABC is a subset of ABCF (C_2), ABCH (C_3), and ABC (C_{13}). For convenience in referencing, let us call this table the **“inclusion table”.** The symbolic compatible, which occurs most in Table 9.36, contains in it the largest number of implied compatibles. If only we could eliminate such compatibles early enough, Table 9.36 gets simplified. Hence we form prime closed sets of symbolic compatibles in the order of their frequency of

Table 9.36 *Inclusion Table for the Machine of Table B.1 Giving Implied Compatibles C_α and Symbolic Compatibles C_β such that $C_\alpha \subseteq C_\beta$*

Implied Compatible C_α	**All Symbolic Compatibles $C\beta \geq C\alpha$**
ABC	C_2, C_3, C_{13}
ABE	C_4, C_5
ACF	C_2
AEF	C_4
BCI	C_1, C_6, C_7
BEF	C_4, C_8, C_{17}
AB	C_2, C_3, C_4, C_5, C_{13}, C_{14}
AC	C_2, C_3, C_{13}, C_{15}
AE	C_4, C_5, C_{16}, C_{29}
AF	C_2, C_4, C_{14}
AH	C_3, C_5, C_{15}, C_{16}
BC	C_1, C_2, C_3, C_6, C_7, C_{13}, C_{30}
BE	C_4, C_5, C_8, C_8, C_{17}, C_{18}, C_{19}, C_{31}
BF	C_2, C_4, C_6, C_8 C_{14}, C_{17}
BG	C_1, C_7, C_9, C_{10}, C_{18}, C_{20}
BI	C_1, C_6, C_7, C_8, C_9, C_{10}, C_{19}, C_{21}
CF	C_2, C_6, C_{22}, C_{33}
CH	C_1, C_3, C_{11}, C_{15}, C_{23}, C_{25}
CI	C_1, C_6, C_7, C_{11}, C_{22}, C_{24}, C_{25}, C_{35}
DI	C_{12}, C_{26}, C_{37}
EF	C_4, C_8, C_{12}, C_{17}, C_{38}

occurrence in the inclusion table 9.36. The compatible C_4 = ABEF occurs most often (nine times). So, we will now find prime closed sets generated by C_4 ignoring in the process all those containing five or more compatibles since the initialised cardinality of the minimal closed cover is now 5 as discussed in Section 9.9. The closure class set E_4 of C_4 = ABEF, noted from Table B.3, is

$$E_4 = \{DI, CF, AC, CH, AH, BG, BI, CI\} \qquad (2)$$

Now, we obtain a Boolean expression in product of sums form, one sum corresponding to each compatible contained in E_4 together with C_4 itself as one sum.

For example, to account for the compatible DI, we must have one of the symbolic compatibles C_{12}, C_{26}, C_{37} noted from Table 9.36. Thus the initial expression will be as follows. The Cs should be considered as Boolean variables in the expression.

$$(C_4)\,(C_{12} + C_{26} + C_{37})\,(C_2 + C_6 + C_{22} + C_{33})\,(C_2 + C_3 + C_{13} + C_{15})$$
$$(C_1 + C_3 + C_{11} + C_{15} + C_{23} + C_{25})\,(C_3 + C_5 + C_{15} + C_{16})$$

$$(C_1 + C_7 + C_9 + C_{10} + C_{18} + C_{20})\ (C_1 + C_6 + C_7 + C_8 + C_9 + C_{10} + C_{19} + C_{21})$$
$$(C_1 + C_6 + C_7 + C_{11} + C_{22} + C_{24} + C_{25} + C_{35}) \qquad (3)$$

In expression (3), C_4 has been underlined to indicate that the closure conditions of C_4 have been taken care of. Noting that the 1st, 2nd, 3rd, 6th and 7th sums do not contain any common terms, we conclude immediately that all the prime closed sets generated by C_4 contain five or more symbolic compatibles. Hence C_4 can be eliminated from set (1) and Table 9.36. As a consequence of striking off C_4 from Table 9.36, we observe that the compatible AEF is no longer contained in any symbolic compatible. Therefore, all those symbolic compatibles, which imply the compatible AEF cannot be members of any covering machine with less than five states and may be eliminated. Thus C_6 and C_8, each of which implies AEF (see also Table B.3), are also eliminated along with C_4.

Next, we generate prime closed sets starting from $C_{2,}$ which occurs eight times in Table 9.36.

$$E_2 = \{DI, BG, CH, BE, EF, BI, CI, AH\} \qquad (4)$$

The initial Boolean product of sums is obtained as follows

$$(C_2)\ (C_{12} + C_{26} + C_{37})\ (C_1 + C_7 + C_9 + C_{10} + C_{18} + C_{20})$$
$$(C_1 + C_3 + C_{11} + C_{15} + C_{23} + C_{25})\ (C_5 + C_9 + C_{17} + C_{18} + C_{19} + C_{31})$$
$$(C_{12} + C_{17} + C_{38})\ (C_1 + C_7 + C_9 + C_{10} + C_{19} + C_{21})$$
$$(C_1 + C_7 + C_{11} + C_{22} + C_{24} + C_{25} + C_{35})\ (C_3 + C_5 + C_{15} + C_{16}) \qquad (5)$$

In expression (5), we first observe that the 1st, 2nd, 3rd, and the 9th sums do not contain any common literals and their product yields all four-literal terms. As we want to ignore all terms with five or more literals, we may ignore in the remaining sums any literal which has not already occurred in the above four sums.

After removing the redundancies we obtain expression (6) given below. Notice that the 6th sum becomes C_{12} and the 8th sum becomes $(C_1 + C_7)$ and so on.

$$(C_2)\ (C_{12})\ (C_1 + C_7)\ (C_3 + C_5 + C_{15} + C_{16})\ (C_5 + C_9 + C_{18})\ (C_1 + C_3 + C_{15}) \qquad (6)$$

Again we observe that the first four sums in expression (6) yield all four-literal terms and hence ignore C_9, C_{18} in the 5th sum and the expression simplifies to

$$(C_2)\ (C_5)\ (C_{12})\ (C_1 + C_7)\ (C_1 + C_3 + C_{15}) \qquad (7)$$

Now, ignoring C_3 and C_{15} in the last sum we obtain

$$C_2\ C_1\ C_5\ C_{12} \qquad (8)$$

We have to now take into account the closure conditions of the literals not underlined in the above expression. The expression obtained after simplification using the rules of Boolean algebra and an additional rule $\underline{x}\ .\ x = \underline{x}$ is

$$C_2\ C_1\ C_5\ C_{12}\ (C_8 + C_{17}) \qquad (9)$$

in which each term contains five literals. We, therefore, conclude that the compatible C_2 cannot generate any prime closed set with less than five symbolic compatibles and hence ignore C_2. Thus, we eliminate C_2 from set (1) and Table 9.36. As a consequence, the compatible ACF will not be contained in any symbolic compatible and hence C_5 and C_{16}, each of which implies ACF, are also eliminated. The elimination of C_5 from Table 9.36 in turn results in the elimination of C_{12} in a similar way.

By now the efficacy of the method and the utility of Table 9.36 must be clear. Every elimination of a compatible from the inclusion table drastically reduces the computational work involved in the generation of prime closed sets for the remaining compatibles. Furthermore, elimination of one compatible may give rise to further eliminations in a chain without any extra work. For example, we have so far generated prime closed sets for C_4 and C_2 only but eliminated not only C_2, C_4 but C_5, C_6, C_8, C_{12}, C_{16} also without any extra computation. The key to reduction of computational work is that we must score out at every stage maximum possible entries in Table 9.36. Another technique also can be used to achieve this purpose. Once we establish that a symbolic compatible C_i implies another symbolic compatible C_j, we may cross out C_i from every row of Table 9.36 containing C_j. For example, at a later stage, we find that each of the symbolic compatibles C_9, C_{10}, C_{18}, and C_{20} implies C_7 and we may cross out the former compatibles from the BG row and BI row of Table 9.36.

After we have eliminated C_2 and with it C_5, C_{12}, C_{16}, we proceed to compute the prime closed sets for C_1 which occurs six times in Table 9.36. It will be seen that each of the prime closed sets generated by C_1 contains four symbolic compatibles. None of them cover all the states of the machine. Hence, we eliminate C_1 also.

Next we examine $C_{3,}$ which occurs six times in Table 9.36 and find that it generates two prime closed sets, each containing four symbolic compatibles. The working for C_3 is shown below.

$$C_3 = \text{ABCH} \tag{10}$$

$$E_3 = \{\text{BEF, DI, AF}\} \tag{11}$$

$$\begin{aligned}&(C_3)\,(C_8 + C_{17})\,(C_{12} + C_{26} + C_{37})\,(C_{14})\\ &= C_3\,C_8\,C_{12}\,C_{14} + C_3\,C_8\,C_{14}\,C_{26} + C_3\,C_8\,C_{14}\,C_{37}\\ &\quad + C_3\,C_{12}\,C_{14}\,C_{17} + C_3\,C_{14}\,C_{17}\,C_{26} + C_3\,C_{14}\,C_{17}\,C_{37}\end{aligned} \tag{12}$$

For each term in expression (12) we now take into account the closure conditions imposed by all the literals not underlined. The logical expressions thus obtained are simplified using Boolean algebra and an additional rule $\underline{x} \,.\, x = x$.

Ignoring the terms containing five or more literals, the final expression for C_3 is obtained in the next step only as

$$C_3\,C_{14}\,C_{17}\,C_{26} + C_3\,C_{14}\,C_{17}\,C_{37} \tag{13}$$

Now we have two prime closed sets in (13). The set given by

$$\{\text{ABCH }(C_3),\ \text{ABF }(C_{14}),\ \text{BEF }(C_{17}),\ \text{DGI }(C_{26})\} \tag{14}$$

further covers all the states of the given machine and hence constitutes a covering machine with four states only. The second prime closed set indicated in (13) does not cover the state G. Hence, it is discarded.

Now that we have a covering machine with four states only, the new value of u′ is 4; for the remaining symbolic compatibles, we need to generate prime closed sets containing three or less compatibles only. This drastically reduces the computational work. It is easily verified that no other prime closed set or union of prime closed sets containing three or less compatibles covers all the states of the machine. Hence set (14) is a minimal closed cover for the machine of Table 9.29 (same as Table B.1 of Appendix B).

The entire procedure for obtaining a minimal closed cover is summarised below. Although the procedure is stated in terms of symbolic compatibles, it is also applicable to the set of reduced symbolic compatibles or any other set of compatibles chosen for the extraction of a minimal closed cover.

Procedure for Finding a Minimal Closed Cover

1. Obtain the set of **symbolic compatibles** following the procedures described in Section 9.6 and tabulate them along with their **closure class sets.**
2. List all **implied compatibles** in the table obtained in step 1 and prepare another table showing all symbolic compatibles containing each implied compatible. For convenience in referencing, we call this an **inclusion table.**
3. Let the index u′ represent the initialised number of states in a minimal covering machine. Find the value of u′ as discussed in section 9.9.
4. Count the number of times each symbolic compatible occurs in the **inclusion table** and perform the following steps selecting compatibles in the order of their frequency of occurrence in the **inclusion table.**
5. Compute **prime closed sets** generated by the **symbolic compatible** selected using the procedure described in Section 9.10. Ignore in the process all **prime closed sets** containing u′ or more compatibles. If no more remain, eliminate the compatible and all compatibles implying it recursively. Repeat this step selecting the next compatible in order until all **symbolic compatibles** are examined.
6. Test every **prime closed set** containing u′– 1 **symbolic compatibles** and ignore if it does not cover all the states of the machine. If no more remain, eliminate the symbolic compatible and all those implying it recursively from the table of step 1 and the inclusion table. Proceed to generate prime closed sets for the next compatible in order, beginning from step 5.

   If a prime closed set containing u′–1 symbolic compatibles and covering all the states of the machine is found, it constitutes the current solution. Record it. Decrease the value of u′ by 1 and repeat this step.

   Record the remaining prime closed sets, if any, and go to step 5 choosing the next compatible in order until all **symbolic compatibles** have been examined.
7. Find a **minimal cover** using the **prime closed sets** recorded with the procedure described in Section 9.11. If this contains less than u′ compatibles, it is a minimal closed cover for the machine. Otherwise the initial solution or the solution found in step 6, if any, whichever contains less number of compatibles, represents the minimal closed cover.

All Minimal Closed Covers

We have, so far, discussed only the problem of finding one minimal closed cover. If, for some reason, all minimal closed covers are to be found, we may adopt the following procedure.

We may delete only the weak compatibles from the list of primary compatibles because weak compatibles can never participate in any minimal closed cover according to the corollary to Theorem 9.1. Further, if an excluded compatible has an identical closure class set with that of one of the excluding compatibles, it has to be retained in the list. This, in effect, means that we have to deal with a larger number of compatibles and it is the price we pay to obtain all minimal closed covers.

Alternatively, we have to generate all prime closed sets of symbolic compatibles containing u′ compatibles until we find a closed cover with less number of compatibles when the value of u′ is updated. This procedure yields all solutions, each of which is a collection of symbolic compatibles only. **Then we have to investigate the representations of each symbolic compatible.** For instance, the symbolic compatible DGI

(C_{26}) of the machine of Table 9.29 (also B.1) represents the primary compatible DEGI. If DGI is now replaced by DEGI and/or ABF is similarly replaced by AF in the minimal closed cover given by set (14), the yielded sets are still closed and cover the machine. Thus we obtain four different solutions. Following this reasoning, one may list more than a dozen minimal solutions for the machine of Table 9.17 of Section 9.7.

SUMMARY

After a brief discussion of limitations of sequential machines and Mealy and Moore machines, conventional methods of minimising completely specified sequential machines has been discussed.

The reduction of incompletely specified sequential machines using the compatibility graph and the pitfalls therein have been pointed out. Bunching graph overcomes such difficulties has been emphasised.

It is not necessary to consider all the compatibles in the extraction of a minimal closed cover for a given ISSM. It is sufficient to consider only the set of symbolic compatibles which is usually much smaller than the entire set of compatibles. Precise criteria for the deletion of compatibles are stated and proved. A procedure to generate a set of symbolic compatibles is developed. The procedure is simple, systematic, and programmable as is evident from its recursive nature. The use of symbolic compatibles significantly reduces the computational work involved in finding a minimal covering machine.

