

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, Irwin

Table of Contents

7/31/2016

Preface

Chapter 0 Digital Systems and Design

0.0 Learning Objectives.....
0.1 Digital Systems
0.1.1 Digital vs Analog Systems.....
0.1.2 Technology Evolution.....
0.1.3 Design Evolution
0.2 Levels of Abstraction.....
0.2.1 System Level.....
0.2.2 Subsystem Level
0.2.3 Instruction Set Architecture (ISA)
0.2.4 Register Transfer Level (RTL)
0.2.5 Logic Level
0.2.6 Electronic Level.....
0.3 Digital System Implementation
0.3.1 Fixed Logic
0.3.2 Programmable Logic
0.3.3 Application Specific Integrated Circuits (ASIC)
0.3.4 Microprocessors
0.4 Applications.....
0.4.1 Computers.....
0.4.2 Controllers.....
0.4.3 Interfacing
0.4.4 Other.....
0.5 Summary and Review Questions
0.6 Collaboration Exercises
0.7 Problems

Chapter 1 Number Systems and Digital Codes

1.0 Learning Objectives.....	1
1.1 Number Systems.....	1
1.1.1 Positional and Polynomial Notations	2
1.1.2 Commonly Used Number Systems	3
1.2 Arithmetic	4
1.2.1 Binary Arithmetic.....	5
1.2.2 Hexadecimal Arithmetic	8
1.3 Base Conversion	10
1.3.1 Conversion Methods and Algorithms.....	10
1.3.2 Conversion When $B = A^k$	15
1.4 Signed Number Representation	16
1.4.1 Sign-Magnitude Numbers.....	16
1.4.2 Complementary Number Systems	17
1.5 Digital Codes.....	30
1.5.1 Numeric Codes.....	30
1.5.2 Character and Other Codes.....	36
1.5.3 Error Detection and Correction Codes.....	39
1.6 Summary and Review Questions	46
1.7 Collaboration Exercises	47
1.8 Problems	48

Chapter 2 Logic Circuits and Boolean Algebra

2.0 Learning Objectives	1
2.1 Logic Gates and Logic Circuits	1
2.1.1 Truth Tables	2
2.1.2 Basic Logic Gates	3
2.1.3 Combinational Logic Circuits.....	6
2.1.4 Sequential Logic Circuits	9
2.2 Hardware Description Languages	10
2.2.1 Verilog	11
2.2.2 VHDL	13
2.3 Boolean Algebra.....	15
2.3.1 Postulates and Fundamental Theorems	15
2.3.2 Boolean Functions and Logic Equations.....	23
2.3.3 Minterms, Maxterms, and Canonical Forms	24
2.3.4 Incompletely Specified Functions (Don't Cares).....	27
2.4 Minimization of Logic Expressions	29
2.4.1 Minimization Goals and Methods	29
2.4.2 Karnaugh Maps (K-Maps)	32
2.4.3 Minimization of Logic Expressions Using K-Maps	41
2.4.4 Quine-McCluskey Method	60
2.5 Summary and Review Questions	67
2.6 Collaboration Exercises	68
2.7 Problems.....	69

Chapter 3 Combinational Logic Circuit Design and Analysis

3.0 Learning Objectives	1
3.1 Design of Combinational Logic Circuits	1
3.1.1 AND-OR and NAND-NAND Circuits	2
3.1.2 OR-AND and NOR-NOR Circuits	3
3.1.3 Two-level Circuits.....	5
3.1.4 Multi-Level Circuits and Factoring	8
3.1.5 EXCLUSIVE OR (XOR) Circuits	11
3.2 Analysis of Combinational Circuits	14
3.2.1 Boolean Algebra	14
3.2.2 Truth Tables	17
3.2.3 Timing Diagrams	19
3.2.4 Positive and Negative Logic	25
3.3 Design Using Higher-Level Devices	26
3.3.1 Decoders	26
3.3.2 Encoders	39
3.3.3 Multiplexers and Demultiplexers	45
3.3.4 Arithmetic Circuits	55
3.4 Summative Design Examples	69
3.4.1 Design Flow	69
3.4.2 Bank Security-Lock Controller	70
3.4.3 Seven-Segment Display Decoder	74
3.4.4 Four-Function Arithmetic Logic Unit	81
3.4.5 Binary Array Multiplier	87
3.5 Summary and Review Questions	91
3.6 Collaboration Exercises	93
3.7 Problems	95

Chapter 4 Introduction to Sequential Circuits

4.0 Learning Objectives.....	1
4.1 Models and Classes of Sequential Circuits	2
4.1.1 Finite State Machines.....	2
4.1.2 State Diagrams and State Tables	4
4.1.3 Algorithmic State Machines	9
4.2 Memory Devices	12
4.2.1 Latches	14
4.2.2 Flip-flops.....	31
4.2.3 Latch and Flip-flop summary	45
4.3 Registers.....	47
4.4 Shift Registers.....	53
4.5 Counters	59
4.5.1 Synchronous Binary Counters	59
4.5.2 Asynchronous Binary Counters.....	63
4.5.3 Modulo-N Counters	65
4.5.4 Ring and Twisted-Ring Counters.....	71
4.6 Summative Design Examples.....	84
4.6.1 Register File Design	84
4.6.2 Multiphase Clock Design	86
4.6.3 Digital Timer Design	89
4.6.4 Programmable Baud Rate Generator.....	94
4.7 Summary and Review Questions	98
4.8 Collaboration Exercises	99
4.9 Problems	100

Chapter 5 Synchronous Sequential Circuit Analysis and Design

5.0 Learning Objectives.....	1
5.1 Analysis of Sequential Circuits.....	1
5.1.1 Circuit Analysis Using State Diagrams and State Tables.....	2
5.1.2 Analysis of Sequential Circuit Logic Diagrams	3
5.1.3 Summary	20
5.2 Design of Synchronous Sequential Circuits	21
5.2.1 Design Procedure.....	22
5.2.2 Flip-flop Input Tables	26
5.2.3 Design Examples	21
5.2.4 One-hot Finite-State Machine Design Method.....	56
5.3 State Reduction in Sequential Circuits	63
5.3.1 Redundant States	63
5.3.2 State Reduction in Completely Specified Circuits	65
5.4 Summative Design Examples.....	71
5.4.1 Drink Vending Machine Control Unit	71
5.4.2 Binary Multiplier.....	74
5.4.3 Traffic Light Controller	78
5.5 Summary and Review Questions	92
5.6 Collaboration Exercises	93
5.7 Problems	98

Chapter 6 Asynchronous Sequential Circuit Analysis and Design

6.0 Learning Objectives.....	1
6.1 Types of Asynchronous Circuits	1
6.2 Analysis and Design of Pulse-Mode Circuits.....	3
6.2.1 Analysis of Pulse-Mode Circuits.....	3
6.2.2 Design of Pulse-Mode Circuits.....	10
6.3 Analysis of Fundamental-Mode Circuits	17
6.3.1 Introduction	19
6.3.2 Excitation and Flow Tables	20
6.3.3 Analysis Procedure	22
6.4 Design of Fundamental-Mode Circuits.....	23
6.4.1 Design Procedure.....	24
6.4.2 Races and Cycles	35
6.4.3 Eliminating Races Conditions	40
6.4.4 Hazards	51
6.5 Summative Design Examples.....	53
6.5.1 Design Flow	53
6.5.2 Asynchronous Vending Machine Controller	53
6.5.3 Asynchronous Bus Arbiters	57
6.6 Summary and Review Questions	66
6.7 Collaboration Exercises	67
6.8 Problems	68

Chapter 7 Programmable Logic Devices (PLDs)

7.0 Learning Objectives.....
7.1 Programmable Digital Logic Device Technology
7.1.1 Taxonomy of Programmable Devices
7.1.2 Device Programming Methodologies.....
7.2 Field Programmable Gate Arrays (FPGAs).....
7.2.1 Lookup Tables (LUTs) for Combinational Logic Functions
7.2.2 FPGA Logic Cells.....
7.2.2.1 LUTs and Flip-flops for Sequential Logic Functions
7.2.2.2 Multiplexer-Based Logic Cell Options
7.2.3 FPGA Input/Output Blocks
7.2.4 FPGA Interconnect Resources.....
7.2.5 Mapping HDL Models onto FPGAs
7.2.6 Design Examples
7.3 Programmable Logic Devices (PLDs)
7.3.1 Array Structures for Combinational Logic Functions.....
7.3.1.1 Programmable AND/OR Circuits
7.3.1.2 Programmable Logic Array (PLA) Structure
7.3.1.3 Read Only Memory (ROM)
7.3.1.4 Programmable Array Logic (PAL)
7.3.2 Programmable Logic Device (PLD) Macrocells.....
7.3.3 Mapping HDL Models onto PLDs
7.3.4 Design Examples.....
7.4 Summative Design Examples.....
7.5 Summary and Review Questions
7.6 Collaboration Exercises
7.7 Problems

Chapter 8 Design of Digital Systems

8.0 Learning Objectives.....
8.1 Design Processes.....
8.1.1 Design flow.....
8.1.2 Tools.....
8.1.3 Standards
8.1.4 Constraints
8.1.5 Tradeoffs
8.1.6 Testing.....
8.2 Design Examples
8.2.1 Tiny RISC Processor.....
8.2.2 One-Lane Traffic Controller
8.2.3 UART
8.2.4 Elevator Controller and Display.....
8.2.5 Automotive Power Window Controller
8.2.6 Digital Alarm Clock.....
8.2.7 Binary Divider.....
8.3 Summary and Review Questions.....
8.4 Collaboration Exercises
8.5 Problems

Appendix A Circuit Technologies

A.1 Introduction
A.2 Basic Circuit Concepts
A.3 Bipolar Technology
A.3.1 Resistor-Transistor Logic (RTL)
A.3.2 Transistor-Transistor Logic (TTL)
A.3.3 Electrical Characteristics of TTL Circuits.....
A.4 Standard TTL Components
A.4.1 Small-Scale Integration (SSI)
A.4.2 Medium-Scale Integration (MSI)
A.5 Positive and Negative Logic
A.6 MOSFET Technology
A.6.1 PMOS
A.6.2 NMOS
A.7 Complementary MOS (CMOS) Technology
A.7.1 CMOS Gates
A.7.2 Electrical Characteristics of CMOS Circuits.....
A.7.3 Evolution of CMOS Technology.....
A.8 Standard CMOS Logic Components
A.8.1 4000-Series.....
A.8.2 7400-Series.....
A.8.3 Programmable Logic
A.9 Memory Technology
A.9.1 Static Random Access Memory (SRAM).....
A.9.2 Dynamic Random Access Memory (DRAM)
A.9.3 Synchronous DRAM (SDRAM)
A.9.4 Flash
A.10 Microprocessor Technology
A.11 References

Appendix B Verilog Primer

B.1 Introduction	1
B.2 General Concepts and Modules.....	2
B.2.1 Module Structure	3
B.2.2 Port Declarations	3
B.2.3 Data Types.....	3
B.2.4 Numbers	4
B.3 Gate-Level Structural Modeling.....	5
B.3.1 Gate Types	5
B.3.2 Gate Delay.....	6
B.4 Dataflow Modeling	6
B.4.1 Expressions, Operands, and Operators.....	6
B.4.2 Continuous Assignment Statements.....	6
B.4.3 Continuous Assignment Statement Delay.....	7
B.5 Behavioral Modeling.....	10
B.5.1 Procedural Blocks	10
B.5.2 Procedural Assignments	11
B.5.3 Timing Control.....	12
B.5.4 case Statements	14
B.5.5 if...else Statements	16
B.5.6 Loop Statements.....	17
B.5.7 Block Execution	19
B.6 Hierarchical Modeling.....	20
B.6.1 Functions and Tasks.....	20
B.6.2 Structural Models	22
B.7 System Tasks and Compiler Directives.....	24
B.8 Test Benches	25
B.9 Summary of Features and Keywords	26
B.10 References.....	26

Appendix C VHDL Primer

C.1 Introduction
C.2 Entities
C.2.1 Port Declarations
C.2.2 Scalar and Vector Signal Data Types
C.2.4 Generic Parameters
C.3 Dataflow Modeling
C.3.1 Signal Assignment Statements
C.3.2 Expressions, Operands, and Operators
C.3.3 Conditional Signal Assignment Statements.....
C.3.4 Signal Assignment Delay
C.3.5 Libraries and Packages
C.3.6 IEEE Standard 1164 Data Types
C.4 Component-Level Structural Models
C.4.1 Component Declarations
C.4.2 Component Instantiation
C.4.3 Gate-Level Models
C.5 Modeling Sequential Behavior
C.5.1 Process Construct
C.5.2 Process Sensitivity List
C.5.3 Synchronous and Asynchronous Control Signals
C.5.4 Sequential Statement Constructs
C.5.5 Wait Statements.....
C.6 Hierarchical System Modeling.....
C.6.1 Functions and Procedures
C.6.2 Register Transfer Level (RTL) Design
C.6.3 System Level Design.....
C.7 Test Benches.....
C.8 Summary of Features and Keywords
C.9 References.....

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, and Irwin

Chapter 1 Number Systems and Digital Codes

(7/31/2016)

Table of Contents

1.0 Learning Objectives.....	1
1.1 Number Systems.....	1
1.1.1 Positional and Polynomial Notations	2
1.1.2 Commonly Used Number Systems	3
1.2 Arithmetic	4
1.2.1 Binary Arithmetic	5
1.2.2 Hexadecimal Arithmetic	8
1.3 Base Conversion	10
1.3.1 Conversion Methods and Algorithms	10
1.3.2 Conversion When $B = A^k$	15
1.4 Signed Number Representation	16
1.4.1 Sign-Magnitude Numbers	16
1.4.2 Complementary Number Systems.....	17
1.5 Digital Codes.....	30
1.5.1 Numeric Codes	30
1.5.2 Character and Other Codes.....	36
1.5.3 Error Detection and Correction Codes.....	39
1.6 Summary and Review Questions	46
1.7 Collaboration Exercises	47
1.8 Problems	48

DRAFT

This page intentionally left blank.

Chapter 1 Number Systems and Digital Codes

(6/8/2016)

Computers and other digital systems process information as their primary function. Therefore, it is necessary to have methods and systems for representing information in forms that can be manipulated and stored using electronic or other types of hardware.

In this chapter we cover number systems and codes that are often employed in computers and digital systems. Topics covered include binary, octal, and hexadecimal number systems and arithmetic; base conversion techniques; negative number representation methods such as sign-magnitude, two's complement, and one's complement; numeric codes for fixed- and floating-point numbers; character codes, including binary coded decimal and ASCII; Gray and excess codes; and error detection and correction codes. Later chapters of the book cover the analysis and design of hardware to process information represented in the forms described here.

1.0 Learning Objectives

Students successfully completing this chapter will have demonstrated the knowledge and/or skills to accomplish the following.

1. Understand how to read and write numbers in base two, eight, ten, and sixteen and be able to convert numbers in one base to another.
2. Be able to perform addition, subtraction, multiplication, and division in base two, ten, and sixteen.
3. Understand how to represent positive and negative binary numbers in sign-magnitude, two's complement, and one's complement forms and be able to perform addition and subtraction in a two's complement number system.
4. Understand and be able to use binary-coded-decimal (BCD) and American Standard Code for Information Interchange (ASCII) codes for representing decimal digits and alphanumeric characters.
5. Understand error detection coding and simple error detection codes such as parity codes and two-out-of-five codes.
6. Understand the basic concepts of Hamming codes.

1.1 Number Systems

A number system consists of an ordered set of symbols, called *digits*, with relations defined for addition (+), subtraction (-), multiplication (x), and division (\div). The *radix* (r), or *base*, of the number system is the total number of digits allowed in the number system. Number systems commonly used in computers and other digital systems include *decimal*

($r = 10$), *binary* ($r = 2$), *octal* ($r = 8$), and *hexadecimal* ($r = 16$). Any number in a given system may have both an integer part and a fractional part, which are separated by a radix point (.). The integer part or the fraction part may be absent in some cases. Now let's examine *positional* and *polynomial* notations for representing numbers.

1.1.1 Positional and Polynomial Notations

Suppose you borrow one hundred twenty-three dollars and thirty-five cents from your local bank. The check you are given indicates the amount as \$123.35. In writing this number, positional notation has been used. The check may be cashed for 1 one hundred dollar bill, 2 ten dollar bills, 3 one dollar bills, 3 dimes, and 5 pennies. Therefore, the position of each digit indicates its relative weight or significance.

Positional Notation

In general, a positive number N can be written in positional form as

$$N = (a_{n-1}a_{n-2}\dots a_1a_0 \cdot a_{-1}a_{-2}\dots a_{-m})_r \quad (1.1)$$

where

\cdot = radix point separating the integer and fractional digits

r = radix or base of the number system being used

n = number of integer digits to the left of the radix point

m = number of fractional digits to the right of the radix point

a_i = integer digit i when $n - 1 \geq i \geq 0$

a_i = fractional digit i when $-1 \geq i \geq -m$

a_{n-1} = most significant digit

a_{-m} = least significant digit

Note that the range of values for all digits a_i is $r - 1 \geq a_i \geq 0$. Using this notation, the bank loan amount would be written $\$(123.35)_{10}$. The parentheses and the subscript denoting the radix may be eliminated if the radix is either known by the context or otherwise specified.

Polynomial Notation

The $(123.35)_{10}$ dollar loan amount can be written in polynomial or series form as

$$\begin{aligned} N &= 1 \times 100 + 2 \times 10 + 3 \times 1 + 3 \times 0.1 + 5 \times 0.01 \\ &= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 3 \times 10^{-1} + 5 \times 10^{-2} \end{aligned}$$

Note that each digit resides in a weighted position and that the weight of each position is a power of the radix 10. In general, any number N of radix r may be written as a polynomial in the form

$$N = \sum_{i=-m}^{n-1} a_i r^i \quad (1.2)$$

where each symbol is defined the same as in Eq. 1.1. For the bank loan, $r = 10$, $a_2 = 1$, $a_1 = 2$, $a_0 = 3$, $a_{-1} = 3$, $a_{-2} = 5$, and $a_i = 0$ for $i \geq 3$ and for $i \leq -3$.

1.1.2 Commonly Used Number Systems

The decimal, binary, octal, and hexadecimal number systems are all important for the study of digital systems. Table 1.1 summarizes the fundamental features of each system and illustrates a limited range of positive integers in each. All the numbers in Table 1.1 are written in positional notation.

TABLE 1.1 IMPORTANT NUMBER SYSTEMS

Name	Decimal	Binary	Octal	Hexadecimal
Radix	10	2	8	16
Digits	0,1,2,3,4, 5,6,7,8,9	0,1	0,1,2,3, 4,5,6,7	0,1,2,3,4,5, 6,7,8,9,A,B, C,D,E,F
First seventeen positive integers	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111 10000	0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 20	0 1 2 3 4 5 6 7 8 9 A B C D E F 10

Digital systems are usually constructed using two-state devices that are either in an *off* state or an *on* state. Hence, the binary number system is ideally suited for representing numbers in digital systems, since only two digits, 0 and 1, commonly called *bits*, are needed. A bit can be stored in a two-state storage device often called a *latch*. Binary numbers of length *n* can be stored in an *n*-bit long device known as a *register*, which is built with *n* latches. An 8-bit register loaded with the binary number 10011010 is shown in Fig. 1.1.

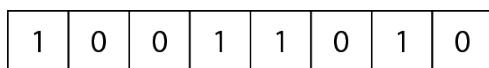


Figure 1.1 An 8-bit register.

1.2 Arithmetic

Every child learns the rudiments of arithmetic by memorizing the base-10 addition and multiplication tables as shown in Tables 1.2a and 1.2b, respectively. Subtraction can be accomplished by using the addition table in reverse. Similarly, long division uses trial and error multiplication and subtraction to obtain the quotient. The foundation for arithmetic in any base is a knowledge of its addition and multiplication tables. Given these tables, arithmetic operations proceed in a similar manner for all bases. Arithmetic in the binary and hexadecimal number systems will be introduced in the remainder of this section.

TABLE 1.2 DECIMAL ARITHMETIC TABLES

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

(a) Addition

x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

(b) Multiplication

1.2.1 Binary Arithmetic

Addition

Tables 1.3a and 1.3b show the addition and multiplication tables, respectively, for the binary number system. The tables are very small since there are only two digits, or *bits*, in the system. Binary arithmetic is very simple as a result. Note that the addition 1+1 produces a sum bit of 0 and a carry bit of 1. The carry must be added to the next column of bits as addition proceeds in the normal pattern from right to left. Two examples of binary addition are given next.

TABLE 1.3 BINARY ARITHMETIC TABLES

+	0	1
0	0	1
1	1	10

(a) Addition

x	0	1
0	0	0
1	0	1

(b) Multiplication

Add the two binary numbers $(111101)_2$ **and** $(10111)_2$.

	1	1	1	1	1	1	Carries
	1	1	1	1	0	1	Augend
+			1	0	1	1	Addend
	1	0	1	0	0	0	Sum

In the above example, two columns were encountered that had two 1 bits and a carry bit of 1, which had to be totaled. This addition of three 1's can be more easily viewed as

$$\begin{aligned}
 1+1+1 &= (1+1)+1 \\
 &= (10)_2 + (01)_2 \\
 &= 11
 \end{aligned}$$

Thus both the sum bit and the carry bit are 1's.

When a long list of binary numbers must be added, the computation is easily performed by adding the numbers in pairs, as demonstrated in the following example.

Add the four numbers $(101101)_2$, $(110101)_2$, $(001101)_2$, **and** $(010001)_2$.

However, we may choose to perform the addition directly to avoid the intermediate steps required by the preceding approach. The direct method is illustrated in the second example below.

$$\begin{array}{r}
 \left. \begin{array}{r}
 1 & 0 & 1 & 1 & 0 & 1 \\
 1 & 1 & 0 & 1 & 0 & 1 \\
 0 & 0 & 1 & 1 & 0 & 1 \\
 \hline
 + & 0 & 1 & 0 & 0 & 0 & 1
 \end{array} \right\} \\
 \left. \begin{array}{r}
 1 & 1 & 1 & 1 & 1 & 1 \\
 1 & 0 & 1 & 1 & 0 & 1 \\
 1 & 1 & 0 & 1 & 0 & 1 \\
 \hline
 1 & 1 & 0 & 0 & 1 & 0
 \end{array} \right\} \\
 \left. \begin{array}{r}
 1 & 1 & 1 & 1 & 1 & 1 \\
 1 & 1 & 0 & 0 & 0 & 1 \\
 0 & 1 & 1 & 1 & 1 & 0 \\
 \hline
 1 & 0 & 0 & 0 & 0 & 0
 \end{array} \right\}
 \end{array}$$

Repeat the previous example by adding one entire column at a time.

10	10	10	10	1	10	Carries
1	0	1	1	0	1	
1	1	0	1	0	1	
0	0	1	1	0	1	
+	0	1	0	0	0	
1	0	0	0	0	0	Sum

Note that the sum of the digits in the first column is $1 + 1 + 1 + 1 = (100)_2$. This results in a sum digit of 0 in that column and a carry of 10 to the next column.

Subtraction

Subtraction may be thought of as the inverse of addition. The rules for binary subtraction follow directly from the binary addition table in Table 1.3a and are

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$0 - 0 = 0$$

$$0 - 1 = 1 \text{ with a borrow of 1, or } 10 - 1 = 1$$

The last rule shows that if a 1 bit is subtracted from a 0 bit then a 1 must be borrowed from the next most significant column. Borrows propagate to the left from column to column, as illustrated next.

Subtract $(10111)_2$ **from** $(1001101)_2$.

6	5	4	3	2	1	0	Column
				10			Borrows
0	10	10	0	0	10		Borrows
1	0	0	1	1	0	1	Minuend
-				1	0	1	Subtrahend
	1	1	0	1	1	0	Difference

In this example, a borrow is first encountered in column 1. The borrow is taken from column 2, resulting in a 10 in column 1 and a 0 in column 2. The 0 now present in column 2 necessitates a borrow from column 3. No other borrows are necessary until column 4. In this case, there is no 1 in column 5 to borrow. Hence, we must first borrow the 1 from column 6, which results in 0 in column 6 and 10 in column 5. Now column 4 borrows a 1 from column 5, leaving 1 in column 5 ($10 - 1 = 9$) and 10 in column 4. This sequence of borrows is shown above the minuend terms.

Multiplication and Division

Binary multiplication is performed in a similar fashion as decimal multiplication except that binary multiplication operations are much simpler, as can be seen in Table 1.3b. Care must be taken, however, when adding the partial products, as illustrated in the following example.

Multiply $(10111)_2$ **by** $(1010)_2$.

	1	0	1	1	1	Multiplicand		
×		1	0	1	0	Multiplier		
	0	0	0	0	0			
	1	0	1	1	1			
	0	0	0	0	0			
1	0	1	1	1				
1	1	1	0	0	1	1	0	Product

Note that there is one partial product for every multiplier bit. This procedure can be performed more efficiently by merely shifting one column to the left, rather than listing an all-zero partial product for a multiplier bit of 0. We can see from this example how easily binary multiplication can be accomplished.

Binary division is performed using the same trial and error procedure as decimal division. However, binary division is easier since there are only two values to try. Copies of the divisor terms are subtracted from the dividend, yielding positive intermediate remainder terms. The following example illustrates binary division.

Divide $(1110111)_2$ **by** $(1001)_2$.

					1	1	0	1	Quotient
Divisor	1	0	0	1	1	1	1	1	Dividend
					1	0	0	1	
						1	0	1	
						1	0	1	
							1	0	
							1	0	Remainder

1.2.2 Hexadecimal Arithmetic

Hexadecimal addition and multiplication tables are more complex than those for decimal and binary number systems and are given in Table 1.4. However, as with other number systems, a knowledge of these tables permits hexadecimal arithmetic to be performed using well-known procedures. The following four examples illustrate hexadecimal arithmetic.

Compute $(2A58)_{16} + (71D0)_{16}$.

$$\begin{array}{r}
 & & 1 & & \\
 & 2 & A & 5 & 8 \\
 + & 7 & 1 & D & 0 \\
 \hline
 9 & C & 2 & 8
 \end{array}
 \begin{array}{l}
 \text{Carries} \\
 \text{Augend} \\
 \text{Addend} \\
 \text{Sum}
 \end{array}$$

Compute $(9F1B)_{16} - (4A36)_{16}$.

$$\begin{array}{r}
 & E & 11 & & \\
 & F & 1 & B & \\
 - & 4 & A & 3 & 6 \\
 \hline
 5 & 4 & E & 5
 \end{array}
 \begin{array}{l}
 \text{Borrows} \\
 \text{Minuend} \\
 \text{Subtrahend} \\
 \text{Difference}
 \end{array}$$

Compute $(5C2A)_{16} \times (71D0)_{16}$.

$$\begin{array}{r}
 & 5 & C & 2 & A \\
 & \times & 7 & 1 & D & 0 \\
 4 & A & E & 2 & 2 & 0 \\
 5 & C & 2 & A \\
 \hline
 2 & 8 & 5 & 2 & 6 & \\
 & & F & 9 & 6 & C & 2 & 0
 \end{array}
 \begin{array}{l}
 \text{Multiplicand} \\
 \text{Multiplier} \\
 \text{Partial products} \\
 \text{Product}
 \end{array}$$

Compute $(27FCA)_{16} \div (3E)_{16}$.

$$\begin{array}{r}
 \begin{array}{c}
 & A & 5 & 1 \\
 & C & & A \\
 \hline
 1 & 3 & C \\
 1 & 3 & 6 \\
 \hline
 6 & A \\
 3 & E \\
 \hline
 2 & C
 \end{array}
 \begin{array}{l}
 \text{Quotient} \\
 \text{Dividend}
 \end{array}
 \\
 \text{Divisor} \quad 3 \quad E \quad \boxed{\begin{array}{cccc} 2 & 7 & F & C \\ 2 & 6 & C & \end{array}}
 \end{array}
 \begin{array}{l}
 \text{Remainder}
 \end{array}$$

TABLE 1.4 HEXADECIMAL ARITHMETIC TABLES

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

(a) Addition

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

(b) Multiplication

1.2 Base Conversions

Users and designers of computers and other digital systems often encounter a need to convert a given number in base A to the equivalent number in base B. Algorithms for performing base conversions will be presented and illustrated in this section.

1.3.1 Conversion Methods and Algorithms

Series Substitution

The polynomial representation of a number previously given by Eq. 1.2 forms the basis of the *series substitution* conversion method. The equation can be written in an expanded form as follows:

$$N = a_{n-1}r^{n-1} + \dots + a_0r^0 + a_{-1}r^{-1} + \dots + a_{-m}r^{-m} \quad (1.3)$$

A number in base A can be converted to a number in base B in two steps.

1. Form the series representation of the number in base A in the format of Eq. 1.3.
2. Evaluate the series using base B arithmetic.

The following four examples illustrate this procedure.

Convert $(10100)_2$ to base 10. We make this conversion by substituting for each digit, according to its weight. Counting from right to left in $(10100)_2$, we find that the rightmost digit, 0, has a weight of 2^0 , the next digit, 0, has weight 2^1 , and so on. Substituting these values into Eq. 1.3 and evaluating the series with base 10 arithmetic gives:

$$\begin{aligned} N &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= (16)_{10} + 0 + (4)_{10} + 0 + 0 \\ &= (20)_{10} \end{aligned}$$

Convert $(274)_8$ to base 10.

$$\begin{aligned} N &= 2 \times 8^2 + 7 \times 8^1 + 4 \times 8^0 \\ &= (128)_{10} + (56)_{10} + (4)_{10} \\ &= (188)_{10} \end{aligned}$$

Convert $(1101.011)_2$ to base 8.

The integer part of the number is converted as in the previous examples. With digits that are to the right of a binary point, we count from left to right. The first digit to the right of the binary point, 0, has weight 2^{-1} , the next digit, 1, has weight 2^{-2} , and the third digit, 1, has weight 2^{-3} . Substituting into Eq. 1.3 gives

$$\begin{aligned}
 N &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\
 &= (10)_8 + (4)_8 + 0 + (1)_8 + 0 + (.2)_8 + (.1)_8 \\
 &= (15.3)_8
 \end{aligned}$$

Convert $(AF3.15)_{16}$ to base 10.

$$\begin{aligned}
 N &= A \times 16^2 + F \times 16^1 + 3 \times 16^0 + 1 \times 16^{-1} + 5 \times 16^{-2} \\
 &= 10_{10} \times 256_{10} + 15_{10} \times 16_{10} + 3_{10} \times 1_{10} \\
 &\quad + 1_{10} \times 0.0625_{10} + 5_{10} \times 0.00390625_{10} \\
 &= 2560_{10} + 240_{10} + 3_{10} + 0.0625_{10} + 0.01953125_{10} \\
 &= (2803.08203125)_{10}
 \end{aligned}$$

Note in the preceding examples that the computations were easier for conversions from base A to base B when $A < B$. Conversion methods will now be described where the converse is true.

Radix Divide Method

The *radix divide* conversion method can be used for converting an integer in base A to the equivalent base B integer. To understand the method, consider the following representation of integer N_I .

$$(N_I)_A = b_{n-1}B^{n-1} + \dots + b_0B^0 \quad (1.4)$$

In Eq. 1.4, the b_i 's represent the digits of $(N_I)_B$ in base A. The least significant digit, $(b_0)_A$, can be found by dividing $(N_I)_A$ by $(B)_A$ as follows.

$$\begin{aligned}
 N_I/B &= (b_{n-1}B^{n-1} + \dots + b_1B^1 + b_0B^0)/B \\
 &= \underbrace{b_{n-1}B^{n-2} + \dots + b_1B^0}_{\text{Quotient, } Q_1} + \underbrace{b_0}_{\text{Remainder, } R_0}
 \end{aligned}$$

In other words, $(b_0)_A$ is the remainder produced when $(N_I)_A$ is divided by $(B)_A$. In general, $(b_i)_A$ is the remainder, R_i , produced when quotient, Q_i , is divided by $(B)_A$. The conversion is completed by converting each $(b_i)_A$ to base B. However, this last step is trivial if $B < A$. The radix divide conversion procedure is summarized as follows.

1. Divide $(N_I)_A$ by the desired base $(B)_A$, producing quotient Q_1 and remainder R_0 . R_0 is the least significant digit, d_0 , of the result.
2. Compute each remaining digit, d_i , for $i = 1 \dots n-1$, by dividing quotient Q_i by $(B)_A$, producing quotient Q_{i+1} and remainder R_i , which represents d_i .
3. Stop when quotient $Q_{i+1} = 0$.

The radix divide method is illustrated in the next two examples.

Convert $(234)_{10}$ **to base 8.**

We solve this problem by repeatedly dividing integer $(234)_{10}$, that is $(N)_A$, by 8, that is $(B)_A$, until the quotient is 0.

$$\begin{array}{r} 2 \quad 9 \\ 8 \overline{)2 \quad 3 \quad 4} \\ 1 \quad 6 \\ \hline 7 \quad 4 \\ 7 \quad 2 \\ \hline 2 \end{array} = b_0$$

$$\begin{array}{r} 3 \\ 8 \overline{)2 \quad 9} \\ 2 \quad 4 \\ \hline 5 \end{array} = b_1$$

$$\begin{array}{r} 0 \\ 8 \overline{)3} \\ 0 \\ \hline 3 \end{array} = b_2$$

Hence, $(234)_{10} = (352)_8$. These calculations may be summarized in the following shorthand format.

$$\begin{array}{r} 2 \quad 3 \quad 4 \\ 8 \overline{)2 \quad 9} \\ 8 \quad | \\ 3 \quad | \\ 0 \end{array} \begin{array}{l} \uparrow \\ | \end{array} \begin{array}{l} \text{LSB} \\ \text{MSB} \end{array}$$

Convert $(234)_{10}$ **to base 16.**

$$\begin{array}{r} 1 \quad 4 \\ 16 \overline{)2 \quad 3 \quad 4} \\ 1 \quad 6 \\ \hline 7 \quad 4 \\ 6 \quad 4 \\ \hline 1 \quad 0 \end{array} = (A)_{16} = b_0$$

$$\begin{array}{r} 0 \\ 16 \overline{)1 \quad 4} \\ 16 \quad | \\ 0 \end{array} = (E)_{16} = b_1$$

Hence, $(234)_{10} = (EA)_{16}$. In the shorthand notation;

$$\begin{array}{r} 2 \quad 3 \quad 4 \quad 10 \\ 16 \overline{)1 \quad 4 \quad 14} \\ 16 \quad | \quad 0 \end{array} \begin{array}{l} \uparrow \\ | \end{array} \begin{array}{l} (A)_{16} \\ (E)_{16} \end{array}$$

Radix Multiply Method

Base conversions for fractions can be accomplished by the *radix multiply* method. Let N_F be a fraction in base A . The fraction can be written in series form as follows.

$$(N_F)_A = b_{-1}B^{-1} + b_{-2}B^{-2} + \dots + b_{-m}B^{-m} \quad (1.5)$$

The b_i 's in Eqn. 1.5 represent the digits of $(N_F)_B$ in base A . The most significant digit $(b_{-1})_A$ can be obtained by multiplying $(N_F)_A$ by $(B)_A$ as follows:

$$\begin{aligned} B \times N_F &= B \times (b_{-1}B^{-1} + b_{-2}B^{-2} + \dots + b_{-m}B^{-m}) \\ &= \underbrace{b_{-1}}_{\text{Integer}, I_{-1}} + \underbrace{b_{-2}B^{-1} + \dots + b_{-m}B^{-(m-1)}}_{\text{Fraction}, F_{-2}} \end{aligned}$$

Thus, $(b_{-1})_A$ is the integer part of the product that results from the multiplication of $(N_F)_A$ by $(B)_A$. In general, $(b_{-i})_A$ is the integer part, I_{-i} , of the product that results from multiplying the fraction $F_{-(i+1)}$ by $(B)_A$. Therefore, the radix multiply procedure is summarized as follows.

1. Let $F_{-1} = (N_F)_A$.
2. Compute digits $(b_{-i})_A$, for $i = 1 \dots m$, by multiplying F_i by $(B)_A$, producing integer I_{-i} , which represents digit $(b_{-i})_A$, and fraction $F_{-(i+1)}$.
3. Convert each digit $(b_{-i})_A$ to base B .

The following two examples illustrate this method.

Convert $(0.1285)_{10}$ to base 8.

$\begin{array}{r} 0 .1285 \\ \times \quad 8 \\ \hline 1 .0280 \end{array}$	$\begin{array}{r} 0 .0280 \\ \times \quad 8 \\ \hline 0 .2240 \end{array}$	$\begin{array}{r} 0 .2240 \\ \times \quad 8 \\ \hline 1 .7920 \end{array}$	$\begin{array}{r} 0 .7920 \\ \times \quad 8 \\ \hline 6 .3360 \end{array}$
\uparrow b_{-1}	\uparrow b_{-2}	\uparrow b_{-3}	\uparrow b_{-4}
\uparrow b_{-5}	\uparrow b_{-6}	\uparrow b_{-7}	\uparrow b_{-8}

Thus

$$0.1285_{10} = (0.10162540\dots)_8$$

Convert $(0.828125)_{10}$ to base 2. A shorthand notation will be used in this example when applying the radix multiply method. On each line, the fraction is multiplied by 2 to get the following line.

MSD	$1.656250 \leftarrow 0.828125 \times 2$ $1.312500 \leftarrow 0.656250 \times 2$ $0.625000 \leftarrow 0.312500 \times 2$ $1.250000 \leftarrow 0.625000 \times 2$ $0.500000 \leftarrow 0.250000 \times 2$ $1.000000 \leftarrow 0.500000 \times 2$
LSD	

Thus

$$0.828125_{10} = (0.110101)_2$$

The examples presented so far demonstrate the principles of base conversion. It is often helpful to define generalized procedures for solving various problems so that basic steps can be applied in the proper sequence. The base conversion methods used will now be formulated into two generalized conversion algorithms.

Algorithm 1.1 To convert a number N from base A to base B , use

- (a) the series substitution method with base B arithmetic, or
- (b) the radix divide or multiply method with base A arithmetic.

Algorithm 1.1 can be used for conversion between any two bases. However, it may be necessary to perform arithmetic in an unfamiliar base when doing so. The following algorithm overcomes this difficulty at the expense of a longer procedure.

Algorithm 1.2 To convert a number N from base A to base B , use

- (a) the series substitution method with base 10 arithmetic to convert N from base A to base 10, and
- (b) the radix divide or multiply method with decimal arithmetic to convert N from base 10 to base B .

Algorithm 1.2 in general requires more steps than Algorithm 1.1. However, the latter is often easier, faster, and less error prone because all arithmetic is performed in decimal.

1.3.2 Conversion between Base A and Base B When $B = A^k$

Simplified conversion procedures can be used when one base is a power of the other, for example, $B = A^k$. These procedures are very useful and are described next.

Algorithm 1.3

- (a) To convert a number N from base A to base B when $B = A^k$ and k is a positive integer, group the digits of N in groups of k digits in both directions from the radix point and then replace each group with the equivalent digit in base B .
- (b) To convert a number N from base B to base A when $B = A^k$ and k is a positive integer, replace each base B digit in N with the equivalent k digits in base A .

The following examples illustrate the power and speed of this algorithm for the case where $A = 2$.

Convert $(1011011.1010111)_2$ **to base 8.**

Algorithm 1.3a can be applied where $B = 8 = 2^3 = A^k$. Therefore, three binary digits are grouped for each octal digit.

$$\begin{array}{ccccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ \underbrace{1} & \underbrace{3} & \underbrace{3} & \underbrace{5} & \underbrace{3} & \underbrace{4} & & & \\ 1011011.1010111_2 & = & (133.534)_8 \end{array}$$

Convert $(AF.16C)_{16}$ **to base 8.**

Since both 16 and 8 are powers of 2, Algorithm 1.3 can be applied twice as follows.

Use Algorithm 1.3b to convert $(AF.16C)_{16}$ to base 2, since $16 = 2^4$. Each hexadecimal digit is replaced by four binary digits.

$$\begin{array}{ccccccccc} A & F & 1 & 6 & C \\ \underbrace{1} & \underbrace{0} & \underbrace{1} & \underbrace{1} & \underbrace{1} \\ 10101111.000101101100 \\ (AF.16C)_{16} & = & (10101111.0001011011)_2 \end{array}$$

Use Algorithm 1.3a to convert the binary number to base 8.

$$\begin{array}{ccccccccc} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \underbrace{2} & \underbrace{5} & \underbrace{7} & \underbrace{0} & \underbrace{5} & \underbrace{5} & \underbrace{4} & \\ 01010111.1000101101100 \end{array}$$

Therefore;

$$(AF.16C)_{16} = (257.0554)_8$$

1.4 Signed Number Representation

The sign of numbers used in digital systems is specified by a digit called the *sign digit*, which is usually placed in the leftmost digit position of the number, as illustrated in Fig. 1.2. Positive numbers are specified by a zero sign digit and negative numbers by a nonzero sign digit. The magnitude of a positive number is simply represented by its positional digits. However, several methods are available for representing the magnitude of negative numbers. Table 1.5 illustrates signed binary numbers in sign magnitude, two's complement, and one's complement methods. Each method is discussed in more detail below.

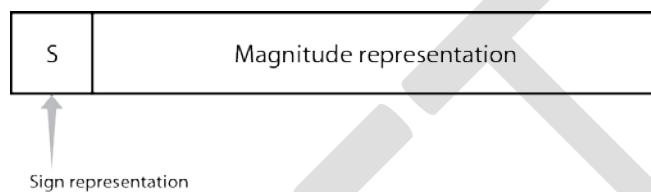


Figure 1.2 Signed number format.

1.4.1 Sign Magnitude Numbers

The simplest method of representing signed numbers is *sign magnitude*. However, the use of this method requires arithmetic hardware and algorithms that are more costly in circuitry and computation time than for other methods. Hence, the sign magnitude representation is not commonly used in practice for representing integer numbers.

A signed number $N = \pm(a_{n-1} \dots a_0.a_{-1} \dots a_{-m})_r$ may be written in sign magnitude form as follows.

$$N = (sa_{n-1} \dots a_0.a_{-1} \dots a_{-m})_{rsm} \quad (1.6)$$

where $s = 0$ if N is positive and $s = r - 1$ if N is negative.

Determine the sign-magnitude code of $N = -(13)_{10}$ in binary ($r = 2$) and decimal ($r = 10$).

In binary:

$$\begin{aligned} N &= -(13)_{10} \\ &= -(1101)_2 \\ &= (1,1101)_{2sm} \end{aligned}$$

In decimal:

$$\begin{aligned} N &= -(13)_{10} \\ &= (9,13)_{10sm} \end{aligned}$$

where 9 is used to represent the negative sign for $r = 10$. See Table 1.5 for more examples of sign magnitude binary numbers. For the sake of clarity, commas are used to delimit the sign digits.

1.4.2 Complementary Number Systems

Complementary numbers form the basis of complementary arithmetic, a powerful method often used in digital systems for handling mathematical operations on signed numbers. In these systems, positive numbers are represented in the same fashion as in a sign magnitude system, while negative numbers are represented as the complement of the corresponding positive number. Radix complement and diminished radix complement are important number systems and are discussed next. Complementary arithmetic is illustrated by examples as well.

TABLE 1.5 SIGNED NUMBER REPRESENTATION EXAMPLES*

Signed Decimal	Sign Magnitude Binary	Two's Complement System	One's Complement System
+15	0,1111	0,1111	0,1111
+14	0,1110	0,1110	0,1110
+13	0,1101	0,1101	0,1101
+12	0,1100	0,1100	0,1100
+11	0,1011	0,1011	0,1011
+10	0,1010	0,1010	0,1010
+9	0,1001	0,1001	0,1001
+8	0,1000	0,1000	0,1000
+7	0,0111	0,0111	0,0111
+6	0,0110	0,0110	0,0110
+5	0,0101	0,0101	0,0101
+4	0,0100	0,0100	0,0100
+3	0,0011	0,0011	0,0011
+2	0,0010	0,0010	0,0010
+1	0,0001	0,0001	0,0001
0	0,0000	0,0000	0,0000
	(1,0000)	--	(1,1111)
-1	1,0001	1,1111	1,1110
-2	1,0010	1,1110	1,1101
-3	1,0011	1,1101	1,1100
-4	1,0100	1,1100	1,1011
-5	1,0101	1,1011	1,1010
-6	1,0110	1,1010	1,1001
-7	1,0111	1,1001	1,1000
-8	1,1000	1,1000	1,0111
-9	1,1001	1,0111	1,0110
-10	1,1010	1,0110	1,0101
-11	1,1011	1,0101	1,0100
-12	1,1100	1,0100	1,0011
-13	1,1101	1,0011	1,0010
-14	1,1110	1,0010	1,0001
-15	1,1111	1,0001	1,0000
-16	--	1,0000	--

*Sign bits are delimited by commas.

Radix Complements

The *radix complement* $[N]_r$ of a number $(N)_r$ as defined in Eq. 1.2 is defined as

$$[N]_r = r^n - (N)_r \quad (1.7)$$

where n is the number of digits in $(N)_r$. The largest positive number (called *positive full scale*) that can be represented is $r^{n-1} - 1$, while the most negative number (called *negative full scale*) is $-r^{n-1}$.

The *two's complement* is the radix complement for binary numbers ($r = 2$) and is given by

$$[N]_2 = 2^n - (N)_2 \quad (1.8)$$

where n is the number of bits in $(N)_2$. Two's complement is the most commonly used format for signed numbers in digital systems and will be illustrated by the following examples.

Determine the two's complement of $(N)_2 = (01100101)_2$.

From Eq. 1.8,

$$\begin{aligned}[N]_2 &= [01100101]_2 \\ &= 2^8 - (01100101)_2 \\ &= (100000000)_2 - (01100101)_2 \\ &= (10011011)_2.\end{aligned}$$

Determine the two's complement of $(N)_2 = (11010100)_2$, and verify that it can be used to represent $-(N)_2$ by showing that $(N)_2 + [N]_2 = 0$.

First we determine the two's complement from Eq. 1.8.

$$\begin{aligned}[N]_2 &= [11010100]_2 \\ &= 2^8 - (11010100)_2 \\ &= (100000000)_2 - (11010100)_2 \\ &= (00101100)_2.\end{aligned}$$

To verify that $[N]_2$ can be used to represent $-(N)_2$, let us compute $(N)_2 + [N]_2$.

$$\begin{array}{r} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \uparrow & & & & & & & & \\ \text{carry} & & & & & & & & \end{array}$$

If we discard the carry bit, we get $(N)_2 + [N]_2 = (00000000)_2$, that is, the sum of a binary number and its two's complement is 0. Therefore, it follows that $[N]_2$ can be used to represent $-(N)_2$.

Determine the two's complement of $[N]_2 = (00101100)_2$, computed in the previous example.

$$\begin{aligned} [[N]_2]_2 &= [00101100]_2 \\ &= 2^8 - (00101100)_2 \\ &= (100000000)_2 - (00101100)_2 \\ &= (11010100)_2 \end{aligned}$$

Note that the result is the original value of $(N)_2$.

From the last example, we see that applying the two's complement operation to a number twice simply produces its original value. This is readily verified for the general case by substituting $-(N)_2$ for $[N]_2$ as follows.

$$\begin{aligned} [[N]_2]_2 &= [-(N)_2]_2 \\ &= -(-(N)_2)_2 \\ &= (N)_2 \end{aligned}$$

Determine the two's complement of $(N)_2 = (10110)_2$ for $n = 8$.

From Eq. 1.8,

$$\begin{aligned} [N]_2 &= [10110]_2 \\ &= 2^8 - (10110)_2 \\ &= (100000000)_2 - (10110)_2 \\ &= (11101010)_2 \end{aligned}$$

Note that we keep 8 bits in the result. The reader is encouraged to verify that this value of $[N]_2$ can be used to represent $-(N)_2$ and that $[[N]_2]_2 = (N)_2$.

While the radix complement of a number can always be determined by the definition given in Eq. 1.7, easier methods are available. The following two algorithms for computing $[N]_r$ given $(N)_r$ are presented without proof.

Algorithm 1.4 Find $[N]_r$ given $(N)_r$.

Copy the digits of N , beginning with the least significant and proceeding toward the most significant until the first nonzero digit has been reached. Replace this digit, a_i , with $r - a_i$. Then continue if necessary by replacing each remaining digit, a_j , of N by $(r-1) - a_j$ until the most significant digit has been replaced.

For the case of binary numbers ($r = 2$), the first nonzero digit, a_i , is by default a 1. Therefore, a_i is replaced by $r - a_i = 2 - 1 = 1$; hence a_i remains unchanged. Each remaining bit, a_j , is replaced by $(r-1) - a_j = 1 - a_j = \bar{a}_j$. Therefore, Algorithm 1.4 is applied to binary

numbers by simply copying, from right to left, all bits up to and including the first 1 bit and then complementing the remaining bits.

Find the two's complement of $N = (01100101)_2$.

$$\begin{aligned} N &= 01100101 \\ &\quad \Downarrow \text{first nonzero bit} \\ [N]_2 &= 10011011 \end{aligned}$$

Find the two's complement of $N = (11010100)_2$.

$$\begin{aligned} N &= 11010100 \\ &\quad \Downarrow \text{first nonzero bit} \\ [N]_2 &= 00101100 \end{aligned}$$

Find the two's complement of $N = (10110)_2$ **for** $n = 8$.

First, since $n = 8$, three zeros must be concatenated in the most significant bit positions to form an 8-bit number. Then apply Algorithm 1.4.

$$\begin{aligned} N &= 00010110 \\ &\quad \Downarrow \text{first nonzero bit} \\ [N]_2 &= 11101010 \end{aligned}$$

Algorithm 1.5 Find $[N]_r$ given $(N)_r$.

First replace each digit, a_k , of $(N)_r$ by $(r-1)-a_k$ and then add 1 to the resultant.

For the case of binary numbers ($r = 2$), we replace each bit, a_k , by $(r-1)-a_k = 1-a_k = \bar{a}_k$. Therefore Algorithm 1.5 is applied by simply complementing each bit and then adding 1 to the result.

Find the two's complement of $N = (01100101)_2$.

$$\begin{array}{rcl} N &= 01100101 & \\ &10011010 & \text{complement the bits} \\ &+1 & \text{add 1} \\ \hline [N]_2 &= 10011011 & \end{array}$$

Find the two's complement of $N = (11010100)_2$.

$$\begin{array}{rcl} N &= 11010100 & \\ &00101011 & \text{complement the bits} \\ &+1 & \text{add 1} \\ \hline [N]_2 &= 00101100 & \end{array}$$

Note that Algorithm 1.4 is convenient for hand calculations, while Algorithm 1.5 is more useful for machine implementation.

Radix Complement Number Systems

Previously the radix complement was defined and several methods for finding the radix complement of a given number were presented and illustrated. We also suggested by example that the radix complement of a number can be used to represent the negative of that number. Next, we describe more precisely a number system that utilizes the two's complement to represent negative numbers. Similar systems could be defined for other bases.

In the *two's complement number system*, positive values are represented in the same fashion as in the sign magnitude system; a leading bit of 0 is used to represent the sign. Negative numbers are represented by the two's complements of the corresponding positive number representations. We shall use the notation $(N)_{2cns}$ to denote a number that is represented in the two's complement number system. Thus, $N = +(a_{n-2}, \dots, a_0)_2 = (0, a_{n-2}, \dots, a_0)_{2cns}$, where $0 \leq N \leq 2^{n-1} - 1$. If $N = (a_{n-1}, a_{n-2}, \dots, a_0)_2$, then $-N$ is represented in the two's complement number system by $[a_{n-1}, \dots, a_0]_2$, where $-1 \geq -N \geq -2^{n-1}$. All negative numbers in the two's complement number system have a sign bit of 1.

The following examples illustrate the encoding of positive and negative numbers in the two's complement number system. The reader is encouraged to verify the two's complement entries in Table 1.5 after studying the examples.

Given $(N)_2 = (1100101)_2$, determine the two's complement number system representations of $\pm(N)_2$ for $n = 8$.

$$\text{By inspection, } +N = (0, 1100101)_{2cns}$$

$$\begin{aligned}\text{From Eq. 1.8, } -(N)_2 &= [+N]_2 \\ &= [0, 1100101]_2 \\ &= 2^8 - (0, 1100101)_2 \\ &= (100000000)_2 - (0, 1100101)_2 \\ &= (1, 0011011)_{2cns}.\end{aligned}$$

From their sign bits, we see that $(0, 1100101)_{2cns}$ represents a positive value and $(1, 0011011)_{2cns}$ is its negative. In this example and those that follow we shall use a comma to facilitate identifying the sign bit.

Find the two's complement number system representations of $\pm(110101)_2$ for $n = 8$.

$$\text{By inspection, } +N = (0, 0110101)_{2cns}$$

$$\begin{aligned}\text{From Eq. 1.8, } -(N)_2 &= [110101]_2 \\ &= 2^8 - (110101)_2 \\ &= (100000000)_2 - (110101)_2 \\ &= (1, 1001011)_{2cns}\end{aligned}$$

Determine the two's complement number system encoding of $-(13)_{10}$ for $n=8$.

We begin by converting $(13)_{10}$ from decimal to binary.

$$+(13)_{10} = +(1101)_2 = (0,0001101)_{2cns}$$

Next we compute the two's complement of $(0,0001101)_{2cns}$ to represent $-(13)_{10}$.

$$\begin{aligned} -(13)_{10} &= -(0,0001101)_{2cns} \\ &= [0,0001101]_2 \\ &= 2^8 - (0,0001101)_2 \\ &= (1,1110011)_{2cns} \end{aligned}$$

Determine the decimal number represented by $N = (1,1111010)_{2cns}$.

From the sign bit, we see that N is a negative number. Therefore, we determine the magnitude of N (the corresponding positive value) by computing its two's complement.

$$\begin{aligned} N &= (1,1111010)_{2cns} \\ &= -[1,1111010]_2 \\ &= -(2^8 - (1,1111010)_2) \\ &= -(0,0000110)_{2cns} \\ &= -(6)_{10} \end{aligned}$$

where $(0,0000110)_{2cns} = +(6)_{10}$. Therefore, $(1,1111010)_{2cns}$ represents $-(6)_{10}$.

Now let us consider some examples of arithmetic with radix complement numbers.

Radix Complement Arithmetic

Most digital computers use a radix complement number system to minimize the amount of circuitry needed to perform integer arithmetic. For example, the operation $A - B$ can be performed by computing $A + (-B)$, where $(-B)$ is represented by the two's complement of B . Hence, the computer need only have binary adder and complementing circuits to handle both addition and subtraction. This point of view is convenient for discussing radix complement arithmetic and will therefore be taken in the paragraphs that follow. Since computer arithmetic is primarily performed in binary, we shall focus our discussion on two's complement arithmetic.

Before beginning our discussion in depth, let us consider a fundamental limitation of the machine representation of numbers. Machines such as digital computers operate with finite number systems imposed by the number of bits that can be used in the representation of numerical quantities. In other words, the number of bits available in the computer's arithmetic unit limits the range of numbers that can be represented in the machine. Numbers that fall outside this range cannot be handled by the system. Machines that use the two's complement number system (2cns) can represent integers in the range

$$-2^{n-1} \leq N \leq 2^{n-1} - 1 \quad (1.9)$$

where n is the number of bits available for representing N . Note that $2^{n-1} - 1 = (0,11\dots1)_{2cns}$ and that $-2^{n-1} = (1,00\dots0)_{2cns}$ (the leftmost bit represents the sign and the remaining $n-1$ bits represent the magnitude).

If an operation produces a result that falls outside the available range as defined by Eq. 1.9, that is, if $N > 2^{n-1} - 1$ or $N < -2^{n-1}$, an *overflow condition* is said to occur. In such cases, the n -bit number produced by the operation will not be a valid representation of the result. Digital computers monitor their arithmetic operations when performing two's complement arithmetic and generate a warning signal when overflow occurs so that invalid numbers are not mistaken for correct results.

Three cases will now be considered for illustrating arithmetic in the two's complement number system: $A = B + C$, $A = B - C$, and $A = -B - C$. Each case will be described in general and then clarified by appropriate examples. All three cases are summarized in Table 1.6. For all cases, assume that $B \geq 0$ and $C \geq 0$. The results are easily generalized to include negative values of B and C .

Case 1 Compute $A = B + C$. Since both B and C are non negative, A will also be non negative, and this simply becomes $(A)_2 = (B)_2 + (C)_2$. Since all three numbers are positive, there is no need to use the two's complement.

The only difficulty that can arise in this case is when $A > 2^{n-1} - 1$, that is when an overflow occurs. An overflow condition is easily detected because the sign bit of A will be incorrect. To show this, consider the sum of the two largest representable n -bit positive numbers:

$$0 \leq A \leq (2^{n-1} - 1) + (2^{n-1} - 1) = 2^n - 2$$

Since the largest representable n -bit positive value is $2^{n-1} - 1$, an overflow condition occurs for any sum in the range

$$A \geq 2^n$$

The n^{th} bit of any binary number in this range will be set to 1. Unfortunately, this happens to be the bit that represents the sign in an n -bit two's complement number. Therefore, the result appears to be a negative number, thus indicating the overflow condition.

It should be noted that since $A < 2^n$ there will never be a carry out of the n^{th} bit of the binary adder for Case 1.

The following examples will utilize the 5-bit two's complement number system whose values are listed in Table 1.5.

Compute $(9)_{10} + (5)_{10}$ **using 5-bit two's complement arithmetic.**

We begin by writing $(9)_{10}$ and $(5)_{10}$ as 5-bit two's complement numbers. Since both numbers are positive, a zero sign bit is used for each. From Table 1.5 (page 17),

$$\begin{aligned} +(9)_{10} &= +(1001)_2 = (0,1001)_{2cns} \\ +(5)_{10} &= +(0101)_2 = (0,0101)_{2cns} \end{aligned}$$

Adding these two 5-bit codes gives

$$\begin{array}{r}
 0 & 1 & 0 & 0 & 1 \\
 + & 0 & 0 & 1 & 0 \\
 \hline
 0 & 1 & 1 & 1 & 0
 \end{array}$$

Since the result also has a zero sign bit, it correctly represents the desired positive sum, which is interpreted as

$$(0,1110)_{2cns} = +(1110)_2 = +(14)_{10}$$

Compute $(12)_{10} + (7)_{10}$.

$$\text{From Table 1.5, } (12)_{10} = +(1100)_2 = (0,1100)_{2cns}$$

$$(7)_{10} = +(0111)_2 = (0,0111)_{2cns}$$

Adding the two 5-bit codes gives

$$\begin{array}{r}
 0 & 1 & 1 & 0 & 0 \\
 + & 0 & 0 & 1 & 1 \\
 \hline
 1 & 0 & 0 & 1 & 1
 \end{array}$$

The result is $(1,0011)_{2cns}$, which from Table 1.5 would be interpreted as

$$(1,0011)_{2cns} = -(1101)_2 = -(13)_{10}$$

This is not the desired result since the correct sum is $+(19)_{10}$. However, the largest number that can be represented in a 5-bit two's complement number system is $(0,1111)_{2cns} = +(15)_{10}$. So an overflow has occurred. The overflow can be detected by observing that the sign bit shows a negative result which is incorrect given that both the augend and addend are positive numbers. That is, the sum of two positive numbers cannot be a negative number. An overflow is also indicated when the sum of two negative numbers produces a result that has a negative sign bit (1).

Case 2 Compute $A = B - C$. The computation is treated as $A = B + (-C)$ in the following manner. Let $A = (B)_2 + (-C)_2$. Encoding the numbers in two's complement leaves the positive number $(B)_2$ unchanged with $-(C)_2$ becoming $[C]_2$. Hence,

$$\begin{aligned}
 A &= (B)_2 + [C]_2 \\
 &= (B)_2 + 2^n - (C)_2 \\
 &= 2^n + (B - C)_2
 \end{aligned}$$

This is the answer we want, except that there is an extra 2^n term. Can we ignore it? If $B \geq C$, then $B - C \geq 0$, making $A \geq 2^n$, so the 2^n term represents a carry bit and can be discarded, leaving $(B - C)_2$ (an n -bit binary adder will generate a carry for any sum $A \geq 2^n$). Therefore,

$$(A)_2 = (B)_2 + [C]_2 \mid_{\text{carry discarded}}$$

If $B < C$, then $B - C < 0$, giving $A = 2^n - (C - B)_2 = [C - B]_2$, or $A = -(C - B)_2$ which is the desired answer. Note that there is no carry in this instance.

When B and C are both positive numbers, the magnitude of $B - C$ will always be less than either of the two numbers. This means that no overflow can occur when computing $B - C$.

Compute $(12)_{10} - (5)_{10}$.

We perform this computation as $(12)_{10} + (-5)_{10}$

$$(12)_{10} = +(1100)_2 = (0,1100)_{2cns}$$

$$-(5)_{10} = -(0101)_2 = (1,1011)_{2cns}$$

Adding the two 5-bit codes gives

$$\begin{array}{r}
 & 0 & 1 & 1 & 0 & 0 \\
 + & 1 & 1 & 0 & 1 & 1 \\
 \hline
 & 1 & 0 & 1 & 1 & 1 \\
 \uparrow & & & & & \\
 \text{Discard} & & & & & \\
 \text{Carry} & & & & &
 \end{array}$$

Discarding the carry, the sign bit is seen to be 0, and therefore the result is interpreted as

$$(0,0111)_{2cns} = +(0111)_2 = +(7)_{10}$$

Reversing the order of the operands from the previous example, compute $(5)_{10} - (12)_{10}$.

We perform the computation as $(5)_{10} + (-12)_{10}$.

$$(5)_{10} = +(0101)_2 = (0,0101)_{2cns}$$

$$-(12)_{10} = -(1100)_2 = (1,0100)_{2cns}$$

Adding the two 5-bit codes gives

$$\begin{array}{r}
 & 0 & 0 & 1 & 0 & 1 \\
 + & 1 & 0 & 1 & 0 & 0 \\
 \hline
 & 1 & 1 & 0 & 0 & 1
 \end{array}$$

In this case there is no carry, and the sign bit is 1, indicating a negative result, which is

$$(1,1001)_{2cns} = -(0111)_2 = -(7)_{10}$$

Compute $(0,0111)_{2cns} - (1,1010)_{2cns}$.

We perform the computation as $(0,0111)_{2cns} + (-1,1010)_{2cns}$. The left-hand operand is already in two's complement number system format. Since its sign bit is 1, the right-hand operand represents a negative number. To negate it, we take the two's complement of this negative number to get the corresponding positive value. Note from the definition of two's complement that

$$\begin{aligned}
 -[X]_2 &= [[X]_2]_2 \\
 &= 2^n - [X]_2 \\
 &= 2^n - (2^n - X) \\
 &= X
 \end{aligned}$$

Therefore,

$$-(1,1010)_{2cns} = (0,0110)_{2cns}$$

Adding the two 5-bit codes gives

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 1 \quad 1 \\ + \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \\ \hline 0 \quad 1 \quad 1 \quad 0 \quad 1 \end{array}$$

The result is positive, as indicated by the 0 sign bit, and is interpreted as

$$(0,1101)_{2cns} = +(1101)_2 = +(13)_{10}$$

The reader should verify that this computation is equivalent to computing $(7)_{10} - (-(6)_{10}) = (13)_{10}$.

Case 3 Compute $A = -B - C$. The desired result is $A = -(B + C) = [B + C]_2$.

Both $-B$ and $-C$ will be represented by the two's complements of their magnitudes, and the computation will be performed as $A = (-B) + (-C)$. Therefore,

$$\begin{aligned} A &= [B]_2 + [C]_2 \\ &= 2^n - (B)_2 + 2^n - (C)_2 \\ &= 2^n + 2^n - (B + C)_2 \\ &= 2^n + [B + C]_2 \end{aligned}$$

If the carry bit (2^n) is discarded, the computation produces the correct result, the two's complement representation of $-(B + C)_2$.

Compute $-(9)_{10} - (5)_{10}$.

We perform the computation as $-(9)_{10} + (-5)_{10}$.

$$\begin{aligned} -(9)_{10} &= -(1001)_2 = (1,0111)_{2cns} \\ -(5)_{10} &= -(0101)_2 = (1,1011)_{2cns} \end{aligned}$$

Adding the two 5-bit codes gives

$$\begin{array}{r} & 1 & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \\ \uparrow & & & & & \\ \text{Discard} & & & & & \\ \text{Carry} & & & & & \end{array}$$

Discarding the carry leaves a sign bit of 1. Therefore, the result is correct and is interpreted as

$$(1,0010)_{2cns} = -(1110)_2 = -(14)_{10}$$

As is the case when adding two positive values, an overflow can occur when adding two negative values, that produce a result in the range

$$A < -2^{n-1}$$

which is indicated by a result having an incorrect sign bit (that is, a result that appears to be positive). This is illustrated in the following example.

Compute $-(12)_{10} - (5)_{10}$.

We perform the computation as $-(12)_{10} + (-5)_{10}$ with

$$-(12)_{10} = -(1100)_2 = (1,0100)_{2cns}$$

$$-(5)_{10} = -(0101)_2 = (1,1011)_{2cns}$$

Adding the two 5-bit codes gives

$$\begin{array}{r}
 & 1 & 0 & 1 & 0 & 0 \\
 + & 1 & 1 & 0 & 1 & 1 \\
 \hline
 & 1 & 0 & 1 & 1 & 1
 \end{array}$$

↑
 Discard
 Carry

Discarding the carry, the result is interpreted as

$$(0,1111)_{2cns} = +(1111)_2 = +(15)_{10}$$

Note that the sign bit is incorrect, indicating an overflow. This result is "too negative"; it exceeds the number range in the negative direction by 1, since the desired result was $-(17)_{10}$. Consequently, because of the overflow the result is incorrectly interpreted as $+(15)_{10}$.

The next example illustrates the utility of two's complement arithmetic in digital computers.

A and B are integer variables in a computer program, with $A = (25)_{10}$ and $B = -(46)_{10}$. Assuming that the computer uses 8-bit two's complement arithmetic, show how it would compute $A+B$, $A-B$, $B-A$, and $-A-B$.

Variables A and B would be stored in the memory of the computer in 8-bit two's complement number system format.

$$A = +(25)_{10} = (0,0011001)_{2cns}$$

$$B = -(46)_{10} = -(0,0101110)_{2cns} = (1,1010010)_{2cns}$$

First, let us compute the two's complements of A and B to represent $-A$ and $-B$, respectively.

$$-A = -(25)_{10} = -(0,0011001)_{2cns} = (1,1100111)_{2cns}$$

$$-B = -(-46)_{10} = -(1,1010010)_{2cns} = (0,0101110)_{2cns}$$

Performing the computations,

$A+B$:

$$\begin{array}{r}
 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 + & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
 \hline
 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1
 \end{array}$$

The result is $(1,1101011)_{2cns} = -(0,0010101)_{2cns} = -(21)_{10}$.

$$A - B = A + (-B)$$

$$\begin{array}{r}
 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 + & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 \hline
 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1
 \end{array}$$

The result is $(0,1000111)_{2cns} = +(71)_{10}$.

$$B - A = B + (-A)$$

$$\begin{array}{r}
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
 + & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 \hline
 \pm & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1
 \end{array}$$

The result is $(1,0111001)_{2cns} = -(0,1000111)_{2cns} = -(71)_{10}$.

$$-A - B = (-A) + (-B)$$

$$\begin{array}{r}
 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
 + & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 \hline
 \pm & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1
 \end{array}$$

The result is $(0,0010101)_{2cns} = +(21)_{10}$. Note that in the last two cases the carry bit is discarded.

A summary of two's complement addition and subtraction is given in Table 1.6.

TABLE 1.6 SUMMARY OF TWO'S COMPLEMENT ADDITION AND SUBTRACTION

Case*	Carry	Sign Bit	Condition	Overflow?
$B + C$	0	0	$B + C \leq 2^{n-1} - 1$	No
	0	1	$B + C > 2^{n-1} - 1$	Yes
$B - C$	1	0	$B \leq C$	No
	0	1	$B > C$	No
$-B - C$	1	1	$-(B + C) \geq -2^{n-1}$	No
	1	0	$-(B + C) < -2^{n-1}$	Yes

* B and C are positive numbers.

Diminished Radix Complement Number Systems

The *diminished radix complement* $[N]_{r-1}$ of a number $(N)_r$ is defined as

$$[N]_{r-1} = r^n - (N)_r - 1 \quad (1.10)$$

where n is the number of digits in $(N)_r$.

The *one's complement* is the diminished radix complement for binary numbers ($r = 2$) and is given by

$$[N]_{2-1} = 2^n - (N)_2 - 1 \quad (1.11)$$

where n is the number of bits in $(N)_2$.

The one's complement of a given binary number can be found directly from Eq. 1.11 as illustrated in the following examples. The reader is encouraged to verify the one's complement entries in Table 1.5 after studying the examples.

Determine the one's complement of $(01100101)_2$.

From Eq. 1.11,

$$\begin{aligned}[N]_{2-1} &= 2^8 - (01100101)_2 - 1 \\ &= (100000000)_2 - (01100101)_2 - (00000001)_2 \\ &= (10011011)_2 - (00000001)_2 \\ &= (10011010)_2.\end{aligned}$$

Determine the one's complement of $(11010100)_2$.

From Eq. 1.11,

$$\begin{aligned}[N]_{2-1} &= 2^8 - (11010100)_2 - (00000001)_2 \\ &= (100000000)_2 - (11010100)_2 - (00000001)_2 \\ &= (00101100)_2 - (00000001)_2 \\ &= (00101011)_2.\end{aligned}$$

While the one's complement of a number can always be determined by the definition given in Eq. 1.11, easier methods are available. The following algorithm for computing $[N]_{r-1}$ given $(N)_r$ is suggested by the preceding examples and is presented without proof.

Algorithm 1.6 Find $[N]_{r-1}$ given $(N)_r$.

Replace each digit a_i of $(N)_r$ by $r-1-a_i$. Note that when $r=2$ this simplifies to complementing each individual bit of $(N)_r$.

A comparison of Eqs. 1.7 and 1.10 indicates that the radix complement and the diminished radix complement of a number $(N)_r$ are related as follows.

$$[N]_r = [N]_{r-1} + 1 \quad (1.12)$$

It should now be clear that Algorithm 1.5 for finding the radix complement follows from Algorithm 1.6.

Number systems that use the diminished radix complement for negative number representation can be formulated in a manner reminiscent of that used with radix complement. However, this will not be done here. Instead, we will simply illustrate the arithmetic.

Diminished Radix Complement Arithmetic

The key features of diminished radix complement arithmetic are illustrated in the following examples which focus on one's complement addition for various combinations of positive and negative operands. The numbers used in these examples are from Table 1.5.

Add $+(1001)_2$ **and** $-(0100)_2$.

The positive number is represented by 01001 and the negative number by the one's complement of 00100, which is 11011. Hence $00100+11011=100100$. Note that this is *not* the correct result. However, the correct result is obtained if the carry-out of the most significant bit is added to the least significant bit position. That is, $00100+1=00101$. This procedure is referred to as an *end-around carry* and is a necessary correction step in diminished complement arithmetic.

Add $+(1001)_2$ **and** $-(1111)_2$.

The positive number is represented by 01001 and the negative by 10000. This results in $01001+10000=11001$. Note that in this case the end-around carry is 0 and therefore does not affect the result.

Add $-(1001)_2$ **and** $-(0011)_2$.

Representing each number by its one's complement yields $10110+11100=110010$. The end-around carry step yields the correct result, that is, $10010+1=10011$.

1.5 Digital Codes

A *code* is a systematic and often standardized use of a given set of symbols for representing information. Simple forms of codes are encountered routinely in everyday life. For example, when a traffic light is approached, it is understood that a red signal means *stop*, a green signal means *go*, and a yellow signal means *caution*. In other words, the code is

Red light:	Stop
Yellow light:	Caution
Green light:	Go

Codes of a more complex nature are used in computers and other digital systems in the processing, storage, and exchange of information of various types. Four important types of digital codes are numeric, character, error detection, and error correction. Some important codes in each of these categories are discussed briefly below.

1.5.1 Numeric Codes

Numeric codes are typically used to represent numbers for processing and/or storage. Fixed-point and floating-point numbers are examples of such codes.

Fixed-point Numbers

Fixed-point numbers are used to represent either signed integers or signed fractions. In both cases, either sign magnitude, two's complement, or one's complement systems are used for representing the signed values. Fixed-point integers have an implied binary point to the right of the least significant bit, as shown in Fig. 1.3a, and fixed-point fractions have the implied binary point between the sign bit and the most significant magnitude bit, as shown in Fig. 1.3b.

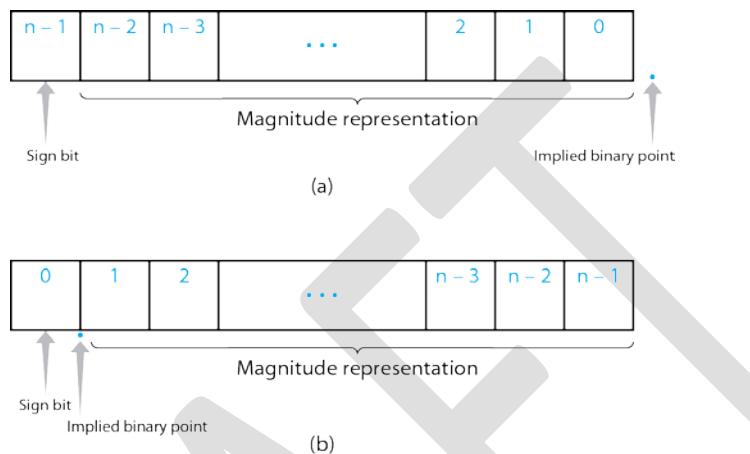


Figure 1.3: Fixed-point number representations. (a) Fixed-point integer. (b) Fixed-point fraction.

Give two possible interpretations of the 8-bit fixed-point number 01101010, assuming a two's complement number system.

Since the sign bit is 0, the number represents either the positive integer 1101010 if the binary point is placed as in Fig. 1.3a, or the positive fraction 0.1101010 if the binary point is placed as in Fig. 1.3b.

Give two possible interpretations of the 8-bit fixed-point number 11101010, assuming a two's complement number system.

The sign bit is 1. Therefore, the number represents either -0010110 or -0.0010110 depending on the convention used for placement of the binary point.

Excess or Biased Representations

An *excess- K* representation of a code C is formed by adding the value K to each code word of C . Excess representations are frequently used in the representation of the exponents of floating-point numbers so that the smallest exponent value will be represented by all zeros. Note that the excess- 2^n numbers are just the two's complement numbers with the sign bit inverted!

The excess-8 representation shown in Table 1.7 is produced by adding $(1000)_2$ to the 4-bit two's complement code. Note that the result is the smallest number (-8) being represented by 0000 and the largest (+7) by 1111.

Floating-point Numbers

Floating-point numbers are similar in form to numbers written in scientific notation. In general, the floating-point form of a number N is written as

$$N = M \times r^E \quad (1.13)$$

where M , the *mantissa* or *significand*, is a fixed-point number containing the most significant digits of N , and E , the *exponent* or *characteristic*, is a fixed-point integer representing the scale of N . In the general case, given a fixed-point number N , where

$$N = \pm(a_{n-1} \dots a_0.a_{-1} \dots a_{-m})_r$$

then in floating-point form

$$N = \pm(.a_{n-1} \dots a_{-m})_r \times r^n$$

When deriving a representation of a floating-point number, the mantissa and exponent are coded separately. The radix is implied and is thus not included in the representation.

TABLE 1.7 EXCESS-8 CODE

Decimal	Two's Complement	Excess-8
+7	0111	1111
+6	0110	1110
+5	0101	1101
+4	0100	1100
+3	0011	1011
+2	0010	1010
+1	0001	1001
0	0000	1000
-1	1111	0111
-2	1110	0110
-3	1101	0101
-4	1100	0100
-5	1011	0011
-6	1010	0010
-7	1001	0001
-8	1000	0000

The mantissa M is often coded in sign magnitude, usually as a fraction, and can be written as

$$M = (S_M.a_{n-1} \dots a_{-m})_{rsm} \quad (1.14)$$

where $(.a_{n-1} \dots a_{-m})_r$ represents the magnitude of M and S_M indicates the sign of the number. S_M is usually chosen so that

$$M = (-1)^{S_M} \times (.a_{n-1} \dots a_{-m})_r \quad (1.15)$$

and thus $S_M = 0$ indicates a positive number, whereas $S_M = 1$ indicates a negative number.

The exponent E is often coded in *excess- K* *two's complement*. The excess- K two's complement of an exponent is formed by adding a *bias* of K to the two's complement integer value of the exponent. For binary floating-point numbers (numbers for which radix $r = 2$), K is usually selected to be 2^{e-1} , where e is the number of bits in the exponent. Therefore, $-2^{e-1} \leq E < 2^{e-1}$ and $0 \leq E + 2^{e-1} < 2^e$ which indicate that the biased value of E is a number that ranges from 0 to $2^e - 1$ as E increases from its most negative to its most positive value. The excess- K form of E can be written as

$$E = (b_{e-1}, b_{e-2} \dots b_0)_{\text{excess-}K} \quad (1.16)$$

where b_{e-1} indicates the sign of E .

M and E , coded via Eqs. 1.14 and 1.16, are combined to produce the following floating-point number format:

$$N = (S_M b_{e-1} b_{e-2} \dots b_0 a_{n-1} \dots a_{-m})_r \quad (1.17)$$

representing the number

$$N = (-1)^{S_M} \times (.a_{n-1} \dots a_{-m})_r \times r^{(b_{e-1} b_{e-2} \dots b_0) - 2^{e-1}} \quad (1.18)$$

One exception to the format of Eq. 1.17 is the number 0, which is treated as a special case and is usually represented by an all-zero word.

Floating-point representations of a given number are not unique. Given a number N , as defined in Eq. 1.13, it can be seen that

$$N = M \times r^E \quad (1.19)$$

$$= (M \div r) \times r^{E+1} \quad (1.20)$$

$$= (M \times r) \times r^{E-1} \quad (1.21)$$

where $(M \div r)$ is performed by shifting the digits of M one position to the right, and $(M \times r)$ is performed by shifting the digits of M one position to the left. Therefore, more than one combination of mantissa and exponent can represent the same number. For example, let $M = +(1101.0101)_2$. Representing M as a sign magnitude fraction in the format of Eq. 1.14 and repeatedly applying Eq. 1.20 gives

$$M = +(1101.0101)_2 \quad (1.22)$$

$$= (0.11010101)_2 \times 2^4 \quad (1.23)$$

$$= (0.011010101)_2 \times 2^5 \quad (1.24)$$

$$= (0.0011010101)_2 \times 2^6 \quad (1.24)$$

⋮

When performing computations in a computer, it is desirable to have a unique representation for each number. *Normalization* is used to provide uniqueness for floating-point numbers. A floating-point number is said to be *normalized* if the exponent is adjusted so that the mantissa has a nonzero value in its most significant digit position. Therefore, Eq. 1.22 gives the normalized representation of N , while the numbers in Eqs. 1.23 and 1.24 are not normalized.

Note that the most significant bit of a normalized binary number is always 1. Therefore, if M is represented in sign magnitude form as a normalized fraction, $0.5 \leq |M| < 1$.

The following examples illustrate the encoding of floating-point numbers.

Write the binary number $N = (101101.101)_2$ in the floating-point format of Eq. 1.17, where $n+m=10$ and $e=5$. Assume that a normalized sign magnitude fraction is used to represent M and that Excess-16 two's complement is used for E .

First, normalize the number and then encode the mantissa (M) and exponent (E).

$$N = (101101.101)_2 = (0.101101101)_2 \times 2^6$$

Writing the mantissa in the format of Eq. 1.14

$$\begin{aligned} M &= +(0.101101101)_2 \\ &= (0.101101101)_{2sm} \end{aligned}$$

The exponent is coded by first deriving its two's complement form and then adding a bias of 16. (Note that the number of exponent bits $e=5$ and that the bias value is $2^{e-1} = 2^4 = 16$). Therefore,

$$\begin{aligned} E &= +(6)_{10} \\ &= +(0110)_2 \\ &= (00110)_{2cns} \end{aligned}$$

Adding the bias value of $16 = (10000)_2$ to the two's complement of E yields

$$\begin{array}{r} 00110 \\ + 10000 \\ \hline 10110 \end{array}$$

So,

$$E = (1,0110)_{excess-16}$$

Note that the sign of the exponent, b_{e-1} , is 1, indicating a positive exponent value.

Combining M and E gives $N = (0101101011011010)_{fp}$

Repeat the previous example for $N = -(0.00101101101)_2$.

Normalizing N yields $N = - (0.101101101)_2 \times 2^2$. Converting the mantissa to sign-magnitude results in $M = (1.101101101)_{2sm}$. The exponent is first encoded in 2's complement and then the bias value is added as follows. $E = - (2)_{10} = - (0010)_2 = (1110)_{2cns} = (01110)_{excess-16}$. Combining M and E according to Eq. 1.17 yields the following representation of N .

$$N = (1011101011011010)_{fp}$$

Arithmetic operations on floating-point numbers require special algorithms to manipulate exponents and mantissas, which are beyond the scope of this text.

Floating-point formats used in computer systems from different manufacturers often vary in the numbers of bits used to represent the mantissa and exponent and the method of coding used for each. However, most systems utilize the general formats illustrated in Fig. 1.4, with the sign stored in the leftmost bit, followed by the exponent and then the mantissa. The one-word format of Fig. 1.4a is typically used in computers with word lengths of 32 bits or more. The two-word format of Fig. 1.4b is used in computers with “short” word lengths for single-precision floating-point numbers or in computers with “long” word lengths for extended-precision (also called double-precision) representation.

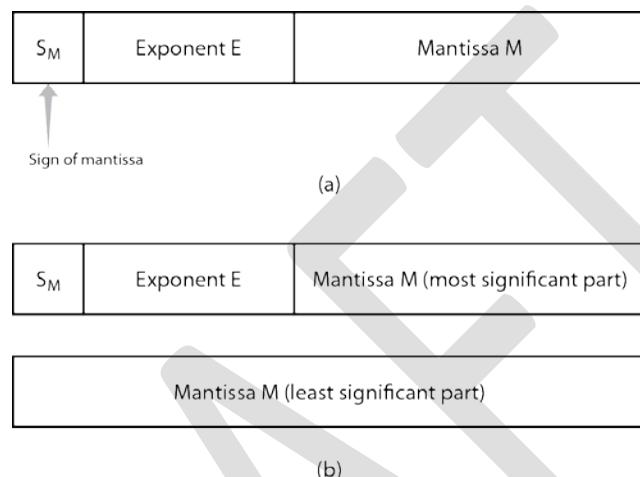


Figure 1.4 Floating-point number formats. **a**Typical single-precision format. **b**Typical extended-precision format.

The Institute of Electrical and Electronic Engineers (IEEE) developed a standard (IEEE Std 754) for floating-point arithmetic that has influenced recent generations of floating-point implementations. The standard was first approved in 1985 and revised in 2008. IEEE 754-2008 specifies three binary and two decimal formats for representing floating-point numbers. Table 1.8 includes a summary of the binary 32, binary 64, and binary 128 formats. Note that all of these formats use a biased exponent, with a varying number of bits. Also, the formats suppress storage of the most significant bit of the mantissa since all numbers are binary values in normalized form and the most significant bit is known to be 1. Therefore, storage of this bit is unnecessary, and thus one additional bit of precision can be gained (denoted by the +1 in the fifth column of Table 1.8).

TABLE 1.8 IEEE 754-2008 BINARY FLOATING-POINT NUMBER FORMATS

Format	Total Bits	Exponent Bits	Exponent Bias	Significand Bits	Mantissa (M) Coding
Binary 32	32	8	127	23 (+1)	
Binary 64	64	11	1,023	52 (+1)	Sign/magnitude, $1 \leq M < 2$
Binary 128	128	15	16,383	112 (+1)	

1.5.2 Character and Other Codes

It is often necessary or desirable to represent information as strings of alphabetical or numerical characters. Numerous character codes have been developed for this purpose and some of the most important ones will now be discussed.

Binary Coded Decimal (BCD)

The *binary coded decimal* or *BCD* code is used for representing the decimal digits 0 through 9 and is an example of a *weighted* code. That is, each bit position in the code has a fixed numerical value or weight associated with it. The digit represented by a given code word can be found by summing up the weighted bits. The BCD code uses 4 bits, with the weights chosen to be the same as those of a 4-bit binary integer. Hence, the BCD code for a given decimal digit is the same as the binary equivalent of the number with leading zeros. BCD codes are sometimes referred to as 8–4–2–1 codes because of the weights used. The complete BCD code is given in Table 1.9.

TABLE 1.9 BINARY CODED DECIMAL (BCD) CODES

Decimal Digit	BCD CODE	Decimal Digit	BCD CODE
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

BCD codes are used to encode numbers for output to numerical displays and for representing numbers in processors that perform decimal arithmetic. Hand-held calculators are an example of the latter.

Encode the decimal number $N = (9750)_{10}$ in BCD.

First, the individual digits are encoded from Table 1.9.

$$9 \rightarrow 1001, 7 \rightarrow 0111, 5 \rightarrow 0101, \text{ and } 0 \rightarrow 0000$$

Then the individual codes are concatenated to give

$$N = (1001011101010000)_{BCD}$$

Extensions of the BCD code have been developed to cover not only the decimal digits but also alphabetical and other printing characters, as well as nonprinting control characters. These codes are typically 7 or more bits in length. They are used for representing data during input or output and for internally representing nonnumeric data such as text. One such code, developed by IBM, is the *Extended Binary Coded Decimal Interchange Code (EBCDIC)*.

ASCII

The most widely used character code in computer applications is the *ASCII* (American Standard Code for Information Interchange) code, pronounced “askey.” The 7-bit ASCII code is given in Table 1.10. An eighth bit is often used with the 7-bit code to provide error detection. This technique, parity coding, is discussed later in the chapter. An 8-bit version of the ASCII code has been developed where the eighth bit is used to double the number of characters that can be encoded. However, the 8-bit code does not include enough characters to meet the international user community needs. So, the Unicode Consortium has developed codes with larger character sets that are compatible with ASCII.

TABLE 1.10 7-BIT ASCII CHARACTER CODE

	000	001	010	011	100	101	110	111
	<i>c₆c₅c₄</i>							
c₃c₂c₁c₀	0000	NUL	DLE	SP	0	@	P	`
	0001	SOH	DC1	!	1	A	Q	a
	0010	STX	DC2	"	2	B	R	b
	0011	ETX	DC3	#	3	C	S	c
	0100	EOT	DC4	\$	4	D	T	d
	0101	ENQ	NAK	%	5	E	U	u
	0110	ACK	SYN	&	6	F	V	f
	0111	BEL	ETB	'	7	G	W	g
	1000	BS	CAN	(8	H	X	h
	1001	HT	EM)	9	I	Y	i
	1010	LF	SUB	*	:	J	Z	j
	1011	VT	ESC	+	;	K	[k
	1100	FF	FS	,	<	L	\	l
	1101	CR	GS	-	=	M]	m
	1110	SO	RS	.	>	N	^	n
	1111	S1	US	/	?	O	_	o
								DEL

Encode the word Digital in 7-bit ASCII code, representing each character by two hexadecimal digits.

Character	Binary Code	Hexadecimal Code
D	1000100	44
i	1101001	69
g	1100111	67
i	1101001	69
t	1110100	74
a	1100001	61
l	1101100	6C

Note that the hexadecimal form is more compact and readable than the binary form. For this reason, the former is often used when representing ASCII coded information as in the above example.

Gray Codes

A *cyclic code* may be defined as any code in which, for any code word, a circular shift produces another code word. The *Gray code* is one of the most common types of cyclic codes and has the characteristic that the code words for two consecutive numbers differ in only 1 bit. That is, the distance between the two code words is 1. In general, the *distance* between two binary code words is equal to the number of bits in which the two words differ.

Define a Gray code for encoding the decimal numbers 0 through 15.

Four bits are needed to represent all the numbers, and the necessary code can be constructed by assigning bit i of the code word to be 0 if bits i and $i+1$ of the corresponding binary number are the same and 1 otherwise. The most significant bit of the number must always be compared with 0 when using this technique. The resulting code is given in Table 1.11.

TABLE 1.11 GRAY CODE FOR DECIMAL NUMBERS 0 THROUGH 15

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

The need to observe or measure the position of a circular shaft occurs in many applications. This can be accomplished by mounting an encoded conducting disk on the shaft and electrically sensing the position of the disk. How can the disk be encoded so that incorrect position indications are not read when the sensors move from one sector of the disk to another?

The desired result can be obtained if the disk sectors are encoded in a Gray code since only one bit position in the code will change as the sensors move from one sector to the next. Figure 1.5 illustrates the solution.

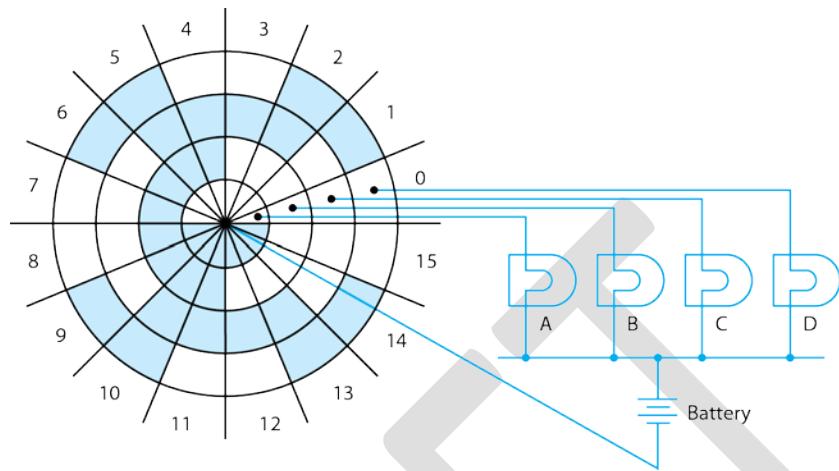


Figure 1.5 Gray-coded disk.

1.5.3 Error Detection and Correction Codes

An *error* in binary data is defined as an incorrect value in one or more bits. A *single error* refers to an incorrect value in only one bit, while a *multiple error* refers to one or more bits being incorrect. Errors may be introduced by hardware failures, external interference (noise), or other unwanted events. Information may be encoded using special codes that allow the detection and sometimes the correction of certain classes of errors. Some simple error detection and correction codes are illustrated next.

It will be useful to state some definitions and notations before presenting specific codes. Let I and J be n -bit binary information words. The *weight* of I , $w(I)$, is defined to be the number of bits of I equal to 1. The *distance between I and J* , $d(I, J)$, is equal to the number of bit positions in which I and J differ.

Determine the weights of I and J and the distance between them if $I = (01101100)$ and $J = (11000100)$.

Counting the 1 bits in each number, we get

$$w(I) = 4 \text{ and } w(J) = 3$$

Next, we compare the two numbers bit by bit, noting where they differ as follows:

0	1	1	0	1	1	0	0
1	1	0	0	0	1	0	0
↑	↑	↑	↑				

The numbers differ in three bit positions, therefore, $d(I, J) = 3$.

General Properties of Error Detection and Correction Codes

If the distance between any two code words of a code C is $\geq d_{\min}$, the code is said to have *minimum distance* d_{\min} . The error detection and correction properties of a code are determined in part by its minimum distance. This is illustrated in Figure 1.6, in which circled dots represent valid code words and uncircled dots represent words that contain errors. Two dots are connected if the corresponding words differ in exactly one bit position. For a given d_{\min} , at least d_{\min} errors are needed to transform one valid code word to another. If there are fewer than d_{\min} errors, then a detectable noncode word results. If the noncode word is "closer" to one valid code word than to any other, the original code word can be deduced, and thus the error can be corrected.

In general, a code provides t *error correction* plus detection of s additional errors if and only if the following inequality is satisfied.

$$2t + s + 1 \leq d_{\min} \quad (1.25)$$

It can be seen from a closer examination of Eq. 1.25 that a single-error detection code ($s = 1, t = 0$) requires a minimum distance of 2. A single-error correction code ($s = 0, t = 1$) requires a minimum distance of 3, and a code with both single-error correction and double-error detection ($s = t = 1$) requires a minimum distance of 4. Figure 1.6 illustrates these and other combinations.

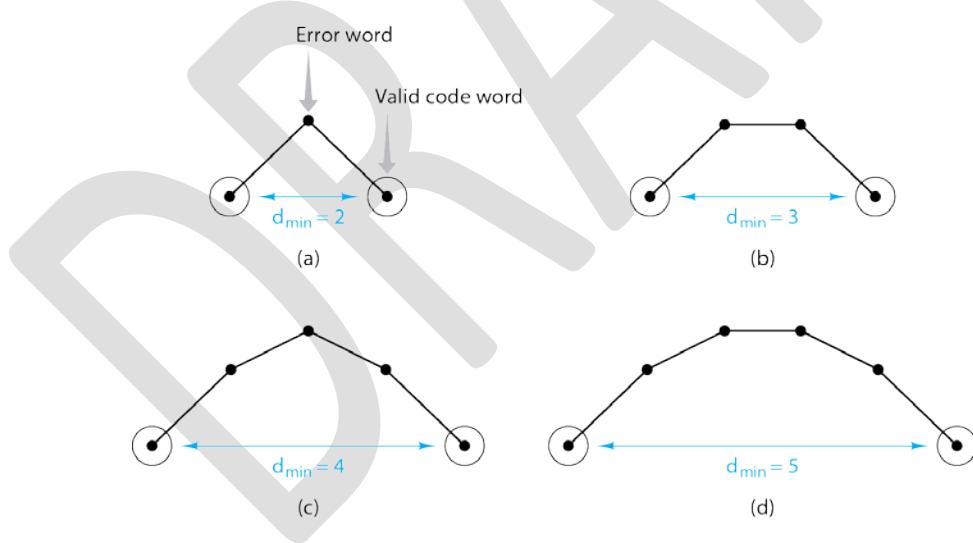


Figure 1.6 Relationship between the minimum distance between code words and the ability to detect and correct bit errors. (Connected words differ in exactly one bit position.)
(a) Single-error detection (SED). (b) Single-error correction (SEC) or double-error correction (DED). (c) (SEC and DED) or TED. (d) DEC, (SEC and 3ED), or 4ED.

Simple Parity Codes

Parity codes are formed from a code C by concatenating (\parallel) a parity bit, P , to each code word of C . Figure 1.7 illustrates the concept. In an *odd-parity code*, the parity bit is specified to be either 0 or 1 as necessary for $w(P|C)$ to be odd. The parity bit of an *even-parity code* is selected so that $w(P|C)$ will be even. Figure 1.8 shows how parity encoding is used on a nine-track magnetic tape.

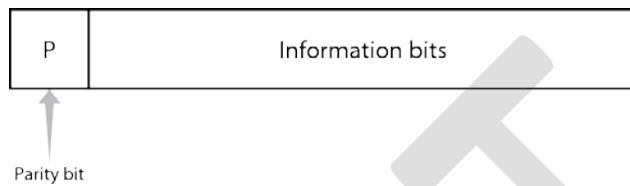


Figure 1.7 Parity-coded information.

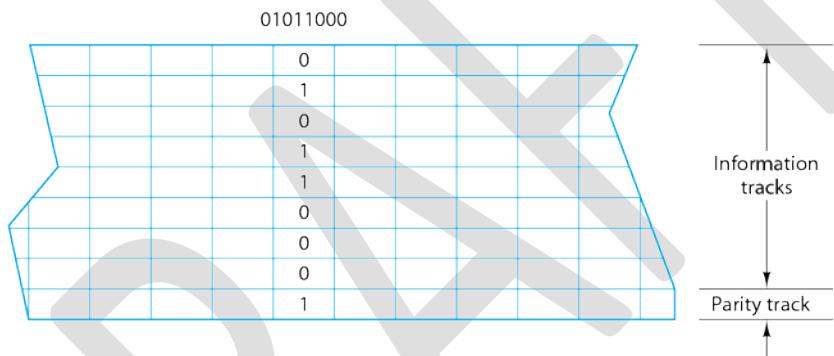


Figure 1.8 Parity coding on magnetic tape.

Concatenate a parity bit to the 7-bit ASCII code of the characters 0, X, =, and BEL to produce an odd parity code.

Character	ASCII Code	Odd-Parity Encoding
0	0110000	10110000
X	1011000	01011000
=	0111100	10111100
BEL	0000111	00000111

Encode the message CATCH 22 in ASCII code with even parity and group the coded word into 16-bit segments.

Segment 1: $(\underbrace{11000011}_{C} \underbrace{01101000001}_{A})_{ASCII}$

Segment 2: $(\underbrace{1101010011}_{T} \underbrace{000011}_{C})_{ASCII}$

Segment 3: $(\underbrace{01001000}_H \underbrace{10100000}_{\text{blank}})_{\text{ASCII}}$

Segment 4: $(\underbrace{10110010}_2 \underbrace{10110010}_2)_{\text{ASCII}}$

Note that this message can be stored in two memory words of a 32-bit computer as

Word X : 11000011010000011101010011000011
 Word $X+1$: 01001000101000001011001010110010

Error detection on parity-encoded information is easily accomplished by checking to see if a code word has the correct parity. For example, if the parity of an odd-parity code word is actually even, then a detectable error has occurred. It is easy to build logic circuits to detect parity, as will be seen later in the text.

Parity codes are minimum-distance-2 codes and thus can be used to detect single errors. In fact, they can be used to detect any odd number of errors since such errors will always change the parity of the code word. On the other hand, errors in an even number of bits will not change the parity and are therefore not detectable using a parity code.

Two-out-of-Five Code

The *two-out-of-five code* is an error detection code having exactly 2 bits equal to 1 and 3 bits equal to 0 in each code word and is representative of m -out-of- n codes. Error detection is accomplished by counting the number of ones in a code word. An error is indicated anytime the number of ones is not exactly equal to 2. It follows that two-out-of-five codes permit the detection of single errors as well as multiple errors in adjacent bits. Table 1.12 presents a two-out-of-five code for the decimal digits.

TABLE 1.12 TWO-OUT-OF-FIVE CODE FOR DECIMAL DIGITS

Decimal Digit	Two-Out-Of-Five Code
0	00011
1	00101
2	01001
3	10001
4	00110
5	01010
6	10010
7	01100
8	10100
9	11000

Hamming Codes

Hamming codes are an important class of error-detection and correction codes that can be considered an extension of parity codes where several parity bits (check bits) are defined, each across a different subset of information bits in such a manner to establish the error detection and correction characteristics of the code. The subsets overlap in such a manner that each information bit is in at least two subsets. *Single-error-correcting (SEC)* codes permit the detection and correction of any single-bit error. *Single-error-correcting/double-error-detecting (SEC/DED)* codes provide the detection but not correction of any double error, in addition to single-error detection and correction.

The error detection and correction properties of a Hamming code are determined by the number of check bits used and how the check bits are defined over the information bits. The minimum distance d_{min} is equal to the weight of the minimum-weight nonzero code word. In other words, d_{min} is equal to the number of ones in the codeword with the fewest ones. It is beyond the scope of this book to discuss the design of Hamming codes in depth. However, the two Hamming codes given in Table 1.13 will be used to illustrate code properties.

TABLE 1.13 TWO HAMMING CODES FOR FOUR-BIT INFORMATION WORDS

Information Words ($i_3i_2i_1i_0$)	Hamming Code 1 ($i_3i_2i_1i_0c_2c_1c_0$)	Hamming Code 2 ($i_3i_2i_1i_0c_3c_2c_1c_0$)
0000	0000000	00000000
0001	0001011	00011011
0010	0010101	00101101
0011	0011110	00110110
0100	0100110	01001110
0101	0101101	01010101
0110	0110011	01100011
0111	0111000	01111000
1000	1000111	10000111
1001	1001100	10011100
1010	1010010	10101010
1011	1011001	10110001
1100	1100001	11001001
1101	1101010	11010010
1110	1110100	11100100
1111	1111111	11111111

Hamming code 1 provides single-error correction but no double-error detection since its minimum distance is 3. This can be seen more clearly in the following analysis. For example, a single error in the leftmost bit of code word 0100110 produces the error word 1100110. Table 1.14 shows the difference and distance between each valid code word and this error

word.

TABLE 1.14 EFFECTS OF ERRORS ON CODE WORDS FOR HAMMING CODE 1

Code Words	Error Word	Difference (Error Pattern)	Distance
0000000	1100110	1100110	4
0001011	1100110	1101101	5
0010101	1100110	1110011	5
0011110	1100110	1111000	4
0100110	1100110	1000000	1
0101101	1100110	1001011	4
0110011	1100110	1010101	4
0111000	1100110	1011110	5
1000111	1100110	0100001	2
1001100	1100110	0101010	3
1010010	1100110	0110100	3
1011001	1100110	0111111	6
1100001	1100110	0000111	3
1101010	1100110	0001100	2
1110100	1100110	0010010	2
1111111	1100110	0011001	3

Note that only the code word in which the error occurred has distance 1 from the error word. This means that no single error in any other code word could have produced this error word. Hence, the detection of the error word 1100110 is equivalent to correcting the error, since the only possible single error that could have produced the pattern is an error in the leftmost bit of code word 0100110.

The preceding analysis also suggests an error detection and correction procedure. That is, we could start by finding the difference between a data word and each possible valid code word. A distance of 0 would indicate a valid match, a distance of 1 would indicate a single error in the corresponding code word in the bit position corresponding to the 1 bit of the difference, and a distance of 2 or more over all code words would indicate a multiple error. While this procedure works in theory, it would not be practical for codes with a large number of code words.

Our analysis also reveals that several code words are distance 2 from the error word. Hence, a double error in each of these words could produce the same error word as the single error (examine Figure 1.6). This implies that double errors cannot in general be detected with this code. Single-error correction in conjunction with double-error detection requires a minimum-distance-4 code as illustrated later with Hamming code 2..

The check bits of Hamming code 1 were defined to provide even parity over a subset of the information bits, as follows.

$$\begin{aligned} c_2: & \quad i_3, \quad i_2, \quad i_1 \\ c_1: & \quad i_3, \quad i_2, \quad i_0 \end{aligned}$$

$$c_0 : \quad i_3, \quad i_1, \quad i_0$$

Decoding a Hamming code word is equivalent to computing the parity of each check bit bit and the corresponding information bits. The concatenated result produces a vector known as the *syndrome* and is illustrated in Table 1.15 for code word 0100110 for cases of no error, single errors, and selected double and triple errors.

TABLE 1.15 SYNDROMES FOR CODE WORD 0100110 FOR HAMMING CODE 1

Code Vector ($i_3i_2i_1i_0c_2c_1c_0$)	Error Pattern ($e_7e_6e_5e_4e_3e_2e_1$)	Syndrome ($s_2s_1s_0$)	Syndrome Interpretation
0100110	0000000	000	No Error
0100111	0000001	001	Error in c_0
0100101	0000010	010	Error in c_1
0100010	0000100	011	Error in c_2
0101110	0001000	100	Error in i_0
0110110	0010000	101	Error in i_1
0000110	0100000	110	Error in i_2
1100110	1000000	111	Error in i_3
1000110	1100000	001	Error in c_0
0101111	0001001	010	Error in c_1
0010110	0110000	011	Error in c_2
0110111	0010001	100	Error in i_0
0100011	0000101	101	Error in i_1
0111110	0011000	110	Error in i_2
0000111	0100001	111	Error in i_3
0011110	0111000	000	No Error

Several characteristics of SEC Hamming codes are illustrated in Table 1.15. First the syndrome for a given code vector indicates either no error or the erroneous bit. Knowing a bit is in error allows it to be corrected. However, double or triple errors will generate the same syndrome as either no error or a single error since the code is not capable of double or triple error detection or correction. As a result, SEC codes become less useful as the likelihood for multiple errors increases. Also, note that the *error pattern* vector can be used to generate a syndrome rather than having to use the actual code word.

Hamming code 2 employs four check bits and has a minimum distance of four. Hence the code has both single-error correction and double-error detection properties meaning that double errors will produce syndromes that are distinct from no error and single-error cases. Additional characteristics of this code will be studied in problems at the end of the chapter.

1.6 Summary and Review Questions

This chapter provided an introduction to number systems and computer codes that are used in computers and other digital systems. The reader should now be familiar with decimal, binary, octal, and hexadecimal number systems and be able to convert numbers from any one of these bases to any other. Moreover, the reader should be able to perform arithmetic operations in decimal, binary, and hexadecimal and should understand how negative numbers are represented in computers. Also, familiarity with fixed-point and floating-point numbers should have been gained. An understanding of binary coded decimal (BCD) and ASCII character codes should have been obtained. Gray codes and excess or biased codes have also been introduced. Finally, a general knowledge of simple error detection and correction codes should have been obtained. The following review questions will help the reader assess her/his level of understanding.

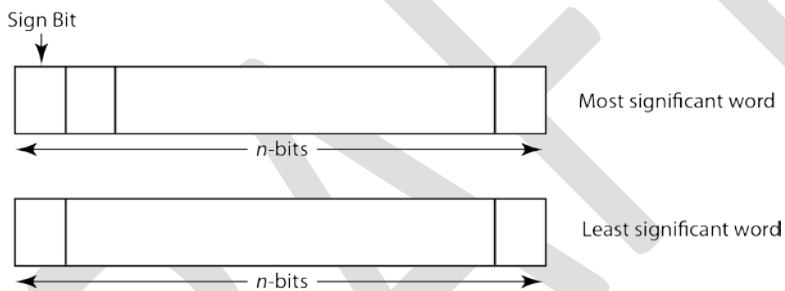
1. Can you interpret and write integers, fractions, and mixed numbers in base 10, 2, 8, or 16?
2. Can you write numbers in positional and polynomial forms?
3. Can you perform basic arithmetic operations (addition, subtraction, multiplication, and division) in decimal, binary, and hexadecimal?
4. Can you convert numbers from one base to another for decimal, binary, octal, and hexadecimal?
5. Do you understand how to represent positive and negative numbers in sign magnitude, radix complement, and diminished-radix complement systems?
6. Can you add and subtract positive and negative binary numbers in a sign-magnitude or a two's complement number system?
7. Do you understand the concept of overflow and be able to detect when an overflow has occurred following an addition or subtraction operation?
8. Do you understand how fixed-point and floating-point numbers are represented in computers?
9. Do you understand the binary coded decimal (BCD) code and how to use it to represent decimal numbers in a computer or other digital systems?
10. Do you understand the American Standard Code for Information Interchange (ASCII) code and how to use it to represent character strings in a computer or other digital system?
11. Do you understand the concept of a Gray code and be able to encode a given set of characters using a Gray code?
12. Do you understand the basics of error detection and correction codes such as error, weight, and distance?
13. Are you able to decode and encode for an even or odd parity code?
14. Do you understand the basic principles of Hamming codes?

1.7 Collaboration Exercises

1. Develop a table like Table 1.6 assuming that A and B are both negative numbers.
2. Discuss the pros and cons of two's complement versus one's complement number systems.
3. Develop a table like Table 1.10 for the 8-bit ASCII code. Use hexadecimal for labeling the rows and columns of the table. Encode the following sentence with the 8-bit code.

My name is your first name.

4. Discuss how emoticons are represented in a computer.
5. Explore how fixed-point and floating-point numbers are represented in current model microprocessors, such as, the Intel Core i7, the ARM Cortex-A9, and the AMD Athlon.
6. Double-word fixed-point numbers are used in some computers. Assuming the format below, what is the range of numbers that can be represented for fixed-point integers? Assume that negative numbers are represented in two's complement. Repeat for fixed-point fractions.



7. Discuss the pros and cons of fixed-point versus floating-point for representing numbers.
8. Design a 6-bit gray code. How many disk-sectors can be encoded with this code?
9. Discuss the error-detection and error-correction properties of 2-out-of-5 codes in the context of Eg. 1.25.
10. Discuss the pros and cons of using parity codes and error-correcting codes.

Problems

- 1.1 Calculate $A + B$, $A - B$, $A \times B$, and $A \div B$ for the following pairs of binary numbers.
- (a) 10101, 1011
 - (b) 1011010, 101111
 - (c) 101, 1011
 - (d) 10110110, 01011011
 - (e) 1101011, 1010
 - (f) 1010101, 101010
 - (g) 10000, 1001
 - (h) 1011.0101, 110.11
- 1.2 Calculate $A + B$, $A - B$, $A \times B$, and $A \div B$ for the following pairs of octal numbers.
- (a) 372, 156
 - (b) 704, 230
 - (c) 1000, 777
 - (d) 423, 651
- 1.3 Calculate $A + B$, $A - B$, $A \times B$, and $A \div B$ for the following pairs of hexadecimal numbers.
- (a) 2CF3, 2B
 - (b) FFFF, 1000
 - (c) 9A5, D17
 - (d) 372, 156
- 1.4 Convert each of the following decimal numbers to binary, octal, and hexadecimal numbers.
- (a) 27
 - (b) 915
 - (c) 0.375
 - (d) 0.65
 - (e) 174.25
 - (f) 250.8
- 1.5 Convert each of the following binary numbers to octal, hexadecimal, and decimal numbers using the most appropriate conversion method.
- (a) 1101
 - (b) 101110
 - (c) 0.101
 - (d) 0.01101
 - (e) 10101.11
 - (f) 10110110.001

1.6 Convert each of the following octal numbers to binary, hexadecimal, and decimal using the most appropriate conversion method.

- (a) 65
- (b) 371
- (c) 240.51
- (d) 2000
- (e) 111111
- (f) 177777

1.7 Convert each of the following hexadecimal numbers to binary, octal, and decimal using the most appropriate conversion method.

- (a) 4F
- (b) ABC
- (c) F8.A7
- (d) 2000
- (e) 201.4
- (f) 3D65E

1.8 Find the two's complement of each of the following binary numbers assuming $n = 8$.

- (a) 101010
- (b) 1101011
- (c) 0
- (d) 11111111
- (e) 10000000
- (f) 11000

1.9 Find the one's complement of each of the following binary numbers assuming $n = 8$.

- (a) 110101
- (b) 1010011
- (c) 0
- (d) 10000000
- (e) 100001
- (f) 01111111

1.10 Calculate $A + B$, $A - B$, $-A + B$, and $-A - B$ for each of the following pairs of numbers assuming a two's complement number system and $n = 8$. Check your results by decimal arithmetic. Explain any unusual results.

- (a) 1010101, 1010
- (b) 1101011, 0101010
- (c) 11101010, 101111
- (d) 10000000, 01111111

1.11 Repeat Problem 1.10 for the following numbers using a one's complement number system.

- (a) 101011, 1101
- (b) 10111010, 11010
- (c) 1010101, 0101010
- (d) 10000000, 01111111

1.12 Show how a 16-bit computer using a two's complement number system would perform the following computations.

- (a) $(16850)_{10} + (2925)_{10} = (?)_{10}$
- (b) $(16850)_{10} - (2925)_{10} = (?)_{10}$
- (c) $(2925)_{10} - (16850)_{10} = (?)_{10}$
- (d) $-(2925)_{10} - (16850)_{10} = (?)_{10}$

1.13 Show how each of the following numbers would be represented as floating-point numbers in IEEE 754-2008 Binary-32 and Binary-64 formats.

- (a) 1101
- (b) 101110
- (c) 0.101
- (d) 0.01101
- (e) 10101.11
- (f) 10110110.001

1.14 Encode each of the following numbers in BCD.

- (a) 39
- (b) 1950
- (c) 94704
- (d) 625

1.15 Encode each of the following character strings in ASCII code. Represent the encoded strings by hexadecimal numbers.

- (a) 1980
- (b) A = b + C
- (c) COMPUTER ENGINEERING
- (d) The End.

1.16 Repeat problem 1.15 assuming that a parity bit is concatenated to the ASCII codes to yield an odd parity code.

1.17 Define a 4-bit code for representing the decimal digits that has the property that the code words for any two digits whose difference is 1 differ in only one bit position and that this property also holds for the digits 0 and 9.

1.18 How many bit errors can be detected in a two-out-of-five code? How many errors, if any, can be corrected in a two-out-of-five code? Prove your answers mathematically.

1.19 Examine the Gray-coded disk of Fig 1.5. Suppose the display lights give the following indications: *A* is off, *B* is on, *C* is on, and *D* is flickering on and off. Locate the position of the disk by sector numbers.

1.20 For the nine-track magnetic tape of Fig 1.8, the following 8-bit messages are to be recorded. Determine the parity bit to establish odd parity for each message.

- (a) P10111010
- (b) P00111000
- (c) P10011001
- (d) P01011010

1.21 Assume that the following are code words from an eight-bit even parity code. Which code words contain errors?

- (a) 00000000
- (b) 00110100
- (c) 01010101
- (d) 10111010

1.22 Develop a table that shows how check bits are defined across the information bits for Hamming Code 2 in Table 1.13.

1.23 Develop a syndrome table for Hamming Code 2 that covers the cases of no error, single errors, and double errors that involved information bit i_3 .

1.24 Find examples that show Hamming Code 2 is not double-error correcting nor triple-error detecting.

1.25 The following defines the check bits of a Hamming code for encoding eight-bit information words ($i_7i_6i_5i_4i_3i_2i_1i_0$).

$$\begin{aligned}C_3: & i_7, i_6, i_5, i_3, i_2 \\C_2: & i_7, i_6, i_4, i_3, i_1 \\C_1: & i_7, i_5, i_4, i_2, i_0 \\C_0: & i_6, i_5, i_4, i_1, i_0\end{aligned}$$

- (a) How many information words are there?
- (b) How many code words?
- (c) Encode information words 00000000, 01010101, 10101010, 11111111.
- (d) What are the error detection and correction properties of this code?
- (e) Develop a syndrome table that covers no error and single-error cases.

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, and Irwin

Chapter 2 Logic Circuits and Boolean Algebra

(7/31/2016)

Table of Contents

2.0 Learning Objectives.....	1
2.1 Logic Gates and Logic Circuits	1
2.1.1 Truth Tables	2
2.1.2 Basic Logic Gates	3
2.1.3 Combinational Logic Circuits.....	6
2.1.4 Sequential Logic Circuits.....	9
2.2 Hardware Description Languages	10
2.2.1 Verilog	11
2.2.2 VHDL	13
2.3 Boolean Algebra.....	15
2.3.1 Postulates and Fundamental Theorems	15
2.3.2 Boolean Functions and Logic Equations	23
2.3.3 Minterms, Maxterms, and Canonical Forms.....	24
2.3.4 Incompletely Specified Functions (Don't Cares)	27
2.4 Minimization of Logic Expressions	29
2.4.1 Minimization Goals and Methods	29
2.4.2 Karnaugh Maps (K-Maps)	32
2.4.3 Minimization of Logic Expressions Using K-Maps	41
2.4.4 Quine-McCluskey Method	60
2.5 Summary and Review Questions	67
2.6 Collaboration Exercises	68
Problems.....	69

This page intentionally left blank.

Chapter 2 Logic Circuits and Boolean Algebra

(4/27/2016)

2.0 Learning Objectives

Students successfully completing this chapter will have demonstrated the knowledge and/or skills necessary to accomplish the following.

1. Understand the concepts of logic variables and functions.
2. Understand the functionality of basic logic gates and combinational circuits.
3. Understand the basic models for combinational and sequential logic circuits.
4. Define and represent logic functions by truth tables, algebraic expressions, circuit diagrams, and hardware description languages.
5. Derive canonical and factored forms of algebraic expressions using the fundamental postulates and theorems of Boolean algebra.
6. Derive minimum forms of algebraic expressions using Boolean algebra, Karnaugh maps, and/or the Quine-McCluskey procedure.

2.1 Logic Gates and Logic Circuits

The primary focus of this chapter is an introduction to the basic tools for logic circuit design and the underlying mathematical principles. The subject itself is important, but also provides the foundation for subsequent more advanced concepts discussed throughout the text. Material in this chapter is presented independent of any specific electronic technology to be used in the realization of logic gates or circuits. Let's begin with an introduction to basic logic gates and circuits as a way to motivate and explain the need for Boolean Algebra and its utility in logic circuit analysis and design.

A logic circuit can be envisioned as a system with binary inputs, x_1, \dots, x_n , and binary outputs, z_1, \dots, z_m , where x_i and $z_j = 0$ or 1 , as illustrated in Figure 2.1 below. Each of the m outputs is a logic function, f , of the n inputs. That is $z_i = f_i(x_1, \dots, x_n)$. There are two classes of logic circuits, combinational and sequential, which will be discussed in more detail below and later in the book.

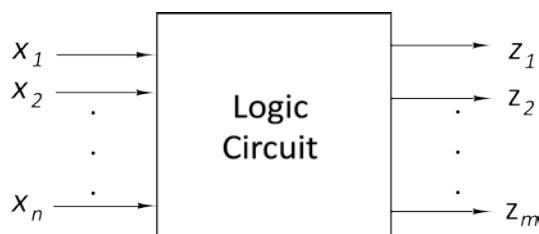


Figure 2.1 – Logic Circuit Block Diagram.

2.1.1 Truth Tables

Truth tables are convenient and commonly used for specifying or defining combinational logic functions and are illustrated in Figure 2.2 for two, three, and four variable odd-parity functions. Each row represents a specific combination of the input variables and the corresponding value of the function. Note that these truth tables have $2^2 = 4$, $2^3 = 8$, and $2^4 = 16$ rows, respectively. In general, an n variable truth table will have 2^n rows.

x_2	x_1	f_{odd}
0	0	0
0	1	1
1	0	1
1	1	0

(a) Two Variable

x_3	x_2	x_1	f_{odd}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(b) Three Variable

x_4	x_3	x_2	x_1	f_{odd}
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(c) Four Variable

Figure 2.2 – Truth Tables of Two, Three, and Four Variable Odd-Parity Functions.

2.1.2 Basic Logic Gates

The basic building blocks of logic circuits are referred to as *logic gates* and correspond to the formal logical operators AND (\wedge), OR (\vee), NOT ($'$), or combinations thereof. Each type of gate is described below in terms of truth tables, logic symbol, and logic equations. Note that in logic circuit literature the notations \cdot and $+$ are usually preferred over \wedge and \vee , respectively, and are used throughout this book to denote AND and OR. Formal logic often uses the terms *TRUE* (T) and *FALSE* (F) for 1 and 0 , respectively.

AND Gates – Let x and y be the inputs of an AND gate and z be the output. Then $z = 1$ iff $x = y = 1$. Otherwise, $z = 0$. The truth table, logic symbol, and logic equation for a two-input AND gate are illustrated in Figure 2.3.

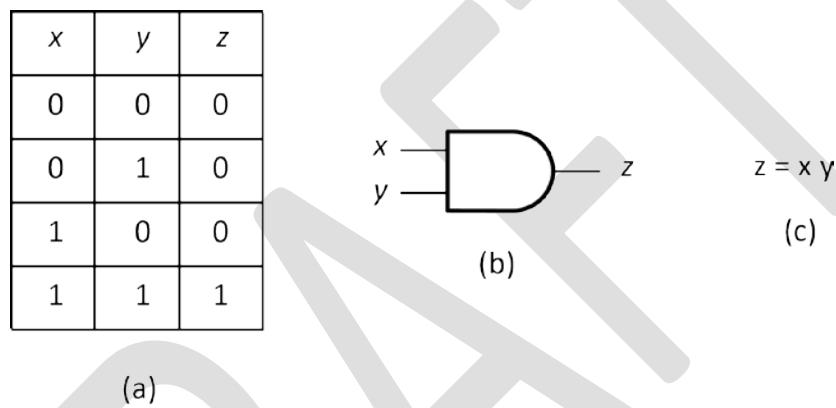


Figure 2.3 – Two-Input AND Gate. (a) Truth Table. (b) Logic Symbol. (c) Logic Equation.

OR Gates – Let x and y be the inputs of an OR gate with output z . Then $z = 1$ iff $x = 1$ or $y = 1$. Otherwise, $z = 0$. The truth table, logic symbol, and logic equation for a two-input OR gate are illustrated in Figure 2.4.

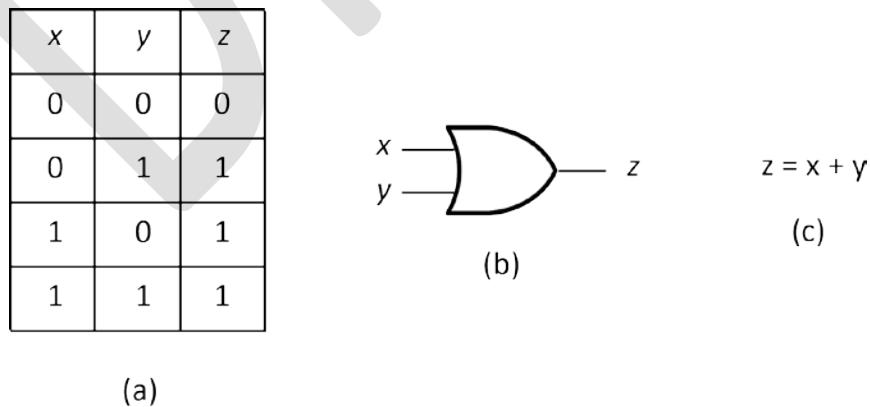


Figure 2.4 – Two-Input OR Gate. (a) Truth Table. (b) Logic Symbol. (c) Logic Equation.

NOT Gates – A NOT gate has one input, x , and one output, z . The output is 1 if the input is 0 and 0 if the input is 1. NOT gates are often referred to as *inverters* since their output value is always the inversion of its input value. The truth table, symbol, and logic equation for a NOT gate are illustrated in Figure 2.5.

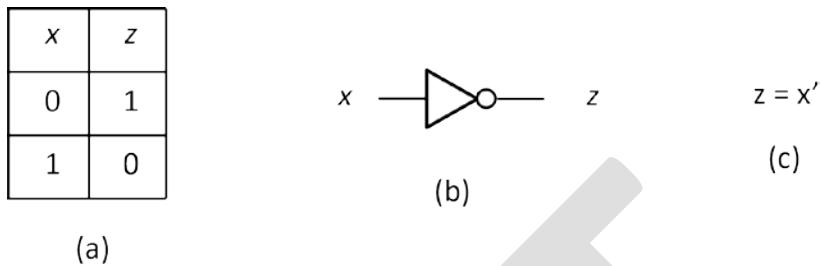


Figure 2.5 – NOT Gate. (a) Truth Table. (b) Logic Symbol. (c) Logic Equation.

AND, OR, and NOT gates are the most fundamental set of logic elements, and any combinational logic function can be described, or realized, as an interconnection of these gates. This property is called *functional completeness* and will be discussed in more detail later in this chapter. In practice, NAND, NOR, XOR, and XNOR gates are widely used in the realization of combinational logic functions and are described below. Each of these gates is actually a combination of two or more of the fundamental gates.

NAND Gates – Let x and y be the inputs of a NAND gate with output z . Then $z = 1$ iff $x = 0$ or $y = 0$. Otherwise, $z = 0$. A NAND gate is the equivalent of an AND gate connected to a NOT as shown in Figure 2.6 (a). The truth table, logic symbol, and logic equation for a two-input NAND gate are illustrated in Figure 2.6 (b) through (d).

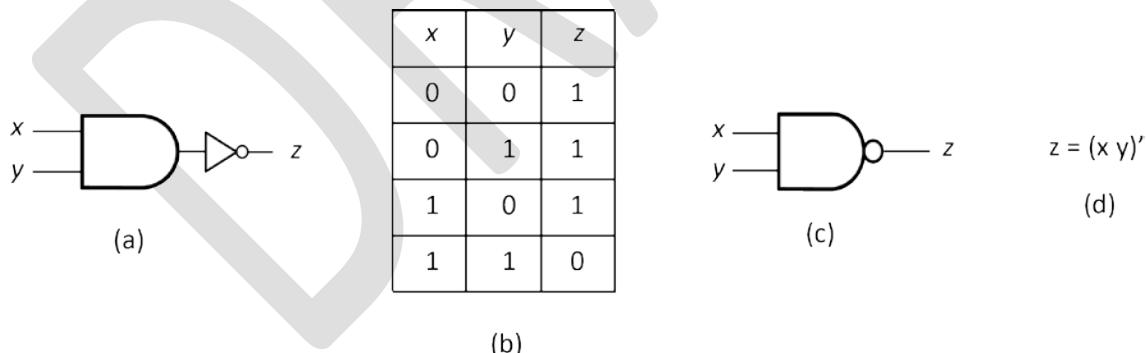


Figure 2.6 – Two-Input NAND Gate. (a) AND-NOT. (b) Truth Table (c) Logic Symbol. (d) Logic Equation.

NOR Gates – Let x and y be the inputs of a NOR gate with output z . Then $z = 1$ iff $x = y = 0$. Otherwise, $z = 0$. A NOR gate is the equivalent of an OR gate connected to a NOT as shown in Figure 2.7 (a). The truth table, logic symbol, and logic equation for a two-input NOR gate are illustrated in Figure 2.7 (b) through (d).

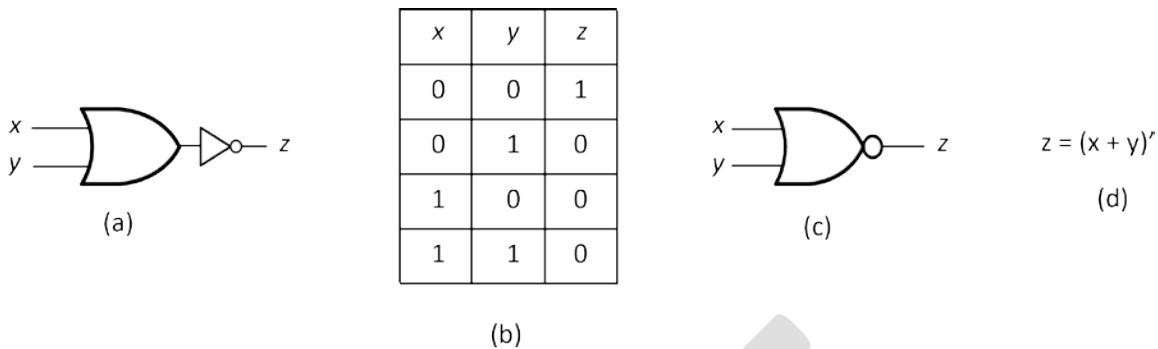


Figure 2.7 – Two-Input NOR Gate. (a) OR-NOT (b) Truth Table. (c) Logic Symbol. (d) Logic Equation.

XOR Gates – Let x and y be the inputs of a XOR gate with output z . Then $z = 1$ iff $x = 0$ and $y = 1$, or $x = 1$ and $y = 0$. Otherwise, $z = 0$. Another way to define an XOR gate is that $z = 1$ iff that $x = 1$ or $y = 1$, exclusively. Hence, XOR is referred to as an EXCLUSIVE-OR. An XOR gate is the equivalent of a network of AND, OR, and NOT gates as shown in Figure 2.8 (a). The truth table, logic symbol, and logic equation for a two-input XOR gate are illustrated in Figure 2.8 (b) through (d).

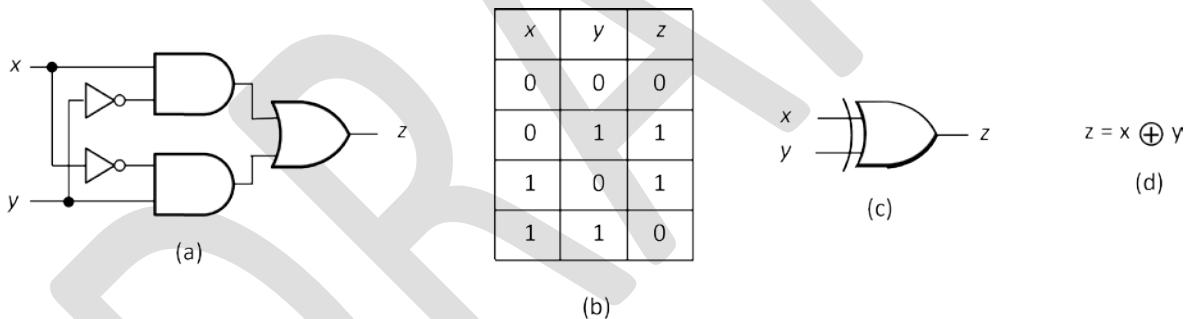


Figure 2.8 – Two-Input XOR Gate. (a) Logic Circuit. (b) Truth Table. (c) Logic Symbol. (d) Logic Equation.

XNOR Gates – Let x and y be the inputs of a XNOR gate with output z . Then $z = 0$ iff $x = 0$ and $y = 1$, or $x = 1$ and $y = 0$. Otherwise, $z = 1$. An XNOR gate is the equivalent of an XOR gate connected to a NOT gate as shown in Figure 2.9 (a). XNOR is sometime called the COINCIDENCE function or gate since its output is 1 when the input values coincide, i.e., both equal 1 or both equal 0. The truth table, logic symbol, and logic equation for a two-input XNOR gate are illustrated in Figure 2.9 (b) through (d).

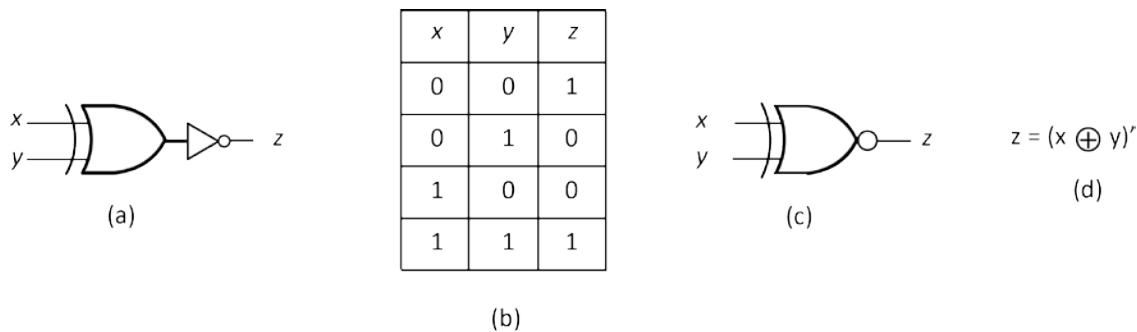


Figure 2.9 – Two-Input XNOR Gate. (a) XOR-NOT. (b) Truth Table. (c) Logic Symbol. (d) Logic Equation.

2.1.3 Combinational Logic Circuits

Combinational logic circuits consist of interconnections, or networks, of logic gates and can be depicted in block diagram form as shown in Figure 2.10. Hence, given binary input variables x_1, \dots, x_n and binary outputs z_1, \dots, z_m , the outputs are logic functions, f_1, \dots, f_m of the inputs as denoted below in Equation 2.1.

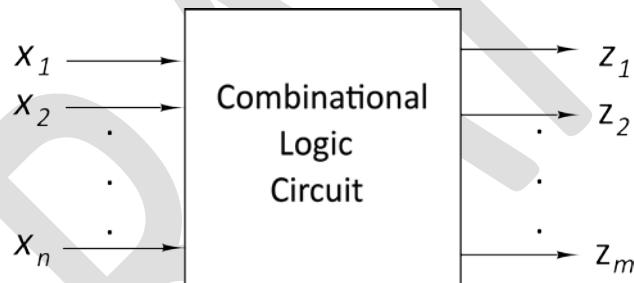


Figure 2.10 – Combinational Logic Circuit Block Diagram.

$$z_i = f_i(x_1, \dots, x_n) \text{ for } i = 1, \dots, m \quad (2.1)$$

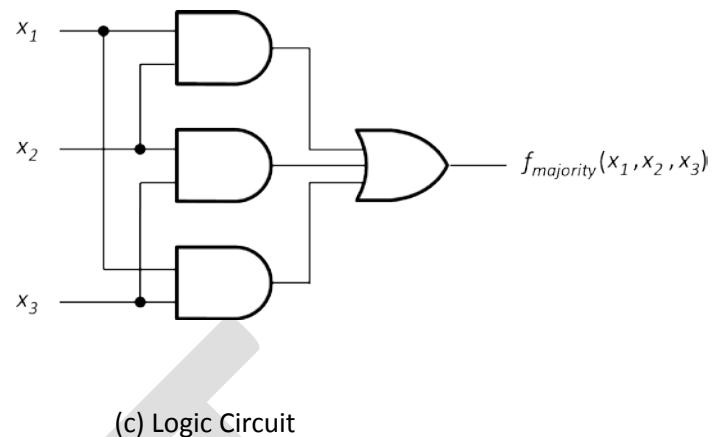
These functions can be specified, or defined, by circuit (schematic) diagrams, logic equations, and/or truth tables in a way similar to how logic gates were defined previously. Functions can also be specified using *hardware description languages (HDLs)* which will be introduced later in this chapter.

Let's take a look at some simple examples that both illustrate the concept of combinational logic circuits and introduce some logic functions that frequently appear in practical applications.

Majority functions – A majority function of n variables, $f_{\text{majority}}(x_1, \dots, x_n) = 1$ when a simple majority of the input variables are 1, $x_i = 1$, and is 0 otherwise. Figure 2.11 illustrates a truth table, logic equation, and logic circuit for a three-input ($n=3$) majority function.

x_3	x_2	x_1	$f_{majority}$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth Table



(c) Logic Circuit

$$f_{majority}(x_1, x_2, x_3) = x_1 x_2 + x_1 x_3 + x_2 x_3$$

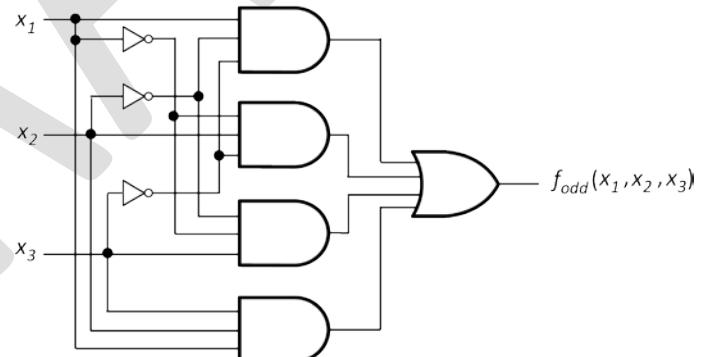
(b) Logic Equation

Figure 2.11 Three-Input Majority Function.

Parity functions – Parity functions can be odd parity or even parity. An odd-parity function, $f_{odd}(x_1, \dots, x_n) = 1$ when an odd number of the input variables are 1, and is 0 otherwise. An even-parity function, $f_{even}(x_1, \dots, x_n) = 1$ when an even number of the input variables are 1, and 0 otherwise. The truth table of f_{odd} for three-variables is shown in Figure 2.2 (b). Figure 2.12 shows a logic equation and logic circuit for the function.

$$f_{odd}(x_1, x_2, x_3) = x_1 x_2' x_3' + x_1' x_2 x_3' + x_1' x_2' x_3 + x_1 x_2 x_3$$

(a) Logic Equation



(b) Logic Circuit

Figure 2.12 – Three-Input Parity Function.

Note that a 2-input XOR gate realizes a 2-variable odd-parity function and a 2-input XNOR realizes a 2-variable even-parity function. The logic equations for these functions are given below.

$$x_1 \oplus x_2 = x_1' x_2 + x_1 x_2' \quad (2.2)$$

$$x_1 \odot x_2 = x_1' x_2' + x_1 x_2 \quad (2.3)$$

Half Adders and Full Adders – Half-adders and full-adders are one-bit binary adders that can be cascaded to realize n -bit binary adders. Half-adders have two binary inputs, a_i and b_i , that are added to produce sum, s_i , and carry-out, c_{outi} , outputs. Full-adders have three binary inputs, a_i , b_i , and c_{inj} . See Figure 2.13 for block diagrams and truth tables. Figure 2.14 illustrates how half-adders and full-adders can be combined to realize a four-bit adder.

Note that the s_i functions are the XOR and odd-parity functions for the half-adder and full-adder, respectively. This is another illustration that odd-parity is a generalization of XOR. Also note that the c_{outi} functions are AND and majority functions for the half-adder and full-adder, respectively.

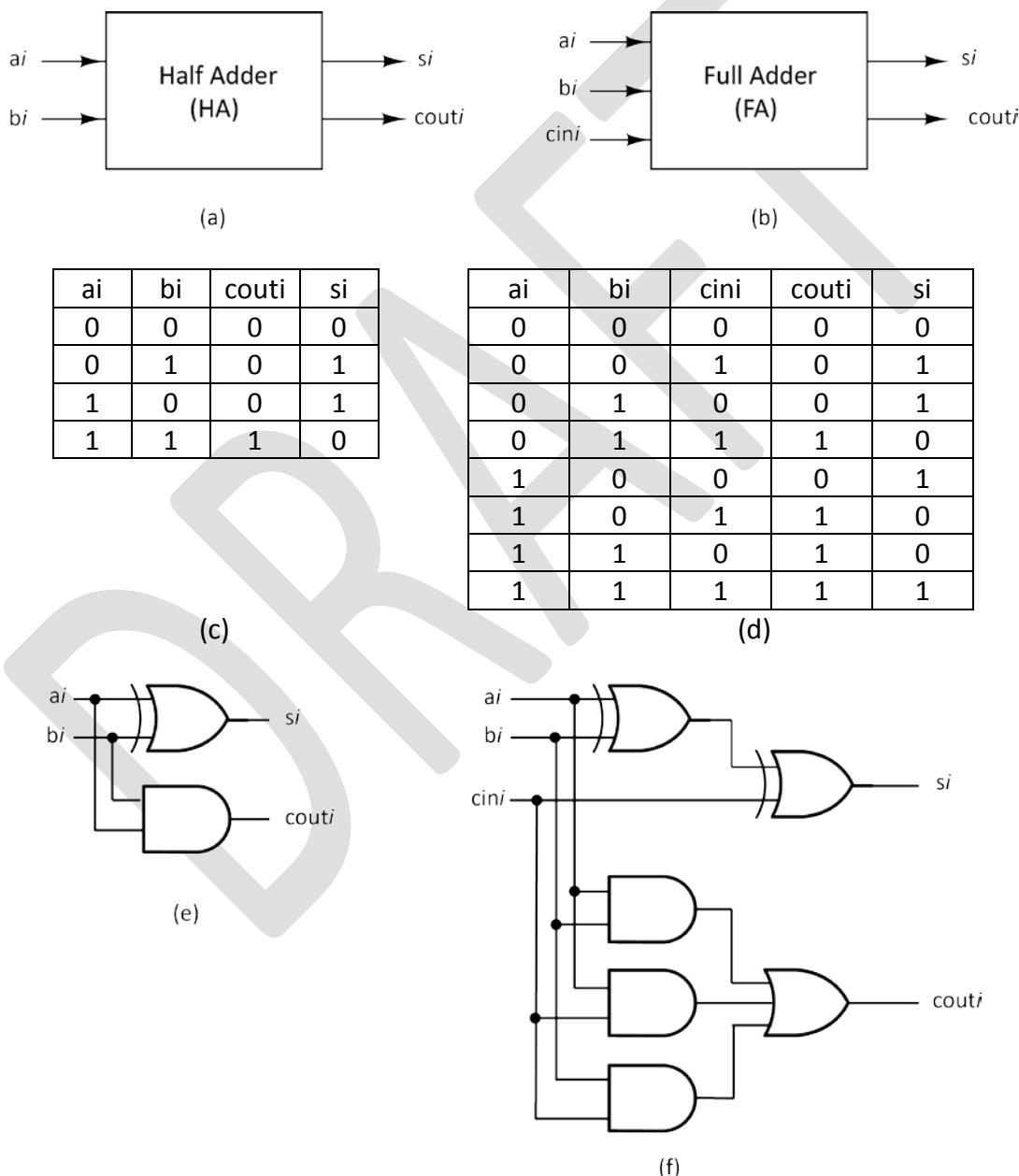


Figure 2.13 – Basic Adders. (a) Half-Adder and (b) Full-Adder Block Diagrams. (c) HA Truth Table. (d) FA Truth Table. (e) HA Circuit. (f) FA Circuit.

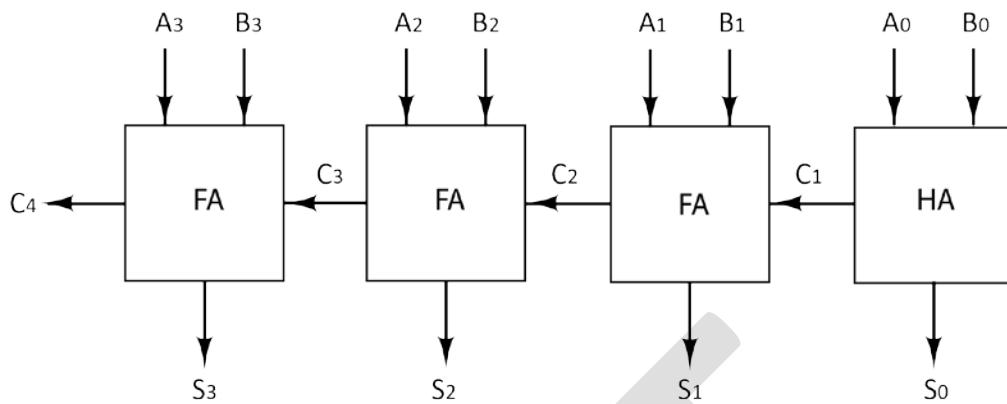


Figure 2.14 – Four-Bit Ripple-Carry Adder.

2.1.4 Sequential Logic Circuits

The output values of a combinational logic circuit can be determined from its truth table, logic equation, circuit diagram, or HDL description if and only if the present input values are known. This implies that such circuits do not possess memory. In other words, the values of previous inputs do not need to be known in order to compute the present output. On the other hand, sequential logic circuits do incorporate memory and will now be introduced.

Sequential logic circuits consist of interconnections or networks of logic gates and memory devices, so they can be modeled as a block of combinational logic with the memory included in feedback loops as depicted in Figure 2.15. Input variables x_1, \dots, x_n and output variables z_1, \dots, z_m are called the *primary inputs* and *outputs*, respectively, of the circuit. Variables y_1, \dots, y_r and Y_1, \dots, Y_r are called *secondary inputs* and *outputs*, respectively, of the circuit.

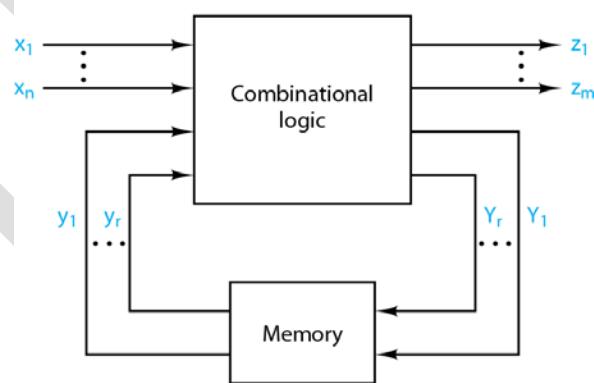


Figure 2.15 – Sequential Logic Circuit Block Diagram.

The input/output relationships in a sequential logic circuit can be described using logic functions as shown in Equations 2.4 and 2.5.

$$z_i = g_i(x_1, \dots, x_n, y_1, \dots, y_r) \text{ where } i = 1, \dots, m \quad (2.4)$$

$$Y_i = h_i(x_1, \dots, x_n, y_1, \dots, y_r) \text{ where } i = 1, \dots, r \quad (2.5)$$

Secondary inputs and outputs are also referred to as *state variables* with y_1, \dots, y_r representing the *present state*, or *state*, and Y_1, \dots, Y_r representing the *next state* of the sequential circuit. So the memory is used to store the state of the circuit while the combinational logic generates the next state and the primary outputs from the state and the primary inputs.

Elevators are sequential devices and can be used to illustrate the concept of inputs, outputs and states. For example, consider the operation of an elevator in a four-story building. The elevator acts as a sequential device because its actions are determined by input signals from its control panels (both on board and on each floor) and its present position at floor 1, 2, 3, or 4. The elevator must in some way “remember” its present position in order to determine its next floor transition. Therefore we define the *present state* of the elevator as a description of its present floor position, including a history of its past floor transitions. For example, the elevator may be “at floor 3 and going up.” This present state must be differentiated from “at floor 3 and going down.” The *next state* (and hence the next floor position) of the elevator is determined by its present state and its *input*, which consists of the inputs from the control buttons on the control panels located in the elevator and on each floor. If the elevator is “at floor 3 and going down,” it will respond to a floor 2 request to go down, but ignore a floor 2 request to go up! Once the next state is determined, a *state transition* is ordered by sending a command to the pulley motor, which drives the elevator to a new floor. The concepts of present state, next state, input, and state transition are fundamental in the study of sequential logic circuits.

There are two fundamental classes of sequential logic circuits based on the method used to initiate, or trigger, state transitions. *Synchronous sequential circuits* use a special input signal, called a *clock*, to trigger the transitions. *Asynchronous sequential circuits* do not rely on a clock, or explicit memory devices, but rather change state as a result of timing delays that ripple through the combinational logic circuitry. Synchronous circuits will be studied in depth in Chapters 4 and 5. Asynchronous circuits are covered in Chapter 6.

2.2 Hardware Description Languages

We have seen how logic functions can be defined and specified with truth tables, algebraic expressions, and/or logic circuit (schematic) diagrams. The latter are representations of the electronic elements or hardware that can be used to implement the logic functions. All of these forms of specification are useful for the analysis and design of logic circuits. However as electronic technologies advanced and became more complex, designers needed tools that allowed higher levels of abstraction to be employed in the definition and specification of logic functions. *Hardware description languages (HDLs)* were developed for this purpose and have become common place in the design of logic circuits and digital systems. HDLs resemble programming languages such as C, C++, and Ada in syntax. However, there are some

distinctions that are worthy of note. HDL code represents the structure and/or behavior of a physical electronic system. As such, statements must usually execute in parallel rather than sequentially. Also, HDLs must be able to handle timing constraints or specifications, describe a system as interconnected components, and incorporate various levels of abstraction.

Verilog and VHDL (VHSIC Hardware Description Language) have become the dominate HDLs used by industry and academe over the past two decades. Both languages will be illustrated below and will be employed throughout the remainder of the book in analysis and design examples. More details on the features and syntax of Verilog and VHDL can be found in appendices B and C, respectively.

2.2.1 Verilog

Verilog was invented in the mid-1980s and has evolved through various upgrades and enhancements into one of the most widely used HDLs. It has been standardized as IEEE 1364 and has syntax similar to the C programming language. Verilog is case sensitive. Verilog models can be synthesized, or compiled, to produce realizations that can be implemented in programmable logic devices such as complex programmable logic devices (CPLDs), field programmable gate arrays (FPGAs), programmable system on chips (PSoC), and application specific integrated circuits (ASICs). Programmable logic devices will be presented in detail later in the book.

Verilog models are a hierarchy of modules with module-to-module communication handled through declared input, output, or bidirectional ports or variables. The following is a Verilog module describing the full adder of Figure 2.13 using structural statements.

```
// Structural description of a full adder
module fulladder (si, couti, ai, bi, cini);
    input ai, bi, cini;
    output si, couti;
    wire d,e,f,g;
    xor (d, ai, bi);
    xor (si, d, cini);
    and (e, ai, bi);
    and (f, ai, cini);
    and (g, bi, cini);
    or (couti, e, f, g);
endmodule
```

The name of the module *fulladder* is followed by the module's input/output variable list. The order of the variables is arbitrary but must be referenced in the specified order thereafter. Variables are specified as input or output using the **input** or **output** keywords, respectively. The keyword **wire** is used to specify internal variables. The inputs and outputs of logical operators **xor**, **and**, and **or** must be listed as output first followed by the inputs. Comment statements may be inserted using the **//**.

ASSIGN statements are available in Verilog and may be used to define the functionality of a module without specifying its structure as illustrated below for the full adder. These are often called *dataflow models*. Verilog also supports behavioral modeling which will be introduced in Chapter 3.

```
// Dataflow description of a full adder
module fulladder (si, couti, ai, bi, cini);
    input ai, bi, cini;
    output si, couti;
    assign si = ai ^ bi ^ cini; // ^ is the XOR operator in Verilog
    assign couti = ai & bi | ai & cini | bi & cini; // & is the AND operator and | is OR
endmodule
```

The ripple carry adder of Figure 2.14 can be described structurally by the following Verilog model using the previously described full adder as a component instantiated four times.

```
//Ripple Carry Adder Structural Model
module RippleCarryAdderStructural (
    input [3:0] A, B,
    output [3:0] S,
    output Cout);
    wire [4:0] C;
    assign C[0] = 1'b0;
    fulladder s0 (S[0], C[1], A[0], B[0], C[0]);
    fulladder s1 (S[1], C[2], A[1], B[1], C[1]);
    fulladder s2 (S[2], C[3], A[2], B[2], C[2]);
    fulladder s3 (S[3], C[4], A[3], B[3], C[3]);
    assign Cout = C[4];
endmodule
```

A Verilog dataflow model of a ripple carry adder can be written as follows.

```
//Ripple Carry Adder Dataflow Model
module RippleCarryAdderDataflow (
    input [3:0] A, B,
    output [3:0] S,
    output Cout);
    wire [4:0] C;
    assign C[0] = 1'b0;
    assign S[0] = A[0]^B[0]^C[0];
    assign C[1] = A[0]&B[0] | A[0]&C[0] | B[0]&C[0];
    assign S[1] = A[1]^B[1]^C[1];
    assign C[2] = A[1]&B[1] | A[1]&C[1] | B[1]&C[1];
    assign S[2] = A[2]^B[2]^C[2];
```

```

assign C[3] = A[2]&B[2] | A[2]&C[2] | B[2]&C[2];
assign S[3] = A[3]^B[3]^C[3];
assign C[4] = A[3]&B[3] | A[3]&C[3] | B[3]&C[3];
assign Cout = C[4];
endmodule

```

The above examples illustrate just a few of the basic features of Verilog. Other basic, and more advanced, features are introduced later in the book. Refer to Appendix B for Verilog fundamentals.

2.2.2 VHDL

VHDL was developed in the mid-to-late 1980's under sponsorship of the United States Department of Defense (DOD). Given DOD's prior support for the development of the Ada programming language, it's not surprising that VHDL is similar in structure and syntax to Ada. VHDL has been standardized as IEEE 1076. VHDL is strongly typed and is not case sensitive. Like Verilog, VHDL can be synthesized or compiled to produce realizations that can be implemented in programmable logic devices. The following examples illustrate VHDL code for the full adders and ripple-carry adders.

```

-- VHDL structural code for a full adder
entity fulladder is
    port (ai, bi, cini: in bit; si, couti: out bit);
end fulladder;

architecture structure of fulladder is
    signal x1, a1, a2, a3: bit; -- internal signal wires
begin
    process (ai, bi, cini)
    begin -- define the schematic diagram
        x1 <= ai xor bi;
        si <= x1 xor cini;
        a1 <= ai and bi;
        a2 <= ai and cini;
        a3 <= bi and cini;
        couti <= a1 or a2 or a3;
    end process;
end structure;

```

```

-- VHDL behavioral code for a full adder
entity fulladder is
    port (ai, bi, cini: in bit; si, couti: out bit);
end fulladder;

```

```

architecture behavior of fulladder is
begin
    process (ai, bi, cini)
    begin
        si <= ai xor bi xor cini;
        couti <= (ai and bi) or (ai and cini) or (bi and cini);
    end process;
end behavior;

```

VHDL can be used to describe a structural model of the ripple-carry adder in Figure 2.14 as shown below. Note that the previously defined fulladder module is declared as a component and then instantiated four times to model the ripple carry adder.

```

--VHDL structural model for a ripple-carry adder
entity ripple_carry_adder is
    port (A, B: in bit_vector (3 downto 0); S: out bit_vector (3 downto 0); Cout: out bit);
end ripple_carry_adder;

architecture behavior of ripple_carry_adder is
    signal C: bit_vector (4 downto 0);
    component fulladder
        port (ai, bi, cini: in bit; si, cout: out bit);
    end component;
begin
    C(0) <= "0";
    s0: fulladder port map (A(0), B(0), C(0), S(0), C(1));
    s1: fulladder port map (A(1), B(1), C(1), S(1), C(2));
    s2: fulladder port map (A(2), B(2), C(2), S(2), C(3));
    s3: fulladder port map (A(3), B(3), C(3), S(3), C(4));
    Cout <= C(4);
end behavior

```

The following is a VHDL behavioral model of the ripple carry adder.

```

--VHDL ripple-carry adder behavioral model
entity ripple_carry_adder is
    port (A, B: in bit_vector (3 downto 0); S: out bit_vector (3 downto 0); Cout: out bit);
end ripple_carry_adder;

```

```

architecture behavior of ripple_carry_adder is
begin
    C(0) <= "0";
    S(0) <= A(0) XOR B(0) XOR C(0);
    C(1) <= (A(0) AND B(0)) OR (A(0) AND C(0)) OR (B(0) AND C(0));

```

```

S(1) <= A(1) XOR B(1) XOR C(1);
C(2) <= (A(1) AND B(1)) OR (A(1) AND C(1)) OR (B(1) AND C(1));
S(2) <= A(2) XOR B(2) XOR C(2);
C(3) <= (A(2) AND B(2)) OR (A(2) AND C(2)) OR (B(2) AND C(2));
S(3) <= A(3) XOR B(3) XOR C(3);
C(4) <= (A(3) AND B(3)) OR (A(3) AND C(3)) OR (B(3) AND C(3));
Cout <= C[4];
end behavior

```

The above examples illustrate a few of the basic features of VHDL and how it compares with Verilog. Other basic, and more advanced, features are introduced later in the book. Refer to Appendix A for VHDL fundamentals.

