INTRODUCTION TO DIGITAL SYSTEMS

INTRODUCTION TO DIGITAL SYSTEMS

Modeling, Synthesis, and Simulation Using VHDL

Mohammed Ferdjallah
The Virginia Modeling, Analysis and Simulation Center
Old Dominion University
Suffolk, Virginia
and ECPI College of Technology

WILEY
A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Ferdjallah, Mohammed.
Introduction to digital systems : modeling, synthesis, and simulation using VHDL / Mohammed Ferdjallah.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-90055-0 (cloth)
1. Digital electronics. 2. Digital electronics–Computer simulation. 3. VHDL (Computer hardware description language) I. Title.
TK7868.D5F47 2011
621.39'2–dc22 2010041036

Printed in the United States of America

oBooK ISBN: 9781118007716
ePDF ISBN: 9781118007693
ePub ISBN: 9781118007709

10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

Digital system design requires rigorous modeling and simulation analysis that eliminates design risks and potential harm to users. Thus, the educational objective of this book is to provide an introduction to digital system design through modeling, synthesis, and simulation computer-aided design (CAD) tools. This book provides an introduction to analytical and computational methods that allow students and users to model, synthesize, and simulate digital principles using very high-speed integrated-circuit hardware description language (VHDL) programming. We present the practical application of modeling and synthesis to digital system design to establish a basis for effective design and provide a systematic tutorial of how basic digital systems function. In doing so, we integrate theoretical principles, discrete mathematical models, computer simulations, and basics methods of analysis. Students and users will learn how to use modeling, synthesis, and simulation concepts and CAD tools to design models for digital systems that will allow them to gain insights into their functions and the mechanisms of their control. Students will learn how to integrate basic models into more complex digital systems. Although the approach designed in this book focuses on undergraduate students, it can also be used for modeling and simulation students who have a limited engineering background with an inclination to digital systems for visualization purposes.

The book includes nine chapters. Each chapter begins with learning objectives that provide a brief overview of the concepts that the reader is about to learn. In addition, the learning objectives can be used as points for classroom discussion. Each chapter ends with problems that will enable students to practice and review the concepts covered in the chapter. Chapter 1 introduces modeling and simulation and its role in digital system evolution. The chapter provides a brief history of modeling and simulation in digital systems, VHDL programming, programmable and reconfigurable systems, and advantages of using modeling and simulation in digital system design. Chapter 2 introduces the mathematical foundations of digital systems and logical reasoning. Described are Boolean theory, its axioms and theorems, and basic logic gates as well as early modeling in digital system design using algebraic manipulations.

Chapter 3 provides an overview of number representations, number conversions, and number codes. The relationships between decimal representation and the less obvious digital number representations are described. Chapter 4 provides a brief history of VHDL programming, the reasons for its creation, and its impact on the evolution of digital systems and modern computer systems. Described are CAD tools, programming structure, and instructions and syntax of VHDL. Chapter 5 provides a simplified view of the progression of integrated systems and their application in

digital logic circuits and computer systems. The role of modeling and simulation in the optimization and verification of digital system design at the transistor level is described. Chapter 6 provides graphical means and Karnaugh maps to streamline and simplify digital system design using visualization schemes. Although these methods are used only when designing circuits with a small number of gates, they provide rudimentary means for the design of automatic CAD tools.

Chapter 7 introduces combinational logic and its applications in multiplexers, decoders, and arithmetic and logic circuits and systems. Chapter 8 introduces sequential logic, with a focus on sequential logic elementary circuits and their applications in complex circuits such as counters and registers. Chapter 9 provides an overview of finite-state machines, especially the synchronous sequential circuit models used to design simple finite-state machines. Also described is asynchronous sequential logic and its advantages and disadvantages for digital systems. All chapters illustrate circuit design using VHDL sample codes that allow students not only to learn and master VHDL programming but also to model and simulate digital circuits.

MOHAMMED FERDJALLAH

1 Digital System Modeling and Simulation

1.1 OBJECTIVES

The objectives of the chapter are to:

- Describe digital systems
- Provide a brief history of digital systems
- Describe standard chips
- Describe custom-designed chips
- Describe programmable logic devices
- Describe field-programmable gated arrays

1.2 MODELING, SYNTHESIS, AND SIMULATION DESIGN

Modeling and simulation have their roots in digital systems. Long before they became the basis of an interdisciplinary field, modeling and simulation were used extensively in digital system design. As electronic and computer technology advanced, so did modeling and simulation concepts. Today, the many computer-aided design (CAD) tools are pushing the limit of modeling, synthesis, and simulation technology. We focus on the implementation of modeling, synthesis, and simulation in digital systems.

A *digital system* is a system that takes digital signals as inputs, processes them, and produces digital output signals. A *digital signal* is a signal in which discrete steps are used to represent information and change values only at discrete (fixed) time intervals. In contrast, *analog signals* have "continuous" variations in signal amplitude over time. At a given instant of time, an analog signal has infinite possible values. A digital signal has discrete amplitude and time. Digital systems are very useful in the areas of signal processing (i.e., audio, images, speech, etc.), computing, communication, and data storage, among others. Digital systems are so commonplace in today's world that we tend to miss seeing them. Almost all electronic systems are partially or totally

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

digitally based. Of course, real-world signals are all analog, and interfacing to the outside world requires conversion of a signal (information) from digital to analog. However, simplicity, versatility, repeatability, and the ability to produce large and complex (as far as functionality is concerned) systems economically make them excellent for processing and storing information (data).

1.3 HISTORY OF DIGITAL SYSTEMS

One of the earliest digital systems was the dial telephone system. Pulses generated by activating a spinning dial were counted and recorded by special switches in a central office. After all the numbers had been dialed and recorded, switches were set to connect the user to the desired party. A switch is a digital device that can take one of two states: open or closed.

In 1939, Harvard University built the Harvard Mark I, which went into operation in 1943. It was used to compute ballistic tables for the U.S. Navy. In the next few years, more machines were built in research laboratories around the world. The ENIAC (Electronic Numerical Integrator and Computer) was placed in operation at the Moore School of Electrical Engineering at the University of Pennsylvania, component by component, beginning with the cycling unit and an accumulator in June 1944. This was followed in rapid succession by the initiating unit and function tables in September 1945 and the divider and square-root unit in October 1945. Final assembly of this primitive computer system took place during the fall of 1945.

The first commercially produced computer was Univac I, which went into operation in 1951. More large digital computers were introduced in the next decade. These first-generation computers used vacuum tubes and valves as primary electronic components and were bulky, expensive, and consumed immense amounts of power. The invention of the transistor in 1948 at the Bell Telephone Laboratories by physicists John Bardeen, Walter Brattain, and William Shockley revolutionized the way that computers were built. Transistors are used as electrical switches that can be in the "on" or "off" state and so can be used to build digital circuits and systems. Transistors were used initially as discrete components, but with the arrival of integrated circuit (IC) technology, their utility increased exponentially. ICs are inexpensive when produced in large numbers, reliable, and consume much less power than do vacuum tubes. IC technology makes it possible to build complete digital building blocks into single, minute silicon "chips." The size of transistors has been shrinking ever since their birth, and today, a complete computer is on one chip (microprocessor), and even large systems are being integrated into a single chip (system-on-a-chip).

1.4 STANDARD LOGIC DEVICES

Many commonly used logic circuits are readily available as integrated circuits. These are referred to as *standard chips* because their functionality and configuration

meet agreed-upon standards. These chips generally have a few hundred transistors at most. They can be bought off-the-shelf, and depending on the application, the designer can build supporting circuitry on a PCB (printed circuit board) or breadboard. The advantages of using standard chips are their ease of use and ready availability. However, their fixed functionality has proved disadvantageous. Also, the fact that they generally do not have complex functionality means that many such chips have to be put together on a PCB, leading to a requirement for more area and components. Examples of standard chips are those in the 7400 series, such as the 7404 (hex inverters) and 7432 (quad two-input OR gates).

1.5 CUSTOM-DESIGNED LOGIC DEVICES

Chips designed to meet the specific requirements of an application are known as *application-specific integrated circuits* (ASICs) or custom-designed chips. The logic chip is designed from scratch. The logic circuitry is designed according to the specifications and then implemented in an appropriate technology. The main advantage of ASICs is that since they are optimized for a specific application, they perform better than do functionally equivalent circuits built from off-the-shelf ICs or programmable logic devices. They occupy very little area, as all of the logic can be built into one chip. Thus, less PCB area would be required, leading to some cost savings. The disadvantage of ASICs is that they can be justifiable economically only when there is bulk production of the ICs. Typically, hundreds of thousands of ASICs must be manufactured to recover the expenditures necessary in the design, manufacturing and testing stages. Another drawback of the custom-design approach is that it requires the work of highly skilled engineers in the design, manufacturing, and test stages. The design time needed for these chips is also high, as a lot of verification has to be carried out to check for correct functionality. The circuitry in the chip cannot be altered once it is fabricated.

1.6 PROGRAMMABLE LOGIC DEVICES

Advances in VLSI technology made possible the design of special chips, which can be configured by a user to implement different logic circuits. These chips, known as *programmable logic devices* (PLDs), have a very general structure and contain *programmable switches,* which allow the user to configure the internal circuitry to perform the desired function. The programmer (end user) has simply to change the configuration of these switches. This is usually done by writing a program in a hardware description language (HDL) such as VHDL or Verilog and "downloading" it into the chip. Most types of PLDs are reprogrammable for a fixed number of times (generally, a very high number). This makes PLDs excellent for use in prototyping of ASICs and standard chips. A designer can program a PLD to perform a particular function and then make changes and reprogram it for retesting on the same chip. Also, there is a great cost savings in using a device that is reprogrammable for

prototyping purposes. The main disadvantage of PLDs is that they may not be the best performing. The performance of a functionally equivalent ASIC or standard chip is likely to be better. This is because all functions have to be realized from existing blocks of logic inside the PLD. The most popular types of PLDs include:

- Simple programmable logic devices (SPLDs)
- Programmable array logic (PAL)
- Programmable logic array (PLA)
- Generic array logic (GAL)
- Complex programmable logic devices (CPLDs)
- FPGA (field-programmable gate arrays)
- FPIC (field-programmable interconnect)

These different types of PLDs vary in their internal architectures. Different manufacturers of PLDs choose different architectures for implementing the logic blocks and the programmable interconnection switch matrices. FPGAs have the highest *gate count* among the various PLDs, which can accommodate much larger designs than can SPLDs and CPLDs. Today's FPGAs have millions of transistors in one chip. PALs and PLAs generally carry just a few hundred or a few thousand gates. PLD manufacturers include, among others, Altera Corporation, Xilinx Inc., Lattice Semiconductor, Cypress Semiconductor, Atmel, Actel, Lucent Technologies, and QuickLogic.

1.7 SIMPLE PROGRAMMABLE LOGIC DEVICES

Simple programmable logic devices (SPLDs) include programmable logic arrays (PLAs) and programmable array logic (PALs). Early SPLDs were simple and consisted of an array of AND gates driving an array of OR gates. An AND gate (known as an AND *plane* or AND *array*) feeds a set of OR gates (an OR *plane*). This helps in realizing a function in the *sum-of-products* form.

Figure 1.1 shows the general architecture of PLAs and PALs. The most common housing of PLAs and PALs was a 20-pin dual-in-line package (DIP). The difference between PALs and PLAs is that in PLA, both the AND and OR planes are programmable, whereas in PALs, the AND plane is programmable but the OR plane is fixed. PLAs were expensive to manufacture and offered somewhat poor performances, due to propagation delays. Therefore, PALs were introduced for their ease of manufacturability, lower cost of production, and better performance. PALs usually contain flip-flops connected to the OR gates to implement sequential circuits. Both PLAs and PALs use antifuse switches, which remain in a high-impedance state until programmed into a low-impedance (fused) state. These devices are generally programmed only once. Generic array logic devices (GALs) are similar to PALs but can be reprogrammed. PLAs, PALs, and GALs are programmed using a PAL programmer device (a burner).

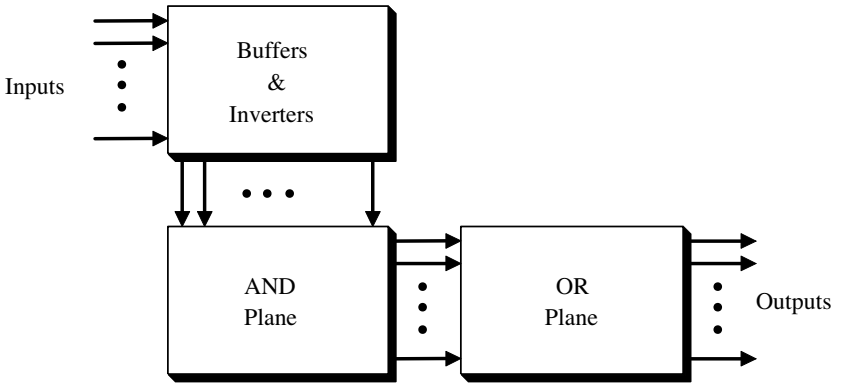


Figure 1.1 Schematic Structure of PALs and PLAs

1.8 COMPLEX PROGRAMMABLE LOGIC DEVICES

PALs and PLAs are useful for small digital circuits which do not require more than 32 inputs and outputs. To implement circuits that need more inputs and outputs, multiple PLAs or PALs can be used. However, this will compromise the performance of the design and also occupy more area on the PCB. In such situations, a complex programmable device (CPLD) would be a better choice. A CPLD comprises multiple circuit blocks on a single chip. Each block is similar to a PLA or PAL. There could be as few as two such blocks in a CPLD and 100 or more such blocks in larger CPLDs. These logic blocks are interconnected through a *programmable switch matrix* or *interconnection array*, which allows all blocks of the CPLD to be interconnected. Figure 1.2 shows the internal structure of a CPLD. As a result of this configuration, the architecture of the CPLD is less flexible. However, the propagation delay of a CPLD is

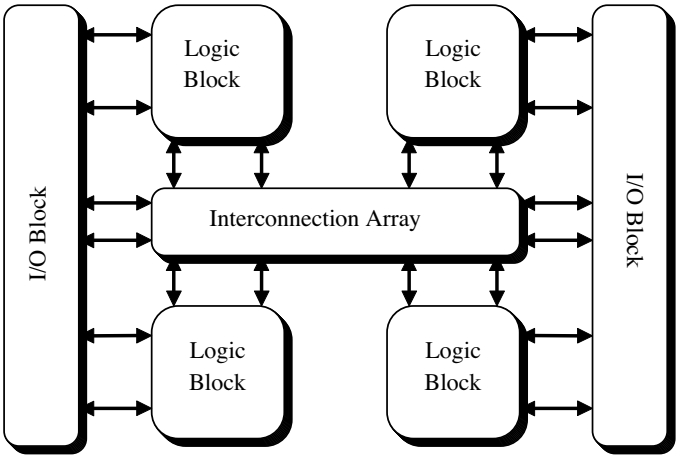


Figure 1.2 CPLD Internal Structure

predictable. This advantage allowed CPLDs to emulate ASIC systems, which operate at higher frequencies.

1.9 FIELD-PROGRAMMABLE GATE ARRAYS

Field-programmable gate arrays (FPGAs) differ from the other PLDs and generally offer the highest logic capacity. An FPGA consists of an array of complex logic blocks (CLBs) surrounded by programmable I/O blocks (IOBs) and connected by a programmable interconnection network. The IOBs provide the control between the input–output package pins and the internal signal lines, and the programmable interconnect resources provide the correct paths to connect the inputs and outputs of CLBs and IOBs into the appropriate networks. The logic cells combinational logic may be implemented physically as a small lookup table memory (LUT) or a set of multiplexers and gates. An LUT is a 1-bit-wide memory array; the memory address lines are logic block inputs and the 1-bit-memory output is the lookup table output.

A typical FPGA may contain tens of thousands of (configurable) logic blocks and an even greater number of flip-flops. The user's logic function is implemented by closing the switches in the interconnect matrix that specify the logic function for each logic cell. Complex designs are then created by combining these basic blocks to create the desired circuit. Typically, FPGAs do not provide a 100% interconnect between logic blocks (Figure 1.3). There are four main categories of FPGAs currently available commercially: symmetrical array, row-based, hierarchical PLD, and sea of gates. Currently, the four technologies in use are static RAM cells, antifuse, EPROM transistors, and EEPROM transistors. Static RAM is common in most FPGAs. It loses all knowledge of the program once power is removed from it. It has no memory built

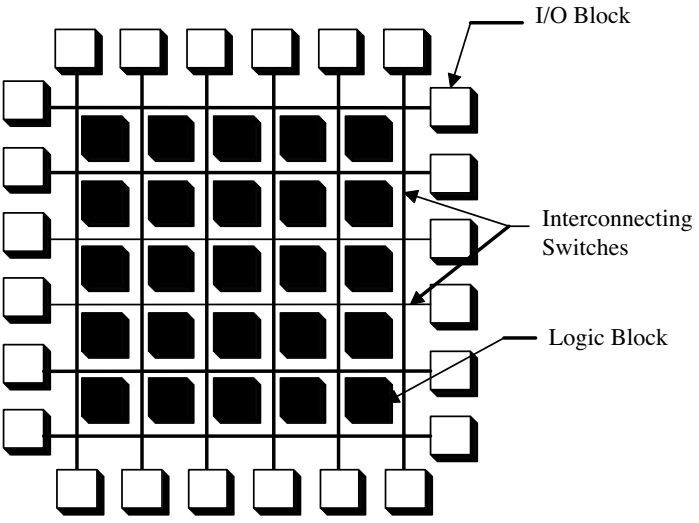


Figure 1.3 FPGA Internal Structure

into the chip and upon each power-up must depend on some external source to upload its memory. EPROM-based programmable chips cannot be reprogrammed in-circuit and need to be cleared with ultraviolet (UV) erasing. EEPROM chips can be erased electrically but generally cannot be reprogrammed in-circuit.

Some FPGA device manufacturing companies are Altera, Cypress, QuickLogic, Xilinx, Actel, and Lattice Semiconductor. In the early years, Xilinx was a leading manufacturer and designer of FPGAs. Xilinx produced the first static random access memory FPGA. The drawback of a SRAM FPGA is the loss of memory after a loss of power. Actel created a more stable FPGA using antifuse technology. This design provided a buffer to the loss of memory, kept the cost of each gate low, ran extremely fast, and provided protection against industrial pirating. SRAMs, were easy to design however, and the addition of antifuse technology would make the design process longer. SRAM FPGAs are the majority choice of designers today.

A FPGA vendor usually provides software that "places and routes" the logic on the device (similar to the way in which a PCB autorouter would place and route components on a board). There are a wide variety of subarchitectures within the FPGA family. The key to the performance of these devices lies in the internal logic contained in their logic blocks and on the performance and efficiency of their switch matrix. The behavior of an FPGA is accomplished using a hardware descriptive language (HDL) or an electronic design automation tool to create a design schematic. When this process is completed, it can be compiled to generate a net list. The net list can then be mapped to the architecture of the FPGA. The binary file that is generated is used to reconfigure the FPGA device. The most common hardware descriptive languages in the design industry are VHDL and Verilog. The design process of programming an FPGA consists of design entry, simulation, synthesis, place and route, and download. Design libraries are a common part of the software used in programming FPGAs. These libraries contain programs of widely used functions and possess the ability to add new programs provided by the user. Design constraints are preset by the need for the design and the flexibility of the components reproduced that are used by the program.

1.10 FUTURE OF DIGITAL SYSTEMS

The latest microprocessors for home computing applications run at about 3 GHz. Most chips available commercially use the bulk-CMOS (complementary metal–oxide semiconductor) process to manufacture the transistor circuits. Also, most digital designs are synchronous in nature. Synchronous systems are also referred to as *clocked systems*. The latest commercially available chips are manufactured using the 90-nm process. Over the next few years, companies expect to move to 65 nm or lower. Some experts in the semiconductor industry see an asynchronous future for digital designs. Asynchronous systems are digital systems that do not use a clock to time events. Chip size has been shrinking continuously, and designs have become more complex than ever. Emerging technologies such as hybrid ASIC and LPGA (laser programmable gate array) make the future exciting. New materials, design

methodologies, better fabrication facilities, and newer applications are certainly making things interesting. The Semiconductor Industry Association (SIA) predicts that the worldwide per capita production of transistors will soon be 1 billion per person.

In particular, FPGAs are leading the way to a technological revolution. Many emerging applications in the communication, computing, and consumer electronics industries demand that their functionality stays flexible after the system has been manufactured. Such flexibility is required in order to cope with changing user requirements, improvements in system features, changing standards, and demands to support a variety of user applications. With the vast array that FPGAs provide, hardware design has never been easier to develop or implement. Design revisions can be implemented effortlessly and painlessly. Currently, they are still under development to become faster and easier to program then their CPLD counterparts are now, but soon the technology will be a reality and the possibility for complete and total reconfigurable systems will become real. One day, a computer could program itself to run faster and more efficiently with no help from the user.

PROBLEMS

1.1 What is a digital system?

1.2 Describe computer-aided design software tools.

1.3 Explain Moore's law.

1.4 What does "PCB" stand for?

1.5 Describe the advantages and disadvantages of standard chips.

1.6 Describe the advantages and disadvantages of programmable logic devices.

1.7 Describe the advantages and disadvantages of custom logic devices.

1.8 Describe the advantages and disadvantages of reconfigurable logic devices.

1.9 Describe the basic design process for digital systems.

2 Number Systems

2.1 OBJECTIVES

The objectives of the chapter are to describe:

- Number systems
- Number conversion
- Data organization
- Unsigned and signed numbers
- Binary arithmetic
- Hexadecimal arithmetic
- Number codes

2.2 BASES AND NUMBER SYSTEMS

The objective of this section is to introduce the various types of number representations used in digital system designs. The general method for numerical representation is called *positional number representation*. Consider the familiar decimal system. A number in decimal representation is made of digits that range from 0 to 9. Consider the following decimal number:

$$(4261)_{10} = 4 \times 10^3 + 2 \times 10^2 + 6 \times 10^1 + 1 \times 10^0$$

This number is normally written as *4261*, as the powers of 10 are implied by the position of that particular digit. Therefore, a decimal number N with n digits can be expressed as follows:

$$(N)_{10} = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

Decimal representations are said to be *base-10* or *radix-10 numbers* because each digit has 10 possible values, weighted as a power of 10, depending on the position of

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

the digit in the number. In a similar way, in binary representation, each binary digit has two possible values, 1 and 0, and the digits are weighted as a power of 2, depending on their position in the number. The *binary system* of representation is also known as a *base-2 system*. Consider the following binary number:

$$(1001)_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (9)_{10}$$

Similarly, a binary number N with n digits can be expressed as follows:

$$(N)_2 = d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \cdots + d_1 \times 2^1 + d_0 \times 2^0$$

In general, any number N can be represented in a base b by the following power series:

$$(N)_r = d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \cdots + d_0 \times r^0 + d_{-1} \times r^{-1} + \cdots + d_{-m} \times r^{-m}$$

The coefficients (d_i) are called *digits*, and r represents the *radix* or *base*. In the binary system the digits are referred to as *bits*. There are four types of numerical representations: binary, decimal, octal, and hexadecimal. Numbers in binary form can be rather long, exhausting, and difficult to remember. Binary numbers are often represented in more compact forms using the octal (base 8) and hexadecimal (base 16) systems. The digits 0 through 7 are used in the octal system. The hexadecimal system uses the digits 0 through 9 and the letters A through F, where A represents the decimal 10 and F represents the decimal 15 (Figure 2.1).

Binary representation is by and far the most commonly used system in computer design. The only reason for using octal and hexadecimal numbers is the convenience in programming. The following examples illustrate power series expansions of a binary, decimal, octal and a hexadecimal number.

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

Figure 2.1 Decimal, Binary, Hexadecimal, and Octal Systems

$$(1834)_{10} = 1 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$
$$(1567)_8 = 1 \times 8^3 + 5 \times 8^2 + 6 \times 8^1 + 7 \times 8^0 = (887)_{10}$$
$$(101011)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (43)_{10}$$
$$(2F3A)_{16} = 2 \times 16^3 + 15 \times 16^2 + 3 \times 16^1 + 10 \times 16^0 = (5678)_{10}$$

2.3 NUMBER CONVERSIONS

Conversions of binary numbers to other number systems, and vice versa, are common in input–output routines. The following sections illustrate the conversion of numbers between number representation systems.

2.3.1 Decimal-to-Binary Conversion

Learning by example would probably be the best way to become familiar with number conversions. The decimal value of a binary number is easily calculated by summing the power terms with nonzero coefficients. Continuous division by 2 obtains the binary form of a decimal number until the final result is equal to zero. The remainder is saved after each division step. The first remainder is the *least significant* bit (LSB) and the last remainder is the *most significant bit* (MSB) of the resulting binary number. The following example illustrates decimal-to-binary conversion.

$(153)_{10} \div 2 = 76$	remainder is	1	LSB
$76 \div 2 = 36$	remainder is	0	
$36 \div 2 = 18$	remainder is	0	
$18 \div 2 = 9$	remainder is	0	
$9 \div 2 = 4$	remainder is	1	
$4 \div 2 = 2$	remainder is	0	
$2 \div 2 = 1$	remainder is	0	
$1 \div 2 = 0$	remainder is	1	MSB

Therefore,

$$(153)_{10} = (10010001)_2$$

A conversion is carried out by first dividing the given decimal number by 2. The quotient that results from each division step is again divided by 2 and the remainders are noted in each step. The remainders form the actual binary number. The quotient from the first division step forms the least significant bit (LSB) and the quotient from the last division step forms the most significant bit (MSB).

2.3.2 Decimal-to-Octal Conversion

The same process may be applied to convert decimal numbers to octal numbers by continuous division by 8. It is similar to converting a decimal number to its binary

form, but instead of dividing by 2, the quotient is divided by 8. The remainders form the octal equivalent. The following example illustrates decimal-to-octal conversion:

$$\begin{aligned}
(3564)_{10} \div 8 = 445 \quad & \text{remainder is } 4 \quad \text{LSB} \\
445 \div 8 = 55 \quad & \text{remainder is } 5 \\
55 \div 8 = 6 \quad & \text{remainder is } 7 \\
6 \div 8 = 0 \quad & \text{remainder is } 6 \quad \text{MSB}
\end{aligned}$$

Therefore,

$$(3564)_{10} = (6754)_8$$

2.3.3 Decimal-to-Hexadecimal Conversion

The same process may be applied to convert decimal numbers to hexadecimal numbers by continuous division by 16. Similarly, the decimal number is divided continuously by 16. The remainders form the hexadecimal equivalent. The following example illustrates decimal-to-hexadecimal conversion.

$$\begin{aligned}
(37,822)_{10} \div 16 &= 2363 \quad \text{remainder is } 14 \text{ or E} \quad \text{LSB} \\
2,363 \div 16 &= 147 \quad \text{remainder is } 11 \text{ or B} \\
147 \div 16 &= 9 \quad \text{remainder is } 3 \\
9 \div 16 &= 0 \quad \text{remainder is } 9 \quad \text{MSB}
\end{aligned}$$

Therefore,

$$(37,822)_{10} = (\text{EB } 39)_{16}$$

2.3.4 Binary-to-Octal and Hexadecimal Conversions

The conversion of a binary number to an octal number or a hexadecimal number requires converting the binary digits in groups of 3 or 4, respectively, starting from the least significant bit. Given a binary number, the octal number is formed by taking groups of 3 bits starting from the LSB and replacing each group with the corresponding octal digit. The following examples illustrate binary-to-octal conversion.

$$(10011001)_2 = 10\ \ 011\ \ 001 = (231)_8$$
$$(111010000110011)_2 = 111\ \ 010\ \ 000\ \ 110\ \ 011 = (72063)_8$$

Similarly, given a binary number, the hexadecimal number is formed by taking groups of 4 bits starting from the LSB and replacing each group with the corresponding hexadecimal digit. The following examples illustrate binary-to-hexadecimal conversion:

$$(110010001010)_2 = 1100\ \ 1000\ \ 1010 = (\text{C8A})_{16}$$
$$(0010111000011011)_2 = 0010\ \ 1110\ \ 0001\ \ 1011 = (\text{2E1B})_{16}$$

To convert an octal number or a hexadecimal number to a binary number, each octal or hexadecimal digit is simply converted to its binary form. The following examples illustrate octal-to-binary and hexadecimal-to-binary conversions:

$$(F1)_{16} = (11110001)_2 \quad (A8)_{16} = (10101000)_2$$
$$(123)_8 = (001010011)_2 \quad (247)_8 = (010100111)_2$$

Once the binary number is formed, it can be converted into any of the other representation systems using the procedures above.

2.4 DATA ORGANIZATION

A binary number is a sequence of bits that may represent an actual binary number, a character, or an instruction. Therefore, microcomputers must use a specific data structure or big groupings to express the various binary representations. As learned earlier, in each binary grouping the rightmost bit is called the least significant bit and the leftmost bit is called the most significant bit. A group of consecutive 4 bits is called a *nibble*. A nibble is used to represent a BCD or hexadecimal digit. A group of consecutive 8 bits is called a *byte*, which is the smallest addressable data in memory. A byte is also used to represent an alpha-numeric character. A group of consecutive 16 bits, called a *word*, can be divided into a high byte and a low byte. For example, in 16-bit general-purpose registers and accumulators, the high and low bytes can be manipulated separately. In general, the size of the microcomputer internal registers determines the size of binary grouping. A 16-bit microcomputer has two bytes, or a 16-bit word size. However, the memory unit is divided into an 8-bit, or byte, word length. For example, to store a 16-bit number, the microcomputer uses two consecutive byte locations in the memory space. High-end (32 and 64-bit) microcomputers use double-word and quad-word data structures. These wide data structures are used mainly in highly pipelined and parallel microcomputers.

2.5 SIGNED AND UNSIGNED NUMBERS

Unsigned binary numbers are, by definition, positive numbers and thus do not require an arithmetic sign. An m-bit unsigned number represents all numbers in the range 0 to $2^m - 1$. For example, the range of 8-bit unsigned binary numbers is from 0 to 255_{10} in decimal and from 00 to FF_{16} in hexadecimal. Similarly, the range of 16-bit unsigned binary numbers is from 0 to $65{,}535_{10}$ in decimal and from 0000 to $FFFF_{16}$ in hexadecimal.

Signed numbers, on the other hand, require an arithmetic sign. The most significant bit of a binary number is used to represent the sign bit. If the sign bit is equal to zero, the signed binary number is positive; otherwise, it is negative. The remaining bits represent the actual number. There are three ways to represent negative numbers.

2.5.1 Sign–Magnitude Representation

In the sign–magnitude representation method, a number is represented in its binary form. The most significant bit (MSB) represents the *sign*. A 1 in the MSB bit position denotes a negative number; a 0 denotes a positive number. The remaining $n-1$ bits are preserved and represent the *magnitude* of the number. The following examples illustrate the sign–magnitude representation:

$$\begin{aligned}(+3) = 0011 \quad &\Rightarrow \quad (-3) = 1011\\ (+7) = 0111 \quad &\Rightarrow \quad (-7) = 1111\\ (+0) = 0000 \quad &\Rightarrow \quad (-0) = 1000\end{aligned}$$

2.5.2 One's-Complement Representation

In the one's-complement form, the MSB represents the sign. The remaining bits are inverted for negative numbers only. Positive numbers are represented in the same way as in the sign–magnitude method. The following examples illustrate the one's-complement representation:

$$\begin{aligned}(+3) = 0011 \quad &\Rightarrow \quad (-3) = 1100\\ (+7) = 0111 \quad &\Rightarrow \quad (-7) = 1000\\ (+0) = 0000 \quad &\Rightarrow \quad (-0) = 1111\end{aligned}$$

The decimal number equivalent to a binary number represented using the one's-complement method can be computed using the expression

$$(N)_{10} = S \times (2^n - 1) + (d_{n-2} \times 2^{n-2} + d_{n-3} \times 2^{n-3} + \cdots + d_1 \times 2^1 + d_0 \times 2^0)$$

where S is the sign bit and n is the number of bits.

2.5.3 Two's-Complement Representation

In the two's-complement method, the negative numbers are inverted and augmented by one. The MSB is the sign bit. The positive numbers are similar to those of the sign–magnitude method. The following examples illustrate the one's-complement representation:

$$\begin{aligned}(+3) = 0011 \quad &\Rightarrow \quad (-3) = 1101\\ (+7) = 0111 \quad &\Rightarrow \quad (-7) = 1001\\ (+0) = 0000 \quad &\Rightarrow \quad (-0) = 0000\end{aligned}$$

The decimal number equivalent to a binary number represented using the two's-complement method is obtained by subtracting an n-bit positive number from 2^n:

$$(N)_{10} = S \times 2^n + (d_{n-2} \times 2^{n-2} + d_{n-3} \times 2^{n-3} + \cdots + d_1 \times 2^1 + d_0 \times 2^0)$$

where S is the sign bit and n is the number of bits.

2.5.4 Negative Number Representation

Figure 2.2 summarizes the three methods used for 4-bit signed binary numbers. The sign–magnitude and one's-complement methods have a major drawback: They both have two different representations for the binary number zero, as indicated in the figure.

Two's complement does not, however, have such confusing representations. The major advantage of the two's-complement method, perhaps, is its simple implementation at the logic-level design. Microcomputers therefore use two's complement to represent n-bit signed binary numbers in the range -2^{n-1} to $+2^{n-1}-1$. For example, the range of 8-bit signed binary numbers is from -128_{10} to $+127_{10}$ in decimal and from 80_{16} to $7F_{16}$ in hexadecimal. The range of 16-bit signed binary numbers is from $-32{,}768_{10}$ to $32{,}767_{10}$ in decimal and from 8000_{16} to $8FFF_{16}$ in hexadecimal.

Signed binary numbers can be sign extended when the data structure size is increased. For example, an 8-bit signed binary number is represented in 16 bits by copying the sign bit in all the bits of the high byte. The examples in Figure 2.3 illustrate the sign extension of signed binary numbers in hexadecimal form.

Similarly, unsigned binary numbers can be zero extended when the data structure size is increased. An 8-bit unsigned binary number is represented in 16 bits by storing zero in all bits of the high byte. The examples in Figure 2.4 illustrate zero extension of unsigned binary numbers in hexadecimal form.

Binary Form	*Sign Magnitude*	*One's Complement*	*Two's Complement*
0000	+0	+0	0
0001	+1	+1	+1
0010	+2	+2	+2
0011	+3	+3	+3
0100	+4	+4	+4
0101	+5	+5	+5
0110	+6	+6	+6
0111	+7	+7	+7
1000	–0	–7	–8
1001	–1	–6	–7
1010	–2	–5	–6
1011	–3	–4	–5
1100	–4	–3	–4
1101	–5	–2	–3
1110	–6	–1	–2
1111	–7	–0	–1

Figure 2.2 Four-Bit Signed Binary Numbers in Sign–Magnitude, One's Complement, and Two's Complement

8-bit	*16-bit*	*32-bit*
82	FF82	FFFFFF82
28	0028	00000028
F5	FFF5	FFFFFFF5
5F	005F	0000005F

Figure 2.3 Sign Extension

8-bit	*16-bit*	*32-bit*
82	0082	00000082
28	0028	00000028
F5	00F5	000000F5
5F	005F	0000005F

Figure 2.4 Zero Extension

Finally, signed binary numbers can be sign contracted when the data structure size is decreased only if the number can be represented in the smaller data structure size. The examples in Figure 2.5 illustrate when signed contraction of signed binary numbers is possible and when it is not.

2.6 BINARY ARITHMETIC

2.6.1 Addition of Unsigned Numbers

Numbers that are always considered to be positive are designated *unsigned numbers*; numbers that can take up negative values are designated *signed numbers*. An addition operation using unsigned numbers is carried out pretty much like a decimal addition process. The only difference is that in binary arithmetic we use only two digits: 0 and 1. The following examples illustrate the four basic results of adding only 2 bits.

$$\begin{aligned} 0+0 &= 0 \quad \text{with} \quad \text{carry} = 0 \\ 0+1 &= 1 \quad \text{with} \quad \text{carry} = 0 \\ 1+0 &= 1 \quad \text{with} \quad \text{carry} = 0 \\ 1+1 &= 1 \quad \text{with} \quad \text{carry} = 1 \end{aligned}$$

16-bit	*8-bit*
FF92	92
0028	28
FE82	Cannot be sign contracted
0200	Cannot be sign contracted

Figure 2.5 Sign Contraction

The following examples illustrate the addition of unsigned binary numbers. Decimal addition was included for arithmetic verification.

	1110	carry		1110	carry
(+5)	0101		(+15)	1111	
+(+7)	+0111		+(+3)	+0011	
(+12)	1100		(+18)	10010	

Note that the result of the second example has 5 bits instead of 4. Therefore, we must be careful when dealing with binary arithmetic for large numbers. When implemented in circuits that have fixed number lengths, the last bit will be considered the *overflow*. When adding unsigned binary numbers, the resulting sum may be larger than the size of the internal registers of the digital system.

2.6.2 Subtraction of Unsigned Numbers

The subtraction operation is performed as an addition operation using the two's-complement method. When subtracting unsigned binary numbers, overflow never occurs; however, special attention is given to the carry from the MSB to find the result of the subtraction. If the carry from the MSB is set, the result is the correct answer and the carry is ignored. On the other hand, if the carry from the MSB is reset, the result is the two's complement of the answer.

2.7 ADDITION OF SIGNED NUMBERS

2.7.1 Addition Using the Sign–Magnitude Method

The addition of signed numbers using the sign–magnitude method is simple if the operands in the addition are of the same sign, wherein the result takes on the sign of the operands. But in case the operands have different signs, the process becomes complicated, and when used in computers it requires logic circuits to compare and subtract the numbers. Since it is possible to carry out the process without this circuitry, this method is not used in computer design.

2.7.2 Addition Using the One's-Complement Method

This method uses the simplicity of one's complement in representing the negative of a number. The process of addition using the one's-complement method may be simple or complicated, depending on the numbers being used. In certain cases, an additional correction may need to be carried out to arrive at the correct answer. The following examples illustrate one's-complement additions for four cases:

(+4)	0100	(−4)	1011
+(+2)	+0010	+(+2)	+0010
(+6)	0110	(−2)	1101

$(+4)$	0100		(-4)	1011	
$+(-2)$	$+1101$		$+(-2)$	$+1101$	
$(+2)$	[1]0001		(-6)	[1]1000	
	$\rightarrow +1$	carry		$\rightarrow +1$	carry
	0010			1001	

These examples show how a correction needs to be used in certain cases to form the result expected. The carryout from the MSB is added to the result to obtain the results expected.

2.7.3 Addition Using the Two's-Complement Method

Using the same examples as above, the two's-complement method is implemented. Addition by this method is always correct when the carryout from the sign bit is ignored. This is illustrated by examples showing four cases of addition for the same numbers from previous examples of one's-complement method addition.

$(+4)$	0100	(-4)	1100
$+(+2)$	$+0010$	$+(+2)$	$+0010$
$(+6)$	0110	(-2)	1110
$(+4)$	0100	(-4)	1100
$+(-2)$	$+1110$	$+(-2)$	$+1110$
$(+2)$	[1]0010	(-6)	[1]1010
	Ignore c_4		Ignore c_4

These examples show that correction is not necessary to find the result expected.

2.7.4 Subtraction Using the Two's-Complement Method

The process of subtraction is carried out similarly to the addition process. The two's complement of the subtrahend is computed and added to the minuend. The results desired are obtained after ignoring the carryout from the sign bit.

$(+4)$	0100	(-4)	1100
$-(-2)$	$+0010$	$-(-2)$	$+0010$
$(+6)$	0110	(-2)	1110
$(+4)$	0100	(-4)	1100
$-(+2)$	$+1110$	$-(+2)$	$+1110$
$(+2)$	[1]0010	(-6)	[1]1010
	Ignore c_4		Ignore c_4

2.7.5 Arithmetic Overflow

When the process of addition or subtraction is carried out for n-bit numbers, the result must be in the range -2^{n-1} to 2^{n-1}. If the result does not fit in this range, an overflow is said to occur. The examples that follow illustrate the various cases and the overflows in each case. Overflow can never occur when the numbers are of different signs, but if they are of the same sign, overflow can occur. There are two carryout that are essential in determining whether overflow occurs. For a 4-bit binary number, the first carryout is C_3, which is the carryout from the MSB position, and the other is C_4, the carryout from the sign bit position. It is a fact that overflows occur when the values of these carryouts are unequal. The result is correct if they have the same value. The following examples illustrate arithmetic overflow.

$$\begin{array}{rr@{\qquad}rr} (+4) & 0100 & (-4) & 1100 \\ +(+2) & \underline{+0010} & +(-2) & \underline{+1110} \\ (+6) & 0110 & (-2) & 11010 \\ c_3 = 0 & c_4 = 0 & c_3 = 1 & c_4 = 1 \end{array}$$

$$\begin{array}{rr@{\qquad}rr} (+7) & 0111 & (-7) & 1001 \\ +(+2) & \underline{+0010} & +(-2) & \underline{+1110} \\ (+9) & 1001 & (-9) & 10111 \\ c_3 = 1 & c_4 = 0 & c_3 = 0 & c_4 = 1 \end{array}$$

For n-bit binary numbers, the overflow can be expressed using the following expression:

$$\text{overflow} = c_n \oplus c_{n-1}$$

2.8 BINARY-CODED DECIMAL REPRESENTATION

Humans beings use decimal numbers in their daily arithmetic operations. Conversion from binary to decimal is not trivial for the common consumer of digital systems, such as a calculator. Digital systems must therefore allow the frequent user inputs and output to be performed in decimal form. A special number system, binary-coded decimal (BCD), has been designed to represent decimal numbers in a particular binary grouping (Figure 2.6). Digits A through F of the hexadecimal system are considered invalid binary forms in the BCD system. The BCD system has various codes, the most popular of which is the 8421 code. Other codes, such as the 5421 and the excess-3 codes, are also used in special cases.

Replacing every digit of a decimal number by its corresponding 4-bit binary code gives the BCD representation of that number. This means that only binary numbers from 0000 to 1001 occur in a system that operates using BCD representation. The other numbers are considered to be "Don't-care" conditions. Although this type of representation offers simplicity in display, its implementation for arithmetic

Decimal	*BCD 8421*	*BCD 5421*	*BCD Excess-3*
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	1000	1000
6	0110	1001	1001
7	0111	1010	1010
8	1000	1011	1011
9	1001	1100	1100

Figure 2.6 BCD 8421, BCD 5421, and BCD Excess-3 Codes

operations becomes complex and also wastes six other possible code combinations: the codes from 1010 to 1111. The BCD system makes it possible for frequency inputs and output to use the decimal system; however, the digital system still performs arithmetic operations on BCD numbers in binary form. Arithmetic operations in the BCD system may lead to invalid BCD numbers. If a resulting binary nibble is an invalid BCD digit, the binary number 0110, 6 in decimal, is added to the binary nibble and the carryout bit is propagated to the next binary nibble. On the other hand, if there is carryout from a valid BCD nibble to the next, the nibble is augmented by the binary number 0110, or decimal 6. This process is applied to all binary nibbles from right to left.

2.9 BCD ADDITION

In the case of BCD addition, the BCD number is first converted to its binary form prior to performing the addition operation. The resulting binary nibbles are converted to their corresponding BCD digits, and the arithmetic operation is then performed. The addition of two BCD numbers is complicated because of the fact that the resulting sum can be greater than 9, which means that corrections need to be applied. Let us consider two BCD numbers, represented by $U = U_3U_2U_1U_0$ and $V = V_3V_2V_1V_0$. If U + V is less than or equal to 9, the process of addition is the same as that of the binary addition of unsigned numbers. But if the sum is greater than 9, we need to add the BCD equivalent of 6 (i.e., 0110) to the first result to get the answer desired. The following examples illustrate BCD addition and the corrections required to obtain the results expected.

$(+4)$	0100		$(+9)$	1001	
$+(+7)$	$+0111$		$+(+7)$	$+0111$	
$(+11)$	1011	invalid	$(+16)$	10000	with carry
	$+0110$	$(+6)_{10}$		$+0110$	$(+6)_{10}$
	10001			10110	

Figure 2.7 shows additional examples of the addition of BCD numbers and the adjustment required to obtain the correct BCD numbers.

BCD Addition	*BCD Result in Hexadecimal*	*Adjustment*	*BCD Result Adjusted*
$(17)_{BCD}+(09)_{BCD}$	$(20)_{16}$	$(06)_{10}$	$(26)_{BCD}$
$(27)_{BCD}+(13)_{BCD}$	$(4A)_{16}$	$(06)_{10}$	$(50)_{BCD}$
$(76)_{BCD}+(30)_{BCD}$	$(A6)_{16}$	$(60)_{10}$	$(106)_{BCD}$
$(78)_{BCD}+(44)_{BCD}$	$(BC)_{16}$	$(66)_{10}$	$(122)_{BCD}$

Figure 2.7 Addition of BCD Numbers

PROBLEMS

2.1 What is the range of unsigned integers that can be represented by the following number of bits?

(a) 8
(b) 10
(c) 12
(d) 16
(e) 32
(f) 64
(g) 128

2.2 How many bits are required to represent the following unsigned integers?

(a) 255
(b) 515
(c) 1242
(d) 1978
(e) 2004
(f) 13,996
(g) 122,365
(h) 8,261,987
(i) 29,141,991

2.3 Convert the following unsigned binary numbers into decimal, octal, and hexadecimal numbers.

(a) 0.1010
(b) 0.0110
(c) 101100
(d) 111001
(e) 11000.11
(f) 11101.01

(g) 01110.101
(h) 10101.111
(i) 10110.001
(j) 11100001.1001
(k) 10101001.0101

2.4 Convert the following decimal numbers into binary, octal, and hexadecimal numbers.

(a) 127
(b) 159
(c) 789
(d) 1987
(e) 509.43
(f) 2961.72
(g) 4325.53
(h) 351.827
(i) 612.075

2.5 Convert the following hexadecimal numbers into binary, octal, and decimal numbers.

(a) 32E.15
(b) 1010.AA
(c) C0DE.02
(d) 11F8.99
(e) CAFE.45
(f) F0AE.4A
(g) EEFF.99
(h) 10EF.75
(i) BABE.01
(j) 2004.FEB

2.6 Compute the following unsigned binary arithmetic operations.

(a) $1101011 + 100111$
(b) $1011001 + 110110$
(c) $1000010 - 101010$
(d) $1100001 - 110010$
(e) $011101111 + 100011010$
(f) $100100111 + 010110011$
(g) $100111000 - 011010011$
(h) $101100001 - 011110000$

2.7 Compute the following signed binary numbers using one's-complement arithmetic operations.

(a) 011101111 + 101010001
(b) 101110000 + 111100101
(c) 011100111 + 111010011
(d) 110011000 – 110011101
(e) 011110111 – 110010011
(f) 101100001 – 011001100

2.8 Compute the following signed binary numbers using two's-complement arithmetic operations.

(a) 110111110 + 011100011
(b) 010110010 + 110011101
(c) 100100110 + 010110101
(d) 100100110 – 010110011
(e) 100111000 – 110101101
(f) 111000001 – 001110010

2.9 Compute the following signed hexadecimal arithmetic operations.

(a) 918 + 112
(b) 53F + 3A8
(c) E48 – A19
(d) 9F5 – 6E4
(e) 1EAA + F98C
(f) 9A7B + C5A0
(g) 5421 + EF9D
(h) 6842 – 7967
(i) 72E4 – 4A8C
(j) CE1E – BB09

2.10 Compute the following numbers using BCD arithmetic operations.

(a) 58 + 28
(b) 95 + 82
(c) 6389 + 7034
(d) 2380 + 1546
(e) 7020 – 1498
(f) 2004 – 3156
(g) 3084 – 8976
(h) 1144 – 1144

3 Boolean Algebra and Logic

3.1 OBJECTIVES

The objectives of the chapter are to:

- Provide an introduction to Boolean theory
- Describe logic variables and logic functions
- Define Boolean axioms and theorems
- Describe logic gates and their truth tables
- Illustrate algebraic simplifications
- Describe sum of products and product of sums
- Describe NAND and NOR equivalent circuit design

3.2 BOOLEAN THEORY

Boolean theory provides the basic fundamentals for logic operators and operations to perform Boolean algebra. *Boolean algebra* is a branch of mathematics that includes methods for manipulating logical variables and logical expressions. The Greek philosopher Aristotle founded a system of logic based on only two types of propositions: true and false. His *bivalent* (two-mode) definition of truth led to the four foundational laws of logic: the Law of Identity (A is A); the Law of Noncontradiction (A is not non-A); the Law of the Excluded Middle (either A or non-A); and the Law of Rational Inference. These "laws" function within the scope of logic where a proposition is limited to one of two possible values, but may not apply in cases where propositions can hold values other than "true" or "false."

The English mathematician George Boole (1815–1864) sought to give symbolic form to Aristotle's system of logic—hence the name Boolean algebra. Starting with his investigation of the laws of thought, Boole constructed a "logical algebra." This investigation into the nature of logic and ultimately of mathematics led subsequent mathematicians and logicians into several new fields of mathematics. Two of these, known as the "algebra of propositions" and the "algebra of classes," were based principally on Boole's work. The algebra now used in the design of logical circuitry is known as Boolean algebra.

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

In the mid-twentieth century, Claude Shannon, an electrical engineer and mathematician, applied Boole's ideas to manipulating logical expressions to the analysis of what are now called *digital circuits*, the foundation of digital electronic devices. The fact that the two distinct logical values, true and false, are also represented by 1 and 0, should hint that Boolean algebra has an application in binary systems, a fundamental feature of modern digital electronic devices.

3.3 LOGIC VARIABLES AND LOGIC FUNCTIONS

Boolean algebra is a scheme for the algebraic description of processes involved in logical thought and reasoning. Like algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. *Logic values* involve elements that take on one of two values, 0 and 1. Therefore, a logic variable can only be equal to 0 or 1. A *logic function* is an expression, that describes the logic operations between its logic variables. Similarly, a logic function can only be equal to 0 or 1.

3.4 BOOLEAN AXIOMS AND THEOREMS

The basic logic operations include logic sum, logic product, and logic complement. If a logic variable is true, its logic complement is false. The following set of logic expressions illustrates the axioms of Boolean algebra:

- $0 * 0 = 0$
- $0 + 0 = 0$
- $1 * 1 = 1$
- $1 + 1 = 1$
- $0 * 1 = 1 * 0 = 0$
- $0 + 1 = 1 + 0 = 1$
- if $x = 0$, then $\bar{x} = 1$
- if $x = 1$, then $\bar{x} = 0$

The character $(*)$ represents the AND logic product, and the character $(+)$ stands for the OR logic sum. A bar over a character represents the NOT logic. From these logic axioms, basic Boolean identities were formulated. The following expressions illustrate these identities.

Identity Property

- $x + 0 = x$
- $x * 1 = x$
- $x + 1 = 1$
- $x * 0 = 0$

Idempotent Property

- $x + x = x$
- $x * x = x$

Complement Property

- $x + \bar{x} = 1$
- $x * \bar{x} = 0$

Involution Property

- $\bar{\bar{x}} = x$

Commutative Property

- $x + y = y + x$
- $x * y = y * x$

Associative Property

- $x + (y + z) = (x + y) + z$
- $x * (y * z) = (x * y) * z$

Distributive Property

- $x * (y + z) = (x * y) + (x * z)$
- $x + (y * z) = (x + y) * (x + z)$

Absorption Property

- $x + (x * y) = x$
- $x * (x + y) = x$

Simplification Property

- $x + (\bar{x} * y) = x + y$
- $x * (\bar{x} + y) = x * y$

Consensus Theorem

- $x * y + \bar{x} * z + y * z = x * y + \bar{x} * z$
- $(x + y) * (\bar{x} + z) * (y + z) = (x + y) * (\bar{x} + z)$

DeMorgan's Theorem

- $\overline{x+y} = \bar{x} * \bar{y}$
- $\overline{x*y} = \bar{x} + \bar{y}$

In general, DeMorgan's theorem states that any logical expression remains unchanged if:

- All variables are changed to their complements
- All AND operations are changed to OR
- All OR operations are changed to AND
- The complement of the entire expression is taken

Example: $\overline{(A+B+C)*D} = \overline{(A+B+C)} + \bar{D} = \bar{A} * \bar{B} * \bar{C} + \bar{D}$

3.5 BASIC LOGIC GATES AND TRUTH TABLES

Logic expressions describe an output as a function of the input and are called *logic functions*. In the digital world, logic gates are used to implement logic functions. There are seven types of logic gates: NOT, AND, OR, NAND, NOR, XOR, and NXOR. These circuit diagrams are symbolic representations of their corresponding logic functions (Figures 3.1 and 3.2).

The logic gates of AND, OR, XOR, NAND, NOR, and NXOR may have more than two inputs. The circuits in Figure 3.3 illustrate multi-input AND and OR logic gates.

3.6 LOGIC REPRESENTATIONS AND CIRCUIT DESIGN

Logic gates are used to represent and implement logic expression into digital circuit diagrams. Consider the following logic function:

$$f(x,y,z) = x \cdot y \cdot z + \bar{x} \cdot \bar{z}$$

The digital circuit (Figure 3.4) implements the function *f*. The digital circuit consists of elementary logic functions, which are implicit in the logic function. The conversion from logic expressions to circuit diagrams obeys the operator precedence rules shown in Figure 3.5. These precedence rules dictate the order in which the implicit elementary logic functions are implemented.

Notice that logic expressions within parentheses are converted first. The implementation of the elementary logic functions within the parentheses obeys the precedence rules as well. In all cases, the NOT function is implemented first, then the AND function, then the OR function. Similarly, a digital circuit can be converted to a logic expression. The logic function of the digital circuit is found by propagating

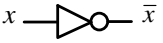
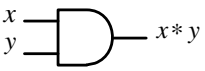
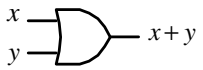
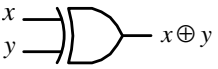
Truth Table	*Function*	*Symbol*
x | $\bar{x}$ 0 | 1 1 | 0	NOT	
x | y | $x*y$ 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1	AND	
x | y | $x+y$ 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1	OR	
x | y | $x \oplus y$ 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0	XOR	

Figure 3.1 NOT, AND, OR, and XOR Gates

the input variables through the gates to the output of the circuit. The logic output expression of a gate is determined by the logic expressions at its inputs. Consider the digital circuit shown in Figure 3.6. The logic expression, which represents the logic circuit diagram in Figure 3.6, is expressed as

$$f(x, y, z) = (x \cdot \bar{y}) \oplus (y + z)$$

The XOR function is not listed in Figure 3.5, but since the XOR function is a composite logic function consisting of two AND functions and one OR function, it has the same level of precedence as the AND function.