For some machines it is still possible to reduce the set of symbolic compatibles yielding a smaller set of what are called reduced symbolic compatibles. This process, however, is a little more involved and is particularly suitable in a computer-aided environment. This is elaborately explained.

A set of symbolic compatibles never exceeds that of prime compatibles. Formal lemmas and theorems are stated, proved, and illustrated in order to conclude that the number of symbolic compatibles never exceeds that of prime compatibles. In fact, the set of symbolic compatibles is usually significantly smaller than the set of prime compatibles.

For the examples considered for the purpose of illustration, it is worthwhile to make a comparison given below.

Table 9.37 *A Comparison of the Cardinalities of Different Sets of Compatibles*

S. No.	ISSM Table No.	Cardinal Numbers of Different Sets of Compatibles			
		Primary	Prime	Symbolic	Reduced Symbolic
1	Table 9.8	40	29	10	10
2	Table 9.17	46	31	17	13
3	Table 9.24	49	31	23	23
4	Table 9.29	97	65	44	44
5	Table 9.32	38	29	16	16

Various upper bounds for the minimal closed cover have been discussed and it is interesting to consider the ***min-max cover*** before undertaking an exhaustive search with the symbolic compatibles. Eventually, minimisation of incomplete sequential machines requires generation of prime closed sets and simplification of PCS covering tables which yields the ***minimal closed cover.***

KEY WORDS

- ❖ Finite state machine (FSM)
- ❖ Finite state automaton (FA)
- ❖ Mealy model
- ❖ Moore model
- ❖ Merger chart
- ❖ Prime closed set
- ❖ Covering table
- ❖ Minimal closed cover
- ❖ Prime compatible
- ❖ Symbolic compatible
- ❖ Maximal compatible
- ❖ Rank of a compatible
- ❖ Implied compatible (IC)
- ❖ Unimplied compatible (UC)
- ❖ Weak compatible
- ❖ Implied part
- ❖ Uncovered part
- ❖ Min-Max cover

REVIEW QUESTIONS

1. Distinguish between equivalence and compatibility.
2. How many types of binary relations exist? List them in a tabular form depending on whether they obey the properties of reflexivity, symmetry and transitivity. Give one example for each.
3. What is meant by coverage relationship?
4. What is meant by implied compatibility classes for a given Incompletely Specified Sequential Machine (ISSM)? What is the significance of a closure class set?
5. How do you obtain maximum compatibility classes for a given ISSM?
6. What is meant by closure property?
7. What is meant by minimal closed cover?
8. Explain how a compatibility graph is drawn for an ISSM.
9. Is the minimal closed cover unique for a given machine?
10. Give an example of a sequence for which no sequential machine can be designed.
11. What is meant by a distinguishing sequence?
12. What is the need for bunching graph?
13. Minimum-row merged flow table discussed in connection with asynchronous machines can be written as a conventional state table if a column for PS is added. Then all stable states in a row will be assigned the same next state entry as the present state corresponding to the row. The unstable states have to be assigned the row corresponding to the internal state. Would that be a Mealy machine or Moore machine? Discuss.
14. Discuss the analogous nature of compatibility and merger. Does merger correspond to minimal closed cover using only MCs?

15. Distinguish between (a) primary compatibles, (b) basic compatibles, and (c) symbolic compatibles.
16. What are prime closed sets? Where are they used?
17. The following binary relations are learnt in this chapter. Indicate whether they obey the properties of reflexivity, symmetry, and transitivity in a table marking a check or cross under R,S,T.
    a) Equivalence
    b) Compatibility
    c) Implication
    d) Coverage
    e) Domination
    f) Deletion
    g) Exclusion
    h) Set inclusion

PROBLEMS

1. For the machine given below, find the equivalence partition and a corresponding reduced machine in standard form.

PS	NS, Z	
	X = 0	X = 1
A	D, 0	H, 1
B	F, 1	C, 1
C	D, 0	F, 1
D	C, 0	E, 1
E	C, 1	D, 1
F	D, 1	D, 1
G	D, 1	C, 1
H	B, 1	A, 1

2. a) Convert the following Mealy machine into a corresponding Moore machine.

PS	NS, Z	
	X = 0	X = 1
A	C, 0	B, 0
B	A, 1	D, 0
C	B, 1	A, 1
D	D, 1	C, 0

   b) Design the circuit for the above table.

3. a) Distinguish between Mealy and Moore machines.
   b) Convert the following Mealy machine into a corresponding Moore machine.

PS	NS, Z	
	X = 0	**X = 1**
A	B, 0	E, 0
B	E, 0	D, 0
C	D, 1	A, 0
D	C, 1	E, 0
E	B, 0	D, 0

4. a) Explain the limitations of finite state machines.

   b) Find the equivalence partition and a corresponding reduced machine in standard form for the machine given below.

PS	NS, Z	
	X = 0	**X = 1**
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

5. a) Explain in detail the Mealy state diagram and ASM chart for it with an example.

   b) Show the eight exit paths in an ASM block emanating from the decision boxes which check the eight possible binary values of three control variables x, y, z.

6. a) Distinguish between Mealy and Moore machines.

   b) Find the equivalence partition for the given machine and a standard form of the corresponding reduced machine.

PS	NS, Z	
	X = 0	**X = 1**
A	B, 0	E, 0
B	E, 0	D, 0
C	D, 1	A, 0
D	C, 1	E, 0
E	B, 0	D, 1

7. Transform the Mealy machine given below into a Moore machine, which will accept identical sequences.

PS	NS			
	I_1	I_2	I_3	I_4
A	E, 1	C, 0	B, 1	E, 1
B	C, 0	F, 1	E, 1	B, 0
C	B, 1	A, 0	D, 1	F, 1
D	G, 0	F, 1	E, 1	B, 0
E	C, 0	F, 1	D, 1	E, 0
F	C, 1	F, 1	D, 0	H, 0
G	D, 1	A, 0	B, 1	F, 1
H	B, 1	C, 0	E, 1	F, 1

8. Find the equivalence partition for the machine shown in table below.

PS	NS, Z	
	X = 0	X = 1
A	B, 1	H, 1
B	F, 1	D, 1
C	D, 0	E, 1
D	C, 1	F, 1
E	D, 1	C, 1
F	C, 1	C, 1
G	C, 1	D, 1
H	C, 0	A, 1

9. Find the equivalence partition for the following sequential machine.

PS	NS, Z	
	X = 0	X = 1
A	F, 0	B, 1
B	F, 0	A, 1
C	D, 0	C, 1
D	C, 0	B, 1
E	D, 0	A, 1
F	E, 1	F, 1
G	E, 1	G, 1

10. Find the minimal state machine which covers the machine given below.

PS	NS, Z	
	X = 0	X = 1
A	B, 0	C, 1
B	D, 0	C, 1

C	A, 0	E, 0
D	---	F, 1
E	G, 1	F, 0
F	B, 0	---
G	D, 0	E, 1

11. Find the minimal state machine which covers the machine given below.

PS	**NS, Z**		
	I_1	**I_2**	**I_3**
A	C, 0	E, 1	---
B	C, 0	E, -	---
C	B, -	C, 0	A, -
D	B, 0	C, -	E, -
E	---	E, 0	A, -

12. Find the minimal state machine which covers the machine given below.

PS	**NS, Z**	
	I_1	**I_2**
A	---	F, 0
B	B, 0	C, 0
C	E, 0	A, 1
D	B, 0	D, 0
E	F, 1	D, 0
F	A, 0	---

13. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	**NS, Z**			
	00	**01**	**11**	**10**
A	A, 00	E, 01	--	A, 01
B	--	C, 10	B, 00	D, 11
C	A, 00	C, 10	--	--
D	A, 00	--	--	D, 11
E	--	E, 01	F, 00	--
F	--	G, 10	F, 00	G, 11
G	A, 00	--	--	G, 11

14. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	NS, Z			
	00	01	11	10
A	A, 0	--	E, -	B, 1
B	E, -	C, 1	B, -	-
C	--	B, 0	-, 1	D, 0
D	A, 0	--	F, 1	B, -
E	B, 0	--	B, 0	--
F	--	C, 1	-, 0	C, 1

15. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	NS, Z			
	X_1	X_2	X_3	X_4
A	G, 1	-, -	-, 0	-, -
B	-, -	B, -	E, -	-, -
C	-, -	-, 1	H, -	E, -
D	-, -	-, 0	-, -	A, 1
E	G, 0	E, -	B, 1	-, -
F	-, -	I, -	-, -	B,0
G	D, -	D, -	-, 0	E, -
H	-, -	C, -	-, -	D, 1
I	F, 1	C, 0	-, 0	-, -

16. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	Nest State, Output Input			
	0	1	3	4
a	--	b, -	--	-, 1
b	a, -	--	c, 0	--
c	--	--	--	d, 1
d	b, -	a, -	b, -	f, 0
e	c, -	c, -	a, -	-, 0
f	-, 0	b, -	--	h, 1
g	-, 1	f, -	e, 1	d, 1
h	-, 1	g, -	--	e, 1

17. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	Nest State, Output Input			
	0	1	2	3
a	--	g, -	e, 1	d, -
b	a, -	d, -	--	-, 0
c	c, -	-, 0	--	g, 1
d	e, 0	--	a, -	--
e	-, 1	f, -	-, 1	-, 1
f	-, 1	e, -	a, 1	-, 1
g	f, -	-, 1	b, -	h, -
h	c, -	--	a, 0	--

18. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	Nest State, Output Input			
	00	11	12	13
A	A, -	--	E, -	B, 1
B	E, -	C, 1	B, -	--
C	--	B, 0	-, 1	D, 0
D	A, 0	--	F, 1	B, -
E	B, 0	--	B, 0	--
F	--	C, 1	-, 0	C, 1

19. For the Incompletely Specified Sequential Machine given below, find the minimal reduced machine which covers the given machine.

PS	NS, Z $X_1 X_2$			
	00	01	11	10
A	E, 0	--	--	--
B	--	F, 1	E, 1	A, 1
C	F, 0	-	A, 0	F, 1
D	--	--	A, 1	--
E	--	C, 0	B, 0	D, 1
F	C, 0	C, 1	--	--
‘G	E, 0	--	--	A, 1

20. a) Find the maximum compatibles for the machine given below.

b) Show that the compatibles (BD) and (CD) can be ‘deleted’. Find the set of symbolic compatibles and a minimal closed cover for the machine.

PS	NS, Z			
	I_1	I_2	I_3	I_4
A	--	B, --	--	--, 1
B	A, --	--	C, 0	--
C	--	--	--	D, 1
D	B, --	A, --	B, --	F, 0
E	C. --	C, --	A, --	--, 0
F	--, 0	B, --	--	H, 1
G	--, 1	F, 1	E, 1	D, 1
H	--, 1	G, --	--	E, 1

21. a) While every maximal compatible is a prime compatible, every MC may not be a symbolic compatible. Justify this statement. In the machine given below, show at least one MC which is not a symbolic compatible.

    b) Reduce the machine using any method.

PS	NS, Z			
	I_0	I_1	I_2	I_3
A	B, O	--	B, O	--
B	A, -	C,1	B, --	--
C	--	B, O	-, 1	D, O
D	E, O	--	F,1	B, -
E	E, O	--	A, -	B, 1
F	--	C, 1	-, O	C, 1

22. a) Obtain the maximal compatibles for the ISSM given below.

    b) Find how many of the MCs can be 'deleted' in the first iteration itself in the process of searching for a minimal closed cover.

PS	NS, Z							
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A	C, -	D, -	B, 0	C, -	F, 1	--	--	--
B	E, -	C, -	--	--	--	F, 0	H, 0	E, -
C	-, 1	--	C, -	A, -	D, 1	--	--	--
D	--	A, 0	--	--	F, -	F, -	--	--
E	--	-, 1	--	E, 1	F, -	--	G, 1	--
F	A, 0	--	G, 1	C, 0	-, 1	D, -	--	E, 1
G	A, 0	--	-, 1	--	G, -	--	F, -	H, 1
H	E, 0	C, 1	F, 1	--	G, -	--	--	--

10

Algorithmic State Machine Charts

Learning Objectives

After studying this chapter, you will know about:

- Usefulness of algorithmic state machine (ASM) charts in the design of control logic in digital machines.
- Elegance in writing the Boolean function by inspecting the ASM chart.
- The Differences between Data processor and Control circuit and the interconnections.
- Differences between conventional flow chart and ASM chart.
- The importance of using the various design techniques for control depending on the complexity.
    - Conventional design using gates.
    - Design using Multiplexers
    - PLA control.

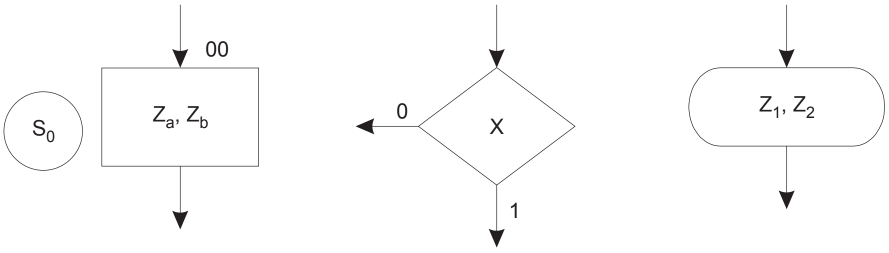
Just as a flow chart serves as a useful aid in writing software programs, Algorithmic State Machine (ASM) charts help in the hardware design of digital systems. ASM charts provide a conceptually simple and easy-to-visualise method of implementing algorithms used in hardware design. For example, Add–Shift algorithm for multiplication of two numbers, Shift–Subtract algorithm for obtaining division, and so on. ASM charts are advantageously used in the design of the control unit of a computer and, in general, control networks in any digital systems. ASM charts look similar to flow charts but differ in that certain specific rules must be observed in constructing them. An ASM chart is equivalent to a state diagram or state graph. It is an useful tool which leads directly to hardware realisations.

