Digital Electronics

Principles, Devices and Applications

Anil K. Maini

WILEY

Digital Electronics

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Digital Electronics

Principles, Devices and Applications

Anil K. Maini
Defence Research and Development Organization (DRDO), India

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, ONT, Canada L5R 4J3

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Anniversary Logo Design: Richard J. Pacifico

Library of Congress Cataloging in Publication Data

Maini, Anil Kumar.
Digital electronics : principles, devices, and applications / Anil Kumar Maini.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-03214-5 (Cloth)
1. Digital electronics. I. Title.
TK7868.D5M275 2007
621.381—dc22 2007020666

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-470-03214-5 (HB)

Typeset in 9/11pt Times by Integra Software Services Pvt. Ltd, Pondicherry, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

In the loving memory of my father, Shri Sukhdev Raj Maini*, who has been a source of inspiration, courage and strength to me to face all challenges in life, and above all instilled in me the value of helping people to make this world a better place.*

Anil K. Maini

Contents

Preface

Digital electronics is essential to understanding the design and working of a wide range of applications, from consumer and industrial electronics to communications; from embedded systems, and computers to security and military equipment. As the devices used in these applications decrease in size and employ more complex technology, it is essential for engineers and students to fully understand both the fundamentals and also the implementation and application principles of digital electronics, devices and integrated circuits, thus enabling them to use the most appropriate and effective technique to suit their technical needs.

Digital Electronics: Principles, Devices and Applications is a comprehensive book covering, in one volume, both the fundamentals of digital electronics and the applications of digital devices and integrated circuits. It is different from similar books on the subject in more than one way. Each chapter in the book, whether it is related to operational fundamentals or applications, is amply illustrated with diagrams and design examples. In addition, the book covers several new topics, which are of relevance to any one having an interest in digital electronics and not covered in the books already in print on the subject. These include digital troubleshooting, digital instrumentation, programmable logic devices, microprocessors and microcontrollers. While the book covers in entirety what is required by undergraduate and graduate level students of engineering in electrical, electronics, computer science and information technology disciplines, it is intended to be a very useful reference book for professionals, R&D scientists and students at post graduate level.

The book is divided into sixteen chapters covering seven major topics. These are: *digital electronics fundamentals* (chapters 1 to 6), *combinational logic circuits* (chapters 7 and 8), *programmable logic devices* (chapter 9), *sequential logic circuits* (chapters 10 and 11), *data conversion devices and circuits* (chapter 12), *microprocessors, microcontrollers and microcomputers* (chapters 13 to 15) and *digital troubleshooting and instrumentation* (chapter 16). The contents of each of the sixteen chapters are briefly described in the following paragraphs.

The first six chapters deal with the fundamental topics of digital electronics. These include different number systems that can be used to represent data and binary codes used for representing numeric and alphanumeric data. Conversion from one number system to another and similarly conversion from one code to another is discussed at length in these chapters. Binary arithmetic, covering different methods of performing arithmetic operations on binary numbers is discussed next. Chapters four and five cover logic gates and logic families. The main topics covered in these two chapters are various logic gates and related devices, different logic families used to hardware implement digital integrated circuits, the interface between digital ICs belonging to different logic families and application information such as guidelines for using logic devices of different families. Boolean algebra and its various postulates and theorems and minimization techniques, providing exhaustive coverage of both Karnaugh mapping and Quine-McCluskey techniques, are discussed in chapter six. The discussion includes application of these minimization techniques for multi-output Boolean functions and Boolean functions with larger number of variables. The concepts underlying different fundamental topics of digital electronics and discussed in first six chapters have been amply illustrated with solved examples.

As a follow-up to logic gates – the most basic building block of combinational logic – chapters 7 and 8 are devoted to more complex combinational logic circuits. While chapter seven covers arithmetic circuits, including different types of adders and subtractors, such as half and full adder and subtractor, adder-subtractor, larger bit adders and subtractors, multipliers, look ahead carry generator, magnitude comparator, and arithmetic logic unit, chapter eight covers multiplexers, de-multiplexers, encoders and decoders. This is followed by a detailed account of programmable logic devices in chapter nine. Simple programmable logic devices (SPLDs) such as PAL, PLA, GAL and HAL devices, complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs) have been exhaustively treated in terms of their architecture, features and applications. Popular devices, from various international manufacturers, in the three above-mentioned categories of programmable logic devices are also covered with regard to their architecture, features and facilities.

The next two chapters, 10 and 11, cover the sequential logic circuits. Discussion begins with the most fundamental building block of sequential logic, that is, *flip flop*. Different types of flip flops are covered in detail with regard to their operational fundamentals, different varieties in each of the categories of flip flops and their applications. Multivibrator circuits, being operationally similar to flip flops, are also covered at length in this chapter. Counters and registers are the other very important building blocks of sequential logic with enormous application potential. These are covered in chapter 11. Particular emphasis is given to timing requirements and design of counters with varying count sequence requirements. The chapter also includes a detailed description of the design principles of counters with arbitrary count sequences. Different types of shift registers and some special counters that have evolved out of shift registers have been covered in detail.

Chapter 12 covers data conversion circuits including digital-to-analogue and analogue-to-digital converters. Topics covered in this chapter include operational basics, characteristic parameters, types and applications. Emphasis is given to definition and interpretation of the terminology and the performance parameters that characterize these devices. Different types of digital-to-analogue and analogue-to-digital converters, together with their merits and drawbacks are also addressed. Particular attention is given to their applications. Towards the end of the chapter, application oriented information in the form of popular type numbers along with their major performance specifications, pin connection diagrams etc. is presented. Another highlight of the chapter is the inclusion of detailed descriptions of newer types of converters, such as quad slope and sigma-delta types of analogue-to-digital converters.

Chapters 13 and 14 discuss microprocessors and microcontrollers – the two versatile devices that have revolutionized the application potential of digital devices and integrated circuits. The entire range of microprocessors and microcontrollers along with their salient features, operational aspects and application guidelines are covered in detail. As a natural follow-up to these, microcomputer fundamentals, with regard to their architecture, input/output devices and memory devices, are discussed in chapter 15.

The last chapter covers digital troubleshooting techniques and digital instrumentation. Troubleshooting guidelines for various categories of digital electronics circuits are discussed. These will particularly benefit practising engineers and electronics enthusiasts. The concepts are illustrated with the help of a large number of troubleshooting case studies pertaining to combinational, sequential and memory devices. A wide range of digital instruments is covered after a discussion on troubleshooting guidelines. The instruments covered include digital multimeters, digital oscilloscopes, logic probes, logic analysers, frequency synthesizers, and synthesized function generators. Computer-instrument interface standards and the concept of virtual instrumentation are also discussed at length towards the end of the chapter.

As an extra resource, a companion website for my book contains lot of additional application relevant information on digital devices and integrated circuits. The information on this website includes numerical and functional indices of digital integrated circuits belonging to different logic families, pin connection diagrams and functional tables of different categories of general purpose digital integrated circuits and application relevant information on microprocessors, peripheral devices and microcontrollers. Please go to URL http://www.wiley.com/go/maini_digital.

The motivation to write this book and the selection of topics to be covered were driven mainly by the absence a book, which, in one volume, covers all the important aspects of digital technology. A large number of books in print on the subject cover all the routine topics of digital electronics in a conventional way with total disregard to the needs of application engineers and professionals. As the author, I have made an honest attempt to cover the subject in entirety by including comprehensive treatment of newer topics that are either ignored or inadequately covered in the available books on the subject of digital electronics. This is done keeping in view the changed requirements of my intended audience, which includes undergraduate and graduate level students, R&D scientists, professionals and application engineers.

Anil K. Maini

1

Number Systems

The study of *number systems* is important from the viewpoint of understanding how data are represented before they can be processed by any digital system including a digital computer. It is one of the most basic topics in digital electronics. In this chapter we will discuss different number systems commonly used to represent data. We will begin the discussion with the decimal number system. Although it is not important from the viewpoint of digital electronics, a brief outline of this will be given to explain some of the underlying concepts used in other number systems. This will then be followed by the more commonly used number systems such as the binary, octal and hexadecimal number systems.

1.1 Analogue Versus Digital

There are two basic ways of representing the numerical values of the various physical quantities with which we constantly deal in our day-to-day lives. One of the ways, referred to as *analogue*, is to express the numerical value of the quantity as a continuous range of values between the two expected extreme values. For example, the temperature of an oven settable anywhere from 0 to 100 °C may be measured to be 65 °C or 64.96 °C or 64.958 °C or even 64.9579 °C and so on, depending upon the accuracy of the measuring instrument. Similarly, voltage across a certain component in an electronic circuit may be measured as 6.5 V or 6.49 V or 6.487 V or 6.4869 V. The underlying concept in this mode of representation is that variation in the numerical value of the quantity is continuous and could have any of the infinite theoretically possible values between the two extremes.

The other possible way, referred to as *digital*, represents the numerical value of the quantity in steps of discrete values. The numerical values are mostly represented using binary numbers. For example, the temperature of the oven may be represented in steps of 1 °C as 64 °C, 65 °C, 66 °C and so on. To summarize, while an analogue representation gives a continuous output, a digital representation produces a discrete output. Analogue systems contain devices that process or work on various physical quantities represented in analogue form. Digital systems contain devices that process the physical quantities represented in digital form.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Digital techniques and systems have the advantages of being relatively much easier to design and having higher accuracy, programmability, noise immunity, easier storage of data and ease of fabrication in integrated circuit form, leading to availability of more complex functions in a smaller size. The real world, however, is analogue. Most physical quantities – position, velocity, acceleration, force, pressure, temperature and flowrate, for example – are analogue in nature. That is why analogue variables representing these quantities need to be digitized or discretized at the input if we want to benefit from the features and facilities that come with the use of digital techniques. In a typical system dealing with analogue inputs and outputs, analogue variables are digitized at the input with the help of an analogue-to-digital converter block and reconverted back to analogue form at the output using a digital-to-analogue converter block. Analogue-to-digital and digital-to-analogue converter circuits are discussed at length in the latter part of the book. In the following sections we will discuss various number systems commonly used for digital representation of data.

1.2 Introduction to Number Systems

We will begin our discussion on various number systems by briefly describing the parameters that are common to all number systems. An understanding of these parameters and their relevance to number systems is fundamental to the understanding of how various systems operate. Different characteristics that define a number system include the number of independent digits used in the number system, the place values of the different digits constituting the number and the maximum numbers that can be written with the given number of digits. Among the three characteristic parameters, the most fundamental is the number of independent digits or symbols used in the number system. It is known as the *radix* or *base* of the number system. The decimal number system with which we are all so familiar can be said to have a radix of 10 as it has 10 independent digits, i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Similarly, the binary number system with only two independent digits, 0 and 1, is a radix-2 number system. The octal and hexadecimal number systems have a radix (or base) of 8 and 16 respectively. We will see in the following sections that the radix of the number system also determines the other two characteristics. The place values of different digits in the integer part of the number are given by r^0, r^1, r^2, r^3 and so on, starting with the digit adjacent to the radix point. For the fractional part, these are r^{-1}, r^{-2}, r^{-3} and so on, again starting with the digit next to the radix point. Here, r is the radix of the number system. Also, maximum numbers that can be written with n digits in a given number system are equal to r^n.

1.3 Decimal Number System

The decimal number system is a radix-10 number system and therefore has 10 different digits or symbols. These are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All higher numbers after '9' are represented in terms of these 10 digits only. The process of writing higher-order numbers after '9' consists in writing the second digit (i.e. '1') first, followed by the other digits, one by one, to obtain the next 10 numbers from '10' to '19'. The next 10 numbers from '20' to '29' are obtained by writing the third digit (i.e. '2') first, followed by digits '0' to '9', one by one. The process continues until we have exhausted all possible two-digit combinations and reached '99'. Then we begin with three-digit combinations. The first three-digit number consists of the lowest two-digit number followed by '0' (i.e. 100), and the process goes on endlessly.

The place values of different digits in a mixed decimal number, starting from the decimal point, are 10^0, 10^1, 10^2 and so on (for the integer part) and 10^{-1}, 10^{-2}, 10^{-3} and so on (for the fractional part).

The value or magnitude of a given decimal number can be expressed as the sum of the various digits multiplied by their place values or weights.

As an illustration, in the case of the decimal number 3586.265, the integer part (i.e. 3586) can be expressed as

$$3586 = 6 \times 10^0 + 8 \times 10^1 + 5 \times 10^2 + 3 \times 10^3 = 6 + 80 + 500 + 3000 = 3586$$

and the fractional part can be expressed as

$$265 = 2 \times 10^{-1} + 6 \times 10^{-2} + 5 \times 10^{-3} = 0.2 + 0.06 + 0.005 = 0.265$$

We have seen that the place values are a function of the radix of the concerned number system and the position of the digits. We will also discover in subsequent sections that the concept of each digit having a place value depending upon the position of the digit and the radix of the number system is equally valid for the other more relevant number systems.

1.4 Binary Number System

The binary number system is a radix-2 number system with '0' and '1' as the two independent digits. All larger binary numbers are represented in terms of '0' and '1'. The procedure for writing higher-order binary numbers after '1' is similar to the one explained in the case of the decimal number system. For example, the first 16 numbers in the binary number system would be 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. The next number after 1111 is 10000, which is the lowest binary number with five digits. This also proves the point made earlier that a maximum of only 16 ($= 2^4$) numbers could be written with four digits. Starting from the binary point, the place values of different digits in a mixed binary number are 2^0, 2^1, 2^2 and so on (for the integer part) and 2^{-1}, 2^{-2}, 2^{-3} and so on (for the fractional part).

Example 1.1

Consider an arbitrary number system with the independent digits as 0, 1 and X. What is the radix of this number system? List the first 10 numbers in this number system.

Solution

- The radix of the proposed number system is 3.
- The first 10 numbers in this number system would be 0, 1, X, 10, 11, 1X, X0, X1, XX and 100.

1.4.1 Advantages

Logic operations are the backbone of any digital computer, although solving a problem on computer could involve an arithmetic operation too. The introduction of the mathematics of logic by George Boole laid the foundation for the modern digital computer. He reduced the mathematics of logic to a binary notation of '0' and '1'. As the mathematics of logic was well established and had proved itself to be quite useful in solving all kinds of logical problem, and also as the mathematics of logic (also known as Boolean algebra) had been reduced to a binary notation, the binary number system had a clear edge over other number systems for use in computer systems.

Yet another significant advantage of this number system was that all kinds of data could be conveniently represented in terms of 0s and 1s. Also, basic electronic devices used for hardware implementation could be conveniently and efficiently operated in two distinctly different modes. For example, a bipolar transistor could be operated either in cut-off or in saturation very efficiently.

Lastly, the circuits required for performing arithmetic operations such as addition, subtraction, multiplication, division, etc., become a simple affair when the data involved are represented in the form of 0s and 1s.

1.5 Octal Number System

The octal number system has a radix of 8 and therefore has eight distinct digits. All higher-order numbers are expressed as a combination of these on the same pattern as the one followed in the case of the binary and decimal number systems described in Sections 1.3 and 1.4. The independent digits are 0, 1, 2, 3, 4, 5, 6 and 7. The next 10 numbers that follow '7', for example, would be 10, 11, 12, 13, 14, 15, 16, 17, 20 and 21. In fact, if we omit all the numbers containing the digits 8 or 9, or both, from the decimal number system, we end up with an octal number system. The place values for the different digits in the octal number system are 8^0, 8^1, 8^2 and so on (for the integer part) and 8^{-1}, 8^{-2}, 8^{-3} and so on (for the fractional part).

1.6 Hexadecimal Number System

The hexadecimal number system is a radix-16 number system and its 16 basic digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The place values or weights of different digits in a mixed hexadecimal number are 16^0, 16^1, 16^2 and so on (for the integer part) and 16^{-1}, 16^{-2}, 16^{-3} and so on (for the fractional part). The decimal equivalent of A, B, C, D, E and F are 10, 11, 12, 13, 14 and 15 respectively, for obvious reasons.

The hexadecimal number system provides a condensed way of representing large binary numbers stored and processed inside the computer. One such example is in representing addresses of different memory locations. Let us assume that a machine has 64K of memory. Such a memory has 64K ($= 2^{16} = 65\,536$) memory locations and needs 65 536 different addresses. These addresses can be designated as 0 to 65 535 in the decimal number system and 00000000 00000000 to 11111111 11111111 in the binary number system. The decimal number system is not used in computers and the binary notation here appears too cumbersome and inconvenient to handle. In the hexadecimal number system, 65 536 different addresses can be expressed with four digits from 0000 to FFFF. Similarly, the contents of the memory when represented in hexadecimal form are very convenient to handle.

1.7 Number Systems – Some Common Terms

In this section we will describe some commonly used terms with reference to different number systems.

1.7.1 Binary Number System

Bit is an abbreviation of the term 'binary digit' and is the smallest unit of information. It is either '0' or '1'. A *byte* is a string of eight bits. The byte is the basic unit of data operated upon as a single unit in computers. A *computer word* is again a string of bits whose size, called the 'word length' or 'word size', is fixed for a specified computer, although it may vary from computer to computer. The word length may equal one byte, two bytes, four bytes or be even larger.

The *1's complement* of a binary number is obtained by complementing all its bits, i.e. by replacing 0s with 1s and 1s with 0s. For example, the 1's complement of $(10010110)_2$ is $(01101001)_2$. The *2's complement* of a binary number is obtained by adding '1' to its 1's complement. The 2's complement of $(10010110)_2$ is $(01101010)_2$.

1.7.2 Decimal Number System

Corresponding to the 1's and 2's complements in the binary system, in the decimal number system we have the 9's and 10's complements. The *9's complement* of a given decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of $(2496)_{10}$ would be $(7503)_{10}$. The *10's complement* is obtained by adding '1' to the 9's complement. The 10's complement of $(2496)_{10}$ is $(7504)_{10}$.

1.7.3 Octal Number System

In the octal number system, we have the 7's and 8's complements. The *7's complement* of a given octal number is obtained by subtracting each octal digit from 7. For example, the 7's complement of $(562)_8$ would be $(215)_8$. The *8's complement* is obtained by adding '1' to the 7's complement. The 8's complement of $(562)_8$ would be $(216)_8$.

1.7.4 Hexadecimal Number System

The 15's and 16's complements are defined with respect to the hexadecimal number system. The *15's complement* is obtained by subtracting each hex digit from 15. For example, the 15's complement of $(3BF)_{16}$ would be $(C40)_{16}$. The *16's complement* is obtained by adding '1' to the 15's complement. The 16's complement of $(2AE)_{16}$ would be $(D52)_{16}$.

1.8 Number Representation in Binary

Different formats used for binary representation of both positive and negative decimal numbers include the sign-bit magnitude method, the 1's complement method and the 2's complement method.

1.8.1 Sign-Bit Magnitude

In the sign-bit magnitude representation of positive and negative decimal numbers, the MSB represents the 'sign', with a '0' denoting a plus sign and a '1' denoting a minus sign. The remaining bits represent the magnitude. In eight-bit representation, while MSB represents the sign, the remaining seven bits represent the magnitude. For example, the eight-bit representation of $+9$ would be 00001001, and that for -9 would be 10001001. An $n-$bit binary representation can be used to represent decimal numbers in the range of $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. That is, eight-bit representation can be used to represent decimal numbers in the range from -127 to $+127$ using the sign-bit magnitude format.

1.8.2 1's Complement

In the 1's complement format, the positive numbers remain unchanged. The negative numbers are obtained by taking the 1's complement of the positive counterparts. For example, +9 will be represented as 00001001 in eight-bit notation, and −9 will be represented as 11110110, which is the 1's complement of 00001001. Again, n-bit notation can be used to represent numbers in the range from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ using the 1's complement format. The eight-bit representation of the 1's complement format can be used to represent decimal numbers in the range from −127 to +127.

1.8.3 2's Complement

In the 2's complement representation of binary numbers, the MSB represents the sign, with a '0' used for a plus sign and a '1' used for a minus sign. The remaining bits are used for representing magnitude. Positive magnitudes are represented in the same way as in the case of sign-bit or 1's complement representation. Negative magnitudes are represented by the 2's complement of their positive counterparts. For example, +9 would be represented as 00001001, and −9 would be written as 11110111. Please note that, if the 2's complement of the magnitude of +9 gives a magnitude of −9, then the reverse process will also be true, i.e. the 2's complement of the magnitude of −9 will give a magnitude of +9. The n-bit notation of the 2's complement format can be used to represent all decimal numbers in the range from $+(2^{n-1}-1)$ to $-(2^{n-1})$. The 2's complement format is very popular as it is very easy to generate the 2's complement of a binary number and also because arithmetic operations are relatively easier to perform when the numbers are represented in the 2's complement format.

1.9 Finding the Decimal Equivalent

The decimal equivalent of a given number in another number system is given by the sum of all the digits multiplied by their respective place values. The integer and fractional parts of the given number should be treated separately. Binary-to-decimal, octal-to-decimal and hexadecimal-to-decimal conversions are illustrated below with the help of examples.

1.9.1 Binary-to-Decimal Conversion

The decimal equivalent of the binary number $(1001.0101)_2$ is determined as follows:

- The integer part = 1001
- The decimal equivalent $= 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 + 0 + 0 + 8 = 9$
- The fractional part = .0101
- Therefore, the decimal equivalent $= 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0 + 0.25 + 0 + 0.0625 = 0.3125$
- Therefore, the decimal equivalent of $(1001.0101)_2 = 9.3125$

1.9.2 Octal-to-Decimal Conversion

The decimal equivalent of the octal number $(137.21)_8$ is determined as follows:

- The integer part = 137
- The decimal equivalent $= 7 \times 8^0 + 3 \times 8^1 + 1 \times 8^2 = 7 + 24 + 64 = 95$

- The fractional part = .21
- The decimal equivalent = $2 \times 8^{-1} + 1 \times 8^{-2} = 0.265$
- Therefore, the decimal equivalent of $(137.21)_8 = (95.265)_{10}$

1.9.3 Hexadecimal-to-Decimal Conversion

The decimal equivalent of the hexadecimal number $(1E0.2A)_{16}$ is determined as follows:

- The integer part = 1E0
- The decimal equivalent = $0 \times 16^0 + 14 \times 16^1 + 1 \times 16^2 = 0 + 224 + 256 = 480$
- The fractional part = 2A
- The decimal equivalent = $2 \times 16^{-1} + 10 \times 16^{-2} = 0.164$
- Therefore, the decimal equivalent of $(1E0.2A)_{16} = (480.164)_{10}$

Example 1.2

Find the decimal equivalent of the following binary numbers expressed in the 2's complement format:

(a) 00001110;
(b) 10001110.

Solution

(a) The MSB bit is '0', which indicates a plus sign.
The magnitude bits are 0001110.
The decimal equivalent $= 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 0 \times 2^6$
$= 0 + 2 + 4 + 8 + 0 + 0 + 0 = 14$

Therefore, 00001110 represents +14

(b) The MSB bit is '1', which indicates a minus sign
The magnitude bits are therefore given by the 2's complement of 0001110, i.e. 1110010
The decimal equivalent $= 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5$
$+ 1 \times 2^6$
$= 0 + 2 + 0 + 0 + 16 + 32 + 64 = 114$

Therefore, 10001110 represents −114

1.10 Decimal-to-Binary Conversion

As outlined earlier, the integer and fractional parts are worked on separately. For the integer part, the binary equivalent can be found by successively dividing the integer part of the number by 2 and recording the remainders until the quotient becomes '0'. The remainders written in reverse order constitute the binary equivalent. For the fractional part, it is found by successively multiplying the fractional part of the decimal number by 2 and recording the carry until the result of multiplication is '0'. The carry sequence written in forward order constitutes the binary equivalent of the fractional part of the decimal number. If the result of multiplication does not seem to be heading towards zero in the case of the fractional part, the process may be continued only until the requisite number of equivalent bits has been obtained. This method of decimal–binary conversion is popularly known as the double-dabble method. The process can be best illustrated with the help of an example.

Example 1.3

We will find the binary equivalent of $(13.375)_{10}$.

Solution

- The integer part = 13

Divisor	Dividend	Remainder
2	13	—
2	6	1
2	3	0
2	1	1
—	0	1

- The binary equivalent of $(13)_{10}$ is therefore $(1101)_2$
- The fractional part = .375
- $0.375 \times 2 = 0.75$ with a carry of 0
- $0.75 \times 2 = 0.5$ with a carry of 1
- $0.5 \times 2 = 0$ with a carry of 1
- The binary equivalent of $(0.375)_{10} = (.011)_2$
- Therefore, the binary equivalent of $(13.375)_{10} = (1101.011)_2$

1.11 Decimal-to-Octal Conversion

The process of decimal-to-octal conversion is similar to that of decimal-to-binary conversion. The progressive division in the case of the integer part and the progressive multiplication while working on the fractional part here are by '8' which is the radix of the octal number system. Again, the integer and fractional parts of the decimal number are treated separately. The process can be best illustrated with the help of an example.

Example 1.4

We will find the octal equivalent of $(73.75)_{10}$.

Solution

- The integer part = 73

Divisor	Dividend	Remainder
8	73	—
8	9	1
8	1	1
—	0	1

- The octal equivalent of $(73)_{10} = (111)_8$
- The fractional part = 0.75
- $0.75 \times 8 = 0$ with a carry of 6
- The octal equivalent of $(0.75)_{10} = (.6)_8$
- Therefore, the octal equivalent of $(73.75)_{10} = (111.6)_8$

1.12 Decimal-to-Hexadecimal Conversion

The process of decimal-to-hexadecimal conversion is also similar. Since the hexadecimal number system has a base of 16, the progressive division and multiplication factor in this case is 16. The process is illustrated further with the help of an example.

Example 1.5

Let us determine the hexadecimal equivalent of $(82.25)_{10}$.

Solution

- The integer part = 82

Divisor	Dividend	Remainder
16	82	—
16	5	2
—	0	5

- The hexadecimal equivalent of $(82)_{10} = (52)_{16}$
- The fractional part = 0.25
- $0.25 \times 16 = 0$ with a carry of 4
- Therefore, the hexadecimal equivalent of $(82.25)_{10} = (52.4)_{16}$

1.13 Binary–Octal and Octal–Binary Conversions

An octal number can be converted into its binary equivalent by replacing each octal digit with its three-bit binary equivalent. We take the three-bit equivalent because the base of the octal number system is 8 and it is the third power of the base of the binary number system, i.e. 2. All we have then to remember is the three-bit binary equivalents of the basic digits of the octal number system. A binary number can be converted into an equivalent octal number by splitting the integer and fractional parts into groups of three bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example 1.6

Let us find the binary equivalent of $(374.26)_8$ and the octal equivalent of $(1110100.0100111)_2$.

Solution

- The given octal number = $(374.26)_8$
- The binary equivalent = $(011\ 111\ 100.010\ 110)_2 = (011111100.010110)_2$

- Any 0s on the extreme left of the integer part and extreme right of the fractional part of the equivalent binary number should be omitted. Therefore, $(011111100.010110)_2 = (11111100.01011)_2$
- The given binary number $= (1110100.0100111)_2$
- $(1110100.0100111)_2 = (1\ 110\ 100.010\ 011\ 1)_2$
  $= (001\ 110\ 100.010\ 011\ 100)_2 = (164.234)_8$

1.14 Hex–Binary and Binary–Hex Conversions

A hexadecimal number can be converted into its binary equivalent by replacing each hex digit with its four-bit binary equivalent. We take the four-bit equivalent because the base of the hexadecimal number system is 16 and it is the fourth power of the base of the binary number system. All we have then to remember is the four-bit binary equivalents of the basic digits of the hexadecimal number system. A given binary number can be converted into an equivalent hexadecimal number by splitting the integer and fractional parts into groups of four bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example 1.7

Let us find the binary equivalent of $(17E.F6)_{16}$ *and the hex equivalent of* $(1011001110.011011101)_2$.

Solution

- The given hex number $= (17E.F6)_{16}$
- The binary equivalent $= (0001\ 0111\ 1110.1111\ 0110)_2$
  $= (000101111110.11110110)_2$
  $= (101111110.1111011)_2$
- The 0s on the extreme left of the integer part and on the extreme right of the fractional part have been omitted.
- The given binary number $= (1011001110.011011101)_2$
  $= (10\ 1100\ 1110.0110\ 1110\ 1)_2$
- The hex equivalent $= (0010\ 1100\ 1110.0110\ 1110\ 1000)_2 = (2CE.6E8)_{16}$

1.15 Hex–Octal and Octal–Hex Conversions

For hexadecimal–octal conversion, the given hex number is firstly converted into its binary equivalent which is further converted into its octal equivalent. An alternative approach is firstly to convert the given hexadecimal number into its decimal equivalent and then convert the decimal number into an equivalent octal number. The former method is definitely more convenient and straightforward. For octal–hexadecimal conversion, the octal number may first be converted into an equivalent binary number and then the binary number transformed into its hex equivalent. The other option is firstly to convert the given octal number into its decimal equivalent and then convert the decimal number into its hex equivalent. The former approach is definitely the preferred one. Two types of conversion are illustrated in the following example.

Example 1.8

Let us find the octal equivalent of $(2F.C4)_{16}$ *and the hex equivalent of* $(762.013)_8$.

Solution

- The given hex number = $(2F.C4)_{16}$.
- The binary equivalent = $(0010\ 1111.1100\ 0100)_2 = (00101111.11000100)_2$
  $= (101111.110001)_2 = (101\ 111.110\ 001)_2 = (57.61)_8$.
- The given octal number = $(762.013)_8$.
- The octal number = $(762.013)_8 = (111\ 110\ 010.000\ 001\ 011)_2$
  $= (111110010.000001011)_2$
  $= (0001\ 1111\ 0010.0000\ 0101\ 1000)_2 = (1F2.058)_{16}$.

1.16 The Four Axioms

Conversion of a given number in one number system to its equivalent in another system has been discussed at length in the preceding sections. The methodology has been illustrated with solved examples. The complete methodology can be summarized as four axioms or principles, which, if understood properly, would make it possible to solve any problem related to conversion of a given number in one number system to its equivalent in another number system. These principles are as follows:

1. Whenever it is desired to find the decimal equivalent of a given number in another number system, it is given by the sum of all the digits multiplied by their weights or place values. The integer and fractional parts should be handled separately. Starting from the radix point, the weights of different digits are r^0, r^1, r^2 for the integer part and r^{-1}, r^{-2}, r^{-3} for the fractional part, where r is the radix of the number system whose decimal equivalent needs to be determined.
2. To convert a given mixed decimal number into an equivalent in another number system, the integer part is progressively divided by r and the remainders noted until the result of division yields a zero quotient. The remainders written in reverse order constitute the equivalent. r is the radix of the transformed number system. The fractional part is progressively multiplied by r and the carry recorded until the result of multiplication yields a zero or when the desired number of bits has been obtained. The carrys written in forward order constitute the equivalent of the fractional part.
3. The octal–binary conversion and the reverse process are straightforward. For octal–binary conversion, replace each digit in the octal number with its three-bit binary equivalent. For hexadecimal–binary conversion, replace each hex digit with its four-bit binary equivalent. For binary–octal conversion, split the binary number into groups of three bits, starting from the binary point, and, if needed, complete the outside groups by adding 0s, and then write the octal equivalent of these three-bit groups. For binary–hex conversion, split the binary number into groups of four bits, starting from the binary point, and, if needed, complete the outside groups by adding 0s, and then write the hex equivalent of the four-bit groups.
4. For octal–hexadecimal conversion, we can go from the given octal number to its binary equivalent and then from the binary equivalent to its hex counterpart. For hexadecimal–octal conversion, we can go from the hex to its binary equivalent and then from the binary number to its octal equivalent.

Example 1.9

Assume an arbitrary number system having a radix of 5 and 0, 1, 2, L and M as its independent digits. Determine:

(a) the decimal equivalent of (12LM.L1);
(b) the total number of possible four-digit combinations in this arbitrary number system.

Solution

(a) The decimal equivalent of (12LM) is given by

$$\begin{aligned} \mathrm{M}\times 5^0+\mathrm{L}\times 5^1+2\times 5^2+1\times 5^3 &= 4\times 5^0+3\times 5^1+2\times 5^2+1\times 5^3(\mathrm{L}=3,\mathrm{M}=4)\\ &= 4+15+50+125=194 \end{aligned}$$

The decimal equivalent of (L1) is given by

$$\mathrm{L}\times 5^{-1}+1\times 5^{-2}=3\times 5^{-1}+5^{-2}=0.64$$

Combining the results, $(\mathrm{12LM.L1})_5 = (194.64)_{10}$.

(b) The total number of possible four-digit combinations $= 5^4 = 625$.

Example 1.10

The 7's complement of a certain octal number is 5264. Determine the binary and hexadecimal equivalents of that octal number.

Solution

- The 7's complement = 5264.
- Therefore, the octal number $= (2513)_8$.
- The binary equivalent $= (010\ 101\ 001\ 011)_2 = (10101001011)_2$.
- Also, $(10101001011)_2 = (101\ 0100\ 1011)_2 = (0101\ 0100\ 1011)_2 = (\mathrm{54B})_{16}$.
- Therefore, the hex equivalent of $(2513)_8 = (\mathrm{54B})_{16}$ and the binary equivalent of $(2513)_8 = (10101001011)_2$.

1.17 Floating-Point Numbers

Floating-point notation can be used conveniently to represent both large as well as small fractional or mixed numbers. This makes the process of arithmetic operations on these numbers relatively much easier. Floating-point representation greatly increases the range of numbers, from the smallest to the largest, that can be represented using a given number of digits. Floating-point numbers are in general expressed in the form

$$N = m\times b^e \tag{1.1}$$

where m is the fractional part, called the *significand* or *mantissa*, e is the integer part, called the *exponent*, and b is the *base* of the number system or numeration. Fractional part m is a p-digit number of the form $(\pm d.dddd\ldots dd)$, with each digit d being an integer between 0 and $b-1$ inclusive. If the leading digit of m is nonzero, then the number is said to be normalized.

Equation (1.1) in the case of decimal, hexadecimal and binary number systems will be written as follows:

Decimal system

$$N = m\times 10^e \tag{1.2}$$

Hexadecimal system

$$N = m \times 16^e \tag{1.3}$$

Binary system

$$N = m \times 2^e \tag{1.4}$$

For example, decimal numbers 0.0003754 and 3754 will be represented in floating-point notation as 3.754×10^{-4} and 3.754×10^3 respectively. A hex number 257.ABF will be represented as $2.57\text{ABF} \times 16^2$. In the case of normalized binary numbers, the leading digit, which is the most significant bit, is always '1' and thus does not need to be stored explicitly.

Also, while expressing a given mixed binary number as a floating-point number, the radix point is so shifted as to have the most significant bit immediately to the right of the radix point as a '1'. Both the mantissa and the exponent can have a positive or a negative value.

The mixed binary number $(110.1011)_2$ will be represented in floating-point notation as $.1101011 \times 2^3 = .1101011e+0011$. Here, .1101011 is the mantissa and $e+0011$ implies that the exponent is +3. As another example, $(0.000111)_2$ will be written as $.111e-0011$, with .111 being the mantissa and $e-0011$ implying an exponent of -3. Also, $(-0.00000101)_2$ may be written as $-.101 \times 2^{-5} = -.101e-0101$, where $-.101$ is the mantissa and $e-0101$ indicates an exponent of -5. If we wanted to represent the mantissas using eight bits, then .1101011 and .111 would be represented as .11010110 and .11100000.

1.17.1 Range of Numbers and Precision

The range of numbers that can be represented in any machine depends upon the number of bits in the exponent, while the fractional accuracy or precision is ultimately determined by the number of bits in the mantissa. The higher the number of bits in the exponent, the larger is the range of numbers that can be represented. For example, the range of numbers possible in a floating-point binary number format using six bits to represent the magnitude of the exponent would be from 2^{-64} to 2^{+64}, which is equivalent to a range of $10^{-19}$to 10^{+19}. The precision is determined by the number of bits used to represent the mantissa. It is usually represented as decimal digits of precision. The concept of precision as defined with respect to floating-point notation can be explained in simple terms as follows. If the mantissa is stored in n number of bits, it can represent a decimal number between 0 and $2^n - 1$ as the mantissa is stored as an unsigned integer. If M is the largest number such that $10^M - 1$ is less than or equal to $2^n - 1$, then M is the precision expressed as decimal digits of precision. For example, if the mantissa is expressed in 20 bits, then decimal digits of precision can be found to be about 6, as $2^{20} - 1$ equals 1 048 575, which is a little over $10^6 - 1$. We will briefly describe the commonly used formats for binary floating-point number representation.

1.17.2 Floating-Point Number Formats

The most commonly used format for representing floating-point numbers is the IEEE-754 standard. The full title of the standard is IEEE Standard for Binary Floating-point Arithmetic (ANSI/IEEE STD 754-1985). It is also known as Binary Floating-point Arithmetic for Microprocessor Systems, IEC

60559:1989. An ongoing revision to IEEE-754 is IEEE-754r. Another related standard IEEE 854-1987 generalizes IEEE-754 to cover both binary and decimal arithmetic. A brief description of salient features of the IEEE-754 standard, along with an introduction to other related standards, is given below.

ANSI/IEEE-754 Format

The IEEE-754 floating point is the most commonly used representation for real numbers on computers including Intel-based personal computers, Macintoshes and most of the UNIX platforms. It specifies four formats for representing floating-point numbers. These include single-precision, double-precision, single-extended precision and double-extended precision formats. Table 1.1 lists characteristic parameters of the four formats contained in the IEEE-754 standard. Of the four formats mentioned, the single-precision and double-precision formats are the most commonly used ones. The single-extended and double-extended precision formats are not common.

Figure 1.1 shows the basic constituent parts of the single- and double-precision formats. As shown in the figure, the floating-point numbers, as represented using these formats, have three basic components including the sign, the exponent and the mantissa. A '0' denotes a positive number and a '1' denotes a negative number. The n-bit exponent field needs to represent both positive and negative exponent values. To achieve this, a bias equal to $2^{n-1} - 1$ is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single-precision format and 1023 for an 11-bit exponent of the double-precision format. The addition of bias allows the use of an exponent in the range from -127 to $+128$, corresponding to a range of 0–255 in the first case, and in the range from -1023 to $+1024$, corresponding to a range of 0–2047 in the second case. A negative exponent is always represented in 2's complement form. The single-precision format offers a range from 2^{-127} to 2^{+127}, which is equivalent to 10^{-38} to 10^{+38}. The figures are 2^{-1023} to 2^{+1023}, which is equivalent to 10^{-308} to 10^{+308} in the case of the double-precision format.

The extreme exponent values are reserved for representing special values. For example, in the case of the single-precision format, for an exponent value of -127, the biased exponent value is zero, represented by an all 0s exponent field. In the case of a biased exponent of zero, if the mantissa is zero as well, the value of the floating-point number is exactly zero. If the mantissa is nonzero, it represents a denormalized number that does not have an assumed leading bit of '1'. A biased exponent of $+255$, corresponding to an actual exponent of $+128$, is represented by an all 1s exponent field. If the mantissa is zero, the number represents infinity. The sign bit is used to distinguish between positive and negative infinity. If the mantissa is nonzero, the number represents a 'NaN' (Not a Number). The value NaN is used to represent a value that does not represent a real number. This means that an eight-bit exponent can represent exponent values between -126 and $+127$. Referring to Fig. 1.1(a), the MSB of byte 1 indicates the sign of the mantissa. The remaining seven bits of byte 1 and the MSB of byte 2 represent an eight-bit exponent. The remaining seven bits of byte 2 and the 16 bits of byte 3 and byte 4 give a 23-bit mantissa. The mantissa m is normalized. The left-hand bit of the normalized mantissa is always

Table 1.1 Characteristic parameters of IEEE-754 formats.

Precision	Sign (bits)	Exponent (bits)	Mantissa (bits)	Total length (bits)	Decimal digits of precision
Single	1	8	23	32	>6
Single-extended	1	≥ 11	≥ 32	≥ 44	>9
Double	1	11	52	64	>15
Double-extended	1	≥ 15	≥ 64	≥ 80	>19

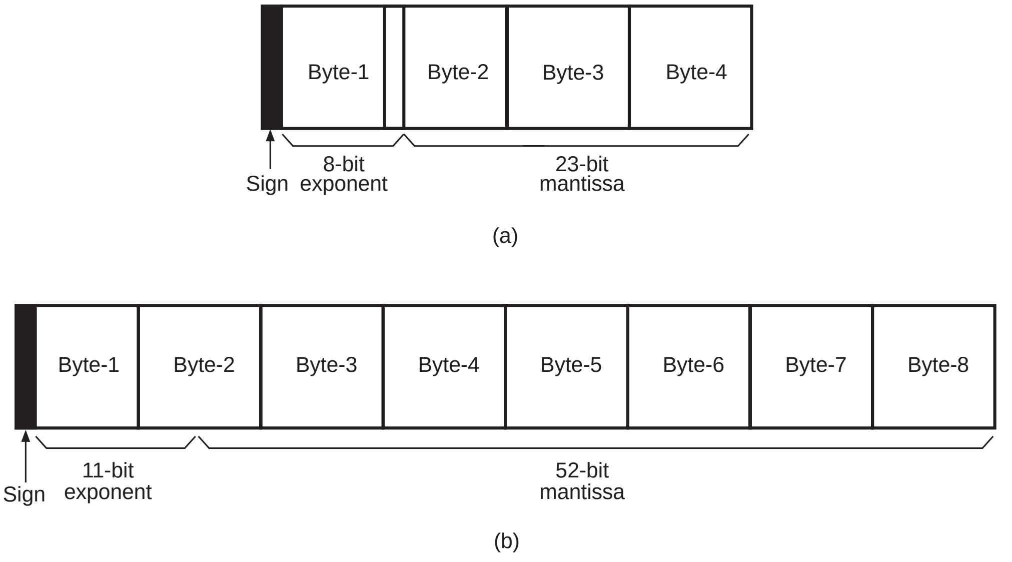


Figure 1.1 Single-precision and double-precision formats.

'1'. This '1' is not included but is always implied. A similar explanation can be given in the case of the double-precision format shown in Fig. 1.1(b).

Step-by-step transformation of $(23)_{10}$ into an equivalent floating-point number in single-precision IEEE format is as follows:

- $(23)_{10} = (10111)_2 = 1.0111e + 0100$.
- The mantissa = 0111000 00000000 00000000.
- The exponent = 00000100.
- The biased exponent = 00000100 + 01111111 = 10000011.
- The sign of the mantissa = 0.
- $(+23)_{10}$ = 01000001 10111000 00000000 00000000.
- Also, $(-23)_{10}$= 11000001 10111000 00000000 00000000.

IEEE-754r Format

As mentioned earlier, IEEE-754r is an ongoing revision to the IEEE-754 standard. The main objective of the revision is to extend the standard wherever it has become necessary, the most obvious enhancement to the standard being the addition of the 128-bit format and decimal format. Extension of the standard to include decimal floating-point representation has become necessary as most commercial data are held in decimal form and the binary floating point cannot represent decimal fractions exactly. If the binary floating point is used to represent decimal data, it is likely that the results will not be the same as those obtained by using decimal arithmetic.

In the revision process, many of the definitions have been rewritten for clarification and consistency. In terms of the addition of new formats, a new addition to the existing binary formats is the 128-bit 'quad-precision' format. Also, three new decimal formats, matching the lengths of binary formats, have been described. These include decimal formats with a seven-, 16- and 34-digit mantissa, which may be normalized or denormalized. In order to achieve maximum range (decided by the number of exponent bits) and precision (decided by the number of mantissa bits), the formats merge part of the exponent and mantissa into a combination field and compress the remainder of the mantissa using densely packed decimal encoding. Detailed description of the revision, however, is beyond the scope of this book.

IEEE-854 Standard

The main objective of the IEEE-854 standard was to define a standard for floating-point arithmetic without the radix and word length dependencies of the better-known IEEE-754 standard. That is why IEEE-854 is called the IEEE standard for radix-independent floating-point arithmetic. Although the standard specifies only the binary and decimal floating-point arithmetic, it provides sufficient guidelines for those contemplating the implementation of the floating point using any other radix value such as 16 of the hexadecimal number system. This standard, too, specifies four formats including single, single-extended, double and double-extended precision formats.

Example 1.11

Determine the floating-point representation of $(-142)_{10}$ *using the IEEE single-precision format.*

Solution

- As a first step, we will determine the binary equivalent of $(142)_{10}$. Following the procedure outlined in an earlier part of the chapter, the binary equivalent can be written as $(142)_{10} = (10001110)_2$.
- $(10001110)_2 = 1.000\ 1110 \times 2^7 = 1.0001110e + 0111$.
- The mantissa = 0001110 00000000 00000000.
- The exponent = 00000111.
- The biased exponent = 00000111 + 01111111 = 10000110.
- The sign of the mantissa = 1.
- Therefore, $(-142)_{10}$ = 11000011 00001110 00000000 00000000.

Example 1.12

Determine the equivalent decimal numbers for the following floating-point numbers:

(a) 00111111 01000000 00000000 00000000 (IEEE-754 single-precision format);
(b) 11000000 00101001 01100 . . . 45 0s (IEEE-754 double-precision format).

Solution

(a) From an examination of the given number:
The sign of the mantissa is positive, as indicated by the ‘0’ bit in the designated position.
The biased exponent = 01111110.
The unbiased exponent = 01111110 − 01111111 = 11111111.
It is clear from the eight bits of unbiased exponent that the exponent is negative, as the 2’s complement representation of a number gives ‘1’ in place of MSB.
The magnitude of the exponent is given by the 2’s complement of $(11111111)_2$, which is $(00000001)_2 = 1$.

Therefore, the exponent $= -1$.
The mantissa bits = 11000000 00000000 00000000 ('1' in MSB is implied).
The normalized mantissa = 1.1000000 00000000 00000000.
The magnitude of the mantissa can be determined by shifting the mantissa bits one position to the left.
That is, the mantissa $= (.11)_2 = (0.75)_{10}$.

(b) The sign of the mantissa is negative, indicated by the '1' bit in the designated position.
The biased exponent = 10000000010.
The unbiased exponent = 10000000010 − 01111111111 = 00000000011.
It is clear from the 11 bits of unbiased exponent that the exponent is positive owing to the '0' in place of MSB. The magnitude of the exponent is 3. Therefore, the exponent $= +3$.
The mantissa bits = 1100101100 . . . 45 0s ('1' in MSB is implied).
The normalized mantissa = 1.100101100 . . . 45 0s.
The magnitude of the mantissa can be determined by shifting the mantissa bits three positions to the right.
That is, the mantissa $= (1100.101)_2 = (12.625)_{10}$.
Therefore, the equivalent decimal number $= -12.625$.

Review Questions

1. What is meant by the radix or base of a number system? Briefly describe why hex representation is used for the addresses and the contents of the memory locations in the main memory of a computer.
2. What do you understand by the l's and 2's complements of a binary number? What will be the range of decimal numbers that can be represented using a 16-bit 2's complement format?
3. Briefly describe the salient features of the IEEE-754 standard for representing floating-point numbers.
4. Why was it considered necessary to carry out a revision of the IEEE-754 standard? What are the main features of IEEE-754r (the notation for IEEE-754 under revision)?
5. In a number system, what decides (a) the place value or weight of a given digit and (b) the maximum numbers representable with a given number of digits?
6. In a floating-point representation, what represents (a) the range of representable numbers and (b) the precision with which a given number can be represented?
7. Why is there a need to have floating-point standards that can take care of decimal data and decimal arithmetic in addition to binary data and arithmetic?

Problems

1. Do the following conversions:

   (a) eight-bit 2's complement representation of $(-23)_{10}$;
   (b) The decimal equivalent of $(00010111)_2$ represented in 2's complement form.

   (a) 11101001; (b) +23

2. Two possible binary representations of $(-1)_{10}$ are $(10000001)_2$ and $(11111111)_2$. One of them belongs to the sign-bit magnitude format and the other to the 2's complement format. Identify.

   $(10000001)_2$ = sign-bit magnitude and $(11111111)_2$ = 2's complement form

3. Represent the following in the IEEE-754 floating-point standard using the single-precision format:

   (a) 32-bit binary number 11110000 11001100 10101010 00001111;
   (b) $(-118.625)_{10}$.

(a) 01001111 01110000 11001100 10101010;
(b) 11000010 11101101 01000000 00000000

4. Give the next three numbers in each of the following hex sequences:

   (a) 4A5, 4A6, 4A7, 4A8, . . . ;
   (b) B998, B999, . . .

(a) 4A9, 4AA, 4AB; (b) B99A, B99B, B99C

5. Show that:

   (a) $(13A7)_{16} = (5031)_{10}$;
   (b) $(3F2)_{16} = (1111110010)_2$.

6. Assume a radix-32 arbitrary number system with 0–9 and A–V as its basic digits. Express the mixed binary number $(110101.001)_2$ in this arbitrary number system.

1L.4

Further Reading

1. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
2. Atiyah, S. K. (2005) *A Survey of Arithmetic*, Trafford Publishing, Victoria, BC, Canada.
3. Langholz, G., Mott, J. L. and Kandel, A. (1998) *Foundations of Digital Logic Design*, World Scientific Publ. Co. Inc., Singapore.
4. Cook, N. P. (2003) *Practical Digital Electronics*, Prentice-Hall, NJ, USA.
5. Lu, M. (2004) *Arithmetic and Logic in Computer Systems*, John Wiley & Sons, Inc., NJ, USA.

2

Binary Codes

The present chapter is an extension of the previous chapter on *number systems*. In the previous chapter, beginning with some of the basic concepts common to all number systems and an outline on the familiar decimal number system, we went on to discuss the binary, the hexadecimal and the octal number systems. While the binary system of representation is the most extensively used one in digital systems, including computers, octal and hexadecimal number systems are commonly used for representing groups of binary digits. The binary coding system, called the straight binary code and discussed in the previous chapter, becomes very cumbersome to handle when used to represent larger decimal numbers. To overcome this shortcoming, and also to perform many other special functions, several binary codes have evolved over the years. Some of the better-known binary codes, including those used efficiently to represent numeric and alphanumeric data, and the codes used to perform special functions, such as detection and correction of errors, will be detailed in this chapter.

2.1 Binary Coded Decimal

The binary coded decimal (BCD) is a type of binary code used to represent a given decimal number in an equivalent binary form. BCD-to-decimal and decimal-to-BCD conversions are very easy and straightforward. It is also far less cumbersome an exercise to represent a given decimal number in an equivalent BCD code than to represent it in the equivalent straight binary form discussed in the previous chapter.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four-bit binary equivalent. As an example, the BCD equivalent of $(23.15)_{10}$ is written as $(0010\ 0011.0001\ 0101)_{BCD}$. The BCD code described above is more precisely known as the 8421 BCD code, with 8, 4, 2 and 1 representing the weights of different bits in the four-bit groups, starting from MSB and proceeding towards LSB. This feature makes it a weighted code, which means that each bit in the four-bit group representing a given decimal digit has an assigned

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Table 2.1 BCD codes.

Decimal	8421 BCD code	4221 BCD code	5421 BCD code
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	1000	0100
5	0101	0111	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

weight. Other weighted BCD codes include the 4221 BCD and 5421 BCD codes. Again, 4, 2, 2 and 1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code represent weights of the relevant bits. Table 2.1 shows a comparison of 8421, 4221 and 5421 BCD codes. As an example, $(98.16)_{10}$ will be written as 1111 1110.0001 1100 in 4221 BCD code and 1100 1011.0001 1001 in 5421 BCD code. Since the 8421 code is the most popular of all the BCD codes, it is simply referred to as the BCD code.

2.1.1 BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent. The first step is straightforward, and the second step was explained in the previous chapter. As an example, we will find the binary equivalent of the BCD number 0010 1001.0111 0101:

- BCD number: 0010 1001.0111 0101.
- Corresponding decimal number: 29.75.
- The binary equivalent of 29.75 can be determined to be 11101 for the integer part and .11 for the fractional part.
- Therefore, $(0010\ 1001.0111\ 0101)_{BCD} = (11101.11)_2$.

2.1.2 Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order. A given binary number can be converted into an equivalent BCD number by first determining its decimal equivalent and then writing the corresponding BCD equivalent. As an example, we will find the BCD equivalent of the binary number 10101011.101:

- The decimal equivalent of this binary number can be determined to be 171.625.
- The BCD equivalent can then be written as 0001 0111 0001.0110 0010 0101.

2.1.3 Higher-Density BCD Encoding

In the regular BCD encoding of decimal numbers, the number of bits needed to represent a given decimal number is always greater than the number of bits required for straight binary encoding of the same. For example, a three-digit decimal number requires 12 bits for representation in conventional BCD format. However, since $2^{10} > 10^3$, if these three decimal digits are encoded together, only 10 bits would be needed to do that. Two such encoding schemes are *Chen-Ho encoding* and the *densely packed decimal*. The latter has the advantage that subsets of the encoding encode two digits in the optimal seven bits and one digit in four bits like regular BCD.

2.1.4 Packed and Unpacked BCD Numbers

In the case of unpacked BCD numbers, each four-bit BCD group corresponding to a decimal digit is stored in a separate register inside the machine. In such a case, if the registers are eight bits or wider, the register space is wasted.

In the case of packed BCD numbers, two BCD digits are stored in a single eight-bit register. The process of combining two BCD digits so that they are stored in one eight-bit register involves shifting the number in the upper register to the left 4 times and then adding the numbers in the upper and lower registers. The process is illustrated by showing the storage of decimal digits '5' and '7':

- Decimal digit 5 is initially stored in the eight-bit register as: 0000 0101.
- Decimal digit 7 is initially stored in the eight-bit register as: 0000 0111.
- After shifting to the left 4 times, the digit 5 register reads: 0101 0000.
- The addition of the contents of the digit 5 and digit 7 registers now reads: 0101 0111.

Example 2.1

How many bits would be required to encode decimal numbers 0 to 9999 in straight binary and BCD codes? What would be the BCD equivalent of decimal 27 in 16-bit representation?

Solution

- Total number of decimals to be represented $= 10\,000 = 10^4 = 2^{13.29}$.
- Therefore, the number of bits required for straight binary encoding = 14.
- The number of bits required for BCD encoding = 16.
- The BCD equivalent of 27 in 16-bit representation = 0000000000100111.

2.2 Excess-3 Code

The excess-3 code is another important BCD code. It is particularly significant for arithmetic operations as it overcomes the shortcomings encountered while using the 8421 BCD code to add two decimal digits whose sum exceeds 9. The excess-3 code has no such limitation, and it considerably simplifies arithmetic operations. Table 2.2 lists the excess-3 code for the decimal numbers 0–9.

The excess-3 code for a given decimal number is determined by adding '3' to each decimal digit in the given number and then replacing each digit of the newly found decimal number by

Table 2.2 Excess-3 code equivalent of decimal numbers.

Decimal number	Excess-3 code	Decimal number	Excess-3 code
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

its four-bit binary equivalent. It may be mentioned here that, if the addition of '3' to a digit produces a carry, as is the case with the digits 7, 8 and 9, that carry should not be taken forward. The result of addition should be taken as a single entity and subsequently replaced with its excess-3 code equivalent. As an example, let us find the excess-3 code for the decimal number 597:

- The addition of '3' to each digit yields the three new digits/numbers '8', '12' and '10'.
- The corresponding four-bit binary equivalents are 1000, 1100 and 1010 respectively.
- The excess-3 code for 597 is therefore given by: 1000 1100 1010 = 100011001010.

Also, it is normal practice to represent a given decimal digit or number using the maximum number of digits that the digital system is capable of handling. For example, in four-digit decimal arithmetic, 5 and 37 would be written as 0005 and 0037 respectively. The corresponding 8421 BCD equivalents would be 0000000000000101 and 0000000000110111 and the excess-3 code equivalents would be 0011001100111000 and 0011001101101010.

Corresponding to a given excess-3 code, the equivalent decimal number can be determined by first splitting the number into four-bit groups, starting from the radix point, and then subtracting 0011 from each four-bit group. The new number is the 8421 BCD equivalent of the given excess-3 code, which can subsequently be converted into the equivalent decimal number. As an example, following these steps, the decimal equivalent of excess-3 number 01010110.10001010 would be 23.57.

Another significant feature that makes this code attractive for performing arithmetic operations is that the complement of the excess-3 code of a given decimal number yields the excess-3 code for 9's complement of the decimal number. As adding 9's complement of a decimal number B to a decimal number A achieves A – B, the excess-3 code can be used effectively for both addition and subtraction of decimal numbers.

Example 2.3

Find (a) the excess-3 equivalent of $(237.75)_{10}$ and (b) the decimal equivalent of the excess-3 number 110010100011.01110101.

Solution

(a) Integer part = 237. The excess-3 code for $(237)_{10}$ is obtained by replacing 2, 3 and 7 with the four-bit binary equivalents of 5, 6 and 10 respectively. This gives the excess-3 code for $(237)_{10}$ as: 0101 0110 1010 = 010101101010.

Fractional part = .75. The excess-3 code for $(.75)_{10}$ is obtained by replacing 7 and 5 with the four-bit binary equivalents of 10 and 8 respectively. That is, the excess-3 code for $(.75)_{10}$ = .10101000.
Combining the results of the integral and fractional parts, the excess-3 code for $(237.75)_{10}$ = 010101101010.10101000.

(b) The excess-3 code = 110010100011.01110101 = 1100 1010 0011.0111 0101.
Subtracting 0011 from each four-bit group, we obtain the new number as: 1001 0111 0000.0100 0010.
Therefore, the decimal equivalent = $(970.42)_{10}$.

2.3 Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953. It is an unweighted binary code in which two successive values differ only by 1 bit. Owing to this feature, the maximum error that can creep into a system using the binary Gray code to encode data is much less than the worst-case error encountered in the case of straight binary encoding. Table 2.3 lists the binary and Gray code equivalents of decimal numbers 0–15. An examination of the four-bit Gray code numbers, as listed in Table 2.3, shows that the last entry rolls over to the first entry. That is, the last and the first entry also differ by only 1 bit. This is known as the *cyclic property* of the Gray code. Although there can be more than one Gray code for a given word length, the term was first applied to a specific binary code for non-negative integers and called the *binary-reflected Gray code* or simply the Gray code.

There are various ways by which Gray codes with a given number of bits can be remembered. One such way is to remember that the least significant bit follows a repetitive pattern of '2' (11, 00, 11, . . .), the next higher adjacent bit follows a pattern of '4' (1111, 0000, 1111, . . .) and so on. We can also generate the n-bit Gray code recursively by prefixing a '0' to the Gray code for $n-1$ bits to obtain the first 2^{n-1} numbers, and then prefixing '1' to the reflected Gray code for $n-1$ bits to obtain the remaining 2^{n-1} numbers. The reflected Gray code is nothing but the code written in reverse order. The process of generation of higher-bit Gray codes using the reflect-and-prefix method is illustrated in Table 2.4. The columns of bits between those representing the Gray codes give the intermediate step of writing the code followed by the same written in reverse order.

Table 2.3 Gray code.

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

Table 2.4 Generation of higher-bit Gray code numbers.

One-bit Gray code		Two-bit Gray code		Three-bit Gray code		Four-bit Gray code
0	0	00	00	000	000	0000
1	1	01	01	001	001	0001
	1	11	11	011	011	0011
	0	10	10	010	010	0010
			10	110	110	0110
			11	111	111	0111
			01	101	101	0101
			00	100	100	0100
					100	1100
					101	1101
					111	1111
					110	1110
					010	1010
					011	1011
					001	1001
					000	1000

2.3.1 Binary–Gray Code Conversion

A given binary number can be converted into its Gray code equivalent by going through the following steps:

1. Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.
2. The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both '1', then the corresponding Gray code bit would be a '0'.
3. The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.
4. The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of $(1011)_2$ into its Gray code equivalent:

Binary	1011
Gray code	1- - -
Binary	1011
Gray code	11- -
Binary	1011
Gray code	111-
Binary	1011
Gray code	1110

2.3.2 Gray Code–Binary Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.
3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.
4. The process continues until we obtain the LSB of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code	1110
Binary	1- - -
Gray code	1110
Binary	10 - -
Gray code	1110
Binary	101
Gray code	1110
Binary	1011

2.3.3 n-ary Gray Code

The binary-reflected Gray code described above is invariably referred to as the 'Gray code'. However, over the years, mathematicians have discovered other types of Gray code. One such code is the n-ary Gray code, also called the non-Boolean Gray code owing to the use of non-Boolean symbols for encoding. The generalized representation of the code is the (n, k)-Gray code, where n is the number of independent digits used and k is the word length. A ternary Gray code ($n=3$) uses the values 0, 1 and 2, and the sequence of numbers in the two-digit word length would be (00, 01, 02, 12, 11, 10, 20, 21, 22). In the quaternary ($n=4$) code, using 0, 1, 2 and 3 as independent digits and a two-digit word length, the sequence of numbers would be (00, 01, 02, 03, 13, 12, 11, 10, 20, 21, 22, 23, 33, 32, 31, 30). It is important to note here that an (n, k)-Gray code with an odd n does not exhibit the cyclic property of the binary Gray code, while in case of an even n it does have the cyclic property.

The (n, k)-Gray code may be constructed recursively, like the binary-reflected Gray code, or may be constructed iteratively. The process of generating larger word-length ternary Gray codes is illustrated in Table 2.5. The columns between those representing the ternary Gray codes give the intermediate steps.

2.3.4 Applications

1. The Gray code is used in the transmission of digital signals as it minimizes the occurrence of errors.
2. The Gray code is preferred over the straight binary code in angle-measuring devices. Use of the Gray code almost eliminates the possibility of an angle misread, which is likely if the

Table 2.5 Generation of a larger word-length ternary Gray code.

One-digit ternary code	Two-digit ternary code		Three-digit ternary code	
0	0	00	00	000
1	1	01	01	001
2	2	02	02	002
	2	12	12	012
	1	11	11	011
	0	10	10	010
	0	20	20	020
	1	21	21	021
	2	22	22	022
			22	122
			21	121
			20	120
			10	110
			11	111
			12	112
			02	102
			01	101
			00	100
			00	200
			01	201
			02	202
			12	212
			11	211
			10	210
			20	220
			21	221
			22	222

angle is represented in straight binary. The cyclic property of the Gray code is a plus in this application.

3. The Gray code is used for labelling the axes of Karnaugh maps, a graphical technique used for minimization of Boolean expressions.
4. The use of Gray codes to address program memory in computers minimizes power consumption. This is due to fewer address lines changing state with advances in the program counter.
5. Gray codes are also very useful in genetic algorithms since mutations in the code allow for mostly incremental changes. However, occasionally a one-bit change can result in a big leap, thus leading to new properties.

Example 2.4

Find (a) the Gray code equivalent of decimal 13 and (b) the binary equivalent of Gray code number 1111.

Solution

(a) The binary equivalent of decimal 13 is 1101.
Binary–Gray conversion

Binary 1101
Gray 1- - -
Binary 1101
Gray 10 - -
Binary 1101
Gray 101 –
Binary 1101
Gray 1011

(b) *Gray–binary conversion*

Gray 1111
Binary 1- - -
Gray 1111
Binary 10- -
Gray 1111
Binary 101-
Gray 1111
Binary 1010

Example 2.5

Given the sequence of three-bit Gray code as (000, 001, 011, 010, 110, 111, 101, 100), write the next three numbers in the four-bit Gray code sequence after 0101.

Solution

The first eight of the 16 Gray code numbers of the four-bit Gray code can be written by appending '0' to the eight three-bit Gray code numbers. The remaining eight can be determined by appending '1' to the eight three-bit numbers written in reverse order. Following this procedure, we can write the next three numbers after 0101 as 0100, 1100 and 1101.

2.4 Alphanumeric Codes

Alphanumeric codes, also called character codes, are binary codes used to represent alphanumeric data. The codes write alphanumeric data, including letters of the alphabet, numbers, mathematical symbols and punctuation marks, in a form that is understandable and processable by a computer. These codes enable us to interface input–output devices such as keyboards, printers, VDUs, etc., with the computer. One of the better-known alphanumeric codes in the early days of evolution of computers, when punched cards used to be the medium of inputting and outputting data, is the 12-bit Hollerith code. The Hollerith code was used in those days to encode alphanumeric data on punched cards. The code has, however, been rendered obsolete, with the punched card medium having completely vanished from the scene. Two widely used alphanumeric codes include the ASCII and the EBCDIC codes. While the former is popular with microcomputers and is used on nearly all personal computers and workstations, the latter is mainly used with larger systems.

Traditional character encodings such as ASCII, EBCDIC and their variants have a limitation in terms of the number of characters they can encode. In fact, no single encoding contains enough characters so as to cover all the languages of the European Union. As a result, these encodings do not permit multilingual computer processing. Unicode, developed jointly by the Unicode Consortium and the International Standards Organization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. Different codes are described in the following.

2.4.1 ASCII code

The ASCII (American Standard Code for Information Interchange), pronounced ‘ask-ee’, is strictly a seven-bit code based on the English alphabet. ASCII codes are used to represent alphanumeric data in computers, communications equipment and other related devices. The code was first published as a standard in 1967. It was subsequently updated and published as ANSI X3.4-1968, then as ANSI X3.4-1977 and finally as ANSI X3.4-1986. Since it is a seven-bit code, it can at the most represent 128 characters. It currently defines 95 printable characters including 26 upper-case letters (A to Z), 26 lower-case letters (a to z), 10 numerals (0 to 9) and 33 special characters including mathematical symbols, punctuation marks and space character. In addition, it defines codes for 33 nonprinting, mostly obsolete control characters that affect how text is processed. With the exception of ‘carriage return’ and/or ‘line feed’, all other characters have been rendered obsolete by modern mark-up languages and communication protocols, the shift from text-based devices to graphical devices and the elimination of teleprinters, punch cards and paper tapes. An eight-bit version of the ASCII code, known as US ASCII-8 or ASCII-8, has also been developed. The eight-bit version can represent a maximum of 256 characters.

Table 2.6 lists the ASCII codes for all 128 characters. When the ASCII code was introduced, many computers dealt with eight-bit groups (or bytes) as the smallest unit of information. The eighth bit was commonly used as a parity bit for error detection on communication lines and other device-specific functions. Machines that did not use the parity bit typically set the eighth bit to ‘0’.

Table 2.6 ASCII code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	EOT	End of transmission
5	05	0000 0101	ENQ	Enquiry
6	06	0000 0110	ACK	Acknowledgement
7	07	0000 0111	BEL	Bell
8	08	0000 1000	BS	Backspace
9	09	0000 1001	HT	Horizontal tab
10	0A	0000 1010	LF	Line feed
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1 (XON)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	DC3	Device control 3 (XOFF)
20	14	0001 0100	DC4	Device control 4
21	15	0001 0101	NAK	Negative acknowledgement
22	16	0001 0110	SYN	Synchronous idle
23	17	0001 0111	ETB	End of transmission block
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	SUB	Substitute
27	1B	0001 1011	ESC	Escape
28	1C	0001 1100	FS	File separator
29	1D	0001 1101	GS	Group separator
30	1E	0001 1110	RS	Record separator
31	1F	0001 1111	US	Unit separator
32	20	0010 0000	SP	Space
33	21	0010 0001	!	Exclamation point
34	22	0010 0010	"	Quotation mark
35	23	0010 0011	#	Number sign, octothorp, pound
36	24	0010 0100	$	Dollar sign
37	25	0010 0101	%	Percent
38	26	0010 0110	&	Ampersand
39	27	0010 0111	’	Apostrophe, prime
40	28	0010 1000	(	Left parenthesis
41	29	0010 1001	)	Right parenthesis
42	2A	0010 1010	*	Asterisk, ‘star’
43	2B	0010 1011	+	Plus sign
44	2C	0010 1100	,	Comma
45	2D	0010 1101	-	Hyphen, minus sign
46	2E	0010 1110	.	Period, decimal Point, ‘dot’
47	2F	0010 1111	/	Slash, virgule
48	30	0011 0000	0	0
49	31	0011 0001	1	1
50	32	0011 0010	2	2
51	33	0011 0011	3	3
52	34	0011 0100	4	4
53	35	0011 0101	5	5
54	36	0011 0110	6	6
55	37	0011 0111	7	7
56	38	0011 1000	8	8
57	39	0011 1001	9	9
58	3A	0011 1010	:	Colon
59	3B	0011 1011	;	Semicolon
60	3C	0011 1100	<	Less-than sign
61	3D	0011 1101	=	Equals sign
62	3E	0011 1110	>	Greater-than sign
63	3F	0011 1111	?	Question mark
64	40	0100 0000	@	At sign
65	41	0100 0001	A	A

(*continued overleaf*)

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
66	42	0100 0010	B	B
67	43	0100 0011	C	C
68	44	0100 0100	D	D
69	45	0100 0101	E	E
70	46	0100 0110	F	F
71	47	0100 0111	G	G
72	48	0100 1000	H	H
73	49	0100 1001	I	I
74	4A	0100 1010	J	J
75	4B	0100 1011	K	K
76	4C	0100 1100	L	L
77	4D	0100 1101	M	M
78	4E	0100 1110	N	N
79	4F	0100 1111	O	O
80	50	0101 0000	P	P
81	51	0101 0001	Q	Q
82	52	0101 0010	R	R
83	53	0101 0011	S	S
84	54	0101 0100	T	T
85	55	0101 0101	U	U
86	56	0101 0110	V	V
87	57	0101 0111	W	W
88	58	0101 1000	X	X
89	59	0101 1001	Y	Y
90	5A	0101 1010	Z	Z
91	5B	0101 1011	[	Opening bracket
92	5C	0101 1100	\	Reverse slash
93	5D	0101 1101	]	Closing bracket
94	5E	0101 1110	∧	Circumflex, caret
95	5F	0101 1111	_	Underline, underscore
96	60	0110 0000	`	Grave accent
97	61	0110 0001	a	a
98	62	0110 0010	b	b
99	63	0110 0011	c	c
100	64	0110 0100	d	d
101	65	0110 0101	e	e
102	66	0110 0110	f	f
103	67	0110 0111	g	g
104	68	0110 1000	h	h
105	69	0110 1001	i	i
106	6A	0110 1010	j	j
107	6B	0110 1011	k	k
108	6C	0110 1100	l	l
109	6D	0110 1101	m	m
110	6E	0110 1110	n	n
111	6F	0110 1111	o	o
112	70	0111 0000	p	p
113	71	0111 0001	q	q
114	72	0111 0010	r	r

Table 2.6 (*continued*).

Decimal	Hex	Binary	Code	Code description
115	73	0111 0011	s	s
116	74	0111 0100	t	t
117	75	0111 0101	u	u
118	76	0111 0110	v	v
119	77	0111 0111	w	w
120	78	0111 1000	x	x
121	79	0111 1001	y	y
122	7A	0111 1010	z	z
123	7B	0111 1011	{	Opening brace
124	7C	0111 1100	|	Vertical line
125	7D	0111 1101	}	Closing brace
126	7E	0111 1110	~	Tilde
127	7F	0111 1111	DEL	Delete

Looking at the structural features of the code as reflected in Table 2.6, we can see that the digits 0 to 9 are represented with their binary values prefixed with 0011. That is, numerals 0 to 9 are represented by binary sequences from 0011 0000 to 0011 1001 respectively. Also, lower-case and upper-case letters differ in bit pattern by a single bit. While upper-case letters 'A' to 'O' are represented by 0100 0001 to 0100 1111, lower-case letters 'a' to 'o' are represented by 0110 0001 to 0110 1111. Similarly, while upper-case letters 'P' to 'Z' are represented by 0101 0000 to 0101 1010, lower-case letters 'p' to 'z' are represented by 0111 0000 to 0111 1010.

With widespread use of computer technology, many variants of the ASCII code have evolved over the years to facilitate the expression of non-English languages that use a Roman-based alphabet. In some of these variants, all ASCII printable characters are identical to their seven-bit ASCII code representations. For example, the eight-bit standard ISO/IEC 8859 was developed as a true extension of ASCII, leaving the original character mapping intact in the process of inclusion of additional values. This made possible representation of a broader range of languages. In spite of the standard suffering from incompatibilities and limitations, ISO-8859-1, its variant Windows-1252 and the original seven-bit ASCII continue to be the most common character encodings in use today.

2.4.2 EBCDIC code

The EBCDIC (Extended Binary Coded Decimal Interchange Code), pronounced 'eb-si-dik', is another widely used alphanumeric code, mainly popular with larger systems. The code was created by IBM to extend the binary coded decimal that existed at that time. All IBM mainframe computer peripherals and operating systems use EBCDIC code, and their operating systems provide ASCII and Unicode modes to allow translation between different encodings. It may be mentioned here that EBCDIC offers no technical advantage over the ASCII code and its variant ISO-8859 or Unicode. Its importance in the earlier days lay in the fact that it made it relatively easier to enter data into larger machines with punch cards. Since, punch cards are not used on mainframes any more, the code is used in contemporary mainframe machines solely for backwards compatibility.

It is an eight-bit code and thus can accommodate up to 256 characters. Table 2.7 gives the listing of characters in binary as well as hex form in EBCDIC. The arrangement is similar to the one adopted for Table 2.6 for the ASCII code. A single byte in EBCDIC is divided into two four-bit groups called

Table 2.7 EBCDIC code.

Decimal	Hex	Binary	Code	Code description
0	00	0000 0000	NUL	Null character
1	01	0000 0001	SOH	Start of header
2	02	0000 0010	STX	Start of text
3	03	0000 0011	ETX	End of text
4	04	0000 0100	PF	Punch off
5	05	0000 0101	HT	Horizontal tab
6	06	0000 0110	LC	Lower case
7	07	0000 0111	DEL	Delete
8	08	0000 1000		
9	09	0000 1001		
10	0A	0000 1010	SMM	Start of manual message
11	0B	0000 1011	VT	Vertical tab
12	0C	0000 1100	FF	Form feed
13	0D	0000 1101	CR	Carriage return
14	0E	0000 1110	SO	Shift out
15	0F	0000 1111	SI	Shift in
16	10	0001 0000	DLE	Data link escape
17	11	0001 0001	DC1	Device control 1
18	12	0001 0010	DC2	Device control 2
19	13	0001 0011	TM	Tape mark
20	14	0001 0100	RES	Restore
21	15	0001 0101	NL	New line
22	16	0001 0110	BS	Backspace
23	17	0001 0111	IL	Idle
24	18	0001 1000	CAN	Cancel
25	19	0001 1001	EM	End of medium
26	1A	0001 1010	CC	Cursor control
27	1B	0001 1011	CU1	Customer use 1
28	1C	0001 1100	IFS	Interchange file separator
29	1D	0001 1101	IGS	Interchange group separator
30	1E	0001 1110	IRS	Interchange record separator
31	1F	0001 1111	IUS	Interchange unit separator
32	20	0010 0000	DS	Digit select
33	21	0010 0001	SOS	Start of significance
34	22	0010 0010	FS	Field separator
35	23	0010 0011		
36	24	0010 0100	BYP	Bypass
37	25	0010 0101	LF	Line feed
38	26	0010 0110	ETB	End of transmission block
39	27	0010 0111	ESC	Escape
40	28	0010 1000		
41	29	0010 1001		
42	2A	0010 1010	SM	Set mode
43	2B	0010 1011	CU2	Customer use 2
44	2C	0010 1100		
45	2D	0010 1101	ENQ	Enquiry
46	2E	0010 1110	ACK	Acknowledge
47	2F	0010 1111	BEL	Bell
48	30	0011 0000		

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
49	31	0011 0001		
50	32	0011 0010	SYN	Synchronous idle
51	33	0011 0011		
52	34	0011 0100	PN	Punch on
53	35	0011 0101	RS	Reader stop
54	36	0011 0110	UC	Upper case
55	37	0011 0111	EOT	End of transmission
56	38	0011 1000		
57	39	0011 1001		
58	3A	0011 1010		
59	3B	0011 1011	CU3	Customer use 3
60	3C	0011 1100	DC4	Device control 4
61	3D	0011 1101	NAK	Negative acknowledge
62	3E	0011 1110		
63	3F	0011 1111	SUB	Substitute
64	40	0100 0000	SP	Space
65	41	0100 0001		
66	42	0100 0010		
67	43	0100 0011		
68	44	0100 0100		
69	45	0100 0101		
70	46	0100 0110		
71	47	0100 0111		
72	48	0100 1000		
73	49	0100 1001		
74	4A	0100 1010	¢	Cent sign
75	4B	0100 1011	.	Period, decimal point
76	4C	0100 1100	$<$	Less-than sign
77	4D	0100 1101	(	Left parenthesis
78	4E	0100 1110	+	Plus sign
79	4F	0100 1111	$\vert$	Logical OR
80	50	0101 0000	&	Ampersand
81	51	0101 0001		
82	52	0101 0010		
83	53	0101 0011		
84	54	0101 0100		
85	55	0101 0101		
86	56	0101 0110		
87	57	0101 0111		
88	58	0101 1000		
89	59	0101 1001		
90	5A	0101 1010	!	Exclamation point
91	5B	0101 1011	$	Dollar sign
92	5C	0101 1100	*	Asterisk
93	5D	0101 1101	)	Right parenthesis
94	5E	0101 1110	;	Semicolon
95	5F	0101 1111	$\wedge$	Logical NOT
96	60	0110 0000	-	Hyphen, minus sign

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
97	61	0110 0001	/	Slash, virgule
98	62	0110 0010		
99	63	0110 0011		
100	64	0110 0100		
101	65	0110 0101		
102	66	0110 0110		
103	67	0110 0111		
104	68	0110 1000		
105	69	0110 1001		
106	6A	0110 1010		
107	6B	0110 1011	,	Comma
108	6C	0110 1100	%	Percent
109	6D	0110 1101	_	Underline, underscore
110	6E	0110 1110	>	Greater-than sign
111	6F	0110 1111	?	Question mark
112	70	0111 0000		
113	71	0111 0001		
114	72	0111 0010		
115	73	0111 0011		
116	74	0111 0100		
117	75	0111 0101		
118	76	0111 0110		
119	77	0111 0111		
120	78	0111 1000		
121	79	0111 1001	‘	Grave accent
122	7A	0111 1010	:	Colon
123	7B	0111 1011	#	Number sign, octothorp, pound
124	7C	0111 1100	@	At sign
125	7D	0111 1101	’	Apostrophe, prime
126	7E	0111 1110	=	Equals sign
127	7F	0111 1111	“	Quotation mark
128	80	1000 0000		
129	81	1000 1001	a	a
130	82	1000 1010	b	b
131	83	1000 1011	c	c
132	84	1000 1100	d	d
133	85	1000 0101	e	e
134	86	1000 0110	f	f
135	87	1000 0111	g	g
136	88	1000 1000	h	h
137	89	1000 1001	i	i
138	8A	1000 1010		
139	8B	1000 1011		
140	8C	1000 1100		
141	8D	1000 1101		
142	8E	1000 1110		
143	8F	1000 1111		
144	90	1001 0000		
145	91	1001 0001	j	j

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
146	92	1001 0010	k	k
147	93	1001 0011	l	l
148	94	1001 0100	m	m
149	95	1001 0101	n	n
150	96	1001 0110	o	o
151	97	1001 0111	p	p
152	98	1001 1000	q	q
153	99	1001 1001	r	r
154	9A	1001 1010		
155	9B	1001 1011		
156	9C	1001 1100		
157	9D	1001 1101		
158	9E	1001 1110		
159	9F	1001 1111		
160	A0	1010 0000		
161	A1	1010 0001	$\sim$	Tilde
162	A2	1010 0010	s	s
163	A3	1010 0011	t	t
164	A4	1010 0100	u	u
165	A5	1010 0101	v	v
166	A6	1010 0110	w	w
167	A7	1010 0111	x	x
168	A8	1010 1000	y	y
169	A9	1010 1001	z	z
170	AA	1010 1010		
171	AB	1010 1011		
172	AC	1010 1100		
173	AD	1010 1101		
174	AE	1010 1110		
175	AF	1010 1111		
176	B0	1011 0000		
177	B1	1011 0001		
178	B2	1011 0010		
179	B3	1011 0011		
180	B4	1011 0100		
181	B5	1011 0101		
182	B6	1011 0110		
183	B7	1011 0111		
184	B8	1011 1000		
185	B9	1011 1001		
186	BA	1011 1010		
187	BB	1011 1011		
188	BC	1011 1100		
189	BD	1011 1101		
190	BE	1011 1110		
191	BF	1011 1111		
192	C0	1100 0000	{	Opening brace
193	C1	1100 0001	A	A

(*continued overleaf*)

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
194	C2	1100 0010	B	B
195	C3	1100 0011	C	C
196	C4	1100 0100	D	D
197	C5	1100 0101	E	E
198	C6	1100 0110	F	F
199	C7	1100 0111	G	G
200	C8	1100 1000	H	H
201	C9	1100 1001	I	I
202	CA	1100 1010		
203	CB	1100 1011		
204	CC	1100 1100		
205	CD	1100 1101		
206	CE	1100 1110		
207	CF	1100 1111		
208	D0	1101 0000	}	Closing brace
209	D1	1101 0001	J	J
210	D2	1101 0010	K	K
211	D3	1101 0011	L	L
212	D4	1101 0100	M	M
213	D5	1101 0101	N	N
214	D6	1101 0110	O	O
215	D7	1101 0111	P	P
216	D8	1101 1000	Q	Q
217	D9	1101 1001	R	R
218	DA	1101 1010		
219	DB	1101 1011		
220	DC	1101 1100		
221	DD	1101 1101		
222	DE	1101 1110		
223	DF	1101 1111		
224	E0	1110 0000	\	Reverse slant
225	E1	1110 0001		
226	E2	1110 0010	S	S
227	E3	1110 0011	T	T
228	E4	1110 0100	U	U
229	E5	1110 0101	V	V
230	E6	1110 0110	W	W
231	E7	1110 0111	X	X
232	E8	1110 1000	Y	Y
233	E9	1110 1001	Z	Z
234	EA	1110 1010		
235	EB	1110 1011		
236	EC	1110 1100		
237	ED	1110 1101		
238	EE	1110 1110		
239	EF	1110 1111		
240	F0	1111 0000	0	0
241	F1	1111 0001	1	1

Table 2.7 (*continued*).

Decimal	Hex	Binary	Code	Code description
242	F2	1111 0010	2	2
243	F3	1111 0011	3	3
244	F4	1111 0100	4	4
245	F5	1111 0101	5	5
246	F6	1111 0110	6	6
247	F7	1111 0111	7	7
248	F8	1111 1000	8	8
249	F9	1111 1001	9	9
250	FA	1111 1010	|	
251	FB	1111 1011		
252	FC	1111 1100		
253	FD	1111 1101		
254	FE	1111 1110		
255	FF	1111 1111	eo	

nibbles. The first four-bit group, called the 'zone', represents the category of the character, while the second group, called the 'digit', identifies the specific character.

2.4.3 Unicode

As briefly mentioned in the earlier sections, encodings such as ASCII, EBCDIC and their variants do not have a sufficient number of characters to be able to encode alphanumeric data of all forms, scripts and languages. As a result, these encodings do not permit multilingual computer processing. In addition, these encodings suffer from incompatibility. Two different encodings may use the same number for two different characters or different numbers for the same characters. For example, code 4E (in hex) represents the upper-case letter 'N' in ASCII code and the plus sign '+' in the EBCDIC code. Unicode, developed jointly by the Unicode Consortium and the International Organization for Standardization (ISO), is the most complete character encoding scheme that allows text of all forms and languages to be encoded for use by computers. It not only enables the users to handle practically any language and script but also supports a comprehensive set of mathematical and technical symbols, greatly simplifying any scientific information exchange. The Unicode standard has been adopted by such industry leaders as HP, IBM, Microsoft, Apple, Oracle, Unisys, Sun, Sybase, SAP and many more.

Unicode and ISO-10646 Standards

Before we get on to describe salient features of Unicode, it may be mentioned that another standard similar in intent and implementation to Unicode is the ISO-10646. While Unicode is the brainchild of the Unicode Consortium, a consortium of manufacturers (initially mostly US based) of multilingual software, ISO-10646 is the project of the International Organization for Standardization. Although both organizations publish their respective standards independently, they have agreed to maintain compatibility between the code tables of Unicode and ISO-10646 and closely coordinate any further extensions.

The Code Table

The code table defined by both Unicode and ISO-10646 provides a unique number for every character, irrespective of the platform, program and language used. The table contains characters required to represent practically all known languages and scripts. The list includes not only the Greek, Latin, Cyrillic, Arabic, Arabian and Georgian scripts but also Japanese, Chinese and Korean scripts. In addition, the list also includes scripts such as Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Telugu, Tamil, Kannada, Thai, Tibetan, Ethiopic, Sinhala, Canadian Syllabics, Mongolian, Myanmar and others. Scripts not yet covered will eventually be added. The code table also covers a large number of graphical, typographical, mathematical and scientific symbols.

In the 32-bit version, which is the most recent version, the code table is divided into 2^{16} subsets, with each subset having 2^{16} characters. In the 32-bit representation, elements of different subsets therefore differ only in the 16 least significant bits. Each of these subsets is known as a plane. Plane 0, called the basic multilingual plane (BMP), defined by 00000000 to 0000FFFF, contains all most commonly used characters including all those found in major older encoding standards. Another subset of 2^{16} characters could be defined by 00010000 to 0001FFFF. Further, there are different slots allocated within the BMP to different scripts. For example, the basic Latin character set is encoded in the range 0000 to 007F. Characters added to the code table outside the 16-bit BMP are mostly for specialist applications such as historic scripts and scientific notation. There are indications that there may never be characters assigned outside the code space defined by 00000000 to 0010FFFF, which provides space for a little over 1 million additional characters.

Different characters in Unicode are represented by a hexadecimal number preceded by ‘U+’. For example, ‘A’ and ‘e’ in basic Latin are respectively represented by U+0041 and U+0065. The first 256 code numbers in Unicode are compatible with the seven-bit ASCII-code and its eight-bit variant ISO-8859-1. Unicode characters U+0000 to U+007F (128 characters) are identical to those in the ASCII code, and the Unicode characters in the range U+0000 to U+00FF (256 characters) are identical to ISO-8859-1.

Use of Combining Characters

Unicode assigns code numbers to combining characters, which are not full characters by themselves but accents or other diacritical marks added to the previous character. This makes it possible to place any accent on any character. Although Unicode allows the use of combining characters, it also assigns separate codes to commonly used accented characters known as precomposed characters. This is done to ensure backwards compatibility with older encodings. As an example, the character ‘ä’ can be represented as the precomposed character U+00E4. It can also be represented in Unicode as U+0061 (Latin lower-case letter ‘a’) followed by U+00A8 (combining character ‘..’).

Unicode and ISO-10646 Comparison

Although Unicode and ISO-10646 have identical code tables, Unicode offers many more features not available with ISO-10646. While the ISO-10646 standard is not much more than a comprehensive character set, the Unicode standard includes a number of other related features such as character properties and algorithms for text normalization and handling of bidirectional text to ensure correct display of mixed texts containing both right-to-left and left-to-right scripts.

2.5 Seven-segment Display Code

Seven-segment displays [Fig. 2.1(a)] are very common and are found almost everywhere, from pocket calculators, digital clocks and electronic test equipment to petrol pumps. A single seven-segment display or a stack of such displays invariably meets our display requirement. There are both LED and

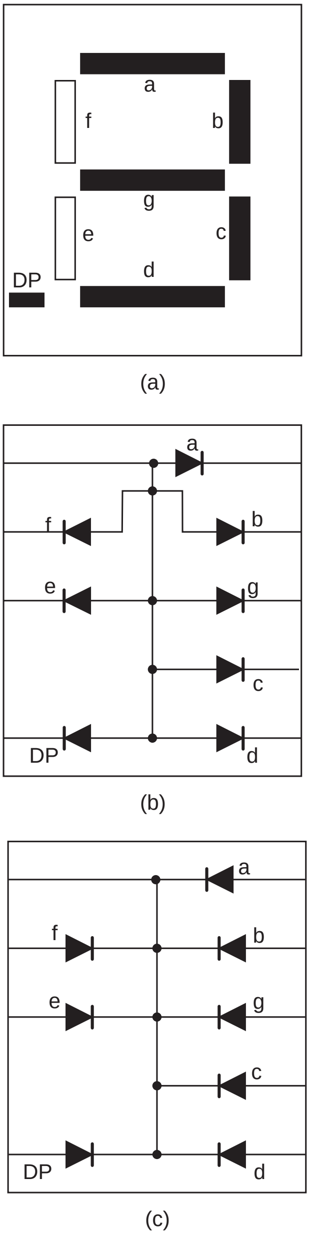


Figure 2.1 Seven-segment displays.

Table 2.8 Seven-segment display code.

Common cathode type '1' means ON									Common anode type '0' means ON								
	a	b	c	d	e	f	g	DP		a	b	c	d	e	f	g	DP
0	1	1	1	1	1	1	0		0	0	0	0	0	0	0	1	
1	0	1	1	0	0	0	0		1	1	0	0	1	1	1	1	
2	1	1	0	1	1	0	1		2	0	0	1	0	0	1	0	
3	1	1	1	1	0	0	1		3	0	0	0	0	1	1	0	
4	0	1	1	0	0	1	1		4	1	0	0	1	1	0	0	
5	1	0	1	1	0	1	1		5	0	1	0	0	1	0	0	
6	0	0	1	1	1	1	1		6	1	1	0	0	0	0	0	
7	1	1	1	0	0	0	0		7	0	0	0	1	1	1	1	
8	1	1	1	1	1	1	1		8	0	0	0	0	0	0	0	
9	1	1	1	0	0	1	1		9	0	0	0	1	1	0	0	
a	1	1	1	1	1	0	1		a	0	0	0	0	0	1	0	
b	0	0	1	1	1	1	1		b	1	1	0	0	0	0	0	
c	0	0	0	1	1	0	1		c	1	1	1	0	0	1	0	
d	0	1	1	1	1	0	1		d	1	0	0	0	0	1	0	
e	1	1	0	1	1	1	1		e	0	0	1	0	0	0	0	
f	1	0	0	0	1	1	1		f	0	1	1	1	0	0	0	

LCD types of seven-segment display. Furthermore, there are common anode-type LED displays where the arrangement of different diodes, designated *a*, *b, c, d, e, f* and *g*, is as shown in Fig. 2.1(b), and common cathode-type displays where the individual diodes are interconnected as shown in Fig. 2.1(c). Each display unit usually has a dot point (DP).

The DP could be located either towards the left (as shown) or towards the right of the figure '8' display pattern. This type of display can be used to display numerals from 0 to 9 and letters from A to F. Table 2.8 gives the binary code for displaying different numeric and alphabetic characters for both the common cathode and the common anode type displays. A '1' lights a segment in the common cathode type display, and a '0' lights a segment in the common anode type display.

2.6 Error Detection and Correction Codes

When we talk about digital systems, be it a digital computer or a digital communication set-up, the issue of error detection and correction is of great practical significance. Errors creep into the bit stream owing to noise or other impairments during the course of its transmission from the transmitter to the receiver. Any such error, if not detected and subsequently corrected, can be disastrous, as digital systems are sensitive to errors and tend to malfunction if the bit error rate is more than a certain threshold level. Error detection and correction, as we will see below, involves the addition of extra bits, called check bits, to the information-carrying bit stream to give the resulting bit sequence a unique characteristic that helps in detection and localization of errors. These additional bits are also called redundant bits as they do not carry any information. While the addition of redundant bits helps in achieving the goal of making transmission of information from one place to another error free or reliable, it also makes it inefficient. In this section, we will examine some common error detection and correction codes.

2.6.1 Parity Code

A parity bit is an extra bit added to a string of data bits in order to detect any error that might have crept into it while it was being stored or processed and moved from one place to another in a digital system.

We have an *even parity*, where the added bit is such that the total number of ls in the data bit string becomes even, and an *odd parity*, where the added bit makes the total number of ls in the data bit string odd. This added bit could be a '0' or a '1'. As an example, if we have to add an even parity bit to 01000001 (the eight-bit ASCII code for 'A'), it will be a '0' and the number will become 001000001. If we have to add an odd parity bit to the same number, it will be a 'l' and the number will become 101000001. The odd parity bit is a complement of the even parity bit. The most common convention is to use even parity, that is, the total number of 1s in the bit stream, including the parity bit, is even.

The parity check can be made at different points to look for any possible single-bit error, as it would disturb the parity. This simple parity code suffers from two limitations. Firstly, it cannot detect the error if the number of bits having undergone a change is even. Although the number of bits in error being equal to or greater than 4 is a very rare occurrence, the addition of a single parity cannot be used to detect two-bit errors, which is a distinct possibility in data storage media such as magnetic tapes. Secondly, the single-bit parity code cannot be used to localize or identify the error bit even if one bit is in error. There are several codes that provide self-single-bit error detection and correction mechanisms, and these are discussed below.

2.6.2 Repetition Code

The repetition code makes use of repetitive transmission of each data bit in the bit stream. In the case of threefold repetition, '1' and '0' would be transmitted as '111' and '000' respectively. If, in the received data bit stream, bits are examined in groups of three bits, the occurrence of an error can be detected. In the case of single-bit errors, '1' would be received as 011 or 101 or 110 instead of 111, and a '0' would be received as 100 or 010 or 001 instead of 000. In both cases, the code becomes self-correcting if the bit in the majority is taken as the correct bit. There are various forms in which the data are sent using the repetition code. Usually, the data bit stream is broken into blocks of bits, and then each block of data is sent some predetermined number of times. For example, if we want to send eight-bit data given by 11011001, it may be broken into two blocks of four bits each. In the case of threefold repetition, the transmitted data bit stream would be 110111011101100110011001. However, such a repetition code where the bit or block of bits is repeated 3 times is not capable of correcting two-bit errors, although it can detect the occurrence of error. For this, we have to increase the number of times each bit in the bit stream needs to be repeated. For example, by repeating each data bit 5 times, we can detect and correct all two-bit errors. The repetition code is highly inefficient and the information throughput drops rapidly as we increase the number of times each data bit needs to be repeated to build error detection and correction capability.

2.6.3 Cyclic Redundancy Check Code

Cyclic redundancy check (CRC) codes provide a reasonably high level of protection at low redundancy level. The cycle code for a given data word is generated as follows. The data word is first appended by a number of 0s equal to the number of check bits to be added. This new data bit sequence is then divided by a special binary word whose length equals $n+1$, n being the number of check bits to be added. The remainder obtained as a result of modulo-2 division is then added to the dividend bit sequence to get the cyclic code. The code word so generated is completely divisible by the divisor used in the generation of the code. Thus, when the received code word is again divided by the same divisor, an error-free reception should lead to an all '0' remainder. A nonzero remainder is indicative of the presence of errors.

The probability of error detection depends upon the number of check bits, n, used to construct the cyclic code. It is 100 % for single-bit and two-bit errors. It is also 100 % when an odd number of bits are in error and the error bursts have a length less than $n + 1$. The probability of detection reduces to $1 - (1/2)^{n-1}$ for an error burst length equal to $n + 1$, and to $1 - (1/2)^n$ for an error burst length greater than $n + 1$.

2.6.4 Hamming Code

We have seen, in the case of the error detection and correction codes described above, how an increase in the number of redundant bits added to message bits can enhance the capability of the code to detect and correct errors. If we have a sufficient number of redundant bits, and if these bits can be arranged such that different error bits produce different error results, then it should be possible not only to detect the error bit but also to identify its location. In fact, the addition of redundant bits alters the 'distance' code parameter, which has come to be known as the Hamming distance. The Hamming distance is nothing but the number of bit disagreements between two code words. For example, the addition of single-bit parity results in a code with a Hamming distance of at least 2. The smallest Hamming distance in the case of a threefold repetition code would be 3. Hamming noticed that an increase in distance enhanced the code's ability to detect and correct errors. Hamming's code was therefore an attempt at increasing the Hamming distance and at the same time having as high an information throughput rate as possible.

The algorithm for writing the generalized Hamming code is as follows:

1. The generalized form of code is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7D_8D_9D_{10}D_{11}P_5\ldots$, where P and D respectively represent parity and data bits.
2. We can see from the generalized form of the code that all bit positions that are powers of 2 (positions 1, 2, 4, 8, 16, ...) are used as parity bits.
3. All other bit positions (positions 3, 5, 6, 7, 9, 10, 11, ...) are used to encode data.
4. Each parity bit is allotted a group of bits from the data bits in the code word, and the value of the parity bit (0 or 1) is used to give it certain parity.
5. Groups are formed by first checking $N - 1$ bits and then alternately skipping and checking N bits following the parity bit. Here, N is the position of the parity bit; 1 for P_1, 2 for P_2, 4 for P_3, 8 for P_4 and so on. For example, for the generalized form of code given above, various groups of bits formed with different parity bits would be $P_1D_1D_2D_4D_5\ldots$, $P_2D_1D_3D_4D_6D_7\ldots$, $P_3D_2D_3D_4D_8D_9\ldots$, $P_4D_5D_6D_7D_8D_9D_{10}D_{11}\ldots$ and so on. To illustrate the formation of groups further, let us examine the group corresponding to parity bit P_3. Now, the position of P_3 is at number 4. In order to form the group, we check the first three bits $(N - 1 = 3)$ and then follow it up by alternately skipping and checking four bits $(N = 4)$.

The Hamming code is capable of correcting single-bit errors on messages of any length. Although the Hamming code can detect two-bit errors, it cannot give the error locations. The number of parity bits required to be transmitted along with the message, however, depends upon the message length, as shown above. The number of parity bits n required to encode m message bits is the smallest integer that satisfies the condition $(2^n - n) > m$.

Table 2.9 Generation of Hamming code.

	P_1	P_2	D_1	P_3	D_2	D_3	D_4
Data bits (without parity)			0		1	1	0
Data bits with parity bit P_1	1		0		1		0
Data bits with parity bit P_2		1	0			1	0
Data bits with parity bit P_3				0	1	1	0
Data bits with parity	1	1	0	0	1	1	0

The most commonly used Hamming code is the one that has a code word length of seven bits with four message bits and three parity bits. It is also referred to as the Hamming (7, 4) code. The code word sequence for this code is written as $P_1P_2D_1P_3D_2D_3D_4$, with P_1, P_2 and P_3 being the parity bits and D_1, D_2, D_3 and D_4 being the data bits. We will illustrate step by step the process of writing the Hamming code for a certain group of message bits and then the process of detection and identification of error bits with the help of an example. We will write the Hamming code for the four-bit message 0110 representing numeral '6'. The process of writing the code is illustrated in Table 2.9, with even parity.

Thus, the Hamming code for 0110 is 1100110. Let us assume that the data bit D_1 gets corrupted in the transmission channel. The received code in that case is 1110110. In order to detect the error, the parity is checked for the three parity relations mentioned above. During the parity check operation at the receiving end, three additional bits X, Y and Z are generated by checking the parity status of $P_1D_1D_2D_4$, $P_2D_1D_3D_4$ and $P_3D_2D_3D_4$ respectively. These bits are a '0' if the parity status is okay, and a '1' if it is disturbed. In that case, *ZYX* gives the position of the bit that needs correction. The process can be best explained with the help of an example.

Examination of the first parity relation gives $X = 1$ as the even parity is disturbed. The second parity relation yields $Y = 1$ as the even parity is disturbed here too. Examination of the third relation gives $Z = 0$ as the even parity is maintained. Thus, the bit that is in error is positioned at 011 which is the binary equivalent of '3'. This implies that the third bit from the MSB needs to be corrected. After correcting the third bit, the received message becomes 1100110 which is the correct code.

Example 2.6

By writing the parity code (even) and threefold repetition code for all possible four-bit straight binary numbers, prove that the Hamming distance in the two cases is at least 2 in the case of the parity code and 3 in the case of the repetition code.

Solution

The generation of codes is shown in Table 2.10. An examination of the parity code numbers reveals that the number of bit disagreements between any pair of code words is not less than 2. It is either 2 or 4. It is 4, for example, between 00000 and 10111, 00000 and 11011, 00000 and 11101, 00000 and 11110 and 00000 and 01111. In the case of the threefold repetition code, it is either 3, 6, 9 or 12 and therefore not less than 3 under any circumstances.

Example 2.7

It is required to transmit letter 'A' expressed in the seven-bit ASCII code with the help of the Hamming (11, 7) code. Given that the seven-bit ASCII notation for 'A' is 1000001 and that the data word gets

Table 2.10 Example 2.6.

Binary number	Parity code	Three-time repetition Code	Binary number	Parity code	Three-time repetition code
0000	00000	000000000000	1000	11000	100010001000
0001	10001	000100010001	1001	01001	100110011001
0010	10010	001000100010	1010	01010	101010101010
0011	00011	001100110011	1011	11011	101110111011
0100	10100	010001000100	1100	01100	110011001100
0101	00101	010101010101	1101	11101	110111011101
0110	00110	011001100110	1110	11110	111011101110
0111	10111	011101110111	1111	01111	111111111111

corrupted to 1010001 in the transmission channel, show how the Hamming code can be used to identify the error. Use even parity.

Solution

- The generalized form of the Hamming code in this case is $P_1P_2D_1P_3D_2D_3D_4P_4D_5D_6D_7 = P_1P_21P_3000P_4001$.
- The four groups of bits using different parity bits are $P_1D_1D_2D_4D_5D_7$, $P_2D_1D_3D_4D_6D_7$, $P_3D_2D_3D_4$ and $P_4D_5D_6D_7$.
- This gives $P_1 = 0$, $P_2 = 0$, $P_3 = 0$ and $P_4 = 1$.
- Therefore, the transmitted Hamming code for 'A' is 00100001001.
- The received Hamming code is 00100101001.
- Checking the parity for the P_1 group gives '0' as it passes the test.
- Checking the parity for the P_2 group gives '1' as it fails the test.
- Checking the parity for the P_3 group gives '1' as it fails the test.
- Checking the parity for the P_4 group gives '0' as it passes the test.
- The bits resulting from the parity check, written in reverse order, constitute 0110, which is the binary equivalent of '6'. This shows that the bit in error is the sixth from the MSB.
- Therefore, the corrected Hamming code is 00100001001, which is the same as the transmitted code.
- The received data word is 1000001.

Review Questions

1. Distinguish between weighted and unweighted codes. Give two examples each of both types of code.
2. What is an excess-3 BCD code? Which shortcoming of the 8421 BCD code is overcome in the excess-3 BCD code? Illustrate with the help of an example.
3. What is the Gray code? Why is it also known as the binary-reflected Gray code? Briefly outline some of the important applications of the Gray code.
4. Briefly describe salient features of the ASCII and EBCDIC codes in terms of their capability to represent characters and suitability for their use in different platforms.
5. What is the Unicode? Why is it called the most complete character code?

6. What is a parity bit? Define even and odd parity. What is the limitation of the parity code when it comes to detection and correction of bit errors?
7. What is the Hamming distance? What is the role of the Hamming distance in deciding the error detection and correction capability of a code meant for the purpose? How does it influence the information throughput rate?
8. With the help of the generalized form of the Hamming code, explain how the number of parity bits required to transmit a given number of data bits is decided upon.

Problems

1. Write the excess-3 equivalent codes of $(6)_{10}$, $(78)_{10}$ and $(357)_{10}$, all in 16-bit format.

   0011001100111001, 0011001110101011, 0011011010001010

2. Determine the Gray code equivalent of $(10011)_2$ and the binary equivalent of the Gray code number 110011.

   11010, $(100010)_2$

3. A 16-bit data word given by 1001100001110110 is to be transmitted by using a fourfold repetition code. If the data word is broken into four blocks of four bits each, then write the transmitted bit stream.

   1001100110011001100010001000100001110111011101110110011001100110

4. Write (a) the Hamming (7, 4) code for 0000 using even parity and (b) the Hamming (11, 7) code for 1111111 using odd parity.

   (a) 0000000; (b) 00101110111

5. Write the last four of the 16 possible numbers in the two-bit quaternary Gray code with 0, 1, 2 and 3 as its independent digits, beginning with the thirteenth number.

   33, 32, 31, 30

Further Reading

1. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Book Companies Inc., USA.
2. Gillam, R. (2002) *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard*, 1st edition, Addison-Wesley Professional, Boston, MA, USA.
3. MacWilliams, F. J. and Sloane, N. J. A. (2006) *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library, Elsevier Ltd, Oxford, UK.
4. Huffman, W. C. and Pless, V. (2003) *Fundamentals of Error-Correcting Codes*, Cambridge University Press, Cambridge, UK.

3

Digital Arithmetic

Having discussed different methods of numeric and alphanumeric data representation in the first two chapters, the next obvious step is to study the rules of data manipulation. Two types of operation that are performed on binary data include arithmetic and logic operations. Basic arithmetic operations include addition, subtraction, multiplication and division. AND, OR and NOT are the basic logic functions. While the rules of arithmetic operations are covered in the present chapter, those related to logic operations will be discussed in the next chapter.

3.1 Basic Rules of Binary Addition and Subtraction

The basic principles of binary addition and subtraction are similar to what we all know so well in the case of the decimal number system. In the case of addition, adding '0' to a certain digit produces the same digit as the sum, and, when we add '1' to a certain digit or number in the decimal number system, the result is the next higher digit or number, as the case may be. For example, 6 + 1 in decimal equals '7' because '7' immediately follows '6' in the decimal number system. Also, 7 + 1 in octal equals '10' as, in the octal number system, the next adjacent higher number after '7' is '10'. Similarly, 9 + 1 in the hexadecimal number system is 'A'. With this background, we can write the basic rules of binary addition as follows:

1. $0 + 0 = 0$.
2. $0 + 1 = 1$.
3. $1 + 0 = 1$.
4. $1 + 1 = 0$ with a carry of '1' to the next more significant bit.
5. $1 + 1 + 1 = 1$ with a carry of '1' to the next more significant bit.

Table 3.1 summarizes the sum and carry outputs of all possible three-bit combinations. We have taken three-bit combinations as, in all practical situations involving the addition of two larger bit

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

Table 3.1 Binary addition of three bits.

A	B	Carry-in (C_{in})	Sum	Carry-out (C_o)	A	B	Carry-in (C_{in})	Sum	Carry-out (C_o)
0	0	0	0	0	1	0	0	1	0
0	0	1	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1
0	1	1	0	1	1	1	1	1	1

numbers, we need to add three bits at a time. Two of the three bits are the bits that are part of the two binary numbers to be added, and the third bit is the carry-in from the next less significant bit column.

The basic principles of binary subtraction include the following:

1. $0 - 0 = 0$.
2. $1 - 0 = 1$.
3. $1 - 1 = 0$.
4. $0 - 1 = 1$ with a borrow of 1 from the next more significant bit.

The above-mentioned rules can also be explained by recalling rules for subtracting decimal numbers. Subtracting '0' from any digit or number leaves the digit or number unchanged. This explains the first two rules. Subtracting '1' from any digit or number in decimal produces the immediately preceding digit or number as the answer. In general, the subtraction operation of larger-bit binary numbers also involves three bits, including the two bits involved in the subtraction, called the minuend (the upper bit) and the subtrahend (the lower bit), and the borrow-in. The subtraction operation produces the difference output and borrow-out, if any. Table 3.2 summarizes the binary subtraction operation. The entries in Table 3.2 can be explained by recalling the basic rules of binary subtraction mentioned above, and that the subtraction operation involving three bits, that is, the minuend (A), the subtrahend (B) and the borrow-in (B_{in}), produces a difference output equal to ($A - B - B_{in}$). It may be mentioned here that, in the case of subtraction of larger-bit binary numbers, the least significant bit column always involves two bits to produce a difference output bit and the borrow-out

Table 3.2 Binary subtraction.

Inputs			Outputs	
Minuend (A)	Subtrahend (B)	Borrow-in (B_{in})	Difference (D)	Borrow-out (B_o)
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

bit. The borrow-out bit produced here becomes the borrow-in bit for the next more significant bit column, and the process continues until we reach the most significant bit column. The addition and subtraction of larger-bit binary numbers is illustrated with the help of examples in sections 3.2 and 3.3 respectively.

3.2 Addition of Larger-Bit Binary Numbers

The addition of larger binary integers, fractions or mixed binary numbers is performed columnwise in just the same way as in the case of decimal numbers. In the case of binary numbers, however, we follow the basic rules of addition of two or three binary digits, as outlined earlier. The process of adding two larger-bit binary numbers can be best illustrated with the help of an example.

Consider two generalized four-bit binary numbers $(A_3\,A_2\,A_1\,A_0)$ and $(B_3\,B_2\,B_1\,B_0)$, with A_0 and B_0 representing the LSB and A_3 and B_3 representing the MSB of the two numbers. The addition of these two numbers is performed as follows. We begin with the LSB position. We add the LSB bits and record the sum S_0 below these bits in the same column and take the carry C_0, if any, to the next column of bits. For instance, if $A_0 = 1$ and $B_0 = 0$, then $S_0 = 1$ and $C_0 = 0$. Next we add the bits A_1 and B_1 and the carry C_0 from the previous addition. The process continues until we reach the MSB bits. The four steps are shown ahead. C_0, C_1, C_2 and C_3 are carrys, if any, produced as a result of adding first, second, third and fourth column bits respectively, starting from LSB and proceeding towards MSB. A similar procedure is followed when the given numbers have both integer as well as fractional parts:

1.
$$\begin{array}{cccc} & & (C_0) & \\ A_3 & A_2 & A_1 & A_0 \\ B_3 & B_2 & B_1 & B_0 \\ \hline & & & S_0 \\ \hline \end{array}$$

2.
$$\begin{array}{ccccc} & & (C_1) & (C_0) & \\ & A_3 & A_2 & A_1 & A_0 \\ & B_3 & B_2 & B_1 & B_0 \\ \hline & & & S_1 & S_0 \\ \hline \end{array}$$

3.
$$\begin{array}{cccc} (C_2) & (C_1) & (C_0) & \\ A_3 & A_2 & A_1 & A_0 \\ B_3 & B_2 & B_1 & B_0 \\ \hline & S_2 & S_1 & S_0 \\ \hline \end{array}$$

4.
$$\begin{array}{ccccc} & (C_2) & (C_1) & (C_0) & \\ & A_3 & A_2 & A_1 & A_0 \\ & B_3 & B_2 & B_1 & B_0 \\ \hline C_3 & S_3 & S_2 & S_1 & S_0 \\ \hline \end{array}$$

3.2.1 Addition Using the 2's Complement Method

The 2's complement is the most commonly used code for processing positive and negative binary numbers. It forms the basis of arithmetic circuits in modern computers. When the decimal numbers to be added are expressed in 2's complement form, the addition of these numbers, following the basic laws of binary addition, gives correct results. Final carry obtained, if any, while adding MSBs should be disregarded. To illustrate this, we will consider the following four different cases:

1. Both the numbers are positive.
2. Larger of the two numbers is positive.
3. The larger of the two numbers is negative.
4. Both the numbers are negative.

Case 1

- Consider the decimal numbers +37 and +18.
- The 2's complement of +37 in eight-bit representation = 00100101.
- The 2's complement of +18 in eight-bit representation = 00010010.
- The addition of the two numbers, that is, +37 and +18, is performed as follows

$$\begin{array}{r} 00100101 \\ +\,\underline{00010010} \\ \underline{00110111} \end{array}$$

- The decimal equivalent of $(00110111)_2$ is (+55), which is the correct answer.

Case 2

- Consider the two decimal numbers +37 and -18.
- The 2's complement representation of +37 in eight-bit representation = 00100101.
- The 2's complement representation of −18 in eight-bit representation = 11101110.
- The addition of the two numbers, that is, +37 and −18, is performed as follows:

$$\begin{array}{r} 00100101 \\ +\,\underline{11101110} \\ \underline{00010011} \end{array}$$

- The final carry has been disregarded.
- The decimal equivalent of $(00010011)_2$ is +19, which is the correct answer.

Case 3

- Consider the two decimal numbers +18 and −37.
- −37 in 2's complement form in eight−bit representation = 11011011.
- +18 in 2's complement form in eight−bit representation = 00010010.
- The addition of the two numbers, that is, −37 and +18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\,\underline{00010010} \\ \underline{11101101} \end{array}$$

- The decimal equivalent of $(11101101)_2$, which is in 2's complement form, is −19, which is the correct answer. 2's complement representation was discussed in detail in Chapter 1 on number systems.

Case 4

- Consider the two decimal numbers −18 and −37.
- −18 in 2's complement form is 11101110.
- −37 in 2's complement form is 11011011.
- The addition of the two numbers, that is, −37 and −18, is performed as follows:

$$\begin{array}{r} 11011011 \\ +\,11101110 \\ \hline 11001001 \\ \hline \end{array}$$

- The final carry in the ninth bit position is disregarded.
- The decimal equivalent of $(11001001)_2$, which is in 2's complement form, is -55, which is the correct answer.

It may also be mentioned here that, in general, 2's complement notation can be used to perform addition when the expected result of addition lies in the range from -2^{n-1} to $+(2^{n-1}-1)$, n being the number of bits used to represent the numbers. As an example, eight-bit 2's complement arithmetic cannot be used to perform addition if the result of addition lies outside the range from -128 to $+127$. Different steps to be followed to do addition in 2's complement arithmetic are summarized as follows:

1. Represent the two numbers to be added in 2's complement form.
2. Do the addition using basic rules of binary addition.
3. Disregard the final carry, if any.
4. The result of addition is in 2's complement form.

Example 3.1

Perform the following addition operations:

1. $(275.75)_{10} + (37.875)_{10}$.
2. $(AF1.B3)_{16} + (FFF.E)_{16}$.

Solution

1. As a first step, the two given decimal numbers will be converted into their equivalent binary numbers (decimal-to-binary conversion has been covered at length in Chapter 1, and therefore the decimal-to-binary conversion details will not be given here):

$$(275.75)_{10} = (100010011.11)_2 \text{ and } (37.875)_{10} = (100101.111)_2$$

The two binary numbers can be rewritten as $(100010011.110)_2$ and $(000100101.111)_2$ to have the same number of bits in their integer and fractional parts. The addition of two numbers is performed as follows:

$$\begin{array}{r} 100010011.110 \\ 000100101.111 \\ \hline 100111001.101 \\ \hline \end{array}$$

The decimal equivalent of $(100111001.101)_2$ is $(313.625)_{10}$.

2. $(AF1.B3)_{16} = (101011110001.10110011)_2$ and $(FFF.E)_{16} = (111111111111.1110)_2$. $(111111111111.1110)_2$ can also be written as $(111111111111.11100000)_2$ to have the same number of bits in the integer and fractional parts. The two numbers can now be added as follows:

$$\begin{array}{r} 0101011110001.10110011 \\ \underline{0111111111111.11100000} \\ \underline{1101011110001.10010011} \end{array}$$

The hexadecimal equivalent of $(1101011110001.10010011)_2$ is $(1AF1.93)_{16}$, which is equal to the hex addition of $(AF1.B3)_{16}$ and $(FFF.E)_{16}$.

Example 3.2

Find out whether 16-bit 2's complement arithmetic can be used to add 14 276 and 18 490.

Solution

The addition of decimal numbers 14 276 and 18 490 would yield 32 766. 16-bit 2's complement arithmetic has a range of -2^{15} to $+(2^{15} - 1)$, i.e. $-32\,768$ to $+32\,767$. The expected result is inside the allowable range. Therefore, 16-bit arithmetic can be used to add the given numbers.

Example 3.3

Add −118 and −32 firstly using eight-bit 2's complement arithmetic and then using 16-bit 2's complement arithmetic. Comment on the results.

Solution

- −118 in eight-bit 2's complement representation = 10001010.
- −32 in eight-bit 2's complement representation = 11100000.
- The addition of the two numbers, after disregarding the final carry in the ninth bit position, is 01101010. Now, the decimal equivalent of $(01101010)_2$, which is in 2's complement form, is +106. The reason for the wrong result is that the expected result, i.e. −150, lies outside the range of eight-bit 2's complement arithmetic. Eight-bit 2's complement arithmetic can be used when the expected result lies in the range from -2^7 to $+ (2^7 - 1)$, i.e. −128 to +127. −118 in 16-bit 2's complement representation = 1111111110001010.
- −32 in 16-bit 2's complement representation = 1111111111100000.
- The addition of the two numbers, after disregarding the final carry in the 17th position, produces 1111111101101010. The decimal equivalent of $(1111111101101010)_2$, which is in 2's complement form, is −150, which is the correct answer. 16-bit 2's complement arithmetic has produced the correct result, as the expected result lies within the range of 16-bit 2's complement notation.

3.3 Subtraction of Larger-Bit Binary Numbers

Subtraction is also done columnwise in the same way as in the case of the decimal number system. In the first step, we subtract the LSBs and subsequently proceed towards the MSB. Wherever the subtrahend (the bit to be subtracted) is larger than the minuend, we borrow from the next adjacent higher bit position having a '1'. As an example, let us go through different steps of subtracting $(1001)_2$ from $(1100)_2$.

In this case, '1' is borrowed from the second MSB position, leaving a '0' in that position. The borrow is first brought to the third MSB position to make it '10'. Out of '10' in this position, '1' is taken to the LSB position to make '10' there, leaving a '1' in the third MSB position. $10-1$ in the LSB column gives '1', $1-0$ in the third MSB column gives '1', $0-0$ in the second MSB column gives '0' and $1-1$ in the MSB also gives '0' to complete subtraction. Subtraction of mixed numbers is also done in the same manner. The above-mentioned steps are summarized as follows:

1.
$$\begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline & & & \mathbf{1} \\ \hline \end{array}$$

2.
$$\begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline & & \mathbf{1} & \mathbf{1} \\ \hline \end{array}$$

3.
$$\begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \hline \end{array}$$

4.
$$\begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \hline \end{array}$$

3.3.1 Subtraction Using 2's Complement Arithmetic

Subtraction is similar to addition. Adding 2's complement of the subtrahend to the minuend and disregarding the carry, if any, achieves subtraction. The process is illustrated by considering six different cases:

1. Both minuend and subtrahend are positive. The subtrahend is the smaller of the two.
2. Both minuend and subtrahend are positive. The subtrahend is the larger of the two.
3. The minuend is positive. The subtrahend is negative and smaller in magnitude.
4. The minuend is positive. The subtrahend is negative and greater in magnitude.
5. Both minuend and subtrahend are negative. The minuend is the smaller of the two.
6. Both minuend and subtrahend are negative. The minuend is the larger of the two.

Case 1

- Let us subtract +14 from +24.
- The 2's complement representation of $+24 = 00011000$.
- The 2's complement representation of $+14 = 00001110$.
- Now, the 2's complement of the subtrahend (i.e. +14) is 11110010.
- Therefore, $+24 - (+14)$ is given by

$$\begin{array}{r} 00011000 \\ +\ 11110010 \\ \hline \underline{00001010} \end{array}$$

  with the final carry disregarded.
- The decimal equivalent of $(00001010)_2$ is +10, which is the correct answer.

Case 2

- Let us subtract +24 from +14.
- The 2's complement representation of +14 = 00001110.
- The 2's complement representation of +24 = 00011000.
- The 2's complement of the subtrahend (i.e. +24) = 11101000.
- Therefore, +14 − (+24) is given by

$$\begin{array}{r} 00001110 \\ +\,\underline{11101000} \\ \underline{11110110} \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is of course in 2's complement form, is −10 which is the correct answer.

Case 3

- Let us subtract −14 from +24.
- The 2's complement representation of +24 = 00011000 = minuend.
- The 2's complement representation of −14 = 11110010 = subtrahend.
- The 2's complement of the subtrahend (i.e. −14) = 00001110.
- Therefore, +24 − (−14) is performed as follows:

$$\begin{array}{r} 00011000 \\ +\,\underline{00001110} \\ \underline{00100110} \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 4

- Let us subtract −24 from +14.
- The 2's complement representation of +14 = 00001110 = minuend.
- The 2's complement representation of −24 = 11101000 = subtrahend.
- The 2's complement of the subtrahend (i.e. −24) = 00011000.
- Therefore, +14 − (−24) is performed as follows:

$$\begin{array}{r} 00001110 \\ +\,\underline{00011000} \\ \underline{00100110} \end{array}$$

- The decimal equivalent of $(00100110)_2$ is +38, which is the correct answer.

Case 5

- Let us subtract −14 from −24.
- The 2's complement representation of −24 = 11101000 = minuend.

- The 2's complement representation of -14=11110010 = subtrahend.
- The 2's complement of the subtrahend = 00001110.
- Therefore, $-24 - (-14)$ is given as follows:

$$\begin{array}{r} 11101000 \\ +\,00001110 \\ \hline 11110110 \end{array}$$

- The decimal equivalent of $(11110110)_2$, which is in 2's complement form, is -10, which is the correct answer.

Case 6

- Let us subtract -24 from -14.
- The 2's complement representation of -14 = 11110010 = minuend.
- The 2's complement representation of -24=11101000 = subtrahend.
- The 2's complement of the subtrahend = 00011000.
- Therefore, $-14 - (-24)$ is given as follows:

$$\begin{array}{r} 11110010 \\ +\,00011000 \\ \hline 00001010 \end{array}$$

  with the final carry disregarded.
- The decimal equivalent of $(00001010)_2$, which is in 2's complement form, is $+10$, which is the correct answer.

It may be mentioned that, in 2's complement arithmetic, the answer is also in 2's complement notation, only with the MSB indicating the sign and the remaining bits indicating the magnitude. In 2's complement notation, positive magnitudes are represented in the same way as the straight binary numbers, while the negative magnitudes are represented as the 2's complement of their straight binary counterparts. A '0' in the MSB position indicates a positive sign, while a '1' in the MSB position indicates a negative sign.

The different steps to be followed to do subtraction in 2's complement arithmetic are summarized as follows:

1. Represent the minuend and subtrahend in 2's complement form.
2. Find the 2's complement of the subtrahend.
3. Add the 2's complement of the subtrahend to the minuend.
4. Disregard the final carry, if any.
5. The result is in 2's complement form.
6. 2's complement notation can be used to perform subtraction when the expected result of subtraction lies in the range from -2^{n-1} to $+(2^{n-1}-1)$, n being the number of bits used to represent the numbers.

Example 3.4

Subtract $(1110.011)_2$ from $(11011.11)_2$ using basic rules of binary subtraction and verify the result by showing equivalent decimal subtraction.

Solution

The minuend and subtrahend are first modified to have the same number of bits in the integer and fractional parts. The modified minuend and subtrahend are $(11011.110)_2$ and $(01110.011)_2$ respectively:

$$\begin{array}{r} 11011.110 \\ -\ \underline{01110.011} \\ \underline{01101.011} \end{array}$$

The decimal equivalents of $(11011.110)_2$ and $(01110.011)_2$ are 27.75 and 14.375 respectively. Their difference is 13.375, which is the decimal equivalent of $(01101.011)_2$.

Example 3.5

Subtract (a) $(-64)_{10}$ from $(+32)_{10}$ and (b) $(29.A)_{16}$ from $(4F.B)_{16}$. Use 2's complement arithmetic.

Solution:

(a) $(+32)_{10}$in 2's complement notation $= (00100000)_2$.
$(-64)_{10}$ in 2's complement notation $= (11000000)_2$.
The 2's complement of $(-64)_{10} = (01000000)_2$.
$(+32)_{10} - (-64)_{10}$ is determined by adding the 2's complement of $(-64)_{10}$ to $(+32)_{10}$.
Therefore, the addition of $(00100000)_2$ to $(01000000)_2$ should give the result. The operation is shown as follows:

$$\begin{array}{r} 00100000 \\ +\ \underline{01000000} \\ \underline{01100000} \end{array}$$

The decimal equivalent of $(01100000)_2$ is +96, which is the correct answer as $+32 - (-64) = +96$.

(b) The minuend $= (4F.B)_{16} = (01001111.1011)_2$.
The minuend in 2's complement notation $= (01001111.1011)_2$.
The subtrahend $= (29.A)_{16} = (00101001.1010)_2$.
The subtrahend in 2's complement notation $= (00101001.1010)_2$.
The 2's complement of the subtrahend $= (11010110.0110)_2$.
$(4F.B)_{16} - (29.A)_{16}$ is given by the addition of the 2's complement of the subtrahend to the minuend.

$$\begin{array}{r} 01001111.1011 \\ +\ \underline{11010110.0110} \\ \underline{00100110.0001} \end{array}$$

with the final carry disregarded. The result is also in 2's complement form. Since the result is a positive number, 2's complement notation is the same as it would be in the case of the straight binary code.
The hex equivalent of the resulting binary number $= (26.1)_{16}$, which is the correct answer.

3.4 BCD Addition and Subtraction in Excess-3 Code

Below, we will see how the excess-3 code can be used to perform addition and subtraction operations on BCD numbers.

3.4.1 Addition

The excess-3 code can be very effectively used to perform the addition of BCD numbers. The steps to be followed for excess-3 addition of BCD numbers are as follows:

1. The given BCD numbers are written in excess-3 form by adding '0011' to each of the four-bit groups.
2. The two numbers are then added using the basic laws of binary addition.
3. Add '0011' to all those four-bit groups that produce a carry, and subtract '0011' from all those four-bit groups that do not produce a carry during addition.
4. The result thus obtained is in excess-3 form.

3.4.2 Subtraction

Subtraction of BCD numbers using the excess-3 code is similar to the addition process discussed above. The steps to be followed for excess-3 substraction of BCD numbers are as follows:

1. Express both minuend and subtrahend in excess-3 code.
2. Perform subtraction following the basic laws of binary subtraction.
3. Subtract '0011' from each invalid BCD four-bit group in the answer.
4. Subtract '0011' from each BCD four-bit group in the answer if the subtraction operation of the relevant four-bit groups required a borrow from the next higher adjacent four-bit group.
5. Add '0011' to the remaining four-bit groups, if any, in the result.
6. This gives the result in excess-3 code.

The process of addition and subtraction can be best illustrated with the help of following examples.

Example 3.6

Add $(0011\ 0101\ 0110)_{BCD}$ *and* $(0101\ 0111\ 1001)_{BCD}$ *using the excess-3 addition method and verify the result using equivalent decimal addition.*

Solution

The excess-3 equivalents of 0011 0101 0110 and 0101 0111 1001 are 0110 1000 1001 and 1000 1010 1100 respectively. The addition of the two excess-3 numbers is given as follows:

$$\begin{array}{r} 0110\ 1000\ 1001 \\ 1000\ 1010\ 1100 \\ \hline 1111\ 0011\ 0101 \\ \hline \end{array}$$

After adding 0011 to the groups that produced a carry and subtracting 0011 from the groups that did not produce a carry, we obtain the result of the above addition as 1100 0110 1000. Therefore, 1100

0110 1000 represents the excess-3 code for the true result. The result in BCD code is 1001 0011 0101, which is the BCD equivalent of 935. This is the correct answer as the addition of the given BCD numbers 0011 0101 0110 = $(356)_{10}$ and 0101 0111 1001 = $(579)_{10}$ yields $(935)_{10}$ only.

Example 3.7

Perform $(185)_{10}-(8)_{10}$ using the excess-3 code.

Solution

- $(185)_{10} = (0001\ 1000\ 0101)_{BCD}$.The excess-3 equivalent of $(0001\ 1000\ 0101)_{BCD}$ = 0100 1011 1000.
- $(8)_{10} = (008)_{10} = (0000\ 0000\ 1000)_{BCD}$. The excess-3 equivalent of $(0000\ 0000\ 1000)_{BCD}$ = 0011 0011 1011.
- Subtraction is performed as follows:

$$\begin{array}{r} 0100\ 1011\ 1000 \\ -\ 0011\ 0011\ 1011 \\ \hline 0001\ 0111\ 1101 \\ \hline \end{array}$$

- In the subtraction operation, the least significant column of four-bit groups needed a borrow, while the other two columns did not need any borrow. Also, the least significant column has produced an invalid BCD code group. Subtracting 0011 from the result of this column and adding 0011 to the results of other two columns, we get 0100 1010 1010. This now constitutes the result of subtraction expressed in excess-3 code.
- The result in BCD code is therefore 0001 0111 0111.
- The decimal equivalent of 0001 0111 0111 is 177, which is the correct result.

3.5 Binary Multiplication

The basic rules of binary multiplication are governed by the way an AND gate functions when the two bits to be multiplied are fed as inputs to the gate. Logic gates are discussed in detail in the next chapter. As of now, it would suffice to say that the result of multiplying two bits is the same as the output of the AND gate with the two bits applied as inputs to the gate. The basic rules of multiplication are listed as follows:

1. $0 \times 0 = 0$.
2. $0 \times 1 = 0$.
3. $1 \times 0 = 0$.
4. $1 \times 1 = 1$.

One of the methods for multiplication of larger-bit binary numbers is similar to what we are familiar with in the case of decimal numbers. This is called the 'repeated left-shift and add' algorithm. Microprocessors and microcomputers, however, use what is known as the 'repeated add and right-shift' algorithm to do binary multiplication as it is comparatively much more convenient to implement than the 'repeated left-shift and add' algorithm. The two algorithms are briefly described below. Also, binary multiplication of mixed binary numbers is done by performing multiplication without considering the binary point. Starting from the LSB, the binary point is then placed after n bits, where n is equal to the sum of the number of bits in the fractional parts of the multiplicand and multiplier.

3.5.1 Repeated Left-Shift and Add Algorithm

In the 'repeated left-shift and add' method of binary multiplication, the end-product is the sum of several partial products, with the number of partial products being equal to the number of bits in the multiplier binary number. This is similar to the case of decimal multiplication. Each successive partial product after the first is shifted one digit to the left with respect to the immediately preceding partial product. In the case of binary multiplication too, the first partial product is obtained by multiplying the multiplicand binary number by the LSB of the multiplier binary number. The second partial product is obtained by multiplying the multiplicand binary number by the next adjacent higher bit in the multiplier binary number and so on. We begin with the LSB of the multiplier to obtain the first partial product. If the LSB is a '1', a copy of the multiplicand forms the partial product, and it is an all '0' sequence if the LSB is a '0'. We proceed towards the MSB of the multiplier and obtain various partial products. The second partial product is shifted one bit position to the left relative to the first partial product; the third partial product is shifted one bit position to the left relative to the second partial product and so on. The addition of all partial products gives the final answer. If the multiplicand and multiplier have different signs, the end result has a negative sign, otherwise it is positive. The procedure is further illustrated by showing $(23)_{10} \times (6)_{10}$ multiplication.

$$
\begin{array}{rr l}
 & 1\,0\,1\,1\,1 & \\
\textit{Multiplicand}: & \times\,1\,1\,0 & \ldots\ldots\ldots\ldots\ (23)_{10} \\
\textit{Multiplier}: & \overline{} & \ldots\ldots\ldots\ldots\ (6)_{10} \\
 & 0\,0\,0\,0\,0 & \\
 & 1\,0\,1\,1\,1 & \\
 & 1\,0\,1\,1\,1 & \\
\hline
 & 1\,0\,0\,0\,1\,0\,1\,0 &
\end{array}
$$

The decimal equivalent of $(10001010)_2$ is $(138)_{10}$, which is the correct result.

3.5.2 Repeated Add and Right-Shift Algorithm

The multiplication process starts with writing an all '0' bit sequence, with the number of bits equal to the number of bits in the multiplicand. This bit sequence (all '0' sequence) is added to another same-sized bit sequence, which is the same as the multiplicand if the LSB of the multiplier is a '1', and an all '0' sequence if it is a '0'. The result of the first addition is shifted one bit position to the right, and the bit shifted out is recorded. The vacant MSB position is replaced by a '0'. This new sequence is added to another sequence, which is an all '0' sequence if the next adjacent higher bit in the multiplier is a '0', and the same as the multiplicand if it is a '1'. The result of the second addition is also shifted one bit position to the right, and a new sequence is obtained. The process continues until all multiplier bits are exhausted. The result of the last addition together with the recorded bits constitutes the result of multiplication. We will illustrate the procedure by doing $(23)_{10} \times (6)_{10}$ multiplication again, this time by using the 'repeated add and right-shift' algorithm:

- The multiplicand $= (23)_{10} = (10111)_2$ and the multiplier $= (6)_{10} = (110)_2$. The multiplication process is shown in Table 3.3.
- Therefore, $(10111)_2 \times (110)_2 = (10001010)_2$.

Table 3.3 Multiplication using the repeated add and right-shift algorithm.

1 0 1 1 1 1 1 0	Multiplicand Multiplier
0 0 0 0 0 + 0 0 0 0 0	Start
0 0 0 0 0	Result of first addition
0 0 0 0 0 + 1 0 1 1 1	0 (Result of addition shifted one bit to right)
1 0 1 1 1	Result of second addition
0 1 0 1 1 + 1 0 1 1 1	10 (Result of addition shifted one bit to right)
1 0 0 0 1 0	Result of third addition
0 1 0 0 0 1	010 (Result of addition shifted one bit to right)

Example 3.8

Multiply (a) $(100.01)_2 \times (10.1)_2$ *by using the 'repeated add and left-shift' algorithm and (b)* $(2B)_{16} \times (3)_{16}$ *by using the 'add and right-shift' algorithm. Verify the results by showing equivalent decimal multiplication.*

Solution

(a) As a first step, we will multiply $(10001)_2$ by $(101)_2$. The process is shown as follows:

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

The multiplication result is then given by placing the binary point three bits after the LSB, which gives $(1010.101)_2$ as the final result. Also, $(100.01)_2 = (4.25)_{10}$ and $(10.1)_2 = (2.5)_{10}$. Moreover, $(4.25)_{10} \times (2.5)_{10} = (10.625)_{10}$ and $(1010.101)_2$ equals $(10.625)_{10}$, which verifies the result.

(b) $(2B)_{16} = 00101011 = 101011$ and $(3)_{16} = 0011 = 11$.
Different steps involved in the multiplication process are shown in Table 3.4.
The result of multiplication is therefore $(10000001)_2$. Also, $(2B)_{16} = (43)_{10}$ and $(3)_{16} = (3)_{10}$. Therefore, $(2B)_{16} \times (3)_{16} = (129)_{10}$. Moreover, $(10000001)_2 = (129)_{10}$, which verifies the result.

3.6 Binary Division

While binary multiplication is the process of repeated addition, binary division is the process of repeated subtraction. Binary division can be performed by using either the 'repeated right-shift and

Table 3.4 Example 3.8.

1 0 1 0 1 1	Multiplicand
1 1	Multiplier
0 0 0 0 0 0	Start
+ 1 0 1 0 1 1	
1 0 1 0 1 1	Result of first addition
0 1 0 1 0 1	1 (Result of addition shifted one bit to right)
+ 1 0 1 0 1 1	
1 0 0 0 0 0 0	Result of second addition
0 1 0 0 0 0 0	01 (Result of addition shifted one bit to right)

subtract' or the 'repeated subtract and left-shift' algorithm. These are briefly described and suitably illustrated in the following sections.

3.6.1 Repeated Right-Shift and Subtract Algorithm

The algorithm is similar to the case of conventional division with decimal numbers. At the outset, starting from MSB, we begin with the number of bits in the dividend equal to the number of bits in the divisor and check whether the divisor is smaller or greater than the selected number of bits in the dividend. If it happens to be greater, we record a '0' in the quotient column. If it is smaller, we subtract the divisor from the dividend bits and record a '1' in the quotient column. If it is greater and we have already recorded a '0', then, as a second step, we include the next adjacent bit in the dividend bits, shift the divisor to the right by one bit position and again make a similar check like the one made in the first step. If it is smaller and we have made the subtraction, then in the second step we append the next MSB of the dividend to the remainder, shift the divisor one bit to the right and again make a similar check. The options are again the same. The process continues until we have exhausted all the bits in the dividend. We will illustrate the algorithm with the help of an example. Let us consider the division of $(100110)_2$ by $(1100)_2$. The sequence of operations needed to carry out the above division is shown in Table 3.5. The quotient = 011 and the remainder = 10.

Table 3.5 Binary division using the repeated right-shift and subtract algorithm.

	Quotient		
First step	0	1 0 0 1 1 0	Dividend
		−1 1 0 0	Divisor
Second step	1	1 0 0 1 1	First five MSBs of dividend
		−1 1 0 0	Divisor shifted to right
		0 1 1 1	First subtraction remainder
Third step	1	0 1 1 1 0	Next MSB appended
		−1 1 0 0	Divisor right shifted
		0 0 1 0	Second subtraction remainder

Table 3.6 Binary division using the repeate subtract and left-shift algorithm.

Quotient	1 0 0 1 −1 1 0 0	1 0
0	1 1 0 1 +1 1 0 0	Borrow exists
	1 0 0 1	Final carry ignored
	1 0 0 1 1 −1 1 0 0	Next MSB appended
1	0 1 1 1	No borrow
	0 1 1 1 0 −1 1 0 0	Next MSB appended
1	0 0 0 1 0	No borrow

3.6.2 Repeated Subtract and Left-Shift Algorithm

The procedure can again be best illustrated with the help of an example. Let us consider solving the above problem using this algorithm. The steps needed to perform the division are as follows. We begin with the first four MSBs of the dividend, four because the divisor is four bits long. In the first step, we subtract the divisor from the dividend. If the subtraction requires borrow in the MSB position, enter a '0' in the quotient column; otherwise, enter a '1'. In the present case there exists a borrow in the MSB position, and so there is a '0' in the quotient column. If there is a borrow, the divisor is added to the result of subtraction. In doing so, the final carry, if any, is ignored. The next MSB is appended to the result of the first subtraction if there is no borrow, or to the result of subtraction, restored by adding the divisor, if there is a borrow. By appending the next MSB, the remaining bits of the dividend are one bit position shifted to the left. It is again compared with the divisor, and the process is repeated. It goes on until we have exhausted all the bits of the dividend. The final remainder can be further processed by successively appending 0s and trying subtraction to get fractional part bits of the quotient. The different steps are summarized in Table 3.6. The quotient = 011 and the remainder = 10.

Example 3.9

Use the 'repeated right-shift and subtract' algorithm to divide $(110101)_2$ by $(1011)_2$. Determine both the integer and the fractional parts of the quotient. The fractional part may be determined up to three bit places.

Solution

The sequence of operations is given in Table 3.7. The operations are self-explanatory.

- The quotient = 100.110.
- Now, $(110101)_2 = (53)_{10}$ and $(1011)_2 = (11)_{10}$.
- $(53)_{10}$ divided by $(11)_{10}$ gives $(4.82)_{10}$.
- $(100.110)_2 = (4.75)_{10}$, which matches with the expected result to a good approximation.

Table 3.7 Example 3.9.

	Quotient		
First step	1	1 1 0 1 0 1 −1 0 1 1	Dividend Divisor
		0 0 1 0	First subtraction
Second step	0	0 0 1 0 0 −1 0 1 1	Next MSB appended Divisor right shifted
Third step	0	0 0 1 0 0 1 −1 0 1 1	Next MSB appended Divisor right shifted
		0 0 1 0 0 1	All bits exhausted
	1	0 0 1 0 0 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 1 1 1	Second subtraction
Fourth step	1	0 1 1 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 0 0 1 1	Third subtraction
Fifth step	0	0 0 0 1 1 0 −1 0 1 1	'0' appended Divisor right shifted
		0 0 1 1	Fourth subtraction

Example 3.10

Use the 'repeated subtract and left-shift' algorithm to divide $(100011)_2$ by $(100)_2$ to determine both the integer and fractional parts of the quotient. Verify the result by showing equivalent decimal division. Determine the fractional part to two bit places.

Solution

The sequence of operations is given in Table 3.8. The operations are self-explanatory.

- The quotient $= (1000.11)_2 = (8.75)_{10}$.
- Now, $(100011)_2 = (35)_{10}$ and $(100)_2 = (4)_{10}$.
- $(35)_{10}$ divided by $(4)_{10}$ gives $(8.75)_{10}$ and hence is verified.

Example 3.11

Divide $(AF)_{16}$ by $(09)_{16}$ using the method of 'repeated right shift and subtract', bearing in mind the signs of the given numbers, assuming that we are working in eight-bit 2's complement arithmetic.

Solution

- The dividend $= (AF)_{16}$.
- As it is a negative hexadecimal number, the magnitude of this number is determined by its 2's complement (or more precisely by its 16's complement in hexadecimal number language).

Table 3.8 Example 3.10.

Quotient	1 0 0 −1 0 0	0 1 1 Dividend Divisor
1	0 0 0	No borrow
	0 0 0 0 −1 0 0	Next MSB appended
0	1 0 0 +1 0 0	Borrow exists
	0 0 0 0 0 0 1 −1 0 0	Final carry ignored Next MSB appended
0	1 0 1 + 1 0 0	Borrow exists
	0 0 1 0 0 1 1 − 1 0 0	Final carry ignored Next MSB appended
0	1 1 1 +1 0 0	Borrow exists
	0 1 1 0 1 1 0 − 1 0 0	Final carry ignored ‘0’ appended
1	0 1 0	No borrow
	0 1 0 0 −1 0 0	‘0’ appended
1	0 0 0	No borrow

- The 16’s complement of $(AF)_{16} = (51)_{16}$.
- The binary equivalent of $(51)_{16} = 01010001 = 1010001$.
- The divisor $= (09)_{16}$.
- It is a positive number.
- The binary equivalent of $(09)_{16} = 00001001$.
- As the dividend is a negative number and the divisor a positive number, the quotient will be a negative number. The division process using the ‘repeated right-shift and subtract’ algorithm is given in Table 3.9.
- The quotient $= 1001 = (09)_{16}$.
- As the quotient should be a negative number, its magnitude is given by the 16’s complement of $(09)_{16}$, i.e. $(F7)_{16}$.
- Therefore, $(AF)_{16}$ divided by $(09)_{16}$ gives $(F7)_{16}$.

3.7 Floating-Point Arithmetic

Before performing arithmetic operations on floating-point numbers, it is necessary to make a few checks, such as finding the signs of the two mantissas, checking any possible misalignment of exponents, etc.

Table 3.9 Example 3.11

1	1 0 1 0 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 1	
0	0 0 0 1 0 −1 0 0 1	Divisor greater than dividend
0	0 0 0 1 0 0 −1 0 0 1	Divisor still greater
1	0 0 0 1 0 0 1 −1 0 0 1	Divisor less than dividend
	0 0 0 0 0 0 0	

For example, if the exponents of the two numbers are not equal, the addition and subtraction operations necessitate that they be made equal. In that case, the mantissa of the smaller of the two numbers is shifted right, and the exponent is incremented for each shift until the two exponents are equal. Once the binary points are aligned and the exponents made equal, addition and subtraction operations become straightforward. While doing subtraction, of course, a magnitude check is also required to determine the smaller of the two numbers.

3.7.1 Addition and Subtraction

If N_1 and N_2 are two floating-point numbers given by

$$N_1 = m_1 \times 2^e$$

$$N_2 = m_2 \times 2^e$$

then

$$N_1 + N_2 = m_1 \times 2^e + m_2 \times 2^e = (m_1 + m_2) \times 2^e$$

and

$$N_1 - N_2 = m_1 \times 2^e - m_2 \times 2^e = (m_1 - m_2) \times 2^e$$

The subtraction operation assumes that $N_1 > N_2$. Post-normalization of the result may be required after the addition or subtraction operation.

3.7.2 Multiplication and Division

In the case of multiplication of two floating-point numbers, the mantissas of the two numbers are multiplied and their exponents are added. In the case of a division operation, the mantissa of the quotient is given by the division of the two mantissas (i.e. dividend mantissa divided by divisor mantissa) and the exponent of the quotient is given by subtraction of the two exponents (i.e. dividend exponent minus divisor exponent).

If

$$N_1 = m_1 \times 2^{e1} and\ N_2 = m_2 \times 2^{e2}$$

then

$$N_1 \times N_2 = (m_1 \times m_2) \times 2^{(e1+e2)}$$

and

$$N_1/N_2 = (m_1/m_2) \times 2^{(e1-e2)}$$

Again, post-normalization may be required after multiplication or division, as in the case of addition and subtraction operations.

Example 3.12

Add (a) $(39)_{10}$ and $(19)_{10}$ and (b) $(1E)_{16}$ and $(F3)_{16}$ using floating-point numbers. Verify the answers by performing equivalent decimal addition.

Solution

(a) $(39)_{10} = 100111 = 0.100111 \times 2^6$.
$(19)_{10} = 10011 = 0.10011 \times 2^5 = 0.010011 \times 2^6$.
Therefore, $(39)_{10} + (19)_{10} = 0.100111 \times 2^6 + 0.010011 \times 2^6$
$= (0.100111 + 0.010011) \times 2^6 = 0.111010 \times 2^6$
$= 111010 = (58)_{10}$
and hence is verified.

(b) $(1E)_{16} = (00011110)_2 = 0.00011110 \times 2^8$.
$(F3)_{16} = (11110011)_2 = 0.11110011 \times 2^8$.
$(1E)_{16} + (F3)_{16} = (0.00011110 + 0.11110011) \times 2^8 = 100010001$
$= 000100010001$
$= (111)_{16}$.
Also, $(1E)_{16} + (\mathrm{F3})_{16} = (111)_{16}$ and hence is proved.

Example 3.13

Subtract $(17)_8$ from $(21)_8$ using floating-point numbers and verify the answer.

Solution

- $(21)_8 = (010001)_2 = 0.010001 \times 2^6$.
- $(17)_8 = (001111)_2 = 0.001111 \times 2^6$.
- Therefore, $(21)_8 - (17)_8 = (0.010001 - 0.001111) \times 2^6$
$= 0.000010 \times 2^6 = 000010 = (02)_8$.
- Also, $(21)_8 - (17)_8 = (02)_8$ and hence is verified.

Example 3.14

Multiply $(37)_{10}$ by $(10)_{10}$ using floating-point numbers. Verify by showing equivalent decimal multiplication.

Solution

- The multiplicand = $(37)_{10} = (100101)_2 = 0.100101 \times 2^6$.
- The multiplier = $(10)_{10} = (1010)_2 = 0.1010 \times 2^4$.
- $(37)_{10} \times (10)_{10} = (0.100101 \times 0.1010) \times 2^{10} = 0.0101110010 \times 2^{10} = 101110010$
  $= (370)_{10}$ and hence is verified.

Example 3.15

Perform $(E3B)_{16} \div (1A)_{16}$ using binary floating-point numbers. Verify by showing equivalent decimal division.

Solution

- Dividend = $(\mathrm{E3B})_{16} = (111000111011)_2 = 0.111000111011 \times 2^{12}$.
- Divisor = $(1\mathrm{A})_{16} = (00011010)_2 = (11010)_2 = 0.11010 \times 2^5$.
- Therefore, $(\mathrm{E3B})_{16} \div (1\mathrm{A})_{16} = (0.111000111011 \div 0.11010) \times 2^7$.
- By performing division of the mantissas using either of the two division algorithms described earlier, we obtain the result of division as $(10001100.00011)_2$.
- $(10001100.00011)_2 = (140.093)_{10}$.
- Also, $(\mathrm{E3B})_{16} = (3643)_{10}$ and $(1A)_{16} = (26)_{10}$.
- $(\mathrm{E3B})_{16} \div (1\mathrm{A})_{16} = (3643)_{10} \div (26)_{10} = (140.1)_{10}$, which is the same as the result obtained with binary floating-point arithmetic to a good approximation.

Review Questions

1. Outline the different steps involved in the addition of larger-bit binary numbers for the following two cases:

   (a) The larger of the two numbers is positive and the other number is negative.
   (b) The larger of the two numbers is negative and the other number is positive.

2. Outline the different steps involved in the subtraction of larger-bit binary numbers for the following two cases:

   (a) The minuend is positive. The subtrahend is negative and smaller in magnitude.
   (b) The minuend is positive. The subtrahend is negative and larger in magnitude.

3. What decides whether a particular binary addition or subtraction operation would be possible with 2's complement arithmetic?
4. Why in microprocessors and microcomputers is the 'repeated add and right-shift' algorithm preferred over the 'repeated left-shift and add' algorithm for binary multiplication? Briefly outline the procedure for multiplication in the case of the former.

5. Prove that the largest six-digit hexadecimal number when subtracted from the largest eight-digit octal number yields zero in decimal.

Problems

1. Perform the following operations using 2's complement arithmetic. The numbers are represented using 2's or 10's or 16's complement notation as the case may be. Express the result both in 2's complement binary as well as in decimal.

   (a) $(7F)_{16} + (A1)_{16}$.
   (b) $(110)_{10} + (0111)_2$.

   (a) $(00100000)_2, (32)_{10}$; (b) $(01110101)_2, (117)_{10}$

2. Evaluate the following to two binary places:

   (a) $(100.0001)_2 \div (10.1)_2$.
   (b) $(111001)_2 \div (1001)_2$.
   (c) $(111.001)_2 \times (1.11)_2$.

   (a) 1.10; (b) 110.01; (c) 1100.01

3. Prove that 16-bit 2's complement arithmetic cannot be used to add +18 150 and +14 618, while it can be used to add −18 150 and −14 618.
4. Add the maximum positive integer to the minimum negative integer, both represented in 16-bit 2's complement binary notation. Express the answer in 2's complement binary.

   1111111111111111

5. The result of adding two BCD numbers represented in excess-3 code is 0111 1011 when the two numbers are added using simple binary addition. If one of the numbers is $(12)_{10}$, find the other.

   $(03)_{10}$

6. Perform the following operations using 2's complement arithmetic:

   (a) $(+43)_{10} - (-53)_{10}$.
   (b) $(1ABC)_{16} + (1DEF)_{16}$.
   (c) $(3E91)_{16} - (1F93)_{16}$.

   (a) 01100000; (b) $(38AB)_{16}$; (c) $(1EFE)_{16}$

Further Reading

1. Ercegovac, M. D. and Lang, T. (2003) *Digital Arithmetic*, Morgan Kaufmann Publishers, CA, USA.
2. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
3. Ashmila, E. M., Dlay, S. S. and Hinton, O. R. (2005) 'Adder methodology and design using probabilistic multiple carry estimates'. *IET Computers and Digital Techniques*, **152**(6), pp. 697–703.
4. Lu, M. (2005) *Arithmetic and Logic in Computer Systems,* John Wiley & Sons, Inc., NJ, USA.

4

Logic Gates and Related Devices

Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinational logic. There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates that are derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVE-OR gate and the EXCLUSIVE-NOR gate. This chapter deals with logic gates and some related devices such as buffers, drivers, etc., as regards their basic functions. The treatment of the subject matter is mainly with the help of respective truth tables and Boolean expressions. The chapter is adequately illustrated with the help of solved examples. Towards the end, the chapter contains application-relevant information in terms of popular type numbers of logic gates from different logic families and their functional description to help application engineers in choosing the right device for their application. Pin connection diagrams are given on the companion website at http://www.wiley.com/go/maini_digital. Different logic families used to hardware-implement different logic functions in the form of digital integrated circuits are discussed in the following chapter.

4.1 Positive and Negative Logic

The binary variables, as we know, can have either of the two states, i.e. the logic '0' state or the logic '1' state. These logic states in digital systems such as computers, for instance, are represented by two different voltage levels or two different current levels. If the more positive of the two voltage or current levels represents a logic '1' and the less positive of the two levels represents a logic '0', then the logic system is referred to as a *positive logic system*. If the more positive of the two voltage or current levels represents a logic '0' and the less positive of the two levels represents a logic '1', then the logic system is referred to as a *negative logic system*. The following examples further illustrate this concept.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

If the two voltage levels are 0 V and +5 V, then in the positive logic system the 0 V represents a logic '0' and the +5 V represents a logic '1'. In the negative logic system, 0 V represents a logic '1' and +5 V represents a logic '0'.

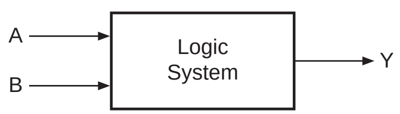
If the two voltage levels are 0 V and −5 V, then in the positive logic system the 0 V represents a logic '1' and the −5 V represents a logic '0'. In the negative logic system, 0 V represents a logic '0' and −5 V represents a logic '1'.

It is interesting to note, as we will discover in the latter part of the chapter, that a positive OR is a negative AND. That is, OR gate hardware in the positive logic system behaves like an AND gate in the negative logic system. The reverse is also true. Similarly, a positive NOR is a negative NAND, and vice versa.

4.2 Truth Table

A truth table lists all possible combinations of input binary variables and the corresponding outputs of a logic system. The logic system output can be found from the logic expression, often referred to as the Boolean expression, that relates the output with the inputs of that very logic system.