3.7 TRUTH TABLE

A truth table is generally the first design step. The designer begins with a word statement that describes the function of a digital system. Next, he or she identifies the inputs and outputs of the system and draws a truth table. The inputs and outputs may be single bits or a collection of bits. The truth table consists of input and output

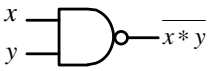
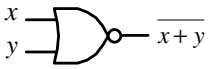
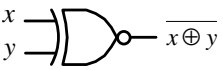
Truth Table			*Function*	*Symbol*
x	y	$\overline{x * y}$	NAND	
0	0	1		
0	1	1		
1	0	1		
1	1	0		
x	y	$\overline{x + y}$	NOR	
0	0	1		
0	1	0		
1	0	0		
1	1	0		
x	y	$\overline{x \oplus y}$	NXOR	
0	0	1		
0	1	0		
1	0	0		
1	1	1		

Figure 3.2 NAND, NOR, and NXOR Gates

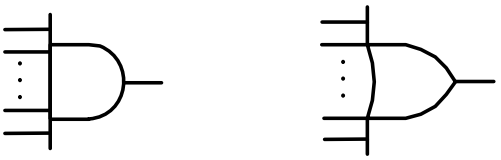


Figure 3.3 Multi-input AND and OR Logic Gates

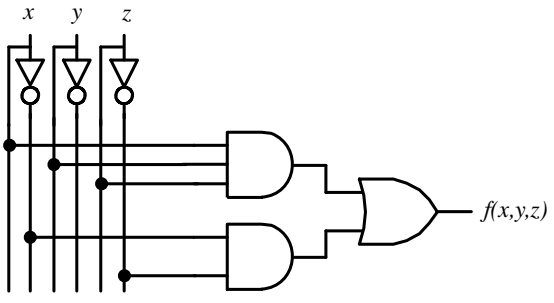


Figure 3.4 Conversion from Logic Function to Circuit Diagram

Precedence Order	*Algebra*	*Boolean Algebra*
First	*Parentheses*	*Parentheses*
Second	*Exponent*	*NOT*
Third	*Multiplication/division*	*AND*
Last	*Addition*	*OR*

Figure 3.5 Precedence Rules for Elementary Logic Function Conversion

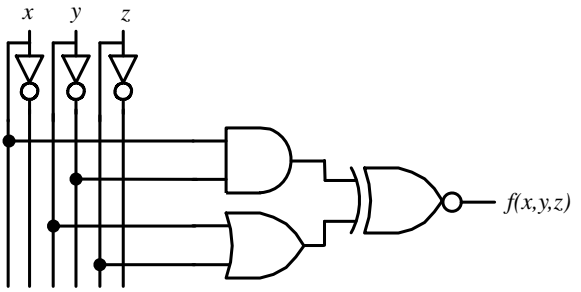


Figure 3.6 Conversion from Circuit Diagram to Logic Expression

columns, which characterize the function of the digital circuit. The input columns consist of all possible combinations of inputs. The maximum number of all possible combinations of inputs is 2^n, where n is the number of inputs.

Consider the truth table in Figure 3.7. The digital circuit described by this truth table has three inputs (single bit) and one output (single bit). Since there are three input variables, the maximum combinations of inputs possible is eight. Although one could list the possible combinations of inputs randomly, it is generally strongly recommended to use the pattern shown in Figure 3.7. Notice that the right most input column changes every row, the next input column every two rows, and the next input column every four rows. A fourth input column would change every eight rows, and so on. The truth table in Figure 3.7 actually represents Figure 3.4. The process of designing a digital circuit from a truth table is described in Section 3.9.

Row	*x*	*y*	*z*	*f*
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Figure 3.7 Truth Table

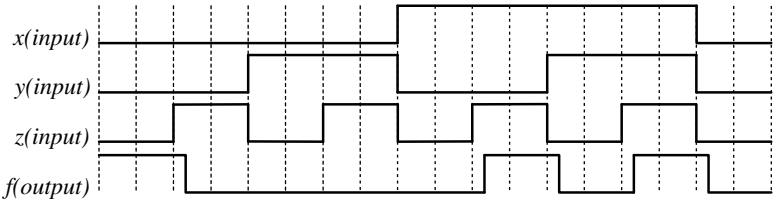


Figure 3.8 Timing Diagram

3.8 TIMING DIAGRAM

A timing diagram is the graphical representation of input and output signals as functions of time. Since the inputs and outputs can only take the values 0 or 1, their graphical representations are series of square pulses with a variety of time lengths. The inputs and outputs are drawn on the same diagram to show the input–output behavior of the digital system. A timing diagram is usually generated by an oscilloscope or logic analyzer. Computer-aided design tools have software simulator that generate timing diagrams. A timing diagram shows all possible input and output patterns, not necessarily in an order similar to that of a truth table.

Consider the timing diagram in Figure 3.8. Notice that the time intervals are equally separated. Notice also that the output transitions do not occur at exactly the same time that the input transitions occur, but a very short time later. The delay in the output transitions, referred to as the *propagation delay*, is the time difference between the time of input application and the time when the outputs become valid. The propagation delay is a real physical effect of electronic components that make a logic gate or a circuit. Timing diagrams should show propagation delays. However, during the initial design of a logic circuit, the actual circuit components are not well defined, and therefore any propagation delay can only be estimated. Propagation delays are explored further in Chapter 5.

3.9 LOGIC DESIGN CONCEPTS

If a function is specified in the form of a truth table, an expression that realizes the function can be obtained by considering the rows in the table for which the function is equal to 1 or 0, called the *sum-of-products* and the *product-of-sums,* respectively. For a function of n variables, a product term in which each of the n variables appears once is called a *minterm*. For each row of the truth table, a minterm is formed by the product of the variables (if equal to 1) or their complements (if equal to 0). Similarly, for each row of the truth table, a *maxterm* is formed by the sum of the variables (if equal to 0) or their complements (if equal to 1). The construction of minterms and maxterms for a logic function is independent of its output. This concept of minterm and maxterm evaluation is illustrated in Figure 3.9, where the rows have been numbered 0 through 7 for reference. All possible combinations of the inputs for a three-variable minterm and maxterms are shown in the figure. The first row, row 0, shows $x = y = z = 0$, which

Row	*x*	*y*	*z*	*Minterms*	*Maxterms*
0	0	0	0	$m_0 = \bar{x} \cdot \bar{y} \cdot \bar{z}$	$M_0 = x + y + z$
1	0	0	1	$m_1 = \bar{x} \cdot \bar{y} \cdot z$	$M_1 = x + y + \bar{z}$
2	0	1	0	$m_2 = \bar{x} \cdot y \cdot \bar{z}$	$M_2 = x + \bar{y} + z$
3	0	1	1	$m_3 = \bar{x} \cdot y \cdot z$	$M_3 = x + \bar{y} + \bar{z}$
4	1	0	0	$m_4 = x \cdot \bar{y} \cdot \bar{z}$	$M_4 = \bar{x} + y + z$
5	1	0	1	$m_5 = x \cdot \bar{y} \cdot z$	$M_5 = \bar{x} + y + \bar{z}$
6	1	1	0	$m_6 = x \cdot y \cdot \bar{z}$	$M_6 = \bar{x} + \bar{y} + z$
7	1	1	1	$m_7 = x \cdot y \cdot z$	$M_7 = \bar{x} + \bar{y} + \bar{z}$

Figure 3.9 Minterms and Maxterms for Three Variables

has a corresponding minterm represented by $\bar{x} * \bar{y} * \bar{z}$ and a corresponding maxterm represented by $x + y + z$. To further simplify reference to individual minterms and maxterms, they are identified by an index that corresponds to the row numbers. For example, the minterm for row 0 will be referred to as m_0, and the maxterm for the same row will be referred to as M_0.

3.10 SUM-OF-PRODUCTS DESIGN

A function can be represented by an expression that is a sum of minterms only, where each minterm is ANDed with every other minterm to represent the function when it is equal to 1. The resulting implementation is functionally correct and unique but not necessarily the lowest-cost realization of the function. The sum of products (SOP) is a logic expression consisting of product (AND) terms that are summed (ORed) with each other. If each product term is a minterm, the expression is called a *canonical* sum of products for the function. For example, consider the truth table in Figure 3.10 of a logic function *f* of three variables. Using the minterms for which the function is equal to 1, the function can be written explicitly as follows:

$$f(x, y, z) = \bar{x} \cdot \bar{y} \cdot z + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z}$$

Row	*x*	*y*	*z*	*f*
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 3.10 Truth Table for the Logic Function *f*

Through the use of Boolean algebra identities, this expression can be simplified algebraically as follows:

$$\begin{aligned} f(x,y,z) &= \bar{x}\cdot\bar{y}\cdot z + x\cdot\bar{y}\cdot\bar{z} + x\cdot\bar{y}\cdot z + x\cdot y\cdot\bar{z} \\ &= (\bar{x}+x)\cdot\bar{y}\cdot z + x\cdot(\bar{y}+y)\cdot\bar{z} \\ &= 1\cdot\bar{y}\cdot z + x\cdot 1\cdot\bar{z} \\ &= \bar{y}\cdot z + x\cdot\bar{z} \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. The cost of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit. Minterms, given their row-number subscripts, can be used to specify a given function in a more concise form. The logic function can also be expressed as

$$f(x,y,z) = \sum(m_1, m_4, m_5, m_6)$$

or as

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

The symbol $\sum$ represents the logical sum (OR) operation.

3.11 PRODUCT-OF-SUMS DESIGN

Contrary to minterms, which represent the product of variables that set the function to 1, the function can also be represented by the sum of variables, which set the function to 0. Variables used to represent the function using the complement to minterms are called *maxterms*. All possible maxterms for three-variable functions are listed in Figure 3.9. Consider the function specified by a truth table in Figure 3.10; its complement function can be represented by a sum of minterms for which the function is equal to 0. For example, the complement of function f can be represented as

$$\begin{aligned} \bar{f}(x,y,z) &= \bar{x}\cdot\bar{y}\cdot\bar{z} + \bar{x}\cdot y\cdot z + \bar{x}\cdot y\cdot\bar{z} + x\cdot y\cdot z \\ &= m_0 + m_2 + m_3 + m_7 \end{aligned}$$

Using DeMorgan's theorem, the function f can be represented as

$$\begin{aligned} f(x,y,z) &= \overline{m_0 + m_2 + m_3 + m_7} \\ &= \bar{m}_0\cdot\bar{m}_2\cdot\bar{m}_3\cdot\bar{m}_7 \\ &= M_0\cdot M_2\cdot M_3\cdot M_7 \\ &= (x+y+z)\cdot(x+\bar{y}+z)\cdot(x+\bar{y}+\bar{z})\cdot(\bar{x}+\bar{y}+\bar{z}) \end{aligned}$$

Using Boolean algebra identities, the function can be reduced:

$$\begin{aligned} f(x,y,z) &= (x+y+z)\cdot(x+\bar{y}+z)\cdot(x+\bar{y}+\bar{z})\cdot(\bar{x}+\bar{y}+\bar{z}) \\ &= [(x+z)+y]\cdot[(x+z)+\bar{y}]\cdot[x+(\bar{y}+\bar{z})]\cdot[\bar{x}+(\bar{y}+\bar{z})] \\ &= (x+z)\cdot(\bar{y}+\bar{z}) \end{aligned}$$

Using the shorthand method to express the function for the product of sums yields

$$\begin{aligned} f(x,y,z) &= \prod(M_0, M_2, M_3, M_7) \\ &= \prod M(0,2,3,7) \end{aligned}$$

The symbol $\prod$ represents the logical product (AND) operation. More algebraic manipulations and simplifications are explored in the Karnaugh mapping sections in Chapter 6.

3.12 DESIGN EXAMPLES

3.12.1 Multiplexer

A *multiplexer* is a combinatorial circuit that has a number (usually, a power of 2) of *data inputs* (2^n) and n *select inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the data input selected. Now let us consider a *2 : 1 multiplexer*. As the name indicates, it has two data inputs, x_1 and x_2, a select input, *s,* and a single output, y. The output of the circuit will be same as the value of input x_1 if $s = 0$; otherwise the output will be equal to x_2. We can construct a truth table based on these requirements. The truth table of a 2 : 1 multiplexer is shown in Figure 3.11.

From the truth table we can derive the logical expression for the output y:

$$y(s, x_2, x_1) = \bar{s}\cdot\bar{x}_2\cdot x_1 + \bar{s}\cdot x_2\cdot x_1 + s\cdot x_2\cdot\bar{x}_1 + s\cdot x_2\cdot x_1$$

After simplification, the expression is reduced to

$$\begin{aligned} y(s, x_2, x_1) &= [\bar{s}\cdot x_1\cdot(\bar{x}_2 + x_2)] + [s\cdot x_2\cdot(\bar{x}_1\cdot x_1)] \\ &= \bar{s}\cdot x_1 + s\cdot x_2 \end{aligned}$$

s	x_2	x_1	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 3.11 Truth Table of a 2 : 1 Multiplexer

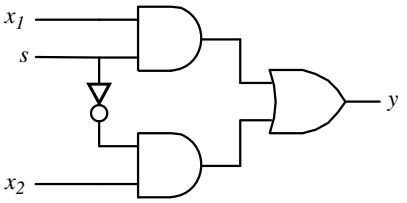


Figure 3.12 Logic Diagram of a 2 : 1 Multiplexer

which can be realized using one OR gate and two AND gates. The logic diagram for a 2 : 1 multiplexer is shown in Figure 3.12.

3.12.2 Half-Adder

The half-adder is an example of a simple functional digital circuit built from two logic gates. A half-adder adds two 1-bit binary numbers, x and y. The output is the sum of the two bits s and the carry c_{out}. The truth table for a 1-bit half-adder is shown in Figure 3.13. The logical expressions for the outputs sum s and carry c_{out} are as follows:

$$s(x, y) = \bar{x} \cdot y + x \cdot \bar{y} = x \oplus y$$
$$c_{\text{out}}(x, y) = x \cdot y$$

These logical expressions can be realized using one XOR gate and one AND gate. The circuit diagram of a 1-bit half-adder is shown in Figure 3.14. The half-adder does not take into account the carry-in from another half-adder. In Chapter 7 we explore full-adder circuits, which can be used to implement the addition of numbers with larger bit sizes.

x	y	c_{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 3.13 Truth Table of a 1-Bit Half-Adder

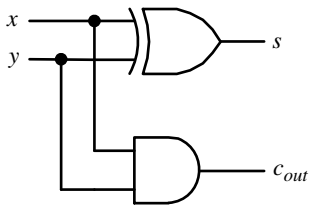


Figure 3.14 Logic Diagram of a 1-Bit Half-Adder

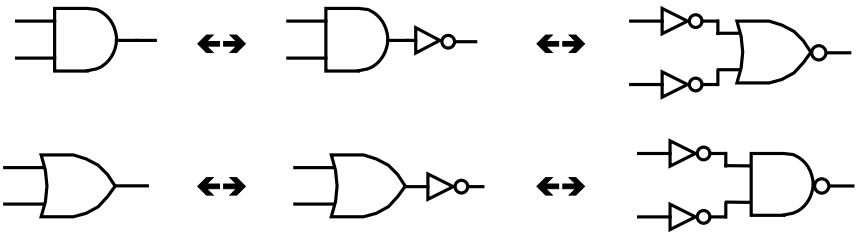


Figure 3.15 AND and OR Equivalent Logic Gates

3.13 NAND AND NOR EQUIVALENT CIRCUIT DESIGN

Through the use of Boolean algebra, it is possible to convert an AND to an OR by inverting the inputs or outputs (DeMorgan's theorem). The same condition holds true for the logical gates NAND and NOR. Figure 3.15 shows equivalencies for AND and OR. Using DeMorgan's theorem, one can generate equivalent logic gates for NAND and NOR gates, as shown in Figure 3.16. The circles at the inputs of the AND and OR gates in Figures 3.15 and 3.16 represent inverters. These inverters are referred to as *invert bubbles*. Because of the gate equivalency, any logic circuit implemented with NOT, AND, and OR gates could be converted to a logic circuit containing only NAND gates or only NOR gates. This conversion is practical when only one type of gate (NAND or NOR) is available to the designer. In addition, fewer integrated circuits are needed when implementing a logic circuit with NAND or NOR gates.

Often, the final logic circuit is implemented with only NAND gates or only NOR gates. In practice, one usually draws the logic circuit using NOT, AND, and OR gates, then implements a graphical conversion of the gates. During the graphical conversion, invert bubbles are inserted to modify AND and OR gates to NAND or NOR gates, as illustrated in Figure 3.15. Each bubble inserted must be compensated for on the same connecting line. Two consecutive invert bubbles cancel each other out. Single bubbles not canceled must be replaced by inverters. An inverter is replaced by a NAND or a NOR gate by connecting both inputs. Consider the logic circuit in Figure 3.17. Its equivalent NAND-only circuit is shown in Figure 3.18 and its equivalent NOR-only circuit is shown in Figure 3.19.

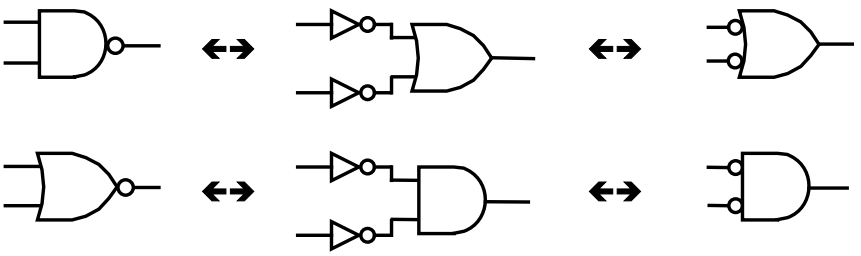


Figure 3.16 NAND and NOR Equivalent Logic Gates

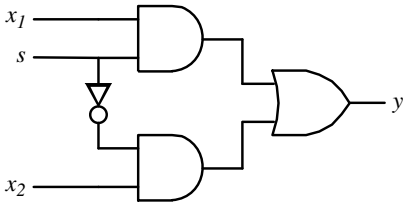


Figure 3.17 Logic Circuit Sample Using NOT, AND, and OR Gates

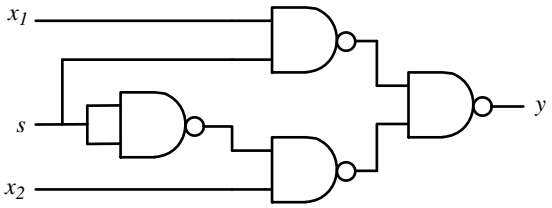


Figure 3.18 Equivalent NAND Circuit of the Logic Circuit in Figure 3.17

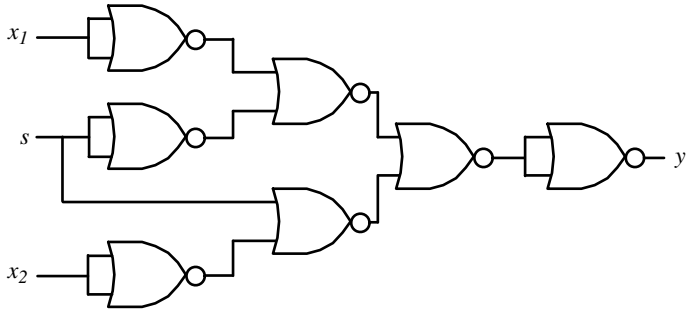


Figure 3.19 Equivalent NOR Circuit of the Logic Circuit in Figure 3.17

3.14 STANDARD LOGIC INTEGRATED CIRCUITS

An approach used widely until the mid-1980s was to connect multiple chips, each containing only a few logic gates. *Standard chips* are integrated circuits that are mass-produced to perform commonly needed electronic functions. They have proven to be very useful in circuit design over the years. Engineers and technicians can connect these components to parts of a larger circuit to accomplish a specific function. Some of the more common uses for standard chips are logic gates: AND, OR, NOT, NAND, NOR, XOR, and XNOR, a wide variety of which are available with different types of logic. Many of these have been created as chips in the 7400 or 4000 series (these part numbers are printed on the top of the chip casing). The external connections of the chip are called *pins* or *leads*. Two pins are used to connect V_{CC} and Gnd, which power the chip. The other pins are used to connect to the inputs and outputs of each individual gate.

Three main types of technology are used to build 7400 series chips. Chips that bear codes beginning with 74LS represent some of the older designs. They are made with TTL (transistor–transistor logic) and have a maximum frequency of about 35 MHz. Their input voltage is always around 5 V, and one of their outputs can drive up to ten 74LS inputs or fifty 74HCT inputs. This driving principle is called *fan-out*. 74HCT chips are compatible with TTL but are created using the technology known as CMOS (complementary metal–oxide semiconductor). These chips have an input voltage of around 5 V. Their maximum frequency is 25 MHz, and one output can drive up to fifty 74HCT, HC, or other CMOS inputs, but only ten 74LS inputs. The 74HC series chips use high-speed CMOS technology and have input voltages between 2 and 6 V and a maximum frequency of 25 MHz. The 4000 series chips are made using CMOS technology, but their maximum frequency is only about 1 MHz. Their input voltage can range from 3 to 15 V, and one output can drive up to fifty CMOS, 74HCT, or 74HC inputs, but only one 74LS input. The power consumed by the chip itself, whether in the 4000 or 7400 series, is only a few microwatts. Figure 3.20 lists some common standard chips.

The actual function of each type is described in a *datasheet*, comprising instructions that explain how an IC functions. Power requirements, the truth table, and IC characteristics are just a few examples of information that may be found on a datasheet. Generally, the first page of a datasheet contains the part number, the gate type of the IC, the schematic, and the truth table for the IC. Some important operating conditions include:

- V_{CC}: supply voltage required to power the IC
- V_{IHMIN}: minimum input voltage required for the IC recognize a "high input value," or 1
- V_{ILMAX}: maximum input voltage required for the IC to recognize a "low input value, or 0
- t_{PLH} (propagation delay time): Time required for the IC to switch from a low-level output to a high-level output
- t_{PHL} (propagation delay time): Time required for the IC to switch from a high-level output to a low-level output

7400 Series	*4000 Series*
7400 - Quad 2-input NAND Gate	4001 - quad 2-input NOR gate
7402 - Quad 2-input NOR Gate	4012 - dual 4-input NAND gate
7404 - Hex Inverter	4049 - hex NOT gate
7408 - Quad 2-input AND Gate	4070 - quad 2-input XOR gate
7414 - Hex Schmitt-Trigger Inverter	4071 - quad 2-input OR gate
7420 - Dual 4-input NAND Gate	4072 - dual 4-input OR gate
7421 - Dual 4-input AND Gate	4077 - quad 2-input XNOR gate
7432 - Quad 2-input OR gate	4081 - quad 2-input AND gate
7486 - Quad 2-input XOR Gate	4082 - dual 4-input AND gate

Figure 3.20 Partial List of Common Standard Logic Chips

PROBLEMS

3.1 Verify the following Boolean algebraic equations.

(a) $(x + xy = x$

(b) $x + \bar{x}y = x + y$

(c) $x(x + y) = x$

(d) $x(\bar{x} + y) = xy$

(e) $(x + y)(x + \bar{y}) = x$

(f) $xy + x\bar{y} = x$

(g) $(x + y)(x + z) = x + yz$

(h) $xz + yz + \bar{x}y = x\bar{y}z + xz + \bar{x}y\bar{z}$

(i) $\bar{x} \cdot \bar{y} + \bar{y} \cdot \bar{z} + x\bar{y}z = \bar{y}$

3.2 Verify the following Boolean algebraic equation using truth tables.

(a) $x(y + z) = xy + xz$

(b) $\overline{xyz} = \bar{x} + \bar{y} + \bar{z}$

(c) $\overline{x + y + z} = \bar{x} \cdot \bar{y} \cdot \bar{z}$

(d) $x \oplus y = x\bar{y} + \bar{x}y$

(e) $\overline{x \oplus y} = xy + \bar{x} \cdot \bar{y}$

3.3 Using algebraic manipulation, simplify the logic functions described in Figure P3.3(a) to (d) and draw their corresponding logic circuits.

(a)

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(b)

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(c)

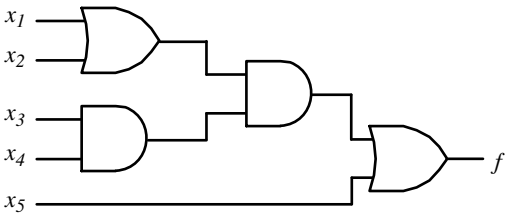
x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

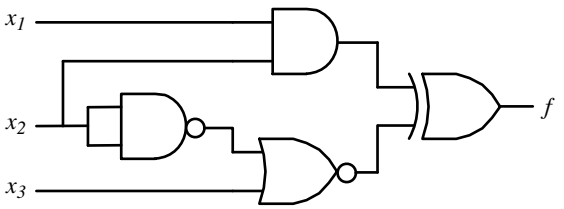
x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figure P3.3

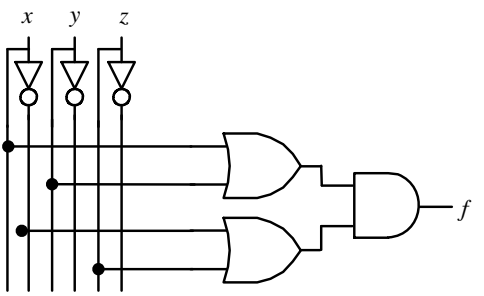
3.4 Find the logic function(s) for the logic circuits in Figure P3.4(a) to (d).



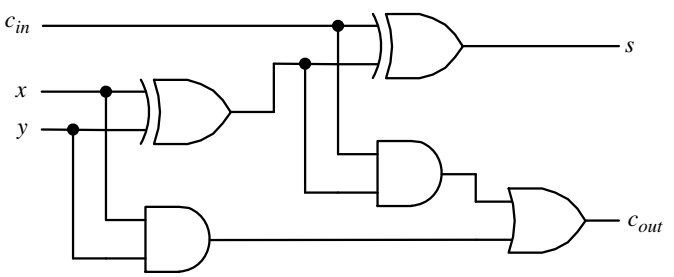
(a)



(b)



(c)



(d)

Figure P3.4

3.5 Using algebraic manipulations, simplify the following logic functions and draw their corresponding logic circuits.

(a) $f(x_1, x_2) = (x_1 + x_2)(\bar{x}_1 + x_2)(x_1 + \bar{x}_2)$

(b) $f(x_1, x_2) = \overline{\overline{x_1 x_2} + (x_1 + x_2)}$

(c) $f(x_1, x_2) = (\bar{x}_1 + \bar{x}_2)\overline{x_1 + x_2}$

(d) $f(x_1, x_2, x_3) = x_1\bar{x}_2 + x_1\bar{x}_2 x_3$

(e) $f(x_1, x_2, x_3) = x_1 + x_1\bar{x}_2 + x_1\bar{x}_2 x_3$

(f) $f(x_1, x_2, x_3) = x_1 x_2 x_3 + x_1 x_2\bar{x}_3 + \bar{x}_1 x_2\bar{x}_3$

(g) $f(x_1, x_2, x_3) = (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3)$

(h) $f(x_1, x_2, x_3, x_4) = \overline{x_1 + \bar{x}_2 + \bar{x}_3 + x_4} + \bar{x}_1 x_2\bar{x}_3 x_4$

(i) $f(x_1, x_2, x_3, x_4) = x_1 x_4 + \bar{x}_1 x_4 + x_2\bar{x}_3 + x_1 x_2 x_3 + \bar{x}_1 x_2 x_3$

(j) $f(x_1, x_2, x_3, x_4) = \bar{x}_1 x_4 + x_2 x_4 + \bar{x}_1 x_2 x_4 + x_1 x_2 x_3$

3.6 Write the truth tables, simplify the logic functions *f(x,y,z)*, and draw the logic circuits described by the timing diagrams in Figure P3.6(a) to (d). Ignore propagation delays.

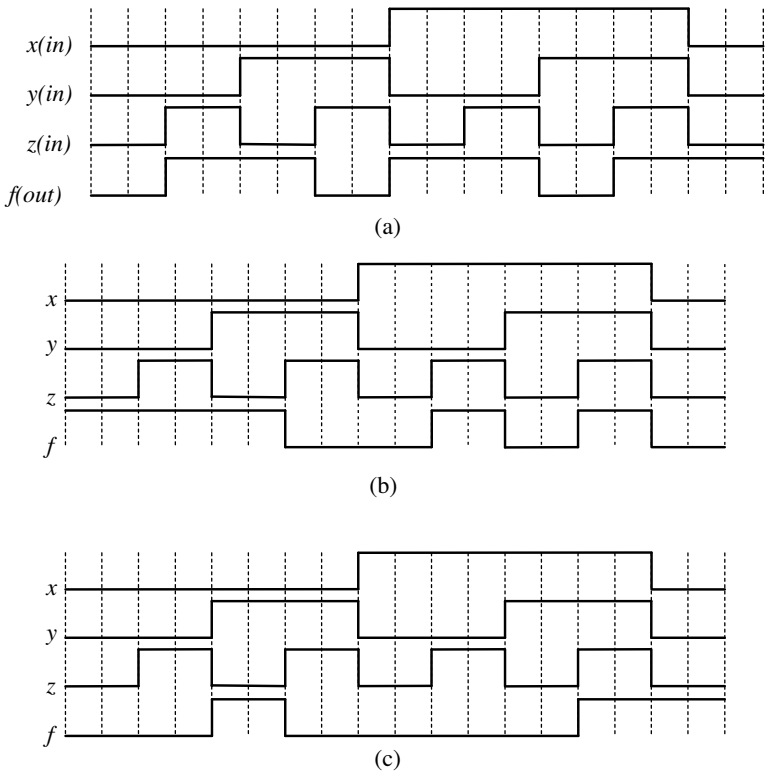
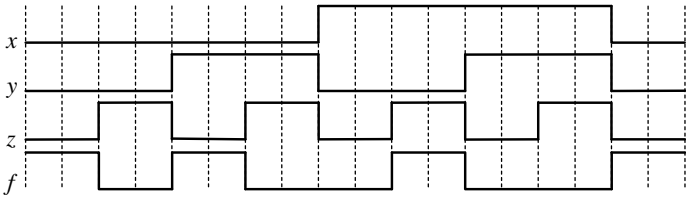


Figure P3.6



(d)

Figure P3.6 (*Continued*)

3.7 Using algebraic manipulations, simplify the following implicit SOP logic functions and draw their corresponding logic circuits.

(a) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_7)$

(b) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_3, m_5)$

(c) $f(x_1, x_2, x_3, x_4) = \sum(m_0, m_2, m_6, m_8, m_{10}, m_{14})$

(d) $f(x_1, x_2, x_3, x_4) = \sum(m_5, m_6, m_7, m_{13}, m_{14}, m_{15})$

(e) $f(x_1, x_2, x_3, x_4) = \sum(m_0, m_1, m_4, m_5, m_8, m_9, m_{14}, m_{15})$

3.8 Using algebraic manipulations, simplify the following implicit POS logic functions and draw their corresponding logic circuits.

(a) $f(x_1, x_2, x_3) = \Pi(M_3, M_4, M_6)$

(b) $f(x_1, x_2, x_3) = \Pi(M_2, M_4, M_5, M_6)$

(c) $f(x_1, x_2, x_3, x_4) = \Pi(M_1, M_3, M_4, M_6, M_9, M_{11})$

(d) $f(x_1, x_2, x_3, x_4) = \Pi(M_3, M_6, M_7, M_{11}, M_{13}, M_{15})$

(e) $f(x_1, x_2, x_3, x_4) = \Pi(M_0, M_1, M_5, M_8, M_9, M_{13}, M_{15})$

3.9 Draw the timing diagrams of the following logic functions.

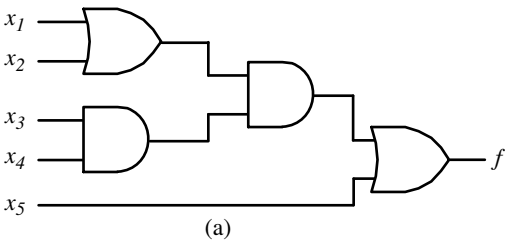
(a) $f(x_1, x_2) = (\bar{x}_1 + \bar{x}_2)(x_1 + \bar{x}_2)(\bar{x}_1 + x_2)$

(b) $f(x_1, x_2) = (x_1 + x_2)\overline{\bar{x}_1 + \bar{x}_2}$

(c) $f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3$

(d) $f(x_1, x_2, x_3) = (\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + x_3)$

3.10 Draw the equivalent logic circuits of the circuits in Figure P3.10(a) to (e) using NAND gates only.



(a)

Figure P3.10

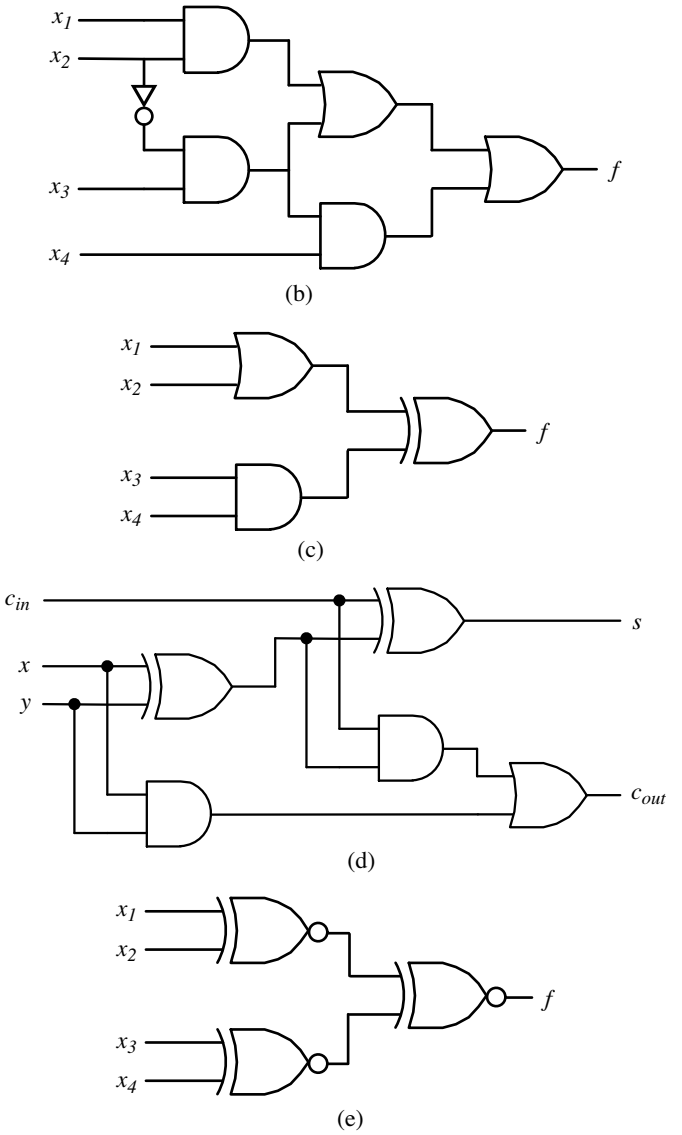


Figure P3.10 (*Continued*)

3.11 Draw the equivalent logic circuits for the circuits in Figure P3.10(a) to (e) using NOR gates only.

3.12 Consider a logic function f with the three variables x_1, x_2, and x_3. The function f is equal to 1 if and only if two variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

3.13 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any odd number of variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

3.14 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any even number of variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

3.15 Consider a logic function f with the three variables x_1, x_2, and x_3. The function f is equal to 1 if $x_1 = 1$ and $x_2 = x_3$ or if $x_1 = 0$ and $x_2 \neq x_3$; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

3.16 Consider a logic function f with the variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any two variables or more are equal to 1; otherwise, the function f is equal to zero, with the following exception: The function f is also equal to zero if x_1 is equal to zero and any other two variables are equal to 1.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

3.17 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if two or more variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using algebraic manipulations, simplify the function f.

(c) Draw a logic circuit that implements the function f.

(d) Draw a timing diagram for the function f.

4 VHDL Design Concepts

4.1 OBJECTIVES

The objectives of the chapter are to:

- Describe VHDL language programming
- Describe CAD tools for digital system design
- Describe hardware description languages
- Provide a history of hardware description languages
- Define VHDL
- Describe the VHDL programming structure
- Define **entity**
- Define **architecture**
- Describe VHDL data types
- Describe VHDL operators
- Describe **Signal** and **generate** statements
- Describe sequential statements
- Describe loops and decision-making statements
- Describe Subcircuit design
- Describe **package** and **component** statements

4.2 CAD TOOL–BASED LOGIC DESIGN

Today's digital integrated circuits are designed to perform highly complex functions. Their design would be very challenging (and nearly impossible) if it were not for the availability of the numerous CAD (computer-aided design) tools used for the following tasks: design entry, simulation, synthesis and optimization, and physical design. Initially, the *specifications* of the system to be designed are derived from the *requirements*.

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

1. ***Design entry***. Design entry is the first step in the design process using CAD tools. A designer describes the circuit to be implemented using some design-entry method provided by the tools. The most common design-entry method is to write a hardware description language program in VHDL or Verilog, which can be *compiled* using CAD tools to simulate the design, *synthesize*, *optimize*, and generate the circuit implementation. Other methods of design entry include schematic entry (the circuit is drawn using symbols from a library supported by the tool), and entry using truth tables (where inputs and corresponding outputs are described using a truth table in a text file).
2. ***Simulation***. Once the design entry is completed, the design is tested for correct functionality using *simulation*. The CAD tool that performs this job is called a *functional simulator*. For simulation, the user has to supply to the simulator "test vectors" for the inputs and expected outputs. Simulation is also performed after *synthesis*, which is the process of translating the design entered (at the design-entry phase) to a physically realizable circuit using logic gates (in the ASIC design flow) or logic blocks inside a PAL/PLA/FPGA. Postsynthesis simulation involves functional testing as well as tests to ensure that the timing constraints of that the circuit are met.
3. ***Synthesis***. A synthesis tool is used to translate the design described in a design-entry method (usually, a program in VHDL or Verilog) into a physically realizable circuit. The same tool is also used to optimize the circuit.
4. ***Place and Route*** (*PAR*). This is the final step in the design process before actual hardware implementation of a digital integrated circuit (IC). PAR, also known as the *physical design* phase, is where the gates are placed and interconnected (*routing*) to complete the circuit. A final *timing simulation* is performed after PAR to make sure that the circuit meets timing constraints when the *parasitic capacitances* due to the transistors and interconnecting metal wires are added.

4.3 HARDWARE DESCRIPTION LANGUAGES

A *hardware description language* (HDL) is a programming language used to describe the *behavior* or *structure* of digital circuits (ICs). HDLs are also used to stimulate the circuit and check its response. Many HDLs are available, but VHDL and Verilog are by far the most popular. Most CAD tools available in the market support these languages. VHDL stands for "very high-speed integrated-circuit hardware description language." Both VHDL and Verilog are officially endorsed IEEE (Institute of Electrical and Electronics Engineers) standards. Other HDLs include JHDL (Java HDL), and proprietary HDLs such as Cypress Semiconductor Corporation's Active-HDL.

In the 1980s, the rapid advances in IC technology necessitated a need to standardize design practices. In 1983, VHDL was developed under the VHSIC program of the U.S. Department of Defense. It was originally intended to serve as

a language to document descriptions of complex digital circuits. It was also used to describe the behavior of digital circuits and could be fed to software tools that were used to simulate a circuit's operation. In 1987, IEEE adopted the VHDL language as standard 1076 (also referred to, as VHDL-87). It was revised in 1993 as the standard VHDL-93. Verilog HDL and a simulator were released by Gateway Design Automation in 1983. In 1989, Cadence Design Systems acquired Gateway Design Automation. In 1990, Cadence separated the HDL from its simulator (Verilog-XL) and released the HDL into the public domain. Verilog HDL is guarded by the Open-Verilog International Organization, now part of Accelera Organization. In 1995, IEEE adopted Verilog HDL as standard 1364.

Hardware description languages, including VHDL, are used to program PLD- and FPGA-based systems. The Altera and Xilinx corporations provide free limited versions (for educational purposes) of CAD software and tools, which can be used to program FPGA-based development boards. The CAD tools include a schematic editor, a VHDL/Verilog editor, compilers, libraries, design simulators, and various utilities tools.

4.4 VHDL LANGUAGE

VHDL ["VHSIC hardware description language" (VHSIC is "very high-speed integrated circuit")] is one of the two most popular HDLs, the other being Verilog HDL. The VHDL language is very popular for the design entry of digital circuits into CAD systems, simulation and documentation. VHDL is an extremely complex and sophisticated language, so learning it completely can be a daunting task. However, most designs can be accomplished by learning only a subset of the language. The material presented here is in no way a comprehensive reference for the VHDL language; it only introduces the reader to some important and fundamental concepts of programming with VHDL. More details of the language can be obtained from VHDL-specific textbooks, Web-based tutorials, and published white papers.

4.5 VHDL PROGRAMMING STRUCTURE

A VHDL program is written in a text file and has the extension ".vhd" (sometimes, ".vhdl"). VHDL is not case sensitive. Every VHDL program has associated with it an **entity**. The interface to the outside world (through pins) is described in this section. Every **entity** has associated with it an **architecture**. The **architecture** describes the *behavior* or *structure* of the design coded in the VHDL program. The *design units* of VHDL, apart from **entity** and **architecture**, are **package**, **package** body, and configuration. These are not *required* to be present in every VHDL design, but designers use them for a better coding style and for convenience. Each design unit of a VHDL design can be in a separate file. It is not required that an **entity** and the corresponding **architecture** be described in the same file. It is important to note that a single IC can be built from many VHDL files. For

{*Library Declaration*}
{*Entity Declaration*}
{*Architecture Declaration*}

Figure 4.1 VHDL Program Structure

example, a design may be built hierarchically. A multiplier may be built from full-adders, which in turn may be built from half-adders. One could write VHDL programs for the half-adder, put two *instances* (copies) of the half-adder in another VHDL program to build a full-adder, and write a third program to put full-adders together to make a multiplier. This is an example of hierarchical design (bottom-up design). VHDL programming is not case sensitive; however, to identify the keywords of a VHDL program, they will be written in lowercase boldface letters.

A typical VHDL program consists of a **library** declaration, an **entity** declaration, and an **architecture** declaration, as illustrated in Figure 4.1. **Library** declarations are generally included in the first lines of the code, which locate the system library and user library sources to resolve and translate the language statements within the body of the program. The IEEE standard library is often included in the VHDL program and used by the VHDL compiler to translate standard inputs, outputs, and expressions listed in the VHDL program. Other references, which are not listed in the IEEE standard library, must be provided by the user's libraries.

4.5.1 Entity

The **entity** of a VHDL program defines the external interface to the design. The term **entity** is a keyword and cannot be used as a variable in the VHDL program. The **entity** defines the input and output ports of a digital system. Every VHDL program must have at least one **entity** with a designated name. The **entity** declaration associates the entity with a particular design, and specifies the names of the ports, their associated data type, and the direction of the ports. The **entity** declaration does *not* describe how the design functions. Consider the logic circuit of a NAND function illustrated in Figure 4.2.

The **entity** declaration of the NAND function is illustrated in Figure 4.3. Notice that only the input and output ports of the logic circuit are identified, there is no reference to the function of the NAND circuit. The name of the **entity** is nand_gate.

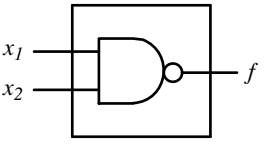


Figure 4.2 NAND Logic Circuit Input and Output Ports

```
entity nand_gate is
      port(
            x1, x2   : in     bit;
            f        : out    bit);
end nand_gate;
```

Figure 4.3 **Entity** Declaration of the NAND Logic Circuit in Figure 4.2

4.5.2 Architecture

The **architecture** describes the internal *structure* or *behavior* of the corresponding **entity**. More than one **architecture** may be associated with an **entity**. It is important to note that in VHDL (unlike software programming languages such as C and C + +), statements inside the **architecture** model events that happen *concurrently*. However, statements inside a **process** block (which is defined inside a **architecture** body) model events that occur simultaneously. Note that comments in a VHDL program begin with "- -" and every line requires a separate "- -" if the comment spans more than one line. Every **architecture** has a name and must include the name of the **entity** with which it is associated. The **architecture** declaration of the NAND circuit in Figure 4.2 is illustrated in Figure 4.4.

Consider the logic circuit shown in Figure 4.5. The VHDL implementation of the circuit requires an entity design, which identifies the input and output ports of the circuit, and an architecture design, which identifies the logic function of the circuit. The VHDL program in Figure 4.6 implements the logic circuit in Figure 4.5. Notice that the function of the circuit is described in the **architecture** of the code. In general, CAD tools have synthesis features that can be used to optimize the function without the user carrying out any prior simplification.

```
architecture Behavior of nand_gate is    begin
          f <= not (x1 and x2);
end Behavior;
```

Figure 4.4 **Architecture** Declaration of the NAND Logic Circuit in Figure 4.2

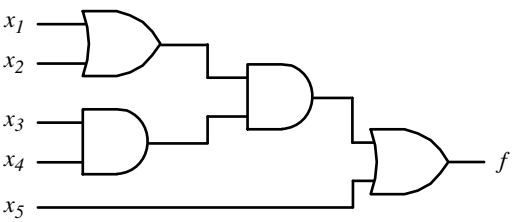


Figure 4.5 Logic Circuit Sample

```
library ieee ;
use ieee.std_logic_1164.all;

entity logic_circuit is
      port(
            x1,x2,x3,x4,x5  :    in   bit;
            f               :    out  bit);
end logic_circuit;
architecture Behavior of logic_circuit is
      begin
            f <= ((x1 or x2) and (x3 and x4)) or x5;
end Behavior;
```

Figure 4.6 VHDL Code for the Logic Circuit in Figure 4.5

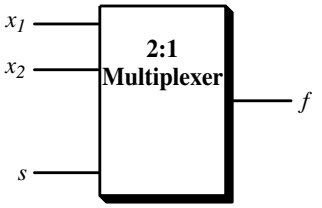


Figure 4.7 Block Diagram of a 2 : 1 Multiplexer

4.6 ASSIGNMENT STATEMENTS

In the examples above, the VHDL programs use simple assignment to describe the output logic function. The 2 : 1 multiplexer described in Chapter 3 has two possible output expressions depending on the value of the signal selected. VHDL provides selected signal assignment statements, which will assign a signal from several values using a selection condition. Consider the logic circuit for a 2 : 1 multiplexer shown in Figure 4.7. The VHDL program in Figure 4.8 implements the 2 : 1 multiplexer in Figure 4.7. The function of the multiplexer is described in the architecture of the code using selected signal assignment statements. The signal assignment statement selected begins with the reserved keyword **with** followed by the selection condition, which is in the input signal, *s*. The reserved keyword **when** selects a possible value for the select signal *s*. The **when others** (reserved keyword) is included to select the last possible value of *s*.

4.7 VHDL DATA TYPES

VHDL is a strongly typed language. This means that every object assumes the value of its nominated type. To put it very simply, the data type of the left-hand side (LHS) and

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux2to1 is
      port (
            x1,x2,s          :    in     bit;
            f                :    out    bit);
end mux2to1;
architecture Behavior of mux2to1 is
      begin
          with s select
            f   <=   x1      when   '0';
                     x2      when others;
end Behavior;
```

Figure 4.8 VHDL Code for the 2 : 1 Multiplexer in Figure 4.7

right-hand side (RHS) of a VHDL statement must be the same. The VHDL 1076 specification describes four classes of data types.

1. **Scalar types** represent a single numeric value or, in the case of enumerated types, an enumeration value. The standard types that fall into this class are integer, real (floating point), physical, and enumerated. All of these basic types can be thought of as numeric values.
2. **Composite types** represent a collection of values. There are two classes of composite types: arrays containing elements of the same type, and records containing elements of different types.
3. **Access types** provide references to objects in much the same way that pointer types are used to reference data in software programming languages.
4. **File types** reference objects (typically disk files) that contain a sequence of values.

The most common data types are described below. The data types define the set of value(s) that an object may take. The name of the type of object must be declared before the object is used in any VHDL statement.

1. ***Integer type***. Integer numbers range from –2147483647 to 2147483647, as defined by the standard using a 32-bit representation. Notice that there are equal number of positive and negative integers number in VHDL. The example in Figure 4.9 illustrates the use of integer type.
2. ***Real type***. Real numbers range between –1.0E38 and 1.0E38.
3. ***Bit and bit_vector types***. These types are predefined in VHDL standards IEEE 1076 and IEEE 1164. Hence, no library is needed to use these data types. A Bit object can take one of the values 0 and 1. An object of bit_vector type is an array of Bit objects. The examples in Figure 4.10 show how to declare and use objects of these types.

```
architecture test_int of test is
      begin
            process (x)
                  variable a: integer;
            begin
                  a : 1;          -- OK
                  a : -1;         -- OK
                  a : 1.0;        -- illegal
            end process;
end test_int;
```

Figure 4.9 Integer Type

```
--Declaration Examples
      signal   x1   :     bit ;
      signal   C    :     bit_vector (1 to 4);
      signal   D    :     bit_vector (7 downto 0);

--Assignment Examples
      x1                  <=    '1';
      C                   <=    "1010";
      D(7 downto 4)       <=    "1100";
```

Figure 4.10 Bit and Bit_Vector Types

4. ***Boolean type***. A Boolean object can take a value that is either true or false. "True" is equal to logic 1 and "false" is equal to logic 0.
5. ***Enumeration type***. The user specifies a list of possible values that an object could take. The examples in Figure 4.11 illustrate the enumeration data type.
6. ***Physical type***. Objects with physical type require associated units. The range of units must be specified. Notice that the "time" is the only physical type object predefined in the VHDL standards. The examples in Figure 4.12 illustrate physical-type definition for resistance units.
7. ***Std_logic and std_logic_vector types***. The std_logic data type is a part of IEEE standard 1164. It provides more flexibility than the Bit type. To use this type, two statements must be included in the library declaration (Figure 4.13). By including these statements, the user will have access to the std_logic_1164 *package*, which defines the std_logic type. The std_logic_vector is a linear array of objects of std_logic type. The IEEE 1164 standard defines the standard type that would allow multiple values to be represented for a wire. These values are listed in Figure 4.14. The examples in Figure 4.15 illustrate std_logic variable declarations and assignments using IEEE standard 1164.
8. ***Array type***. The array type is used to group elements of the same data type into a single VHDL object. The array may be constrained or unconstrained. Arrays may be one or multidimensional. The examples in Figure 4.16 illustrate objects of array data type.

```
type binary is (on, off)
… statements …
architecture test_enum of test is
      begin
            process (x)
                variable a : binary;
            begin
               a := ON;        -- OK
… statements …
                  a := OFF;       -- OK
… statements …
       end process;
end test_enum;
```

Figure 4.11 Enumeration Type

```
type resistance is range 0 to 1000000
      units
            Ohm;     -- ohm
            Kohm     = 1000 ohm;          -- i.e. 1 kΩ
            Mohm     = 1000 Kohm;         -- i.e. 1 MΩ
end units;
```

Figure 4.12 Physical Type

```
library ieee;
use ieee.std_logic_1164.all;
```

Figure 4.13 Std_Logic and Std_Logic_Vector Types

Value	Description
U	Un-initialized
X	Unknown
0	Logic 0 (driven)
1	Logic 1 (driven)
Z	High impedance
W	Weak 1
L	Logic 0 (read)
H	Logic 1 (read)
-	Don't-care

Figure 4.14 Std_Logic Values as Defined by IEEE 1164

```
--Declaration Examples
    A    :    std_logic_vector (3 downto 0);
    B    :    std_logic_vector (0 to 3);
    C    :    std_logic_vector (3 downto 0);
    Clk  :    std_logic;

--Assignment are similar to BIT-VECTOR data type
```

Figure 4.15 Std_Logic Variable Declaration

```
type reg_type is array (15 downto 0) of bit;
        variable  X  :  reg_type;
        variable  Y  :  BIT;
                Y := X(4);                -- Y gets value of element at index 4
```

Figure 4.16 Array Type

4.8 VHDL OPERATORS

VHDL provides predefined operators which are used as hardware modeling units. These include logical (or Boolean), arithmetic, and relational operators. The logical operators are listed in Figure 4.17. The NOT operator has one input and one output, whereas the remaining operators are binary operators, which have two input ports and one output port. VHDL relational operators are listed in Figure 4.18. Relational operators are a set of binary logical operators that generate a result of Boolean type: either "true" or "false." The relational operands are of either bit type or Boolean type, but not of mixed type.

VHDL arithmetic operators are listed in Figure 4.19. Arithmetic operators accept operands of integer type or floating-point type (real type). VHDL does not accept implicit type conversion between integer and floating-point numbers. The operators REM and MOD are defined only for integer numbers. The exponential operator

Operator	*Description*	*Operand Type*	*Result Type*
Not	*Not*	*Any Bit or Boolean type*	*Same Type*
And	*And*	*Any Bit or Boolean type*	*Same Type*
Or	*Or*	*Any Bit or Boolean type*	*Same Type*
Nand	*Not And*	*Any Bit or Boolean type*	*Same Type*
Nor	*Not Or*	*Any Bit or Boolean type*	*Same Type*
Xor	*Exclusive OR*	*Any Bit or Boolean type*	*Same Type*
Xnor	*Exclusive NOR*	*Any Bit or Boolean type*	*Same Type*

Figure 4.17 VHDL Logical (Boolean) Operators

Operator	*Description*	*Operand Type*	*Result Type*
=	*Equality*	*Any type*	*Boolean*
/=	*Inequality*	*Any type*	*Boolean*
<	*Less than*	*Any scalar type or discrete array*	*Boolean*
<=	*Less than or equal*	*Any scalar type or discrete array*	*Boolean*
>	*Greater than*	*Any scalar type or discrete array*	*Boolean*
>=	*Greater than or equal*	*Any scalar type or discrete array*	*Boolean*

Figure 4.18 VHDL Relational Operators

Operator	*Description*	*Operand Type*	*Result Type*
+	*Addition or positive sign*	*Same integer or floating-point type*	*Same type*
–	*Subtraction or negative sign*	*Same integer or floating-point type*	*Same type*
*	*Multiplication*	*Same integer or floating-point type*	*Same type*
/	*Division*	*Same integer or floating-point type*	*Same type*
MOD	*Modulus*	*Integer type*	*Same type*
REM	*Remainder*	*Integer type*	*Same type*
ABS	*Absolute value*	*Integer or floating-point type*	*Same type*
**	*Exponentiation*	*Left operant — integer or floating point type and Right operand (exponent) — integer type*	*Same type of left operand*
&	*Concatenation*	*Element or one-dimensional array type*	*One-dimensional array type*

Figure 4.19 VHDL Arithmetic Operators

accepts both integer and floating-point numbers with exceptions. The data type of the left operand of the exponential operator defines the data type of the result, and can be integer or floating point. The right operand (exponent) must be of integer type.

Precedence among VHDL operators is grouped into several operator classes. The operator class that has the highest precedence includes the exponential, the ABS (absolute value), and the NOT operators. The operator class that has the lowest precedence includes the logical operators AND, OR, NAND, NOR, XOR, and NXOR. In VHDL the operators that belong to the same operator class do not have precedence over each other. Parentheses must be used to define precedence among operators of the same operator class. The operators within the innermost parentheses are evaluated first, then the second set, and so on.

4.9 VHDL SIGNAL AND GENERATE STATEMENTS

4.9.1 Signal Statement

Signal is a VHDL keyword. It declares a signal of specified data type. A **signal** declaration is used to represent internal signals within an **architecture** declaration.

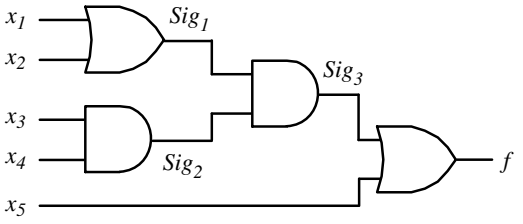


Figure 4.20 Logic Circuit with Internal Signals

Unlike **entity** ports, internal signals do not have a direction. Signal assignment statements execute only when the associated signals (appearing on the right-hand side of the assignment statement) change values. In VHDL, the order of concurrent statements in VHDL code does not affect the order in which the statements are executed. Signal assignments are concurrent and could execute in parallel fashion. Consider the logic circuit in Figure 4.20 where the internal signals are identified. Notice, that from the point of view of an **entity** declaration, the signals Sig_1, Sig_2, and Sig_3 are internal signals. They are neither input ports nor output ports, and therefore do not have a direction.

The VHDL program in Figure 4.21 implements the logic circuit in Figure 4.20. The **entity** declaration is similar to that in the VHDL program of Figure 4.6. The **architecture** declaration has been modified to include the internal signals. The logic function of the circuit is described in an indirect way using the internal signals. Both VHDL implementations (Figures 4.6 and 4.21) have the same logic function and should yield the same synthesized logic circuit.