It is necessary at this stage to distinguish between **“data processing”** and **“control”.** The former involves not merely addition, subtraction, complementation, and such other combinational operations but sequential operations such as shifting and counting as well. The control part of a system has to take care of the timing and proper sequencing of the various operations involved. It has to necessarily provide clock pulses and move the system from one state to the other where certain predesigned operations are performed in each state. Control constitutes the heart of any digital system such as the digital computer. An ASM chart is a useful tool in designing control circuits which are invariably sequential circuits containing flip-flops.

10.1 COMPONENTS OF ASM CHART

ASM charts are constructed from ASM blocks. For each state of a given machine, there will be one ASM block. An ASM block comprises the following components.

1. **State Box** The state box is represented by a rectangle. The name of the state is written in a circle to the left of the state box and the binary code assigned to the state is indicated outside on the top right side of the box. The outputs, if any, associated with the state are written within the box. These are clearly Moore type outputs.
2. **Decision Box** The decision box is represented by a diamond-shaped symbol with true and false branches. A Boolean variable or input or expression written inside the diamond indicates a condition which is evaluated to determine which branch to take.
3. **Condional Output Box** The conditional output box is represented by a rectangle with rounded corners or by an oval within which the outputs that depend on both the state of the system and the inputs are indicated. These are Mealy type outputs. At times, some operations like addition may also be indicated which involve register transfers.



State Box Decision Box Conditional Output Box

10.2 INTRODUCTORY EXAMPLES OF ASM CHARTS

Example 1—Mod-5 Counter

Recall the synchronous sequential machines discussed in an earlier chapter. Look at the state diagram of Mod-5 counter. An ASM chart equivalent to the state diagram is shown in Fig. 10.1. The states are now represented by state boxes instead of nodes. Transitions are still represented by arrows but the inputs indicated adjacent to arrows in the state diagram are replaced by decision boxes with true and false branches. In the state graph, we had indicated the outputs along with inputs using a separator slash or comma. In the ASM chart, Mealy type outputs are now indicated in conditional output boxes and Moore type outputs are indicated within the state box itself.

Take a look at the ASM chart which has five state boxes named S_0 through S_4. For every clock pulse, x is sensed. If 1, then the machine goes to the next state; if 0, then the machine remains in the same state, as

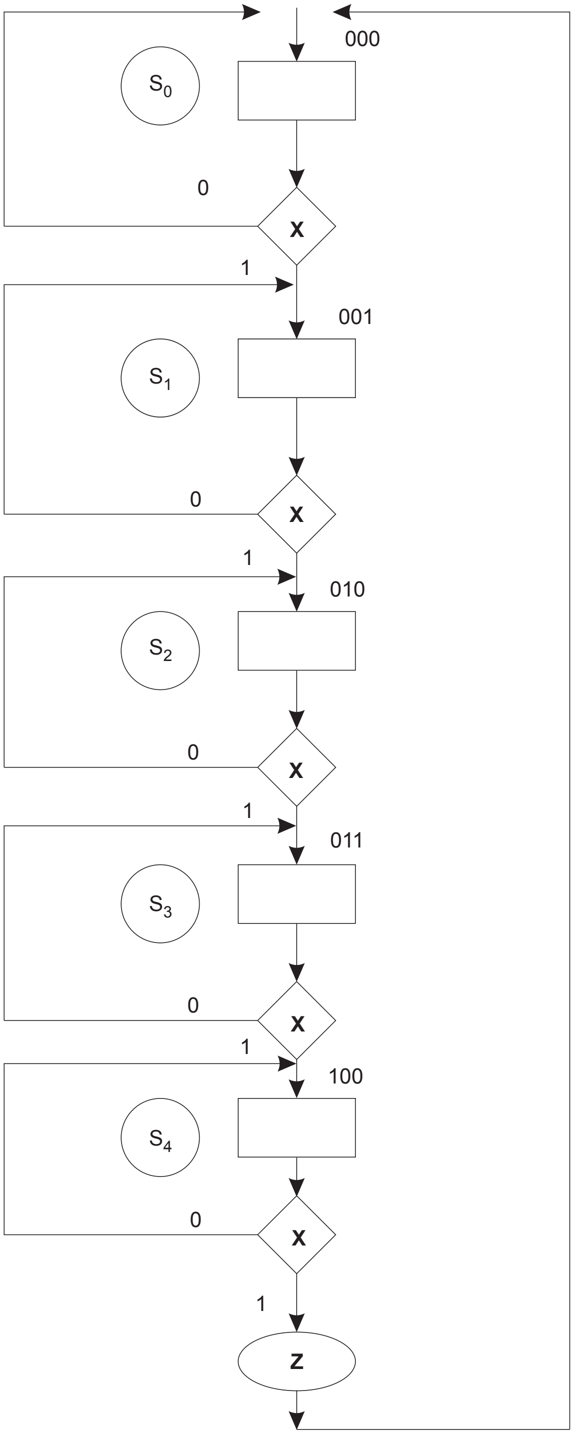


Fig. 10.1 ASM chart for Mod-5 counter

depicted by a return branch. All this happens in the same clock cycle. When the machine is in state S_4, on the occurrence of the clock pulse, if x is 1, the machine produces an output pulse indicated as Z and goes to the initial state S_0. Note specially that all this happens simultaneously during the same clock cycle, that is, the same clock pulse period. Remember that the states are assigned binary names using three flip-flops.

Focus now on synthesis. Earlier, we synthesised using T-flip-flops. Let us, now, use D flip-flops. The reader is advised to appreciate the elegance in synthesis with ASM charts. With D flip-flops, the excitation D_i has to be the same as the next state variable Y_i. Observing only state assignment indicated on the ASM chart, we notice that the next value Y_0 has to become 1 for the present states, $y_2 y_1 y_0$ = 000, 010, only. Hence, using decimal codes and remembering that 6, 7 never occur, we get

$$D_0 = Y_0 = x\left[\sum(0, 2) + \sum\phi(5, 6, 7)\right]$$

Similarly, we obtain expressions for all the excitations and outputs by merely inspecting the ASM chart.

$$D_1 = Y_1 = x\left[\sum(1, 2) + \sum\phi(5, 6, 7)\right]$$

$$D_2 = Y_2 = x\left[\sum(3) + \sum\phi(5, 6, 7)\right]$$

$$Z = x\left[\sum(4) + \sum\phi(5, 6, 7)\right]$$

Notice that the input x is ANDed with each of the expressions for the excitations. If the input x is exclusively provided, then the circuit counts the number of clock pulses in the duration when x is at level 1. In other words, x enables the counter. Sometimes, x is not provided, in which case the clock pulses are counted modulo-5 and an output pulse is produced for every five clock pulses.

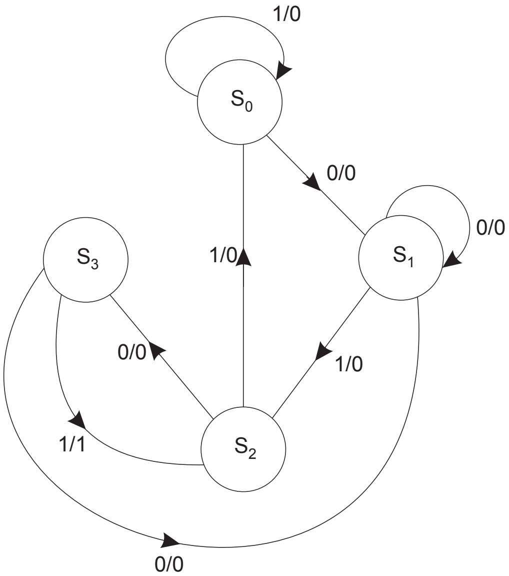
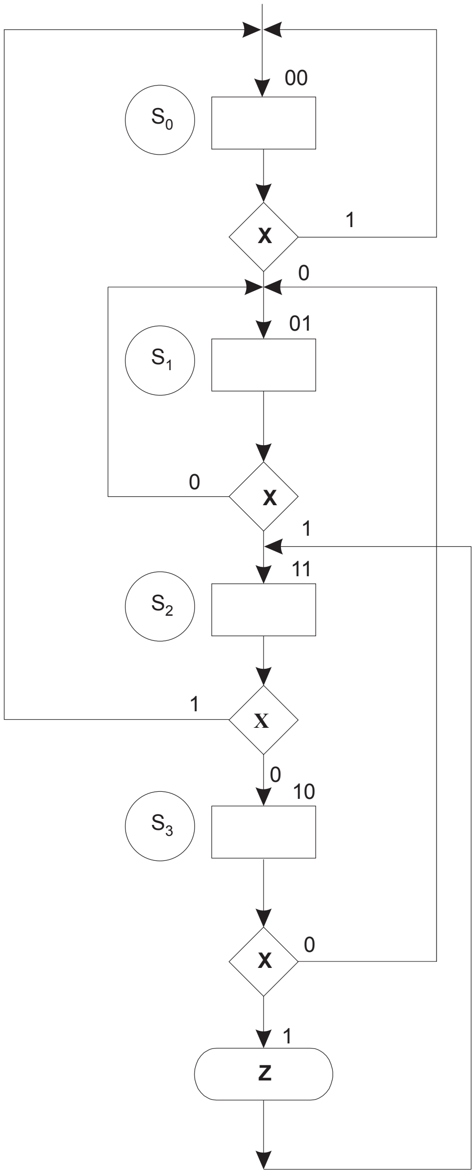


Fig. 10.2 Sequence detector for 0101: overlapping sequences allowed

Example 2—Sequence Detector

We designed a sequence detector for the sequence 0101 in an earlier chapter. The reduced state diagram (also called state graph) is given in Fig. 10.2 for the purpose of reference.

It is fairly simple to draw an equivalent **ASM chart** for a given state diagram and vice-versa. The ASM chart is shown in Fig. 10.3. Notice that the ASM chart has four state boxes. It has four **decision boxes** and **branches** corresponding to the arcs in the state diagram for x = 1 or 0. It has one **conditional output box**



By inspection

$D_0 = Y_0 = x'y_1'y_0' + xy_1'y_0$

$D_1 = Y_1 = xy_1'y_0 + x'y_1y_0$

$Z = xy_1y_0'$

Fig. 10.3 ASM chart for sequence 0101 detector

corresponding to the output ‘Z’ becoming ‘1’ if input x = 1 when the machine is in state S_3 and a clock pulse occurs. Notice that there are four ASM blocks, each containing one state box associated with one or more decision boxes and conditional output boxes. It is important to note that ASM charts have timing considerations in contrast with the flow charts used by programmers. All the operations associated with a state box form one ASM block and they have to be performed at the same time in one clock pulse period.

Focus now on synthesis. Take a note of the binary values, $y_1\, y_0$ assigned to the states S_0, S_1, S_2, S_3. If we decide to use two D flop-flops, the excitation table is identical to the transition table and $D_i = Y_i$. In this ASM chart, note that the next state variable Y_0 becomes 1 in S_1 and S_2. The corresponding present states are S_0 ($y_1 y_0$ = 00) and S_1 ($y_1 y_0$ = 01) and transition occurs if x = 0 and x = 1 respectively. Thus, we may write the expressions for excitations by inspection as follows.

$$D_0 = Y_0 = x'y_1'y_0' + xy_1'y_0$$

Note that Y_1 becomes 1 for S_2 and S_3 on x = 1 from S_1 and x = 0 from S_2 respectively.

$$D_1 = Y_1 = xy_0'y_0 + x'y_1y_0'$$

By similar reasoning, we obtain the expression for Z.

$$Z = xy_1y_0'$$

Drawing the actual logic circuit consisting of two D flip-fops is left to the reader.

Example 3—Serial Adder

As a third example, let us use serial adder presented in an earlier chapter to reinforce the concepts on ASM charts. The state graph is reproduced in Fig. 10.4 for the purpose of reference.

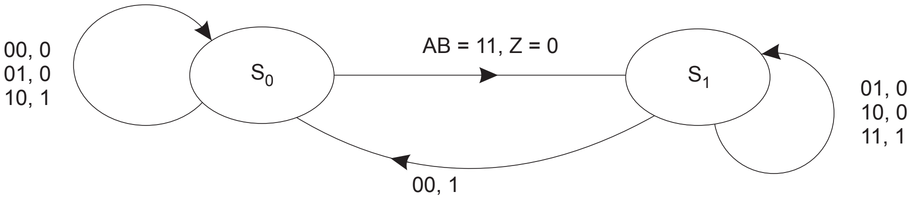


Fig. 10.4 State graph of serial adder

Fig. 10.5 shows the ASM chart equivalent to the state graph describing the same behaviour with primitive steps; hence, it is called an algorithmic chart. Notice the four exit paths depending on the values of A and B arranged in the form of a binary tree. Notice that all the paths merge into one entry path for each state. Compare now with the state graph. From each state, there is an exit path to the other state and three re-entry paths to itself. This is because there are four different values that A B can assume 00, 01, 10 and 11 and hence there must be as many exit paths–three of them happen to be re-entry paths in this example. Notice in the state graph that there are four incoming paths which merge into one entry path to each state. Correspondingly, there are as many branches in the ASM chart. Notice specially that if the number of inputs is large, the state graph can be quite complex and it follows that the ASM chart too can become unwieldy. The decision boxes are usually binary as we aim at an algorithmic procedure comprising primitive steps. As we have two inputs, we have to