When the number of input binary variables is only one, then there are only two possible inputs, i.e. '0' and '1'. If the number of inputs is two, there can be four possible input combinations, i.e. 00, 01, 10 and 11. Figure 4.1(b) shows the truth table of the two-input logic system represented by Fig. 4.1(a). The logic system of Fig. 4.1(a) is such that $Y = 0$ only when both $A = 0$ and $B = 0$. For all other possible input combinations, output $Y = 1$. Similarly, for three input binary variables, the number of possible input combinations becomes eight, i.e. 000, 001, 010, 011, 100, 101, 110 and 111. This statement can be generalized to say that, if a logic circuit has n binary inputs, its truth table will have 2^n possible input combinations, or in other words 2^n rows. Figure 4.2 shows the truth table of a three-input logic circuit, and it has 8 ($= 2^3$) rows. Incidentally, as we will see later in the chapter, this is the truth table of a three-input AND gate. It may be mentioned here that the truth table of a three-input AND gate as given in Fig. 4.2 is drawn following the positive logic system, and also that, in all further discussion throughout the book, we will use a positive logic system unless otherwise specified.



(a)

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

(b)

Figure 4.1 Two-input logic system.

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

Figure 4.2 Truth table of a three-input logic system

4.3 Logic Gates

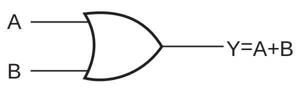
The logic gate is the most basic building block of any digital system, including computers. Each one of the basic logic gates is a piece of hardware or an electronic circuit that can be used to implement some basic logic expression. While laws of Boolean algebra could be used to do manipulation with binary variables and simplify logic expressions, these are actually implemented in a digital system with the help of electronic circuits called logic gates. The three basic logic gates are the OR gate, the AND gate and the NOT gate.

4.3.1 OR Gate

An OR gate performs an ORing operation on two or more than two logic variables. The OR operation on two independent logic variables A and B is written as $Y = A + B$ and reads as Y equals A OR B and not as A plus B. An OR gate is a logic circuit with two or more inputs and one output. The output of an OR gate is LOW only when all of its inputs are LOW. For all other possible input combinations, the output is HIGH. This statement when interpreted for a positive logic system means the following. The output of an OR gate is a logic '0' only when all of its inputs are at logic '0'. For all other possible input combinations, the output is a logic '1'. Figure 4.3 shows the circuit symbol and the truth table of a two-input OR gate. The operation of a two-input OR gate is explained by the logic expression

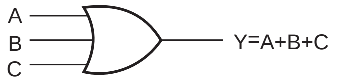
$$Y = A + B \tag{4.1}$$

As an illustration, if we have four logic variables and we want to know the logical output of $(A + B + C + D)$, then it would be the output of a four-input OR gate with A, B, C and D as its inputs.

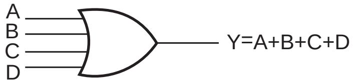


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.3 Two-input OR gate.



(a)



(b)

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

(c)

Figure 4.4 (a) Three-input OR gate, (b) four-input OR gate and (c) the truth table of a three-input OR gate.

Figures 4.4(a) and (b) show the circuit symbol of three-input and four-input OR gates. Figure 4.4(c) shows the truth table of a three-input OR gate. Logic expressions explaining the functioning of three-input and four-input OR gates are $Y = A + B + C$ and $Y = A + B + C + D$.

Example 4.1

How would you hardware-implement a four-input OR gate using two-input OR gates only?

Solution

Figure 4.5(a) shows one possible arrangement of two-input OR gates that simulates a four-input OR gate. A, B, C and D are logic inputs and $Y3$ is the output. Figure 4.5(b) shows another possible arrangement. In the case of Fig. 4.5(a), the output of OR gate 1 is $Y1 = (A + B)$. The second

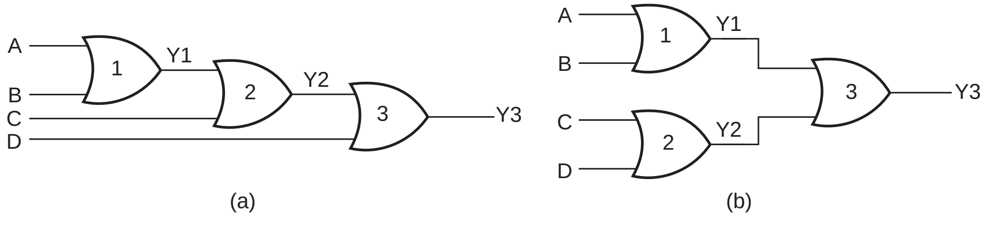


Figure 4.5 Example 4.1.

OR gate produces the output $Y2 = (Y1 + C) = (A + B + C)$. Similarly, the output of OR gate 3 is $Y3 = (Y2 + D) = (A + B + C + D)$. In the case of Fig. 4.5(b), the output of OR gate 1 is $Y1 = (A + B)$. The second OR gate produces the output $Y2 = (C + D)$. Output $Y3$ of the third OR gate is given by $(Y1 + Y2) = (A + B + C + D)$.

Example 4.2

Draw the output waveform for the OR gate and the given pulsed input waveforms of Fig. 4.6(a).

Solution

Figure 4.6(b) shows the output waveform. It can be drawn by following the truth table of the OR gate.

4.3.2 AND Gate

An AND gate is a logic circuit having two or more inputs and one output. The output of an AND gate is HIGH only when all of its inputs are in the HIGH state. In all other cases, the output is LOW. When interpreted for a positive logic system, this means that the output of the AND gate is a logic '1' only when all of its inputs are in logic '1' state. In all other cases, the output is logic '0'. The logic symbol and truth table of a two-input AND gate are shown in Figs 4.7(a) and (b) respectively. Figures 4.8(a) and (b) show the logic symbols of three-input and four-input AND gates respectively. Figure 4.8(c) gives the truth table of a four-input AND gate.

The AND operation on two independent logic variables A and B is written as $Y = A.B$ and reads as Y equals A AND B and not as A multiplied by B. Here, A and B are input logic variables and Y is the output. An AND gate performs an ANDing operation:

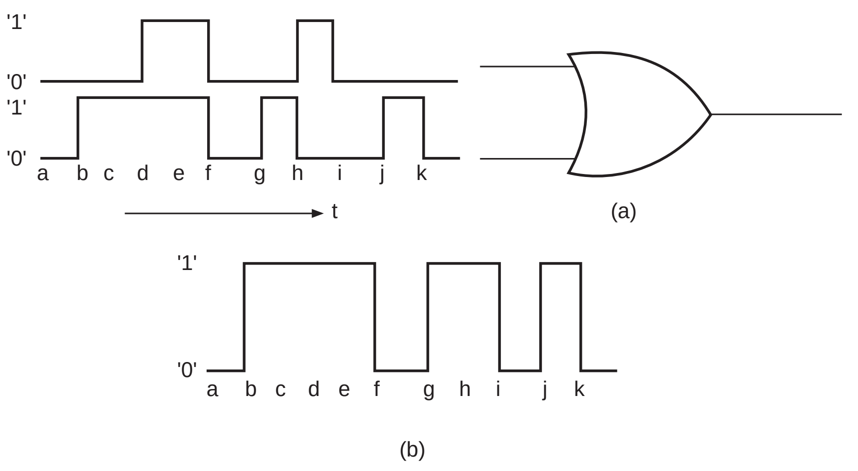
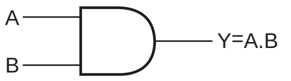


Figure 4.6 Example 4.2.

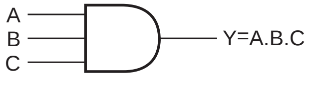


(a)

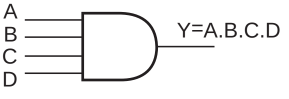
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Figure 4.7 Two-input AND gate.



(a)



(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c)

Figure 4.8 (a) Three-input AND gate, (b) four-input AND gate and (c) the truth table of a four-input AND gate.

- for a two-input AND gate, $Y = A.B$;
- for a three-input AND gate, $Y = A.B.C$;
- for a four-input AND gate, $Y = A.B.C.D$.

If we interpret the basic definition of OR and AND gates for a negative logic system, we have an interesting observation. We find that an OR gate in a positive logic system is an AND gate in a negative logic system. Also, a positive AND is a negative OR.

Example 4.3

Show the logic arrangement for implementing a four-input AND gate using two-input AND gates only.

Solution

Figure 4.9 shows the hardware implementation of a four-input AND gate using two-input AND gates. The output of AND gate 1 is $Y1 = A.B$. The second AND gate produces an output $Y2$ given by $Y2 = Y1.C = A.B.C$. Similarly, the output of AND gate 3 is $Y = Y2.D = A.B.C.D$ and hence the result.

4.3.3 NOT Gate

A NOT gate is a one-input, one-output logic circuit whose output is always the complement of the input. That is, a LOW input produces a HIGH output, and vice versa. When interpreted for a positive logic system, a logic '0' at the input produces a logic '1' at the output, and vice versa. It is also known as a 'complementing circuit' or an 'inverting circuit'. Figure 4.10 shows the circuit symbol and the truth table.

The NOT operation on a logic variable X is denoted as $\overline{X}$ or X'. That is, if X is the input to a NOT circuit, then its output Y is given by $Y = \overline{X}$ or X' and reads as Y equals NOT X. Thus, if $X = 0$, $Y = 1$ and if $X = 1$, $Y = 0$.

Example 4.4

For the logic circuit arrangements of Figs 4.11(a) and (b), draw the output waveform.

Solution

In the case of the OR gate arrangement of Fig. 4.11(a), the output will be permanently in logic '1' state as the two inputs can never be in logic '0' state together owing to the presence of the inverter. In the case of the AND gate arrangement of Fig. 4.11(b), the output will be permanently in logic '0' state as the two inputs can never be in logic '1' state together owing to the presence of the inverter.

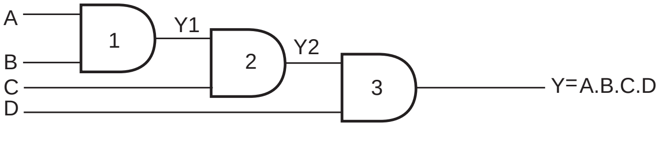
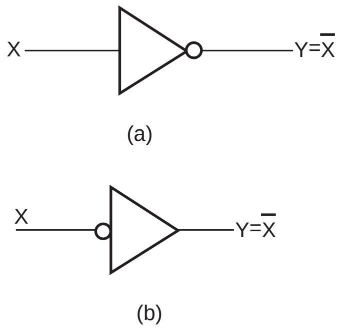


Figure 4.9 Implementation of a four-input AND gate using two-input AND gates.



X	Y
0	1
1	0

Figure 4.10 (a) Circuit symbol of a NOT circuit and (b) the truth table of a NOT circuit.

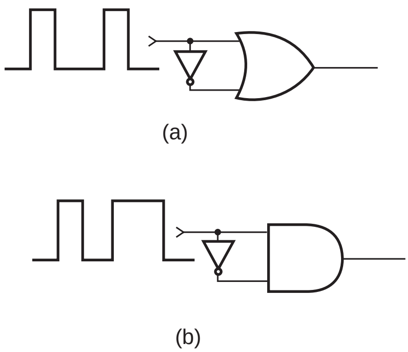
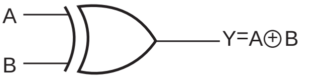


Figure 4.11 Example 4.4.

4.3.4 EXCLUSIVE-OR Gate

The EXCLUSIVE-OR gate, commonly written as EX-OR gate, is a two-input, one-output gate. Figures 4.12(a) and (b) respectively show the logic symbol and truth table of a two-input EX-OR gate. As can be seen from the truth table, the output of an EX-OR gate is a logic '1' when the inputs are unlike and a logic '0' when the inputs are like. Although EX-OR gates are available in integrated circuit form only as two-input gates, unlike other gates which are available in multiple inputs also, multiple-input EX-OR logic functions can be implemented using more than one two-input gates. The truth table of a multiple-input EX-OR function can be expressed as follows. The output of a multiple-input EX-OR logic function is a logic '1' when the number of 1s in the input sequence is odd and a logic '0' when the number of 1s in the input sequence is even, including zero. That is, an all 0s input sequence also produces a logic '0' at the output. Figure 4.12(c) shows the truth table of a four-input EX-OR function. The output of a two-input EX-OR gate is expressed by

$$Y = (A \oplus B) = \overline{A}B + A\overline{B} \tag{4.2}$$



(a)

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

(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(c)

Figure 4.12 (a) Circuit symbol of a two-input EXCLUSIVE-OR gate, (b) the truth table of a two-input EXCLUSIVE-OR gate and (c) the truth table of a four-input EXCLUSIVE-OR gate

Example 4.5

How do you implement three-input and four-input EX-OR logic functions with the help of two-input EX-OR gates?

Solution

Figures 4.13(a) and (b) show the implementation of a three-input EX-OR logic function and a four-input EX-OR logic function using two-input logic gates:

- For Fig. 4.13(a), the output $Y1$ is given by $A \oplus B$. The final output Y is given by $Y = (Y1 \oplus C) = (A \oplus B) \oplus C = A \oplus B \oplus C$.
- Figure 4.13(b) can be explained on similar lines.

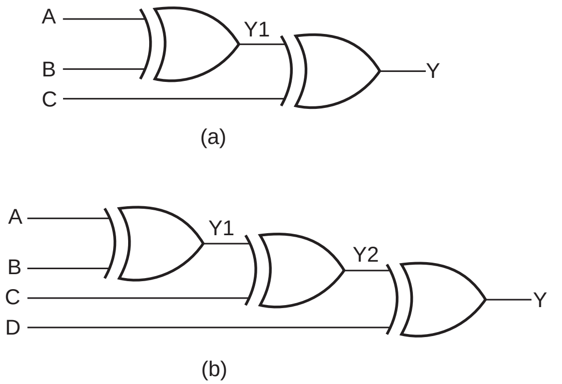


Figure 4.13 (a) Three-input EX-OR gate and (b) a four-input EX-OR gate.

Example 4.6

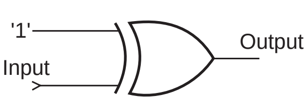
How can you implement a NOT circuit using a two-input EX-OR gate?

Solution

Refer to the truth table of a two-input EX-OR gate reproduced in Fig. 4.14(a). It is clear from the truth table that, if one of the inputs of the gate is permanently tied to logic '1' level, then the other input and output perform the function of a NOT circuit. Figure 4.14(b) shows the implementation.

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

(a)



(b)

Figure 4.14 Implementation of a NOT circuit using an EX-OR gate.

4.3.5 NAND Gate

NAND stands for NOT AND. An AND gate followed by a NOT circuit makes it a NAND gate [Fig. 4.15(a)]. Figure 4.15(b) shows the circuit symbol of a two-input NAND gate. The truth table of a NAND gate is obtained from the truth table of an AND gate by complementing the output entries [Fig. 4.15(c)]. The output of a NAND gate is a logic '0' when all its inputs are a logic '1'. For all other input combinations, the output is a logic '1'. NAND gate operation is logically expressed as

$$Y = \overline{A.B} \tag{4.3}$$

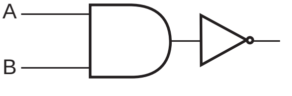
In general, the Boolean expression for a NAND gate with more than two inputs can be written as

$$Y = \overline{(A.B.C.D...)} \tag{4.4}$$

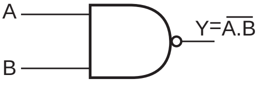
4.3.6 NOR Gate

NOR stands for NOT OR. An OR gate followed by a NOT circuit makes it a NOR gate [Fig. 4.16(a)]. The truth table of a NOR gate is obtained from the truth table of an OR gate by complementing the output entries. The output of a NOR gate is a logic '1' when all its inputs are logic '0'. For all other input combinations, the output is a logic '0'. The output of a two-input NOR gate is logically expressed as

$$Y = \overline{(A+B)} \tag{4.5}$$



(a)



(b)

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

(c)

Figure 4.15 (a) Two-input NAND implementation using an AND gate and a NOT circuit, (b) the circuit symbol of a two-input NAND gate and (c) the truth table of a two-input NAND gate.

(a) (b)

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

(c)

Figure 4.16 (a) Two-input NOR implementation using an OR gate and a NOT circuit, (b) the circuit symbol of a two-input NOR gate and (c) the truth table of a two-input NOR gate.

In general, the Boolean expression for a NOR gate with more than two inputs can be written as

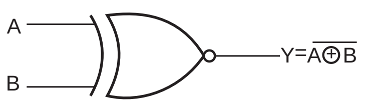
$$Y = \overline{(A + B + C + D...)} \tag{4.6}$$

4.3.7 *EXCLUSIVE-NOR Gate*

EXCLUSIVE-NOR (commonly written as EX-NOR) means NOT of EX-OR, i.e. the logic gate that we get by complementing the output of an EX-OR gate. Figure 4.17 shows its circuit symbol along with its truth table.

The truth table of an EX-NOR gate is obtained from the truth table of an EX-OR gate by complementing the output entries. Logically,

$$Y = (\overline{A \oplus B}) = (A.B + \overline{A}.\overline{B}) \tag{4.7}$$



(a)

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

(b)

Figure 4.17 (a) Circuit symbol of a two-input EXCLUSIVE-NOR gate and (b) the truth table of a two-input EXCLUSIVE-NOR gate.

The output of a two-input EX-NOR gate is a logic '1' when the inputs are like and a logic '0' when they are unlike. In general, the output of a multiple-input EX-NOR logic function is a logic '0' when the number of 1s in the input sequence is odd and a logic '1' when the number of 1s in the input sequence is even including zero. That is, an all 0s input sequence also produces a logic '1' at the output.

Example 4.7

Show the logic arrangements for implementing:

(a) a four-input NAND gate using two-input AND gates and NOT gates;
(b) a three-input NAND gate using two-input NAND gates;
(c) a NOT circuit using a two-input NAND gate;
(d) a NOT circuit using a two-input NOR gate;
(e) a NOT circuit using a two-input EX-NOR gate.

Solution

(a) Figure 4.18(a) shows the arrangement. The logic diagram is self-explanatory. The first step is to get a four-input AND gate using two-input AND gates. The output thus obtained is then complemented using a NOT circuit as shown.
(b) Figure 4.18(b) shows the arrangement, which is again self-explanatory. The first step is to get a two-input AND from a two-input NAND. The output of the two-input AND gate and the third input then feed the inputs of another two-input NAND to get the desired output.
(c) Shorting the inputs of the NAND gives a one-input, one-output NOT circuit. This is because when all inputs to a NAND are at logic '0' level the output is a logic '1', and when all inputs to a NAND are at logic '1' level the output is a logic '0'. Figure 4.18(c) shows the implementation.
(d) Again, shorting the inputs of a NOR gate gives a NOT circuit. From the truth table of a NOR gate it is evident that an all 0s input to a NOR gate gives a logic '1' output and an all 1s input gives a logic '0' output. Figure 4.18(d) shows the implementation.
(e) It is evident from the truth table of a two-input EX-NOR gate that, if one of the inputs is permanently tied to a logic '0' level and the other input is treated as the input, then it behaves as a NOT circuit between input and output [Fig. 4.18(e)]. When the input is a logic '0', the two inputs become 00, which produces a logic '1' at the output. When the input is at logic '1' level, a 01 input produces a logic '0' at the output.

Example 4.8

How do you implement a three-input EX-NOR function using only two-input EX-NOR gates?

Solution

Figure 4.19 shows the arrangement. The first two EX-NOR gates implement a two-input EX-OR gate using two-input EX-NOR gates. The second EX-NOR gate here has been wired as a NOT circuit. The output of the second gate and the third input are fed to the two inputs of the third EX-NOR gate.

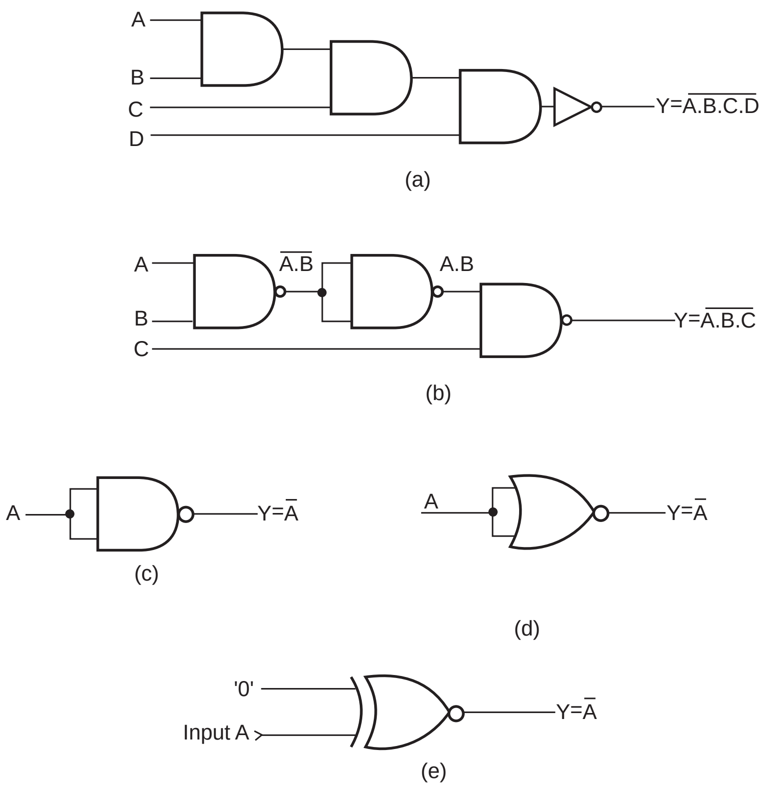


Figure 4.18 Example 4.7.

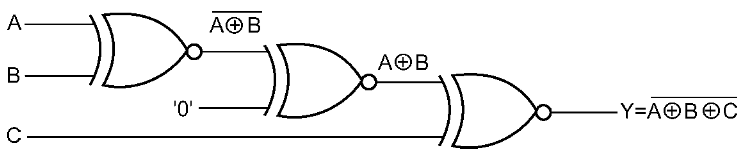


Figure 4.19 Example 4.8.

4.3.8 INHIBIT Gate

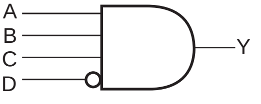
There are many situations in digital circuit design where the passage of a logic signal needs to be either enabled or inhibited depending upon certain other control inputs. INHIBIT here means that the gate produces a certain fixed logic level at the output irrespective of changes in the input logic level. As an illustration, if one of the inputs of a four-input NOR gate is permanently tied to logic '1' level, then the output will always be at logic '0' level irrespective of the logic status of other inputs. This gate will behave as a NOR gate only when this control input is at logic '0' level. This is an example of the INHIBIT function. The INHIBIT function is available in integrated circuit form for an AND gate, which is basically an AND gate with one of its inputs negated by an inverter. The negated input acts to inhibit the gate. In other words, the gate will behave like an AND gate only when the negated input is driven to a logic '0'. Figure 4.20 shows the circuit symbol and truth table of a four-input INHIBIT gate.

Example 4.9

Refer to the INHIBIT gate of Fig. 4.21(a). If the waveform of Fig. 4.21(b) is applied to the INHIBIT input, draw the waveform at the output.

Solution

Since all other inputs of the gate have been permanently tied to logic '1' level, a logic '0' at the INHIBIT input would produce a logic '1' at the output and a logic '1' at the INHIBIT input would produce a logic '0' at the output. The output waveform is therefore the inversion of the input waveform and is shown in Fig. 4.22.



(a)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(b)

Figure 4.20 INHIBIT gate.

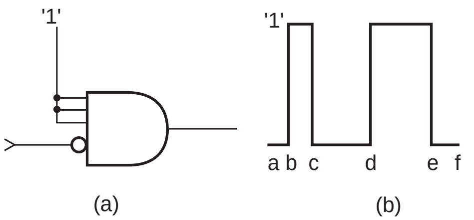


Figure 4.21 Example 4.9.

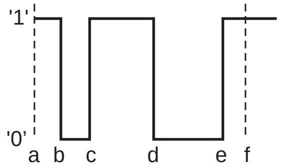


Figure 4.22 Solution to example 4.9.

Example 4.10

Refer to the INHIBIT gate shown in Fig. 4.23(a) and the INHIBIT input waveform shown in Fig. 4.23(b). Sketch the output waveform.

Solution

The output will always be at logic '1' level as two of the inputs of the logic gate, which is a NAND, are permanently tied to logic '0' level. This would have been so even if one of the inputs of the gate were at logic '0' level.

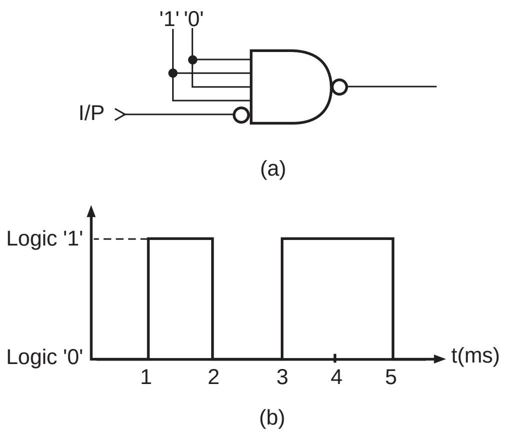


Figure 4.23 Example 4.10.

4.4 Universal Gates

OR, AND and NOT gates are the three basic logic gates as they together can be used to construct the logic circuit for any given Boolean expression. NOR and NAND gates have the property that they individually can be used to hardware-implement a logic circuit corresponding to any given Boolean expression. That is, it is possible to use either only NAND gates or only NOR gates to implement any Boolean expression. This is so because a combination of NAND gates or a combination of NOR gates can be used to perform functions of any of the basic logic gates. It is for this reason that NAND and NOR gates are universal gates.

As an illustration, Fig. 4.24 shows how two-input NAND gates can be used to construct a NOT circuit [Fig. 4.24(a)], a two-input AND gate [Fig. 4.24(b)] and a two-input OR gate [Fig. 4.24(c)]. Figure 4.25 shows the same using NOR gates. Understanding the conversion of NAND to OR and NOR to AND requires the use of DeMorgan's theorem, which is discussed in Chapter 6 on Boolean algebra.

4.5 Gates with Open Collector/Drain Outputs

These are gates where we need to connect an external resistor, called the pull-up resistor, between the output and the DC power supply to make the logic gate perform the intended logic function. Depending on the logic family used to construct the logic gate, they are referred to as gates with open collector output (in the case of the TTL logic family) or open drain output (in the case of the MOS logic family). Logic families are discussed in detail in Chapter 5.

The advantage of using open collector/open drain gates lies in their capability of providing an ANDing operation when outputs of several gates are tied together through a common pull-up resistor,

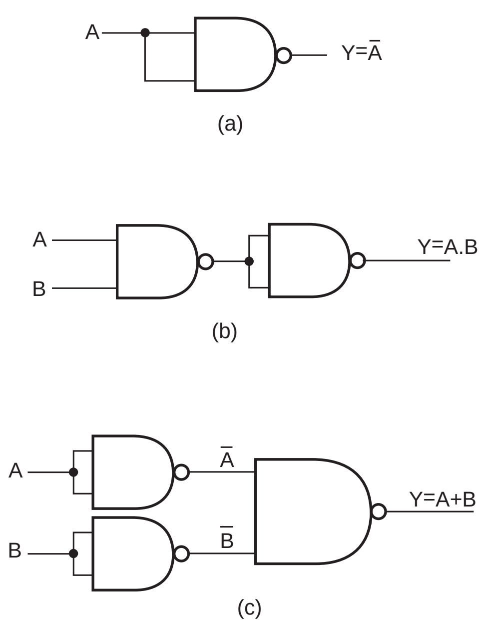


Figure 4.24 Implementation of basic logic gates using only NAND gates.

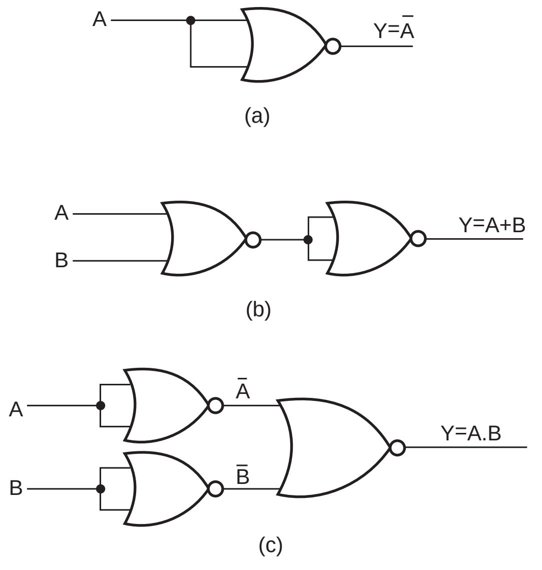


Figure 4.25 Implementation of basic logic gates using only NOR gates.

without having to use an AND gate for the purpose. This connection is also referred to as WIRE-AND connection. Figure 4.26(a) shows such a connection for open collector NAND gates. The output in this case would be

$$Y = \overline{AB}.\overline{CD}.\overline{EF} \tag{4.8}$$

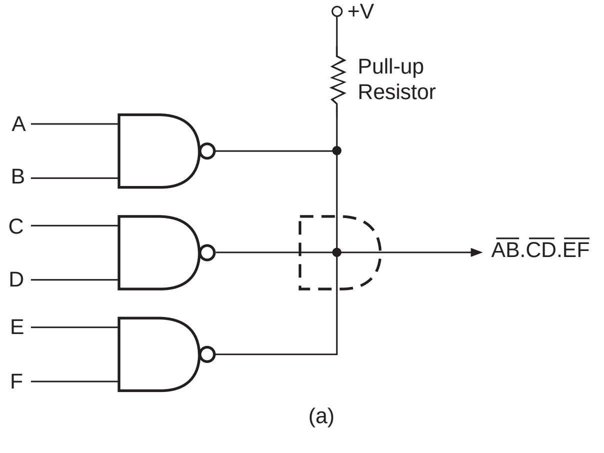


Figure 4.26 WIRE-AND connection with open collector/drain devices.

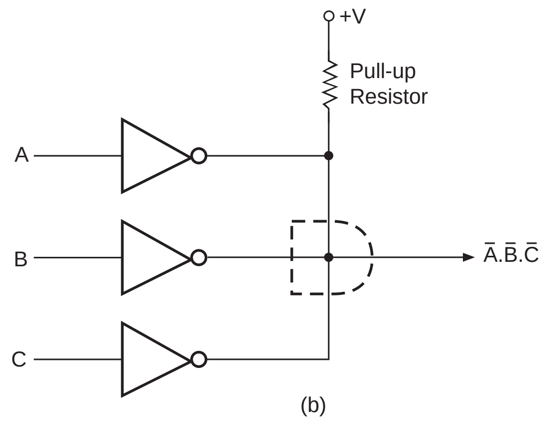


Figure 4.26 (*continued*).

Figure 4.26(b) shows a similar arrangement for NOT gates. The disadvantage is that they are relatively slower and noisier. Open collector/drain devices are therefore not recommended for applications where speed is an important consideration.

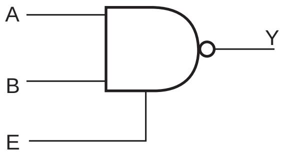
4.6 Tristate Logic Gates

Tristate logic gates have three possible output states, i.e. the logic '1' state, the logic '0' state and a high-impedance state. The high-impedance state is controlled by an external ENABLE input. The ENABLE input decides whether the gate is active or in the high-impedance state. When active, it can be '0' or '1' depending upon input conditions. One of the main advantages of these gates is that their inputs and outputs can be connected in parallel to a common bus line. Figure 4.27(a) shows the circuit symbol of a tristate NAND gate with active HIGH ENABLE input, along with its truth table. The one shown in Fig. 4.27(b) has active LOW ENABLE input. When tristate devices are paralleled, only one of them is enabled at a time. Figure 4.28 shows paralleling of tristate inverters having active HIGH ENABLE inputs.

4.7 AND-OR-INVERT Gates

AND-OR and OR-AND gates can be usefully employed to implement sum-of-products and product-of-sums Boolean expressions respectively. Figures 4.29(a) and (b) respectively show the symbols of AND-OR-INVERT and OR-AND-INVERT gates.

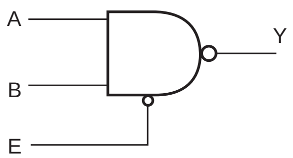
Another method for designating the gates shown in Fig. 4.29 is to call them two-wide, two-input AND-OR-INVERT or OR-AND-INVERT gates as the case may be. The gate is two-wide as there are two gates at the input, and two-input as each of the gates has two inputs. Other varieties such as two-wide, four-input AND-OR-INVERT (Fig. 4.30) and four-wide, two-input AND-OR-INVERT (Fig. 4.31) are also available in IC form.



A	B	E	Y
0	0	0	Z
0	0	1	1
0	1	0	Z
0	1	1	1
1	0	0	Z
1	0	1	1
1	1	0	Z
1	1	1	0

Z= High Impedance state

(a)



A	B	E	Y
0	0	0	1
0	0	1	Z
0	1	0	1
0	1	1	Z
1	0	0	1
1	0	1	Z
1	1	0	0
1	1	1	Z

Z= High Impedance state

(b)

Figure 4.27 Tristate devices.

4.8 Schmitt Gates

The logic gates discussed so far have a single-input threshold voltage level. This threshold is the same for both LOW-to-HIGH and HIGH-to-LOW output transitions. This threshold voltage lies somewhere between the highest LOW voltage level and the lowest HIGH voltage level guaranteed by the manufacturer of the device. These logic gates can produce an erratic output when fed with a slow

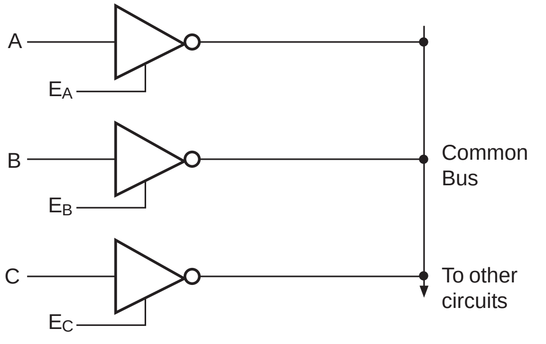


Figure 4.28 Paralleling of tristate inverters.

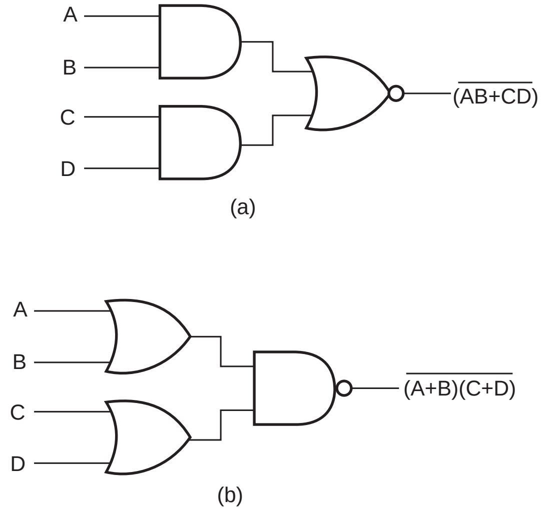


Figure 4.29 AND-OR-INVERT and OR-AND-INVERT gates.

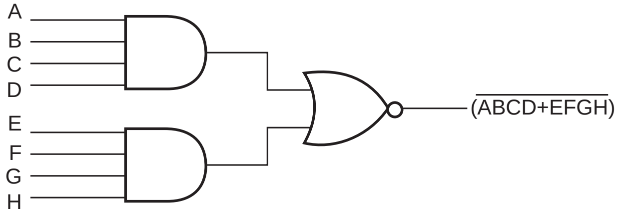


Figure 4.30 Two-wide, four-input AND-OR-INVERT gate.

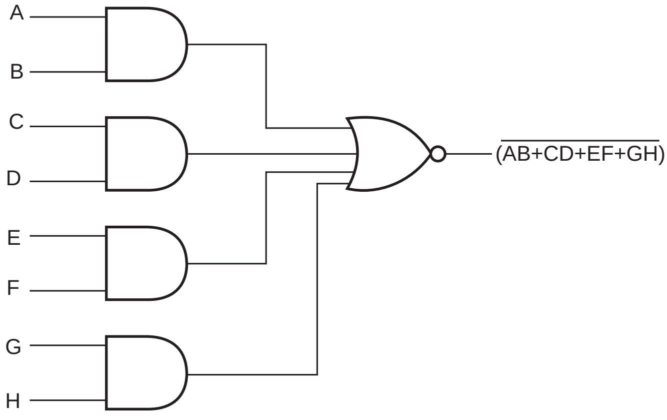


Figure 4.31 Four-wide, two-input AND-OR-INVERT gate.

varying input. Figure 4.32 shows the response of an inverter circuit when fed with a slow varying input both in the case of an ideal signal [Fig. 4.32(a)] and in the case of a practical signal having a small amount of AC noise superimposed on it [Fig. 4.32(b)]. A possible solution to this problem lies in having two different threshold voltage levels, one for LOW-to-HIGH transition and the other for HIGH-to-LOW transition, by introducing some positive feedback in the internal gate circuitry, a phenomenon called hysteresis.

There are some logic gate varieties, mainly in NAND gates and inverters, that are available with built-in hysteresis. These are called Schmitt gates, which interpret varying input voltages according to two threshold voltages, one for LOW-to-HIGH and the other for HIGH-to-LOW output transition. Figures 4.33(a) and (b) respectively show circuit symbols of Schmitt NAND and Schmitt inverter. Schmitt gates are distinguished from conventional gates by the small 'hysteresis' symbol reminiscent of the $B-H$ loop for a ferromagnetic material. Figure 4.33(c) shows typical transfer characteristics for such a device. The difference between the two threshold levels is

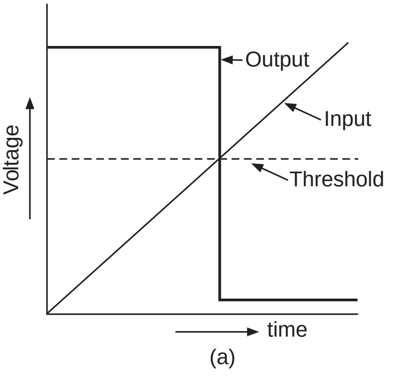
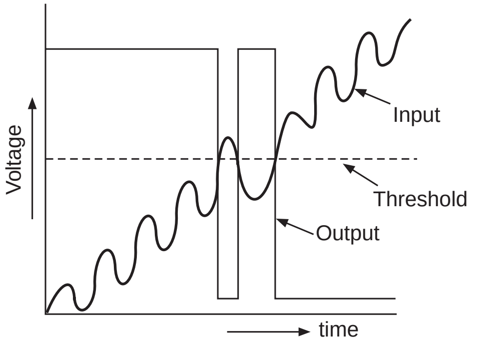


Figure 4.32 Response of conventional inverters to slow varying input.



(b)

Figure 4.32 (*continued*).

the hysteresis. These characteristics have been reproduced from the data sheet of IC 74LS132, which is a quad two-input Schmitt NAND belonging to the low-power Schottky TTL family. Figure 4.33(d) shows the response of a Schmitt inverter to a slow varying noisy input signal. We will learn more about different logic families in Chapter 5. It may be mentioned here that hysteresis increases noise immunity and is used in applications where noise is expected on input signal lines.

4.9 Special Output Gates

There are many applications where it is desirable to have both noninverted and inverted outputs. Examples include a single-input gate that is both an inverter and a noninverting buffer, or a two-input logic gate that is both an AND and a NAND. Such gates are called complementary output gates and are particularly useful in circuits where PCB space is at a premium. These are also useful in circuits where the addition of an inverter to obtain the inverted output introduces an undesirable time delay between inverted and noninverted outputs. Figure 4.34 shows the circuit symbols of complementary buffer, AND, OR and EX-OR gates.

Example 4.11

Draw the circuit symbols for (a) a two-wide, four-input OR-AND-INVERT gate and (b) a four-wide, two-input OR-AND-INVERT gate.

Solution

(a) Refer to Fig. 4.35(a).
(b) Refer to Fig. 4.35(b).

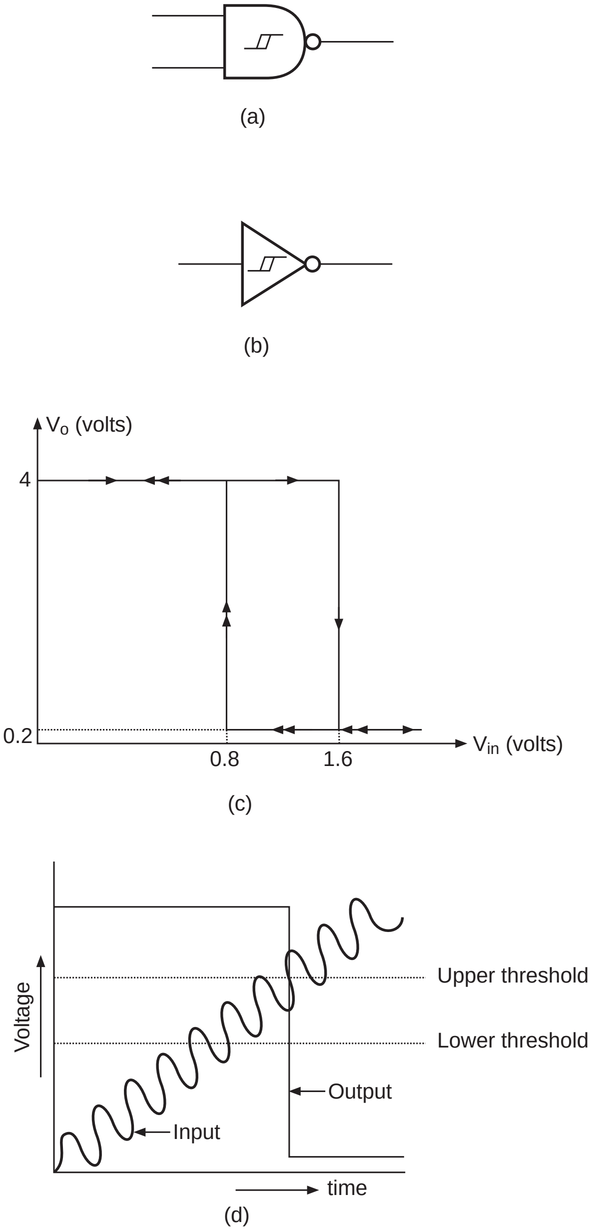


Figure 4.33 Schmitt gates.

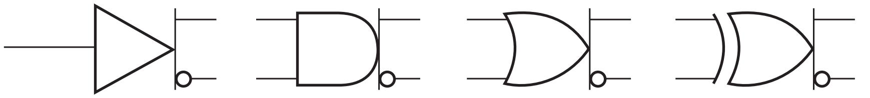
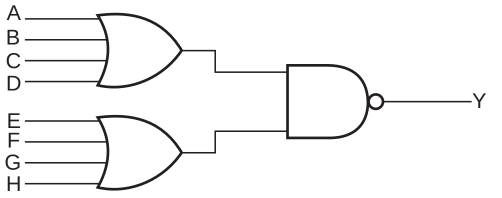
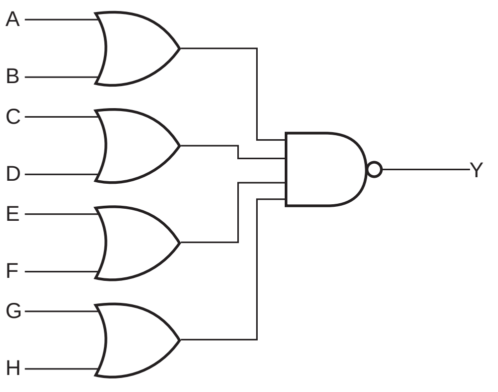


Figure 4.34 Complementary gates.



(a)



(b)

Figure 4.35 Example 4.11.

Example 4.12

Refer to Fig. 4.36(a). If the NAND gate used has the transfer characteristics of Fig. 4.36(b), sketch the expected output waveform.

Solution

The output waveform is shown in Fig. 4.36(c). The output is initially in logic '1' state. It goes from logic '1' to logic '0' state as the input exceeds 2 V. The output goes from logic '0' to logic '1' state as the input drops below 1 V.

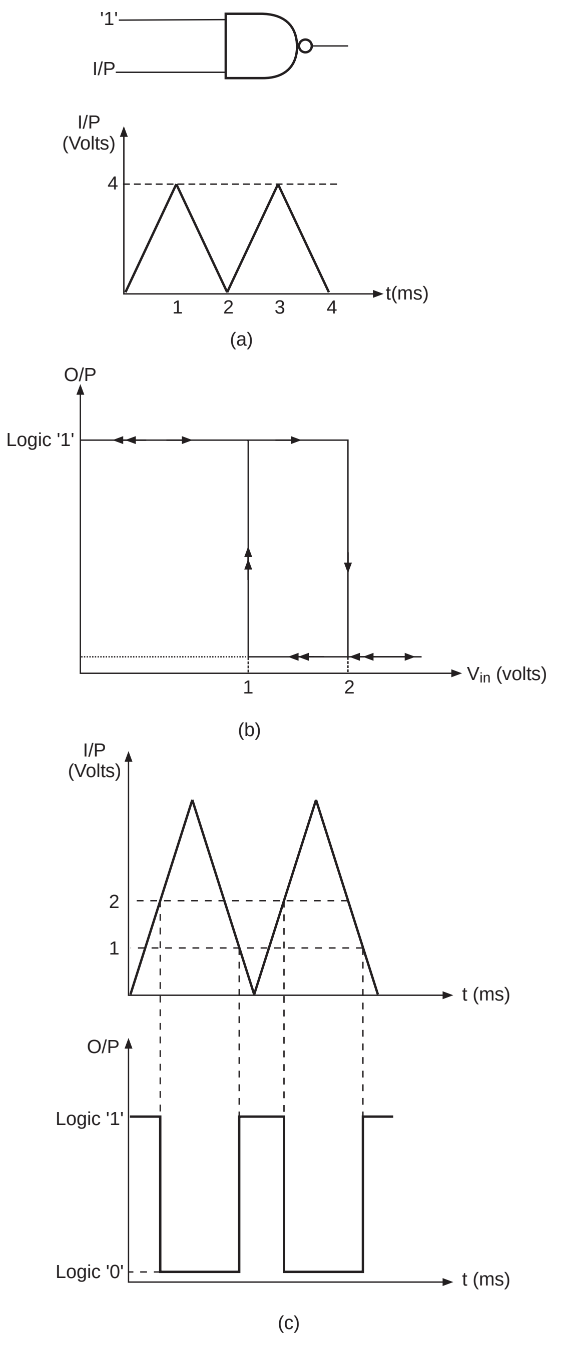


Figure 4.36 Example 4.12.

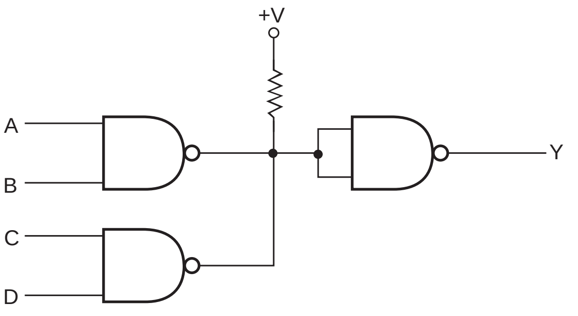


Figure 4.37 Example 4.13.

Example 4.13

Refer to the logic arrangement of Fig. 4.37. Write the logic expression for the output Y.

Solution

The NAND gates used in the circuit are open collector gates. Paralleling of the two NAND gates at the input leads to a WIRE-AND connection. Therefore the logic expression at the point where the two outputs combine is given by the equation

$$(\overline{AB}.\overline{CD}) \tag{4.9}$$

Using DeMorgan's theorem (discussed in Chapter 6 on Boolean algebra),

$$(\overline{AB}.\overline{CD}) = (\overline{AB + CD}) \tag{4.10}$$

The third NAND is wired as an inverter. Therefore, the final output can be written as

$$Y = (AB + CD) \tag{4.11}$$

4.10 Fan-Out of Logic Gates

It is a common occurrence in logic circuits that the output of one logic gate feeds the inputs of several others. It is not practical to drive the inputs of an unlimited number of logic gates from the output of a single logic gate. This is limited by the current-sourcing capability of the output when the output of the logic gate is HIGH and by the current-sinking capability of the output when it is LOW, and also by the requirement of the inputs of the logic gates being fed in the two states.

To illustrate the point further, let us say that the current-sourcing capability of a certain NAND gate is I_{OH} when its output is in the logic HIGH state and that each of the inputs of the logic gate that it is driving requires an input current I_{IH}, as shown in Fig. 4.38(a). In this case, the output of the logic gate will be able to drive a maximum of I_{OH}/I_{IH} inputs when it is in the logic HIGH state. When the output of the driving logic gate is in the logic LOW state, let us say that it has a maximum current-sinking capability I_{OL}, and that each of the inputs of the driven logic gates requires a sinking current I_{IL}, as shown in Fig. 4.38(b). In this case the output of the logic gate will be able to drive a maximum of

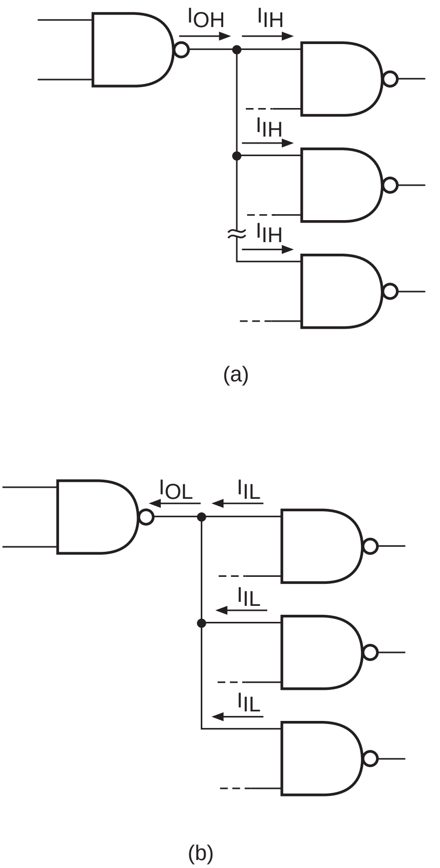


Figure 4.38 Fan-out of logic gates.

I_{OL}/I_{IL} inputs when it is in the logic LOW state. Thus, the number of logic gate inputs that can be driven from the output of a single logic gate will be I_{OH}/I_{IH} in the logic HIGH state and I_{OL}/I_{IL} in the logic LOW state. The number of logic gate inputs that can be driven from the output of a single logic gate without causing any false output is called fan-out. It is the characteristic of the logic family to which the device belongs. If in a certain case the two values I_{OH}/I_{IH} and I_{OL}/I_{IL} are different, the fan-out is taken as the smaller of the two. Figure 4.39 shows the actual circuit diagram where the output of a single NAND gate belonging to a standard TTL logic family feeds the inputs of multiple NAND gates belonging to the same family when the output of the feeding gate is in the logic HIGH state [Fig. 4.39(a)] and the logic LOW state [Fig. 4.39(b)]. We will learn in Chapter 5 on logic families that the maximum HIGH-state output sourcing current $(I_{OH})_{max}$ and maximum HIGH-state input current $(I_{IH})_{max}$ specifications of standard TTL family devices are 400 μA and 40 μA respectively. Also, the maximum LOW-state output sinking current $(I_{OL})_{max}$ and maximum LOW-state input current $(I_{IL})_{max}$ specifications are 16 mA and 1.6 mA respectively. Considering both the sourcing and sinking capability of standard TTL family devices, we obtain a fan-out figure of 10 both for HIGH and for LOW logic states. If the maximum sourcing and sinking current specifications are exceeded, the output voltage levels in the logic HIGH and LOW states will go out of the specified ranges.

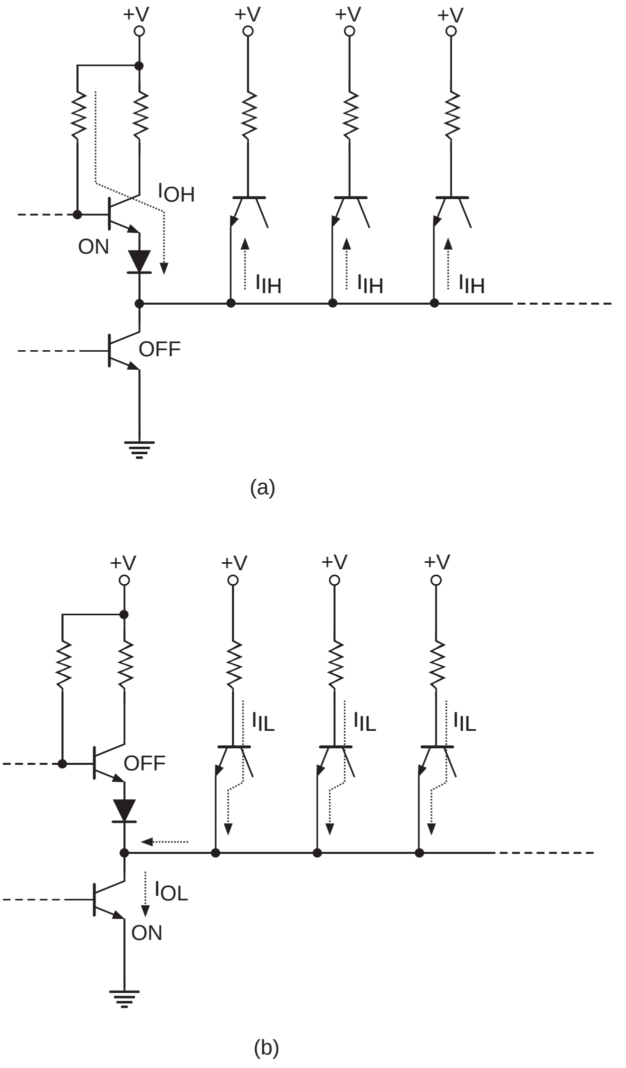


Figure 4.39 Fan-out of the standard TTL logic family.

Example 4.14

A certain logic family has the following input and output current specifications:

1. *The maximum output HIGH-state current = 1 mA.*
2. *The maximum output LOW-state current = 20 mA.*
3. *The maximum input HIGH-state current = 50 μA.*
4. *The maximum input LOW-state current = 2 mA.*

The output of an inverter belonging to this family feeds the clock inputs of various flip-flops belonging to the same family. How many flip-flops can be driven by the output of this inverter providing the clock signal? Incidentally, the data given above are taken from the data sheet of a Schottky TTL family.

Solution

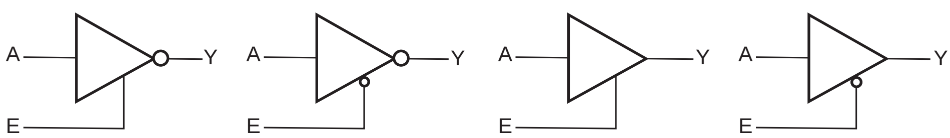
- The HIGH-state fan-out = (1/0.05) = 20 and the LOW-state fan-out = (20/2) = 10.
- Since the lower of the two fan-out values is 10, the inverter output can drive a maximum of 10 flip-flops.

4.11 Buffers and Transceivers

Logic gates, discussed in the previous pages, have a limited load-driving capability. A buffer has a larger load-driving capability than a logic gate. It could be an inverting or noninverting buffer with a single input, a NAND buffer, a NOR buffer, an OR buffer or an AND buffer. ‘Driver’ is another name for a buffer. A driver is sometimes used to designate a circuit that has even larger drive capability than a buffer. Buffers are usually tristate devices to facilitate their use in bus-oriented systems. Figure 4.40 shows the symbols and functional tables of inverting and noninverting buffers of the tristate type.

A transceiver is a bidirectional buffer with additional direction control and ENABLE inputs. It allows flow of data in both directions, depending upon the logic status of the control inputs. Transceivers, like buffers, are tristate devices to make them compatible with bus-oriented systems. Figures 4.41(a) and (b) respectively show the circuit symbols of inverting and noninverting transceivers. Figure 4.42 shows a typical logic circuit arrangement of a tristate noninverting transceiver with its functional table [Fig. 4.42(b)].

Some of the common applications of inverting and noninverting buffers are as follows. Buffers are used to drive circuits that need more drive current. Noninverting buffers are also used to increase the fan-out of a given logic gate. This means that the buffer can be used to increase the number of logic gate inputs to which the output of a given logic gate can be connected. Yet another application of a noninverting buffer is its use as a delay line. It delays the signal by an amount equal to the propagation delay of the device. More than one device can be connected in cascade to get larger delays.



A	E	Y
X	0	Z
0	1	1
1	1	0

A	E	Y
X	1	Z
0	0	1
1	0	0

A	E	Y
X	0	Z
0	1	0
1	1	1

A	E	Y
X	1	Z
0	0	0
1	0	1

Z = High Impedance State

Figure 4.40 (a) Inverting tristate buffers and (b) noninverting tristate buffers.

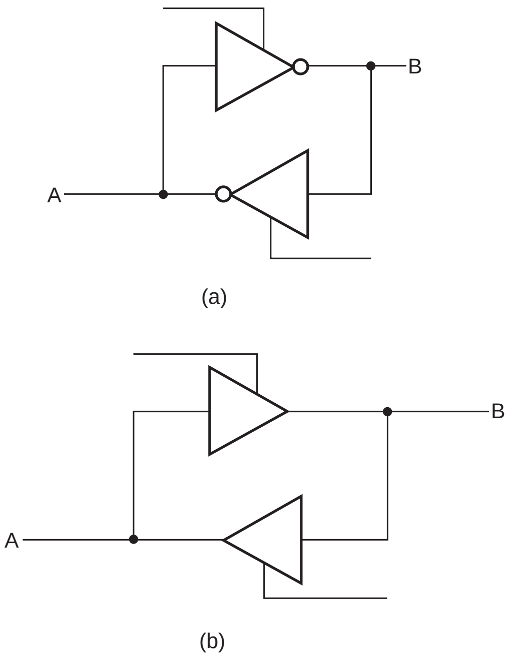


Figure 4.41 (a) Inverting transceivers and (b) noninverting transceivers.

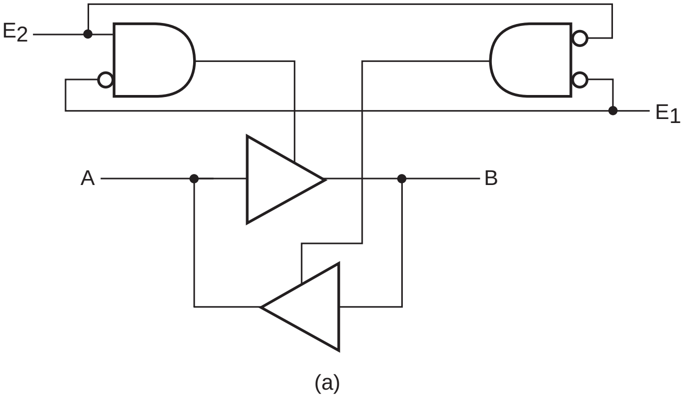


Figure 4.42 Tristate noninverting transceiver.

E_1	E_2	Operation
L	L	Data flow from B to A
L	H	Data flow from A to B
H	X	Isolation

(b)

Figure 4.42 (*continued*).

4.12 IEEE/ANSI Standard Symbols

The symbols used thus far in the chapter for representing different types of gate are the ones that are better known to all of us and have been in use for many years. Each logic gate has a symbol with a distinct shape. However, for more complex logic devices, e.g. sequential logic devices like flip-flops, counters, registers or arithmetic circuits, such as adders, subtractors, etc., these symbols do not carry any useful information. A new set of standard symbols was introduced in 1984 under IEEE/ANSI Standard 91–1984. The logic symbols given under this standard are being increasingly used now and have even started appearing in the literature published by manufacturers of digital integrated circuits. The utility of this new standard will be more evident in the following paragraphs as we go through its salient features and illustrate them with practical examples.

4.12.1 IEEE/ANSI Standards – Salient Features

This standard uses a rectangular symbol for all devices instead of a different symbol shape for each device. For instance, all logic gates (OR, AND, NAND, NOR) will be represented by a rectangular block.

A right triangle is used instead of a bubble to indicate inversion of a logic level. Also, the right triangle is used to indicate whether a given input or output is active LOW. The absence of a triangle indicates an active HIGH input or output. As far as logic gates are concerned, a special notation inside the rectangular block describes the logic relationship between output and inputs. A ‘1’ inside the block indicates that the device has only one input. An AND operation is expressed by ‘&’, and an OR operation is expressed by the symbol ‘$\geq$1’. Figure 4.43 shows the ANSI counterparts of various logic gates. A ‘$\geq$1’ symbol indicates that the output is HIGH when one or more than one input is HIGH. An ‘&’ symbol indicates that the output is HIGH only when all the inputs are HIGH. The two-input EX-OR is represented by the symbol ‘=1’ which implies that the output is HIGH only when one of its inputs is HIGH.

A special dependency notation system is used to indicate how the outputs depend upon the input. This notation contains almost the entire functional information of the logic device in question. This will be more clear as we illustrate this new standard with the help of ANSI symbols for some of the actual devices belonging to the category of flip-flops, counters, etc., in the following chapters. All those control inputs that control the timing of change in output states as per logic status of inputs are designated by the letter ‘C’. These are ENABLE inputs in latches or CLOCK inputs in flip-flops.

Most of the digital ICs contain more than one similar function on one chip such as IC 7400 (quad two-input NAND), IC 7404 (hex inverter), IC 74112 (dual-edge triggered JK flip-flop), IC 7474 (dual D-type latch), IC 7475 (quad D-type latch) and so on. Those inputs to such ICs that are common to

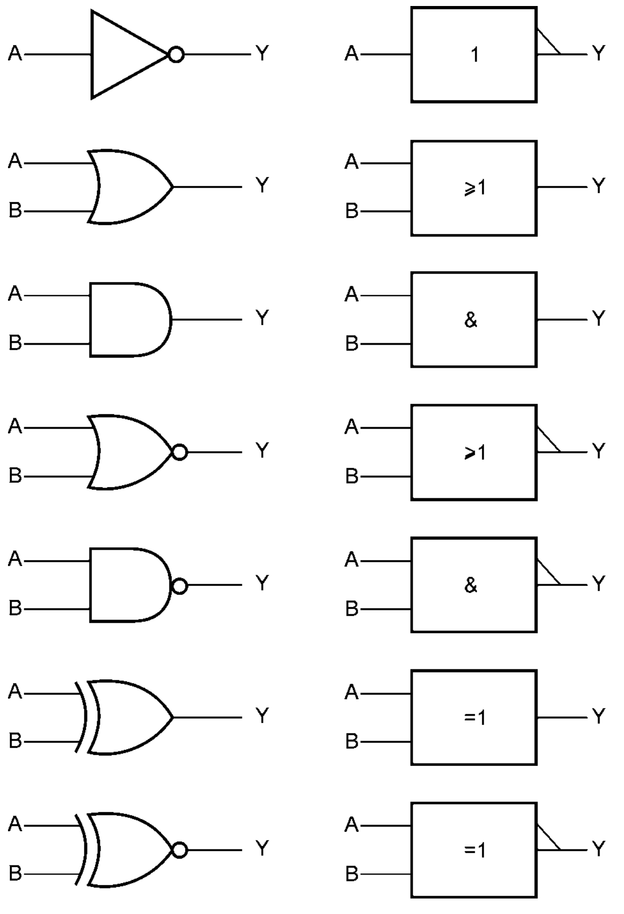


Figure 4.43 IEEE / ANSI symbols.

all the functional blocks or in other words similarly affect various individual but similar functions are represented by a separate notched rectangle on the top of the main rectangle.

4.12.2 ANSI Symbols for Logic Gate ICs

Figure 4.44 shows the ANSI symbol for IC 7400, which is a quad two-input NAND gate. The figure is self-explanatory with the background given in the preceding paragraphs. Any other similar device, i.e. another quad two-input NAND gate belonging to another logic family, would also be represented by the same ANSI symbol. As another illustration, Fig. 4.45 shows the ANSI symbol for IC 7420, which is a dual four-input NAND gate.

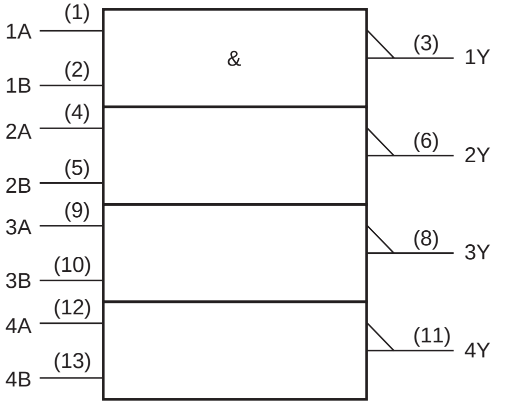


Figure 4.44 ANSI symbol for IC 7400.

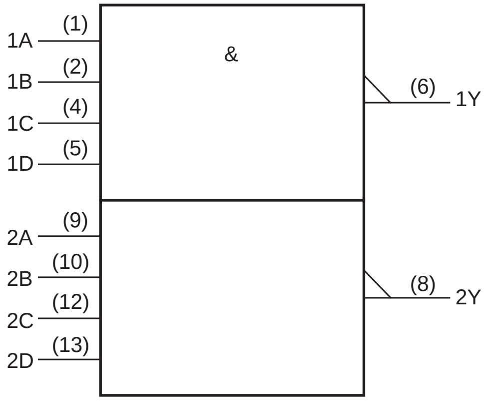


Figure 4.45 ANSI symbol for IC 7420.

Example 4.15

Draw the IEEE/ANSI symbol representation of the logic circuit shown in Fig. 4.46.

Solution

Figure 4.47 shows the circuit using IEEE/ANSI symbols.

4.13 Some Common Applications of Logic Gates

In this section, we will briefly look at some common applications of basic logic gates. The applications discussed here include those where these devices are used to provide a specific function in a larger digital circuit. These also include those where one or more logic gates, along with or without some external components, can be used to build some digital building blocks.

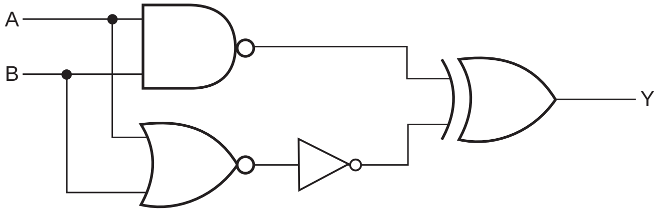


Figure 4.46 Example 4.15.

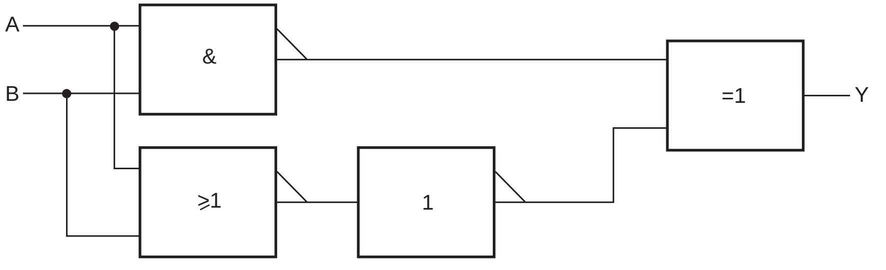


Figure 4.47 Solution to example 4.15.

4.13.1 OR Gate

An OR gate can be used in all those situations where the occurrence of any one or more than one event needs to be detected or acted upon. One such example is an industrial plant where any one or more than one parameter exceeding a preset limiting value should lead to initiation of some kind of protective action. Figure 4.48 shows a typical schematic where the OR gate is used to detect either temperature or pressure exceeding a preset threshold value and produce the necessary command signal for the system.

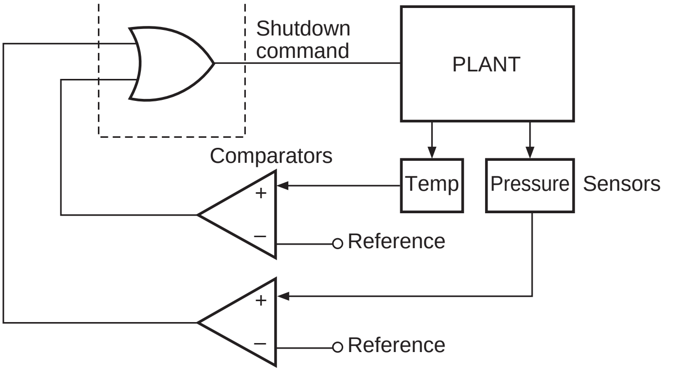


Figure 4.48 Application of an OR gate.

4.13.2 AND Gate

An AND gate is commonly used as an ENABLE or INHIBIT gate to allow or disallow passage of data from one point in the circuit to another. One such application of enabling operation, for instance, is in the measurement of the frequency of a pulsed waveform or the width of a given pulse with the help of a counter. In the case of frequency measurement, a gating pulse of known width is used to enable the passage of the pulse waveform to the counter's clock input. In the case of pulse width measurement, the pulse is used to enable the passage of the clock input to the counter. Figure 4.49 shows the arrangement.

4.13.3 EX-OR/EX-NOR Gate

EX-OR and EX-NOR logic gates are commonly used in parity generation and checking circuits. Figures 4.50(a) and (b) respectively show even and odd parity generator circuits for four-bit data. The circuits are self-explanatory.

The parity check operation can also be performed by similar circuits. Figures 4.51(a) and (b) respectively show simple even and odd parity check circuits for a four-bit data stream. In the circuits shown in Fig. 4.51, a logic '0' at the output signifies correct parity and a logic '1' signifies one-bit error. Parity generator/checker circuits are available in IC form. 74180 in TTL and 40101 in CMOS are nine-bit odd/even parity generator/checker ICs. Parity generation and checking circuits are further discussed in Chapter 7 on arithmetic circuits.

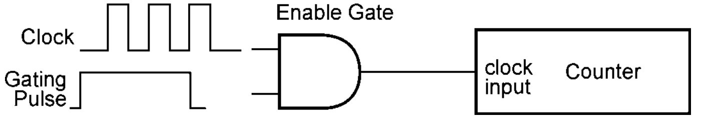


Figure 4.49 Application of an AND gate.

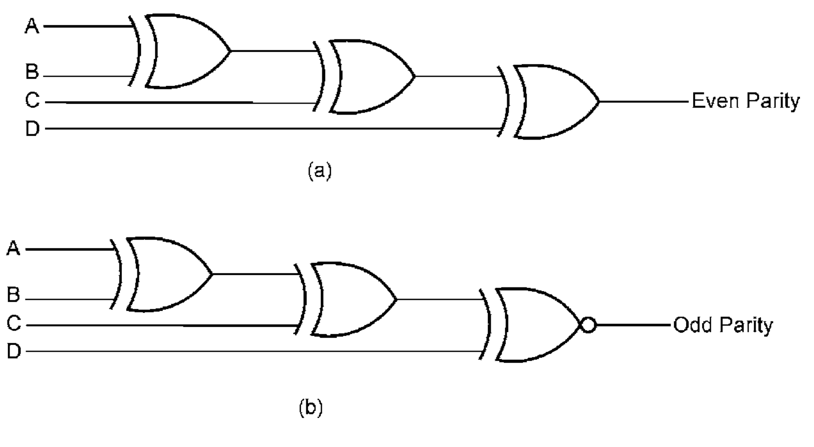


Figure 4.50 Parity generation using EX-OR/EX-NOR gates.

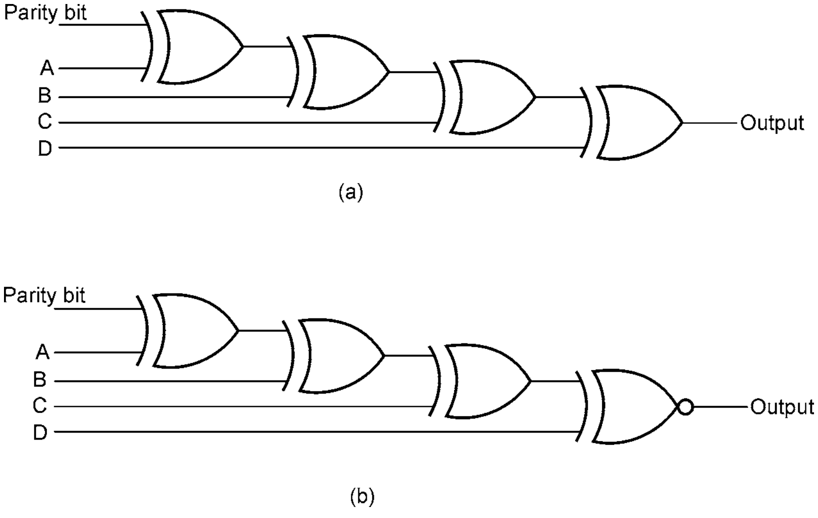


Figure 4.51 Parity check using EX-OR and EX-NOR gates.

4.13.4 Inverter

CMOS inverters are commonly used to build square-wave oscillators for generating clock signals. These clock generators offer good stability, operation over a wide supply voltage range (3–15 V) and frequency range (1 Hz to in excess of 15 MHz), low power consumption and an easy interface to other logic families.

The most fundamental circuit is the ring configuration of any odd number of inverters. Figure 4.52 shows one such circuit using three inverters. Inverting gates such as NAND and NOR gates can also be used instead. This configuration does not make a practical oscillator circuit as its frequency of oscillation is highly susceptible to variation with temperature, supply voltage and external loading. The frequency of oscillation is given by the equation

$$f = 1/(2nt_p) \tag{4.12}$$

where n is the number of inverters and t_p is the propagation delay per gate.

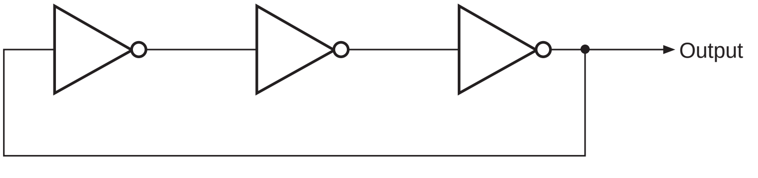


Figure 4.52 Square-wave oscillator using a ring configuration.

Figure 4.53(a) shows a practical oscillator circuit. The frequency of oscillation in this case is given by Equation (4.13) (the duty cycle of the waveform is approximately 50 %):

$$f = 1/2C(0.405R_{eq} + 0.693R_1) \tag{4.13}$$

where $R_{eq} = R_1.R_2/(R_1 + R_2)$.

Figure 4.53(b) shows another circuit using two inverters instead of three inverters. The frequency of oscillation of this circuit is given by the equation

$$f = 1/2.2RC \tag{4.14}$$

The circuits shown in Fig. 4.53 are not as sensitive to supply voltage variations as the one shown in Fig. 4.52. Figure 4.54 shows yet another circuit that is configured around a single Schmitt inverter. The capacitor charges (when the output is HIGH) and discharges (when the output is LOW) between the

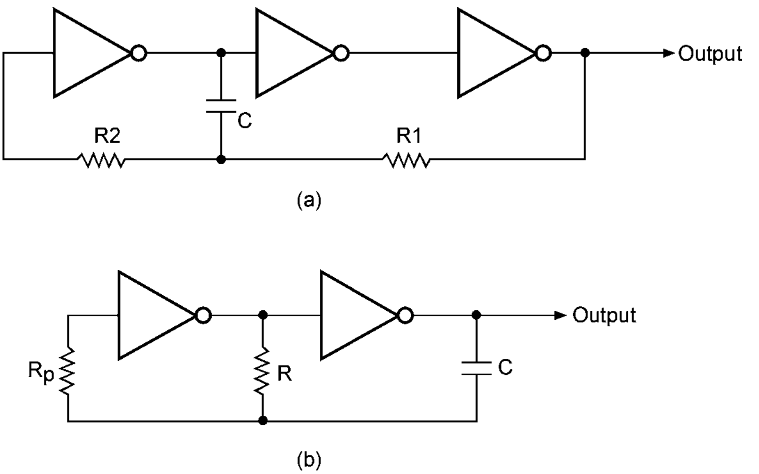


Figure 4.53 Square-wave oscillator with external components.

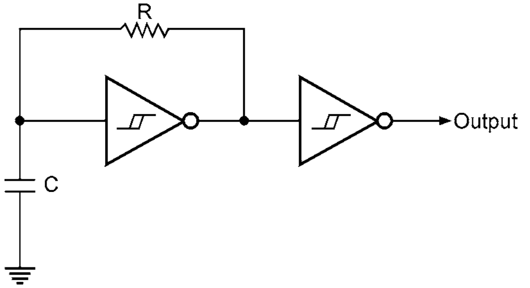


Figure 4.54 Schmitt inverter based oscillator.

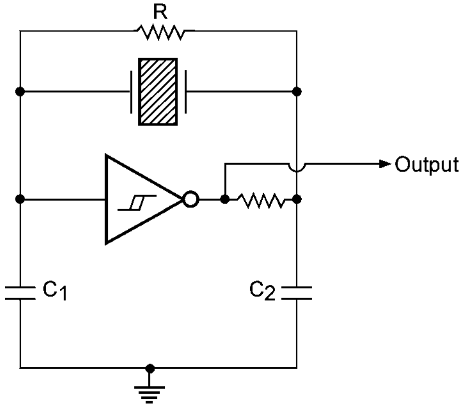


Figure 4.55 Crystal oscillator.

two threshold voltages. The frequency of oscillation, however, is sensitive to supply voltage variations. It is given by the equation

$$f = 1/RC \tag{4.15}$$

Figure 4.55 shows a crystal oscillator configured around a single inverter as the active element. Any odd number of inverters can be used. A larger number of inverters limits the highest attainable frequency of oscillation to a lower value.

4.14 Application-Relevant Information

Table 4.1 lists the commonly used type numbers along with the functional description and the logic family. The pin connection diagrams and the functional tables of the more popular type numbers are given in the companion website.

Table 4.1 Functional index of logic gates.

Type number	Function	Logic family
7400	Quad two-input NAND gate	TTL
7401	Quad two-input NAND gate (open collector)	TTL
7402	Quad two-input NOR gate	TTL
7403	Quad two-input NAND gate (open collector)	TTL
7404	Hex inverter	TTL
7405	Hex inverter (open collector)	TTL
7408	Quad two-input AND gate	TTL
7409	Quad two-input AND gate (open collector)	TTL
7410	Triple three-input NAND gate	TTL

(*Continued overleaf*)

Table 4.1 *(continued).*

Type number	Function	Logic family
7411	Triple three-input AND gate	TTL
7412	Triple three-input NAND gate (open collector)	TTL
7413	Dual four-input Schmitt NAND gate	TTL
7414	Hex Schmitt trigger inverter	TTL
7418	Dual four-input Schmitt NAND gate	TTL
7419	Hex Schmitt trigger inverter	TTL
7420	Dual four-input NAND gate	TTL
7421	Dual four-input AND gate	TTL
7422	Dual four-input NAND gate (open collector)	TTL
7427	Triple three-input NOR gate	TTL
7430	Eight-input NAND gate	TTL
7432	Quad two-input OR gate	TTL
7451	Dual two-wide two-input three-input AND-OR-INVERT gate	TTL
7454	Four-wide two-input AND-OR-INVERT gate	TTL
7455	Two-wide four-input AND-OR-INVERT gate	TTL
7486	Quad two-input EX-OR gate	TTL
74125	Quad tristate noninverting buffer (LOW ENABLE)	TTL
74126	Quad tristate noninverting buffer (HIGH ENABLE)	TTL
74132	Quad two-input Schmitt trigger NAND gate	TTL
74133	13-input NAND gate	TTL
74136	Quad two-input EX-OR gate (open collector)	TTL
74240	Octal tristate inverting bus/line driver	TTL
74241	Octal tristate bus/line driver	TTL
74242	Quad tristate inverting bus transceiver	TTL
74243	Quad tristate noninverting bus transceiver	TTL
74244	Octal tristate noninverting driver	TTL
74245	Octal tristate noninverting bus transceiver	TTL
74266	Quad two-input EXCLUSIVE-NOR gate (open collector)	TTL
74365	Hex tristate noninverting buffer with common ENABLE	TTL
74366	Hex tristate inverting buffer with common ENABLE	TTL
74367	Hex tristate noninverting buffer, four-bit and two-bit	TTL
74368	Hex tristate inverting buffer, four-bit and two-bit	TTL
74386	Quad two-input EX-OR gate	TTL
74465	Octal tristate noninverting buffer Gated ENABLE inverted	TTL
74540	Octal tristate inverting buffer/line driver	TTL
74541	Octal tristate noninverting buffer/line driver	TTL
74640	Octal tristate inverting bus transceiver	TTL
74641	Octal tristate noninverting bus transceiver (open collector)	TTL
74645	Octal tristate noninverting bus transceiver	TTL
4001B	Quad two-input NOR gate	CMOS
4002B	Dual four-input NOR gate	CMOS
4011B	Quad two-input NAND gate	CMOS
4012B	Dual four-input NAND gate	CMOS
4023B	Triple three-input NAND gate	CMOS
4025B	Triple three-input NOR gate	CMOS
4030B	Quad two-input EX-OR gate	CMOS
4049B	Hex inverting buffer	CMOS

Table 4.1 *(continued).*

Type number	Function	Logic family
4050B	Hex noninverting buffer	CMOS
40097B	Tristate hex noninverting buffer	CMOS
40098B	Tristate inverting buffer	CMOS
4069UB	Hex inverter	CMOS
4070B	Quad two-input EX-OR gate	CMOS
4071B	Quad two-input OR gate	CMOS
4081B	Quad two-input AND gate	CMOS
4086B	Four-wide two-input AND-OR-INVERT gate	CMOS
4093B	Quad two-input Schmitt NAND	CMOS
10100	Quad two-input NOR gate with strobe	ECL
10101	Quad two-input OR/NOR gate	ECL
10102	Quad two-input NOR gate	ECL
10103	Quad two-input OR gate	ECL
10104	Quad two-input AND gate	ECL
10113	Quad two-input EX-OR gate	ECL
10114	Triple line receiver	ECL
10115	Quad line Receiver	ECL
10116	Triple Line receiver	ECL
10117	Dual two-wide two- to three-input OR-AND/OR-AND-INVERT gate	ECL
10118	Dual two-wide three-input OR-AND gate	ECL
10123	Triple 4-3-3 input bus driver	
10128	Dual bus driver	ECL
10129	Quad bus driver	ECL
10188	Hex buffer with ENABLE	ECL
10192	Quad bus driver	ECL
10194	Dual simultaneous transceiver	ECL
10195	Hex buffer with invert/noninvert control	ECL

Review Questions

1. How do you distinguish between positive and negative logic systems? Prove that an OR gate in a positive logic system is an AND gate in a negative logic system.
2. Give brief statements that would help one remember the truth table of AND, NAND, OR, NOR, EX-OR and EX-NOR logic gate functions, irrespective of the number of inputs used.
3. Why are NAND and NOR gates called universal gates? Justify your answer with the help of examples.
4. What are Schmitt gates? How does a Schmitt gate overcome the problem of occurrence of an erratic output for slow varying input transitions?
5. What are logic gates with open collector or open drain outputs? What are the major advantages and disadvantages of such devices?
6. Draw the circuit symbol and the associated truth table for the following:

   (a) a tristate noninverting buffer with an active HIGH ENABLE input;
   (b) a tristate inverting buffer with an active LOW ENABLE input;

(c) a three-input NAND with an open collector output;
(d) a four-input INHIBIT gate.

7. Define the fan-out specification of a logic gate. Which parameters would you need to know from the data sheet of a logic gate to determine for yourself the fan-out in case it is not mentioned in the data sheet? Explain the procedure for determining the fan-out specification from those parameters. What are the consequences of exceeding the fan-out specification?
8. What is the main significance of IEEE/ANSI symbols when compared with the conventional ones? Draw the ANSI symbols for four-input OR, two-input AND, two-input EX-OR and two-input NAND gates.

Problems

1. What is the only input combination that:

(a) Will produce a logic '1' at the output of an eight-input AND gate?
(b) Will produce a logic '0' at the output of a four-input NAND gate?
(c) Will produce a logic '1' at the output of an eight-input NOR gate?
(d) Will produce a logic '0' at the output of a four-input OR gate?

(a) 11111111; (b) 1111; (c) 00000000; (d) 0000

2. Draw the truth table of the logic circuit shown in Fig. 4.56.

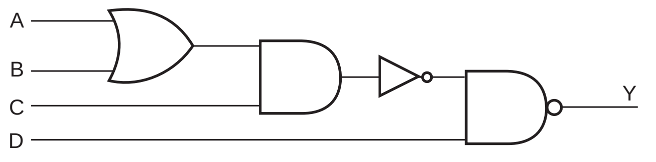


Figure 4.56 Problem 2.

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Figure 4.57 Solution of problem 2.

3. Redraw the logic circuit of Fig. 4.56 using IEEE/ANSI symbols.

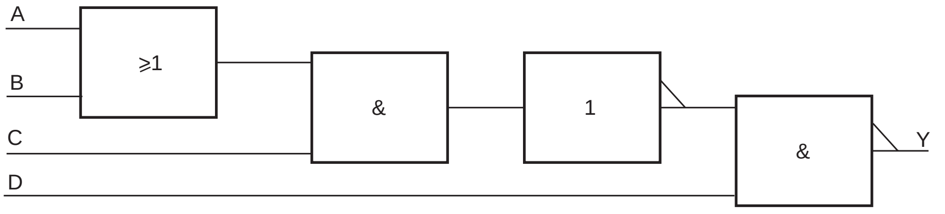


Figure 4.58 Solution to problem 3.

4. Refer to Fig. 4.59(a). The ENABLE waveforms applied at A and B inputs are respectively shown in Figs 4.59(b) and (c). What is the output state of inverter 3 and inverter 4 at (i) $t = 3$ ms and (ii) $t = 5$ ms?

(i) The output of inverter 3 = high Z, while the output of inverter 4 = logic '1'
(ii) The output of inverter 3 = logic '0', while the output of inverter 4 = high Z

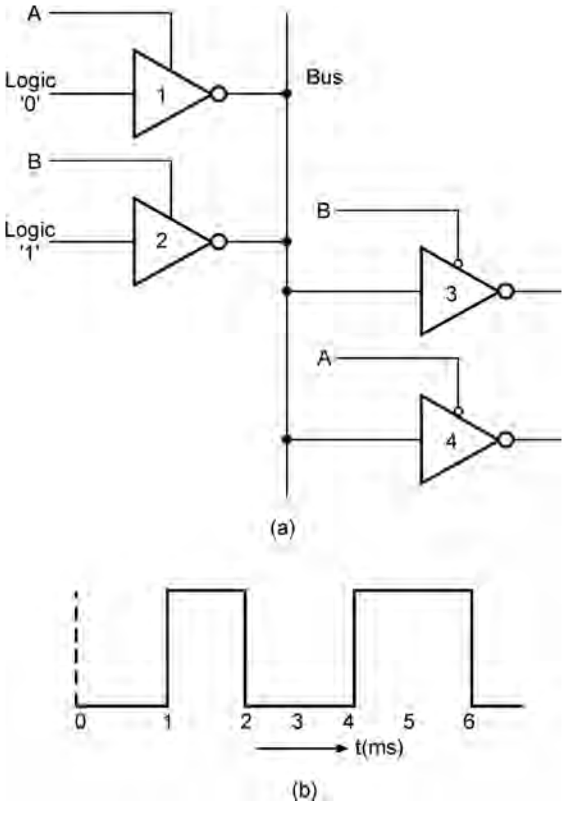


Figure 4.59 Problem 4.

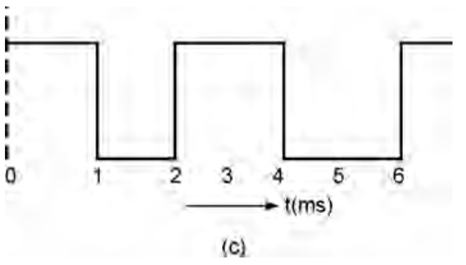


Figure 4.59 (*Continued*)

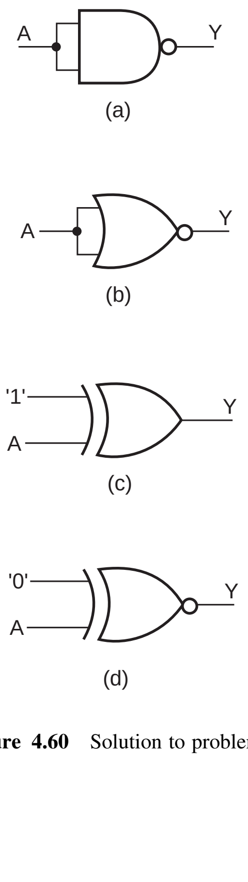


Figure 4.60 Solution to problem 5.

5. Draw logic implementation of an inverter using (i) two-input NAND, (ii) two-input NOR, (iii) two-input EX-OR and (iv) two-input EX-NOR.

(i) Fig. 4.60(a); (ii) Fig. 4.60(b); (iii) Fig. 4.60(c); (iv) Fig. 4.60(d)

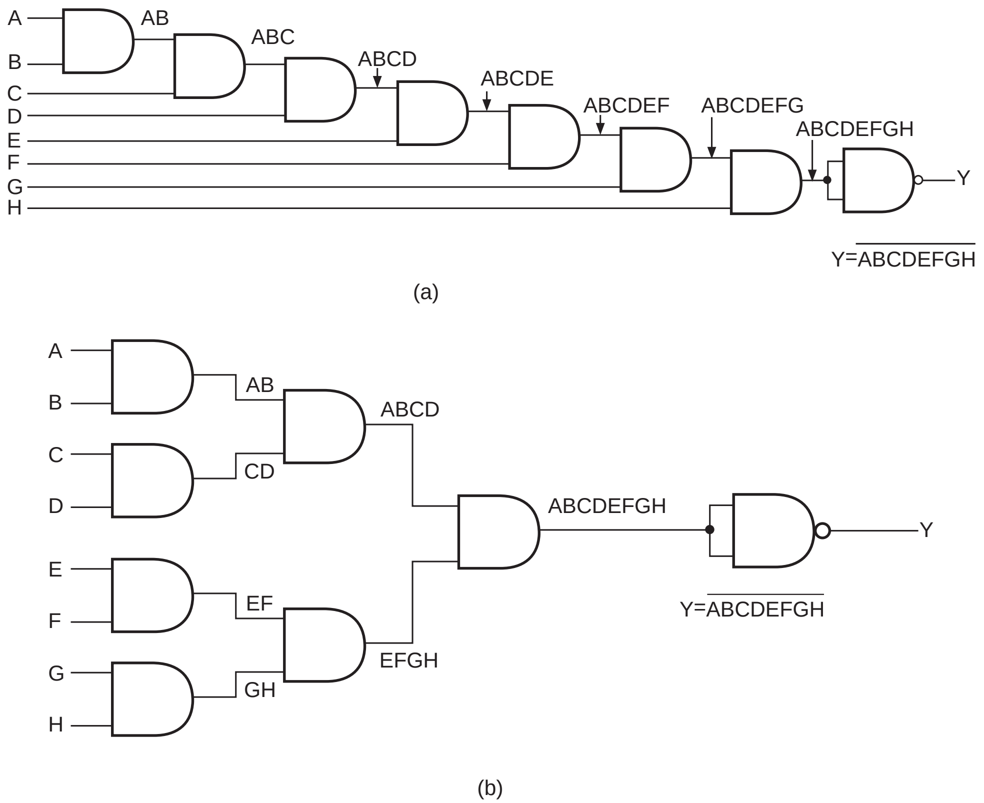


Figure 4.61 Solution to problem 6.

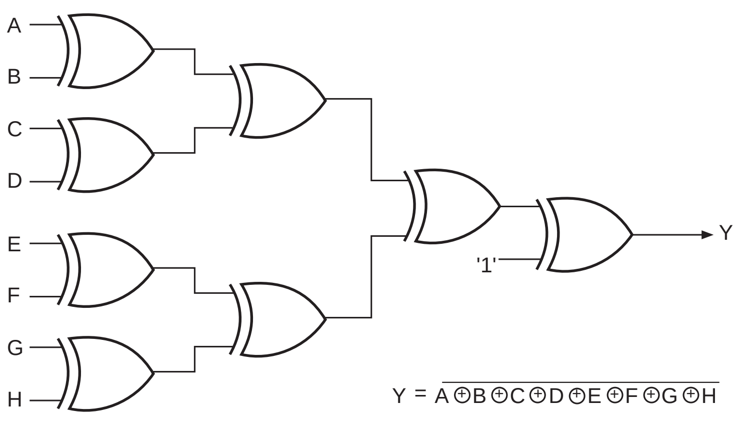


Figure 4.62 Solution to problem 7.

6. It is proposed to construct an eight-input NAND gate using only two-input AND gates and two-input NAND gates. Draw the logic arrangement that uses the minimum number of logic gates.
   The two possible logic circuits are shown in Figs 4.61(a) and (b)

7. Draw the logic diagram to implement an eight-input EX-NOR function using the minimum number of two-input logic gates.

Further Reading

1. Cook, N. P. (2003) *Practical Digital Electronics*, Prentice-Hall, NJ, USA.
2. Fairchild Semiconductor Corporation (October 1974) *CMOS Oscillators*, Application Note 118, South Portland, ME, USA.
3. Holdsworth, B. and Woods, C. (2002) *Digital Logic Design*, Newnes, Oxford, UK.
4. Langholz, G., Mott, J. L. and Kandel, A. (1998) *Foundations of Digital Logic Design*, World Scientific Publ. Co. Inc., Singapore.
5. Chen, W.-K. (2003) *Logic Design*, CRC Press, FL, USA.

5

Logic Families

Digital integrated circuits are produced using several different circuit configurations and production technologies. Each such approach is called a specific logic family. In this chapter, we will discuss different logic families used to hardware-implement different logic functions in the form of digital integrated circuits. The chapter begins with an introduction to logic families and the important parameters that can be used to characterize different families. This is followed by a detailed description of common logic families in terms of salient features, internal circuitry and interface aspects. Logic families discussed in the chapter include transistor transistor logic (TTL), metal oxide semiconductor (MOS) logic, emitter coupled logic (ECL), bipolar-CMOS (Bi-CMOS) logic and integrated injection logic (I^2L).

5.1 Logic Families – Significance and Types

There are a variety of circuit configurations or more appropriately various approaches used to produce different types of digital integrated circuit. Each such fundamental approach is called a *logic family*. The idea is that different logic functions, when fabricated in the form of an IC with the same approach, or in other words belonging to the same logic family, will have identical electrical characteristics. These characteristics include supply voltage range, speed of response, power dissipation, input and output logic levels, current sourcing and sinking capability, fan-out, noise margin, etc. In other words, the set of digital ICs belonging to the same logic family are electrically compatible with each other.

5.1.1 Significance

A digital system in general comprises digital ICs performing different logic functions, and choosing these ICs from the same logic family guarantees that different ICs are compatible with respect to each

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

other and that the system as a whole performs the intended logic function. In the case where the output of an IC belonging to a certain family feeds the inputs of another IC belonging to a different family, we must use established interface techniques to ensure compatibility. Understanding the features and capabilities of different logic families is very important for a logic designer who is out to make an optimum choice for his new digital design from the available logic family alternatives. A not so well thought out choice can easily underkill or overkill the design with either inadequate or excessive capabilities.

5.1.2 Types of Logic Family

The entire range of digital ICs is fabricated using either bipolar devices or MOS devices or a combination of the two. Different logic families falling in the first category are called bipolar families, and these include diode logic (DL), resistor transistor logic (RTL), diode transistor logic (DTL), transistor transistor logic (TTL), emitter coupled logic (ECL), also known as current mode logic (CML), and integrated injection logic (I^2L). The logic families that use MOS devices as their basis are known as MOS families, and the prominent members belonging to this category are the PMOS family (using P-channel MOSFETs), the NMOS family (using N-channel MOSFETs) and the CMOS family (using both N- and P-channel devices). The Bi-MOS logic family uses both bipolar and MOS devices.

Of all the logic families listed above, the first three, that is, diode logic (DL), resistor transistor logic (RTL) and diode transistor logic (DTL), are of historical importance only. Diode logic used diodes and resistors and in fact was never implemented in integrated circuits. The RTL family used resistors and bipolar transistors, while the DTL family used resistors, diodes and bipolar transistors. Both RTL and DTL suffered from large propagation delay owing to the need for the transistor base charge to leak out if the transistor were to switch from conducting to nonconducting state. Figure 5.1 shows the simplified schematics of a two-input AND gate using DL [Fig. 5.1(a)], a two-input NOR gate using RTL [Fig. 5.1(b)] and a two-input NAND gate using DTL [Fig. 5.1(c)]. The DL, RTL and DTL families, however, were rendered obsolete very shortly after their introduction in the early 1960s owing to the arrival on the scene of transistor transistor logic (TTL).

Logic families that are still in widespread use include TTL, CMOS, ECL, NMOS and Bi-CMOS. The PMOS and I^2L logic families, which were mainly intended for use in custom large-scale integrated (LSI) circuit devices, have also been rendered more or less obsolete, with the NMOS logic family replacing them for LSI and VLSI applications.

5.1.2.1 TTL Subfamilies

The TTL family has a number of subfamilies including standard TTL, low-power TTL, high-power TTL, low-power Schottky TTL, Schottky TTL, advanced low-power Schottky TTL, advanced Schottky TTL and fast TTL. The ICs belonging to the TTL family are designated as 74 or 54 (for standard TTL), 74L or 54L (for low-power TTL), 74H or 54H (for high-power TTL), 74LS or 54LS (for low-power Schottky TTL), 74S or 54S (for Schottky TTL), 74ALS or 54ALS (for advanced low-power Schottky TTL), 74AS or 54AS (for advanced Schottky TTL) and 74F or 54F (for fast TTL). An alphabetic code preceding this indicates the name of the manufacturer (DM for National Semiconductors, SN for Texas Instruments and so on). A two-, three- or four-digit numerical code tells the logic function performed by the IC. It may be mentioned that 74-series devices and 54-series devices are identical except for their operational temperature range. The 54-series devices are MIL-qualified (operational temperature range: $-55\,°C$ to $+125\,°C$) versions of the corresponding 74-series ICs (operational temperature range: $0\,°C$ to $70\,°C$). For example, 7400 and 5400 are both quad two-input NAND gates.

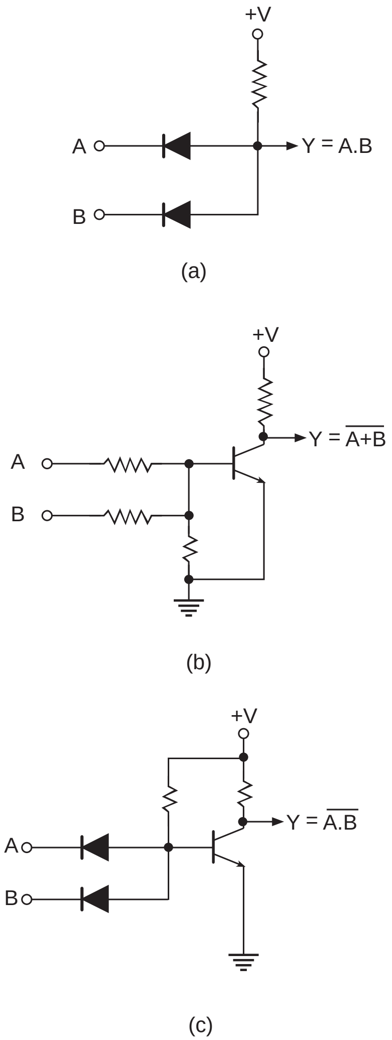


Figure 5.1 (a) Diode logic (b) resistor transistor logic and (c) diode transistor logic.

5.1.2.2 CMOS Subfamilies

The popular CMOS subfamilies include the 4000A, 4000B, 4000UB, 54/74C, 54/74HC, 54/74HCT, 54/74AC and 54/74ACT families. The 4000A CMOS family has been replaced by its high-voltage versions in the 4000B and 4000UB CMOS families, with the former having buffered and the latter having unbuffered outputs. 54/74C, 54/74HC, 54/74HCT, 54/74AC and 54/74ACT are CMOS logic families with pin-compatible 54/74 TTL series logic functions.

5.1.2.3 ECL Subfamilies

The first monolithic emitter coupled logic family was introduced by ON Semiconductor, formerly a division of Motorola, with the MECL-I series of devices in 1962, with the MECL-II series following it up in 1966. Both these logic families have become obsolete. Currently, popular subfamilies of ECL logic include MECL-III (also called the MC 1600 series), the MECL-10K series, the MECL-10H series and the MECL-10E series (ECLinPS and ECLinPSLite). The MECL-10K series further divided into the 10 100-series and 10 200-series devices.

5.2 Characteristic Parameters

In this section, we will briefly describe the parameters used to characterize different logic families. Some of these characteristic parameters, as we will see in the paragraphs to follow, are also used to compare different logic families.

- **HIGH-level input current, I_{IH}**. This is the current flowing into (taken as positive) or out of (taken as negative) an input when a HIGH-level input voltage equal to the minimum HIGH-level output voltage specified for the family is applied. In the case of bipolar logic families such as TTL, the circuit design is such that this current flows into the input pin and is therefore specified as positive. In the case of CMOS logic families, it could be either positive or negative, and only an absolute value is specified in this case.
- **LOW-level input current, I_{IL}**. The LOW-level input current is the maximum current flowing into (taken as positive) or out of (taken as negative) the input of a logic function when the voltage applied at the input equals the maximum LOW-level output voltage specified for the family. In the case of bipolar logic families such as TTL, the circuit design is such that this current flows out of the input pin and is therefore specified as negative. In the case of CMOS logic families, it could be either positive or negative. In this case, only an absolute value is specified.

HIGH-level and LOW-level input current or loading are also sometimes defined in terms of *unit load* (UL). For devices of the TTL family, 1 UL (HIGH) = 40 μA and 1 UL (LOW) = 1.6 mA.

- **HIGH-level output current, I_{OH}**. This is the maximum current flowing out of an output when the input conditions are such that the output is in the logic HIGH state. It is normally shown as a negative number. It tells about the current sourcing capability of the output. The magnitude of I_{OH} determines the number of inputs the logic function can drive when its output is in the logic HIGH state. For example, for the standard TTL family, the minimum guaranteed I_{OH} is −400 μA, which can drive 10 standard TTL inputs with each requiring 40 μA in the HIGH state, as shown in Fig. 5.2(a).
- **LOW-level output current, I_{OL}**. This is the maximum current flowing into the output pin of a logic function when the input conditions are such that the output is in the logic LOW state. It tells about the current sinking capability of the output. The magnitude of I_{OL} determines the number of inputs the logic function can drive when its output is in the logic LOW state. For example, for the standard TTL family, the minimum guaranteed I_{OL} is 16 mA, which can drive 10 standard TTL inputs with each requiring 1.6 mA in the LOW state, as shown in Fig. 5.2(b).
- **HIGH-level off-state (high-impedance state) output current, I_{OZH}**. This is the current flowing into an output of a tristate logic function with the ENABLE input chosen so as to establish a high-impedance state and a logic HIGH voltage level applied at the output. The input conditions are chosen so as to produce logic LOW if the device is enabled.

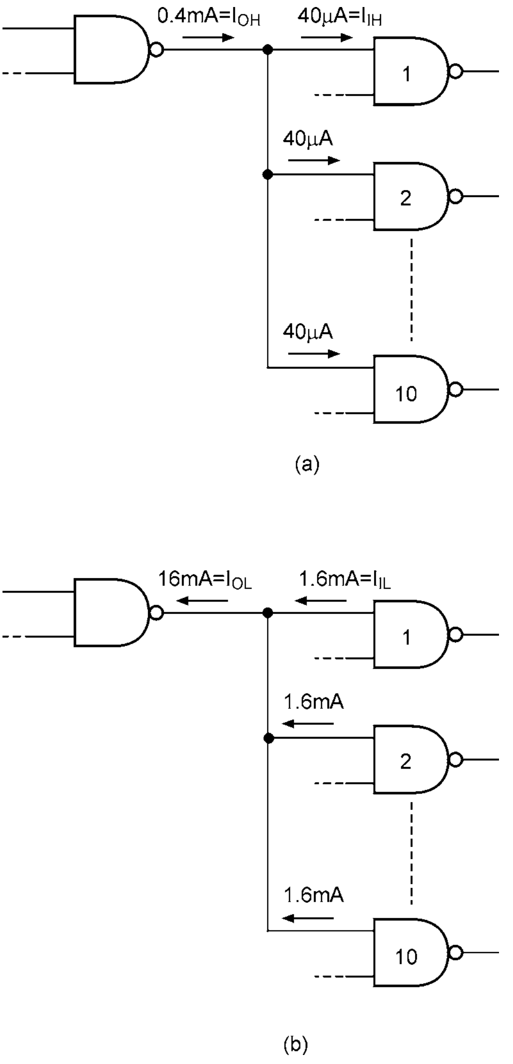


Figure 5.2 Input and output current specifications.

- **LOW-level off-state (high-impedance state) output current, I_{OZL}.** This is the current flowing into an output of a tristate logic function with the ENABLE input chosen so as to establish a high-impedance state and a logic LOW voltage level applied at the output. The input conditions are chosen so as to produce logic HIGH if the device is enabled.
- **HIGH-level input voltage, V_{IH}.** This is the minimum voltage level that needs to be applied at the input to be recognized as a legal HIGH level for the specified family. For the standard TTL family, a 2 V input voltage is a legal HIGH logic state.

- **LOW-level input voltage, V_{IL}.** This is the maximum voltage level applied at the input that is recognized as a legal LOW level for the specified family. For the standard TTL family, an input voltage of 0.8 V is a legal LOW logic state.
- **HIGH-level output voltage, V_{OH}.** This is the minimum voltage on the output pin of a logic function when the input conditions establish logic HIGH at the output for the specified family. In the case of the standard TTL family of devices, the HIGH level output voltage can be as low as 2.4 V and still be treated as a legal HIGH logic state. It may be mentioned here that, for a given logic family, the V_{OH} specification is always greater than the V_{IH} specification to ensure output-to-input compatibility when the output of one device feeds the input of another.
- **LOW-level output voltage, V_{OL}.** This is the maximum voltage on the output pin of a logic function when the input conditions establish logic LOW at the output for the specified family. In the case of the standard TTL family of devices, the LOW-level output voltage can be as high as 0.4 V and still be treated as a legal LOW logic state. It may be mentioned here that, for a given logic family, the V_{OL} specification is always smaller than the V_{IL} specification to ensure output-to-input compatibility when the output of one device feeds the input of another.

The different input/output current and voltage parameters are shown in Fig. 5.3, with HIGH-level current and voltage parameters in Fig. 5.3(a) and LOW-level current and voltage parameters in Fig. 5.3(b). It may be mentioned here that the direction of the LOW-level input and output currents shown in Fig. 5.3(b) is applicable to logic families with current-sinking action such as TTL.

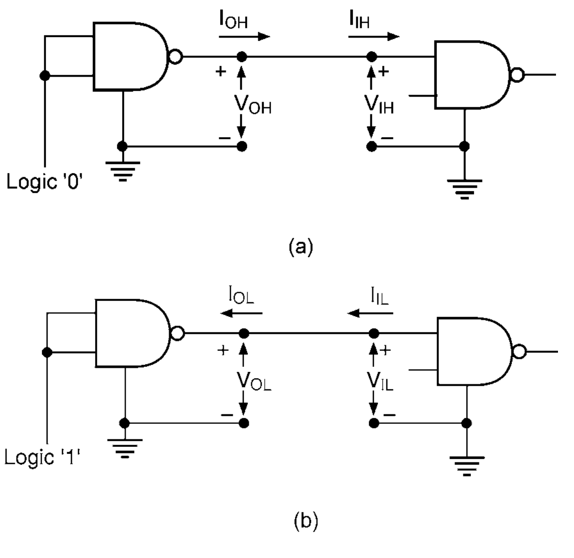


Figure 5.3 (a) HIGH-level current and voltage parameters and (b) LOW-level current and voltage parameters.

- **Supply current, I_{CC}.** The supply current when the output is HIGH, LOW and in the high-impedance state is respectively designated as I_{CCH}, I_{CCL} and I_{CCZ}.
- **Rise time, t_r.** This is the time that elapses between 10 and 90 % of the final signal level when the signal is making a transition from logic LOW to logic HIGH.
- **Fall time, t_f.** This is the time that elapses between 90 and 10 % of the signal level when it is making HIGH to LOW transition.
- **Propagation delay t_p.** The propagation delay is the time delay between the occurrence of change in the logical level at the input and before it is reflected at the output. It is the time delay between the specified voltage points on the input and output waveforms. Propagation delays are separately defined for LOW-to-HIGH and HIGH-to-LOW transitions at the output. In addition, we also define enable and disable time delays that occur during transition between the high-impedance state and defined logic LOW or HIGH states.
- **Propagation delay t_{pLH}.** This is the time delay between specified voltage points on the input and output waveforms with the output changing from LOW to HIGH.
- **Propagation delay t_{pHL}.** This is the time delay between specified voltage points on the input and output waveforms with the output changing from HIGH to LOW. Figure 5.4 shows the two types of propagation delay parameter.
- **Disable time from the HIGH state, t_{pHZ}.** Defined for a tristate device, this is the time delay between specified voltage points on the input and output waveforms with the tristate output changing from the logic HIGH level to the high-impedance state.
- **Disable time from the LOW state, t_{pLZ}.** Defined for a tristate device, this is the time delay between specified voltage points on the input and output waveforms with the tristate output changing from the logic LOW level to the high-impedance state.
- **Enable time from the HIGH state, t_{pZH}.** Defined for a tristate device, this is the time delay between specified voltage points on the input and output waveforms with the tristate output changing from the high-impedance state to the logic HIGH level.

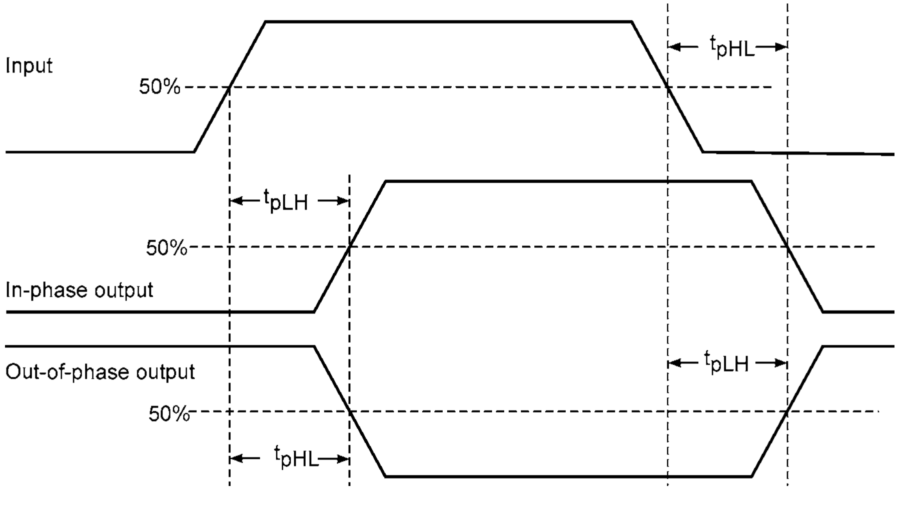


Figure 5.4 Propagation delay parameters.

- **Enable time from the LOW state, t_{pZL}.** Defined for a tristate device, this is the time delay between specified voltage points on the input and output waveforms with the tristate output changing from the high-impedance state to the logic LOW level.
- **Maximum clock frequency, f_{max}.** This is the maximum frequency at which the clock input of a flip-flop can be driven through its required sequence while maintaining stable transitions of logic level at the output in accordance with the input conditions and the product specification. It is also referred to as the maximum toggle rate for a flip-flop or counter device.
- **Power dissipation.** The power dissipation parameter for a logic family is specified in terms of power consumption per gate and is the product of supply voltage V_{CC} and supply current I_{CC}. The supply current is taken as the average of the HIGH-level supply current I_{CCH} and the LOW-level supply current I_{CCL}.
- **Speed–power product.** The speed of a logic circuit can be increased, that is, the propagation delay can be reduced, at the expense of power dissipation. We will recall that, when a bipolar transistor switches between cut-off and saturation, it dissipates the least power but has a large associated switching time delay. On the other hand, when the transistor is operated in the active region, power dissipation goes up while the switching time decreases drastically. It is always desirable to have in a logic family low values for both propagation delay and power dissipation parameters. A useful figure-of-merit used to evaluate different logic families is the speed–power product, expressed in picojoules, which is the product of the propagation delay (measured in nanoseconds) and the power dissipation per gate (measured in milliwatts).
- **Fan-out.** The fan-out is the number of inputs of a logic function that can be driven from a single output without causing any false output. It is a characteristic of the logic family to which the device belongs. It can be computed from I_{OH}/I_{IH} in the logic HIGH state and from I_{OL}/I_{IL} in the logic LOW state. If, in a certain case, the two values I_{OH}/I_{IH} and I_{OL}/I_{IL} are different, the fan-out is taken as the smaller of the two. This description of the fan-out is true for bipolar logic families like TTL and ECL. When determining the fan-out of CMOS logic devices, we should also take into consideration how much input load capacitance can be driven from the output without exceeding the acceptable value of propagation delay.
- **Noise margin.** This is a quantitative measure of noise immunity offered by the logic family. When the output of a logic device feeds the input of another device of the same family, a legal HIGH logic state at the output of the feeding device should be treated as a legal HIGH logic state by the input of the device being fed. Similarly, a legal LOW logic state of the feeding device should be treated as a legal LOW logic state by the device being fed. We have seen in earlier paragraphs while defining important characteristic parameters that legal HIGH and LOW voltage levels for a given logic family are different for outputs and inputs. Figure 5.5 shows the generalized case of legal HIGH and LOW voltage levels for output [Fig. 5.5(a)] and input [Fig. 5.5(b)]. As we can see from the two diagrams, there is a disallowed range of output voltage levels from V_{OL}(max.) to V_{OH}(min.) and an indeterminate range of input voltage levels from V_{IL}(max.) to V_{IH}(min.). Since V_{IL}(max.) is greater than V_{OL}(max.), the LOW output state can therefore tolerate a positive voltage spike equal to V_{IL}(max.) $-$ V_{OL}(max.) and still be a legal LOW input. Similarly, V_{OH}(min.) is greater than V_{IH} (min.), and the HIGH output state can tolerate a negative voltage spike equal to V_{OH}(min.) $-$ V_{IH} (min.) and still be a legal HIGH input. Here, V_{IL}(max.) $-$ V_{OL}(max.) and V_{OH}(min.) $-$ V_{IH} (min.) are respectively known as the LOW-level and HIGH-level noise margin.

Let us illustrate it further with the help of data for the standard TTL family. The minimum legal HIGH output voltage level in the case of the standard TTL is 2.4 V. Also, the minimum legal HIGH input voltage level for this family is 2 V. This implies that, when the output of one device feeds the input of another, there is an available margin of 0.4 V. That is, any negative voltage spikes of amplitude

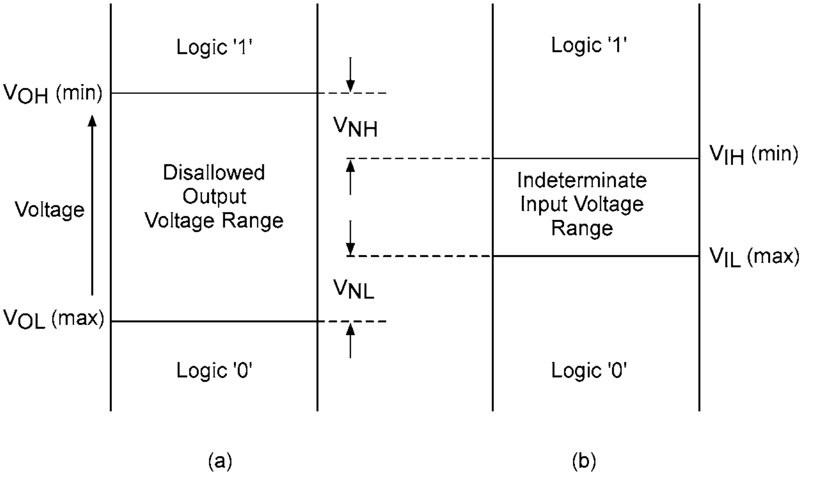


Figure 5.5 Noise margin.

less than or equal to 0.4 V on the signal line do not cause any spurious transitions. Similarly, when the output is in the logic LOW state, the maximum legal LOW output voltage level in the case of the standard TTL is 0.4 V. Also, the maximum legal LOW input voltage level for this family is 0.8 V. This implies that, when the output of one device feeds the input of another, there is again an available margin of 0.4 V. That is, any positive voltage spikes of amplitude less than or equal to 0.4 V on the signal line do not cause any spurious transitions. This leads to the standard TTL family offering a noise margin of 0.4 V. To generalize, the noise margin offered by a logic family, as outlined earlier, can be computed from the HIGH-state noise margin, $V_{NH} = V_{OH}$(min.) – V_{IH}(min.), and the LOW-state noise margin, $V_{NL} = V_{IL}$(max.) – V_{OL}(max.). If the two values are different, the noise margin is taken as the lower of the two.

Example 5.1

The data sheet of a quad two-input NAND gate specifies the following parameters: I_{OH}(max.) = 0.4 mA, V_{OH}(min.) = 2.7 V, V_{IH}(min.) = 2V, V_{IL}(max.) = 0.8 V, V_{OL}(max.) = 0.4 V, I_{OL}(max.) = 8 mA, I_{IL}(max.) = 0.4 mA, I_{IH} (max.) = 20 μA, I_{CCH}(max.) = 1.6 mA, I_{CCL}(max.) = 4.4 mA, $t_{pLH} = t_{pHL}$ = 15 ns and a supply voltage range of 5 V. Determine (a) the average power dissipation of a single NAND gate, (b) the maximum average propagation delay of a single gate, (c) the HIGH-state noise margin and (d) the LOW-state noise margin

Solution

(a) The average supply current = $(I_{CCH} + I_{CCL})/2 = (1.6 + 4.4)/2 = 3$ mA.
The supply voltage $V_{CC} = 5$ V.
Therefore, the power dissipation for all four gates in the IC = 5 × 3 = 15 mW.
The average power dissipation per gate = 15/4 = 3.75 mW.
(b) The propagation delay = 15 ns.
(c) The HIGH-state noise margin = V_{OH}(min.) – V_{IH}(min.) = 2.7 – 2 = 0.7 V.
(d) The LOW-state noise margin = V_{IL}(max.) – V_{OL}(max.) = 0.8 – 0.4 = 0.4 V.

Example 5.2

Refer to example 5.1. How many NAND gate inputs can be driven from the output of a NAND gate of this type?

Solution

- This figure is given by the worst-case fan-out specification of the device.
- Now, the HIGH-state fan-out = I_{OH}/I_{IH} = 400/20 = 20.
- The LOW-state fan-out = I_{OL}/I_{IL} = 8/0.4 = 20.
- Therefore, the number of inputs that can be driven from a single output = 20.

Example 5.3

Determine the fan-out of IC 74LS04, given the following data: input loading factor (HIGH state) = 0.5 UL, input loading factor (LOW state) = 0.25 UL, output loading factor (HIGH state) = 10 UL, output loading factor (LOW state) = 5 UL, where UL is the unit load.

Solution

- The HIGH-state fan-out can be computed from: fan-out = output loading factor (HIGH)/input loading factor (HIGH) = 10 UL/0.5 UL = 20.
- The LOW-state fan-out can be computed from: fan-out = output loading factor (LOW)/input loading factor (LOW) = 5 UL/0.25 UL = 20.
- Since the fan-out in the two cases turns out to be the same, it follows that the fan-out = 20.

Example 5.4

A certain TTL gate has I_{IH} = 20 μA, I_{IL} = 0.1 mA, I_{OH} = 0.4 mA and I_{OL} = 4 mA. Determine the input and output loading in the HIGH and LOW states in terms of UL.

Solution

- 1 UL (LOW state) = 1.6 mA and 1 UL (HIGH state) = 40 μA.
- The input loading factor (HIGH state) = 20 μA = 20/40 = 0.5 UL.
- The input loading factor (LOW state) = 0.1 mA = 0.1/1.6 = 1/16 UL
- The output loading factor (HIGH state) = 0.4 mA = 0.4/0.04 = 10 UL.
- The output loading factor (LOW state) = 4 mA = 4/1.6 = 2.5 UL.

5.3 Transistor Transistor Logic (TTL)

TTL as outlined above stands for transistor transistor logic. It is a logic family implemented with bipolar process technology that combines or integrates NPN transistors, PN junction diodes and diffused resistors in a single monolithic structure to get the desired logic function. The NAND gate is the basic building block of this logic family. Different subfamilies in this logic family, as outlined earlier, include standard TTL, low-power TTL, high-power TTL, low-power Schottky TTL, Schottky TTL, advanced low-power Schottky TTL, advanced Schottky TTL and fast TTL. In the following paragraphs, we will briefly describe each of these subfamilies in terms of internal structure and characteristic parameters.

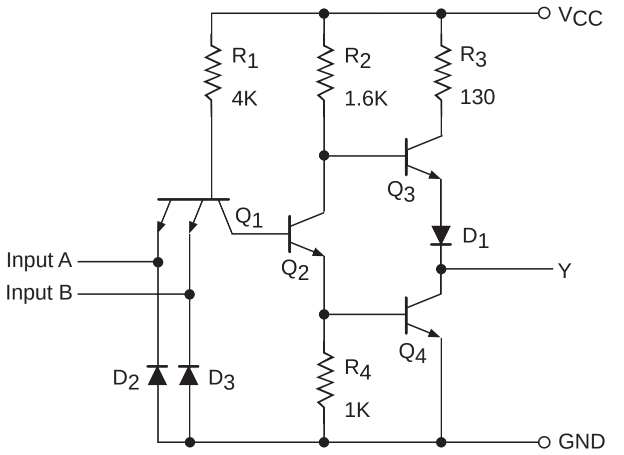


Figure 5.6 Standard TTL NAND gate.

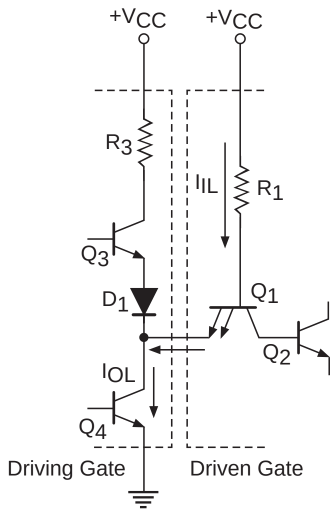
5.3.1 *Standard TTL*

Figure 5.6 shows the internal schematic of a standard TTL NAND gate. It is one of the four circuits of 5400/7400, which is a quad two-input NAND gate. The circuit operates as follows. Transistor Q_1 is a two-emitter NPN transistor, which is equivalent to two NPN transistors with their base and emitter terminals tied together. The two emitters are the two inputs of the NAND gate. Diodes D_2 and D_3 are used to limit negative input voltages. We will now examine the behaviour of the circuit for various possible logic states at the two inputs.

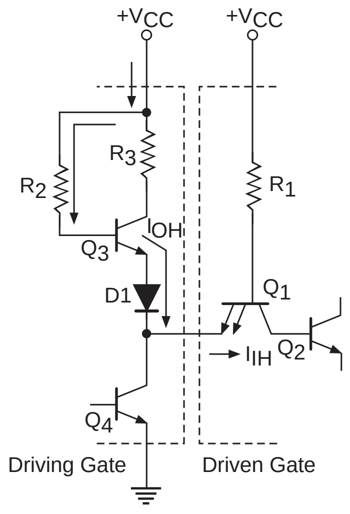
5.3.1.1 Circuit Operation

When both the inputs are in the logic HIGH state as specified by the TTL family ($V_{IH} = 2$ V minimum), the current flows through the base-collector PN junction diode of transistor Q_1 into the base of transistor Q_2. Transistor Q_2 is turned ON to saturation, with the result that transistor Q_3 is switched OFF and transistor Q_4 is switched ON. This produces a logic LOW at the output, with V_{OL} being 0.4 V maximum when it is sinking a current of 16 mA from external loads represented by inputs of logic functions being driven by the output. The current-sinking action is shown in Fig. 5.7(a). Transistor Q_4 is also referred to as the current-sinking or pull-down transistor, for obvious reasons. Diode D_1 is used to prevent transistor Q_3 from conducting even a small amount of current when the output is LOW. When the output is LOW, Q_4 is in saturation and Q_3 will conduct slightly in the absence of D_1. Also, the input current I_{IH} in the HIGH state is nothing but the reverse-biased junction diode leakage current and is typically 40 μA.

When either of the two inputs or both inputs are in the logic LOW state, the base-emitter region of Q_1 conducts current, driving Q_2 to cut-off in the process. When Q_2 is in the cut-off state, Q_3 is driven to conduction and Q_4 to cut-off. This produces a logic HIGH output with V_{OH}(min.) = 2.4 V guaranteed for minimum supply voltage V_{CC} and a source current of 400 μA. The current-sourcing action is shown in Fig. 5.7(b). Transistor Q_3 is also referred to as the current-sourcing or pull-up transistor. Also, the LOW-level input current I_{IL}, given by $(V_{CC} - V_{BE1})/R_1$, is 1.6 mA (max.) for maximum V_{CC}.



(a)



(b)

Figure 5.7 (a) Current sinking action and (b) current sourcing action.

5.3.1.2 Totem-Pole Output Stage

Transistors Q_3 and Q_4 constitute what is known as a totem-pole output arrangement. In such an arrangement, either Q_3 or Q_4 conducts at a time depending upon the logic status of the inputs. The totem-pole arrangement at the output has certain distinct advantages. The major advantage of using a totem-pole connection is that it offers low-output impedance in both the HIGH and LOW output states. In the HIGH state, Q_3 acts as an emitter follower and has an output impedance of about 70 Ω. In the LOW state, Q_4 is saturated and the output impedance is approximately 10 Ω. Because of the low output impedance, any stray capacitance at the output can be charged or discharged very rapidly through this low impedance, thus allowing quick transitions at the output from one state to the other. Another advantage is that, when the output is in the logic LOW state, transistor Q_4 would need to conduct a fairly large current if its collector were tied to V_{CC} through R_3 only. A nonconducting Q_3 overcomes this problem. A disadvantage of the totem-pole output configuration results from the switch-off action of Q_4 being slower than the switch-on action of Q_3. On account of this, there will be a small fraction of time, of the order of a few nanoseconds, when both the transistors are conducting, thus drawing heavy current from the supply.

5.3.1.3 Characteristic Features

To sum up, the characteristic parameters and features of the standard TTL family of devices include the following: $V_{IL} = 0.8\,V$; $V_{IH} = 2\,V$; $I_{IH} = 40\,\mu A$; $I_{IL} = 1.6\,mA$; $V_{OH} = 2.4\,V$; $V_{OL} = 0.4\,V$; $I_{OH} = 400\,\mu A$; $I_{OL} = 16\,mA$; $V_{CC} = 4.75–5.25\,V$ (74-series) and 4.5–5.5 V (54-series); propagation delay (for a load resistance of 400 Ω, a load capacitance of 15 pF and an ambient temperature of 25 °C) = 22 ns (max.) for LOW-to-HIGH transition at the output and 15 ns (max.) for HIGH-to-LOW output transition; worst-case noise margin = 0.4 V; fan-out = 10; I_{CCH} (for all four gates) = 8 mA; I_{CCL} (for all four gates) = 22 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 100 pJ; maximum flip-flop toggle frequency = 35 MHz.

5.3.2 Other Logic Gates in Standard TTL

As outlined earlier, the NAND gate is the fundamental building block of the TTL family. In the following paragraphs we will look at the internal schematics of the other logic gates and find for ourselves their similarity to the schematic of the NAND gate discussed in detail in earlier paragraphs.

5.3.2.1 NOT Gate (or Inverter)

Figure 5.8 shows the internal schematic of a NOT gate (inverter) in the standard TTL family. The schematic shown is that of one of the six inverters in a hex inverter (type 7404/5404). The internal schematic is just the same as that of the NAND gate except that the input transistor is a normal single emitter NPN transistor instead of a multi-emitter one. The circuit is self-explanatory.

5.3.2.2 NOR Gate

Figure 5.9 shows the internal schematic of a NOR gate in the standard TTL family. The schematic shown is that of one of the four NOR gates in a quad two-input NOR gate (type 7402/5402). On the input side there are two separate transistors instead of the multi-emitter transistor of the NAND gate. The inputs are fed to the emitters of the two transistors, the collectors of which again feed the bases of the two transistors with their collector and emitter terminals tied together. The resistance values used are the same as those used in the case of the NAND gate. The output stage is also the same totem-pole output stage. The circuit is self-explanatory. The only input condition for which transistors Q_3 and Q_4

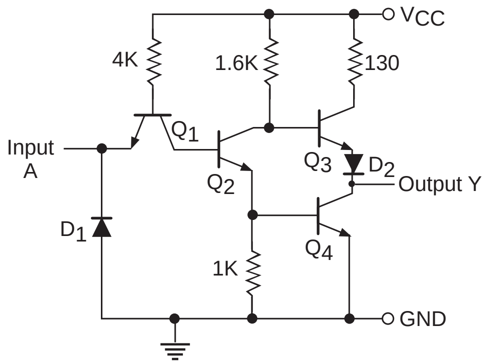


Figure 5.8 Inverter in the standard TTL.

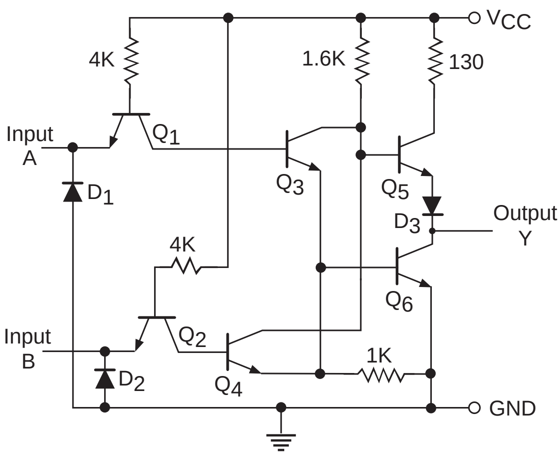


Figure 5.9 NOR gate in the standard TTL.

remain in cut-off, thus driving Q_6 to cut-off and Q_5 to conduction, is the one when both the inputs are in the logic LOW state. The output in such a case is logic HIGH. For all other input conditions, either Q_3 or Q_4 will conduct, driving Q_6 to saturation and Q_5 to cut-off, producing a logic LOW at the output.

5.3.2.3 AND Gate

Figure 5.10 shows the internal schematic of an AND gate in the standard TTL family. The schematic shown is that of one of the four AND gates in a quad two-input AND gate (type 7408/5408). In order to explain how this schematic arrangement behaves as an AND gate, we will begin by investigating the input condition that would lead to a HIGH output. A HIGH output implies Q_6 to be in cut-off and Q_5 to be in conduction. This can happen only when Q_4 is in cut-off. Transistor Q_4 can be in the cut-off

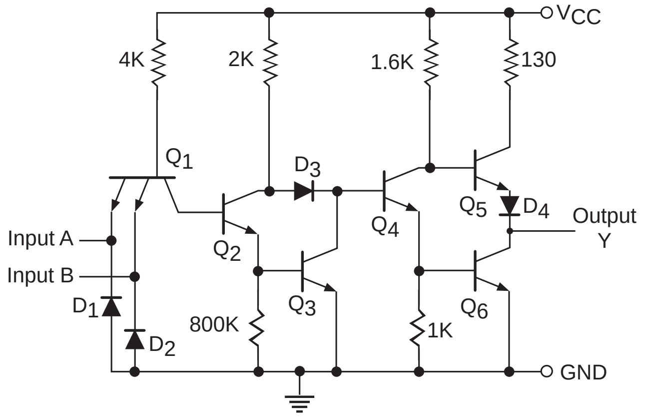


Figure 5.10 AND gate in standard TTL.

state only when both Q_2 and Q_3 are in conduction. This is possible only when both inputs are in the logic HIGH state. Let us now see what happens when either of the two inputs is driven to the LOW state. This drives Q_2 and Q_3 to the cut-off state, which forces Q_4 and subsequently Q_6 to saturation and Q_5 to cut-off.

5.3.2.4 OR Gate

Figure 5.11 shows the internal schematic of an OR gate in the standard TTL family. The schematic shown is that of one of the four OR gates in a quad two-input OR gate (type 7432/5432). We will begin by investigating the input condition that would lead to a LOW output. A LOW output demands a saturated Q_8 and a cut-off Q_7. This in turn requires Q_6 to be in saturation and Q_5, Q_4 and Q_3 to

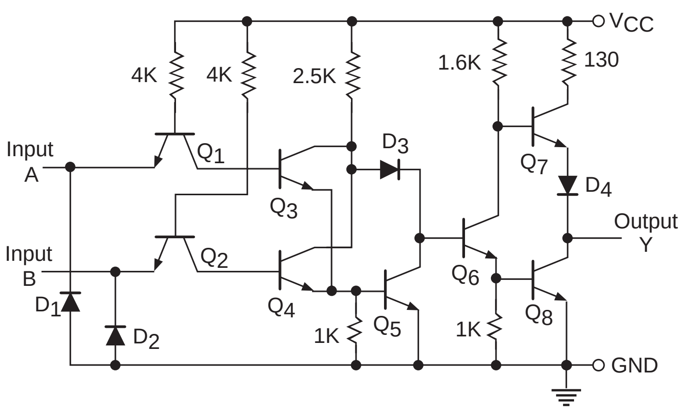


Figure 5.11 OR gate in the standard TTL.

be in cut-off. This is possible only when both Q_1 and Q_2 are in saturation. That is, both inputs are in the logic LOW state. This verifies one of the entries of the truth table of the OR gate. Let us now see what happens when either of the two inputs is driven to the HIGH state. This drives either of the two transistors Q_3 and Q_4 to saturation, which forces Q_5 to saturation and Q_6 to cut-off. This drives Q_7 to conduction and Q_8 to cut-off, producing a logic HIGH output.

5.3.2.5 EXCLUSIVE-OR Gate

Figure 5.12 shows the internal schematic of an EX-OR gate in the standard TTL family. The schematic shown is that of one of the four EX-OR gates in a quad two-input EX-OR gate (type 7486/5486). We will note the similarities between this circuit and that of an OR gate. The only new element is the interconnected pair of transistors Q_7 and Q_8. We will see that, when both the inputs are either HIGH or LOW, both Q_7 and Q_8 remain in cut-off. In the case of inputs being in the logic HIGH state, the base and emitter terminals of both these transistors remain near the ground potential. In the case of inputs being in the LOW state, the base and emitter terminals of both these transistors remain near V_{CC}. The result is conducting Q_9 and Q_{11} and nonconducting Q_{10}, which leads to a LOW output. When either of the inputs is HIGH, either Q_7 or Q_8 conducts. Transistor Q_7 conducts when input B is HIGH, and transistor Q_8 conducts when input A is HIGH. Conducting Q_7 or Q_8 turns off Q_9 and Q_{11} and turns on Q_{10}, producing a HIGH output. This explains how this circuit behaves as an EX-OR gate.

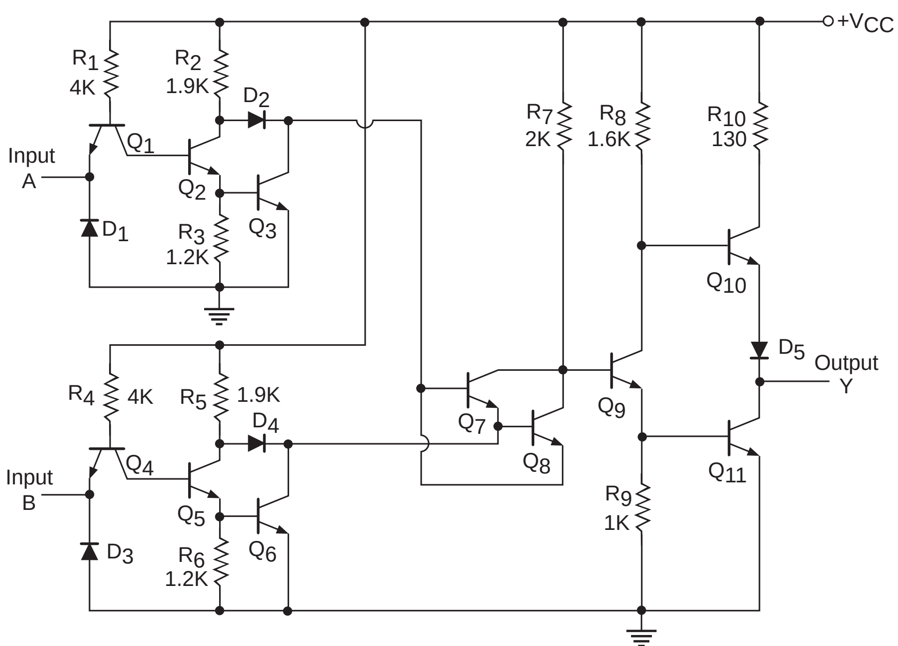


Figure 5.12 EX-OR gate in the standard TTL.

5.3.2.6 AND-OR-INVERT Gate

Figure 5.13 shows the internal schematic of a two-wide, two-input AND-OR-INVERT or AND-NOR gate. The schematic shown is that of one of the two gates in a dual two-wide, two-input AND-OR-INVERT gate (type 7450/5450). The two multi-emitter input transistors Q_1 and Q_2 provide ANDing of their respective inputs. Drive splitters comprising Q_3, Q_4, R_3 and R_4 provide the OR function. The output stage provides inversion. The number of emitters in each of the input transistors determines the number of literals in each of the minterms in the output sum-of-products Boolean expression. How wide the gate is going to be is decided by the number of input transistors, which also equals the number of drive splitter transistors.

5.3.2.7 Open Collector Gate

An open collector gate in TTL is one that is without a totem-pole output stage. The output stage in this case does not have the active pull-up transistor. An external pull-up resistor needs to be connected from the open collector terminal of the pull-down transistor to the V_{CC} terminal. The pull-up resistor is typically 10 kΩ. Figure 5.14 shows the internal schematic of a NAND gate with an open collector output. The schematic shown is that of one of the four gates of a quad two-input NAND (type 74/5401). The advantage of open collector outputs is that the outputs of different gates can be wired together, resulting in ANDing of their outputs. WIRE-AND operation was discussed in Chapter 4 on logic gates.

It may be mentioned here that the outputs of totem-pole TTL devices cannot be tied together. Although a common tied output may end up producing an ANDing of individual outputs, such a connection is impractical. This is illustrated in Fig. 5.15, where outputs of two totem-pole output TTL

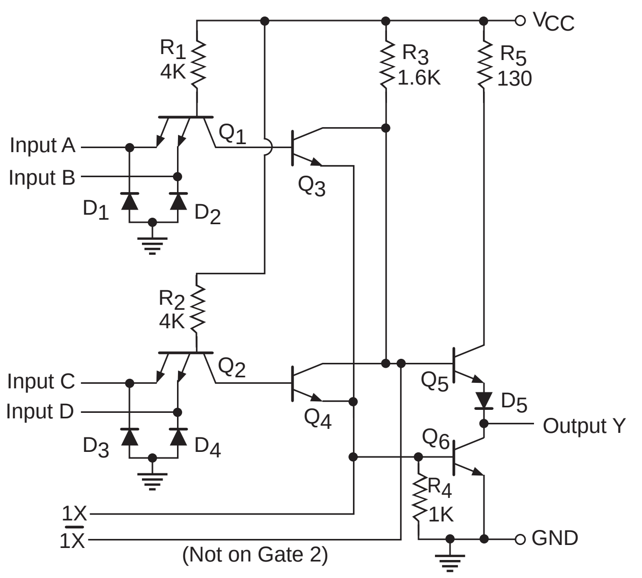


Figure 5.13 Two-input, two-wide AND-OR-INVERT gate.

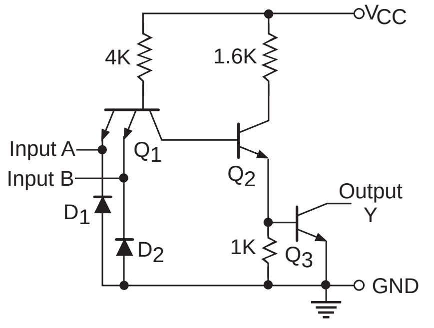


Figure 5.14 NAND gate with an open collector output.

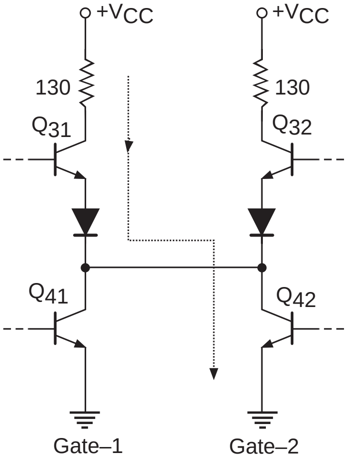


Figure 5.15 Totem-pole output gates tied at the output.

gates have been tied together. Let us assume that the output of one of the gates, say gate-2, is LOW, and the output of the other is HIGH. The result is that a relatively heavier current flows through Q_{31} and Q_{42}. This current, which is of the order of 50–60 mA, exceeds the I_{OL}(max.) rating of Q_{42}. This may eventually lead to both transistors getting damaged. Even if they survive, V_{OL}(max.) of Q_{42} is no longer guaranteed. In view of this, although totem-pole output TTL gates are not tied together, an accidental shorting of outputs is not ruled out. In such a case, both devices are likely to get damaged. In the case of open collector devices, deliberate or nondeliberate, shorting of outputs produces ANDing of outputs with no risk of either damage or compromised performance specifications.

5.3.2.8 Tristate Gate

Tristate gates were discussed in Chapter 4. A tristate gate has three output states, namely the logic LOW state, the logic HIGH state and the high-impedance state. An external enable input decides

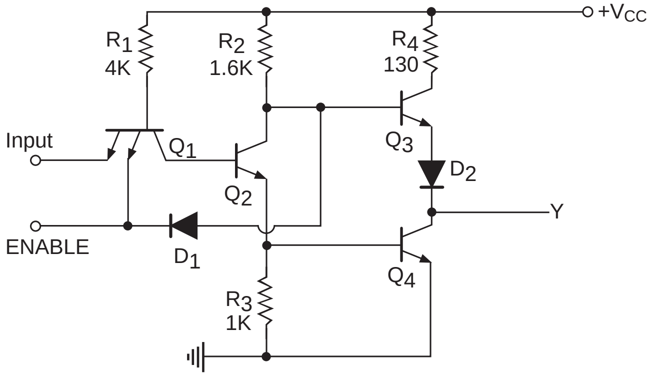


Figure 5.16 Tristate inverter in the TTL.

whether the logic gate works according to its truth table or is in the high-impedance state. Figure 5.16 shows the typical internal schematic of a tristate inverter with an active HIGH enable input. The circuit functions as follows. When the enable input is HIGH, it reverse-biases diode D_1 and also applies a logic HIGH on one of the emitters of the input transistor Q_1. The circuit behaves like an inverter. When the enable input is LOW, diode D_1 becomes forward biased. A LOW enable input forces Q_2 and Q_4 to cut-off. Also, a forward-biased D_1 forces Q_3 to cut-off. With both output transistors in cut-off, the output essentially is an open circuit and thus presents high output impedance.

5.3.3 Low-Power TTL

The low-power TTL is a low-power variant of the standard TTL where lower power dissipation is achieved at the expense of reduced speed of operation. Figure 5.17 shows the internal schematic of a

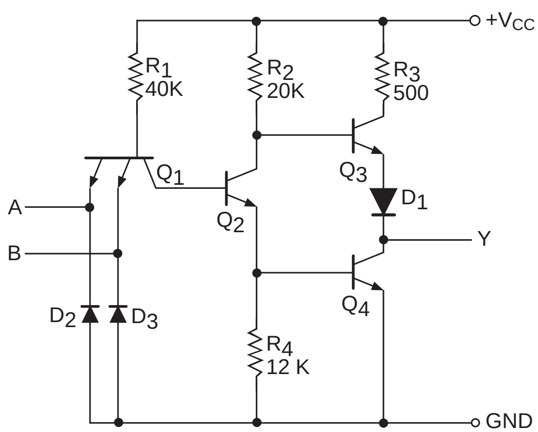


Figure 5.17 NAND gate in the low-power TTL.

low-power TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74L00 or 54L00). The circuit, as we can see, is the same as that of the standard TTL NAND gate except for an increased resistance value of the different resistors used in the circuit. Increased resistance values lead to lower power dissipation.

5.3.3.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2$ V; $V_{IL} = 0.7$ V; $I_{IH} = 10$ μA; $I_{IL} = 0.18$ mA; $V_{OH} = 2.4$ V; $V_{OL} = 0.4$ V; $I_{OH} = 200$ μA; $I_{OL} = 3.6$ mA; $V_{CC} = 4.75$–5.25 V (74-series) and 4.5–5.5 V (54-series); propagation delay (for a load resistance of 4000 Ω, a load capacitance of 50 pF, $V_{CC} = 5$ V and an ambient temperature of 25 °C) = 60 ns (max.) for both LOW-to-HIGH and HIGH-to-LOW output transitions; worst-case noise margin = 0.3 V; fan-out = 20; I_{CCH} (for all four gates) = 0.8 mA; I_{CCL} (for all four gates) = 2.04 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 33 pJ; maximum flip-flop toggle frequency = 3 MHz.

5.3.4 High-Power TTL (74H/54H)

The high-power TTL is a high-power, high-speed variant of the standard TTL where improved speed (reduced propagation delay) is achieved at the expense of higher power dissipation. Figure 5.18 shows the internal schematic of a high-power TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74H00 or 54H00). The circuit, as we can see, is nearly the same as that of the standard TTL NAND gate except for the transistor Q_3–diode D_1 combination in the totem-pole output stage having been replaced by a Darlington arrangement comprising Q_3, Q_5 and R_5. The Darlington arrangement does the same job as diode D_1 in the conventional totem-pole arrangement. It ensures that Q_5 does not conduct at all when the output is LOW. The decreased resistance values of different resistors used in the circuit lead to higher power dissipation.

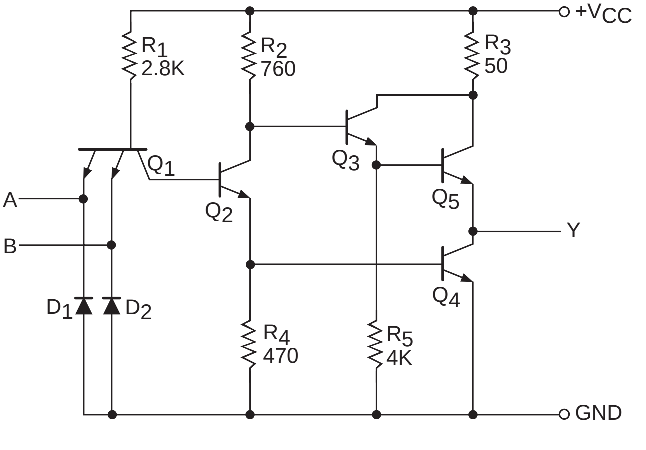


Figure 5.18 NAND gate in the high-power TTL.

5.3.4.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2$ V; $V_{IL} = 0.8$ V; $I_{IH} = 50\,\mu$A; $I_{IL} = 2$ mA; $V_{OH} = 2.4$ V; $V_{OL} = 0.4$ V; $I_{OH} = 500\,\mu$A; $I_{OL} = 20$ mA; $V_{CC} = 4.75$–5.25 V (74-series) and 4.5–5.5 V (54-series); propagation delay (for a load resistance of 280 Ω, a load capacitance of 25 pF, $V_{CC} = 5$ V and an ambient temperature of 25 °C) = 10 ns (max.) for both LOW-to-HIGH and HIGH-to-LOW output transitions; worst–case noise margin = 0.4 V; fan-out = 10; I_{CCH} (for all four gates) = 16.8 mA; I_{CCL} (for all four gates) = 40 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 132 pJ; maximum flip-flop frequency = 50 MHz.

5.3.5 Schottky TTL (74S/54S)

The Schottky TTL offers a speed that is about twice that offered by the high-power TTL for the same power consumption. Figure 5.19 shows the internal schematic of a Schottky TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74S00 or 54S00). The circuit, as we can see, is nearly the same as that of the high-power TTL NAND gate. The transistors used in the circuit are all Schottky transistors with the exception of Q_5. A Schottky Q_5 would serve no purpose, with Q_4 being a Schottky transistor. A Schottky transistor is nothing but a conventional bipolar transistor with a Schottky diode connected between its base and collector terminals. The Schottky diode with its metal–semiconductor junction not only is faster but also offers a lower forward voltage drop of 0.4 V as against 0.7 V for a P–N junction diode for the same value of forward current. The presence of a Schottky diode does not allow the transistor to go to deep saturation. The moment the collector voltage of the transistor tends to go below about 0.3 V, the Schottky diode becomes forward biased and bypasses part of the base current through it. The collector voltage is thus not allowed to go to the saturation value of 0.1 V and gets clamped around 0.3 V. While the power consumption of a Schottky TTL gate is almost the same as that of a high-power TTL gate owing to nearly the same values of the resistors used in the circuit, the Schottky TTL offers a higher speed on account of the use of Schottky transistors.

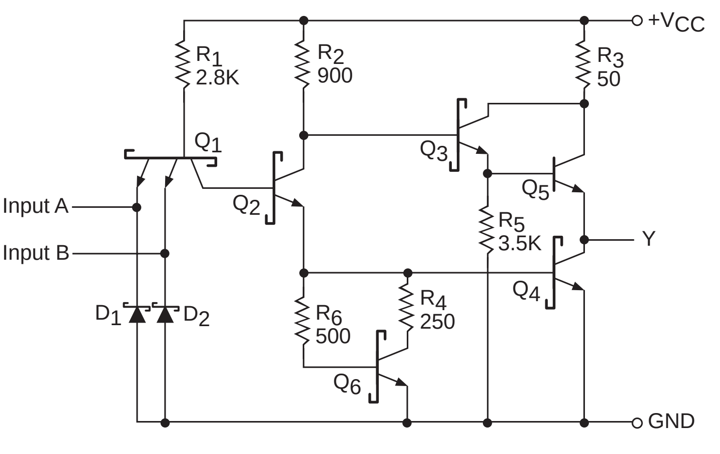


Figure 5.19 NAND gate in the Schottky TTL.

5.3.5.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2$ V; $V_{IL} = 0.8$ V; $I_{IH} = 50\,\mu$A; $I_{IL} = 2$ mA; $V_{OH} = 2.7$ V; $V_{OL} = 0.5$ V; $I_{OH} = 1$ mA; $I_{OL} = 20$ mA; $V_{CC} = 4.75$–5.25 V (74-series) and 4.5–5.5 V (54-series); propagation delay (for a load resistance of 280 Ω, a load capacitance of 15 pF, $V_{CC} = 5$ V and an ambient temperature of 25 °C) = 5 ns (max.) for LOW-to-HIGH and 4.5 ns (max.) for HIGH-to-LOW output transitions; worst-case noise margin = 0.3 V; fan-out = 10; I_{CCH} (for all four gates) = 16 mA; I_{CCL} (for all four gates) = 36 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 57 pJ; maximum flip-flop toggle frequency = 125 MHz.