```
library ieee ;
use ieee.std_logic_1164.all;

  entity logic_circuit is
        port(
              x1,x2,x3,x4,x5      :     in    std_logic ;
              f                   :     out   std_logic);
end logic_crcuit;
architecture Behavior of logic_circuit is
       signal Sig1,Sig2,Sig3   :    std_logic;
      begin
              Sig1 <= x1 or  x2;
              Sig2 <= x3 and  x4;
              Sig3 <= Sig1 and  Sig2;
              f <= Sig3 or x5;
end Behavior;
```

Figure 4.21 VHDL Code for the Logic Circuit in Figure 4.20

```
iterative:                                    -- for generate
for I in 0 to 5 generate
      … enclosed statements …;
end generate iterative;

conditional:                                  -- if generate
if ( I <= 4 and I >=1) generate
      … enclosed statements …;
 end generate conditional;
```

Figure 4.22 Generate Statement

4.9.2 Generate Statement

Generate is a VHDL keyword used to replicate a set of concurrent statements or to selectively execute a set of concurrent statements if a specified condition is met. The **generate** statement provides a method of repeating a logic function or a component instantiation without manually writing the logic function or the component instantiation. Components and component instantiations are described in Section 4.13. There are two types of **generate** statements: **for generate**, which is an iterative **generate** statement, and **if generate**, which is a conditional **generate** statement. The examples in Figure 4.22 illustrate both **generate** statements.

A **generate** statement begins with a unique label that identifies it. The label also appears at the close of the statement. The **generate** statement repeats the enclosed statements at each iteration or conditional pass. The index I is declared implicitly inside the **generate** statement and cannot be changed by the program.

4.10 SEQUENTIAL STATEMENTS

Sequential VHDL statements allow the designer to describe the operation, or *behavior*, of a circuit as a sequence of related events. Sequential statements are found within processes, functions, and procedures. Sequential statements differ from concurrent statements in that they have order dependency, which may or may not imply a sequential circuit (one involving memory elements). VHDL's **process** statement is the primary way to enter a sequential statement. A **process** statement, including all declarations and sequential statements within it, is actually a single concurrent statement within a VHDL architecture. This means that the designer can write as many processes and other concurrent statements as are necessary to describe a design, without worrying about the order in which the simulator will process each statement. Thus, the **process** statement constitutes the behavioral statement in VHDL. The example in Figure 4.23 illustrates the general structure of the **process** statement.

A **process** statement is listed in the **architecture** declaration. The **process** statement begins with the reserved keyword **process**, following the **begin** statement

```
Label:                                    Process statement label
process (sensitivity list)
      … Declaration section;

begin
      … Statement section;

end process ;
```

Figure 4.23 Sequential Declaration

of the architecture. The **process** statement includes a **begin** statement inside it as well. The **process** statement concludes with an **end process** statement. The process may have a label, which also appears at the close of the **process** statement. The **process** statement structure is composed of a declaration section and a statement section. The declaration section declares the objects that will be used in the statement section of the **process** statement. The statement section includes sequential statements, which describe the sequential behavior of the logic circuit. The sequential statements, a set of loops and decision-making statements, are discussed in Section 4.11. The **process** statement may include a sensitivity list, which defines the activation and suspension of the **process** statement based on the changes in signals included in the sensitivity list. Because the **process** statement represents sequential behavior, it must include an explicit **wait** statement to control the activation and suspension of the process statement. When a sensitivity list is included in the **process** statement, the **wait** statement is not necessary. The sensitivity list provides an implicit wait, which describes the events of the signals listed.

4.11 LOOPS AND DECISION-MAKING STATEMENTS

Loop statements are a category of control structure that allow a designer to specify repeating sequences of behavior in a circuit. There are three primary types of loops in VHDL: **for** loops, **while** loops, and infinite loops. VHDL also provides **if–then–else** and **case** statements to implement control structures.

4.11.1 For Loop

A **for** loop is a sequential statement that allows a designer to specify a fixed number of iterations in a behavioral design description. It is important to note that in VHDL, unlike other software programs, each iteration occurs concurrently, which means that the loop is "unrolled." A **for** loop can be used only inside a sequential statement, such as a **process** statement, a function, or a procedure. The example in Figure 4.24 illustrates the form of a **for** statement. A **for** loop executes the sequential statement within its body each time the index of the loop changes its value within the range of the loop. The label of the **for** loop is optional; the user may choose not to include it.

```
For_Loop:                                   For loop label
for index in range
      loop
          sequential statement;
          sequential statement;

end loop For_loop;
```

Figure 4.24 For Loop

```
while_Loop:                                 While loop label
while condition
      loop
          sequential statement;
          sequential statement;

end loop while_loop;
```

Figure 4.25 While Loop

4.11.2 While Loop

A **while** loop is another form of sequential loop statement that specifies the conditions under which the loop should continue rather than specifying a discrete number of iterations. The condition must be of type **Boolean**. A **while** loop executes the sequential statements in its body each time the condition is checked and evaluated to be true. Otherwise, the **while** loop terminates. Similar to a **for** loop, the label of a **while** loop is optional and can be omitted. The general form of a **while** loop is illustrated in the code sample shown in Figure 4.25.

4.11.3 If–Then–Else Statement

The **if–then–else** statement is the most commonly used form of control statement in VHDL. Its general form is illustrated in the code sample shown in Figure 4.26. The condition statements must be expressions of type **Boolean**.

```
if first_condition then
      sequential statements;
elsif second_condition then
      sequenctial statements;
else
      sequential statements;
end if;
```

Figure 4.26 If–Then–Else Statement

```
case control_expression is
      when test_expression1 => sequential statements;
      when test_expression2 => sequential statements;
     …
     ..
     when others => sequential statements;
end case;
```

Figure 4.27 Case Statement

4.11.4 Case Statement

A **case** statement can be used as an alternative to an **if–then–else** control structure. However, it is important to understand the differences in the circuits generated by CAD tools for the **if** and **case** statements, and to use the appropriate control structure for a design. A **case** statement is generally used when making decisions among options that have equal priority. The making-decision control expression must have a finite set of values. The **when** statement switches between the possible values. The **when others** statement is the final statement with a **case** statement. A **when others** statement is used to represent the remaining possible values of the control expression, not listed in the previous **when** statements. The general form of a **case** statement is illustrated in the code sample shown in Figure 4.27. The **when others** condition makes sure that all cases (that are not listed within the case body) are covered.

4.12 SUBCIRCUIT DESIGN

VHDL provides many high-level features that help a designer manage complex designs. In fact, design management is one of VHDL's key strengths compared to alternative design entry languages and methods. Like modularity features (functions and procedures), design partitioning is another important aspect of design management. Design partitioning goes beyond simpler design modularity methods to provide comprehensive design management across multiple projects and to allow alternative structural implementations to be tried out with minimal effort. Design partitioning is particularly useful for those designs being developed in a team environment, as it promotes cooperative design efforts and well-defined system interfaces.

4.13 PACKAGES AND COMPONENTS

4.13.1 Package Statement

Packages are intended to hold commonly used declarations such as constants, type declarations, and global subprograms. Packages can be included within the same

source file as other design units (such as entities and architectures) or can be placed in a separate source file and compiled into a named library. Packages may contain the following types of objects and declarations:

- Type and subtype declarations
- Constant declarations
- File and alias declarations
- Component declarations
- Attribute declarations
- Functions and procedures
- Shared variables

When items from the package are required in other design units, the **use** statement must be included to make the package and its contents visible for each design unit. The package is stored in a separate VHDL code. It can also be included at the end of a VHDL code, which contains the entity of the architecture of the subcircuit. In both cases the package is compiled and stored in a library, which can be accessed by employing the **use** statement. The entity and architecture of the subcircuit referenced in a package must be compiled and made available to the VHDL code, which uses the package reference. The VHDL compiler will use the subcircuit to synthesize the complete circuit. The example in Figure 4.28 illustrates a **package** declaration for a 2 : 1 multiplexer circuit. Notice that the **package** declaration included the 2 : 1 multiplexer as a component declaration, which is described in the next section.

The VHDL codes for the 2 : 1 multiplexer and its associated package must be compiled and stored in a defined directory. If both codes were available in the working directory, a library declaration is not necessary because the compiler has access to the working directory. The example shown in Figure 4.29 illustrates how to access the 2 : 1 multiplexer package by employing the **use** statement. The "all" clause indicates that all declarations included in the package are visible to the calling program, whether or not it is using them.

```
library ieee;
use ieee.std_logic_1164.all;

package mux2to1_package is
      component mux2to1
            port (
                  x1,x2,s    :   in    bit;
                  f          :   out   bit);
      end component;
end mux2to1_package;
```

Figure 4.28 Package Statement

```
library ieee;
use ieee.std_logic_1164.all;
use work.mux2to1_package.all;

entity example is
      -- entity declaration;
end example;

architecture behavior of example is
      -- architecture declaration;
end behavior;
```

Figure 4.29 **Use** Statement

4.13.2 Component Statement

The **component** declaration specifies the input and output ports of a subcircuit. A **component** declaration can appear in an **architecture** or **package** declaration with a structure similar to that of an **entity** declaration. A **component** declaration includes the name of the component and its ports. The difference between a **component** and an **entity** declaration is that an **entity** declaration declares a circuit model that has an architecture, whereas a **component** declaration declares a virtual circuit template, which must be instantiated to take effect during the design. A **component** declaration could also be listed inside a package rather than the architecture. Components are used

```
library ieee;
use ieee.std_logic_1164.all;
use work.mux2to1_package.all;

entity mux4to1 is
      port (
            x     :     in     std_logic_vector(0 to 3);
            s     :     in     std_logic_vector(1 downto 0);
            f     :     out    std_logic);
end mux4to1;
architecture behavior of mux4to1 is
      signal sig   :   std_logic_vector (0 to 1)
      component mux2to1
            port (
                  x1,x2,s     :     in     bit;
                  f           :     out    bit);
      end component;
begin
      mux1 : mux2to1 port map (x(0),x(1),s(0),sig(0));
      mux2 : mux2to1 port map (x(2),x(3),s(0),sig(1));
      mux3 : mux2to1 port map (sig(0),sig(1),s(1),f);
end behavior;
```

Figure 4.30 **Component** Statement

to create subblocks that can be put together to form larger hierarchical designs. For example, a multiplier may be built from full-adders, which in turn may be built from half-adders. One could write VHDL programs for the half-adder, put two instances (copies) of the half-adder in another VHDL program to build a full-adder and write a third program to put full-adders together to make a multiplier. This is an example of hierarchical design (bottom-up design). The following example shows how to define components and use them in a structural VHDL program. The example illustrates how to design a 4 : 1 multiplexer from three 2 : 1 multiplexers (Figure 4.30). Multiplexers are described in Chapter 7. The reserved keyword **port map** is used to instantiate the 2 : 1 multiplexer component as needed in the design of the full 4 : 1 multiplexer. Note that the component "mux2to1" has to be precompiled. Port mapping is done to connect signals of the design in which the components are instantiated with the ports of the component itself.

PROBLEMS

4.1 Describe the basic CAD tools for logic design.

4.2 Describe the major design entry tools.

4.3 What is the difference between simulation and synthesis?

4.4 Why should the designer perform a function simulation before final hardware implementation?

4.5 What is VHDL?

4.6 Describe the basic programming structure of VHDL code.

4.7 What is the difference between an **entity** declaration and an **architecture** declaration?

4.8 Can an **entity** have more than one **architecture**?

4.9 What is a VHDL **signal** statement? What is it used for?

4.10 What is the function of the "when others" clause in a **case** statement?

4.11 Describe the difference between the **process** and **generate** statements.

4.12 What is the difference between a VHDL component and a VHDL package?

4.13 Write VHDL code to implement a XOR logic gate.

4.14 Write VHDL code to implement a NOR logic gate.

4.15 Write VHDL code to implement a NXOR logic gate.

4.16 Write VHDL code to implement the logic circuit in Figure P4.16.

4.17 Write VHDL code to implement the logic circuit in Figure P4.16 using signal data objects.

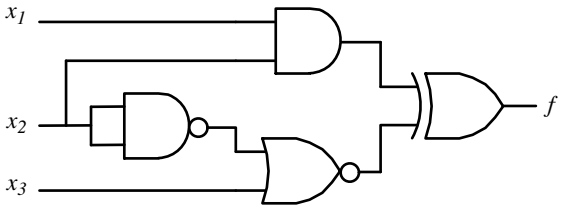


Figure P4.16

4.18 Write VHDL code to implement the logic circuit in Figure P4.18.

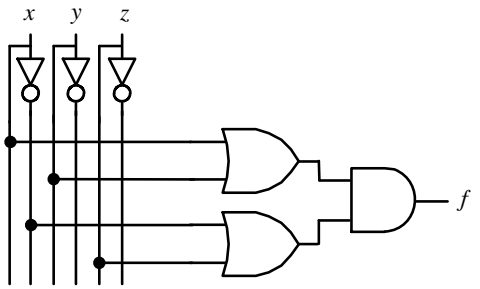


Figure P4.18

4.19 Write VHDL code to implement the logic circuit in Figure P4.18 using signal data objects.

4.20 Write VHDL code to implement the logic circuit in Figure P4.20.

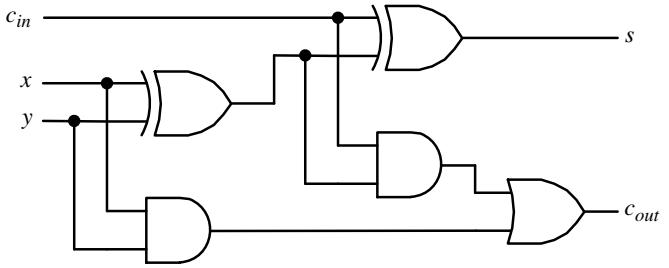


Figure P4.20

4.21 Write VHDL code to implement the logic circuit in Figure P4.20 using signal data objects.

4.22 Write VHDL code to implement the logic circuit in Figure P4.22.

4.23 Write VHDL code to implement the logic circuit in Figure P4.22 using component declarations for NOT, AND, and OR logic gates.

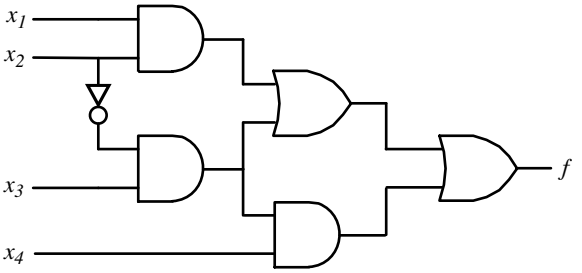


Figure P4.22

4.24 Write VHDL code to implement the logic circuit in Figure P4.22 using a package declaration which includes component declarations for NOT, AND, and OR logic gate.

4.25 Write VHDL code to implement the logic circuit in Figure P4.25.

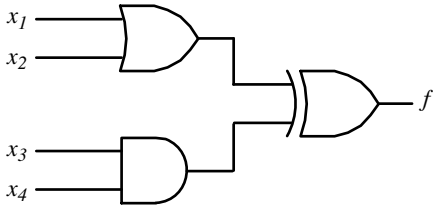


Figure P4.25

4.26 Write VHDL code to implement the logic circuit in Figure P4.25 using component declarations for AND, OR, and XOR logic gates.

4.27 Write VHDL code to implement the logic circuit in Figure P4.25 using a package declaration which includes component declarations for AND, OR, and XOR logic gates.

4.28 Write VHDL code to implement the logic circuit in Figure P4.28.

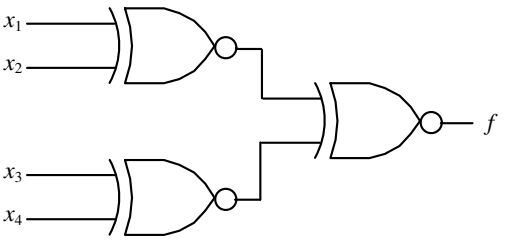


Figure P4.28

4.29 Write VHDL code to implement the logic circuit in Figure P4.28 using a component declaration for XOR logic gate.

4.30 Write VHDL code to implement the logic circuit in Figure P4.28 using a package declaration which includes a component declaration for XOR logic gate.

4.31 Write VHDL code to implement the following logic expressions.

(a) $f_1 = (x_1 + x_2)(\bar{x}_1 + x_2)(x_1 + \bar{x}_2)$

(b) $f_2 = \overline{\overline{x_1 x_2} + (x_1 + x_2)}$

4.32 Write VHDL code to implement the following expressions.

(a) $f_1 = x_1\bar{x}_2 + x_1\bar{x}_2 x_3$

(b) $f_2 = \overline{x_1 + \bar{x}_2 + \bar{x}_3 + x_4} + \bar{x}_1 x_2 \bar{x}_3 x_4$

5 Integrated Logic

5.1 OBJECTIVES

The objectives of the chapter are to describe:

- Logic signals
- Logic switches
- NMOS and PMOS gates
- CMOS gates
- CMOS static and dynamic behaviors
- TTL gates and design practical aspects
- Transmission gates

5.2 LOGIC SIGNALS

Logic variables can be used to represent such electronic signals as voltage, current, and frequency. There are a number of systems for representing binary information in physical systems, such as:

- A voltage signal with zero (0) corresponding to 0 V and one (1) corresponding to 5 or 3 V.
- A sinusoidal signal with zero (0) corresponding to a frequency and one (1) corresponding to another frequency.
- A current signal with zero (0) corresponding to 4 mA and one (1) corresponding to 20 mA.
- For switches, *open* is indicated by 0 and *closed* is indicated by 1.

Now let us describe the logic levels using voltage signals. To represent voltage signals using logic levels, we first have to define *threshold voltage.* Any voltage below the threshold voltage represents one logic value, and voltages above this reference voltage represent the other logic value. We can assign either 1 or 0 to represent any

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

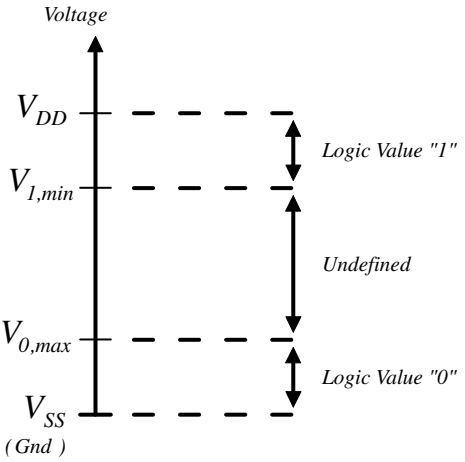


Figure 5.1 Logic Levels for Positive Logic

level, depending on the logic system. There are two different types of logic systems: positive and negative. In a *positive logic system*, any voltage greater than the threshold voltage is represented by logic 1, and voltages below the reference level are represented by logic 0. The reverse notation is used for a *negative logic system*: Logic 1 and logic 0 are used to represent lower voltage levels and higher voltage levels, respectively.

Generally, we use voltage ranges rather than a fixed voltage level to represent logic levels. Figure 5.1 shows the ranges: Any voltage within V_{DD} and $V_{1,\mathrm{min}}$ is represented by logic 1 and the voltages between V_{SS} and $V_{0,\mathrm{max}}$ are represented by logic 0. The range of voltages between $V_{1,\mathrm{min}}$ and $V_{0,\mathrm{max}}$ is undefined. Here V_{DD} and V_{SS} are the maximum and minimum voltages, respectively. Figure 5.1 illustrates voltage levels for positive logic. Typically, V_{DD} is equal to 5 V and V_{SS} is equal to 0 V or ground.

5.3 LOGIC SWITCHES

As described earlier, the logic 1 and logic 0 states represent closed and open switches, respectively. We know that the operation of a transistor is similar to that of a simple switch, so logic circuits can be built using transistors. Generally, *metal–oxide semiconductor field-effect transistors* (MOSFETs) are used to implement the switches. There are two different types of MOSFETs: *n*-channel and *p*-channel. An *n*-channel MOSFET has four electrical terminals, known as *gate* (*G*), *drain* (*D*), *source* (*S*), and *body/substrate* (*B*). The operation of an *n*-channel MOSFET is as follows. When the gate voltage (V_G) is greater than the threshold voltage (V_{TH}), a connection is established between the source and the drain and the transistor is said to be turned on and acts as a *closed switch*. If the gate voltage (V_G) is low, there is no connection between the source and the drain and it acts as an *open switch*. The symbol and the equivalent circuit for a simple *n*-channel MOSFET are shown in Figure 5.2.

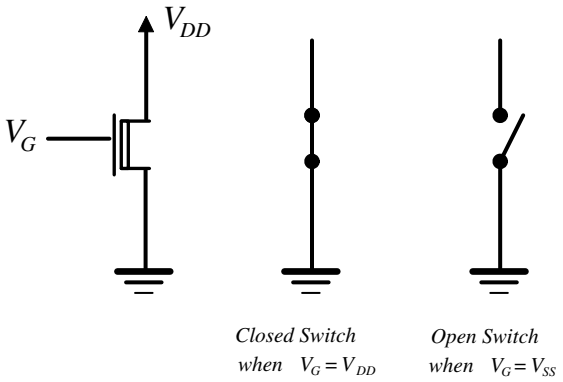


Figure 5.2 NMOS Transistor as a Switch

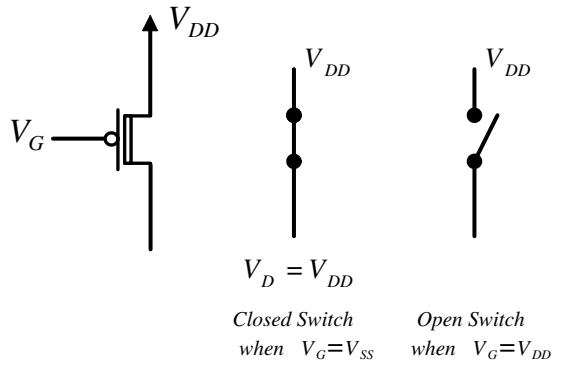


Figure 5.3 PMOS Transistor as a Switch

PMOS transistors behave in exactly the opposite way to NMOS transistors. In this case the transistor acts as an open switch for logic 1 and as a closed switch for logic 0. The PMOS transistor also has four electrical terminals, known as *gate, drain, source,* and *body*. Generally in logic circuits, the body is connected to V_{DD}. If V_G is high, the transistor is turned off, implying that there will be no connection between the source and the drain terminals, and it acts like an open switch. When V_G is low, a connection is established between the source and drain terminals, and the transistor works as a closed switch. The symbol and the equivalent circuit for a simple PMOS transistor are shown in Figure 5.3.

5.4 NMOS AND PMOS LOGIC GATES

5.4.1 NMOS Inverter

Consider the circuit shown in Figure 5.4. The operation of the circuit can be explained as follows. When $V_G = 0$ V (logic 0), the NMOS switch is closed and the NMOS

Truth Table	*Integrated Circuit*

V_G	T_1	V_{out}
0	*off*	1
1	*on*	0

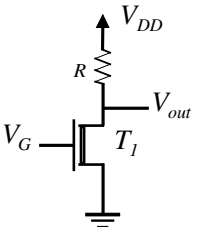


Figure 5.4 NMOS Inverter Gate and Its Truth Table

transistor T_1 is off and no current flows through resistor R. The output voltage V_{out} is equal to V_{DD} (logic 1). However, if $V_G = V_{DD}$ (logic 1), the NMOS switch is closed and the NMOS transistor T_1 starts conducting, thereby pulling down the output node to ground. Thus, the output voltage is logic 0. The circuit in Figure 5.4 acts as an inverter gate. The purpose of resistor R is to limit the current when the NMOS transistor is turned on. In other words, this resistor acts as a current source load. It will be replaced with a PMOS transistor in later circuit design. The truth table is also shown in Figure 5.4.

5.4.2 NMOS NAND Gate

Now observe the circuit diagram shown in Figure 5.5. Consider the case when both inputs are high (i.e., logic 1) and NMOS transistors T_1 and T_2 are both turned, pulling

Truth Table	*Integrated Circuit*

V_{G1}	V_{G2}	T_1	T_2	V_{out}
0	0	*off*	*off*	1
0	1	*off*	*on*	1
1	0	*on*	*off*	1
1	1	*on*	*on*	0

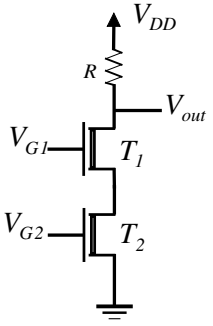


Figure 5.5 NMOS (Two-Input) NAND Gate and Its Truth Table

the output node down to ground, resulting in logic 0 as output. On the other hand, if any one of the inputs or both inputs is low (i.e., logic 0), one or both transistors will be turned off and act as an open switch, and thus the output will remain high (i.e., logic 1). The resulting truth table for this circuit is also shown in Figure 5.5, which is the same as that of the NAND gate.

5.4.3 NMOS NOR Gate

As we have seen above, the series connection of transistors implies a NAND operation. Following the principle of duality, to perform a NOR operation the transistors are connected in parallel. A circuit that realizes the function of a (two-input) NOR gate is illustrated in Figure 5.6. In this case, if any of the inputs is logic high, the output node is pulled down to ground, resulting in logic 0 at the output node. For the case when both the inputs are low (i.e., logic 0), the output is at logic 1. The corresponding truth table is also shown in Figure 5.6.

In the process of designing any logical circuit in NMOS technology, we have two blocks, known as a *pull-up network* and a *pull-down network*. In all the NMOS circuits above, resistor *R* acts as a pull-up network and all the NMOS transistors work as pull-down networks. To save the chip area we can replace the pull-up resistor with a PMOS transistor. Such circuits are called *pseudo-NMOS circuits*. NMOS and pseudo-NMOS circuits have the same pull-down network.

5.5 CMOS LOGIC GATES

Here we are going to use CMOS transistors, known as *complementary MOS transistors*, consisting of both PMOS and NMOS transistors. As for NMOS logic circuits, these CMOS logic circuits have pull-up and pull-down networks. However,

Truth Table

V_{G1}	V_{G2}	T_1	T_2	V_{out}
0	0	*off*	*off*	1
0	1	*off*	*on*	0
1	0	*on*	*off*	0
1	1	*on*	*on*	0

Integrated Circuit

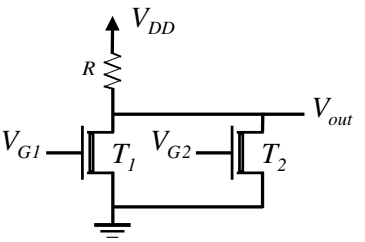


Figure 5.6 NMOS (Two-Input) NOR Gate and Its Truth Table

Truth Table	*Integrated Circuit*

V_{in}	T_1	T_2	V_{out}
0	*off*	*on*	1
1	*off*	*off*	0

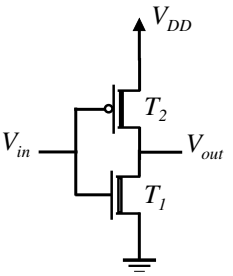


Figure 5.7 CMOS NOT Gate and Its Truth Table

for CMOS logic, the pull-up network consists of PMOS transistors. The functions realized by pull-up and pull-down networks are complementary to each other. The *n*-channel network is between the output and ground, and the *p*-channel network is between the output and V_{DD}. The number of transistors in each network is equal to the number of inputs.

5.5.1 CMOS Inverter

The circuit diagram for a CMOS inverter is shown in Figure 5.7. For the logic high input, transistor T_1 will be turned *on* and T_2 will be *off*, thus pulling down the output node to ground, resulting in logic 0 at the output. On the other hand, for logic 0 input, T_1 will be *off* and T_2 will be *on*, thus connecting the output node to the higher voltage, V_{DD}. Notice that there is no protective resistance.

5.5.2 CMOS NAND Gate

A CMOS (two-input) NAND gate is shown in Figure 5.8. For this network, if all the inputs are high, the NMOS transistors will be *on*, the PMOS transistors will be *off*, and the output will be pulled low to Ground. Conversely, if any of the inputs is a 0, one of the NMOS transistors will be off and the *n*-channel network offers infinite impedance. One of the PMOS transistors will be on and the output will be pulled up to V_{DD} (logic 1). Thus, any 0 input guarantees a 1 output and all '1's give a 0 output, describing the functionality of a NAND gate.

5.5.3 CMOS NOR Gate

Figure 5.9 shows the circuit diagram of a two-input CMOS NOR gate. Note that the NMOS transistor network is the same as that seen previously for the NMOS NOR gate. The only difference here is that the complement of the two parallel NMOS

Truth Table

V_{G1}	V_{G2}	T_1	T_2	T_3	T_4	V_{out}
0	0	*off*	*off*	*on*	*on*	1
0	1	*off*	*on*	*on*	*off*	1
1	0	*on*	*off*	*off*	*on*	1
1	1	*on*	*on*	*off*	*off*	0

Integrated Circuit

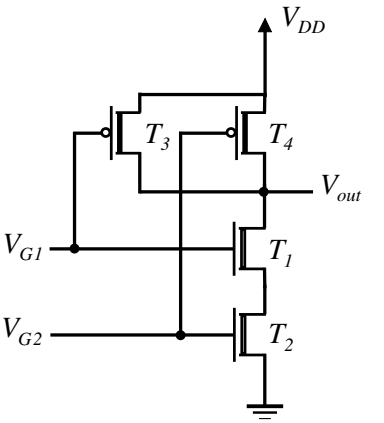


Figure 5.8 CMOS (Two-Input) NAND Gate and Its Truth Table

transistors is a pull-up network. If any input is high, at least one of the NMOS transistors will be on, pulling the output to ground (i.e., logic 0). Conversely, if all the inputs are low, all the NMOS transistors will be off and all PMOS transistors will be on, thus pulling the output to V_{DD} (i.e., logic 0). The corresponding truth table is also shown in Figure 5.9.

Truth Table

V_{G1}	V_{G2}	T_1	T_2	T_3	T_4	V_{out}
0	0	*off*	*off*	*on*	*on*	1
0	1	*off*	*on*	*on*	*off*	0
1	0	*on*	*off*	*off*	*on*	0
1	1	*on*	*on*	*off*	*off*	0

Integrated Circuit

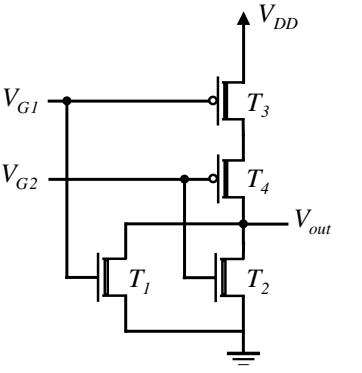


Figure 5.9 CMOS (Two-Input) NOR Gate and Its Truth Table

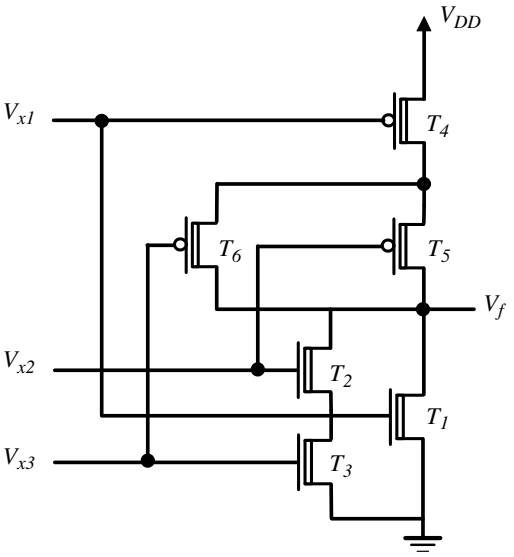


Figure 5.10 Integrated CMOS (Three-Input) Logic Circuit

5.6 CMOS LOGIC NETWORKS

Design of logic networks using integrated transistors is an extensive discipline referred to as *VLSI technology*. Physical aspects of transistors play a major role in the layout of logic gates. VLSI technology is beyond our scope here. Nonetheless, the design concepts illustrated in Section 5.5 can be used to design integrated logic gates using CMOS transistors without exploring their practical aspects in detail. To understand how PMOS and NMOS transistors can be interconnected to realize a logic function, we analyze the CMOS logic circuit in Figure 5.10. We describe the logic function of the circuit by determining the state of each PMOS and NMOS transistor as the input voltages change according to the truth table.

An NMOS transistor is off when its input gate voltage is equal to 0 V, and it is on when its input gate voltage is equal to 5 V. A PMOS transistor is off when its input gate voltage is equal to 5 V, and it is on when its input gate voltage is equal to 0 V. Using these properties of NMOS and PMOS transistors, one should be able to determine the state of each transistor in the logic circuit. The truth table of the integrated logic circuit is also shown in Figure 5.11.

Using the SOP method, the logic function of the integrated logic circuit is evaluated and simplified to the expression

$$f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot \bar{x}_2 \cdot x_3 + \bar{x}_1 \cdot x_2 \cdot \bar{x}_3 = \bar{x}_1(\bar{x}_2 + \bar{x}_3)$$

Note that the logic expression is in a complemented form which combines NAND and NOR operations. Therefore, to obtain a simple integrated logic circuit, the

V_{G1}	V_{G2}	V_{G3}	T_1	T_2	T_3	T_4	T_5	T_6	V_{out}
0	0	0	*off*	*off*	*off*	*on*	*on*	*on*	1
0	0	1	*off*	*off*	*on*	*on*	*on*	*off*	1
0	1	0	*off*	*on*	*off*	*on*	*off*	*on*	1
0	1	1	*off*	*on*	*on*	*on*	*off*	*off*	0
1	0	0	*on*	*off*	*off*	*off*	*on*	*on*	0
1	0	1	*on*	*off*	*on*	*off*	*on*	*off*	0
1	1	0	*on*	*on*	*off*	*off*	*off*	*on*	0
1	1	1	*on*	*on*	*on*	*off*	*off*	*off*	0

Figure 5.11 Truth Table of the CMOS Logic Circuit in Figure 5.10

designer should attempt to express the logic function as a complemented form using DeMorgans theorem. Complemented products are implemented using NMOS transistors in series, and complemented sums are implemented using PMOS transistors in parallel. To design an integrated logic circuit from explicit logic functions, which may not be complemented easily, we use the CMOS logic gates described in Section 5.5. The resulting integrated circuits are only gross approximations of the actual circuits. In practice, however, advanced VLSI methods and propriety methods are used to design efficient CMOS integrated logic circuits which take into account the physical aspects of CMOS transistors.

5.7 PRACTICAL ASPECTS OF LOGIC GATES

5.7.1 Fan-in and Fan-out Effects

In this section we describe the physical aspects of logic gates, which include the fan-in, fan-out, noise margin, power dissipation, and propagation delays. The *fan-in* is the number of inputs of a logic gate. For examples, a two-input AND gate has a fan-in of 2 and a three-input NAND gate has a fan-in of 3. So a NOT gate always has a fan-in of 1, which implies that for any logic circuit the inputs cannot be increased beyond a finite number (i.e., fan-in). If the number of inputs is increased, the parasitic capacitance and thus the propagation delay is increased and the noise margin is lowered. Normally, the propagation delay increases as a quadratic function of the fan-in. The number of gates that each logic gate can drive while providing voltage levels in the guaranteed range is called the *standard load* or *fan-out*. The fan-out depends on the amount of electric current that a gate can source or sink while driving other gates. The effects of loading a logic gate output with more than its rated fan-out include the following:

- In the LOW state the output voltage V_{OL} may increase above V_{OLmax}.
- In the HIGH state the output voltage V_{OH} may decrease below V_{OHmin}.
- The operating temperature of the device may increase, thereby reducing the reliability of the device and eventually causing the device to fail.
- Output rise and fall times may increase beyond specifications.
- The propagation delay may rise above the value specified.

Normally, as in the case of fan-in, the propagation delay introduced by a gate increases as the fan-out increases.

5.7.2 Noise Margins

Gate circuits are constructed to sustain voltage variations in input and output voltage levels. Voltage variations are usually the result of several factors.

- Batteries lose their full potential, causing the supply voltage to drop.
- High operating temperatures may cause a drift in transistor voltage and current characteristics.
- Spurious pulses may be introduced on signal lines by normal surges of current in neighboring supply lines.

All these undesirable voltage variations that are superimposed on normal operating voltage levels are called *noise*. Gates are designed to tolerate a certain amount of noise on their input and output ports. The maximum noise voltage level that is tolerated by a gate, called a *noise margin*, is derived from the *voltage transfer characteristic* measured under different operating conditions. It is normally supplied in the documentation about the gate provided by the manufacturer.

- **NM_L (low noise margin)**: the largest noise amplitude that is guaranteed not to change the output voltage level when it is superimposed on the input voltage of the logic gate (when this voltage is in the LOW interval). $NM_L = V_{ILmax} - V_{OLmax}$.
- **NM_H (high noise margin)**: the largest noise amplitude that is guaranteed not to change the output voltage level when it is superimposed on the input voltage of the logic gate (when this voltage is in the HIGH interval). $NM_H = V_{OHmin} - V_{IHmin}$.

5.7.3 Propagation Delay

Gate propagation delay is the delay introduced by the gate for a signal appearing at its input, before it reaches the gate's output. Gate propagation delay may not be the same for both transitions (i.e., gate propagation delay will be different for low-to-high transition as compared to high-to-low transition). Low-to-high transition is called *turn-on delay*, and high-to-low transition is called *turn-off delay*. Different technical terms are used to specify the gate propagation delay: rise time, fall time, and propagation delay.

- *Rise time* is the time required for the output voltage to increase (rise) from V_{ILmax} to V_{IHmin}.
- *Fall time* is the time required for the output voltage to decrease (fall) from V_{IHmin} to V_{ILmin}.

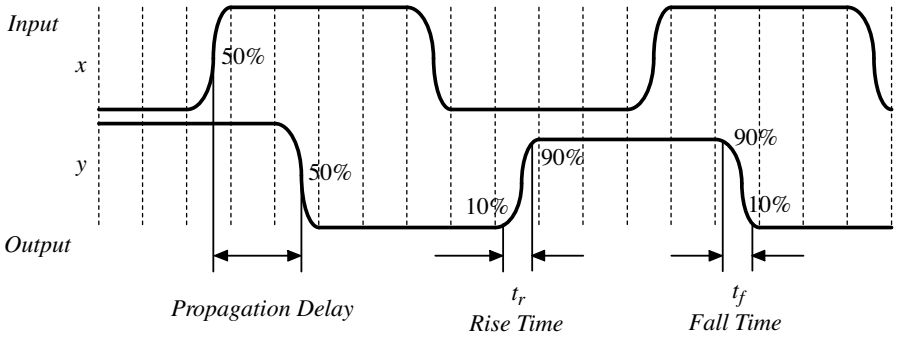


Figure 5.12 Time Characteristics and Propagation Delay of a Typical Inverter

- *Propagation delay* is the time between the logic transition on an input and the corresponding logic transition on the output of the logic gate. In other words, the propagation delay is the time it takes the output of a gate to respond to a change at its inputs. Generally, it is measured at midpoints (50% of the signal).

Figure 5.12 illustrates time characteristics and the propagation delay of a typical inverter (NOT) logic gate. The propagation delay, a critical parameter in a sequential circuit, is described in Chapters 7 and 8. In general, input and output waveforms are drawn without showing the propagation delays introduced by the various gates. The worst-case propagation delay must be considered when designing logic circuits using CAD tools to obtain functional digital circuits. The VHDL code in Figure 5.13 includes the propagation delay introduced by a typical inverter gate.

The clause **after** 20 ns follows the concurrent assignment expression of the output *f*. The 20 ns is the time it takes for the output to take effect after any change in the input signals *x*. This time interval mimics the actual propagation of a practical inverter. In practice, when using CAD tools, a synthesized logic circuit is generally resynthesized with realistic propagation delays, which take into account propagation delays of the targeted platforms.

```
library ieee ;
use ieee.std_logic_1164.all;

