VHDL

DESIGN, SYNTHESIS, AND SIMULATION

Debaprasad Das
Professor and Head
Department of Electronics and Communication Engineering
Assam University
Silchar

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trademark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
Ground Floor, 2/11, Ansari Road, Daryaganj, New Delhi 110002, India

First published in 2018

ISBN-13: 978-0-19-809329-9
ISBN-10: 0-19-809329-2

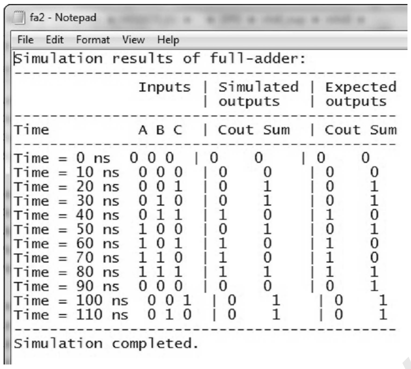
Typeset in AGaramondPro-Regular
by MacroTex Solutions, Chennai
Printed in India by Magic International (P) Ltd., Greater Noida

Cover image: VLADGRIN / Shutterstock

Features of

Example 11.8. Write a VHDL testbench program to read the input bit patterns from a file "fa2.vec" and generate the input waveforms, simulate the design and write the simulated input/outputs into another file "fa2.out".

```
library ieee, std;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;
```



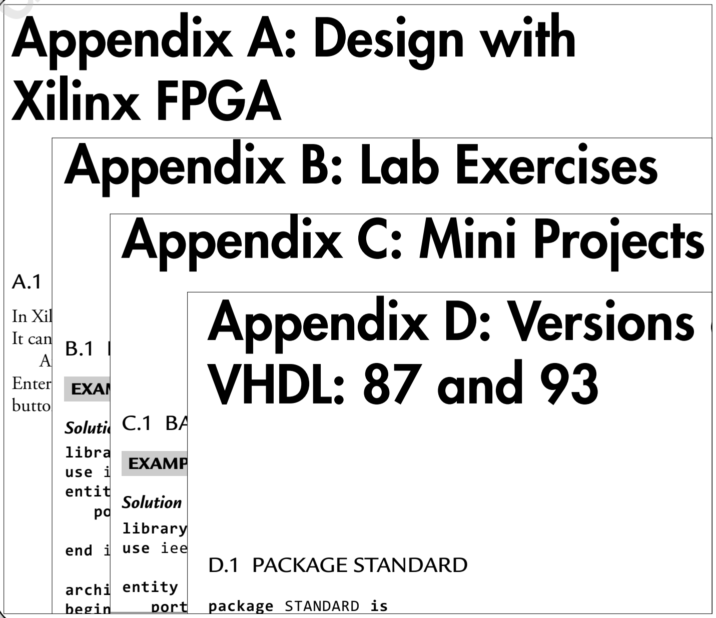
```
Simulation results of full-adder:
---------------------------------------------------
                 Inputs | Simulated | Expected
                        | outputs   | outputs
---------------------------------------------------
Time              A B C  | Cout Sum  | Cout Sum
---------------------------------------------------
Time = 0 ns   0 0 0  | 0    0    | 0    0
Time = 10 ns   0 0 0  | 0    0    | 0    0
Time = 20 ns   0 0 1  | 0    1    | 0    1
Time = 30 ns   0 1 0  | 0    1    | 0    1
Time = 40 ns   0 1 1  | 1    0    | 1    0
Time = 50 ns   1 0 0  | 0    1    | 0    1
Time = 60 ns   1 0 1  | 1    0    | 1    0
Time = 70 ns   1 1 0  | 1    0    | 1    0
Time = 80 ns   1 1 1  | 1    1    | 1    1
Time = 90 ns   0 0 0  | 0    0    | 0    1
Time = 100 ns   0 0 1  | 0    1    | 0    1
Time = 110 ns   0 1 0  | 0    1    | 0    1
---------------------------------------------------
Simulation completed.
```

FIGURE 11.10 Content of fa2.out file

Solved Examples
The book comprises various solved examples to provide readers with a deep understanding of the concepts discussed in the chapters.

Appendices
Appendices A, B, C, and D are provided at the end of the book for readers to gain additional knowledge. Appendix A discusses design with Xilinx FPGA, B covers lab exercises, C consists of some mini projects, and D includes versions of VHDL-87 and 93.

Material based on or adapted from figures and text owned by Xilinx, Inc., courtesy of Xilinx, Inc. © Copyright Xilinx [2018].



Appendix A: Design with Xilinx FPGA

A.1

In Xil
It can
A
Enter
butto

Appendix B: Lab Exercises

B.1

EXAN

Soluti
libra
use i
entit
po
end i
archi
begin

Appendix C: Mini Projects

C.1 BA

EXAMP

Solution
library
use iee
entity
port

Appendix D: Versions VHDL: 87 and 93

D.1 PACKAGE STANDARD

package STANDARD is

the Book

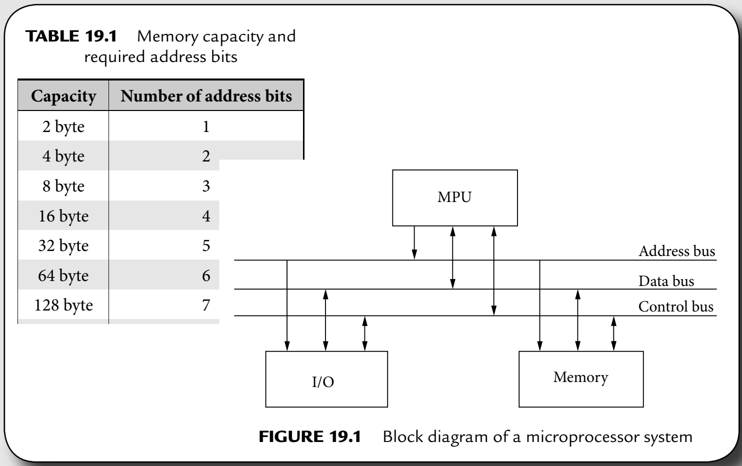


TABLE 19.1 Memory capacity and required address bits

Capacity	Number of address bits
2 byte	1
4 byte	2
8 byte	3
16 byte	4
32 byte	5
64 byte	6
128 byte	7

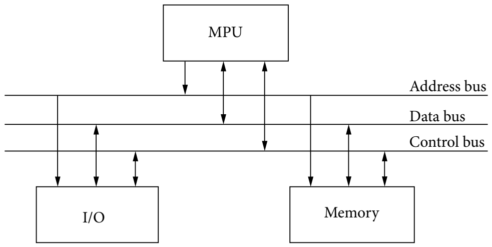
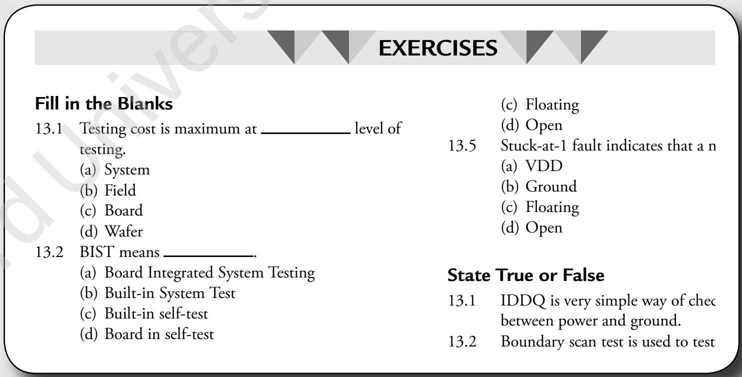


FIGURE 19.1 Block diagram of a microprocessor system

Illustrations
Each chapter includes figures and tables supporting text for readers to get better understanding of the concepts.

Chapter-end Exercises
The book covers enough chapter-end exercises, such as fill in the blanks, multiple choice questions, true or false, short-answer type questions, and long-answer type questions for readers to test their knowledge.



EXERCISES

Fill in the Blanks

13.1 Testing cost is maximum at ________ level of testing.
(a) System
(b) Field
(c) Board
(d) Wafer

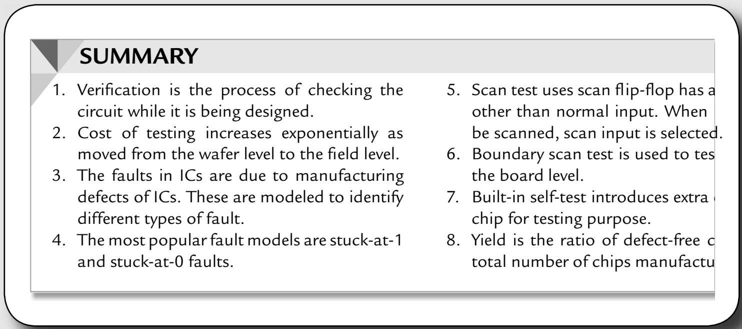
13.2 BIST means ________.
(a) Board Integrated System Testing
(b) Built-in System Test
(c) Built-in self-test
(d) Board in self-test

(c) Floating
(d) Open

13.5 Stuck-at-1 fault indicates that a n
(a) VDD
(b) Ground
(c) Floating
(d) Open

State True or False

13.1 IDDQ is very simple way of chec between power and ground.
13.2 Boundary scan test is used to test



SUMMARY

1. Verification is the process of checking the circuit while it is being designed.
2. Cost of testing increases exponentially as moved from the wafer level to the field level.
3. The faults in ICs are due to manufacturing defects of ICs. These are modeled to identify different types of fault.
4. The most popular fault models are stuck-at-1 and stuck-at-0 faults.
5. Scan test uses scan flip-flop has a other than normal input. When be scanned, scan input is selected.
6. Boundary scan test is used to tes the board level.
7. Built-in self-test introduces extra chip for testing purpose.
8. Yield is the ratio of defect-free c total number of chips manufactu

Summary
The chapter-end summary presents all the important concepts explained in the chapter in the form of points. This helps to provide a quick grasp of concepts learnt in each chapter.

Preface

VHDL is a hardware description language that is extensively used for design, synthesis, and simulation of digital logic circuits. The language is different from other programming languages in that it supports the most important feature of concurrent statements. In normal programming languages, the statements are executed sequentially, however, in VHDL the statements can be executed concurrently. VHDL is very popular in both academics as well as industry.

VHDL is a case insensitive language. It has a syntax and certain keywords similar to other programming languages, and can support different data types and data objects. It also supports different modelling styles—dataflow, behavioural, structural, and mixed. These modelling styles are provided in order to support the different levels of hardware abstraction.

I have always thought that there should be a book on VHDL in which the purpose of the book will not be just to learn the language but to learn how to design using VHDL. Therefore, I have designed the book from the perspective of digital design techniques.

After going through the basics of writing codes in VHDL, students will be able to design efficient programs too since the book has a lot of programming examples and exercises. The students will also develop a basic understanding about digital logic circuits and the concepts of hardware design.

ABOUT THE BOOK

VHDL: Design, Synthesis, and Simulation is designed as a textbook for undergraduate students of electronics and communication engineering, computer science and engineering, and information technology, as well as postgraduate students of computer applications. It will also help the students of allied engineering disciplines. The objective of the book is to introduce the concepts of digital logic design, and then help students apply these concepts in VHDL programs and develop applications for real world problems. Starting with basic logic gates, the book explains how to design logic circuits up to the complexity of a central processing unit. This will help students to get used to the techniques and applications of VHDL. After discussing digital logic circuits, the book then elucidates the basics of hardware description language. It is also useful as a reference and resource to professional hardware designers working on VHDL and other similar languages.

It explains the primary and basic constructs, data objects, data types, operators, various modelling styles and different statements, functions, procedures, attributes, and configurations in subsequent chapters. Different modelling styles are presented with the concepts on how to choose and when to use which modelling style. Different types of statements are discussed with suitable examples so that the students understand the applicability of the statements for different applications. Delay modelling is presented for gate delay and interconnect delay. Verification of logic circuits and testing methodologies are discussed. Numerous design examples including memory, finite state machines, arithmetic logic unit (ALU), and microcontroller are provided to explain the design concepts behind developing systems with higher complexity. Design with programmable logic devices, field programmable gate array, and complex programmable logic device are included so that students can understand the concepts of system level realization. Writing testbench programs is explained in a separate chapter with a variety of approaches for developing testbench programs for testing the digital logic circuits. A separate chapter on Verilog HDL is provided at the end to help students port already written VHDL programs for a system into Verilog programs, if the situation demands so.

In order to further improve the understanding of the subject, numerous objective and subjective type questions, and programming exercises are provided at the end of each chapter.

KEY FEATURES

- *Simple and lucid explanations* for basic concepts of digital logic design using illustrations and examples for easy understanding
- A lot of compiled and tested *programs along with their outputs* to help students improve their programming skills
- *Case studies* within the text to demonstrate the implementation of the concepts learnt in various chapters
- Numerous *chapter-end exercises* including fill in the blanks, true/false, multiple choice questions with answers, short-answer type questions, and long-answer type questions for self-check and practice
- Point-wise *summary* at the end of each chapter and *Glossary* of key terms at the end of the book to help students quickly revise the important concepts

ORGANIZATION OF THE BOOK

The book is divided into 21 chapters and 4 appendices.

Chapter 1 provides an introduction to digital logic design. For a brief recapitulation of the students, the Boolean algebra, basic logic gates, and different combinational and sequential logic circuits are discussed. After that the finite state machines, memory, and control logic are discussed. Finally, the basics of algorithmic state machine is also explained in the chapter.

Chapter 2 is the introduction to VHDL language. It starts with the historical background of development of VHDL followed by basic language syntax. Then it presents data objects, data types, and operators supported by the language. Different hardware modelling styles are discussed next. Various statements such as concurrent statements, sequential statements, different control and looping statements, signal and variable assignments, block statement, etc. are discussed in this chapter.

Chapter 3 deals with the dataflow modelling of the digital logic circuits. After introduction, it presents dataflow modelling of basic logic gates, followed by different combinational logic circuits. It also presents dataflow modelling using block statements. Finally, dataflow modelling of multiplier and divider is presented.

Chapter 4 deals with behavioural modelling. The concept of behavioural modelling with the help of sequential statements is presented in this chapter. Starting with combinational logic circuits, the sequential logic circuits such as shift registers, counters, and memory design using behavioural modelling is explained with suitable examples. The chapter ends with the design of ALU and traffic light controller example.

Chapter 5 presents the structural modelling of digital logic circuits. It discusses the hierarchical description using component declaration and instantiation. Logic circuits at a higher level such as adder, subtractor, multiplier, shift register, barrel shifter are discussed. Design of bigger size multiplexer/decoder using smaller size multiplexer/decoder is presented at the end.

Chapter 6 discusses the mixed modelling style which is a combination of dataflow, behavioural, and structural modelling styles. Several examples of mixed modelling styles are presented to explain the necessity of such a style.

Chapter 7 deals with concurrent statements. It presents block statement, process statement, procedural call statement, signal assignment statement, assertion statement, component instantiation statement, and generate statement with suitable examples.

Chapter 8 deals with sequential statements. It presents wait statement, report statement, different types of signal assignment statements, and different types of variable assignment statements, if, case, loop, exit, next, return, and null statements.

Chapter 9 introduces the advanced features of VHDL. It discusses several attributes, group, configuration, subprogram, procedure, operator overloading, alias declaration, signatures, guarded signal, and qualified expression.

Chapter 10 presents the design of ALU. It first illustrates the concept with the help of 1-bit ALU, then it discusses the behavioural design of 16-bit ALU followed by structural design of ALU. Each of the individual blocks is designed first and finally, the blocks are integrated to design the ALU.

Chapter 11 elucidates on model simulation. After all these chapters, the students must learn how to verify the functionality of the design. It introduces the concept of testbench and then generation of input stimulus using different schemes. It also presents how to check the outputs. Different types of testbench are presented with examples. Finally, it discusses event driven simulation and cycle based simulation techniques.

Chapter 12 introduces delay modelling in VHDL. It explains the two kinds of delay in digital circuits- delay associated with logic gates called delay, and delay associated with interconnects/wires called interconnect delay. The chapter presents how effectively these two kinds of delay can be modelled in VHDL. Finally, the setup and hold time checks of a sequential element are also discussed.

Chapter 13 introduces the concepts of verification and testing of digital logic circuits. Several verification methods such as simulation, formal verification, and static timing analysis have been discussed. Next the testing of logic circuits, fault models, different testing techniques, and algorithms are presented in this chapter.