5.3.6 Low-Power Schottky TTL (74LS/54LS)

The low-power Schottky TTL is a low power consumption variant of the Schottky TTL. Figure 5.20 shows the internal schematic of a low-power Schottky TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74LS00 or 54LS00). We can notice the significantly increased value of resistors R_1 and R_2 used to achieve lower power consumption. Lower power consumption, of course, occurs at the expense of reduced speed or increased propagation delay. Resistors R_3 and R_5, which primarily affect speed, have not been increased in the same proportion with respect to the corresponding values used in the Schottky TTL as resistors R_1 and R_2. That is why, although the low-power Schottky TTL draws an average maximum supply current of 3 mA (for all four gates) as against 26 mA for the Schottky TTL, the propagation delay is 15 ns in LS-TTL as against 5 ns for S-TTL. Diodes D_3 and D_4 reduce the HIGH-to-LOW propagation delay. While D_3 speeds up the turn-off of Q_4, D_4 sinks current from the load. Another noticeable difference in the internal schematics of the low-power Schottky TTL NAND and Schottky TTL NAND is the replacement of the

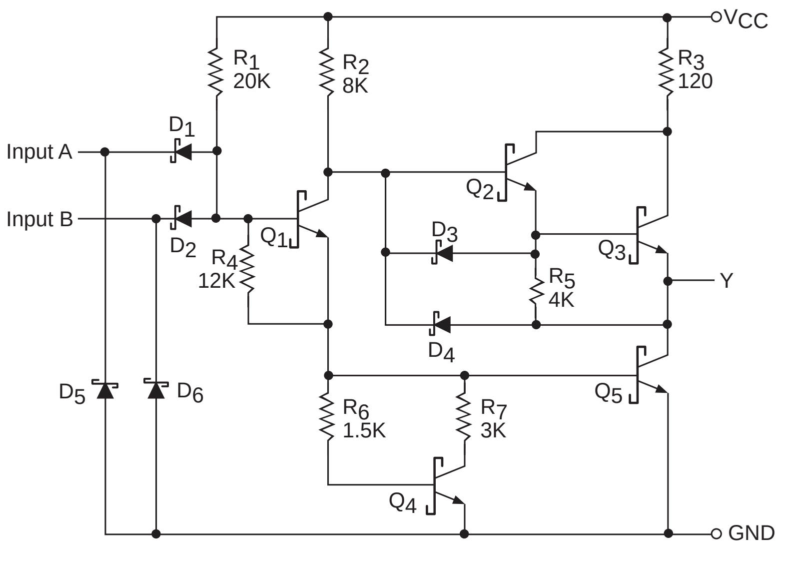


Figure 5.20 NAND gate in the low-power Schottky TTL.

multi-emitter input transistor of the Schottky TTL by diodes D_1 and D_2 and resistor R_1. The junction diodes basically replace the two emitter-base junctions of the multi-emitter input transistor Q_1 of the Schottky TTL NAND (Fig. 5.19). The reason for doing so is that Schottky diodes can be made smaller than the transistor and therefore will have lower parasitic capacitances. Also, since Q_1 of LS-TTL (Fig. 5.20) cannot saturate, it is not necessary to remove its base charge with a bipolar junction transistor.

5.3.6.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2\,V$; $V_{IL} = 0.8\,V$; $I_{IH} = 20\,\mu A$; $I_{IL} = 0.4\,mA$; $V_{OH} = 2.7\,V$; $V_{OL} = 0.5\,V$; $I_{OH} = 0.4\,mA$; $I_{OL} = 8\,mA$; $V_{CC} = 4.75$–$5.25\,V$ (74-series) and 4.5–5.5 V (54-series); propagation delay (for a load resistance of 280 Ω, a load capacitance of 15 pF, $V_{CC} = 5\,V$ and an ambient temperature of 25 °C) = 15 ns (max.) for both LOW-to-HIGH and HIGH-to-LOW output transitions; worst-case noise margin = 0.3 V; fan-out = 20; I_{CCH} (for all four gates) = 1.6 mA; I_{CCL} (for all four gates) = 4.4 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 18 pJ; maximum flip-flop toggle frequency = 45 MHz.

5.3.7 Advanced Low-Power Schottky TTL (74ALS/54ALS)

The basic ideas behind the development of the advanced low-power Schottky TTL (ALS-TTL) and advanced Schottky TTL (AS-TTL) discussed in Section 5.3.8 were further to improve both speed and power consumption performance of the low-power Schottky TTL and Schottky TTL families respectively. In the TTL subfamilies discussed so far, we have seen that different subfamilies achieved improved speed at the expense of increased power consumption, or vice versa. For example, the low-power TTL offered lower power consumption over standard TTL at the cost of reduced speed. The high-power TTL, on the other hand, offered improved speed over the standard TTL at the expense of increased power consumption. ALS-TTL and AS-TTL incorporate certain new circuit design features and fabrication technologies to achieve improvement of both parameters. Both ALS-TTL and AS-TTL offer an improvement in speed–power product respectively over LS-TTL and S-TTL by a factor of 4. Salient features of ALS-TTL and AS-TTL include the following:

1. All saturating transistors are clamped by using Schottky diodes. This virtually eliminates the storage of excessive base charge, thus significantly reducing the turn-off time of the transistors. Elimination of transistor storage time also provides stable switching times over the entire operational temperature range.
2. Inputs and outputs are clamped by Schottky diodes to limit the negative-going excursions.
3. Both ALS-TTL and AS-TTL use ion implantation rather than a diffusion process, which allows the use of small geometries leading to smaller parasitic capacitances and hence reduced switching times.
4. Both ALS-TTL and AS-TTL use oxide isolation rather than junction isolation between transistors. This leads to reduced epitaxial layer–substrate capacitance, which further reduces the switching times.
5. Both ALS-TTL and AS-TTL offer improved input threshold voltage and reduced low-level input current.
6. Both ALS-TTL and AS-TTL feature active turn-off of the LOW-level output transistor, producing a better HIGH-level output voltage and thus a higher HIGH-level noise immunity.

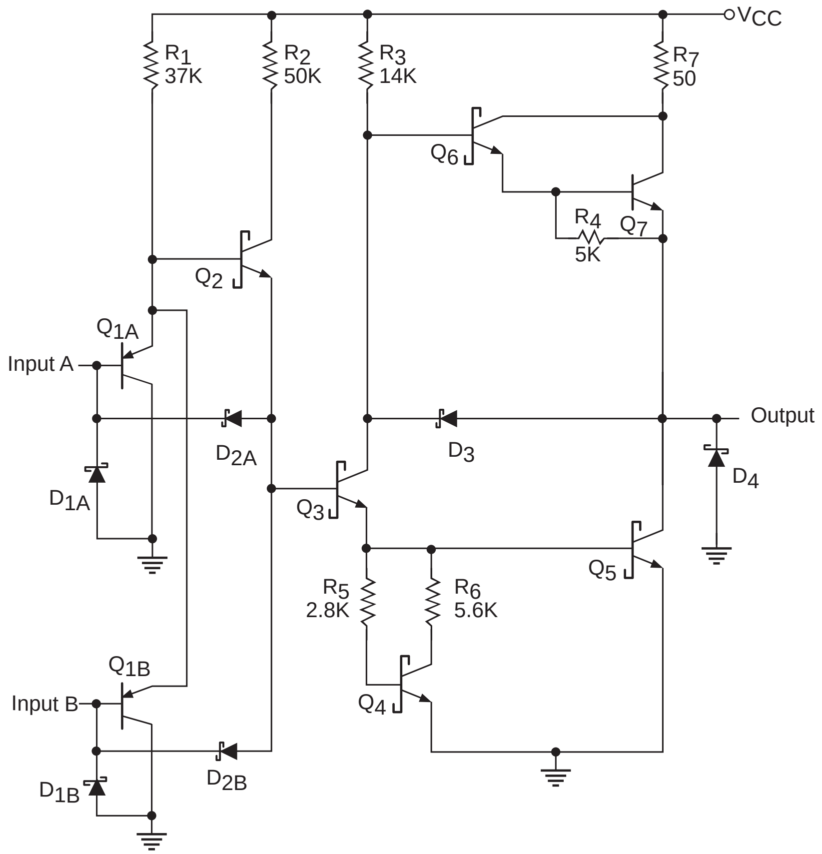


Figure 5.21 NAND gate in the ALS-TTL.

Figure 5.21 shows the internal schematic of an advanced low-power Schottky TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74ALS00 or 54ALS00) The multi-emitter input transistor is replaced by two PNP transistors Q_{1A} and Q_{1B}. Diodes D_{1A} and D_{1B} provide input clamping to negative excursions. Buffering offered by Q_{1A} or Q_{1B} and Q_2 reduces the LOW-level input current by a factor of (1 + h_{FE} of Q_{1A}). HIGH-level output voltage is determined primarily by V_{CC}, transistors Q_6 and Q_7 and resistors R_4 and R_7 and is typically $(V_{CC} - 2)$ V. LOW-level output voltage is determined by the turn-on characteristics of Q_5. Transistor Q_5 gets sufficient base drive through R_3 and a conducting Q_3 whose base terminal in turn is driven by a conducting Q_2 whenever either or both inputs are HIGH. Transistor Q_4 provides active turn-off for Q_5.

5.3.7.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2$ V; $V_{IL} = 0.8$ V; $I_{IH} = 20\,\mu$A; $I_{IL} = 0.1$ mA; $V_{OH} = (V_{CC} - 2)$ V; $V_{OL} = 0.5$ V; $I_{OH} = 0.4$ mA; $I_{OL} = 8$ mA (74ALS) and 4 mA (54ALS);

$V_{CC} = 4.5–5.5$ V; propagation delay (for a load resistance of 500 Ω, a load capacitance of 50 pF, $V_{CC} = 4.5–5.5$ V and an ambient temperature of minimum to maximum) = 11 ns/16 ns (max.) for LOW-to-HIGH and 8 ns/13 ns for HIGH-to-LOW output transitions (74ALS/54ALS); worst-case noise margin = 0.3 V; fan-out = 20; I_{CCH} (for all four gates) = 0.85 mA; I_{CCL} (for all four gates) = 3 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 4.8 pJ; maximum flip-flop toggle frequency = 70 MHz.

5.3.8 Advanced Schottky TTL (74AS/54AS)

Figure 5.22 shows the internal schematic of an advanced Schottky TTL NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74AS00 or 54AS00). Salient

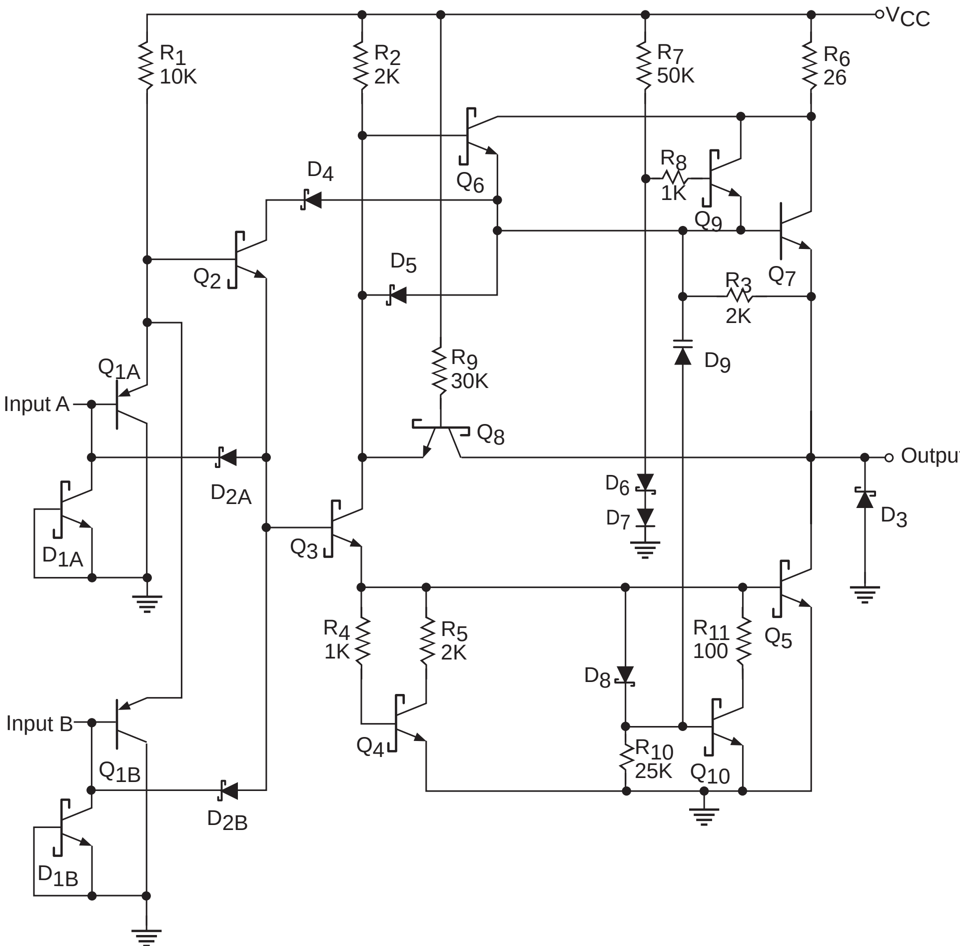


Figure 5.22 NAND gate in the AS-TTL.

features of ALS-TTL and AS-TTL have been discussed at length in the preceding paragraphs. As is obvious from the internal circuit schematic of the AS-TTL NAND gate, it has some additional circuits not found in ALS-TTL devices. These are added to enhance the throughput of AS-TTL family devices. Transistor Q_{10} provides a discharge path for the base-collector capacitance of Q_5. In the absence of Q_{10}, a rising voltage across the output forces current into the base of Q_5 through its base-collector capacitance, thus causing it to turn on. Transistor Q_{10} turns on through D_9, thus keeping transistor Q_5 in the cut-off state.

5.3.8.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2\,V$; $V_{IL} = 0.8\,V$; $I_{IH} = 20\,\mu A$; $I_{IL} = 0.5\,mA$; $V_{OH} = (V_{CC} - 2)\,V$; $V_{OL} = 0.5\,V$; $I_{OH} = 2\,mA$; $I_{OL} = 20\,mA$; $V_{CC} = 4.5$–$5.5\,V$; propagation delay (for a load resistance of $50\,\Omega$, a load capacitance of 50 pF, $V_{CC} = 4.5$–$5.5\,V$ and an ambient temperature of minimum to maximum) = 4.5 ns/5 ns (max.) for LOW-to-HIGH and 4 ns/5 ns (max.) for HIGH-to-LOW output transitions (74AS/54AS); worst-case noise margin = 0.3 V; fan-out = 40; I_{CCH} (for all four gates) = 3.2 mA; I_{CCL} (for all four gates) = 17.4 mA; operating temperature range = 0–70 °C (74-series) and −55 to +125 °C (54-series); speed–power product = 13.6 pJ; maximum flip-flop toggle frequency = 200 MHz.

5.3.9 Fairchild Advanced Schottky TTL (74F/54F)

The Fairchild Advanced Schottky TTL family, commonly known as FAST logic, is similar to the AS-TTL family. Figure 5.23 shows the internal schematic of a Fairchild Advanced Schottky TTL

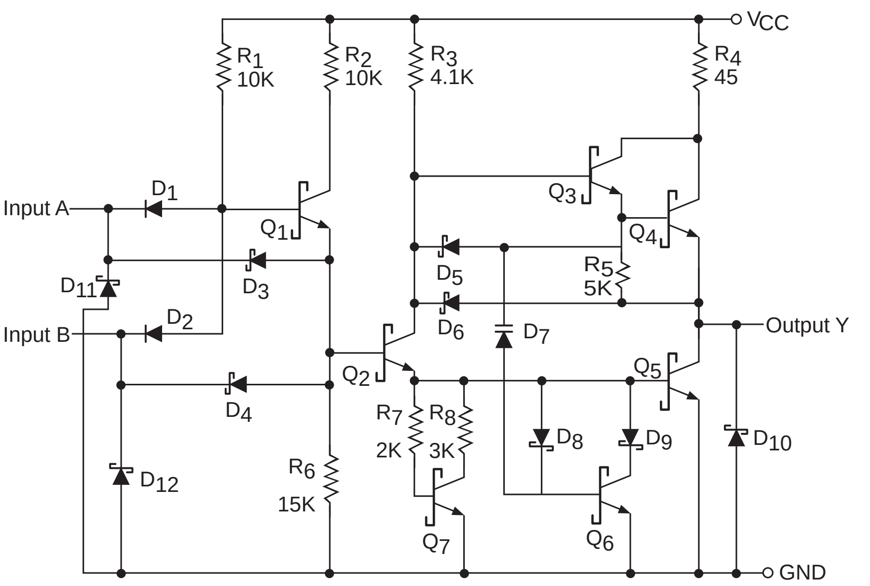


Figure 5.23 NAND gate in the FAST TTL.

NAND gate. The circuit shown is that of one of the four gates inside a quad two-input NAND (type 74F00 or 54F00). The DTL kind of input with emitter follower configuration of Q_1 provides a good base drive to Q_2. The 'Miller killer' configuration comprising varactor diode D_7, transistor Q_6 and associated components speeds up LOW-to-HIGH transition. During LOW-to-HIGH transition, voltage at the emitter terminal of Q_3 begins to rise while Q_5 is still conducting. Varactor diode D_7 conducts, thus supplying base current to Q_6. A conducting Q_6 provides a discharge path for the charge stored in the base-collector capacitance of Q_5, thus expediting its turn-off.

5.3.9.1 Characteristic Features

Characteristic features of this family are summarized as follows: $V_{IH} = 2$ V; $V_{IL} = 0.8$ V; $I_{IH} = 20$ µA; $I_{IL} = 0.6$ mA; $V_{OH} = 2.7$ V; $V_{OL} = 0.5$ V; $I_{OH} = 1$ mA; $I_{OL} = 20$ mA; $V_{CC} = 4.75$–5.25 V (74F) and 4.5–5.5 V (54F); propagation delay (a load resistance of 500 Ω, a load capacitance of 50 pF and full operating voltage and temperature ranges) = 5.3 ns/7 ns (max.) for LOW-to-HIGH and 6 ns/6.5 ns (max.) for HIGH-to-LOW output transitions (74AS/54AS); worst-case noise margin = 0.3 V; fan-out = 40; I_{CCH} (for all four gates) = 2.8 mA; I_{CCL} (for all four gates) = 10.2 mA; operating temperature range = 0–70 °C (74F-series) and −55 to +125 °C (54F-series); speed–power product = 10 pJ; maximum flip-flop toggle frequency = 125 MHz.

5.3.10 Floating and Unused Inputs

The floating input of TTL family devices behaves as if logic HIGH has been applied to the input. Such behaviour is explained from the input circuit of a TTL device. When the input is HIGH, the input emitter-base junction is reverse biased and the current that flows into the input is the reverse-biased diode leakage current. The input diode will be reverse biased even when the input terminal is left unconnected or floating, which implies that a floating input behaves as if there were logic HIGH applied to it.

As an initial thought, we may tend to believe that it should not make any difference if we leave the unused inputs of NAND and AND gates as floating, as logic HIGH like behaviour of the floating input makes no difference to the logical behaviour of the gate, as shown in Figs 5.24(a) and (b). In spite of this, it is strongly recommended that the unused inputs of AND and NAND gates be connected to a logic HIGH input [Fig. 5.24(c)] because floating input behaves as an antenna and may pick up stray noise and interference signals, thus causing the gate to function improperly. 1 kΩ resistance is connected to protect the input from any current spikes caused by any spikes on the power supply line. More than one unused input (up to 50) can share the same 1 kΩ resistance, if needed.

In the case of OR and NOR gates, unused inputs are connected to ground (logic LOW), as shown in Fig. 5.25(c), for obvious reasons. A floating input or an input tied to logic HIGH in this case produces a permanent logic HIGH (for an OR gate) and LOW (for a NOR gate) at the output as shown in Figs 5.25(a) and (b) respectively. An alternative solution is shown in Fig. 5.25(d), where the unused input has been tied to one of the used inputs. This solution works well for all gates, but one has to be conscious of the fact that the fan-out capability of the output driving the tied inputs is not exceeded.

If we recall the internal circuit schematics of AND and NAND gates, we will appreciate that, when more than one input is tied together, the input loading, that is, the current drawn by the tied inputs from the driving gate output, in the HIGH state is n times the loading of one input (Fig. 5.26); n is the number of inputs tied together. When the output is LOW, the input loading is the same as that of a single input. The reason for this is that, in the LOW input state, the current flowing out of the gate is determined by the resistance R_1, as shown in Fig. 5.27. However, the same is not true in the case of

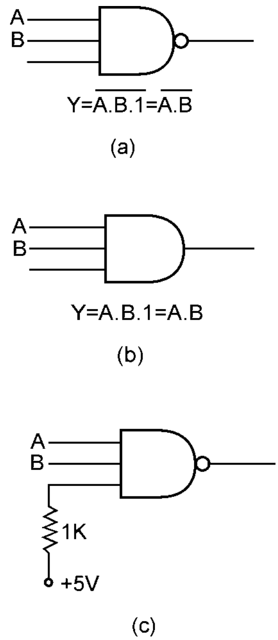


Figure 5.24 Handling unused inputs of AND and NAND gates.

OR and NOR gates, which do not use a multi-emitter input transistor and use separate input transistors instead, as shown in Fig. 5.28. In this case, the input loading is n times the loading of a single input for both HIGH and LOW states.

5.3.11 Current Transients and Power Supply Decoupling

TTL family devices are prone to occurrence of narrow-width current spikes on the power supply line. Current transients are produced when the totem-pole output stage of the device undergoes a transition from a logic LOW to a logic HIGH state. The problem becomes severe when in a digital circuit a large number of gates are likely to switch states at the same time. These current spikes produce voltage spikes due to any stray inductance present on the line. On account of the large rate of change in current in the current spike, even a small value of stray inductance produces voltage spikes large enough adversely to affect the circuit performance.

Figure 5.29 illustrates the phenomenon. When the output changes from LOW to HIGH, there is a small fraction of time when both the transistors are conducting because the pull-up transistor Q_3 has switched on and the pull-down transistor Q_4 has not yet come out of saturation. During this small fraction of time, there is an increase in current drawn from the supply; I_{CCL} experiences a positive spike before it settles down to a usually lower I_{CCH}. The presence of any stray capacitance C across the output owing to any stray wiring capacitance or capacitance loading of the circuit being fed also adds to the problem. The problem of voltage spikes on the power supply line is usually overcome by connecting small-value, low-inductance, high-frequency capacitors between V_{CC} terminal and ground. It is standard practice to use a 0.01 or 0.1 μF ceramic capacitor from V_{CC} to ground. This

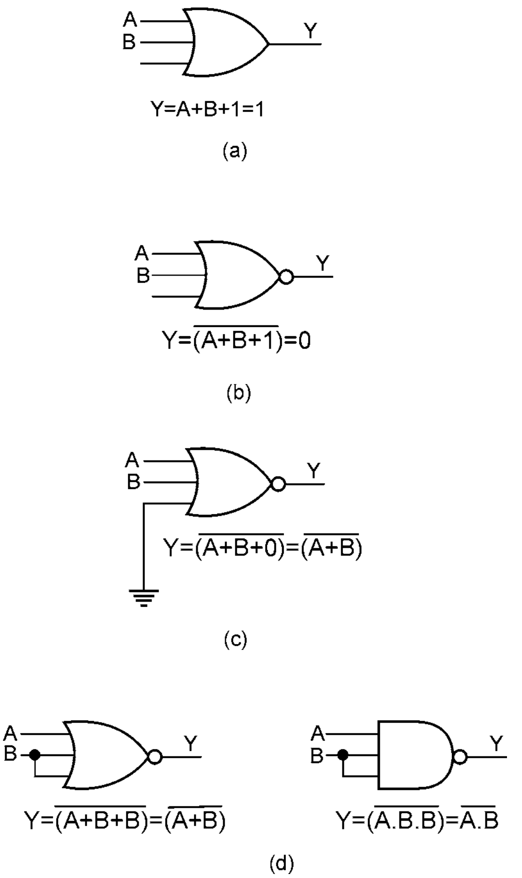


Figure 5.25 Handling unused inputs of OR and NOR gates.

capacitor is also known by the name of power supply decoupling capacitor, and it is recommended to use a separate capacitor for each IC. A decoupling capacitor is connected as close to the V_{CC} terminal as possible, and its leads are kept to a bare minimum to minimize lead inductance. In addition, a single relatively large-value capacitor in the range of 1–22 μF is also connected between V_{CC} and ground on each circuit card to take care of any low-frequency voltage fluctuations in the power supply line.

Example 5.5

Refer to Fig. 5.30. Determine the current being sourced by gate 1 when its output is HIGH and sunk by it when its output is LOW. All gates are from the standard TTL family, given that $I_{IH} = 40\,\mu A$ and $I_{IL} = 1.6\,mA$.

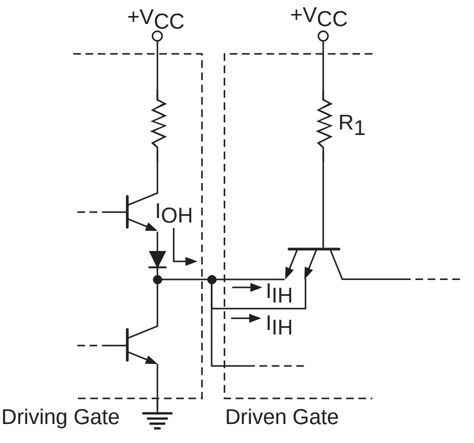


Figure 5.26 Input loading in the case of HIGH tied inputs of NAND and AND gates.

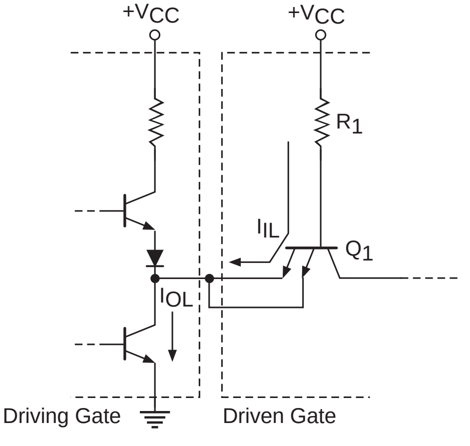


Figure 5.27 Input loading in the case of LOW tied inputs of NAND and AND gates.

Solution

- When the output is HIGH, the inputs of all gates draw current individually.
- Therefore, the input loading factor = equivalent of seven gate inputs = $7 \times 40\,\mu\text{A} = 280\,\mu\text{A}$.
- The current being sourced by the gate 1 output = $280\,\mu\text{A}$.
- When the output is LOW, shorted inputs of AND and NAND gates offer a load equal to that of a single input owing to a multi-emitter transistor at the input of the gate. The inputs of OR and NOR gates draw current individually on account of the use of separate transistors at the input of the gate.
- Therefore, the input loading factor = equivalent of five gate inputs = $5 \times 1.6 = 8$ mA.
- The current being sunk by the gate 1 output = 8 mA.

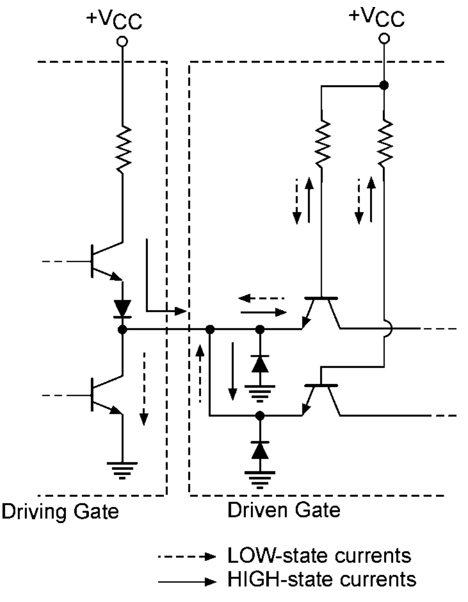


Figure 5.28 Input loading in the case of tied inputs of NOR and OR gates.

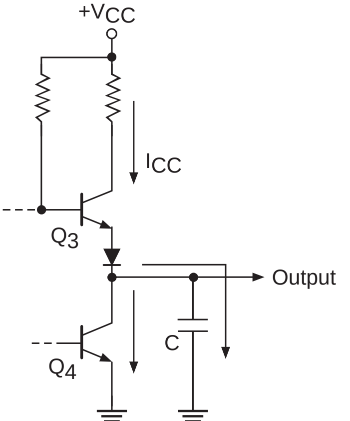


Figure 5.29 Current transients and power supply decoupling.

Example 5.6

Refer to the logic diagram of Fig. 5.31. Gate 1 and gate 4 belong to the standard TTL family, while gate 2 and gate 3 belong to the Schottky TTL family and the low-power Schottky TTL family respectively. Determine whether the fan-out capability of gate 1 is being exceeded. Relevant data for the three logic families are given in Table 5.1.

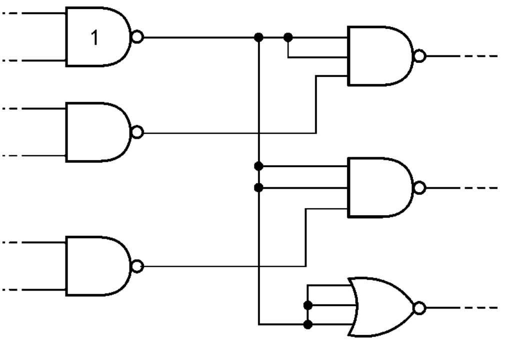


Figure 5.30 Example 5.5.

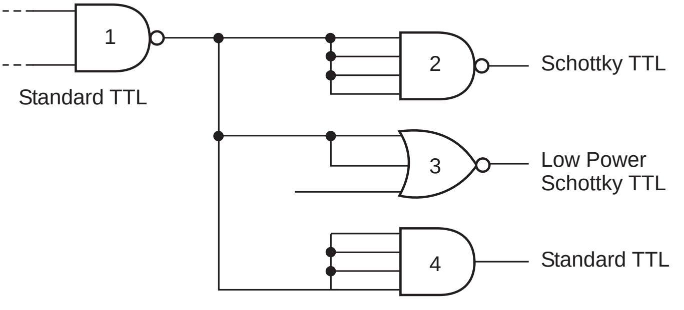


Figure 5.31 Example 5.6.

Table 5.1 Example 5.6

Logic family	$I_{IH}(\mu A)$	$I_{OH}(mA)$	$I_{IL}(mA)$	$I_{OL}(mA)$
Standard TTL	40	0.4	1.6	16
LS-TTL	20	0.4	0.4	8.0
S-TTL	50	1.0	2.0	20

Solution

- In the HIGH-state:

  – the gate 1 output sourcing capability = 400 μA;
  – the gate 2 input requirement = 50 × 4 = 200 μA;
  – the gate 3 input requirement = 20 × 2 = 40 μA;
  – the gate 4 input requirement = 40 × 4 = 160 μA;
  – the total input current requirement = 400 μA;
  – therefore, the fan-out is not exceeded in the HIGH state.

- In the LOW-state,

  – the gate 1 output sinking capability = 16 mA;
  – the gate 2 input sinking requirement = 2 mA;
  – the gate 3 input sinking requirement = 0.4 × 2 = 0.8 mA;
  – the gate 4 input sinking requirement = 1.6 mA;
  – the total input current requirement = 4.4 mA;
  – since the output of gate 1 has a current sinking capability of 16 mA, the fan-out capability is not exceeded in the LOW state either.

5.4 Emitter Coupled Logic (ECL)

The ECL family is the fastest logic family in the group of bipolar logic families. The characteristic features that give this logic family its high speed or short propagation delay are outlined as follows:

1. It is a nonsaturating logic. That is, the transistors in this logic are always operated in the active region of their output characteristics. They are never driven to either cut-off or saturation, which means that logic LOW and HIGH states correspond to different states of conduction of various bipolar transistors.
2. The logic swing, that is, the difference in the voltage levels corresponding to logic LOW and HIGH states, is kept small (typically 0.85 V), with the result that the output capacitance needs to be charged and discharged by a relatively much smaller voltage differential.
3. The circuit currents are relatively high and the output impedance is low, with the result that the output capacitance can be charged and discharged quickly.

5.4.1 Different Subfamilies

Different subfamilies of ECL logic include MECL-I, MECL-II, MECL-III, MECL 10K, MECL 10H and MECL 10E (ECLinPS™and ECLinPS Lite™).

5.4.1.1 MECL-I, MECL-II and MECL-III Series

MECL-I was the first monolithic emitter coupled logic family introduced by ON Semiconductor (formerly a division of Motorola SPS) in 1962. It was subsequently followed up by MECL-II in 1966. Both these logic families have become obsolete and have been replaced by MECL-III (also called the MC1600 series) introduced in 1968. Although, chronologically, MECL-III was introduced before the MECL-10K and MECL-10H families, it features higher speed than both of its successors. With a propagation delay of the order of 1 ns and a flip-flop toggle frequency of 500 MHz, MECL-III is used in high-performance, high-speed systems.

The basic characteristic parameters of MECL-III are as follows: gate propagation delay = 1 ns; output edge speed (indicative of the rise and fall time of output transition) = 1 ns; flip-flop toggle frequency = 500 MHz; power dissipation per gate = 50 mW; speed–power product = 60 pJ; input voltage = 0–V_{EE} (V_{EE} is the negative supply voltage); negative power supply range (for $V_{CC} = 0$) = −5.1V to −5.3 V; continuous output source current (max.) = 40 mA; surge output source current (max.) = 80 mA; operating temperature range = −30 °C to +85 °C.

5.4.1.2 MECL-10K Series

The MECL-10K family was introduced in 1971 to meet the requirements of more general-purpose high-speed applications. Another important feature of MECL-10K family devices is that they are compatible with MECL-III devices, which facilitates the use of devices of the two families in the same system. The increased propagation delay of 2 ns in the case of MECL-10K comes with the advantage of reduced power dissipation, which is less than half the power dissipation in MECL-III family devices.

The basic characteristic parameters of MECL-10K are as follows: gate propagation delay = 2 ns (10100-series) and 1.5 ns (10200-series); output edge speed = 3.5 ns (10100-series) and 2.5 ns (10200-series); flip-flop toggle frequency = 125 MHz (min.) in the 10100-series and 200 MHz (min.) in the 10200-series; power dissipation per gate = 25 mW; speed–power product = 50 pJ (10100-series) and 37 pJ (10200-series); input voltage = 0–V_{EE} (V_{EE} is the negative supply voltage); negative power supply range (for $V_{CC} = 0$) = −4.68 to −5.72 V; continuous output source current (max.) = 50 mA; surge output source current (max.) = 100 mA; operating temperature range = −30 °C to +85 °C.

5.4.1.3 MECL-10H Series

The MECL-10H family, introduced in 1981, combines the high speed advantage of MECL-III with the lower power dissipation of MECL-10K. That is, it offers the speed of MECL-III with the power economy of MECL-10K. Backed by a propagation delay of 1 ns and a power dissipation of 25 mW per gate, MECL-10H offers one of the best speed–power product specifications in all available ECL subfamilies. Another important aspect of this family is that many of the MECL-10H devices are pin-out/functional replacements of MECL-10K series devices, which allows the users or the designers to enhance the performance of existing systems by increasing speed in critical timing areas.

The basic characteristic parameters of MECL-10H are as follows: gate propagation delay = 1 ns; output edge speed = 1 ns; flip-flop toggle frequency = 250 MHz (min.); power dissipation per gate = 25 mW; speed–power product = 25 pJ; input voltage = 0–V_{EE} (V_{EE} is the negative supply voltage); negative power supply range (for $V_{CC} = 0$) = −4.94 to −5.46 V; continuous output source current (max.) = 50 mA; surge output source current (max.) = 100 mA; operating temperature range = 0 °C to + 75 °C.

5.4.1.4 MECL-10E Series (ECLinPS™ and ECLinPSLite™)

The ECLinPS™ family, introduced in 1987, has a propagation delay of the order of 0.5 ns. ECLinPSLite™ is a recent addition to the ECL family. It offers a propagation delay of the order of 0.2 ns. The ECLPro™ family of devices is a rapidly growing line of high-performance ECL logic, offering a significant speed upgrade compared with the ECLinPSLite™ devices.

5.4.2 Logic Gate Implementation in ECL

OR/NOR is the fundamental logic gate of the ECL family. Figure 5.32 shows a typical internal schematic of an OR/NOR gate in the 10K-series MECL family. The circuit in essence comprises a differential amplifier input circuit with one side of the differential pair having multiple transistors depending upon the number of inputs to the gate, a voltage- and temperature-compensated bias network and emitter follower outputs. The internal schematic of the 10H-series gate is similar, except that the bias network is replaced with a voltage regulator circuit and the source resistor R_{EE} of the differential amplifier is replaced with a constant current source. Typical values of power supply voltages are

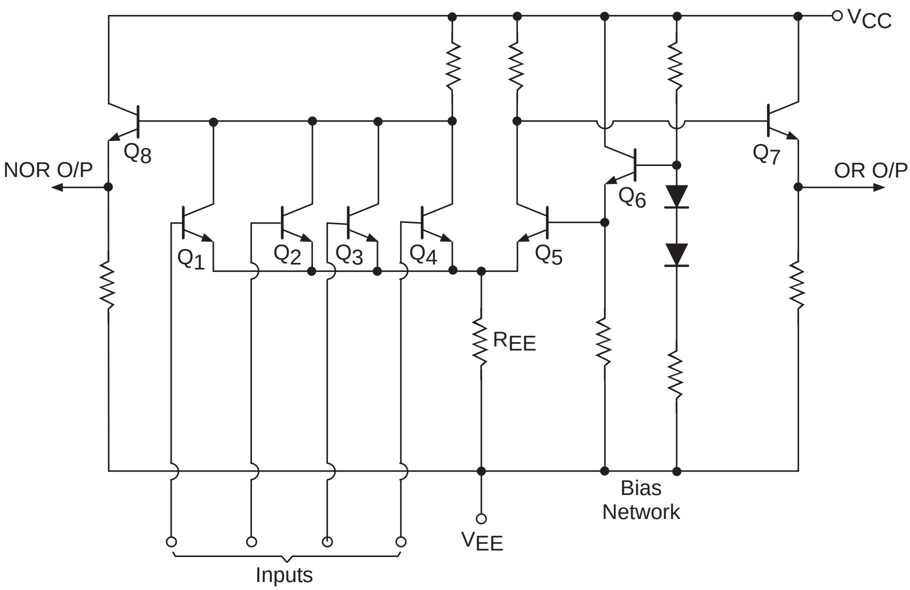


Figure 5.32 OR/NOR in ECL.

$V_{CC} = 0$ and $V_{EE}=-5.2$ V. The nominal logic levels are logic LOW = logic '0' = −1.75 V and logic HIGH = logic '1' = −0.9 V, assuming a positive logic system. The circuit functions as follows.

The bias network configured around transistor Q_6 produces a voltage of typically −1.29 V at its emitter terminal. This leads to a voltage of −2.09 V at the junction of all emitter terminals of various transistors in the differential amplifier, assuming 0.8 V to be the required forward-biased P–N junction voltage. Now, let us assume that all inputs are in a logic '0' state, that is, the voltage at the base terminals of various input transistors is −1.75 V. This means that the transistors Q_1, Q_2, Q_3 and Q_4 will remain in cut-off as their base-emitter junctions are not forward biased by the required voltage. This leads us to say that transistor Q_7 is conducting, producing a logic '0' output, and transistor Q_8 is in cut-off, producing a logic '1' output.

In the next step, let us see what happens if any one or all of the inputs are driven to logic '1' status, that is, a nominal voltage of −0.9 V is applied to the inputs. The base-emitter voltage differential of transistors Q_1–Q_4 exceeds the required forward-biasing threshold, with the result that these transistors start conducting. This leads to a rise in voltage at the common-emitter terminal, which now becomes approximately −1.7 V as the common-emitter terminal is now 0.8 V more negative than the base-terminal voltage. With rise in the common-emitter terminal voltage, the base-emitter differential voltage of Q_5 becomes 0.31 V, driving Q_5 to cut-off. The Q_7 and Q_8 emitter terminals respectively go to logic '1' and logic '0'.

This explains how this basic schematic functions as an OR/NOR gate. We will note that the differential action of the switching transistors (where one section is ON while the other is OFF) leads to simultaneous availability of complementary signals at the output. Figure 5.33 shows the circuit symbol and switching characteristics of this basic ECL gate. It may be mentioned here that positive ECL (called PECL) devices operating at +5 V and ground are also available. When used in PECL mode, ECL devices must have their input/output DC parameters adjusted for proper operation. PECL DC parameters can be computed by adding ECL levels to the new V_{CC}.

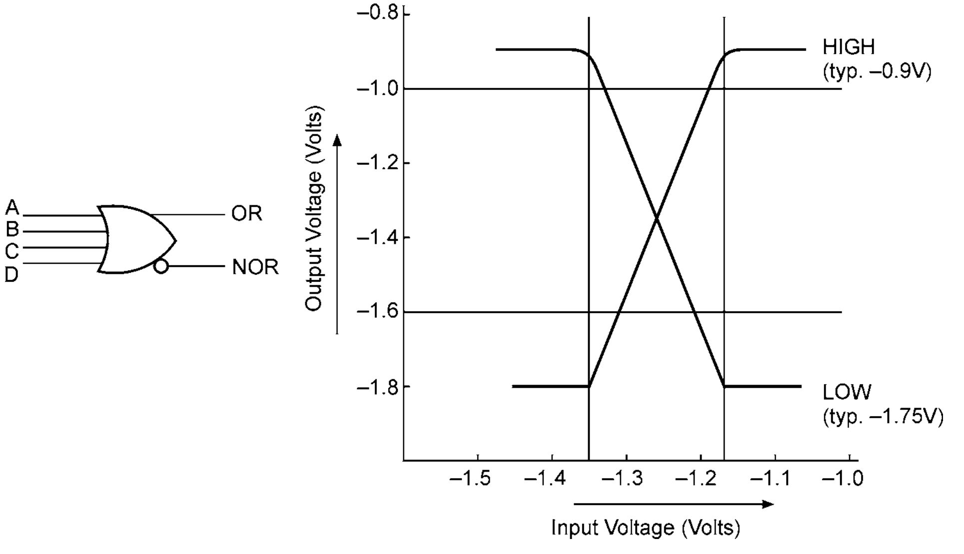


Figure 5.33 ECL input/output characteristics.

We will also note that voltage changes in ECL are small, largely governed by V_{BE} of the various conducting transistors. In fact, the magnitude of the currents flowing through various conducting transistors is of greater relevance to the operation of the ECL circuits. It is for this reason that emitter coupled logic is also sometimes called *current mode logic* (CML).

5.4.3 Salient Features of ECL

There are many features possessed by MECL family devices other than their high speed characteristics that make them attractive for many high-performance applications. The major ones are as follows:

1. ECL family devices produce the true and complementary output of the intended function simultaneously at the outputs without the use of any external inverters. This in turn reduces package count, reduces power requirements and also minimizes problems arising out of time delays that would be caused by external inverters.
2. The ECL gate structure inherently has high input impedance and low output impedance, which is very conducive to achieving large fan-out and drive capability.
3. ECL devices with open emitter outputs allow them to have transmission line drive capability. The outputs match any line impedance. Also, the absence of any pull-down resistors saves power.
4. ECL devices produce a near-constant current drain on the power supply, which simplifies power supply design.
5. On account of the differential amplifier design, ECL devices offer a wide performance flexibility, which allows ECL circuits to be used both as linear and as digital circuits.
6. Termination of unused inputs is easy. Resistors of approximately 50 kΩ allow unused inputs to remain unconnected.

5.5 CMOS Logic Family

The CMOS (Complementary Metal Oxide Semiconductor) logic family uses both N-type and P-type MOSFETs (enhancement MOSFETs, to be more precise) to realize different logic functions. The two types of MOSFET are designed to have matching characteristics. That is, they exhibit identical characteristics in switch-OFF and switch-ON conditions. The main advantage of the CMOS logic family over bipolar logic families discussed so far lies in its extremely low power dissipation, which is near-zero in static conditions. In fact, CMOS devices draw power only when they are switching. This allows integration of a much larger number of CMOS gates on a chip than would have been possible with bipolar or NMOS (to be discussed later) technology. CMOS technology today is the dominant semiconductor technology used for making microprocessors, memory devices and application-specific integrated circuits (ASICs). The CMOS logic family, like TTL, has a large number of subfamilies. The prominent members of CMOS logic were listed in an earlier part of the chapter. The basic difference between different CMOS logic subfamilies such as 4000A, 4000B, 4000UB, 74C, 74HC, 74HCT, 74AC and 74ACT is in the fabrication process used and not in the design of the circuits employed to implement the intended logic function. We will firstly look at the circuit implementation of various logic functions in CMOS and then follow this up with a brief description of different subfamilies of CMOS logic.

5.5.1 Circuit Implementation of Logic Functions

In the following paragraphs, we will briefly describe the internal schematics of basic logic functions when implemented in CMOS logic. These include inverter, NAND, NOR, AND, OR, EX-OR, EX-NOR and AND-OR-INVERT functions.

5.5.1.1 CMOS Inverter

The inverter is the most fundamental building block of CMOS logic. It consists of a pair of N-channel and P-channel MOSFETs connected in cascade configuration as shown in Fig. 5.34. The circuit

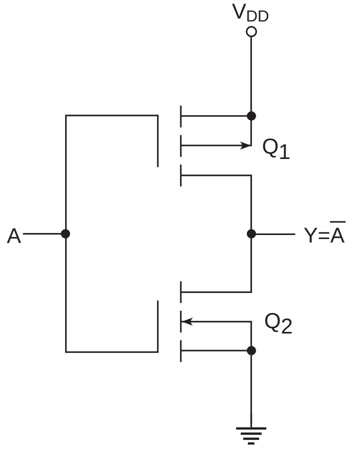


Figure 5.34 CMOS inverter.

functions as follows. When the input is in the HIGH state (logic '1'), P-channel MOSFET Q_1 is in the cut-off state while the N-channel MOSFET Q_2 is conducting. The conducting MOSFET provides a path from ground to output and the output is LOW (logic '0'). When the input is in the LOW state (logic '0'), Q_1 is in conduction while Q_2 is in cut-off. The conducting P-channel device provides a path for V_{DD} to appear at the output, so that the output is in HIGH or logic '1' state. A floating input could lead to conduction of both MOSFETs and a short-circuit condition. It should therefore be avoided. It is also evident from Fig. 5.34 that there is no conduction path between V_{DD} and ground in either of the input conditions, that is, when input is in logic '1' and '0' states. That is why there is practically zero power dissipation in static conditions. There is only dynamic power dissipation, which occurs during switching operations as the MOSFET gate capacitance is charged and discharged. The power dissipated is directly proportional to the switching frequency.

5.5.1.2 NAND Gate

Figure 5.35 shows the basic circuit implementation of a two-input NAND. As shown in the figure, two P-channel MOSFETs (Q_1 and Q_2) are connected in parallel between V_{DD} and the output terminal, and two N-channel MOSFETs (Q_3 and Q_4) are connected in series between ground and output terminal. The circuit operates as follows. For the output to be in a logic '0' state, it is essential that both the series-connected N-channel devices conduct and both the parallel-connected P-channel devices remain in the cut-off state. This is possible only when both the inputs are in a logic '1' state. This verifies one of the entries of the NAND gate truth table. When both the inputs are in a logic '0' state, both the N-channel devices are nonconducting and both the P-channel devices are conducting, which produces a logic '1' at the output. This verifies another entry of the NAND truth table. For the remaining two input combinations, either of the two N-channel devices will be nonconducting and either of the two parallel-connected P-channel devices will be conducting. We have either Q_3 OFF and Q_2 ON or Q_4 OFF and Q_1 ON. The output in both cases is a logic '1', which verifies the remaining entries of the truth table.

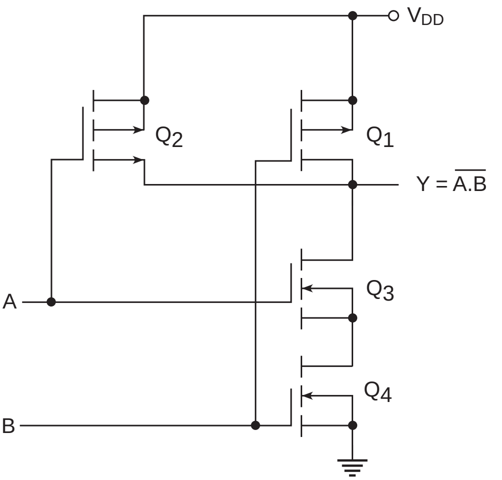


Figure 5.35 CMOS NAND.

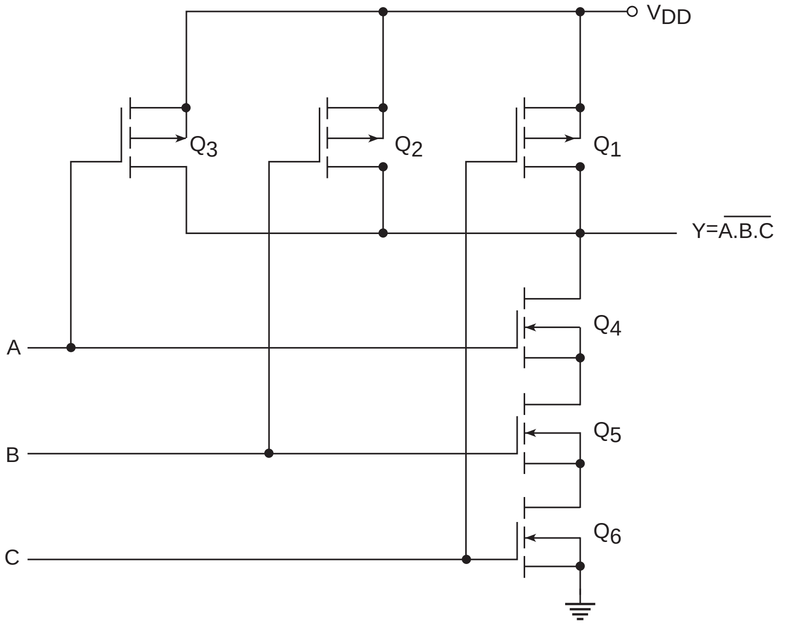


Figure 5.36 Three-input NAND in CMOS.

From the circuit schematic of Fig. 5.35 we can visualize that under no possible input combination of logic states is there a direct conduction path between V_{DD} and ground. This further confirms that there is near-zero power dissipation in CMOS gates under static conditions. Figure 5.36 shows how the circuit of Fig. 5.35 can be extended to build a three-input NAND gate. Operation of this circuit can be explained on similar lines. It may be mentioned here that series connection of MOSFETs adds to the propagation delay, which is greater in the case of P-channel devices than it is in the case of N-channel devices. As a result, the concept of extending the number of inputs as shown in Fig. 5.36 is usually limited to four inputs in the case of NAND and to three inputs in the case of NOR. The number is one less in the case of NOR because it uses series-connected P-channel devices. NAND and NOR gates with larger inputs are realized as a combination of simpler gates.

5.5.1.3 NOR Gate

Figure 5.37 shows the basic circuit implementation of a two-input NOR. As shown in the figure, two P-channel MOSFETs (Q_1 and Q_2) are connected in series between V_{DD} and the output terminal, and two N-channel MOSFETs (Q_3 and Q_4) are connected in parallel between ground and output terminal. The circuit operates as follows. For the output to be in a logic ‘1’ state, it is essential that both the series-connected P-channel devices conduct and both the parallel-connected N-channel devices remain in the cut-off state. This is possible only when both the inputs are in a logic ‘0’ state. This verifies one of the entries of the NOR gate truth table. When both the inputs are in a logic ‘1’ state, both the N-channel devices are conducting and both the P-channel devices are nonconducting, which produces a logic ‘0’ at the output. This verifies another entry of the NOR truth table. For the remaining two

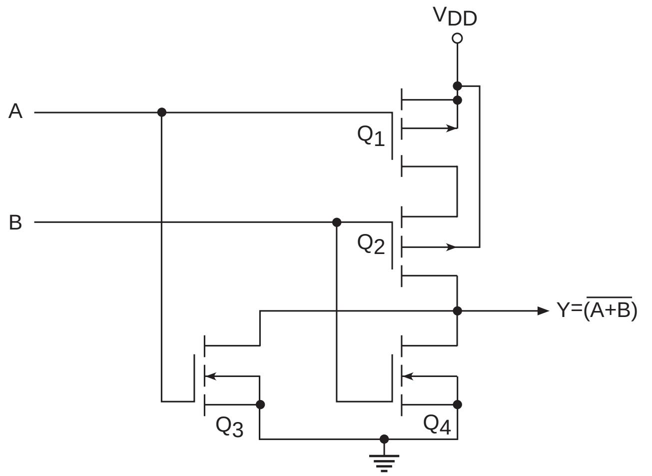


Figure 5.37 Two-input NOR in CMOS.

input combinations, either of the two parallel N-channel devices will be conducting and either of the two series-connected P-channel devices will be nonconducting. We have either Q_1 OFF and Q_3 ON or Q_2 OFF and Q_4 ON. The output in both cases is logic ‘0’, which verifies the remaining entries of the truth table.

Figure 5.38 shows how the circuit of Fig. 5.37 can be extended to build a three-input NOR gate. The operation of this circuit can be explained on similar lines. As already explained, NOR gates with more than three inputs are usually realized as a combination of simpler gates.

5.5.1.4 AND Gate

An AND gate is nothing but a NAND gate followed by an inverter. Figure 5.39 shows the internal schematic of a two-input AND in CMOS. A buffered AND gate is fabricated by using a NOR gate schematic with inverters at both of its inputs and its output feeding two series-connected inverters.

5.5.1.5 OR Gate

An OR gate is nothing but a NOR gate followed by an inverter. Figure 5.40 shows the internal schematic of a two-input OR in CMOS. A buffered OR gate is fabricated by using a NAND gate schematic with inverters at both of its inputs and its output feeding two series-connected inverters.

5.5.1.6 EXCLUSIVE-OR Gate

An EXCLUSIVE-OR gate is implemented using the logic diagram of Fig. 5.41(a). As is evident from the figure, the output of this logic arrangement can be expressed by

$$[\overline{(\overline{A+B})+A.B} = (\overline{\overline{A}.\overline{B}+A.B})] = \text{EX}-\text{OR function} \qquad (5.1)$$

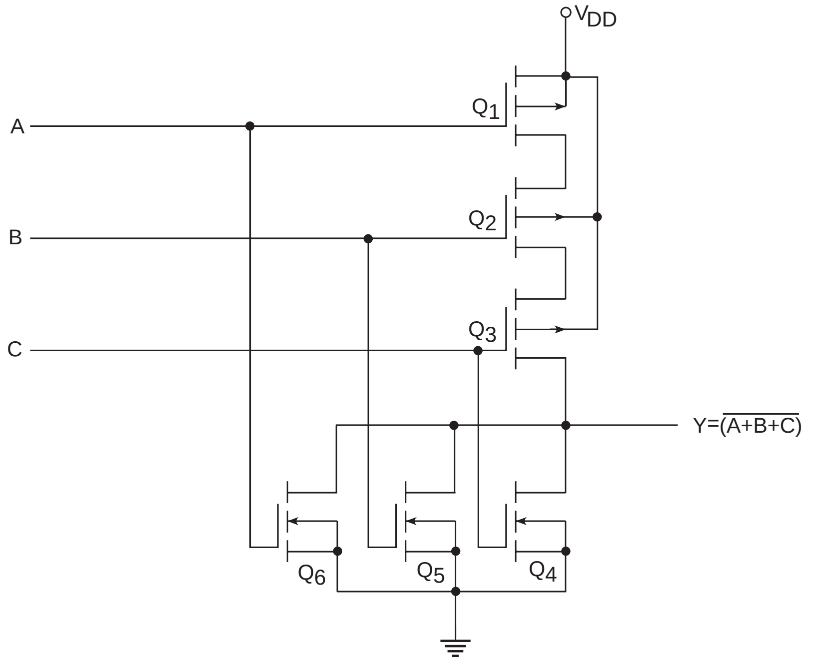


Figure 5.38 Three-input NOR.

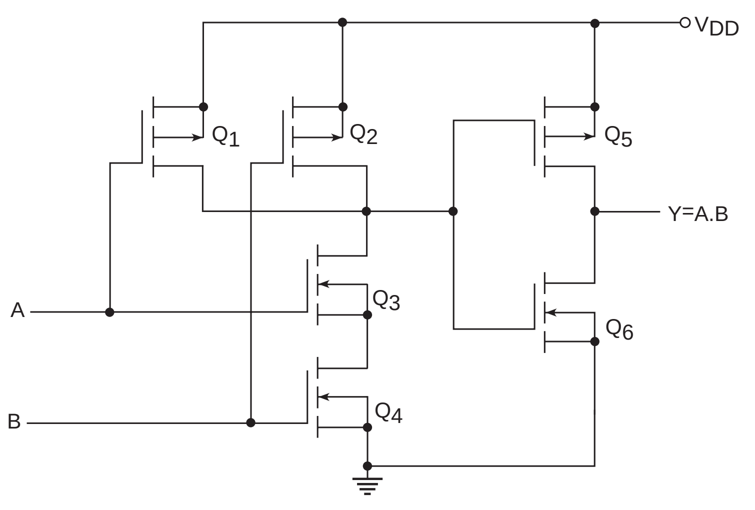


Figure 5.39 Two-input AND in CMOS.

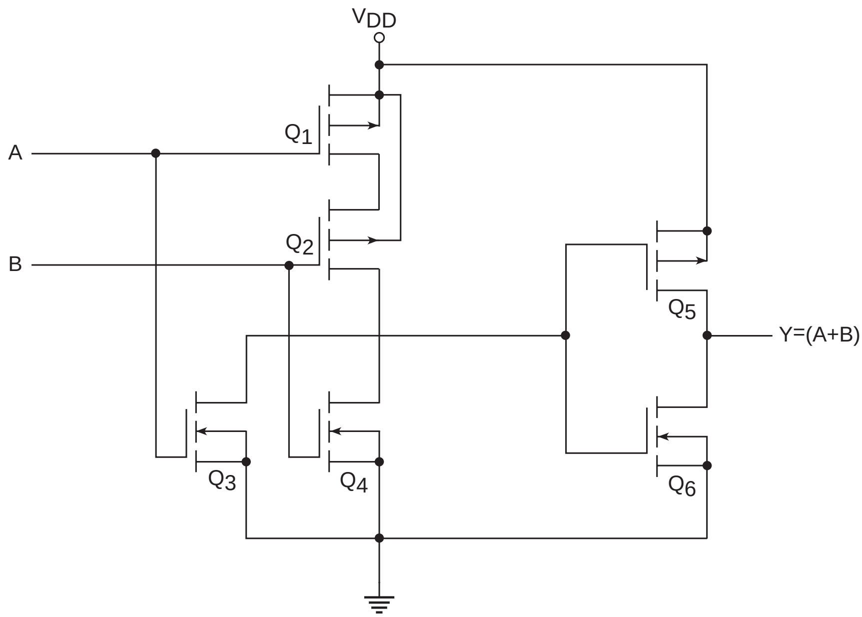


Figure 5.40 Two-input OR in CMOS.

Figure 5.41(b) shows the internal schematic of a two-input EX-OR gate. MOSFETs Q_1–Q_4 constitute the NOR gate. MOSFETS Q_5 and Q_6 simulate ANDing of A and B, and MOSFET Q_7 provides ORing of the NOR output with ANDed output. Since MOSFETs Q_8–Q_{10} make up the complement of the arrangement of MOSFETs Q_5–Q_7, the final output is inverted. Thus, the schematic of Fig. 5.41(b) implements the logic arrangement of Fig. 5.41(a) and hence a two-input EX-OR gate.

5.5.1.7 EXCLUSIVE-NOR Gate

An EXCLUSIVE-NOR gate is implemented using the logic diagram of Fig. 5.42(a). As is evident from the figure, the output of this logic arrangement can be expressed by

$$\overline{[(\overline{A.B}).(A+B)]} = \overline{[(\overline{A}+\overline{B}).(A+B)]} = \text{EX}-\text{NOR function} \tag{5.2}$$

Figure 5.42(b) shows the internal schematic of a two-input EX-NOR gate. MOSFETs Q_1–Q_4 constitute the NAND gate. MOSFETS Q_5 and Q_6 simulate ORing of A and B, and MOSFET Q_7 provides ANDing of the NAND output with ORed output. Since MOSFETs Q_8–Q_{10} make up the complement of the arrangement of MOSFETs Q_5–Q_7, the final output is inverted. Thus, the schematic of Fig. 5.42(b) implements the logic arrangement of Fig. 5.42(a) and hence a two-input EX-NOR gate.

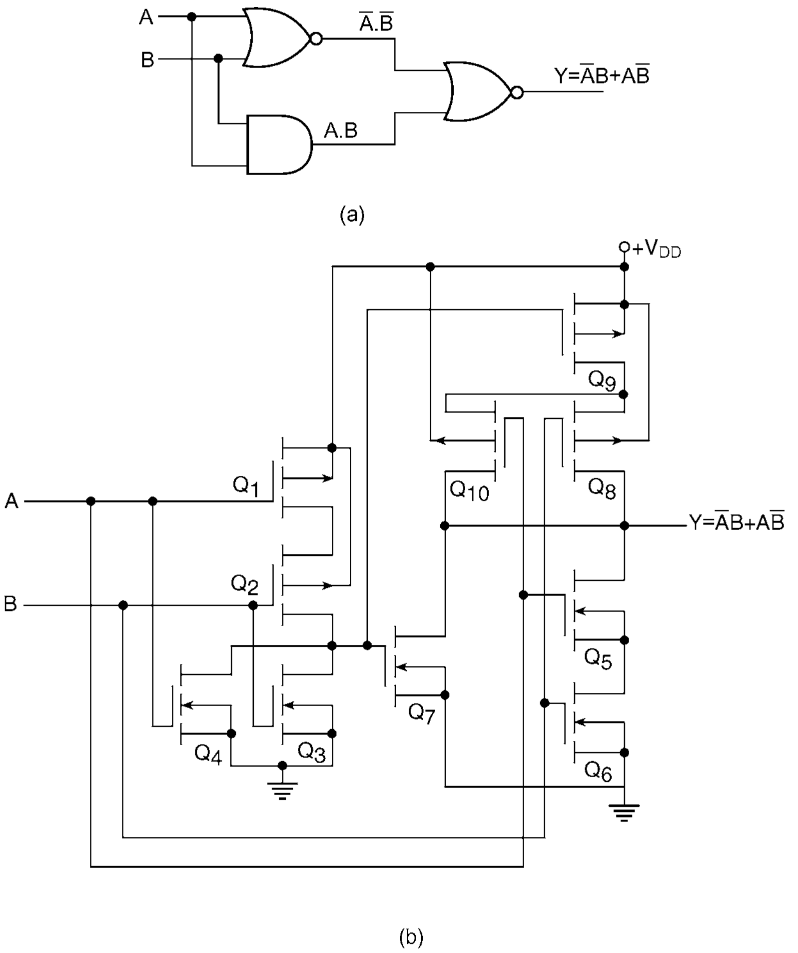


Figure 5.41 Two-input EX-OR in CMOS.

5.5.1.8 AND-OR-INVERT and OR-AND-INVERT Gates

Figure 5.43 shows the internal schematic of a typical two-wide, two-input AND-OR-INVERT gate. The output of this gate can be logically expressed by the Boolean equation

$$Y = (\overline{A.B + C.D}) \tag{5.3}$$

From the above expression, we can say that the output should be in a logic '0' state for the following input conditions:

1. When either $A.B =$ logic '1' or $C.D =$ logic '1'
2. When both $A.B$ and $C.D$ equal logic '1'.

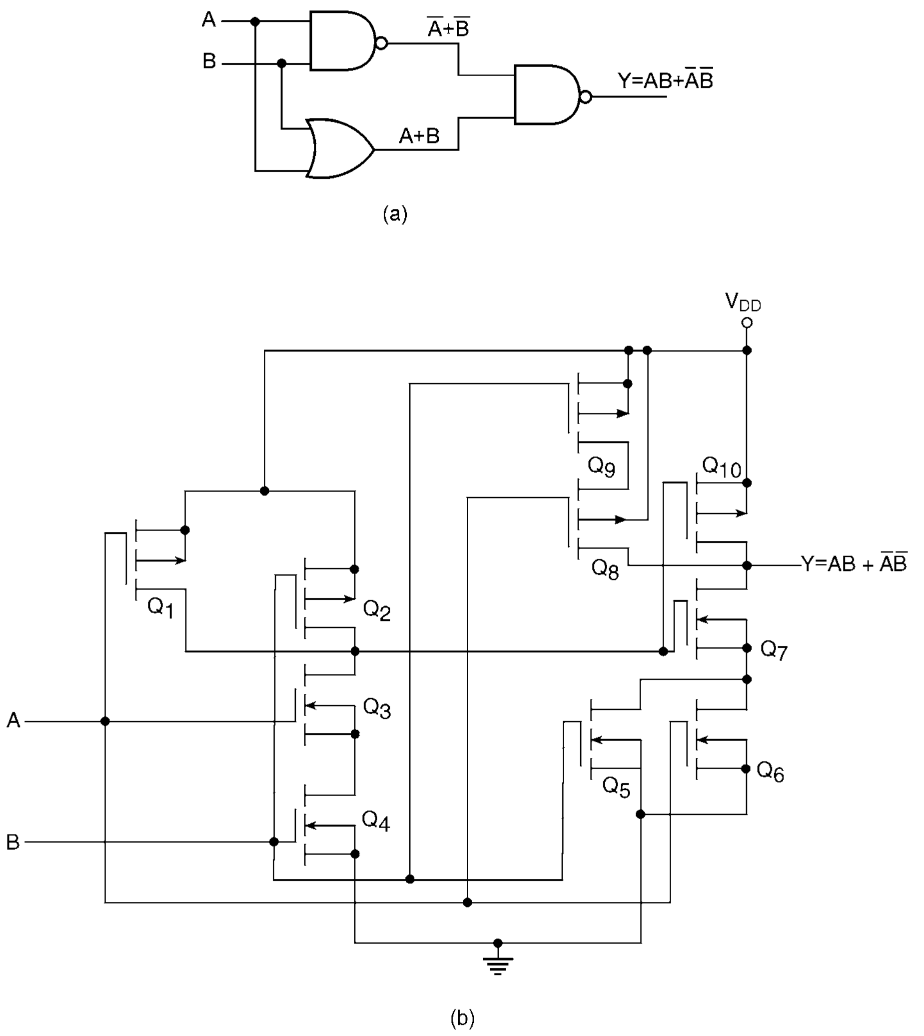


Figure 5.42 Two-input EX-NOR in CMOS.

For both these conditions there is a conduction path available from ground to output, which verifies that the circuit satisfies the logic expression. Also, according to the logic expression for the AND-OR-INVERT gate, the output should be in a logic '1' state when both $A.B$ and $C.D$ equal logic '0'. This implies that:

1. Either A or B or both are in a logic '0' state.
2. Either C or D or both are in a logic '0' state.

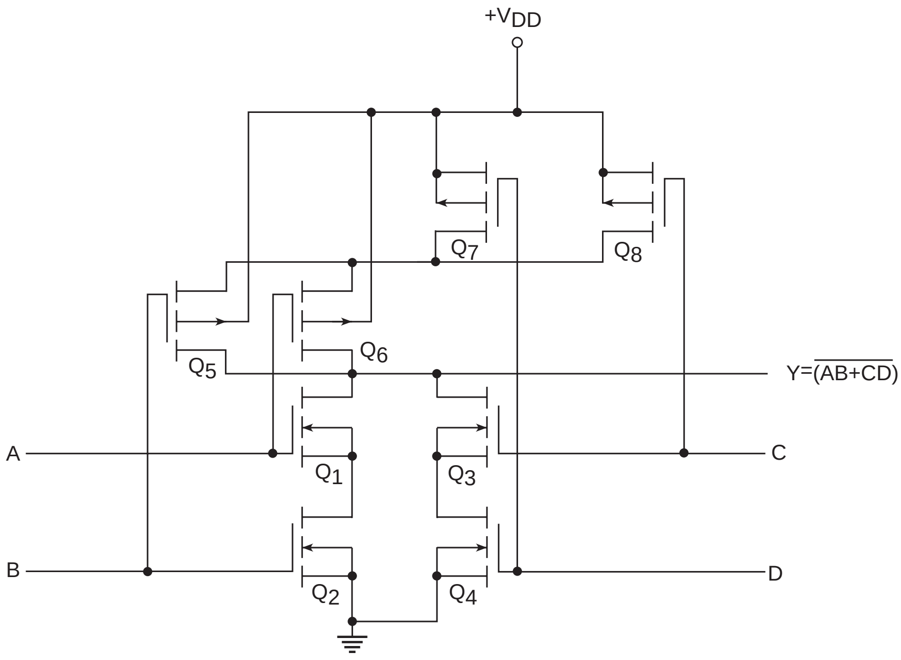


Figure 5.43 Two-wide, two-input AND-OR-INVERT gate in CMOS.

If these conditions are applied to the circuit of Fig. 5.43, we find that the ground will remain disconnected from the output and also that there is always a path from V_{DD} to output. This leads to a logic '1' at the output. Thus, we have proved that the given circuit implements the intended logic expression for the AND-OR-INVERT gate.

The OR-AND-INVERT gate can also be implemented in the same way. Figure 5.44 shows a typical internal schematic of a two-wide, two-input OR-AND-INVERT gate. The output of this gate can be expressed by the Boolean equation

$$Y = \overline{(A+B).(C+D)} \tag{5.4}$$

It is very simple to draw the internal schematic of an AND-OR-INVERT or OR-AND-INVERT gate. The circuit has two parts, that is, the N-channel MOSFET part of the circuit and the P-channel part of the circuit. Let us see, for instance, how Boolean equation (5.4) relates to the circuit of Fig. 5.44. The fact that we need (*A* OR *B*) AND (*C* OR *D*) explains why the N-channel MOSFETs representing *A* and *B* inputs are in parallel and also why the N-channel MOSFETs representing *C* and *D* are also in parallel. The two parallel arrangements are then connected in series to achieve an ANDing operation. The complementary P-channel MOSFET section achieves inversion. Note that the P-channel section is the complement of the N-channel section with N-channel MOSFETs replaced by P-channel MOSFETs and parallel connection replaced by series connection, and vice versa. The operation of an AND-OR-INVERT gate can be explained on similar lines to the case of an OR-AND-INVERT gate. Expansion of both AND-OR-INVERT and OR-AND-INVERT gates should be obvious, ensuring that we do not have more than three devices in series.

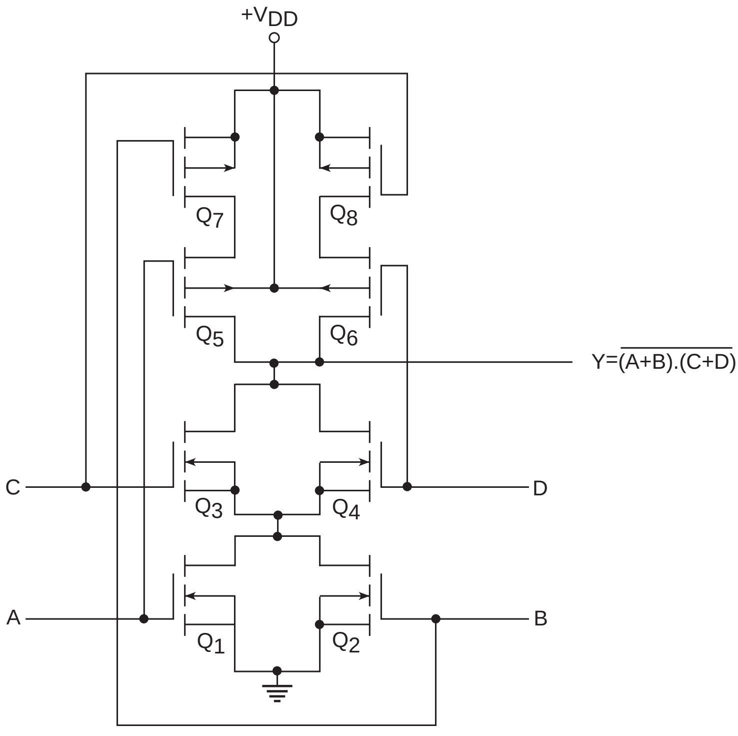


Figure 5.44 Two-wide, two-input OR-AND-INVERT gate.

5.5.1.9 Transmission Gate

The transmission gate, also called the *bilateral switch*, is exclusive to CMOS logic and does not have a counterpart in the TTL and ECL families. It is essentially a single-pole, single-throw (SPST) switch. The opening and closing operations can be controlled by externally applied logic levels. Figure 5.45(a) shows the circuit symbol. If a logic '0' at the control input corresponds to an open switch, then a logic '1' corresponds to a closed switch, and vice versa. The internal schematic of a transmission gate is nothing but a parallel connection of an N-channel MOSFET and a P-channel MOSFET with the control input applied to the gates, as shown in Fig. 5.45(b). Control inputs to the gate terminals of two MOSFETs are the complement of each other. This is ensured by an inbuilt inverter.

When the control input is HIGH (logic '1'), both devices are conducting and the switch is closed. When the control input is LOW (logic '0'), both devices are open and therefore the switch is open. It may be mentioned here that there is no discrimination between input and output terminals. Either of the two can be treated as the input terminal for the purpose of applying input. This is made possible by the symmetry of the two MOSFETs.

It may also be mentioned here that the ON-resistance of a conducting MOSFET depends upon drain and source voltages. In the case of an N-channel MOSFET, if the source voltage is close to V_{DD}, there is an increase in ON-resistance, leading to an increased voltage drop across the switch. A similar phenomenon is observed when the source voltage of a P-channel MOSFET is close to

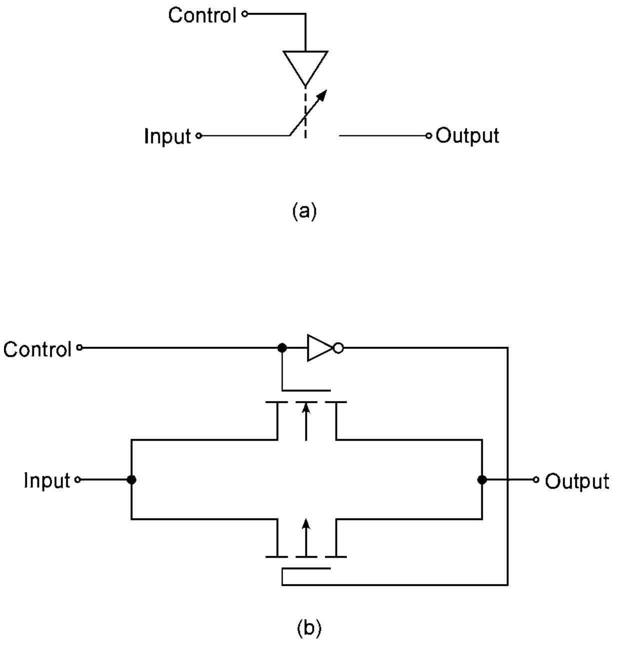


Figure 5.45 Transmission gate.

ground. Such behaviour causes no problem in static CMOS logic gates, where source terminals of all N-channel MOSFETs are connected to ground and source terminals of all P-channel MOSFETs are connected to V_{DD}. This would cause a problem if a single N-channel or P-channel device were used as a switch. Such a problem is overcome with the use of parallel connection of N-channel and P-channel devices. Transmission gate devices are available in 4000-series as well as 74HC series of CMOS logic.

5.5.1.10 CMOS with Open Drain Outputs

The outputs of conventional CMOS gates should never be shorted together, as illustrated by the case of two inverters shorted at the output terminals (Fig. 5.46). If the input conditions are such that the output of one inverter is HIGH and that of the other is LOW, the output circuit is then like a voltage divider network with two identical resistors equal to the ON-resistance of a conducting MOSFET. The output is then approximately equal to $V_{DD}/2$, which lies in the indeterminate range and is therefore unacceptable. Also, an arrangement like this draws excessive current and could lead to device damage.

This problem does not exist in CMOS gates with open drain outputs. Such a device is the counterpart to gates with open collector outputs in the TTL family. The output stage of a CMOS gate with an open drain output is a single N-channel MOSFET with an open drain terminal, and there is no P-channel MOSFET. The open drain terminal needs to be connected to V_{DD} through an external pull-up resistor. Figure 5.47 shows the internal schematic of a CMOS inverter with an open drain output. The pull-up resistor shown in the circuit is external to the device.

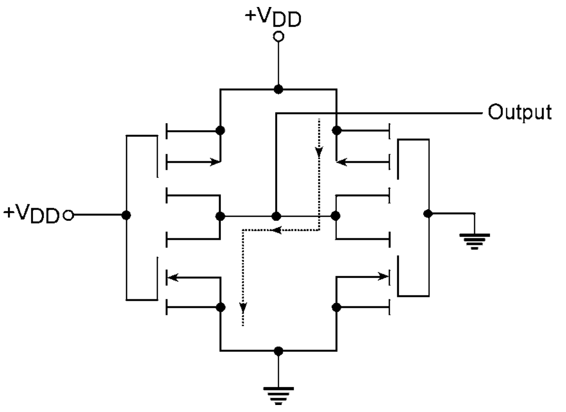


Figure 5.46 CMOS inverters with shorted outputs.

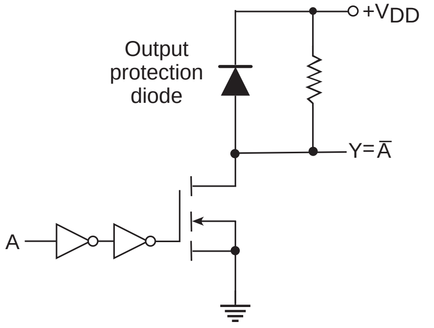


Figure 5.47 CMOS inverter with an open drain output.

5.5.1.11 CMOS with Tristate Outputs

Like tristate TTL, CMOS devices are also available with tristate outputs. The operation of tristate CMOS devices is similar to that of tristate TTL. That is, when the device is enabled it performs its intended logic function, and when it is disabled its output goes to a high-impedance state. In the high-impedance state, both N-channel and P-channel MOSFETs are driven to an OFF-state. Figure 5.48 shows the internal schematic of a tristate buffer with active LOW ENABLE input. The circuit shown is that of one of the buffers in CMOS hex buffer type CD4503B. The outputs of tristate CMOS devices can be connected together in a bus arrangement, like tristate TTL devices with the same condition that only one device is enabled at a time.

5.5.1.12 Floating or Unused Inputs

Unused inputs of CMOS devices should never be left floating or unconnected. A floating input is highly susceptible to picking up noise and accumulating static charge. This can often lead to simultaneous

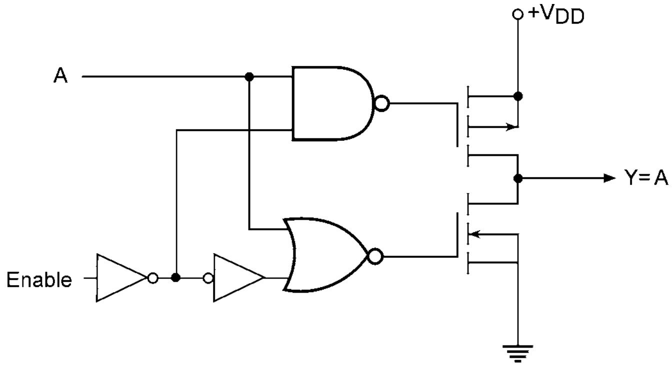


Figure 5.48 Tristate buffer in CMOS.

conduction of P-channel and N-channel devices on the chip, which causes increased power dissipation and overheating. Unused inputs of CMOS gates should either be connected to ground or V_{DD} or shorted to another input. The same is applicable to the inputs of all those gates that are not in use. For example, we may be using only two of the four gates available on an IC having four gates. The inputs of the remaining two gates should be tied to either ground or V_{DD}.

5.5.1.13 Input Protection

Owing to the high input impedance of CMOS devices, they are highly susceptible to static charge build-up. As a result of this, voltage developed across the input terminals could become sufficiently high to cause dielectric breakdown of the gate oxide layer. In order to protect the devices from this static charge build-up and its damaging consequences, the inputs of CMOS devices are protected by using a suitable resistor–diode network, as shown in Fig. 5.49(a). The protection circuit shown is typically used in metal-gate MOSFETs such as those used in 4000-series CMOS devices. Diode D_2 limits the positive voltage surges to $V_{DD} + 0.7$ V, while diode D_3 clamps the negative voltage surges to -0.7 V. Resistor R_1 limits the static discharge current amplitude and thus prevents any damagingly large voltage from being directly applied to the input terminals. Diode D_1 does not contribute to input protection. It is a distributed P–N junction present owing to the diffusion process used for fabrication of resistor R_1. The protection diodes remain reverse biased for the normal input voltage range of 0 to V_{DD}, and therefore do not affect normal operation.

Figure 5.49(b) shows a typical input protection circuit used for silicon-gate MOSFETs used in 74C, 74HC, etc., series CMOS devices. A distributed P–N junction is absent owing to R_1 being a polysilicon resistor. Diodes D_1 and D_2 do the same job as diodes D_2 and D_3 in the case of metal-gate devices. Diode D_2 is usually fabricated in the form of a bipolar transistor with its collector and base terminals shorted.

5.5.1.14 Latch-up Condition

This is an undesired condition that can occur in CMOS devices owing to the existence of parasitic bipolar transistors (NPN and PNP) embedded in the substrate. While N-channel MOSFETs lead to the

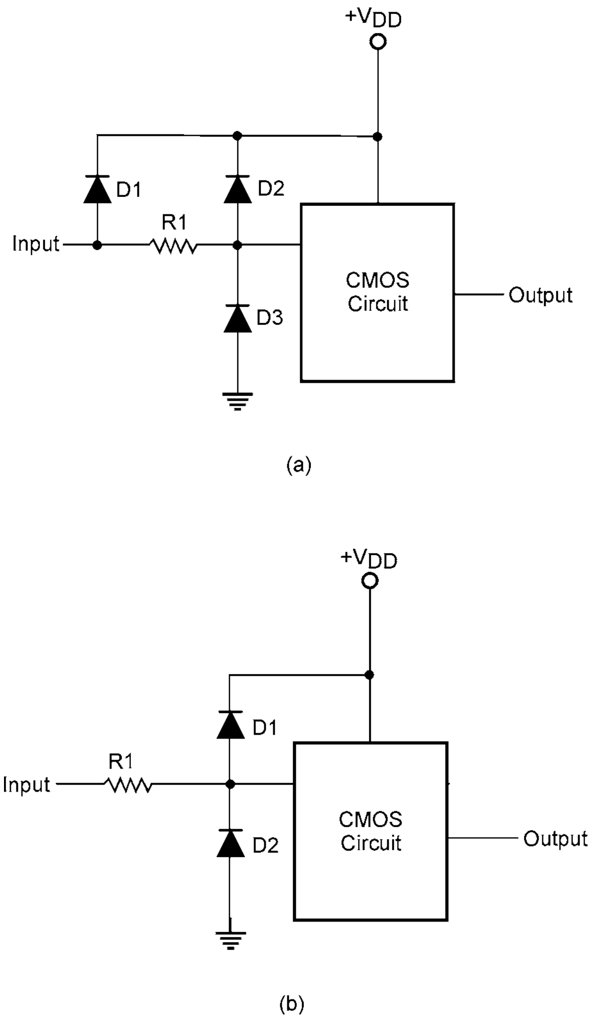


Figure 5.49 (a) Input protection circuit-metal-gate devices and (b) input protection circuit-silicon-gate devices.

presence of NPN transistors, P-channel MOSFETs are responsible for the existence of PNP transistors. If we look into the arrangement of different semiconductor regions in the most basic CMOS building block, that is, the inverter, we will find that these parasitic NPN and PNP transistors find themselves interconnected in a back-to-back arrangement, with the collector of one transistor connected to the base of the other, and vice versa. Two such pairs of transistors connected in series exist between V_{DD} and ground in the case of an inverter, as shown in Fig. 5.50. If for some reason these parasitic elements are triggered into conduction, on account of inherent positive feedback they get into a latch-up condition and remain in conduction permanently. This can lead to the flow of large current and subsequently to destruction of the device. A latch-up condition can be triggered by high voltage spikes and ringing

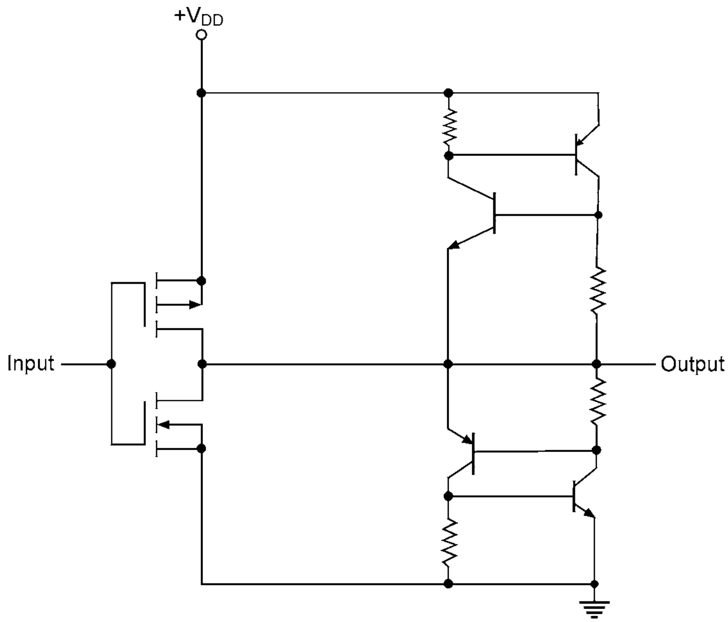


Figure 5.50 CMOS inverter with parasitic elements.

present at the device inputs and outputs. The device can also be prone to latch-up if its maximum ratings are exceeded. Modern CMOS devices use improved fabrication techniques so as to minimize factors that can cause this undesired effect. The use of external clamping diodes at inputs and outputs, proper termination of unused inputs and regulated power supply with a current-limiting feature also helps in minimizing the chances of occurrence of the latch-up condition and in minimizing its effects if it occurs.

5.5.2 CMOS Subfamilies

In the following paragraphs, we will briefly describe various subfamilies of CMOS logic, including subfamilies of the 4000 series and those of TTL pin-compatible 74C series.

5.5.2.1 4000-series

The 4000A-series CMOS ICs, introduced by RCA, were the first to arrive on the scene from the CMOS logic family. The 4000A CMOS subfamily is obsolete now and has been replaced by 4000B and 4000UB subfamilies. We will therefore not discuss it in detail. The 4000B series is a high-voltage version of the 4000A series, and also all the outputs in this series are buffered. The 4000UB series is also a high-voltage version of the 4000A series, but here the outputs are not buffered. A buffered CMOS device is one that has constant output impedance irrespective of the logic status of the inputs. If we recall the internal schematics of the basic CMOS logic gates described in the previous pages, we will see that, with the exception of the inverter, the output impedance of other gates depends upon the logic status of the inputs. This variation in output impedance occurs owing to the varying combination of MOSFETs that conduct for a given input combination. All buffered devices are designated by the suffix 'B' and referred to as the 4000B series. The 4000-series devices that meet 4000B series specifications except for the V_{IL} and V_{IH} specifications and that the outputs are not buffered are called unbuffered devices and are said to belong to the 4000UB series.

Figures 5.51 and 5.52 show a comparison between the internal schematics of a buffered two-input NOR (Fig. 5.51) and an unbuffered two-input NOR (Fig. 5.52). A buffered gate has been implemented by using inverters at the inputs to a two-input NAND whose output feeds another inverter. This is the typical arrangement followed by various manufacturers, as the inverters at the input enhance noise immunity. Another possible arrangement would be a two-input NOR whose output feeds two series-connected inverters.

Variation in the output impedance of unbuffered gates is larger for gates with a larger number of inputs. For example, unbuffered gates have an output impedance of 200–400 Ω in the case of two-input gates, 133–400 Ω for three-input gates and 100–400 Ω for gates with four inputs. Buffered gates have an output impedance of 400 Ω. Since they have the same maximum output impedance, their minimum I_{OL} and I_{OH} specifications are the same.

Characteristic features of 4000B and 4000UB CMOS devices are as follows: V_{IH} (buffered devices) = 3.5 V (for V_{DD} = 5 V), 7.0 V (for V_{DD} = 10 V) and 11.0 V (for V_{DD} = 15 V); V_{IH} (unbuffered devices) = 4.0 V (for V_{DD} = 5 V), 8.0 V (for V_{DD} = 10 V) and 12.5 V (for V_{DD} = 15 V); I_{IH} = 1.0 μA; I_{IL} = 1.0 μA; I_{OH} = 0.2 mA (for V_{DD} = 5 V), 0.5 mA (for V_{DD} = 10 V) and 1.4 mA (for V_{DD} = 15 V); I_{OL} = 0.52 mA (for V_{DD} = 5 V), 1.3 mA (for V_{DD} = 10 V) and 3.6 mA (for V_{DD} = 15 V); V_{IL} (buffered devices) = 1.5 V (for V_{DD} = 5 V), 3.0 V (for V_{DD} = 10 V) and 4.0 V (for V_{DD} = 15 V); V_{IL} (unbuffered devices) = 1.0 V (for V_{DD} = 5 V), 2.0 V (for V_{DD} = 10 V) and 2.5 V (for V_{DD} = 15 V); V_{OH} = 4.95 V

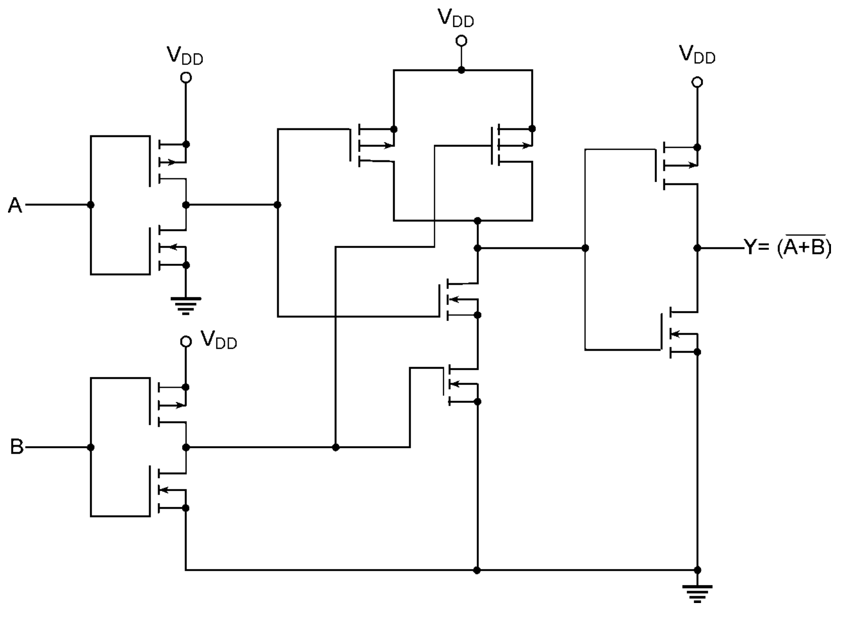


Figure 5.51 Buffered two-input NOR.

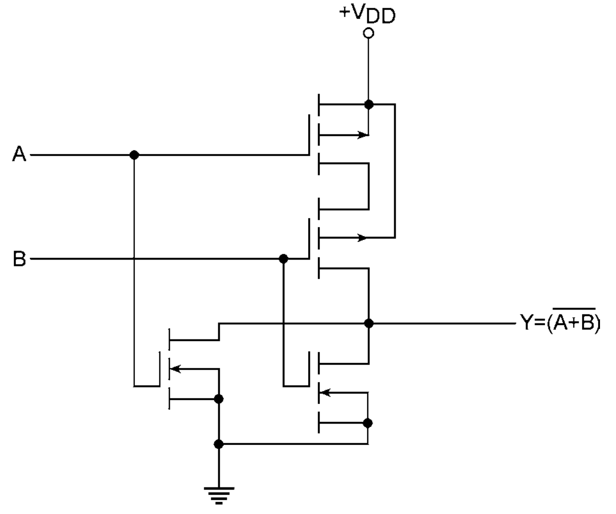


Figure 5.52 Unbuffered two-input NOR.

(for $V_{DD} = 5$ V), 9.95 V (for V_{DD} = 10 V) and 14.95 V (for $V_{DD} = 15$ V); $V_{OL} = 0.05$ V; $V_{DD} = 3$–15 V; propagation delay (buffered devices) = 150 ns (for $V_{DD} = 5$ V), 65 ns (for $V_{DD} = 10$ V) and 50 ns (for $V_{DD} = 15$ V); propagation delay (unbuffered devices) = 60 ns (for $V_{DD} = 5$ V), 30 ns (for $V_{DD} = 10$ V) and 25 ns (for $V_{DD} = 15$ V); noise margin (buffered devices) = 1.0 V (for $V_{DD} = 5$ V), 2.0 V (for $V_{DD} = 10$ V) and 2.5 V (for $V_{DD} = 15$ V); noise margin (unbuffered devices) = 0.5 V (for $V_{DD} = 5$ V), 1.0 V (for $V_{DD} = 10$ V) and 1.5 V (for $V_{DD} = 15$ V); output transition time (for $V_{DD} = 5$ V and $C_L = 50$ pF) = 100 ns (buffered devices) and 50–100 ns (for unbuffered devices); power dissipation per gate (for $f = 100$ kHz) = 0.1 mW; speed–power product (for $f = 100$ kHz) = 5 pJ; maximum flip-flop toggle rate = 12 MHz.

5.5.2.2 74C Series

The 74C CMOS subfamily offers pin-to-pin replacement of the 74-series TTL logic functions. For instance, if 7400 is a quad two-input NAND in standard TTL, then 74C00 is a quad two-input NAND with the same pin connections in CMOS. The characteristic parameters of the 74C series CMOS are more or less the same as those of 4000-series devices.

5.5.2.3 74HC/HCT Series

The 74HC/HCT series is the high-speed CMOS version of the 74C series logic functions. This is achieved using silicon-gate CMOS technology rather than the metal-gate CMOS technology used in earlier 4000-series CMOS subfamilies. The 74HCT series is only a process variation of the 74HC series. The 74HC/HCT series devices have an order of magnitude higher switching speed and also a much higher output drive capability than the 74C series devices. This series also offers pin-to-pin replacement of 74-series TTL logic functions. In addition, the 74HCT series devices have TTL-compatible inputs.

5.5.2.4 74AC/ACT Series

The 74AC series is presently the fastest CMOS logic family. This logic family has the best combination of high speed, low power consumption and high output drive capability. Again, 74ACT is only a process variation of 74AC. In addition, 74ACT series devices have TTL-compatible inputs.

The characteristic parameters of the 74C/74HC/74HCT/74AC/74ACT series CMOS are summarized as follows (for V_{DD}= 5 V): V_{IH} (min.) = 3.5 V (74C), 3.5 V (74HC and 74AC) and 2.0 V (74HCT and 74ACT); V_{OH} (min.) = 4.5 V (74C) and 4.9 V (74HC, 74HCT, 74AC and 74ACT); V_{IL}(max.) = 1.5 V (74C), 1.0 V (74HC), 0.8 V (74HCT), 1.5 V (74AC) and 0.8 V (74ACT); V_{OL} (max.) = 0.5 V (74C) and 0.1 V (74HC, 74HCT, 74AC and 74ACT); I_{IH}(max.) = 1 μA; I_{IL} (max.) = 1 μA; I_{OH} (max.) = 0.4 mA (74C), 4.0 mA (74HC and 74HCT) and 24 mA (74AC and 74ACT); I_{OL} (max.) = 0.4 mA (74C), 4.0 mA (74HC and 74HCT) and 24 mA (74AC and 74ACT); V_{NH} = 1.4 V (74C, 74HC and 74AC) and 2.9 V (74HCT and 74ACT); V_{NL} = 1.4 V (74C), 0.9 V (74HC), 0.7 V (74HCT and 74ACT) and 1.4 V (74AC); propagation delay = 50 ns (74C), 8 ns (74HC and 74HCT) and 4.7 ns (74AC and 74ACT); power dissipation per gate (for f = 100 kHz) = 0.1 mW (74C), 0.17 mW (74HC and 74HCT) and 0.08 mW (74AC and 74ACT); speed–power product (for f = 100 kHz) = 5 pJ (74C), 1.4 pJ (74HC and 74HCT) and 0.37 pJ (74AC and 74ACT); maximum flip-flop toggle rate = 12 MHz (74C), 40 MHz (74HC and 74HCT) and 100 MHz (74AC and 74ACT).

Example 5.7

Draw the internal schematic of: (a) a two-wide, four-input AND-OR-INVERT logic function in CMOS and (b) a two-wide, four-input OR-AND-INVERT logic function in CMOS.

Solution

(a) Let us assume that A, B, C, D, E, F, G and H are the logic variables. The output Y of this logic function can then be expressed by the equation

$$Y = \overline{A.B.C.D+E.F.G.H}) \tag{5.5}$$

Following the principles explained earlier in the text, the internal schematic is shown in Fig. 5.53(a). Series connection of N-channel MOSFETs on the left simulates ANDing of A, B, C and D, whereas series connection of N-channel MOSFETs on the right simulates ANDing of E, F, G and H. Parallel connection of two branches produces ORing of the ANDed outputs. Since the P-channel MOSFET arrangement is the complement of the N-channel MOSFET arrangement, the final output is what is given by Equation (5.5).

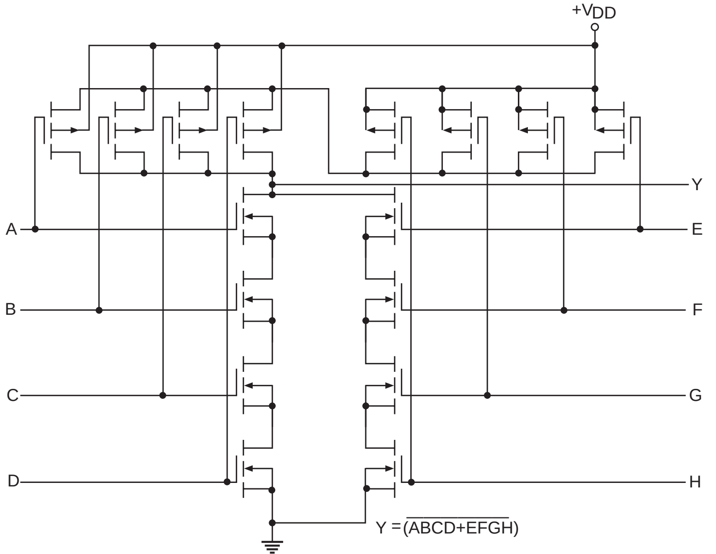
(b) The output Y of this logic function can be expressed by the equation

$$Y = \overline{(A+B+C+D).(E+F+G+H}) \tag{5.6}$$

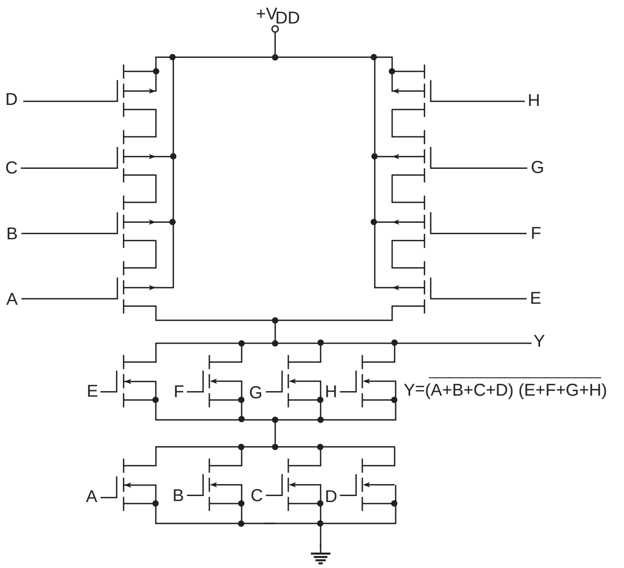
Figure 5.53(b) shows the internal schematic, which can be explained on similar lines.

Example 5.8

Determine the logic function performed by the CMOS digital circuit of Fig. 5.54.



(a)



(b)

Figure 5.53 Example 5.7.

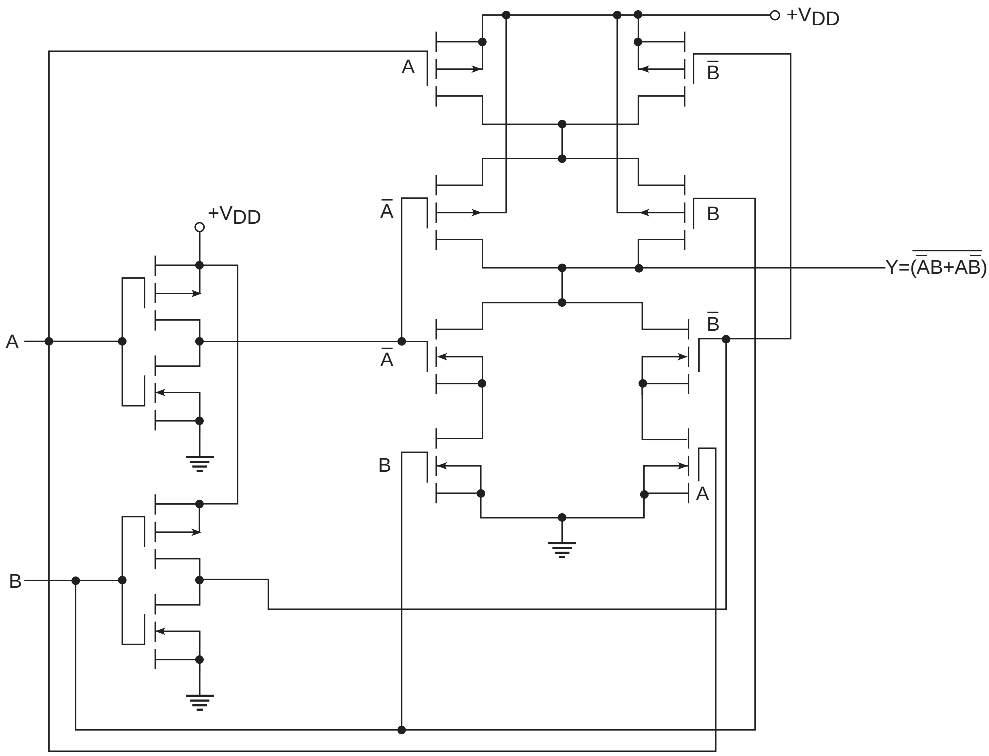


Figure 5.54 Example 5.8.

Solution

The given circuit can be divided into two stages. The first stage comprises two inverters that produce $\overline{A}$ and $\overline{B}$. The second stage is a two-wide, two-input AND-OR-INVERT circuit. Inputs to the first AND are $\overline{A}$ and B, and inputs to the second AND are A and $\overline{B}$. The final output is therefore given by $Y = \overline{(A.\overline{B} + \overline{A}.B)}$, which is an EX-NOR function.

5.6 BiCMOS Logic

The BiCMOS logic family integrates bipolar and CMOS devices on a single chip with the objective of deriving the advantages individually present in bipolar and CMOS logic families. While bipolar logic families such as TTL and ECL have the advantages of faster switching speed and larger output drive current capability, CMOS logic scores over bipolar counterparts when it comes to lower power dissipation, higher noise margin and larger packing density. BiCMOS logic attempts to get the best of both worlds. Two major categories of BiCMOS logic devices have emerged over the years since its introduction in 1985. In one type of device, moderate-speed bipolar circuits are combined with high-performance CMOS circuits. Here, CMOS circuitry continues to provide low power dissipation and larger packing density. Selective use of bipolar circuits gives improved performance. In the other category, the bipolar component is optimized to produce high-performance circuitry. In the following paragraphs, we will briefly describe the basic BiCMOS inverter and NAND circuits.

5.6.1 BiCMOS Inverter

Figure 5.55 shows the internal schematic of a basic BiCMOS inverter. When the input is LOW, N-channel MOSFETs Q_2 and Q_3 are OFF. P-channel MOSFET Q_1 and N-channel MOSFET Q_4 are ON. This leads transistors Q_5 and Q_6 to be in the ON and OFF states respectively. Transistor Q_6 is OFF because it does not get the required forward-biased base-emitter voltage owing to a conducting Q_4. Conducting Q_5 drives the output to a HIGH state, sourcing a large drive current to the load. The HIGH-state output voltage is given by the equation

$$V_{OH} = V_{DD} - V_{BE}(Q_5) \tag{5.7}$$

When the input is driven to a HIGH state, Q_2 and Q_3 turn ON. Initially, Q_4 is also ON and the output discharges through Q_3 and Q_4. When Q_4 turns OFF owing to its gate-source voltage falling below the required threshold voltage, the output continues to discharge until the output voltage equals the forward-biased base-emitter voltage drop of Q_6 in the active region. The LOW-state output voltage is given by the equation

$$V_{OL} = V_{BE}(Q_6 \text{ in active mode}) = 0.7V \tag{5.8}$$

5.6.2 BiCMOS NAND

Figure 5.56 shows the internal schematic of a two-input NAND in BiCMOS logic. The operation of this circuit can be explained on similar lines to the case of an inverter. Note that MOSFETs Q_1–Q_4

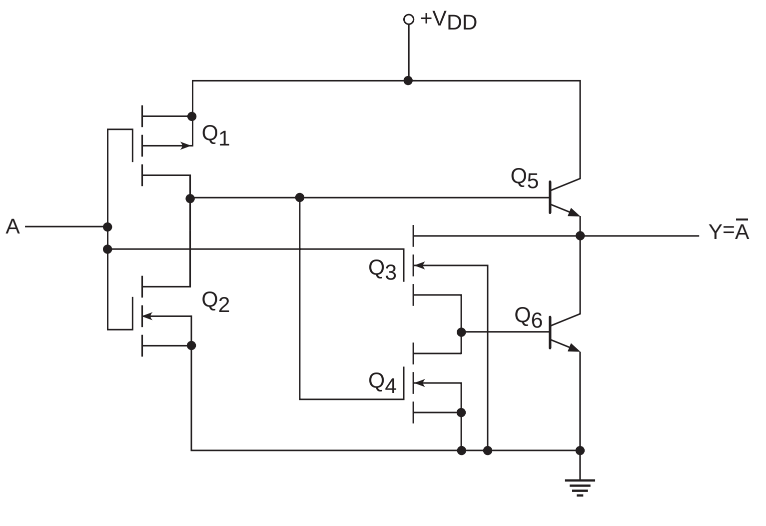


Figure 5.55 BiCMOS inverter.

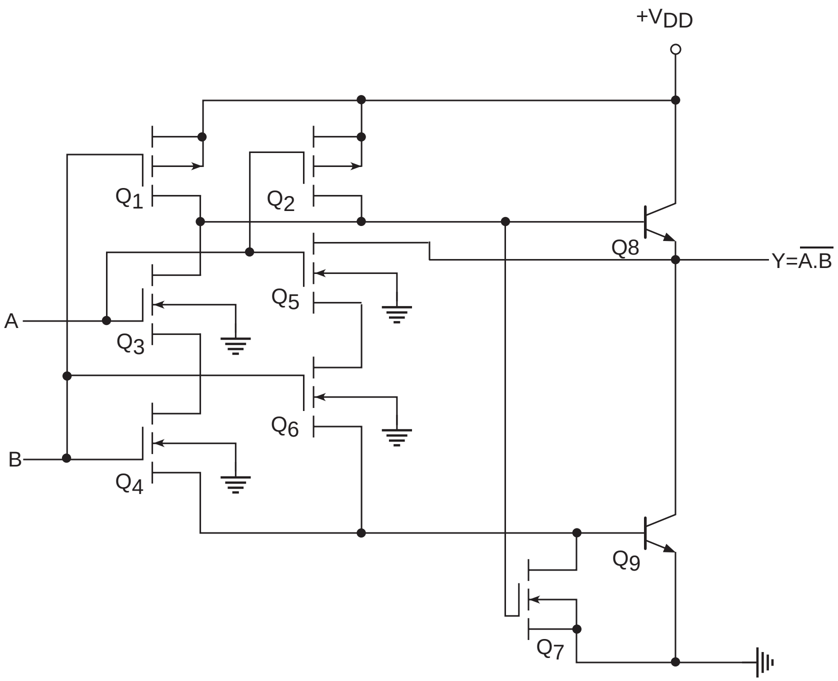


Figure 5.56 BiCMOS two-input NAND.

constitute a two-input NAND in CMOS. Also note the similarity of this circuit to the one shown in Fig. 5.55. The CMOS inverter stage of Fig. 5.55 is replaced by CMOS NAND in Fig. 5.56. N-channel MOSFET Q_3 in Fig. 5.55 is replaced by a series connection of N-channel MOSFETs Q_5 and Q_6 to accommodate the two inputs. The HIGH-state and LOW-state output voltage levels of this circuit are given by the equations

$$V_{OH} = (V_{DD} - 0.7) \tag{5.9}$$

$$V_{OL} = 0.7 \tag{5.10}$$

5.7 NMOS and PMOS Logic

Logic families discussed so far are the ones that are commonly used for implementing discrete logic functions such as logic gates, flip-flops, counters, multiplexers, demultiplexers, etc., in relatively less complex digital ICs belonging to the small-scale integration (SSI) and medium-scale integration (MSI) level of inner circuit complexities. The TTL, the CMOS and the ECL logic families are not suitable for implementing digital ICs that have a large-scale integration (LSI) level of inner circuit complexity and above. The competitors for LSI-class digital ICs are the PMOS, the NMOS and the integrated injection logic (I^2L). The first two are briefly discussed in this section, and the third is discussed in Section 5.8.

5.7.1 PMOS Logic

The PMOS logic family uses P-channel MOSFETS. Figure 5.57(a) shows an inverter circuit using PMOS logic. MOSFET Q_1 acts as an active load for the MOSFET switch Q_2. For the circuit shown, GND and $-V_{DD}$ respectively represent a logic '1' and a logic '0' for a positive logic system. When the input is grounded (i.e. logic '1'), Q_2 remains in cut-off and $-V_{DD}$ appears at the output through

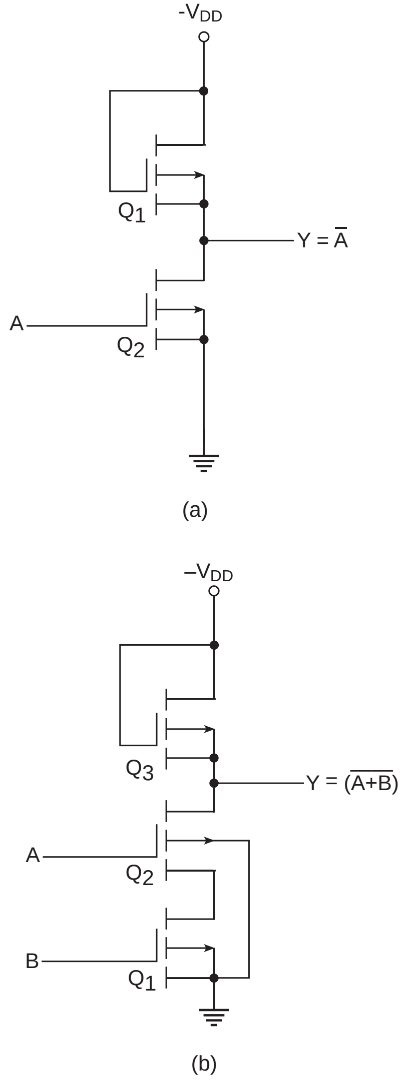


Figure 5.57 (a) PMOS logic inverter and (b) PMOS logic two-input NOR.

the conducting Q_1. When the input is at $-V_{DD}$ or near $-V_{DD}$, Q_2 conducts and the output goes to near-zero potential (i.e. logic '1').

Figure 5.57(b) shows a PMOS logic based two-input NOR gate. In the logic arrangement of Fig. 5.57(b), the output goes to logic '1' state (i.e. ground potential) only when both Q_1 and Q_2 are conducting. This is possible only when both the inputs are in logic '0' state. For all other possible input combinations, the output is in logic '0' state, because, with either Q_1 or Q_2 nonconducting, the output is nearly $-V_{DD}$ through the conducting Q_3. The circuit of Fig. 5.57(b) thus behaves like a two-input NOR gate in positive logic. It may be mentioned here that the MOSFET being used as load [Q_1 in Fig. 5.57(a) and Q_3 in Fig. 5.57(b)] is designed so as to have an ON-resistance that is much greater than the total ON-resistance of the MOSFETs being used as switches [Q_2 in Fig. 5.57(a) and Q_1 and Q_2 in Fig. 5.57(b)].

5.7.2 NMOS Logic

The NMOS logic family uses N-channel MOSFETS. N-channel MOS devices require a smaller chip area per transistor compared with P-channel devices, with the result that NMOS logic offers a higher density. Also, owing to the greater mobility of the charge carriers in N-channel devices, the NMOS logic family offers higher speed too. It is for this reason that most of the MOS memory devices and microprocessors employ NMOS logic or some variation of it such as VMOS, DMOS and HMOS. VMOS, DMOS and HMOS are only structural variations of NMOS, aimed at further reducing the propagation delay. Figures 5.58(a), (b) and (c) respectively show an inverter, a two-input NOR and a two-input NAND using NMOS logic. The logic circuits are self-explanatory.

5.8 Integrated Injection Logic (I^2L) Family

Integrated injection logic (I^2L), also known as current injection logic, is well suited to implementing LSI and VLSI digital functions and is a close competitor to the NMOS logic family. Figure 5.59 shows the basic I^2L family building block, which is a multicollector bipolar transistor with a current source driving its base. Transistors Q_3 and Q_4 constitute current sources. The magnitude of current depends upon externally connected R and applied $+V$. This current is also known as the injection current, which gives it its name of injection logic. If input A is HIGH, the injection current through Q_3 flows through the base-emitter junction of Q_1. Transistor Q_1 saturates and its collector drops to a low voltage, typically 50–100 mV. When A is LOW, the injection current is swept away from the base-emitter junction of Q_1. Transistor Q_1 becomes open and the injection current through Q_4 saturates Q_2, with the result that the Q_1 collector potential equals the base-emitter saturation voltage of Q_2, typically 0.7 V.

The speed of I^2L family devices is a function of the injection current I and improves with increase in current, as a higher current allows a faster charging of capacitive loads present at bases of transistors. The programmable injection current feature is made use of in the I^2L family of digital ICs to choose the desired speed depending upon intended application. The logic '0' level is V_{CE}(sat.) of the driving transistor (Q_1 in the present case), and the logic '1' level is V_{BE}(sat.) of the driven transistor (Q_2 in the present case). Typically, the logic '0' and logic '1' levels are 0.1 and 0.7 V respectively. The speed–power product of the I^2L family is typically under 1 pJ.

Multiple collectors of different transistors can be connected together to form wired logic. Figure 5.60 shows one such arrangement, depicting the generation of OR and NOR outputs of two logic variables A and B.

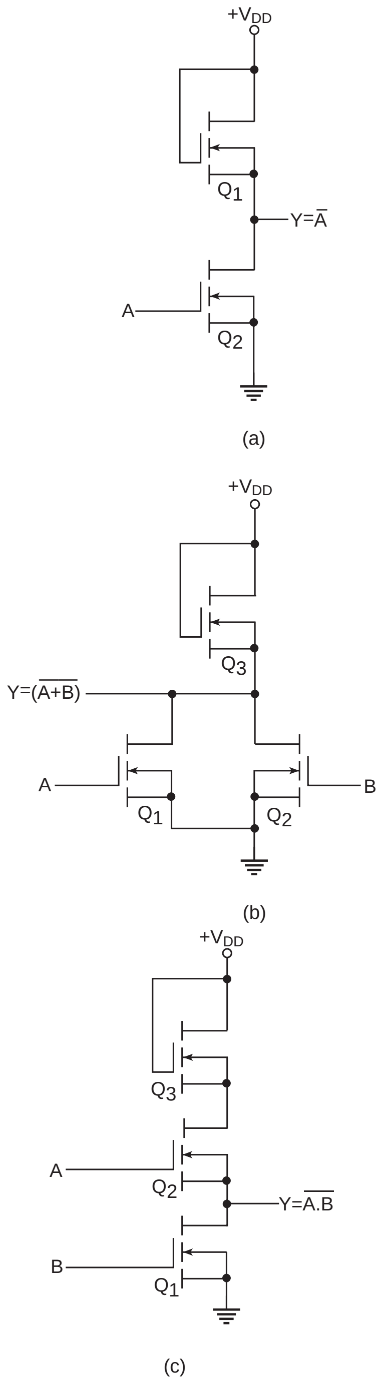


Figure 5.58 (a) NMOS logic circuit inverter, (b) NMOS logic two-input NOR and (c) NMOS logic two-input NAND.

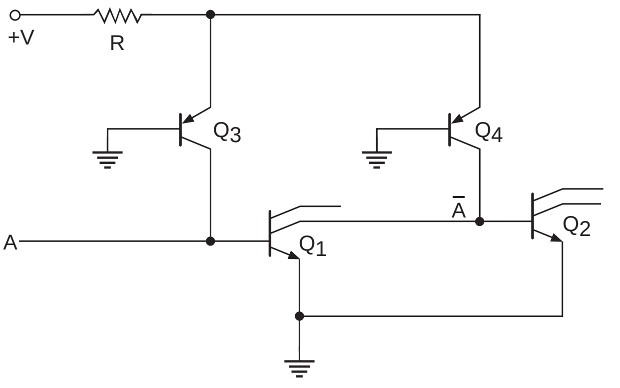


Figure 5.59 Integrated injection logic (I^2L).

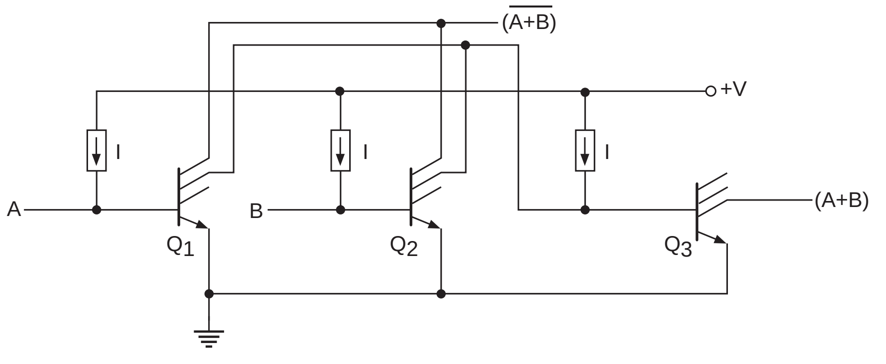


Figure 5.60 Wired logic in I^2L.

5.9 Comparison of Different Logic Families

Table 5.2 gives a comparison of various performance characteristics of important logic families for quick reference. The data given in the case of CMOS families are for $V_{DD} = 5$ V. In the case of ECL families, the data are for $V_{EE} = -5.2$ V. The values of various parameters given in the table should be used only for rough comparison. It is recommended that designers refer to the relevant data books for detailed information on these parameters along with the conditions under which those values are valid.

5.10 Guidelines to Using TTL Devices

The following guidelines should be adhered to while using TTL family devices:

1. Replacing a TTL IC of one TTL subfamily with another belonging to another subfamily (the type numbers remaining the same) should not be done blindly. The designer should ensure that

Table 5.2 Comparison of various performance characteristics of important logic families.

Logic family		Supply voltage (V)	Typical propagation delay (ns)	Worst-case noise margin (V)	Speed–power product (pJ)	Maximum flip-flop toggle frequency (MHz)
TTL	Standard	4.5 to 5.5	17	0.4	100	35
	L	4.5 to 5.5	60	0.3	33	3
	H	4.5 to 5.5	10	0.4	132	50
	S	4.5 to 5.5	5	0.3	57	125
	LS	4.5 to 5.5	15	0.3	18	45
	ALS	4.5 to 5.5	10	0.3	4.8	70
	AS	4.5 to 5.5	4.5	0.3	13.6	200
	F	4.5 to 5.5	6	0.3	10	125
CMOS	4000	3 to 15	150	1.0	5	12
	74C	3 to 13	50	1.4	5	12
	74HC	2 to –6	8	0.9	1.4	40
	74HCT	4.5 to 5.5	8	1.4	1.4	40
	74AC	2 to 6	4.7	0.7	0.37	100
	74ACT	4.5 to 5.5	4.7	0.72.9	0.37	100
ECL	MECL III	–5.1 to –5.3	1	0.2	60	500
	MECL 10K	–4.68 to –5.72	2.5	0.2	50	200
	MECL 10H	–4.94 to –5.46	1	0.15	25	250
	ECLINPS™	–4.2 to –5.5	0.5	0.15	10	1000
	ECLINPS LITE™	–4.2 to –5.5	0.2	0.15	10	2800

the replacement device is compatible with the existing circuit with respect to parameters such as output drive capability, input loading, speed and so on. As an illustration, let us assume that we are using 74S00 (quad two-input NAND), the output of which drives 20 different NAND inputs implemented using 74S00, as shown in Fig. 5.61. This circuit works well as the Schottky TTL family has a fan-out of 20 with an output HIGH drive capability of 1 mA and an input HIGH current requirement of 50 μA. If we try replacing the 74S00 driver with a 74LS00 driver, the circuit fails to work as 74LS00 NAND has an output HIGH drive capability of 0.4 mA only. It cannot feed 20 NAND input loads implemented using 74S00. By doing so, we will be exceeding the HIGH-state fan-out capability of the device. Also, 74LS00 has an output current-sinking specification of 8 mA, whereas the input current-sinking requirement of 74S00 is 2 mA. This implies that 74LS00 could reliably feed only four inputs of 74S00 in the LOW state. By feeding as many as 20 inputs, we will be exceeding the LOW-state fan-out capability of 74LS00 by a large margin.

2. None of the inputs and outputs of TTL ICs should be driven by more than 0.5 V below ground reference.
3. Proper grounding techniques should be used while designing the PCB layout. If the grounding is improper, the ground loop currents give rise to voltage drops, with the result that different ICs will not be at the same reference. This effectively reduces the noise immunity.

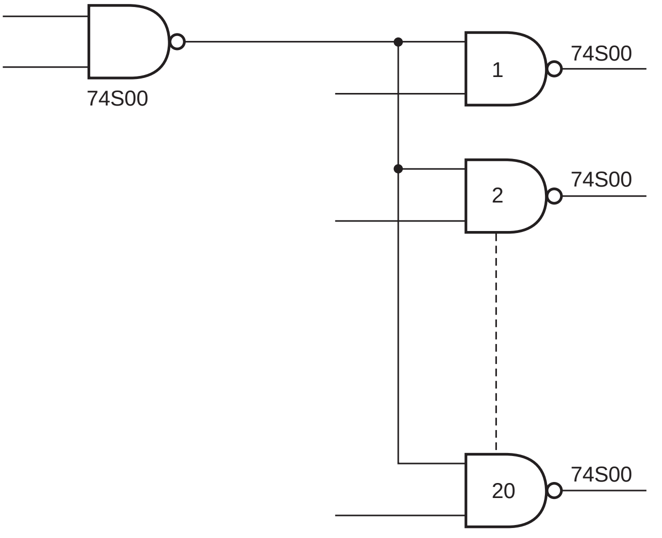


Figure 5.61 Output of one TTL subfamily driving another.

4. The power supply rail must always be properly decoupled with appropriate capacitors so that there is no drop in V_{CC} rail as the inputs and outputs make logic transitions. Usually, two capacitors are used at the V_{CC} point of each IC. A 0.1 μF ceramic disc should be used to take care of high-frequency noise, while typically a 10–20 μF electrolytic is good enough to eliminate any low-frequency variations resulting from variations in I_{CC} current drawn from V_{CC}, depending upon logic states of inputs and outputs. To be effective, the decoupling capacitors should be wired as close as feasible to the V_{CC} pin of the IC.
5. The unused inputs should not be left floating. All unused inputs should be tied to logic HIGH in the case of AND and NAND gates, and to ground in the case of OR and NOR gates. An alternative is to connect the unused input to one of the used inputs.
6. While using open collector devices, resistive pull-up should be used. The value of pull-up resistance should be determined from the following equations:

$$R_X = [V_{CC}(\text{max.}) - V_{OL}]/[I_{OL} - N_2(\text{LOW}) \times 1.6] \tag{5.11}$$

$$R_X(\text{max.}) = [V_{CC}(\text{min.}) - V_{OH}]/[N_1 \times I_{OH} + N_2(\text{HIGH}) \times 40] \tag{5.12}$$

where R_X is the external pull-up resistor; R_X(max.) is the maximum value of the external pull-up resistor; N_1 is the number of WIRED-OR outputs; N_2 is the number of unit input loads being driven; I_{OH} is the output HIGH leakage current (in mA); I_{OL} is the LOW-level output current of the driving element (in mA); V_{OL} is the LOW-level output voltage; and V_{OH} is the HIGH-level output voltage. One TTL unit load in the HIGH state = 40 mA, and one TTL unit load in the LOW-state = 1.6 mA.

5.11 Guidelines to Handling and Using CMOS Devices

The following guidelines should be adhered to while using CMOS family devices:

1. Proper handling of CMOS ICs before they are used and also after they have been mounted on the PC boards is very important as these ICs are highly prone to damage by electrostatic discharge. Although all CMOS ICs have inbuilt protection networks to guard them against electrostatic discharge, precautions should be taken to avoid such an eventuality. While handling unmounted chips, potential differences should be avoided. It is good practice to cover the chips with a conductive foil. Once the chips have been mounted on the PC board, it is good practice again to put conductive clips or conductive tape on the PC board terminals. Remember that PC board is nothing but an extension of the leads of the ICs mounted on it unless it is integrated with the overall system and proper voltages are present.
2. All unused inputs must always be connected to either V_{SS} or V_{DD} depending upon the logic involved. A floating input can result in a faulty logic operation. In the case of high-current device types such as buffers, it can also lead to the maximum power dissipation of the chip being exceeded, thus causing device damage. A resistor (typically 220 kΩ to 1 MΩ) should preferably be connected between input and the V_{SS} or V_{DD} if there is a possibility of device terminals becoming temporarily unconnected or open.
3. The recommended operating supply voltage ranges are 3–12 V for A-series (3–15 V being the maximum rating) and 3–15 V for B-series and UB-series (3–18 V being the maximum). For CMOS IC application circuits that are operated in a linear mode over a portion of the voltage range, such as RC or crystal oscillators, a minimum V_{DD} of 4 V is recommended.
4. Input signals should be maintained within the power supply voltage range $V_{SS} < V_i < V_{DD}$ (-0.5 V $< V_i < V_{DD} + 0.5$ V being the absolute maximum). If the input signal exceeds the recommended input signal range, the input current should be limited to ± 100 mA.
5. CMOS ICs like active pull-up TTL ICs cannot be connected in WIRE-OR configuration. Paralleling of inputs and outputs of gates is also recommended for ICs in the same package only.
6. The majority of CMOS clocked devices have maximum rise and fall time ratings of normally 5–15 μs. The device may not function properly with larger rise and fall times. The restriction, however, does not apply to those CMOS ICs that have inbuilt Schmitt trigger shaping in the clock circuit.

5.12 Interfacing with Different Logic Families

CMOS and TTL are the two most widely used logic families. Although ICs belonging to the same logic family have no special interface requirements, that is, the output of one can directly feed the input of the other, the same is not true if we have to interconnect digital ICs belonging to different logic families. Incompatibility of ICs belonging to different families mainly arises from different voltage levels and current requirements associated with LOW and HIGH logic states at the inputs and outputs. In this section, we will discuss simple interface techniques that can be used for CMOS-to-TTL and TTL-to-CMOS interconnections. Interface guidelines for CMOS–ECL, ECL–CMOS, TTL–ECL and ECL–TTL are also given.

5.12.1 CMOS-to-TTL Interface

The first possible type of CMOS-to-TTL interface is the one where both ICs are operated from a common supply. We have read in earlier sections that the TTL family has a recommended supply voltage of 5 V, whereas the CMOS family devices can operate over a wide supply voltage range of 3–18 V. In the present case, both ICs would operate from 5 V. As far as the voltage levels in the two logic states are concerned, the two have become compatible. The CMOS output has a V_{OH}(min.) of 4.95 V (for $V_{CC} = 5$ V) and a V_{OL}(max.) of 0.05 V, which is compatible with V_{IH}(min.) and V_{IL}(max.) requirements of approximately 2 and 0.8 V respectively for TTL family devices. In fact, in a CMOS-to-TTL interface, with the two devices operating on the same V_{CC}, voltage level compatibility is always there. It is the current level compatibility that needs attention. That is, in the LOW state, the output current-sinking capability of the CMOS IC in question must at least equal the input current-sinking requirement of the TTL IC being driven. Similarly, in the HIGH state, the HIGH output current drive capability of the CMOS IC must equal or exceed the HIGH-level input current requirement of TTL IC. For a proper interface, both the above conditions must be met. As a rule of thumb, a CMOS IC belonging to the 4000B family (the most widely used CMOS family) can feed one LS TTL or two low-power TTL unit loads. When a CMOS IC needs to drive a standard TTL or a Schottky TTL device, a CMOS buffer (4049B or 4050B) is used. 4049B and 4050B are hex buffers of inverting and noninverting types respectively, with each buffer capable of driving two standard TTL loads. Figure 5.62(a) shows a CMOS-to-TTL interface with both devices operating from 5 V supply and the CMOS IC driving a low-power TTL or a low-power Schottky TTL device. Figure 5.62(b) shows a CMOS-to-TTL interface where the TTL device in use is either a standard TTL or a Schottky TTL. The CMOS-to-TTL interface when the two are operating on different power supply voltages can be achieved in several ways. One such scheme is shown in Fig. 5.62(c). In this case, there is both a voltage level as well as a current level compatibility problem.

5.12.2 TTL-to-CMOS Interface

In the TTL-to-CMOS interface, current compatibility is always there. The voltage level compatibility in the two states is a problem. V_{OH} (min.) of TTL devices is too low as regards the V_{IH} (min.) requirement of CMOS devices. When the two devices are operating on the same power supply voltage, that is, 5 V, a pull-up resistor of 10 kΩ achieves compatibility [Fig. 5.63(a)]. The pull-up resistor causes the TTL output to rise to about 5 V when HIGH. When the two are operating on different power supplies, one of the simplest interface techniques is to use a transistor (as a switch) in-between the two, as shown in Fig. 5.63(b). Another technique is to use an open collector type TTL buffer [Fig. 5.63(c)].

5.12.3 TTL-to-ECL and ECL-to-TTL Interfaces

TTL-to-ECL and ECL-to-TTL interface connections are not as straightforward as TTL-to-CMOS and CMOS-to-TTL connections owing to widely different power supply requirements for the two types and also because ECL devices have differential inputs and differential outputs. Nevertheless, special chips are available that can take care of all these aspects. These are known as level translators. MC10124 is one such quad TTL-to-ECL level translator. That is, there are four independent single-input and complementary-output translators inside the chip. Figure 5.64(a) shows a TTL-to-ECL interface using MC10124.

MC10125 is a level translator for ECL-to-TTL interfaces; it has differential inputs and a single-ended output. Figure 5.64(b) shows a typical interface schematic using MC10125. Note that in the interface schematics of Figs 5.64(a) and (b), only one of the available four translators has been used.

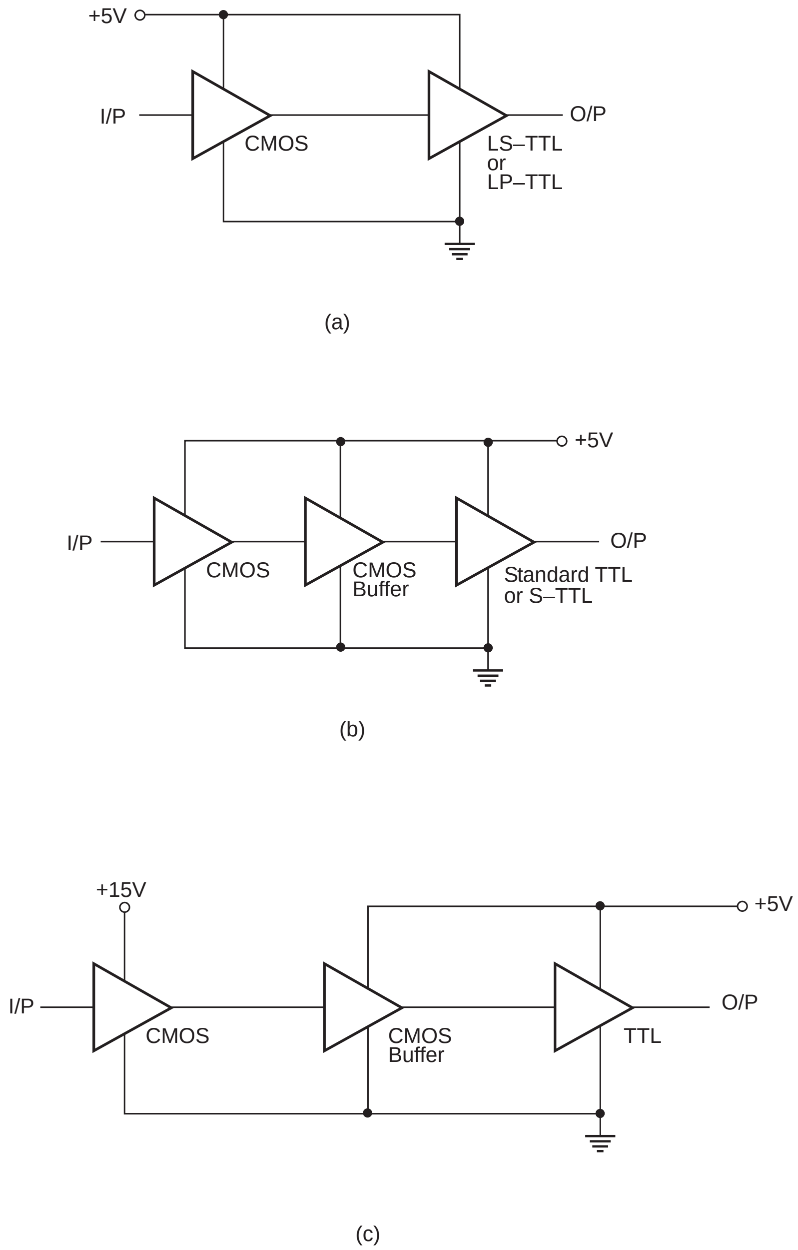


Figure 5.62 CMOS-to-TTL interface.

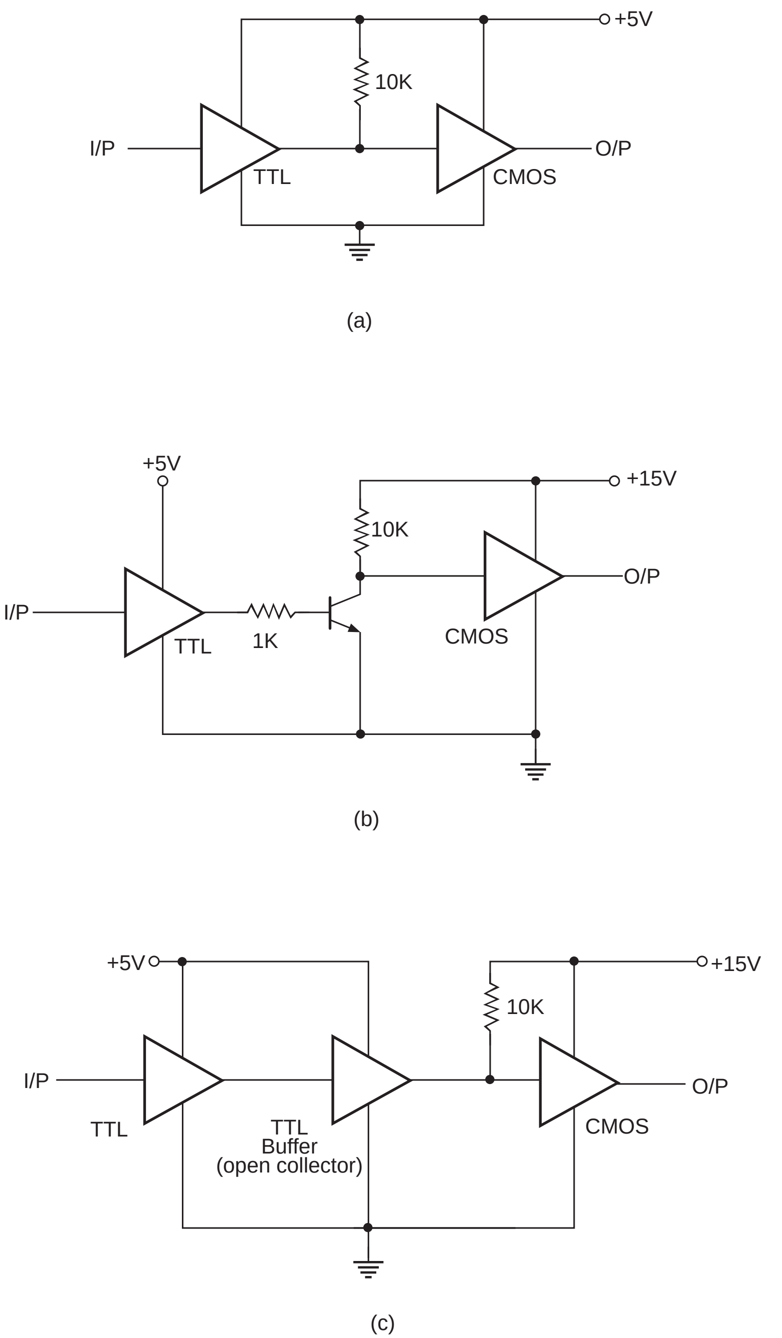


Figure 5.63 TTL-to-CMOS interface.

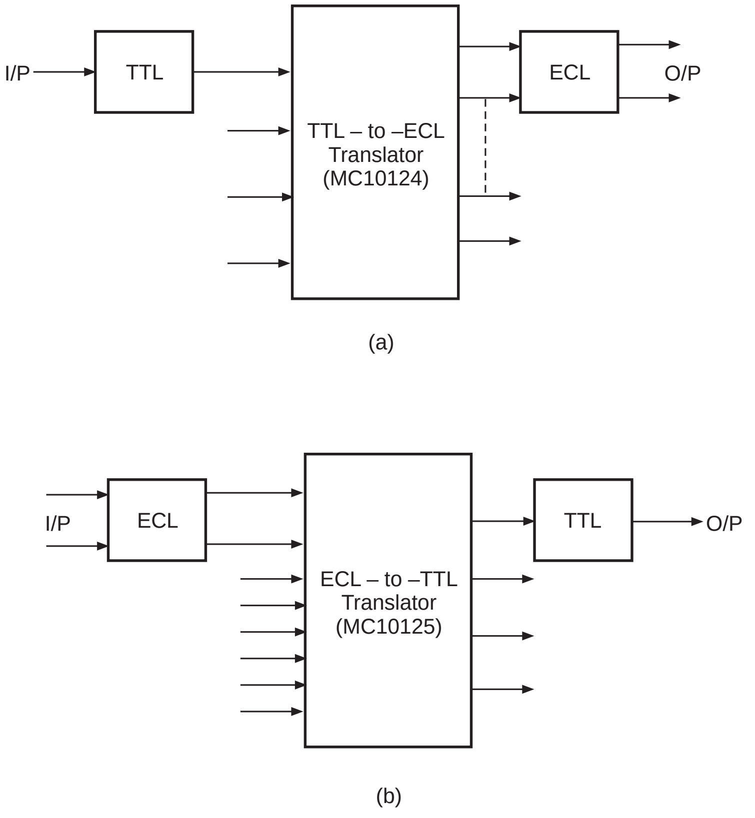


Figure 5.64 TTL-to-ECL and ECL-to-TTL interfaces.

5.12.4 CMOS-to-ECL and ECL-to-CMOS Interfaces

CMOS-to-ECL and ECL-to-CMOS interfaces are similar to the TTL-to-ECL and ECL-to-TTL interfaces described. Again, dedicated level translators are available. MC10352, for instance, is a quad CMOS-to-ECL level translator chip. A CMOS-to-ECL interface is also possible by having firstly a CMOS-to-TTL interface followed by a TTL-to-ECL interface using MC10124 or a similar chip. Figure 5.65(a) shows the arrangement. Similarly, an ECL-to-CMOS interface is possible by having an ECL-to-TTL interface using MC10125 or a similar chip followed by a TTL-to-CMOS interface. Figure 5.65(b) shows a typical interface schematic.

5.13 Classification of Digital ICs

We are all familiar with terms like SSI, MSI, LSI, VLSI and ULSI being used with reference to digital integrated circuits. These terms refer to groups in which digital ICs are divided on the basis of the complexity of the circuitry integrated on the chip. It is common practice to consider the complexity of

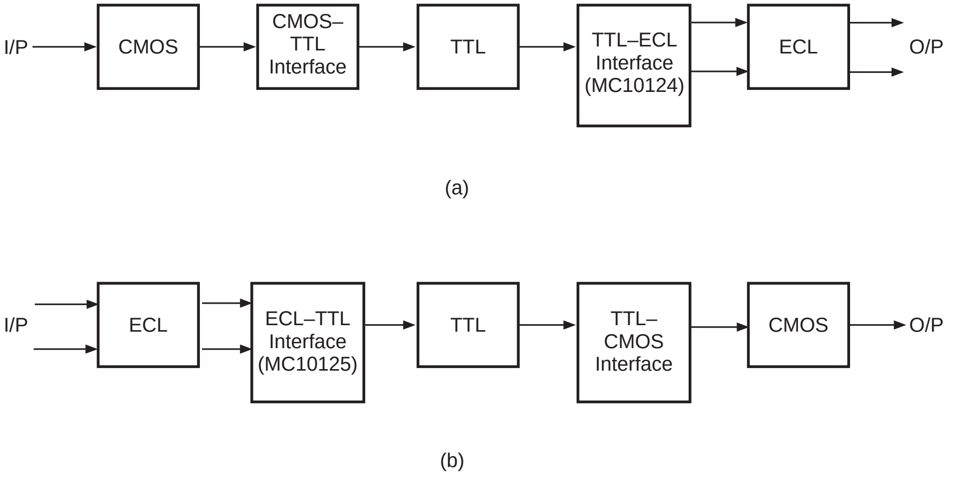


Figure 5.65 CMOS-to-ECL and ECL-to-CMOS interfaces.

a logic gate as a reference for defining the complexities of the other digital IC functions. A broadly accepted definition of different groups of ICs mentioned above is as follows.

A small-scale integration (SSI) chip is one that contains circuitry equivalent in complexity to less than or equal to 10 logic gates. This category of digital ICs includes basic logic gates and flip-flops. A medium-scale integration (MSI) chip is one that contains circuitry equivalent in complexity to 10–100 gates. This category of digital ICs includes multiplexers, demultiplexers, counters, registers, small memories, arithmetic circuits and others. A large-scale integration (LSI) chip is one that contains circuitry equivalent in complexity to 100–10 000 gates. A very-large-scale integration (VLSI) chip contains circuitry equivalent in complexity to 10 000–100 000 gates. Large-sized memories and microprocessors come in the category of LSI and VLSI chips. An ultralarge-scale integration (ULSI) chip contains circuitry equivalent in complexity to more than 100 000 gates. Very large memories, larger microprocessors and larger single-chip computers come into this category.

5.14 Application-Relevant Information

Table 5.3 lists the commonly used type numbers of level translator ICs, along with the functional description. The pin connection diagrams and functional tables for TTL-to-ECL level translator IC type MC10124 and ECL-to-TTL level translator IC type MC10125 are given in the companion website.

Table 5.3 Functional index of level translators

Type number	Function
10124	Quad TTL-to-ECL translator
10125	Quad ECL-to-TTL translator
10177	Triple ECL-to-CMOS translator
10352	Quad CMOS-to-ECL translator

Review Questions

1. What do you understand by the term logic family? What is the significance of the logic family with reference to digital integrated circuits (ICs)?
2. Briefly describe propagation delay, power dissipation, speed–power product, fan-out and noise margin parameters, with particular reference to their significance as regards the suitability of the logic family for a given application.
3. Compare the standard TTL, low-power Schottky TTL and Schottky TTL on the basis of speed, power dissipation and fan-out capability.
4. What is the totem-pole output stage? What are its advantages?
5. What are the basic differences between buffered and unbuffered CMOS devices? How is a buffered NAND usually implemented in 4000B-series CMOS logic?
6. With the help of relevant circuit schematics, briefly describe the operation of CMOS NAND and NOR gates.
7. Compare standard TTL and 4000B CMOS families on the basis of speed and power dissipation parameters.
8. Why is ECL called nonsaturating logic? What is the main advantage accruing from this? With the help of a relevant circuit schematic, briefly describe the operation of ECL OR/NOR logic.
9. What is the main criterion for the suitability of a logic family for use in fabricating LSI and VLSI logic functions? Name any two popular candidates and compare their features.
10. Why is it not recommended to leave unused logic inputs floating? What should we do to such inputs in the case of TTL and CMOS logic gates?
11. What special precautions should we observe in handling and using CMOS ICs?
12. With the help of suitable schematics, briefly describe how you would achieve TTL-to-CMOS and CMOS-to-TTL interfaces?
13. What is Bi-CMOS logic? What are its advantages?
14. What in a logic family decides the fan-out, speed of operation, noise immunity and power dissipation?

Problems

1. The data sheet of a quad two-input AND gate (type 74S08) specifies the propagation delay and power supply parameters as $V_{CC} = 5.0\,V$ (typical), I_{CCH} (for all four gates) $= 18$ mA, I_{CCL} (for all four gates) $= 32$ mA, $t_{pLH} = 4.5$ ns and $t_{pHL} = 5.0$ ns. Determine the speed–power product specification.

   148.4 pJ

2. How many inputs of a low-power Schottky TTL NAND can be reliably driven from a single output of a Schottky TTL NAND, given the following relevant specifications for the devices of two TTL subfamilies:
   Schottky TTL: $I_{OH} = 1.0$ mA; $I_{IH} = 0.05$ mA; $I_{OL} = 20.0$ mA; $I_{IL} = 2.0$ mA
   Low-power Schottky TTL: $I_{OH} = 0.4$ mA; $I_{IH} = 0.02$ mA; $I_{OL} = 8.0$ mA; $I_{IL} = 0.4$ mA

   50

3. Refer to the logic diagram in Fig. 5.66. Determine the current being sourced by the NAND gate when its output is HIGH and also the current sunk by it when its output is LOW, given that I_{IH} (AND gate) $= 0.02$ mA, I_{IL} (AND gate) $= 0.4$ mA, I_{IH} (OR gate) $= 0.04$ mA, I_{IL} (OR gate) $=$ 1.6 mA, I_{OH}(NAND gate) $= 1.0$ mA, I_{OL}(NAND gate) $= 20.0$ mA.

   HIGH-state current = 0.08 mA; LOW-state current = 2.0 mA

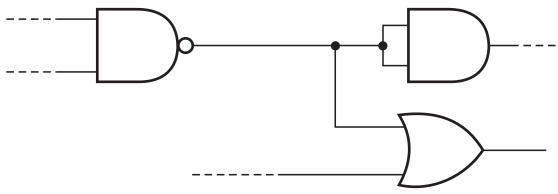


Figure 5.66 Problem 3.

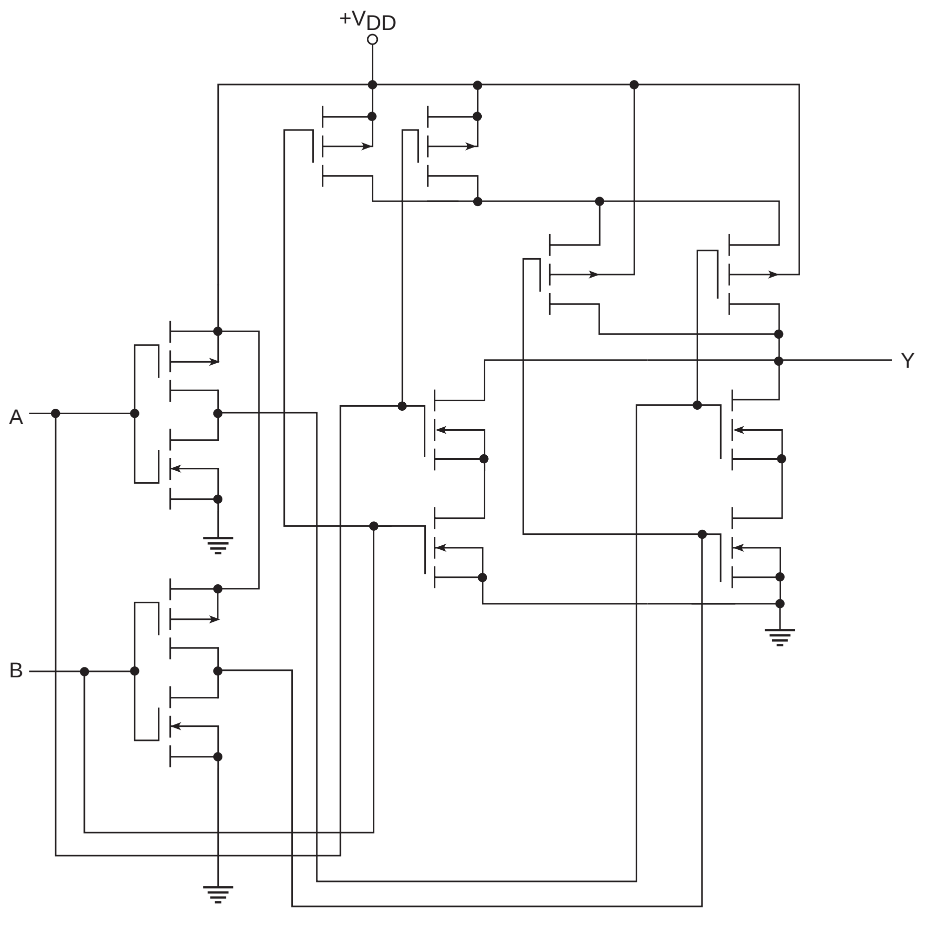


Figure 5.67 Problem 5.

4. Write the logic expression for the CMOS circuit of Fig. 5.67.

$$Y = (A.B + \overline{A}.\overline{B})$$

5. Refer to the data given for 4000B-series CMOS, 74LS-TTL and 74HCT CMOS logic. Determine:

(a) the number of 74LS-TTL inputs that can be reliably driven from a single 4000B output;
(b) the number of 74LS-TTL inputs that can be reliably driven from a single 74HCT output.
4000B: $I_{OH} = 0.4\,\text{mA}$; $I_{IH} = 1.0\,\mu\text{A}$; $I_{OL} = 0.4\,\mu\text{A}$; $I_{IL} = 1.0\,\mu\text{A}$
74HCT: $I_{OH} = 4.0\,\text{mA}$; $I_{IH} = 1.0\,\mu\text{A}$; $I_{OL} = 4.0\,\mu\text{A}$; $I_{IL} = 1.0\,\mu\text{A}$
74LS-TTL: $I_{OH} = 0.4\,\text{mA}$; $I_{IH} = 20.0\,\mu\text{A}$; $I_{OL} = 8.0\,\mu\text{A}$; $I_{IL} = 0.4\,\text{mA}$

(a) 1; (b) 10

Further Reading

1. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
2. Demassa, T. A. and Ciccone, Z. (1995) *Digital Integrated Circuits,* John Wiley & Sons, Inc., New York, USA.
3. Fairchild Semiconductor (August 1973) *74C Family Characteristics*, Application Note 90, South Portland, ME, USA.
4. Wakeman, L. (April 1998) *DC Electrical Characteristics of MM74HC High-speed CMOS Logic*, Application Note 313, Fairchild Semiconductor, South Portland, ME, USA.
5. Funk, R. E. (October 2002) *Understanding Buffered and Unbuffered CD4XXXB-series Device Characteristics*, Application Report SCHA004, Texas Instruments, USA.
6. Buchanan, J. E. and Buchanan, B. D. (1995) *Signal and Power Integrity in Digital Systems: TTL, CMOS, and BiCMOS,* McGraw-Hill Companies, NJ, USA.
7. Lancaster, D. E. (1997) *CMOS Cookbook*, Butterworth-Heinemann, USA.
8. Elmasry, M. I. (1994) *BiCMOS Integrated Circuit Design,* IEEE Press, USA.

6

Boolean Algebra and Simplification Techniques

Boolean algebra is mathematics of logic. It is one of the most basic tools available to the logic designer and thus can be effectively used for simplification of complex logic expressions. Other useful and widely used techniques based on Boolean theorems include the use of Karnaugh maps in what is known as the mapping method of logic simplification and the tabular method given by Quine–McCluskey. In this chapter, we will have a closer look at the different postulates and theorems of Boolean algebra and their applications in minimizing Boolean expressions. We will also discuss at length the mapping and tabular methods of minimizing fairly complex and large logic expressions.

6.1 Introduction to Boolean Algebra

Boolean algebra, quite interestingly, is simpler than ordinary algebra. It is also composed of a set of symbols and a set of rules to manipulate these symbols. However, this is the only similarity between the two. The differences are many. These include the following:

1. In ordinary algebra, the letter symbols can take on any number of values including infinity. In Boolean algebra, they can take on either of two values, that is, 0 and 1.
2. The values assigned to a variable have a numerical significance in ordinary algebra, whereas in its Boolean counterpart they have a logical significance.
3. While '.' and '+' are respectively the signs of multiplication and addition in ordinary algebra, in Boolean algebra '.' means an AND operation and '+' means an OR operation. For instance, $A+B$ in ordinary algebra is read as A plus B, while the same in Boolean algebra is read as A OR B. Basic logic operations such as AND, OR and NOT have already been discussed at length in Chapter 4.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

4. More specifically, Boolean algebra captures the essential properties of both logic operations such as AND, OR and NOT and set operations such as intersection, union and complement. As an illustration, the logical assertion that both a statement and its negation cannot be true has a counterpart in set theory, which says that the intersection of a subset and its complement is a null (or empty) set.
5. Boolean algebra may also be defined to be a set A supplied with two binary operations of logical AND (Λ), logical OR (V), a unary operation of logical NOT ($\neg$) and two elements, namely logical FALSE (0) and logical TRUE (1). This set is such that, for all elements of this set, the postulates or axioms relating to the associative, commutative, distributive, absorption and complementation properties of these elements hold good. These postulates are described in the following pages.