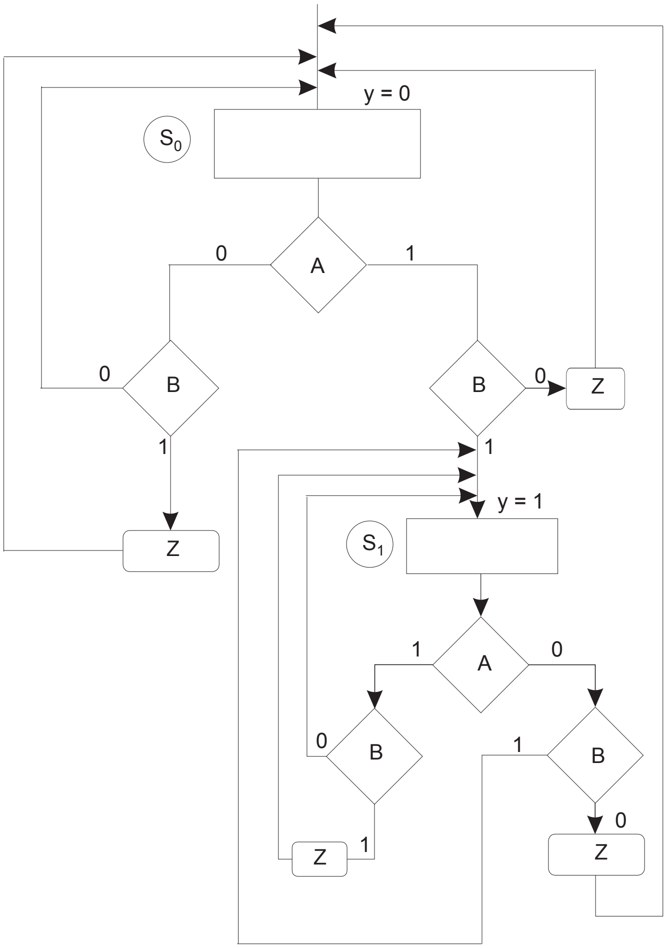
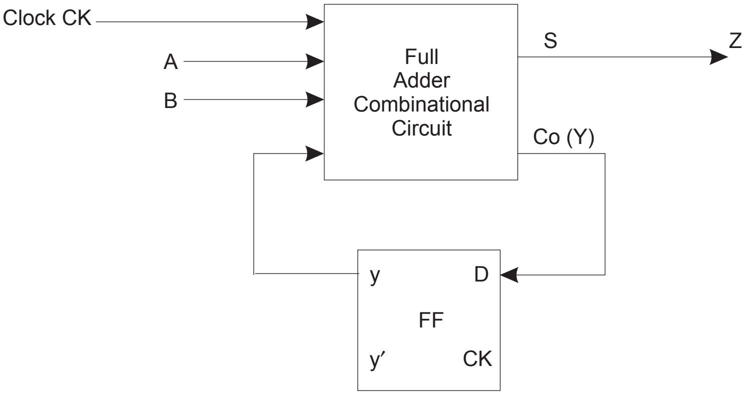


Fig. 10.5 ASM chart of serial adder



Structure of a serial adder

show four exit paths from each state. If we provide a decision box with a function like A′ • B, we should be in a position to specify uniquely what the machine should do for both the possibilities 0 or 1. We prefer the binary tree structure shown in Fig. 10.5. The advantage of an ASM chart is that it helps the designer to conceptually visualise the various operations and their time relationships.

Synthesis of the circuit By inspection of the ASM chart, we note that there are only two states S_0 and S_1 and hence we need to have only one flip-flop. If we choose a D flip-flop, D is easily synthesised looking at the entry paths into each state. For the only flip-flop, D becomes 1 for state S_1. There are four arrows confluent on entry into S_1. Hence, by inspection and tracing the paths indicated by arrows, we write the expression for D given below. Remember that y is the present state variable of the only flip-flop.

$$\begin{aligned} D &= y'AB + yAB' + yAB + yA'B \\ &= AB + By + Ay \end{aligned}$$

By a similar reasoning, the state S_0 is characterised by y = 0. The corresponding excitation is D′. There are four paths confluent on the entry into S_0. By inspection and tracing the paths, we may write

$$\begin{aligned} D' &= y \cdot A'B' + y' \cdot A'B' + y' \cdot A'B + y' \cdot AB' \\ &= A'B' + A'y' + B'y' \end{aligned}$$

Note The reader is advised to check consistency in the expressions for D and D′.

There are four conditional output boxes. Remember that the output Z is the sum bit to be produced serially. Tracing the corresponding paths, it is easy to write the expression for Z.

$$\begin{aligned} &y'(A'B + AB') + y(A'B' + AB) \\ &= A \oplus B \oplus y \end{aligned}$$

Notice that expressions Z and D are same as those for sum S and output carry C_0 of full adder. The structure is shown next to the ASM chart.

10.3 SALIENT FEATURES OF ASM CHART

Let us put at one place all the important features of ASM charts before we take up some interesting and useful applications presented next.

1. Each ASM block contains exactly one state box together with the decision boxes and conditional output boxes associated with that state.
2. An ASM block has **exactly one entrance path and one or more exit paths.** All entry lines have to be merged and only one entry line to be shown for each state. For every valid combination of input variables, there must be exactly one exit path defined which leads to a unique next state. It is possible that several paths may merge into one exit path.
3. An ASM block describes the machine operations during the time the machine is in that particular state (usually one clock pulse period). When a digital system enters the state associated with a given ASM block, the outputs indicated within the state box become true (logical 1). The conditions associated with the decision boxes are evaluated to determine which path or paths to be followed through the ASM block. If a conditional output box is encountered along a path, the corresponding outputs become

true. All this happens during one clock period and the next clock pulse takes the machine to the next state as indicated by the exit path.

4. A path through an ASM block from entrance to exit is referred to as a **link path.**
5. Internal feedback within an ASM block is **not** permitted. Even so, following a decision box or conditional output box, the machine may re-enter the same state.
6. While a flow chart gives the order in which various operations are to be performed to achieve a given task, it has no reference to clock periods or states of a machine. In an ASM chart, all the operations such as evaluating conditions given in the decision boxes and giving the outputs as indicated in the state box or conditional output boxes associated with a block are performed in one clock period; they are done simultaneously even though they are indicated at different places within the same ASM block.

10.4 ALGORITHMIC STATE MACHINE FOR WEIGHT COMPUTATION

Our objective is to design a sequential machine which computes the number of 1s in a given binary word. In the algorithm for tabular minimisation of Boolean functions, we need to arrange the minterms in the ascending order of their weights. This is only one of the many situations when we have to examine the 1s of a given binary word.

The weight of a binary number (word or code or vector) is defined as the number of 1s contained in its binary representation. Let the digital system contain a register R and another register W which acts as a counter and one flip-flop F. Initially, the number has to be loaded into register R and W has to be initialised. Then, shift each bit of register R into the flip-flop F, one at a time. Sense the value of F; whenever it is 1, the register W is incremented by one. At the end, W contains the weight of the word.

The ASM chart is shown in Fig. 10.6. It has three inputs S, Z, and F and four states S_0, S_1, S_2, S_3. S is the start input; Z is for sensing all zeros in R, and the value of F decides whether W is to be incremented or not.

Normally, the machine is in state S_0 until the start signal S is made 1 to enable computing the weight of the word in R. While in S_0, if S is 1, the clock pulse causes three jobs to be done simultaneously.

1. Input word is loaded in parallel into register R.
2. Counter W is set to all 1s and
3. The machine is transferred to state $S_{1.}$

While in state S_1, the clock pulse causes two jobs to be done simultaneously.

1. Increment W. (In the first round, all 1s become all 0's) and
2. If Z is 0, the machine goes to the state S_2; If Z is 1, the machine goes to the state S_0.

The machine reaches this state again from S_3, if the value of F is sensed as 1. While in state S_1, Z has to be sensed; if 0, go to S_2; if 1, go to S_0. If Z is 0, it means that the weight of the word loaded into register R is 1 or more and the machine has to go to S_2 to continue the process. If Z is 1, it means that the word in R contains in it all zeros and hence the weight of the word is already indicated by W. In this case, the machine should go back to the initial state S_0 having completed the task of computing the weight of the given word. In state S_2, register R is shifted so that one bit appears in the flip-flop F. While in state $S_{3,}$ the value of F decides whether the machine goes to S_1 to increment W or S_2 to shift R into F. Remember that all the operations occur in coincidence with the clock pulse while in the corresponding state. Also notice that the register R should eventually contain all 0s when the last 1 is shifted into F.

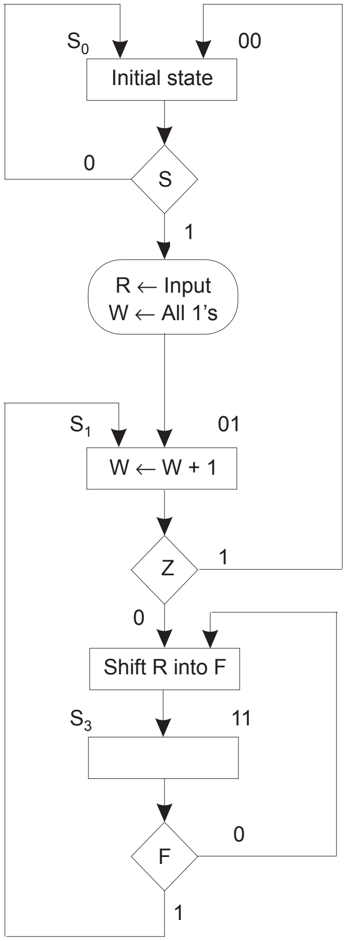


Fig. 10.6 ASM chart for weight computation machine

System Design

The system (also referred to as a machine) consists of two subsystems—data processor and control. The data processor subsystem shown in Fig. 10.7 performs the tasks prescribed for each state of the ASM blocks. The control subsystem ensures the proper sequence of states and transitions depending on the inputs and the present state of the machine. Let T_0 be 1 when the machine is in state S_0; let T_1 be 1 when the machine is in state S_1, and so on, that is, when the machine is in state S_i, T_i is 1. Keep referring to Fig. 10.6 of ASM chart and Fig. 10.7 of data processor.

Data Processor Subsystem

A data processor subsystem is shown in Fig. 10.7. T_0, T_1, T_2, and T_3 are the signals produced by the control subsystem, which is discussed later. In respect of data processing, the reader is urged to verify and get a clear picture of what is happening. Initially, the machine will be in state S_0, that is, T_0 is1. For starting the process, make the input S equal to 1. The clock pulse occurring while in S_0 loads the input word parallelly into register

R, and sets the register W to all 1s at the same time. The same pulse takes the machine to state S_1 and T_1 becomes 1.

While in state S_1, the clock pulse increments W and makes it 0 to begin with. The same pulse senses Z. If Z = 1, then it would mean that the word in R contains all 0s and so the same clock pulse takes the machine to the initial state S_0 as the weight of the given word is already indicated by W and the process ends. If Z is sensed as 0 during $T_{1,}$ then it means that the binary word in the register contains one or more 1s in it. So the process has to continue and the control ensures that the same clock pulse takes the machine to state S_2 and T_2 becomes 1.

While in state S_2 (T_2 = 1), the clock pulse shifts R left by one bit so that the MSB of R appears in F and 0 is shifted into the LSB place. The same clock pulse takes the machine to state S_3 and T_3 becomes 1.

During T_3, F is sensed by the control circuit. If F = 1, the clock pulse takes the machine to S_1 and T_1 becomes 1. If F is 0, the same clock pulse takes the machine to S_2 and T_2 becomes 1. The process is repeated until all the bits of R are sensed or Z becomes one, whichever is earlier.

Notice in particular the block which checks for 0. One simple way is to obtain the logical NOR function of all the bits of R. The AND gate between R and F is enabled when T_2 = 1, this is, in state S_2.

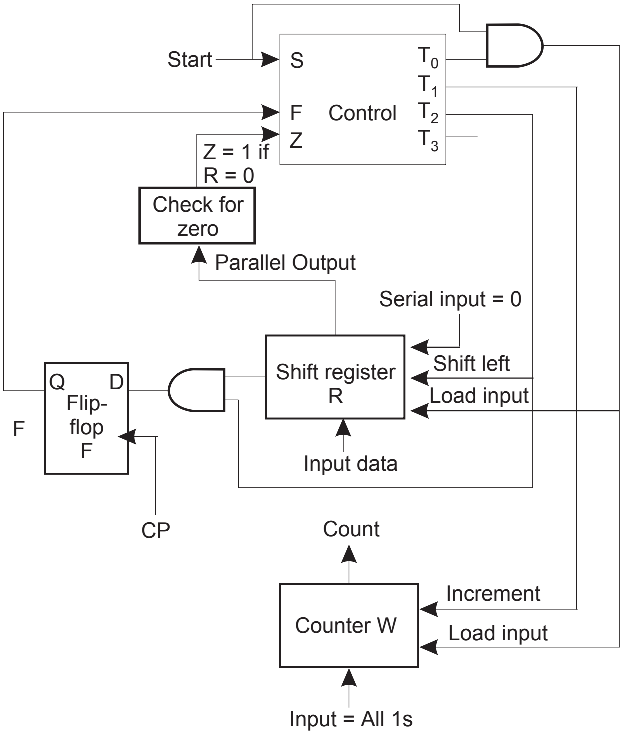


Fig. 10.7 Data processor subsystem for weight computation

Control Subsystem

There are several techniques in vogue to design the control subsystem. D flip-flops are extensively used. Multiplexer control and PLA control have special features. These are discussed below one after another.

Conventional Hardware Realisation

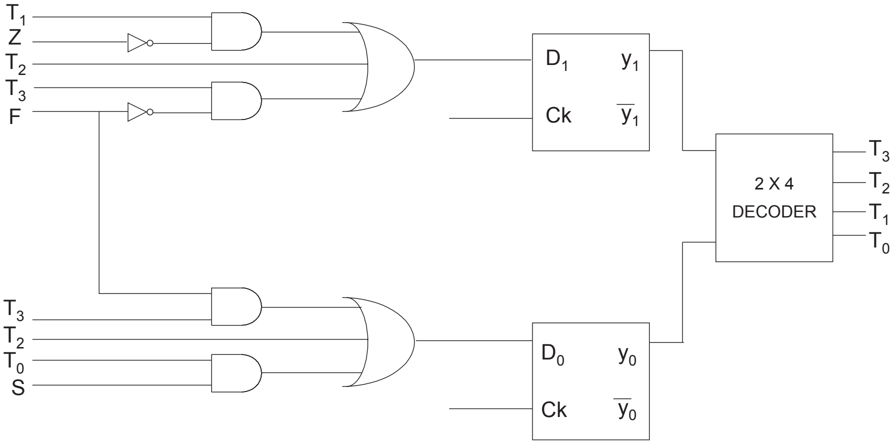
A state table conventionally consists of rows labelled by state variables and columns labelled by inputs. The entries in the cells are "Next state values". For the purpose of this example, it is better to modify the format of the state table in order to visualise the process and develop a simple conceptual framework. The system requires four states, which can be provided with two binary variables y_1, y_0 (also called state variables). By decoding the state variables, one can obtain the control timing signals T_0, T_1, T_2, T_3.