Chapter 14 is about synthesis of digital logic circuits. It is basically to explain the process of implementing the design on a FPGA chip. The examples of different synthesis illustrate the idea behind the allocation of resources available in FPGA system. Different constraints used for synthesis are also discussed.

Chapter 15 presents placement and routing techniques. When design is synthesized, it is then mapped to the available logic blocks and then routing is done to make the interconnections between the logic blocks. Different placement and routing algorithms are explained in this chapter.

Chapter 16 elucidates the file handling features of VHDL. It explains how to declare a file, and open and close a file. The reading and writing procedures are explained with suitable examples. File handling capability is mainly useful for writing testbench programs.

Chapter 17 is all about floating point arithmetic. It explains different number system representations and then how to represent floating point numbers. The operations such as addition and multiplication with floating point numbers are discussed with suitable examples.

Chapter 18 discusses the basics of programmable logic devices such as PLD, SPLD, CPLD, and FPGA. It explains the architecture of basic FPGA and CPLD devices. Finally, it describes the process of FPGA based digital design.

Chapter 19 discusses the modelling of memory and bus. Different types of memories such as ROM, RAM, SRAM, DRAM, and Dual port RAM are discussed. The concept of buses and modelling of buses is discussed with suitable examples.

Chapter 20 presents several design examples. It starts with different types of multiplier, followed by divider, then memory, and state machines. The concept of microprogramming, UART design, design of microcontroller, and finally a vending machine are explained with VHDL programs.

Chapter 21 is the last chapter of the book and explains Verilog. Verilog is another hardware description language (HDL). To give the students a flavour of the language, basic features of Verilog, different types of modelling and statements, and a few examples are provided.

The four appendices namely, ***Appendix A*** discusses the design with Xilinx FPGA, ***Appendix B*** includes some laboratory exercises, ***Appendix C*** provides a few mini projects, and ***Appendix D*** covers versions of VHDL: 87 and 93.

ONLINE RESOURCES

To help teachers and students, the book is accompanied with the following online resources that are available at http://oupinheonline.com/book/das-VHDL/9780198093299:

For Faculty

- Chapter-wise PPTs
- Chapter-wise solutions for select problems

For Students

- Some important codes from different chapters

ACKNOWLEDGMENTS

I am blessed to have great teachers during my undergraduate and postgraduate studies from Contai P. K. College, Institute of Radiophysics and Electronics, Jadavpur University, and Indian Institute of Engineering Science and Technology (formerly Bengal Engineering and Science University), Shibpur, who have always been a great inspiration in my life. I am also fortunate to have had great colleagues while working in Texas Instruments, Bangalore and Meghnad Saha Institute of Technology, Kolkata, who were always supportive of my decisions. It is my pleasure to express my heartfelt indebtedness to all of them.

The writing of this textbook was a challenging task for which a lot of help was required from many people. Fortunately, I have had the fine support of my family, friends, and fellow members of the teaching and non-teaching staff at the Triguna Sen School of Technology, Assam University, Silchar.

I pay my tribute to my parents, Smt. Pratima Das and (Late) Sri Birendranath Das, who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my wife, Joyita, and my loving daughters Adrija and Adrisha, who have been very patient and cooperative in letting me realize my dreams. My sincere thanks to all my dear students, friends, and all well-wishers, for their inspiration in writing this book.

Last but not the least, I thank the editorial team at Oxford University Press, India for their help and support.

Your comments and suggestions for the improvement of the book are welcome. You can connect with me at dasdebaprasad@yahoo.co.in.

Debaprasad Das

Brief Contents

Detailed Contents

Introduction to Digital Logic Design

1

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Boolean algebra
- Basic logic gates
- Combinational logic circuits
- Sequential logic circuits
- Finite state machine
- Memory
- Control logic circuits
- Algorithmic state machine

1.1 INTRODUCTION

In early ages, human beings used to live in caves, kill animals, and eat raw flesh. With time, they learned how to light fire and make weapons with stones. This age is termed as Stone Age. After the Stone Age, they started learning the use of metals. They made weapons and ornaments with metals, such as Copper and Bronze (alloy of Copper and Tin). This age is termed as Bronze Age. Then they found a harder metal called Iron. The Iron Age was dominated after the Bronze Age. But now it is the Age of Semiconductor. During the nineteenth and twentieth centuries, there has been tremendous growth in electronics and communication technology. It first started with Sir J. C. Bose when he invented first semiconductor detector or Galena detector during the year 1894–1898. Today, we live in the age of modern electronics and communication. We are fully surrounded by electronic gadgets. Starting from the cell phones, camera, watch, toys to personnel computers, all work using the digital technology. Although all practical signals are analog in nature, they are converted to digital because of many advantages of digital technology.

The main purpose of this book is to provide key concepts behind the digital circuit design and learn how to design a complete digital system with the help of formal hardware description language (HDL). Digital circuits are often called logic circuits because of their principle of working is binary in nature. When a large number of such logic circuits are manufactured on a single piece of semiconductor, we get an integrated circuit (IC). The ICs that are used in various electronic gadgets contain huge number of such logic circuits. The technology that is used to manufacture such a large integrated circuit is very-large-scale integration (VLSI) technology. Figure 1.1 illustrates the VLSI design flow where the initial few steps are elaborated in detail.

The very first step is to describe the system from an idea or a concept of end product. This is followed by the architecture or functional design of the system. Then the logic design takes place, which implements the whole big system by using basic logic gates. Once the logical design is finalized, the physical design step starts in which all the circuit components are designed physically and connected by wires within a given chip area. This is followed by the generation of layout of the chip which is nothing but the description of geometrical drawing of the chip. The next step is to generate mask, which can be thought like a negative of photograph. The mask is then taken

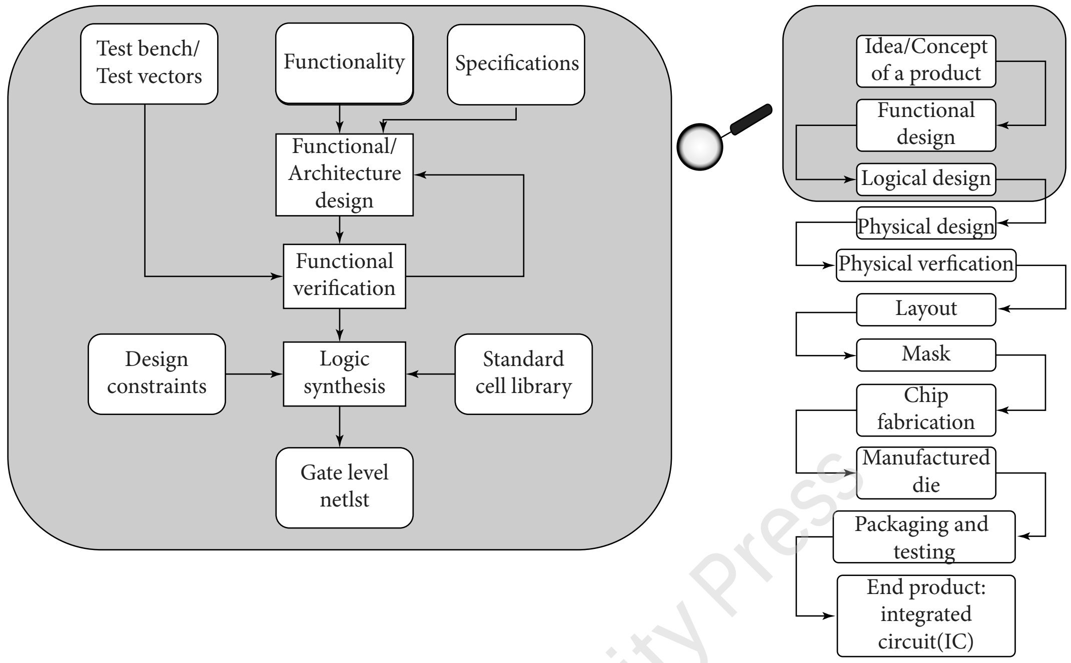


FIGURE 1.1 Complete process of making integrated circuit

to the manufacturing unit where wafers are processed to create patterns on them, and the dies are manufactured. The dies are tested and packaged, and the chips are made ready for use. These entire set of steps is very much complex and exhaustive. To know more about the steps, you can refer any text book on VLSI design [Das15]. This paragraph just gives some basic idea about the process of chip design. The important aspect of the diagram is to emphasize the starting step of the design flow. It is required to first describe the behavior of the system using a formal language and therefore we need a language that can describe the behavior of the system efficiently. That is how the concept of a HDL came. VHDL is a formal HDL that is very popular in both academia and industry.

VHDL is used to describe digital logic circuits at different levels of hierarchy starting from very basic gate level up to the system level. The designers can describe the logic circuits in the form of Boolean expressions, or using the behavioral statements that describe the truth table, or using structural description using the components of the design. It can be used for synthesis of the design, timing analysis, and functional verification by doing simulation.

Before we start describing the details of the language, let us first discuss the digital circuits. This course on digital logic circuit design is usually taught in early semester of the undergraduate course. In this chapter, we shall discuss the concepts of digital logic circuit design.

The digital logic circuit works on two states: one state is called a logic high or logic 1 and the other one is called a logic low or logic 0. The logic states can be described by several ways—such as voltage, current, switch mechanical positions, and light on/off state. The logic circuits have finite number of inputs and finite number of outputs. Depending on the inputs, the outputs are either at logic high or at low state. Generally, digital logic circuits are classified into two types: one is called combinational logic and the other one is called sequential logic. In the combinational logic, the outputs are entirely dependent on the present inputs, whereas, in the sequential circuit, the outputs depend on present inputs as well the past output states. Therefore, the combinational circuits are memory-less circuits and the sequential circuits have memory.

1.2 BOOLEAN ALGEBRA

Boolean algebra was first introduced by George Boole in 1854. It defines different types of logical operations and works with variables called Boolean variables. The value of the Boolean variables can be either TRUE/HIGH or FALSE/LOW. The digital systems work on binary inputs and produce binary output. In binary system, a TRUE or HIGH state is represented as logic 1 and a FALSE or LOW state is represented as logic 0 in positive logic. In negative logic, a HIGH state is represented by logic 0 and a LOW state is represented by logic 1.

The basic laws of Boolean algebra are shown in Table 1.1.

TABLE 1.1 Basic laws of Boolean algebra

$A + 0 = A$	$A \cdot 0 = 0$
$A + 1 = 1$	$A \cdot 1 = A$
$A + A = A$	$A \cdot A = A$
$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$

1.2.1 Boolean Theorems

Boolean algebra supports three basic laws of simple algebra. These laws are stated as follows:

1. Commutative law:

$$A + B = B + A \tag{1.1}$$

$$A \cdot B = B \cdot A \tag{1.2}$$

2. Associative law:

$$A + (B + C) = (A + B) + C \tag{1.3}$$

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C \tag{1.4}$$

3. Distributive law:

$$A \cdot (B + C) = A \cdot B + A \cdot C \tag{1.5}$$

$$A + B \cdot C = (A + B) \cdot (A + C) \tag{1.6}$$

DeMorgan's theorems are also very important Boolean theorems, which are extensively used in simplifying the Boolean expressions. These are as follows:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{1.7}$$

$$\overline{A \cdot B} = \overline{A} + \overline{B} \tag{1.8}$$

1.3 BASIC LOGIC GATES

In binary logic, logical operations are used between the binary variables to determine the final output. There are three basic logical operations: NOT, AND, and OR. The electronic circuit that performs the binary operation is called a logic gate.

All digital logic circuits are based on three primary logic gates: NOT, AND, and OR gates. Any logic functionality can be achieved by these three primary logic functions. Typically, digital circuits work on the principle of binary logic where there are only two logic values: logic 1 and logic 0. In positive logic, logic 1 is represented by a high-voltage level and logic 0 is represented by a low-voltage level, whereas, in negative logic, logic 1 is represented by a low-voltage level and logic 0 is represented by a high-voltage level. In this book, we shall use positive logic unless stated otherwise.

1.3.1 NOT Gate

A NOT gate has one input and one output. The symbol of NOT gate is shown in Fig. 1.2.

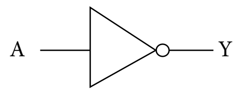


FIGURE 1.2 Symbol of NOT gate

The output (Y) of NOT gate is the complement of the input (A). The Boolean expression of a NOT gate is described as follows:

$$Y = \overline{A} \tag{1.9}$$

The truth table of NOT gate is shown in Table 1.2.

TABLE 1.2 Truth table of NOT gate

A	Y
0	1
1	0

The NOT gate is also known as inverter.

1.3.2 AND Gate

The AND gate has many inputs and one output. The symbols of two-input and three-input AND gates are shown in Fig. 1.3.

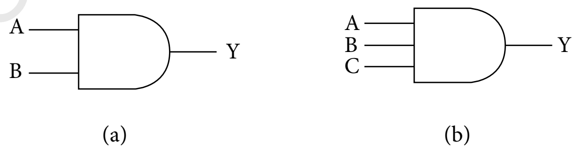


FIGURE 1.3 Symbol of AND gate: (a) two-input, (b) three-input

The Boolean expression of the output (Y) for a two-input AND gate is given as follows:

$$Y = A \cdot B \tag{1.10}$$

The truth table of two-input AND gate is shown in Table 1.3.

The AND logic function is associative; that is, any AND function with more than two inputs can be realized using two-input AND functions. For example,

$$Y(A, B, C) = (A \cdot B) \cdot C = A \cdot (B \cdot C) \tag{1.11}$$

TABLE 1.3 Truth table of AND gate

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

1.3.3 OR Gate

The OR gate has many inputs and one output. The symbols of two-input and three-input OR gates are shown in Fig. 1.4.

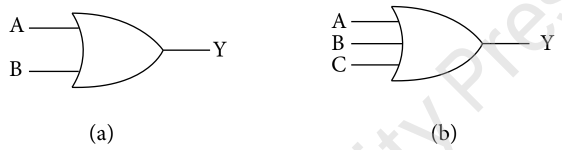


FIGURE 1.4 Symbol of OR gate: (a) two-input, (b) three-input

The Boolean expression of the output (Y) for a two-input OR gate is given as follows:

$$Y = A + B \tag{1.12}$$

The truth table of two-input OR gate is shown in Table 1.4.

TABLE 1.4 Truth table of OR gate

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

The OR logic function is associative; that is, any OR function with more than two inputs can be realized using two-input OR functions. For example,

$$Y(A, B, C) = (A + B) + C = A + (B + C) \tag{1.13}$$

1.3.4 XOR Gate

The XOR gate has two or more inputs and one output. The symbols of two-input and three-input XOR gates are shown in Fig. 1.5.

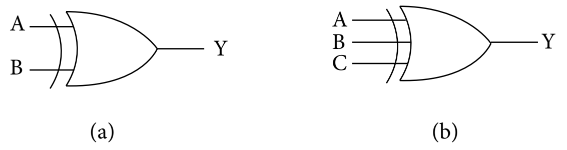


FIGURE 1.5 Symbol of XOR gate: (a) two-input, (b) three-input

The Boolean expression of the output (Y) for a two-input XOR gate is given as follows:

$$Y = A \oplus B = A\overline{B} + \overline{A}B \tag{1.14}$$

The truth table of two-input XOR gate is shown in Table 1.5.

TABLE 1.5 Truth table of XOR gate

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

The XOR logic function is associative; that is, any XOR function with more than two inputs can be realized using two-input XOR functions. For example,

$$Y(A, B, C) = (A \oplus B) \oplus C = A \oplus (B \oplus C) \tag{1.15}$$

1.3.5 NAND Gate

The NAND gate has two inputs and one output. The symbol of NAND gate is shown in Fig. 1.6(a).

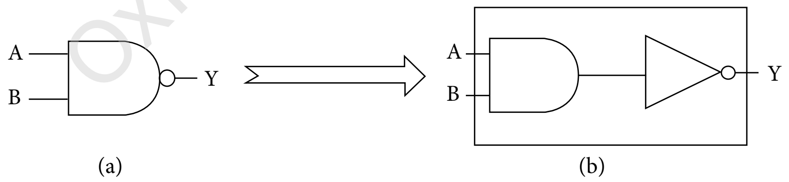


FIGURE 1.6 (a) Symbol of NAND gate and (b) its equivalent representation

A NAND gate is equivalent to an AND gate followed by a NOT gate as illustrated in Fig. 1.6(b). The Boolean expression of the output (Y) for a NAND gate is given as follows:

$$Y = \overline{A \cdot B} = \overline{A} + \overline{B} \tag{1.16}$$

The truth table of NAND gate is shown in Table 1.6.