6.1.1 Variables, Literals and Terms in Boolean Expressions

Variables are the different symbols in a Boolean expression. They may take on the value '0' or '1'. For instance, in expression (6.1), A, B and C are the three variables. In expression (6.2), P, Q, R and S are the variables:

$$\overline{A}+A.B+A.\overline{C}+\overline{A}.B.C \tag{6.1}$$

$$(\overline{P}+Q).(R+\overline{S}).(P+\overline{Q}+R) \tag{6.2}$$

The complement of a variable is not considered as a separate variable. Each occurrence of a variable or its complement is called a *literal*. In expressions (6.1) and (6.2) there are eight and seven literals respectively. A term is the expression formed by literals and operations at one level. Expression (6.1) has five terms including four AND terms and the OR term that combines the first-level AND terms.

6.1.2 Equivalent and Complement of Boolean Expressions

Two given Boolean expressions are said to be *equivalent* if one of them equals '1' only when the other equals '1' and also one equals '0' only when the other equals '0'. They are said to be the *complement* of each other if one expression equals '1' only when the other equals '0', and vice versa. The complement of a given Boolean expression is obtained by complementing each literal, changing all '.' to '+' and all '+' to '.', all 0s to 1s and all 1s to 0s. The examples below give some Boolean expressions and their complements:

Given Boolean expression

$$\overline{A}.B+A.\overline{B} \tag{6.3}$$

Corresponding complement

$$(A+\overline{B}).(\overline{A}+B) \tag{6.4}$$

Given Boolean expression

$$(A+B).(\overline{A}+\overline{B}) \tag{6.5}$$

Corresponding complement

$$\overline{A}.\overline{B}+A.B \tag{6.6}$$

When ORed with its complement the Boolean expression yields a '1', and when ANDed with its complement it yields a '0'. The '.' sign is usually omitted in writing Boolean expressions and is implied merely by writing the literals in juxtaposition. For instance, $A.B$ would normally be written as AB.

6.1.3 Dual of a Boolean Expression

The dual of a Boolean expression is obtained by replacing all '.' operations with '+' operations, all '+' operations with '.' operations, all 0s with 1s and all 1s with 0s and leaving all literals unchanged. The examples below give some Boolean expressions and the corresponding dual expressions:

Given Boolean expression

$$\overline{A}.B+A.\overline{B} \tag{6.7}$$

Corresponding dual

$$(\overline{A}+B).(A+\overline{B}) \tag{6.8}$$

Given Boolean expression

$$(A+B).(\overline{A}+\overline{B}) \tag{6.9}$$

Corresponding dual

$$A.B+\overline{A}.\overline{B} \tag{6.10}$$

Duals of Boolean expressions are mainly of interest in the study of Boolean postulates and theorems. Otherwise, there is no general relationship between the values of dual expressions. That is, both of them may equal '1' or '0'. One may even equal '1' while the other equals '0'. The fact that the dual of a given logic equation is also a valid logic equation leads to many more useful laws of Boolean algebra. The principle of duality has been put to ample use during the discussion on postulates and theorems of Boolean algebra. The postulates and theorems, to be discussed in the paragraphs to follow, have been presented in pairs, with one being the dual of the other.

Example 6.1

Find (a) the dual of $A.\overline{B}+B.\overline{C}+C.\overline{D}$ *and (b) the complement of* $[(A.\overline{B}+\overline{C}).D+\overline{E}].F$.

Solution

(a) The dual of $A.\overline{B}+B.\overline{C}+C.\overline{D}$ is given by $(A+\overline{B}).(B+\overline{C}).(C+\overline{D})$.
(b) The complement of $[(A.\overline{B}+\overline{C}).D+\overline{E}].F$ is given by $[(\overline{A}+B).C+\overline{D}].E+\overline{F}$.

Example 6.2

Simplify $(A.B+C.D).[(\overline{A}+\overline{B}).(\overline{C}+\overline{D})]$.

Solution

- Let $(A.B+C.D)=X$.
- Then the given expression reduces to $X.\overline{X}$.
- Therefore, $(A.B+C.D).[(\overline{A}+\overline{B}).(\overline{C}+\overline{D})]=0$.

6.2 Postulates of Boolean Algebra

The following are the important postulates of Boolean algebra:

1. $1.1=1, 0+0=0$.
2. $1.0=0.1=0, 0+1=1+0=1$.
3. $0.0=0, 1+1=1$.
4. $\overline{1}=0$ and $\overline{0}=1$.

Many theorems of Boolean algebra are based on these postulates, which can be used to simplify Boolean expressions. These theorems are discussed in the next section.

6.3 Theorems of Boolean Algebra

The theorems of Boolean algebra can be used to simplify many a complex Boolean expression and also to transform the given expression into a more useful and meaningful equivalent expression. The theorems are presented as pairs, with the two theorems in a given pair being the dual of each other. These theorems can be very easily verified by the method of 'perfect induction'. According to this method, the validity of the expression is tested for all possible combinations of values of the variables involved. Also, since the validity of the theorem is based on its being true for all possible combinations of values of variables, there is no reason why a variable cannot be replaced with its complement, or vice versa, without disturbing the validity. Another important point is that, if a given expression is valid, its dual will also be valid. Therefore, in all the discussion to follow in this section, only one of the theorems in a given pair will be illustrated with a proof. Proof of the other being its dual is implied.

6.3.1 Theorem 1 (Operations with '0' and '1')

$$\text{(a) } 0.X=0 \quad \text{and} \quad \text{(b) } 1+X=1 \tag{6.11}$$

where X is not necessarily a single variable – it could be a term or even a large expression.

Theorem 1(a) can be proved by substituting all possible values of X, that is, 0 and 1, into the given expression and checking whether the LHS equals the RHS:

- For $X=0$, LHS $=0.X=0.0=0=$ RHS.
- For $X=1$, LHS $=0.1=0=$ RHS.

Thus, $0.X=0$ irrespective of the value of X, and hence the proof.

Theorem 1(b) can be proved in a similar manner. In general, according to theorem 1, 0.(Boolean expression) = 0 and 1 + (Boolean expression) = 1. For example, $0.(A.B+B.C+C.D)=0$ and $1+(A.B+B.C+C.D)=1$, where A, B and C are Boolean variables.

6.3.2 Theorem 2 (Operations with '0' and '1')

$$\text{(a) } 1.X = X \quad \text{and} \quad \text{(b) } 0 + X = X \tag{6.12}$$

where X could be a variable, a term or even a large expression. According to this theorem, ANDing a Boolean expression to '1' or ORing '0' to it makes no difference to the expression:

- For $X = 0$, LHS $= 1.0 = 0 =$ RHS.
- For $X = 1$, LHS $= 1.1 = 1 =$ RHS.

Also, 1.(Boolean expression) = Boolean expression and 0 + (Boolean expression) = Boolean expression. For example,

$$1.\ (A + B.C + C.D) = 0 + (A + B.C + C.D) = A + B.C + C.D.$$

6.3.3 Theorem 3 (Idempotent or Identity Laws)

$$\text{(a) } X.X.X.\ldots.X = X \quad \text{and} \quad \text{(b)} X + X + X + \cdots + X = X \tag{6.13}$$

Theorems 3(a) and (b) are known by the name of *idempotent laws*, also known as *identity laws*. Theorem 3(a) is a direct outcome of an AND gate operation, whereas theorem 3(b) represents an OR gate operation when all the inputs of the gate have been tied together. The scope of idempotent laws can be expanded further by considering X to be a term or an expression. For example, let us apply idempotent laws to simplify the following Boolean expression:

$$\begin{aligned}(A.\overline{B}.\overline{B} + C.C).(A.\overline{B}.\overline{B} + A.\overline{B} + C.C) &= (A.\overline{B} + C).(A.\overline{B} + A.\overline{B} + C)\\ &= (A.\overline{B} + C).(A.\overline{B} + C) = A.\overline{B} + C\end{aligned}$$

6.3.4 Theorem 4 (Complementation Law)

$$\text{(a) } X.\overline{X} = 0 \quad \text{and} \quad \text{(b) } X + \overline{X} = 1 \tag{6.14}$$

According to this theorem, in general, any Boolean expression when ANDed to its complement yields a '0' and when ORed to its complement yields a '1', irrespective of the complexity of the expression:

- For $X = 0$, $\overline{X} = 1$. Therefore, $X.\overline{X} = 0.1 = 0$.
- For $X = 1$, $\overline{X} = 0$. Therefore, $X.\overline{X} = 1.0 = 0$.

Hence, theorem 4(a) is proved. Since theorem 4(b) is the dual of theorem 4(a), its proof is implied.
The example below further illustrates the application of complementation laws:

$$(A + B.C)(\overline{A + B.C}) = 0 \quad \text{and} \quad (A + B.C) + (\overline{A + B.C}) = 1$$

Example 6.3

Simplify the following:

$$[1+L.M+L.\overline{M}+\overline{L}.M].[(L+\overline{M}).(\overline{L}.M)+\overline{L}.\overline{M}.(L+M)].$$

Solution

- We know that (1 + Boolean expression) = 1.
- Also, $(\overline{L}.M)$ is the complement of $(L+\overline{M})$ and $(\overline{L}.\overline{M})$ is the complement of $(L+M)$.
- Therefore, the given expression reduces to 1.(0 + 0) = 1.0 = 0.

6.3.5 Theorem 5 (Commutative Laws)

$$\text{(a) } X+Y=Y+X \quad \text{and} \quad \text{(b) } X.Y=Y.X \tag{6.15}$$

Theorem 5(a) implies that the order in which variables are added or ORed is immaterial. That is, the result of A OR B is the same as that of B OR A. Theorem 5(b) implies that the order in which variables are ANDed is also immaterial. The result of A AND B is same as that of B AND A.

6.3.6 Theorem 6 (Associative Laws)

$$\text{(a) } X+(Y+Z)=Y+(Z+X)=Z+(X+Y)$$

and

$$\text{(b) } X.(Y.Z)=Y.(Z.X)=Z.(X.Y) \tag{6.16}$$

Theorem 6(a) says that, when three variables are being ORed, it is immaterial whether we do this by ORing the result of the first and second variables with the third variable or by ORing the first variable with the result of ORing of the second and third variables or even by ORing the second variable with the result of ORing of the first and third variables. According to theorem 6(b), when three variables are being ANDed, it is immaterial whether you do this by ANDing the result of ANDing of the first and second variables with the third variable or by ANDing the result of ANDing of the second and third variables with the first variable or even by ANDing the result of ANDing of the third and first variables with the second variable.

For example,

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

Also

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

Theorems 6(a) and (b) are further illustrated by the logic diagrams in Figs 6.1(a) and (b).

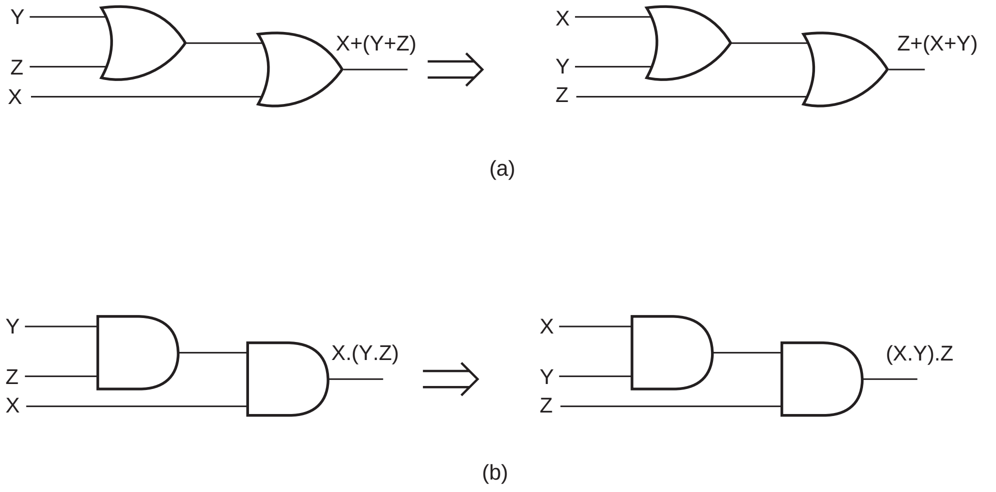


Figure 6.1 Associative laws.

6.3.7 Theorem 7 (Distributive Laws)

$$\text{(a) } X.(Y+Z) = X.Y + X.Z \quad \text{and} \quad \text{(b) } X + Y.Z = (X+Y).(X+Z) \tag{6.17}$$

Theorem 7(b) is the dual of theorem 7(a). The distribution law implies that a Boolean expression can always be expanded term by term. Also, in the case of the expression being the sum of two or more than two terms having a common variable, the common variable can be taken as common as in the case of ordinary algebra. Table 6.1 gives the proof of theorem 7(a) using the method of perfect induction. Theorem 7(b) is the dual of theorem 7(a) and therefore its proof is implied. Theorems 7(a) and (b) are further illustrated by the logic diagrams in Figs 6.2(a) and (b). As an illustration, theorem 7(a) can be used to simplify $\overline{A}.\overline{B} + \overline{A}.B + A.\overline{B} + A.B$ as follows:

$$\overline{A}.\overline{B} + \overline{A}.B + A.\overline{B} + A.B = \overline{A}.(\overline{B} + B) + A.(\overline{B} + B) = \overline{A}.1 + A.1 = \overline{A} + A = 1$$

Table 6.1 Proof of distributive law.

X	Y	Z	Y+Z	XY	XZ	X(Y+Z)	XY+XZ
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

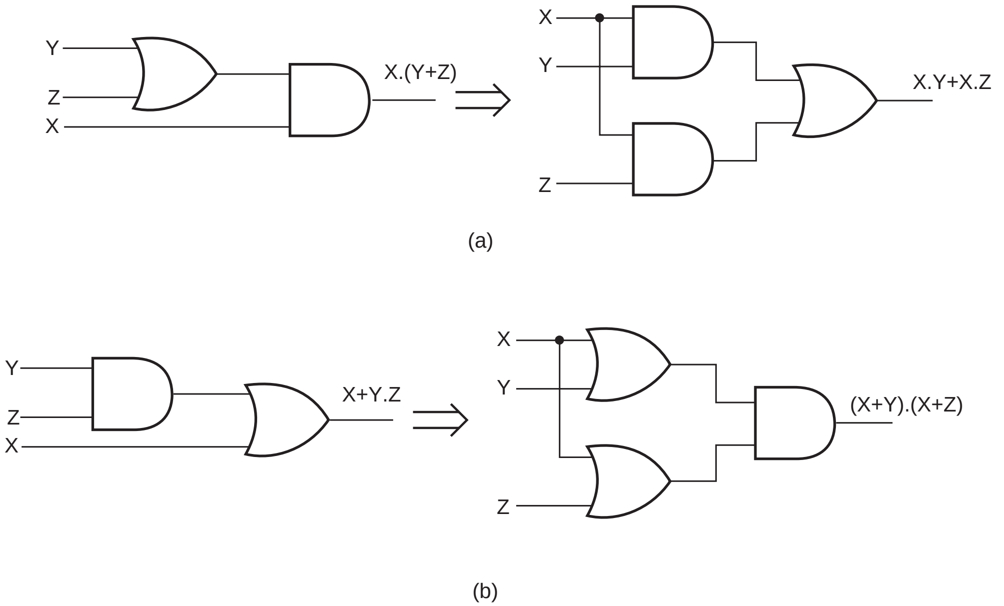


Figure 6.2 Distributive laws.

Theorem 7(b) can be used to simplify $(\overline{A}+\overline{B}).(\overline{A}+B).(A+\overline{B}).(A+B)$ as follows:

$$(\overline{A}+\overline{B}).(\overline{A}+B).(A+\overline{B}).(A+B) = (\overline{A}+\overline{B}.B).(A+\overline{B}.B) = (\overline{A}+0).(A+0) = \overline{A}.A = 0$$

6.3.8 Theorem 8

$$\text{(a) } X.Y + X.\overline{Y} = X \quad \text{and} \quad \text{(b) } (X+Y).(X+\overline{Y}) = X$$

This is a special case of theorem 7 as

$$X.Y + X.\overline{Y} = X.(Y+\overline{Y}) = X.1 = X \quad \text{and} \quad (X+Y).(X+\overline{Y}) = X + Y.\overline{Y} = X + 0 = X$$

This theorem, however, has another very interesting interpretation. Referring to theorem 8(a), there are two two-variable terms in the LHS expression. One of the variables, Y, is present in all possible combinations in this expression, while the other variable, X, is a common factor. The expression then reduces to this common factor. This interpretation can be usefully employed to simplify many a complex Boolean expression.

As an illustration, let us consider the following Boolean expression:

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

In the above expression, variables B, C and D are present in all eight possible combinations, and variable A is the common factor in all eight product terms. With the application of theorem 8(a), this expression reduces to A. Similarly, with the application of theorem 8(b), $(A+\overline{B}+\overline{C}).(A+\overline{B}+C).(A+B+\overline{C}).(A+B+C)$ also reduces to A as the variables B and C are present in all four possible combinations in sum terms and variable A is the common factor in all the terms.

6.3.9 Theorem 9

$$\text{(a) } (X+\overline{Y}).Y = X.Y \quad \text{and} \quad \text{(b) } X.\overline{Y}+Y = X+Y \tag{6.18}$$

$$(X+\overline{Y}).Y = X.Y+\overline{Y}.Y = X.Y$$

Theorem 9(b) is the dual of theorem 9(a) and hence stands proved.

6.3.10 Theorem 10 (Absorption Law or Redundancy Law)

$$\text{(a) } X+X.Y = X \quad \text{and} \quad \text{(b) } X.(X+Y) = X \tag{6.19}$$

The proof of absorption law is straightforward:

$$X+X.Y = X.(1+Y) = X.1 = X$$

Theorem 10(b) is the dual of theorem 10(a) and hence stands proved.

The crux of this simplification theorem is that, if a smaller term appears in a larger term, then the larger term is redundant. The following examples further illustrate the underlying concept:

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

and

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

6.3.11 Theorem 11

$$\text{(a) } Z.X+Z.\overline{X}.Y = Z.X+Z.Y$$

and

$$\text{(b) } (Z+X).(Z+\overline{X}+Y) = (Z+X).(Z+Y) \tag{6.20}$$

Table 6.2 gives the proof of theorem 11(a) using the method of perfect induction. Theorem 11(b) is the dual of theorem 11(a) and hence stands proved. A useful interpretation of this theorem is that, when

Table 6.2 Proof of theorem 11(a).

X	Y	Z	ZX	ZY	$Z\overline{X}$	$Z\overline{X}Y$	$ZX+Z\overline{X}Y$	ZX+ZY
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	1	1
1	1	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1

a smaller term appears in a larger term except for one of the variables appearing as a complement in the larger term, the complemented variable is redundant.

As an example, $(A+\overline{B}).(\overline{A}+\overline{B}+C).(\overline{A}+\overline{B}+D)$ can be simplified as follows:

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

6.3.12 Theorem 12 (Consensus Theorem)

$$\text{(a) } X.Y+\overline{X}.Z+Y.Z=X.Y+\overline{X}.Z$$

and

$$\text{(b) } (X+Y).(\overline{X}+Z).(Y+Z)=(X+Y).(\overline{X}+Z) \quad (6.21)$$

Table 6.3 shows the proof of theorem 12(a) using the method of perfect induction. Theorem 12(b) is the dual of theorem 12(a) and hence stands proved.

A useful interpretation of theorem 12 is as follows. If in a given Boolean expression we can identify two terms with one having a variable and the other having its complement, then the term that is formed by the product of the remaining variables in the two terms in the case of a sum-of-products expression

Table 6.3 Proof of theorem 12(a).

X	Y	Z	XY	$\overline{X}Z$	YZ	$XY+\overline{X}Z+YZ$	$XY+\overline{X}Z$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	0	1	1	1

or by the sum of the remaining variables in the case of a product-of-sums expression will be redundant. The following example further illustrates the point:

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

If we consider the first two terms of the Boolean expression, $B.C.D$ becomes redundant. If we consider the first and third terms of the given Boolean expression, $A.C.D$ becomes redundant.

Example 6.4

Prove that $A.B.C.D+A.B.\overline{C}.\overline{D}+A.B.C.\overline{D}+A.B.\overline{C}.D+A.B.C.D.E+A.B.\overline{C}.\overline{D}.\overline{E}+A.B.\overline{C}.D.E$ *can be simplified to* $A.B$.

Solution

$$A.B.C.D+A.B.\overline{C}.\overline{D}+A.B.C.\overline{D}+A.B.\overline{C}.D+A.B.C.D.E+A.B.\overline{C}.\overline{D}.\overline{E}+A.B.\overline{C}.D.E$$

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

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

- $A.B.C.D$ appears in $A.B.C.D.E$, $A.B.\overline{C}.\overline{D}$ appears in $A.B.\overline{C}.\overline{D}.\overline{E}$ and $A.B.\overline{C}.D$ appears in $A.B.\overline{C}.D.E$.
- As a result, all three five-variable terms are redundant.
- Also, variables C and D appear in all possible combinations and are therefore redundant.

6.3.13 Theorem 13 (DeMorgan's Theorem)

$$\text{(a) } [\overline{X_1+X_2+X_3+\ldots+X_n}]=\overline{X_1}.\overline{X_2}.\overline{X_3}.\ldots.\overline{X_n} \tag{6.22}$$

$$\text{(b) } [\overline{X_1.X_2.X_3.\ldots.X_n}]=[\overline{X_1}+\overline{X_2}+\overline{X_3}+\ldots+\overline{X_n}] \tag{6.23}$$

According to the first theorem the complement of a sum equals the product of complements, while according to the second theorem the complement of a product equals the sum of complements. Figures 6.3(a) and (b) show logic diagram representations of De Morgan's theorems. While the first theorem can be interpreted to say that a multi-input NOR gate can be implemented as a multi-input bubbled AND gate, the second theorem, which is the dual of the first, can be interpreted to say that a multi-input NAND gate can be implemented as a multi-input bubbled OR gate.

DeMorgan's theorem can be proved as follows. Let us assume that all variables are in a logic '0' state. In that case

$$\text{LHS}=[\overline{X_1+X_2+X_3+\cdots+X_n}]=[\overline{0+0+0+\cdots+0}]=\overline{0}=1$$

$$\text{RHS}=\overline{X_1}.\overline{X_2}.\overline{X_3}.\ldots.\overline{X_n}=\overline{0}.\overline{0}.\overline{0}.\ldots.\overline{0}=1.1.1.\ldots.1=1$$

Therefore, LHS = RHS.

Now, let us assume that any one of the n variables, say X_1, is in a logic HIGH state:

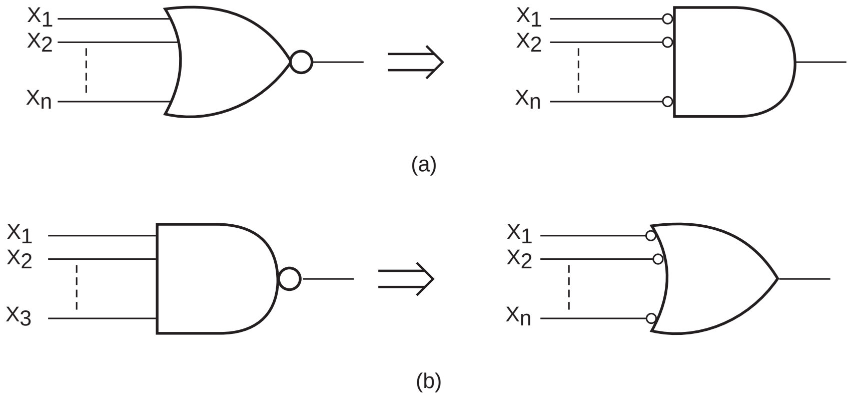


Figure 6.3 DeMorgan's theorem.

$$\text{LHS} = [\overline{X_1 + X_2 + X_3 + \cdots X_n}] = [\overline{1 + 0 + 0 + \cdots + 0}] = \overline{1} = 0$$

$$\text{RHS} = \overline{X_1}.\overline{X_2}.\overline{X_3}. \ldots .\overline{X_n} = \overline{1}.\overline{0}.\overline{0}. \ldots .\overline{0} = 0.1.1. \ldots .1 = 0$$

Therefore, again LHS = RHS.

The same holds good when more than one or all variables are in the logic '1' state. Therefore, theorem 13(a) stands proved. Since theorem 13(b) is the dual of theorem 13(a), the same also stands proved. Theorem 13(b), though, can be proved on similar lines.

6.3.14 Theorem 14 (Transposition Theorem)

$$\text{(a) } X.Y + \overline{X}.Z = (X + Z).(\overline{X} + Y)$$

and

$$\text{(b) } (X + Y).(\overline{X} + Z) = X.Z + \overline{X}.Y \tag{6.24}$$

This theorem can be applied to any sum-of-products or product-of-sums expression having two terms, provided that a given variable in one term has its complement in the other. Table 6.4 gives the proof of theorem 14(a) using the method of perfect induction. Theorem 14(b) is the dual of theorem 14(a) and hence stands proved.

As an example,

$$\overline{A}.B + A.\overline{B} = (A + B).(\overline{A} + \overline{B}) \quad \text{and} \quad A.B + \overline{A}.\overline{B} = (A + \overline{B}).(\overline{A} + B)$$

Incidentally, the first expression is the representation of a two-input EX-OR gate, while the second expression gives two forms of representation of a two-input EX-NOR gate.

Table 6.4 Proof of theorem 13(a).

X	Y	Z	XY	$\overline{X}Z$	X+Z	$\overline{X}+Y$	$XY+\overline{X}Z$	(X+Z)($\overline{X}+Y$)
0	0	0	0	0	0	1	0	0
0	0	1	0	1	1	1	1	1
0	1	0	0	0	0	1	0	0
0	1	1	0	1	1	1	1	1
1	0	0	0	0	1	0	0	0
1	0	1	0	0	1	0	0	0
1	1	0	1	0	1	1	1	1
1	1	1	1	0	1	1	1	1

6.3.15 Theorem 15

$$\text{(a) } X.f(X, \overline{X}, Y, Z, \ldots) = X.f(1, 0, Y, Z, \ldots) \tag{6.25}$$

$$\text{(b) } X + f(X, \overline{X}, Y, Z, \ldots) = X + f(0, 1, Y, Z, \ldots) \tag{6.26}$$

According to theorem 15(a), if a variable X is multiplied by an expression containing X and $\overline{X}$ in addition to other variables, then all Xs and $\overline{X}$s can be replaced with 1s and 0s respectively. This would be valid as $X.X = X$ and $X.1 = X$. Also, $X.\overline{X} = 0$ and $X.0 = 0$. According to theorem 15(b), if a variable X is added to an expression containing terms having X and $\overline{X}$ in addition to other variables, then all Xs can be replaced with 0s and all $\overline{X}$s can be replaced with ls. This is again permissible as $X + X$ as well as $X + 0$ equals X. Also, $X + \overline{X}$ and $\overline{X} + 1$ both equal 1.

This pair of theorems is very useful in eliminating redundancy in a given expression. An important corollary of this pair of theorems is that, if the multiplying variable is $\overline{X}$ in theorem 15(a), then all Xs will be replaced by 0s and all $\overline{X}$s will be replaced by ls. Similarly, if the variable being added in theorem 15(b) is $\overline{X}$, then Xs and $\overline{X}$s in the expression are replaced by 1s and 0s respectively. In that case the two theorems can be written as follows:

$$\text{(a) } \overline{X}.f(X, \overline{X}, Y, Z, \ldots) = \overline{X}.f(0, 1, Y, Z, \ldots) \tag{6.27}$$

$$\text{(b) } \overline{X} + f(X, \overline{X}, Y, Z, \ldots) = \overline{X} + f(1, 0, Y, Z, \ldots) \tag{6.28}$$

The theorems are further illustrated with the help of the following examples:

1. $A.[\overline{A}.B + A.\overline{C} + (\overline{A} + D).(A + \overline{E})] = A.[0.B + 1.\overline{C} + (0 + D).(1 + \overline{E})] = A.(\overline{C} + D)$.
2. $\overline{A} + [\overline{A}.B + A.\overline{C} + (\overline{A} + B).(A + \overline{E})] = \overline{A} + [0.B + 1.\overline{C} + (0 + B).(1 + \overline{E})] = \overline{A} + \overline{C} + B$.

6.3.16 Theorem 16

$$\text{(a) } f(X, \overline{X}, Y, \ldots, Z) = X.f(1, 0, Y, \ldots, Z) + \overline{X}.f(0, 1, Y, \ldots, Z) \tag{6.29}$$

$$\text{(b) } f(X, \overline{X}, Y, \ldots, Z) = [X + f(0, 1, Y, \ldots, Z)][\overline{X} + f(1, 0, Y, \ldots, Z)] \tag{6.30}$$

The proof of theorem 16(a) is straightforward and is given as follows:

$$f(X,\overline{X},Y,\ldots,Z) = X.f(X,\overline{X},Y,\ldots,Z) + \overline{X}.f(X,\overline{X},Y,\ldots,Z)$$
$$= X.f(1,0,Y,\ldots,Z) + \overline{X}.f(0,1,Y,\ldots,Z)[(\text{Theorem 15(a)}]$$

Also

$$f(X,\overline{X},Y,\ldots,Z) = [X + f(X,\overline{X},Y,\ldots,Z)][\overline{X} + .f(X,\overline{X},Y,\ldots,Z)]$$
$$= [X + f(0,1,Y,\ldots,Z)][\overline{X} + f(1,0,Y,\ldots,Z)][\text{Theorem 15(b)}]$$

6.3.17 Theorem 17 (Involution Law)

$$\overline{\overline{X}} = X \tag{6.31}$$

Involution law says that the complement of the complement of an expression leaves the expression unchanged. Also, the dual of the dual of an expression is the original expression. This theorem forms the basis of finding the equivalent product-of-sums expression for a given sum-of-products expression, and vice versa.

Example 6.5

Prove the following:

1. $L.(M+\overline{N}) + \overline{L}.\overline{P}.Q = (L+\overline{P}.Q).(\overline{L}+M+\overline{N})$.
2. $[A.\overline{B}+\overline{C}+\overline{D}].[D+(E+\overline{F}).G] = D.(A.\overline{B}+\overline{C}) + \overline{D}.G.(E+\overline{F})$.

Solution

1. Let us assume that $L = X$, $(M+\overline{N}) = Y$ and $\overline{P}.Q = Z$.
   The LHS of the given Boolean equation then reduces to $X.Y + \overline{X}.Z$.
   Applying the transposition theorem,

$$X.Y + \overline{X}.Z = (X+Z).(\overline{X}+Y) = (L+\overline{P}.Q)(\overline{L}+M+\overline{N}) = \text{RHS}$$

2. Let us assume $\overline{D} = X$, $A.\overline{B}+\overline{C} = Y$ and $(E+\overline{F}).G = Z$.
   The LHS of given the Boolean equation then reduces to $(X+Y).(\overline{X}+Z)$.
   Applying the transposition theorem,

$$(X+Y).(\overline{X}+Z) = X.Z + \overline{X}.Y = \overline{D}.G.(E+\overline{F}) + D.(A.\overline{B}+\overline{C}) = \text{RHS}$$

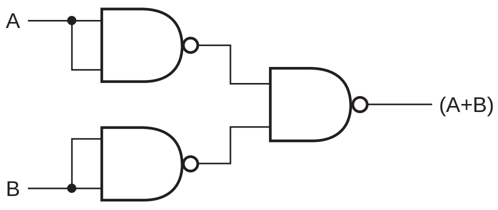


Figure 6.4 Example 6.6.

Example 6.6

Starting with the Boolean expression for a two-input OR gate, apply Boolean laws and theorems to modify it in such a way as to facilitate the implementation of a two-input OR gate by using two-input NAND gates only.

Solution

- A two-input OR gate is represented by the Boolean equation $Y = (A + B)$, where A and B are the input logic variables and Y is the output.
- Now, $(A + B) = (\overline{\overline{A}} + \overline{\overline{B}})$ Involution law

  $= (\overline{\overline{A}.\overline{B}})$ DeMorgan's theorem

  $= [\overline{(\overline{A.A}).(\overline{B.B})}]$ Idempotent law
- Figure 6.4 shows the NAND gate implementation of a two-input OR gate.

Example 6.7

Apply suitable Boolean laws and theorems to modify the expression for a two-input EX-OR gate in such a way as to implement a two-input EX-OR gate by using the minimum number of two-input NAND gates only.

Solution

- A two-input EX-OR gate is represented by the Boolean expression $Y = \overline{A}.B + A.\overline{B}$.
- Now, $\overline{A}.B + A.\overline{B} = \overline{\overline{\overline{A}.B}} + \overline{\overline{A.\overline{B}}}$ Involution law

  $= \overline{\overline{\overline{A}.B}.\overline{A.\overline{B}}}$ DeMorgan's law

  $= [\overline{\overline{B.(\overline{A}+\overline{B})}.\overline{A.(\overline{A}+\overline{B})}}]$

  $$= \overline{(\overline{B.\overline{A.B}}).(\overline{A.\overline{A.B}})} \qquad (6.32)$$
- Equation (6.32) is in a form that can be implemented with NAND gates only.
- Figure 6.5 shows the logic diagram.

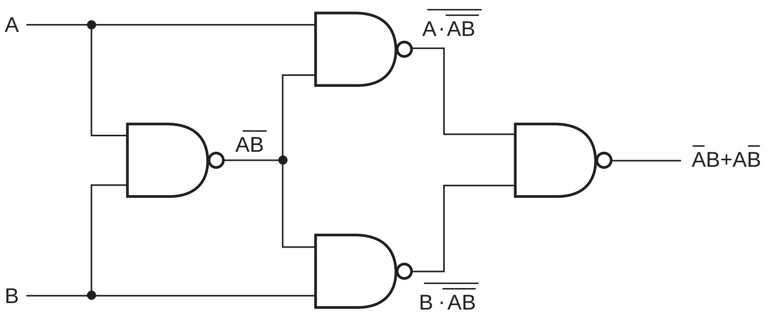


Figure 6.5 Example 6.7.

6.4 Simplification Techniques

In this section, we will discuss techniques other than the application of laws and theorems of Boolean algebra discussed in the preceding paragraphs of this chapter for simplifying or more precisely minimizing a given complex Boolean expression. The primary objective of all simplification procedures is to obtain an expression that has the minimum number of terms. Obtaining an expression with the minimum number of literals is usually the secondary objective. If there is more than one possible solution with the same number of terms, the one having the minimum number of literals is the choice. The techniques to be discussed include:

(a) the Quine–McCluskey tabular method;
(b) the Karnaugh map method.

Before we move on to discuss these techniques in detail, it would be relevant briefly to describe sum-of-products and product-of-sums Boolean expressions. The given Boolean expression will be in either of the two forms, and the objective will be to find a minimized expression in the same or the other form.

6.4.1 Sum-of-Products Boolean Expressions

A sum-of-products expression contains the sum of different terms, with each term being either a single literal or a product of more than one literal. It can be obtained from the truth table directly by considering those input combinations that produce a logic '1' at the output. Each such input combination produces a term. Different terms are given by the product of the corresponding literals. The sum of all terms gives the expression. For example, the truth table in Table 6.5 can be represented by the Boolean expression

$$Y = \overline{A}\,.\overline{B}.\overline{C} + \overline{A}.B.C + A.B.\overline{C} + A.\overline{B}.C \tag{6.33}$$

Considering the first term, the output is '1' when $A = 0$, $B = 0$ and $C = 0$. This is possible only when $\overline{A}$, $\overline{B}$ and $\overline{C}$ are ANDed. Also, for the second term, the output is '1' only when B, C and $\overline{A}$ are ANDed. Other terms can be explained similarly. A sum-of-products expression is also known as a *minterm expression*.

Table 6.5 truth table of boolean expression of equation 6.33.

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

6.4.2 Product-of-Sums Expressions

A product-of-sums expression contains the product of different terms, with each term being either a single literal or a sum of more than one literal. It can be obtained from the truth table by considering those input combinations that produce a logic ‘0’ at the output. Each such input combination gives a term, and the product of all such terms gives the expression. Different terms are obtained by taking the sum of the corresponding literals. Here, ‘0’ and ‘1’ respectively mean the uncomplemented and complemented variables, unlike sum-of-products expressions where ‘0’ and ‘1’ respectively mean complemented and uncomplemented variables.

To illustrate this further, consider once again the truth table in Table 6.5. Since each term in the case of the product-of-sums expression is going to be the sum of literals, this implies that it is going to be implemented using an OR operation. Now, an OR gate produces a logic ‘0’ only when all its inputs are in the logic ‘0’ state, which means that the first term corresponding to the second row of the truth table will be $A+B+\overline{C}$. The product-of-sums Boolean expression for this truth table is given by $(A+B+\overline{C}).(A+\overline{B}+C).(\overline{A}+B+C).(\overline{A}+\overline{B}+\overline{C})$.

Transforming the given product-of-sums expression into an equivalent sum-of-products expression is a straightforward process. Multiplying out the given expression and carrying out the obvious simplification provides the equivalent sum-of-products expression:

$$
\begin{aligned}
&(A+B+\overline{C}).(A+\overline{B}+C).(\overline{A}+B+C).(\overline{A}+\overline{B}+\overline{C})\\
&=(A.A+A.\overline{B}+A.C+B.A+B.\overline{B}+B.C+\overline{C}.A+\overline{C}.\overline{B}+\overline{C}.C).(\overline{A}.\overline{A}+\overline{A}.\overline{B}+\overline{A}.\overline{C}+B.\overline{A}+B.\overline{B}\\
&\quad +B.\overline{C}+C.\overline{A}+C.\overline{B}+C.\overline{C}\\
&=(A+B.C+\overline{B}.\overline{C}).(\overline{A}+B.\overline{C}+C.\overline{B})=A.B.\overline{C}+A.\overline{B}.C+\overline{A}.B.C+\overline{A}.\overline{B}.\overline{C}
\end{aligned}
$$

A given sum-of-products expression can be transformed into an equivalent product-of-sums expression by (a) taking the dual of the given expression, (b) multiplying out different terms to get the sum-of-products form, (c) removing redundancy and (d) taking a dual to get the equivalent product-of-sums expression. As an illustration, let us find the equivalent product-of-sums expression of the sum-of-products expression

$$A.B+\overline{A}.\overline{B}$$

The dual of the given expression $=(A+B).(\overline{A}+\overline{B})$:

$$(A+B).(\overline{A}+\overline{B})=A.\overline{A}+A.\overline{B}+B.\overline{A}+B.\overline{B}=0+A.\overline{B}+B.\overline{A}+0=A.\overline{B}+\overline{A}.B$$

The dual of $(A.\overline{B}+\overline{A}.B)=(A+\overline{B}).(\overline{A}+B)$. Therefore

$$A.B+\overline{A}.\overline{B}=(A+\overline{B}).(\overline{A}+B)$$

6.4.3 Expanded Forms of Boolean Expressions

Expanded sum-of-products and product-of-sums forms of Boolean expressions are useful not only in analysing these expressions but also in the application of minimization techniques such as the Quine–McCluskey tabular method and the Karnaugh mapping method for simplifying given Boolean expressions. The expanded form, sum-of-products or product-of-sums, is obtained by including all possible combinations of missing variables.

As an illustration, consider the following sum-of-products expression:

$$A.\overline{B}+B.\overline{C}+A.B.\overline{C}+\overline{A}.C$$

It is a three-variable expression. Expanded versions of different minterms can be written as follows:

- $A.\overline{B}=A.\overline{B}.(C+\overline{C})=A.\overline{B}.C+A.\overline{B}.\overline{C}$.
- $B.\overline{C}=B.\overline{C}.(A+\overline{A})=B.\overline{C}.A+B.\overline{C}.\overline{A}$.
- $A.B.\overline{C}$ is a complete term and has no missing variable.
- $\overline{A}.C=\overline{A}.C.(B+\overline{B})=\overline{A}.C.B+\overline{A}.C.\overline{B}$.

The expanded sum-of-products expression is therefore given by

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

As another illustration, consider the product-of-sums expression

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

It is four-variable expression with A, B, C and D being the four variables. $\overline{A}+B$ in this case expands to $(\overline{A}+B+C+D).(\overline{A}+B+C+\overline{D}).(\overline{A}+B+\overline{C}+D).(\overline{A}+B+\overline{C}+\overline{D})$.

The expanded product-of-sums expression is therefore given by

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

6.4.4 Canonical Form of Boolean Expressions

An expanded form of Boolean expression, where each term contains all Boolean variables in their true or complemented form, is also known as the *canonical form* of the expression.

As an illustration, $f(A.B,C)=\overline{A}.\overline{B}.\overline{C}+\overline{A}.\overline{B}.C+A.B.C$ is a Boolean function of three variables expressed in canonical form. This function after simplification reduces to $\overline{A}.\overline{B}+A.B.C$ and loses its canonical form.

6.4.5 Σ and Π Nomenclature

Σ and Π notations are respectively used to represent sum-of-products and product-of-sums Boolean expressions. We will illustrate these notations with the help of examples. Let us consider the following Boolean function:

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

We will represent this function using Σ notation. The first step is to write the expanded sum-of-products given by

$$\begin{aligned} f(A, B, C, D) &= A.\overline{B}.\overline{C}.(D + \overline{D}) + A.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.\overline{C}.D \\ &= A.\overline{B}.\overline{C}.D + A.\overline{B}.\overline{C}.\overline{D} + A.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.\overline{C}.D \end{aligned}$$

Different terms are then arranged in ascending order of the binary numbers represented by various terms, with true variables representing a '1' and a complemented variable representing a '0'. The expression becomes

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

The different terms represent 0001, 0101, 1000, 1001 and 1111. The decimal equivalent of these terms enclosed in the Σ then gives the Σ notation for the given Boolean function. That is, $f(A, B, C, D) = \sum 1, 5, 8, 9, 15$.

The complement of $f(A, B, C, D)$, that is, $f'(A, B, C, D)$, can be directly determined from Σ notation by including the left-out entries from the list of all possible numbers for a four-variable function. That is,

$$f'(A, B, C, D) = \sum 0, 2, 3, 4, 6, 7, 10, 11, 12, 13, 14$$

Let us now take the case of a product-of-sums Boolean function and its representation in Π nomenclature. Let us consider the Boolean function

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

The expanded product-of-sums form is given by

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

The binary numbers represented by the different sum terms are 0011, 1011, 1100 and 0111 (true and complemented variables here represent 0 and 1 respectively). When arranged in ascending order, these numbers are 0011, 0111, 1011 and 1100. Therefore,

$$f(A, B, C, D) = \prod 3, 7, 11, 12 \quad \text{and} \quad f'(A, B, C, D) = \prod 0, 1, 2, 4, 5, 6, 8, 9, 10, 13, 14, 15$$

An interesting corollary of what we have discussed above is that, if a given Boolean function $f(A,B,C)$ is given by $f(A, B, C) = \sum 0, 1, 4, 7$, then

$$f(A, B, C) = \prod 2, 3, 5, 6 \quad \text{and} \quad f'(A, B, C) = \sum 2, 3, 5, 6 = \prod 0, 1, 4, 7$$

Optional combinations can also be incorporated into Σ and Π nomenclature using suitable identifiers; ϕ or d are used as identifiers. For example, if $f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.\overline{C} + A.\overline{B}.C$ and $\overline{A}.B.C$, $A.B.C$ are optional combinations, then

$$f(A, B, C) = \sum 0, 4, 5 + \sum_{\phi} 3, 7 = \sum 0, 4, 5 + \sum_{d} 3, 7$$

$$f(A, B, C) = \prod 1, 2, 6 + \prod_{\phi} 3, 7 = \prod 1, 2, 6 + \prod_{d} 3, 7$$

Example 6.8

For a Boolean function $f(A, B) = \sum 0, 2$, *prove that* $f(A, B) = \prod 1, 3$ *and* $f'(A, B) = \sum 1, 3 = \prod 0, 2$.

Solution

- $f(A, B) = \sum 0, 2 = \overline{A}.\overline{B} + A.\overline{B} = \overline{B}.(A + \overline{A}) = \overline{B}$.
- Now, $\prod 1, 3 = (A + \overline{B}).(\overline{A} + \overline{B}) = A.\overline{A} + A.\overline{B} + \overline{B}.\overline{A} + \overline{B}.\overline{B} = A.\overline{B} + \overline{A}.\overline{B} + \overline{B} = \overline{B}$.
- Now, $\sum 1, 3 = \overline{A}.B + A.B = B.(\overline{A} + A) = B$.
  and $\prod 0, 2 = (A + B).(\overline{A} + B) = A.\overline{A} + A.B + B.\overline{A} + B.B = A.B + \overline{A}.B + B = B$.
- Therefore, $\sum 1, 3 = \prod 0, 2$.
- Also, $f(A, B) = \overline{B}$.
- Therefore, $f'(A, B) = B$ or $f'(A, B) = \sum 1, 3 = \prod 0, 2$.

6.5 Quine–McCluskey Tabular Method

The Quine–McCluskey tabular method of simplification is based on the complementation theorem, which says that

$$X.Y + X.\overline{Y} = X \tag{6.34}$$

where X represents either a variable or a term or an expression and Y is a variable. This theorem implies that, if a Boolean expression contains two terms that differ only in one variable, then they can be combined together and replaced with a term that is smaller by one literal. The same procedure is applied for the other pairs of terms wherever such a reduction is possible. All these terms reduced by one literal are further examined to see if they can be reduced further. The process continues until the terms become irreducible. The irreducible terms are called *prime implicants*. An optimum set of prime implicants that can account for all the original terms then constitutes the minimized expression. The technique can be applied equally well for minimizing sum-of-products and product-of-sums expressions and is particularly useful for Boolean functions having more than six variables as it can be mechanized and run on a computer. On the other hand, the Karnaugh mapping method, to be discussed later, is a graphical method and becomes very cumbersome when the number of variables exceeds six.

The step-by-step procedure for application of the tabular method for minimizing Boolean expressions, both sum-of-products and product-of-sums, is outlined as follows:

1. The Boolean expression to be simplified is expanded if it is not in expanded form.
2. Different terms in the expression are divided into groups depending upon the number of 1s they have. True and complemented variables in a sum-of-products expression mean ‘1’ and ‘0’ respectively.

The reverse is true in the case of a product-of-sums expression. The groups are then arranged, beginning with the group having the least number of 1s in its included terms. Terms within the same group are arranged in ascending order of the decimal numbers represented by these terms.

As an illustration, consider the expression

$$A.B.C+\overline{A}.B.C+A.\overline{B}.\overline{C}+A.\overline{B}.C+\overline{A}.\overline{B}.\overline{C}$$

The grouping of different terms and the arrangement of different terms within the group are shown below:

$\overline{A}.\overline{B}.\overline{C}$		000	First group
$A.\overline{B}.\overline{C}$		100	Second group
$\overline{A}.B.C$	$\longrightarrow$	011	Third group
$A.\overline{B}.C$		101	
ABC		111	Fourth group

As another illustration, consider a product-of-sums expression given by

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

The formation of groups and the arrangement of terms within different groups for the product-of-sums expression are as follows:

$A.B.C.D$		0000
$A.B.\overline{C}.\overline{D}$		0011
$A.\overline{B}.C.\overline{D}$		0101
$\overline{A}.B.\overline{C}.D$	$\longrightarrow$	1010
$A.\overline{B}.\overline{C}.\overline{D}$		0111
$\overline{A}.\overline{B}.\overline{C}.D$		1110
$\overline{A}.\overline{B}.\overline{C}.\overline{D}$		1111

It may be mentioned here that the Boolean expressions that we have considered above did not contain any optional terms. If there are any, they are also considered while forming groups. This completes the first table.

3. The terms of the first group are successively matched with those in the next adjacent higher-order group to look for any possible matching and consequent reduction. The terms are considered matched when all literals except for one match. The pairs of matched terms are replaced with a

single term where the position of the unmatched literals is replaced with a dash (—). These new terms formed as a result of the matching process find a place in the second table. The terms in the first table that do not find a match are called the prime implicants and are marked with an asterisk ($*$). The matched terms are ticked (✓).

4. Terms in the second group are compared with those in the third group to look for a possible match. Again, terms in the second group that do not find a match become the prime implicants.
5. The process continues until we reach the last group. This completes the first round of matching. The terms resulting from the matching in the first round are recorded in the second table.
6. The next step is to perform matching operations in the second table. While comparing the terms for a match, it is important that a dash (—) is also treated like any other literal, that is, the dash signs also need to match. The process continues on to the third table, the fourth table and so on until the terms become irreducible any further.
7. An optimum selection of prime implicants to account for all the original terms constitutes the terms for the minimized expression. Although optional (also called 'don't care') terms are considered for matching, they do not have to be accounted for once prime implicants have been identified.

Let us consider an example. Consider the following sum-of-products expression:

$$\overline{A}.B.C+\overline{A}.\overline{B}.D+A.\overline{C}.D+B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D \tag{6.35}$$

In the first step, we write the expanded version of the given expression. It can be written as follows:

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

The formation of groups, the placement of terms in different groups and the first-round matching are shown as follows:

A	B	C	D
0	0	0	1
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	1
1	1	0	0
1	1	0	1

A	B	C	D	
0	0	0	1	✓
0	1	0	0	✓
0	0	1	1	✓
0	1	0	1	✓
0	1	1	0	✓
1	0	0	1	✓
1	1	0	0	✓
0	1	1	1	✓
1	1	0	1	✓

A	B	C	D	
0	0	–	1	✓
0	–	0	1	✓
–	0	0	1	✓
0	1	0	–	✓
0	1	–	0	✓
–	1	0	0	✓
0	–	1	1	✓
0	1	–	1	✓
–	1	0	1	✓
0	1	1	–	✓
1	–	0	1	✓
1	1	0	–	✓

The second round of matching begins with the table shown on the previous page. Each term in the first group is compared with every term in the second group. For instance, the first term in the first group 00−1 matches with the second term in the second group 01−1 to yield 0−−1, which is recorded in the table shown below. The process continues until all terms have been compared for a possible match. Since this new table has only one group, the terms contained therein are all prime implicants. In the present example, the terms in the first and second tables have all found a match. But that is not always the case.

A	B	C	D	
0	–	–	1	*
–	–	0	1	*
0	1	–	–	*
–	1	0	–	*

The next table is what is known as the prime implicant table. The prime implicant table contains all the original terms in different columns and all the prime implicants recorded in different rows as shown below:

0001	0011	0100	0101	0110	0111	1001	1100	1101		
✓	✓		✓		✓				0− −1	$P \to \overline{A}.D$
✓			✓			✓		✓	− −01	$Q \to \overline{C}.D$
		✓	✓	✓	✓				01− −	$R \to \overline{A}.B$
		✓	✓				✓	✓	−10−	$S \to B.\overline{C}$

Each prime implicant is identified by a letter. Each prime implicant is then examined one by one and the terms it can account for are ticked as shown. The next step is to write a product-of-sums expression using the prime implicants to account for all the terms. In the present illustration, it is given as follows.

$$(P+Q).(P).(R+S).(P+Q+R+S).(R).(P+R).(Q).(S).(Q+S)$$

Obvious simplification reduces this expression to $PQRS$ which can be interpreted to mean that all prime implicants, that is, P, Q, R and S, are needed to account for all the original terms.

Therefore, the minimized expression $= \overline{A}.D+\overline{C}.D+\overline{A}.B+B.\overline{C}$.

What has been described above is the formal method of determining the optimum set of prime implicants. In most of the cases where the prime implicant table is not too complex, the exercise can be done even intuitively. The exercise begins with identification of those terms that can be accounted for by only a single prime implicant. In the present example, 0011, 0110, 1001 and 1100 are such terms. As a result, P, Q, R and S become the essential prime implicants. The next step is to find out if any terms have not been covered by the essential prime implicants. In the present case, all terms have been covered by essential prime implicants. In fact, all prime implicants are essential prime implicants in the present example.

As another illustration, let us consider a product-of-sums expression given by

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

The procedure is similar to that described for the case of simplification of sum-of-products expressions. The resulting tables leading to identification of prime implicants are as follows:

A	*B*	*C*	*D*
0	1	0	1
0	1	1	1
1	1	0	1
1	1	1	0
1	1	1	1

A	*B*	*C*	*D*	
0	1	0	1	✓
0	1	1	1	✓
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✓

A	*B*	*C*	*D*	
0	1	–	1	✓
–	1	0	1	✓
–	1	1	1	✓
1	1	–	1	✓
1	1	1	–	*

A	*B*	*C*	*D*	
–	1	–	1	*

The prime implicant table is constructed after all prime implicants have been identified to look for the optimum set of prime implicants needed to account for all the original terms. The prime implicant table shows that both the prime implicants are the essential ones:

0101	0111	1101	1110	1111	Prime implicants
			✓	✓	111–
✓	✓	✓		✓	–1–1

The minimized expression = $(\overline{A}+\overline{B}+\overline{C}).(\overline{B}+\overline{D})$.

6.5.1 *Tabular Method for Multi-Output Functions*

When it comes to a multi-output logic network, a network that has more than one output, sharing of some logic blocks between different functions is highly probable. For an optimum logic implementation of the multi-output function, different functions cannot be and should not be minimized in isolation because a possible common term that could have been shared may not turn out to be a prime implicant if the functions are worked out individually. The method of applying the tabular approach to multi-output functions is to get a minimized set of expressions that would lead to an optimum overall system. The method is illustrated by the following example.

Consider a logic system with two outputs that is described by the following Boolean expressions:

$$Y_1 = \overline{A}.B.D + \overline{A}.C.D + \overline{A}.\overline{C}.\overline{D} \tag{6.36}$$

$$Y_2 = \overline{A}.B.C + A.C.D + A.\overline{B}.C.\overline{D} + \overline{A}.\overline{B}.C.\overline{D} \tag{6.37}$$

The expanded forms of the two functions are as follows:

$$Y_1 = \overline{A}.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.B.C.D + \overline{A}.\overline{B}.C.D + \overline{A}.B.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}.\overline{D}$$

$$Y_1 = \overline{A}.B.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.\overline{B}.C.D + \overline{A}.B.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}.\overline{D}$$

$$Y_2 = \overline{A}.B.C.D + \overline{A}.B.C.\overline{D} + A.B.C.D + A.\overline{B}.C.D + A.\overline{B}.C.\overline{D} + \overline{A}.\overline{B}.C.\overline{D}$$

The rows representing different terms are arranged in the usual manner, with all the terms contained in the two functions finding a place without repetition, as shown in the table below:

ABCD	1	2	
0000	✓		✓
0010		✓	✓
0100	✓		✓
0011	✓		✓
0101	✓		✓
0110		✓	✓
1010		✓	✓
0111	✓	✓	✓
1011		✓	✓
1111		✓	✓

Each term is checked under the column or columns depending upon the functions in which it is contained. For instance, if a certain term is contained in the logic expressions for both output 1 and output 2, it will be checked in both output columns. The matching process begins in the same way as described earlier for the case of single-output functions, with some modifications outlined as follows:

1. Only those terms can be combined that have at least one check mark in the output column in common. For instance, 0000 cannot combine with 0010 but can combine with 0100.
2. In the resulting row, only the common outputs are checked. For instance, when 0101 is matched with 0111, then, in the resulting term 01–1, only output 1 will be checked.
3. A combining term can be checked off only if the resulting term accounts for all the outputs in which the term is contained.

The table below shows the results of the first round of matching:

ABCD	1	2	
0–00	✓		*
0–10		✓	*
–010		✓	*
010–	✓		*
0–11	✓		*
01–1	✓		*
011–		✓	*
101–		✓	*
–111		✓	*
1–11		✓	*

No further matching is possible. The prime implicant table is shown below:

Output 1					Output 2						
0000	0011	0100	0101	0111	0010	0110	0111	1010	1011	1111	ABCD
✓		✓									0–00
					✓	✓					0–10
					✓			✓			–010
		✓	✓								010–
	✓			✓							0–11
			✓	✓							01–1
						✓	✓				011–
								✓	✓		101–
							✓			✓	–111
									✓	✓	1–11

For each prime implicant, check marks are placed only in columns that pertain to the outputs checked off for this prime implicant. For instance, 0-00 has only output 1 checked off. Therefore, the relevant terms under output 1 will be checked off. The completed table is treated as a whole while marking the required prime implicants to be considered for writing the minimized expressions. The minimized expressions are as follows:

$$Y_1 = \overline{A}.\overline{C}.\overline{D} + \overline{A}.C.D + \overline{A}.B.\overline{C} \quad \text{and} \quad Y_2 = B.C.D + A.\overline{B}.C + \overline{A}.C.\overline{D}$$

Example 6.9

Using the Quine–McCluskey tabular method, find the minimum sum of products for $f(A, B, C, D) = \sum(1, 2, 3, 9, 12, 13, 14) + \sum_{\phi}(0, 7, 10, 15)$.

Solution

The different steps to finding the solution to the given problem are tabulated below. As we can see, eight prime implicants have been identified. These prime implicants along with the inputs constitute the prime implicant table. Remember that optional inputs are not considered while constructing the prime implicant table:

A	*B*	*C*	*D*	
0	0	0	0	✓
0	0	0	1	✓
0	0	1	0	✓
0	0	1	1	✓
1	0	0	1	✓
1	0	1	0	✓
1	1	0	0	✓
0	1	1	1	✓
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✓

A	*B*	*C*	*D*	
0	0	0	–	✓
0	0	–	0	✓
0	0	–	1	✓
–	0	0	1	*
0	0	1	–	✓
–	0	1	0	*
0	–	1	1	*
1	–	0	1	*
1	–	1	0	*
1	1	0	–	✓
1	1	–	0	✓
–	1	1	1	*
1	1	–	1	✓
1	1	1	–	✓

A	*B*	*C*	*D*	
0	0	–	–	*
1	1	–	–	*

The product-of-sums expression that tells about the combination of prime implicants required to account for all the terms is given by the expression

$$(L+S).(M+S).(N+S).(L+P).(T).(P+T).(Q+T) \tag{6.38}$$

After obvious simplification, this reduces to the expression

$$\begin{aligned} &T.(L+S).(M+S).(N+S).(L+P) \\ &\quad = T.(LM+LS+MS+S).(LN+PN+LS+PS) \\ &\quad = T.(LM+S).(LN+PN+LS+PS) \\ &\quad = T.(LMN+LMPN+LMS+LMPS+LNS+PNS+LS+PS) \\ &\quad = T.(LMN+LMPN+LS+PS) \\ &\quad = TLMN+TLMPN+TLS+TPS \end{aligned} \tag{6.39}$$

0001	0010	0011	1001	1100	1101	1110	Prime implicants	
✓			✓				−001	L
	✓						−010	M
		✓					0−11	N
			✓		✓		1−01	P
						✓	1−10	Q
							−111	R
✓	✓	✓					00−−	S
				✓	✓	✓	11−−	T

The sum-of-products Boolean expression (6.39) states that all the input combinations can be accounted for by the prime implicants (T, L, M, N) or (T, L, M, P, N) or (T, L, S) or (T, P, S). The most optimum expression would result from either *TLS* or *TPS*. Therefore, the minimized Boolean function is given by

$$f(A, B, C, D) = A.B+\overline{B}.\overline{C}.D+\overline{A}.\overline{B} \tag{6.40}$$

or by

$$f(A, B, C, D) = A.B+\overline{A}.\overline{B}+A.\overline{C}.D \tag{6.41}$$

Example 6.10

A logic system has three inputs A, B and C and two outputs Y_1 and Y_2. The output functions Y_1 and Y_2 are expressed by $Y_1=\overline{A}.B.C+B.\overline{C}+\overline{A}.\overline{C}+A.\overline{B}.C+A.B.C$ and $Y_2=\overline{A}.B+A.\overline{C}+A.B.C$. Determine the minimized output logic functions using the Quine–McCluskey tabular method.

Solution

The expanded forms of Y_1 and Y_2 are written as follows:

$$\begin{aligned} Y_1 &= \overline{A}.B.C+A.B.\overline{C}+\overline{A}.B.\overline{C}+\overline{A}.B.\overline{C}+\overline{A}.\overline{B}.\overline{C}+A.\overline{B}.C+A.B.C \\ &= \overline{A}.B.C+A.B.\overline{C}+\overline{A}.B.\overline{C}+\overline{A}.\overline{B}.\overline{C}+A.\overline{B}.C+A.B.C \\ Y_2 &= \overline{A}.B.C+\overline{A}.B.\overline{C}+A.B.C+A.B.\overline{C}+A.\overline{B}.\overline{C} \end{aligned}$$

The different steps leading to construction of the prime implicant table are given in tabular form below:

A	*B*	*C*	1	2	
0	0	0	✓		✓
0	1	0	✓	✓	✓
1	0	0		✓	✓
0	1	1	✓	✓	✓
1	0	1	✓		✓
1	1	0	✓	✓	✓
1	1	1	✓	✓	✓

A	*B*	*C*	1	2	
0	–	0	✓		*
0	1	–	✓	✓	✓
–	1	0	✓	✓	✓
1	–	0		✓	*
–	1	1	✓	✓	✓
1	–	1	✓		*
1	1	–	✓	✓	✓

A	*B*	*C*	1	2	
–	1	–	✓	✓	*

Y_1						Y_2					**ABC**
000	**010**	**011**	**101**	**110**	**111**	**010**	**011**	**100**	**110**	**111**	
✓	✓										**0–0**
								✓	✓		**1–0**
			✓		✓						**1–1**
	✓	✓		✓	✓	✓	✓		✓	✓	**–1–**

From the prime implicant table, the minimized output Boolean functions can be written as follows:

$$Y_1 = B + \overline{A}.\overline{C} + A.C \tag{6.42}$$

$$Y_2 = B + A.\overline{C} \tag{6.43}$$

6.6 Karnaugh Map Method

A Karnaugh map is a graphical representation of the logic system. It can be drawn directly from either minterm (sum-of-products) or maxterm (product-of-sums) Boolean expressions. Drawing a Karnaugh map from the truth table involves an additional step of writing the minterm or maxterm expression depending upon whether it is desired to have a minimized sum-of-products or a minimized product-of-sums expression.

6.6.1 Construction of a Karnaugh Map

An n-variable Karnaugh map has 2^n squares, and each possible input is allotted a square. In the case of a minterm Karnaugh map, ‘1’ is placed in all those squares for which the output is ‘1’, and ‘0’ is placed in all those squares for which the output is ‘0’. 0s are omitted for simplicity. An ‘X’ is placed in squares corresponding to ‘don’t care’ conditions. In the case of a maxterm Karnaugh map, a ‘1’ is placed in all those squares for which the output is ‘0’, and a ‘0’ is placed for input entries corresponding to a ‘1’ output. Again, 0s are omitted for simplicity, and an ‘X’ is placed in squares corresponding to ‘don’t care’ conditions.

The choice of terms identifying different rows and columns of a Karnaugh map is not unique for a given number of variables. The only condition to be satisfied is that the designation of adjacent rows and adjacent columns should be the same except for one of the literals being complemented. Also, the extreme rows and extreme columns are considered adjacent. Some of the possible designation styles for two-, three- and four-variable minterm Karnaugh maps are given in Figs 6.6, 6.7 and 6.8 respectively.

The style of row identification need not be the same as that of column identification as long as it meets the basic requirement with respect to adjacent terms. It is, however, accepted practice to adopt a uniform style of row and column identification. Also, the style shown in Figs 6.6(a), 6.7(a) and 6.8(a) is more commonly used. Some more styles are shown in Fig. 6.9. A similar discussion applies for maxterm Karnaugh maps.

Having drawn the Karnaugh map, the next step is to form groups of 1s as per the following guidelines:

1. Each square containing a ‘1’ must be considered at least once, although it can be considered as often as desired.
2. The objective should be to account for all the marked squares in the minimum number of groups.
3. The number of squares in a group must always be a power of 2, i.e. groups can have 1, 2, 4, 8, 16, . . . squares.
4. Each group should be as large as possible, which means that a square should not be accounted for by itself if it can be accounted for by a group of two squares; a group of two squares should not be made if the involved squares can be included in a group of four squares and so on.
5. ‘Don’t care’ entries can be used in accounting for all of 1-squares to make optimum groups. They are marked ‘X’ in the corresponding squares. It is, however, not necessary to account for all ‘don’t care’ entries. Only such entries that can be used to advantage should be used.

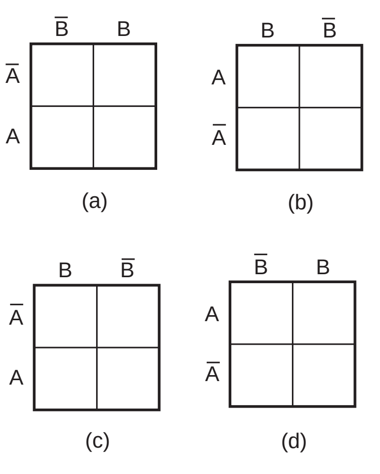


Figure 6.6 Two-variable Karnaugh map.

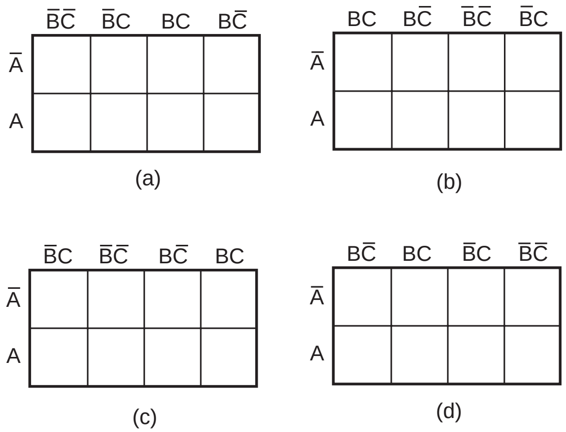


Figure 6.7 Three-variable Karnaugh map.

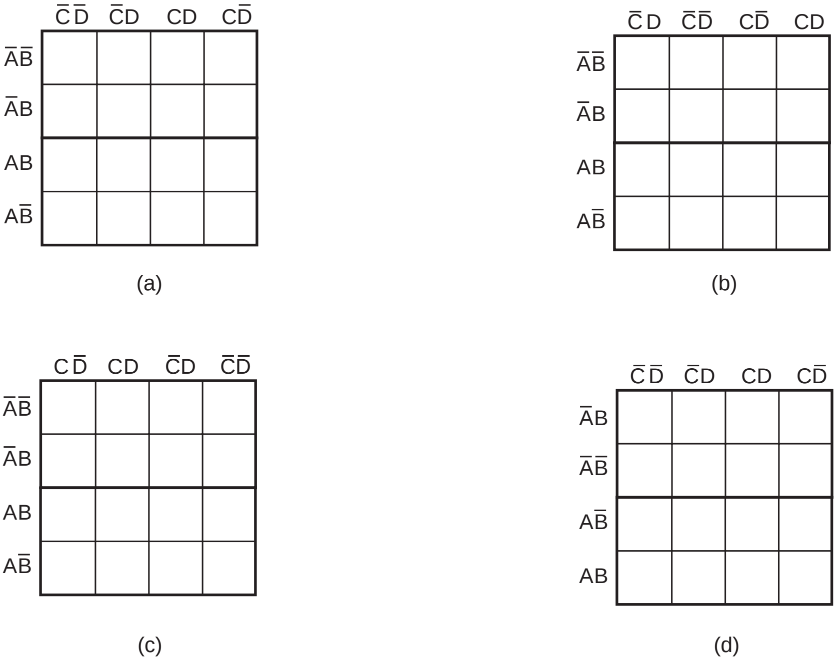


Figure 6.8 Four-variable Karnaugh map.

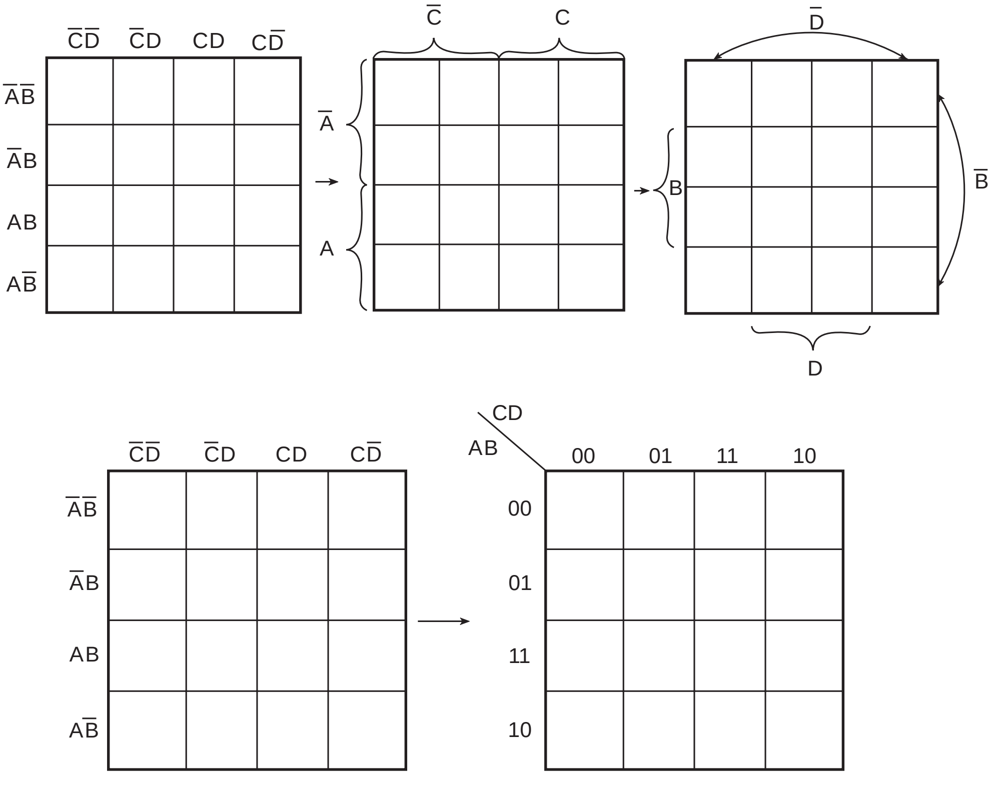


Figure 6.9 Different styles of row and column identification.

Having accounted for groups with all 1s, the minimum 'sum-of-products' or 'product-of-sums' expressions can be written directly from the Karnaugh map.

Figure 6.10 shows the truth table, minterm Karnaugh map and maxterm Karnaugh map of the Boolean function of a two-input OR gate. The minterm and maxterm Boolean expressions for the two-input OR gate are as follows:

$$Y = A + B \text{ (maxterm or product-of-sums)} \tag{6.44}$$

$$Y = \overline{A}.B + A.\overline{B} + A.B \text{ (minterm or sum-of-products)} \tag{6.45}$$

Figure 6.11 shows the truth table, minterm Karnaugh map and maxterm Karnaugh map of the three-variable Boolean function

$$Y = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.\overline{C} \tag{6.46}$$

$$Y = (\overline{A} + \overline{B} + \overline{C}).(\overline{A} + B + \overline{C}).(A + \overline{B} + \overline{C}).(A + B + \overline{C}) \tag{6.47}$$

Truth table

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

	$\overline{B}$	B
$\overline{A}$		1
A	1	1

Sum-of-products K-map

	$\overline{B}$	B
$\overline{A}$		
A		1

Product-of-sums K-map

Figure 6.10 Two-variable Karnaugh maps.

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

	$\overline{B}\overline{C}$	$\overline{B}C$	BC	$B\overline{C}$
$\overline{A}$	1			1
A	1			1

Sum-of-products K-map

	$\overline{B}+\overline{C}$	$\overline{B}+C$	$B+C$	$B+\overline{C}$
$\overline{A}$	1			1
A	1			1

Product-of-sums K-map

Figure 6.11 Three-variable Karnaugh maps.

Figure 6.12 shows the truth table, minterm Karnaugh map and maxterm Karnaugh map of the four-variable Boolean function

$$Y=\overline{A}.\overline{B}.\overline{C}.\overline{D}+\overline{A}.\overline{B}.\overline{C}.D+\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.\overline{C}.D+A.B.\overline{C}.\overline{D}+A.B.\overline{C}.D \tag{6.48}$$

$$\begin{aligned} Y=&(A+B+\overline{C}+D).(A+B+\overline{C}+\overline{D}).(A+\overline{B}+\overline{C}+D).(A+\overline{B}+\overline{C}+\overline{D})\\ &.(\overline{A}+B+\overline{C}+D).(\overline{A}+B+\overline{C}+\overline{D}).(\overline{A}+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+\overline{D}) \end{aligned} \tag{6.49}$$

To illustrate the process of forming groups and then writing the corresponding minimized Boolean expression, Figs 6.13(a) and (b) respectively show minterm and maxterm Karnaugh maps for the Boolean functions expressed by equations (6.50) and (6.51). The minimized expressions as deduced from Karnaugh maps in the two cases are given by Equation (6.52) in the case of the minterm Karnaugh map and Equation (6.53) in the case of the maxterm Karnaugh map:

$$Y=\overline{A}.\overline{B}.\overline{C}.\overline{D}+\overline{A}.\overline{B}.C.\overline{D}+\overline{A}.B.\overline{C}.D+\overline{A}.B.C.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.C.\overline{D}+A.B.\overline{C}.D+A.B.C.D \tag{6.50}$$

$$\begin{aligned} Y=&(A+B+C+\overline{D}).(A+B+\overline{C}+\overline{D}).(A+\overline{B}+C+D).(A+\overline{B}+C+\overline{D}).(A+\overline{B}+\overline{C}+\overline{D})\\ &.(A+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+C+\overline{D}).(\overline{A}+\overline{B}+\overline{C}+\overline{D}).(\overline{A}+B+C+\overline{D}).(\overline{A}+B+\overline{C}+\overline{D}) \end{aligned} \tag{6.51}$$

$$Y=\overline{B}.\overline{D}+B.D \tag{6.52}$$

$$Y=\overline{D}.(A+\overline{B}) \tag{6.53}$$

Truth table

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

	$\overline{C}\,\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\,\overline{B}$	1	1		
$\overline{A}B$	1	1		
AB	1	1		
$A\overline{B}$	1	1		

Sum-of-products K-map

	$\overline{C}+\overline{D}$	$\overline{C}+D$	$C+D$	$C+\overline{D}$
$\overline{A}+\overline{B}$	1	1		
$\overline{A}+B$	1	1		
$A+B$	1	1		
$A+\overline{B}$	1	1		

Product-of-sums K-map

Figure 6.12 Four-variable Karnaugh maps.

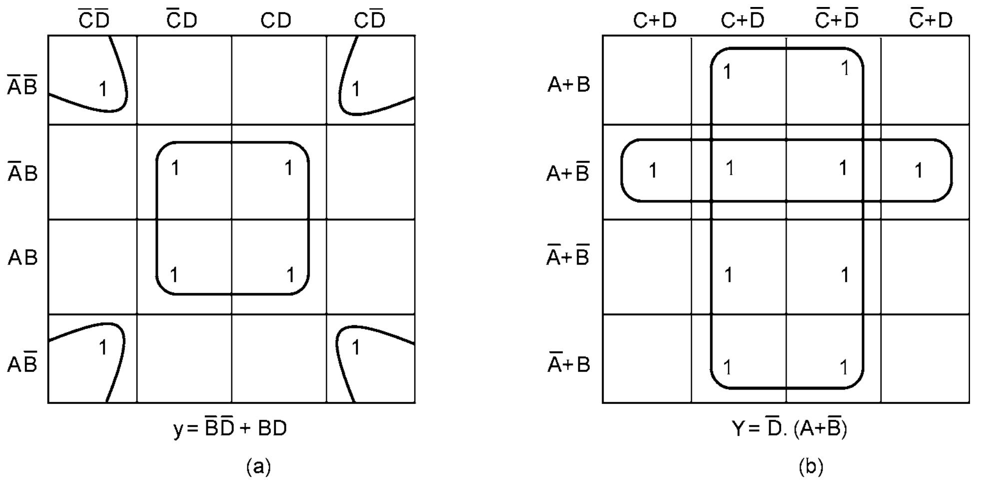


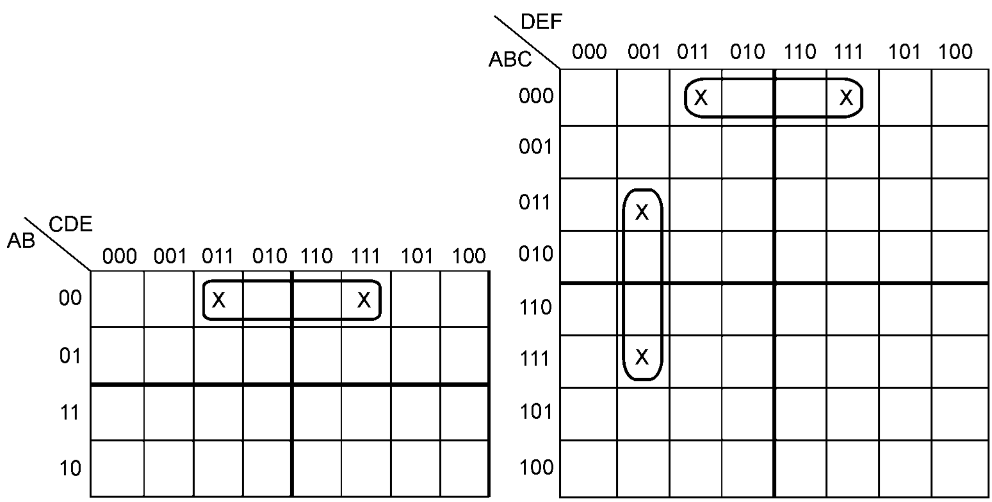
Figure 6.13 Group formation in minterm and maxterm Karnaugh maps.

6.6.2 Karnaugh Map for Boolean Expressions with a Larger Number of Variables

The construction of Karnaugh maps for a larger number of variables is a complex and cumbersome exercise, although manageable up to six variables. Five- and six-variable representative Karnaugh maps are shown in Figs 6.14(a) and (b) respectively. One important point to remember while forming groups in Karnaugh maps involving more than four variables is that terms equidistant from the central horizontal and central vertical lines are considered adjacent. These lines are shown thicker in Figs 6.14(a) and (b). Squares marked 'X' in Figs 6.14(a) and (b) are adjacent and therefore can be grouped.

Boolean expressions with more than four variables can also be represented by more than one four-variable map. Five-, six-, seven- and eight-variable Boolean expressions can be represented by two, four, eight and 16 four-variable maps respectively. In general, an n-variable Boolean expression can be represented by 2^{n-4} four-variable maps. In such multiple maps, groups are made as before, except that, in addition to adjacencies discussed earlier, corresponding squares in two adjacent maps are also considered adjacent and can therefore be grouped. We will illustrate the process of formation of groups in multiple Karnaugh maps with a larger number of variables with the help of examples. Consider the five-variable Boolean function given by the equation

$$Y = A.\overline{B}.\overline{C}.D.E + A.\overline{B}.C.D.E + \overline{A}.B.C.\overline{D}.E + A.B.\overline{C}.D.E + \overline{A}.\overline{B}.C.D.\overline{E} + \overline{A}.B.C.D.\overline{E} + A.B.C.D.\overline{E} + A.\overline{B}.C.D.\overline{E} + \overline{A}.B.C.\overline{D}.\overline{E} \tag{6.54}$$



(a) (b)

Figure 6.14 Five-variable and six-variable Karnaugh maps.

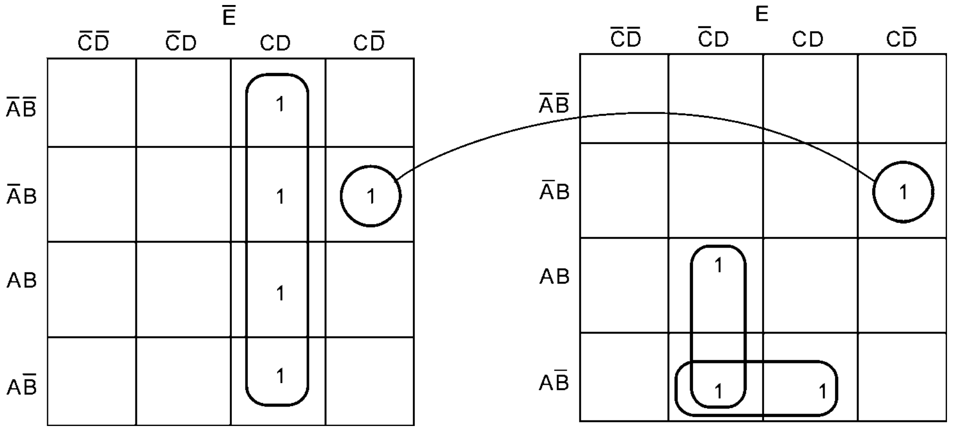


Figure 6.15 Multiple Karnaugh map for a five-variable Boolean function.

The multiple Karnaugh map for this five-variable expression is shown in Fig. 6.15. The construction of the Karnaugh map and the formation of groups are self-explanatory.

The minimized expression is given by the equation

$$Y = C.D.\overline{E} + \overline{A}.B.C.\overline{D} + A.\overline{C}.D.E + A.\overline{B}.D.E \tag{6.55}$$

As another illustration, consider a six-variable Boolean function given by the equation

$$\begin{aligned} Y =& \overline{A}.B.C.\overline{D}.\overline{E}.\overline{F} + A.B.\overline{C}.D.\overline{E}.F + \overline{A}.\overline{B}.\overline{C}.\overline{D}.\overline{E}.\overline{F} + A.B.C.D.E.F + A.\overline{B}.C.D.E.\overline{F} \\ & + \overline{A}.\overline{B}.\overline{C}.\overline{D}.\overline{E}.F + \overline{A}.B.C.\overline{D}.E.\overline{F} \end{aligned} \tag{6.56}$$

Figure 6.16 gives the Karnaugh map for this six-variable Boolean function, comprising four four-variable Karnaugh maps. The figure also shows the formation of groups. The minimized expression is given by the equation

$$Y = \overline{A}.\overline{B}.\overline{C}.\overline{D}.\overline{E} + \overline{A}.B.C.\overline{D}.\overline{F} + A.\overline{B}.C.D.E.\overline{F} + A.B.\overline{C}.D.\overline{E}.F + A.B.C.D.E.F \tag{6.57}$$

Example 6.11

Minimize the Boolean function

$$f(A, B, C) = \sum 0, 1, 3, 5 + \sum_{\phi} 2, 7$$

using the mapping method in both minimized sum-of-products and product-of-sums forms.

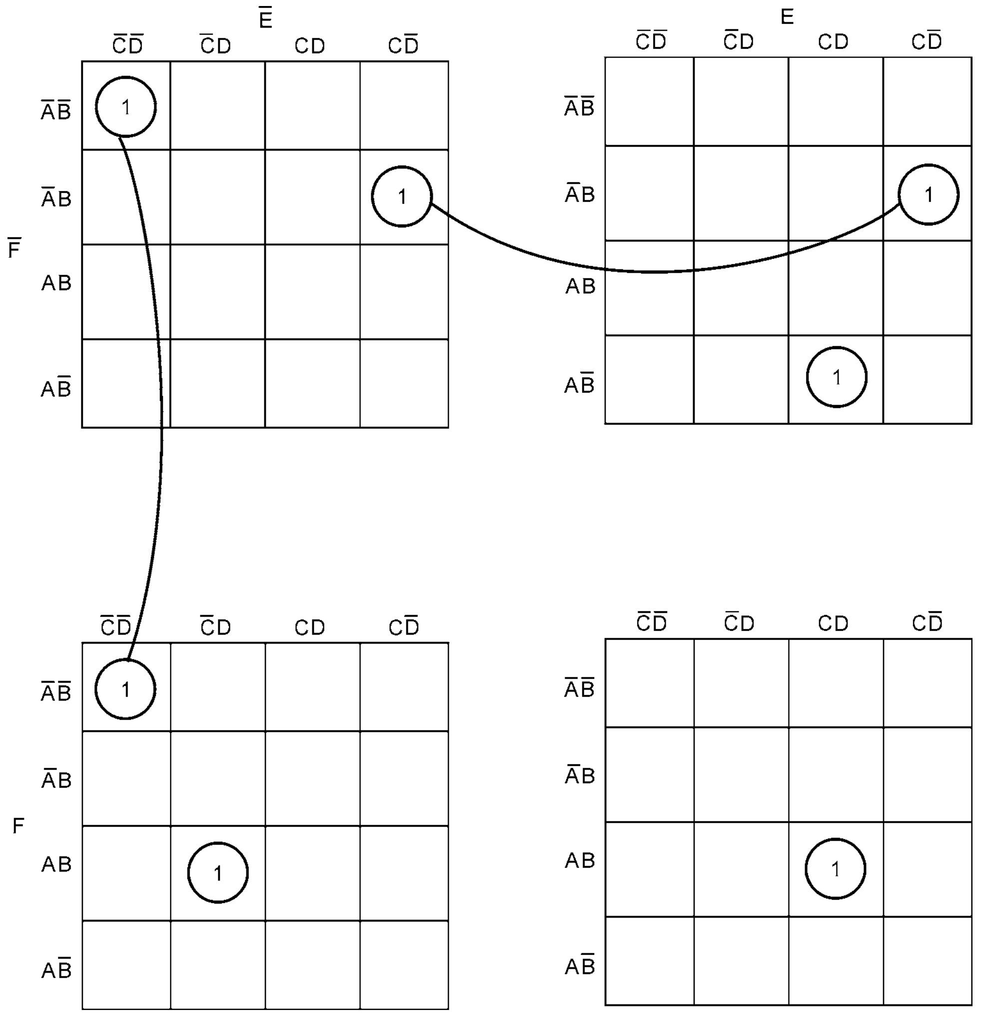


Figure 6.16 Multiple Karnaugh map for a six-variable Boolean function.

Solution

- $f(A, B, C) = \sum 0, 1, 3, 5 + \sum_{\phi} 2, 7 = \prod 4, 6 + \prod_{\phi} 2, 7.$
- From given Boolean functions in Σ and Π notation, we can write sum-of-products and product-of-sums Boolean expressions as follows:

$$f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + \overline{A}.B.C + A.\overline{B}.C \tag{6.58}$$

$$f(A, B, C) = (\overline{A} + B + C).(\overline{A} + \overline{B} + C) \tag{6.59}$$

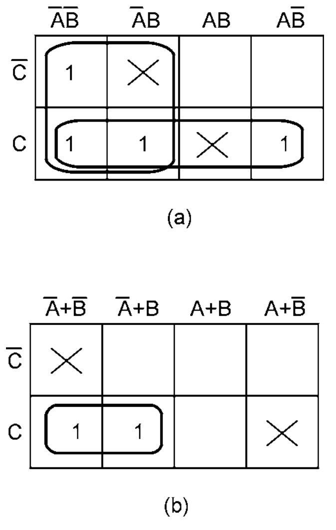


Figure 6.17 Example 6.11.

- The 'don't care' input combinations for the sum-of-products Boolean expression are $\overline{A}.B.\overline{C}$, $A.B.C$.
- The 'don't care' input combinations for the product-of-sums expression are $(A+\overline{B}+C).(\overline{A}+\overline{B}+\overline{C})$.
- The Karnaugh maps for the two cases are shown in Figs 6.17(a) and (b).
- The minimized sum-of-products and product-of-sums Boolean functions are respectively given by the equations

$$f(A, B, C) = C + \overline{A} \tag{6.60}$$

$$f(A, B, C) = \overline{A} + C \tag{6.61}$$

6.6.3 Karnaugh Maps for Multi-Output Functions

Karnaugh maps can be used for finding minimized Boolean expressions for multi-output functions. To begin with, a Karnaugh map is drawn for each function following the guidelines described in the earlier pages. In the second step, two-function Karnaugh maps are drawn. In the third step, three-function Karnaugh maps are drawn. The process continues until we have a single all-function Karnaugh map. As an illustration, for a logic system having four outputs, the first step would give four Karnaugh maps for individual functions. The second step would give six two-function Karnaugh maps (1–2, 1–3, 1–4, 2–3, 2–4 and 3–4). The third step would yield four three–function Karnaugh maps (1–2–3, 1–2–4, 1–3–4 and 2–3–4) and lastly we have one four-function Karnaugh map. A multifunction Karnaugh map is basically an intersection of the Karnaugh maps of the functions involved. That is, a '1' appears in a square of a multifunction map only if a '1' appears in the corresponding squares of the maps of all the relevant functions. To illustrate further, a two-function map involving functions 1 and 2 would be an intersection of maps for functions 1 and 2. In the two-function map, squares will have a '1' only when the corresponding squares in functions 1 and 2 also have a '1'. Figure 6.18 illustrates the formation of a three-function Karnaugh map from three given individual functions.

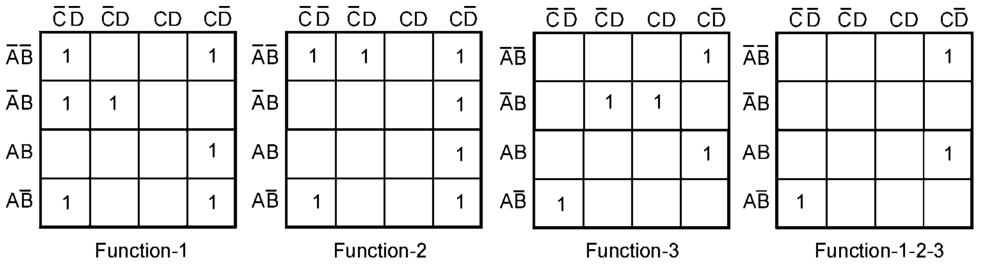


Figure 6.18 Three-function Karnaugh map.

The formation of groups begins with the largest multifunction map, which is nothing but the intersection of maps of all individual functions. Then we move to the Karnaugh maps one step down the order. The process continues until we reach the maps corresponding to individual functions. The groups in all the Karnaugh maps other than the largest map are formed subject to the condition that, once a group is identified in a certain function, then the same cannot be identified in any map of a subset of that function. For example, a group identified in a four-function map cannot be identified in a three-, two- or one-function map. With the formation of groups, prime implicants are identified. These prime implicants can be compiled in the form of a table along with input combinations of different output functions in the same way as for the tabular method to write minimized expressions. If the expressions corresponding to different output functions are not very complex, then the minimized expressions can even be written directly from the set of maps.

Example 6.12

Using Karnaugh maps, write the minimized Boolean expressions for the output functions of a two-output logic system whose outputs Y_1 and Y_2 are given by the following Boolean functions:

$$Y_1 = \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.C + \overline{A}.\overline{B}.\overline{C} \tag{6.62}$$

$$Y_2 = \overline{A}.\overline{B}.C + A.B.\overline{C} + \overline{A}.\overline{B}.\overline{C} + A.\overline{B}.C + A.B.C \tag{6.63}$$

Solution

The individual Karnaugh maps and the two-function map are shown in Fig. 6.19 along with the formation of groups. The prime implicant table along with the input combinations for the two output functions is given below:

Y_1				Y_2					Prime implicants		
000	010	100	111	000	001	101	110	111			
✓				✓					0	0	0
			✓					✓	1	1	1
✓	✓								0	–	0
✓		✓							–	0	0
							✓	✓	1	1	–
					✓	✓			–	0	1

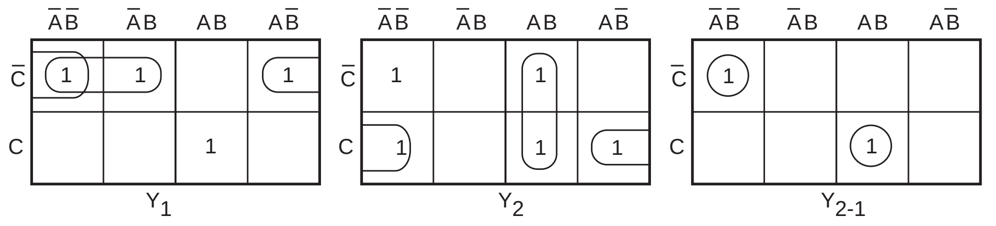


Figure 6.19 Example 6.12.

The minimized expressions for Y_1 and Y_2 are as follows:

$$Y_1 = \overline{B}.\overline{C} + \overline{A}.\overline{C} + A.B.C \tag{6.64}$$

$$Y_2 = A.B + \overline{A}.\overline{B}.\overline{C} + \overline{B}.C \tag{6.65}$$

Example 6.13

Write the simplified Boolean expression given by the Karnaugh map shown in Fig. 6.20.

Solution

- The Karnaugh map is shown in Fig. 6.21.
- Consider the group of four 1s at the top left of the map. It yields a term $\overline{A}.\overline{C}$.
- Consider the group of four 1s, two on the extreme left and two on the extreme right. This group yields a term $\overline{A}.\overline{D}$.
- The third group of two 1s is in the third row of the map. The third row corresponds to the intersection of A and B, as is clear from the map. Therefore, this group yields a term ABC.
- The simplified Boolean expression is given by $\overline{A}.\overline{C} + \overline{A}.\overline{D} + A.B.C$.

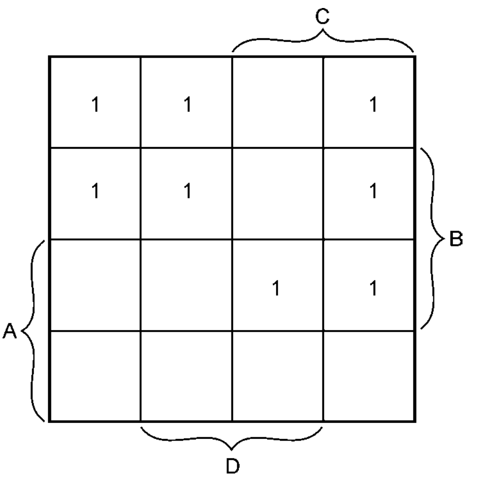


Figure 6.20 Example 6.13.

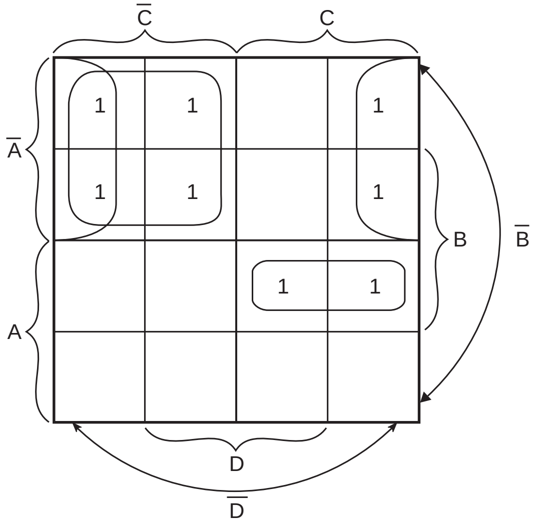


Figure 6.21 Solution to example 6.13.

Example 6.14

Minimizing a given Boolean expression using the Quine–McCluskey tabular method yields the following prime implicants: −0−0, −1−1, 1−10 and 0−00. Draw the corresponding Karnaugh map.

Solution

- As is clear from the prime implicants, the expression has four variables. If the variables are assumed to be A, B, C and D, then the given prime implicants correspond to the following terms:

1. $-0-0 \rightarrow \overline{B}.\overline{D}$.
2. $-1-1 \rightarrow B.D$.
3. $1-10 \rightarrow A.C.\overline{D}$.
4. $0-00 \rightarrow \overline{A}.\overline{C}.\overline{D}$.

- The Karnaugh map can now be drawn as shown in Fig. 6.22.

Example 6.15

$\overline{A}.B+C.D$ is a simplified Boolean expression of the expression $A.B.C.D+\overline{A}.\overline{B}.C.D+\overline{A}.B$. Determine if there are any 'don't care' entries.

Solution

The expanded version of the given expression is given by the equation

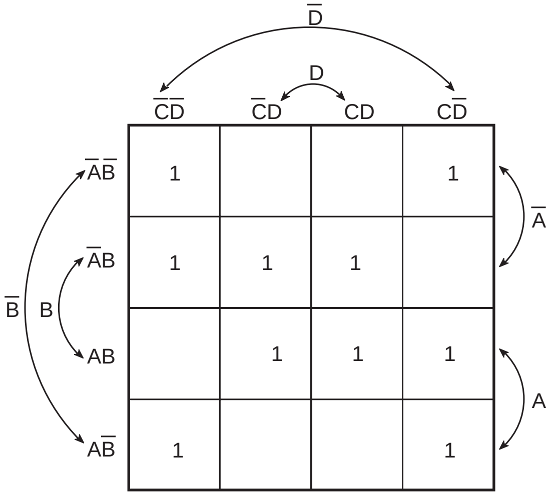


Figure 6.22 Solution to example 6.14.

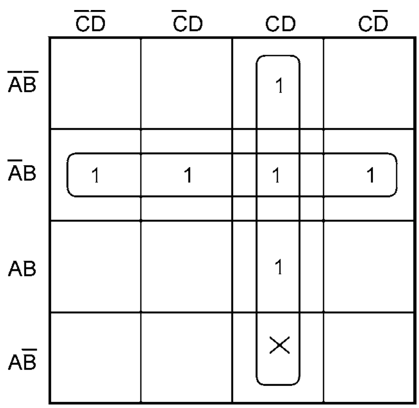


Figure 6.23 Example 6.15.

$$A.B.C.D+\overline{A}.\overline{B}.C.D+\overline{A}.B.(\overline{C}.\overline{D}+\overline{C}.D+C.D+C.\overline{D}) \quad (6.66)$$

$$=A.B.C.D+\overline{A}.\overline{B}.C.D+\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D+\overline{A}.B.C.D+\overline{A}.B.C.\overline{D} \quad (6.63)$$

The Karnaugh map for this Boolean expression is shown in Fig. 6.23. Now, if it is to be a simplified version of the expression $\overline{A}.B+C.D$, then the lowermost square in the CD column should not be empty. This implies that there is a 'don't care' entry. This has been reflected in the map by putting X in the relevant square. With the groups formed along with the 'don't care' entry, the simplified expression becomes the one stated in the problem.

Review Questions

1. Read the following statements carefully. For each one of these, identify the law associated with it. Define the law and illustrate the same with one or two examples.

   (a) While a NAND gate is equivalent to a bubbled OR gate, a NOR gate is equivalent to a bubbled AND gate.
   (b) When all the inputs of an AND gate or an OR gate are tied together to get a single-input, single-output gate, both AND and OR gates with all their inputs tied together produce an output that is the same as the input.
   (c) When a variable is ORed with its complement the result is a logic '1', and when it is ANDed with its complement the result is a logic '0', irrespective of the logic status of the variable.
   (d) When two variables are ANDed and the result of the AND operation is ORed with one of the variables, the result is that variable. Also, when two variables are ORed and the result of the OR operation is ANDed with one of the variables, the result is that variable.

2. Write both sum-of-products and product-of-sums Boolean expressions for (a) a two-input AND gate, (b) a two-input NAND-gate, (c) a two-input EX-OR gate and (d) a two-input NOR gate from their respective truth tables.
3. What do you understand by canonical and expanded forms of Boolean expressions? Illustrate with examples.
4. With the help of an example, prove that in an n-variable Karnaugh map, a group formed with 2^{n-m} 1s will yield a term having m literals, where m = 1, 2, 3, . . . , n.
5. With the help of an example, prove that the dual of the complement of a Boolean expression is the same as the complement of the dual of the same.

Problems

1. Simplify the following Boolean expressions:

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

   (a) 1; (b) $(\overline{A}+B).(C+D)$

2. (a) Find the dual of $A.B.C.\overline{D}+A.\overline{B}.\overline{C}.D+\overline{A}.\overline{B}.\overline{C}.\overline{D}$.
   (b) Find the complement of $A+[(B+\overline{C}).D+\overline{E}].F$.

   (a) $(A+B+C+\overline{D}).(A+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+\overline{D})$; $(b)\overline{A}.[(\overline{B}.C+\overline{D}).E+\overline{F}]$

3. The dual of the complement of a certain Boolean expression is given by $A.B.C+\overline{D}.E+B.\overline{C}.E$. Find the expression.

   $\overline{A}.\overline{B}.\overline{C}+D.\overline{E}+\overline{B}.C.\overline{E}$

4. Consider the Boolean expression given by

$$\overline{B}.\overline{C}.\overline{D}.\overline{E}+B.\overline{C}.\overline{D}.E+\overline{A}.B.C.E+A.B.C.D.E+A.\overline{B}.C.\overline{D}.\overline{E}+\overline{A}.B.\overline{C}.D.E+\overline{A}.\overline{B}.D.\overline{E}$$
$$+\overline{A}.\overline{B}.C.\overline{D}.\overline{E}+A.\overline{B}.\overline{C}.D.\overline{E}$$

The simplified version of this Boolean expression is given by $B.E+\overline{B}.D.\overline{E}+\overline{B}.\overline{D}.\overline{E}$. Determine if there are any 'don't care' entries. If yes, find them.

Yes, $A.B.\overline{C}.D.E,\ A.B.C.\overline{D}.E,\ A.\overline{B}.C.D.\overline{E}$

5. Write minterm and maxterm Boolean functions expressed by $f(A, B, C) = \Pi\ 0, 3, 7$

minterm: $\overline{A}.\overline{B}.C+\overline{A}.B.\overline{C}+A.\overline{B}.\overline{C}+A.\overline{B}.C+A.B.\overline{C}$
maxterm: $(A+B+C).(A+\overline{B}+\overline{C}).(\overline{A}+\overline{B}+\overline{C})$

6. Write a simplified maxterm Boolean expression for Π 0, 4, 5, 6, 7, 10, 14 using the Karnaugh mapping method.

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

7. Simplify the following Boolean functions using the Quine–McCluskey tabulation method:

   (a) $f(A, B, C, D, E, F, G) = \Sigma$ (20, 21, 28, 29, 52, 53, 60, 61);
   (b) $f(A, B, C, D, E, F) = \Sigma$ (6, 9, 13,18,19, 25, 26, 27, 29, 41, 45, 57, 61).

(a) $\overline{A}.C.E.\overline{F}$*; (b)* $C.\overline{E}.F+\overline{A}.B.\overline{D}.E+\overline{A}.\overline{B}.\overline{C}.D.E.\overline{F}$

8. (a) Simplify the Boolean function $f(X, Y, Z) = Y.Z+\overline{X}.\overline{Z}$ for the 'don't care' condition expressed as $X.\overline{Y}+X.Y.\overline{Z}+\overline{X}.\overline{Y}.Z$.
   (b) Simplify the Boolean function given by $f(A, B, C) = (A+B+C).(\overline{A}+B+\overline{C}).(A+\overline{B}+C)$ for the don't care condition expressed as $(\overline{A}+\overline{B}).(\overline{A}+B+C)$.

(a) 1; (b) $\overline{A}.C$

Further Reading

1. Holdsworth, B. and Woods, C. (2002) *Digital Logic Design*, Newnes, Oxford, UK.
2. Chen, W.-K. (2003) *Logic Design*, CRC Press, FL, USA.
3. Floyd, T. L. (2005) *Digital Fundamentals*, Prentice-Hall Inc., USA.
4. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.

7

Arithmetic Circuits

Beginning with this chapter, and in the two chapters following, we will take a comprehensive look at various building blocks used to design more complex combinational circuits. A combinational logic circuit is one where the output or outputs depend upon the present state of combination of the logic inputs. The logic gates discussed in Chapter 4 constitute the most fundamental building block of a combinational circuit. More complex combinational circuits such as adders and subtractors, multiplexers and demultiplexers, magnitude comparators, etc., can be implemented using a combination of logic gates. However, the aforesaid combinational logic functions and many more, including more complex ones, are available in monolithic IC form. A still more complex combinational circuit may be implemented using a combination of these functions available in IC form. In this chapter, we will cover devices used to perform arithmetic and other related operations. These include adders, subtractors, magnitude comparators and look-ahead carry generators. Particular emphasis is placed upon the functioning and design of these combinational circuits. The text has been adequately illustrated with the help of a large number of solved problems, the majority of which are design oriented.

7.1 Combinational Circuits

A *combinational circuit* is one where the output at any time depends only on the present combination of inputs at that point of time with total disregard to the past state of the inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions. The other category of logic circuits, called *sequential logic circuits*, comprises both logic gates and memory elements such as flip-flops. Owing to the presence of memory elements, the output in a sequential circuit depends upon not only the present but also the past state of inputs. Basic building blocks of sequential logic circuits are described in detail in Chapters 10 and 11.

Figure 7.1 shows the block schematic representation of a generalized combinational circuit having n input variables and m output variables or simply outputs. Since the number of input variables is

Digital Electronics: Principles, Devices and Applications Anil K. Maini
© 2007 John Wiley & Sons, Ltd. ISBN: 978-0-470-03214-5

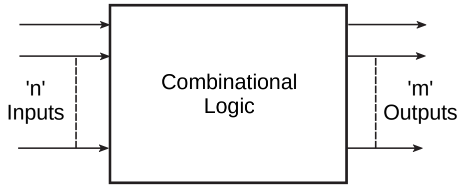


Figure 7.1 Generalized combinational circuit.

n, there are 2^n possible combinations of bits at the input. Each output can be expressed in terms of input variables by a Boolean expression, with the result that the generalized system of Fig. 7.1 can be expressed by m Boolean expressions. As an illustration, Boolean expressions describing the function of a four-input OR/NOR gate are given as

$$Y_1 \text{ (OR output)} = A+B+C+D \quad \text{and} \quad Y_2 \text{ (NOR output)} = \overline{A+B+C+D}$$

Also, each of the input variables may be available as only the normal input on the input line designated for the purpose. In that case, the complemented input, if desired, can be generated by using an inverter, as shown in Fig. 7.2(a), which illustrates the case of a four-input, two-output combinational function. Also, each of the input variables may appear in two wires, one representing the normal literal and the other representing the complemented one, as shown in Fig. 7.2(b).

In combinational circuits, input variables come from an external source and output variables feed an external destination. Both source and destination in the majority of cases are storage registers, and these

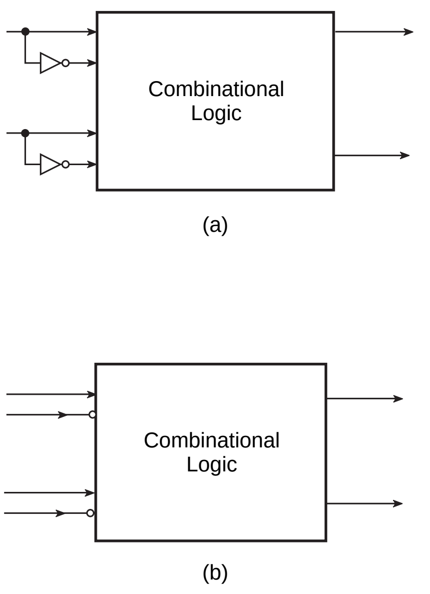


Figure 7.2 Combinational circuit with normal and complemented inputs.

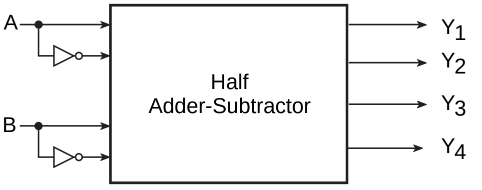


Figure 7.3 Two-input, four-output combinational circuit.

storage devices provide both normal as well as complemented outputs of the stored binary variable. As an illustration, Fig. 7.3 shows a simple two-input (A, B), four-output (Y_1, Y_2, Y_3, Y_4) combinational logic circuit described by the following Boolean expressions

$$Y_1 = A.\overline{B} + \overline{A}.B \tag{7.1}$$

$$Y_2 = A.\overline{B} + \overline{A}.B \tag{7.2}$$

$$Y_3 = A.B \tag{7.3}$$

$$Y_4 = \overline{A}.B \tag{7.4}$$

The implementation of these Boolean expressions needs both normal as well as complemented inputs. Incidentally, the combinational circuit shown is that of a half-adder–subtractor, with A and B representing the two bits to be added or subtracted and Y_1, Y_2, Y_3, Y_4 representing SUM, DIFFERENCE, CARRY and BORROW outputs respectively. Adder and subtractor circuits are discussed in Sections 7.3, 7.4 and 7.5.

7.2 Implementing Combinational Logic

The different steps involved in the design of a combinational logic circuit are as follows:

1. Statement of the problem.
2. Identification of input and output variables.
3. Expressing the relationship between the input and output variables.
4. Construction of a truth table to meet input–output requirements.
5. Writing Boolean expressions for various output variables in terms of input variables.
6. Minimization of Boolean expressions.
7. Implementation of minimized Boolean expressions.

These different steps are self-explanatory. One or two points, however, are worth mentioning here. There are various simplification techniques available for minimizing Boolean expressions, which have been discussed in the previous chapter. These include the use of theorems and identities, Karnaugh mapping, the Quinne–McCluskey tabulation method and so on. Also, there are various possible minimized forms of Boolean expressions. The following guidelines should be followed while choosing the preferred form for hardware implementation:

1. The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.
2. There should be a minimum number of interconnections, and the propagation time should be the shortest.
3. Limitation on the driving capability of the gates should not be ignored.

It is difficult to generalize as to what constitutes an acceptable simplified Boolean expression. The importance of each of the above-mentioned aspects is governed by the nature of application.

7.3 Arithmetic Circuits – Basic Building Blocks

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction, as was outlined in Chapter 2 dealing with binary arithmetic. We will begin with the basic building blocks that form the basis of all hardware used to perform the aforesaid arithmetic operations on binary numbers. These include half-adder, full adder, half-subtractor, full subtractor and controlled inverter.

7.3.1 Half-Adder

A *half-adder* is an arithmetic circuit block that can be used to add two bits. Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY. Figure 7.4 shows the truth table of a half-adder, showing all possible input combinations and the corresponding outputs.

The Boolean expressions for the SUM and CARRY outputs are given by the equations

$$\text{SUM } S = A.\overline{B} + \overline{A}.B \tag{7.5}$$

$$\text{CARRY } C = A.B \tag{7.6}$$

An examination of the two expressions tells that there is no scope for further simplification. While the first one representing the SUM output is that of an EX-OR gate, the second one representing the

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

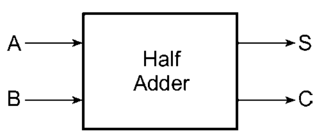


Figure 7.4 Truth table of a half-adder.

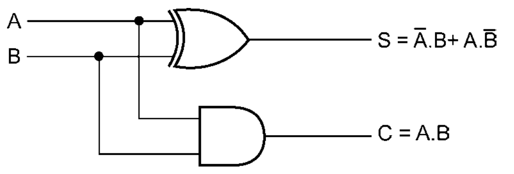


Figure 7.5 Logic implementation of a half-adder.

CARRY output is that of an AND gate. However, these two expressions can certainly be represented in different forms using various laws and theorems of Boolean algebra to illustrate the flexibility that the designer has in hardware-implementing as simple a combinational function as that of a half-adder. We have studied in Chapter 6 on Boolean algebra how various logic gates can be implemented in the form of either only NAND gates or NOR gates. Although the simplest way to hardware-implement a half-adder would be to use a two-input EX-OR gate for the SUM output and a two-input AND gate for the CARRY output, as shown in Fig. 7.5, it could also be implemented by using an appropriate arrangement of either NAND or NOR gates. Figure 7.6 shows the implementation of a half-adder with NAND gates only.

A close look at the logic diagram of Fig. 7.6 reveals that one part of the circuit implements a two-input EX-OR gate with two-input NAND gates. EX-OR implementation using NAND was discussed in the previous chapter. The AND gate required to generate CARRY output is implemented by complementing an already available NAND output of the input variables.

7.3.2 Full Adder

A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output. Such a building block becomes a necessity when it comes to adding binary numbers with a large number of bits. The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. Let us recall the procedure for adding larger binary numbers. We begin with the addition of LSBs of the two numbers. We record the sum under the LSB column and take the carry, if any, forward to the next higher column bits. As a result, when we add the next adjacent higher column bits, we would be required to add three bits if there were a carry from the previous addition. We have a similar situation for the other higher column bits

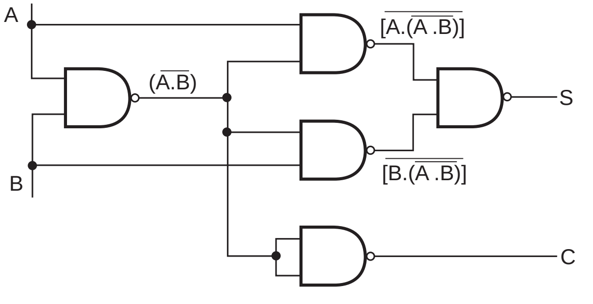
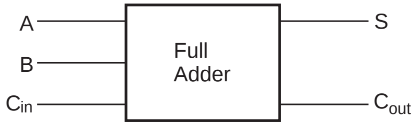


Figure 7.6 Half-adder implementation using NAND gates.



A	B	C_{in}	SUM (S)	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 7.7 Truth table of a full adder.

also until we reach the MSB. A full adder is therefore essential for the hardware implementation of an adder circuit capable of adding larger binary numbers. A half-adder can be used for addition of LSBs only.

Figure 7.7 shows the truth table of a full adder circuit showing all possible input combinations and corresponding outputs. In order to arrive at the logic circuit for hardware implementation of a full adder, we will firstly write the Boolean expressions for the two output variables, that is, the SUM and CARRY outputs, in terms of input variables. These expressions are then simplified by using any of the simplification techniques described in the previous chapter. The Boolean expressions for the two output variables are given in Equation (7.7) for the SUM output (S) and in Equation (6.6) for the CARRY output (C_{out}):

$$S = \overline{A}.\overline{B}.C_{in} + \overline{A}.B.\overline{C}_{in} + A.\overline{B}.\overline{C}_{in} + A.B.C_{in} \tag{7.7}$$

$$C_{out} = \overline{A}.B.C_{in} + A.\overline{B}.C_{in} + A.B.\overline{C}_{in} + A.B.C_{in} \tag{7.8}$$

The next step is to simplify the two expressions. We will do so with the help of the Karnaugh mapping technique. Karnaugh maps for the two expressions are given in Fig. 7.8(a) for the SUM output and Fig. 7.8(b) for the CARRY output. As is clear from the two maps, the expression for the SUM (S) output cannot be simplified any further, whereas the simplified Boolean expression for C_{out} is given by the equation

$$C_{out} = B.C_{in} + A.B + A.C_{in} \tag{7.9}$$

Figure 7.9 shows the logic circuit diagram of the full adder. A full adder can also be seen to comprise two half-adders and an OR gate. The expressions for SUM and CARRY outputs can be rewritten as follows:

$$S = \overline{C}_{in}.(\overline{A}.B + A.\overline{B}) + C_{in}.(A.B + \overline{A}.\overline{B})$$

$$S = \overline{C}_{in}.(\overline{A}.B + A.\overline{B}) + C_{in}.(\overline{\overline{A}.B + A.\overline{B}}) \tag{7.10}$$

Similarly, the expression for CARRY output can be rewritten as follows:

$$\begin{aligned} C_{out} &= B.C_{in}.(A + \overline{A}) + A.B + A.C_{in}.(B + \overline{B}) \\ &= A.B + A.B.C_{in} + \overline{A}.B.C_{in} + A.B.C_{in} + A.\overline{B}.C_{in} = A.B + A.B.C_{in} + \overline{A}.B.C_{in} + A.\overline{B}.C_{in} \\ &= A.B.(1 + C_{in}) + C_{in}.(\overline{A}.B + A.\overline{B}) \end{aligned}$$

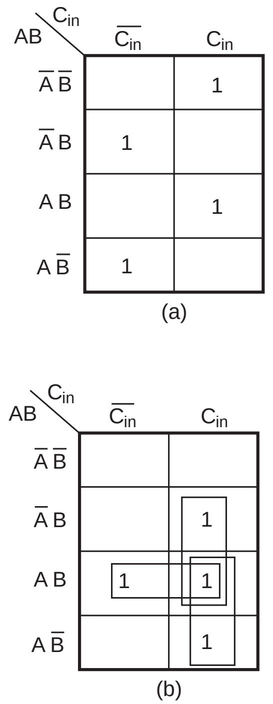


Figure 7.8 Karnaugh maps for the sum and carry-out of a full adder.

$$C_{\text{out}} = A.B + C_{\text{in}}.(\overline{A}.B + A.\overline{B}) \tag{7.11}$$

Boolean expression (7.10) can be implemented with a two-input EX-OR gate provided that one of the inputs is C_{in} and the other input is the output of another two-input EX-OR gate with A and B as its inputs. Similarly, Boolean expression (7.11) can be implemented by ORing two minterms. One of them is the AND output of A and B. The other is also the output of an AND gate whose inputs are C_{in} and the output of an EX-OR operation on A and B. The whole idea of writing the Boolean expressions in this modified form was to demonstrate the use of a half-adder circuit in building a full adder. Figure 7.10(a) shows logic implementation of Equations (7.10) and (7.11). Figure 7.10(b) is nothing but Fig. 7.10(a) redrawn with the portion of the circuit representing a half-adder replaced with a block.

The full adder of the type described above forms the basic building block of binary adders. However, a single full adder circuit can be used to add one-bit binary numbers only. A cascade arrangement of these adders can be used to construct adders capable of adding binary numbers with a larger number of bits. For example, a four-bit binary adder would require four full adders of the type shown in Fig. 7.10 to be connected in cascade. Figure 7.11 shows such an arrangement. $(A_3A_2A_1A_0)$ and $(B_3B_2B_1B_0)$ are the two binary numbers to be added, with A_0 and B_0 representing LSBs and A_3 and B_3 representing MSBs of the two numbers.

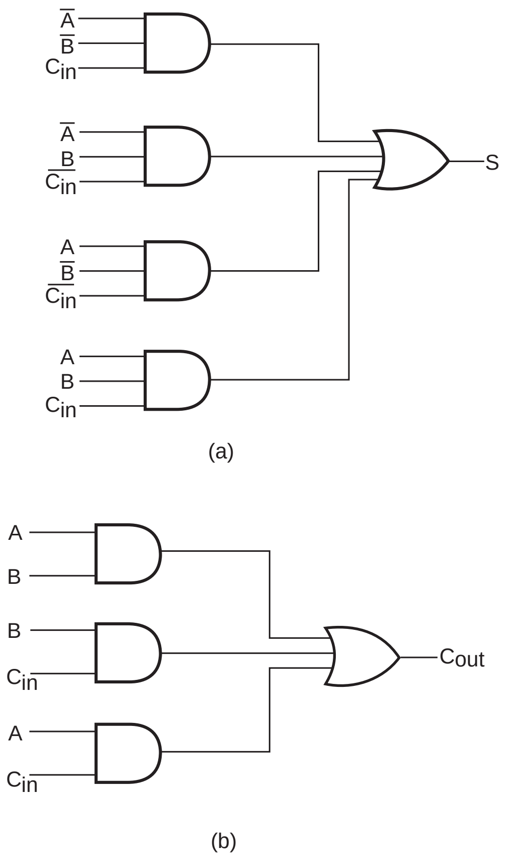


Figure 7.9 Logic circuit diagram of a full adder.

7.3.3 Half-Subtractor

We have seen in Chapter 3 on digital arithmetic how subtraction of two given binary numbers can be carried out by adding 2's complement of the subtrahend to the minuend. This allows us to do a subtraction operation with adder circuits. We will study the use of adder circuits for subtraction operations in the following pages. Before we do that, we will briefly look at the counterparts of half-adder and full adder circuits in the half-subtractor and full subtractor for direct implementation of subtraction operations using logic gates.

A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a '1' has been borrowed to perform the subtraction. The truth table of a half-subtractor, as shown in Fig. 7.12, explains this further. The Boolean expressions for the two outputs are given by the equations

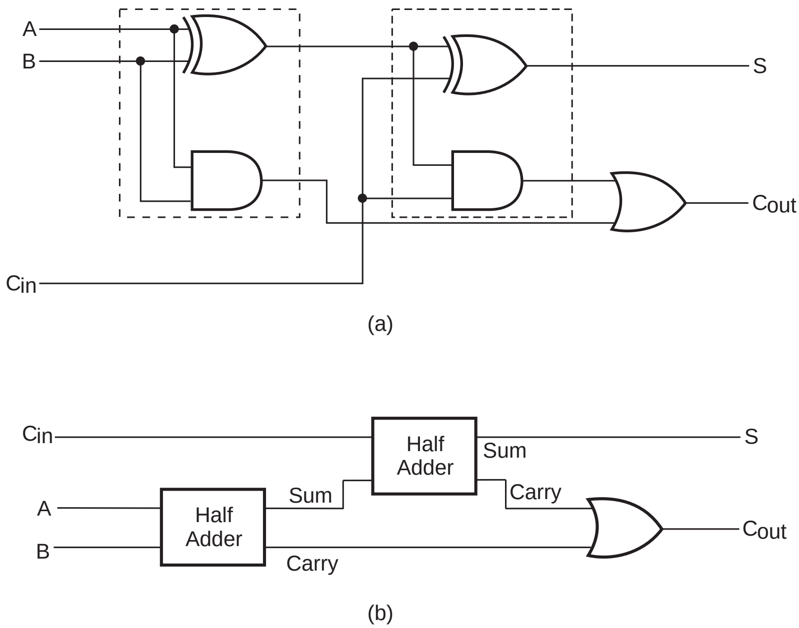


Figure 7.10 Logic implementation of a full adder with half-adders.

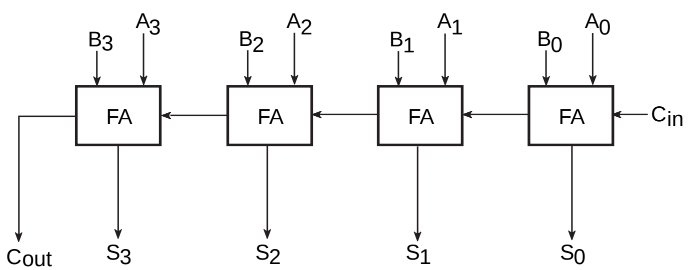
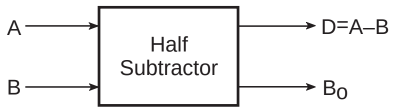


Figure 7.11 Four-bit binary adder.

$$D = \overline{A}.B + A.\overline{B} \tag{7.12}$$

$$B_o = \overline{A}.B \tag{7.13}$$

It is obvious that there is no further scope for any simplification of the Boolean expressions given by Equations (7.12) and (7.13). While the expression for the DIFFERENCE (D) output is that of



A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Figure 7.12 Half-subtractor.

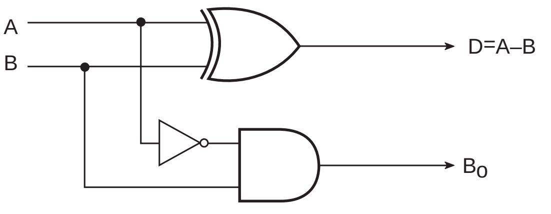


Figure 7.13 Logic diagram of a half-subtractor.

an EX-OR gate, the expression for the BORROW output (B_o) is that of an AND gate with input A complemented before it is fed to the gate. Figure 7.13 shows the logic implementation of a half-subtractor. Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, that is, the minuend, is complemented, an AND gate can be used to implement the BORROW output. Note the similarities between the logic diagrams of Fig. 7.5 (half-adder) and Fig. 7.13 (half-subtractor).

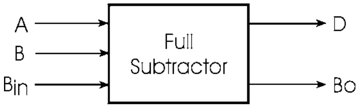
7.3.4 Full Subtractor

A *full subtractor* performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a '1' has already been borrowed by the previous adjacent lower minuend bit or not. As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o. The BORROW output bit tells whether the minuend bit needs to borrow a '1' from the next possible higher minuend bit. Figure 7.14 shows the truth table of a full subtractor.

The Boolean expressions for the two output variables are given by the equations

$$D = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + A.\overline{B}.\overline{B}_{in} + A.B.B_{in} \tag{7.14}$$

$$B_o = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + \overline{A}.B.B_{in} + A.B.B_{in} \tag{7.15}$$



Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_o)
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

Figure 7.14 Truth table of a full subtractor.

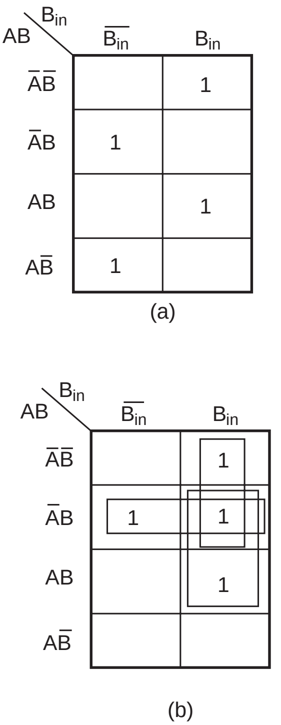
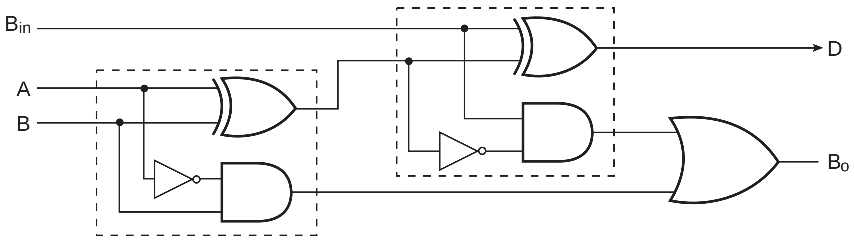


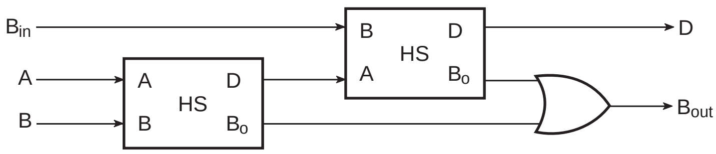
Figure 7.15 Karnaugh maps for difference and borrow outputs.

The Karnaugh maps for the two expressions are given in Fig. 7.15(a) for DIFFERENCE output D and in Fig. 7.15(b) for BORROW output B_o. As is clear from the two Karnaugh maps, no simplification is possible for the difference output D. The simplified expression for B_o is given by the equation

$$B_o = \overline{A}.B + \overline{A}.B_{in} + B.B_{in} \tag{7.16}$$



(a)



(b)

Figure 7.16 Logic implementation of a full subtractor with half-subtractors.

If we compare these expressions with those derived earlier in the case of a full adder, we find that the expression for DIFFERENCE output D is the same as that for the SUM output. Also, the expression for BORROW output B_o is similar to the expression for CARRY-OUT C_o. In the case of a half-subtractor, the A input is complemented. By a similar analysis it can be shown that a full subtractor can be implemented with half-subtractors in the same way as a full adder was constructed using half-adders. Relevant logic diagrams are shown in Figs 7.16(a) and (b) corresponding to Figs 7.10(a) and (b) respectively for a full adder.

Again, more than one full subtractor can be connected in cascade to perform subtraction on two larger binary numbers. As an illustration, Fig. 7.17 shows a four-bit subtractor.

7.3.5 Controlled Inverter

A *controlled inverter* is needed when an adder is to be used as a subtractor. As outlined earlier, subtraction is nothing but addition of the 2's complement of the subtrahend to the minuend. Thus, the first step towards practical implementation of a subtractor is to determine the 2's complement of the subtrahend. And for this, one needs firstly to find 1's complement. A controlled inverter is used to find 1's complement. A one-bit controlled inverter is nothing but a two-input EX-OR gate with one of its inputs treated as a control input, as shown in Fig. 7.18(a). When the control input is LOW, the input bit is passed as such to the output. (Recall the truth table of an EX-OR gate.) When the control input is HIGH, the input bit gets complemented at the output. Figure 7.18(b) shows an eight-bit controlled inverter of this type. When the control input is LOW, the output (Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0) is the same as the input (A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0). When the control input is HIGH, the output is 1's complement

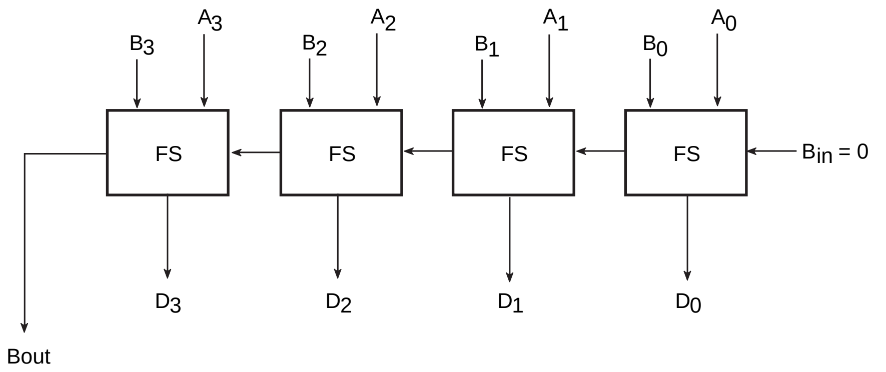


Figure 7.17 Four-bit subtractor.

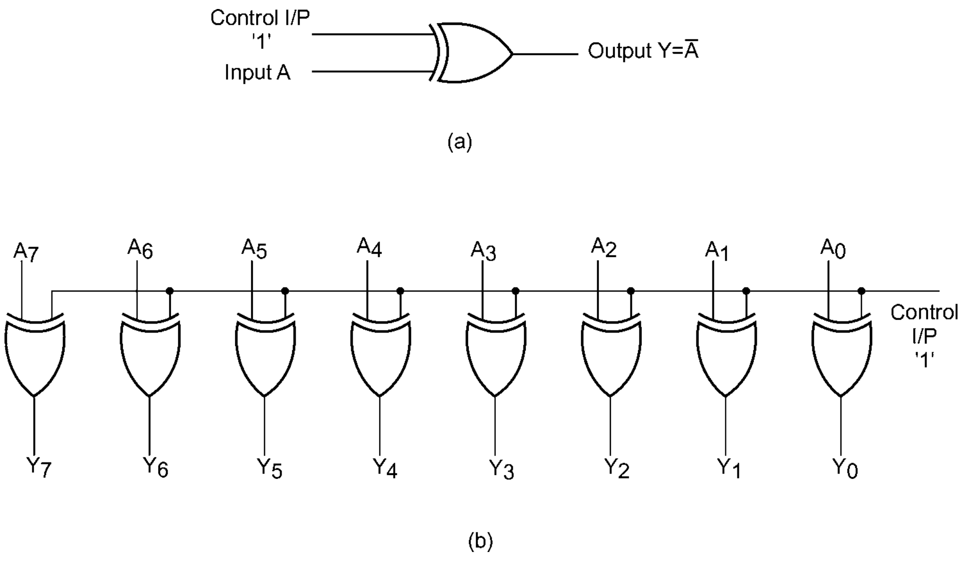


Figure 7.18 (a) One-bit controlled inverter and (b) eight-bit controlled inverter.

of the input. As an example, 11010010 at the input would produce 00101101 at the output when the control input is in a logic ‘1’ state.

7.4 Adder–Subtractor

Subtraction of two binary numbers can be accomplished by adding 2’s complement of the subtrahend to the minuend and disregarding the final carry, if any. If the MSB bit in the result of addition is

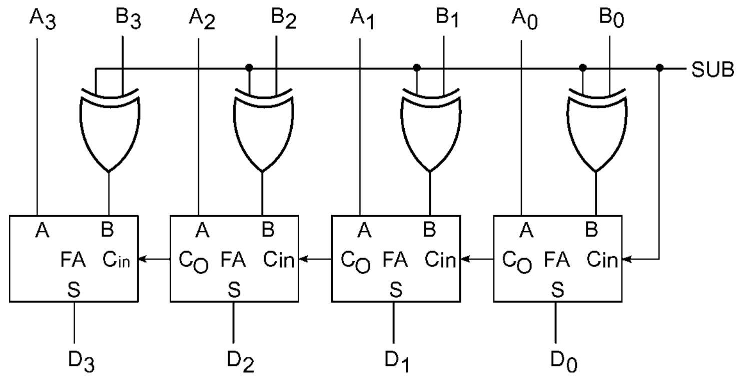


Figure 7.19 Four-bit adder-subtractor.

a '0', then the result of addition is the correct answer. If the MSB bit is a '1', this implies that the answer has a negative sign. The true magnitude in this case is given by 2's complement of the result of addition.

Full adders can be used to perform subtraction provided we have the necessary additional hardware to generate 2's complement of the subtrahend and disregard the final carry or overflow. Figure 7.19 shows one such hardware arrangement. Let us see how it can be used to perform subtraction of two four-bit binary numbers. A close look at the diagram would reveal that it is the hardware arrangement for a four-bit binary adder, with the exception that the bits of one of the binary numbers are fed through controlled inverters. The control input here is referred to as the SUB input. When the SUB input is in logic '0' state, the four bits of the binary number ($B_3\,B_2\,B_1\,B_0$) are passed on as such to the B inputs of the corresponding full adders. The outputs of the full adders in this case give the result of addition of the two numbers. When the SUB input is in logic '1' state, four bits of one of the numbers, ($B_3\,B_2\,B_1\,B_0$) in the present case, get complemented. If the same '1' is also fed to the CARRY-IN of the LSB full adder, what we finally achieve is the addition of 2's complement and not 1's complement. Thus, in the adder arrangement of Fig. 7.19, we are basically adding 2's complement of ($B_3\,B_2\,B_1\,B_0$) to ($A_3\,A_2\,A_1\,A_0$). The outputs of the full adders in this case give the result of subtraction of the two numbers. The arrangement shown achieves $A - B$. The final carry (the CARRY-OUT of the MSB full adder) is ignored if it is not displayed.

For implementing an eight-bit adder–subtractor, we will require eight full adders and eight two-input EX-OR gates. Four-bit full adders and quad two-input EX-OR gates are individually available in integrated circuit form. A commonly used four-bit adder in the TTL family is the type number 7483. Also, type number 7486 is a quad two-input EX-OR gate in the TTL family. Figure 7.20 shows a four-bit binary adder–subtractor circuit implemented with 7483 and 7486. Two each of 7483 and 7486 can be used to construct an eight-bit adder–subtractor circuit.

7.5 BCD Adder

A *BCD adder* is used to perform the addition of BCD numbers. A BCD digit can have any of the ten possible four-bit binary representations, that is, 0000, 0001, . . . , 1001, the equivalent of decimal numbers 0, 1, . . . , 9. When we set out to add two BCD digits and we assume that there is an input carry too, the highest binary number that we can get is the equivalent of decimal number 19 ($9 + 9 + 1$).

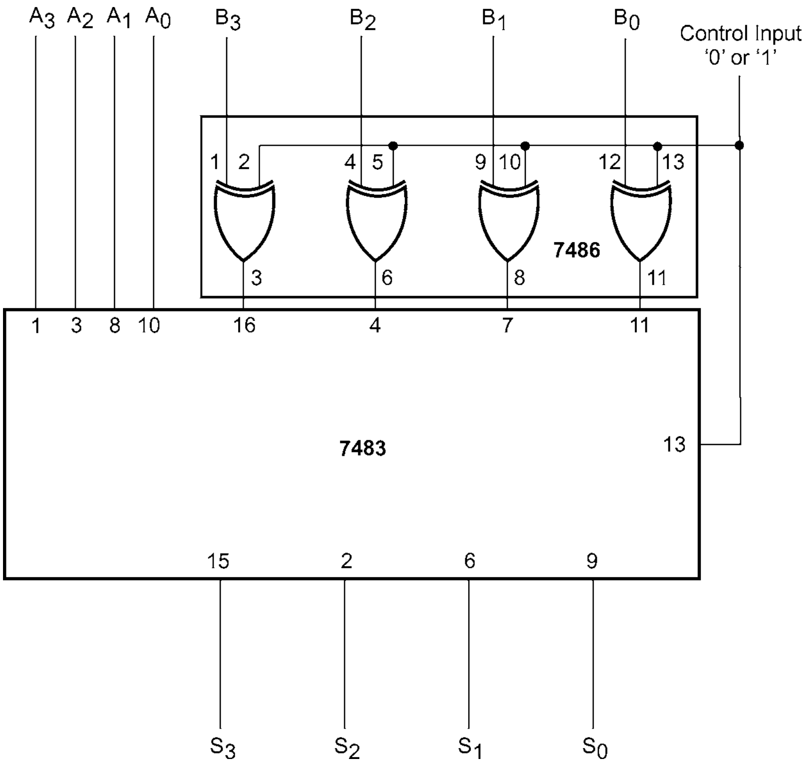


Figure 7.20 Four-bit adder-subtractor.

This binary number is going to be $(10011)_2$. On the other hand, if we do BCD addition, we would expect the answer to be $(0001\ 1001)_{\mathrm{BCD}}$. And if we restrict the output bits to the minimum required, the answer in BCD would be $(1\ 1001)_{\mathrm{BCD}}$. Table 7.1 lists the possible results in binary and the expected results in BCD when we use a four-bit binary adder to perform the addition of two BCD digits. It is clear from the table that, as long as the sum of the two BCD digits remains equal to or less than 9, the four-bit adder produces the correct BCD output.

The binary sum and the BCD sum in this case are the same. It is only when the sum is greater than 9 that the two results are different. It can also be seen from the table that, for a decimal sum greater than 9 (or the equivalent binary sum greater than 1001), if we add 0110 to the binary sum, we can get the correct BCD sum and the desired carry output too. The Boolean expression that can apply the necessary correction is written as

$$C = K + Z_3.Z_2 + Z_3.Z_1 \tag{7.17}$$

Equation (7.17) implies the following. A correction needs to be applied whenever $K = 1$. This takes care of the last four entries. Also, a correction needs to be applied whenever both Z_3 and Z_2 are '1'. This takes care of the next four entries from the bottom, corresponding to a decimal sum equal to

Table 7.1 Results in binary and the expected results in BCD using a four-bit binary adder to perform the addition of two BCD digits.

Decimal sum	Binary sum					BCD sum				
	K	Z_3	Z_2	Z_1	Z_0	C	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

12, 13, 14 and 15. For the remaining two entries corresponding to a decimal sum equal to 10 and 11, a correction is applied for both Z_3 and Z_1, being '1'. While hardware-implementing, 0110 can be added to the binary sum output with the help of a second four-bit binary adder. The correction logic as dictated by the Boolean expression (7.17) should ensure that (0110) gets added only when the above expression is satisfied. Otherwise, the sum output of the first binary adder should be passed on as such to the final output, which can be accomplished by adding (0000) in the second adder. Figure 7.21 shows the logic arrangement of a BCD adder capable of adding two BCD digits with the help of two four-bit binary adders and some additional combinational logic.

The BCD adder described in the preceding paragraphs can be used to add two single-digit BCD numbers only. However, a cascade arrangement of single-digit BCD adder hardware can be used to perform the addition of multiple-digit BCD numbers. For example, an n-digit BCD adder would require n such stages in cascade. As an illustration, Fig. 7.22 shows the block diagram of a circuit for the addition of two three-digit BCD numbers. The first BCD adder, labelled LSD (Least Significant Digit), handles the least significant BCD digits. It produces the sum output ($S_3\,S_2\,S_1\,S_0$), which is the BCD code for the least significant digit of the sum. It also produces an output carry that is fed as an input carry to the next higher adjacent BCD adder. This BCD adder produces the sum output ($S_7\,S_6\,S_5\,S_4$), which is the BCD code for the second digit of the sum, and a carry output. This output carry serves as an input carry for the BCD adder representing the most significant digits. The sum outputs ($S_{11}\,S_{10}\,S_9\,S_8$) represent the BCD code for the MSD of the sum.

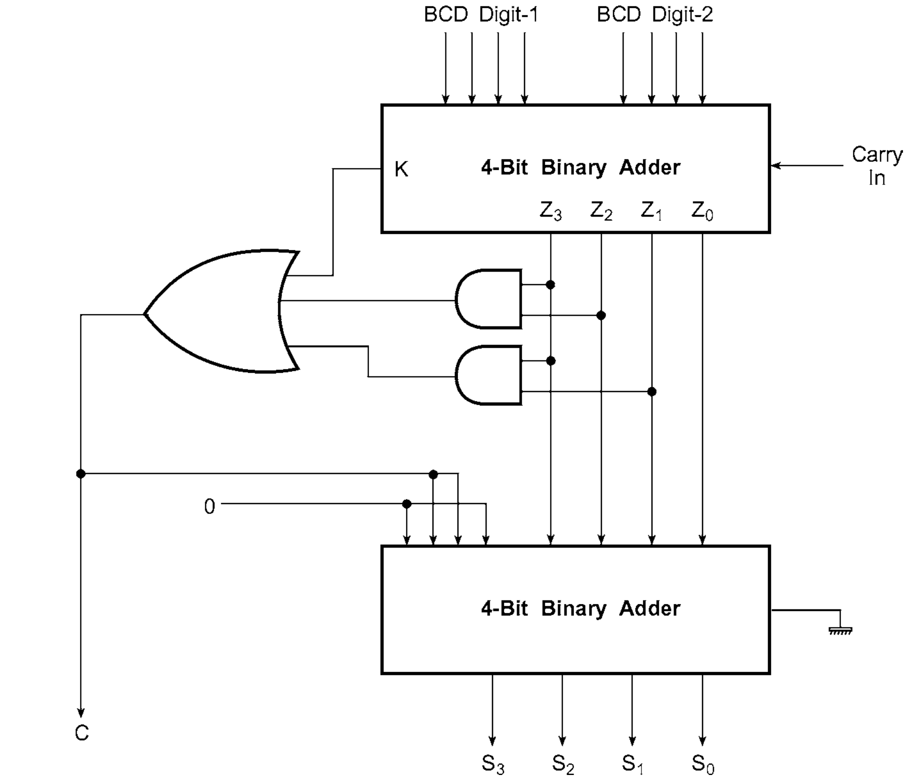


Figure 7.21 Single-digit BCD adder.

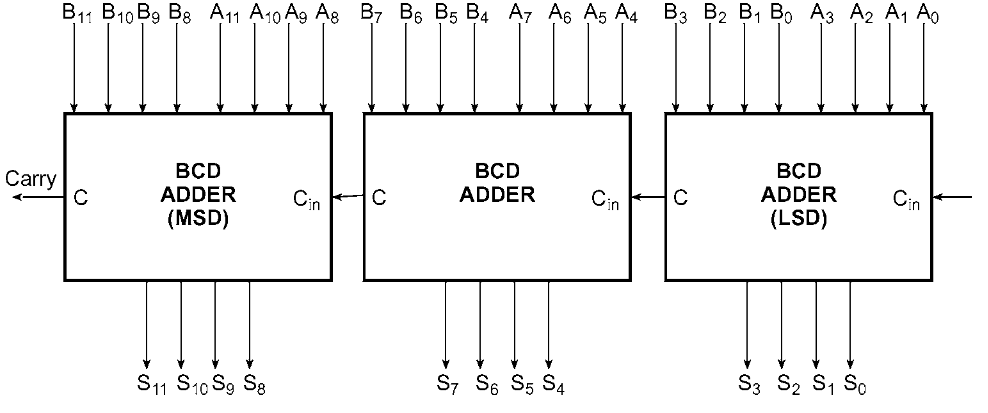


Figure 7.22 Three-digit BCD adder.

Example 7.1

For the half-adder circuit of Fig. 7.23(a), the inputs applied at A and B are as shown in Fig. 7.23(b). Plot the corresponding SUM and CARRY outputs on the same scale.

Solution

The SUM and CARRY waveforms can be plotted from our knowledge of the truth table of the half-adder. All that we need to remember to solve this problem is that $0+0$ yields a '0' as the SUM output and a '0' as the CARRY. $0+1$ or $1+0$ yield '1' as the SUM output and '0' as the CARRY. $1+1$ produces a '0' as the SUM output and a '1' as the CARRY. The output waveforms are as shown in Fig. 7.24.

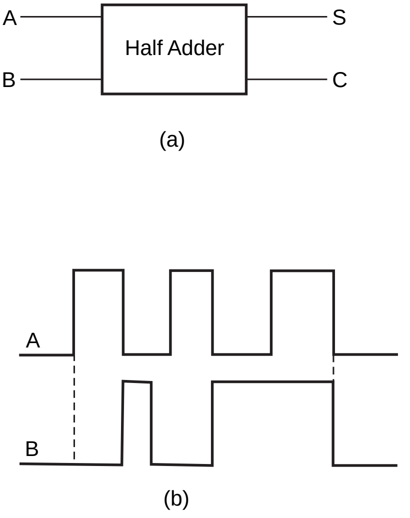


Figure 7.23 Example 7.1.

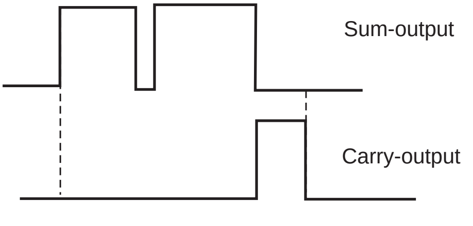


Figure 7.24 Solution to example 7.1.

Example 7.2

Given the relevant Boolean expressions for half-adder and half-subtractor circuits, design a half-adder–subtractor circuit that can be used to perform either addition or subtraction on two one-bit numbers. The desired arithmetic operation should be selectable from a control input.

Solution

Boolean expressions for the half-adder and half-subtractor are given as follows:

Half-adder

$$\text{SUM output} = \overline{A}B + A\overline{B} \quad \text{and} \quad \text{CARRY output} = AB$$

Half-subtractor

$$\text{DIFFERENCE output} = \overline{A}B + A\overline{B} \quad \text{and} \quad \text{BORROW output} = \overline{A}B$$

If we use a controlled inverter for complementing A in the case of the half-subtractor circuit, then the same hardware can also be used to add two one-bit numbers. Figure 7.25 shows the logic circuit diagram. When the control input is ‘0’, input variable A is passed uncomplemented to the input of the NAND gate. In this case, the AND gate generates the CARRY output of the addition operation. The EX-OR gate generates the SUM output. On the other hand, when the control input is ‘1’, the AND gate generates the BORROW output and the EX-OR gate generates the DIFFERENCE output. Thus, ‘0’ at the control input makes it a half-adder, while ‘1’ at the control input makes it a half-subtractor.

Example 7.3

Refer to Fig. 7.26. Write the simplified Boolean expressions for DIFFERENCE and BORROW outputs.

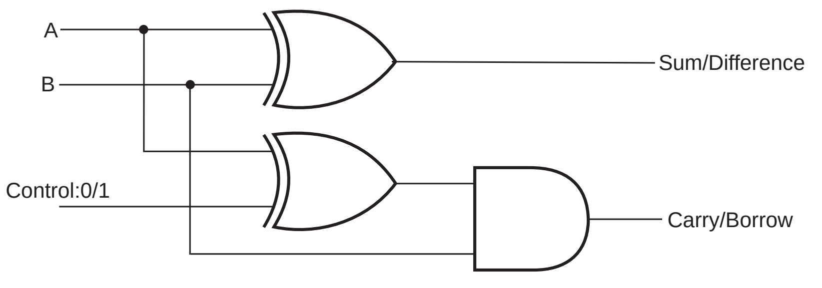


Figure 7.25 Solution to example 7.2.

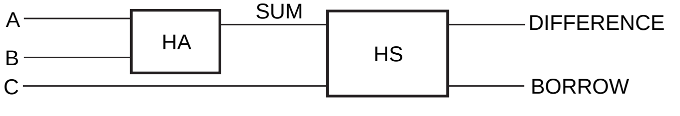


Figure 7.26 Example 7.3.

Solution

Let us assume that the two inputs to the half-subtractor circuit are X and Y, with X equal to the SUM output of the half-adder and Y equal to C. DIFFERENCE and BORROW outputs can then be expressed as follows:

$$\text{DIFFERENCE output} = X \oplus Y = \overline{X}.Y + X.\overline{Y} \quad \text{and} \quad \text{BORROW output} = \overline{X}.Y$$

Also, $X = \overline{A}.B + A.\overline{B}$ and $Y = C$.

Substituting the values of X and Y, we obtain

$$\begin{aligned}\text{DIFFERENCE output} &= (\overline{\overline{A}.B + A.\overline{B}}).C + (\overline{A}.B + A.\overline{B}).\overline{C} = (A.B + \overline{A}.\overline{B}).C + (\overline{A}.B + A.\overline{B}).\overline{C} \\ &= A.B.C + \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C}\end{aligned}$$

$$\text{BORROW output} = \overline{X}.Y = (\overline{\overline{A}.B + A.\overline{B}}).C = (A.B + \overline{A}.\overline{B}).C = A.B.C + \overline{A}.\overline{B}.C$$

Example 7.4

Design an eight-bit adder–subtractor circuit using four-bit binary adders, type number 7483, and quad two-input EX-OR gates, type number 7486. Assume that pin connection diagrams of these ICs are available to you.

Solution

IC 7483 is a four-bit binary adder, which means that it can add two four-bit binary numbers. In order to add two eight-bit numbers, we need to use two 7483s in cascade. That is, the CARRY-OUT (pin 14) of the 7483 handling less significant four bits is fed to the CARRY-IN (pin 13) of the 7483 handling more significant four bits. Also, if $(A_0 \ldots A_7)$ and $(B_0 \ldots B_7)$ are the two numbers to be operated upon, and if the objective is to compute $A - B$, bits B_0, B_1, B_2, B_3, B_4, B_5, B_6 and B_7 are complemented using EX-OR gates. One of the inputs of all EX-OR gates is tied together to form the control input. When the control input is in logic '1' state, bits B_0 to B_7 get complemented. Also, feeding this logic '1' to the CARRY-IN of lower 7483 ensures that we get 2's complement of bits $(B_0 \ldots B_7)$. Therefore, when the control input is in logic '1' state, the two's complement of $(B_0 \ldots B_7)$ is added to $(A_0 \ldots A_7)$. The output is therefore $A - B$. A logic '0' at the control input allows $(B_0 \ldots B_7)$ to pass through EX-OR gates uncomplemented, and the output in that case is $A + B$. Figure 7.27 shows the circuit diagram.

Example 7.5

The logic diagram of Fig. 7.28 performs the function of a very common arithmetic building block. Identify the logic function.

Solution

Writing Boolean expressions for X and Y,

$$X = \overline{(\overline{\overline{A}.B}).(\overline{A.\overline{B}})} = (\overline{\overline{\overline{A}.B}} + \overline{\overline{A.\overline{B}}}) = \overline{A}.B + A.\overline{B} \quad \text{and} \quad Y = \overline{(\overline{A} + \overline{B})} = A.B$$

Boolean expressions for X and Y are those of a half-adder. X and Y respectively represent SUM and CARRY outputs.

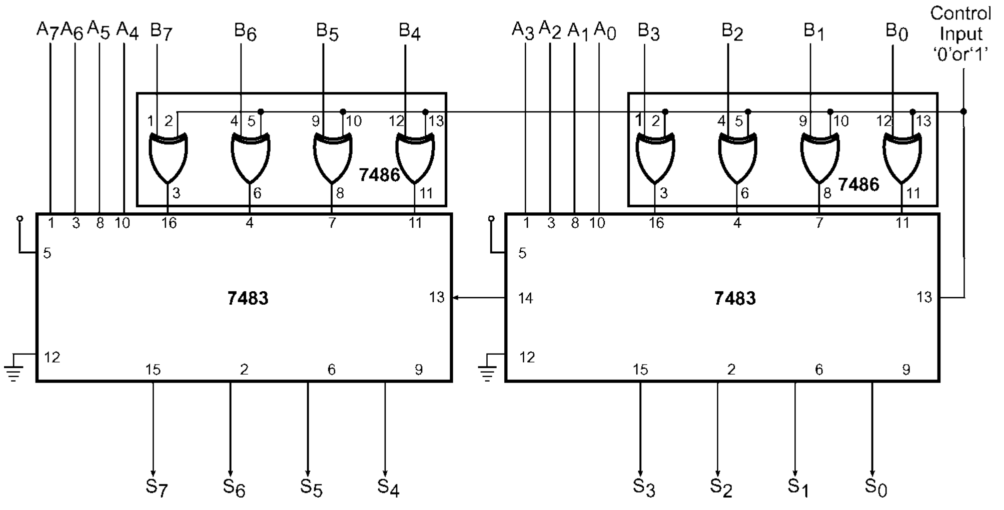


Figure 7.27 Solution to example 7.4.