Let Y_1, Y_0 be the next state variables which depend on the inputs S, Z, F and the present state variables y_1, y_0. If D flip-flops are used, the excitations D_1, D_0 are obtained straight away as $D_1 = Y_1$ and $D_0 = Y_0$. It now remains to synthesise the Ds as functions of y_1, y_0, S, Z, F. Normally it requires the use of a five-variable map. Instead, we may as well work with a table such as the one given as in Fig. 10.8(a) wherein there are many don't

Present State		Inputs			Next State		Hardware Realisation	
y_1	y_0	S	Z	F	Y_1	Y_0	$D_1 = Y_1$	$D_0 = Y_0$
0	0	0	-	-	0	0	0	S
		1	-	-	0	1		
0	1	-	1	-	0	0	$\bar{Z}$	0
		-	0	-	1	0		
1	0	-	-	-	1	1	1	1
1	1	-	-	0	1	0	$\bar{F}$	F
		-	-	1	0	1		

$$\begin{aligned} D_1 &= Y_1 = y_1'y_0 Z' + y_1y_0' + y_1y_0 F' \text{ in SOP form} \\ &= T_1 Z' + T_2 + T_3 F' \\ D_0 &= Y_0 = y_1'y_0'S + y_1y_0' + y_1y_0 F \text{ in SOP form} \\ &= T_0 S + T_2 + T_3 F \end{aligned}$$

(a) State table and excitations



(b) Logic circuit

Fig. 10.8 Realisation of weight computation machine

care entries. In this modified state stable, we use one row for each transition. At first, focus on the rows of the present state $y_1 y_0 = 00$.

Observe in particular how D_1 and D_0 are assigned values. For instance, when the machine is in the present state $y_1 y_0 = 00$, the next state $Y_1 Y_0$ may be 00 or 01, depending on whether the input S is 0 or 1. Inputs Z and F become don't care conditions. Thus D_1 has to be assigned 0 as $Y_1 = 0$ in both cases. D_0 has to be assigned a value equal to S.

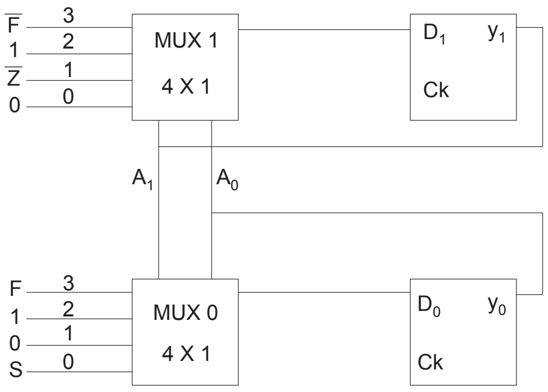
Starting from the present state $y_1 y_0 = 01$, the next state may be $Y_1 Y_0 = 00$ or 10, depending on whether the input Z is 1 or 0. Hence, the excitation D_1 of the flip-flop 1 has to be 0 if Z is 1 and 1 if Z is 0, which means $D_1 = Z'$. Flip-flop 0 has to go to 0 in either case and hence $D_0 = 0$. The reader is advised to work out the other entries in the table and get a feel for the procedure adopted.

Having assigned values for the excitations, it is now a simple job to obtain the logic functions for D_1 and D_0. They are given in Fig. 10.8(a) and the resulting logic circuit is shown in Fig. 10.8(b).

There are other elegant methods used in designing the control subsystem. They are presented next.

Present State		**Inputs**			**Next State**		**MUX Realisation**	
y_1	y_0	S	Z	F	Y_1	Y_0	$D_1 = Y_1$ MUX 1	$D_0 = Y_0$ MUX 0
0	0	0	-	-	0	0	0	S
		1	-	-	0	1		
0	1	-	1	-	0	0	$\overline{Z}$	0
		-	0	-	1	0		
1	0	-	-	-	1	1	1	1
1	1	-	-	0	1	0	$\overline{F}$	F
		-	-	1	0	1		

(a) State table



(b) Logic circuit

Fig. 10.9 Multiplexer control

Multiplexer Control

Use of multiplexers in the control circuit makes the design much more elegant and helps the designer to conceive how the states are changing with every clock pulse. The strategy is to use one multiplexer for each excitation. The present state variables, that is, outputs of the flip-flops are connected to the select (address) inputs of MUX's are shown in Fig. 10.9(b). In the present example, the machine has four states realised by two D type flip-flops, D_1 and D_0. The outputs of MUXs feed the flip-flops and the MUXs will have as many inputs as there are states in the machine. These are actually next state variables. One of these is chosen to flow to the output depending on the present state fed to the select inputs. For the convenience of the reader, the state table of Fig. 10.8(a) is reproduced as Fig. 10.9(a) for the purpose of immediate reference. In our present example, outputs of MUX 1 and MUX 0 are the excitations D_1 and D_0. These excitations would be the same as the next state variables.

For the purpose of illustrations, let the machine be in state S_0 initially. Only when the input signal S is made 1, the machine starts functioning in accordance with the state table given in 10.9(a). So long as S is kept at '0', the machine remains in state S_0. Let us examine the effect of the inputs S = 1, Z = 0, F = 1. For the present state being 00 (S_0), the next state, reached on the occurrence of the clock pulse, is given by 01 (S_1) because the input S is 1. For the next clock pulse, S_1 becomes the present state and the next state is easily read from the table as 10 because the input Z = 0. Starting from the state S_2, whenever clock pulse occurs, the machine reaches state S_3 regardless of the inputs. The reader is urged to workout carefully the transition for the given assignment of inputs and get a feel for the working of the machine. For example, assignment of the inputs namely S = 1, Z = 0, F = 1 and assuming the machine starting from S_0, the transitions in coincidence with the clock pulses in succession, are indicated below. Initially, while in S_0, the 0th input of both the MUXs are selected . When in state S_1, MUXs select the 1st inputs which are $Z' = 1$ for MUX 1 and 0 for MUX 0. Outputs of MUXs feed the flip-flop which change state as shown below.

$$\begin{array}{lrcrcrcrcrcr}
 & & & & & \multicolumn{3}{c}{S = 1,\ Z = 0,\ F = 1} & & & & \\
 & y_1 = 0 & \rightarrow & 0 & \rightarrow & \overline{Z} = 1 & \rightarrow & 1 & \rightarrow & \overline{F} = 0 & \rightarrow & \overline{Z} = 1 \\
 & y_0 = 0 & \rightarrow & S = 1 & \rightarrow & 0 & \rightarrow & 1 & \rightarrow & F = 1 & \rightarrow & 0 \\
\text{State} & S_0 & \rightarrow & S_1 & \rightarrow & S_2 & \rightarrow & S_3 & \rightarrow & S_1 & \rightarrow & S_2
\end{array}$$

PLA Control

Besides the conventional hardware control or the multiplexer control, there is a third option of using a PLA for designing the control subsystem.

Figure 10.10 shows the PLA control block. Notice that the state variables y_1, y_0 and the inputs S, Z, F become the inputs numbered as 1 through 5. The excitations D_1, D_0 for the flip-flops (same as next state variables Y_1, Y_0) and the control signals T_0, T_1, T_2, T_3 become the outputs of the PLA. The PLA program table is shown in Fig. 10.10(a). It consists of products, inputs, and outputs. Note that the product terms are listed from the logic expressions of D_1 and D_0, which are reproduced below again for reference purpose.

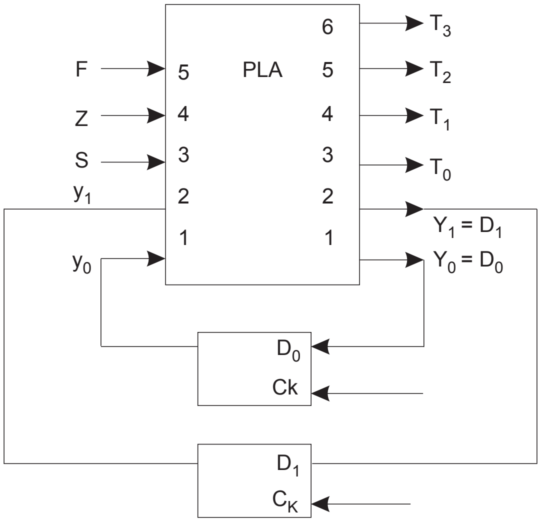
$$\begin{aligned}
D_1 &= Y_1 = y_1' y_0 Z' + y_1 y_0' + y_1 y_0 F' \text{ in SOP form} \\
&= T_1 Z' + T_2 + T_3 F' \\
D_0 &= Y_0 = y_1' y_0 S + y_1 y_0' + y_1 y_0 F \text{ in SOP form} \\
&= T_0 S + T_2 + T_3 F
\end{aligned}$$

PLA control with reference to ASM chart and program table The reader is urged to work out this example independently and get a feel for the working. Remember that the machine will be in state S_0 when $T_0 = 1$, in S_1 when $T_1 = 1$, in S_2 when $T_2 = 1$ and in S_3 when $T_3 = 1$. These control signals T_0, T_1, T_2, T_3 are usually obtained by decoding the state variables y_1, y_0. If we decide to use a PLA for the control subsystem, we may as well dispense with a separate decoder and integrate the complete design.

Take a keen look at the table in Fig. 10.10(a). In each row, the input variables contained in the product are marked 0 or 1, depending on whether the variable appears in a complemented form or an uncomplemented form. For instance, row 3 represents the product $y_1 y_0'$ and hence the corresponding input column 1 representing y_0 is marked 0 and Column 2 indicating y_1 is marked 1. In the same row 3, some output columns also are marked 1, which will be explained below.

Product Terms		Inputs					Outputs					
		y_0 1	y_1 2	S 3	Z 4	F 5	$D_0 = Y_0$ 1	$D_1 = Y_1$ 2	T_0 3	T_1 4	T_2 5	T_3 6
$T_0 = y_1' y_0'$	1	0	0	-	-	-			1			
$T_1 = y_1' y_0$	2	1	0	-	-	-				1		
$T_2 = y_1 y_0'$	3	0	1	-	-	-	1	1			1	
$T_3 = y_1 y_0$	4	1	1	-	-	-						1
$T_1 Z = y_1' y_0 Z'$	5	1	0	-	0	-		1		1		
$T_3 F' = y_1 y_0 F'$	6	1	1	-	-	0		1				1
$T_0 S = y_1' y_0' S$	7	0	0	1	-	-	1		1			
$T_3 F = y_1 y_0 F$	8	1	1	-	-	1	1					1

(a) PLA program table



(b) Block diagram

Fig. 10.10 PLA control

Now take a look at the output columns. Each column represents a sum of products and is hence marked 1 in the corresponding row. For example, D_0 (Col. 1) = $y_1 y_0'$ (row 3) + $y_1 y_0' S$ (row 7) + $y_1 y_0 F$ (row 8). Hence in the output column 1, rows 3, 7, and 8 are marked 1. Blanks are to be assumed as 0s. Don't care entries are marked by dashes.

Similarly, D_1 (Col. 2) = $y_1 y_0'$ (row 3) + $y_1' y_0 Z'$ (row 5) + $y_1 y_0 F'$ (row 6). Hence, in column 2, rows 3, 5, and 6 are marked 1. Output columns 3, 4, 5, 6 represent the control signals corresponding to the states of the machine. Observe in output column 3 corresponding to T_0, row 7 given by $y_1' y_0' S$ will be a redundant term marked 1. Similar situation exists for columns 4 and 6 also. Recall that the outputs are produced by OR gates in a PLA.

Let us now correlate state transitions in the ASM chart with the entries in the PLA program table. Look at column T_0 when the machine will be in state S_0. Look at row 7; whenever input S (input Column 3) becomes 1 while T_0 is 1, the excitation D_0 becomes 1 causing state transition from 00 to 01, that is, from S_0 to S_1 (see at the ASM chart).

Observe T_1 (output column 4) to study further transitions from the state S_1 i.e., $y_1 y_0 = 01$. Notice that in row 5, input Z is 0 and the corresponding $D_1 = 1$ and $D_0 = 0$. This causes state transition from 01 to 10, that is, from S_1 to S_2 (look at the ASM chart). If Z = 1, that is, all 0s in R in state S_1, D_1 remains unaltered at 0 and D_0 becomes 0 the transition will be from 01 to 00.

Observe T_2 (output column 5) to visualise the state transition from S_2 (look at the ASM chart). In this column, there is a single 1 corresponding to decoding of state variables, that is, $y_1' y_0'$ in row 3. There are no entries in the rows 5, 6, 7, 8 of this column which means that there are no input conditions stipulated for further transition. Also notice that when T_2 is 1, both D_1 and D_0 are at 1 level consistent with the logic functions obtained earlier. Therefore, the next clock pulse unconditionally takes the machine to state S_3 making $T_3 = 1$.

Lastly, look at column 6 giving the behaviour of the machine in state S_3 when $T_3 = 1$. In the table, $D_0 = Y_0$ is marked first and $D_1 = Y_1$ next. In column 6, the rows 6 and 8 have 1s which imply that D_1 would become 1 if F = 0 (see row 6) and D_0 would become 1 if F = 1 (see row 8). Thus the next state will be $Y_1 Y_0 = 10$, that is, S_2 if the input F = 0 or $Y_1 Y_0 = 01$, that is, S_1 if F = 1. This may be verified from the ASM chart.

This completes the design of the example machine using the three methods—firstly, the conventional hardware logic design; secondly, the multiplexer control; and thirdly, the PLA control.