The NAND logic is not associative. The NAND function with more than two inputs cannot be realized using two-input NAND gates.

TABLE 1.6 Truth table of NAND gate

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

1.3.6 NOR Gate

The NOR gate has two inputs and one output. The symbols of NOR gate is shown in Fig. 1.7.

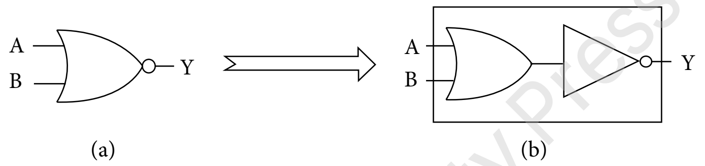


FIGURE 1.7 (a) Symbol of NOR gate and (b) its equivalent representation

A NOR gate is equivalent to an OR gate followed by a NOT gate as illustrated in Fig. 1.7(b). The Boolean expression of the output (Y) for a NOR gate is given as follows:

$$Y = \overline{(A+B)} = \overline{A}\cdot\overline{B} \tag{1.17}$$

The truth table of NOR gate is shown in Table 1.7.

TABLE 1.7 Truth table of NOR gate

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

The NOR logic is not associative. The NOR function with more than two inputs cannot be realized using two-input NOR gates.

1.3.7 XNOR Gate

The XNOR gate has two or more inputs and one output. The symbols of two input and three input XNOR gates are shown in Fig. 1.8.

The Boolean expression of the output (Y) for a two-input XNOR gate is given as follows:

$$Y = A \odot B = AB + \overline{A}\,\overline{B} \tag{1.18}$$

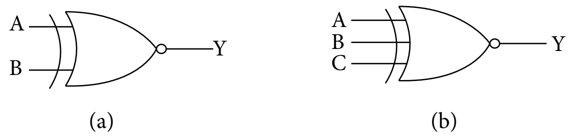


FIGURE 1.8 Symbol of XNOR gate: (a) two-input, (b) three-input

The truth table of two-input XNOR gate is shown in Table 1.8.

TABLE 1.8 Truth table of XNOR gate

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

The XNOR logic function is associative; that is, any XOR function with more than two inputs can be realized using two-input XOR functions. For example,

$$Y(A, B, C) = (A \odot B) \odot C = A \odot (B \odot C) \tag{1.19}$$

1.3.8 Universal Gates

The NAND and NOR logic gates are called universal gates because any logic function can be realized using only either NAND or NOR gates.

EXAMPLE 1.1 Realize NOT gate using NAND gate.

Solution

The NOT gate can be realized using NAND gate as follows: The NAND gate has two inputs, whereas the NOT gate has single input. If both inputs of NAND gate are shorted together, then the output of NAND gate is as follows:

$$Y = \overline{A \cdot B} = \overline{A \cdot A} = \overline{A} \tag{1.20}$$

Figure 1.9 shows the NOT gate realized using a NAND gate.

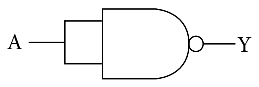


FIGURE 1.9 NOT gate realized using NAND gate

EXAMPLE 1.2 Realize NOT gate using NOR gate.

Solution

The NOT gate can be realized using NOR gate as follows: The NOR gate has two inputs, whereas the NOT gate has single input. If both inputs of NOR gate are shorted together, then the output of NOR gate is as follows:

$$Y = \overline{A + B} = \overline{A + A} = \overline{A} \tag{1.21}$$

Figure 1.10 shows the NOT gate realized using a NOR gate.

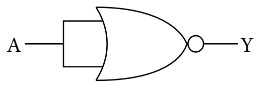


FIGURE 1.10 NOT gate realized using NOR gate

EXAMPLE 1.3 Realize AND gate using NAND gate.

Solution
The AND gate can be realized using NAND gate as shown in Fig. 1.11.

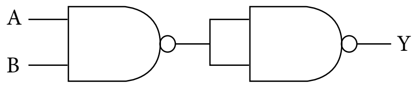


FIGURE 1.11 AND gate realized using NAND gate

EXAMPLE 1.4 Realize OR gate using NOR gate.

Solution
The OR gate can be realized using NOR gate as shown in Fig. 1.12.

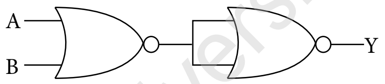


FIGURE 1.12 OR gate realized using NOR gate

EXAMPLE 1.5 Realize OR gate using NAND gate.

Solution
The OR gate can be realized using NAND as follows: The Boolean expression of NAND gate is given here.

$$F = \overline{A \cdot B} = \overline{A} + \overline{B} \tag{1.22}$$

Now if $A = \overline{X}$ and $B = \overline{Y}$, then it can be written as follows:

$$F = \overline{\overline{X}} + \overline{\overline{Y}} = X + Y \tag{1.23}$$

Figure 1.13 shows the realization of OR gate using NAND gates.

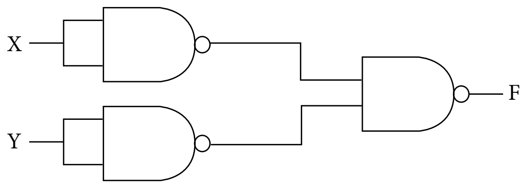


FIGURE 1.13 OR gate realized using NAND gate

EXAMPLE 1.6 Realize AND gate using NOR gate.

Solution
The AND gate can be realized using NOR gate as follows: The Boolean expression of NOR gate is given here.

$$F = \overline{A + B} = \overline{A} \cdot \overline{B} \tag{1.24}$$

Now, if $A = \overline{X}$ and $B = \overline{Y}$, then it can be written as follows:

$$F = \overline{\overline{X}} \cdot \overline{\overline{Y}} = X \cdot Y \tag{1.25}$$

Figure 1.14 shows the realization of AND gate using NOR gates.

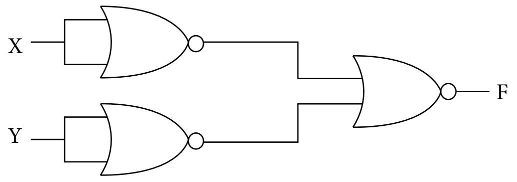


FIGURE 1.14 AND gate realized using NOR gate

EXAMPLE 1.7 Realize XOR gate using NAND gate.

Solution
The XOR gate can be realized using NAND as follows: The Boolean expression of NAND gate is given here.

$$F = \overline{A \cdot B} = \overline{A} + \overline{B} \tag{1.26}$$

Now, if $\overline{A} = \overline{X}Y$ and $\overline{B} = X\overline{Y}$, then it can be written as follows:

$$F = \overline{X}Y + X\overline{Y} \tag{1.27}$$

Figure 1.15 shows the realization of XOR gate using NAND gates.

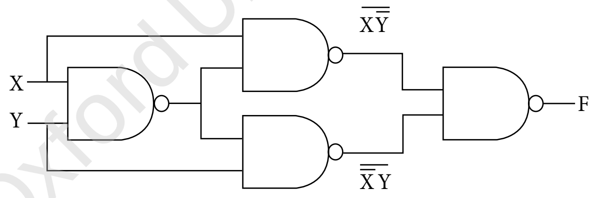


FIGURE 1.15 XOR gate realized using NAND gate

EXAMPLE 1.8 Realize XNOR gate using NOR gate.

Solution
The XNOR gate can be realized using NOR as follows: The Boolean expression of NOR gate is given here.

$$F = \overline{A + B} = \overline{A} \cdot \overline{B} \tag{1.28}$$

Now, if $\overline{A} = \overline{X}Y$ and $\overline{B} = X\overline{Y}$, then it can be written as follows:

$$F = \overline{\overline{X}Y} \cdot \overline{X\overline{Y}} = XY + \overline{X}\,\overline{Y} \tag{1.29}$$

Figure 1.16 shows the realization of XNOR gate using NOR gates.

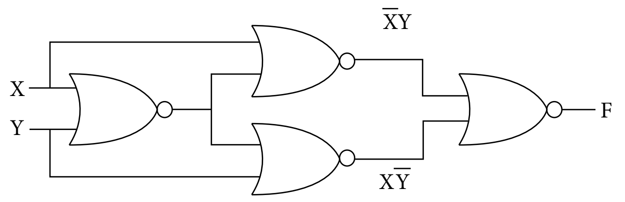


FIGURE 1.16 XNOR gate realized using NOR gate

1.3.9 AND-OR-INVERT (AOI) Gate

The AOI gate is realized using a combination of AND, OR, and NOT gates. For example, the Boolean expression given in Eq. (1.30) can be realized as shown in Fig. 1.17.

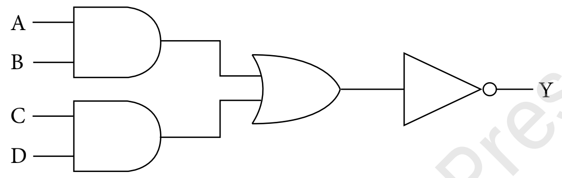


FIGURE 1.17 AND-OR-INVERT gate

$$Y = \overline{AB + CD} \tag{1.30}$$

AOI is useful in realizing any Boolean expression expressed in sum-of-product (SOP) form.

1.3.10 OR-AND-INVERT (OAI) Gate

The OAI gate is realized using a combination of OR, AND, and NOT gates. For example, the Boolean expression given in Eq. (1.31) can be realized as shown in Fig. 1.18.

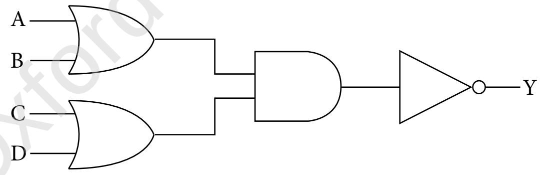


FIGURE 1.18 OR-AND-INVERT gate

$$Y = \overline{(A + B) \cdot (C + D)} \tag{1.31}$$

OR-AND-INVERT is useful in realizing any Boolean expression expressed in product-of-sum (POS) form.

1.3.11 Buffer Gate

A buffer is a logic gate with one input and one output. The symbol of a buffer is shown in Fig. 1.19.

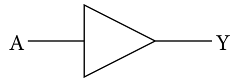


FIGURE 1.19 Buffer gate

It is implemented using two NOT gates connected in cascade. The Boolean expression for the output (Y) is as follows:

$$Y = A \tag{1.32}$$

The truth table of buffer is shown in Table 1.9.

TABLE 1.9 Truth table of buffer gate

A	*Y*
0	0
1	1

A buffer is used in digital circuits to increase the drive capability of a logic gate. For example, let us consider the scenario illustrated in Fig. 1.20. Here, the OR gate is driving four other logic gates.

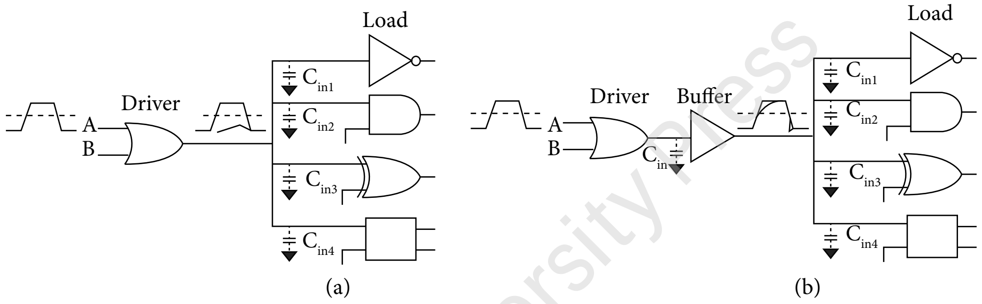


FIGURE 1.20 Driver-load configuration: (a) without a buffer, (b) with a buffer

The effective load capacitance seen by the OR gate is the sum of the input capacitances of four gates. Thus, the total load that the OR gate has to drive is given by $C_t = C_{\text{in}1} + C_{\text{in}2} + C_{\text{in}3} + C_{\text{in}4}$. This is known as the fan-out of the driver OR gate. If the drive strength of the driver gate is not enough, then the output of the driver will not reach up to the logic high level. This may create logic failure in extreme case when the output of driver is below the logic threshold. This problem can be solved by adding a buffer at the output of the driver. The buffer has less input capacitance ($C_{\text{in}} << C_t$) as compared to C_t but has large drive strength so that its output will be able to charge the effective load capacitance C_t up to the logic high level.

1.3.12 Tri-state Logic Gate

The tri-state logic gate has one data input, one control input, and one output. The control input can be active low or active high. Figure 1.21 shows the symbols of tri-state logic gate with active low and active high control inputs.

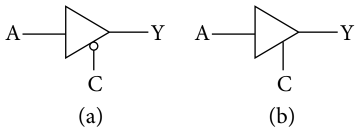


FIGURE 1.21 Symbol of tri-state logic gate: (a) active low control input, (b) active high control input

The truth table of tri-state logic gate is shown in Table 1.10. When the active low control input (C) is at logic 0, the buffer is enabled and the output (Y) is same as the input (A). When the active low control input is at logic 1, the buffer is disabled and the output is at high impedance state, which is represented by 'Z'. The high impedance state indicates that the output is neither '1' nor '0', that is, no current flows in this state. The tri-state buffer with active high control input works in opposite way of the tri-state buffer with active low control input.

TABLE 1.10 Truth table of tri-state logic gate with active low and active high inputs

Inputs		Output	Inputs		Output
C (active low)	*A*	*Y*	*C* (active high)	*A*	*Y*
0	0	1	0	0	*Z*
0	1	0	0	1	*Z*
1	0	*Z*	1	0	0
1	1	*Z*	1	1	1

1.3.13 Multi-bit Tri-state Buffer

A multi-bit tri-state buffer has multiple data inputs and outputs but only one control input. In Fig. 1.22, eight tri-state buffers are connected in parallel with common control input. When control is enabled, all eight tri-state buffers transfer their data inputs to their outputs. In the disable state, all the buffer outputs are in high impedance stage.

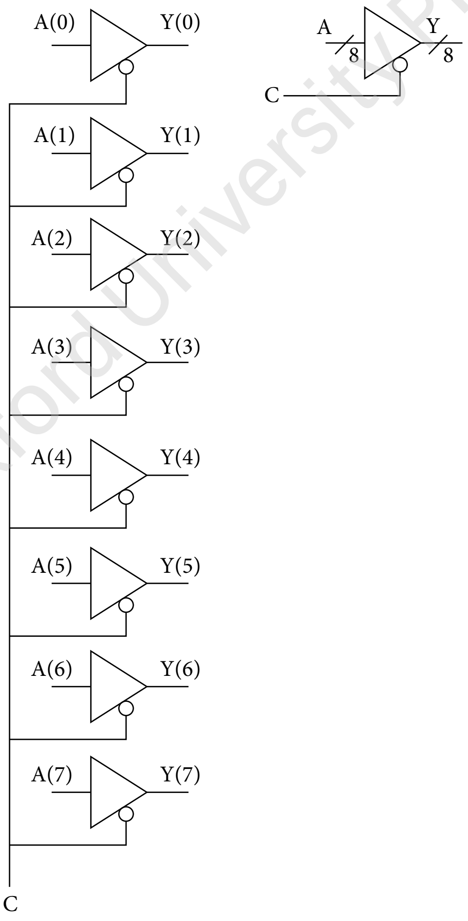


FIGURE 1.22 Multi-bit tri-state buffer

It is used in digital circuits where a bus is driven by multiple logic devices as illustrated in Fig. 1.23.

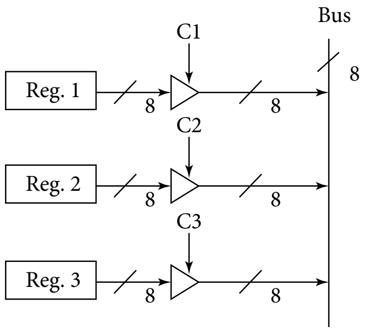


FIGURE 1.23 Bus architecture with tri-state buffer

In a digital system, it is common that many devices communicate with each other through common bus architecture. Thus, the bus is driven by multiple drivers. However, during the operation, the data on the bus must come from only one device, although there can be multiple devices reading from it. As multiple devices (such as registers) produce output simultaneously and the devices are connected to common bus, there must be a way to control which data gets on the bus, and which does not. The control of data that goes to the bus can be achieved through tri-state buffer. Another possible solution is to use multiplexers, which we shall discuss later in this chapter.