entity inverter_circuit is
      port(
            x           :       in      std_logic;
            f           :       out     std_logic);
end inverter_circuit;
architecture behavior of inverter_circuit is
      begin
            f   <=   not x   after   20 ns;
end behavior;
```

Figure 5.13 VHDL Code for a Typical Inverter with Propagation Delay

5.7.4 Power Dissipation

Power dissipation is an important metric, for two reasons. The amount of current and power available in a battery is nearly constant. The power dissipation of a circuit or system defines the battery's life. The greater the power dissipation, the shorter the battery life. The power dissipation is proportional to the heat generated by the chip or system. Excessive heat dissipation may increase the operating temperature and cause the gate circuitry to drift out of its normal operating range, and may cause gates to generate improper output values. Thus, the power dissipation of any gate implementation must be kept as low as possible. The power consumed by a logic gate under steady-state conditions is known as *static power dissipation*. *Dynamic power dissipation* is the power consumed during input and output transitions.

- P_s **(static power dissipation)**: the power consumed when the output or input is not changing or when the clock is turned off. Normally, static power dissipation is caused by leakage current. [As the size of the transistor is reduced (below 90 nm), the leakage current could be as high as 40% of the total power dissipation.]
- P_d **(dynamic power dissipation)**: the power consumed during output and input transitions when the signal level changes. It is caused by the switching current, which charges and discharges parasitic capacitance. It is also caused by the short-circuit current when both NMOS and PMOS transistors are momentarily on at the same time. P_d is the actual power consumed by transistors plus the leakage current.

NMOS circuits consume static and dynamic power, whereas CMOS circuits consume only dynamic power. Thus, CMOS power dissipation is less than that of NMOS circuits. The total power dissipation is the sum of the static and dynamic power dissipation.

5.8 TRANSMISSION GATES

A *transmission gate* is constructed from a normally open switch (NMOS transistor) wired in parallel with a normally closed switch (PMOS transistor), with complementary control signals. Figure 5.14 shows the transistor and schematic representations of a transmission gate. NMOS transistors pass logic 1 well and logic 0 poorly, and the opposite is true for PMOS. However, a transmission gate is equally good at passing a 0 or 1 when an external control signal is asserted.

5.8.1 Multiplexer

CMOS transmission gates provide an efficient way to build steering logic. *Steering logic circuits* are circuits that route data inputs to outputs based on the settings of control signals. As an example of such a circuit, consider a circuit that has two data inputs, x_1 and x_2, a single output, f, and a control input, s. The function steers x_1 to f

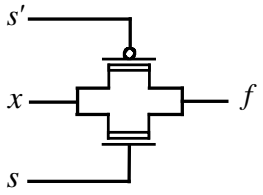
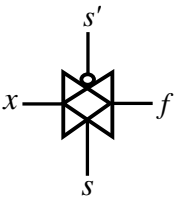
Transmission Logic Circuit	*Graphical Symbol*
	

Figure 5.14 CMOS Transmission Gate

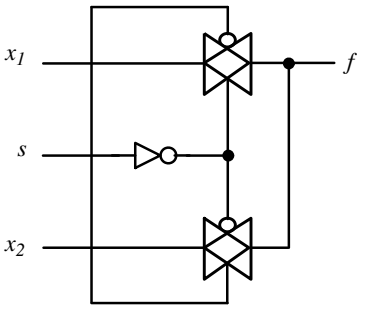


Figure 5.15 Implementation of a 2 : 1 Multiplexer Using CMOS Transmission Gates

when s is equal to 0 and x_2 to f when s is equal to 1. This selector function, referred to as a 2 : 1 multiplexer, is shown in Figure 5.15.

5.8.2 XOR Gate

The circuit diagram of a two-input XOR gate is shown in Figure 5.16. The number of transistors required to implement this function is greatly reduced using CMOS transmission gates than using the CMOS or NMOS technologies. The output f is set

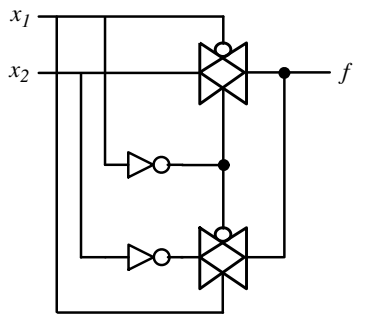


Figure 5.16 XOR Gate Using Transmission Gates

to the value of x_2 when x_1 is equal to 0 by the top transmission gate. On the other hand, the bottom transmission gate sets the output f to the complement value of x_2 when x_1 is equal to 1.

PROBLEMS

5.1 Implement a two-input AND gate using CMOS transistors and calculate the number of transistors required.

5.2 Implement a two-input OR gate using CMOS transistors and calculate the number of transistors required.

5.3 Implement a three-input NAND gate using CMOS transistors and calculate the number of transistors required.

5.4 Implement a three-input NOR gate using CMOS transistors and calculate the number of transistors required.

5.5 Implement a three-input AND gate using CMOS transistors and calculate the number of transistors required.

5.6 Implement a three-input OR gate using CMOS transistors and calculate the number of transistors required.

5.7 Determine the output logic function $f(x_1, x_2, x_3)$ of the three-input CMOS logic circuit in Figure P5.7.

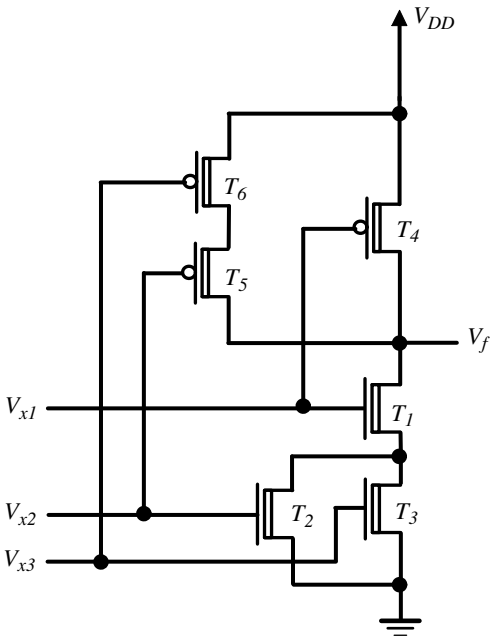


Figure P5.7

5.8 Implement the following logic functions using CMOS transistors and calculate the number of transistors required.

(a) $f(x_1, x_2) = \overline{x_1 \oplus x_2}$

(b) $f(x_1, x_2, x_3) = x_1 + x_2 x_3$

(c) $f(x_1, x_2, x_3) = (x_1 + x_2)x_3$

(d) $f(x_1, x_2, x_3, x_4) = (x_1 + x_2) + \bar{x}_1 x_3$

(e) $f(x_1, x_2, x_3, x_4) = \overline{x_1 + x_2 x_3 + x_4}$

(f) $f(x_1, x_2, x_3, x_4) = \overline{x_1 x_2 + x_3 x_4}$

5.9 Determine the number of transistors in the logic circuit in Figure P5.9 if the gates were implemented in CMOS technology.

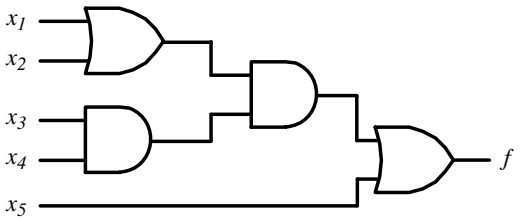


Figure P5.9

5.10 Determine the number of transistors in the logic circuit in Figure P5.10 if the gates were implemented in CMOS technology.

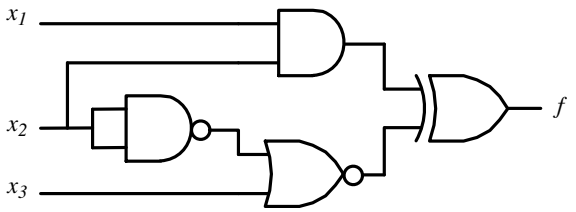


Figure P5.10

5.11 Determine the number of transistors in the logic circuit in Figure P5.11 if the gates were implemented in CMOS technology.

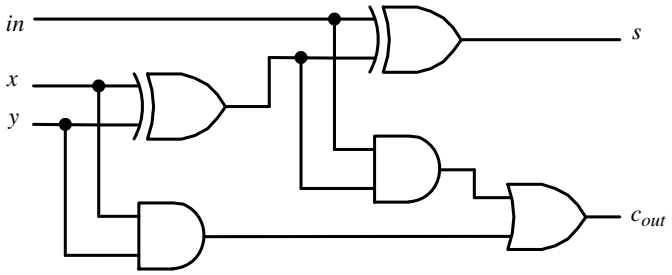


Figure P5.11

5.12 Determine the number of transistors in the logic circuit in Figure P5.12 if the gates were implemented in CMOS technology.

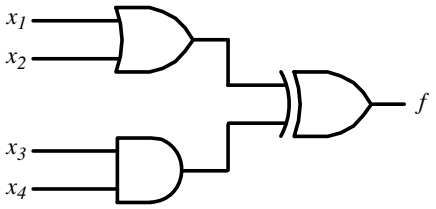


Figure P5.12

5.13 Determine the number of transistors in the logic circuit in Figure P5.13 if the gates were implemented in CMOS technology.

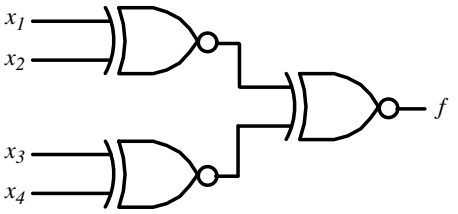


Figure P5.13

5.14 Determine the output waveform of the two-input NAND gate in Figure P5.14. Assume that the NAND gate has a propagation delay of 5 ns. The timing diagram has a 5-ns resolution.

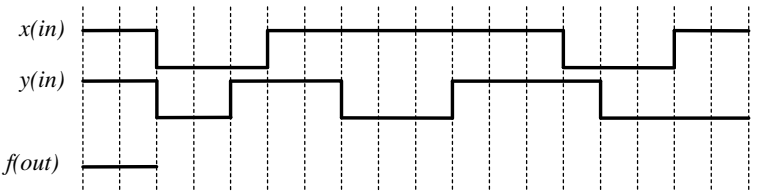


Figure P5.14

5.15 Determine the output waveform of the two-input NAND gate in Figure P5.14. Assume that the NAND gate has a propagation delay of 10 ns.

5.16 Write VHDL code to implement a two-input NAND gate. Assume that the NAND gate has a propagation delay of 10 ns.

5.17 Determine the output waveform of the two-input XOR gate in Figure P5.17. Assume that the XOR gate has a propagation delay of 5 ns. The timing diagram has a 5-ns resolution.

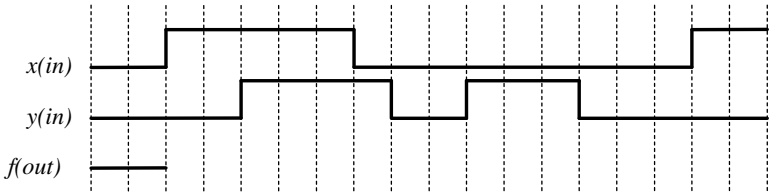
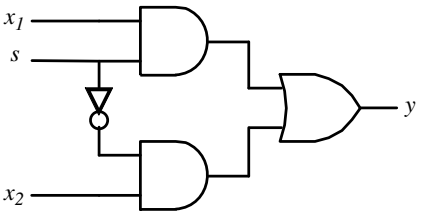


Figure P5.17

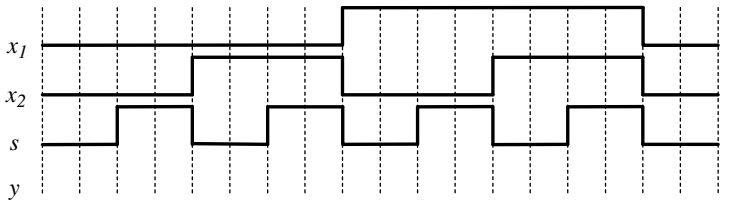
5.18 Determine the output waveform of the two-input XOR gate in Figure P5.17. Assume that the XOR gate has a propagation delay of 10 ns.

5.19 Write VHDL code to implement a two-input XOR gate. Assume that the XOR gate has a propagation delay of 10 ns.

5.20 Determine the output waveform of the logic circuit in Figure P5.20(a). Assume that all gates have a propagation delay of 5 ns. The timing diagram in Figure P5.20(b) has a 5-ns resolution.



(a)

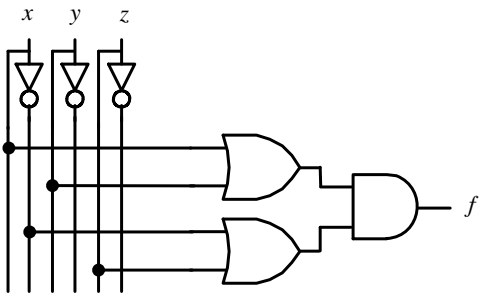


(b)

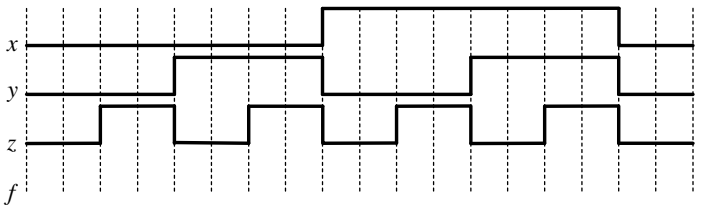
Figure P5.20

5.21 Write VHDL code to implement the logic circuit in Figure P5.20(a). Assume that all gates have a propagation delay of 10 ns.

5.22 Determine the output waveform of the logic circuit in Figure P5.22(a). Assume that all gates have a propagation delay of 5 ns. The timing diagram in Figure P5.22(b) has a 5-ns resolution.



(a)



(b)

Figure P5.22

5.23 Write VHDL code to implement the logic circuit in Figure P5.22(a). Assume that all gates have a propagation delay of 10 ns.

5.24 Write VHDL code to implement the logic circuit in Figure P5.24. Assume that all gates except the NOT gate have a propagation delay of 10 ns. The NOT gate has a propagation delay of 5 ns.

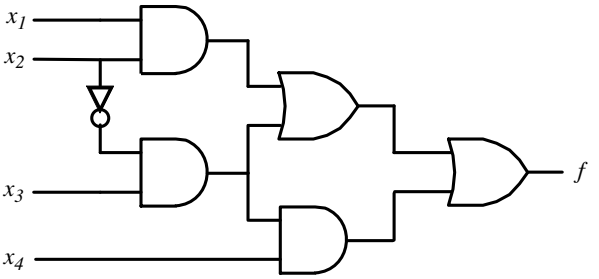


Figure P5.24

5.25 Write VHDL code to implement the logic circuit in Figure P5.25. Assume that all gates except the XOR gate have a propagation delay of 10 ns. The XOR gate has a propagation delay of 15 ns.

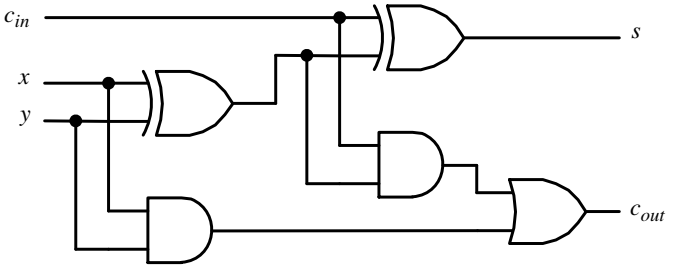


Figure P5.25

5.26 Draw a timing diagram for each of the following logic functions showing propagation delays. Assume a propagation delay of 1 ns through all NOT, AND, and OR logic gates. (*Hint*: Draw a timing diagram with 1-ns resolution.)

(a) $f(x_1, x_2) = (\bar{x}_1 + \bar{x}_2)(x_1 + \bar{x}_2)(\bar{x}_1 + x_2)$
(b) $f(x_1, x_2) = (x_1 + x_2)\,\overline{\bar{x}_1 + \bar{x}_2}$
(c) $f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2x_3$
(d) $f(x_1, x_2, x_3) = (\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + x_3)$

5.27 Repeat Problem 5.26 after simplifying the logic functions. Compare the resulting propagation delays to those of Problem 5.26.

6 Logic Function Optimization

6.1 OBJECTIVES

The objectives of the chapter are to:

- Describe logic function optimization
- Describe the Karnaugh map optimization method
- Describe the optimization of incomplete logic functions
- Provide design examples
- Describe the Quine–McCluskey optimization method

6.2 LOGIC FUNCTION OPTIMIZATION PROCESS

The process of digital design consists of three steps. The first step is to design a truth table according to specifications, then minimize the logical expression obtained from the truth table using optimization techniques, and finally, implement the minimized expression using logic gates. The minimization or optimization step is very important in both ASIC- and PLD-based designs. *Minimization* is the process of deriving the logical expression with a minimal number of literals, thereby reducing the number of gates and gate inputs and thus reducing the cost and chip area. By optimizing the logical expression, we minimize the number of inputs on first- and second-level gates. While minimizing any expression, we assume that both true and complemented versions of all inputs are available. There are different methods of optimizing logic expressions. The most popular ways to illustrate minimization steps are to use Karnaugh maps and prime implicant charts.

6.3 KARNAUGH MAPS

Earlier it was explained that the key to finding the minimum-cost expression for a given logic function is to reduce the number of product (or sum) terms needed in the expression by applying Boolean algebra theorems. *Karnaugh mapping* is a method

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

Row	*x*	*y*	*z*	*f*
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 6.1 Truth Table Sample

used to simplify a truth table using sum of products or product of sums along with simultaneous optimization of the output function. Karnaugh maps are the graphical equivalent of a truth table. In other words, Karnaugh maps are an easy way of designing and optimizing a circuit from a truth table. Consider the function f in Figure 6.1.

The canonical (not reduced) SOP expression for the logic function f consists of the minterms m_0, m_2, m_4, m_5, and m_6, and can be written

$$f = \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z}$$

Using Boolean algebraic manipulations, the logic function f can be simplified in the following steps:

$$\begin{aligned} f &= (\bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot y \cdot \bar{z}) + (x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot \bar{z}) + (x \cdot \bar{y} \cdot z + x \cdot \bar{y} \cdot \bar{z}) \\ &= \bar{x}(y + \bar{y})\bar{z} + x(y + \bar{y})\bar{z} + x \cdot \bar{y}(z + \bar{z}) \\ &= \bar{x} \cdot \bar{z} + x \cdot \bar{z} + x \cdot \bar{y} \\ &= (x + \bar{x})\bar{z} + x \cdot \bar{y} \end{aligned}$$

The minimum expression for the logic function f can be written

$$f = \bar{z} + x \cdot \bar{y}$$

The expression above can be checked by comparing it with the truth table. The expression has the product term $\bar{z}$ because $f = 1$ when $z = 0$, regardless of the values x or y. Ignoring z and just looking at x and y, the remaining 1 occurs when $x = 1$ and $y = 0$. Hence, the final portion of the minimum expression is realized. This pattern may not be recognized until the Boolean algebra reduction theorems are implemented. However, using these Boolean algebra theorems in the process of reducing the logical expression to the minimum expression can become tedious when there are more variables.

The Karnaugh map method is an alternative to the truth table form for representing a function. The map consists of cells that correspond to the rows of a truth table. Consider the two-variable Karnaugh map in Figure 6.2, which shows the truth table form, where each of the four rows is identified by a minterm.

x_1	x_2	f
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

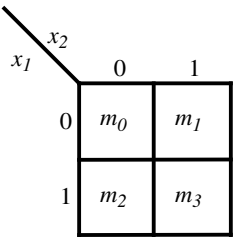


Figure 6.2 Two-Variable Karnaugh Map

It also shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_2, and x_1 labels the rows. This configuration creates a table that houses the locations of the minterms, as shown in Figure 6.2. Karnaugh maps have the advantage of allowing easy recognition of minterms that can be combined using the properties of Boolean algebra. Minterms in any two cells that are adjacent, either row or column, can be combined to produce a simplified minterm.

6.4 TWO-VARIABLE KARNAUGH MAP

An example of a Karnaugh map for a two-variable function is shown in Figure 6.3. Each value of the truth table is represented in the Karnaugh map. A 1 appears in both columns of the top row. Therefore, there exists a single product term that can cause f to be equal to 1 when the input variables have values that correspond to either of these cells. These values have been circled and are identified as $x_1 = 0$, but x_2 equals 0 for the left column and 1 for the right. This implies that if $x_1 = 0$, then $f = 1$, regardless of the value of x_2. The product term represented by this circle is simply $\bar{x}_1$. Similarly,

x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1

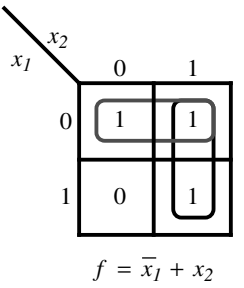


Figure 6.3 Karnaugh Map Simplification of a Two-Variable Logic Function

if x_2 is 1, then regardless of x_1, the function f will also equal 1. Hence, the minimum realization for the logic function f can be expressed as

$$f = \bar{x}_1 + x_2$$

Algebraic simplification yields the same logic expression. Therefore, to find a minimum implementation for a given logic function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases. At the same time, the number of these product terms should be as low as possible. Notice how some of the 1's are used more than once because a product term that covers two adjacent cells is cheaper than a single-cell product term to implement.

6.5 THREE-VARIABLE KARNAUGH MAP

Karnaugh maps can be modified to handle a greater number of inputs. For example, combining two two-variable maps together can create a three-variable Karnaugh map. Figure 6.4 shows a three-variable truth table and a three-variable Karnaugh map. Here x_1 and x_2 identify the rows of the map and x_3 identifies the columns. To assure that all the minterms in the adjacent cells of the map can be combined into a single product term, the adjacent cells must differ by only one bit position. As you may notice, the values of x_1 and x_2 count in the order 00, 01, 11, 10 rather than the usual 00, 01, 10, 11. This ensures that each cell varies by only one bit position from each adjacent cell. The map also wraps around itself, so the top and bottom cells are also adjacent to each other. The cell adjacency of a Karnaugh map obeys the Gray code, which consists of a sequence of code where each value differs by only one bit position at a time.

x_1	x_2	x_3	f
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

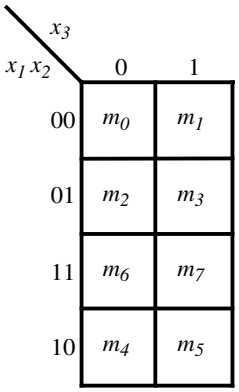


Figure 6.4 Three-Variable Karnaugh Map

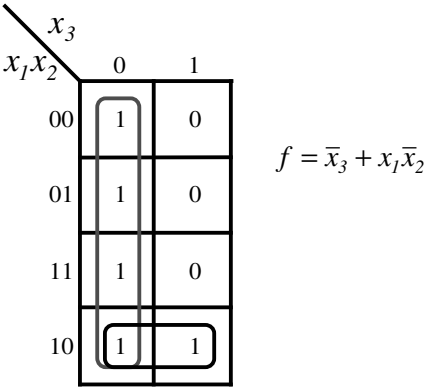


Figure 6.5 Karnaugh Map Simplification of a Three-Variable Logic Function

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. An example of this case is shown in Figure 6.5, which is the Karnaugh map that represents the truth table in Figure 6.1. The four cells in the left column correspond to (x_1, x_2, x_3) 000, 010, 110, and 100. In this case, $f = 1$ when $x_3 = 0$, regardless of the values of x_1 or x_2 and hence the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, which can be represented by $x_1 \cdot \bar{x}_2 \cdot x_3$. But if the adjacent cell is included, the final value can be reduced further, to $x_1 \cdot \bar{x}_2$. The complete realization of f can be expressed as

$$f = \bar{x}_3 + x_1 \cdot \bar{x}_2$$

which is similar to the result found by algebraic simplification in Section 6.3. It is possible to have all the cells in the Karnaugh map set to 1, which would permit every cell in the map to be enclosed within one circle. This is a trivial case where f always equals 1 no matter what the input is.

6.6 FOUR-VARIABLE KARNAUGH MAP

A four-variable map is constructed similarly to the three-variable Karnaugh map, but now two three-variable maps are combined. Figure 6.6 shows the structure and minterm locations of a four-variable Karnaugh map. Notice that the values of x_1 and x_2 and of x_3 and x_4 count in the order 00, 01, 11, 10 rather than the usual 00, 01, 10, 11. This ensures that each cell varies by only one bit position from each adjacent cell. The map also wraps around itself, so the top and bottom cells are adjacent to each other. The cells of the rightmost column are adjacent to those in the leftmost column of the map. The four corner cells are also adjacent and form a group of four minterms.

Figure 6.7 shows an example of a four-variable function reduced in a four-variable Karnaugh map. Notice that just as the top and bottom edges of a map are adjacent,

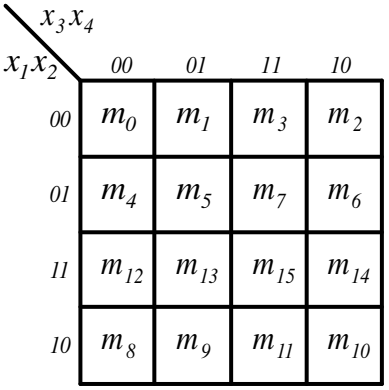


Figure 6.6 Four-Variable Karnaugh Map

so are the right and left edges. This is shown by the 1's in the four corners of the map, which can be enclosed within one circle, forming a group of four 1's. The rest of the 1's are then enclosed within the other circles to complete the function as shown in Figure 6.7.

In all previous examples, the Karnaugh maps had unique solutions for the function *f*. However, other choices are sometimes available to minimize the function. In this case the solution is not unique. Such an example is shown in Figure 6.8. The groups of four 1's in the top-left and bottom-right corners of the map are realized by the terms $\bar{x}_1\bar{x}_3$ and x_1x_3, respectively. This leaves the two 1's, which correspond to the term $x_1x_2\bar{x}_3$. But these two 1's can be realized more efficiently by treating them as a group of four 1's instead. As shown in the map, the last 1's can be grouped in two different ways. One choice leads to the product term x_1x_2 while the other leads to $x_2\bar{x}_3$. Both are valid and produce a function with the same minimum implementation.

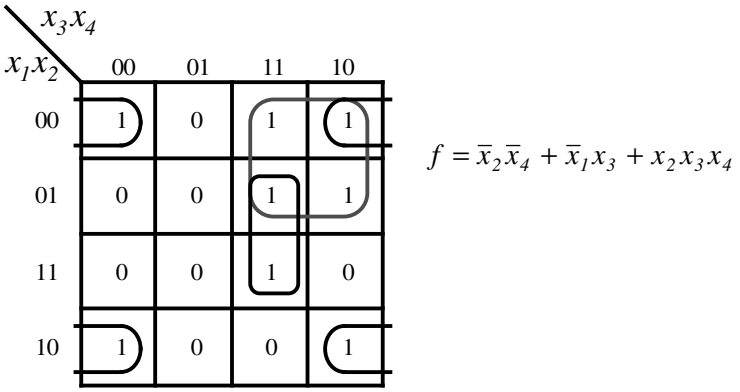


Figure 6.7 Karnaugh Map Simplification of a Four-Variable Logic Function

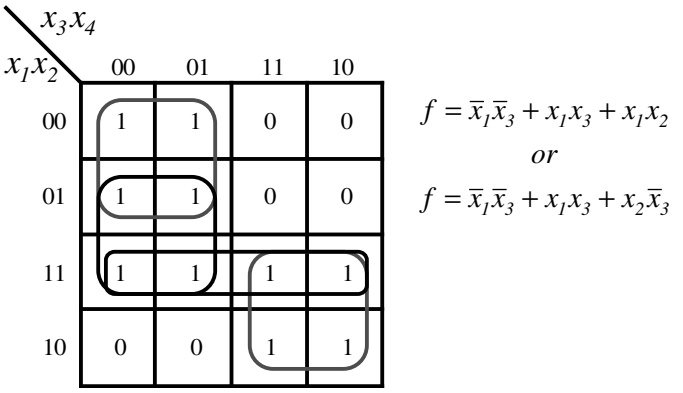


Figure 6.8 Multiple Karnaugh Map Solutions

6.7 FIVE-VARIABLE KARNAUGH MAP

A five-variable Karnaugh map can be constructed using two four-variable Karnaugh maps. Imagine a map like those drawn previously, with an identically sized map stacked directly on top of the other. The two halves are distinguished by the fifth variable, which can be equal to 0 or 1. Since it is difficult to draw a three-dimensional object on two-dimensional paper, a five-variable Karnaugh map can have both parts drawn side by side as illustrated in Figure 6.9.

The logic function mapped into the Karnaugh map of Figure 6.9 has five variables. The two groups of four 1's shown in the upper right corners of each map section can be circled as one group and denote the product term $\bar{x}_1 x_3$. Since the 1's appear in both maps when $x_5 = 0$ and $x_5 = 1$, they do not depend on x_5. The same is true for the pair of

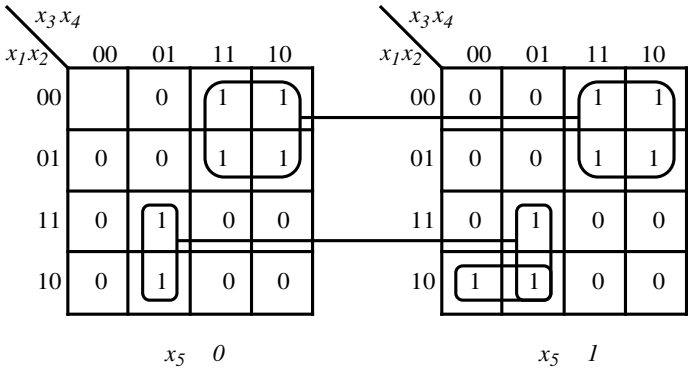


Figure 6.9 Karnaugh Map Simplification of a Five-Variable Logic Function

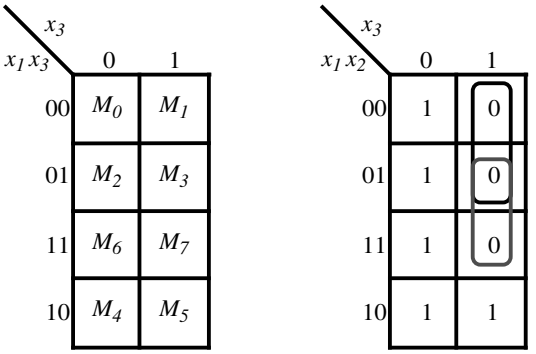


Figure 6.10 Karnaugh Map Simplification of a Two-Variable POS Logic Function

1's in the vertical line in both maps. The last 1 appears only in the $x_5 = 1$ and hence gives the largest product term, as seen in the final function of f.

All the Karnaugh map examples illustrated earlier used the sum-of-products (SOP) form, which considers the cells of values equal to 1. It is also possible to synthesize f by considering the values that make $f = 0$. This can easily be done using Karnaugh maps. The method is the same, but now we use the cells of value 0 in the Karnaugh map rather than the cells with 1's. For example, let us use the same three-variable Karnaugh map as that used in Figure 6.5, but now use the 0's in the map instead. The corresponding Karnaugh map is illustrated in Figure 6.10. The cells of the Karnaugh map contain the maxterms. Therefore, the method used to write in products-of-sums (POS) form is used to write the function of the Karnaugh map. The same Karnaugh rules apply to maxterms to simplify the logic function.

6.8 XOR AND NXOR KARNAUGH MAPS

Certain patterns for a Karnaugh map should be recognized readily. These patterns are the XOR and NXOR logic functions, which can reduce greatly the implementation of the logic function. Figure 6.11 shows patterns of Karnaugh maps that result in a XOR logic expression. Similarly, Figure 6.12 shows the equivalent NXOR Karnaugh maps. NXOR are the complement of the XOR Karnaugh maps.

6.9 INCOMPLETE LOGIC FUNCTIONS

In digital systems there are certain input conditions for which a specific function can never occur or possibly would simply be unimportant. For example, suppose that you have two interlocking switches, A and B, such that both switches cannot be closed at the same time. Therefore, there are only three possible states for the switches. Switch A is open while switch B is closed, switch A is closed while

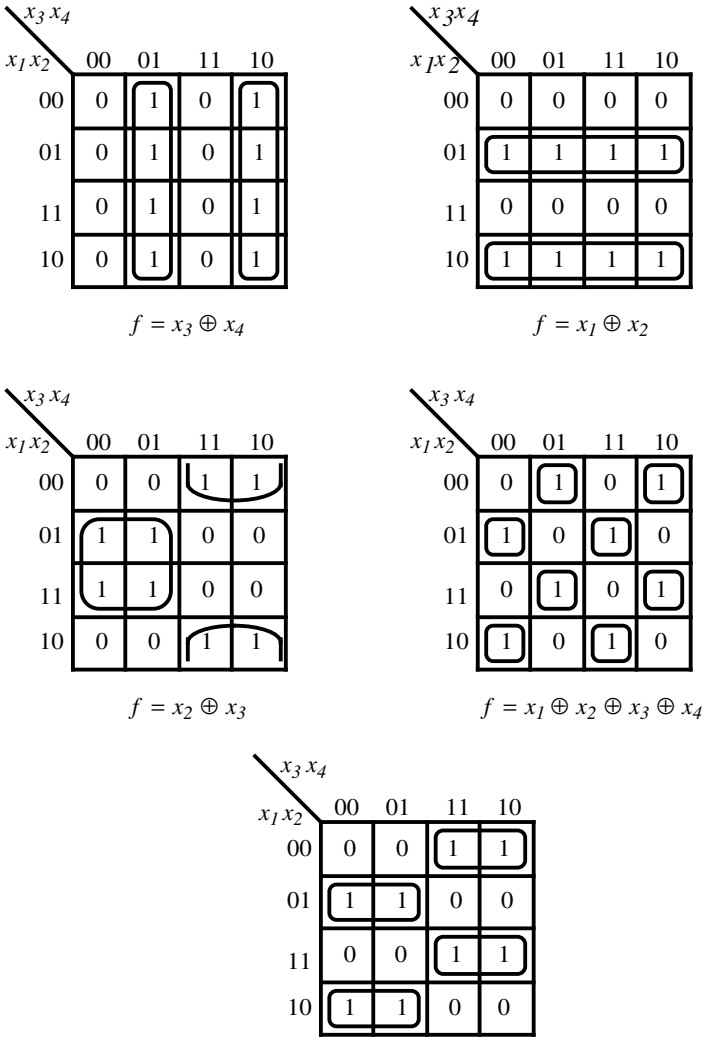


Figure 6.11 Karnaugh Map Patterns for Various XOR Logic Expressions

switch B is open, or switches B and A are both open. In digital terms, the inputs are 10, 01, or 00. But the value 11 will never occur. The value that will never occur is called a *don't-care condition*. Any function with don't-care conditions is said to be specified incompletely. Don't-care states can be an advantage in designing logic circuits. Since a don't care will never occur, the designer can assume that the event is either 1 or 0, whichever helps minimize the cost. Consider the Karnaugh map example in Figure 6.13, which includes don't-care states. Notice that not all the *d*'s are used in the Karnaugh map. Since the don't-care states can be either 1 or 0, they are used to realize minimum sum-of-products or product-of-sums functions.

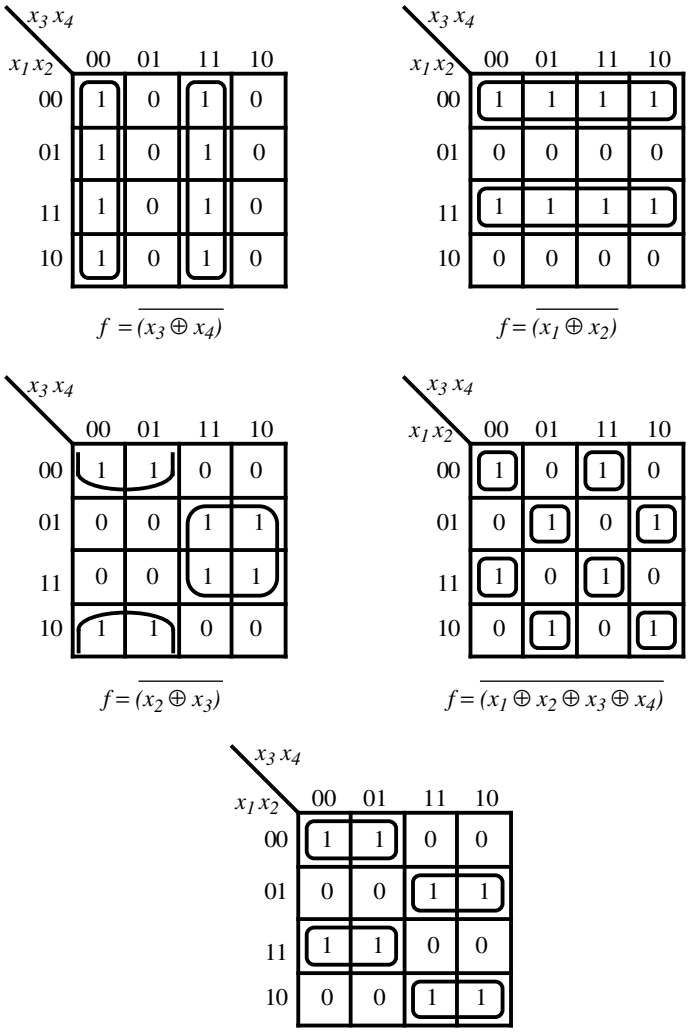
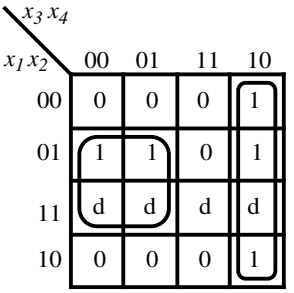


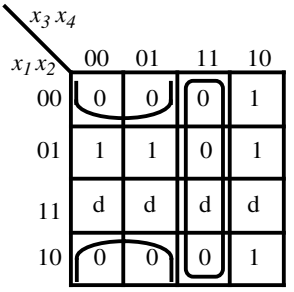
Figure 6.12 Karnaugh Map Patterns for Various NXOR Logic Expressions

6.10 QUINE–McCLUSKEY MINIMIZATION

The Karnaugh map method is used to minimize logic functions of up to five variables. For logic functions with more than five variables, the Karnaugh map method becomes impractical. The Karnaugh method uses maps, which become very difficult to design as the number of input variables increases. Pattern recognition of adjacent cells becomes tedious or impossible. An alternative method is the Quine–McCluskey method. Quine–McCluskey is based on the same basic principles of the Karnaugh map method. It uses an adjacency theorem to reduce minterms for which the logic



SOP Implementation
$f = x_2\overline{x}_3 + x_3\overline{x}_4$



POS Implementation
$f = (x_2 + x_3) \cdot (\overline{x}_3 + \overline{x}_4)$

Figure 6.13 Karnaugh Maps with Don't-Care States for Both SOP and POS Design

function is equal to 1. Unlike a Karnaugh map, the Quine–McCluskey method uses tables, which provide simple pattern recognitions.

Quine–McCluskey method is a tabular method that has an advantage over Karnaugh maps when a large number of inputs are present. With more inputs, pattern recognition in Karnaugh maps can be tedious or sometimes even impossible. The Quine–McCluskey method does not require pattern recognition. It consists of two steps: (1) finding all prime implicants of the function, and (2) selecting a minimal set of prime implicants of the function. An *implicant* is the product of some of the variables (in complemented or uncomplemented form) for which the function is equal to 1. Minterms for which the function is equal are considered implicants. Product terms that result from minterm simplifications are also implicants. An implicant is called a *prime implicant* if it cannot be combined with other implicant or implicants to form a reduced product of variables. This is similar to combining overlapping or adjacent groupings of minterms to form the largest grouping (power of 2) of adjacent cells in the Karnaugh map method. If a prime implicant includes a minterm that does not belong to any other prime implicants, it is called an *essential prime implicant*. Any primary implicant that is not an essential prime implicant is a *secondary prime implicant*.

In the first phase of Quine–McCluskey simplification, a prime implicant table is constructed using the following steps:

1. List minterms in a column using their binary representation.
   (a) Group minterms into groups containing the same number of 1's.

(b) Place groupings differing by one literal adjacent to one another (literals are another name for the variables of the function).

2. Search for logically adjacent terms between adjacent groups.
   (a) Combine each pair of terms into a single term by replacing the differing literal with "-."
   (b) Repeat until no further columns can be created.
3. Construct a prime implicant chart with minterms listed horizontally, prime implicants vertically, and place an × wherever a prime implicant covers a minterm.
4. Select all essential prime implicants.
5. Select a minimal cover from the remaining prime implicants by eliminating rows covered by another row and columns covered by another column.
6. Include don't cares in the prime implicant table but not in the prime implicant chart.

Similar to the Karnaugh map method, the Quine–McCluskey method can also be applied to POS design and incomplete logic functions. The following example illustrates the process of Quine–McCluskey minimization. Consider the logic function defined by the following implicit SOP expression:

$$f = \sum m(1, 3, 7, 14, 15)$$

First, we design the prime implicant table shown in Figure 6.14. The second column of the table consists of SOP minterms, which are grouped accordingly to the number of complemented variables in the minterms. The third column consists of reduced implicants by combining implicants from the first column. The same process will be duplicated if further combinations of the reduced implicants are possible; otherwise, the process terminates. Prime implicants are indicted by an asterisk (*).

Once no further logic reduction is possible, the implicants, which are not reduced, constitute the prime implicants. To identify essential prime implicants and secondary prime implicants, another table is generated, which lists the prime implicants versus the SOP minterms as shown in Figure 6.15. An × mark is checked if the primary

Number of Uncomplemented Variables	*Implicants (SOP Minterms)*	*Simplification First Round*
1	0001√	0001 with 0011=>00-1*
2	0011√	0011 with 0111=>0-11*
3	0111√	0111 with 1111=>-111*
	1110√	1110 with 1111=>111-*
4	1111√	

Figure 6.14 Prime Implicant Table

Minterms Covered by Prime Implicants		1	3	7	14	15
1,3	00-1	⊗	x			
3,7	00-1		x	x		
7,15	-111			x		x
14,15	111-				⊗	x

Figure 6.15 Prime Implicant Chart

implicant covered the corresponding minterm. A prime implicant which covers a minterm that is not covered by any other prime implicant, an essential prime implicant, is indicated by a circled $\times$ ($\otimes$). The remaining prime implicants are secondary prime implicants. Minterm columns, for which minterms are covered by essential prime implicants, are removed from the table. The essential prime implicants will be included in the simplified logic function. From the remaining secondary prime implicants, only those that cover all the remaining minterms with the fewest secondary prime implicants will be selected. The number of secondary prime implicants that belong to the same column may provide alternative solutions. Whenever possible, only one secondary prime implicant should be considered from a single column. Therefore, the final simplified expression of the logic function is

$$f = x_1x_2x_3 + x_2x_3x_4 + \bar{x}_1\bar{x}_2x_4$$

PROBLEMS

6.1 Using Karnaugh maps, simplify the following implicit SOP logic functions.

(a) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_7)$
(b) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_3, m_5)$
(c) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_3, m_7)$
(d) $f(x_1, x_2, x_3) = \sum(m_0, m_3, m_5, m_6)$
(e) $f(x_1, x_2, x_3) = \sum(m_0, m_1, m_2, m_5, m_7)$
(f) $f(x_1, x_2, x_3, x_4) = \sum(m_2, m_3, m_6, m_8, m_9, m_{12})$
(g) $f(x_1, x_2, x_3, x_4) = \sum(m_0, m_2, m_6, m_8, m_{10}, m_{14})$
(h) $f(x_1, x_2, x_3, x_4) = \sum(m_0, m_5, m_6, m_7, m_{14}, m_{15})$
(i) $f(x_1, x_2, x_3, x_4) = \sum(m_5, m_6, m_7, m_{13}, m_{14}, m_{15})$
(j) $f(x_1, x_2, x_3, x_4) = \sum(m_7, m_{10}, m_{11}, m_{13}, m_{14}, m_{15})$
(k) $f(x_1, x_2, x_3, x_4) = \sum(m_2, m_4, m_8, m_8, m_9, m_{10}, m_{14})$

6.2 Using Karnaugh maps, simplify the following implicit POS logic functions.

(a) $f(x_1, x_2, x_3) = \Pi(M_3, M_4, M_6)$
(b) $f(x_1, x_2, x_3) = \Pi(M_2, M_4, M_5, M_6)$
(c) $f(x_1, x_2, x_3) = \Pi(M_1, M_2, M_4, M_7)$

(d) $f(x_1, x_2, x_3, x_4) = \Pi(M_3, M_6, M_7, M_{11}, M_{13})$
(e) $f(x_1, x_2, x_3, x_4) = \Pi(M_1, M_3, M_4, M_6, M_9, M_{11})$
(f) $f(x_1, x_2, x_3, x_4) = \Pi(M_3, M_6, M_7, M_{11}, M_{13}, M_{15})$
(g) $f(x_1, x_2, x_3, x_4) = \Pi(M_1, M_3, M_5, M_{12}, M_{14}, M_{15})$
(h) $f(x_1, x_2, x_3, x_4) = \Pi(M_3, M_6, M_7, M_{11}, M_{12}, M_{14})$
(i) $f(x_1, x_2, x_3, x_4) = \Pi(M_0, M_1, M_5, M_8, M_9, M_{13}, M_{15})$
(j) $f(x_1, x_2, x_3, x_4) = \Pi(M_2, M_3, M_6, M_{10}, M_{12}, M_{13}, M_{14})$
(k) $f(x_1, x_2, x_3, x_4) = \Pi(M_0, M_1, M_2, M_4, M_6, M_8, M_{10}, M_{12})$

6.3 Using Karnaugh maps, simplify the following incomplete SOP logic functions.

(a) $f(x_1, x_2, x_3, x_4) = \sum(m_3, m_4, m_{13}, m_{15}) + d(1, 7, 14)$
(b) $f(x_1, x_2, x_3, x_4) = \sum(m_1, m_7, m_{13}, m_{14}) + d(3, 4, 15)$
(c) $f(x_1, x_2, x_3, x_4) = \sum(m_1, m_4, m_{14}, m_{15}) + d(0, 5, 8, 9)$
(d) $f(x_1, x_2, x_3, x_4) = \sum(m_4, m_9, m_{12}) + d(1, 3, 6, 11, 14)$
(e) $f(x_1, x_2, x_3, x_4) = \sum(m_3, m_6, m_{12}) + d(0, 5, 9, 10, 15)$
(f) $f(x_1, x_2, x_3, x_4) = \sum(m_7, m_8, m_9, m_{11}, m_{15}) + d(0, 1, 4, 5)$
(g) $f(x_1, x_2, x_3, x_4) = \sum(m_0, m_2, m_6, m_7) + d(8, 9, 10, 11, 13)$
(h) $f(x_1, x_2, x_3, x_4) = \sum(m_1, m_4, m_{14}) + d(0, 2, 5, 8, 10, 15)$
(i) $f(x_1, x_2, x_3, x_4) = \sum(m_1, m_4, m_8, m_{10}, m_{15}) + d(0, 2, 5, 9, 13)$
(j) $f(x_1, x_2, x_3, x_4) = \sum(m_2, m_6, m_8, m_{13}, m_{15}) + d(0, 5, 7, 10, 14)$

6.4 Using Karnaugh maps, simplify the following incomplete POS logic functions.

(a) $f(x_1, x_2, x_3, x_4) = \Pi(M_1, M_7, M_{11}, M_{13}, M_{14}) + d(2, 4, 8)$
(b) $f(x_1, x_2, x_3, x_4) = \Pi(M_2, M_7, M_{10}, M_{15}) + d(0, 5, 8, 13)$
(c) $f(x_1, x_2, x_3, x_4) = \Pi(M_5, M_9, M_{13}) + d(1, 2, 6, 10, 14)$
(d) $f(x_1, x_2, x_3, x_4) = \Pi(M_{10}, M_{11}, M_{13}, M_{14}) + d(0, 1, 4, 7)$
(e) $f(x_1, x_2, x_3, x_4) = \Pi(M_0, M_2, M_5, M_6, M_8) + d(9, 10, 11, 12)$
(f) $f(x_1, x_2, x_3, x_4) = \Pi(M_0, M_{11}, M_{13}, M_{15}) + d(2, 3, 5, 6, 7)$
(g) $f(x_1, x_2, x_3, x_4) = \Pi(M_1, M_2, M_3, M_4, M_5, M_6) + d(0, 8, 9, 10, 12)$
(h) $f(x_1, x_2, x_3, x_4) = \Pi(M_2, M_4, M_9, M_{11}, M_{12}) + d(0, 1, 3, 8, 10, 13)$

6.5 Implement the following logic function using only NAND gates.

(a) $f(x_1, x_2) = x_1 \oplus x_2$
(b) $f(x_1, x_2, x_3) = (x_1 + x_2)(x_1 + x_3)$
(c) $f(x_1, x_2, x_3) = x_3 + x_1 x_2$
(d) $f(x_1, x_2, x_3) = x_1 x_2 + (\bar{x}_1 + x_3)$
(e) $f(x_1, x_2, x_3) = (\bar{x}_1 + x_2)(x_2 + \bar{x}_3)$
(f) $f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2 + x_2\bar{x}_3 x_4 + x_1 x_4 + \bar{x}_2\bar{x}_4$
(g) $f(x_1, x_2, x_3, x_4) = (x_2 + x_4)(\bar{x}_1 + \bar{x}_2 + x_3 + x_4)(x_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$

(h) $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2)(x_3 \oplus x_4)$

(i) $f(x_1, x_2, x_3, x_4) = \overline{x_1 \oplus x_2} \oplus \overline{x_3 \oplus x_4}$

6.6 Implement the following logic functions using only NOR gates.

(a) $f(x_1, x_2) = x_1 \oplus x_2$

(b) $f(x_1, x_2, x_3) = (x_1 + x_2)(x_1 + x_3)$

(c) $f(x_1, x_2, x_3) = x_3 + x_1x_2$

(d) $f(x_1, x_2, x_3) = x_1x_2 + (\bar{x}_1 + x_3)$

(e) $f(x_1, x_2, x_3) = (\bar{x}_1 + x_2)(x_2 + \bar{x}_3)$

(f) $f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2 + x_2\bar{x}_3x_4 + x_1x_4 + \bar{x}_2\bar{x}_4$

(g) $f(x_1, x_2, x_3, x_4) = (x_2 + x_4)(\bar{x}_1 + \bar{x}_2 + x_3 + x_4)(x_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$

(h) $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2)(x_3 \oplus x_4)$

(i) $f(x_1, x_2, x_3, x_4) = \overline{x_1 \oplus x_2} \oplus \overline{x_3 \oplus x_4}$

6.7 Design a logic circuit to detect whether a 4-bit binary number is an even number.

6.8 Design a logic circuit to detect whether a 4-bit binary number is an odd number.

6.9 Design a logic circuit to detect whether a 4-bit binary number has even parity. A binary number with even parity has an even number of 1's.

6.10 Design a logic circuit to detect whether a 4-bit binary number has odd parity. A binary number with odd parity has an odd number of 1's.

6.11 Consider a logic function f with the three variables x_1, x_2, and x_3. The function f is equal to 1 if and if only two variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.12 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any odd number of variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.13 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any even number of variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.14 Consider a logic function f with the three variables x_1, x_2, and x_3. The function f is equal to 1 if ($x_1 = 1$ and $x_2 = x_3$) or if ($x_1 = 0$ and $x_2 \neq x_3$); otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.15 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if any two variables or more are equal to 1; otherwise, the function f is equal to zero with the following exception. The function f is also equal to zero if x_1 is equal to zero and any two other variables are equal to 1.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.16 Consider a logic function f with the four variables x_1, x_2, x_3, and x_4. The function f is equal to 1 if two or more variables are equal to 1; otherwise, the function f is equal to zero.

(a) Draw a truth table for the function f.

(b) Using Karnaugh maps, simplify the function f.

(c) Draw a logic circuit that implements the function f.

6.17 Design a logic circuit that implements the truth table of a 2-bit adder (Figure P6.17). The 2-bit adder adds a 2-bit binary number (x_2x_1) to a 2-bit binary number (y_2y_1) and outputs a 3-bit binary number result ($s_3s_2s_1$).

x_2	x_1	y_2	y_1	s_3	s_1	s_1
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Figure P6.17

6.18 Design a logic circuit that implements the truth table of a 2-bit subtractor (Figure P6.18). The 2-bit subtractor subtracts a 2-bit number binary (y_2y_1) from a 2-bit binary number (x_2x_1) and outputs a 3-bit binary number ($s_3s_2s_1$).

x_2	x_1	y_2	y_1	s_3	s_1	s_1
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	1	0	1	1	0
0	0	1	1	1	0	1
0	1	0	0	0	0	1
0	1	0	1	0	0	0
0	1	1	0	1	1	1
0	1	1	1	1	1	0
1	0	0	0	0	1	0
1	0	0	1	0	0	1
1	0	1	0	0	0	0
1	0	1	1	1	1	1
1	1	0	0	0	1	1
1	1	0	1	0	1	0
1	1	1	0	0	0	1
1	1	1	1	0	0	0

Figure P6.18

6.19 Design a logic circuit that implements the truth table of a 2-bit multiplier (Figure P6.19). The 2-bit multiplier multiplies a 2-bit binary number (x_2x_1) by a 2-bit binary number (y_2y_1) and outputs a 4-bit binary number result ($p_4p_3p_2p_1$).

x_2	x_1	y_2	y_1	p_4	p_3	p_1	p_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Figure P6.19

6.20 Design a logic circuit that implements the truth table of a BCD-to-excess-3 code converter (Figure P6.20).

BCD Numbers				*Excess-3 Code Numbers*			
x_3	x_2	x_1	x_0	y_3	y_1	y_1	y_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Figure P6.20

6.21 Design a logic circuit that implements the truth table of BCD-to-Gray code converter (Figure P6.21).

BCD Numbers				*Gray Code Numbers*			
x_3	x_2	x_1	x_0	y_3	y_1	y_1	y_0
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	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

Figure P6.21

6.22 Design a logic circuit to determine whether the 2-bit binary number x_2x_1 is equal to, greater than, or less than the 2-bit binary number y_2y_1. [*Hint*: Design a truth table, that has four inputs (x_1, x_2, y_1, y_2) and three outputs: E (equal to), G (greater than), and L (less than).]

6.23 Using the Quine–McCluskey method, simplify the implicit SOP functions for Problem 6.1.

6.24 Using the Quine–McCluskey method, simplify the implicit POS functions for Problem 6.2.

6.25 Using the Quine–McCluskey method, simplify the implicit incomplete SOP functions for Problem 6.3.

6.26 Using the Quine–McCluskey method, simplify the implicit incomplete POS functions for Problem 6.4.

7 Combinational Logic

7.1 OBJECTIVES

The objectives of the chapter are to describe:

- Combinational logic circuits
- Multiplexers and demultiplexers
- Decoders and encoders
- Code converters
- Arithmetic circuits
- Comparison circuits
- VHDL codes for basic combinational circuits

7.2 COMBINATIONAL LOGIC CIRCUITS

Building a digital system using the methods learned thus far is very possible, although very unrealistic. For simplicity, consider the NAND gate as the primitive element to design all other logic functions. A microcomputer chip such as the Motorola 68000 device comprising the equivalent of some 70,000 gates would require some 17,500 integrated-circuit (IC) packages. Designing such a complex digital system at the gate level would be a very difficult, if not impossible task. An alternative design method is to use a combination of gates as building blocks, referred to as *combinational circuits*.

The primary tools required to build combinational circuits include truth table design, basic knowledge of Boolean algebra, and implementation using logic gates. In some special cases the specification is given in the form of a complete truth table, but most often the procedure of designing a combinational logic circuit will start with a method to determine a truth table from a verbal or written statement, which a customer provides to the designer. A Boolean expression describes the circuit behavior required as expressed by the truth table. It is helpful to convert a Boolean expression with an arbitrary mixture of operations (AND, OR, and NOT) to a form, which is easier to implement with a combinational logic circuit.

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition.
Mohammed Ferdjallah.

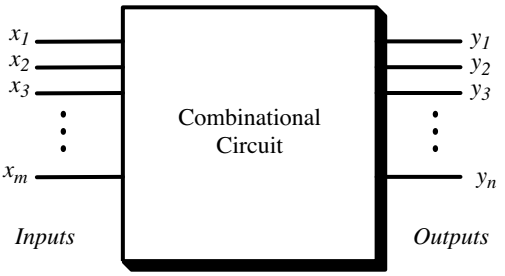


Figure 7.1 Block Diagram of a Combinational Circuit

Combinational logic is probably the easiest circuitry to design. The outputs from a combinational logic circuit depend only on the current inputs. In other words, a combinational circuit is a set of interconnected gates whose output at any time is a function of the input at that time. The appearance of input is followed almost immediately by output, with only gate propagation delays.

On the other hand, sequential circuits are composed of combinational circuits and memory elements with a set of m inputs and a set of n outputs. At any given time, an output is a function of the sequence of the inputs, implying that an output is a function not only of the current input but also of the inputs in the past, which need to be remembered by the memory elements. Thus, a sequential system is allowed to have feedback paths from the output of memory element to the input. Figure 7.1 shows a general schematic of a combinational circuit. The inputs of a combinational circuit include data inputs and control or status inputs. The control inputs generally specify how the data inputs affect the output of the combinational circuit. Simple and often-used combinational logic circuits include multiplexers, decoders, encoders, and code converters.

7.3 MULTIPLEXERS

A *multiplexer* (or "mux") is a digital switch that has 2^M data inputs, M select (control) inputs, and a single output. It routes data from one of 2^M data inputs to its single output. Figure 7.2 shows the graphical symbol of a $2^M : 1$ (pronounced "2^M to 1") multiplexer. The select input lines control which data input is connected to the output. Thus, a multiplexer acts as a programmable digital switch.

7.3.1 2 : 1 Multiplexer

A 2 : 1 multiplexer has two data inputs, one select input, and a single output. The function of a 2 : 1 multiplexer is described by the truth table shown in Figure 7.3. The figure also shows the logic implementation of the 2 : 1 multiplexer. The same circuit can be realized using transmission gates, as described in Section 3.9. The VHDL code, which implements a 2 : 1 multiplexer, is illustrated in Figure 7.4.

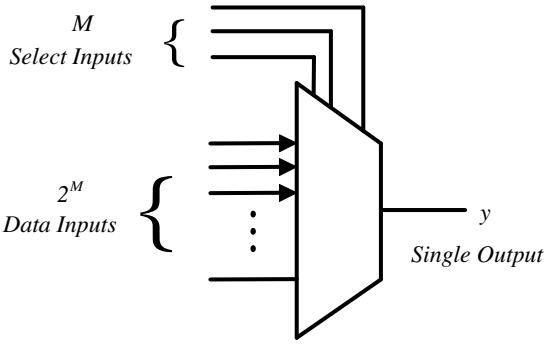


Figure 7.2 Graphical Symbol of a 2^M : 1 Multiplexer

Truth Table				*Logic Implementation*
s	x_2	x_1	y	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	
1	0	1	0	
1	1	0	1	
1	1	1	1	

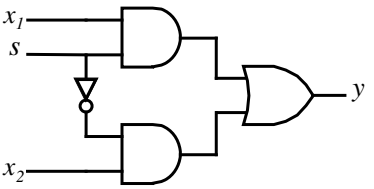


Figure 7.3 2 : 1 Multiplexer

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux2to1 is
      port(
            x1,x2,s          :   in    std_logic;
            f                :   out   std_logic);
end mux2to1;
architecture circuit_behavior of mux2to1 is
      begin
          with s select
            f    <=    x1    when     '0';
                       x2    when others;
end circuit_behavior;
```

Figure 7.4 VHDL Code for a 2 : 1 Multiplexer Using Select Signal Assignment

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux2to1 is
     port(
          x1,x2,s          :   in    std_logic;
          f                :   out   std_logic);
end mux2to1;
architecture circuit_behavior of mux2to1 is
     begin
          f   <=   x1   when   s='0'   else   x2;
end circuit_behavior;
```

Figure 7.5 VHDL Code for a 2 : 1 Multiplexer Using Conditional Signal Assignment

The clause **with–select–when** is used as a select signal assignment to switch between the two inputs.

A second VHDL code implementation of the 2 : 1 multiplexer, which uses the conditional signal assignment clause **when–else**, is illustrated in Figure 7.5. A third VHDL code implementation of the 2 : 1 multiplexer, which uses the **if–else–then** conditional statement, is illustrated in Figure 7.6. Notice that the **if–else–then** statement is a sequential conditional statement and thus a **process** statement is required. The inputs of the multiplexer are included in the sensitivity list of the **process** statement for its implicit activation/deactivation.

A fourth VHDL code implementation of the 2 : 1 multiplexer, which uses the **case** statement, is illustrated in Figure 7.7. Notice that a **process** statement is required since the **case** statement is also a sequential conditional statement. The inputs of the

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux2to1 is
     port(
          x1,x2,s          :   in    std_logic;
          f                :   out   std_logic);
end mux2to1;
architecture circuit_behavior of mux2to1 is
     begin
       process (x1,x2,s)
         begin
           if   s='0'   then;
                f <= x1;
           else
                f <= x2;
           end if;
       end process;
end circuit_behavior;
```

Figure 7.6 VHDL Code for a 2 : 1 Multiplexer Using an **If–Else–Then** Statement

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux2to1 is
     port(
          x1,x2,s          :   in    std_logic;
          f                :   out   std_logic);
end mux2to1;
architecture circuit_behavior of mux2to1 is
     begin
        process (x1,x2,s)
          begin
            case s is;
                when      '0'    => f <= x1;
                when others      => f <= x2;
            end case;
        end process;
end circuit_behavior;
```

Figure 7.7 VHDL Code for a 2 : 1 Multiplexer Using a **Case** Statement

multiplexer are included in the sensitivity list of the process to provide an implicit wait for the **process** statement.

7.3.2 4 : 1 Multiplexer

A 4 : 1 multiplexer has four data inputs, two select inputs, and a single output. The function of the 4 : 1 multiplexer is described using a truth table and can be implemented using basic gates, as illustrated in Figure 7.8. The VHDL code that implements a 4 : 1 multiplexer is illustrated in Figure 7.9. The clause **with–select–when** is used as a select signal assignment to switch between the two inputs.

A larger multiplexer can be designed by direct implementation of the truth table of the multiplexer. However, a larger multiplexer could be designed using smaller

Truth Table			*Graphic Symbol*
s_1	s_0	y	
0	0	x_1	
0	1	x_2	
1	0	x_3	
1	1	x_4	

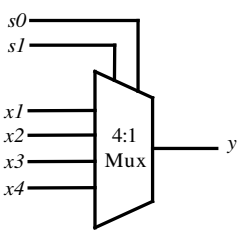


Figure 7.8 4 : 1 Multiplexer

```
library ieee ;
use ieee.std_logic_1164.all;

entity mux4to1 is
     port(
          x1,x2,x3,x4    :   in      std_logic;
          s              :   in      std_logic_vector(1 downto 0);
          f              :   out     std_logic);
end mux4to1;
architecture circuit_behavior of mux4to1 is
     begin
          with s select
            f   <=   x1   when    "00";
                     x2   when    "01";
                     x3   when    "10";
                     x4   when others;
end circuit_behavior;
```

Figure 7.9 VHDL Code for a 4 : 1 Multiplexer Using a Select Signal Assignment

multiplexers as modules. The data inputs of the smaller multiplexers are funneled down to a single output. For example, the 4 : 1 multiplexer can be implemented using three 2 : 1 multiplexers, as illustrated in Figure 7.10.

The data inputs of two 2 : 1 multiplexers are funneled down to the single output of the third multiplexer (Figure 7.5). The data inputs x_1 and x_2 are applied to Mux0, and the data inputs x_3 and x_4 are applied to Mux1. The outputs of Mux0 and Mux1 are applied to Mux3. To select the data inputs x_1 and x_2, the select inputs must be set to ($s_1 = 0$, $s_0 = x$). If s_0 is set to 0, the data input x_1 is routed to the output of Mux2. Otherwise ($s_0 = 1$), the data input x_2 is output. Similarly, if s_1 is set to 1, the data input x_3 or x_4 will be routed to the output of Mux2, depending on the value of s_0. This modular multiplexer is implemented in the VHDL code illustrated in Figure 7.11 using component declaration.

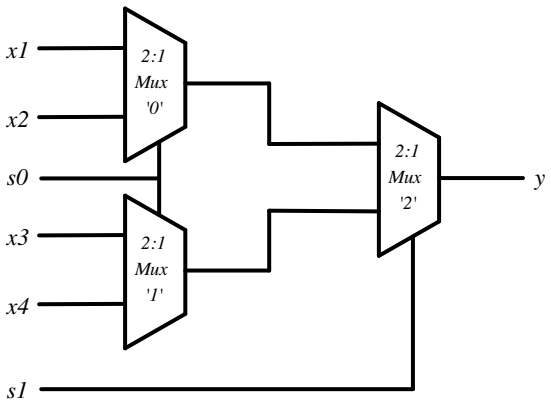


Figure 7.10 Implementation of a 4 : 1 Multiplexer Using Three 2 : 1 Multiplexers