10.5 ALGORITHMIC STATE MACHINE FOR A BINARY MULTIPLIER

The objective of this section is to design a binary multiplier using add-shift algorithm. The manual working is given below using the unsigned numbers 01101 as the **multiplicand** and 01011 as the **multiplier.** The procedure is simple. The **partial products,** formed by multiplying the multiplicand with each bit of the multiplier starting from LSB and proceeding to MSB are successively written one below the other, shifting left each time by one bit position. Adding all the partial products results in the **product.** If the **word length** of each number is n bits, then the product would be of length 2n bits at most.

0 1 1 0 1	←	$(13)_{10}$ … Multiplicand
0 1 0 1 1	←	$(11)_{10}$ … Multiplier
0 1 1 0 1	——	partial product 1
0 1 1 0 1 •	——	partial product 2

0 0 0 0 0 • •	———	partial product 3
0 1 1 0 1 • • •	———	partial product 4
0 0 0 0 0 • • • •	———	partial product 5
0 1 0 0 0 1 1 1 1	←———	$(143)_{10}$ … Product

Digital implementation requires the following changes.

1. In manual working, we perform **left shift** on the subsequent partial product which is yet to be formed. This kind of anticipatory job is not done by a physically realisable machine. A real machine can operate only on the existing operands but not on future results. Hence, we shift the partial product already formed to the right by one bit and add the next partial product in its natural position. This would produce the correct result as the relative positions of the operands for addition are as they should be.
2. Instead of forming all the partial products and then adding, which would require a large number of registers to store them, each partial product is added to register A (accumulator) and shifted right. This job is repeated n times where n is the word length of the machine and so is the length of the multiplier. With this background, let us now proceed to design the data processor subsystem first and the control subsystem next.

The data processing subsystem is shown in Fig. 10.11. It comprises the following.

1. Register B holds the multiplicand.
2. Register QM holds the multiplier M. As the same register is used to hold the quotient Q, in division, this is usually called the QM register. Let us call this by the single letter Q.
3. Register A, called accumulator, to hold the cumulative sum of partial products.
4. A parallel adder circuit.
5. One flip-flop C to hold the carry, if any, produced in addition.
6. A counter P which is initialised to the word length n.
7. Provision to decrement P and check for zero.
8. Provision to concatenate the registers C, A, Q to form a combined register of length $1 + n + n = 2n + 1$ bits with facility to shift right.

As to how a digital computer performs multiplication using add-shift algorithm, is illustrated in Fig. 10.12. Initially, multiplicand is loaded into register B and multiplier is loaded into register Q. C and A are cleared. The word length n is loaded into a register P which acts as a counter. The least significant bit Q_0 of the multiplier Q is sensed by the machine. If $Q_0 = 1$, then form the partial product by adding B to A, that is, $A \leftarrow A + B$ and then shift the combined register CAQ to the right by one bit position. Q_0 goes out and in its place, present Q_1 becomes Q_0 for the next iteration. A_0 becomes Q_{n-1}, C becomes A_{n-1} and the place of C is filled by 0. If $Q_0 = 0$, then the partial product is 0 and hence there is no need to add. Only shift CAQ right. Repeat the process n times in a loop starting from $(n - 1)$ and proceed to 0. After traversing through the loop, test the counter p. If it shows up 0, stop and exit from the loop.

The algorithmic state machine chart for the above procedure is shown in Fig. 10.13. Notice that there are four states including the initial state for the specific tasks listed below. Also, notice that they are assigned binary names $y_1 y_0$.

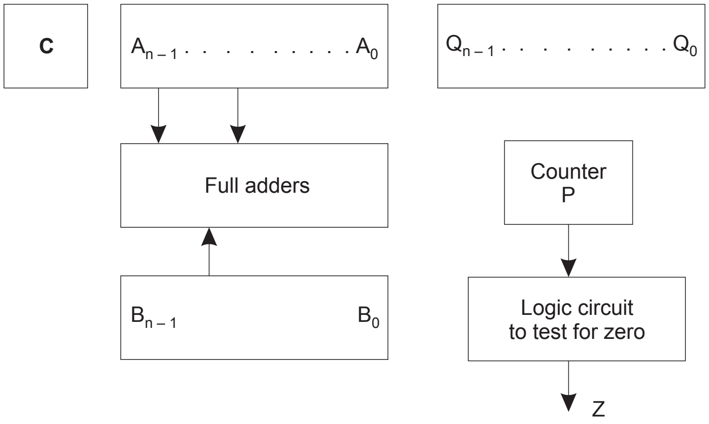

C
A_{n-1} . . . . A_0
Q_{n-1} . . . . Q_0
Full adders
Counter P
B_{n-1}
B_0
Logic circuit to test for zero
Z

Fig. 10.11 Data processor subsystem for multiplication

B	C	A	Q	Components	Count P
0 1 1 0 1	0	0 0 0 0 0	0 1 0 1 1	B ← Multiplicand	1 0 1 (5)
				Q ← Multiplier	
				A ← 0, C ← 0, P ← n	
				P ← P – 1,	1 0 0 (4)
0 1 1 0 1	0	0 1 1 0 1	0 1 0 1 **1**	Q_0 = 1, A ← A + B,	
	0	0 0 1 1 0	1 0 1 0 **1**	CAQ Shifted right	
				P ← p – 1,	0 1 1 (3)
0 1 1 0 1	0	1 0 0 1 1	1 0 1 0 1	Q_0 =1, A ← A + B,	
	0	0 1 0 0 1	1 1 0 1 **0**	CAQ Shifted right	
				P ← p – 1,	0 1 0 (2)
				Q_0 = 0	
0 1 1 0 1	0	0 0 1 0 0	1 1 1 0 **1**	CAQ Shifted right	
				P ← p – 1,	0 0 1 (1)
0 1 1 0 1	0	1 0 0 0 1	1 1 1 0 1	Q_0 =1, A ← A + B,	
	0	0 1 0 0 0	1 1 1 1 0	CAQ Shifted right	
				P ← p – 1,	0 0 0 (0)
				Q_0 = 0	
0 1 1 0 1	0	0 0 1 0 0	0 1 1 1 1	CAQ Shifted right. STOP	

Fig. 10.12 Multiplication in a computer

S_0 → Initial state. While in this state, the clock pulse occurring when S = 1 takes the machine to the state S_1 by producing the required excitations to flip-flops.

S_1 → While in this state, the clock pulse initialises the registers B, Q, A, C, P and the same pulse takes the machine to state S_2.

$S_2 \rightarrow$ The clock pulse occurring while the machine is in state S_2 decrements P. In the first traversal of the loop, P becomes n – 1. The same pulse does either of two jobs depending on the value of the bit Q_0 (LSB of multiplier).

If $Q_0 = 1$, then $A \leftarrow A + B$ and go to the next state S_3.

If $Q_0 = 0$, do nothing and go to the next state S_3.

Note: All this will be done in the same clock cycle and the same pulse will take the machine to state S_3. This timing pulse does three jobs at the same time.

(i) $P \leftarrow p - 1$

(ii) $A \leftarrow A + B$ if $Q_0 = 1$

(iii) Transition to S_3

$S_3 \rightarrow$ The clock pulse occurring in this state enables two jobs at the same time.

1. Shift CAQ right by one bit position
2. Transition to S_2 if Z = 0 or to S_0 if Z = 1

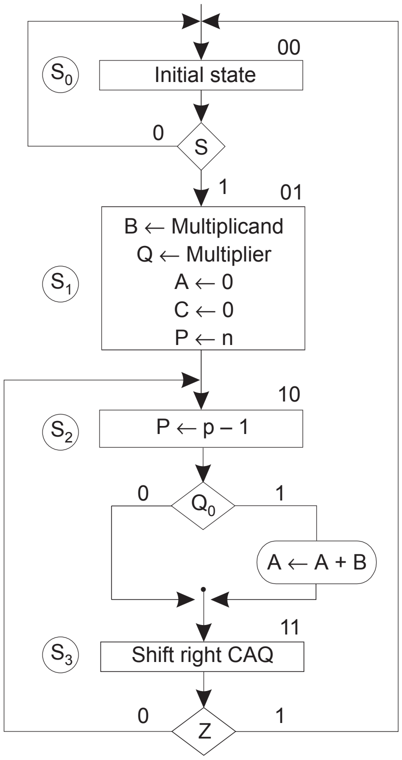


Fig. 10.13 ASM chart for binary multiplier

Notice that the ASM chart contains three decision boxes within which the variables S, Q_0, and Z are written. These are the input variables each serving a specific purpose.

S → for starting the machine

Q → to decide whether or not to add B to A

Z → to monitor the counter which keeps, decrementing from n – 1 with each traversal of loop and show up when p becomes 0. Then Z becomes 1 and the machine goes to the initial state to be available for starting another multiplication.

The timing wave forms T_0, T_1, T_2, T_3 corresponding to the states S_0, S_1, S_2, $S_{3.}$ and the add-enable signal Q_0 are to be provided by the control unit.

Control Subsystem

The state table for the control subsystem of a binary multiplier is shown in Fig. 10.14(a). It is extremely simple to fill this table. Look at the present state PS ($y_1 y_0$) and the next state NS ($Y_1 Y_0$) columns first. Notice in the ASM chart of Fig. 10.13 that there are two transitions from the state 00. They are 00 to 00 (re-entry) and 00 to 01. Hence, provide two rows and enter the corresponding columns of PS and NS. Again from the ASM chart, notice that the transition 00 to 00 occurs if S is 0 and the other transition 00 to 01 occurs if S is 1. Make these entries in the S column. Other inputs for these transitions become don't care entries, shown by dashes.

For PS of 01, there is only one transition to state 10 in the ASM chart. Hence only one row is provided and the columns PS and NS are filled accordingly. This transition has no input conditions and hence the columns of S, Q_0, Z become don't cares, indicated by dashes. In a similar manner, the remaining rows are filled.

For PS of 10 the transition to state 11 occurs in two paths depending on Q_0. If $Q_0 = 0$, then the machine goes straight to the state 11. If $Q_0 = 1$, then the machine has to add B to A and go to state 11. Although this could have been represented by one row, we show two rows for clarity as the value of Q_0 is different.

For PS of 11, there are two possible transitions—11 to 10 if Z = 0 or 11 to 00 if Z = 1.

Now look at the output columns. $T_0 = 1$ for PS 00, T_1 for PS 01, T_2 for PS 10 and T_3 for PS 11. These are obtained by decoding the state variables y_1, y_0. Finally, the output $Q_0 \bullet T_2$ is for the purpose of enabling addition in state 10 as depicted by the oval block in ASM chart. All these outputs eventually provide the control timing signals.

The control unit needs two flip-fops. If we choose D flip-flops, it is easy to get the following expressions for the excitations as functions of state variables y_1, y_0 and the inputs S, Q_0, Z. These are shown in Fig. 10.14(b) for the purpose of immediate reference.

$$\begin{aligned} D_1 = \; & Y_1 = y_1' y_0 + y_1 y_0' + y_1 y_0 Z' \\ = \; & y_1' y_0 + y_1 y_0' + y_0 Z' \\ D_0 = \; & Y_0 = y_1' y_0' S + y_1 y_0' \\ = \; & y_0' S + y_1 y_0' \end{aligned}$$

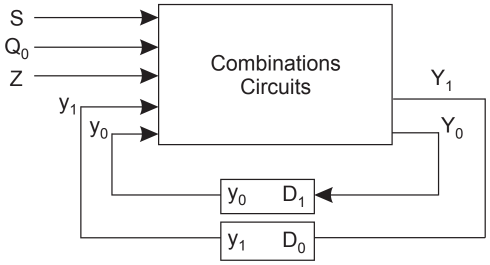
The structure of the control unit is indicated in Fig. 10.14(c).

Note: The input Q_0 merely enables the adder in state 10 and hence does not participate in the excitation functions. Nevertheless, it has to produce the control signal $Q_0 \bullet T_2$ to enable the adder.

PS (y_1y_0)		Inputs			NS (Y_1Y_0)		Outputs (Timing Signals)				
y_1	y_0	S	Q_0	Z	Y_1	Y_0	T_0	T_1	T_2	T_3	$Q_0 \bullet T_2$
0	0	0	--	--	0	0	1				
0	0	1	--	--	0	1	1				
0	1	--	--	--	1	0		1			
1	0	--	0	--	1	1			1		
1	0	--	1	--	1	1			1		1
1	1	--	--	0	1	0				1	
1	1	--	--	1	0	0				1	

(a) State table for the control subsystem

$$\begin{aligned} D_1 = \; & Y_1 = y_1'y_0 + y_1y_0' + y_1y_0 Z' \\ & = y_1'y_0 + y_1y_0' + y_0 Z' \\ D_0 = \; & Y_0 = y_1'y_0' S + y_1y_0' \\ & = y_0' S + y_1y_0' \end{aligned}$$



(b) Excitations

Fig. 10.14 Control subsystem of a multiplier

PLA Control

If we wish to design a PLA for the control unit, we need to list the products, inputs, and outputs given in Fig. 10.15(a) which serves as the PLA program table. Dashes in the table are don't care entries and blanks are 0s.

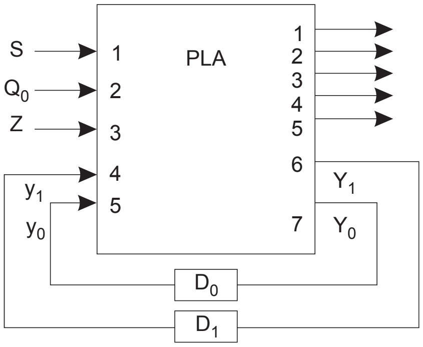
Multiplexer Control

If we wish to design a multiplexer control, we need to have two MUXs, one each for the state variables y_1, y_0. The present state variables will be fed to address (selected) inputs. Each MUX must have four inputs corresponding to the number of states. S_0, S_1, S_2, S_3 and the corresponding next values will be fed to the inputs. The state table and the circuit are shown in Fig. 10.16 (a) and (b) respectively.

Look at the state table. There are two rows for the present state 00. One corresponds to re-entry, that is, the next state is the same state 00. The second row depicts the transition from present state 00 to the next state 01.