1.3.14 Bus

In digital systems, there is a concept of bus architecture. A bus is a group of wires that are used to transmit data from one device to another. For example, in microprocessor architecture, we learned about data bus, address bus, and control bus. The bus size is simply the number of wires in the bus. A bus is schematically drawn by a line with a dash across it. The number beside the dash represents the size of the bus. Figure 1.24 illustrates an 8-bit bus.

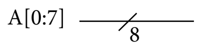


FIGURE 1.24 Schematic of an 8-bit bus

The advantage of using bus is that multiple devices can be connected to a bus. Multiple devices can access/read data from the bus simultaneously but they must write into the bus one at a time. There are mainly three kinds of buses: data bus, address bus, and control bus. The data bus is used to read data from the memory or write into the memory. The address bus is used to specify the address of the memory location from which the data is to be read or the address of the memory location to which the data is to be written. Other than data and address buses, there is a control bus, which is used to control the operations in a digital system. One important attribute of bus is its speed. It defines how fast the data can be changed per second. The speed of the bus determines how fast the central processing unit (CPU) can exchange data with the peripheral devices.

1.3.15 Bus Holder Circuit

In bus architecture, it can so happen that multiple devices can simultaneously attempt to write into the bus. This can create erroneous data on the bus and in extreme case may damage the bus wiring. This problem is known as bus contention. The opposite problem of bus contention is bus floating. When no devices drive the bus, the data on the bus is unknown or floating. This floating condition also may damage the logic gates. In order to prevent the bus floating condition, bus holder circuit is used. Bus holder is a weak latch circuit implemented by two NOT gates connected back to back as shown in Fig. 1.25. It holds last value on a tri-state bus and is also known as *bus* *keeper* circuit. As the bus keeper is made of weak inverters, it can easily be driven to change its output state. It must be noted that although a bus holder works as a latch it must not be used as practical latches.

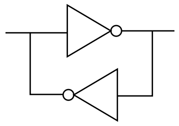


FIGURE 1.25 Bus holder circuit

1.3.16 Bidirectional Buffer

Sometimes it is required to have buffer that acts both as input buffer and as output buffer. This type of buffer is known as bidirectional buffer. It has one data input and one data output port. The control input determines whether to transfer the data output to the I/O pad or from I/O pad to the data input. The schematic of a bidirectional buffer is shown in Fig. 1.26.

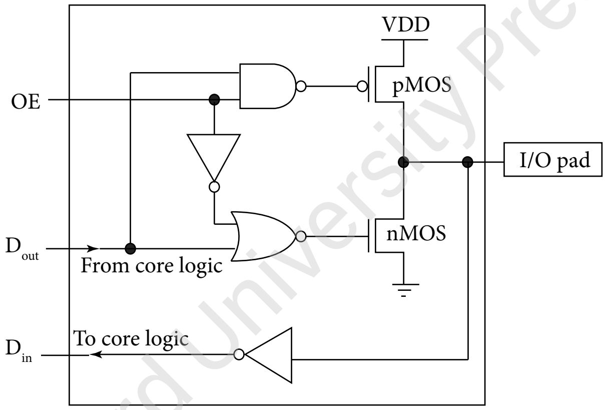


FIGURE 1.26 Schematic of a bidirectional buffer

When the signal OE is high, both the NAND and the NOR gates are enabled. If D_{out} is logic high, the output of NAND gate is low and the output of NOR gate is also low. So the pMOS is turned on and the nMOS is turned off. Thus, the I/O pad is connected to VDD, that is, the signal D_{out} is transferred to I/O pad. When D_{out} is logic low, the output of NOR gate is high making the nMOS on and the output of NAND gate is high making pMOS off. Thus, I/O pad is connected to ground, that is, the signal D_{out} is transferred to I/O pad. When the signal OE is low, both the NAND and NOR gates are disabled. Both pMOS and nMOS transistors are off. Now the signal from I/O pad is transferred to the core logic through D_{in}.

1.4 COMBINATIONAL LOGIC CIRCUITS

The logic circuits without memory elements are called combinational logic circuits. In these circuits, the outputs are entirely dependent upon the present input logic levels. Some of the examples of combinational circuits are adder, subtractor, comparator, decoder, encoder, multiplexer, demultiplexer, parity generator, parity checker, code converters, barrel shifters, etc.

In this section, we shall discuss the basic combinational circuits. The main steps of designing combinational circuits are as follows:

1. Identify the number of input and output variables from the Boolean expressions or truth tables.

2. Minimize the Boolean expressions using Karnaugh-map method or Quine-McCluskey method.
3. Implement the simplified Boolean expressions using basic logic gates.

1.4.1 Half-adder

A half-adder is a logic circuit that adds two bits. It has two inputs *A* and *B*, and two outputs Sum and Carry. The behavior of half-adder is expressed in the form of truth table as shown in Table 3.2.

TABLE 1.11 Truth table of half-adder

Input bits		***A* + *B***		**Output bits**	
A	*B*	In decimal	In binary	Carry ($\times 2^1$)	Sum ($\times 2^0$)
0	0	0	00	0	0
0	1	1	01	0	1
1	0	1	01	0	1
1	1	2	10	1	0

When two bits are added, the maximum sum is 2. Thus, two bits are required to represent the sum. Out of these two bits, carry has a binary weight of 2^1 and sum has a binary weight of 2^0. The Sum and Carry are expressed as follows:

$$Sum = \overline{A}B + A\overline{B} = A \oplus B \tag{1.33}$$

$$Carry = AB \tag{1.34}$$

From the expressions of Sum and Carry, it is clear that to implement the half-adder simply one two-input XOR and two-input AND gates are required. The structure of a half-adder is shown in Fig. 1.27.

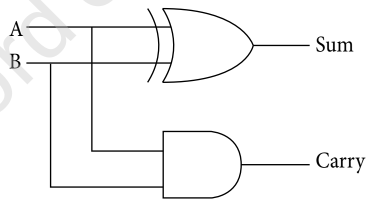


FIGURE 1.27 Half-adder

1.4.2 Full-adder

A full-adder (FA) is a logic circuit, which adds three bits. When three bits are added, the maximum sum can be 3. Thus, two bits are required to represent the output. Out of these two bits, the bit with a binary weight of 2^1 is known as carry (C_{out}) and the bit with a binary weight of 2^0 is known as sum (Sum). The behavior of FA can be expressed in the form of truth table as shown in Table 1.12.

The Boolean expression for Sum of FA is as follows:

$$\begin{aligned} Sum &= \overline{A}\,\overline{B}C_{\text{in}} + \overline{A}B\overline{C}_{\text{in}} + A\overline{B}\,\overline{C}_{\text{in}} + ABC_{\text{in}} && (1.35)\\ &= \overline{A}(\overline{B}C_{\text{in}} + B\overline{C}_{\text{in}}) + A(\overline{B}\,\overline{C}_{\text{in}} + BC_{\text{in}}) && (1.36)\\ &= \overline{A}\cdot(B \oplus C_{\text{in}}) + A\cdot\overline{(B \oplus C_{\text{in}})} && (1.37)\\ &= A \oplus B \oplus C_{\text{in}} && (1.38) \end{aligned}$$

TABLE 1.12 Truth table of full-adder

Input bits			**$A + B + C_{in}$**		**Output bits**	
A	B	C_{in}	In decimal	In binary	Carry ($\times 2^1$)	Sum ($\times 2^0$)
0	0	0	0	00	0	0
0	0	1	1	01	0	1
0	1	0	1	01	0	1
0	1	1	2	10	1	0
1	0	0	1	01	0	1
1	0	1	2	10	1	0
1	1	0	2	10	1	0
1	1	1	3	11	1	1

The Boolean expression for C_{out} of FA is as follows:

$$C_{out} = \overline{A}BC_{in} + A\overline{B}C_{in} + AB\overline{C_{in}} + ABC_{in} \tag{1.39}$$

$$= BC_{in}(\overline{A} + A) + AC_{in}(\overline{B} + B) + AB(\overline{C_{in}} + C_{in}) \tag{1.40}$$

$$= AB + C_{in}(A + B) \tag{1.41}$$

A full-adder can be represented at the gate level as shown in Fig. 1.28. In this implementation, only two-input gates are used. Therefore, the number of logic stage is three for implementing the C_{out} output.

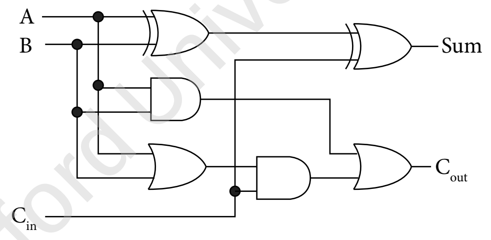


FIGURE 1.28 Gate level representation of full-adder

1.4.3 Full-adder using Half-adders

A full-adder can be implemented using two half-adders. Let us explain this design with the help of the truth table 1.13.

It is observed that Sum output of the second half-adder is the final Sum output of FA. The final carry output of FA can be obtained by ORing the Carry outputs of two half-adders. Therefore, the logic circuit of FA can be implemented using two half-adders and one OR gate as illustrated in Fig. 1.29.

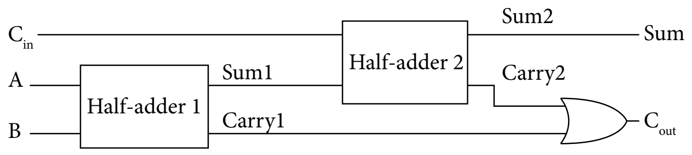


FIGURE 1.29 Full-adder using two half-adders and one OR gate

TABLE 1.13 Truth table of a full-adder

Input bits			Outputs of first half-adder		Outputs of second half-adder		Final outputs	
A	B	C_{in}	Carry1 $=AB$	Sum1 $= A \oplus B$	Carry2 $= C_{in} \cdot$ Sum1	Sum2 $=C_{in} \oplus$ Sum1	C_{out} =Carry1+Carry2	Sum =Sum2
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1
0	1	0	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	0	1	0	1
1	0	1	0	1	1	0	1	0
1	1	0	1	0	0	0	1	0
1	1	1	1	0	0	1	1	1

1.4.4 *n*-Bit Serial Adder

An *n*-bit adder adds to numbers of size *n*-bit each. It produces the result of size (n+1)-bit of which 1 bit is Carry and *n*-bit Sum. The serial adder adds bit-by-bit serially. That is why it is known as serial adder. It is smallest hardware for adding *n*-bit numbers. The size of the hardware is independent of the word size (n). The structure of a serial adder is shown in Fig. 1.30.

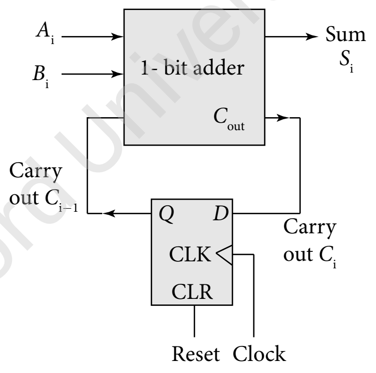


FIGURE 1.30 Serial binary adder

The basic components of a serial adder are one FA and one D-type flip-flop (DFF). The C_{out} of FA is connected to the D-input of DFF and Q-output of DFF is connected to the C_{in} input of FA so that it can be added with next two bits. It adds numbers bit by bit with previous carry. Hence, to add *n*-bit numbers, it requires *n* clock pulses.

1.4.5 *n*-Bit Parallel Adder

A parallel adder adds two numbers in parallel. It uses as many full adders as the number of bits in *n*-bit word. It can add two *n*-bit binary numbers with a previous carry. An *n*-bit parallel adder circuit is shown in Fig. 1.31. The carry bit generated from one FA ripples through the next FA circuit. For this reason, it is also known as ripple carry adder (RCA).

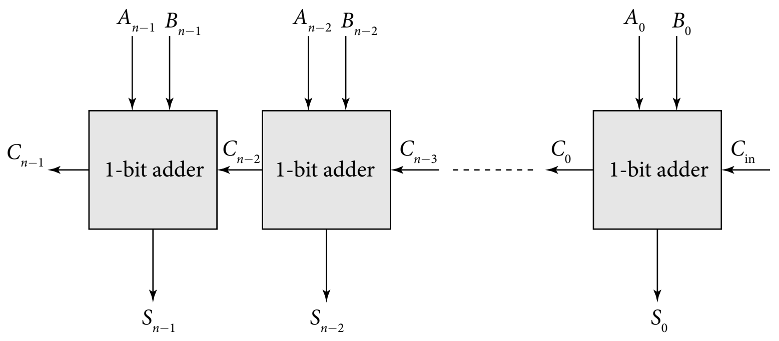


FIGURE 1.31 *n*-Bit parallel adder

If each FA takes time t_{carry} to evaluate Carry bit and time t_{sum} to evaluate the Sum bit, the total delay for an n-bit parallel adder will be as follows:

$$t_{\text{adder}} = (n-1)t_{\text{carry}} + t_{\text{sum}} \tag{1.42}$$

For long word size, this delay can be significantly large. Due to this reason, RCA is not preferred for signals with long words.

1.4.6 Carry Look-ahead Adder

Ripple carry adder is slow as the final output carry depends on all the FA stages to generate the carry signals. If τ is delay of each FA block, it takes $n\tau$ time to generate the final output carry. Carry look-ahead adder (CLA) is one of the high-speed adder circuits that do not wait for each FA to generate the carry signals. Instead it finds the carry signal ahead by using a carry look-ahead generator circuit. This way CLA can add two n-bit binary numbers very fast.

Principle: The ith sum and carry of an FA in the RCA are given by the following equations:

$$S_i = A_i \oplus B_i \oplus C_{i-1} \tag{1.43}$$

$$C_i = A_iB_i + B_iC_{i-1} + A_iC_{i-1} \tag{1.44}$$

To generate the carry signal, let us introduce two auxiliary signals, generate and propagate, as given by the following equations:

$$G_i = A_iB_i \tag{1.45}$$

$$P_i = A_i + B_i \tag{1.46}$$

Now, Eq. (1.44) can be written as follows:

$$C_i = G_i + P_iC_{i-1} \tag{1.47}$$

Similarly, C_{i-1} can be written as follows:

$$C_{i-1} = G_{i-1} + P_{i-1}C_{i-2} \tag{1.48}$$

Substituting Eq. (1.48) in Eq. (1.47), we get

$$C_i = G_i + P_iG_{i-1} + P_iP_{i-1}C_{i-2} \tag{1.49}$$

For a four-stage CLA, we can write the equation as follows:

$$C_0 = G_0 + P_0 C_{\text{in}} \tag{1.50}$$

$$C_1 = G_1 + P_1 C_0 = G_1 + P_1 G_0 + P_1 P_0 C_{\text{in}} \tag{1.51}$$

The gate level structure of a 4-bit CLA circuit is shown in Fig. 1.32. All the sum and final carry bits are generated using four-level logic. If the average propagation delay of each logic level is τ, then adder generates all the output bits after 4τ time delay, which is independent of word size n.