2.3 Boolean Algebra

Above, truth tables and logic equations have been used to define and represent logic functions and circuits. An understanding of Boolean Algebra is needed in order to fully utilize truth tables and logic equations in the analysis and design of logic circuits. This section will develop that understanding.

In 1847, George Boole presented an algebraic formulation of the processes of logical thought and reason [1] now known as Boolean algebra. Claude Shannon [2] recognized the utility of Boolean Algebra for analyzing and designing networks of electrical relays called switching circuits in 1938. Shannon's version is still used today by logic circuit designers.

2.3.1 Postulates and Fundamental Theorems

The basic description of the Boolean algebra formulation is based on concepts from set theory, in which a *Boolean algebra* is formally defined as a distributive, complemented lattice [2]. We present here a summary of this definition as a set of postulates that summarizes the basic elements and properties of a Boolean algebra.

Postulate 1. Definition A *Boolean algebra* is a closed algebraic system containing a set K of two or more elements and the two operators \cdot and $+$; alternatively, for every a and b in set K , $a \cdot b$ belongs to K and $a + b$ belongs to K ($+$ is called OR and \cdot is called AND).

Postulate 2. Existence of 1 and 0 elements There exist unique elements 1 (one) and 0 (zero) in set K such that for every a in K $a + 0 = a$, $a \cdot 1 = a$, where 0 is the identity element for the $+$ operation and 1 is the identity element for the \cdot operation.

Postulate 3 Commutativity of the $+$ and \cdot operations For every a and b in K $a + b = b + a$, $a \cdot b = b \cdot a$.

Postulate 4 Associativity of the + and · operations For every a, b , and c in K

$$a + (b + c) = (a + b) + c, \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$

Postulate 5 Distributivity of + over · and · over + For every a, b , and c in K

$$a + (b \cdot c) = (a + b) \cdot (a + c), \quad a \cdot (b + c) = (a \cdot b) + (a \cdot c).$$

Postulate 6 Existence of the complement For every a in K there exists a unique element called \bar{a} (*complement* of a) in K such that $a + \bar{a} = 1$, $a \cdot \bar{a} = 0$.

Upon this set of postulates, we may now develop other useful relationships, which we call theorems. To simplify notation in the remainder of the text, the dot (\cdot) will usually be omitted when indicating the \cdot operation as illustrated below.

$$\begin{aligned} a + b \cdot c &= (a + b) \cdot (a + c) \\ a + bc &= (a + b)(a + c) \end{aligned}$$

Before proceeding to theorem development, let us discuss the principle of duality and how it applies in Boolean Algebra.

Duality – The principle of *duality* states that, if an expression is valid in Boolean algebra, then the dual of the expression is also valid. The dual of an expression is found by replacing all $+$ operators with \cdot , all \cdot operators with $+$, all ones with zeros, and all zeros with ones. The principle of duality follows from postulates 2 to 6 since they are each defined as duals.

Example – Find the dual of the expression

$$a + (bc) = (a + b)(a + c)$$

Changing all $+$ operators to \cdot and vice versa, yields the dual expression

$$a(b + c) = ab + ac$$

When deriving a dual, we must be careful not to alter the location of parentheses, if they are present. Note that the two expressions in the last example are parts (a) and (b) of Postulate 5.

The principle of duality will be used extensively in proving Boolean algebra theorems. In fact, once we have employed the postulates and previously proven theorems to demonstrate the validity of an expression, duality is used to prove the validity of the dual expression.

We will now present and prove several useful theorems in Boolean algebra. In these theorems, the letters a, b, c, \dots represent elements of a Boolean algebra. Proofs may

invoke the postulates and previously proven theorems. The first theorem describes the property of idempotency and is proved from only the postulates.

Theorem 1. Idempotency

- (a) $a + a = a$.
- (b) $aa = a$.

Proof. We may prove either part (a) or (b) of this theorem. Suppose we prove part (a). Part (b) will follow from duality.

$$\begin{aligned}
 a + a &= (a + a)1 && [P2(b)] \\
 &= (a + a)(a + \bar{a}) && [P6(a)] \\
 &= a + a\bar{a} && [P5(a)] \\
 &= a + 0 && [P6(b)] \\
 &= a && [P2(a)]
 \end{aligned}$$

The postulates used to justify a particular step are listed to the right. An important point to remember is that symbols on opposite sides of the equal sign may be used interchangeably; for example, Theorem 1 tells us that we may exchange a for $a \cdot a$, and vice versa.

The next theorem further emphasizes the properties of the unique elements 1 and 0.

Theorem 2 Null elements for + and · operators

- (a) $a + 1 = 1$.
- (b) $a \cdot 0 = 0$.

Proof. Let us again prove part (a) of the theorem.

$$\begin{aligned}
 a + 1 &= (a + 1)1 && [P2(b)] \\
 &= 1 \cdot (a + 1) && [P3(b)] \\
 &= (a + \bar{a})(a + 1) && [P6(a)] \\
 &= a + \bar{a} \cdot 1 && [P5(a)] \\
 &= a + \bar{a} && [P2(b)] \\
 &= 1 && [P6(a)]
 \end{aligned}$$

Since part (a) of this theorem is valid, it follows from the principle of duality that part (b) is valid also.

Theorem 3 Involution

$$\bar{\bar{a}} = a.$$

Proof. From Postulate 5, $a \cdot \bar{a} = 0$ and $a + \bar{a} = 1$. Therefore, \bar{a} is the complement of a , and also a is the complement of \bar{a} . Since the complement of \bar{a} is unique, it follows that $\bar{\bar{a}} = a$.

At this point let us use the preceding material to summarize all the properties of the unique elements 1 and 0 in Table 2.1. The · (AND) properties of 1 and 0 remind us of the

fundamental properties of multiplication in standard mathematics; however, the + (OR) properties quickly indicate that we are not dealing with the mathematics we previously have studied, and none of the mathematical properties that we employed there can be assumed for use in Boolean algebra. We may use only the postulates and theorems we are currently developing, since we are now working in a completely new and different system.

Table 2.1 Properties of 0 and 1 Elements

OR	AND	COMPLEMENT
$a + 0 = a$	$a \cdot 0 = 0$	$0' = 1$
$a + 1 = 1$	$a \cdot 1 = a$	$1' = 0$

The Boolean algebra property of absorption is stated in the next theorem. Absorption has no counterpart in "ordinary" algebra.

Theorem 4. Absorption

(a) $a + ab = a$.

(b) $a(a + b) = a$.

Proof. Let us prove part (a).

$$\begin{aligned}
 a + ab &= a \cdot 1 + ab && [P2(b)] \\
 &= a(1 + b) && [P5(b)] \\
 &= a(b + 1) && [P3(b)] \\
 &= a \cdot 1 && [T2(a)] \\
 &= a && [P2(b)]
 \end{aligned}$$

The following examples illustrate the use of theorem 4 to simplify logic expressions.

$$(X + Y) + (X + Y)Z = X + Y \quad [T4(a)]$$

$$A\bar{B}(\bar{A}\bar{B} + \bar{B}C) = A\bar{B} \quad [T4(b)]$$

$$A\bar{B}C + \bar{B} = \bar{B} \quad [T4(a)]$$

The following three theorems are similar to Theorem 4 in that they can be employed to eliminate extra elements from a Boolean expression.

Theorem 5. Absorption

$$a + \bar{a}b = a + b . a(\bar{a} + b) = ab .$$

Proof. Part (a) of the theorem is proved as follows:

$$\begin{aligned}
 a + \bar{a}b &= (a + \bar{a})(a + b) && [P5(a)] \\
 &= 1 \cdot (a + b) && [P6(a)] \\
 &= (a + b) \cdot 1 && [P3(b)] \\
 &= (a + b) && [P2(b)]
 \end{aligned}$$

The following examples illustrate the use of Theorem 5 in simplifying logic expressions.

$$B + A\bar{B}\bar{C}D = B + A\bar{C}D \quad [T5(a)]$$

$$\bar{Y}(X + Y + Z) = \bar{Y}(X + Z) \quad [T5(b)]$$

$$(X + Y)(\overline{(X + Y)} + Z) = (X + Y)Z \quad [T5(b)]$$

$$AB + \overline{(AB)}\bar{C}\bar{D} = AB + \bar{C}\bar{D} \quad [T5(a)]$$

Theorem 6. Absorption

$$ab + a\bar{b} = a. \quad (a + b)(a + \bar{b}) = a.$$

Proof. Part (a) of the theorem is proved as follows:

$$ab + a\bar{b} = a(b + \bar{b}) \quad [P5(b)]$$

$$= a \cdot 1 \quad [P6(a)]$$

$$= a \quad [P2(b)]$$

The following examples illustrate the use of Theorem 6 in simplifying logic expressions.

$$ABC + A\bar{B}C = AC \quad [T6(a)]$$

$$(AD + B + C)(AD + \overline{(B + C)}) = AD \quad [T6(b)]$$

Simplify $(\bar{W} + \bar{X} + \bar{Y} + \bar{Z})(\bar{W} + \bar{X} + \bar{Y} + Z)(\bar{W} + \bar{X} + Y + \bar{Z})(\bar{W} + \bar{X} + Y + Z)$.

$$= (\bar{W} + \bar{X} + \bar{Y})(\bar{W} + \bar{X} + Y + \bar{Z})(\bar{W} + \bar{X} + Y + Z) \quad [T6(b)]$$

$$= (\bar{W} + \bar{X} + \bar{Y})(\bar{W} + \bar{X} + Y) \quad [T6(b)]$$

$$= (\bar{W} + \bar{X}) \quad [T6(b)]$$

Theorem 7. Consensus

$$ab + \bar{a}c + bc = ab + \bar{a}c.$$

$$(a + b)(\bar{a} + c)(b + c) = (a + b)(\bar{a} + c).$$

Proof. $ab + \bar{a}c + bc = ab + \bar{a}c + 1 \cdot bc \quad [P2(b)]$

$$= ab + \bar{a}c + (a + \bar{a})bc \quad [P6(a)]$$

$$= ab + \bar{a}c + abc + \bar{a}bc \quad [P5(b)]$$

$$= (ab + abc) + (\bar{a}c + \bar{a}cb) \quad [P3, P4]$$

$$= ab + \bar{a}c \quad [T4(a)]$$

The key to using this theorem is to find an element and its complement, note the associated terms, and eliminate the included term (the "consensus" term), which is composed of the associated terms.

The consensus theorem is useful both in reducing and expanding expressions as follows.

$$AB + \overline{A}CD + BCD = AB + \overline{A}CD \quad [T7(a)]$$

$$(a + \overline{b})(\overline{a} + c)(\overline{b} + c) = (a + \overline{b})(\overline{a} + c) \quad [T7(b)]$$

We will find later in this chapter that the postulates and theorems form the basis of methods for minimizing logic expressions.

In working with Boolean algebra, we often need to determine the complement of a logic expression. The following theorem provides the basis for this operation.

Theorem 8. DeMorgan's theorem

$$\overline{a+b} = \overline{a} \cdot \overline{b} . \quad \overline{a \cdot b} = \overline{a} + \overline{b} .$$

Proof. Let us prove part (a).

If $X = a + b$, then $\overline{X} = \overline{(a + b)}$. By Postulate 6, $X \cdot \overline{X} = 0$ and $X + \overline{X} = 1$. If $X \cdot Y = 0$ and $X + Y = 1$, then $Y = \overline{X}$ because the complement of X is unique. Therefore, we let $Y = \overline{ab}$ and test $X \cdot Y$ and $X + Y$:

$$\begin{aligned} X \cdot Y &= (a + b)(\overline{ab}) \\ &= (\overline{ab})(a + b) \\ &= (\overline{ab})a + (\overline{ab})b \\ &= a(\overline{ab}) + (\overline{ab})b \\ &= (a\overline{a})\overline{b} + \overline{a}(b\overline{b}) \\ &= 0 \cdot \overline{b} + \overline{a}(b \cdot \overline{b}) \\ &= \overline{b} \cdot 0 + \overline{a} \cdot 0 \\ &= 0 + 0 \\ &= 0 \end{aligned} \quad \begin{array}{l} [P3(b)] \\ [P5(b)] \\ [P3(b)] \\ [P4(b)] \\ [P6(b), P3(b)] \\ [P3(b), P6(b)] \\ [T2(b)] \\ [P2(a)] \end{array}$$

$$\begin{aligned} X + Y &= (a + b) + \overline{ab} \\ &= (b + a) + \overline{ab} \\ &= b + (a + \overline{ab}) \\ &= b + (a + \overline{b}) \\ &= (a + \overline{b}) + b \\ &= a + (\overline{b} + b) \\ &= a + (b + \overline{b}) \\ &= a + 1 \\ &= 1 \end{aligned} \quad \begin{array}{l} [P3(a)] \\ [P4(a)] \\ [T5(a)] \\ [P3(a)] \\ [P4(a)] \\ [P3(a)] \\ [P6(a)] \\ [T2(a)] \end{array}$$

Therefore, by the uniqueness of \overline{X} , $Y = \overline{X}$, and therefore $\overline{ab} = \overline{a+b}$

Theorem 8 may be generalized as follows.

$$\overline{a+b+\dots+z} = \overline{a} \cdot \overline{b} \cdots \overline{z} . \quad \overline{ab\dots z} = \overline{a} + \overline{b} + \cdots + \overline{z} .$$

The rule to follow when complementing an expression is to use relation (a) or (b), replacing each + (OR) operator with an · (AND) operator, and vice versa, and replacing each variable with its complement.

A note of caution is in order here. In applying DeMorgan's theorem, operator precedence must be observed: · takes precedence over +. The following example illustrates this important point.

Example -- Complement the expression $a+bc$.

$$\begin{aligned}\overline{a+b \cdot c} &= \overline{a} + \overline{(b \cdot c)} && [P4] \\ &= \overline{a} \cdot \overline{(b \cdot c)} && [T8(a)] \\ &= \overline{a} \cdot (\overline{b} + \overline{c}) && [T8(b)] \\ &= \overline{a}\overline{b} + \overline{a}\overline{c} && [P5(b)]\end{aligned}$$

Note that: $\overline{a+b \cdot c} \neq \overline{a} \cdot \overline{b} + \overline{c}$

The following examples further illustrate the use of DeMorgan's theorem.

$$\begin{aligned}\overline{X + \overline{Y}} &= \overline{X} \cdot \overline{\overline{Y}} && [T8(a)] \\ &= \overline{X} \cdot Y && [T3]\end{aligned}$$

Example -- Complement the expression $a(b+z(x+\overline{a}))$, and simplify the result so that the only complemented terms are individual variables.

$$\begin{aligned}\overline{a(b+z(x+\overline{a}))} &= \overline{a} + \overline{(b+z(x+\overline{a}))} && [T8(b)] \\ &= \overline{a} + \overline{b} \overline{(z(x+\overline{a}))} && [T8(a)] \\ &= \overline{a} + \overline{b} (\overline{z} + \overline{(x+\overline{a})}) && [T8(b)] \\ &= \overline{a} + \overline{b} (\overline{z} + \overline{x} \cdot \overline{\overline{a}}) && [T8(a)] \\ &= \overline{a} + \overline{b} (\overline{z} + \overline{x}a) && [T3] \\ &= \overline{a} + \overline{b} (\overline{z} + \overline{x}) && [T5(a)]\end{aligned}$$

Example -- Complement the expression $a(b+c) + \overline{ab}$.

$$\begin{aligned}\overline{a(b+c) + \overline{ab}} &= \overline{ab + ac + \overline{ab}} && [P5(b)] \\ &= \overline{b + ac} && [T6(a)] \\ &= \overline{b} \overline{(ac)} && [T8(a)] \\ &= \overline{b} (\overline{a} + \overline{c}) && [T8(b)]\end{aligned}$$

As illustrated by this last example, the process of complementing an expression can often be simplified by reducing the expression prior to applying DeMorgan's theorem.

The following example illustrates how the consensus and DeMorgan's theorems can be used to simplify an expression.

$$\begin{aligned}
 ABC + \bar{A}D + \bar{B}D + CD &= ABC + (\bar{A} + \bar{B})D + CD & [P5(b)] \\
 &= ABC + \bar{A}\bar{B}D + CD & [T8(b)] \\
 &= ABC + \bar{A}\bar{B}D & [T7(a)] \\
 &= ABC + (\bar{A} + \bar{B})D & [T8(b)] \\
 &= ABC + \bar{A}D + \bar{B}D & [P5(b)]
 \end{aligned}$$

DeMorgan's theorem, thusly, presents the general technique for complementing logic expressions. It will be especially useful in manipulating logic expressions into formats suitable for realization with specific types of logic gates.

In each of the preceding examples, an element or expression and its complement offer the key to reducing the expression.

It is important to note that the theorems presented can be quickly demonstrated via Venn diagrams. Hence readers are encouraged to use this graphical picture as an aid in remembering these important theorems. Table 2.2 summarizes the basic postulates and theorems of Boolean algebra.

Table 2.2 Boolean Algebra Postulates and Theorems

Expression	Dual
P2(a): $a + 0 = a$	P2(b): $a \cdot 1 = a$
P3(a): $a + b = b + a$	P3(b): $a \cdot b = b \cdot a$
P4(a): $a + (b + c) = (a + b) + c$	P4(b): $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
P5(a): $a + bc = (a + b)(a + c)$	P5(b): $a \cdot (b + c) = a \cdot b + a \cdot c$
P6(a): $a + a' = 1$	P6(b): $a \cdot a' = 0$
T1(a): $a + a = a$	T1(b): $a \cdot a = a$
T2(a): $a + 1 = 1$	T2(b): $a \cdot 0 = 0$
T3(a): $(a')' = a$	T3(b): $(a')' = a$
T4(a): $a + ab = a$	T4(b): $a(a + b) = a$
T5(a): $a + a'b = a + b$	T5(b): $a(a' + b) = a \cdot b$
T6(a): $ab + ab' = a$	T6(b): $(a + b)(a + b') = a$
T7(a): $ab + a'c + bc = ab + a'c$	T7(b): $(a + b)(a' + c)(b + c) = (a + b)(a' + c)$
T8(a): $(a + b)' = a'b'$	T8(b): $(ab)' = a' + b'$

2.3.2 Boolean (Logic) Functions and Equations

The postulates and theorems of Boolean algebra presented previously are given in general terms without the elements of the set K being specified. Hence, the results are valid for any Boolean algebra. In the discussions that follow, emphasis will focus on the Boolean algebra in which $K = \{0,1\}$.

The concept of a function is well known to those familiar with ordinary algebra. Boolean functions represent the corresponding concept for Boolean algebra and can be defined as follows. Let X_1, X_2, \dots, X_n be symbols called variables, each of which represents either the element 0 or 1 of a Boolean algebra (0 or 1 is said to be the *value* of the variable), and let $f(X_1, X_2, \dots, X_n)$ represent a Boolean function of X_1, X_2, \dots, X_n . The function f represents the value 0 or the value 1 depending on the set of values assigned to X_1, X_2, \dots, X_n . Since there are n variables and each variable has two possible values, there are 2^n ways of assigning these values to the n variables. Furthermore, there are two possible values for the function $f(x_1, x_2, \dots, x_n)$ for each possible assignment. Therefore, there are 2^{2^n} different Boolean functions of n variables.

For the case in which $n = 0$, the two Boolean functions of zero variables are

$$f_0 = 0 \quad f_1 = 1$$

For $n = 1$, the four functions of the variable A are

$$\begin{aligned} f_0 &= 0, & f_2 &= A \\ f_1 &= \bar{A}, & f_3 &= 1 \end{aligned}$$

For $n = 2$, there are 16 functions, f_0, \dots, f_{15} which are defined in Table 2.3.

Table 2.3 All Sixteen Logic (Boolean) Functions of Two Variables

AB	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
10	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
11	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Algebraic expressions for each of these functions can be derived using the following logic equation, for $i = 0, \dots, 15$. Let

$$f_i(A, B) = i_3 AB + i_2 A\bar{B} + i_1 \bar{A}B + i_0 \bar{A}\bar{B} \quad (2.6)$$

where $(i)_{10} = (i_3 i_2 i_1 i_0)_2$ represents the binary values 0000, 0001, 0010, ..., 1111. Successively applying these values to Eq. 2.6 yields the following algebraic expressions for the 16 functions of two variables.

$$f_0(A, B) = 0$$

$$f_1(A, B) = \bar{A}\bar{B}$$

$$f_2(A, B) = \bar{A}B$$

$$\begin{aligned}
 f_3(A, B) &= \overline{A}B + \overline{A}\overline{B} = \overline{A} \\
 f_4(A, B) &= A\overline{B} \\
 f_5(A, B) &= A\overline{B} + \overline{A}\overline{B} = \overline{B} \\
 f_6(A, B) &= A\overline{B} + \overline{A}B \\
 f_7(A, B) &= A\overline{B} + \overline{A}B + \overline{A}\overline{B} = \overline{A} + \overline{B} \\
 f_8(A, B) &= AB \\
 f_9(A, B) &= AB + \overline{A}\overline{B} \\
 f_{10}(A, B) &= AB + \overline{A}B = B \\
 f_{11}(A, B) &= AB + \overline{A}B + \overline{A}\overline{B} = \overline{A} + B \\
 f_{12}(A, B) &= AB + A\overline{B} = A \\
 f_{13}(A, B) &= AB + A\overline{B} + \overline{A}\overline{B} = A + \overline{B} \\
 f_{14}(A, B) &= AB + A\overline{B} + \overline{A}B = A + B \\
 f_{15}(A, B) &= AB + A\overline{B} + \overline{A}B + \overline{A}\overline{B} = 1
 \end{aligned}$$

2.3.3 Minterms, Maxterms, and Canonical Forms

The truth table of a given function is unique. However, different algebraic expressions, and circuit diagrams, may actually represent the same function as seen above. The equivalence of two expressions can be demonstrated by applying the postulates and theorems of Boolean algebra or by constructing and comparing their truth tables.

Given that, in general, algebraic forms of functions are not unique, it is, however, possible to define standard forms that are unique for a given function. We will now examine two such forms, the standard or canonical sum of products (CSOP) and the standard or canonical product of sums (CPOS) and show how each relates to truth tables.

Sum of products and product of sum forms – an algebraic expression is said to be in *sum of products (SOP)* form if and only if it consists of a sum (ORing) of product terms (ANDed), where each product term is formed by ANDing a number of uncomplemented or complemented variables, each of which is called a *literal*. The following expressions for $f_{majority}(x, y, z)$ are examples of SOP forms.

$$\begin{aligned}
 f_{majority}(x, y, z) &= xyz' + x'y'z + xy'z + xyz \\
 f_{majority}(x, y, z) &= xy + yz + xz
 \end{aligned}$$

An algebraic expression is a *product of sums (POS)* if and only if it consists of a product (ANDing) of sum terms (ORed) terms, where each sum term is formed by ORing a number of literals. The expressions below both represent the $f_{majority}(x, y, z)$ in POS form.

$$f_{majority}(x, y, z) = (x+y+z)(x+y'+z)(x'+y+z)(x'+y+z')$$

$$f_{\text{majority}}(x,y,z) = (x+y)(x+z)(y+z)$$

Expressions that are neither SOP nor POS are often referred to as *factored forms*. For example, $A(B + C') + A'C$.

Minterms and maxterms – A *minterm* of n variables is a product term of n literals where each variable appears only once. For example, product terms xyz' , $x'yz$, $xy'z$, and xyz are minterms of the 3-variable *majority* (x,y,z) function. On the other hand, product terms xy , yz , and xz are not minterms since only two of the three variables appear in each term. A *maxterm* of n variables is a sum term of n literals where each variable appears once. For the above POS form of *majority* (x,y,z), sum terms $x+y+z$, $x+y'+z$, $x'+y+z$, and $x'+y+z'$ are maxterms and $x+y$, $x+z$, and $y+z$ are not.

Canonical forms – An n -variable SOP expression is a *canonical sum of products (CSOP)* if and only if the product terms included in the expression are minterms. An n -variable POS expression is a *canonical product of sums (CPOS)* if and only if the sum terms included are maxterms. The CSOP and CPOS expressions representing a given function are each unique, i.e., there is only one CSOP and one CPOS for a given function. For example, the following equations represent the canonical forms for 3-variable *majority* and *odd-parity* functions. We will learn how to derive canonical forms later in this chapter.

$$\text{CSOP: } f_{\text{majority}}(x,y,z) = xyz' + x'y'z + xy'z + xyz$$

$$\text{CPOS: } f_{\text{majority}}(x,y,z) = (x+y+z)(x+y'+z)(x'+y+z)(x'+y+z')$$

$$\text{CSOP: } f_{\text{odd}}(x,y,z) = xy'z' + x'y'z + x'yz' + xyz$$

$$\text{CPOS: } f_{\text{odd}}(x,y,z) = (x+y+z)(x+y'+z')(x'+y+z')(x'+y'+z')$$

Minterm and maxterms lists -- To simplify writing the canonical SOP form, a special notation is commonly used in which each minterm is represented by an n -bit binary code. Each bit represents one of the variables of the minterm as follows:

Uncomplemented variable: 1

Complemented variable: 0

Variables are listed in the same order in each minterm. The significance of this notation is that a given minterm will evaluate to logic value 1 when the values of uncomplemented variables equal 1 and complemented variables equal 0, respectively. Using this code, the minterms of *majority* (x,y,z) can be written in any of the following notations.

Algebraic	Binary Code	Abbreviation
$x'yz$	011	m_3
$xy'z$	101	m_5
xyz'	110	m_6
Xyz	111	m_7

Hence, the CSOP of *majority* (x,y,z) can be written in abbreviated list forms as illustrated below.

$$\begin{aligned}f_{\text{majority}}(x,y,z) &= m_3 + m_5 + m_6 + m_7 \\f_{\text{majority}}(x,y,z) &= \sum m(2, 5, 6, 7)\end{aligned}$$

Corresponding notations exist for writing canonical POS forms using maxterms. Variables in maxterms are represented, as shown below, by values that result in the maxterm evaluating to 0.

Uncomplemented variable: 0
Complemented variable: 1

So maxterms of *majority* (x,y,z) can be written using any of the following.

Algebraic	Binary Code	Abbreviation
$x+y+z$	000	M_0
$x+y+z'$	001	M_1
$x+y'+z$	010	M_2
$x'+y+z$	100	M_4

Given this notation, the abbreviated CPOS forms of *majority* (x,y,z) follow.

$$\begin{aligned}f_{\text{majority}}(x,y,z) &= M_0 \cdot M_1 \cdot M_2 \cdot M_4 \\f_{\text{majority}}(x,y,z) &= \prod M(0, 1, 2, 4)\end{aligned}$$

Deriving canonical forms – The canonical representations of a function can be easily derived from its truth table. Let $f(A,B,C)$ be defined by the following table which also shows the minterm and maxterm abbreviations and valuations for each row.

Row (i)	ABC	F	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	000	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
1	001	0	0	1	0	0	0	0	0	0	1	0	1	1	1	1	1	1
2	010	1	0	0	1	0	0	0	0	0	1	1	0	1	1	1	1	1
3	011	0	0	0	0	1	0	0	0	0	1	1	1	0	1	1	1	1
4	100	1	0	0	0	0	1	0	0	0	1	1	1	1	0	1	1	1
5	101	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	1	1
6	110	1	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1
7	111	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0

Note that the bolded minterms are valued 1 when $f = 1$ and the bolded maxterms are valued 0 when $f = 0$. So if the bolded minterms are ORed together, each 1-value of f is covered and the following CSOP form of f is derived.

$$\begin{aligned}f(A,B,C) &= m_0 + m_2 + m_4 + m_6 \\&= \sum m(0, 2, 4, 6) \\&= A'B'C' + A'BC' + AB'C' + ABC'\end{aligned}$$

Similarly, choosing the bolded maxterms will cover the 0's of f and produce the following CPOS form of f .

$$\begin{aligned} f(A,B,C) &= M_1 \cdot M_3 \cdot M_5 \cdot M_7 \\ &= \prod M(1, 3, 5, 7) \\ &= (A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C') \end{aligned}$$

Canonical forms can also be derived from algebraic expressions using the postulates and theorems of Boolean algebra. Theorem 6 is particularly useful for this purpose as illustrated in the following two examples.

Example -- Convert the following function to canonical SOP form.

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

Let us apply Theorem 6a to each of the three product terms of this expression.

$$\begin{aligned} AB &= ABC + ABC = m_6 + m_7 \\ A\bar{C} &= A\bar{C}B + A\bar{C}B = A\bar{B}\bar{C} + A\bar{B}\bar{C} = m_4 + m_6 \\ \bar{A}C &= \bar{A}CB + \bar{A}CB = \bar{A}\bar{B}C + \bar{A}\bar{B}C = m_1 + m_3 \end{aligned}$$

Therefore,

$$\begin{aligned} f(A,B,C) &= AB + A\bar{C} + \bar{A}C \\ &= (m_6 + m_7) + (m_4 + m_6) + (m_1 + m_3) \\ &= \sum m(1,3,4,6,7) \end{aligned}$$

Example -- Expand the following function to canonical POS form.

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

Theorem 6b can be applied as follows to produce maxterms.

$$\begin{aligned} A &= (A + \bar{B})(A + B) \\ &= (A + \bar{B} + \bar{C})(A + \bar{B} + C)(A + B + \bar{C})(A + B + C) \\ &= M_3 M_2 M_1 M_0 \\ (A + \bar{C}) &= (A + \bar{C} + \bar{B})(A + \bar{C} + B) \\ &= (A + \bar{B} + \bar{C})(A + B + \bar{C}) \\ &= M_3 M_1 \end{aligned}$$

$$\text{Therefore, } A(A + \bar{C}) = (M_3 M_2 M_1 M_0)(M_3 M_1) = \prod M(0,1,2,3)$$

2.3.4 Incompletely Specified Functions (Don't Cares)

In the design of logic circuits, we often encounter cases in which logic functions are not completely specified. In other words, a function may be required to contain certain minterms and omit others, with the remaining minterms optional. In this case, the optional minterms may be included in the logic design if they help simplify the logic circuit, or otherwise be omitted. A minterm that is optional is called a *don't-care minterm*. If we express a function in terms of its

maxterms, the don't-care minterms are usually written in the corresponding maxterm form, in which case they would be called *don't-care maxterms*.

Don't-cares arise in two ways. First, certain input combinations will never be applied to a particular logic circuit; hence, since they never occur, their minterms may be used in any manner we choose. Such don't-care conditions arise quite naturally in many practical applications. For example, suppose a logic circuit has inputs $a_3a_2a_1a_0$, which represent binary coded decimal (BCD) digits as defined in Table 2.4. (Recall that BCD codes were discussed in Chapter 1.) Only 10 minterms, $m_0 \dots m_9$, occur, corresponding to the 10 decimal digits. The remaining six minterms, $m_{10} \dots m_{15}$, don't occur and are therefore don't-cares. Consequently, these terms may either be included or omitted as desired in logic expressions for any function $f(a_3a_2a_1a_0)$ of these inputs.

Table 2.4 Binary Coded Decimal (BCD) Codes

0: 0000	5: 0101
1: 0001	6: 0110
2: 0010	7: 0111
3: 0011	8: 1000
4: 0100	9: 1001

Don't-care conditions also arise where all input combinations do occur for a given circuit, but the output is required to be 1 or 0 only for certain combinations.

When writing Boolean expressions, don't-care minterms will be labeled d_i instead of m_i , and don't-care maxterms as D_i instead of M_i , as shown in the following example.

Example -- Suppose that we are given a function $f(A, B, C)$ that has minterms m_0 , m_3 , and m_7 and don't-care conditions d_4 and d_5 . We wish to express the function and its complement in both minterm and maxterm form and then reduce the function to its simplest form.

The minterm list form for this function is

$$f(A, B, C) = \sum m(0, 3, 7) + d(4, 5)$$

and the maxterm list is

$$f(A, B, C) = \prod M(1, 2, 6) \cdot D(4, 5)$$

So a given input combination is either a minterm, maxterm, or don't care. Note that the don't cares appear in both lists. Hence, the complement of a function with don't cares is found by interchanging minterms and maxterms and leaving don't cares unchanged as shown below.

$$\begin{aligned}\bar{f}(A, B, C) &= \sum m(1, 2, 6) + d(4, 5) \\ &= \prod M(0, 3, 7) \cdot D(4, 5)\end{aligned}$$

2.4 Minimization of Logic Expressions

Above we learned how Boolean algebra can be used to eliminate unnecessary terms and literals from logic expressions. Traditionally, this has been important since the number of terms and the number of literals relates to the number of logic gates needed to realize a logic function and to the number of inputs needed per gate. So minimizing an expression minimizes the number of gates in its realization making it less expensive and less complex. Unfortunately, algebraic manipulation can be tricky to identify where the different postulates and theorems can be applied, making its use practical only for functions of a few variables. In this section we will examine minimization methods that make the process easier and more practical for both completely-specified and incompletely-specified functions. Two methods are introduced – Karnaugh maps and Quine-McCluskey. Karnaugh maps can be used to graphically derive minimal sum of products and minimal product of sums expressions, and the Quine-McCluskey tabular method can be used to simplify single- and multiple-output functions. Karnaugh maps are particularly useful for manual processes applied to functions of two to six variables. Quine-McCluskey is useful for automated processes and functions of more than six variables.

The advent of hardware description languages for specifying logic functions and programmable logic devices for realizing functions has somewhat changed the value proposition for minimization. In other words, these tools and devices sometimes make it less important to minimize logic equations than to minimize design time or other considerations. This will be further addressed in later chapters.

2.4.1 Minimization Goals and Methods

The goal is to minimize the cost of realizing a logic function with physical circuit elements. However, the definition of cost depends on the nature of the circuit elements to be used. In general, it is desirable to minimize the number of circuit elements and to make each element as simple as possible. In a sum of products realization, minimizing cost implies reducing the number of product terms in the expression (to reduce gate count) and minimizing the number of literals in each product term (to minimize gate complexity, measured here in terms of the number of gate inputs). When using some programmable logic devices, to be described later in the book, the number of inputs to the logic forming the product terms is constant, so reducing the number of inputs to a gate provides no hardware savings, and thus only the number of terms is significant. In other cases, algebraic forms other than sum of product or product of sums might be desirable. In designing printed circuit boards, the total number of integrated circuit (IC) devices may be a more important factor than the number of individual gates. Field programmable gate arrays often employ look-up tables (LUTs) for realizing logic functions. In such cases, there is generally no value in reducing the number of terms and literals.

In all cases, a design must be made to fit within the constraints of the circuit elements to be used. The elements might have a limited number of inputs, or *fan-ins*, and may be limited in the number of output gates they can drive, or *fan-outs*. In some cases, the designer may be

restricted to a specific type of circuit element. Finally, timing considerations may dictate that a faster two-level realization be utilized, rather than a slower, three-level or higher, one. In addition, steps may need to be taken to prevent undesirable momentary output changes, called *hazards*, from occurring due to uneven propagation delays through a circuit.

In this section, the traditional approach to logic circuit minimization will be taken, that is, minimize the number of gates needed for a two-level realization (minimum number of products in a SOP form, or minimum number of sums in a POS form). If two or more expressions can be found containing the same number of terms, the expression with the fewest number of literals will be selected to allow gates with the lowest fan-in to be used. The following example illustrates these cost factors.

Example -- Determine the number of terms and literals in the following functions.

$$\begin{aligned}g(A, B, C) &= A\bar{B} + \bar{A}B + AC \\f(X, Y, Z) &= \bar{X}Y(Z + \bar{Y}X) + \bar{Y}Z\end{aligned}$$

$g(A, B, C)$ is a two-level form having three product terms and six literals. $f(X, Y, Z)$ is a factored form having seven literals, combined via three products and two sums.

Minimal forms – An expression in sum of products (SOP) form is a *minimal sum of products (MSOP)* if and only if there is no equivalent SOP that contains fewer product terms or the same number of product terms but fewer total literals. An expression in product of sums form (POS) is a *minimal product of sums (MPOS)* if and only if there is no equivalent POS that contains fewer sum terms or the same number of sum terms but fewer total literals.

Example -- Use Boolean algebra to find minimal SOP and POS forms of the function $f(X, Y, Z)$ from the previous example.

This expression can be minimized as follows.

$$\begin{aligned}f(X, Y, Z) &= \bar{X}Y(Z + \bar{Y}X) + \bar{Y}Z \\&= \bar{X}YZ + \bar{X}Y\bar{Y}X + \bar{Y}Z && [\text{P5(b)}] \\&= \bar{X}YZ + \bar{Y}Z && [\text{P6(b), P2(a)}] \\&= (X'Y + Y')Z && [\text{P5(b)}] \\&= (\bar{X} + \bar{Y})Z && [\text{T5(a)}] \\&= \bar{X}Z + \bar{Y}Z && [\text{P5(b)}]\end{aligned}$$

The last two forms represent the minimum POS and SOP forms, respectively. The minimum SOP form has two terms and a total of four literals. It therefore requires two two-input AND gates and one two-input OR gate to realize. However, the POS form has two terms and three literals and requires one two-input OR gate and one two-input AND gate to realize.

Example -- Use Boolean algebra to find a minimal SOP expression of the function

$$f(A, B, C, D) = ABC + ABD + \overline{ABC} + CD + \overline{BD}$$

which has four variables and 13 literals.

$$\begin{aligned}
 f(A, B, C, D) &= ABC + ABD + \overline{ABC} + CD + \overline{BD} && [\text{P3(a), P5(b), T5(a)}] \\
 &= ABC + AB + \overline{ABC} + CD + \overline{BD} && [\text{P5(b), T5(a)}] \\
 &= ABC + AB + \overline{BC} + CD + \overline{BD} && [\text{T4(a)}] \\
 &= AB + \overline{BC} + CD + \overline{BD} && [\text{P3(a), P5(b)}] \\
 &= AB + CD + B(\overline{C} + \overline{D}) && [\text{T8(b)}] \\
 &= AB + CD + \overline{BCD} && [\text{T5(a)}] \\
 &= AB + CD + B && [\text{T4(a)}] \\
 &= B + CD
 \end{aligned}$$

So the number of literals has been reduced from 13 to 3.

Minimizing functions with don't cares – Use Boolean algebra to minimize the function

$$f(A, B, C) = \sum m(0, 3, 7) + d(4, 5)$$

We can write the minterms and don't cares in algebraic form as follows

$$f(A, B, C) = \overline{ABC} + \overline{ABC} + ABC + d(\overline{ABC} + \overline{ABC})$$

The second and third terms differ in a single literal and hence may be grouped to produce

$$f(A, B, C) = \overline{ABC} + BC + d(\overline{ABC} + \overline{ABC})$$

Without the use of the don't-cares, no further minimization of the expression is possible. However, recall that the don't-cares by definition can be used as either zero or one. Therefore, we can use the corresponding minterms or omit them, depending on whether they do or do not aid in the simplification. If in the preceding expression we choose to use d_4 and omit d_5 , the expression becomes

$$\begin{aligned}
 f(A, B, C) &= \overline{ABC} + BC + \overline{ABC} \\
 &= \overline{BC} + BC
 \end{aligned}$$

which is its minimum SOP form. A similar analysis could be performed with the function in maxterm form. This approach for using don't cares to simplify expressions is adhoc and doesn't easily scale to more complex cases. However, Karnaugh maps and the Quine-McCluskey method can easily handle don't cares as will be demonstrated later in the chapter.

In the preceding examples, the optimality of the final expression depends on the skill and experience of the designer to determine the best postulate or theorem to apply at each step of the minimization process. This becomes a difficult task as the complexity of the

expressions to be minimized increases. Two systematic methods, the Karnaugh map and Quine-McCluskey, are presented in the following sections for simplifying logic expressions.

2.4.2 Karnaugh Maps (K-Maps)

The *Karnaugh map (K-map)* is a graphical tool that facilitates the minimization process for expressions of two to six variables. The Karnaugh map is actually nothing more than an extension of the concepts of truth tables, Venn diagrams, and minterms. To make the extension explicit, let us now transform a Venn diagram into a Karnaugh map. Consider the Venn diagram shown in Fig. 2.16 (a).

The two variables A and B are represented by designated subdivisions of the universal set. Figure 2.16 (b) illustrates that each unique disjoint subdivision of the Venn diagram is formed by the intersections AB , $\bar{A}B$, $A\bar{B}$, and $\bar{A}\bar{B}$. The reader should note that these intersections are just the minterms of two variables. The subdivisions of the Venn diagram are relabeled as minterms m_0 , m_1 , m_2 , and m_3 in Figure 2.16 (c). This form of the Venn diagram has unequal areas for the four minterms. However, we may adjust the areas and make them all the same, as shown in Fig. 2.16 (d). Note that adjacent areas of the Venn diagram are also adjacent in Fig. 2.16 (d). However, now one half of the diagram represents the variable A and one half also represents B . Since the minterm notation is identified with each square on the diagram, we may omit the letter m and leave only the subscript, as seen in Fig. 2.16 (e). This is one form of the Karnaugh map. A second form for the Karnaugh map is shown in Figure 2.16 (f). In this last form, the association of a map square with a particular variable, say A , is indicated by 0 for \bar{A} and 1 for A .

The two variables A and B can also be thought of as representing the columns and rows of the map, respectively, with the left column corresponding to $A = 0$ and the right to $A = 1$. The top row corresponds to $B = 0$ and the bottom to $B = 1$. Cells formed by intersections of columns with rows represent the corresponding combination of values. These combinations would be 00, 01, 11, 10 going counterclockwise from the upper left cell. Note that cells that are physically adjacent horizontally or vertically, but not diagonally, on the map are also logically adjacent. That is they differ in the value of only one variable.

It is important to emphasize that the Karnaugh map is also a graphical or pictorial representation of a truth table and hence there exists a one-to-one mapping between the two. The truth table has one row for each minterm while the Karnaugh map has one cell per minterm. This is illustrated in Fig. 2.16(g). Likewise, there is a one-to-one correspondence between truth table rows and Karnaugh map cells if maxterms are being utilized.

The development of the Karnaugh map for three variables is shown in Fig. 2.17. An important point that requires careful analysis is the step between Fig. 2.17(c) and (d). For example, consider the minterm m_0 . This minterm is adjacent to m_1 , m_2 , and m_4 in Fig. 2.17(c). However, in Fig. 2.17(d), m_0 is not physically adjacent to m_4 . To reconcile this inconsistency, the left and right edges of the map are considered to be touching. In other words, the left edge can be folded over until it touches the right edge, making the Karnaugh map for three variables appear as a cylinder. In practice, the map is drawn as in Fig. 2.17(e) or (f), and the left and right edges are imagined to be coincident.

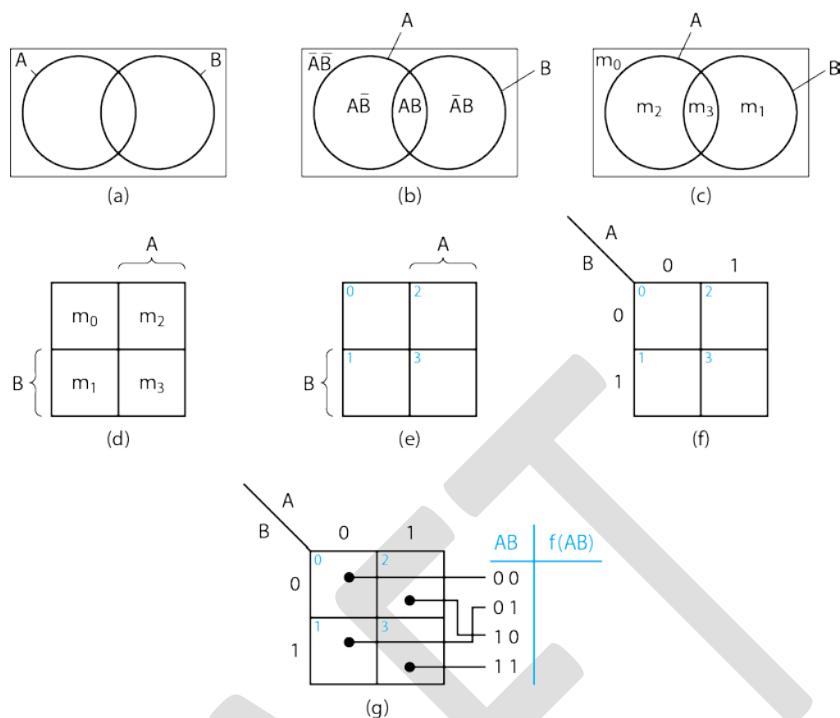


Figure 2.16 Two-Variable Karnaugh Map.

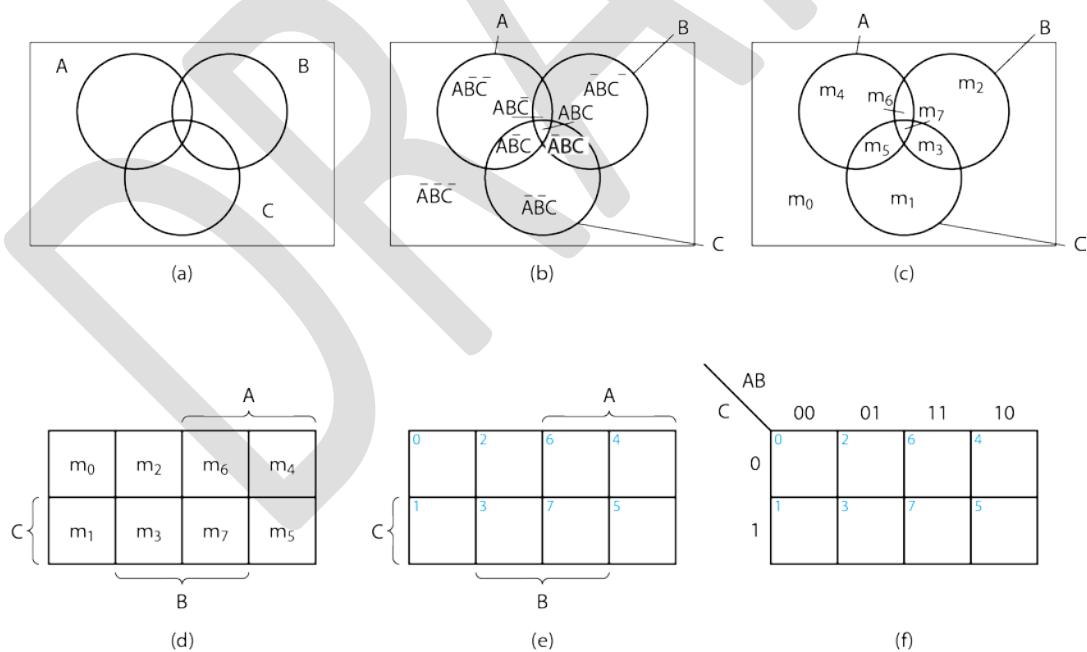


Figure 2.17 – Three-Variable Karnaugh Map.

K-maps for four, five, and six variables are illustrated in Figure 2.18. Note that the four-variable map is simply an extension of the three-variable map. The five-variable map for $f(A, B, C, D, E)$ is split into two halves, the left representing minterms containing \bar{A} and the right representing minterms containing A . The two halves should be viewed as being stacked one on top of the other, with vertically adjacent cells differing only in variable A and thus logically adjacent. For example, the cells corresponding to minterms m_5 ($\bar{A}\bar{B}C\bar{D}E$) and m_{21} ($A\bar{B}C\bar{D}E$) are logically adjacent. Likewise, the six-variable map is divided into four quadrants, each representing one combination of variables A and B . The quadrants should be viewed as being stacked on top of each other, with vertically adjacent cells being adjacent.

The five and six variable K-maps illustrated in Figure 2.18 (c) through (f) are often referred to as having stacked or overlayed structure for handling the fifth and sixth variables. An alternative structure based on Gray code labeling of cells is shown in Figure 2.19. The main advantage of the Gray-code structure is that it can be envisioned as two-dimensional rather than three. However, it is perhaps more difficult to identify all of the internal adjacencies.

As we've previously learned, logic functions may be expressed in a wide variety of forms, ranging from minterm/maxterm lists to simple SOP/POS expressions to more complex factored expressions. However, each has a unique canonical SOP/POS form. Now we'll examine methods for plotting, or mapping, switching functions of different forms on a Karnaugh map.

Logic functions may readily be plotted on a K-map if they are expressed in canonical form, since each minterm/maxterm of the canonical form corresponds to one cell on the K-map. Suppose we wish to find the K-map for the following function.

$$\begin{aligned} f(A, B, C) &= m(0, 3, 5) = m_0 + m_3 + m_5 \\ &= \prod M(1, 2, 4, 6, 7) = M_1 M_2 M_4 M_6 M_7 \end{aligned}$$

Recall that the maxterm list is readily derived from the minterm list, and vice versa. First, let us consider the representation of the function as the sum of minterms 0, 3, and 5. Using the set diagram form of the K-map, the function $f(A, B, C)$ represents the shaded areas shown in Figure 2.20(a). This same function plotted on a standard K-map is shown in Figure 2.20(b). Note that shaded areas are normally not used on K-maps. Instead, we employ the familiar 1 and 0 used in truth tables, with each shaded area (each minterm) represented by 1 and each unshaded area (each maxterm) represented by a 0. Under these conditions, the K-map of Figure 2.20(b) corresponds directly to the truth table of the function, with each cell of the K-map corresponding to one row of the truth table. When the function is represented as a sum of minterms, we normally omit the maxterms from the map and represent the function as shown in Fig. 2.20(c). Likewise, if the function is expressed as a product of maxterms, we omit the minterms and represent the function as shown in Figure 2.20(d).

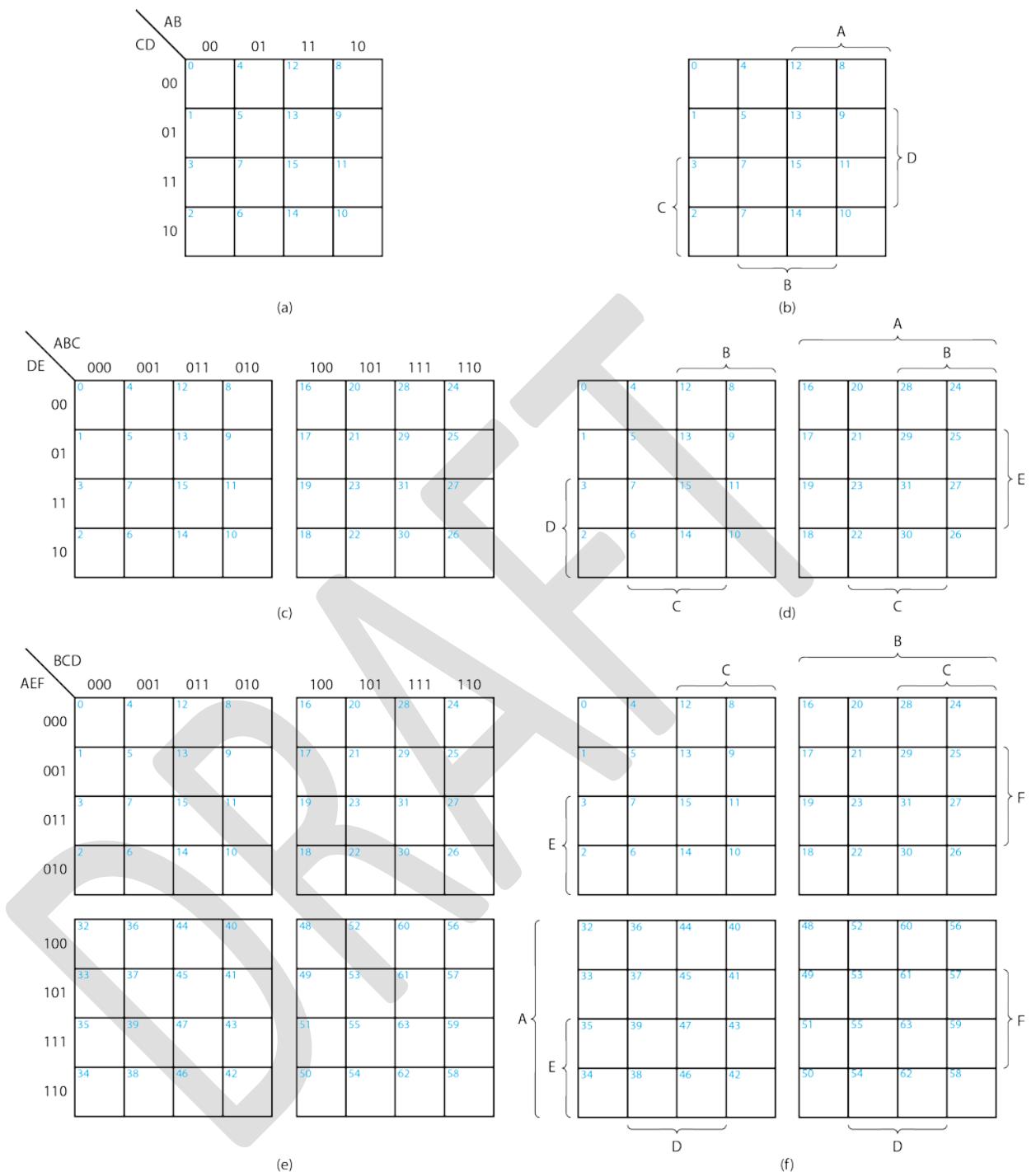


Figure 2.18 – Four, Five, and Six-Variable Karnaugh Maps.

		AB				
		CDE	00	01	11	10
000		0	8	24	16	
001		1	9	25	17	
011		3	11	27	19	
010		2	10	26	18	
110		6	14	30	22	
111		7	15	31	23	
101		5	13	29	21	
100		4	12	28	20	

(a) Five Variables (A,B,C,D,E)

		ABC								
		DEF	000	001	011	010	110	111	101	100
000		0	8	24	16	48	56	40	32	
001		1	9	25	17	49	57	41	33	
011		3	11	27	19	51	59	43	35	
010		2	10	26	18	50	58	42	34	
110		6	14	30	22	54	62	46	38	
111		7	15	31	23	55	63	47	39	
101		5	13	29	21	53	61	45	37	
100		4	12	28	20	52	60	44	36	

(b) Six Variables (A,B,C,D,E,F)

Figure 2.19 – Gray-Code Structured K-Maps for Five and Six Variables

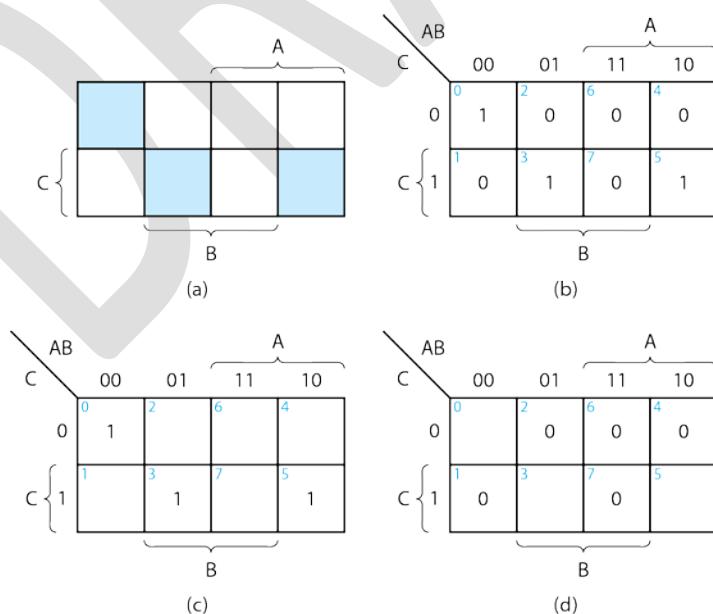


Figure 2.20 – Mapping Logic Functions.

Example -- Let us plot the following function on a K-map.

$$\begin{aligned}f(a,b,Q,G) &= m(0,3,5,7,10,11,12,13,14,15) \\&= \prod M(1,2,4,6,8,9)\end{aligned}$$

The function, expressed as a sum of minterms, is plotted on a version of the K-map labeled with both conventions in Figure 2.21(a). The function, expressed as a product of maxterms, is shown on the K-map in Figure 2.21(b).

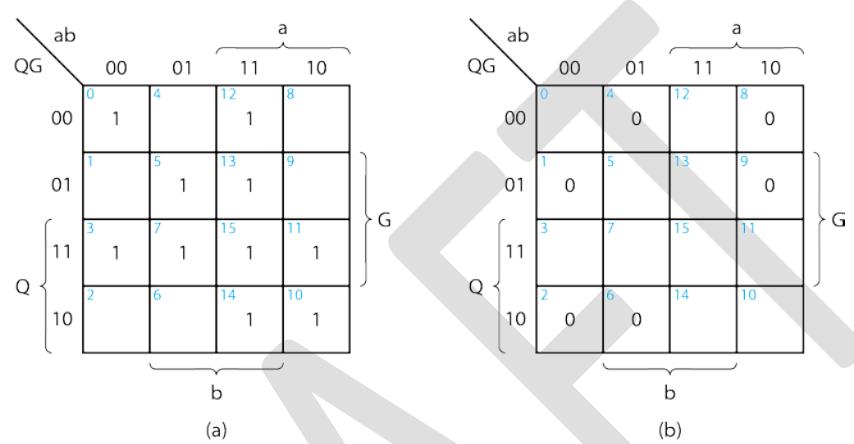


Figure 2.21 – Plotting a Function in Minterm List Form.

A most important point should be noted about the ordering of the variables. As was demonstrated in the last section, the minterm and maxterm numbers in the list change if the order of the variables is altered. Therefore, the order of the variables in the function fixes the order of the variables on the K-map.

Example -- Let us repeat the previous example with variables reordered to give $f(Q,G,b,a)$

First, write the minterms of $f(a,b,Q,G)$

$$\begin{aligned}f(a,b,Q,G) &= \sum m(0,3,5,7,10,11,12,13,14,15) \\&= \bar{a}\bar{b}\bar{Q}\bar{G} + \bar{a}\bar{b}Q\bar{G} + \bar{a}b\bar{Q}\bar{G} + \bar{a}bQ\bar{G} + a\bar{b}Q\bar{G} \\&\quad + \bar{a}\bar{b}QG + ab\bar{Q}\bar{G} + ab\bar{Q}G + abQ\bar{G} + abQG\end{aligned}$$

Next, rearrange the variables

$$\begin{aligned}f(Q,G,b,a) &= \bar{Q}\bar{G}\bar{b}\bar{a} + Q\bar{G}\bar{b}\bar{a} + \bar{Q}Gb\bar{a} + QGb\bar{a} + Q\bar{G}\bar{b}a \\&\quad + Q\bar{G}ba + \bar{Q}\bar{G}ba + \bar{Q}Gba + Q\bar{G}ba + QGba \\&= \sum m(0,12,6,14,9,13,3,7,11,15) \\&= \sum m(0,3,6,7,9,11,12,13,14,15)\end{aligned}$$

The function is plotted on the map of Figure 2.22, which is equivalent to the one in Figure 2.21 (a).

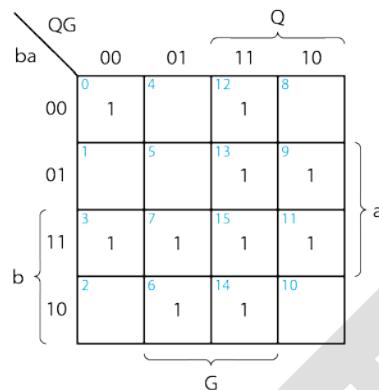


Figure 2.22 – Plotting a Function in Maxterm List Form.

K-maps can also be conveniently used to expand a function into canonical form. To illustrate this technique, we shall continue to use a combination of the two K-map forms shown in Figure 2.18.

Example -- Consider the following function, which is expressed as a sum of products.

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

We wish to plot the function on a K-map and determine its minterm and maxterm lists.

The map illustrating the two product terms is shown in Figure 2.23(a). The term AB represents the portion of the map where both A and B are 1, that is, minterms 6 and 7, as seen in Figure 2.23(b). The term $B\bar{C}$ represents the area on the map where B is 1 and C is 0, that is, minterms 2 and 6. Rather than shade in areas on the K-map, we usually plot the ones directly on the map, as shown in Figure 2.23(b). The map illustrating the maxterms of the function is derived directly from the minterm map and is shown in Figure 2.23(c).

In this example we note that minterm 6 is used twice, that is, is "covered" by both of the original product terms. The multiple use of minterms and maxterms is often the rule, rather than the exception, in logic functions. Also, note that it was not really necessary to draw the K-map of Figure 2.23(c) since the maxterms can be identified directly on the K-map of Figure 2.23(b) by simply noting the cells not set to 1.

From Figures 2.23(b) and (c), the function can be expressed in minterm and maxterm forms as

$$f(A, B, C) = m(2, 6, 7) = \prod M(0, 1, 3, 4, 5)$$

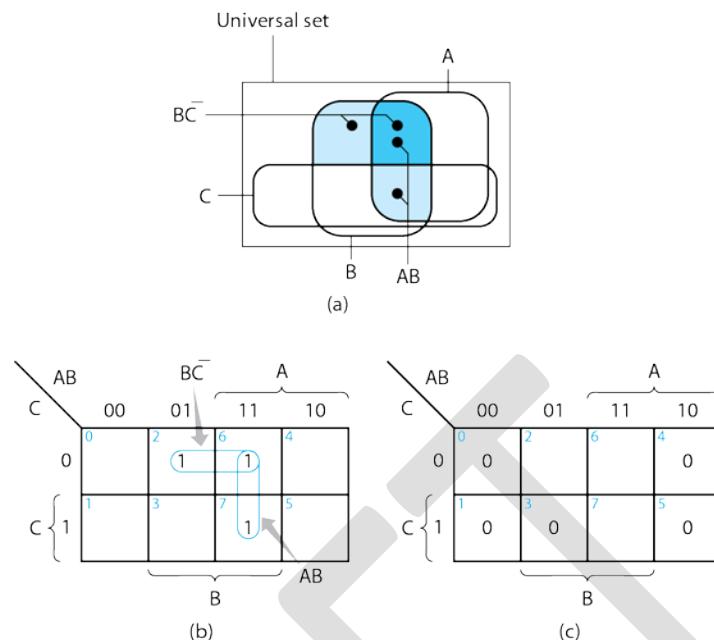


Figure 2.23 – Deriving Minterm and Maxterm Lists Using a Karnaugh Map.

Example -- Let us plot the following function on the K-map and determine its minterm and maxterm lists.

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

Since this expression is in POS form, we may plot the zeros (its maxterms) on the K-map. The map illustrating the maxterms is shown in Figure 2.24(a).

The term $A + C$ will force the function to have a zero value when $A = C = 0$, so it represents the area of the map for which $A = 0$ and $C = 0$, representing, that is, covering, maxterms 0, 1, 4, and 5. Likewise, $B + C$ represents maxterms 0, 1, 8, and 9. The term $\bar{B} + \bar{C} + D$ represents maxterms 6 and 14 because the term is 0 when $B = 1$, $C = 1$, and $D = 0$. The corresponding map illustrating the minterms is shown in Figure 2.24(b).

From Figures 2.24(a) and (b), the function can be expressed as

$$\begin{aligned} f(A, B, C, D) &= \prod M(0, 1, 4, 5, 6, 8, 9, 14) \\ &= \sum m(2, 3, 7, 10, 11, 12, 13, 15) \end{aligned}$$

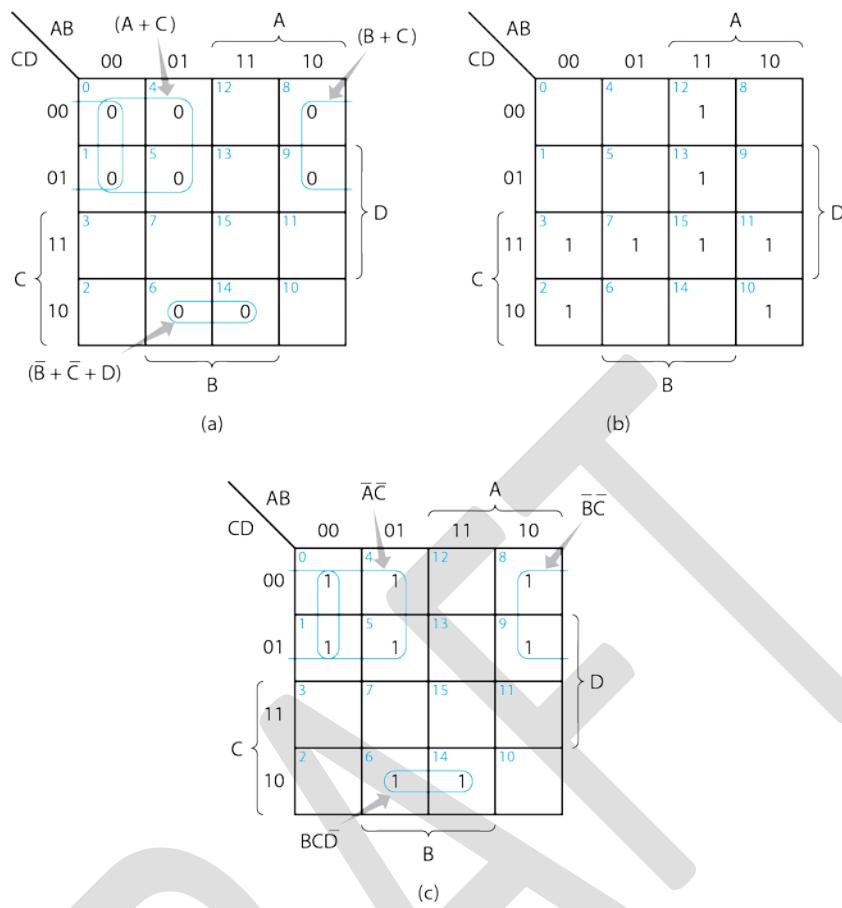


Figure 2.24 – Deriving Minterm and Maxterm Lists.

Some designers find plotting POS expressions to be awkward. An alternative method is to complement the function and apply DeMorgan's theorem to obtain a SOP expression for $\bar{f}(A, B, C, D)$.

$$\begin{aligned}\bar{f}(A, B, C, D) &= \overline{(A+C)(B+C)(\bar{B}+\bar{C}+D)} \\ &= \overline{(A+C)} + \overline{(B+C)} + \overline{(\bar{B}+\bar{C}+D)} \\ &= \overline{AC} + \overline{BC} + \overline{BCD}\end{aligned}$$

The SOP form of $\bar{f}(A, B, C, D)$ is then plotted on a K-map as shown in Figure 2.24(c). Recalling that the minterms of $\bar{f}(A, B, C, D)$ are the maxterms of $f(A, B, C, D)$, and vice versa, the K-map of Figure 2.24(a) is produced by simply converting each 0 cell (maxterm) in the K-map of $\bar{f}(A, B, C, D)$ to a 1 cell (minterm) in the K-map of $f(A, B, C, D)$. Note, also, that the expression for each product term plotted in Figure 2.24(b) is the complement of the expression of the corresponding sum term in Figure 2.24(c).

As was indicated in a previous example, it was not necessary to draw the K-map of Figure 2.24(b) to determine the minterm list, since the minterms are simply the nonzero cells in the K-map of Figure 2.24(a).

Example -- Derive the minterm list of the function

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

We begin by complementing the function and applying DeMorgan's theorem

$$\begin{aligned}\bar{f}(A, B, C, D) &= \overline{(\bar{A} + \bar{B})(\bar{A} + C + \bar{D})(\bar{B} + \bar{C} + \bar{D})} \\ &= \overline{\bar{A} + \bar{B}} + \overline{\bar{A} + C + \bar{D}} + \overline{\bar{B} + \bar{C} + \bar{D}} \\ &= AB + A\bar{C}D + BCD\end{aligned}$$

$\bar{f}(A, B, C, D) = AB + A\bar{C}D + BCD$ is plotted on the K-map as shown in Figure 2.25(a). From this K-map we can write

$$\bar{f}(A, B, C, D) = \sum m(7, 9, 12, 13, 14, 15)$$

Since the zero cells in the K-map of Fig. 2.25(a) represent $f(A, B, C, D)$, by inspection of the K-map we can write

$$f(A, B, C, D) = \sum m(0, 1, 2, 3, 4, 5, 6, 8, 10, 11)$$

The function is plotted on the K-map of Figure 2.25(b).

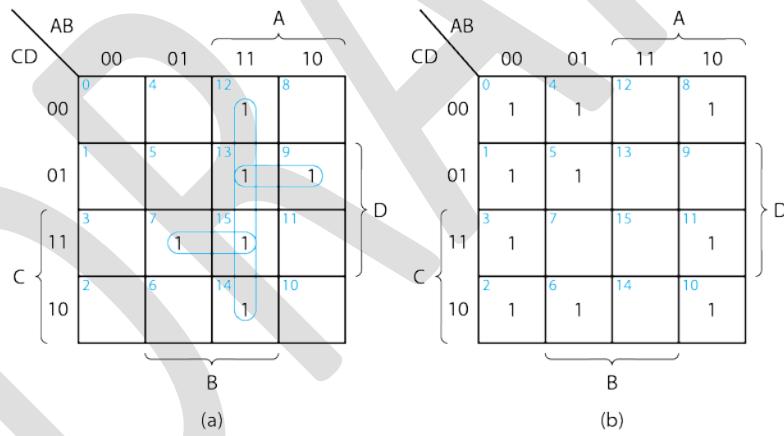


Figure 2.25 – Employing DeMorgan’s Theorem.

2.4.3 Minimization of Logic Expressions Using K-Maps

In the previous section we learned how to map Boolean functions on a K-map. Now let's learn how to use K-maps to find MSOP and MPOS forms of a function. Minimization of functions on the K-map is expedited by the fact that minterms or maxterms that are logically adjacent are also physically adjacent. *Logically adjacent* minterms are defined as two minterms, m_i and m_j , that differ in only one variable position. For example, $AB\bar{C}\bar{D}$ and $A\bar{B}\bar{C}\bar{D}$ are logically adjacent minterms of four variables since they differ only in variable

position D . From Theorem 6(a) we know that $A\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}D = A\overline{B}\overline{C}$; therefore, terms $A\overline{B}\overline{C}\overline{D}$ and $A\overline{B}\overline{C}D$ combine, eliminating variable D . In general, any two logically adjacent minterms can be combined, eliminating one variable. Larger groups of minterms can be combined to eliminate two or more variables as will be demonstrated in later examples. Maxterms can similarly be combined using Theorem 6(b), i.e., $(w + x + y + z)(w + x' + y + z) = (w + y + z)$, eliminating variable x .

On the K-map we illustrate combining terms by drawing a ring around the terms that, when combined, yield a simpler expression, that is, one with fewer literals. The following example illustrates the process of combining logically adjacent terms, using both Boolean algebra and K-map methods.

Example -- Minimize the following function using both Boolean algebra and a K-map.

$$f(A, B, C, D) = \sum m(1, 2, 4, 6, 9)$$

The minimization of this function via switching algebra may be performed as follows:

Step 1. Combine m_1 and m_9 .

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

Step 2. Combine m_2 and m_6 , duplicating m_6 first.

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

Step 3. Combine m_4 and m_6 .

$$\begin{aligned} f(A, B, C, D) &= \overline{B}\overline{C}D + \overline{A}CD + (\overline{A}BC\overline{D} + A\overline{B}\overline{C}D) \\ &= \overline{B}\overline{C}D + \overline{A}CD + \overline{A}BD \end{aligned}$$

The corresponding K-map minimization is shown in Figure 2.26. The minimization involves encircling sets of physically adjacent cells, corresponding to groups of logically adjacent minterms. Since opposite edges of the map are actually coincident, the horse shoe shapes in Figure 2.26 are really “circles” where each indicates the manner in which the minterms are combined to yield a simpler product term.

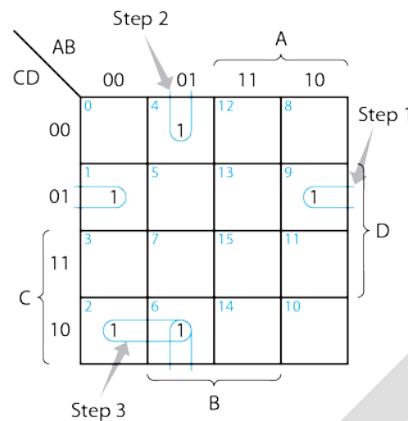


Figure 2.26 -- K-map for previous example.

In step 1, minterms m_1 and m_9 , which are adjacent on the K-map, are combined by encircling the corresponding cells. Comparing these two cells, we see that the variable that changes from 0 in m_1 to 1 in m_9 is the variable A ; hence, when the two minterms are combined, this variable is eliminated, as shown in step 1 of the Boolean algebra approach. Minterms m_4 and m_6 are combined on the K-map in step 2. Comparing these two cells we see that they differ only in the variable C ; hence, this variable is eliminated when cells 4 and 6 are combined. Finally, in step 3, combining cells 2 and 6 eliminates the variable B . Thus, the three steps on the K-map are equivalent to the corresponding steps indicated in the Boolean algebra simplification. The reader is reminded, as is demonstrated in steps 2 and 3, that minterms can be used more than once because $X = X + X$ by idempotency (Theorem 1).

Guidelines for Minimizing Functions Using K-maps -- There are five important points to keep in mind when minimizing functions on K-maps.

1. Each cell (minterm or maxterm) on a K-map of two variables has two cells (minterms or maxterms) that are logically adjacent, each cell on a K-map of three variables has three adjacent cells, and so on. In general, each cell on a K-map of n variables has n logically adjacent cells, with each pair of adjacent cells differing in exactly one variable.
2. When combining terms (cells) on a K-map we always group cells in powers of 2, that is, two cells, four cells, eight cells, and so on. Grouping two cells eliminates one variable, grouping four cells eliminates two variables, and so on. In general, grouping 2^n cells eliminates n variables.
3. Group as many cells together as possible; the larger the group is, the fewer literals in the resulting term.
4. Make as few groups as possible to cover all the cells (minterms or maxterms) of the function. A minterm or maxterm is *covered* if it is included in at least one group. The fewer the groups, the fewer the number of product terms in the minimized function. Each minterm or maxterm may be used as many times as it is needed in steps 4 and 5; however, it must be used at least once. As soon as all minterms or maxterms are used once, stop.

5. In combining cells on the map, always begin with those cells for which there are the fewest number of adjacent cells (the "loneliest" cells on the map). Minterms or maxterms with multiple adjacent minterms (called *adjacencies*) offer more possible combinations and should therefore be combined later in the minimization process.

General Principles for Minimizing SOP Expressions – The previous discussion illustrated the relationship between the Boolean algebra and K-map procedures for simplifying a Boolean function. We now define four terms that are not only useful in K-map minimization, but in the Quine-McCluskey method that will be presented later. These terms are *implicant*, *prime implicant*, *essential prime implicant*, and *cover*.

An *implicant* is a product term (that is, a product of one or more literals) that covers one or more minterms of a function. There are eleven implicants of the function mapped in Figure 2.27. Five of the implicants are minterms $\{\bar{A}\bar{B}C, \bar{A}\bar{B}\bar{C}, \bar{A}BC, A\bar{B}\bar{C}, ABC\}$, five are covers of two minterms $\{\bar{A}B, AB, \bar{A}C, BC, B\bar{C}\}$, and one is a cover of four minterms $\{B\}$.

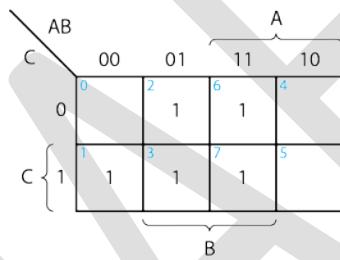


Figure 2.27 -- K-map illustrating implicants.

A *prime implicant* is an implicant that is not a part of (covered by) any other implicant of the function. Recall from the previous example that as we combine minterms (implicants) we eliminate variables. As we combine implicants (in powers of 2) into maximal groups, we form prime implicants. On the K-map a prime implicant is equivalent to a set of cells that is not a subset of any set containing a larger number of cells. Prime implicants represent the largest groupings of minterms that can be derived for a given function. In the K-map of Figure 2.27, there are only two prime implicants: B and $\bar{A}C$. Prime implicant B covers implicants $\bar{A}\bar{B}\bar{C}$, $\bar{A}\bar{B}C$, $A\bar{B}\bar{C}$, ABC , $\bar{A}B$, AB , $B\bar{C}$, and BC . Prime implicant $\bar{A}C$ covers implicants $\bar{A}\bar{B}C$ and $\bar{A}\bar{B}\bar{C}$.

An *essential prime implicant* is a prime implicant that covers at least one minterm that is not covered by any other prime implicant. In the K-map of Figure 2.27, prime implicant $\bar{A}C$ is essential because it is the only prime implicant that covers minterm 1, and prime implicant B is essential because it is the only prime implicant that covers minterms 2, 6, and 7. An essential prime implicant is easily identified on the K-map by noting that it covers at least one minterm that is circled only once.

Finally, a *cover* of a function is a set of prime implicants for which each minterm of the function is contained in (covered) by at least one prime implicant. All essential prime implicants of a function must be selected in any cover of that function. For the K-map of Figure 2.27, the set of prime implicants $\{B, \bar{A}C\}$ represents a cover of the function which yields the MSOP expression $B + A'C$.

Procedure for Deriving Minimal SOP Expressions with K-maps -- Our primary goal in minimizing a function is to find a minimum set of prime implicants that covers the function, from which a minimal sum of products expression result. With this in mind, we now present a procedure for finding a minimum cover of a function that is plotted on a K-map. These procedures are designed to help the user follow the five guidelines presented previously, thereby simplifying the minimization process and ensuring a high likelihood of finding a minimal cover of a given function.

Procedure 2.1 – Finding the MSOP of an n variable Boolean function using a K-map.

1. Map the minterms of the function.
2. Find all minterms that are isolated, that is, not adjacent to any other minterm. Such minterms cannot be simplified and must be included in the MSOP as essential prime implicants of n literals. Encircle each to mark it has been covered.
3. Find a minterm that is adjacent to only one other minterm. If none exists, go to step 4.
4. Combine these to produce an essential prime implicant of $n-1$ literals. The literals included will be those that are common across the two minterms. Encircle the two minterms to indicate they have been covered. Continue this process until all such minterms have been found.
4. Find an uncovered minterm with the fewest number of adjacencies. If none exist, then the minimum cover has already been found and the MSOP has been derived. If more than one such minterm exists, choose one that covers the most uncovered minterms. Combine these minterms to produce a prime implicant. Encircle the minterms to show they have been covered. The number of literals in the prime implicant will depend upon the number of minterms in the group and will range from $n-1$ to 0. The literals included will be those that are common across all minterms in the group.
5. If all minterms have been covered, then the MSOP has been derived and consists of the sum of the prime implicants that have been produced. Otherwise, repeat Step 4.

We now illustrate the application of Procedure 2.1 with several examples.

Example – Find a minimum SOP expression for $f(x, y, z) = \sum m(2, 5, 6)$.

Step 1 – The function is mapped as shown in Figure 2.28a.

Step 2 – Minterm m_5 is isolated which makes it an essential prime implicant.

Step 3 – Minterms m_2 and m_6 are adjacent and grouped to produce essential prime implicant 2-6 as shown in Figure 2.28b.

Steps 4 and 5 – All minterms are covered by the minimal cover {5, 2-6} which is represented by the following MSOP expression.

$$f(x, y, z) = xy'z + yz'$$

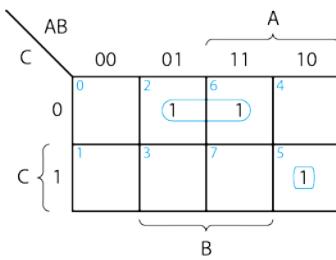


Figure 2.28. $f(x, y, z) = \sum m(2, 5, 6)$.

Example -- Find a minimum SOP expression for $f(A, B, C) = \sum m(1, 2, 3, 6)$.

Step 1 – The function is plotted on the K-map of Figure 2.29.

Step 2 – There are no isolated minterms.

Step 3 – Minterms m_1 and m_6 each have one adjacency and can be grouped with minterms m_3 and m_2 , respectively, to produce essential prime implicants 1-3 and 2-6.

Steps 4, and 5 – All four minterms are covered, so {1-3, and 2-6} is a minimal cover generating MSOP $BC' + A'C$. Note that m_2 and m_3 could be grouped to produce prime implicant 2-3. However, 2-3 is not needed since the two essential prime implicants cover the minterms of the function.

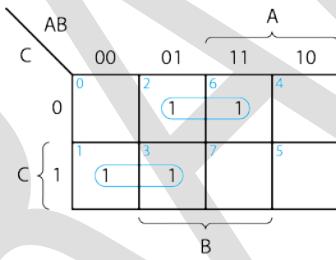


Figure 2.29. $f(A, B, C) = \sum m(1, 2, 3, 6) = \overline{A}C + B\overline{C}$.

Example -- Use a K-map and Procedure 2.1 to find a minimum sum-of-products of the function

$$f(A, B, C, D) = \sum m(2, 3, 4, 5, 7, 8, 10, 13, 15)$$

Step 1 -- Plot the function on the map as shown in Figure 2.30a.

Step 2 – There are no isolated minterms, so go to Step 3.

Step 3 – From Figure 2.30a, we see that minterm m_4 is only adjacent to minterm m_5 , so they must be grouped to produce essential prime implicant 4-5. Likewise, m_8 is only adjacent to m_{10} and must be grouped to produce essential prime implicant 8-10. These groupings are encircled to indicate the corresponding minterms have been covered as shown in Figure 2.30b.

Step 4 – Figure 2.30b also reveals that minterms m_2 , m_3 , m_{13} , and m_{15} are uncovered and each have two adjacencies. Since m_2 and m_3 are adjacent they can be selected to form prime implicant 2-3 as shown in Figure 2.30c. This leaves minterms m_7 , m_{13} , and m_{15} uncovered. These three can be grouped with minterm m_5 to produce essential prime implicant 5-7-13-15. It's essential because it's the only prime implicant covering m_{13} and m_{15} .

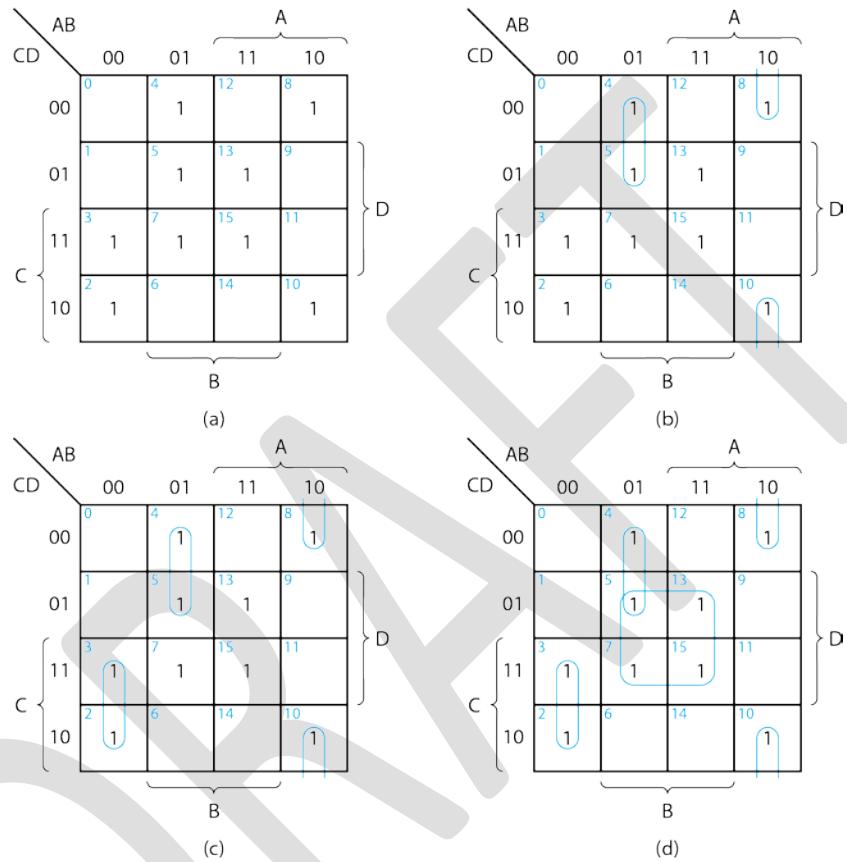


Figure 2.30. K-maps Illustrating Algorithm 2.1. (a) Plot of the function. (b) Prime implicants 4-5 and 8-10. (c) Prime implicant 2-3 covers m_2 . (d) Prime implicant 5-7-13-15 completes the cover.

Step 5 – All minterms have now been covered as shown in Figure 2.30d with minimum cover { 4-5, 8-10, 2-3, 5-7-13-15 }. The product terms that the prime implicants represent are obtained as follows. On the map, prime implicant 2-3 is located outside A and B and inside C ; therefore, the term is $\bar{A}\bar{B}C$. The prime implicant 4-5 is outside A and C and inside B , therefore, the term is $\bar{A}\bar{B}\bar{C}$. Prime implicant 5-7-13-15 is inside B and D ; therefore, the term is BD . Finally, prime implicant 8-10 is located inside A and outside B and D ; hence, the term is $A\bar{B}\bar{D}$. Therefore, the minimum sum of products of f is

$$f(A, B, C, D) = \bar{A}\bar{B}C + \bar{A}\bar{B}\bar{C} + BD + A\bar{B}\bar{D}$$

Example -- Use a K-map to find a MSOP of the following function.

$$f(A, B, C, D) = \sum m(0, 5, 7, 8, 10, 12, 14, 15)$$

Step 1 -- The function is plotted on the map in Figure 2.31a.

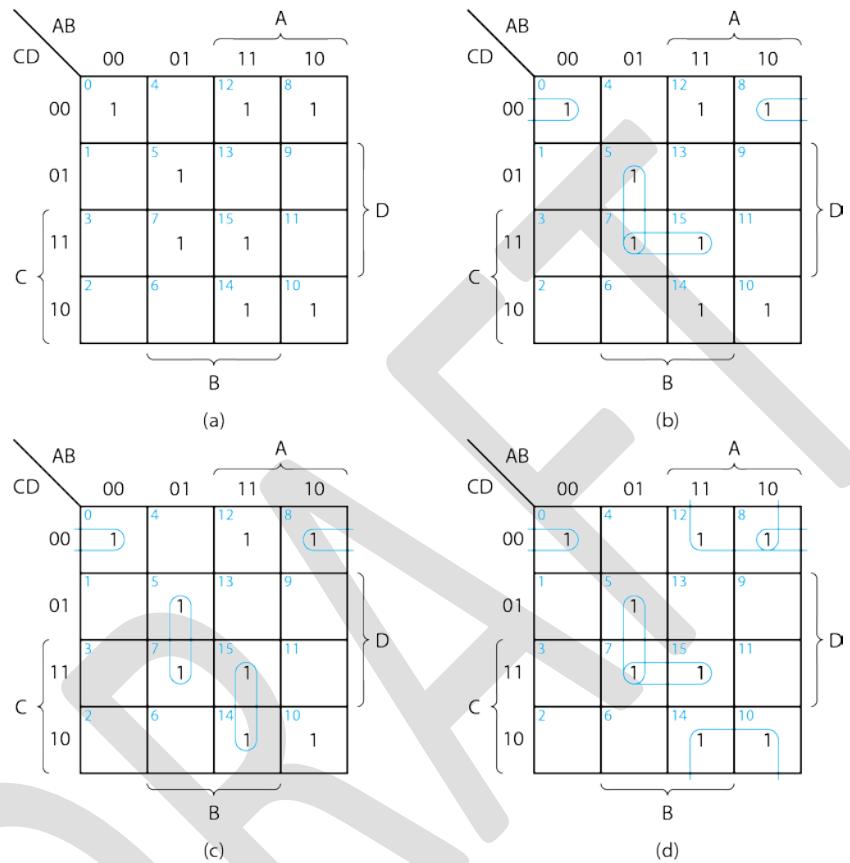


Figure 2.31. K-maps for $f(A, B, C, D) = \sum m(0, 5, 7, 8, 10, 12, 14, 15)$. (a) Plot of the function. (b) PIs 0-8, 5-7, and 7-15 (c) PIs 0-8, 5-7, and 14-15. (d) Minimal cover {0-8, 5-7, 7-15, 8-10-12-14}.

Step 2 – There are no isolated minterms.

Step 3 – Minterms m_0 and m_5 have only one adjacency each. Grouping m_0 and m_8 produces essential prime implicant 0-8, and grouping m_5 and m_7 produces essential prime implicant 5-7.

Step 4 – Minterm m_{15} is not covered and has two adjacencies, m_7 and m_{14} . Choosing m_7 produces prime implicant 7-15 while m_{14} produces 14-15. Figure 2.31b shows the former grouping and Figure 2.31c the latter. Only one group is needed so 7-15 is selected. This leaves minterms m_8 , m_{10} , m_{12} , and m_{14} uncovered, but they can all be grouped together to form

essential prime implicant 8-10-12-14 as shown in Figure 2.31d. It is essential since it's the only prime implicant that covers m_{10} and m_{12} .

Step 5 – All minterms have been covered with minimal cover {0-8, 5-7, 7-15, 8-10-12-14} which yields the following MSOP.

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

Had prime implicant 14-15 been chosen rather than 7-15, the minimal cover would be {0-8, 5-7, 14-15, 8-10-12-14} yielding MSOP

$$f(A, B, C, D) = \overline{BCD} + \overline{ABD} + A\overline{D} + ABC$$

Since both SOP forms contain the same number of terms and literals, either represents a MSOP expression for the function.

The following examples are presented in rapid order without direct reference to Procedure 2.1. Each example contains, first, the function to be minimized, second, a K-map with the prime implicants of the minimal cover identified (circled), and, finally, an expression for the minimized function, which is a minimal covering of the function selected from the prime implicants.

Example -- Find a minimum SOP expression for $f(A, B, C, D) = \sum m(0,1,2,7,8,9,10,15)$.

For this function, as seen in Figure 2.32, there are three prime implicants, 7-15, 0-1-8-9, and 0-2-8-10, all of which are essential and therefore form the minimum cover of the function. Note that prime implicant 0-2-8-10 covers the four corners of the K-map. The four corners are adjacent by virtue of the fact the top and bottom rows are adjacent (differing in variable C), as are the leftmost and rightmost columns (differing in variable A).

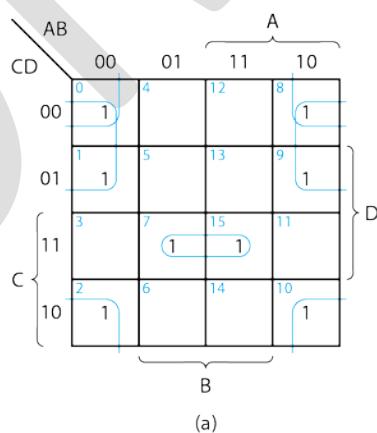


Figure 2.32 $f(A, B, C, D) = \overline{BD} + \overline{BC} + BCD$.

Example -- Find a minimal SOP expression for $f(A, B, C, D) = \sum m(0, 4, 5, 7, 8, 10, 14, 15)$.

From the K-map of Figure 2.33a, we can see that each minterm is covered by two prime implicants, and therefore none of the prime implicants is essential. In addition, each minterm has exactly two adjacencies. This condition is referred to as a *cycle* in the K-map. Whenever a cycle occurs, we must break it by making an arbitrary initial selection.

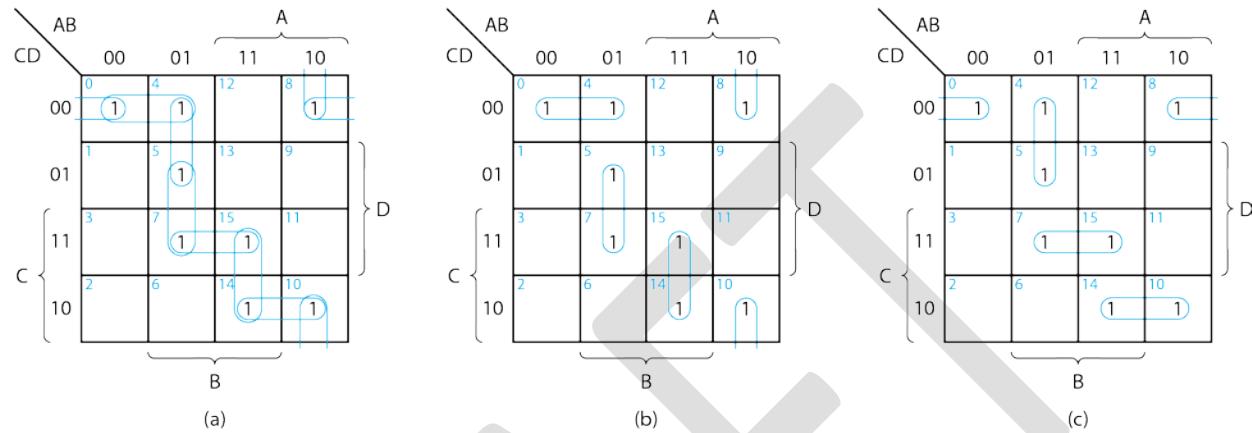


Figure 2.33: Function with no essential prime implicants. (a) All prime implicants. (b) Minimal cover 1. (c) Minimal cover 2.

Let us begin by covering minterm m_0 . We see that it can be covered with either of prime implicants 0-4 or 0-8. Selection of 0-4 leads to the minimum cover shown in Fig. 2.33b, for which the minimal SOP expression is

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

Selection of prime implicant 0-8 to cover m_0 leads to the minimum cover shown in Fig. 2.33c, for which the minimum SOP expression is

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

Thus, there are two totally different minimal SOP expressions for the function, which have the same number of terms and literals and therefore the same cost.

Example -- Find a minimum SOP for the following five variable function

$$f(A, B, C, D, E) = \sum m(0, 2, 4, 7, 10, 12, 13, 18, 23, 26, 28, 29)$$

For five-variable maps, as shown in Figure 2.34, each minterm has only five possible adjacencies; for example, the adjacencies for m_7 are m_3, m_5, m_6, m_{15} , and m_{23} . Hence, minterms that appear in similar positions in the two parts of the map are adjacent and can be combined. Another way to look at this is to imagine that the two halves of the map are stacked so that adjacencies align vertically between the halves as well as horizontally on each half. In

in the previous example, m_7 and m_{23} align vertically and can be grouped. Similarly, the minterms m_{12} and m_{13} on one plane can be grouped with m_{28} and m_{29} from the other.

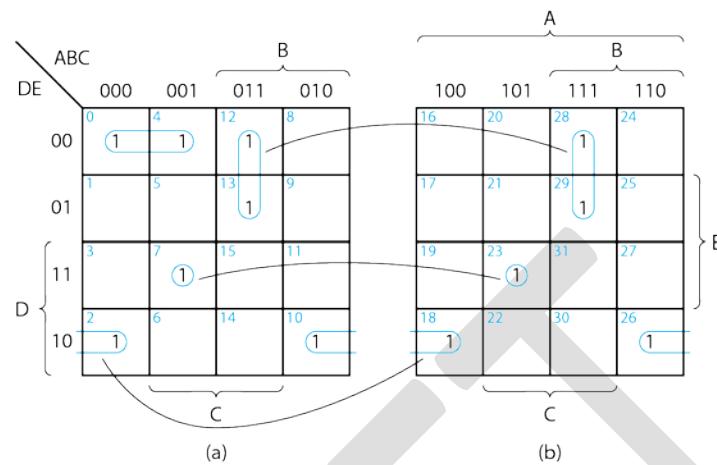


Figure 2.34: Minimizing a five-variable function.

Using the above and Procedure 2.1 as a guide, an MSOP expression can be derived as follows. First, note there are no isolated minterms, so look for minterms having only one adjacency and group them with their neighbor to form essential prime implicants. Thusly, m_7 and m_{23} form essential prime implicant 7-23. Now look for minterms with two adjacencies. This yields prime implicants 0-4, 0-2, 2-4, 2-10-18-26, and 12-13-28-29 with the latter two being essential. The three essential prime implicants cover all minterms except m_0 and m_4 which can be covered with a single prime implicant 0-4. Hence the minimal cover is {7-23, 0-4, 2-10-18-26, 12-13-28-29} resulting in the following minimum SOP.

$$f(A,B,C,D,E) = \overline{ABDE} + BCD + \overline{BCDE} + \overline{CDE}$$

General Principles for Minimizing POS Expressions -- Thus far we have concentrated our discussion on the use of the K-map for minimizing expressions in SOP form. Analogous procedures can be employed to minimize a function in POS (product of sums) form. Furthermore, all the techniques we have learned for combining minterms can be applied in the combination of maxterms to produce a minimum POS form. In this regard, we define the terms *implicate*, *prime implicate*, *essential prime implicate*, and *cover* for use in deriving minimal POS expressions.

An *implicate* is a sum term, that is, a sum of one or more literals, that covers one or more maxterms of a function. Note that an implicate represents an input combination for which the function evaluates to 0. On the K-map, an implicate is a group of adjacent maxterms, or 0 cells. A *prime implicate* is an implicate that is not covered by any other prime implicate of the function. On the K-map, a prime implicate is a group of adjacent maxterms that is not covered by a larger group of maxterms. An *essential prime implicate* is a prime implicate that

covers at least one maxterm that is not covered by any other prime implicate. On the K-map, an essential prime implicate covers at least one maxterm that is circled only once. A *cover* of a function is a set of implicants for which each maxterm of the function is contained in (covered by) at least one prime implicate.

Procedure for Deriving Minimal POS Expressions from K-maps -- Procedure 2.1 for deriving minimal SOP expressions is easily modified for deriving minimal POS expressions. The process of grouping cells into maximal groups and then selecting a minimum number of groups to cover the function is identical, except that the cells represent maxterms rather than minterms, and the groups are called implicants rather than implicants. Therefore, we have the following procedure for deriving a minimum product of sums (MPOS).

Procedure 2.2 – Finding the MPOS of an n variable Boolean function using a K-map.

1. Map the maxterms of the function.
2. Find all maxterms that are isolated, that is, not adjacent to any other maxterm. Such maxterms cannot be simplified and must be included in the MPOS as essential prime implicants of n literals. Encircle each to mark that it has been covered.
3. Find a maxterm that is adjacent to only one other maxterm. If none exists, go to step 4. Combine these to produce an essential prime implicate of $n-1$ literals. The literals included will be those that are common across the two maxterms. Encircle the two maxterms to indicate they have been covered. Continue this process until all such maxterms have been found.
4. Find an uncovered maxterm with the fewest number of adjacencies. If none exist, the minimum cover has already been found and the MPOS has been derived. If more than one such maxterm exists, choose one that covers the most uncovered maxterms. Combine these maxterms to produce a prime implicate. Encircle the maxterms to show that they have been covered. The number of literals in the prime implicate will depend upon the number of maxterms in the group and will range from $n-1$ to 0. The literals included will be those that are common across all maxterms in the group.
5. If all maxterms have been covered, then the MPOS has been derived and consists of the product of the prime implicants that have been produced. Otherwise, repeat Step 4.

The following examples illustrate the application of Procedure 2.2.

Example – Find a minimum POS form of the function $f(A, B, C, D) = \prod M(0,1,2,3,6,9,14)$.

Step 1 – The function is plotted as shown in Figure 2.35a.

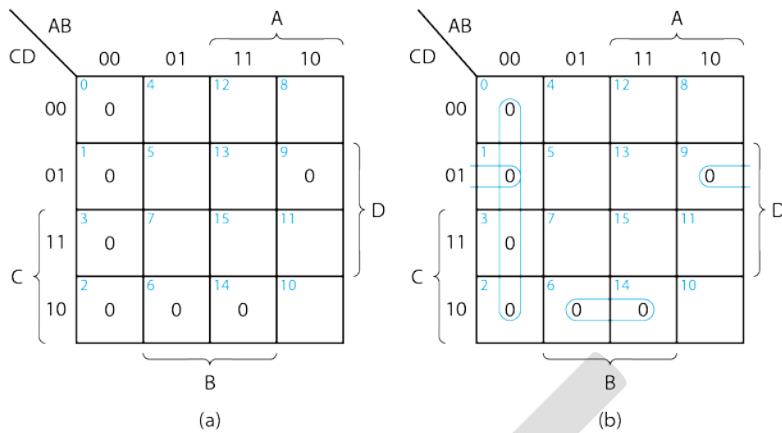


Figure 2.35: K-maps for Example 3.17. (a) Plot of the function. (b) Minimum cover.

Step 2 – There are no isolated maxterms.

Step 3 – Maxterms M_9 and M_{14} each have one adjacency. Therefore, we cover M_9 with prime implicate 1-9, and M_{14} with prime implicate 6-14.

Step 4 – Next looking for uncovered maxterms with two adjacencies, we find maxterm M_0 and cover it with prime implicate 0-1-2-3.

Step 5 – All maxterms are now covered with {1-9, 6-14, 0-1-2-3} as shown in Figure 2.35b. The resulting MPOS is $f(A, B, C, D) = (A+B)(B+C+\bar{D})(\bar{B}+\bar{C}+D)$.

Example – Find minimal POS and SOP expressions for the following function

$$f(A, B, C, D) = \prod M(0, 2, 3, 9, 11, 12, 13, 15)$$

To find the minimal POS expression, we plot the maxterms of the function as shown in Figure 2.36a. Using Procedure 2.2 as a guide, we find prime implicate 0-2 to cover M_2 , 12-13 to cover M_{12} , and 9-11-13-15 to cover M_9 . This leaves M_3 , which can be covered with either 3-11 or 2-3. Since each contains the same number of literals, let us arbitrarily select 2-3. The resulting minimal POS expression is

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

We can also generate the minimal POS form by plotting \bar{f} , as in Figure 2.36b, and then deriving a minimum SOP expression for \bar{f} :

$$\bar{f}(A, B, C, D) = \bar{A}\bar{B}\bar{D} + A\bar{B}\bar{C} + AD + \bar{A}\bar{B}C$$

Complementing, we get

$$\begin{aligned} f(A, B, C, D) &= \overline{\bar{A}\bar{B}\bar{D}} + \overline{A\bar{B}\bar{C}} + \overline{AD} + \overline{\bar{A}\bar{B}C} \\ &= (\bar{A}\bar{B}\bar{D})(\bar{A}\bar{B}\bar{C})(\bar{A}\bar{D})(\bar{A}\bar{B}C) \\ &= (A+B+D)(\bar{A}+\bar{B}+C)(\bar{A}+\bar{D})(A+B+\bar{C}) \end{aligned}$$

To generate a minimal SOP expression for $f(A, B, C, D)$, we plot the minterms of the function instead of the maxterms, as shown in Figure 2.36c. Procedure 2.1 produces the minimal cover shown in Figure 2.36c. Taking the sum of the prime implicants gives

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

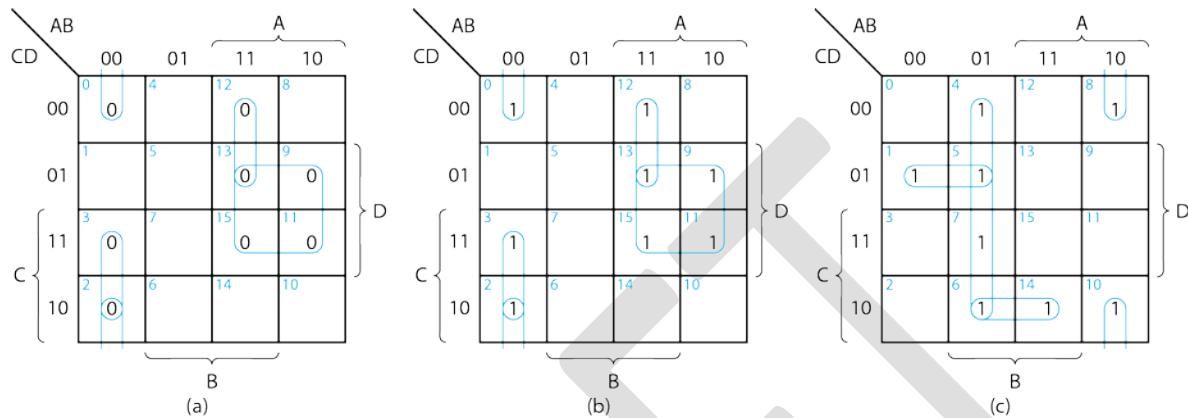


Figure 2.36. Deriving POS and SOP forms of a function. (a) Maxterms of f . (b) Minterms of \bar{f} . (c) Minterms of f .

The previous example illustrates an important point. From any description of a function, we can generate either a minimum SOP or a minimum POS expression. To generate the minimal SOP expression, we work with the minterms of the function, whereas for a minimal POS expression we work with the maxterms. Once we have plotted the maxterms of a function on a K-map, we automatically have a map of the minterms, and vice versa. Therefore, the initial function format need not affect whether we choose a SOP or POS form for the minimized function.

Some designers find working with maxterms and prime implicants awkward and prefer to use K-maps for deriving minimal SOP forms. As illustrated in the previous example, SOP methods can be utilized to produce a minimal POS form of a function, f , by first deriving the MSOP of \bar{f} and then complementing the result to get the MSOP of f . We begin by plotting the complement of the function \bar{f} on the K-map. This converts the maxterms (zeros) into minterms (ones). We then use Procedure 2.1 to derive a minimal SOP expression for \bar{f} . Next, the SOP expression is complemented by applying DeMorgan's theorem (Theorem 8) to produce the desired POS form of f . Thus we minimize \bar{f} as a SOP function and then complement it to get f . This procedure is summarized as follows.

Procedure 2.3 -- Find the minimum POS of function f .

1. Plot the minterms of function \bar{f} on the K-map.
2. Use Procedure 2.1 to derive a minimum SOP expression of \bar{f} .

3. Complement the MSOP expression using DeMorgan's theorem to produce a minimum POS expression of f .

Example -- Use Procedure 2.3 to find the MSOP of $f(A, B, C, D) = \prod M(0, 1, 2, 3, 6, 9, 14)$.

As illustrated in Figure 2.37a, we begin by plotting its complement on a K-map

$$\bar{f}(A, B, C, D) = \sum m(0, 1, 2, 3, 6, 9, 14)$$

This is shown on the K-map of Figure 2.37a. Note that this map is identical to that of Figure 2.35a, but with the zeros replaced by ones.

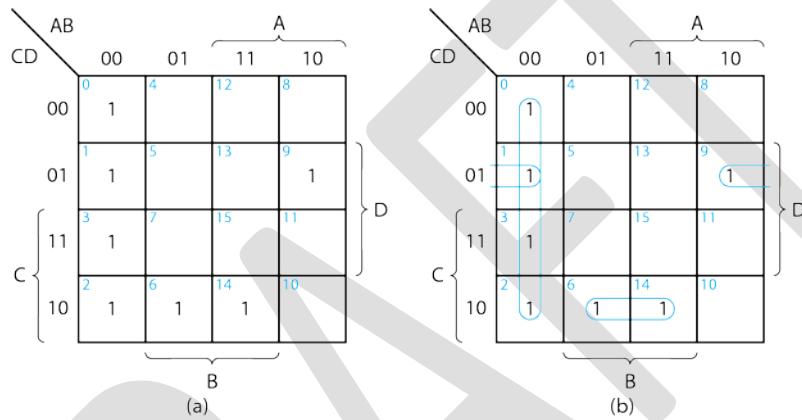


Figure 2.37: K-map of $\bar{f}(A, B, C, D)$ (a) Minterms of the function. (b) Minimal cover.

Using Procedure 2.1, we obtain the minimal cover $\{1-9, 6-14, 0-1-2-3\}$. Note that the order in which terms are considered and the resulting cover are the same as in the earlier example, except that here the cover represents prime implicants of \bar{f} , rather than prime implicants of f .

We now write the minimal SOP expression for \bar{f} from the prime implicants in the minimal cover

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

Finally, we complement the expression using DeMorgan's theorem to obtain the minimum POS form of $f(A, B, C, D)$.

$$\begin{aligned} f(A, B, C, D) &= \overline{\bar{A}\bar{B}} + \overline{\bar{B}\bar{C}D} + \overline{B\bar{C}\bar{D}} \\ &= (\overline{\bar{A}\bar{B}})(\overline{\bar{B}\bar{C}D})(\overline{B\bar{C}\bar{D}}) \\ &= (A + B)(B + C + \bar{D})(\bar{B} + \bar{C} + D) \end{aligned}$$

The primary advantage of this method is that we can use the SOP rules for K-map simplification on both SOP and POS functions. However, the additional steps of complementing

the expression and applying DeMorgan's theorem are required to produce a minimum POS form. The following examples illustrate both methods.

Example – Find a minimal POS expression for $f(A, B, C, D) = \prod M(3, 4, 6, 8, 9, 11, 12, 14)$.

The maxterms of the function are plotted on the K-map of Figure 2.38a. Using Procedure 2.2, we first cover maxterm M_3 with prime implicate 3-11, M_4 with 4-6-12-14, and then M_8 with 8-9. The resulting minimal cover is shown in Figure 2.38a. The minimal POS expression is derived by writing the product of the sum terms corresponding to the prime implicants, where 3-11 represents $(B + \bar{C} + \bar{D})$, 4-6-12-14 represents $(\bar{B} + D)$, and 8-9 represents $(\bar{A} + B + C)$. The resulting POS expression is

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

To use Procedure 2.3, we plot the complement of the function, as shown in Figure 2.38b, and then form the same combination of cells: 3-11, 4-6-12-14, and 8-9. Writing the sum of these prime implicants,

$$\bar{f}(A, B, C, D) = \bar{B}\bar{D} + \bar{B}CD + A\bar{B}\bar{C}$$

Complementing the expression using DeMorgan's theorem gives the same final result,

$$\begin{aligned} f(A, B, C, D) &= \overline{\bar{B}\bar{D} + \bar{B}CD + A\bar{B}\bar{C}} \\ &= (\overline{\bar{B}\bar{D}})(\overline{\bar{B}CD})(\overline{A\bar{B}\bar{C}}) \\ &= (\bar{B} + D)(B + \bar{C} + \bar{D})(\bar{A} + B + C) \end{aligned}$$

Example – Derive a minimum POS expression for the function

$$f(A, B, C, D, E) = \prod M(0, 2, 4, 11, 14, 15, 16, 20, 24, 30, 31)$$

The function and its minimum cover are plotted on the K-map of Figure 2.39.

The minimal cover contains five prime implicants, all of which are essential. Writing these prime implicants as sums and taking their product, we get

$$\begin{aligned} f(A, B, C, D, E) &= (A + B + C + E)(B + D + E)(\bar{B} + \bar{C} + \bar{D}) \\ &\quad (A + \bar{B} + \bar{D} + \bar{E})(\bar{A} + C + D + E) \end{aligned}$$

The reader should verify that the same result is obtained by finding a minimal SOP expression for \bar{f} and then complementing it to obtain f .

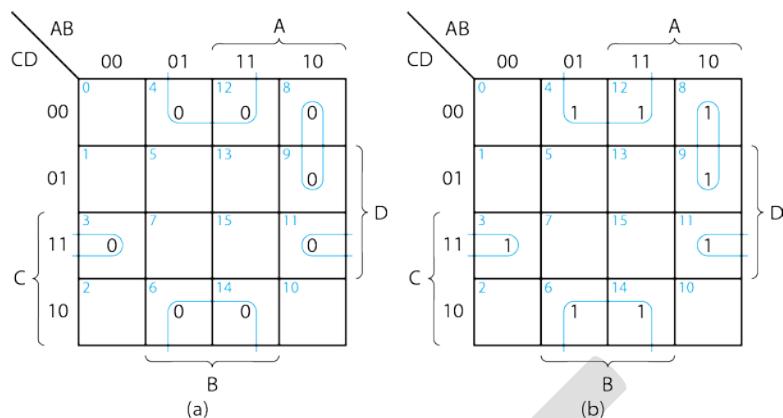


Figure 2.38: Minimum covers of $f(A, B, C, D) = \prod M(3,4,6,8,9,11,12,14)$ and its complement. (a) $f(A, B, C, D)$. (b) $\bar{f}(A, B, C, D)$.

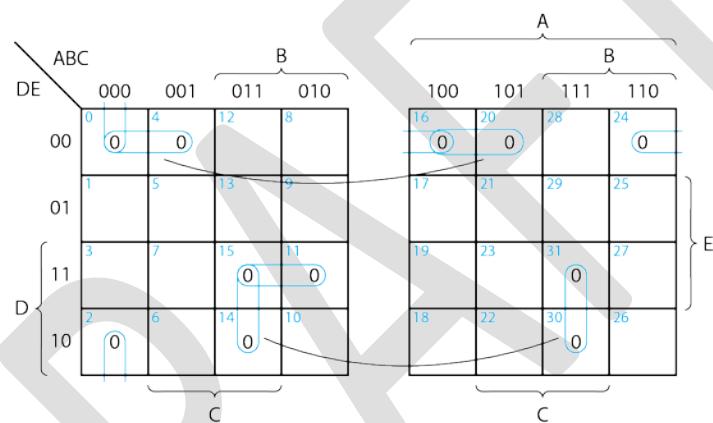


Figure 2.39: Finding a minimum POS expression for a 5-variable function.

Incompletely Specified Functions – When don't-care terms are present, we adjoin one additional rule to those previously discussed for minimizing functions via maps. Recall that don't-cares by definition can be either 0 or 1. Hence, when deriving MSOP or MPOS expressions, we choose the don't-cares to be 1 or 0 if, in doing so, the set of cells on the map that can be grouped together is larger than would otherwise be possible without including the don't-cares. Then, when deriving the minimal cover, we ignore the don't-cares and select only enough prime implicants/implicates to cover the minterms or maxterms. In other words, with regard to don't-cares, we can take them (or leave them) depending on whether they do (or do not) aid in the minimization of a function. This approach will now be illustrated with several examples.

Example – Minimize the following function in both SOP and POS forms using K-maps.

$$\begin{aligned} f(A, B, C, D) &= \sum m(1, 3, 4, 7, 11) + d(5, 12, 13, 14, 15) \\ &= \prod M(0, 2, 6, 8, 9, 10) \cdot D(5, 12, 13, 14, 15) \end{aligned}$$

The maps for the function $f(A, B, C, D)$ are shown in Figure 2.40a and b.

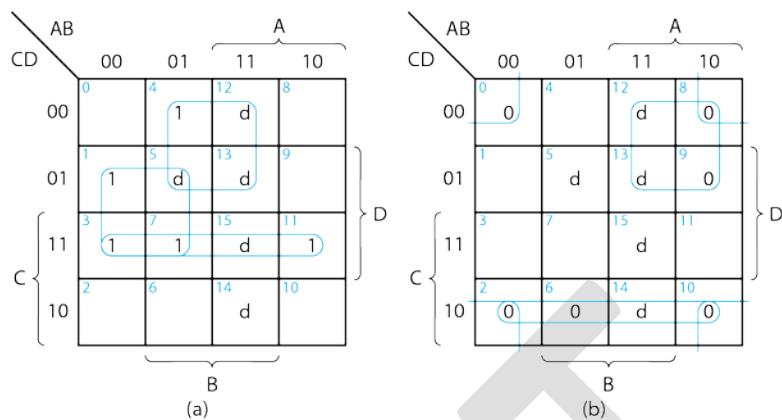


Figure 2.40: K-maps for Example 3.22. (a) Sum of products. (b) Product of sums.

The minimum SOP form derived from the map of Figure 2.40a is

$$f(A, B, C, D) = BC + \bar{A}D + CD$$

The minimum POS form derived from the map of Figure 2.40b is

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

Note that, if we use Boolean algebra on the minimum POS form to produce a SOP form, we get

$$f(A, B, C, D) = \bar{A}\bar{B}\bar{C} + \bar{A}D + CD$$

which is not identical to the minimum SOP form obtained from the map. This situation can occur because of the presence of don't-cares, which can be used differently when optimizing each derived expression. In this case, several don't-care terms (5, 12, 13) were used as ones to derive the minimum SOP form and as zeros to derive the minimum POS form. However, in each case, the *required* minterms and maxterms have been covered correctly. The following example will serve to illustrate how don't-cares occur and how they are used.

Example – Design a 4-bit binary-coded decimal (BCD) input/single output logic circuit that will be used to distinguish digits that are greater than or equal to 5 from those that are less than 5. The input will be the BCD representation of the decimal digits 0, 1, ..., 9, and the output should be 1 if the input is greater than or equal to 5 and 0 if the input is less than 5.

The block diagram of the circuit is shown in Figure 2.41a, and the truth table for this operation is shown in Figure 2.41b.

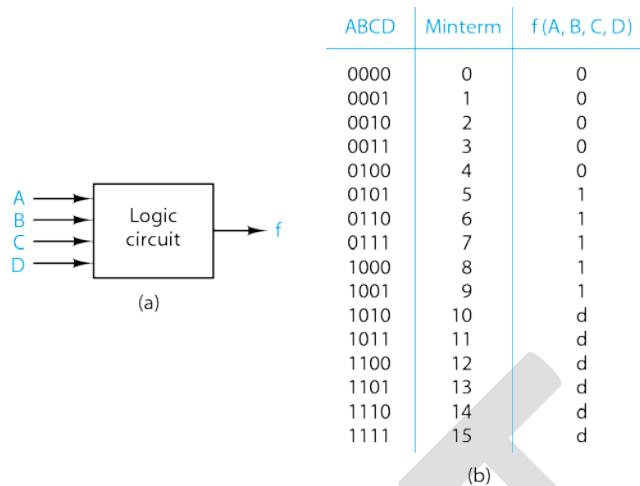


Figure 2.41: Example 3.23 block diagram and truth table. (a) Block diagram. (b) Truth table.]

Note that the don't cares appear in the table because these particular inputs do not represent BCD digits and hence cannot possibly occur. Therefore the output function f is

$$f(A, B, C, D) = \sum m(5, 6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)$$

This function is plotted on the map of Figure 2.42a.

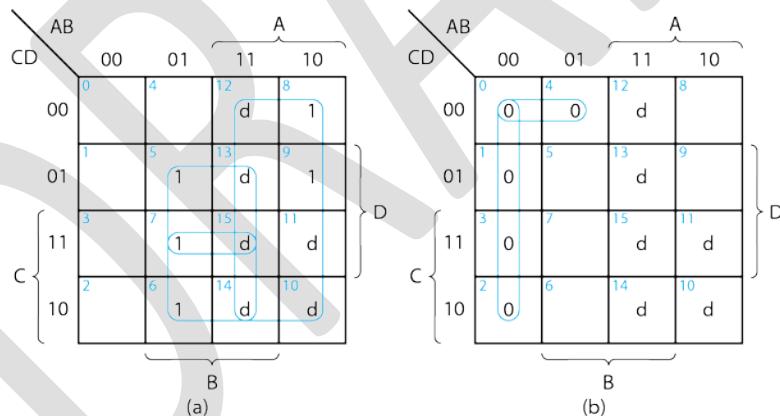


Figure 2.42: Use of don't cares for SOP and POS forms. (a) Minimum SOP cover. (b) Minimum POS cover.

From the map we obtain the minimum SOP form

$$f(A, B, C, D) = A + BD + BC$$

The reader should verify that if we combine maxterms and don't-cares, as shown in Figure 2.42b, we can obtain the minimum POS form

$$f(A, B, C, D) = (A + B)(A + C + D).$$

Note that this function is much simpler than it would have been without the inclusion of the don't-cares. In addition, note in Figure 2.42a that all the don't-cares were used, that is, chosen to be ones, whereas in Figure 2.42b none of the don't cares were used, that is, chosen to be zeros. This will not always be the case.

2.4.4 Quine-McCluskey Method

The *Quine-McCluskey (Q-M) method* is a tabular approach to Boolean function minimization [5, 6, 7]. Basically, the Q-M method has two advantages over the K-map. It is a straightforward, systematic method for producing a minimal function that is less dependent on the designer's ability to recognize patterns than the K-map method. Also, the method is a viable scheme for handling a large number of variables as opposed to the K-map, which, practically, is limited to about five or six variables. First, the Q-M method performs an ordered search over the minterms of a function to find all its prime implicants. Then, a minimal cover of the prime implicants is found yielding the function's MSOP.

The Quine-McCluskey method begins with a list of the n -variable minterms of the function and successively derives all implicants of $n-1$ variables, then implicants of $n-2$ variables, and so on, until all prime implicants are identified. A minimal covering of the function is then derived from the set of all prime implicants. The four steps of the process are described below. Each step will be illustrated by the examples that follow.

Step 1 – List in a column all the minterms of the function to be minimized in their binary representation. Partition them into groups according to the number of 1 bits in their binary representations. This partitioning simplifies identification of logically adjacent minterms since, to be logically adjacent, two minterms must differ in exactly one literal, and therefore the binary representation of one minterm must have either one more or one fewer 1 bit than the other.

Step 2 – Perform an exhaustive search between neighboring groups for adjacent minterms and combine them into a column of $(n-1)$ -variable implicants, checking off each minterm that is combined. The binary representation of each new implicant contains a dash in the position of the eliminated variable. Repeat for each column, combining $(n-1)$ -variable implicants into $(n-2)$ -variable implicants, and so on, until no further implicants can be combined. Any term not checked off represents a prime implicant of the function, since it is not covered by a larger implicant. The final result is a list of prime implicants of the switching function.

Step 3 – Construct a prime implicant chart that lists minterms along the horizontal and prime implicants along the vertical, with an \times entry placed wherever a certain prime implicant (row) covers a given minterm (column).

Step 4 – Select a minimum number of prime implicants that cover all the minterms of the switching function.

A complete example will now be presented that demonstrates these four steps.

Example – Use the Q-M method to minimize the function

$$f(A, B, C, D) = \sum m(2, 4, 6, 8, 9, 10, 12, 13, 15)$$

Step 1 – To begin the Q-M minimization technique, the minterms are grouped according to the number of ones in the binary representation of the minterm number. This grouping of terms is illustrated in the following table.

Minterms	ABCD	Comments
2	0010	
4	0100	Group 1 (a single 1)
8	1000	
6	0110	
9	1001	Group 2 (two 1's)
10	1010	
12	1100	
13	1101	Group 3 (three 1's)
15	1111	Group 4 (four 1's)

Step 2 – Once this table has been formed, an exhaustive search for all combinations of logically adjacent terms is initiated. The method of performing this functional reduction is summarized here and explained in detail later. Consider the minimizing table shown next containing the three minterm lists. The two terms can be combined if and only if they differ in a single literal. Hence, in list 1 we can combine terms in group 1 only with those in group 2. When all the combinations between these two groups have been made and they have been entered in list 2, a line is drawn under these combinations, and we begin combining the terms in group 2 with those in group 3. This simple procedure is repeated from one list to another in order to generate the entire minimizing table.

Minterm	List 1		List 2			List 3		
	ABCD	Covered	ABCD	Covered	ABCD	Covered	ABCD	Covered
2	0010	✓	2,6	0-10	PI ₂		8,9,12,13	
4	0100	✓	2,10	-010	PI ₃			
8	1000	✓	4,6	01-0	PI ₄			
6	0110	✓	4,12	-100	PI ₅			
9	1001	✓	8,9	100-	✓			
10	1010	✓	8,10	10-0	PI ₆			
12	1100	✓	8,12	1-00	✓			
13	1101	✓	9,13	1-01	✓			
15	1111	✓	12,13	110-	✓			
			13,15	11-1	PI ₇			

There are a number of items in the table that beg for explanation. Note that the first element in list 2 indicates that minterms 2 and 6 have been combined since they differ in only a single literal. The terms differed in the variable *B* and hence a dash appears in that position in

the combination 2, 6, indicating that variable B was eliminated when the two minterms were combined. This combination can easily be checked by Boolean algebra.

$$\text{minterm2} = \overline{A}\overline{B}C\overline{D}, \quad \text{minterm6} = \overline{A}B\overline{C}\overline{D}$$

and

$$\overline{A}\overline{B}C\overline{D} + \overline{A}B\overline{C}\overline{D} = \overline{A}C\overline{D} \Rightarrow 0-10$$

Each minterm in list 1 that is combined with another is checked off with a \checkmark , indicating that it has been included in a larger set. Although a term may be combined more than once, it is only checked off once.

Once list 2 has been generated from list 1, an exhaustive search is made to combine the terms in list 2 to generate list 3. It is at this point that it becomes evident why it is important to indicate which of the variables has been eliminated. Since, as before, two terms in list 2 can be combined only if they differ in a single literal, only terms that have the same missing literal (a dash in the same position) can possibly be combined. Note that in list 2 minterm combinations 8, 12 and 9, 13 and also 8, 9 and 12, 13 can be combined to yield the combination 8, 9, 12, 13 in list 3. Inspection of list 2 shows that minterm combinations 8, 12 and 9, 13 both have the same missing literal and differ by one other literal. The same is true for the other combination. Hence all four terms are checked off in list 2 in the table. No other terms in list 2 can be combined. Hence, all the terms that are not checked off across the entire table are prime implicants and are labeled PI₁... PI₇. The function could now be realized as a sum of all the prime implicants; however, we are looking for a minimal realization, and hence we want to use the fewest number of prime implicants that cover all minterms.

Step 3 – To find a minimal cover, we employ a prime implicant chart as shown below. Each column corresponds to a minterm and each row corresponds to a prime implicant found in Step 2. An X indicates that the prime implicant covers the minterm. The bold horizontal line between PI₁ and PI₂ separates prime implicants that contain different numbers of literals.

	2	4	6	8	9	10	12	13	15
**PI ₁				X	☒		X	X	
PI ₂	X		X						
PI ₃	X					X			
PI ₄		X	X						
PI ₅		X					X		
PI ₆				X		X			
**PI ₇							X	☒	

Step 4 – An examination of the minterm columns in the prime implicant chart indicates that minterms 9 and 15 are each covered by only one prime implicant (shown encircled). Therefore, prime implicants 1 and 7 must be chosen, and hence they are essential prime implicants (as indicated by the double asterisks). Note that in choosing these two prime implicants we have

also covered minterms 8, 12, and 13. All five of the covered minterms are checked off in the table.

The problem now remaining is that of selecting as few additional (nonessential) prime implicants as are necessary to cover the minterms 2, 4, 6, and 10. This can be approached by forming a reduced prime implicant chart by deleting rows corresponding to previously selected PIs and columns corresponding to previously covered minterms. The reduced chart is shown below.

	✓	✓	✓	✓
2	2	4	6	10
PI ₂	X		X	
*PI ₃	X			X
*PI ₄		X	X	
PI ₅		X		
PI ₆				X

Which remaining PIs should we select? Selecting PI₂ would require two additional PIs to be selected, either PI₄ and PI₆ or PI₅ and PI₆. Another option would be to select PI₃ and PI₄ which would cover all four minterms with only two PIs. The single asterisk indicates our selection, and the checks above all the remaining minterms mean we have generated a complete cover. Therefore, a minimal sum of products of the original function would be the following.

$$\begin{aligned}
 f(A, B, C, D) &= PI_1 + PI_3 + PI_4 + PI_7 \\
 &= 1-0- + -010 + 01-0 + 11-1 \\
 &= AC' + B'CD' + A'BD' + ABD
 \end{aligned}$$

The same result could be found using a K-map as shown in Figure 2.43.

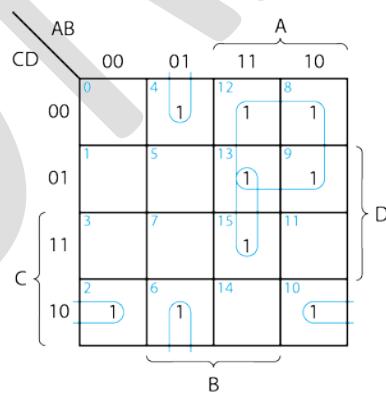


Figure 2.43: Grouping of terms.

Covering Procedure – The problem of selecting a minimum number of prime implicants to realize a switching function is called the *covering problem*. The following procedure can be employed to systematically choose a minimum number of nonessential prime implicants from the prime implicant chart.

First remove all essential prime implicant rows, as well as the minterm columns that they cover, from the chart, as in the last example. Consider the PI chart below for the switching function $f(A, B, C, D) = \sum m(0, 1, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15)$.

	v 0	v 1	v 5	v 6	v 7	v 8	v 9	v 10	v 11	v 13	v 14	v 15
**PI ₁	(X)	X				X	X					
PI ₂		X	X				X			X		
PI ₃			X		X					X		X
PI ₄						X	X		X			
PI ₅							X	X	X	X		X
PI ₆									X		X	X
**PI ₇				(X)	X			X			X	X

For this chart, PI₁ and PI₇ are essential PIs as marked with double asterisks. Deleting these two rows as well as all columns in which the rows have \times entries results in the following reduced PI chart.

	5	10	11	13
PI ₂	X			X
PI ₃	X			X
PI ₄		X	X	
PI ₅			X	X
PI ₆		X	X	

This chart can be further reduced using the concept of row and column covering. A row (column) i of a PI chart *covers* row (column) j if row (column) i contains an \times in each column (row) in which j contains an \times . Each row represents a nonessential prime implicant PI _{i} , while each column represents a minterm m_i of the switching function. According to the definition of row and column covering stated earlier, row PI₂ covers row PI₃ (and vice versa), row PI₄ covers row PI₆ (and vice versa), column 11 covers column 10, and column 13 covers column 5.

The following rules capture the concept and when applied yield the table below.

Rule 1. Delete all rows *covered* by another row. When identical rows are present, all but one of the rows may be deleted. In the example, rows PI₃ and PI₆ may be deleted.

Rule 2. Delete all columns that *cover* another column may be deleted. All but one column from a set of identical columns may be deleted. In the example, columns 11 and 13 can be deleted.

	v	v
5		10
*PI ₂	X	
*PI ₄		X

Hence we choose PI₂ and PI₄ along with the essential PI₁ and PI₇ to obtain a minimum cover for the function.

A *cyclic* PI chart is a chart that contains no essential PI and that cannot be reduced by rules 1 and 2. An example of a cyclic chart is shown next for the switching function

$$f(A, B, C) = \sum m(1, 2, 3, 4, 5, 6)$$

	v 1	2	v 3	4	5	6
*PI ₁	X		X			
PI ₂		X	X			
PI ₃		X				X
PI ₄				X		X
PI ₅				X	X	
PI ₆	X				X	

The procedure to follow for cyclic chart reduction is to arbitrarily select one PI from the chart. The row corresponding to this PI and the columns corresponding to the minterms covered by the PI are then deleted. If the resulting reduced chart is not cyclic, then rules 1 and 2 may be applied. However, if another cyclic chart is produced, the procedure for a cyclic chart is repeated and another arbitrary choice is made. For example, arbitrarily choose PI₁ in the preceding cyclic chart. The following noncyclic chart is obtained by deleting row PI₁ and columns 1 and 3.

	2	4	5	6
PI ₂	X			
PI ₃	X			X
PI ₄		X		X
PI ₅		X	X	
PI ₆			X	

Rules 1 and 2 may now be applied to further reduce this chart. PI₃ covers row PI₂; hence row PI₂ may be deleted. Row PI₅ covers row PI₆, so we can delete PI₆. The resulting reduced chart is

	v 2	v 4	v 5	v 6
*PI ₃	(X)			X
PI ₄		X		X
*PI ₅		X	(X)	

PI₃ and PI₅ must be chosen to cover the chart.

So a minimal cover for the logic function is PI₁, PI₃, and PI₅. A different minimal cover would have resulted had we started with PI₆ instead of PI₁.

Incompletely Specified Functions – The minimization of functions involving don't-cares proceeds exactly as shown in the preceding examples with one important exception, which will be demonstrated below.

Example – Use the Q-M approach to minimize the function

$$f(A, B, C, D, E) = m(2,3,7,10,12,15,27) + d(5,18,19,21,23)$$

Following the procedure demonstrated in a previous example, all the minterms *and don't-cares* are listed in the minimizing table and combined in the manner previously illustrated. The results of this procedure are shown in the following table.

Minterm	List 1		List 2			List 3		
	ABCDE	Covered	Minterms	ABCDE	Covered	Minterms	ABCDE	Covered
2	00010	✓	2,3	0001-	✓	2,3,18,19	-001-	PI ₁
3	00011	✓	2,10	01010	PI ₄	3,7,19,23	-0-11	PI ₂
5	00101	✓	2,18	-0010	✓	5,7,21,23	-01-1	PI ₃
10	01010	✓	3,7	00-11	✓			
12	01100	PI ₇	3,19	-0011	✓			
18	10010	✓	5,7	001-1	✓			
7	00111	✓	5,21	-0101	✓			
19	10011	✓	18,19	1001-	✓			
21	10101	✓	7,15	0-111	PI ₅			
15	01111	✓	7,23	-0111	✓			
23	10111	✓	19,23	10-11	✓			
27	11011	✓	19,27	1-011	PI ₆			
			21,23	101-1	✓			

A prime implicant chart for the example function is now constructed as shown below. It is at this point that the method differs from that described earlier. Since some of the terms in list 1 are don't-cares, there is no need to cover them. Only the specified minterms must be covered, and thus the only minterms that appear in the prime implicant chart. In other words, do *not* list don't-cares in the PI chart.

	✓		✓	✓	✓	✓	✓
2	X	X					
PI ₁			X				
PI ₂		X	X				
PI ₃			X				
**PI ₄	X			(X)			
**PI ₅			X			(X)	
**PI ₆							(X)
**PI ₇					(X)		

It can be seen from the chart that PI₄, PI₅, PI₆, and PI₇ are essential prime implicants. Minterm 3 is not covered by the essential prime implicants. It can be covered using PI₁ or PI₂, so there are two minimal covers for this function.

$$f(A, B, C, D, E) = PI_1 + PI_4 + PI_5 + PI_6 + PI_7 \quad \text{or} \quad f(A, B, C, D, E) = PI_2 + PI_4 + PI_5 + PI_6 + PI_7$$

The minimal SOP forms of the function are

$$f(A, B, C, D, E) = \overline{B}\overline{C}D + \overline{A}\overline{C}D\overline{E} + \overline{A}CDE + A\overline{C}DE + \overline{A}BC\overline{D}\overline{E}$$

or $f(A, B, C, D, E) = \overline{B}DE + \overline{A}\overline{C}D\overline{E} + \overline{A}CDE + A\overline{C}DE + \overline{A}BC\overline{D}\overline{E}$

2.5 Summary and Review Questions

This chapter provided an introduction to Boolean algebra and how it relates to logic circuits. The AND, OR, and NOT operations of Boolean algebra can be realized as logic gates and can be interconnected to form logic circuits that realize Boolean functions. Commonly used circuits such as half-adders, full-adders, ripple-carry adders, majority circuits, and parity circuits were introduced as a means to illustrate the concepts more clearly. It was explained that the functionality of logic circuits can be described in several ways including truth tables, logic expressions, Karnaugh maps, and hardware description languages (HDLs).

The postulates and theorems of Boolean algebra were introduced to provide the fundamentals needed to manipulate logic expressions in the various forms needed to describe logic circuits, and vice versa. The concepts of minterms, maxterms, minterm and maxterm lists, canonical sum of products (CSOP), canonical product of sums (CPOS), minimum sum of products (MSOP), and minimum product of sums (MPOS) were introduced. Two techniques, Karnaugh maps and the Quine-McCluskey method, were introduced for finding MSOP and MPOS forms from other functional representations.

Material presented in this chapter is fundamental for the understanding of the concepts and techniques covered in the remainder of the book and should be mastered before moving on. The following review questions will help the reader assess her/his level of knowledge.

1. Can you construct truth tables for the basic logic gates – AND, OR, NOT, NAND, NOR, XOR, and XNOR?
2. Can you construct truth tables for logic functions such as *majority*, *parity*, *sum*, and *carry*?
3. Do you understand the basic concepts of combinational and sequential logic circuits?
4. Can you construct a block (I/O) diagram and truth table from an HDL model of a combinational logic circuit?
5. Can you use the postulates and theorems of Boolean algebra to find canonical sum of products and product of sums forms of a logic function?
6. Can you use the postulates and theorems of Boolean algebra to find minimum sum of products and product of sums forms of a logic function?

7. Can you find the minterms and maxterms of a logic function from a truth table or logic expression?
8. Can you write minterm and maxterm list representations of a logic function?
9. Can you use the postulates and theorems of Boolean algebra to factor and expand logic expressions?
10. Do you understand the concept of a *don't-care*?
11. Can you plot logic functions on Karnaugh maps (K-map)?
12. Can you find the minimum sum of products of a logic function using a K-map?
13. Can you find the minimum product of sums of a logic function using a K-map?
14. Can you find the minimum sum of products of a logic function using the Quine-McCluskey procedure?
15. Can you find the minimum product of sums of a logic function using the Quine-McCluskey procedure?

2.6 Collaboration Exercises

1. Boolean algebra has many applications other than modeling logic circuits. Identify some of them including Boole's original one.
2. An elevator was used, in section 2.1.4, to illustrate the concept of a sequential device or machine. Identify other machines that illustrate the concept.
3. The canonical sum of products of a Boolean function is said to be *unique*. Discuss what this means and why it's important. Repeat for canonical product of sums.
4. Equation 2.6 is used to derive Boolean expressions all possible functions of two variables. Write a similar equation for three variables, A, B, C. Generalize the equation for n -variables. Use the equation to prove there are 2^{2^n} functions of n variables. Describe a procedure for using the equation for finding the canonical sum of products of a Boolean function specified by a list of minterms.
5. A *majority function* of n -variables is defined to have value 1 if a majority of its variables have value 1 and else to have value 0. This definition is ambiguous when n is an even number. Discuss ways that the majority function can be redefined to resolve this ambiguity.
6. Discuss concepts of minterms, prime implicants, essential prime implicants, and how they relate. Give specific examples for a four-variable Boolean function.
7. Explain, using postulates and theorems of Boolean algebra, why K-maps work?
8. Discuss the pros and cons of the stacked structure for five- and six-variable K-maps versus the Gray-code structure.
9. Write a computer program to implement the Quine-McCluskey method.
10. Discuss the concept of *don't cares* in defining Boolean functions. How can they occur and why are they important? Give some examples.

Problems

2.1 Construct truth tables for the following AND and OR gates.



(a)



(b)



(c)



(d)

2.2 Construct truth tables for the following NAND and NOR gates.



(a)



(b)



(c)



(d)

2.3 Construct truth tables for the following XOR and XNOR gates.



(a)



(b)



(c)



(d)

2.4 Prove that the 3-variable *XOR* function is equivalent to the *sum* function of a full-adder.

2.5 Prove that the 3-variable *majority* function is equivalent to the *carry* function of a full-adder.

2.6 Prove that a 3-variable *XOR* function is equivalent to a 3-variable *odd parity* function.

2.7 Construct a truth table for the 3-variable *even parity* function.

2.8 Draw a block (I/O) diagram and construct a truth table for the function defined by the following Verilog model.

```
module problem_2_8 (x, y, z);
    input x, y;
    output z;
    wire a, b, c;
    nand (a, x, y);
    nand (b, a, x);
    nand (c, a, y);
    nand (z, b, c);
endmodule
```

2.9 Repeat problem 2.8 for the following model.

```
module problem_2_9 (A,B,C,f,g);
    input A,B,C;
    output f,g;
    assign f =  $\sim A \& \sim B \& C \mid B \& \sim C \mid A \& B$ ;
    assign g =  $\sim A \& C \mid A \& \sim B \& C \mid A \& B \& \sim C$ ;
endmodule
```

2.10 Draw a block (I/O) diagram and construct a truth table for the function defined by the following VHDL model.

```
entity prob2_10 is
    port (x, y: in bit; z: out bit);
end prob2_10;

architecture structure of prob2_10 is
signal a, b, c: bit;
begin
    process (x,y)
        begin
            a <= x nor y;
            b <= x nor a;
            c <= y nor a;
            z <= a nor b;
        end process;
    end structure;
```

2.11 Repeat problem 2.10 for the following model.

```
entity prob2_11 is
    port (A, B, C: in bit; f, g: out bit);
end prob2_11;

architecture structure of prob2_11 is
begin
    process (x,y)
        begin
```

```

f <= (A not)and(B not) or B and C or A and C;
g <= (A not)and(B not) or (B not)and(C not) or A and B and C;
end process;
end structure;

```

2.12 Construct truth tables for the following functions.

- | | |
|------------------------------------|--|
| (a) $f(a, b, c) = ab + a'c$ | (b) $F(A, B, C, D) = A(B' + CD') + A'BC'D$ |
| (c) $g(a, b, c) = \sum m(1, 4, 5)$ | (d) $h(a, b, c) = \prod M(2, 5, 6, 7)$ |

2.13 Simplify (minimum literals) the following using the postulates and theorems of Boolean algebra.

- | | |
|--------------------------|---------------------------------------|
| (a) $(A'B + AC)(A + B')$ | (b) $x + xyz + x'yz + wx + w'x + x'y$ |
|--------------------------|---------------------------------------|

2.14 Use DeMorgan's theorem to complement the following expressions.

- | | |
|---------------------|-------------------------|
| (a) $XY + AC' + DE$ | (b) $X(Y + Z'(Q + R'))$ |
|---------------------|-------------------------|

2.15 Prove the following using appropriate methods.

- | | |
|---|--|
| (a) $A'C + AB + B'C' = A'B' + BC + AC'$ | (b) $AC' + BC + AB' + B'C' + A'B + AC$ |
|---|--|

2.16 Find canonical SOP and POS forms for the functions in Problems 2.12 (a) and (c).

2.17 Find minterm and maxterm list forms for f and f' as defined in the following truth table.

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

2.18 Using Boolean algebra, simplify each of the following expressions.

- | |
|--|
| (a) $f(w, x, y, z) = x + xyz + \bar{x}yz + wx + \bar{w}x + \bar{x}y$ |
| (b) $f(A, B, C, D, E) = (AB + C + D)(\bar{C} + D)(\bar{C} + D + E)$ |
| (c) $f(x, y, z) = y\bar{z}(\bar{z} + \bar{z}x) + (\bar{x} + \bar{z})(\bar{x}y + \bar{x}z)$ |

2.19 Simplify each of the following logic expressions.

- | |
|--|
| (a) $f(A, B, C, D) = \overline{(A + \bar{C} + D)(\bar{B} + C)(A + \bar{B} + D)(\bar{B} + C)(\bar{B} + C + \bar{D})}$ |
| (b) $f(A, B, C, D) = \overline{AB + \bar{A}\bar{D} + B\bar{D} + \bar{A}B + C\bar{D}A + \bar{A}D + CD + \bar{A}\bar{B}D}$ |

$$(c) f(A, B, C, D) = \overline{A\bar{B}C + AB + \bar{ABC} + A\bar{C} + ABC}$$

$$(d) f(A, B, C) = \overline{(B + \bar{A})(AB + C) + A\bar{B} + \bar{A}\bar{B}C + (A + B)(\bar{A} + C)}$$

$$(e) f(A, B, C) = \overline{(\bar{A} + \bar{B})(A + \bar{A}B)(\bar{A} + \bar{B} + \bar{A}\bar{B}C) + (A + B)(\bar{A} + C)}$$

2.20 Prove part (b) of Theorem 4.

2.21 Prove part (b) of Theorem 5.

2.22 Prove part (b) of Theorem 7 (consensus).

2.23 Simplify (minimum literals) each of the following logic expressions.

$$(a) f(A, X, Z) = \bar{X}(X + Z) + \bar{A} + AZ$$

$$(b) f(X, Y, Z) = (\bar{X}Y + XZ)(X + \bar{Y})$$

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

2.24 Find the simplest logic expression for each of the following functions.

$$(a) f(A, B, C) = \sum m(1, 4, 5)$$

$$(b) f(A, B, C, D) = \prod M(0, 2, 4, 5, 8, 11, 15)$$

$$(c) f(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 13)$$

2.25 Given the function $f(x, y, z) = x\bar{y} + x\bar{z}$ write $f(x, y, z)$ as a sum of minterms and as a product of maxterms.

2.26 Use the postulates and theorems of Boolean algebra to find an MSOP logic expression for each of the following functions.

$$(a) f(a, b, c) = \sum m(1, 4, 5, 6) \quad (b) g(A, B, C, D) = A(B' + CD') + A'BC'D$$

$$(c) h(a, b, c) = \prod M(5, 6, 7)$$

2.27 Use the postulates and theorems of Boolean algebra to find an MPOS logic expression for each of the functions given in problem 2.26.

2.28 Map the functions given in problem 2.26 on K-maps showing both minterms (1s) and maxterms (0s).

2.29 Use K-maps to find MSOP and MPOS expressions for each of the following logic functions.

$$(a) F(x, y, z) = \sum m(0, 1, 3, 4, 6, 7) \quad (b) G(a, b, c, d) = \sum m(0, 1, 2, 3, 7, 8, 9, 12, 13, 14)$$

$$(c) H(a, b, c, d) = \prod M(4, 5, 6, 7, 13, 15)$$

2.30 Use K-maps to find MSOP and MPOS expressions for the complement function (F') from problem 2.29 (a).

2.31 Use K-maps to find MSOP and MPOS expressions for the following incompletely specified functions.

(a) $f(a,b,c,d) = \sum m(1,5,7,9) + d(6,13)$ (b) $g(a,b,c,d) = \prod M(3,6,8,9,14) \cdot D(1,2,4,7,11,12,13)$

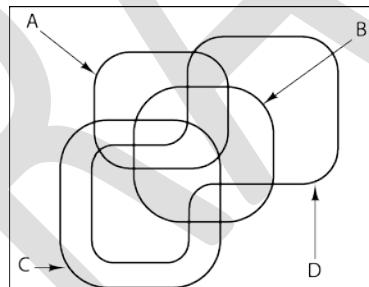
2.32 Use Venn diagrams to determine which of the following switching functions are equivalent.

- (a) $f_1(A,B,C) = A\bar{B}\bar{C} + B + \bar{A}\bar{B}C$
- (b) $f_2(A,B,C) = \bar{A}\bar{B}\bar{C} + B + A\bar{B}C$
- (c) $f_3(A,B,C) = \bar{A}\bar{C} + AC + BC + \bar{A}B$
- (d) $f_4(A,B,C) = A\bar{C} + AB + B\bar{C} + \bar{A}C$

2.33 Sketch the following functions on a Venn diagram.

- (a) $f(A,B) = AB + \bar{A}\bar{B}$
- (b) $f(A,B,C) = AB + \bar{A}\bar{C}$
- (c) $f(A,B,C,D) = A + \bar{B}CD + \bar{A}BD$
- (d) $f(A,B,C,D) = \bar{A}B + C\bar{D}$

Hint: Each new variable is represented by a contour that divides each disjoint segment of the Venn diagram into two segments. A four-variable Venn diagram is shown below.