Product No.	**Inputs and State Variables**					**Outputs**						
	W 1	**Q_0** 2	**Z** 3	**y_1** 4	**Y_0** 5	**T_0** 1	**T_1** 2	**T_2** 3	**T_3** 4	**$Q_0\,T_2$** 5	**Y_1** 6	**Y_0** 7
$y_1'\,y_0' = 1$	--	--	--	0	0	1						
$y_1'\,y_0 = 2$	--	--	--	0	1		1				1	
$y_1\,y_0' = 3$	--	--	--	1	0			1			1	1
$y_1\,y_0 = 4$	--	--	--	1	1				1			
$y_0\,Z' = 5$	--	--	0	--	1						1	
$y_0'\,S = 6$	1	--	--	--	0							1
$y_1 y_0'\,Q_0 = 7$	--	1	--	1	0					1		

(a) PLA program table



(b) Block diagram

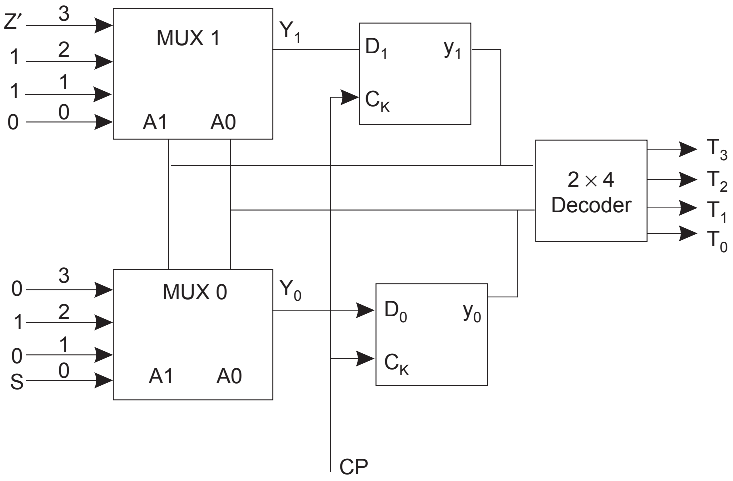
Fig. 10.15 PLA control for multiplier

In both the rows, the next state variable Y_1 is 0 and hence the 0th input of MUX 1 is to be 0, as shown in Fig. 10.16(b). Y_0 is 0 in one row in which S is 0 and Y_0 is 1 in the other row in which S is 1. It follows that Y_0 = S and hence for MUX 0, the 0th input should be S as shown in Fig. 10.16(b). Using similar reasoning, the columns of MUX 1 and MUX 0 are filled. The inputs of MUXs corresponding to the present states are assigned to the four input leads. These appear as Ys, which drive the D flip-flops. Finally the state variables y_1, y_0 are decoded to produce the four control signals T_0, T_1, T_2, T_3 corresponding to the four states S_0, S_1, S_2, S_3.

A serious learner will wonder what is the role of Q_0 in the control subsystem. Look at the state table of Fig. 10.16(a) and the rows corresponding to state 10. Notice that regardless of the value of Q_0, the next state is 11. We, therefore, infer that Q_0 has no role in causing state transitions. But $Q_0 = 1$ enables the adder in state 10. We convince ourselves that Q_0 serves a purpose in a data processor subsystem but not directly in control. Even so, we have to provide the control signal $Q_0\,T_2$.

PS (y_1 y_0)		S	Q_0	Z	NS (Y_1 Y_0)		MUX 1	MUX 0
0	0	0	--	--	0	0		
0	0	1	--	--	0	1	0	S
0	1	--	--	--	1	0	1	0
1	0	--	0	--	1	1		
1	0	--	1	--	1	1	1	1
1	1	--	--	0	1	0		
1	1	--	--	1	0	0	Z′	0

(a) State table with MUXs



(b) Multiplexer control for binary multiplier

Fig. 10.16 Multiplier with MUX control

SUMMARY

In the previous chapters we used various types of flip-flops such as S-R, J-K, T and D to design *synchronous sequential circuits,* also called *finite state machines or finite automata.*

Use of D flip-flops in conjunction with a decoder was the simplest method as the excitations were the same as the next state variables, that is, D_i º Y_i . If we use one as shown we may dispense with the need for a decoder. The outputs of the individual flip-flops will then furnish the timing signals.

While a *data processing subsystem* comprises both the combinational as well as sequential circuits, a *control subsystem* essentially consists of sequential circuits which provide different states for different operations and interstate transitions based on some predesigned conditions. An *ASM chart* is a convenient tool for this purpose.

Conventional realisations might end up in an irregular structure which is difficult to interpret even by the very designer, leave alone the users. Multiplexer control results in a systematically organised structure which is easily interpreted and the sequence of states is easily found and visualised.

PLA control affords the facility of integration in a chip form.

KEY WORDS

❖ Finite state machine (FSM)
❖ Finite state automaton (FA)
❖ 'ASM chart'
❖ State box
❖ Decision box
❖ Conditional output box
❖ Branch
❖ Entrance
❖ Exits
❖ Link path

REVIEW QUESTIONS

1. Sketch a part of an ASM chart that specifies a conditional operation to increment register W during state T_1 and transfer to state T_2 if control inputs Z and F are equal to 0 and 1, respectively.
2. Show the eight exit paths in an ASM block emanating from the decision boxes that check the eight possible binary values of three control variables, x, y, and z.
3. Obtain the ASM chart for the following state transition.
   If x = 0, control goes from state T_1 to T_2; if x = 1, generate a conditional output F and go from T_1 to T_2.
4. Obtain the ASM chart for the following state transition.
   If x = 1, control goes from T_1 to T_2 and then to T_3; if x = 0, control goes from T_1 to T_3.
5. Obtain the ASM chart for the following state transition:
   Start from state T_1; then if xy = 00, go to T_2; if xy = 01, go to T_3; if xy = 10, go to T_1; otherwise, go to T_3.
6. Prove that the multiplication of two n-bit numbers gives a product of length at most 2n bits.
7. How do you indicate Moore outputs and Mealy outputs in an ASM block?
8. Distinguish between a data processing subsystem and a control subsystem.
9. What is the advantage of using D flip-flops in control ?
10. What is the advantage of using MUXs for control ?
11. How many MUXs are required for the control subsystem?
12. What is the advantage of using a PLA for control ?
13. What is the basic difference between a conventional flow chart and algorithmic state machine chart?
14. What is the advantage of using one flip-flop per state in designing a control unit?

1. a) Discuss the salient features of an ASM chart.
   b) Draw the portion of an ASM chart that specifies the conditional operation to increment register *R* during state T_1 and transfer to state T_2, if control inputs z and y are =1 and 0 respectively.

PROBLEMS

2. a) How do you indicate Moore and Mealy output in an ASM chart? Show an example.
   b) Sketch the eight exit paths in an ASM block emanating from the decision boxes that check the eight possible binary values of three control variables x, y, z.
3. a) Distinguish between an ASM chart and a conventional flow chart with the help of an example.
   b) Obtain the ASM charts for the following state transitions.
      i) If x = 0, control goes from T_1 to state T_2; if x = 1, generate a conditional operation and go from T_1 to T_2.
      ii) If x = 1, control goes from T_1 to T_2 and then to T_3; if x = 0, control goes from T_1 to T_3.
      iii) Start from state T_1, then if xy =00, go to T_2, if xy = 01, then go to T_3, if xy = 10, then go to T_1, otherwise go to T_3.
4. Discuss the merits and demerits of the following methods of designing the control subsystem of a digital system.
   a) Method using K flip-flops for 2^K states
   b) One flip-flop per state
   c) Using J-K flip-flops versus D flip-flops
   d) Multiplexer control
   e) PLA control
5. Construct an ASM chart for a digital system that counts the number of people in a room. People enter the room from one door that has a photocell which changes a signal x from 1 to 0 when the light is interrupted. They leave the room from a second door with a similar photocell with a signal y. Both x and y are synchronised with the clock, but they may stay on or off for more than one clock-pulse period. The data processor subsystem consists of an up-down counter with a display of its contents.
6. Design a digital system with three 4-bit registers, A, B, and C, to perform the following operations.
   a) Transfer two binary numbers to A and B when a start signal is enabled.
   b) If A < B, shift left the contents of A and transfer the result to register C.
   c) If A > B, shift right the contents of B and transfer the result to register C.
   d) If A = B, transfer the number to register C unchanged.
7. Design a digital system that multiplies two binary numbers by the repeated addition method. For example, to multiply 5 by 4, the digital system evaluates the product by adding the multiplicand four times: 5 + 5 + 5 + 5 = 20. Let the multiplicand be in a register B, the multiplier in register Q, and the product in the concatenated register CAQ. An adder circuit adds the contents of B to A and carry produced, if any, will be in one flip-flop C. A zero-detection circuit Z checks when Q becomes 0 after each time that it is decremented.
8. Obtain an ASM chart to implement Booth’s algorithm for multiplication. Discuss the data processor subsystem and the control subsystem separately.
9. Develop an ASM chart for performing “division” using the restoring technique.

10. Develop an ASM chart for performing "division" using the non-restoring technique.
11. Design the control for the state table given in the table using two multiplexers, a register, and a decoder.

Present Sate Symbol	Present State		Inputs			Next State		Outputs		
	y_1	y_2	S	A_3	A_4	Y_1	Y_2	T_0	T_1	T_2
T_0	0	0	0	X	X	0	0	1	0	0
T_0	0	0	1	X	X	0	1	1	0	0
T_1	0	1	X	0	X	0	1	0	1	0
T_1	0	1	X	1	0	0	1	0	1	0
T_1	0	1	X	1	1	1	1	0	1	0
T_2	1	1	X	X	X		0	0	0	1

Appendix A: ASCII Code

	Graphic or Control	**ASCII (Hexadecimal)**
NUL	Null	00
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of Transmission	04
ENQ	Enquiry	05
ACK	Acknowledge	06
BEL	Bell	07
BS	Backspace	08
HT	Horizontal Tabulation	09
LF	Line Feed	0A
VT	Vertical Tabulation	0B
FF	Form Feed	0C
CR	Carriage Return	0D
SO	Shift Out	0E
SI	Shift In	0F
DLE	Data Link Escape	10
DC1	Device Control 1	11
DC2	Device Control 2	12
DC3	Device Control 3	13
DC4	Device Control 4	14
NAK	Negative Acknowledge	15
SYN	Synchronous Idle	16
ETB	End of Transmission Block	17

(Contd.)

Contd.

CAN	Cancel	18
EM	End of Medium	19
SUB	Substitute	1A
ESC	Escape	1B
FS	File Separator	1C
GS	Group Separator	1D
RS	Record Separator	1E
US	Unit Separator	1F
SP	Space	20

Graphic or Control	ASCII (Hexadecimal)	Graphic or Control	ASCII (Hexadecimal)
!	21	Q	51
“	22	R	52
#	23	S	53
$	24	T	54
%	25	U	55
&	26	V	56
‘	27	W	57
(	28	X	58
)	29	X	58
*	2A	Y	59
+	2B	Z	5A
’	2C	[	5B
-	2D	\	5C
.	2E	]	5D
/	2F	^	5E
0	30	¯	5F
1	31		60
2	32	a	61
3	33	b	62
4	34	c	63
5	35	d	64
6	36	e	65
7	37	f	66
8	38	g	67
9	39	h	68
:	3A	i	69
;	3B	j	6A
<	3C	k	6B
=	3D	l	6C

(Contd.)

Contd.

>	3E	m	6D
?	3F	n	6E
@	40	o	6F
A	41	p	70
B	42	q	71
C	43	r	72
D	44	s	73
E	45	t	74
F	46	u	75
G	47	v	76
H	48	w	77
I	49	x	78
J	4A	y	79
K	4B	z	7A
L	4C	{	7B
M	4D	|	7C
N	4E	}	7D
O	4F	~	7E
P	50	DEL Delete	7F

Appendix B: Symbolic Compatibles (For the Machine of Table 9.29)

This appendix complements the discussion of Section 9.9. While the machine under discussion is represented by Table B.1, its implication chart is given in Table B.2. All the compatibles together with their closure class sets are listed in Table B.3; the compatibles are grouped in accordance with their order (number of states in each compatible) and each group is listed in an alphabetical order. Deletions and exclusions are indicated in the remarks column by the letters D and E in the same notation used in Chapter 9. Table B.4 shows the implied compatibles and unimplied compatibles; the MICs and MUCs are noted below this table. Only one iteration suffices for this machine. Note that the symbolic compatibles are indicated in the immediately adjacent parentheses as (C_i), $1 \le i \le 44$.