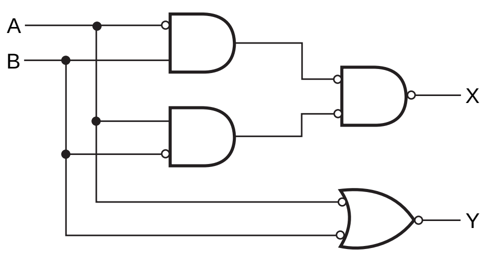


Figure 7.28 Example 7.5.

Example 7.6

Design a BCD adder circuit capable of adding BCD equivalents of two-digit decimal numbers. Indicate the IC type numbers used if the design has to be TTL logic family compatible.

Solution

The desired BCD adder is a cascaded arrangement of two stages of the type of BCD adder discussed in the previous pages. Figure 7.29 shows the logic diagram, and it follows the generalized cascaded arrangement discussed earlier and shown in Fig. 7.22 for a three-digit BCD adder. The BCD adder of Fig. 7.21 can be used to add four-bit BCD equivalents of two single-digit decimal numbers. A cascaded arrangement of two such stages, where the output C of Fig. 7.21 (CARRY-OUT) is fed to the CARRY-IN of the second stage, is shown in Fig. 7.29. In terms of IC type numbers, IC 7483 can be used for four-bit binary adders as shown in the diagram, IC 7408 can be used for implementing

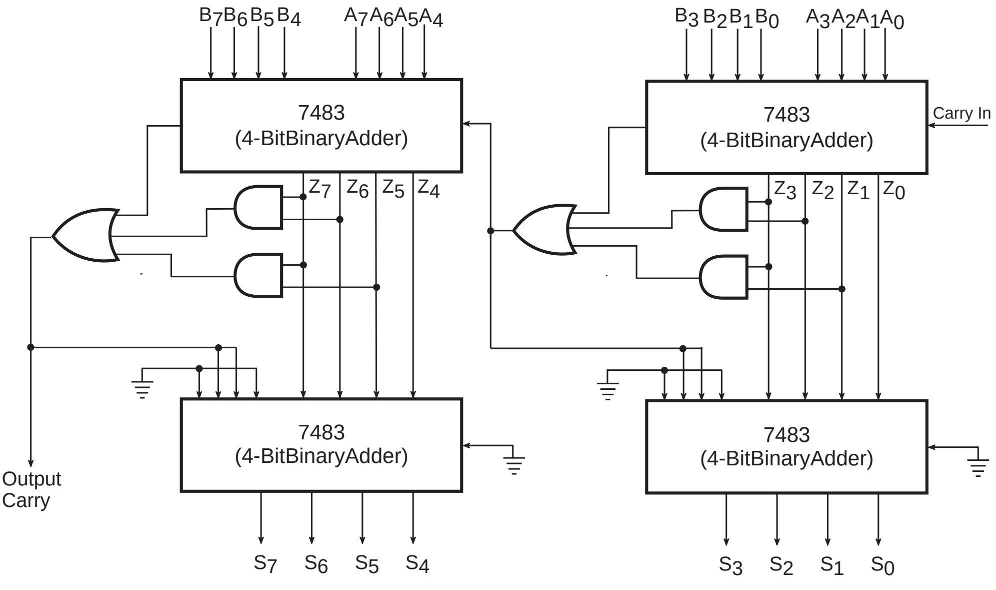


Figure 7.29 Example 7.6.

the required four two-input AND gates (IC 7408 is a quad two-input AND) and IC 7432 can be used to implement the required two three-input OR gates. IC 7432 is a quad two-input OR. Two two-input OR gates can be connected in cascade to get a three-input OR gate.

7.6 Carry Propagation–Look-Ahead Carry Generator

The four-bit binary adder described in the previous pages can be used to add two four-bit binary numbers. Multiple numbers of such adders are used to perform addition operations on larger-bit binary numbers. Each of the adders is composed of four full adders (FAs) connected in cascade. The block schematic arrangement of a four-bit adder is reproduced in Fig. 7.30(a) for reference and further discussion. This type of adder is also called a parallel binary adder because all the bits of the augend and addend are present and are fed to the full adder blocks simultaneously. Theoretically, the addition operation in various full adders takes place simultaneously. What is of importance and interest to users, more so when they are using a large number of such adders in their overall computation system, is whether the result of addition and carry-out are available to them at the same time. In other words, we need to see if this addition operation is truly parallel in nature. We will soon see that it is not. It is in fact limited by what is known as *carry propagation time*. Refer to Figs 7.30(a) and (b). Figure 7.30(b) shows the logic diagram of a full adder. Here, C_i and C_{i+1} are the input and output CARRY; P_i and G_i are two new binary variables called CARRY PROPAGATE and CARRY GENERATE and will be addressed a little later.

For $i=1$, the diagram in Fig. 7.30(b) is that of the LSB full adder of Fig. 7.30(a). We can see here that C_2, which is the output CARRY of FA (1) and the input CARRY for FA (2), will appear at the output after a minimum of two gate delays plus delay due to the half adder after application of A_i, B_i and C_i inputs.

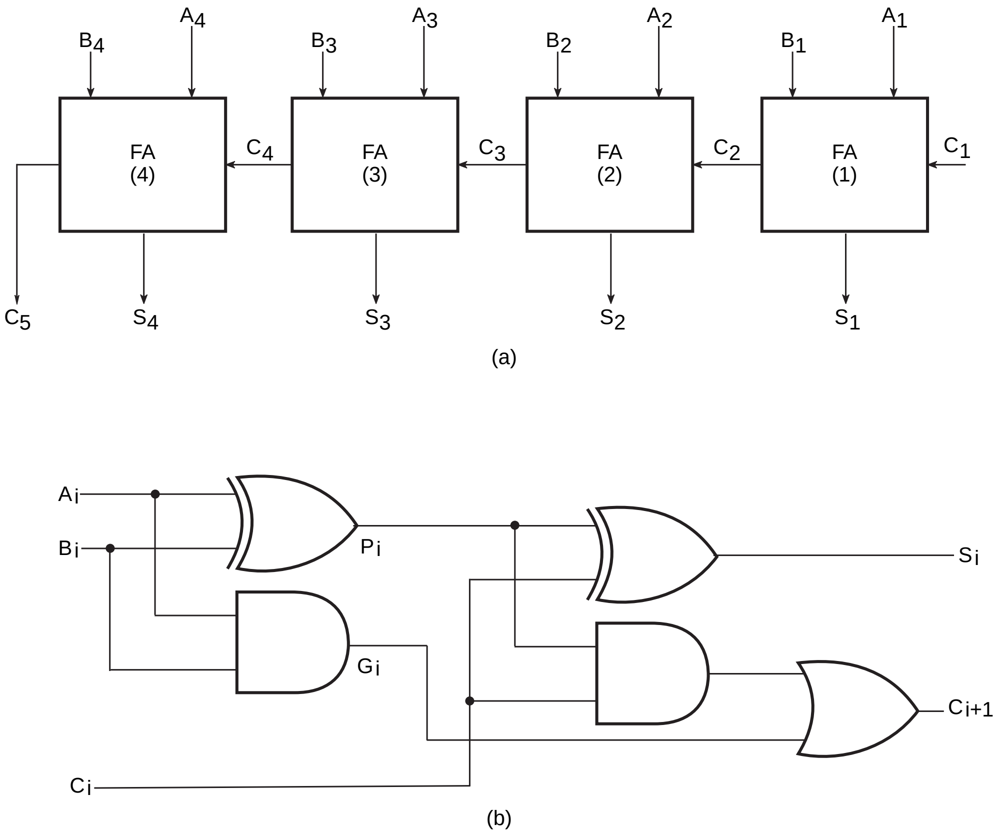


Figure 7.30 Four-bit binary adder.

The steady state of C_2 will be delayed by two gate delays after the appearance of C_1. Similarly, C_3 and C_4 steady state will be four and six gate delays respectively after C_1. And final carry C_5 will appear after eight gate delays.

Extending it a little further, let us assume that we are having a cascade arrangement of two four-bit adders to be able to handle eight-bit numbers. Now, C_5 will form the input CARRY to the second four-bit adder. The final output CARRY C_9 will now appear after 16 gate delays. This carry propagation delay limits the speed with which two numbers are added. The outputs of any such adder arrangement will be correct only if signals are given enough time to propagate through gates connected between input and output. Since subtraction is also an addition process and operations like multiplication and division are also processes involving successive addition and subtraction, the time taken by an addition process is very critical.

One of the possible methods for reducing carry propagation delay time is to use faster logic gates. But then there is a limit below which the gate delay cannot be reduced. There are other hardware-related techniques, the most widely used of which is the concept of look-ahead carry. This concept attempts to look ahead and generate the carry for a certain given addition operation that would otherwise have resulted from some previous operation. In order to explain the concept, let us define two new binary variables: P_i called CARRY PROPAGATE and G_i called CARRY GENERATE. Binary variable G_i is so called as it generates a carry whenever A_i and B_i are '1'. Binary variable P_i is called CARRY PROPAGATE as it is instrumental in propagation of C_i to C_{i+1}. CARRY, SUM, CARRY GENERATE and CARRY PROPAGATE parameters are given by the following expressions:

$$P_i = A_i \oplus B_i \tag{7.18}$$

$$G_i = A_i.B_i \tag{7.19}$$

$$S_i = P_i \oplus C_i \tag{7.20}$$

$$C_{i+1} = P_i.C_i + G_i \tag{7.21}$$

In the next step, we write Boolean expressions for the CARRY output of each full adder stage in the four-bit binary adder. We obtain the following expressions:

$$C_2 = G_1 + P_1.C_1 \tag{7.22}$$

$$C_3 = G_2 + P_2.C_2 = G_2 + P_2.(G_1 + P_1.C_1) = G_2 + P_2.G_1 + P_1.P_2.C_1 \tag{7.23}$$

$$C_4 = G_3 + P_3.C_3 = G_3 + P_3.(G_2 + P_2.G_1 + P_1.P_2.C_1)$$

$$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_1.P_2.P_3.C_1 \tag{7.24}$$

From the expressions for C_2, C_3 and C_4 it is clear that C_4 need not wait for C_3 and C_2 to propagate. Similarly, C_3 does not wait for C_2 to propagate. Hardware implementation of these expressions gives us a kind of look-ahead carry generator. A look-ahead carry generator that implements the above expressions using AND-OR logic is shown in Fig. 7.31.

Figure 7.32 shows the four-bit adder with the look-ahead carry concept incorporated. The block labelled *look-ahead carry generator* is similar to that shown in Fig. 7.31. The logic gates shown to the left of the block represent the input half-adder portion of various full adders constituting the four-bit adder. The EX-OR gates shown on the right are a portion of the output half-adders of various full adders.

All sum outputs in this case will be available at the output after a delay of two levels of logic gates. 74182 is a typical look-ahead carry generator IC of the TTL logic family. This IC can be used to generate relevant carry inputs for four four-bit binary adders connected in cascade to perform operation on two 16-bit numbers. Of course, the four-bit adders should be of the type so as to produce CARRY GENERATE and CARRY PROPAGATE outputs. Figure 7.33 shows the arrangement. In the figure shown, C_n is the CARRY input, G_0, G_1, G_2 and G_3 are CARRY GENERATE inputs for 74182 and P_0, P_1, P_2 and P_3 are CARRY PROPAGATE inputs for 74182. C_{n+x}, C_{n+y} and C_{n+z} are the CARRY outputs generated by 74182 for the four-bit adders. The G and P outputs of 74182 need to be cascaded. Figure 7.34 shows the arrangement needed for adding two 64-bit numbers.

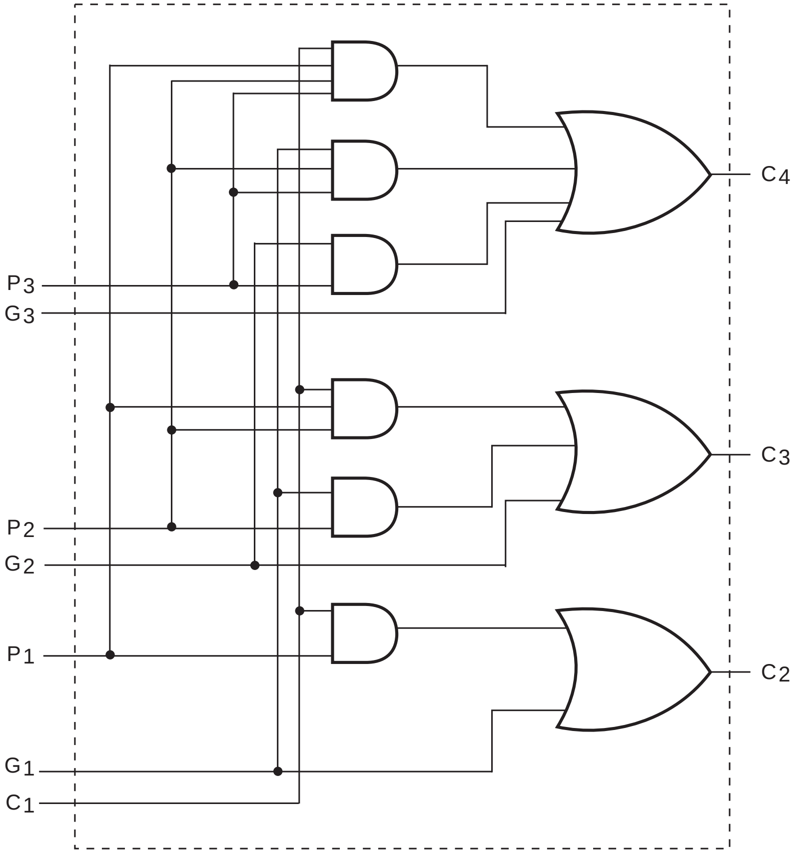


Figure 7.31 Look-ahead carry generator.

Example 7.7

If the CARRY GENERATE G_i and CARRY PROPAGATE P_i are redefined as $P_i = (A_i + B_i)$ and $G_i = A_iB_i$, show that the CARRY output C_{i+1} and the SUM output S_i of a full adder can be expressed by the following Boolean functions:

$$C_{i+1} = (\overline{\overline{C_i}.\overline{G_i} + \overline{P_i}}) = G_i + P_i.C_i \quad \text{and} \quad S_i = (P_i.\overline{G_i}) \oplus C_i$$

Solution

$$C_{i+1} = (\overline{\overline{C_i}.\overline{G_i} + \overline{P_i}}) = [\overline{\overline{C_i}.(\overline{A_{i.}B_i}) + (\overline{A_i + B_i})}]$$

$$= [\overline{\overline{C_i}.(\overline{A_i.B_i})}.(A_i + B_i)]$$

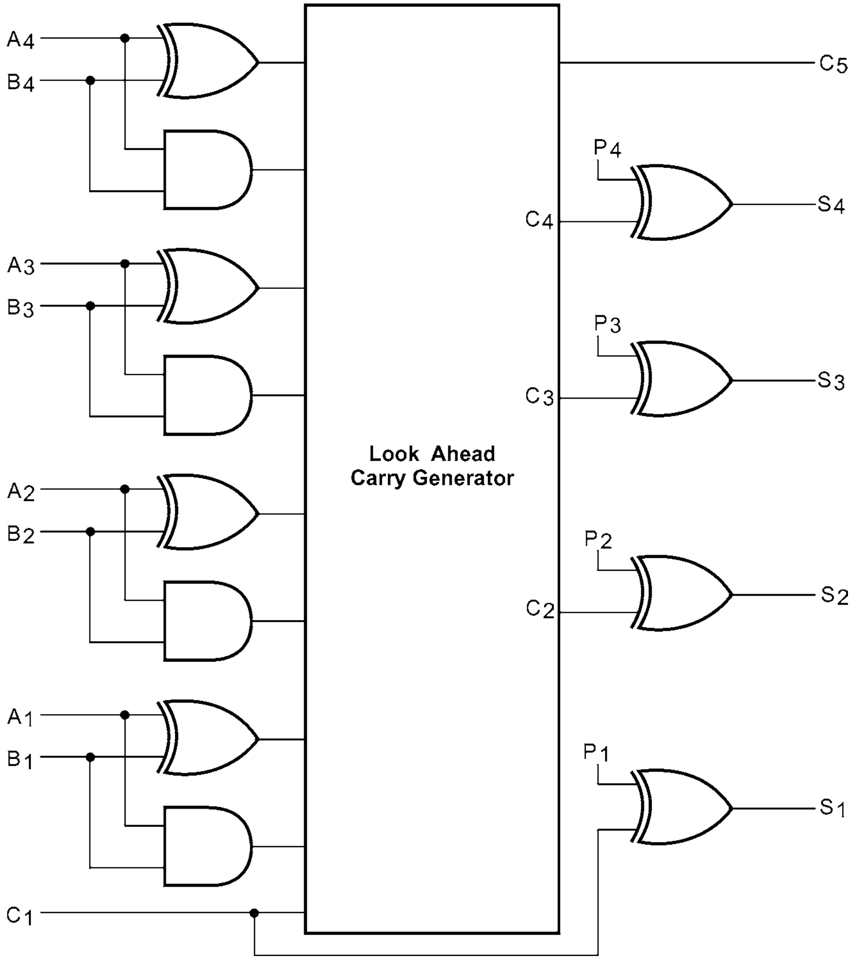


Figure 7.32 Four-bit full adder with a look-ahead carry generator.

$$= (C_i + A_i.B_i).(A_i + B_i) = C_i.(A_i + B_i) + A_i.B_i.(A_i + B_i)$$
$$= C_i.(A_i + B_i) + A_i.B_i = P_i.C_i + G_i$$

$$S_i = (A_i \oplus B_i) \oplus C_i = (\overline{A_i}.B_i + A_i.\overline{B_i}) \oplus C_i$$

Also

$$(P_i.\overline{G_i}) \oplus C_i = [(A_i + B_i).(\overline{A_i.B_i})] \oplus C_i$$
$$= [(A_i + B_i).(\overline{A_i} + \overline{B_i})] \oplus C_i = (\overline{A_i}.B_i + A_i.\overline{B_i}) \oplus C_i$$

Therefore, $S_i = (P_i.\overline{G_i}) \oplus C_i$.

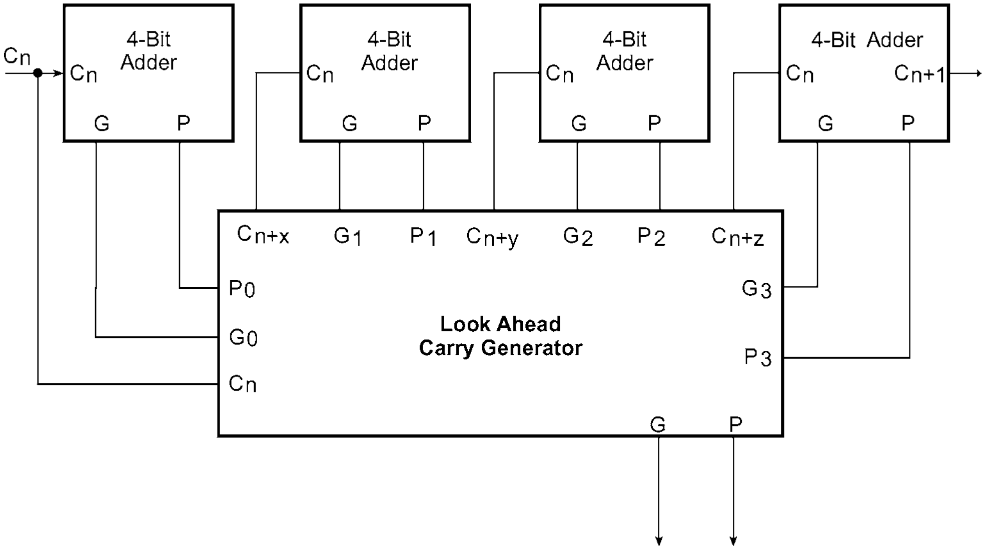


Figure 7.33 IC 74182 interfaced with four four-bit adders.

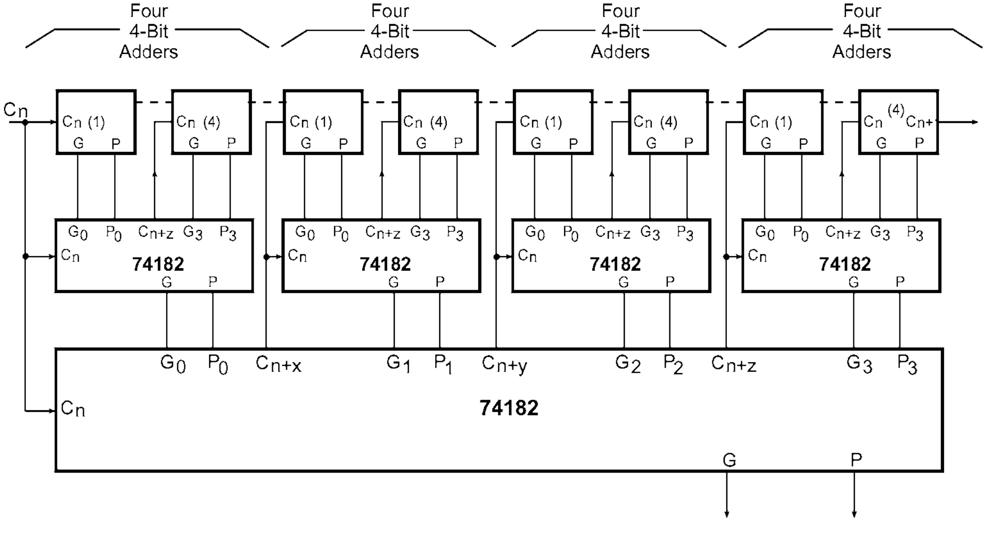


Figure 7.34 Look-ahead carry generation for adding 64-bit numbers.

7.7 Arithmetic Logic Unit (ALU)

The *arithmetic logic unit* (ALU) is a digital building block capable of performing both arithmetic as well as logic operations. Arithmetic logic units that can perform a variety of arithmetic operations such as addition, subtraction, etc., and logic functions such as ANDing, ORing, EX-ORing, etc., on two four-bit numbers are usually available in IC form. The function to be performed is selectable from *function select* pins. Some of the popular type numbers of ALU include 74181, 74381, 74382, 74582 (all from the TTL logic family) and 40181 (from the CMOS logic family). Functional details of these ICs are given in the latter part of the chapter under the heading of *Application-Relevant Information*. More than one such IC can always be connected in cascade to perform arithmetic and logic operations on larger bit numbers.

7.8 Multipliers

Multiplication of binary numbers is usually implemented in microprocessors and microcomputers by using *repeated addition and shift* operations. Since the binary adders are designed to add only two binary numbers at a time, instead of adding all the partial products at the end, they are added two at a time and their sum is accumulated in a register called the *accumulator register*. Also, when the multiplier bit is '0', that very partial product is ignored, as an all '0' line does not affect the final result. The basic hardware arrangement of such a binary multiplier would comprise shift registers for the multiplicand and multiplier bits, an accumulator register for storing partial products, a binary parallel adder and a clock pulse generator to time various operations.

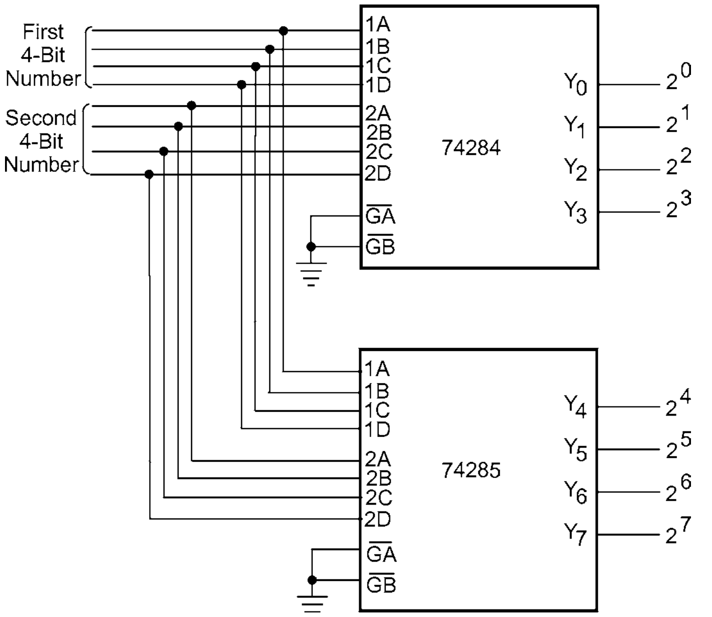


Figure 7.35 4 × 4 bit multiplier.

Binary multipliers are also available in IC form. Some of the popular type numbers in the TTL family include 74261 which is a 2 × 4 bit multiplier (a four-bit multiplicand designated as B_0,B_1,B_2, B_3 and B_4, and a two-bit multiplier designated as M_0, M_1 and M_2).

The MSBs B_4 and M_2 are used to represent signs. 74284 and 74285 are 4 × 4 bit multipliers. They can be used together to perform high-speed multiplication of two four-bit numbers. Figure 7.35 shows the arrangement. The result of multiplication is often required to be stored in a register. The size of this register (accumulator) depends upon the number of bits in the result, which at the most can be equal to the sum of the number of bits in the multiplier and multiplicand. Some multiplier ICs have an in-built register.

Many microprocessors do not have in their ALU the hardware that can perform multiplication or other complex arithmetic operations such as division, determining the square root, trigonometric functions, etc. These operations in these microprocessors are executed through software. For example, a multiplication operation may be accomplished by using a software program that does multiplication through repeated execution of addition and shift instructions. Other complex operations mentioned above can also be executed with similar programs. Although the use of software reduces the hardware needed in the microprocessor, the computation time in general is higher in the case of software-executed operations when compared with the use of hardware to perform those operations.

7.9 Magnitude Comparator

A *magnitude comparator* is a combinational circuit that compares two given numbers and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions $A = B$, $A > B$ and $A < B$, if A and B are the two numbers being compared. Depending upon the relative magnitude of the two numbers, the relevant output changes state. If the two numbers, let us say, are four-bit binary numbers and are designated as $(A_3\,A_2\,A_1\,A_0)$ and $(B_3\,B_2\,B_1\,B_0)$, the two numbers will be equal if all pairs of significant digits are equal, that is, $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$. In order to determine whether A is greater than or less than B, we inspect the relative magnitude of pairs of significant digits, starting from the most significant position. The comparison is done by successively comparing the next adjacent lower pair of digits if the digits of the pair under examination are equal. The comparison continues until a pair of unequal digits is reached. In the pair of unequal digits, if $A_i = 1$ and $B_i = 0$, then $A > B$, and if $A_i = 0$, $B_i = 1$ then $A < B$. If X, Y and Z are three variables respectively representing the $A = B$, $A > B$ and $A < B$ conditions, then the Boolean expression representing these conditions are given by the equations

$$X = x_3.x_2.x_1.x_0 \quad \text{where } x_i = A_i.B_i + \overline{A_i}.\overline{B_i} \tag{7.25}$$

$$Y = A_3.\overline{B_3} + x_3.A_2.\overline{B_2} + x_3.x_2.A_1.\overline{B_1} + x_3.x_2.x_1.A_0.\overline{B_0} \tag{7.26}$$

$$Z = \overline{A_3}.B_3 + x_3.\overline{A_2}.B_2 + x_3.x_2.\overline{A_1}.B_1 + x_3.x_2.x_1.\overline{A_0}.B_0 \tag{7.27}$$

Let us examine equation (7.25). x_3 will be '1' only when both A_3 and B_3 are equal. Similarly, conditions for x_2, x_1 and x_0 to be '1' respectively are equal A_2 and B_2, equal A_1 and B_1 and equal A_0 and B_0. ANDing of x_3, x_2, x_1 and x_0 ensures that X will be '1' when x_3, x_2, x_1 and x_0 are in the logic '1' state. Thus, $X = 1$ means that $A = B$. On similar lines, it can be visualized that equations (7.26) and

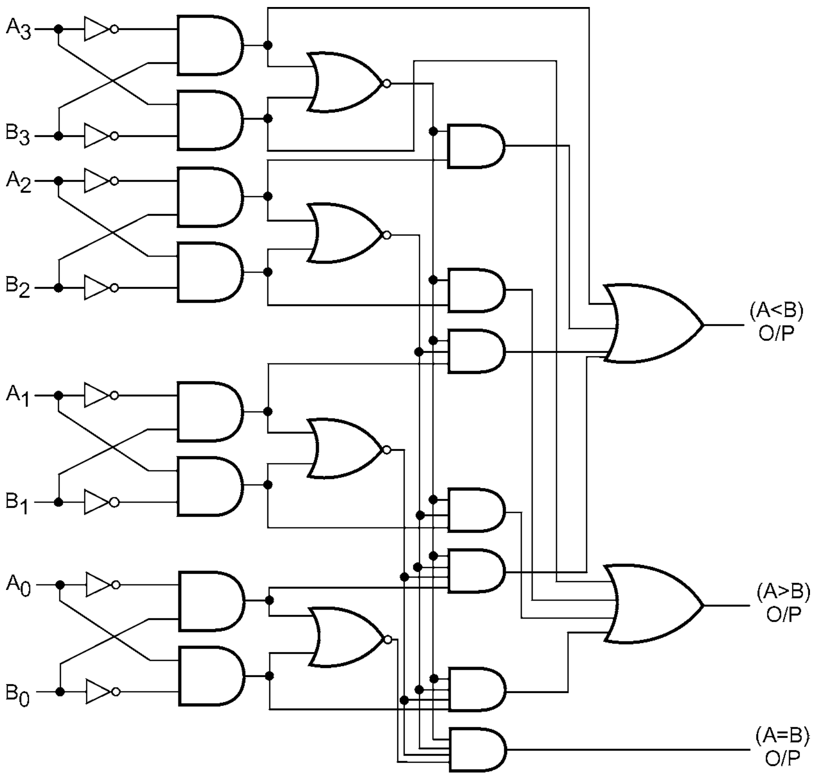


Figure 7.36 Four-bit magnitude comparator.

(7.27) respectively represent $A > B$ and $A < B$ conditions. Figure 7.36 shows the logic diagram of a four-bit magnitude comparator.

Magnitude comparators are available in IC form. For example, 7485 is a four-bit magnitude comparator of the TTL logic family. IC 4585 is a similar device in the CMOS family. 7485 and 4585 have the same pin connection diagram and functional table. The logic circuit inside these devices determines whether one four-bit number, binary or BCD, is *less than*, *equal to* or *greater than* a second four-bit number. It can perform comparison of straight binary and straight BCD (8-4-2-1) codes. These devices can be cascaded together to perform operations on larger bit numbers without the help of any external gates. This is facilitated by three additional inputs called cascading or expansion inputs available on the IC. These cascading inputs are also designated as $A = B$, $A > B$ and $A < B$ inputs. Cascading of individual magnitude comparators of the type 7485 or 4585 is discussed in the following paragraphs. IC 74AS885 is another common magnitude comparator. The device is an eight-bit magnitude comparator belonging to the advanced Schottky TTL family. It can perform high-speed arithmetic or logic comparisons on two eight-bit binary or 2's complement numbers and produces two fully decoded decisions at the output about one number being either greater than or less than the other. More than one of these devices can also be connected in a cascade arrangement to perform comparison of numbers of longer lengths.

7.9.1 Cascading Magnitude Comparators

As outlined earlier, magnitude comparators available in IC form are designed in such a way that they can be connected in a cascade arrangement to perform comparison operations on numbers of longer lengths. In cascade arrangement, the $A = B$, $A > B$ and $A < B$ outputs of a stage handling less significant bits are connected to corresponding inputs of the next adjacent stage handling more significant bits. Also, the stage handling least significant bits must have a HIGH level at the $A = B$ input. The other two cascading inputs ($A > B$ and $A < B$) may be connected to a LOW level. We will illustrate the concept by showing the arrangement of an eight-bit magnitude comparator using two four-bit magnitude comparators of the type 7485 or 4585. Figure 7.37 shows the cascaded arrangement of the two comparators. We can see the three comparison outputs of the comparator handling less significant four bits of the two numbers being connected to the corresponding cascading inputs of the comparator handling more significant four bits of the two numbers. Also, cascading inputs of the less significant comparator have been connected to a HIGH or LOW level as per the guidelines mentioned in the previous paragraph.

Operation of this circuit can be explained by considering the functional table of IC 7485 or IC 4585 as shown in Table 7.2. The two numbers being compared here are $(A_7 \ldots A_0)$ and $(B_7 \ldots B_0)$. The less significant comparator handles (A_3, A_2, A_1, A_0) and (B_3, B_2, B_1, B_0), and the more significant comparator handles (A_7, A_6, A_5, A_4) and (B_7, B_6, B_5, B_4). Let us take the example of the two numbers being such that $A_7 > B_7$. From the first-row entry of the function table it is clear that, irrespective of the status of other bits of the more significant comparator, and also regardless of the status of its cascading inputs, the final output produces a HIGH at the $A > B$ output and a LOW at the $A < B$ and $A = B$ outputs. Since the status of cascading inputs of the more significant comparator depends upon the status of comparison bits of the less significant comparator, the cascade arrangement produces the correct output for $A_7 > B_7$ regardless of the status of all other comparison bits. On similar lines, the circuit produces a valid output for any given status of comparison bits.

Example 7.8

Design a two-bit magnitude comparator. Also, write relevant Boolean expressions.

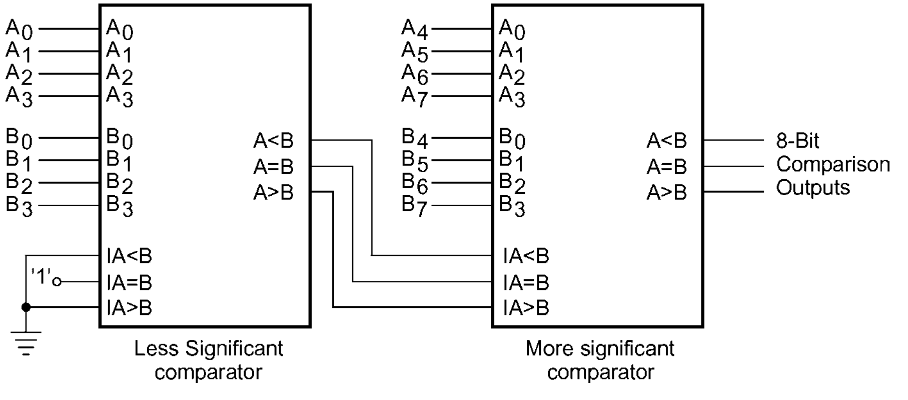


Figure 7.37 Cascading of individual magnitude comparators.

Table 7.2 Functional table of IC 7485 or IC 4585.

Comparison inputs				Cascading inputs			Outputs		
A_3,B_3	A_2,B_2	A_1,B_1	A_0,B_0	A>B	A<B	A=B	A>B	A<B	A=B
$A_3 > B_3$	X	X	X	X	X	X	HIGH	LOW	LOW
$A_3 < B_3$	X	X	X	X	X	X	LOW	HIGH	LOW
$A_3 = B_3$	$A_2 > B_2$	X	X	X	X	X	HIGH	LOW	LOW
$A_3 = B_3$	$A_2 < B_2$	X	X	X	X	X	LOW	HIGH	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 > B_1$	X	X	X	X	HIGH	LOW	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 < B_1$	X	X	X	X	LOW	HIGH	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 > B_0$	X	X	X	HIGH	LOW	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 < B_0$	X	X	X	LOW	HIGH	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	HIGH	LOW	LOW	HIGH	LOW	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	LOW	HIGH	LOW	LOW	HIGH	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	LOW	LOW	HIGH	LOW	LOW	HIGH
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	X	X	HIGH	LOW	LOW	HIGH
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	HIGH	HIGH	LOW	LOW	LOW	LOW
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	LOW	LOW	LOW	HIGH	HIGH	LOW

Solution

Let $A\,(A_1A_0)$ and $B\,(B_1B_0)$ be the two numbers. If X, Y and Z represent the conditions $A = B$, $A > B$ and $A < B$ respectively (that is, $X = 1$, $Y = 0$ and $Z = 0$ for A =B; X= 0, $Y = 1$ and $Z = 0$ for $A > B$; and $X = 0$, $Y = 0$ and $Z = 1$ for $A < B$), then expressions for X, Y and Z can be written as follows:

$$X = x_1.x_0 \text{ where } x_1 = A_1.B_1 + \overline{A_1}.\overline{B_1} \quad \text{and} \quad x_0 = A_0.B_0 + \overline{A_0}.\overline{B_0}$$
$$Y = A_1.\overline{B_1} + x_1.A_0.\overline{B_0}$$
$$Z = \overline{A_1}.B_1 + x_1.\overline{A_0}.B_0$$

Figure 7.38 shows the logic diagram of the two-bit comparator.

Example 7.9

Hardware-implement a three-bit magnitude comparator having one output that goes HIGH when the two three-bit numbers are equal. Use only NAND gates.

Solution

The equivalence condition of the two three-bit numbers is given by the equation $X = x_2.x_1.x_0$, where $x_2 = A_2.B_2 + \overline{A_2}.\overline{B_2}$, $x_1 = A_1.B_1 + \overline{A_1}.\overline{B_1}$ and $x_0 = A_0.B_0 + \overline{A_0}.\overline{B_0}$.

Figure 7.39 shows the logic diagram. x_2, x_1 and x_0 are respectively given by EX-NOR operation of (A_2, B_2), (A_1, B_1) and (A_0, B_0). These are then ANDed to get X.

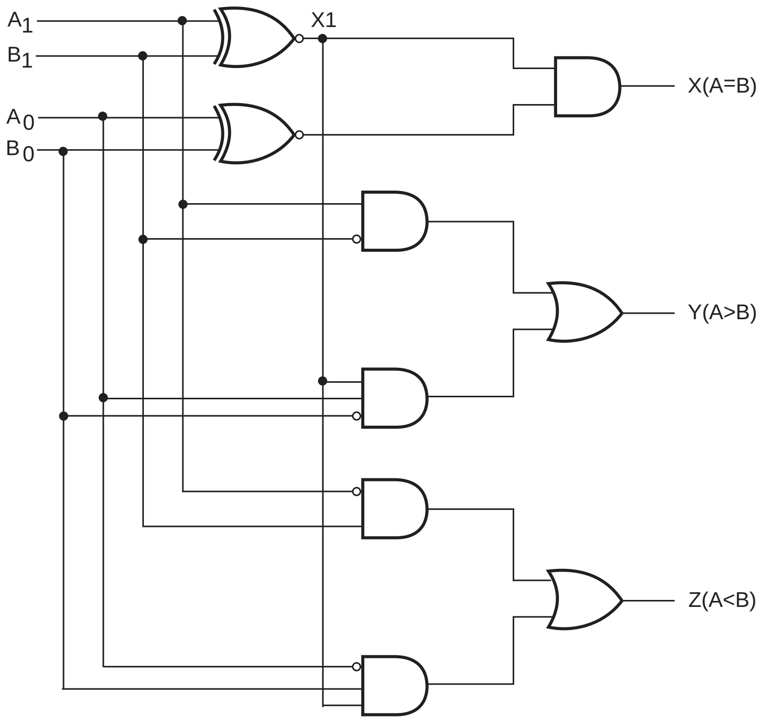


Figure 7.38 Solution to example 7.8.

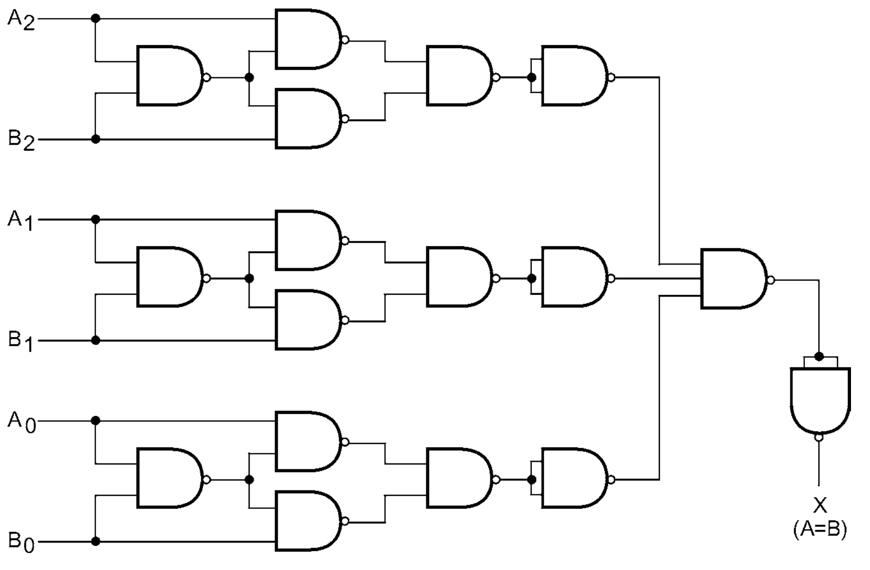


Figure 7.39 Solution to example 7.9.

Table 7.3 Commonly used IC type numbers used for arithmetic operations.

IC type number	Function	Logic family
7483	Four-bit full adder	TTL
7485	Four-bit magnitude comparator	TTL
74181	Four-Bit ALU and function generator	TTL
74182	Look-ahead carry generator	TTL
74183	Dual carry save full adder	TTL
74283	Four-bit full binary adder	TTL
74885	Eight-bit magnitude comparator	TTL
4008	Four-bit binary full adder	CMOS
4527	BCD rate multiplier	CMOS
4585	Four-bit magnitude comparator	CMOS
40181	Four-bit arithmetic logic unit	CMOS
40182	Look-ahead carry generator	CMOS
10179	Look-ahead carry block	ECL
10180	Dual high-speed two-bit adder/subtractor	ECL
10181	Four-bit arithmetic logic unit/function generator	ECL
10182	Four-bit arithmetic logic unit/function generator	ECL
10183	4×2 multiplier	ECL

7.10 Application-Relevant Information

Table 7.3 lists commonly used IC type numbers used for arithmetic operations. Application-relevant information such as pin connection diagrams, truth tables, etc., in respect of the more popular of these type numbers is given on the companion website.

Review Questions

1. How do you characterize or define a combinational circuit? How does it differ from a sequential circuit? Give two examples each of combinational and sequential logic devices.
2. Beginning with the statement of the problem, outline different steps involved in the design of a suitable combinational logic circuit to implement the hardware required to solve the given problem.
3. Write down Boolean expressions representing the SUM and CARRY outputs in terms of three input binary variables to be added. Design a suitable combinational circuit to hardware-implement the design using NAND gates only.
4. Draw the truth table of a full subtractor circuit. Write a minterm Boolean expression for DIFFERENCE and BORROW outputs in terms of minuend variable, subtrahend variable and BORROW-IN. Minimize the expressions and implement them in hardware.
5. Draw the logic diagram of a three-digit BCD adder and briefly describe its functional principle.
6. Briefly describe the concept of look-ahead carry generation with respect to its use in adder circuits. What is its significance while implementing hardware for addition of binary numbers of longer lengths?
7. With the help of a block schematic of the logic circuit, briefly describe how individual four-bit magnitude comparators can be used in a cascade arrangement to perform magnitude comparison of binary numbers of longer lengths.

Problems

1. A, B, B_{in}, D and B_{out} are respectively the minuend, the subtrahend, the BORROW-IN, the DIFFERENCE output and the BORROW-OUT in the case of a full subtractor. Determine the bit status of D and B_{out} for the following values of A, B and B_{in}:

   (a) $A = 0$, $B = 1$, $B_{in} = 1$
   (b) $A = 1$, $B = 1$, $B_{in} = 0$
   (c) $A = 1$, $B = 1$, $B_{in} = 1$
   (d) $A = 0$, $B = 0$, $B_{in} = 1$

   (a) $D = 0$, $B_{out} = 1$; (b) $D = 0$, $B_{out} = 0$; (c) $D = 1$, $B_{out} = 1$; (d) $D = 1$, $B_{out} = 1$

2. Determine the number of half and full adder circuit blocks required to construct a 64-bit binary parallel adder. Also, determine the number and type of additional logic gates needed to transform this 64-bit adder into a 64-bit adder–subtractor.

   For a 64-bit adder: HA=1, FA=63

   For a 64-bit adder–subtractor: HA = 1, FA = 63, EX-OR gates = 64

3. If the minuend, subtrahend and BORROW-IN bits are respectively applied to the Augend, Addend and the CARRY-IN inputs of a full adder, prove that the SUM output of the full adder will produce the correct DIFFERENCE output.
4. Prove that the logic diagram of Fig. 7.40 performs the function of a half-subtractor provided that Y represents the DIFFERENCE output and X represents the BORROW output.
5. Determine the number of 7483s (four-bit binary adders) and 7486s (quad two-input EX-OR gates) required to design a 16-bit adder–subtractor circuit.

   Number of 7483 = 4; number of 7486 = 4

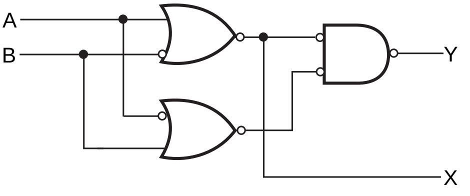


Figure 7.40 Problem 4.

6. The objective is to design a BCD adder circuit using four-bit binary adders and additional combinational logic. If the decimal numbers to be added can be anywhere in the range from 0 to 9999, determine the number of four-bit binary adder circuit blocks of type IC 7483 required to do the job.

   Number of four-bit adders = 8

Further Reading

1. Koren, I. (2001) *Computer Arithmetic Algorithms*, A. K. Peters Ltd, Natick, MA, USA.
2. Ercegovac, M. D. and Lang, T. (2003) *Digital Arithmetic*, Morgan Kaufmann Publishers, CA, USA.
3. Rafiquzzaman, M. (2005) *Fundamentals of Digital Logic and Microcomputer Design*, Wiley-Interscience, New York, USA.
4. Morris Mano, M. and Kime, C. R. (2003) *Logic and Computer Design Fundamentals*, Prentice-Hall, USA.
5. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
6. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
7. Malvino, A. P. and Leach, D. P. (1994) *Digital Principles and Applications*, McGraw-Hill Book Company, USA.

8

Multiplexers and Demultiplexers

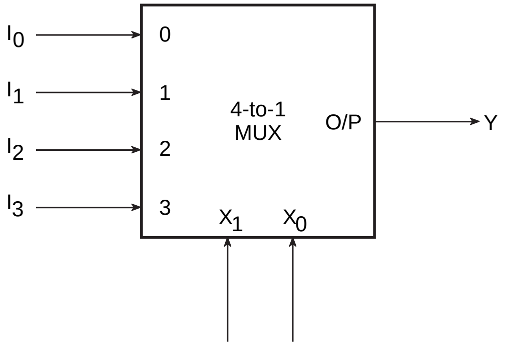
In the previous chapter, we described at length those combinational logic circuits that can be used to perform arithmetic and related operations. This chapter takes a comprehensive look at yet another class of building blocks used to design more complex combinational circuits, and covers building blocks such as multiplexers and demultiplexers and other derived devices such as encoders and decoders. Particular emphasis is given to the operational basics and use of these devices to design more complex combinational circuits. Application-relevant information in terms of the list of commonly used integrated circuits available in this category, along with their functional description is given towards the end of the chapter. The text has been adequately illustrated with the help of a large number of solved examples.

8.1 Multiplexer

A *multiplexer* or *MUX*, also called a *data selector*, is a combinational circuit with more than one input line, one output line and more than one selection line. There are some multiplexer ICs that provide complementary outputs. Also, multiplexers in IC form almost invariably have an ENABLE or STROBE input, which needs to be active for the multiplexer to be able to perform its intended function. A multiplexer selects binary information present on any one of the input lines, depending upon the logic status of the selection inputs, and routes it to the output line. If there are n selection lines, then the number of maximum possible input lines is 2^n and the multiplexer is referred to as a 2^n-to-1 multiplexer or $2^n \times 1$ multiplexer. Figures 8.1(a) and (b) respectively show the circuit representation and truth table of a basic 4-to-1 multiplexer.

To familiarize readers with the practical multiplexer devices available in IC form, Figs 8.2 and 8.3 respectively show the circuit representation and function table of 8-to-1 and 16-to-1 multiplexers. The 8-to-1 multiplexer of Fig. 8.2 is IC type number 74151 of the TTL family. It has an active LOW ENABLE input and provides complementary outputs. Figure 8.3 refers to IC type number 74150 of the TTL family. It is a 16-to-1 multiplexer with active LOW ENABLE input and active LOW output.

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

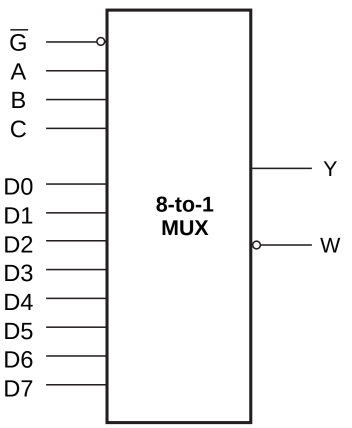


(a)

X_1	X_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b)

Figure 8.1 (a) 4-to-1 multiplexer circuit representation and (b) 4-to-1 multiplexer truth table.



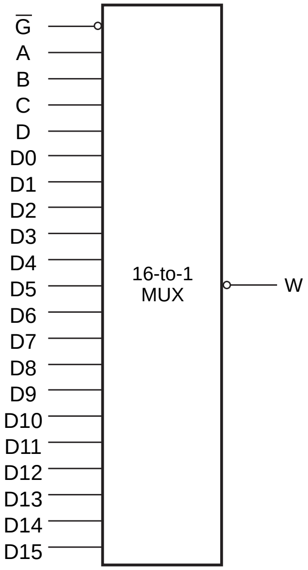
(a)

Inputs				Output	
Select			Enable		
C	B	A	$\overline{G}$	Y	W
X	X	X	H	L	H
L	L	L	L	D0	$\overline{D0}$
L	L	H	L	D1	$\overline{D1}$
L	H	L	L	D2	$\overline{D2}$
L	H	H	L	D3	$\overline{D3}$
H	L	L	L	D4	$\overline{D4}$
H	L	H	L	D5	$\overline{D5}$
H	H	L	L	D6	$\overline{D6}$
H	H	H	L	D7	$\overline{D7}$

$\overline{G}$: ENABLE input
A, B, C : Select inputs
D0-D7 : Data inputs
Y,W : outputs

(b)

Figure 8.2 (a) 8-to-1 multiplexer circuit representation and (b) 8-to-1 multiplexer truth table.



(a)

Inputs					Output W
Select				Enable	
D	C	B	A	$\overline{G}$	
X	X	X	X	H	H
L	L	L	L	L	$\overline{D0}$
L	L	L	H	L	$\overline{D1}$
L	L	H	L	L	$\overline{D2}$
L	L	H	H	L	$\overline{D3}$
L	H	L	L	L	$\overline{D4}$
L	H	L	H	L	$\overline{D5}$
L	H	H	L	L	$\overline{D6}$
L	H	H	H	L	$\overline{D7}$
H	L	L	L	L	$\overline{D8}$
H	L	L	H	L	$\overline{D9}$
H	L	H	L	L	$\overline{D10}$
H	L	H	H	L	$\overline{D11}$
H	H	L	L	L	$\overline{D12}$
H	H	L	H	L	$\overline{D13}$
H	H	H	L	L	$\overline{D14}$
H	H	H	H	L	$\overline{D15}$

(b)

Figure 8.3 (a) 16-to-1 multiplexer circuit representation and (b) 16-to-1 multiplexer truth table.

8.1.1 Inside the Multiplexer

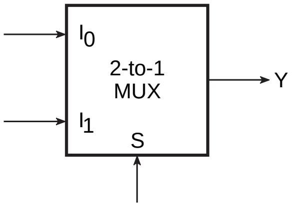
We will briefly describe the type of combinational logic circuit found inside a multiplexer by considering the 2-to-1 multiplexer in Fig. 8.4(a), the functional table of which is shown in Fig. 8.4(b). Figure 8.4(c) shows the possible logic diagram of this multiplexer. The circuit functions as follows:

- For $S = 0$, the Boolean expression for the output becomes $Y = I_0$.
- For $S = 1$, the Boolean expression for the output becomes $Y = I_1$.

Thus, inputs I_0 and I_1 are respectively switched to the output for $S = 0$ and $S = 1$. Extending the concept further, Fig. 8.5 shows the logic diagram of a 4-to-1 multiplexer. The input combinations 00, 01, 10 and 11 on the select lines respectively switch I_0, I_1, I_2 and I_3 to the output.The operation of the circuit is governed by the Boolean function (8.1). Similarly, an 8-to-1 multiplexer can be represented by the Boolean function (8.2):

$$Y = I_0.\overline{S_1}.\overline{S_0} + I_1.\overline{S_1}.S_0 + I_2.S_1.\overline{S_0} + I_3.S_1.S_0 \tag{8.1}$$

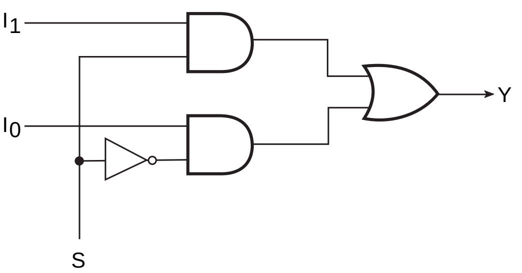
$$\begin{aligned} Y = {} & I_0.\overline{S_2}.\overline{S_1}.\overline{S_0} + I_1.\overline{S_2}.\overline{S_1}.S_0 + I_2.\overline{S_2}.S_1.\overline{S_0} + I_3.\overline{S_2}.S_1.S_0 + I_4.S_2.\overline{S_1}.\overline{S_0} \\ & + I_5.S_2.\overline{S_1}.S_0 + I_6.S_2.S_1.\overline{S_0} + I_7.S_2.S_1.S_0 \end{aligned} \tag{8.2}$$



(a)

S	Y
0	I_0
1	I_1

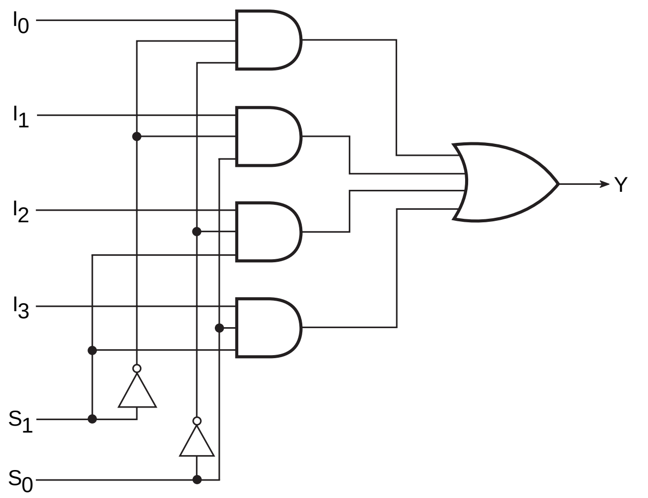
(b)



(c)

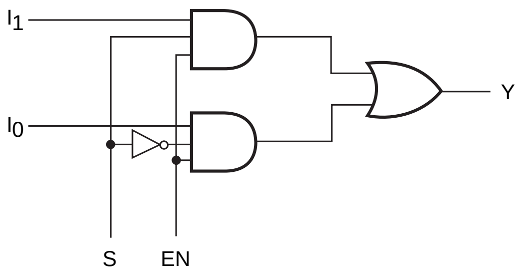
Figure 8.4 (a) 2-to-1 multiplexer circuit representation, (b) 2-to-1 multiplexer truth table and (c) 2-to-1 multiplexer logic diagram.

As outlined earlier, multiplexers usually have an ENABLE input that can be used to control the multiplexing function. When this input is enabled, that is, when it is in logic '1' or logic '0' state, depending upon whether the ENABLE input is active HIGH or active LOW respectively, the output is enabled. The multiplexer functions normally. When the ENABLE input is inactive, the output is disabled and permanently goes to either logic '0' or logic '1' state, depending upon whether the output is uncomplemented or complemented. Figure 8.6 shows how the 2-to-1 multiplexer of Fig. 8.4 can be modified to include an ENABLE input. The functional table of this modified multiplexer is also shown in Fig. 8.6. The ENABLE input here is active when HIGH. Some IC packages have more than one multiplexer. In that case, the ENABLE input and selection inputs are common to all multiplexers within the same IC package. Figure 8.7 shows a 4-to-1 multiplexer with an active LOW ENABLE input.



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure 8.5 Logic diagram of a 4-to-1 multiplexer.



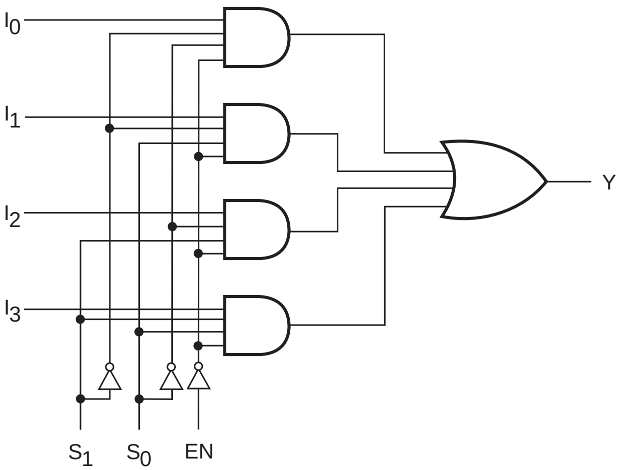
S	EN	Y
X	0	0
0	1	I_0
1	1	I_1

Figure 8.6 2-to-1 multiplexer with an ENABLE input.

8.1.2 Implementing Boolean Functions with Multiplexers

One of the most common applications of a multiplexer is its use for implementation of combinational logic Boolean functions. The simplest technique for doing so is to employ a 2^n-to-1 MUX to implement an n-variable Boolean function. The input lines corresponding to each of the minterms present in the Boolean function are made equal to logic '1' state. The remaining minterms that are absent in the Boolean function are disabled by making their corresponding input lines equal to logic '0'. As an example, Fig. 8.8(a) shows the use of an 8-to-1 MUX for implementing the Boolean function given by the equation

$$f(A, B, C) = \sum 2, 4, 7 \tag{8.3}$$



S_1	S_0	EN	Y
X	X	1	0
0	0	0	I_0
0	1	0	I_1
1	0	0	I_2
1	1	0	I_3

Figure 8.7 4-to-1 multiplexer with an ENABLE input.

In terms of variables A, B and C, equation (8.3) can be written as follows:

$$f(A, B, C) = \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.C \tag{8.4}$$

As shown in Fig. 8.8, the input lines corresponding to the three minterms present in the given Boolean function are tied to logic '1'. The remaining five possible minterms absent in the Boolean function are tied to logic '0'.

However, there is a better technique available for doing the same. In this, a 2^n-to-1 MUX can be used to implement a Boolean function with $n + 1$ variables. The procedure is as follows. Out of n + 1 variables, n are connected to the n selection lines of the 2^n-to-1 multiplexer. The left-over variable is used with the input lines. Various input lines are tied to one of the following: '0', '1', the left-over variable and the complement of the left-over variable. Which line is given what logic status can be easily determined with the help of a simple procedure. The complete procedure is illustrated for the Boolean function given by equation (8.3).

It is a three-variable Boolean function. Conventionally, we will need to use an 8-to-1 multiplexer to implement this function. We will now see how this can be implemented with a 4-to-1 multiplexer. The chosen multiplexer has two selection lines. The first step here is to determine the truth table of the given Boolean function, which is shown in Table 8.1.

In the next step, two of the three variables are connected to the two selection lines, with the higher-order variable connected to the higher-order selection line. For instance, in the present case, variables B and C are the chosen variables for the selection lines and are respectively connected to selection lines S_1 and S_0. In the third step, a table of the type shown in Table 8.2 is constructed. Under the inputs to the multiplexer, minterms are listed in two rows, as shown. The first row lists those terms where remaining variable A is complemented, and second row lists those terms where A is uncomplemented. This is easily done with the help of the truth table.

The required minterms are identified or marked in some manner in this table. In the given table, these entries have been highlighted. Each column is inspected individually. If neither minterm of a certain column is highlighted, a '0' is written below that. If both are highlighted, a '1' is

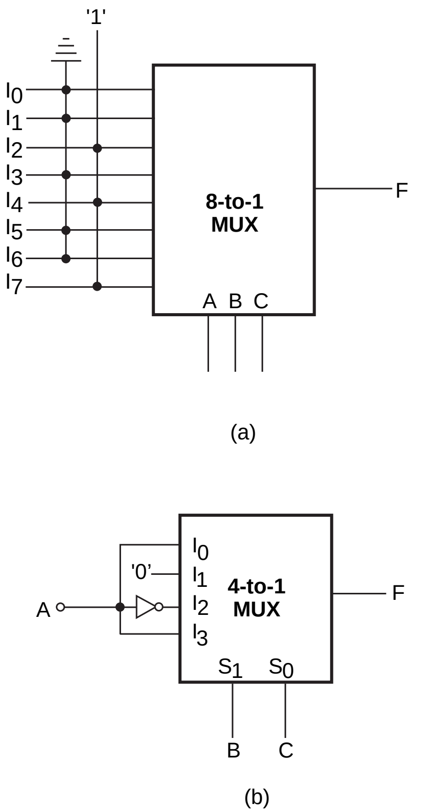


Figure 8.8 Hardware implementation of the Boolean function given by equation (8.3).

Table 8.1 Truth table.

Minterm	*A*	*B*	*C*	*f(A,B,C)*
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

written. If only one is highlighted, the corresponding variable (complemented or uncomplemented) is written. The input lines are then given appropriate logic status. In the present case, I_0, I_1, I_2 and I_3 would be connected to A, 0, $\overline{A}$ and A respectively. Figure 8.8(b) shows the logic implementation.

Table 8.2 Implementation table for multiplexers.

	I_0	I_1	I_2	I_3
$\overline{A}$	0	1	**2**	3
A	**4**	5	6	**7**
	A	0	$\overline{A}$	A

Table 8.3 Implementation table for multiplexers.

	I_0	I_1	I_2	I_3
$\overline{C}$	0	**2**	**4**	6
C	1	3	5	**7**
	0	$\overline{C}$	$\overline{C}$	C

It is not necessary to choose only the leftmost variable in the sequence to be used as input to the multiplexer. Any of the variables can be used provided the implementation table is constructed accordingly. In the problem illustrated above, A was chosen as the variable for the input lines, and accordingly the first row of the implementation table contained those entries where 'A' was complemented and the second row contained those entries where A was uncomplemented. If we consider C as the left-out variable, the implementation table will be as shown in Table 8.3.

Figure 8.9 shows the hardware implementation. For the case of B being the left-out variable, the implementation table is shown in Table 8.4 and the hardware implementation is shown in Fig. 8.10.

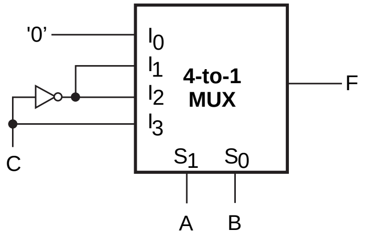


Figure 8.9 Hardware implementation using a 4-to-1 multiplexer.

Table 8.4 Implementation table for multiplexers.

	I_0	I_1	I_2	I_3
$\overline{B}$	0	1	**4**	5
B	**2**	3	6	**7**
	B	0	$\overline{B}$	B

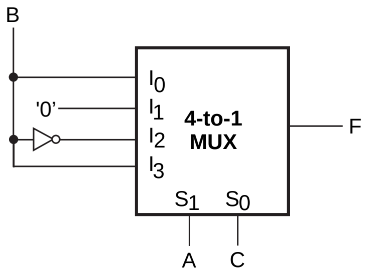


Figure 8.10 Hardware implementation using a 4-to-1 multiplexer.

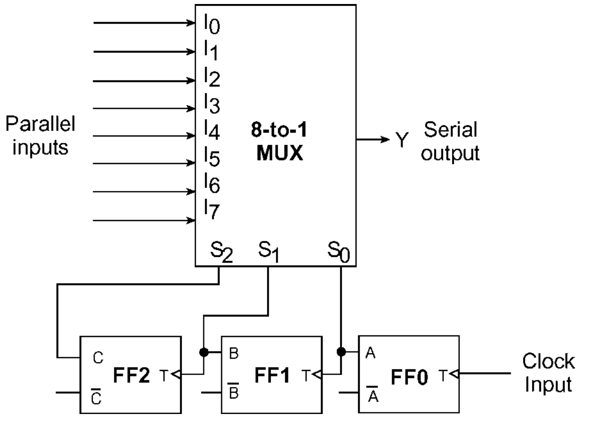


Figure 8.11 Multiplexer for parallel-to-serial conversion.

8.1.3 Multiplexers for Parallel-to-Serial Data Conversion

Although data are processed in parallel in many digital systems to achieve faster processing speeds, when it comes to transmitting these data relatively large distances, this is done serially. The parallel arrangement in this case is highly undesirable as it would require a large number of transmission lines. Multiplexers can possibly be used for parallel-to-serial conversion. Figure 8.11 shows one such arrangement where an 8-to-1 multiplexer is used to convert eight-bit parallel binary data to serial form. A three-bit counter controls the selection inputs. As the counter goes through 000 to 111, the multiplexer output goes through I_0 to I_7. The conversion process takes a total of eight clock cycles. In the figure shown, the three-bit counter has been constructed with the help of three toggle flip-flops. A variety of counter circuits of various types and complexities are, however, available in IC form. Flip-flops and counters are discussed in detail in Chapters 10 and 11 respectively.

Example 8.1

Implement the product-of-sums Boolean function expressed by $\Pi 1,2,5$ by a suitable multiplexer.

Solution

- Let the Boolean function be $f(A, B, C) = \prod 1, 2, 5$.
- The equivalent sum-of-products expression can be written as $f(A, B, C) = \sum 0, 3, 4, 6, 7$.

The truth table for the given Boolean function is given in Table 8.5. The given function can be implemented with a 4-to-1 multiplexer with two selection lines. Variables A and B are chosen for the selection lines. The implementation table as drawn with the help of the truth table is given in Table 8.6. Figure 8.12 shows the hardware implementation.

Table 8.5 Truth table.

C	B	A	$f(A,B,C)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 8.6 Implementation table.

	I_0	I_1	I_2	I_3
$\overline{C}$	**0**	1	2	**3**
C	**4**	5	**6**	**7**
	1	0	C	1

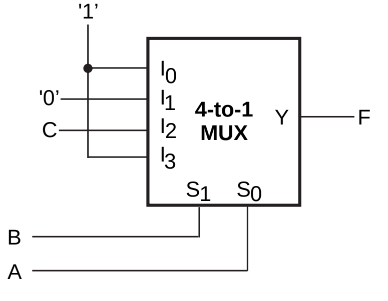


Figure 8.12 Example 8.1.

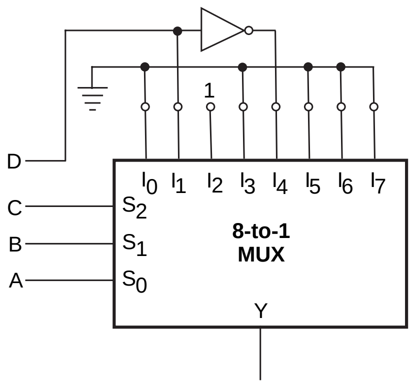


Figure 8.13 Example 8.2.

Example 8.2

Figure 8.13 shows the use of an 8-to-1 multiplexer to implement a certain four-variable Boolean function. From the given logic circuit arrangement, derive the Boolean expression implemented by the given circuit.

Solution

This problem can be solved by simply working backwards in the procedure outlined earlier for designing the multiplexer-based logic circuit for a given Boolean function. Here, the hardware implementation is known and the objective is to determine the corresponding Boolean expression.

From the given logic circuit, we can draw the implementation table as given in Table 8.7. The entries in the first row (0, 1, 2, 3, 4, 5, 6, 7) and the second row (8, 9, 10, 11, 12, 13, 14, 15) are so because the selection variable chosen for application to the inputs is the MSB variable D. Entries in the first row include all those minterms that contain $\overline{D}$, and entries in the second row include all those minterms that contain D. After writing the entries in the first two rows, the entries in the third row can be filled in by examining the logic status of different input lines in the given logic circuit diagram. Having completed the third row, relevant entries in the first and second rows are highlighted. The Boolean expression can now be written as follows:

$$\begin{aligned} Y = \sum 2, 4, 9, 10 &= \overline{D}.\overline{C}.B.\overline{A} + \overline{D}.C.\overline{B}.\overline{A} + D.\overline{C}.\overline{B}.A + D.\overline{C}.B.\overline{A} \\ &= \overline{C}.B.\overline{A}.(\overline{D} + D) + \overline{D}.C.\overline{B}.\overline{A} + D.\overline{C}.\overline{B}.A \\ &= \overline{C}.B.\overline{A} + \overline{D}.C.\overline{B}.\overline{A} + D.\overline{C}.\overline{B}.A \end{aligned}$$

Table 8.7 Implementation table.

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

8.1.4 Cascading Multiplexer Circuits

There can possibly be a situation where the desired number of input channels is not available in IC multiplexers. A multiple number of devices of a given size can be used to construct multiplexers that can handle a larger number of input channels. For instance, 8-to-1 multiplexers can be used to construct 16-to-1 or 32-to-1 or even larger multiplexer circuits. The basic steps to be followed to carry out the design are as follows:

1. If 2^n is the number of input lines in the available multiplexer and 2^N is the number of input lines in the desired multiplexer, then the number of individual multiplexers required to construct the desired multiplexer circuit would be 2^{N-n}.
2. From the knowledge of the number of selection inputs of the available multiplexer and that of the desired multiplexer, connect the less significant bits of the selection inputs of the desired multiplexer to the selection inputs of the available multiplexer.
3. The left-over bits of the selection inputs of the desired multiplexer circuit are used to enable or disable the individual multiplexers so that their outputs when ORed produce the final output. The procedure is illustrated in solved example 8.3.

Example 8.3

Design a 16-to-1 multiplexer using two 8-to-1 multiplexers having an active LOW ENABLE input.

Solution

A 16-to-1 multiplexer can be constructed from two 8-to-1 multiplexers having an ENABLE input. The ENABLE input is taken as the fourth selection variable occupying the MSB position. Figure 8.14 shows the complete logic circuit diagram. IC 74151 can be used to implement an 8-to-1 multiplexer.

The circuit functions as follows. When S_3 is in logic '0' state, the upper multiplexer is enabled and the lower multiplexer is disabled. If we recall the truth table of a four-variable Boolean function, S_3 would be '0' for the first eight entries and '1' for the remaining eight entries. Therefore, when $S_3 = 0$ the final output will be any of the inputs from D_0 to D_7, depending upon the logic status of S_2, S_1 and S_0. Similarly, when $S_3 = 1$ the final output will be any of the inputs from D_8 to D_{15}, again depending upon the logic status of S_2, S_1 and S_0. The circuit therefore implements the truth table of a 16-to-1 multiplexer.

8.2 Encoders

An *encoder* is a multiplexer without its single output line. It is a combinational logic function that has 2^n (or fewer) input lines and n output lines, which correspond to n selection lines in a multiplexer. The n output lines generate the binary code for the possible 2^n input lines. Let us take the case of an octal-to-binary encoder. Such an encoder would have eight input lines, each representing an octal digit, and three output lines representing the three-bit binary equivalent. The truth table of such an encoder is given in Table 8.8. In the truth table, D_0 to D_7 represent octal digits 0 to 7. A, B and C represent the binary digits.

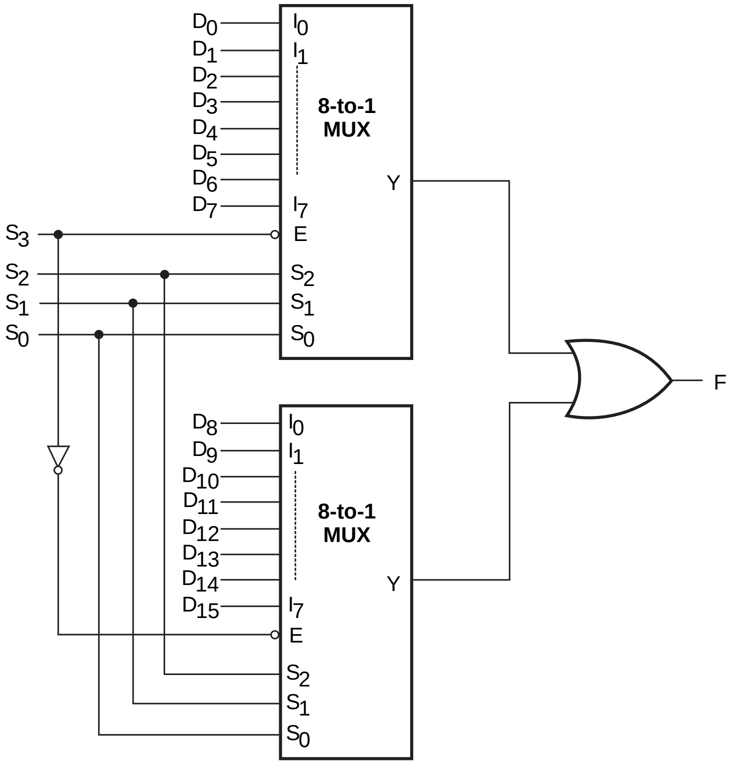


Figure 8.14 Example 8.3.

The eight input lines would have $2^8 = 256$ possible combinations. However, in the case of an octal-to-binary encoder, only eight of these 256 combinations would have any meaning. The remaining combinations of input variables are 'don't care' input combinations. Also, only one of the input lines at a time is in logic '1' state. Figure 8.15 shows the hardware implementation of the octal-to-binary encoder described by the truth table in Table 8.8. This circuit has the shortcoming that it produces an all 0s output sequence when all input lines are in logic '0' state. This can be overcome by having an additional line to indicate an all 0s input sequence.

8.2.1 Priority Encoder

A *priority encoder* is a practical form of an encoder. The encoders available in IC form are all priority encoders. In this type of encoder, a priority is assigned to each input so that, when more than one input is simultaneously active, the input with the highest priority is encoded. We will illustrate the concept of priority encoding with the help of an example. Let us assume that the octal-to-binary encoder described in the previous paragraph has an input priority for higher-order digits. Let us also assume that input lines D_2, D_4 and D_7 are all simultaneously in logic '1' state. In that case, only D_7 will be encoded and the output will be 111. The truth table of such a priority

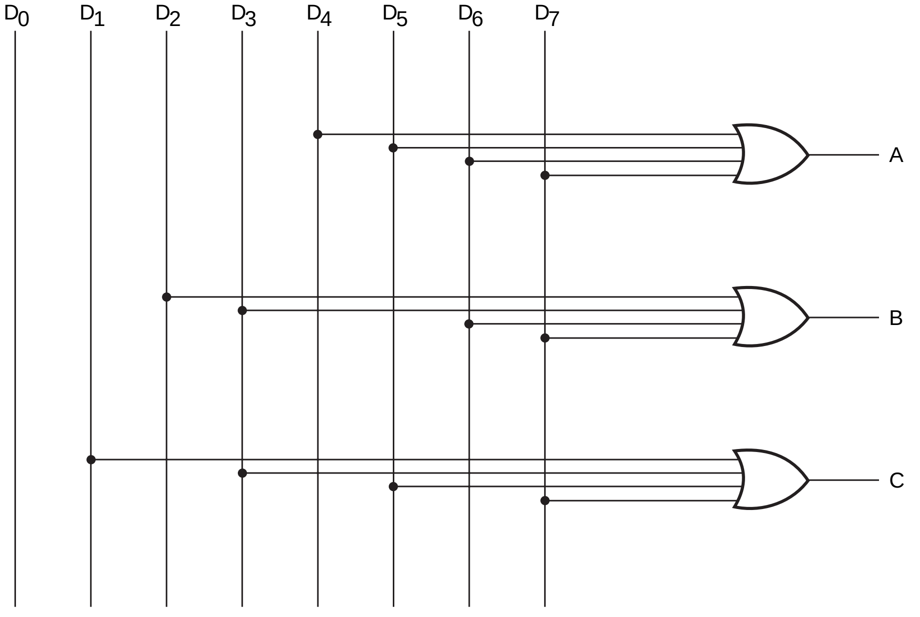


Figure 8.15 Octal-to-binary encoder.

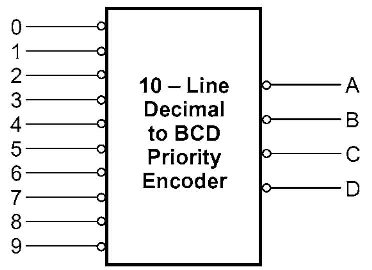
Table 8.8 Truth table of an encoder.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

encoder will then be modified to what is shown in Table 8.9. Looking at the last row of the table, it implies that, if $D_7 = 1$, then, irrespective of the logic status of other inputs, the output is 111 as D_7 will only be encoded. As another example, Fig. 8.16 shows the logic symbol and truth table of a 10-line decimal to four-line BCD encoder providing priority encoding for higher-order digits, with digit 9 having the highest priority. In the functional table shown, the input line with highest priority having a LOW on it is encoded irrespective of the logic status of the other input lines.

Table 8.9 Priority encoder.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	1
X	X	1	0	0	0	0	0	0	1	0
X	X	X	1	0	0	0	0	0	1	1
X	X	X	X	1	0	0	0	1	0	0
X	X	X	X	X	1	0	0	1	0	1
X	X	X	X	X	X	1	0	1	1	0
X	X	X	X	X	X	X	1	1	1	1



Inputs										Outputs			
0	1	2	3	4	5	6	7	8	9	D	C	B	A
X	X	X	X	X	X	X	X	X	0	0	1	1	0
X	X	X	X	X	X	X	X	0	1	0	1	1	1
X	X	X	X	X	X	X	0	1	1	1	0	0	0
X	X	X	X	X	X	0	1	1	1	1	0	0	1
X	X	X	X	X	0	1	1	1	1	1	0	1	0
X	X	X	X	0	1	1	1	1	1	1	0	1	1
X	X	X	0	1	1	1	1	1	1	1	1	0	0
X	X	0	1	1	1	1	1	1	1	1	1	0	1
X	0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 8.16 10-line decimal to four-line BCD priority encoder.

Some of the encoders available in IC form provide additional inputs and outputs to allow expansion. IC 74148, which is an eight-line to three -line priority encoder, is an example. ENABLE-IN (EI) and ENABLE-OUT (EO) terminals on this IC allow expansion. For instance, two 74148s can be cascaded to build a 16-line to four-line priority encoder.

Example 8.4

We have an eight-line to three-line priority encoder circuit with $D_0, D_1, D_2, D_3, D_4, D_5, D_6$ and D_7 as the data input lines. the output bits are A (MSB), B and C (LSB). Higher-order data bits have been assigned a higher priority, with D_7 having the highest priority. If the data inputs and outputs are active when LOW, determine the logic status of output bits for the following logic status of data inputs:

(a) All inputs are in logic '0' state.
(b) D_1 to D_4 are in logic '1' state and D_5 to D_7 are in logic '0' state.
(c) D_7 is in logic '0' state. The logic status of the other inputs is not known.

Solution

(a) Since all inputs are in logic '0' state, it implies that all inputs are active. Since D_7 has the highest priority and all inputs and outputs are active when LOW, the output bits are $A = 0$, $B = 0$ and $C = 0$.
(b) Inputs D_5 to D_7 are the ones that are active. among these, D_7 has the highest priority. Therefore, the output bits are $A = 0$, $B = 0$ and $C = 0$.
(c) D_7 is active. Since D_7 has the highest priority, it will be encoded irrespective of the logic status of other inputs. Therefore, the output bits are $A = 0$, $B = 0$ and $C = 0$.

Example 8.5

Design a four-line to two-line priority encoder with active HIGH inputs and outputs, with priority assigned to the higher-order data input line.

Solution

The truth table for such a priority encoder is given in Table 8.10, with D_0, D_1, D_2 and D_3 as data inputs and X and Y as outputs.

The Boolean expressions for the two output lines X and $Y$are given by the equations

$$X = D_2.\overline{D}_3 + D_3 = D_2 + D_3 \tag{8.5}$$

$$Y = D_1.\overline{D}_2.\overline{D}_3 + D_3 = D_1.\overline{D}_2 + D_3 \tag{8.6}$$

Figure 8.17 shows the logic diagram that implements the Boolean functions given in equations (8.5) and (8.6).

Table 8.10 Example 8.5.

D_0	D_1	D_2	D_3	X	Y
1	0	0	0	0	0
X	1	0	0	0	1
X	X	1	0	1	0
X	X	X	1	1	1

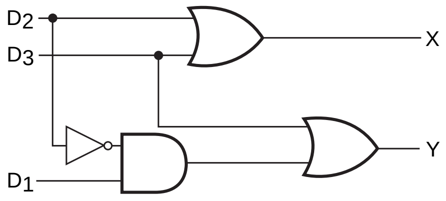
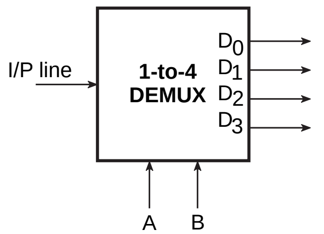


Figure 8.17 Example 8.5.

8.3 Demultiplexers and Decoders

A *demultiplexer* is a combinational logic circuit with an input line, 2^n output lines and n select lines. It routes the information present on the input line to any of the output lines. The output line that gets the information present on the input line is decided by the bit status of the selection lines. A *decoder* is a special case of a demultiplexer without the input line. Figure 8.18(a) shows the circuit representation of a 1-to-4 demultiplexer. Figure 8.18(b) shows the truth table of the demultiplexer when the input line is held HIGH.

A decoder, as mentioned earlier, is a combinational circuit that decodes the information on n input lines to a maximum of 2^n unique output lines. Figure 8.19 shows the circuit representation of 2-to-4, 3-to-8 and 4-to-16 line decoders. If there are some unused or 'don't care' combinations in the n-bit code, then there will be fewer than 2^n output lines. As an illustration, if there are three input lines, it



(a)

I/P	Select		O/P			
	A	B	D_0	D_1	D_2	D_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(b)

Figure 8.18 1-to-4 demultiplexer.

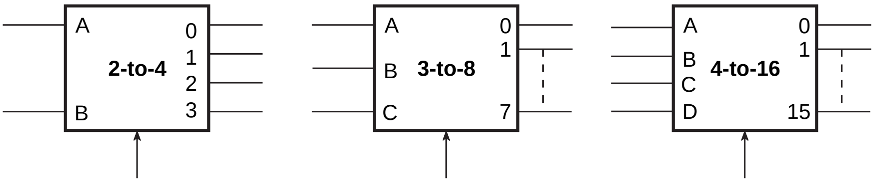


Figure 8.19 Circuit representation of 2-to-4, 3-to-8 and 4-to-16 line decoders.

can have a maximum of eight unique output lines. If, in the three-bit input code, the only used three-bit combinations are 000, 001, 010, 100, 110 and 111 (011 and 101 being either unused or don't care combinations), then this decoder will have only six output lines. In general, if n and m are respectively the numbers of input and output lines, then $m \leq 2^n$.

A decoder can generate a maximum of 2^n possible minterms with an n-bit binary code. In order to illustrate further the operation of a decoder, consider the logic circuit diagram in Fig. 8.20. This logic circuit, as we will see, implements a 3-to-8 line decoder function. This decoder has three inputs designated as A, B and C and eight outputs designated as D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7. From the truth table given along with the logic diagram it is clear that, for any given input combination, only one of the eight outputs is in logic '1' state. Thus, each output produces a certain minterm that corresponds to the binary number currently present at the input. In the present case, D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7 respectively represent the following minterms:

$$D_0 \to \overline{A}.\overline{B}.\overline{C}, D_1 \to \overline{A}.\overline{B}.C, D_2 \to \overline{A}.B.\overline{C}, D_3 \to \overline{A}.B.C$$

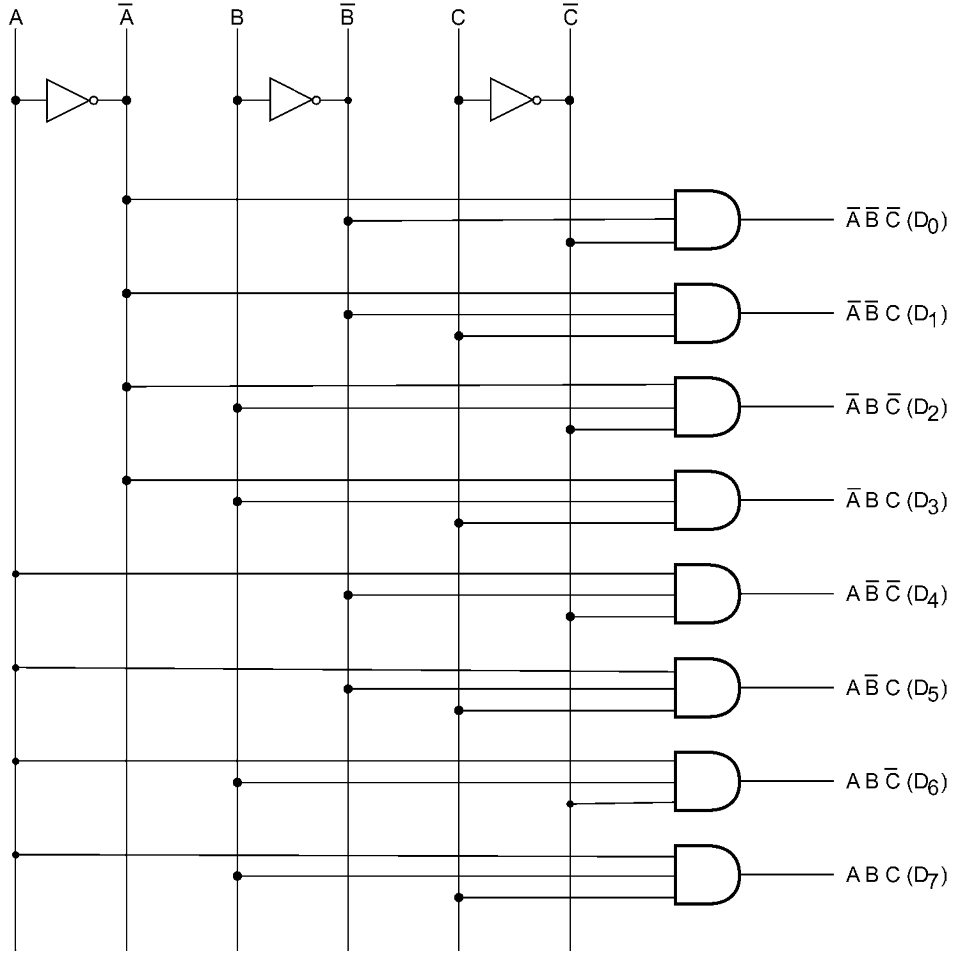
$$D_4 \to \overline{A}.\overline{B}.\overline{C}, D_5 \to A.\overline{B}.C, D_6 \to A.B.\overline{C}, D_7 \to A.B.C$$

8.3.1 Implementing Boolean Functions with Decoders

A decoder can be conveniently used to implement a given Boolean function. The decoder generates the required minterms and an external OR gate is used to produce the sum of minterms. Figure 8.21 shows the logic diagram where a 3-to-8 line decoder is used to generate the Boolean function given by the equation

$$Y = A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.B.C + \overline{A}.\overline{B}.\overline{C} \tag{8.7}$$

In general, an n-to-2^n decoder and m external OR gates can be used to implement any combinational circuit with n inputs and m outputs. We can appreciate that a Boolean function with a large number of minterms, if implemented with a decoder and an external OR gate, would require an OR gate with an equally large number of inputs. Let us consider the case of implementing a four-variable Boolean function with 12 minterms using a 4-to-16 line decoder and an external OR gate. The OR gate here needs to be a 12-input gate. In all such cases, where the number of minterms in a given Boolean function with n variables is greater than $2^n/2$ (or 2^{n-1}), the complement Boolean function will have fewer minterms. In that case it would be more advantageous to do NORing of minterms of the complement Boolean function using a NOR gate rather than doing ORing of the given function using an OR gate. The output will be nothing but the given Boolean function.



INPUTS			OUTPUTS							
A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 8.20 Logic diagram of a 3-to-8 line decoder.

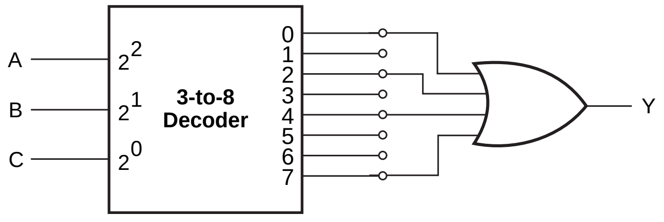


Figure 8.21 Implementing Boolean functions with decoders.

8.3.2 Cascading Decoder Circuits

There can possibly be a situation where the desired number of input and output lines is not available in IC decoders. More than one of these devices of a given size may be used to construct a decoder that can handle a larger number of input and output lines. For instance, 3-to-8 line decoders can be used to construct 4-to-16 or 5-to-32 or even larger decoder circuits. The basic steps to be followed to carry out the design are as follows:

1. If n is the number of input lines in the available decoder and N is the number of input lines in the desired decoder, then the number of individual decoders required to construct the desired decoder circuit would be 2^{N-n}.
2. Connect the less significant bits of the input lines of the desired decoder to the input lines of the available decoder.
3. The left-over bits of the input lines of the desired decoder circuit are used to enable or disable the individual decoders.
4. The output lines of the individual decoders together constitute the output lines, with the outputs of the less significant decoder constituting the less significant output lines and those of the higher–order decoders constituting the more significant output lines. The concept is further illustrated in solved example 8.8, which gives the design of a 4-to-16 decoder using 3-to-8 decoders.

Example 8.6

Implement a full adder circuit using a 3-to-8 line decoder.

Solution

A decoder with an OR gate at the output can be used to implement the given Boolean function. The decoder should at least have as many input lines as the number of variables in the Boolean function to be implemented. The truth table of the full adder is given in Table 8.11, and Fig. 8.22 shows the hardware implementation.

From the truth table, Boolean functions for SUM and CARRY outputs are given by the following equations:

$$\text{Sum output } S = \Sigma\ 1, 2, 4, 7 \tag{8.8}$$

$$\text{Carry output } C_o = \Sigma\ 3, 5, 6, 7 \tag{8.9}$$

Table 8.11 Example 8.6.

A	B	C	S	C_o
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

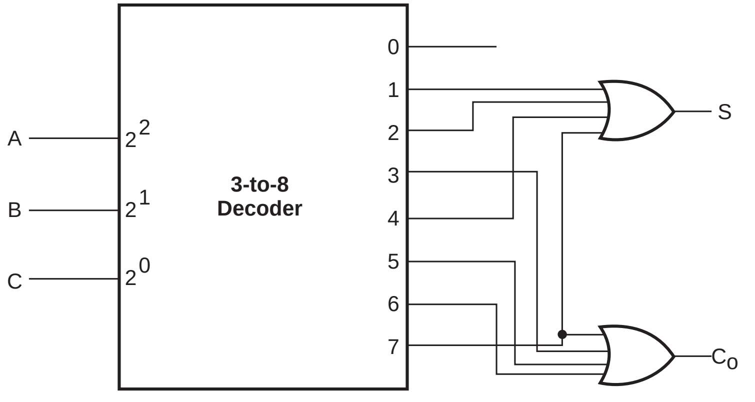


Figure 8.22 Example 8.6.

Example 8.7

A combinational circuit is defined by F = Σ 0, 2, 5, 6, 7. Hardware implement the Boolean function F with a suitable decoder and an external OR/NOR gate having the minimum number of inputs.

Solution

The given Boolean function has five three-variable minterms. This implies that the function can be implemented with a 3-to-8 line decoder and a five-input OR gate. Also, $\overline{F}$ will have only three three-variable minterms, which means that F could also be implemented by considering minterms corresponding to the complement function and using a three-input NOR gate at the output. The second option uses a NOR gate with fewer inputs and therefore is used instead. $F = \Sigma$ 0, 2, 5, 6, 7. Therefore, $\overline{F} = \Sigma$ 1, 3, 4.

Figure 8.23 shows the hardware implementation of Boolean function F.

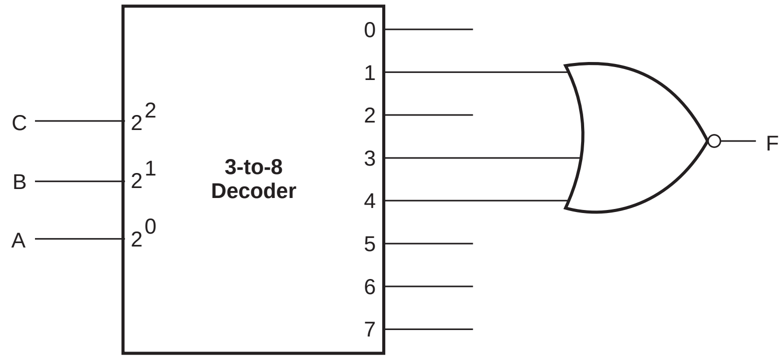


Figure 8.23 Example 8.7.

Example 8.8

Construct a 4-to-16 line decoder with two 3-to-8 line decoders having active LOW ENABLE inputs.

Solution

Let us assume that A (LSB), B, C and D (MSB) are the input variables for the 4-to-16 line decoder. Following the steps outlined earlier, A (LSB), B and C (MSB) will then be the input variables for the two 3-to-8 line decoders. If we recall the 16 possible input combinations from 0000 to 1111 in the case of a 4-to-16 line decoder, we find that the first eight combinations have $D = 0$, with CBA going through 000 to 111. The higher-order eight combinations all have $D = 1$, with CBA going through 000 to 111. If we use the D-bit as the ENABLE input for the less significant 3-to-8 line decoder and the $\overline{D}$-bit as the ENABLE input for the more significant 3-to-8 line decoder, the less significant 3-to-8 line decoder will be enabled for the less significant eight of the 16 input combinations, and the more significant 3-to-8 line decoder will be enabled for the more significant of the 16 input combinations. Figure 8.24 shows the hardware implementation. One of the output lines D_0 to D_{15} is activated as the input bit sequence $DCBA$ goes through 0000 to 1111.

Example 8.9

Figure 8.25 shows the logic symbol of IC 74154, which is a 4-to-16 line decoder/demultiplexer. The logic symbol is in ANSI/IEEE format. Determine the logic status of all 16 output lines for the following conditions:

(a) $D = HIGH$, $C = HIGH$, $B = LOW$, $A = HIGH$, $\overline{G_1} = LOW$ *and* $\overline{G_2} = LOW$.
(b) $D = HIGH$, $C = HIGH$, $B = LOW$, $A = HIGH$, $\overline{G_1} = HIGH$ *and* $\overline{G_2} = HIGH$.
(c) $D = HIGH$, $C = HIGH$, $B = LOW$, $A = HIGH$, $\overline{G_1} = HIGH$ *and* $\overline{G_2} = HIGH$.

Solution

It is clear from the given logic symbol that the device has active HIGH inputs, active LOW outputs and two active LOW ENABLE inputs. Also, both ENABLE inputs need to be active for the decoder to function owing to the indicated ANDing of the two ENABLE inputs.

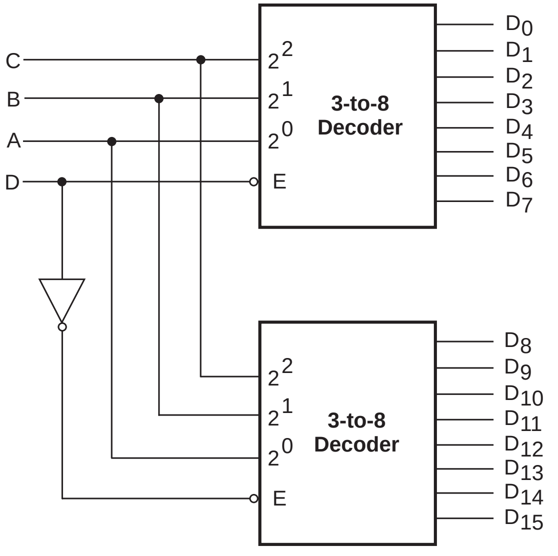


Figure 8.24 Example 8.8.

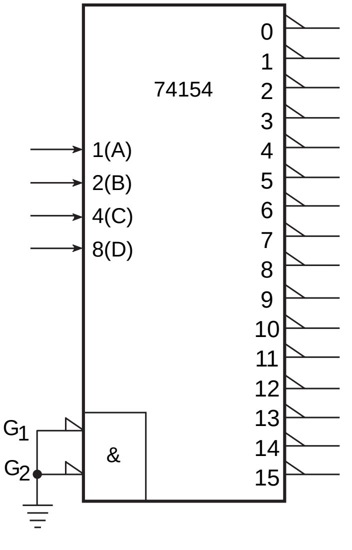


Figure 8.25 Example 8.9.

(a) Since both ENABLE inputs are active, the decoder outputs will therefore be active depending upon the logic status of the input lines. For the given logic status of the input lines, decoder output line 13 will be active and therefore LOW. All other output lines will be inactive and therefore in the logic HIGH state.
(b) Since neither ENABLE input is active, all decoder outputs will be inactive and in the logic HIGH state.
(c) The same as (b).

Example 8.10

The decoder of example 8.9 is to be used as a 1-of-16 demultiplexer. A logically compatible pulsed waveform is to be switched between output line 9 and line 15 when the logic status of an external control input is LOW and HIGH respectively. Draw the logic diagram indicating the logic status of ENABLE inputs and DCBA inputs and the point of application of the pulsed waveform.

Solution

Figure 8.26 shows the logic diagram. When the external control input is in the logic LOW state, $D =$ HIGH, $C =$ LOW, $B =$ LOW and $A =$ HIGH. This means that output line 9 is activated. When the external control input is in the logic HIGH state, $D =$ HIGH, $C =$ HIGH, $B =$ HIGH and $A =$ HIGH. This means that output line 15 is activated. In the logic diagram shown in Fig. 8.26, the two ENABLE inputs are tied together and the pulsed waveform is applied to a common point. This means that either both ENABLE inputs are active (when the input waveform is in the logic LOW state) or inactive (when the input waveform is in the logic HIGH state). Thus, when the input waveform is in the logic LOW state, output line 9 will be in the logic LOW state and all other output lines will be in the logic HIGH state provided the external control input is also in the logic LOW state. If the external

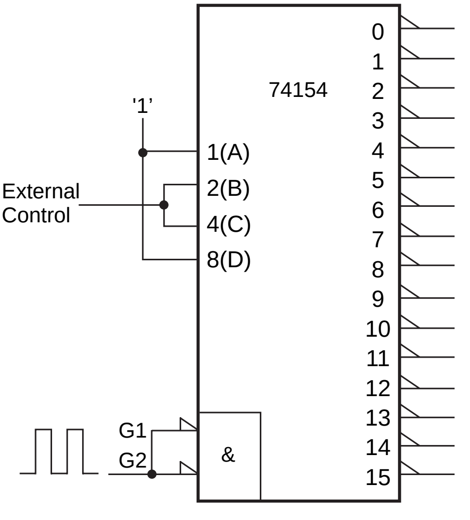


Figure 8.26 Example 8.10.

control input is in the logic HIGH state, logic LOW in the input waveform appears at output line 15. In essence, the logic status of the input waveform is reproduced at either line 9 or line 15, depending on whether the external control signal is LOW or HIGH.

8.4 Application-Relevant Information

Table 8.12 lists commonly used IC type numbers used as multiplexers, encoders, demultiplexers and decoders. Application-relevant information such as the pin connection diagram, truth table, etc., in respect of the more popular of these type numbers is given in the companion website.

Table 8.12 Commonly used IC type numbers used as multiplexers, encoders, demultiplexers and decoders.

IC Type number	Function	Logic family
7442	1-of-10 decoder	TTL
74138	1-of-8 decoder/demultiplexer	TTL
74139	Dual 1-of-4 decoder/demultiplexer	TTL
74145	1-of-10 decoder/driver (open collector)	TTL
74147	10-line to four-line priority encoder	TTL
74148	Eight-line to three-line priority encoder	TTL
74150	16-input multiplexer	TTL
74151	Eight-input multiplexer	TTL
74152	Eight-input multiplexer	TTL
74153	Dual four-input multiplexer	TTL
74154	4-of-16 decoder/demultiplexer	TTL
74155	Dual 1-of-4 decoder/demultiplexer	TTL
74156	Dual 1-of-4 decoder/demultiplexer (open collector)	TTL
74157	Quad two-input noninverting multiplexer	TTL
74158	Quad two-input inverting multiplexer	TTL
74247	BCD to seven-segment decoder/driver (open collector)	TTL
74248	BCD to seven-segment decoder/driver with Pull-ups	TTL
74251	Eight-input three-state multiplexer	TTL
74253	Dual four-input three-state multiplexer	TTL
74256	Dual four-bit addressable latch	TTL
74257	Quad two-input non-inverting three-state multiplexer	TTL
74258	Quad two-input inverting three-state multiplexer	TTL
74259	Eight-bit addressable latch	TTL
74298	Dual two-input multiplexer with output latches	TTL
74348	Eight-line to three-line priority encoder (three-state)	TTL
74353	Dual four-input multiplexer	TTL
74398	Quad two-input multiplexer with output register	TTL
74399	Quad two-input multiplexer with output register	TTL
4019	Quad two-input multiplexer	CMOS
4028	1-of-10 decoder	CMOS
40147	10-line to four-line BCD priority encoder	CMOS
4511	BCD to seven-segment latch/decoder/driver	CMOS
4512	Eight-input three-state multiplexer	CMOS
4514	1-of-16 decoder/demultiplexer with input latch	CMOS

(*continued overleaf*)

Table 8.12 *(continued).*

IC Type number	Function	Logic family
4515	1-of-16 decoder/demultiplexer with input latch	CMOS
4532	Eight-line to three-line priority encoder	CMOS
4539	Dual four-input multiplexer	CMOS
4543	BCD to seven-segment latch/decoder/driver for LCD displays	CMOS
4555	Dual 1-of-4 decoder/demultiplexers	CMOS
4556	Dual 1-of-4 decoder/demultiplexers	CMOS
4723	Dual four-bit addressable latch	CMOS
4724	Eight-bit addressable latch	CMOS
10132	Dual two-input multiplexer with latch and common reset	ECL
10134	Dual multiplexer with latch	ECL
10158	Quad two-input multiplexer (non-inverting)	ECL
10159	Quad two-input multiplexer (inverting)	ECL
10161	3-to-8 line decoder (LOW)	ECL
10162	3-to-8 line decoder (HIGH)	ECL
10164	Eight-line multiplexer	ECL
10165	Eight-input priority encoder	ECL
10171	Dual 2-to-4 line decoder (LOW)	ECL
10172	Dual 2-to-4 line decoder (HIGH)	ECL
10173	Quad two-input multiplexer/latch	ECL
10174	Dual 4-to-1 multiplexer	ECL

Review Questions

1. What is a multiplexer circuit? Briefly describe one or two applications of a multiplexer?
2. Is it possible to enhance the capability of an available multiplexer in terms of the number of input lines it can handle by using more than one device? If yes, briefly describe the procedure to do so, with the help of an example.
3. What is an encoder? How does a priority encoder differ from a conventional encoder? With the help of a truth table, briefly describe the functioning of a 10-line to four-line priority encoder with active LOW inputs and outputs and priority assigned to the higher-order inputs.
4. What is a demultiplexer and how does it differ from a decoder? Can a decoder be used as a demultiplexer? If yes, from where do we get the required input line?
5. Briefly describe how we can use a decoder optimally to implement a given Boolean function? Illustrate your answer with the help of an example.
6. Draw truth tables for the following:

   (a) an 8-to-1 multiplexer with active LOW inputs and an active LOW ENABLE input;
   (b) a four-line to 16-line decoder with active HIGH inputs and active LOW outputs and an active LOW ENABLE input;
   (c) an eight-line to three-line priority encoder with active LOW inputs and outputs and an active LOW ENABLE input.

Problems

1. Implement the three-variable Boolean function $F(A, B, C) = \overline{A}.C + A.\overline{B}.C + A.B.\overline{C}$ using (i) an 8-to-1 multiplexer and (ii) a 4-to-1 multiplexer.

   (i) Fig. 8.27(a); (ii) Fig. 8.27(b)

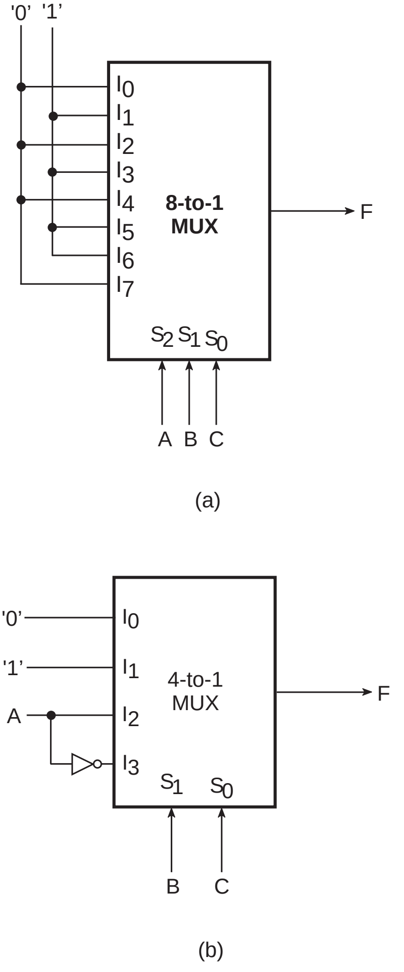


Figure 8.27 Problem 1.

2. Design a 32-to-1 multiplexer using 8-to-1 multiplexers having an active LOW ENABLE input and a 2-to-4 decoder.

   Fig. 8.28

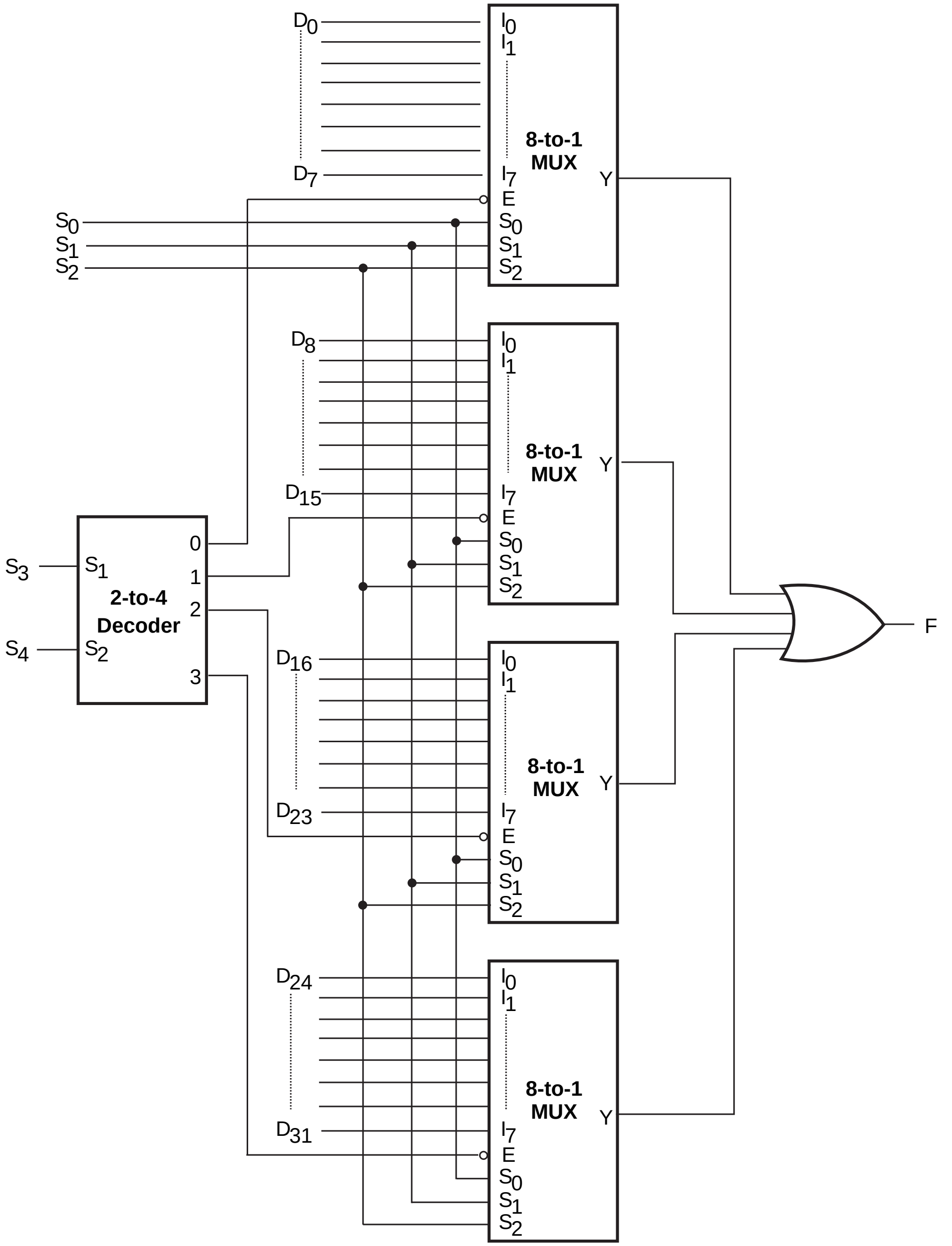


Figure 8.28 Answer to problem 2.

3. Determine the function performed by the combinational circuit of Fig. 8.29.

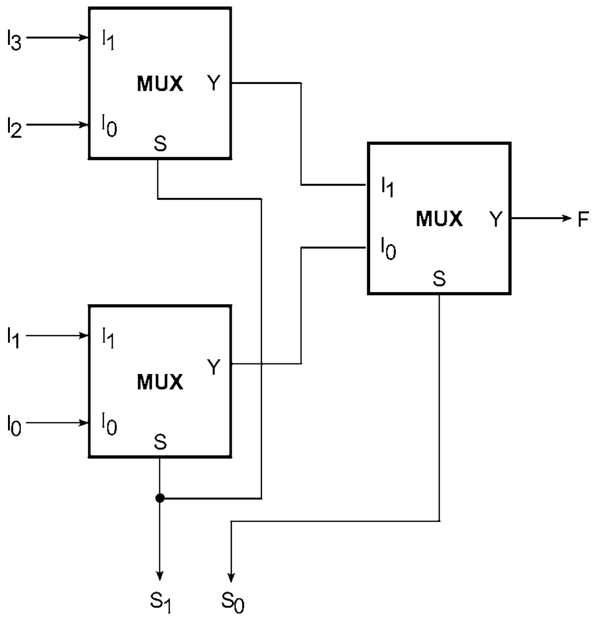


Figure 8.29 Problem 3.

4-to-1 multiplexer

4. Implement a full subtractor combinational circuit using a 3-to-8 decoder and external NOR gates.
Fig. 8.30

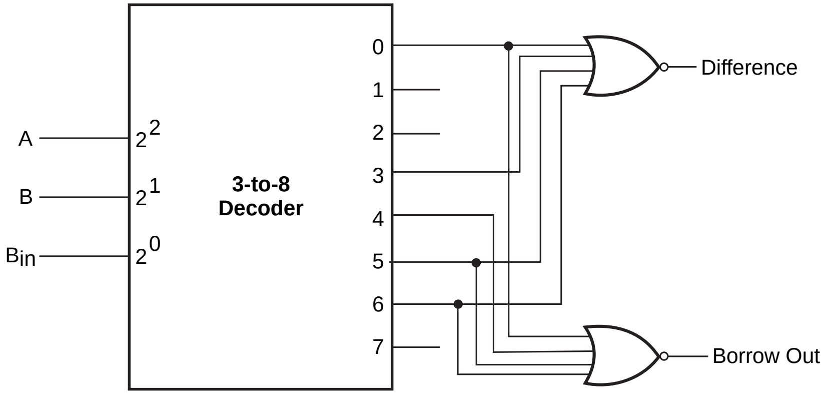


Figure 8.30 Answer to problem 4.

Further Reading

1. Floyd, T. L. (2005) *Digital Fundamentals*, Prentice-Hall Inc., USA.
2. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
3. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
4. Cook, N. P. (2003) *Practical Digital Electronics*, Prentice-Hall, NJ, USA.
5. Rafiquzzaman, M. (2005) *Fundamentals of Digital Logic and Microcomputer Design*, Wiley-Interscience, New York, USA.
6. Morris Mano, M. and Kime, C. R. (2003) *Logic and Computer Design Fundamentals*, Prentice-Hall Inc., USA.

9

Programmable Logic Devices

Logic devices constitute one of the three important classes of devices used to build digital electronics systems, memory devices and microprocessors being the other two. Memory devices such as ROM and RAM are used to store information such as the software instructions of a program or the contents of a database, and microprocessors execute software instructions to perform a variety of functions, from running a word-processing program to carrying out far more complex tasks. Logic devices implement almost every other function that the system must perform, including device-to-device interfacing, data timing, control and display operations and so on. So far, we have discussed those logic devices that perform fixed logic functions decided upon at the manufacturing stage. Logic gates, multiplexers, demultiplexers, arithmetic circuits, etc., are some examples. Sequential logic devices such as flip-flops, counters, registers, etc., to be discussed in the following chapters, also belong to this category of logic devices. In the present chapter, we will discuss a new category of logic devices called *programmable logic devices* (PLDs). The function to be performed by a programmable logic device is undefined at the time of its manufacture. These devices are programmed by the user to perform a range of functions depending upon the logic capacity and other features offered by the device. We will begin with a comparison of fixed and programmable logic, and then follow this up with a detailed description of different types of PLDs in terms of operational fundamentals, salient features, architecture and typical applications. A brief introduction to the devices offered by some of the major manufacturers of PLDs and PLD programming languages is given towards the end of the chapter.

9.1 Fixed Logic Versus Programmable Logic

As outlined in the introduction, there are two broad categories of logic devices, namely fixed logic devices and programmable logic devices. Whereas a fixed logic device such as a logic gate or a multiplexer or a flip-flop performs a given logic function that is known at the time of device manufacture, a programmable logic device can be configured by the user to perform a large variety of

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

logic functions. In terms of the internal schematic arrangement of the two types of device, the circuits or building blocks and their interconnections in a fixed logic device are permanent and cannot be altered after the device is manufactured.

A *programmable logic device* offers to the user a wide range of logic capacity in terms of digital building blocks, which can be configured by the user to perform the intended function or set of functions. This configuration can be modified or altered any number of times by the user by reprogramming the device. Figure 9.1 shows a simple logic circuit comprising four three-input AND gates and a four-input OR gate. This circuit produces an output that is the sum output of a full adder. Here, *A* and *B* are the two bits to be added, and *C* is the carry-in bit. It is a fixed logic device as the circuit is unalterable from outside owing to fixed interconnections between the various building blocks.

Figure 9.2 shows the logic diagram of a simple programmable device. The device has an array of four six-input AND gates at the input and a four-input OR gate at the output. Each AND gate can handle three variables and thus can produce a product term of three variables. The three variables (*A*, *B* and *C* in this case) or their complements can be programmed to appear at the inputs of any of the four AND gates through fusible links called antifuses. This means that each AND gate can produce the desired three-variable product term. It may be mentioned here that an antifuse performs a function that is opposite to that performed by a conventional electrical fuse. A fuse has a low initial resistance and permanently breaks an electrically conducting path when current through it exceeds a certain limiting value. In the case of an antifuse, the initial resistance is very high and it is designed to create a low-resistance electrically conducting path when voltage across it exceeds a certain level. As a result, this circuit can be programmed to generate any three-variable sum-of-products Boolean function having four minterms by activating the desired fusible links. For example, the circuit could be programmed to produce the sum output resulting from the addition of three bits (the sum output in the case of a full adder) or to produce difference outputs resulting from subtraction of two bits with a borrow-in (the difference output in the case of a full subtractor).

We can visualize that the logic circuit of Fig. 9.2 has a programmable AND array at the input and a fixed OR gate at the output. Incidentally, this is the architecture of programmable logic devices called programmable array logic (PAL). Practical PAL devices have a much larger number of programmable AND gates and fixed OR gates to have enhanced logic capacity and performance capability. PAL devices are discussed in detail in the latter part of the chapter.

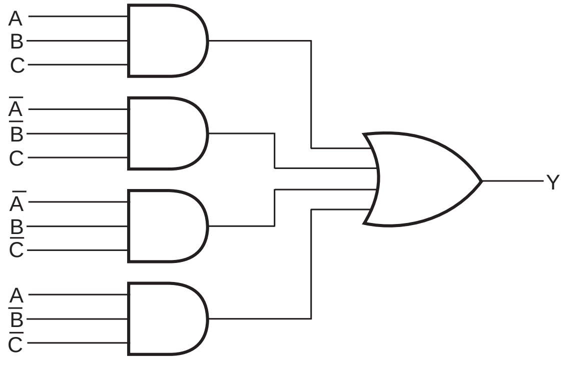


Figure 9.1 Fixed logic circuit.

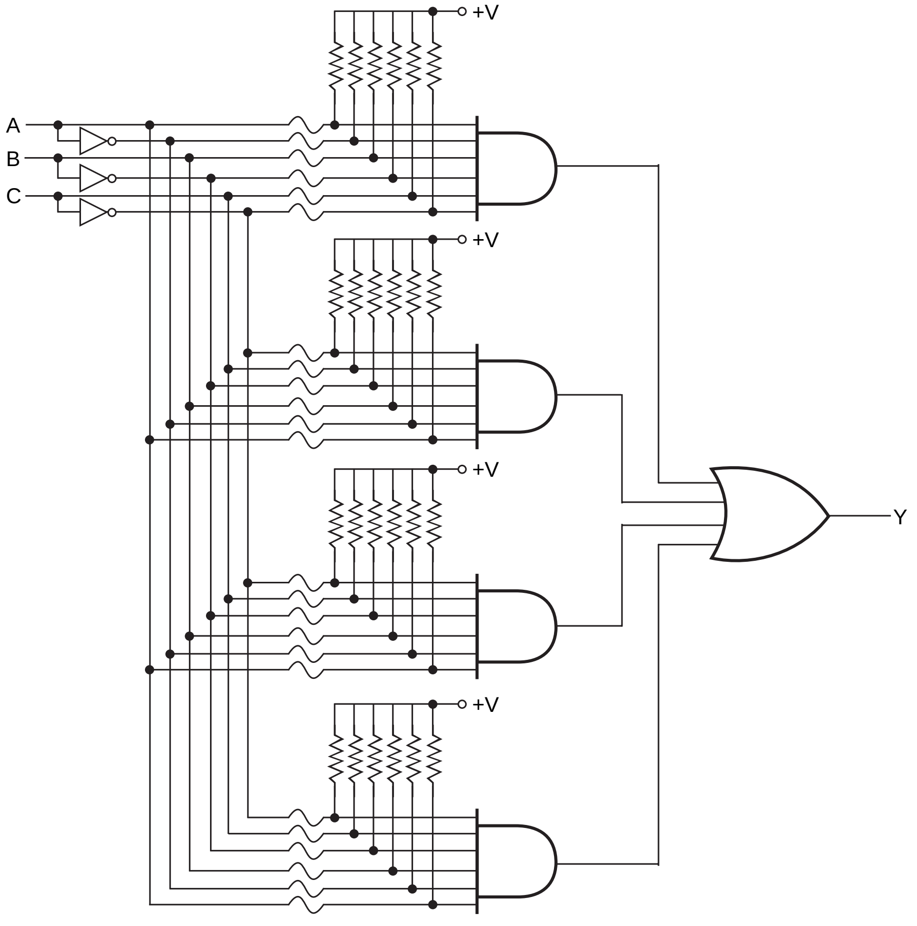


Figure 9.2 Simple programmable logic circuit.

9.1.1 Advantages and Disadvantages

1. If we want to build a fixed logic device to perform a certain specific function, the time required from design to the final stage when the manufactured device is actually available for use could easily be several months to a year or so. PLD-based design requires much less time from design cycle to production run.
2. In the case of fixed logic devices, the process of design validation followed by incorporation of changes, if any, involves substantial nonrecurring engineering (NRE) costs, which leads to an enhanced cost of the initial prototype device. In the case of PLDs, inexpensive software tools can be used for quick validation of designs. The programmable feature of these devices allows quick incorporation of changes and also a quick testing of the device in an actual application environment. In this case, the device used for prototyping is the same as the one that would qualify for use in the end equipment.

3. In the case of programmable logic devices, users can change the circuit as often as they want to until the design operates to their satisfaction. PLDs offer to the users much more flexibility during the design cycle. Design iterations are nothing but changes to the programming file.
4. Fixed logic devices have an edge for large-volume applications as they can be mass produced more economically. They are also the preferred choice in applications requiring the highest performance level.

9.2 Programmable Logic Devices – An Overview

There are many types of programmable logic device, distinguishable from one another in terms of architecture, logic capacity, programmability and certain other specific features. In this section, we will briefly discuss commonly used PLDs and their salient features. A detailed description of each of them will follow in subsequent sections.

9.2.1 Programmable ROMs

PROM (Programmable Read Only Memory) and EPROM (Erasable Programmable Read Only Memory) can be considered to be predecessors to PLDs. The architecture of a programmable ROM allows the user to hardware-implement an arbitrary combinational function of a given number of inputs. When used as a memory device, n inputs of the ROM (called address lines in this case) and m outputs (called data lines) can be used to store $2^n m$-bit words. When used as a PLD, it can be used to implement m different combinational functions, with each function being a chosen function of n variables. Any conceivable n-variable Boolean function can be made to appear at any of the m output lines. A generalized ROM device with n inputs and m outputs has 2^n hard-wired AND gates at the input and m programmable OR gates at the output. Each AND gate has n inputs, and each OR gate has 2^n inputs. Thus, each OR gate can be used to generate any conceivable Boolean function of n variables, and this generalized ROM can be used to produce m arbitrary n-variable Boolean functions. The AND array produces all possible minterms of a given number of input variables, and the programmable OR array allows only the desired minterms to appear at their inputs. Figure 9.3 shows the internal architecture of a PROM having four input lines, a hard-wired array of 16 AND gates and a programmable array of four OR gates. A cross ($\times$) indicates an intact (or unprogrammed) fusible link or interconnection, and a dot (•) indicates a hard-wired interconnection. PROMs, EPROMs and EEPROMs (Electrically Erasable Programmable Read Only Memory) can be programmed using standard PROM programmers. One of the major disadvantages of PROMs is their inefficient use of logic capacity. It is not economical to use PROMs for all those applications where only a few minterms are needed. Other disadvantages include relatively higher power consumption and an inability to provide safe covers for asynchronous logic transitions. They are usually much slower than the dedicated logic circuits. Also, they cannot be used to implement sequential logic owing to the absence of flip-flops.

9.2.2 Programmable Logic Array

A *programmable logic array* (PLA) device has a programmable AND array at the input and a programmable OR array at the output, which makes it one of the most versatile PLDs. Its architecture differs from that of a PROM in the following respects. It has a programmable AND array rather than a hard-wired AND array. The number of AND gates in an m-input PROM is always equal to 2^m. In the case of a PLA, the number of AND gates in the programmable AND array for m input variables

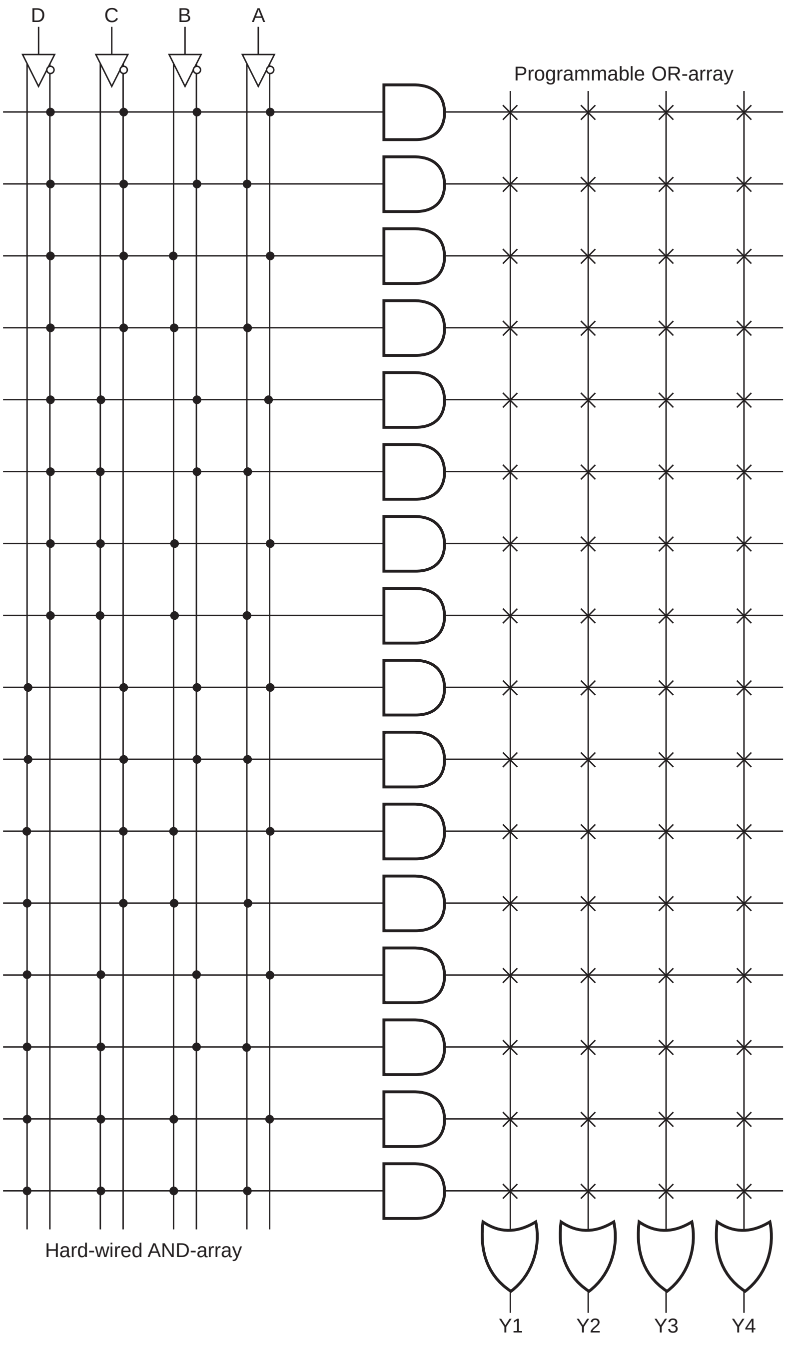


Figure 9.3 Internal architecture of a PROM.

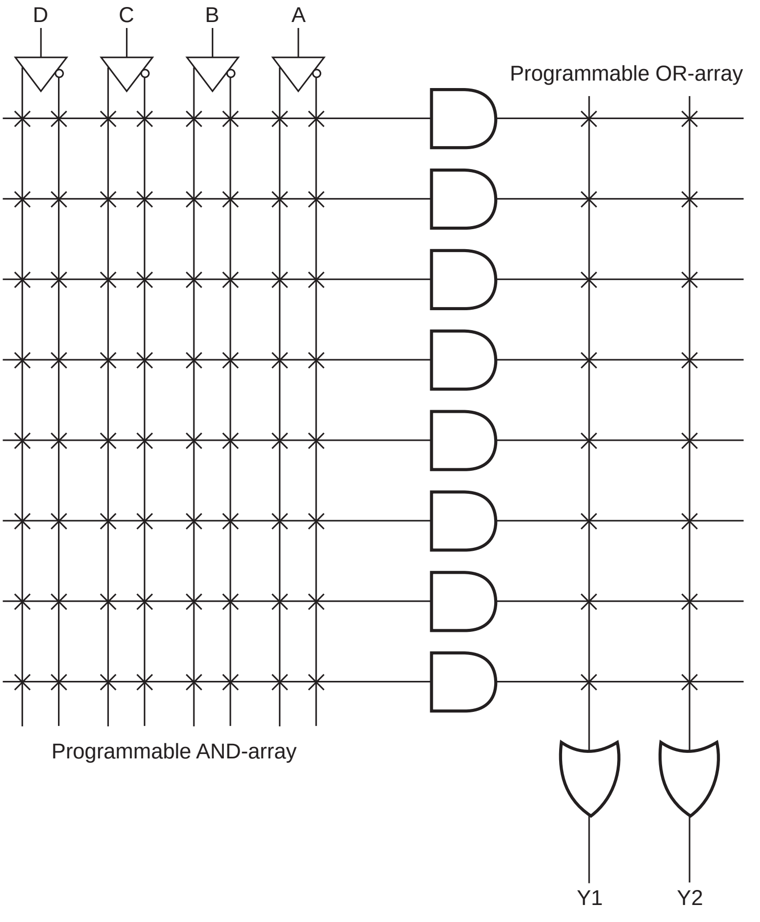


Figure 9.4 Internal architecture of a PLA device.

is usually much less than 2^m, and the number of inputs of each of the OR gates equals the number of AND gates. Each OR gate can generate an arbitrary Boolean function with a maximum of minterms equal to the number of AND gates. Figure 9.4 shows the internal architecture of a PLA device with four input lines, a programmable array of eight AND gates at the input and a programmable array of two OR gates at the output. A PLA device makes more efficient use of logic capacity than a PROM. However, it has its own disadvantages resulting from two sets of programmable fuses, which makes it relatively more difficult to manufacture, program and test.

9.2.3 Programmable Array Logic

Programmable array logic (PAL) architecture has a programmable AND array at the input and a fixed OR array at the output. The programmable AND array of a PAL device is similar to that of a PLA device. That is, the number of programmable AND gates is usually smaller than the number required to generate all possible minterms of the given number of input variables. The OR array is fixed and the AND outputs are equally divided between available OR gates. For instance, a practical PAL device may have eight input variables, 64 programmable AND gates and four fixed OR gates, with each OR gate having 16 inputs. That is, each OR gate is fed from 16 of the 64 AND outputs. Figure 9.5 shows the internal architecture of a PAL device that has four input lines, an array of eight AND gates at the input and two OR gates at the output, to introduce readers to the arrangement of various building blocks inside a PAL device and allow them a comparison between different programmable logic devices.

9.2.4 Generic Array Logic

A *generic array logic* (GAL) device is similar to a PAL device and was invented by Lattice Semiconductor. It differs from a PAL device in that the programmable AND array of

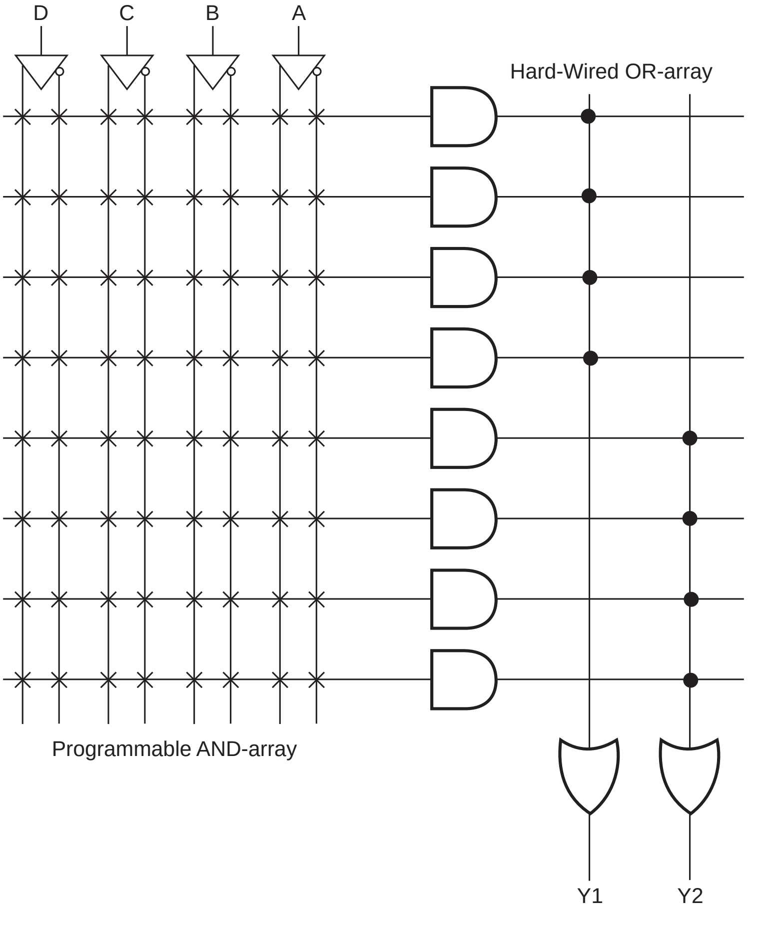


Figure 9.5 Internal architecture of a PAL device.

a GAL device can be erased and reprogrammed. Also, it has reprogrammable output logic. This feature makes it particularly attractive at the device prototyping stage, as any bugs in the logic can be corrected by reprogramming. A similar device called PEEL (Programmable Electrically Erasable Logic) was introduced by the International CMOS Technology (ICT) Corporation.

9.2.5 Complex Programmable Logic Device

Programmable logic devices such as PLAs, PALs, GALs and other PAL-like devices are often grouped into a single category called *simple programmable logic devices* (SPLDs) to distinguish them from the ones that are far more complex. A *complex programmable logic device* (CPLD), as the name suggests, is a much more complex device than any of the programmable logic devices discussed so far. A CPLD may contain circuitry equivalent to that of several PAL devices linked to each other by programmable interconnections. Figure 9.6 shows the internal structure of a typical CPLD. Each of the four logic blocks is equivalent to a PLD such as a PAL device. The number of logic blocks in a CPLD could be more or less than four. Each of the logic blocks has programmable interconnections. A switch matrix is used for logic block to logic block interconnections. Also, the switch matrix in a CPLD may or may not be fully connected. That is, some of the possible connections between logic block outputs and inputs may not be supported by a given CPLD. While the complexity of a typical PAL device may be of the order of a few hundred logic gates, a CPLD may have a complexity equivalent to tens of thousands of logic gates. When compared with FPGAs, CPLDs offer predictable timing characteristics owing to their less flexible internal architecture and are thus ideal for critical control applications and other applications where a high performance level is required. Also, because of their relatively much lower power consumption and lower cost, CPLDs are an ideal solution for battery-operated portable applications such as mobile phones, digital assistants and so on. A CPLD can be programmed either by using a PAL programmer or by feeding it with a serial data stream from a PC after soldering it on the PC board. A circuit on the CPLD decodes the data stream and configures it to perform the intended logic function.

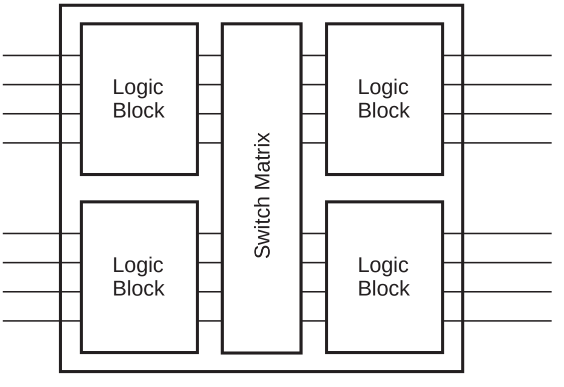


Figure 9.6 CPLD architecture.

9.2.6 Field-Programmable Gate Array

A *field-programmable gate array* (FPGA) uses an array of logic blocks, which can be configured by the user. The term 'field-programmable' here signifies that the device is programmable outside the factory where it is manufactured. The internal architecture of an FPGA device has three main parts, namely the array of logic blocks, the programmable interconnects and the I/O blocks. Figure 9.7 shows the architecture of a typical FPGA. Each of the I/O blocks provides an individually selectable input, output or bidirectional access to one of the general-purpose I/O pins on the FPGA package. The logic blocks in an FPGA are no more complex than a couple of logic gates or a look-up table feeding a flip-flop. The programmable interconnects connect logic blocks to logic blocks and also I/O blocks to logic blocks.

FPGAs offer a much higher logic density and much larger performance features compared with CPLDs. Some of the contemporary FPGA devices offer a logic complexity equivalent to that of eight million system gates. Also, these devices offer features such as built-in hard-wired processors,

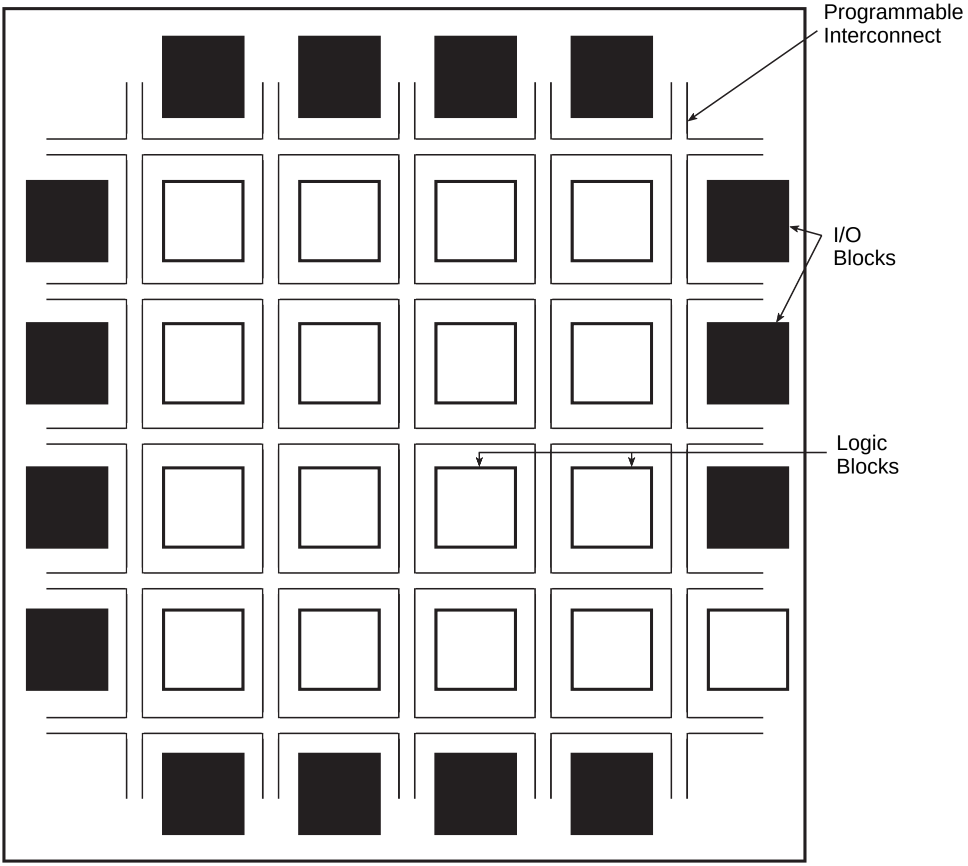


Figure 9.7 FPGA architecture.

large memory, clock management systems and support for many of the contemporary device-to-device signalling technologies. FPGAs find extensive use in a variety of applications, which include data processing and storage, digital signal processing, instrumentation and telecommunications.

FPGAs are also programmed like CPLDs after they are soldered onto the PC board. In the case of FPGAs, the programmed configuration is usually volatile and therefore needs to be reloaded whenever power is applied or a different functionality is required.

9.3 Programmable ROMs

A *read only memory* (ROM) is essentially a memory device that can be used to store a certain fixed set of binary information. As outlined earlier, these devices have certain inherent links that can be made or broken depending upon the type of fusible link to store any user-specified binary information in the device. While, in the case of a conventional fusible link, relevant interconnections are broken to program the device, in the case of an antifuse the relevant interconnections are made to do the same job. This is illustrated in Fig. 9.8. Figure 9.8(a) shows the internal logic diagram of a 4×2 PROM. The figure shows an unprogrammed PROM. Figures 9.8(b) and (c) respectively show the use of a fuse and an antifuse to produce output-1 = AB. Note that in the case of a fuse an unprogrammed interconnection is a 'make' connection, whereas in the case of an antifuse it is a 'break' connection.

Once a given pattern is formed, it remains as such even if power is turned off and on. In the case of PROMs, the user can erase the data already stored on the ROM chip and load it with fresh data. Memory-related issues of ROMs are discussed in detail in Chapter 15 on microcomputer fundamentals. In the present section, we will discuss the use of a PROM as a programmable logic device for implementation of combinational logic functions, which is one of the most widely exploited applications of PROMs. A PROM in general has n input lines and m output lines and is designated as a $2^n \times m$ PROM. Looking at the internal architecture of a PROM device, it is a combinational circuit with the AND gates wired as a decoder and having OR gates equal to the number of outputs. A PROM with five input lines and four output lines, for instance, would have the equivalent of a 5×32 decoder at the input that would generate 32 possible minterms or product terms. Each of these four OR gates would be a 32-input gate fed from 32 outputs of the decoder through fusible links.

Figure 9.9 shows the internal architecture of a 32×4 PROM. We can see that the input side is hard-wired to produce all possible 32 product terms corresponding to five variables. All 32 product terms or minterms are available at the inputs of each of the OR gates through programmable interconnections. This allows the users to have four different five-variable Boolean functions of their choice. Very complex combinational functions can be generated with PROMs by suitably making or breaking these links.

To sum up, for implementing an n-input or n-variable, $m-$output combinational circuit, one would need a $2^n \times m$ PROM. As an illustration, let us see how a PROM can be used to implement the following Boolean function with two outputs given by the equations

$$F_1(A, B, C) = \Sigma 0, 2 \tag{9.1}$$

$$F_2(A, B, C) = \Sigma 1, 4, 7 \tag{9.2}$$

Implementation of this Boolean function would require an 8×2 PROM. The internal logic diagram of the PROM in this case, after it is programmed, would be as shown in Fig. 9.10. Note that, in the programmed PROM of Fig. 9.10, an unprogrammed interconnection indicated by a cross ($\times$) is a 'make' connection.

It may be mentioned here that in practice a PROM would not be used to implement as simple a Boolean function as that illustrated above. The purpose here is to indicate to readers how a PROM

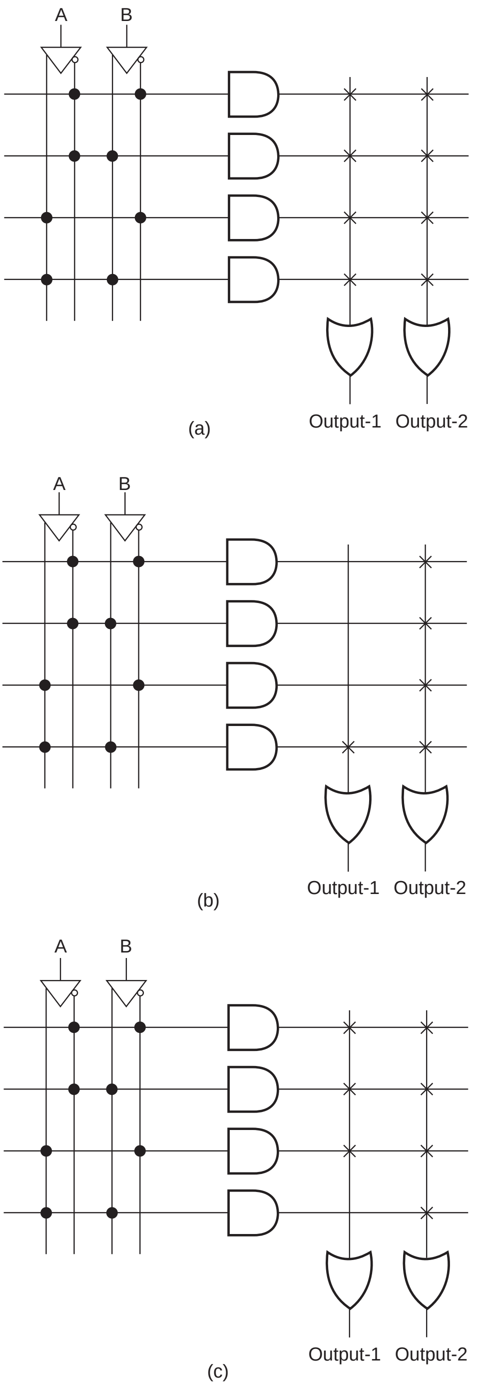


Figure 9.8 Use of fuse and antifuse.

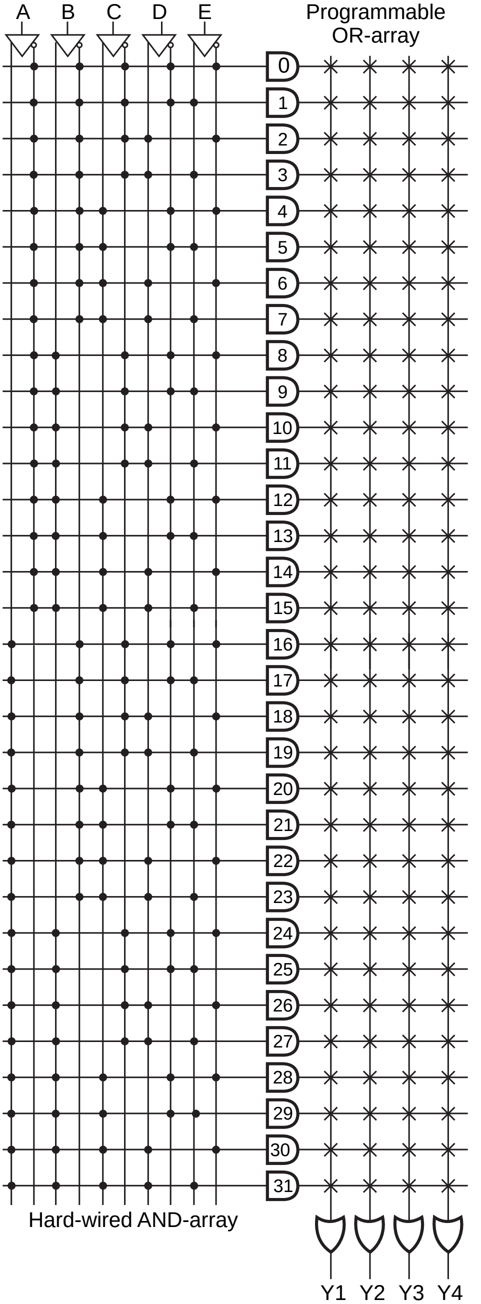


Figure 9.9 Internal architecture of a 32 × 4 PROM.

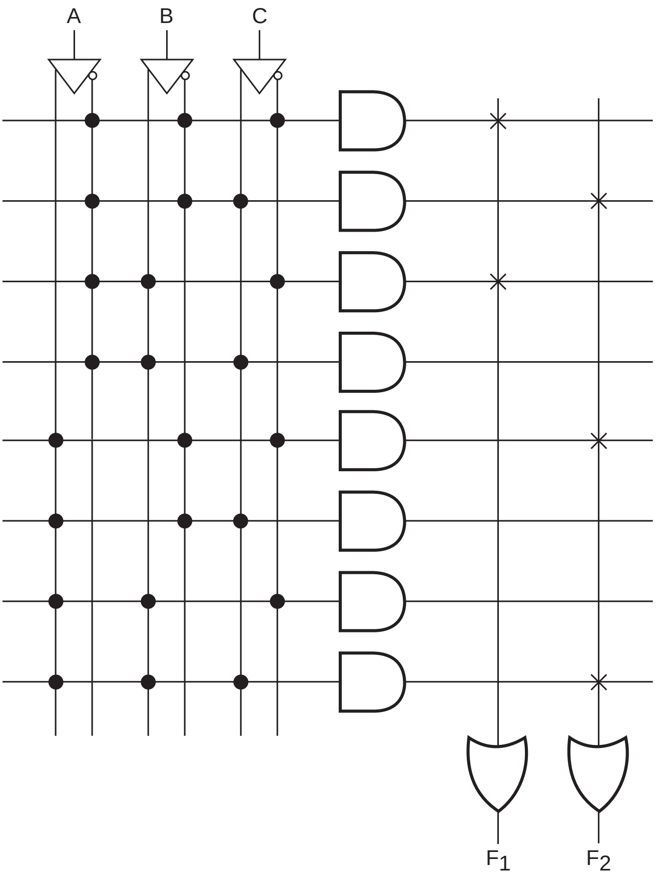


Figure 9.10 8×2 PROM internal logic diagram to implement given Boolean function.

implements a Boolean function. In actual practice, PROMs would be used only in the case of very complex Boolean functions.

Another noteworthy point is that, when it comes to implementing Boolean functions with PROMs, it is not economical to use PROM for those Boolean functions that have a large number of 'don't care' conditions. In the case of a PROM, each 'don't care' condition would have either all 0s or all 1s. In other words, the space on the chip is not optimally utilized. Other programmable logic devices such as a PLA or PAL are more suitable in such situations.

Example 9.1

Determine the size of the PROM required for implementing the following logic circuits:

(a) a binary multiplier that multiplies two four-bit numbers;
(b) a dual 8-to-1 multiplexer with common selection inputs;
(c) a single-digit BCD adder/subtractor with a control input for selection of operation.

Solution

(a) The number of inputs required here would be eight. The result of multiplication would be in eight bits. Therefore, the size of the PROM $= 2^8 \times 8 = 256 \times 8$.

(b) The number of inputs $= 8+8+3 = 19$ (the number of selection inputs $= 3$). The number of outputs $= 2$. Therefore, the size of the PROM $= 2^{19} \times 2 = 512\text{K} \times 2$.

(c) The number of inputs $= 4$ (augend bits) $+ 4$ (addend bits) $+ 1$ (carry-in) $+ 1$ (control input) $= 10$. The number of outputs $= 4$ (sum or subtraction output bits) $+ 1$ (carry or borrow bit) $= 5$. The size of the PROM $= 2^{10} \times 5 = 1024 \times 5 = 1\text{K} \times 5$.

9.4 Programmable Logic Array

A *programmable logic array* (PLA) enables logic functions expressed in sum-of-products form to be implemented directly. It is similar in concept to a PROM. However, unlike a PROM, the PLA does not provide full decoding of the input variables and does not generate all possible minterms. While a PROM has a fixed AND gate array at the input and a programmable OR gate array at the output, a PLA device has a programmable AND gate array at the input and a programmable OR gate array at the output. In a PLA device, each of the product terms of the given Boolean function is generated by an AND gate which can be programmed to form the AND of any subset of inputs or their complements. The product terms so produced can be summed up in an array of programmable OR gates. Thus, we have a programmable OR gate array at the output. The input and output gates are constructed in the form of arrays with input lines orthogonal to product lines and product lines orthogonal to output lines.

Figure 9.11 shows the internal architecture of a PLA device with four input lines, eight product lines and four output lines. That is, the programmable AND gate array has eight AND gates. Each of the AND gates here has eight inputs, corresponding to four input variables and their complements. The input to each of the AND gates can be programmed to be any of the possible 16 combinations of four input variables and their complements. Four OR gates at the output can generate four different Boolean functions, each having a maximum of eight minterms out of 16 minterms possible with four variables. The logic diagram depicts the unprogrammed state of the device. The internal architecture shown in Fig. 9.11 can also be represented by the schematic form of Fig. 9.12. PLAs usually have inverters at the output of OR gates to enable them to implement a given Boolean function in either AND-OR or AND-OR-INVERT form.

Figure 9.13 shows a generalized block schematic representation of a PLA device having n inputs, m outputs and k product terms, with n, m and k respectively representing the number of input variables, the number of OR gates and the number of AND gates. The number of inputs to each OR gate and each AND gate are k and $2n$ respectively.

A PLA is specified in terms of the number of inputs, the number of product terms and the number of outputs. As is clear from the description given in the preceding paragraph, the PLA would have a total of $2\text{K}n + \text{K}m$ programmable interconnections. A ROM with the same number of input and output lines would have $2^n \times \text{m}$ programmable interconnections.

A PLA could be either mask programmable or field programmable. In the case of a mask-programmable PLA, the customer submits a program table to the manufacturer to produce a custom-made PLA having the desired internal paths between inputs and outputs. A *field-programmable logic array* (FPLA) is programmed by the users themselves by means of a hardware programmer unit available commercially.