2.34 Prove that the following expressions are valid using Venn diagrams.

- (a) $A + B = A\bar{B} + \bar{A}B + AB = \bar{\bar{AB}}$
- (b) $A\bar{C} + BC + A\bar{B} \neq \bar{B}\bar{C} + \bar{A}B + AC$
- (c) $\bar{A}\bar{C} + AB + \bar{B}\bar{C} = \bar{A}\bar{B} + BC + AC$
- (d) $AD + A\bar{C}\bar{D} + AB + \bar{A}BD + \bar{A}\bar{B}\bar{C} = AB + BD + AD + \bar{B}\bar{C}$

2.35 Use Theorem 5 to simplify the following expressions:

- (a) $\bar{X} + XABC\bar{C} + \bar{B}C$
- (b) $\bar{XY} + (X + Y)Z$
- (c) $Z(\bar{Z} + AB\bar{C}) + \bar{A}\bar{B}$
- (d) $(\bar{X} + \bar{Y})(XY + Z)$

2.36 Use Theorem 8 (DeMorgan's) to complement the following expressions:

- (a) $X(Y + \bar{Z}(Q + \bar{R}))$
- (b) $X + Y(\bar{Z} + Q\bar{R})$
- (c) $XY + A\bar{C} + IQ$
- (d) $(A + B\bar{C})(\bar{A} + \bar{D}E)$

2.37 Apply Boolean algebra Theorem 7 (consensus) to simplify the following expressions:

- (a) $QR + \bar{X}Q + RX$
- (b) $(X + Y)Z + \bar{X}\bar{Y}W + ZW$
- (c) $(\bar{X} + Y)WZ + X\bar{Y}V + VWZ$
- (d) $(X + Y + Z + \bar{W})(V + X)(\bar{V} + Y + Z + \bar{W})$

2.38 Find truth tables for the following switching functions.

- (a) $f(A, B) = A + \bar{B}$
- (b) $f(A, B, C) = AB + \bar{A}C$
- (c) $f(a, b, c) = \bar{a}\bar{b}c + b\bar{c}$
- (d) $f(a, b, c) = a(b + \bar{c})(\bar{b} + c)$

2.39 Find truth tables for each of the following logic functions.

- (a) $f(A, B, C, D) = A\bar{B}\bar{C}D + A\bar{B}CD$
- (b) $f(A, B, C, D) = AB + \bar{A}\bar{B} + CD$
- (c) $f(A, B, C, D) = A(\bar{B} + \bar{C}\bar{D}) + \bar{A}B\bar{C}D$

2.40 Find the minterm and maxterm list forms for the logic functions of Problem 2.39.

2.41 Find the canonical SOP form for the logic functions of Problem 2.39.

2.42 Expand the following function into canonical SOP form.

$$f(x_1, x_2, x_3) = x_1\bar{x}_3 + x_2\bar{x}_3 + x_1x_2x_3$$

2.43 Expand the following function into canonical POS form.

$$f(W, X, Q) = (Q + \bar{W})(X + \bar{Q})(W + X + Q)(\bar{W} + \bar{X})$$

2.44 Plot each of the following functions on Karnaugh maps.

- (a) $f(A, B, C) = \bar{A}\bar{B} + \bar{B}C + \bar{A}C$
- (b) $f(A, B, C, D) = \bar{B}\bar{C}D + \bar{A}B\bar{C} + ABD$
- (c) $f(A, B, C, D, E) = \bar{B}\bar{C}\bar{E} + \bar{B}CE + \bar{C}\bar{D}E + \bar{A}BCD + ABCDE$

2.45 Minimize each of the following functions using K-maps.

$$(a) f(A, B, C) = \sum m(3, 5, 6, 7)$$

$$(b) f(A, B, C, D) = \sum m(0, 1, 4, 6, 9, 13, 14, 15)$$

$$(c) f(A, B, C, D) = \sum m(0, 1, 2, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$(d) f(A, B, C, D, E) = \sum m(3, 4, 6, 9, 11, 13, 15, 18, 25, 26, 27, 29, 31)$$

$$(e) f(A, B, C, D, E) = \sum m(1, 5, 8, 10, 12, 13, 14, 15, 17, 21, 24, 26, 31)$$

2.46 Minimize the following functions containing don't-cares using the K-map.

$$(a) f(A, B, C, D) = \sum m(2, 9, 10, 12, 13) + d(1, 5, 14)$$

$$(b) f(A, B, C, D) = \sum m(1, 3, 6, 7) + d(4, 9, 11)$$

$$(c) f(A, B, C, D, E) = \sum m(3, 11, 12, 19, 23, 29) + d(5, 7, 13, 27, 28)$$

2.47 Use the K-map to expand the following POS functions to canonical form.

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

$$(b) f(A, B, C, D) = (A + \bar{D})(\bar{A} + C)$$

2.48 Minimize the following functions using a K-map.

$$(a) f(A, B, C, D) = \sum m(3, 4, 6, 8, 9, 12, 14)$$

$$(b) f(A, B, C, D, E) = \sum m(1, 3, 4, 9, 11, 12, 13, 15, 17, 19, 22, 25, 27, 29, 30, 31)$$

2.49 Use K-maps to expand the following switching functions to canonical SOP form.

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

$$(b) f(A, B, C, D) = A\bar{B} + \bar{A}CD + B\bar{C}\bar{D}$$

$$(c) f(A, B, C, D) = (A + \bar{B})(C + \bar{D})(\bar{A} + C)$$

$$(d) f(A, B, C, D, E) = \bar{A}E + BCD$$

2.50 Determine which of the following functions are equivalent.

$$f_1(A, B, C, D) = AC + BD + A\bar{B}\bar{D}$$

$$f_2(A, B, C, D) = A\bar{B}\bar{D} + AB + \bar{A}BC$$

$$f_3(A, B, C, D) = BD + A\bar{B}\bar{D} + ACD + ABC$$

$$f_4(A, B, C, D) = AC + A\bar{B}\bar{C}\bar{D} + \bar{A}BD + B\bar{C}\bar{D}$$

$$f_5(A, B, C, D) = (B + \bar{D})(A + B)(A + \bar{C})$$

2.51 Use a K-map to find the following forms of the logic function below.

(a) Canonical SOP form (b) Canonical POS form

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

2.52 Use K-maps to find the following functions.

- (a) $f_1(A, B, C, D) = f_\alpha(A, B, C, D) \cdot f_\beta(A, B, C, D)$
- (b) $f_2(A, B, C, D) = f_\alpha(A, B, C, D) + f_\beta(A, B, C, D)$
- (c) $f_3(A, B, C, D) = \bar{f}_1(A, B, C, D) \cdot f_2(A, B, C, D)$
- (d) $f_4(A, B, C, D) = f_\alpha(A, B, C, D) \oplus f_\beta(A, B, C, D)$

where

$$\begin{aligned}f_\alpha(A, B, C, D) &= AB + BD + \overline{A}\overline{B}C \\f_\beta(A, B, C, D) &= \overline{A}B + B\overline{D}\end{aligned}$$

2.53 Use K-maps to generate all the prime implicants for the a two output logic circuit that realizes the following functions.

$$\begin{aligned}f_\alpha(A, B, C, D) &= AB + BD + \overline{A}\overline{B}C \\f_\beta(A, B, C, D) &= \overline{A}B + B\overline{D}\end{aligned}$$

2.54 Repeat 2.53 using the multiple-output Quine-McCluskey technique. Compare your prime implicant charts.

2.55 Minimize the following functions using the Quine--McCluskey method.

- (a) $f(A, B, C, D) = \sum m(0, 2, 4, 5, 7, 9, 11, 12)$
- (b) $f(A, B, C, D, E) = \sum m(0, 1, 2, 7, 9, 11, 12, 23, 27, 28)$

2.56 Use the Quine-McCluskey method to minimize the following functions with don't-cares.

- (a) $f(A, B, C, D) = \sum m(0, 6, 9, 10, 13) + d(1, 3, 8)$
- (b) $f(A, B, C, D) = \sum m(1, 4, 7, 10, 13) + d(5, 14, 15)$

2.57 Minimize the following multiple-output functions using the Q--M technique.

$$\begin{aligned}f_\alpha(A, B, C, D) &= \sum m(0, 1, 2, 9, 15) \\f_\beta(A, B, C, D) &= \sum m(0, 2, 8, 12, 15) \\f_\alpha(A, B, C, D) &= \sum m(3, 7, 9, 14) + d(1, 4, 6, 11) \\f_\beta(A, B, C, D) &= \sum m(6, 7, 12) + d(3, 14)\end{aligned}$$

2.58 Apply the covering procedure to obtain a minimum list of prime implicants for the function

$$f(A, B, C, D) = \sum m(1, 3, 4, 6, 7, 9, 13, 15)$$

2.59 Plot the following functions on the K-map and determine the minterm lists.

- (a) $f(A, B, C) = \overline{B} + A\overline{C}$
- (b) $f(A, B, C) = \overline{A}\overline{C} + \overline{A}B + BC$

2.60 Plot the following functions on the K-map and determine the minterm lists.

$$(a) f(A, B, C) = \overline{A}B + BC + AC + A\overline{B}$$

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

2.61 Plot the following functions on the K-map and determine the minterm lists.

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

$$(b) f(A, B, C, D) = \overline{A}\overline{B}\overline{C} + \overline{B}CD + ABD + ABC$$

2.62 Plot the following functions on the K-map and determine the minterm lists.

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

$$(b) f(A, B, C, D) = \overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C} + \overline{A}\overline{C}\overline{D} + BCD + ABC$$

2.63 Plot the following functions on the K-map and determine the minterm lists.

$$(a) f(A, B, C, D, E) = \overline{B}\overline{C}D + \overline{B}DE + \overline{A}BC\overline{D} + BC\overline{D}E + \overline{A}B\overline{D}\overline{E} + B\overline{C}\overline{D}\overline{E} + AB\overline{C}\overline{E}$$

$$(b) f(A, B, C, D, E) = \overline{A}\overline{B}\overline{D}\overline{E} + \overline{A}\overline{B}D + BE + A\overline{B}CD + A\overline{C}DE$$

2.64 Plot the following functions on the K-map and determine the maxterm lists.

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

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

2.65 Plot the following functions on the K-map and determine the maxterm lists.

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

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

2.66 Plot the following functions on the K-map and determine the maxterm list.

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

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

2.67 Plot the following functions on the K-map and determine the maxterm list.

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

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

2.68 Plot the following function on the K-map and determine the maxterm list.

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

$$\cdot (\overline{A} + B + D + E)(\overline{B} + D + \overline{E})$$

2.69 Use the K-map to simplify the following functions.

$$(a) f(A, B, C) = \sum m(1, 5, 6, 7)$$

$$(b) f(A, B, C) = \sum m(0, 1, 2, 3, 4, 5)$$

2.70 Use the K-map to simplify the following functions.

$$(a) f(A, B, C) = \sum m(0, 2, 3, 5)$$

$$(b) f(A, B, C) = \sum m(0, 3, 4, 6, 7)$$

2.71 Find MSOP and MPOS forms of the following functions using a K-map.

$$(a) f(A, B, C, D) = \sum m(0, 2, 5, 7, 8, 10, 13, 15)$$

$$(b) f(A, B, C, D) = \sum m(1, 3, 4, 5, 6, 7, 9, 11, 12, 13, 14, 15)$$

2.72 Use the K-map to simplify the following functions.

$$(a) f(A, B, C, D) = \sum m(0, 4, 5, 7, 8, 10, 11, 15)$$

$$(b) f(A, B, C, D) = \sum m(1, 4, 5, 6, 9, 11, 15)$$

2.73 Find MSOP and MPOS forms of the following functions using a K-map.

$$(a) f(A, B, C, D) = \sum m(1, 2, 5, 6, 7, 9, 11, 15)$$

$$(b) f(A, B, C, D) = \sum m(0, 1, 2, 5, 12, 13, 14, 15)$$

2.74 Use the K-map to simplify the following functions.

$$(a) f(A, B, C, D) = \sum m(1, 4, 5, 6, 8, 9, 11, 13, 15)$$

$$(b) f(A, B, C, D) = \sum m(1, 2, 4, 5, 6, 9, 12, 14)$$

2.75 Find MSOP and MPOS forms of the following functions using a K-map.

$$(a) f(A, B, C, D, E) = \sum m(0, 4, 6, 7, 8, 11, 15, 20, 22, 24, 26, 27, 31)$$

$$(b) f(A, B, C, D, E) = \sum m(2, 7, 10, 12, 13, 22, 23, 26, 27, 28, 29)$$

2.76 Use the K-map to simplify the following functions.

$$(a) f(A, B, C, D, E) = \sum m(1, 3, 8, 9, 11, 12, 14, 17, 19, 20, 22, 24, 25, 27)$$

$$(b) f(A, B, C, D, E) = \sum m(0, 7, 8, 10, 13, 15, 16, 24, 28, 29, 31)$$

2.77 Find MSOP and MPOS forms of the following functions using a K-map.

$$(a) f(A, B, C, D, E) = \sum m(1, 2, 5, 6, 13, 15, 16, 18, 22, 24, 29)$$

$$(b) f(A, B, C, D, E) = \sum m(1, 7, 9, 12, 14, 15, 16, 23, 24, 28, 30)$$

2.78 Use the K-map to simplify the following functions.

$$(a) f(A, B, C, D, E) = \sum m(0, 5, 10, 11, 13, 15, 16, 18, 29, 31)$$

$$(b) f(A, B, C, D, E) = \sum m(4, 5, 7, 8, 9, 12, 13, 16, 18, 23, 24, 25, 28, 29)$$

2.79 Find the minimum POS form for the following functions.

$$(a) f(A, B, C) = \prod M(0, 2, 3, 4)$$

$$(b) f(A, B, C) = \prod M(0, 3, 4, 7)$$

2.80 Find the minimum POS form for the following functions.

(a) $f(A, B, C) = \prod M(0, 1, 4, 5, 6)$

(b) $f(A, B, C) = \prod M(1, 2, 3, 6)$

2.81 Find the minimum POS form for the following functions.

(a) $f(A, B, C) = \prod M(1, 2, 5, 7)$

(b) $f(A, B, C) = \prod M(1, 2, 3, 4)$

2.82 Find the minimum POS form for the following functions.

(a) $f(A, B, C) = \prod M(0, 1, 3, 4, 6, 7)$

(b) $f(A, B, C) = \prod M(2, 3, 5, 7)$

2.83 Find the MSOP and MPOS forms of the following functions.

(a) $f(A, B, C, D) = \prod M(0, 1, 5, 7, 8, 10, 11, 15)$

(b) $f(A, B, C, D) = \prod M(0, 1, 2, 4, 6, 7, 8, 10, 14)$

2.84 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D) = \prod M(2, 3, 4, 5, 7, 12, 13)$

(b) $f(A, B, C, D) = \prod M(1, 2, 5, 7, 11, 13, 15)$

2.85 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D) = \prod M(0, 2, 4, 5, 6, 9, 11, 13)$

(b) $f(A, B, C, D) = \prod M(1, 3, 4, 5, 6, 9, 11, 12, 13)$

2.86 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D) = \prod M(0, 1, 5, 7, 9, 11, 12, 14)$

(b) $f(A, B, C, D) = \prod M(3, 4, 5, 7, 8, 9, 10)$

2.87 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D, E) = \prod M(3, 4, 6, 13, 15, 16, 19, 24, 29, 31)$

(b) $f(A, B, C, D, E) = \prod M(1, 4, 7, 9, 15, 17, 20, 22, 25, 30)$

2.88 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D, E) = \prod M(0, 1, 2, 5, 7, 8, 10, 15, 17, 21, 22, 24, 26, 29)$

(b) $f(A, B, C, D, E) = \prod M(0, 2, 4, 6, 9, 11, 13, 15, 16, 19, 20, 25, 27, 29, 31)$

2.89 Find the minimum SOP form for the following functions.

(a) $f(A, B, C, D) = \sum m(1, 2, 7, 12, 15) + d(5, 9, 10, 11, 13)$

(b) $f(A, B, C, D) = \sum m(0, 2, 5, 15) + d(8, 9, 12, 13)$

2.90 Determine the minimum SOP form for the following functions.

(a) $f(A, B, C, D) = \sum m(4, 7, 9, 15) + d(1, 2, 3, 6)$

(b) $f(A, B, C, D) = \sum m(0, 2, 3, 4, 5) + d(8, 9, 10, 11)$

2.91 Find the minimum SOP form for the following function.

$$f(A, B, C, D, E) = \sum m(7, 9, 12, 13, 19, 22) + d(0, 3, 20, 25, 27, 28, 29)$$

2.92 Determine the minimum POS form for the following functions.

(a) $f(A, B, C, D) = \prod M(4, 7, 9, 11, 12) \cdot D(0, 1, 2, 3)$

(b) $f(A, B, C, D) = \prod M(0, 3, 7, 12) \cdot D(2, 10, 11, 14)$

2.93 Find the minimum POS form for the following functions.

(a) $f(A, B, C, D) = \prod M(3, 4, 10, 13, 15) \cdot D(6, 7, 14)$

(b) $f(A, B, C, D) = \prod M(0, 7, 11, 13) \cdot D(1, 2, 3)$

2.94 Find the minimum POS form for the following function.

$$f(A, B, C, D, E) = \prod M(0, 5, 6, 9, 21, 28, 31) \cdot D(2, 12, 13, 14, 15, 25, 26)$$

2.95 Use the Quine–McCluskey method to minimize the following functions.

(a) $f(A, B, C, D) = \sum m(0, 2, 3, 5, 7, 11, 12, 14, 15)$

(b) $f(A, B, C, D) = \sum m(0, 1, 6, 8, 9, 13, 14, 15)$

2.96 Use the Quine–McCluskey method to minimize the following functions.

(a) $f(A, B, C, D) = \sum m(1, 4, 5, 6, 8, 9, 10, 12, 14)$

(b) $f(A, B, C, D) = \sum m(4, 5, 6, 8, 11, 13, 15)$

2.97 Minimize the following functions using the Quine–McCluskey method.

(a) $f(A, B, C, D) = \sum m(1, 3, 6, 7, 8, 9, 12, 14)$

(b) $f(A, B, C, D) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$

2.98 Use the Quine–McCluskey method to minimize the following functions with don't-cares.

(a) $f(A, B, C, D) = \sum m(1, 6, 7, 9, 12) + d(8, 11, 15)$

(b) $f(A, B, C, D) = \sum m(7, 8, 13, 15) + d(3, 4, 10, 14)$

2.99 Minimize the following functions with don't-cares using the Quine–McCluskey method.

(a) $f(A, B, C, D) = \sum m(5, 7, 11, 12, 27, 29) + d(14, 20, 21, 22, 23)$

(b) $f(A, B, C, D) = \sum m(1, 4, 6, 9, 14, 17, 22, 27, 28) + d(12, 15, 20, 30, 31)$

2.100 Minimize the following multiple-output function using the Quine–McCluskey method.

$$f_{\alpha}(A, B, C, D, E) = \sum m(0, 2, 8, 9, 20, 24) + d(4, 10, 14, 26, 30)$$

$$f_{\beta}(A, B, C, D, E) = \sum m(3, 4, 8, 11, 24) + d(10, 14, 20, 26, 30)$$

DRAFT

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, and Irwin

Chapter 3 Combinational Logic Circuit Design and Analysis

(6/15/2016)

Table of Contents

3.0 Learning Objectives	1
3.1 Design of Combinational Logic Circuits	1
3.1.1 AND-OR and NAND-NAND Circuits	2
3.1.2 OR-AND and NOR-NOR Circuits	3
3.1.3 Two-level Circuits	5
3.1.4 Multi-Level Circuits and Factoring	8
3.1.5 EXCLUSIVE OR (XOR) Circuits	11
3.2 Analysis of Combinational Circuits	14
3.2.1 Boolean Algebra	14
3.2.2 Truth Tables	17
3.2.3 Timing Diagrams	19
3.2.4 Positive and Negative Logic	25
3.3 Design Using Higher-Level Devices	26
3.3.1 Decoders	26
3.3.2 Encoders	39
3.3.3 Multiplexers and Demultiplexers	45
3.3.4 Arithmetic Circuits	55
3.4 Summative Design Examples	69
3.4.1 Design Flow	69
3.4.2 Bank Security-Lock Controller	70
3.4.3 Seven-Segment Display Decoder	74
3.4.4 Four-Function Arithmetic Logic Unit	81
3.4.5 Binary Array Multiplier	87
3.5 Summary and Review Questions	91
3.6 Collaboration Exercises	93
3.7 Problems	95

This page intentionally left blank.

DRAFT

3 Combinational Logic Circuit Design and Analysis

(6/6/2016)

The essential concepts and tools needed to analyze and design combinational logic circuits were presented in Chapters 1 and 2. Techniques and methods for using these concepts and tools in analysis and design processes will be covered in this chapter. The material is intended to be independent of any specific implementation technology to be used in the realization of digital circuits. However, some examples use standard logic components to make the examples more meaningful. Also, standard logic components are still found in the parts libraries of most computer aided design tools and are therefore useful in the design process even when they are not used to implement the final designs. Being able to understand and maintain legacy systems is another reason to study standard components. See Appendix A for a survey of electronic technologies commonly used to implement digital logic circuits. Verilog and VHDL models are also included for most of the examples. See Appendices B and C for the specifics of Verilog and VHDL, respectively.

3.0 Learning Objectives

Students successfully completing this chapter will have gained the knowledge and/or skills necessary to demonstrate the following.

1. Design combinational logic circuits using basic gates – AND, OR, NOT, NAND, NOR, XOR and AOI.
2. Analyze combinational logic circuits realized with basic gates.
3. Design combinational logic circuits using higher-level components such as encoders, decoders, multiplexers, and adders.
4. Analyze combinational logic circuits realized with higher-level components.
5. Design combinational logic functions using hardware description languages such as Verilog and VHDL.
6. Analyze combinational logic functions described by Verilog or VHDL.
7. Understand propagation delay and timing diagrams.
8. Understand hierarchical design and how to use it.

3.1 Design of Combinational Logic Circuits

Combinational circuit design is the process of turning logic requirements or specifications in to a realization consisting of an interconnection of gates, PLDs, and/or other circuit elements. Requirements may be given as word descriptions, Boolean expressions,

truth tables, timing diagrams, block diagrams, HDLs, and/or other means. This section will be devoted to the design of combinational circuits composed of basic gates – AND, OR, NOT, NAND, and NOR.

3.1.1 AND-OR and NAND-NAND Circuits

A logic expression written in sum of products (SOP) form can easily be realized with an AND-OR circuit with each product term realized by an AND gate and an OR gate used to sum the product terms to produce the output. For example, the function

$$f_d(p, q, r, s) = p\bar{r} + qrs + \bar{p}s$$

can be implemented with AND-OR logic as shown in Fig. 3.1(a).

An SOP expression can be realized with a NAND-NAND circuit by replacing each gate in an AND-OR circuit with a NAND gate. This is illustrated in Fig. 3.1(b) and 3.1(c). The validity of this replacement is described below.

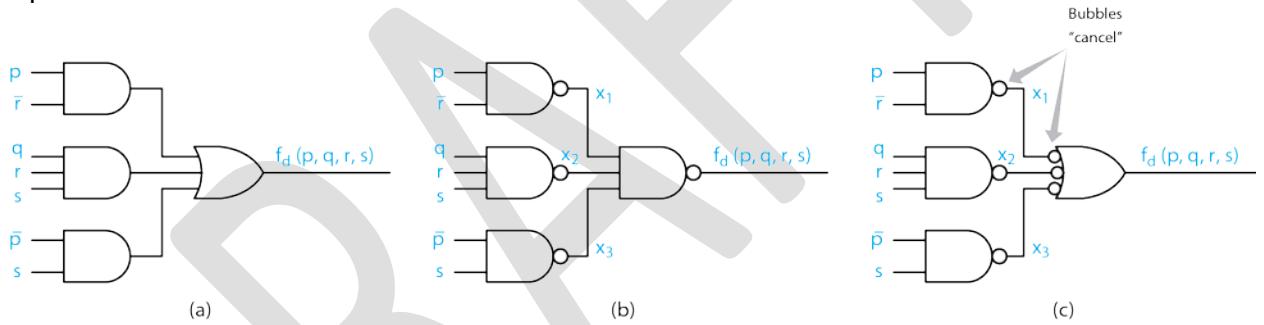


Figure 3.1 Implementations of $f_d(p, q, r, s) = p\bar{r} + qrs + \bar{p}s$. a AND-OR circuit. b NAND circuit. c Alternate NAND circuit.

A simple translation using Boolean algebra may be employed to transform a sum of products expression into an appropriate form for direct NAND implementation. Place two bars over the entire SOP function; then use DeMorgan's theorem (Theorem 8) to find the NAND form for the function

$$\begin{aligned} f_d(p, q, r, s) &= \overline{\overline{p\bar{r}} + \overline{qrs} + \overline{\bar{p}s}} & [T3] \\ &= \overline{\overline{p\bar{r}} \cdot \overline{qrs} \cdot \overline{\bar{p}s}} & [T8(a)] \\ &= \overline{x_1 \cdot x_2 \cdot x_3} \end{aligned}$$

where $x_1 = \overline{p\bar{r}}$, $x_2 = \overline{qrs}$, and $x_3 = \overline{\bar{p}s}$. The NAND realization of this function is shown in Fig. 3.1b.

Fig. 3.1c presents the same circuit, but with the output NAND gate shown in its DeMorgan-equivalent form. Writing the logic expression for the output,

$$\begin{aligned} f_d(p, q, r, s) &= \overline{x_1} + \overline{x_2} + \overline{x_3} \\ &= \overline{\overline{p\bar{r}}} + \overline{\overline{qrs}} + \overline{\overline{\bar{p}s}} \end{aligned}$$

$$= p\bar{r} + qrs + \bar{p}s$$

Note that the inversion bubbles on both ends of lines x_1 , x_2 , and x_3 in Fig. 3.9c effectively cancel each other, making the diagram equivalent to that of Fig. 3.1a, and thus clearly illustrating that a sum of products form is being implemented.

DeMorgan's theorem can also be used to show that both of the symbols below represent NAND gates. In the first case, $Y = (AB)'$. In the second, $Y = A' + B'$. However, DeMorgan's theorem demonstrates that $(AB)' = A' + B'$. So the two expressions are logically equivalent which proves that both symbols represent NAND gates and can be used interchangeably. This can be generalized to any number of input variables.



SOP expressions with one or more single-literal product terms require special attention when converting from AND-OR to NAND-NAND realizations. For example, the function

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

has the AND-OR realization shown in Figure 3.2(a) and the NAND-NAND realization shown in Figure 3.2(b). Note that the x variable is uncomplemented in the AND-OR circuit and complemented in the NAND-NAND. This is necessary to offset the not-bubble on the corresponding NAND gate input.

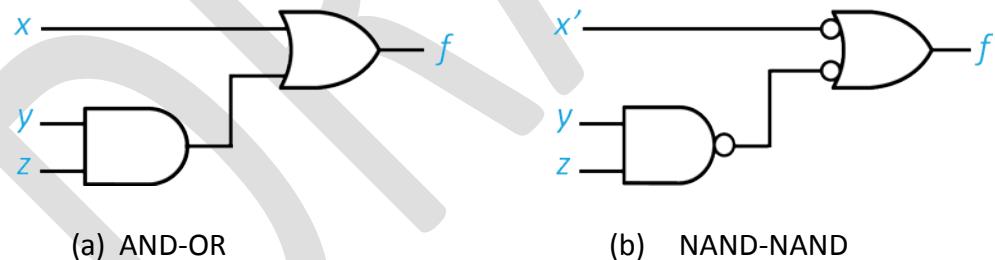


Figure 3.2 – Handling Single-Literal Product Terms

The techniques employed here may be used on any sum of products function to derive an AND-OR or NAND-NAND realization. It is not necessary to perform the algebraic exercise each time once you understand it clearly.

3.1.2 OR-AND and NOR-NOR Circuits

An OR-AND circuit employs OR gates to form sum terms and an AND gate to form the product of these sums. Therefore, logic expressions in product of sums (POS) form can easily be realized in OR-AND form. For example, the function

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

is realized in OR-AND logic as shown in Fig. 3.3a.

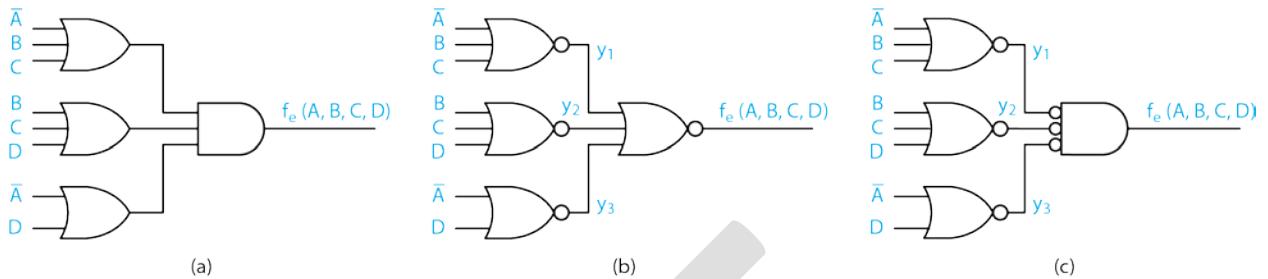


Figure 3.3 Implementations of $f_e(A, B, C, D) = (\bar{A} + B + C)(B + C + D)(\bar{A} + D)$. a OR-AND network. b NOR network. c NOR network (preferred form).

A POS expression can be realized with a NOR-NOR circuit by replacing each gate in the OR-AND circuit with a NOR gate as illustrated in Fig. 3.3b and 3.3c. This replacement can be justified using Boolean algebra as below.

The same Boolean algebra transformation that was employed previously can be used to express $f_e(A, B, C, D)$ in an appropriate form for direct NOR implementation, again using DeMorgan's theorem:

$$\begin{aligned} f_e(A, B, C, D) &= \overline{\overline{(\bar{A} + B + C)}(B + C + D)\overline{(\bar{A} + D)}} & [T3] \\ &= \overline{\overline{\overline{\bar{A} + B + C}} + \overline{B + C + D} + \overline{\overline{\bar{A} + D}}} & [T8(b)] \\ &= y_1 + y_2 + y_3 \end{aligned}$$

where $y_1 = \overline{\overline{\bar{A} + B + C}}$, $y_2 = \overline{\overline{B + C + D}}$, and $y_3 = \overline{\overline{\bar{A} + D}}$. The NOR realization of this function is shown in Fig. 3.3b.

Fig. 3.3c presents the same circuit, but with the output NOR gate shown in its DeMorgan-equivalent form. Writing the output logic expression,

$$\begin{aligned} f_e(A, B, C, D) &= \overline{\overline{y_1} \cdot \overline{\overline{y_2}} \cdot \overline{\overline{y_3}}} \\ &= \overline{\overline{(\bar{A} + B + C)} \cdot \overline{(B + C + D)} \cdot \overline{(\bar{A} + D)}} \\ &= (\bar{A} + B + C)(B + C + D)(\bar{A} + D) \end{aligned}$$

As was the case with two-level NAND circuits, the inversion bubbles effectively cancel out. This format more clearly illustrates that a function in product of sums form is being implemented.

Like NAND gates, NOR gates have two symbols as shown below that can be verified to be equivalent by applying DeMorgan's theorem.



Similar to the SOP case, POS expressions with single-literal sum terms require special consideration as illustrated below. The following function has the OR-AND and NOR-NOR realizations shown in Figure 3.4(a) and (b), respectively.

$$f(x, y, z) = x(y + z)$$

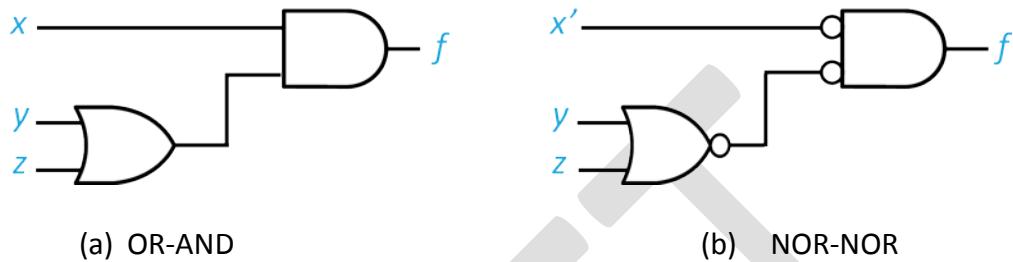


Figure 3.4 – Handling Single-Literal Sum Terms

The preceding method may be generalized to implement any product of sums function in NOR-NOR logic. Again it's not necessary to perform the algebra each time once you understand the process.

3.1.3 Two-level Circuits

Circuits that have a structure like those shown in Figs. 3.1 and 3.3 are referred to as *two-level circuits* since input signals must pass through two levels of gates before reaching the output. As illustrated in Fig. 3.5a, the first level is defined as the level containing the gate that produces the output. Gates that receive the circuit inputs are on the second level. When NOT gates are required on input lines, a three-level circuit is produced, as illustrated in Fig. 3.5b. A circuit has n levels when at least one input signal must pass through n gates before reaching the output.

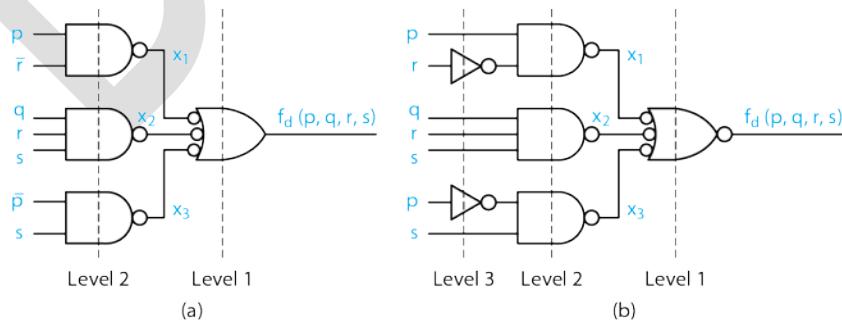


Figure 3.5 Two-level circuit structures. a Two-level circuit. b Two-level circuit with inverters.

Logic expressions in the SOP or POS form can be implemented directly in two-level circuits when the inputs are available in both complemented and uncomplemented form. A three-level network is required when only one form of the inputs is available. In the latter case, only NOT gates are needed on level 3.

At this point the reader has all the tools necessary to take a logic expression in minterm or maxterm list form and implement it in NAND or NOR logic, respectively. The implementation procedure for NAND logic is outlined next; the terms in parentheses are used if the implementation is to be in NOR logic. (a) Express the function in minterm (maxterm) list form. (b) Write out the minterms (maxterms) in algebraic form. (c) Simplify the function in sum of products (product of sums) form using Boolean algebra. (d) Use Theorem 8a (8b) and Theorem 3 to transform the expression into the NAND (NOR) formulation. (e) Draw the NAND (NOR) logic diagram. This procedure will now be illustrated using $f_\phi(X, Y, Z) = \sum m(0, 3, 4, 5, 7)$.

Example -- Implement $f_\phi(X, Y, Z) = \sum m(0, 3, 4, 5, 7)$ **in NAND logic.**

$$1. \quad f_\phi(X, Y, Z) = \sum m(0, 3, 4, 5, 7)$$

$$2. \quad f_\phi(X, Y, Z) = m_0 + m_3 + m_4 + m_5 + m_7 = \overline{XYZ} + \overline{XYZ} + X\overline{YZ} + X\overline{Y}\overline{Z} + XY\overline{Z}$$

$$3. \quad f_\phi(X, Y, Z) = \overline{YZ} + YZ + XZ \quad [T6(a)]$$

$$4a. \quad f_\phi(X, Y, Z) = \overline{\overline{YZ}} + \overline{YZ} + \overline{XZ} \quad [T3]$$

or

$$4b. \quad f_\phi(X, Y, Z) = \overline{\overline{YZ}} + YZ + XZ \\ = \overline{\overline{YZ}} \cdot \overline{YZ} \cdot \overline{XZ} \quad [T3] \quad [T8(a)]$$

The logic diagram of Fig. 3.7a is derived from the expression in step 4 and is said to be a *minimum two-level SOP realization* of the logic function. This example completely illustrates the design procedure.

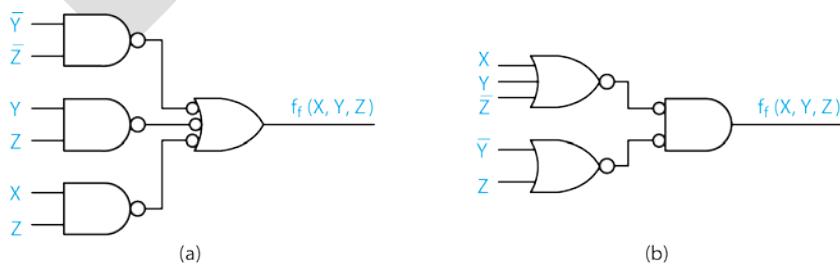


Figure 3.7 Canonical form of $f_\phi(X, Y, Z) = \sum m(0, 3, 4, 5, 7)$ to NAND and NOR circuits. a NAND realization. b NOR realization.

Example -- Implement $f_\phi(X, Y, Z) = \sum m(0, 3, 4, 5, 7)$ **in NOR logic.**

$$1. \quad f_\phi(X, Y, Z) = \prod M(1, 2, 6)$$

$$2. \quad f_\phi(X, Y, Z) = M_1 \cdot M_2 \cdot M_6 = (X + Y + \bar{Z})(X + \bar{Y} + Z)(\bar{X} + \bar{Y} + Z)$$

$$3. \quad f_\phi(X, Y, Z) = (X + Y + \bar{Z})(\bar{Y} + Z) \quad [T6(b)]$$

$$4a. \quad f_\phi(X, Y, Z) = \overline{(X + Y + \bar{Z})} \cdot \overline{(\bar{Y} + Z)} \quad [T3]$$

or

$$4b. \quad f_\phi(X, Y, Z) = \overline{(X + Y + \bar{Z})(\bar{Y} + Z)} \\ = \overline{(X + Y + \bar{Z})} + \overline{(\bar{Y} + Z)} \quad [T3] \quad [T8(b)]$$

The NOR network derived from step 4 is shown in Fig. 3.7b and is said to be a *minimum two-level POS realization* of the logic function. Each circuit shown in Fig. 3.7 implements the function $f_\phi(X, Y, Z)$. Note that steps four and five of the procedures can be eliminated once the process becomes routine.

Verilog and VHDL structural models of the NAND circuit are given below. The reader is referred to Appendices B and C for details on the structure, syntax, and features of Verilog and VHDL, respectively. It should be noted that Verilog provides operators for the basic gates that can be used in structural models. On the other hand, VHDL relies on libraries for defining these components. The *ieee.std_logic_1164* library is used in the following and other examples that appear later in the book. Structural models require the naming of internal nodes of the circuit being described. In the following Verilog model, a **wire** statement is used to specify internal nodes *a*, *b*, *c*. They are specified by a **signal** statement in the VHDL model.

```
//Verilog structural model
//NAND gate circuit -- Figure 3.7 (a)
module fig3_7nand (X,Y,Z,f);
    input X,Y,Z;
    output f;
    wire a,b,c;
    nand (a,~Y,~Z);
    nand (b,Y,Z);
    nand (c,X,Z);
    nand (f,a,b,c);
endmodule
```

```
--VHDL structural model
--NAND gate circuit -- Figure 3.7 (a)
library ieee;
use ieee.std_logic_1164.all;
entity fig3_7nand is
    port ( X,Y,Z: in std_logic;
           f: out std_logic);
end;
architecture eqns of fig3_7nand is
    signal a,b,c: std_logic;
begin
    a <= (not X) nand (not Y);
    b <= Y nand Z;
    c <= X nand Z;
    f <= (a nand b) nand c;
end;
```

3.1.4 Multi-Level Circuits and Factoring

Multi-level circuit realizations, meaning circuits with more than two-levels of gates, are advantageous if fewer gates are needed when compared to two-level circuits, and/or if gate fan-in constraints dictate that two-level realizations are not practical. The down-side is that multi-level circuits introduce more propagation delay than two-level circuits. Propagation delay is discussed later in the chapter. Factoring of Boolean expressions is employed to obtain the forms necessary for realizing multi-level circuit forms. When fan-in constraints apply, factoring is used to reduce the number of literals in large product or sum terms to values less than or equal to the available number of gate inputs.

Circuits with more than two levels are often needed when there are gate fan-in limits. For example, the function $f(a,b,c,d,e) = abcde$ can be realized with a single five-input AND gate as shown in Fig. 3.6a. However, if the designer is restricted to working with only two-input AND gates, then a three- or four-level circuit will be needed, as shown in Figs. 3.6b and c, respectively. The reader should verify that these circuits are equivalent. Multi-level circuits will be discussed in more detail in the next section.

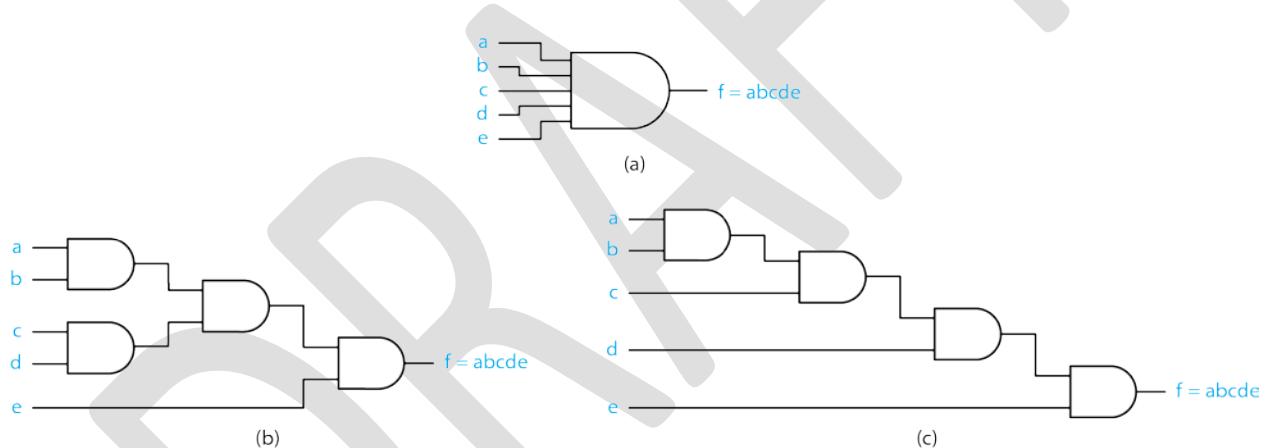


Figure 3.6. Multilevel circuit made necessary by fan-in restrictions. a A single five-input AND gate. b Three-level circuit of two-input gates. c Four-level circuit of two-input gates.

Multi-level design techniques are sometimes more difficult to use than the straight-forward techniques for two-level circuits since factoring of Boolean expressions is involved. Factoring, which normally involves the use of the distributive law (Postulate 5) of Boolean algebra, requires more skill to manipulate than for SOP and POS expressions. The technique is further complicated by the fact that redundancy may have to be added at an intermediate step in order to obtain a simpler realization through factoring. This method will be demonstrated via the following examples.

Example -- Suppose we are given the following four-variable function

$$f_\lambda(A, B, C, D) = A\bar{B} + A\bar{D} + A\bar{C}$$

The two-level realization of this function using NAND gates is shown in Fig. 3.8a.

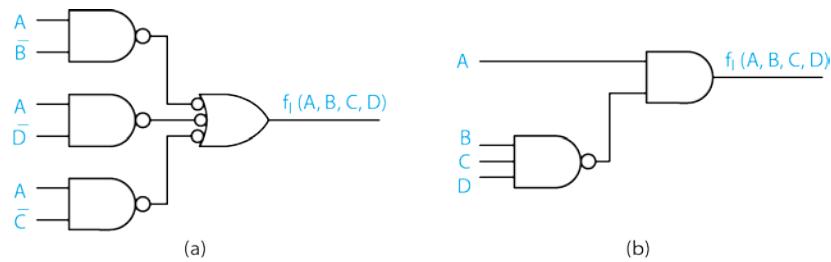


Figure 3.8 Realizations of $f_l(A, B, C, D)$. a Original form. b After factoring.

Note that the two-level realization of the function requires four gates and nine gate inputs. However, if we apply factoring to the function, we can obtain a higher-order realization, as follows:

$$\begin{aligned}f_l(A, B, C, D) &= A\bar{B} + A\bar{D} + A\bar{C} \\&= A(\bar{B} + \bar{D} + \bar{C}) \\&= A(\overline{BCD})\end{aligned}$$

This realization of $f_l(A, B, C, D)$, shown in Fig. 3.8b, requires only two gates and five gate inputs.

Example -- Realize the function $f(a, b, c, d) = \sum m(8,13)$ using only two-input AND and OR gates.

We begin by writing the canonical SOP form

$$\begin{aligned}f(a, b, c, d) &= \sum m(8,13) \\&= ab\bar{c}\bar{d} + ab\bar{c}d\end{aligned}\tag{3.1}$$

The two product terms in Eq. 3.1 cannot be reduced using Boolean algebra. Therefore, two four-input AND gates and one two-input OR gate would be required to realize a two-level AND-OR circuit.

Since only two-input gates are available, we can factor out the term ac' to reduce the size of the product terms to two literals each, as follows:

$$\begin{aligned}f(a, b, c, d) &= ab\bar{c}\bar{d} + ab\bar{c}d \\&= (a\bar{c})(bd + \bar{b}\bar{d})\end{aligned}\tag{3.2}$$

In Eq. 3.2, no product or sum term contains more than two literals. Thus, this logic expression can be realized entirely with two-input gates, as shown in Fig. 3.9. Note that the circuit contains three levels of logic gates, plus the input inverters, resulting in a four-level circuit.

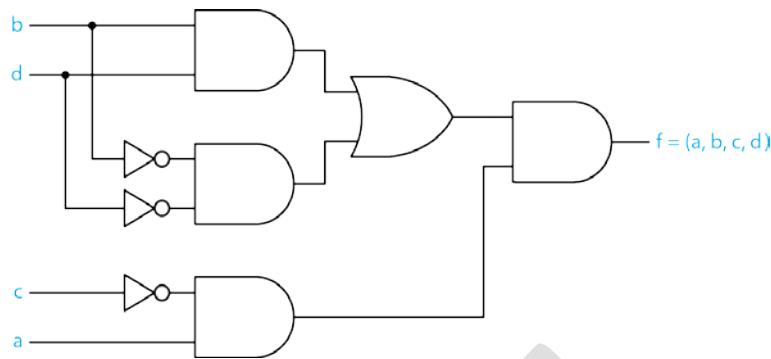
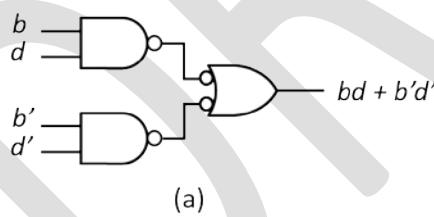


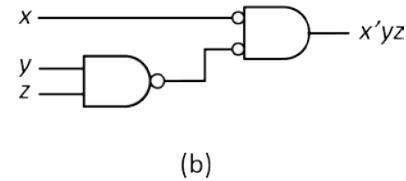
Figure 3.9. Factoring used to realize $f(a,b,c,d) = \sum m(8,13)$ with two-input AND and OR gates.

Example -- Realize the function $f(a,b,c,d) = \sum m(8,13)$ from the previous example using only two-input NAND and NOR gates.

The requirement to use two-input NAND or NOR gates sometimes further complicates the design and/or requires additional levels of logic as illustrated in this example. The SOP expression $bd + b'd'$ can be realized easily with three 2-input NAND gates as shown in Figure 3.10 (a) using the bubble method. However, the realization of the 3-input AND function is more complicated. Figure 3.10 (b) illustrates how this can be accomplished using the bubble procedure to develop a NAND-NOR circuit realization. Putting these two circuits together to realize f requires a NOT gate be inserted between the two as shown in Figure 3.11 (a) or realizing $(bd + b'd')' = bd' + b'd$ as shown in Figure 3.11 (b). The latter is generally preferred since it has fewer levels of logic.

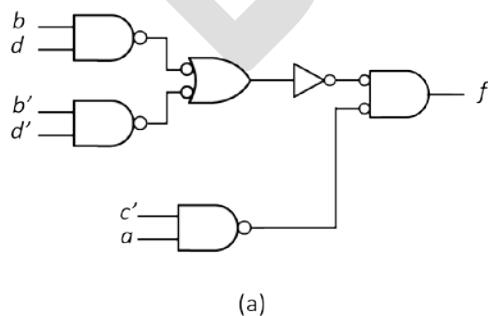


(a)

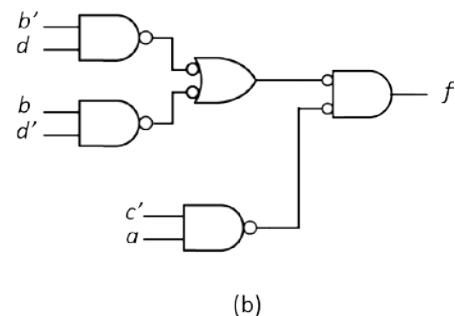


(b)

Figure 3.10. Factored components for realizing $f(a,b,c,d) = \sum m(8,13)$.



(a)



(b)

Figure 3.11. Realizations of $f(a,b,c,d) = \sum m(8,13)$ with 2-input NAND and NOR gates.

Example – Realize the function $F(a,b,c,d) = ad + bd + a'b'c'd'$ with 2-input NAND gates.

The above expression does not have a common literal or term that can be factored-out, so another approach to factoring is illustrated in this example. The given logic expression can be rewritten as shown below by applying the distributive property $x + yz = (x + y)(x + z)$.

$$ad + bd + a'b'c'd' = (ad + a'b')(ad + c'd') + bd$$

The rewritten expression can be realized with 2-input AND and OR gates as shown in Figure 3.12 (a). The bubble method can be applied to convert to an all NAND realization as illustrated in Fig. 3.12 (b).

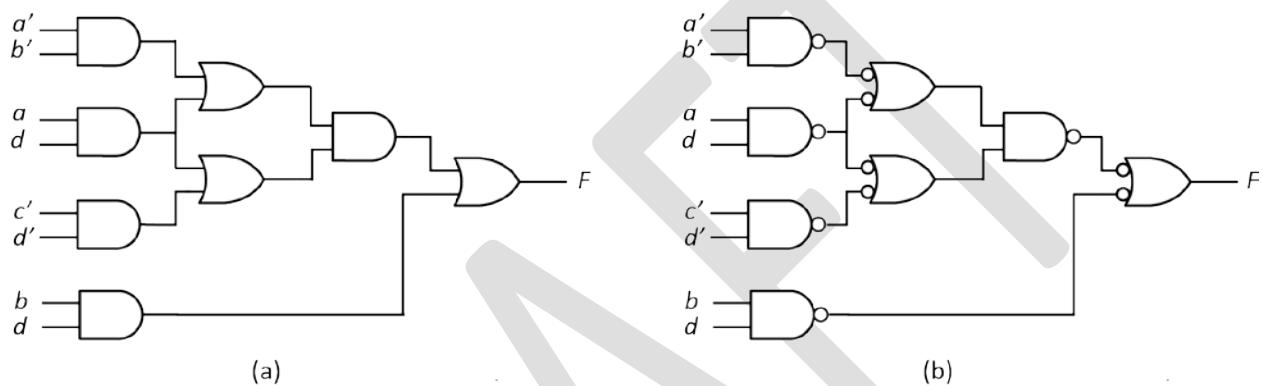


Figure 3.12. Realization of $F = ad + bd + a'b'c'd'$ with 2-input NAND gates.

3.1.5 EXCLUSIVE OR (XOR) Circuits

The EXCLUSIVE OR (XOR) function was first introduced in chapter 2 and shown to be logically equivalent to the sum function found in half adders and full adders. Its importance in realizing arithmetic circuits and error detection and correction circuits justifies further coverage here.

Figure 3.17 illustrates how two-input XOR gates can be used to realize three, four, and five variable XOR functions. The following example shows how the XOR function can be used to encode and decode simple parity codes.

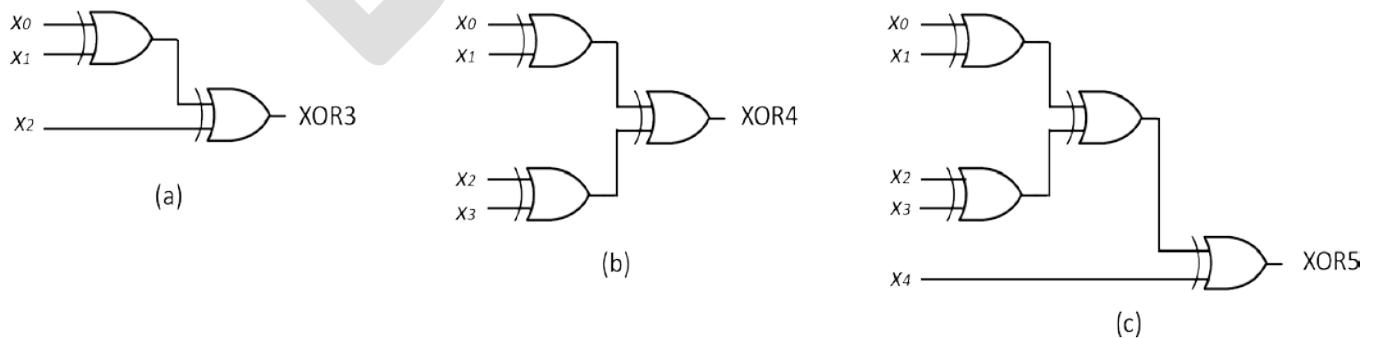
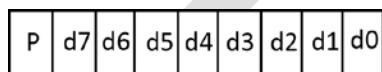


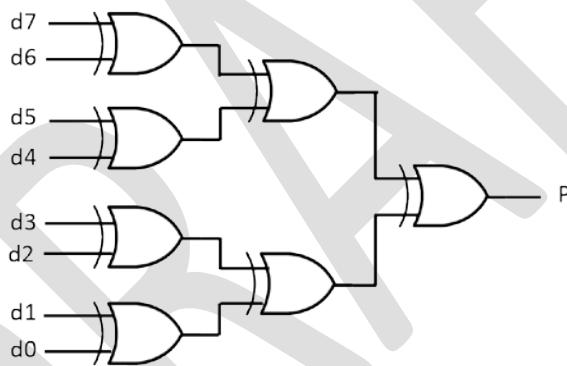
Figure 3.17. Realizations of three, four, and five variable XOR circuits.

An extra bit, called a parity bit, is sometimes appended to the data bits in memory devices and communication channels to provide single-bit error detection capability. Recall that parity codes were discussed in Chapter 1. The following example illustrates how EXCLUSIVE OR circuits can be used to generate parity bits and to check for an error.

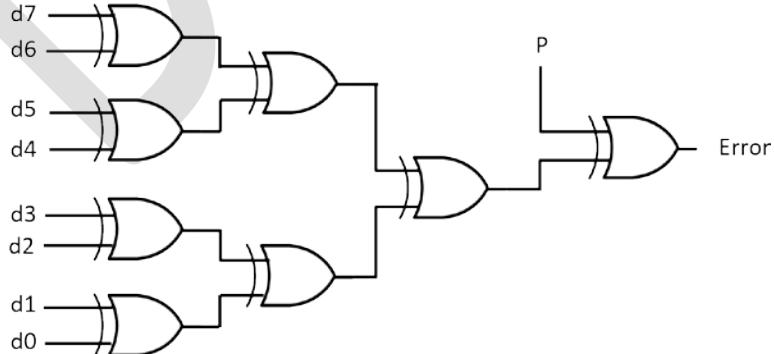
Example – Let the number of information, or data, bits be *eight* for a hypothetical storage device. Adding a parity bit will allow for single error detection during *read/write* operations. Figure 3.18 (a) shows the format of the encoded data. Also assume that an even-parity code will be used for encoding the data. This means that the weight of the encoded data will be an even number. So encoding consists of assigning a parity bit = 0 when the weight of a data word is even and 1 when the weight is odd. Error detection then becomes checking the encoded word for even parity. EXCLUSIVE OR circuits can be used for both as illustrated in Figures 3.18 (b) and (c).



(a) Parity Code Format



(b) Even Parity Bit Generation Circuit



(c) Even Parity Bit Check Circuit

Figure 3.18 Even-Parity Code Circuits

Such circuits are sometimes called *parity trees*. Verilog and VHDL structural models for the even-parity encoding function, Fig. 3.18 (b) are shown below.

```
//Verilog structural model
//Even-Parity Code Encoder
module ParityEncoderStructural (d,P);
    input [7:0] d;
    output P;
    wire e0,e1,e2,e3,e4,f0,f1;
    xor(e0,d[1],d[0]);
    xor(e1,d[3],d[2]);
    xor(e2,d[5],d[4]);
    xor(e3,d[7],d[6]);
    xor(f0,e1,e0);
    xor(f1,e3,e2);
    xor(P,f1,f0);
endmodule
```

```
-- VHDL structural model
--Even-Parity Code Encoder
library ieee;
use ieee.std_logic_1164.all;
entity ParityEncoder is
    port (d: in std_logic_vector(7 downto 0);
          P: out std_logic);
end;
architecture structure of ParityEncoder is
signal e0,e1,e2,e3,e4,f0,f1: std_logic;
begin
    e0 <= d(1) xor d(0);
    e1 <= d(3) xor d(2);
    e2 <= d(5) xor d(4);
    e3 <= d(7) xor d(6);
    f0 <= e1 xor e0;
    f1 <= e3 xor e2;
    P  <= f1 xor f0;
end;
```

Dataflow models of the even-parity encoder are shown below for Verilog and VHDL.

```
//Verilog dataflow model
//Even-Parity Code Encoder
module ParityEncoderBehavioral (d,P);
    input [7:0] d;
    output P;
    assign P = d[7]^d[6]^d[5]^d[4]^d[3]^d[2]^d[1]^d[0];
endmodule

--VHDL dataflow model
--Even-Parity Code Encoder
library ieee;
use ieee.std_logic_1164.all;
entity ParityEncoder is
    port (d: in std_logic_vector(7 downto 0);
          P: out std_logic);
end;
architecture behavioral of ParityEncoder is
begin
    P  <= ((d(7) xor d(6)) xor (d(5) xor d(4))) xor ((d(3) xor d(2)) xor (d(1) xor d(0)));
end;
```

Another useful application of a two-input EXCLUSIVE OR gate is to produce the complement, or not, of one input as a function of the other input. This is illustrated in Figure 3.19. If $x = 1$, then $g = f'$. Hence, the XOR2 gate functions as a controllable NOT gate with x as the control variable. This functionality is useful for implementing a two's complement adder/subtractor circuit as will be illustrated late in the chapter.

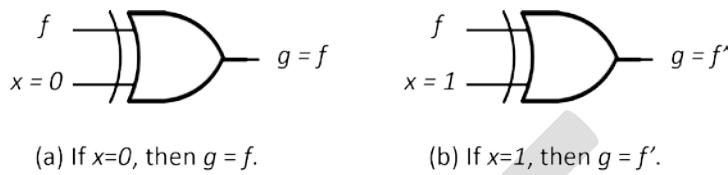


Figure 3.19 Controllable NOT Gate

This section has covered the design of logic circuits from basic logic gates using schematic diagrams and hardware description languages (HDLs) to describe the designs. The use of HDLs for functional/dataflow descriptions was also introduced. In design, the job is to derive or determine a realization of a function, or set of functions, given the definition, or requirement, of the function(s).

The following section will focus on analysis of logic circuits. In analysis, the job is to determine which function or functions are realized given the realization or design. In essence, analysis is the reverse of design, and vice versa.

3.2 Analysis of Combinational Circuits

Combinational circuit analysis is the process of determining the logic function(s) realized by a circuit given its hardware realization in gates, programmable logic devices (PLDs), or other realization forms. The analysis process is sometimes called *reverse engineering*. The function can be derived using Boolean expressions, truth tables, timing diagrams, or other tools such as logic simulators. Analysis is used to verify that the behavior of a circuit matches its specifications, or to assist in converting the circuit to a different form, either to reduce the number of gates or to realize it with different elements. In a broader sense, behavior refers to both output functionality and timing relationships between the hardware input signals and output signals. Timing diagrams are particularly useful for this purpose.

3.2.1 Boolean Algebra

Combinational circuits may be realized by interconnecting logic gates or programming PLDs as presented in the previous chapter. Any given combinational circuit may be represented by a Boolean expression or function, and, thus, all the power of Boolean algebra

may be applied to manipulate the expression in to any form we desire.

A series of three examples will now be used to illustrate the use of Boolean algebra for analyzing logic circuits.

Example -- Find a simplified logic expression and circuit for the circuit given in Fig. 3.20a.

We proceed by writing a Boolean expression for the output of each gate.

$$P_1 = \overline{ab}$$

$$P_2 = \overline{\bar{a} + c}$$

$$P_3 = b \oplus \bar{c}$$

$$P_4 = P_1 \cdot P_2 = \overline{ab} (\overline{\bar{a} + c})$$

The output is

$$\begin{aligned} f(a, b, c) &= \overline{P_3 + P_4} \\ &= \overline{(b \oplus \bar{c}) + \overline{ab} (\overline{\bar{a} + c})} \end{aligned}$$

To analyze this function, we may convert it to a simpler form using Boolean algebra:

$$\bar{f}(a, b, c) = (b \oplus \bar{c}) + \overline{ab} \overline{\bar{a} + c} \quad [\text{Eq. 2.2}]$$

$$= bc + \bar{b}\bar{c} + \overline{ab} \overline{\bar{a} + c} \quad [\text{T8}]$$

$$= bc + \bar{b}\bar{c} + (\bar{a} + \bar{b})ac \quad [\text{T5(b)}]$$

$$= bc + \bar{b}\bar{c} + ab\bar{c} \quad [\text{T4(a)}]$$

$$\bar{f}(a, b, c) = bc + \bar{b}\bar{c} \quad [\text{Eq. 2.3}]$$

Therefore, from Eq. 2.3,

$$f(a, b, c) = \overline{bc + \bar{b}\bar{c}} = b \oplus c$$

This function has been reduced to a single exclusive-OR gate, which is shown in Fig. 3.20b.

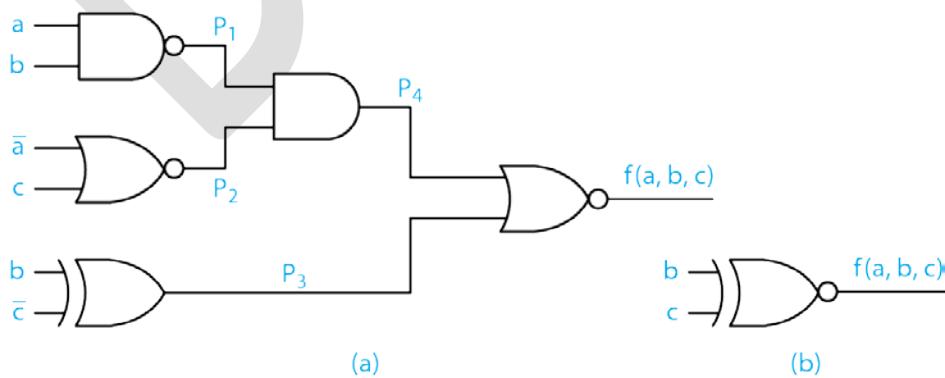


Figure 3.20 Equivalent switching networks.

Both logic circuits shown in Fig. 3.20 have the same truth table and are therefore equivalent. It is obvious that the one in Figure 3.20b is more desirable since it is less complex.

Example – Find a simplified switching expression and circuit for the circuit of Fig. 3.21.a

The logic expression for each gate output is shown on the logic diagram of Fig. 3.21. From these, we derive the output expression as

$$\begin{aligned}
 f(a,b,c) &= \overline{(a \oplus b)(b \oplus c)} \cdot (\overline{\bar{a} + \bar{b}} + \overline{a + c}) \\
 &= \overline{(a \oplus b)(b \oplus c)} + \overline{\bar{a} + \bar{b}} + \overline{a + c} && [\text{T8(b)}] \\
 &= (a \oplus b)(b \oplus c) + (\bar{a} + \bar{b})(a + c) && [\text{T8(a)}] \\
 &= (ab + \bar{a}\bar{b})(b\bar{c} + \bar{b}c) + (\bar{a} + \bar{b})(a + c) && [\text{Eq. 2.2}] \\
 &= ab\bar{b}\bar{c} + ab\bar{b}c + \bar{a}bb\bar{c} + \bar{a}b\bar{b}c + \bar{a}a + \bar{a}c + a\bar{b} + \bar{b}c && [\text{P5(b)}] \\
 &= a\bar{b}c + \bar{a}b\bar{c} + \bar{a}c + a\bar{b} + \bar{b}c && [\text{P6(b), T4(a)}] \\
 &= \bar{a}b\bar{c} + \bar{a}c + a\bar{b} + \bar{b}c && [\text{T4(a)}] \\
 &= \bar{a}b\bar{c} + \bar{a}c + a\bar{b} && [\text{T7(a)}] \\
 &= \bar{a}b + \bar{a}c + a\bar{b} && [\text{P5(b), T5(a)}] \\
 &= \bar{a}c + a \oplus b && [\text{Eq. 2.2}]
 \end{aligned}$$

Notice that this is the form of the logic circuit in Fig. 3.21b.

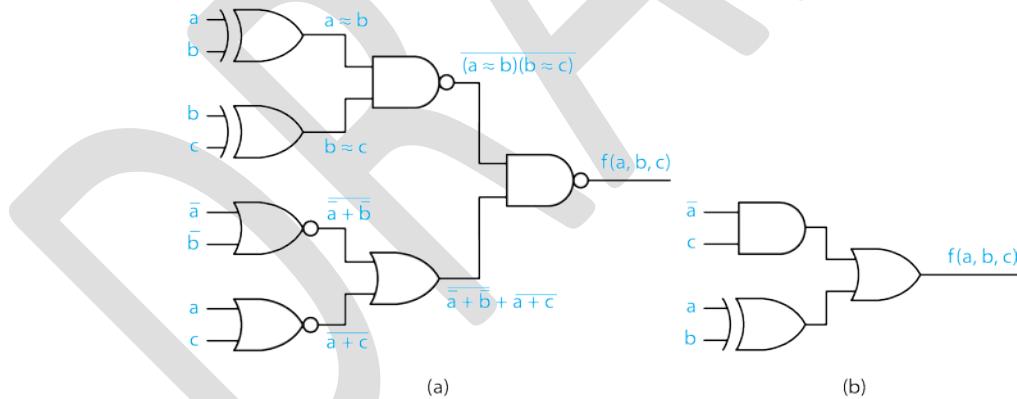
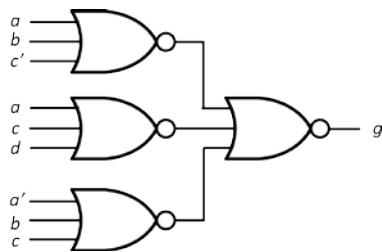


Figure 3.21 Equivalent switching networks.

Example – Verify that the circuit in Fig. 3.22 realizes the following logic function.

$$g(a,b,c,d) = ab + ac + bc + a'c'd$$

**Figure 3.22** Design Verification Example.

Note that the circuit is a two-level NOR realization, so the function realized can be written in POS form and, using the postulates and theorems of Boolean algebra, converted to the given SOP expression. Hence, the circuit correctly realizes function.

$$\begin{aligned}
 (a + b + c')(a + c + d)(a' + b + c) &= (a + (b + c'))(c + d))(a' + b + c) \\
 &= (a + bc + bd + c'c + c'd)(a' + b + c) \\
 &= (a + bc + bd + c'd)(a' + b + c) \\
 &= aa' + ab + ac + a'bc + bc + bc + a'bd + bd + bdc \\
 &\quad + a'c'd + bc'd + cc'd \\
 &= ab + ac + bc + bd + a'c'd \\
 &= ab + ac + bc + a'c'd
 \end{aligned}$$

A K-map can also be used to solve this problem rather than manipulating Boolean expressions. First, map the 0's of the function from the POS expression derived from the circuit. Then, show how the 1's can be grouped to derive the desired SOP expression.

		ab	00	01	11	10
cd		00	0	0	1	0
		01	1	1	1	0
		11	0	1	1	1
		10	0	1	1	1

$$(a + b + c')(a + c + d)(a' + b + c) = ab + ac + bc + a'c'd$$

In this section we have shown how to take a given logic circuit, analyze it by deriving its Boolean expression. Then we can simplify the expression using Boolean algebra, and obtain an equivalent, but simpler circuit. Analysis can also be used to verify that a circuit realizes the correct function or to determine which function a circuit realizes.

3.2.2 Truth Tables

In Chapter 2, the truth table for a logic function was derived from a Boolean expression by evaluating the expression one part at a time. An analogous approach may be taken with logic circuit diagrams, deriving the truth table one gate at a time.

The truth table for the circuit shown in Fig. 3.21 (b) may be found using such an approach as demonstrated below.

abc	$\bar{a}c$	$a \oplus b$	$f(a,b,c) = (\bar{a}c) + (a \oplus b)$
0 0 0	0	0	0
0 0 1	1	0	1
0 1 0	0	1	1
0 1 1	1	1	1
1 0 0	0	1	1
1 0 1	0	1	1
1 1 0	0	0	0
1 1 1	0	0	0

Column $\bar{a}c$ which corresponds to the AND gate output is 1 whenever $\bar{a}=1$ and $c=1$ or whenever $a=0$ and $c=1$. Column $a \oplus b$ which corresponds to the XOR gate is 1 whenever $a \neq b$. These two columns are ORed to create $f(a,b,c)$; therefore, $f(a,b,c)$ is 0 whenever both $\bar{a}c$ and $a \oplus b$ are 0.

Truth tables may also be derived from HDL models as shown in the following example.

Example – Derive the truth table of the logic circuit described by the following models.

//Verilog analysis example

```
module VerilogAnalysisExample (a,b,c,d,f,g);
    input a,b,c,d;
    output f,g;
    assign f = a&d | b&~c | ~b&c;
    assign g = a&b | c&d;
endmodule
```

-- VHDL analysis example

```
library ieee;
use ieee.std_logic_1164.all;
entity VHDLAnalysisExample is
    port (a,b,c,d: in std_logic;
          f,g: out std_logic);
end;
architecture equations of
VHDLAnalysisExample is
begin
    f <= (a and d) or (b and not c) or (not b and c);
    g <= (a and b) or (c and d);
end;
```

In Verilog, the declaration statements identify a, b, c, d as the input variables and f and g as the output functions. Assign statements are used to define the functions in SOP-like forms, so the truth table can be derived in a manner similar to the previous example as shown below. A similar approach is used with VHDL.

$abcd$	ad	bc'	$b'c$	ab	cd	$f(a,b,c,d)$	$g(a,b,c,d)$
0000	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0
0010	0	0	1	0	0	1	0
0011	0	0	1	0	1	1	1
0100	0	1	0	0	0	1	0
0101	0	1	0	0	0	1	0
0110	0	0	0	0	0	0	0
0111	0	0	0	0	1	0	1
1000	0	0	0	0	0	0	0
1001	1	0	0	0	0	1	0
1010	0	0	1	0	0	1	0
1011	1	0	1	0	1	1	1
1100	0	1	0	1	0	1	1
1101	1	1	0	1	0	1	1
1110	0	0	0	1	0	0	1
1111	1	0	0	1	1	1	1

3.2.3 Timing Diagrams

Thus far, we have analyzed logic circuits by deriving Boolean expressions and/or truth tables for the logic functions realized by the circuits. Another method of analysis is to apply a sequence of values to the inputs of a circuit over a period of time, either manually, experimentally, or with a logic simulation program, and observe the relationship between the inputs and the corresponding sequence of outputs in the form of a timing diagram. From this timing diagram, we can derive the logic function realized by the circuit and study the effects of gate propagation delays on circuit behavior.

Timing Diagrams

A *timing diagram* is a graphical representation of input and output signal relationships in a logic circuit, as might be seen on the display of an oscilloscope or logic analyzer or in a logic simulation program. Often, intermediate signals are also included. In addition, timing diagrams may show propagation delays introduced by the logic devices as signals propagate through the circuit. A properly constructed timing diagram can depict all the information contained in the truth table, as shown in the following example.

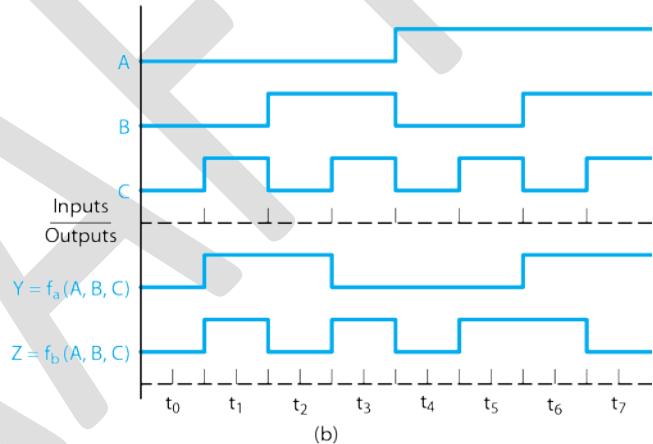
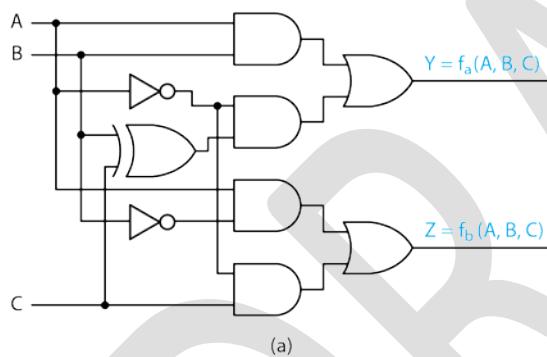
The circuit of Fig. 3.23a is stimulated with a sequence of inputs, producing the timing diagram of Fig. 3.23b. In this example, a 1 is represented by a high signal and a 0 by a low signal. Let us determine the truth table and minterm lists for the two functions $f_\alpha(A,B,C)$ and $f_\beta(A,B,C)$ realized by this circuit.

The input and output signals are drawn on the circuit diagram. The input patterns have been selected so that each possible combination of inputs A , B , and C occurs for one unit of time.

Examining the timing diagram at times t_0, t_1, \dots, t_7 , we determine the values of the inputs and outputs at each time and write them in truth table form as in Fig. 3.23c. From the truth table we can write the minterm list and then derive a simplified logic expression for each function, as follows:

$$\begin{aligned}f_a(A, B, C) &= \sum m(1, 2, 6, 7) \\&= \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC \\&= \overline{A}\overline{B}C + B\overline{C} + AB\end{aligned}$$

$$\begin{aligned}f_b(A, B, C) &= \sum m(1, 3, 5, 6) \\&= \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + ABC \\&= \overline{A}C + A\overline{B}C + ABC\end{aligned}$$



Time	Inputs A B C	Outputs	
		$f_a(A, B, C)$	$f_b(A, B, C)$
t_0	0 0 0	0	0
t_1	0 0 1	1	1
t_2	0 1 0	1	0
t_3	0 1 1	0	1
t_4	1 0 0	0	0
t_5	1 0 1	0	1
t_6	1 1 0	1	1
t_7	1 1 1	1	0

(c)

Figure 3.23 Derivation of a truth table from a timing diagram. a Logic circuit realizing two functions. b Timing diagram. c Truth table.

In the previous example, all gate outputs were shown as changing instantaneously following an input change. In reality, there is always a delay between the time of an input change and the corresponding output change in a circuit. In the next section we consider the effects of gate propagation delays on circuit operation.

Propagation Delay and Other Physical Constraints

In addition to behavior or functionality, a designer must often be concerned with a number of physical characteristics of digital logic circuits, including the following.

- Propagation delays
- Gate fan-in and fan-out restrictions
- Power consumption
- Size and weight

These characteristics are physical properties of the lower-level circuits used to construct the logic gates, and the number and configuration of gates in a given circuit. Lower-level devices are beyond the scope of this book and will not be discussed here. However, propagation delays and fan-in/fan-out restrictions can have a significant impact on logic design, and must often be considered during any digital logic circuit analysis and/or design.

A physical logic gate requires a nonzero amount of time to react to input changes and produce changes in its output state. This delay between the time of an input change and the corresponding output change is called *propagation delay*. Thus, if a logic circuit realizes a function $z = f(x_1, \dots, x_n)$, the propagation delay is the time that it takes changes to "propagate" or travel from some input x_i through the circuit to the output z . Propagation delays are functions of the circuit complexity, the electronic technology used, and such factors as gate fan-out (the number of other gate inputs driven by a single gate output), temperature, and chip voltage levels.

Following an input change, the output of electronic logic circuits may take different amounts of time to switch from low to high than from high to low. Hence, two propagation delay parameters are typically specified for a given logic gate.

$$\begin{aligned}t_{PLH} &= \text{propagation delay time, low- to high-level output} \\t_{PHL} &= \text{propagation delay time, high- to low-level output}\end{aligned}$$

with t_{PLH} and t_{PHL} measured from the time of the input change to the time of the corresponding output change.

When precise timing information is not needed, a single propagation delay parameter, denoted by t_{PD} , is used to approximate both t_{PLH} and t_{PHL} . Usually, t_{PD} is computed as the average of t_{PLH} and t_{PHL} .

$$t_{PD} = \frac{t_{PLH} + t_{PHL}}{2}$$

For the AND gate illustrated in Fig. 3.24a, Figs. 3.24b-d illustrate the response of the gate output to a sequence of changes in its input values. In Fig. 3.24b, the ideal case is shown, in which the outputs change instantaneously, that is, the propagation delay is 0. In Fig. 3.24c, all output changes are shown as being delayed by an average propagation delay, t_{PD} . Figure 3.24d presents a more precise picture of the timing, with separate parameters for t_{PLH} and t_{PHL} .

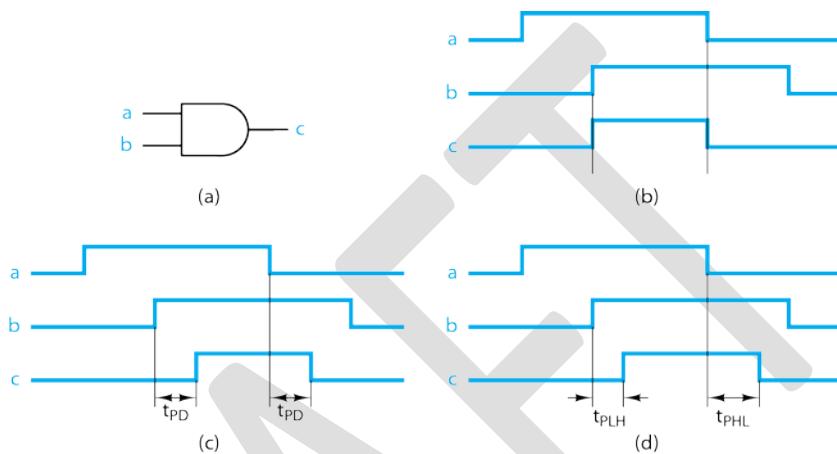


Figure 3.24 Propagation delay through a logic gate. (a) Two-input AND gate. (b) Ideal (zero) delay. (c) $t_{PD} = t_{PLH} = t_{PHL}$. (d) $t_{PLH} < t_{PHL}$.

Typical propagation delay and other physical characteristics are summarized in the table below for various electronic technologies. Those interested in a more in depth study of propagation delay and digital electronic technologies are referred to Appendix A. Values used in the following discussions will be paramaterized or generic.

Technology	Propagation Delay (nanoseconds)	Power Consumption (milliwatts)	Voltage Levels		Power Supply
			V_H	V_L	
TTL – 74LS00	10	2	2 to 5	0 to 0.8	5 volts
cMOS – 74HC00	11	0.17	3.5 to 5	0 to 1.5	5 volts
cMOS – 1.8 volt	0.1	0.1	1.2 to 1.8	0 to 0.63	1.8 volts

Example -- A sequence of inputs is applied to the circuit of Fig. 3.25a, producing the timing diagram of Fig. 3.25b. Each gate has propagation delay of one time unit t_{PD} . Find the truth table and minimum logic expression for this circuit. Also, find the maximum propagation delay through the circuit.

From the timing diagram, the truth table of Fig. 3.25c is derived by examining the outputs of each gate following each of the input changes. Since signals take different amounts of time to propagate to each gate output, we must wait until all signal propagation is complete before determining the output corresponding to the current input. Note that no signal will

propagate through more than three gates, therefore no more than three time units will elapse between an input change and a stable output. So the maximum propagation delay is $3t_{PD}$.

For example, at time t_1 input C changes from 0 to 1, which causes inverter output D and AND gate output G to both change at time t_1+1 . The change at G causes OR gate output Y to change from 0 to 1 at time t_1+2 . Thus the input change required two time units to propagate from circuit input C to circuit output Y . Therefore, we should wait until after t_1+2 to determine the final value of Y . Note that the input change at time t_2 also required two time units to propagate to the output, whereas the changes at times t_4 and t_7 require three time units.

From the truth table we can write the list of minterms and derive a minimum switching expression as follows:

$$\begin{aligned} f(A, B, C) &= \sum m(1, 4, 5, 6) \\ &= \overline{A}\overline{B}C + A\overline{B}\overline{C} + \overline{A}\overline{B}C + A\overline{B}C \\ &= \overline{AC} + \overline{BC} \end{aligned}$$

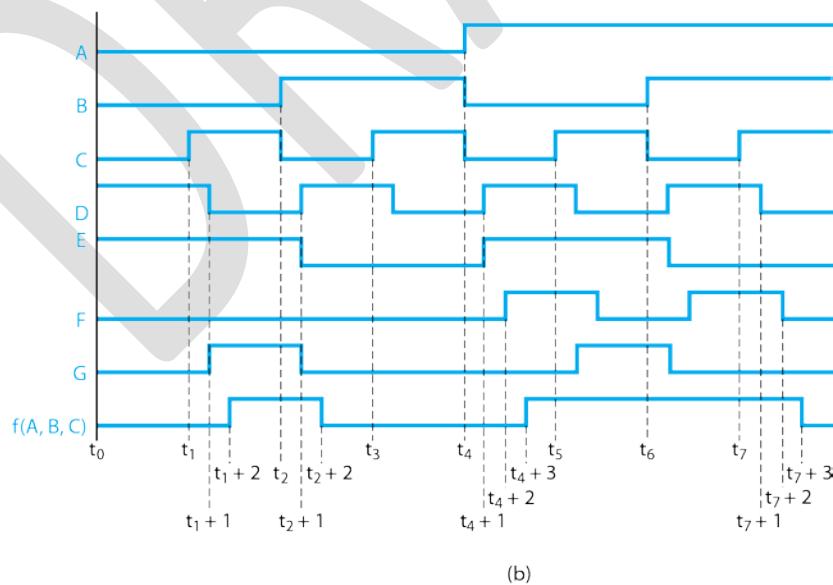
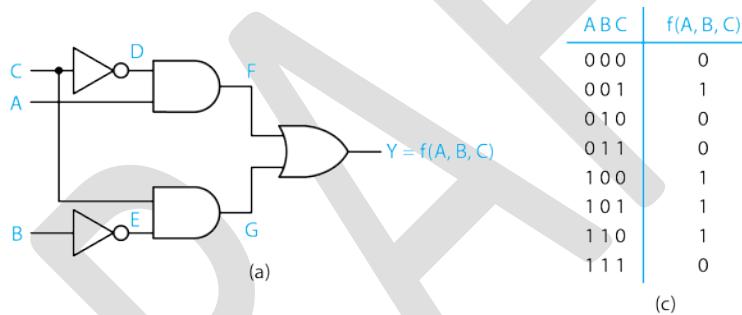


Figure 3.25 Derivation of a truth table from a timing diagram. a Circuit diagram. b Timing diagram. c Truth table.

Now let's look at propagation delay for some of the circuits we've studied earlier in the chapter. We will assume the propagation of all types of gates is t_g . When calculating the propagation delay of a circuit, we look for the maximum length path from an input to an output and add the delays along the path. This approach will provide the worst-case scenario for propagation delay through a circuit given the assumption that all gates have the same delay. The speed of a circuit is inversely related to its propagation delay.

Two-level circuits such as those shown in Figures 3.1, 3.3, and 3.5 have a maximum path of two, so all two-level circuits have a propagation delay of $2t_g$. If input NOT gates are used, the delay is increased to $3t_g$.

Multi-level circuits like those in Figure 3.6 have longer delays. The circuit in Figure 3.6 (b) has three levels and a propagation delay of $3t_g$ while the circuit in Figure 3.6 (c) has four levels and a delay of $4t_g$. The circuits in Figures 3.11 (a) and (b) have delays of $4t_g$ and $3t_g$, respectively. If input NOT gates are used, the delays would be $5t_g$ and $4t_g$.

Propagation delay can be specified in Verilog models using the #<value> notation in statements as illustrated below for the circuit Fig. 25 (a). VHDL uses the *after* operator to specify delay. Note that propagation delay is parameterized and set equal to 10 time-units. It is up to the designer to interpret a time-unit in terms of real time. A time unit of one nanosecond is assumed in the following example. Also note that propagation delays in Verilog and VHDL models are useful for simulation purposes but are ignored for synthesis.

```
//Verilog model
//Circuit with propagation delays
module PropagationDelayExample
(A,B,C,Y);
    input A,B,C;
    output Y;
    wire D,E,F,G;
    parameter tpd = 10;
    not #tpd (D,C);
    not #tpd (E,B);
    and #tpd (F,A,D);
    and #tpd (G,C,E);
    or #tpd (Y,F,G);
endmodule
```

```
--VHDL model
-- Circuit with propagation delays
library ieee; use ieee.std_logic_1164.all;
entity PropagationDelayExample is
    generic (tpd: time := 10 ns);
    port (A,B,C:  in std_logic;
          Y: out std_logic);
end;
architecture delay of
PropagationDelayExample is
    signal D,E,F,G: std_logic;
begin
    D <= not C after tpd;
    E <= not B after tpd;
    F <= A and D after tpd;
    G <= C and E after tpd;
    Y <= F or G after tpd;
end;
```

3.2.4 Positive and Negative Logic

The high-levels and low-levels used in the above timing diagrams represent high (V_H) and low (V_L) voltages in the electronic technologies used to implement the logic devices. Typical values for various technologies were shown in the previous table.

Positive logic has been assumed in the previous discussions. Meaning that the high-level represents *logic 1* and the low-level *logic 0*. It is important to note that this is merely a convention, albeit a common one. *Negative logic* is the convention of letting the high-level represent *logic 0* and the low-level *logic 1*. A *mixed-logic* system uses positive logic for some signals and negative logic for others. Positive logic is commonly used in practice and will be throughout this text unless otherwise noted. However, the terminology described in the following paragraph is useful to know for a better understanding of the various conventions.

A signal that is set to logic 1 is said to be *asserted*, or *active*, or *true*. An *active-high* signal is asserted when it is high (positive logic), whereas an *active-low* signal is asserted when it is low (negative logic). If not asserted, that is, if set to logic 0, a signal is said to be *deasserted*, or *negated*, or *false*. The term *polarity* is used to refer to the active-high or active-low nature of a logic signal.

When representing signals by logic variables, active-low signal names are written in complemented form (for example, \bar{a} , a' , a^*); active-high signal names are written in uncomplemented form (a). Each signal name should be selected so that the name suggests the purpose of that signal. For example, the signal name *RUN* suggests a signal that is asserted (high) to make a piece of equipment start running. If the signal is active low, the signal name *RUN* should be used to indicate that the equipment will be made to run when the signal is asserted low. In this text we will primarily use active-high signals. However, since many commercial circuit modules have active-low inputs and/or outputs, a number of examples will also be presented in which active-low signals are used.

The logic function realized by a given logic circuit depends upon which convention is being used as illustrated below. Let the diagram in Fig. 3.26 (a) represent a logic circuit with input variables, X and Y , and output function Z . The function realized by the circuit is not known but can be experimentally derived by applying all four combinations of V_H and V_L which are known to be 4.9 and 0 volts, respectively, and measuring the corresponding output. The results are tabulated in Fig. 3.26 (b). This corresponds to the truth table in Fig. 3.26 (c) for positive logic and the table in Fig. 3.26 (d) for negative logic. So $Z = X + Y$ for positive logic and $Z = X \cdot Y$ for negative logic.

(a) Input/Output Diagram: A block diagram showing a 'Black Box' with two inputs, X and Y, and one output, Z.

(b) Voltage Level Table:

Voltage Level Table		
V_X	V_Y	V_Z
0	0	0
0	4.9	4.9
4.9	0	4.9
4.9	4.9	4.9

(c) Positive Logic Truth Table:

Positive Logic Truth Table		
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

(d) Negative Logic Truth Table:

Negative Logic Truth Table		
X	Y	Z
1	1	1
1	0	0
0	1	0
0	0	0

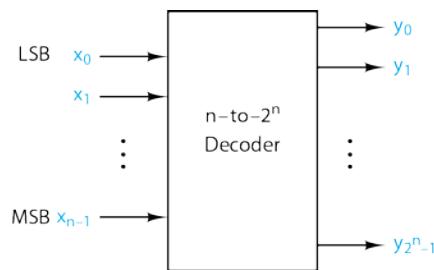
Figure 3.26 Positive and Negative Logic Example. (a) Input/Output Diagram.
(b) Voltage Table. (c) Positive Logic Truth Table. (d) Negative Logic Truth Table.

3.3 Design Using Higher-Level Devices

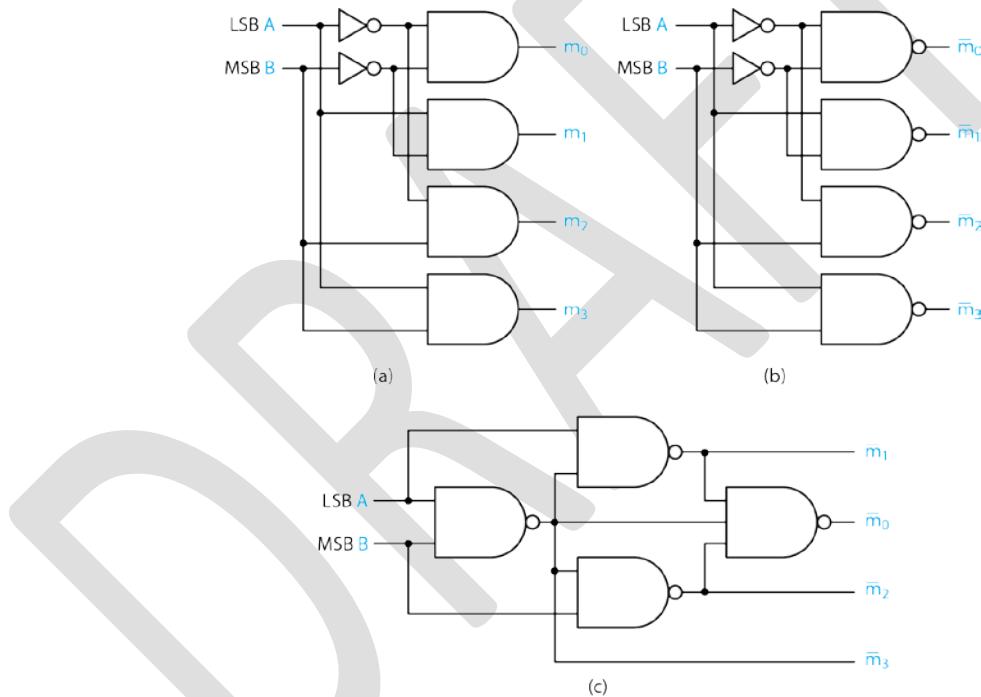
Many combinational logic operations such as decoding, encoding, multiplexing, and demultiplexing are used so frequently in logic circuits that they are defined as higher-level modules without the need to know how they are realized at the gate level. The same can be said for various arithmetic operations. The most common of these modules will be presented in this section at both module and gate levels. Their use as module-level circuit design components will also be demonstrated.

3.3.1 Decoders

An n -to- 2^n *decoder* is a multiple-output combinational logic circuit with n input lines and 2^n output signals, as illustrated in Fig. 3.27. For each possible input condition, one and only one output signal will be at logic 1. Therefore, we may consider the n -to- 2^n decoder as simply a minterm generator, with each output corresponding to exactly one minterm. Decoders are important tools in the logic designer's repertoire. They are used for such things as addressing memory in order to select a particular word from the many that are available, code conversion (for example, binary to decimal), and routing of data.

**Figure 3.27.** n -to- 2^n decoder module.

Before looking at applications of n -to- 2^n decoders, let us examine some basic circuit structures used to implement these modules.

**Figure 3.28.** Two-bit decoder two-level circuit structures. a Active-high outputs. b Active-low outputs. c Alternate (three-level) structure.

The logic circuit of a *2-to-4 decoder* is shown in Fig. 3.28a. As can be seen from the figure, an input combination or *vector* of $BA = 00$ activates the m_0 output line, $BA = 01$ selects the m_1 output line, and so on meaning the corresponding output is logic 1 and all others are logic 0 as described by the following truth table and logic equations 3.3.

BA	m_0	m_1	m_2	m_3
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

$$\begin{aligned}m_0 &= \overline{BA} \\ m_1 &= \overline{B}\overline{A} \\ m_2 &= B\overline{A} \\ m_3 &= BA\end{aligned}\tag{3.3}$$

Figure 3.28b shows an alternative implementation of the 2-to-4 decoder, using NAND and NOT gates. Figure 3.28c presents another configuration, using only NAND gates with no inverters. In the NAND gate design, an output of 0 indicates the presence of the corresponding minterm. In this case, the outputs are said to be *active low* since an output value of 0 (the "active" level) signifies the occurrence of a "significant" input (the occurrence of a particular minterm). The output value is 1 (the "inactive" level) at all other times. An *active high* signal uses the value 1 to indicate a significant event and is 0 otherwise, such as in the decoder circuit of Fig. 3.28a. The output signals of the decoder in Fig. 3.28b may also be considered to be in complemented form.

Note that in the AND and NAND gate realizations of the n -to- 2^n decoder shown in Figs. 3.28a and b there are two levels of logic and that one n -input AND or NAND gate is required for each of the 2^n output lines. However, a problem is soon encountered in this configuration as n becomes large because the number of inputs to the gates (the *fan-in*) exceeds practical limits (eight). This problem can be alleviated through the use of a multi-level *tree-structured decoder*, such as that shown in Fig. 3.29b, which can be compared to the single-level decoder of Fig. 3.29a. This type of decoder employs multi-level logic with only two-input AND gates, independent of the number of input lines. A final structure, called the *dual tree*, is illustrated in Fig. 3.29c for a 4-to-16 decoder. In the dual-tree structure the n input lines are divided into j and k groups ($j+k=n$), and then two smaller decoders j -to- 2^j and k -to- 2^k are used to generate 2^j and 2^k internal signals. Then two-input AND gates are used to combine these signals to form the 2^n output lines for the total decoder circuit.

Decoders modules often include one or more *enable* inputs in addition to the data inputs, as shown in Fig. 3.30. Enable inputs are used to either inhibit (disable) the designated function or allow (enable) it to be performed. The decoding function of a decoder is inhibited by forcing all its outputs to the inactive state (logic 0 for active-high devices and 1 for active-low devices). In other words, enable inputs are used to either activate or deactivate all of the outputs of the decoder. For example, output y_0 of the 2-to-4 decoder in Fig. 3.30a is given by $y_0 = \bar{x}_1 \bar{x}_0 E = m_0 E$. In general,

$$y_k = m_k E \tag{3.4}$$

When $E = 0$, all outputs are forced to 0, whereas for $E = 1$, each output y_k is equal to m_k .

A common use of the enable input is cascading multiple decoders to realize a larger decoder. This is illustrated in Fig. 3.31, with the 2-to-4 decoder of Fig.3.28b being used to realize 3-to-8 and 4-to-16 decoders.

In Fig. 3.31a, input $I_2 = 0$ enables the top decoder, which is thus enabled for input codes $I_2 I_1 I_0$ equal to 000, 001, 010, and 011 (codes 0 through 3). The bottom decoder is enabled by $I_2 = 1$, enabling that module for input codes 4 through 7. Figure 3.31b illustrates a hierarchical decoding of a 4-bit number, with the first-level decoder enabling exactly one of the four second-level decoders.

Decoders can be described using structural Verilog and VHDL models as shown below. These models correspond directly to the circuit realization shown in Fig. 3.28 (b).

```
//Verilog structural model
//Two to Four Active-Low Decoder
module Two2FourLow (A,B,m0,m1,m2,m3);
  input A, B;
  output m0,m1,m2,m3;
  nand (m0,~B,~A);
  nand (m1,~B,A);
  nand (m2,B,~A);
  nand (m3,A,B);
endmodule
```

```
--VHDL structural model
--Two to Four Active Low Decoder
library ieee; use ieee.std_logic_1164.all;
use work.gates.all;
entity Two2FourLow is
  port(A,B: in std_logic;
       m0,m1,m2,m3: out std_logic);
end;
architecture structure of Two2FourLow is
  signal An, Bn: std_logic;
begin
  IN1: inv port map(A, An);
  IN2: inv port map(B, Bn);
  NG1: nand02 port map(An, Bn, m0);
  NG2: nand02 port map(A, Bn, m1);
  NG3: nand02 port map(An, B, m2);
  NG4: nand02 port map(A, B, m3);
end;
```

Behavioral modeling of decoders using Verilog and VHDL is illustrated in the following examples. Behavioral modeling is a powerful technique that can be used to describe the functionality of a logic circuit or module without having to describe its circuit realization.

```
//Verilog behavioral model
//Two to Four Active Low Decoder
module Two2FourActiveLow
(A,B,m0,m1,m2,m3);
  input A,B;
  output reg m0,m1,m2,m3;
  always
```

```
case ({B,A})
  2'b00: begin m0=0; m1=1; m2=1; m3=1; end
  2'b01: begin m0=1; m1=0; m2=1; m3=1; end
  2'b10: begin m0=1; m1=1; m2=0; m3=1; end
  2'b11: begin m0=1; m1=1; m2=1; m3=0; end
endcase
endmodule
```

```
--VHDL behavioral model
--Two to Four Active Low Decoder
library ieee; use ieee.std_logic_1164.all;
entity Two2FourActiveLow is
  port(A,B: in std_logic;
       m0,m1,m2,m3: out std_logic);
end;
architecture behavior of Two2FourActiveLow is
  signal S: std_logic_vector(1 downto 0);
  signal M: std_logic_vector(3 downto 0);
```

```
begin
  S <= B & A;
  with S select
    M <= "1110" when "00",
    "1101" when "01",
    "1011" when "10",
    "0111" when "11",
    "1111" when others;
  m0 <= M(0); m1 <= M(1); m2 <= M(2); m3 <= M(3);
end;
```

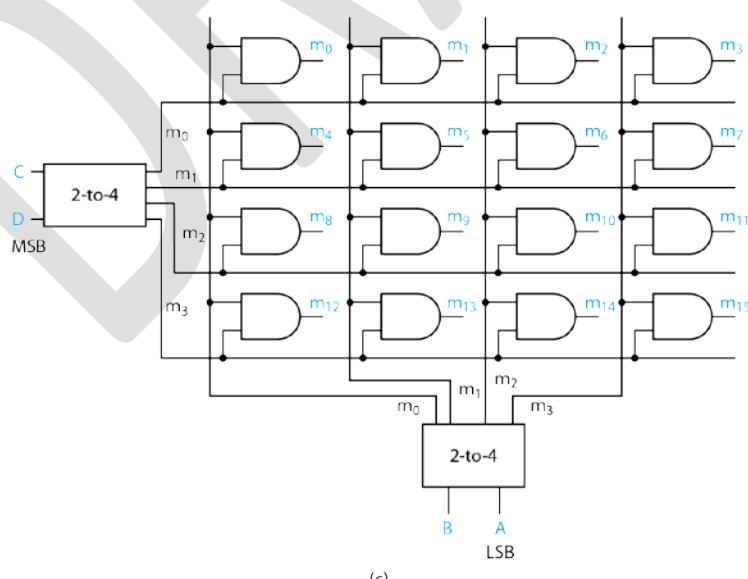
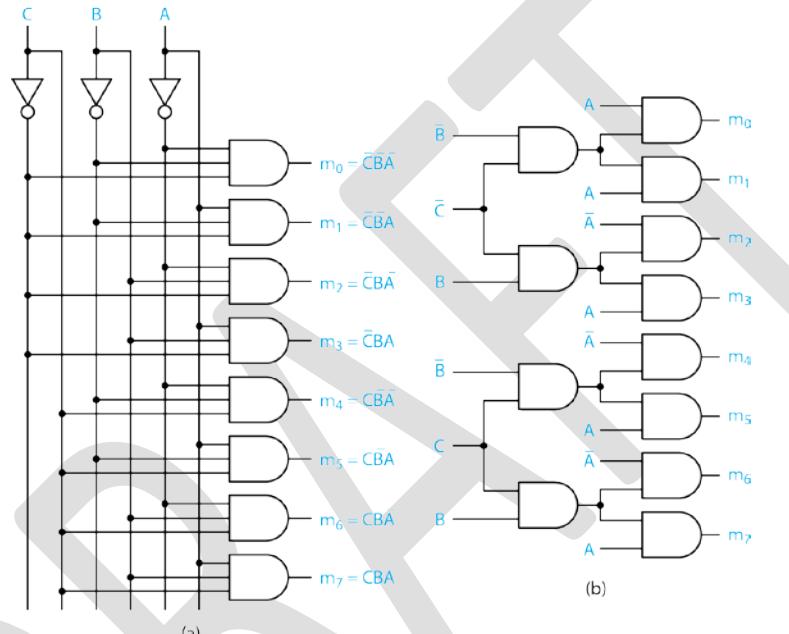


Figure 3.29. n -to- 2^n decoder structures. (a) 3-bit two-level decoder. (b) 3-bit tree-type decoder. (c) 4-bit dual-tree type decoder.

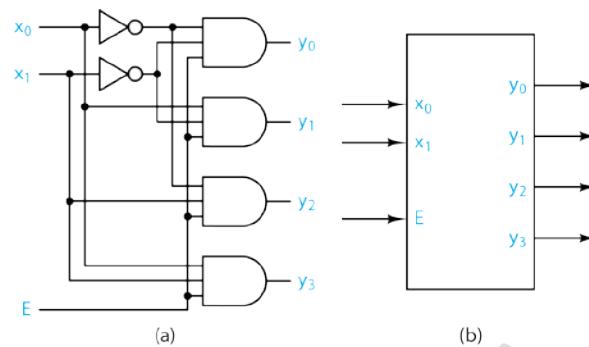


Figure 3.30. 2-to-4 decoder with enable input E . (a) Schematic diagram. (b) I/O diagram.

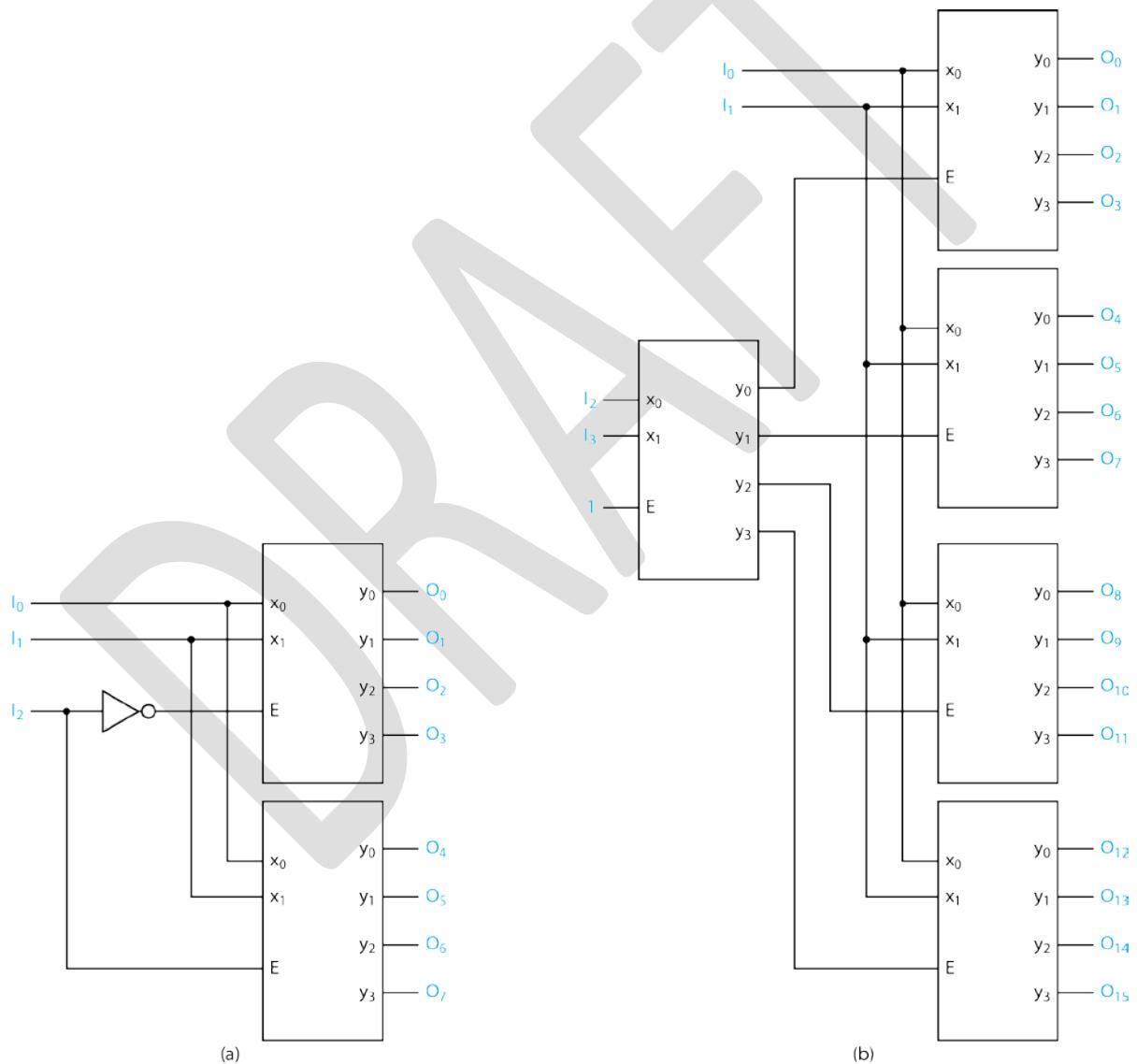


Figure 3.31. Use of 2-to-4 decoder modules to realize larger decoders. (a) 3-to-8 decoder. (b) 4-to-16 decoder.

Decoders are often realized with multiple enable signals to enhance their interconnection to built larger decoders while minimizing the need for additional gates. Enable inputs may be active-high, active-low, or a combination of the two.

Figure 3.32 describes a 3-to-8 decoder module. As seen in the logic diagram of Fig. 3.32a, the circuit has active-low outputs, and is enabled by a combination of three enable inputs: G1, G2A, and G2B. G1 is active-high while G2A and G2B are active-low. Examining a typical output, Y_i , the output equation is:

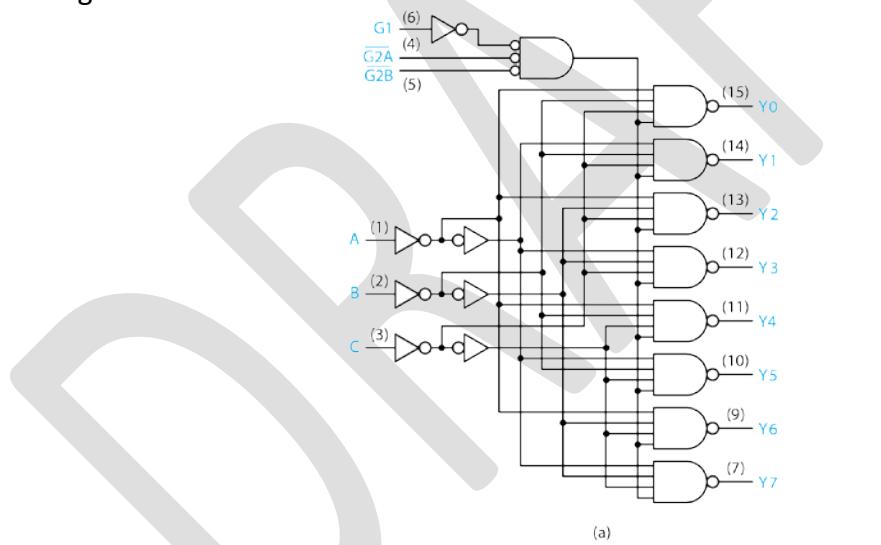
$$Y_i = m_i \cdot (\overline{G1} \cdot \overline{\overline{G2A}} \cdot \overline{\overline{G2B}}) \quad (3.5)$$

where m_i is the i th minterm of inputs C , B , and A . From this equation, note that the decoder is enabled only when $G1=1$, $\overline{G2A}=0$, and $\overline{G2B}=0$. For example,

$$Y_6 = m_6 \cdot (\overline{G1} \cdot \overline{\overline{G2A}} \cdot \overline{\overline{G2B}})$$

where $m_6 = CBA$, with C being the most significant bit in the minterm code.

A function table describing operation of the decoder is presented in Fig. 3.32b. In the table, L (low) represents a logic 0 and H (high) represents a logic 1. A generic logic symbol is shown in Fig. 3.32c.



Inputs		Outputs							
Enable	Select	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
G1	G2*	C	B	A					
H	L	L	L	L	L	H	H	H	H
H	L	L	L	H	H	L	H	H	H
H	L	L	H	L	H	H	L	H	H
H	L	L	H	H	H	H	L	H	H
H	L	H	L	L	H	H	H	L	H
H	L	H	L	H	H	H	H	L	H
H	L	H	H	L	H	H	H	H	L
X	H	X	X	X	H	H	H	H	H

$$\overline{G2^*} = \overline{G2A} + \overline{G2B}$$

(b)

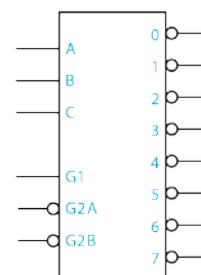


Figure 3.32. A 3-to-8 decoder module. (a) Logic circuit. (b) Function table. (c) Logic symbol.

A 4-to-16 decoder is described in Fig. 3.33. The decoder has two enable inputs, G_1 and G_2 , both of which are active low. Let us examine the logic diagram of Fig. 3.33a. Consider a typical minterm, say m_{14} , of a four-variable function. The switching expression realized by output 14 of the decoder is $\overline{DCBA}(\overline{G_1} \cdot \overline{G_2}) = m_{14}(\overline{G_1} \cdot \overline{G_2})$. So, in general,

$$Y_i = m_i(\overline{G_1} \cdot \overline{G_2}) \quad (3.6)$$

It is important to note that D is the most significant bit and A is the least significant bit of the minterm code (D, C, B, A) and that the outputs are active low (that is, when the decoder is enabled, output 14 is simply \overline{m}_{14}). In this module, two gate control signals, $\overline{G_1}$ and $\overline{G_2}$, provide the enable function, that is, the decoder's outputs are enabled only when $\overline{G_1}$ and $\overline{G_2}$ are both equal to 0 ($\overline{G_1} \cdot \overline{G_2} = 1$).

A function table for the decoder is shown in Fig. 3.33b and a logic symbol in Fig. 3.33c.

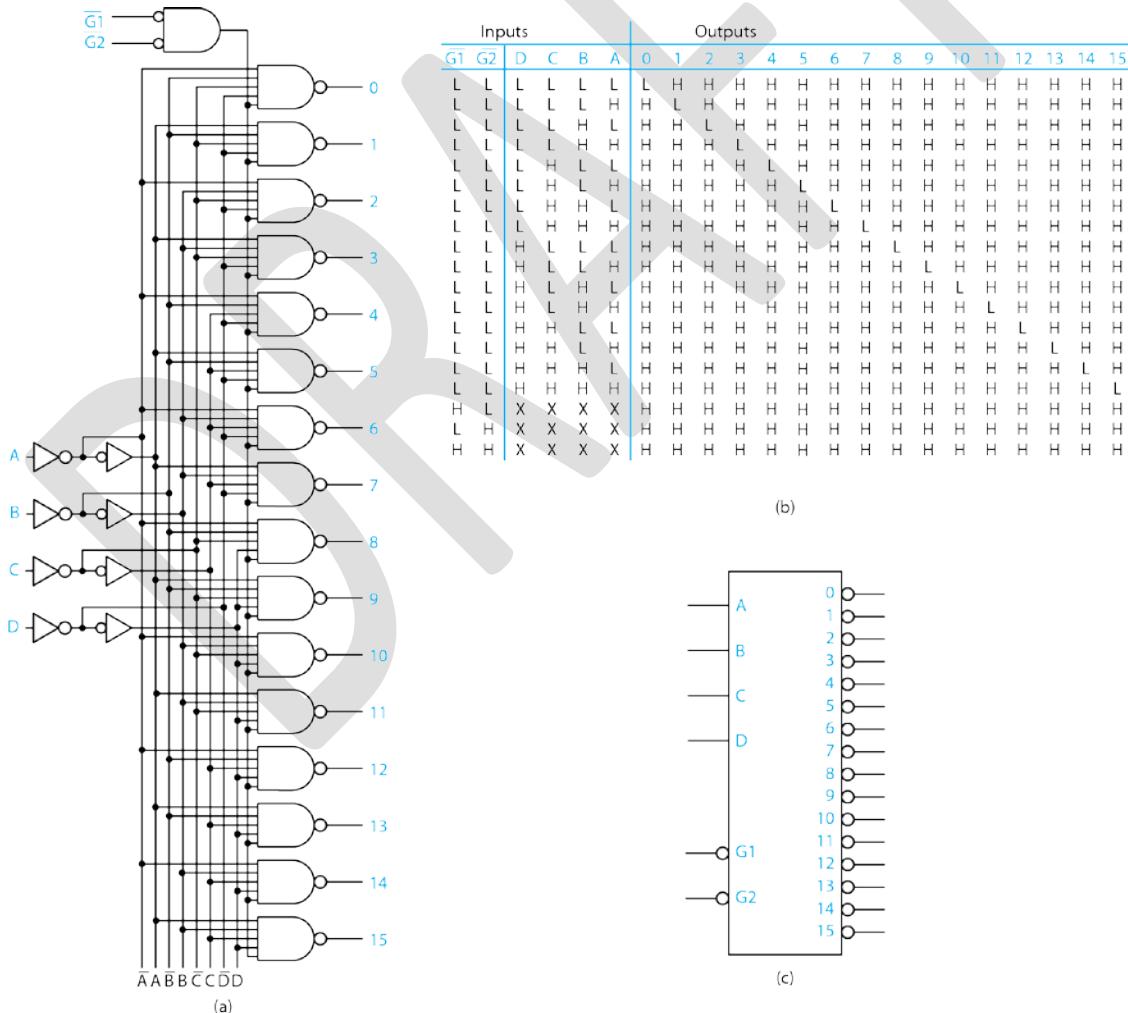


Figure 3.33. A 4-to-16 decoder module. (a) Logic circuit. (b) Function table. (c) Logic symbol.

Decoders have many applications in the design of logic circuits. However, their use as *address decoders* in computer memories and input/output systems is perhaps one of the most important. In this application, each of 2^n devices (memory cells or input/output ports) is assigned a unique n -bit binary number, or *address*, that allows it to be distinguished from other like devices. A computer designates a specific device to take part in an operation by broadcasting its address over n signal lines. As shown in Fig. 3.34, an n -to- 2^n decoder decodes the n -bit address by activating one of 2^n select lines to access one of the devices or memory locations. For example, in a computer memory, each address would correspond to one group of bits of information stored in the memory. In most memories, this group of bits would be called a byte (B) and have 8 bits. In a simple 4KB ($1K = 2^{10} = 1024$) memory, where $n=12$, a total of 4096 select, or address, lines are required. A 1MB memory would need 20 address lines and a 20-to- 2^{20} decoder. A 4GB memory requires a 32-to- 2^{32} decoder. The complexity of decoders for large memories has led to memory organizations that support independent row and column decoding thus allowing the use of dual-tree type decoders. Using this approach, a 20-to- 2^{20} decoder can be replaced by two 10-to- 2^{10} decoders resulting in much simpler decoder hardware. A 32-to- 2^{32} decoder can be replaced by two 16-to- 2^{16} decoders. Even the latter are complex decoders, so other organizational schemes have been developed but are beyond the scope of this book.

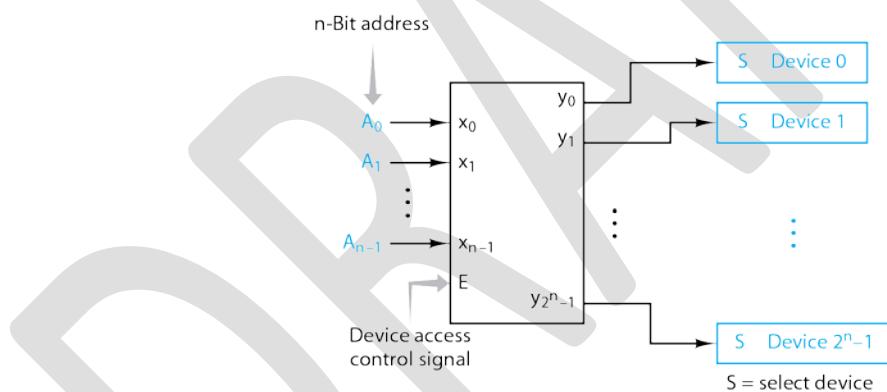


Figure 3.34. Address decoding in a digital system.

Another application of decoders is the realization of logic functions. Decoders may be used with additional logic gates to realize logic functions straight from minterm or maxterm list forms as done in PLAs. For example, decoders with active-low, or complemented, output signals may be used with either a NAND gate or an AND gate to realize a function. Decoders with active-high outputs can be used with NOR or OR gates. The following example illustrates how this can be done.

Example -- Implement the following logic functions using decoders and appropriate gates.

$$f(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$$

We may implement the function in four different ways depending on the decoder type. The four resulting implementations are shown in Figs. 3.35a through d, respectively, and explained.

1. Use a decoder with active-high outputs with an OR gate. In this case, the decoder is producing minterms and the OR gate ORs them to produce f . This is equivalent to an AND/OR realization of a canonical sum of products (CSOP) written in list for as follows.

$$f(Q, X, P) = m_0 + m_1 + m_4 + m_6 + m_7$$

2. Use a decoder with active-low outputs with a NAND gate. This is equivalent to a NAND/NAND realization of the CSOP.

$$f(Q, X, P) = \overline{\overline{m}_0 \cdot \overline{m}_1 \cdot \overline{m}_4 \cdot \overline{m}_6 \cdot \overline{m}_7}$$

3. Use a decoder with active-high outputs with a NOR gate. This is equivalent to a NOR/NOR realization of the canonical product of sums (CPOS).

$$f(Q, X, P) = \overline{m_2 + m_3 + m_5}$$

4. Use a decoder with active-low outputs with an AND gate. In this case, the decoder produces maxterms and the realization is equivalent to an OR/AND realization of the CPOS of f .

$$f(Q, X, P) = \overline{\overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_5}$$

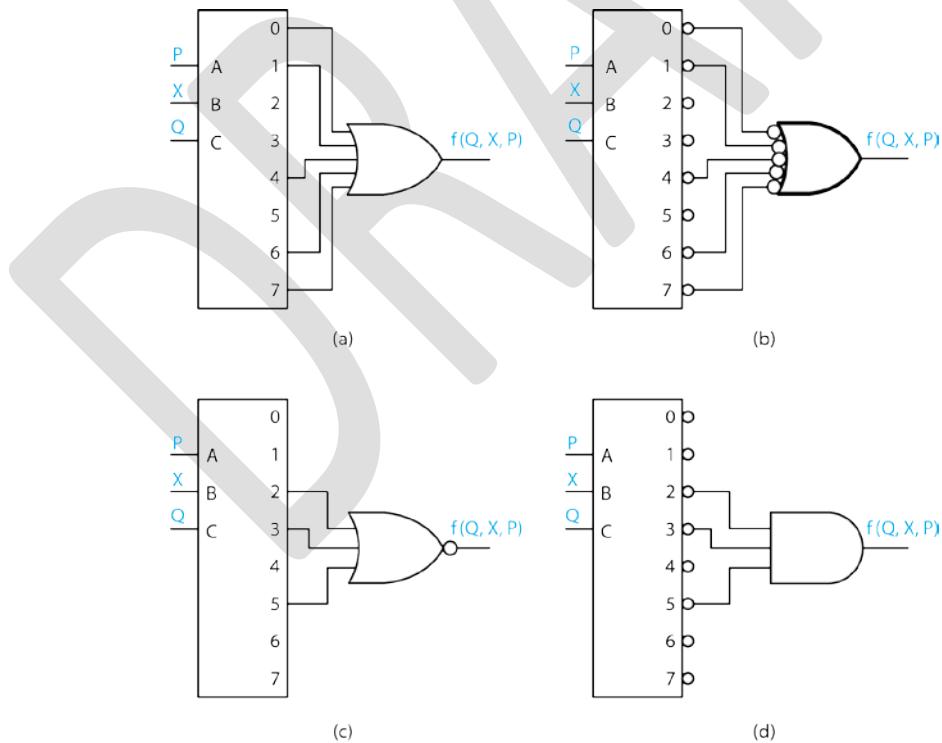


Figure 3.35. Using decoders to implement active-high logic functions. (a) Active-high decoder with OR gate. (b) Active-low decoder with NAND gate. (c) Active-high decoder with NOR gate. (d) Active-low decoder with AND gate.

In general, if $f(A, B, \dots, Z) = m_i + m_j + \dots + m_k$ then it can be realized by an active-high decoder and a k -input OR gate.

From DeMorgan's theorem $f(A, B, \dots, Z) = \overline{\overline{m}_i \cdot \overline{m}_j \cdot \dots \cdot \overline{m}_k}$ (3.7)

And can be realized by an active-low decoder and a k -input NAND gate.

We may also consider each decoder output to represent a maxterm of a function since

$$M_i = \overline{m}_i$$

Therefore, a function can be implemented from its maxterm list canonical form:

$$f(A, B, \dots, Z) = M_1 \cdot M_2 \cdot \dots \cdot M_k \quad (3.8)$$

using a decoder with active-low outputs and an p -input AND gate or a decoder with active-high outputs with a p -input NOR gate.

The following example illustrates how a decoder and gates can be used to efficiently realize a circuit that has two outputs. Realize the following functions using a four-to-sixteen decoder and logic gates.

$$f_1(W, X, Y, Z) = \sum m(1, 9, 12, 15)$$

$$f_2(W, X, Y, Z) = \sum m(0, 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 14, 15)$$

Using implementations 2 and 3 from Example 4.1:

$$f_1(W, X, Y, Z) = \overline{\overline{m}_1 \overline{m}_9 \overline{m}_{12} \overline{m}_{15}} \quad \text{and} \quad f_2(W, X, Y, Z) = \overline{m}_6 \cdot \overline{m}_9$$

Therefore, we may use a decoder, a NAND4 and an AND2 to realize f_1 and f_2 as shown in Figure 3.36. Note that we must connect $W = D$, $X = C$, $Y = B$, and $Z = A$ since D is the most significant input and A the least significant input of the decoder. Also, note that the enable signals must be grounded in this application.

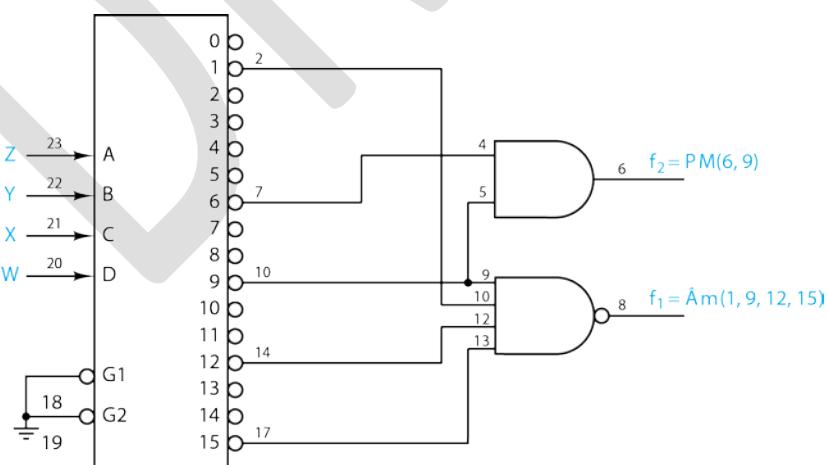


Figure 3.36. Realization of two logic functions with a decoder and gates.

Combinational logic circuits are often used to convert data encoded in one code into another code or format. Code converters are used for converting BCD to decimal, excess-3 to decimal, binary to excess-3, and so forth. For example, a BCD to decimal decoder is shown in Fig. 3.37a, with the BCD codes and their corresponding digits shown in Fig. 3.37b. This decoder is similar to the 4-to-16 decoder described earlier, but with only 10 outputs, one for each decimal digit.

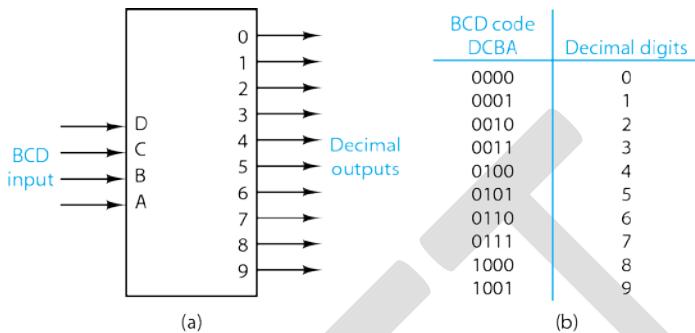


Figure 3.37. BCD to decimal decoder. a Logic symbol. b BCD codes and decimal digits.

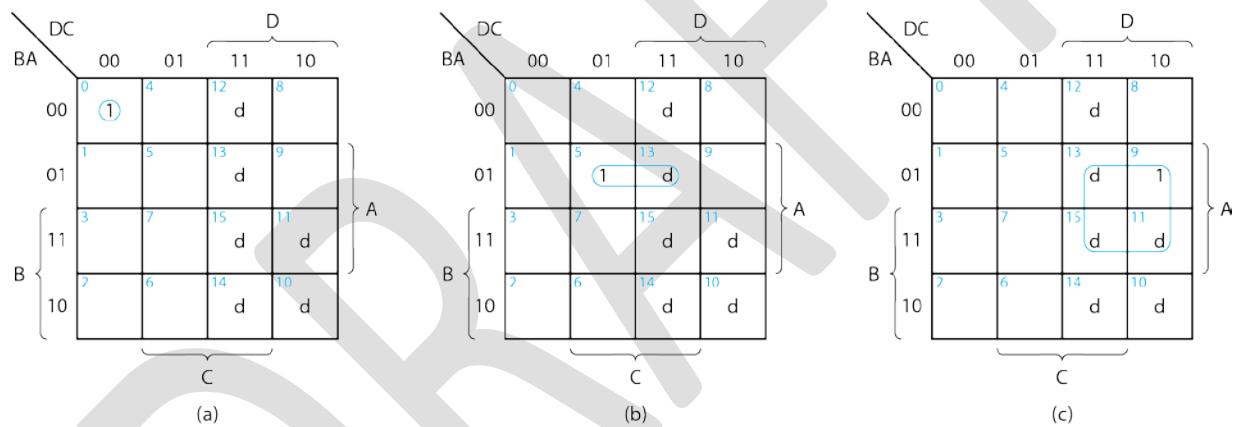


Figure 3.38. K-maps for outputs 0, 5, and 9 of a BCD to decimal code converter. (a) Decimal $0 = \overline{DCBA}$. (b) Decimal $5 = \overline{CBA}$. (c) Decimal $9 = DA$.

To design the BCD to decimal decoder, we can draw a K-map for each of the 10 outputs and derive its logic equation. Each K-map contains exactly one minterm, corresponding to the decimal number of that output, and six don't-care terms, 10 to 15, since these numbers don't exist in BCD code. Three of the K-maps are shown in Fig. 3.38. The complete set of logic equations that describes the BCD decoder is shown below.

Decimal 0	\overline{DCBA}	Decimal 5	\overline{CBA}
1	\overline{DCBA}	6	\overline{CBA}
2	\overline{CBA}	7	CBA
3	\overline{CBA}	8	DA
4	CBA	9	DA

Since the binary and BCD codes are identical for the digits 0 to 9, a 4-to-16 binary decoder can also be used to implement a BCD to decimal decoder by simply using outputs \overline{m}_0 through \overline{m}_9 and ignoring outputs \overline{m}_{10} through \overline{m}_{15} . The four-to-sixteen decoder may be more expensive than a minimum realization of the logic equations for the BCD decoder listed here, but it would be a convenient solution.

Verilog and VHDL models of an active high BCD to decimal decoder are shown below. Note that the first is a dataflow model is based on Boolean expressions as derived above and the other is a behavioral model using a case statement that follows directly from the truth table in Fig. 3.37 (b). The latter example declares the outputs as a ten-bit register $d[9:0]$ rather than ten individual bits. One advantage of this is that the assignment statements for each case can be made with one statement rather than ten. An error output is defined to indicate when an invalid BCD code is applied to the input.

```
//Verilog dataflow model
//Active-high Binary to BCD decoder
module Binary2BCDFunctional (A,B,C,D,d0,d1,d2,d3,d4,d5,d6,d7,d8,d9);
    input A,B,C,D;
    output d0,d1,d2,d3,d4,d5,d6,d7,d8,d9;
    assign d0 = ~D&~C&~B&~A;
    assign d1 = ~D&~C&~B&A;
    assign d2 = ~C&B&~A;
    assign d3 = ~C&B&A;
    assign d4 = C&~B&~A;
    assign d5 = C&~B&A;
    assign d6 = C&B&~A;
    assign d7 = C&B&A;
    assign d8 = D&~A;
    assign d9 = D&A;
endmodule

--VHDL dataflow model
-- Active-high Binary to BCD decoder
library ieee;
use ieee.std_logic_1164.all;
entity Binary2BCDFunctional is
    port (A,B,C,D: in std_logic;
          Y: out std_logic_vector(9 downto 0));
end;
architecture equations of Binary2BCDFunctional is
begin
    Y(0) <= not D and not C and not B and not A;
    Y(1) <= not D and not C and not B and A;
    Y(2) <= not C and B and not A;
    Y(3) <= not C and B and A;
    Y(4) <= C and not B and not A;
    Y(5) <= C and not B and A;
    Y(6) <= C and B and not A;
    Y(7) <= C and B and A;
    Y(8) <= D and not A;
    Y(9) <= D and A;
end;
```

```
//Verilog behavioral model
//Active-high Binary to BCD decoder
module Binary2BCDCase (A,B,C,D,d,error);
  input A,B,C,D;
  output reg [9:0] d;
  output reg error;
  always
    case ({D,C,B,A})
      4'b0000: begin d = 10'h1; error = 1'b0; end
      4'b0001: begin d = 10'h2; error = 1'b0; end
      4'b0010: begin d = 10'h4; error = 1'b0; end
      4'b0011: begin d = 10'h8; error = 1'b0; end
      4'b0100: begin d = 10'h10; error = 1'b0; end
      4'b0101: begin d = 10'h20; error = 1'b0; end
      4'b0110: begin d = 10'h40; error = 1'b0; end
      4'b0111: begin d = 10'h80; error = 1'b0; end
      4'b1000: begin d = 10'h100; error = 1'b0; end
      4'b1001: begin d = 10'h200; error = 1'b0; end
      4'b1010: begin d = 10'h400; error = 1'b1; end
      4'b1011: begin d = 10'h800; error = 1'b1; end
      4'b1100: begin d = 10'h1000; error = 1'b1; end
      4'b1101: begin d = 10'h2000; error = 1'b1; end
      4'b1110: begin d = 10'h4000; error = 1'b1; end
      4'b1111: begin d = 10'h8000; error = 1'b1; end
    endcase
  endmodule
```

```
--VHDL behavioral model
-- Active-high Binary to BCD decoder
library ieee; use ieee.std_logic_1164.all;
entity Binary2BCDCase is
  port (A,B,C,D: in std_logic;
        Y: out std_logic_vector(9 downto 0);
        error: out std_logic);
end;
architecture behavioral of Binary2BCDCase is
  signal S: std_logic_vector(3 downto 0);
begin
  S <= D & C & B & A;
  with S select
    Y <= "0000000001" when "0000",
    "0000000010" when "0001",
    "0000000100" when "0010",
    "0000001000" when "0011",
    "0000010000" when "0100",
    "0000100000" when "0101",
    "0001000000" when "0110",
    "0010000000" when "0111",
    "0100000000" when "1000",
    "1000000000" when "1001",
    "0000000000" when others;
  error <= '1' when S > "1001" else '0';
end;
```

Another common decoding application is the conversion of encoded data to a format suitable for driving a numeric or alphanumeric display. For example, digital watches and other electronic equipment display BCD-encoded decimal digits on seven-segment displays. The design of a seven-segment display decoder will be illustrated later in this chapter.

3.3.2 Encoders

An *encoder* is a combinational logic module that assigns a unique output code (a binary number) for each input signal applied to the device; as such, it is the opposite of a decoder. If an encoder module has n inputs, the number of outputs s must satisfy the expression

$$2^s \geq n \quad (3.9)$$

or

$$s \geq \log_2 n$$

Consider the case in which encoder inputs are mutually exclusive; that is, one (and only one) of the input lines is active at any particular instant in time, i.e., two or more input lines are never simultaneously active. The input combinations that never occur are treated as don't-care conditions. The following example illustrates the design of such an encoder.

Design an encoder for four input lines assuming that one and only one line is active at any moment in time. See Fig. 3.39 (a). Let us define the code as follows.

	A_1	A_0
$X_0 \rightarrow$	0	0
$X_1 \rightarrow$	0	1
$X_2 \rightarrow$	1	0
$X_3 \rightarrow$	1	1

The output functions yield the binary value of the input variable's subscript. The truth table and K-maps for the encoder are presented in Figs. 3.39 (b) and (c). From the K-maps,

$$A_1 = X_3 + X_2 \quad (3.10)$$

$$A_0 = X_3 + X_1 \quad (3.11)$$

The logic circuit diagram for the encoder is given in Fig. 3.39 (d).

Verilog and VHDL behavioral models of the encoder are shown below.

```
//Verilog behavioral model
//Four to Two Encoder
module Four2TwoEncoder (x3,x2,x1,x0,A1,A0);
    input x3,x2,x1,x0;
    output reg A1,A0;
    always begin
        if (x1 || x3) A0 = 1'b1; else A0 = 1'b0;
        if (x2 || x3) A1 = 1'b1; else A1 = 1'b0;
    end
endmodule
```

```
--VHDL behavioral model
--Four to Two Encoder
library ieee; use ieee.std_logic_1164.all;
entity Four2TwoEncoder is
    port (x3,x2,x1,x0: in std_logic;
          A1,A0: out std_logic);
end;
architecture behavioral of Four2TwoEncoder is
begin
    A0 <= '1' when x1 = '1' or x3 = '1' else '0';
    A1 <= '1' when x2 = '1' or x3 = '1' else '0';
end;
```

Another type of encoder is the *priority encoder*. The priority encoder allows multiple input lines to be active simultaneously and encodes the binary value of the subscript of the input line with highest priority. Priority encoders are particularly useful for implementing prioritized interrupts in the interfacing of computer processing units to peripheral devices. To simplify the design, the highest priority is assigned to the highest subscript, the next highest priority to the second highest subscript, and so on. Consider the priority encoder of Fig. 3.40. The input lines are encoded as follows.

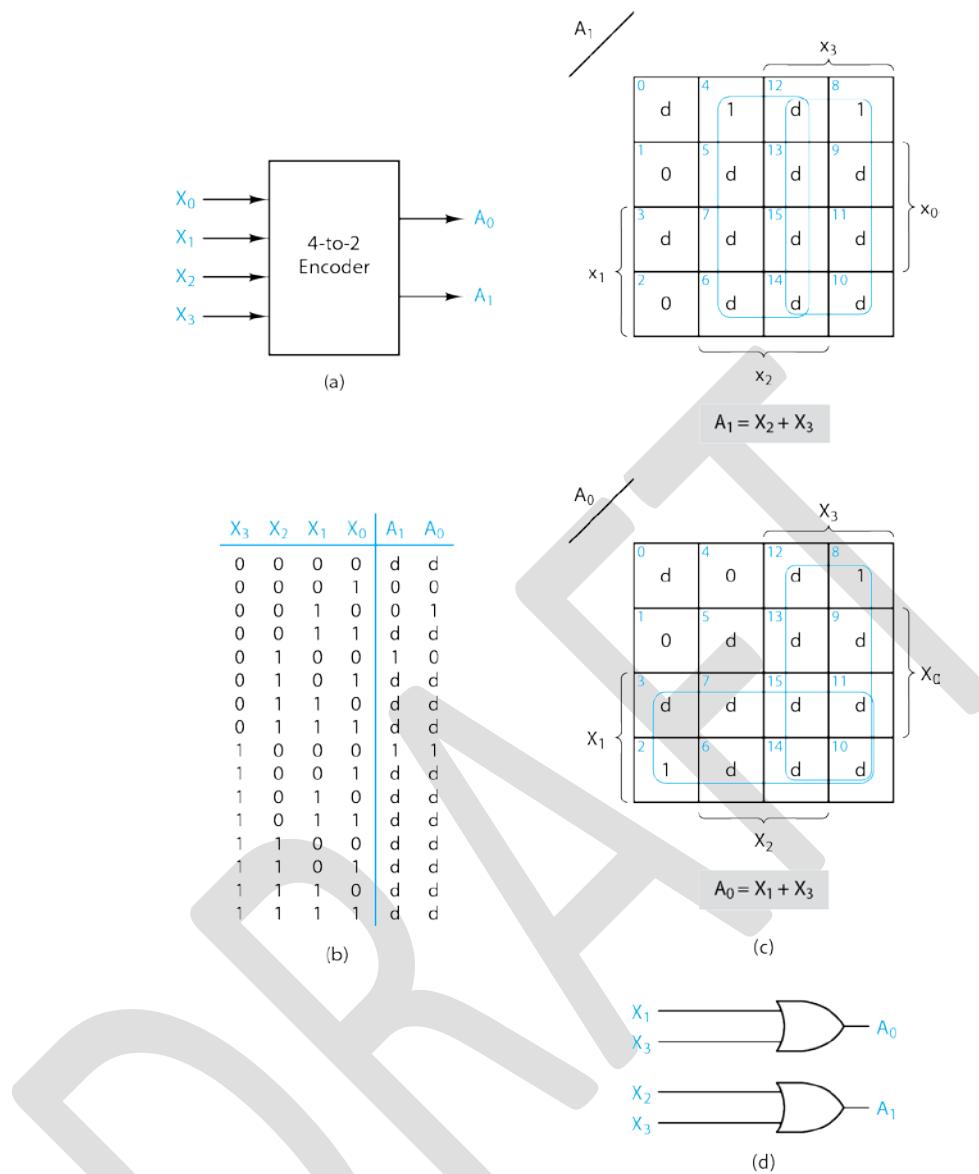


Figure 3.39. Four-to-two line encoder. (a) Functional diagram. (b) Truth table. (c) K-maps. (d) Logic diagram.

	A_1	A_0	GS
$X_0 \rightarrow$	0	0	1
$X_1 \rightarrow$	0	1	1
$X_2 \rightarrow$	1	0	1
$X_3 \rightarrow$	1	1	1
<i>None</i>	d	d	0

If no input line is active, the priority encoder outputs $(A_1 A_0) = (00)$. If a single line is active, the encoder outputs the binary value of the subscript of the active line. If more than one

input is active, the encoder outputs the binary value of the largest subscript of the active lines. Figure 3.40 (b) displays the truth table for the encoder. Note that the two additional output lines indicate that no input line is active ($EO = 1$) and one or more inputs are active ($GS = 1$). Figures 3.40 (c) and (d) present the K-map and logic diagram of the function, which reduces to

$$A_1 = X_2 + X_3 \quad (3.12)$$

$$A_0 = X_3 + X_1\bar{X}_2 \quad (3.13)$$

and

$$EO = \overline{GS} = \overline{X_3 + X_2 + X_1 + X_0} \quad (3.14)$$

The two output functions A_1 and A_0 are independent of X_0 . Verilog and VHDL models of the priority encoder are shown below.

```
//Verilog behavioral model
//Four to Two Priority Encoder
module Four2TwoPriorityEncoder (x3,x2,x1,x0,A0,A1,GS,EO);
input x3,x2,x1,x0;
output reg A0,A1,GS,EO;
always begin
  if (x2 || x3) A1 = 1'b1; else A1 = 1'b0;
  if (x1&&!x2 || x3) A0 = 1'b1; else A0 = 1'b0;
  if (x3||x2||x1||x0)
    begin GS = 1'b1; EO = 1'b0; end
    else begin GS = 1'b0; EO = 1'b1; end
end
endmodule

--VHDL behavioral model
--Four to Two Priority Encoder
library ieee; use ieee.std_logic_1164.all;
entity Four2TwoEncoder is
port (x3,x2,x1,x0: in std_logic;
      A1,A0:     out std_logic;
      GS,EO:     out std_logic);
end;
architecture behavioral of Four2TwoEncoder is
  signal GE: std_logic;
begin
  A0 <= '1' when (x1 = '1' and x2 = '0') or x3 = '1' else '0';
  A1 <= '1' when x2 = '1' or x3 = '1' else '0';
  GE <= '1' when x3 = '1' or x2 = '1' or x1 = '1' or x0 = '1' else '0';
  GS <= GE;
  EO <= not GE;
end;
```

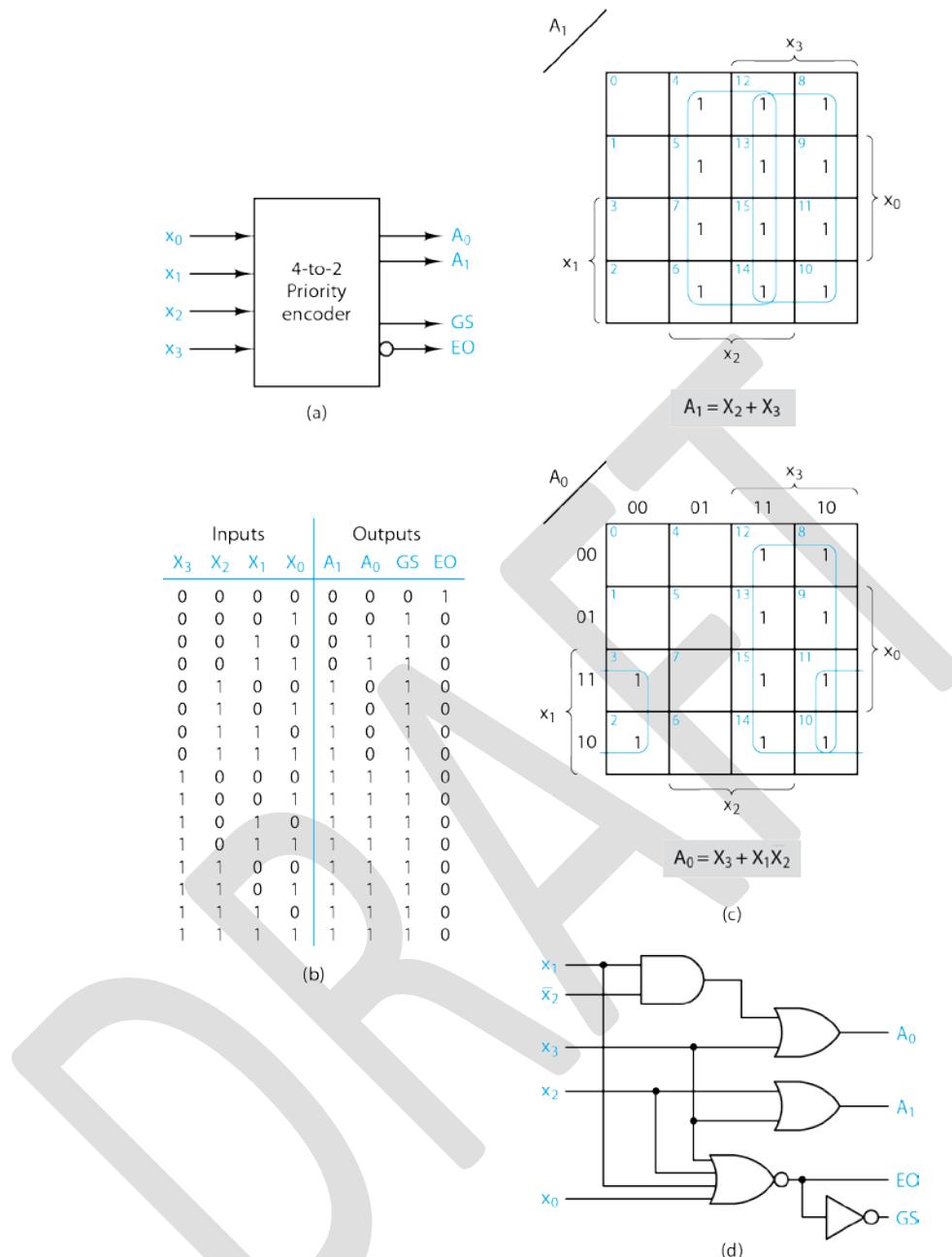


Figure 3.40. Four-to-two line priority encoder. (a) Functional diagram. (b) Truth table. (c) K-maps. (d) Logic diagram.

An 8-bit to 3-bit priority encoder module is described in Fig 3.41. The encoder takes eight lines ($0, 1, \dots, 7$) and encodes them to three lines (A_2, A_1, A_0) according to the function table in Fig. 3.41b. Note that all are active low. An input enable signal EI is connected to all the first-level logic gates to control their operation; when EI is active (low) the circuit operates. The encoder also has two additional output signals, EO and GS . EO is active (low) when none of the input lines is active. GS is active (low) when one or more

input lines are active. These outputs can be used to connect multiple encoder modules to encode more than eight bits.

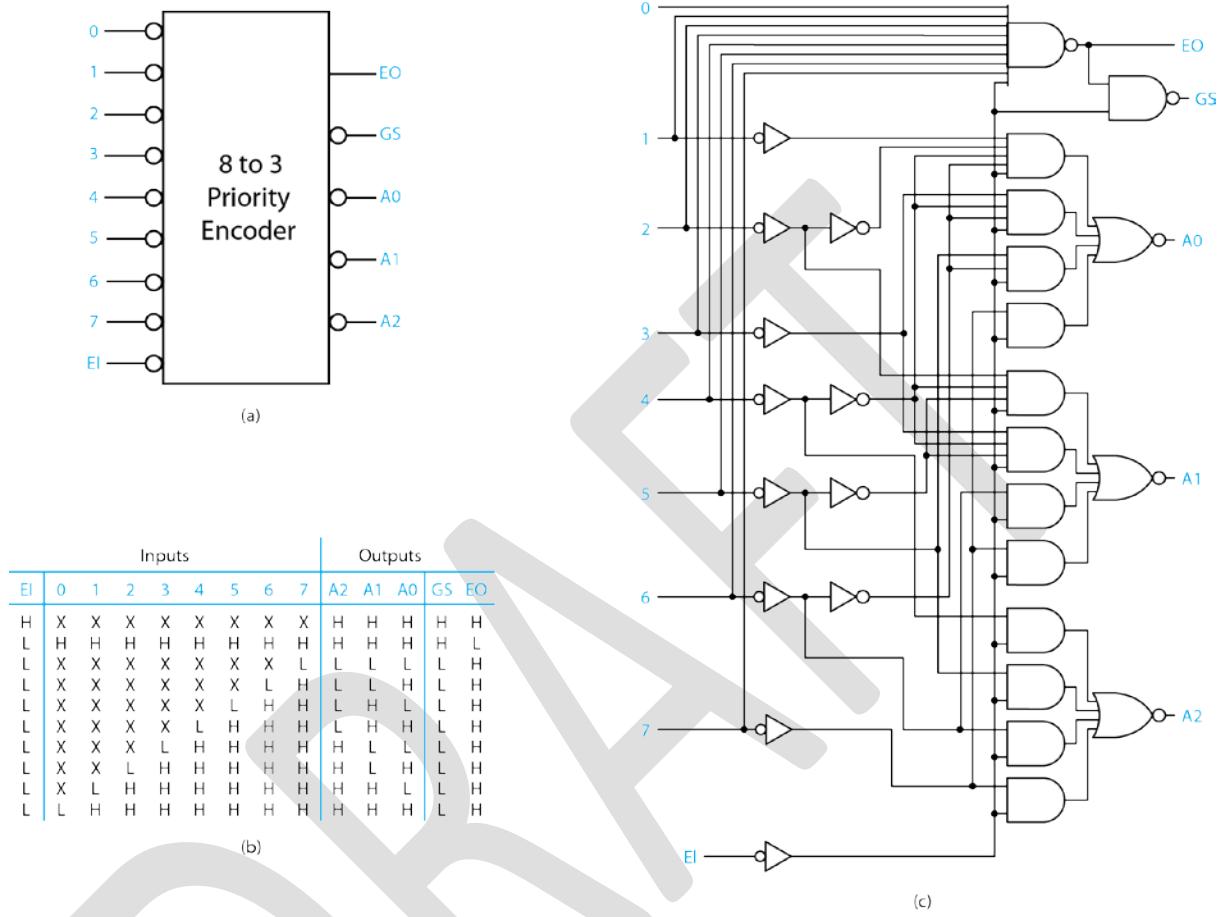


Figure 3.41. An 8-bit to 3-bit priority encoder. (a) Generic symbol, (b) Truth table, (c) Logic circuit.

Example. For the above encoder, what is the output code ($EO, GS, A2, A1, A0$) under the following conditions?

$$(EI, 7, 6, 5, 4, 3, 2, 1, 0) = (0, 1, 0, 1, 0, 1, 0, 1, 1)$$

Since the device is enabled ($EI = 0$) and three lines are active (inputs 6, 4, and 2), GS will be active low and $A2$, $A1$, $A0$ will encode line 6 (001). Therefore,

$$(EO, GS, A2, A1, A0) = (1, 0, 0, 0, 1)$$

3.3.3 Multiplexers and Demultiplexers

A *multiplexer*, or *data selector*, is a logic device that selects one of multiple input lines to connect to a single output line. A *demultiplexer*, or *data distributor*, performs the inverse operation; it takes a single input line and connects, or routes, it to one of several output lines.

The general concept of multiplexing and demultiplexing is shown in Fig. 3.42 (a). The rotary switch SW_1 moves from input line A to B to C , and so on. The rotary switch SW_2 at the output of the channel is synchronized to SW_1 and it too moves from output line A to B to C , and so on. This multiplex/demultiplex configuration illustrates one manner in which data are selected and routed. The logic configuration is shown in Fig. 3.42 (b). Here the signals a, b, \dots, k are control signals that select which set of inputs/outputs will be using the "single channel." The channel in this configuration could be contained within a computer system or could be a mechanism with which the computer communicates with the outside world.

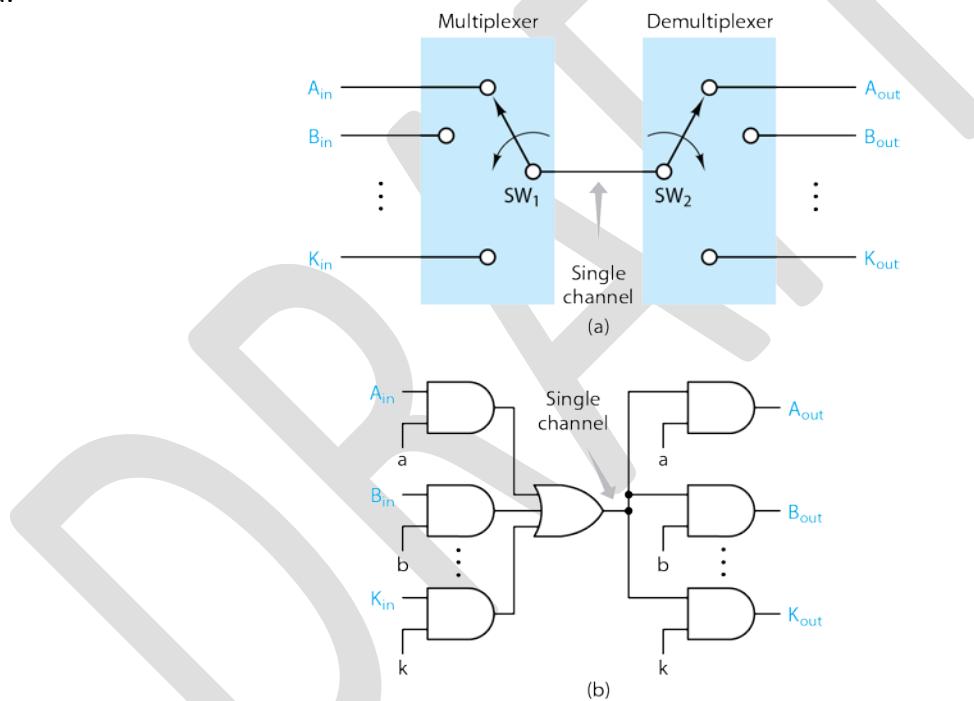


Figure 3.42. K-channel multiplexing/demultiplexing system. (a) Multiplex/demultiplex operation. (b) Simple logic configuration for (a).

For an n -to-1 line multiplexer, one of the n input data lines ($D_{n-1}, D_{n-2}, \dots, D_0$) is selected for connection to the single output line (Y) by a selection code (S_{k-1}, \dots, S_0), where $n = 2^k$. A four-to-one multiplexer will now be illustrated as shown in Figure 3.43, where $B = S_1$ and $A = S_0$. The circuit will connect data line D_i to the output Y when the code

$$i = (BA)_2 \quad (3.15)$$

is applied to the selection terminals. Figure 3.43 (b) displays the truth table of the multiplexer.

From the truth table we observe that

$$Y = (\overline{BA})D_0 + (\overline{B}A)D_1 + (B\overline{A})D_2 + (BA)D_3 \quad (3.16)$$

The selection code forms the minterms of two variables, B and A . Hence we may write

$$Y = \sum_{i=0}^3 m_i D_i \quad (3.17)$$

where the m_i are the minterms of the selection code. The logic circuit for the 4-to-1 multiplexer is shown in Fig. 3.43 (c). An equivalent circuit, using two-level AND-OR logic, is presented in Fig. 3.43(d).