```
library ieee ;
use ieee.std_logic_1164.all;
use work.mux2to1_package.all;

entity mux4to1 is
      port(x    :    in      std_logic_vector(0 to 3);
           s    :    in      std_logic_vector(1 downto 0);
           f    :    out     std_logic);
end mux4to1;
architecture circuit_behavior of mux4to1 is
      signal sig :   std_logic_vector(0 to 1)
      component mux2to1
           port(  x1,x2,s  :      in     std_logic;
                  f        :      out    std_logic);
      end component;
begin
      mux0 : mux2to1 port map (x(0),x(1),s(0),sig(0));
      mux1 : mux2to1 port map (x(2),x(3),s(0),sig(1));
      mux2 : mux2to1 port map (sig(0),sig(1),s(1),f);
end circuit_behavior;
```

Figure 7.11 VHDL Code of a 4 : 1 Multiplexer Using Component Declaration

7.4 LOGIC DESIGN WITH MULTIPLEXERS

The main function of a multiplexer is to route only one selected data input to its single output. Because of its unique structure, a multiplexer can also be used to implement a logic function. Consider the 4 : 1 multiplexer circuit illustrated in Figure 7.12. The truth table of the circuit identifies the function of the circuit as the AND logic function. In fact, any logic function with N variables can be implemented directly with a 2^N : 1 multiplexer. The data inputs of the multiplexer are the function values, and the select inputs are the variables of the function. However, complex functions make their direct implementation with multiplexers impractical. Fortunately, it is possible to design logic functions with smaller size multiplexers by identifying patterns between the

Truth Table		*Logic Function Implementation*
x_1	x_2	*f(x1,x2)*
0	0	0
0	1	0
1	0	0
1	1	1

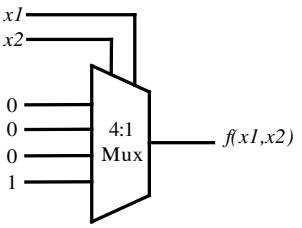


Figure 7.12 AND Logic Function Implementation with a 4 : 1 Multiplexer

Truth Table	*Logic Function Implementation*

x_1	*f(x1,x2)*
0	0
1	x_2

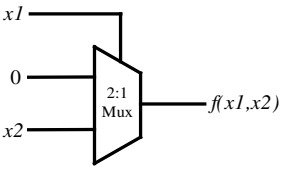


Figure 7.13 AND Logic Function Implementation with a 2 : 1 Multiplexer

Truth Table	*Logic Function Implementation*

x_1	x_2	*f(x1,x2)*
0	0	0
0	1	1
1	0	1
1	1	0

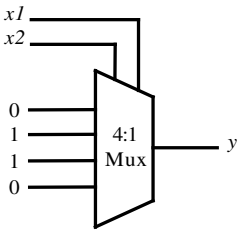


Figure 7.14 XOR Logic Function Implementation with a 4 : 1 Multiplexer

function and its variables in the truth table. Consider again the truth table in Figure 7.12. The truth table can be organized by selecting x_1 select inputs and relating the data input x_2 to the function as illustrated in Figure 7.13. Therefore, the AND function can be implemented using a 2 : 1 multiplexer with additional connections and logic gates when necessary.

Similarly, an XOR function can be implemented with a 4 : 1 multiplexer as illustrated in Figure 7.14. With simple modifications of the truth table of Figure 7.14, an XOR function could be implemented with a 2 : 1 multiplexer and additional gates as illustrated in Figure 7.15.

7.5 DEMULTIPLEXERS

The opposite of the multiplexer circuit, logically enough, is the *demultiplexer*. This circuit takes a single data input and one or more address inputs and selects which of multiple outputs will receive the input signal. The same circuit can also be used as a decoder by using the address inputs as a binary number and producing an output signal on the single output that matches the binary address input. In this application, the data

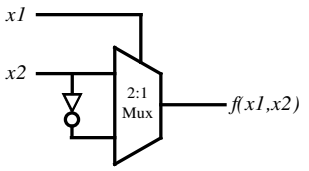
Truth Table	*Logic Function Implementation*
x_1 | $f(x1,x2)$ 0 | x_2 1 | $\overline{x}_2$	

Figure 7.15 XOR Logic Function Implementation with a 2 : 1 Multiplexer

input line functions as a circuit enabler. If the circuit is disabled no output will show activity regardless of the binary input number. A 1 : 2 decoder/demultiplexer circuit uses the same AND gates and the same addressing scheme as the 2 : 1 multiplexer circuit described earlier. The basic difference is that it is the inputs that are combined and the outputs that are separate. By making this change, a 1 : 2 demultiplexer circuit is the inverse of the 2 : 1 multiplexer circuit.

7.6 DECODERS

A *decoder* is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code, and there is one-to-one mapping from input code words into output code words. The general structure of a decoder circuit is shown in Figure 7.16. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. The most commonly used input code is an N-bit binary code, where an N-bit word represents one of 2^N different coded values. Normally, they range from 0 through $2^N - 1$. The input code lines select which output

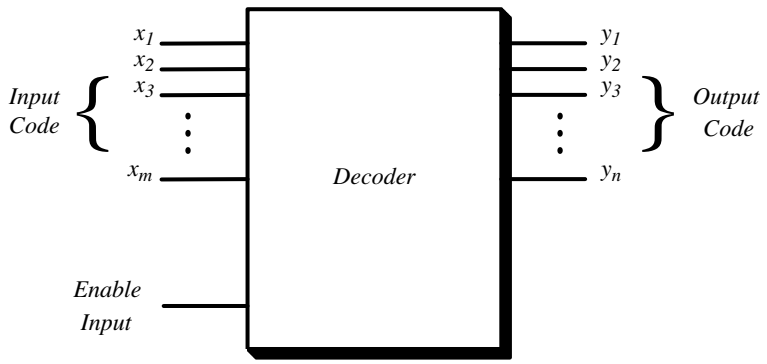
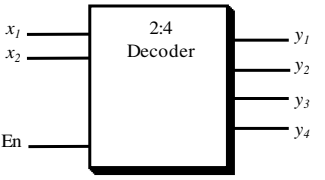


Figure 7.16 Block Diagram of a $N : 2^N$ Decoder

Truth Table

En	x_2	x_1	y_4	y_3	y_2	y_1
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Block Diagram



Logic Circuit Implementation

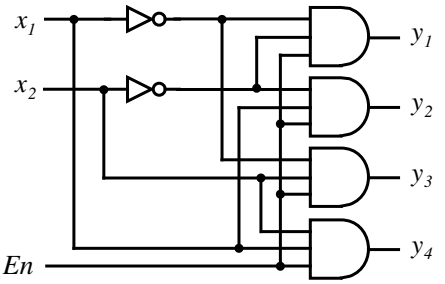


Figure 7.17 Logic Implementation of a 2 : 4 Decoder

is active. The remaining output lines are disabled. Thus, the decoder is intended to provide a binary code to other circuits, such as a memory circuit. In this case, the decoder is referred to as an address decoder because it selects one address of a memory location. However, a decoder could also be used to channel a stream of data on a designated output line selected by the input code lines.

A 2 : 4 decoder is illustrated in Figure 7.17. The two data inputs are x_1 and x_2. These inputs represent a 2-bit binary number that causes the decoder to assert one of the outputs y_1, y_2, y_3, and y_4. The decoders can be designed to have either active-high or active-low outputs. By setting inputs x_1 and x_2 to 00, 01, 10, or 11, it causes the output y_1, y_2, y_3, or y_4 to be set to 1, respectively. Truth table, graphical symbol, and logic circuit implementation of a 2 : 4 decoder are illustrated in Figure 7.17.

Similar to the modular design of larger multiplexers, large decoders can be constructed from smaller decoders. Figure 7.18 shows the implementation of a 3 : 8 decoder using two 2 : 4 decoders. The x_3 input drives the enable inputs of the two decoders. The decoder 0 is enabled if x_3 is equal to 0, and the decoder 1 is enabled if x_3 is set to 1. The VHDL code implementation of a 2 : 4 decoder is illustrated in Figure 7.19.

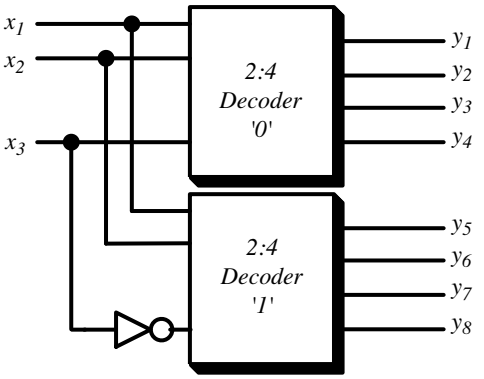


Figure 7.18 Implementation of a 3 : 8 Decoder Using Two 2 : 4 Decoders

```
library ieee ;
use ieee.std_logic_1164.all;

entity dec2to4 is
     port(
          x              :   in      std_logic_vector(1 downto 0);
          En             :   in      std_logic;
          y              :   out     std_logic_vector(0 to 3));
end dec2to4;
architecture circuit_behavior of dec2to4 is
         signal Enx     :       std_logic_vector(2 downto 0);
     begin
          Enx <= En& x;
          with Enx select
            y   <=    "1000"     when      "100";
                      "0100"     when      "101";
                      "0010"     when      "110";
                      "0001"     when      "111"
                      "0000"     when others;
end circuit_behavior;
```

Figure 7.19 VHDL Code of a 2 : 4 Decoder

7.7 ENCODERS

A decoder's output code normally has more bits than its input code. If the device's output code has fewer bits than the input code, the device is usually called an *encoder*. So an encoder performs the function opposite to that of a decoder. It encodes the given information into a more compact form. The most commonly used encoders are binary encoders and priority encoders.

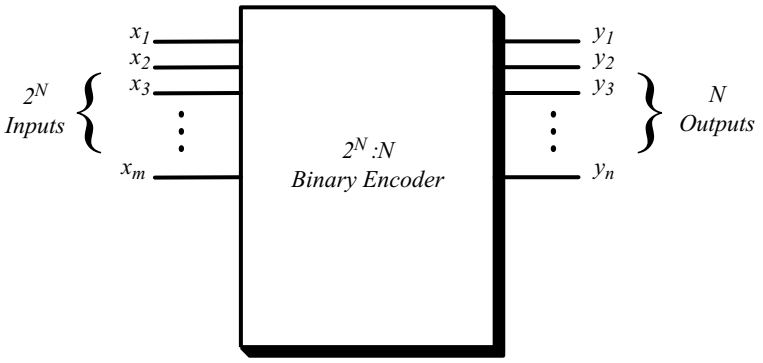


Figure 7.20 Block Diagram of a $2^N : N$ Binary Encoder

7.7.1 Binary Encoder

A *binary encoder* encodes information from 2^N inputs into an N-bit code. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. A binary encoder converts only one input at a time into a binary code. The general structure of a binary encoder circuit is shown in Figure 7.20. A 4 : 2 binary encoder, which has four inputs and two outputs, is illustrated in Figure 7.21. The modified truth table is shown in Figure 7.21. Notice that only one input is set at any time. All other states with multiple inputs, which are set, are considered don't-care states and are not shown in the truth table. Consequently, the logic circuit implementation of the 4 : 2 binary encoder has only two OR gates. Notice also that x_1 does not have any effect on either output y_1 or y_2.

7.7.2 Priority Encoder

A *priority encoder* is an encoder where more than one input can be activated simultaneously. Each input is assigned a priority order. When the input with the highest priority is asserted, the remaining inputs are ignored. A 4 : 2 priority encoder is illustrated in Figure 7.22. It is assumed that x_1 has the lowest priority and x_4 the highest. The outputs y_2 and y_1 represent the binary number that identifies the highest-priority input set to 1. An input condition marked with an $\times$ represents a don't-care state combination.

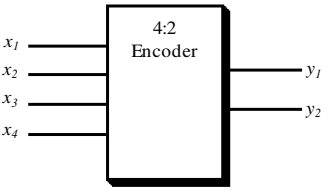
7.8 CODE CONVERTERS

Decoder and encoder circuits are used to convert from one type of input encoding to a different output encoding. For example, a 2 : 4 decoder converts a 2-bit binary number input to a one-hot encoding sequence (see Section 9.6) at the output. Similarly, a 4 : 2 binary encoder performs the opposite conversion. There are many other possible types of code converters known as BCD-to-seven-segment code converter, BCD-to-Gray code converter, BCD-to-excess-3 code converters, and so on.

Truth Table

x_4	x_3	x_2	x_1	y_2	y_1
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Block Diagram



Logic Circuit Implementation

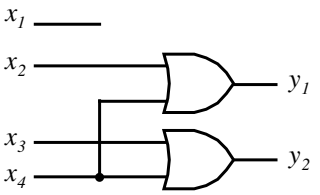


Figure 7.21 4 : 2 Binary Encoder

Truth Table

x_4	x_3	x_2	x_1	y_2	y_1
0	0	0	1	0	0
1	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

Logic Implementation

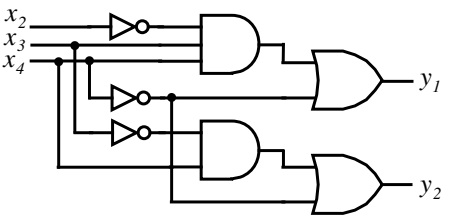


Figure 7.22 4 : 2 Priority Encoder

7.8.1 BCD-to-Seven-Segment Code Converter

A BCD-to-seven-segment code converter converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented display such as a seven-

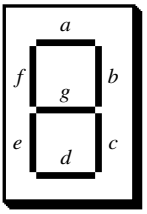
Truth Table											*7-Segment Display*
x_4	x_3	x_2	x_1	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	
0	0	0	1	0	1	1	0	0	0	0	
0	0	1	0	1	1	0	1	1	0	1	
0	0	1	1	1	1	1	1	0	0	1	
0	1	0	0	0	1	1	0	0	1	1	
0	1	0	1	1	0	1	1	0	1	1	
0	1	1	0	1	0	1	1	1	1	1	
0	1	1	1	1	1	1	0	0	0	0	
1	0	0	0	1	1	1	1	1	1	1	
1	0	0	1	1	1	1	1	0	1	1	

Figure 7.23 BCD-to-Seven-Segment Code Converter

segment display. In other words, it converts the BCD digit into seven signals that are used to drive the segments in the display. The circuit of the BCD-to-seven-segment code converter has four binary inputs (x_1, x_2, x_3, and x_4), representing the BCD number, and seven binary outputs (a, b, c, d, e, f, and g) representing the LED segments. Each segment is a small LED, which glows when driven by an electrical signal. The BCD-to-segment code conversion truth table and a seven-segment LED display are shown in Figure 7.23. The VHDL code in Figure 7.24 implements a BCD-to-seven-segment code converter.

7.8.2 BCD-to-Gray Code Converter

Gray code is an encoding scheme in which two consecutive binary numbers change by only 1 bit. Thus, a binary sequence with the property that only 1 bit changes between any two consecutive elements is called a Gray code. Each number in a Gray code sequence differs from its predecessor by exactly 1 bit. The circuit has four inputs (x_1, x_2, x_3, and x_4) representing the BCD number, and four outputs (y_1, y_2, y_3, and y_4) representing the 4-bit Gray code number. The truth table for a BCD-to-Gray code converter is shown in Figure 7.25. Gray code has applications in various fields, including data acquisition systems, communication coding, and mechanical position sensors. Mechanical encoders use Gray code to convert the angular position of a disk to digit form.

7.8.3 BCD-to-Excess-3 Code Converter

The BCD-to-excess-3 code converter converts a given BCD number into a number that can be obtained by adding the number three (0011) to it. Excess-3 code was

```
library ieee ;
use ieee.std_logic_1164.all;

entity bcd_to_7seg is
     port(
          bcd          :  in     std_logic_vector(3 downto 0);
          led_seg      :  out    std_logic_vector(1 to 7));
end bcd_to_7seg;
architecture circuit_behavior of bcd_to_7seg is
begin
     process (bcd)
     begin
        case bcd is
          when "0000" => led_seg  <= "1111110";
          when "0001" => led_seg  <= "0110000";
          when "0010" => led_seg  <= "1101101";
          when "0011" => led_seg  <= "1111001";
          when "0100" => led_seg  <= "0110011";
          when "0101" => led_seg  <= "1011011";
          when "0110" => led_seg  <= "1011111";
          when "0111" => led_seg  <= "1110000";
          when "1000" => led_seg  <= "1111111";
          when "1001" => led_seg  <= "1111011";
          when others  => led_seg  <= "-------";
        end case;
     end process
end circuit_behavior;
```

Figure 7.24 VHDL Code Implementation of the BCD-to-Seven-Segment Code Converter

used in older computer systems. An important characteristic of excess-3 code is its self-complementing property, similar to that of the 1's-complement method. The truth table and logic expressions of the BCD-to-excess-3 code converter are illustrated in Figure 7.26.

Truth Table								*Output Logic Expressions*
x_4	x_3	x_2	x_1	y_4	y_3	y_2	y_1	$y_1 = x_4x_1 + x_2\bar{x}_1 + x_3\bar{x}_2x_1$
0	0	0	0	0	0	0	0	$y_2 = x_3 \oplus x_2$
0	0	0	1	0	0	0	0	$y_3 = x_4 + x_3$
0	0	1	0	0	0	1	1	$y_4 = x_4$
0	0	1	1	0	0	1	0	
0	1	0	0	0	1	1	0	
0	1	0	1	0	1	1	1	
0	1	1	0	0	1	0	1	
0	1	1	1	0	1	0	0	
1	0	0	0	1	1	0	0	
1	0	0	1	1	1	0	1	

Figure 7.25 BCD-to-Gray Code Converter

Truth Table								*Output Logic Expressions*
x_4	x_3	x_2	x_1	y_4	y_3	y_2	y_1	$y_1 = \overline{x}_1$
0	0	0	0	0	0	1	1	$y_2 = \overline{(x_2 \oplus x_1)}$
0	0	0	1	0	1	0	0	$y_3 = \overline{x}_3 x_2 + \overline{x}_3 x_1 + \overline{x}_4 x_3 \overline{x}_2 \overline{x}_1$
0	0	1	0	0	1	0	1	$y_4 = x_4 + x_3 x_2 + x_3 x_1$
0	0	1	1	0	1	0	0	
0	1	0	0	0	1	1	1	
0	1	0	1	1	0	0	0	
0	1	1	0	1	0	0	1	
0	1	1	1	1	0	1	0	
1	0	0	0	1	0	1	1	
1	0	0	1	1	1	0	0	

Figure 7.26 BCD-to-Excess-3 Code Converter

7.9 ARITHMETIC CIRCUITS

Addition is the most commonly performed arithmetic operation in digital systems. An *adder* is a digital circuit that adds two *N*-bit numbers and generates an *N*-bit number. The adder circuit could also generate an overflow indication bit. The same adder circuit is used to combine two arithmetic operands, which can be unsigned or two's-complement numbers. Therefore, an adder can perform subtraction as the addition of the minuend and the complemented subtrahend. Direct implementation of an *N*-bit adder would be very complex and not scalable. Each time the size of the adder is changed, a new implementation must be designed. A simpler approach would be to build an *N*-bit adder from smaller module circuits, which can be duplicated and expanded as the size of the adder increases. The process of adding two *N*-bit numbers can be accomplished by a sequence of *N* simple 1-bit addition operations. In the following sections, several arithmetic circuit modules, which can be used to build larger arithmetic circuits, are described.

7.9.1 Half-Adder

A 1-bit half-adder adds two 1-bit operands, x and y, and produces a 2-bit result. The result can range from 0 to 2, which requires 2 bits to represent. The low-order bit of the sum is referred to as s (sum) and the high-order bit as c_{out} (carryout). The truth table and the optimized logic expressions of the 1-bit half-adder are illustrated in Figure 7.27. The logic implementation of a 1-bit half-adder is shown in Figure 7.28. A 1-bit half-adder can, however, only be used to add 1-bit numbers such as the LSB bits of two *N*-bit numbers. Therefore, a 1-bit half-adder cannot be used as a circuit module to build a larger adder.

Truth Table				*Output Logic Expressions*
x	***y***	c_{out}	***s***	$s = x \oplus y$ $c_{out} = x \cdot y$
0	0	0	0	
0	1	0	1	
1	0	0	1	
1	1	1	0	

Figure 7.27 Truth Table and Optimized Expressions of a 1-Bit Half-Adder

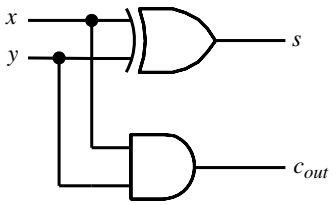


Figure 7.28 Logic Implementation of a 1-Bit Half-Adder

7.9.2 Full-Adder

When a carry-in bit is available, another 1-bit adder must be used, since a 1-bit half-adder does not take a carry-in (c_{in}) bit. A 1-bit full-adder adds three operands and generates a 2-bit result: the sum bit (s) and the carryout bit (c_{out}). The truth table and the minimized expressions of a 1-bit full-adder are illustrated in Figure 7.29. Using the minimized expressions in Figure 7.29, direct logic implementation of a 1-bit

Truth Table					*Output Logic Expressions*
x	***y***	c_{in}	c_{out}	***s***	$s = x \oplus y \oplus c_{in}$ $c_{out} = x \cdot y + x \cdot c_{in} + y \cdot c_{in}$
0	0	0	0	0	
0	0	1	0	1	
0	1	0	0	1	
0	1	1	1	0	
1	0	0	0	1	
1	0	1	1	0	
1	1	0	1	0	
1	1	1	1	1	

Figure 7.29 Truth Table and Minimized Expressions of a 1-Bit Full-Adder

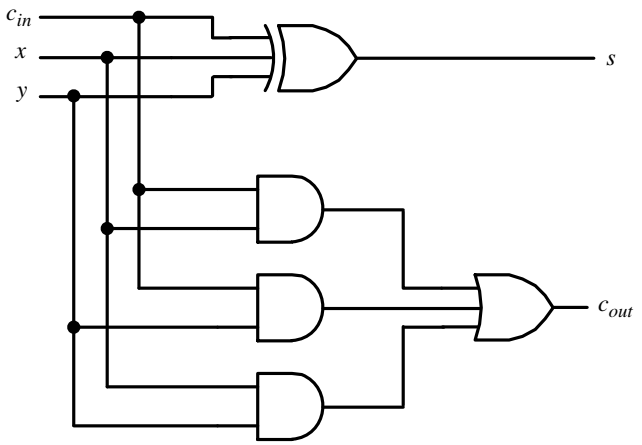


Figure 7.30 Logic Implementation of a 1-Bit Full-Adder

full-adder is shown in Figure 7.30. The VHDL code implementation of a 1-bit full-adder is illustrated in Figure 7.31. The logic implementation in Figure 7.30 requires one three-input XOR gate, three two-input AND gates, and one three-input OR gate in two-gate-level design. Using the truth table in Figure 7.29, the output logic expressions can be written as follows:

$$c_{\text{out}} = x \cdot y \cdot c_{\text{in}} + x \cdot \bar{y} \cdot c_{\text{in}} + \bar{x} \cdot y \cdot c_{\text{in}} + x \cdot y \cdot \bar{c}_{\text{in}}$$
$$c_{\text{out}} = x \cdot y + (x \oplus y) \cdot c_{\text{in}}$$
$$s = x \oplus y \oplus c_{\text{in}}$$

In the logic expressions above, one would recognize the logic expressions of a 1-bit half-adder. A 1-bit full-adder can be accomplished by cascading two 1-bit half-adders as illustrated in Figure 7.32. The final circuit has three-gate-level design.

```
library ieee ;
use ieee.std_logic_1164.all;

entity full_adder is
     port(
          x,y,cin          :   in    std_logic;
          s,cout           :   out   std_logic);
end full_adder;
architecture circuit_behavior of full_adder is
     begin
          s      <=  x xor y xor cin;
          cout   <=  (x and y) or (x and cin) or (y and cin);
end circuit_behavior;
```

Figure 7.31 VHDL Code Implementation of a 1-Bit Full-Adder

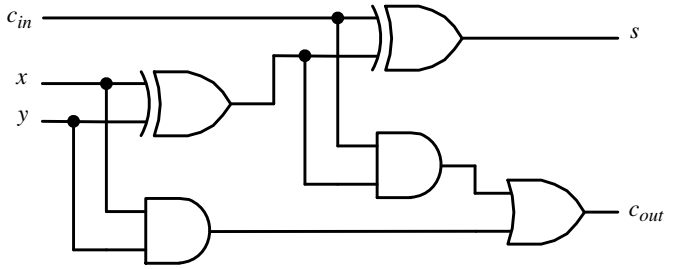


Figure 7.32 Logic Implementation of a 1-Bit Full-Adder Using 1-Bit Half-Adders

7.9.3 Half-Subtractor

Although binary subtraction can be accomplished using the same adding circuitry, specially designed subtractor circuits subtract two binary numbers and output a difference binary number and 1 bit borrow. In particular, the 1-bit half-subtractor is a combinational circuit which subtracts two inputs, x (minuend) and y (subtrahend), and generates two 1-bit outputs, d (difference) and b_{out} (borrow-out). The truth table and minimized logic expressions of a 1-bit half subtractor are illustrated in Figure 7.33. The logic implementation of 1-bit half-subtractor is shown in Figure 7.34.

Truth Table				***Output Logic Expressions***
x	***y***	b_{out}	***d***	$d = x \oplus y$
0	0	0	0	$b_{out} = \overline{x} \cdot y$
0	1	1	1	
1	0	0	1	
1	1	0	0	

Figure 7.33 Truth Table and Minimized Expressions of a 1-Bit Half-Subtractor

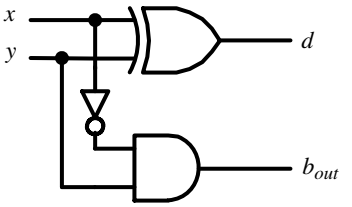


Figure 7.34 Logic Implementation of a 1-Bit Half-Subtractor

Truth Table					*Output Logic Expressions*
x	y	b_{in}	b_{out}	d	$d = x \oplus y \oplus c_{in}$
0	0	0	0	0	$b_{out} = \overline{x} \cdot y + \overline{x} \cdot b_{in} + y \cdot b_{in}$
0	0	1	1	1	
0	1	0	1	1	
0	1	1	1	0	
1	0	0	0	1	
1	0	1	0	0	
1	1	0	0	0	
1	1	1	1	1	

Figure 7.35 Truth Table and Minimized Expressions of a 1-Bit Full-Subtractor

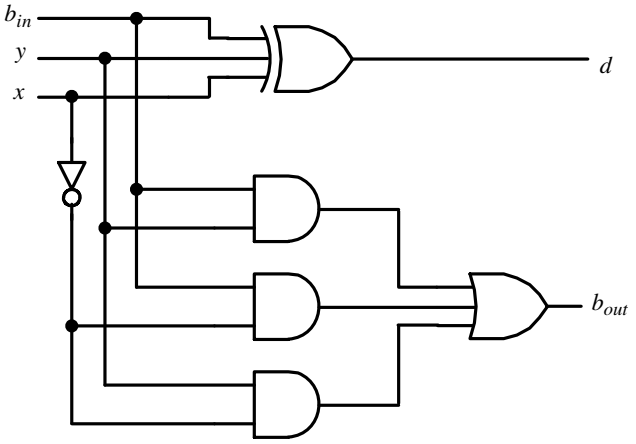


Figure 7.36 Logic Implementation of a 1-Bit Full-Subtractor

7.9.4 Full-Subtractor

A 1-bit full-subtractor is a combinational circuit that performs subtraction by using three bits: the minuend (x), the subtrahend (y), and the borrow-in (b_{in}). The truth table and the minimized expressions are shown in Figure 7.35. Using the minimized expressions in Figure 7.35, direct logic implementation of a 1-bit full-subtractor is shown in Figure 7.36. Similar to a 1-bit full-adder, the logic implementation of a 1-bit full-subtractor has two-gate-level design. Cascading two 1-bit half-subtractors can also result in a 1-bit full-subtractor. Notice again that the final circuit has three-gate-level design.

7.9.5 Ripple-Carry Adder

Several 1-bit full-adders can be cascaded to add numbers with several bits. Two binary numbers, each with N bits, can be added using a *ripple-carry adder*, a cascade

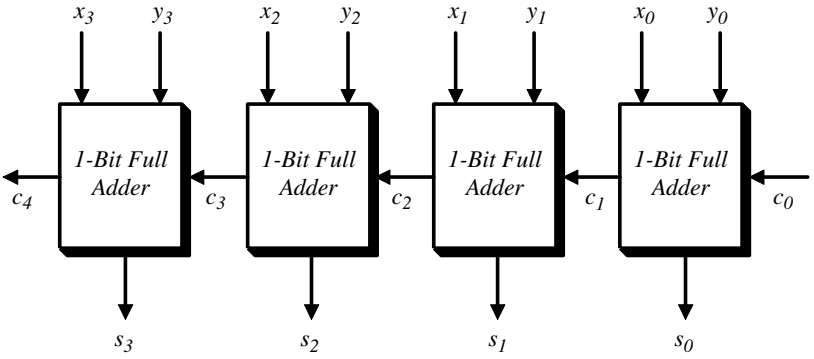


Figure 7.37 Four-Bit Ripple-Carry Adder

of N full-adder stages, each of which handles 1 bit. Figure 7.37 shows the circuit for a 4-bit ripple-carry adder. The carry-in to the least significant bit (c_0) is normally set to 0, and the carryout of each full-adder is connected to the carry-in of the next most significant full-adder. Because a full-adder stage had to wait for the carryout of the previous full-adder stage, a ripple-carry adder is generally slow. In the worst case a carry must propagate from the least significant full-adder stage to the most significant full-adder stage. Each full-adder introduces a certain propagation delay before its s_i and c_{i+1} outputs are valid. Let this delay be denoted Δt. Thus, the complete sum is available after a delay of $n\,\Delta t$. The propagation delay incurred to produce the final sum and carryout in a ripple-carry adder depends on the size of numbers. While the adders are working in parallel, the carry bits must "ripple" from the least significant bit and work their way to the most significant bit. Thus, the carry computation slows down the circuit, making the speed degradation of the addition operation linearly related to the bit-size number of the full ripple-carry adder.

It is not practical to have separate circuits for addition and subtraction operations, especially when the adder circuit could be modified to implement subtraction as well. Consider the logic circuit in Figure 7.38, which consists of a 4-bit ripple-carry adder with four XOR gates, each connected to the input of a 1-bit full-adder stage with c_0 as an input and y as the other input. If c_0 is equal to 0, the input y passes through without change. In this condition the circuit acts as an adder. However, if c_0 is equal to 1, the input y is inverted. Notice that in this case c_0 is equal to 1 and is applied to the first 1-bit full-adder stage, which effectively performs the two's complement of the input number y. Therefore, the circuit acts as a subtractor using the two's-complement method. VHDL code implementation of a 4-bit ripple-carry adder using component declaration is illustrated in Figure 7.39.

7.9.6 Carry Look-Ahead Adder

The ripple-carry adder performs arithmetic operations in a manner similar to those performed manually. To speed up the performance of a ripple-carry adder, the ripple effect of the carry must be eliminated. *Carry look-ahead adders* add much faster than

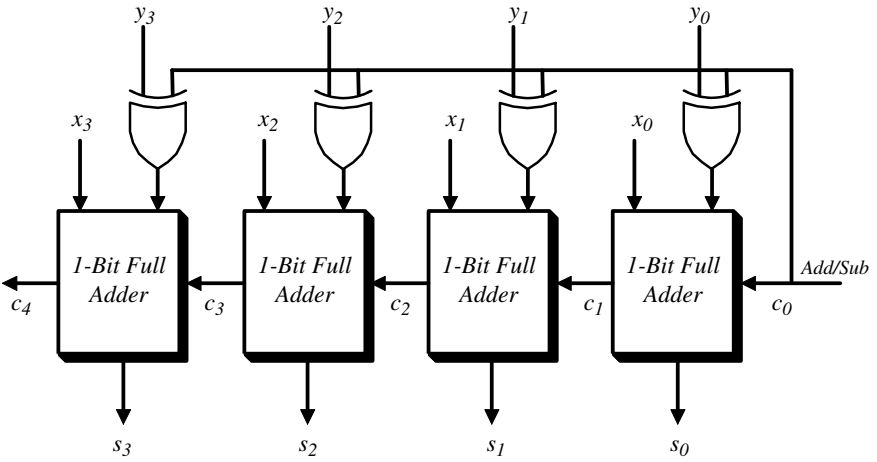


Figure 7.38 Four-Bit Ripple-Carry Adder/Subtractor

ripple-carry adders by computing the carry-in parallel. They are based on the fact that a carry signal will be generated in two cases: when bits A_i and B_i are both 1, or when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1. The carry look-ahead adder includes an additional circuit that computes the carryout bits at once, without waiting for the carry output from the preceding stage. The carry look-ahead circuit can be designed by analyzing the carry output logic expression. Consider

```
library ieee ;
use ieee.std_logic_1164.all;

entity rcadder_4bit is
     port(
          cin        :    in     std_logic;
          x,y        :    in     std_logic_vector(3 downto 0);
          s          :    out    std_logic_vector(3 downto 0);
          cout       :    out    std_logic);
end rcadder_4bit;
architecture ripple_behavior of rcadder_4bit is
     signal     c     :     std_logic_vector(1 to 3);
     component full_adder
          port(  cin,x,y   :   in      std_logic;
                 s,cout    :   out     std_logic);
     end component;
begin
       fulladder0 : full_adder port map (cin,x(0),y(0),s(0),c(1));
       fulladder1 : full_adder port map (c(1),x(1),y(1),s(1),c(2));
       fulladder2 : full_adder port map (c(2),x(2),y(2),s(2),c(3));
       fulladder3 : full_adder port map (c(3),x(3),y(3),s(3),cout);
end ripple_behavior;
```

Figure 7.39 VHDL Code Implementation of a 4-Bit Ripple-Carry Adder

the following logic expression of the carryout (c_{i+1}) from the ith bit addition:

$$c_{i+1} = x_i y_i + (x_i + y_i)c_i$$
$$c_{i+1} = g_i + p_i c_i$$

Here g_i and p_i are called the *generate* and *propagate* terms, respectively, and can be expressed as

$$g_i = x_i y_i$$
$$p_i = (x_i + y_i)$$

The propagate and generate terms depend only on the input bits. These terms will be used to compute the carryout directly without waiting for the previous carryout to ripple down. If the propagation delay through AND and OR gates is one gate delay, the propagate and generate terms will be available after one gate delay. A 4-bit carry look-ahead adder would require additional circuitry to implement the following logic expressions of the carryout bits:

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

The carryout bit c_4 of the last stage will be available after three delays. One gate delay to calculate the propagate and generate terms, and two additional delays for the additional AND and OR gates are required to compute c_4. The sum bit s_i can be calculated as

$$s_i = x_i \oplus y_i \oplus c_i$$

If we assume two gate delays through the XOR gate, the sum bit s_4 (for a 4-bit carry look-ahead adder) is available after a total of five gate delays after the input signals x_i and y_i have been applied. The delays to computer c_4 and s_4 are always the same and are independent of the number of bits and the size of the adder circuit. Figure 7.40 illustrates a 2-bit carry look-ahead adder.

Notice that as the size of the adder increases, the circuitry to compute the carryout becomes more complex. The resulting propagation delays overcome the assumptions made above, and the carry look-ahead adder becomes slower for larger numbers, despite their complexity. A combined architecture of carry look-ahead and ripple carry can be designed to get the best of both methods: simplicity and speed. A hybrid adder design is illustrated in Figure 7.41, which consists of four 8-bit carry look-ahead adders cascaded in a ripple-carry circuit to implement a 32-bit adder.

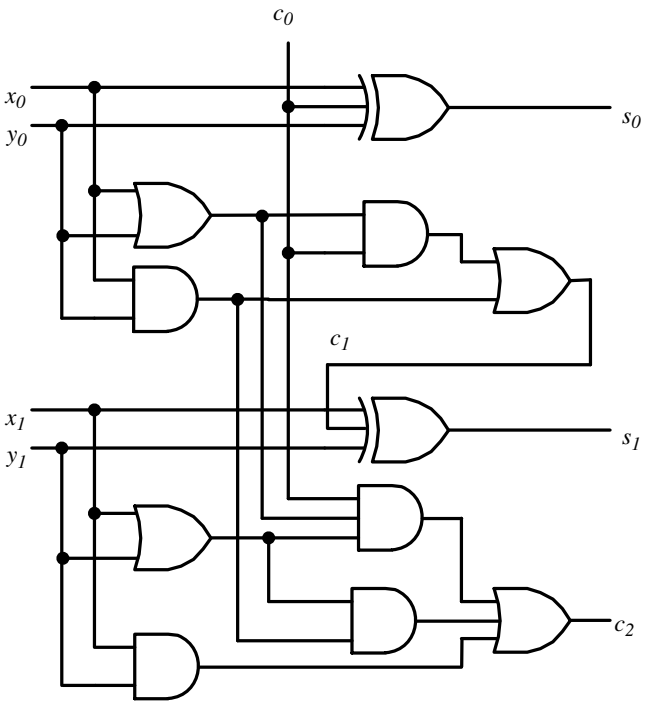


Figure 7.40 Two-Bit Carry Look-Ahead Adder

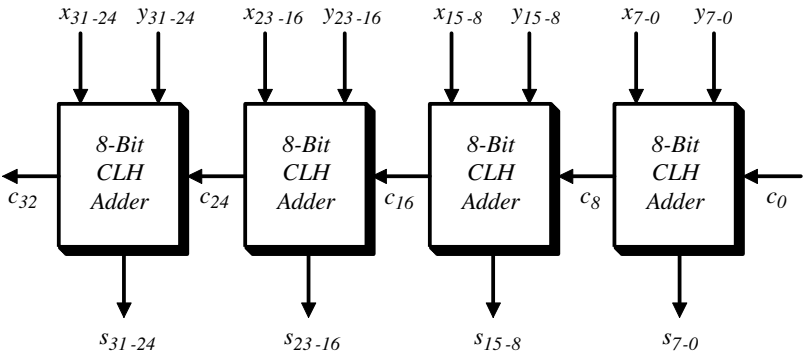


Figure 7.41 32-Bit Hybrid Adder

7.9.7 Comparison Circuits

Comparison circuits are logic circuits that determine if binary number inputs are equal to, greater than, or less than each other. Comparison circuits can be implemented directly from truth tables using SOP or POS methods. However, comparator circuits are best designed using logic expressions derived from simple logical observations.

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity compare_8bit is
     port(
          x,y          :     in    std_logic_vector(7 downto 0);
          Eq,Lg,Ls     :     out   std_logic);
end compare_8bit;
architecture circuit_behavior of compare_8bit is
     begin
          Eq   <= '1' when x = y else '0';
          Lg   <= '1' when x > y else '0';
          Ls   <= '1' when x < y else '0';
end circuit_behavior;
```

Figure 7.42 VHDL Code Implementation of an 8-Bit Comparator

Two binary numbers are equal if their corresponding bits are equal. To determine whether two bits are equal, one could XOR the two bits and monitor the output result. Thus, two bits are equal if the complement of their XOR function is equal to 1. Consider two 3-bit numbers, x ($x_3x_2x_1$) and y ($y_3y_2y_1$):

$$x = y \Rightarrow \overline{x_3 \oplus y_3} \cdot \overline{x_2 \oplus y_2} \cdot \overline{x_1 \oplus y_1} \quad \text{is} \quad \text{true}$$

To determine whether one binary number is greater than another, the most significant bits (MSBs) are tested first. If they were not equal, the number with an MSB of 1 is the larger number. If the MSBs are equal, the next bit positions are tested using the same process. The complement of the process is used to determine whether one binary number is less than another. For two 3-bit numbers x ($x_3x_2x_1$) and y ($y_3y_2y_1$), the greater than and less than operations are evaluated by the following logic expressions:

$$x > y \Rightarrow x_3\bar{y}_3 + \overline{x_3 \oplus y_3} \cdot x_2\bar{y}_2 + \overline{x_2 \oplus y_2} \cdot x_1\bar{y}_1 \quad \text{is} \quad \text{true}$$
$$x < y \Rightarrow \bar{x}_3y_3 + \overline{x_3 \oplus y_3} \cdot \bar{x}_2y_2 + \overline{x_2 \oplus y_2} \cdot \bar{x}_1y_1 \quad \text{is} \quad \text{true}$$

The logic expressions above can be used to implement a 3-bit comparator. The VHDL code illustrated in Figure 7.42 implements an 8-bit comparator.

PROBLEMS

7.1 Determine the logic functions of the multiplexer-based logic circuits in Figure P7.1(a) to (d).

7.2 Determine the logic functions of the multiplexer-based logic circuits in Figure P7.2(a) to (d).

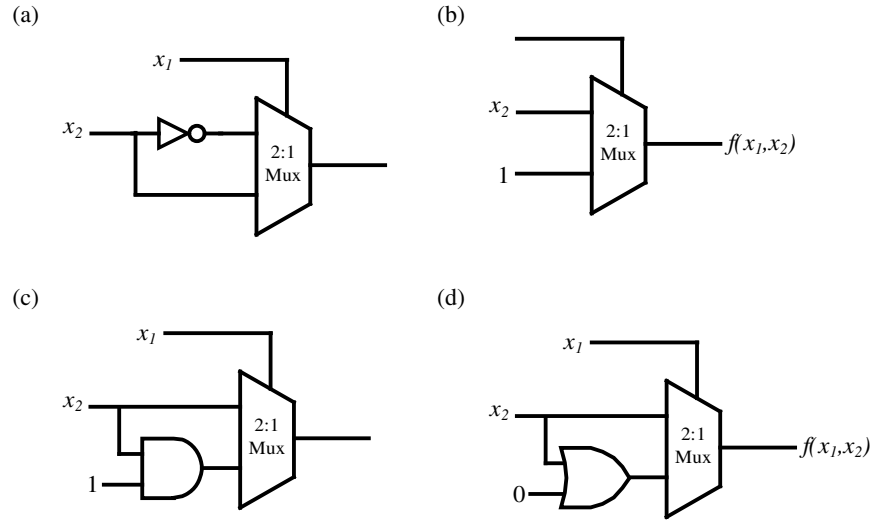


Figure P7.1

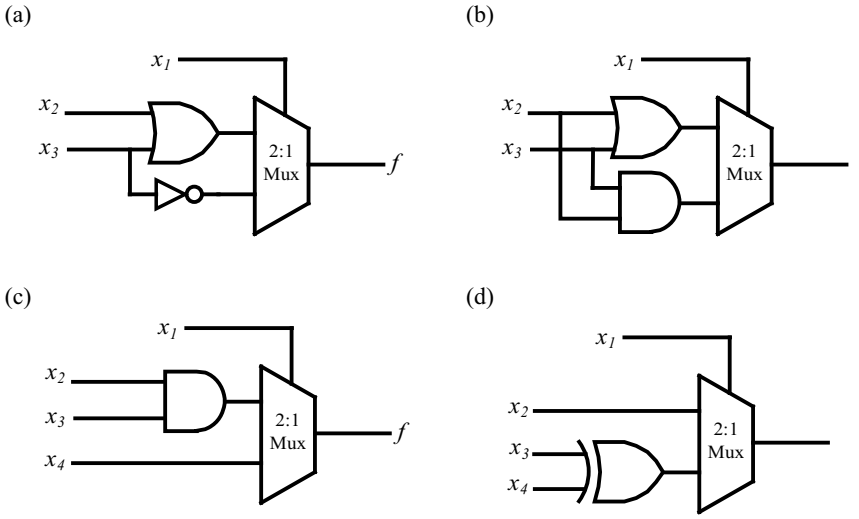


Figure P7.2

7.3 Implement the function of the logic circuit in Figure P7.3 using an 8 : 1 multiplexer.

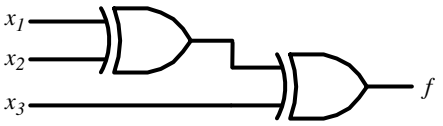


Figure P7.3

7.4 Implement the function of the logic circuit in Figure P7.3 using a 4 : 1 multiplexer and additional logic gates.

7.5 Implement the function of the logic circuit in Figure P7.3 using a 2 : 1 multiplexer and additional logic gates.

7.6 Implement the function of the logic circuit in Figure P7.6 using a 16 : 1 multiplexer.

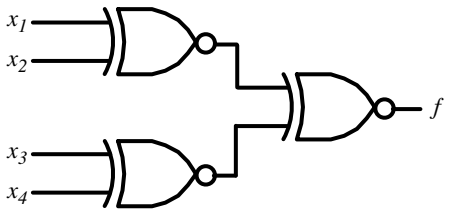


Figure P7.6

7.7 Implement the function of the logic circuit in Figure P7.6 using an 8 : 1 multiplexer and additional logic gates.

7.8 Implement the function of the logic circuit in Figure P7.6 using a 4 : 1 multiplexer and additional logic gates.

7.9 Implement the function of the logic circuit in Figure P7.6 using a 2 : 1 multiplexer and additional logic gates.

7.10 Implement the following logic functions using multiplexers.

(a) $f(x_1, x_2) = (x_1 + x_2)(\bar{x}_1 + x_2)(x_1 + \bar{x}_2)$

(b) $f(x_1, x_2) = \overline{x_1 x_2} + (x_1 + x_2)$

(c) $f(x_1, x_2) = (\bar{x}_1 + \bar{x}_2)\overline{x_1 + x_2}$

(d) $f(x_1, x_2) = \overline{x_1 \oplus x_2}$

(e) $f(x_1, x_2, x_3) = x_1\bar{x}_2 + x_1\bar{x}_2 x_3$

(f) $f(x_1, x_2, x_3) = x_1 + x_1\bar{x}_2 + x_1\bar{x}_2 x_3$

7.11 Implement the following logic functions using 2 : 1 multiplexers only.

(a) $f(x_1, x_2, x_3) = (x_1 + x_2)(x_1 + x_3)$

(b) $f(x_1, x_2, x_3) = x_3 + x_1 x_2$

(c) $f(x_1, x_2, x_3) = x_1 x_2 + (\bar{x}_1 + x_3)$

(d) $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2)(x_3 \oplus x_4)$

7.12 Write VHDL code to implement a 4 : 1 multiplexer.

7.13 Write VHDL code to implement a 4 : 1 multiplexer using a **when–else** statement.

7.14 Write VHDL code to implement a 4 : 1 multiplexer using an **if–then–else** statement.

7.15 Write VHDL code to implement a 4 : 1 multiplexer using a **case** statement.

7.16 Write VHDL code to implement a 16 : 1 multiplexer using 4 : 1 multiplexers.

7.17 Write VHDL code to implement an 8 : 3 priority encoder.

7.18 Write VHDL code to implement a 3 : 8 binary decoder.

7.19 Write VHDL code to implement a 3-bit priority encoder.

7.20 Write VHDL code to implement a BCD-to-seven-segment encoder using a select signal assignment.

7.21 Write VHDL code to implement a BCD-to-Gray code converter.

7.22 Write VHDL code to implement a BCD-to-excess-3 code converter.

7.23 Verify that $c_i = x_i \oplus y_i \oplus s_i$, where x_i and y_i are the inputs, s_i is the sum, and c_i is the carryout from $(i-1)$th bit position.

7.24 Verify that a 1-bit full-subtractor can be designed using two cascaded 1-bit half-subtractors.

7.25 Design a 4-bit ripple-carry adder.

7.26 Design a 4-bit carry look-ahead adder.

7.27 Compare the number of gates of an 8-bit ripple-carry adder to that of an 8-bit carry look-ahead adder.

7.28 Write VHDL code to implement a 32-bit hybrid adder by using four 8-bit carry look-ahead adders. Use a **component** statement to declare an 8-bit carry look-ahead adder as a subcircuit.

7.29 Design a logic circuit to compare two 4-bit numbers, A and B. The circuit should provide outputs for the following conditions: $A = B$, $A > B$, $A < B$, and $A \neq B$.

7.30 Write VHDL code to implement a 32-bit comparator.

7.31 Write VHDL code to implement an 8-bit comparator.

8 Sequential Logic

8.1 OBJECTIVES

The objectives of the chapter are to describe:

- Sequential logic circuits
- SR and DF latches
- SR, D, JK, and T flip-flops
- Shift and parallel registers
- Asynchronous and synchronous counters

8.2 SEQUENTIAL LOGIC CIRCUITS

In Chapter 7 we discussed combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the output depend not only on the present values of the inputs but also on the past behavior of the circuit. Circuits that have this behavior are referred to as *sequential circuits*. The output values of a sequential circuit depend on the temporal sequence of input values. A sequential circuit is described in terms of logic conditions referred to as *logic states*. A logic state is the logic value of a circuit, which is momentarily preserved. Therefore, sequential circuits include memory elements that store the values of the logic states. The general structure of a sequential circuit is illustrated in Figure 8.1.

A stable state of a sequential circuit is described as previous, present, or next. The *present state* is the present logic output of the circuit. The *previous state* is the logic output of the circuit before the present state. The *next state* is the logic output of the circuit after the present state. Notice that the previous state cannot jump to the next state without going through the next state. When the previous, present, and next states occur during consecutive equal time periods, the sequential circuit is referred to as *synchronous*.

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

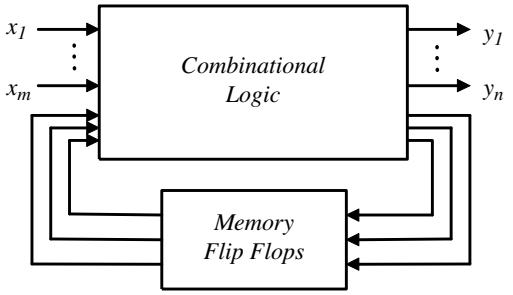


Figure 8.1 Sequential Circuit Block Diagram

8.3 LATCHES

Latches and flip-flops are the basic building blocks of most sequential circuits. A *latch* is a sequential circuit that watches all the inputs continuously and changes its outputs at any time independently of a clocking signal. On the other hand, a *flip-flop* changes its outputs only at times determined by a clocking signal. A latch is a bistable (two stable output states) device that can store one bit (a logic 0 or 1) of data. Because of their storing capacity, latches are sometimes referred to as *bistable memory devices*. Latches may be used in groups of 4, 8, 16, or 32 for temporary storage of a nibble, byte, or word of data. They are also used often in microprocessor-based design. The most commonly used types of latches are described in the following sections.

8.3.1 SR Latch

A *set–reset latch*, commonly referred to as an *SR latch*, has two inputs (S and R), one true output (Q), and one complemented output ($\bar{Q}$), as shown in Figure 8.2. The crossing of the outputs is known as *cross coupling*. This circuit is said to employ cross-coupled feedback. The feedback connects the output of a circuit to its input. When output Q is equal to 1, the latch is said to be in the *set state*; similarly, when Q is equal to 0, the latch is said to be in the *clear* (or *reset*) *state*. For the basic NOR latch circuit, both inputs normally are at the 0 logic level. If one input (S or R) changed to the logic 1 level, the corresponding output (Q or $\bar{Q}$) will be forced to logic 0. The same output logic level 0 will also be applied, through the feedback, to the second input of the other NOR gate, forcing its output to change to logic level 1. This output, in turn, feeds back to the second input of the original NOR gate, forcing its output to remain at logic level

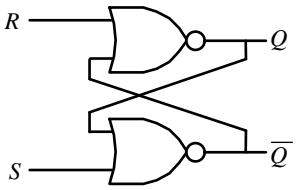


Figure 8.2 Logic Circuit Diagram of an SR Latch

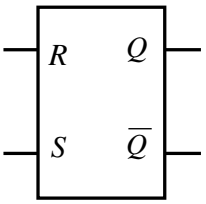
Truth Table					*Graphic Symbol*
S	***R***	***Q***	$\overline{Q}$	***Comments***	
0	0	1/0	0/1	*No change*	
0	1	0	1	*Reset*	
1	0	1	0	*Set*	
1	1	0	0	*Not used*	

Figure 8.3 Truth Table and Logic Symbol of an SR Latch

0 even after the external input is removed. The logic circuit diagram of the SR latch is illustrated in Figure 8.2.

The truth table and graphic symbol of an SR latch are shown in Figure 8.3. For clarity and simplicity, the graphic symbol will be used to represent a complex logic circuit. Notice that when both inputs are at logic level 1, the SR latch does not maintain a stable output. These inputs force both outputs to logic level 0, overriding the feedback latching action. This state is sometimes referred to as the *forbidden state*. To understand why this state is fatal to the functioning of an SR latch, consider the following cases. From the truth table of Figure 8.3, one could notice that whichever input goes to logic level 0 first will lose control, while the other input (still at logic level 1) controls the resulting state of the latch. If both inputs go to logic level 0 simultaneously, the result is a race condition, and the final state of the latch cannot be determined ahead of time. The timing diagram shown in Figure 8.4 illustrates the behavior of an SR latch.

The truth table in Figure 8.3 can be written to identify the present state ($Q(t)$) and the next state ($Q(t+1)$) of an SR latch as a function of time. The present input variable combinations and the latch state at time t ($Q(t)$) will determine the next state of the latch at time $(t+1)$ ($Q(t+1)$). The resulting truth table, referred to as the *characteristic table* of the SR latch, is illustrated in Figure 8.5. An $\overline{S}\overline{R}$ latch with active-low set and reset inputs is designed using NAND gates as shown in Figure 8.6. The operation of the $\overline{S}\overline{R}$ latch is similar to that of the SR latch, with two major differences. First the latch holds its previous state when $\overline{S} = \overline{R} = 1$. Second, when S and $\overline{R}$ are

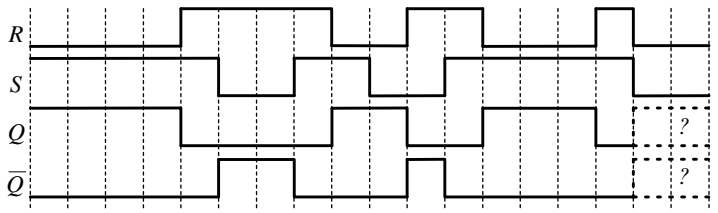


Figure 8.4 Timing Diagram of an SR Latch

S	*R*	*Present State Q(t)*	*Next State Q(t+1)*	*Comments*
0	0	0	0	*No change*
0	0	1	1	*No change*
0	1	0	0	*Reset*
0	1	1	0	*Reset*
1	0	0	1	*Set*
1	0	1	1	*Set*
1	1	0	Oscillating	*Invalid*
1	1	1	Oscillating	*Invalid*

Figure 8.5 Characteristic Table of an SR Latch

Truth Table

$\overline{S}$	$\overline{R}$	*Q*	$\overline{Q}$	*Comments*
1	1	1/0	0/1	*No change*
0	1	1	0	*Set*
1	0	0	1	*Reset*
0	0	1	1	*Not used*

Logic Diagram

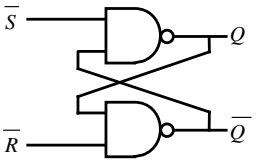


Figure 8.6 Truth Table and Logic Diagram of an $\overline{S}\overline{R}$ Latch

asserted simultaneously, both latch outputs go to 1 and not to 0 as in an SR latch. Under these input conditions, the $\overline{S}\overline{R}$ latch is unstable. Figure 8.6 illustrates the truth table and logic diagram of an $\overline{S}\overline{R}$ latch.