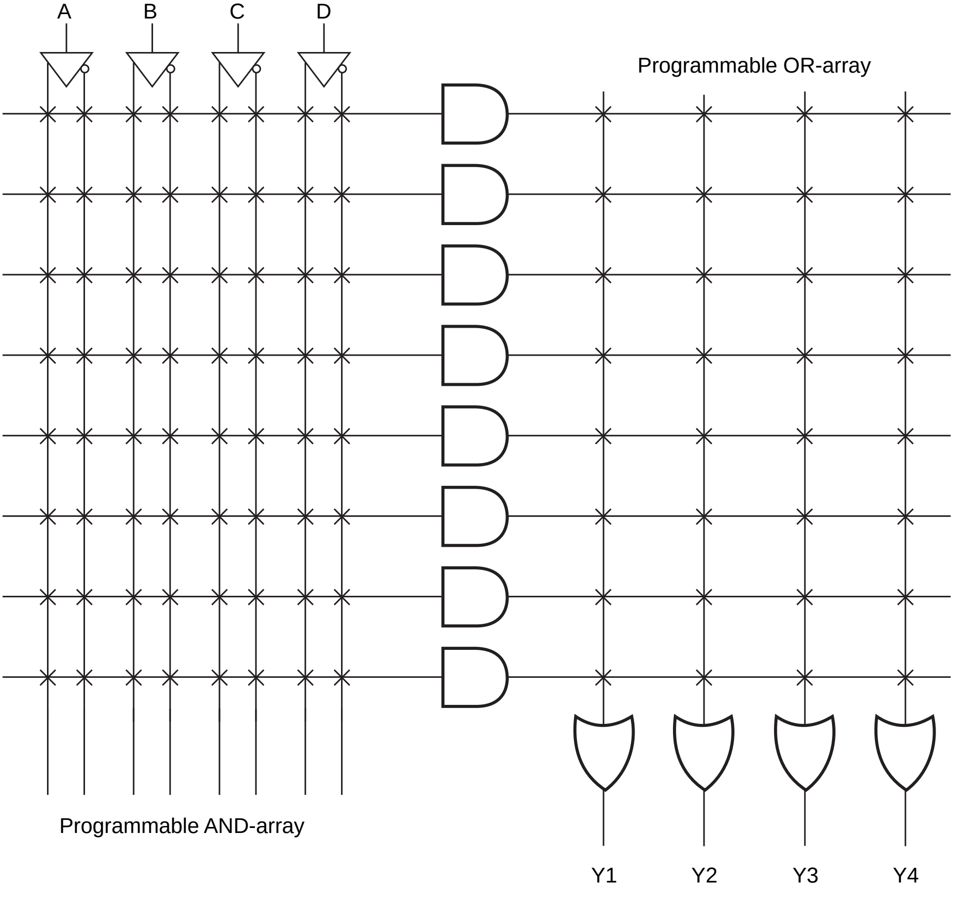


Figure 9.11 Internal architecture of a PLA device.

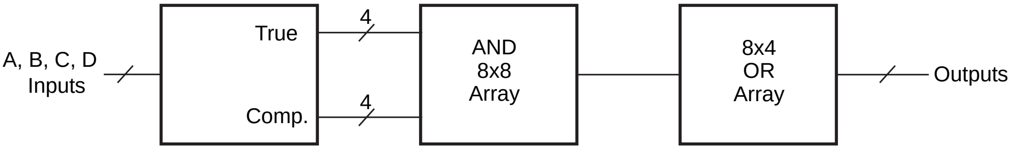


Figure 9.12 Alternative representation of PLA architecture.

While implementing a given Boolean function with a PLA, it is important that each expression is simplified to a minimum number of product terms which would minimize the number of AND gates required for the purpose. Since all input variables are available to different AND gates, simplification of Boolean functions to reduce the number of literals in various product terms is not important. In fact,

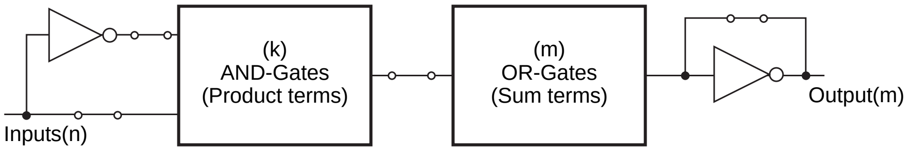


Figure 9.13 Generalized representation of PLA architecture.

each of the Boolean functions and their complements should be simplified. What is desirable is to have fewer product terms and product terms that are common to other functions. We would recall that PLAs offer the flexibility of implementing Boolean functions in both AND-OR and AND-OR-INVERT forms.

Example 9.2

Show the logic arrangement of both a PROM and a PLA required to implement a binary full adder.

Solution

The truth table of a full adder is given in Table 9.1. The Boolean expressions for sum S and carry-out C_o can be written as follows:

$$S = \Sigma 1, 2, 4, 7 \tag{9.3}$$

$$C_o = \Sigma 3, 5, 6, 7 \tag{9.4}$$

Figure 9.14 shows the implementation with an 8×2 PROM.

If we simplify the Boolean expressions for the sum and carry outputs, we will find that the expression for the sum output cannot be simplified any further, and also that the expression for carry-out can be simplified to three product terms with fewer literals. If we examine even the existing expressions, we find that we would need seven AND gates in the PLA implementation. And if we use the simplified expressions, even then we would require the same number of AND gates. Therefore, the simplification here would not help as far as its implementation with a PLA is concerned. Figure 9.15 shows the implementation of a full adder with a PLA device.

Table 9.1 Truth table for example 9.2.

A	B	Carry-in (C_i)	Sum (S)	Carry-out (C_o)
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

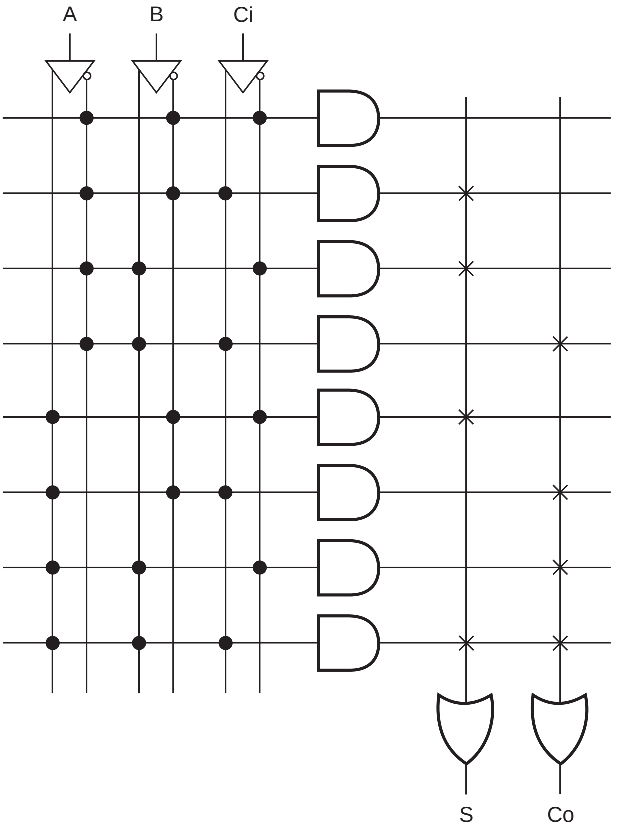


Figure 9.14 Solution to problem 9.2 using a PROM.

Example 9.3

We have two two-bit binary numbers A_1A_0 and B_1B_0. Design a PLA device to implement a magnitude comparator to produce outputs for A_1A_0 being 'equal to', 'not equal to', 'less than' and 'greater than' B_1B_0.

Solution

Table 9.2 shows the function table with inputs and desired outputs. The Boolean expressions for the desired outputs are given in the following equations:

$$\text{Output 1(equal to)} = \overline{A_1}.\overline{A_0}.\overline{B_1}.\overline{B_0} + \overline{A_1}.A_0.\overline{B_1}.B_0 + A_1.A_0.B_1.B_0 + A_1.\overline{A_0}.B_1.\overline{B_0} \tag{9.5}$$

Output 2 (not equal to)

$$\begin{aligned} &= \overline{A_1}.\overline{A_0}.\overline{B_1}.B_0 + \overline{A_1}.\overline{A_0}.B_1.\overline{B_0} + \overline{A_1}.\overline{A_0}.B_1.B_0 + \overline{A_1}.A_0.\overline{B_1}.\overline{B_0} + \overline{A_1}.A_0.B_1.\overline{B_0} + \overline{A_1}.A_0.B_1.B_0 + A_1.\overline{A_0}.\overline{B_1}.\overline{B_0} \\ &\quad + A_1.\overline{A_0}.\overline{B_1}.B_0 + A_1.\overline{A_0}.B_1.B_0 + A_1.A_0.\overline{B_1}.\overline{B_0} + A_1.A_0.\overline{B_1}.B_0 + A_1.A_0.B_1.\overline{B_0} \end{aligned} \tag{9.6}$$

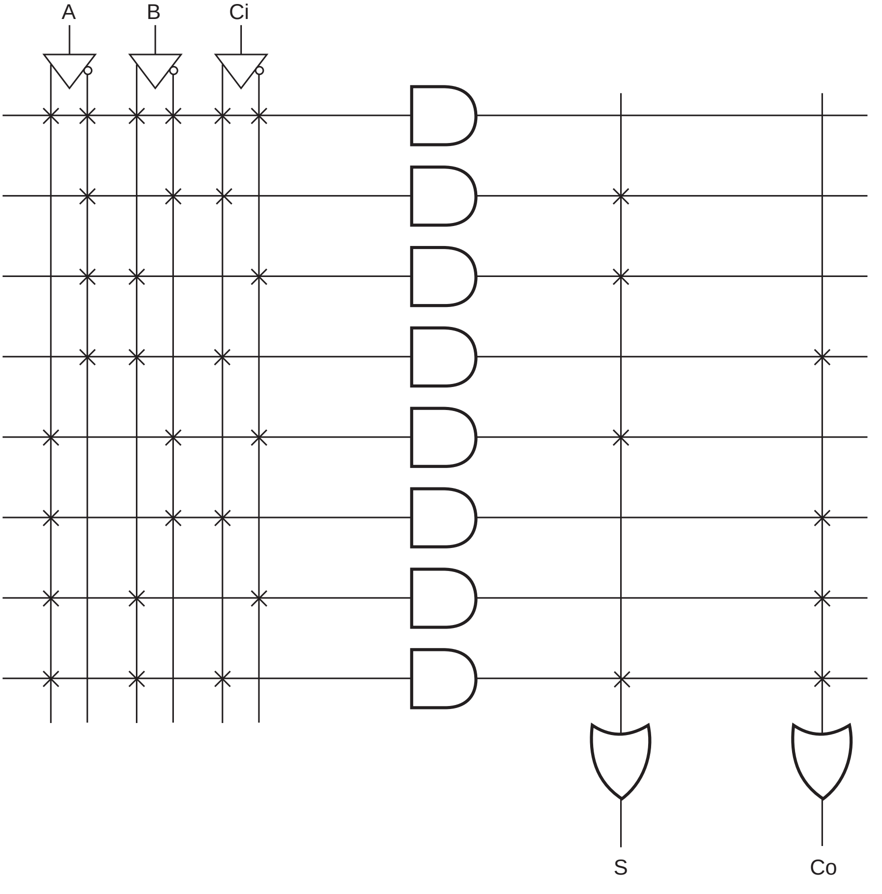


Figure 9.15 Solution to problem 9.2 using a PLA.

Output 3 (less than)

$$= \overline{A_1}.\overline{A_0}.\overline{B_1}.B_0 + \overline{A_1}.\overline{A_0}.B_1.\overline{B_0} + \overline{A_1}.\overline{A_0}.B_1.B_0 + \overline{A_1}.A_0.B_1.\overline{B_0} + \overline{A_1}.A_0.B_1.B_0 + A_1.\overline{A_0}.B_1.B_0 \quad (9.7)$$

Output 4 (greater than)

$$= \overline{A_1}.A_0.\overline{B_1}.\overline{B_0} + A_1.\overline{A_0}.\overline{B_1}.\overline{B_0} + A_1.\overline{A_0}.\overline{B_1}.B_0 + A_1.A_0.\overline{B_1}.\overline{B_0} + A_1.A_0.\overline{B_1}.B_0 + A_1.A_0.B_1.\overline{B_0} \quad (9.8)$$

Figures 9.16(a) to (d) show the Karnaugh maps for the four outputs. The minimized Boolean expressions can be written from the Karnaugh maps as follows:

$$\text{Output 1(equal to)} = \overline{A_1}.\overline{A_0}.\overline{B_1}.\overline{B_0} + \overline{A_1}.A_0.\overline{B_1}.B_0 + A_1.A_0.B_1.B_0 + A_1.\overline{A_0}.B_1.\overline{B_0} \quad (9.9)$$

$$\text{Output 2(not equal to)} = \overline{A_1}.B_1 + A_1.\overline{B_1} + \overline{A_0}.B_0 + A_0.\overline{B_0} \quad (9.10)$$

Table 9.2 Function table for example 9.3.

A_1	A_0	B_1	B_0	Output 1	Output 2	Output 3	Output 4
0	0	0	0	1	0	0	0
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	1	0	1
1	1	0	1	0	1	0	1
1	1	1	0	0	1	0	1
1	1	1	1	1	0	0	0

$$\text{Output 3(less than)} = \overline{A_1}.B_1 + \overline{A_1}.\overline{A_0}.B_0 + \overline{A_0}.B_1.B_0 \tag{9.11}$$

$$\text{Output 4(Greater than)} = A_1.\overline{B_1} + A_1.A_0.\overline{B_0} + A_0.\overline{B_1}.\overline{B_0} \tag{9.12}$$

Examination of minimized Boolean expressions (9.9) to (9.12) reveals that there are 12 different product terms to be accounted for. Therefore, a PLA device with 12 AND gates will meet the requirement. Also, since there are four outputs, we need to have four OR gates at the output. Figure 9.17 shows the programmed PLA device. Note that, in the programmed PLA device, an unprogrammed interconnection indicated by a cross ($\times$) is a ‘make’ connection.

9.5 Programmable Array Logic

The *programmable array logic* (PAL) device is a variant of the PLA device. As outlined in Section 9.2, it has a programmable AND gate array at the input and a fixed OR gate array at the output. The idea to have a fixed OR gate array at the output and make the device less complex originated from the fact that there were many applications where the product-term sharing capability of the PLA was not fully utilized and thus wasted. The PAL device is a trademark of Advanced Micro Devices Inc. PAL devices are however less flexible than PLA devices. The flexibility of a PAL device can be enhanced by having different output logic configurations including the availability of both OR (also called active HIGH) and NOR (also called active LOW) outputs and bidirectional pins that can act both as inputs and outputs, having clocked flip-flops at the outputs to provide what is called registered outputs. These features allow the device to be used in a wider range of applications than would be possible with a device with fixed input and output allocations. The mask-programmed version of PAL is known as the HAL (Hard Array Logic) device. A HAL device is pin-to-pin compatible with its PAL counterpart.

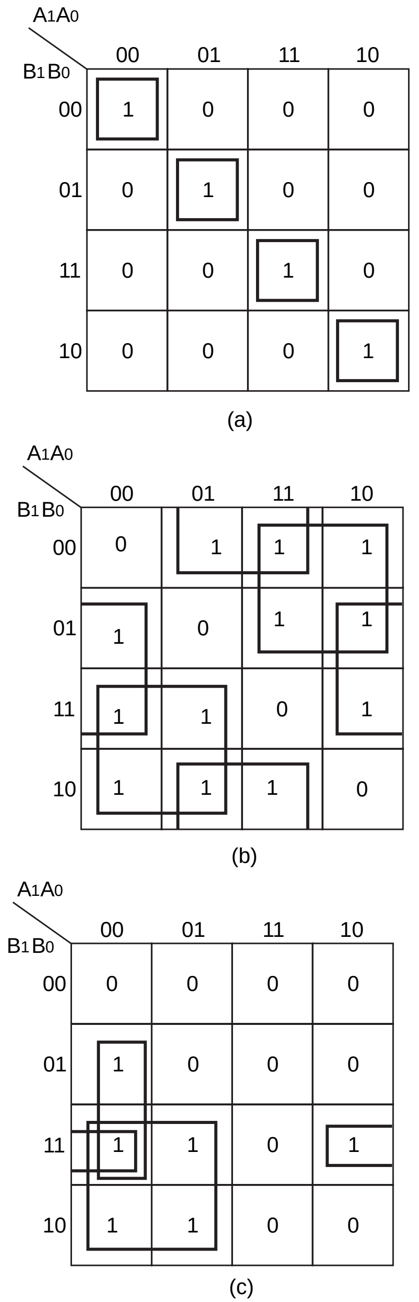


Figure 9.16 Karnaugh maps (example 9.3).

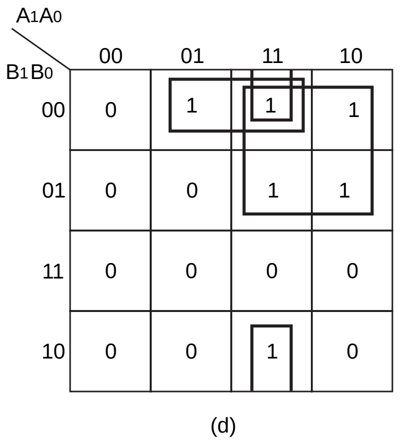


Figure 9.16 (*continued*).

9.5.1 PAL Architecture

Figure 9.18 shows the block schematic representation of the generalized architecture of a PAL device. As we can see from the arrangement shown, the device has a programmable AND gate array that is fed with various input variables and their complements. Programmable input connections allow any of the input variables or their complements to appear at the inputs of any of the AND gates in the array. Each of the AND gates generates a minterm of a user-defined combination of input variables and their complements. As an illustration, Fig. 9.19 gives an example of the generation of minterms.

Outputs from the programmable AND array feed an array of hard-wired OR gates. Here, the output of each of the AND gates does not feed the input of each of the OR gates. Each OR gate is fed from a subset of AND gates in the array. This implies that the sum-of-product Boolean functions generated by each of the OR gates at the output will have only a restricted number of minterms depending upon the number of AND gates from which it is being fed. Outputs from the PAL device, as is clear from the generalized form of representation shown in Fig. 9.18, are available both as OR outputs as well as complemented (or NOR) outputs.

Practical PAL devices offer various output logic arrangements. One of them, of course, is the availability of both OR and NOR outputs as mentioned in the previous paragraph. Another feature available with many PAL devices is that of registered outputs. In the case of registered outputs, the OR gate output drives the D-input of a *D*-type flip-flop, which is loaded with the data on either the LOW-to-HIGH or the HIGH-to-LOW edge of a clock signal. Yet another feature is the availability of bidirectional pins, which can be used both as outputs and inputs. This facility allows the user to feed a product term back to the programmable AND array. It helps particularly in those multi-output function logic circuits that share some common minterms. Some of the common output logic arrangements available with PAL devices are shown in Fig. 9.20.

Some PAL devices offer an EX-OR gate following the OR gate at each output. One of the inputs to the EX-OR gate is programmable, which allows the user to configure it as either an inverter or a noninverting buffer or as a two-input EX-OR gate. This feature is particularly useful while implementing parity and arithmetic operations.

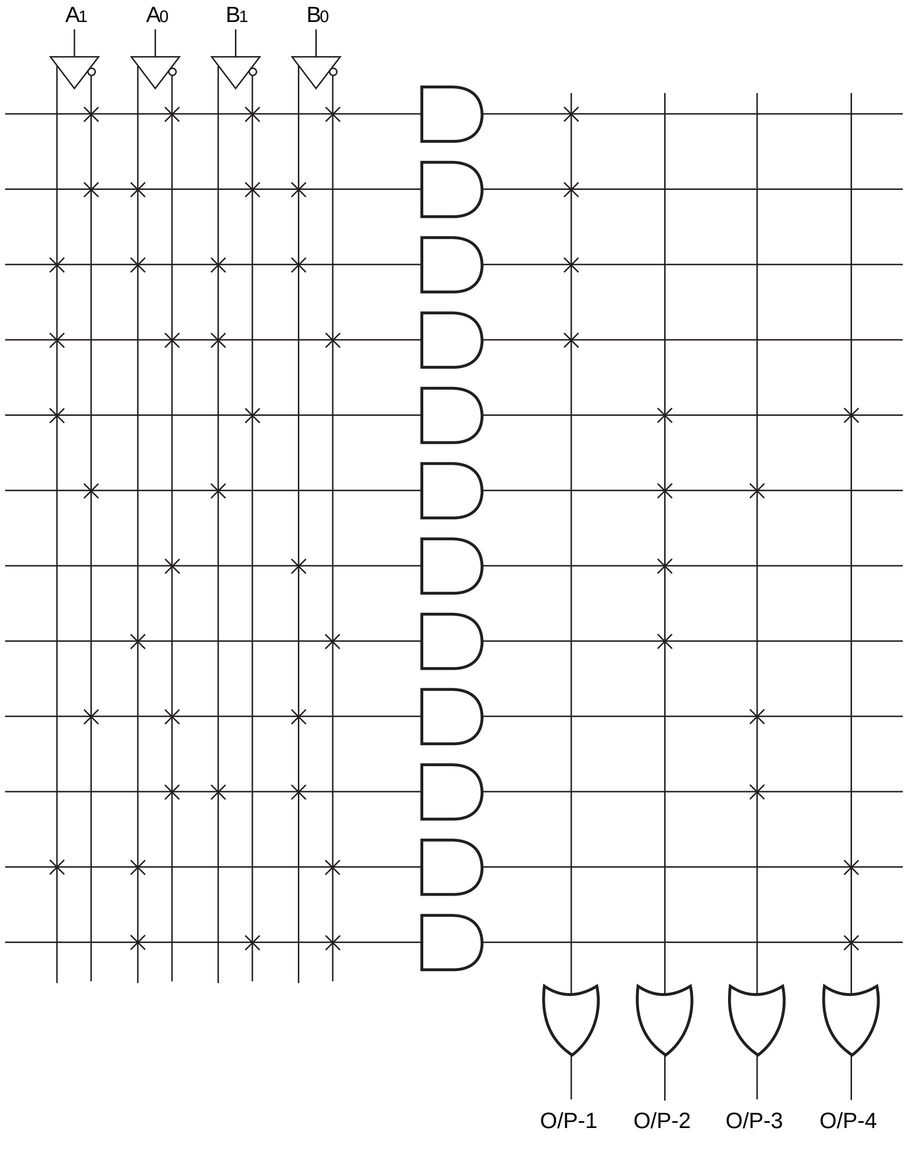


Figure 9.17 Programmed PLA device (example 9.3).

9.5.2 PAL Numbering System

The standard PAL numbering system uses an alphanumeric designation comprising a two-digit number indicating the number of inputs followed by a letter that tells about the architecture/type of logic output. Table 9.3 gives an interpretation of different letter designations in use. Another number following the

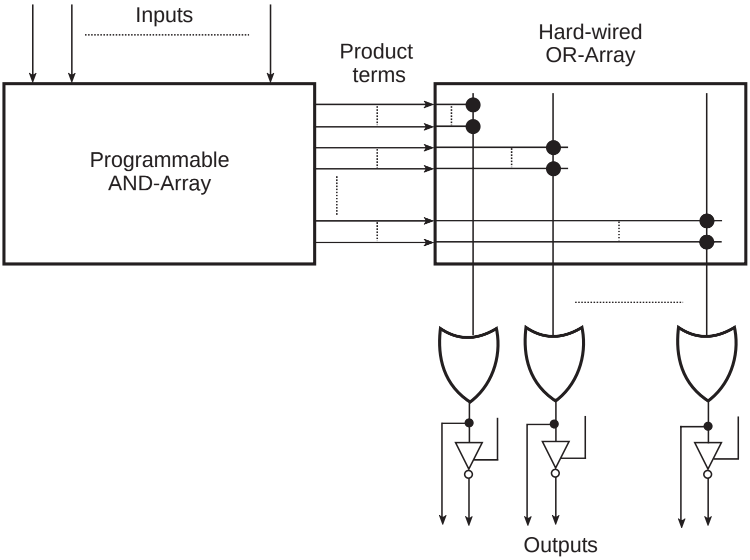


Figure 9.18 Generalized PAL device.

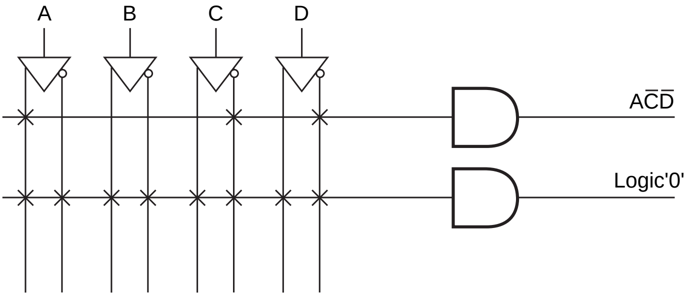


Figure 9.19 Programmability of inputs in a PAL device.

letter indicates the number of outputs. In the case of PAL devices offering a combination of different types of logic output, the rightmost number indicates the number of the output type implied by the letter used in the designation. For example, a PAL device designated PAL-16L8 will have 16 inputs and eight active LOW outputs. Another PAL device designated PAL-16R4 has 16 inputs and four registered outputs. Also, the number of inputs as given by the number designation includes dedicated inputs, user-programmable inputs accessible from combinational I/O pins and any feedback inputs

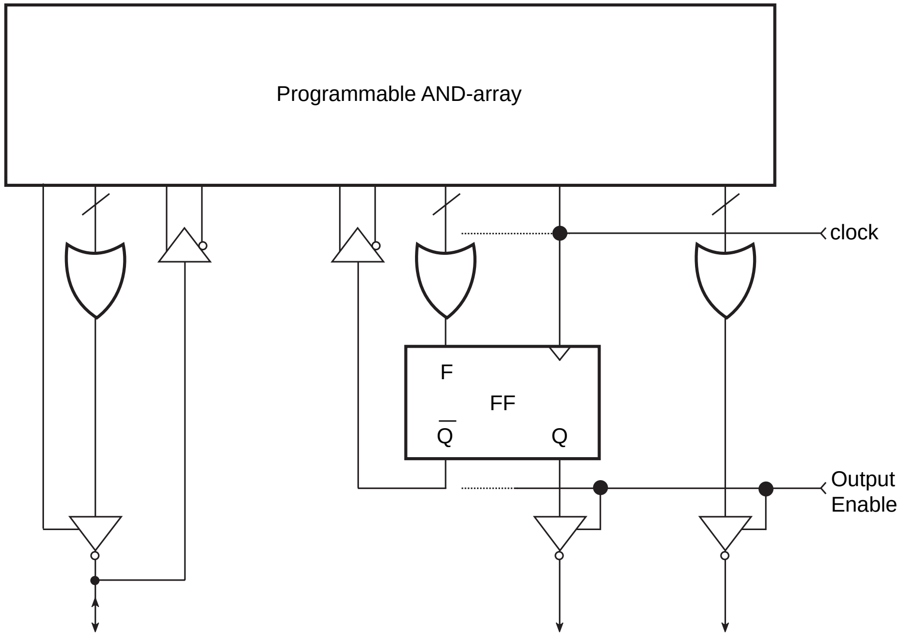


Figure 9.20 Output logic arrangements in a PAL device.

Table 9.3 PAL numbering system.

Architecture – Combinational devices		Architecture – Registered devices	
Code Letter	Description	Code letter	Description
H	Active HIGH outputs	R	Registered outputs
L	Active LOW outputs	X	EXCLUSIVE-OR gates
P	Programmable output polarity	RP	Registered polarity Programmable
C	Complementary outputs	RS	Registered-term steering
XP	EXCLUSIVE-OR gate-Programmable	V	Versatile varied Product terms
S	Product term steering	RX	Registered EX-OR
		MA	Macrocell

from combinational and registered outputs. For example, PAL-16L8 has 10 dedicated inputs and six inputs accessible from I/O pins.

In addition to the numbering system described above, an alphanumeric designation on the extreme left may be used to indicate the technology used. ‘C’ stands for CMOS, ‘10H’ for 10KH ECL and ‘100’ for 100K ECL. TTL is represented by a blank. A letter on the extreme right may be used to indicate the power level, with 'L' and 'Q' respectively indicating low and quarter power levels and a blank representing full power.

Example 9.4

Table 9.4 shows the function table of a converter. Starting with the Boolean expressions for the four outputs (P, Q, R, S), minimize them using Karnaugh maps and then hardware-implement this converter with a suitable PLD with PAL architecture.

Solution

From the given function table, we can write the Boolean expressions for the four outputs as follows:

$$P=\overline{A}.B.\overline{C}.D+\overline{A}.B.C.\overline{D}+\overline{A}.B.C.D+A.\overline{B}.\overline{C}.\overline{D}+A.\overline{B}.\overline{C}.D \tag{9.13}$$

$$Q=\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D \tag{9.14}$$

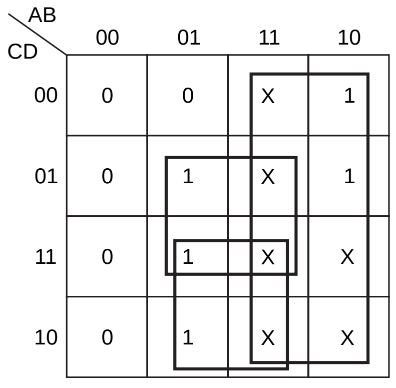
$$R=\overline{A}.\overline{B}.C.\overline{D}+\overline{A}.\overline{B}.C.D+\overline{A}.B.\overline{C}.\overline{D}+\overline{A}.B.\overline{C}.D+\overline{A}.B.C.\overline{D}+\overline{A}.B.C.D \tag{9.15}$$

$$S=\overline{A}.\overline{B}.\overline{C}.D+\overline{A}.\overline{B}.C.\overline{D}+\overline{A}.B.C.D+A.\overline{B}.\overline{C}.\overline{D} \tag{9.16}$$

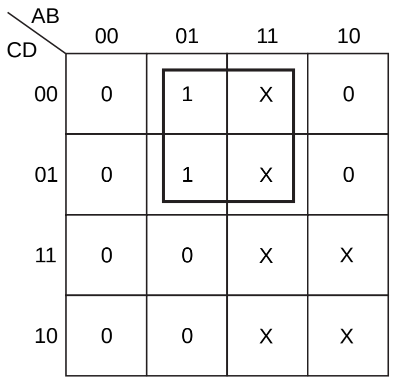
Karnaugh maps for the four outputs P,Q,R and S are respectively shown in Figs 9.21(a) to (d). The minimized Boolean expressions are given by the equations

Table 9.4 Function table in example 9.4.

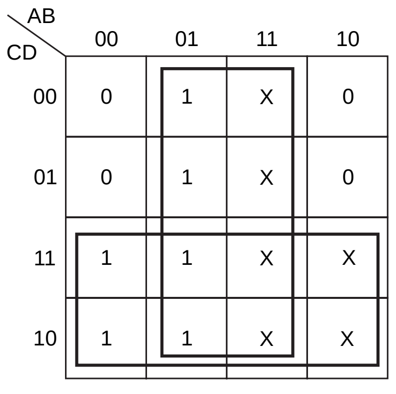
A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X



(a)

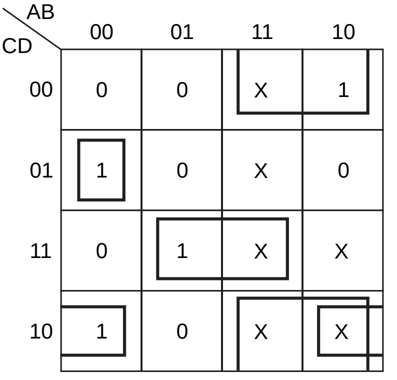


(b)



(c)

Figure 9.21 Karnaugh maps (example 9.4).



(d)

Figure 9.21 (*continued*).

$$P = B.D + B.C + A \tag{9.17}$$

$$Q = B.\overline{C} \tag{9.18}$$

$$R = B + C \tag{9.19}$$

$$S = \overline{A}.\overline{B}.\overline{C}.D + B.C.D + A.\overline{D} + \overline{B}.C.\overline{D} \tag{9.20}$$

The next step is to choose a suitable PAL device. Since there are four output functions, we will need a PAL device with at least four OR gates at the output. Since each of the OR gates is to be hard wired to only a subset of programmable AND arrays, and also because one of the output functions has four product terms, we will need an AND array of 16 AND gates. Since there are four input variables, we need each AND gate in the array to have eight inputs to cater for four variables and their complements. To sum up, we choose a PAL device that has eight inputs, 16 AND gates in the programmable AND array and four OR gates at the output. Each OR gate has four inputs.

Figure 9.22 shows the architecture of the programmed PAL device. We can see that the P output has only three product terms. The fourth input to the relevant OR gate needs to be applied a logic '0' input. This is achieved by feeding the inputs of the corresponding AND gate with all four variables and their complements. Logic 0s, wherever required, are implemented in the same manner. Note that, in the programmed PAL device of Fig. 9.22, an unprogrammed interconnection indicated by a cross ($\times$) is a 'make' connection.

9.6 Generic Array Logic

Generic array logic (GAL) is characterized by a reprogrammable AND array, a fixed OR array and a reprogrammable output logic. It is similar to a PAL device, with the difference that the AND

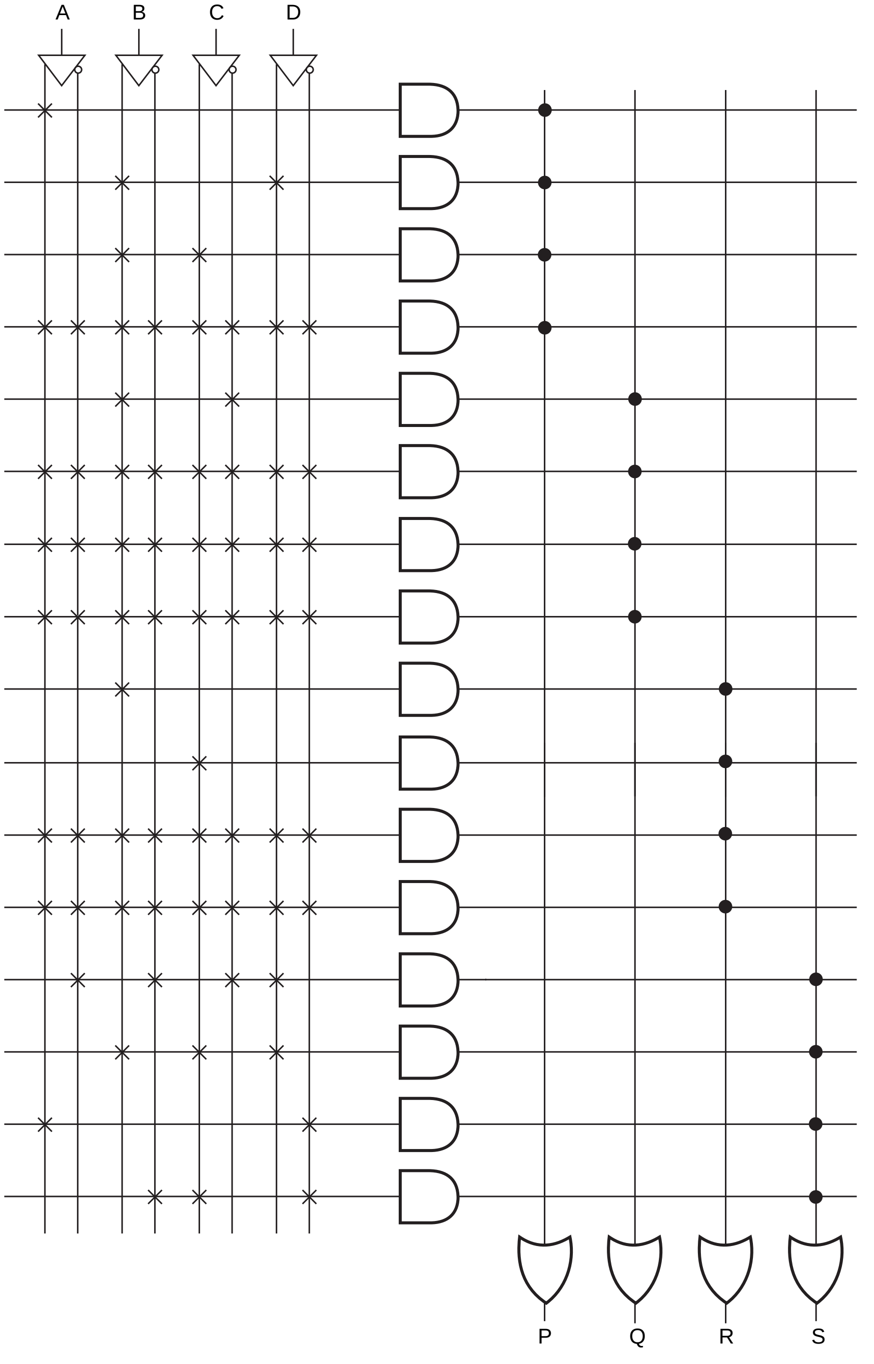


Figure 9.22 Programmed PAL (example 9.4).

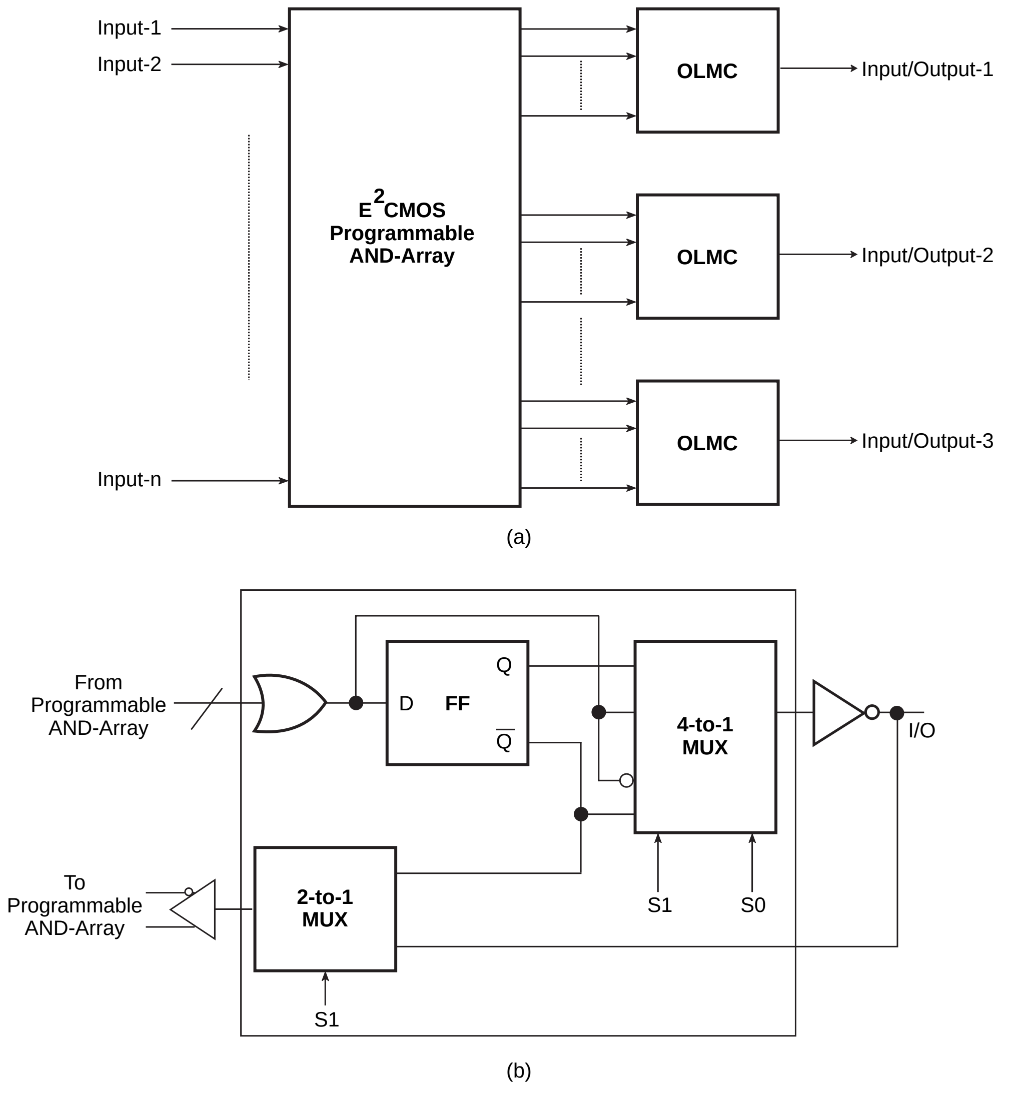


Figure 9.23 (a) Generic array logic generalized block schematic and (b) architecture of an OLMC.

array is not just programmable as is the case in a PAL device but is reprogrammable. That is, it can be reprogrammed any number of times. This has been made possible by the use of electrically erasable PROM cells for storing the programming pattern. The other difference is in the use of reprogrammable output logic, which provides more flexibility to the designer. GAL devices employ output logic macrocells (OLMCs) at the output, which allows the designer to configure the outputs either as combinational outputs or registered outputs.

Figures 9.23(a) and (b) respectively show the block schematic representation of a GAL device and the architecture of a typical OLMC used with GAL devices. The OLMC of the type shown in Fig. 9.23(b) can be configured to produce four different outputs depending upon the selection inputs. These include the following:

1. $S_1S_0 = 00$: registered mode with active LOW output.
2. $S_1S_0 = 01$: registered mode with active HIGH output.
3. $S_1S_0 = 10$: combinational mode with active LOW output.
4. $S_1S_0 = 11$: combinational mode with active HIGH output.

We can see that two of the four inputs to the 4-to-1 multiplexer are combinational outputs, and the other two are the registered outputs. Also, of the two combinational outputs, one is an active HIGH output while the other is an active LOW output. The same is the case with registered outputs. Of the four inputs to the multiplexer, the one appearing at the output depends upon selection inputs. The 2-to-1 multiplexer ensures that the final output is also available as feedback to the programmable AND array.

9.7 Complex Programmable Logic Devices

If we examine the internal architecture of simple programmable logic devices (SPLDs) such as PLAs and PALs, we find that it is not practical to increase their complexity beyond a certain level. This is because the size of the programmable plane (such as the programmable AND plane in a PLA or PAL device) increases too rapidly with increase in the number of inputs to make it a practically viable device. One way to increase the logic capacity of simple programmable logic devices is to integrate multiple SPLDs on a single chip with a programmable interconnect between them. These devices have the same basic internal structure that we see in the case of SPLDs and are grouped together in the category of complex programmable logic devices (CPLDs). Typically, CPLDs may offer a logic capacity equivalent to that of about 50 SPLDs. Programmable logic devices with much higher logic capacities would require a different approach rather than simple extension of the concept of SPLDs.

9.7.1 Internal Architecture

As outlined in the previous paragraph, a CPLD is nothing but the integration of multiple PLDs, a programmable interconnect matrix and an I/O control block on a single chip. Each of the identical PLDs is referred to as a *logic block* or *function block*. Figure 9.24 shows the architecture of a typical CPLD. As is evident from the block schematic arrangement, the programmable interconnect matrix is capable of connecting the input or output of any of the logic blocks to any other logic block. Also, input and output pins connect directly to both the interconnect matrix as well as logic blocks.

Logic blocks may further comprise smaller logic units called macrocells, where each of the macrocells is a subset of a PLD-like logic block. Figure 9.25 shows the structure of a logic block along with its interconnections with the programmable interconnect matrix and I/O block. The horizontal grey-coloured bars inside the logic block constitute an array of macrocells. Typically, each macrocell comprises a set of product terms generated by a subset of the programmable AND array and feeding a configurable output logic. The output logic typically comprises an OR gate, an EX-OR gate and a flip-flop. The flip-flop in the case of most contemporary CPLDs is configurable as a *D*-type, *J-K*, *T*, or *R-S* flip-flop or can even be transparent. Also, the OR gate can be fed with any or all of the product terms generated within the macrocell. Most contemporary CPLDs also offer an architecture where the

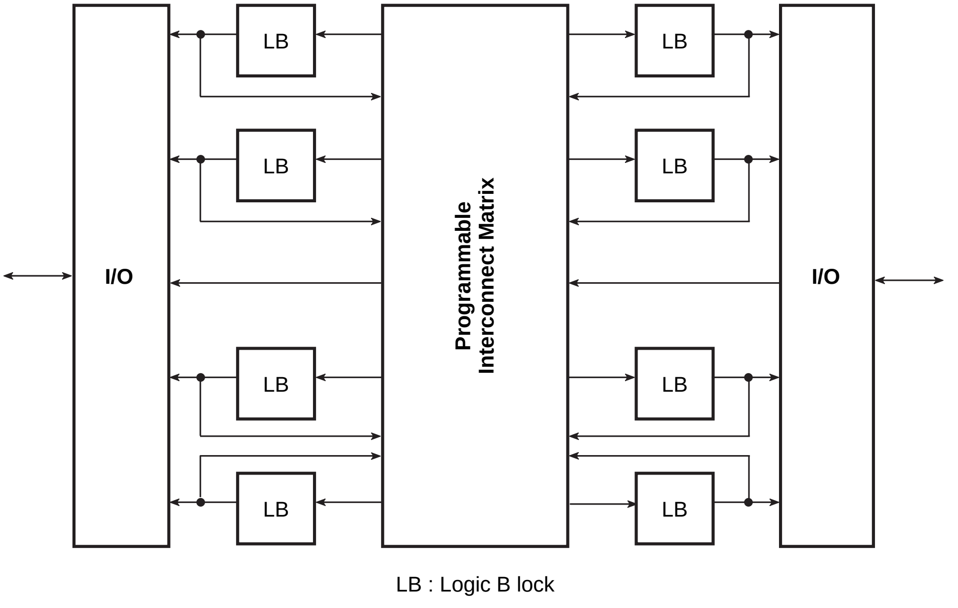


Figure 9.24 CPLD architecture.

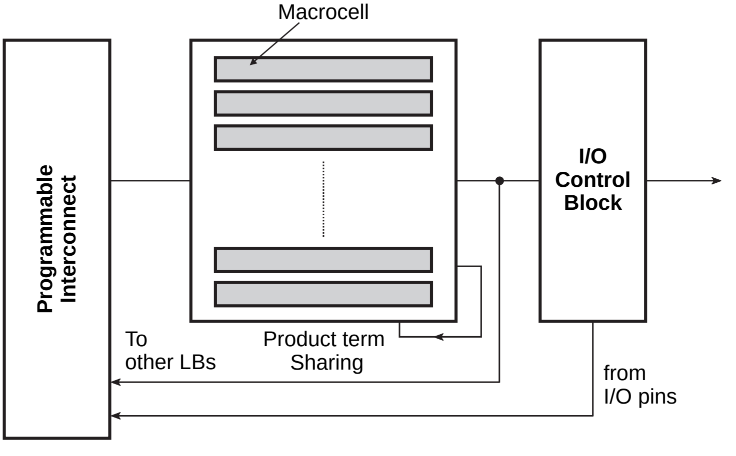


Figure 9.25 Logic block structure.

OR gate can also be fed with some additional product terms generated within other macrocells of the same logic block. For example, a logic block in the case of the MAX-7000 series of CPLDs from Altera offers this product-term flexibility, where the OR gate of each macrocell can have up to 15

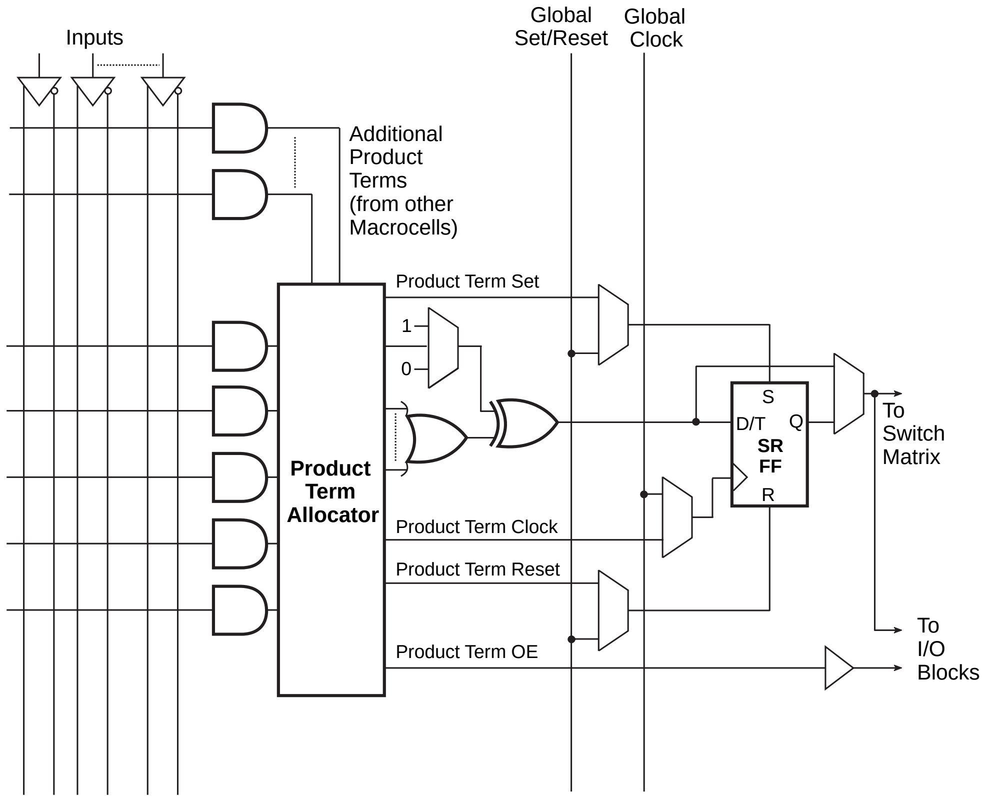


Figure 9.26 Macrocell architecture.

additional product terms from other macrocells in the same logic block, apart from a maximum of five product terms from within the same macrocell.

Figure 9.26 shows the logic diagram of a macrocell typical of macrocells in the logic blocks of most contemporary CPLDs. The diagram is self-explanatory. There may be minor variations in devices from different manufacturers. For example, macrocells in the XC-7000 series CPLDs from Xilinx have two OR gates fed from a two-bit arithmetic logic unit (ALU) and its output feeds a configurable flip-flop.

9.7.2 *Applications*

Owing to their less flexible internal architecture leading to predictable timing performance, high speed and a range of logic capacities, CPLDs find extensive use in a wide assortment of applications. These include the implementation of random glue logic in prototyping small gate arrays, implementing critical control designs such as graphics controllers, cache control, UARTs, LAN controllers and many more.

CPLDs are fast replacing SPLDs in complex designs. Complex designs using a large number of SPLDs can be replaced with a CPLD-based design with a much smaller number of devices. This is particularly attractive in portable applications such as mobile phones, digital assistants and so on.

CPLD architecture particularly suits those designs that exploit wide AND/OR gates and do not require a large number of flip-flops.

The reprogramming feature of CPLDs makes the incorporation of design changes very easy. With the availability of CPLDs having an in-circuit programming feature, it is even possible to reconfigure the hardware without power down. Changing protocol in a communication circuit could be one such example. One of the most significant advantages of CPLD architecture comes from its simple SPLD-like structure, which allows the design to partition naturally into SPLD-like blocks. This leads to a much more predictable timing or speed performance than would be possible if the design were split into many pieces and mapped into different areas of the chip.

9.8 Field-Programmable Gate Arrays

As outlined earlier, it is not practical to increase the logic capacity with a CPLD architecture beyond a certain point. The highest-capacity general-purpose logic chips available today are the traditional gate arrays, which comprise an array of prefabricated transistors. The chip can be customized during fabrication as per the user's logic design by specifying the metal interconnect pattern. These chips are also referred to as *mask-programmable gate arrays* (MPGAs). These, however, are not field-programmable devices. A field-programmable gate array (FPGA) chip is the user-programmable equivalent of an MPGA chip.

9.8.1 Internal Architecture

An FPGA consists of an array of uncommitted configurable logic blocks, programmable interconnects and I/O blocks. The basic architecture of an FPGA was shown earlier in Fig. 9.7 when presenting an overview of programmable logic devices. As outlined earlier, the basic difference between a CPLD and an FPGA lies in their internal architecture. CPLD architecture is dominated by a relatively smaller number of programmable sum-of-products logic arrays feeding a small number of clocked flip-flops, which makes the architecture less flexible but with more predictable timing characteristics. On the other hand, FPGA architecture is dominated by programmable interconnects, and the configurable logic blocks are relatively simpler. Logic blocks within an FPGA can be as small as the macrocells in a PLD, called fine-grained architecture, or larger and more complex, called coarse-grained architecture. However, they are never as large as the entire PLD like the logic blocks of a CPLD. This feature makes these devices far more flexible in terms of the range of designs that can be implemented with these devices.

Contemporary FPGAs have an on-chip presence of higher-level embedded functions and embedded memories. Some of them even come with an on-chip microprocessor and related peripherals to constitute what is called a complete 'system on a programmable chip'. Virtex-II Pro and Virtex-4 FPGA devices from Xilinx are examples. These devices have one or more PowerPC processors embedded within the FPGA logic fabric.

Figure 9.27 shows a typical logic block of an FPGA. It consists of a four-input look-up table (LUT) whose output feeds a clocked flip-flop. The output can either be a registered output or an unregistered LUT output. Selection of the output takes place in the multiplexer. An LUT is nothing but a small one-bit wide memory array with its address lines representing the inputs to the logic block and a one-bit output acting as the LUT output. An LUT with n inputs can realize any logic function of n inputs by programming the truth table of the desired logic function directly into the memory.

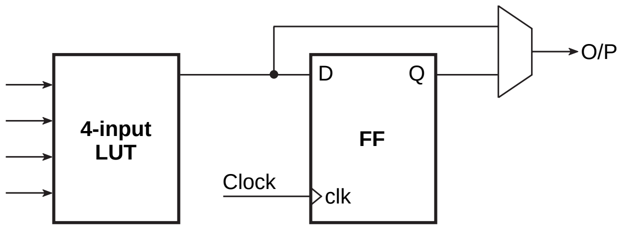


Figure 9.27 Logic block of a typical FPGA.

Logic blocks can have more than one LUT and flip-flops also to give them the capability of realizing more complex logic functions. Figure 9.28 shows the architecture of one such logic block. The architecture shown in Fig. 9.28 is that of a logic block of the XC4000 series of FPGAs from Xilinx. This logic block has two four-input LUTs fed with logic block inputs and a third LUT that can be used in conjunction with the two LUTs to offer a wide range of functions. These include two separate logic functions of four inputs each, a single logic function of up to nine inputs and many more. The logic block contains two flip-flops.

Figure 9.29 shows another similar LUT-based architecture that uses multiple LUTs and flip-flops. The architecture shown in Fig. 9.29 is that of a logic block called a programmable function unit

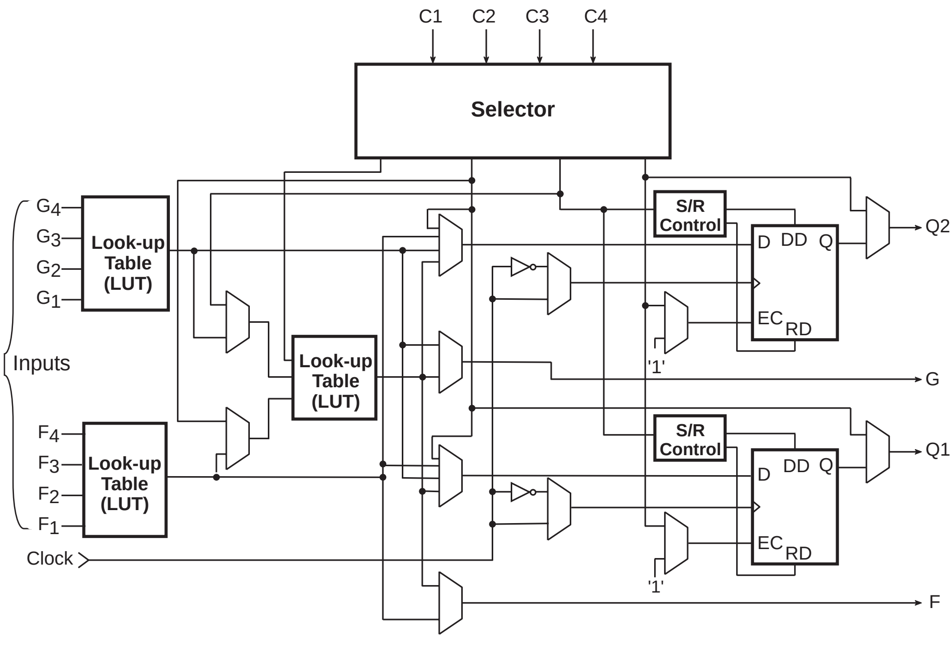


Figure 9.28 Logic block architecture of the XC4000 FPGA from Xilinx.

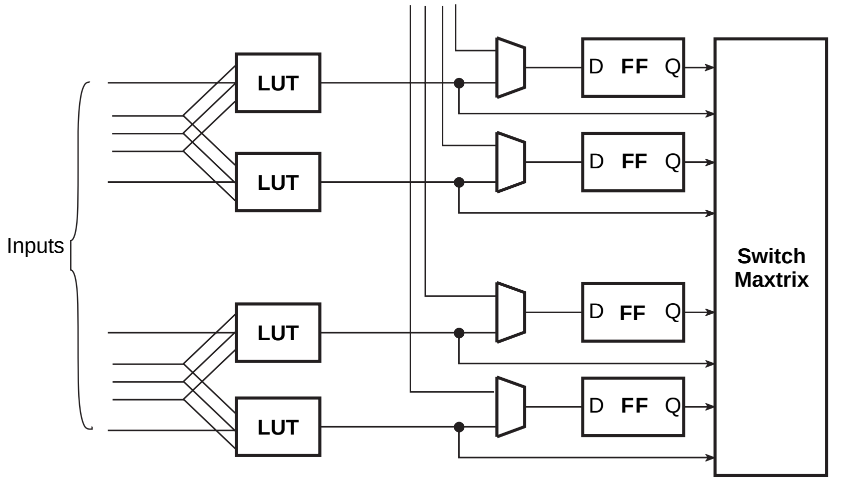


Figure 9.29 Logic block architecture of an AT&T FPGA.

(PFU) by the manufacturer of AT&T FPGA devices. This logic block can be configured either as four four-input LUTs or two five-input LUTs or one six-input LUT.

9.8.2 Applications

In the early days of their arrival on the scene, FPGAs began as competitors to CPLDs for applications such as glue logic for PCBs. With increase in their logic capacity and capability, the availability of a large embedded memory, higher-level embedded functions such as adders and multipliers, the emergence of hybrid technologies combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and the facility of full or partial in-system reconfiguration have immensely widened the scope of applications of FPGAs. FPGAs today offer a complete system solution on a single chip, although very complex systems might be implemented with more than one FPGA device.

Some of the major application areas of FPGA devices include digital signal processing, data storage and processing, software-defined radio, ASIC prototyping, speech recognition, computer vision, cryptography, medical imaging, defence systems, bioinformatics, computer hardware emulation and reconfigurable computing. Reconfigurable computing, also called customized computing, involves the use of programmable parts to execute software rather than compiling the software to be run on a regular CPU. This has been made possible by in-system reconfiguration, which allows the internal design to be altered on-the-fly.

9.9 Programmable Interconnect Technologies

The programmable features of every PLD, be it simple programmable logic devices (SPLDs) such as PLAs, PALs and GALs or complex programmable logic devices (CPLDs) or even field-programmable gate arrays (FPGAs), come from their programmable interconnect structure. Interconnect technologies that have evolved over the years for programming PLDs include fuses, EPROM or EEPROM floating-gate transistors, static RAM and antifuses.

Each one of these is briefly described in the following paragraphs.

9.9.1 Fuse

A fuse is an electrical device that has a low initial resistance and is designed permanently to break an electrically conducting path when current through it exceeds a specified limit. It uses bipolar technology and is nonvolatile and one-time programmable. It was the first user-programmable switch developed for use in PLAs. They were earlier used in smaller PLDs and are now being rapidly replaced by newer technologies.

9.9.2 Floating-Gate Transistor Switch

This interconnect technology is based on the principle of placing a floating-gate transistor between two wires in such a way as to facilitate a WIRE-AND function. This concept is used in EPROM and EEPROM devices, and that is why the floating-gate transistor is sometimes referred to as an EPROM or EEPROM transistor. Figure 9.30 shows the use of floating-gate transistor interconnects in the AND plane of a CPLD or SPLD. All those inputs that are required to be part of a particular product term are activated to drive the product wire to a logic '0' level through the EPROM transistor. For inputs that are not part of the product term, relevant transistors are switched off.

This technology is commonly used in SPLDs and CPLDs. A floating-gate transistor based switch matrix, however, requires a large number of interconnects and therefore transistors. For example, a CPLD with 128 macrocells with four inputs and one output each would require as many as 65 536 interconnects for 100 % routability. A large number of interconnects also adds to the propagation delay.

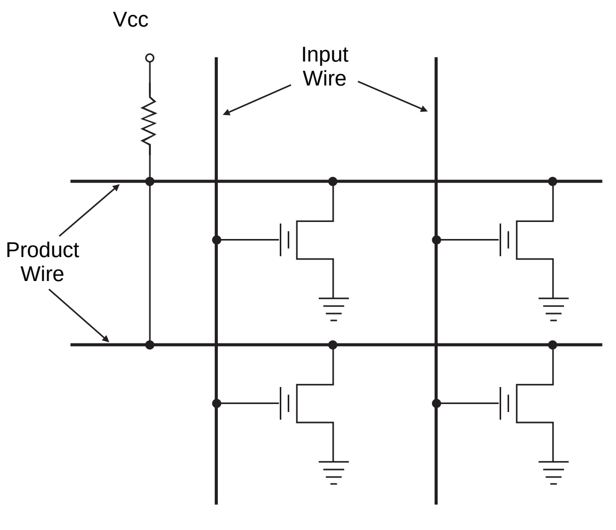


Figure 9.30 Floating-gate transistor interconnect.

The use of multiplexers can reduce this number significantly and can also address the problem of increased propagation delay. An MUX-based interconnect matrix is being used in CPLDs. CPLD type XPLA3 from Xilinx is an example.

9.9.3 Static RAM-Controlled Programmable Switches

Static RAM (SRAM) is basically a semiconductor memory, and the word 'static' implies that it is a nonvolatile memory. That is, the memory retains its contents as long as power is on. A SRAM with m address lines and n data lines is referred to as a $2^m \times n$ memory and is capable of storing 2^m n-bit words. Figure 9.31 shows the basic SRAM cell comprising six MOSFET switches, with four of them connected as cross-coupled inverters. A basic SRAM cell can store one bit of information. The reading operation is carried out by precharging both the bit lines (BL and $\overline{BL}$) to logic '1' and then asserting the WL line. The writing operation is done by giving the desired logic status to the BL line and its complement to the $\overline{BL}$ line and then asserting the WL line.

Figure 9.32 shows the use of SRAM-controlled switches. SRAMs are used to control not only the gate nodes but also the select inputs of multiplexers that drive the logic block inputs. The figure illustrates the routing scheme for feeding the output of one logic block to the input of another via SRAM-controlled pass transistor switches and a SRAM-controlled multiplexer. It may be mentioned here that a SRAM-controlled programmable interconnect matrix does not necessarily use both pass transistors and multiplexers. Whether it uses pass transistors or multiplexers or both is product specific.

9.9.4 Antifuse

An antifuse is an electrical device with a high initial resistance and is designed permanently to create an electrically conducting path typically when voltage across it exceeds a certain level. Antifuses

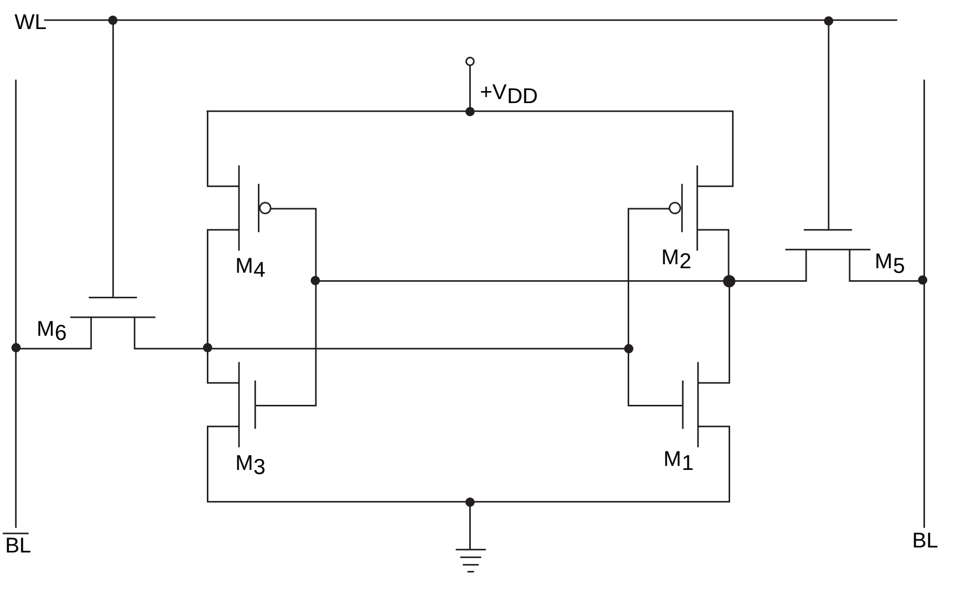


Figure 9.31 SRAM cell.

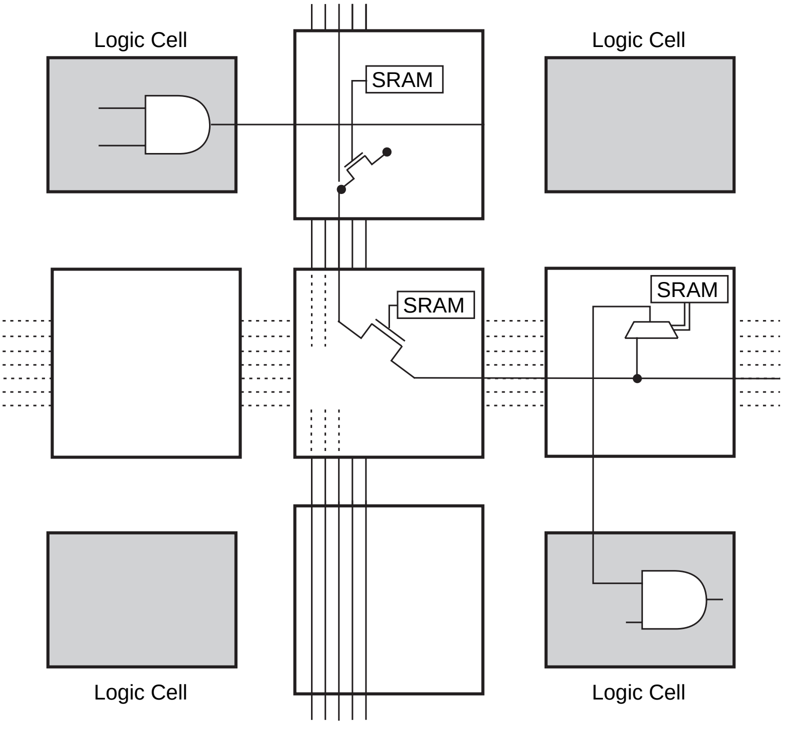
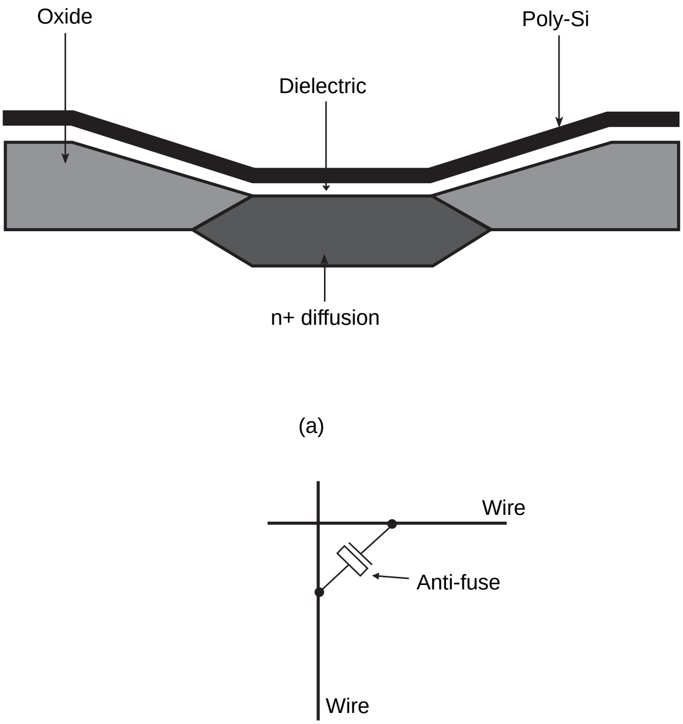


Figure 9.32 SRAM-controlled interconnect.

use CMOS technology, which is one of the main reasons for their wide use in PLDs, FPGAs in particular. A typical antifuse consists of an insulating layer sandwiched between two conducting layers. In the unprogrammed state, the insulating layer isolates the top and bottom conducting layers. When programmed, the insulating layer is transformed into a low-resistance link. Typically, metal is used for conductors and amorphous silicon for the insulator. The application of high voltage across amorphous silicon permanently transforms it into a polycrystalline silicon–metal alloy having a low resistance. There are other antifuse structures too, such as that used in the Actel antifuse. This antifuse, known as PLICE, uses polysilicon and n+ diffusion as conductors and ONO as insulator. Figure 9.33(a) shows the construction. This type of antifuse is usually triggered by a small current of the order of a few milliamperes. The high current density produced in the thin insulating layer produces heat, thus melting the insulating layer and creating an irreversible resistive silicon link.

Antifuses are widely used as programmable interconnects in PLDs [Fig. 9.33(b)]. Antifuse PLDs are one-time programmable, in contrast to SRAM-controlled interconnect-based PLDs, which are reprogrammable. It may be mentioned here that the reprogrammable feature helps the designers fix logic bugs or add new functions. Antifuse PLDs have advantages of nonvolatility and usually higher speeds. Antifuses may also be used in PROMs. In that case, each bit contains both a fuse and an antifuse. The device is programmed by triggering one of the two.



(b)

Figure 9.33 (a) Actel's antifuse and (b) the antifuse as a programmable interconnect.

9.10 Design and Development of Programmable Logic Hardware

In this section, we will briefly discuss the various steps involved in the design and development of programmable logic hardware. Figure 9.34 shows a block diagram representation of the sequence of steps involved, in the order in which they are executed.

The process begins with a description of behavioural aspects and the architecture of the intended hardware. This is done by writing a source code in a high-level *hardware description language* (HDL) such as VHDL or Verilog. This step is known as *design entry*. Although schematic capture is also an option for design entry, it has been replaced with language-based tools owing to the designs becoming more and more complex, and also owing to advances in language-based tools.

The most important difference between a hardware and software design is as follows. While software developers tend to think sequentially, hardware designers must think and program in parallel. All input signals are processed in parallel as they travel through a series of macrocells and associated interconnects towards their destination. As a result, statements of HDL create structures, which are executed at the same time. It may be mentioned here that the transfer of information from macrocell to macrocell is synchronized to another signal such as a clock.

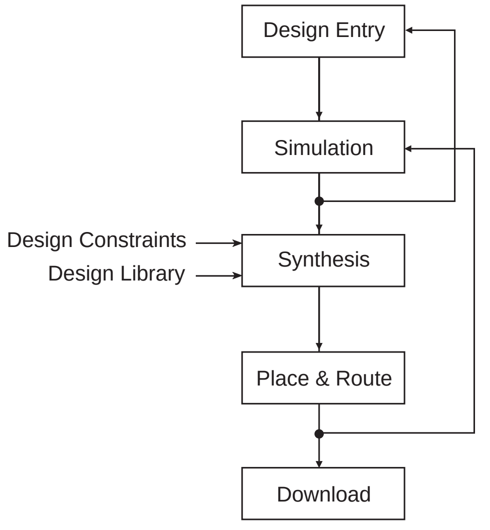


Figure 9.34 Programmable logic design and development process.

The design entry step is either followed by or interspersed with periodic functional *simulation*. The simulator executes the design for a given set of inputs and confirms that the logic is functionally correct.

Hardware compilation comes next. It involves two steps. The first step is *synthesis*, and the result of that is a hardware representation called a netlist. The netlist is device independent and its contents do not depend on the parameters of the PLD to be programmed. It is usually stored in a standard format called the electronic design interchange format (EDIF). The second step, called *place and route*, involves mapping of the logical structure described in the netlist onto actual logic blocks, interconnects and inputs/outputs. The place and route process produces a bit stream, which is nothing but the binary data that must be loaded into CPLD/FPGA to make the chip execute the intended hardware design. It may be mentioned here that each device family has its own proprietary bit stream format.

9.11 Programming Languages

During the PLD development cycle, from design entry to the generation of a bit stream that can be loaded onto the chip using some kind of electronic programming system, two types of software program are needed to perform two different functions.

The first is a *hardware description language* (HDL), which is needed at the design entry stage. HDL is a software programming language that is used to model or describe the intended operation of a piece of hardware. In the present case, this is the function that the PLD chip is intended to perform after it is programmed. It may be worth mentioning here that modern computer languages, including both hardware description languages and high-level programming languages, almost invariably contain declarative and executable statements, and the hardware description languages are particularly rich in the former. If we compare the results of a high-level programming language such as C++ and an HDL, it will be an executable program in the case of the former and declarative in the case of the latter. Hardware description languages that have evolved over the years include ABEL-HDL, VHDL (VHSIC HDL), Verilog and JHDL (Java HDL). VHSIC stands for Very High-Speed Integrated Circuit.

The second type of software program is a computer program, called a logic compiler, that is used to transform a source code written in HDL into a bit stream. Logic compilers are available from manufacturers or third-party vendors. In the paragraphs to follow, we will briefly describe each of the hardware description languages mentioned above.

9.11.1 ABEL-Hardware Description Language

ABEL-HDL from DATA I/O was intended for relatively simpler PLD circuit designs that could be implemented on SPLDs. ABEL allows the designers to describe the digital circuit designs expressed in the form of truth tables, Boolean functions, state diagrams or any combination of these. It also allows the designer to optimize the design through design validation without specifying a device. In other words, ABEL-HDL facilitates writing hardware-independent programs, and it is only after the design verification and optimization have taken place that the PLD device is chosen. The source code written in the ABEL environment is in standard format to have interface compatibility with other tools.

9.11.2 VHDL-VHSIC Hardware Description Language

VHDL is the most widely used hardware description language used for the purpose of describing complex digital circuit designs that would be implemented on CPLDs and FPGAs. VHDL was originally developed to document the behaviour of ASICs used by various manufacturers in their equipment. It was subsequently followed by the development of logic simulation and synthesis tools that could read VHDL files and output a definition of the physical implementation of the circuit. With modern synthesis tools capable of extracting various digital building blocks such as counters, RAMs, arithmetic blocks, etc., and implementing them as specified by the user, the same VHDL code could be synthesized differently for optimum performance.

VHDL is a strongly typed language. One of the key features of VHDL is that it allows the behaviour of the intended hardware to be described and then verified before the design is translated into actual hardware with the help of synthesis tools. Another feature of VHDL that makes it attractive for digital system design is that it allows description of a concurrent system.

9.11.3 Verilog

Verilog, like VHDL, supports design, design validation and subsequent implementation of analogue, digital and mixed signal circuits at various levels of abstraction. Verilog-based design consists of a hierarchy of modules whose behaviour is defined by concurrent and sequential statements. Sequential statements are placed inside a 'begin/end' block and sequential statements contained inside the block are executed sequentially. All concurrent statements and all 'begin/end' blocks in the design are executed in parallel. A subset of statements in Verilog is synthesizable. Therefore, if in a given design the different modules use only synthesizable statements, the design can be translated into a netlist, which can further be translated into a bit stream.

Verilog has some similarities and dissimilarities with C-language. It has a similar preprocessor, similar major control keywords like 'if', 'while', etc., and also a similar formatting mechanism in the printing routines and language operators. Dissimilarities include the use of 'begin/end' instead of curly braces to define a block of code, and also that Verilog does not have structures, pointers and recursive subroutines. Also, the definition of constants in Verilog requires bit width along with their base.

9.11.4 Java HDL

Java HDL (JHDL) was developed in the Configurable Computing Laboratory of Brigham Young University (BYU). It is a low-level hardware description language that primarily uses an object-oriented approach to build circuits. It was developed primarily for the design of FPGA-based hardware, and developers have paid particular attention to supporting the Xilinx series of FPGA chips.

9.12 Application Information on PLDs

In this section, we will look at salient features of some of the commonly used programmable logic devices including SPLDs such as PALs/GALs, CPLDs and FPGAs covering a wide spectrum of devices from leading international manufacturers. Other application-relevant information such as internal architecture, pin connection diagram, etc., is also given for some of the more popular type numbers.

9.12.1 SPLDs

Some of the famous companies that offer SPLDs include Advanced Micro Devices (AMD), Altera, Philips-Signetics, Cypress, Lattice Semiconductor Corporation and ICT. A large range of SPLD products are available from these companies. All of these SPLDs share some common features in terms of the nature of the programmable logic planes, configurable output logic, etc. However, each of these logic devices does offer some unique features that make it particularly attractive for some applications. Some of the widely exploited SPLDs include the 16XX series (16L8, 16R8,16R6 and 16R4) and 22V10 from AMD and EP610 from Altera. These devices are also widely second-sourced by many companies. The Plus 16XX series from Philips is 100 % pin and functional compatible with the 16XX series. 16R8 in the 16XX series and 22V10 PAL devices are industry standards and are widely second-sourced. We will discuss 16XX and 22V10 in a little more detail in the following paragraphs.

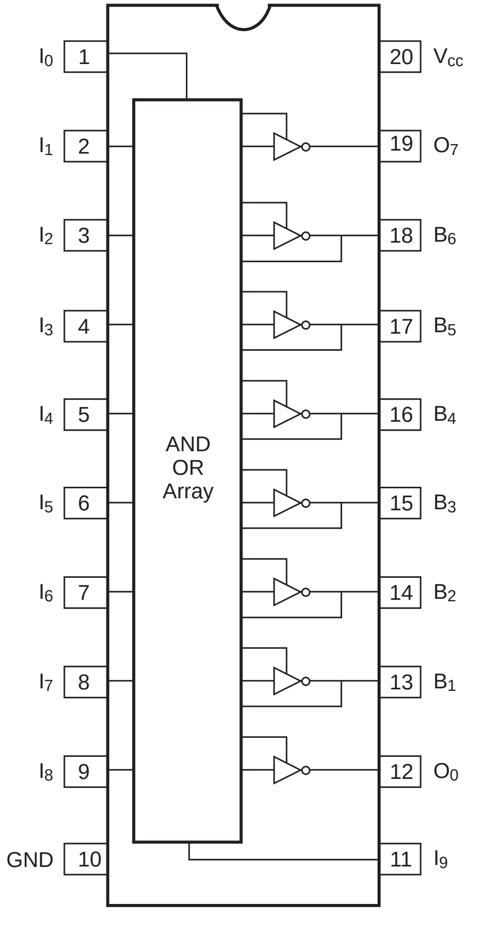
The 16XX family of PAL devices employs the familiar sum-of-products implementation comprising a programmable AND array and a fixed OR array. The family offers four PAL-type devices including 16L8, 16R8, 16R6 and 16R4.

Each of the devices in the 16XX family is characterized by a certain number of combinational and registered outputs available to the designer. The devices have three-state output buffers on each output pin, which can be programmed for individual control of all outputs. Other features include the availability of programmable bidirectional pins and output registers. These devices are capable of replacing an equivalent of four or more SSI/MSI integrated circuits. The I/O configuration of the four devices in the 16XX family is summarized in Table 9.5. Figures 9.35(a) to (d) give the basic architecture/pin connections of 16L8, 16R8, 16R6 and 16R4 respectively.

As outlined earlier, many companies offer 22V10 PAL devices. These are available in both bipolar and CMOS technologies. One such contemporary device is GAL 22V10 from Lattice Semiconductor Corporation. As inherent in the type number, the device offers a maximum of 22 inputs and 10 outputs. The outputs are versatile. That is, each one of them can be configured by the user to be either a combinational or registered output. Also, the outputs can be configured to be either active HIGH or active LOW.

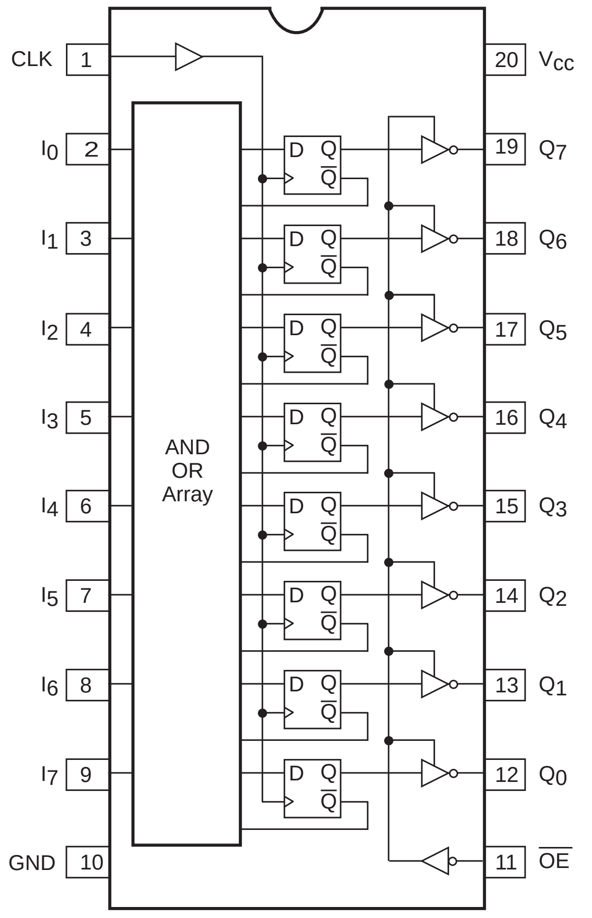
Table 9.5 Input/output configuration of the 16XX family.

Device number	Dedicated inputs	Combinational outputs	Registered outputs
16L8	10	8 (6 I/O)	0
16R8	8	0	8
16R6	8	2 I/O	6
16R4	8	4 I/O	4



(a)

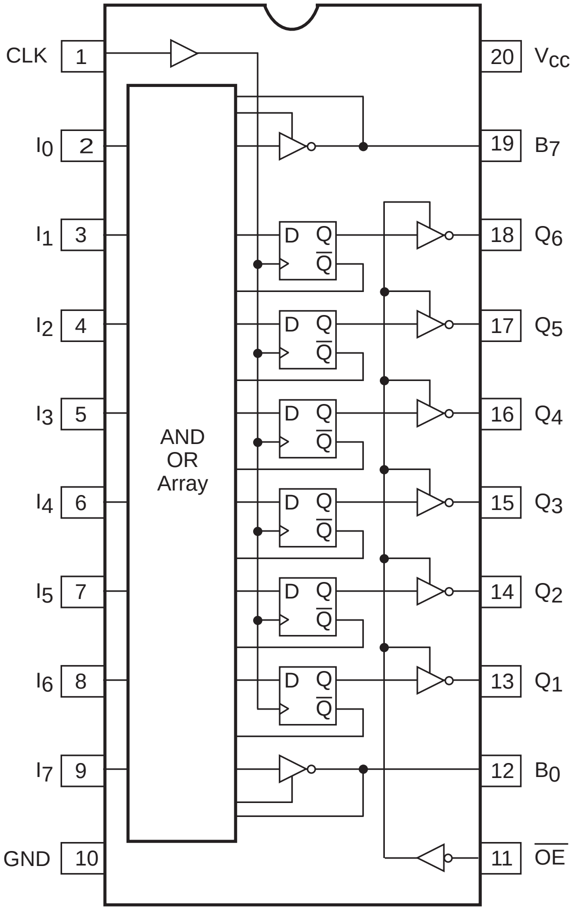
Figure 9.35 Basic architecture/pin connections of the 16XX-series PAL devices.



(b)

Figure 9.35 (*continued*).

GAL 22V10 uses E^2CMOS (electrically erasable CMOS) technology which allows the device to be reprogrammable through the use of an electrically erasable (E^2) floating-gate technology and consume much less power compared with bipolar 22V10 devices owing to the use of advanced CMOS technology. The device specifies 100 erase/write cycles, a 50–75 % saving in power consumption compared with bipolar equivalents and a maximum propagation delay of 4 ns. Each of the output logic macrocells offers two primary functional modes, which include combinational I/O and registered modes. The type of mode (whether combinational I/O or registered) and the output polarity (whether active HIGH or active LOW) are decided by the selection inputs S_0 and S_1, which are normally controlled by the logic compiler. For S_1S_0 equal to 00, 01, 10 and 11, outputs are active LOW registered, active LOW combinational, active HIGH registered and active HIGH combinational respectively.



(c)

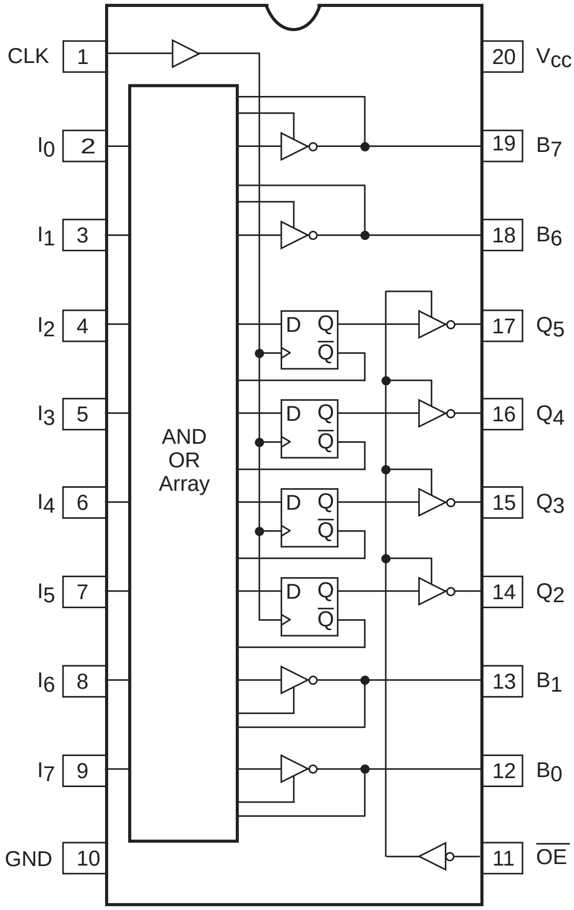
Figure 9.35 (*continued*).

Figure 9.36 shows the basic architecture and pin connection diagram of GAL 22V10. The internal architecture of the output logic macrocell (OLMC) shown as a block in Fig. 9.36 is given in Fig. 9.37.

9.12.2 CPLDs

Major CPLD manufacturers include Altera, Lattice Semiconductor Corporation, Advanced Micro Devices, ICT, Cypress and Xilinx. A large variety of CPLD devices are available from these companies. In the following paragraphs, some of the popular type numbers of CPLDs offered by some of these companies are examined in terms of their characteristic features.

We will begin with CPLDs from Altera. Altera offers three families of CPLDs. These include MAX-5000, MAX-7000 and MAX-9000. MAX-5000 uses an older technology and is used in applications



(d)

Figure 9.35 (*continued*).

where the designer is looking for cost-effective solutions. The MAX-7000 series of CPLDs are the most widely used ones. MAX-9000 is similar to MAX-7000 except for its higher logic capacity. MAX-7000 series devices use advanced CMOS technology and (E^2PROM)-based architecture and offer densities from 32 to 512 macrocells with pin-to-pin propagation delays as small as 3.5 ns. MAX-7000 devices support in-system programmability and are available with 5.0, 3.3 and 2.5 V core operating voltages. There are three types of device in the MAX-7000 series. These include MAX-7000S, MAX-7000AE and MAX-7000B. Three types are pin-to-pin compatible when used in the same package. Figure 9.38 shows the basic architecture of the MAX-7000 series of CPLDs.

AMD offers the Mach-1 to Mach-5 series of CPLDs. While Mach-1 and Mach-2 are configured around 22V10 PALs, Mach-3 and Mach-4 use 34V16 PALs. Mach-5 is similar to the Mach-4 CPLD except that it offers higher speed performance. All Mach devices use E^2PROM technology.

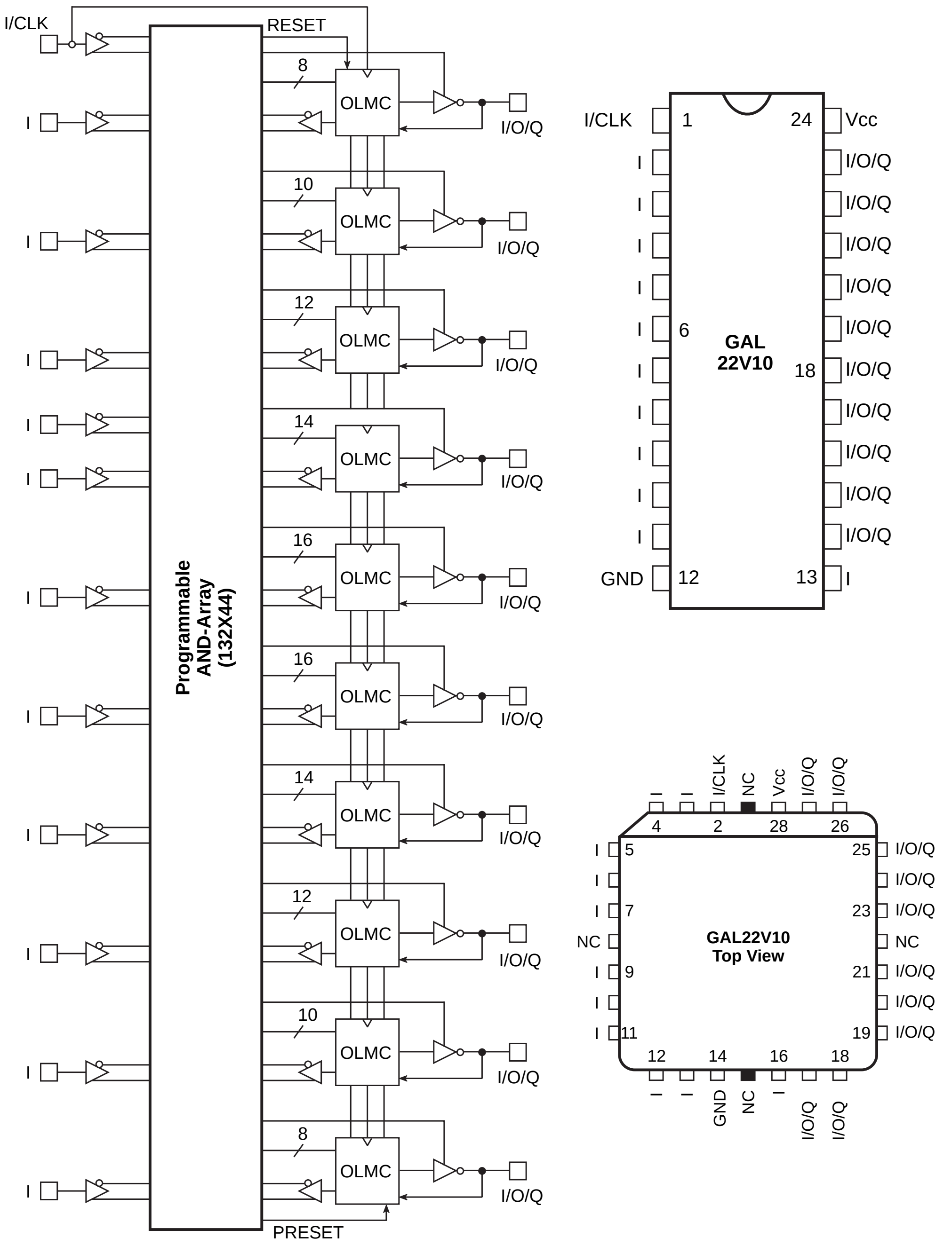


Figure 9.36 Basic architecture and pin connections of 22V10.

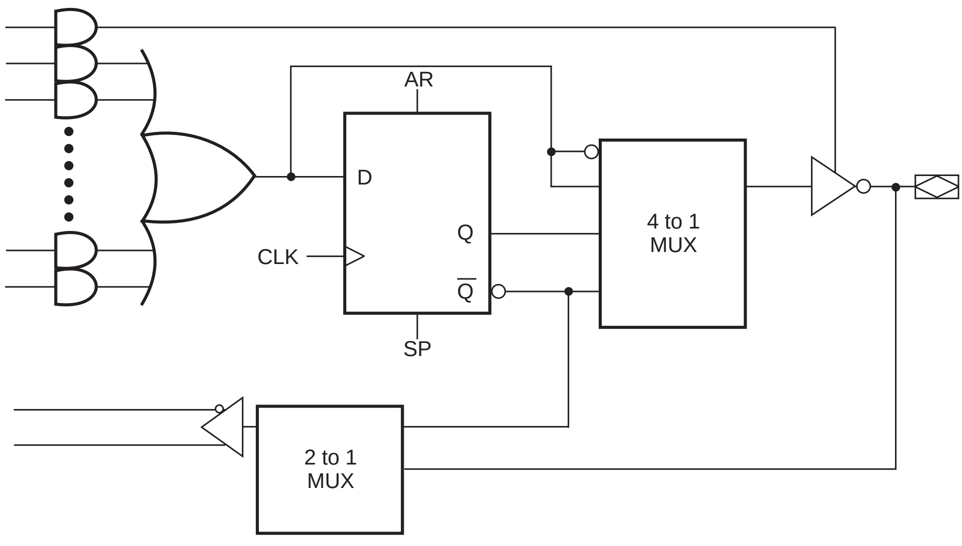


Figure 9.37 Architecture of an output GAL 22V10 logic macrocell.

Figure 9.39 shows the basic architecture of Mach-4 CPLDs. The number of 34V16-like PALs used varies from 6 to 16. Each 34V16-like PAL block consists of a maximum of 34 inputs and 16 outputs. The 34 inputs include 16 outputs that are fed back. All connections in the case of Mach-4 CPLDs, from one PAL block to another and also from a PAL block back to itself, are routed through a central switching matrix, on account of which all connections travel through the same path. This feature gives more predictable time delays in circuits implemented on Mach-4 devices.

Lattice offers the pLSI and ispLSI 1000-series, 2000-series and 3000-series of CPLDs. ispLSI devices are similar to pLSI devices except that they are in-system programmable. The three series of devices differ mainly in logic capacities and speed performance. The logic capacity in the case of the 1000-series CPLDs ranges from about 1200 to 4000 gates, and the pin-to-pin propagation delay is of the order of 10 ns. The ispLSI-1016 CPLD is one such device from the 1000-series of devices. It has a logic capacity of 2000 PLD gates and a pin-to-pin propagation delay of 7.5 ns. The device has four dedicated inputs, 32 universal I/O pins and 96 registers. It uses high-performance E^2CMOS technology, because of which it offers reprogrammability of the logic as well as the interconnects to provide truly reconfigurable systems.

The 2000-series devices have a logic capacity of 600–2000 equivalent gates that offer a higher ratio of macrocells to I/O pins. With a pin-to-pin propagation delay of 5.5 ns, they offer a higher speed performance compared with 1000-series devices. Of the three device families, the 3000-series has the highest logic capacity (up to 5000 equivalent gates). The propagation delay is in the range 10–15 ns. The 3000-series of devices offers some enhancements over the other two series of CPLDs to support more recent design approaches.

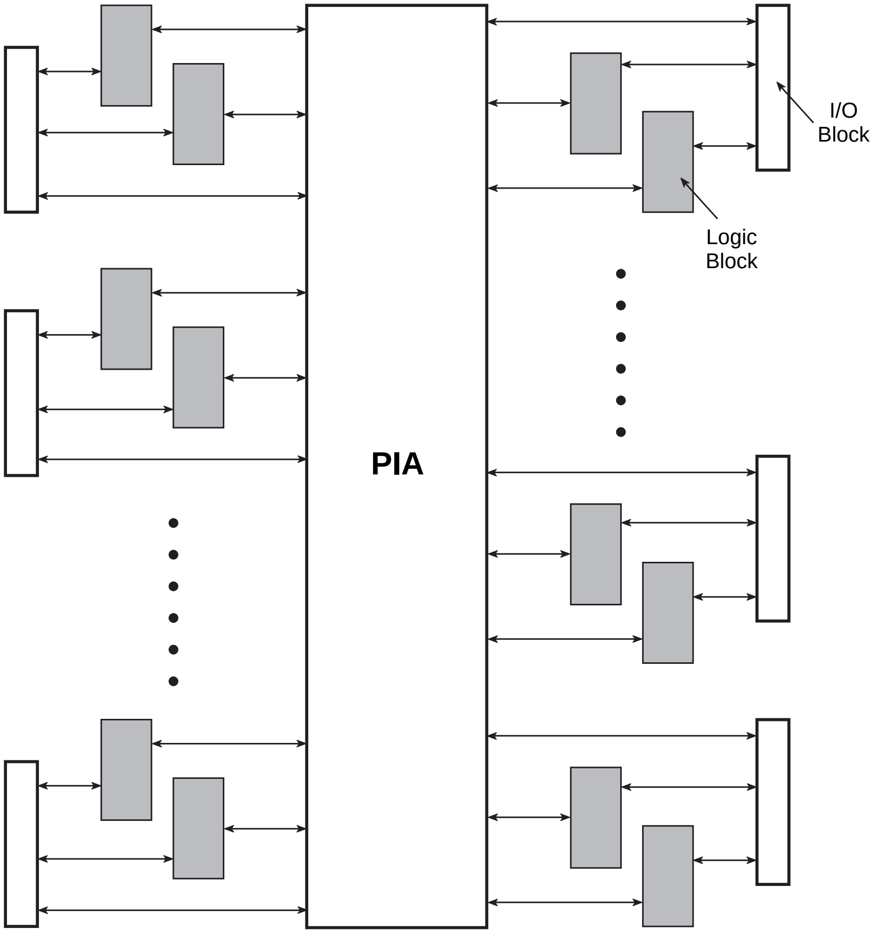


Figure 9.38 MAX-7000 series CPLD architecture.

FLASH-370 from Cypress is yet another popular class of CPLDs. FLASH-370 CPLDs use FLASH E^2PROM technology. Devices are not in-system programmable. One of the salient features of these devices is that they provide more inputs/outputs than the competing products featuring a linear relationship between the number of macrocells and the number of bidirectional I/O pins. FLASH-370 has a typical CPLD architecture as shown in Fig. 9.40, with multiple PAL-like blocks and a programmable interconnect matrix to connect them.

Xilinx, although mainly known for their range of FPGAs, offer CPLDs too. Major families of CPLDs from Xilinx include the XC-7000, CoolRunner and XC-9500 in-system programmable family of devices. The XC-7000 family of CPLDs further comprises two major series, namely XC-7200 and XC-7300. XC-7300 is an enhanced version of XC-7200 in terms of gate capacity and speed

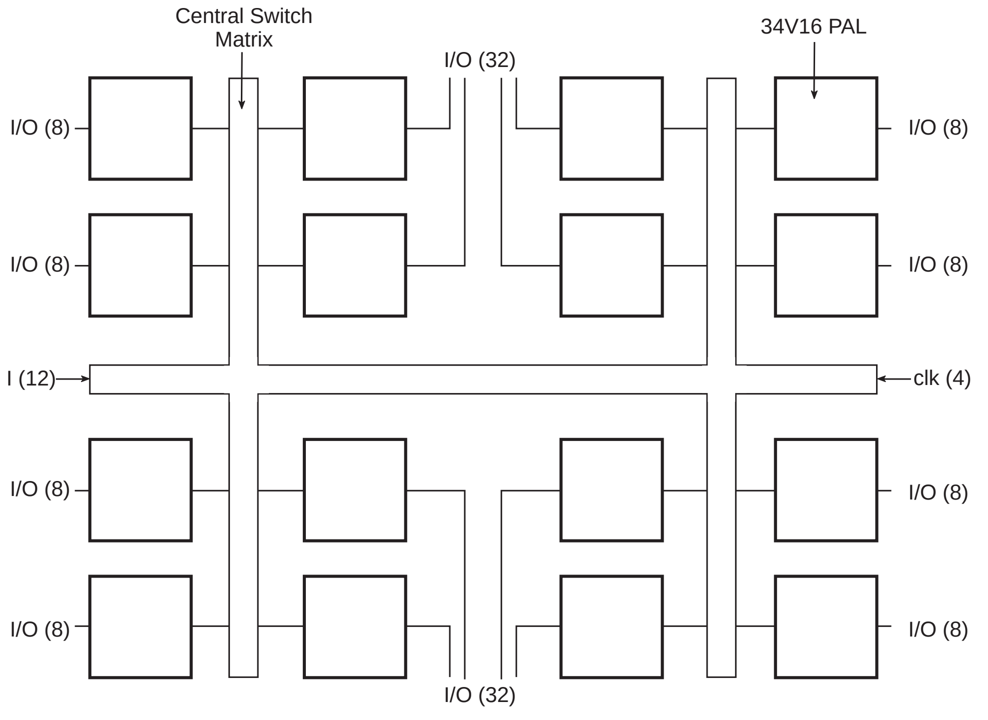


Figure 9.39 Mach-4 CPLD architecture.

performance. XC-7200 offers a logic capacity of 600–1500 gates with a speed performance of 25 ns pin-to-pin propagation delay. XC-7300 offers a gate capacity of up to 3000 gates. Each device in the XC-7000 family contains SPLD-like logic blocks, with each block having nine macrocells. A notable difference between the XC-7000 family of CPLDs and their counterparts from other manufacturers is that each macrocell has two OR gates whose outputs feed a two-bit arithmetic logic unit (ALU), which in turn can generate any function of its two inputs. The ALU output feeds a configurable flip-flop.

The CoolRunner family of CPLDs is characterized by high speed (5 ns pin-to-pin propagation delay) and low power consumption (100 μA of standby current). The family includes the XPLAE series of devices, available in 32, 64 and 128 macrocell versions, the XPLA2-series, which is SRAM-based and available in 320 and 920 macrocell capacities, and the XPLA3 series, available in 32, 64, 128, 256 and 384 macrocell versions.

The XC-9500 family of devices comprises the XC-9536, XC-9572, XC-95108, XC-95144, XC-95216 and XC-95288 series of CPLDs. The family offers a logic capacity ranging from 800 gates (in the case of XC-9536) to 6400 gates (in the case of XC-95288), with a propagation delay varying from 5 ns (in the case of XC-9536) to 15 ns (in the case of XC-95288). Architectural features of the XC-9500 family of CPLDs provide in-system programmability with a minimum of 10 000 program/erase cycles. Other features include output slew rate control and user-programmable ground pins, which help reduce system noise.

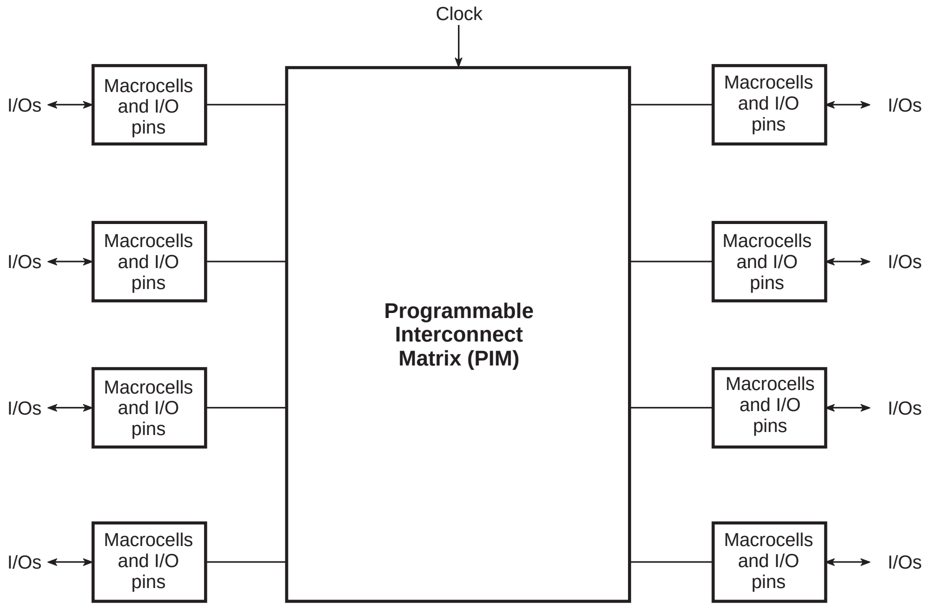


Figure 9.40 FLASH-370 CPLD architecture.

9.12.3 FPGAs

There are two broad categories of FPGAs, namely SRAM-based FPGAs and antifuse-based FPGAs. While Xilinx and Altera are the major players in the former category, antifuse-based devices are offered mainly by Xilinx, Actel, Quicklogic and Cypress. FPGAs were introduced by Xilinx with the XC-2000 series of devices, which have been subsequently followed up by the XC-3000 series, XC-4000 series and XC-5000 series of devices. Of all these, the XC-4000 series is the most widely used one. These are all SRAM-based. Xilinx has also introduced an antifuse-based FPGA family of FPGAs called XC-8100.

The basic architecture of the XC-4000 family is built around a two-dimensional array of configurable logic blocks (CLBs) that can be interconnected by horizontal and vertical routing channels and are surrounded by a perimeter of programmable input/output blocks (IOBs). CLBs provide the functional elements for constructing the user-desired logic function, and IOBs provide the interface between the package pins and internal signal lines. These devices are reconfigurable and are in-system programmable. Table 9.6 gives salient features of the XC-4000X and XC-4000E series of FPGAs.

Altera offers the FLEX-8000 and FLEX-10000 series of FPGAs. FLEX-8000 is SRAM-based. It combines the fine-grained architecture and high register count characteristics of FPGAs with the high speed and predictable interconnect timing delays of CPLDs. The basic logic element comprises a four-input look-up table (LUT) that provides combinational capability and a programmable register that provides sequential capability. Table 9.7 outlines salient features of the FLEX-8000 series of devices.

The FLEX-10000 series offers all the features of FLEX-8000 series devices, with the addition of variable-sized blocks of SRAM called embedded array blocks (EABs). Each of the EABs can be

Table 9.6 Salient features of the XC-4000X and XC-4000E series of FPGAs.

Device number	Logic cells	Maximum logic gates (no RAM)	CLB matrix	Number of CLBs	Number of flip-flops	Maximum user I/Os
XC4002XL	152	1 600	8×8	64	256	64
XC4003E	238	3 000	10×10	100	360	80
XC4005E/XL	466	5 000	14×14	196	616	112
XC4006E	608	6 000	16×16	256	768	128
XC4008E	770	8 000	18×18	324	936	144
XC4010E/XL	950	10 000	20×20	400	1120	160
XC4013E/XL	1368	13 000	24×24	576	1536	192
XC4020E/XL	1862	20 000	28×28	784	2016	224
XC4025E	2432	25 000	32×32	1024	2560	256
XC4028EX/	3078	28 000	32×32	1024	2560	256
XC4036EX/XL	3078	36 000	36×36	1296	3168	288
XC4044XL	3800	44 000	40×40	1600	3840	320
XC4052	4598	52 000	44×44	1936	4576	352
XC4062XL	5472	62 000	48×48	2304	5376	384
XC4085	7448	85 000	56×56	3136	7168	448

Table 9.7 Salient features of the FLEX-8000 series of devices.

Device number	Usable Gates	Flip-flops	Logic Array Blocks (LAB)	Logic Elements (LE)	Maximum User I/O PIns
EPF 8282A/AV	2 500	282	26	208	78
EPF 8452A	4 000	452	42	336	120
EPF 8636A	6 000	636	63	504	136
EPF 8820A	8 000	820	84	672	152
EPF 81188A	12 000	1188	126	1008	184
EPF 81500A	16 000	1500	162	1296	208

configured to serve as an SRAM block with a variable aspect ratio of 256×8, 512×4, $1K \times 2$ or $2K \times 1$.

AT&T offers SRAM-based FPGAs that are similar in architecture to those offered by Xilinx. The overall structure is called an optimized reconfigurable cell array (ORCA). The basic logic block is referred to as a programmable function unit (PFU). Similarities with the Xilinx-4000 series FPGAs include arithmetic circuitry being a part of the PFU and PFU configurability as a RAM. The PFU can be configured as either four four-input LUTs or as two five-input LUTs or as one six-input LUT. When configured as four-input LUTs, it is essential that the various LUT inputs come from the same PFU input. Although on the one hand this reduces the functionality of the PFU, on the other hand it significantly reduces the associated wiring cost.

Actel FPGAs use antifuse technology. Actel offers three main families of FPGA devices, namely Act-1, Act-2 and Act-3. All three series of devices have similar features. The structure is similar to that

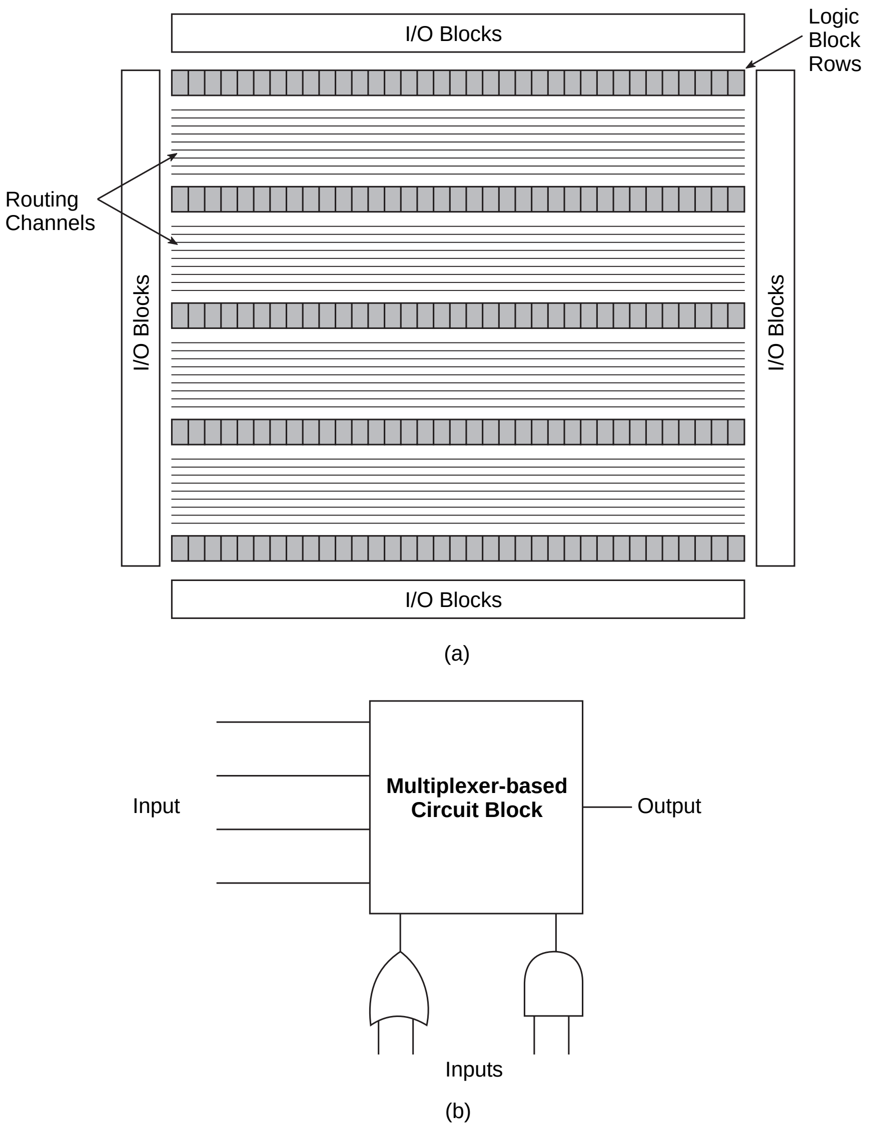


Figure 9.41 Actel FPGA.

of traditional gate arrays comprising logic blocks arranged in horizontal rows with horizontal routing channels between adjacent rows, as shown in Fig. 9.41(a). Actel chips also have vertical wires that overlay the logic blocks to provide signal paths that span multiple rows. These are not shown in Fig. 9.41(a). The logic block is not LUT based. Instead, it comprises an AND gate and an OR gate feeding a multiplexer circuit block, as shown in Fig. 9.41(b). The multiplexer circuit, along with the two gates, can realize a large range of logic functions. In the case of Act-3 FPGAs, 50 % of the logic blocks also contain a flip-flop.

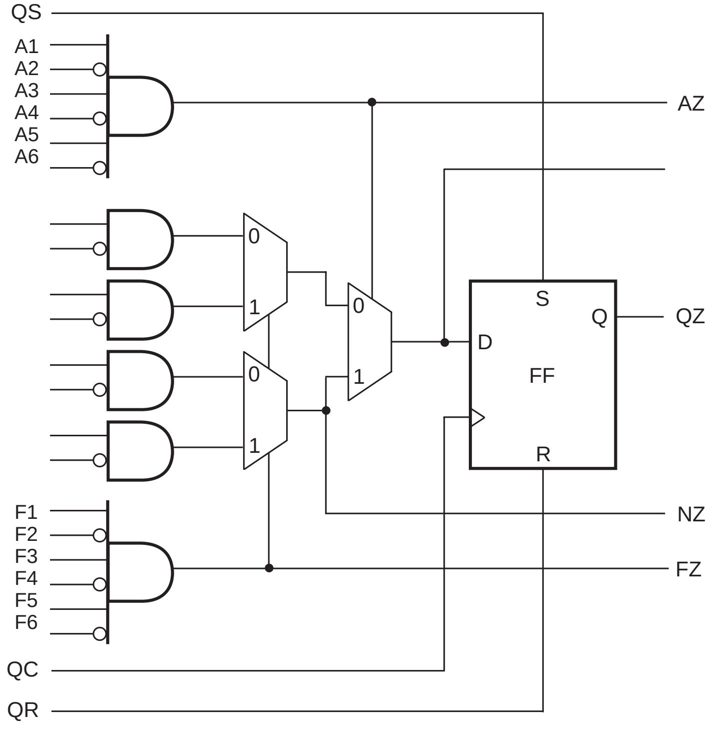


Figure 9.42 Quicklogic FPGA logic block.

Quicklogic also offers antifuse-based FPGAs, like Actel. They offer two families of devices, namely pASIC and pASIC-2. pASIC-2 is an enhanced version of pASIC. The overall structure is array based like the Xilinx FPGAs. The logic blocks are similar to those used in the Actel FPGAs, although more complex than their Actel counterparts. Also, each logic block contains a flip-flop. Figure 9.42 shows the architecture.

Review Questions

1. How does a programmable logic device differ from a fixed logic device? What are the primary advantages of using programmable logic devices?
2. Distinguish between a programmable logic array (PLA) device and a programmable array logic (PAL) device in terms of architecture and capability to implement Boolean functions.
3. How does a generic array logic (GAL) device differ from its PAL counterpart? Do they differ in their internal architecture? If yes, then how?
4. What are complex programmable logic devices (CPLDs)? Briefly outline salient features of these devices and application areas where these devices fit the best.
5. How does the architecture of a typical FPGA device differ from that of a CPLD? In what way does the architecture affect the timing performance in the two cases?

6. What are the various interconnect technologies used for the purpose of programming PLDs? Briefly describe each one of them.
7. What is a hardware description language? What are the requirements of a good HDL? Briefly describe the salient features of VHDL and Verilog.
8. What do you understand by the following as regards programmable logic devices?

   (a) combinational and registered outputs;
   (b) configurable output logic cell;
   (c) reprogrammable PLD;
   (d) in-system programmability.

Problems

1. Figure 9.43 shows a portion of the internal logic diagram of a certain PAL device that uses antifuse interconnect technology. In the diagram shown, a cross ($\times$) represents an unprogrammed interconnect and the absence of a cross ($\times$) at an intersection of input and product lines represents programmed interconnects; a dot ($\bullet$) represents a hard-wired interconnect. Write (a) the Boolean expression for Y and (b) the Boolean expression for Y if the interconnect technology were fuse based.

(a) $Y = \overline{A}.B + A.\overline{B}$; (b) the same as in the case of (a)

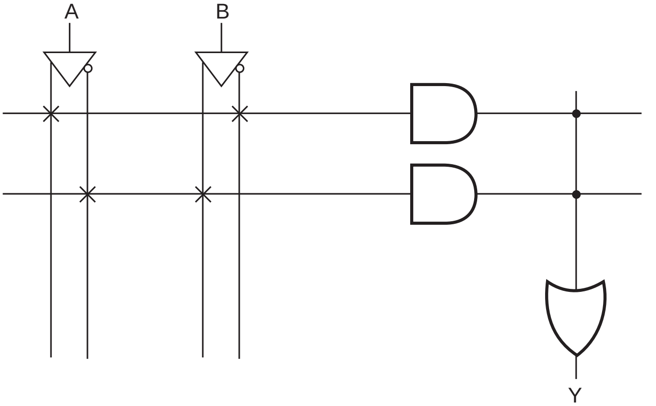


Figure 9.43 Problem 1.

2. Determine the size of PROM required for implementing the following logic circuits.

   (a) 16-to-1 multiplexer;
   (b) four-bit binary adder.

(a) 1M$\times$1; (b) 512$\times$5

3. Determine the number of programmable interconnections in the following programmable logic devices.

   (a) 1K $\times$ 4 PROM;
   (b) PLA device with four input variables, 32 AND gates and four OR gates;
   (c) PAL device with eight input variables, 16 AND gates and four OR gates.

   (a) 4096; (b) 384; (c) 256

4. A and B are two binary variables. The objective is to design a magnitude comparator to produce $A = B$, $A < B$ and $A > B$ outputs. Design a suitable PLD with a PAL-like architecture using anti-fuse based interconnects.

   Fig. 9.44

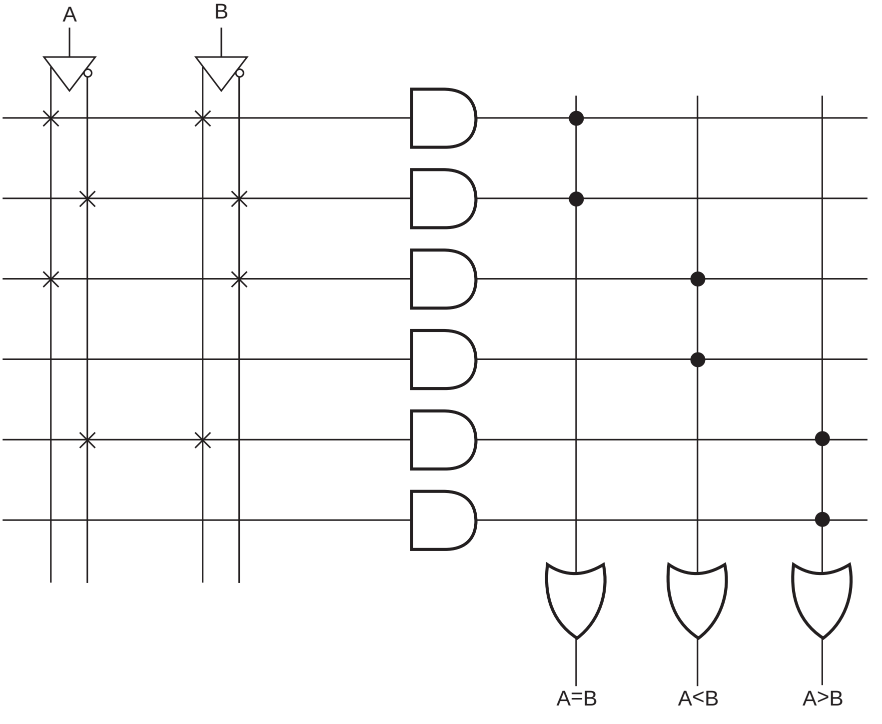


Figure 9.44 Answer to problem 4.

5. Figure 9.45 shows a programmed PAL device using fuse-based interconnects. Examine the logic diagram and determine the logic block implemented by the PLD. A cross ($\times$) represents an unprogrammed interconnection and a dot ($\bullet$) represents a hard-wired interconnection.

   Full adder

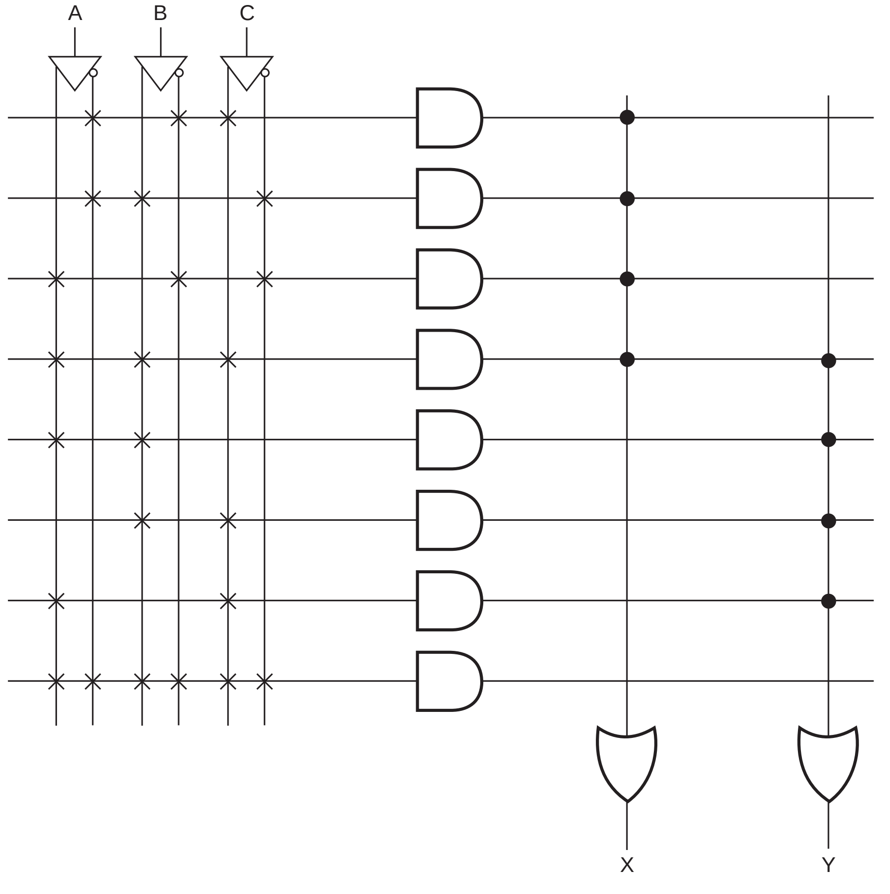


Figure 9.45 Problem 5.

Further Reading

1. Seals, R. C. and Whapshott, G. F. (1997) *Programmable Logic: PLDs and FPGAs*, McGraw-Hill, USA.
2. Dueck, R. (2003) *Digital Design with CPLD Applications and VHDL*, Thomson Delmar Learning, New York, USA.
3. Chartrand, L. (2003) *Digital Fundamentals: Experiments and Concepts with CPLD*, Thomson Delmar Learning, New York, USA.
4. Oldfield, J. and Dorf, R. (1995) *Field Programmable Gate Arrays*, John Wiley & Sons, Inc., New York, USA.
5. Trimberger, S. (Ed.) (1994) *Field Programmable Gate Array Technology*, Kluwer Academic Publishers, MA, USA.
6. Brown, S., Francis, R., Rose, J. and Vranesic, Z. (1992) *Field Programmable Gate Arrays*, Kluwer Academic Publishers, MA, USA.

10

Flip-Flops and Related Devices

Having discussed combinational logic circuits at length in previous chapters, the focus in the present chapter and in Chapter 11 will be on sequential logic circuits. While a logic gate is the most basic building block of combinational logic, its counterpart in sequential logic is the flip-flop. The chapter begins with a brief introduction to different types of multivibrator, including the bistable multivibrator, which is the complete technical name for a flip-flop, the monostable multivibrator and the astable multivibrator. The flip-flop is not only used individually for a variety of applications; it also forms the basis of many more complex logic functions. Counters and registers, to be covered in Chapter 11 are typical examples. There is a large variety of flip-flops having varying functional tables, input clocking requirements and other features. In this chapter, we will discuss all these basic types of flip-flop in terms of their functional aspects, truth tables, salient features and application aspects. The text is suitably illustrated with a large number of solved examples. Application-relevant information, including a comprehensive index of flip-flops and related devices belonging to different logic families, is given towards the end of the chapter. Pin connection diagrams and functional tables are given in the companion website.

10.1 Multivibrator

Multivibrators, like the familiar sinusoidal oscillators, are circuits with regenerative feedback, with the difference that they produce pulsed output. There are three basic types of multivibrator, namely the bistable multivibrator, the monostable multivibrator and the astable multivibrator.

10.1.1 Bistable Multivibrator

A *bistable multivibrator* circuit is one in which both LOW and HIGH output states are stable. Irrespective of the logic status of the output, LOW or HIGH, it stays in that state unless a change is

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

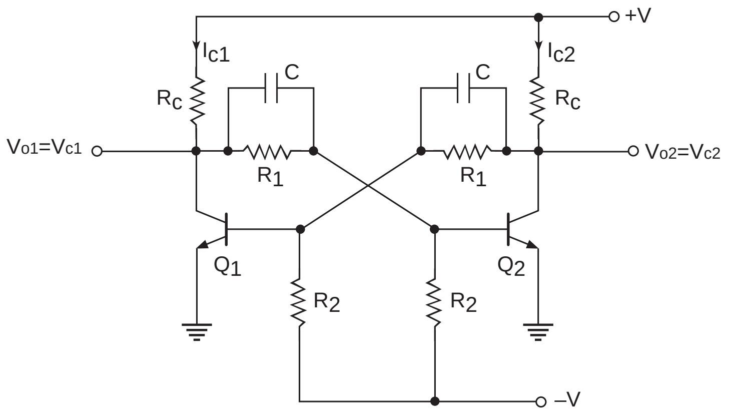


Figure 10.1 Bistable multivibrator.

induced by applying an appropriate trigger pulse. As we will see in the subsequent pages, the operation of a bistable multivibrator is identical to that of a flip-flop. Figure 10.1 shows the basic bistable multivibrator circuit. This is the fixed-bias type of bistable multivibrator. Other configurations are the self-bias type and the emitter-coupled type. However, the operational principle of all types is the same. The multivibrator circuit of Fig. 10.1 functions as follows.

In the circuit arrangement of Fig. 10.1 it can be proved that both transistors Q_1 and Q_2 cannot be simultaneously ON or OFF. If Q_1 is ON, the regenerative feedback ensures that Q_2 is OFF, and when Q_1 is OFF, the feedback drives transistor Q_2 to the ON state. In order to vindicate this statement, let us assume that both Q_1 and Q_2 are conducting simultaneously. Owing to slight circuit imbalance, which is always there, the collector current in one transistor will always be greater than that in the other. Let us assume that $I_{c2} > I_{c1}$. Lesser I_{c1} means a higher V_{c1}. Since V_{c1} is coupled to the Q_2 base, a rise in V_{c1} leads to an increase in the Q_2 base voltage. Increase in the Q_2 base voltage results in an increase in I_{c2} and an associated reduction in V_{c2}. Reduction in V_{c2} leads to a reduction in Q_1 base voltage and an associated fall in I_{c1}, with the result thatV_{c1} increases further. Thus, a slight circuit imbalance has initiated a regenerative action that culminates in transistor Q_1 going to cut-off and transistor Q_2 getting driven to saturation. To sum up, whenever there is a tendency of one of the transistors to conduct more than the other, it will end up with that transistor going to saturation and driving the other transistor to cut-off. Now, if we take the output from the Q_1 collector, it will be LOW ($= V_{CE1}$ sat.) if Q_1 was initially in saturation. If we apply a negative-going trigger to the Q_1 base to cause a decrease in its collector current, a regenerative action would set in that would drive Q_2 to saturation and Q_1 to cut-off. As a result, the output goes to a HIGH ($= +V_{CC}$) state. The output will stay HIGH until we apply another appropriate trigger to initiate a transition. Thus, both of the output states, when the output is LOW and also when the output is HIGH, are stable and undergo a change only when a transition is induced by means of an appropriate trigger pulse. That is why it is called a bistable multivibrator.

10.1.2 Schmitt Trigger

A Schmitt trigger circuit is a slight variation of the bistable multivibrator circuit of Fig. 10.1. Figure 10.2 shows the basic Schmitt trigger circuit. If we compare the bistable multivibrator circuit of Fig. 10.1

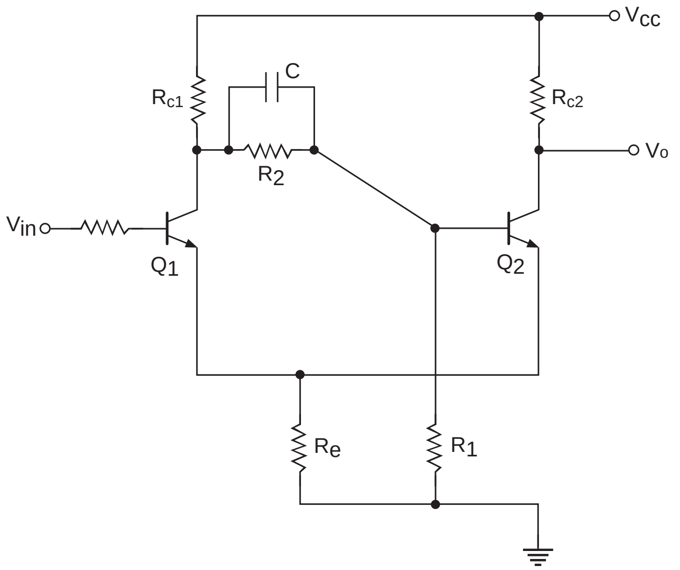


Figure 10.2 Schmitt trigger circuit.

with the Schmitt trigger circuit of Fig. 10.2, we find that coupling from Q_2 collector to Q_1 base in the case of a bistable circuit is absent in the case of a Schmitt trigger circuit. Instead, the resistance R_e provides the coupling. The circuit functions as follows.

When V_{in} is zero, transistor Q_1 is in cut-off. Coupling from Q_1 collector to Q_2 base drives transistor Q_2 to saturation, with the result that V_o is LOW. If we assume that V_{CE2} (sat.) is zero, then the voltage across R_e is given by the equation

$$\text{Voltage across } R_e = [V_{CC}.R_e/(R_e + R_{c2})] \tag{10.1}$$

This is also the emitter voltage of transistor Q_1. In order to make transistor Q_1 conduct, V_{in} must be at least 0.7 V more than the voltage across R_e. That is,

$$V_{in}(\text{min.}) = [V_{CC}.R_e/(R_e + R_{c2})] + 0.7 \tag{10.2}$$

When V_{in} exceeds this voltage, Q_1 starts conducting. The regenerative action again drives Q_2 to cut-off. The output goes to the HIGH state. Voltage across R_e changes to a new value given by the equation

$$\text{Voltage across } R_e = [V_{CC}.R_e/(R_e + R_{c1})] \tag{10.3}$$

$$V_{in} = [V_{CC}.R_e/(R_e + R_{c1})] + 0.7 \tag{10.4}$$

Transistor Q_1 will continue to conduct as long as V_{in} is equal to or greater than the value given by Equation (10.4). If V_{in} falls below this value, Q_1 tends to come out of saturation and conduct less heavily. The regenerative action does the rest, with the process culminating in Q_1 going to cut-off and Q_2 to saturation. Thus, the state of output (HIGH or LOW) depends upon the input voltage level.

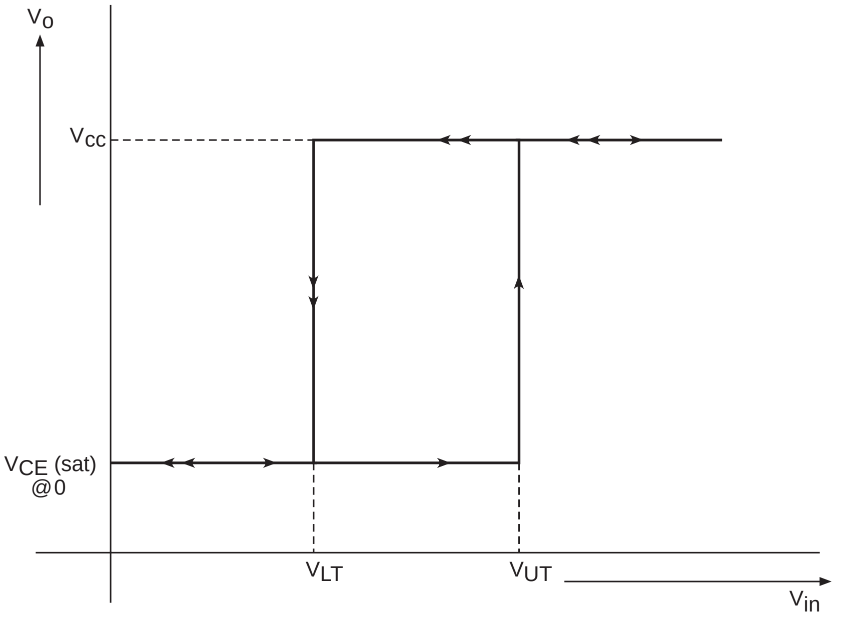


Figure 10.3 Transfer characteristics of a Schmitt trigger.

The HIGH and LOW states of the output correspond to two distinct input levels given by Equations (10.2) and (10.4) and therefore the values of R_{c1},R_{c2},R_e and V_{CC}. The Schmitt trigger circuit of Fig. 10.2 therefore exhibits hysteresis. Figure 10.3 shows the transfer characteristics of the Schmitt trigger circuit. The lower trip point V_{LT} and the upper trip point V_{UT} of these characteristics are respectively given by the equations

$$V_{LT} = [V_{CC}.R_e/(R_e + R_{c1})] + 0.7 \tag{10.5}$$

$$V_{UT} = [V_{CC}.R_e/(R_e + R_{c2})] + 0.7 \tag{10.6}$$

10.1.3 Monostable Multivibrator

A *monostable multivibrator*, also known as a *monoshot*, is one in which one of the states is stable and the other is quasi-stable. The circuit is initially in the stable state. It goes to the quasi-stable state when appropriately triggered. It stays in the quasi-stable state for a certain time period, after which it comes back to the stable state. Figure 10.4 shows the basic monostable multivibrator circuit. The circuit functions as follows. Initially, transistor Q_2 is in saturation as it gets its base bias from $+V_{CC}$ through R. Coupling from Q_2 collector to Q_1 base ensures that Q_1 is in cut-off. Now, if an appropriate trigger pulse induces a transition in Q_2 from saturation to cut-off, the output goes to the HIGH state. This HIGH output when coupled to the Q_1 base turns Q_1 ON. Since there is no direct coupling from Q_1 collector to Q_2 base, which is necessary for a regenerative process to set in, Q_1 is not necessarily

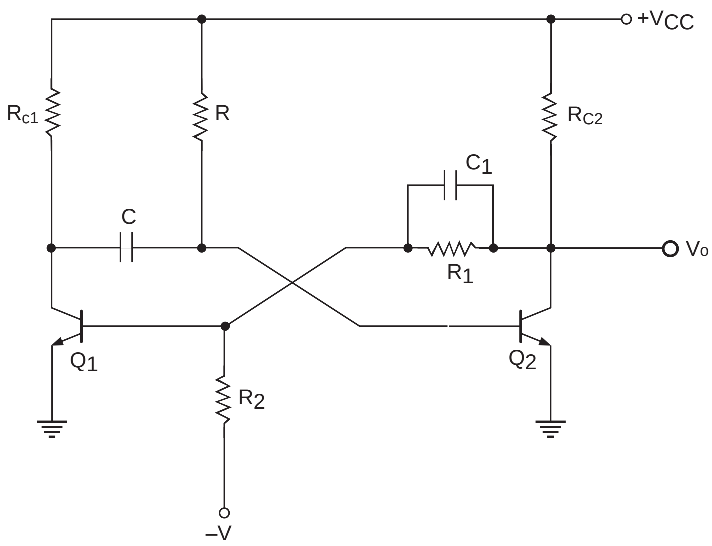


Figure 10.4 Monostable multivibrator.

in saturation. However, it conducts some current. The Q_1 collector voltage falls by $I_{c1}R_{c1}$ and the Q_2 base voltage falls by the same amount, as the voltage across a capacitor (C in this case) cannot change instantaneously. To sum up, the moment we applied the trigger, Q_2 went to cut-off and Q_1 started conducting. But now there is a path for capacitor C to charge from V_{CC} through R and the conducting transistor. The polarity of voltage across C is such that the Q_2 base potential rises. The moment the Q_2 base voltage exceeds the cut-in voltage, it turns Q_2 ON, which, owing to coupling through R_1, turns Q_1 OFF. And we are back to the original state, the stable state. Whenever we trigger the circuit into the other state, it does not stay there permanently and returns back after a time period that depends upon R and C. The greater the time constant RC, the longer is the time for which it stays in the other state, called the quasi-stable state.

10.1.3.1 Retriggerable Monostable Multivibrator

In a conventional monostable multivibrator, once the output is triggered to the quasi-stable state by applying a suitable trigger pulse, the circuit does not respond to subsequent trigger pulses as long as the output is in quasi-stable state. After the output returns to its original state, it is ready to respond to the next trigger pulse. There is another class of monostable multivibrators, called *retriggerable monostable multivibrators*. These respond to trigger pulses even when the output is in the quasi-stable state. In this class of monostable multivibrators, if n trigger pulses with a time period of T_t are applied to the circuit, the output pulse width, that is, the time period of the quasi-stable state, equals $(n-1)T_t + T$, where T is the output pulse width for the single trigger pulse and $T_t < T$. Figure 10.5 shows the output pulse width in the case of a retriggerable monostable multivibrator for repetitive trigger pulses.

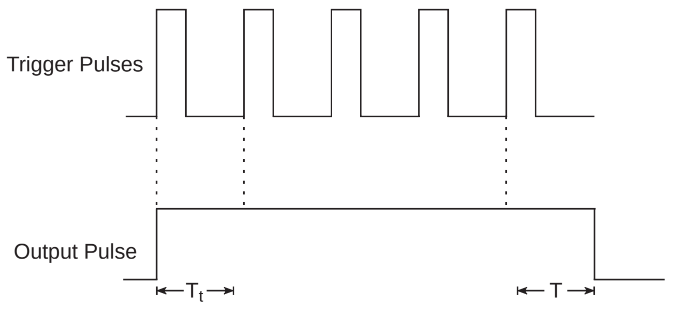


Figure 10.5 Retriggerable monostable multivibrator output for repetitive trigger pulses.

10.1.4 Astable Multivibrator

In the case of an astable multivibrator, neither of the two states is stable. Both output states are quasi-stable. The output switches from one state to the other and the circuit functions like a free-running square-wave oscillator. Figure 10.6 shows the basic astable multivibrator circuit. It can be proved that, in this type of circuit, neither of the output states is stable. Both states, LOW as well as HIGH, are quasi-stable. The time periods for which the output remains LOW and HIGH depends upon R_2C_2 and R_1C_1 time constants respectively. For $R_1C_1 = R_2C_2$, the output is a symmetrical square waveform. The circuit functions as follows. Let us assume that transistor Q_2 is initially conducting, that is, the output is LOW. Capacitor C_2 in this case charges through R_2 and the conducting transistor from V_{CC}, and, the moment the Q_1 base potential exceeds its cut-in voltage, it is turned ON. A fall in Q_1 collector

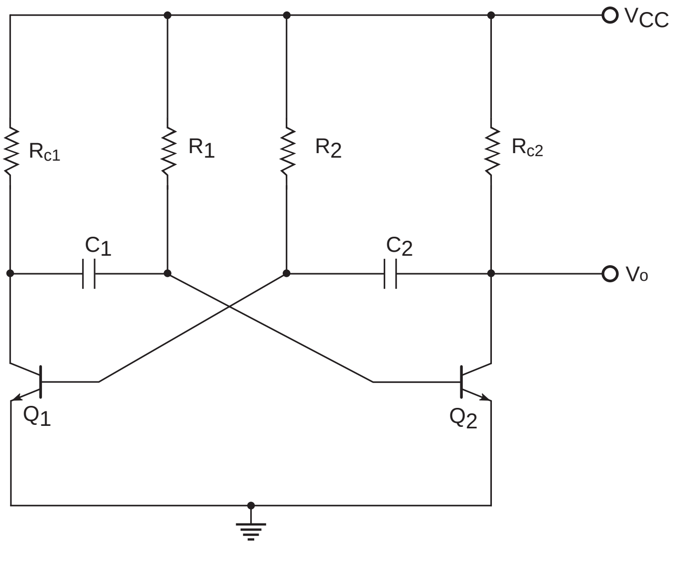


Figure 10.6 Astable multivibrator.

potential manifests itself at the Q_2 base as voltage across a capacitor cannot change instantaneously. The output goes to the HIGH state as Q_2 is driven to cut-off. However, C_1 has now started charging through R_1 and the conducting transistor Q_1 from V_{CC}. The moment the Q_2 base potential exceeds the cut-in voltage, it is again turned ON, with the result that the output goes to the LOW state. This process continues and, owing to both the couplings (Q_1 collector to Q_2 base and Q_2 collector to Q_1 base) being capacitive, neither of the states is stable. The circuit produces a square-wave output.

10.2 Integrated Circuit (IC) Multivibrators

In this section, we will discuss monostable and astable multivibrator circuits that can be configured around some of the popular digital and linear integrated circuits. The bistable multivibrator, which is functionally the same as a flip-flop, will not be discussed here. Flip-flops are discussed at length from Section 10.3 onwards.

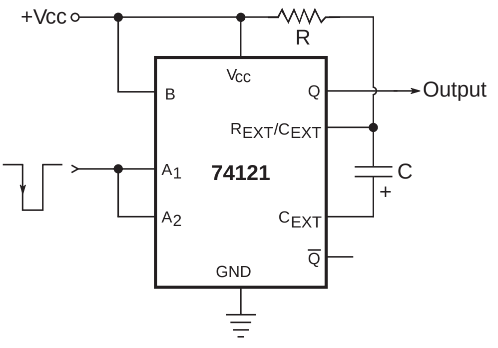
10.2.1 Digital IC-Based Monostable Multivibrator

Some of the commonly used digital ICs that can be used as monostable multivibrators include 74121 (single monostable multivibrator), 74221 (dual monostable multivibrator), 74122 (single retriggerable monostable multivibrator) and 74123 (dual retriggerable monostable multivibrator), all belonging to the TTL family, and 4098B (dual retriggerable monostable multivibrator) belonging to the CMOS family. Figure 10.7 shows the use of IC 74121 as a monostable multivibrator along with a trigger input. The IC provides features for triggering on either LOW-to-HIGH or HIGH-to-LOW edges of the trigger pulses. Figure 10.7(a) shows one of the possible application circuits for HIGH-to-LOW edge triggering, and Fig. 10.7(b) shows one of the possible application circuits for LOW-to-HIGH edge triggering. The output pulse width depends on external R and C. The output pulse width can be computed from $T = 0.7\,RC$. Recommended ranges of values for R and C are 4–40 KΩ and 10 pf to 1000 μF respectively. The IC provides complementary outputs. That is, we have a stable LOW or HIGH state and the corresponding quasi-stable HIGH or LOW state available on Q and $\overline{Q}$ outputs.

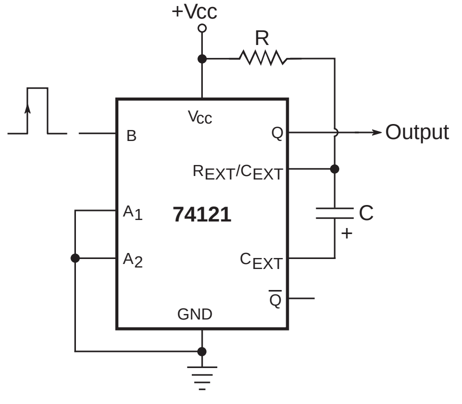
Figure 10.8 shows the use of 74123, a retriggerable monostable multivibrator. Like 74121, this IC, too, provides features for triggering on either LOW-to-HIGH or HIGH-to-LOW edges of the trigger pulses. The output pulse width depends on external R and C. It can be computed from $T = 0.28RC\times [1 + (0.7/R)]$, where R and C are respectively in kiloohms and picofarads and T is in nanoseconds. This formula is valid for $C > 1000$ pF. The recommended range of values for R is 5–50 KΩ . Figures 10.8(a) and (b) give application circuits for HIGH-to-LOW and LOW-to-HIGH triggering respectively. It may be mentioned here that there can be other triggering circuit options for both LOW-to-HIGH and HIGH-to-LOW edge triggering of monoshot.

10.2.2 IC Timer-Based Multivibrators

IC timer 555 is one of the most commonly used general-purpose linear integrated circuits. The simplicity with which monostable and astable multivibrator circuits can be configured around this IC is one of the main reasons for its wide use. Figure 10.9 shows the internal schematic of timer IC 555. It comprises two opamp comparators, a flip-flop, a discharge transistor, three identical resistors and an output stage. The resistors set the reference voltage levels at the noninverting input of the lower comparator and the inverting input of the upper comparator at ($+V_{CC}/3$) and ($+2V_{CC}/3$). The outputs of the two comparators feed the SET and RESET inputs of the flip-flop and thus decide the logic status



(a)



(b)

Figure 10.7 74121 as a monoshot.

of its output and subsequently the final output. The flip-flop complementary outputs feed the output stage and the base of the discharge transistor. This ensures that when the output is HIGH the discharge transistor is OFF, and when the output is LOW the discharge transistor is ON. Different terminals of timer 555 are designated as *ground* (terminal 1), *trigger* (terminal 2), *output* (terminal 3), *reset* (terminal 4), *control* (terminal 5), *threshold* (terminal 6), *discharge* (terminal 7) and $+V_{CC}$ (terminal 8). With this background, we will now describe the astable and monostable circuits configured around timer 555.

10.2.2.1 Astable Multivibrator Using Timer IC 555

Figure 10.10(a) shows the basic 555 timer based astable multivibrator circuit. Initially, capacitor C is fully discharged, which forces the output to go to the HIGH state. An open discharge transistor allows the capacitor C to charge from $+V_{CC}$ through R_1 and R_2. When the voltage across C exceeds $+2V_{CC}/3$, the output goes to the LOW state and the discharge transistor is switched ON at the same time.

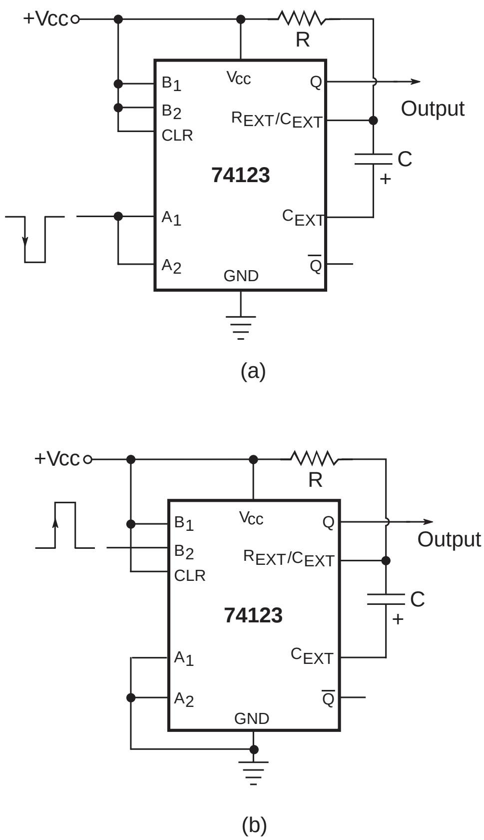


Figure 10.8 74123 as a retriggerable monoshot.

Capacitor C begins to discharge through R_2 and the discharge transistor inside the IC. When the voltage across C falls below $+V_{CC}/3$, the output goes back to the HIGH state. The charge and discharge cycles repeat and the circuit behaves like a free-running multivibrator. Terminal 4 of the IC is the RESET terminal. usually, it is connected to $+V_{CC}$. If the voltage at this terminal is driven below 0.4 V, the output is forced to the LOW state, overriding command pulses at terminal 2 of the IC. The HIGH-state and LOW-state time periods are governed by the charge ($+V_{CC}/3$ to $+2V_{CC}/3$) and discharge ($+2V_{CC}/3$ to $+V_{CC}/3$) timings. these are given by the equations

$$\text{HIGH-state time period } T_{\text{HIGH}} = 0.69(R_1 + R_2).C \tag{10.7}$$

$$\text{LOW-state time period } T_{\text{LOW}} = 0.69R_2.C \tag{10.8}$$

The relevant waveforms are shown in Fig. 10.10(b). The time period T and frequency f of the output waveform are respectively given by the equations

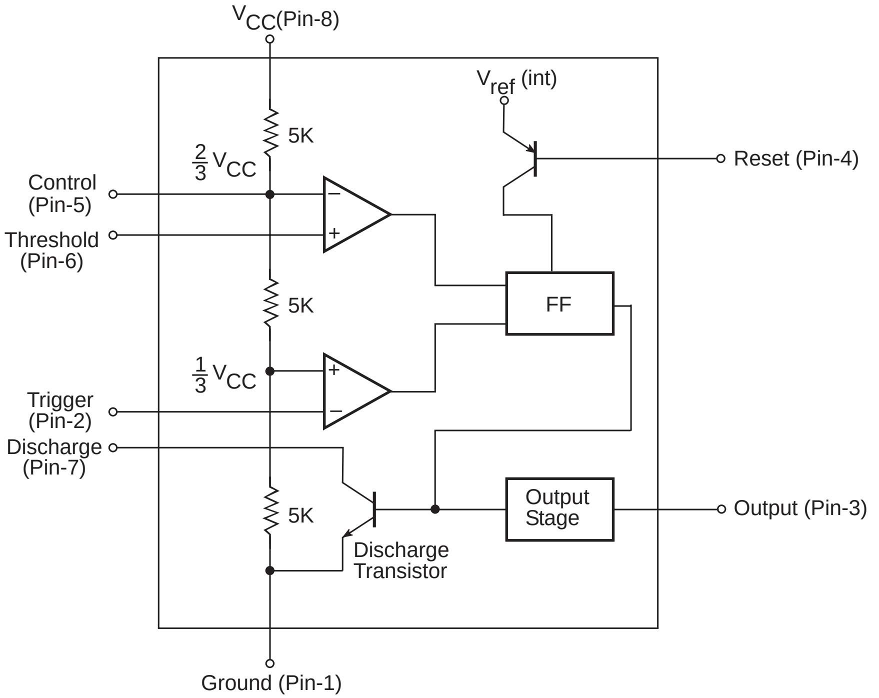


Figure 10.9 Internal schematic of timer IC 555.

$$\text{Time period } T = 0.69(R_1 + 2R_2).C \tag{10.9}$$

$$\text{Frequency } F = 1/[0.69(R_1 + 2R_2).C] \tag{10.10}$$

Remember that, when the astable multivibrator is powered, the first-cycle HIGH-state time period is about 30 % longer, as the capacitor is initially discharged and it charges from 0 (rather than $+V_{CC}/3$) to $+2V_{CC}/3$.