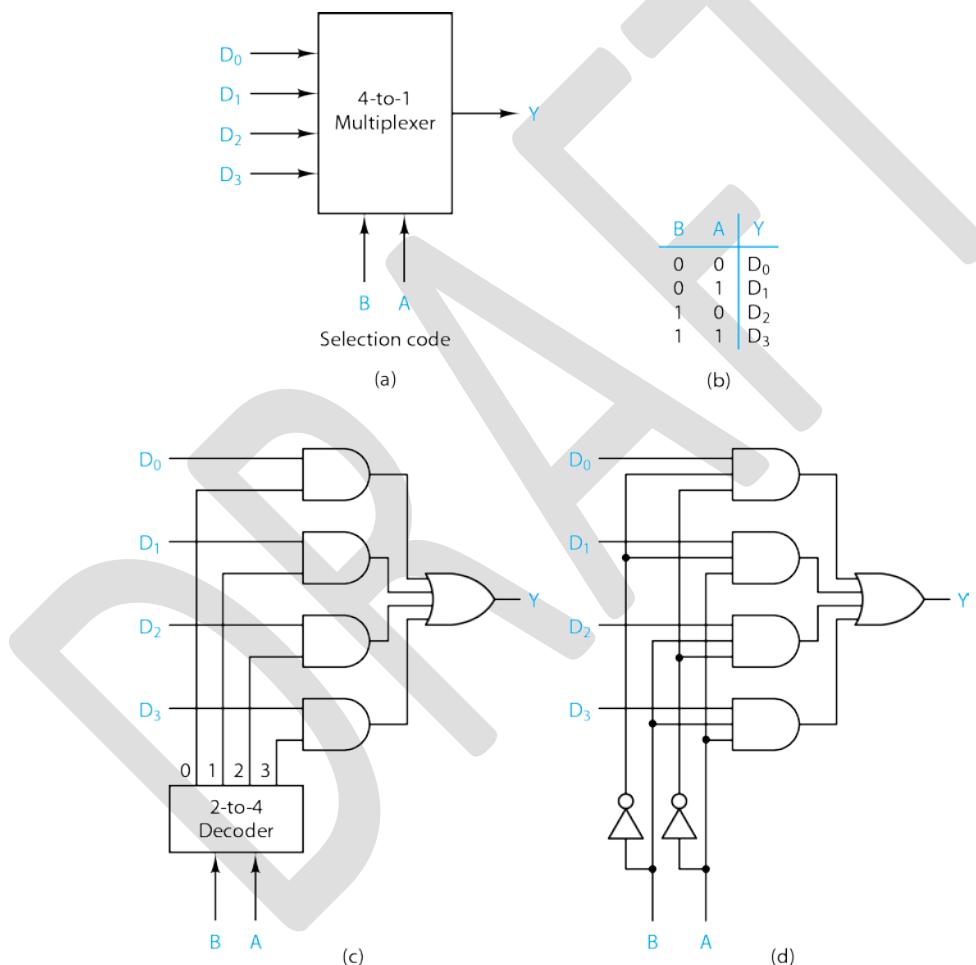


Figure 3.43. Four-to-one multiplexer design. (a) Logic symbol. (b) Truth table. (c) Logic circuit. (d) Equivalent two-level circuit.

Multiplexers can be modeled in Verilog and VHDL very effectively as shown in the two examples below. The first models the four-to-one multiplexer from Fig. 3.43 and the second an eight-to-one multiplexer with *gate* (*G*) or *strobe* input.

```
//Verilog behavioral model
//Four to One Multiplexer
module Four2OneMUX (A,B,D0,D1,D2,D3,Y);
input A,B,D0,D1,D2,D3;
output reg Y;
always
  case ({B,A})
    2'b00: Y = D0;
    2'b01: Y = D1;
    2'b10: Y = D2;
    2'b11: Y = D3;
  endcase
endmodule
```

```
--VHDL behavioral model
-- Four to One Multiplexer
library ieee;  use ieee.std_logic_1164.all;
entity Four2OneMUX is
  port (D0,D1,D2,D3: in std_logic;
        Y: out std_logic;
        S: in std_logic_vector(1 downto 0));
end;
architecture behavioral of Four2OneMUX is
begin
  with S select
    Y <=  D0 when "00",
              D1 when "01",
              D2 when "10",
              D3 when others;
end;
```

```
//Verilog behavioral model
//Eight to One Gated Multiplexer
module Eight2OneGatedMUX (A,B,C,G,D,Y);
input A,B,C,G;
input [7:0] D;
output reg Y;
always begin
  if (G) Y = 1'b0; else
    case ({C,B,A})
      3'b000: Y = D[{C,B,A}];
      3'b001: Y = D[{C,B,A}];
      3'b010: Y = D[{C,B,A}];
      3'b011: Y = D[{C,B,A}];
      3'b100: Y = D[{C,B,A}];
      3'b101: Y = D[{C,B,A}];
      3'b110: Y = D[{C,B,A}];
      3'b111: Y = D[{C,B,A}];
    endcase
  end
endmodule
```

```
--VHDL behavioral model
-- Eight to One Gated Multiplexer
library ieee;  use ieee.std_logic_1164.all;
entity Binary2BCDFunctional is
  port (A,B,C: in std_logic;
        G: in std_logic;
        D: in std_logic_vector(7 downto 0);
        Y:  out std_logic);
end;
architecture equations of Binary2BCDFunctional is
  signal S: std_logic_vector(2 downto 0);
begin
  S <= C & B & A;
  Y <= '0' when G = '0' else
    D(0) when S = "000" else
    D(1) when S = "001" else
    D(2) when S = "010" else
    D(3) when S = "011" else
    D(4) when S = "100" else
    D(5) when S = "101" else
    D(6) when S = "110" else
    D(7) when S = "111";
end;
```

The 4-to-1 multiplexer of Fig. 3.43 can also be used in a tree-type network, as shown in Fig. 3.44 in which four multiplexers are used to feed another 4-to-1 multiplexer, thereby creating a 16-to-1 multiplexer. Even larger configurations can be generated in the same manner.

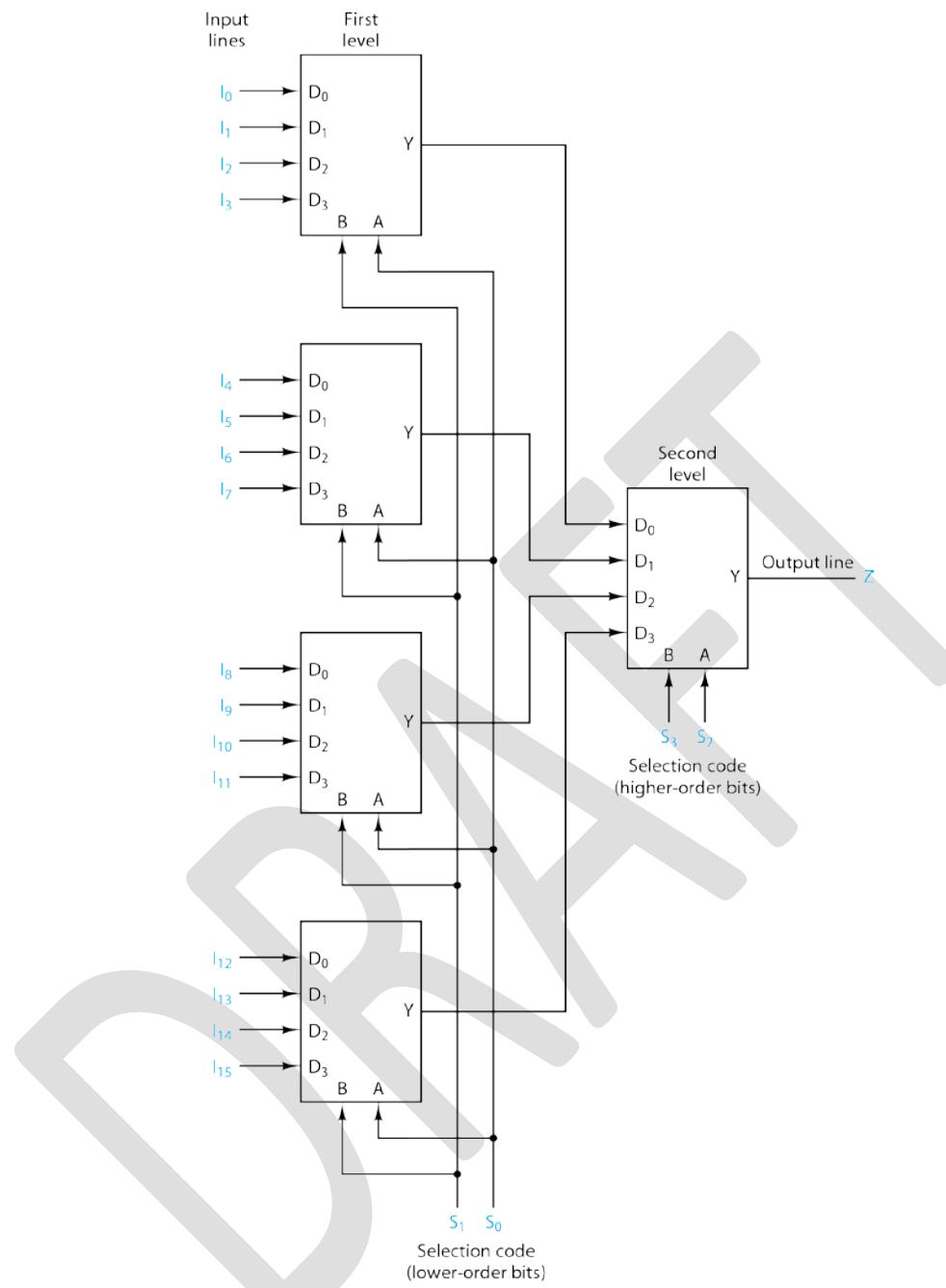


Figure 3.44. 16-to-1 multiplexer realized with a tree-type network of 4-to-1 multiplexers.

Many standard integrated circuit implementations of multiplexers have been produced over the years including an 8-to-1 multiplexer, a 16-to-1 multiplexer, a dual 4-to-one multiplexer, and a quad 2-to-1 multiplexer. An 8-to-1 and a quad 2-to-1 will be covered as illustrative examples.

An 8-to-1 multiplexer is shown in Fig. 3.45. For this circuit we may write the output equation as

$$\begin{aligned}
 Y &= [(\overline{C}\overline{B}A)D_0 + (\overline{C}\overline{B}A)D_1 + (\overline{C}\overline{B}\overline{A})D_2 + (\overline{C}BA)D_3 \\
 &\quad + (\overline{C}\overline{B}\overline{A})D_4 + (\overline{C}\overline{B}A)D_5 + (C\overline{B}\overline{A})D_6 + (CBA)D_7] \bullet \overline{G} \\
 &= \left(\sum_{i=0}^7 m_i D_i \right) \overline{G}
 \end{aligned} \tag{3.18}$$

The strobe (\overline{G}) acts as an enable signal (active low), forcing the output to 0 when $\overline{G} = 1$. The second output W is the complement of Y .

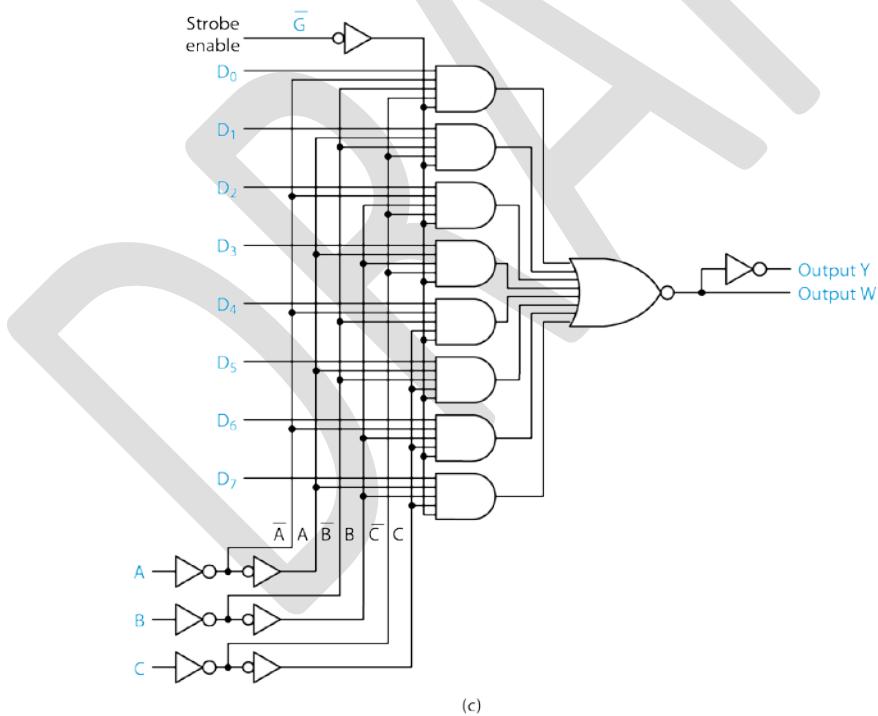
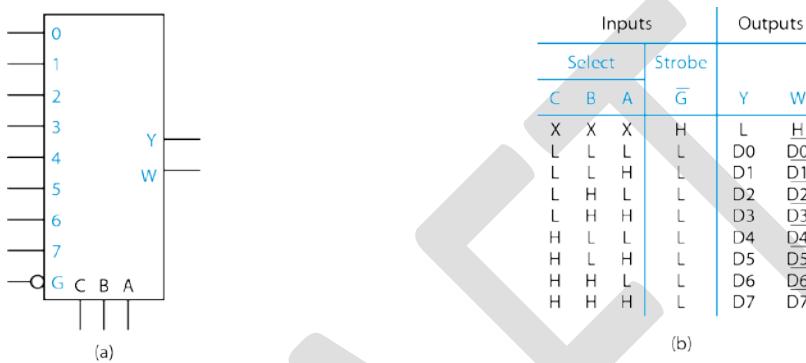


Figure 3.45. An 8-to-1 multiplexer. (a) Logic symbol. (b) Function table. (c) Logic circuit.

Figure 3.46 shows a quad (4-bit) two 2-to-1 multiplexer module, which connects one of two 4-bit inputs to the 4-bit output as selected by control signal S . Control signal \overline{G} enables and disables the output; the output lines are all forced to 0 if $\overline{G} = 1$.

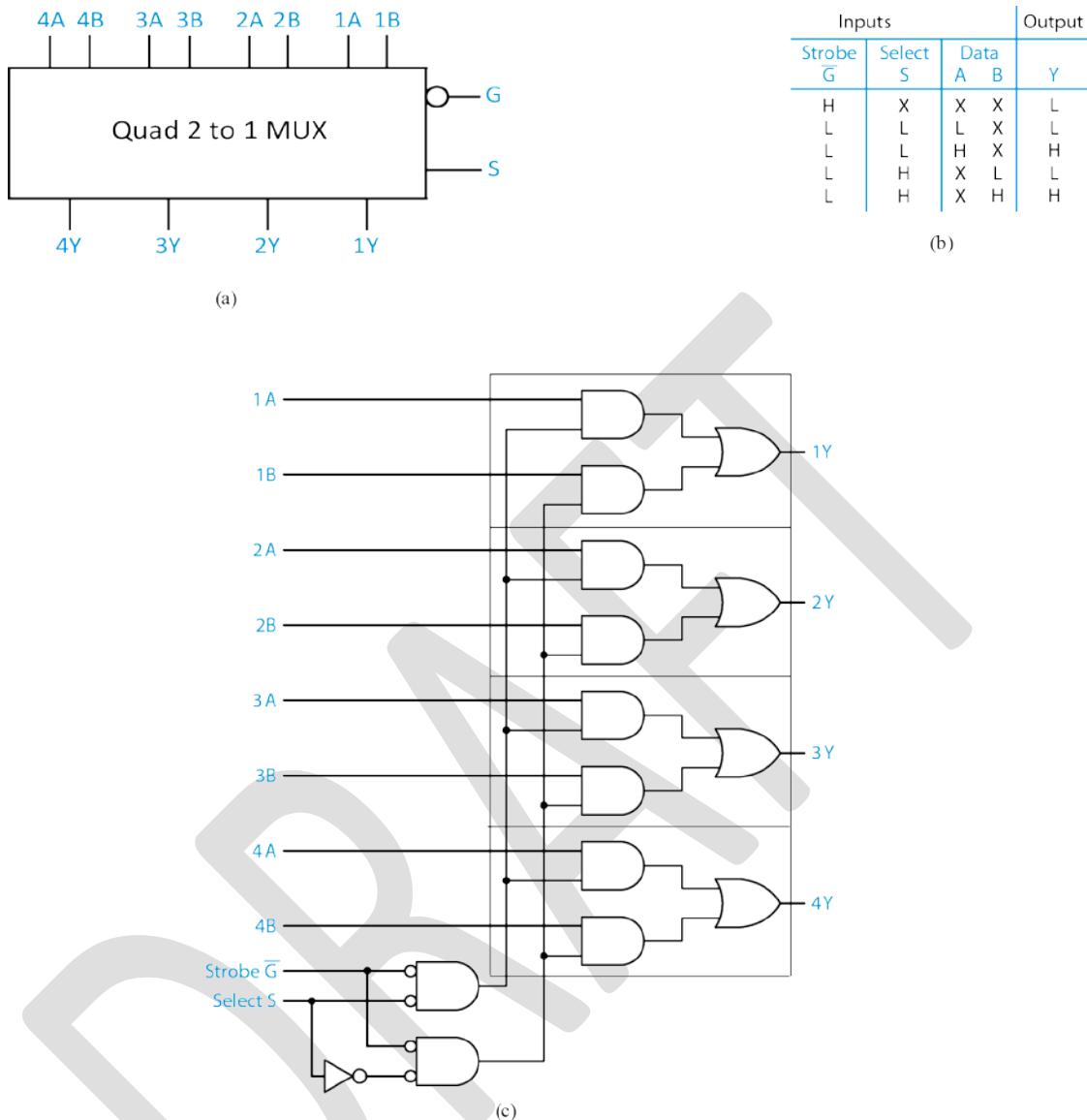


Figure 3.46. A quad (4-bit) 2-to-1 multiplexer. (a) Logic symbol. (b) Function table. (c) Logic circuit.

Multiple quad 2 to 1 modules can be utilized to create other multiplexer configurations of different path widths and numbers of inputs. In Fig. 3.47a, two quads are used to create an octal (8-bit) two-input multiplexer by controlling the select line S on both modules with the same select signal. In this case, the 8-bit input from source X is routed to the destination when select = 0, with the upper 4 bits routed through one module and the lower 4 bits through the other. Source W is routed to the destination in the same manner when select = 1.

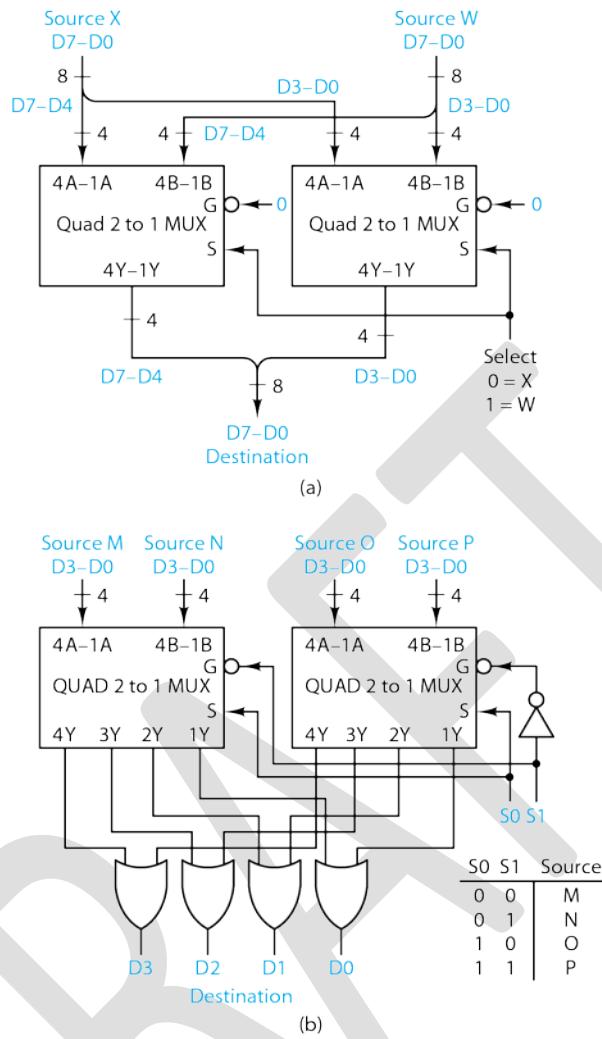


Figure 3.47. Quad 2 to 1 multiplexer applications. (a) 8-bit, 2 to 1 multiplexer realization. (b) 4-bit, 4 to 1 multiplexer.

Figure 3.47b shows a quad (4-bit) four-input multiplexer realized with two quad 2 to 1 modules modules. Select signal S_1 enables one of the two modules and disables the outputs of the other, forcing one of the two inputs to each of the OR gates in Fig. 3.47b to be 0. Select signal S_0 selects one of the two 4-bit inputs of the enabled module, sending the 4 bits of the selected source to the other OR gate inputs. Consequently, each OR gate output is simply the corresponding bit of the selected source.

The multiplexers/data selectors may also be used to implement logic functions. The fundamental idea is to use the *selection code to select the minterms* of the function, and to use the data lines D_i to *establish the minterms* present in the function. This is illustrated in the following example.

Example -- Use an 8-to-1 multiplexer to implement $f(x_1, x_2, x_3) = \sum m(0, 2, 3, 5)$. Figure 3.48(a) lists the truth table for the function. The minterms are gated onto the output Y by setting $D_0 = D_2 = D_3 = D_5 = 1$. The remaining data lines are grounded as shown in Fig. 3.48(b). Note that (x_1, x_2, x_3) are connected to (C, B, A) . The order of the variables is very important.

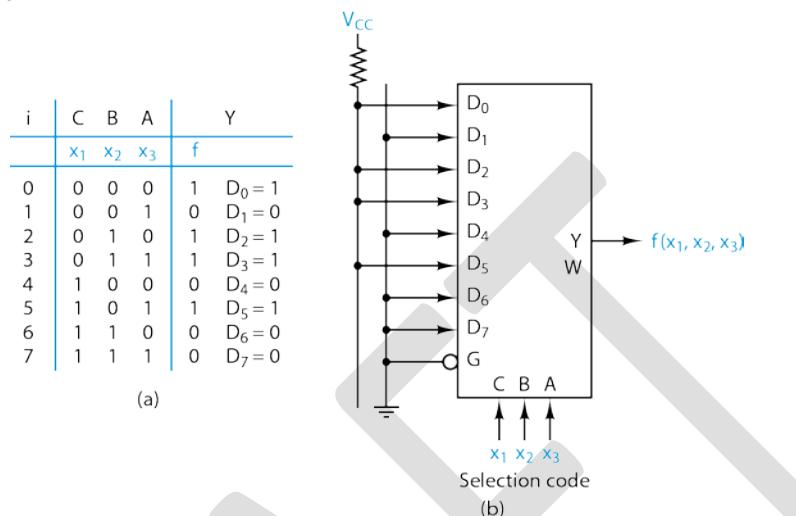


Figure 3.48. Realization of $f(x_1, x_2, x_3) = \sum m(0, 2, 3, 5)$. (a) Truth table. (b) Implementation with an 8-to-1 multiplexer.

The concept demonstrated in the previous example may be extended to implement higher-order functions. That is, an n -to-1 line multiplexer can be used to implement a $k+1$ variable function ($n = 2^k$) by connecting k of the variables to the selection lines of the multiplexer, with the $(k+1)$ st variable used (along with ground and the power supply) to establish the data input lines. This approach is illustrated by the following example.

Example -- Implement $f(a, b, c) = ab + \bar{b}c$ **using the 4-to-1 multiplexer of Fig. 3.43.**

In this case, there are three variables and two selection lines on the multiplexer. The first step is to express the function in canonical SOP form.

$$\begin{aligned} f(a, b, c) &= ab + \bar{b}c \\ &= abc + abc + \bar{a}\bar{b}c + a\bar{b}c \end{aligned}$$

The next step is to select two of the variables to connect to the multiplexer select lines and factor these terms out of the canonical SOP form. Let us use a and b for this example. Factoring out a and b gives

$$f(a, b, c) = \bar{a}\bar{b}(c) + \bar{a}\bar{b}(c) + ab(\bar{c} + c)$$

From this expression, $f(a, b, c)$ can be evaluated for each combination of a and b . The result is listed in truth table form in Fig. 3.49(a), which shows the expression for $f(a, b, c)$ for each combination of a and b . This truth table is implemented by the multiplexer of Fig. 3.49(b). Note that each row of the truth table corresponds to one of the multiplexer inputs.

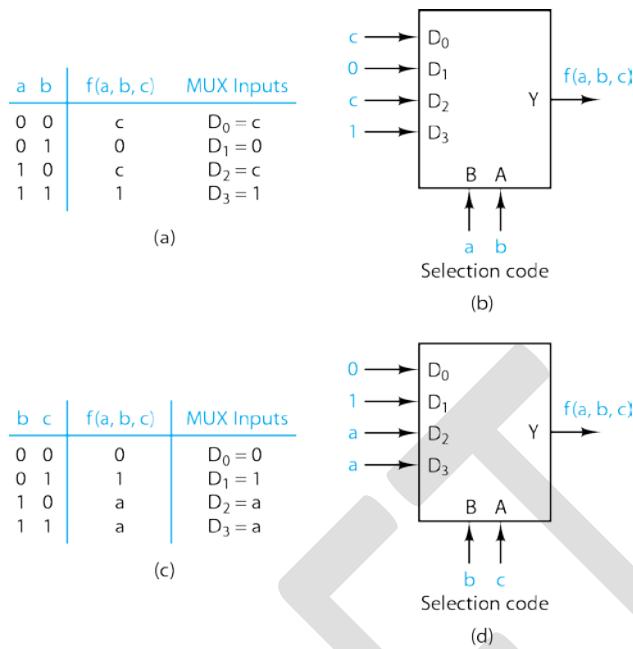


Figure 3.49. Realization of $f(a,b,c) = ab + \bar{b}c$ with 4-to-1 multiplexers. (a) Truth table; $f(a,b,c)$ evaluated for all values of a and b . (b) Multiplexer realization. (c) Alternate truth table; $f(a,b,c)$ evaluated for all values of b and c . (d) Alternate realization.

Any two variables can be connected to the multiplexer select lines. For example, if variables b and c are chosen, the truth table given in Fig. 3.49(c) results. This table is implemented by the multiplexer of Fig. 3.49(d).

A *demultiplexer*, as previously mentioned, may be considered the inverse of a multiplexer in that a demultiplexer connects a single input line to one of n output lines, the specific output line being determined by an s -bit selection code, where

$$2^s \geq n \quad (3.19)$$

A functional diagram for a 1-to- n demultiplexer is shown in Fig. 3.50(a). The selection code is used to generate a minterm of s variables; that minterm then steers the input data to the proper output terminal. See Fig. 3.50(b) for a specific example. This 1-to-4 data distributor has an enable signal (E) that controls the operation of the circuit. When E is 1, the circuit is operational. We may thus describe the operation of the device by

$$Y_i = (m_i D)E \quad (3.20)$$

where D is the input signal to be distributed to the n output lines.

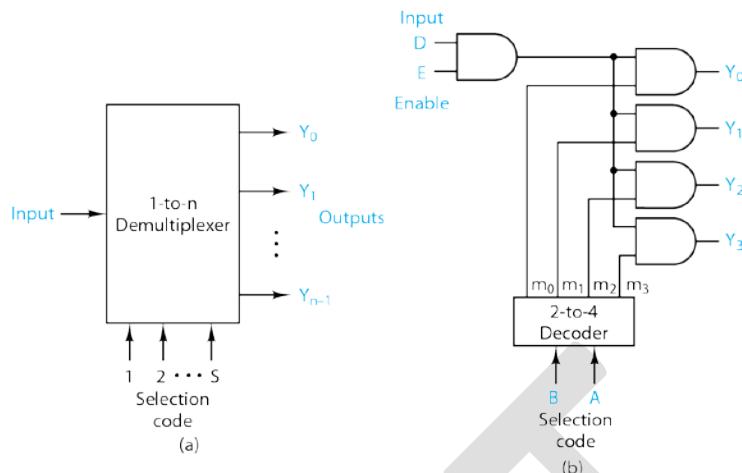


Figure 3.50. Demultiplexer/data distributor. (a) Functional diagram. (b) 1-to-4 demultiplexer with enable.

The following example will illustrate the design of a simple multiplexer/demultiplexer system using standard components introduced previously.

Example -- Design an 8-line to 8-line multiplexer/demultiplexer system.

The goal is to replace a cable of 8 signal lines with a smaller number in order to save wire. Figure 3.51 illustrates a solution in which the signals (X_0, X_1, \dots, X_7) are multiplexed onto one line (\bar{Q}) using the signal channel code (C_2, C_1, C_0). At the other end, the four lines are then used to demultiplex the data back to 8 lines for further processing. It is important to note that the 8-to-4 line reduction has come at the expense of system utility since now, at any instant in time, one and only one of the 8 signal lines may be in use. That is, time slots must be assigned to each of the 8 lines, and a time schedule for line \bar{Q} must be enforced. In this configuration, the 8 output lines are active low and will be high when they are not scheduled to use the single input channel Q . In essence, the multiplexer acts as a parallel-to-serial converter and the demultiplexer converts the serial data back to parallel.

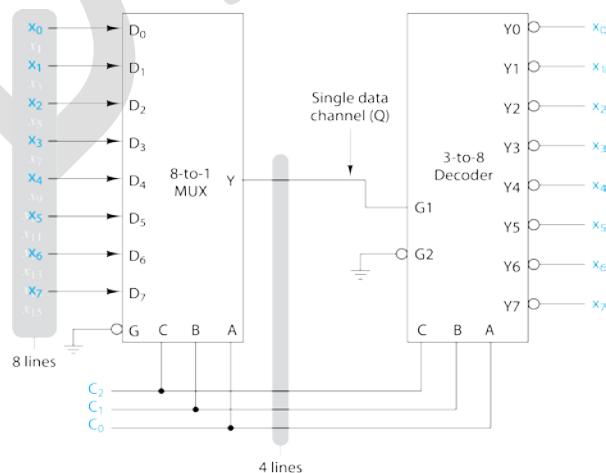


Figure 3.51. Multiplexer/Demultiplexer data transmission example.

3.3.4 Arithmetic Circuits

This section will be devoted to coverage of the basic logic circuits used to perform binary addition and subtraction. More complex operations, including multiplication and division, will be covered later. The section will conclude with a discussion of *comparators* which are logic circuits that are used to determine the relative magnitude of two binary numbers.

It was demonstrated in Chapter 1 that the addition of binary numbers in a two's complement number system is sufficient to perform both addition and subtraction operations. Half-adders (HAs), full-adders (FAs), and ripple-carry adders were introduced in chapter 2. These and other arithmetic circuits will be explored more in depth below. See Fig. 3.52 for a review of HAs and FAs.

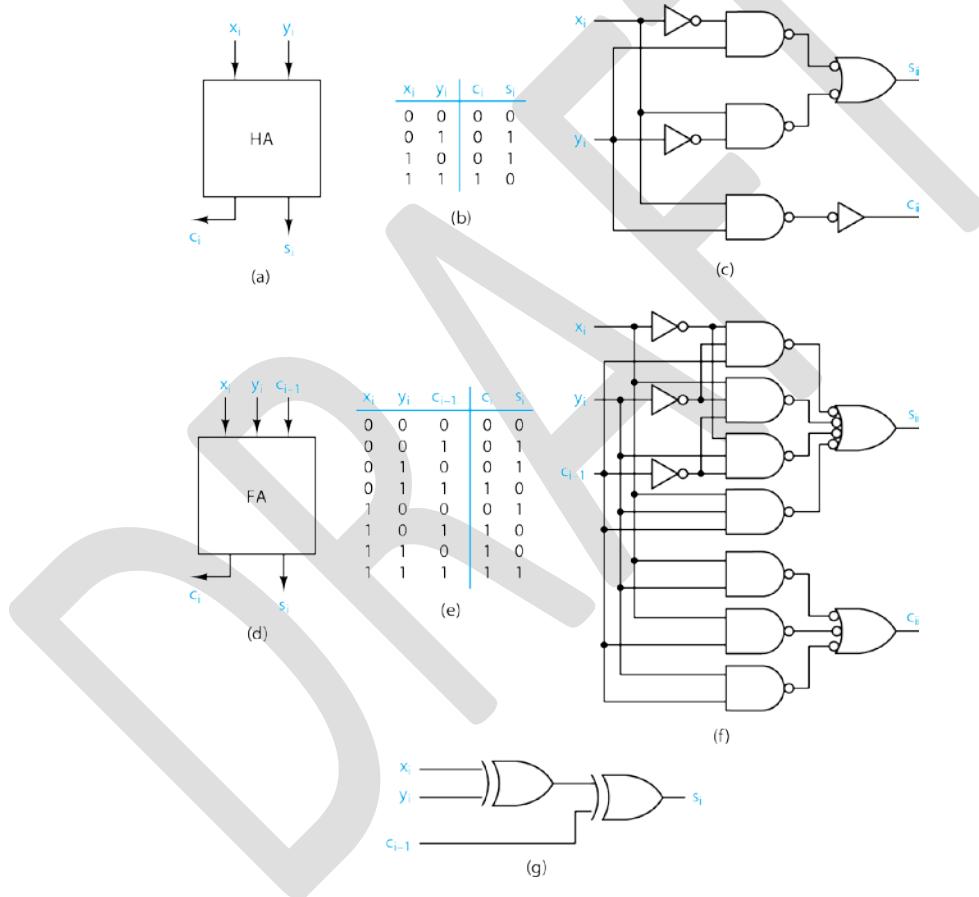


Figure 3.52. Binary half-adder and full-adder circuits. (a) Half-adder. (b) HA truth table. (c) NAND gate HA circuit. (d) Full-adder. (e) FA truth table. (f) NAND gate FA circuit. (g) XOR gate realization of FA s_i output.

Recall for half-adders,

$$s_i = x_i \oplus y_i \quad (3.21)$$

$$c_i = x_i y_i$$

The equations for full-adders are

$$\begin{aligned}s_i &= x_i \oplus y_i \oplus c_{i-1} \\ c_i &= x_i y_i + x_i c_{i-1} + y_i c_{i-1}\end{aligned}\quad (3.22)$$

A logic designer uses the half-adder or full-adder as modules to create larger circuits, using the logic symbols shown in Fig. 3.52(a) and (d). One HA module and $n-1$ FA modules can be used to implement an n -bit ripple-carry, or pseudoparallel, adder as shown in Fig. 3.53.

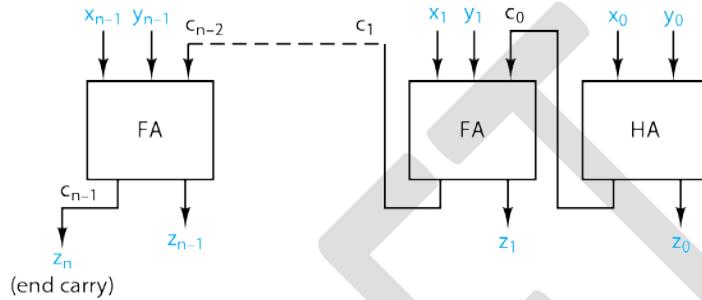


Figure 3.53. Ripple-Carry (pseudoparallel) adder.

This configuration uses one adder circuit for each bit position of the two input data words. The operation performed is

$$(x_{n-1}x_{n-2} \dots x_1x_0)_2 + (y_{n-1}y_{n-2} \dots y_1y_0)_2 = (z_nz_{n-1}z_{n-2} \dots z_1z_0)_2$$

The configuration is called *ripple-carry* or *pseudoparallel* because carries must propagate, or ripple, through the length of the adder unit. In general, the worst-case propagation path is through one half-adder and $n-1$ full-adders from the inputs x_0 and y_0 to the end sum and carry positions z_{n-1} and z_n . In each full or half adder, the carry is generated by two levels of logic. The carry is then combined with the input signals to generate the sum with two additional levels of logic. Hence we may say, in general, that the worst-case delay, t_{delay} , for a pseudoparallel adder constructed from these modules is

$$t_{\text{delay}} = (2n + 2)t_{\text{gate}} \quad (3.23)$$

where n is the word length of the adder unit. Here, t_{delay} is measured in terms of the total number of levels of logic through which the carries must propagate, each level delaying the signal by an amount of time t_{gate} .

The speed of ripple-carry adders can be improved by replacing the full-adders with high-speed four-bit adders. Figure 3.54 shows the logic diagram of one such four-bit adder. This module can be used to implement adders for word lengths that are multiples of four. For this module, C_0 is the carry-in and C_4 the carry-out. The internal carries (C_1, C_2, C_3) are

not propagated explicitly; that is, within the adder module the output bits ($\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4$) are formed in parallel. From the logic diagram we may write

$$\begin{aligned} P_i &= (\overline{B_i \cdot A_i})(A_i + B_i) \\ &= (\overline{A_i} + \overline{B_i})(A_i + B_i) \\ &= A_i \oplus B_i \end{aligned} \quad (3.25)$$

$$\begin{aligned} \Sigma_i &= P_i \oplus C_{i-1} \\ &= A_i \oplus B_i \oplus C_{i-1} \end{aligned} \quad (3.26)$$

and

$$\begin{aligned} C_1 &= \overline{(\overline{C}_0 \cdot \overline{A}_1 \cdot \overline{B}_1)} + \overline{(A_1 + B_1)} \\ &= (\overline{\overline{C}_0 \cdot \overline{A}_1 \cdot \overline{B}_1}) \cdot (A_1 + B_1) \\ &= (C_0 + (A_1 \cdot B_1)) \cdot (A_1 + B_1) \\ &= C_0 \cdot A_1 + C_0 \cdot B_1 + A_1 \cdot B_1 \end{aligned} \quad (3.27)$$

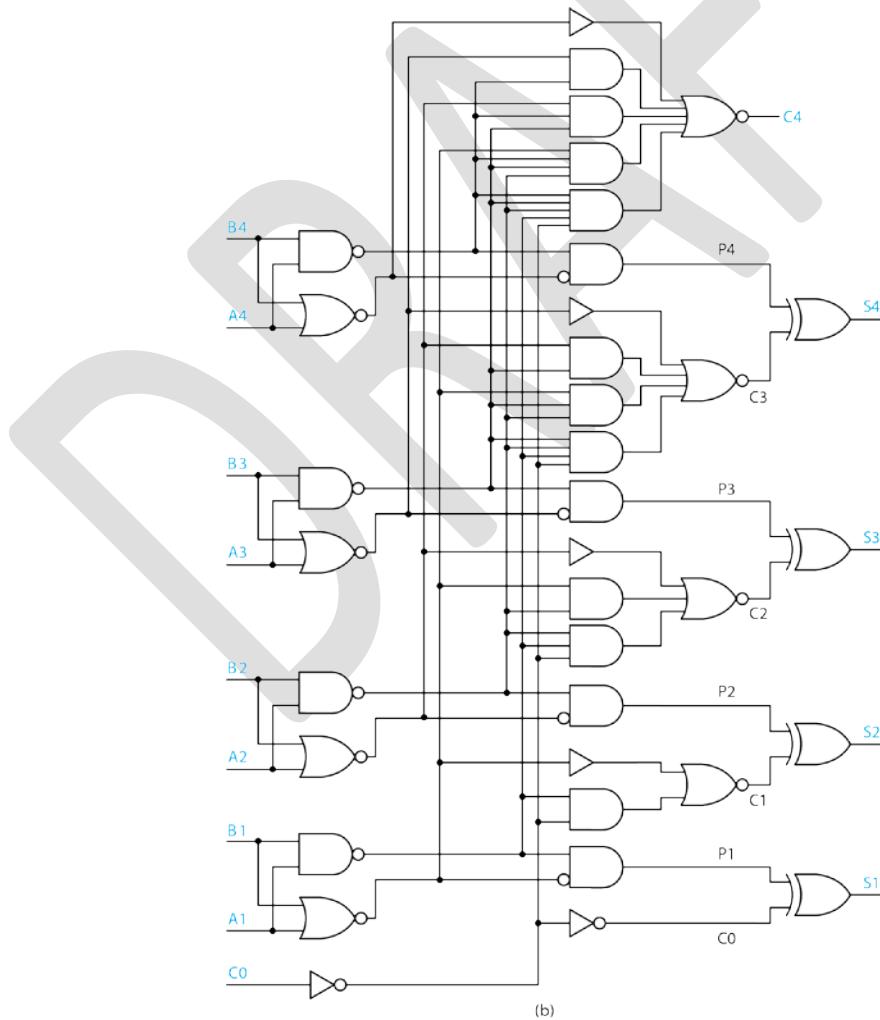


Figure 3.54. A High-Speed Four-bit Adder Circuit.

In a similar manner, we may find

$$C_i = C_{i-1} \cdot A_i + C_{i-1} \cdot B_i + A_i \cdot B_i \quad (3.28)$$

which agrees with Eq. 3.22. Within the module, the P_i signals are valid in two gate delays, the C_i signals in three gate delays, and the Σ_i outputs one XOR gate delay after the C_i terms. Since the module generates four sum terms, we can build an n -bit pseudoparallel adder of

$$m = \lceil n/4 \rceil \quad (3.29)$$

modules, where the notation $\lceil x \rceil$ means the smallest integer greater than x . Thus, the resulting unit will have an overall worst-case delay of three gate delays for the carry out of each module except the last, which will have four gate delays to its sum outputs:

$$t_{\text{delay}} = (3m + 1)t_{\text{gate}} \quad (3.30)$$

The pseudoparallel adders are simple to build and are reasonably fast. However, these designs are unsatisfactory in some high-speed applications because of the increased delay as n becomes large.

In the design of arithmetic circuits, we often strive to improve circuit performance by increasing the speed with which different operations, such as binary addition, can be done. The speed of a circuit can be improved by selecting a technology in which the individual logic gates have shorter propagation delays or by designing the circuit to minimize the number of gate delays required to complete the operation. In most cases, the number of gates needed to realize a design must be increased to reduce the total propagation delay, resulting in a trade-off of cost for performance. In the following paragraphs we examine a number of methods used to reduce the number of propagation delays in binary adder circuits. In each case we will evaluate the number of gates needed for the design and the total number of gate delays needed to produce the sum of two numbers.

The fastest adder design would be fully parallel. That is, all the inputs would be applied simultaneously and propagate through two levels of logic to obtain the result. However, this approach would require an enormous amount of logic circuitry and is not practical to employ. Consider the generation of the first three carry bits of a fully parallel adder.

$$c_0 = x_0 y_0 \quad (3.31)$$

$$\begin{aligned} c_1 &= x_1 y_1 \bar{c}_0 + x_1 y_1 c_0 + x_1 \bar{y}_1 c_0 + \bar{x}_1 y_1 c_0 \\ &= x_1 y_1 + (x_1 \oplus y_1) c_0 \\ &= x_1 y_1 + (x_1 \oplus y_1)(x_0 y_0) \end{aligned} \quad (3.32)$$

$$\begin{aligned} c_2 &= x_2 y_2 + (x_2 \oplus y_2) c_1 \\ &= x_2 y_2 + (x_2 \oplus y_2)[x_1 y_1 + (x_1 \oplus y_1)(x_0 y_0)] \\ &= x_2 y_2 + (x_2 \oplus y_2)(x_1 y_1) + (x_2 \oplus y_2)(x_1 \oplus y_1)(x_0 y_0) \end{aligned} \quad (3.33)$$

These equations can be further simplified into sum of products form, allowing each to be realized with two levels of logic, independent of the word width of the adder. However, the gate count increases considerably with each higher bit position as does gate fan in.

Several compromises are employed between the pseudoparallel and fully parallel alternatives. *Carry look-ahead adders* divide the full adders into groups and employ carry bypass logic to speed up the carry propagation. This technique is reasonable to employ when numerical data are to be added at high-speed fixed intervals. Examining Eqs. 3.31 through 3.33, let us define the two terms

$$g_i = x_i y_i \quad (3.34)$$

and

$$p_i = x_i \oplus y_i \quad (3.35)$$

Using these terms, Eqs. 3.31 through 3.33 can be rewritten as follows

$$c_0 = g_0 \quad (3.36)$$

$$\begin{aligned} c_1 &= g_1 + p_1 c_0 \\ &= g_1 + p_1 g_0 \end{aligned} \quad (3.37)$$

$$\begin{aligned} c_2 &= g_2 + p_2 c_1 \\ &= g_2 + p_2 g_1 + p_2 p_1 g_0 \end{aligned} \quad (3.38)$$

Within each bit position, if $g_i = 1$ a carry is *generated*, independent of the carry input c_{i-1} . Likewise, a carry input of 1 will be *propagated* from the input to the output of stage i if $p_i = 1$. Hence, the terms g_i and p_i are referred to as the *carry generate* and *carry propagate* terms, respectively, for stage i . Note that these terms can be derived in parallel for all bit positions in a single gate delay and that the carry bits can be computed in parallel from the generate and propagate terms in two additional gate delays. A circuit realizing Eqs. 3.34 and 3.35 is given in Fig. 3.55a, and a circuit realizing Eqs. 3.36 through 3.38 is given in Fig. 3.55b.

The sum term realized by a full adder, as defined by Eq. 3.22, can be rewritten in terms of the carry and propagate terms as follows

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus c_{i-1} \\ &= p_i \oplus c_{i-1} \end{aligned} \quad (3.39)$$

Therefore, once the carry terms are available, the sum terms can be computed in one additional gate delay, resulting in a total adder delay of

$$t_{cla} = 4t_{gate}$$

independent of the word width of the adder. Circuits for a single adder module and a complete 3-bit carry look-ahead adder are presented in Figs. 3.55a and c. It should be noted that the delays of exclusive-OR gates may be almost twice those of simple NAND/NOR gates. Therefore, a more realistic delay estimate is

$$4t_{gate} \leq t_{cla} \leq 6t_{gate}$$

where t_{gate} is the delay through a typical NAND gate.

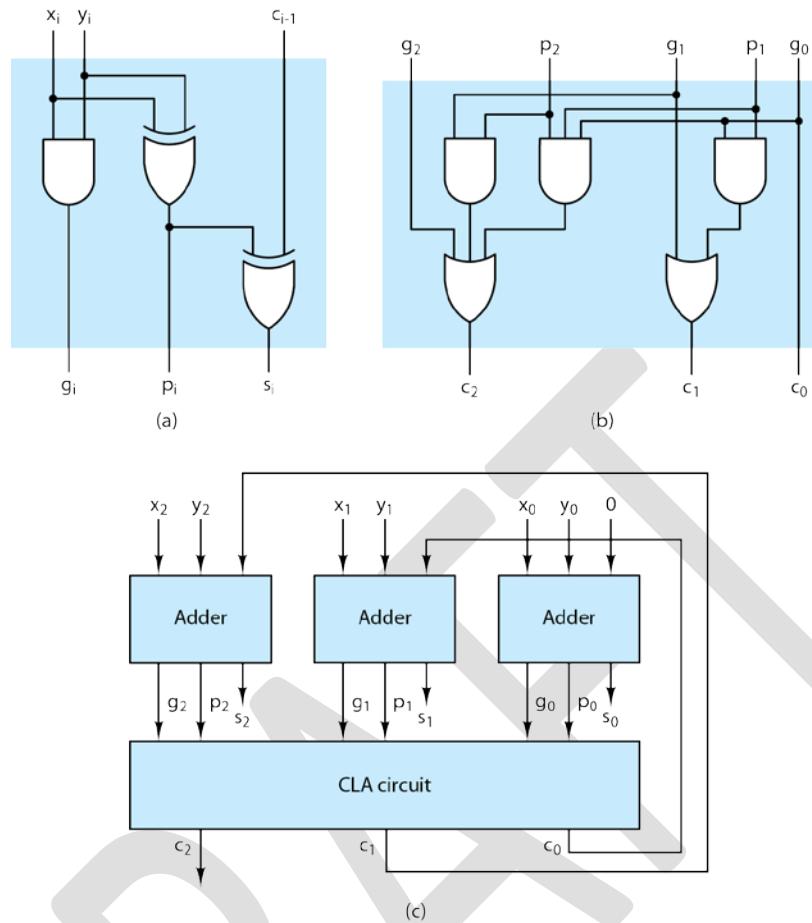


Figure 3.55. Carry look-ahead adder design. (a) Adder module producing g_i and p_i . (b) Carry look-ahead (CLA) circuit. (c) Complete carry look-ahead adder.

Other methods for implementing fast adders include *carry-completion-detection* and *carry-save*. However, these methods will not be covered in this book.

Circuits to perform subtraction of binary numbers can be developed in the same manner as for binary addition. Half- and full-subtracter modules can be designed using the procedure illustrated for half and full adders previously. Then these modules can be cascaded to form n -bit pseudoparallel subtracters. This procedure is left as an exercise for the reader.

When both addition and subtraction must be performed, such as in the arithmetic unit of a digital computer, the overall design can be simplified by the use of two's complement arithmetic. Recall that subtraction in the two's complement number system is performed as follows:

$$\begin{aligned}
 (R)_2 &= (P)_2 - (Q)_2 \\
 &= (P)_2 + (-Q)_2 \\
 &= (P)_2 + [Q]_2 \\
 &= (P)_2 + (\bar{Q})_2 + 1
 \end{aligned}$$

where $[Q]_2 = (\bar{Q})_2 + 1$ from Algorithm 1.4. Figure 3.56 illustrates the use of a binary adder to perform both addition and subtraction. The adder module realizes the function

$$(\Sigma)_2 = (A)_2 + (B)_2 + C0 \quad (3.40)$$

When the select line is 0, the multiplexer routes its inputs A to its outputs Y , so $(Q)_2$ is connected to the B inputs of the adder module. Since the select line is also connected to $C0$ of the adder, $C0 = 0$. Under these conditions, the adder performs the following operation:

$$(\Sigma)_2 = (A)_2 + (B)_2 + C0$$

where

$$(A)_2 = (P)_2$$

$$(B)_2 = (Q)_2$$

$$C0 = 0$$

So

$$\begin{aligned} (R)_2 &= (\Sigma)_2 \\ &= (P)_2 + (Q)_2 + 0 \\ &= (P)_2 + (Q)_2 \end{aligned}$$

This is the addition operation.

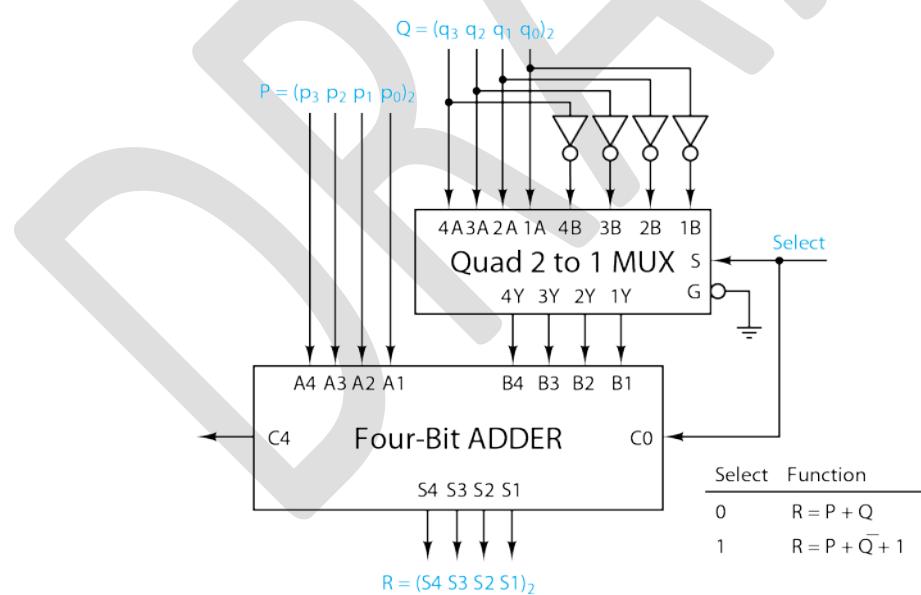


Figure 3.56. Two's complement adder/subtractor.

Now consider the case when $S = 1$. For $S = 1$, the multiplexer module routes its B inputs to its Y outputs, so the complement of $(Q)_2$ is connected to the B inputs of the adder module. The select line S also drives the $C0$ input of the adder module, so

$$\begin{aligned}(R)_2 &= (\Sigma)_2 \\ &= (A)_2 + (B)_2 + C0\end{aligned}$$

where

$$(A)_2 = (P)_2$$

$$(B)_2 = (\bar{Q})_2$$

$$C0 = 1$$

So

$$\begin{aligned}R &= (P)_2 + (\bar{Q})_2 + 1 \\ &= (P)_2 + [Q]_2 \\ &= (P)_2 - (Q)_2\end{aligned}$$

This is the subtraction operation.

Therefore, both addition and subtraction functions can be implemented with a single adder module and a multiplexer, as illustrated in Fig. 3.56.

As discussed in Chapter 1, the range of values that can be represented by n -bit numbers in the two's complement number system is

$$-2^{n-1} \leq N \leq 2^{n-1} - 1$$

Any arithmetic operation that produces a result outside this range is said to cause an *overflow* condition. In these cases the resulting n -bit number is not a valid representation of the result. Therefore, all overflow conditions must be detected so that invalid results are not used inadvertently.

In Chapter 1 it was shown that, in two's complement arithmetic, overflow conditions are produced by adding two positive values whose sum is greater than $2^{n-1} - 1$ or two negative values whose sum is less than -2^{n-1} . In both cases the result will have an incorrect sign bit. Therefore, overflows can be detected by observing the sign bits of the operands and the result.

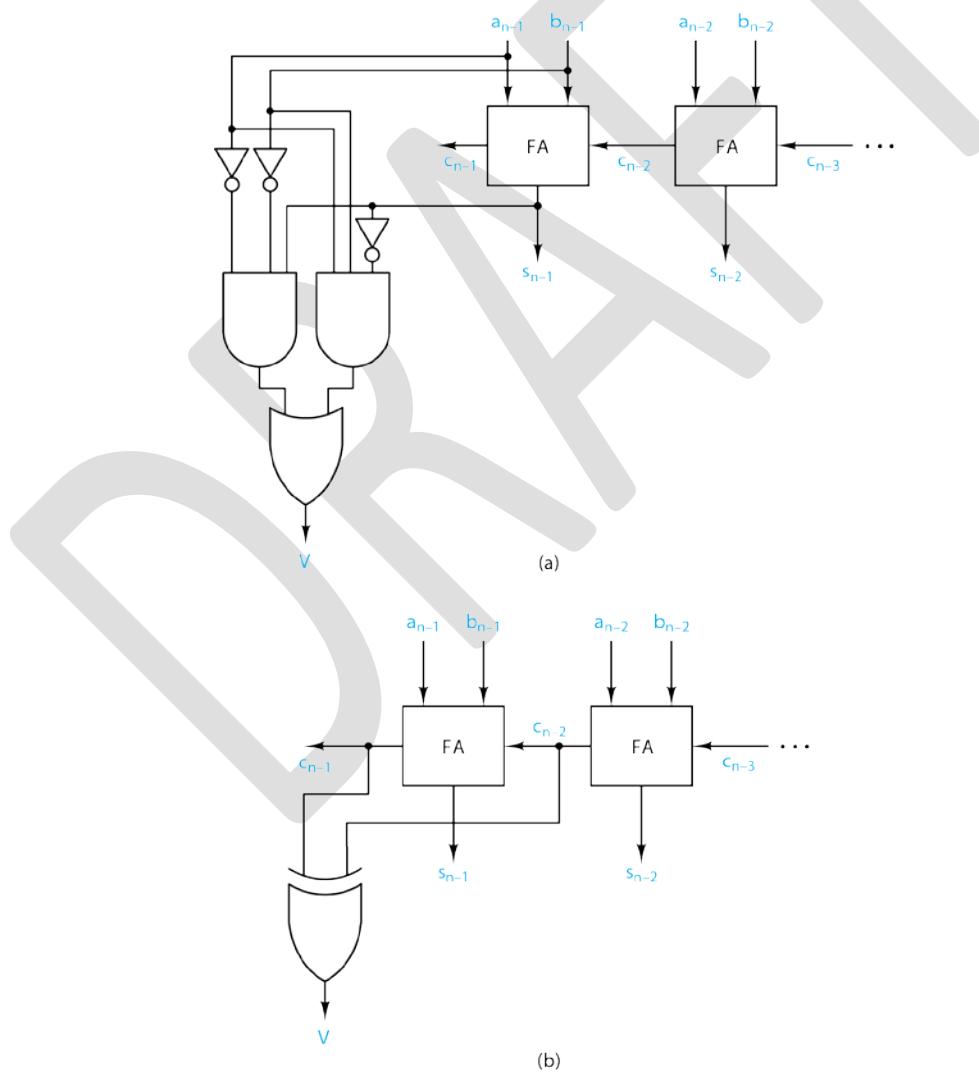
Table 2 shows the eight possible situations that can occur in the most significant stage of an n -bit adder. Bits a_{n-1} and b_{n-1} represent the sign bits of the numbers being added and are therefore inputs to the stage, along with carry bit c_{n-2} . The outputs of the stage are the carry-out and sum bits c_{n-1} and s_{n-1} , respectively. As seen in the table, an overflow condition occurs in two cases: the addition of two positive values producing sign bit $s_{n-1} = 1$, indicating a negative result, and the addition of two negative values producing sign bit $s_{n-1} = 0$, indicating a positive result. Therefore, a logic expression for the overflow condition V is

$$V = \bar{a}_{n-1} \bar{b}_{n-1} s_{n-1} + a_{n-1} b_{n-1} \bar{s}_{n-1} \quad (3.41)$$

An AND/OR logic circuit realization of Eq. 3.41 is presented in Fig. 3.57(a).

Table 2. Most significant stage of an n -bit adder.

Adder Inputs			Adder Outputs		Overflow
a_{n-1}	b_{n-1}	c_{n-2}	c_{n-1}	s_{n-1}	V
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

**Figure 3.57.** Two's complement overflow detection. (a) Using sign bits. (b) Using carry bits.

Overflows can also be detected by observing the carry-in and carry-out bits, c_{n-2} and c_{n-1} , respectively, of the most significant full-adder stage. Looking at the two rows of Table 2 in which overflows occur, it can be observed that these are the only two cases in which $c_{n-2} \neq c_{n-1}$. As seen in the first two rows of Table 2, the addition of two positive numbers always results in a carry output $c_{n-1} = 0$. Therefore, the carry input to the most significant bit, c_{n-2} , must also be 0 to produce a correct positive sum. A carry of 1 into this bit results in $s_{n-1} = 1$, which is incorrect. A similar situation exists in row 7 of Table 2. The sum of two negative numbers produces $c_{n-1} = 1$. The absence of a carry into this bit results in an incorrect sign bit $s_{n-1} = 0$. Thus, the overflow condition is signaled by $c_{n-2} \neq c_{n-1}$ or $c_{n-2} \oplus c_{n-1} = 1$. Therefore,

$$V = c_{n-2} \oplus c_{n-1} \quad (3.42)$$

The corresponding logic circuit requires a single XOR gate, as shown in Fig. 3.57(b). This circuit is simpler than that of Fig. 3.57(a), but requires access to the carry bit c_{n-2} between the last two adder stages. This bit is not always accessible, as is the case when using adder modules such as the one described in Fig. 3.54 and a number of other parallel-adder designs. In such cases, overflows must be detected by examining sign bits, using Eq. 3.41.

A *comparator* is an arithmetic circuit that determines the relative magnitude of two binary numbers and finds applications in numerous digital systems. In general, a comparator can perform a magnitude comparison of two numbers A and B in either straight binary or BCD codes. Three fully decoded decisions about the two words are made and are available at the outputs; that is, $A > B$, $A < B$, and $A = B$. See Fig. 3.58(a). If

$$\begin{aligned} A &= (A_{n-1}A_{n-2}\dots A_0)_2 \\ B &= (B_{n-1}B_{n-2}\dots B_0)_2 \end{aligned} \quad (3.43)$$

then the comparator will generate three output signals as follows

$$\begin{aligned} f_1 &= 1, \text{ if } A < B \\ f_2 &= 1, \text{ if } A = B \\ f_3 &= 1, \text{ if } A > B \end{aligned} \quad (3.44)$$

In other words, a comparator is a $2n$ -input, 3-output combinational logic circuit that determines the relative magnitude of two n -bit numbers.

Design a comparator that will compare the two words $A = (A_1A_0)_2$ and $B = (B_1B_0)_2$ in binary code.

The truth table for the output signals defined by Eqs. 3.44 is shown in Fig. 3.58(b), and the K-map for the output function $(A_1A_0)_2 < (B_1B_0)_2$ is shown in Fig. 3.58(c). Note that a 1

appears in every block where the binary value of $(A_1A_0)_2$ is less than that of $(B_1B_0)_2$. The K-maps for the outputs $(A_1A_0)_2 = (B_1B_0)_2$ and $(A_1A_0)_2 > (B_1B_0)_2$ are also shown in Fig. 3.58(c). The output functions for these maps are

$$f_1 = \overline{A}_1B_1 + \overline{A}_1\overline{A}_0B_0 + \overline{A}_0B_1B_0 \text{ for } (A_1A_0)_2 < (B_1B_0)_2$$

$$f_2 = \overline{A}_1\overline{A}_0\overline{B}_1\overline{B}_0 + \overline{A}_1A_0\overline{B}_1B_0 + A_1\overline{A}_0B_1\overline{B}_0 + A_1A_0B_1B_0 \text{ for } (A_1A_0)_2 = (B_1B_0)_2$$

$$f_3 = A_1\overline{B}_1 + A_0\overline{B}_1\overline{B}_0 + A_1A_0\overline{B}_0 \text{ for } (A_1A_0)_2 > (B_1B_0)_2$$

A realization of this circuit is shown in Fig. 3.59.

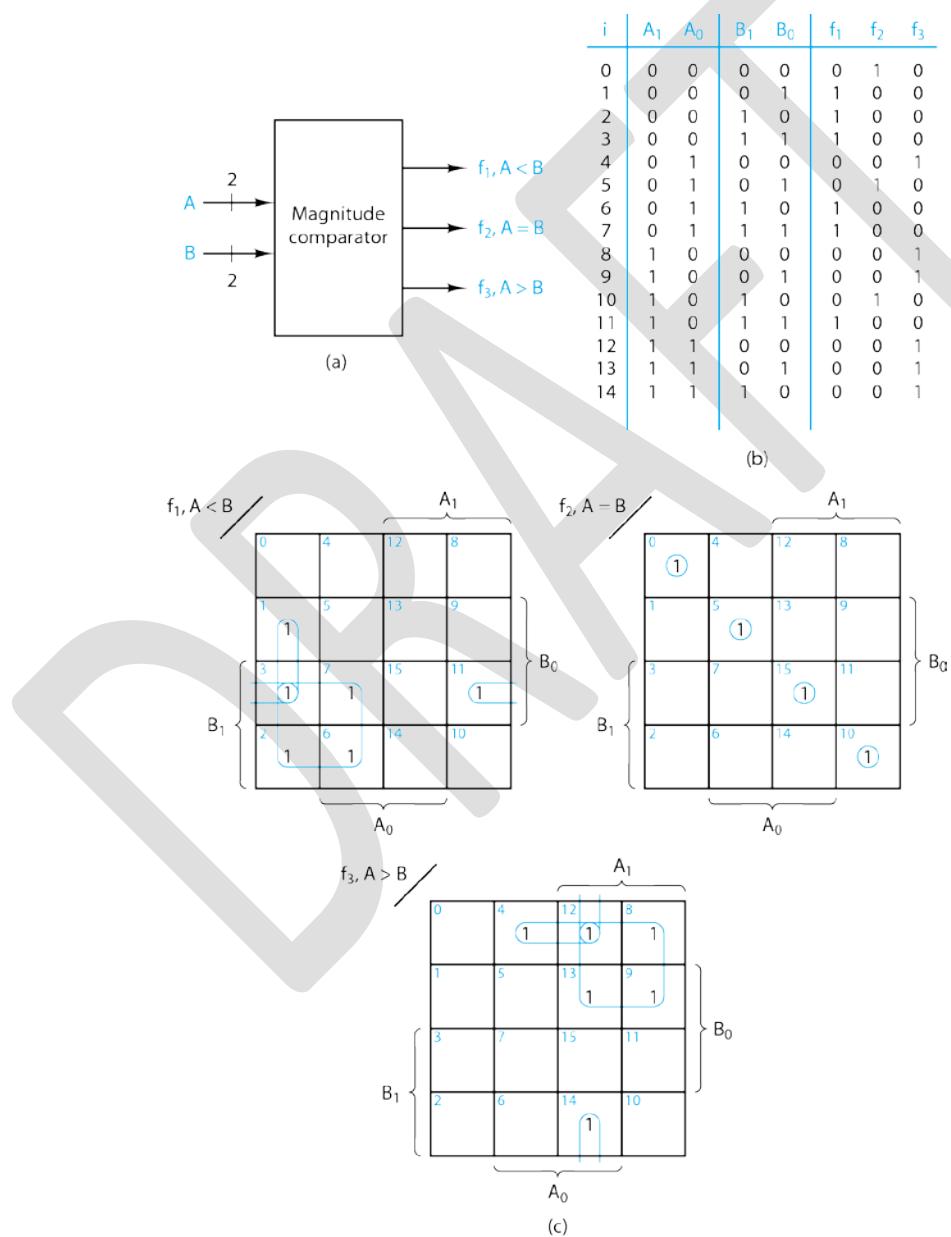


Figure 3.58. Two-bit comparator design. (a) Functional diagram. (b) Truth table. (c) K-maps.

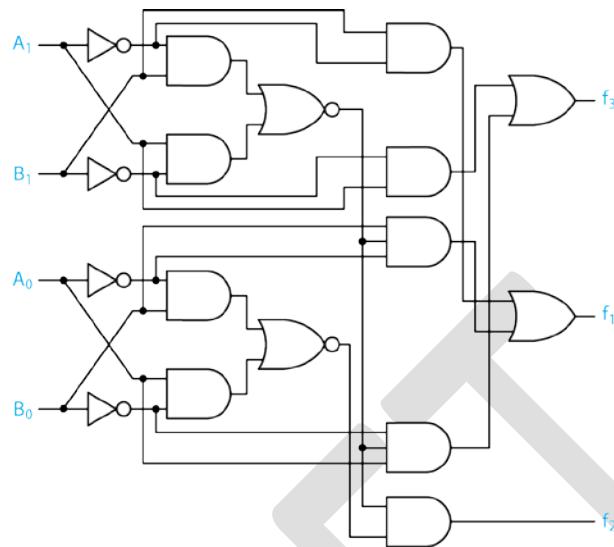


Figure 3.59. Logic realization of a 2-bit comparator.

A 4-bit magnitude comparator similar to the one derived in Example 4.10 is illustrated in Fig. 3.60. Note that this circuit can be put in block diagram form as shown in Fig. 3.61(a). The circuits data inputs are

$$\begin{aligned} A &= (A_3, A_2, A_1, A_0)_2 \\ B &= (B_3, B_2, B_1, B_0)_2 \end{aligned} \quad (3.45)$$

Cascade inputs, C1, C2, and C3, are also provided that allow multiple modules to be interconnected to compare larger numbers.

$$\begin{aligned} C1 &\rightarrow A < B \\ C2 &\rightarrow A = B \\ C3 &\rightarrow A > B \end{aligned} \quad (3.46)$$

In this form it is clear that words of length greater than 4 bits may be compared by cascading these units. For example, the outputs of a stage handling the 4 least significant bits can be connected to the input terminals $A < B$, $A = B$, and $A > B$ of the next stage, together with the 4 most significant bits to generate an 8-bit comparator. In a similar manner, these units can be fully expanded to any number of bits. This is illustrated in the following example.

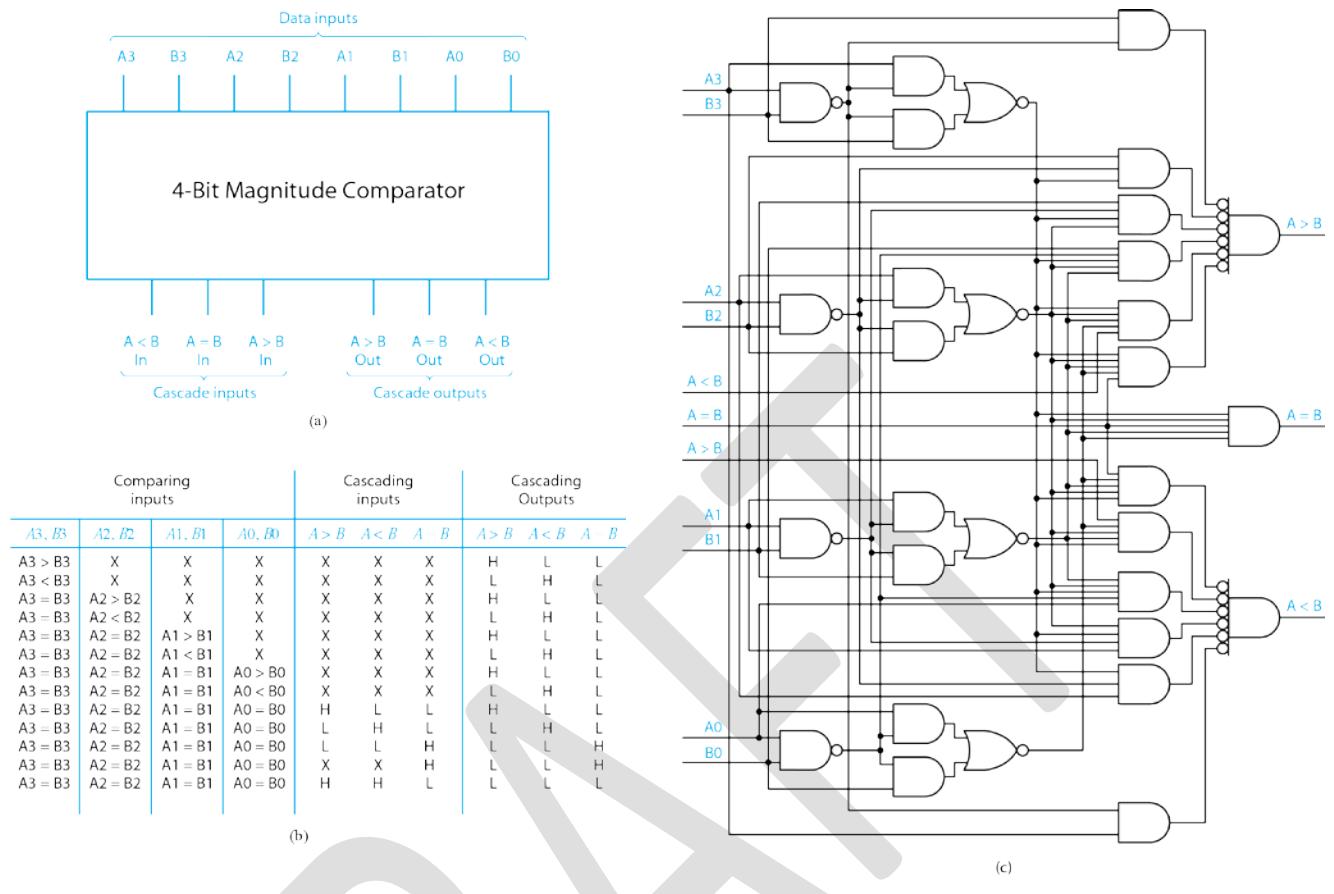


Figure 3.60. A 4-bit magnitude comparator. (a) Logic symbol. (b) Function table. (c) Logic diagram.

Example -- Use the 4-bit magnitude comparator to construct a 16-bit magnitude comparator.

Four 4-bit modules may be cascaded as shown in Fig. 3.61(b) to produce a 16-bit comparator. Note that the initial conditions on the first state must be

$$(C_1, C_2, C_3) = (0, 1, 0) \quad (3.47)$$

indicating $A = B$. The circuit output is f_1, f_2, f_3 from the most significant stage. The circuit compares the numbers

$$A = (A_{15}, A_{14}, \dots, A_0)_2 \quad (3.48)$$

$$B = (B_{15}, B_{14}, \dots, B_0)_2$$

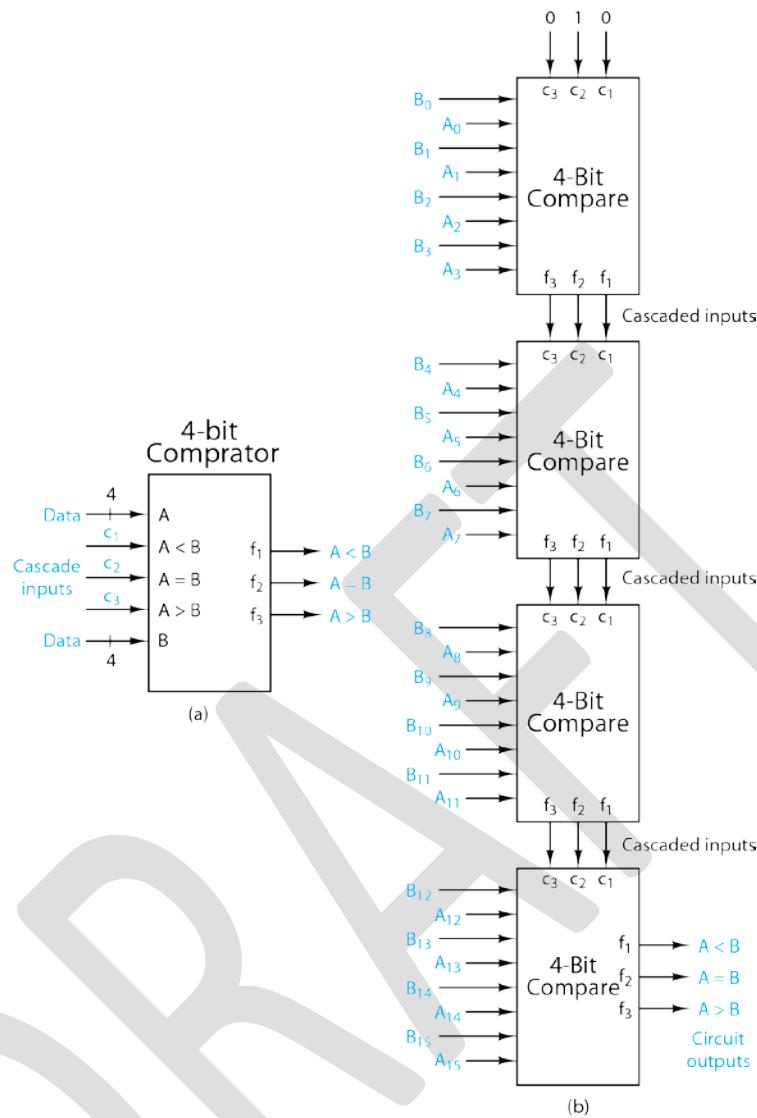


Figure 3.61. Cascading the 4-bit magnitude comparator. (a) I/O diagram. (b) 16-bit comparator.

3.4 Summative Design Examples

3.4.1 Design Flow

Design is the process of turning ideas in to reality and is a cornerstone of engineering. The design process can be systematized, to a certain extent, as a series of steps often referred to as the design flow. The design process as presented here begins with an idea, or need, for a new product and ends with the development of a prototype that meets product requirements. Figure 3.62 shows a flow chart of this process. Transforming a prototype in to a commercial product is a follow-on process and beyond the scope of this book.

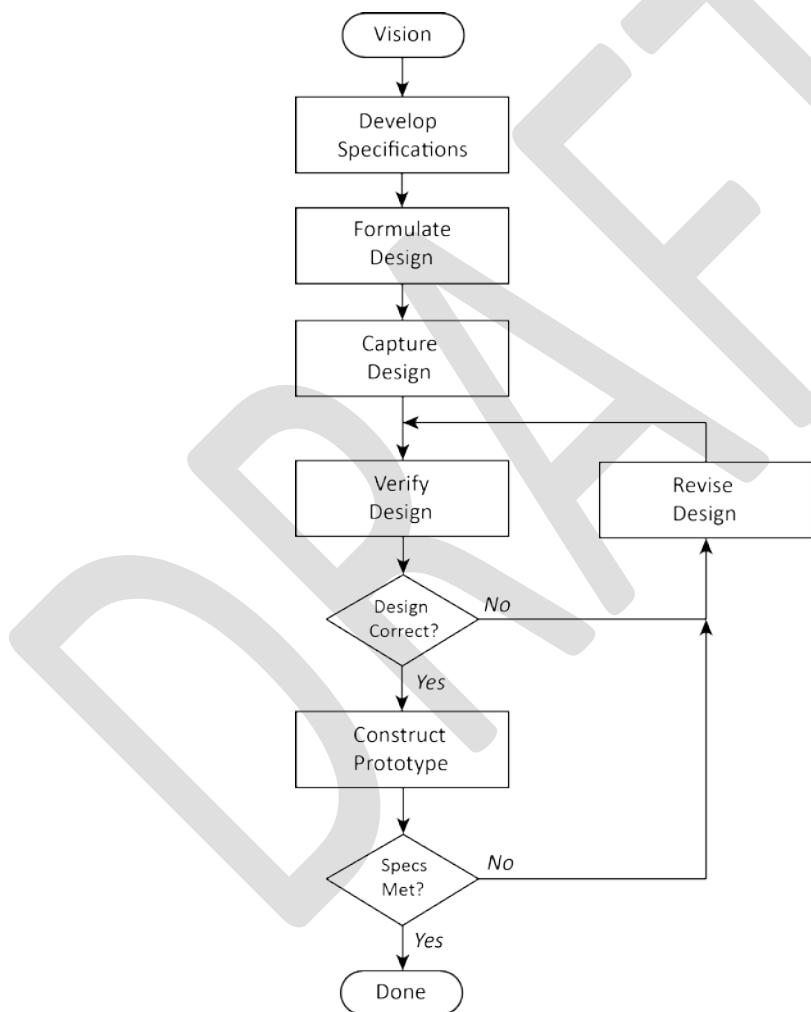


Figure 3.62 Design Flow

The following examples illustrate design formulation and design capture for four combinational logic circuits of varying degrees of complexity. Verification, revision, and prototype construction are beyond the scope of this section.

3.4.2 Bank Security-Lock Controller

This example illustrates the design of a simple combinational logic circuit that serves as the controller for a lock that secures the vault area of *Unsecure Bank and Trust*. Bank officials may open the lock according to the following protocol.

- During business hours – the bank president (P) or both of the vice presidents (VP1, VP2).
- Off hours – the president and either vice president.

The task is to design a combinational logic circuit that will UNLOCK the lock when the appropriate combination of officials enter ID codes. Assume that logic 1 indicates that the proper code has been entered by the respective officer and that logic 0 means that the proper code was not entered. The variable OPEN=1 indicates that the bank is open for business. A block diagram of the controller is shown in Fig. 3.63(a). Design proceeds as follows.

1. Construct a truth table that defines the UNLOCK function of variables P, VP1, VP2, and OPEN that meets the specifications given previously. See Fig. 3.63(b) for the completed truth table.
2. Derive logic equations for UNLOCK using concepts learned in Chapter 2. Minterm list, canonical sum of products (CSP), minimum sum of products (MSP), and factored forms of the equations are shown below.

$$\begin{aligned}
 \text{UNLOCK} &= \sum m(5, 6, 7, 11, 12, 13, 14, 15) && [\text{Minterm list}] \\
 &= O' \cdot P \cdot VP1' \cdot VP2 + O' \cdot P \cdot VP1 \cdot VP2' + O' \cdot P \cdot VP1 \cdot VP2 \\
 &\quad + O \cdot P' \cdot VP1 \cdot VP2 + O \cdot P \cdot VP1' \cdot VP2' + O \cdot P \cdot VP1' \cdot VP2 \\
 &\quad + O \cdot P \cdot VP1 \cdot VP2' + O \cdot P \cdot VP1 \cdot VP2 && [\text{CSP}] \\
 &= P \cdot VP1 + P \cdot VP2 + O \cdot P + O \cdot VP1 \cdot VP2 && [\text{MSP}] \\
 &= P(VP1 + VP2) + O(P + VP1 \cdot VP2) && [\text{Factored MSP}]
 \end{aligned}$$

3. Capture the design of UNLOCK in schematic, HDL, and/or netlist forms as described in the following paragraphs.

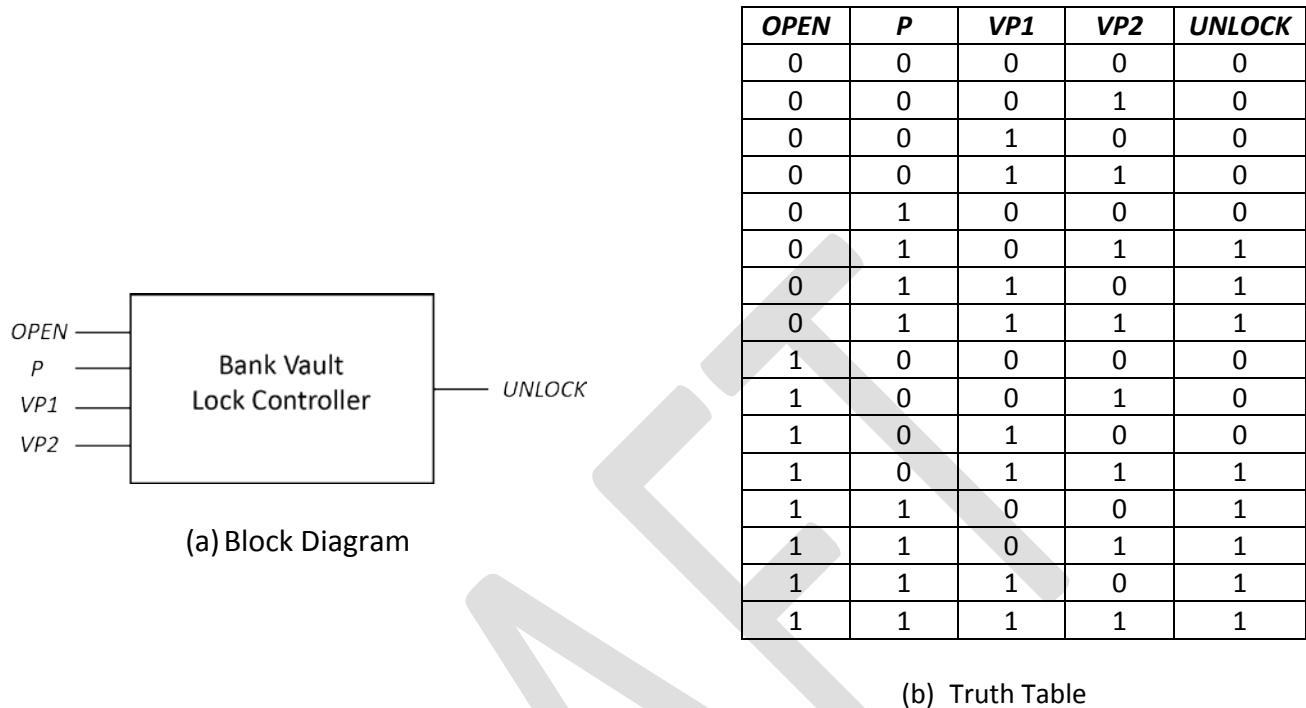


Figure 3.63 Bank Vault Controller Example

3.4.2.1 Schematic – The MSP form of UNLOCK can be used to generate AND-OR and NAND-NAND realizations as shown in Figs 3.64(a) and (b), respectively. The factored form yields multi-level realizations as shown in Figs. 3.64(c) and (d). Given the four realizations, which one should be chosen to implement the controller? Both the AND-OR and NAND-NAND circuits require five gates and have logic levels of two. However, the AND-OR circuit uses a three-input AND gate which is usually not available in IC form. This would require the use of two, two-input AND as a substitute which would result in a six-gate, three-level circuit. On the other hand, three-input NAND gates are readily available meaning that the NAND-NAND circuit can be realized as shown. The factored-form realizations benefit from only requiring two-input gates but have four logic-levels. So, in conclusion, the NAND-NAND circuit is preferred for implementation with standard gates.

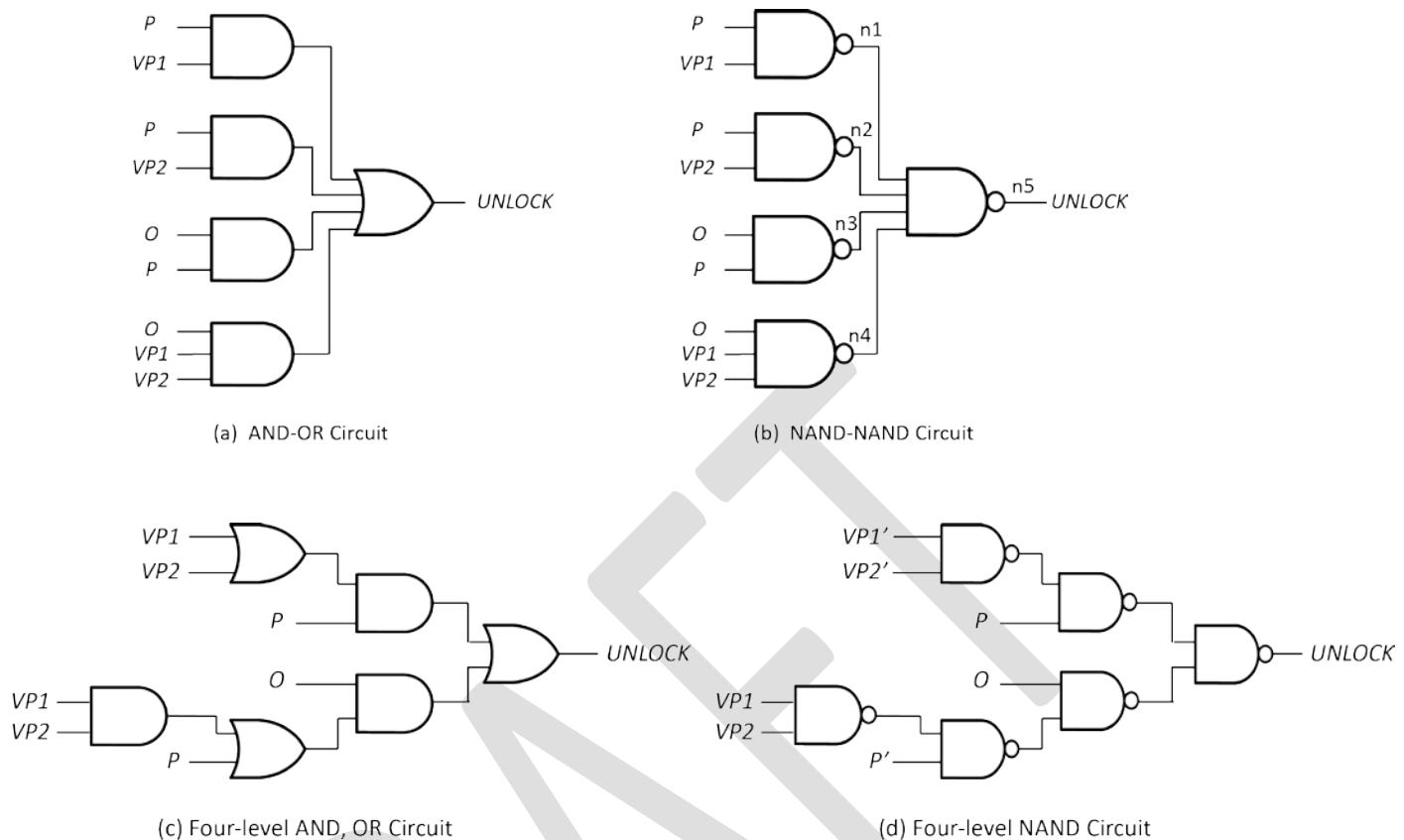


Figure 3.64 Bank Vault Controller Gate-Level Realizations

3.4.2.2 Verilog code – The controller design can also be captured using a hardware description language such as Verilog. Figure 3.65(a) shows a structural description of the controller that corresponds to the previous NAND-NAND realization. Figure 3.65(b) shows a Verilog model that describes the controller functionally in correspondence to the MSP form of the UNLOCK function.

```
//Bank-lock controller – structural model
module BankLockController(Unlock,P,VP1,VP2,O);
input P,VP1,VP2,O;
output Unlock;
wire n1,n2,n3,n4;
  nand (n1,P,VP1);
  nand (n2,P,VP2);
  nand (n3,O,P);
  nand (n4,O,VP1,VP2);
  nand (Unlock,n1,n2,n3,n4);
endmodule
```

```
//Bank-controller – dataflow model
module BankLockController (Unlock,P,VP1,VP2,O);
input P,VP1,VP2,O;
output Unlock;
  assign Unlock = (P&VP1)|(P&VP2)|(O&P)|(O&VP1&VP2);
endmodule
```

Figure 3.65 – Verilog models for the Bank-Lock Controller

3.4.2.3 VHDL code

```
-- Bank lock controller – structural model
library ieee; use ieee.std_logic_1164.all;
use work.gates.all;
entity BankLockController is
    port (Unlock: out std_logic;
          P, VP1, VP2, O: in std_logic);
end;
architecture structure of BankLockController is
    signal n1,n2,n3,n4: std_logic;
begin
    G1: nand02 port map (P,VP1,n1);
    G2: nand02 port map (P,VP2,n2);
    G3: nand02 port map (O, P,n3);
    G4: nand03 port map (O,VP1,VP2,n4);
    G5: nand04 port map (n1,n2,n3,n4,Unlock);
end;
```

```
-- Bank lock controller - behavioral model
library ieee; use ieee.std_logic_1164.all;
entity BankLockController is
    port (Unlock: out std_logic;
          P, VP1, VP2, O: in std_logic);
end;
architecture behavior of BankLockController is
begin
    Unlock <= (P and VP1) or (P and VP2) or (O and
VP1 and VP2);
end;
```

Figure 3.66 – VHDL Models for the Bank-Lock Controller.

3.4.2.4 Netlists – Following design capture, designs are usually converted to netlist form for input to CAD tools such as simulators, placers, and/or routers that produce the physical design of the circuit for realization on printed circuit boards (PCBs) and/or programmable logic devices (PLDs). A *net* is defined as a wire or signal-line whose logic value is common throughout a logic circuit. A *netlist* is a list of all the nets of a circuit and is sufficient to define the structure of the circuit. The following is the general form of each net in a netlist. Note that *Ii* and *Oj* are used to declare input *i* and output *j*, respectively.

element_name element_type output input1 input2 ... inputN

The netlist for the NAND-NAND circuit of fig. 3.64(b) is the following.

/1	Input	P			
/2	Input	VP1			
/3	Input	VP2			
/4	Input	O			
N1	NAND2	n1	P	VP1	
N2	NAND2	n2	P	VP2	
N3	NAND2	n3	O	P	
N4	NAND3	n4	O	VP1	VP2
N5	NAND4	Unlock	n1	n2	n3
O1	Output	Unlock			

Designs captured using HDLs are converted to a netlist during compiling according to conventions built in to the compiler or as specified by the designer. A possible netlist for the functional Verilog model is the following.

<i>I</i> 1	Input	<i>P</i>
<i>I</i> 2	Input	<i>VP</i> 1
<i>I</i> 3	Input	<i>VP</i> 2
<i>I</i> 4	Input	<i>O</i>
<i>A</i> 1	AND2	<i>n</i> 1 P <i>VP</i> 1
<i>A</i> 2	AND2	<i>n</i> 2 P <i>VP</i> 2
<i>A</i> 3	AND2	<i>n</i> 3 O P
<i>A</i> 4	AND3	<i>n</i> 4 O <i>VP</i> 1 <i>VP</i> 2
<i>R</i> 5	OR4	Unlock <i>n</i> 1 <i>n</i> 2 <i>n</i> 3 <i>n</i> 4
<i>O</i> 1	Output	Unlock

3.4.3 Seven-Segment Display Decoder

Seven-segment LED displays comprise seven light-emitting diodes (LEDs) arranged as shown in Fig. 3.67. Selected combinations of the LEDs are illuminated to create numeric digits and other symbols. For example, the segments normally activated to display the decimal digits are shown in Fig. 3.68.

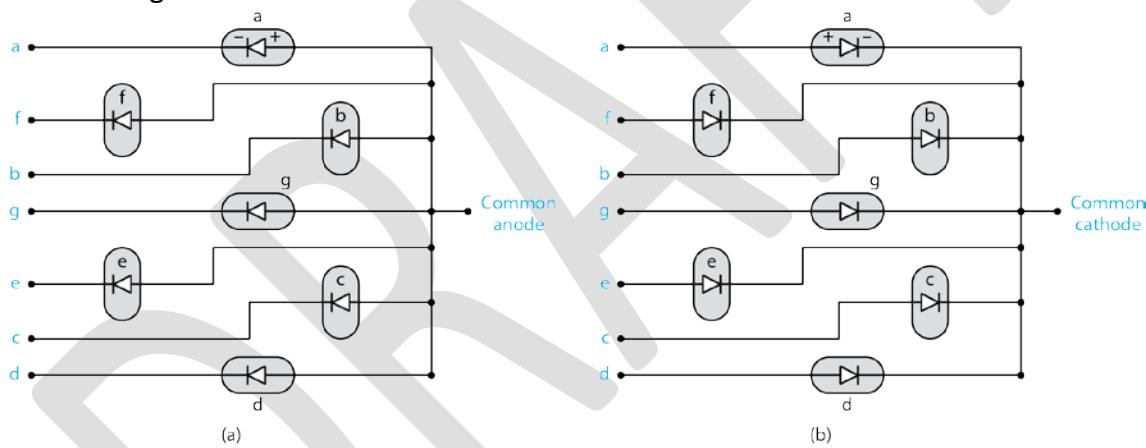


Figure 3.67 7-segment display elements. (a) Common anode. (b) Common cathode.

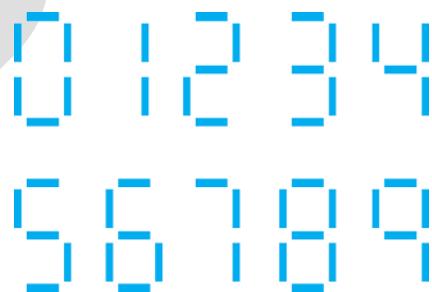


Figure 3.68 Decimal digits displayed on 7-segment display elements.

An LED emits light when the voltage at its anode is made sufficiently higher than the

voltage at its cathode. This is illustrated on diode a in Figs. 3.67 (a) and (b), in which the anode is marked with a ``+'' and the cathode with a ``-''. In digital circuits, these voltages are created by applying a high voltage to the anode and a low voltage to the cathode.

To minimize the number of control signals, the anodes of the LEDs are usually connected at a common point, called the *common anode*, as shown in Fig. 3.67 (a), or else the cathodes are connected at a *common cathode*, as in Fig. 3.66(b). In the common-anode configuration, the anodes are usually connected to a high voltage and the cathodes are controlled individually. Consequently, a logic 0 applied to a cathode illuminates that LED, whereas a logic 1 voltage disables the LED. The opposite conditions are used in the common-cathode configuration. Therefore, the inputs to the common-anode configuration may be considered active low, since low signals activate the LEDs, and the inputs to the common cathode configuration are active high.

To display a number encoded in BCD format, we can design a decoder to convert the BCD codes to the logic values needed for the seven segments. A block diagram of the decoder is shown in Fig. 3.69. We begin by creating a truth table listing the segments to be activated for each decimal digit, as shown in Table 3.3, assuming active-high outputs. A K-map can then be drawn for each segment from which minimum logic equations can be derived. For example, Fig. 3.69 shows K-maps for segments a and b , with product of sums expressions derived. Table 3.4 shows MSP and MPS expressions for each of the seven segments.

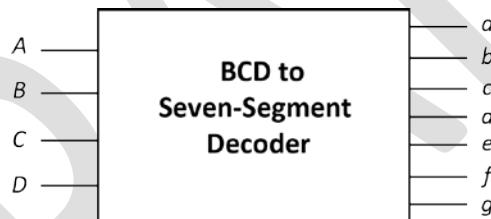


Figure 3.69 -- BCD to Seven-Segment Decoder

Table 3.3 BCD code to seven-segment code conversion (Active High)

Digit	BCD Code				Display Segments						
	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1

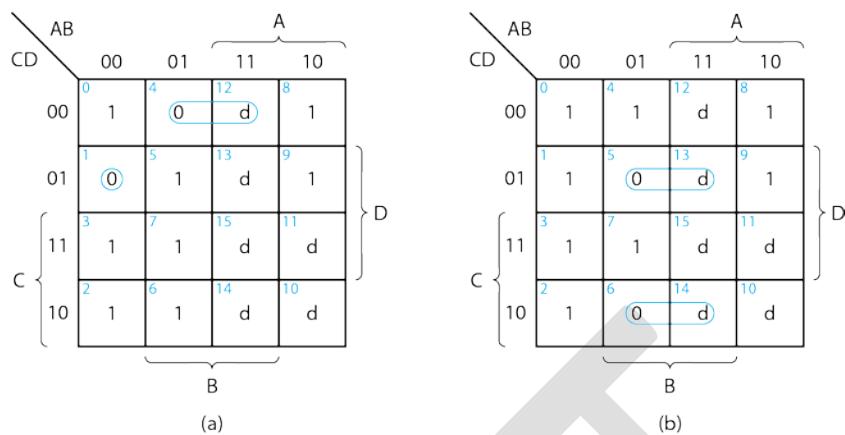


Figure 3.70 K-maps for a BCD to 7-segment code converter. (a) Segment $a = (\bar{B} + D)(A + B + C + \bar{D})$. (b) Segment $b = (\bar{B} + C + \bar{D})(\bar{B} + \bar{C} + D)$.

Table 3.4 Logic Expressions for Each Segment of a BCD to Seven-Segment Decoder

Segment	MSP Form	MPS Form
<i>a</i>	$A+C+BD+B'D'$	$(B'+C+D)(A+B+C+D')$
<i>b</i>	$B'+C'D'+CD$	$(B'+C+D')(B'+C'+D)$
<i>c</i>	$B+C'+D$	$B+C'+D$
<i>d</i>	$BC'D+B'D'+B'C+CD'$	$(B+C+D')(B'+C+D)(B'+C'+D')$
<i>e</i>	$B'D'+CD'$	$(B'+C)D'$
<i>f</i>	$A+BC'+BD'+C'D'$	$(A+B+D')(B+C')(C'+D')$
<i>g</i>	$A+B'C+BC'+BD'$	$(A+B+C)(B'+C'+D')$

This design procedure can be generalized to design logic circuits for converting data from any arbitrary code format to another. Using the truth table format of Table 3.3, all input codes are simply listed as inputs to the circuit, with the corresponding output codes as circuit outputs. Then a K-map is drawn and a logic equation derived for each output, with any unspecified input codes mapped as don't-care conditions.

Now let's take a look at realizations of the BCD to seven-segment decoder.

3.4.3.1 Schematic – A NAND-NAND realization of the decoder derived from the MSP expressions in Table 3.4 is shown in Fig. 3.71. Note that prime implicants $B'D'$, CD , $C'D'$, CD' , BC' , $B'C$, and BD' appear in more than one output function, so the NAND gate realizing one of these terms fans out to feed more than one output gate accordingly. Sharing gates across more than one function reduces the number of gates needed in the realization.

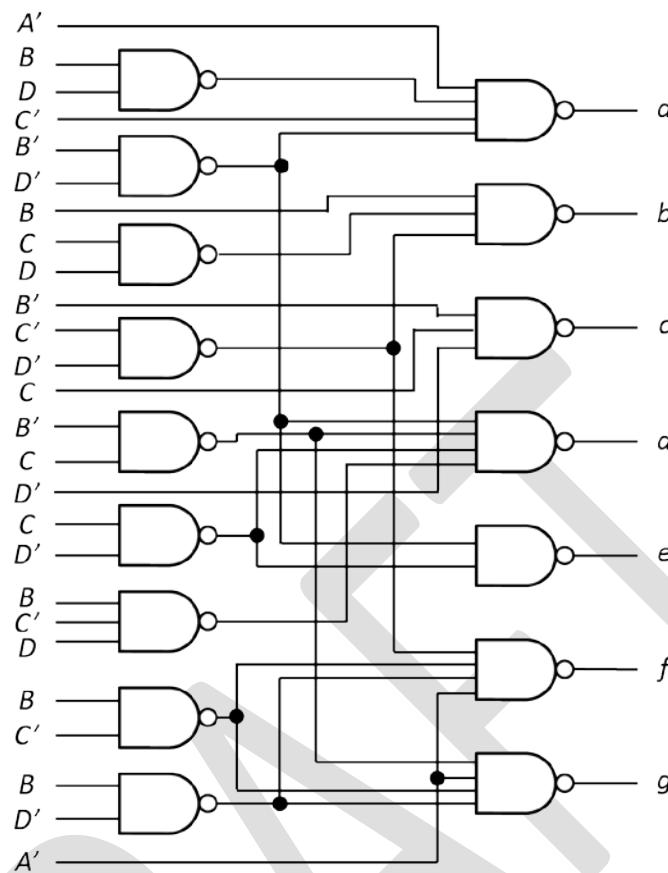


Figure 3.71 BCD to Seven-Segment NAND-NAND Realization

A realization of the BCD to seven-segment decoder with NOR gates derived from the MPS expressions in Table 3.4 is illustrated in Fig. 3.72. Note that $B' + C' + D'$ is the only prime implicate that appears in more than one MPS expression which means that only one NOR gate can be shared across functions in this realization.

Both of the above realizations produce logic 1 outputs when a segment should be lighted as defined in Table 3.3. So they are appropriate for driving common-cathode LEDs. A decoder for common-anode LEDs needs to realize the complement function for each segment. Figure 3.73 shows such a realization in NOR-NOR form that is derived from the MSP expressions in Table 3.4 by employing DeMorgan's theorem.

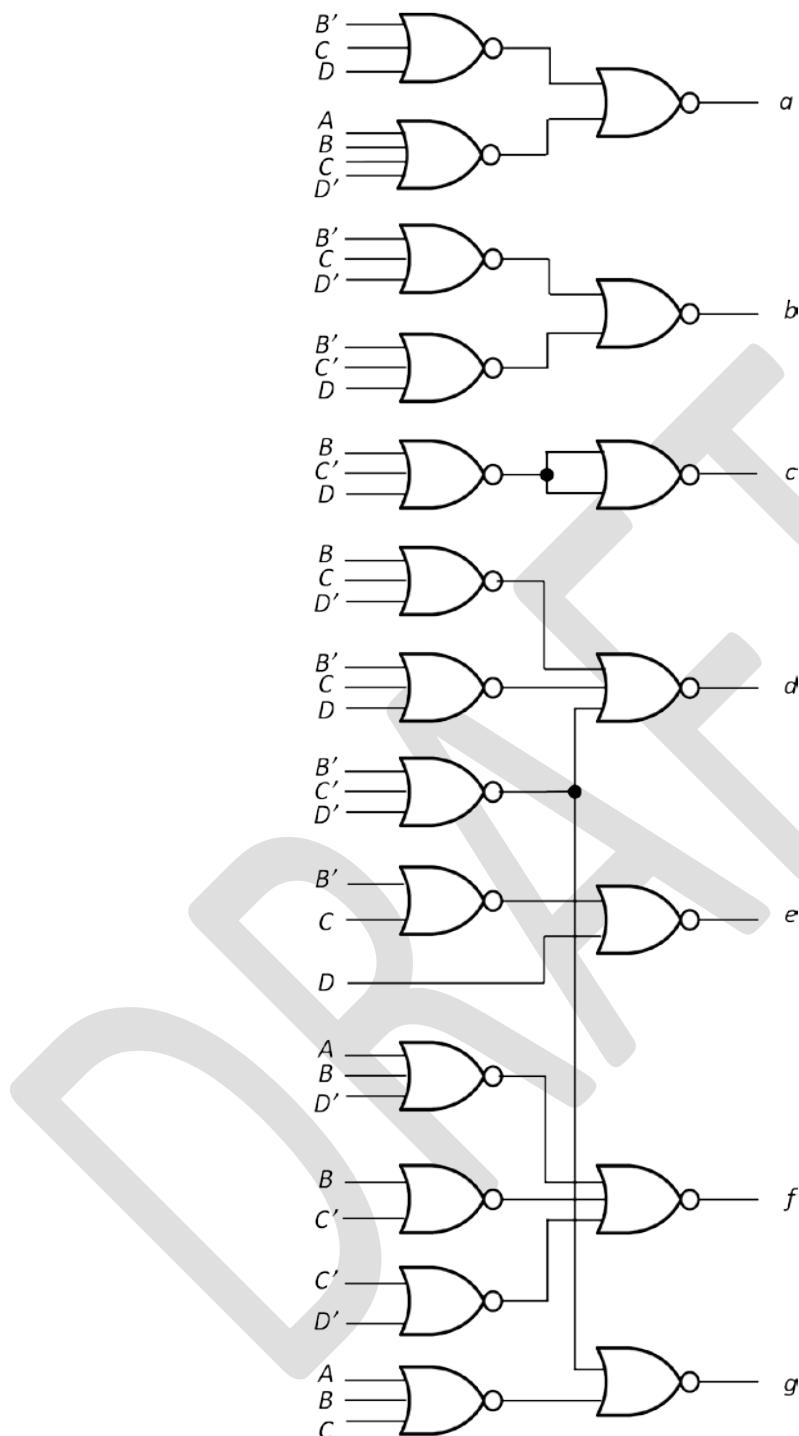


Figure 3.72 BCD to Seven-Segment NOR-NOR Realization

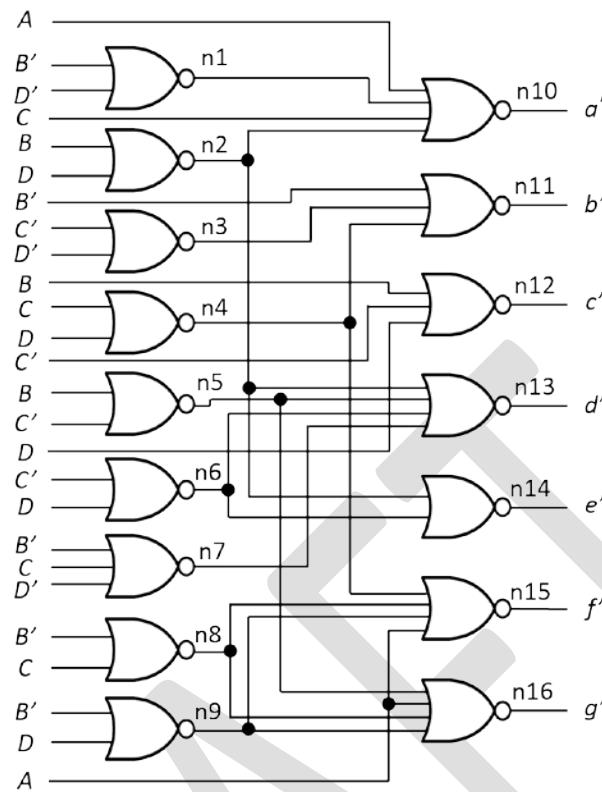


Figure 3.73 BCD to Seven-Segment NOR-NOR Active-Low Realization

3.4.3.2 Netlist – The netlist for the NOR-NOR circuit in Figure 3.73 is shown below. Note in this example it is assumed that variables A, B, C, D and their complements are inputs.

```

I1 Input A
I2 Input A'
I3 Input B
I4 Input B'
I5 Input C
I6 Input C'
I7 Input D
I8 Input D'

N1 NOR2 n1 B' D'
N2 NOR2 n2 B D
N3 NOR2 n3 C'D'
N4 NOR2 n4 C D
N5 NOR2 n5 B C'
N6 NOR2 n6 C'D
N7 NOR3 n7 B' C D'
N8 NOR2 n8 B' C
N9 NOR2 n9 B' D
N10 NOR4 a' A C n1 n2
N11 NOR3 b' B' n3 n4

```

N12	NOR3	c'	B	C'	D
N13	NOR4	d'	n2	n5	n6 n7
N14	NOR2	e'	n2	n6	
N15	NOR4	f'	n4	n8	n9 A
N16	NOR4	g'	n5	n8	n9 A
O1	Output	a'			
O2	Output	b'			
O3	Output	c'			
O4	Output	d'			
O5	Output	e'			
O6	Output	f'			
O7	Output	g'			

3.4.3.3 Verilog and VHDL Models

Verilog and VHDL behavioral models of an active-low BCD to seven-segment decoder are shown below. For these models, the output functions for the display segments are designated as a seven-bit register *sl[0:6]* for convenience. Also, an active-high *error* output is provided to indicate when invalid BCD codes are applied. Additionally, the seven-segment display is blanked when invalid inputs are applied. This example shows clearly the effectiveness and efficiency of HDLs for describing the functionality of combinational circuits.

```
//Verilog behavioral model
//Binary Coded Decimal to Seven Segment Display Decoder -- Active Low
module BCD2SevenSegmentLow (A,B,C,D,sl,error);
    input A,B,C,D;
    output reg [0:6] sl;
    output reg error;
    always
        case ({A,B,C,D})
            4'b0000: begin sl = 7'b0000001; error=1'b0; end
            4'b0001: begin sl = 7'b1001111; error=1'b0; end
            4'b0010: begin sl = 7'b0010010; error=1'b0; end
            4'b0011: begin sl = 7'b0000110; error=1'b0; end
            4'b0100: begin sl = 7'b1001100; error=1'b0; end
            4'b0101: begin sl = 7'b0100100; error=1'b0; end
            4'b0110: begin sl = 7'b0100000; error=1'b0; end
            4'b0111: begin sl = 7'b0001111; error=1'b0; end
            4'b1000: begin sl = 7'b0000000; error=1'b0; end
            4'b1001: begin sl = 7'b0001100; error=1'b0; end
            4'b1010: begin sl = 7'b1111111; error=1'b1; end
            4'b1011: begin sl = 7'b1111111; error=1'b1; end
            4'b1100: begin sl = 7'b1111111; error=1'b1; end
            4'b1101: begin sl = 7'b1111111; error=1'b1; end
            4'b1110: begin sl = 7'b1111111; error=1'b1; end
            4'b1111: begin sl = 7'b1111111; error=1'b1; end
        endcase
    endmodule
```

```
--VHDL behavioral model
--Binary Coded Decimal to Seven Segment Display Decoder -- Active Low
library ieee; use ieee.std_logic_1164.all;
entity BCD2SevenSegmentLow is
    port(A,B,C,D:  in  std_logic;
         Y: out std_logic_vector(6 downto 0);
         error:  out std_logic);
end;
architecture behavior of BCD2SevenSegmentLow is
    signal S: std_logic_vector(3 downto 0);
begin
    S <= D & C & B & A;
    with S select
        Y <= "0000001" when "0000",
        "1001111" when "0001",
        "0010010" when "0010",
        "0000110" when "0011",
        "1001100" when "0100",
        "0100100" when "0101",
        "0100000" when "0110",
        "0001111" when "0111",
        "0000000" when "1000",
        "0001100" when "1001",
        "1111111" when others;
    error <= '1' when S > "1001" else '0';
end;
```

3.4.4 Four-Function Arithmetic Logic Unit (add, subtract, AND, XOR)

Arithmetic logic units (ALUs) perform the functions needed to execute the arithmetic and logical operations that are included in a computer's instruction set. ALUs are also found in calculators and other devices where such functionality is required. A basic ALU that can perform the arithmetic operations *addition* or *subtraction* or logical operations *AND* or *XOR* on two four-bit numbers, *A* and *B*, will be designed in this section. A two's complement number system will be used. The concepts illustrated can be extended to ALUs with more bits and greater functionality. Design of multipliers and dividers will be covered in later sections and chapters. Figure 3.74 shows the logic symbol and function table for the ALU. The ALU output function *F* is determined by which operation is specified by selection codes *S₁* and *S₀*.

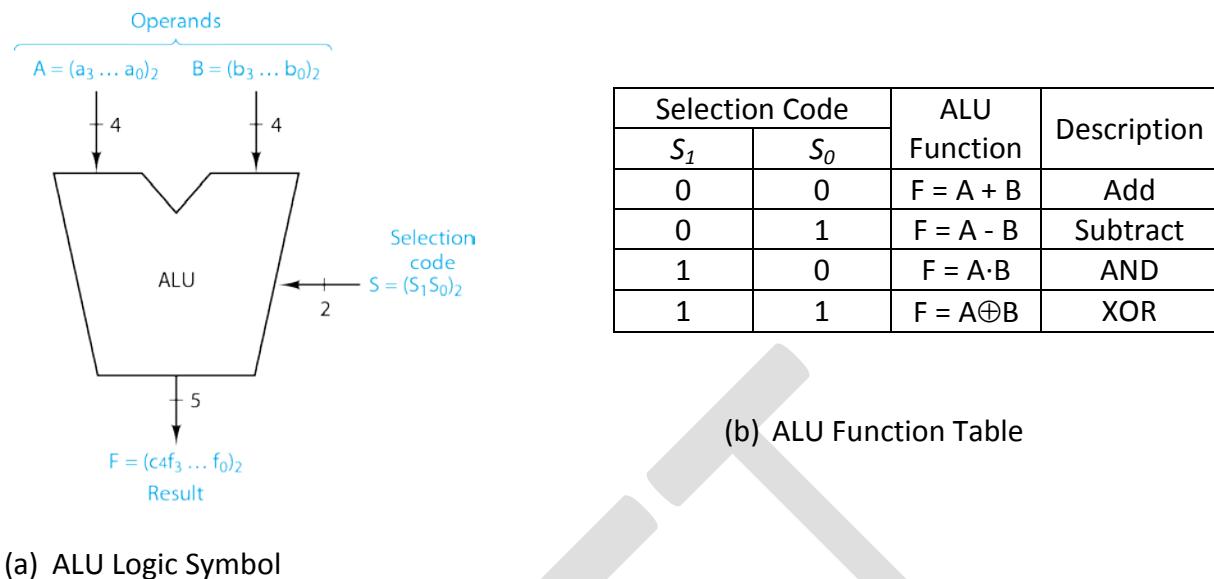


Figure 3.74 Four-Function Arithmetic Logic Unit

A *hierarchical, top-down* approach will be used in the design process. With this in mind, it is convenient to partition the ALU in to an *arithmetic unit* (AU) and a *logical unit* (LU) and to use a two-to-one multiplexer, or selector, to choose which unit needs to be used for a given operation. Note that selection codes were defined such that $S_1 = 0$ for addition and subtraction a 1 for AND and XOR, so S_1 is used as the select input of the multiplexer. This is shown in Figure 3.75.

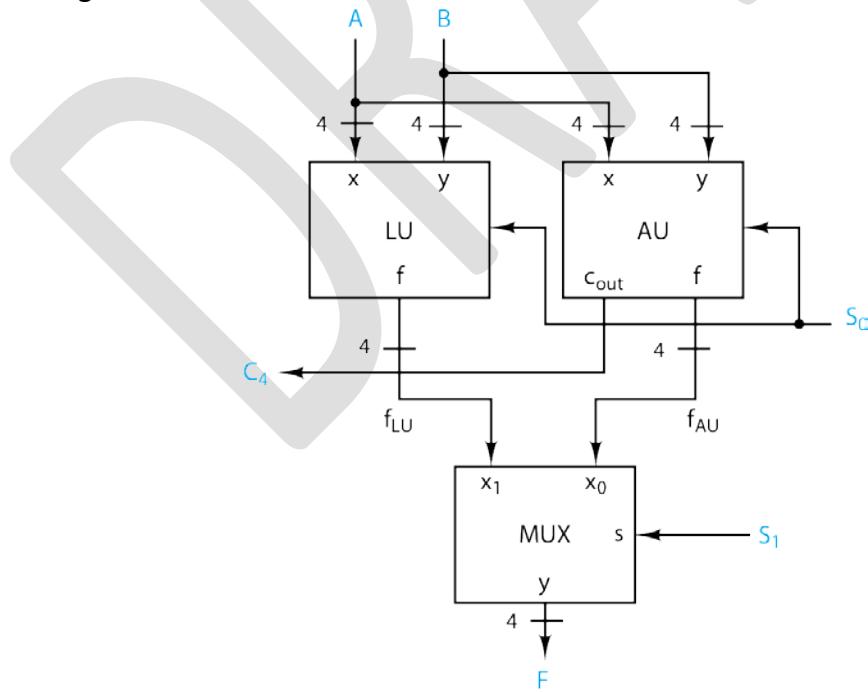


Figure 3.75 ALU Partitioned into Modules

3.4.4.1 Schematic design – Let's now design the *AU* and the *LU*. The *AU* performs either addition or subtraction as specified by the value of S_0 . Similarly, the value of S_0 determines whether the *LU* performs AND or XOR. The two's-complement adder/subtractor shown in Figure 3.56 can be used as the *AU* with the following mapping of terminology.

$$A \rightarrow P, B \rightarrow Q, F \rightarrow R, S_0 \rightarrow \text{Select}$$

Figure 3.76 shows a simpler realization of the adder/subtractor using full-adders and XOR gates. Note that XOR gates can replace the NOT gate and the multiplexer in the previous realization by taking advantage of the controllable NOT gate feature illustrated in Fig. 3.17.

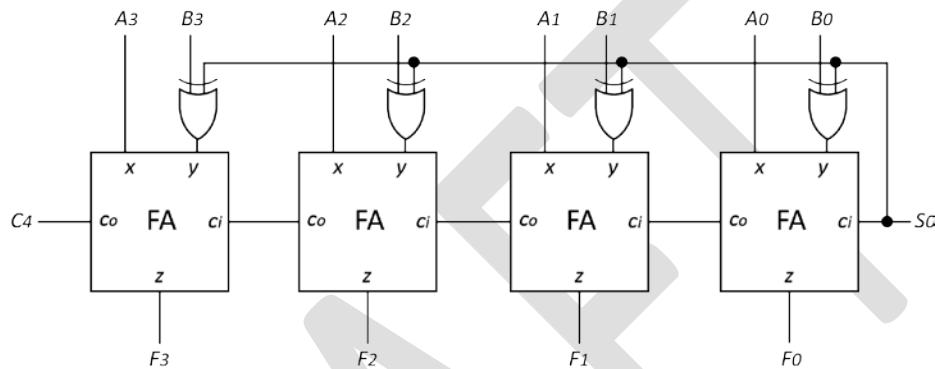


Figure 3.76 Arithmetic Unit Realization

The *LU* can be realized using AND and XOR gates and a two-to-one multiplexer with select input S_0 as shown in Fig. 3.77.

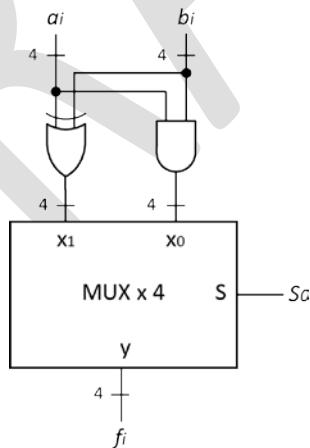


Figure 3.77 Logic Unit Realization

3.4.4.2 Verilog design – This example will illustrate how Verilog can be used as a hierarchical, or modular, design tool based on partitioning of the ALU as shown in Figure 3.75. The *AU* Verilog module given below incorporates two additional modules, *XOR2x4* and *RCAx4*. *XOR2x4* is a QUAD 2-input XOR module and *RCAx4* is a four-bit ripple-carry adder module patterned after the structural module described in Chapter 2, section 2.2.1. Note that the

RCAx4 module incorporates a previously defined full adder module. Design of the XOR2x4 module is given as a problem at the end of this chapter.

```
//Four-bit Arithmetic Unit
module AU (A,B,S,C4,F);
  input [3:0] A,B;
  input [1:0] S;
  output reg [3:0] F;
  output reg C4;
  wire [3:0] Y,C;
  XOR2x4 (B[3:0],{S[0],S[0],S[0],S[0]},Y[3:0]);
  RCAx4(A[3:0],Y[3:0],F[3:0],S[0],C4);
endmodule
```

A QUAD two-to-one multiplexer is used in both the LU and ALU. The following behavioral Verilog model of such a multiplexer will be used later in describing those modules.

```
//QUAD 2-to-1 multiplexer
module MUX2x4 (A,B,Y,S);
  input [3:0] A,B;
  input S;
  output reg [3:0] Y;
  always
    if (S==0) Y = A; else Y = B;
endmodule
```

The following is a Verilog description of the LU realization from Figure 3.77 that incorporates the above multiplexer module.

```
//Four-bit Logic Unit
module LU (A,B,F,S);
  input [3:0] A,B;
  input [1:0] S;
  output [3:0] F;
  wire [3:0] X1, X0;
  XOR2x4 (A,B,X1);
  AND2x4 (A,B,X0);
  MUX2x4 (X0,X1,F,S[0]);
endmodule
```

The Verilog modules of the AU, LU, and MUX2x4 can be integrated to describe the ALU as shown below.

```
//Four-bit ALU
module ALU (A,B,F,C4,S);
  input [3:0] A,B;
  input [1:0] S;
  output [3:0] F;
  output C4;
  wire [3:0] flu, fau;
  LU (A,B,flu,S[0]);
  AU (A,B,S[0],C4,fau);
  MUX2x4 (fau,flu,F,S[1]);
endmodule
```

3.4.4.3 VHDL Design – VHDL can also be used to describe hierarchical design structures and components. The following models the Arithmetic Unit (AU) corresponding to the partitioning given in Fig. 3.75.

--Four-bit Arithmetic Unit

```
library ieee; use ieee.std_logic_1164.all;
entity AU is
    port (A,B:  in  std_logic_vector(3 downto 0);
          F:      out std_logic_vector(3 downto 0);
          S:      in   std_logic;
          Cout:   out  std_logic);
end;
architecture structure of AU is
    component RCAx4
        port (A,B:  in  std_logic_vector(3 downto 0);
              S:      out std_logic_vector(3 downto 0);
              Cin:   in   std_logic;
              Cout:  out  std_logic);
    end component;
    component xor02
        port (A,B: in  std_logic;
              Y:     out std_logic);
    end component;
    signal Y: std_logic_vector(3 downto 0);
begin
    -- S=0 for A+B, S=1 for A-B
    X1: xor02 port map (B(3),S,Y(3));
    X2: xor02 port map (B(2),S,Y(2));
    X3: xor02 port map (B(1),S,Y(1));
    X4: xor02 port map (B(0),S,Y(0));
    -- Compute sum or difference
    R1: RCAx4 port map (A,Y,F,S,Cout);
end;
```

The following model describes the behavior of the Logic Unit (LU) shown in Fig. 3.75.

--Four-bit Logic Unit

```
library ieee; use ieee.std_logic_1164.all;
entity LU is
    port (A,B:  in  std_logic_vector(3 downto 0);
          F:      out std_logic_vector(3 downto 0);
          S:      in   std_logic);
end;
architecture structure of LU is
    signal X0,X1: std_logic_vector(3 downto 0);
begin
    X0 <= A xor B;
    X1 <= A and B;
    F  <= A when S = '0' else B;
end;
```

The final component of the ALU is the quad two-to-one multiplexer (MUX2x4) which is modeled below.

```
--QUAD 2-to-1 multiplexer
library ieee; use ieee.std_logic_1164.all;
entity MUX2x4 is
  port (A,B: in std_logic_vector(3 downto 0);
        Y: out std_logic_vector(3 downto 0);
        S: in std_logic);
end;
architecture behavioral of MUX2x4 is
begin
  with S select
    Y <= A when '0',
    B when others;
end;
```

The following model incorporates the AU, LU, and MUX2x4 to describe the ALU.

```
--Four-bit ALU
library ieee; use ieee.std_logic_1164.all;
entity ALU is
  port (A,B: in std_logic_vector(3 downto 0);
        F: out std_logic_vector(3 downto 0);
        C4: out std_logic;
        S: in std_logic_vector(1 downto 0));
end;
architecture structure of ALU is
  component LU
    port (A, B: in std_logic_vector(3 downto 0);
          F: out std_logic_vector(3 downto 0);
          S: in std_logic);
  end component;
  component AU
    port (A, B: in std_logic_vector(3 downto 0);
          F: out std_logic_vector(3 downto 0);
          S: in std_logic;
          Cout: out std_logic);
  end component;
  component MUX2x4
    port (A, B: in std_logic_vector(3 downto 0);
          Y: out std_logic_vector(3 downto 0);
          S: in std_logic);
  end component;
  signal flu, fau: std_logic_vector(3 downto 0);
begin
  Lunit: LU      port map (A, B, flu, S(0));
  Aunit: AU      port map (A, B, fau, S(0), C4);
  Munit: MUX2x4 port map (flu, fau, F, S(1));
end;
```

3.4.5 Binary Array Multiplier

Multiplication and division are more difficult to perform by hand than addition and subtraction and the same is true in computer hardware. This section will focus on the design of a combinational logic circuit for multiplying two binary numbers known as a *binary array multiplier (BAM)*. Sequential circuits for multiplication and division will be discussed in later chapters.

BAMs are based on the method used to perform multiplication by hand and on the observation that the multiplication of two one-bit numbers is equivalent to the AND of two one-bit numbers. This is illustrated below for two unsigned four-bit numbers.

$$\begin{array}{r}
 1010 \quad \text{multiplicand} \\
 \times \underline{0101} \quad \text{multiplier} \\
 \hline
 1010 \quad \text{partial product 0} \\
 0000 \quad \text{partial product 1} \\
 1010 \quad \text{partial product 2} \\
 \underline{0000} \quad \text{partial product 3} \\
 \hline
 00110010 \quad \text{product}
 \end{array}$$

Partial products are calculated by ANDing the corresponding bit of the multiplier with each bit of the multiplicand. So each partial product can be generated using four 2-bit AND gates. The partial products are shifted left according to the position of the multiplier bit with each shift equivalent to multiplying the partial product by 2. Partial products are then added to produce the product. An array of full adders can be used to add the partial products with each row of full adders shifted accordingly. Since multiple rows will be added, carries across several columns often result. This requires care in connecting the rows and columns of full-adders. The following algebraic equations for the multiplication of two unsigned binary numbers $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$ show how the various bit-pairs and partial products are generated, how partial products are shifted and added and how carries (C) are handled.

$$\begin{aligned}
 (a_3a_2a_1a_0) \times (b_3b_2b_1b_0) &= (a_3b_0 a_2b_0 a_1b_0 a_0b_0) \times 2^0 + (a_3b_1 a_2b_1 a_1b_1 a_0b_1) \times 2^1 \\
 &\quad + (a_3b_2 a_2b_2 a_1b_2 a_0b_2) \times 2^2 + (a_3b_3 a_2b_3 a_1b_3 a_0b_3) \times 2^3 \\
 &= a_0b_0 + (a_1b_0 + a_0b_1) \times 2^1 + (a_2b_0 + a_1b_1 + a_0b_2 + C_1) \times 2^2 \\
 &\quad + (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 + C_2) \times 2^3 \\
 &\quad + (a_3b_1 + a_2b_2 + a_1b_3 + C_3) \times 2^4 + (a_3b_2 + a_2b_3 + C_4) \times 2^5 \\
 &\quad + (a_3b_3 + C_5) \times 2^6 + C_6 \times 2^7 \\
 &= p_7p_6p_5p_4p_3p_2p_1p_0
 \end{aligned}$$

3.4.5.1 Schematic

The schematic diagram of a 4-by-4 (4 x 4) BAM that realizes the previous equation is shown in Figure 3.78. In general, n^2 AND gates and $n^2 - n$ full adders are needed to realize an $n \times n$ BAM.

The multiplication time of the four-by-four BAM is defined by the worst-case propagation delay through the circuit which occurs when the least significant bits of the opeands, a_0 or b_0 , affect the most significant bit of the product, p_7 . The longest propagation path in the above circuit involves the AND gates in the second column, two full adders in each of the first two rows of full adders plus all of those in the last row for a total of one AND gate and eight full adders. This results in a multiplication time of $t_{gate} + 8(2t_{gate}) = 17t_{gate}$. For an $n \times n$ multiplier, the multiplication time is $(6n - 7)t_{gate}$.

3.4.5.2 Verilog code

The AND2x4 and RCAx4 modules introduced in the previous ALU example can be used to describe the 4 x 4 BAM as shown below.

```
//Four-by-Four Binary Array Multiplier
module FourXFourBAM (A,B,P);
  input [3:0] A,B;
  output [7:0] P;
  wire [3:0] X1,Y1,X2,Y2,X3,Y3,Z1,Z2;
  AND2x4 (A,{B[0],B[0],B[0],B[0]},{Y1[2:0],P[0]});
  AND2x4 (A,{B[1],B[1],B[1],B[1]},X1);
  RCAx4 (X1,{1'b0,Y1[2:0]},{Z1[3:1],P[1]},1'b0,Y2[3]);
  AND2x4 (A,{B[2],B[2],B[2],B[2]},X2);
  AND2x4 (A,{B[3],B[3],B[3],B[3]},X3);
  RCAx4 (X2,{Y2[3],Z1[3:1]},{Z2[3:1],P[2]},1'b0,Y3[3]);
  RCAx4 (X3,{Y3[3],Z2[3:1]},P[6:3],1'b0,P[7]);
endmodule
```

3.4.5.3 VHDL Code

```
--Four-by-Four Binary Array Multiplier
library ieee; use ieee.std_logic_1164.all;
entity FourXFourBAM is
  port (A,B: in std_logic_vector(3 downto 0);
        P: out std_logic_vector(7 downto 0));
end;
architecture structure of FourXFourBAM is
  component RCAx4
    port (A, B: in std_logic_vector (3 downto 0);
          S: out std_logic_vector (3 downto 0);
          Cout: out std_logic);
  end component;
  component and02
    port (A,B: in std_logic;
          Y: out std_logic);
  end component;
  signal X0,X1,X2,X3: std_logic_vector(3 downto 0);
  signal Z1,Z2: std_logic_vector(4 downto 0);
```

```
begin
    -- First partial product: B0 x A
    A00: and02 port map (A(0),B(0),P(0));
    A01: and02 port map (A(1),B(0),X0(0));
    A02: and02 port map (A(2),B(0),X0(1));
    A03: and02 port map (A(3),B(0),X0(2));
        X0(3) <= '0';
    -- Second partial product: B1 x A
    A10: and02 port map (A(0),B(1),X1(0));
    A11: and02 port map (A(1),B(1),X1(1));
    A12: and02 port map (A(2),B(1),X1(2));
    A13: and02 port map (A(3),B(1),X1(3));
    -- Add first and second partial products
    AD1: RCAx4 port map (X0,X1,Z1(3 downto 0),Z1(4));
        P(1) <= Z1(0);
    -- Third partial product: B2 x A
    A20: and02 port map (A(0),B(2),X2(0));
    A21: and02 port map (A(1),B(2),X2(1));
    A22: and02 port map (A(2),B(2),X2(2));
    A23: and02 port map (A(3),B(2),X2(3));
    -- Add in third partial product
    AD2: RCAx4 port map (X2,Z1(4 downto 1),Z2(3 downto 0),Z2(4));
        P(2) <= Z2(0);
    -- Fourth partial product: B3 x A
    A30: and02 port map (A(0),B(3),X3(0));
    A31: and02 port map (A(1),B(3),X3(1));
    A32: and02 port map (A(2),B(3),X3(2));
    A33: and02 port map (A(3),B(3),X3(3));
    -- Add in fourth partial product
    AD3: RCAx4 port map (X3,Z2(4 downto 1),P(6 downto 3), P(7));
end;
```

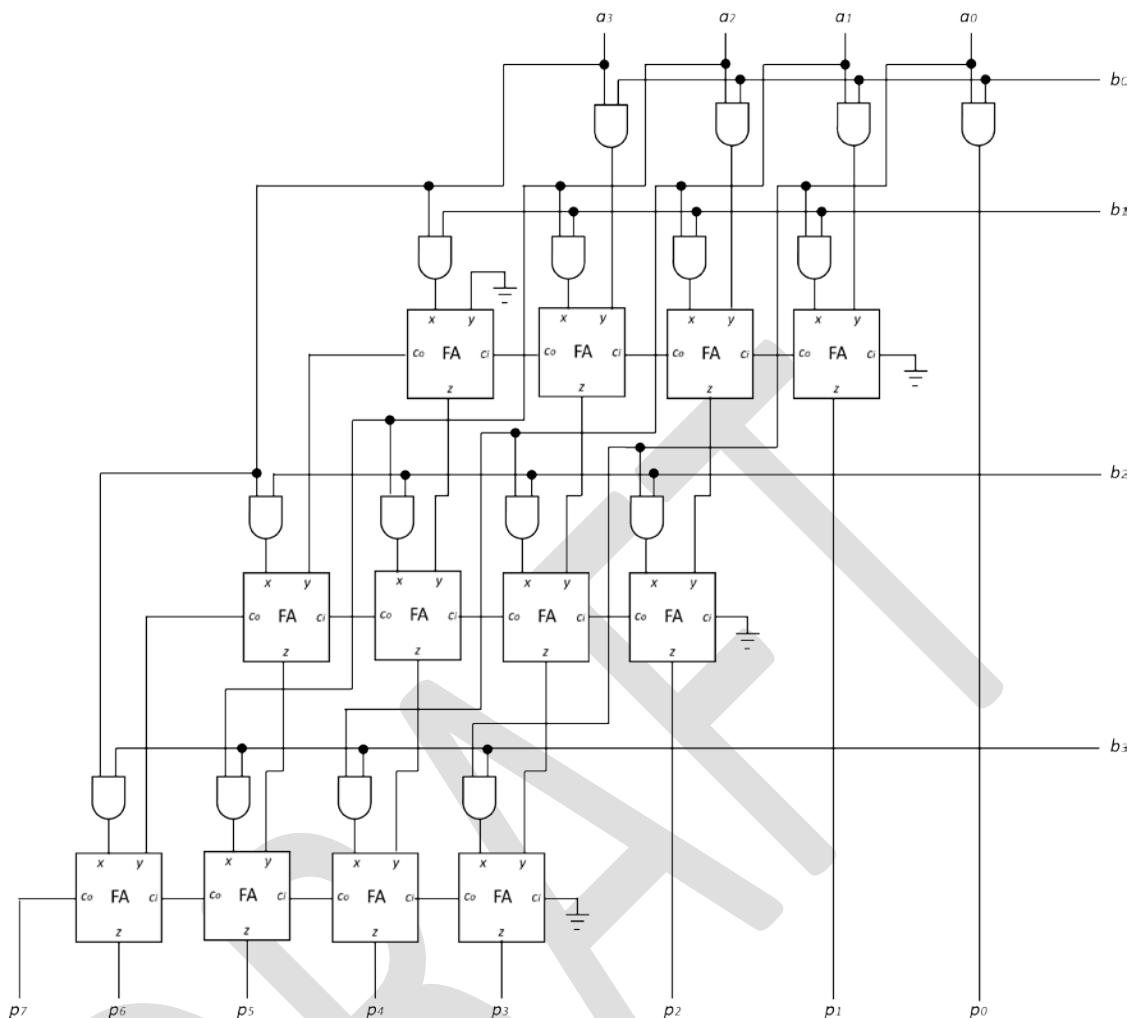


Figure 3.78 Four-by-Four Binary Array Multiplier

3.5 Summary and Review Questions

This chapter provided an introduction to the fundamentals of combinational logic circuit design and analysis. It was shown that any Boolean function can be realized with a combination of AND, OR, and NOT gates in a two-level circuit derived from sum of products (SOP) or product of sums (POS) forms. SOP expressions yield AND-OR circuit structures and POS yield OR-AND. It was also shown that any two-level AND-OR circuit can be transformed in to an equivalent two-level NAND-NAND circuit and that any OR-AND circuit can be transformed in to a NOR-NOR equivalent circuit. This demonstrated that {AND, OR, NOT}, {NAND}, and {NOR} are functionally complete sets of operations or gates. Exclusive-OR (XOR), AND-OR-Invert (AOI), and OR-AND-Invert (OAI) gates were also introduced. XOR gates were shown to be useful in realizing arithmetic circuits and various coding applications. AOI and OAI gates are often referred to as complex gates and are included in the standard cell libraries of many computer-aided design tools.

Fan-in was defined to be the maximum number of inputs that are allowed for a given logic gate. Fan-in may prevent a given logic function from being realized in the form of a two-level logic circuit. Under such circumstances, realizations will need to have three or more levels of logic gates. It was demonstrated how factoring of logic expressions using the postulates and theorems of Boolean algebra can be used to find these realizations.

The chapter demonstrated how a logic circuit can be analyzed, or reverse engineered, to determine the logic function realized. Truth tables, Boolean algebra, and timing diagrams are useful tools to be used in the analysis process. It was shown that timing diagrams can also be used to show the effects of propagation, or gate, delays on the behavior of a logic circuit. Examples illustrated how the speed or circuit propagation time can be formulated in terms of gate delay.

Higher-level devices such as decoders, encoders, multiplexers, and demultiplexers were also introduced in the chapter. It was demonstrated that decoders and multiplexers, with other logic circuitry when necessary, can be used to realize Boolean functions. Seven-segment displays were discussed as were binary-coded-decimal (BCD) to seven-segment decoders. Arithmetic circuits were described for addition, subtraction, multiplication, and comparison.

Schematic and hardware description language (HDL) design approaches were covered. Schematic design results in realizations in the form of logic circuits represented by the interconnection of gates, higher-level logic devices, or other hardware. HDL design results in programming-language-like code that describes the functions to be realized but not the actual realization which will typically be a programmable logic device. Both Verilog and VHDL were used to illustrate the designs.

Material presented in this chapter is fundamental for the understanding of sequential circuits and more advanced topics covered later in the book and should be mastered before moving on. The following review questions will help the reader assess her/his level of understanding.

1. Can you design two-level AND-OR and NAND-NAND realizations of a given Boolean logic function?
2. Can you design two-level OR-AND and NOR-NOR realizations of a given Boolean logic function?
3. Can you design multi-level, i.e., three or more, realizations of a given logic function with logic gates having fan-in constraints?
4. Can you use XOR gates to realize odd and even parity functions?
5. Do you understand the concept of complex gates and how to use them in the realization of logic functions?
6. Can you use Boolean algebra to derive the Boolean expression realized by a given combinational logic circuit?
7. Can you use Boolean algebra to derive the truth table of a given combinational logic circuit?
8. Do you understand the basic concept of a timing diagram?
9. Given a logic circuit diagram and a timing diagram of its inputs, can you derive a timing diagram of the circuit outputs?
10. Can you derive the maximum propagation delay through a logic circuit?
11. Can you formulate a structural Verilog or VHDL module to describe a given combinational logic circuit?
12. Can you formulate a behavioral Verilog or VHDL module to describe a given combinational logic circuit?
13. Can you formulate a Verilog or VHDL module to meet design specifications for a combinational logic circuit?
14. Can you analyze a Verilog or VHDL module and determine its truth table?
15. Can you design a decoder to translate from one binary code to another?
16. Can you use a decoder and other logic circuitry to realize a given logic function?
17. Can you use a multiplexer and other logic circuitry to realize a given logic function?
18. Can you design an eight-bit two's complement adder/subtractor?
19. Can you design a four-bit ALU for the operations add, subtract, OR, and NOT?
20. Can you design an eight-bit priority encoder?

3.6 Collaboration Exercises

These exercises are intended to be worked in class in groups of two or three students with solutions presented to the remainder of the class. The exercises range from easy to test basic understanding to challenging to promote deeper understanding and development of problem solving skills. The design-oriented exercise solutions may also be realized and demonstrated to the class if appropriate equipment and facilities are available.

3.1. Find AND-OR, and NAND-NAND realizations of the following function.

$$h(x,y,z) = \sum m(1,3,6,7)$$

Minimize the number of gates in each realization. Assume that uncomplemented and complemented literals (x, x', y, y', z, z') are available as inputs.

3.2. Find OR-AND and NOR-NOR realizations of the function.

$$h(x,y,z) = \sum m(1,3,6,7)$$

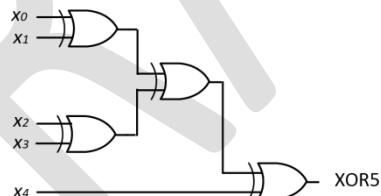
Minimize the number of gates in each realization. Assume that uncomplemented and complemented literals (x, x', y, y', z, z') are available as inputs.

3.3. Implement the following function using only NAND2 gates.

$$f(A,B,C) = \sum m(2,3,4,5,7)$$

Assume all literals are available as inputs.

3.4. Derive the truth table and logic equation for the following circuit.



3.5. Write Verilog structural and dataflow models of the circuit in exercise 4.

3.6. Design a realization of the function in exercise 1 using a NAND gate and a three-to-eight decoder as logic elements.

3.7. Design a realization of the function in exercise 1 using a four-to-one multiplexer as a logic element. Minimize the number of additional logic elements that are needed.

3.8. Design a realization of a full adder that uses a decoder and two NOR gates as logic elements.

3.9. Design a two-to-four decoder that can be used as a building block for larger decoders. Draw a NAND gate realization. Write a Verilog model.

- 3.10. Using your two-to-four decoder from exercise 9, design an 8-to-256 decoder. Draw a circuit diagram and write a Verilog model.
- 3.11. Design a four-bit carry-look-ahead adder using basic gates as logic elements. Capture your design as a structural Verilog model.
- 3.12. Write a Verilog or VHDL behavioral model of a four-bit magnitude comparator like the one in Fig. 3.60.
- 3.13. Redesign the four-function ALU in section 3.4.4 by replacing the ripple-carry adder with a carry-look-ahead adder.
- 3.14. Design a four-bit ALU that performs addition, subtraction, multiplication, and XOR. Assume a two's complement number system.
- 3.15. Given the four-function ALU in section 3.4.4, design the logic for displaying the inputs and outputs in decimal on seven-segment displays.
- 3.16. Design an encoder for a ten-button numeric key-pad. Assume only one key will be depressed at a given time. Is this a realistic assumption? Capture your design as a Verilog or a VHDL model.
- 3.17. Enhance the ALU in section 3.3.4 by adding an output (Z) that indicates when a result is zero and an output (N) that indicates when a result is negative.
- 3.18. The ALU in section 3.4.4 does not explicitly define how carry-out and overflow are defined for the AND and XOR functions. Redesign the ALU so that Z and N are always logic 0 for these functions.
- 3.19. Design a logic circuit for encoding the eight-bit ASCII as an odd-parity code. Capture your design as a parity-tree circuit and as a Verilog dataflow model.
- 3.20. Design a circuit for checking the parity of the encoded ASCII code from the previous exercise. Capture your design as a parity-tree circuit and as a Verilog dataflow model.
- 3.21. SystemVerilog is an extension of Verilog. Research the history of SystemVerilog and Verilog and identify the latest standards of each language. Summarize the primary features of SystemVerilog that are not found in Verilog. Are there any features of Verilog not available in SystemVerilog?

Problems

3.1 Design minimal 2-level AND-OR and NAND-NAND realizations of the following logic function. Draw circuit diagrams of your realizations.

$$f(x,y,z) = \sum m(0,1,4,5,6)$$

3.2 Design minimal 2-level OR-AND and NOR-NOR realizations of the function defined in problem 3.1. Draw circuit diagrams.

3.3 Write Verilog and/or VHDL dataflow models for the logic function given in problem 3.1.

3.4 Write Verilog and/or VHDL structural models for your NAND-NAND design from problem 3.1.

3.5 Design minimal 2-level AND-OR and NAND-NAND realizations of the following logic function. Draw circuit diagrams.

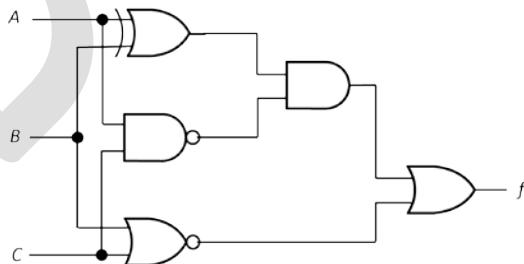
$$g(a,b,c,d) = \prod M(2,3,6,8,9,12)$$

3.6 Design minimal 2-level OR-AND and NOR-NOR realizations of the function defined in problem 3. Draw circuit diagrams.

3.7 Write Verilog and/or VHDL dataflow models for the function defined in problem 3.5.

3.8 Write Verilog and/or VHDL structural models for your NOR-NOR design from problem 3.5.

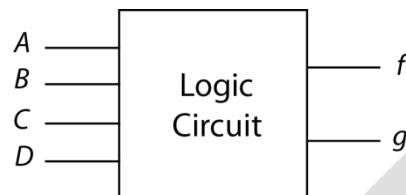
3.9 Find MSOP and MPOS expressions for the function realized by the logic circuit below. What is the worst-case propagation delay through this circuit if each gate has a delay of t_{gate} ?



3.10 Design a minimal realization of the circuit from problem 3.9 using only NAND2 and/or NOR2 gates. Draw the circuit diagram.

3.11 Design a logic circuit with inputs A, B, C, D and outputs f and g as shown below. The

inputs represent the four bits in a BCD digit. Output f should be *logic-1* if the input digit is divisible by 3. Output g should be *logic-0* if and only if the input is a valid BCD digit. Assume you have NAND2, NAND3, NAND4, and NOT gates to use in your realization. Minimize the number of gates and literals used. Draw your circuit diagram.



3.12 Design a minimum two-level NAND realization for the following logic function.

$$f(A, B, C) = \sum m(1, 2, 3, 5, 6, 7, 8, 9, 12, 14)$$

3.13 Repeat problem 3.12 for NOR gates.

3.14 Design a minimal two-level NAND realization for each of the following logic functions.

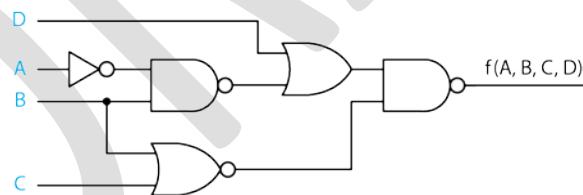
(a) $f(A, B, C) = \sum m(0, 2, 3, 7)$

(b) $f(A, B, C, D) = \sum m(0, 2, 8, 10, 14, 15)$

(c) $f(A, B, C, D, E) = \sum m(4, 5, 6, 7, 25, 27, 29, 31)$

3.15 Design a minimal two-level NOR realization for the functions in problem 3.14.

3.16 Find a minimum NAND-NAND equivalent circuit for the one below.

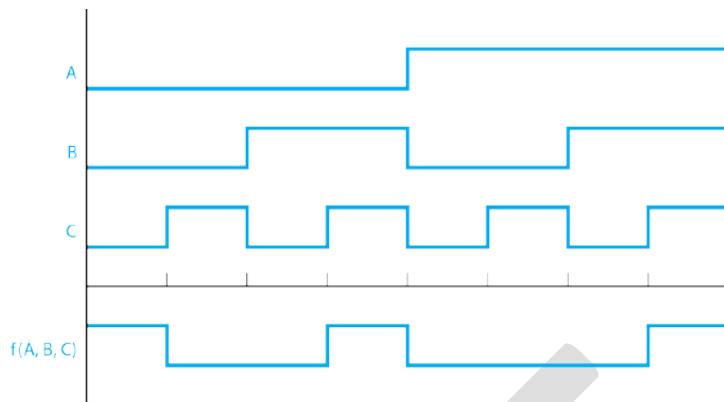


3.17 Find a minimum NOR-NOR equivalent circuit for the one above.

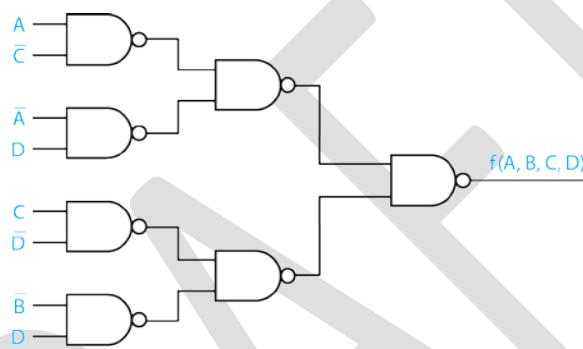
3.18 Write Verilog and/or VHDL dataflow models of the function from Problem 3.16.

3.19 Write Verilog and/or VHDL structural models of the circuit in Problem 3.16.

3.20 Given the timing diagram below, find the simplest Boolean expression for $f(A, B, C)$.



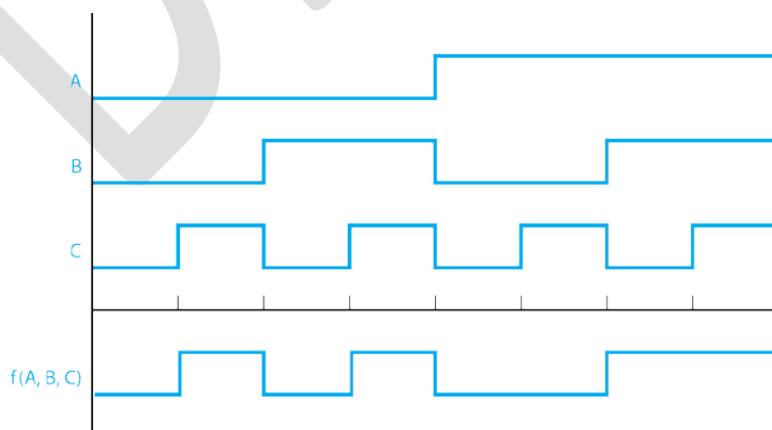
3.21 Given the logic circuit below, find a minimum two-level NOR-NOR realization.



3.22 Write Verilog and/or VHDL structural models for the circuit given in Problem 3.21.

3.23 Write Verilog and/or dataflow models for the function realized by the circuit in Problem 3.21.

3.24 For the timing diagram shown below, find both a minimum NAND-NAND and a minimum NOR-NOR realization.



3.25 A long hallway has three doors, one at each end and one in the middle. A switch is located at each door to operate the lights along the hallway. Label the switches *A*, *B*, and *C*. Assume that flipping any of the switches will turn the light *on* if it's currently *off*, or *off* if it's currently *on*. Write a Verilog and/or VHDL model of the logic needed to control the lights.

3.26 Joe, Jack, and Jim get together once a week to either go to a movie or go bowling. To decide what to do, they vote and a simple majority wins. Assuming a vote for the movie is represented as a 1, design a logic circuit that automatically computes the decision. Realize your design with a minimum number of NAND and NOR gates. Write a Verilog and/or VHDL behavioral model of design.

3.27 Derive the logic equation for a circuit with three inputs *A*, *B*, *C* and output *z*.. The output is to be high only when exactly one of the three inputs is high. Realize your circuit with a minimum of NAND and NOR gates. Also write a Verilog and/or VHDL behavioral model.

3.28 We wish to design a logic circuit with four inputs *A*, *B*, *C*, and *D*. The output is to be high only when a majority of the inputs is high. Realize the circuit using only NOR gates.

3.29 A logic circuit has four inputs *A*, *B*, *C*, and *D*. Find the logic equations for the circuit if the output is to be high only when an odd number of the inputs is high. Realize the circuit using a minimum number of logic gates.

3.30 The input to a logic circuit consists of four signal lines *A*, *B*, *C*, and *D*. These lines represent a 4-bit binary number, where *A* is the most significant bit and *D* the least significant bit. Design the logic circuit such that the output is high only when the binary input is less than $(0111)_2 = 7_{10}$. Use any type of logic gate to realize the circuit.

3.31 A burglar alarm controller is designed so that it senses four input signal lines. Line *A* is from a secret control switch, line *B* is from a pressure sensor under a steel safe in a locked closet, line *C* is from a battery-powered clock, and line *D* is connected to a switch on the locked closet door. The following conditions produce a logic 1 voltage on the corresponding input line.

- A*: The control switch is closed.
- B*: The safe is in its normal position in the closet.
- C*: The clock is between 0900 and 1600 hours.
- D*: The closet door is closed.

Write a logic expression for the burglar alarm that produces a logic 1 (rings a bell) when the safe is moved and the control switch is closed, or when the closet is opened after banking hours, or when the closet is opened with the control switch open. Write a Verilog and/or VHDL model for alarm controller.

- 3.32** Derive the logic equations for a circuit that will subtract two 2-bit binary numbers, $(X_1 X_0)_2 - (Y_1 Y_0)_2$, and produce as an output the resulting number $(D_1 D_0)_2$ and borrow condition B_1 . Write a Verilog and/or VHDL model of the subtractor.

$$\begin{array}{r} X_1 X_0 \\ - Y_1 Y_0 \\ \hline B_1 D_1 D_0 \end{array}$$

- 3.33** Design a logic circuit that accepts BCD inputs and gives an output of logic 1 only when the input decimal digit is divisible by 3. Use a four-variable K-map to design your circuit.

- 3.34** Design a logic circuit that has five input variables and one output variable. Four of the input variables represent BCD digits, and the fifth is a control line. While the control line is at logic 0, the output should be logic 1 only if the BCD digit is greater than or equal to 5. While the control line is high, the output should be logic 1 only if the BCD digit is less than or equal to 5.

- 3.35** Design a multiple-output logic circuit whose input is a BCD digit and whose outputs are defined as follows: f_1 : Detects input digits that are divisible by 4, f_2 : Detects numbers greater than or equal to 3, f_3 : Detects numbers less than 7. Write a Verilog and/or VHDL model of the circuit. Also, realize the circuit using NAND gates.