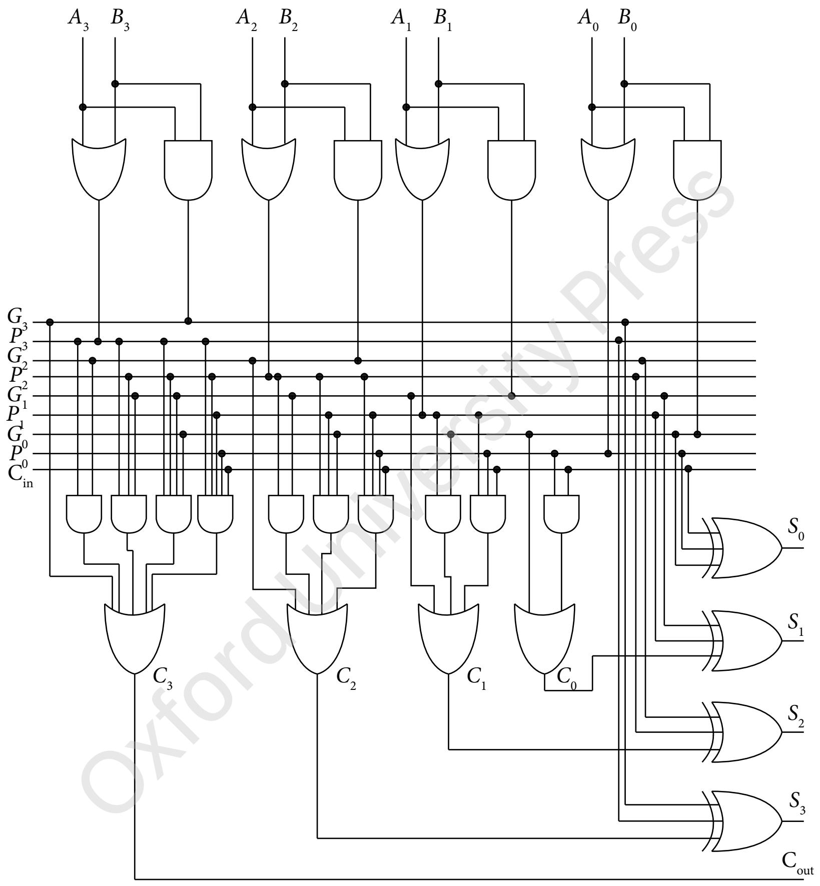


FIGURE 1.32 4-Bit carry look-ahead adder

1.4.7 Subtractor

A subtractor is a logic circuit that subtracts two binary numbers. In digital logic, subtraction is performed using addition technique where the subtrahend or negative number is represented in 2's complement form.

In 2's complement form, the most significant bit (MSB) is reserved to represent the sign of the number. If the sign bit is '1', the number is negative and if the sign bit is '0' the number is positive. Table 3.3 shows the unsigned and 2's complement format for 4-bit numbers.

In 2's complement form using n-bits, the range of decimal numbers that can be represented is from $+(2^{n-1}-1)$ to $-(2^{n-1})$.

TABLE 1.14 Binary number representation of +ve and −ve numbers

Decimal	Unsigned form	Signed form	2's complement form
+8	–	–	–
+7	111	0111	0111
+6	110	0110	0110
+5	101	0101	0101
+4	100	0100	0100
+3	011	0011	0011
+2	010	0010	0010
+1	001	0001	0001
+0	000	0000	0000
−0	–	1000	–
−1	–	1001	1111
−2	–	1010	1110
−3	–	1011	1101
−4	–	1100	1100
−5	–	1101	1011
−6	–	1110	1010
−7	–	1111	1001
−8	–	–	1000

1.4.8 Half-subtractor

A half-subtractor is a digital logic circuit that subtracts two binary bits. It subtracts *B* (subtrahend) from A (minuend). It produces two output bits, difference and borrow. The truth table of a half-subtractor is shown in Table 3.4.

TABLE 1.15 Truth table of half-subtractor

Input bits		*A* − *B*		Output bits	
A	*B*	In decimal	In binary	Borrow ($\times -2^1$)	Difference ($\times 2^0$)
0	0	0	00	0	0
0	1	−1	11	1	1
1	0	1	01	0	1
1	1	0	00	0	0

The Boolean expressions for difference and borrow are given by the following equations:

$$Difference = A \oplus B \tag{1.52}$$

$$Borrow = \overline{A}B \tag{1.53}$$

A schematic circuit of half-subtractor is shown in Fig. 1.33.

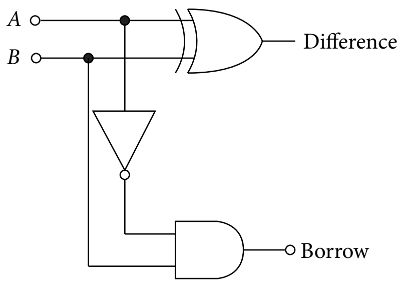


FIGURE 1.33 Schematic of half-subtractor

1.4.9 Full-subtractor

A full-subtractor subtracts *B* (subtrahend) from *A* (minuend) taking into account the borrow of the previous stage (B_{in}). The truth table of full-subtractor is shown in Table 3.5.

TABLE 1.16 Truth table of full-subtractor

Input bits			$A - B - B_{in}$		Output bits	
A	*B*	Bin	In decimal	In binary	Borrow ($\times -2^1$)	Difference ($\times 2^0$)
0	0	0	0	00	0	0
0	0	1	−1	11	1	1
0	1	0	−1	11	1	1
0	1	1	−2	10	1	0
1	0	0	1	01	0	1
1	0	1	0	00	0	0
1	1	0	0	00	0	0
1	1	1	−1	11	1	1

A schematic circuit of full-subtractor is shown in Fig. 1.34.

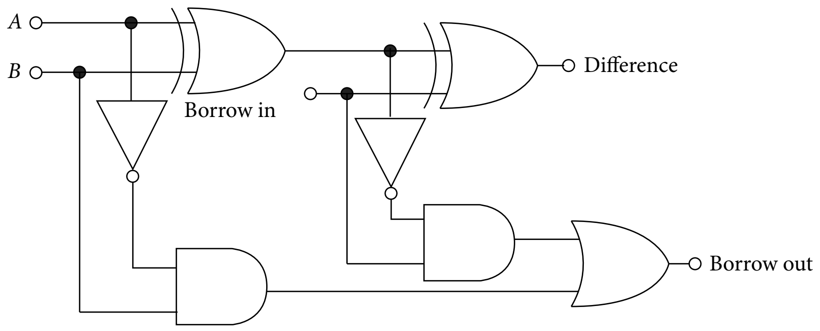


FIGURE 1.34 Schematic of full-subtractor

1.4.10 Adder/Subtractor Circuit

In digital logic, the subtraction operation is performed by the addition of 2's complement of the number to be subtracted. Therefore, an adder circuit can add as well as subtract two binary numbers with a small modification in the inputs. An adder/subtractor circuit implemented using an adder circuit with slight modification is shown in Fig. 1.35.

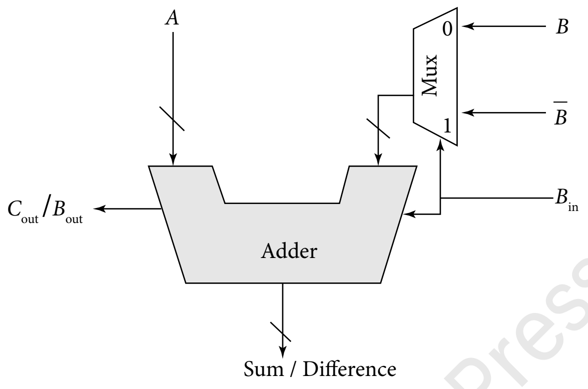


FIGURE 1.35 Adder/Subtractor circuit

Let us consider two 4-bit numbers *A* and *B*. The addition of *A* and *B* can be expressed as shown in Table 1.17. At first, the two least significant bits (LSBs), *A*0 and *B*0, are added with input carry $C_{in0} = 0$ using an FA. It produces the sum *S*0 and carry C_{out0}. Next, the 2SBs, *A*1 and *B*1, are added with input carry $C_{in1} = C_{out0}$ using an FA. In this manner, 3SBs *A*2, *B*2 and MSBs *A*3, *B*3 are added with previous carries C_{out1} and C_{out2}, respectively.

TABLE 1.17 Carry propagation in a 4-bit adder

	$A3$	$A2$	$A1$	$A0$
	$B3$	$B2$	$B1$	$B0$
	$C_{in3} = C_{out2}$	$C_{in2} = C_{out1}$	$C_{in1} = C_{out0}$	$C_{in0} = 0$
$C_{out} = C_{out3}$	$S3$	$S2$	$S1$	$S0$

Similarly, the subtraction of *A* and *B* can be expressed as shown in Table 1.18.

TABLE 1.18 Borrow propagation in a 4-bit subtractor

	$A3$	$A2$	$A1$	$A0$
	$\overline{B}3$	$\overline{B}2$	$\overline{B}1$	$\overline{B}0$
	$B_{in3} = B_{out2}$	$B_{in2} = B_{out1}$	$B_{in1} = B_{out0}$	$B_{in0} = 1$
$B_{out} = B_{out3}$	$D3$	$D2$	$D1$	$D0$

The complete structure of a 4-bit adder/subtractor circuit is shown in Fig. 1.36.

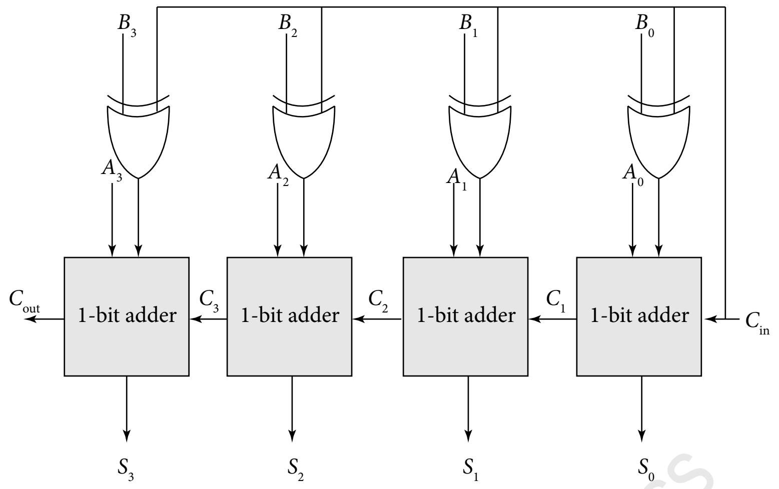


FIGURE 1.36 4-Bit adder/subtractor circuit

1.4.11 Multiplexer

A multiplexer is a combinational logic circuit with multiple data input lines and single output line. It passes one of its data input to the output. The select input lines determine which data input will go to the output. Multiplexer is also known as MUX. In general, with *n* select inputs, a MUX has 2^n data inputs and one output. It is denoted as $2^n : 1$ MUX. A 4:1 MUX has two select inputs, four data inputs, and one output line. Figure 1.37 shows the symbol of 4:1 MUX.

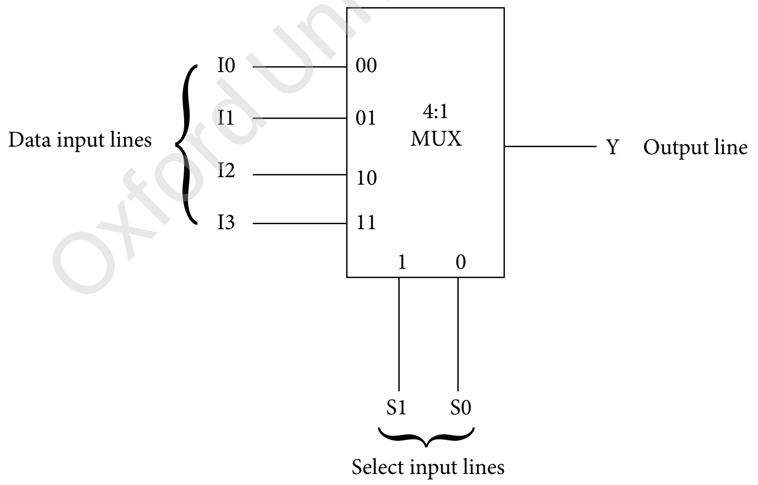


FIGURE 1.37 4:1 Multiplexer

The operation of the 4:1 MUX is illustrated with the help of Table 1.19.

1. When $S1 = 0$ and $S0 = 0$, $I0$ input goes to the output, that is, $Y = I0\overline{S1}\,\overline{S0}$.
2. When $S1 = 0$ and $S0 = 1$, $I1$ input goes to the output, that is, $Y = I1\overline{S1}\,S0$.
3. When $S1 = 1$ and $S0 = 0$, $I2$ input goes to the output, that is, $Y = I2S1\,\overline{S0}$.
4. When $S1 = 1$ and $S0 = 1$, $I3$ input goes to the output, that is, $Y = I3S1\,S0$.

The truth table of 4:1 MUX is shown in Table 1.19.

TABLE 1.19 Truth table of 4:1 MUX

Select inputs		Output
$S1$	$S0$	Y
0	0	$I0$
0	1	$I1$
1	0	$I2$
1	1	$I3$

The Boolean expression of the output of 4:1 MUX is given by the following equation:

$$Y = I0\overline{S1}\,\overline{S0} + I1\overline{S1}\,S0 + I2S1\,\overline{S0} + I3S1\,S0 \quad (1.54)$$

The 4:1 MUX can be implemented using basic logic gates as shown in Fig. 1.38.

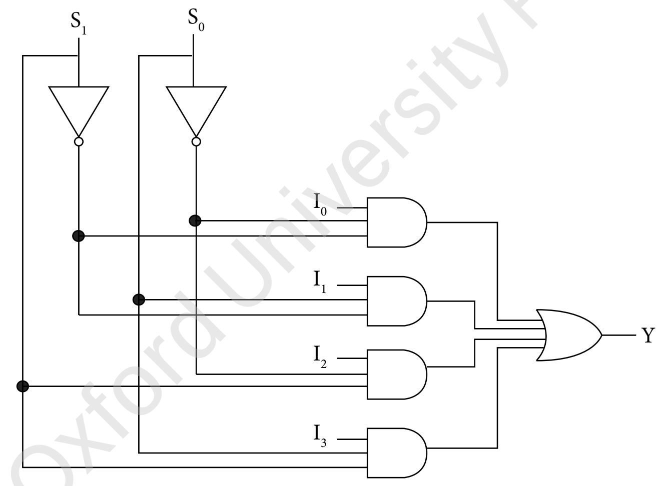


FIGURE 1.38 Logic circuit of 4:1 MUX

EXAMPLE 1.9 Design a 16:1 MUX using 4:1 MUXs.

Solution

A 16:1 MUX has 16 data input lines and 4 select input lines. The 16:1 MUX can be implemented using five 4:1 MUXs as shown in Fig. 1.39.

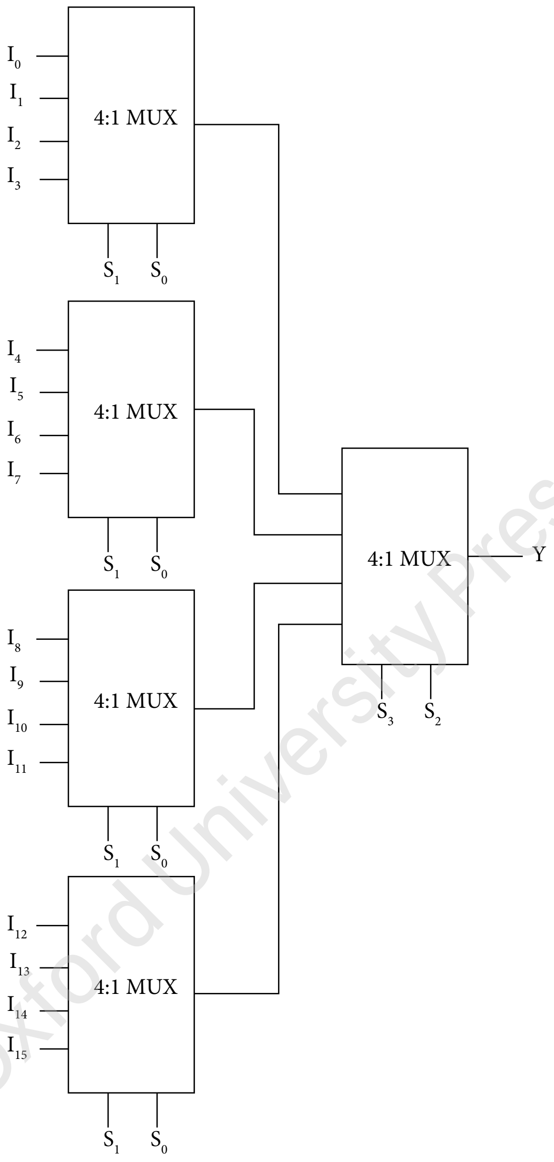


FIGURE 1.39 Design of 16:1 MUX using 4:1 MUXs

EXAMPLE 1.10 (a) Design a two-input XOR gate using 4:1 MUX; (b) repeat the same using 2:1 MUX.

Solution

(a) The truth table of a two-input XOR gate is given by Table 1.20. The table illustrates how the XOR gate can be realized using 4:1 MUX. Two select inputs act as primary inputs A and B of the XOR logic. The data inputs are set to the desired logic values for each combination of the select inputs.