In the case of the astable multivibrator circuit in Fig. 10.10(a), the HIGH-state time period is always greater than the LOW-state time period. Figures 10.10(c) and (d) show two modified circuits where the HIGH-state and LOW-state time periods can be chosen independently. For the astable multivibrator circuits in Fig. 10.10(c) and (d), the two time periods are given by the equations

$$\text{HIGH-state time period} = 0.69R_1.C \tag{10.11}$$

$$\text{LOW-state time period} = 0.69R_2.C \tag{10.12}$$

For $R_1 = R_2 = R$

$$T = 1.38RC \text{ and } f = 1/1.38RC \tag{10.13}$$

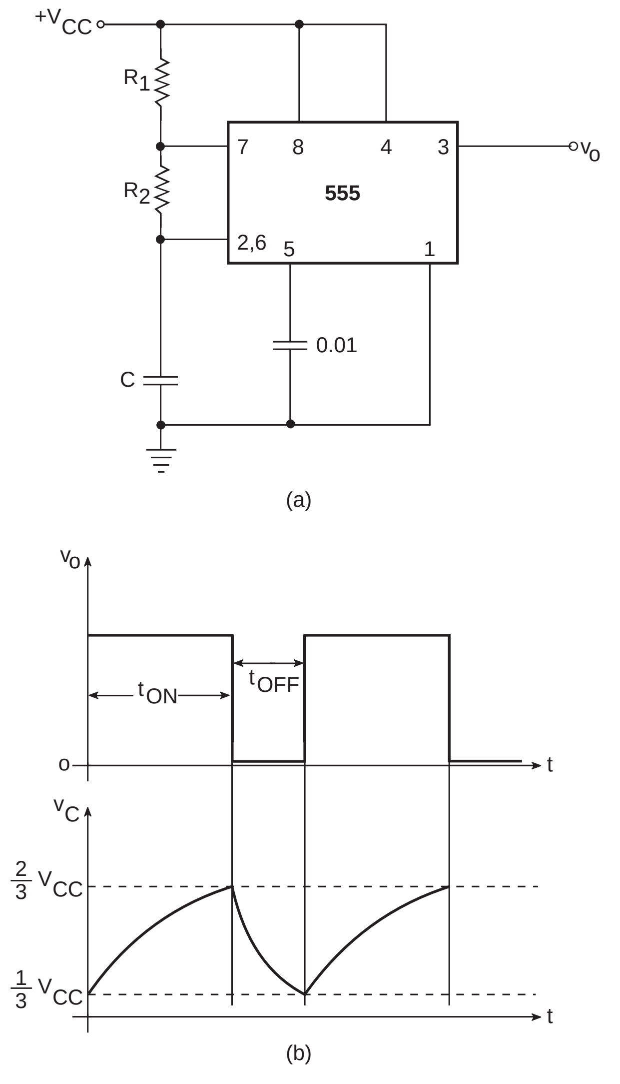
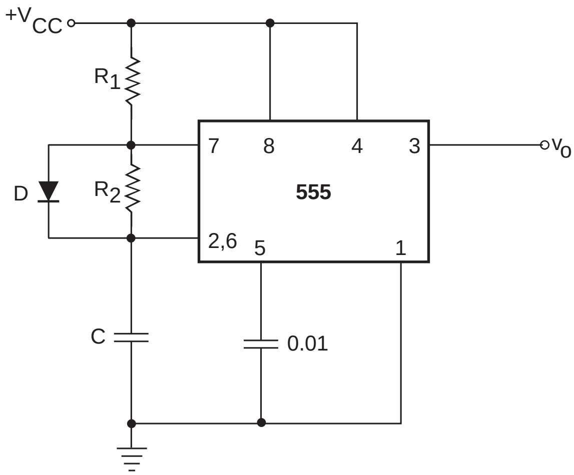


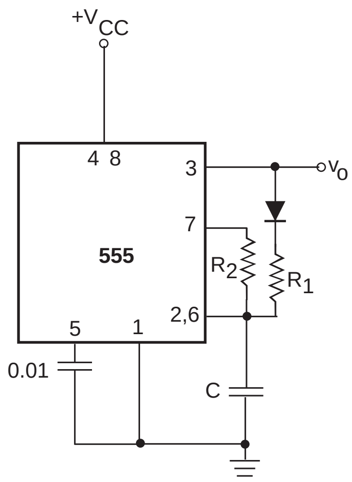
Figure 10.10 (a) Astable multivibrator using timer IC 555, (b) astable multivibrator relevant waveforms and (c, d) modified versions of the astable multivibrator using timer IC 555.

10.2.2.2 Monostable Multivibrator Using Timer IC 555

Figure 10.11(a) shows the basic monostable multivibrator circuit configured around timer 555. A trigger pulse is applied to terminal 2 of the IC, which should initially be kept at $+V_{CC}$. A HIGH at terminal 2 forces the output to the LOW state. A HIGH-to-LOW trigger pulse at terminal 2 holds the output in the HIGH state and simultaneously allows the capacitor to charge from $+V_{CC}$ through R. Remember that a LOW level of the trigger pulse needs to go at least below $+V_{CC}/3$. When the capacitor voltage exceeds $+2V_{CC}/3$, the output goes back to the LOW state. We will need to apply another trigger pulse to



(c)



(d)

Figure 10.10 (*continued*).

terminal 2 to make the output go to the HIGH state again. Every time the timer is appropriately triggered, the output goes to the HIGH state and stays there for the time it takes the capacitor to charge from 0 to $+2V_{CC}/3$. This time period, which equals the monoshot output pulse width, is given by the equation

$$T = 1.1RC \tag{10.14}$$

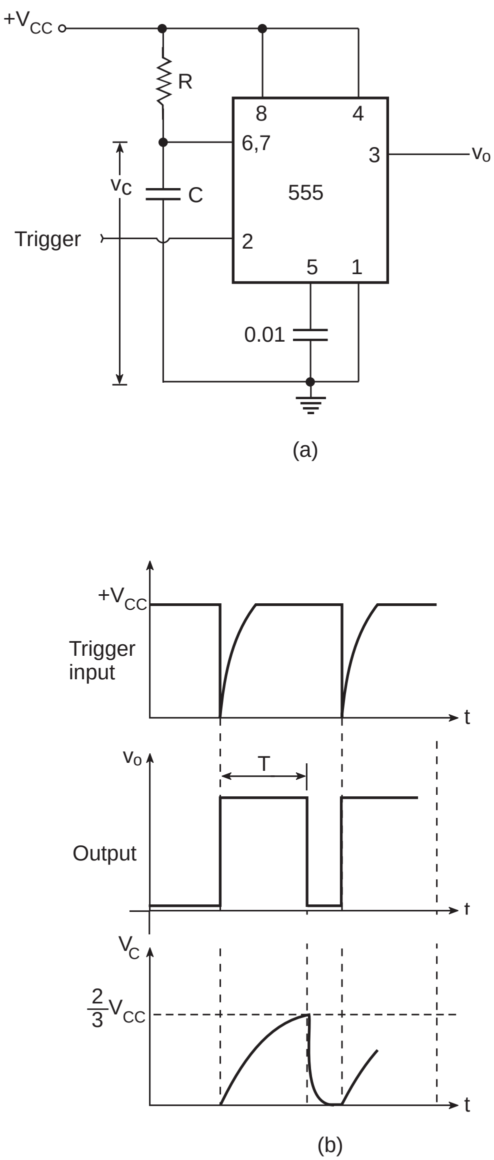


Figure 10.11 (a) Monostable multivibrator using timer 555 and (b) monostable multivibrator relevant waveforms.

Figure 10.11(b) shows the relevant waveforms for the circuit of Fig. 10.11(a).

It is often desirable to trigger a monostable multivibrator either on the trailing (HIGH-to-LOW) or leading (LOW-to-HIGH) edges of the trigger waveform. In order to achieve that, we will need an external circuit between the trigger waveform input and terminal 2 of timer 555. The external circuit ensures that terminal 2 of the IC gets the required trigger pulse corresponding to the desired edge of

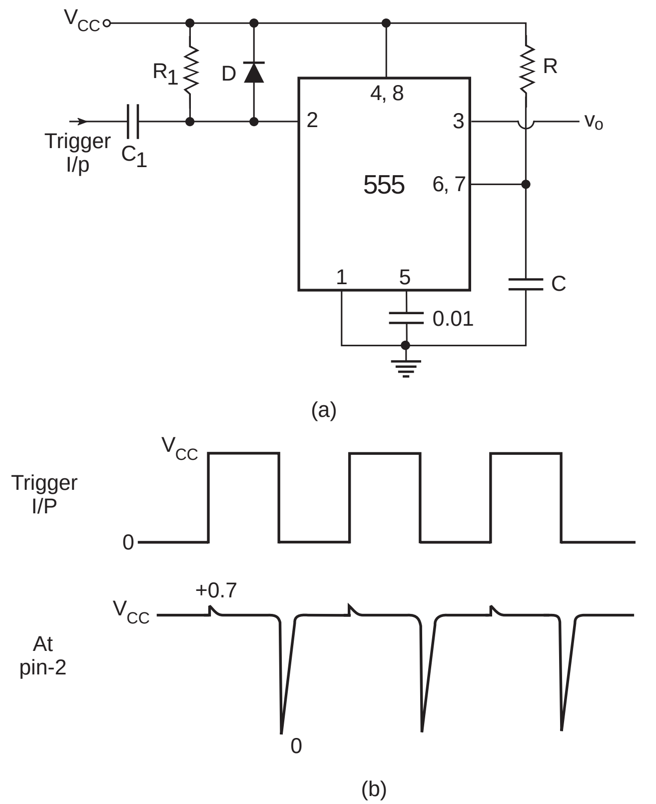


Figure 10.12 555 monoshot triggering on trailing edges.

the trigger waveform. Figure 10.12(a) shows the monoshot configuration that can be triggered on the trailing edges of the trigger waveform. R_1–C_1 constitutes a differentiator circuit. One of the terminals of resistor R_1 is tied to $+V_{CC}$, with the result that the amplitudes of differentiated pulses are $+V_{CC}$ to $+2V_{CC}$ and $+V_{CC}$ to ground, corresponding to the leading and trailing edges of the trigger waveform respectively. Diode D clamps the positive-going differentiated pulses to about +0.7 V. The net result is that the trigger terminal of timer 555 gets the required trigger pulses corresponding to HIGH-to-LOW edges of the trigger waveform. Figure 10.12(b) shows the relevant waveforms.

Figure 10.13(a) shows the monoshot configuration that can be triggered on the leading edges of the trigger waveform. The R_1–C_1 combination constitutes the differentiator producing positive and negative pulses corresponding to LOW-to-HIGH and HIGH-to-LOW transitions of the trigger waveform. Negative pulses are clamped by the diode, and the positive pulses are applied to the base of a transistor switch. The collector terminal of the transistor feeds the required trigger pulses to terminal 2 of the IC. Figure 10.13(b) shows the relevant waveforms.

For the circuits shown in Figs 10.12 and 10.13 to function properly, the values of R_1 and C_1 for the differentiator should be chosen carefully. Firstly, the differentiator time constant should be much smaller than the HIGH time of the trigger waveform for proper differentiation. Secondly, the differentiated pulse width should be less than the expected HIGH time of the monoshot output.

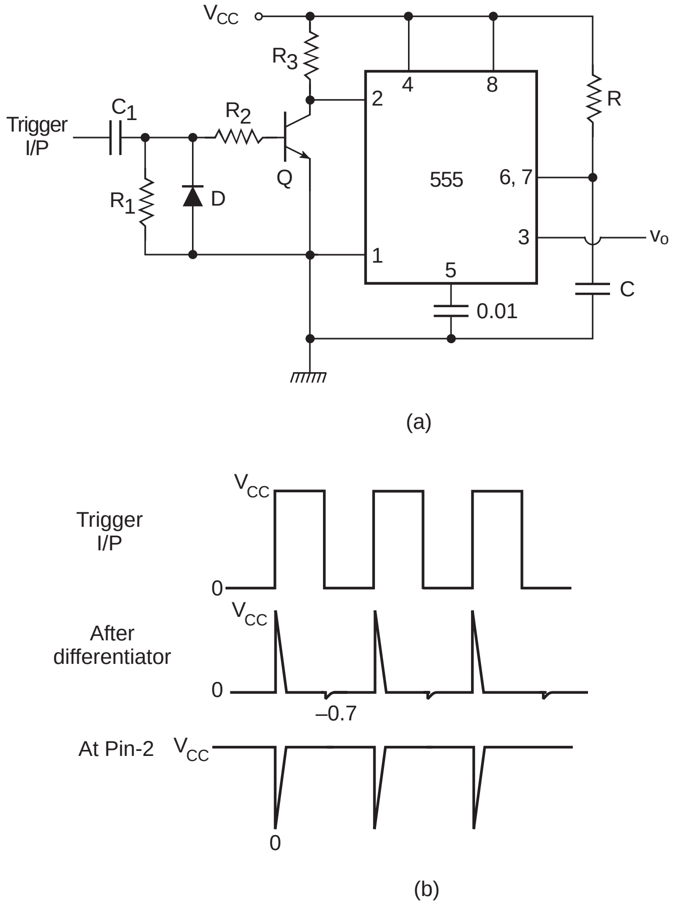


Figure 10.13 555 monoshot triggering on leading edges.

Example 10.1

The pulsed waveform of Fig. 10.14(b) is applied to the RESET terminal of the astable multivibrator circuit of Fig. 10.14(a). Draw the output waveform.

Solution

The circuit shown in Fig. 10.14(a) is an astable multivibrator with a 500 Hz symmetrical waveform applied to its RESET terminal. The RESET terminal is alternately HIGH and LOW for 1.0 ms. When the RESET input is LOW, the output is forced to the LOW state. When the RESET input is HIGH, an astable waveform appears at the output. The HIGH and LOW time periods of the astable multivibrator are determined as follows:

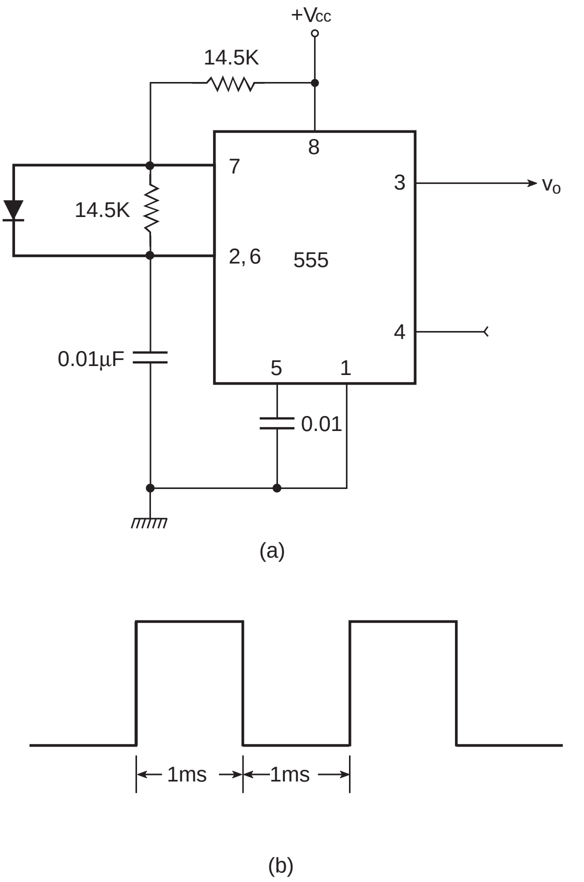


Figure 10.14 Example 10.1.

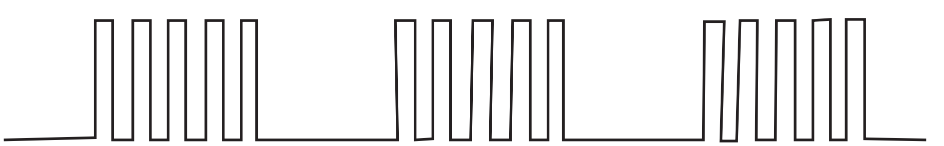


Figure 10.15 Solution to example 10.1.

$$\text{HIGH time} = 0.69 \times 14.5 \times 10^3 \times 0.01 \times 10^{-6} = 100\,\mu\text{s}$$

$$\text{LOW time} = 0.69 \times 14.5 \times 10^3 \times 0.01 \times 10^{-6} = 100\,\mu\text{s}$$

The astable output is thus a 5 kHz symmetrical waveform. Every time the RESET terminal goes to HIGH for 1.0 ms, five cycles of 5 kHz waveform appear at the output. Figure 10.15 shows the output waveform appearing at terminal 3 of the timer IC.

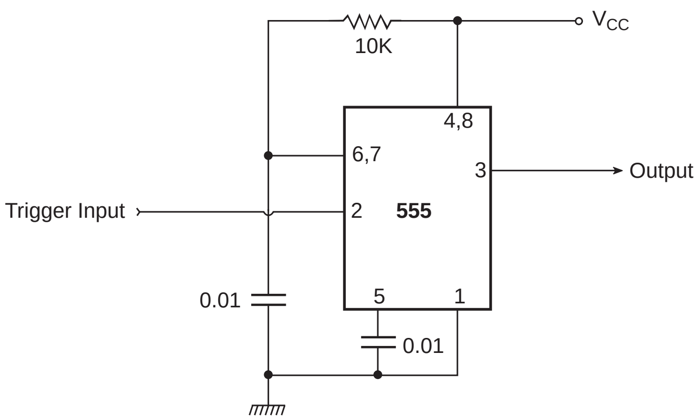


Figure 10.16 Example 10.2.

Example 10.2

Refer to the monostable multivibrator circuit in Fig. 10.16. The trigger terminal (pin 2 of the IC) is driven by a symmetrical pulsed waveform of 10 kHz. Determine the frequency and duty cycle of the output waveform.

Solution

- The frequency of the trigger waveform = 10 kHz.
- The time period between two successive leading or trailing edges = 100 μs.
- The expected pulse width of the monoshot output = $1.1RC = 1.1 \times 10^4 \times 10^{-8} = 110\,\mu\text{s}$.
- The trigger waveform is a symmetrical one; it has HIGH and LOW time periods of 50 μs each. Since the LOW-state time period of the trigger waveform is less than the expected output pulse width, it can successfully trigger the monoshot on its trailing edges.
- Since the time period between two successive trailing edges is 100 μs and the expected output pulse width is 110 μs, only alternate trailing edges of the trigger waveform will trigger the monoshot.
- The frequency of the output waveform = 10/2 = 5 kHz.
- The time period of the output waveform = $1/(5 \times 10^3) = 200$ μs.
- Therefore, the duty cycle of the output waveform = 110/200 = 0.55.

10.3 *R-S* Flip-Flop

A flip-flop, as stated earlier, is a bistable circuit. Both of its output states are stable. The circuit remains in a particular output state indefinitely until something is done to change that output status. Referring to the bistable multivibrator circuit discussed earlier, these two states were those of the output transistor in saturation (representing a LOW output) and in cut-off (representing a HIGH output). If the LOW and HIGH outputs are respectively regarded as '0' and '1', then the output can either be a '0' or a '1'. Since either a '0' or a '1' can be held indefinitely until the circuit is appropriately triggered to go to the other state, the circuit is said to have memory. It is capable of storing one binary digit or one bit of digital information. Also, if we recall the functioning of the bistable multivibrator circuit, we find that, when one of the transistors was in saturation, the other was in cut-off. This implies that, if we had taken outputs from the collectors of both transistors, then the two outputs would be complementary. In the flip-flops of various types that are available in IC form, we will see that all these devices offer complementary outputs usually designated as Q and $\overline{Q}$.

The *R-S* flip-flop is the most basic of all flip-flops. The letters '*R*' and '*S*' here stand for RESET and SET. When the flip-flop is SET, its Q output goes to a '1' state, and when it is RESET it goes to a '0' state. The $\overline{Q}$ output is the complement of the Q output at all times.

10.3.1 R-S Flip-Flop with Active LOW Inputs

Figure 10.17(a) shows a NAND gate implementation of an *R-S* flip-flop with active LOW inputs. The two NAND gates are cross-coupled. That is, the output of NAND 1 is fed back to one of the inputs of NAND 2, and the output of NAND 2 is fed back to one of the inputs of NAND 1. The remaining inputs of NAND 1 and NAND 2 are the S and R inputs. The outputs of NAND 1 and NAND 2 are respectively Q and $\overline{Q}$ outputs.

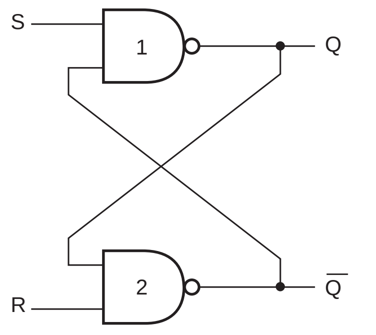
The fact that this configuration follows the function table of Fig. 10.17(c) can be explained. We will look at different entries of the function table, one at a time.

Let us take the case of $R = S = 1$ (the first entry in the function table). We will prove that, for $R = S = 1$, the Q output remains in its existing state. In the truth table, Q_n represents the existing state and Q_{n+1} represents the state of the flip-flop after it has been triggered by an appropriate pulse at the R or S input. Let us assume that $Q = 0$ initially. This '0' state fed back to one of the inputs of gate 2 ensures that $\overline{Q} = 1$. The '1' state of $\overline{Q}$ fed back to one of the inputs of gate 1 along with $S = 1$ ensures that $Q = 0$. Thus, $R = S = 1$ holds the existing stage. Now, if Q was initially in the '1' state and not the '0' state, this '1' fed back to one of the inputs of gate 2 along with $R = 1$ forces $\overline{Q}$ to be in the '0' state. The '0' state, when fed back to one of the inputs of gate 1, ensures that Q remains in its existing state of logic '1'. Thus, whatever the state of Q, $R = S = 1$ holds the existing state.

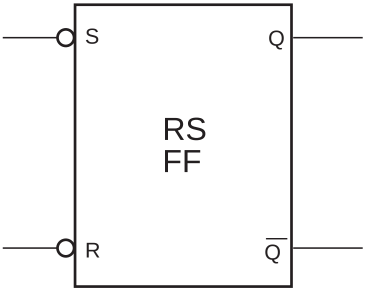
Let us now look at the second entry of the function table where $S = 0$ and $R = 1$. We can see that such an input combination forces the Q output to the '1' state. On similar lines, the input combination $S = 1$ and $R = 0$ (third entry of the truth table) forces the Q output to the '0' state. It would be interesting to analyse what happens when $S = R = 0$. This implies that both Q and $\overline{Q}$ outputs should go to the '1' state, as one of the inputs of a NAND gate being a logic '0' should force its output to the logic '1' state irrespective of the status of the other input. This is an undesired state as Q and $\overline{Q}$ outputs are to be the complement of each other. The input condition (i.e. $R = S = 0$) that causes such a situation is therefore considered to be an invalid condition and is forbidden. Figure 10.17(b) shows the logic symbol of such a flip-flop. The R and S inputs here have been shown as active LOW inputs, which is obvious as this flip-flop of Fig. 10.17(a) is SET (that is, $Q = 1$) when $S = 0$ and RESET (that is, $Q = 0$) when $R = 0$. Thus, R and S are active when LOW. The term CLEAR input is also used sometimes in place of RESET. The operation of the *R-S* flip-flop of Fig. 10.17(a) can be summarized as follows:

1. SET = RESET = 1 is the normal resting condition of the flip-flop. It has no effect on the output state of the flip-flop. Both Q and $\overline{Q}$ outputs remain in the logic state they were in prior to this input condition.
2. SET = 0 and RESET = 1 sets the flip-flop. Q and $\overline{Q}$ respectively go to the '1' and '0' state.
3. SET = 1 and RESET = 0 resets or clears the flip-flop. Q and $\overline{Q}$ respectively go to the '0' and '1' state.
4. SET = RESET = 0 is forbidden as such a condition tries to set (that is, $Q = 1$) and reset (that is, $\overline{Q} = 1$) the flip-flop at the same time. To be more precise, SET and RESET inputs in the R-S flip-flop cannot be active at the same time.

The *R-S* flip-flop of Fig. 10.17(a) is also referred to as an *R-S* latch. This is because any combination at the inputs immediately manifests itself at the output as per the truth table.



(a)



(b)

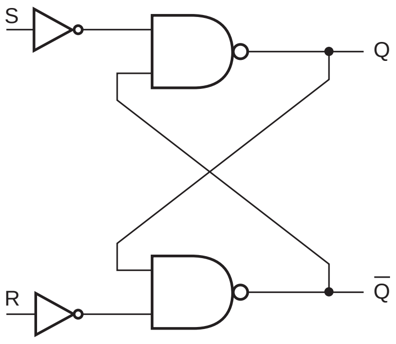
Operation Mode	S	R	Q_{n+1}
No change	1	1	Q_n
SET	0	1	1
RESET	1	0	0
Forbidden	0	0	—

(c)

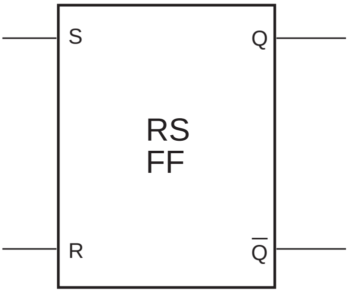
Figure 10.17 *R-S* flip-flop with active LOW inputs.

10.3.2 R-S Flip-Flop with Active HIGH Inputs

Figure 10.18(a) shows another NAND gate implementation of the *R-S* flip-flop. Figures 10.18(b) and (c) respectively show its circuit symbol and function table. Such a circuit would have active HIGH inputs. The input combination $R = S = 1$ would be forbidden as SET and RESET inputs in an *R-S* flip-flop cannot be active at the same time.



(a)



(b)

Operation Mode	S	R	Q_{n+1}
No change	0	0	Q_n
SET	1	0	1
RESET	0	1	0
Forbidden	1	1	—

(c)

Figure 10.18 *R-S* flip-flop with active HIGH inputs.

The *R-S* flip-flops (or latches) of Figs 10.17(a) and 10.18 (a) may also be implemented with NOR gates. The NOR gate counterparts of Fig. 10.17(a) and Fig. 10.18(a) are respectively shown in Figs 10.19(a) and (b).

So far we have discussed the operation of an *R-S* flip-flop with the help of its logic diagram and the function table on lines similar to the case of combinational circuits. We do, however, appreciate that a sequential circuit would be better explained if we expressed its output (immediately after it was clocked) in terms of its present output and its inputs. The function tables of Figs 10.17(c) and 10.18(c) may be redrawn as shown in Figs 10.20(a) and (b) respectively. This new form of representation is known as the characteristic table. Having done this, we could even write simplified Boolean expressions,

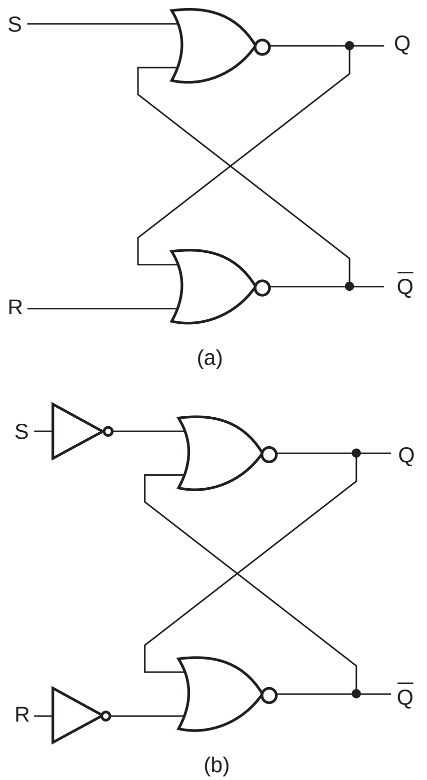


Figure 10.19 NOR implementation of an *R-S* flip-flop.

called characteristic equations, using any of the minimization techniques, such as Karnaugh mapping. The K-maps for the characteristic tables of Figs 10.20(a) and (b) are given in Figs 10.20(c) and (d) respectively. Characteristic equations for *R-S* flip-flops with active LOW and active HIGH inputs are given by the equations

$$Q_{n+1} = \overline{S} + R.Q_n \quad \text{and} \quad S + R = 1 \tag{10.15}$$

$$Q_{n+1} = S + \overline{R}.Q_n \quad \text{and} \quad S.R = 0 \tag{10.16}$$

$S + R = 1$ indicates that $R = S = 0$ is a prohibited entry. Similarly, $S.R = 0$ only indicates that $R = S = 1$ is a prohibited entry.

10.3.3 Clocked R-S Flip-Flop

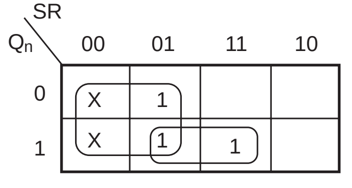
In the case of a clocked *R-S* flip-flop, or for that matter any clocked flip-flop, the outputs change states as per the inputs only on the occurrence of a clock pulse. The clocked flip-flop could be a level-triggered one or an edge-triggered one. The two types are discussed in the next section. For the

Qn	S	R	Q_{n+1}
0	0	0	Indeter
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	Indeter
1	0	1	1
1	1	0	0
1	1	1	1

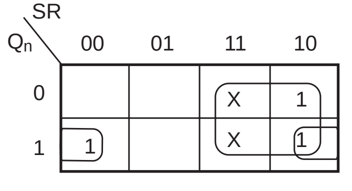
(a)

Qn	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Indeter
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Indeter

(b)



(c)

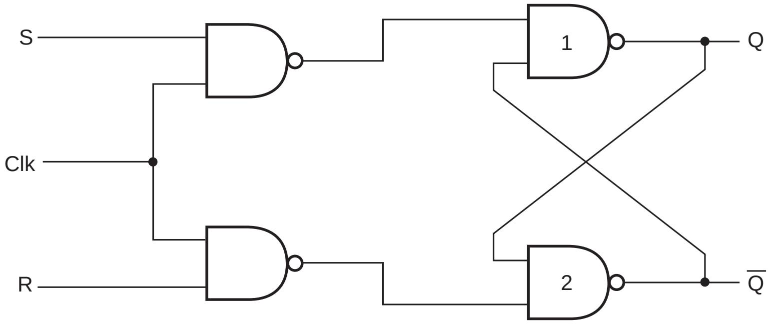


(d)

Figure 10.20 (a) Characteristic table of an *R-S* flip-flop with active LOW inputs, (b) the characteristic table of an *R-S* flip-flop with active HIGH inputs, (c) the K-map solution of an *R-S* flip-flop with active LOW inputs and (d) the K-map solution of an *R-S* flip-flop with active HIGH inputs.

time being, let us first see how the flip-flop of the previous section can be transformed into a clocked flip-flop. Figure 10.21(a) shows the logic implementation of a clocked flip-flop that has active HIGH inputs. The function table for the same is shown in Fig. 10.21(b) and is self-explanatory.

The basic flip-flop is the same as that shown in Fig. 10.17(a). The two NAND gates at the input have been used to couple the R and S inputs to the flip-flop inputs under the control of the clock signal. When the clock signal is HIGH, the two NAND gates are enabled and the S and R inputs are passed on to flip-flop inputs with their status complemented. The outputs can now change states as per the status of R and S at the flip-flop inputs. For instance, when $S = 1$ and $R = 0$ it will be passed on as 0 and 1 respectively when the clock is HIGH. When the clock is LOW, the two NAND gates produce a '1' at their outputs, irrespective of the S and R status. This produces a logic '1' at both inputs of the flip-flop, with the result that there is no effect on the output states. Figure 10.22(a) shows the clocked R-S flip-flop with active LOW R and S inputs. The logic implementation here is a modification of the basic R-S flip-flop in Fig. 10.18(a). The truth table of this flip-flop, as given in Fig. 10.22(b), is self-explanatory.

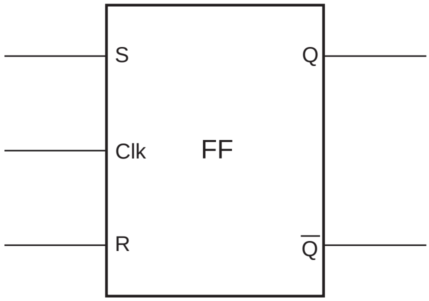


(a)

S	R	Clk	Q_{n+1}
0	0	0	Q_n
0	0	1	Q_n
0	1	0	Q_n
0	1	1	0
1	0	0	Q_n
1	0	1	1
1	1	0	Q_n
1	1	1	Invalid

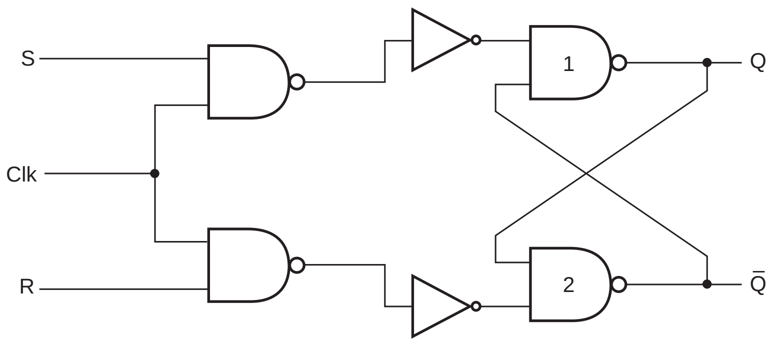
(b)

Figure 10.21 Clocked R-S flip-flop with active HIGH inputs.



(c)

Figure 10.21 (*continued*).

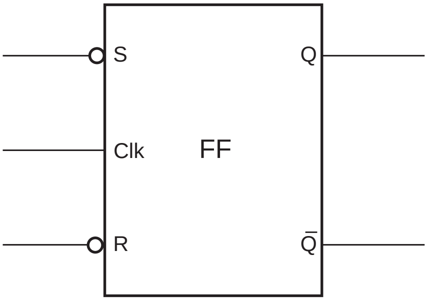


(a)

S	R	Clk	Q_{n+1}
0	0	0	Q_n
0	0	1	Invalid
0	1	0	Q_n
0	1	1	1
1	0	0	Q_n
1	0	1	0
1	1	0	Q_n
1	1	1	Q_n

(b)

Figure 10.22 Clocked *R-S* flip-flop with active LOW inputs.



(c)

Figure 10.22 (*continued*).

10.4 Level-Triggered and Edge-Triggered Flip-Flops

In a *level-triggered* flip-flop, the output responds to the data present at the inputs during the time the clock pulse level is HIGH (or LOW). That is, any changes at the input during the time the clock is active (HIGH or LOW) are reflected at the output as per its function table. The clocked *R-S* flip-flop described in the preceding paragraphs is a level-triggered flip-flop that is active when the clock is HIGH.

In an *edge-triggered* flip-flop, the output responds to the data at the inputs only on LOW-to-HIGH or HIGH-to-LOW transition of the clock signal. The flip-flop in the two cases is referred to as positive edge triggered and negative edge triggered respectively. Any changes in the input during the time the clock pulse is HIGH (or LOW) do not have any effect on the output. In the case of an edge-triggered flip-flop, an edge detector circuit transforms the clock input into a very narrow pulse that is a few nanoseconds wide. This narrow pulse coincides with either LOW-to-HIGH or HIGH-to-LOW transition of the clock input, depending upon whether it is a positive edge-triggered flip-flop or a negative edge-triggered flip-flop. This pulse is so narrow that the operation of the flip–flop can be considered to have occurred on the edge itself.

Figure 10.23 shows the clocked *R-S* flip-flop of Fig. 10.21 with the edge detector block incorporated in the clock circuit. Figures 10.24 (a) and (b) respectively show typical edge detector circuits for positive

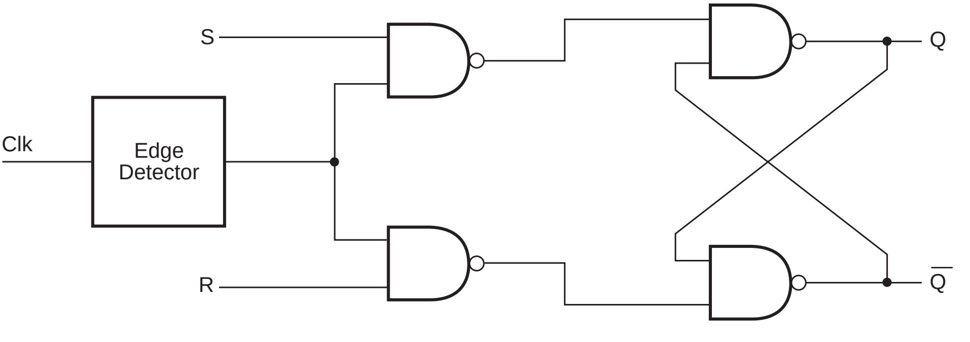


Figure 10.23 Edge-triggered *R-S* flip-flop.

and negative edge triggering. The width of the narrow pulse generated by this edge detector circuit is equal to the propagation delay of the inverter. Figure 10.25 shows the circuit symbol for the flip-flop of Fig. 10.23 for the positive edge-triggered mode [Fig. 10.25(a)] and the negative edge-triggered mode [Fig. 10.25(b)].

10.5 *J-K* Flip-Flop

A *J-K* flip-flop behaves in the same fashion as an *R-S* flip-flop except for one of the entries in the function table. In the case of an *R-S* flip-flop, the input combination $S = R = 1$ (in the case of a flip-flop with active HIGH inputs) and the input combination $S = R = 0$ (in the case of a flip-flop with active LOW inputs) are prohibited. In the case of a *J-K* flip-flop with active HIGH inputs, the output of the flip-flop toggles, that is, it goes to the other state, for $J = K = 1$. The output toggles for $J = K = 0$ in the case of the flip-flop having active LOW inputs. Thus, a *J-K* flip-flop overcomes the problem of a forbidden input combination of the *R-S* flip-flop. Figures 10.26(a) and (b) respectively show the circuit symbol of level-triggered *J-K* flip-flops with active HIGH and active LOW inputs, along with their function tables. Figure 10.27 shows the realization of a *J-K* flip-flop with an *R-S* flip-flop.

The characteristic tables for a *J-K* flip-flop with active HIGH *J* and *K* inputs and a *J-K* flip-flop with active LOW *J* and *K* inputs are respectively shown in Figs 10.28(a) and (b). The corresponding Karnaugh maps are shown in Fig. 10.28(c) for the characteristics table of Fig. 10.28(a) and in Fig. 10.28(d) for the characteristic table of Fig. 10.28(b). The characteristic equations for the Karnaugh maps of Figs 10.28(c) and (d) are respectively

$$Q_{n+1} = J.\overline{Q_n} + \overline{K}.Q_n \tag{10.17}$$

$$Q_{n+1} = \overline{J}.\overline{Q_n} + K.Q_n \tag{10.18}$$

10.5.1 J-K Flip-Flop with PRESET and CLEAR Inputs

It is often necessary to clear a flip-flop to a logic '0' state ($Q_n = 0$) or preset it to a logic '1' state ($Q_n = 1$). An example of how this is realized is shown in Fig. 10.29(a). The flip-flop is cleared (that is, $Q_n = 0$) whenever the CLEAR input is '0' and the PRESET input is '1'. The flip-flop is preset to the logic '1' state whenever the PRESET input is '0' and the CLEAR input is '1'. Here, the CLEAR and PRESET inputs are active when LOW. Figure 10.29(b) shows the circuit symbol of this presettable, clearable, clocked *J-K* flip-flop. Figure 10.29(c) shows the function table of such a flip-flop. It is evident from the function table that, whenever the PRESET input is active, the output goes to the '1' state irrespective of the status of the clock, *J* and *K* inputs. Similarly, when the flip-flop is cleared, that is, the CLEAR input is active, the output goes to the '0' state irrespective of the status of the clock, *J* and *K* inputs. In a flip-flop of this type, both PRESET and CLEAR inputs should not be made active at the same time.

10.5.2 Master–Slave Flip-Flops

Whenever the width of the pulse clocking the flip-flop is greater than the propagation delay of the flip-flop, the change in state at the output is not reliable. In the case of edge-triggered flip-flops, this pulse width would be the trigger pulse width generated by the edge detector portion of the flip-flop

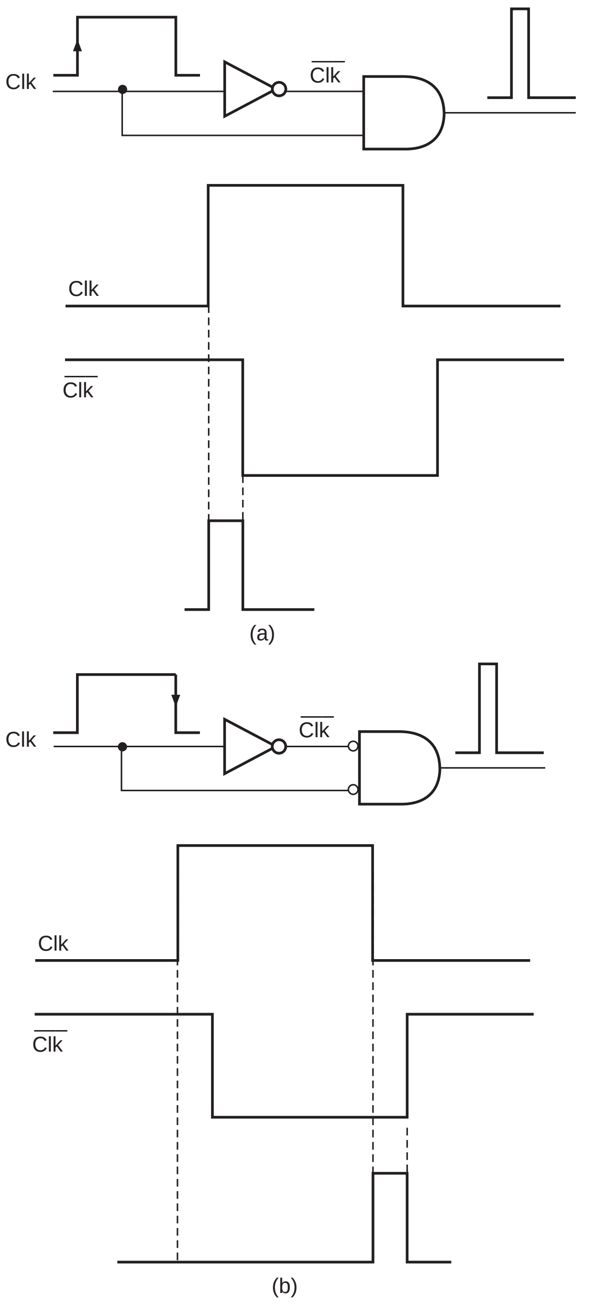


Figure 10.24 (a) Positive edge-triggered edge detector circuits and (b) negative edge-triggered edge detector circuits.

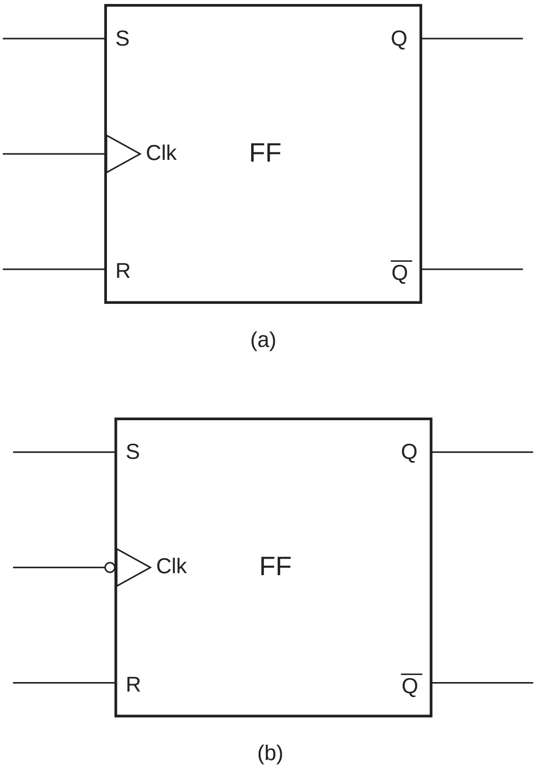
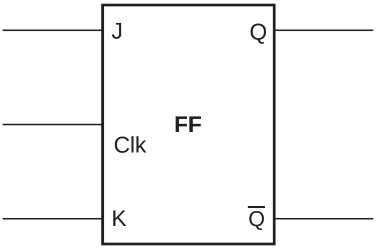


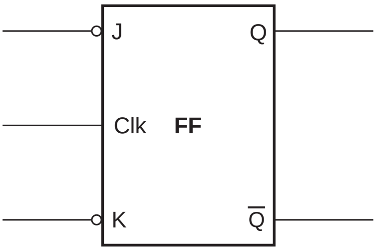
Figure 10.25 (a) Circuit symbol of a positive edge-triggered *R-S* flip-flop and (b) the circuit symbol of a negative edge-triggered *R-S* flip-flop.

and not the pulse width of the input clock signal. This phenomenon is referred to as the *race problem*. As the propagation delays are normally very small, the likelihood of the occurrence of a race condition is reasonably high. One way to get over this problem is to use a *master–slave* configuration. Figure 10.30(a) shows a master–slave flip-flop constructed with two *J-K* flip-flops. The first flip-flop is called the master flip-flop and the second is called the slave. The clock to the slave flip-flop is the complement of the clock to the master flip-flop. When the clock pulse is present, the master flip-flop is enabled while the slave flip-flop is disabled. As a result, the master flip-flop can change state while the slave flip-flop cannot. When the clock goes LOW, the master flip-flop gets disabled while the slave flip-flop is enabled. Therefore, the slave *J-K* flip-flop changes state as per the logic states at its *J* and *K* inputs. The contents of the master flip-flop are therefore transferred to the slave flip-flop, and the master flip-flop, being disabled, can acquire new inputs without affecting the output. As would be clear from the description above, a master–slave flip-flop is a pulse-triggered flip-flop and not an edge-triggered one. Figure 10.30(b) shows the truth table of a master–slave *J-K* flip-flop with active LOW PRESET and CLEAR inputs and active HIGH *J* and *K* inputs. The master–slave configuration has become obsolete. The newer IC technologies such as 74LS, 74AS, 74ALS, 74HC and 74HCT do not have master–slave flip-flops in their series.



Operation Mode	J	K	Clk	Q_{n+1}
SET	1	0	1	1
RESET	0	1	1	0
NO CHANGE	0	0	1	Q_n
TOGGLE	1	1	1	$\overline{Q_n}$

(a)



Operation Mode	J	K	Clk	Q_{n+1}
SET	0	1	1	1
RESET	1	0	1	0
NO CHANGE	1	1	1	Q_n
TOGGLE	0	0	1	$\overline{Q_n}$

(b)

Figure 10.26 (a) *J-K* flip-flop active HIGH inputs and (b) *J-K* flip-flop active LOW inputs.

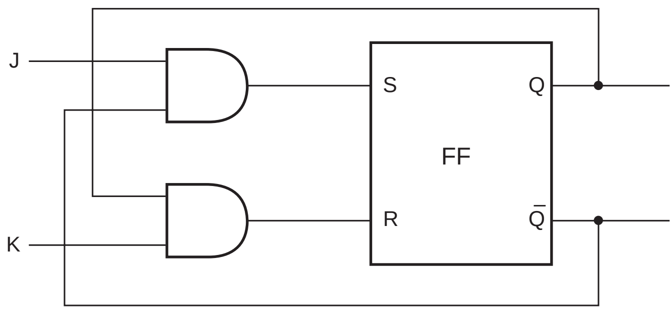


Figure 10.27 Realization of a *J-K* flip-flop using an *R-S* flip-flop.

Example 10.3

Draw the circuit symbol of the flip-flop represented by the function table of Fig. 10.31(a).

Solution

The first three entries of the function table indicate that the *J-K* flip-flop has active HIGH PRESET and CLEAR inputs. Referring to the fourth and fifth entries of the function table, it has active LOW *J* and *K* inputs. The seventh row of the function table confirms this. The output responds to positive (LOW-to-HIGH) edges of the clock input. Thus, the flip-flop represented by the given function table is a presettable, clearable, positive edge-triggered flip-flop with active HIGH PRESET and CLEAR

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(a)

Q_n	J	K	Q_{n+1}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(b)

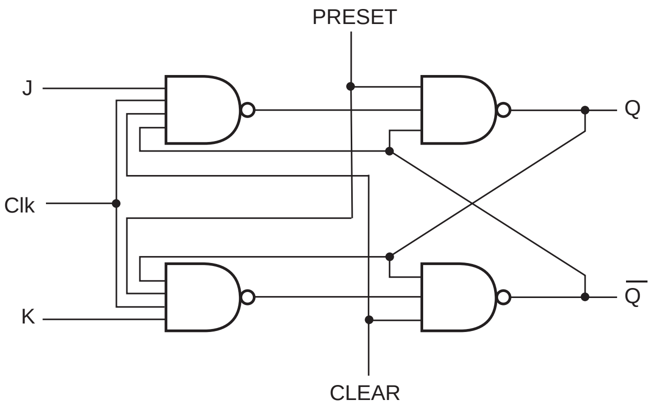
Q_n \ JK	00	01	11	10
0			1	1
1	1			1

(c)

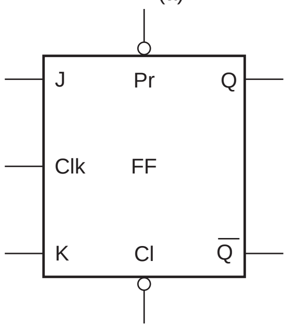
Q_n \ JK	00	01	11	10
0	1	1		
1		1	1	

(d)

Figure 10.28 (a) Characteristic table of a *J*-*K* flip-flop with active HIGH inputs, (b) the characteristic table of a *J*-*K* flip-flop with active LOW inputs, (c) the K-map solution of a *J*-*K* flip-flop with active HIGH inputs and (d) the K-map solution of a *J*-*K* flip-flop with active LOW inputs.



(a)

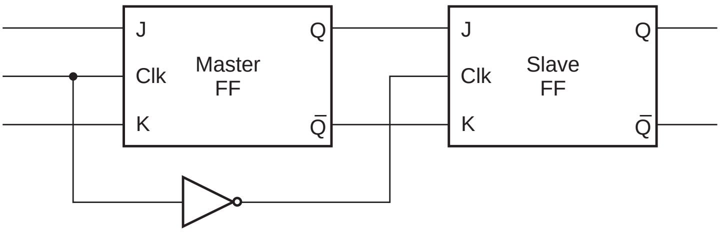


(b)

PR	CL	CLK	J	K	Q_{n+1}	$\overline{Q}_{n+1}$
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	---	---
1	1	1	0	0	Q_n	$\overline{Q}_n$
1	1	1	1	0	1	0
1	1	1	0	1	0	1
1	1	1	1	1	Toggle	
1	1	0	X	X	Q_n	$\overline{Q}_n$

(c)

Figure 10.29 *J-K* flip-flop with PRESET and CLEAR inputs.



(a)

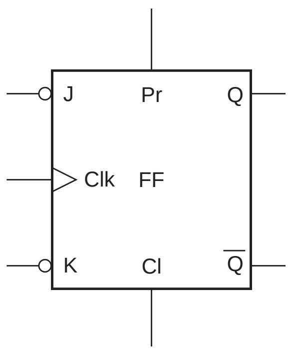
PR	CLR	CLK	J	K	Q_{n+1}	$\overline{Q_{n+1}}$
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	Unstable	
1	1	⎍	0	0	Q_n	$\overline{Q_n}$
1	1	⎍	1	0	1	0
1	1	⎍	0	1	0	1
1	1	⎍	1	1	Toggle	

(b)

Figure 10.30 Master–slave flip-flop.

PR	CLR	CLK	J	K	Q_{n+1}	$\overline{Q_{n+1}}$
1	0	X	X	X	1	0
0	1	X	X	X	0	1
1	1	X	X	X	Unstable	
0	0	↑	0	1	1	0
0	0	↑	1	0	0	1
0	0	↑	1	1	Q_n	$\overline{Q_n}$
0	0	↑	0	0	Toggle	

(a)



(b)

Figure 10.31 Example 10.3.

and active LOW J and K inputs. Figure 10.31(b) shows the circuit symbol of the flip-flop represented by this truth table.

Example 10.4

The 100 kHz square waveform of Fig. 10.32(a) is applied to the clock input of the flip-flops shown in Figs. 10.32(b) and (c). If the Q output is initially '0', draw the Q output waveform in the two cases. Also, determine the frequency of the Q output in the two cases.

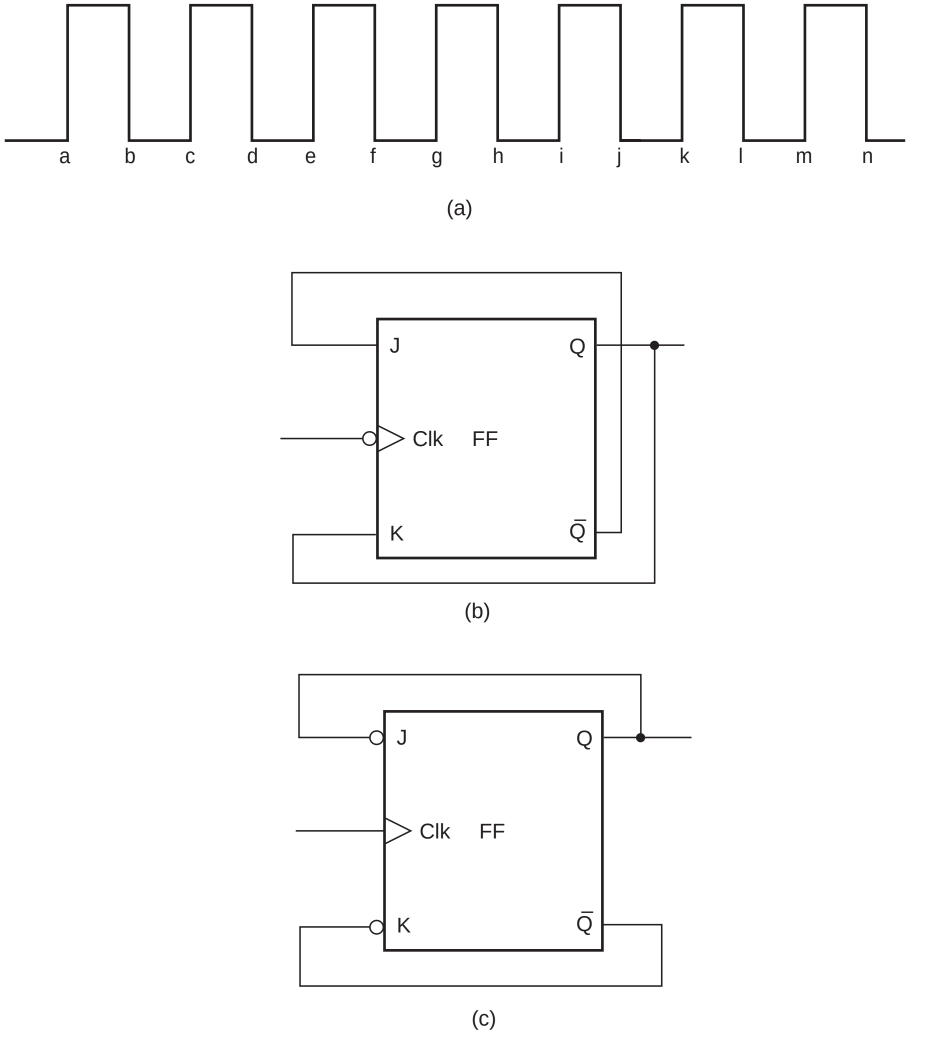


Figure 10.32 Example 10.4.

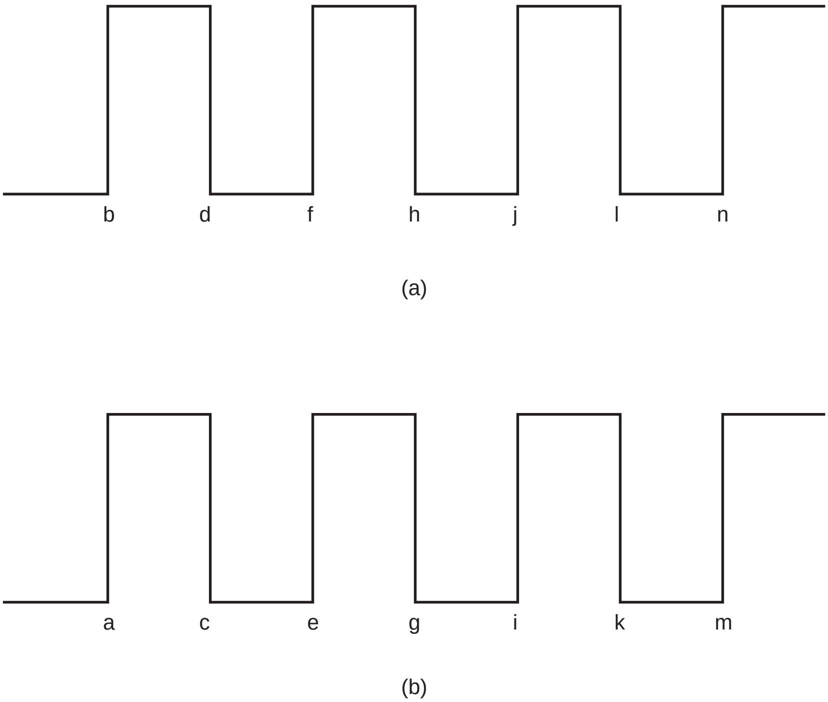


Figure 10.33 Solution to example 10.4.

Solution

Refer to the flip-flop of Fig. 10.32(b). Q is initially '0'. This makes the J and K inputs be initially '1' and '0' respectively. With the first trailing edge of the clock input, Q goes to the '1' state. Thus, J and K acquire a logic status of '0' and '1' respectively. With the next trailing edge of the clock input, Q goes to logic '0'. This process continues, and Q alternately becomes '1' and '0'. The Q output waveform for this case is shown in Fig. 10.33(a). In the case of the flip-flop of Fig. 10.32(c), J and K are initially '0' and '1' respectively. Thus, J is active. With the first leading edge of the clock input, Q and therefore J go to the logic '1' state. The second leading edge forces Q to go to the logic '0' state as now it is the K input that is in the logic '0' state and active. This circuit also behaves in the same way as the flip-flop of Fig. 10.32(b). The output goes alternately to the logic '0' and '1' state. However, the transitions occur on the leading edge of the clock input. Figure 10.33(b) shows the Q output waveform for this case. The frequency of the Q output waveform in the two cases is equal to half the frequency of the clock input, for obvious reasons, and is therefore 50 kHz.

10.6 Toggle Flip-Flop (T Flip-Flop)

The output of a *toggle flip-flop*, also called a T flip-flop, changes state every time it is triggered at its T input, called the toggle input. That is, the output becomes '1' if it was '0' and '0' if it was '1'.

Figures 10.34(a) and (b) respectively show the circuit symbols of positive edge-triggered and negative edge-triggered T flip-flops, along with their function tables.

If we consider the T input as active when HIGH, the characteristic table of such a flip-flop is shown in Fig. 10.34(c). If the T input were active when LOW, then the characteristic table would be as shown in Fig. 10.34(d). The Karnaugh maps for the characteristic tables of Figs 10.34(c) and (d) are shown in Figs 10.34(e) and (f) respectively. The characteristic equations as written from the Karnaugh maps are as follows:

$$Q_{n+1} = T.\overline{Q_n} + \overline{T}.Q_n \tag{10.19}$$

$$Q_{n+1} = \overline{T}.\overline{Q_n} + T.Q_n \tag{10.20}$$

It is obvious from the operational principle of the T flip-flop that the frequency of the signal at the Q output is half the frequency of the signal applied at the T input. A cascaded arrangement of nT flip-flops, where the output of one flip-flop is connected to the T input of the following flip-flop, can be used to divide the input signal frequency by a factor of 2^n. Figure 10.35 shows a divide-by-16 circuit built around a cascaded arrangement of four T flip-flops.

10.6.1 J-K Flip-Flop as a Toggle Flip-Flop

If we recall the function table of a J-K flip-flop, we will see that, when both J and K inputs of the flip-flop are tied to their active level ('1' level if J and K are active when HIGH, and '0' level when J and K are active when LOW), the flip-flop behaves like a toggle flip-flop, with its clock input serving as the T input. In fact, the J-K flip-flop can be used to construct any other flip-flop. That is why it is also sometimes referred to as a *universal flip-flop*. Figure 10.36 shows the use of a J-K flip-flop as a T flip-flop.

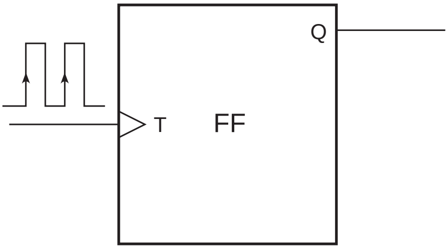
Example 10.5

Refer to the cascaded arrangement of two T flip-flops in Fig. 10.37(a). Draw the Q output waveform for the given input signal. If the time period of the input signal is 10 ms, find the frequency of the output signal? If, in the flip-flop arrangement of Fig. 10.37(a), FF-2 were positive edge triggered, draw the Q output waveform.

Solution

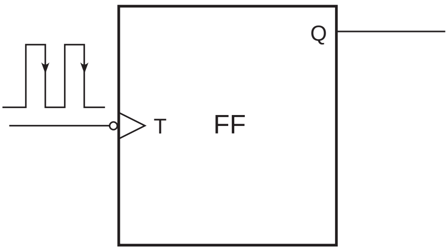
The Q output waveform is shown in Fig. 10.37(b) along with the Q output of FF-1. The output of the first T flip-flop changes state for every negative-going edge of the input clock waveform. Its frequency is therefore half the input signal frequency. The output of the first flip-flop acts as the clock input for the second T flip-flop in the cascade arrangement. The second flip-flop, too, toggles for every negative-going edge of the waveform appearing at its input. The final output thus has a frequency that is one-fourth of the input signal frequency:

- Now the time period of the input signal = 10 ms.
- Therefore, the frequency = 100 kHz.
- The frequency of the output signal = 25 kHz.



T	Q_n	Q_{n+1}
↑	0	1
↑	1	0

(a)



T	Q_n	Q_{n+1}
↓	0	1
↓	1	0

(b)

Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

(c)

Q_n	T	Q_{n+1}
0	0	1
0	1	0
1	0	0
1	1	1

(d)

Figure 10.34 (a) Positive edge-triggered toggle flip-flop, (b) a negative edge-triggered toggle flip-flop, (c, d) characteristic tables of level-triggered toggle flip-flops and (e, f) Karnaugh maps for characteristic tables (c, d).

Q_n \ T	0	1
0		1
1	1	

(e)

Q_n \ T	0	1
0	1	
1		1

(f)

Figure 10.34 (*continued*).

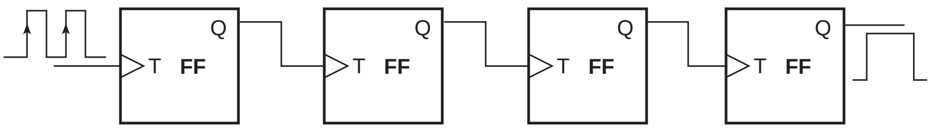


Figure 10.35 Cascade arrangement of *T* flip-flops.

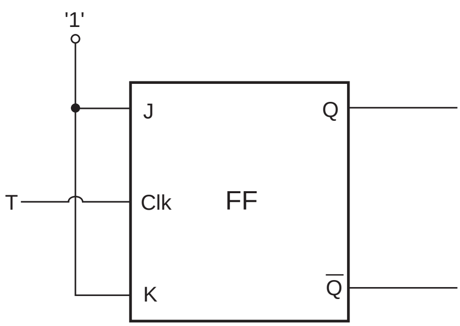


Figure 10.36 *J*-*K* flip-flop as a *T* flip-flop.

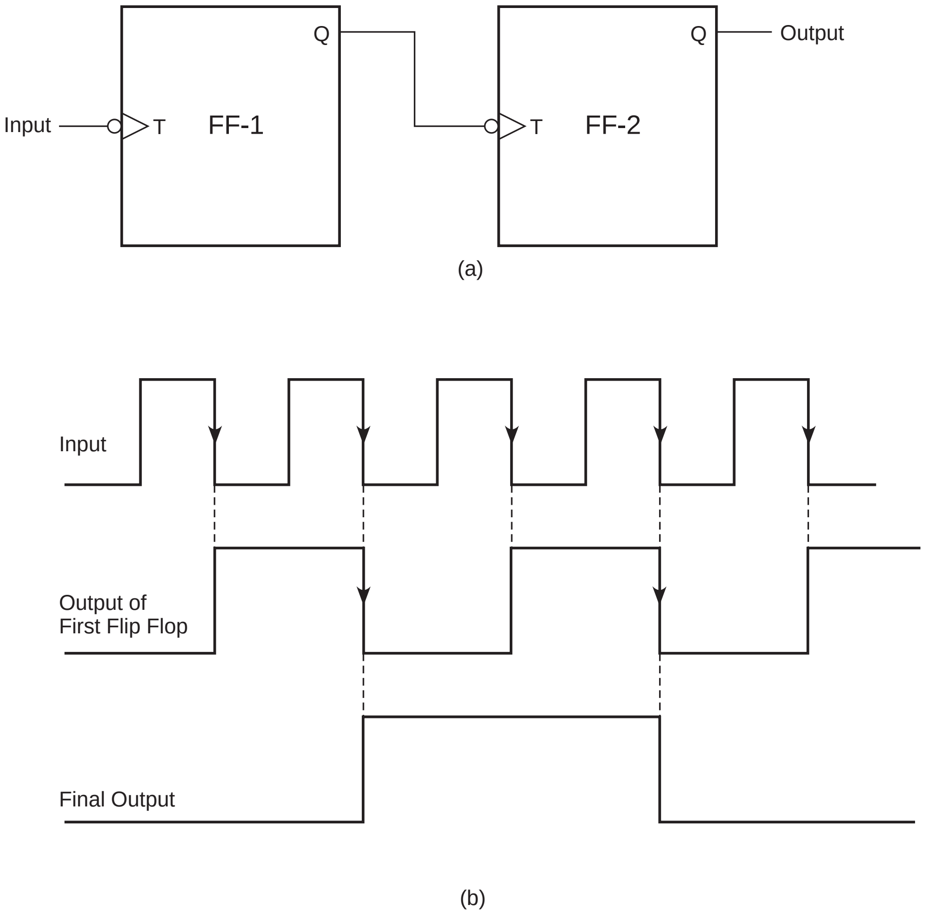


Figure 10.37 Example 10.5.

When the second flip-flop (FF-2) is a positive edge-triggered one, it will respond to the LOW-to-HIGH edges of the waveform appearing at its *T* input, which is the waveform appearing at the *Q* output of FF-1. The relevant waveforms in this case are shown in Fig. 10.38.

10.7 *D* Flip-Flop

A *D* flip-flop, also called a *delay flip-flop*, can be used to provide temporary storage of one bit of information. Figure 10.39(a) shows the circuit symbol and function table of a negative edge-triggered *D* flip-flop. When the clock is active, the data bit (0 or 1) present at the *D* input is transferred to the output. In the *D* flip-flop of Fig. 10.39, the data transfer from *D* input to *Q* output occurs on the negative-going (HIGH-to-LOW) transition of the clock input. The *D* input can acquire new status

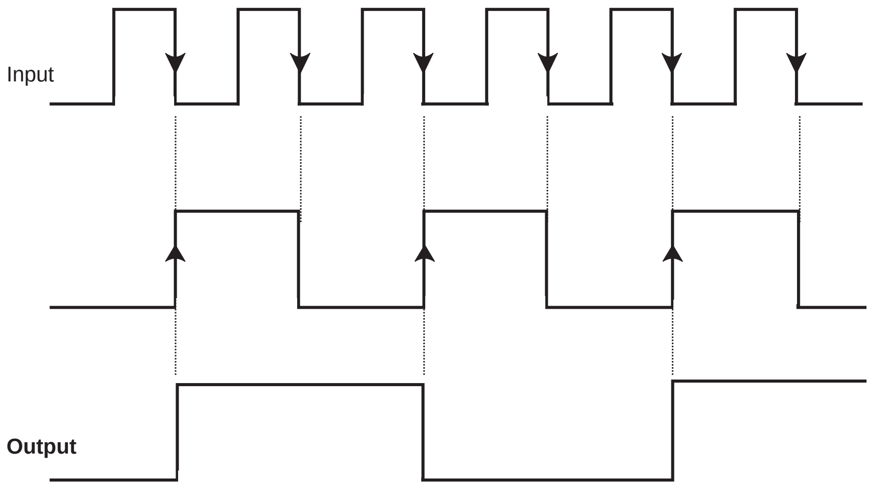


Figure 10.38 Example 10.5.

when the clock is inactive, which is the time period between successive HIGH-to-LOW transitions. The *D* flip-flop can provide a maximum delay of one clock period.

The characteristic table and the corresponding Karnaugh map for the *D* flip-flop of Fig. 10.39(a) are shown in Figs 10.39(c) and (d) respectively. The characteristic equation is as follows:

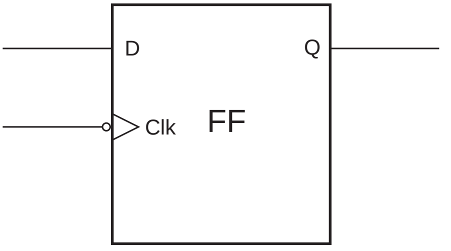
$$Q_{n+1} = D \tag{10.21}$$

10.7.1 J-K Flip-Flop as D Flip-Flop

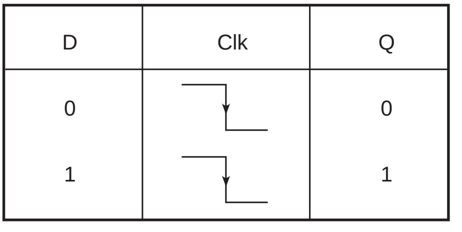
Figure 10.40 shows how a *J-K* flip-flop can be used as a *D* flip-flop. When the *D* input is a logic ‘1’, the *J* and *K* inputs are a logic ‘1’ and ‘0’ respectively. According to the function table of the *J-K* flip-flop, under these input conditions, the *Q* output will go to the logic ‘1’ state when clocked. Also, when the *D* input is a logic ‘0’, the *J* and *K* inputs are a logic ‘0’ and ‘1’ respectively. Again, according to the function table of the *J-K* flip-flop, under these input conditions, the *Q* output will go to the logic ‘0’ state when clocked. Thus, in both cases, the *D* input is passed on to the output when the flip-flop is clocked.

10.7.2 D Latch

In a *D* latch, the output *Q* follows the *D* input as long as the clock input (also called the ENABLE input) is HIGH or LOW, depending upon the clock level to which it responds. When the ENABLE input goes to the inactive level, the output holds on to the logic state it was in just prior to the ENABLE input becoming inactive during the entire time period the ENABLE input is inactive.



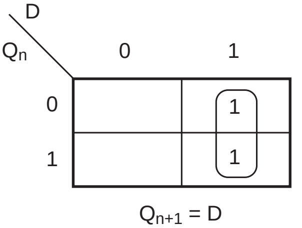
(a)



(b)

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

(c)



$Q_{n+1} = D$

(d)

Figure 10.39 *D* flip-flop.

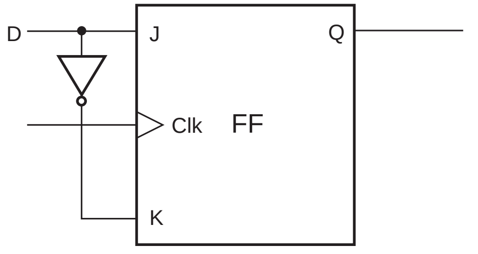


Figure 10.40 *J-K* flip-flop as a *D* flip-flop.

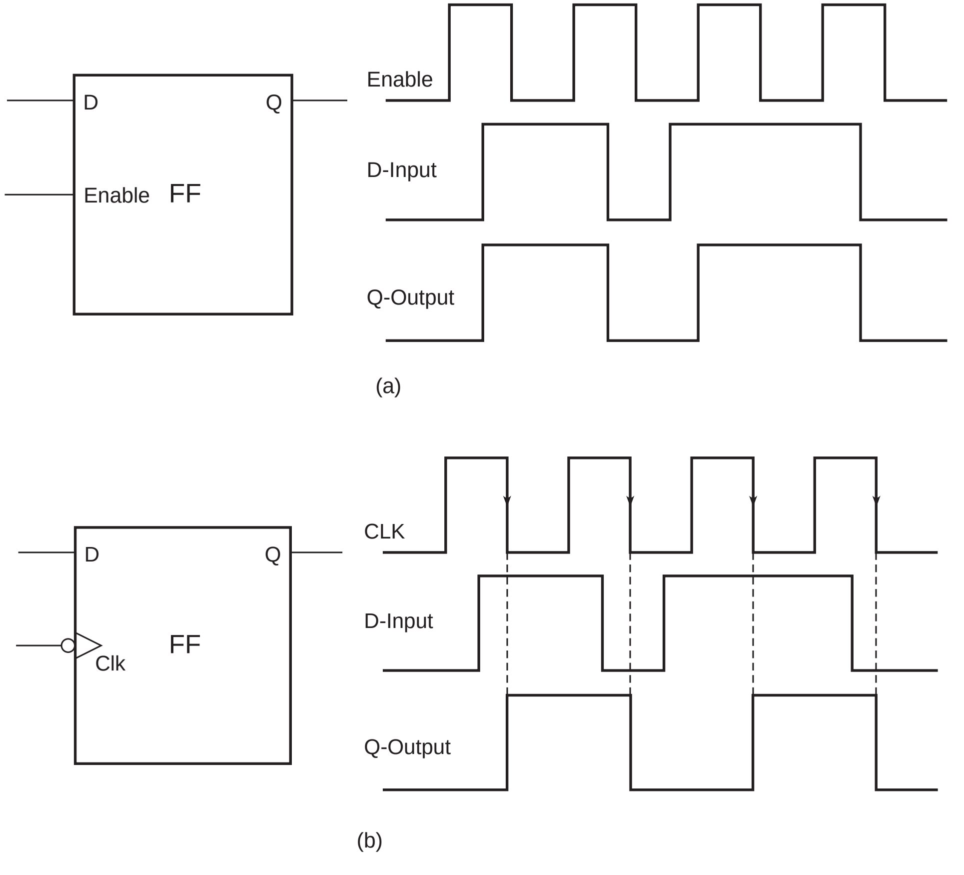


Figure 10.41 Comparison between a *D*-type latch and a *D* flip-flop.

A *D* flip-flop should not be confused with a *D* latch. In a *D* flip-flop, the data on the *D* input are transferred to the *Q* output on the positive- or negative-going transition of the clock signal, depending upon the flip-flop, and this logic state is held at the output until we get the next effective clock transition. The difference between the two is further illustrated in Figs 10.41(a) and (b) depicting the functioning of a *D* latch and a *D* flip-flop respectively.

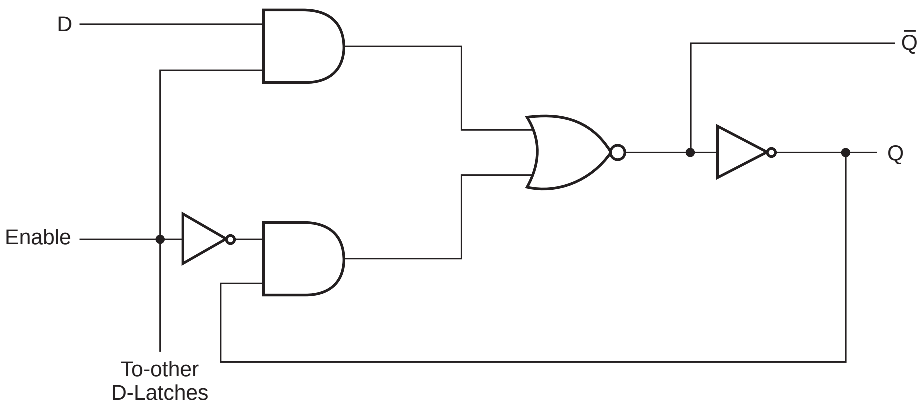


Figure 10.42 Example 10.6.

Example 10.6

Figure 10.42 shows the internal logic circuit diagram of one of the four D latches of a four-bit D latch in IC 7475. (a) Give an argument to prove that the Q output will track the D input only when the ENABLE input is HIGH. (b) Also, prove that the Q output holds the value it had just before the ENABLE input went LOW during the time the ENABLE input is LOW.

Solution

(a) When the ENABLE input is HIGH, the upper AND gate is enabled while the lower AND gate is disabled. The outputs of the upper and lower AND gates are D and logic '0' respectively. They constitute inputs of the NOR gate whose output is $\overline{D}$. The Q output is therefore D.

(b) When the ENABLE input goes LOW, the upper AND gate is disabled (with its output going to logic '0') and the lower AND gate is enabled (with its output becoming the same as the Q output owing to the feedback). The NOR gate output in this case is $\overline{Q}$, which means that the Q output holds its state as long as the ENABLE input is LOW.

10.8 Synchronous and Asynchronous Inputs

Most flip-flops have both synchronous and asynchronous inputs. Synchronous inputs are those whose effect on the flip-flop output is synchronized with the clock input. R, S, J, K and D inputs are all synchronous inputs. Asynchronous inputs are those that operate independently of the synchronous inputs and the input clock signal. These are in fact override inputs as their status overrides the status of all synchronous inputs and also the clock input. They force the flip-flop output to go to a predefined state irrespective of the logic status of the synchronous inputs. PRESET and CLEAR inputs are examples of asynchronous inputs. When active, the PRESET and CLEAR inputs place the flip-flop Q output in the '1' and '0' state respectively. Usually, these are active LOW inputs. When it is desired that the flip-flop functions as per the status of its synchronous inputs, the asynchronous inputs are kept in their inactive state. Also, both asynchronous inputs, if available on a given flip-flop, are not made active simultaneously.

10.9 Flip-Flop Timing Parameters

Certain timing parameters would be listed in the specification sheet of a flip-flop. Some of these parameters, as we will see in the paragraphs to follow, are specific to the logic family to which the flip-flop belongs. There are some parameters that have different values for different flip-flops belonging to the same broad logic family. It is therefore important that one considers these timing parameters before using a certain flip-flop in a given application. Some of the important ones are set-up and hold times, propagation delay, clock pulse HIGH and LOW times, asynchronous input active pulse width, clock transition time and maximum clock frequency.

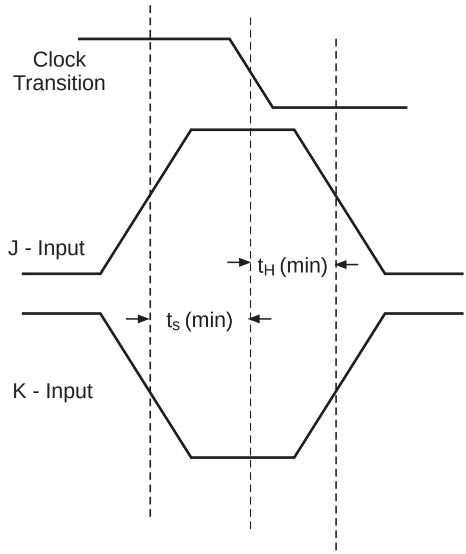
10.9.1 Set-Up and Hold Times

The *set-up time* is the minimum time period for which the synchronous inputs (for example, R, S, J, K and D) and asynchronous inputs (for example, PRESET and CLEAR) must be stable prior to the active clock transition for the flip-flop output to respond reliably at the clock transition. It is usually denoted by t_s (min) and is usually defined separately for synchronous and asynchronous inputs. As an example, if in a J-K flip-flop the J and K inputs were to go to '1' and '0' respectively, and if the flip-flop were negative edge triggered, the set-up time would be as shown in Fig. 10.43(a). The set-up time in the case of 74ALS109A, which is a dual J-K positive edge-triggered flip-flop belonging to the advanced low-power Schottky TTL logic family, is 15 ns. Also, the asynchronous inputs, such as PRESET and CLEAR, if there, should be inactive prior to the clock transition for a certain minimum time period if the outputs have to respond as per synchronous inputs. In the case of 74ALS109A, the asynchronous input set-up time is 10 ns. The asynchronous input set-up time for active low PRESET and CLEAR inputs is shown in Fig. 10.43(b), assuming a positive edge-triggered flip-flop.

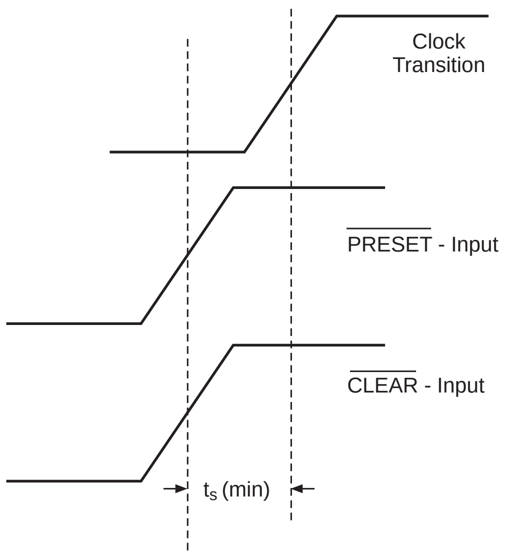
The *hold time* t_H (min) is the minimum time period for which the synchronous inputs (R, S, J, K, D) must remain stable in the desired logic state after the active clock transition for the flip-flop to respond reliably. The same is depicted in Fig. 10.43(a) if the desired logic status for J and K inputs is '1' and '0' respectively and the flip-flop is negative edge triggered. The hold time for flip-flop 74ALS109A is specified to be zero. To sum up, for a flip-flop to respond properly and reliably at the active clock transition, the synchronous inputs must be stable in their intended logic states and the asynchronous inputs must be stable in their inactive states for at least a time period equal to the specified minimum set-up times prior to the clock transition, and the synchronous inputs must be stable for a time period equal to at least the specified minimum hold time after the clock transition.

10.9.2 Propagation Delay

There is always a time delay, known as the *propagation delay*, from the time instant the signal is applied to the time the output makes the intended change. The flip-flop data sheet usually specifies propagation delays for both HIGH-to-LOW (t_{pHL}) and for LOW-to-HIGH (t_{pLH}) output transitions. The propagation delay is measured between 50 % points on input and output waveforms and is usually specified for all types of input including synchronous and asynchronous inputs. The propagation delays for LOW-to-HIGH and HIGH-to-LOW output transitions for a positive edge-triggered flip-flop are shown in Fig. 10.44. For flip-flop 74ALS109A, t_{pHL} and t_{pLH} for clock input to output are respectively 18 and 16 ns. The same for the asynchronous input to output for this flip-flop are 15 and 13 ns respectively.



(a)



(b)

Figure 10.43 Set-up and hold times of a flip-flop.

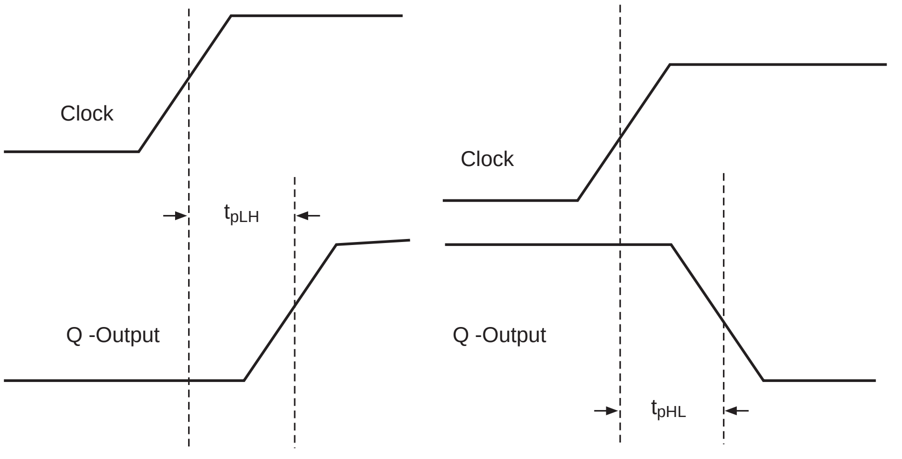


Figure 10.44 Propagation delay.

10.9.3 Clock Pulse HIGH and LOW Times

The clock pulse HIGH time t_W (H) and clock pulse LOW time, t_W (L) are respectively the minimum time durations for which the clock signal should remain HIGH and LOW. Failure to meet these requirements can lead to unreliable triggering. Figure 10.45 depicts these timing parameters. t_W (H) and t_W (L) for 74ALS109A are 4 and 5.5 ns respectively.

10.9.4 Asynchronous Input Active Pulse Width

This is the minimum time duration for which the asynchronous input (PRESET or CLEAR) must be kept in its active state, usually LOW, for the output to respond properly. It is 4 ns in the case of flip-flop 74ALS109A. Figure 10.46 shows this timing parameter.

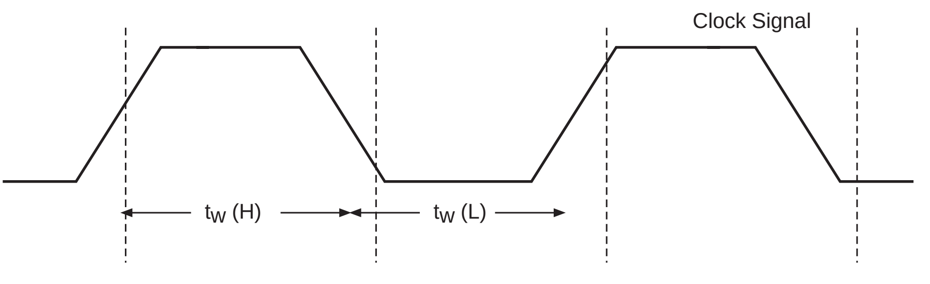


Figure 10.45 Clock pulse HIGH and LOW times.

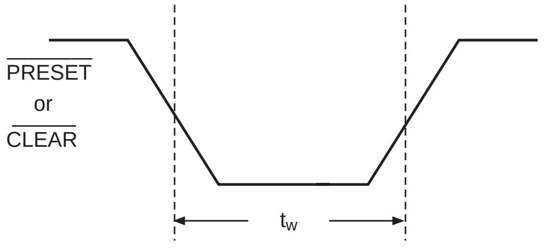


Figure 10.46 Asynchronous input active pulse width.

10.9.5 Clock Transition Times

The manufacturers specify the maximum transition times (rise time and fall time) for the output to respond properly. If these specified figures are exceeded, the flip-flop may respond erratically or even may not respond at all. This parameter is logic family specific and is not specified for individual devices. The allowed maximum transition time for TTL devices is much smaller than that for CMOS devices. Also, within the broad TTL family, it varies from one subfamily to another.

10.9.6 Maximum Clock Frequency

This is the highest frequency that can be applied to the clock input. If this figure is exceeded, there is no guarantee that the device will work reliably and properly. This figure may vary slightly from device to device of even the same type number. The manufacturer usually specifies a safe value. If this specified value is not exceeded, the manufacturer guarantees that the device will trigger reliably. It is 34 MHz for 74ALS109A.

10.10 Flip-Flop Applications

Flip-flops are used in a variety of application circuits, the most common among these being the *frequency division and counting* circuits and *data storage and transfer* circuits. These application areas are discussed at length in Chapter 11 on counters and registers. Both these applications use a cascaded arrangement of flip-flops with or without some additional combinational logic to perform the desired function. Counters and registers are available in IC form for a variety of digital circuit applications.

Other applications of flip-flops include their use for switch debouncing, where even an unclocked flip-flop (such as a NAND or a NOR latch) can be used, for synchronizing asynchronous inputs with the clock input and for identification of edges of synchronous inputs. These are briefly described in the following paragraphs.

10.10.1 Switch Debouncing

Owing to the switch bounce phenomenon, the mechanical switch cannot be used as such to produce a clean voltage transition. Refer to Fig. 10.47(a). When the switch is moved from position 1 to position 2, what is desired at the output is a clean voltage transition from 0 to $+V$ volts, as shown in Fig. 10.47(b). What actually happens is shown in Fig. 10.47(c). The output makes several transitions between 0 and

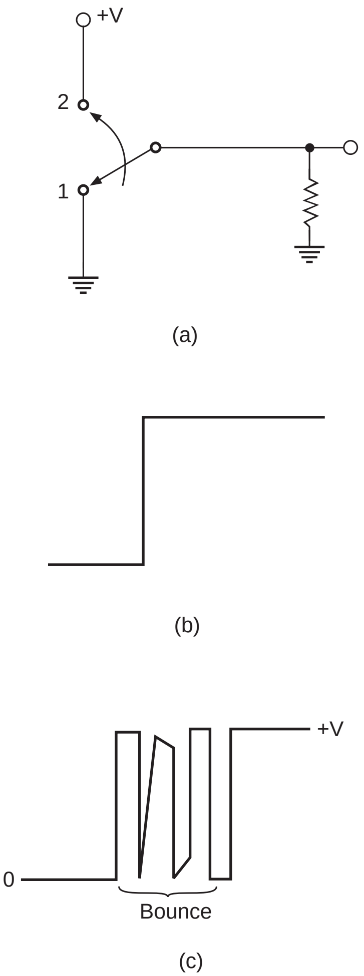


Figure 10.47 Switch bounce phenomenon.

$+V$ volts for a few milliseconds owing to contact bounce before it finally settles at $+V$ volts. Similarly, when it is moved from position 2 back to position 1, it makes several transitions before coming to rest at 0 V. Although this random behaviour lasts only for a few milliseconds, it is unacceptable for many digital circuit applications. A NAND or a NOR latch can solve this problem and provide a clean output transition. Figure 10.48 shows a typical switch debounce circuit built around a NAND latch. The circuit functions as follows.

When the switch is in position 1, the output is at a ‘0’ level. When it is moved to position 2, the output goes to a ‘1’ level within a few nanoseconds (depending upon the propagation delay of the NAND gate) after its first contact with position 2. When the switch contact bounces, it makes and breaks contact with position 2 before it finally settles at the intended position. Making of contact

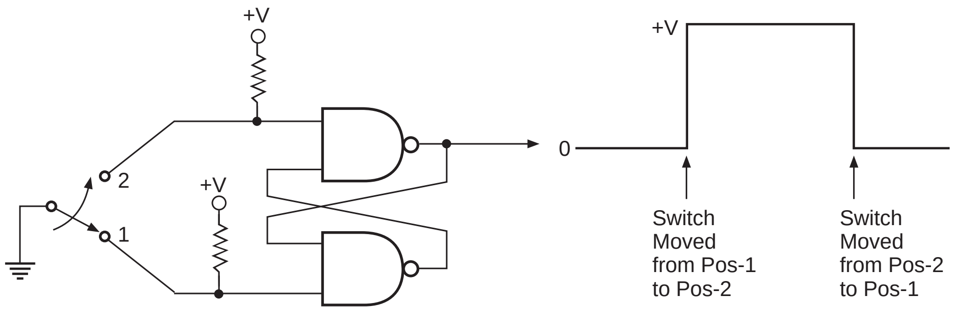


Figure 10.48 Switch debounce circuit.

always leads to a ‘1’ level at the output, and breaking of contact also leads to a ‘1’ level at the output owing to the fact that the contact break produces a ‘1’ level at both inputs of the latch which forces the output to hold its existing logic state. The fact that when the switch is brought back to position 1 the output makes a neat transition to a ‘0’ level can be explained on similar lines.

10.10.2 Flip-Flop Synchronization

Consider a situation where a certain clock input, which works in conjunction with various synchronous inputs, is to be gated with an asynchronously generated gating pulse, as shown in Fig. 10.49. The output in this case has the clock pulses at one or both ends shortened in width, as shown in Fig. 10.49. This problem can be overcome and the gating operation synchronized with the help of a flip-flop, as shown in Fig. 10.50.

10.10.3 Detecting the Sequence of Edges

Flip-flops can also be used to detect the sequence of occurrence of rising and falling edges. Figure 10.51 shows how a flip-flop can be used to detect whether a positive-going edge A follows or precedes another positive-going edge B. The two edges are respectively applied to *D* and clock inputs of a

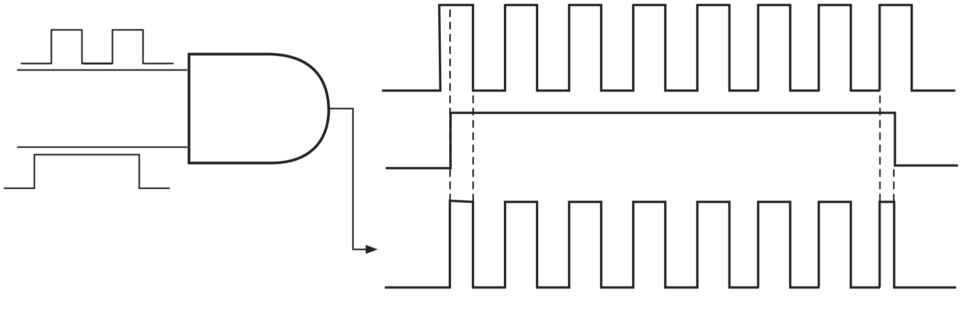


Figure 10.49 Gating of a clock signal.

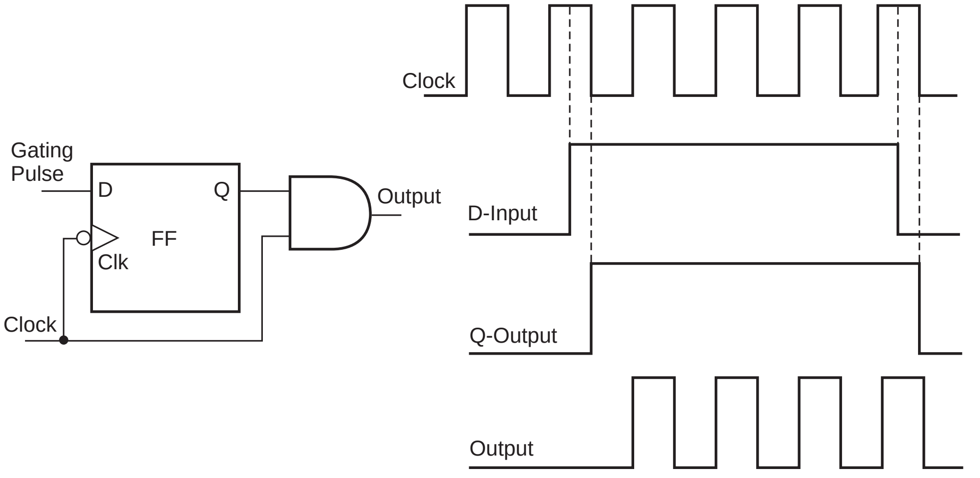


Figure 10.50 Flip-flop synchronization.

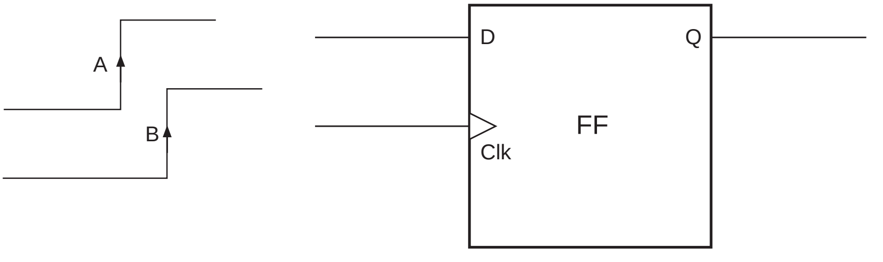


Figure 10.51 Detection of the sequence of edges.

positively edge-triggered *D* flip-flop. If edge A arrives first, then, on arrival of edge B, the output goes from 0 to 1. If it is otherwise, it stays at a '0' level.

Example 10.7

Figure 10.52 shows two pulsed waveforms A and B, with waveform A leading waveform B in phase, as shown in the figure. Suggest a flip-flop circuit to detect this condition by producing (a) a logic '1' Q output and (b) a logic '0' Q output.

Solution

(a) A positive edge-triggered *D* flip-flop, as shown in Fig. 10.53(a), can be used for the purpose. Waveform A is applied to the *D* input, and waveform B is applied to the clock input. If we examine the two waveforms, we will find that, on every occurrence of the leading edge of waveform B,

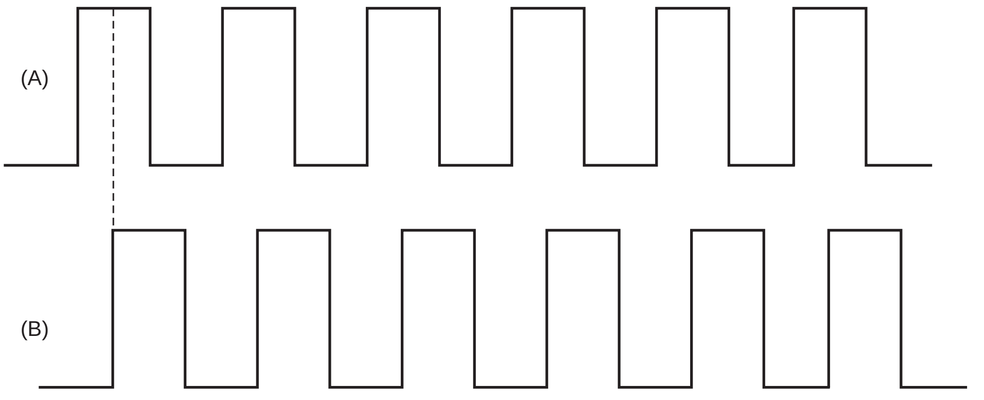


Figure 10.52 Example 10.7.

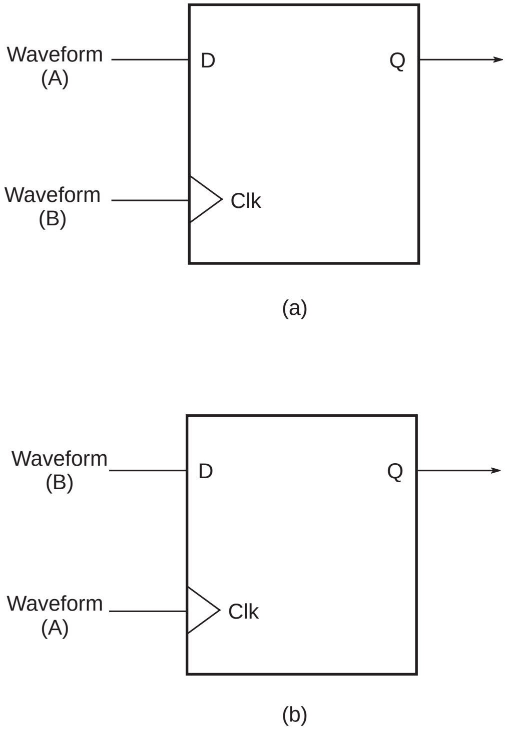


Figure 10.53 Solution to example 10.7.

waveform A is in a logic '1' state. Thus, the Q output in this case will always be in a logic '1' state.

(b) By interchanging the connections of waveforms A and B as shown in Fig. 10.53(b), the Q output will be in a logic '0' state as long as waveform A leads waveform B in phase. In this case, on every occurrence of the leading edge of waveform A (clock input), waveform B (D input) is in a logic '0' state.

10.11 Application-Relevant Data

Table 10.1 lists popular type numbers of flip-flops belonging to TTL, CMOS and ECL logic families. Application-relevant information of some of the popular type numbers is given in the companion website. The information given includes the pin connection diagram, package style and function table.

Table 10.1 Popular type numbers of flip-flops belonging to the TTL, CMOS and ECL logic families.

IC type number	Function	Logic family
54/7473	Dual J-K negative edge-triggered flip-flop with CLEAR	TTL
54/7474	Dual D-type positive edge-triggered flip-flop with PRESET and CLEAR	TTL
54/7475	Four-bit D-type latch	TTL
54/7476	Dual J-K flip-flop with PRESET and CLEAR	TTL
54/7478	Dual J-K flip-flop with PRESET and CLEAR	TTL
54/74107	Dual J-K flip-flop with CLEAR	TTL
54/74109	Dual J-K positive edge-triggered flip-flop with PRESET and CLEAR	TTL
54/74112	Dual J-K negative edge-triggered flip-flop with PRESET and CLEAR	TTL
54/74113	Dual J-K negative edge-triggered flip-flop with PRESET	TTL
54/74114	Dual J-K negative edge-triggered flip-flop with PRESET and CLEAR	TTL
54/74121	Monostable multivibrator	TTL
54/74122	Retriggerable monostable multivibrator	TTL
54/74123	Dual retriggerable monostable multivibrator	TTL
54/74174	Hex D-type flip-flop with CLEAR	TTL
54/74175	Quad edge triggered D-type flip-flop with CLEAR	TTL
54/74221	Dual monostable multivibrator	TTL
54/74256	Dual four-bit addressable latch	TTL
54/74259	Eight-bit addressable latch	TTL
54/74273	Octal D-type flip-flop with MASTER RESET	TTL
54/74279	Quad SET/RESET latch	TTL
54/74373	Octal transparent latch (three-state)	TTL
54/74374	Octal D-type flip-flop (three-state)	TTL
54/74377	Octal D-type flip-flop with common ENABLE	TTL
54/74378	Hex D-type flip-flop with ENABLE	TTL
54/74379	Four-bit D-type flip-flop with ENABLE	TTL
54/74533	Octal transparent latch (three-state)	TTL
54/74534	Octal D-type flip-flop (three-state)	TTL
54/74573	Octal D-type latch (three-state)	TTL
54/74574	Octal D-type flip-flop (three-state)	TTL

(continued overleaf)

Table 10.1 (*continued*).

IC type number	Function	Logic family
4013	Dual D-type flip-flop	CMOS
4027	Dual J-K flip-flop	CMOS
4042	Quad D-type latch	CMOS
4044	Quad R-S latch with three-state output	CMOS
4047	Low-power monostable/astable multivibrator	CMOS
4076	Quad D-type flip-flop with three-state output	CMOS
40174	Hex D-type flip-flop	CMOS
40175	Quad D-type flip-flop	CMOS
4511	BCD to seven-segment latch/decoder/driver	CMOS
4528	Dual retriggerable resettable monostable multivibrator	CMOS
4543	BCD to seven-segment latch/decoder/driver for LCD	CMOS
4723	Dual four-bit addressable latch	CMOS
4724	Eight-bit addressable latch	CMOS
MC10130	Quad D-type latch	ECL
MC10131	Dual D-type master/slave flip-flop	ECL
MC10133	Quad D-type latch (negative transition)	ECL
MC10135	Dual J-K master/slave flip-flop	ECL
MC10153	Quad latch (positive transition)	ECL
MC10168	Quad D-type latch	ECL
MC10175	Quint latch	ECL
MC10176	Hex D-type master/slave flip-flop	ECL
MC10198	Monostable multivibrator	ECL
MC10231	High-Speed dual D-type M/S flip-flop	ECL
MC1666	Dual clocked R-S flip-flop	ECL
MC1668	Dual clocked latch	ECL
MC1670	D-type master/slave flip-flop	ECL
MC1658	Voltage-controlled multivibrator	ECL

Review Questions

1. Briefly describe the operational aspects of bistable, monostable and astable multivibrators. Which multivibrator closely resembles a flip-flop?
2. What is a flip-flop? Show the logic implementation of an R-S flip-flop having active HIGH R and S inputs. Draw its truth table and mark the invalid entry.
3. With the help of the logic diagram, describe the operation of a clocked R-S flip-flop with active LOW R and S inputs. Draw the truth table of this flip-flop if it were negatively edge triggered.
4. What is a clocked J-K flip-flop? What improvement does it have over a clocked R-S flip-flop?
5. Differentiate between:

   (a) synchronous and asynchronous inputs;
   (b) level-triggered and edge-triggered flip-flops;
   (c) active LOW and active HIGH inputs.

6. Briefly describe the following flip-flop timing parameters:

(a) set-up time and hold time;
(b) propagation delay;
(c) maximum clock frequency.

7. Draw the truth table for the following types of flip-flop:

   (a) a positive edge-triggered J-K flip-flop with active HIGH J and K inputs and active LOW PRESET and CLEAR inputs;
   (b) a negative edge-triggered J-K flip-flop with active LOW J and K inputs and active LOW PRESET and CLEAR inputs.

8. What is meant by the race problem in flip-flops? How does a master–slave configuration help in solving this problem?
9. Differentiate between a D flip-flop and a D latch.
10. Draw the function table for (a) a negative edge-triggered D flip-flop and (b) a D latch with an active LOW ENABLE input.
11. With the help of a schematic arrangement, explain how a J-K flip-flop can be used as a (a) a D flip-flop and (b) a T flip-flop.
12. With the help of a suitable circuit, briefly explain how a D flip-flop can be used to detect the sequence of occurrence of edges of synchronous inputs.

Problems

1. A 100 kHz clock signal is applied to a J-K flip-flop with $J = K = 1$.

   (a) If the flip-flop has active HIGH J and K inputs and is negative edge triggered, determine the frequency of the Q and $\overline{Q}$ outputs.
   (b) If the flip-flop has active LOW J and K inputs and is positive edge triggered, what should the frequency of the Q and $\overline{Q}$ outputs be? Assume that Q is initially '0'.

   (a) Q output = 50 kHz, $\overline{Q}$ output = 50 kHz;
   (b) both outputs remain in a logic '0' state

2. In a Schmitt trigger inverter circuit, the two trip points are observed to occur at 1.8 and 2.8 V. At what input voltage levels will this device make (a) HIGH-to-LOW transition and (b) LOW-to-HIGH transition?

   (a) 2.8 V; (b) 1.8 V

3. In the case of a presettable, clearable J-K flip-flop with active HIGH J and K inputs and active LOW PRESET and CLEAR inputs, what would the Q output logic status be for the following input conditions, assuming that Q is initially '0', immediately after it is clocked?

   (a) $J = 1$, $K = 0$, PRESET = 1, CLEAR = 1;
   (b) $J = 1$, $K = 1$, PRESET = 0, CLEAR = 1;
   (c) $J = 0$, $K = 1$, PRESET = 1, CLEAR = 0;
   (d) $J = K = 0$, PRESET = 0, CLEAR = 1.

   (a) 1; (b) 1; (c) 0; (d) 1

4. Figure 10.54 shows the function table of a certain flip-flop. Identify the flip-flop.

Negative edge-triggered J-K flip-flop with active HIGH J and K inputs and active LOW PRESET and CLEAR inputs

Pr	Cl	Clk	J	K	Qn+1	$\overline{\text{Qn+1}}$
1	0	X	X	X	0	1
0	1	X	X	X	1	0
0	0	X	X	X	Unstable	
0	0	↓	1	1	1	0
0	0	↓	0	1	0	1
0	0	↓	1	1	Toggle	
0	0	↓	0	0	Qn	Qn

Figure 10.54 Problem 4.

5. Derive the expression for Q_{n+1} in terms of Q_n and J and K inputs for a clocked J-K flip-flop with active LOW J and K inputs. Q_n and Q_{n+1} have the usual meaning.

$$Q_{n+1} = \overline{J}.\overline{Q_n} + K.Q_n$$

6. Consider a J-K flip-flop (J-$\overline{K}$ flip-flop to be more precise) where an inverter has been wired between the external $\overline{K}$ input and the internal K input as shown in Fig. 10.55. With the help of a characteristic table, write the characteristic equation for this flip-flop.

$$Q_{n+1} = J.\overline{Q_n} + K.Q_n$$

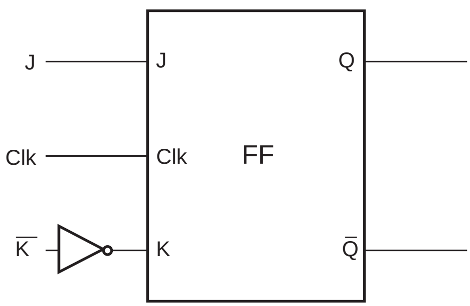


Figure 10.55 Problem 6.

Further Reading

1. Cook, N. P. (2003), *Practical Digital Electronics*, Prentice-Hall, NJ, USA.
2. Whitaker, C. (1996) *The Electronics Handbook,* CRC Press (in cooperation with IEEE Press), FL, USA.
3. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
4. Tocci, R. J. (2006), *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
5. Malvino, A. P. and Leach, D. P. (1994) *Digital Principles and Applications*, McGraw-Hill, USA.

11

Counters and Registers

Counters and *registers* belong to the category of MSI sequential logic circuits. They have similar architecture, as both counters and registers comprise a cascaded arrangement of more than one flip-flop with or without combinational logic devices. Both constitute very important building blocks of sequential logic, and different types of counter and register available in integrated circuit (IC) form are used in a wide range of digital systems. While counters are mainly used in counting applications, where they either measure the time interval between two unknown time instants or measure the frequency of a given signal, registers are primarily used for the temporary storage of data present at the output of a digital circuit before they are fed to another digital circuit. We are all familiar with the role of different types of register used inside a microprocessor, and also their use in microprocessor-based applications. Because of the very nature of operation of registers, they form the basis of a very important class of counters called *shift counters*. In this chapter, we will discuss different types of counter and register as regards their operational basics, design methodology and application-relevant aspects. Design aspects have been adequately illustrated with the help of a large number of solved examples. A comprehensive functional index of a large number of integrated circuit counters and registers is given towards the end of the chapter.

11.1 Ripple (Asynchronous) Counter

A *ripple counter* is a cascaded arrangement of flip-flops where the output of one flip-flop drives the clock input of the following flip-flop. The number of flip-flops in the cascaded arrangement depends upon the number of different logic states that it goes through before it repeats the sequence, a parameter known as the modulus of the counter.

In a ripple counter, also called an *asynchronous counter* or a *serial counter*, the clock input is applied only to the first flip-flop, also called the input flip-flop, in the cascaded arrangement. The clock input to any subsequent flip-flop comes from the output of its immediately preceding flip-flop. For instance, the output of the first flip-flop acts as the clock input to the second flip-flop, the output of the second flip-flop feeds the clock input of the third flip-flop and so on. In general, in an arrangement of n

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

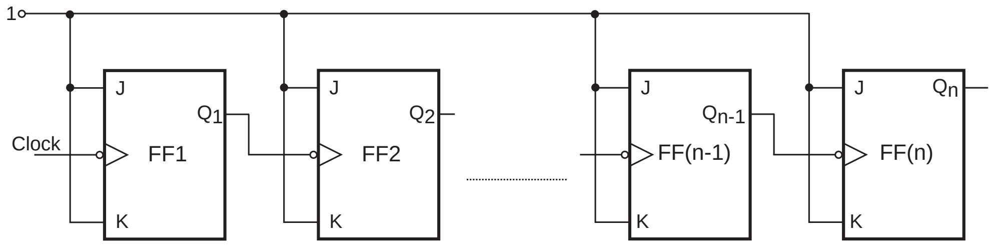


Figure 11.1 Generalized block schematic of n-bit binary ripple counter.

flip-flops, the clock input to the nth flip-flop comes from the output of the $(n-1)$th flip-flop for $n > 1$. Figure 11.1 shows the generalized block schematic arrangement of an n-bit binary ripple counter.

As a natural consequence of this, not all flip-flops change state at the same time. The second flip-flop can change state only after the output of the first flip-flop has changed its state. That is, the second flip-flop would change state a certain time delay after the occurrence of the input clock pulse owing to the fact that it gets its own clock input from the output of the first flip-flop and not from the input clock. This time delay here equals the sum of propagation delays of two flip-flops, the first and the second flip-flops. In general, the nth flip-flop will change state only after a delay equal to n times the propagation delay of one flip-flop. The term 'ripple counter' comes from the mode in which the clock information ripples through the counter. It is also called an 'asynchronous counter' as different flip-flops comprising the counter do not change state in synchronization with the input clock.

In a counter like this, after the occurrence of each clock input pulse, the counter has to wait for a time period equal to the sum of propagation delays of all flip-flops before the next clock pulse can be applied. The propagation delay of each flip-flop, of course, will depend upon the logic family to which it belongs.

11.1.1 Propagation Delay in Ripple Counters

A major problem with ripple counters arises from the propagation delay of the flip-flops constituting the counter. As mentioned in the preceding paragraphs, the effective propagation delay in a ripple counter is equal to the sum of propagation delays due to different flip-flops. The situation becomes worse with increase in the number of flip-flops used to construct the counter, which is the case in larger bit counters. Coming back to the ripple counter, an increased propagation delay puts a limit on the maximum frequency used as clock input to the counter. We can appreciate that the clock signal time period must be equal to or greater than the total propagation delay. The maximum clock frequency therefore corresponds to a time period that equals the total propagation delay. If t_{pd} is the propagation delay in each flip-flop, then, in a counter with N flip-flops having a modulus of less than or equal to 2^N, the maximum usable clock frequency is given by $f_{max} = 1/(N \times t_{pd})$. Often, two propagation delay times are specified in the case of flip-flops, one for LOW-to-HIGH transition (t_{pLH}) and the other for HIGH-to-LOW transition (t_{pHL}) at the output. In such a case, the larger of the two should be considered for computing the maximum clock frequency.

As an example, in the case of a ripple counter IC belonging to the low-power Schottky TTL (LSTTL) family, the propagation delay per flip-flop typically is of the order of 25 ns. This implies that a four-bit ripple counter from this logic family can not be clocked faster than 10 MHz. The upper limit on the clock frequency further decreases with increase in the number of bits to be handled by the counter.

11.2 Synchronous Counter

In a *synchronous counter*, also known as a *parallel counter*, all the flip-flops in the counter change state at the same time in synchronism with the input clock signal. The clock signal in this case is simultaneously applied to the clock inputs of all the flip-flops. The delay involved in this case is equal to the propagation delay of one flip-flop only, irrespective of the number of flip-flops used to construct the counter. In other words, the delay is independent of the size of the counter.

11.3 Modulus of a Counter

The *modulus* (MOD number) of a counter is the number of different logic states it goes through before it comes back to the initial state to repeat the count sequence. An n-bit counter that counts through all its natural states and does not skip any of the states has a modulus of 2^n. We can see that such counters have a modulus that is an integral power of 2, that is, 2, 4, 8, 16 and so on. These can be modified with the help of additional combinational logic to get a modulus of less than 2^n.

To determine the number of flip-flops required to build a counter having a given modulus, identify the smallest integer m that is either equal to or greater than the desired modulus and is also equal to an integral power of 2. For instance, if the desired modulus is 10, which is the case in a decade counter, the smallest integer greater than or equal to 10 and which is also an integral power of 2 is 16. The number of flip-flops in this case would be 4, as $16 = 2^4$. On the same lines, the number of flip-flops required to construct counters with MOD numbers of 3, 6, 14, 28 and 63 would be 2, 3, 4, 5 and 6 respectively. In general, the arrangement of a minimum number of N flip-flops can be used to construct any counter with a modulus given by the equation

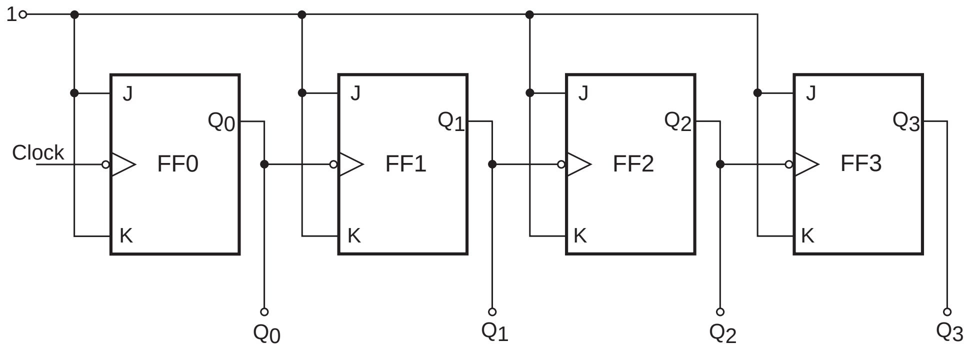
$$(2^{N-1} + 1) \leq modulus \leq 2^N \tag{11.1}$$

11.4 Binary Ripple Counter – Operational Basics

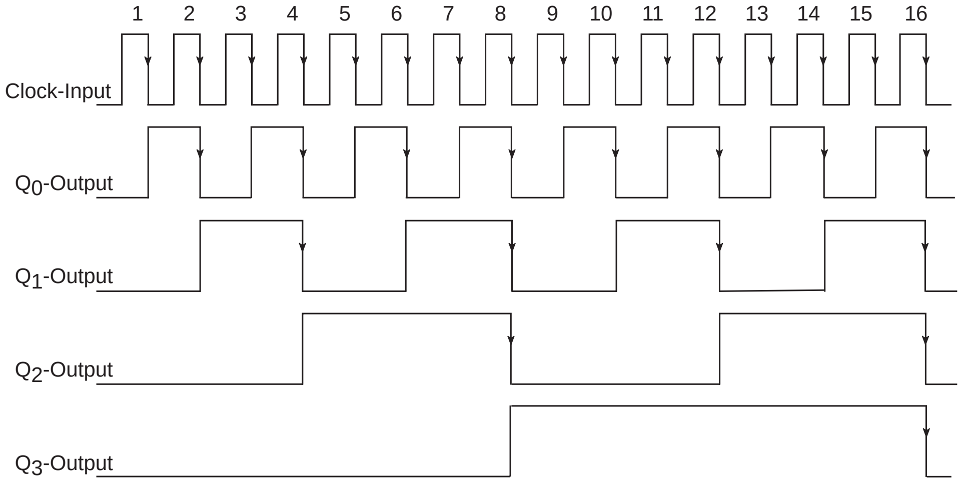
The operation of a binary ripple counter can be best explained with the help of a typical counter of this type. Figure 11.2(a) shows a four-bit ripple counter implemented with negative edge-triggered *J-K* flip-flops wired as toggle flip-flops. The output of the first flip-flop feeds the clock input of the second, and the output of the second flip-flop feeds the clock input of the third, the output of which in turn feeds the clock input of the fourth flip-flop. The outputs of the four flip-flops are designated as Q_0 (LSB flip-flop), Q_1, Q_2 and Q_3 (MSB flip-flop). Figure 11.2(b) shows the waveforms appearing at Q_0, Q_1, Q_2 and Q_3 outputs as the clock signal goes through successive cycles of trigger pulses. The counter functions as follows.

Let us assume that all the flip-flops are initially cleared to the '0' state. On HIGH-to-LOW transition of the first clock pulse, Q_0 goes from '0' to '1' owing to the toggling action. As the flip-flops used are negative edge-triggered ones, the '0' to '1' transition of Q_0 does not trigger flip-flop FF1. FF1, along with FF2 and FF3, remains in its '0' state. So, on the occurrence of the first negative-going clock transition, $Q_0 = 1$, $Q_1 = 0$, $Q_2 = 0$ and $Q_3 = 0$.

On the HIGH-to-LOW transition of the second clock pulse, Q_0 toggles again. That is, it goes from '1' to '0'. This '1' to '0' transition at the Q_0 output triggers FF1, the output Q_1 of which goes from '0'



(a)



(b)

Figure 11.2 Four-bit binary ripple counter.

to '1'. The Q_2 and Q_3 outputs remain unaffected. Therefore, immediately after the occurrence of the second HIGH-to-LOW transition of the clock signal, $Q_0 = 0$, $Q_1 = 1$, $Q_2 = 0$ and $Q_3 = 0$. On similar lines, we can explain the logic status of Q_0, Q_1, Q_2 and Q_3 outputs immediately after subsequent clock transitions. The logic status of outputs for the first 16 relevant (HIGH-to-LOW in the present case) clock signal transitions is summarized in Table 11.1.

Thus, we see that the counter goes through 16 distinct states from 0000 to 1111 and then, on the occurrence of the desired transition of the sixteenth clock pulse, it resets to the original state of 0000 from where it had started. In general, if we had N flip-flops, we could count up to 2^N pulses before the counter resets to the initial state. We can also see from the Q_0, Q_1, Q_2 and Q_3 waveforms, as shown

Table 11.1 Output logic states for different clock signal transitions for a four-bit binary ripple counter.

Clock signal transition number	Q_0	Q_1	Q_2	Q_3
After first clock transition	1	0	0	0
After second clock transition	0	1	0	0
After third clock transition	1	1	0	0
After fourth clock transition	0	0	1	0
After fifth clock transition	1	0	1	0
After sixth clock transition	0	1	1	0
After seventh clock transition	1	1	1	0
After eighth clock transition	0	0	0	1
After ninth clock transition	1	0	0	1
After tenth clock transition	0	1	0	1
After eleventh clock transition	1	1	0	1
After twelfth clock transition	0	0	1	1
After thirteenth clock transition	1	0	1	1
After fourteenth clock transition	0	1	1	1
After fifteenth clock transition	1	1	1	1
After sixteenth clock transition	0	0	0	0

in Fig. 11.2(b), that the frequencies of the Q_0, Q_1, Q_2 and Q_3 waveforms are $f/2$,$f/4$, $f/8$ and $f/16$ respectively. Here, f is the frequency of the clock input. This implies that a counter of this type can be used as a divide-by-2^N circuit, where N is the number of flip-flops in the counter chain. In fact, such a counter provides frequency-divided outputs of $f/2^N$, $f/2^{N-1}$, $f/2^{N-2}$, $f/2^{N-3}$, . . . , $f/2$ at the outputs of the Nth, ($N-$ 1)th, ($N-$ 2)th, ($N-$ 3)th, . . . , first flip-flops. In the case of a four-bit counter of the type shown in Fig. 11.2(a), outputs are available at $f/2$ from the Q_0 output, at $f/4$ from the Q_1 output, at $f/8$ from the Q_2 output and at $f/16$ from the Q_3 output. It may be noted that frequency division is one of the major applications of counters.

Example 11.1

A four-bit binary ripple counter of the type shown in Fig. 11.2(a) is initially in the 0000 state before the clock input is applied to the counter. The clock pulses are applied to the counter at some time instant t_1 and then again removed some time later at another time instant t_2. The counter is observed to read 0011. How many negative-going clock transitions have occurred during the time the clock was active at the counter input?

Solution

It is not possible to determine the number of clock edges – it could have been 3, 19, 35, 51, 67, 83 . . . – as there is no means of finding out whether the counter has recycled or not from the given data. Remember that this counter would come back to the 0000 state after every 16 clock pulses.

Example 11.2

It is desired to design a binary ripple counter of the type shown in Fig. 11.1 that is capable of counting the number of items passing on a conveyor belt. Each time an item passes a given point, a pulse is generated that can be used as a clock input. If the maximum number of items to be counted is 6000, determine the number of flip-flops required.

Solution

- The counter should be able to count a maximum of 6000 items.
- An N-flip-flop would be able to count up to a maximum of $2^N - 1$ counts.
- On the 2^Nth clock pulse, it will get reset to all 0s.
- Now, $2^N - 1$ should be greater than or equal to 6000.
- That is, $2^N - 1 \geq 6000$, which gives $N \geq \log 6001/\log 2 \geq 3.778/0.3010 \geq 12.55$.
- The smallest integer that satisfies this condition is 13.
- Therefore, the minimum number of flip-flops required = 13

11.4.1 Binary Ripple Counters with a Modulus of Less than 2^N

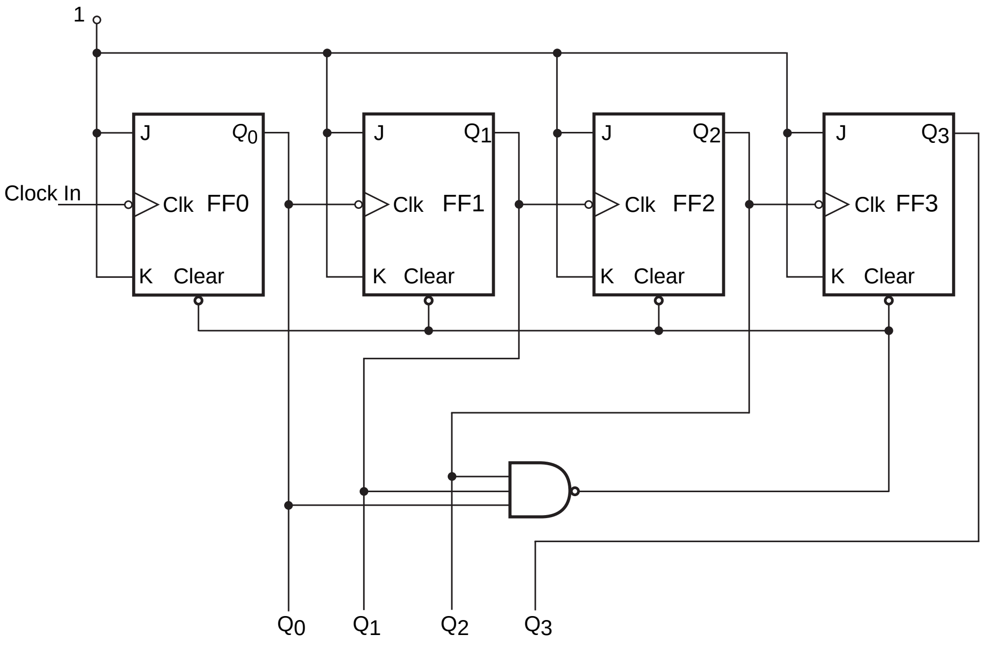
An N-flip-flop binary ripple counter can be modified, as we will see in the following paragraphs, to have any other modulus less than 2^N with the help of simple externally connected combinational logic. We will illustrate this simple concept with the help of an example.

Consider the four-flip-flop binary ripple counter arrangement of Fig. 11.3(a). It uses *J-K* flip-flops with an active LOW asynchronous CLEAR input. The NAND gate in the figure has its output connected to the CLEAR inputs of all four flip-flops. The inputs to this three-input NAND gate are from the Q outputs of flip-flops FF0, FF1 and FF2. If we disregard the NAND gate for some time, this counter will go through its natural binary sequence from 0000 to 1111. But that is not to happen in the present arrangement. The counter does start counting from 0000 towards its final count of 1111. The counter keeps counting as long as the asynchronous CLEAR inputs of the different flip-flops are inactive. That is, the NAND gate output is HIGH. This is the case until the counter reaches 0110. With the seventh clock pulse it tends to go to 0111, which makes all NAND gate inputs HIGH, forcing its output to LOW. This HIGH-to-LOW transition at the NAND gate output clears all flip-flop outputs to the logic '0' state, thus disallowing the counter to settle at 0111. From the eighth clock pulse onwards, the counter repeats the sequence. The counter thus always counts from 0000 to 0110 and resets back to 0000. The remaining nine states, which include 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111, are skipped, with the result that we get an MOD-7 counter.

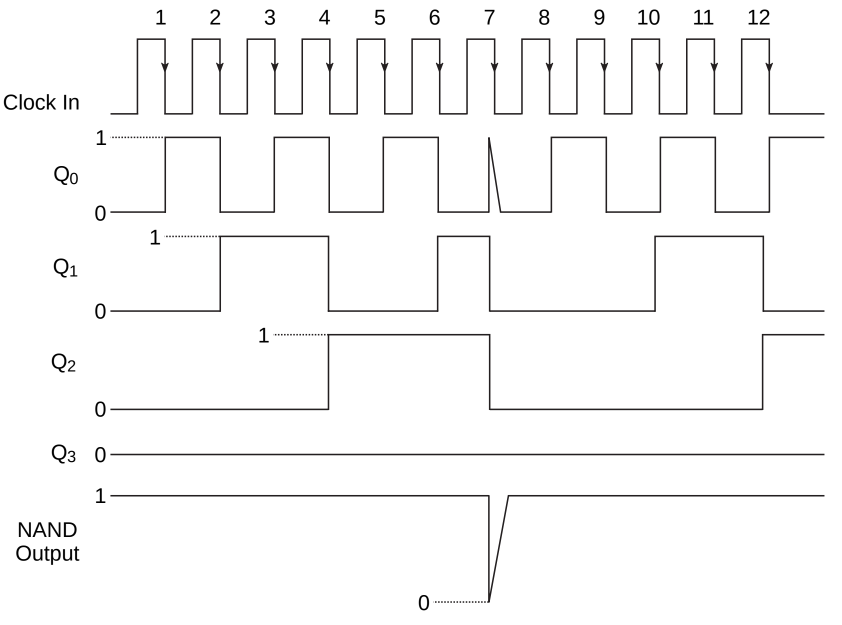
Figure 11.3(b) shows the timing waveforms for this counter. By suitably choosing NAND inputs, one can get a counter with any MOD number less than 16. Examination of timing waveforms also reveals that the frequency of the Q_2 output is one-seventh of the input clock frequency.

The waveform at the Q_2 output is, however, not symmetrical as it would be if the counter were to go through its full binary sequence. The Q_3 output stays in the logic LOW state. It is expected to be so because an MOD-7 counter needs a minimum of three flip-flops. That is why the fourth flip-flop, which was supposed to toggle on the HIGH-to-LOW transition of the eighth clock pulse, and on every successive eighth pulse thereafter, never gets to that stage. The counter is cleared on the seventh clock pulse and every successive seventh clock pulse thereafter.

As another illustration, if the NAND gate used in the counter arrangement of Fig. 11.3(a) is a two-input NAND and its inputs are from the Q_1 and Q_3 outputs, the counter will go through 0000 to 1001 and then reset to 0000 again, as, the moment the counter tends to switch from the 1001 to the 1010 state, the NAND gate goes from the '1' to the '0' state, clearing all flip-flops to the '0' state.



(a)



(b)

Figure 11.3 Binary ripple counter with a modulus of less than 2^N.

Steps to be followed to design any binary ripple counter that starts from 0000 and has a modulus of X are summarized as follows:

1. Determine the minimum number of flip-flops N so that $2^N \geq X$. Connect these flip-flops as a binary ripple counter. If $2^N = X$, do not go to steps 2 and 3.
2. Identify the flip-flops that will be in the logic HIGH state at the count whose decimal equivalent is X. Choose a NAND gate with the number of inputs equal to the number of flip-flops that would be in the logic HIGH state. As an example, if the objective were to design an MOD-12 counter, then, in the corresponding count, that is, 1100, two flip-flops would be in the logic HIGH state. The desired NAND gate would therefore be a two-input gate.
3. Connect the Q outputs of the identified flip-flops to the inputs of the NAND gate and the NAND gate output to asynchronous clear inputs of all flip-flops.

11.4.2 Ripple Counters in IC Form

In this section, we will look at the internal logic diagram of a typical binary ripple counter and see how close its architecture is to the ripple counter described in the previous section. Let us consider binary ripple counter type number 74293. It is a four-bit binary ripple counter containing four master–slave-type *J-K* flip-flops with additional gating to provide a divide-by-2 counter and a three-stage MOD-8 counter. Figure 11.4 shows the internal logic diagram of this counter. To get the full binary sequence of 16 states, the Q output of the LSB flip-flop is connected to the B input, which is the clock input of the next higher flip-flop. The arrangement then becomes the same as that shown in Fig. 11.2(a), with the exception of the two-input NAND gate of Fig. 11.4, which has been included here for providing the clearing features. The counter can be cleared to the 0000 logic state by driving both RESET inputs to the logic HIGH state. Tables 11.2 and 11.3 respectively give the functional table and the count sequence.

Example 11.3

Refer to the binary ripple counter of Fig. 11.5. Determine the modulus of the counter and also the frequency of the flip-flop Q_3 output.

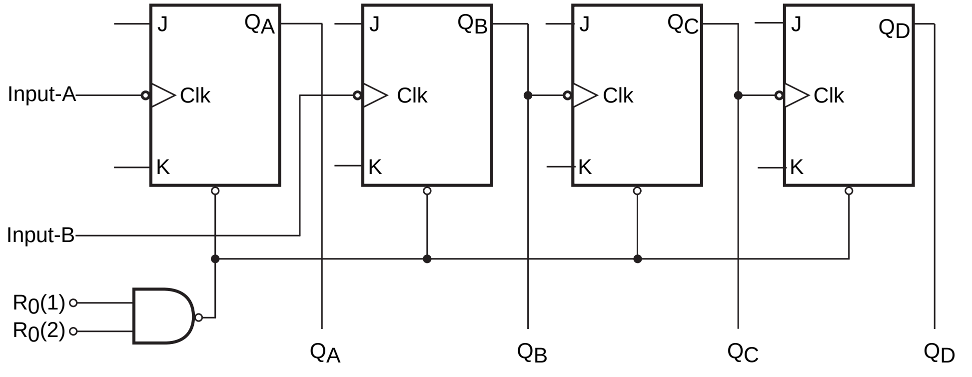


Figure 11.4 Logic diagram of IC 74293.

Table 11.2 Functional table for binary ripple counter, type number 74293.

RESET inputs		Outputs			
$R_0(1)$	$R_0(2)$	Q_D	Q_C	Q_B	Q_A
H	H	L	L	L	L
L	X		Count		
X	L		Count		

Table 11.3 Count sequence for binary ripple counter, type number 74293.

Count	Outputs			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H
10	H	L	H	L
11	H	L	H	H
12	H	H	L	L
13	H	H	L	H
14	H	H	H	L
15	H	H	H	H

Solution

- The counter counts in the natural sequence from 0000 to 1011.
- The moment the counter goes to 1100, the NAND output goes to the logic '0' state and immediately clears the counter to the 0000 state.
- Thus, the counter is not able to stay in the 1100 state. It has only 12 stable states from 0000 to 1011.
- Therefore, the modulus of the counter = 12.
- The Q_3 output is the input clock frequency divided by 12.
- Therefore, the frequency of the Q_3 output waveform $= 1.2 \times 10^3/12 = 100$ kHz.

Example 11.4

Design a binary ripple counter that counts 000 and 111 and skips the remaining six states, that is, 001, 010, 011, 100, 101 and 110. Use presentable, clearable negative edge-triggered J-K flip-flops with active LOW PRESET and CLEAR inputs. Also, draw the timing waveforms and determine the frequency of different flip-flop outputs for a given clock frequency, f_c.

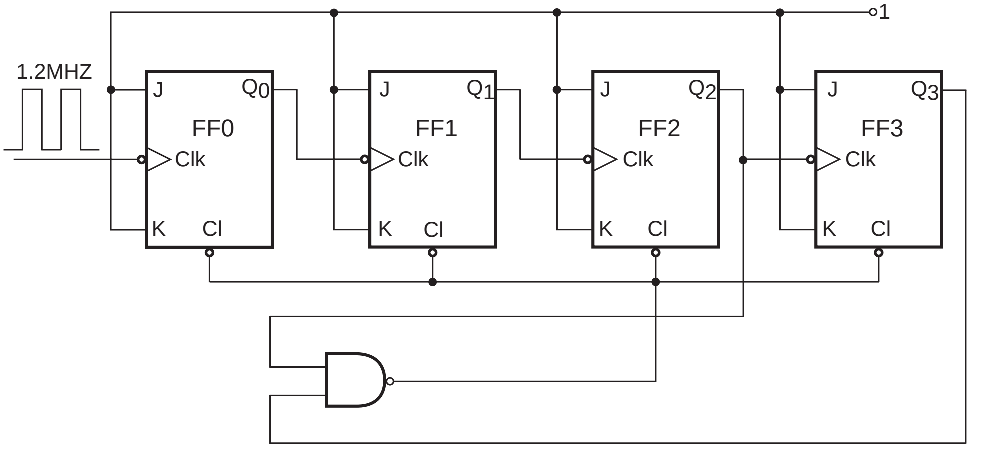


Figure 11.5 Example 11.3.

Solution

The counter is required to go to the 111 state from the 000 state with the first relevant clock transition. The second transition brings it back to the 000 state. That is, the three flip-flops toggle from logic '0' state to logic '1' state with every odd-numbered clock transition, and also the three flip-flops toggle from logic '1' state to logic '0' state with every even-numbered clock transition. Figure 11.6(a) shows the arrangement. The $\overline{\text{PRESET}}$ inputs of the three flip-flops have been tied to the NAND output whose inputs are Q_A, $\overline{Q_B}$ and $\overline{Q_C}$. Every time the counter is in the 000 state and is clocked, the NAND output momentarily goes from logic '1' state to logic '0' state, thus presetting the Q_A, Q_B and Q_C outputs to the logic '1' state. The timing waveforms as shown in Fig. 11.6(b) are self-explanatory.

The Q_A, $Q_B$and $Q_C$waveforms are identical, and each of them has a frequency of $f_c/2$, where f_c is the clock frequency.

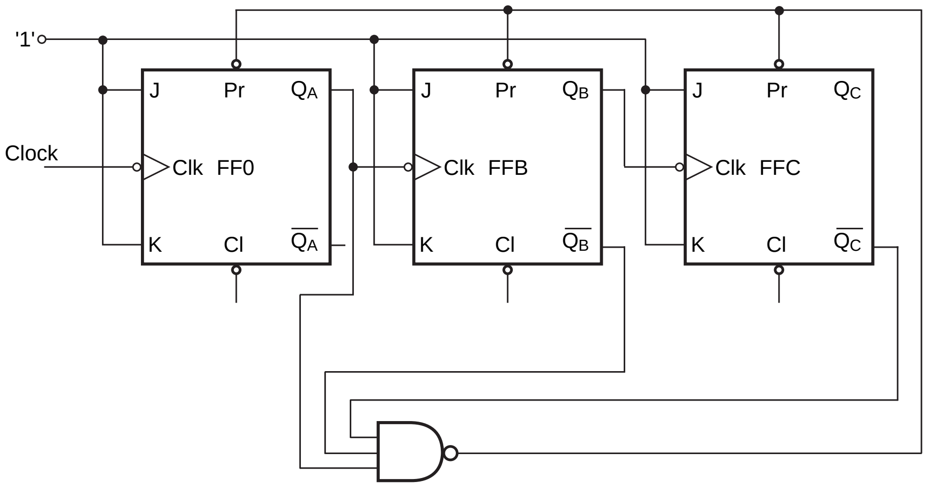
Example 11.5

Refer to the binary ripple counter arrangement of Fig. 11.7. Write its count sequence if it is initially in the 0000 state. Also draw the timing waveforms.

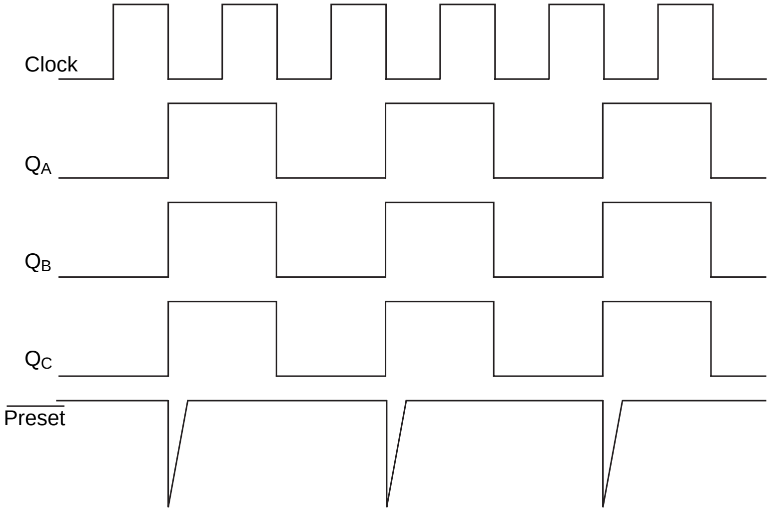
Solution

The counter is initially in the 0000 state. With the first clock pulse, Q_0 toggles from the '0' to the '1' state, which means $\overline{Q_0}$ toggles from '1' to '0'. Since $\overline{Q_0}$ here feeds the clock input of next flip-flop, flip-flop FF1 also toggles. Thus, Q_1 goes from '0' to '1'. Since flip-flops FF2 and FF3 are also clocked from complementary outputs of their immediately preceding flip-flops, they also toggle. Thus, the counter moves from the 0000 state to the 1111 state with the first clock pulse.

With the second clock pulse, Q_0 toggles again, but the other flip-flops remain unaffected for obvious reasons and the counter is in the 1110 state. With subsequent clock pulses, the counter keeps counting downwards by one LSB at a time until it reaches 0000 again, after which the process repeats. The count sequence is given as 0000, 1111, 1110, 1101,1100, 1011, 1010, 1001, 1000,



(a)



(b)

Figure 11.6 Example 11.4.

0111, 0110, 0101, 0100, 0011, 0010, 0001 and 0000. The timing waveforms are shown in Fig. 11.8. Thus, we have a four-bit counter that counts in the reverse sequence, beginning with the maximum count. This is a DOWN counter. This type of counter is discussed further in the subsequent paragraphs.

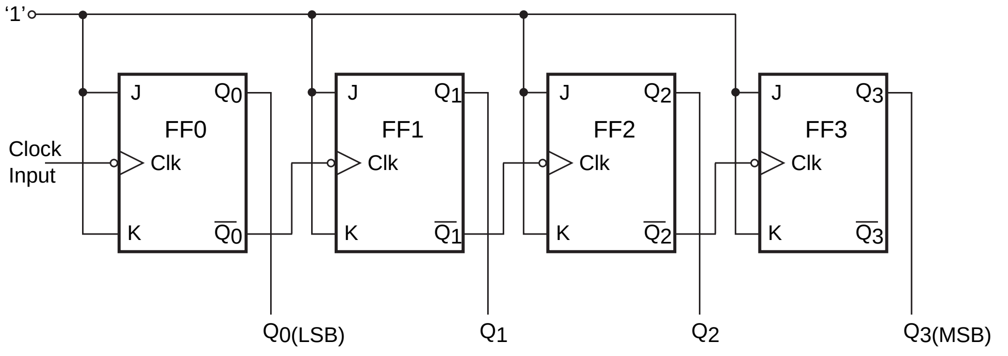


Figure 11.7 Counter schematic, example 11.5.

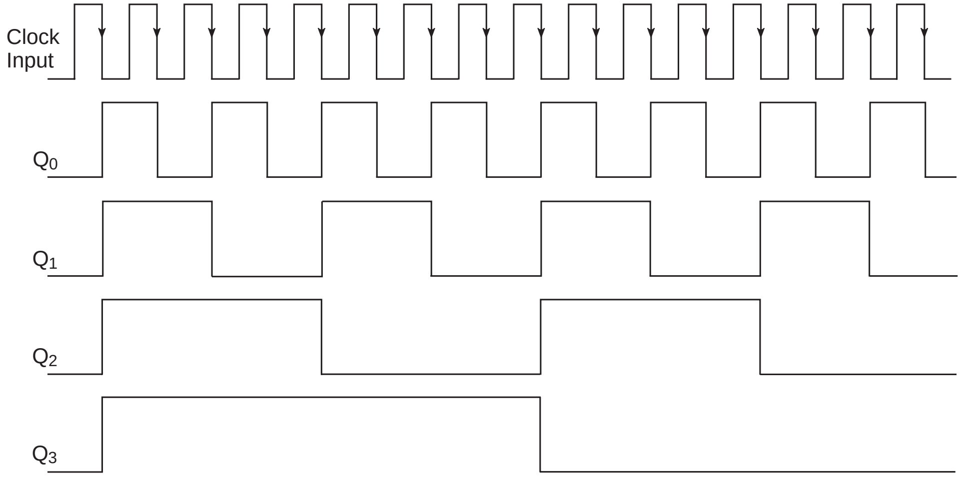


Figure 11.8 Timing waveforms, example 11.5.

From what we have discussed for a binary ripple counter, including the solved examples given to supplement the text, we can make the following observations:

1. If the flip-flops used to construct the counter are negative (HIGH-to-LOW) edge triggered and the clock inputs are fed from Q outputs, the counter counts in the normal upward count sequence.
2. If the flip-flops used to construct the counter are negative edge triggered and the clock inputs are fed from $\overline{Q}$ outputs, the counter counts in the reverse or downward count sequence.
3. If the flip-flops used to construct the counter are positive (LOW-to-HIGH) edge triggered and the clock inputs are fed from Q outputs, the counter counts in the reverse or downward count sequence.
4. If the flip-flops used to construct the counter are positive edge triggered and the clock inputs are fed from the $\overline{Q}$ outputs, the counter counts in the normal upward count sequence.

11.5 Synchronous (or Parallel) Counters

Ripple counters discussed thus far in this chapter are asynchronous in nature as the different flip-flops comprising the counter are not clocked simultaneously and in synchronism with the clock pulses. The total propagation delay in such a counter, as explained earlier, is equal to the sum of propagation delays due to different flip-flops. The propagation delay becomes prohibitively large in a ripple counter with a large count. On the other hand, in a synchronous counter, all flip-flops in the counter are clocked simultaneously in synchronism with the clock, and as a consequence all flip-flops change state at the same time. The propagation delay in this case is independent of the number of flip-flops used.

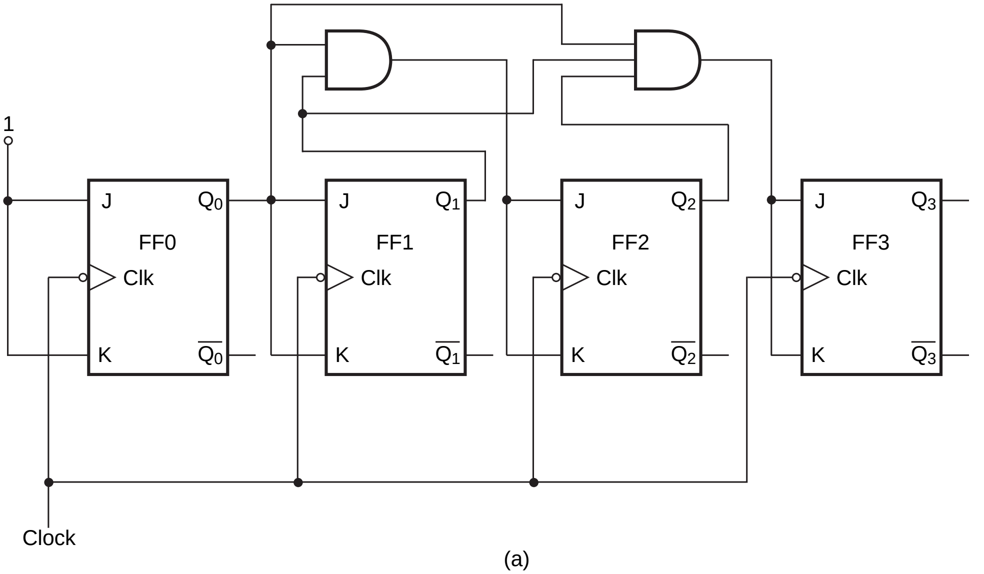
Since the different flip-flops in a synchronous counter are clocked at the same time, there needs to be additional logic circuitry to ensure that the various flip-flops toggle at the right time. For instance, if we look at the count sequence of a four-bit binary counter shown in Table 11.4, we find that flip-flop FF0 toggles with every clock pulse, flip-flop FF1 toggles only when the output of FF0 is in the '1' state, flip-flop FF2 toggles only with those clock pulses when the outputs of FF0 and FF1 are both in the logic '1' state and flip-flop FF3 toggles only with those clock pulses when Q_0, Q_1 and Q_2 are all in the logic '1' state. Such logic can be easily implemented with AND gates. Figure 11.9(a) shows the schematic arrangement of a four-bit synchronous counter. The timing waveforms are shown in Fig. 11.9(b). The diagram is self-explanatory. As an example, ICs 74162 and 74163 are four-bit synchronous counters, with the former being a decade counter and the latter a binary counter.

A synchronous counter that counts in the reverse or downward sequence can be constructed in a similar manner by using complementary outputs of the flip-flops to drive the J and K inputs of the following flip-flops. Refer to the reverse or downward count sequence as given in Table 11.5. As is evident from the table, FF0 toggles with every clock pulse, FF1 toggles only when Q_0 is logic '0', FF2 toggles only when both Q_0 and Q_1 are in the logic '0' state and FF3 toggles only when Q_0, Q_1 and Q_2 are in the logic '0' state.

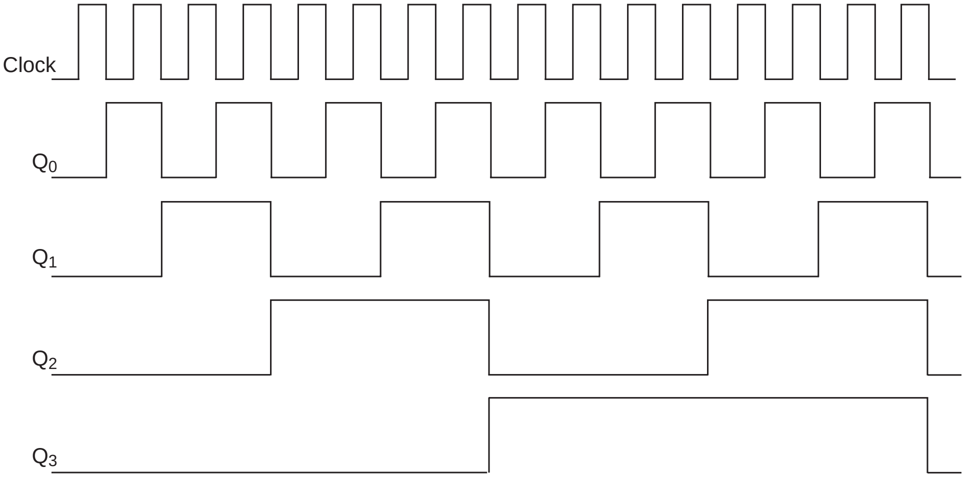
Referring to the four-bit synchronous UP counter of Fig. 11.9(a), if the J and K inputs of flip-flop FF1 are fed from the $\overline{Q_0}$ output instead of the Q_0 output, the inputs to the two-input AND gate are $\overline{Q_0}$ and $\overline{Q_1}$ instead of Q_0 and Q_1, and the inputs to the three-input AND gate are $\overline{Q_0}$, $\overline{Q_1}$ and $\overline{Q_2}$ instead of Q_0, Q_1 and $Q_{2,}$, we get a counter that counts in reverse order. In that case it becomes a four-bit synchronous DOWN counter.

Table 11.4 Count sequence of a four-bit binary counter.

Count	Q_3	Q_2	Q_1	Q_0	Count	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	8	1	0	0	0
1	0	0	0	1	9	1	0	0	1
2	0	0	1	0	10	1	0	1	0
3	0	0	1	1	11	1	0	1	1
4	0	1	0	0	12	1	1	0	0
5	0	1	0	1	13	1	1	0	1
6	0	1	1	0	14	1	1	1	0
7	0	1	1	1	15	1	1	1	1



(a)



(b)

Figure 11.9 Four-bit synchronous counter.

Table 11.5 Reverse or downward count sequence synchronous counter.

Count	Q_3	Q_2	Q_1	Q_0	Count	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	8	1	0	0	0
1	1	1	1	1	9	0	1	1	1
2	1	1	1	0	10	0	1	1	0
3	1	1	0	1	11	0	1	0	1
4	1	1	0	0	12	0	1	0	0
5	1	0	1	1	13	0	0	1	1
6	1	0	1	0	14	0	0	1	0
7	1	0	0	1	15	0	0	0	1

11.6 UP/DOWN Counters

Counters are also available in integrated circuit form as UP/DOWN counters, which can be made to operate as either UP or DOWN counters. As outlined in Section 11.5, an UP counter is one that counts upwards or in the forward direction by one LSB every time it is clocked. A four-bit binary UP counter will count as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 0000, 0001, . . . and so on. A DOWN counter counts in the reverse direction or downwards by one LSB every time it is clocked. The four-bit binary DOWN counter will count as 0000, 1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000, 0111, 0110, 0101, 0100, 0011, 0010, 0001, 0000, 1111, . . . and so on.

Some counter ICs have separate clock inputs for UP and DOWN counts, while others have a single clock input and an UP/DOWN control pin. The logic status of this control pin decides the counting mode. As an example, ICs 74190 and 74191 are four-bit UP/DOWN counters in the TTL family with a single clock input and an UP/DOWN control pin. While IC 74190 is a BCD decade counter, IC 74191 is a binary counter. Also, ICs 74192 and 74193 are four-bit UP/DOWN counters in the TTL family, with separate clock input terminals for UP and DOWN counts. While IC 74192 is a BCD decade counter, IC 74193 is a binary counter.

Figure 11.10 shows a three-bit binary UP/DOWN counter. This is only one possible logic arrangement. As we can see, the counter counts upwards when UP control is logic ‘1’ and DOWN

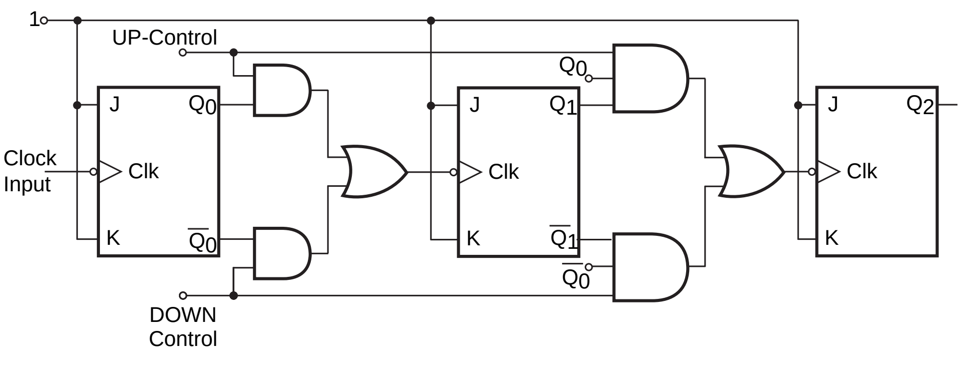


Figure 11.10 Four-bit UP/DOWN counter.