- 3.36** Design a multiple-output combinational network that has two input signals x_0 and x_1 , two control signals c_0 and c_1 , and two output functions f_0 and f_1 . The control signals have the following effect on the outputs. For example, when $c_0 = 0$ and $c_1 = 1$, then $f_0(x_0, x_1, c_0, c_1) = x_0$ and $f_1(x_0, x_1, c_0, c_1) = 0$. Write a Verilog and/or VHDL model of the circuit. Also, realize the circuit using NOR gates.

c_0	c_1	f_0	f_1
0	0	0	0
0	1	x_0	0
1	0	0	x_1
1	1	x_0	x_1

- 3.37** Derive logic expressions for outputs 5 and 11 of the 4-to-16 decoder module in Fig. 3.33. Using these expressions, describe the operation of the decoder and the function of the enable inputs.

- 3.38** Design a Verilog and/or VHDL behavioral module for a 4-to-16 decoder. The inputs are $\{D, C, B, A\}$ and the outputs are active low $\{\overline{O}_0, \overline{O}_1, \dots, \overline{O}_{15}\}$. The decoder should have one active-high enable line, E .

3.39 Design a 5-to-32 decoder using 3-to-8 decoder modules as building blocks. Assume each 3-to-8 decoder has one active-low enable input, \bar{E}_1 , and one active-high enable input, E_2 .

3.40 Realize each of the following sets of functions using a 4-to-16 decoder module and output logic gates (choose NAND or AND gates to minimize the fan-in of the output gates).

$$(a) \quad f_1(a,b,c,d) = \sum m(2,4,10,11,12,13)$$

$$f_2(a,b,c,d) = \prod M(0\text{to}3,6\text{to}9,12,14,15)$$

$$f_3(a,b,c,d) = \bar{b}c + \bar{a}\bar{b}d$$

$$(b) \quad f_1(a,b,c,d) = \sum m(0,1,7,13)$$

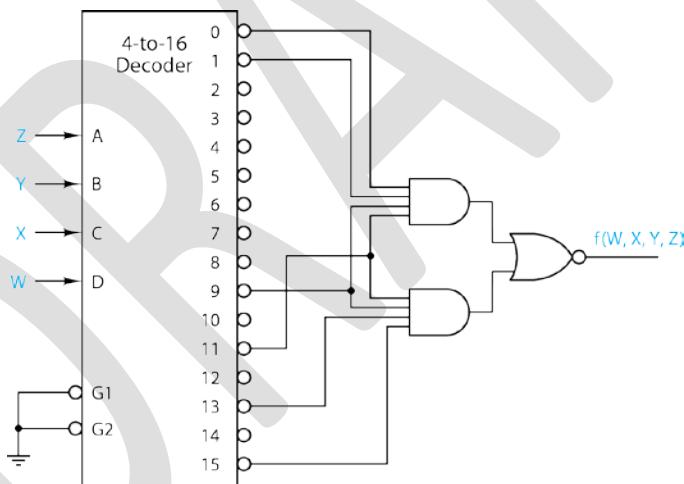
$$f_2(a,b,c,d) = ab\bar{c} + acd$$

$$f_3(a,b,c,d) = \prod M(0,1,2,5,6,7,8,9,11,12,15)$$

(c) Repeat part (a) for the complements of the three functions.

(d) Repeat part (b) for the complements of the three functions.

3.41 Given the circuit below, find the minimum SOP expression for $f(W, X, Y, Z)$.



3.42 Design a logic circuit realization binary-to-decimal decoder whose inputs are active-high, 4-bit, BCD-encoded numbers ($x_3x_2x_1x_0$) and whose outputs are the active-low lines ($d_9, d_8, \dots, d_1, d_0$).

3.43 Design a behavioral Verilog and/or VHDL module for the decoder described above.

3.44 Design a decoder whose input is a 4-bit code ($c_3c_2c_1c_0$) representing the hexadecimal digits {0 to 9,A,b,C,d,E,F}, with the outputs driving a seven-segment display digit to display the corresponding character. (The letters *B* and *D* are normally displayed in lowercase to distinguish them from the numerals 8 and 0, respectively.)

3.45 Write a behavioral Verilog and/or VHDL model for the HEX to seven-segment decoder described in problem 3.44.

3.46 Design a logic circuit that converts a 4-bit number from sign magnitude format to two's complement format. Use a two-level AND-OR circuit for each of the four outputs.

3.47 Repeat problem 3.46 for behavioral Verilog and/or VHDL.

3.48 Design a decoder that converts a 4-bit number from Gray code to binary code.

3.49 Repeat problem 3.48 for behavioral Verilog and/or VHDL.

3.50 Design a 4-to-2 bit priority encoder circuit using only NOR gates. The inputs are $a_3a_2a_1a_0$ with a_3 having the highest priority and a_0 the lowest. The outputs are y_1y_0 , indicating the highest-priority active input, and G , which indicates that at least one input is active.

3.51 Repeat problem 3.50 using behavioral Verilog and/or VHDL.

3.52 Derive logic expressions for the outputs of the AND gates driven by inputs D_3 and D_6 of the 8-to-1 multiplexer module in Fig. 3.45. Using these expressions, describe the operation of the multiplexer and the function of the strobe (enable) input.

3.53 Write a behavioral Verilog and/or VHDL module describing the functionality of the 8-to-1 multiplexer in Fig. 3.52.

3.54 Design a 5-to-1 multiplexer circuit, minimizing the number of gates in the circuit as much as possible.

3.55 Write a behavioral Verilog and/or VHDL module to describe the functionality of a 5-to-1 multiplexer.

3.56 Design a three-input/3-bit multiplexer. Use only NAND gates in your realization.

3.57 Repeat problem 3.56 using behavioral Verilog and/or VHDL.

3.58 Design an 8-to-1 multiplexer using only 4-to-1 multiplexer modules without enable lines. (Do not use any additional gates.)

3.59 Design a dual (2-bit) 16-input multiplexer using only 8-to-1 multiplexer modules, OR gates, and inverters.

3.60 Write a behavioral Verilog and/or VHDL model of an 8-to-1 multiplexer. Then write a model of a 2-bit, 16-to-1 multiplexer that uses the 8-to-1 multiplexer as a component.

3.61 Realize each of the following functions with a 4-to-1 multiplexer module.

$$(a) f_1(a,b,c) = \sum m(2,4,5,7)$$

$$(b) f_2(a,b,c) = \prod M(0,6,7)$$

$$(c) f_3(a,b,c) = (a + \bar{b})(\bar{b} + c)$$

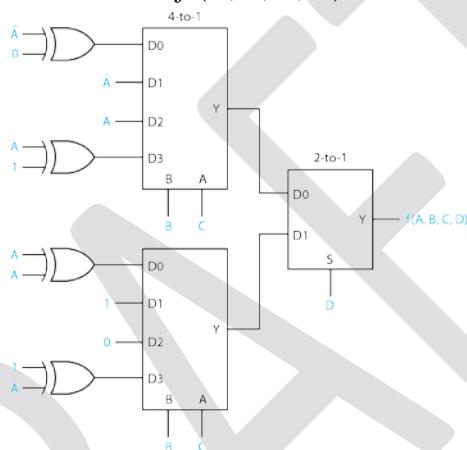
3.62 Realize each of the following functions with an 8-to-1 multiplexer module.

$$(a) f(b,c,d) = \sum m(0,2,3,5,7)$$

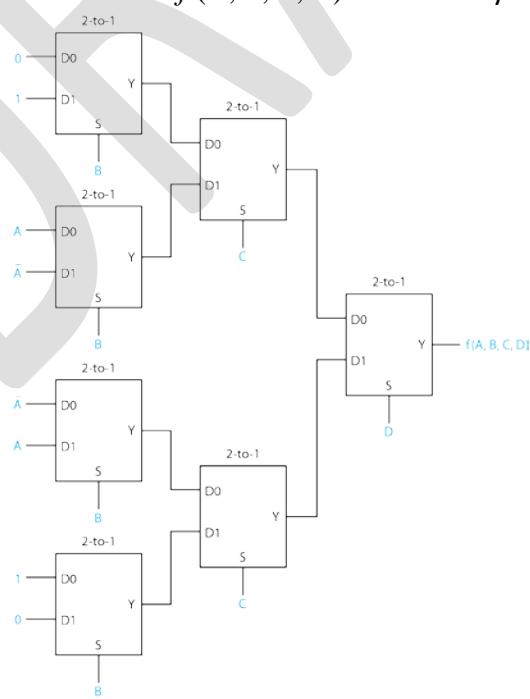
$$(b) f(b,c,d) = \bar{c} + b$$

$$(c) f(a,b,c,d) = \prod M(0,1,2,3,6,7,8,9,12,14,15)$$

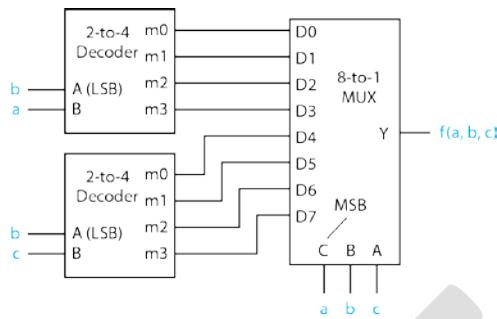
3.63 Find the minterm list of the function $f(A,B,C,D)$ realized by the circuit below.



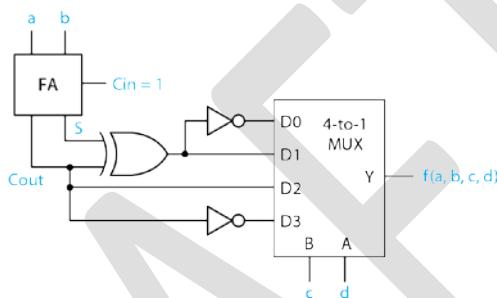
3.64 Find the minterms of the function $f(A,B,C,D)$ realized by the following circuit.



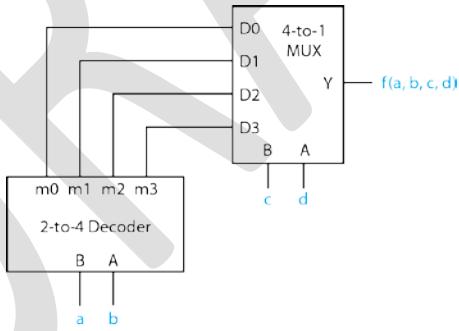
3.65 Find the minterms of the function realized by the circuit below.



3.66 Determine the function realized by the circuit below in minterm list form. Consider B to be the MSB for the multiplexer.



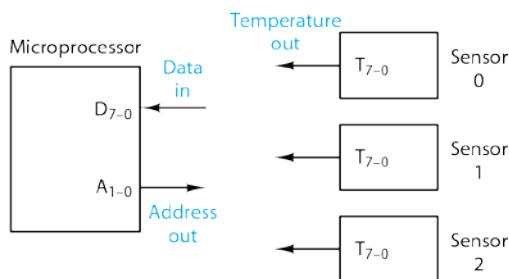
3.67 Given the circuit below, derive the output $f(a, b, c, d)$ in minterm list form. Consider B to be the MSB for both the decoder and the multiplexer. Assume positive logic (active-high inputs and outputs).



3.68 Design a full adder module with data inputs A and B , carry input C_{in} , sum output S , and carry output C_{out} .

- (a) Use a 3-to-8 decoder and NAND gates
- (b) Use a four-input, 2-bit multiplexer

3.69 Three temperature sensors, shown below, indicate measured temperature with 8-bit binary values on their output lines, T_7 to T_0 . Show with a block diagram how to use multiplexer modules to allow an 8-bit microprocessor to read any one of these sensors using its data input lines, D_7 to D_0 , by issuing a 2-bit address, A_1A_0 .



3.70 Design a 2-bit adder circuit using a two-level NAND gate circuit for each output. The inputs are the 2-bit binary numbers a_1a_0 and b_1b_0 . The outputs are the 2-bit binary sum s_1s_0 and the carry output c_1 .

3.71 Design a 16-bit ripple-carry adder using 2-bit adder modules like the one designed in Problem 3.70. Compute add time in terms of gate delay, t_g . Compare the add time to that of a 16-bit ripple-carry adder.

3.72 Using only half-adders, design a circuit that will add 3 bits, x_i, y_i, z_i , together, producing carry and sum bits c_i, s_i as shown in the following table.

x_i	y_i	z_i	c_i	s_i
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

3.73 Design a 4-bit full-adder using carry look-ahead, rather than ripple carry.

3.74 Design a 1-bit full-subtracter module, using only NOR gates, and then construct a 4-bit subtracter using only these modules.

3.75 Describe the overflow condition as applied to two's complement addition and subtraction.

3.76 Design an overflow detection circuit for Fig. 3.56, assuming that the unit is to be used to add and subtract numbers in a 4-bit two's complement number system.

3.77 The four-bit adder module described in Fig. 3.54 is faster than a four-bit ripple-carry adder because the carry for each stage is calculated from all the inputs, rather than being propagated through each stage. Find the equation for the four-bit adder internal carry C_2 in terms of only the $A(i)$, $B(i)$, and C_0 inputs. Using $A2A1$, $B2B1$, and $C0$

as the numbers being added together to determine the carry (with A_2 and B_2 as MSBs), find the combinations of $(A_2, A_1, B_2, B_1, C_0)$ that result in $C_2 = 1$. (Find the minterms for C_2 .)

3.78 Design a BCD adder that adds two unsigned BCD digits and produces a BCD result and a carry output. Implement your design using NAND gates.

3.79 Write a behavioral Verilog and/or VHDL model of a two-digit BCD adder.

3.80 Design a 3-bit magnitude comparator with inputs $A = (a_2 a_1 a_0)_2$ and $B = (b_2 b_1 b_0)_2$ and with three outputs: $EQ(A = B)$, $GT(A > B)$, and $LT(A < B)$. Implement your design with NAND and/or NOR gates.

3.81 Write a Verilog and/or VHDL model of a 3-bit magnitude comparator as above.

3.82 With appropriate gating and *one* 4-bit magnitude comparator, design a circuit that compares two 5-bit binary numbers $A = (a_4 \dots a_0)$ and $B = (b_4 \dots b_0)$ with $f_3 = 1$ when $A > B$, $f_2 = 1$ when $A = B$, and $f_1 = 1$ when $A < B$. (*Hint:* Use the cascade inputs to compare the two most significant digits.)

3.83 Design a logic circuit to compare three 4-bit numbers $X = (x_3 x_2 x_1 x_0)_2$, $Y = (y_3 y_2 y_1 y_0)_2$, and $Z = (z_3 z_2 z_1 z_0)_2$ that has the functionality described in the table below. Use four-bit magnitude comparators and logic gates to realize the circuit.

Condition	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
$X > Y > Z$	1	0	0	0	0	0	0	0
$X > Z > Y$	0	1	0	0	0	0	0	0
$Y > X > Z$	0	0	1	0	0	0	0	0
$Y > Z > X$	0	0	0	1	0	0	0	0
$Z > X > Y$	0	0	0	0	1	0	0	0
$Z > Y > X$	0	0	0	0	0	1	0	0
$X = Y = Z$	0	0	0	0	0	0	1	0
Any other case	0	0	0	0	0	0	0	1

3.84 Write a Verilog and/or VHDL behavioral model of the comparator described in problem 3.83.

3.85 Design a logic circuit that multiplies two 2-bit numbers, $(a_1 a_0)_2$ and $(b_1 b_0)_2$, using only NAND gates. The product should be a 4-bit number $(p_3 p_2 p_1 p_0)_2$.

3.86 Design a logic circuit that multiplies two 4-bit numbers, $(a_3a_2a_1a_0)_2$ and $(b_3b_2b_1b_0)_2$, using only AND gates and half- and full-adder modules. The product should be an 8-bit number $(p_7p_6p_5p_4p_3p_2p_1p_0)_2$.

3.87 Design a logic circuit to test the equality of two 4-bit numbers, $(a_3a_2a_1a_0)_2$ and $(b_3b_2b_1b_0)_2$. The circuit output, e , should be 0 when the numbers are equal and 1 when they are not. Draw a circuit diagram.

3.88 Write a Verilog and/or VHDL model for problem 3.87.

3.89 Design a decoder that converts unsigned 4-bit binary numbers to two BCD digits that represent the decimal value of the binary number. Describe your design as a Verilog and/or VHDL module.

3.90 Design a decoder that converts unsigned 4-bit binary numbers to codes for displaying the decimal value of the binary number on two seven-segment displays. Leading zeros should be displayed as a blank display. Describe your design as Verilog and/or VHDL modules.

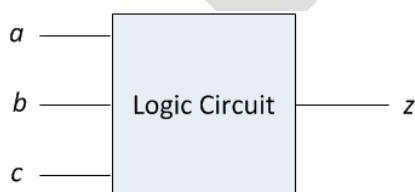
3.91 Design a decoder that converts 4-bit 2's complement numbers to codes for displaying the sign and decimal value of the binary number on two seven-segment displays. Describe your design as Verilog and/or VHDL modules.

3.92 Two-to-1 multiplexers are included in some cell libraries as primitive components. Use two-to-1 multiplexers plus AND, OR, and NOT gates, if needed, to realize the following logic functions. Use as few components as possible.

- | | |
|------------------|--------------------------------------|
| (a) AB | (e) $AC' + BC$ |
| (b) AB' | (f) $\sum m(0,2,3,5)$ |
| (c) $A + B$ | (g) $ab'c' + a'c + bc$ |
| (d) $A \oplus B$ | (h) $\sum m(0,1,2,3,4,5,9,13,14,15)$ |

3.93 Derive the truth table for the logic circuit described below.

- (a) Assuming positive logic.
- (b) Assuming negative logic.



V_a	V_b	V_c	V_z
0	0	0	0
0	0	4.9	4.9
0	4.9	0	0
0	4.9	4.9	4.9
4.9	0	0	0
4.9	0	4.9	0
4.9	4.9	0	4.9
4.9	4.9	4.9	4.9

3.94 Design Verilog and/or VHDL structural and datflow models for the circuit given in Problem 3.9.

3.95 Design a Verilog and/or VHDL model for the XOR2x4 module used in the example on page 92.

3.96 Compute the propagation delay for the circuits in Figure 3.18.

3.97 Write a Verilog and/or VHDL model for the multiplexer described in Figure 3.46.

3.98 Given the seven-bit ASCII code defined in Table 1.10 in Chapter 1, design the following circuits assuming that you have only XOR2 gates available as logic elements. Draw schematics of each design.

- (a) A parity-bit generator for an odd-parity encoding.
- (b) An error detector for detecting single-bit errors assuming odd parity.

3.99 Repeat problem 3.95 for an even-parity encoding.

3.100 Given Hamming Code 1 defined in Chapter 1, on page 43. Write a Verilog and/or VHDL behavioral model of an encoder for the code.

3.101 Write a Verilog and/or VHDL model of a decoder for Hamming Code 1. The decoder should detect and correct single errors.

3.102 Repeat problem 3.100 for Hamming Code 2.

3.103 Repeat problem 3.101 for Hamming Code 2. Additionally, the decoder should detect double errors.

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, and Irwin

Chapter 4 Introduction to Sequential Circuits

(7/31/2016)

Table of Contents

4.0 Learning Objectives.....	1
4.1 Models and Classes of Sequential Circuits	2
4.1.1 Finite State Machines.....	2
4.1.2 State Diagrams and State Tables	4
4.1.3 Algorithmic State Machines.....	9
4.2 Memory Devices	12
4.2.1 Latches	14
4.2.2 Flip-flops.....	31
4.2.3 Latch and Flip-flop summary	45
4.3 Registers.....	47
4.4 Shift Registers.....	53
4.5 Counters.....	59
4.5.1 Synchronous Binary Counters	59
4.5.2 Asynchronous Binary Counters.....	63
4.5.3 Modulo-<i>N</i> Counters.....	65
4.5.4 Ring and Twisted-Ring Counters	71
4.6 Summative Design Examples.....	84
4.6.1 Register File Design	84
4.6.2 Multiphase Clock Design	86
4.6.3 Digital Timer Design	89
4.6.4 Programmable Baud Rate Generator.....	94
4.7 Summary and Review Questions	98
4.8 Collaboration Exercises	99
4.9 Problems	100

This page intentionally left blank.

DRAFT

Chapter 4 Introduction to Sequential Circuits

(7/13/2016)

The basic concept of a sequential device is an important one that is fundamental in the design of digital systems. Recall that in combinational logic networks the output is a function of only the present input. In sharp contrast, the output of a sequential device depends not only on the present input but on previous inputs as well. This history of a sequential circuit's input is retained through the use of storage devices, that is, *memory*. It is the use of memory which adds a new dimension to logic design by providing the capability to solve numerous problems that cannot be handled by combinational logic alone.

The sequential concept is not restricted to digital systems. For example, consider the operation of an elevator in a four-story building. The elevator acts as a sequential device because its actions are determined by input signals from its control panels (both on board and on each floor) and its present position at floor 1, 2, 3, or 4. The elevator must in some way "remember" its present position in order to determine its next floor transition. Therefore we define the *present state* of the elevator as a description of its present floor position, including a history of its past floor transitions. For example, the elevator may be "at floor 3 and going up." This present state must be differentiated from "at floor 3 and going down." The *next state* (and hence the next floor position) of the elevator is determined by its present state and its *input*, which consists of the condition of the control buttons on the control panels located in the elevator and stationed on each floor. If the elevator is "at floor 3 and going down," it will respond to a floor 2 request to go down, but ignore a floor 2 request to go up! Once the next state is determined, a *state transition* is ordered by sending a command, an *output* signal, to the pulley motor, which drives the elevator to a new floor. The concepts of present state, next state, input, output, and state transition are fundamental in the study of sequential logic circuits.

Another simple example of a sequential device, and one that finds wide application in digital systems, is a counter, in which the present state is simply the current value of the count. This device can be employed to perform such functions as totaling the number of cars entering a parking lot, counting the number of bits transmitted or received on a communication link, or keeping track of certain functions being performed within a large computer system. Counters are covered in detail later in this chapter.

4.0 Learning Objectives

In this chapter we introduce the basic sequential circuit model, and then describe the design and operation of a number of common memory elements, including latches and flip-flops. Students completing this chapter will have gained the following knowledge and skills.

1. Understanding of sequential behavior, in which the output of a circuit is a function of a sequence of inputs.
2. Understanding of memory elements used in sequential circuits, including latches and flip flops.
3. Ability to design some simple sequential logic circuit modules, such as registers, counters and shift registers.

4.1 Models and Classes of Sequential Circuits

A sequential logic circuit can be described using models of various types. Such models aid in the design and analysis of such circuits by enabling the behavior of the circuit to be defined, either mathematically or graphically. In some cases, models can be verified through simulation or mathematical proofs, to ensure that the intended behavior is correct. Once the behavior has been verified, then a circuit can be designed and realized to implement that behavior. In this section, several modeling techniques for sequential circuits are presented, including finite state machines, state diagrams, state tables, and algorithmic state machines.

4.1.1 Finite State Machines

In our study of combinational logic networks we found that we could represent these circuits as shown in Fig. 4.1a. The mathematical relationship that describes this network is

$$z_i = f_i(x_1, x_2, \dots, x_n), \quad i = 1, \dots, m \quad (4.1)$$

This equation simply states that the output is a function of only the present input. All the signals in Eq. 4.1 are assumed to be either of the two values, 0 or 1.

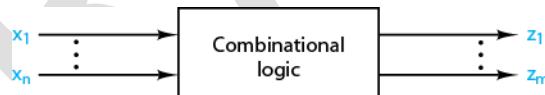


Figure 4.1: Combinational logic circuit model

General models for a sequential logic circuit, as shown in Fig. 4.2, contain both combinational logic and memory elements. The n -tuple (x_1, \dots, x_n) and the m -tuple (z_1, \dots, z_m) will be referred to as the *input* and *output* of the circuit, respectively. Each memory element stores one bit of an r -bit binary code that represents the *state* of the circuit. The r -tuple (y_1, \dots, y_r) represents the *present state* of the circuit. The r -tuple (Y_1, \dots, Y_r) represents the *next state* of the circuit, which is the state that the circuit will enter the next time it changes. State variable y_i is stored in memory element M_i , while Y_i indicates the next state of M_i . Since there is a finite number of possible combinations of state variable values, sequential logic circuits are often referred to as *finite state machines*. The relationships that exist among the state and input variables may be expressed mathematically as

$$Y_i = h_i(x_1, \dots, x_n, y_1, \dots, y_r), \quad i = 1, \dots, r \quad (4.2)$$

where the h_i are Boolean functions.

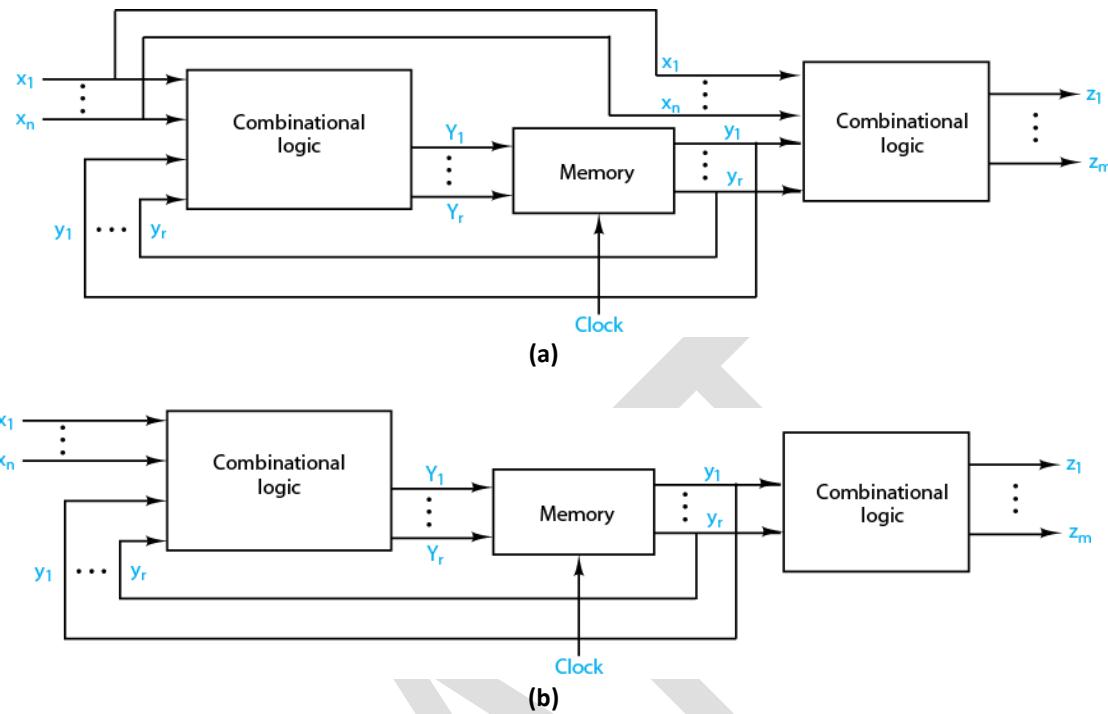


Figure 4.2: Sequential logic circuit models.

(a) Mealy model. (b) Moore model.

Figure 4.2 illustrates two options for determining the circuit outputs. The model shown in Figure 4.2a is called the *Mealy model* [1], in which the output is a function of both the inputs and the present state. The relationships that exist among the state, input and output variables of a Mealy model sequential circuit may be expressed mathematically as

$$z_i = g_i(x_1, \dots, x_n, y_1, \dots, y_r), \quad i = 1, \dots, m \quad (4.3)$$

where the g_i are Boolean functions.

The model shown in Figure 4.2b is called the *Moore model* [2], and it is distinguished from the Mealy by the output being a function only of the present state. The relationships that exist among the state and output variables of a Moore model sequential circuit may be expressed mathematically as

$$z_i = g_i(y_1, \dots, y_r), \quad i = 1, \dots, m \quad (4.4)$$

where the g_i are Boolean functions.

Equations 4.2 - 4.4 may be written in vector notation as

$$Y = h(x, y) \quad (4.5)$$

$$z = g(x, y) \quad (\text{Mealy model}) \quad (4.6)$$

$$\text{or } z = g(y) \quad (\text{Moore model}) \quad (4.7)$$

where

$$z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_r \end{bmatrix} \quad (4.8)$$

Note that z_i , x_i , y_i , and Y_i are all binary variables (their values are logic 0 or logic 1).

All the vectors in Eq. 4.8 are time dependent; we shall adopt the convention that vector y has the value $y(t)$ at time t . Occasionally, we shall examine a signal $y(t)$ at evenly spaced points in time. If $t_k = k\Delta t$ (k is an integer), then

$$y(t_k) = y(k\Delta t) \quad (4.9)$$

where Δt is some fixed increment of time. For convenience, we often suppress the time variable.

The memory devices in the block diagrams of Fig. 4.2 may be of several types. The most common memories are semiconductor flip-flops, but other possibilities include magnetic devices, delay lines, mechanical relays, rotation switches, and many others. Several semiconductor memory devices will be examined in the following sections.

The operation of a sequential logic circuit can further be classified as being either synchronous or asynchronous. A *synchronous* sequential circuit is one in which all state changes are synchronized to a common signal, called a *clock*. In general, a clock signal is simply a sequence of pulses, most often occurring at a fixed rate, or frequency. These clock pulses enable the memory elements of the circuit to change states; otherwise, the memory elements hold their states.

Some applications require the use of sequential circuits that are not synchronized in any way with a clock signal. These are referred to as *asynchronous* sequential circuits. Asynchronous circuits require special attention since there is no clock signal to provide common timing information to the circuit elements. Hence, asynchronous circuits respond immediately to any change of input, rather than responding to the inputs present during a clock pulse. Input changes can be momentary pulses, or simple changes in logic level. The absence of a clock signal also means that memory element transitions must be initiated by some other means. Therefore, precautions must be taken to avoid timing problems. The various facets and ramifications of asynchronous sequential circuit design will be discussed in Chapter 6.

4.1.2 State Diagrams and State Tables

The logic equations 4.2 - 4.4 and vector equations 4.5 – 4.7 completely define the behavior of the sequential circuits modeled in Figs. 4.2a and 4.2b for a given set of memory devices. However, the description, although complete, does not present a very lucid picture of the relationships that exist among the pertinent variables. The functional relationship that exists among the input, output, present state, and next state is very vividly illustrated by either the state diagram or the state table. The *state diagram* is a graphical representation of the behavior of a sequential circuit in which the states of the circuit are represented by circles and the state transitions (the transfer from the present state y to the next state Y) are shown by arrows. Each arrow is labeled with the input x and for Mealy models, the resulting circuit

output z , as shown in Fig. 4.3a. In the case of Moore models, the circuit output z is listed with the state in each circle, as shown in Fig. 4.3c, since the output is a function only of the state.

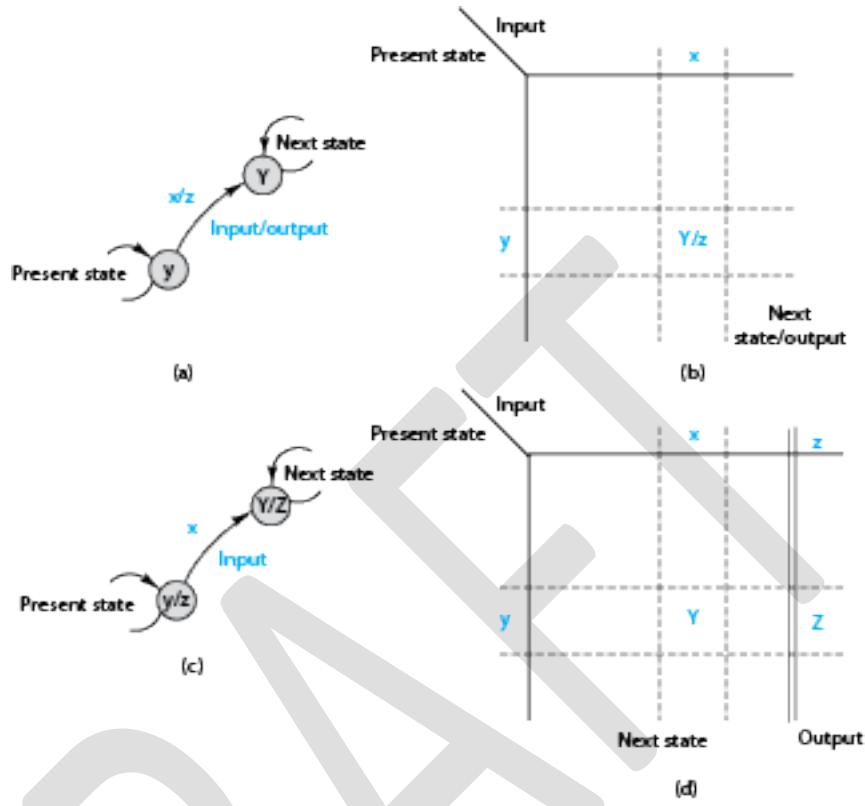


Figure 4.3: State tables and diagrams.

- (a) Mealy model state diagram. (b) Mealy model state table.
- (c) Moore model state diagram. (d) Moore model state table.

Figures 4.3b and 4.3d illustrate the *state table* representation. All circuit input vectors x are listed across the top, while all state vectors y are listed down the left side. For a Mealy model, entries in the table are the next state Y and the output z , as shown in Fig. 4.3b. The table is read as follows: For an input x with the sequential circuit in state y , the circuit will proceed to the next state Y with an output z . The Mealy model is called a *transition-assigned circuit* because the circuit output is associated with the state transitions, that is, the arcs in the state diagram. In other words, the circuit outputs are functions of the present state and inputs, as given by Eq. 4.3.

For a Moore model, the state table is in a slightly different format, as shown in Fig. 4.3d. The output may be removed from the next state entries in the state table since the output is independent of the input, i.e. the output z is a function only of the present state; therefore a separate column of outputs is shown. It is important to remember that these outputs belong to the present state and not to the next one.

In practice, the state diagrams and tables are usually labeled using symbols rather than vectors. For example, consider a sequential circuit with two present-state variables y_1 and y_2 .

Then

$$y = [y_1, y_2]$$

Therefore, the vector y can have any of the four possible values:

$$y = [00] = A, \quad y = [10] = C$$

(4.10)

$$y = [01] = B, \quad y = [11] = D$$

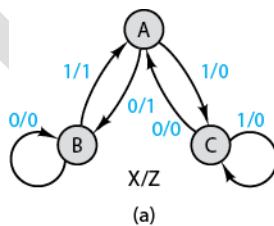
Thus, the sequential circuit has only four possible states, which may be labeled A, B, C , and D . In general, if r represents the number of memory devices in a circuit with N_s states, these two quantities are related by the expression

$$2^{r-1} < N_s \leq 2^r \quad (4.11)$$

However, there are other methods for representing states with binary codes, which will be discussed later, such as the “one-hot” state assignment, which uses one memory element per state.

Consider the Mealy model sequential circuit whose state diagram and state table are shown in Figs. 4.4a and 4.4b. This circuit has one input variable x , two state variables y_1 and y_2 , and one output variable z .

Inputs:	$x = 0$
	$x = 1$
States:	$[y_1, y_2] = [00] \equiv A$
	$[y_1, y_2] = [01] \equiv B$
	$[y_1, y_2] = [10] \equiv C$
Outputs:	$z = 0$
	$z = 1$



Present state	Input x		Next state/output
	0	1	
A	B/1	C/0	
B	B/0	A/1	
C	A/0	C/0	

(b)

Figure 4.4: Example Mealy model sequential circuit.

(a) State table. (b) State diagram.

In this case, the circuit has only three states, so the combination $[y_1, y_2] = [10] \equiv D$ is not used, and it is assumed that this combination cannot occur during normal operation. The relationship between the input, state, and output is shown by the following example.

EXAMPLE 4.1. Let us determine the output response of the sequential circuit defined in Fig. 4.4 to the input sequence $x = 011010$, where the values in the sequence are applied prior to times $T_0, T_1, T_2, T_3, T_4, T_5$, respectively.

Let us assume that the circuit is initially in state A . At time 0 an input of $x=0$ is applied. From either the state diagram or the state table, we can read that the output is $z=1$ and the next state will be B , with the state changing on the falling edge of the clock pulse at time T_0 . If an input of $x=1$ is now applied, with the circuit in state B , we can see from the state diagram that the output changes to $z=1$ immediately, and the state will become A at time T_1 . Continuing for the remaining values in the input sequence, applied prior to times T_2, T_3 , etc., the circuit will behave as illustrated in the diagram shown in Figure 4.5.

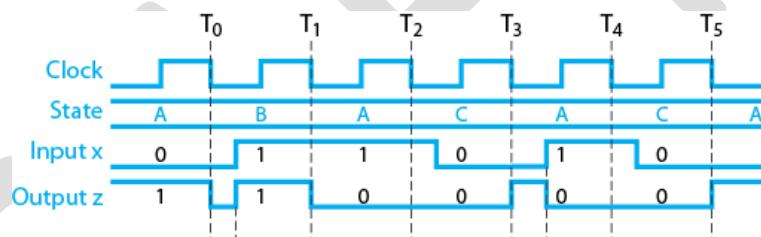


Figure 4.5: Mealy model timing diagram.

Hence, this input sequence applied to the machine in state A causes the output sequence

$$z = 110000$$

and leaves the circuit in final state A .

Figure 4.5 illustrates the actual circuit timing for the given input sequence. In this diagram it is assumed that the state changes on the high-to-low transition of the clock. Note that the output z can change any time either the input or the state changes, since z is a function of both. This gives rise to two unexpected output changes shown in the timing diagram. At time T_0 , z drops to 0 when the state changes to B , and then goes back to 1 when input x changes to 1. A similar event occurs at time T_3 . Hence, we must be careful to sample the output of a Mealy model circuit only when the circuit has stabilized after an input change.

A similar analysis can be applied to a Moore model circuit, keeping in mind that the output is a function only of the current state of the circuit. Consider the synchronous sequential circuit whose state diagram and state table are given in Figs. 4.6a and 4.6b. Note that the circuit outputs are listed with the states in the circles of the state diagram, and a separate column of output values is shown in the state table in Fig. 4.6b.

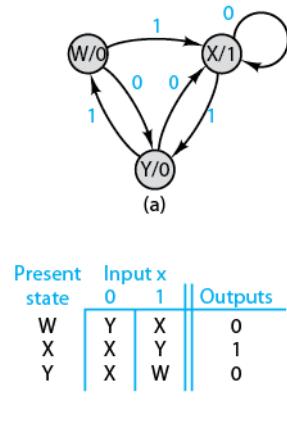


Figure 4.6: Synchronous sequential circuit - Moore model.
a State diagram. b State table.

EXAMPLE 4.2. Let us determine the output response of the Moore model sequential circuit in Fig. 4.6 to the input sequence $x = 011010$.

The output is always identified by the present state from either the state diagram or the state table. Figure 4.7 illustrates the actual circuit timing for the given input sequence. As in the previous example, all state changes occur on the high-to-low transition of the clock. Note that the output changes only when the state changes, and is not otherwise affected by input changes.

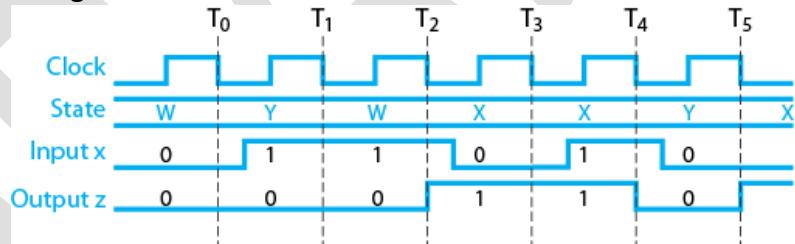


Figure 4.7: Moore model timing diagram.

In Fig. 4.7, note that both state changes and output changes in a Moore model circuit are synchronized to the clock, since output z is a function of the state only and can therefore change only when the state changes. Consequently, the output remains stable during any input changes, unlike the outputs of the Mealy model circuit illustrated previously in Fig. 4.5. Therefore, the outputs of a Moore model circuit typically exhibit better behavior than those of a Mealy model circuit; that is, input changes do not result in unwanted glitches in the outputs.

The primary advantage of using a Mealy versus a Moore model for a sequential circuit design is that, since the outputs of a Mealy model are functions of both the inputs and the state, the designer has more flexibility in designing output and state transition functions, and thus fewer states will be needed than in an equivalent Moore model circuit, where the outputs are functions of only the state variables.

In the sections that follow, examples of both Mealy and Moore model circuits will be presented.

4.1.3 Algorithmic State Machines

A variation of the state diagram that is useful when designing system control units and other finite-state machines is the *algorithmic state machine* (ASM) diagram. An *algorithm* is a well-defined sequence of steps that produces a desired sequence of actions in response to a given sequence of inputs. The ASM diagram is a convenient tool for expressing algorithms to be realized by sequential logic circuits, much as a flow chart might be used to describe a software algorithm to be programmed on a computer. ASM diagrams are constructed of three elements:

State box: Represents one state of the circuit, and is therefore equivalent to one node of a state diagram. The state name is listed in the box and, for Moore circuits, the outputs to be produced, that is, the actions to be performed while in that state. A state box always has a single entry point and a single exit point, as illustrated in Fig. 4.8a.

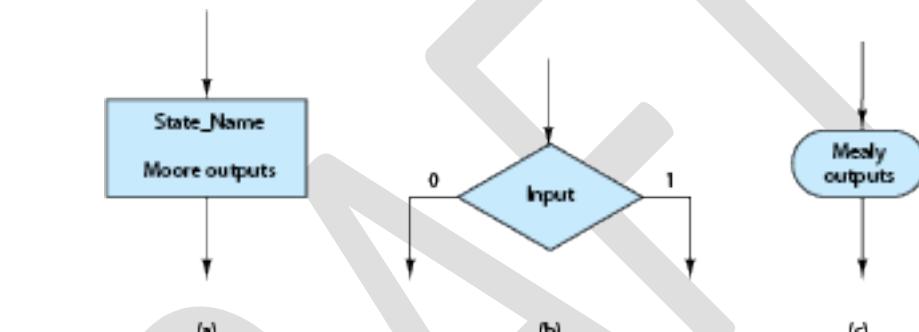


Figure 4.8: Elements of an ASM diagram. (a) State box. (b) Decision box. (c) Conditional output box.

Decision box: Represents a state transition decision based on a test of one circuit input. A decision box, as illustrated in Fig. 4.8b, lists the input to be tested and has a single entry point and two exit points, one for the input equal to 0 and one for the input equal to 1. A separate decision box is used for each input test.

Conditional output box: Specifies outputs associated with state transitions for a given input in a Mealy circuit. A conditional output box is placed in the path between a decision box and a state box and has a single entry and exit point, as shown in Fig. 4.8c.

ASM diagrams can be used to develop both Mealy and Moore model circuits. The ASM diagram and corresponding state diagram for a Mealy circuit are presented in Figs. 4.9a and 4.9b, respectively. In the ASM diagram, note that output z is specified in conditional output boxes, one for each state and input combination. This corresponds to associating the output with the arcs of the state diagram.

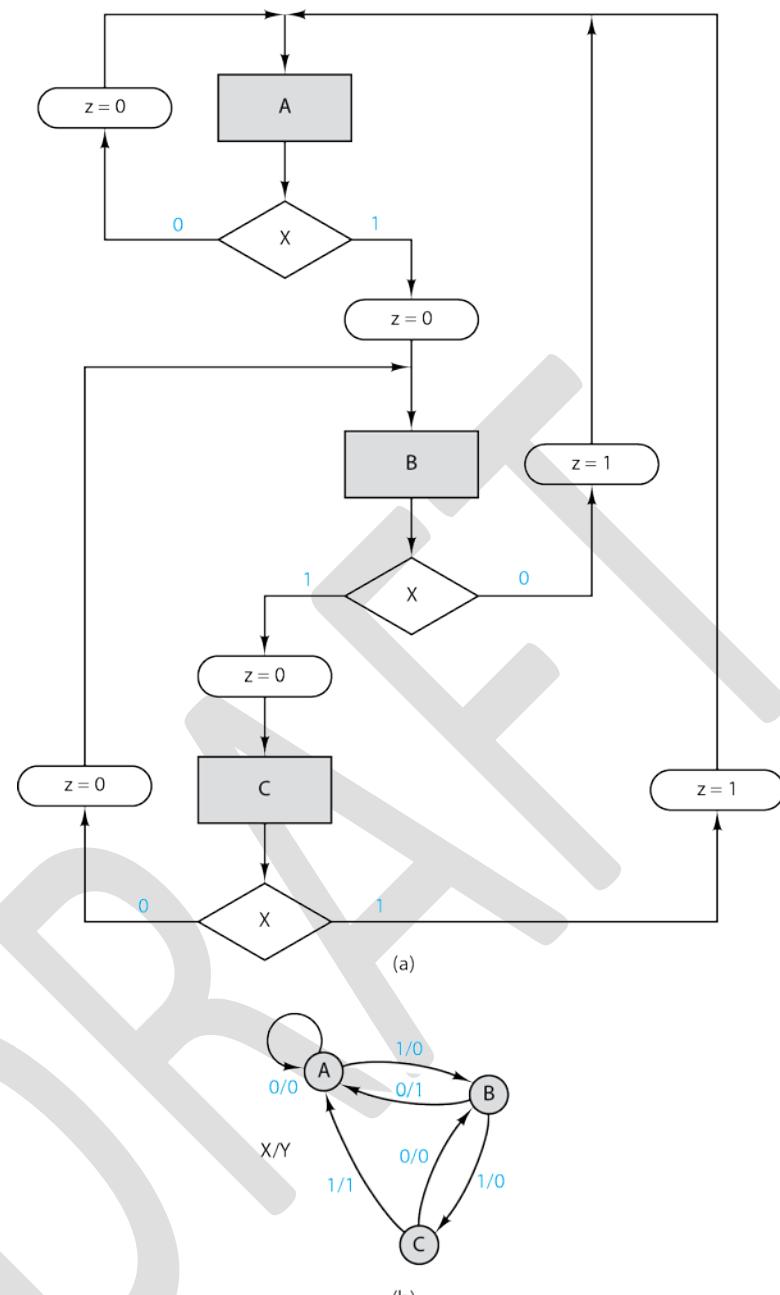


Figure 4.9: ASM Representation of a Mealy model sequential circuit. a ASM diagram. b Equivalent state diagram.

The ASM diagram and corresponding state diagram for a Moore circuit are presented in Figs. 4.10a and 4.10b, respectively. Note that there are no conditional output boxes in the ASM diagram. The outputs of a Moore circuit are functions of only the state variables and are therefore specified within the state boxes in the ASM diagram and within the nodes of the state diagram.¹

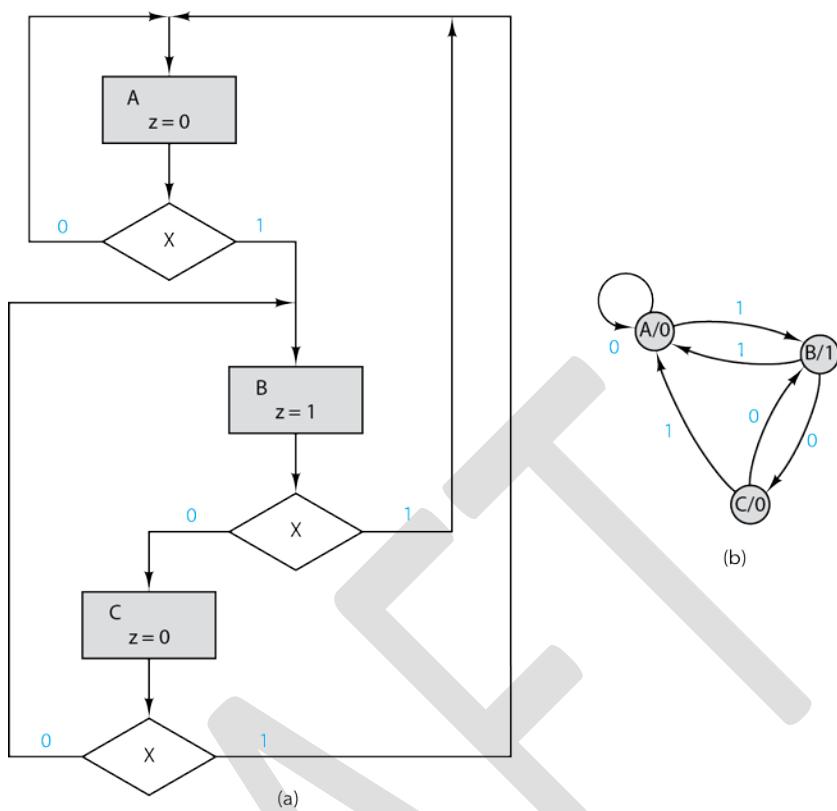


Figure 4.10: ASM representation of a Moore model sequential circuit.
a ASM diagram.
b Equivalent state diagram.

4.2 Memory Devices

As indicated earlier, memory is an integral part of a sequential machine. Our discussion will be concerned primarily with the external characteristics of various memory devices and not their detailed internal functions. In other words, our analysis will be confined to the use of these elements in the design of digital systems.

In switching circuit applications, memory elements are generally *bistable* electronic circuits; that is, they exist indefinitely in one of two possible stable states, 0 and 1. Binary data are stored in a memory element by forcing the element into the 0 state to store a 0 and into the 1 state to store a 1. The output Q of the circuit indicates the present ("quiescent") state of the memory. Each memory circuit has one or more *excitation inputs*, so called because they are used to "excite" or drive the circuit into a desired state. Different memory devices are typically named in accordance with their particular excitation inputs, which differ from device to device.

The two memory element types most commonly used in switching circuits are latches and flip-flops. A *latch* is a memory element whose excitation inputs control the state of the device. A latch excitation input that forces the state of the device to 1 is called a *set input*. An excitation input that forces the device state to 0 is called a *reset input*. A latch having both set and reset excitation inputs is called a *set-reset latch*. The operation of a set-reset latch is illustrated in Fig. 4.11a. If a latch has a single excitation input whose value represents the desired state of the latch, the latch is called a *D (data) latch*. Latch operation will be described in the next section.

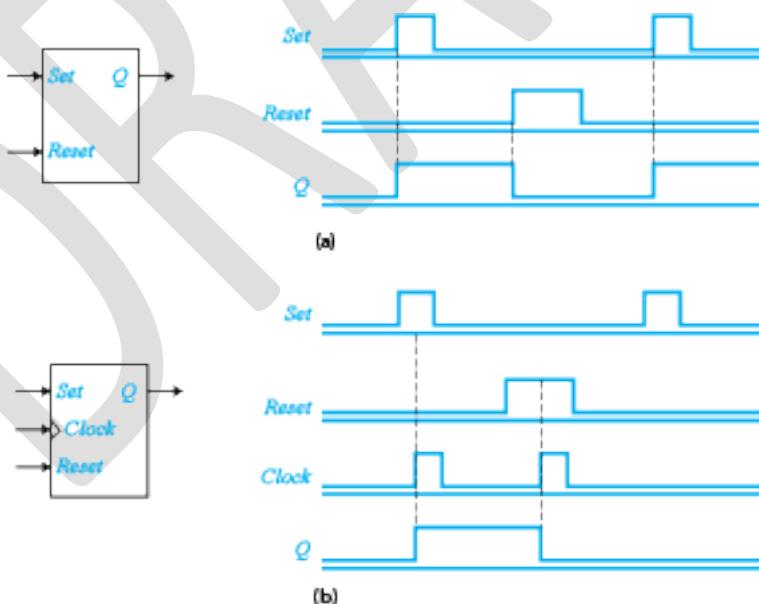


Figure 4.11: Latch and flip-flop timing.

(a) Latch responds immediately to excitation. (b) Flip-flop responds only on a clock signal.

A *flip-flop* differs from a latch in that it has a control input called a *clock*. The clock input

carries a signal to the flip-flop that triggers a state change in accordance with the flip-flop's excitation inputs. The operation of a set-reset flip-flop is illustrated in Fig. 4.11b. In both latches and flip-flops, the next state is determined by the excitation inputs. However, as illustrated in Fig. 4.11, an excitation input change causes a latch to change state, while a flip-flop waits for a clock signal before changing states. The state of a flip-flop is determined by its excitation values at the time the clock signal occurs. In this manner, multiple flip-flops in a sequential circuit can be synchronized to a common clock signal so that they all change states together. As in the case of latches, flip-flops can have different types of excitation inputs, with the flip-flop type named after them.

Formal procedures for designing latch and flip-flop circuits will be presented in Chapter 5. In this chapter, several generic latch and flip-flop circuits will be described. Many different latch and flip-flop circuits are available, implemented with different technologies. Some of these are commercially available as integrated circuits (ICs). Both generic and technology-specific latches and flip flops are available in computer-aided design libraries, from which they can be selected and incorporated into custom or semicustom integrated circuit designs. We have selected examples to give the reader an introduction to the various types of devices and device features available. You must understand these features so that you can choose a proper device for each situation as you design sequential circuits.

4.2.1 Latches

4.2.1.1 Set-Reset Latch

Using Feedback to Create Simple Latches

Fig. 4.12a presents a simple example of a memory element. Constructed from two inverters, the state of the memory element is represented by the value of the line labeled Q , the output of the lower inverter. The value of Q is continuously regenerated by feeding it back to itself through the two inverters. Examining the inverter outputs, we see that:

$$Q = \overline{Q_B} \text{ and } Q_B = \overline{Q} \quad (4.12)$$

and therefore the feedback path gives:

$$Q = \overline{Q_B} = \overline{\overline{Q}} = Q$$

This results in a stable condition that maintains the value of Q at its current level. Thus, if $Q = 1$, the feedback forces it to remain 1, and if $Q = 0$ the feedback forces it to remain 0. We say that this memory element is in a "hold" condition, because the state does not change, i.e. it holds (stores) the value of Q in the memory element. Unfortunately, there is no simple mechanism for changing the state of this memory element, and thus the state is permanent. It simply remains in whatever state was assumed when the element was powered on.

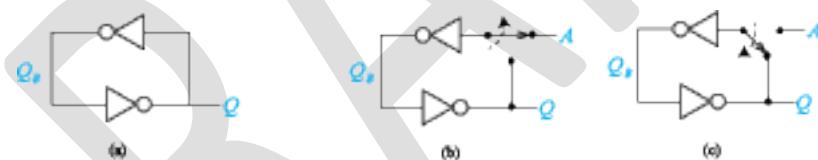


Figure 4.12: Simple memory circuit.

(a) Memory cell state Q is fixed. (b) Excitation input A forces $Q = A$. (c) Feedback loop holds the state of Q .

In Figs. 4.12b and 4.12c, a switch has been added to the memory circuit to provide a mechanism for changing its state. With the switch in the up position, as shown in Fig. 4.12b, the feedback loop is broken and excitation signal A is connected to the input of the upper inverter, making $Q_B = \overline{A}$ and $Q = A$. Moving the switch to the down position, as shown in Fig. 4.12c, disconnects the input and closes the feedback loop, recreating the hold condition of eq. 4.12. The value of Q is thus held, or *latched*, at the value set by input A at the instant the switch changed positions.

Instead of adding a switch to the memory element, practical latch circuits can be constructed from simple logic gates. In Fig. 4.13a the upper NOT gate of the memory circuit of Fig. 4.12a has been replaced by a NOR gate. One input of the NOR gate is connected to excitation signal S , while the other gate input is used to close the feedback loop. The resulting logic equations at the gate outputs are:

$$\begin{aligned} Q_B &= \overline{S + Q} \\ Q &= \overline{Q_B} = S + Q \end{aligned} \tag{4.13}$$

From eq. 4.13 we can see that whenever $S = 0$, the output of the NOR gate is simply $Q_B = \overline{0+Q} = \overline{Q}$ and the output of the inverter is $Q = 0 + Q = Q$; this is equivalent to the hold condition defined by eq. 4.12. When $S = 1$, the NOR gate output becomes $Q_B = \overline{1+Q} = \overline{1} = 0$, and the inverter output becomes $Q = \overline{Q_B} = \overline{0} = 1$. Therefore the state of the latch is set to 1. However, once the latch has been set to state 1, there is no way to reset it to 0, since $Q = 1$ at the input of the NOR gate forces its output to $Q_B = 0$, independent of the value of S . The Q output will remain set, that is at logic 1, permanently, ignoring further changes in S . Basically, if the latch is initially in the reset state ($Q = 0$) a pulse on S changes the latch to the set state ($Q = 1$), with no further changes possible. Hence the circuit of Fig. 4.12a is called a *set latch*, and the S input is referred to as the *Set* input.

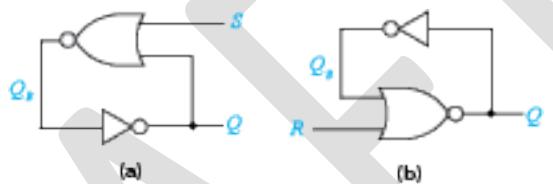


Figure 4.13: Set and Reset latches.

(a) Set latch: $S = 1$ sets Q to 1. (b) Reset latch: $R = 1$ resets Q to 0.

By symmetry, replacing the lower NOT gate of the memory circuit in Fig. 4.12a with a NOR gate results in a *reset latch*, illustrated in Fig. 4.13b, with one input of the NOR gate connected to excitation signal R , which stands for *Reset*. The operation of the reset latch is symmetric to that of the set latch, as defined by the following equations at the gate outputs:

$$\begin{aligned} Q &= \overline{R + Q_B} \\ Q_B &= \overline{Q} = R + Q_B \end{aligned} \tag{4.14}$$

$R = 0$ produces the hold state, i.e. $Q = \overline{Q_B}$ and $Q_B = \overline{Q}$. The reader should verify that setting $R = 1$ will force $Q = 0$, which in turn forces $Q_B = 1$, putting the latch into the *reset state*. Once $Q_B = 1$, the feedback will subsequently hold $Q = 0$, inhibiting further changes as the value of input R changes. Therefore, if the latch is initially in the set state a pulse on R changes the latch to the reset state, with no further changes possible.

Set--Reset Latch: NOR Structure

Devices that stay permanently in one logic state are not very useful except in very unusual design situations. If we combine the features of both latches described previously into one circuit, we can set or reset the state of the latch as needed. Replacing both NOT gates in the circuit of Fig. 4.12a with two-input NOR gates, as shown in Fig. 4.14a, creates a *set-reset latch* (*SR latch*). We will use this latch as a building block for other circuits in this section. Turning the upper NOR gate around produces the more traditional view of the latch circuit, the

cross-coupled form of Fig. 4.14b with inputs on the left and outputs on the right. Let us adopt the logic symbol of Fig. 4.14c to represent the SR latch, where the inversion bubble indicates the Q_B output, which is effectively \bar{Q} in normal operation. The alternate logic symbol shown in Fig. 4.14d is also used in many CAD programs and books, with the complementary output labeled \bar{Q} instead of being indicated via an inversion bubble.

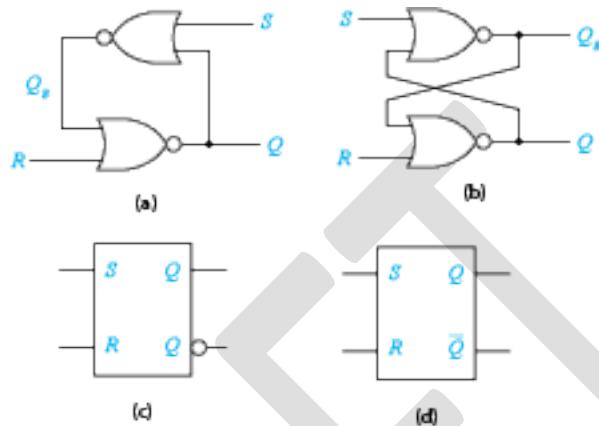


Figure 4.14: Set-reset latch (SR latch).

(a) SR latch circuit. (b) Traditional view of SR latch. (c) Logic symbol of an SR latch. (d) Alternative logic symbol of an SR latch.

The operation of the SR latch follows directly from the operations of reset and set latches, and can be studied by examining the logic expressions realized by the two NOR gates.

$$\text{Upper NOR gate: } Q_B = \overline{S + Q} \quad (4.15)$$

$$\text{Lower NOR gate: } Q = \overline{R + Q_B} \quad (4.16)$$

Consider the NOR gate outputs for all possible values of S and R.

Case 1. $S = 0, R = 0$: $Q_B = \overline{Q}$ and $Q = \overline{Q_B}$, which is the hold condition defined in eq. 4.12.

Case 2. $S = 1, R = 0$: Setting $S = 1$ forces $Q_B = 0$. Since $R = 0$, we still have $Q = \overline{Q_B}$, and therefore $Q_B = 0$ forces $Q = 1$, which is the set condition.

Case 3. $S = 0, R = 1$: Setting $R = 1$ forces $Q = 0$. Since $S = 0$, we still have $Q_B = \overline{Q}$ and therefore $Q = 0$ forces $Q_B = 1$, which is the reset condition.

In all cases, the circuit holds the new state whenever both inputs are returned to logic 0.

Problems arise if both S and R are set to 1 at the same time. From eqs. 4.15 and 4.16, we can see that placing logic 1 signals on both the R and S inputs forces both outputs, Q and \overline{Q} , to logic 0. If both S and R are returned to logic 0 at the same time, a *race condition* is created. Both NOR gate outputs change to 1, and if the delays through the gates are identical, feedback causes both gates to change back to 0 at the same instant, and the gate outputs will continue to change back and forth, resulting in an unstable oscillatory condition, often referred to as a *race condition*. In reality, one of the gates will likely be a bit

faster than the other, and the circuit will settle into one state or the other. However, the final state is almost impossible to predict. Consequently, we restrict the use of the SR latch to exclude the input combination $S = R = 1$.

The operation of any memory circuit may be examined with the help of a timing diagram. Figure 4.15a illustrates the state changes induced in the SR latch of Fig. 4.14 by applying sequences of pulses to its set and reset inputs, S and R . In this example, the waveforms for S and R have been selected to illustrate the various features of the latch. The initial state of the latch is assumed to be logic 0 ($Q = 0$). The latch responds to the first of a series of pulses on one of its inputs, but ignores subsequent ones until a pulse on the other input has intervened to change the latch to the complementary state. Note that setting $S = R = 1$ is shown as producing an unknown state.

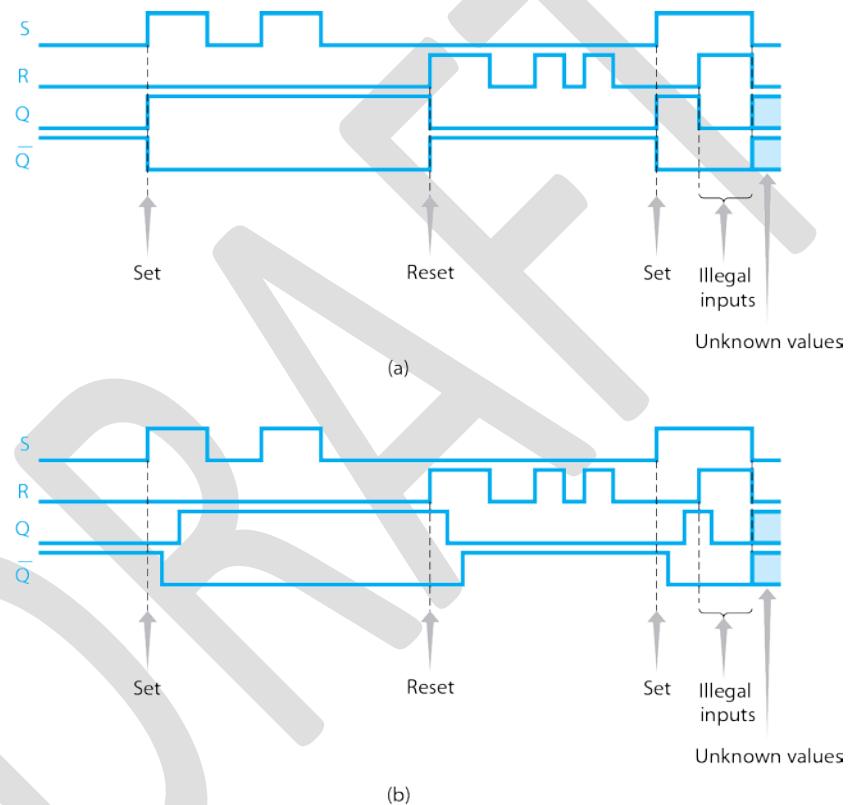


Figure 4.15: Set-reset latch timing diagram. **(a)** Ideal (zero-gate-delay) case. **(b)** Actual timing with non-zero gate delays.

The timing diagram of Fig. 4.15a represents an ideal situation in which gate propagation delays are assumed to be 0. In reality, every circuit output requires a nonzero amount of time to respond to changes on its inputs, as illustrated in Fig. 4.15b. These response delays are specified by parameters t_{PLH} and t_{PHL} , which we defined in Chapter 2 and are illustrated in Figure 4.16. Recall that the parameter t_{PLH} designates the delay time between an input change and a corresponding low-to-high transition of an output. Likewise, t_{PHL} is the delay between an input change and a corresponding high-to-low output transition. For a latch circuit, t_{PLH} and t_{PHL} parameters represent the sum of the propagation delays through the gates

between a given latch input and output. Separate delay parameters are usually specified for each input/output pair.

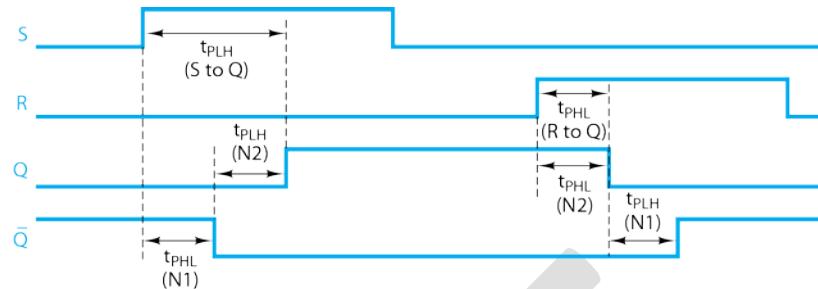
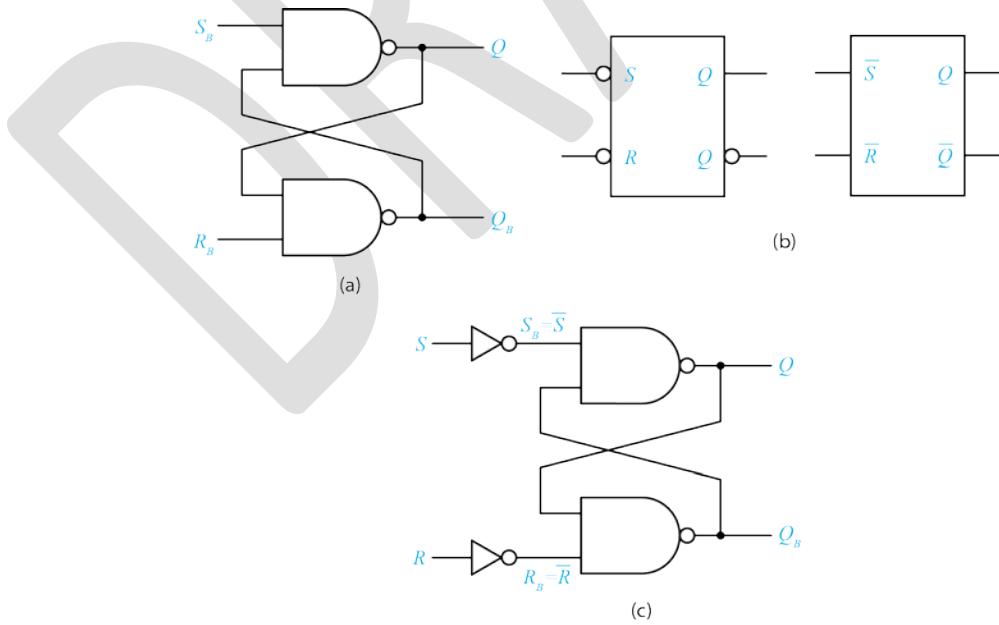


Figure 4.16: SR latch propagation delays.

SR Latch: NAND Structure

Can we produce the same functional device that we derived with NOR gates by using NAND gates? Let us replace the NOR gates in Fig. 4.14b with NAND gates, producing the circuit of Fig. 4.17a. The inputs to this circuit are labeled S_B and R_B because, as will be shown, their operation is complementary to the S and R inputs of the NOR circuit in that they are effectively active-low signals, i.e. a logic 0 on S_B or R_B will set or reset the latch, respectively. Therefore, this NAND circuit is commonly referred to as an $\overline{S}\overline{R}$ latch to indicate the active-low operation of the inputs. The logic symbols in Fig. 4.17b are commonly used to represent the $\overline{S}\overline{R}$ latch. Adding NOT gates to the excitation inputs, as shown in Fig. 4.17c, converts the latch inputs to S and R , producing a circuit equivalent to the NOR latch of Figure 4.14.

Figure 4.17: NAND $\overline{S}\overline{R}$ and SR latches.

(a) $\overline{S}\overline{R}$ cross-coupled form. (b) Logic symbols. (c) Inverters added to create an SR latch.

The logic equations realized by the two NAND gates are:

$$\text{Upper NAND gate: } Q = \overline{S_B Q_B} \quad (4.17)$$

$$\text{Lower NAND gate: } Q_B = \overline{R_B Q} \quad (4.18)$$

Consider the NAND gate outputs for all possible values of S and R.

Case 1. $S_B = 1, R_B = 1$: $Q_B = \overline{Q}$ and $Q = \overline{Q_B}$, which is the hold condition defined in eq. 4.12.

Case 2. $S_B = 0, R_B = 1$: Activating input S_B (setting $S_B = 0$) forces $Q = 1$. Since $R_B = 1$, we still have $Q_B = \overline{Q}$, and therefore $Q = 1$ forces $Q_B = 0$, which is the set condition.

Case 3. $S_B = 1, R_B = 0$: Activating input R_B (setting $R_B = 0$) forces $Q_B = 1$. Since $S = 0$, we still have $Q = \overline{Q_B}$ and therefore $Q_B = 1$ forces $Q = 0$, which is the reset condition.

The circuit holds the new state whenever both inputs are returned to logic 1. The combination $S_B = R_B = 0$ is not allowed, since it would produce the same unstable oscillatory behavior as setting $S = R = 1$ in an SR latch.

Representing Latch Behavior

Above, the behavior of an SR latch was studied by analyzing its logic circuit. When using a latch or flip-flop as an elementary component in a digital circuit, it is preferable to represent its behavior explicitly without referring to the memory circuit implementation details. This behavior can be expressed in several forms, including excitation tables, state diagrams, characteristic equations, or HDL models.

Excitation inputs		Present state	Next state Q^*
S	R	Q	
0	0	0	0 No change
0	0	1	1
0	1	0	0 Reset
0	1	1	0
1	0	0	1 Set
1	0	1	1
1	1	0	¥ Not allowed
1	1	1	¥

(a)

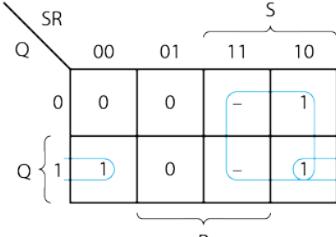
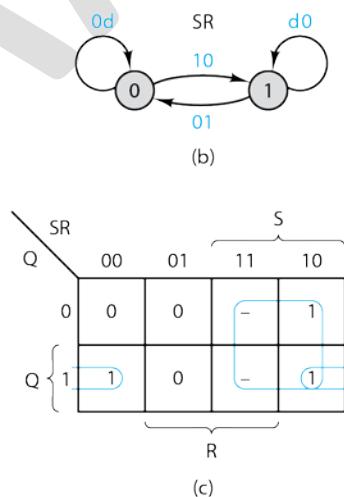


Figure 4.18: SR latch behavior. (a) Excitation table. (b) State diagram. (c) K-map of latch output Q^* .

The logical operation of the SR latch is summarized in the *excitation table* of Fig. 4.18a. The excitation table is simply the state table of the latch, showing the state transitions for each combination of excitation inputs. Each row of the table lists one combination of excitation input values S and R applied to an SR latch, the state Q of the SR latch at the instant those inputs are applied, and the state Q^* of the SR latch *after* the SR inputs have been applied and a steady-state condition has been achieved. We call column Q the *present state* of the SR latch and column Q^* the *next state*.

The information of Fig. 4.18a is often represented graphically in the form of a *state diagram*, as shown in Fig. 4.18b. As described in Section 4.1.2, each possible state of the circuit is represented by a circle in the state diagram, and transitions between states are shown as directed arcs. Each arc is labeled with the combinations of excitation input values that cause that transition. For example, from the state table of Fig. 4.18a, a transition from state 0 to state 1 is caused by setting $S = 1$ while $R = 0$. This is shown in the state diagram of 4.18b by an arc from state 0 to state 1 labeled with $SR = 10$. Likewise, the latch is reset by $SR = 01$, shown by the arc from state 1 to state 0. The arc that originates and ends at state 0 signifies that the latch remains in state 0 state for the indicated excitation inputs. $SR = 00$ is the hold condition and $SR = 01$ forces the latch to the reset state. Therefore, if the latch is in state 0, it remains in state 0 as long as $S = 0$, independent of whether $R = 0$ or $R = 1$. Consequently, this arc is labeled with $SR = 0d$, indicating that R is a don't care condition. Likewise, the arc originating and ending at state 1 is labeled with $SR = d0$.

When designing circuits containing memory elements, it is often useful to represent the operation of each memory element by its *characteristic equation*. Fig. 4.18c shows the information from the SR latch state table of Fig. 4.18a, plotted as a K-map, with the value of the next state Q^* plotted as a function of excitation inputs S and R and the present state Q . From this K-map can be derived the following expression.

$$Q^* = S + \bar{R}Q \quad (4.19)$$

Equation 4.19 is called the *characteristic equation* of the SR latch because it characterizes the operation of the latch. For example, we can classify the operation of the latch into three cases by substituting S and R values into Eq. 4.12 as follows.

$S = 0, R = 0: Q^* = 0 + \bar{0}Q = Q$, representing the hold condition.

$S = 1, R = 0: Q^* = 1 + \bar{0}Q = 1$, representing the set operation.

$S = 0, R = 1: Q^* = 0 + \bar{1}Q = 0$, representing the reset operation.

The behavior of the SR latch can also be described with a Verilog or VHDL model, as illustrated in Fig. 4.19. Here, the latch behavior is described with a Verilog always block or VHDL process that is sensitive to changes in excitation inputs S or R . Whenever either input changes, if $S = 1$ the latch will be set, and if $R = 1$ the latch will be reset. If $S = R = 0$, there will be no state change (the hold condition). These models assume that the condition $S = R = 1$ will not be allowed. However, if this condition should occur, then in this model the latch will be set since S is checked by itself before R is tested.

```
//SR Latch Behavioral Model
//
module SRlatch (S, R, Q, Qbar);
input S, R; //Declare excitation inputs
output reg Q, Qbar; //Declare complementary outputs
always begin
    if (S == 1'b1) begin //Set the latch on S=1
        Q = 1'b1;
        Qbar = 1'b0; end
    else if (R == 1'b1) begin //Reset the latch on R=1
        Q = 1'b0;
        Qbar = 1'b1; end
end //State doesn't change if S=R=0
endmodule
```

(a) Verilog model

```
-- SR Latch Behavioral Model
--
entity SRlatch is
port ( S,R:  in bit;      -- Excitation inputs
       Q,Qbar:out bit); -- Complementary outputs
end SRlatch;
architecture behavior of SRlatch is
begin
process(S,R) -- Latch reacts to S and R
begin
    if (S='1') then      -- Set the latch if S=1
        Q <= '1';
        Qbar <= '0';
    elsif (R = '1') then -- Reset the latch if R=1
        Q <= '0';
        Qbar <= '1';
    end if; -- No change if S=R=0
end process;
end;
```

(b) VHDL model

Figure 4.19: HDL models of an SR latch

4.2.1.2 Gated SR Latch

In many applications it is desirable to use a special control signal to inhibit state changes in a latch while its excitation inputs are changing. When the excitation inputs are ready, this control signal is activated to enable the latch to respond to the new values. This device is commonly referred to as a *gated latch*, since the control signal can be thought of as opening a gate through which signals on the excitation inputs can propagate to the output.

Circuit Structures

In Fig. 4.20a, a gating control signal, G , and a pair of AND gates are added to the SR latch of Fig. 4.14 to apply excitation inputs S and R to the latch during time intervals when $G = 1$. Thus, when $G = 1$ we say that the latch is "enabled" (i.e. the "gate" is open). When G is logic 0, the AND gate outputs are both 0, and therefore the SR latch inputs are held in the $S = R = 0$ (hold) state. In this case we say that the latch is "disabled" since when $G = 0$, no change occurs and thus the device is stable. When $G = 1$, the SR latch excitation table of Fig. 4.18a and the SR latch characteristic equation, Eq. 4.12, describe its function.

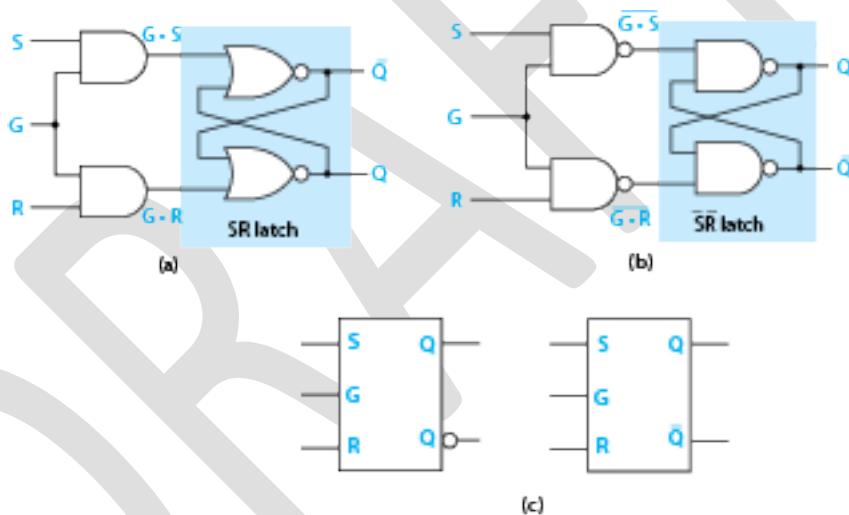


Figure 4.20: Gated SR latch. **(a)** With NOR gate SR latch. **(b)** With all NAND gates. **(c)** Logic symbols.

A more efficient circuit structure is created by using an $\bar{S}\bar{R}$ latch and changing the AND gates to NAND gates, as illustrated in Fig. 4.20b. This structure has the advantage of using all NAND gates, which are smaller and faster than AND gates. The generic logic symbols for the gated SR latch are presented in Fig. 4.20c.

Enable inputs	Excitation inputs		Present state Q	Next state Q^*
	G	S R		
0	x x	0	0	0 Hold
0	x x	1	1	
1	0 0	0	0	No change
1	0 0	1	1	
1	0 1	0	0	Reset
1	0 1	1	0	
1	1 0	0	1	Set
1	1 0	1	1	
1	1 1	0	x	Not allowed
1	1 1	1	x	

(a)

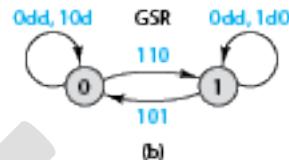


Figure 4.21: Gated SR latch characteristics. (a) Excitation table. (b) State diagram.

Characteristic Equation

The excitation table and state diagram of the gated SR latch are given in Figs. 4.21a and b, respectively. From the excitation table we can derive the K-map of Fig. 12a, and from this we can find the characteristic equation for the gated SR latch:

$$Q^* = SG + \bar{R}Q + \bar{G}Q \quad (4.20)$$

Note that when $G = 0$, Eq. 4.20 reduces to $Q^* = Q$, which means that the present latch state is *held*, i.e. the latch is disabled. Substituting $G = 1$ in Eq. 4.20 produces $Q^* = S + \bar{R}Q$, the characteristic equation of the simple SR latch, and thus the latch is enabled.

HDL Models

The behavior of the gated SR latch is described as an HDL model, as illustrated in Fig. 4.22. Comparing these to the models of the simple SR latch in Fig. 4.19, we see that the process is sensitive to changes in any of the three inputs G , S , and R . Changes to the latch state are conditioned on $G = 1$. Thus the latch state can change only if G changes from 0 to 1, or if S or R change while $G = 1$.

```
//Gated SR Latch Behavioral Model
//
module GatedSRlatch (G, S, R, Q, Qbar);
    input G, S, R;                                //Declare gate and excitation inputs
    output reg Q, Qbar;                            //Declare complementary outputs
    always                                         //Latch enabled by G=1
        if (G == 1'b1 & S == 1'b1) begin           //Set the latch on S=1
            Q = 1'b1;
            Qbar = 1'b0; end
        else if (G == 1'b1 & R == 1'b1) begin      //Reset (clear) the latch on R=1
            Q = 1'b0;
            Qbar = 1'b1; end
                                            //State doesn't change if S=R=0
endmodule
```

(a) Verilog model

```
-- Gated SR Latch Behavioral Model
entity Gated_SRlatch is
    port ( G,S,R: in bit;          -- Control and excitation inputs
           Q,Qbar: out bit);     -- Complementary outputs
end Gated_SRlatch;
architecture behavior of Gated_SRlatch is
begin
    process(G,S,R)                -- Latch reacts to G, S, or R
    begin
        if (G = '1') then         -- Latch is enabled by G=1
            if (S = '1') then     -- Set the latch
                Q <= '1';
                Qbar <= '0';
            elsif (R = '1') then -- Reset the latch
                Q <= '0';
                Qbar <= '1';
            end if;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.22: HDL models of a gated SR latch

4.2.1.3 D Latch

The most common application of latches in digital systems is for capturing and storing data. In such applications, we would like the memory element excitation inputs to be simply the data to be stored. Consequently, we need a device with a single excitation input D , whose value can be captured in the cross-coupled storage cell of a latch, becoming the new latch state, when the gating control signal G is activated.

Circuit Structure and Characteristic Equation

The logic symbols of the *D latch* (also called *delay latch* or *data latch*), are shown in Fig. 4.23a. We can make such a device from a gated SR latch. Examine Fig. 4.21a, the excitation table of the gated SR latch. If we assign $S = D$ and $R = \bar{D}$, then when enabled we restrict the operation of the latch to cases in which $S = \bar{R}$, corresponding to the four rows of the table in which $S = 1$ and $R = 0$, the set condition, or $S = 0$ and $R = 1$, the reset condition. The gated SR latch excitation table can therefore be reduced to the D latch excitation table given in Fig. 4.24a. The corresponding state diagram is shown in Fig. 4.24b.

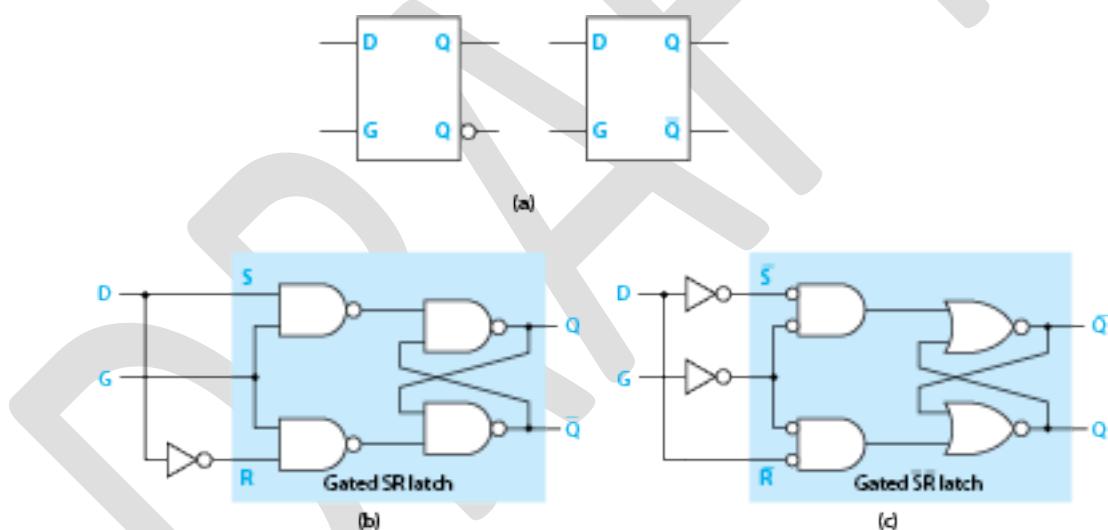
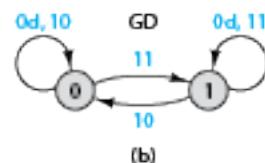


Figure 4.23: D latch. (a) Logic symbols. (b) NAND implementation. (c) NOR implementation

Enable input G	Excitation input D	Present state Q	Next state Q^*
0	x	0	0 Hold
0	x	1	1
1	0	0	0 Store 0
1	0	1	0
1	1	0	1 Store 1
1	1	1	1

Figure 4.24: D latch behavior. (a) Excitation table. (b) State diagram.



A NAND implementation of the D latch is shown in Fig. 4.23b. Note that this circuit is simply a gated SR latch with inputs $S = D$ and $R = \bar{D}$. An equivalent D latch implementation using NOR gates is given in Fig. 4.23c, in which a gated $\bar{S}\bar{R}$ latch is used, with inverters on the D and C inputs to change them from active low to active high. Therefore, the functional operation of this NOR implementation is identical to that of the NAND implementation of Fig. 4.23b.

The characteristic equation of the D latch can be derived from that of the gated SR latch by substituting D for S and \bar{D} for R in Eq. 4.13 and applying the Consensus Theorem, as follows:

$$\begin{aligned}
 Q^* &= SG + \bar{R}Q + \bar{G}Q \\
 &= DG + (\bar{D})Q + \bar{G}Q \\
 &= DG + DQ + \bar{G}Q \\
 &= DG + \bar{G}Q
 \end{aligned} \tag{4.21}$$

Characteristic equation 4.21 describes the behavior of the D latch. When the enable signal is low ($G = 0$), Eq. 4.21 reduces to $Q^* = Q$. In this case, the latch is placed in the *hold state*, that is, it holds the last value of D that was captured. In other words, when $G = 0$, data are held, or stored, in the latch. Substituting $G = 1$ in Eq. 4.14 gives $Q^* = D$. Thus, whenever the enable signal G is high the latch state Q^* is forced to be the value of the input excitation D ; that is, the excitation input D is gated directly to output Q (i.e. Q follows D), effectively making the latch “transparent”. In this case, the D latch is said to be in the *gated* or *enabled* mode.

The operation of the D latch is illustrated in the timing diagram of Fig. 4.25. Note that, when G is high, the latch appears transparent in that any and all changes on D will pass through to the latch's output. In the hold state, the latch stores the last value of D that was present on its input at the instant in which the enable signal transitioned from high to low, that is, for a $1 \rightarrow 0$ transition on signal G . This behavior is also represented by the HDL models of Fig. 4.26. These models are similar to those of the SR latch, except that when enabled ($G = 1$), the latch state Q is simply assigned the value of excitation input D .

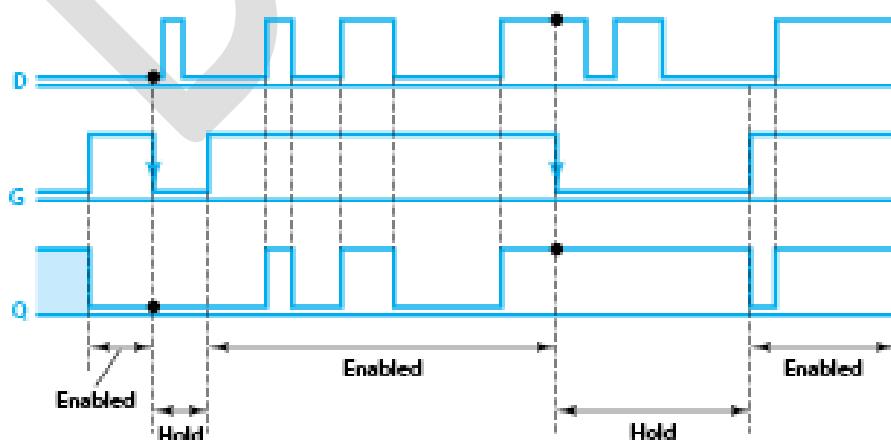


Figure 4.25: D Latch Timing Diagram

```
//Gated D Latch Behavioral Model
//
module GatedDlatch (G, D, Q, Qbar);
input G, D;                                //Declare gate and excitation inputs
output reg Q, Qbar;                         //Declare complementary outputs
always
    if (G == 1'b1) begin                     //Latch enabled by G=1
        Q = D;                               //Data input value transferred to latch
        Qbar = ~D; end
endmodule
(a) Verilog model
```

```
-- D Latch Behavioral Model
--
entity Dlatch is
    port ( G,D:      in bit;           -- Control and excitation inputs
           Q,Qbar: out bit);          -- Complementary outputs
end Dlatch;
architecture behavior of Dlatch is
begin
    process(G,D)                      -- Latch reacts to G and D
    begin
        if (G='1') then               -- Latch is enabled when G=1
            Q <= D;                 -- Data input transferred to the latch
            Qbar <= not D;
        end if;
    end process;
end;
(b) VHDL model
```

Figure 4.26: HDL models of a D latch

Setup Time, Hold Time, and Pulse-width Constraints

To ensure that a specific value on excitation input D will determine the final state of the latch, D must not be allowed to change too near the time at which the enable signal makes its transition from high to low. Therefore, gated latches have two time constraint parameters that specify when the excitation input must be held constant to guarantee correct operation. These are illustrated in Fig. 4.27.

The latch *setup time*, denoted t_{su} , is defined as the period of time immediately preceding the enable signal transition during which the excitation input must be stable; that is, the excitation input must be "set up" and stable at least t_{su} prior to the enable signal transition and should not change until well after the transition.

The latch *hold time*, denoted t_h , is defined as the period of time immediately following the enable signal transition during which D should not change. Therefore, the excitation input must be held constant for at least t_h following the enable signal transition to ensure that the correct value has been latched.

Setup and hold times are illustrated in the diagram of Fig. 4.27. Here we have shown $t_{PLH} = t_{PHL} = 0$ to make the diagram easier to understand. Note the two constraint violations on the timing diagram. The change in D from $0 \rightarrow 1$ too close to the enable signal edge represents a setup time violation, and therefore the latch output may or may not change from $0 \rightarrow 1$ as desired. Likewise, the change in D from $1 \rightarrow 0$ too soon after the enable signal edge may result in an unpredictable state.

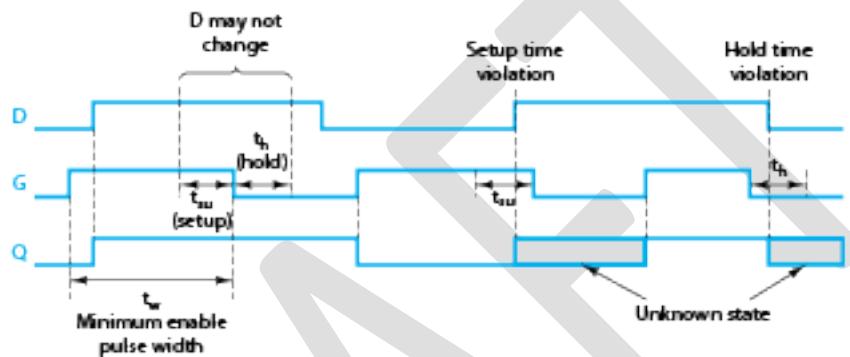


Figure 4.27 D Latch Timing Constraints

In addition to setup and hold time constraints, most gated latches require a minimum pulse width on the enable input to guarantee a correct state change. This minimum pulse width is denoted by t_w , as shown on Fig. 4.27. Any pulse whose width is shorter than the specified minimum t_w may not be sufficient to initiate a desired state change.

Consequently, it is the responsibility of the user of a gated latch to ensure that all enable pulses are of sufficient width to cause state changes and that, for any enable input transition at time T , no excitation input changes occur within the time period $[T - t_{su}, T + t_h]$.

Direct Implementations

Whereas the D latch presented in Fig. 4.23 was created by modifying a gated SR latch, efficient latch circuits can also be derived directly from their characteristic equations. The logic circuit displayed in Fig. 4.28a is a direct implementation of the D latch characteristic equation, Eq. 4.14. If desired, a complementary output \bar{Q} can be produced by adding an inverter to the Q output.

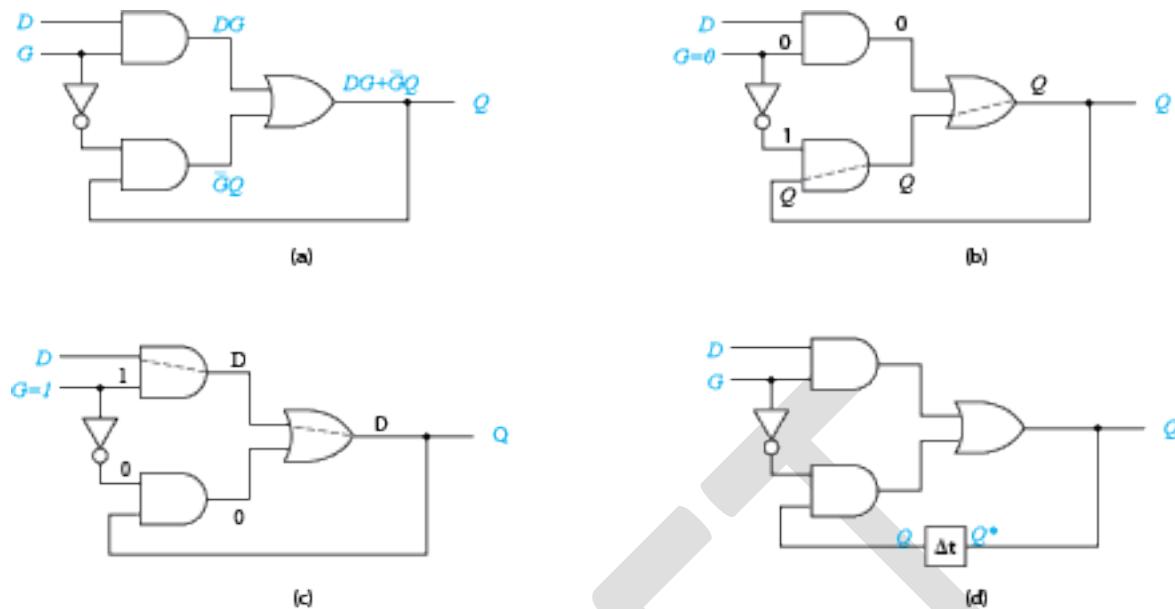


Figure 4.28: Direct D latch implementation. **(a)** Logic diagram. **(b)** Hold, or storage, mode. **(c)** Gated/transparent mode. **(d)** Sequential circuit model.

This circuit has the same operating modes as the previous D latch design in Fig. 4.23. The operation of the latch can easily be seen by observing the circuit of Fig. 4.28. As illustrated in Fig. 4.28b, when G is held at logic 0 the upper AND gate is disabled, sending a logic 0 to the OR gate. At the same time, a logic 1 is applied to the lower AND gate, making it behave as a transparent gate, passing feedback signal Q to and through the OR gate to the output, effectively regenerating the output. Therefore, the latch is in its *hold* or *storage* mode.

Now suppose we change the enable input G to logic 1 as shown in Fig. 4.28c. This mode of operation enables a path from input D through the upper AND gate and the OR gate to the latch's output Q ($Q = D$). Since the input is gated directly to the output, we call this the *gated* operating mode.

Correct operation of the latch requires that there be some delay between changes in inputs D or G and any corresponding changes in the feedback signal Q , so that next state, Q^* , will indeed be determined by the present state, Q , and the inputs. This delay is provided by the propagation delays through the gates of the two-level latch circuit. We often show this delay as illustrated in the sequential circuit model for the latch shown in Fig. 4.28d, in which the the delays of the logic gates are modeled as a single lumped parameter, Δt . This delay is a function of the particular technology used to implement the gates.

Hazard-free D Latch

While the D latch circuit of Fig. 4.28a uses a minimum number of logic gates, the circuit contains a static 1 hazard, making it subject to output glitches; that is, its output may momentarily change to 0 when it is expected to remain at a constant 1 level during certain input changes. A hazard-free design for this D latch may be needed in some applications to eliminate such glitches at the output. We can design a hazard-free version of this circuit from a

K-map of its characteristic equation, presented in Fig. 4.29a. The static 1 hazard occurs when G changes from 1 to 0, traversing from the cell containing product term DG to that containing product term $\bar{Q}G$ on this map. These two terms correspond to the two AND gates in Fig. 4.28a.

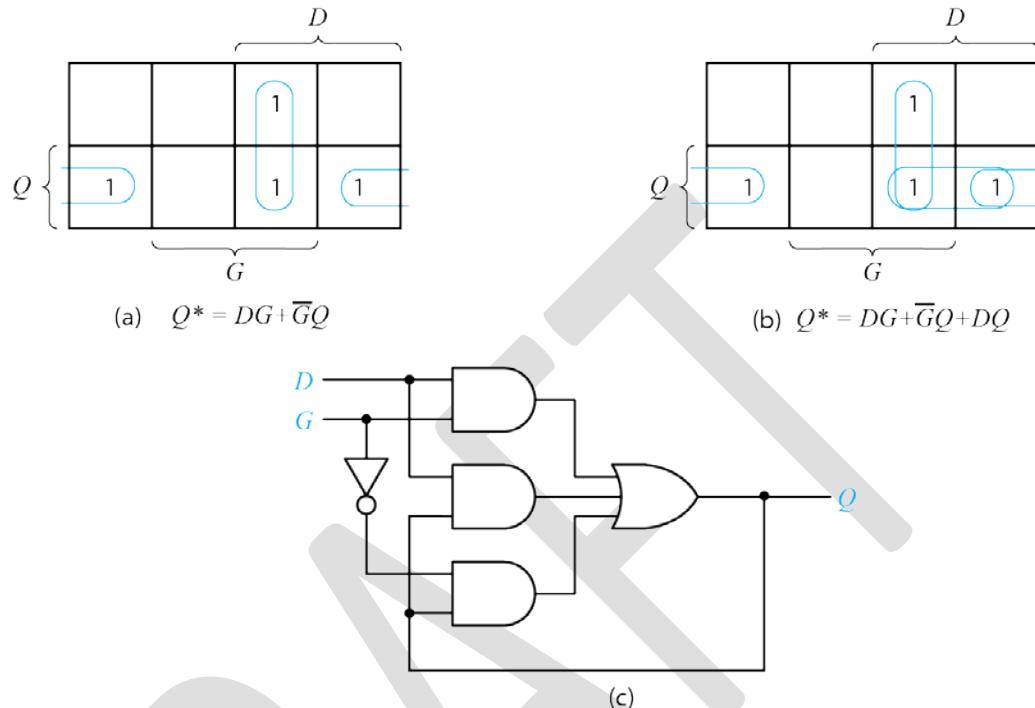


Figure 4.29: Hazard-free D latch, the SN74116. (a) K-map for state Q^* of a D latch. (b) K-map for hazard-free D latch. (c) Hazard-free D latch logic diagram.

As described in Chapter 2, we can eliminate the static 1 hazard by adding the redundant product term DQ , as shown in the K-map of Fig. 4.29b, producing the logic equation

$$Q^* = DG + \bar{G}\bar{Q} + DQ \quad (4.22)$$

The resulting logic diagram for the hazard-free design thus requires three AND gates, as shown in Fig. 4.29c.

4.2.2 Flip-flops

The latch circuits presented thus far are often inappropriate for synchronous sequential logic circuit applications. When the enable signal G is active, the latch is transparent, enabling excitation inputs to be gated directly to the output Q . Thus, any input changes are immediately reflected at the latch output. Recall our model for the synchronous sequential circuit, presented in Fig. 4.2. The output signals from the memory elements are the input signals to the combinational logic, and vice versa. When a latch enable input is active, making the latch transparent, the combinational logic circuit outputs are effectively gated back to its inputs, with the possibility of producing oscillations and unstable transient behavior.

This is illustrated by the circuit in Fig. 4.30a, in which an input N is combined with the latch output Q and fed back to the latch input, giving $D = N \oplus Q$. If $N = 1$, then the latch input becomes $D = 1 \oplus Q = \bar{Q}$. Therefore, when $G = 1$ the latch is in transparent mode, and the state of the latch becomes $D = \bar{Q}$, thereby complementing the state of the latch. However, after the latch and XOR gate propagation delays, this change is fed back to input D , and the latch state is complemented again. As shown in the timing diagram in Fig. 4.30b, the state will continue to toggle back and forth until the latch is put into the hold mode by $G = 0$. The final state of the latch is therefore unpredictable, as it depends on the duration of the enable pulse and the lengths of the propagation delays. This problem is solved by modifying the memory design and using a special timing control signal called a *clock* to restrict the times at which the states of the memory elements may change.

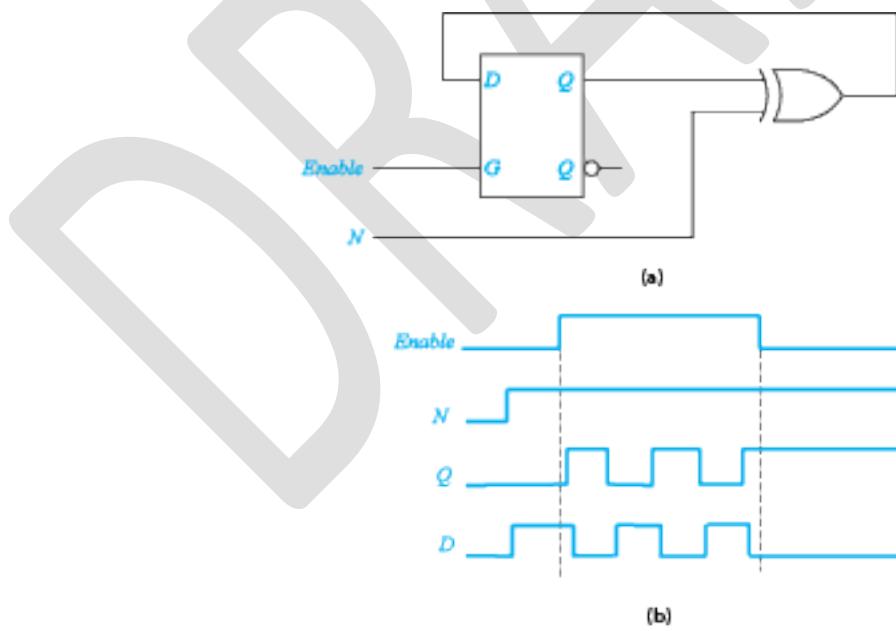


Figure 4.30: Oscillations created by using a latch in a sequential circuit. **(a)** Logic diagram. **(b)** Timing behavior.

4.2.2.1 Master-Slave D Flip-Flops

Circuit Structure and Operation

One method to prevent the unstable behavior just described is to employ two latches in a *master-slave* configuration, as shown in Fig. 4.31a, thus creating a *flip-flop*. In this figure, the master is a D latch, and therefore this configuration is called a *D flip-flop*. The enable signals of the two latches are driven by complementary levels of a clock signal, C . When the clock signal is low, the master latch is in the gated mode and the slave is in the hold mode. Therefore, changes on the excitation input signal D are gated through the master latch to Q_M , while the slave latch holds its state and output Q constant, ignoring any changes on Q_M . When the clock changes to logic 1, the two latches exchange roles. The master latch enters the hold mode and ignores any further changes on its inputs, holding Q_M constant, while the slave latch enters the gated mode, passing Q_M to the flip-flop output Q . Since the master holds Q_M constant while the slave latch is enabled, no further changes will occur on output Q . The same behavior occurs in the alternate D flip flop circuit in Fig 4.31c, with a gated SR latch used as the slave. When the slave latch enters the gated mode, $Q_M = 1$ results in $S = 1$ and $R = 0$, setting Q to 1, and $Q_M = 0$ results in $S = 0$ and $R = 1$, resetting Q to 0.

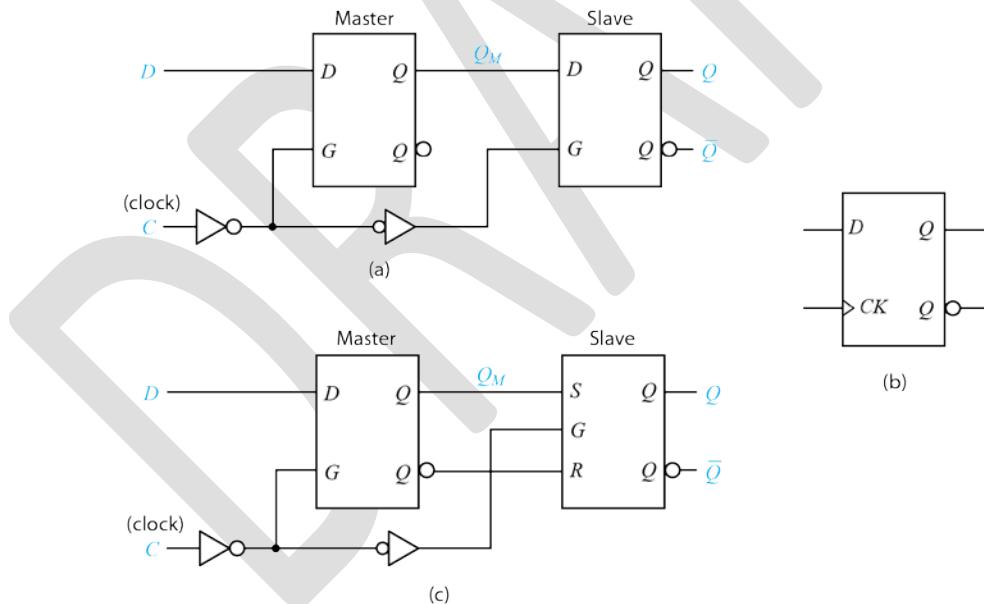


Figure 4.31: Master-slave D flip-flop. (a) Logic diagram. (b) Logic symbol c) Gated SR latch used as the slave

Master-slave flip-flops like the one in Fig. 4.31 require pulses on the clock input, i.e. both logic $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions, in order to operate properly. This is illustrated in Fig. 4.32. On each $1 \rightarrow 0$ transition (times T2, T4, T6), the master enters the enabled mode, allowing Q_M to change to the value on excitation input D . On each $0 \rightarrow 1$ transition (times

T_1, T_3, T_5, T_7), the slave enters the enabled mode, allowing Q to change to the value on master output Q_M . Since output Q changes only when the slave enters the enabled mode on the rising (positive) edge of the clock pulse, this flip flop can be referred to as *rising-edge triggered*. This is indicated by a triangle on the clock input CK of the D flip flop logic symbol, as shown in Fig. 4.31b. Note that changes on input D while the master is in the hold mode, i.e. between times T_1 and T_2 , and between times T_5 and T_6 , are ignored until the master next enters the gated mode. Likewise, changes on Q_M are ignored by the slave until it enters the gated mode at times T_3, T_5 and T_7 . Thus, if the D flip-flop is used in a synchronous sequential circuit, unstable oscillations are prevented because, at all times, either the master latch or the slave latch is in the hold mode, effectively preventing changes in the excitation inputs from propagating directly to the output, as happens in an enabled latch.

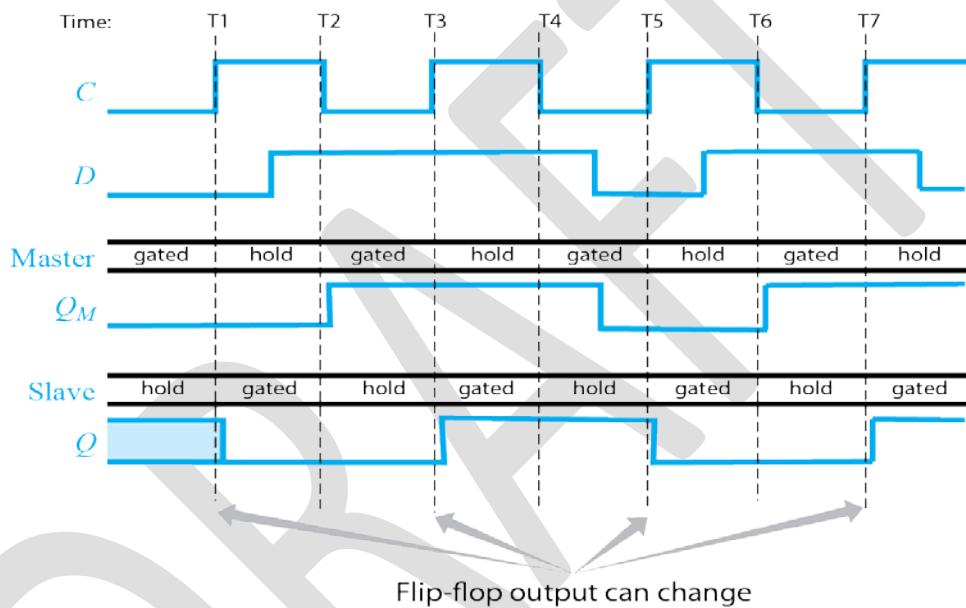


Figure 4.32: D flop-flop timing

Some applications require flip flops whose outputs change on the falling edge of the clock pulse. Removing the inverter from the clock input, as shown in Figure 4.33a, results in a *falling-edge triggered* D flip flop, in which the master enters the gated mode on the $0 \rightarrow 1$ transition, and the slave on the $1 \rightarrow 0$ transition of the clock pulse. Therefore, output Q changes on the falling edge of the clock, when the slave enters the gated mode. On the D flip flop logic symbol, as shown in Fig. 4.33b, a bubble on the clock input indicates that the output changes on the falling (negative) edge of the clock.

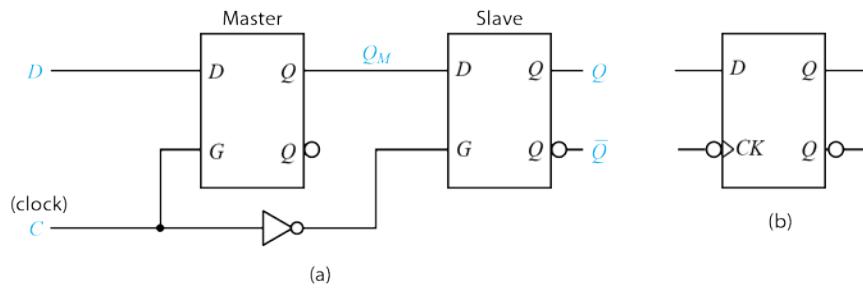


Figure 4.33: Falling-edge triggered D flip flop. (a) Logic diagram. (b) Logic symbol.

Timing Constraints

To ensure proper operation of the D flip flop, the D input to the master latch should be stable before the clock transition that puts the master into the hold mode. Therefore, the flip-flop input is subject to similar setup and hold time constraints described earlier for gated latches and illustrated in Fig. 4.27. Figure 4.34 illustrates the setup and hold times for the D flip-flop of Fig. 4.31a. Since the excitation input D affects only the master latch, its setup and hold times are defined relative to the rising edge of the clock signal, which is the clock transition that changes the master latch from the gated mode to the hold mode. To ensure that the desired level of D is captured by the master latch, D may not change too close to the rising edge of the clock. The excitation inputs of the slave latch are connected to the outputs of the master latch and are therefore not directly affected by the external excitation inputs.

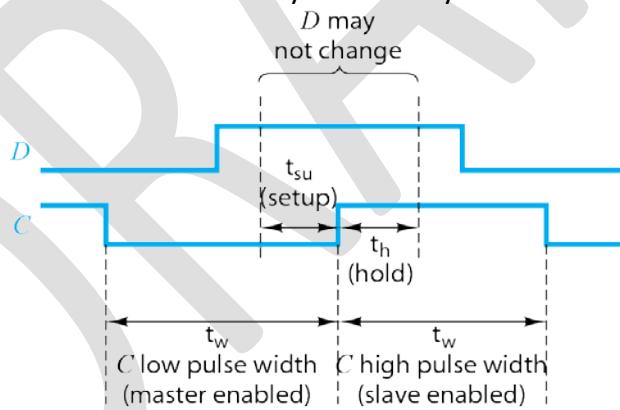


Figure 4.34: D flip flop setup and hold timing constraints

Figure 4.34 also illustrates minimum clock pulse-width constraints for the master-slave flip-flop. The low pulse-width parameter is the minimum pulse width required for proper operation of the master latch, while the high pulse-width parameter is the minimum pulse width required for the slave latch. The sum of these two pulse widths determines the minimum period of any clock signal to be used for the flip-flop.

Excitation Table and Characteristic Equation

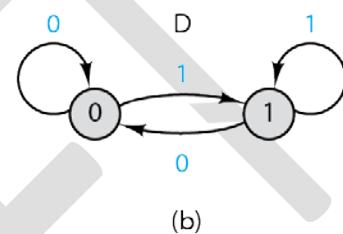
The excitation table of the master–slave D flip-flop is given in Fig. 4.35a and the state diagram in Fig. 4.35b. The overall behavior of the D flip-flop output Q can be summarized by noting that Q will assume the value of D on the rising edge of the clock C . Therefore, the characteristic equation for a master–slave D flip-flop is simply

$$Q^* = D \quad (4.23)$$

This behavior is expressed as HDL models in Fig. 4.36. Note that the Verilog always block and the VHDL process are sensitive only to clock transitions, and that the output is assigned the value of input D only when CK makes a transition to logic 1.

D	Q	C	Q^*
0	0	↑	0 Store 0
0	1	↑	0
1	0	↑	1 Store 1
1	1	↑	1

(a)



(b)

Figure 4.35: Master-slave D flip-flop characteristics. (a) Excitation table. (b) State diagram.

```
//Positive-Edge Triggered D Flip-Flop Behavioral Model
//
module Dflipflop(D, CK, Q, Qbar);
    input D, CK;
    output reg Q, Qbar;
    always @ (posedge CK)      //Flip flop triggers on 0 to 1 transition of CK (clock)
        begin
            Q = D;           //Data input transferred to flip flop
            Qbar = ~D;
        end
endmodule
```

(a) Verilog model

```
-- D Flip-Flop Behavioral Model
--
entity Dflipflop is
    port ( CK,D: in bit;          -- Clock and excitation inputs
           Q,Qbar:out bit);     -- Complementary outputs
end Dflipflop;
architecture behavior of Dflipflop is
begin
    process(CK)                  -- Flip flop reacts to clock only
    begin
        if CK'event and (CK='1') then -- Flip flop triggers when CK becomes 1
            Q <= D;               -- Data input transferred to the flip flop
            Qbar <= not D;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.36: HDL models of a rising-edge triggered D flip flop

4.2.2.2 Other Master-Slave Flip-flop Types

Other useful flip flop types can be created from the master-slave configuration by manipulating the excitation inputs to the master latch. Such flip-flops operate in the same manner as the D version described above, and can be either rising-edge or falling-edge triggered. The master latch is gated when the clock is at one level and the slave, when the clock is at the other level.

SR Flip Flop

We can build a master–slave SR flip-flop from gated SR latches as shown in Figure 4.37a. Note that this flip-flop operates in the same manner as the D version of Fig. 4.31c. The master latch is gated when the clock is low, and the slave when the clock is high. The logic symbol for this rising-edge triggered device is shown in Fig. 4.37b. Note that the logic symbol indicates that the outputs change on the rising (positive) edge of a pulse on the clock signal.

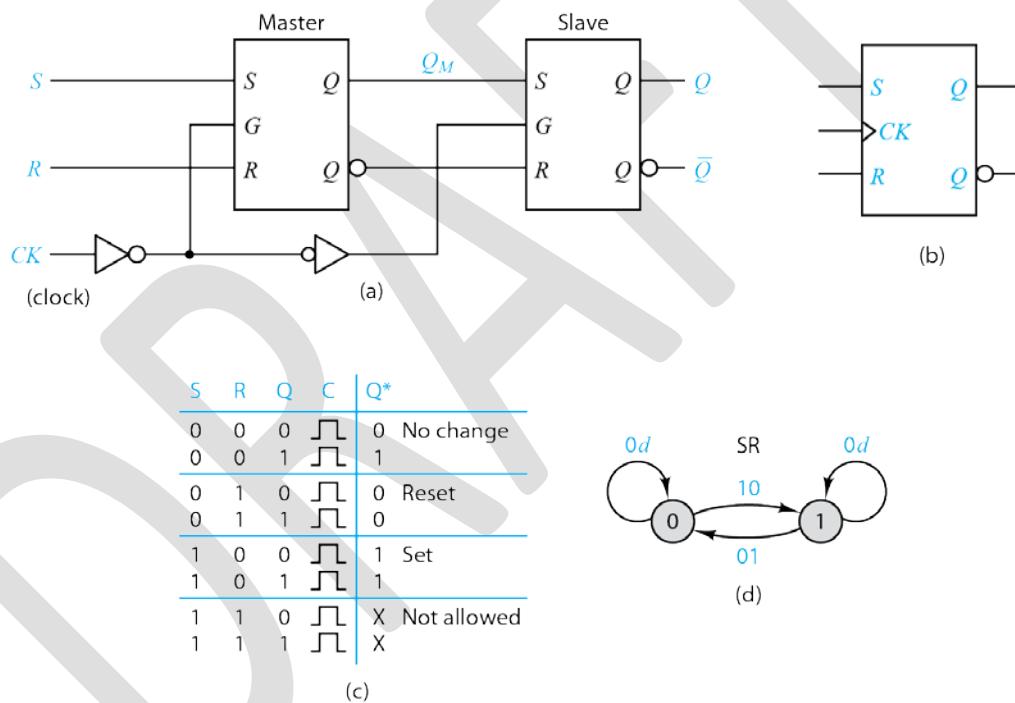


Figure 4.37: Master-slave SR flip-flop. (a) Logic Diagram (b) Logic symbol. (c) Excitation table. (d) State diagram.

The excitation table and state diagram for the SR master–slave flip-flop are presented in Figs. 4.37c and 4.37d, respectively. Note that the columns S , R , and Q of the excitation table denote the conditions on the flip-flop signals *before* the clock pulse is applied. The column Q^* denotes the flip-flop output *after* the clock pulse has been applied. Comparing this table to Fig. 4.18a, we see that the operation of the master–slave SR flip-flop is similar to that of the simple SR latch. Likewise, the state diagrams are identical, with the difference being that the latch changes states immediately when S or R changes, whereas all flip-flop state changes

are triggered by clock pulses. Consequently, the same characteristic equation describes the operation of both devices:

$$Q^* = S + \bar{R}Q \quad (4.24)$$

JK Flip Flop

The JK flip-flop may be considered an extension of the SR design examined above, with excitation inputs J and K corresponding to SR flip-flop inputs S and R, respectively. However, whereas the $S = R = 1$ input combination is not allowed, the JK uses this special case to incorporate a very useful mode of operation. The additional feature of the JK device is that when $J = K = 1$, its state *toggles*, that is, changes from $0 \rightarrow 1$ or from $1 \rightarrow 0$. The four modes of operation (hold, set, reset, and toggle) are summarized in the excitation table presented in Fig. 4.38a and the corresponding state diagram in Fig. 4.38b.

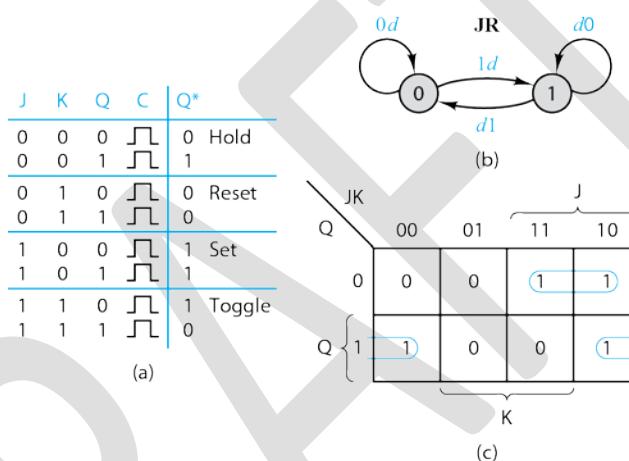


Figure 4.38: JK flip-flop characteristics. (a) Excitation table. (b) State diagram. (c) K-map for Q^* .

Examine the state diagram of Fig. 4.38b. The JK flip-flop will change from the 0 state to the 1 state with an input of $J=1$ and $K=0$ (set) or $J=1$ and $K=1$ (toggle). That is, a logic 1 on J will force the device into the 1 state no matter what value is placed on input K . Therefore, K is a don't-care condition, denoted on the state diagram by a value of d . The remainder of the diagram may be derived from the excitation table.

By plotting the next state Q^* on a K-map, as shown in Fig. 4.38c, the characteristic equation of the JK flip-flop can be derived:

$$Q^* = J\bar{Q} + \bar{K}Q \quad (4.25)$$

From this equation, the logic diagram for the JK flip-flop can be derived from a D flip flop, as presented in Fig. 4.39a, by recalling that $Q^* = D$. Alternatively, the simpler structure shown in Figure 4.39b, can be derived using the characteristic equation of the SR flip flop, and assigning $S = J\bar{Q}$ and $R = KQ$.

$$\begin{aligned}
 Q^* &= S + \bar{R}Q \\
 &= J\bar{Q} + (\bar{K}\bar{Q})Q \\
 &= J\bar{Q} + (\bar{K} + \bar{Q})Q \\
 &= J\bar{Q} + \bar{K}Q
 \end{aligned} \tag{4.26}$$

The logic symbol for the JK flip flop is shown in Fig. 4.39c. Note that this is a rising-edge triggered flip flop. The falling-edge triggered device can be obtained either by using a falling-edge triggered D or SR flip flop in Figs. 4.39a or 4.39b, respectively, or by using an inverter on clock input CK.

HDL models of the JK flip flop are illustrated in Fig. 4.40. Note the conditions for the set, reset, and toggle operation for the corresponding input combinations of J and K .

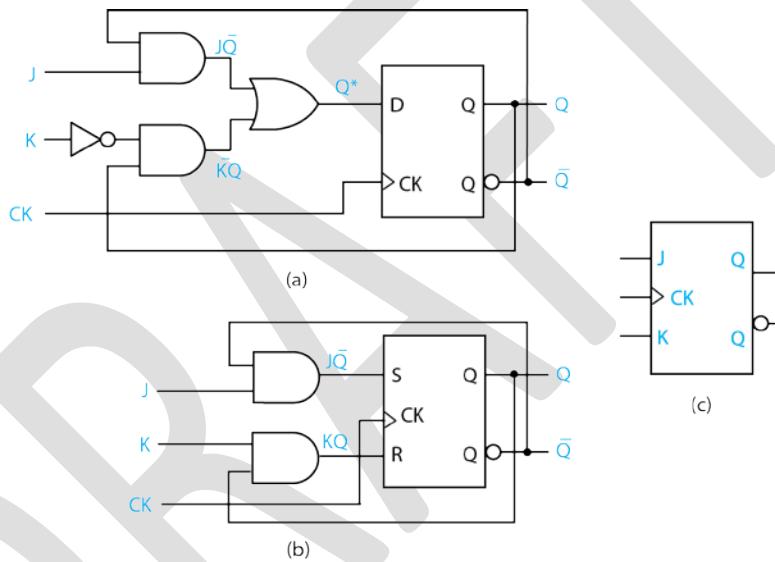


Figure 4.39: JK flip-flop. (a) Created from a D flip flop. (b) Created from an SR flip flop. (c) Logic symbol.

Note that the JK flip flop is comparable to the SR flip flop in complexity, with only two additional gates. Therefore, since the JK flip-flop includes all of the functionality of the SR flip-flop, and has the additional toggle operation that eliminates the potential of the $S = R = 1$ race condition, SR flip flops are rarely used in practice, with JK flip flops used instead.

```
//Positive-Edge Triggered JK Flip-Flop Behavioral Model
module JKflip_flop (J,K,CK,Q,Qbar);
input J, K, CK;
output reg Q, Qbar;
    always @ (posedge CK)
        if (J & ~K) begin          //Set flip flop when J=1 and K=0.
            Q = 1'b1;
            Qbar = 1'b0; end
        else if (~J & K) begin     //Reset flip flop when J=0 and K=1.
            Q = 1'b0;
            Qbar = 1'b1; end
        else if (J & K) begin      //Toggle flip flop when J=K=1.
            Q <= Qbar;
            Qbar <= Q; end
    endmodule
//No state change when J=K=0.
```

(a) Verilog model

```
-- JK Flip-Flop Behavioral Model
entity JKflipflop is
port ( CK,J,K: in bit;           -- Clock and excitation inputs
       Q,Qbar: out bit);         -- Complementary outputs
end JKflipflop;
architecture behavior of JKflipflop is
    signal Qint,QBint: bit;      --internal flip-flop state
begin
    process(CK)                  -- Flip flop reacts to clock only
    begin
        if CK'event and (CK='1') then   -- Flip flop triggers when CK becomes 1
            if (J = '0') and (K = '1') then
                Qint <= '0';           -- Reset flip flop state to 0
                QBint <= '1';
            elsif (J = '1') and (K = '0') then
                Qint <= '1';           -- Set flip flop state to 1
                QBint <= '0';
            elsif (J = '1') and (K = '1') then
                Qint <= QBint;         -- Toggle flip flop state
                QBint <= Qint;
            end if;                  -- No state change if J=K=0
        end if;
    end process;
    Q <= Qint;  Qbar <= QBint; --drive outputs
end;
```

(b) VHDL model

Figure 4.40: HDL models of a positive-edge triggered JK flip flop

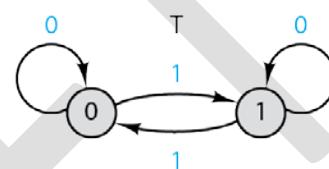
T Flip Flop

A common building block used in sequential logic circuits that perform counting functions is the T (*trigger* or *toggle*) flip-flop. The function of this device is to change (toggle) its state on the rising edge of each clock pulse when excitation input $T = 1$, and hold its state when $T = 0$. This behavior is shown in the excitation table of Fig. 4.41a and the state diagram of Fig. 4.41b.

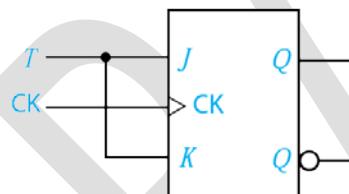
Although the T flip flop is not typically available as a stand-alone device, it can be easily constructed from a JK flip-flop with the J and K inputs tied together to create a single input T , as shown in Fig. 4.41c. This is the most commonly used implementation, since JK flip-flops are readily available. Referring to the JK flop flop excitation table in Fig. 4.38a, we use only two cases: the toggle condition when $T = 1$, making $J = K = 1$, or the hold condition when $T = 0$ making $J = K = 0$. The T flip flop logic symbol is shown in Fig. 4.41d.

T	Q	C	Q^*
0	0	—	0 Hold
0	1	—	1
1	0	—	1 Toggle
1	1	—	0

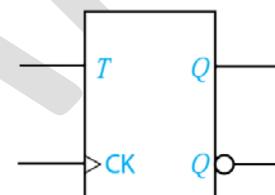
(a)



(b)



(c)



(d)

Figure 4.41: T flip-flop. (a) Created with JK flip flop. (b) Logic symbol.

The characteristic equation of the T flip-flop can be derived from that of the JK flip-flop by substituting T for J and K as follows:

$$\begin{aligned} Q^* &= J\bar{Q} + \bar{K}Q \\ &= T\bar{Q} + \bar{T}Q \end{aligned} \quad (4.27)$$

For $T = 0$, the characteristic equation reduces to $Q^* = Q$, which is the hold condition, while for $T = 1$, the characteristic equation becomes $Q^* = \bar{Q}$, which represents the toggle condition.

4.2.2.3 Asynchronous Inputs

It is often desirable or necessary to force latches or flip flops in a sequential circuit to known states by activating a "reset" signal. For example, one might want to initialize the states of memory elements that power up in random states, or force a circuit back into a given starting state at some arbitrary point in time.

Gated latches and flip flops may have one or more *asynchronous inputs* for this purpose. The term "asynchronous" implies that these input signals are not synchronized with latch enable or flip-flop clock inputs. Asynchronous inputs bypass the gate/clock to set or reset the state directly and immediately. In contrast, the excitation inputs described previously are called *synchronous inputs*, because they only affect the state at the time of a clock transition or pulse, i.e. they are synchronized to the clock.

The gated SR latch circuit, shown in Fig. 4.42a along with the corresponding logic symbol, has two asynchronous inputs labeled *PRE* ("Preset") and *CLR* ("Clear"), which are used to force the latch into the set or reset states, respectively. These inputs are applied to the output SR latch of the circuit, bypassing the clock gating circuit, and effectively acting as second *S* and *R* inputs to the output latch.

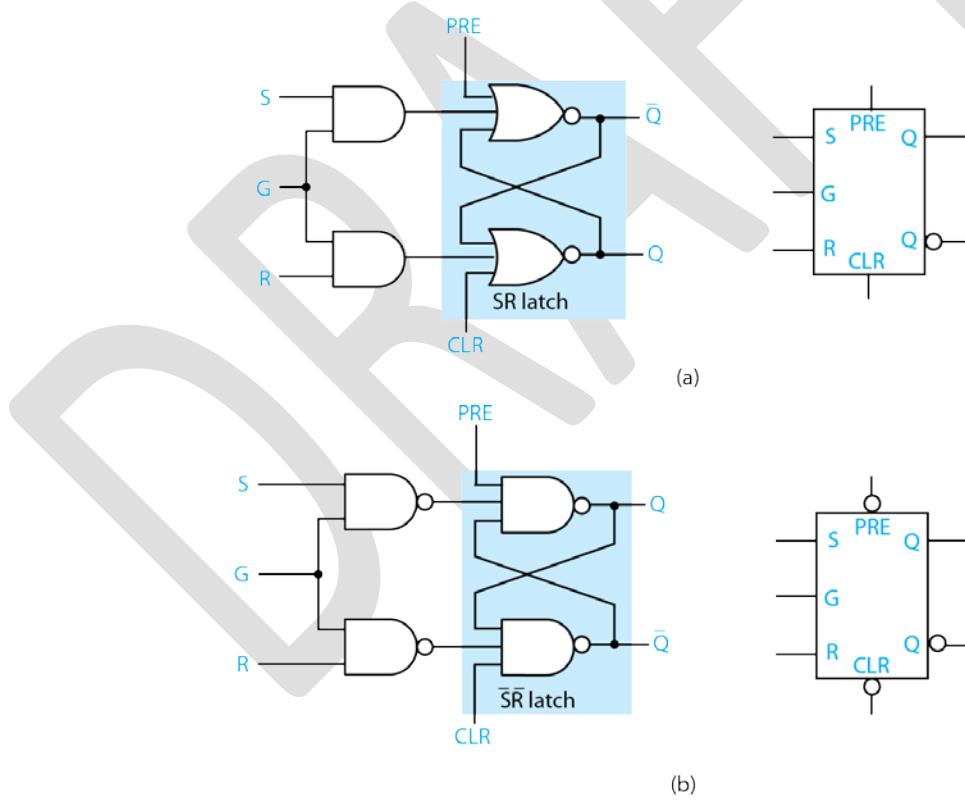


Figure 4.42: Latches with asynchronous inputs. (a) Active-high PRE and CLR.
(b) Active-low PRE and CLR.

Let us examine the asynchronous *CLR* input (sometimes called *CLEAR* or *RESET*) in Fig. 4.42a. When $CLR = 0$, this input has no effect on the operation of the latch, and the output is determined by the *S*, *R*, and *G* inputs as usual. If we make $CLR = 1$, then the

NOR gate driving the Q output is forced to 0 and feedback forces \bar{Q} to 1, independent of the rest of the circuit. Thus the latch state is reset to 0. Note that as long as we hold $CLR = 1$, the output latch will be held in the reset state, and thus will not be affected by any changes in the SR or G inputs.

By symmetry, the asynchronous PRE input (sometimes called SET or PR) acts as a second S input to the latch. Making $PRESET = 1$ forces the latch into the set state ($Q = 1$ and $\bar{Q} = 0$), independent of the clock. When $PRESET = 0$, this input has no effect on the latch output. As is the case with latch S and R inputs, the condition $CLR = PRE = 1$ will produce improper behavior, and is thus not allowed.

If the output latch is formed with NAND gates (an $\bar{S}\bar{R}$ latch), as shown in Fig. 4.39b, then the asynchronous inputs are *active low*, that is they set/reset the state when they are 0, effectively acting as additional \bar{S} and \bar{R} inputs to the output latch. Note the use of inversion bubbles on the logic symbol in Fig. 4.42b to indicate the active-low asynchronous inputs.

Some latches and flip flops have a single asynchronous input, either PRE or CLR , connected to one of the gates of the output latch. In such cases, initialization of the latch/flip flop is limited to the state corresponding to the name of the asynchronous input.

Asynchronous inputs to flip flops are generally configured as in Fig. 4.42, connected to the output stages of each of the master and slave latches, so that each is forced to the desired state, bypassing the clock, for each asynchronous input.

HDL models of a rising-edge triggered D flip flop with active-high preset and clear inputs are presented in Fig. 4.43. Note that signals PRE and CLR have been added to the Verilog always block and VHDL process sensitivity lists, since changes in these signals can trigger changes in the flip-flop state, in addition to clock transitions. Within the process, the ordering of the statements effectively gives preset precedence over clear and the clock, since the state of PRE is checked first and therefore if $PRE=1$, the state of the flip flop is set to 1, without checking CLR or CK . If $PRE = 0$, then the CLR input is tested, and will reset the state if active. The clock only triggers state changes if PRE and CLR are both inactive, i.e. $PRE = CLR = 0$. In this model, it is assumed that PRE and CLR will never be activated at the same time.

```
//Positive-Edge Triggered D Flip-Flop With Active-Low Preset and Clear Behavioral Model
//
module DflipflopPreClr (D,CK,PRE,CLR,Q,Qbar);
    input D, CK, PRE, CLR;
    output reg Q, Qbar;
always @ (negedge PRE,negedge CLR,posedge CK)           //Detect an input change.
    if (CLR==0) begin Q <= 1'b0; Qbar <= 1'b1; end          //Reset flip flop when CLR=0.
    else if (PRE==0) begin Q <= 1'b1; Qbar <= 1'b0; end //Set flip flop when PRE=0.
    else if (CK==1)
        begin
            Q <= D;           //Transfer input to flip flop when CK=PRE=CLR=1.
            Qbar <= ~D;
        end
endmodule
```

(a) Verilog model

```
-- D Flip-Flop model with asynchronous inputs
--
entity Dflipflop is
port ( CK,D: in bit;           -- Clock and excitation inputs
       PRE,CLR: in bit;      -- Asynchronous preset and clear inputs
       Q,Qbar:out bit);     -- Complementary outputs
end Dflipflop;
architecture behavior of Dflipflop is
begin
    process(CK,PRE,CLR)           -- Flip flop reacts to CK, PRE or CLR
    begin
        if (PRE = '1') then      -- PRE given precedence over CLR and CK
            Q <= '1';           -- Set state to 1
            Qbar <= '0';
        elsif (CLR = '1') then   -- CLR given precedence over CK
            Q <= '0';           -- Reset state to 0
            Qbar <= '1';
        elsif CK'event and (CK='1') then -- Flip flop triggers when CK becomes 1
            Q <= D;             -- Data input transferred to the flip flop
            Qbar <= not D;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.43: HDL models of a rising-edge triggered D flip flop with asynchronous PRE and CLR inputs.

4.2.2.4 Edge-triggered Flip-flops

The flip-flops described in the previous sections all require both high and low levels of a clock pulse for proper operation, to separately enable master and slave latches, with the flip-flop output changing at the instant the clock pulse transitions to the level that enables the slave latch to change, making it appear as though the flip-flop responds only to that clock transition. The master–slave arrangement introduces a buffering mechanism to eliminate unstable transient conditions in sequential circuits with feedback elements, which were illustrated in Fig. 4.30. Another approach to solving the problem of unstable transients is to design the flip-flop circuitry so that it is sensitive to its excitation inputs only during rising or falling transitions of the clock. A circuit with this design feature is called *rising (or positive) edge triggered* if it responds to a $0 \rightarrow 1$ clock transition, or *falling (negative) edge triggered* if it responds to a $1 \rightarrow 0$ clock transition. The edge-sensitive feature eliminates unstable transients by drastically reducing the period during which the input excitation signals are applied to the internal latches. There are a number of common circuit structures used to create edge-triggered operation. The reader is referred to [1] for examples.

4.2.3 Latch and Flip-flop Summary

In the previous sections we have examined latch and flip-flop memory devices. Latch circuits are used primarily in situations where data are to be captured from signal lines and stored. The simple SR latch captures random pulses on its S and R inputs, since each pulse sets or resets the state of the latch. The gated SR and D latches change state only during times in which the latch is enabled. Therefore, gated latches are used to capture data that arrive and stabilize before the end of an enable pulse.

Flip-flops are used primarily for sequential circuit designs in which all state changes are to be synchronized to transitions of a clock signal. Most of these circuits utilize JK or D flip-flops, depending on which requires the smallest number of gates to derive the excitation inputs for each given design. SR flip-flops are rarely used, since JK flip-flops provide the same operating modes and add the additional toggle mode, eliminating the problem of having to avoid the condition $S = R = 1$. T flip-flops are used mainly in counter designs.-1

Device	Characteristic Equation
SR latch	$Q^* = S + \bar{R}Q$
Gated SR latch	$Q^* = SC + \bar{Q}R + \bar{C}Q$
D latch	$Q^* = DC + \bar{C}Q$
D flip-flop	$Q^* = D$
SR flip-flop	$Q^* = S + \bar{R}Q$
JK flip-flop	$Q^* = \bar{K}Q + J\bar{Q}$
T flip-flop	$Q^* = T\bar{Q} + \bar{T}Q$

Table 4.2: Summary of Latch and Flip-flop Characteristics

Many other kinds of storage devices may be used as memory elements in sequential circuits, several of which are magnetic cores, capacitors, magnetic films, superconductive cryotron elements, and electromechanical relays. Since these other types of memories are rarely used in today's computers, their explanation is beyond the scope of this text. The reader is referred to [5] and [6] for further information on various memory technologies and devices.

DRAFT

4.3 Registers

Thus far we have examined various binary memory elements in the form of latches and flip-flops. In the next few sections, we now investigate some simple *sequential logic modules*, in which combinational logic elements are combined with flip-flops to realize several commonly-used functions. First we will look at *registers*, which are used for storing binary data. Then we will connect the flip-flops to form a *shift register* that can manipulate data. Finally, we will examine *counters*, which are configurations of flip-flops that generate sequences of binary numbers. All of the modules in this chapter can be designed with straightforward interconnections of gates and memory elements. In the next chapter we will examine formal methods for designing sequential circuits.

Registers are among the most basic building blocks of digital computers and other digital systems, storing binary numbers and other information. An n -bit register stores information in n flip-flops that are controlled by a common clock, as illustrated in Figure 4.44a. The n input bits, and therefore the information represented by the n bits, are loaded concurrently into their respective flip flops on an active clock transition. As shown in Fig. 4.44b, a register may also include a common reset and/or preset signal to initialize the contents of the register. To simplify system-level diagrams, the logic symbols shown in Figs. 4.44c and 4.44d are often used to represent registers, with the latter a more compact form, showing data inputs and outputs as *buses* (i.e. bundles of signals).

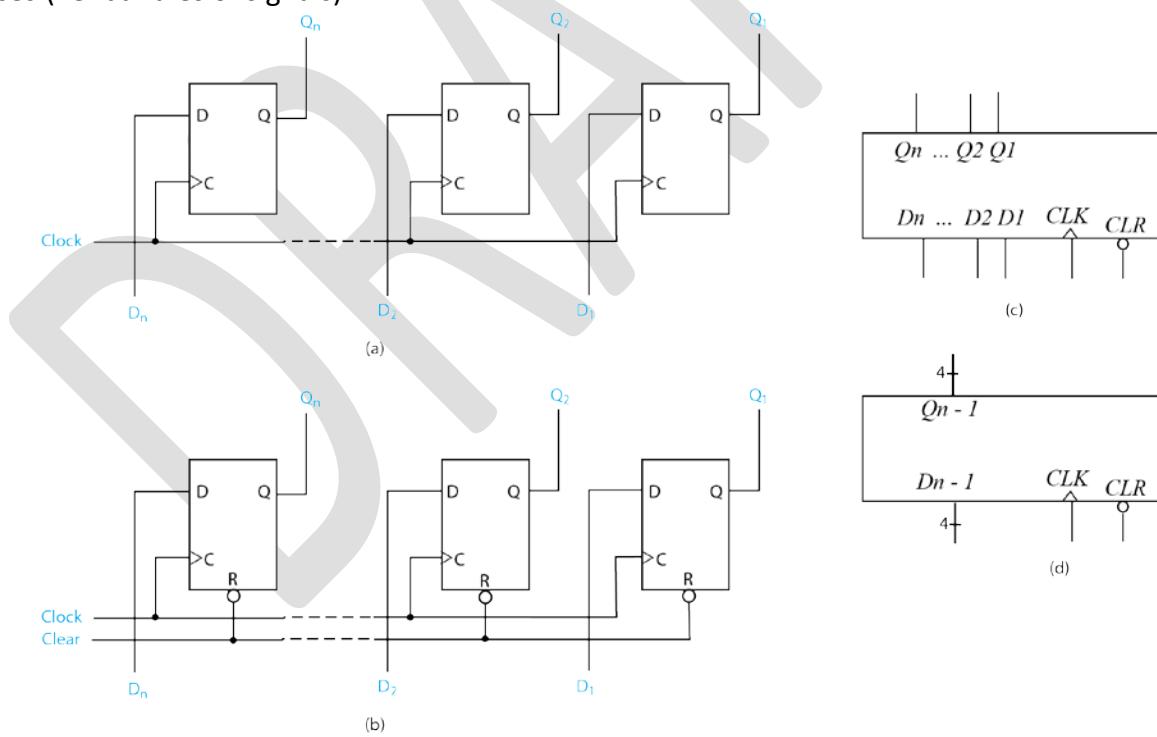


Figure 4.44: n -bit register. (a) logic diagram, (b) with asynchronous clear, (c) logic symbol, (d) compact logic symbol.

The behavior of the n -bit register of Figure 4.44b can be represented by the HDL models of Figure 4.45. Note that these models are nearly identical to that of the D flip-flop models in Fig.

4.36, with the only difference being that D and Q are bit vectors instead of single bits. In these models, a *generic parameter*, N, has been used as the most significant bit number in D and Q. This allows registers of different widths to be represented by the same model when the registers are instantiated in higher-level system designs. If the generic parameter is not defined when the register is instantiated, the specified default value of 8 will be used.

```
//N-bit register with active-low asynchronous clear
module NbitRegisterWclear (D,Q,CLK,CLR);
    input [N:1] D;                      //declare N-bit data input
    input CLK, CLR;                    //declare clock and clear inputs
    output reg [N:1] Q;                //declare N-bit data output
    parameter N = 8;                  //declare default value for N
    always @ (posedge CLK, negedge CLR) begin //detect change of clock or clear
        if (CLR==1'b0) Q = 0;           //register loaded with all 0's
        else if (CLK==1'b1) Q = D;     //data input values loaded in register
    end
endmodule
(a) Verilog model
```

```
-- N-bit register with asynchronous clear
entity RegisterN is
generic (N: integer := 8);          -- register width (default value 8)
port ( CLK,CLRB: in bit;           -- Clock and clear inputs
       D: in bit_vector(N downto 1); -- N-bit input data
       Q: out bit_vector(N downto 1));-- N-bit output data
end RegisterN;
architecture behavior of RegisterN is
begin
    process(CLK,CLRB)              -- Register reacts to CLK or CLR
    begin
        if (CLRB = '0') then        -- CLRB given precedence over CLK
            Q <= (others => '0');   -- Reset all register bits to 0
        elsif CLK'event and (CLK='1') then -- Register triggers when CK becomes 1
            Q <= D;                -- Data input transferred to the
register
        end if;
    end process;
end;
(b) VHDL model
```

Figure 4.45: HDL models of a rising-edge triggered N-bit register with asynchronous clear input.

The operation of a digital system is commonly described as sequences of *register transfers*, which indicate movements of data from one or more source registers to a destination register. Combinational logic may perform arithmetic or other operations on the data as it is being transferred. For example, assume that the outputs of two registers, A and B, are applied to the inputs of an adder, with the adder output applied to the inputs of register C, as illustrated in Fig. 4.46. When the clock input to register C is activated, we say that the register transfer $A + B \rightarrow C$ is performed, since register C captures the output of the adder, which is the sum of the values in registers A and B.

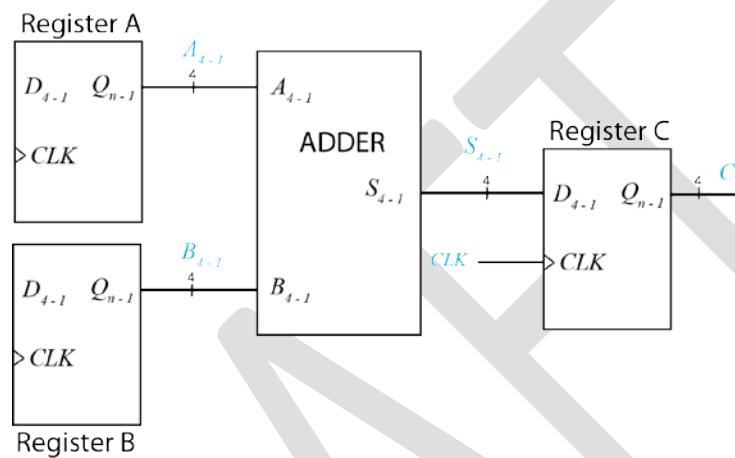


Figure 4.46: Register transfer $A + B \rightarrow C$ is performed on the rising edge of CLK .

To illustrate how a system can be described by a sequence of register transfers, consider the circuit of Fig. 4.47a. This circuit is often referred to as an *accumulator*, in that register A accumulates the sum of a sequence of numbers applied to input DIN . This circuit can perform two register transfers: $0 \rightarrow A$ when input CLR is activated, and $A + DIN \rightarrow A$ when CLK is activated. The timing diagram of Fig. 4.47b illustrates the operation of the accumulator in adding a sequence of numbers $\{5,4,1,2,2\}$. The following sequence of register transfers is performed at the indicated times.

T0:	$0 \rightarrow A$	
T2:	$A + DIN \rightarrow A$	$(0 + 5 = 5)$
T3:	$A + DIN \rightarrow A$	$(5 + 4 = 9)$
T4:	$A + DIN \rightarrow A$	$(9 + 1 = 10)$
T5:	$A + DIN \rightarrow A$	$(10 + 2 = 12)$
T6:	$A + DIN \rightarrow A$	$(12 + 2 = 14)$

At time T0, CLR is activated to initialize register A to 0. Then the first number to be added (5) is applied to DIN and at time T2, register transfer $A + DIN \rightarrow A$ changes A to $0+5=5$. Likewise, each remaining value in the sequence is applied to DIN , with $A + DIN \rightarrow A$ performed at

times T3-T6 to add each number to the total. The final sum in register A is equal to 14 (hexadecimal E).

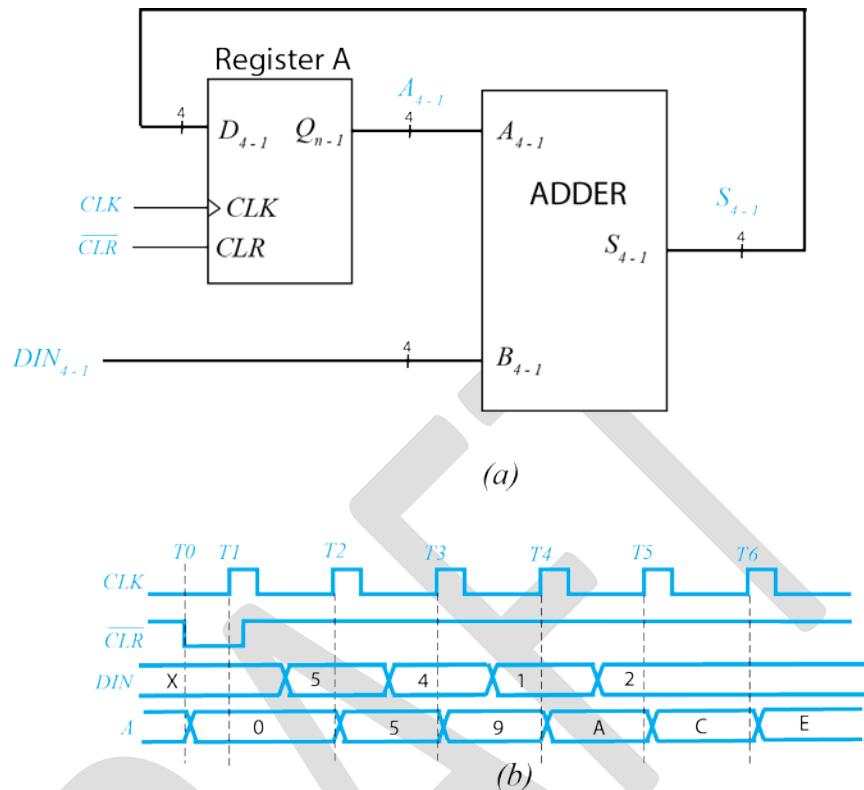


Figure 4.47: Accumulator circuit. (a) logic diagram, (b) timing diagram showing addition of a sequence of numbers applied to input DIN .

To facilitate the design of systems that contain multiple registers, registers often include a *clock enable* (CE) signal, as shown in Fig. 4.48. As can be seen in Fig. 4.48a, when $CE = 1$, the output of the AND gate is simply *Clock*, and therefore the clock signal is gated to the flip-flop CLK inputs; hence we say that the clock is “enabled”. When $CE = 0$, the output of the AND gate is 0, and therefore we say that the *Clock* is “disabled”, i.e. prevented from triggering the flip-flops. The behavior of the n -bit register of Figure 4.48 can be represented by the HDL models of Figure 4.49.

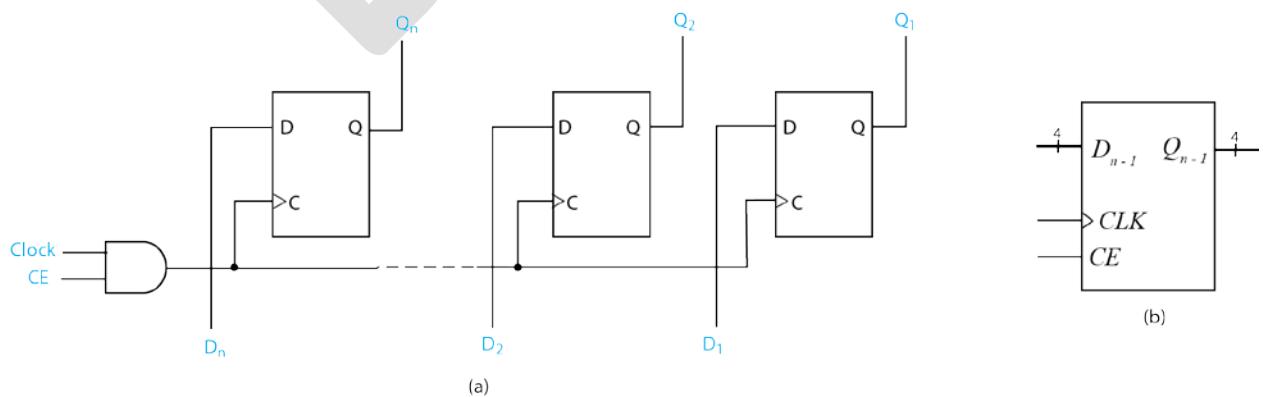


Figure 4.48: Register with a clock enable (CE) signal. (a) logic diagram, (b) logic symbol.