A 4:1 MUX has two select inputs, $S1$ and $S0$. For each combination of the select inputs, output is determined by the data inputs. For example, when $S1 = 0$ and $S0 = 0$, input $I0$ is selected at the output. Again, for two-input XOR gate, when $A = 0$ and $B = 0$, output $Y = 0$. Therefore, if we make $S1 = A$ and $S0 = B$ and connect

TABLE 1.20 Truth table of an XOR gate implemented using 4:1 MUX

Truth table XOR gate			Truth table 4:1 MUX implementing an XOR logic			
Inputs		Output	Select inputs		Output	Required inputs of 4:1 MUX
A	B	Y	$S1$	$S0$	Y	$S1 = A$ and $S0 = B$
0	0	0	0	0	$I0$	$I0 = 0$
0	1	1	0	1	$I1$	$I1 = 1$
1	0	1	1	0	$I2$	$I2 = 1$
1	1	0	1	1	$I3$	$I3 = 0$

logic 0 to input $I0$, then it produces the first row in the truth table of XOR gate. In this way, we can implement two-input XOR gate using 4:1 MUX as shown in Fig. 1.40(a). When all the select lines of a MUX are used as the inputs to implement any combinational logic, the design style is known as type-0 design.

(b) Let us now implement a two-input XOR gate using 2:1 MUX. In 2:1 MUX, there is only one select input. Therefore, only one input (preferably A) can act as select input and the other input B must be connected to data inputs. As shown in Table 1.21, in a two-input XOR gate, it is observed that

1. when $A = 0$, $Y = B$, and
2. when $A = 1$, $Y = \overline{B}$

TABLE 1.21 Truth table of 2:1 MUX

Inputs		Output	Output	Select input	Output	Required inputs of 2:1 MUX
A	B	Y	Y	S	Y	$S = A$
0 0	0 1	0 1	$Y = B$ if $A = 0$	0	$I0$	$I0 = B$
1 1	0 1	1 0	$Y = \overline{B}$ if $A = 1$	1	$I1$	$I1 = \overline{B}$

Therefore, we can implement two-input XOR gate by connecting A to S, B to $I0$, and $\overline{B}$ to $I1$ as shown in Fig. 1.40(b). When one of the inputs of the combinational circuit to be implemented using MUX, is removed from the select lines, the design style is known as type-1 design.

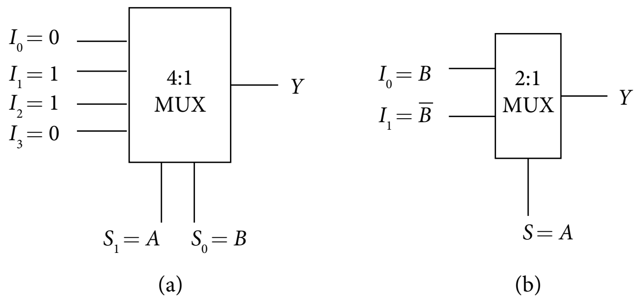


FIGURE 1.40 Two-input XOR gate implemented using (a) 4:1 MUX (b) 2:1 MUX

1.4.12 Demultiplexer

A demultiplexer has a logic circuit, which does the reverse operation of a multiplexer. It has a single data input line and 2^n output lines. The data input goes to one of the output lines depending on the select input lines. A $1 : 2^n$ demultiplexer has 2^n output lines and n select lines. A demultiplexer is also known as DEMUX. A 1:4 DEMUX is shown in Fig. 1.41.

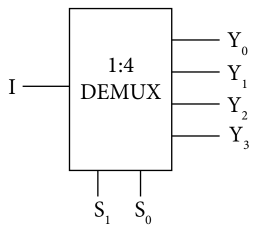


FIGURE 1.41 1:4 Demultiplexer

The operation of 1:4 DEMUX is illustrated in Table 1.22.

1. When $S1 = 0$ and $S0 = 0$, I input goes to the output $Y0$, that is, $Y0 = I\overline{S1}\,\overline{S0}$.
2. When $S1 = 0$ and $S0 = 1$, I input goes to the output $Y1$, that is, $Y1 = I\overline{S1}\,S0$.
3. When $S1 = 1$ and $S0 = 0$, I input goes to the output $Y2$, that is, $Y2 = IS1\,\overline{S0}$.
4. When $S1 = 1$ and $S0 = 1$, I input goes to the output $Y3$, that is, $Y3 = IS1\,S0$.

TABLE 1.22 Truth table of 1:4 DEMUX

Select inputs		Outputs			
$S1$	$S0$	$Y3$	$Y2$	$Y1$	$Y0$
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

The 1:4 DEMUX can be implemented using basic logic gates as shown in Fig. 1.42.

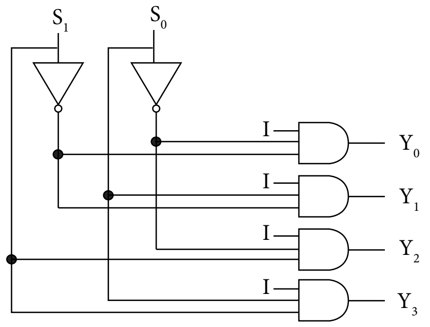


FIGURE 1.42 Logic circuit of 1:4 DEMUX

1.4.13 Decoder

A decoder is a combinational logic circuit with n input lines and 2^n output lines. Depending on the input combinations, one of the output lines becomes logic 1 and the remaining outputs become logic 0. For example, a 2:4 decoder has two input lines and four output lines as shown in Fig. 1.43.

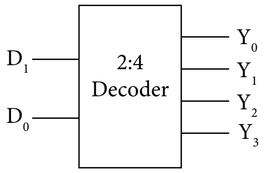


FIGURE 1.43 2:4 Decoder

The operation of a 2:4 decoder is illustrated in Table 1.23.

1. When $D1 = 0$ and $D0 = 0$, $Y0 = 1$ and $Y3 = Y2 = Y1 = 0$. Thus, $Y0 = \overline{D1}\,\overline{D0}$.
2. When $D1 = 0$ and $D0 = 1$, $Y1 = 1$ and $Y3 = Y2 = Y0 = 0$. Thus, $Y1 = \overline{D1}\,D0$.
3. When $D1 = 1$ and $D0 = 0$, $Y2 = 1$ and $Y3 = Y1 = Y0 = 0$. Thus, $Y2 = D1\,\overline{D0}$.
4. When $D1 = 1$ and $D0 = 1$, $Y3 = 1$ and $Y2 = Y1 = Y0 = 0$. Thus, $Y3 = D1\,D0$.

TABLE 1.23 Truth table of 2:4 decoder

Inputs		Outputs			
$D1$	$D0$	$Y3$	$Y2$	$Y1$	$Y0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

A 2:4 decoder can be realized using basic logic gates as shown in Fig. 1.44.

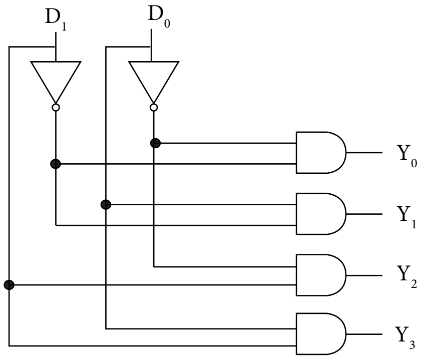


FIGURE 1.44 Logic circuit of 2:4 decoder

A decoder with n inputs implements 2^n minterms at the output. For example, a 2:4 decoder with inputs A and B implements four outputs as m_0, m_1, m_2, and m_3, where $m_0 = \overline{A}\,\overline{B}$, $m_1 = \overline{A}\,B$, $m_2 = A\,\overline{B}$, and $m_3 = AB$.

Thus, it can be used to implement any Boolean expression of two inputs. Similarly, in general a decoder with n inputs can implement any Boolean function of n input variables.

EXAMPLE 1.11 Design a full-adder using 3:4 decoder.

Solution

A full-adder has three inputs: A, B, and C_{in}. The Boolean expressions for its outputs Sum and C_{out} are given as follows:

$$Sum = \overline{A}\,\overline{B}C_{\text{in}} + \overline{A}B\overline{C_{\text{in}}} + A\overline{B}\,\overline{C_{\text{in}}} + ABC_{\text{in}} = m_1 + m_2 + m_4 + m_7 \tag{1.55}$$

$$C_{\text{out}} = \overline{A}BC_{\text{in}} + A\overline{B}C_{\text{in}} + AB\overline{C_{\text{in}}} + ABC_{\text{in}} = m_3 + m_5 + m_6 + m_7 \tag{1.56}$$

Thus, sum can be implemented by ORing four minterms: m_1, m_2, m_4, and m_7, and C_{out} can be implemented by ORing four minterms: m_3, m_5, m_6, and m_7 as shown in Fig. 1.45.

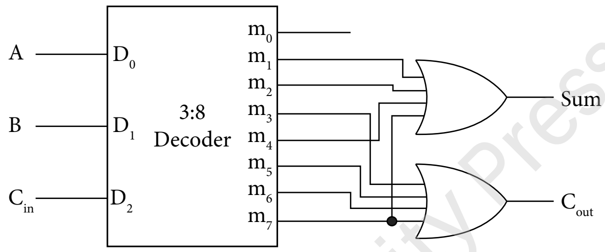


FIGURE 1.45 Full-adder implemented using a 3:8 decoder and two 4-input OR gates

1.4.14 Encoder

The encoder is a logic circuit that does the reverse operation of a decoder. It has 2^n input lines and n output lines. For example, a 4:2 encoder has four input lines and two output lines as shown in Fig. 1.46. The operation of 4:2 encoder is illustrated in Table 1.24.

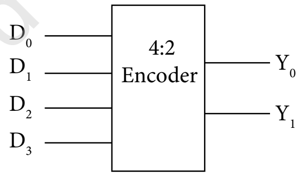


FIGURE 1.46 4:2 Encoder

TABLE 1.24 Truth table of 4:2 encoder

Inputs				Outputs	
$D3$	$D2$	$D1$	$D0$	$Y1$	$Y0$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

The Boolean expressions for the outputs of a 4:2 encoder are given as follows:

$$Y0 = D1 + D3 \tag{1.57}$$

and

$$Y1 = D2 + D3 \tag{1.58}$$

The logic circuit that implements a 4:2 encoder is shown in Fig. 1.47.

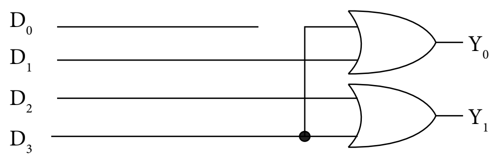


FIGURE 1.47 Logic circuit of 4:2 encoder

1.4.15 Comparator

A comparator is a digital logic circuit that compares two numbers A and B and produces three outputs $Y2$, $Y1$, and $Y0$ to indicate the following relations between A and B:

1. $Y2 = 1$, $Y1 = 0$, and $Y0 = 0$ if $A > B$.
2. $Y2 = 0$, $Y1 = 1$, and $Y0 = 0$ if $A = B$.
3. $Y2 = 0$, $Y1 = 0$, and $Y0 = 1$ if $A < B$.

Figure 1.48 shows a 4-bit comparator.

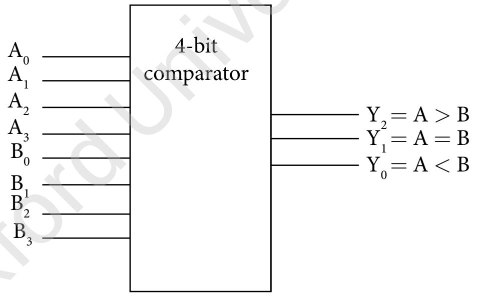


FIGURE 1.48 4-Bit comparator

EXAMPLE 1.12 Design a 2-bit comparator circuit using basic logic gates.

Solution

The functionality of a 2-bit comparator is described in Table 3.6.

The Boolean expressions for the outputs can be obtained using Karnaugh map method as follows:

$$Y2 = A1\overline{B1} + A0\overline{B0}(A1 + \overline{B1}) \tag{1.59}$$

$$Y1 = (\overline{A1}\,\overline{B1} + A1B1)(\overline{A0}\,\overline{B0} + A0B0) \tag{1.60}$$

$$Y0 = \overline{A1}B1 + \overline{A0}B0(\overline{A1} + B1) \tag{1.61}$$

TABLE 1.25 Truth table of a 2-bit comparator

Inputs				Outputs		
*A*1	*A*0	*B*1	*B*0	*Y*2	*Y*1	*Y*0
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Figure 1.49 shows the implementation of the 2-bit comparator.

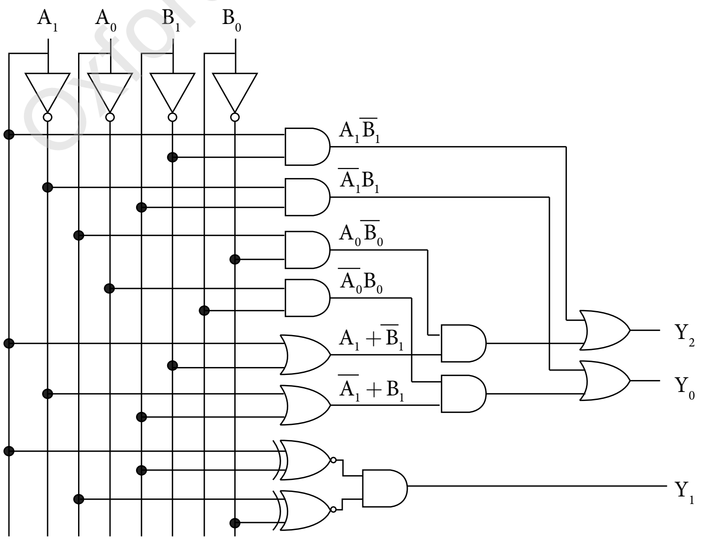


FIGURE 1.49 Logic circuit of a 2-bit comparator

1.4.16 Code Converter

A code converter is a logic circuit that converts a binary code to another binary code. For example, binary-to-grey code converter converts binary inputs to grey outputs. The truth table of the binary-to-grey code converter is shown in Table 1.26.

TABLE 1.26 Truth table of binary-to-grey code converter

Inputs				Outputs			
*B*3	*B*2	*B*1	*B*0	*G*3	*G*2	*G*1	*G*0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

The Boolean expression for the outputs obtained using K-map method is as follows:

$$G3 = B3 \tag{1.62}$$

$$G2 = B3 \oplus B2 \tag{1.63}$$

$$G1 = B2 \oplus B1 \tag{1.64}$$

$$G0 = B1 \oplus B0 \tag{1.65}$$

Figure 1.50 shows the logic circuit for binary-to-grey code converter.

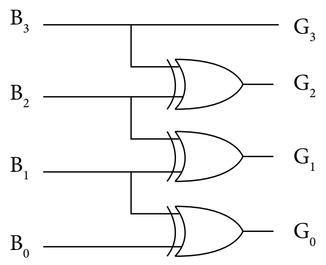


FIGURE 1.50 Logic circuit of binary-to-grey code converter

1.4.17 Parity Generator and Checker

Parity generation and checking are two processes adopted in the transmitter and receiver to detect any error in the process of data transmission from the transmitter to the receiver. An extra bit is added at the transmitter. This extra bit is known as parity bit. There are two different parity generation and checking process. These are even parity and odd parity. In even parity, the parity bit added to make to number of '1's even in a binary data. In odd parity, the parity bit added to make to number of '1's odd in a binary data. At the receiver end, if the received data does not match with the parity type followed at the transmitter, then an error is detected. Parity generator is the circuit that generates the parity bit, whereas parity checker is the circuit that checks the parity.

Parity Generator

Let us consider the case of even parity. Thus, the number of '1' in the binary data to be even including the parity bit. The truth table for a 3-bit even parity generator is given in Table 1.27.

TABLE 1.27 Truth table of even parity generator

Inputs			**Outputs**
3-bit data inputs			Even parity bit
*D*2	*D*1	*D*0	*P*
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

The Boolean expression for the parity bit *P* is obtained using K-map method as follows:

$$P = D2 \oplus D1 \oplus D0 \tag{1.66}$$

Figure 1.51(a) shows the logic circuit of 3-bit even parity generator.