Table B.1(9.29) *An Example of an Incomplete Sequential Machine*

Present State	Next State, Output Input			
	00	**01**	**11**	**10**
A	B, -	D, -	--	C, -
B	F, -	I, -	--	--
C	--	--	G, -	H, -
D	B, -	A, -	F, -	E, -
E	--	--	--	F, -
F	A, 0	--	B, -	-, 1
G	E, 1	B, -	--	--
H	E, -	--	--	A, 0
I	E, -	C, -	--	--

Table B.2 (9.30) *Implication Chart for the Machine of Table B.1*

B	BF DI							
C	CH	√						
D	~~CE~~	~~BF~~ ~~AI~~	~~FG~~ ~~EH~~					
E	CF	√	~~FH~~	EF				
F	AB	AF	BG	AB BF	√			
G	~~BE~~ ~~BD~~	EF BI	√	BE AB	√	✗		
H	BE AC	EF	AH	BE AE	AF	✗	√	
I	~~BE~~ ~~CD~~	EF CI	√	BE AC	√	AE	BC	√
	A	B	C	D	E	F	G	H
MCs: BCGHI, BEGHI, DEGHI, ABCF, ABCH, ABEF, ABEH, BCFI, BEFI, DEFI								

Table B.3 *Compatibles for the Machine of Table B.1*

PC (C_i)	**Closure Class Set (E_i)**	**Remarks**
$\underline{B}$ $\underline{C}$ G $\underline{H}$ I (C_1)	EF, AH; BE, AC	
$\underline{B}$ $\underline{E}$ G $\underline{H}$ $\underline{I}$	EF, BCI, AF; AB, BF, DI, AC, CH, AH	D2 (BEGI)
D $\underline{E}$ $\underline{G}$ $\underline{H}$ $\underline{I}$	BE, ABC, AEF; BF, CH, CF, AH, BG, BI, CI	D2 (DI)
$\underline{A}$ $\underline{B}$ $\underline{C}$ $\underline{F}$ (C2)	DI, BG, CH; BE, EF, BI, CI, AH	
$\underline{A}$ $\underline{B}$ C H (C3)	BEF, DI; AF	
$\underline{A}$ $\underline{B}$ E $\underline{F}$ (C4)	DI, CF; AC, CH, AH, BG, BI, CI	
$\underline{A}$ $\underline{B}$ $\underline{E}$ $\underline{H}$ (C5)	BEF, DI, ACF; BG, CH, BI, CI	
$\underline{B}$ $\underline{C}$ $\underline{F}$ $\underline{I}$ (C6)	AEF, BG; AB, DI, BE, AC, CH, AH	
$\underline{B}$ $\underline{C}$ G $\underline{H}$	EF, BI, AH; CI, BE, AC	E (BCGHI)
B C G I (C7)	EF;	
$\underline{B}$ $\underline{C}$ $\underline{H}$ I	EF, AH; BE, AC	E (BCGHI)
$\underline{B}$ $\underline{E}$ $\underline{F}$ $\underline{I}$ (C8)	AEF, CI, BG; AB, CF, DI, AC, CH, AH	
$\underline{B}$ $\underline{E}$ G $\underline{H}$	EF, BI, AF; CI, AB, BF, DI, AC, CH, AH	D2 (BEG)
$\underline{B}$ $\underline{E}$ G $\underline{I}$ (C9)	EF, BCI;	
$\underline{B}$ $\underline{E}$ $\underline{H}$ $\underline{I}$	EF, CI, AF; AB, BF, DI, AC, CH, AH	D2 (BEI)
$\underline{B}$ G H $\underline{I}$ (C10)	EF, BCI;	
$\underline{C}$ G $\underline{H}$ I (C11)	BC, AH; BE, AC, CH	
D $\underline{E}$ $\underline{F}$ $\underline{I}$ (C12)	ABE, AC, BF; CF, CH, AF, BG, AH, BI, CI	
$\underline{D}$ $\underline{E}$ $\underline{G}$ $\underline{H}$	BE, AB, AEF; BF, DI, CF, AC, BG, CH, AH, BI, CI	D1 (Ø)

(Table B.3 Contd.)

Table B.3 Contd.

D E G I	BE, ABC, EF; BF, CH, AF, AH	D2 (DGI)
D E H I	BE, AC, AEF; CH, AB, CF, AH, BF, BG, BI, CI	D2 (DI)
D G H I	BE, ABC, AE; BF, CH, CF, AF, AH, BG, EF, BI, CI	D2 (DI)
E G H I A B C (C13)	BC, AF; AB, BF, DI, BE, AC, CH, AH BF, DI, CH; AF, BE, AH	D1 (G)
A B E	BF, DI, CF; AF, AC, BG, CH, EF, BI AH, CI	E (ABEF)
A B F (C14)	DI; BE, AC, CH, AH	
A B H	BEF, DI, AC; CH, AF	E (ABCH)
A C F	AB, BG, CH; BF, DI, EF, BI, AH, BE, CI	E (ABCH)
A C H (C15)	BE;	
A E F	AB, CF; BF, DI, BG, BE, AC, BI, CH, CI, AH	E (ABEF)
A E H (C16)	BE, ACF; AB, BG, CH, BF, DI, EF, BI, CI	
B C F	AF, BG; AB, DI, EF, BI, BE, AC, CI, CH, AH	E (ABCF)
B C G	EF, BI; CI	E (BCGI)
B C H	EF, AH; BE, AC	E (BCGHI)
B C I	EF;	E (BCGI)
B E F (C17)	AF, AB, DI, AC, CH, AH	
B E G (C18)	EF, BI; CI	
B E H	EF, AF; AB, BF, DI, AC, CH, AH	D2 (BE)
B E I (C19)	EF, CI	
B F I	AEF, CI; AB, CF, DI, BG, BE, AC, CH, AH	E (BCFI, BEFI)
B G H (C20)	EF, BI; CI	
B G I	EF, BCI;	E (BCGI, BEGI, BGHI)
B H I (C21)	EF, CI	
C F I (C22)	AE, BG; EF, BI	
C G H (C 23)	AH; BE, AC	
C G I (C24)	BC;	
C H I (C25)	AH; BE, AC	
D E F	AB, BF; DI, BE, AC, CH, AH, AF	D2 (EF)
D E G	BE, AB, EF; BF, DI, AF, AC, CH, AH	D1 (G)
D E H	BE, AEF; AB, CF, BF, DI, BG, AC, BI, CH, CI, AH	D1 (Ø)
D E I	BE, AC, EF; CH, AH	D2 (DI)
D F I	ABE, AC, BF; CF, AF, BG, CH, EF, BI, AH, CI	D2 (DI)
D G H	BE, AB, AE; BF, DI, AF, AC, CH, AH, EF, BI, CI, CF, BG	D1 (Ø)
D G I (C26)	BE, ABC; BF, CH, AF, AH	
D H I	BE, AC, AE; CH, AH, CF, BG, EF, BI, CI	D2 (DI)
E F I	AE; CF, BG, BI, CI	D2 (EF)
E G H	AF; AB, BF, DI, BE, AC, CH, AH	D1 (G)
E G I (C27)	BC	

(Table B.3 Contd.)

Table B.3 Contd.

E H I	AF; AB, BF, DI, BE, AC, CH, AH	D1 (Ø)
G H I (C28)	BC;	
A B	BF, DI; AF, BE, AC, CH, AH	E (ABC, ABF)
A C	CH; AH, BE	E (ACH)
A E (C29)	CF; BG, EF, BI, I	
A F	AB; BF, DI, BE, AC, CH, AH	E (ABF)
A H	BE, AC; CH	E (ACH)
B C (C30)	Ø	
B E (C31)	Ø	
B F	AF; AB, DI, BE, AC, CH, AH	E (ABF, BEF)
B G	EF, BI; CI	E (BCGI, BEG, BGH)
B H (C32)	EF;	
B I	EF, CI;	E (BCGI, BHI)
C F (C33)	BG; EF, BI, CI	
C G (C34)	Ø	
C H	AH; BE, AC	E (ACH, CGH, CHI)
CI (C35)	Ø	
D E	EF;	D1 (D)
D F	AB, BF; DI, AF, BE, AC, CH, AH	D1 (Ø)
D G	BE, AB; BF, DI, AF, AC, CH, AH	D1 (G)
D H (C36)	BE, AE; CF, BG, EF, BI, CI	
D I (C37)	BE, AC; CH, AH	
E F (C38)	Ø	
E G (C39)	Ø	
E H	AF; AB, BF, DI, BE, AC, CH, AH	D1 (Ø)
E I (C40)	Ø	
F I	AE; CF, BG, EF, BI, CI	D1 (Ø)
G H (C41)	Ø	
G I	BC;	E (CGI, EGI, GHI)
H I (C42)	Ø	
A (C43)	Ø	
B	Ø	E (BC, BE)
C	Ø	E (BC, CG, CI)
D (44)	Ø	
E	Ø	E (BE, EF, EG, EI)
F	Ø	E (EF)
G	Ø	E (CG, EG, GH)
H	Ø	E(GH, HI)
I	Ø	E (CI, EI, HI)

Table B.4 *Implied (I) and Unimplied (U) Compatibles for the Machine of Table B.1*

B	I							
C	I	I						
D								
E	I	I	U					
F	I	I	I	U	I			
G	I	U	U	U				
H	I	U	I	U	U	U		
I		I	I	I	U	U	U	U
	A	B	C	D	E	F	G	H
MICs: ABCF, ABEF, ACH, BCI, BG, DI MUCs: BH, CG, DEGH, DF, EGHI, FI								

Appendix C: Deleted and Excluded Compatibles (For the Machine of Table 9.32)

The working for the derivation of symbolic compatibles for the machine of Table 9.32 is shown in this Appendix. In Table C.1 are listed all the deleted compatibles together with their closure class sets. The states of a compatible which are covered in its closure class set are underlined. The symbol D1 or D2 in the remarks column indicates as to which of the two Deletion Theorems led to the deletion of the corresponding compatible. The compatible in parentheses is the symbolic compatible which represents the deleted cover. Table C.2 shows the excluded basic compatibles. The symbol E in the remarks column stands for “excluded by” and the compatible(s) in parentheses are the excluding compatibles. Table C.3 shows the implied and unimplied compatibles, while Table C.4 indicates the MICs and MUCs which serve as a reference in the process of deletion of compatibles.

Table C.1 *Deleted Primary Compatibles for the Machine of Table 5.1*

PC (C_i)	**Closure Class Set (E_i)**	**Remarks**
$\underline{C}$ G $\underline{H}$ I	GH, AC, BC, BE, AD; AF, BH, DF, EF	D2 (GHI)
B $\underline{C}$ H	EF, AC; DF	D2 (BH)*
$\underline{C}$ $\underline{F}$ $\underline{H}$	AC, BE, AD, AF; DF, BH	D2 (FH)
$\underline{C}$ $\underline{G}$ $\underline{H}$	FH, AC, GI, AD; AF, BH, DF, BE	D2 (GH)
$\underline{C}$ $\underline{G}$ I	FH, BC, GH, BE, AD; AF, BH, AC, EF, DF	D2 (GI)
$\underline{C}$ H I	AC, BC, BE, AD; DF, EF	D1 (HI)
B $\underline{C}$	EF, AC; DF	D1 (B)*
$\underline{C}$ H	AC; DF	D1 (H)
$\underline{C}$ I	BC; AC, EF, DF	D1 (I)
	*Deleted in 2nd iteration	

Table C.2 *Excluded Basic Compatibles for the Machine of Table 5.1*

PC (C_i)	Closure Class Set (E_i)	Remarks
AF	BH;	E (ADF)
DE	Ø	E (DEF)
DF	Ø	E (DEF)
EF	Ø	E (DEF)
GH	GI, AD; BE	E (GHI)
GI	GH, BE; AD	E (GHI)
HI	BE, AD;	E (GHI)
A	Ø	E (AD)
B	Ø	E (BE, BH)
D	Ø	E (DEF, AD)
E	Ø	E (DEF, BE)
F	Ø	E (DEF)
H	Ø	E (BH)

Table C.3 *Implied (I) and Unimplied (U) Compatibles for the machine of Table 5.1*

B	U							
C	I	I						
D	I							
E		I		I				
F	I		U	I	I			
G			U					
H		I	U			I	I	
I		I	U				I	U
	A	B	C	D	E	F	G	H

Table C.4 *MICs and MUCs for the Machine of Table 5.1*

Iteration No.	**Maximal Implied Compatibles**	**Maximal Unimplied Compatibles**
1	AC, ADF, BC, BE, BH, DEF, FH, GH, GI	AB, CF, CG, CHI
2 (bc → UC)	AC, ADF, BE, BH, DEF, FH, GH, GI	AB, BC, CF, CG, CHI
3 (ac → UC)	ADF, BE, BH, DE, FH, GH; GI No further deletions are possible	ABC, CF, CG, CHI
After Exclusions gh, gi → UC'S)	No further reduction is possible	ABC, CF, CGHI

References

1. Bennetts, R.C., J.L. Washington, and D.W. Lewin (1972). “A Computer Algorithm for State Table Reductions”, *IERE Radio and Electron. Engg.*, Vol. 42, November, pp. 513–520.
2. Booth, T.L. (1967). *Sequential Machines and Automata Theory.* New York: John Wiley.
3. Hill, P.J. and G.R. Peterson (1981). *Introduction to Switching Theory and Logical Design.* New York: Wiley.
4. Holdsworth, B. (1987). *Digital Logic Design*. New Delhi: Butherworth & Co.
5. Jain, R.P. (1984). *Modern Digital Electronics.* New Delhi: TMH.
6. Kohavi, Z. (1978). *Switching and Finite Automata Theory.* New Delhi: TMH.
7. Luccio, F. (1969). “Extending the Definition of Prime Compatibility Classes of States in Incomplete Sequential Machine Reduction”, *IEEE Trans. Compu.*, Vol. C-18, June, pp. 537–540.
8. Malvino, A.P. and Leach, D.P. (1986). *Digital Principles and Application.* New Delhi: TMH.
9. Markus, M.P. (1967). *Switching Circuits for Engineers*. Englewood Cliffs, N.J.: Prentice-Hall.
10. McCluskey, E.J. (1965). *Introduction to the Theory of Switching Circuits.* New York: McGraw-Hill.
11. Pager, D. (1971). “Conditions for the Existence of Minimal Closed Covers Composed of Maximal Compatibles”, *IEEE Trans. Comput.* (Short Notes), Vol. C-20, April, pp. 450–452.
12. Rao, C.V.S. and N.N. Biswas. (1975). “Minimization of Incompletely Specified Sequential Machines”, *IEEE Trans. Comput*, Vol. C-24, November, pp. 1089–1100.
13. Rao, C.V.S. and N.N. Biswas (1976). “On the Minimization of Incomplete Sequential Machines Using Compatibility Graph”, B-14, Proc. Computer Society of India Annual Convention, 20-23 January, Hyderabad, India.
14. Rao, C.V.S. and N.N. Biswas (1976). “Reduction of Symbolic Compatible Set in Incompletely Specified Sequential Machines”, B-13, Proc. Computer Society of India Annual Convention, 20-23 January, Hyderabad, India.
15. Rao, C.V.S. and N.N. Biswas (1976). Further Comments on “Closure Partition Method for Minimizing Incomplete Sequential Machines”, *IEEE Trans. Comput.*, July.
16. Roth, C.H. Jr. (1999). *Fundamentals of Logic Design.* New Delhi: Jaico.

17. Torng, H.C. (1972). *Switching Circuits: Theory and Logic Design.* Reading, Massachusetts: Addison-Wesley Publishing Company.

18. Unger, S.H. (1969). *Asynchronous Sequential Circuits.* New York: Wiley-Interscience.

19. Wakerly, J.F. (2001). *Digital Design Principles and Practices.* New Delhi: Pearson Education.

Index