Again notice that when both inputs are at logic level 0, the SR latch does not maintain a stable output. These inputs force both outputs to a logic level 1, overriding the feedback latching action, a state again sometimes referred to as the forbidden state. Whichever input goes to logic level 1 first will lose control, while the other input (still at logic level 0) controls the resulting state of the latch. If the two inputs go to logic level 1 simultaneously, the result is a race condition, and the final state of the latch cannot be determined ahead of time.

8.3.2 Gated SR Latch

The SR latches described in Section 8.3.1 are sensitive to *S* and *R* inputs at all times. These SR latches are called *asynchronous* because the outputs respond immediately to any changes in the inputs. However, the SR latch may be modified to create an SR latch, which is sensitive to its inputs only when an enabling input signal is asserted. Such an SR latch with an enable signal (*Clk*), referred to as a *gated SR latch*, is shown in Figure 8.7. When *Clk* is equal to 1, the circuit behaves like an SR latch, and when *Clk* is equal to 0, the circuit holds its preceding state. The truth table and graphic symbol are illustrated in Figure 8.8.

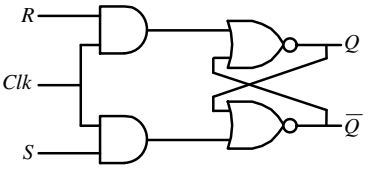


Figure 8.7 Logic Circuit Diagram of a Gated SR Latch

Truth Table

Clk	*S*	*R*	*Q*	$\overline{Q}$	*Comments*
1	0	0	1/0	0/1	*No change*
1	0	1	1	0	*Set*
1	1	0	0	1	*Reset*
1	1	1	1	1	*Not used*
0	x	x	1/0	0/1	*Disabled*

Graphic Symbol

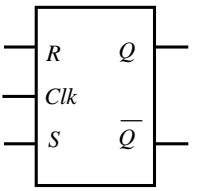


Figure 8.8 Truth Table and Logic Symbol of a Gated SR Latch

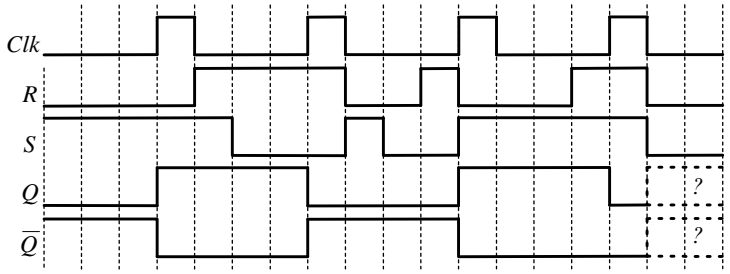


Figure 8.9 Timing Diagram of a Gated SR Latch

The enabling signal *Clk* is generally a periodic signal referred to as the *clock*. Thus, a gated SR latch is a synchronous latch. When the clock signal is HIGH (equal to logic level 1), the SR latch is operational, whereas when the clock signal is LOW (equal to logic level 0), the SR latch is disabled. The timing diagram shown in Figure 8.9 illustrates the function of the gated SR latch (*Q* is reset initially). Notice that the forbidden input conditions are still possible with a gated SR latch.

8.3.3 D Latch

One very useful variation of a gated SR latch circuit is the *data latch*, or *D latch* as it is generally called. As shown in Figure 8.10, a D latch is constructed by connecting *S* and *R* inputs through an inverter. The single combined input is designated *D* to distinguish its operation from that of other types of latches. The truth table and graphic symbol of a gated D latch are shown in Figure 8.11. In a gated D latch, when the *Clk* input is equal to

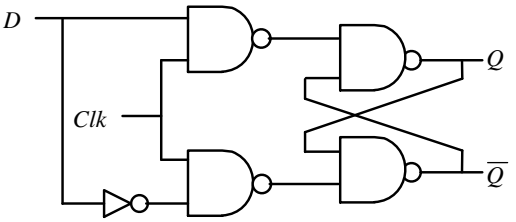


Figure 8.10 Logic Circuit Diagram of a Gated D Latch

Truth Table

Clk	*D*	*Q*	$\overline{Q}$	*Comments*
1	0	0	1	Reset
1	1	1	0	Set
0	x	1/0	0/1	Hold

Graphic Symbol

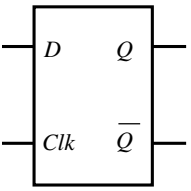


Figure 8.11 Truth Table and Logic Symbol of a Gated D Latch

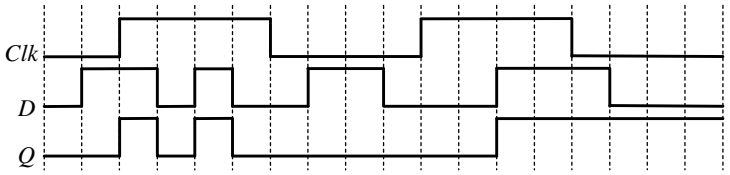


Figure 8.12 Timing Diagram of a Gated D Latch

logic level 1, the *Q* output will always reflect the logic level present at the *D* input. When the *Clk* input is equal to logic level 0, the last state of the *D* input is held at the output of the latch. Because the single *D* input is also inverted to provide the signal to reset the latch, the D-latch circuit cannot experience an unstable state. The timing diagram shown in Figure 8.12 illustrates the function of the gated D latch (*Q* is reset initially).

The VHDL code implementation of a gated D latch is illustrated in Figure 8.13 using an **if–then–else** statement. The entity description of the D latch includes two input signals, *D* and *Clk*. Since both affect the output of the circuit, they are included in the sensitivity list of the **process** statement.

8.4 FLIP-FLOPS

Flip-flops are synchronous bistable storage devices capable of storing one bit. The term *synchronous* means that the output state changes only at a specified point on a

```
library ieee ;
use ieee.std_logic_1164.all;

entity D_latch is
      port(
            D,clk          :     in      std_logic;
            Q              :     out     std_logic);
end D_latch;
architecture circuit_behavior of D_latch is
      begin
         process ( D, clk )
              begin
                  if clk = '1' then
                      Q <= D;
                  end if;
         end process
end circuit_behavior;
```

Figure 8.13 VHDL Code Implementation of a Gated D Latch

triggering input called the *clock signal*. That is, the output changes are synchronized with the clock signal. A basic flip-flop consists of two parts: a primary latch (master) and a secondary latch (slave). A flip-flop is also known as a *two-section flip-flop*. The input latch is operating as the master section and the output latch is slaved to the master during half of each clock cycle. We use the terms *primary* and *secondary* instead of *master* and *slave*.

8.4.1 SR Flip-Flop

An *SR flip-flop* consists of two identical gated SR latches. However, the inverter connected between the two *Clk* inputs ensures that the two sections will be enabled during opposite half-cycles of the clock signal. This is the key to the operation of this flip-flop circuit. Outputs Q and $\bar{Q}$ of the primary SR latch are connected to the inputs of the secondary SR latch. The *primary* SR latch is also called *master*, and the *secondary* is called *slave*. The logic circuit of an SR flip-flop is illustrated in Figure 8.14.

The two latches use a common clock signal, but an inverter gate is connected between them. When the clock signal goes HIGH, the inputs of the primary latch are able to change the outputs of the primary latch or the inputs of the secondary latch. However, at the same time the secondary latch cannot change the state of its outputs, which are the actual outputs of the flip-flop, because its clock signal is LOW. When the clock signal changes to LOW, the inputs of the primary are isolated from the secondary. At the same time, the inverted clock signal allows the secondary to change the output signals of the flip-flop. Because an SR flip-flop consists of two latches clocked at different phases of the clock signal, its operation is considerably different from that of a gated SR latch.

Starting with the *Clk* input at the logic level 0, the S and R inputs are disconnected from the inputs of the primary latch. Therefore, any changes in the input signals

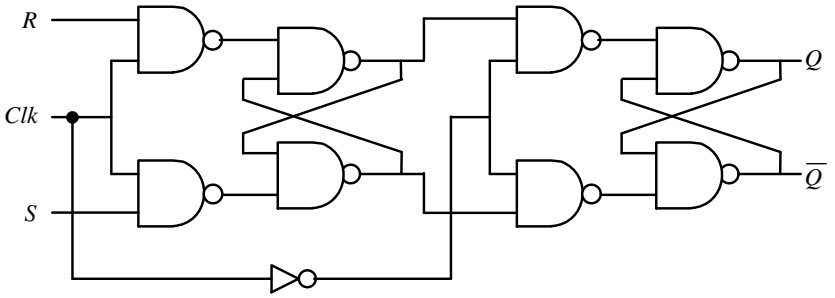


Figure 8.14 Logic Circuit Diagram of an SR Flip-Flop

Truth Table

Clk	*S*	*R*	*Q(t+1)*
↓	0	0	*Q(t)*
↓	0	1	0 *(Reset)*
↓	1	0	1 *(Set)*
↓	1	1	*Undefined*

Graphic Symbol

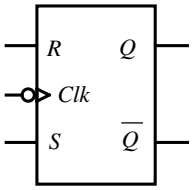


Figure 8.15 Truth Table and Logic Symbol of an SR Flip-Flop

cannot affect the state of the final outputs. When the *Clk* signal changes to logic level 1, the *S* and *R* inputs are able to control the state of the primary latch, just as with the single RS latch. However, at the same time the inverted *Clk* signal applied to the secondary latch prevents the state of the primary latch from having any effect on the outputs of the secondary. Therefore, any changes in the *S* and *R* input signals are tracked by the primary latch while *Clk* is at logic level 1, but the changes are not reflected at the *Q* and $\bar{Q}$ outputs, because the secondary latch is disabled. When the *Clk* signal changes to logic level 0, the *S* and *R* inputs are isolated from the primary latch. At the same time, the inverted *Clk* signal allows the current state of the primary latch to reach the output of the secondary latch. Therefore, the *Q* and $\bar{Q}$ outputs can change state only when the *Clk* signal falls from logic level 1 to logic level 0. This is known as the *falling* (*negative*) *edge* of the *Clk* signal: hence the designation *edge-triggered flip-flop*. The truth table and graphic symbol of an SR flip-flop are shown in Figure 8.15. The timing diagram shown in Figure 8.16 illustrates the function of a negative (falling) edge-triggered SR flip-flop (*Q* is reset initially).

8.4.2 D Flip-Flop

A *D flip-flop* performs a function very similar to that of a D latch. In a D flip-flop, however, the rising or falling edge of the clock signal is used to capture the input of the

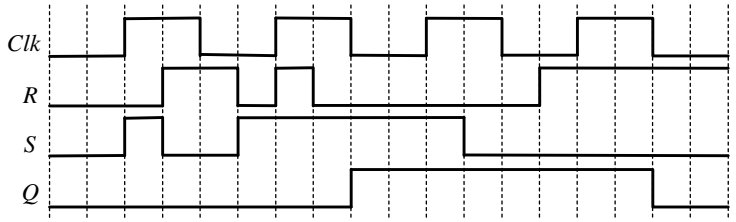


Figure 8.16 Timing Diagram of an SR Flip-Flop

flip-flop. This device is very useful when it is necessary to track a logic signal, which is very rapidly varying. This type of device is said to be *edge-triggered*. Rising edge-triggered (0-to-1 transition) or falling edge-triggered (1-to-0 transition) devices are available.

The D flip-flop shown in Figure 8.17 is a modification of a clocked SR flip-flop. The *D* input connects directly to the *S* input, and the complement of the *D* input connects to the *R* input. The *D* input is sampled during the occurrence of a clock signal transition. If *D* is equal to 1, the flip-flop is switched to the set state (unless it was already set). If *D* is equal to 0, the flip-flop switches to the clear state. Figure 8.17 shows the logic circuit of a D flip-flop. The truth table and graphic symbol of a D flip-flop are illustrated in Figure 8.18.

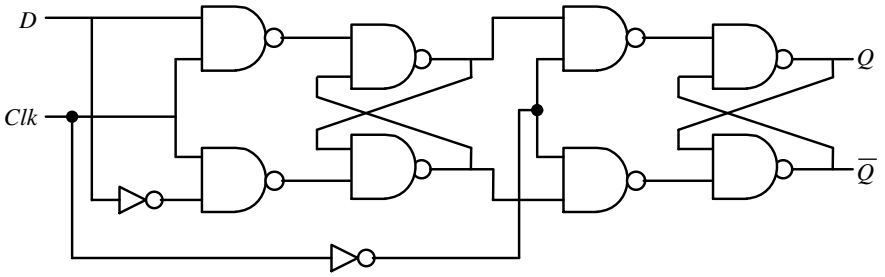


Figure 8.17 Logic Circuit Diagram of a D Flip-Flop

Truth Table

Clk	*D*	*Q(t)*	*Q(t+1)*
↓	0	0	0
↓	0	1	0
↓	1	0	1
↓	1	1	1

Graphic Symbol

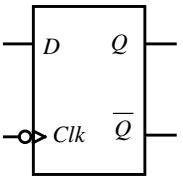


Figure 8.18 Truth Table and Logic Symbol of a D Flip-Flop

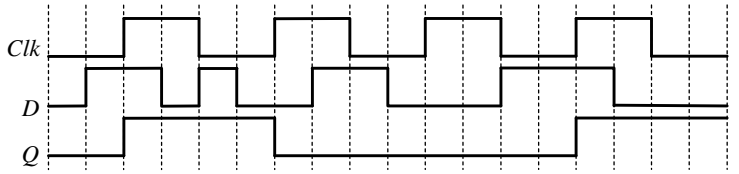


Figure 8.19 Timing Diagram of a D Flip-Flop

```
library ieee ;
use ieee.std_logic_1164.all;

entity D_flipflop is
      port (
            D,clk          :     in      std_logic;
            Q              :     out     std_logic);
end D_flipflop;
architecture circuit_behavior of D_flipflop is
      begin
          process (clk)
              begin
                  if clk' event and clk = '1' then
                      Q <= D;
                  end if;
          end process
end circuit_behavior;
```

Figure 8.20 VHDL Code Implementation of a D Flip-Flop Using **if–else–then**

The timing diagram shown in Figure 8.19 illustrates the function of a positive (rising) edge-triggered D flip-flop (*Q* is initially reset). The VHDL code shown in Figure 8.20 implements a D flip-flop. The code uses the **if–else–then** statement. Notice the clause "clk'**event and** clk = '1'", which is used to represent the positive edge of the clock signal. The clause '**event** is an attribute that defines any change in the clock signal. Thus, the "clk'**event and** clk = '1'" clause means that when the clock signal changes, it is now equal to logic level 1 (which refers to the positive edge of the clock signal). The sequential statements inside the **if–else–then** statement are then executed.

The VHDL code in Figure 8.21 is another implementation of the D flip-flop, but the code uses the **wait until** clause instead of the **if–else–then** statement. Notice that there is a sensitivity list for the **process** statement. That is because the **wait until** clause uses the clock signal as an implicit sensitivity signal.

8.4.3 JK Flip-Flop

The *JK flip-flop* is the most versatile and most commonly used flip-flop when discrete devices are used to implement arbitrary finite-state machines, which are described in Chapter 9. A JK flip-flop can be configured to work as a D, RS, or T flip-flop. The

```
library ieee ;
use ieee.std_logic_1164.all;

entity D_flipflop is
      port (
            D,clk          :     in      std_logic;
            Q              :     out     std_logic);
end D_flipflop;
architecture circuit_behavior of D_flipflop is
      begin
         process
            begin
               wait until clk' event and clk = '1';
                  Q <= D;
         end process
end circuit_behavior;
```

Figure 8.21 VHDL Code Implementation of a D Flip-Flop Using **wait–until**

T flip-flop is described in the next section. Similar to a RS flip-flop, a JK flip-flop has two data inputs, *J* and *K*, and a clock input. However, it does not have undefined states or a race condition. As a flip-flop the JK flip-flop is edge triggered. The JK flip-flop has the following edge-triggered operating characteristics:

- If one input (*J* or *K*) is at logic level 0 and the other input is at logic level 1, the output is set or reset (by *J* or *K*, respectively).
- If both inputs are at logic level 0, the output remains unchanged.
- If both inputs are at logic level 1, the flip-flop changes its state (toggles) at the edge of a clock signal.

Figure 8.22 shows the truth table and graphic symbol of a positive edge-triggered JK flip-flop. JK flip-flops can be designed using latches or flip-flops. Figure 8.23 shows a JK flip-flop constructed using a D flip-flop and additional logic gates. The outputs of the D flip-flop are ANDed together with *J* and *K*, respectively. Notice that

Truth Table

Clk	*J*	*K*	*Q(t+1)*
↑	0	0	*Q(t) (Hold)*
↑	0	1	0 *(Reset)*
↑	1	0	1 *(Set)*
↑	1	1	$\overline{Q}_{(t)}$ *(Toggle)*

Graphic Symbol

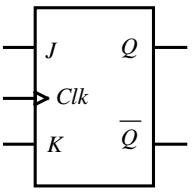


Figure 8.22 Truth Table and Logic Symbol of a JK Flip-Flop

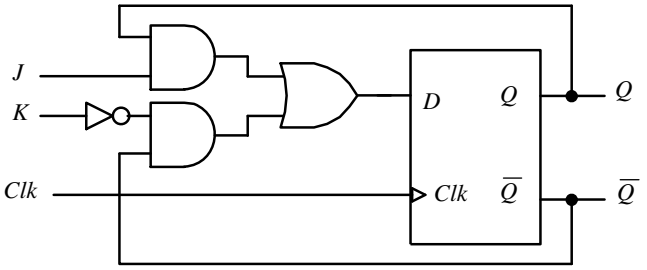


Figure 8.23 Logic Circuit Diagram of a JK Flip-Flop

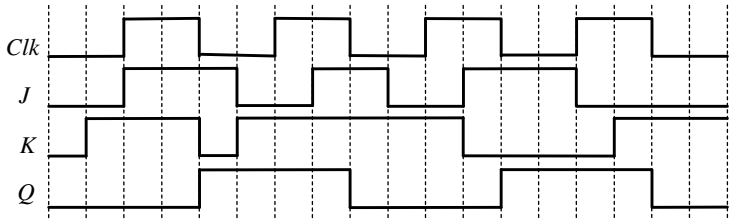


Figure 8.24 Timing Diagram of a JK Flip-Flop

the connection between outputs and inputs to AND gates determines the input conditions to D when $J = K = 1$. This connection is what causes the flip-flop to toggle. Figure 8.24 illustrates a timing diagram of a JK flip-flop with a falling-edge trigger. The output Q of the flip-flop is reset initially.

8.4.4 T Flip-Flop

The output of a *T flip-flop* ("toggle flip-flop") holds its current state when the input, T, is LOW (equal to logic level 0). It reverses its current state when the input is HIGH (equal to logic level 1). Although a T flip-flop is a useful element for building counter circuits, it is not generally available in a commercial product.

The T flip-flop is a simplified version of a JK flip-flop that is used in many types of circuits, especially counters and dividers. Its only function is that it toggles itself with every clock pulse (on either the rising or falling edge). It can be constructed from a D flip-flop as illustrated in Figure 8.25. The T flip-flop is normally set, or "loaded," with preset and clear inputs. It can be used to generate a periodic signal with a frequency half that of a clock signal.

Figure 8.26 shows the truth table and graphic symbol of a positive edge-triggered T flip-flop. Figure 8.27 illustrates a timing diagram of a T flip-flop with a falling-edge trigger. The output Q of the flip-flop is reset initially. The VHDL code shown in Figure 8.28 implements a T flip-flop with a positive edge-triggered clock signal. The code uses nested **if–else–then** statements. Notice the clause "clk'**event and** clk = '1'", which is used to represent the positive edge of the clock signal.

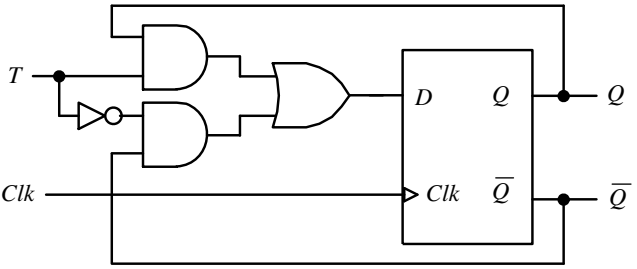


Figure 8.25 Logic Circuit Diagram of a T Flip-Flop

Truth Table			*Graphic Symbol*
Clk	***T***	***Q(t+1)***	
↑	0	*Q(t) (Hold)*	
↑	1	$\overline{Q}_{(t)}$ *(Toggle)*	

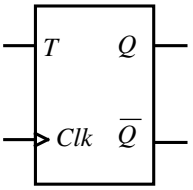


Figure 8.26 Truth Table and Logic Symbol of a T Flip-Flop

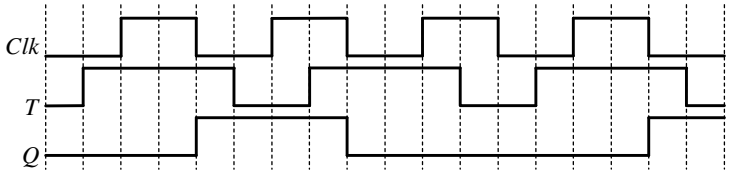


Figure 8.27 Timing Diagram of a T Flip-Flop

8.5 REGISTERS

The term *register* refers to a group of *N* flip-flops operating as a single unit to store or shift data. A register is a set of flip-flops with a common clock to all the flip-flops. For example, a shift register is an *N*-bit register which shifts its stored data by one bit position (left or right) at each (rising or falling) edge of the clock. The flip-flops are connected in a chain so that the output of one flip-flop is the input of the next flip-flop. A common clock drives all the flip-flops, and all flip-flops are set or reset simultaneously. In the following sections, the basic types of shift registers are described.

```
library ieee ;
use ieee.std_logic_1164.all;

entity T_flipflop is
      port (
            T,clk          :     in      std_logic;
            Q              :     out     std_logic);
end T_flipflop;
architecture circuit_behavior of T_flipflop is
      begin
         process (T,clk)
            begin
               if clk' event and clk = '1' then
                  if T = '1' then
                     Q <= not Q;
                  end if;
               end if;
         end process
end circuit_behavior;
```

Figure 8.28 VHDL Code Implementation of a T Flip-Flop Using Nested **if–else–then** Statements

8.5.1 Serial-In, Serial-Out Shift Registers

An *N*-bit shift register can be constructed using *N* D flip-flops connected in series. Each D flip-flop stores 1 bit of information. The serial input, D_{in}, specifies a new bit to be shifted into one end at each clock cycle. This bit appears at the serial output, D_{out}, after *N* clock cycles and is lost one clock cycle later. Figure 8.29 shows the structure of a 4-bit serial-in, serial-out right-shift register.

The operation of the 4-bit right-shift register is illustrated in Figure 8.30 with a random input sequence. The register is first cleared. Each D flip-flop is equipped with a clear input, which forces the output of the flip-flop to logic level 0. The clear input is an asynchronous input independent of the clock signal. By ANDing the clear input with the clock signal, the clear input becomes synchronized to the clock signal. The clear input is generally active LOW, which means that a logic level 0 is required to clear the flip-flop.

The input data are then applied sequentially to the D_{in} of the first flip-flop on the left. During each clock cycle, one bit is transmitted from left to right at each positive

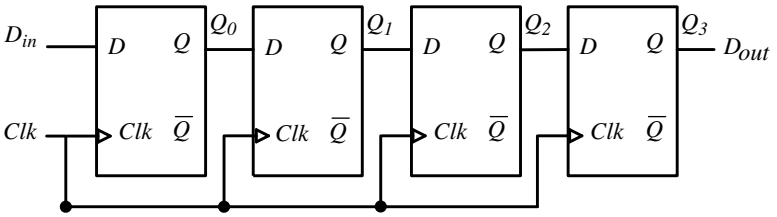


Figure 8.29 Logic Circuit Diagram of a 4-Bit Serial-In, Serial-Out Register

Time	D_{in}	Q_0	Q_1	Q_2	Q_3
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	0	0	1	0	0
t_3	1	0	0	1	0
t_4	0	1	0	0	1
t_5	1	0	1	0	0
t_6	1	1	0	1	0
t_7	0	1	1	0	1
t_8	1	0	1	1	0

Figure 8.30 Data Shifting by a 4-Bit Serial-In, Serial-Out Register

```
library ieee ;
use ieee.std_logic_1164.all;

entity register_shift4 is
      port (
            clk, Din      :    in          std_logic;
            Q             :    buffer      std_logic_vector(3 downto 0));
end register_shift4;
architecture circuit_behavior of register_shift4 is
      begin
         process
         begin
            wait until clk'event and clk = '1';
               Q(0)     <=    Q(1);
               Q(1)     <=    Q(2);
               Q(2)     <=    Q(3);
               Q(3)     <=    Din;
         end process;
end circuit_behavior;
```

Figure 8.31 VHDL Code Implementation of a Serial-In, Serial-Out Register

edge of the clock. Notice that the input data are not shifted in at the current rising edge of the clock signal. We assumed that the input data came after the rising edge of the clock, and it is not seen by the flip-flop until the next rising edge of the clock signal.

The VHDL code shown in Figure 8.31 implements a 4-bit serial-in, serial-out right-shift register. The clause **buffer** is used to designate a special port of the **entity**. Signals can flow in either direction through a **buffer** port. The signal flowing into the **buffer** can be used inside the **entity**. Notice that the signal *Q* can appear on both sides of the assignment operator.

8.5.2 Serial-In, Parallel-Out Shift Registers

For a serial-in, parallel-out shift register, data bits are shifted in serially in a manner similar to that of a serial-in, serial-out shift register. The only difference between the

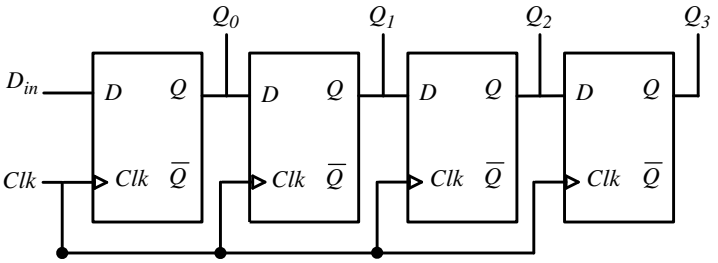


Figure 8.32 Logic Circuit Diagram of a Serial-In, Parallel-Out Register

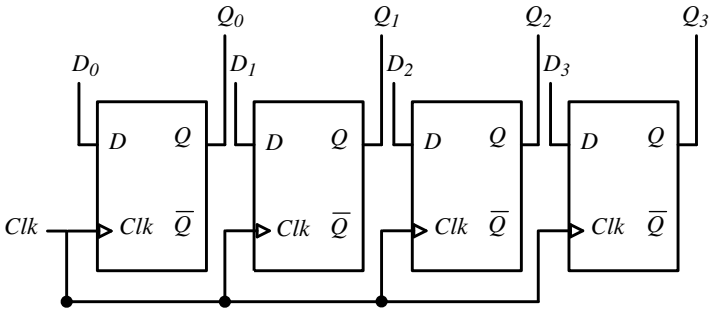


Figure 8.33 Logic Circuit Diagram of a Parallel-In, Parallel-Out Register

two registers is the method in which the data bits are read out from the latter register. Once the data are shifted in and stored in their respective bit positions, all bits appear on their respective outputs and are available simultaneously. A typical application of serial-in, parallel-out shift registers is serial-to-parallel conversion of the data format between input–output devices and processors. Figure 8.32 illustrates a 4-bit serial-in, parallel-out right-shift register. Parallel-in, serial-out shift registers perform a function opposite to that of serial-in, parallel-out shift registers. The data bits are stored simultaneously and shifted out serially.

For parallel-in parallel-out shift registers, the data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. The logic circuit shown in Figure 8.33 is a 4-bit parallel-in, parallel-out shift register constructed using D flip-flops. The D's are the parallel inputs and the Q's are the parallel outputs. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously. The VHDL code shown in Figure 8.34 implements a 4-bit parallel-in, parallel-out right-shift register.

8.5.3 Bidirectional Shift Registers

The serial shift registers discussed in previous sections involved only right-shift operations, which means that the data bits were shifted from left to right. Notice that a right-shift operation has the effect of dividing the binary number by 2. A left-shift operation will have the effect of multiplying the number by 2. Reversing the order and

```
library ieee ;
use ieee.std_logic_1164.all;

entity register4 is
      port (
            D              :    in        std_logic_vector( 3 downto 0);
            clk, Din       :    in        std_logic;
            Q              :    buffer    std_logic_vector(3 downto 0));
end register4;
architecture circuit_behavior of register4 is
      begin
         process
         begin
            wait until clk' event and clk = '1';
                Q  <=   D;
          end process;
end circuit_behavior;
```

Figure 8.34 VHDL Code Implementation of a Parallel-In, Parallel-Out Register

direction of the flip-flops in the right-shift register, the register can perform left-shift operation. A *bidirectional* or *reversible, shift register* can be designed with proper gating arrangement. A bidirectional shift register shifts data either to the left or right. An input control line (R/L) is provided to switch between the two modes.

8.6 COUNTERS

A *counter* is simply a device that counts up or down. Counters have various important applications. Counters may be used to count numbers, operations, quantities, or periods of time. They may also be used for dividing frequencies, for addressing information in storage, or for temporary storage. Counters are a series of flip-flops wired together to perform the type of counting desired. They will count up or down by ones, twos, or more. The total number of counts or stable states that a counter can indicate is called the *modulus*. For example, the modulus of a four-stage counter would be 16_{10}, since it is capable of indicating $(0000)_2$ to $(1111)_2$. The term *modulo* is used to describe the count capability of counters. Modulo-16 represents a four-stage binary counter, modulo-11 represents a decade counter, modulo-8 represents a 3-bit binary counter, and so on.

The number of flip-flops used and how they are connected determine the number of states and the sequence of the states that the counter goes through in each complete cycle. Counters can be classified into two broad categories according to the way they are clocked:

1. *Asynchronous* (also referred to as *ripple*) counters. In this type of counter the first flip-flop is clocked by the external clock signal. Each successive flip-flop is clocked by either the Q or $\bar{Q}$ output of the preceding flip-flop.

2. Synchronous counters. In this type of counter all memory elements (flip-flops) are triggered simultaneously by the same clock signal.

In the following sections, various asynchronous and synchronous up–down counters are described.

8.6.1 Asynchronous Up–Down Counters

In asynchronous counters, only the first flip-flop is clocked by an external clock signal, and each successive flip-flop is clocked by either the Q or $\bar{Q}$ output of the preceding flip-flop. Thus, the remaining flip-flops do not have a common clock control. Asynchronous counters are also referred to as *ripple counters* because the information ripples from the less significant bit to the more significant bit, one bit at a time. T flip-flops are the building blocks of most asynchronous counters. A 4-bit asynchronous counter is constructed using four T flip-flops, as illustrated in Figure 8.35.

As illustrated in Figure 8.35, the external clock is connected to the clock input of the first flip-flop only (left). Each subsequent flip-flop is connected to the inverted output $\bar{Q}$ of the preceding flip-flop. The input to the first flip-flop is always HIGH (equal to logic level 1); therefore, it changes state at the rising edge of the clock signal. The next flip-flop changes only when triggered by the rising edge of the Q output of the first flip-flop. Because of the inherent propagation delays through the flip-flops, the transition of the input clock signal and a transition of the Q output of the first flip-flop can never occur at exactly the same time. Because of the cumulative propagation delays resulting from the ripple action of the clock signal, the flip-flops cannot be triggered simultaneously. This ripple effect produces asynchronous operation of the counter. The timing diagram of a 3-bit asynchronous up counter shown in Figure 8.36 illustrates the effects of propagation delays.

A variation of the counter shown in Figure 8.35 is illustrated in Figure 8.37. Notice that only the first flip-flop is triggered by the clock signal. Each subsequent flip-flop is connected to the output Q of the preceding flip-flop instead. The resulting counter will count down from $(1111)_2$ to $(0000)_2$. Therefore, the counter is a 4-bit asynchronous down counter.

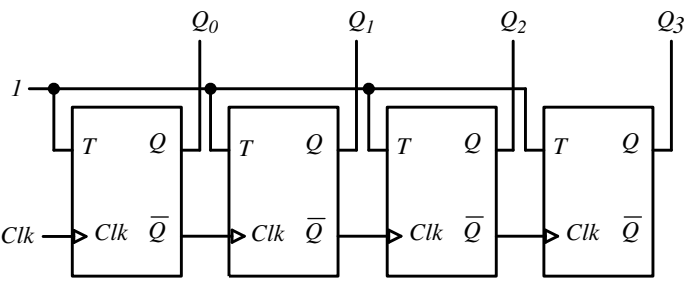


Figure 8.35 Logic Circuit Diagram of a 4-Bit Asynchronous Up Counter

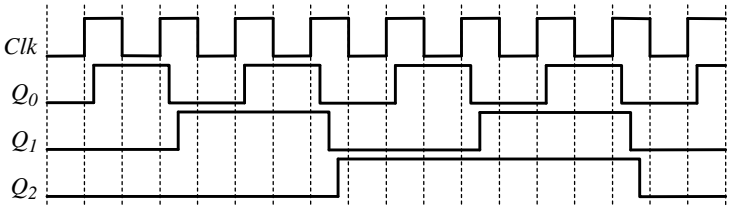


Figure 8.36 Timing Diagram of a 3-Bit Asynchronous Up Counter

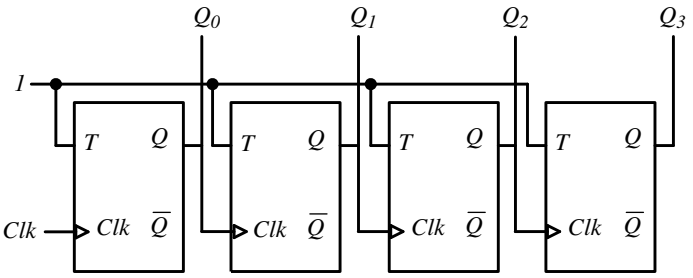


Figure 8.37 Logic Circuit Diagram of a 4-Bit Asynchronous Down Counter

8.6.2 Asynchronous Decade Counter

The general binary counters count in serial count sequence up or down. An *N*-bit counter has 2^N states and counts from 0 to $(2^N - 1)$ or $(2^N - 1)$ to 0. However, counters can have a truncated count sequence with fewer states. The truncated sequence is achieved by forcing the counter to reinitialize before going through all of its normal states. This type of counter is referred to as a *modulus-M counter*, where M represents the states beyond which the remaining states are truncated. A common truncated sequence counter is the modulus-10 counter, also referred to as a *decade counter*. The 10 states of a decade counter represent the BCD numbers from 0 to 9. The logic circuit shown in Figure 8.38 is an implementation of an asynchronous decade counter.

Once the counter counts to 10 (1010), all the flip-flops are reset by the NAND output. Notice that when the counter reaches 10, Q_1 and Q_3 are at logic level 1, whereas Q_0 and Q_2 are at logic level 0. Thus, only Q_1 and Q_3 are used to decode the state, which corresponds to a count of 10. This is called *partial decoding*. The timing diagram shown in Figure 8.39 illustrates an asynchronous modulus-6 counter. To illustrate, Figure 8.39 includes propagation delays. Despite the fact that the asynchronous modulus-6 counter counts up to 5, the last state is valid only for a shorter period of time. Thus, the last state may not be seen by devices that use this counter. Therefore, the counter acts as a modulus-5 rather than a modulus-6 counter.

8.6.3 Synchronous Counters

As shown earlier, asynchronous counters are simple to design, but their cumulative propagation delays severely degrade its counting function. For larger asynchronous

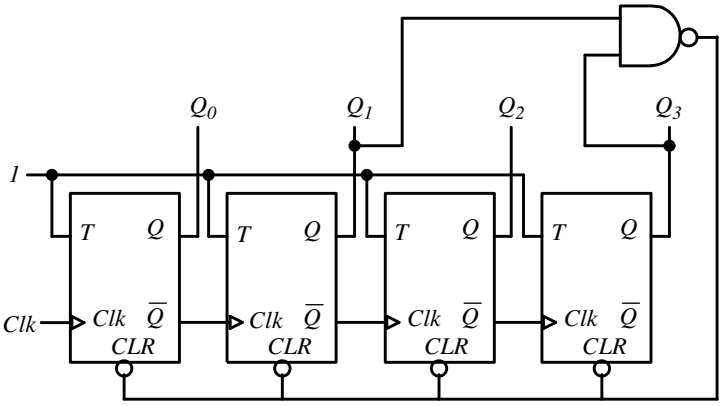


Figure 8.38 Logic Circuit Diagram of an Asynchronous Decade Counter

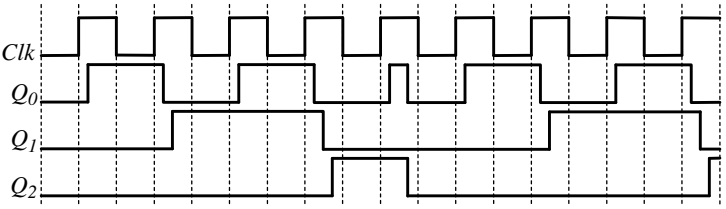


Figure 8.39 Timing Diagram of an Asynchronous Modulus-6 Counter

counters, the cumulative propagation delay may be as large as the period of the clock signal. Of course, one would increase the period of the clock signal to compensate for the propagation delays, but longer clock signal periods mean slower counting. To overcome the propagation delays of the "ripple" effects of asynchronous counters, synchronous counters are generally used instead. The difference between a synchronous counter and an asynchronous counter is that all flip-flops in the synchronous counter receive a common clock signal and change their states at the same time (in parallel).

Thus, the ripple effects are eliminated and the propagation delays are minimized. Therefore, synchronous counters are faster than asynchronous counters. Synchronous counters can be designed to count up or down. They can also be designed to generate special count sequences of nonconsecutive numbers. Because all flip-flops are synchronized to the same clock signal, the count sequence will depend on the logic functions which drive the individual flip-flops. For example, to design a 3-bit synchronous up counter using T flip-flops, one needs to list the count sequence (Figure 8.40) of the counter and infer design criteria.

For the count sequence listed in Figure 8.40, the following observations can be made:

- The output Q_0 toggles on each clock cycle, thus, $T_0 = 1$.
- The output Q_1 toggles on the next clock cycle each time that $Q_0 = 1$; thus, $T_1 = Q_0$.

Clock Cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

Figure 8.40 Count Sequence of a 3-Bit Synchronous Up Counter

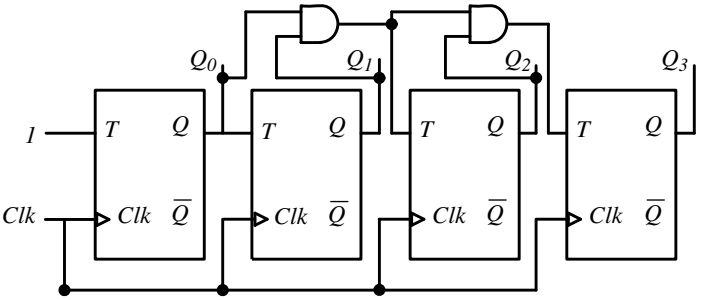


Figure 8.41 Four-Bit Synchronous Up Counter Designed with T Flip-Flops

- The output Q_2 toggles on the next clock cycle each time that $Q_0 = 1$ and $Q_1 = 1$; thus, $T_2 = Q_1Q_0$.
- For an N-bit counter, one would infer that $T_{N\text{-}1} = Q_{N\text{-}1} \cdots Q_1Q_0$.

Figure 8.41 is an example of a 4-bit synchronous up counter designed with T flip-flops. Notice that the additional AND gates are connected so as to satisfy the relations derived from the observations made above.

The VHDL code in Figure 8.42 shows an implementation of an 8-Bit synchronous up counter. Similar to asynchronous counters, synchronous modulus-N counters can also be designed. The VHDL code shown in Figure 8.43 illustrates a synchronous modulus-6 counter. The synchronous modulus-6 counter counts up from 000 to 101 and then recycles to 000 again.

8.6.4 BCD Counters

Digital counters are very useful in many applications. They can be found in digital clocks and parallel-to-serial data conversion. Parallel-to-serial conversion is normally accomplished by the use of a counter to provide a binary sequence for the data-select inputs of a multiplexer. Although counters can be designed to perform dedicated counting tasks, in general, a typical counter may include means to enable, clear, and load the counter. The enable input is designed to disable the counter when it is not

```
library ieee ;
use ieee.std_logic_1164.all;

entity syn_upcounter is
      port(
            clk, clear        :     in        std_logic;
            Q                 :     buffer    std_logic_vector(7 downto 0));
end syn_upcounter;
architecture circuit_behavior of syn_upcounter is
      begin
         process (clk, clear)
         begin
           if clear = '1' then
               Q <= 0;
           elsif (clk' event and clk = '1') then
               Q  <=    Q+1;
           end if;
         end process;
end circuit_behavior;
```

Figure 8.42 VHDL Code Implementation of an 8-Bit Synchronous Up Counter

in-use. The clear input can be either synchronous or asynchronous. On the other hand, the load input is provided as a means to initialize the counter to any value within its counting range. Figure 8.44 illustrates a typical counter with enable input, load select input, and data load inputs. The counter is assumed to be synchronous. Figure 8.45

```
library ieee ;
use ieee.std_logic_1164.all;

entity modulus6_upcounter is
      port(
            clk, clear        :     in        std_logic;
            Q                 :     buffer    std_logic_vector(2 downto 0));
end syn_upcounter;
architecture circuit_behavior of modulus6_upcounter is
      begin
         process (clk, clear)
         begin
           if clear = '1' then
               Q <= 0;
           elsif (clk' event and clk = '1') then
               if Q = "101" then
                   Q  <=  0;
               else
                   Q  <=  Q+1;
               end if;
           end if;
         end process;
end circuit_behavior;
```

Figure 8.43 VHDL Code Implementation of a Synchronous Modulus-6 Up Counter

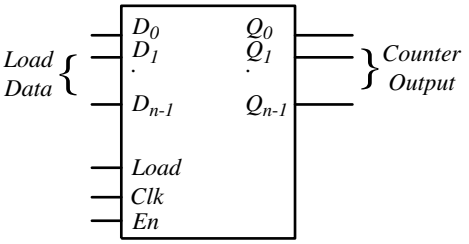


Figure 8.44 Typical Counter with Load and Enable Inputs

illustrates a 4-bit synchronous modulus-10 counter. This counter is also referred to as a *1-bit BCD counter*. The timing diagram of the counter is shown in Figure 8.46.

BCD counters are used in many human interfaces. Typical applications of BCD counters include cascaded BCD counters and digital clocks. The BCD counter illustrated in Figure 8.47 consists of two 1-bit BCD counters cascaded to form a 2-bit (BCD_1BCD_0) BCD counter. The 2-bit BCD counter counts from 00 to 99. Notice that both 1-bit BCD counters are initialed by a load data input of 0. The BCD_0 counter counts (the units) from 0 to 9 at every clock cycle (assume a clock rising edge) because its enable input is active all the time. The output Q_0 and Q_3 are ANDed to trigger the load input to initialize the counter to 0000. The AND gate is active only

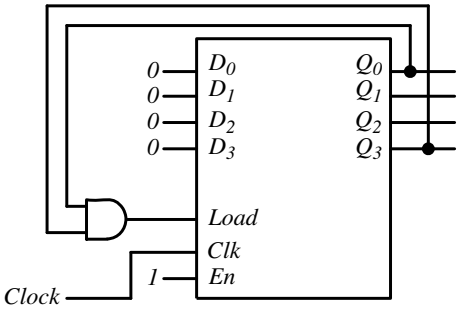


Figure 8.45 One-Bit BCD 0-to-9 Counter

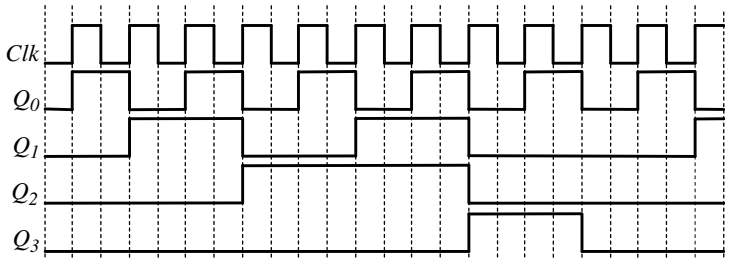


Figure 8.46 Timing Diagram of a 1-Bit BCD 0-to-9 Counter

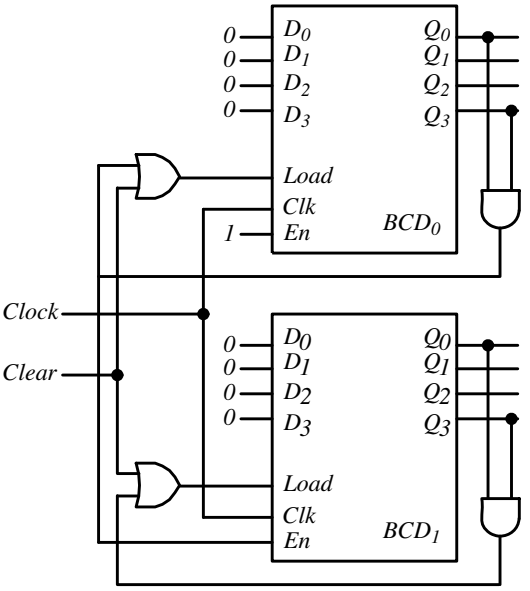


Figure 8.47 Two-Bit BCD 00-to-99 Counter

when both Q_0 and Q_3 are equal to logic level 1, which corresponds to the BCD number 9, after which the counter resets. The BCD_1 counter counts (the tents) from 0 to 9, and it is active only when both Q_0 and Q_3 of the BCD_0 counter are equal to logic level 1. Its enable input is connected to the AND gate of Q_0 and Q_3. Similar to a BCD_0 counter, a BCD_1 counter has its output Q_0 and Q_3 ANDed to force it to reinitialize after it reaches the BCD number 9.

Using the same design approach, one could cascade several 1-bit BCD counters with different counting ranges. Three similar cascaded BCD counters would count from 000 to 999. A BCD counter can be designed to make it possible to change the reinitialization condition. For example, Figure 8.47 could be modified to count from 00 to 59 instead. The BCD1 counter needs to be modified to count from 0 to 5 by ANDing the Q_0 and Q_2 outputs. The VHDL code shown in Figure 8.48 implements a 2-bit BCD counter, which counts from 00 to 99.

8.6.5 Special Counters

There is a special class of synchronous counters, which have been use in the past to provide simple bit patterns for coding and decoding processes. These counters include the Johnson and ring counters. A *Johnson counter* generates a sequence of binary numbers where only one bit position changes between two consecutive numbers. Figure 8.49 illustrates a 4-bit Johnson counter designed with four D flip-flops. The most significant inverted output $\bar{Q}$ (of the last flip-flop on the right) is connected to the input of the least significant bit position (first flip-flop on the left). Starting with an initial value of $Q_0Q_1Q_2Q_3$ equal to 0000, the Johnson counter

```
library ieee ;
use ieee.std_logic_1164.all;

entity bcd_counter is
      port(
            clk, clear, enable    :      in       std_logic;
            bcd0, bcd1            :      buffer   std_logic_vector(3 downto 0));
end bcd_counter;
architecture circuit_behavior of bcd_counter is
      begin
         process (clk)
         begin
            if clk' event and  clk = '1' then
                 if clear = '1' then
                      bcd1 <= "0000"; bcd0 <= "0000";
                 elsif enable = '1' then
                      if bcd0 = "1001" then
                           bcd0 <= "0000";
                           if bcd1 = "1001" then
                                bcd1 <= "0000";
                           else
                                bcd1 <= bcd1 + '1';
                           end if;
                      else
                           bcd0 <= bcd0 + '1';
                      end if;
                 end if;
            end if;
         end process;
end circuit_behavior;
```

Figure 8.48 VHDL Code Implementation of a 2-Bit BCD Counter

periodically generates the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000. Notice that only one bit position changes between two consecutive numbers, as in the case of the Gray code. For a Johnson counter to work properly, it must be reset initially to 0000.

The *ring counter* is a variation of the Johnson counter. Rings counters are used in time-division multiplexing applications, where a series of outputs must be enabled in

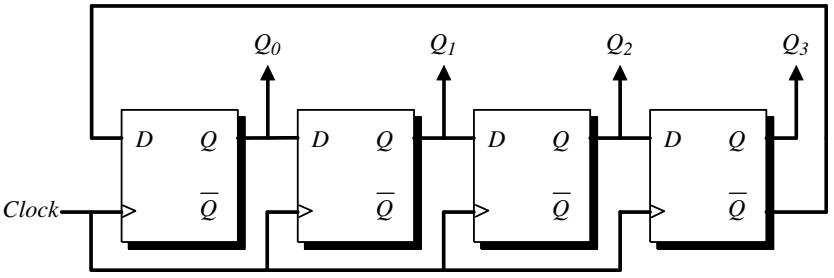


Figure 8.49 Four-Bit Johnson Counter

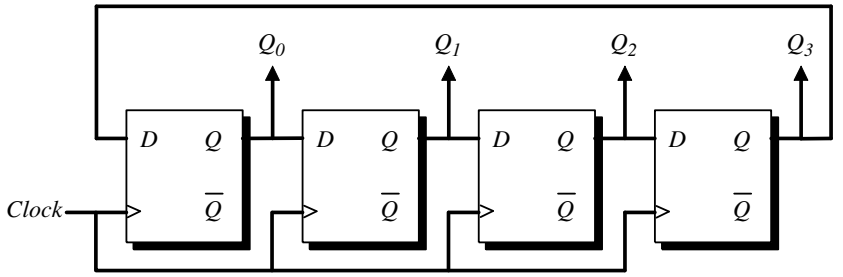


Figure 8.50 Four-Bit Ring Counter

a given time sequence. A typical application of ring counters is their use in display circuits, where lights ripple on and off in a cyclic pattern. Figure 8.50 illustrates a 4-bit ring counter constructed with D flip-flops. Notice that it is the output Q of the last flip-flop which is connected to the input of the first flip-flop. Starting from an initial value of $Q_0Q_1Q_2Q_3$ equal to 1000, the ring counter periodically generates the cyclic sequence 1000, 0100, 0010, 0001, 1000. Notice that only one bit position at a time is equal to logic level 1 in a cyclic pattern. For a ring counter to work properly, it must be reset initially to 1000.

PROBLEMS

8.1 Using only NAND gates, design a gated SR latch.

8.2 Using only NOR gates, design a gated D latch.

8.3 Determine the output Q of an SR latch for the input waveforms depicted in Figure P8.3. Assume that the output Q is reset initially.

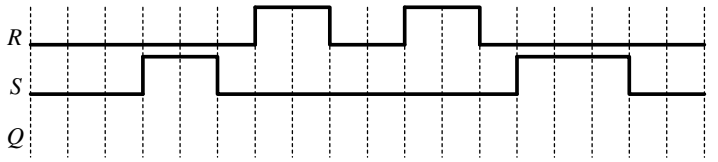


Figure P8.3

8.4 Determine the output Q of a gated SR latch for the input waveforms depicted in Figure P8.4. Assume that the output Q is reset initially.

8.5 Determine the output Q of a gated D latch for the input waveforms depicted in Figure P8.5. Assume that the output Q is reset initially.

8.6 Determine the output Q of a gated D latch for the input waveforms depicted in Figure P8.6. Assume that the output Q is reset initially.

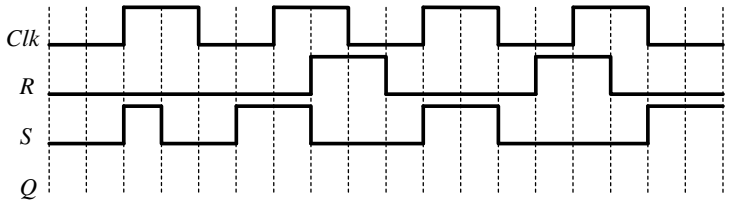


Figure P8.4

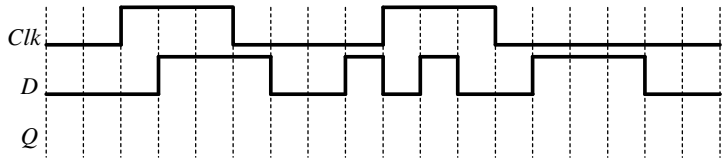


Figure P8.5

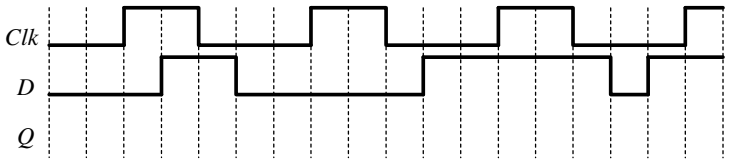


Figure P8.6

8.7 Determine the output Q of a gated D latch for the input waveforms depicted in Figure P8.7. Assume that the output Q is reset initially.

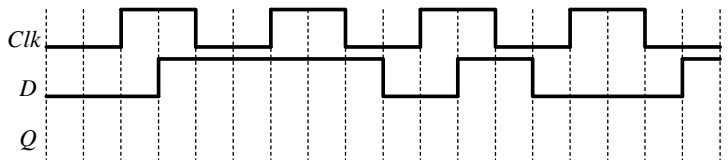


Figure P8.7

8.8 Determine the output Q of a positive edge-triggered D flip-flop for the input waveforms depicted in Figure P8.8. Assume that the output Q is reset initially.

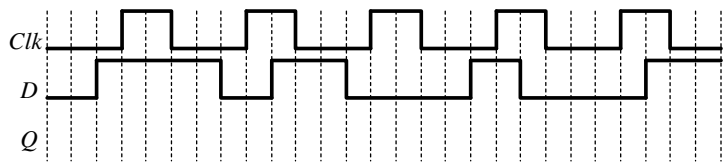


Figure P8.8

8.9 Determine the output Q of a negative edge-triggered D flip-flop for the input waveforms depicted in Figure P8.8. Assume that the output Q is initially set.

8.10 Determine the output Q of a positive edge-triggered D flip-flop for the input waveforms depicted in Figure P8.10. Assume that the output Q is reset initially.

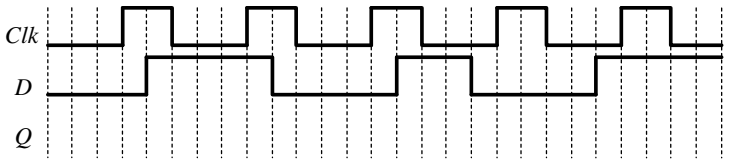


Figure P8.10

8.11 Determine the output Q of a negative edge-triggered D flip-flop for the input waveforms depicted in Figure P8.10. Assume that the output Q is set initially.