```
//N-bit register with clock enable
//
module NbitRegisterWclockEnable (D,Q,CLK,CE);
    input [N:1] D;                                //declare N-bit data input
    input CLK,CE;                               //declare clock and enable inputs
    output reg [N:1] Q;                           //declare N-bit data output
    parameter N = 8;                            //declare default value for N
    always @ (posedge CLK) begin                //detect positive edge of clock
        if (CE==1'b1) Q = D;                     //register loaded with data inputs
    end
endmodule
```

(a) Verilog model

```
-- N-bit register with clock enable and asynchronous clear
```

```
--
```

```
entity RegisterN is
generic (N: integer := 8);           -- register width, default value 8
port ( CLK,CE: in bit;              -- Clock and Clock Enable inputs
       D: in bit_vector(N downto 1);   -- N-bit input data
       Q: out bit_vector(N downto 1);  -- N-bit output data
);
end RegisterN;
architecture behavior of RegisterN is
begin
    process(CLK)                      -- Register reacts to CLK
    begin
        if CLK'event and (CLK='1') then -- Register triggers on rising edge of CLK
            if CE = '1' then          -- Register state changes if clock enabled
                Q <= D;               -- Data input transferred to the register
            end if;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.49: HDL models of a rising-edge triggered N-bit register with clock enable.

The use of registers with clock enable inputs allows a common clock signal to be applied concurrently to multiple registers, with only those registers whose clock enable inputs are active loaded on any clock transition. This is illustrated in Fig. 4.50a, in which a clock signal *CLK* is applied to three registers, A, B, and C, each of which has its own clock enable signal. On a rising clock transition, if *LDA* = 1 input data *DIN* is loaded into register A (register transfer

$DIN \rightarrow A$), if $LDB = 1$, DIN is loaded into register B (register transfer $DIN \rightarrow B$), and if $LDC = 1$, the output of the adder is loaded into register C (register transfer $A + B \rightarrow C$). The timing diagram of Fig. 4.50b shows a sequence of three register transfers that loads data value 5 into register A at time $T1$, loads 3 into register B at time $T2$, and then captures the sum of those values (8) in register C at time $T3$.

- T1: $DIN \rightarrow A$ ($5 \rightarrow A$)
- T2: $DIN \rightarrow B$ ($3 \rightarrow B$)
- T3: $A + B \rightarrow C$ ($5 + 3 = 8 \rightarrow C$)

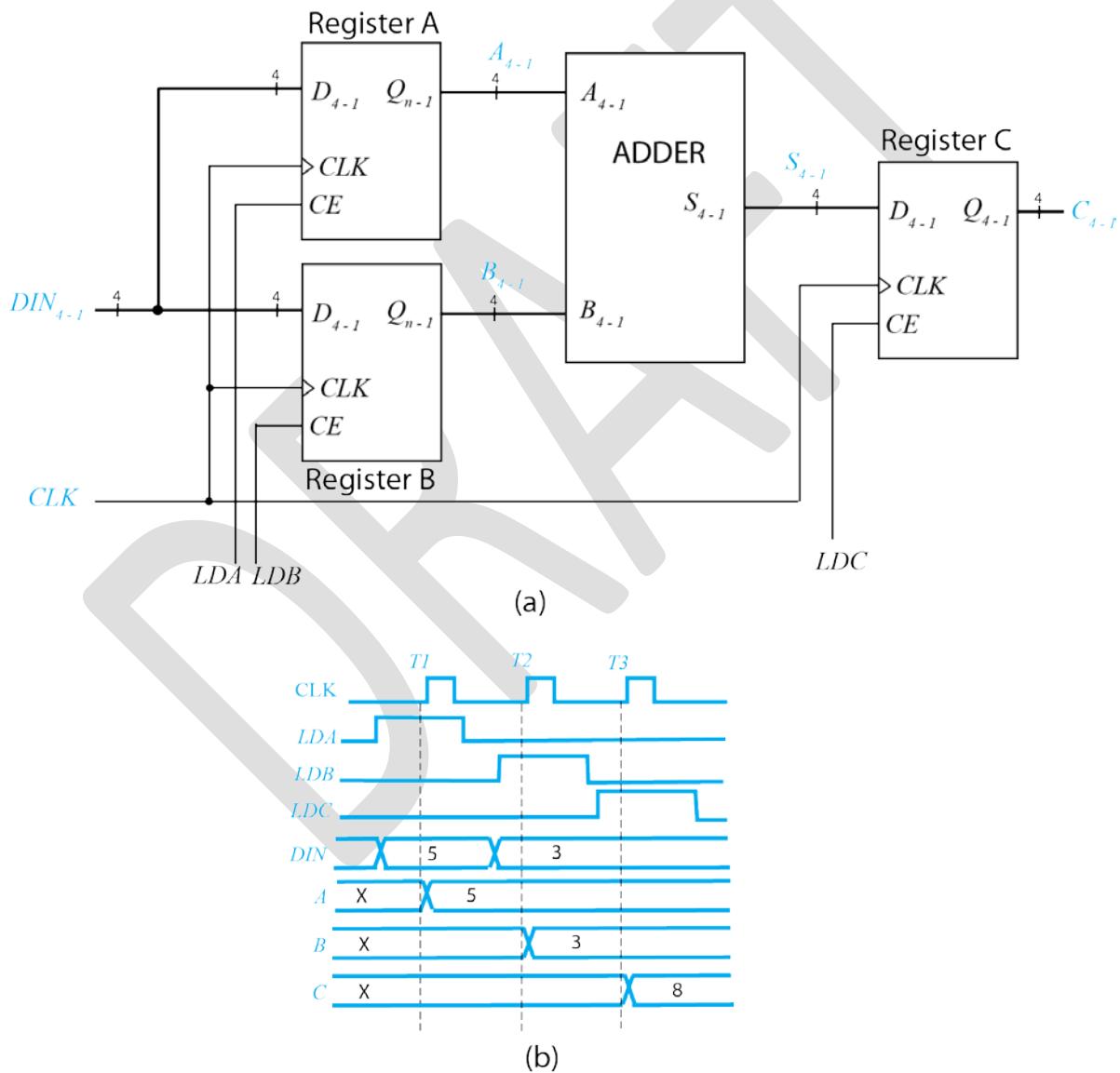


Figure 4.50: Clock enable signals used to select register transfers. (a) logic diagram, (b) timing diagram showing the addition $5 + 3 = 8$.

4.4 Shift Registers

A *shift register* is a sequential logic module constructed from flip-flops that stores and manipulates the bit positions of binary data by shifting the data bits to the left or right. Shift registers are used extensively in digital computers to manipulate data and in digital communication systems for serial transmission and reception of data. A typical shift register is illustrated in Fig. 4.51. The n -bit shift register in Fig. 4.51a holds n bits of binary data and is constructed from rising-edge-triggered D flip-flops, each holding one data bit. Since the Q output of each flip-flop is connected to the D input of the flip-flop to its right, an active transition ($0 \rightarrow 1$) of the shift control clock moves, or shifts, the binary data one position to the right. Thus, the binary bit in cell Q_i is transferred to cell Q_{i+1} . In other words, the binary data in the shift register is shifted one position to the right. The exception is the left-most bit Q_1 of the shift register, which is loaded with a value supplied on an external *Serial-in* signal. Consequently, we call this register a *serial-in, serial-out* shift register. The serial-in and serial-out terminals have been labeled in Fig. 4.51a.

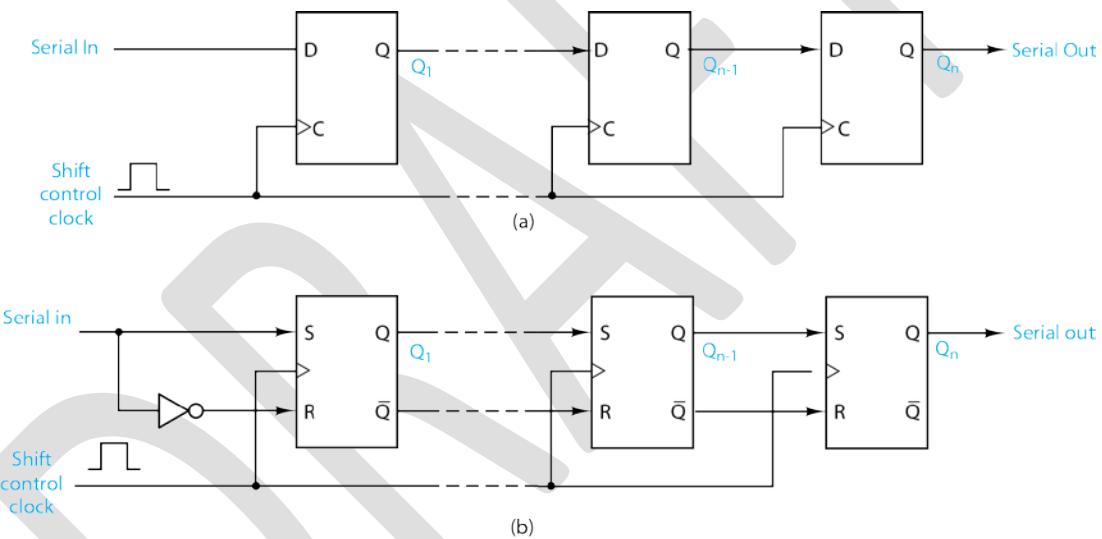


Figure 4.51: Serial-in, serial-out shift register. (a) Shift register with D flip-flops. (c) Equivalent shift register with SR flip-flops.

Figure 4.51b illustrates an alternate shift register implementation, using SR flip-flops. In this configuration, the S and R inputs of cell Q_{i+1} are driven by cell outputs Q_i and \bar{Q}_i , respectively. When $Q_i = 1$, then $S = 1$ and $R = 0$, setting $Q_{i+1} = 1$ on the next clock pulse, and when $Q_i = 0$, then $S = 0$ and $R = 1$, setting $Q_{i+1} = 0$ on the next clock pulse. Thus, the binary bit in cell Q_i is transferred to cell Q_{i+1} . Likewise, in the left-most bit Q_1 , when *Serial In* = 1 then $S=1$ and $R=0$, setting $Q_1 = 1$ and when *Serial In* = 0 then $S=0$ and $R=1$, setting $Q_1 = 0$.

In various applications, we may need the ability to access all n shift register bits in parallel and/or load data bits into all n bits of the shift register concurrently. This capability is available in the general purpose shift register circuit of Figure 4.52a, which can be represented by the logic symbol of Fig. 4.52b. The labels on the diagram indicate the following:

Parallel in ($D_i, i = 1, n$): one input line to each flip-flop with data to be entered into the register on a clock transition.

Parallel out ($Q_i, i = 1, n$): one output line from each flip-flop Q terminal

Serial in: data line feeding the first cell in the shift register; a bit enters on each *Shift* pulse

Serial out: data line from terminal Q_n of the last flip-flop in the register; a bit exits on each *Shift* pulse

Load/Shift: high to select parallel load and low to select serial shift.

Clock: load or shift operation occurs on the rising edge of the clock.

Clear: asynchronous clear signal to reset the states of the flip-flops.

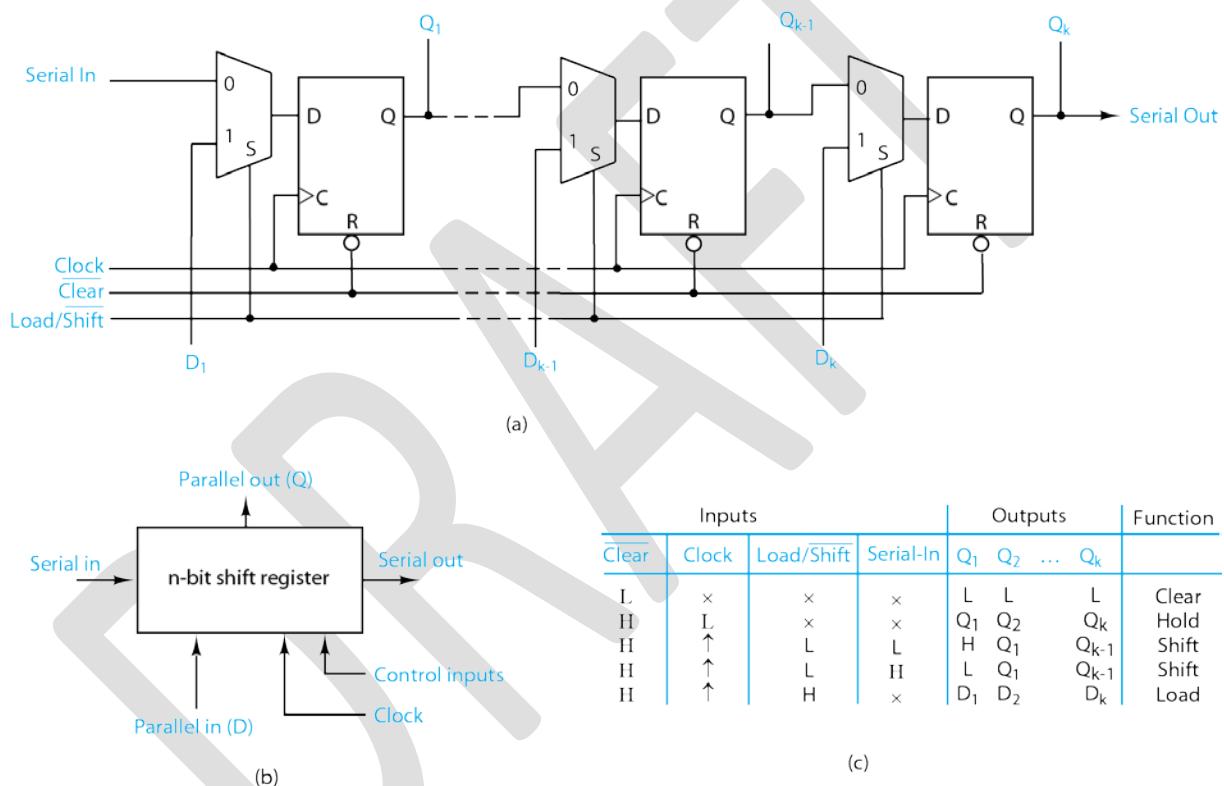


Figure 4.52: Generic shift register with serial and parallel inputs and outputs. (a) Logic diagram. (b) Function table. (c) Logic symbol.

The operation of this shift register is defined by the function table of Fig. 4.52c. When the asynchronous **Clear** signal activates (low), all bits are set to 0. If **Clear** is not active, then on the rising edge of the clock, the flip-flop inputs are selected by the multiplexers according to the value of **Load/Shift**. The parallel inputs are loaded into the shift register when **Load/Shift** = 1, and the register bits are shifted to the right, with **Serial In** loaded into the left-most bit, when **Load/Shift** = 0. This behavior is also represented by the HDL models of Fig. 4.53.

```
//N-bit shift register with clock enable and asynchronous clear
module NbitShiftRegister (CLK,CLR,LoadShift,SerialIn,D,Q);
    input CLK, CLR, LoadShift, SerialIn;
    input [N:1] D;
    output reg [N:1] Q;
    parameter N=8;
    integer i;
    always @ (posedge CLK, negedge CLR) //detect positive CLK edge or negative CLR edge
    begin
        if (CLR==1'b0) Q = 0; //load 0's in register if CLR is low
        else if (CLK==1'b1 & LoadShift==1'b1) Q = D; //load parallel data if LoadShift = 1
        else if (CLK==1'b1 & LoadShift==1'b0) begin //shift data bits right if laodshift = 0
            for (i = 2; i <= 8; i = i + 1) //bit shifting loop
                begin Q[i] = Q[i-1]; end
            Q[1] = SerialIn; //load serial input line in register location Q[1]
        end
    end
endmodule
```

(a) Verilog model

```
-- N-bit shift register with clock enable and asynchronous clear
entity ShiftRegisterN is
    generic (N: integer := 8); -- register width, default value 8
    port ( CLK, ClearN, Load_ShiftN: in bit; -- Clock and control inputs
           SerialIn: in bit; -- serial input
           D: in bit_vector(1 to N); -- N-bit input data
           Q: out bit_vector(1 to N); -- N-bit output data
    );
end ShiftRegisterN;
architecture behavior of ShiftRegisterN is
begin
    process(CLK, ClearN) -- Register reacts to CLK or ClearN
    begin
        if ClearN = '0' then -- asynchronous clear
            Q <= (others => '0'); -- all bits set to 0
        elsif CLK'event and (CLK='1') then -- register triggers on rising edge of CLK
            if Load_ShiftN = '1' then -- Load selected
                Q <= D; -- Data input transferred to the register
            else
                Q <= SerialIn & Q(1 to N-1); -- Shift data to right
            end if;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.53: HDL models of the shift register of Fig. 4.52.

The shift register of Fig. 4.52 may be operated in four modes: serial in and serial out, parallel in and serial out, serial in and parallel out, or parallel in and parallel out. The all-serial mode requires that the data be in a serial format synchronized with the *Shift pulse*. The shift register then serves as an n -pulse time delay for a stream of serial data.

To illustrate the use of shift registers in serial communication systems, Fig. 4.54 shows two systems communicating via the *Serial Peripheral Interface (SPI) Bus*. Each system contains an 8-bit shift register of the structure in Fig. 4.52. One system is designated the *Master* and the other the *Slave*. The Master provides a shift clock, SCLK, to control the shift operations in both systems. The MOSI signal is the serial output of the Master shift register (Master Out) and the serial input of the Slave shift register (Slave In). Likewise, the MISO signal is the Master Input and Slave output. To transmit a byte of data from the Master to the Slave, the data is loaded into the Master's shift register using the load operation. Then shift mode is selected for both devices and eight clock pulses are generated. On each clock pulse, the two registers are shifted to the right and, since the serial output of the Master is connected to the serial input of the Slave, one bit is transferred from the Master to the Slave. Thus, eight clock pulses will shift the entire 8-bit data value from the Master shift register to the Slave shift register, and vice versa. Note that this also transmits one byte from the Slave to the Master. The two systems can then read the received data bytes via the parallel outputs.

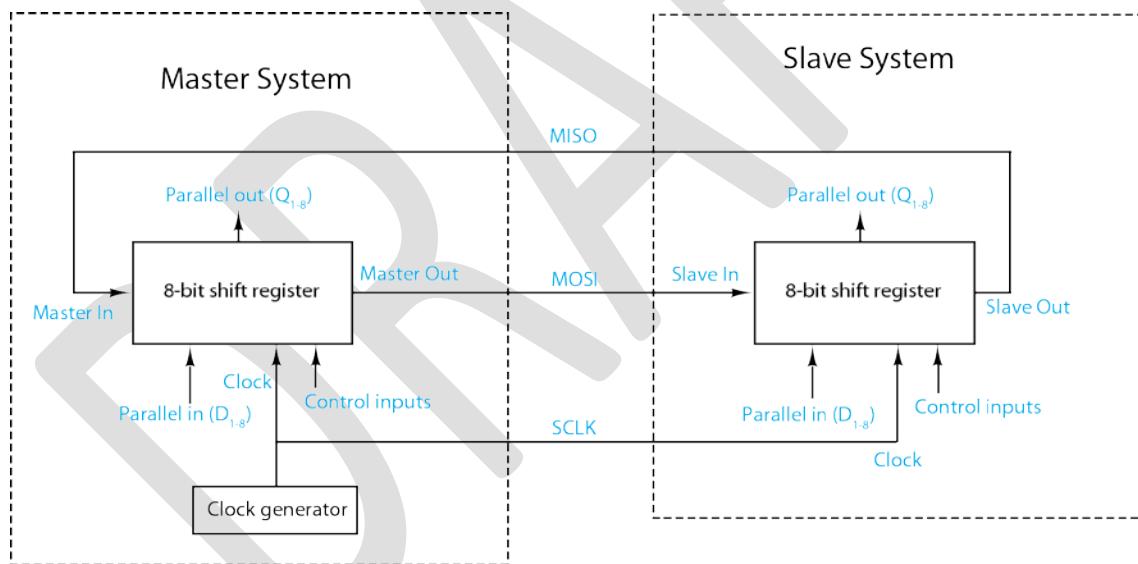


Figure 4.54. Two systems using shift registers to communicate via a *Serial Peripheral Interface (SPI) Bus*.

The exchange of data between Master and Slave is illustrated by the following example. Assume that the Master shift register initially contains the 8-bit pattern *abcdefghijklm*, and the slave shift register contains the 8-bit pattern *mnopqrstuvwxyz*. The following shows the contents of the two shift registers after each of the eight clock pulses is applied.

Clock Cycle	Master Shift Register	Slave Shift Register
-	<i>abcdefg</i>	<i>mnopqrst</i>
1	<i>tabcdefg</i>	<i>hmnopqrs</i>
2	<i>stabcdef</i>	<i>ghmnopqr</i>
3	<i>rstabcde</i>	<i>fghmnopq</i>
4	<i>qrstabcd</i>	<i>efghmnop</i>
5	<i>pqrstabc</i>	<i>defghmno</i>
6	<i>opqrstab</i>	<i>cdefghmn</i>
7	<i>nopqrsta</i>	<i>bcdefghm</i>
8	<i>mnopqrst</i>	<i>abcdefg</i>

Digital designers employ shift registers in many different applications. Various forms of the generic shift register module depicted in Fig. 4.52 find widespread usage in digital systems design. This generic design can be readily modified to support additional features, as will be demonstrated in the following example.

Example Design a 4-bit bidirectional, serial-in, parallel-out shift register with asynchronous common clear and four synchronous operating modes: hold (no state change), shift right, shift left, and parallel load.

The shift register should comprise four D flip-flops, with the following inputs/outputs to support the required functions: parallel outputs $Q_1Q_2Q_3Q_4$, parallel inputs $D_1D_2D_3D_4$, serial-in bit for each shift direction (*SerIn-Left* and *SerIn-Right*), control bits S_1S_0 to select a synchronous function, asynchronous \overline{CLEAR} input, and CLK input. A logic symbol for the shift register is presented in Fig. 4.55a and the required function table, derived from that of Fig. 4.52, is presented in Fig. 4.55b. The four synchronous operations are as follows.

$S_1S_0 = 00$: Selects the “hold” condition. $Q_1Q_2Q_3Q_4$ should not change on the CLK transition.

$S_1S_0 = 01$: Selects the “shift right” function. $Q_1Q_2Q_3$ should be transferred to $Q_2Q_3Q_4$, with *SerIn_Left* loaded as Q_1 .

$S_1S_0 = 10$: Selects the “shift left” function. $Q_2Q_3Q_4$ should be transferred to $Q_1Q_2Q_3$, with *SerIn_Right* loaded as Q_4 .

$S_1S_0 = 11$: Selects the “parallel load” function. $Q_1Q_2Q_3Q_4$ becomes $D_1D_2D_3D_4$.

Since there are four possible inputs to each D flip-flop, these functions can be implemented by using a four-to-one multiplexer to allow S_1S_0 to select the D input of each flip-flop, as illustrated in Fig. 4.55c. For each flip-flop k , the multiplexer selects Q_k for the “hold” function, Q_{k-1} for the right shift function, Q_{k+1} for the left shift function, and D_k for the parallel load function. For the left-most flip-flop, *SerIn_Left* is used as Q_{k-1} , and for the right-most flip-flop, *SerIn_Right*

is used as Q_{k+1} . Note that the hold operation differs from that implemented in Fig. 4.52. Instead of disabling the clock to prevent the register from changing, designers often prefer to allow the clock to run continuously, and simply load the flip-flop outputs back into their inputs to maintain the flip-flop state.

If desired, the design of Fig. 4.55c can easily be modified to include additional functions by using multiplexers with additional inputs.

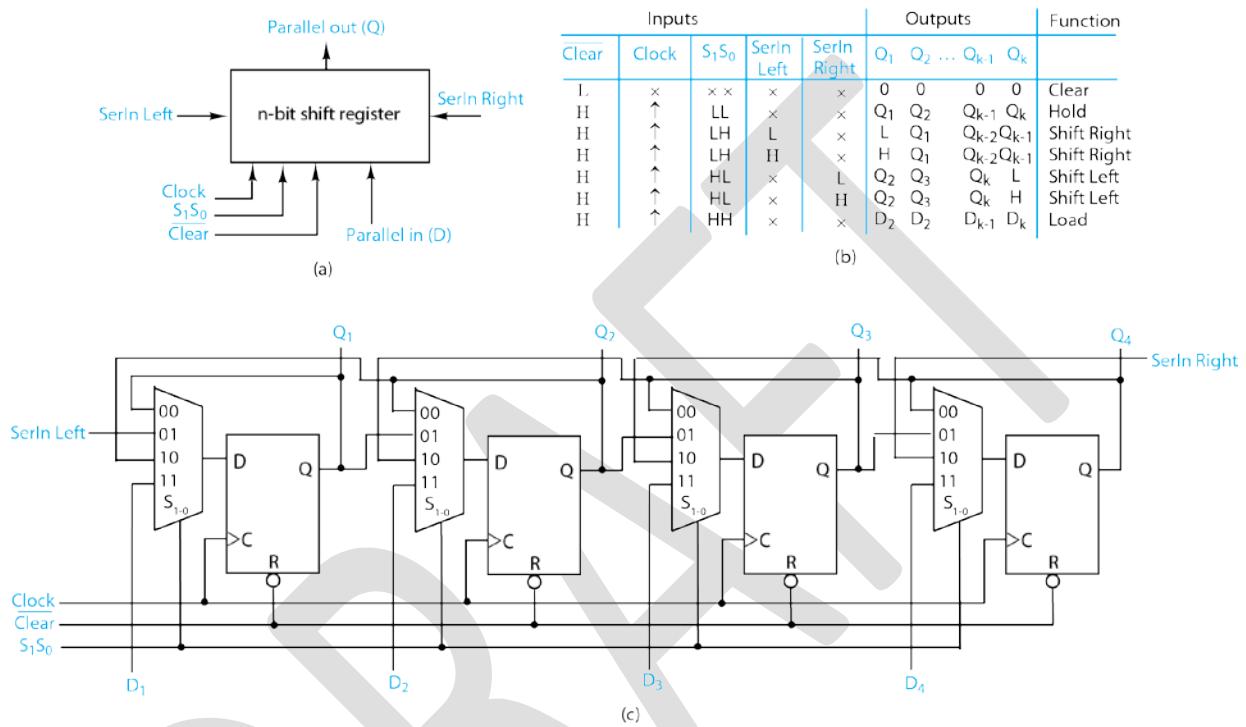


Figure 4.55: Bidirectional, serial-in, parallel-out shift register with asynchronous clear.
(a) Logic symbol. (b) Function table (c) Logic diagram.

4.5 Counters

Counters are a class of sequential logic circuits that tally a series of input pulses; the input pulses may be regular or irregular in nature. The counter is a fundamental part of most digital logic applications. It is used in timing units, control circuits, signal generators, and numerous other devices.

Counters may be categorized as synchronous/asynchronous and binary/nonbinary. Examples of each will be discussed in this section, including such features as enable, synchronous and asynchronous clear, synchronous and asynchronous load, and ripple carry output. Binary and modulo-N counters are illustrated, as are ring counter structures.

4.5.1 Synchronous Binary Counters

A synchronous n -bit binary counter constructed of clocked JK flip-flops is illustrated in Fig. 4.56a, with the flip-flops configured to use their toggle mode of operation. A binary counter of n flip-flops would normally begin in the all-zero state and sequence through the numbers $0,1,2,3,\dots,2^n - 1,0,1,2,\dots$, and so forth. In other words, the counter will have 2^n unique states and will repeat the states, as shown in the counting sequence in Fig. 4.56b, as long as clock pulses are applied. The circuit design of Fig. 4.56a is suggested by the state sequence listed in Fig. 4.56b. Note that bit Q_1 should be complemented on each count pulse. Bit Q_2 should be complemented on the next count pulse if $Q_1 = 1$. Bit Q_3 should be complemented on the next count pulse if both Q_1 and Q_2 are 1. In general, each bit Q_i should be complemented on the next count pulse if all bits Q_k , for $k = 1, \dots, i-1$, are at logic 1. Hence, a two-input AND gate may be used at each counter flip-flop to generate a toggle control signal for the next more significant bit in the counter chain, which is 1 when all of the less significant bits are 1. A counter flip-flop and its associated control circuitry are sometimes called a *counter stage*. It is interesting to note that the first k stages of an n -bit binary counter act as a k -bit binary counter, cycling through states $0,1,2,3,\dots,2^k - 1,0,1,2,\dots$.

Under normal operating conditions, the J and K inputs to each flip-flop should remain stable at either logic 1 or 0 while the count pulse undergoes its $0 \rightarrow 1 \rightarrow 0$ transitions. A logic 1 on the *Clear* control line will force all counter outputs to logic 0 and hold them there until the *Clear* line is returned to logic 0 (its normal logic value). The *Inhibit* control signal is used to block the count pulses and leave the counter in its current state, if such data-hold behavior is required for a particular application.

When the counter reaches the all-one state, the *Overflow* signal will go high. In some counter designs, this overflow signal is used to drive cascaded counter modules to produce larger counter word lengths. In these cases, the overflow signal is commonly called the *ripple carry-out* (RCO).

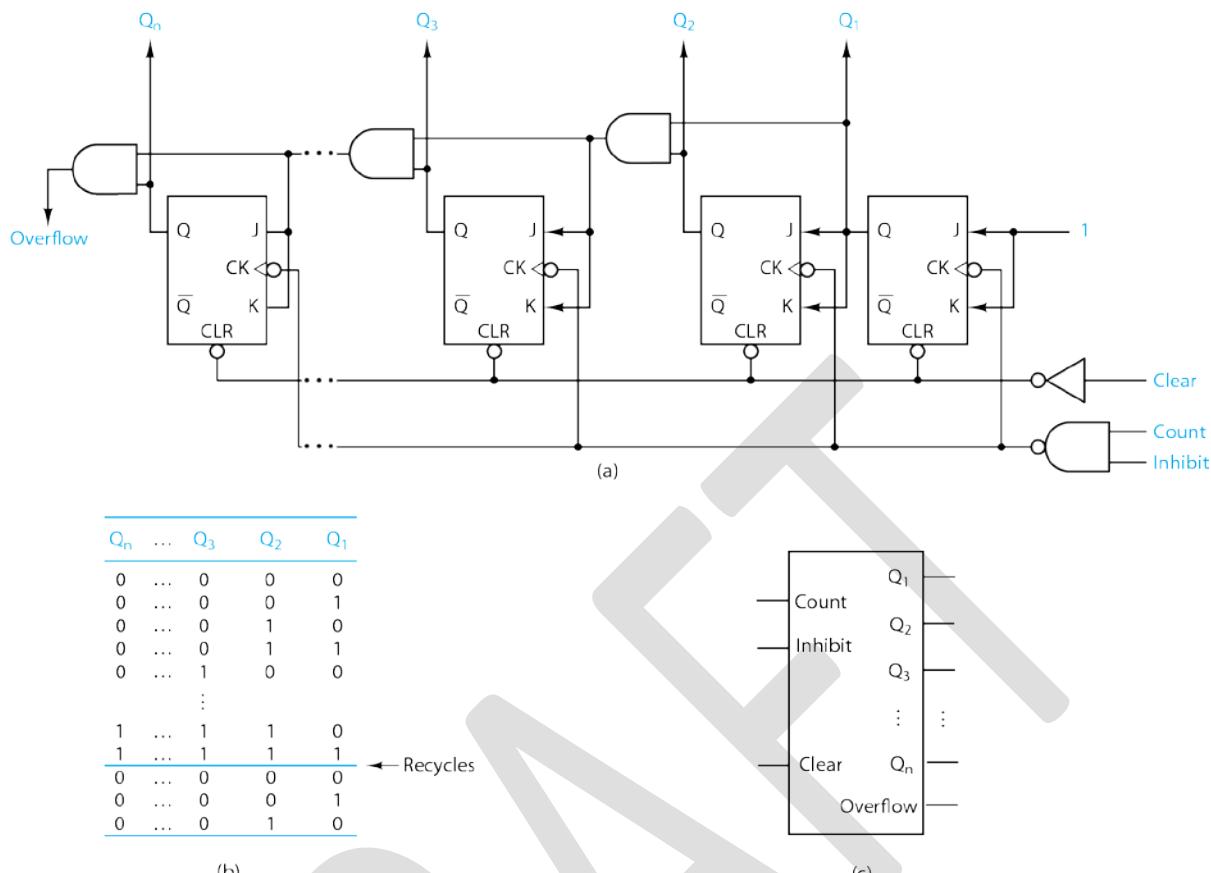


Figure 4.56: Synchronous binary counter. (a) Logic diagram. (b) State sequence. (c) Logic symbol.

For $N=3$, there will be eight states (2^3) and with a state diagram as in Fig. 4.57. HDL behavioral models of an N -bit binary counter are given in Fig. 4.58.

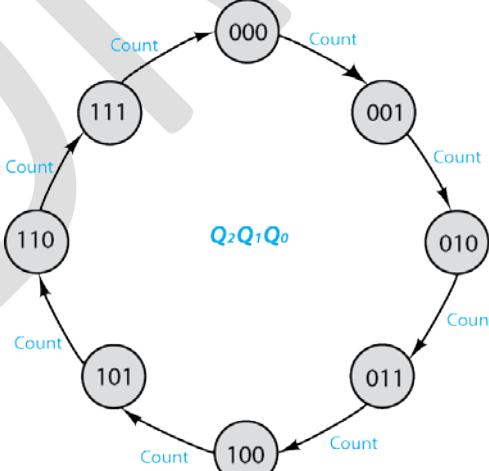


Figure 4.57: Three-bit binary counter state diagram

```
// Behavioral description of an N-bit binary counter
//
module NbitBinaryCounter (COUNT, CLEAR, Q);
input COUNT, CLEAR;                                //define input variables
output reg [N-1:0] Q;                            // Q is defined as an N-bit output register
parameter N=3;                                     //Define default value of N=3
always @ (posedge COUNT, negedge CLEAR)           //Detect input variable changes
  if (CLEAR==0) Q <= 0;                           // Q loaded with all 0's on negative edge of CLEAR
  else begin                                       //Begin counting
    if (Q == 2**N - 1)                            //Check for maximum count
      Q <= 0;                                     //Once Q = all 1's it returns to all 0's
    else
      Q <= Q + 1'b1;                            //Q is incremented on positive edge of CLOCK
  end
endmodule
```

(a) Verilog

```
-- Behavioral description of an N-bit binary counter
library ieee; use ieee.numeric_bit.all;
entity SyncBinCntN is
  generic (N: integer := 4);
  port (Q: out bit_vector(N downto 1);          -- N-bit output
        Overflow: out bit;                      -- Overflow indicator
        Clear: in bit;                          -- Async clear
        Count: in bit;                          -- Count pulse
        Inhibit: in bit);                      -- Inhibit counting
end SyncBinCntN;
architecture Behavior of SyncBinCntN is
  constant MaxCnt: UNSIGNED(N downto 1) := to_unsigned(2**N-1,N);
  signal Cnt: UNSIGNED(N downto 1);
  signal Clk: bit;
begin
  Clk <= Count nand Inhibit; -- enable count pulse on ff clock
  Counter:
    process (Clear,Clk) -- trigger with Clear or Clk
    begin
      if Clear = '1' then                   -- async clear active
        Cnt <= (others => '0');          -- reset to all 0's
      elsif Clk'Event and Clk = '0' then   -- clock 1->0
        Cnt <= Cnt + 1;                  -- increment count
      end if;
    end process;
  Q <= bit_vector(Cnt);                  -- drive outputs
  Overflow <= '1' when Cnt = MaxCnt else '0'; -- overflow on max count
end;
```

(b) VHDL model

Figure 4.58: HDL models of an n-bit synchronous binary counter.

Using the same design method as for the binary counter of Fig. 4.56, we can also create a *down*, or *backward*, counter whose state transitions are reversed from those of the standard counter, which is also known as an *up*, or *forward*, counter. Examine the state tables of Fig. 4.59a. The down counter behaves as a complemented up counter, that is, bit Q_i should be complemented on the next count pulse if all bits Q_k , for $k = 1, \dots, i-1$, are at logic 0. Hence, a synchronous down counter may be constructed using clocked JK flip-flops, as indicated in Fig. 4.59b, with J and K inputs both set to 1 if the outputs of the previous flip-flops are all at logic 0 (i.e. the complemented outputs \bar{Q}_i are all logic 1.) The *Clear* control signal drives the counter to the 0 state, and the *Count* control signal must be logic 1 in order for the clock pulses to cause counter state changes.

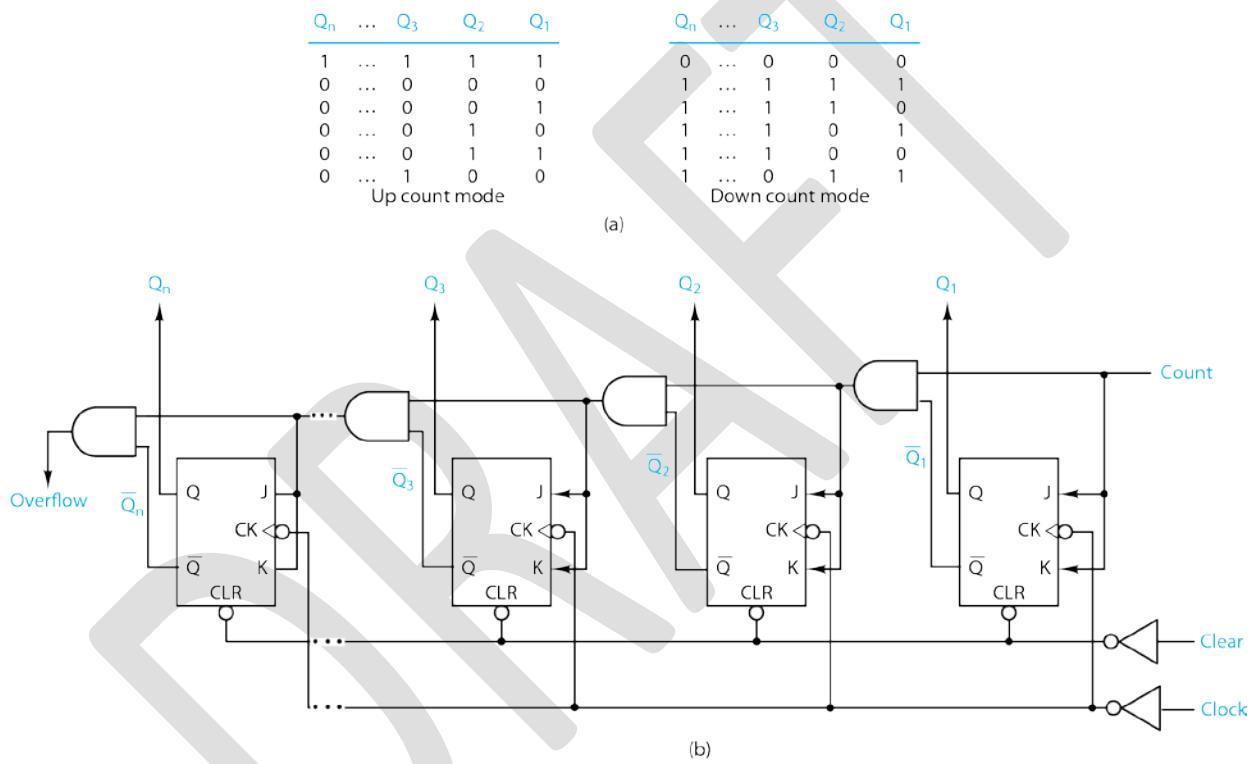


Figure 4.59: Asynchronous down counter. (a) State sequences. (b) Logic diagram.

Example Design an n-bit synchronous binary counter that can function in both the up and down modes of operation, as selected by a control line *Up/down*.

Many digital systems require a counter design that can function in both the up and down modes of operation. Referring to the up and down counter designs of Figs. 4.56 and 4.59, respectively, the combination up/down synchronous counter shown in Fig. 4.60 can be derived. This counter is either in the up or down mode since the down control signal is the complement

of the up control signal. Therefore, the up/down control line is labelled *Up/down*. In the up mode, the Q outputs of the flip-flops control the J and K terminals of flip-flops higher in the chain, using the upper AND gates. In the down mode, the \bar{Q} outputs of the flip-flops fill this role, using the lower AND gates.

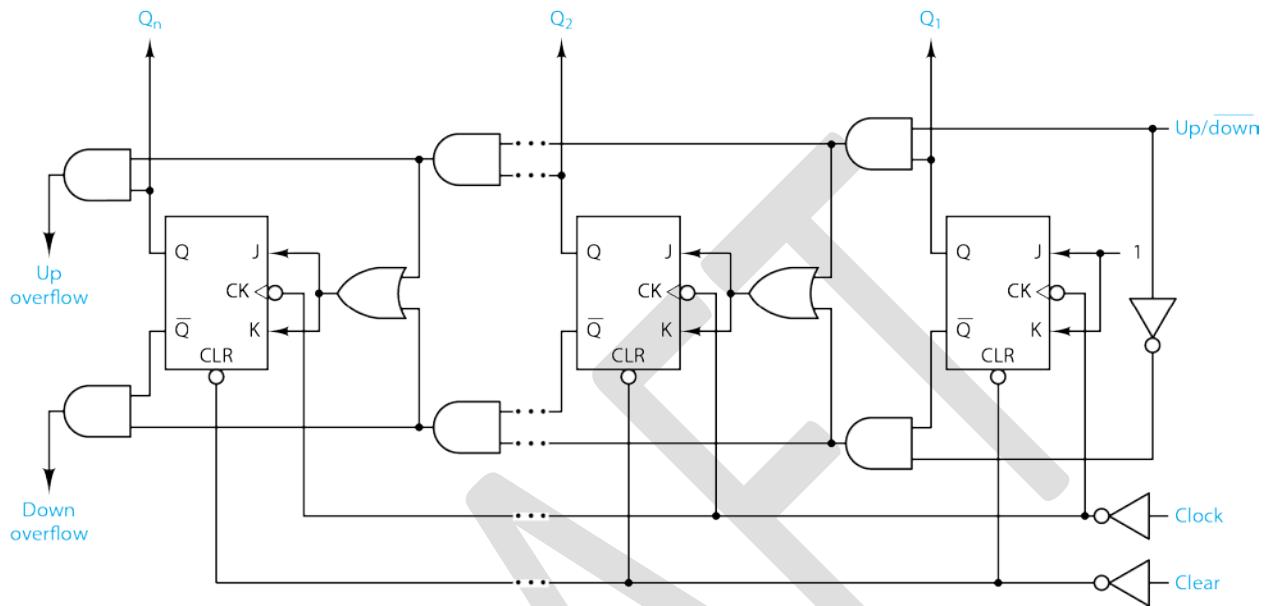


Figure 4.60: Synchronous up/down counter.

4.5.2 Asynchronous Binary Counters

An asynchronous binary counter is one whose state changes are not controlled by a synchronizing clock pulse. By eliminating the requirement for clock synchronization, a reduced amount of circuitry can be used to implement the counter. Consider the synchronous design of Fig. 4.56a. One may eliminate the AND gates in the synchronous design by observing the counter state transitions of Fig. 4.56b from another viewpoint. Counter stage Q_i is complemented each time state Q_{i-1} makes a $1 \rightarrow 0$ transition; as in the synchronous counter, stage Q_1 is complemented on each count pulse. A counter based on these observations is demonstrated in Fig. 4.61a. An asynchronous common *Clear* signal may be used to initialize the counter to the 0 state, and the *Count* control signal is held at logic 1 for counting; logic 0 on *Count* inhibits all counting and leaves the counter in a constant state; this is the data-hold mode.

To see the effects of using asynchronous, rather than synchronous, operation, let us examine the behavior of the asynchronous binary counter as overflow occurs. Just before overflow, all counter stages are at logic 1 as shown in Fig. 4.61b. After the clock pulse falls, the flip-flop of counter stage Q_1 responds in t_{PHL} seconds. Each stage then follows in a similar manner until all counter stages have reached the logic 0 state. It is important that the reader note the transient condition produced by this count sequence. Instead of the desired state

change $(2^n - 1)_{10}$ to $(0)_{10}$, the counter has passed through the following state sequence:

$$(2^n - 1)_{10} \rightarrow (2^n - 2)_{10} \rightarrow (2^n - 4)_{10} \rightarrow (2^n - 8)_{10} \dots \rightarrow (2^{n-1})_{10} \rightarrow (0)_{10}$$

Although these transitions are rapid, they can generate unwanted transient conditions if the counter outputs are used to drive a combinational logic circuit. See the discussions of hazards in Chapter 2. Because of the transient behavior described here, an asynchronous counter is sometimes called a *ripple* counter, with stages changing states, one after the other, from left to right.

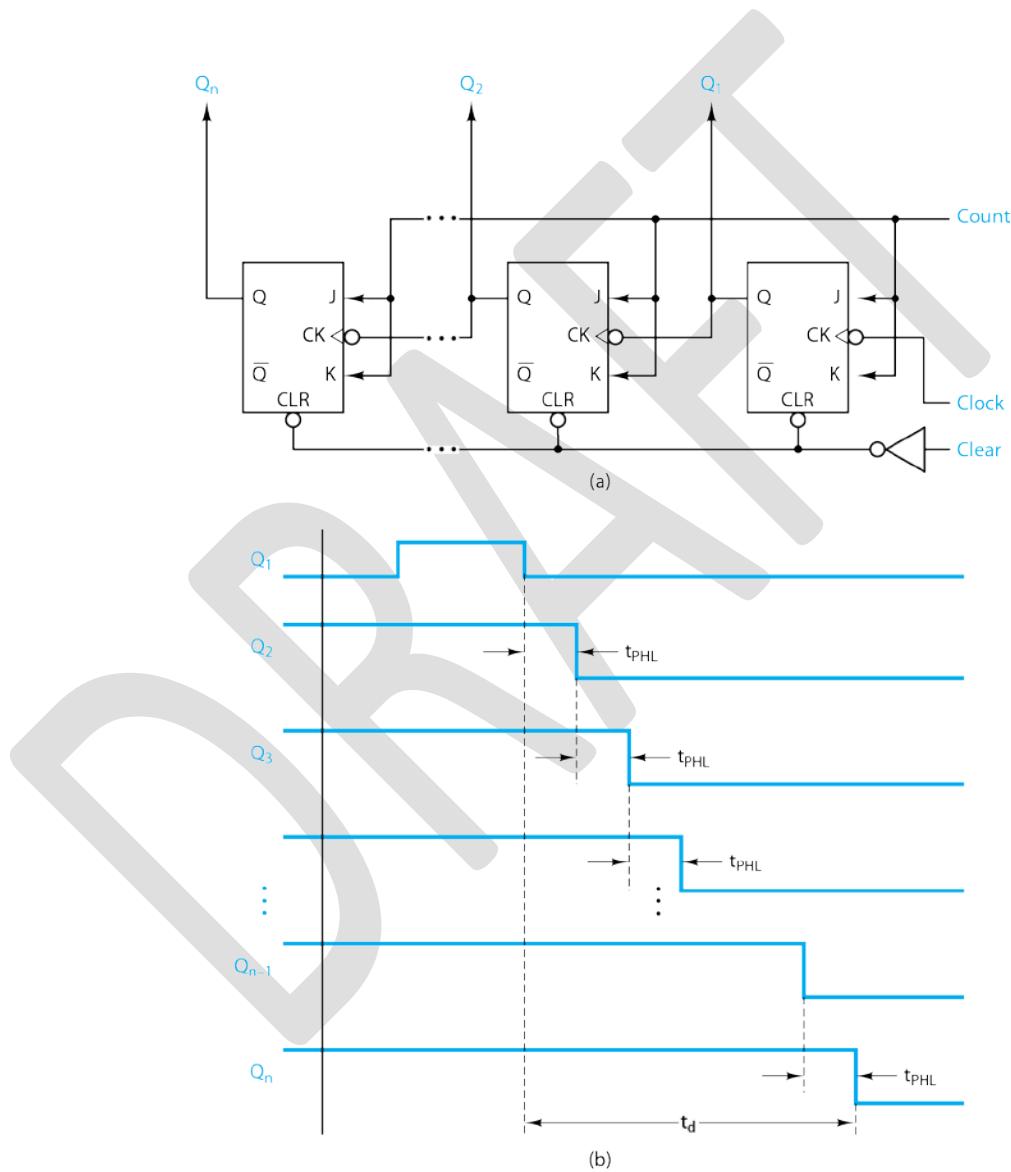


Figure 4.61: Asynchronous binary counter. (a) Logic diagram. (b) Timing diagram.

4.5.3 Modulo- N Counters

Many occasions arise in the design of digital systems in which a counter is needed that can count from state 0 through state $N-1$ and then cycle back to state 0; such counters are said to be *modulo- N* counters. The most common modulo- N counters are the binary ones previously discussed. For binary counters, N is equal to 2^n , where n is the number of counter stages. Counters with other values for N are also very useful. For example, $N=10$ (decade) counters are frequently encountered in digital systems design.

A synchronous BCD (binary coded decimal) counter is a modulo-10, or *decade*, counter. As illustrated in Fig. 4.62a, the BCD counter must behave like a binary counter until state 9 is reached. At this point the control circuitry must prepare the flip-flop inputs so that the next clock pulse will force the counter back to state (0000) instead of allowing the next binary-counter state (1010) to be reached. The BCD counter circuit shown in Fig. 4.62b can be derived by examining the counting sequence of Fig. 4.62a, identifying states in which each bit toggles, and then using this information to determine the J and K inputs of each flip-flop.

- Q_1 should be complemented on each clock pulse.
- Q_2 should be complemented on the next clock pulse when $Q_4 = 0$ and $Q_1 = 1$.
- Q_3 should be complemented on the next clock pulse when $Q_2 = 1$ and $Q_1 = 1$.
- Q_4 should be complemented on the next clock pulse when $Q_3Q_2Q_1 = 111$ or $Q_4Q_1 = 11$.

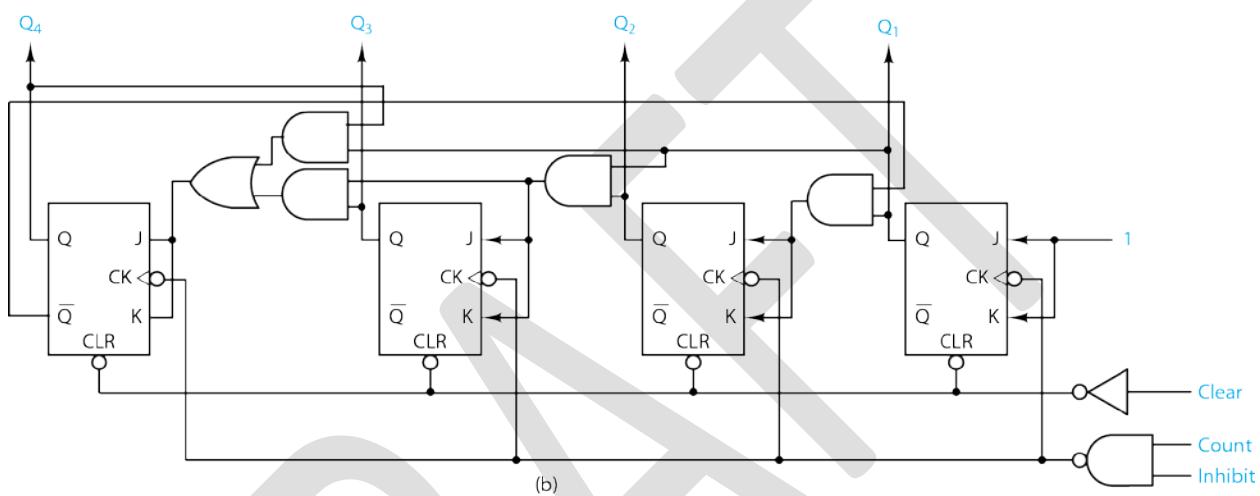
Note that the conditions for toggling stages Q_2 and Q_4 are more complex than for the simple binary counter, since we require that the count return to state (0000) from state (1001).

The clear and counting operations of the counter are similar to those of the synchronous binary counter in Fig. 4.56, with the exception of the states of the count sequence.

HDL behavioral models of a BCD counter are shown in Fig. 4.63.

Q_4	Q_3	Q_2	Q_1
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
0	0	0	0
0	0	0	0

(a)



(b)

Figure 4.62: Synchronous BCD (decade) counter. (a) Counting sequence. (b) Logic diagram.

```
// Behavioral description of a BCD counter
module BCDCOUNTER (COUNT, CLEAR, Q);
    input COUNT, CLEAR; //Define input variables
    output reg [3:0] Q; //Define output Q as an N-bit output register
    always @ (posedge COUNT, negedge CLEAR)
        if (CLEAR==0) Q <= 0; //Q is loaded with 0
        else
            begin
                if (Q == 4'b1001) //Check for BCD 9
                    Q <= 0; //Once Q = 1001 it returns to 0
                else
                    Q <= Q + 1'b1; //Q is incremented to next value
            end
    endmodule
```

(a) Verilog model

```
-- Behavioral description of a BCD counter
library ieee; use ieee.numeric_bit.all;
entity SyncBCDCounter is
    port (Q: out bit_vector(4 downto 1);
          Clear: in bit;
          Count: in bit;
          Inhibit: in bit);
end SyncBCDCounter;
architecture Behavior of SyncBCDCounter is
    signal Cnt: UNSIGNED(4 downto 1);
    signal Clk: bit;
begin
    Clk <= Count nand Inhibit;
    Q <= bit_vector(Cnt);
    process (Clear,Clk)
    begin
        if Clear = '1' then
            Cnt <= "0000";
        elsif Clk'Event and Clk = '0' then
            if Cnt = "1001" then
                Cnt <= "0000";
            else
                Cnt <= Cnt + 1;
            end if;
        end if;
    end process;
end;
```

(b) VHDL model

Figure 4.63 HDL models of a synchronous BCD counter

The synchronous decade counter of Fig. 4.62 can be replaced by an asynchronous counter in many applications. An asynchronous, or *ripple*, decade counter can be designed by modifying the asynchronous binary counter of Fig. 4.61a. Suppose we add a logic circuit to detect state 10 and use this signal to drive the counter immediately to state 0 via the common reset line. State 10 has the binary value $(Q_4Q_3Q_2Q_1) = (1010)$. We can decode state 10 using a two-input AND gate with inputs Q_4 and Q_2 . As can be seen in the counting sequence in Fig. 4.62a, the condition $Q_4 = Q_2 = 1$ is unique since the counter will cycle through states 0, 1, 2, ..., 9 before reaching state 10. No other state in the sequence satisfies the condition $Q_4 = Q_2 = 1$. The logic diagram for the asynchronous decade counter is shown in Fig. 4.64a. Let us now examine the transient behavior of this ripple counter design.

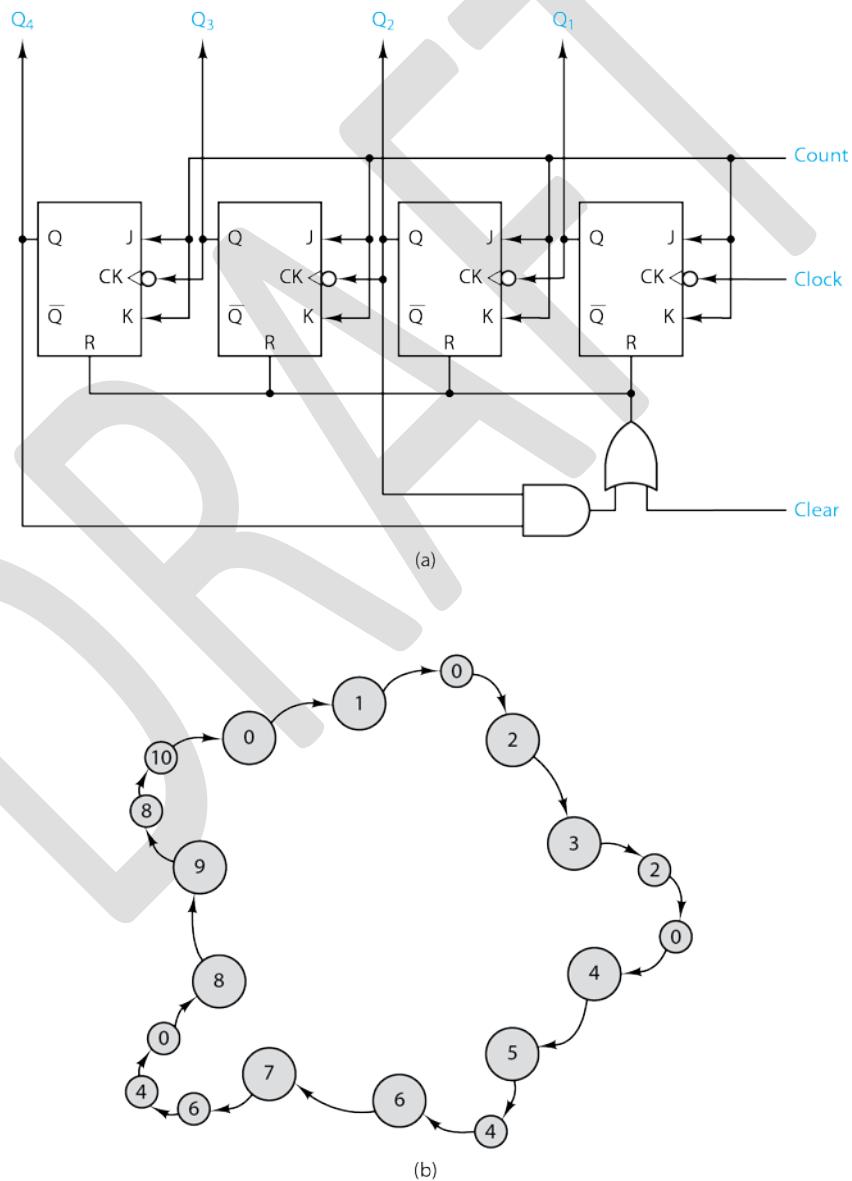


Figure 4.64: Asynchronous BCD counter. (a) Logic diagram. (b) State diagram.

In Fig. 4.64b, a state diagram is used to describe the behavior of this circuit. The timing diagram presented earlier, in Fig. 4.56b, showed how transient states occur during the counting process. In the state diagram of Fig. 4.64b the smaller circles represent the transient states. The worst-case transient condition occurs on the transition from state 7 to state 8. Due to the rippling effect, three intermediate states are observed. Now consider the transition from state 9 to state 0. The ripple effect will cause the counter to briefly enter state 8 and then state 10. However, the feedback circuit will detect state 10 and generate a common clear signal to reset all the counter stages to logic 0.

The design technique used in the asynchronous BCD counter of Fig. 4.64 may be generalized for exploitation in any general modulo- N counter, as shown in Fig. 4.65.

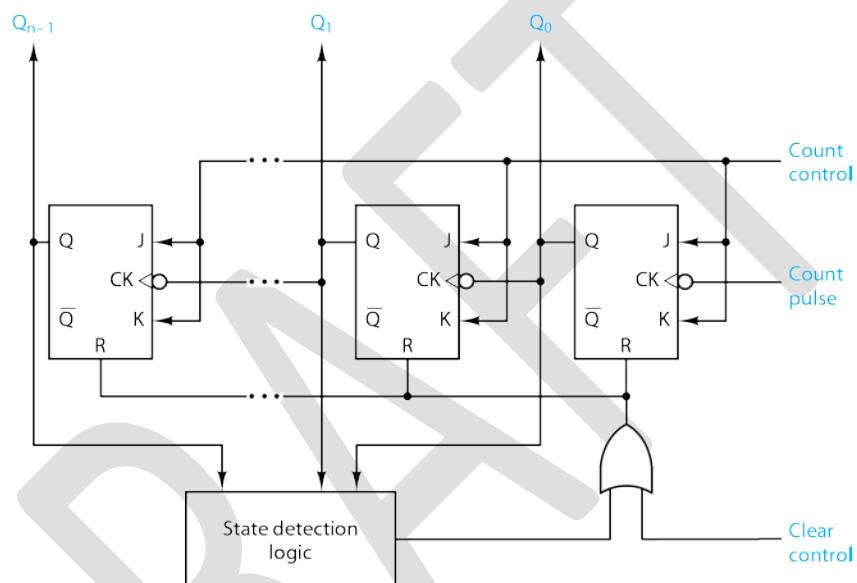


Figure 4.65: General modulo- N asynchronous counter.

The state-detection logic consists of an AND gate with appropriate inputs to detect the state N , the modulus of the counter. The number n of counter stages needed is determined by the relation

$$2^{n-1} < N < 2^n$$

This relation assumes that N is not a power of 2 because the feedback network is not needed in those cases. The generalized counter of Fig. 4.65 requires an asynchronous common clear control signal. The stable states of the counter will be $0, 1, 2, \dots, N-1$. The state detection logic will sense state N and immediately force the counter past state N to state 0. Thus, these asynchronously resetting counters will always have a transient state N .

Example Design a modulo-13 counter using the binary counter circuit of Fig. 4.56.

Considering the n -bit synchronous binary counter of Fig. 4.56, since $N = 13$, $2^3 < N = 13 < 2^4$, and therefore the number of counter stages must be $n = 4$. State 13

represents the counter value

$$(Q_4, Q_3, Q_2, Q_1) = (1, 1, 0, 1)$$

In normal operation, the counter will sequence through states 0, 1, ..., 12 before reaching state 13. Therefore, we can use the fact that state 13 will be the first occurrence of the logic condition $Q_4 = Q_3 = Q_1 = 1$. Therefore, we can use a 3-input AND gate to decode state 13 as shown in Fig. 4.66a. When state 13 occurs, the AND gate generates a *Reset* control signal and applies it to the counter's *Clear* input. When the counter has stabilized in state 0, the reset control signal is returned to its inactive condition, logic 0.

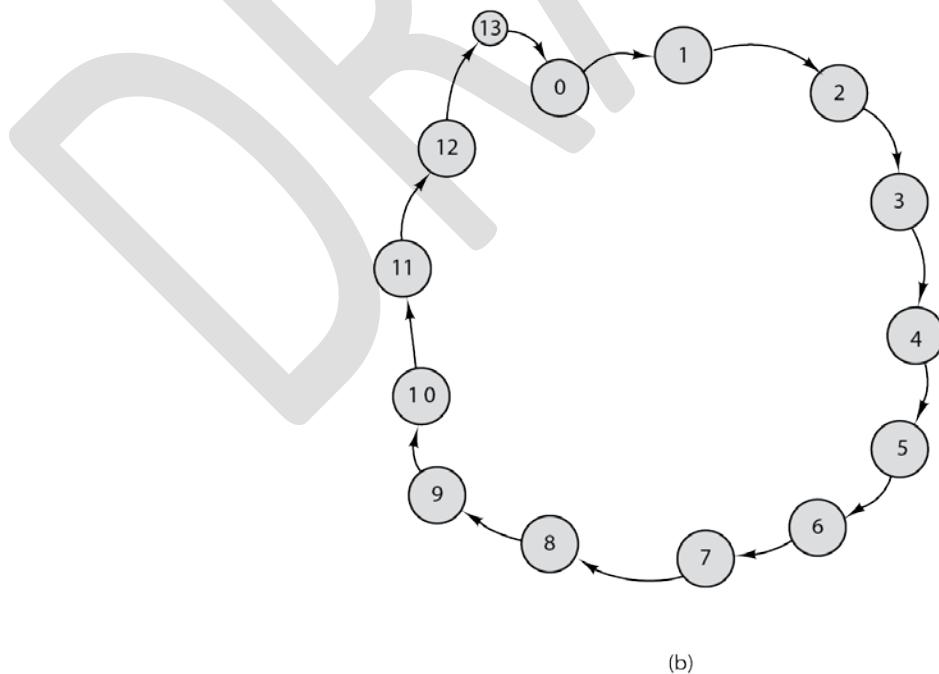
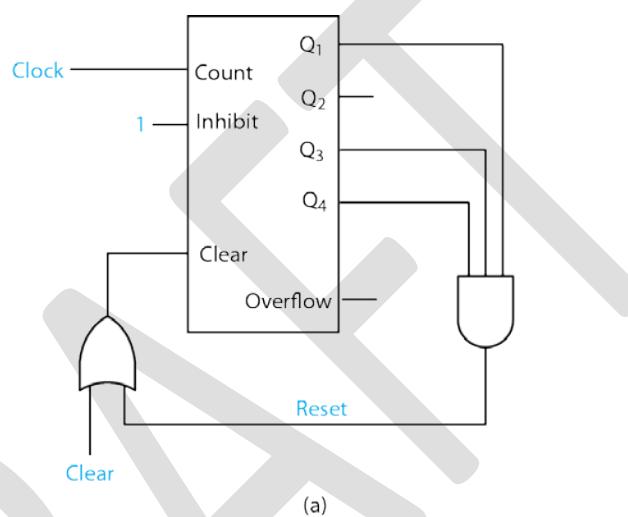


Figure 4.66: Asynchronously-resetting modulo-13 counter. (a) Logic diagram. (b) State diagram.

The state diagram of Fig. 4.66b illustrates the transition states for this counter. Compare Fig. 4.66b with Fig. 4.64b. Since the 4-bit binary counter module is synchronous, note that there are no unstable states until the state detection logic changes state 13 in Fig. 4.66b into a transient state and jumps directly to state 0.

4.5.4 Ring and Twisted-Ring Counters

Now let us examine another class of counters. The shift registers that we covered earlier in this chapter may be used as counters in special circumstances. Counters constructed using shift registers with feedback are commonly called *ring counters* and *twisted-ring counters*. We will now investigate the behavior of these devices.

A *ring counter* is a sequential device that has one shift-register flip-flop per counter state. The output of the shift register's serial output is fed back to the register's serial input pin. The resulting circuit circulates a bit pattern around the register. If we initialize the register so that it has a single logic 1 in its first flip-flop and logic 0 in all the others, the device will circulate the single logic 1 around its loop of flip-flops as illustrated in Fig. 4.67. Let n be the number of flip-flops and hence the number of states in the counter. The shift-register flip-flops are labeled Q_1, Q_2, \dots, Q_n . The operation of the counter begins with a pulse in control line Initialize. This drives flip-flop Q_1 high and Q_2, Q_3, \dots, Q_{n-1} , and Q_n low. At this point a single logic 1 is residing in flip-flop Q_1 . On the next falling edge of the input signal Clock, the logic 1 is transferred from flip-flop Q_1 to flip-flop Q_2 . The process continues until the logic 1 reaches the end of the shift register, flip-flop Q_n . On the falling edge of the next clock pulse, the logic 1 is transferred by the feedback line to the first flip-flop in the shift register, Q_1 . And then the process is repeated. In other words, the logic 1 circles through the shift register every n clock pulses. So the ring counter has one unique state for each flip-flop. The state sequence may be described in decimal values as $(Q_n, Q_{n-1}, \dots, Q_1)_2 = 1_{10}, 2_{10}, 4_{10}, 8_{10}, \dots, (2^{n-1})_{10}$. For example, a 5-bit ring counter will sequence through states 1, 2, 4, 8, and 16.

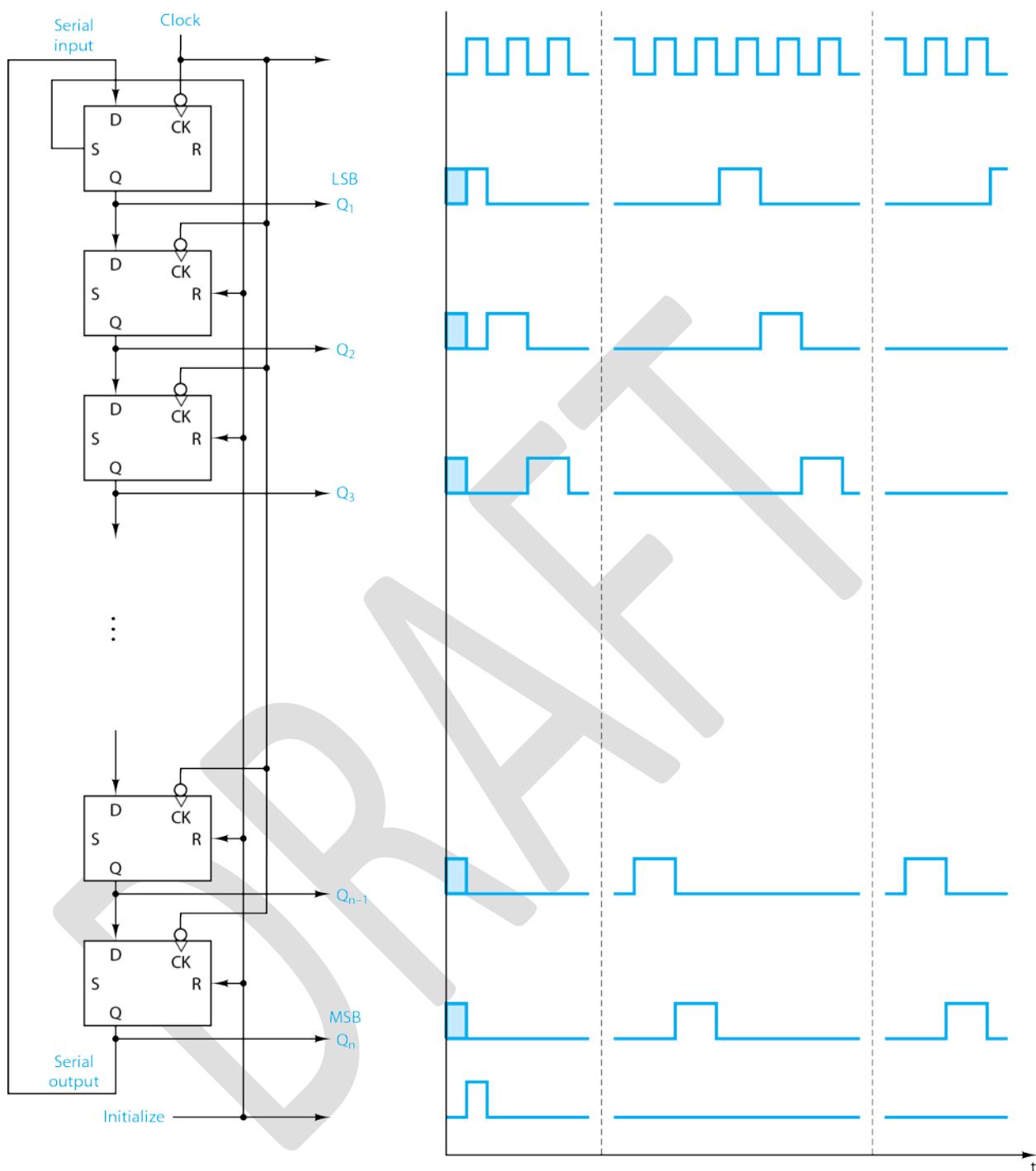


Figure 4.67. Ring counter logic diagram and timing

Example Design a 5-bit ring counter using the generic shift register structure of Fig. 4.52 and illustrate its operation using a timing diagram.

The shift register structure of Fig. 4.52 can be made to operate as a ring counter by configuring it as shown in Fig. 4.68a. Here the serial output Q_E has been fed back and connected to the serial input data line. To initialize the state to logic 1 into its least significant bit Q_A and logic 0 in the remaining bits Q_B, Q_C, Q_D, Q_E , the parallel inputs are set to $(D_A, D_B, D_C, D_D, D_E) = (1, 0, 0, 0, 0)$ so that this pattern can be loaded into the shift register via its parallel load mode. The operation of the circuit is illustrated in the timing diagram of Fig. 4.68b. The initial state is loaded into the shift register by setting the *Initialize* input to logic 1 to select parallel load mode for the first clock cycle. Then *Initialize* is returned to logic 0 to select shift mode. Each subsequent clock pulse then transfers the logic 1 to the next flip-flop until it reaches Q_E . Since Q_E is connected to *Serial input*, the logic 1 is transferred back to Q_A and the process is repeated. Two full cycles of the ring counter state sequence are shown in Fig. 4.68b. Note that the counter state sequence is $(Q_E, Q_D, Q_C, Q_B, Q_A) = 1, 2, 4, 8, 16, 1$, and so on.

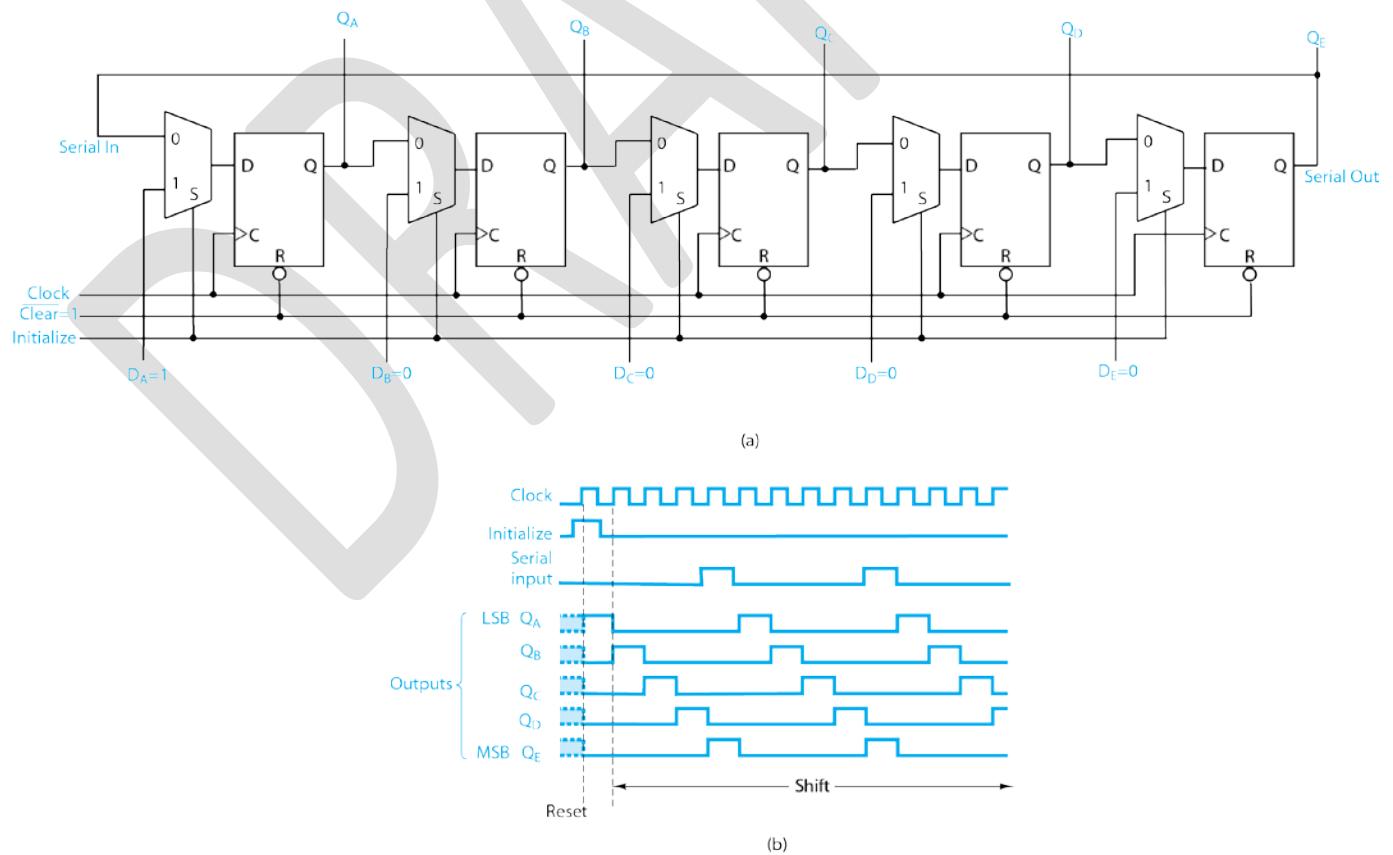


Figure 4.68: Ring counter example. (a) Logic diagram. (b) Timing diagram.

Now let us contrast the ring counter to a k -bit binary counter connected to a k -to- 2^k decoder as diagrammed in Fig. 4.69. This circuit is equivalent to the ring counter of Fig. 4.67 when the number of flip-flops in the ring is a power of 2, that is, $n = 2^k$. The Initialize signal clears the binary counter to state 0. Since a decoder is a minterm generator, one and only one of its outputs will be active high at any given moment in time. So when the counter is in state 0, the decoder's output line 0 will be high (signal Q_1 on the timing diagram). The next clock pulse will drive the counter to state 1, causing the decoder's output line Q_2 to go high. As each new clock pulse arrives, the binary counter changes state and moves the logic 1 down to the next decoder output. When the counter reaches its maximum state, the last decoder output line will go high, placing the logic 1 on signal Q_n . The next clock pulse will then cycle the counter to state 0 and hence send the logic 1 back to the first output line Q_1 . So the binary counter and decoder act like a ring counter.

What if a designer needs a ring counter for $n \neq 2^k$? The counter--decoder equivalent configuration of Fig. 4.69 can still be used by replacing the binary counter with a modulo- n counter. The decoder must satisfy the relationship

$$2^k > n > 2^{k-1} - 1$$

The decoder outputs labeled 0, 1, 2, ..., $n-2$, $n-1$ will be used as the ring counter output lines $Q_1, Q_2, Q_3, \dots, Q_{n-1}, Q_n$. The decoder output lines labeled $n, n+1, \dots, 2^k - 1$ are not used.

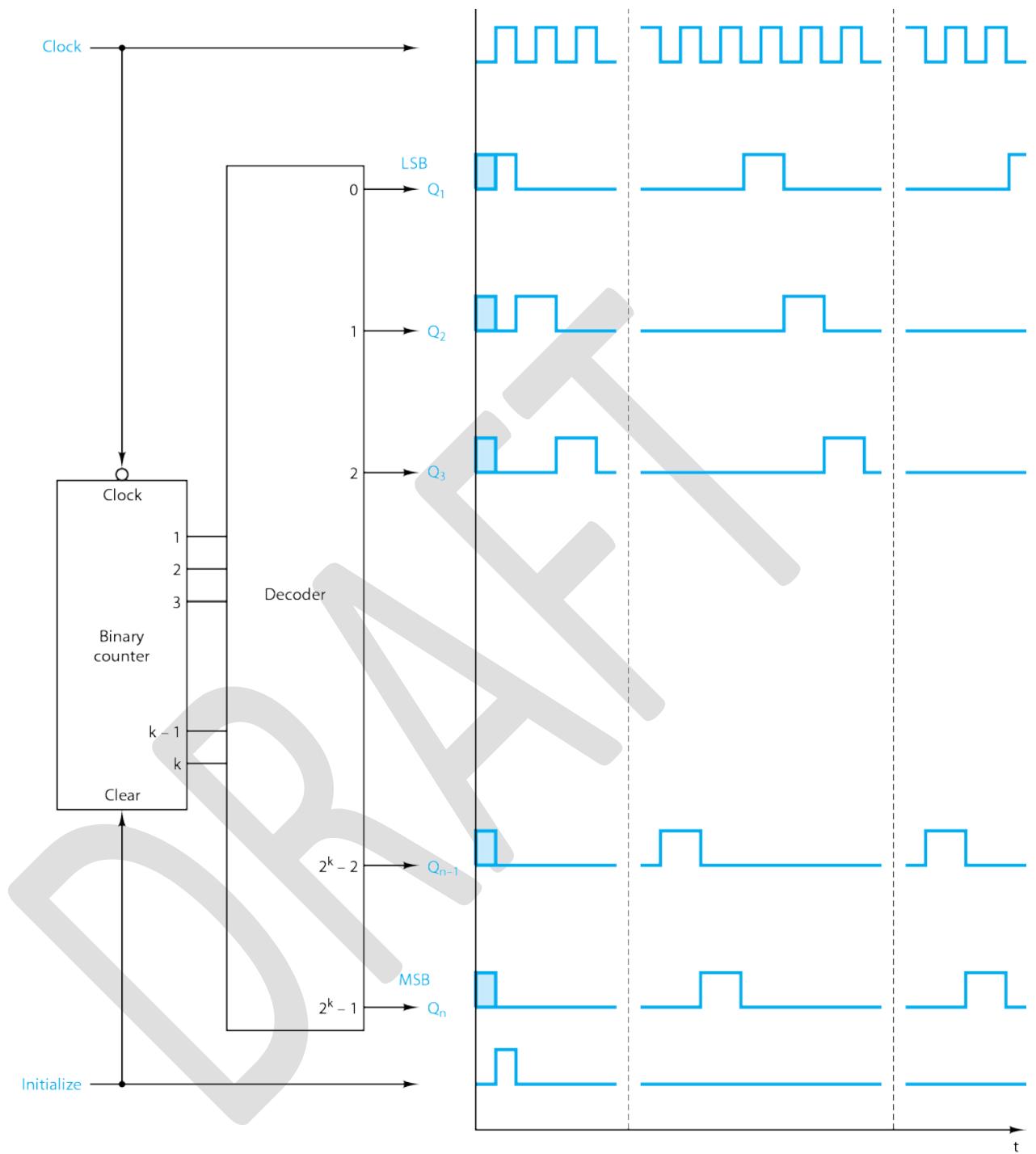


Figure 4.69: Ring counter equivalent.

Example Design a 13-state ring counter with active low outputs using a counter and a decoder.

Since $2^4 > 13 > 2^3$, the desired ring counter can be constructed from a modulo-13 counter and a 4-to-16 decoder. Let us use the modulo-13 counter from Fig. 4.66a. The logic diagram for the design is illustrated in Fig. 4.70.

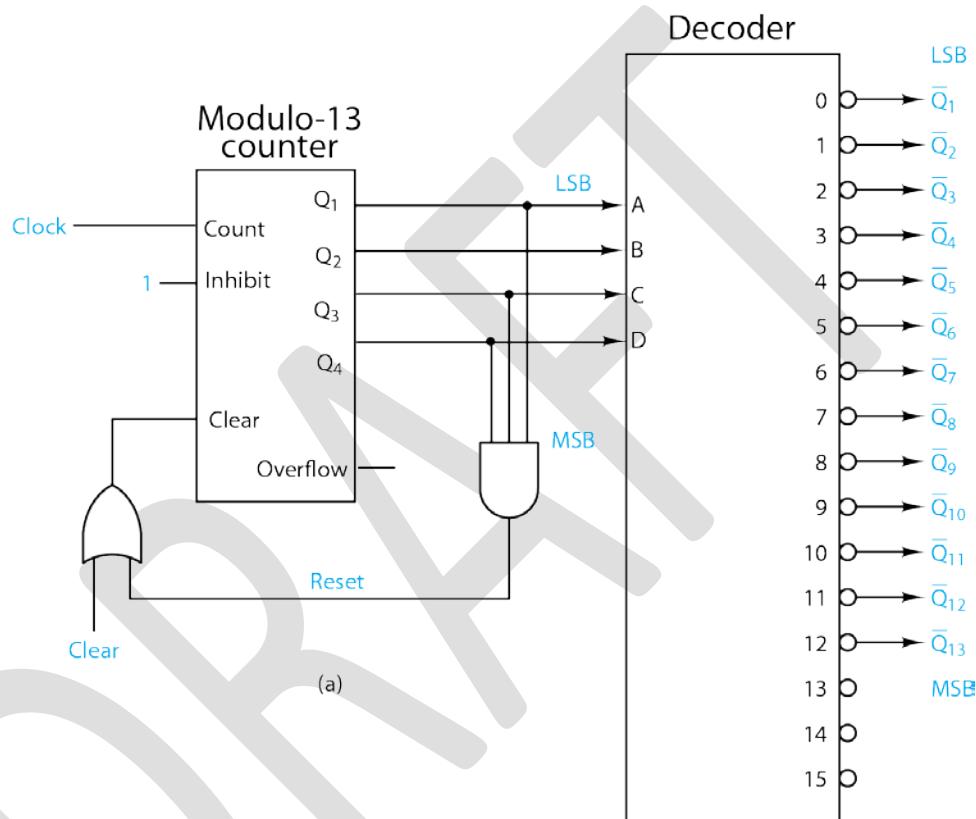


Figure 4.70: Thirteen-state ring counter equivalent.

A ring counter with a NOT gate in the feedback loop is called a *twisted-ring counter*. Sometimes this circuit is also called a *Johnson counter*. Take a look at the logic diagram in Fig. 4.71. Here we have inserted a NOT gate between the most significant counter bit Q_n and the shift register's *Serial input* line. Also notice that the *Initialize* signal connections have been altered to force the shift register to the all-zero state.

Let us examine the operation of the shift register in this configuration. First the Initialize signal clears the register. The NOT gate will therefore initially be supplying a logic 1 into the Serial input of the shift register (to the D input of the first shift-register flip-flop). As a matter of fact, this logic 1 feedback will continue until Q_n goes high later in the counting sequence. So, as each pulse is applied to the Clock input of the shift register, the logic 1 moves down the register one position (like a wave front) until it reaches the last shift-register flip-flop Q_n .

When the logic 1 arrives at Q_n , it changes the feedback signal on Serial input to logic 0 so that a logic 0 waveform now moves down the register as additional clock pulses are applied. The timing diagram of Fig. 4.71 shows a complete cycle of the twisted-ring counter. Each output signal Q_i ($i=1,n$) is a square wave, each delayed from its neighbors by one time period of the clock signal.

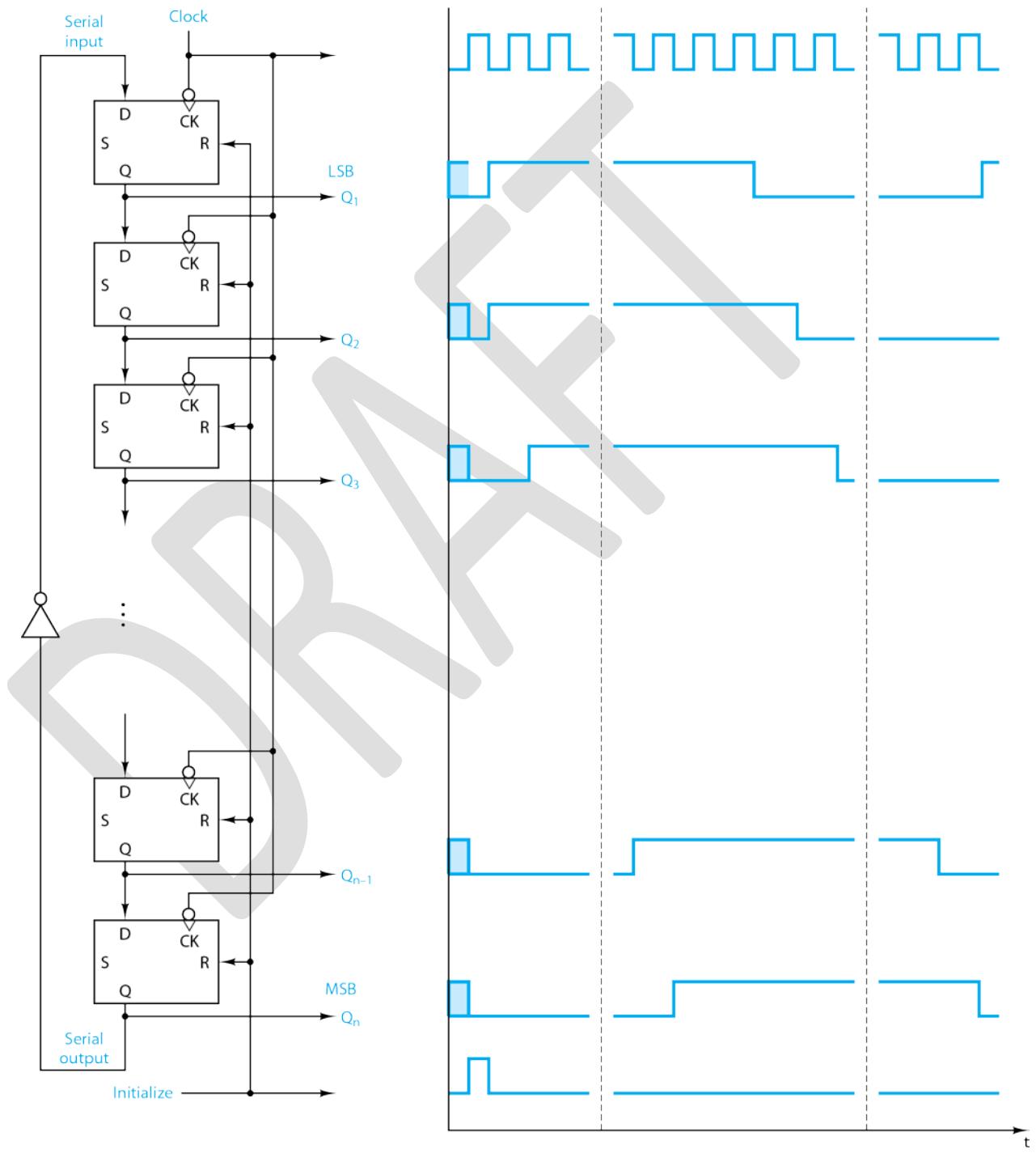


Figure 4.71: Twisted-ring counter.

How many unique states does this counter possess? Since it takes n clock pulses to propagate the logic 1 down the register and then another n clock pulses to return the register to the all-zero state, the twisted-ring counter has $2n$ unique states, where n is the number of flip-flops in the shift register.

Example Design a twisted-ring counter that has 10 unique states, using the generic shift register structure of Fig. 4.52.

The number of flip-flops needed to produce 10 states will be

$$n = 10 \div 2 = 5$$

Let us choose the generic shift register structure of Fig. 4.52 and create a feedback loop by connecting serial output Q_E to the serial input through a NOT gate, as shown in Fig. 4.72a. Since the parallel load feature of the shift register is not required, the *Load/Shift* signal is fixed at logic 0 and the parallel data input signals (D_A, D_B, D_C, D_D, D_E) are tied to logic 0 to increase the noise immunity of the implementation.

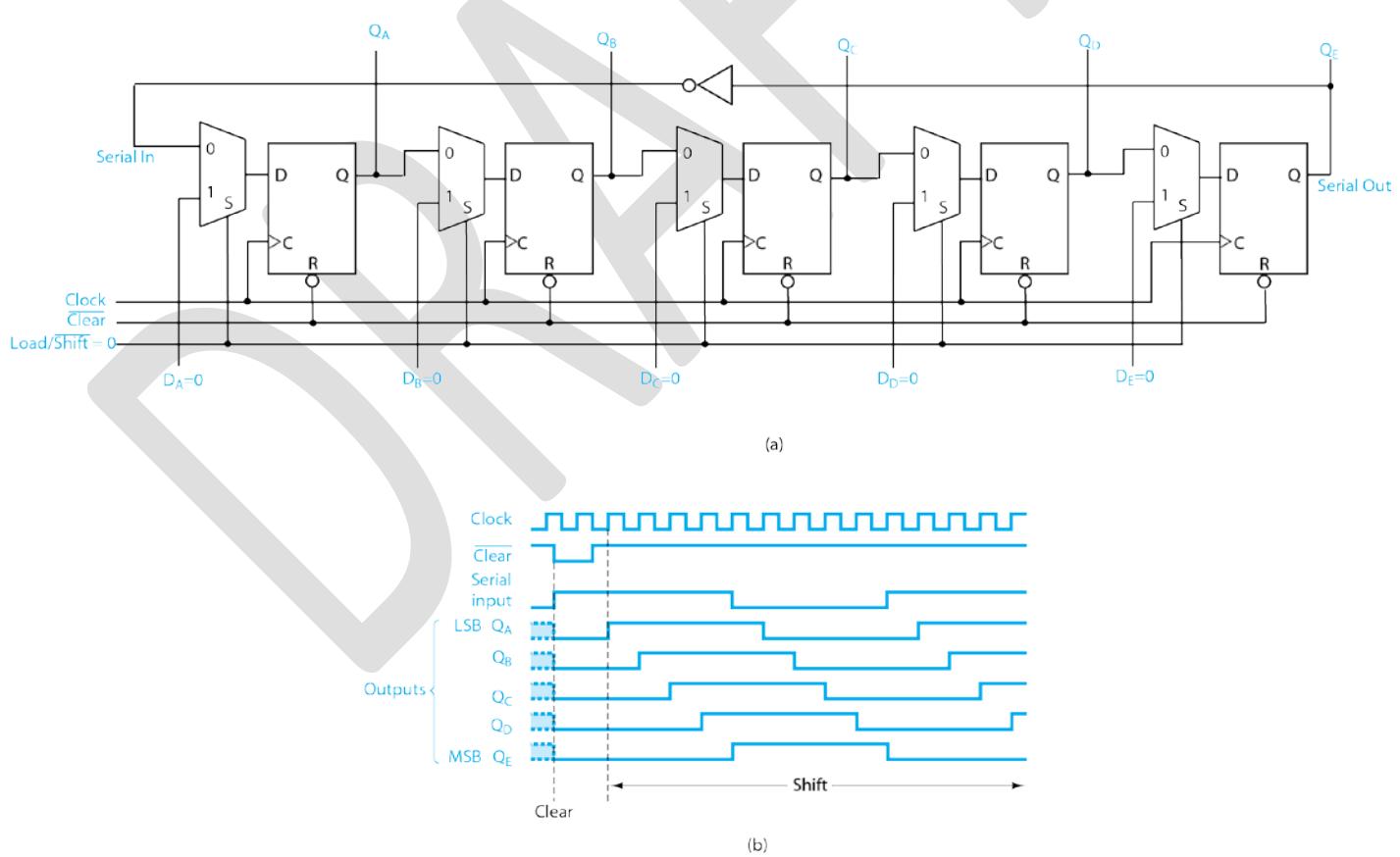


Figure 4.72: Ten-state twisted ring counter. (a) Logic diagram. (b) Timing diagram.

The timing diagram of Fig. 4.72b illustrates the proper use of the circuit. An active-low *Clear* signal initializes the register to state 0, forcing a logic 1 on the Serial input signal. On the rising edge of each clock pulse, the logic 1 moves from left to right through the register until Q_E goes high. The NOT gate then forces logic 0 into the register on the next five clock pulses to return the register to state 0. Applying 10 clock pulses drives the twisted-ring counter through all 10 of its unique states.

The state diagram of the resulting 10-state counter is given in Fig. 4.73. Figure 4.74 presents HDL behavioral models of a $2N$ state twisted-ring counter with a default value of $N = 5$.

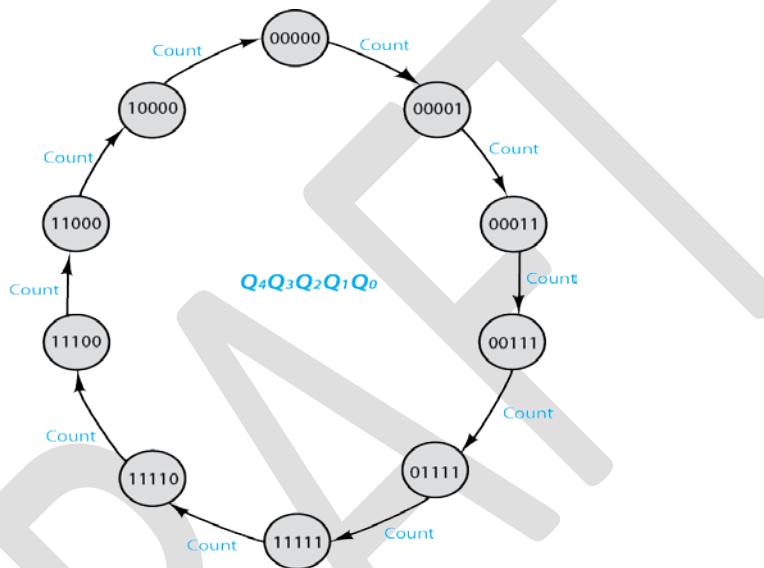


Figure 4.73: Ten-state twisted-ring counter state diagram.

```
//2N-state twisted-ring counter
//
module TwoNStateTwistedRing (Clock, Clear, Q);
input Clock, Clear;                                //define input variables
output reg [N-1:0] Q;                            //define output as an N-bit register
parameter N = 5;                                  //specify default value of N
always @(posedge Clock, negedge Clear) begin      //check for input changes
    if (Clear==0) Q <= 0;                          //initialize state to all 0's
    else begin                                     //begin counting in a Grey code sequence
        Q[0] <= ~Q[N-1];
        Q[N-1:1] <= Q[N-2:0];
    end
end
endmodule
```

(a) Verilog model

```
library ieee; use ieee.std_logic_1164.all;
entity TenStateTwistedRing is
    port( Q: out std_logic_vector(1 to 5);          -- ring counter outputs
          D: in std_logic_vector(1 to 5);           -- parallel inputs
          Clock: in std_logic;
          ClearN: in std_logic;
          Load_ShiftN: in std_logic);
end TenStateTwistedRing;
architecture Behavior of TenStateTwistedRing is
begin
    process(Clock, ClearN)
        variable Qint: std_logic_vector(1 to 5);
    begin
        if (ClearN = '0') then
            Qint := "00000";
        elsif Clock'Event and Clock = '1' then
            if Load_ShiftN = '1' then
                Qint := D;
            else
                Qint := not Qint(5) & Qint(1 to 4);
            end if;
        end if;
        Q <= Qint;
    end process;
end Behavior;
```

(b) VHDL model

Figure 4.74 HDL models of a 2N state twisted-ring counter

The states of the twisted-ring counter may be decoded using two-input AND gates. The table shown in Fig. 4.75 lists the twisted-ring counter output signals in tabular form. A unique logic condition for each state is shown in the column labeled State Decoder Logic Equations. For example, look at the third line in the table. When $Q_3 = 0$ and $Q_2 = 1$, this twisted-ring counter state is uniquely indicated. Scan the Q_3 and Q_2 columns to satisfy yourself that no other row (or counter state) has this bit pattern for Q_3 and Q_2 . All the other rows may also be uniquely identified by using only two appropriate logic signals, as indicated in the last column of the table. If the twisted-ring counter has both true and complemented output signals, a 2-input AND gate may be used to decode each output state. If both are not available, NOT gates will be required.

Twisted-ring counter output signals								State decoder logic equations
Q_n	Q_{n-1}	Q_{n-2}	Q_{n-3}	...	Q_3	Q_2	Q_1	
0	0	0	0		0	0	0	$\bar{Q}_n \cdot \bar{Q}_1$
0	0	0	0		0	0	1	$\bar{Q}_2 \cdot Q_1$
0	0	0	0		0	1	1	$\bar{Q}_3 \cdot Q_2$
0	0	0	0		1	1	1	$\bar{Q}_4 \cdot Q_3$
⋮								
0	0	0	0	0	1	1	1	$\bar{Q}_{n-3} \cdot Q_{n-4}$
0	0	0	0	1	1	1	1	$\bar{Q}_{n-2} \cdot Q_{n-3}$
0	0	0	1	1	1	1	1	$\bar{Q}_{n-1} \cdot Q_{n-2}$
0	0	1	1	1	1	1	1	$\bar{Q}_n \cdot Q_{n-1}$
1	1	1	1	1	1	1	1	$Q_n \cdot Q_1$
1	1	1	1	1	1	0	0	$Q_2 \cdot \bar{Q}_1$
1	1	1	1	1	0	1	0	$Q_3 \cdot \bar{Q}_2$
1	1	1	1	1	0	0	0	$Q_4 \cdot \bar{Q}_3$
⋮								
1	1	1	1	0	0	0	0	$Q_{n-3} \cdot \bar{Q}_{n-4}$
1	1	1	1	0	0	0	0	$Q_{n-2} \cdot \bar{Q}_{n-3}$
1	1	1	0	0	0	0	0	$Q_{n-1} \cdot \bar{Q}_{n-2}$
1	0	1	0	0	0	0	0	$Q_n \cdot \bar{Q}_{n-1}$
0	0	0	0	0	0	0	1	$\bar{Q}_n \cdot Q_1$
0	0	0	0	0	0	1	1	$\bar{Q}_2 \cdot Q_1$
0	0	0	0	0	0	1	1	$Q_3 \cdot Q_2$

Figure 4.75: Twisted-ring counter state decoding.

Example

Design a timing signal generator using a twisted-ring counter and state-decoding logic that meets the following specifications:

1. The timing signals will be active high during one period of the clock signal (go high on the rising edge of one clock pulse and then go low on the rising edge of the next clock pulse).
2. An initialization signal will be used to synchronize the timing generator with the other system components. The signal Initialize will be active high.
3. The first timing control signal (f_1) will generate a pulse that will go high on the leading edge of the second clock pulse applied after the initialize command.
4. The second timing control signal (f_2) will generate a pulse that will go high on the leading edge of the eighth clock pulse applied after the initialize command.
5. The third timing control signal (f_3) will generate a pulse that will go high on the leading edge of the eleventh clock pulse applied after the initialize command.

6. The timing waveforms (f_1, f_2, f_3) will be repeated every 16 clock pulses.

First let us generate the state sequence for a 16-state twisted-ring counter since the sequence of signals is to be repeated every 16 clock pulses. Since a twisted-ring counter has 2^n states, where n is the number of flip-flops, let us select the serial-in, parallel-out shift register of Fig. 4.52a for our implementation. The outputs of the shift register are ($Q_H, Q_G, Q_F, Q_E, Q_D, Q_C, Q_B, Q_A$). If we use a NOT gate to feed back the complement of Q_H to the serial inputs, the state behavior of the twisted-ring counter may be described in the table of Fig. 4.76.

Clock pulse No.	State No. (decimal)	Q_H	Q_G	Q_F	Q_E	Q_D	Q_C	Q_B	Q_A	State (decoder)
0	0	0	0	0	0	0	0	0	0	$\bar{Q}_H \cdot \bar{Q}_A$
1	1	0	0	0	0	0	0	0	1	$\bar{Q}_B \cdot Q_A$
2	3	0	0	0	0	0	0	1	1	$f_1 = \bar{Q}_C \cdot Q_B$
3	7	0	0	0	0	0	1	1	1	$\bar{Q}_D \cdot Q_C$
4	15	0	0	0	0	1	1	1	1	$\bar{Q}_E \cdot Q_D$
5	31	0	0	0	1	1	1	1	1	$\bar{Q}_F \cdot Q_E$
6	63	0	0	1	1	1	1	1	1	$\bar{Q}_G \cdot Q_F$
7	127	0	1	1	1	1	1	1	1	$\bar{Q}_H \cdot Q_G$
8	255	1	1	1	1	1	1	1	1	$f_2 = Q_H \cdot Q_A$
9	254	1	1	1	1	1	1	0	0	$Q_B \cdot \bar{Q}_A$
10	252	1	1	1	1	1	0	0	0	$Q_C \cdot \bar{Q}_B$
11	248	1	1	1	1	0	0	0	0	$f_3 = Q_D \cdot \bar{Q}_C$
12	240	1	1	1	0	0	0	0	0	$Q_E \cdot \bar{Q}_D$
13	224	1	1	0	0	0	0	0	0	$Q_F \cdot \bar{Q}_E$
14	192	1	0	0	0	0	0	0	0	$Q_G \cdot \bar{Q}_F$
15	128	1	0	0	0	0	0	0	0	$Q_H \cdot \bar{Q}_G$
0	0	0	0	0	0	0	0	0	0	$\bar{Q}_H \cdot \bar{Q}_A$
1	1	0	0	0	0	0	0	1	1	$\bar{Q}_B \cdot Q_A$
2	3	0	0	0	0	0	1	1	1	$f_1 = \bar{Q}_C \cdot Q_B$

Figure 4.76: State sequence for the 16-state twisted-ring counter example.

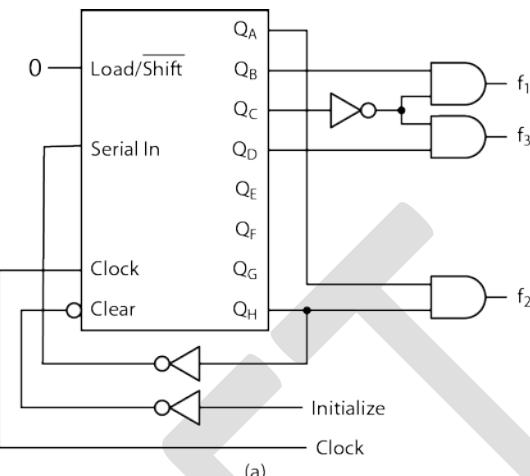
When the shift register is initialized to the all-zero state, it assumes the values in the first row of the table (labeled clock pulse 0). The first clock pulse will drive the register to the second row (labeled clock pulse 1). The second clock pulse forces the register into the third row (labeled clock pulse 2). This twisted-ring counter state is to be decoded as timing signal f_1 , as shown in the last column of the table. We may continue this same procedure to identify the counter states to be decoded for timing signals f_2 and f_3 . The decoding logic for the three timing signals is

$$f_1 = \bar{Q}_C \cdot Q_B$$

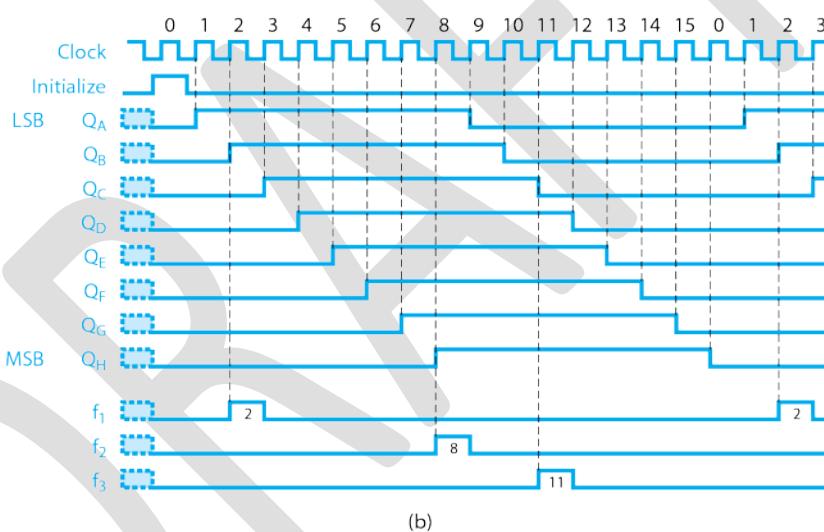
$$f_2 = Q_H \cdot Q_A$$

$$f_3 = Q_D \cdot \bar{Q}_C$$

These signals will be repeated every 16 clock pulses. The logic and timing diagrams for the design are illustrated in Fig. 4.77a and 4.77b.



(a)



(b)

Figure 4.77: 16-state twisted-ring counter example. (a) Logic diagram. (b) Timing diagram.

4.6 Summative Design Examples

In this section, we present several examples that summarize the concepts introduced in this chapter.

4.6.1 Register File Design

Digital systems often require small amounts of information to be stored for easy access. Memory devices to hold such information can range in capacity from a few bits to many Gigabytes. Section 4.3 illustrated the use of registers to hold data values. It is often useful to have an array of registers, often called a *register file*, to hold multiple data values and other information in a computer's central processing unit. As illustrated in Fig. 4.78a, such an array has two dimensions. The number of bits, n , per register determines the size of the stored items. Data to be written into a selected register and data read from a selected register are thus n -bit values DIN_{n-1} and $DOUT_{n-1}$, respectively. The number of registers $N = 2^k$ in the register file determines the number of unique items that can be stored. As illustrated in Fig. 4.78b, one k -bit number, address WA_{k-1} , is required to select a register to be written and a second address, RA_{k-1} , selects a register to be read. Since new information is not necessarily written in every clock cycle, a write-enable signal, $WrEn$, is active if and only if DIN is to be written to register WA on a given clock transition. In contrast, $DOUT$ continuously contains the value of selected register RA .

Figure 4.78c presents an implementation of a $4 \times N$ register file. Data is read from the register designated by read address RA by using a four-to-one multiplexer to select one of the four register outputs as $DOUT$. Data is written to a selected register by applying DIN to the inputs of all four registers, and using a two-to-four decoder to activate the clock enable signal, CE , of the register corresponding to write address WA . The write-enable signal $WrEn$ controls the decoder enable signal, EN , so that a selected decoder output is activated if and only if $WrEn = 1$.

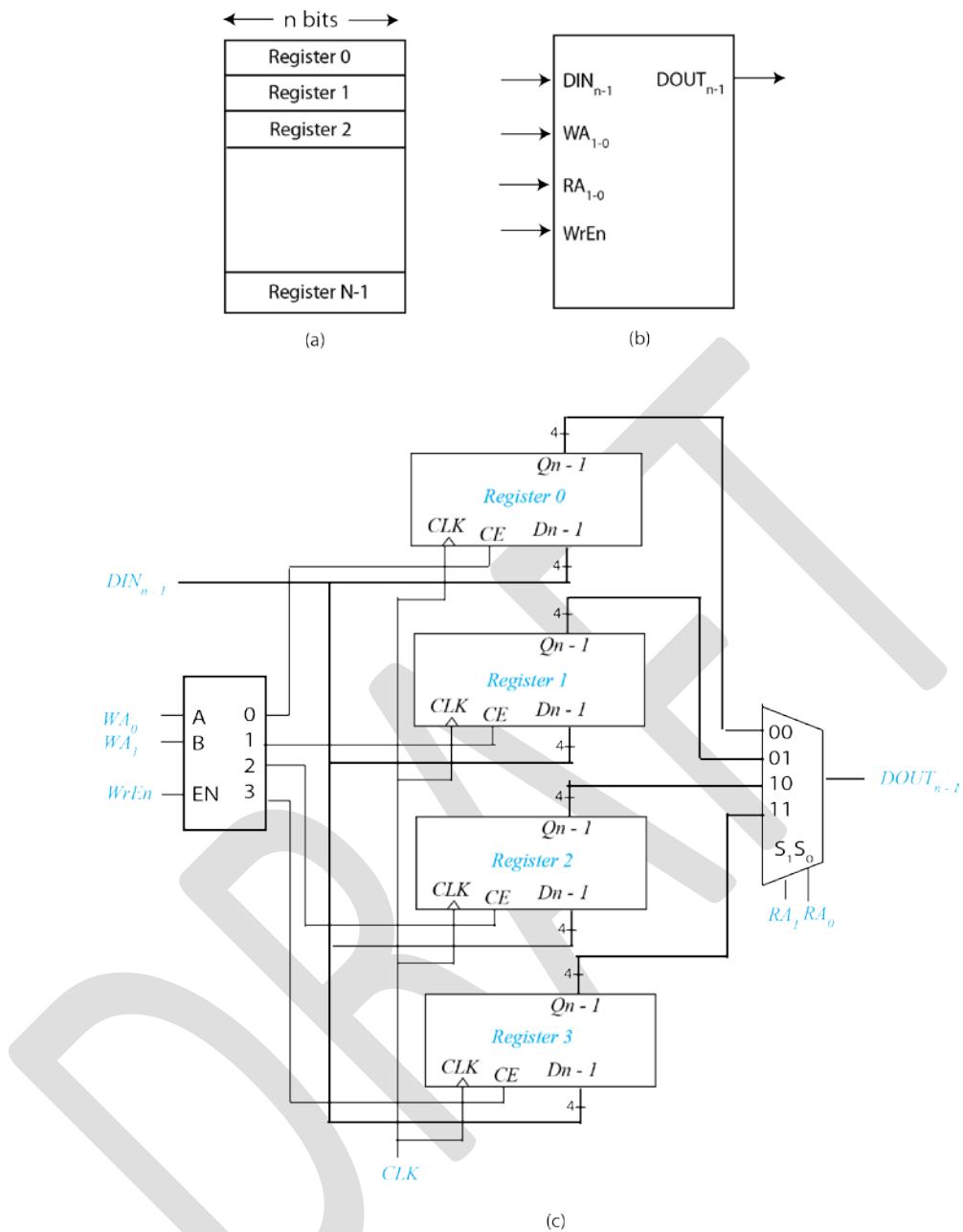


Figure 4.78: 4 x n register file. (a) Data storage format. (b) Logic symbol. (c) Implementation.

4.6.2 Multiphase Clock Design

A multiphase clock is a synchronous circuit that produces a sequence of non-overlapping square waves in sync with a clock signal input. Such circuits are useful for controlling the timing of register transfers during the instruction cycle of a computer or other applications where sequences of operations must be performed in a specific order. Figure 4.79 shows a block diagram and a timing diagram for a four-phase clock. The clock will first be designed for implementation with traditional logic devices and then for programmable logic.

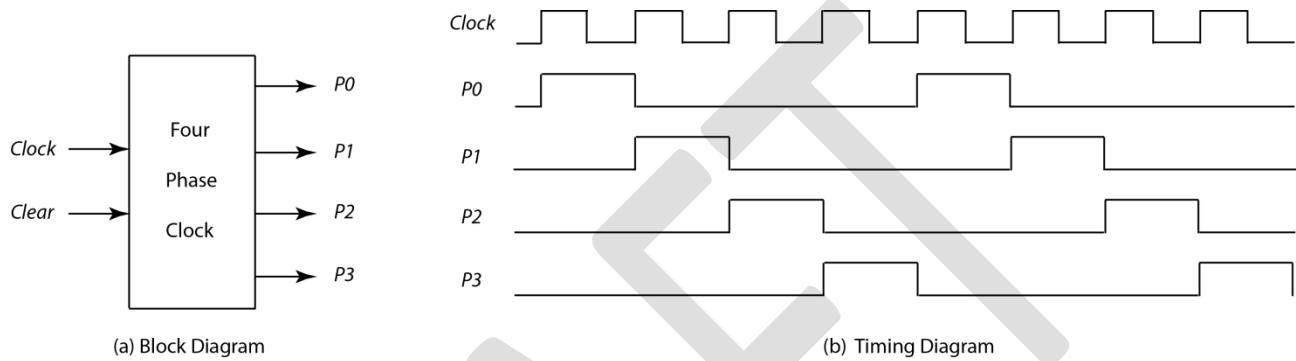


Figure 4.79: Four-Phase Clock.

The design of multiphase clock circuits is straight forward and will now be illustrated for the four-phase case. Partitioning the circuit in to a counter and output logic is a convenient first step and is illustrated in Fig. 4.80a. A four-state counter is needed in order to produce four clock phases and can be realized with two flip-flops. The output logic must decode each counter state and produce the corresponding clock phase. This can be done with four AND gates or with a two-to-four decoder. Each will be illustrated below. But first let's design the counter.

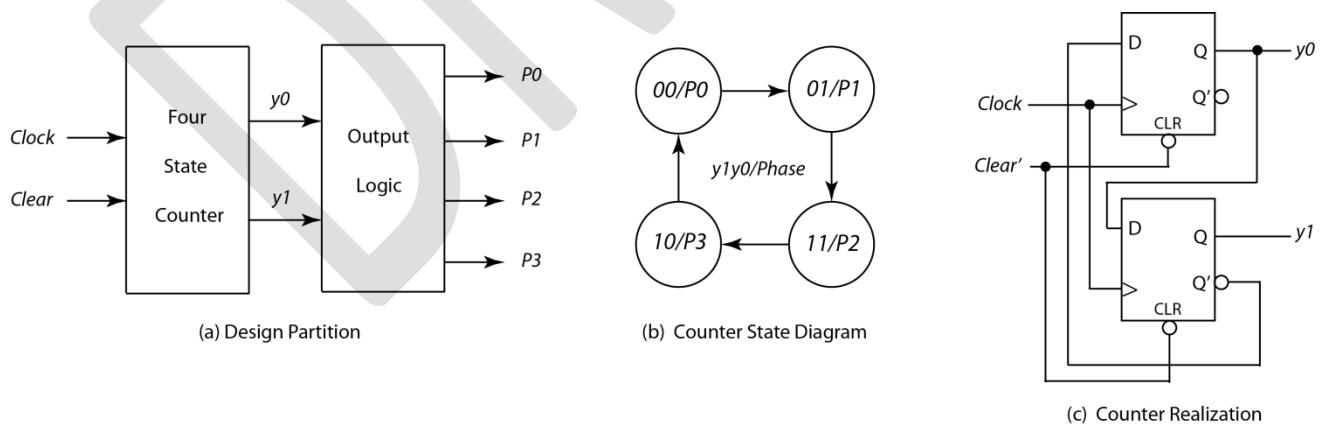


Figure 4.80: Four-Phase Clock Design.

There are many different ways to realize a four-state counter, but two considerations lead to a good solution for this application. Recall that the clock phases must be non-overlapping which implies that transient states should be avoided. This can be accomplished by counting in a Gray-code sequence such as 00 – 01 – 11 – 10 which requires only one flip-flop to change at

a time. Figure 4.80b shows a state diagram of the four-phase clock assuming this state sequence. The counter design can be completed by recalling that twisted-ring counters count in a Gray-code sequence and can be easily realized with D flip-flops. Figure 4.80c shows the realization for this example.

The design is completed by incorporating the output logic. Figure 4.81a shows a solution where each state is decoded by a dedicated AND gate to produce the corresponding clock phase. Figure 4.81b shows a solution where a two-to-four decoder is used to decode the states. The AND gate approach is good for the four-phase case. However, the decoder design is preferred for larger numbers of phases. Note that P_2 is produced by decoder output 3 and P_3 by decoder output 2 since the counter produces a 00-01-11-10 state sequence.

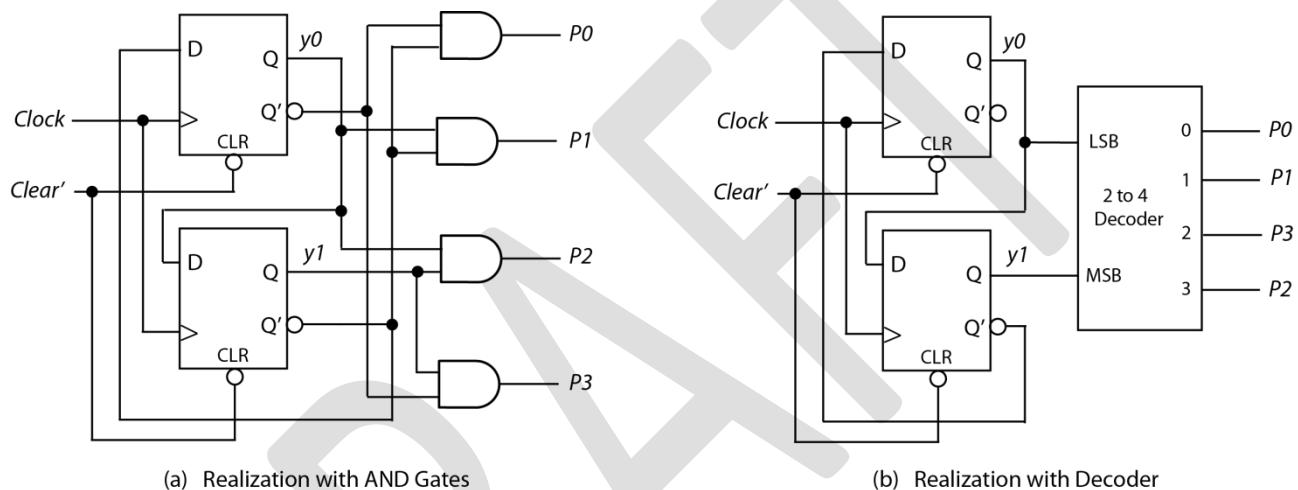


Figure 4.81: Four-Phase Clock Realizations.

The four-phase clock can be designed for programmable logic implementation by writing an HDL model. Verilog will be used for this example. One approach is to write a structural Verilog model of the previous design. However, the preferred approach is to write a behavioral model based on the state diagram in Figure 4.80b. This is more efficient from a design time perspective since there is no need to design a gate-level realization before writing the Verilog code.

See Figure 4.82 for a Verilog model of the four-phase clock. In this model, the clock output phase signals are specified as elements of a register, $P[0:3]$, rather than as individual signals as shown previously. The model also uses *state* and *nextstate* internal variables to specify the current and next state of the clock, respectively, and to facilitate the proper timing of state and output changes. The value of *state* is updated on the positive edge of the *Clock* input to the value of *nextstate*. A case statement is used to derive the *output* and *next state* for each state as specified in the state diagram given in Figure 4.80 (b). The occurrence of a *Clear* = 0 signal takes precedence over the *Clock* and places the counter in state S0.

```
//Verilog behavioral model of a four-phase clock
module FourPhaseClockVerilog (           //name the module
    input Clock, Clear,                 //declare inputs
    output reg [0:3] P);               //declare outputs as a register for convenience
    reg [1:0] state, nextstate;        //declare internal state and next state variables
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b11, S3 = 2'b10; //parameterize states

    always @ (posedge Clock, negedge Clear) //watch for changes of Clock or Clear
        if (Clear == 0) state <= S0; //active-low Clear takes the machine to S0
        else state <= nextstate; //state change occurs on positive edge of Clock
    always @ (state) //derive output and next state after each state change
        case (state) //case statement specifies output and next state for each state
            S0: begin P <= 4'b1000; nextstate <= S1; end
            S1: begin P <= 4'b0100; nextstate <= S2; end
            S2: begin P <= 4'b0010; nextstate <= S3; end
            S3: begin P <= 4'b0001; nextstate <= S0; end
        endcase
    endmodule
```

Figure 4.82: Verilog Behavioral Model of the Four-Phase Clock.

4.6.2 Digital Timer Design

This example illustrates how counters can be used as frequency dividers to realize a basic digital timer that can count up in minutes and seconds from 0 to 59-minutes and 59-seconds and then repeat. The timer can be stopped and restarted at any point in the count and can also be cleared to 0 when desired. Figure 4.83 shows a block diagram of the timer. The Timer has two inputs, *Clear* and *Start/Stop*, to control its operation and a *Clock* input to provide synchronization with real time. Typically, the *Clock* signal would be generated internally by a crystal-controlled oscillator or other pulse generator circuit but is considered as an external signal here in order to focus on the logic design aspects of the problem. A 50-MHz clock frequency is assumed. Some of the design details would change for a different frequency, but the basic approach would remain the same.

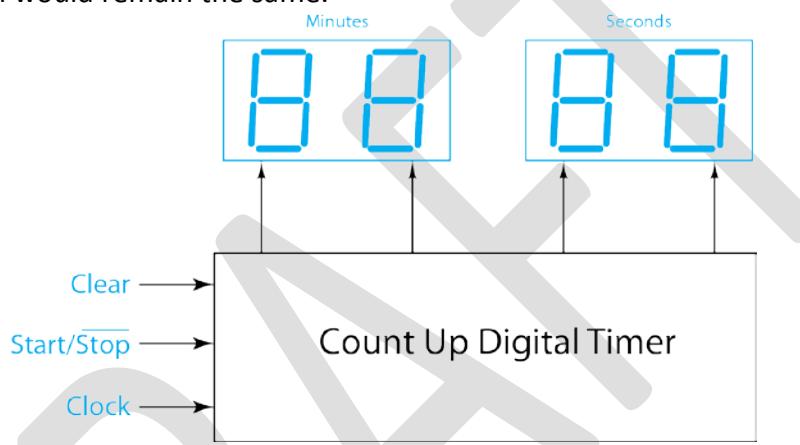


Figure 4.83: Count Up Digital Timer Block Diagram

The Timer can be realized by using a series of counters to step down the 50-MHz clock to seconds and then to minutes. Figure 4.84 shows how this approach is organized. A divide-by-50 counter is used to step the frequency down to 1-MHz and is followed by a divide-by- 10^6 counter to produce a 1-Hz pulse. This pulse is used to increment the least significant digit of the Timer. A divide-by-10 counter is then used to produce a tenth-of-seconds pulse that increments the most significant seconds digit. Displaying minutes requires a one pulse per minute signal which can be produced by dividing the tenth-of-seconds pulse by six using a divide-by-6 counter. The least and most significant digits of the minutes are displayed in a similar fashion to seconds using divide-by-10 and divide-by-6 counters in series which has the effect of dividing by 60. It should be noted that these counters not only synchronize the next counter in the sequence but also drive the corresponding digit of the display. Not shown on the organization diagram are the BCD to seven-segment decoders that are needed to drive the seven-segment display devices.

Realization of the various counters can be accomplished using traditional gates, flip-flops, and counter modules or using programmable logic devices such as FPGAs. The latter will be illustrated with Verilog models of the basic counters needed in the Timer.

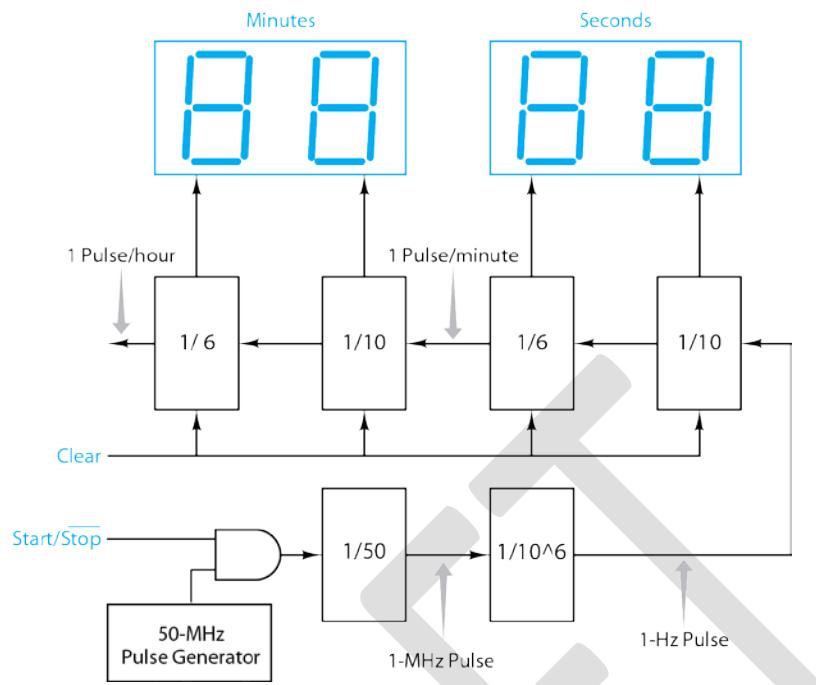


Figure 4.84: Count-Up Digital Timer Organization.

The divide-by-50 counter can be realized by cascading a divide-by-5 counter and a divide-by-10 counter as shown in Figure 4.85a. Six divide-by-10 counters can be cascaded to produce the divide-by- 10^6 counter as illustrated in Figure 4.85b. Divide-by-60 counters are realized by cascading divide-by-10 and divide-by-6 counters as previously described in Figure 4.84. Thusly, three types of counters are needed to realize the Timer – divide-by-5, divide-by-6, and divide-by-10. Verilog models for each are given in Figures 4.86, 4.87, and 4.88.

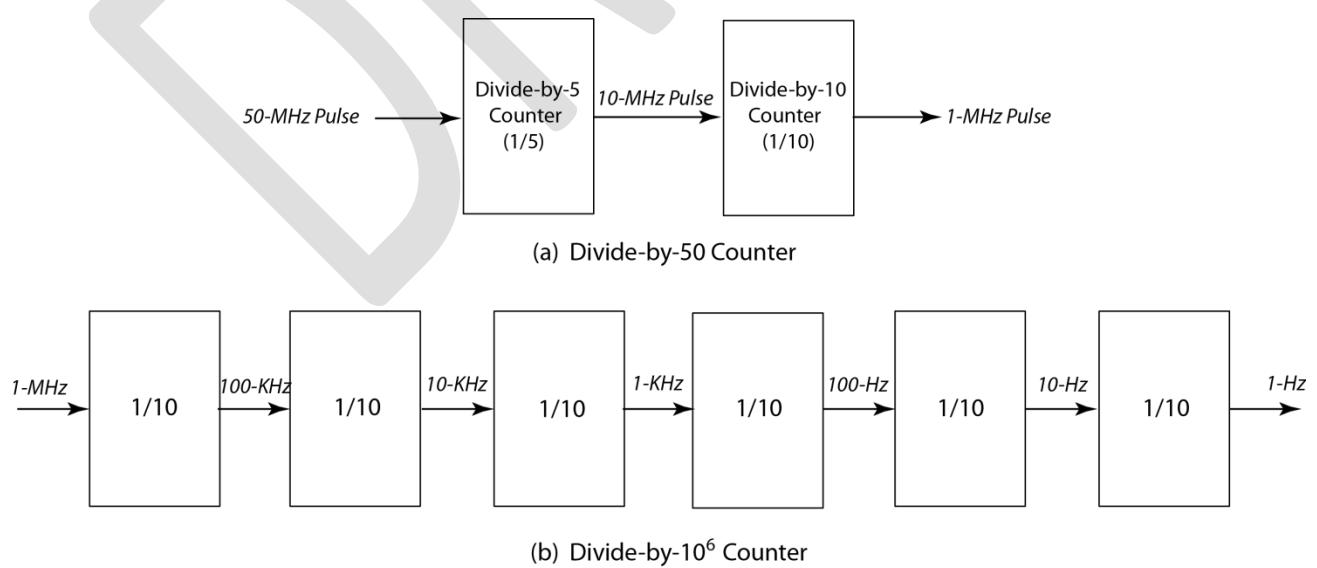


Figure 4.85: Cascades of Counters

```
//Divide-By-Five Counter
module DivideBy5Counter (CLK, CLR, FIVE, state, nextstate);
input CLK, CLR;
output reg FIVE;
output reg [2:0] state, nextstate;
parameter
S0=3'b000,S1=3'b001,S2=3'b010,S3=3'b011,S4=3'b100,S5=3'b101,S6=3'b110,S7=3'b111;
//Count to five in binary using a Moore machine model
    always @(negedge CLK, negedge CLR)
        if (CLR==0) state <= 3'b0;
            else state <= nextstate;
    always @ (state,CLK)
        case (state)
            S0: begin nextstate <= S1; FIVE <= 0; end
            S1: begin nextstate <= S2; FIVE <= 0; end
            S2: begin nextstate <= S3; FIVE <= 0; end
            S3: begin nextstate <= S4; FIVE <= 0; end
            S4: begin nextstate <= S0; FIVE <= 1; end
            S5: nextstate <= S5;
            S6: nextstate <= S6;
            S7: nextstate <= S7;
        endcase
endmodule
```

Figure 4.86: Divide-by-Five Counter.

```
//Divide-By-Six Counter
module DivideBy6Counter (CLK, CLR, SIX, state, nextstate);
input CLK, CLR;
output reg SIX;
output reg [2:0] state, nextstate;
parameter
S0=3'b000,S1=3'b001,S2=3'b010,S3=3'b011,S4=3'b100,S5=3'b101,S6=3'b110,S7=3'b111;
//Count to six in binary using a Moore machine model
    always @(negedge CLK, negedge CLR)
        if (CLR==0) state <= 3'b0;
            else state <= nextstate;
    always @ (state,CLK)
        case (state)
            S0: begin nextstate <= S1; SIX <= 0; end
            S1: begin nextstate <= S2; SIX <= 0; end
            S2: begin nextstate <= S3; SIX <= 0; end
            S3: begin nextstate <= S4; SIX <= 0; end
            S4: begin nextstate <= S5; SIX <= 0; end
            S5: begin nextstate <= S0; SIX <= 1; end
            S6: nextstate <= S6;
            S7: nextstate <= S7;
        endcase
endmodule
```

Figure 4.87: Divide-by-Six Counter.

```
//Divide-By-Ten Counter. Sometimes called a BCD counter.
module DivideBy10Counter (CLK, CLR, TEN, state, nextstate);
input CLK, CLR;
output reg TEN;
output reg [3:0] state, nextstate;
parameter S0=4'b0,S1=4'b1,S2=4'b10,S3=4'b11,S4=4'b100,S5=4'b101,S6=4'b110,S7=4'b111,
      S8=4'b1000,S9=4'b1001,S10=4'b1010,S11=4'b1011,S12=4'b1100,S13=4'b1101,S14=4'b1
110,
      S15=4'b1111;
//Count to ten in binary using a Moore machine model
always @(negedge CLK, negedge CLR)
  if (CLR==0) state <= S0;
  else state <= nextstate;
always @ (state,CLK)
  case (state)
    S0: begin nextstate <= S1; TEN <= 0; end
    S1: begin nextstate <= S2; TEN <= 0; end
    S2: begin nextstate <= S3; TEN <= 0; end
    S3: begin nextstate <= S4; TEN <= 0; end
    S4: begin nextstate <= S5; TEN <= 0; end
    S5: begin nextstate <= S6; TEN <= 0; end
    S6: begin nextstate <= S7; TEN <= 0; end
    S7: begin nextstate <= S8; TEN <= 0; end
    S8: begin nextstate <= S9; TEN <= 0; end
    S9: begin nextstate <= S0; TEN <= 1; end
    S10: nextstate <= S10;
    S11: nextstate <= S11;
    S12: nextstate <= S12;
    S13: nextstate <= S13;
    S14: nextstate <= S14;
    S15: nextstate <= S15;
  endcase
endmodule
```

Figure 4.88: Divide-by-Ten Counter

4.6.3 Programmable Baud Rate Generator

This example illustrates how counters and multiplexers can be used to realize a programmable baud rate generator. UARTs (Universal Asynchronous Receiver/Transmitters) are hardware devices commonly found in computers and used to translate internal data in parallel format to/from external data in asynchronous serial format. The design of a UART will be presented in Chapter 8. One component of a UART is a *baud rate generator* which is a clocking device that controls the speed, or baud rate, of the asynchronous data transfer. Several standard baud rates are in use, so it is desirable for a UART to support more than one. This can be accomplished by having a baud rate generator that can be programmed to produce the desired frequency of clock.

Such a programmable baud rate generator will be designed in this example. Figure 4.89a shows the block diagram of a baud rate generator that produces a clock signal, *BaudOut*, at one of eight baud rates – 19.2-Kbps, 9600-bps, 4800-bps, 2400-bps, 1200-bps, 600-bps, 300-bps, and 150-bps, as specified by control signals, s_2, s_1, s_0 . Figure 4.89b specifies the control signal codes used in the example. Since baud rates are standardized across the computer and communication industries, it is important for clock frequencies to be accurate. This can be accomplished by using a timing signal from a crystal-based oscillator. Several standard frequencies are available with 2.45765-MHz used for this example as shown in Figure 4.89a.

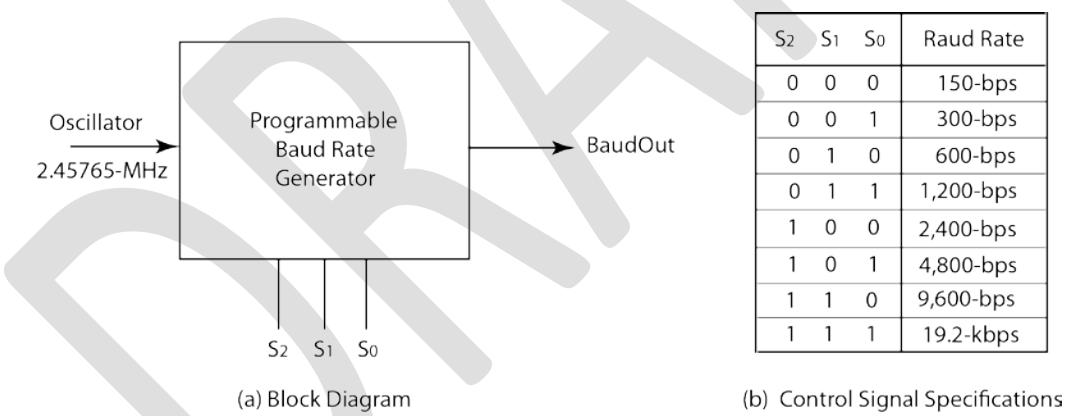


Figure 4.89: Programmable Baud Rate Generator.

The oscillator frequency is much faster than the baud rates, so it is necessary to use counters as frequency dividers to produce the desired rates. This can be done using two stages of counters as illustrated in Figure 4.90. The first stage is a divide-by-64 counter that produces a 38.4-KHz signal. The second stage is an 8-bit binary counter that successively divides the 38.4-KHz signal by two to produce the desired frequencies of 19.2-KHz, 9600-Hz, ... , 150-Hz, respectively. The desired baud rate signal is then selected using an eight-to-one multiplexer to produce the output signal *BaudOut*.

The counters in Figure 4.90 will now be designed, first using standard hardware components,

and then with Verilog. Note that a divide-by-64 counter can be realized by cascading two divide-by-8 counters. Similarly, an 8-bit binary counter can be realized by cascading two 4-bit binary counters. Recall that 4-bit binary counters are standard and can be configured and interconnected in various ways to realize other counters. Figure 4.91 shows how to use two 4-bit counters and two AND gates to realize the divide-by-64 counter. In essence, the AND gates are used to detect when each counter reaches state 1111 and generates a signal that triggers the next stage.

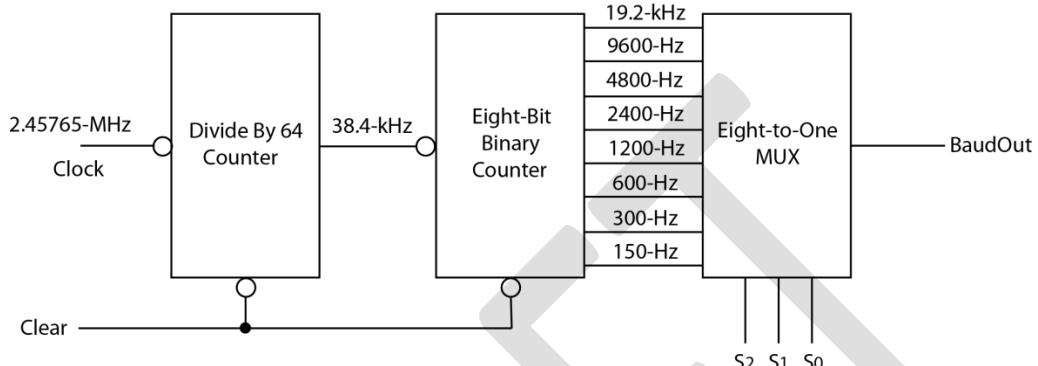


Figure 4.90: Baud Rate Generator High-Level Organization.

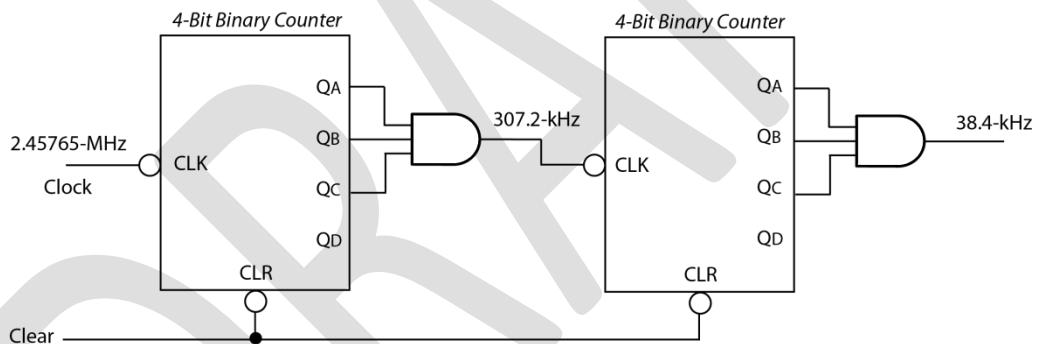


Figure 4.91: Realization of the Divide-by-64 Counter.

Figure 4.92 shows a realization of the 8-bit binary counter. The first stage is triggered by a 38.4-kHz signal from the divide-by-64. An AND is used to detect when the first stage reaches state 1111 and to generate a signal triggering the next second stage.

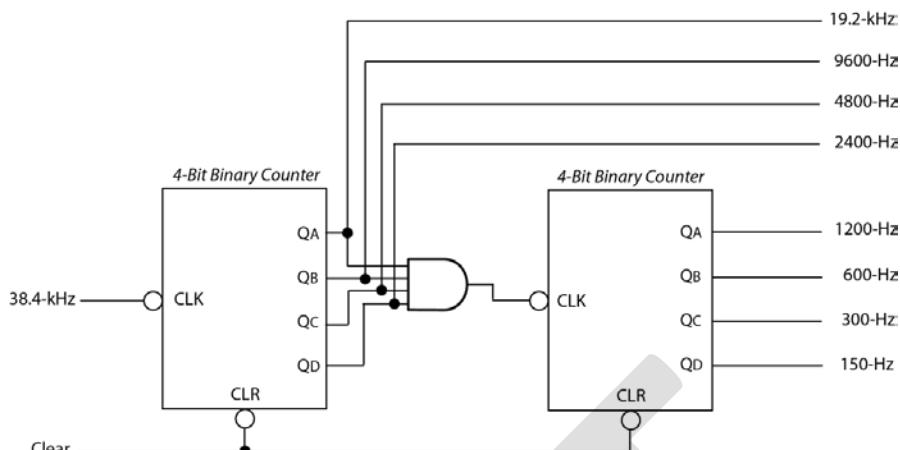


Figure 4.92: Realization of an 8-Bit Binary Counter.

The traditional design approaches require the use of standard components such as 4-bit binary counters. Designing with Verilog for FPGA realization is not so constrained and can often be more efficient and creative as will now be illustrated. A Verilog behavioral model of a generic N-bit binary counter was presented earlier in this chapter. That model forms the basis of the models for the counters needed in Figure 4.92. The divide-by-64 model is shown in Figure 4.93. First, a 6-bit binary counter is realized. Then output *out* is added that generates a signal after every 64 states to effect the divide-by-64 functionality. The 8-bit counter is straight-forward use of the generic model with N=8 and is shown in Figure 4.94.

```
// Behavioral description of a divide-by-64 counter.
module SixtyFour (COUNT, CLEAR, Q, out);
    input COUNT, CLEAR;
    output reg [N-1:0] Q; // Q is defined as a N-bit output register
    output reg out;
    parameter N=6; //Define default value of N=6
    always @ (posedge COUNT, negedge CLEAR)
        if (CLEAR==0) Q <= 1'b0; // Q is loaded with all 0's
        else
            begin
                if (Q == 2**N - 1)
                    begin
                        out <= 1'b1; Q <= 1'b0;
                    end //Once Q = all 1's it returns to all 0's
                else
                    begin
                        out <= 1'b0; Q <= Q + 1'b1;
                    end //Q is incremented
            end
endmodule
```

Figure 4.93: Verilog Model of the Divide-by-64 Counter.

```
// Behavioral description of an 8-bit binary counter.
module TwoFiftySix (COUNT, CLEAR, Q);
    input COUNT, CLEAR;
    output reg [N-1:0] Q;           // Q is defined as a N-bit output register
    parameter N=8;                //Define default value of N=8
    always @ (posedge COUNT, negedge CLEAR)
        if (CLEAR==0) Q <= 1'b0;      // Q is loaded with all 0's
        else
            begin
                if (Q == 2**N - 1)
                    Q <= 1'b0;          //Once Q = all 1's it returns to all 0's
                else
                    Q <= Q + 1'b1;      //Q is incremented
            end
endmodule
```

Figure 4.94: Verilog Model of the 8-Bit Binary Counter.

Completing the Verilog design requires a model of the eight-to-one decoder which is shown in Figure 4.95 and a top-level module that combines the components. Figure 4.96 shows the top-level model which relies on module instantiation and structural Verilog interconnections using net datatypes (wire).

```
//Verilog behavioral model
//Eight to One Gated Multiplexer
module Eight2OneMUX (A,B,C,G,D,Y);
    input A,B,C,G;
    input [0:7] D;
    output reg Y;
    always begin
        if (G) Y = 1'b0; else
            case ({C,B,A})
                3'b000: Y = D[{C,B,A}];
                3'b001: Y = D[{C,B,A}];
                3'b010: Y = D[{C,B,A}];
                3'b011: Y = D[{C,B,A}];
                3'b100: Y = D[{C,B,A}];
                3'b101: Y = D[{C,B,A}];
                3'b110: Y = D[{C,B,A}];
                3'b111: Y = D[{C,B,A}];
            endcase
    end
endmodule
```

Figure 4.95: Eight-to-One Multiplexer Verilog Model.