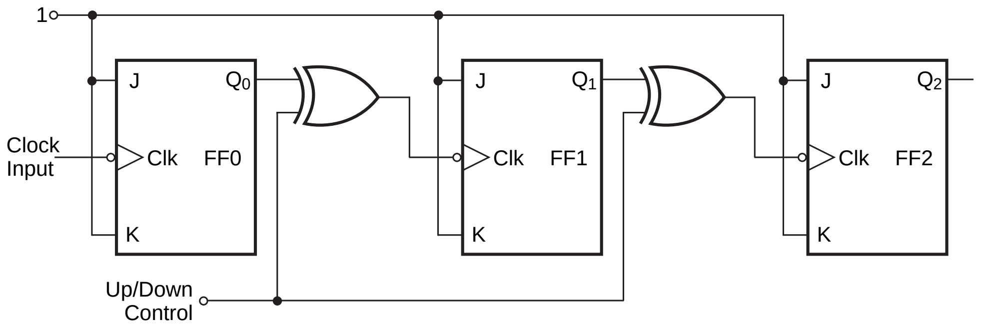


Figure 11.11 Three-bit UP/DOWN counter with a common clock input.

control is logic '0'. In this case the clock input of each flip-flop other than the LSB flip-flop is fed from the normal output of the immediately preceding flip-flop. The counter counts downwards when the UP control input is logic '0' and DOWN control is logic '1'. In this case, the clock input of each flip-flop other than the LSB flip-flop is fed from the complemented output of the immediately preceding flip-flop. Figure 11.11 shows another possible configuration for a three-bit binary ripple UP/DOWN counter. It has a common control input. When this input is in logic '1' state the counter counts downwards, and when it is in logic '0' state it counts upwards.

11.7 Decade and BCD Counters

A *decade counter* is one that goes through 10 unique output combinations and then resets as the clock proceeds further. Since it is an MOD-10 counter, it can be constructed with a minimum of four flip-flops. A four-bit counter would have 16 states. By skipping any of the six states by using some kind of feedback or some kind of additional logic, we can convert a normal four-bit binary counter into a decade counter. A decade counter does not necessarily count from 0000 to 1001. It could even count as 0000, 0001, 0010, 0101, 0110, 1001, 1010, 1100, 1101, 1111, 0000, . . . In this count sequence, we have skipped 0011, 0100, 0111, 1000, 1011 and 1110.

A *BCD counter* is a special case of a decade counter in which the counter counts from 0000 to 1001 and then resets. The output weights of flip-flops in these counters are in accordance with 8421-code. For instance, at the end of the seventh clock pulse, the counter output will be 0111, which is the binary equivalent of decimal 7. In other words, different counter states in this counter are binary equivalents of the decimal numbers 0 to 9. These are different from other decade counters that provide the same count by using some kind of forced feedback to skip six of the natural binary counts.

11.8 Presettable Counters

Presettable counters are those that can be preset to any starting count either asynchronously (independently of the clock signal) or synchronously (with the active transition of the clock signal). The presetting operation is achieved with the help of PRESET and CLEAR (or MASTER RESET) inputs available on the flip-flops. The presetting operation is also known as the 'preloading' or simply the 'loading' operation.

Presettable counters can be UP counters, DOWN counters or UP/DOWN counters. Additional inputs/outputs available on a presettable UP/DOWN counter usually include PRESET inputs, from where any desired count can be loaded, parallel load (*PL*) inputs, which when active allow the PRESET inputs to be loaded onto the counter outputs, and terminal count (*TC*) outputs, which become active when the counter reaches the terminal count.

Figure 11.12 shows the logic diagram of a four-bit presettable synchronous UP counter. The data available on P_3, P_2, P_1 and P_0 inputs are loaded onto the counter when the parallel load ($\overline{PL}$) input goes LOW.

When the $\overline{PL}$ input goes LOW, one of the inputs of all NAND gates, including the four NAND gates connected to the PRESET inputs and the four NAND gates connected to the CLEAR inputs, goes to the logic '1' state. What reaches the PRESET inputs of FF3, FF2, FF1 and FF0 is $\overline{P_3}$, $\overline{P_2}$, $\overline{P_1}$ and $\overline{P_0}$ respectively, and what reaches their CLEAR inputs is P_3, P_2, P_1 and P_0 respectively. Since PRESET and CLEAR are active LOW inputs, the counter flip-flops FF3, FF2, FF1 and FF0 will respectively be loaded with P_3, P_2, P_1 and P_0. For example, if $P_3 = 1$, the PRESET and CLEAR inputs of FF3 will be in the '0' and '1' logic states respectively. This implies that the Q_3 output will go to the logic '1' state. Thus, FF3 has been loaded with P_3. Similarly, if $P_3 = 0$, the PRESET and CLEAR inputs of flip-flop FF3 will be in the '1' and '0' states respectively. The flip-flop output (Q_3 output) will be cleared to the '0' state. Again, the flip-flop is loaded with P_3 logic status when the $\overline{PL}$ input becomes active.

Counter ICs 74190, 74191, 74192 and 74193 are asynchronously presettable synchronous UP/DOWN counters. Many synchronous counters use synchronous presetting whereby the counter is preset or loaded with the data on the active transition of the same clock signal that is used for counting. Presettable counters also have terminal count ($\overline{TC}$) outputs, which allow them to be cascaded together to get counters with higher MOD numbers. In the cascade arrangement, the terminal count output of the lower-order counter feeds the clock input of the next higher-order counter. Cascading of counters is discussed in Section 11.10.

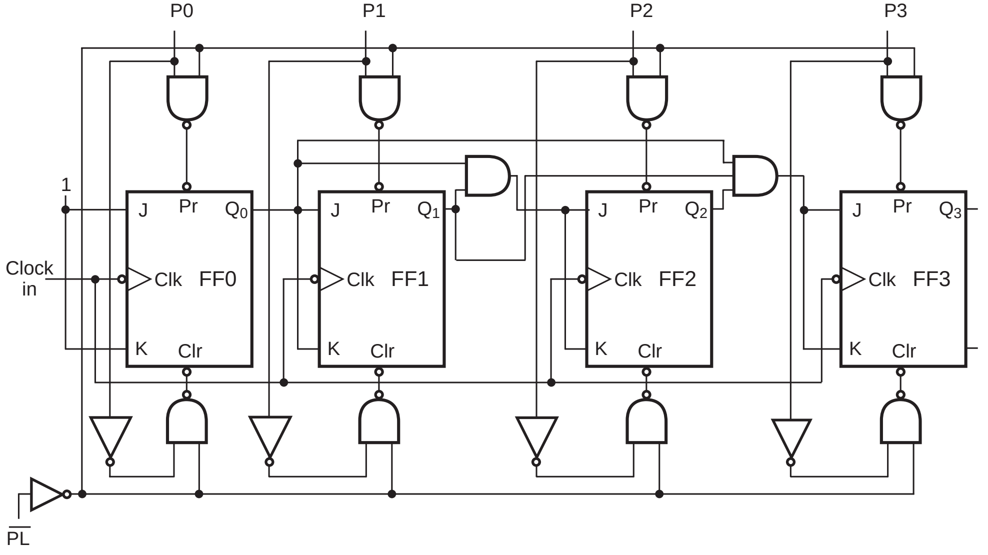


Figure 11.12 Four-bit presettable, clearable counter.

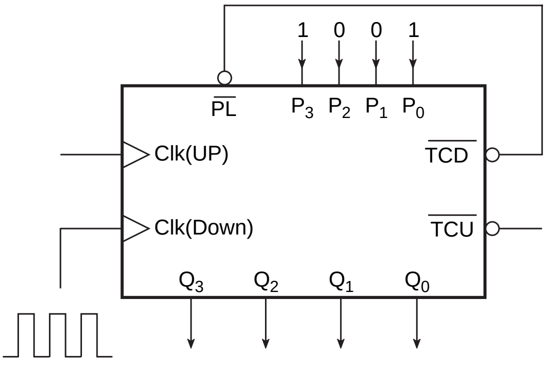


Figure 11.13 Presettable four-bit counter.

11.8.1 Variable Modulus with Presettable Counters

Presettable counters can be wired as counters with a modulus of less than 2^N without the need for any additional logic circuitry. When a presettable counter is preset with a binary number whose decimal equivalent is some number 'X', and if this counter is wired as a DOWN counter, with its terminal count (DOWN mode) output, also called borrow-out (B_o), fed back to the parallel load (PL) input, it works like an MOD-X counter.

We will illustrate this with the help of an example. Refer to Fig. 11.13. It shows a presettable four-bit synchronous UP/DOWN binary counter having separate clock inputs for UP and DOWN counting (both positive edge triggered), an active LOW parallel load input ($\overline{PL}$) and active LOW terminal count UP ($\overline{TCU}$) and terminal count DOWN ($\overline{TCD}$) outputs. This description is representative of IC counter type 74193. Let us assume that the counter is counting down and is presently in the 1001 state at time instant t_0. The $\overline{TCD}$ output is in the logic '1' state, and so is the $\overline{PL}$ input. That is, both are inactive. The counter counts down by one LSB at every positive-going edge of the clock input. Immediately after the ninth positive-going trigger (at time instant t_9), the counter is in the 0000 state, which is the terminal count. Coinciding with the negative-going edge of the same clock pulse, the $\overline{TCD}$ output goes to the logic '0' state, and so does the $\overline{PL}$ input. This loads the counter with 1001 at time instant t_{10}, as shown in the timing waveforms of Fig. 11.14. With the positive-going edges of the tenth clock pulse and thereafter, the counter repeats its DOWN count sequence. Examination of the Q_3 output waveform tells that its frequency is one-ninth of the input clock frequency. Thus, it is an MOD-9 counter. The modulus of the counter can be varied by varying the data loaded onto the parallel PRESET/LOAD inputs.

11.9 Decoding a Counter

The output state of a counter at any time instant, as it is being clocked, is in the form of a sequence of binary digits. For a large number of applications, it is important to detect or decode different states of the counter whose number equals the modulus of the counter. One typical application could be a need to initiate or trigger some action after the counter reaches a specific state. The decoding network therefore is going to be a logic circuit that takes its inputs from the outputs of the different flip-flops constituting the counter and then makes use of those data to generate outputs equal to the modulus or MOD-number of the counter.

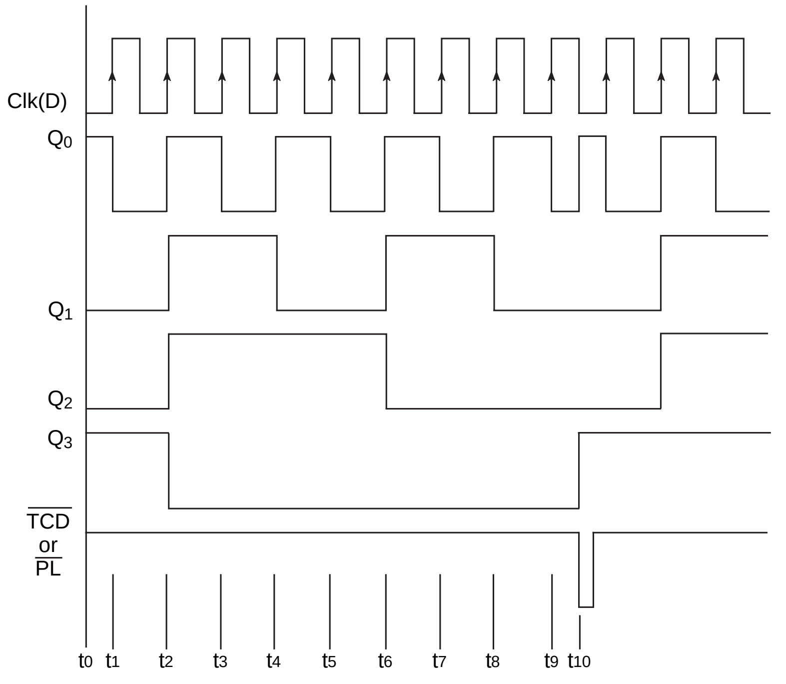


Figure 11.14 Timing waveforms for the counter of Fig. 11.13.

Depending upon the logic status of the decoded output, there are two basic types of decoding, namely *active HIGH* decoding and *active LOW* decoding. In the case of the former the decoder outputs are normally LOW, and for a given counter state the corresponding decoder output goes to the logic HIGH state. In the case of active LOW decoding, the decoder outputs are normally HIGH and the decoded output representing the counter state goes to the logic LOW state.

We will further illustrate the concept of decoding a counter with the help of an example. Consider the two-stage MOD-4 ripple counter of Fig. 11.15(a). This counter has four possible logic states, which need to be decoded. These include 00, 01, 10 and 11. Let us now consider the arrangement of four two-input AND gates as shown in Fig. 11.15(b) and what their outputs look like as the counter clock goes through the first four pulses. Before we proceed further, we have two important observations to make. Firstly, the number of AND gates used in the decoder network equals the number of logic states to be decoded, which further equals the modulus of the counter. Secondly, the number of inputs to each AND gate equals the number of flip-flops used in the counter. We can see from the waveforms of Fig. 11.15(b) that, when the counter is in the 00 state, the AND gate designated '0' is in the logic HIGH state and the outputs of the other gates designated '1', '2' and '3' are in the logic LOW state. Similarly, for 01, 10 and 11 states of the counter, the outputs of gates 1, 2 and 3 are respectively in the logic HIGH state. This is incidentally active HIGH decoding. We can visualize that, if the AND gates were replaced with NAND gates, with the inputs to the gates remaining the same, we would get an active LOW decoder. For a counter that uses N flip-flops and has a modulus of 'X', the decoder will have 'X' number of N-input AND or NAND gates, depending upon whether we want an active HIGH or active LOW decoder.

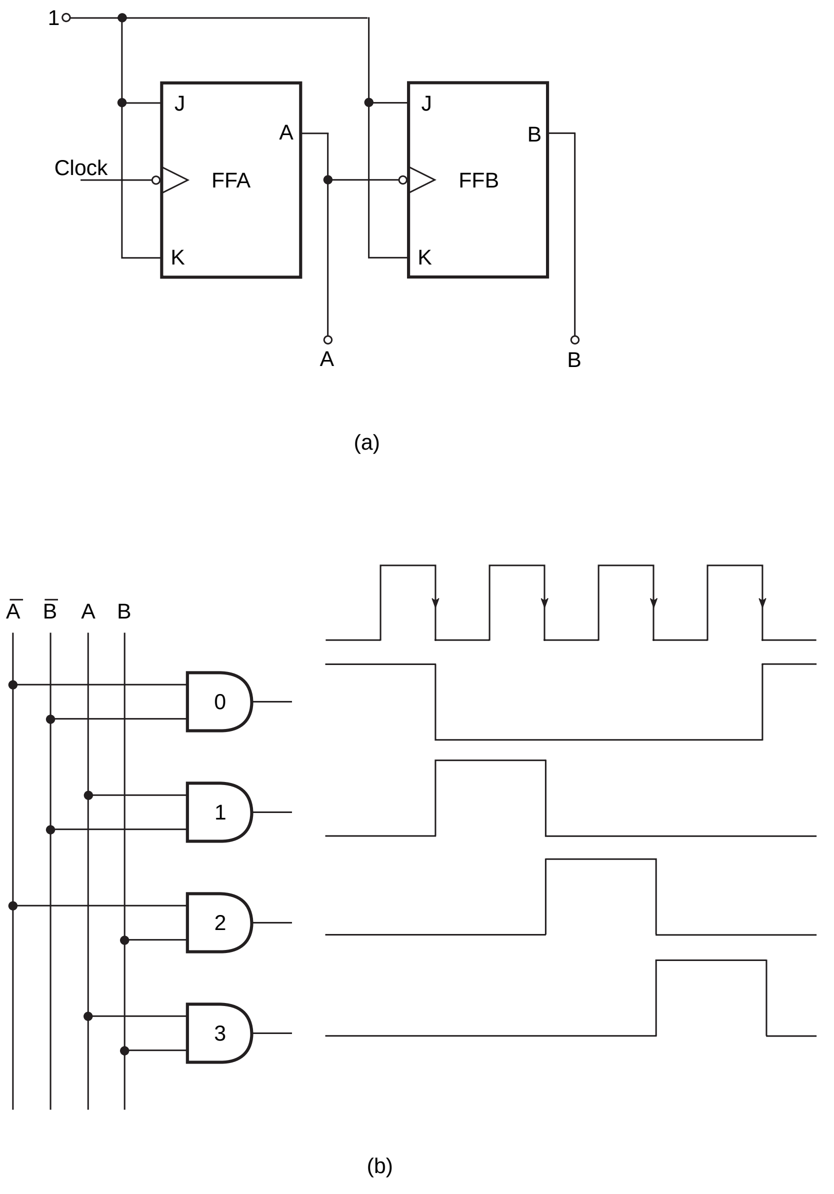


Figure 11.15 MOD-4 ripple counter with decoding logic.

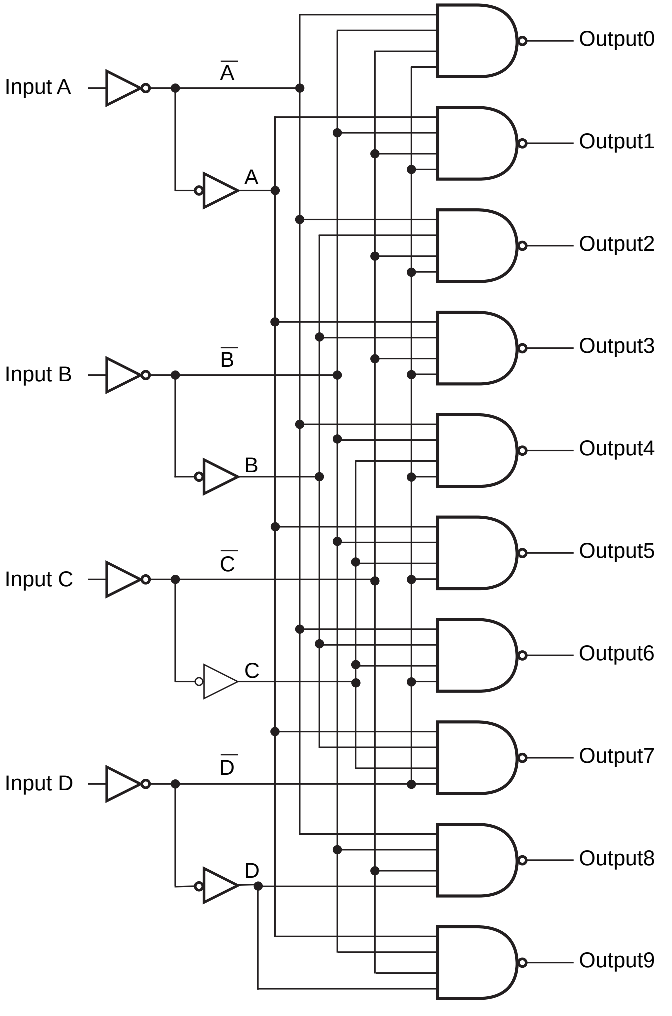


Figure 11.16 Logic diagram of four-line BCD-to-decimal decoder (IC 7442).

Figure 11.16 shows the logic diagram of a four-line BCD to decimal decoder with active low outputs. Full decoding of valid input logic states ensures that all outputs remain off or inactive for all invalid input conditions. Table 11.6 gives the functional table of the decoder of Fig. 11.16. The logic diagram shown in Fig. 11.16 is the actual logic diagram of IC 7442, which is a four-line BCD to decimal decoder in the TTL family.

The decoding gates used to decode the states of a ripple counter produce glitches (or spikes) in the decoded waveforms. These glitches basically result from the cumulative propagation delay as we move from one flip-flop to the next in a ripple counter. It can be best illustrated with the help of the MOD-4 counter shown in Fig. 11.17. The timing waveforms are shown in Fig. 11.18 and are self-explanatory.

Table 11.6 Functional table of the decoder of Fig. 11.16.

Decimal number	BCD input				Decimal output									
	D	*C*	*B*	*A*	0	1	2	3	4	5	6	7	8	9
0	L	L	L	L	L	H	H	H	H	H	H	H	H	H
1	L	L	L	H	H	L	H	H	H	H	H	H	H	H
2	L	L	H	L	H	H	L	H	H	H	H	H	H	H
3	L	L	H	H	H	H	H	L	H	H	H	H	H	H
4	L	H	L	L	H	H	H	H	L	H	H	H	H	H
5	L	H	L	H	H	H	H	H	H	L	H	H	H	H
6	L	H	H	L	H	H	H	H	H	H	L	H	H	H
7	L	H	H	H	H	H	H	H	H	H	H	L	H	H
8	H	L	L	L	H	H	H	H	H	H	H	H	L	H
9	H	L	L	H	H	H	H	H	H	H	H	H	H	L
Invalid	H	L	H	L	H	H	H	H	H	H	H	H	H	H
Invalid	H	L	H	H	H	H	H	H	H	H	H	H	H	H
Invalid	H	H	L	L	H	H	H	H	H	H	H	H	H	H
Invalid	H	H	L	H	H	H	H	H	H	H	H	H	H	H
Invalid	H	H	H	L	H	H	H	H	H	H	H	H	H	H
Invalid	H	H	H	H	H	H	H	H	H	H	H	H	H	H

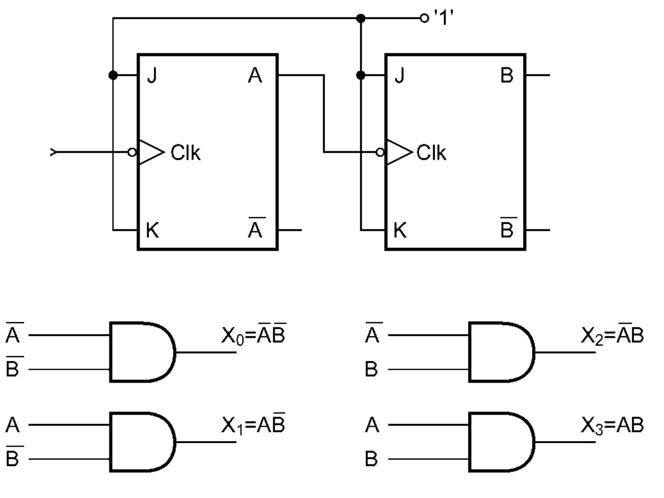


Figure 11.17 MOD-4 counter with decoding gates.

We can see the appearance of glitches at the output of decoding gates that decode X_0 and X_2 states.This problem for all practical purposes is absent in synchronous counters. Theoretically, it can even exist in a synchronous counter if the flip-flops used have different propagation delays.

One way to overcome this problem is to use a strobe signal which keeps the decoding gates disabled until all flip-flops have reached a stable state in response to the relevant clock transition. To implement

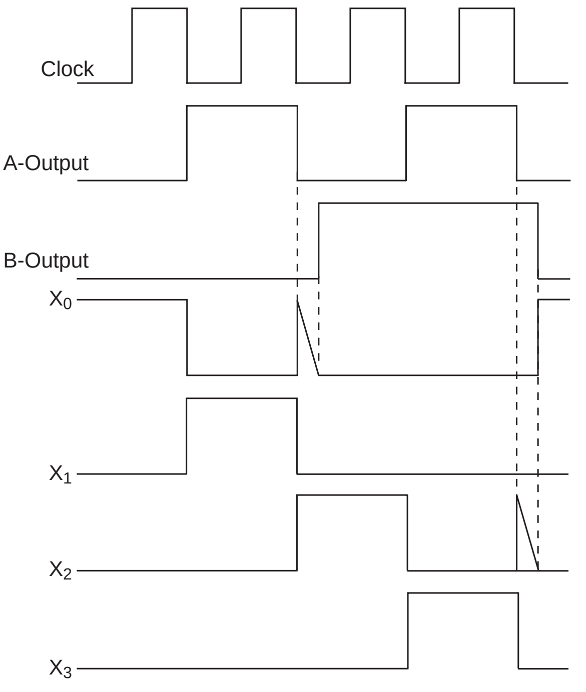


Figure 11.18 Glitch problem in decoders.

this, each of the decoding gates will have an additional input. This additional input of all decoding gates is tied together and the strobe signal applied to the common point.

One such decoder with additional strobe inputs to take care of glitch-related problems is IC 74154, which is a four-line to 16-line decoder in the TTL family. Figure 11.19 shows the internal logic diagram of IC 74154. We can see all NAND gates having an additional input line, which is controlled by strobe inputs $\overline{G}_1$ and $\overline{G}_2$.

11.10 Cascading Counters

A cascade arrangement allows us to build counters with a higher modulus than is possible with a single stage. The terminal count outputs allow more than one counter to be connected in a cascade arrangement. In the following paragraphs, we will examine some such cascade arrangements in the case of binary and BCD counters.

11.10.1 Cascading Binary Counters

In order to construct a multistage UP counter, all counter stages are connected in the count UP mode. The clock is applied to the clock input of a lowest-order counter, the terminal count UP (*TCU*), also called the carry-out (C_o), of this counter is applied to the clock input of the next higher counter stage

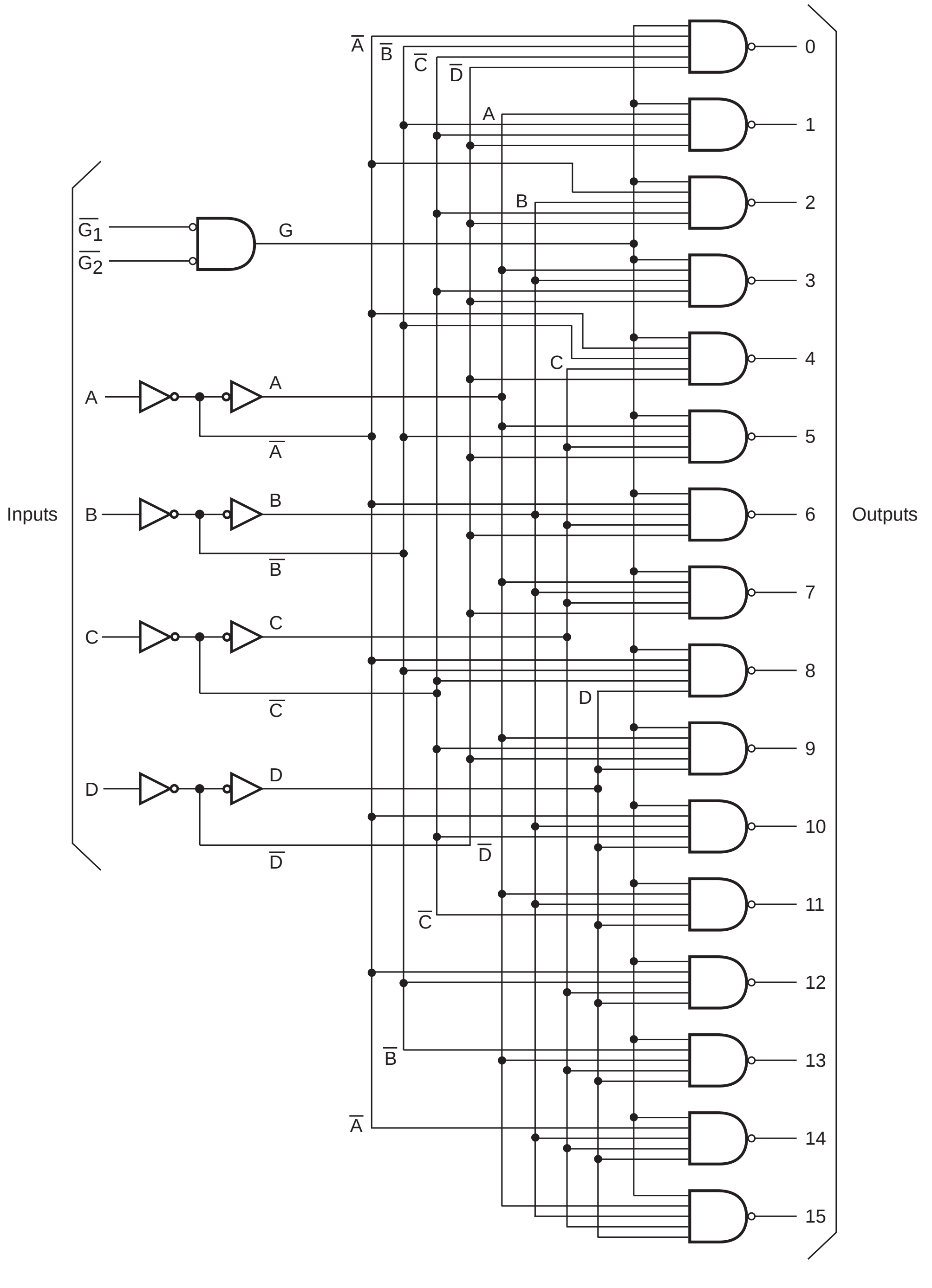


Figure 11.19 Logic diagram of IC 74154.

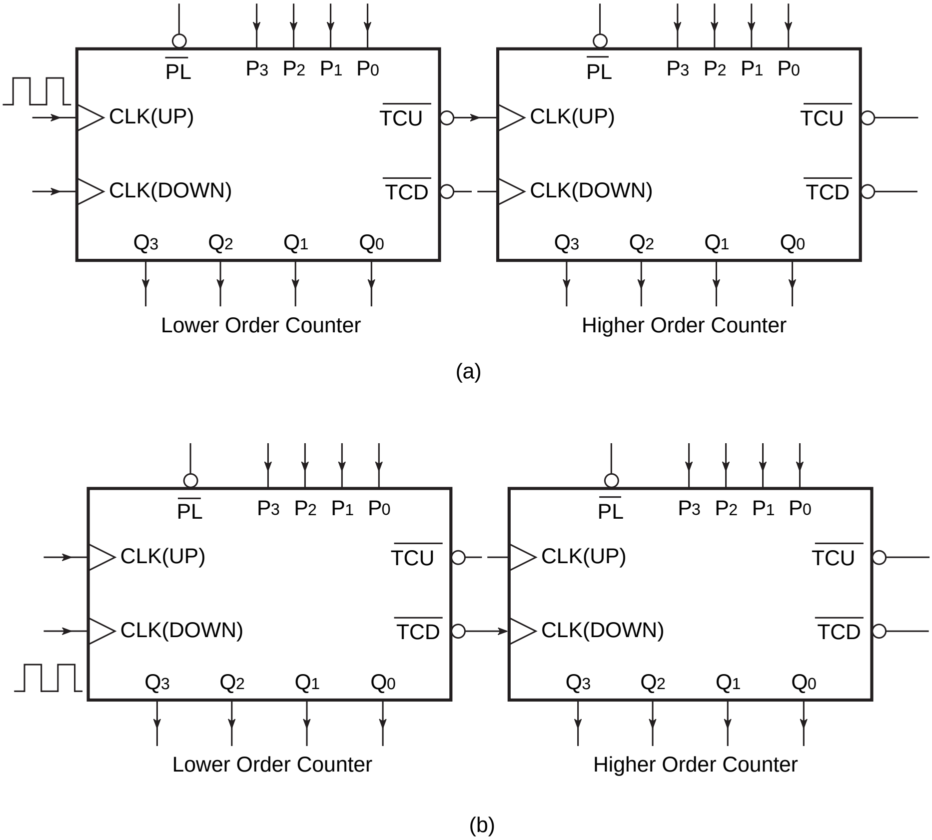


Figure 11.20 Cascading binary counters.

and the process continues. If it is desired to build a multistage DOWN counter, all counters are wired as DOWN counters, the clock is applied to the clock input of the lowest-order counter and the terminal count DOWN (*TCD*), also called the borrow-out (B_o), of the lowest-order counter is applied to the clock input of the next higher counter stage. The process continues in the same fashion, with the TCD output of the second stage feeding the clock input of the third stage and so on. The modulus of the multistage counter arrangement equals the product of the moduli of individual stages. Figures 11.20(a) and (b) respectively show two-stage arrangements of four-bit synchronous UP and DOWN counters respectively.

11.10.2 Cascading BCD Counters

BCD counters are used when the application involves the counting of pulses and the result of counting is to be displayed in decimal. A single-stage BCD counter counts from 0000 (decimal equivalent '0') to 1001 (decimal equivalent '9') and thus is capable of counting up to a maximum of nine pulses. The output in a BCD counter is in binary coded decimal (BCD) form. The BCD output needs

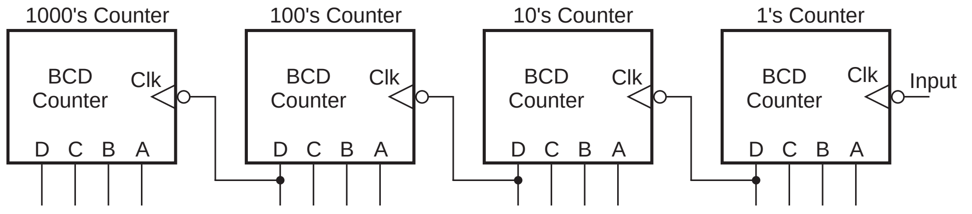


Figure 11.21 Cascading BCD counters.

to be decoded appropriately before it can be displayed. Decoding a counter has been discussed in the previous section. Coming back to the question of counting pulses, more than one BCD counter stage needs to be used in a cascade arrangement in order to be able to count up to a larger number of pulses. The number of BCD counter stages to be used equals the number of decimal digits in the maximum number of pulses we want to count up to. With a maximum count of 9999 or 3843, both would require a four-stage BCD counter arrangement with each stage representing one decimal digit.

Figure 11.21 shows a cascade arrangement of four BCD counter stages. The arrangement works as follows. Initially, all four counters are in the all 0s state. The counter representing the decimal digit of 1's place is clocked by the pulsed signal that needs to be counted. The successive flip-flops are clocked by the MSB of the immediately previous counter stage. The first nine pulses take 1's place counter to 1001. The tenth pulse resets it to 0000, and '1' to '0' transition at the MSB of 1's place counter clocks 10's place counter. 10's place counter gets clocked on every tenth input clock pulse. On the hundredth clock pulse, the MSB of 10's counter makes a '1' to '0' transition which clocks 100's place counter. This counter gets clocked on every successive hundredth input clock pulse. On the thousandth input clock pulse, the MSB of 100's counter makes 'l' to '0' transition for the first time and clocks 1000's place counter. This counter is clocked thereafter on every successive thousandth input clock pulse. With this background, we can always tell the output state of the cascade arrangement. For example, immediately after the 7364th input clock pulse, the state of 1000's, 100's, 10's and 1's BCD counters would respectively be 0111, 0011, 0110 and 0100.

Example 11.6

Figure 11.22 shows a cascade arrangement of two 74190s. Both the UP/DOWN counters are wired as UP counters. What will be the logic status of outputs designated as A, B, C, D, E, F, G and H after the 34th clock pulse?

Solution

The cascade arrangement basically constitutes a two-stage BCD counter that can count from 0 to 99. The counter shown on the left forms 1's place counter, while the one on the right is 10's place counter. The ripple clock ($\overline{RC}$) output internally enabled by the terminal count ($\overline{\text{TC}}$) clocks 10's place counter on the tenth clock pulse and thereafter on every successive tenth clock pulse. At the end of the 34th clock pulse, 1's counter stores the binary equivalent of '4' and 10's counter stores the binary equivalent of '3'. Therefore, the logic status of A, B, C, D, E, F, G and H outputs will be 0, 0, 1, 0, 1,1, 0 and 0 respectively.

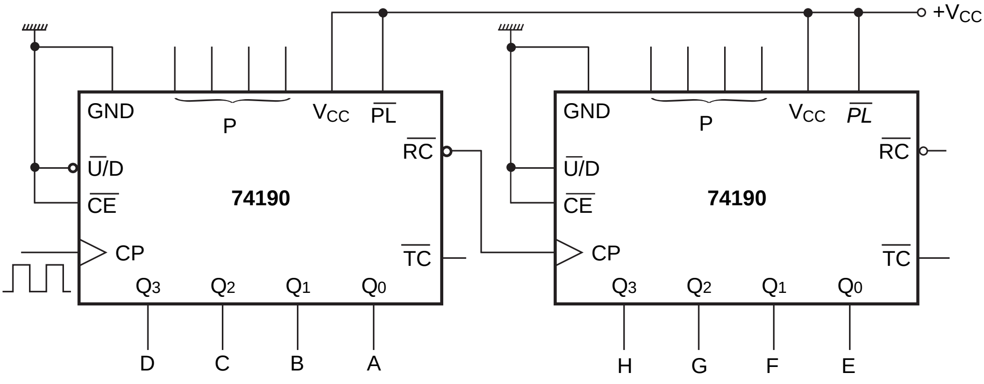


Figure 11.22 Cascade arrangement of two 74190s (example 11.6).

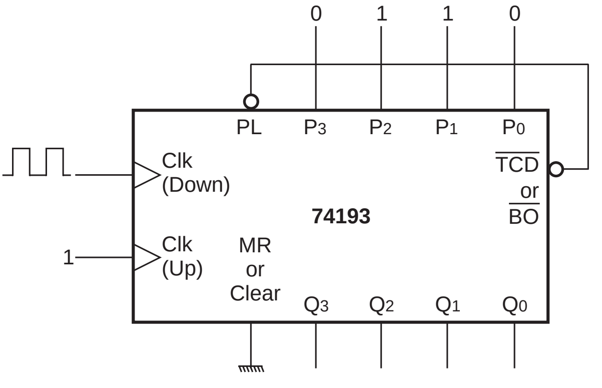


Figure 11.23 Presettable counter (example 11.7).

Example 11.7

Determine the modulus of the presettable counter shown in Fig. 11.23. If the counter were initially in the 0110 state, what would be the state of the counter immediately after the eighth clock pulse be?

Solution

- This presettable counter has been wired as a DOWN counter.
- The preset data input is 0110.
- Therefore, the modulus of the counter is 6 (the decimal equivalent of 0110).
- Now, the counter is initially in the 0110 state.
- Therefore, at the end of the sixth clock pulse, immediately after the leading edge of the sixth clock pulse, the counter will be in the 0000 state.

- A HIGH-to-LOW transition at the $\overline{TCD}$ output,coinciding with the trailing edge of the sixth clock pulse, loads 0110 to the counter output.
- Therefore, immediately after the leading edge of the eighth clock pulse, the counter will be in the 0100 state.

11.11 Designing Counters with Arbitrary Sequences

So far we have discussed different types of synchronous and asynchronous counters. A large variety of synchronous and asynchronous counters are available in IC form, and some of these have been mentioned and discussed in the previous sections. The counters discussed hitherto count in either the normal binary sequence with a modulus of 2^N or with slightly altered binary sequences where one or more of the states are skipped. The latter type of counter has a modulus of less than 2^N, N being the number of flip-flops used. Nevertheless, even these counters have a sequence that is either upwards or downwards and not arbitrary. There are applications where a counter is required to follow a sequence that is arbitrary and not binary. As an example, an MOD-10 counter may be required to follow the sequence 0000, 0010, 0101, 0001, 0111, 0011, 0100, 1010, 1000, 1111, 0000, 0010 and so on. In such cases, the simple and seemingly obvious feedback arrangement with a single NAND gate discussed in the earlier sections of this chapter for designing counters with a modulus of less than 2^N cannot be used.

There are several techniques for designing counters that follow a given arbitrary sequence. In the present section, we will discuss in detail a commonly used technique for designing synchronous counters using *J-K* flip-flops or *D* flip-flops. The design of the counters basically involves designing a suitable combinational logic circuit that takes its inputs from the normal and complemented outputs of the flip-flops used and decodes the different states of the counter to generate the correct logic states for the inputs of the flip-flops such as J, K, D, etc. But before we illustrate the design procedure with the help of an example, we will explain what we mean by the excitation table of a flip-flop and the state transition diagram of a counter. An excitation table in fact can be drawn for any sequential logic circuit, but, once we understand what it is in the case of a flip-flop, which is the basic building block of sequential logic, it would be much easier for us to draw the same for more complex sequential circuits such as counters, etc.

11.11.1 Excitation Table of a Flip-Flop

The excitation table is similar to the characteristic table that we discussed in the previous chapter on flip-flops. The excitation table lists the present state, the desired next state and the flip-flop inputs (J, K, D, etc.) required to achieve that. The same for a *J-K* flip-flop and a *D* flip-flop are shown in Tables 11.7 and 11.8 respectively. Referring to Table 11.7, if the output is in the logic '0' state and it is desired that it goes to the logic '1' state on occurrence of the clock pulse, the J input must be in the logic '1' state and the K input can be either in the logic '0' or logic '1' state. This is true as, for a '0' to '1' transition, there are two possible input conditions that can achieve this. These are $J = 1$, $K = 0$ (SET mode) and $J = K = 1$ (toggle mode), which further leads to $J = 1,\ K = X$ (either 0 or 1). The other entries of the excitation table can be explained on similar lines.

In the case of a *D* flip-flop, the *D* input is the same as the logic status of the desired next state. This is true as, in the case of a *D* flip-flop, the *D* input is transferred to the output on the occurrence of the clock pulse, irrespective of the present logic status of the Q output.

Table 11.7 Excitation table of a *J-K* flip-flop.

Present state (Q_n)	Next state (Q_{n+1})	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Table 11.8 Excitation table of a *D* flip-flop.

Present state (Q_n)	Next state (Q_{n+1})	*D*
0	0	0
0	1	1
1	0	0
1	1	1

11.11.2 State Transition Diagram

The state transition diagram is a graphical representation of different states of a given sequential circuit and the sequence in which these states occur in response to a clock input. Different states are represented by circles, and the arrows joining them indicate the sequence in which different states occur. As an example, Fig. 11.24 shows the state transition diagram of an MOD-8 binary counter.

11.11.3 Design Procedure

We will illustrate the design procedure with the help of an example. We will do this for an MOD-6 synchronous counter design, which follows the count sequence 000, 010, 011, 001, 100, 110, 000, 010, . . . :

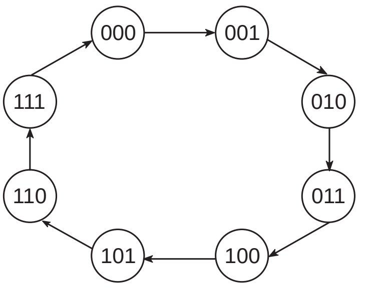


Figure 11.24 State transition diagram for an MOD-8 binary counter.

1. Determine the number of flip-flops required for the purpose. Identify the undesired states. In the present case, the number of flip-flops required is 3 and the undesired states are 101 and 111
2. Draw the state transition diagram showing all possible states including the ones that are not desired. The undesired states should be depicted to be transiting to any of the desired states. We have chosen the 000 state for this purpose. It is important to include the undesired states to ensure that, if the counter accidentally gets into any of these undesired states owing to noise or power-up, the counter will go to a desired state to resume the correct sequence on application of the next clock pulse. Figure 11.25 shows the state transition diagram

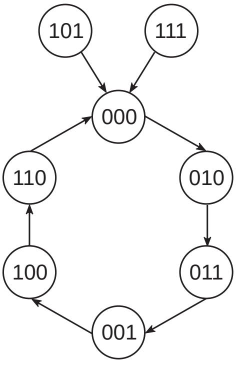


Figure 11.25 State transition diagram.

3. Draw the excitation table for the counter, listing the present states, the next states corresponding to the present states and the required logic status of the flip-flop inputs (the J and K inputs if the counter is to be implemented with J-K flip-flops). The excitation table is shown in Table 11.9

Table 11.9 Excitation table.

Present state			Next state			Inputs					
C	B	A	C	B	A	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	1	0	0	X	1	X	0	X
0	0	1	1	0	0	1	X	0	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	0	0	1	0	X	X	1	X	0
1	0	0	1	1	0	X	0	1	X	0	X
1	0	1	0	0	0	X	1	0	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X
1	1	1	0	0	0	X	1	X	1	X	1

The circuit excitation table can be drawn very easily once we know the excitation table of the flip-flop to be used for building the counter. For instance, let us look at the first row of the excitation table (Table 11.9). The counter is in the 000 state and is to go to 010 on application of a clock pulse. That is, the normal outputs of C, B and A flip-flops have to undergo '0' to '0', '0' to '1' and '0' to '0' transitions respectively. Referring to the excitation table of a *J-K* flip-flop, the desired transitions can be realized if the logic status of J_A, K_A, J_B, K_B, J_C and K_C is as shown in the excitation table.

4. The next step is to design the logic circuits for generating J_A, K_A, J_B, K_B, J_C and K_C inputs from available A, $\overline{A}$, B, $\overline{B}$, C and $\overline{C}$ outputs. This can be done by drawing Karnaugh maps for each one of the inputs, minimizing them and then implementing the minimized Boolean expressions. The Karnaugh maps for J_A, K_A, J_B, K_B, J_C and K_C are respectively shown in Figs 11.26(a), (b), (c), (d), (e) and (f). The minimized Boolean expressions are as follows:

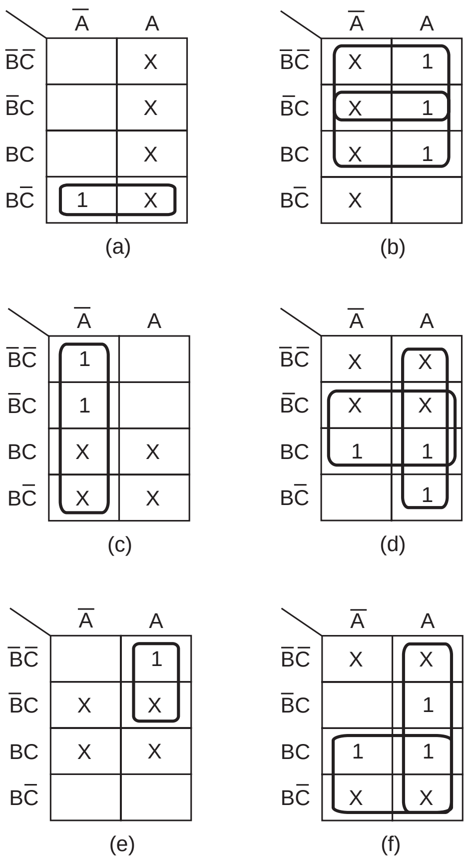


Figure 11.26 Karnaugh maps.

$$J_A = B.\overline{C} \tag{11.2}$$

$$K_A = \overline{B} + C \tag{11.3}$$

$$J_B = \overline{A} \tag{11.4}$$

$$K_B = A + C \tag{11.5}$$

$$J_C = A.\overline{B} \tag{11.6}$$

$$K_C = A + B \tag{11.7}$$

The above expressions can now be used to implement combinational circuits to generate J_A, K_A, J_B, K_B, J_C and K_C inputs. Figure 11.27 shows the complete counter circuit

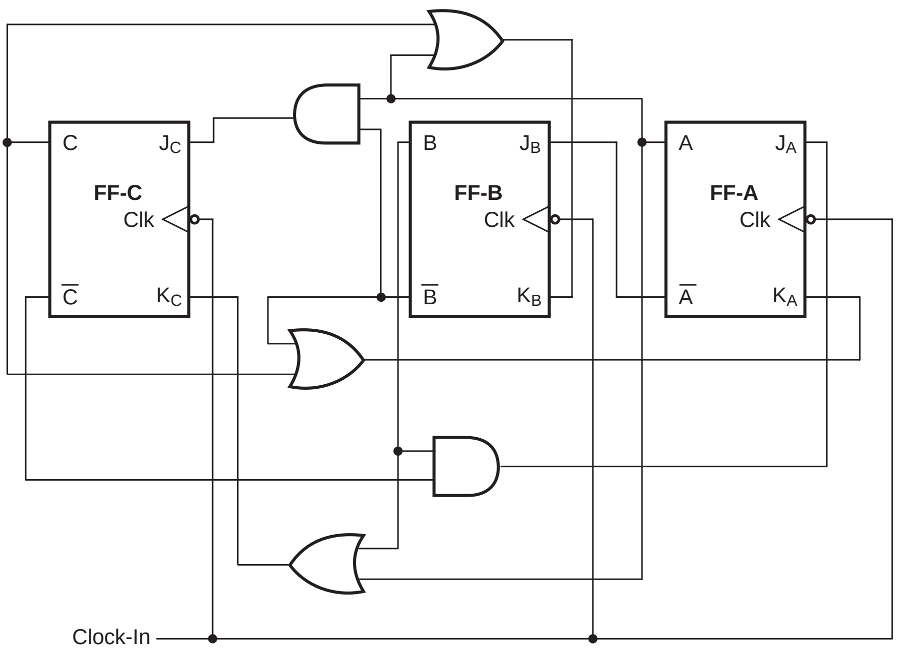


Figure 11.27 Counter with an arbitrary sequence.

The design procedure illustrated above can be used to design a synchronous counter for any given count sequence with the condition that no state occurs more than once in one complete cycle of the given count sequence as the design cannot handle a situation where a particular present state has more than one future state.

Table 11.10 Example 11.8.

Present state (Q_n)	Next state (Q_{n+1})	Inputs	
		X_1	X_2
0	0	0	0
0	1	0	1
1	0	1	X
1	1	X	1

X = don't care condition.

Example 11.8

Table 11.10 gives the excitation table of a certain flip-flop having X_1 and X_2 as its inputs. Draw the circuit excitation table of an MOD-5 synchronous counter using this flip-flop for the count sequence 000, 001, 011, 101, 110, 000, . . . If the present state is an undesired one, it should transit to 110 on application of a clock pulse. Design the counter circuit using the flip-flop whose excitation circuit is given in Table 11.10.

Solution

- The circuit excitation table is shown in Table 11.11.
- The number of flip-flops required is 3.
- X_1 (A) and X_2 (A) are the inputs of flip-flop A, which is also the LSB flip-flop.
- X_1 (B) and X_2 (B) represent the inputs to flip-flop B.
- X_1 (C) and X_2 (C) are the inputs to flip-flop C, which is also the MSB flip-flop.
- The next step is to draw Karnaugh maps, one each for different inputs to the three flip-flops.
- Figures 11.28(a) to (f) show the Karnaugh maps for X_1 (A), X_2 (A), X_1 (B), X_2 (B), X_1 (C) and X_2 (C) respectively.
- The minimized expressions are as follows:

$$X_1(A) = A \tag{11.8}$$

$$X_2(A) = A + \overline{B}.\overline{C} \tag{11.9}$$

$$X_1(B) = B \tag{11.10}$$

$$X_2(B) = A + B + C \tag{11.11}$$

$$X_1(C) = C \tag{11.12}$$

$$X_2(C) = B + C \tag{11.13}$$

- Figure 11.29 shows the circuit implementation.

Example 11.9

Design a synchronous counter that counts as 000, 010, 101, 110, 000, 010, . . . Ensure that the unused states of 001, 011, 100 and 111 go to 000 on the next clock pulse. Use J-K flip-flops. What will the counter hardware look like if the unused states are to be considered as 'don't care's.

Table 11.11 Example 11.8.

Present state			Next state			Inputs					
C	*B*	*A*	*C*	*B*	*A*	$X_1(A)$	$X_2(A)$	$X_1(B)$	$X_2(B)$	$X_1(C)$	$X_2(C)$
0	0	0	0	0	1	0	1	0	0	0	0
0	0	1	0	1	1	X	1	0	1	0	0
0	1	0	1	1	0	0	0	X	1	0	1
0	1	1	1	0	1	X	1	1	X	0	1
1	0	0	1	1	0	0	0	0	1	X	1
1	0	1	1	1	0	1	X	0	1	X	1
1	1	0	0	0	0	0	0	1	X	1	X
1	1	1	1	1	0	1	X	X	1	X	1

X = don’t care condition.

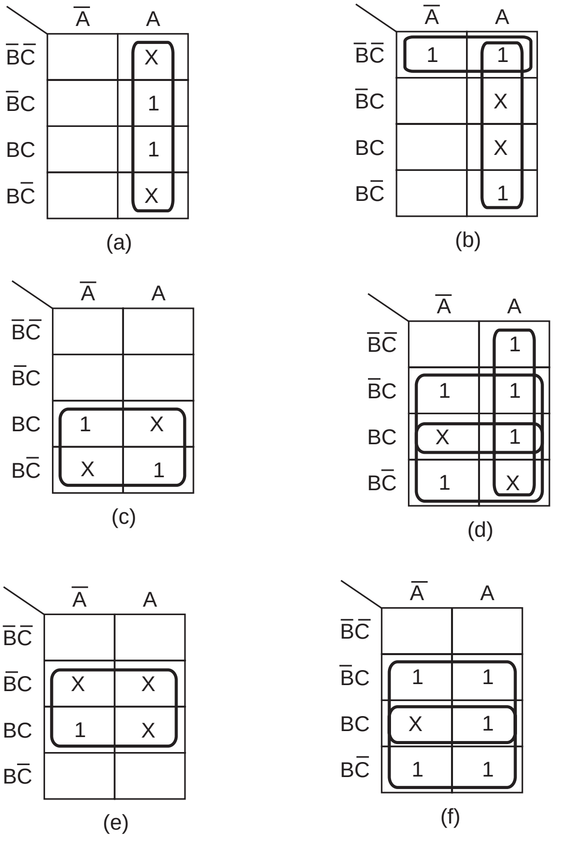


Figure 11.28 Karnaugh maps (example 11.8).

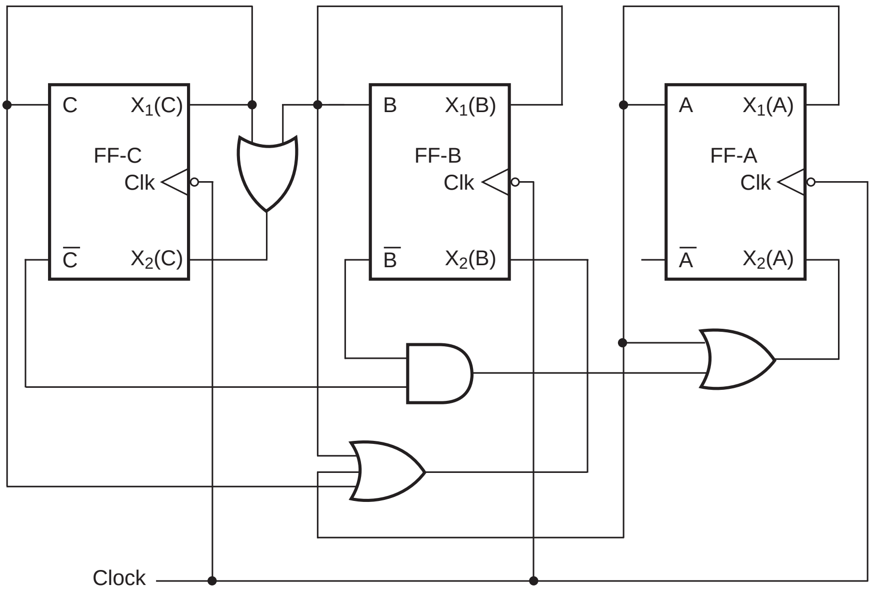


Figure 11.29 Counter circuit (example 11.8).

Table 11.12 Example 11.9.

Present state			Next state			Inputs					
C	B	A	C	B	A	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	0	0	X	1	X	0	X
0	0	1	0	0	0	X	1	0	X	0	X
0	1	0	1	0	1	1	X	X	1	1	X
0	1	1	0	0	0	X	1	X	1	0	X
1	0	0	0	0	0	0	X	0	X	X	1
1	0	1	1	1	0	X	1	1	X	X	0
1	1	0	0	0	0	0	X	X	1	X	1
1	1	1	0	0	0	X	1	X	1	X	1

Solution

- The number of flip-flops required is three.
- Table 11.12 shows the desired circuit excitation table.
- The Karnaugh maps for J_A, K_A, J_B, K_B, J_C and K_C are shown in Figs 11.30(a) to (f) respectively.
- The simplified Boolean expressions are as follows:

$$J_A = B.\overline{C} \tag{11.14}$$

$$K_A = 1 \tag{11.15}$$

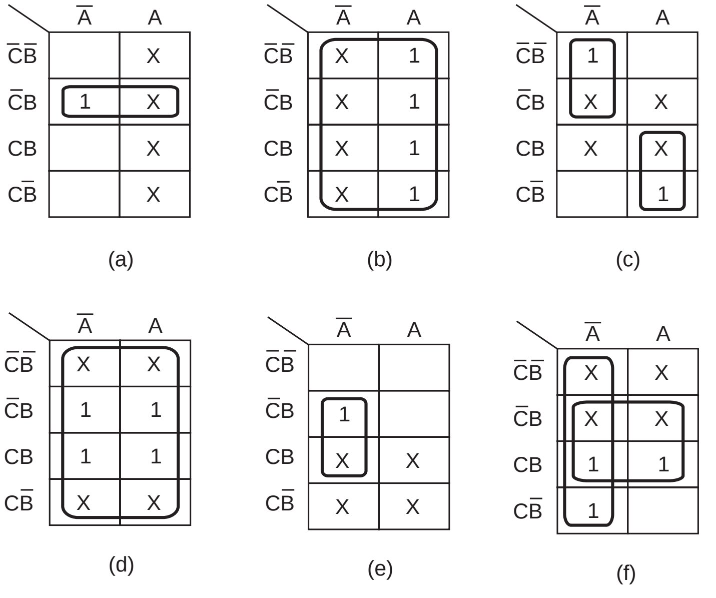


Figure 11.30 Karnaugh maps (example 11.9).

$$J_B = A.C + \overline{A}.\overline{C} \tag{11.16}$$

$$K_B = 1 \tag{11.17}$$

$$J_C = \overline{A}.B \tag{11.18}$$

$$K_C = \overline{A} + B \tag{11.19}$$

- The hardware implementation is shown in Fig. 11.31.
- In the case where the unused inputs are considered as 'don't cares', the circuit excitation table is modified to that shown in Table 11.13.
- Modified Karnaugh maps are shown in Fig. 11.32.
- The minimized Boolean expressions are derived from the Karnaugh maps of Figs 11.32(a) to (f).
- Minimized expressions for J_A, K_A, J_B, K_B, J_C and K_C respectively are as follows:

$$J_A = B.\overline{C} \tag{11.20}$$

$$K_A = 1 \tag{11.21}$$

$$J_B = 1 \tag{11.22}$$

$$K_B = 1 \tag{11.23}$$

$$J_C = B \tag{11.24}$$

$$K_C = \overline{A} \tag{11.25}$$

- Figure 11.33 shows the hardware implementation.

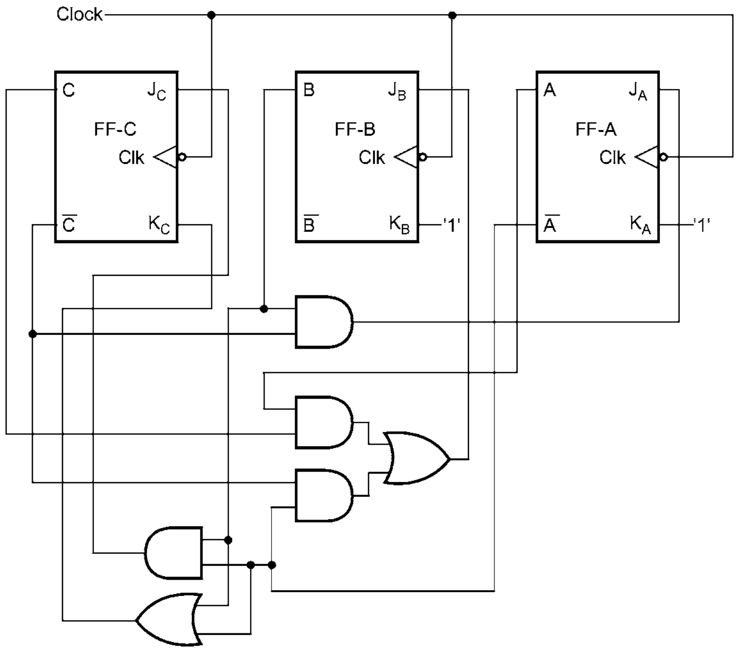


Figure 11.31 Hardware implementation of the counter circuit (example 11.9).

Table 11.13 Example 11.9.

Present state			Next state			Inputs					
C	B	A	C	B	A	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	0	0	X	1	X	0	X
0	0	1	X	X	X	X	X	X	X	X	X
0	1	0	1	0	1	1	X	X	1	1	X
0	1	1	X	X	X	X	X	X	X	X	X
1	0	0	X	X	X	X	X	X	X	X	X
1	0	1	1	1	0	X	1	1	X	X	0
1	1	0	0	0	0	0	X	X	1	X	1
1	1	1	X	X	X	X	X	X	X	X	X

11.12 Shift Register

A *shift register* is a digital device used for storage and transfer of data. The data to be stored could be the data appearing at the output of an encoding matrix before they are fed to the main digital system for processing or they might be the data present at the output of a microprocessor before they are fed

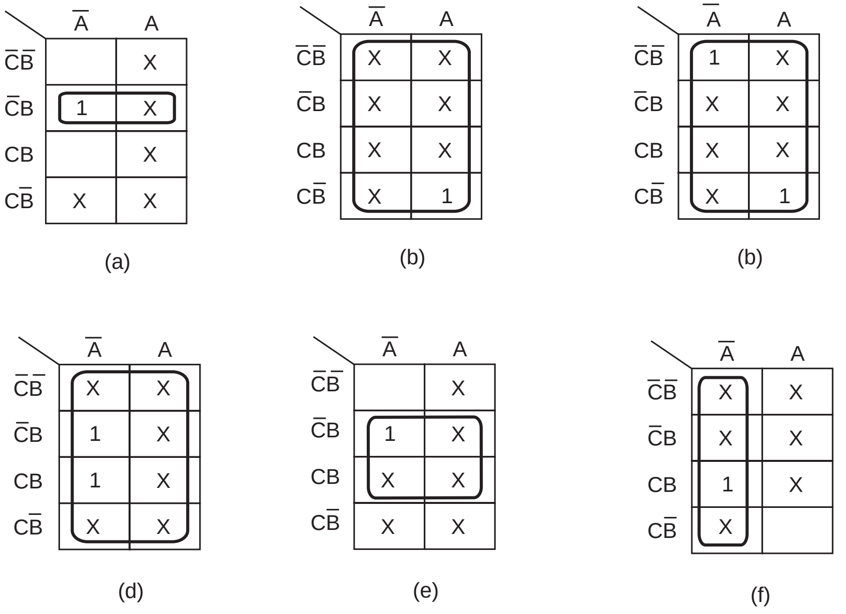


Figure 11.32 Modified Karnaugh maps (example 11.9).

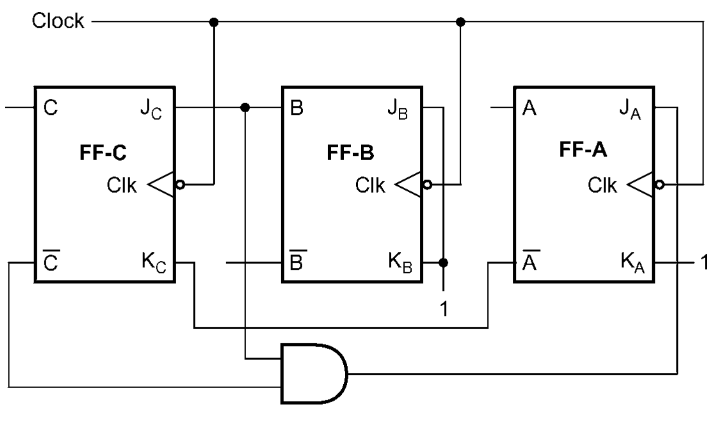


Figure 11.33 Hardware implementation of the counter circuit (example 11.9).

to the driver circuitry of the output devices. The shift register thus forms an important link between the main digital system and the input/output channels. The shift registers can also be configured to construct some special types of counter that can be used to perform a number of arithmetic operations such as subtraction, multiplication, division, complementation, etc. The basic building block in all shift registers is the flip-flop, mainly a D-type flip-flop. Although in many of the commercial shift register ICs their internal circuit diagram might indicate the use of *R-S* flip-flops, a careful examination will reveal that these *R-S* flip-flops have been wired as *D* flip-flops only.

The storage capacity of a shift register equals the total number of bits of digital data it can store, which in turn depends upon the number of flip-flops used to construct the shift register. Since each flip-flop can store one bit of data, the storage capacity of the shift register equals the number of flip-flops used. As an example, the internal architecture of an eight-bit shift register will have a cascade arrangement of eight flip-flops.

Based on the method used to load data onto and read data from shift registers, they are classified as serial-in serial-out (SISO) shift registers, serial-in parallel-out (SIPO) shift registers, parallel-in serial-out (PISO) shift registers and parallel-in parallel-out (PIPO) shift registers.

Figure 11.34 shows a circuit representation of the above-mentioned four types of shift register.

11.12.1 Serial-In Serial-Out Shift Register

Figure 11.35 shows the basic four-bit serial-in serial-out shift register implemented using *D* flip-flops. The circuit functions as follows. A reset applied to the CLEAR input of all the flip-flops resets their Q outputs to 0s. Refer to the timing waveforms of Fig. 11.36. The waveforms shown include the clock pulse train, the waveform representing the data to be loaded onto the shift register and the Q outputs of different flip-flops.

The flip-flops shown respond to the LOW-to-HIGH transition of the clock pulses as indicated by their logic symbols. During the first clock transition, the Q_A output goes from logic ‘0’ to logic ‘1’.

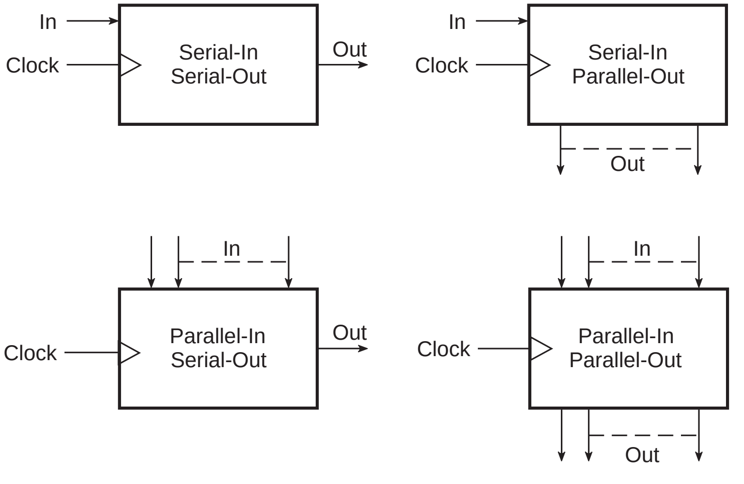


Figure 11.34 Circuit representation of shift registers.

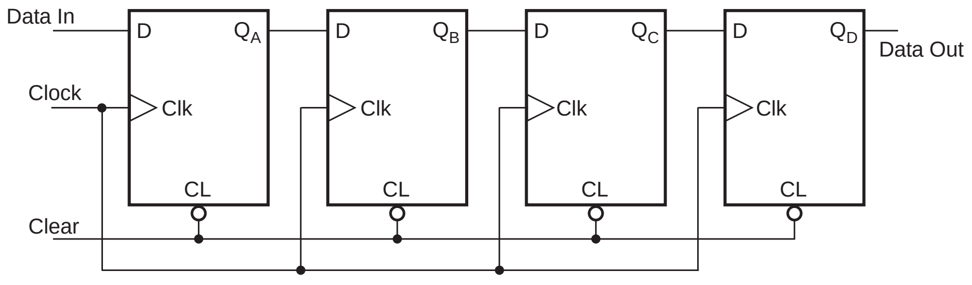


Figure 11.35 Serial-in, serial-out shift register.

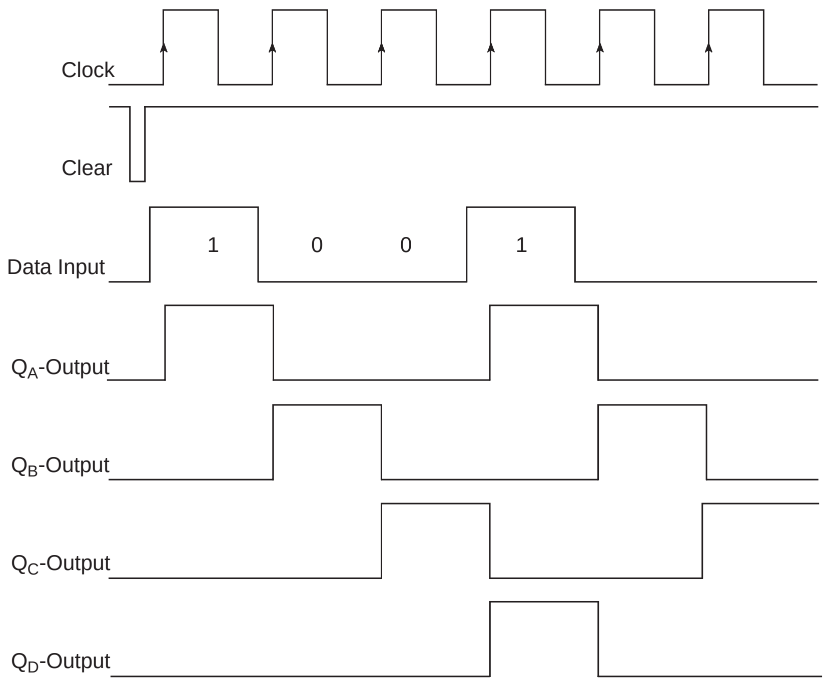


Figure 11.36 Timing waveforms for the shift register of Fig. 11.35.

The outputs of the other three flip-flops remain in the logic '0' state as their D inputs were in the logic '0' state at the time of clock transition. During the second clock transition, the Q_A output goes from logic '1' to logic '0' and the Q_B output goes from logic '0' to logic '1', again in accordance with the logic status of the D inputs at the time of relevant clock transition.

Thus, we have seen that a logic '1' that was present at the data input prior to the occurrence of the first clock transition has reached the Q_B output at the end of two clock transitions. This bit will reach the Q_D output at the end of four clock transitions. In general, in a four-bit shift register of the type

Table 11.14 Contents of four-bit serial-in serial-out shift register for the first eight clock cycles.

Clock	Q_A	Q_B	Q_C	Q_D
Initial contents	0	0	0	0
After first clock transition	1	0	0	0
After second clock transition	0	1	0	0
After third clock transition	0	0	1	0
After fourth clock transition	**1**	**0**	**0**	**1**
After fifth clock transition	0	1	0	0
After sixth clock transition	0	0	1	0
After seventh clock transition	0	0	0	1
After eighth clock transition	**0**	**0**	**0**	**0**

shown in Fig. 11.35, a data bit present at the data input terminal at the time of the nth clock transition reaches the Q_D output at the end of the $(n+4)$th clock transition. During the fifth and subsequent clock transitions, data bits continue to shift to the right, and at the end of the eighth clock transition the shift register is again reset to all 0s. Thus, in a four-bit serial-in serial-out shift register, it takes four clock cycles to load the data bits and another four cycles to read the data bits out of the register. The contents of the register for the first eight clock cycles are summarized in Table 11.14. We can see that the register is loaded with the four-bit data in four clock cycles, and also that the stored four-bit data are read out in the subsequent four clock cycles.

IC 7491 is a popular eight-bit serial-in serial-out shift register. Figure 11.37 shows its internal functional diagram, which is a cascade arrangement of eight R-S flip-flops. Owing to the inverter between the R and S inputs of the data input flip-flop, it is functionally the same as a D flip-flop. The data to be loaded into the register serially can be applied either at A or B input of the NAND gate. The other input is then kept in the logic HIGH state to enable the NAND gate. In that case, data present at A or B get complemented as they appear at the NAND output. Another inversion provided by the inverter, however, restores the original status so that for a logic '1' at the data input there is a logic '1' at the SET input of the flip-flop and a logic '0' at the RESET input of the flip-flop, and for a logic '0' at the data input there is a logic '0' at the SET input and a logic '1' at the RESET input of the flip-flop. The NAND gate provides only a gating function, and, if it is not required, the two inputs of the NAND can be shorted to have a single-line data input. The shift register responds to the LOW-to-HIGH transitions of the clock pulses.

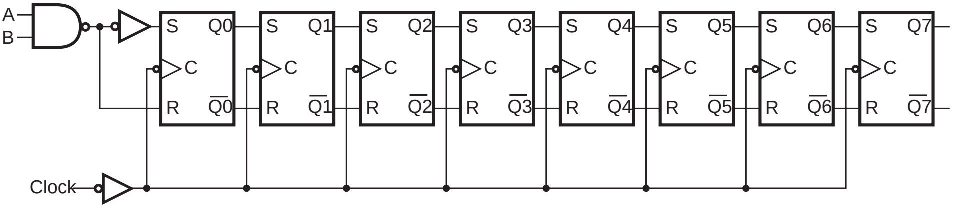


Figure 11.37 Logic diagram of IC 7491.

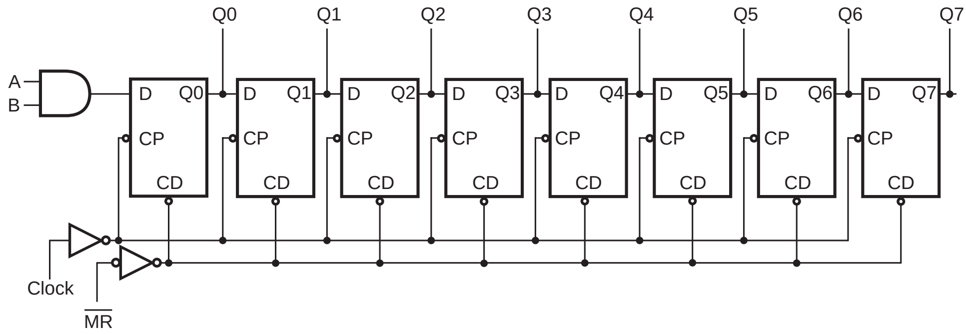


Figure 11.38 Logic diagram of IC 74164.

11.12.2 Serial-In Parallel-Out Shift Register

A serial-in parallel-out shift register is architecturally identical to a serial-in serial-out shift register except that in the case of the former all flip-flop outputs are also brought out on the IC terminals. Figure 11.38 shows the logic diagram of a typical serial-in parallel-out shift register. In fact, the logic diagram shown in Fig. 11.38 is that of IC 74164, a popular eight-bit serial-in parallel-out shift register. The gated serial inputs *A* and *B* control the incoming serial data, as a logic LOW at either of the inputs inhibits entry of new data and also resets the first flip-flop to the logic LOW level at the next clock pulse. Logic HIGH at either of the inputs enables the other input, which then determines the state of the first flip-flop.

Data at the serial inputs may be changed while the clock input is HIGH or LOW, and the register responds to LOW-to-HIGH transition of the clock. Figure 11.39 shows the relevant timing waveforms.

11.12.3 Parallel-In Serial-Out Shift Register

We will explain the operation of a parallel-in serial-out shift register with the help of the logic diagram of a practical device available in IC form. Figure 11.40 shows the logic diagram of one such shift register. The logic diagram is that of IC 74166, which is an eight-bit parallel/serial-in, serial-out shift register belonging to the TTL family of devices.

The parallel-in or serial-in modes are controlled by a SHIFT/LOAD input. When the SHIFT/LOAD input is held in the logic HIGH state, the serial data input AND gates are enabled and the circuit behaves like a serial-in serial-out shift register. When the SHIFT/LOAD input is held in the logic LOW state, parallel data input AND gates are enabled and data are loaded in parallel, in synchronism with the next clock pulse. Clocking is accomplished on the LOW-to-HIGH transition of the clock pulse via a two-input NOR gate. Holding one of the inputs of the NOR gate in the logic HIGH state inhibits the clock applied to the other input. Holding an input in the logic LOW state enables the clock to be applied to the other input. An active LOW CLEAR input overrides all the inputs, including the clock, and resets all flip-flops to the logic '0' state. The timing waveforms shown in Fig. 11.41 explain both serial-in, serial-out as well as parallel-in, serial-out operations.

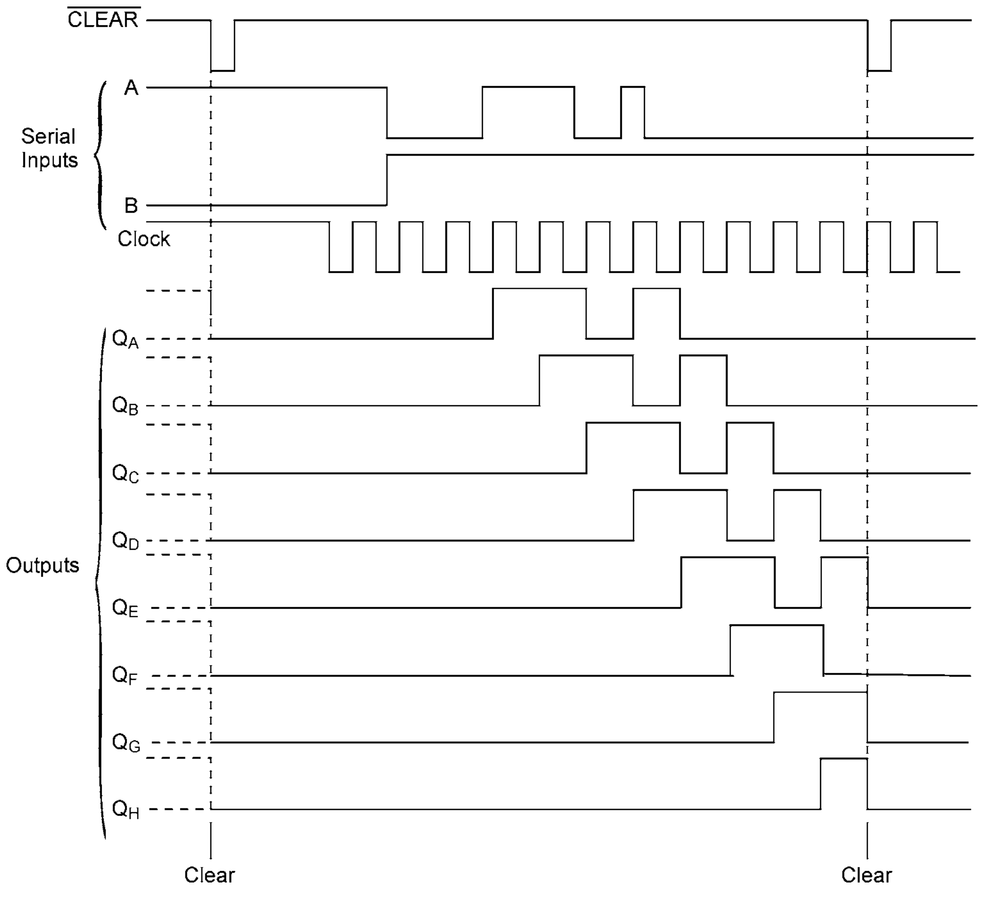


Figure 11.39 Timing waveforms of IC 74164.

11.12.4 Parallel-In Parallel-Out Shift Register

The hardware of a parallel-in parallel-out shift register is similar to that of a parallel-in serial-out shift register. If in a parallel-in serial-out shift register the outputs of different flip-flops are brought out, it becomes a parallel-in parallel-out shift register. In fact, the logic diagram of a parallel-in parallel-out shift register is similar to that of a parallel-in serial-out shift register. As an example, IC 74199 is an eight-bit parallel-in parallel-out shift register. Figure 11.42 shows its logic diagram. We can see that the logic diagram of IC 74199 is similar to that of IC 74166 mentioned in the previous section, except that in the case of the former the flip-flop outputs have been brought out on the IC terminals.

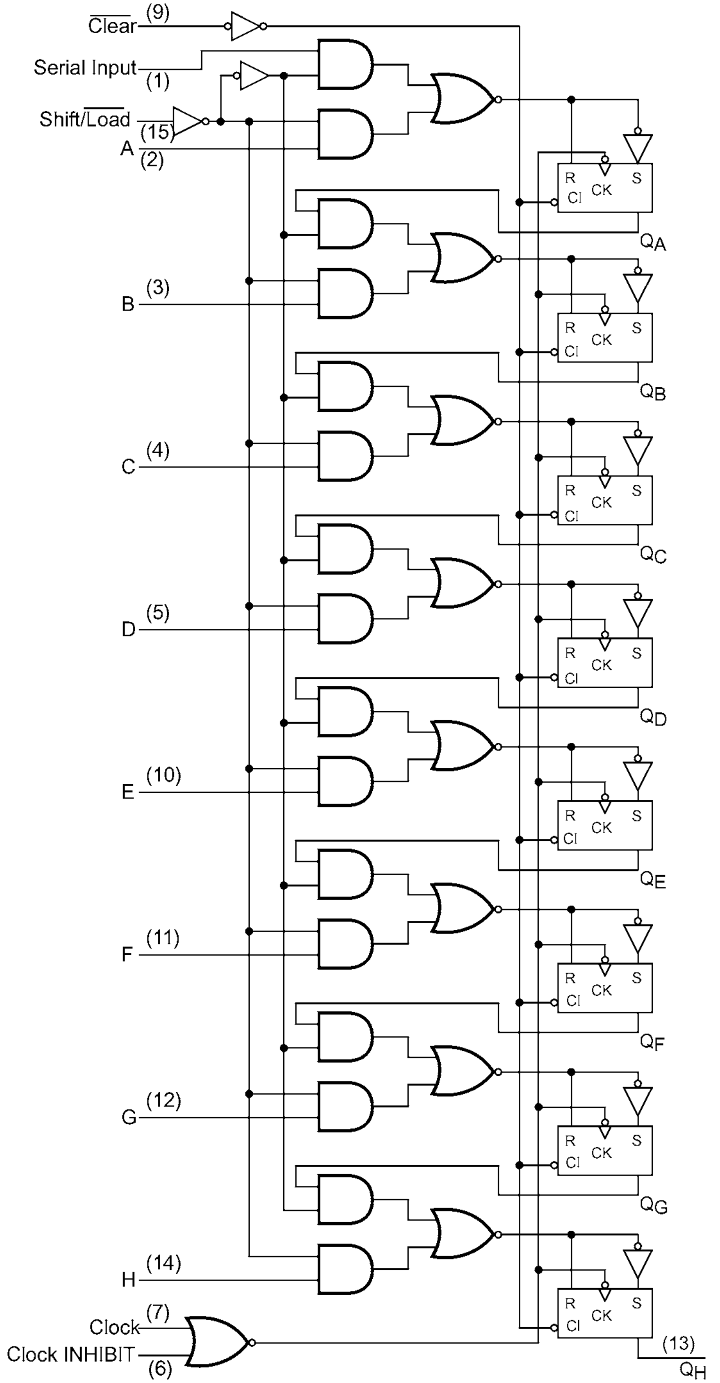


Figure 11.40 Logic diagram of 74166.

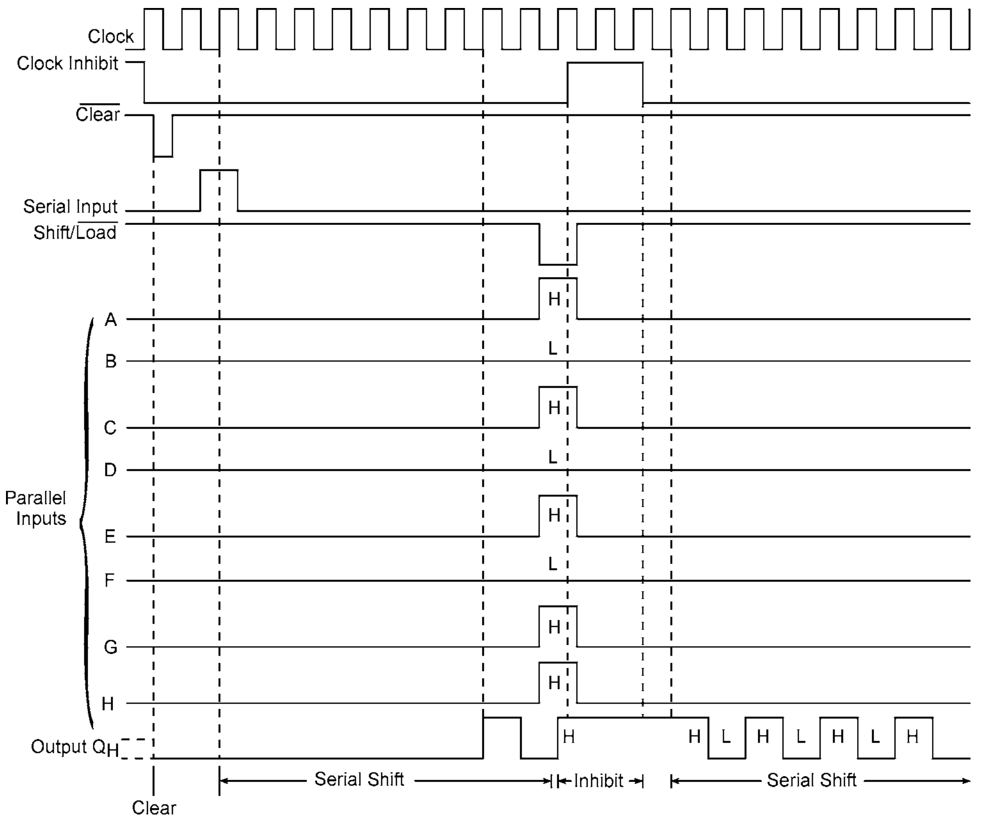


Figure 11.41 Timing waveforms of IC 74166.

11.12.5 Bidirectional Shift Register

A bidirectional shift register allows shifting of data either to the left or to the right. This is made possible with the inclusion of some gating logic having a control input. The control input allows shifting of data either to the left or to the right, depending upon its logic status.

11.12.6 Universal Shift Register

A universal shift register can be made to function as any of the four types of register discussed in previous sections. That is, it has serial/parallel data input and output capability, which means that it can function as serial-in serial-out, serial-in parallel-out, parallel-in serial out and parallel-in parallel-out shift registers.

IC 74194 is a common four-bit bidirectional universal shift register. Figure 11.43 shows the logic diagram of Ic 74194. the device offers four modes of operation, namely (a) inhibit clock, (b) shift right, (c) shift left and (d) parallel load. Clocking of the device is inhibited when both the mode control inputs S_1 and S_0 are in the logic LOW state. shift right and shift left operations are accomplished

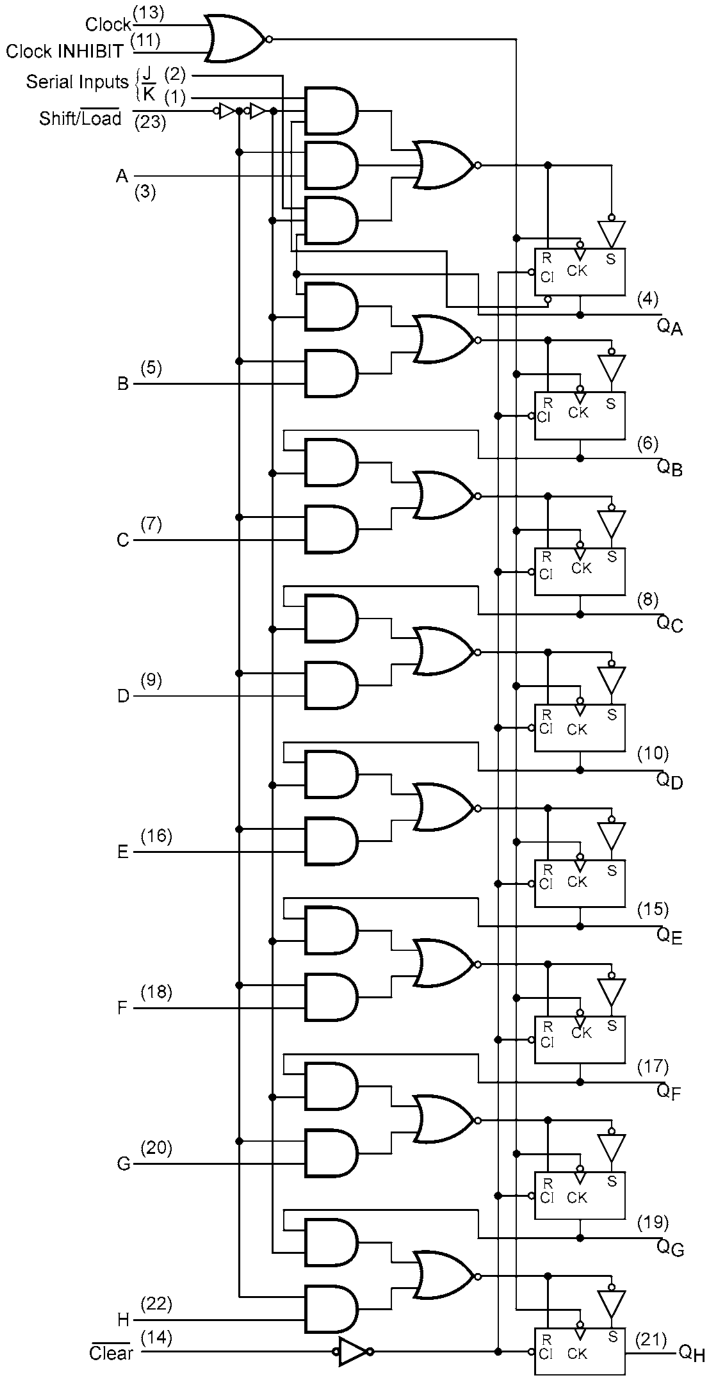


Figure 11.42 Logic diagram of IC 74199.

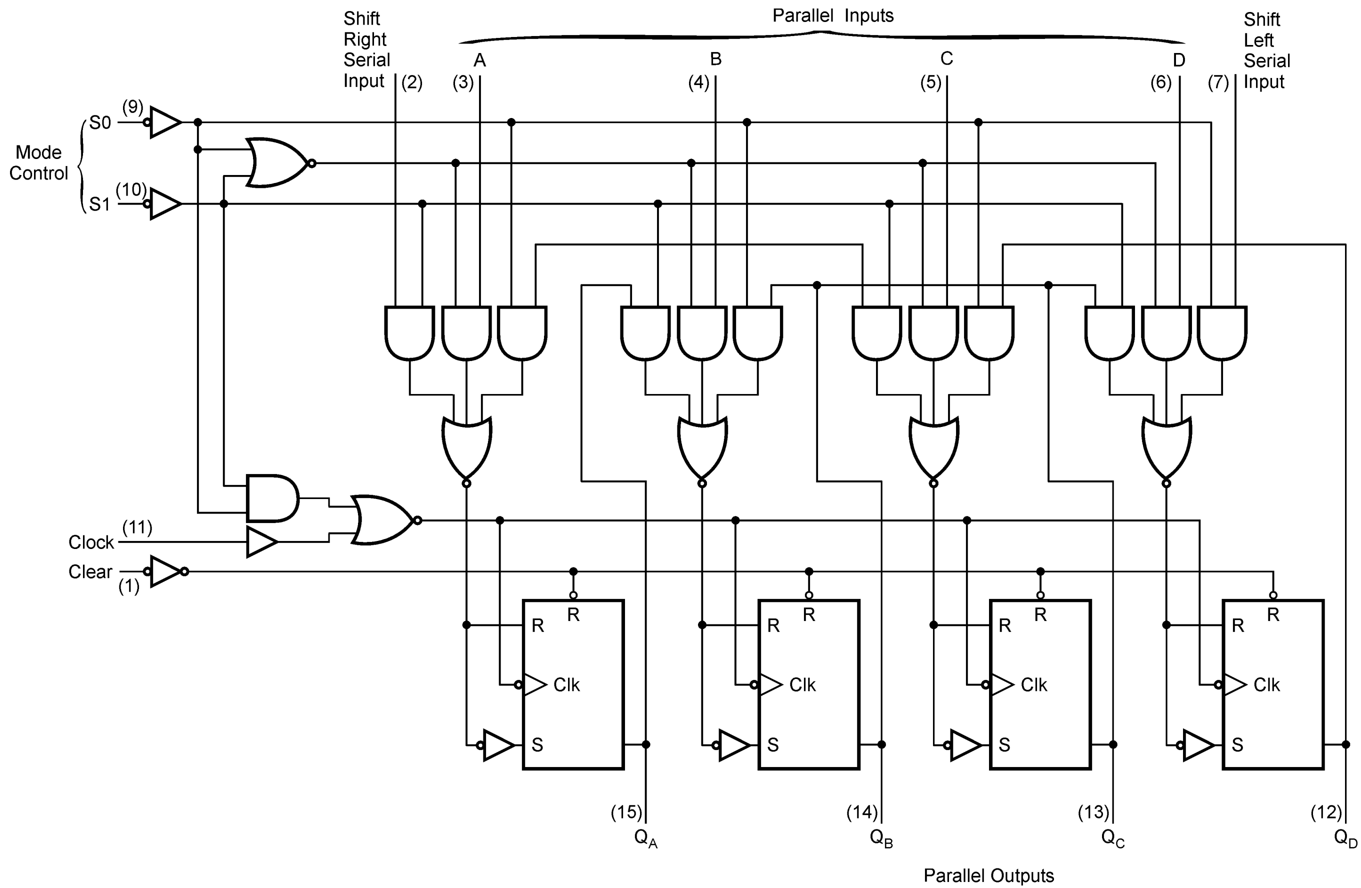


Figure 11.43 Logic diagram of IC 74194.

synchronously with LOW-to-HIGH transition of the clock with S_1 LOW and S_0 HIGH (for shift right) and S_1 HIGH and S_0 LOW (for shift left). Serial data are entered in the case of shift right and shift left operations at the corresponding data input terminals. Parallel loading is also accomplished synchronously with LOW-to-HIGH clock transitions by applying four bits of data and then driving the mode control inputs S_1 and S_0 to the logic HIGH state. Data are loaded into corresponding flip-flops and appear at the outputs with LOW-to-HIGH clock transition. Serial data flow is inhibited during parallel loading. Different modes of operation are apparent in the timing waveforms of Fig. 11.44.

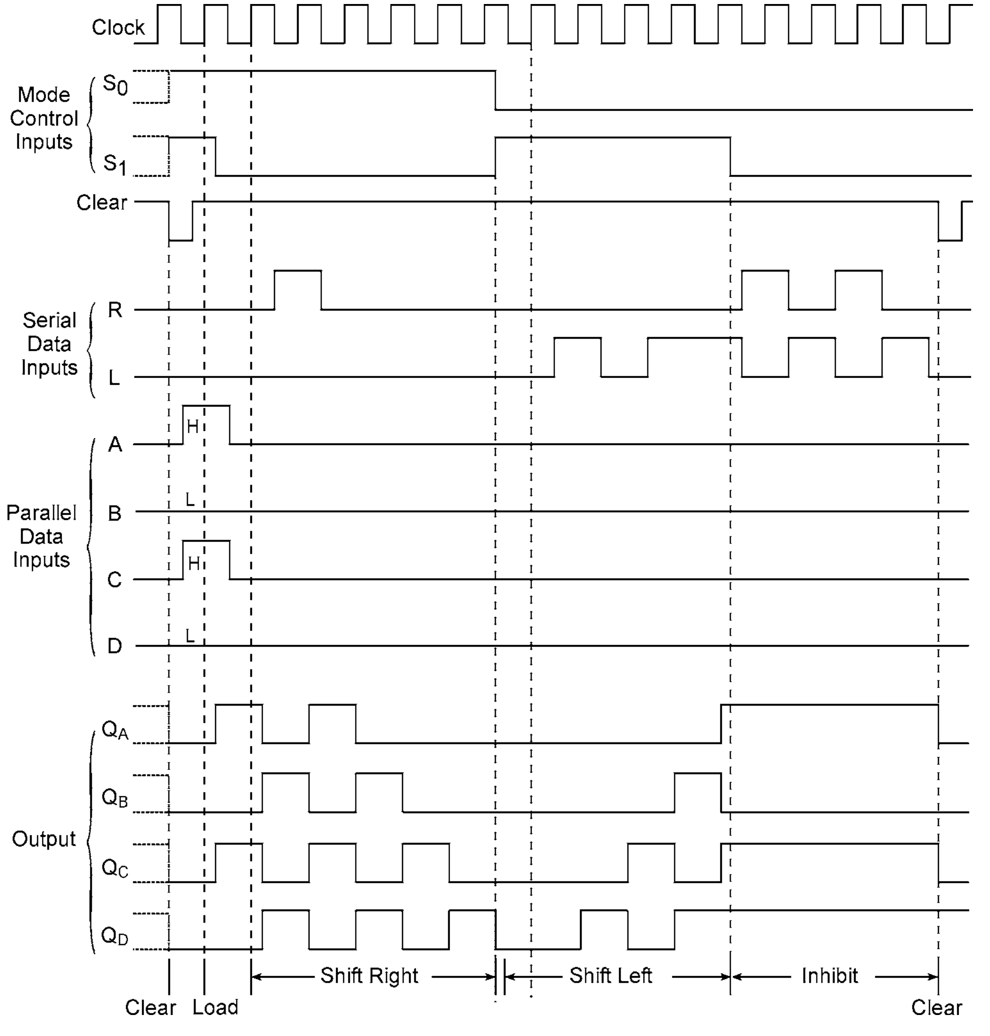


Figure 11.44 Timing waveforms of IC 74194.

11.13 Shift Register Counters

We have seen that both counters and shift registers are some kinds of cascade arrangement of flip-flops. A shift register, unlike a counter, has no specified sequence of states. However, if the serial output of the shift register is fed back to the serial input, we do get a circuit that exhibits a specified sequence of states. The resulting circuits are known as *shift register counters*. Depending upon the nature of the feedback, we have two types of shift register counter, namely the *ring counter* and the *shift counter*, also called the *Johnson counter*. These are briefly described in the following paragraphs.

11.13.1 Ring Counter

A *ring counter* is obtained from a shift register by directly feeding back the true output of the output flip-flop to the data input terminal of the input flip-flop. If *D* flip-flops are being used to construct the shift register, the ring counter, also called a circulating register, can be constructed by feeding back the *Q* output of the output flip-flop back to the *D* input of the input flip-flop. If *J-K* flip-flops are being used, the *Q* and $\overline{Q}$ outputs of the output flip-flop are respectively fed back to the *J* and *K* inputs of the input flip-flop. Figure 11.45 shows the logic diagram of a four-bit ring counter. Let us assume that flip-flop FF0 is initially set to the logic '1' state and all other flip-flops are reset to the logic '0' state. The counter output is therefore 1000. With the first clock pulse, this '1' gets shifted to the second flip-flop output and the counter output becomes 0100. Similarly, with the second and third clock pulses, the counter output will become 0010 and 0001. With the fourth clock pulse, the counter output will again become 1000. The count cycle repeats in the subsequent clock pulses. Circulating registers of this type find wide application in the control section of microprocessor-based systems where one event should follow the other. The timing waveforms for the circulating register of Figure 11.45, as shown in Fig. 11.46, further illustrate their utility as a control element in a digital system to generate control pulses that must occur one after the other sequentially.

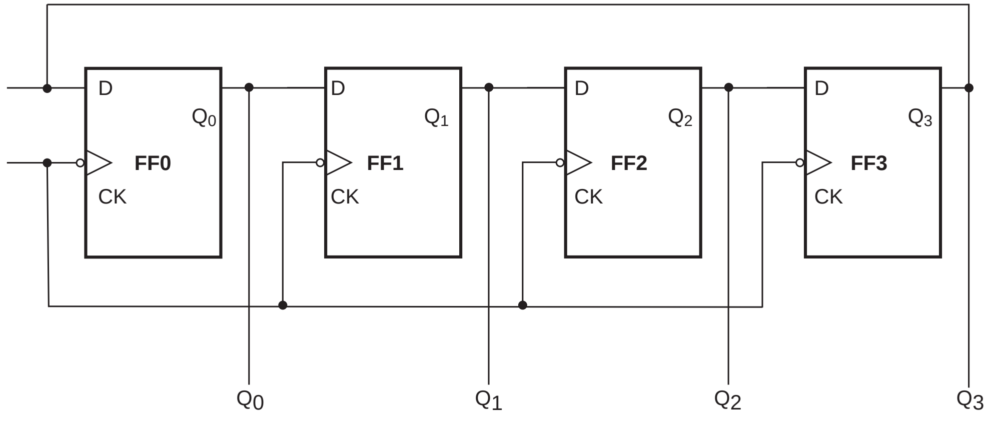


Figure 11.45 Four-bit ring counter.

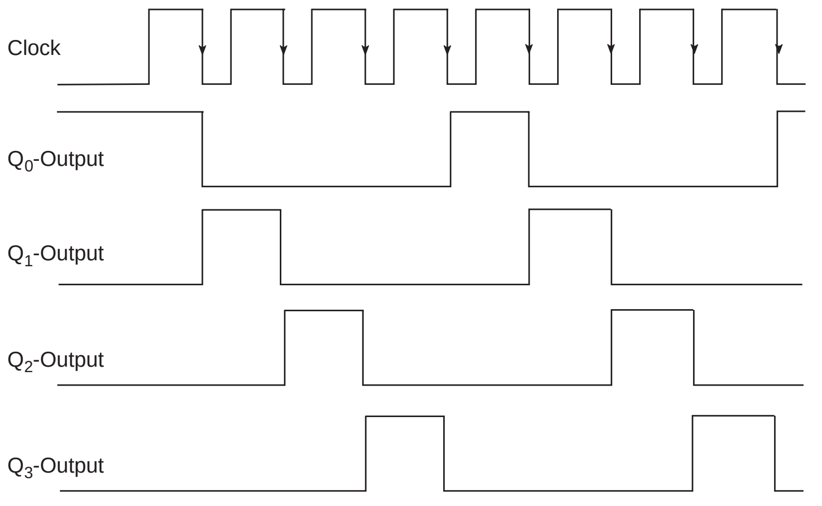


Figure 11.46 Timing waveforms of the four-bit ring counter.

11.13.2 Shift Counter

A *shift counter* on the other hand is constructed by having an inverse feedback in a shift register. For instance, if we connect the Q output of the output flip-flop back to the K input of the input flip-flop and the $\overline{Q}$ output of the output flip-flop to the J input of the input flip-flop in a serial shift register, the result is a shift counter, also called a *Johnson counter*. If the shift register employs D flip-flops, the $\overline{Q}$ output of the output flip-flop is fed back to the D input of the input flip-flop. If R-S flip-flops are used, the Q output goes to the R input and the $\overline{Q}$ output is connected to the S input. Figure 11.47 shows the logic diagram of a basic four-bit shift counter.

Let us assume that the counter is initially reset to all 0s. With the first clock cycle, the outputs will become 1000. With the second, third and fourth clock cycles, the outputs will respectively be 1100, 1110 and 1111. The fifth clock cycle will change the counter output to 0111. The sixth, seventh and eighth clock pulses successively change the outputs to 0011, 0001 and 0000. Thus, one count cycle

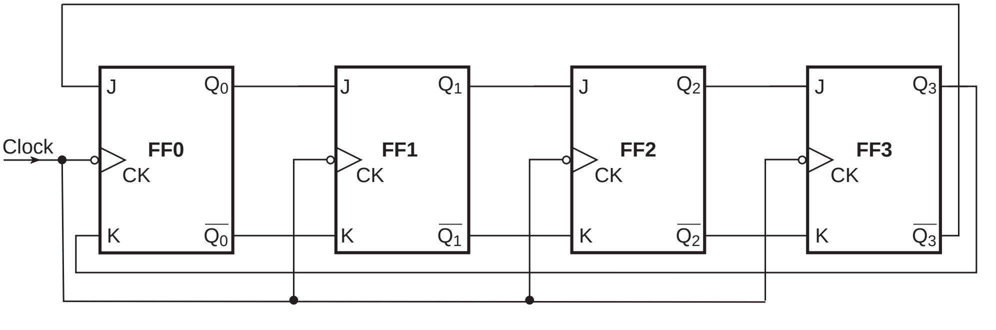


Figure 11.47 Four-bit shift counter.

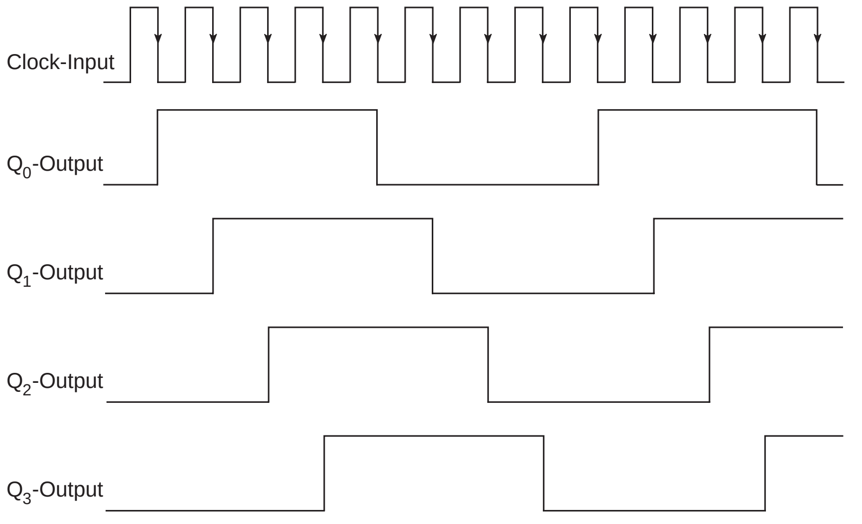


Figure 11.48 Timing waveforms of the shift counter.

is completed in eight cycles. Figure 11.48 shows the timing waveforms. Different output waveforms are identical except for the fact that they are shifted from the immediately preceding one by one clock cycle. Also, the time period of each of these waveforms is 8 times the period of the clock waveform. That is, this shift counter behaves as a divide-by-8 circuit.

In general, a shift counter comprising n flip-flops acts as a divide-by-$2n$ circuit. Shift counters can be used very conveniently to construct counters having a modulus other than the integral power of 2.

Example 11.10

Refer to Fig. 11.49, which shows an application circuit of eight-bit serial-in serial-out shift register type IC 7491 along with the waveform applied at the shorted A and B inputs:

(a) *What will be the data bit present at the output at the end of the eleventh LOW-to-HIGH transition of the clock waveform?*
(b) *If there is a logic '1' at the end of the nth LOW-to-HIGH clock transition at the Q_3 output, what will the Q_5 output at the end of the $(n+2)$th transition be?*

Solution

(a) At the end of the eighth LOW-to-HIGH clock transition, the data bits loaded into the register will be 10110010, with the '0' on the extreme right appearing at the Q_7 output (refer to the logic diagram of IC 7491 shown in Fig. 11.37). The ninth clock transition will shift this '0' out of the register, and the next adjacent bit (that is, '1') will take its place on the Q_7 output. Each subsequent clock pulse will shift the bits one step towards the right, with the result that at the end of the eleventh clock transition the Q_7 output will be a logic '0'.
(b) It will be a logic '1' only. The Q_3 output will be shifted two bit positions to the right by two clock transitions.

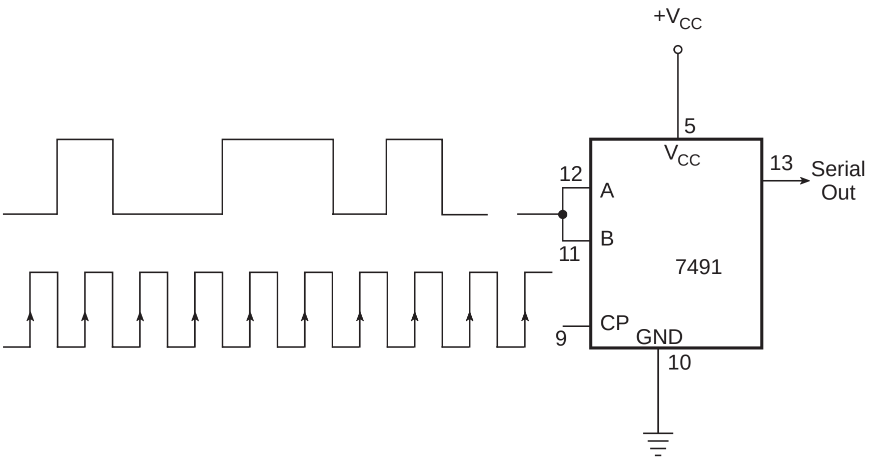


Figure 11.49 Example 11.10.

Example 11.11

Determine the number of flip-flops required to construct (a) a MOD-10 ring counter and (b) a MOD-10 Johnson counter. Also, write the count sequence in the two cases.

Solution

(a) The modulus of a ring counter is the same as the number of bits (or flip-flops). Therefore, the number of flip-flops required = 10. The count sequence is 1000000000, 0100000000, 0010000000, 0001000000, 0000100000, 0000010000, 0000001000, 0000000100, 0000000010, 0000000001 and back to 1000000000.

(b) The modulus of a Johnson counter is twice the number of flip-flops. Therefore, the number of flip-flops = 5. The count sequence is 00000, 10000, 11000, 11100, 11110, 11111, 01111, 00111, 00011, 00001 and back to 00000.

Example 11.12

Refer to the logic circuit of Fig. 11.50. Determine the modulus of this counter and write its counting sequence.

Solution

The LSB of the five-bit ring counter feeds the clock input of the *J-K* flip-flop that has been wired as a toggle flip-flop. The ring counter has a modulus of 5, and the *J-K* flip-flop works like a divide-by-2 circuit. The modulus of the counter circuit obtained by the cascade arrangement of the two is therefore 10. The counting sequence of this arrangement is given in Table 11.15.

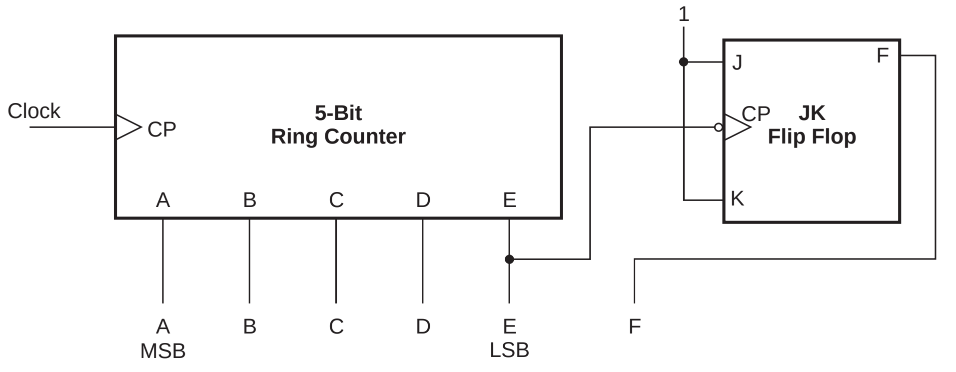


Figure 11.50 Example 11.12.

Table 11.15 Example 11.11.

Clock pulse	Outputs					
	A	*B*	*C*	*D*	*E*	*F*
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	1	0	0	0	0	1
7	0	1	0	0	0	1
8	0	0	1	0	0	1
9	0	0	0	1	0	1
10	0	0	0	0	1	1
11	1	0	0	0	0	0

It is very simple to write the count sequence. Firstly, we write the first 10 states of the ring counter output (designated by A, B, C, D and E). The logic status of F can be written by examining the logic status of E. F toggles whenever E undergoes '1' to '0' transition.

Example 11.13

Refer to the logic circuit arrangement of Fig. 11.51 built around an eight-bit serial-in/parallel-out shift register, type number 74164. A and B are the data inputs. The serial data feeding the register are obtained by an ANDing operation of A and B inputs inside the IC. $\overline{MR}$ is an active LOW master reset. Write the logic status of register outputs for the first eight clock pulses. Q_0 represents the first flip-flop in this serial shift register.

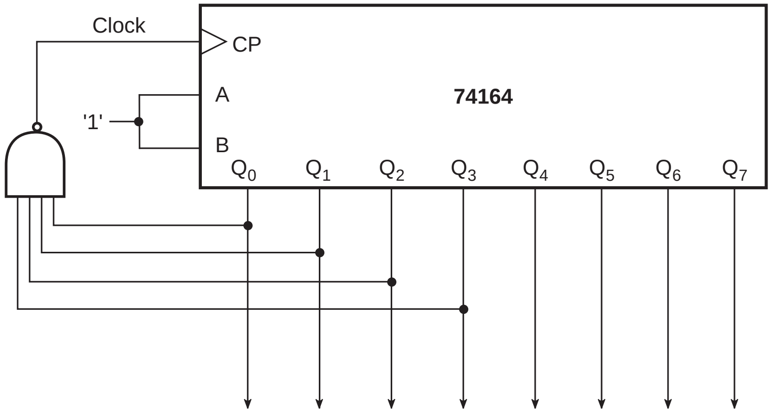


Figure 11.51 Example 11.13.

Solution

Initially, all outputs are in the logic '0' state. Since $A = B = 1$, the serial input to the shift register is a logic '1'. The $\overline{\text{MR}}$ input is initially inactive. For the first three clock pulses, the output status is 10000000, 11000000 and 11100000. With the fourth clock pulse, the output tends to go to 11110000, but it cannot be stable state as the NAND output goes from '1' to '0'. This resets the register to 00000000. Thus, the register transits from 11100000 to 00000000. With the fifth, sixth and seventh clock pulses, the circuit goes through 10000000, 11000000 and 11100000. The eight clock pulse again resets it to 00000000.

11.14 IEEE/ANSI Symbology for Registers and Counters

We introduced IEEE/ANSI symbology for digital integrated circuits as contained in IEEE/ANSI Standard 91-1984 in Section 4.22 of Chapter 4 on logic gates and related devices. A brief description of salient features of this symbology and its particular significance to sequential logic devices such as flip-flops, counters, registers, etc., was given, highlighting the use of dependency notation to provide almost complete functional information of the device. In this section, we will illustrate IEEE/ANSI symbology for counters and registers with the help of IEEE/ANSI symbols of some popular devices.

11.14.1 Counters

As an illustration, we will consider IEEE/ANSI symbols of a decade counter, type number 7490, and a presettable four-bit binary UP/DOWN counter, type number 74193. The IEEE/ANSI notation for IC 7490 and IC 74193 is shown in Figs 11.52(a) and (b) respectively.

The upper portion of the notation represents the common control block that affects all flip-flops constituting the counter. The lower portion represents individual flip-flops. Before we interpret different labels and inputs/outputs for the two counter ICs, we should know the following:

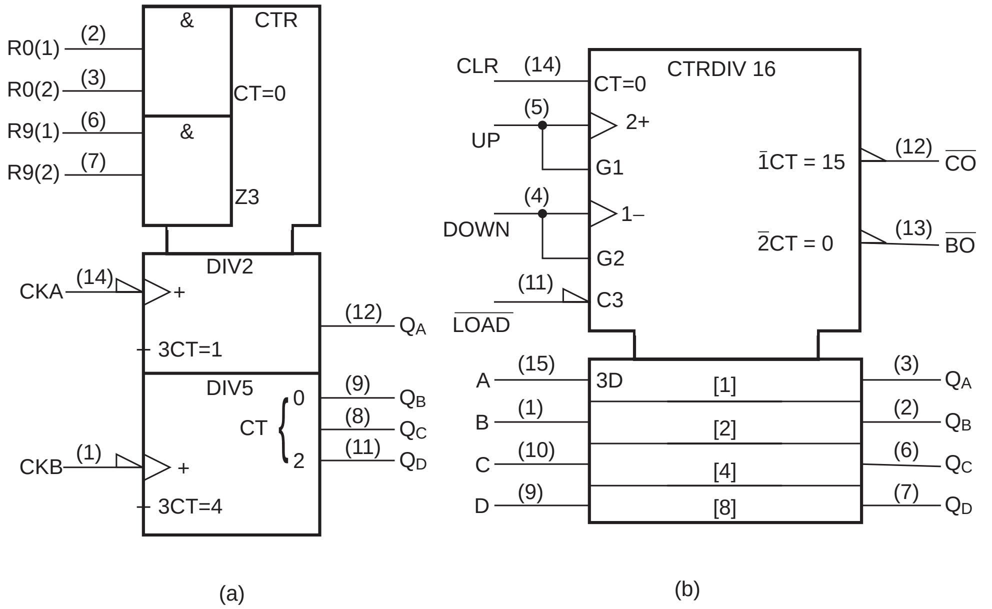


Figure 11.52 IEEE/ANSI notation for (a) IC 7490 and (b) IC 74193.

1. Letter 'C' represents control dependency. Use of the letter 'C' in the label of a certain input means that that particular input controls the entry of data into a storage element such as a flip-flop. The storage element or elements that are controlled by this input are indicated by a digit used as a suffix to the letter 'C'. The same digit appears as a prefix in the labels of all those storage elements that are controlled by this input.
2. Letter 'G' represents an AND dependency. The use of the letter 'G' followed by a digit in the label of an input means that this input is internally ANDed with another input or output and that the input or output will have the same digit as a prefix in its label.
3. Plus (+) and minus (−) signs in the labels indicate the count direction, with the former implying an UP count sequence and the latter implying a DOWN count sequence. These signs are used with clock inputs.

We will now interpret different inputs and outputs for the two counters. We will begin with IC 7490. Reset inputs R_0 (1) and R_0 (2) have an AND dependency, and when both of them are driven to the logic HIGH state the counter is reset to all 0s. Reset inputs R_9 (1) and R_9 (2) also have an AND dependency when both of them are driven to the logic HIGH state, the divide-by-2 portion of the counter is reset to count '1' (which is also the logic '1' state for the flip-flop true output) and the divide-by-5 portion of the counter is reset to count '4' (which is the 100 state for the counter outputs). If the two portions were used in cascade, the counter output would become 1001, which would mean that the counter is reset to count '9'. Clock A (CKA) and clock B (CKB) inputs allow the two portions of the counter to count in the upward sequence as indicated by the (+) sign.

We will now look at the IEEE/ANSI symbol of the other counter, that is, the counter IC type number 74193. Label CTR DIV16 means that IC 74193 is a divide-by-16 counter. Label CT = 0 with master reset (*MR*) input implies that the counter is reset to all 0s when the *MR* input is in the logic HIGH state. Label C3 with parallel load (*PL*) input means that the data on parallel load inputs P_0, P_1, P_2 and P_3 are loaded onto the corresponding flip-flops when the *PL* input is in the logic LOW state. We can see the prefix 3 in the labels of the flip-flops. The *CPU* input has an AND dependency with the *TCU* output and *CPD* input. In the case of the former, the *TCU* output goes to the logic LOW state when the *CPU* is LOW and the count reaches '15'. In the case of the latter, the *CPU* input should be in the logic HIGH state in order to allow the *CPD* to perform the count DOWN function. Similarly, the *CPD* input has an AND dependency with the *TCD* output and *CPU* input. In the case of the former, the *TCD* output goes to the logic LOW state when the *CPD* is LOW and the count reaches '0'. In the case of the latter, the *CPD* input should be in the logic HIGH state in order to allow the *CPU* to perform the count UP function.

11.14.2 Registers

As an illustration, we will consider IEEE/ANSI symbols of a serial-in serial-out shift register, type number 7491, and a serial-in parallel-out shift register, type number 74164. Figures 11.53(a) and (b) show the IEEE/ANSI notations for IC 7491 and IC 74164 respectively.

We will begin with shift register type number 7491. Label SRG8 stands for eight-bit shift register. Label C1/→ with the clock input means that the relevant clock transition performs two functions. Firstly, it loads data onto the data input as indicated by prefix '1' with the *D* input. Secondly, it performs a right shift operation. The *A* and *B* inputs have an AND dependency. When data are entered through either of the two inputs, the other input must be held in the logic HIGH state to allow the data bit to be loaded onto the data input terminal.

We will now consider shift register type number 74164. Label 'R' stands for reset operation. Whenever the *MR* input is driven to the logic LOW state, the shift register is reset to all 0s. The rest of the notations have already been explained in the case of register type number 7491.

11.15 Application-Relevant Information

Table 11.16 lists the commonly used IC counters and registers belonging to the TTL, CMOS and ECL logic families. Application-relevant information on more popular type numbers is given in the companion website. The information includes the pin configuration diagram, functional table and timing waveforms in some cases.

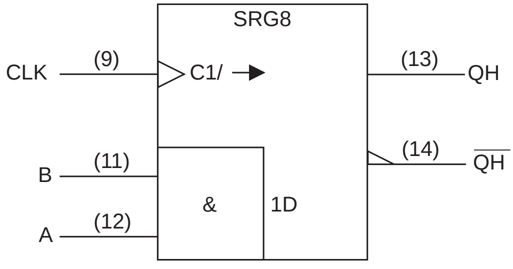
Review Questions

1. Differentiate between:

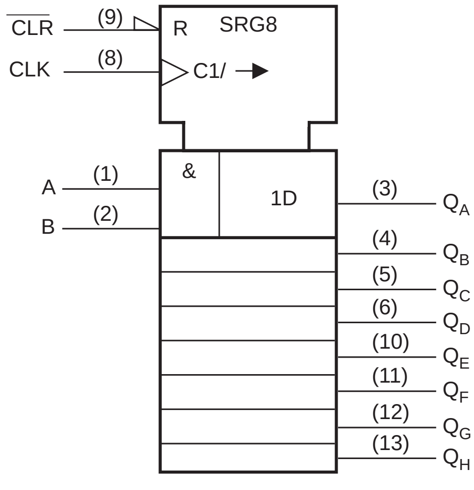
   (a) asynchronous and synchronous counters;
   (b) UP, DOWN and UP/DOWN counters;
   (c) presettable and clearable counters;
   (d) BCD and decade counters.

2. Indicate the difference between the counting sequences of:

   (a) a four-bit binary UP counter and a four-bit binary DOWN counter;
   (b) a four-bit ring counter and a four-bit Johnson counter.



(a)



(b)

Figure 11.53 IEEE/ANSI notation for (a) IC 7491 and (b) IC 74164.

3. Briefly describe:

   (a) how the architecture of an asynchronous UP counter differs from that of a DOWN counter;
   (b) how the architecture of a ring counter differs from that of a shift counter.

4. Briefly explain why the maximum usable clock frequency of a ripple counter decreases as more flip-flops are added to the counter to increase its MOD-number.
5. Why is the maximum usable clock frequency in the case of a synchronous counter independent of the size of counter?
6. How can presettable counters be used to construct counters with variable modulus?

7. Indicate the type of shift register:

   (a) into which a complete binary number can be loaded in one operation and then shifted out one bit at a time;
   (b) into which data can be entered only one bit at a time but have all data bits available as outputs;
   (c) in which we have access to only the leftmost or rightmost flip-flop.

Table 11.16 Commonly used IC counters and registers belonging to the TTL, CMOS and ECL logic families.

Type number	Function	Logic family
7490	Decade counter	TTL
7491	Eight-bit shift register (serial-in/serial-out)	TTL
7493	Four-bit binary counter	TTL
74160	BCD decade counter with asynchronous CLEAR	TTL
74161	Four-bit binary counter with asynchronous CLEAR	TTL
74162	BCD decade counter with synchronous CLEAR	TTL
74163	Four-bit binary counter with synchronous CLEAR	TTL
74164	Eight-bit shift register (serial-in/parallel-out)	TTL
74165	Eight-bit shift register (parallel-in/serial-out)	
74166	Eight-bit shift register (parallel-in/serial-out)	TTL
74178	Four-bit parallel access shift register	TTL
74190	Presettable BCD decade UP/DOWN counter	TTL
74191	Presettable four-bit binary UP/DOWN counter	TTL
74192	Presettable BCD decade UP/DOWN counter	TTL
74193	Presettable four-bit binary UP/DOWN counter	TTL
74194	Four-bit right/left universal shift register	TTL
74198	Eight-bit universal shift register (parallel-in/parallel-out bidirectional)	TTL
74199	Eight-bit universal shift register (parallel-in/parallel-out bidirectional)	TTL
74290	Decade counter	TTL
74293	Four-bit binary counter	TTL
74390	Dual decade counter	TTL
74393	Dual four-bit binary counter	TTL
4014 B	Eight-bit static shift register (synchronous parallel or serial-in/serial-out)	CMOS
4015 B	Dual four-bit static shift register (serial-in/parallel-out)	CMOS
4017 B	Five-stage Johnson counter	CMOS
4021 B	Eght-bit static shift register (asynchronous parallel-in or synchronous serial-in/serial-out)	CMOS
4029 B	Synchronous presettable four-bit UP/DOWN counter	CMOS
4035 B	Four-bit universal shift register	CMOS
40160 B	Decade counter with asynchronous CLEAR	CMOS
40161 B	Binary counter with asynchronous CLEAR	CMOS
40162 B	Decade counter	CMOS
40163 B	Binary Counter	CMOS
40192 B	Presettable BCD UP/DOWN counter	CMOS
40193 B	Presettable Binary UP/DOWN counter	CMOS
4510 B	Presettable UP/DOWN BCD counter	CMOS

Table 11.16 (*continued*).

Type number	Function	Logic family
4518 B	Dual four-bit decade counter	CMOS
4520B	Dual four-bit binary counter	CMOS
4522 B	Four-bit BCD programmable divide-by-N counter	CMOS
4722 B	Programmable counter/timer	CMOS
4731 B	Quad 64-bit static shift register	CMOS
MC 10136	Universal hexadecimal counter	ECL
MC 10137	Universal decade counter	ECL
MC 10141	Four-bit universal shift register	ECL
MC 10154	Binary counter (four-bit)	ECL
MC 10178	Four-bit binary counter	ECL

8. What do you understand when the PRESET, CLEAR, UP/DOWN, master reset and parallel load functions of a counter are designated as $\overline{PR}$, CLR, $U/\overline{D}$, $\overline{MR}$ and PL respectively?
9. What are counters with arbitrary count sequences? Briefly describe the procedure for designing a counter with a given arbitrary count sequence.
10. Give at least one IC type number for:

    (a) a four-bit binary ripple counter;
    (b) a four-bit synchronous counter;
    (c) an eight-bit serial-in serial-out shift register;
    (d) a bidirectional universal shift register.

Problems

1. For the multistage counter arrangement of Fig.11.54, determine the frequency of the output signal.
125 Hz

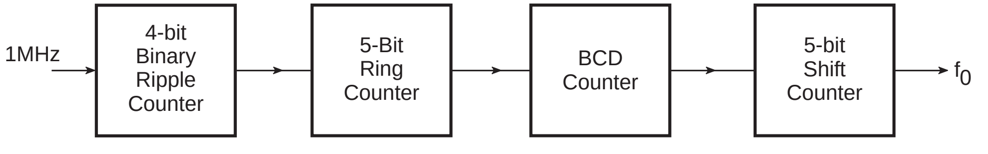


Figure 11.54 Problem 1.

2. A four-bit binary UP counter is initially in the 0000 state. Then the clock pulses are applied. Some time later the clock pulses are removed, and at that the counter is observed to be in the 0011 state. What is the minimum number of clock pulses that could possibly have occurred?
3

3. An eight-bit binary ripple UP counter with a modulus of 256 is holding the count 01111111. What will be the count after 135 clock pulses be?
00000110

4. Three four-bit BCD decade counters are connected in cascade. The MSB output of the first counter is fed to the clock input of the second counter, and the MSB output of the second counter is fed to the clock input of the third counter. If the counters are negatively edge triggered and the input clock frequency is 256 kHz, what is the frequency of the waveform available at the MSB of the third counter?

256 Hz

5. The flip-flops used in a four-bit binary ripple counter have a HIGH-to-LOW and LOW-to-HIGH propagation delay of 25 and 10 ns respectively. Determine the maximum usable clock frequency of this counter.

10 MHz

6. Refer to the counter schematic shown in Fig. 11.55. Determine the count sequence of this counter.

000, 001, 010, 011, 100, 101, 110, 000, . . .

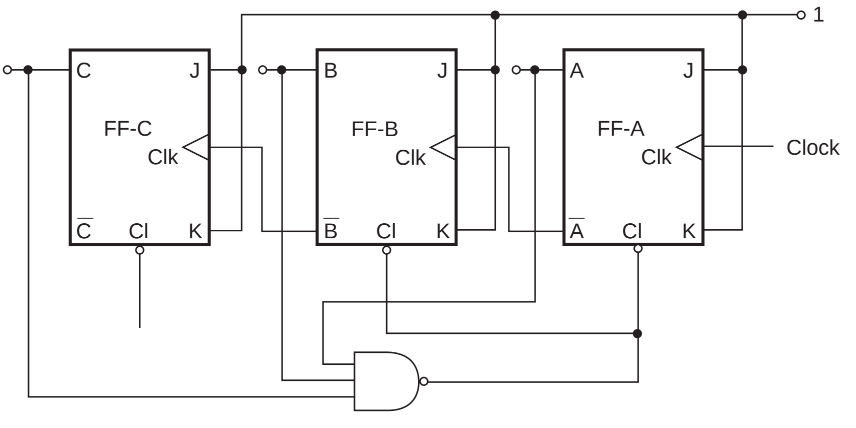


Figure 11.55 Problem 6.

7. Refer to the counter arrangement of Fig. 11.56. Determine the modulus of the counter and also the frequency of the *B* output and the duty cycle of the *C* output if the clock frequency is 600 kHz.

3; 200 kHz; 0 %

8. A four-bit ring counter and a four-bit Johnson counter are in turn clocked by a 10 MHz clock signal. Determine the frequency and duty cycle of the output of the output flip-flop in the two cases.

Ring counter: 2.5 MHz, 25 %; Johnson counter: 1.25 MHz, 50 %

9. A 100-stage serial-in/serial-out shift register is clocked at 100 kHz. How long will the data be delayed in passing through this register?

1 ms

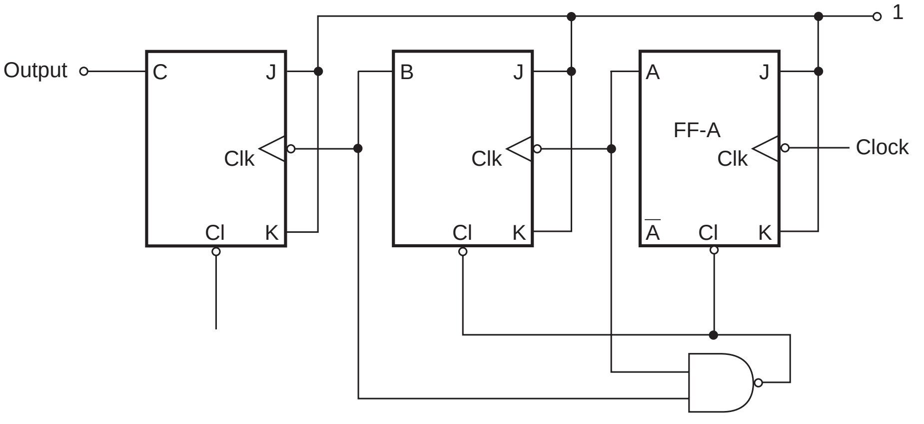


Figure 11.56 Problem 7.

10. Refer to the three-bit counter arrangement of Fig. 11.57. Determine its count sequence and also determine whether the counter is self-starting. (A counter is self-starting if it automatically goes to one of the desired states with subsequent clock pulse in case it lands itself accidentally into any of the undesired states.)

000, 001, 010, 011, 100, 000, . . . ; not self starting

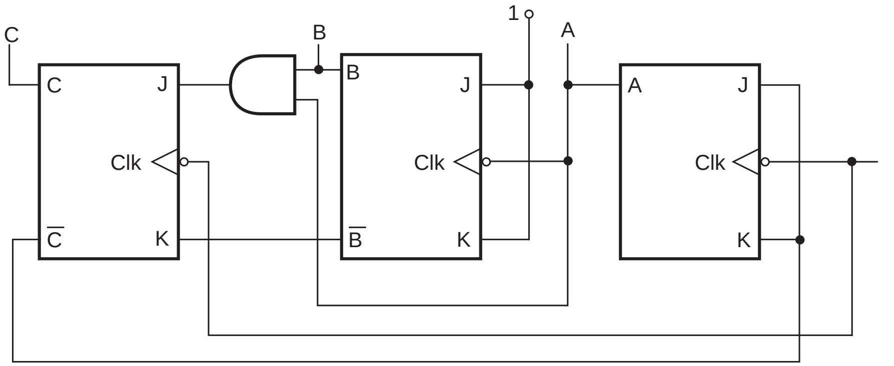


Figure 11.57 Problem 10.

Further Reading

1. Langholz, G., Mott, J. L. and Kandel, A (1998) *Foundations of Digital Logic Design*, World Scientific Publ. Co. Inc., NJ, USA.
2. Cook, N. P. (2003) *Practical Digital Electronics*, Prentice-Hall, NJ, USA.

3. Floyd, T. L. (2005) *Digital Fundamentals*, Prentice-Hall Inc., USA.
4. Tokheim, R. L. (1994) *Schaum's Outline Series of Digital Principles*, McGraw-Hill Companies Inc., USA.
5. Tocci, R. J. (2006) *Digital Systems – Principles and Applications*, Prentice-Hall Inc., NJ, USA.
6. Malvino, A. P. and Leach, D. P. (1994) *Digital Principles and Applications*, McGraw-Hill Book Company, USA.

12

Data Conversion Circuits – D/A and A/D Converters

Digital-to-analogue (D/A) and analogue-to-digital (A/D) converters constitute an essential link when digital devices interface with analogue devices, and vice versa. They are important building blocks of any digital system, including both communication and noncommunication systems, besides having other applications. A D/A converter is important not only because it is needed at the output of most digital systems, where it converts a digital signal into an analogue voltage or current so that it can be fed to a chart recorder, for instance, for measurement purposes, or a servo motor in a control application; it is also important because it forms an indispensable part of the majority of A/D converter types. An A/D converter, too, has numerous applications. When it comes to transmitting analogue data, it forms an essential interface with a digital communication system where the analogue signal to be transmitted is digitized at the sending end with an A/D converter. It is invariably used in all digital read-out test and measuring equipment. Whether it is a digital multimeter or a digital storage oscilloscope or even a pH meter, an A/D converter is an important and essential component of all of them. In this chapter, we will discuss the operational fundamentals, the major performance specifications, along with their significance, and different types and applications of digital-to-analogue and analogue-to-digital converters, in addition to application-relevant information of some of the popular devices. A large number of solved examples is also included to illustrate the concepts.

12.1 Digital-to-Analogue Converters

A D/A converter takes digital data at its input and converts them into analogue voltage or current that is proportional to the weighted sum of digital inputs. In the following paragraphs it is briefly explained

Digital Electronics: Principles, Devices and Applications Anil K. Maini
 ISBN: 978-0-470-03214-5

how different bits in the digital input data contribute a different quantum to the overall output analogue voltage or current, and also that the LSB has the least and the MSB the highest weight.

12.1.1 Simple Resistive Divider Network for D/A Conversion

Simple resistive networks can be used to convert a digital input into an equivalent analogue output. Figure 12.1 shows one such resistive network that can convert a three-bit digital input into an analogue output. This network, however, can be extended further to enable it to perform digital-to-analogue conversion of digital data with a larger number of bits. In the network of Fig. 12.1, if R_L is much larger than R, it can be proved with the help of simple network theorems that the output analogue voltage is given by

$$V_A = \frac{[V_1/R] + [V_2/(R/2)] + [V_3/(R/4)]}{[1/R] + [1/(R/2)] + [1/(R/4)]} \tag{12.1}$$

$$= \frac{[V_1/R] + [2V_2/R] + [4V_3/R]}{[1/R] + [2/R] + [4/R]} \tag{12.2}$$

$$= \frac{V_1 + 2V_2 + 4V_3}{7} \tag{12.3}$$

which can be further expressed as

$$V_A = \frac{V_1 \times 2^0 + V_2 \times 2^1 + V_3 \times 2^2}{2^3 - 1} \tag{12.4}$$

The generalized expression of Equation (12.4) can be extended further to an n-bit D/A converter to get the following expression:

$$V_A = \frac{V_1 \times 2^0 + V_2 \times 2^1 + V_3 \times 2^2 + \cdots + V_n \times 2^{n-1}}{2^n - 1} \tag{12.5}$$

In expression (12.5), if $V_1 = V_2 = \ldots = V_n = V$, then a logic '1' at the LSB position would contribute $V/(2^n - 1)$ to the analogue output, and a logic '1' in the next adjacent higher bit position would

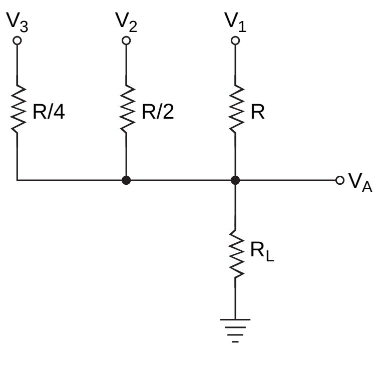


Figure 12.1 Simple resistive network for D/A conversion.

contribute $2V/(2^n - 1)$ to the output. The contributions of successive higher bit positions in the case of a logic '1' would be $4V/(2^n - 1)$, $8V/(2^n - 1)$, $16V/(2^n - 1)$ and so on. That is, the contribution of any given bit position owing to the presence of a logic '1' is twice the contribution of the adjacent lower bit position and half that of the adjacent higher bit position. When all input bit positions have a logic '1', the analogue output is given by

$$V_A = \frac{V(2^0 + 2^1 + 2^2 + \cdots + 2^{n-1})}{2^n - 1} = V \tag{12.6}$$

In the case of all inputs being in the logic '0' state, $V_A = 0$. Therefore, the analogue output varies from 0 to V volts as the digital input varies from an all 0s to an all 1s input.

12.1.2 Binary Ladder Network for D/A Conversion

The simple resistive divider network of Fig. 12.1 has two serious drawbacks. One, each resistor in the network is of a different value. Since these networks use precision resistors, the added expense becomes unattractive. Two, the resistor used for the most significant bit (MSB) is required to handle a much larger current than the LSB resistor. For example, in a 10-bit network, the current through the MSB resistor will be about 500 times the current through the LSB resistor.

To overcome these drawbacks, a second type of resistive network called the *binary ladder* (or *R*/2*R* ladder) is used in practice. The binary ladder, too, is a resistive network that produces an analogue output equal to the weighted sum of digital inputs. Figure 12.2 shows the binary ladder network for a four-bit D/A converter. As is clear from the figure, the ladder is made up of only two different values of resistor. This overcomes one of the drawbacks of the resistive divider network. It can be proved with the help of simple mathematics that the analogue output voltage V_A in the case of binary ladder network of Fig. 12.2 is given by

$$V_A = \frac{V_1 \times 2^0 + V_2 \times 2^1 + V_3 \times 2^2 + V_4 \times 2^3}{2^4} \tag{12.7}$$

In general, for an n-bit D/A converter using a binary ladder network

$$V_A = \frac{V_1 \times 2^0 + V_2 \times 2^1 + V_3 \times 2^2 + \cdots + V_n \times 2^{n-1}}{2^n} \tag{12.8}$$

For $V_1 = V_2 = V_3 = \cdots = V_n = V$, $V_A = [(2^n - 1)/2^n]V$. For $V_1 = V_2 = V_3 = \cdots = V_n = 0$, $V_A = 0$.

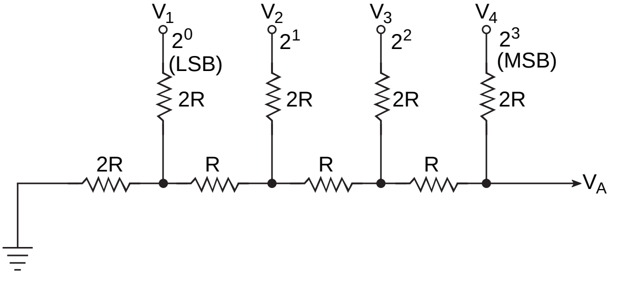


Figure 12.2 Binary ladder network for D/A conversion.

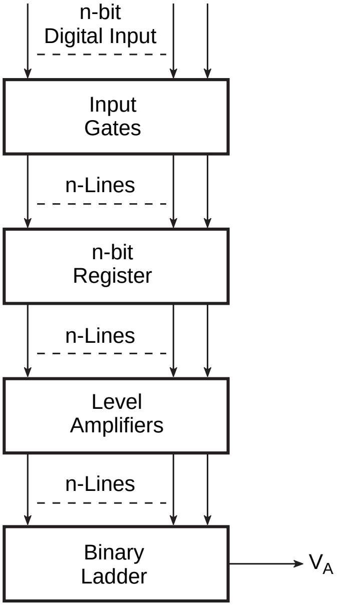


Figure 12.3 Block schematic representation of a D/A converter.

The analogue output voltage in this case varies from 0 (for an all 0s input) to $[(2^n - 1)/2^n]V$ (for an all 1s input).

Also, in the case of a resistive divider network, the LSB contribution to the analogue output is $[1/(2^n - 1)]V$. This is also the minimum possible incremental change in the analogue output voltage. The same in the case of a binary ladder network would be $(1/2^n)V$.

A binary ladder network is the most widely used network for digital-to-analogue conversion, for obvious reasons. Although actual D/A conversion takes place in this network, a practical D/A converter device has additional circuitry such as a register for temporary storage of input digital data and level amplifiers to ensure that the digital signals presented to the resistive network are all of the same level. Figure 12.3 shows a block schematic representation of a complete n-bit D/A converter. D/A converters of different sizes (eight-bit, 12-bit, 16-bit, etc.) are available in the form of integrated circuits.

12.2 D/A Converter Specifications

The major performance specifications of a D/A converter include resolution, accuracy, conversion speed, dynamic range, nonlinearity (NL) and differential nonlinearity (DNL) and monotonocity.

12.2.1 Resolution

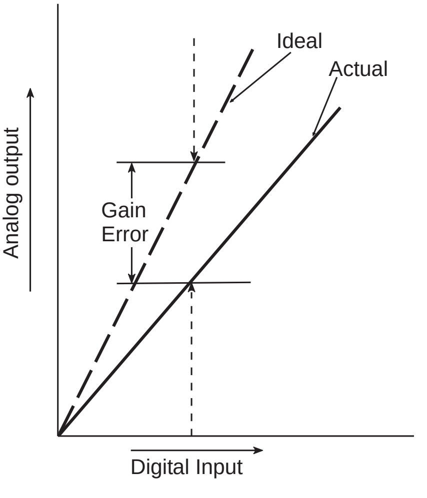
The *resolution* of a D/A converter is the number of states (2^n) into which the full-scale range is divided or resolved. Here, n is the number of bits in the input digital word. The higher the number of bits, the better is the resolution. An eight-bit D/A converter has 255 resolvable levels. It is said to have a percentage resolution of $(1/255) \times 100 = 0.39\,\%$ or simply an eight-bit resolution. A 12-bit D/A converter would have a percentage resolution of $(1/4095) \times 100 = 0.0244\,\%$. In general, for an n-bit D/A converter, the percentage resolution is given by $(1/2^n - 1) \times 100$. The resolution in millivolts for the two cases for a full-scale output of 5 V is approximately 20 mV (for an eight-bit converter) and 1.2 mV (for a 12-bit converter).

12.2.2 Accuracy

The *accuracy* of a D/A converter is the difference between the actual analogue output and the ideal expected output when a given digital input is applied. Sources of error include the *gain error* (or full-scale error), the *offset error* (or zero-scale error), *nonlinearity errors* and a drift of all these factors. The gain error [Fig. 12.4(a)] is the difference between the actual and ideal output voltage, expressed as a percentage of full-scale output. It is also expressed in terms of LSB. As an example, an accuracy of $\pm 0.1\,\%$ implies that the analogue output voltage may be off by as much as ± 5 mV for a full-scale output of 5 V throughout the analogue output voltage range. The offset error is the error at analogue zero [Fig. 12.4(b)].

12.2.3 Conversion Speed or Settling Time

The *conversion speed* of a D/A converter is expressed in terms of its settling time. The *settling time* is the time period that has elapsed for the analogue output to reach its final value within a specified error band after a digital input code change has been effected. General-purpose D/A converters have a settling time of several microseconds, while some of the high-speed D/A converters have a settling



(a)

Figure 12.4 (a) Gain error and (b) offset error.

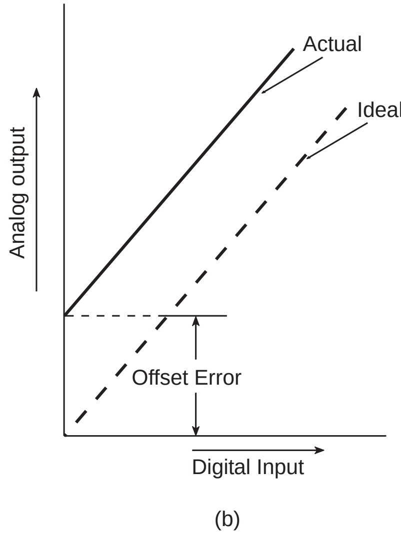


Figure 12.4 (*continued*).

time of a few nanoseconds. The settling time specification for D/A converter type number AD 9768 from Analog Devices USA, for instance, is 5 ns.

12.2.4 Dynamic Range

This is the ratio of the largest output to the smallest output, excluding zero, expressed in dB. For linear D/A converters it is $20 \times \log 2^n$, which is approximately equal to $6n$. For companding-type D/A converters, discussed in Section 12.3, it is typically 66 or 72 dB.

12.2.5 Nonlinearity and Differential Nonlinearity

Nonlinearity (NL) is the maximum deviation of analogue output voltage from a straight line drawn between the end points, expressed as a percentage of the full-scale range or in terms of LSBs. *Differential nonlinearity* (DNL) is the worst-case deviation of any adjacent analogue outputs from the ideal one-LSB step size.

12.2.6 Monotonocity

In an ideal D/A converter, the analogue output should increase by an identical step size for every one-LSB increment in the digital input word. When the input of such a converter is fed from the output of a counter, the converter output will be a perfect staircase waveform, as shown in Fig. 12.5. In such cases, the converter is said to be exhibiting perfect monotonocity. A D/A converter is considered as monotonic if its analogue output either increases or remains the same but does not decrease as the digital input code advances in one-LSB steps. If the DNL error of the converter is less than or equal to twice its worst-case nonlinearity error, it guarantees monotonocity.

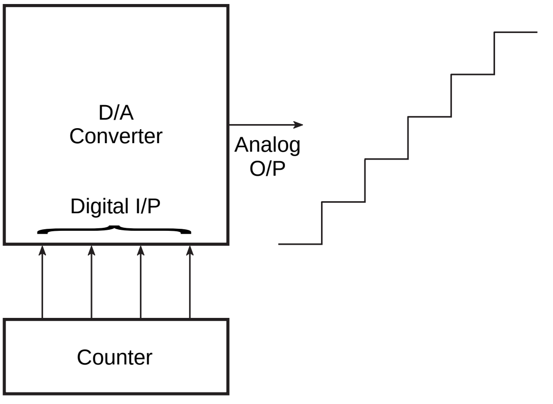


Figure 12.5 Monotonocity in a D/A converter.

12.3 Types of D/A Converter

The D/A converters discussed in this section include the following:

1. Multiplying-type D/A converters.
2. Bipolar-output D/A converters.
3. Companding D/A converters.

12.3.1 Multiplying D/A Converters

In a *multiplying-type D/A converter*, the converter multiplies an analogue reference by the digital input. Figure 12.6 shows the circuit representation. Some D/A converters can multiply only positive digital words by a positive reference. This is known as single quadrant (QUAD-I) operation. Two-quadrant operation (QUAD-I and QUAD-III) can be achieved in a D/A converter by configuring the output for bipolar operation. This is accomplished by offsetting the output by a negative MSB (equal to the analogue output of 1/2 of the full-scale range) so that the MSB becomes the sign bit.

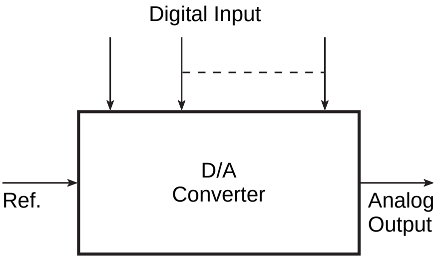


Figure 12.6 Multiplying-type D/A converter.

Some D/A converters even provide four-quadrant operation by allowing the use of both positive and negative reference. Multiplying D/A converters are particularly useful when we are looking for digitally programmable attenuation of an analogue input signal.

12.3.2 Bipolar-Output D/A Converters

In *bipolar-output D/A converters* the analogue output signal range includes both positive and negative values. The transfer characteristics of an ideal two-quadrant bipolar-output D/A converter are shown in Fig. 12.7.

12.3.3 Companding D/A Converters

Companding-type D/A converters are so constructed that the more significant bits of the digital input have a larger than binary relationship to the less significant bits. This decreases the resolution of the more significant bits, which in turn increases the analogue signal range. The effect of this is to compress more data into more significant bits.

12.4 Modes of Operation

D/A converters are usually operated in either of the following two modes of operation:

1. Current steering mode.
2. Voltage switching mode.

12.4.1 Current Steering Mode of Operation

In the *current steering mode* of operation of a D/A converter, the analogue output is a current equal to the product of a reference voltage and a fractional binary value D of the input digital word. D is equal to the sum of fractional binary values of different bits in the digital word. Also, fractional binary values of different bits in an n-bit digital word starting from the LSB are $2^0/2^n$, $2^1/2^n$, $2^2/2^n, \ldots, 2^{n-1}/2^n$.

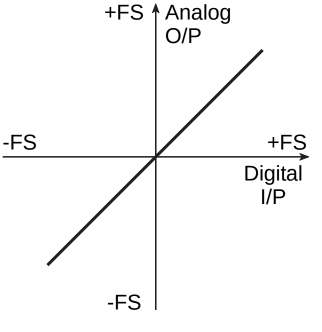


Figure 12.7 Bipolar-output D/A converter transfer characteristics.

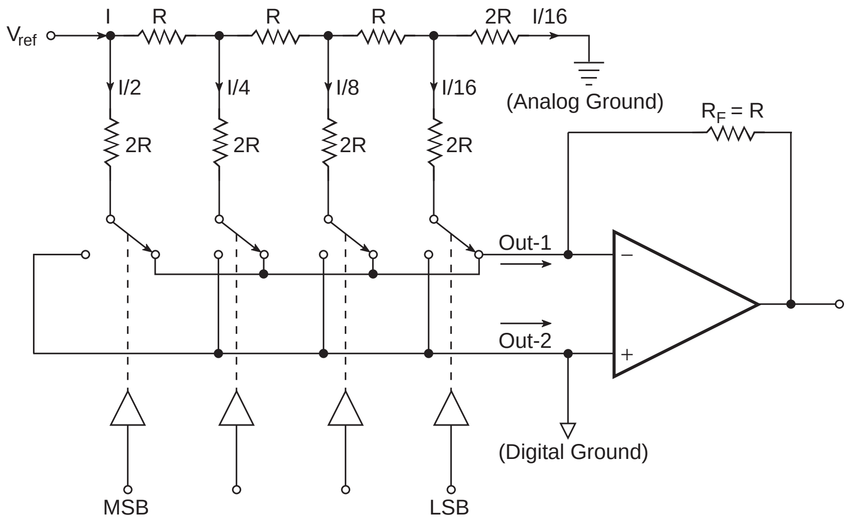


Figure 12.8 Current steering mode of operation of a D/A converter.

The output current is often converted into a corresponding voltage using an external opamp wired as a current-to-voltage converter. Figure 12.8 shows the circuit arrangement. The majority of D/A converters in IC form have an in-built opamp that can be used for current-to-voltage conversion. For the circuit arrangement of Fig. 12.8, if the feedback resistor R_F equals the ladder resistance R, the analogue output voltage at the opamp output is $-(D.V_{\mathrm{ref}})$.

The arrangement of the four-bit D/A converter of Fig. 12.8 can be conveniently used to explain the operation of a D/A converter in the current steering mode. The $R/2R$ ladder network divides the input current I due to a reference voltage V_{ref} applied at the reference voltage input of the D/A converter into binary weighted currents, as shown. These currents are then steered to either the output designated Out-1 or Out-2 by the current steering switches. The positions of these current steering switches are controlled by the digital input word. A logic '1' steers the corresponding current to Out-1, whereas a logic '0' steers it to Out-2. For instance, a logic '1' in the MSB position will steer the current $I/2$ to Out-1. A logic '0' steers it to Out-2, which is the ground terminal. In the four-bit converter of Fig. 12.8, the analogue output current (or voltage) will be maximum for a digital input of 1111. The analogue output current in this case will be $I/2 + I/4 + I/8 + I/16 = (15/16)I$. The analogue output voltage will be $(-15/16)IR_F = (-15/16)IR$. Also, $I = V_{\mathrm{ref}}/R$ as the equivalent resistance of the ladder network across V_{ref} is also R. The analogue output voltage is then $[(-15/16)(V_{\mathrm{ref}})/R] \times R = (-15/16)V_{\mathrm{ref}}$. Here, 15/16 is nothing but the fractional binary value of digital input 1111. In general, the maximum analogue output voltage is given by $-(1-2^{-n}) \times V_{\mathrm{ref}}$, where n is the number of bits in the input digital word.

12.4.2 Voltage Switching Mode of Operation

In the *voltage switching mode* of operation of a $R/2R$ ladder type D/A converter, the reference voltage is applied to the Out-1 terminal and the output is taken from the reference voltage terminal. Out-2 is joined to analogue ground. Figure 12.9 shows a four-bit D/A converter of the $R/2R$ ladder type in