8.12 Determine the output Q of a positive edge-triggered JK flip-flop for the input waveforms depicted in Figure P8.12. Assume that the output Q is reset initially.

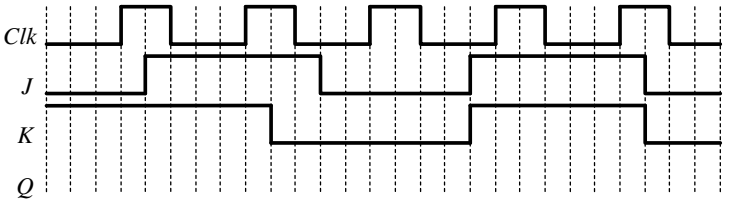


Figure P8.12

8.13 Determine the output Q of a negative edge-triggered JK flip-flop for the input waveforms depicted in Figure P8.12. Assume that the output Q is set initially.

8.14 Determine the output Q of a positive edge-triggered JK flip-flop for the input waveforms depicted in Figure P8.14. Assume that the output Q is reset initially.

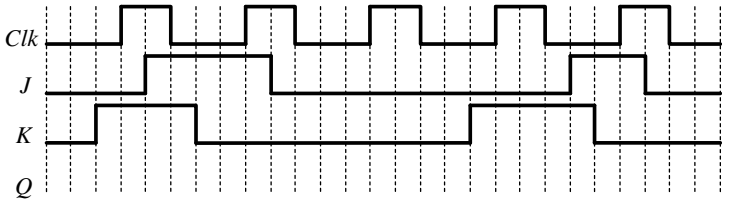


Figure P8.14

8.15 Determine the output Q of a negative edge-triggered JK flip-flop for the input waveforms depicted in Figure P8.14. Assume that the output Q is set initially.

8.16 Determine the output Q of a positive edge-triggered JK flip-flop for the input waveforms depicted in Figure P8.16. Assume that the output Q is reset initially.

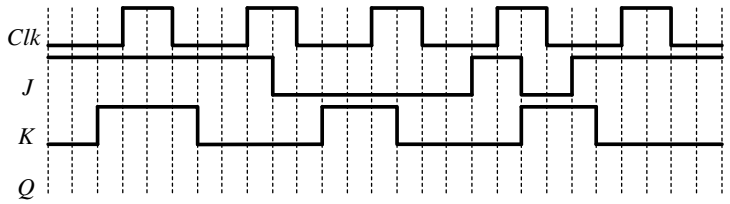


Figure P8.16

8.17 Determine the output Q of a negative edge-triggered JK flip-flop for the input waveforms depicted in Problem 8.16. Assume that the output Q is set initially.

8.18 Design a D flip-flop with a load feature (load input). Use a 2 : 1 multiplexer to load the data in.

8.19 Write VHDL code to implement a gated SR latch.

8.20 Write VHDL code to implement an SR flip-flop.

8.21 Write VHDL code to implement a D flip-flop with a synchronous reset.

8.22 Write VHDL code to implement a D flip-flop with an asynchronous reset.

8.23 Write VHDL code to implement a JK flip-flop.

8.24 Determine the output Q of a positive edge-triggered T flip-flop for the input waveforms in Figure P8.24. Assume that the output Q is reset initially.

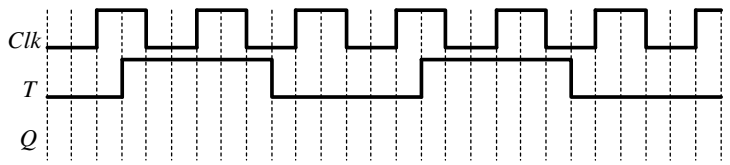


Figure P8.24

8.25 Determine the output Q of a negative edge-triggered T flip-flop for the input waveforms depicted in Figure P8.24. Assume that the output Q is set initially.

8.26 Determine the output Q of a positive edge-triggered T flip-flop for the input waveforms in Figure P8.26. Assume that the output Q is reset initially.

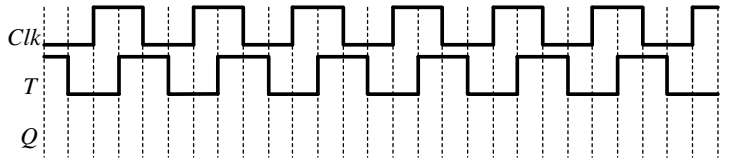
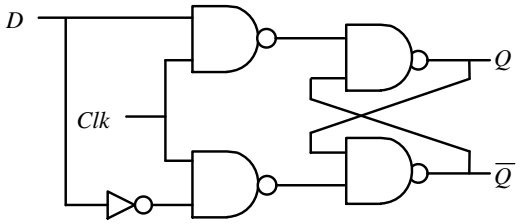


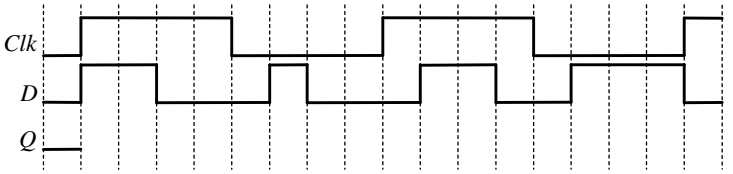
Figure P8.26

8.27 Determine the output Q of a negative edge-triggered T flip-flop for the input waveforms in Figure P8.26. Assume that the output Q is set initially.

8.28 Determine the output Q of the logic circuit in Figure P8.28(a). Assume that the output Q is reset initially and that the propagation delay of the NAND and NOT gates is 1 ns. The timing diagram in Figure P8.28(b) has a 1-ns resolution.



(a)



(b)

Figure P8.28

8.29 Implement the SR flip-flop using a D flip-flop and additional logic gates.

8.30 Implement the JK flip-flop using a T flip-flop and additional logic gates.

8.31 Determine the output of a 4-bit right-shift register after the input sequence 01010101 has been shifted eight times, starting with the most significant bit. Assume that the output of the register is reset initially to 0000. Draw a timing diagram and list the output state.

8.32 Determine the output of a 4-bit right-shift register after the input sequence 00111100 has been shifted eight times, starting with the most significant bit. Assume that the output of the register is set initially to 1111. Draw a timing diagram and list the output state.

8.33 Determine the output sequence of the 4-bit right-shift register depicted in Figure P8.33. Draw a timing diagram and list the output sequence.

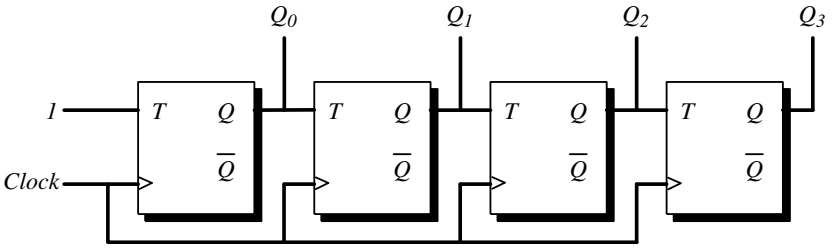


Figure P8.33

8.34 Write VHDL code to implement an 8-bit right-shift register using an **if–else–then** statement.

8.35 Write VHDL code to implement an 8-bit right-shift register constructed with D flip-flops using a component declaration for a D flip-flop.

8.36 Write VHDL code to implement an 8-bit serial-in, parallel-out right-shift register.

8.37 Write VHDL code to implement an 8-bit serial-in, parallel-out right-shift register with an asynchronous reset.

8.38 Write VHDL code to implement an 8-bit serial-in, parallel-out right-shift register with a synchronous reset.

8.39 Design a 4-bit parallel-in, serial-out right-shift register.

8.40 Write VHDL code to implement an 8-bit parallel-in, serial-out right-shift register.

8.41 Design a 4-bit bidirectional shift register.

8.42 Write VHDL code to implement an 8-bit bidirectional shift register.

8.43 Using T flip-flops, design a 4-bit asynchronous up counter.

8.44 Using T flip-flops, design a 4-bit asynchronous down counter.

8.45 Using T flip-flops and other logic gates, design a 4-bit asynchronous up–down counter. Include an additional input to select the counting direction.

8.46 Using T flip-flops, design a 4-bit synchronous up counter.

8.47 Using T flip-flops, design a 4-bit synchronous down counter.

8.48 Write VHDL code to implement an 8-bit synchronous up counter.

8.49 Write VHDL code to implement an 8-bit synchronous down counter.

8.50 Design a 2-bit BCD counter to count from 00 to 59.

8.51 Design a BCD counter to count from 000 to 999.

8.52 Design a 2-bit BCD counter to count from 01 to 12.

8.53 Write VHDL code to implement a 2-bit BCD counter that counts from 00 to 59.

8.54 Write VHDL code to implement a 2-bit BCD counter that counts from 01 to 12.

8.55 Write VHDL code to implement a 3-bit BCD counter that counts from 000 to 999.

8.56 Determine the output sequence of the 4-bit ring counter depicted in Figure P8.56. Assume that the initial output state is equal to 1000. Draw a timing diagram and list the output sequence.

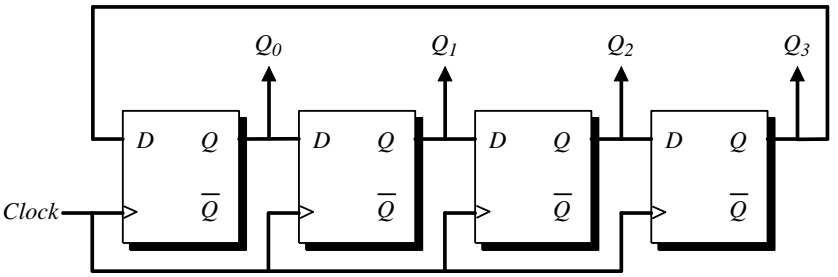


Figure P8.56

8.57 Determine the output sequence of the 4-bit Johnson counter depicted in Figure P8.57. Assume that the initial output state is equal to 0000. Draw a timing diagram and list the output sequence.

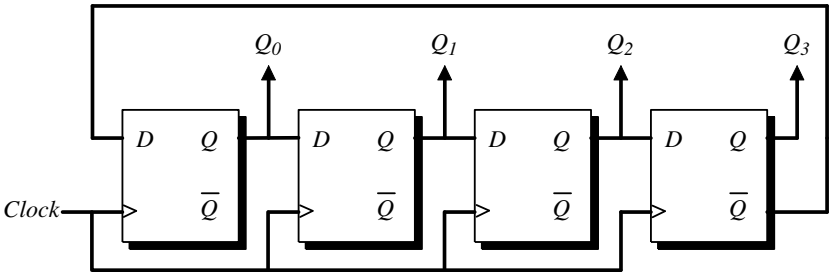


Figure P8.57

8.58 Write VHDL code to implement an 8-bit Johnson counter.

8.59 Write VHDL code to implement an 8-bit ring counter.

9 Synchronous Sequential Logic

9.1 OBJECTIVES

The objectives of the chapter are to describe:

- Synchronous sequential design
- Synthesis of a finite-state machine
- Analysis of a finite-state machine
- Flip-flop selection
- State assignment
- State optimization
- FSM VHDL programming

9.2 SYNCHRONOUS SEQUENTIAL CIRCUITS

In a combinational circuit, the outputs at any point in time are fully determined by the inputs present at that point in time. In a sequential logic circuit, the outputs depend not only on the present inputs but also on the past behavior of the circuit. Thus, a sequential logic circuit consists of a combinational circuit and memory elements that form a feedback system, as illustrated in Figure 9.1.

The sequential circuits described in Chapter 8 perform simple functions such as shifting and counting. Registers shift data inputs in response to a clock signal. Similarly, counters generate a predetermined sequence of states and have no inputs other than the initial conditions and the clock signal. The counters designed in Chapter 8 were constructed by cascading flip-flops to count in an orderly fashion. Changing the count order requires a sequence of data input. The design methods designed so far are not adequate to design a complex sequential circuit.

A sequential circuit which has additional inputs that may change its present state is also referred to as a *finite-state machine* (FSM). In general, the output sequence of an

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.

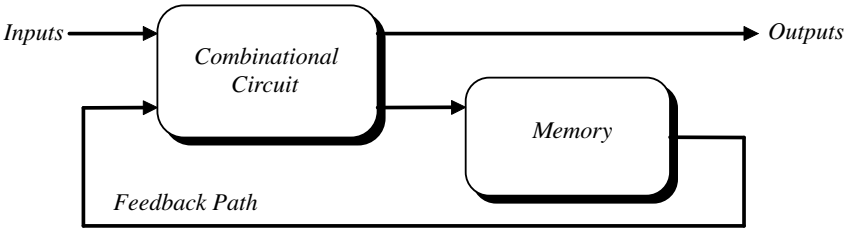


Figure 9.1 General Structure of a Sequential Circuit

FSM depends on the input sequence and the present state of flip-flops of the FSM. There are two classes of sequential circuits, synchronous and asynchronous. If a clock signal is used to control the operation of a sequential circuit, the circuit is known as a *synchronous sequential circuit*. *Asynchronous sequential circuits* do not require a clock signal but rely on the time delays of its components. Propagation delays of logic gates are used as timing delays to accomplish the required feedback. In general, synchronous circuits are easier to design and are used in the vast majority of practical applications. The general structure of a synchronous sequential circuit is illustrated in Figure 9.2. Notice that a clock signal is applied only to the memory section of an FSM. In most synchronous sequential circuits, the state of the memory changes on the transition of the clock signal.

The inputs of a combinational circuit consist of the inputs and the present state of the memory, and the outputs of a combinational circuit are the updated inputs of the memory. The memory of the circuit consists of flip-flops. A combinational circuit depends on the type of flip-flops used.

In this chapter we examine simple synchronous sequential circuits, which can implement only a fixed number of possible states. The counters designed in Chapter 8 are rather simple finite-state machines. Their outputs and states are identical, and there is no choice of the sequence in which states are selected. The outputs and next state of a finite-state machine are combinational logic functions of their inputs and present states. A simple timing diagram may not be adequate to determine the state sequence of the flip-flops. Additional design tools such as a state diagram, a state table, and a state assigned table will be introduced to describe the behavior of a finite-state machine.

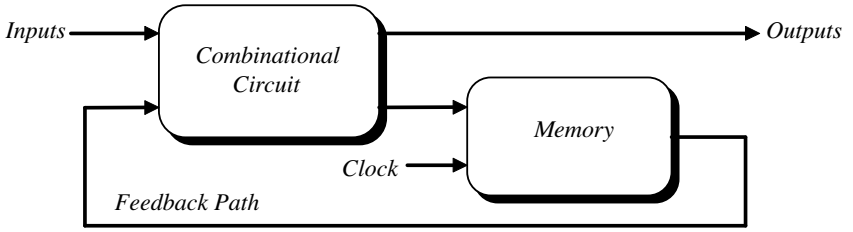


Figure 9.2 General Structure of a Synchronous Sequential Circuit

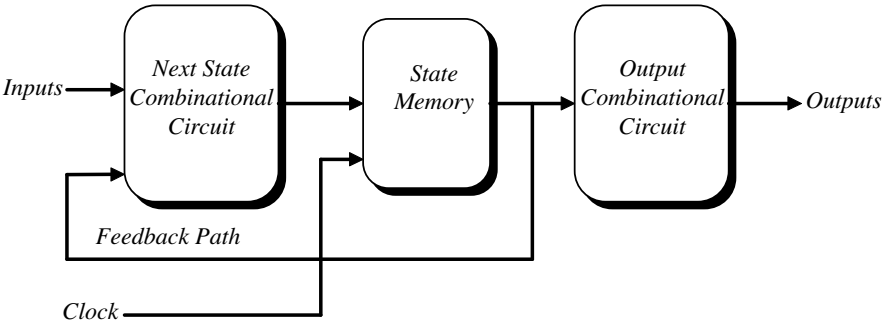


Figure 9.3 Moore Model Structure of a Finite-State Machine

9.3 FINITE-STATE MACHINE DESIGN CONCEPTS

The general structure of a sequential circuit consists of two parts: flip-flops and combinational circuits. The flip-flops hold the state memory of the sequential circuit. Depending on how the output of the finite-state machine is related to the state of the flip-flops, two structure models can be constructed:

1. ***The Moore model***. In the Moore model the output of the finite-state machine depends only on the present state of the flip-flops. In other words, the output does not depend on the present inputs but, rather, on the previous inputs. The output does not explicitly include the inputs in its logic expression. The output is not valid until the flip-flops are updated during the current clock cycle. Figure 9.3 illustrates the structure of the Moore model. There is no direct connection between the inputs and the outputs of a finite-state machine. The output of a finite-state machine is a function of its present state.
2. ***The Mealy model***. The Mealy model is the other type of sequential circuit. The Mealy model is a variation of the Moore model in which the output of the finite-state machine depends on the present states of the flip-flops and the present inputs of the finite-state machine. The logic expression of the output depends explicitly on the inputs of the finite-state machine. The output is valid before the flip-flops are updated during the current clock cycle. Figure 9.4 illustrates the structure of the Mealy model.

In either model, the logic expressions of the combinational circuits, the number of states, and the transitions between states are necessary to design a finite-state machine. In the following sections we define terms that will be used to determine the states and the combinational logic circuits of a finite-state machine.

9.3.1 State Diagram

The behavior of a finite-state machine can be described by many different representations. A *state diagram* is a graphical representation that consists of circles (nodes) and

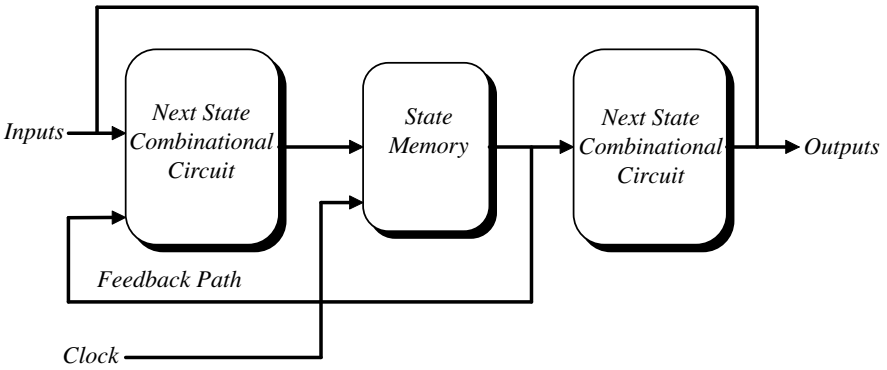


Figure 9.4 Mealy Model Structure of a Finite-State Machine

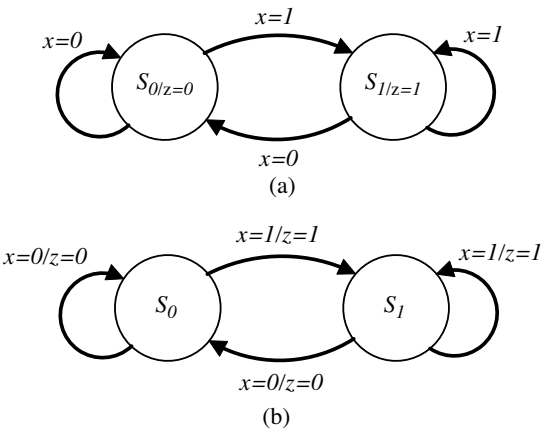


Figure 9.5 State Diagrams for (a) Moore and (b) Mealy Models

directional arcs. The nodes represent the states and the arcs represent the transitions between the states. Two state diagram samples, which may not represent the same finite-state machine, are illustrated in Figure 9.5. In both state diagrams x is the input and z is the output.

In the Moore model, each state is appended with an output value. That is because the output of a Moore finite-state machine is a function of its present state. The arcs are labeled with input values, which cause the corresponding transition between two states. In contrast, in the Mealy model, each state is not appended with an output value, but the transition arcs are labeled with the input values and the output value is separated by a slash. In either model, a starting state is selected to receive the reset conditions.

9.3.2 State and State Assigned Tables

A *state table* is a direct enumeration of a state diagram into a useful table. The state table, also referred to as a *state transition table*, is one step closer to FSM circuit

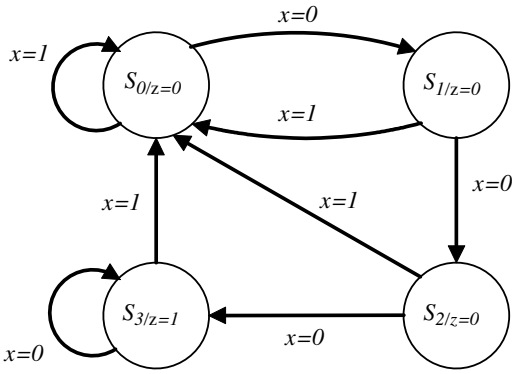


Figure 9.6 State Diagram

implementation. Consider the state diagram in Figure 9.6. The state table consists of three columns: the present state, the next state, and the output. Alphanumeric letters represent the states of the FSM. Under this form, the state table has a very limited use. Figure 9.7 illustrates the state table of the state diagram shown in Figure 9.6. The state table has four states: S_0, S_1, S_2, and S_3. Each state can be implemented by a flip-flop. It takes 2-bit binary numbers to represent the four states. Assigning binary numbers to the states leads to the state assigned table. The state assigned table is exactly similar to the state table, except that the states are represented by binary numbers. The resulting state assigned table is illustrated in Figure 9.8. States S_0, S_1, S_2, and S_3 are assigned the values 00, 01, 10, and 11, respectively.

The state table in Figure 9.8 is called a *state assigned table* because binary numbers are assigned selectively to the states. The present states are designated as y_2 and y_1 and the next states are designated as Y_2 and Y_1. A different state assignment would lead to the state assigned table shown in Figure 9.9. Notice that states S_0, S_1, S_2, and S_3 are assigned the values 00, 01, 11, and 10 instead. We will discover in the next sections that different state assignments have different logic implementations with various degrees of circuit complexity. Finding the appropriate state assignment is not a trivial matter. CAD tools usually have proprietary search methods to find the optimum implementation.

Present State	Next State		Output
	$x = 0$	$x = 1$	z
S_0	S_1	S_0	0
S_1	S_2	S_0	0
S_2	S_3	S_0	0
S_3	S_3	S_0	1

Figure 9.7 State Table of the State Diagram in Figure 9.6

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	01	00	0
01	10	00	0
10	11	00	0
11	00	00	1

Figure 9.8 State Assigned Table of the State Table in Figure 9.7

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	01	00	0
01	11	00	0
11	10	00	0
10	00	00	1

Figure 9.9 Alternative State Assigned Table of the State Table in Figure 9.7

9.3.3 Next-State and Output Logic Functions

The state assigned table lists the inputs, the present states, the next states, and the outputs of the finite-state machine in a tabular form similar to that of a truth table. The inputs and the present states are the input columns, and the next states and the output are the output columns.

$$\begin{aligned} \textit{nextstates} &= \text{function}\,(\textit{present states}, \textit{inputs}) \\ \textit{output} &= \text{function}\,(\textit{present states}, \textit{inputs}) \rightarrow \text{Mealy model} \\ \textit{output} &= \text{function}\,(\textit{present states}) \rightarrow \text{Moore model} \end{aligned}$$

Therefore, the next states and the outputs of the finite-state machine can be expressed and simplified using algebraic or graphical manipulations (Karnaugh maps). The next states and outputs can then be implemented using logic gates and flip-flops. Because the present states are delayed versions of the next states, D flip-flops are the better choice. In later sections we explore other types of flip-flops by modifying and introducing new columns in the state assigned table.

9.3.4 Finite-State Machine Design Procedures

Now that we have defined a few important terms, the following procedure summarizes the steps in the design of a synchronous sequential circuit (FSM).

1. Obtain the behavior specifications of the finite-state machine.
2. Design a state diagram and state table from a word description using state labels.
3. Select a starting state for reset conditions.
4. Minimize the number of states, if necessary.
5. Choose state variables and assign binary values to the states.
6. Design a state assigned table from state table or state diagram.
7. Choose a flip-flop type to implement FSM memory.
8. Derive the simplified next-states and output logic expressions.
9. Draw the logic circuit for the finite-state machine.

9.4 FINITE-STATE MACHINE SYNTHESIS

Synthesis is the process of designing a finite-state machine from a problem statement which describes its specifications and behavior. We use the design procedures provided in Section 9.3.4, with the exception of state minimization, which is covered in detail in Section 9.8. We will implement a simple finite-state machine using Moore and Mealy models to illustrate similarities and differences between the two models. The finite-state machine is designed using D flip-flops. In Section 9.8 we describe how to use other flip-flops, such as JK and T flip-flops.

Below we work with a finite-state machine with one input, x, and one output, z. Output z is equal to 1 when the sequence 111 has been detected at the input; otherwise, output z is equal to 0. This finite-state machine is referred to as a *sequence detector*.

9.4.1 Moore Model Design

Recall that in the Moore model, the output of a finite-state machine depends only on the present states of the machine. Therefore, input sequence 111 would have occurred in the preceding three clock cycles for output z to change to 1. A sample input–output time sequence is illustrated in Figure 9.10. Notice that output z changes to 1 only if input x was equal to 1 during the preceding three clock cycles. As long as input x is equal to 0, there is no change in the state memory of output z of the FSM. Therefore, we select state S_0 as a starting state, where input x is equal to 0 and output z is equal to 0. If input x remains equal to 0, the FSM remains in state S_0 and output z remains equal to 0. If input x changes to 1, the first bit of the desired input sequence 111 is detected. The FSM moves to state S_1 to indicate that the first bit of the sequence has been detected. Output z remains equal to 0, however, because the entire sequence has not yet been detected.

x	0	0	1	1	0	1	1	1	0	0	0	1	1	1	1	0	0
z	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0

Figure 9.10 Input–Output Sequence Sample of the Sequence Detector FSM

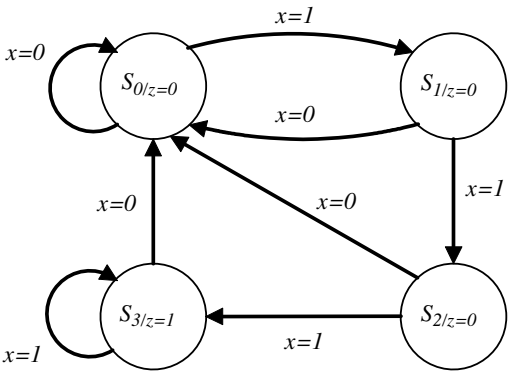


Figure 9.11 State Diagram of the Moore Model Sequence Detector FSM

Once at state S_1, if input x changes to 0, the FSM returns to state S_0. However, if input x remains equal to 1, the FSM moves to state S_2. The second bit of the desired input sequence has been detected. Output z remains equal to 0. Once at state S_2, if input x is changed to 0, the FSM returns to state S_0 because the sequence has been broken. Output z remains at 0. However, if input x remains equal to 1, the FSM moves to state S_3. Output z becomes equal to 1 because the desired sequence 111 has been detected. If input x remains equal to 1, the FSM remains in S_3 because the FSM keeps detecting a moving sequence of 111. Output z remains equal to 1. However, if input x changes to 0, the FSM returns to state S_0, and the output becomes equal to 0.

The state diagram shown in Figure 9.11 summarizes the states and state transitions of the Moore type of the sequence detector FSM. Notice how the states are appended with the output z values. The transitions between states are labeled with the changes in input x. From the state diagram the state table is constructed and is shown in Figure 9.12. Notice that the output column does not depend on input x as the next state does. Since there are four states, it takes 2-bit binary numbers to represent the states of the FSM. Assigning the binary values 00, 01, 10, and 11 to states S_0, S_1, S_2, and S_3, respectively, the corresponding state assigned table is illustrated in Figure 9.13. The present states are designated y_2 and y_1 and the next states are designated Y_2 and Y_1. Once the state assigned table has been constructed, the next-to-last task is to determine the logic expressions of the next states and the output of the

Present State	Next State		Output z
	$x = 0$	$x = 1$	
S_0	S_0	S_1	0
S_1	S_0	S_2	0
S_2	S_0	S_3	0
S_3	S_0	S_3	1

Figure 9.12 State Table of the FSM in Figure 9.11

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	00	01	0
01	00	10	0
10	00	11	0
11	00	11	1

Figure 9.13 State Assigned Table of the FSM in Figure 9.11

FSM. Using the Karnaugh map method, the optimized logic expressions are as shown in Figure 9.14.

The final step of synthesis is to design a finite-state machine using logic gates and flip-flops. D flip-flops are the favorite choice due to their straightforward application. Next states Y_1 and Y_2 become the inputs D of the flip-flops and present states y_1 and y_2

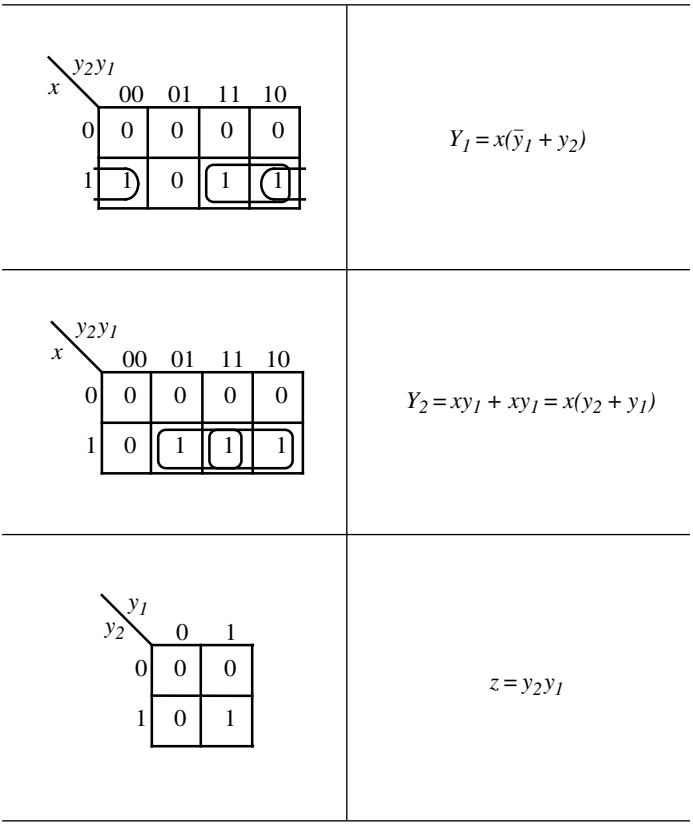


Figure 9.14 Next-State and Output Logic Expressions of the FSM in Figure 9.13

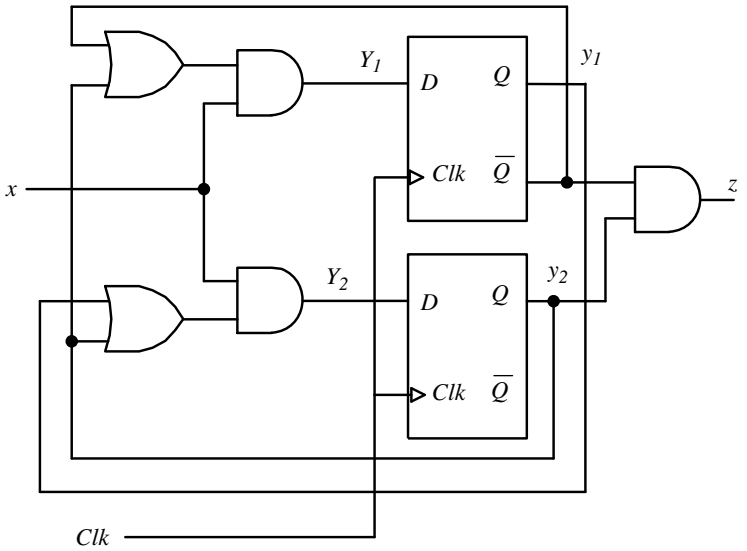


Figure 9.15 Logic Implementation of the FSM in Figure 9.13

are the outputs Q of the flip-flops. Sometimes, however, other types of flip-flops may reduce the combinational circuitry. The next-state and output expressions are implemented using logic gates, as illustrated in Figure 9.15. Notice that output z is a function of present states y_1 and y_2 only, whereas next states Y_1 and Y_2 are functions of present states y_1 and y_2 and input x.

For certain finite-state machines, the number of states is not a power of 2. The states of the FSM should be expended to include don't-care states to obtain a number of states that is a power of 2. Notice that the exponent or the power of the number is exactly equal to the number of flip-flops needed to implement the FSM. For example, if an FSM has five states, the user needs to add three additional don't-care states to obtain a number of states equal to 8 (the next power of 2 number). The number of flip-flops required to implement the finite-state machine would be three since 8 is equal to 2^3.

9.4.2 VHDL Implementation of a Moore Model

A finite-state machine can be described using VHDL programming. Because an FSM is a sequential circuit, there are several ways to implement it in VHDL code using control and decision-making statements. The VHDL code implementation of the FSM in Figure 9.13 is illustrated in Figure 9.16. The entity of the FSM describes only the input and output ports: that is, the input x, the output z, and the clock signal *Clk*. The states of the FSM are described in the architecture. Notice that the VHDL code uses the reserved keyword **type**, which allows the user to define unique data types. The state_type is described to have four possible states: S_0, S_1, S_2, and S_3. Using this unique description, the signal y is defined as the present state of the FSM. Notice that the bit

```
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_detector is
      port(
            x, clk          :      in          std_logic;
            z               :      out         std_logic);
end sequence_detector;
architecture circuit_behavior of sequence_detector is
         type state_type is (S0, S1, S2, S3);
         signal y : state_type;
      begin
         process (clk)
         begin
            if clk' event and clk = '1' then
                case y is
                  when S0 =>
                       if x = '0' then
                          y <= S0;
                       else
                          y <= S1;
                       end if;
                  when S1 =>
                       if x = '0'  then
                          y <= S0;
                       else
                          y <= S2;
                       end if;
                  when S2 =>
                       if x = '0' then
                          y <= S0;
                       else
                          y <= S3;
                       end if;
                end case;
            end if;
         end process;
         z <= '1' when y = S3 else '0';
end circuit_behavior;
```

Figure 9.16 VHDL Implementation of the FSM Table in Figure 9.13

size of the present state signal y is not specified. The VHDL compiler will automatically generate the number of flip-flops required and assign the appropriate binary numbers to states S_0, S_1, S_2, and S_3. CAD tools use proprietary methods to generate optimized circuit implementations.

9.4.3 Mealy Model Design

In the Mealy model, the outputs of the finite-state machine depend not only on the present states of the FSM but also on the inputs. For the sequence detector example, the desired input sequence 111 is detected during three clock cycles: two previous

x	0	0	1	1	0	1	1	1	0	0	0	1	1	1	1	0	0
z	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0

Figure 9.17 Modified Input–Output Sequence of the Sequence Detector FSM

clock cycles and the present clock cycle (Figure 9.17). The same input sample sequence as in the Moore model is used to compare the behavior of the Mealy model to that of the Moore model. Notice that the sequence 111 is detected in three consecutive clock cycles, including the present cycle. Similar to the Moore model, consider state S_0, where input x is equal to 0 and output z is equal to 0. If input x changes to 1 and output z remains at 0, the FSM moves to state S_1. Once at state S_1, if x changes back to 0, the FSM returns to state S_0. However, if input x remains at 1, the FSM moves to state S_2, with output z equal to 0. Once at state S_2, if input x changes to 0, the FSM moves to state S_0. If input x remains at 1, the FSM remains in state S_2 and output z changes to 1 during the same clock cycle. The desired input sequence 111 has been detected. If input x continues to be equal to 1, the FSM will remain in state S_2 because the sequence 111 is being detected continuously. The state diagram shown in Figure 9.18 graphically depicts the behavior of the FSM.

The state table illustrated in Figure 9.19 is constructed from the state diagram. The FSM has three states: S_0, S_1, and S_2. Notice that the output column depends on input x and the present states, as do the next states. Two-bit binary numbers are required to represent the three states of the FSM. Assigning the binary numbers 00, 01, and 10 to states S_0, S_1, and S_2, respectively, the state assigned table is illustrated in Figure 9.20. Notice that the number of states is not a power of 2. An additional don't-care state is added and assigned the binary number 11. Two flip-flops are required to store the state memory of the FSM. The present states are designated with y_2 and y_1 and the next states are designated with Y_2 and Y_1. Next states Y_2 and Y_1 and output z are mapped into

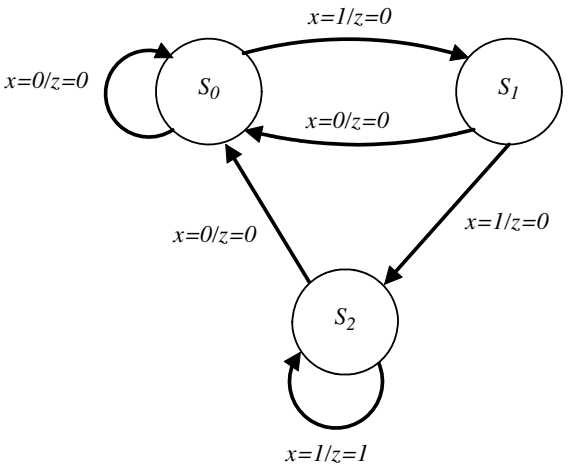


Figure 9.18 State Diagram of the Mealy Model Sequence Detector FSM

Present State	Next State		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S_0	S_0	S_1	0	0
S_1	S_0	S_2	0	0
S_2	S_0	S_2	0	1

Figure 9.19 State Table of the FSM in Figure 9.17

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	00	01	0	0
01	00	10	0	0
10	00	10	0	1
11	dd	dd	d	d

Figure 9.20 State Assigned Table of the FSM in Figure 9.17

Karnaugh maps as shown in Figure 9.21. Notice that the don't-care state is also mapped and used when appropriate. The logic expressions for the next states and output are derived and listed in Figure 9.21. In this case, the don't-care state was helpful in obtaining simpler logic expressions. Using D flip-flops, the Mealy-type

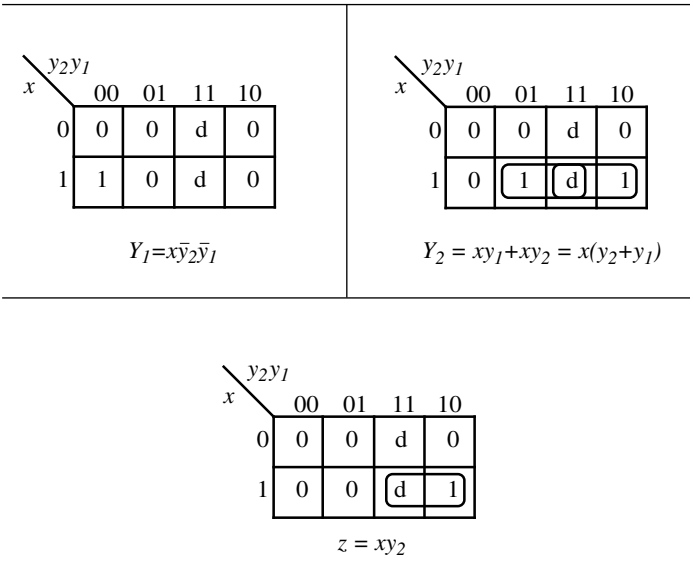


Figure 9.21 Next-State and Output Logic Expressions of the FSM in Figure 9.20

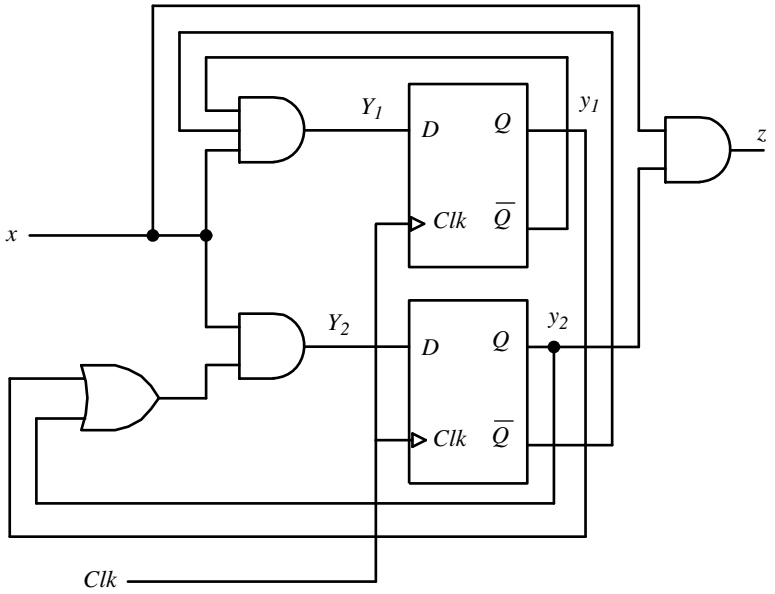


Figure 9.22 Logic Implementation of the FSM in Figure 9.20

sequence detector finite-state machine is implemented by the logic circuit shown in Figure 9.22.

9.4.4 VHDL Implementation of the Mealy Model

The VHDL code shown in Figure 9.23 implements a Mealy-type FSM. Notice that the **type** attribute was used to define the state_type signal. Neither the bit size of the state_type signal nor the number of flip-flops was specified explicitly. The VHDL compiler automatically generates the number of flip-flops required and assigns the appropriate binary numbers to states S_0, S_1, and S_2 to implement the FSM.

9.5 STATE ASSIGNMENT

Recall that the state assigned table is constructed by assigning randomly binary numbers to the states of the FSM. The problem is that there is no priori knowledge to determine which state assignment sequence would produce a minimum circuit implementation. To illustrate the effects of state assignment, consider the state assigned table shown in Figure 9.13. An alternative state assigned table is shown in Figure 9.24, where the binary values of states S_2 and S_3 were switched. States S_2 and S_3 are assigned the binary numbers 11 and 10, respectively.

Next states Y_1 and Y_2 and output z logic expressions are derived using the Karnaugh maps shown in Figure 9.25. Using D flip-flops, the circuit implementation of the FSM is illustrated in Figure 9.26. Notice that the combinational circuit of the FSM has fewer

```
library ieee ;
use ieee.std_logic_1164.all;

entity Mealy_fsm is
      port(
            x, clk          :       in          std_logic;
            z               :       out         std_logic);
end Mealy_fsm;
architecture circuit_behavior of Mealy_fsm is
        type state_type is (S0, S1, S2);
        signal y : state_type;
      begin
        process (clk)
        begin
          if clk' event and clk = '1' then
              case y is
                when S0 =>
                    if x = '0' then y <= S0;
                    else y <= S1;
                    end if;
                when S1 =>
                    if x = '0' then y <= S0;
                    else y <= S2;
                    end if;
                when S2 =>
                    if x = '0' then y <= S0;
                    else y <= S2;
                    end if;
              end case;
          end if;
        end process;

        process (x,y)
        begin
            case y is
                when S0 => z <= '0';
                when S1 => z <= '0';
                when S2 => z <= x;
            end case;
        end process;
end circuit_behavior;
```

Figure 9.23 VHDL Implementation of the FSM in Figure 9.20

logic gates than the implementation circuit in Figure 9.15. This example illustrates that different state assignments generate different implementation circuits with varying degrees of complexity. Although there is no simple way to guess an appropriate state assignment, a visual inspection of Karnaugh maps of an initial state assignment would give insights on how to modify the Karnaugh maps to obtain reduced implicants, which generate combinational circuits with fewer logic gates. Of course, for a larger number of states, visual inspections become impossible. As mentioned before, modern CAD tools use proprietary methods to generate optimized implementations.

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	00	01	0
01	00	11	0
11	00	10	0
10	00	10	1

Figure 9.24 Alternative State Assigned Table of the FSM in Figure 9.11

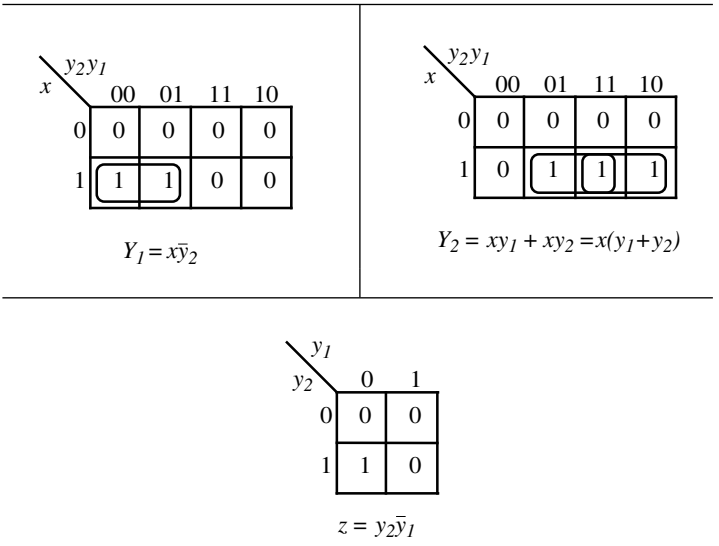


Figure 9.25 Next-State and Output Logic Expressions of the FSM in Figure 9.24

9.6 ONE-HOT ENCODING METHOD

One-hot encoding is an alternative state assignment method which attempts to minimize the combinational logic by increasing the number of flip-flops. The goal of the method is to try to reduce the number of connections between the logic gates in the combinational circuit of the FSM. The presence of more gate interconnections results into longer propagation delays and a slower FSM. Since the propagation delay through the flip-flops is faster, FSMs require fewer logic gates but not necessarily fewer flip-flops.

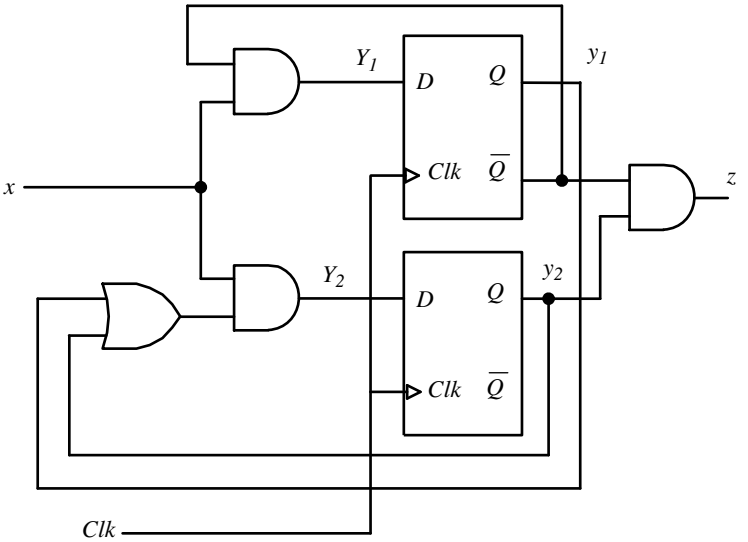


Figure 9.26 Logic Implementation of the FSM in Figure 9.24

One-hot encoding assigns one flip-flop for each state. For example, a finite-state machine with N states requires N flip-flops. The states are assigned N-bit binary numbers; where only the corresponding bit position is equal to 1, the remaining bits are equal to 0. For example, in a finite-state machine with four states S_0, S_1, S_2, and S_3, the states are assigned the binary values 0001, 0010, 0100, and 1000, respectively. Notice that only one bit position is equal to 1; the other bits are all equal to 0. The reamaining 12 binary combinations are assigned to don't-care states. Consider the Mealy-type finite-state machine described by the state diagram shown in Figure 9.19. The state diagram has three states: S_0, S_1, and S_2. One-hot encoding assigns the binary number values 001, 010, and 100 to the states, as illustrated in Figure 9.27.

Three flip-flops are required to store the state memory of the FSM. The present states are designated y_3, y_2, and y_1 and the next states are designated Y_3, Y_2, and Y_1. Notice that the number of next states and present states is increased. Next states Y_3, Y_2, and Y_1 and output z are mapped into Karnaugh maps as shown in Figure 9.28. The 12

Present State	Next State		Output
	$x = 0$	$x = 1$	
$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
001	001	010	0
010	001	100	0
100	001	100	1

Figure 9.27 One-Hot State Assignment of the FSM in Figure 9.19

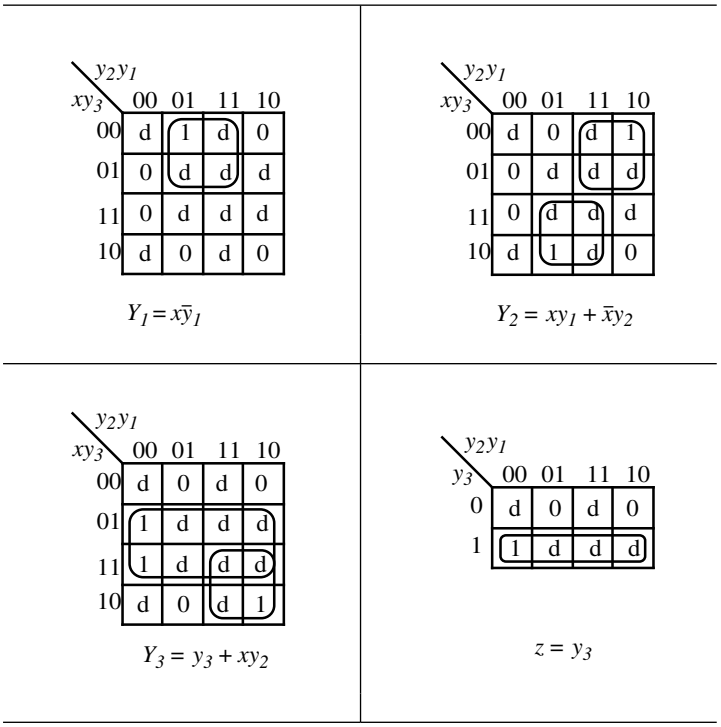


Figure 9.28 Next-State and Output Logic Expressions of the FSM in Figure 9.27

don't-care states are also mapped into Karnaugh maps and are used to simplify the next-state and output logic expressions. The logic expressions derived do not seem to simplify the combinational logic. In fact, for smaller numbers of states, one-hot encoding does not necessarily generate minimum circuit implementations. However, one-hot encoding has proven to generate minimum implementations as the number of states increases.

The circuit implementation of a FSM using one-hot encoding is left to the reader as an exercise. Compare the combinational circuit to that of Figure 9.22. Are the propagation delays any better than those of Figure 9.22?

9.7 FINITE-STATE MACHINE ANALYSIS

Analysis of a finite-state machine is the process of finding the function of the FSM by determining the relationships among the inputs, the outputs, and the states of the flip-flops. Recall that synthesis of a finite-state machine is the process of finding a circuit implementation that satisfies the behavior of the FSM. On the other hand, analysis is breaking the FSM apart to determine its behavior and eventually its function.

We adopt the following steps to analyze a finite-state machine.

1. Identify the inputs and outputs of the finite-state machine.
2. Determine the logic expressions of the next states and the outputs simply by reading the logic networks that interconnect the inputs and outputs of the flip-flops.
3. Determine the number of all possible states, including the don't-care states. Recall that the number of states is equal to 2^N, where N is the number of flip-flops of the FSM.
4. Determine the necessary bit size of the binary numbers required to represent the states of the FSM.
5. Decide on a state assignment and construct a state assigned table using the next-state and output logic expressions.
6. Construct a state table from the state assigned table.
7. Construct a state diagram from the state table if necessary.
8. Draw or list a sample input–output timing sequence that describes the behavior of the FSM.
9. Determine the function of the FSM.

Consider the finite-state machine logic circuit illustrated in Figure 9.29. We will attempt to find the function and behavior of the finite-state machine using the steps listed above. The finite-state machine has one input, x, and one output, z. Its operations are synchronized by the clock signal *Clk*. The state memory of the finite-state machine consists of two D flip-flops. The next states and present states

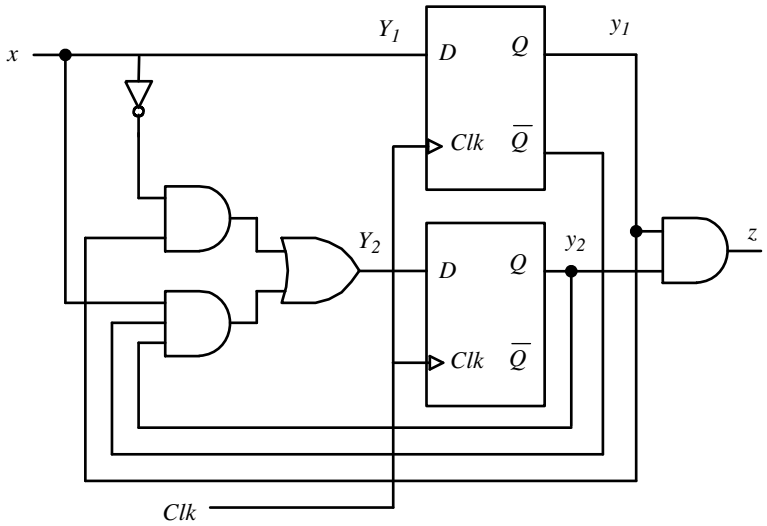


Figure 9.29 FSM Circuit

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	00	01	0
01	10	01	0
10	00	11	0
11	10	01	1

Figure 9.30 State Assigned Table of the FSM Circuit in Figure 9.29

Present State	Next State		Output z
	$x = 0$	$x = 1$	
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_0	S_3	0
S_3	S_2	S_1	1

Figure 9.31 State Table of the FSM Circuit in Figure 9.29

are designated Y_2 and Y_1 and y_2 and y_1, respectively. Y_2 and Y_1 are the inputs of the flip-flops, and y_2 and y_1 are the outputs of the flip-flops. From the combinational logic that interconnects the inputs and outputs of the flip-flops, the next-state and output logic expressions can be written as follows:

$$\begin{aligned} Y_1 &= x \\ Y_2 &= x\bar{y}_1 + xy_2\bar{y}_1 \\ z &= y_2y_1 \end{aligned}$$

Because the finite-state machine has two flip-flops, the FSM may have a maximum of four states S_0, S_1, S_2, and S_3, where one state may be a don't-care state. By assigning the binary numbers 00, 01, 10, and 11 to states S_0, S_1, S_2, and S_3, respectively, the state assigned table shown in Figure 9.30 is constructed by evaluating next states Y_2 and Y_1 and output z logic expressions using all possible values for input x.