```
//Baud Rate Generator--19.2-kHz, 9600-Hz, 4800-Hz, 2400-Hz, 1200-Hz, 600-Hz, 300-Hz, 150-Hz
module BaudRateVerilog (clock,clear,S2,S1,S0,BaudOut);
    input clock, clear, S2, S1, S0;
    output BaudOut;
    wire [5:0] Q;
    wire [7:0] rates;
    wire out;

    SixtyFour (clock,clear,Q,out);
    TwoFiftySix (out,clear,rates);
    Eight2OneMUX (S0,S1,S2,1'b0,rates,BaudOut);
endmodule
```

Figure 4.96: Top-Level Verilog Model of the Baud Rate Generator.

4.7 Summary and Review Questions

In this chapter we examined the design and operation of a number of standard sequential logic modules, including registers, shift registers and counters. Many of these modules are available as standard TTL functions [1], as well as being available in design libraries used to create VLSI circuits, programmable gate arrays, and printed circuit boards. A number of examples were presented to illustrate the use of these modules as building blocks to create larger circuits. It is often advantageous to utilize these modules in a design rather than creating custom circuits. In Chapter 5 we will present formal design methods for the design and analysis of synchronous sequential circuits. The reader is referred to [2], [3], and [4] for further examples of sequential logic modules and their applications.

- [1] *The TTL Data Book, Volume 2*. Dallas, Texas: Texas Instruments, Inc., 1988.
- [2] *Digital Design Principles and Practices*. Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 349–369.
- [3] *Contemporary Logic Design*. Menlo Park, CA: Benjamin/Cummings Publishing Co., 1994, pp. 282–321.
- [4] *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley Publishing Co., 1993, pp. 405–447.

Test your understanding by answering the following questions.

1. Do you understand the basic concept of a sequential machine and the finite state machine (FSM) model?

2. Do you understand the difference between Mealy and Moore machines?
3. Do you understand the difference between a latch and a flip-flop?
4. Can you draw the state diagrams for D and JK flip-flops?
5. Can you write Verilog and/or VHDL models for D and JK flip-flops?
6. Do you understand the basic concepts of registers and shift registers?
7. Can you design a four-bit parallel-in/parallel-out register using D flip-flops?
8. Can you write Verilog and/or VHDL for registers and shift registers?
9. Do you understand the basic concept of a counter?
10. Can you draw the state diagram of a four-bit binary counter?
11. Can you write Verilog and/or VHDL models of a four-bit binary counter?
12. Can you draw the state diagram of a decade counter? Write its Verilog and/or VHDL model?
13. Can you draw the state diagram of a modulo-5 counter?
14. Do you know how many flip-flops are needed to realize a modulo-12 counter?
15. Can you draw state diagrams for four-bit ring and twisted-ring counters? Write their Verilog and/or VHDL models?

4.8 Collaboration Exercises

1. Finite state machines (FSMs) were used in this chapter to model synchronous sequential circuits. However, they can be used to model other computing structures as well. Identify some of these structures.
2. Many different types of latches and flip-flops were described in this chapter. Compare their similarities and differences and discuss which type or types are the most useful in today's logic design space.
3. Flip-flops can be designed to be either positive-edge triggered or negative-edge triggered. Are there any advantages or disadvantages of one over the other? How can one type be converted to the other?
4. Binary counters can be implemented as either synchronous circuits or asynchronous circuits. Explore the advantages and disadvantages of each type. Identify applications where one type is better than the other.
5. Modulo- N counters sequence through N states and repeat. Such counters can be realized using binary, ring, or twisted-ring counters with added logic that resets the counter to state 0 after state $N-1$. For each value of N , in the range 2 to 8, evaluate which type counter is the best to use in terms of the number of flip-flops and the amount of added logic. Are there other factors that should be considered?

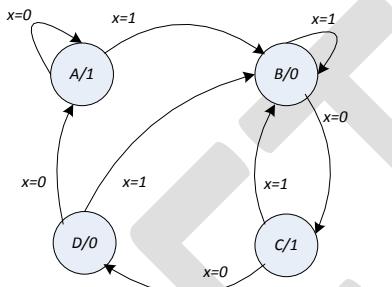
4.9 Problems

4.1 Draw state diagrams and determine the number of states for the following circuits.

- a. Four-bit binary up counter. How many states are in an n -bit binary counter?
- b. Four-bit ring counter. How many states are in an n -bit ring counter?
- c. Four-bit twisted-ring counter. How many states are in an n -bit twisted-ring counter?

4.2 Derive a state table for the sequential machine defined by the following state diagram.

Is this a Mealy or a Moore machine?



4.3 Construct a state diagram from the following state table. Is this a Mealy or a Moore machine?

Present State	Input x		Next state/Output z
	0	1	
S0	S3/1	S1/0	
S1	S3/0	S2/1	
S2	S3/0	S0/1	
S3	S1/1	S2/0	

4.4 Construct a state diagram from the following state table: What is the logic equation for the output variable z ?

Present state	Input x		Next state/Output z
	0	1	
A	D/1	B/0	
B	D/1	C/0	
C	D/1	A/0	
D	B/1	C/0	

4.5 Given the following state table, if the circuit starts in state A find the output and state sequences for the input sequence $x = 01010101$.

Present state	Input x		Next state/Output z
	0	1	
A	D/0	B/0	
B	C/0	B/0	
C	B/0	C/0	
D	B/0	C/1	

4.6 For the following sequential circuit, draw a state diagram of the circuit and determine the output sequence for the input sequence $x = 0010110101$, if the starting state is A.

Present state	Input x		Next state/Output z
	0	1	
A	B/0	C/1	
B	C/1	B/0	
C	A/0	A/1	

4.7 For the following sequential circuit, draw a state diagram of the circuit and determine the output sequence for the input sequence $x = 100101000$ if the starting state is A.

Present state	Input x		Output z	Next state
	0	1		
A	C	B	0	
B	A	B	0	
C	D	C	1	
D	B	C	0	

4.8 The state diagrams in Figs. P4.5a and P4.5b are those of a Mealy model circuit M1, with output Z_1 , and a similar Moore model circuit M2, with output Z_2 .

- Derive state tables for each of these two models.
- Complete the timing diagram of Fig. P4.5c by determining the next states and outputs of the two circuits, assuming that both circuits receive the same input signal x and that state changes occur on the rising edge of the clock.
- Discuss the differences in the next states and outputs of the two circuits.

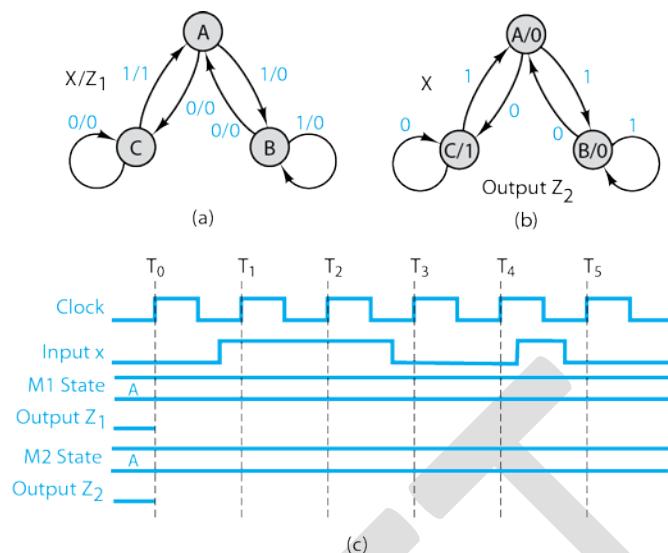


Figure P4.8 a. Mealy model M1 b. Moore model M2 c. Timing diagram

4.9 Derive the state diagram and characteristic equation of the latch circuit in Fig. P4.6.

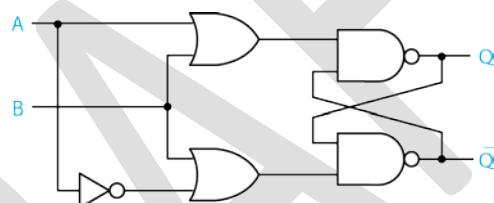


Figure P4.9

4.10 Find the excitation table of the latch circuit in Fig. P4.10 and describe its behavior in words.

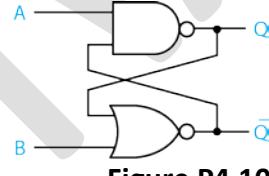


Figure P4.10

4.11 Is the circuit of Fig. P4.11a a valid latch design? Explain. If it is a latch, complete the excitation table of Fig. P4.11b. Can it be used as a gated SR latch? If so, how?

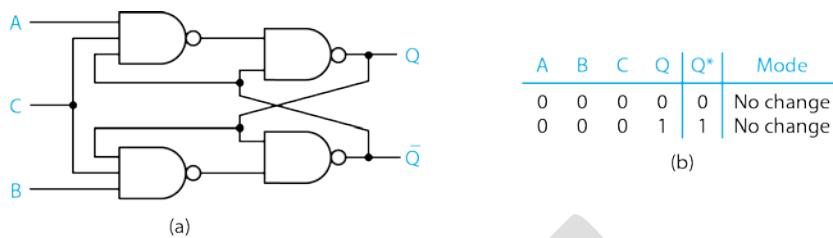


Figure P4.11

4.12 Construct state diagrams for the following flip-flop types.

- (a) D flip-flop
- (c) T flip-flop
- (b) SR flip-flop
- (d) JK flip-flop

4.13 Derive the characteristic state equations shown in Table 4.2 for the following components.

- (a) SR latch
- (c) JK flip-flop
- (b) Clocked D latch
- (d) T flip-flop

4.14 Given the JK flip-flop of Fig. P4.14a, complete the timing diagram of Fig. P4.14b by determining the waveform of the output Q .

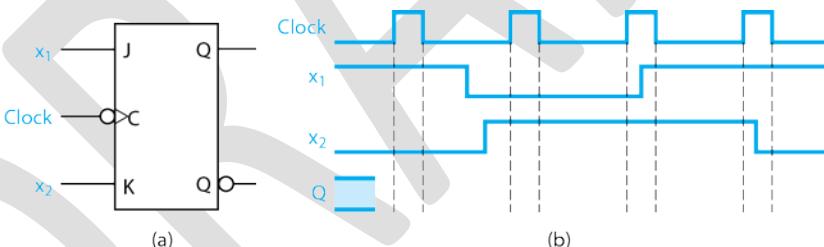


Figure P4.14: a. Flip-flop. b. Timing diagram.

4.15 Given the SR flip-flop of Fig. P4.15a, complete the timing diagram of Fig. P4.15b by determining the waveform of the output Q . The condition $S = R = 1$ is produced twice by the inputs. Will this lead to unstable operation? Explain.

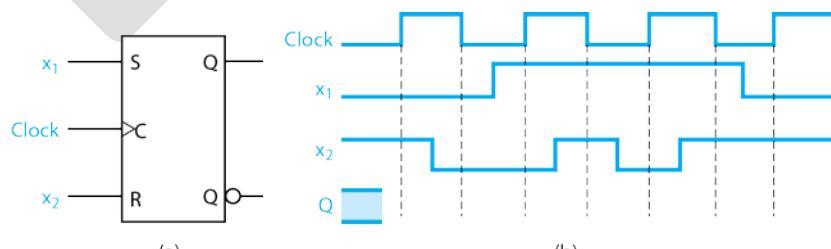


Figure P4.15: a. Flip-flop. b. Timing diagram.

4.16 The waveforms of Fig. P4.16 are applied to the inputs of a positive-edge-triggered JK flip-flop with asynchronous present and clear inputs. Complete the timing diagram by drawing the waveforms of flip-flop outputs Q and \bar{Q} .

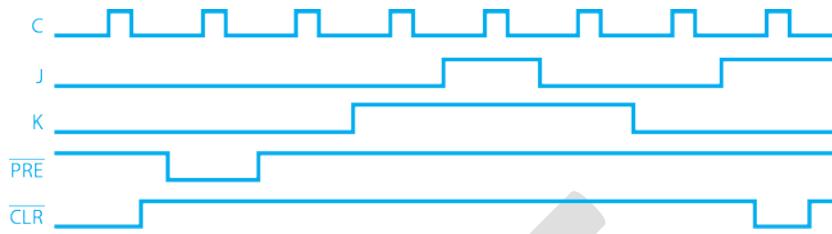


Figure P4.16

4.17 The circuit of Fig. P4.17a contains a D latch, a positive-edge-triggered D flip-flop, and a negative-edge-triggered D flip-flop. Complete the timing diagram of Fig. P4.17b by drawing the waveforms of signals y_1 , y_2 , and y_3 .

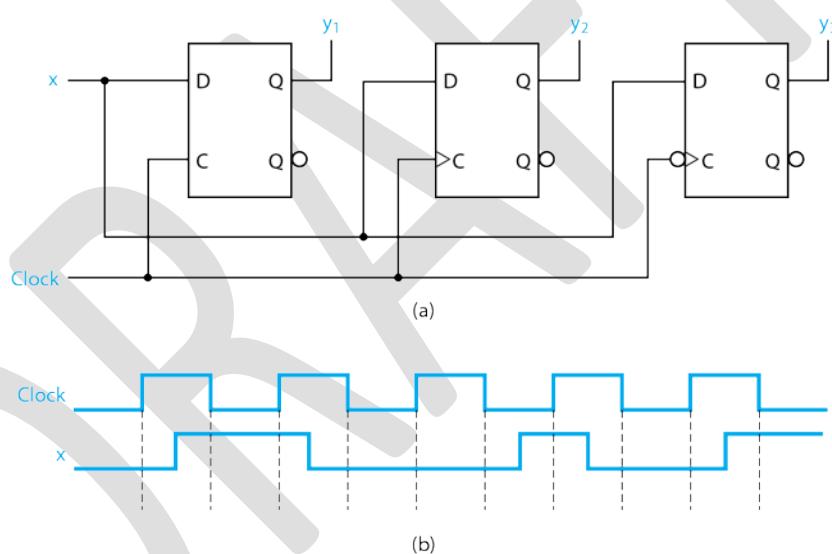


Figure P4.17: a. Logic diagram. B. Timing diagram.

4.18 The circuit of Fig. P4.18 is initially in state $Q_1 = Q_2 = 0$. Sketch a timing diagram showing the states of signals Q_1 and Q_2 for a sequence of seven clock pulses.

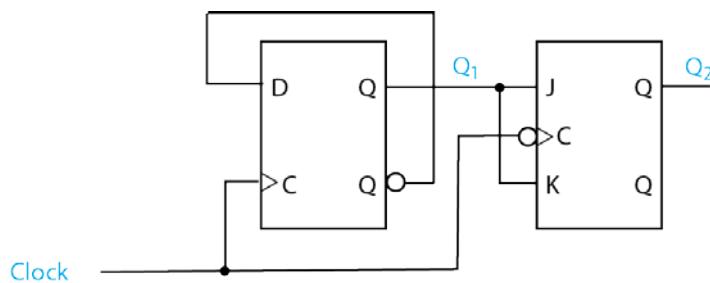


Figure P4.18.

4.19 The circuit of Fig. P4.19a contains a JK flip-flop and a D flip-flop. Complete the timing diagram of Fig. P4.19b by drawing the waveforms of signals Q_1 and Q_2 .

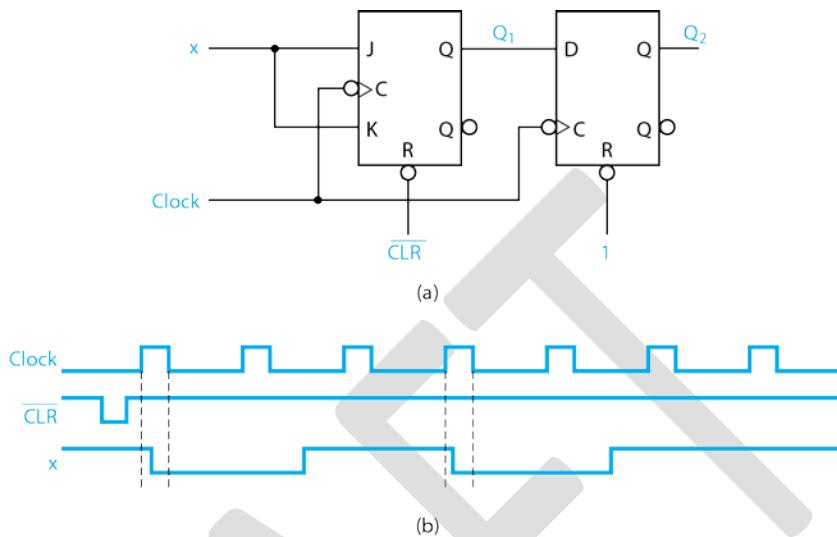


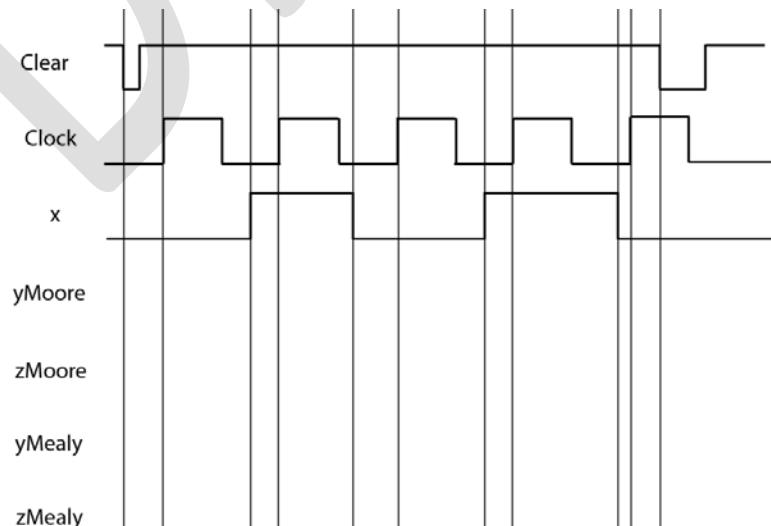
Figure P4.19: a. Logic diagram. B. Timing diagram.

4.20 Derive state diagrams for the circuits in Fig P4.20.



Figure P4.20

4.21 Complete the following timing diagram for the circuits in Fig. P4.20. Assume each flip-flop has an active-low clear input.



4.22 Complete the following table of flip-flop excitation values required to produce the indicated flip-flop state changes, where y indicates the present state and Y the desired next state of the flip-flop.

Present State y	Next State Y	JK flip-flop		D flip-flop	SR flip-flop		T flip-flop T
0	0	J	K	D	S	R	
0	1						
1	0						
1	1						

4.23 Discuss why the condition $S = R = 1$ leads to an unstable condition for an SR latch.

4.24 Describe how the unstable condition $S = R = 1$ is avoided in the storage latch of the following:

4.25 The circuit of Fig. P4.25 is intended to operate as a JK latch. Discuss whether or not this circuit is stable for the condition $J = K = C = 1$. If the circuit is unstable, discuss what could be done to the clock signal, C , to make the circuit operate as expected.

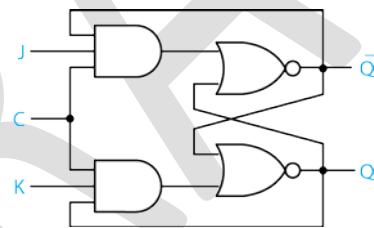


Figure P4.25

4.26 Design a master–slave JK flip-flop with asynchronous present and clear inputs using only NOR gates.

4.27 Describe the operational difference between a clocked D-type *latch* and a D-type master-slave *flip-flop* as observed from the outputs of the devices.

4.28 Describe how a master–slave flip-flop appears to operate as an *edge-triggered* device when observed from its external outputs.

4.29 Using the hazard-free D latch circuit structure shown in Fig. 4.29c, design a rising-edge triggered D flip-flop using only NAND gates and inverters.

4.30 Examine the D flip-flop circuit of Fig. 4.44 and describe how it operates as an edge-triggered device.

4.31 Examine the SR latch circuit of Fig. 4.39b, and then design a falling-edge-triggered D flip-flop with an active-low asynchronous \overline{CLR} input. Describe why the \overline{CLR} input is referred to as an *asynchronous* input, while the D input is called a *synchronous* input.

4.32 Connect a D flip-flop in such a manner that it will perform like a clocked T flip-flop.

4.33 Construct a D flip-flop using only a JK flip-flop and an inverter, but no additional gates.

4.34 Using three rising-edge-triggered T flip-flops and a minimum number of additional gates, construct a circuit that will operate as a binary counter with an enable signal E . When $E = 1$, the counter should increment from 0 to 7 on each clock pulse, with (flip-flop outputs $Q_2Q_1Q_0 = 000-001-010-011-100-101-110-111$) and then roll over to 000 and repeat the sequence. When $E=0$, the counter should stop and hold its current count.

4.35 The circuit of Fig. P4.35 is similar to that used in some commercial programmable logic sequencer chips. It is designed to operate as either a JK flip-flop or as a D flip-flop according to the settings of switches SW_1 and SW_2 . Determine the settings (open or closed) of the two switches required for JK and D flip-flop operation, and describe how this operation is achieved.

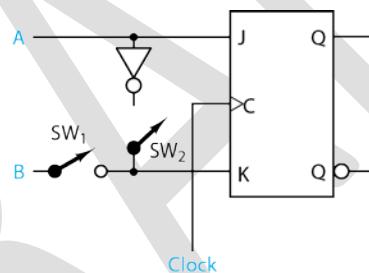


Figure P4.35

4.36 Using falling-edge-triggered D flip-flops with active-low asynchronous clear inputs, design and sketch a circuit for an 8-bit register. The register has data inputs $D_7 - D_0$, data outputs Q_7-Q_0 , clock CLK , and clear input \overline{CLR} . On a clock transition, the data inputs should be captured in the register. When \overline{CLR} is active, all outputs should reset to 0. Use logic symbols for the D flip-flops and not gate-level schematics.

4.37 Develop a function table for the SN74198 8-bit bidirectional shift register presented in Fig. P4.37.

4.38 Use a 2-to-4 decoder, NAND gates, and edge-triggered D flip-flops to design a 4-bit shift register module that has the following function table. Draw a logic diagram for your module.

S1	S0	Mode
0	0	Shift right (all 4 bits)
0	1	Shift left (all 4 bits)
1	0	Synchronous common clear
1	1	Synchronous parallel load

- 4.39** Use a 3-to-8 decoder, NAND gates, and edge-triggered D flip-flops to design a 4-bit shift register module that has the following function table. Draw a logic diagram for your module.

S2	S1	S0	Mode
0	0	0	Shift right (all 4 bits)
0	0	1	Shift left (all 4 bits)
0	1	0	Synchronous common clear
0	1	1	Synchronous parallel load
1	0	0	Synchronous preset MSB to 1 and clear other bits
1	0	1	Synchronous data hold
1	1	0	Ring counter (Q output of LSB is fed back as serial input to the MSB)
1	1	1	Twisted-ring counter (\bar{Q} output of LSB is fed back as serial input to the MSB)

- 4.40** Use logic gates and flip-flops to design a synchronous modulo-15 counter.
- 4.41** Use logic gates and flip-flops to design an asynchronous modulo-15 counter.
- 4.42** Use sequential circuit modules to design an asynchronous modulo-65 counter.
- 4.43** Use sequential circuit modules to design a synchronous modulo-80 counter.
- 4.44** Use a shift register module to design an eight-state ring counter. Provide an asynchronous initialize control signal.
- 4.45** Use a shift register module to design a 14-state ring counter. Provide an asynchronous initialize control signal.
- 4.46** Use a shift register module to design an eight-state twisted-ring counter. Provide an asynchronous initialize control signal.
- 4.47** Use a shift register module to design a 14-state twisted-ring counter. Provide an asynchronous initialize control signal.
- 4.48** Use counter and decoder modules to design an eight-state ring counter equivalent to that of Problem 4.44. Provide an asynchronous initialize control signal.
- 4.49** Use counter and decoder modules to design a 14-state ring counter equivalent to that of Problem 4.45. Provide an asynchronous initialize control signal.
- 4.50** Design three equivalent timing signal generators using the counters of Problems 4.44, 4.46, and 4.48. The outputs of the three circuits should generate a pulse on the first and fifth clock pulses after the initialize signal. The sequence should repeat every eight clock pulses.

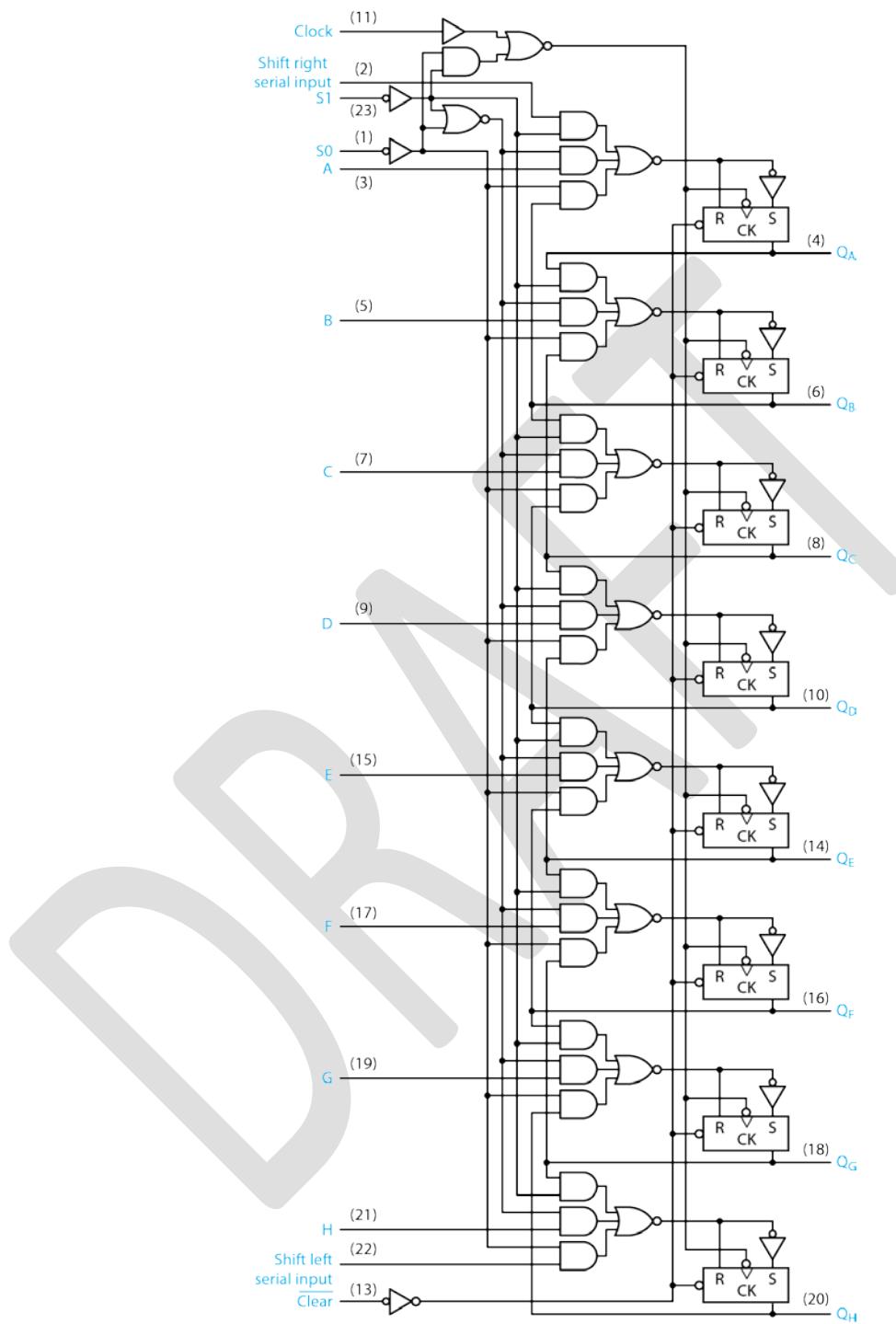


Figure P4-37: The SN74198 8-bit bidirectional shift register.

4.51 Design three equivalent timing signal generators using the counters of Problems 4.45, 4.47, and 4.49. The outputs of the three circuits should match the example of Fig. 4.71b, except that the timing waveforms will be repeated every 14 clock pulses instead of every 16 pulses as shown in Fig. 4.71b.

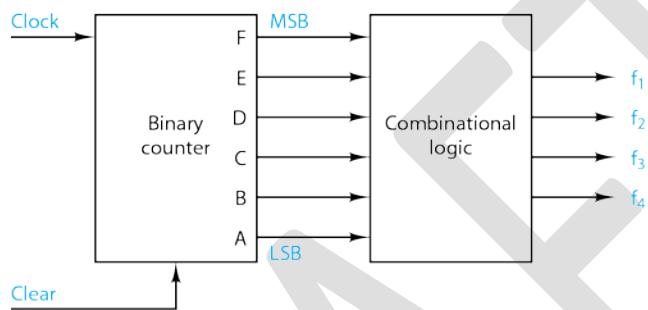
4.52 Design a timing generator using the structure of Fig. P4.47a. The unit should generate four signals as shown in Fig. P4.47b to meet the following specification:

f_1 goes low on clock pulses 2, 9, 17, 38, and 60

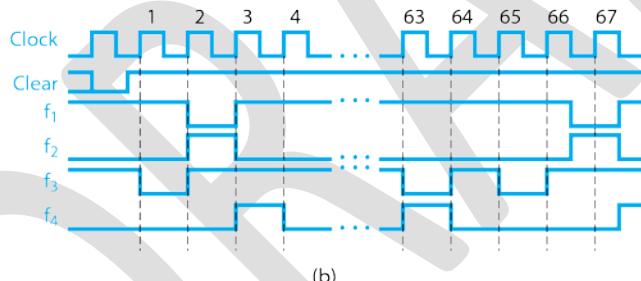
f_2 goes high on clock pulses 2, 8, 15, 35, and 56

f_3 goes low on clock pulses 1, 8, 16, 37, and 63

f_4 goes high on clock pulses 3, 27, 39, 41, and 63



(a)



(b)

Figure P4.52: (a) Block diagram. (b) Timing diagram.

Digital Logic Circuit Analysis and Design, 2nd Edition

Nelson, Carroll, Nagle, and Irwin

Chapter 5 Synchronous Sequential Logic Circuit Analysis and Design

(7/31/2016)

Table of Contents

5.0 Learning Objectives.....	1
5.1 Analysis of Sequential Circuits.....	1
5.1.1 Circuit Analysis Using State Diagrams and State Tables	2
5.1.2 Analysis of Sequential Circuit Logic Diagrams.....	3
5.1.3 Summary	20
5.2 Design of Synchronous Sequential Circuits	21
5.2.1 Design Procedure	22
5.2.2 Flip-flop Input Tables	26
5.2.3 Design Examples	21
5.2.4 One-hot Finite-State Machine Design Method	56
5.3 State Reduction in Sequential Circuits	63
5.3.1 Redundant States.....	63
5.3.2 State Reduction in Completely Specified Circuits	65
5.4 Summative Design Examples.....	71
5.4.1 Drink Vending Machine Control Unit	71
5.4.2 Binary Multiplier	74
5.4.3 Traffic Light Controller	78
5.5 Summary and Review Questions	92
5.6 Collaboration Exercises	93
5.7 Problems	98

This page intentionally left blank.

Chapter 5 Synchronous Sequential Logic Circuit Analysis and Design

(7/13/2016)

As we have seen in the previous chapter, memory devices can be designed from simple gates, and employed with combinational logic to build many interesting digital functions. As we progressed from simple to more complicated functions, we used a combination of ingenuity and experience to combine smaller modules into larger ones. For example, we used latches to build flip-flops, flip-flops to build shift registers, and shift registers to build twisted-ring counters. In another example, we built a binary counter out of JK flip-flops by exploiting our knowledge of the operation of the JK flip-flop in its toggle mode.

Suppose that a digital designer is given an arbitrary state diagram, or other behavioral description of a sequential circuit, and asked to find a schematic diagram for a hardware implementation. Ingenuity and knowledge of a particular type of flip-flop will not easily solve this problem. We call this the *general synchronous sequential circuit synthesis problem*. In this chapter we examine methods and tools to solve such problems.

5.0 Learning Objectives

In this chapter, we examine synchronous sequential circuit design from two perspectives. First, we wish to analyze a given sequential circuit to determine its behavior, which can be described in the form of a state diagram, HDL model, response to sequences of inputs, etc. Second, given a description of some desired circuit behavior, we wish to design and synthesize a synchronous sequential circuit to realize that behavior using standard gates and flip flops, or HDLs. Students completing the chapter will gain the following knowledge and skills.

1. Ability to analyze the behavior of a synchronous sequential circuit.
2. Ability to design and synthesize a synchronous sequential circuit to produce a specified behavior.

5.1 Analysis of Sequential Circuits

Before we dive into the synthesis problem, let us first reverse the process. That is, suppose we are given a schematic diagram of a synchronous sequential circuit and asked to determine and describe its operation.

Analysis is the process of determining the behavior of a given circuit or circuit model in the form of its output response to a given input sequence. To do so, it is most convenient to first create a model, for example, a state table, state diagram or HDL description of the given circuit. This process will first be illustrated by a series of examples and then summarized in the form of a step-by-step procedure.

5.1.1 Circuit Analysis Using State Diagrams and State Tables

Let us begin by examining the operation of a synchronous sequential circuit as described by a state diagram.

Example 5.1. The state diagram given in Figure 5.1 models a synchronous sequential circuit.

Assume that the circuit has negative-edge-triggered flip-flops for memory elements. Determine the behavior of the circuit by finding the response of the circuit to input sequence 001110110.

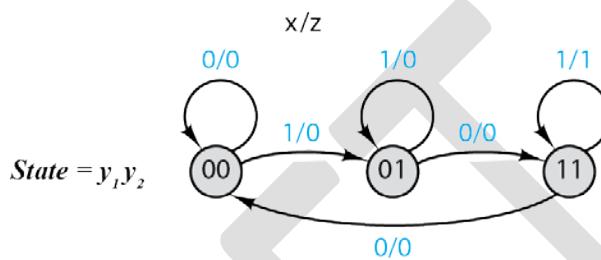


Figure 5.1: State diagram model of a synchronous sequential circuit.

From the state diagram, we can construct a timing diagram that illustrates the circuit's behavior in response to the input sequence 001110110, assuming that the circuit starts in state 00. The desired timing diagram is shown in Figure 5.2 and will now be explained.

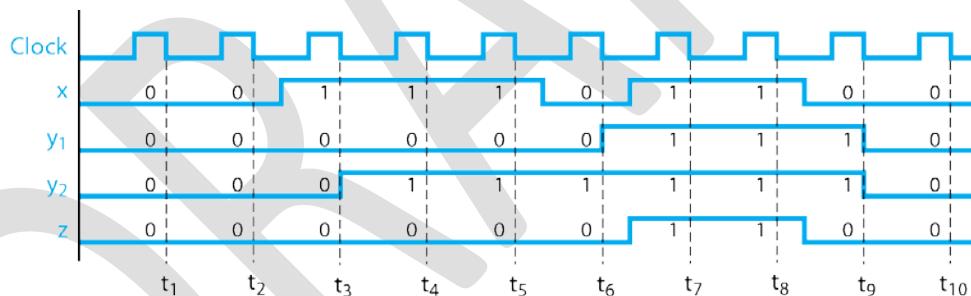


Figure 5.2: Timing diagram for a synchronous sequential circuit.

First, a clock signal is used to synchronize input signals and state changes. Since the memory elements are assumed to be negative-edge-triggered flip-flops, the state variables y_1 and y_2 change value only on $1 \rightarrow 0$ transitions of the clock signal, which are shown as dashed lines at times t_1, t_2, \dots, t_{10} in Fig. 5.2. The values of the state variables following a clock transition are determined by the value of x and the values of the state variables y_1 and y_2 at the instant the clock transition occurs. However, since the state diagram is that of a Mealy model circuit, the output z is not synchronized with the clock. Output z is simply a combinational function of the input x and the state variables y_1 and y_2 ; therefore, z may change when any of these three signals change.

For example, at times t_1 and t_2 we see that $y_1 = 0$, $y_2 = 0$, and $x = 0$. The state diagram in Fig. 5.1 shows that when the circuit is in state $[y_1y_2] = 00$ and $x = 0$, then the circuit loops back to (i.e. remains in) state 00 with $z = 0$. At time t_3 , x has changed to 1; the state diagram shows that when the circuit is in state $[y_1y_2] = 00$ and $x = 1$, the state

of the circuit will change to $[y_1y_2] = 01$ on the next clock transition, while output z remains 0. The rest of the timing diagram can be derived in a similar manner. Note in the state diagram that output $z = 1$ only when the state of the circuit is $[y_1y_2] = 11$ and $x = 1$; this occurs between times t_6 and t_9 in the timing diagram.

The previous example illustrates the following important point. A timing diagram can be constructed that represents the behavior of a circuit modeled by a state diagram, state table, or HDL model, given the type of circuit, the type of memory element, an input sequence, and a starting state. However, the timing of the input signal transitions are not precisely known unless the input sequence is itself given in the form of a timing diagram. Otherwise, we know only that each input value in a sequence occurs in time to be stable at the time of the next clock transition.

5.1.2 Analysis of Sequential Circuit Logic Diagrams

Let us now consider the problem of analyzing a synchronous sequential circuit defined by a logic diagram. To analyze the behavior of such a sequential circuit it is helpful to first determine the state table and/or state diagram that defines its operation. From this state table or diagram we can then determine the response of the circuit to any given input sequence.

Let us consider the sequential circuit shown in Figure 5.3. This sequential circuit is built of AND, OR, and NOT gates and a D flip-flop, and operates in a synchronous manner under the control of an external clock. Since the memory element in this circuit is a positive-edge-triggered D flip-flop, the memory changes state only on a $0 \rightarrow 1$ transition of a clock pulse.

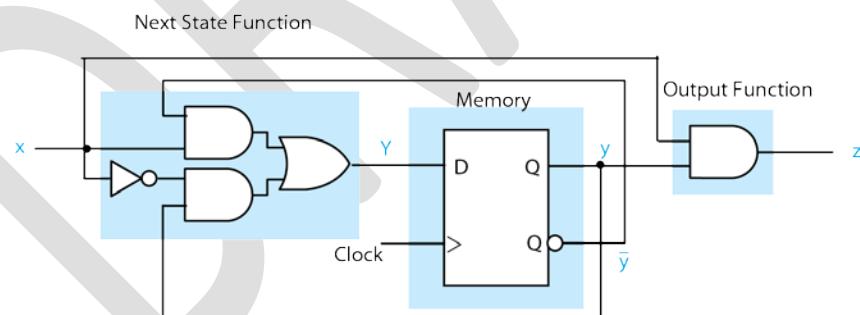


Figure 5.3: Sequential circuit example

Deriving the State Table and State Diagram

As discussed in section 4.1.2, the operation of a sequential circuit may be completely defined by a state table that lists all possible operating conditions. To derive the state table for a given sequential circuit, we begin by deriving information about the circuit from its logic diagram, including the number of states, flip-flop excitation equation(s), and output equations.

This sequential circuit in Fig. 5.3 has only one flip-flop and hence only two states, 0 and

1. The input, output, and state conditions for this circuit can be summarized as follows:

$$\begin{array}{ll} \text{Inputs: } & x = 0 \\ & x = 1 \end{array}$$

$$\begin{array}{ll} \text{States: } & y = 0 \\ & y = 1 \end{array}$$

$$\begin{array}{ll} \text{Outputs: } & z = 0 \\ & z = 1 \end{array}$$

The logic equation for the excitation input of the flip-flop and circuit output z are derived from the logic diagram as follows:

$$D = x\bar{y} + \bar{x}y = x \oplus y \quad (5.1)$$

$$z = xy \quad (5.2)$$

From section 4.2.2.1 we know that the characteristic equation of a D flip-flop is simply $Y = D$. Therefore, Eqn. (5.1) can be rewritten to indicate the next state of the flop flop:

$$Y = D = x \oplus y \quad (5.3)$$

A blank state table is shown in Figure 5.4a for a circuit with two states and two input values. The upper left-hand corner of the table corresponds to a present state $y = 0$ and the input $x = 0$. From Eq. (5.3), we find that these values of x and y produce the next state $Y = 0$, and from Eq. (5.2), they produce output $z = 0$. Hence, the entry in the upper-left block is $Y/z = 0/0$.

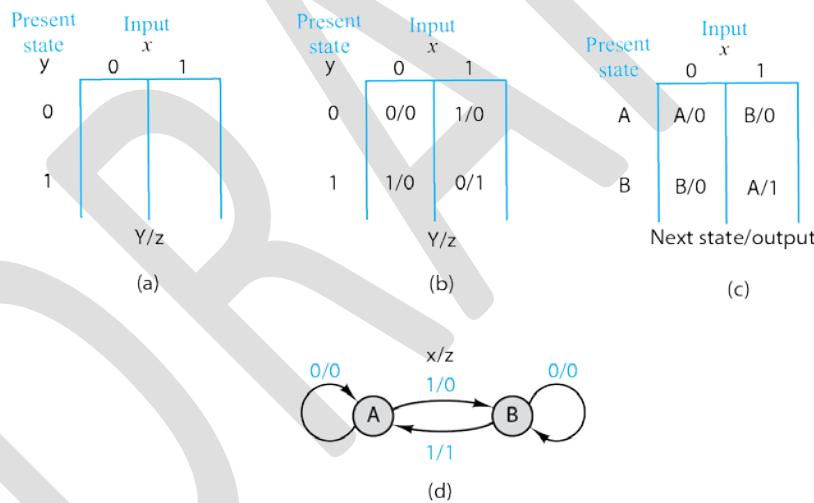


Figure 5.4: State table and diagram for the circuit of Figure 5.3.

(a) Blank table. (b) State transition table. (c) State table. (d) State diagram.

The initial conditions for the upper-right block are $y = 0$ and $x = 1$. Applying these values in Eqs. (5.3) and (5.2) yields $Y = 1$ and $z = 0$. The entry in this block is therefore $Y/z = 1/0$. The two lower block entries are determined in a similar manner. The results of the analysis are shown in the completed state transition table shown in Fig. 5.4b.

We sometimes replace the present state vectors y by symbols to simplify notation, especially when the use of symbolic state names can provide insight into the operation of the circuit. For example, in Fig. 5.4b, we may represent the states as follows:

$$y = [y] = [0] \equiv A$$

$$y = [y] = [1] \equiv B$$

We call this mapping from a symbolic state to a binary value the *state assignment*. Replacing the binary values of the state y with the corresponding symbols results in the state diagram in Figure 5.4c.

As described in section 4.1.2, the information in a state table can also be presented in the form of a state diagram. It is often easier to study circuit behavior by examining its state diagram. Figure 5.4d shows the state diagram corresponding to the state table of Fig. 5.4c.

Deriving the State Table from Karnaugh Maps

It is a straight-forward process to derive the state transition table shown in Fig. 5.4c directly from the circuit logic equations, using K-maps. K-maps for next state and output equations (5.3) and (5.2) are shown in Figs. 5.5a and b. The state table is constructed by simply merging the two K-maps into the state transition table, shown in Fig. 5.5c. Note that this state transition table is identical to the one in Fig. 5.4b. The binary values of the states can then be replaced with their assigned symbols to create the state table shown in Fig. 5.4c, from which the state diagram of Fig. 5.4d can be derived.

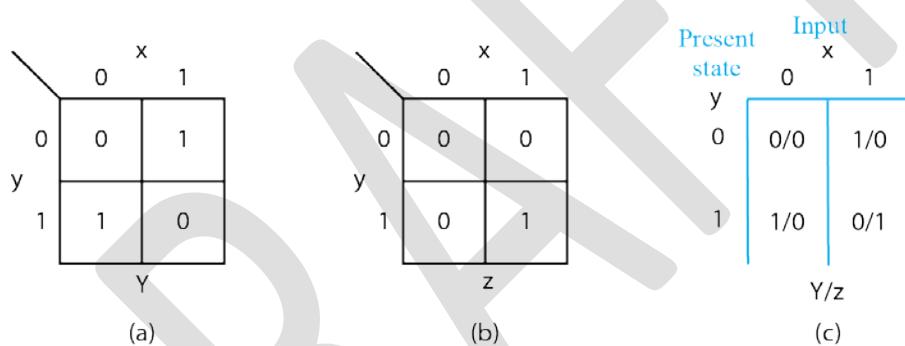


Figure 5.5: K-maps for the circuit of Figure 5.3.

(a) Map for $Y = f(x, y) = x \oplus y$. (b) Map for $z = f(x, y) = xy$. (c) Maps merged into a state transition table.

Expressing Sequential Circuit Behavior with an HDL Model

Synchronous sequential circuits can be described using structural or behavioral HDL models. There are a number of advantages to developing an HDL model of a sequential logic circuit, including the ability to simulate the operation of the circuit to study its behavior. A structural model would be composed of statements corresponding to the logic gates and flip-flop in the circuit shown in Figure 5.3. A behavioral model describes the finite state machine (FSM) of the circuit corresponding to the state table or state diagram derived in Figure 5.4. Note that a *case* statement can be used to conveniently describe the next state and output relations specified in a state table, with the input tested in *if-else* statements. Both structural and behavioral models of the circuit in Example 5.3 are shown below. The structural model employs a D flip-flop model defined separately.

```
//Circuit of Figure 5.3, Structural Verilog model
module Circuit5_3structural (
    input x, Clock,
    output z);
    wire x_not,y,y_not,d,a1,a2;
    not (x_not,x);
    and (a1,y_not,x);
    and (a2,x_not,y);
    or (d,a1,a2);
    Dflip_flop (d,Clock,y1,y1_not);
    and (z,x,y);
endmodule
```

```
//Circuit of Figure 5.3, Behavioral Verilog Model
module Circuit5_3behavioral (
    input x, Clock, CLR,
    output z);
    reg y;
    parameter A = 1'b0, B = 1'b1;
    always @ (posedge Clock, posedge CLR)
        if (CLR == 1) y <= A;
        else
            case (y)
                A: if (~x) y <= A; else y <= B; //transitions from state A
                B: if (~x) y <= B; else y <= A; //transitions from state B
            endcase
    assign z = ((y == B) && (x == 1)) ? 1 : 0; //output z = f(x,y)
endmodule
```

```
-- Circuit of Figure 5.3, Behavioral VHDL Model
entity Example5_1 is
  port ( x: in bit;    -- input
         clk: in bit;   -- clock
         z: out bit); -- output
end Example5_1;
architecture behavior of Example5_1 is
  type states is (A,B);      -- two states
  signal y: states := A;     -- initial state A for simulation
begin
  z <= '1' when y = B and x = '1' else '0'; -- Mealy model output

  process(clk) begin          -- State changes
    if clk'event and clk = '1' then --rising clk edge
      case y is
        when A => if x = '0' then -- current state A
          y <= A;           -- next state A if x=0
        else
          y <= B;           -- next state B if x=1
        end if;
        when B => if x = '0' then -- current state B
          y <= B;           -- next state B if x=0
        else
          y <= A;           -- next state A if x=1
        end if;
      end case;
    end if;
  end process;
end;
```

Circuit Behavior Expressed as a Timing Diagram

Let us now examine the behavior of the circuit in Figure 5.3 by deriving a timing diagram that shows the response of the circuit to a sequence of inputs, given that the circuit begins in a specified starting state. For example, suppose we wish to determine the response of the circuit to an input sequence

$$x = 01101000$$

given that the circuit's starting state is $y = 0$.

We begin by drawing the clock waveform, as shown in Fig. 5.6, to use as a time reference. Since the flip-flop in Fig. 5.3 is positive-edge triggered, we note that the flip-flop state changes from its current state y to next state Y , as determined by the flip-flop's D input at the instant the clock makes a $0 \rightarrow 1$ transition. The clock period, Δt , is selected to allow the circuit sufficient time for state changes and for new inputs to be applied before the next state transition is triggered.

Signals x , y , Y and z are actually continuous in time, i.e. they would be more precisely indicated in the timing diagram as continuous functions $x(t)$, $y(t)$, $Y(t)$, and $z(t)$. However, we are usually interested in the values of these signals primarily at discrete times, corresponding to the instants of the clock transitions:

$$x(k\Delta t), y(k\Delta t), Y(k\Delta t), z(k\Delta t) \text{ for } k = 0, 1, 2, \dots$$

where $k\Delta t$ is the time at which the k^{th} clock transition occurs.

To simplify analysis, the x -axis in Fig. 5.6 is labelled $t/\Delta t$. Note that the $0 \rightarrow 1$ clock transitions occur at times Δt , $2\Delta t$, $3\Delta t$, ..., which correspond to $t/\Delta t = 1, 2, 3, \dots$, with the k^{th} clock transition at $t/\Delta t = k$. We will also suppress the time notation and simply represent the signals as x , y , Y , and z .

From Eqs. (5.2) and (5.3), we can build the rest of the timing diagram in Figure 5.6. During the interval $t/\Delta t = 0$ to $t/\Delta t = 1$, the input is $x=0$ and the present state is $y=0$. Hence,

$$Y = D = x \oplus y = 0 \oplus 0 = 0$$

$$z = xy = 0 \cdot 0 = 0$$

Therefore, the clock pulse at $t/\Delta t = 1$ clocks the next state $Y=0$ into the D flip-flop. During the period from $t/\Delta t = 1$ to $t/\Delta t = 2$, the present state is $y=0$ and the input changes to $x=1$. Therefore, at $t/\Delta t = 2$,

$$Y = x \oplus y = 1 \oplus 0 = 1$$

$$z = xy = 1 \cdot 0 = 0$$

As the clock pulse at $t/\Delta t = 2$ occurs, the state of the sequential device will change to 1. As soon as the state changes, we have $y=1$ and $x=1$, and since $z = xy$, output z changes to 1. In a similar manner, the remainder of the timing diagram is determined.

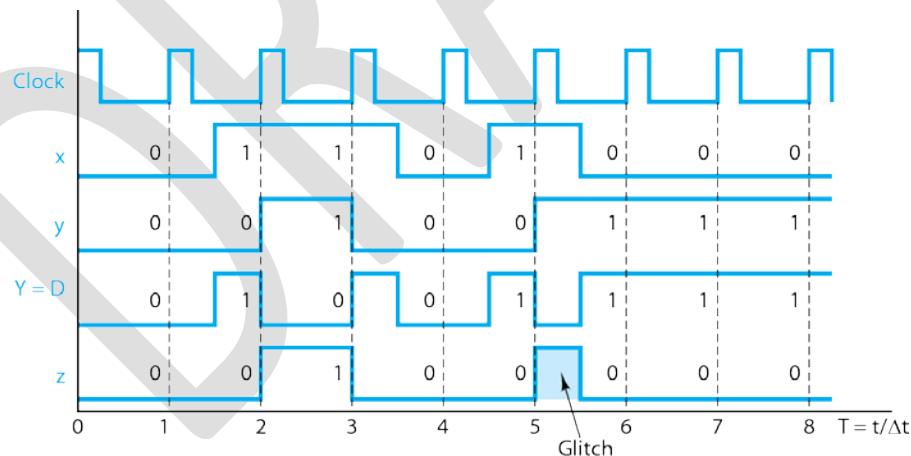


Figure 5.6: Timing diagram for the circuit of Fig. 5.3a.

From the timing diagram we observe that the output sequence is

$$z = 00100000$$

The timing diagram also shows a momentary change, or *glitch*, in the output at time $t/\Delta t = 5$. Since $z = xy$, the change in state variable y at time $t/\Delta t = 5$ results in $z=1$ at that time, and then the subsequent change in input x causes z to return to its desired value of 0. Note that the width of this pulse on z depends on when input x changes.

Also note on the timing diagram that changes in Y occurred at $t/\Delta t = 3$ and at $t/\Delta t = 5$ that did not affect the next state of the circuit. In each of these cases the value of Y changed when the state changed, and again when x changed, prior to the next clock transition. Only the value of Y at the time of the clock transition determines the next state of y . Therefore, the momentary changes of Y at $t/\Delta t = 3$ and $t/\Delta t = 5$ are ignored.

This completes our introductory synchronous sequential circuit example. The following analysis procedure summarizes the processes illustrated in the preceding examples.

Synchronous Sequential Circuit Analysis Procedure

- Step 1.** If a state table or state diagram is given, proceed to steps 6 or 7, respectively. Otherwise, continue. Use combinational logic analysis techniques to determine the flip-flop input equations and the circuit output equations. Go to step 7 if only a timing diagram is needed.
- Step 2.** Construct K-maps for all the logic equations from step 1.
- Step 3.** Combine the K-maps for all flip-flop input equations into a single map.
- Step 4.** Using the characteristic equations of the flip-flops, construct a next-state map.
- Step 5.** Combine the next-state map and the output maps into a single map. This step results in a binary state table.
- Step 6.** Construct a binary state diagram and/or HDL behavioral model from the binary state table if desired. Otherwise, go to step 7.
- Step 7.** Draw a timing diagram showing the clock, the given input sequence, and the starting state.
- Step 8.** On the timing diagram, derive the waveforms for the flip-flop input(s) and flip-flop state(s) for all the remaining circuit input values.
- Step 9.** On the timing diagram, derive the waveforms of the circuit output(s).

The following examples will illustrate this procedure. We begin by considering the problem of analyzing a clocked sequential circuit containing a JK flip-flop. This analysis differs from the previous example in that we must utilize knowledge of JK flip-flop operation to determine the next state of the flip-flop for a given excitation input.

Example 5.2. Determine the behavior of the circuit in Fig. 5.7 by developing a timing diagram, state table, and state diagram. The timing diagram should show the circuit response to input sequence $x = 01101000$, with the circuit beginning in state $y = 0$.

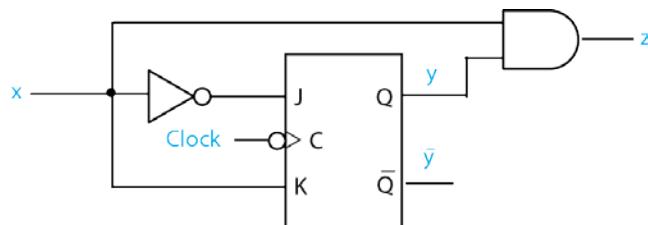


Figure 5.7: Synchronous sequential circuit with a JK flip-flop.

Step 1

We begin by deriving excitation and output equations from the logic diagram of Fig. 5.7. This sequential circuit has a single flip-flop and hence only two states, 0 and 1. The input, output, and state conditions for this network are summarized as follows:

$$\begin{array}{ll} \text{Inputs: } & x = 0 \\ & x = 1 \end{array}$$

$$\begin{array}{ll} \text{States: } & y = 0 \\ & y = 1 \end{array}$$

$$\begin{array}{ll} \text{Outputs: } & z = 0 \\ & z = 1 \end{array}$$

The logic equations for the circuit in Fig. 5.7 are:

$$\begin{aligned} z &= xy \\ J &= \bar{x} \\ K &= x \end{aligned} \tag{5.4}$$

where J and K determine the next state of the flip-flop for a given present state y and input x . From Eqs. (5.4) we derive the state diagram of the circuit in the following steps.

Step 2

K-maps for Eqs. (5.4) are shown in Figs. 5.8a and b; the K-map variables are the input x and the present state y . The maps for J and K define the signal to the J and K inputs of the flip-flop when a clock pulse is present for various conditions of x and y , and the map for z defines the output for various conditions of x and y .

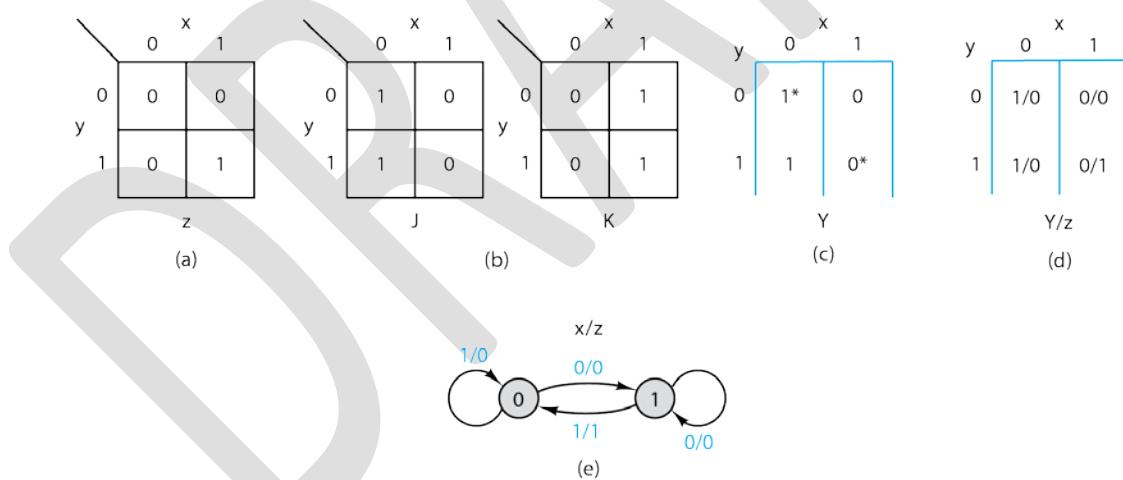


Figure 5.8: Derivation of the state table from K-maps. (a) Output K-map. (b) Excitation K-maps. (c) State transition table. (d) Binary state table. (e) State diagram.

Step 3

Since this circuit contains a single flip-flop, there is only one set of excitation input K-maps, as shown in Fig. 5.8b. If there had been multiple flip-flops, we would combine their excitation input K-maps in this step.

Step 4

Using the maps for J and K in Fig. 5.8b, and our knowledge of JK flip-flop operation, we now find the state transition table for the next state Y , shown in Fig. 5.8c. Recall that $J = 1$ and $K = 0$ forces the flip-flop state to 1 when a clock pulse occurs, whereas $J = 0$ and $K = 1$ forces the flip-flop state to 0. The next-state, Y , for each combination of present state and input is determined by examining the corresponding entries in the K-map for J and K . The asterisks in Fig. 5.8c indicate where the J and K inputs caused the state to change.

The initial conditions for the upper-left block of the K-map in Fig. 5.8b are $y = 0$ and $x = 0$, resulting in $J = 1$ and $K = 0$. Therefore, the flip-flop state will be changed from 0 to 1 on the next clock transition, and we enter 1 as the value of next state Y for initial conditions $y = 0$ and $x = 0$, as shown in the state transition table in Fig. 5.8c.

Likewise, the initial conditions for the upper-right block of the K-map are $y = 0$ and $x = 1$, producing $J = 0$ and $K = 1$; therefore the next state will be forced to 0 (in this case, the state will not change), and we enter 0 as upper-right entry for next state Y in Fig. 5.8c. In a similar manner, we can determine the next states for the bottom row of the state transition table.

We can also derive the state transition table without first deriving K-maps for J and K by utilizing the characteristic equation of the JK flip-flop to derive the next state Y as follows. The characteristic equation of a JK flip-flop, Eq. (4.26), can be written as.

$$Y = J\bar{y} + \bar{K}y$$

Substituting $J = \bar{x}$ and $K = x$ from Eq. (5.4) gives:

$$\begin{aligned} Y &= \bar{x}\bar{y} + \bar{x}y \\ &= \bar{x} \end{aligned}$$

Now, plotting the K-map of $Y = \bar{x}$ produces the state transition table of Fig. 5.8c.

Step 5

Combining the state transition table and output K-maps of Figs. 5.8a and c yields the binary state table in Fig. 5.8d, which completely defines the behavior of the sequential circuit of Fig. 5.7.

Step 6

From the state table of Fig. 5.8d, we can derive the state diagram shown in Fig. 5.8e. Note that the state diagram contains the exact same information as the state table, but in graphical form. Since this is a Mealy model circuit, output z is shown on each arc, being a function of both present state y and input x .

From either the state table of Fig. 5.8d or the state diagram of Fig. 5.8e, we can derive the corresponding HDL behavioral models as follows.

```
//Circuit of Example 5.2, Behavioral Verilog Model
module Example5_2 (
    input x, Clock, CLR,           //declare input variables
    output z);                   //declare output variable
    reg y;                       //declare state variable
    parameter A = 1'b0, B = 1'b1; //make state assignment
    always @ (negedge Clock, negedge CLR) //detect negative edge of Clock or CLR
        if (CLR == 0) y <= A;          //go to state A if CLR is high
        else
            case (y)    //derive next states as specified in state table
                A: if (~x) y <= B; else y <= A; //transitions from state A
                B: if (~x) y <= B; else y <= A; //transitions from state B
            endcase
        assign z = ((y == B) && (x == 1)) ? 1 : 0; //output z = f(x,y)
endmodule
```

```
-- Circuit of Example 5.2, Behavioral VHDL Model
entity Example5_2 is
    port ( x: in bit;      -- input
           clk: in bit;     -- clock
           z: out bit);   -- output
end Example5_2;
architecture behavior of Example5_2 is
    signal y: bit := '0'; -- two states, initially '0' for simulation
begin
    z <= '1' when y = '1' and x = '1' else '0'; -- Mealy model output

    process(clk) begin -- State changes
        if clk'event and clk = '0' then -- falling edge of clk
            case y is
                when '0' => if x = '0' then -- current state 0
                    y <= '1'; -- next state 1 if x=0
                else
                    y <= '0'; -- next state 0 if x=1
                end if;
                when '1' => if x = '0' then -- current state 1
                    y <= '1'; -- next state 1 if x=0
                else
                    y <= '0'; -- next state 0 if x=1
                end if;
            end case;
        end if;
    end process;
end;
```

Step 7

A specific timing diagram for a given sequential circuit is valid for only one given input sequence and starting state. For this example, the specified input sequence is

$$x = 01101000$$

and the starting state is $y = 0$. The clock and input sequence are drawn in Fig. 5.9. For the input sequence, each value of x specifies the value present on input x at the instant of the falling edge of the clock signal, since the flip-flop is negative-edge triggered. Actually, x can change at any time prior to the clock transition, so long as it is stable at the time of the clock transition.

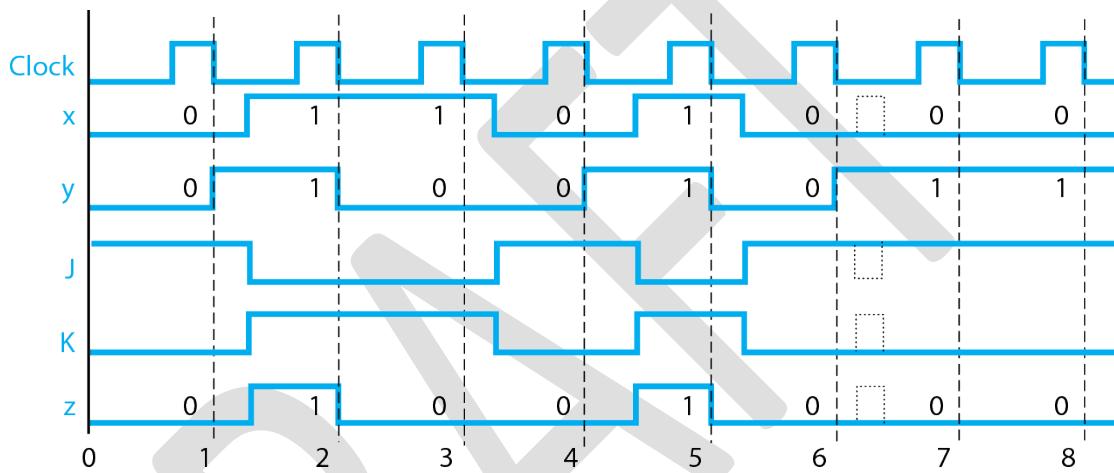


Figure 5.9: Timing diagram for the JK flip-flop synchronous sequential circuit.

Step 8

From either the state table of Fig. 5.8d or the state diagram of Fig. 5.8e, we can determine and will cause the state to change to $y = 1$. Therefore, the value of y changes from 0 to 1 at time 1 on the diagram. At time 2, $y = 1$ and $x = 1$; from the state diagram we see that if the circuit is in state $y = 1$, then input $x = 1$ will change the state from 1 to 0. The rest of the waveform for state variable y can be derived in a similar manner.

Suppose an asynchronous input pulse occurs after time $t/\Delta t = 7$ (see dotted pulse in Fig. 5.9); although this asynchronous input pulse causes a change in the J and K inputs to the flip-flop, the pulse has no effect on the flip-flop because it occurs while the clock is in logic state 0, so the flip-flop output is unchanged.

Step 9

To derive the output signal z , we can either use the logic equation for z derived in step 1 or we can use the information on the state diagram or state table. The output logic equation is $z = xy$; therefore output $z = 1$ when both $x = 1$ and $y = 1$, which occurs at times 2 and 5, and momentarily during the glitch between times 6 and 7, and $z = 0$ otherwise. This is illustrated in the timing diagram of Fig. 5.9.

Note that steps 7-9 can also be performed by using a logic circuit simulator to simulate operation of the schematic diagram of Fig. 5.7, or an HDL simulator to simulate the HDL model derived in Step 6.

Symbolic state table/diagram

As was discussed in the previous example, sometimes the state variable binary codes are replaced by symbols to simplify notation; if we code the sequential circuit's states as

$$y = [y] = [0] \equiv A$$

$$y = [y] = [1] \equiv B$$

the state table in Fig. 5.10a and the state diagram of Fig. 5.10b are obtained from the corresponding diagram and table in Fig. 5.8d and e. At this point we may use the state diagram to determine the response of the circuit to an input sequence, where the state y is designated by its symbol. The following shows input x , state y , and output z at the time of each clock transition.

Time	0	1	2	3	4	5	6	7	8
$x =$	0	1	1	0	1	0	0	0	0
$y =$	A	B	A	A	B	A	B	B	B
$z =$	0	1	0	0	1	0	0	0	0

The starting state is A ($y=0$) and the final state is B ($y=1$). Note that this behavior is identical to the timing diagram of Fig. 5.9, which was to be expected.

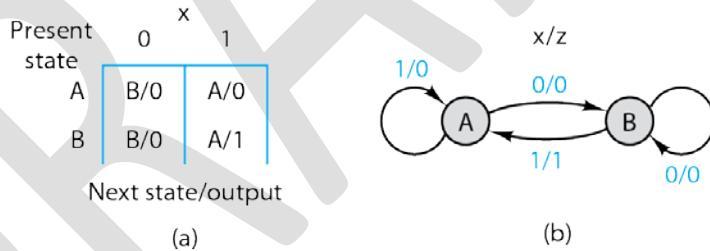


Figure 5.10: State table and diagram for the JK flip-flop sequential circuit example. (a) State table. (b) State diagram.

Example 5.3. Analyze the circuit shown in Fig. 5.11.

The circuit contains two negative-edge-triggered JK flip-flops and hence has four states. Unlike the previous two examples, which were Mealy model circuits, in this circuit we see that output z is a function only of the two state variables, y_1 and y_2 , and therefore this is a Moore model circuit.

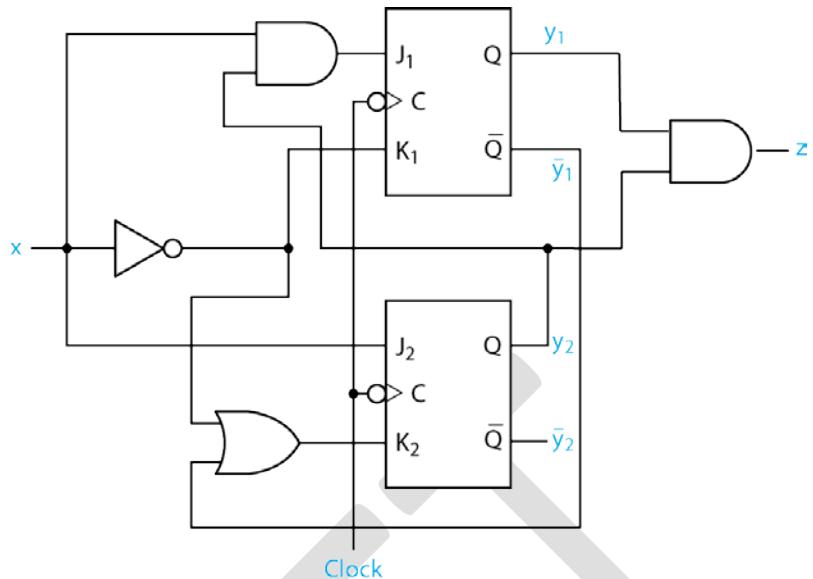


Figure 5.11: Synchronous sequential circuit with JK flip-flops.

To facilitate studying the circuit's behavior, let us begin by deriving its state table. The equations that describe the circuit's operation are

$$\begin{aligned} J_1 &= xy_2, & J_2 &= x, & z &= y_1y_2 \\ K_1 &= \bar{x}, & K_2 &= \bar{x} + \bar{y}_1. \end{aligned}$$

The K-maps for these equations are given in Fig. 5.12; note that the K-map for output z represents a function of only the two state variables, y_1 and y_2 .

For convenience in determining the next states, Y_1 and Y_2 , of the flip-flops, the K-maps for J and K are combined into a single table, shown in Fig. 5.13a. Using the J and K signals and our knowledge of JK flip-flop operation, we can determine the state changes, yielding the transition table of Fig. 5.13b. For example, consider the second row of the merged K-map in Fig. 5.13a, which corresponds to current state $y_1y_2 = 01$. In the column for $x = 1$, we see that $J_1K_1 = 10$, which will set the state of the first flip-flop to 1, and $J_2K_2 = 11$, which will toggle the state of the second flip-flop from 1 to 0, resulting in $Y_1Y_2 = 10$, as shown in the corresponding row and column of Fig. 5.13b. In a similar manner we can derive the rest of the table in Fig. 5.13b.

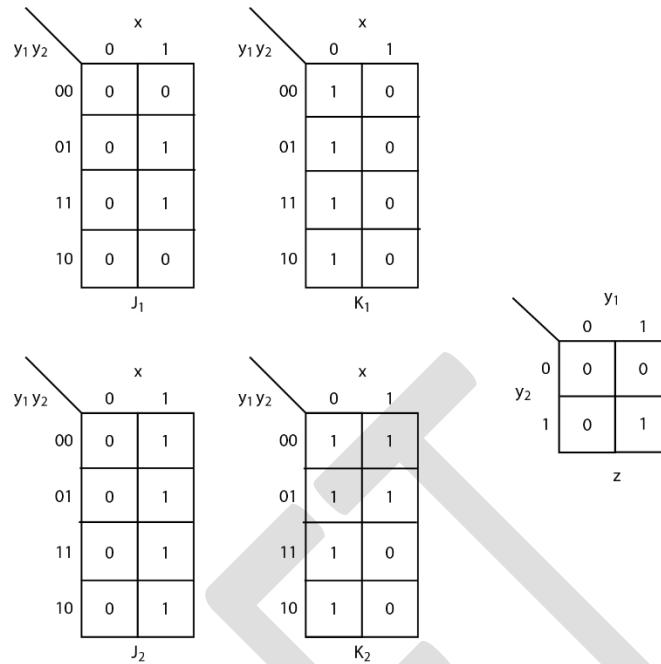
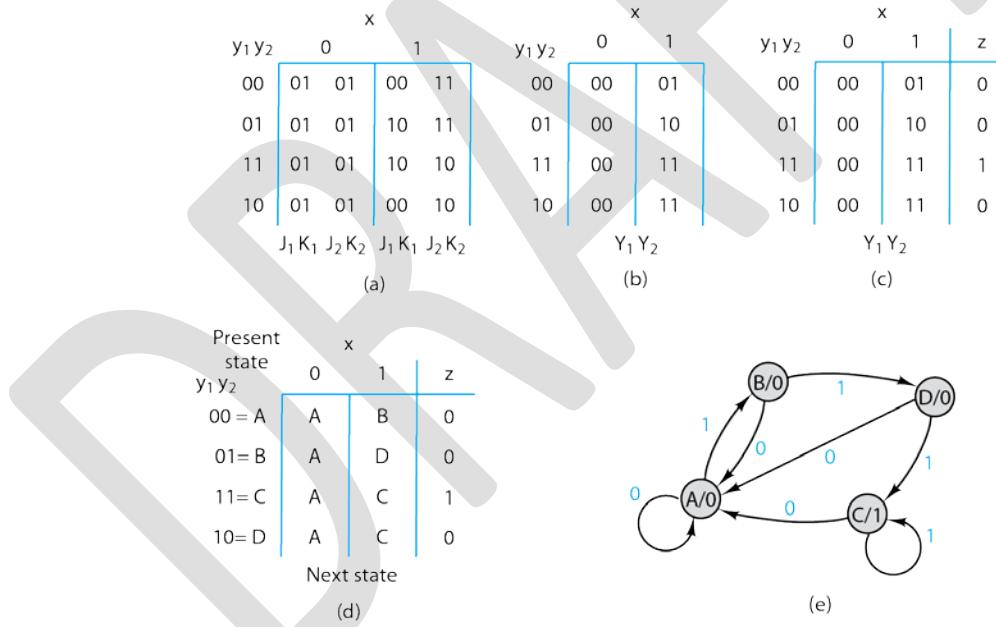


Figure 5.12: K-maps for logic equations that describe the example circuit.

Figure 5.13: Combining the K-maps into the state table. (a) Merge maps for J and K . (b) Determine next states $Y_1 Y_2$. (c) Merge next states with z . (d) State table with symbolic state names. (e) State diagram.

Alternatively, we can derive the next state of each flip-flop directly from the characteristic equation of the JK flip-flop, without plotting K-maps for J and K . Using the characteristic equation, Eq. (4.26), for each JK flip-flop, and substituting the logic equations for the flip-flop excitation inputs J and K , we have:

$$\begin{aligned}Y_1 &= J_1 \bar{y}_1 + \bar{K}_1 y_1 \\&= (xy_2) \bar{y}_1 + (\bar{x}) y_1 \\&= xy_2 + xy_1\end{aligned}$$

$$\begin{aligned}Y_2 &= J_2 \bar{y}_2 + \bar{K}_2 y_2 \\&= x\bar{y}_2 + (\bar{x} + \bar{y}_1) y_2 \\&= x\bar{y}_2 + xy_1\end{aligned}$$

Plotting Y_1 and Y_2 on the same K-map produces the map shown in Fig. 5.13b.

Combining the transition table with the K-map values for the output z yields the binary state table shown in Fig. 5.13c. Using the state assignment:

$$\begin{array}{ll} [y_1y_2] = 00 = A & [y_1y_2] = 11 = C \\ [y_1y_2] = 01 = B & [y_1y_2] = 10 = D \end{array}$$

we can derive the symbolic state table shown in Fig. 5.13d, and the state diagram in Fig. 5.13e.

An analysis similar to those presented previously yields the timing diagram of Fig. 5.14, which is shown for an input sequence and starting state of

$$x = 0011110$$

$$y_1y_2 = 10$$

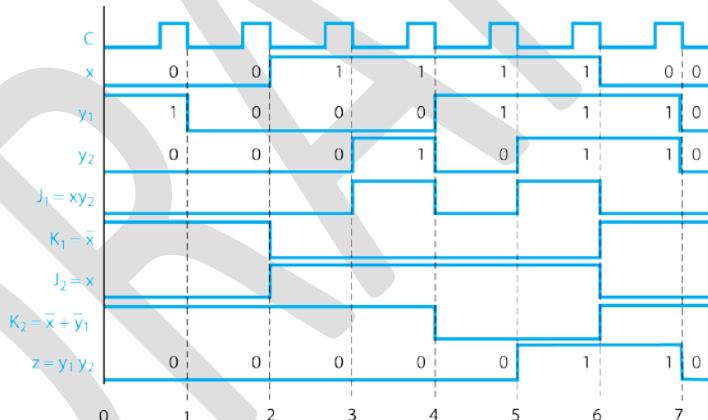


Figure 5.14: Timing diagram for the circuit of Fig. 5.11.

We may also wish to derive a structural and/or behavioral HDL model of the circuit. A structural model would be composed of statements corresponding to the logic gates and flip-flops in the circuit shown in Figure 5.11. A behavioral model describes the FSM of the circuit corresponding to the state table given in Figure 5.13c or d. Note that a *case* statement can be used to conveniently describe the next state and output relations specified in a state table. Both structural and behavioral models of Example 5.3 are shown below. The structural model employs a JK flip-flop model defined separately.

```
//Example 5.3 Structural Verilog model
module example5_3structural (
    input x, Clock,
    output z);
    wire x_not,y1,y1_not,y2,y2_not,j1,k1,j2,k2;
    not (x_not,x);
    and (j1,x,y2);
    or (k2,x_not,y1_not);
    and (z,y1,y2);
    JKflip_flop_N (j1,x_not,Clock,y1,y1_not);
    Jkflip_flop_N (x,k2,Clock,y2,y2_not);
endmodule
```

//declare input variables
//declare output variable
//declare internal nodes
//compute x_not
//compute J1
//compute K2
//compute z
//compute y1 and y1_not
//compute y2 and y2_not

```
//Example 5.3, Behavioral Verilog Model
module Example5_3 (
    input x, Clock, CLR,
    output z);
    reg [1:0] state;
    parameter A = 2'b00, B = 2'b01, C = 2'b11, D = 2'b10; //make state assignment
    always @ (negedge Clock, negedge CLR) //detect negative edge of Clock or CLR
        if (CLR == 0) state <= A; //go to state A if CLR is low
        else
            case (state) //derive next states as specified in state table
                A: if (~x) state <= A; else state <= B;
                B: if (~x) state <= A; else state <= D;
                C: if (~x) state <= A; else state <= C;
                D: if (~x) state <= A; else state <= C;
            endcase
    assign z = (state == C) ? 1'b1 : 1'b0; //derive output as specified in table
endmodule
```

```
-- Example 5.3, Behavioral VHDL Model
entity Example5_3 is
    port ( x:  in bit;      -- input
           clk: in bit;      -- clock
           z:  out bit);   -- output
end Example5_3;
architecture behavior of Example5_3 is
    type states is (A,B,C,D);  -- four states
    signal y: states := A;     -- initial state A for simulation
begin
    z <= '1' when y = C else '0';  -- Moore model output

    process(clk) begin      -- State changes
        if clk'event and clk = '0' then  -- falling clk edge
            case y is
                when A => if x = '0' then  -- current state A
                    y <= A;  -- next state A if x=0
                else
                    y <= B;  -- next state B if x=1
                end if;
                when B => if x = '0' then  -- current state B
                    y <= A;  -- next state A if x=0
                else
                    y <= D;  -- next state D if x=1
                end if;
                when C => if x = '0' then  -- current state C
                    y <= A;  -- next state A if x=0
                else
                    y <= C;  -- next state C if x=1
                end if;
                when D => if x = '0' then  -- current state D
                    y <= A;  -- next state A if x=0
                else
                    y <= C;  -- next state C if x=1
                end if;
            end case;
        end if;
    end process;
end;
```

5.1.3 Summary

In this section we have examined methods for analyzing various types of synchronous sequential circuits. The reader should now be able to take the logic diagram or HDL model for a sequential circuit, apply the analysis techniques of this chapter, and derive a state table and/or timing diagram description of the circuit. In the next section we shall reverse this procedure and find logic circuit diagrams or HDL model to realize a specified state table or diagram.

DRAFT

5.2 Design of Synchronous Sequential Circuits

In the previous section, the analysis of sequential circuits was introduced using several examples. In each case a logic diagram was given and the resulting analysis determined the circuit's behavior by producing a state table, state diagram, HDL model, and/or timing diagram. In this section the reverse procedure, the synthesis process, will be addressed [3–9]. For a given state table, state diagram, or HDL model, well-defined tools will be used to generate an equivalent logic diagram for the sequential circuit in question. All sequential circuits in this chapter are to have clocked memory elements and are thus *synchronous* sequential circuits.

The synthesis of a synchronous sequential circuit begins with the specification of the desired behavior, in the form of a state table (or diagram). Circuits for which every next state/output pair is completely defined are termed *completely specified circuits*. In our previous work, each time we analyzed a sequential circuit, we found that the resulting state table was completely specified. However, occasionally a circuit is connected in such a manner that the state table cannot be completely defined (for example, a particular combination of state variables and/or inputs cannot occur, or a circuit causes a 1 input to both the *S* and *R* terminals of an SR flip-flop). Those for which several next states or outputs are arbitrary are termed *incompletely specified circuits*. An example of each type of circuit is shown in Fig. 5.15. The following sections deal with completely specified circuits. Incompletely specified circuits will be considered later in the chapter.

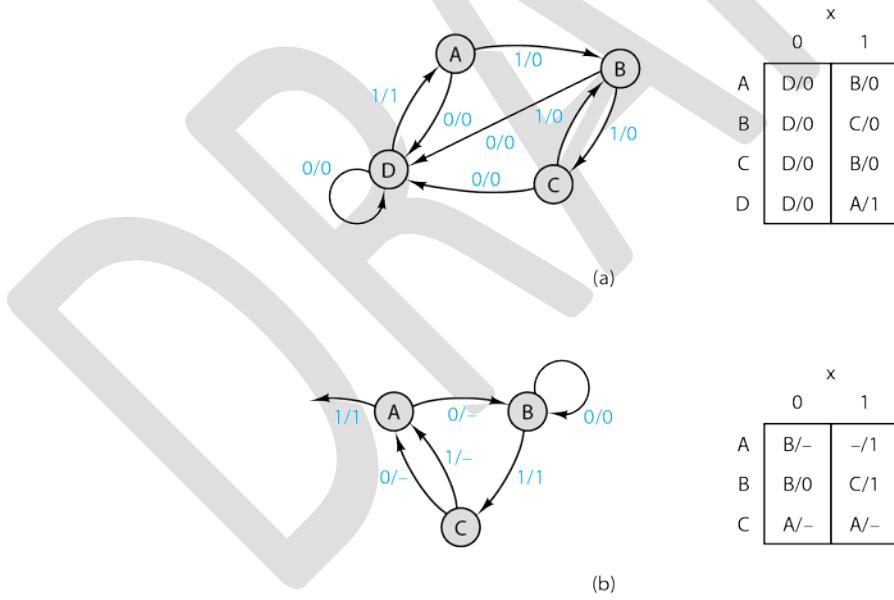


Figure 5.15: Types of sequential circuits. (a) Completely specified circuit. (b) Incompletely specified circuit.

5.2.1 Design Procedure

The procedure for designing synchronous sequential circuits will be introduced by a simple example.

Example 5.4. Let us find a D flip-flop realization for the sequential circuit defined in the state table of Fig. 5.16a.

First we must adopt some coding scheme to represent the symbolic states. This process is called *state assignment*. Let us arbitrarily choose the code in Fig. 5.16b.

If we replace the symbolic states with their code equivalents, a binary state table is obtained or, in other words, the *state transition table* shown in Fig. 5.16c. The transition table contains all the necessary information for generating the switching functions for the combinational logic portion of the circuit. Then we separate the transition table into an output K-map and D flip-flop input K-maps as shown in Figs. 5.16d and e, respectively. The flip-flop input maps are often called *excitation maps*, since they produce logic equations for the flip-flop excitation inputs. Note that the D input K-maps also represent the next states of their respective flip-flops, since the characteristic equation of a D flip-flop is simply $Y = D$. From the excitation and output maps,

$$\begin{aligned}D_1 &= Y_1 = y_1 \bar{y}_2 + xy_2 \\D_2 &= Y_2 = \bar{x}y_1 + x\bar{y}_1 = x \oplus y_1 \\z &= x\bar{y}_1 y_2 + \bar{x}y_1 \bar{y}_2\end{aligned}$$

The logic diagram for the completed design is presented in Fig. 5.16f. In this case the combinational logic is realized using two levels of NAND gates.

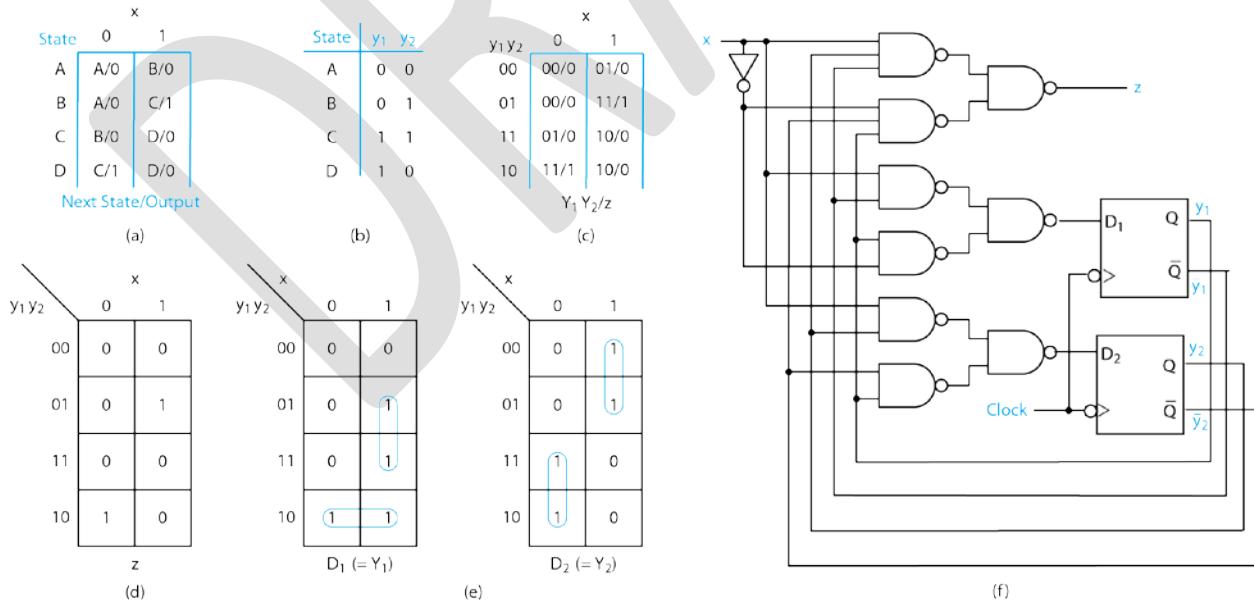


Figure 5.16: An introductory example. (a) State table. (b) State assignment. (c) Transition table. (d) Output K-map. (e) Excitation K-maps. (f) Logic diagram.

In the foregoing example, several questions remain unanswered. How do we choose the state assignment? What happens if the design requires another type of flip-flop? How do we know whether the given state table is the best one to use? Many state tables have redundant states that can be eliminated.

Before we attempt to answer these questions, let us outline the synthesis procedure for synchronous sequential logic circuits.

Synchronous Sequential Circuit Synthesis Procedure

- Step 1.** From a word description of the problem, derive a state diagram or a state table.
- Step 2.** Verify that the state table produces the desired behavior for expected input sequences.
- Step 3.** Use state reduction techniques to find the state table of a minimum-state equivalent circuit.
- Step 4.** Choose a state assignment.
- Step 5.** Generate the state transition table.
- Step 6.** Determine the memory device or flip-flop to be used and find the flip-flop excitation maps.
- Step 7.** From the excitation maps, produce the switching logic equations. Also, form output maps and determine the output logic equations.
- Step 8.** Draw the logic diagram of the sequential circuit using the derived logic equations and the chosen memory devices.

The first step requires intuition on the part of the logic designer and must be learned through trial and error experience. It is often the case that desired behavior can be more easily represented graphically by a state diagram, with the state table then derived from the state diagram. In addition, it might also be beneficial to derive an HDL model from the state diagram or state table, to use for simulation of the circuit or as an input to an automated synthesis tool.

The second step verifies that the state table represents the desired behavior, with any errors corrected before continuing with the rest of the synthesis procedure. From the state diagram or table, we can manually determine the next state and output and/or produce a timing diagram to verify the correctness of the circuit's response to various input sequences, as was done in Example 5.1. For example, assume the circuit modeled by the state table of Fig. 5.16a begins in state A and that we wish to verify the response of the circuit to input sequence 0101101110. We can use the state table to determine the next state and output for each input as follows.

Time	=	0	1	2	3	4	5	6	7	8	9	10
$x =$		0	1	0	1	1	0	1	1	1	0	
$y =$		A	A	B	A	B	C	B	C	D	D	C
$Y =$		A	B	A	B	C	B	C	D	D	C	
$z =$		0	0	0	0	1	0	1	0	0	0	1

Alternatively, we can design a behavioral HDL model of the state table/diagram and use this model to simulate the operation of the circuit to verify its behavior. The following is a behavioral HDL model of the state table of Fig. 5.16a.

```
//Behavioral Verilog model of Example 5.4
module Example_5_4_Verilog (                                //name the module
    input x, Clock, CLR,                                     //declare the inputs
    output reg z);                                         //declare the output
    reg [1:0] state, nextstate;                            //declare state variables
    parameter A = 2'b00, B = 2'b01, C = 2'b11, D = 2'b10; //assign states
    always @ (negedge Clock, negedge CLR)                  //detect change of Clock or CLR
        if (CLR == 0) state <= A;                         //place FSM in state A if CLR is low
        else state <= nextstate;                           //make state change
    always @ (state, x)                                    //detect change in state or input x
        case (state)                                      //derive next state as specified in state table
            A: if (~x) nextstate <= A; else nextstate <= B;
            B: if (~x) nextstate <= A; else nextstate <= C;
            C: if (~x) nextstate <= B; else nextstate <= D;
            D: if (~x) nextstate <= C; else nextstate <= D;
        endcase
    always @ (state, x)                                    //detect change in state or input x
        case (state)                                      //derive output z as specified in state table
            A,C: z <= 0;
            B: z <= x;
            D: z <= ~x;
        endcase
endmodule
```

```
-- Behavioral VHDL Model of Example 5.4
entity Example5_4 is
    port ( x:  in bit;      -- input
           clk: in bit;      -- clock
           z:  out bit);   -- output
end Example5_4;
architecture behavior of Example5_4 is
    type states is (A,B,C,D);  -- four states
    signal y: states := A;     -- initial state A for simulation
begin
    z <= '1' when (y = B and x = '1') or (y = D and x = '0') else '0'; -- Mealy model output

    process(clk) begin      -- State changes
        if clk'event and clk = '0' then  -- falling clk edge
            case y is
                when A => if x = '0' then  -- current state A
                    y <= A;  -- next state A if x=0
                else
                    y <= B;  -- next state B if x=1
                end if;
            end case;
        end if;
    end process;
end architecture;
```

```

when B => if x = '0' then -- current state B
    y <= A; -- next state A if x=0
else
    y <= C; -- next state C if x=1
end if;
when C => if x = '0' then -- current state C
    y <= B; -- next state B if x=0
else
    y <= D; -- next state D if x=1
end if;
when D => if x = '0' then -- current state D
    y <= C; -- next state C if x=0
else
    y <= D; -- next state D if x=1
end if;
end case;
end if;
end process;
end;

```

Step 3 of the synthesis procedure is used to minimize the number of memory elements required to build a circuit by eliminating unnecessary states from the state table. We shall see that the successful designer employs standard algorithms and rules of thumb in completing this step. We shall examine these algorithms and rules later in this chapter, after we have gained some experience in the fundamentals of synchronous sequential circuit design.

In step 4, we may either choose an arbitrary state assignment or else apply one of several algorithms that select optimal state assignments to minimize the amount of combinational logic needed to realize a circuit. The relationship between the number of states (N_s) and the number of flip-flops (N_{FF}) is given by the expression

$$2^{N_{FF}-1} < N_s \leq 2^{N_{FF}} \quad (5.5)$$

For example, a four-state circuit would require two flip-flops, a ten-state circuit would require four flip-flops, and so on. Algorithms for optimal state assignment are beyond the scope of this book, but can be found in [1-3]. For simple circuits, we may wish to try several arbitrary state assignments to determine which produces the best result.

In Step 5 we simply replace symbolic state names with their assigned binary codes to transform the symbolic state table to the binary state transition table.

Step 6 requires analysis of the characteristics of the selected flip-flop types, as presented in section 4.2.2, to derive the flip-flop excitation maps.

The skills required in step 7 are found in Chapter 2, and hence it is assumed that the reader is familiar with them.

Step 8 is the obvious conclusion of the synthesis procedure and is included for completion.

Now that the synthesis procedure has been defined, we shall first examine techniques for designing logic circuits to realize a given state table or state diagram. Then we will address the problem of creating state tables and diagrams from verbal problem descriptions. Methods to remove redundant states in step 3 will be discussed later in this chapter.

5.2.2 Flip-flop Input Tables

In the introductory synthesis example of Fig. 5.16, an edge-triggered D flip-flop was employed for circuit realization. In particular, the characteristics of the D flip-flop were used to generate the flip-flop excitation maps from the transition table. The transition table defines the necessary state transitions for each memory flip-flop. A *flip-flop input table* for each type of flip-flop memory element may be derived from the characteristic equation of the flip-flop and used to determine the required excitation inputs. Consider the input tables of Fig. 5.17, which represent the characteristics of the D and JK flip-flop types discussed in section 4.2.2. The notation in this table is as follows: t is the time at which the clock signal is activated; $Q(t)$ is the state of the flip-flop at the instant the clock signal is activated; $Q(t + \Delta t)$ is the next state of the flip-flop after the clock signal has been activated.

State transitions		Required inputs		State transitions		Required inputs	
$Q(t)$	$Q(t + \Delta t)$	$D(t)$		$Q(t)$	$Q(t + \Delta t)$	$J(t)$	$K(t)$
0	0	0		0	0	0	d
0	1	1		0	1	1	d
1	0	0		1	0	d	1
1	1	1		1	1	d	0

Figure 5.17: Flip-flop input tables. (a) D flip-flop. (b) JK flip-flop.

The D flip-flop is convenient to employ because its next state is simply its present input; hence the excitation maps are taken directly from the transition table. However, in some cases D flip-flops might not be available for use, or might not produce optimal circuits. A JK flip-flop requires the application of the input table of Fig. 5.17b in the production of the excitation K-maps. This will be illustrated by the following example.

Example 5.5. Let us repeat the implementation of the sequential circuit of Example 5.4, but in this case using JK flip-flops.

Assume the same state assignment used in Fig. 5.16b. The transition table remains unchanged and is reproduced in Fig. 5.18a for convenience.

For a JK flip-flop implementation, the input table of Fig. 5.17b is used to obtain the excitation tables of Fig. 5.18b. One particular state transition is emphasized both in the transition table and in the corresponding entry of the excitation table; that is, the transition of y_2 shown in Fig. 5.18a from 1 to 0 requires $J_2 = d$ and $K_2 = 1$, as illustrated in Fig. 5.18b. Other JK values are determined in a similar manner. Next the excitation tables are transformed into excitation K-maps and the required minimal Boolean logic equations are derived as follows:

$$\begin{array}{ll} J_1 = xy_2 & K_1 = \bar{x}y_2 \\ J_2 = x \oplus y_1 & K_2 = x \otimes y_1 = \bar{J}_2 \end{array}$$

The logic diagram is shown in Fig. 5.19. Note that the output logic is not dependent on the flip-flop type, and is therefore unchanged from Fig. 5.16f.

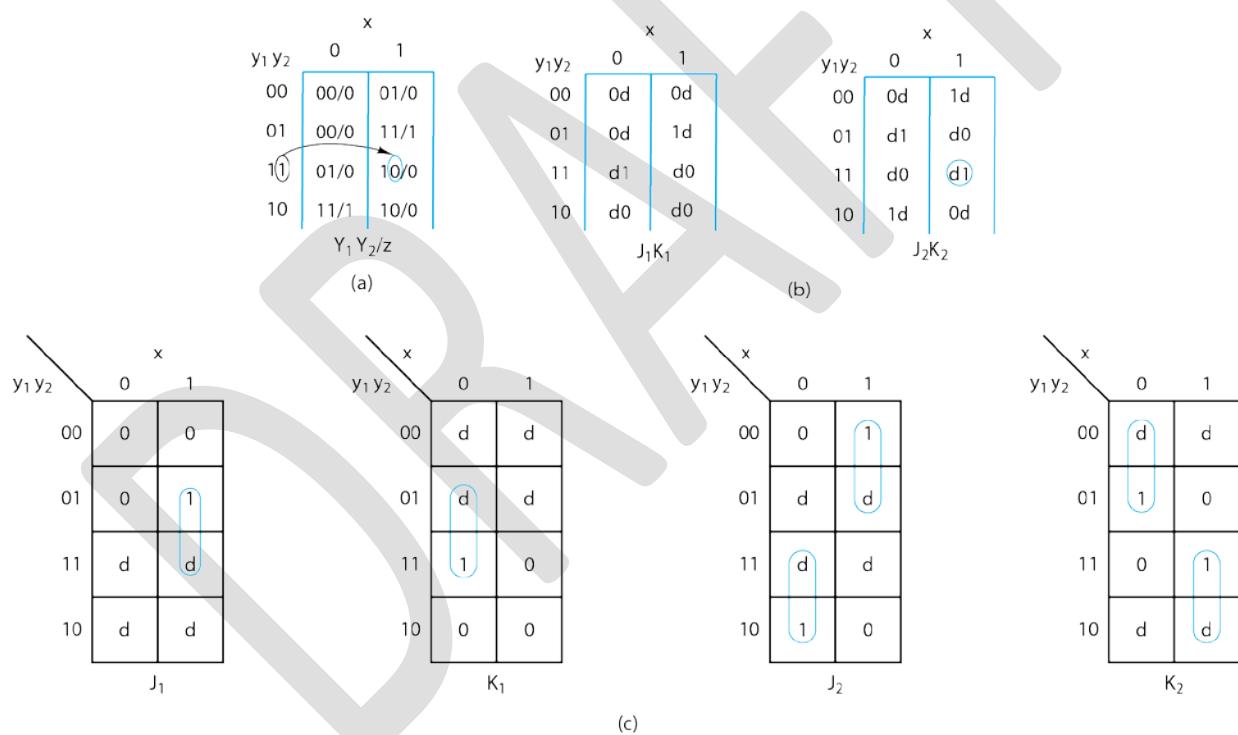


Figure 5.18: Generating the excitation maps for JK flip-flops. (a) Transition table.
(b) Excitation tables. (c) Excitation maps.

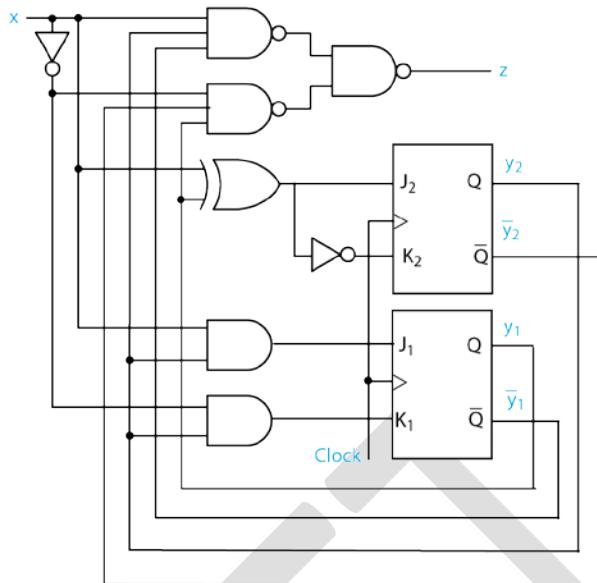


Figure 5.19: JK flip-flop implementation.

5.2.3 Design Examples

The synthesis procedure for synchronous sequential circuits was presented earlier in this chapter as an eight-step process. The reader should now be able to complete all the steps except step 3, state reduction. Let us postpone the development of the theory of this topic and, instead, dwell on some practical applications of the other synthesis steps for the moment. This change of pace has two purposes: first, to allow the reader to practice her or his skills of state table/state diagram formulation and excitation map generation and, second, to provide a basis from which to present the state table reduction and state assignment problems.

The design procedure will be demonstrated by several examples. In each example it is assumed that the circuit is under the control of a periodic clock pulse and that transitions in the circuits occur only as initiated by this clock.

Sequence Recognizers

Sequence recognizers are synchronous sequential circuits that produce a designated response on their outputs when specific sequences of input values are detected. Each set of values in an input sequence is assumed to arrive prior to the active transition of the clock, with successive values in a sequence arriving during consecutive clock periods.

Example 5.6. Design a synchronous sequential circuit with one input x and one output z that recognizes the input sequence 01, i.e. the values of x at two successive clock transitions are 0 and 1.

This circuit can be used to recognize a 0 to 1 transition on the input x. In other words, the circuit should produce an output sequence $z = 01$ whenever the input sequence $x = 01$ occurs. For example, if the input sequence is

$$x = 01110100111$$

then the output sequence will be

$$z = 01000100100$$

Step 1

The first step in the design procedure is the construction of a state diagram that represents the input/output behavior just described. The diagram is constructed as shown in Fig. 5.20. First it is assumed that the circuit is in some starting state *A* and that the first input is a 1. Since a 1 is not the first element in the input string to be recognized, the circuit remains in state *A* and yields an output $z=0$, as shown in Fig. 5.20a. However, if the circuit is in the initial state *A* and the input is a 0, then, because this input is the first symbol in the string to be recognized, the circuit moves to a new state *B* and produces an output of 0, as shown in Fig. 5.20b. Now suppose that the circuit is in state *B* and that the input symbol is a 0. Because this is the first symbol in the string to be recognized, and not the second symbol in the sequence 01, the circuit merely remains in state *B* and yields an output of 0, as shown in Fig. 5.20c. Finally, if the circuit is in state *B* and the next input symbol is a 1, the circuit returns to state *A* and produces an output $z=1$ to indicate that the sequence 01 has occurred. This final diagram, shown in Fig. 5.20d, results in the input/output sequence given previously and thus recognizes the input sequence $x=01$. The state table that corresponds to the final state diagram is shown in Fig. 5.21a. We shall assume that this state table contains the minimum number of states possible for this circuit.

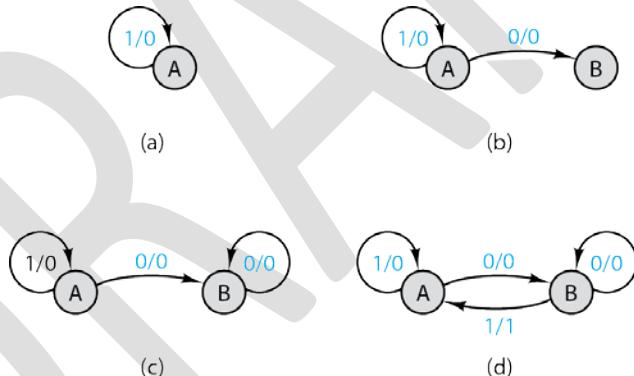


Figure 5.20: Sequence 01 recognizer state diagram.

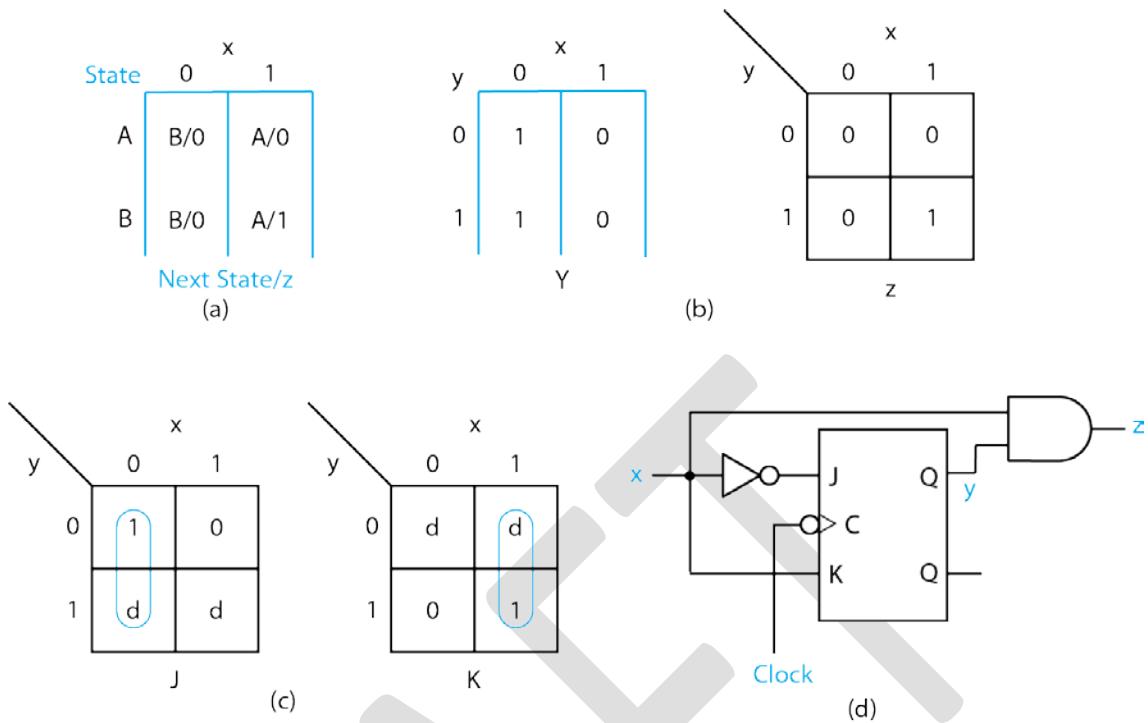


Figure 5.21: Synthesis of a sequence 01 recognizer. (a) State table. (b) Transition table and output map. (c) Excitation maps. (d) Logic diagram.

Step 2

To verify the correctness of the design, we can use the state diagram or state table to determine the response of the circuit to the input sequence given above. Assuming that the circuit begins in state A, for each input in the given sequence we determine the next state and output from the state diagram/table, producing the sequence of states and outputs listed below. This verifies that the sequence 01 is recognized three times in this set of inputs.

Time	=	0	1	2	3	4	5	6	7	8	9	10	11
x =	0	1	1	1	0	1	0	0	0	1	1	1	
y =	A	B	A	A	A	B	A	B	B	A	A	A	
Y =	B	A	A	A	B	A	B	B	A	A	A	A	
z =	0	1	0	0	0	1	0	0	0	1	0	0	

We can also verify the design by writing and simulating a behavioral HDL model of the state table, as follows.

```
//Circuit of Example 5.6, Behavioral Verilog Model
module Example5_6 (
    input x, Clock, CLR,           //declare input variables
    output z);                   //declare output variable
    reg y;                      //declare state variable
    parameter A = 1'b0, B = 1'b1; //make state assignment
    always @ (negedge Clock, negedge CLR) //detect negative edge of Clock or CLR
        if (CLR == 0) y <= A;          //go to state A if CLR is high
        else
            case (y)    //derive next states as specified in state table
                A: if (~x) y <= B; else y <= A; //transitions from state A
                B: if (~x) y <= B; else y <= A; //transitions from state B
            endcase
    assign z = ((y == B) && (x == 1)) ? 1 : 0; //output z = f(x,y)
endmodule
```

```
-- Circuit of Example 5.6, Behavioral VHDL Model
entity Example5_6 is
    port ( x: in bit;      -- input
           clk: in bit;     -- clock
           z: out bit);   -- output
end Example5_6;
architecture behavior of Example5_6 is
    type states is (A,B);      -- two states
    signal y: states := A;    -- initial state A for simulation
begin
    z <= '1' when y = B and x = '1' else '0'; -- Mealy model output
    process(clk) begin           -- State changes
        if clk'event and clk = '1' then    --rising clk edge
            case y is
                when A => if x = '0' then -- current state A
                    y <= B;           -- next state B if x=0
                    else
                        y <= A;       -- next state A if x=1
                    end if;
                when B => if x = '0' then -- current state B
                    y <= B;           -- next state B if x=0
                    else
                        y <= A;       -- next state A if x=1
                    end if;
            end case;
        end if;
    end process;
end;
```

Using this model, the operation of the circuit can be simulated for the given input sequence, producing the timing diagram shown in Fig. 5.22a.

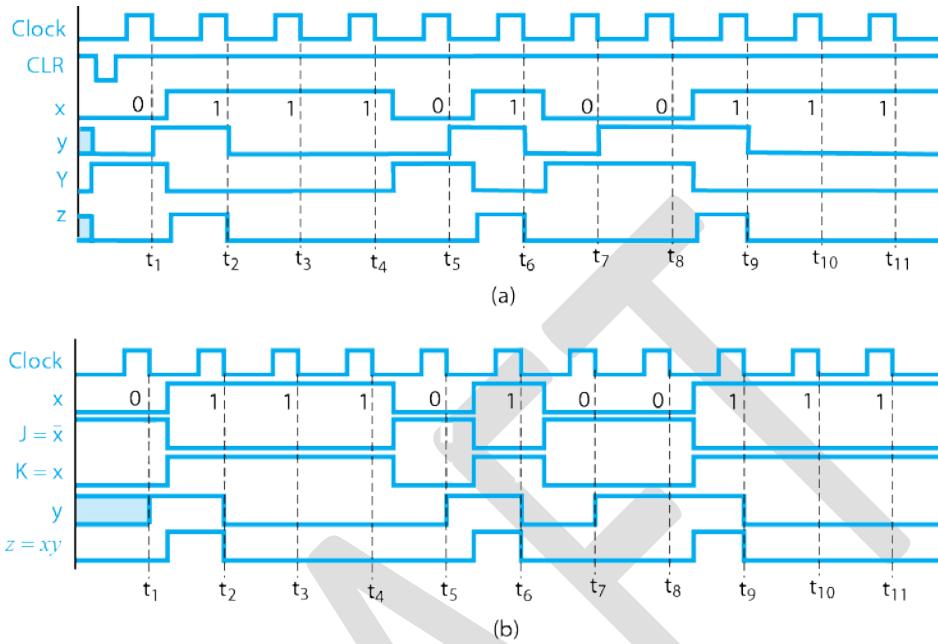


Figure 5.22. Verifying the circuit design. (a) Timing diagram from simulation of the HDL behavioral model. (b) Timing diagram from simulation of the HDL structural model.

Step 3

We will take up the problem of identifying redundant states later in this chapter, and simply assume, for now, that the state table of Fig. 5.21a contains the minimum number of states.

Step 4

The next step in the synthesis procedure is to determine the number of flip-flops required and the state assignment. Since the number of states in the state table of Fig. 5.21a is two, from Eq. (5.5) we see that we require a single flip-flop. The state assignment is arbitrarily chosen here as $A=0$ and $B=1$; it could, however, just as easily have been selected in the opposite manner ($A=1, B=0$), without affecting the behavior of the circuit.

Step 5.

Once the state assignment has been chosen, the state table in Fig. 5.21a can be redrawn as the transition table of Fig. 5.21b. Here y denotes the present state of the circuit, which is the current output of the flip-flop. The symbol Y denotes the next state of the circuit, that is, the output of the flip-flop after the occurrence of the next state transition. The K-map for the output is drawn separately to facilitate derivation of the output equation in step 7.

Step 6.

Suppose that we want to realize the circuit with JK flip-flops. The problem then becomes one of determining the proper signals on the J and K input lines to effect the transitions shown in Fig. 5.21b. Using the JK flip-flop input table of Fig. 5.17b, we may derive the excitation maps shown in Fig. 5.21c. For example, consider the transition in the upper-left corner of the transition table shown in Fig. 5.21b: $y = 0$, $x = 0$ and $Y = 1$. To effect a state change from $y = 0$ to 1, the signals that must appear on the set and reset lines are $J = 1$ and either $K = 0$ (to force the state to 0) or $K = 1$ (to toggle the state). Hence, these signals appear in the corresponding positions in the excitation maps of Fig. 5.21c. Next consider the state transition in the upper-right corner of the transition table: $y = 0$, $x = 1$ and $Y = 0$. Since no change in state is to occur, the signal on the J input must be $J = 0$, while the signal on the K input does not matter; that is, K is a don't-care. The reader should recall that a JK flip-flop will not change to the set state with $J = 0$ and $K = 0$ or with $J = 0$ and $K = 1$. The remaining blocks in the excitation maps are determined in a similar manner.

Step 7.

The output and excitation maps can now be used to derive the logic circuit equations:

$$\begin{aligned} J &= \bar{x}, \quad K = x \\ z &= xy \end{aligned}$$

Step 8.

The circuit obtained from these logic equations is shown in Fig. 5.21d. The reader may now check the circuit to see if it does indeed recognize the input sequence 01. A timing diagram with x shown as a level input signal that is not synchronized with the clock is drawn in Fig. 5.22b. This diagram can be generated manually, by studying the circuit diagram of Fig. 5.21d, or by simulating a schematic diagram or HDL structural model of the circuit. Note that the circuit operates as follows:

1. A logic 0 on x causes the JK flip-flop to set on the next falling edge of the clock.
2. The flip-flop then remains at logic 1 waiting for the input to change from 0 to 1.
3. Then x goes high, and the output z also goes high.
4. Finally, the flip-flop resets on the next falling edge of the clock.

So the three 0 to 1 transitions on x have been detected as shown by the three pulses on z . In addition, note that the states and outputs are identical in Figs. 5.22a and b, verifying that the logic circuit produces the behavior corresponding to the original state diagram of Fig. 5.20d.

Example 5.7. For practice, let us realize the circuit of the previous example with D flip-flops.

If a D flip-flop is to be used to implement the sequence recognizer, we must derive the excitation maps for the flip-flop shown in Fig. 5.23a. These map is derived from the transition table in Fig. 5.21b. Referring to the input table for a D flip-flop in Fig. 5.17a, recall that D is simply the next state Y . Therefore the D flip-flop excitation map is identical to the transition

table. From the excitation map in Fig. 5.23a, the logic equation for D is given by the expression

$$D = \bar{x}$$

The output equation is identical to that obtained earlier. The implementation of the logic equation with a D flip-flop is shown in Fig. 5.23b.

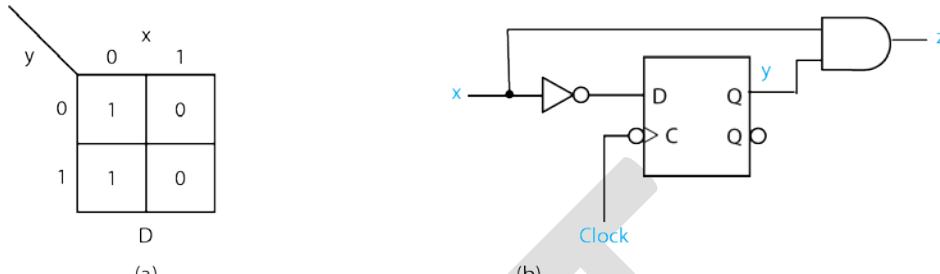


Figure 5.23: D flip-flop realization. (a) D flip-flop excitation map.
(b) D flip-flop implementation.

Example 5.8. Let us design a synchronous sequential circuit with one input line and one output line that recognizes the input string $x = 1111$, i.e. input x is 1 at four consecutive clock transitions. The circuit is also required to recognize overlapping sequences, as can be seen in the output string z that results from the following input sequence x :

$$\begin{aligned} x &= 1101111111010 \\ z &= 0000001111000 \end{aligned}$$

The state diagram and the corresponding reduced state table for the sequential circuit that will recognize the input string $x = 1111$ are shown in Figs. 5.24a and b. Note that if state A is assumed to be the initial state the circuit changes state every time an input $x = 1$ occurs, with the exception of the fourth and succeeding ones. Every time an $x = 0$ occurs, the circuit resets by returning to state A. States B, C, and D are entered when $x = 1$ at one, two, and three consecutive clock transitions, respectively. The loop with $x = 1$ at state D satisfies the overlapping input sequence criterion by producing a 1 at the output for each additional logic 1 at the input, since the condition represented by state D is that the previous three values of x were logic 1.

The state assignment for the example is arbitrarily chosen as follows:

$$\begin{array}{ll} [y_1y_2] = 00 = A & [y_1y_2] = 10 = C \\ [y_1y_2] = 01 = B & [y_1y_2] = 11 = D \end{array}$$

The resulting transition table is shown in Fig. 5.24c. The output map is shown in Fig. 5.24d.

As previously, to verify the behavior of the circuit, the behavior of the recognizer can be modeled using an HDL model, derived from the state table as follows. Simulation of this model can be shown to produce the output sequence listed above.

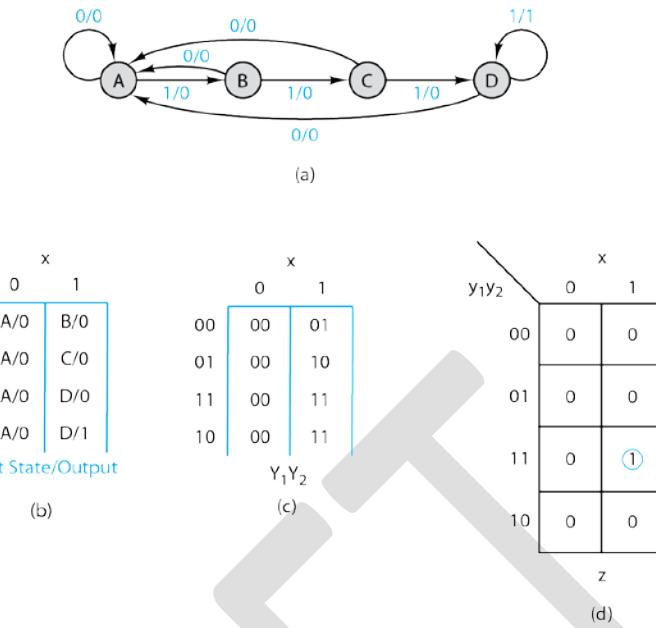


Figure 5.24: A sequence 1111 recognizer. (a) State diagram. (b) State table.
(a) Transition table. (d) Output map.

```
//Behavioral Verilog model of a sequence 1111 recognizer
module Example5_8 (
    input x, Clock, CLR,           //declare inputs
    output z);                   //and outputs
    reg [1:0] state, nextstate;   //declare state variables
    parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11; //assign states
    always @ (negedge Clock, negedge CLR) //detect Clock or CLR changes
        if (CLR == 0) state <= A;          //place in state A if CLR low
        else state <= nextstate;         //change state on negative edge of Clock
    always @ (state, x)             //detect state or input change
        case (state)                  //derive nextstate as specified in state table
            A: if (~x) nextstate <= A; else nextstate <= B;
            B: if (~x) nextstate <= A; else nextstate <= C;
            C: if (~x) nextstate <= A; else nextstate <= D;
            D: if (~x) nextstate <= A; else nextstate <= D;
        endcase
        assign z = (state == D) ? x : 0; //derive output as specified in state table
    endmodule
```

```
-- Behavioral VHDL Model of a sequence 1111 recognizer
entity Example5_8 is
    port ( x: in bit;      -- input
           clk: in bit;      -- clock
           z: out bit);     -- output
end Example5_8;
architecture behavior of Example5_8 is
    type states is (A,B,C,D);    -- four states
    signal y: states := A;       -- initial state A for simulation
begin
    z <= '1' when y = D and x = '1' else '0';  -- Mealy model output

    process(clk) begin      -- State changes
        if clk'event and clk = '0' then    -- falling clk edge
            case y is
                when A => if x = '0' then    -- current state A
                    y <= A;    -- next state A if x=0
                else
                    y <= B;    -- next state B if x=1
                end if;
                when B => if x = '0' then    -- current state B
                    y <= A;    -- next state A if x=0
                else
                    y <= C;    -- next state C if x=1
                end if;
                when C => if x = '0' then    -- current state C
                    y <= A;    -- next state A if x=0
                else
                    y <= D;    -- next state D if x=1
                end if;
                when D => if x = '0' then    -- current state D
                    y <= A;    -- next state A if x=0
                else
                    y <= D;    -- next state D if x=1
                end if;
            end case;
        end if;
    end process;
end;
```

The excitation maps for a D flip-flop realization of the circuit are given in Fig. 5.25a, and the corresponding logic circuit equations are

$$D_1 = y_1x + y_2x, \quad z = xy_1y_2$$

$$D_2 = y_1x + \bar{y}_2x$$

Excitation maps for a JK flip-flop realization of the circuit are given in Fig. 5.25b. The logic equations obtained from these K-maps are given next, and the hardware used to realize the equations is shown in Fig. 5.26:

$$J_1 = y_2x, \quad J_2 = x, \quad z = xy_1y_2$$

$$K_1 = \bar{x}, \quad K_2 = \bar{y}_1 + \bar{x}$$

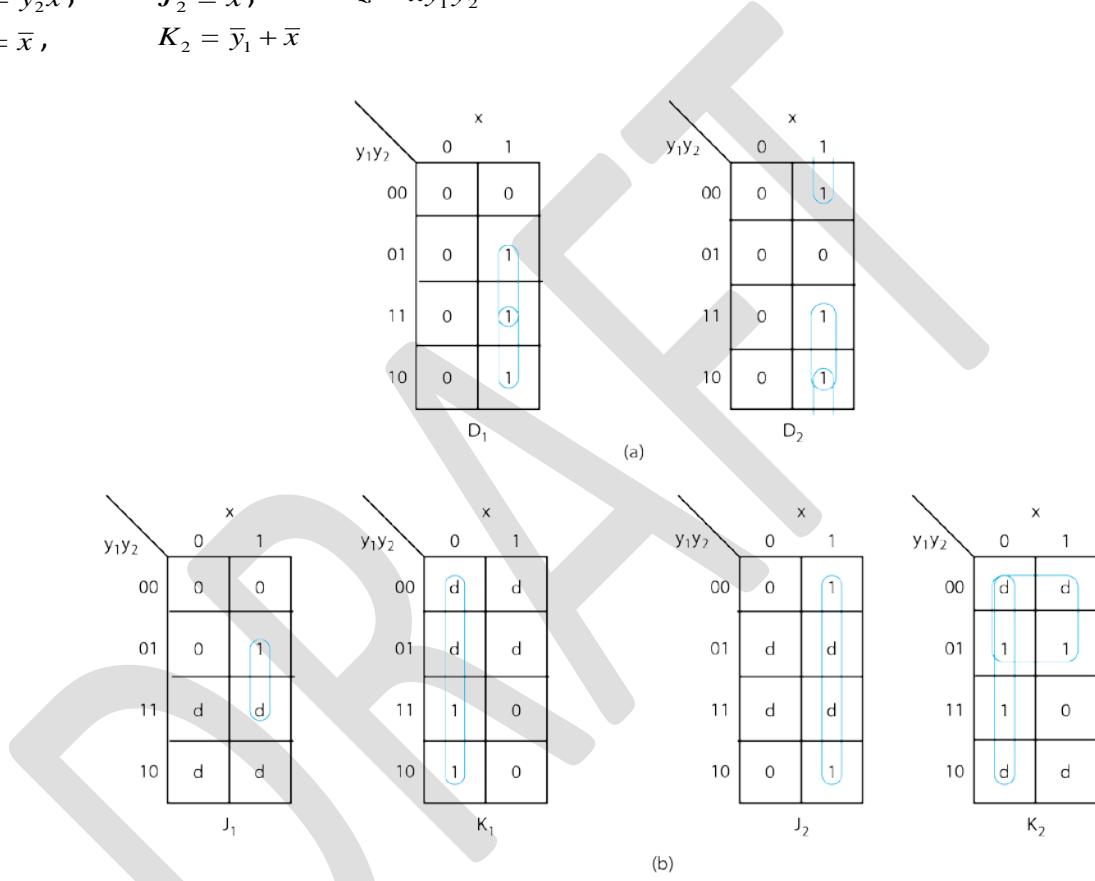


Figure 5.25: Different flip-flop realizations. (a) D flip-flop excitation maps.

(b) JK flip-flop excitation maps.

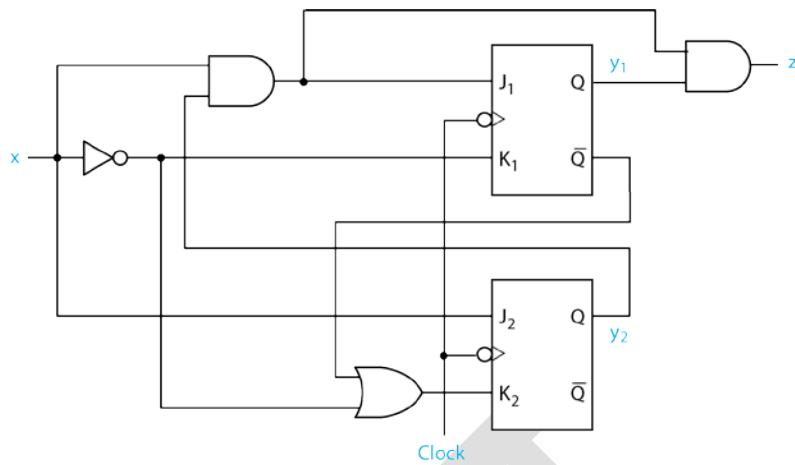


Figure 5.26: Logic diagram of the JK flip-flop realization.

Example 5.9. Here we want to design a clocked sequential circuit that recognizes the input sequence consisting of exactly two zeros followed by a 10. In other words, the following output sequence should result from the given input sequence.

$$\begin{aligned} x &= 001001000010010 \\ z &= 000100100001001 \end{aligned}$$

One way to approach the design of a sequence recognizer is to establish a string of state transitions corresponding to the "correct" input sequence. Examine the partial state diagram of Fig. 5.27a. If the circuit begins in state *A*, an input sequence of 0010 will produce an output $z = 1$ coincident with the last $x = 0$ in the sequence. It is helpful to note the input conditions corresponding to each state.

- A: initial condition
- B: value of x at the last clock transition was 0
- C: values of x at the last two clock transitions were 00
- D: values of x at the last three clock transitions were 001
- E: values of x at the last four clock transitions were 0010

Transition to state *E*, therefore, occurs when the desired input sequence has been received.

Note that we have added an extra state *F* to be used for error conditions; that is, we will send the circuit to this state when the input sequence does not match the correct pattern. Figure 5.27b illustrates some obvious transitions to the error states. To complete the state diagram, we must have two arcs exiting each state, one for each input condition. Since state *C* is entered when the last two values of x have been 0, the arc exiting state *C* loops back to itself to indicate that this is still the case, allowing for overlapping $x = 0$ values at the start of a sequence. Notice that the $x = 0$ arc exiting state *E* has not yet been defined. This arc should loop back to state *C* to allow for overlapping sequences, i.e. the last two values of x have been 0, which may be the start of a new sequence. The correct transition is drawn in Fig. 5.27c. Next we must complete the transitions exiting error state *F*. If an input $x = 0$ occurs while in state *F*, then this can be the beginning of a valid input sequence, so we send the circuit to state *B* in the state diagram, since state *B* indicates that the last x value was 0. If input

$x=1$ occurs while in state F , then the transition loops back to state F until the next $x=0$ occurs, as shown in Fig. 5.27d. This completes the state diagram.

We should now look for equivalent states in our design. Examining the state table for the circuit in Fig. 5.27e, we note that state $A = F$ and state $B = E$, since they have identical rows. Therefore we may eliminate rows E and F , and replace any remaining next state E with B (see row D), and any remaining next state F with A (see rows A and B). The reduced state diagram and state table are shown in Figs. 5.27f and g. More will be said about equivalent states and state table reduction later in this chapter.

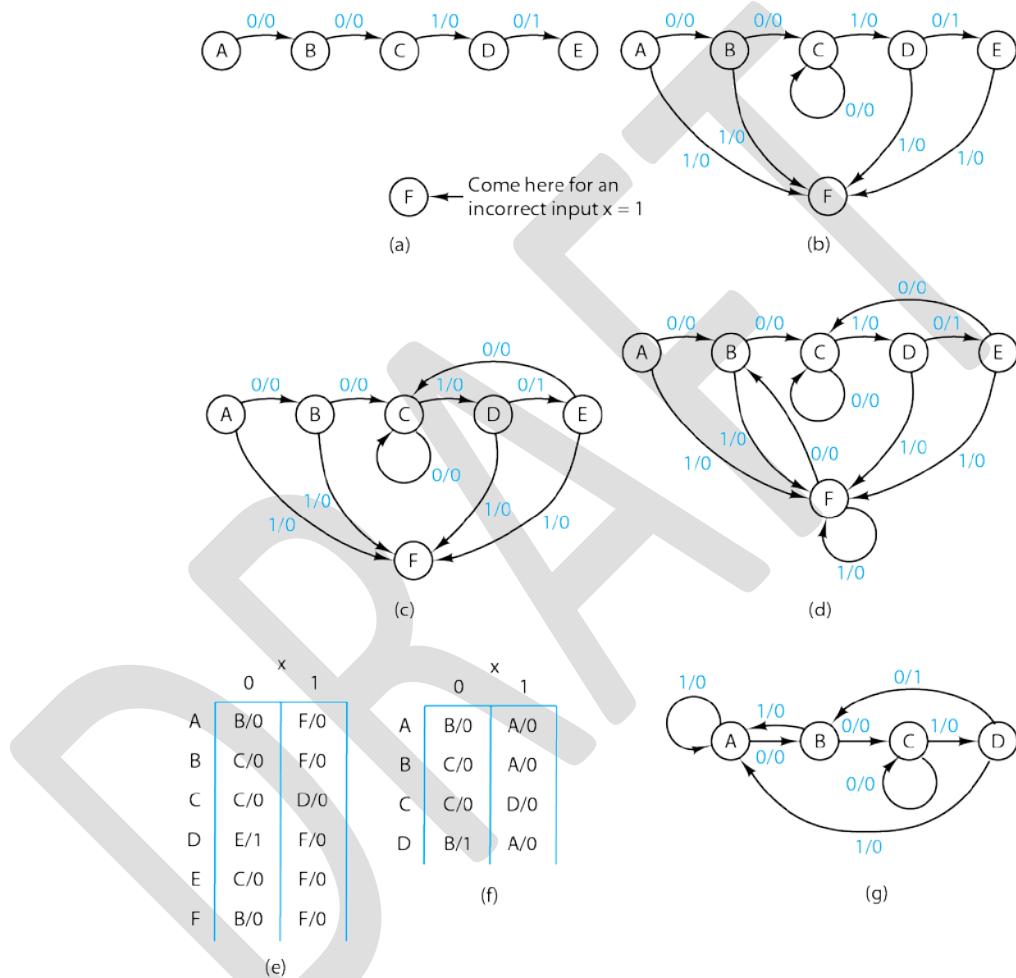


Figure 5.27: A sequence 0010 recognizer. (a) Partial state diagram. (b) State transitions to the error states. (c) Feedback for overlapping sequences. (d) Complete state diagram. (e) State table. (f) Reduced state table. (g) State diagram.

Realization of this circuit in hardware will require two flip-flops; note that if we had not eliminated the redundant states, we would have needed three flip-flops. To complete the solution, we need only follow the procedure as shown in the previous examples.

The state table of Figure 5.27 (f) is captured in the following HDL model. This and previous examples illustrate how *case* statements can be used to easily model state tables by

mapping each row of the state table to a corresponding case in the model. Input conditions are modeled using *if ... else* statements within each row. While next states and outputs can be specified in a single *always* block, it is often simpler to have an *always* block for deriving next states and a separate *always* block or *assign* statement for deriving outputs as shown in the examples.

```
//Behavioral Verilog model of a sequence 0010 recognizer
module Example_5_9 (
    input x, Clock, CLR,           //declare inputs
    output z);                   //declare output
    reg [1:0] state, nextstate;   //declare state and next state variables
    parameter A = 2'b00, B = 2'b01, C = 2'b11, D = 2'b10; //make state assignments
    always @ (negedge Clock, negedge CLR) //detect negative edge of Clock or CLR
        if (CLR == 0) state <= A;           //go to state A if CLR is low
        else state <= nextstate;          //change state on negative edge of Clock
    always @ (state, x)             //detect change of state or input
        case (state)                  //derive next state given present state and x
            A: if (~x) nextstate <= B; else nextstate <= A;
            B: if (~x) nextstate <= C; else nextstate <= A;
            C: if (~x) nextstate <= C; else nextstate <= D;
            D: if (~x) nextstate <= B; else nextstate <= A;
        endcase
        assign z = (state == D) ? ~x : 0; //derive output as specified in state table
    endmodule
```

```
-- Behavioral VHDL Model of a sequence 0010 recognizer
entity Example5_9 is
    port ( x: in bit;      -- input
           clk: in bit;     -- clock
           z: out bit);   -- output
end Example5_9;
architecture behavior of Example5_9 is
    type states is (A,B,C,D); -- four states
    signal y: states := A;    -- initial state A for simulation
begin
    z <= '1' when y = D and x = '0' else '0'; -- Mealy model output

    process(clk) begin      -- State changes
        if clk'event and clk = '0' then -- falling clk edge
            case y is
                when A => if x = '0' then -- current state A
                    y <= B; -- next state B if x=0
                else
                    y <= A; -- next state A if x=1
            endcase
        endprocess
    end
```

```

        end if;
when B => if x = '0' then -- current state B
            y <= C; -- next state C if x=0
        else
            y <= A; -- next state A if x=1
        end if;
when C => if x = '0' then -- current state C
            y <= C; -- next state C if x=0
        else
            y <= D; -- next state D if x=1
        end if;
when D => if x = '0' then -- current state D
            y <= B; -- next state B if x=0
        else
            y <= A; -- next state A if x=1
        end if;
end case;
end if;
end process;
end;

```

A number of problems may be approached in the same manner as for sequence recognizers. This is illustrated by the following example, in which we design an arithmetic circuit to operate on sequences of binary digits.

Example 5.10. Let us design a serial binary adder that computes the sum of two n -bit binary numbers, $a_{n-1} \dots a_1 a_0$ and $b_{n-1} \dots b_1 b_0$, one bit at a time, beginning with the least significant bit.

The serial adder is to be used as illustrated in the block diagram of Fig. 5.28a. The addend and augend are stored in shift registers A and B , respectively. The sum replaces the addend in register A . Bits are presented to the serial adder from the shift registers. In clock cycle i , the adder inputs are addend bit a_i and augend bit b_i , and the output is the sum bit s_i , which is computed as the sum of the two inputs and the carry bit, c_{i-1} , produced while generating sum bit s_{i-1} . The circuit must remember the carry from the previous clock cycle. Therefore, the state of the adder in clock cycle i should reflect the value of c_{i-1} . Consequently, two states are needed for the circuit. State 0 represents the condition $c_{i-1} = 0$ and state 1 represents $c_{i-1} = 1$. At the end of each clock cycle, the sum bit s_i is shifted into register A to replace addend bit a_i , while b_i is returned to register B .

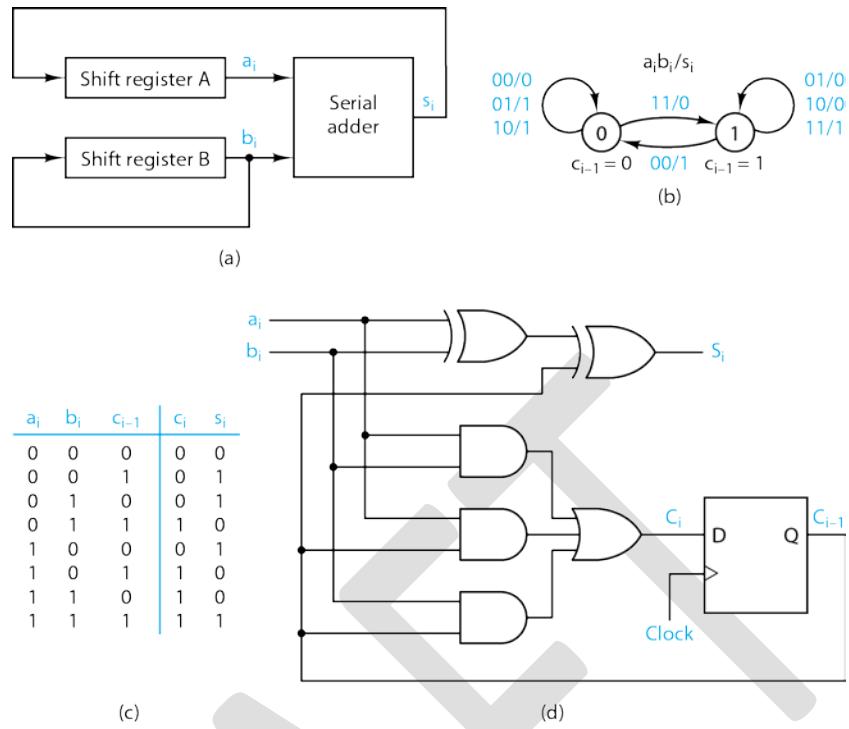


Figure 5.28: Serial binary adder design. (a) Block diagram. (b) State diagram. (c) State table. (d) Logic diagram.

The serial adder state diagram is given in Fig. 5.28b and the corresponding state table in Fig. 5.28c. Using a D flip-flop as the memory element, it can be shown that the excitation and output equations are simply the full-adder equations derived in Chapter 3.

$$\begin{aligned}s_i &= a_i \oplus b_i \oplus c_{i-1} \\D &= c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}\end{aligned}$$

where c_{i-1} represents the present state and c_i the next state of the controller. The completed logic diagram is given in Fig. 5.28d.

Counter Circuits

The design and operation of several different counter modules were presented in Section 4.4.3. However, we used ad hoc procedures in the design of those modules. In the examples that follow, we consider the design of counter circuits using the methods presented in this chapter. In these examples, Moore models are assumed, with the state and output reflecting the current value of the counter.

Example 5.11. Design an up/down counter with four states (0, 1, 2, 3) using JK flip-flops. A control signal x is to be used as follows: When $x=0$ the circuit counts forward (up); when $x=1$, it counts backward (down).

A state diagram depicting this counter is illustrated in Fig. 5.29a. From this diagram the state table shown in Fig. 5.29b is derived. Notice that the output of the counter is just its present state, and thus the counter is a Moore model circuit. If we choose a state assignment

$$[y_1y_2] = 00 = 0$$

$$[y_1y_2] = 01 = 1$$

$$[y_1y_2] = 10 = 2$$

$$[y_1y_2] = 11 = 3$$

which is standard for counters, the transition table may be produced as illustrated in Fig. 5.29c. Note that rows for states 2 and 3 were swapped in the transition table, to facilitate derivation of excitation maps by ordering state variable values for y_1y_2 in Gray code order.

Using the input table for the JK flip-flop (see Fig. 5.17b), the excitation maps for the two flip-flops y_1 and y_2 are obtained in Fig. 5.29d. Using these K-maps, the following relations are found:

$$J_1 = K_1 = x\bar{y}_2 + \bar{x}y_2 = x \oplus y_2$$

$$J_2 = K_2 = 1$$

Hence, the logic diagram for the four-state up/down counter is drawn in Fig. 5.30. If the signal x is controlled by a toggle switch and the clock period is very slow (say 1 second), the action of this device may be observed by attaching light-emitting diodes (LEDs) to the flip-flop outputs.

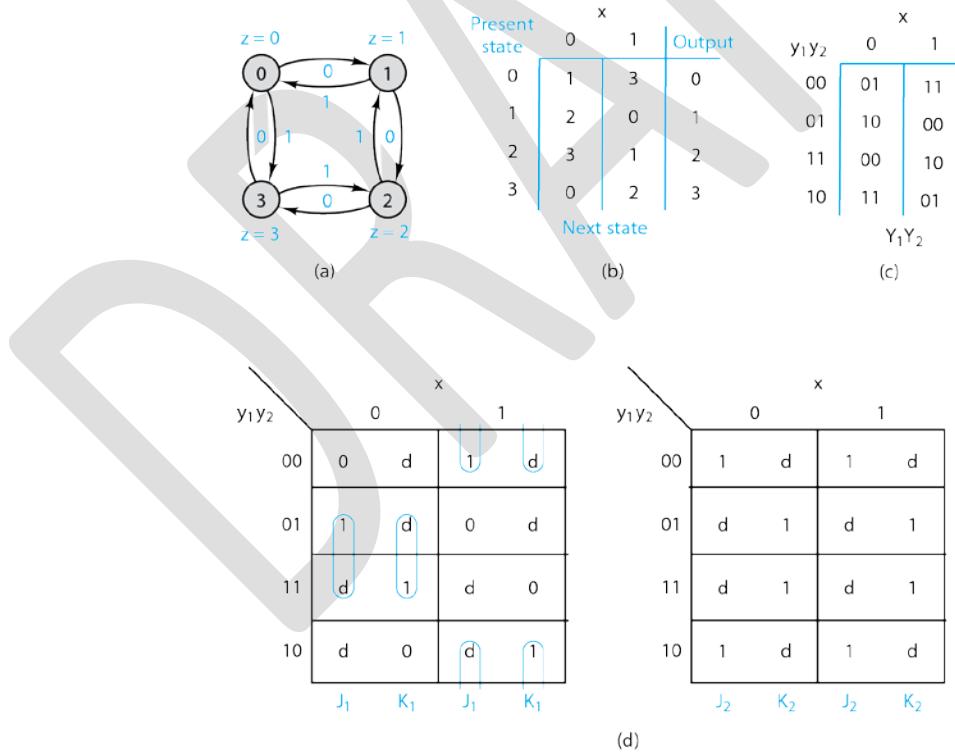


Figure 5.29: Up/down counter synthesis. (a) State diagram. (b) State table.
(c) Transition table. (d) Excitation maps.

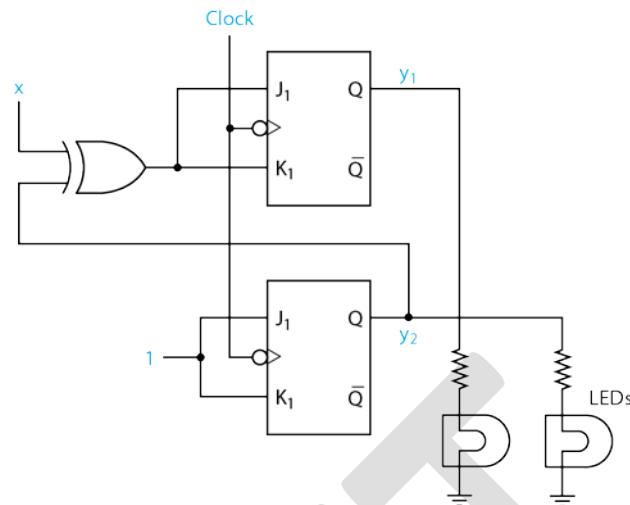


Figure 5.30: Implementation of an up/down counter.

Note that the previous design produced a Moore machine which, as you recall, means the output variables are functions of state variables and not of input variables. In this case the output variables and the state variables are one and the same.

A behavioral HDL model of the state table from Figure 5.29b is given below. For Moore machines, it is straight forward to use a single *always* block to derive both next states and outputs.

```
// Verilog behavioral model of a four-state up/down counter. Example 5.11
module Example_5_11 (
    input x, Clock, CLR,                               //declare inputs
    output reg Y1,Y2);                                //declare outputs
    reg [1:0] state,nextstate;                         //declare state and next state variables
    parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11; //make state assignment
    always @ (negedge Clock, negedge CLR)              //detect negative edge of Clock or CLR
        if (CLR == 0) state <= A;                      //go to state A if CLR is low
        else state <= nextstate;                        //change state on negative edge of Clock
    always @ (state, x)                                //detect change in state or input
        case (state)                                    //derive output and next state
            A: begin {Y1,Y2} <= A; if (~x) nextstate <= B; else nextstate <= D; end
            B: begin {Y1,Y2} <= B; if (~x) nextstate <= C; else nextstate <= A; end
            C: begin {Y1,Y2} <= C; if (~x) nextstate <= D; else nextstate <= B; end
            D: begin {Y1,Y2} <= D; if (~x) nextstate <= A; else nextstate <= C; end
        endcase
endmodule
```

```
-- Behavioral VHDL Model of a four-state up/down counter. Example 5.11
entity Example5_11 is
    port ( x:  in bit;      -- input
           clk: in bit;      -- clock
           clear: in bit;    -- asynchronous clear
           z:   out bit_vector(1 downto 0)); -- two-bit output
end Example5_11;
architecture behavior of Example5_11 is
    signal y: bit_vector(1 downto 0) := "00"; -- initial state 0 for simulation
begin
    z <= y; -- Moore model counter output = state

    process(clk, clear) begin      -- State changes
        if clear = '0' then
            state <= "00";
        elsif clk'event and clk = '0' then -- falling clk edge
            case y is
                when "00" => if x = '0' then -- current state 0
                    y <= "01"; -- count up if x=0
                else
                    y <= "11"; --count down if x=1
                end if;
                when "01" => if x = '0' then -- current state 1
                    y <= "10"; -- count up if x=0
                else
                    y <= "00"; --count down if x=1
                end if;
                when "10" => if x = '0' then -- current state 2
                    y <= "11"; -- count up if x=0
                else
                    y <= "01"; --count down if x=1
                end if;
                when "11" => if x = '0' then -- current state 0
                    y <= "00"; -- count up if x=0
                else
                    y <= "10"; --count down if x=1
                end if;
            end case;
        end if;
    end process;
end;
```

Now let us tackle a more formidable counter design.

Example 5.12. Use JK flip-flops to design a circuit that counts in the BCD code. The counter has one control signal, x . When $x=1$, the counter counts; otherwise, it holds the current state. The output showing the value of the count is to be in the form of four lights. For example, if the count is 3, then the lights would read OFF, OFF, ON, ON.

Because of its simplicity, a state diagram is unnecessary since the state table can be constructed directly. Ignoring the output for the moment, the state table for this example is shown in Fig. 5.31a.

To satisfy the output readout, the states will be assigned so that they are a direct indication of the count; that is, each is assigned its BCD representation, and hence the output can be obtained by merely monitoring the outputs of the flip-flops and using these signals to turn the lights on and off. Therefore, the state assignment is

$$\begin{array}{ll}
 [y_3y_2y_1y_0] = 0000 = 0 & [y_3y_2y_1y_0] = 0101 = 5 \\
 [y_3y_2y_1y_0] = 0001 = 1 & [y_3y_2y_1y_0] = 0110 = 6 \\
 [y_3y_2y_1y_0] = 0010 = 2 & [y_3y_2y_1y_0] = 0111 = 7 \\
 [y_3y_2y_1y_0] = 0011 = 3 & [y_3y_2y_1y_0] = 1000 = 8 \\
 [y_3y_2y_1y_0] = 0100 = 4 & [y_3y_2y_1y_0] = 1001 = 9
 \end{array}$$

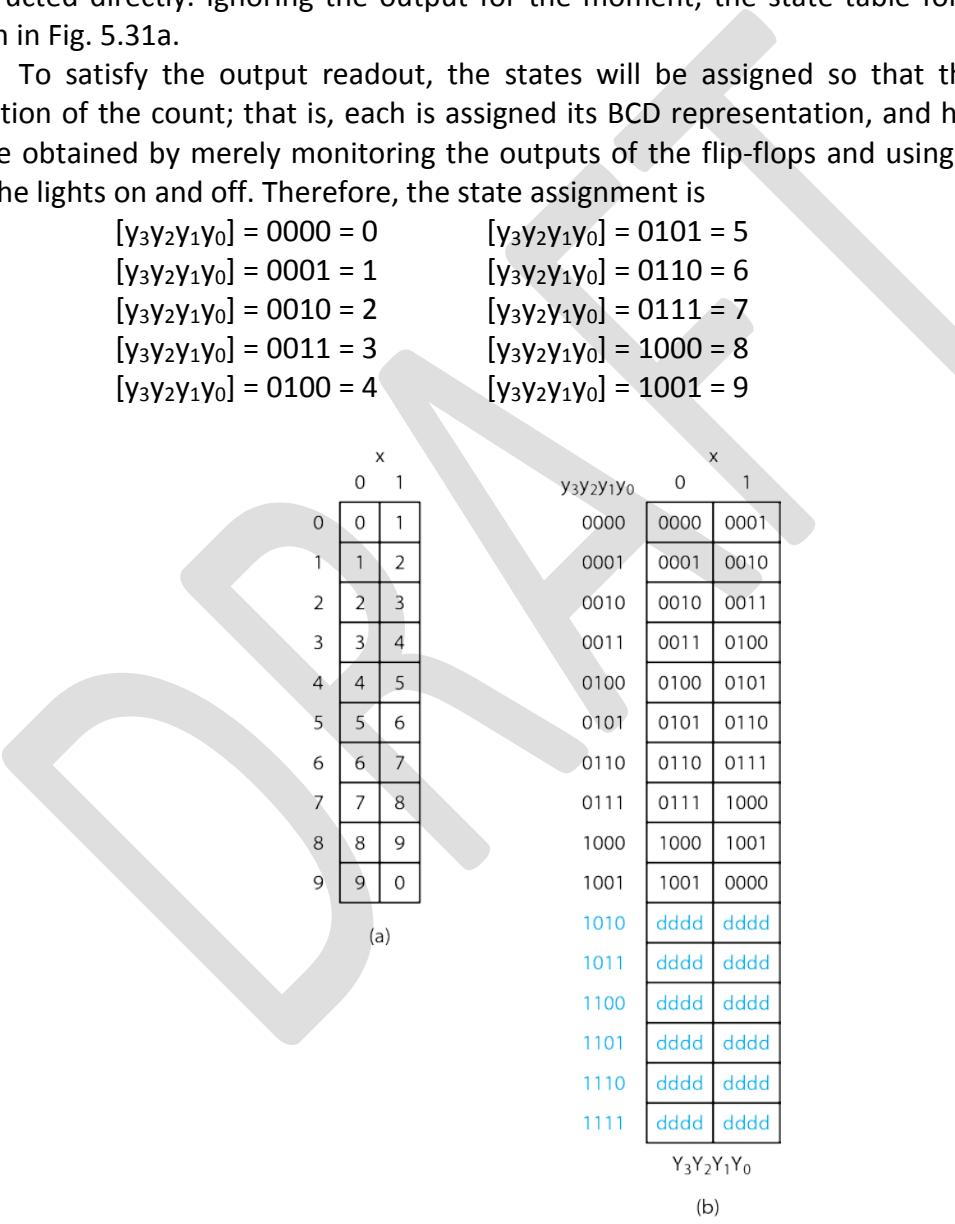


Figure 5.31 consists of two tables, (a) and (b), illustrating the design of a BCD counter. Table (a) is a state table showing the next state for each combination of current state and control signal x . Table (b) is a transition table showing the next state for each combination of current state and control signal x , along with the corresponding output values for four lights (y_3, y_2, y_1, y_0).

	x	
	0	1
0	0 1	
1	1 2	
2	2 3	
3	3 4	
4	4 5	
5	5 6	
6	6 7	
7	7 8	
8	8 9	
9	9 0	

(a)

	x	
	0	1
$y_3y_2y_1y_0$	0000	0001
0000	0000	0001
0001	0001	0010
0010	0010	0011
0011	0011	0100
0100	0100	0101
0101	0101	0110
0110	0110	0111
0111	0111	1000
1000	1000	1001
1001	1001	0000
1010	dddd	dddd
1011	dddd	dddd
1100	dddd	dddd
1101	dddd	dddd
1110	dddd	dddd
1111	dddd	dddd

(b)

Figure 5.31: Design of the BCD counter. (a) State table. (b) Transition table.

$y_3y_2y_1y_0$	x	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0000	0	0	d	0	d	0	d	0	d
0001	0	0	d	d	0	d	d	0	d
0010	0	0	d	d	0	d	d	0	d
0011	0	0	d	d	0	1	d	d	d
0100	0	0	d	d	d	d	0	0	0
0101	0	0	d	d	d	d	0	1	0
0110	0	0	d	d	d	d	0	1	1
0111	0	1	d	d	d	d	0	1	d
1000	d	d	0	0	0	d	d	0	d
1001	d	d	0	1	0	d	d	0	1
1010	d	d	d	d	d	d	d	d	d
1011	d	d	d	d	d	d	d	d	d
1100	d	d	d	d	d	d	d	d	d
1101	d	d	d	d	d	d	d	d	d
1110	d	d	d	d	d	d	d	d	d
1111	d	d	d	d	d	d	d	d	d

(c)

Figure 5.31: (Continued) design of the BCD counter. (c) Excitation tables for the BCD counter.

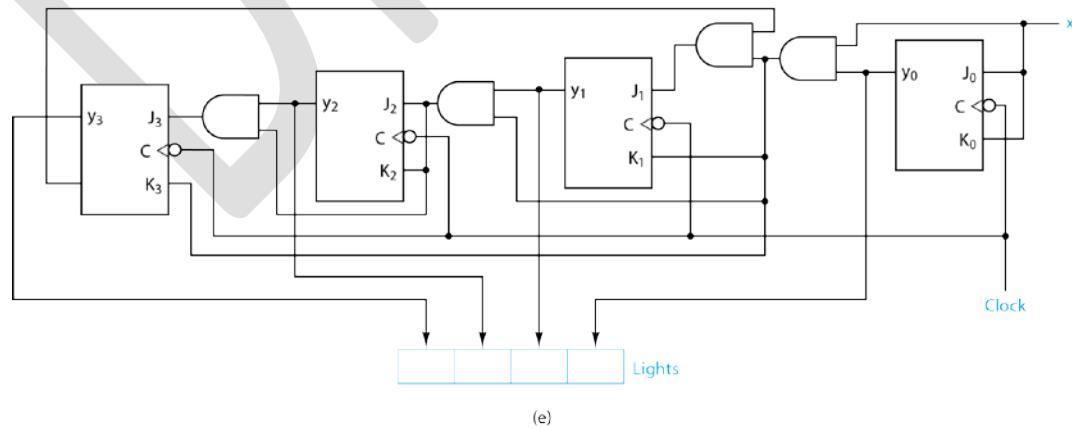
y_3y_2	00	01	11	10	
y_1y_0	00	0	d	d	0
	01	0	d	d	0
	11	0	d	d	d
	10	0	d	d	d

$x = 0$

00	01	11	10
0	d	d	0
0	d	d	0
1	d	d	d
0	d	d	d

$x = 1$

(d)



(e)

Figure 5.31: (Continued) design of the BCD counter. (d) K-map for J_2 . (e) Realization of the BCD counter.

The transition table for this assignment is given in Fig. 5.31b. Note that there are no actual states corresponding to $[y_3y_2y_1y_0] = \{1010 \dots 1111\}$, since the operation of the circuit is limited to the ten states specified by the state table and state assignment. Note that it does not matter what the next states *might be* if $[y_3y_2y_1y_0] = \{1010\dots 1111\}$, since the circuit cannot enter states corresponding to these state variable values. Therefore, the next state entries in these rows of the transition table of Fig. 5.33b are entered as don't care values.

Next we derive the excitation maps for the four JK flip-flops using Fig. 5.17b. The resulting tables are presented in Fig. 5.31c. The d 's in the tables represent don't-cares; since states 10-15 will not occur, their next states can be left unspecified. These tables may be reorganized into K-map form as shown for input J_2 in Fig. 5.31d. All the resulting Boolean logic equations are as follows:

$$\begin{aligned} J_3 &= y_2y_1y_0x, & K_3 &= y_0x \\ J_2 &= y_1y_0x, & K_2 &= y_1y_0x \\ J_1 &= \bar{y}_3y_0x, & K_1 &= y_0x \\ J_0 &= x, & K_0 &= x \end{aligned}$$

Note from the logic equations that the input x acts as a gating signal to disable or enable all flip-flop inputs simultaneously, since $x = 0$ results in all J and K inputs being 0, causing the flip-flops to hold their current states. The actual implementation of the circuit using JK flip-flops is shown in Fig. 5.31e.

The following behavioral model of the BCD counter is similar to that of the counter in Example 5.11, except that four state variables are needed instead of two. Four state variables define sixteen states even though only ten are needed. The six extra states can be omitted from the HDL model or included as shown below. The latter is preferred since undefined cases can be problematic when synthesizing HDL models.