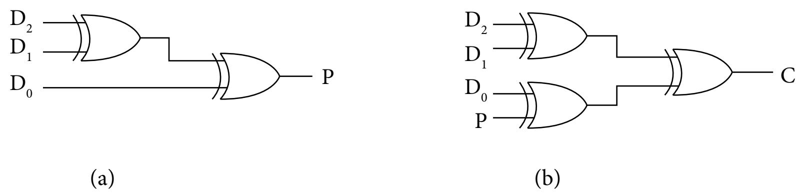


FIGURE 1.51 (a) Logic circuit of 3-bit even parity generator, (b) logic circuit of 4-bit even parity checker

Parity Checker

A parity checker checks if the parity is maintained in the binary data after the transmission. The output of parity checker is logic 1 when an error is detected in the transmitted data. Let us consider the 3-bit binary data with parity bit that is transmitted. Table 1.28 shows the truth table of 4-bit parity checker.

TABLE 1.28 Truth table of even parity checker

Inputs				**Outputs**
4-bit data inputs				Output of even parity checker
D2	*D1*	*D0*	*P*	*C*
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

The Boolean expression for the parity checker output C is obtained using K-map method as follows:

$$C = D2 \oplus D1 \oplus D0 \oplus P \qquad (1.67)$$

Figure 1.51(b) shows the logic circuit of 4-bit even parity checker.

1.4.18 Array Multiplier

Multipliers are the most useful building blocks after the adders in digital signal processor (DSP) or arithmetic computing systems. A multiplier has two binary inputs—one is called Multiplicand and the other is called Multiplier. It has one binary output, which is the product of Multiplicand and Multiplier. If A and B are the two 4-bit numbers, then their product can be written as follows:

$$P = \sum_{i=0}^{3} a_i 2^i \times \sum_{j=0}^{3} b_j 2^j \tag{1.68}$$

The 4×4-bit multiplication is illustrated in Fig. 1.52.

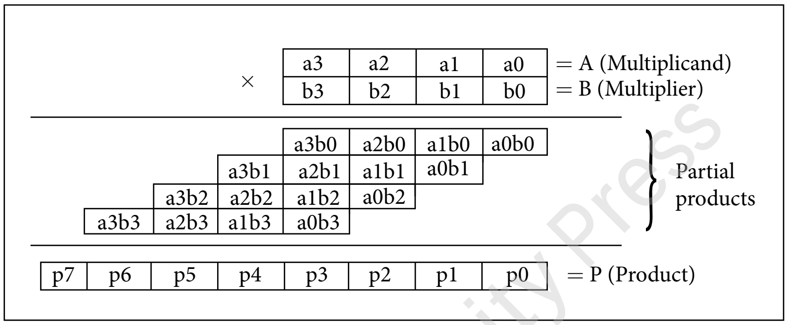

a3 a2 a1 a0 = A (Multiplicand)
×
b3 b2 b1 b0 = B (Multiplier)
a3b0 a2b0 a1b0 a0b0
a3b1 a2b1 a1b1 a0b1
a3b2 a2b2 a1b2 a0b2
a3b3 a2b3 a1b3 a0b3
Partial products
p7 p6 p5 p4 p3 p2 p1 p0 = P (Product)

FIGURE 1.52 Process of 4×4-bit multiplication

In general, the multiplier can be either unsigned type or signed type. An unsigned multiplier takes two binary inputs in unsigned format and produces product in unsigned format. The signed multiplier takes two binary inputs in 2's complement format and produces result in 2's complement format.

An unsigned array multiplier design is based on simple pen and paper method of multiplication. That is taking bit by bit from the multiplier, the partial products are generated by multiplying the bit to the multiplicand. Then the partial products are written in rows by shifting each row by one bit position to the right. The partial product rows are added column wise. Figure 1.53 illustrates the logic circuit of an unsigned array multiplier.

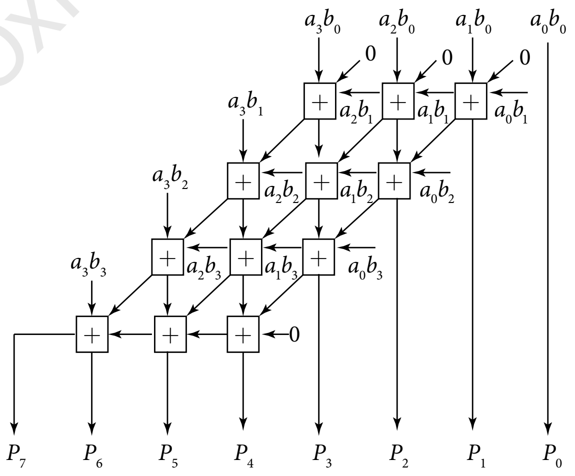


FIGURE 1.53 Unsigned array multiplier

1.4.19 Programmable Logic Device

Programmable logic devices (PLDs) are standard products but can be programmed to function in a specific application. The programming can be done either by end user or by the manufacturer. The PLDs that are programmed by the manufacturer are known as mask-programmable logic devices (MPLDs). The PLDs that are programmed by the end user are called field-programmable logic devices (FPLDs). The architecture of PLDs is very regular and fixed. It cannot be changed by the end user. The PLDs have wide range of applications and have low risk and cost in manufacturing in large volume. Hence, the PLDs are cheaper. As the PLDs are pre-manufactured, tested, and placed in inventory in advance, the design cycle time is very short. The PLDs are classified into three categories based on the architecture and programmability. They are given as follows:

1. Read-only Memory (ROM)
2. Programmable Logic Array (PLA)
3. Programmable Array Logic (PAL)

This section describes the architecture of PLDs and their applications.

Read-only Memory

Read-only memory is a storage device, which can be programmed once. Once it is programmed, the data remains intact and can be read as many times as possible. The stored data is not lost even if the power is removed, unlike random access memory (RAM). The structure of a ROM is shown in Fig. 1.54.

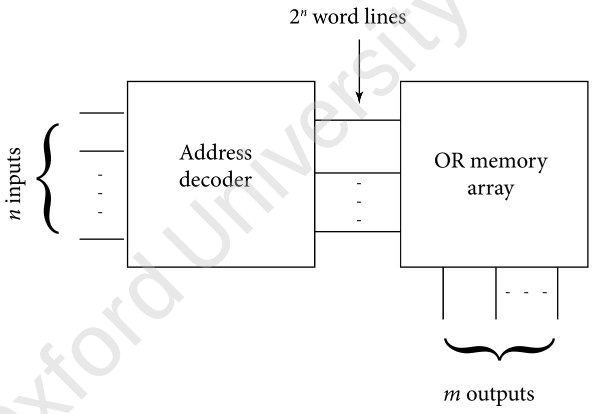


FIGURE 1.54 $2^n \times m$ ROM architecture

It consists of an address decoder with n input lines and programmable OR array with m output lines. The decoder produces minterms based on the n input lines. The minterms are ORed through programmable switches, which can be made ON or OFF to select a particular minterm. The programmable switches can be implemented by either bipolar, CMOS, nMOS, or pMOS technologies.

EXAMPLE 1.13 Design a combinational circuit using ROM that takes 3-bit number and produces outputs as the square of the input numbers.

Solution

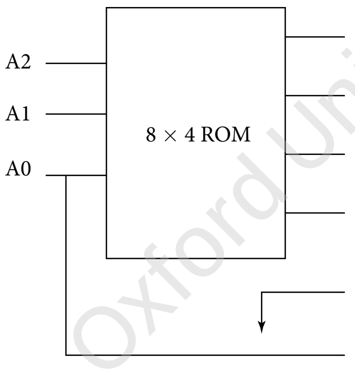
Let us first derive the truth table of the combinational circuit that takes 3-bit number and produces its square as the output.

The three input bits are $A2$, $A1$, and $A0$, which can have at most eight combinations starting from 000 to 111. The maximum value of input is 7 when squared the result is 49. Therefore, the maximum decimal equivalent value is 49, which require six bits for representation. Hence, the combinational circuit would require at most six output bits, which are represented as $Y5$, $Y4$, $Y3$, $Y2$, $Y1$, and $Y0$. The truth table of the circuit is shown in Table 1.29.

TABLE 1.29 Truth table of the circuit of Example 1.13

Inputs				Outputs						
*A*2	*A*1	*A*0	Decimal	*Y*5	*Y*4	*Y*3	*Y*2	*Y*1	*Y*0	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	1	1
0	1	0	2	0	0	0	1	0	0	4
0	1	1	3	0	0	1	0	0	1	9
1	0	0	4	0	1	0	0	0	0	16
1	0	1	5	0	1	1	0	0	1	25
1	1	0	6	1	0	0	1	0	0	36
1	1	1	7	1	1	0	0	0	1	49

Out of six output bits, two bits $Y1$ and $Y0$ can be implemented directly, as $Y1$ is always zero and $Y0$ is same as input $A0$. The remaining four bits $Y5$, $Y4$, $Y3$, and $Y2$ can be implemented using an 8×4 ROM as shown in Fig. 1.55.



(a)

A2 A1 A0	Y5 Y4 Y3 Y2
0 0 0	0 0 0 0
0 0 1	0 0 0 0
0 1 0	0 0 0 1
0 1 1	0 0 1 0
1 0 0	0 1 0 0
1 0 1	0 1 1 0
1 1 0	1 0 0 1
1 1 1	1 1 0 0

(b)

FIGURE 1.55 Implementation of the combinational circuit of Example 1.13, (a) Simplified form of the circuit using ROM (b) ROM truth table

Programmable Logic Array

Programmable logic array (PLA) is an integrated circuit chip used for two-level combinational logic circuits. It consists of an AND array followed by an OR array. Both the AND array and the OR array are programmable. The architecture of PLA is shown in Fig. 1.56.

The AND array, also called AND plane, implements the product terms and the OR array, also called OR plane, implements the sum of product (SOP) terms. In PLA, both the arrays are programmable. PLA has limited number of product terms, not the minterms. Hence, to implement a logic using PLA, a minimal SOP form should be derived so that the logic can be implemented using the available product terms.

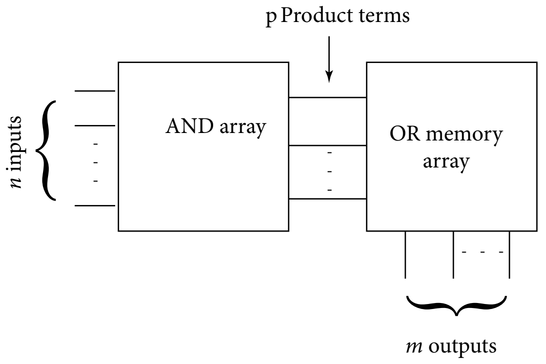


FIGURE 1.56 PLA architecture in block diagram

EXAMPLE 1.14 Implement the following Boolean functions using a PLA.

$$F_1 = A\overline{B} + AC + BC \tag{1.69}$$

$$F_2 = \overline{A}B\overline{C} + AC \tag{1.70}$$

Solution

In the given functions, there are four AND terms: $A\overline{B}$, AC, BC, and $\overline{A}B\overline{C}$. These AND terms can be implemented in the AND plane of PLA. The AND terms are then ORed to implement the function F_1 and F_2. To implement the product terms of the functions F_1 and F_2, the AND plane is programmed as shown in Fig. 1.57. The AND terms are then ORed by the OR array program as shown in Fig. 1.57. The cross-point having 'X' indicates an electrical connection between the horizontal and the vertical wires.

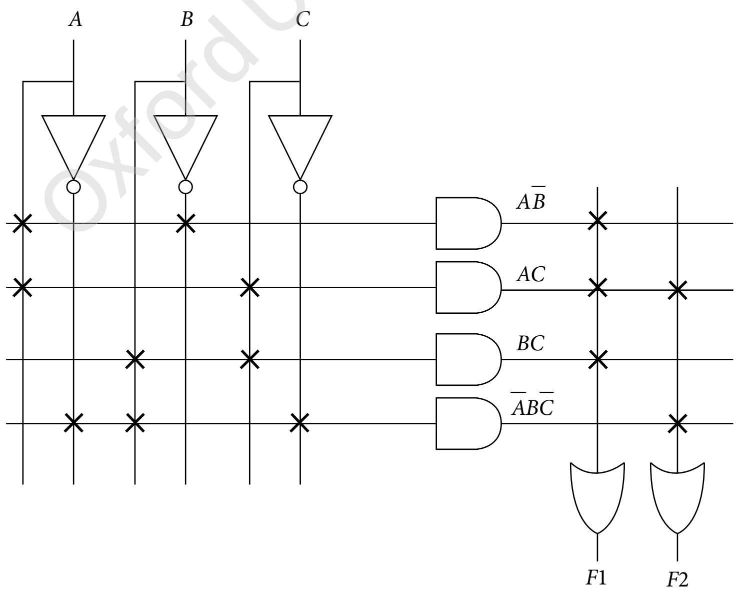


FIGURE 1.57 PLA architecture in block diagram

EXAMPLE 1.15 Implement the following Boolean functions using PLA.

$$Sum(A, B, C_{in}) = \sum m(1,2,4,7) \tag{1.71}$$

$$C_{out}(A, B, C_{in}) = \sum m(3,5,6,7) \tag{1.72}$$

Solution

The Boolean expressions for the given functions can be written as follows:

$$Sum = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + ABC_{in} \tag{1.73}$$

$$C_{out} = AB + AC_{in} + BC_{in} \tag{1.74}$$

The functionality of a PLA to implement these functions can be represented in the form of a table as shown in Table 1.30.

TABLE 1.30 PLA implementation table

Product terms	Inputs			Outputs	
	A	B	C_{in}	Sum	C_{out}
$\overline{A}\,\overline{B}C_{in}$	0	0	1	1	0
$\overline{A}B\overline{C_{in}}$	0	1	0	1	0
$A\overline{B}\,\overline{C_{in}}$	1	0	0	1	0
ABC_{in}	1	1	1	1	0
AB	1	1	-	0	1
AC_{in}	1	-	1	0	1
BC_{in}	-	1	1	0	1

Inputs are represented by 1 for true form, 0 for complement form, and - for don't care. The outputs are represented by 1 if the term is present in the function and 0 if the term is absent in the function. A simplified form of PLA is shown in Fig. 1.58.

The connection between a vertical and horizontal line is represented by a cross-point 'X'.

Programmable Array Logic

Programmable array logic (PAL) is another class of PLD with AND array followed by OR array, where the AND array is programmable but the OR array is fixed. The PAL architecture is shown in Fig. 1.59.

The OR array has permanently programmed connections as shown by dots in Fig. 1.59. The OR plane cannot be programmed. In the aforementioned PAL architecture, each OR gate has two inputs; hence, the SOP must have two product terms. Remember unlike PLA the product terms cannot be shared between the OR gates. Each function must be simplified individually to reduce the product terms to maximum two. If the SOP expression contains more than two product terms, each OR gate can be used to implement the function partially and then summed using additional OR gate to implement the complete function. The following example illustrates the implementation of Boolean functions using PAL.