From the state assigned table of Figure 9.30, the state table is constructed and illustrated in Figure 9.31. Comparing the state table in Figure 9.31 to that in Figure 9.12, one could recognize that the logic circuit in Figure 9.29 is a sequence detector FSM which detects the sequence 101.

9.8 SEQUENTIAL SERIAL ADDER

Sequential serial adders are economically efficient and simple to build. A serial adder consists of a 1-bit full-adder and several shift registers. In serial adders, pairs of bits

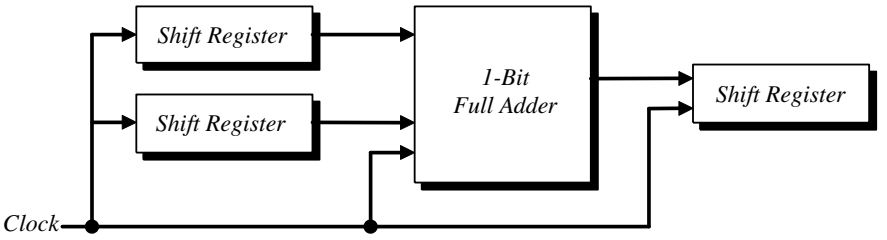


Figure 9.32 Block Diagram of a Serial Adder

A	*B*	*S*	s_i	c_{i+1}
1011	0011	0000	0	1
0101	0001	1000	1	1
0010	0000	1100	1	0
0001	0000	1110	1	0

Figure 9.33 Time Sequence of the Operation of a 4-bit Serial Adder

are added simultaneously during each clock cycle. Two right-shift registers are used to hold the numbers (*A* and *B*) to be added, while one left-shift register is used to hold the sum (*S*). A block diagram of a serial adder is shown in Figure 9.32.

A finite-state machine adder performs the addition operation on the values stored in the input shift registers and stores the sum in a separate shift register during several clock cycles. During each clock cycle, two input bits a_i and b_i are shifted from the two input right-shift registers into the 1-bit full-adder, which adds the two bits and evaluates the sum bit s_i and the carryout bit c_{i+1}. The sum bit s_i is shifted out to the left-shift register and the carryout bit c_{i+1} is stored in the state memory of the serial adder for the next two bits. The time sequence of the operation of a 4-bit serial adder is illustrated in Figure 9.33.

The state memory of a serial adder can only hold a bit for the carryout from a single 2-bit addition. Thus, the FSM for a serial adder will have two states: one state when the carryout is equal to 0 and the other state when the carryout is equal to 1. The carryout c_{i+1} depends on output s_i and inputs a_i and b_i and is added to the inputs in the same clock cycle. The state diagram shown in Figure 9.34 illustrates a Mealy model for the serial adder. Starting from state S_0, where the carryout is equal to 0, if the carryout

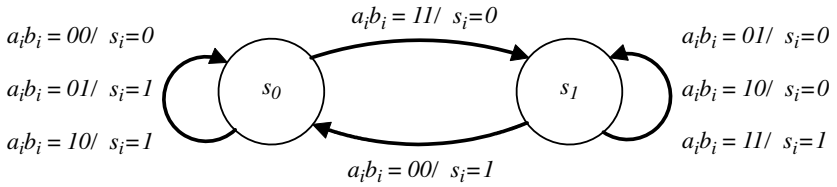


Figure 9.34 State Diagram of a Mealy Model Serial Adder

Present State	Next State				Output s_i			
	a_ib_i				a_ib_i			
	00	01	10	11	00	01	10	11
S_0	S_0	S_0	S_0	S_1	0	1	1	1
S_1	S_0	S_1	S_1	S_1	0	0	0	1

Figure 9.35 State Table of a Mealy Model Serial Adder

remains equal to 0, the FSM serial adder remains in state S_0. However, if the carryout changes to 1, the serial adder moves to state S_1. Once in state S_1, the serial adder returns to state S_0 if the carryout changes to 0; otherwise, the serial adder remains in state S_1. The input and output values of the serial adder cause transitions between states S_0 and S_1. The state table in Figure 9.35 summarizes the behavior of a serial adder. Figure 9.36 shows a state assigned table of the serial adder.

Notice that the FSM serial adder has two states, and therefore only one flip-flop is required to store the state memory of the FSM serial adder. The next state, c_{i+1}, and output s_i logic expressions are derived from the state assigned table as follows:

$$Y = c_{i+1} = a_ib_i + a_iy + b_iy = a_ib_i + a_ic_i + b_ic_i$$
$$s_i = a_i \oplus b_i \oplus y = a_i \oplus b_i \oplus c_i$$

The logic expressions are similar to those derived for the full-adder designed in Chapter 7. Using one D flip-flop, a logic circuit implementation of the serial adder is illustrated in Figure 9.37. The logic circuit of a 1-bit full-adder has been discussed in Section 7.4.

The state diagram of a Mealy model serial adder can be modified to that of a Moore model by appending the output values to the states of the FSM. Since the state memory and the output of a Mealy model serial adder consist of a 1-bit binary number, that is, the carryout c_{i+1} and the output s_i, a Moore model serial adder would have four states. The transitions between states S_0, S_1, S_2, and S_3 are caused by the changes in input bits a_i and b_i. Figure 9.38 illustrates the state diagram of the Moore model of a serial adder. Two-bit binary numbers are required to represent the states of the FSM serial adder.

Present State	Next State				Output			
	a_ib_i				a_ib_i			
	00	01	10	11	00	01	10	11
y	Y				s_i			
0	0	0	0	1	0	1	1	1
1	0	1	1	1	0	0	0	1

Figure 9.36 State Assigned Table of a Mealy Model Serial Adder

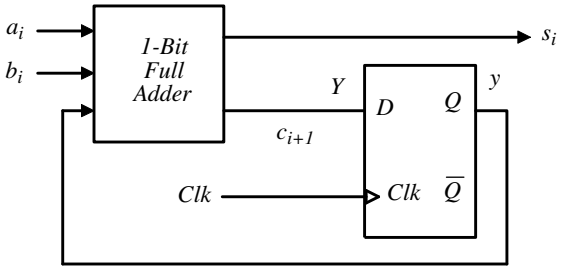


Figure 9.37 Logic Implementation of a Mealy Model Serial Adder

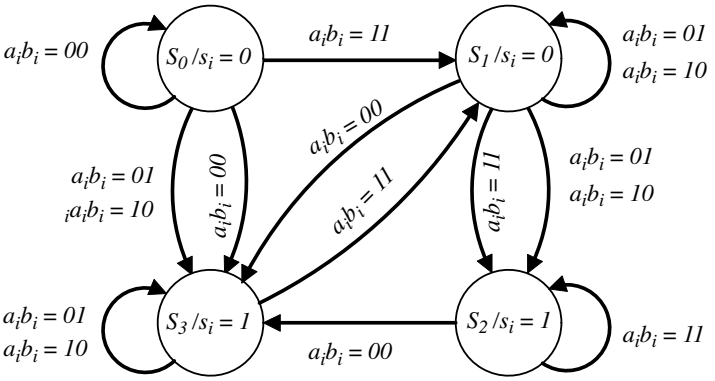


Figure 9.38 State Diagram of a Moore Model Serial Adder

The state table and state assigned table are illustrated in Figures 9.39 and 9.40, respectively.

The logic expressions of next states Y_2 and Y_1 and output s_i are derived from the state assigned table as follows:

$$Y_1 = a_i \oplus b_i \oplus y_2 = a_i \oplus b_i \oplus c_i$$
$$Y_2 = c_{i+1} = a_ib_i + a_iy_2 + b_iy_2 = a_ib_i + a_ic_i + b_ic_i$$
$$s_i = y_1$$

Present State	Next State a_ib_i				Output s_i
	00	01	10	11	
S_0	S_0	S_1	S_1	S_2	0
S_1	S_0	S_1	S_1	S_2	1
S_2	S_1	S_2	S_2	S_3	0
S_3	S_1	S_2	S_2	S_3	1

Figure 9.39 State Table of a Moore Model Serial Adder

Present State	Next State				Output
	a_ib_i				
	00	01	10	11	
y_2y_1	Y_2Y_1				s_i
00	00	01	01	10	0
01	00	01	01	10	1
10	01	10	10	11	0
11	01	10	10	11	1

Figure 9.40 State Assigned Table of a Moore Model Serial Adder

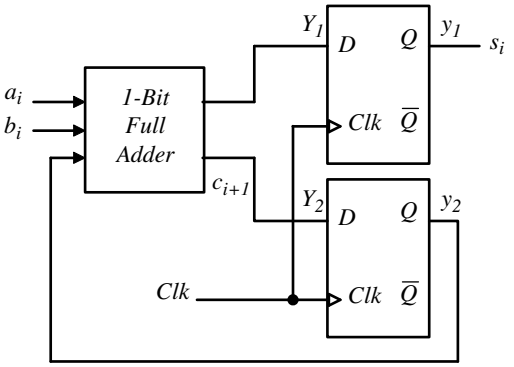


Figure 9.41 Logic Implementation of a Moore Model Serial Adder

Notice that next states Y_2 and Y_1 describe the sum and the carryout of a Moore model serial adder. Notice also that output s_i is equal to present state y_1, which is a delayed copy of next state Y_1. Figure 9.41 illustrates a circuit implementation of a Moore model serial adder using D flip-flops.

9.9 SEQUENTIAL CIRCUIT COUNTERS

A *sequential counter* is a logic circuit that generates a predetermined count sequence. In Chapter 8, we designed synchronous and asynchronous counters by cascading flip-flops. Often, these counters are under the control of a clock signal. A clock signal causes counting transitions. Moreover, these counters can produce only a limited number of count sequences. For counters that count in random sequences, cascading flip-flops to generate these sequences may not be a trivial task. The design becomes complicated when the user attempts to count asynchronous events without a clock signal.

To design synchronous counters, the general design procedures of sequential circuits may be used. These procedures allow the design of complex counters which can count in any counter order. To illustrate the design concepts of sequential

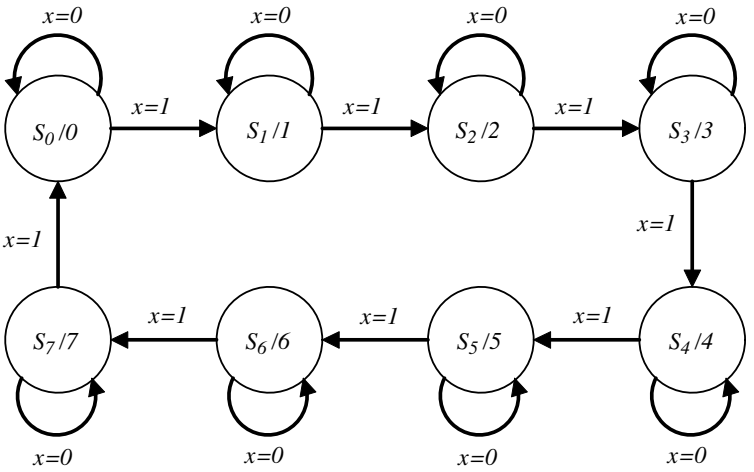


Figure 9.42 State Diagram of a 3-Bit Synchronous Up Counter

counters, we will design a 3-bit binary up counter which counts from 000 to 111. Input x will be used to cause the counting transitions. When x is equal to 1, the counter will increment; otherwise, the counter is idle at the last count. The example counter is designed to count events described by input signal x. The clock signal can also be used as an input signal as illustrated in the examples of Chapter 8.

Because input x does not cause a direct change in the output of the counter but, rather, its state memory, it is more appropriate to use the Moore model. The output of the counter is equal to the content of its state memory. Thus, each count is associated with a state. A 3-bit up counter will have eight states (number of counts), and the transitions between the states are controlled by input signal x and the clock signal. For example, if S_0 is the state associated with the count 000, the counter will move to S_1 (the state associated with the count 001) when input signal x is equal to 1; otherwise, the counter will remain in state S_0. The counter will move from state to state when the input x is equal to 1. After the last state, S_7, the counter will reset to state S_0 and start all over again. Figure 9.42 illustrates the states and transitions between states for a 3-bit synchronous up counter.

The state table shown in Figure 9.43 lists the states of a 3-bit up counter as described in the state diagram of Figure 9.42. Notice that as long as input x is equal to 0, the counter remains in the corresponding state. It is only after input x has changed to 1 that the counter increments. The 3-bit up counter has eight states. Therefore, 3-bit binary numbers are required to represent the states of the counter. Because the output of the counter is equal to the present state of the counter, it is convenient to assign each state to the corresponding output of the counter. States S_0 through S_7 will be assigned binary numbers 000 through 111. The output signals will be the same as the signals representing the state variable. No additional combinational logic is necessary for the outputs. The state assigned table is illustrated in Figure 9.44. The present states are designated y_3, y_2, and y_1 and the next states are designated Y_3, Y_2, and Y_1.

Present State	Next State		Output
	$x = 0$	$x = 1$	z
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_3	2
S_3	S_3	S_4	3
S_4	S_4	S_5	4
S_5	S_5	S_6	5
S_6	S_6	S_7	6
S_7	S_7	S_0	7

Figure 9.43 State Table of a 3-Bit Synchronous Up Counter

Once the state assigned table has been constructed, the next task is to determine the logic expressions of the next states and the output of the counter. Notice that we do not need to determine the logic expressions for the outputs since the outputs are equal to the present states of the counter. Using the Karnaugh maps shown in Figure 9.45, the optimized logic expressions for the next states are evaluated as follows:

$$Y_1 = \bar{x}y_1 + x\bar{y}_1 = x \oplus y_1$$
$$Y_2 = \bar{x}y_2 + y_2\bar{y}_1 + x\bar{y}_2y_1$$
$$Y_3 = \bar{x}y_3 + y_3\bar{y}_1 + y_3\bar{y}_2 + x\bar{y}_3y_2y_1$$

with the outputs of the counter expressed as follows:

$$z_1 = y_1$$
$$z_2 = y_2$$
$$z_3 = y_3$$

Present State	Next State		Output
	$x = 0$	$x = 1$	
$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	$z_3z_2z_1$
000	000	001	000
001	001	010	001
010	010	011	010
011	011	100	011
100	100	101	100
101	101	110	101
110	110	111	110
111	111	000	111

Figure 9.44 State Assigned Table of a 3-Bit Synchronous Up Counter

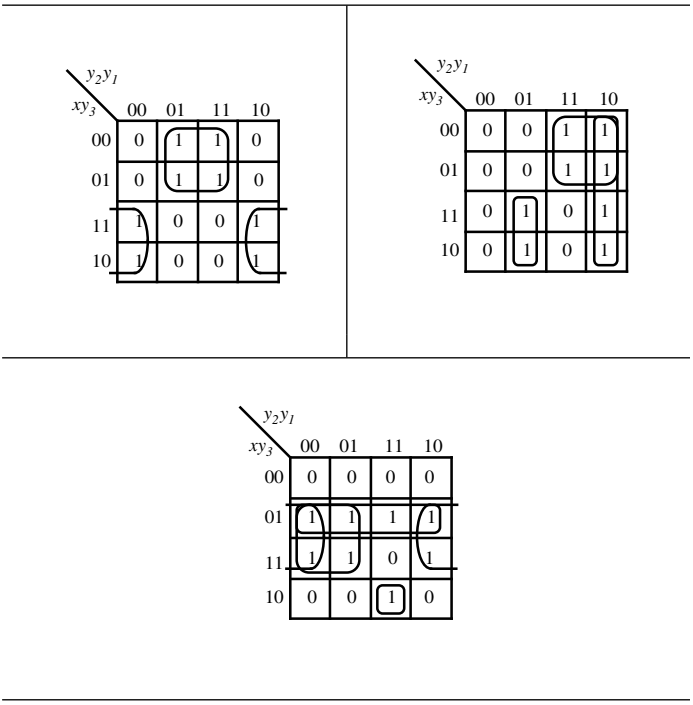


Figure 9.45 Next-State Logic Expressions for a 3-Bit Synchronous Up Counter

9.9.1 D Flip-Flop Implementation

To implement the 3-bit up counter, the final step of design is to choose the appropriate flip-flops. Of course, the simplest choice is to use D flip-flops. However, D flip-flops may not yield simple combinational logic, and other flip-flops must be considered as well. We will design the counter using D, JK, and T flip-flops.

Figure 9.46 shows a circuit implementation of the 3-bit counter using D flip-flops. Each next state is connected to the D input of the flip-flop, which provides the present state of the same variable. Notice that the combination logic requires a large number of logic gates. The complexity of the combinational logic will increase as the size of the counter increases. Figure 9.46 shows two-level logic design, but as the size of the counter increases, the fan-in and fan-out become serious problems that need to be addressed. When designing with discrete components, other flip-flops must be considered to minimize the combinational logic. Next, we attempt to reduce the combinational logic by using JK and T flip-flops.

9.9.2 JK Flip-Flop Implementation

JK flip-flops have been used widely in the past to implement digital counters. They can be configured (wired) to operate as D, SR, or T flip-flops. Recall that the next states

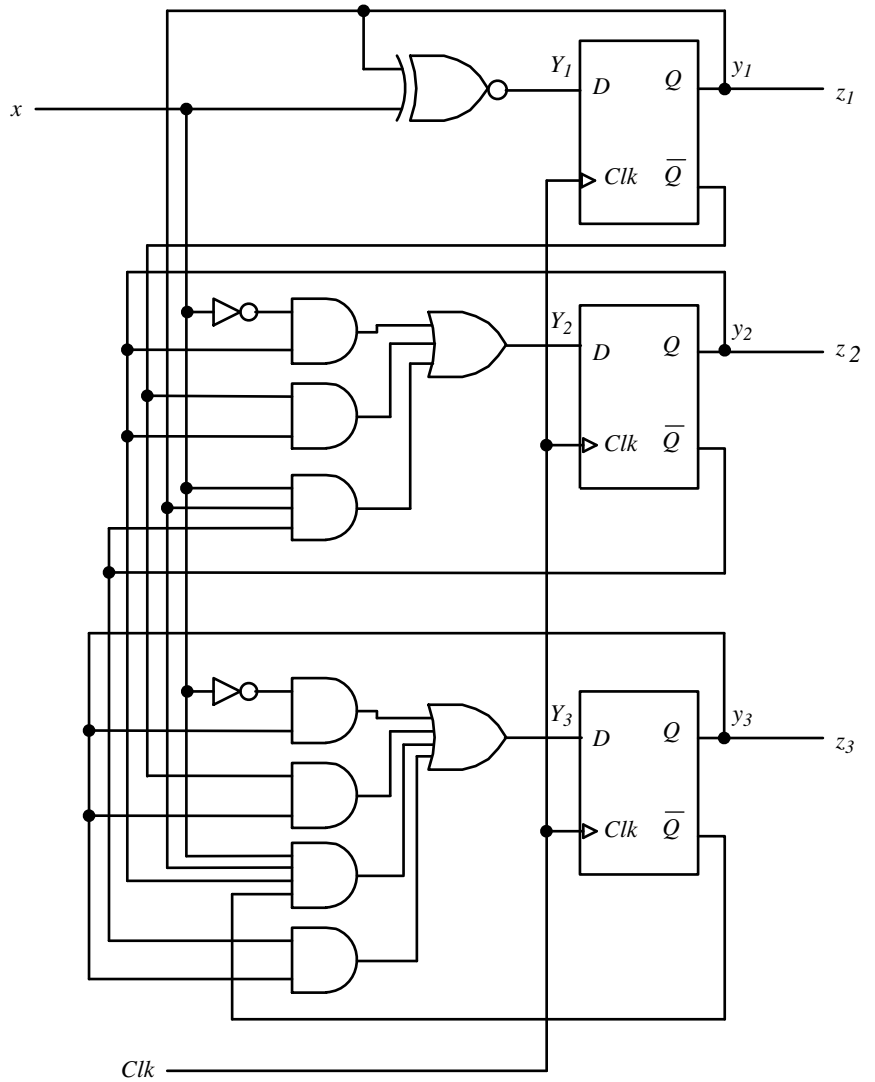


Figure 9.46 Logic Implementation of a 3-Bit Synchronous Up Counter Using D Flip-Flops

are the inputs of the flip-flops, and the present states are the outputs of the flip-flops. Therefore, to use JK flip-flops to implement sequential circuits, inputs J and K must be determined from the output states of the JK flip-flop. The input characteristics of the JK flip-flop are listed in Figure 9.47. Notice that for certain values of the present and next states, J and K may be undetermined. Using Figure 9.47, the new state assigned table of the counter is listed in Figure 9.48.

Using the Karnaugh maps shown in Figure 9.49, the optimized logic expressions for the next states described by J and K are evaluated and listed in the figure.

Present State	Next State	Input Conditions	
Q(t)	*Q(t+1)*	*J*	*K*
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

Figure 9.47 Input Characteristics of a JK Flip-Flop

Present State	Next State								Output
	$x = 0$				$x = 1$				
$y_3y_2y_1$	$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$Y_3Y_2Y_1$	J_3K_3	J_2K_2	J_1K_1	$z_3z_2z_1$
000	000	0d	0d	0d	001	0d	0d	1d	000
001	001	0d	0d	d0	010	0d	d1	d1	001
010	010	0d	d0	0d	011	0d	d0	1d	010
011	011	0d	d0	d0	100	1d	d1	d1	011
100	100	d0	0d	0d	101	d0	0d	1d	100
101	101	d0	0d	d0	110	d0	1d	d1	101
110	110	d0	d0	0d	111	d0	d0	1d	110
111	111	d0	d0	d0	000	d1	d1	d1	111

Figure 9.48 State Assigned Table of a 3-Bit Up Counter Using JK Flip-Flops

Figure 9.50 shows a circuit implementation of a 3-bit counter using JK flip-flops. Notice that the don't-care states in the new state assigned table in Figure 9.48 have reduced the combinational logic substantially. Therefore, the use of JK flip-flops would result in a simpler logic implementation of digital counters.

9.9.3 T Flip-Flop Implementation

A T flip-flop is a special operation of a JK flip-flop. For many counters, the state transitions occur when the state variables toggle in predictable patterns. Therefore, T flip-flops may be attractive alternatives. In fact, the digital counters described in Chapter 8 were designed primarily with T flip-flops. Similarly, to use T flip-flops to implement sequential circuits, the input T must be determined from the output states of the T flip-flop. The input characteristics of the T flip-flop are listed in Figure 9.51.

Because a T flip-flop is a JK flip-flop with the inputs wired together, there are no undetermined states as observed in a JK flip-flop. Using the input characteristics of the T flip-flop shown in Figure 9.51, the new state assigned table of the counter is

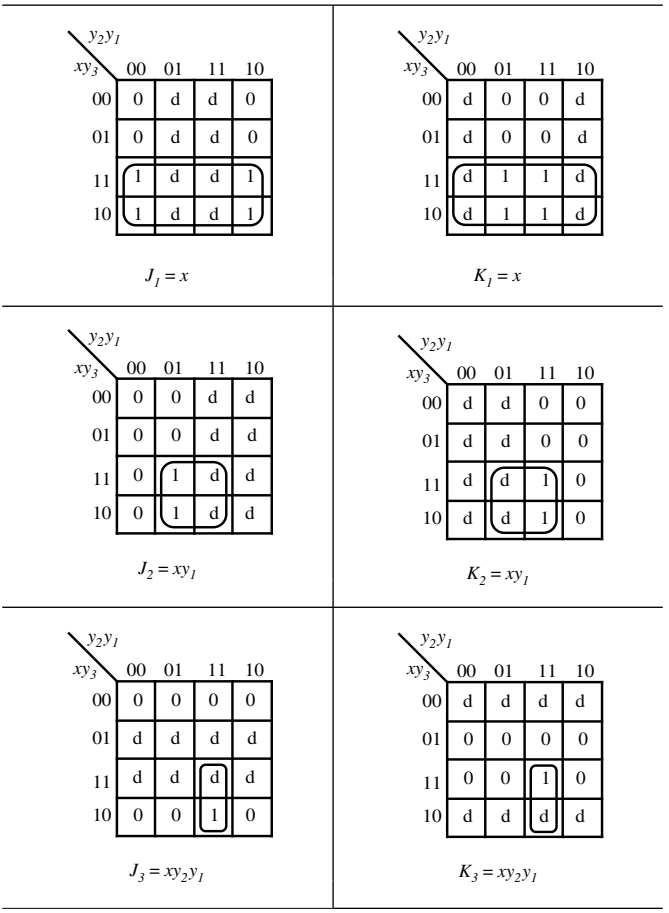


Figure 9.49 Next-State Logic Expressions for a 3-Bit Synchronous Up Counter Using JK Flip-Flops

given in Figure 9.52. From this table and using the Karnaugh maps shown in Figure 9.53, the optimized logic expressions for the next states are evaluated as follows:

$$T_1 = x$$
$$T_2 = xy_1$$
$$T_3 = xy_2y_1$$

Figure 9.54 shows a circuit implementation of the 3-bit counter using T flip-flops. Notice that the combinational logic is exactly similar to implementation with JK flip-flops. From a careful inspection of Figure 9.50 one would observe that inputs J and K of all the JK flip-flops are wired together to implement T flip-flops. One also could recognize the similarity between Figures 9.54 and 8.41.

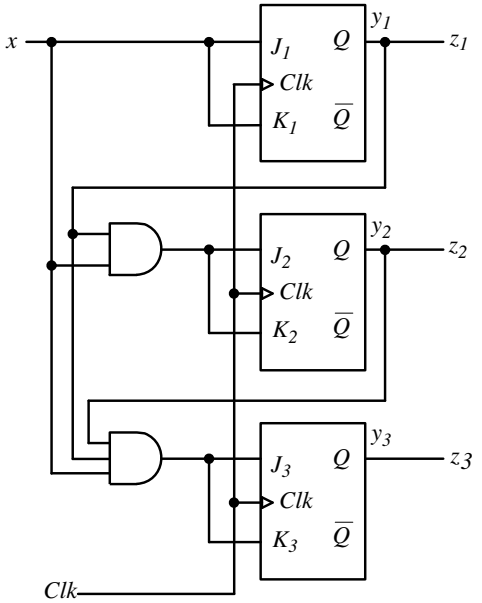


Figure 9.50 Logic Implementation of a 3-Bit Synchronous Up Counter Using JK Flip-Flops

Present State	Next State	Input Condition
Q(t)	*Q(t+1)*	*T*
0	0	0
0	1	1
1	0	1
1	1	0

Figure 9.51 Input Characteristics of the T Flip-Flop

9.10 STATE OPTIMIZATION

In the finite-state machine examples we discussed in previous sections that the number of states was very small. It was very easy to determine the minimum number of states possible that describe the behavior of the finite-state machine without introducing unnecessary (redundant) states. However, as the number of states increases, it becomes difficult to distinguish between possible states. The initial design attempt may include more states than are required by a finite-state machine. Some of the states are redundant, which increases the complexity of the finite-state machine unnecessarily.

Present State	Next State							Output
	$x = 0$				$x = 1$			
$y_3y_2y_1$	$Y_3Y_2Y_1$	T_3	T_2	T_1	$Y_3Y_2Y_1$	T_3	T_2	T_1	$z_3z_2z_1$
000	000	0	0	0	001	0	0	1	000
001	001	0	0	0	010	0	1	1	001
010	010	0	0	0	011	0	0	1	010
011	011	0	0	0	100	1	1	1	011
100	100	0	0	0	101	0	0	1	100
101	101	0	0	0	110	0	1	1	101
110	110	0	0	0	111	0	0	1	110
111	111	0	0	0	000	1	1	1	111

Figure 9.52 State Assigned Table of a 3-Bit Up Counter Using T Flip-Flops

State minimization is the process of finding and eliminating the redundant states, which are also referred to as *equivalent states*. Eliminating equivalent states will reduce the number of flip-flops and simplify the combinational logic of the finite-state machine. Two states are equivalent if and only if for all possible input sequences, the finite-state machine generates the same output regardless of whether it starts at one or the other state. It follows that the next states to two equivalent states must also be equivalent. Knowing all possible input and output sequences of a finite-state machine,

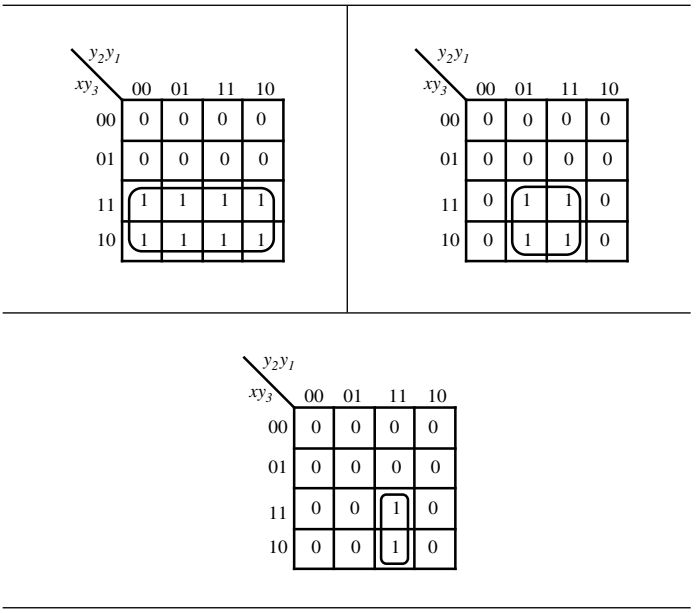


Figure 9.53 Next-State Logic Expressions for a 3-Bit Synchronous Up Counter Using T Flip-Flops

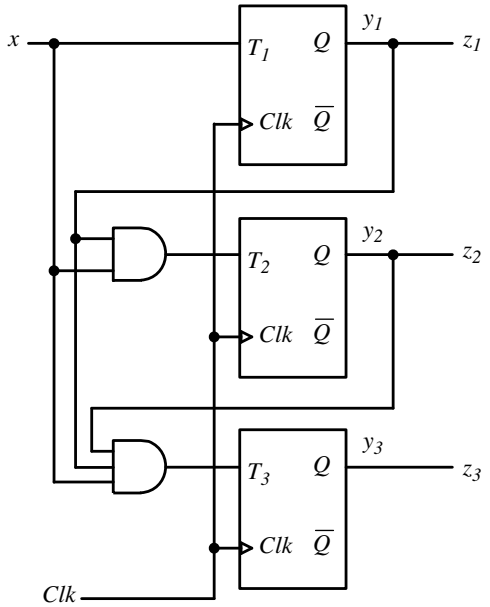


Figure 9.54 Logic Implementation of a 3-Bit Synchronous Up Counter Using JK Flip-Flops

one could automate the process by comparing two states at a time. Another less tedious method is to identify the states, which are not equivalent and study the remaining states closely to see whether they are equivalent. This method is referred to as the *partitioning minimization method.*

The partitioning minimization method will be illustrated in a finite-state machine that has one input and one output. The partitioning minimization procedure is an iterative method that partitions the states into blocks. Each block contains states that may be equivalent to each other but are not equivalent to the states in other blocks. At each iteration, the partitioning minimization method attempts to separate the states into blocks of nonequivalent state groupings. Therefore, the partitioning minimization method does not attempt to find states that are equivalent but, rather, finds states that are not equivalent. Two states are not equivalent:

1. If for the same input sequence, they generate a different output sequence
2. If their next states are not equivalent, that is, they belong to different blocks

We will use these two criteria to minimize the states of the finite-state machine described by the state table illustrated in Figure 9.55. Notice that the finite-state machine has one input x and one output z. There are eight states, of which some are equivalent. The initial partition P_0 contains one block, which consists of all the states. The states are assumed equivalent.

$$P_0 = (A, B, C, D, E, F, G, H)$$

Present State	Next State		Output z
	$x = 0$	$x = 1$	
A	A	B	0
B	C	F	0
C	C	B	0
D	A	G	0
E	E	H	1
F	D	E	0
G	E	G	1
H	G	H	1

Figure 9.55 State Table

In the next iteration, the new partition P_1 contains two blocks since there are two possible input values. Each block contains states that generate the same output. Clearly, two states that generate different outputs must not be equivalent. States A, B, C, D, and F generate an output equal to 0, whereas states E, G, and H generate an output equal to 1.

$$P_1 = (A, B, C, D, F)(E, G, H)$$

Partition P_1 implies that states A, B, C, D, and F are not equivalent to states E, G, and H. However, partition P_1 does not imply that states A, B, C, D, and F are equivalent, only that they may be equivalent. To test whether the states in a block remain in the same block, we need to find the next states for each state. If for each input values, the next states belong to the same block, the state remains in the current block. However, if the next states do not belong to the same block, the state is not equivalent to the other states in the same block. A new block is added to the new partition. The following expressions show the next states for each input value:

$$\text{For } x = 0, (A, B, C, D, F) \rightarrow (A, C, C, A, D) \text{ and } (E, G, H) \rightarrow (E, E, G)$$
$$\text{For } x = 1, (A, B, C, D, F) \rightarrow (B, F, B, G, E) \text{ and } (E, G, H) \rightarrow (H, G, H)$$

Notice that the next states of states D and F do not belong to the same blocks. Thus, states D and F are not equivalent to A, B, and C. Notice also that the next states of states E, G, and H belong to the same blocks. Thus, states E, G, and H may be equivalent. The new partition P_2 is constructed as follows:

$$P_2 = (A, B, C)(D)(F)(E, G, H)$$

We repeat the process of finding next states for the blocks that contain more than one state. The next states for blocks (A, B, C) and (E, G, H) are as follows:

Present State	Next State		Output
	$x = 0$	$x = 1$	z
A	A	B	0
B	A	F	0
D	A	E	0
E	E	E	1
F	D	E	0

Figure 9.56 Minimized State Table of Figure 9.55

$$\text{For } x = 0, (A, B, C) \rightarrow (A, C, C) \text{ and } (E, G, H) \rightarrow (E, E, G)$$
$$\text{For } x = 1, (A, B, C) \rightarrow (B, F, B) \text{ and } (E, G, H) \rightarrow (H, G, H)$$

Notice that the next states of state B do not belong to the same block. Thus, state B is not equivalent to A and C. Notice again that the next states of states E, G, and H belong to the same blocks. Thus, states E, G, and H may be equivalent. The new partition P_3 is constructed as follows:

$$P_3 = (A, C)(B)(D)(F)(E, G, H)$$

We repeat again the process of finding next states for blocks that contain more than one state. The next state for blocks (A, C) and (E, G, H) are as follows:

$$\text{For } x = 0, (A, C) \rightarrow (A, C) \text{ and } (E, G, H) \rightarrow (E, E, G)$$
$$\text{For } x = 1, (A, C) \rightarrow (B, B) \text{ and } (E, G, H) \rightarrow (H, G, H)$$

Notice now that the next states of states A and C belong to the same block. Thus, states A and C may be equivalent. Notice again that the next states of states E, G, and H belong to the same blocks. Thus, states E, G, and H may be equivalent. The new partition P_4 is constructed as follows:

$$P_4 = (A, C)(B)(D)(F)(E, G, H)$$

Since partition P_4 is exactly similar to P_3, the partitioning method stops. Because we cannot find any more states that are not equivalent, states that belong to the same block are indeed equivalent. Therefore, states A and C are equivalent and states E, G, and H are equivalent. From each block we select only one state to construct the minimized state table shown in Figure 9.56.

9.11 ASYNCHRONOUS SEQUENTIAL CIRCUITS

Finite-state machines also include another class of sequential circuits, known as *asynchronous sequential circuits*. Asynchronous sequential circuits do not require a

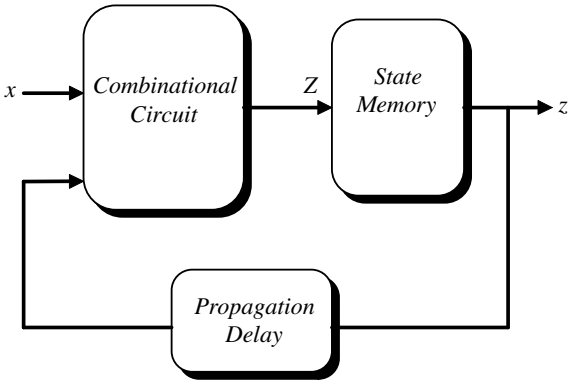


Figure 9.57 Structure of an Asynchronous Sequential Circuit

clock to function. The memory of the asynchronous sequential circuit may include flip-flops or time-delay devices. Whereas state transitions in a synchronous circuit are controlled by changes in the clock, asynchronous circuits depend on the time-delay propagation of the logic gates. The time-delay propagation, however, is not always consistent throughout the stages of the circuit. Thus, the feedback time delay may not be predictable. For this reason, asynchronous sequential circuits have limited applications. A typical application of asynchronous sequential circuits is where a circuit must respond to an input change promptly rather than waiting for a change in the clock. Consider the asynchronous sequential circuit in Figure 9.57. Its state is represented by the expression

$$Z = f(x, z)$$

where Z is the next state, z is the present state, and x is the input of the circuit. The propagation time delay consists of the feedback loop in the circuit.

For example, consider the case where the input changes from 0 to 1 and causes the next state to change from 0 to 1. While the change in the input is traveling through the gates, the present state and next state are temporarily equal to 0. After the time-delay propagation, the asynchronous sequential circuit eventually reaches a stable state. This temporary instability can lead to hazards. The key design of asynchronous sequential circuits is therefore to control the time-delay instability in the feedback loops. Time propagation delays can introduce hazards that can alter the function of the circuit temporarily. There are two types of hazards. A *static hazard* is a momentary change in a signal as it is transitioning to its state value when the input changes. A *dynamic hazard*, on the other hand, occurs when there is an imbalance in the propagation delays of intersecting paths. Therefore, the design of asynchronous sequential circuits must resolve all hazards to ensure proper functioning of the circuit. Consequently, asynchronous sequential circuits are difficult to design due to the inherent complications with propagation delays. In general, asynchronous circuits are faster than synchronous sequential circuits. Currently, a hybrid design between the synchronous and asynchronous models is used when faster circuits are warranted.

PROBLEMS

9.1 What is a finite-state machine?

9.2 List the procedural steps for finite-state machine design.

9.3 What is a Mealy machine?

9.4 What is a Moore machine, and how does it differ from a Mealy machine?

9.5 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.5.

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	00	01	0
01	00	10	0
10	00	10	1
11	00	10	1

Figure P9.5

9.6 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.6.

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	00	01	0
01	00	11	0
10	00	10	0
11	00	10	1

Figure P9.6

9.7 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.7.

9.8 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.8.

9.9 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.9.

Present State	Next State		Output
	$x = 0$	$x = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	z
00	01	10	1
01	10	01	0
10	11	00	0
11	00	11	1

Figure P9.7

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	00	01	0	0
01	00	10	0	0
10	00	10	0	1
11	00	10	1	1

Figure P9.8

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	00	01	0	0
01	00	11	0	0
10	00	10	0	0
11	00	10	0	1

Figure P9.9

9.10 Using D flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.10.

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	01	10	0	1
01	00	11	0	0
10	11	00	0	0
11	10	00	0	1

Figure P9.10

9.11 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.5.

9.12 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.6.

9.13 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.7.

9.14 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.8.

9.15 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.9.

9.16 Using JK flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.10.

9.17 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.5.

9.18 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.6.

9.19 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.7.

9.20 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.8.

9.21 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.9.

9.22 Using T flip-flops, design a logic circuit for the finite-state machine described by the state assigned table in Figure P9.10.

9.23 Determine the minimum states of the finite-state machine described by the state table in Figure P9.23.

Present State	Next State		Output
	$x = 0$	$x = 1$	z
A	A	B	1
B	C	F	0
C	C	B	0
D	A	G	0
E	E	H	0
F	D	E	0
G	E	G	1
H	G	H	1

Figure P9.23

9.24 Determine the minimum states of the finite-state machine described by the state table in Figure P9.24.

Present State	Next State		Output
	$x = 0$	$x = 1$	z
A	C	B	0
B	F	D	0
C	E	F	1
D	G	B	1
E	C	F	0
F	D	E	1
G	G	F	1
H	H	G	1

Figure P9.24

9.25 Determine the minimum states of the finite-state machine described by the state table in Figure P9.25.

Present State	Next State		Output
	$x = 0$	$x = 1$	z
A	C	D	1
B	H	F	1
C	D	E	0
D	E	A	1
E	A	C	0
F	B	F	0
G	H	B	1
H	G	C	0

Figure P9.25

9.26 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.5.

9.27 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.6.

9.28 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.7.

9.29 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.8.

9.30 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.9.

9.31 Write VHDL code to implement the finite-state machine described by the state assigned table in Figure P9.10.

9.32 Consider the finite-state machine logic implementation in Figure P9.32.

(a) Determine the next-state and output logic expressions.

(b) Determine the number of possible states.

(c) Construct a state assigned table.

(d) Construct a state table.

(e) Construct a state diagram.

(f) Determine the function of the finite-state machine.

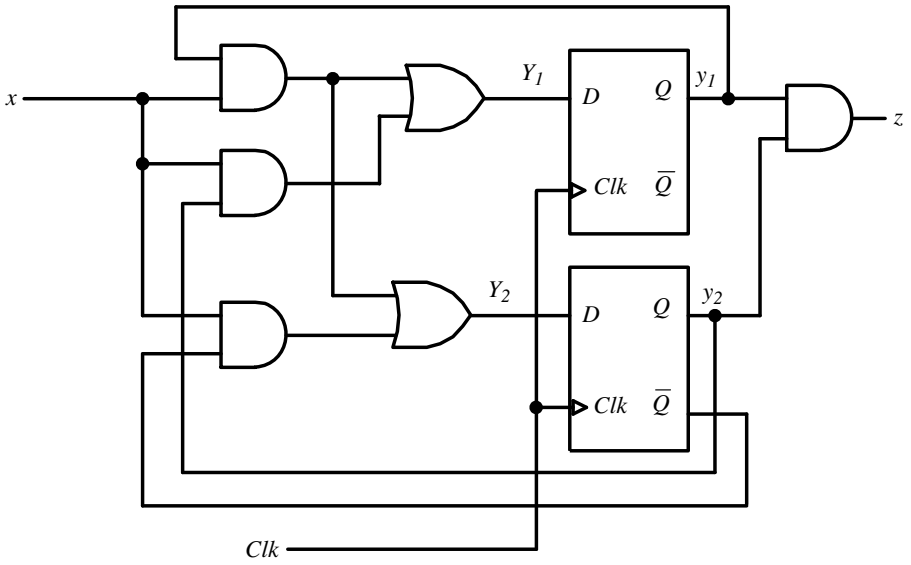


Figure P9.32

9.33 Consider the finite-state machine logic implementation in Figure P9.33.

(a) Determine the next-state and outputs logic expressions.

(b) Determine the number of possible states.

(c) Construct a state assigned table.

(d) Construct a state table.

(e) Construct a state diagram.

(f) Determine the function of the finite-state machine.

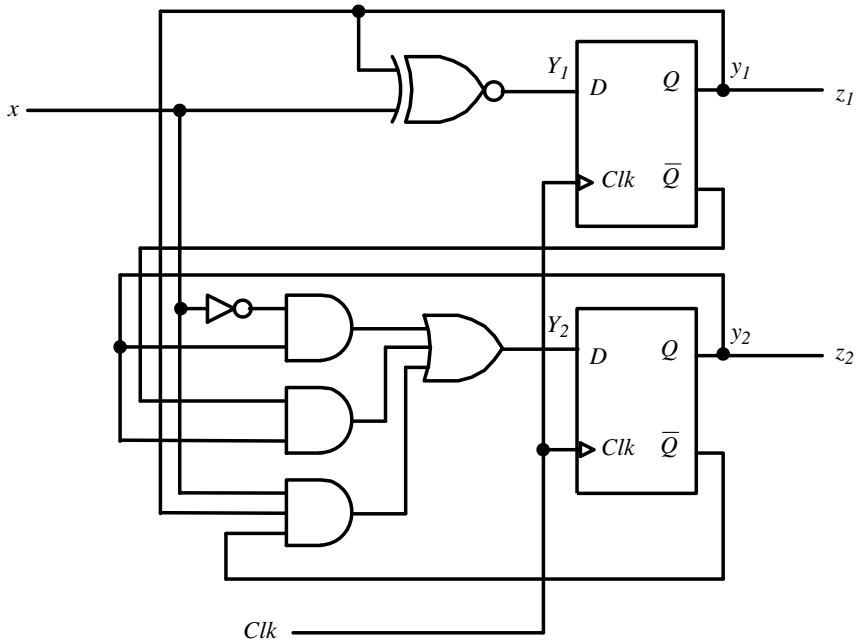


Figure P9.33

9.34 Consider the finite-state machine logic implementation in Figure P9.34.

(a) Determine the next-state and output logic expressions.
(b) Determine the number of possible states.
(c) Construct a state assigned table.
(d) Construct a state table.
(e) Construct a state diagram.
(f) Determine the function of the finite-state machine.

9.35 Consider the finite-state machine logic implementation in Figure P9.35.

(a) Determine the next-state and output logic expressions.
(b) Determine the number of possible states.
(c) Construct a state assigned table.
(d) Construct a state table.
(e) Construct a state diagram.
(f) Determine the function of the finite-state machine.

9.36 Consider the finite-state machine logic implementation in Figure P9.36.

(a) Determine the next-state and output logic expressions.
(b) Determine the number of possible states.

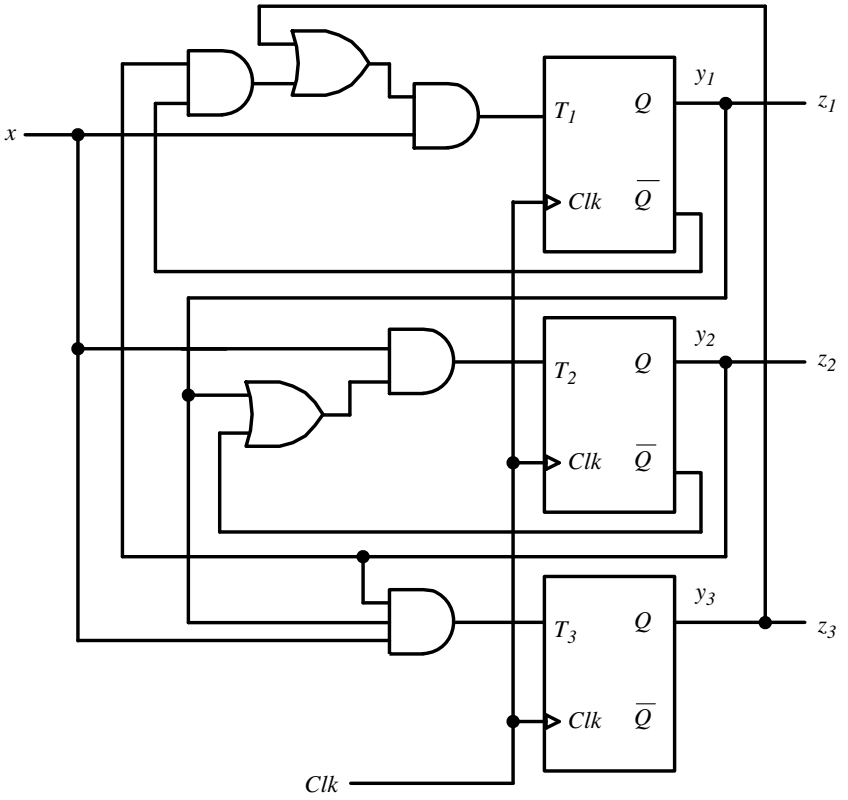


Figure P9.34

(c) Construct a state assigned table.

(d) Construct a state table.

(e) Construct a state diagram.

(f) Determine the function of the finite-state machine.

9.37 Design a logic circuit to implement a sequential parity checker. The parity bit is added to a group of 7 bits during transmission or storage. If the number of 1's in the 7-bit group is odd, the parity is odd. If the number of 1 in the 7-bit group is even, the parity is even.

9.38 Design a logic circuit to implement a Moore-type sequence detector to detect each of the following input sequences.

(a) 00

(b) 01

(c) 10

(d) 11

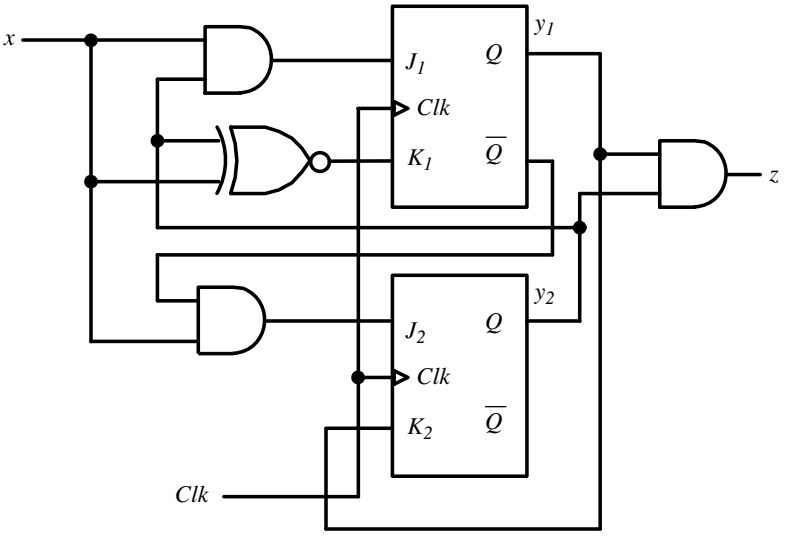


Figure P9.35

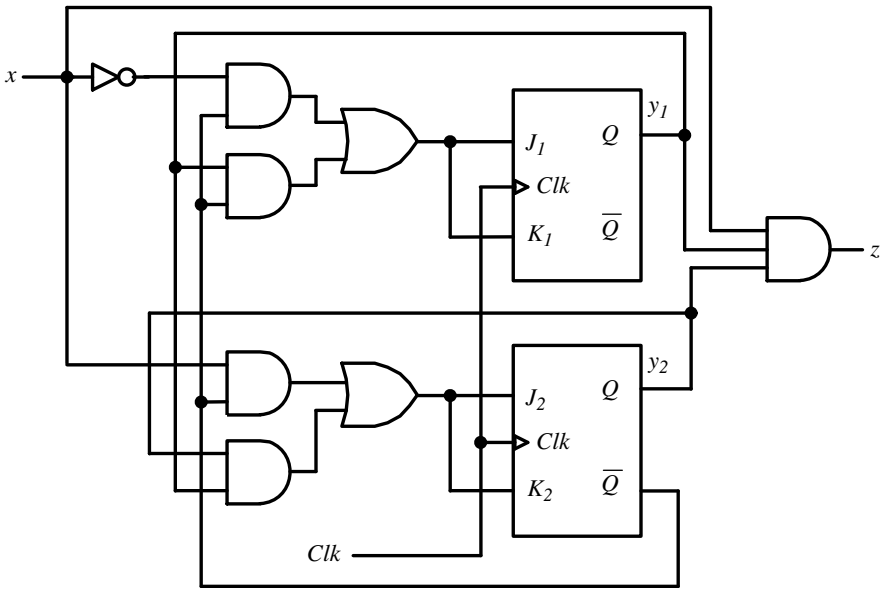


Figure P9.36

9.39 Design a logic circuit to implement a Mealy-type sequence detector to detect each of the input sequences of Problem 9.38.

9.40 Design a logic circuit to implement a Moore-type sequence detector to detect each of the following input sequences.

(a) 000 **(b)** 100
(c) 001 **(d)** 101
(e) 010 **(f)** 110
(g) 011 **(h)** 111

9.41 Design a logic circuit to implement a Mealy-type sequence detector to detect each of the input sequences of Problem 9.40.

9.42 Design a logic circuit to implement a Moore-type sequence detector to detect each of the following input sequences.

(a) 00 and 11

(b) 01 and 10

(c) 10 and 11

(d) 00 and 01

9.43 Design a logic circuit to implement a Mealy-type sequence detector to detect each of the input sequences of Problem 9.42.

9.44 Design a logic circuit to implement a Moore-type sequence detector to detect each of the following input sequences.

(a) 000 and 111 **(b)** 100 and 010
(c) 001 and 100 **(d)** 101 and 110
(e) 010 and 101 **(f)** 110 and 011
(g) 011 and 100 **(h)** 111 and 001

9.45 Design a logic circuit to implement a Mealy-type sequence detector to detect each of the input sequences of Problem 9.44.

9.46 Using D flip-flops, design a logic circuit to implement a JK flip-flop.

9.47 Using D flip-flops, design a logic circuit to implement a T flip-flop.

9.48 Using D flip-flops, design a synchronous counter that counts in the sequence 1, 3, 0, 2, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.49 Using D flip-flops, design a synchronous counter that counts in the sequence 0, 2, 4, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.50 Using D flip-flops, design a synchronous counter that counts in the sequence 1, 3, 5, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.51 Using D flip-flops, design a synchronous counter that counts in the sequence 0, 3, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.52 Using D flip-flops, design a synchronous counter that counts in the sequence 1, 4, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.53 Using D flip-flops, design a modulo-5 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.54 Using D flip-flops, design a modulo-6 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.55 Using D flip-flops, design a modulo-10 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.56 Using JK flip-flops, design a synchronous counter that counts in the sequence 1, 3, 0, 2, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.57 Using JK flip-flops, design a synchronous counter that counts in the sequence 0, 2, 4, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.58 Using JK flip-flops, design a synchronous counter that counts in the sequence 1, 3, 5, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.59 Using JK flip-flops, design a synchronous counter that counts in the sequence 0, 3, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.60 Using JK flip-flops, design a synchronous counter that counts in the sequence 1, 4, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.61 Using JK flip-flops, design a modulo-5 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.62 Using JK flip-flops, design a modulo-6 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.63 Using JK flip-flops, design a modulo-10 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.64 Using T flip-flops, design a synchronous counter that counts in the sequence 1, 3, 0, 2, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.65 Using T flip-flops, design a synchronous counter that counts in the sequence 0, 2, 4, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.66 Using T flip-flops, design a synchronous counter that counts in the sequence 1, 3, 5, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.67 Using T flip-flops, design a synchronous counter that counts in the sequence 0, 3, 6, 0, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.68 Using T flip-flops, design a synchronous counter that counts in the sequence 1, 4, 7, 1, ... The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.69 Using T flip-flops, design a modulo-5 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.70 Using T flip-flops, design a modulo-6 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

9.71 Using T flip-flops, design a modulo-10 synchronous counter. The counter counts only when its enable input x is equal to 1; otherwise, the counter is idle.

INDEX

Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL, First Edition. Mohammed Ferdjallah.