```
//BCD or decade counter. Verilog behavioral model. Example 5.12
module BCDcounter (
    input x, Clock, CLR,                      //declare inputs
    output reg y3,y2,y1,y0,                     //declare outputs
    output reg [3:0] state, nextstate);          //declare states and next states
//make state assignments
parameter
S0=4'b0,S1=4'b1,S2=4'b10,S3=4'b11,S4=4'b100,S5=4'b101,S6=4'b110,S7=4'b111,S8=4'b1000,S9
=4'b1001,S10=4'b1010,S11=4'b1011,S12=4'b1100,S13=4'b1101,S14=4'b1110,S15=4'b1111;
//Count to 10 using a Moore machine model
always @ (negedge Clock, negedge CLR)           //detect negative edge of Clock or CLR
    if (CLR==0) state <= S0;                   //go to state S0 if CLR is low
    else state <= nextstate;                  //make state change on negative edge of Clock
always @ (x,state)                            //detect change of state or input
    case (state)                             //derive next state and output
        S0: begin if (x) nextstate <= S1; else nextstate <= S0; {y3,y2,y1,y0} <= S0; end
        S1: begin if (x) nextstate <= S2; else nextstate <= S1; {y3,y2,y1,y0} <= S1; end
        S2: begin if (x) nextstate <= S3; else nextstate <= S2; {y3,y2,y1,y0} <= S2; end
```

```

S3: begin if (x) nextstate <= S4; else nextstate <= S3; {y3,y2,y1,y0} <= S3; end
S4: begin if (x) nextstate <= S5; else nextstate <= S4; {y3,y2,y1,y0} <= S4; end
S5: begin if (x) nextstate <= S6; else nextstate <= S5; {y3,y2,y1,y0} <= S5; end
S6: begin if (x) nextstate <= S7; else nextstate <= S6; {y3,y2,y1,y0} <= S6; end
S7: begin if (x) nextstate <= S8; else nextstate <= S7; {y3,y2,y1,y0} <= S7; end
S8: begin if (x) nextstate <= S9; else nextstate <= S8; {y3,y2,y1,y0} <= S8; end
S9: begin if (x) nextstate <= S0; else nextstate <= S9; {y3,y2,y1,y0} <= S9; end
//account for unused states
S10: nextstate <= S10;
S11: nextstate <= S11;
S12: nextstate <= S12;
S13: nextstate <= S13;
S14: nextstate <= S14;
S15: nextstate <= S15;
endcase
endmodule

```

-- BCD or decade counter. VHDL behavioral model. Example 5.12

```

entity BCDcounter is
port ( x: in bit;      -- input
       clk: in bit;      -- clock
       clear: in bit;    -- asynchronous clear
       z: out bit_vector(3 downto 0)); -- 4-bit output
end BCDcounter ;
architecture behavior of BCDcounter is
  signal y: bit_vector(3 downto 0) := "0000";      -- initial state 0 for simulation
begin
  z <= y;  -- Moore model counter output = state

  process(clk, clear) begin      -- State changes
    if clear = '0' then
      state <= "0000";           -- reset the counter
    elsif clk'event and clk = '0' then  -- falling clk edge
      case y is
        when "0000" => if x = '1' then y <= "0001"; end if; -- no change for x = '0'
        when "0001" => if x = '1' then y <= "0010"; end if; -- no change for x = '0'
        when "0010" => if x = '1' then y <= "0011"; end if; -- no change for x = '0'
        when "0011" => if x = '1' then y <= "0100"; end if; -- no change for x = '0'
        when "0100" => if x = '1' then y <= "0101"; end if; -- no change for x = '0'
        when "0101" => if x = '1' then y <= "0110"; end if; -- no change for x = '0'
        when "0110" => if x = '1' then y <= "0111"; end if; -- no change for x = '0'
        when "0111" => if x = '1' then y <= "1000"; end if; -- no change for x = '0'
        when "1000" => if x = '1' then y <= "1001"; end if; -- no change for x = '0'
        when "1001" => if x = '1' then y <= "0000"; end if; -- no change for x = '0'
      end case
    end if
  end process
end architecture;

```

```

end case;
end if;
end process;
end;

```

Finite-state Controllers

Many applications require control circuits that perform designated sequences of actions in response to externally applied signals or as a result of conditions produced within the circuit, such as a carry generated in an addition operation or a counter reaching a terminal value. Such control circuits are characterized by having a finite number of states; hence they are referred to as *finite-state controllers*.

One of the most common applications of finite-state controllers is as control units for computers and other digital systems. Such systems generally comprise two parts: a data path and a control unit. The data path performs various operations on data elements, such as arithmetic operations and other transformations. Data paths typically comprise combinational logic modules, such as arithmetic logic units and multiplexers, and may include registers for data storage.

The control unit supplies commands to the data path to enable selected operations. These commands must be sequenced properly to ensure that the proper operations are carried out in response to various inputs and conditions. We design a control unit for a digital system by identifying its inputs and outputs and then developing the control algorithm in the form of a state diagram.

Finite-state controllers are designed using the general synchronous sequential circuit synthesis procedure defined earlier. We begin by defining the inputs and outputs of the controller and then the algorithm to be implemented. The algorithm is often specified in state diagram format. Then the remaining steps of the synthesis procedure are performed. This is illustrated by the following examples.

Example 5.13. We wish to design a finite-state controller for the simple robot of Fig. 5.32 so that it can find its way out of the maze shown in the figure.

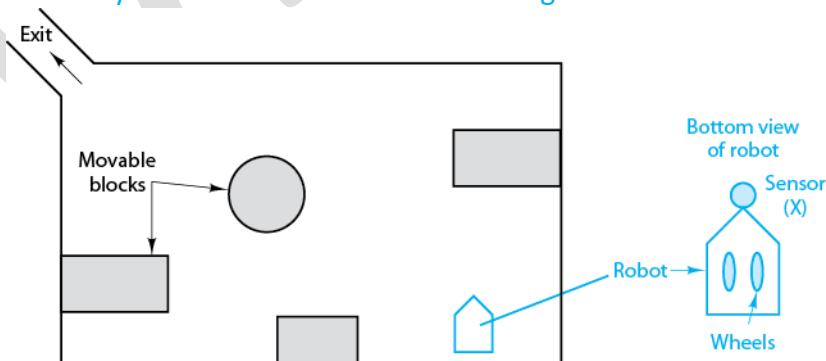


Figure 5.32: Robot and maze.

The robot is to maneuver by turning whenever it comes in contact with an obstacle. On the nose of the robot is a sensor whose output $x = 1$ whenever it is in contact with an obstacle; $x = 0$ otherwise. The robot has two control lines: $z_1 = 1$, which turns the robot to the left, and $z_2 = 1$ which turns the robot to the right. When it encounters an obstacle, the robot should turn right until no obstacle is detected. The next time an obstacle is detected, the robot should turn left until the obstacle is cleared, and so on.

The robot controller requires four states as follows:

State A = no obstacle detected, last turn was left

State B = obstacle detected, turning right

State C = no obstacle detected, last turn was right

State D = obstacle detected, turning left

The control unit state diagram is given in Fig. 5.33a. Note that the controller stays in state *A* with no turns until an obstacle is encountered. Then it enters state *B* and turns right until it no longer detects the obstacle, at which time it enters state *C* and stops turning. The controller then remains in state *C* until another obstacle is detected, at which time it enters state *D* and turns left until the obstacle is no longer detected; it then returns to state *A*.

The state table is given in Fig. 5.33b. Let us select the state assignment $A = 00$, $B = 01$, $C = 11$, and $D = 10$. The binary transition table is given in Fig. 5.33c. From the transition table, we can form the output tables for z_1 and z_2 , shown in Fig. 5.33d, from which we can derive the following output equations.

$$z_1 = xy_1$$

$$z_2 = x\bar{y}_1$$

For the state variables, let us use D flip-flops. The excitation tables for D_1 and D_2 are given in Fig. 5.33e. From these tables, we derive the excitation equations as follows.

$$D_1 = xy_1 + \bar{x}y_2$$

$$D_2 = x\bar{y}_1 + \bar{x}y_2$$

The completed logic circuit diagram for the robot controller is given in Fig. 5.33f.

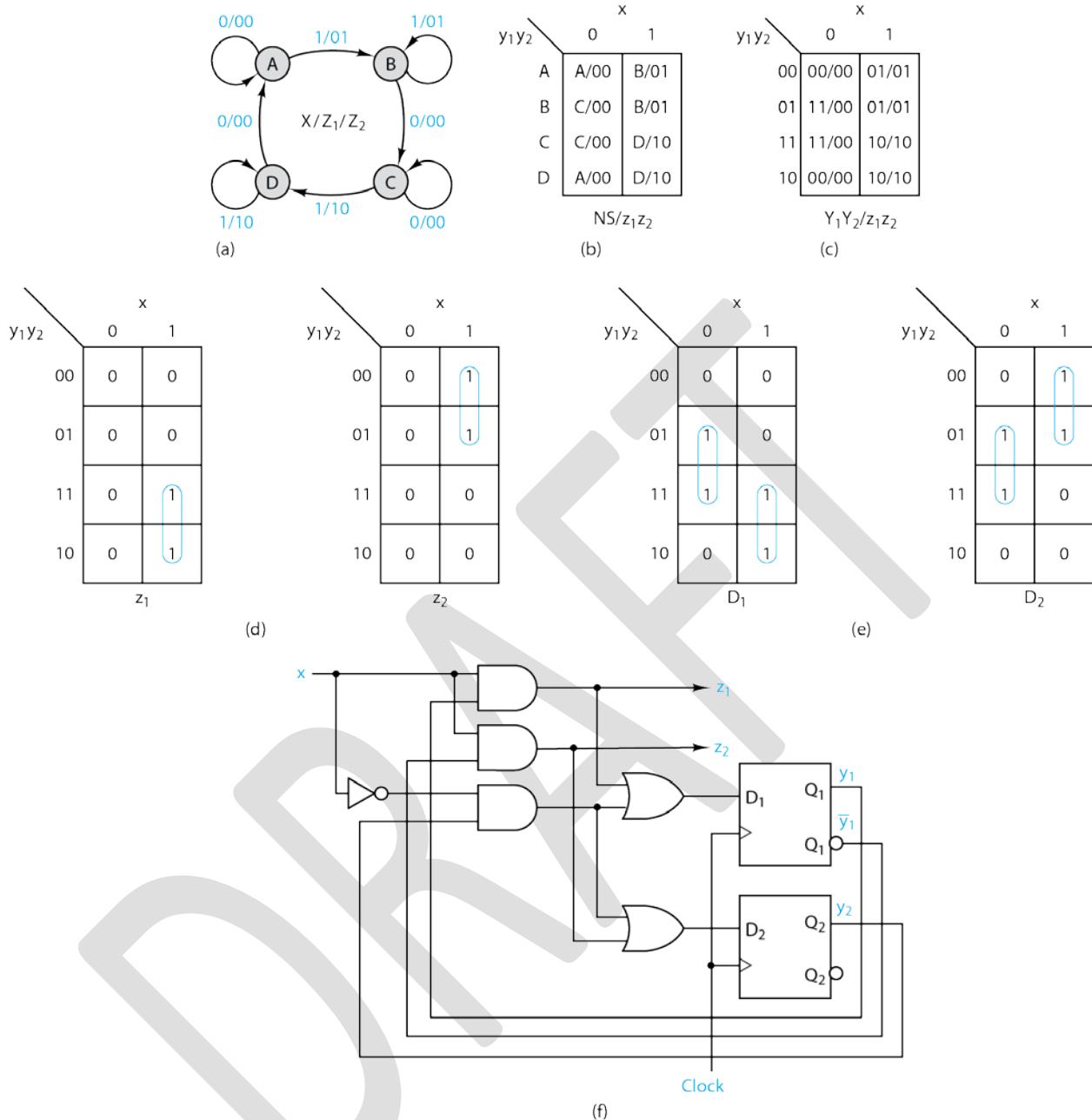


Figure 5.33: Robot controller design. (a) State diagram. (b) State table. (c) Transition table. (d) Output maps. (e) Excitation maps. (f) Logic circuit diagram.

To allow the design to be verified via simulation, the following HDL model is derived from the state table in Figure 5.33b. Note that a single *always* block is used to derive both next state and output values even though this is a Mealy machine. Simulation results for this behavioral model can be compared to those for an HDL structural model, or other logic simulation of the circuit in Fig. 5.33f, to ensure that the synthesized circuit produces the intended behavior.

```
//Robot Controller, Example 5.13, Verilog Behavioral Model
module RobotController (
    input x, clock, clear,           //declare inputs
    output reg z1, z2);             //declare outputs
    reg [1:0] state, nextstate;     //declare state and next state
    parameter A = 2'b00, B = 2'b01, C = 2'b11, D = 2'b10; //make a state assignment
    //detect positive edge of Clock or negative edge of clear
    always @ (posedge clock, negedge clear)
        if (clear==0) state <= A;           //go to state A if clear is low
        else state <= nextstate;          //change states on positive edge of clock
    always @ (state, x)               //detect change of state or x
        case (state)                   //derive next states and outputs
            A: if (~x) begin nextstate <= A; z1 <= 0; z2 <= 0; end
                else begin nextstate <= B; z1 <= 0; z2 <= 1; end
            B: if (~x) begin nextstate <= C; z1 <= 0; z2 <= 0; end
                else begin nextstate <= B; z1 <= 0; z2 <= 1; end
            C: if (~x) begin nextstate <= C; z1 <= 0; z2 <= 0; end
                else begin nextstate <= D; z1 <= 1; z2 <= 0; end
            D: if (~x) begin nextstate <= A; z1 <= 0; z2 <= 0; end
                else begin nextstate <= D; z1 <= 1; z2 <= 0; end
        endcase
    endmodule
```

```
-- Robot Controller, Example 5.13, VHDL Behavioral Model
entity RobotController is
    port ( x:  in bit;      -- input
            clk: in bit;       -- clock
            clear: in bit;    -- clear
            z1,z2: out bit); -- outputs
end RobotController;
architecture behavior of RobotController is
    type states is (A,B,C,D); -- four states
    signal y: states := A;    -- initial state A for simulation
begin
    z1 <= '1' when (y = C or y = D) and x = '1' else '0'; -- Mealy model outputs
    z2 <= '1' when (y = A or y = B) and x = '1' else '0';

    process(clk, clear) begin      -- State changes
        if clear = '0' then
            y <= A;                  -- reset the state
        elsif clk'event and clk = '0' then -- falling clk edge
            case y is
                when A => if x = '0' then -- current state A
                    y <= A; -- next state A if x=0
```

```

        else
            y <= B; -- next state B if x=1
        end if;
    when B => if x = '0' then -- current state B
        y <= C; -- next state C if x=0
    else
        y <= B; -- next state B if x=1
    end if;
    when C => if x = '0' then -- current state C
        y <= C; -- next state C if x=0
    else
        y <= D; -- next state D if x=1
    end if;
    when D => if x = '0' then -- current state D
        y <= A; -- next state A if x=0
    else
        y <= D; -- next state D if x=1
    end if;
end case;
end if;
end process;
end;

```

ASM Diagrams

Section 4.2.4 introduced the use of ASM diagrams to represent the behavior of sequential circuits. The use of ASM diagrams in sequential circuit design is demonstrated by the following two examples.

Example 5.14. Let us design an ASM diagram for a serial two's completer that will implement Algorithm 1.4 presented in Chapter 1.

Algorithm 1.4 requires that we examine the bits of the number from right to left, copying the bits until the first 1 bit has been copied and then complementing the remaining bits. The ASM diagram of the serial two's complement algorithm is given in Fig. 5.34. Data bits are supplied to the circuit sequentially on input x , beginning with the least significant bit. The circuit output z is the corrected data bit. In state A , we are looking for the first 1 bit, and therefore $z = x$. In state B , the first 1 bit has been detected, and therefore $z = \bar{x}$. Note that a Mealy model has been used, since z is a function of x and the state. Therefore, all outputs are specified in conditional output boxes.

While the ASM diagram of Fig. 5.34 defines the sequence of operations on input x , note that a complete circuit would also include a counter that is incremented once for each value of x , and then tested to stop the algorithm after all bits of the data word have been processed. The use of such a counter is illustrated in the next example.

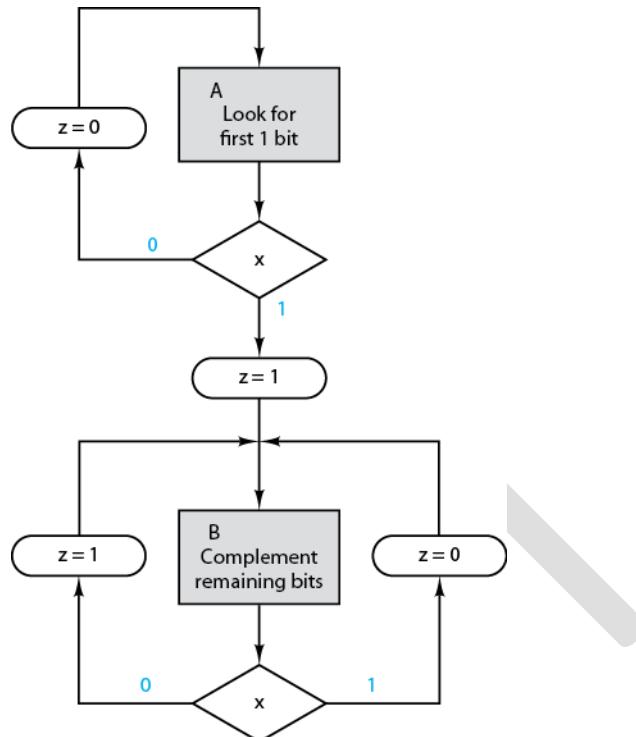


Figure 5.34: ASM representation of a serial two's completer.

The following is an HDL model of the two's completer. Note that the two loops in the ASM model are implemented in nested *if ... else* statements employing a state variable *j* to indicate when the first 1 bit has been detected.

```

//Example 5.14 Serial Two's Complementer Verilog Behavioral Model
module TwosComp (
    input x, clock,
    output reg z);
    //declare inputs
    //declare output
    //declare state variable j, j=0 (state A) means a 1 bit has not been detected
    reg j;
    initial j <= 1'b0;           //initialize j to 0 (state A)
    always @ (posedge clock)    //detect positive edge of clock
    begin                      //begin finding the 2's complement
        if (j==0)              //bits are not complemented if j=0
            begin if (x==0) begin z <= 0; j <= 1'b0; end //stay in state 0, z = x
                else      begin z <= 1; j <= 1'b1; end //go to state 1, z = x
            end
        else                  //bits are complemented if j=1
            if (x==0) z <= 1; else z <= 0;   //stay in state 1, z = ~x
        end
    endmodule

```

5.2.4 One-hot Finite-State Machine Design Method

In the design of sequential circuits described by ASM diagrams, a simplified design approach using the one-hot state assignment method is often used to reduce design time. A *one-hot state assignment* uses one state variable, and consequently one flip-flop, per state, as shown in Table 5.1. All the state variables except one are equal to 0 at any given time. The single state whose state variable is equal to 1 is called the *hot state*. Therefore, an n -state sequential circuit requires n state variables rather than $\lceil \log_2 n \rceil$. The resulting circuit contains more flip-flops, but can be designed more easily and often requires fewer combinational logic gates.

State	Sequential Assignment	One-hot Assignment
	y_1y_0	$y_3y_2y_1y_0$
A	00	0001
B	01	0010
C	10	0100
D	11	1000

Table 5.1: State Assignments for a Four-State Sequential Circuit

A benefit of one-hot designs is that they can be derived directly from ASM diagrams, allowing most of the steps in the synthesis procedure to be omitted. Figure 5.40 illustrates logic circuit implementations of the various ASM constructs. D flip-flops are used for the memory elements, with one flip-flop per state box. Simple state sequencing is implemented by cascading the flip-flops, as illustrated in Fig. 5.35a. When in state A, flip-flop output $Q_A = 1$ and all other flip-flop outputs are 0. Since $D_B = Q_A$, flip-flop output Q_B will be set to 1 on the next clock pulse, while Q_A resets to 0. In this manner, the hot state passes from one flip-flop to the next. For Moore circuits, the circuit outputs are simply the flip-flop outputs, as shown in Fig. 5.35a, since they are functions of only the state.

To initialize the circuit, a reset signal can be used to pulse an asynchronous PRESET input on one flip-flop, and asynchronous CLEAR inputs on the remaining flip-flops. Alternatively all flip-flops can be reset to 0, and then a 1 loaded into the flip-flop corresponding to the initial state. For example, the *Begin* signal in Fig. 5.35a can be used to set $D_A = 1$ for one clock pulse, to set $Q_A = 1$, and then set $D_A = 0$ for the next clock pulse to return Q_A to 0.

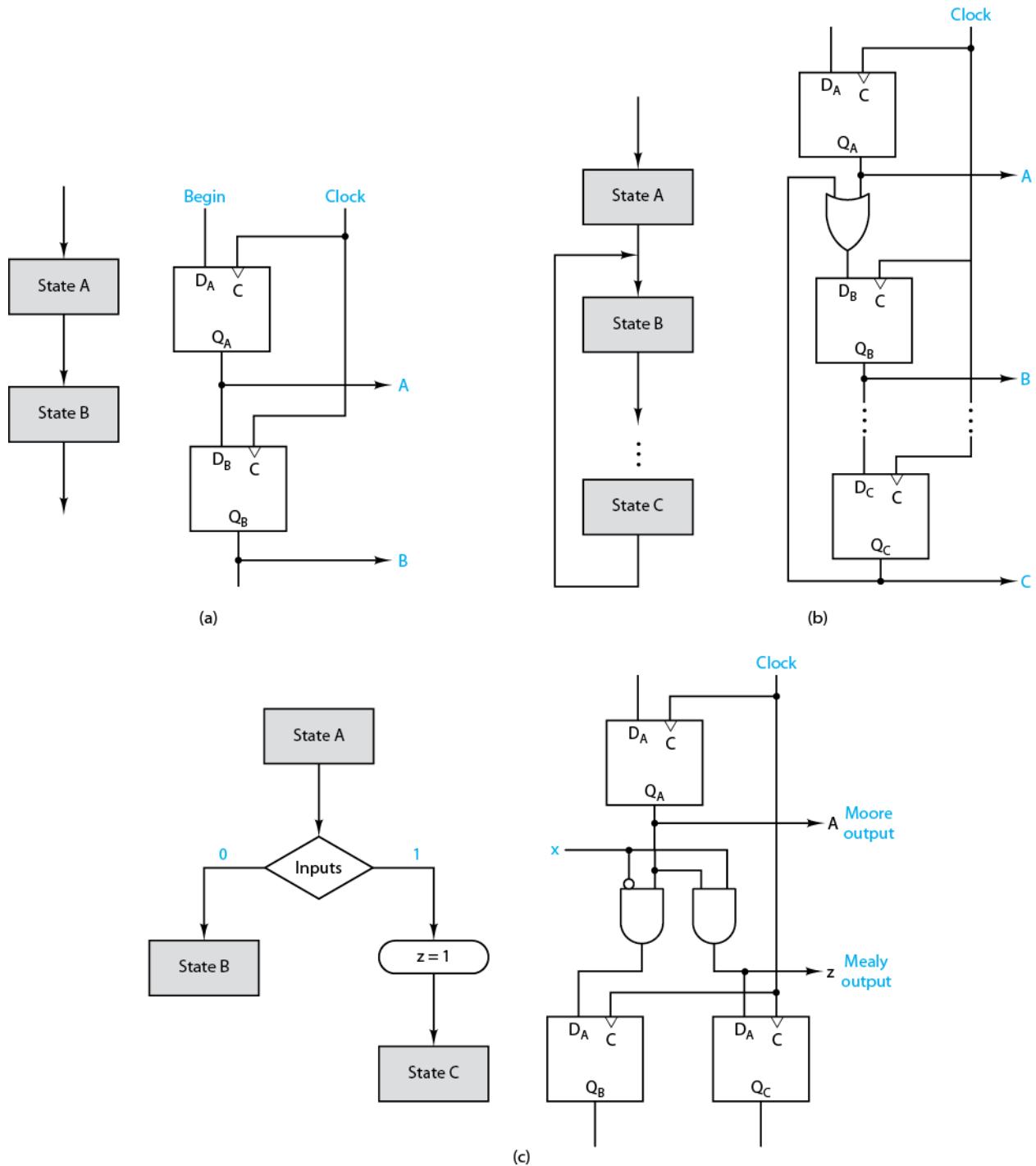


Figure 5.35: Controllers designed with the one-hot method. (a) Simple state sequencing.
(b) Merging control paths. (c) Control decision.

Control paths that merge on the ASM diagram are merged with an OR gate in the control circuit, as illustrated in Fig. 5.35b. In this circuit, flip-flop output Q_B will be set to 1 on the next clock pulse if either $Q_A = 1$ or $Q_C = 1$, to change from state B to state A or C, respectively.

ASM decision boxes are implemented as illustrated in Fig. 5.35c. In this case, when flip-flop output $Q_A = 1$ and input $x = 0$, the AND gates set flip-flop inputs $D_B = 1$ and $D_C = 0$, causing flip-flop output Q_B to be set to 1 on the next clock pulse. If $x = 1$, then $D_B = 0$ and $D_C = 1$.

Conditional output boxes in an ASM diagram are realized by connecting the circuit outputs to the outputs of the AND gates that implement the decision boxes, as shown in Fig. 5.35c, since these signals are functions of both the state and the input.

The following example illustrates the one-hot design method for deriving Moore and Mealy model circuits from ASM diagrams.

Example 5.15. Let us use the one-hot design method to implement the serial two's complements ASM diagram of Fig. 5.34 and the multiplier control unit ASM diagram of Fig. 5.35b.

Figure 5.36a shows the serial two's complements circuit. This circuit structure is derived directly from the ASM diagram of Fig. 5.34, as described above. In this case a Mealy model is used. Therefore, the output z is a function of the state variables and the input x . In this circuit, $z = 1$ if the circuit is in state 0 and $x = 1$ or, if the circuit is in state 1 and $x = 0$.

Figure 5.36b shows the logic diagram of the multiplier control unit. Note that the circuit structure is derived directly from the ASM diagram. Since this is a Moore model, the outputs are simply driven by four flip-flop outputs. The algorithm is initiated by providing a pulse on the *Begin* control line to set the first flip-flop to 1 in one clock cycle and then to 0 on the next clock cycle, assuming that both flip-flops are initially reset to 0.

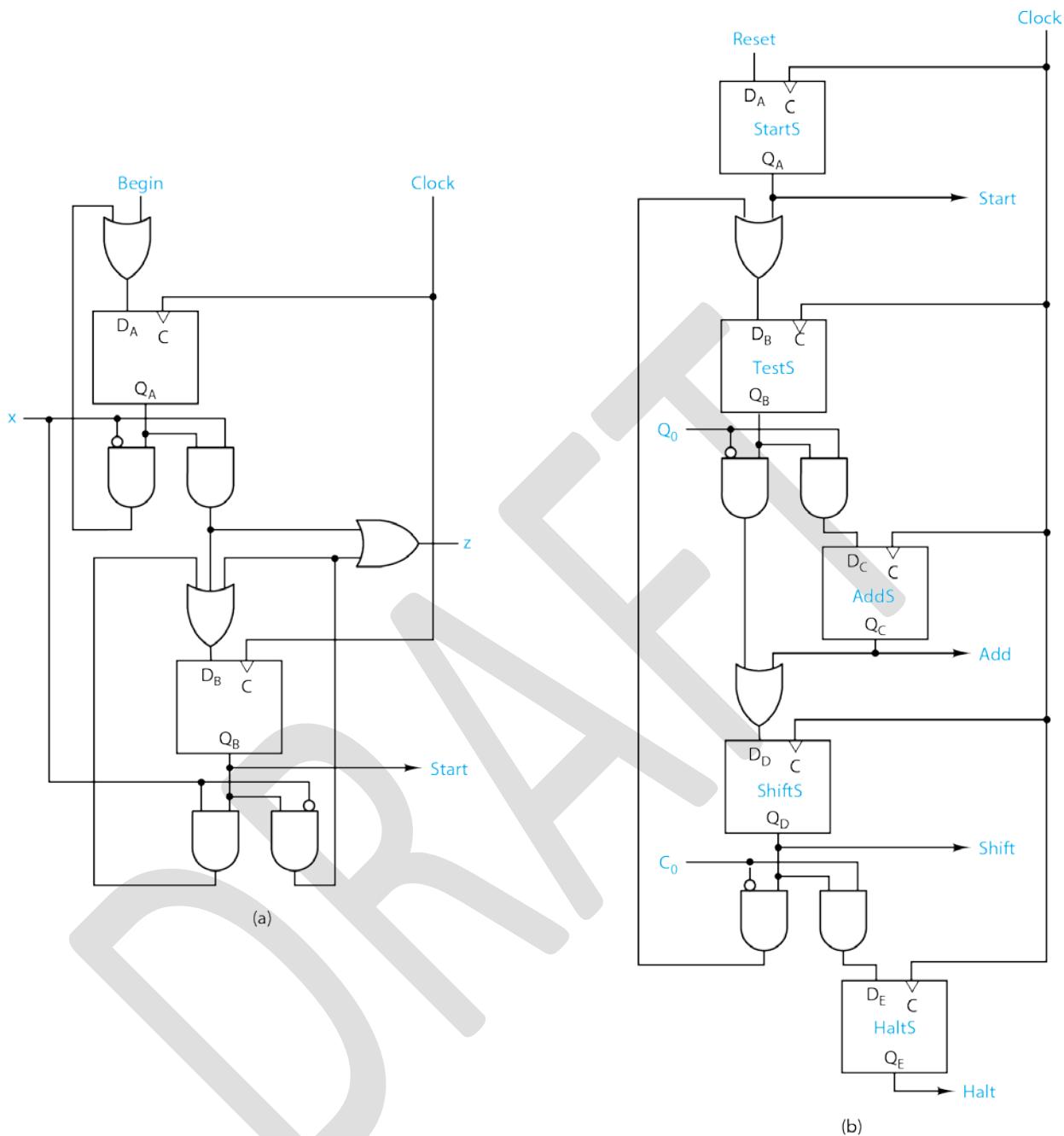


Figure 5.36: Controllers designed with the one-hot method. (a) Serial two's complementer control unit. (b) Multiplier control unit.

The sequential circuit for the drink machine controller of example 5.14 is said to be *incompletely specified*, since its state table contains don't-cares. These don't-cares arise normally in some circuits due to the fact that only a certain set of inputs can ever be applied. Hence, states and outputs that may occur because of forbidden inputs are never attained and we may assign them as don't-cares. The following example will illustrate this idea.

Example 5.16. Let us design a detonator circuit as shown in Fig. 5.37a that exhibits the behavior of the state diagram in Fig. 5.37b.

When the device is active and $x = 0$, the device rests in an idle state A . The detonation sequence is initiated by setting $x = 1$. The device will move from state A to B , then C , and finally D , where it issues a pulse ($z = 1$) to detonate an explosive. The circuitry prior to the detonator circuit is designed so that once the first $x = 1$ occurs the device cannot be reset; that is, no $x = 0$ input will occur once $x = 1$ is received.

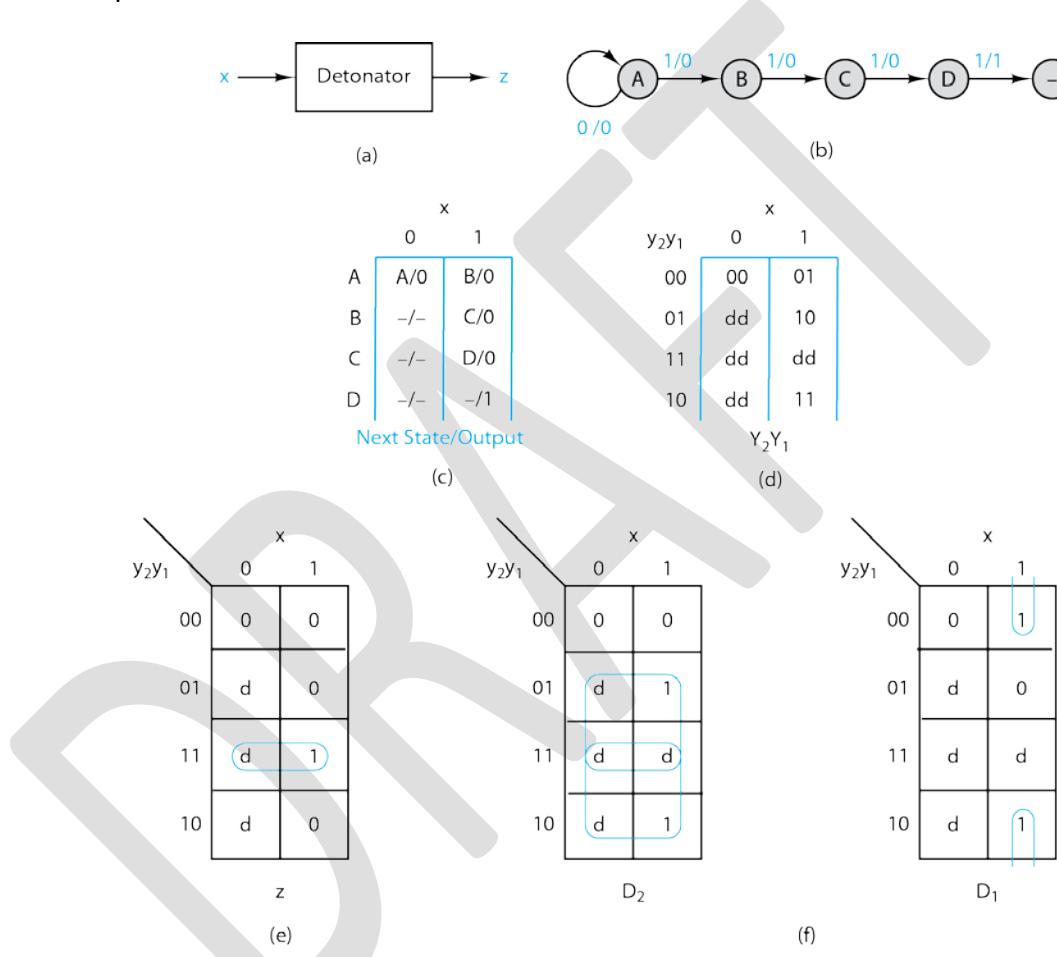


Figure 5.37: Detonator circuit. (a) Block diagram. (b) Partial state diagram. (c) State table. (d) Transition table. (e) Output map. (f) Excitation maps

The partial state diagram and complete state table for the detonator are shown in Figs. 5.37b and c, respectively. Here again note that once the detonator sequence has begun it will continue without interruption until the detonate pulse is generated. The final state is a don't-care because the explosive has ignited. In the following analysis the detonator circuit will be realized using D flip-flops. If we choose the state assignment (y_2y_1) as

$$\begin{aligned}[y_2y_1] = 00 &= A \\ [y_2y_1] = 01 &= B\end{aligned}$$

$$\begin{aligned}[y_2y_1] = 10 &= C \\ [y_2y_1] = 11 &= D\end{aligned}$$

The state transition table for the circuit and the maps for the circuit realization are shown in Figs. 5.42d, e, and f. The following equations follow directly from the maps:

$$\begin{aligned}D_1 &= y_1 + y_2 \\ D_2 &= x_1 \bar{y}_1 \\ z &= y_1 y_2\end{aligned}$$

The actual circuit for the detonator is shown in Fig. 5.38. Since our analysis is valid only during the clock pulse, we use the clock pulse to gate the output.

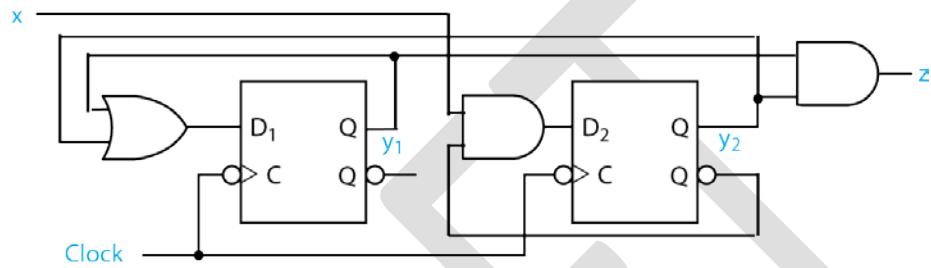


Figure 5.38: Detonator implementation.

It is important for the reader to realize that incompletely specified circuits have an advantage over completely specified circuits from a hardware realization standpoint. The advantage stems from the presence of don't-cares in the state table. In other words, these don't-cares may be able to be grouped with the ones in the excitation maps to produce a simpler circuit than would have been possible if all terms were completely specified. Figure 5.38 illustrates this case.

Once a reduced state table has been determined, the state assignment and circuit logic equations may be produced using exactly the same rules employed in the completely specified case. However, since certain entries in the state table are unspecified, there is usually a larger number of don't-cares in the generated K-maps, resulting in better logic minimization. In addition, different algorithms are used to detect redundant states.

Example 5.17. For the binary state table of Fig. 5.39a, let us complete the realization using D and JK flip-flops.

From the binary state table, the K-maps for each realization are derived in Figs. 5.39b through d, and the corresponding logic equations are listed as follows:

$$D_2 = \bar{x}y_1 + y_2y_1$$

$$D_1 = y_2\bar{y}_1$$

$$J_2 = \bar{x}y_1$$

$$K_2 = \bar{y}_1$$

$$J_1 = y_2$$

$$K_1 = 1$$

$$z = xy_2 + \bar{x}\bar{y}_1$$

Notice that the D flip-flop realization requirea eight inputs to the gates, and the JK, two. In general, the JK flip-flop gives better logic reduction because it has more control logic internal to the device itself.

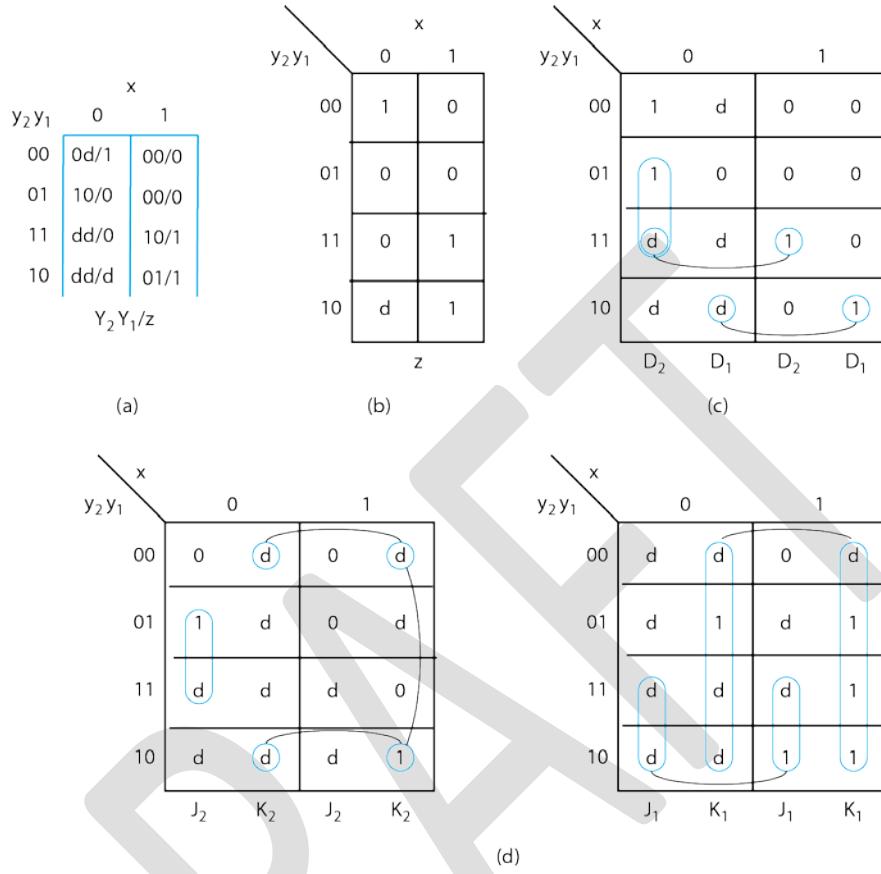


Figure 5.39: Logic realizations. (a) Binary state table. (b) K-map for z .
(c) D flip-flops. (d) JK flip-flops.

5.3 State reduction in sequential circuits

Thus far we have examined methods for designing synchronous sequential circuits. Minimization was limited to applying combinational logic methods to reduce excitation and output equations. Significant savings can also be made in step 3 of the synchronous sequential circuit synthesis procedure. In the design of state diagrams and/or state tables, it is often the case that more states than necessary are included when creating a design. Since the number of states determines the number of memory elements, it is advantageous to reduce this number where possible. In this section we will discuss the identification and elimination of redundant states in both completely and incompletely specified sequential circuits.

5.3.1 Redundant States

In a general sense we say that two states are *equivalent* if we cannot distinguish between them. In other words, we cannot determine in which of two equivalent states a sequential circuit starts by applying sequences of inputs and observing the outputs. If this condition exists for every possible input sequence, one of these states is redundant and can be removed without altering the circuit's behavior.

Redundant states normally arise in an early design phase when a word description of the sequential circuit's function is transformed into a state diagram or state table. The removal of redundant states is important for a number of reasons:

1. *Cost*: The number of memory elements is directly related to the number of states.
2. *Complexity*: The more states the circuit contains, the more complex the design and its associated implementation become.
3. *Aids failure analysis*: Diagnostic routines are often predicated on the assumption that no redundant states exist.

Let us introduce the idea of state equivalence through a simple example. Consider the sequential circuit shown in Figs. 5.40a and b. Suppose that the initial state is unknown. If an input $x=0$ is applied to the circuit and the output is $z=1$, all that is known concerning the initial state is that it is either A or B or C . Likewise, if the output is $z=0$ when input $x=0$ is applied, the initial state is either D or E . Note that a similar conclusion is obtained for the input $x=1$. Therefore, we conclude that states A , B , and C are equivalent and that states D and E are equivalent for an input sequence of length 1, that is, *1-equivalent*. The behavior for input sequences of lengths 2 and 3 is shown in Fig. 5.40c and d. Note that states B and C and states D and E are 2-equivalent, since they produced identical output sequences for any input sequence of length 2. States B and C are also 3-equivalent, and in fact it can be shown that these two states are K -equivalent for all K .

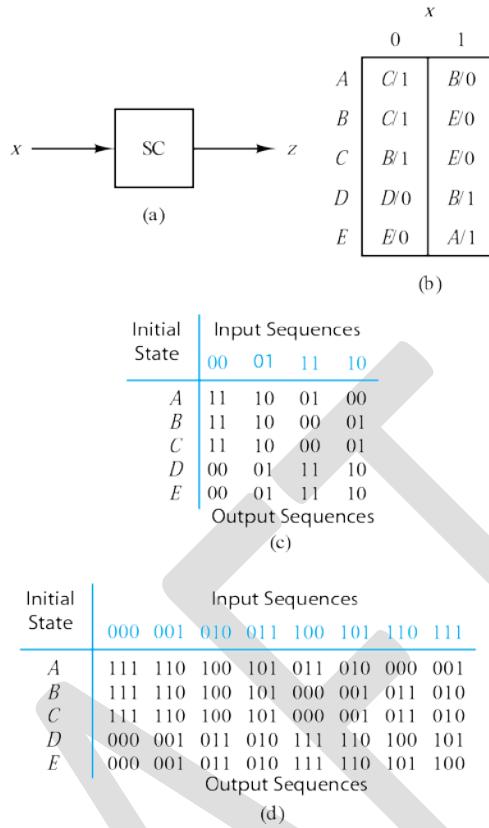


Figure 5.40: Redundant states. (a) Sequential circuit SC . (b) State table. (c) Output Sequences of length 2. (d) Output sequences of length 3.

With these facts as basic background we now define precisely what is meant by equivalent states.

State equivalence and compatibility

The states S_1, S_2, \dots, S_j of a sequential circuit are said to be *equivalent* if and only if, for every possible input sequence, the same output sequence will be produced by the circuit regardless of whether S_1, S_2, \dots, S_j is the initial state.

This definition can be stated in another manner for pairs of states. Let S_k and S_l be the next states of sequential circuit SC when input I_p is applied while the circuit is in states S_i and S_j , respectively. Then S_i and S_j are *equivalent* if and only if, for every possible input I_p ,

1. The output produced by state S_i is equal to the output produced by state S_j .
2. The next states S_k and S_l are equivalent.

The second definition can be deduced from the first as follows. If S_i produces a different output for any input I_p than S_j produces for I_p , then S_i and S_j cannot be equivalent. Hence, the first condition is necessary. If S_k and S_l are not equivalent, there is an input sequence $I_1 I_2 \dots I_k$ that produces a different output sequence for S_i as a starting state than

for S_j as a starting state. Therefore, $I_p I_1 I_2 \dots I_k$ will produce a different output sequence for S_i as a starting state than for S_j as a starting state. Hence, S_i and S_j cannot be equivalent unless the second condition is satisfied. Finally, the conditions are clearly sufficient for S_i and S_j to be equivalent; and therefore, the two definitions are synonymous. These two conditions form the basis for all state reduction techniques.

Let x and y be elements of a set S . Suppose x and y are related by a property r , which is denoted $x \ r \ y$. A *relation* R on the set S is the set of all ordered pairs (s_i, s_j) such that s_i and s_j are elements of S and such that $s_i \ r \ s_j$. R is *reflexive* if and only if $s_i \ r \ s_i$ for all s_i in S . R is *symmetric* if and only if $s_i \ r \ s_j$ implies that $s_j \ r \ s_i$. R is *transitive* if and only if $s_i \ r \ s_j$ and $s_j \ r \ s_k$ imply that $s_i \ r \ s_k$. An *equivalence relation* on S is a relation on S that is symmetric, reflexive, and transitive. The elements of S can be partitioned by an equivalence relation into disjoint subsets called *equivalence classes*.

It can be shown that state equivalence defines an equivalence relation on the set of states of a completely specified sequential circuit. Hence, the equivalence classes are used to define the states of the reduced state table.

A relation on S is said to be a *compatibility relation* if and only if the relation is reflexive and symmetric. Compatibility relations define subsets of S referred to as *compatibility classes*. These subsets are not, in general, disjoint. The subject of compatibility classes will be important when the reduction of incompletely specified state tables is discussed in the next chapter. The reader is referred to references [1–3] for more detailed discussions of equivalence and compatibility relations.

5.3.2 State Reduction in Completely Specified Circuits

We now present two techniques for determining equivalent states in completely specified sequential circuits:

1. Inspection
2. The implication table

In each case we use the technique to determine the equivalence classes of the circuit and then eliminate all but one state from each equivalence class to reduce the state table.

Inspection

The simplest and most obvious technique is that of recognizing equivalent states by inspection. In this approach, we need only recognize multiple rows in the state table that perform the same function and then remove the redundant states.

Example 5.18. In this example we show three examples of equivalent rows.

First, examine the sequential circuit defined by the state table shown in Fig. 5.41a. It can be reduced by inspection by noting that states B and D perform exactly the same function, i.e. their rows in the state table are identical, satisfying both

conditions for equivalence by having the same output and same next state for each input. Hence, state D can be removed from the table by simply removing row D and replacing state D in the remainder of the table by its equivalent state B . This procedure results in the reduced state table shown in Fig. 5.41b.

Second, analyze the sequential circuit of Fig. 5.41c. If the circuit is in state B , under an input of logic 0 it stays in the same state (loops back to itself on a state diagram) with a 0 output, and under an input of logic 1 it transitions to state A with a 1 output. On the other hand, if the circuit is in state D , we could make the same statement: under an input of logic 0 it stays in the same state with a 0 output, and under an input of logic 1 it transitions to state A with a 1 output. Hence, states B and D are equivalent by inspection. The reduced state table is given in Fig. 5.41d.

Our third example is illustrated in Fig. 5.41e. This example closely resembles circuit 2; however, we have exchanged the next states for B and D in column $x=0$. Now, analyzing the behavior of states B and D together, we note that if the pair of states are combined the behavior under input 0 requires the sequential circuit to remain in the same state (loops back to itself) with an output of logic 0, and under input $x=1$, the sequential circuit transitions to state A with an output of logic 1. So the reduced circuit of Fig. 5.41 is also valid for circuit 3. This is another case of state reduction by inspection.

In summary, two states are equivalent by inspection when the next-state rows are identical or when the next-state rows are identical except for the "self-loop-back" entries.

	x	
	0	1
A	B/0	C/1
B	C/0	A/1
C	D/1	B/0
D	C/0	A/1

(a)

	x	
	0	1
A	B/0	C/1
B	C/0	A/1
C	B/1	B/0

(b)

	x	
	0	1
A	B/0	C/1
B	B/0	A/1
C	D/1	B/0
D	D/0	A/1

(c)

	x	
	0	1
A	B/0	C/1
B	B/0	A/1
C	B/1	B/0

(d)

	x	
	0	1
A	B/0	C/1
B	D/0	A/1
C	D/1	B/0
D	B/0	A/1

(e)

Figure 5.41: State equivalence by inspection. (a) Circuit 1. (b) Reduced circuit 1. (c) Circuit 2. (d) Reduced circuit 2. (e) Circuit 3.

Implication Table

The implication table is a tool that can be used to determine state equivalence. This technique is more general in that it can also be applied to incompletely specified sequential circuits, and is better able to handle more complex state tables than the inspection method.

The example in Fig. 5.42a will be used to explain the procedure.

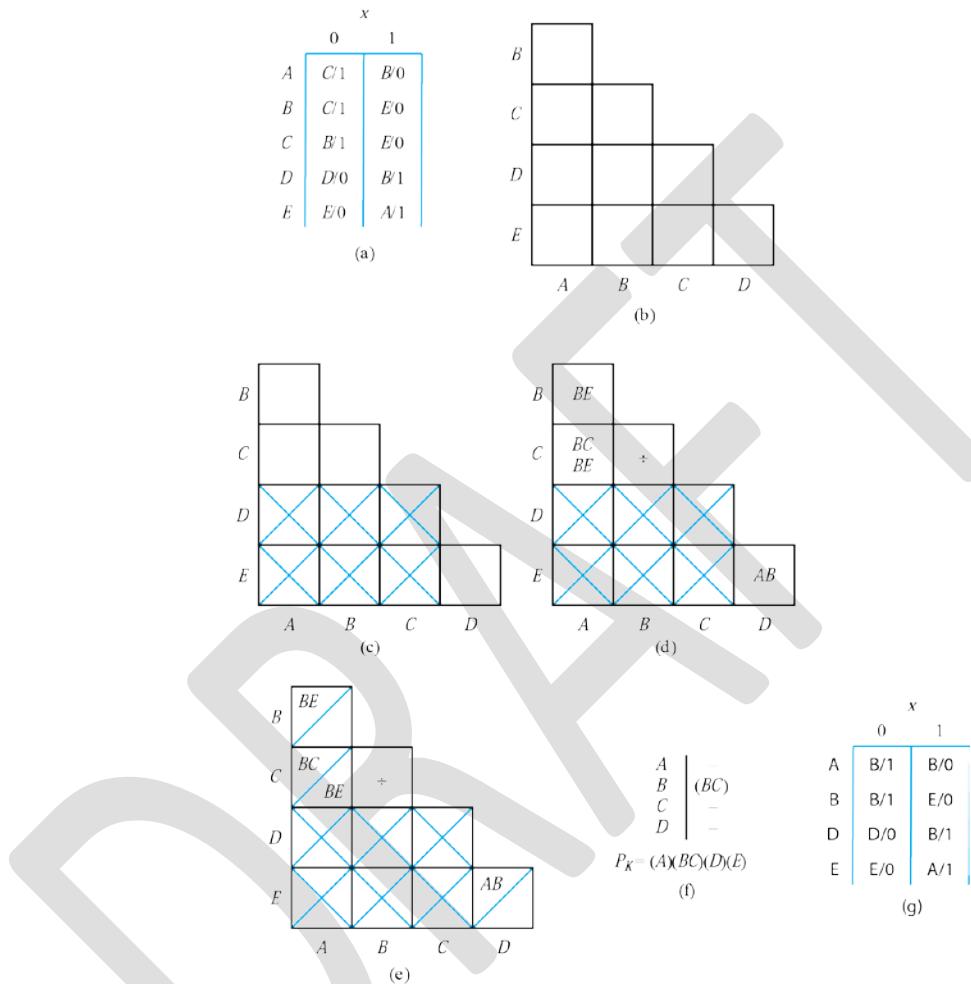


Figure 5.42: The implication table for a five-state circuit. (a) State table. (b) Implication table. (c) Output partitioning. (d) Implied pairs. (e) Completed table. (f) Equivalence partition. (g) Reduced state table.

Implication Table Procedure

- Step 1.** Form a table using the structure shown in Fig. 5.42b, which is derived by listing vertically all states in the table except the first and horizontally all states except the last. The resulting table displays all possible combinations of two states, and hence each cell in the table corresponding to the intersection of a row and column represents two states being tested for equivalence.
- Step 2.** Since only states with identical outputs can possibly be equivalent (condition 1 for equivalent states), a cross is placed in the cells corresponding to those pairs of states

whose outputs are not equal for every input. This has been done in Fig. 5.42c for the example.

- Step 3.** Using condition 2 for equivalent states, the vacant cells in Fig. 5.42c must now be completed. Into these blocks are placed the pairs of next states whose equivalence is "implied" by the two states whose intersection defines the cell. As an illustration of this, consider the cell defined by states *A* and *B*. From the state table it can be seen that, for *A* and *B* to be equivalent states, *B* and *E* must be equivalent. Hence, the pair *BE* is listed in the cell defined by *A* and *B* as shown in Fig. 5.42d. Note that if the states of the implied pair, *B* and *E*, are not equivalent, then there exists an input string beginning with $x=0$ that will produce different outputs depending on whether the initial state is *A* or *B*, meaning *A* and *B* are not equivalent.

If the implied pairs for any cell contain only the states that define the cell or if the next states of the two states defining the cell are the same state for a given input, then a check mark (\checkmark) is placed in the cell indicating that the two states defining the cell are equivalent by *inspection* and independent of any implied pairs. This condition is illustrated in Fig. 5.42d by the cell defined by states *B* and *C* and is similar to the case of equivalence by inspection seen in the example of Fig. 5.41e.

- Step 4.** Once the table has been completely filled, successive passes are made through the entire table to determine if any cells should be crossed off other than those crossed out in step 2. A cell in the table is crossed out if it contains at least one implied pair that defines a cell in the table that has previously been crossed out. This operation has been performed for the example, and the resulting table is shown in Fig. 5.42e. For example, the cell defined by *A* and *B* was crossed out because it contained the pair *BE* which defines a cell that was already crossed out. This procedure is repeated until no additional cells can be crossed off.
- Step 5.** Finally, the table shown in Fig. 5.42f is obtained by listing as a column the states that define the horizontal row of the implication table. Then the implication table is examined column by column from left to right to see if any cells are not crossed out. The states that define any cell that has not been crossed out are equivalent and are listed as an equivalent pair in the table in Fig. 5.42f. Pairs are combined using transitivity.

$$(s_i, s_j)(s_j, s_k) \rightarrow (s_i, s_j, s_k) \quad (5.6)$$

In the example, all the cells in columns *A*, *C*, and *D* are crossed out and hence dashes are placed in these rows in the table of Fig. 5.42f. In column *B* of the implication table the cell defined by states *B* and *C* is not crossed out, and hence the pair *(BC)* is placed in row *B* of the table. The equivalence partition then consists of all the equivalent states found in the table, that is, *(BC)* together with the remaining states of the circuit that are not equivalent to any other state. Figure 5.42g shows the final reduced state table, with row *C* removed, and all next state entries of *C* changed to *B*.

Example 5.21. Use an implication table to determine the equivalence partition for the sequential circuit of Fig. 5.43a.

For the circuit presented in Fig. 5.43a, the analysis is shown in Figs. 5.43b and c, with the final reduced state table shown in Fig. 5.43d, with rows D, E and F removed.

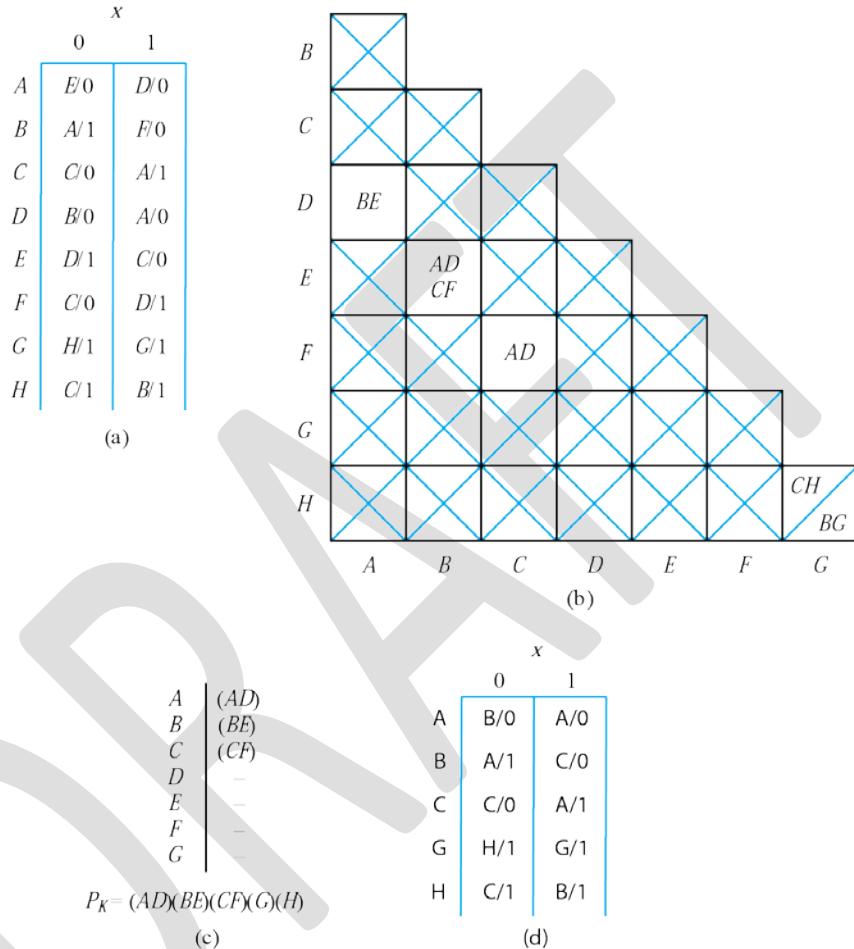


Figure 5.43: The implication table for an eight-state circuit. (a) State table. (b) Implication table. (c) Equivalence partition. (d) Reduced state table.

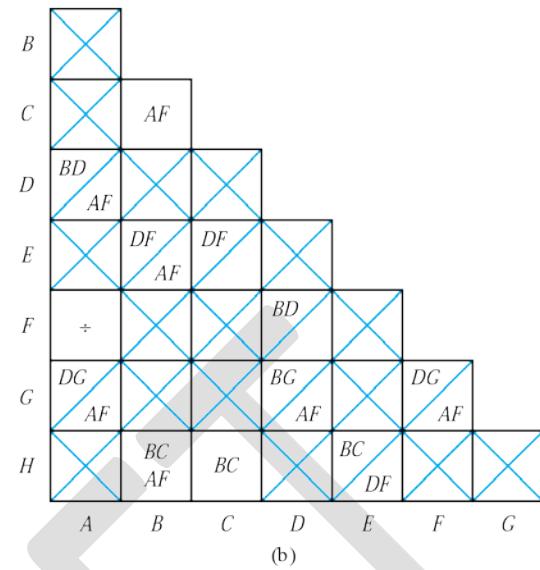
Example 5.20. For the state table of Fig. 5.44a, use the implication table method to derive a reduced state table.

The equivalence partition for the sequential circuit described by the state table of Fig. 5.44a is determined from the implication table in Fig. 5.44b, and is shown in Fig. 5.44c. This example, although straightforward, does contain one salient feature. In row *B* of Fig. 5.44c is listed the set of equivalent states $(BC)(BH)$, while (CH) is listed in row *C*. Equation (5.6) may be used to combine these states into the larger equivalence class (BCH) . Therefore, both rows *C* and *H* are deleted in the reduced state table of Fig. 5.44d, in addition to row *F*, since we

also had equivalence class (*AF*).

	00	01	11	10
	x_1	x_2		
A	D/0	D/0	F/0	A/0
B	C/1	D/0	E/1	F/0
C	C/1	D/0	E/1	A/0
D	D/0	B/0	A/0	F/0
E	C/1	F/0	E/1	A/0
F	D/0	D/0	A/0	F/0
G	G/0	G/0	A/0	A/0
H	B/1	D/0	E/1	A/0

(a)



(b)

A	(AF)
B	(BC)(BH)
C	(CH)
D	-
E	-
F	-
G	-

Note: $(BC)(BH)(CH) = (BCH)$

$$P_K = (AF)(BCH)(D)(E)(G)$$

(c)

	00	01	11	10
	x_1	x_2		
A	D/0	D/0	A/0	A/0
B	B/1	D/0	E/1	A/0
D	D/0	B/0	A/0	A/0
E	B/1	A/0	E/1	A/0
G	G/0	G/0	A/0	A/0

(d)

Figure 5.44: The implication table for a seven-state circuit. (a) State table. (b) Implication table. (c) Equivalence partition. (d) Reduced state table.

The above examples, which illustrate the use of the implication table, can be compared with the previously used inspection method. In general, the implication table approach is more routine, but it is also more tedious than other approaches.

In this section we have examined two techniques for minimizing the number of states of synchronous sequential circuits. Either one may be used as step 3 of the synthesis process.

5.4 Summative Design Examples

5.4.1 Drink Vending Machine Control Unit

Let us design a control unit for a simple drink vending machine. Drinks cost \$1.50, and the machine accepts dollar bills and quarters. Change should be returned if more than \$1.50 is deposited. No more one piece of currency can be deposited at a time; therefore, the maximum change is three quarters.

A block diagram of the drink machine is given in Fig. 5.45. The control unit has two inputs, D and Q , which are outputs of the currency detector. The currency detector generates a 1 on its output signal D if a dollar is deposited and a 1 on its output signal Q if a quarter is deposited. The D and Q lines automatically reset to 0 on the next clock pulse. We shall assume that it is physically impossible to insert both a dollar and a quarter at the same time, and therefore we cannot have $D = Q = 1$ in the same clock period.

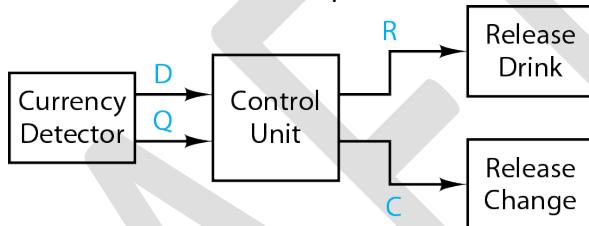


Figure 5.45: Coin-operated drink machine design. (a) Block diagram. (b) Control unit state diagram.

In addition to inputs D and Q from the currency detector, the control unit has two outputs, R and C , which activate drink and coin release mechanisms, respectively. The drink is released by a 1 on signal R , and a quarter in change is released by a 1 on signal C .

The state of the control unit represents the total amount of money deposited for the current transaction. The set of states is thus $\{S_0, S_{25}, S_{50}, S_{75}, S_{100}, S_{125}, S_{200}, S_{225}\}$, and the operation of the controller can be represented by the state diagram of Fig. 5.4637a. For a given state, the R and C outputs depend on which item of currency is inserted; therefore, a Mealy model has been used. If a deposited item increases the amount to \$1.50 or more, the control unit will return to state 0 while releasing the drink, along with releasing change, if necessary. If a dollar is inserted after \$1.00 has already been deposited, the circuit will enter a temporary state, S_{200} , releasing the drink and one quarter in change, and then return to state S_0 on the next clock cycle while releasing the second quarter in change. If a dollar is inserted after \$1.25 has been deposited, for a total of \$2.25, requiring three quarters in change, the circuit will enter temporary state S_{225} , while releasing the drink and one quarter, and then enter state S_{200} while releasing the second quarter, and finally return to state S_0 while releasing the third quarter.

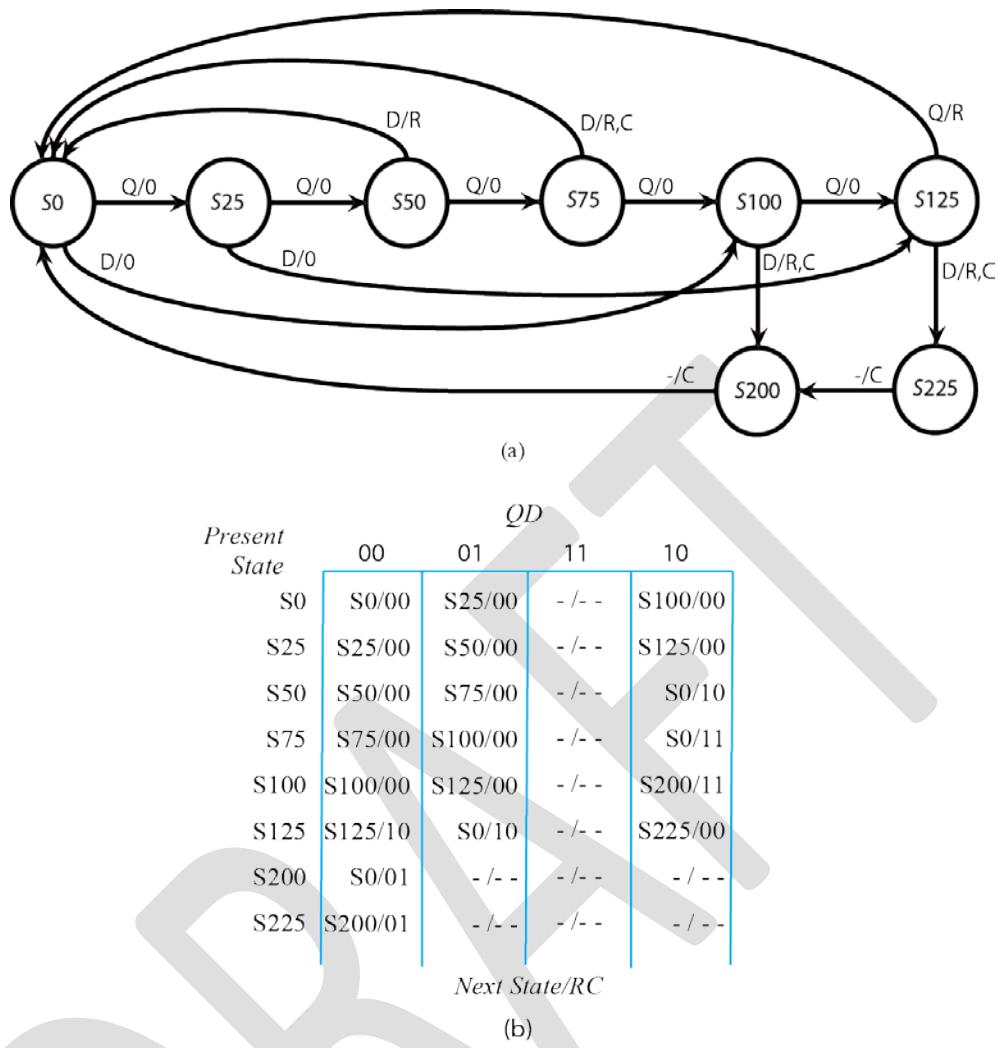


Figure 5.46. Drink machine controller. (a) State diagram.(b) State table.

Note that the state diagram of Fig. 5.46a has been simplified to facilitate analysis. At most two arcs are shown leaving each state, labelled *D* if a dollar has been inserted (i.e. the input combination is $DQ = 10$), and *Q* if a quarter has been inserted (input combination $DQ = 01$). Arcs for $DQ = 00$ are not shown, since it is implied that the state does not change if nothing is deposited, and arcs for $DQ = 11$ are not shown, since two items cannot be deposited simultaneously. The state table is shown in Fig. 5.46b. Note that this state table indicates an incompletely-specified finite state machine. Since $DQ = 11$ cannot occur, in that column of the state table don't care conditions are entered for the next states and outputs. The next state entries in the column for $DQ = 00$ result in the state remaining the same, with the exception of states \$200 and \$225. If either of these two states is entered, it is assumed that no additional currency will be inserted until after this transaction is complete. Therefore, don't cares are entered in all columns except for $DQ = 00$. In column $DQ = 00$, the state changes from S225 to S200, and from S200 back to S0, as indicated in the state diagram.

The following HDL behavioral model of the drink machine controller can be used to verify the behavior of the state table of Fig. 5.46b. The completion of a synchronous sequential logic circuit design from the state table is left as an exercise for the reader.

```

//Vending Machine Controller
module NewVendingMachineController (
    input Q, D, CLK, CLR,           //declare inputs
    output reg R, C);             //declare outputs
    reg [2:0] state;              // state variables

    // state assignment
    parameter S0=3'b000, S25=3'b001, S50=3'b011, S75=3'b111, S100=3'b110, S125=3'b100,
    S200=3'b010, S225=3'b101;
    // state transitions on negative edge of CLK or CLR
    always @ (negedge CLK, negedge CLR)
        if (CLR==0) state <= S0;      // reset the controller to state S0
        else
            case (state)
                S0: if (Q == 1) state <= S25; else if (D == 1) state <= S100;
                S25: if (Q == 1) state <= S50; else if (D == 1) state <= S125;
                S50: if (Q == 1) state <= S75; else if (D == 1) state <= S0;
                S75: if (Q == 1) state <= S100; else if (D == 1) state <= S0;
                S100: if (Q == 1) state <= S125; else if (D == 1) state <= S200;
                S125: if (Q == 1) state <= S0; else if (D == 1) state <= S225;
                S200: state <= S0;
                S225: state <= S200;
            endcase
    // Mealy model outputs depend on state and inputs
    always @ (state, Q, D)
        case (state)                  // amount deposited
            S0: begin R = 0; C = 0; end // 0
            S25: begin R = 0; C = 0; end // $0.25
            S50: if (Q == 1) begin R = 0; C = 0; end // $0.75
                    else if (D == 1) begin R = 1; C = 0; end // $1.50: drink
                    else begin R = 0; C = 0; end // $0.50
            S75: if (Q == 1) begin R = 0; C = 0; end // $1.00
                    else if (D == 1) begin R = 1; C = 1; end // $1.75: drink + 1 quarter
                    else begin R = 0; C = 0; end // $0.75
            S100: if (Q == 1) begin R = 0; C = 0; end // $1.25
                    else if (D == 1) begin R = 1; C = 1; end // $2.00: drink + 2 quarters
                    else begin R = 0; C = 0; end // $1.00
            S125: if (Q == 1) begin R = 1; C = 0; end // $1.50: drink
                    else if (D == 1) begin R = 1; C = 1; end // $2.25: drink + 3 quarters
                    else begin R = 0; C = 0; end // $1.25
            S200: begin R = 0; C = 1; end // 1 extra quarter
            S225: begin R = 0; C = 1; end // 1 extra quarter
        endcase
    endmodule

```

5.4.2 Binary multiplier design example

We wish to design the control unit for a binary multiplier that will compute the 8-bit product of two 4-bit unsigned binary numbers using a series of add and shift operations. The multiplier begins when pulsed by a *Reset* signal and halts with the product on its outputs. A *Halt* signal is to indicate the end of the operation.

We develop the multiplication algorithm by first examining the "pencil and paper" algorithm, as described in Chapter 1. Consider the product of $(0111)_2$ and $(1010)_2$.

	0	1	1	1	Multiplicand
\times	1	0	1	0	Multiplier
	0	0	0	0	Partial product 1
	0	1	1	1	Partial product 2
	0	0	0	0	Partial product 3
	0	1	1	1	Partial product 4
	1	0	0	0	Product

Multiplier bits are examined sequentially from right to left. If the multiplier bit is 1, the partial product is simply the multiplicand, and if the multiplier bit is 0, the partial product is 0000. Each new partial product is shifted one bit position to the left before adding it to the total, since multiplier bits are weighted by factors of 2. Alternatively, we can keep the position of the partial product fixed and shift the total to the right after each addition. We shall use the latter approach for our control unit.

The data path of the binary multiplier requires three registers and a binary adder, as illustrated in Fig. 5.47a. The registers serve the following functions.

Register A: A 5-bit shift register that holds the four most significant bits of the product and the carry from the adder. This register must be able to be cleared to all 0s, loaded with the adder output, and shifted right, with 0 shifted into to leftmost bit.

Register Q: A 4-bit shift register. Q is initially loaded with the multiplier. In each iteration, Q will be shifted one position to the right, with its leftmost bit replaced by one bit of the product, so that at the end of the operation Q will contain the lower four bits of the product.

Register M: A 4-bit parallel register that holds the multiplicand.

In addition to these components, a 2-bit binary counter, CNT , is used to count the number of iterations. It will be initialized to 00 and incremented after each shift operation, returning to 00 after the fourth iteration. A logic gate will be used to indicate the 11 condition, which occurs in the fourth iteration of the algorithm, indicating that the algorithm should be terminated.

The product will be computed by adding the multiplicand to the current total in register A when the tested multiplier bit is 1. However, instead of adding a partial product of 0000 to the total when the multiplier bit is 0, we shall simply skip the addition step.

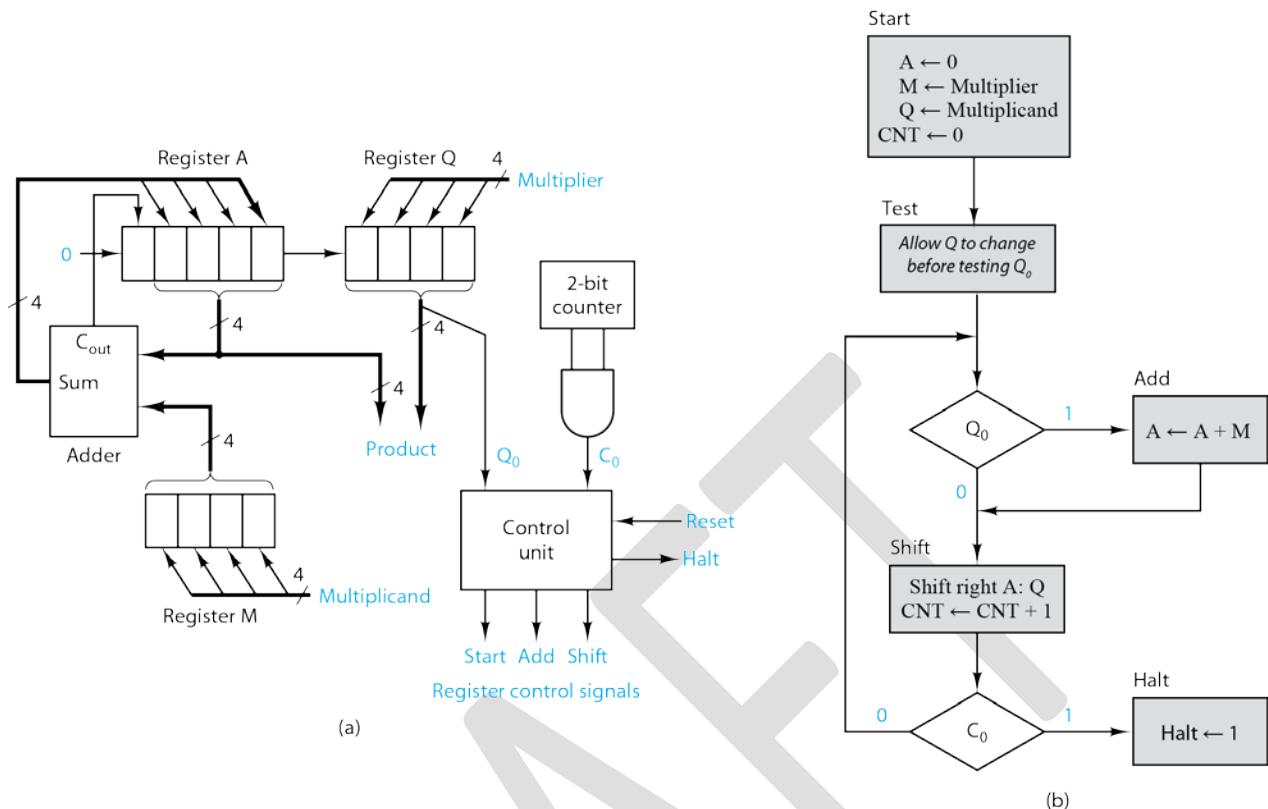


Figure 5.47: Binary multiplier. (a) Block diagram. (b) ASM diagram.

The controller has three inputs: the *Reset* signal initializes the algorithm, Q_0 is the multiplier bit to be tested, and C_0 is 1 when the counter contains 11, indicating the end of the fourth add and shift cycle. The ASM diagram of the controller is given in Fig. 5.47b. This diagram is for a Moore model circuit with the following five states:

- Start:** Load the operands into the M and Q registers, and clear the A register and counter CNT . This is the initial state following activation of the *Reset* signal.
- Test:** This state allows time for register Q to change, after the *Start* and *Shift* states, before testing Q_0 . No other operations are performed in this state.
- Add:** Add the multiplicand (M) to the current partial product (A) by loading the sum and carry outputs of the binary adder into the A register.
- Shift:** Shift the partial product and multiplier 1 bit to the right by enabling the shift control lines of the A and Q registers. Also, increment the counter by enabling its increment control line.
- Halt:** The final product in the A and Q registers and output *Halt* is activated.

One multiplier bit is tested in each iteration of the algorithm to determine whether to enter the *Add* state or to skip the *Add* state and go directly to the *Shift* state. The counter output is tested after the *Shift* state to determine whether to halt or perform another iteration.

The controller has four outputs, active in the Start, Add, Shift and Halt states of the ASM diagram in Fig. 5.47b. The *Start* output activates the load control lines of the *M* and *Q* registers and the clear lines of the *A* register and counter, to perform the four operations listed in the Start state. The *Add* output activates the load control line of the *A* register to load the sum of the *A* and *M* registers. The *Shift* output activates the shift control lines of the *A* and *Q* registers and the increment control line of the counter. The *Halt* output signals the end of the operation.

To verify the design, the behavior of the ASM diagram of Fig. 5.47b can be modeled as shown in the following HDL model of the multiplier controller, and then used to simulate the operation of the multiplier. In this model of the controller, note that five flip-flops have been used to represent the state of the controller. This assumes the use of a *one hot* state assignment, with one state variable per state, as will be discussed in the next section.

```
//Multiplier controller. Verilog behavioral model.
module MultControl (
    input Clock, Reset, Q0, C0,           //declare inputs
    output Start, Add, Shift, Halt);      //declare outputs
    reg [4:0] state;                    //five states (one hot – one flip-flop per state)
    //one-hot state assignments for five states
    parameter StartS=5'b00001, TestS=5'b00010, AddS=5'b00100, ShiftS=5'b01000,
    HaltS=5'b10000;
    reg [1:0] Counter; //2-bit counter for # of algorithm iterations

    // State transitions on positive edge of Clock or Resets
    always @(posedge Clock, posedge Reset)
        if (Reset==1) state <= StartS;          //enter StartS state on Reset
        else                      //change state on Clock
            case (state)
                StartS: state <= TestS;          // StartS to TestS
                TestS: if (Q0) state <= AddS;     // TestS to AddS if Q0=1
                    else state <= ShiftS;        // TestS to ShiftS if Q0=0
                AddS: state <= ShiftS;          // AddS to ShiftS
                ShiftS: if (C0) state <= HaltS;   // ShiftS to HaltS if C0=1
                    else state <= TestS;        // ShiftS to TestS if C0=0
                HaltS: state <= HaltS;          // stay in HaltS
            endcase

    // Moore model - activate one output per state
    assign Start = state[0]; // Start=1 in state StartS, else 0
    assign Add = state[2]; // Add=1 in state AddS, else 0
    assign Shift = state[3]; // Shift=1 in state ShiftS, else 0
    assign Halt = state[4]; // Halt=1 in state HaltS, else 0
endmodule
```

The following is an HDL model of the multiplier block diagram in Fig. 5.47a, with the controller of the previous HDL example instantiated within this top-level model. Registers A, Q and M, and the 2-bit counter, have each been modeled with a single block triggered on the rising edge of the clock. The different functions of each register/counter are selected by the controller output signals, and the adder has been incorporated into accumulator register A.

```
//Multiplier. Verilog behavioral model.
module Multiplier (
    input Clock, Reset,           //declare inputs
    input [3:0] Multiplicand,
    input [3:0] Multiplier,
    output [7:0] Product,          //declare outputs
    output Halt);
    reg [3:0] RegQ, RegM;        // Q and M registers
    reg [4:0] RegA;              // A register
    reg [1:0] Count;             //2-bit iteration counter
    wire C0, Start, Add, Shift;
    assign Product = {RegA[3:0],RegQ}; //product = A:Q

    // 2-bit counter for #iterations
    always @(posedge Clock)
        if (Start == 1) Count <= 2'b00;           // clear in Start state
        else if (Shift == 1) Count <= Count + 1; // increment in Shift state
        assign C0 = Count[1] & Count[0];          // detect count = 3

    // Multiplicand register (load only)
    always @(posedge Clock)
        if (Start == 1) RegM <= Multiplicand; // load in Start state

    // Multiplier register (load, shift)
    always @(posedge Clock)
        if (Start == 1) RegQ <= Multiplier;      // load in Start state
        else if (Shift == 1) RegQ <= {RegA[0],RegQ[3:1]}; // shift in Shift state

    // Accumulator register (clear, load, shift)
    always @(posedge Clock)
        if (Start == 1) RegA <= 5'b00000;        // clear in Start state
        else if (Add == 1) RegA <= RegA + RegM; // load in Add state
        else if (Shift == 1) RegA <= RegA >> 1; // shift in Shift state

    // Instantiate controller module
    MultControl Ctrl (Clock, Reset, RegQ[0], C0, Start, Add, Shift, Halt);

endmodule
```

These two HDL models can be simulated to verify the design of both the controller and the top-level design. For proper verification of the design, correct multiplication of a range of operands should be verified in simulation, with timing of the clock, controller state, controller outputs, and register values examined to identify any unexpected results.

5.4.3 Traffic Light Controller Design Example

A traffic light controller is an example of a sequential circuit that is pervasive around the world. The complexity of a traffic light controller depends upon the traffic patterns to be controlled and any special features that may be provided. Complexity is also influenced by the logic components selected to realize the design. The design and realization of a basic controller will be illustrated first using a variety of components. Various features will then be added to the requirements to further illustrate the design of sequential circuits.

Basic Traffic Light Controller – The basic controller will be designed to handle traffic at the intersection of two two-lane roads as illustrated Fig. 5.48a. It is assumed that standard *Green-Yellow-Red* traffic lights are used to display right-a-way conditions to drivers in each of the four directions. The controller will first allow north-bound and south-bound traffic to flow concurrently while cross-traffic is stopped in both directions. Traffic flow will then be switched to give the right-a-way to east-bound and west-bound traffic. A warning light will be displayed before flow patterns are switched. Thusly, the controller must generate outputs to control north-south lights (G_{NS} , Y_{NS} , R_{NS}) and east-west lights (G_{EW} , Y_{EW} , R_{EW}) as illustrated in Fig. 5.48b. Inputs to the controller are *Clock* and *Reset* signals. The clock period is assumed to be 10-seconds. Design of the clock is not considered in this example.

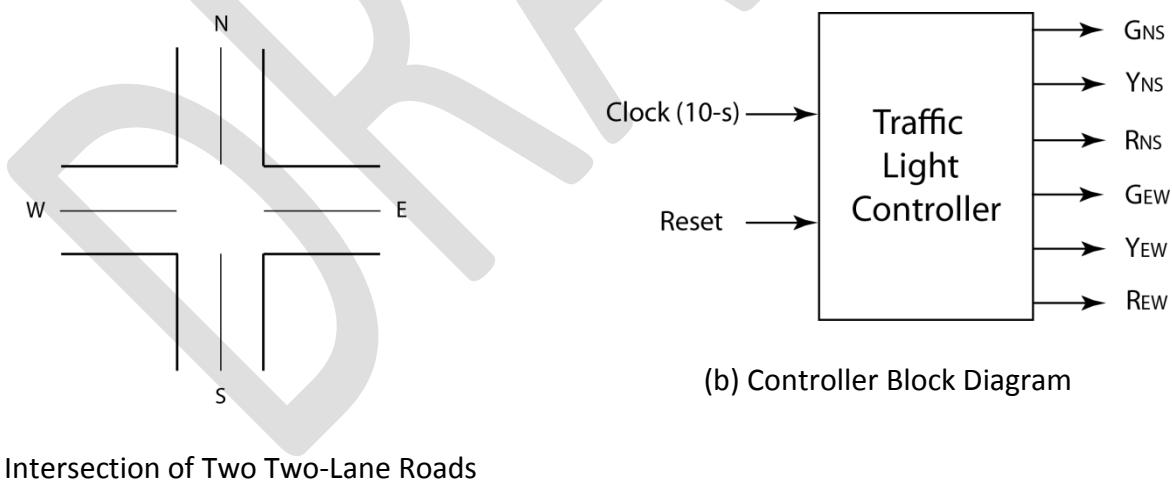


Figure 5.48: Basic Traffic Light Controller.

The first step in our design procedure is to develop a state diagram of a machine that meets these requirements. Do we need a Mealy machine or a Moore machine or does it matter? Traffic-light pattern changes need to be glitch-free in order to avoid unsafe and confusing control sequences. This suggests that a Moore machine model should be used for the controller rather than a Mealy machine. Thusly, a four-state Moore machine can be defined

with each state corresponding to one of the required output patterns as follows.

- State A – north-south green (G_{NS}) and east-west red (R_{EW}) *On* and all others *Off*.
- State B – north-south yellow (Y_{NS}) and east-west red (R_{EW}) *On* and all others *Off*.
- State C – east-west green (G_{EW}) and north-south red (R_{NS}) *On* and all others *Off*.
- State D – east-west yellow (Y_{EW}) and north-south red (R_{NS}) *On* and all others *Off*.

These states and the required transitions between states are defined in state diagram form in Fig. 5.49. Transitions occur each clock cycle, so the time spent in each state is 10-seconds. Output variables are shown for lights that are *On* and are omitted for lights that are *Off*.

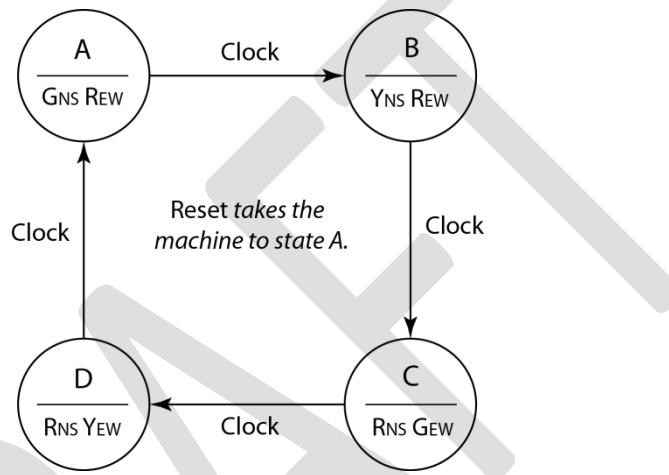


Figure 5.49: State Diagram of Basic Traffic Light Controller.

The above Moore machine meets the basic requirements of the traffic light controller, but the *green time* of 10-seconds in each direction may be too short for practical purposes. *Green time* can be doubled, for example, by increasing the clock period to 20-seconds, but that would also double *yellow time* which may not be needed or desired. Another approach to doubling *green time* while keeping *yellow time* unchanged would be to add additional *green* states (A2 and C2) for each traffic pattern as shown in Fig. 5.50. This approach could be extended to provide even more *green time* as desired. The remainder of this example will be based on the modified state diagram in Fig. 5.50.

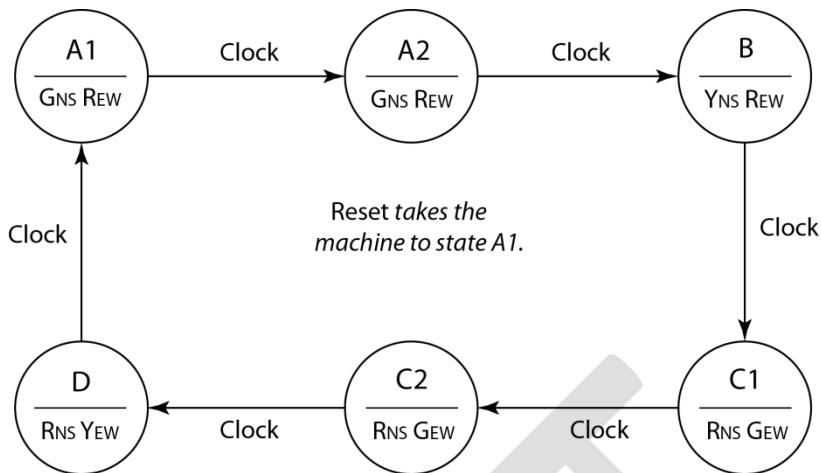


Figure 5.50: Modified State Diagram.

The next step in our process is to look for redundant states that can be eliminated. However, the simplicity and regularity of the modified state diagram strongly suggests that there are no redundant states, so this step will be omitted in this example.

The next steps in the design process are to construct a state table from the state diagram, as shown in Fig. 5.51, and to make a state assignment. State assignments have a strong influence on the complexity of the final realizations, so three different state assignments will be used to illustrate this point and to find the *best* solution in terms of the number of devices needed.

Present State	Next State		Outputs					
	Clock = 0	Clock = 1	G _{NS}	Y _{NS}	R _{NS}	G _{EW}	Y _{EW}	R _{EW}
A1	A1	A2	1	0	0	0	0	1
A2	A2	B	1	0	0	0	0	1
B	B	C1	0	1	0	0	0	1
C1	C1	C2	0	0	1	1	0	0
C2	C2	D	0	0	1	1	0	0
D	D	A1	0	0	1	0	1	0

Figure 5.51: State Table for Modified State Diagram.

The three state assignments are detailed in Figure 5.52. Assignment 1 corresponds to a binary code sequence 110 and 111 being unused. Assignment 2 is a Grey code sequence which can be realized by a 3-bit twisted-ring counter. Assignment 3 is a one-hot assignment which can be realized by a 6-bit ring counter.

<i>State</i>	<i>Assignment 1</i>	<i>Assignment 2</i>	<i>Assignment 3</i>
<i>(State Variables)</i>	$(y_2y_1y_0)$	$(y_2y_1y_0)$	$(y_5y_4y_3y_2y_1y_0)$
A1	000	000	000001
A2	001	001	000010
B	010	011	000100
C1	011	111	001000
C2	100	110	010000
D	101	100	100000

Figure 5.52: Alternative State Assignments for Modified State Diagram.

The transition table using Assignment 1 is shown in Figure 5.53. The output equations can be easily derived from the table given that we're designing a Moore machine. They are

$$G_{NS} = y_2'y_1'$$

$$G_{EW} = y_2'y_1y_0 + y_2y_1'y_0'$$

$$Y_{NS} = y_2'y_1y_0'$$

$$Y_{EW} = y_2y_1'y_0$$

$$R_{NS} = (G_{NS} + Y_{NS})'$$

$$R_{EW} = (G_{EW} + Y_{EW})'$$

$y_2y_1y_0$	$y_2y_1y_0$		Outputs					
	Clock = 0	Clock = 1	G_{NS}	Y_{NS}	R_{NS}	G_{EW}	Y_{EW}	R_{EW}
000	000	001	1	0	0	0	0	1
001	001	010	1	0	0	0	0	1
010	010	011	0	1	0	0	0	1
011	011	100	0	0	1	1	0	0
100	100	101	0	0	1	1	0	0
101	101	000	0	0	1	0	1	0

Figure 5.53: Transition Table for State Assignment 1.

The type of flip-flop to be used for memory must now be selected before the detailed design can be completed. Positive-edge triggered JK flip-flops with active-low Preset will be used for this example. K-maps showing next state transitions and JK inputs for each state variable are shown in Fig. 5.54.

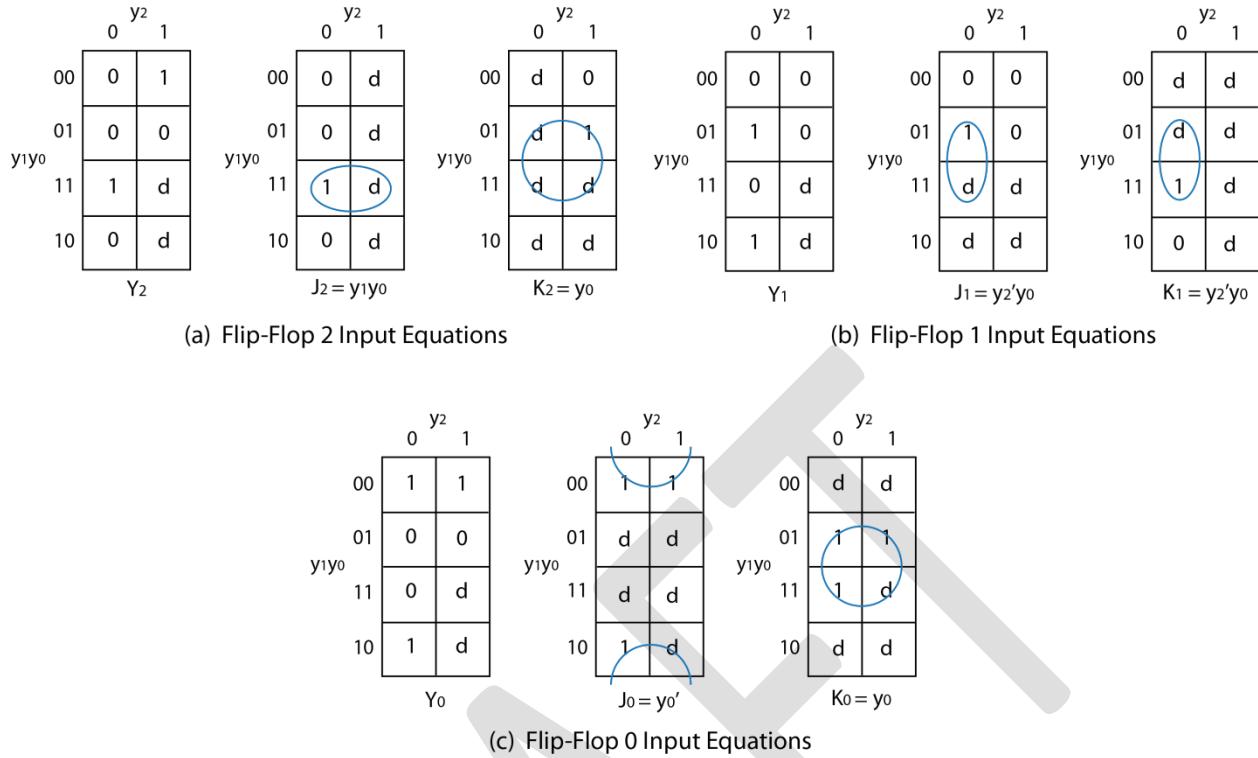


Figure 5.54: Derivation of Flip-Flop Input Equations.

Using NAND and NOR gates for the combinational logic functions produces the realization of the controller as shown in Figure 5.55. The cost of this realization is three flip-flops and ten gates. If standard integrated circuits were used, two chips would be needed for the flip-flops, three for NOR gates, and one for NAND gates for a total of six chips.

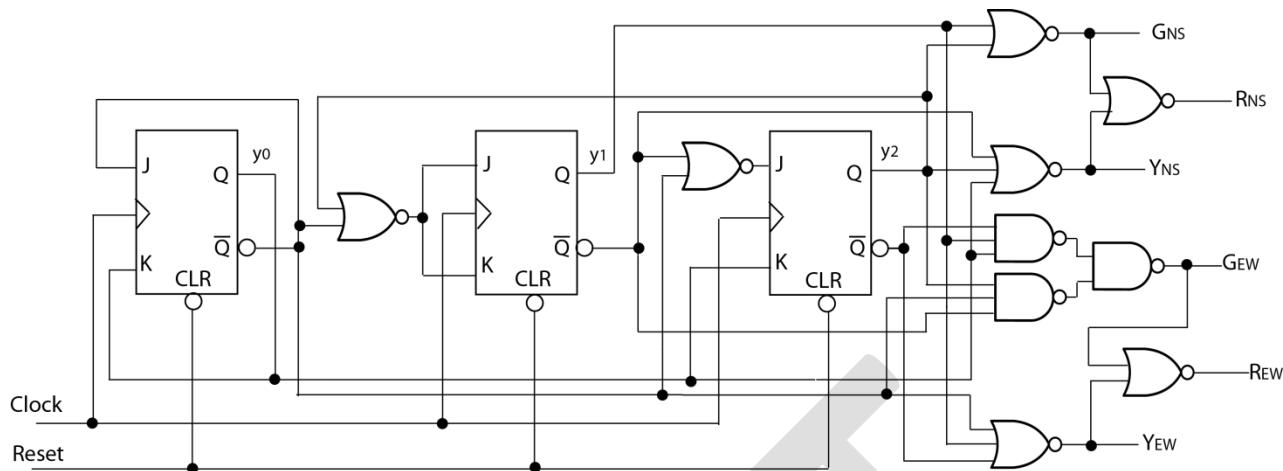


Figure 5.55: Controller Realization Using State Assignment 1 and JK Flip-Flops.

Now let's develop a realization based on state assignment 2. The transition table for this assignment is shown in Figure 5.56. As before, output equations can be derived from the table and are listed below. Note the equation for G_{EW} is considerably simpler than before due to the logic adjacencies produced by using the Grey code state assignment.

$$\begin{array}{lll} G_{NS} = y_2'y_1' & Y_{NS} = y_2'y_1y_0 & R_{NS} = (G_{NS} + Y_{NS})' \\ G_{EW} = y_2y_1 & Y_{EW} = y_2y_1'y_0' & R_{EW} = (G_{EW} + Y_{EW})' \end{array}$$

$y_2y_1y_0$	$y_2y_1y_0$		Outputs					
	Clock = 0	Clock = 1	G_{NS}	Y_{NS}	R_{NS}	G_{EW}	Y_{EW}	R_{EW}
000	000	001	1	0	0	0	0	1
001	001	011	1	0	0	0	0	1
011	011	111	0	1	0	0	0	1
111	111	110	0	0	1	1	0	0
110	110	100	0	0	1	1	0	0
100	100	000	0	0	1	0	1	0

Figure 5.56: Transition Table for State Assignment 2.

Another advantage of using a twister-ring counter is that the flip-flop input equations can be easily derived without the need to employ K-Maps. Recall from Chapter 4 that for an n -bit twisted-ring counter input $D_i = y_{i-1}$, $i = 1$ to $n-1$, and $D_0 = y_{n-1}$. So for this example, $D_0 = y_2'$, $D_1 = y_0$, and $D_2 = y_1$. These coupled with the above output equations yields the realization shown in Figure 5.57. The cost of this circuit is three flip-flops and six gates, or four standard integrated circuits.

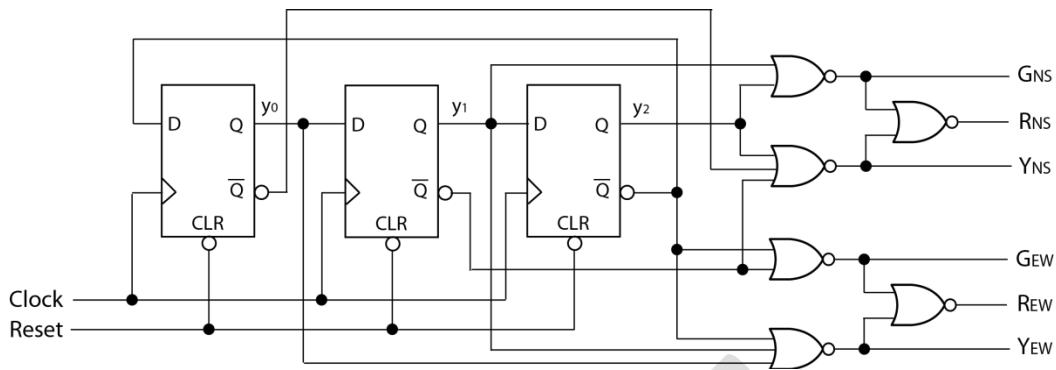


Figure 5.57: Controller Realization Using a Twisted-Ring Counter.

Now let's realize the controller using a one-hot state assignment which in this example is equivalent to using a 6-bit ring counter for the sequencer part of the design. The specific assignment and corresponding transition table is given in Figure 5.58.

$Y_5Y_4Y_3Y_2Y_1Y_0$	$Y_5Y_4Y_3Y_2Y_1Y_0$		Outputs					
	Clock = 0	Clock = 1	G _{NS}	Y _{NS}	R _{NS}	G _{EW}	Y _{EW}	R _{EW}
000001	000001	000010	1	0	0	0	0	1
000010	000010	000100	1	0	0	0	0	1
000100	000100	001000	0	1	0	0	0	1
001000	001000	010000	0	0	1	1	0	0
010000	010000	100000	0	0	1	1	0	0
100000	100000	000001	0	0	1	0	1	0

Figure 5.58: Transition Table for State Assignment 3.

The one-hot assignment yields simple output equations and flip-flop input equations as listed below. The main drawback is the number of flip-flops that are needed. Figure 5.59 shows the resulting circuit of six flip-flops and four gates. Three integrated circuits are required for flips and two for the gates for a total of five.

$$\begin{aligned}
 G_{NS} &= y_0 + y_1 & Y_{NS} &= y_2 & R_{NS} &= (G_{NS} + Y_{NS})' \\
 G_{EW} &= y_3 + y_4 & Y_{EW} &= y_5 & R_{EW} &= (G_{EW} + Y_{EW})' \\
 D_0 &= y_5 & D_1 &= y_0 & D_2 &= y_1 & D_3 &= y_2 & D_4 &= y_3 & D_5 &= y_4
 \end{aligned}$$

So which realization is the best? Cost using integrated circuit count as the measure favors the design using a twisted-ring counter. Wiring complexity favors the ring-counter based design. The chip count of the first design could be reduced by using a single-chip binary counter rather than three flip-flops. Yet it would still be the most complex to wire.

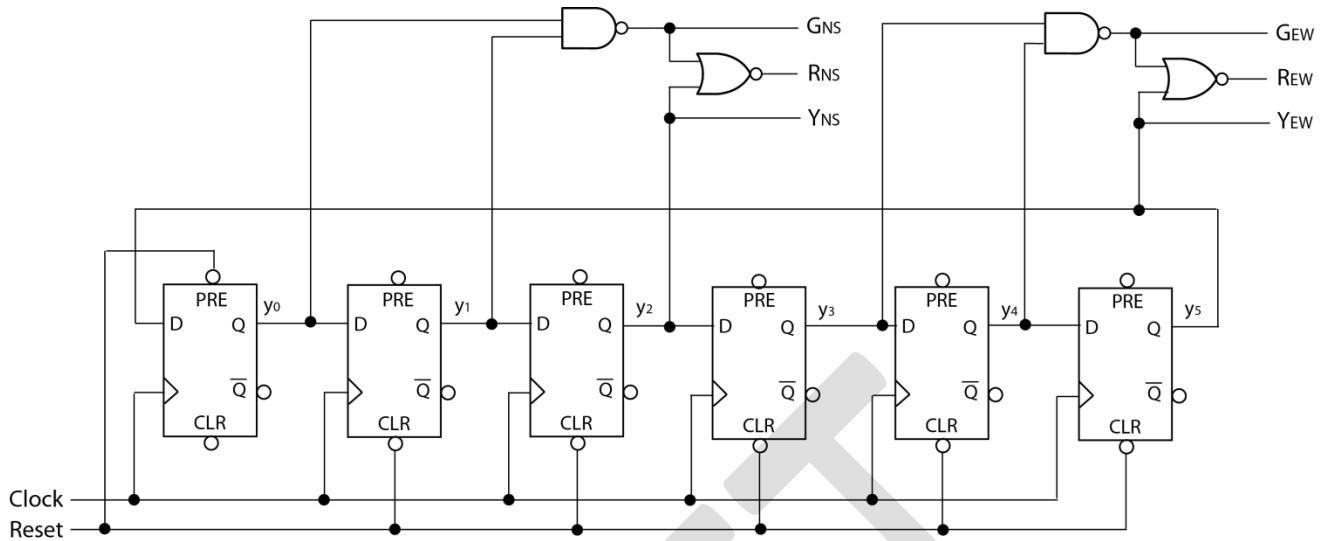


Figure 5.59: Controller Realization Using a Ring-Counter.

Before adding features, such as emergency vehicle, pedestrian, and left-turn cycles, let's write a Verilog behavioral model for the basic controller that can be used for an FPGA realization.

```
//Verilog Behavioral Model of A Basic Trafic Light Controller
module TLCbasicVerilog (Clock, Reset, Gns, Yns, Rns, Gew, Yew, Rew);
    input Clock, Reset;                                //declare input variables
    output reg Gns, Yns, Rns, Gew, Yew, Rew;          //declare output signals
    reg [2:0] state, nextstate;                        //declare state and next state variables
    parameter A1 = 3'b0, A2 = 3'b1, B = 3'b11, C1 = 3'b111, C2 = 3'b110, D = 3'b100;
    always @ (posedge Clock, negedge Reset)           //Detect input variable transitions
        if (Reset == 1'b0) state = A1;                //Take controller to state A1 on Reset
        else state = nextstate;                       //Change state
    always @ (state)                                 //Derive next state and ouptuts according to state diagram in Figure 4
        case (state)
            A1: begin nextstate=A2;Gns=1'b1;Yns=1'b0;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1; end
            A2: begin nextstate=B; Gns=1'b1;Yns=1'b0;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1; end
            B:  begin nextstate=C1;Gns=1'b0;Yns=1'b1;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1; end
            C1: begin nextstate=C2;Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b1;Yew=1'b0;Rew=1'b0; end
            C2: begin nextstate=D;Gns=1'b0;Yn=1'b0;Rns=1'b1;Gew=1'b1;Yew=1'b0;Rew=1'b0; end
            D:  begin nextstate=A1;Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b1;Rew=1'b0; end
            default: begin nextstate=A1;Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1; end
        endcase
endmodule
```

Figure 5.60: Verilog Behavioral Model of Modified Controller.

Emergency Vehicle Cycle – Many traffic light controllers feature a cycle that allows emergency vehicles (EV) to take control of an intersection while responding to a call. The previous

controller will now be revised to accommodate such a requirement. An additional input, EV, is needed to signal the approach of an emergency vehicle either directly from the vehicle or from a central control point. No change is needed for the controller outputs. The revised block diagram is shown in Fig. 5.61a. The revised controller will be designed to enter and stay an emergency state (E) when $EV = 1$ and return to state A1 and resume a normal cycle when EV returns to 0. The revised state diagram reflecting this requirement is shown in Fig. 5.61b.

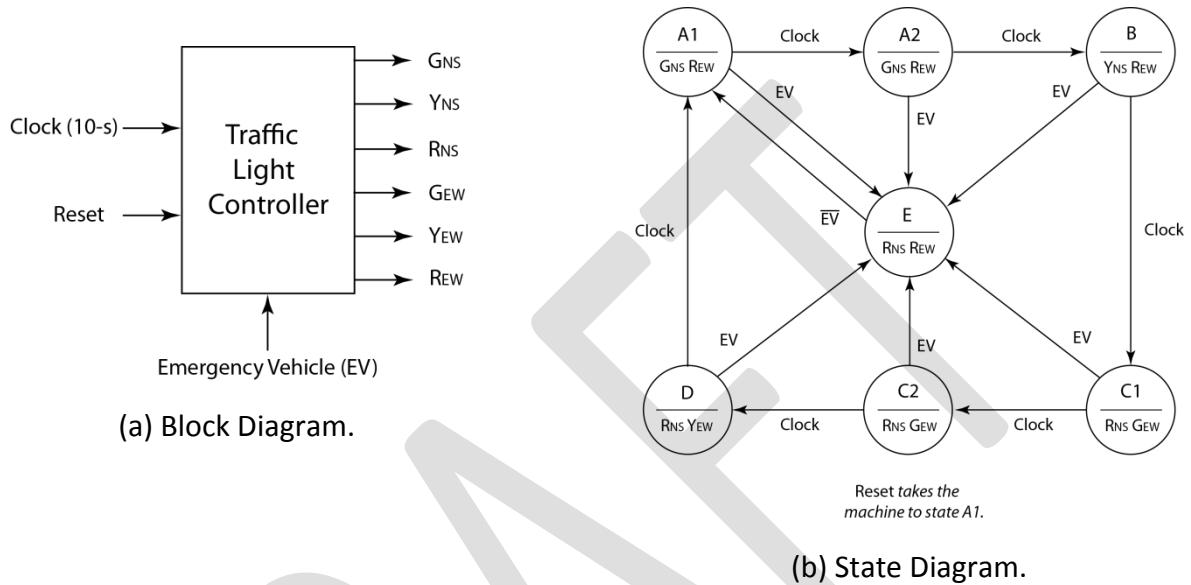


Figure 5.61: Controller with Emergency Cycle.

Since EV is an asynchronous event, the design approach used here is to employ flip-flop PRE and CLR inputs to drive the controller to state E when $EV = 1$. This will allow a previous design to be used to produce the normal sequence when $EV = 0$. The Grey code design was selected and results in the transition table shown in Fig. 5.62. The state assignment is the same Grey code state assignment previously used except for state E which is assigned state 010.

State	$y_2y_1y_0$	$y_2y_1y_0$		Outputs					
		$EV = 0$	$EV = 1$	GNS	YNS	RNS	GEW	YEW	REW
A1	000	001	010	1	0	0	0	0	1
A2	001	011	010	1	0	0	0	0	1
B	011	111	010	0	1	0	0	0	1
C1	111	110	010	0	0	1	1	0	0
C2	110	100	010	0	0	1	1	0	0
D	100	000	010	0	0	1	0	1	0
E	010	000	010	0	0	1	0	0	1

Figure 5.62: Transition Table for Controller with Emergency Cycle.

When $EV = 0$, the controller follows the Grey code sequence 000-001-011-111-110-100-000 and

never reaches state 010. When $EV = 1$, the controller goes to state 010 and stays until $EV = 0$ when it will go to state 000 and resume the normal cycle. Designing the logic circuits to drive the various transitions requires considering the following four cases.

Case 1 – Placing and holding the controller in state 010 can be accomplished by applying the following values to PRE and CLR flip-flop inputs.

For $EV = 1$ and Reset = 1: $PRE2,CLR2 = 1,0$; $PRE1,CLR1 = 0,1$; $PRE0,CLR0 = 1,0$.

Case 2 – Normal state sequencing, $EV = 0$ and Reset = 1, requires $PRE2 = CLR2 = PRE1 = CLR1 = PRE0 = CLR0 = 1$.

Case 3 – Let's assume that Reset has precedence over EV, so Reset = 0 should always drive the controller to state 000 requiring $PRE2,CLR2 = 1,0$; $PRE1, CLR1 = 1,0$; $PRE0,CLR0 = 1,0$.

Case 4 – Returning to state 000 following an emergency cycle is triggered by the transition of EV from 1 to 0 while in state 010 and Reset = 1.

The following logic equations capture these requirements and can be realized using as shown in Fig. 5.63.

$$\begin{aligned} PRE2 &= PRE0 = 1 \quad (V_{CC}) \\ CLR2 &= CLR0 = EV' \bullet \text{Reset} \end{aligned}$$

$$\begin{aligned} PRE1 &= (EV \bullet \text{Reset})' \\ CLR1 &= (EV' \bullet y_2' \bullet y_1 \bullet y_0' + \text{Reset}')' \end{aligned}$$

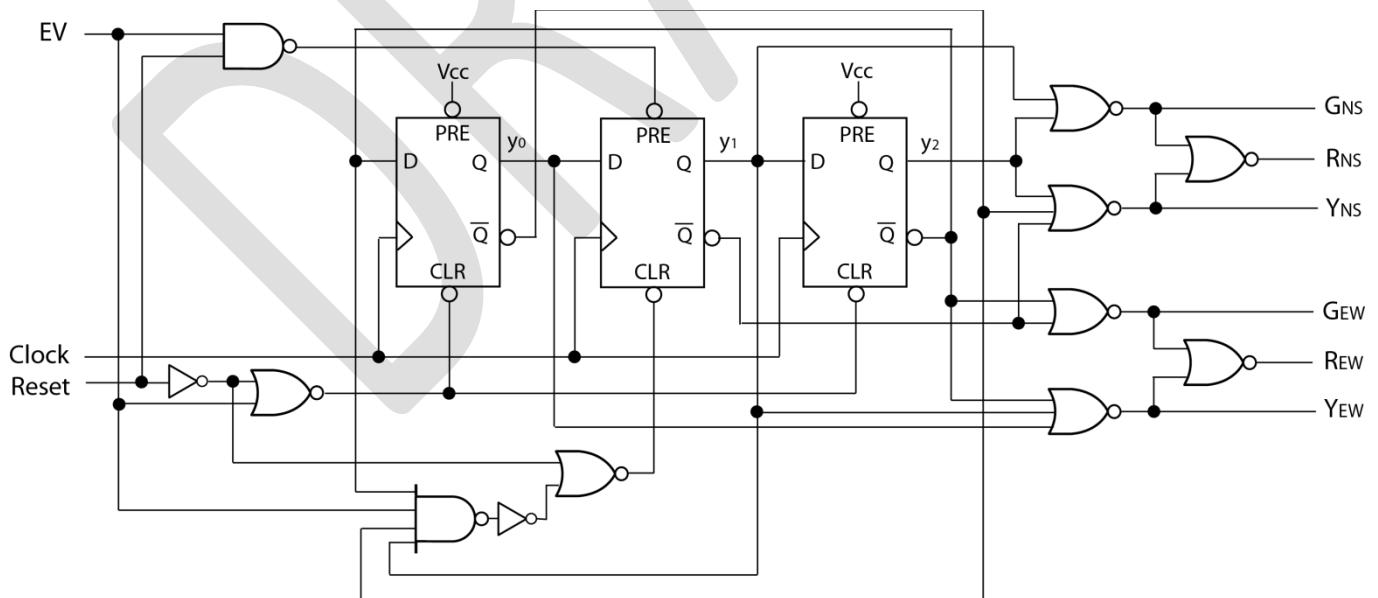


Figure 5.63: Realization of Traffic Light Controller with Emergency Vehicle Cycle.

Pedestrian Cycles – Traffic lights at intersections with significant pedestrian traffic often have push buttons that allow pedestrians to request a green light in their direction. Let's modify our basic controller to provide this feature. Figure 5.64a shows the block diagram of the controller with inputs P_{NS} and P_{EW} representing requests from pedestrians. Assume that north-south traffic has precedence over east-west should simultaneous requests be received. The approach used in this example is to either shorten or extend the current green time when pedestrian requests are received. For example if G_{NS} is on and P_{NS} is received, G_{NS} will be extended to three clock cycles rather than the normal two. If G_{NS} is on and P_{EW} is received, G_{NS} will be shortened to one clock cycle. The state diagram in Fig. 5.64b captures these requirements for both north-south and east-west traffic. A Verilog description of the controller is shown in Figure 5.65.

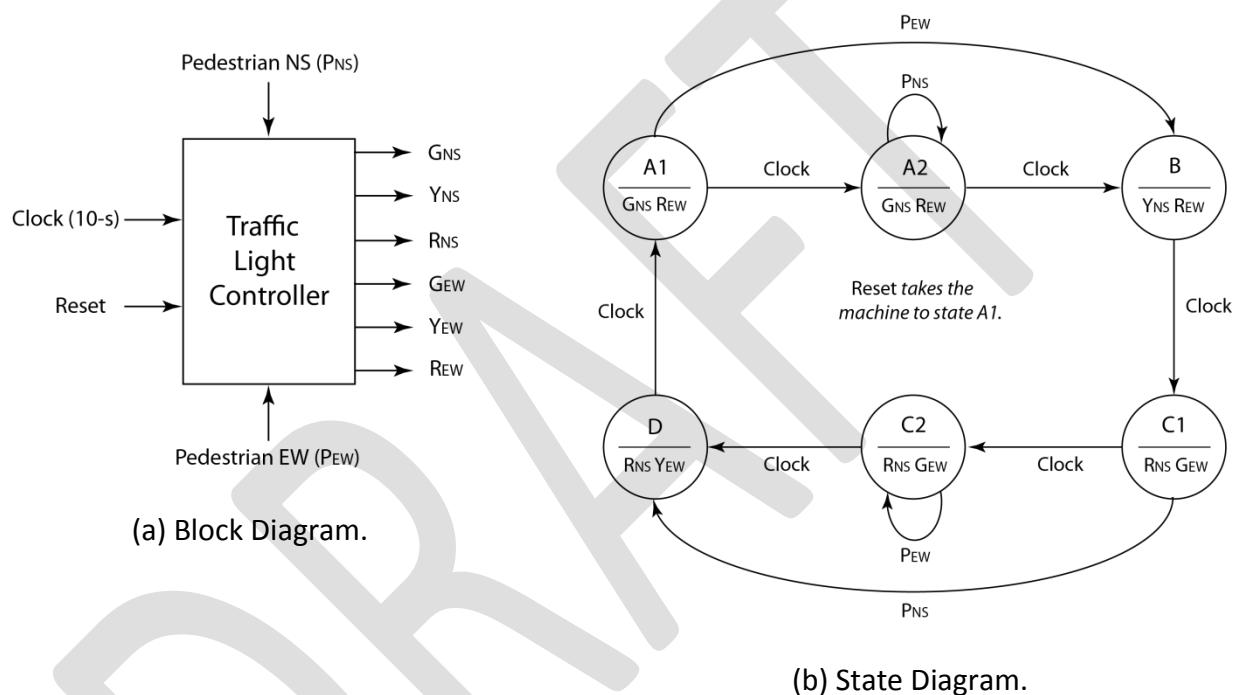


Figure 5.64: Traffic Light Controller with Pedestrian Cycles.

Traffic Light Controller with Left-Turn Cycles – Turn cycles are often provided in traffic light controllers. This design incorporates left-turn cycles into the basic controller designed previously. The controller provides protected left-turn cycles after each straight-through green cycle and accommodates an intersection of two two-lane roads with left turn lanes as shown in Fig. 5.66a. Such traffic patterns require left-turn lights that are controlled independently from the straight-through lights. This increases the number of control outputs but does change the inputs as shown in Fig. 5.66b. Four additional states are also required to provide for the left-turn cycles as shown in Figure 5.67. A Verilog model of the controller is shown in Figure 5.68.

```
//Verilog Behavioral Model of A Trafic Light Controller with Pedestrian Cycles
module TLCPedVerilog (Clock, Reset, Pns, Pew, Gns, Yns, Rns, Gew, Yew, Rew);
    input Clock, Reset, Pns, Pew;           //declare input variables
    output reg Gns, Yns, Rns, Gew, Yew, Rew; //declare output signals
    reg [2:0] state, nextstate;            //declare state and next state variables
    parameter A1 = 3'b0, A2 = 3'b1, B = 3'b11, C1 = 3'b111, C2 = 3'b110, D = 3'b100;
    always @ (posedge Clock, negedge Reset) //Detect input variable transitions
        if (Reset == 1'b0) state = A1;      //Take controller to state A1 on Reset
        else state = nextstate;           //Change state
    always @ (state)                      //Derive next state and outputs according to state diagram in Figure 17
        case (state)
            A1: begin if (Pew == 1'b1) nextstate=B; else nextstate=A2; Gns=1'b1; Yns=1'b0; Rns=1'b0; Gew=1'b0; Yew=1'b0; Rew=1'b1; end
            A2: begin if (Pns == 1'b1) nextstate=A2; else nextstate=B; Gns=1'b1; Yns=1'b0; Rns=1'b0; Gew=1'b0; Yew=1'b0; Rew=1'b1; end
            B: begin nextstate=C1; Gns=1'b0; Yns=1'b1; Rns=1'b0; Gew=1'b0; Yew=1'b0; Rew=1'b1; end
            C1: begin if (Pns == 1'b1) nextstate=D; else nextstate=C2; Gns=1'b0; Yns=1'b0; Rns=1'b1; Gew=1'b1; Yew=1'b0; Rew=1'b0; end
            C2: begin if (Pew == 1'b1) nextstate=C2; else nextstate=D; Gns=1'b0; Yns=1'b0; Rns=1'b1; Gew=1'b1; Yew=1'b0; Rew=1'b0; end
            D: begin nextstate=A1; Gns=1'b0; Yns=1'b0; Rns=1'b1; Gew=1'b0; Yew=1'b1; Rew=1'b0; end
            default: begin nextstate=A1; Gns=1'b0; Yns=1'b0; Rns=1'b1; Gew=1'b0; Yew=1'b0; Rew=1'b1; end
        endcase
endmodule
```

Figure 5.65: Verilog Model Traffic Light Controller with Pedestrian Cycles.

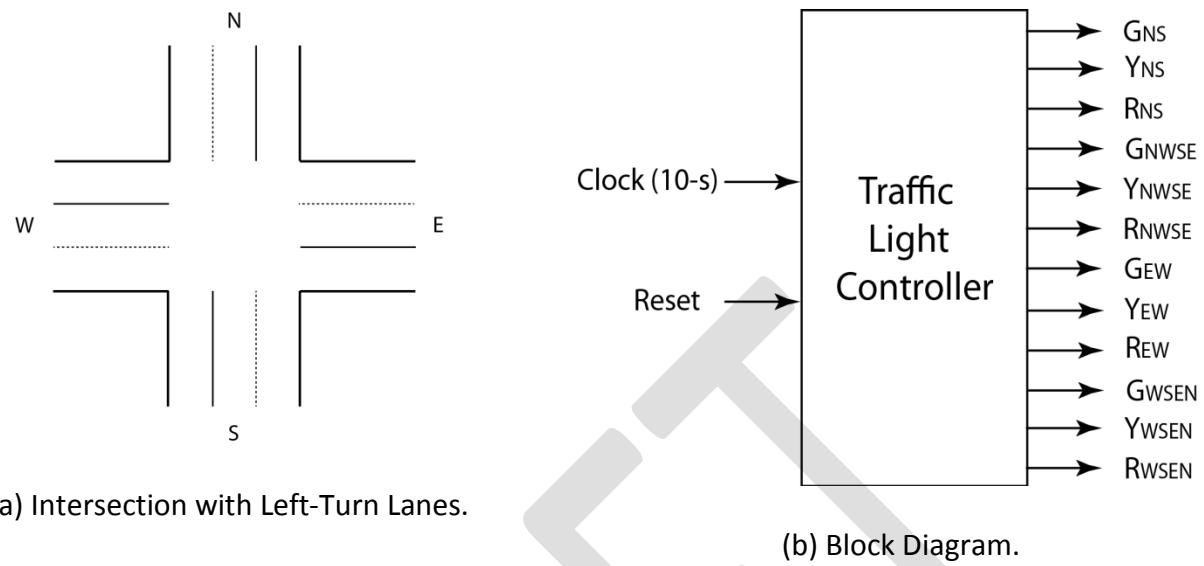


Figure 5.66: Traffic-Light Controller with Left-Turn Cycles.

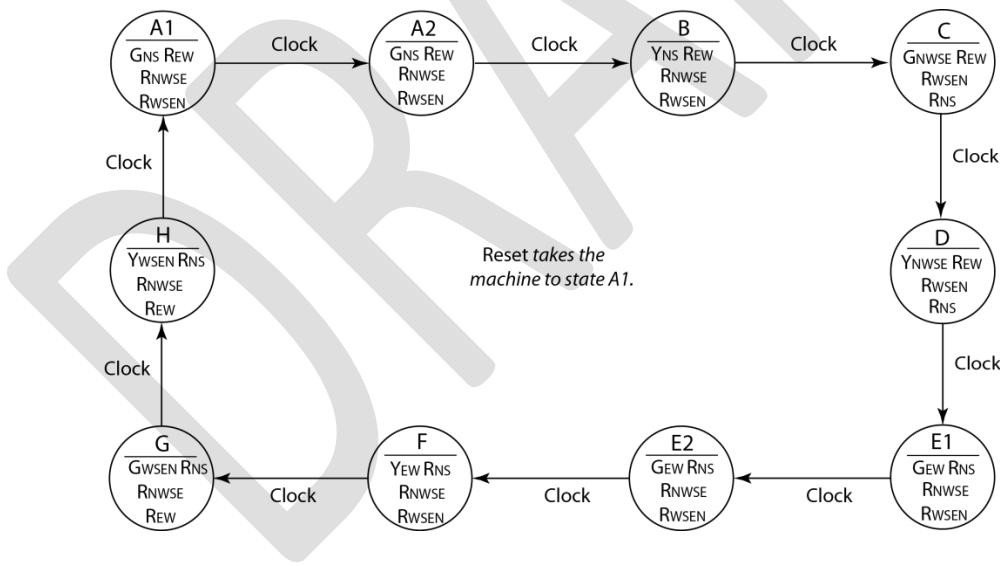


Figure 5.67: Traffic Light Controller with Left-Turn Cycles State Diagram.

```

//Verilog Behavioral Model of A Traffic Light Controller with Left-Turn Cycles
module TLCltVerilog (Clock,Reset,Gns,Yns,Rns,Gew,Yew,Rew,Gnwse,Ynwse,Rnwse,Gwsen,Ywsen,Rwsen);
    input Clock, Reset;                                //declare input variables
    output reg Gns,Yns,Rns,Gew,Yew,Rew,Gnwse,Ynwse,Rnwse,Gwsen,Ywsen,Rwsen; //declare output signals
    reg [4:0] state, nextstate;                         //declare state and next state variables
    parameter A1 = 5'b0, A2 = 5'b1, B = 5'b11, C = 5'b111, D = 5'b1111, E1 = 5'b11111, E2 = 5'b11110, F =
5'b11100, G = 5'b11000, H = 5'b10000;
    always @ (posedge Clock, negedge Reset)           //Detect input variable transitions
        if (Reset == 1'b0) state = A1;                //Take controller to state A1 on Reset
        else state = nextstate;                      //Change state
    always @ (state)                                 //Derive next state and outputs according to state diagram in Figure 5.67
        case (state)
            A1: begin nextstate = A2;
            Gns=1'b1;Yns=1'b0;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            A2: begin nextstate = B;
            Gns=1'b1;Yns=1'b0;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            B: begin nextstate = C;
            Gns=1'b0;Yns=1'b1;Rns=1'b0;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b0;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            C: begin nextstate = D;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b1;Ynwse=1'b0;Rnwse=1'b0;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            D: begin nextstate = E1;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b1;Rnwse=1'b0;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            E1: begin nextstate = E2;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b1;Yew=1'b0;Rew=1'b0;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            E2: begin nextstate = F;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b1;Yew=1'b0;Rew=1'b0;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            F: begin nextstate = G;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b1;Rew=1'b0;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
            G: begin nextstate = H;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b1;Ywsen=1'b0;Rwsen=1'b0; end
            H: begin nextstate = A1;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b1;Rwsen=1'b0; end
            default: begin nextstate = A1;
            Gns=1'b0;Yns=1'b0;Rns=1'b1;Gew=1'b0;Yew=1'b0;Rew=1'b1;Gnwse=1'b0;Ynwse=1'b0;Rnwse=1'b1;
            Gwsen=1'b0;Ywsen=1'b0;Rwsen=1'b1; end
        endcase
    endmodule

```

Figure 5.68: Verilog Model of the Traffic Light Controller with Left-Turn Cycles.

5.5 Summary and Review Questions

In this chapter we have presented procedures to analyze a synchronous sequential circuit, given its logic diagram, state table, or state diagram, and procedures to realize both completely and incompletely specified synchronous sequential logic circuits. Flip-flop input tables and the generation of excitation maps received attention, as did the development of state diagrams and state tables from verbal problem descriptions. Many design examples were completed to illustrate the synthesis techniques. The reader should now have a good grasp of the synchronous sequential circuit synthesis problem. For more detailed information on the synthesis problem, the reader is encouraged to pursue further reading from the selected references presented next.

- [1] ``A Method for Synthesizing Sequential Circuits," *Bell Sys. Tech. J.*, Vol. 34, September 1955, pp. 1045--1079.
- [2] ``Gedanken---Experiments on Sequential Machines," *Automata Studies, Annals of Mathematical Studies*, No. 34. Princeton, NJ: Princeton University Press, 1956, pp. 129--153.
- [3] ``The Synthesis of Sequential Switching Circuits," *J. Franklin Inst.*, Vol. 257, Nos. 3 and 4, March and April 1954, pp. 161--190, 275--303.
- [4] *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill Book Co., 1965.
- [5] *Switching and Finite Automata Theory*. New York: McGraw-Hill Book Co., 1970.
- [6] *Digital Networks and Computer Systems*. New York: Wiley, 1971.
- [7] *Digital Design Principles and Practices*, 2nd Ed., Chapter 2. Englewood Cliffs, NJ: Prentice Hall, 1994.
- [8] *Introduction to Digital Logic Design*, Reading, MA: Addison-Wesley, 1993.
- [9] *Contemporary Logic Design*, Menlo Park, CA: Benjamin/Cummings Publishing Co., 1994.
- [10] *Chip-level Modeling with VHDL*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- [11] Mentor Graphics Corp., 8005 S.W. Boeckman Rd., Wilsonville, OR 97070, April 1992.
- [12] *Automatic Logic Synthesis for Digital Systems*, New York: McGraw-Hill Book Co., 1992.

Review the following questions to test your understanding of the topics covered in this chapter.

1. Can you derive the state table and state diagram of a sequential logic circuit?

2. Can you derive the state table and state diagram from a Verilog and/or VHDL model of a sequential machine?
3. Can you derive the response, in timing diagram form, of a sequential circuit given the circuit diagram and a sequence of inputs?
4. Can you design a sequential circuit from a state diagram using D flip-flops for memory?
5. Can you design a sequential circuit from a state diagram using JK flip-flops for memory?
6. Can you write a Verilog and/or VHDL model of a state diagram?
7. Can you develop the state diagram of a finite state machine (FSM) that captures the logic requirements of a sequential circuit?
8. Do you know how many flip-flops are needed to realize a state diagram of N states?
9. Do you know how to find redundant states in a state diagram?
10. Do you understand how to partition a design problem into functional components?

5.6 Collaboration Exercises

These exercises are intended to be worked in class in groups of two or three students, with solutions presented to the remainder of the class. The exercises range from easy, to test basic understanding, to challenging, to promote deeper understanding and development of problem solving skills. The design-oriented exercise solutions may also be realized and demonstrated to the class if appropriate equipment and facilities are available.

1. The Moore model sequential circuit state diagram in Fig. E5.1 has state variables $y_1y_2y_3$, one input, x , and one output, z . Derive the state table corresponding to this state diagram.

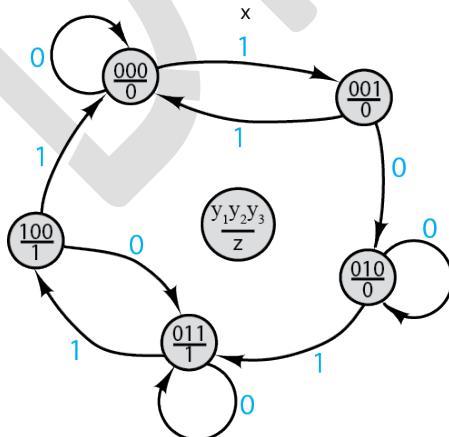


Figure E5.1

2. The initial state of the circuit in Fig. E5.1 is $y_1y_2y_3 = 000$.
- Give a general description of input sequences that will result in output z changing from 0 to 1, and input sequences that will result in output z changing back to 0.
 - List the sequence of states and outputs that will occur if input sequence 111001010110 is applied to input x.
 - Sketch a timing diagram, assuming that the state changes on the positive edge of a clock signal c/k. You may assume that the input changes in the middle of each clock period.
3. Write a Verilog (or VHDL) description of the state diagram in Fig. E5.1.
4. The Mealy model sequential circuit state diagram in Fig. E5.4 has state variables $y_1y_2y_3$, one input, x, and one output, z. Derive the state table corresponding to this state diagram.

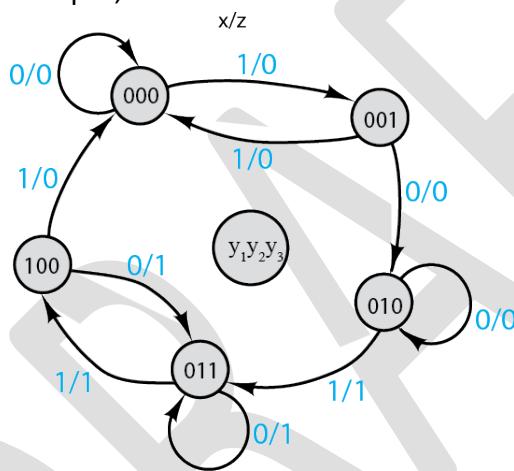


Figure E5.4

5. The initial state of the circuit in Fig. E5.4 is $y_1y_2y_3 = 000$.
- Give a general description of input sequences that will result in output z changing from 0 to 1, and input sequences that will result in output z changing back to 0.
 - List the sequence of states and outputs that will occur if input sequence 111001010110 is applied to input x.
 - Sketch a timing diagram, assuming that the state changes on the positive edge of a clock signal c/k. You may assume that the input changes in the middle of each clock period.
6. Write a Verilog (or VHDL) description of the state diagram in Fig. E5.4.

7. Given the synchronous sequential circuit diagram in Fig. E5.7:
- Find the state table and state diagram.
 - Draw a timing diagram for $x = 000101011$ and initial state $y = 0$.

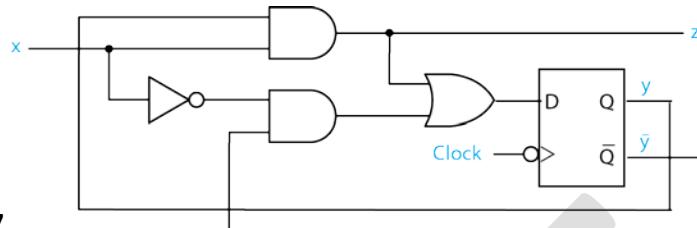


Figure E5.7

8. Analyze the synchronous sequential circuit in Fig. E5.8 by deriving its state table and state diagram, assuming that the following state assignment is used.

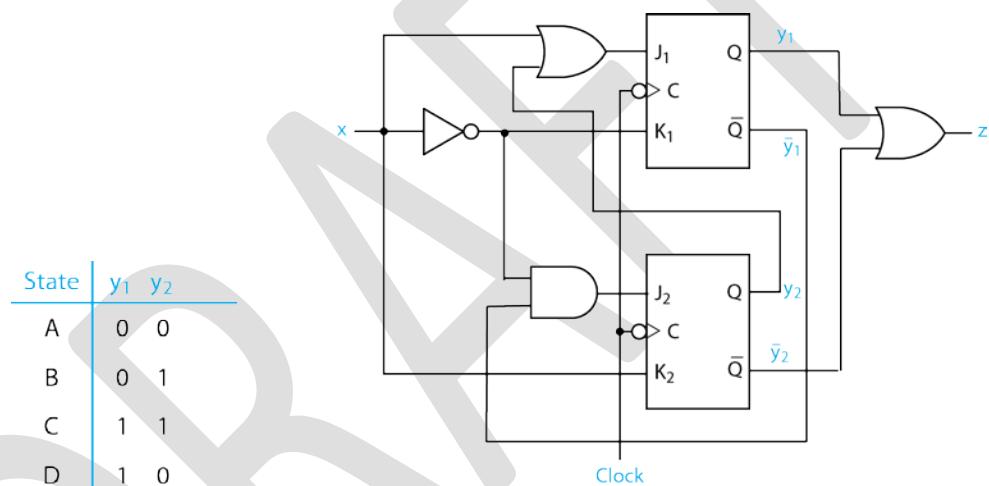


Figure E5.8

9. Find the state diagram for the sequential circuit of Fig. E5.9 using the state assignment and determine the response of the circuit to input sequence $x = 01001010$, assuming initial state $y_1 = y_2 = 0$,

State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

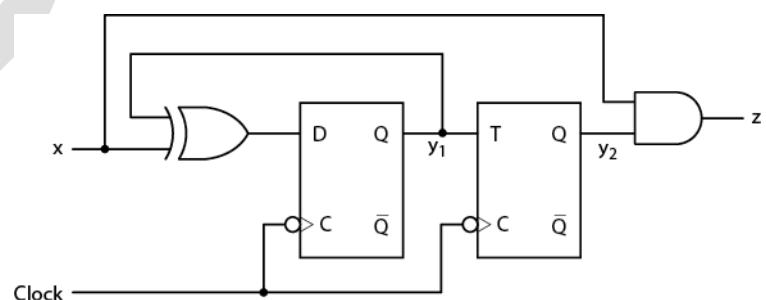


Figure E5.9

10. Derive the logic equations to implement the four-state sequential circuit defined by the following state table, using the indicated state assignment and:

- D flip-flops.
- JK flip-flops.
- T flip-flops.

State	y_1	y_2	x	0	1
A	0	0	A	B/0	C/0
B	0	1	B	D/0	A/1
C	1	1	C	A/1	D/0
D	1	0	D	D/1	B/1

11. Derive the logic equations to implement the six-state sequential circuit defined by the following state table, using the indicated state assignment and D flip-flops.

State	y_1	y_2	y_3	x	0	1
A	0	0	0	A	B/0	A/0
B	0	0	1	B	D/0	C/1
C	0	1	1	C	A/0	B/1
D	0	1	0	D	E/1	F/0
E	1	0	0	E	A/0	F/1
F	1	0	1	F	C/0	D/1

12. Derive the minimum state diagram of a clocked sequential circuit that recognizes the input sequence 1110. For example,

$$\begin{aligned} x &= 001011100111010 \\ z &= 000000010000100 \end{aligned}$$

13. Find the state table of a synchronous sequential circuit that detects the input sequence 110011. The sequences may overlap as follows:

$$\begin{aligned} x &= 0101100110011010 \\ z &= 0000000010001000 \end{aligned}$$

14. Design a modulo-5 counter, which counts from 0 to 4 and repeats. The counter is to have three outputs, reflecting the current count, and three inputs: CLK, R, and EN. R is an asynchronous reset signal; the counter is to reset to 0 whenever R = 1. The count should change on the rising edge of clock input CLK. EN is an enable signal: the count is to change if and only if EN = 1. Show all steps in the design process. You may use any flip-flop type.

15. Using an implication table, reduce the following sequential circuit to a minimum number of states:

	x	
	0	1
A	B/0	A/0
B	F/0	E/1
C	D/0	B/0
D	B/0	A/0
E	C/0	B/1
F	A/0	E/0
G	E/0	G/0

16. Reduce the number of states of the following sequential circuit using an implication table.

	J	K	M
A	C/0	D/1	C/0
B	B/0	D/1	B/0
C	B/0	E/1	A/0
D	A/1	F/0	F/0
E	B/1	F/0	F/0
F	A/1	E/1	D/1

5.7 Problems

5.1 For the synchronous sequential circuit of Fig. P5.1, find:

- The state table using K-maps and state assignment $A \equiv 0, B \equiv 1$.
- The state diagram if the circuit input is in pulse form.
- The timing diagram for an input sequence $x = 00100110$ and starting state $y = 1$.

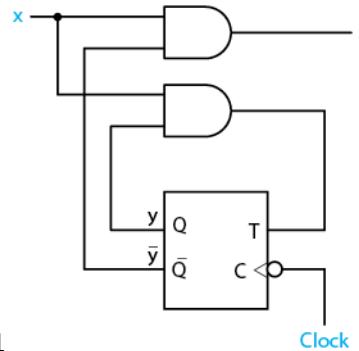


Figure P5.1

5.2 Given the synchronous sequential circuit of Fig. P5.2:

- Draw a timing diagram for $x = 000101011$ and initial state $y = 0$.
- Find the state diagram.
- Find the state table.

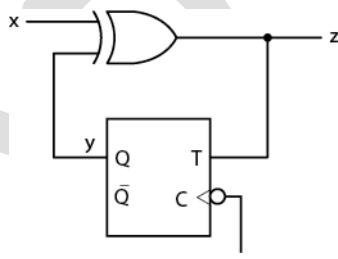
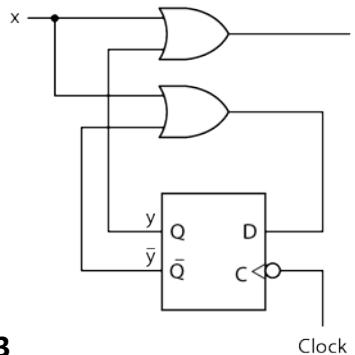


Figure P5.2

5.3 For the sequential circuit in Fig. P5.3, find:

- The state table ($A \equiv 0, B \equiv 1$).
- The state diagram.
- A timing diagram if the starting state is $y = 0$ and $x = 001011000$.

**Figure P5.3**

5.4 From the following switching functions,

$$z = T_1 = x\bar{y}_2$$

$$T_2 = x \oplus y_1$$

- (a) Draw the logic diagram for a synchronous sequential circuit using T flip-flops
- (b) Find a state diagram of the circuit using the assignment

State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

5.5 For a sequential circuit realizing the following logic equations:

$$Y_1 = \bar{x} \oplus y_1$$

$$Y_2 = x + y_1 + y_2$$

$$z = xy_1\bar{y}_2$$

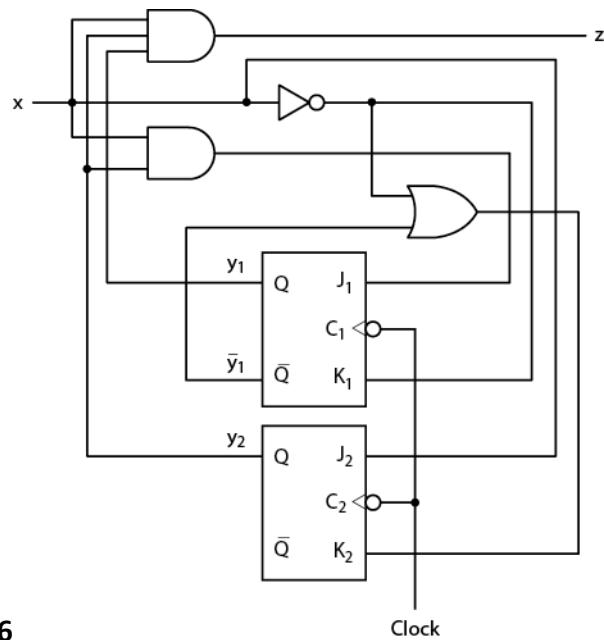
- (a) Draw the logic diagram for a D flip-flop implementation of the circuit.
- (b) Find a binary state table for this circuit.

5.6 Analyze the synchronous sequential circuit of Fig. P5.6. Assume the inputs are binary levels and that the following state assignment is used:

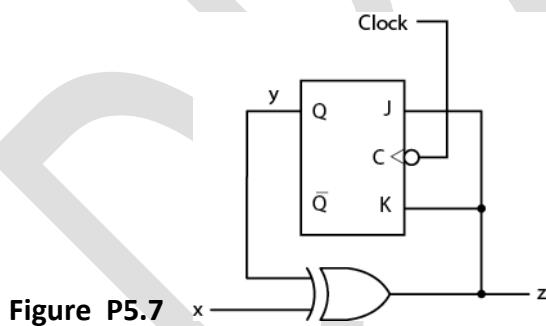
State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

Use K-maps to find:

- (a) The state table.
- (b) The state diagram.

**Figure P5.6**

5.7 If the sequential circuit of Fig. P5.7 yields an output sequence
 $z = 11011111$
when we apply the input sequence
 $x = 01101010$
what is the starting state?

**Figure P5.7**

5.8 Find the state table for the sequential circuit in Fig. P5.8.

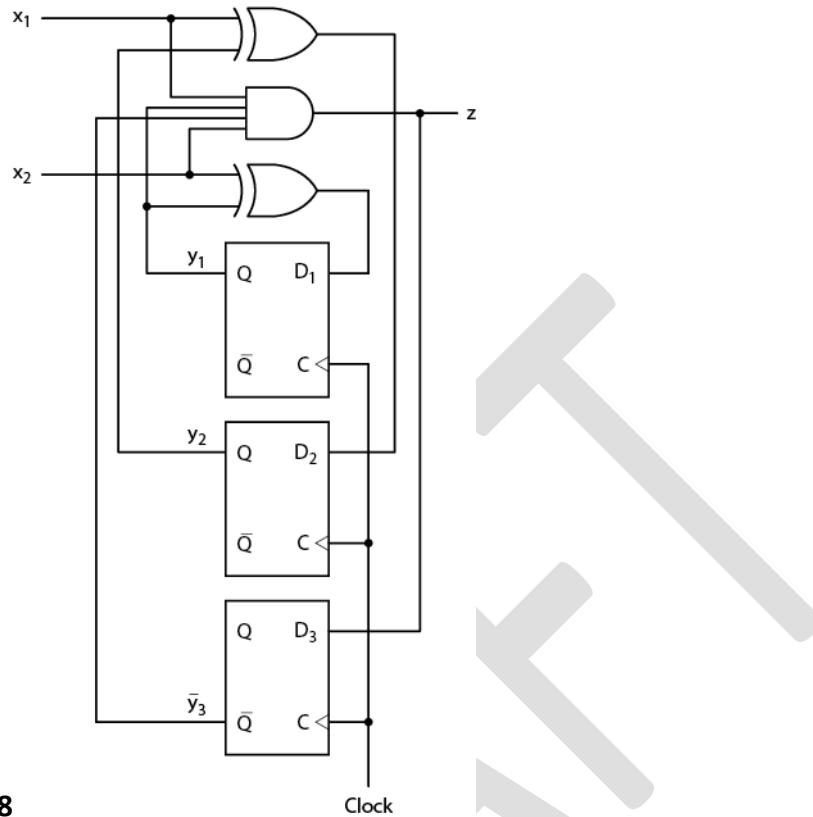


Figure P5.8

5.9 Consider a sequential circuit consisting of two cascaded circuits illustrated in Fig. P5.9. If the starting state is $y_1 = y_2 = 0$, what is the output sequence generated by the input sequence

$$x = 0110111010$$

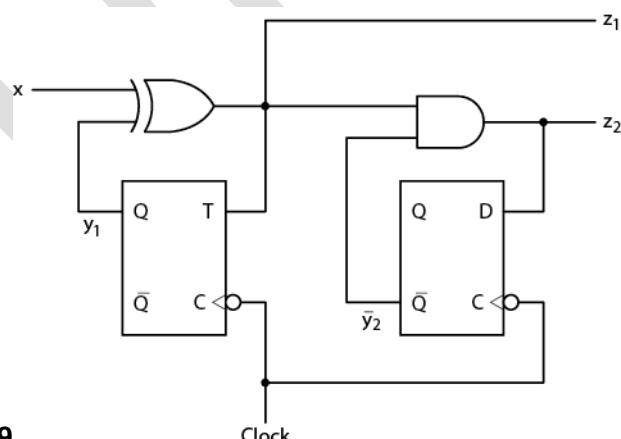


Figure P5.9

5.10 Find the state diagram for the sequential circuit of Fig. P5.10 using the state assignment

State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

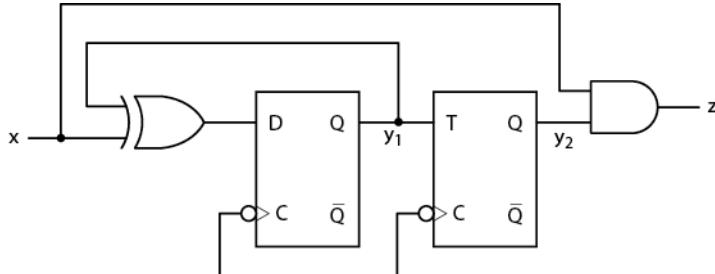


Figure P5.10 Clock

5.11 Find the D flip-flop implementation for the sequential circuit defined by the following state table. Use the state assignment listed. Draw the logic circuit diagram.

State	y_1	y_2	y_3
A	0	0	0
B	0	0	1
C	0	1	1
D	0	1	0
E	1	0	0
F	1	0	1

A	x	
	0	1
A	D/0	C/0
B	E/0	A/1
C	F/1	B/1
D	A/1	F/1
E	C/0	E/0
F	B/0	D/1

5.12 Obtain a D flip-flop realization for the synchronous sequential circuit specified by the following state table. Use the indicated state assignment. Write the combinational logic equations.

State	y_1	y_2	y_3
A	0	0	0
B	0	0	1
C	0	1	0
D	0	1	1
E	1	0	0

A	x	
	0	1
A	B/0	E/0
B	A/1	C/1
C	B/0	C/1
D	C/0	E/0
E	D/1	A/0

5.13 Determine four state diagrams for synchronous sequential circuits as specified by the following requirements. Each circuit has a single input line x and a single output line z .

- (a) The first circuit must produce an output $z = 1$ when two consecutive logic 1 inputs x have occurred. The next input after the two logic ones resets the output to logic 0. For example,

$$x = 01100111110$$

$$z = 00100010100$$

- (b) The second circuit must detect the input sequence 101 by producing $z=1$ as the last 1 occurs. The output z is reset to 0 on the next clock pulse. Two 101 sequences may overlap. For example,

$$x = 010101101$$

$$z = 000101001$$

- (c) Repeat Problem 4.44b but do not permit overlapping sequences. For example,

$$x = 010101101$$

$$z = 000100001$$

- (d) The fourth circuit detects a 01 sequence. The sequence sets $z=1$, which is reset only by a 00 input sequence. For all other cases, $z=0$. For example,

$$x = 010100100$$

$$z = 011110110$$

- 5.14 Derive the minimum state diagram of a clocked sequential circuit that recognizes the input sequence 1010. Sequences may overlap. For example,

$$x = 00101001010101110$$

$$z = 00000100001010000$$

- 5.15 Find the state table of a synchronous sequential circuit that detects the input sequence 0101. The sequences may overlap as follows:

$$x = 010101001101011$$

$$z = 000101000000010$$

- 5.16 Obtain a minimum state diagram for a clocked sequential circuit that recognizes the input sequence 1001 including overlap. For example:

$$x = 0101001000110010010$$

$$z = 00000010000000010010$$

- 5.17 Derive the logic equations to implement the four-state sequential circuit defined by the following state table, using the indicated state assignment and:

(a) D flip-flops.

(b) JK flip-flops.

(c) SR flip-flops.

State	y ₁ y ₂		x		
	0	1		A	B/0 C/0
A	0 0			B	D/0 A/1
B	0 1			C	A/1 D/0
C	1 1			D	D/1 B/1
D	1 0				

- 5.18 For the following circuit with the given state assignment, find a JK flip-flop implementation. Write the logic equations and sketch the logic diagram.

State	y_1	y_2	x	0	1	z
A	0	0				0
B	0	1				0
C	1	0				0
D	1	1				1

5.19 Implement the circuit of Problem 4.49 using T flip-flops.

5.20 Implement the circuit of Problem 4.49 using D flip-flops.

5.21 Given the following reduced state table and assignment, find the logic equations and logic diagram:

- (a) Using D flip-flops.
- (b) Using JK flip-flops.

State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

A	0	1	x
A	A/0	B/0	
B	C/0	B/0	
C	D/0	B/0	
D	A/1	B/0	

5.22 Find the logic diagram of an implementation of the following sequential circuit, given the state assignment and:

- (a) D flip-flops.
- (b) JK flip-flops.
- (c) T flip-flops.

State	y_1	y_2
A	0	0
B	0	1
C	1	1
D	1	0

A	0	1	x
A	A/0	B/0	
B	C/0	B/0	
C	B/0	B/0	
D	D/1	A/0	

5.23 Find a JK flip-flop realization for the following reduced state table and assignment:

State	y_1	y_2	y_3
A	0	0	0
B	1	0	1
C	1	0	0
D	0	0	1
E	0	1	0
F	1	1	0

A	0	1	x
A	B/0	D/0	
B	A/0	C/1	
C	D/1	C/0	
D	B/1	E/1	
E	C/0	A/0	
F	E/0	F/1	

- 5.24 Design a 2-bit up/down, modulo-3 counter with the following function table using JK flip-flops.

$s_1 \ s_0$	Mode
0 0	Up
0 1	Down
1 0	Modulo 3
1 1	Modulo 3

- 5.25 Use D flip-flops to design a 3-bit counter/pseudorandom number generator. The circuit has one control input x . When $x = 0$, the circuit should operate as a binary up-counter. Otherwise, it should operate as a pseudorandom number generator according to the following function table.

Present State	Binary Up-Counter $x = 0$	Pseudo-Random Number Gen. $x = 1$
	Next State	Next State
0	1	0
1	2	4
2	3	5
3	4	1
4	5	2
5	6	6
6	7	7
7	0	3

- 5.26 Design a serial subtractor that will perform the operation $A - B$, where $A = a_{n-1} \dots a_1 a_0$ and $B = b_{n-1} \dots b_1 b_0$. The operands are applied to the serial subtractor sequentially, beginning with bits a_0 and b_0 . Use JK flip-flops.
- 5.27 Design a serial parity generation circuit. The circuit receives a sequence of bits and determines whether the sequence contains an even or odd number of ones. The circuit output p should be 0 for even parity, that is, if the sequence contains an even number of ones, and 1 for odd parity.
- 5.28 Design a logic circuit to implement the candy machine control unit designed in Figure 5.36. Use JK flip-flops.
- 5.29 Design a logic circuit for the binary multiplier control unit whose ASM diagram was designed in Figure 5.38 using a minimum number of JK flip-flops.

5.30 Modify the binary multiplier design of Figure 5.38 so that it will perform a binary division operation, dividing an 8-bit dividend by a 4-bit divisor using a sequence of subtract and shift operations. The dividend should initially be loaded into the A and Q registers and the divisor placed in the M register. At the end of the algorithm, the quotient should be in the Q register and the remainder in the A register.

5.31 Find a minimized state table for the following synchronous sequential circuit by

- (a) Inspection.
- (b) Implication table.

	I	J
A	B/0	A/1
B	C/0	A/0
C	C/0	B/0
D	E/0	D/1
E	C/0	D/0

5.32 Reduce the following state tables by inspection:

	I	J
A	B/1	C/0
B	A/1	C/0
C	D/1	A/0
D	C/1	A/1

	I	J
A	A/0	E/1
B	E/1	C/0
C	A/1	D/1
D	F/0	G/1
E	B/1	C/0
F	F/0	E/1
G	A/1	D/1

	I	J	K
A	A/0	B/1	E/1
B	B/0	A/1	F/1
C	A/1	D/0	E/0
D	F/0	C/1	A/0
E	A/0	D/1	E/1
F	B/0	D/1	F/1

5.33 Reduce the state tables of Problem 5.32 using implication tables.

5.34 Find a reduced state table for the following synchronous sequential circuit:

	<i>I</i>	<i>J</i>
<i>A</i>	<i>B</i> /0	<i>C</i> /0
<i>B</i>	<i>D</i> /0	<i>E</i> /0
<i>C</i>	<i>F</i> /0	<i>G</i> /0
<i>D</i>	<i>A</i> /1	<i>B</i> /1
<i>E</i>	<i>C</i> /0	<i>D</i> /0
<i>F</i>	<i>F</i> /0	<i>G</i> /0
<i>G</i>	<i>B</i> /0	<i>F</i> /0

5.35 Using an implication table, reduce the following sequential circuit to a minimum number of states:

	<i>I</i>	<i>J</i>
<i>A</i>	<i>A</i> /0	<i>C</i> /0
<i>B</i>	<i>D</i> /1	<i>A</i> /0
<i>C</i>	<i>F</i> /0	<i>F</i> /0
<i>D</i>	<i>E</i> /1	<i>B</i> /0
<i>E</i>	<i>G</i> /1	<i>G</i> /0
<i>F</i>	<i>C</i> /0	<i>C</i> /0
<i>G</i>	<i>B</i> /1	<i>H</i> /0
<i>H</i>	<i>H</i> /0	<i>C</i> /0

5.36 Reduce the number of states of the following sequential circuit using an implication table.

	<i>I</i>	<i>J</i>	<i>K</i>
<i>A</i>	<i>D</i> /1	<i>C</i> /0	<i>E</i> /1
<i>B</i>	<i>D</i> /0	<i>E</i> /0	<i>C</i> /1
<i>C</i>	<i>A</i> /0	<i>E</i> /0	<i>B</i> /1
<i>D</i>	<i>A</i> /1	<i>B</i> /0	<i>E</i> /1
<i>E</i>	<i>A</i> /1	<i>C</i> /0	<i>B</i> /1

5.37 Find a D flip-flop realization for the following sequential circuit using each of the three unique assignments for four-state circuits:

	<i>x</i>	
	0	1
<i>A</i>	<i>B</i> /0	<i>D</i> /0
<i>B</i>	<i>C</i> /0	<i>A</i> /0
<i>C</i>	<i>D</i> /0	<i>A</i> /0
<i>D</i>	<i>B</i> /1	<i>C</i> /1

5.38 For the following state table, find circuit implementations with each of the three unique state assignments for four-state circuits and memory elements of:

- (a) T flip-flops.
- (b) JK flip-flops.
- (c) SR flip-flops.

	x	
	0	1
A	C/0	D/0
B	C/0	A/0
C	B/0	D/0
D	A/1	B/1

5.39 Derive the logic equations to implement the four-state sequential circuit defined by the following state table, using each of the three unique state assignments and:

- (a) D flip-flops.
- (b) JK flip-flops.
- (c) SR flip-flops.
- (d) T flip-flops.

	x	
	0	1
A	B/0	C/0
B	D/0	A/1
C	A/1	D/0
D	D/1	B/1