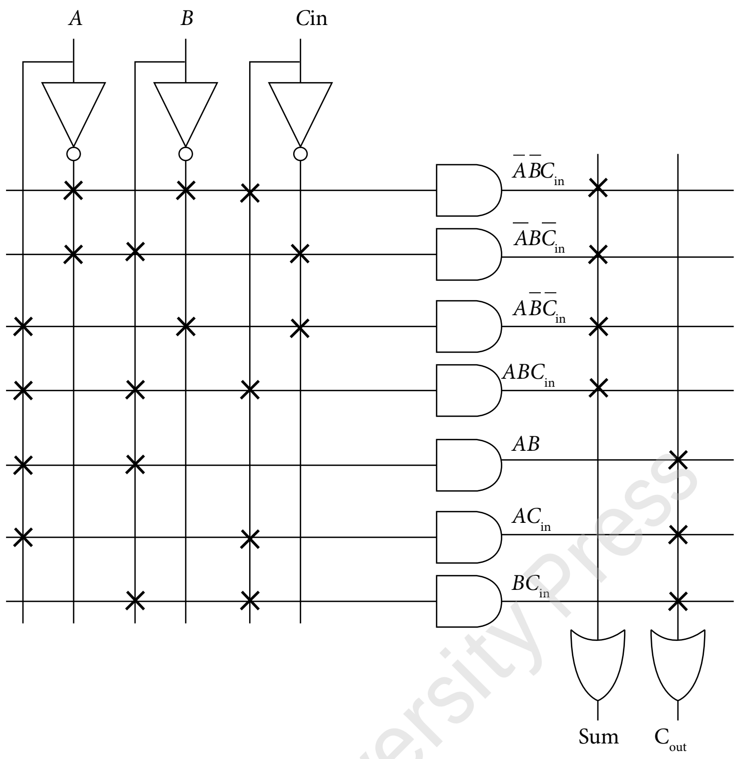


FIGURE 1.58 Simplified form of PLA implementing Boolean functions for Sum and C_{out}

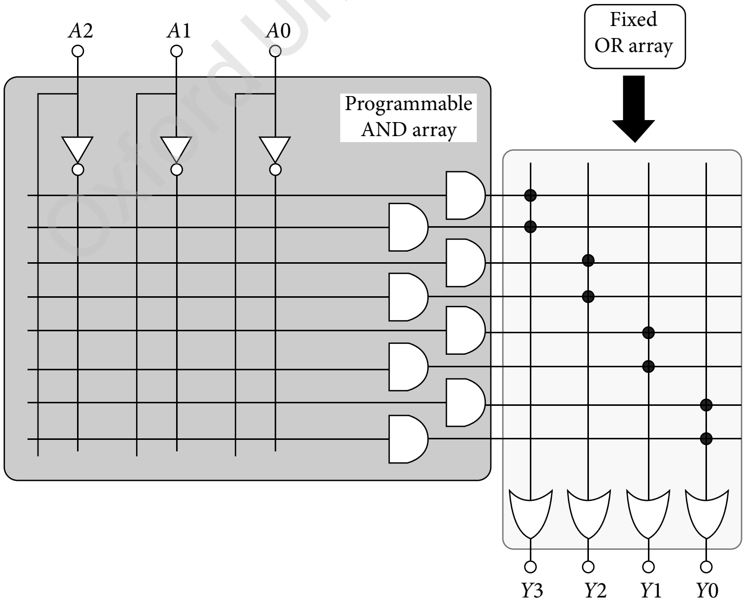


FIGURE 1.59 PAL architecture

EXAMPLE 1.16 Implement the following Boolean logic using PAL.

$$F1(A, B, C) = \sum m(1, 3, 4, 5, 6, 7) \tag{1.75}$$

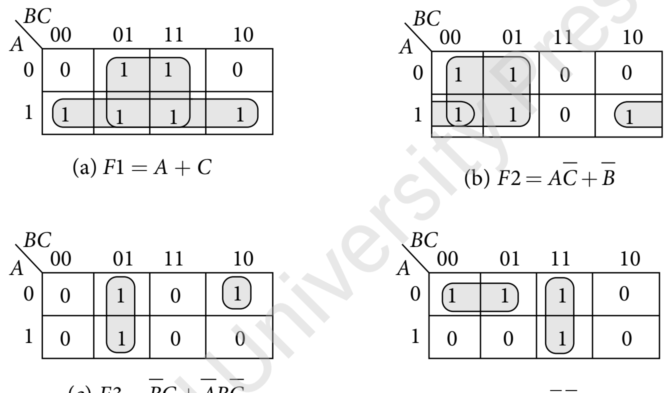
$$F2(A, B, C) = \sum m(0, 1, 4, 5, 6) \tag{1.76}$$

$$F3(A, B, C) = \sum m(1, 2, 5) \tag{1.77}$$

$$F4(A, B, C) = \sum m(0, 1, 3, 7) \tag{1.78}$$

Solution

Let us first find out a minimum SOP form of the given function using the Karnaugh map method. The K-maps and the corresponding minimum SOP forms are shown in Fig. 1.60.



(a) $F1 = A + C$

(b) $F2 = A\overline{C} + \overline{B}$

(c) $F3 = \overline{B}C + \overline{A}B\overline{C}$

(d) $F4 = \overline{A}\,\overline{B} + BC$

FIGURE 1.60 K-map minimization of functions of Example 1.16

We can see that each Boolean function has two product terms. Hence, the simple three-input four-output PAL architecture as shown in Fig. 1.59 can be used to implement the four Boolean functions. The PAL implementation is shown in Fig. 1.61. The AND plane is programmed to generate the product terms. The connections between the vertical and the horizontal lines in the AND plane are represented by '*X*'.

In ROM-based design, the addition of input signal increases the ROM size by two times. This in turn doubles the size of the AND and OR array. But in case of PLA or PAL additional input can easily be accommodated without doubling the size. Commercially available PAL can have at most 22 input lines.

1.4.20 Sequential PLD

The PLDs that we have discussed contain only combinational logic gates but no sequential elements or flip-flops. But digital systems are to be designed using both combinational and sequential circuits. Hence, to implement sequential programmable devices, flip-flops must be used externally with PLDs. In order to avoid the external use of flip-flops, the sequential PLDs (SPLDs) are developed with D or JK flip-flops. The sequential PLD is also known as simple PLD or simply SPLD. The SPLD architecture is mostly based on combinational PAL and DFF. The section of a SPLD that implements one SOP output through register is known as macrocell. A macrocell is shown in Fig. 1.62.

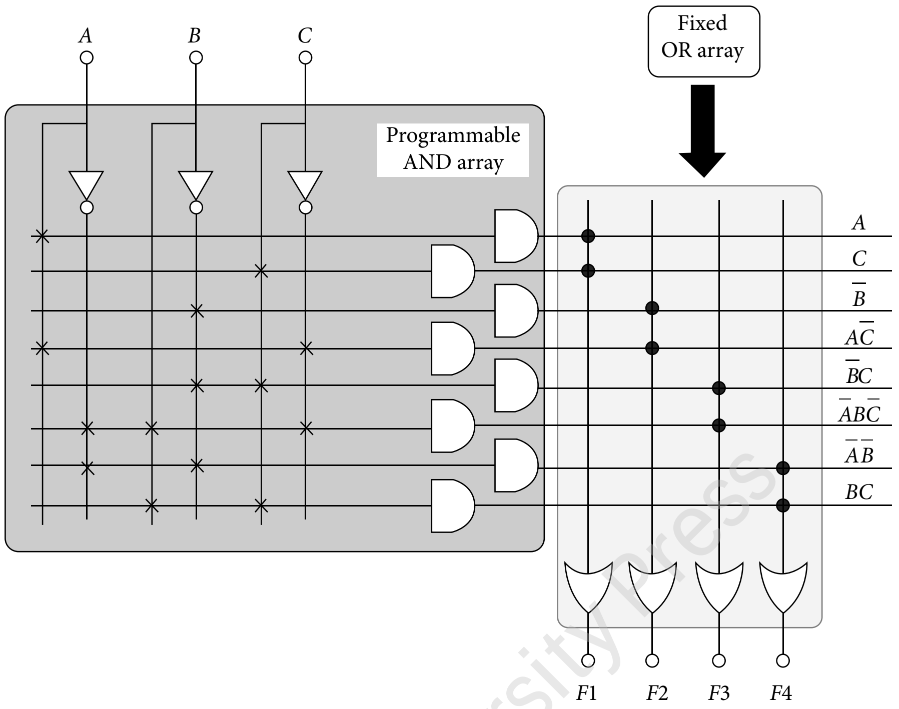


FIGURE 1.61 PAL implementation of Boolean functions of Example 1.16

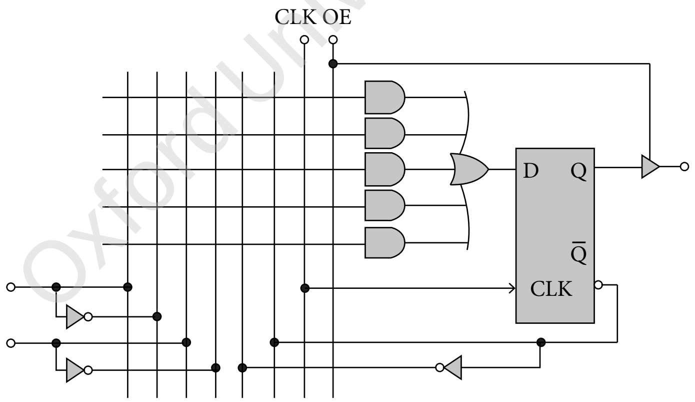


FIGURE 1.62 Typical macrocell architecture

The AND–OR array is similar to the PAL architecture. The output of AND–OR array is passed through a DFF triggered by a clock signal CLK. The final output is available through a tri-state buffer controlled by output-enabled signal OE. The true and complemented form of the output signal is fed back to the input of the AND array. This provides the previous state of the output signal. A typical SPLD chip has 8–10 macrocells.

1.4.21 Keypad Scanner

Keypad scanner is used to enter data manually in different electronic systems, such as digital telephone, computer keyboard, and different embedded systems developed using microprocessor and microcontrollers. We shall discuss the design of a keypad scanner. The schematic of the keypad is shown in Fig. 1.63. It has total 12 numbers of keys: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *, 0, and #. There are four row lines: R0, R1, R2, and R3, and three column lines: C2, C1, and C0. When a key is pressed, a connection is established between the corresponding row and column lines. All the row lines are connected to ground through resistors. When no key is pressed, all the row lines are connected to ground, that is, $R0 = R1 = R2 = R3 = 0$. When a column line is pulled to high voltage, if any key

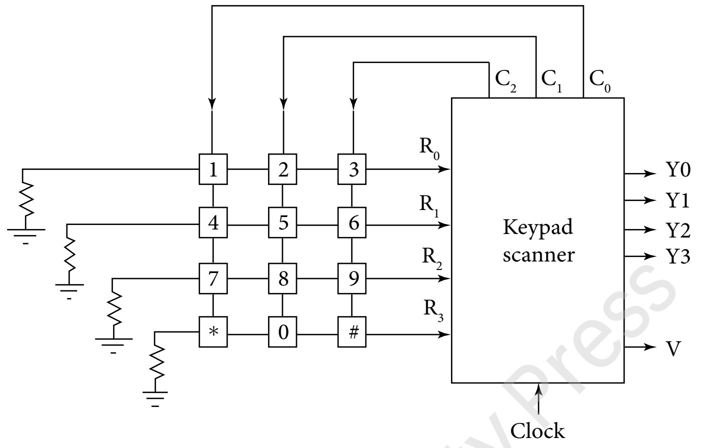


FIGURE 1.63 Schematic of a keypad scanner

is pressed in that column, the corresponding row will be pulled to high voltage and the other rows will remain at 0 V. For example, let the column line C0 is pulled to high voltage and key '1' is pressed. In this case, row *R*0 will be high and the other rows *R*3, *R*2, and *R*1 will remain at 0 V. So the scanner will detect that key '1' is pressed. Table 1.31 shows the input–outputs of the keypad scanner.

TABLE 1.31 Keypad scanner input–outputs

Key	*R3 R2 R1 R0*	*C0 C1 C2*	Code (*Y3Y2Y1Y0*)
1	0001	100	0001
2	0001	010	0010
3	0001	001	0011
4	0010	100	0100
5	0010	010	0101
6	0010	001	0110
7	0100	100	0111
8	0100	010	1000
9	0100	001	1001
*	1000	100	1010
0	1000	010	0000
#	1000	001	1011

The keypad scanner performs three operations. Firstly, it detects whether a key is pressed. Secondly, it identifies which key is pressed. Then it generates a unique code for the key that is pressed.

At first, it pulls up all the column lines to high and check if any key is pressed by sensing the row lines. If no key is pressed, all the row lines will remain at 0 V which indicates that no key is pressed. If any key is pressed, any one of the four row lines will be pulled high that means a valid key is pressed. So it will set $V = 1$. To identify which key is pressed, it pulls up the column lines one by one and detects the key. The signal V indicates that the generated code is valid. It remains high for one clock cycle to indicate a valid code in the output signals: $Y3$, $Y2$, $Y1$, and $Y0$.

1.4.22 Features of PLD

The biggest advantage of using PLDs is that it reduces the total cost of the system. The design cycle time using PLDs is very fast and therefore the time-to-market of the final product is less. The risks associated in the product development using PLDs are also less. Any last minute change can easily be accommodated without redesigning the circuit boards. The cost involved in printed circuit board (PCB) design, assembly, test, and repair is also very less when PLDs are used, as the design usually requires fewer components.

The features of the PLDs are also enhanced to accommodate multitude of designs. They are capable of performing control functions, bus interface, memory interface, and DSP. They have grown in density, variety, and complexity. The state-of-the-art PLDs can handle designs of hundreds of thousands of gates to even a million gates. It is being used to design larger portions of the system, even in some case the entire systems on chip (SoC). Therefore, the PLDs have a great future in days to come.

1.4.23 One/Zero Detector

A zero-detector circuit detects if all the input bits are logic 0. Similarly, a one-detector circuit detects if all the input bits are logic 1. Figure 1.64 shows the schematic of one and zero detectors.

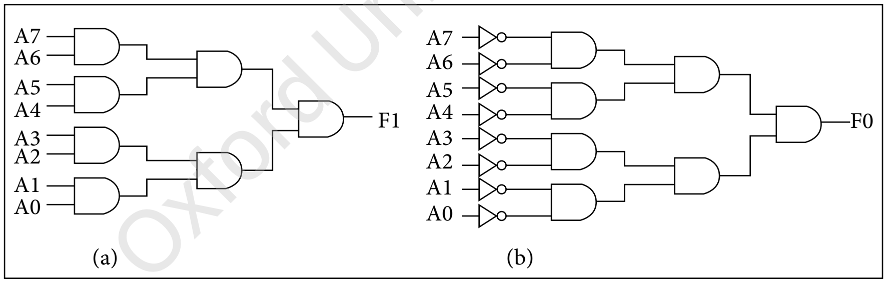


FIGURE 1.64 (a) One detector (b) Zero detector

In one detector, when all inputs are at logic 1, the output is 1. Otherwise, the output is zero. Thus, it detects all ones at the input. In zero detector, when all inputs are at logic 0, the output is 1. Thus, it detects all zeros at the input.

1.4.24 Barrel Shifter

The one-bit shifter can shift data by one bit position in one clock cycle. But it is often required to shift data by more than one bit position in one clock cycle in DSP or general-purpose processors. The multi-bit shift is achieved using a barrel shifter. The advantage of the barrel shifter is that any bit in a word can connect to any other bit, so that large bit shifts can be completed in a single operation.

Let us consider an 8-bit barrel shifter as shown in Fig. 1.65. It has one 8-bit data input A and 8-bit data output B, with a 3-bit control input B. Table 5.4 illustrates the data shifting in an 8-bit barrel shifter.

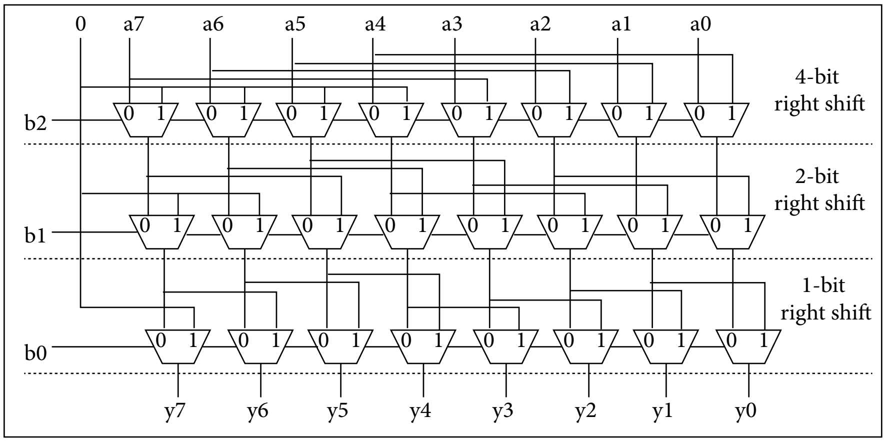


FIGURE 1.65 Schematic of an 8-bit barrel shifter

TABLE 1.32 Shifting operations of an 8-bit barrel shifter

Sl. No.	Inputs			Outputs								Shift operation
	*b*2	*b*1	*b*0	*y*7	*y*6	*y*5	*y*4	*y*3	*y*2	*y*1	*y*0	
1	0	0	0	a7	a6	a5	a4	a3	a2	a1	a0	No shift
2	0	0	1	0	a7	a6	a5	a4	a3	a2	a1	Shift right by 1 bit
3	0	1	0	0	0	a7	a6	a5	a4	a3	a2	Shift right by 2 bit
4	0	1	1	0	0	0	a7	a6	a5	a4	a3	Shift right by 3 bit
5	1	0	0	0	0	0	0	a7	a6	a5	a4	Shift right by 4 bit
6	1	0	1	0	0	0	0	0	a7	a6	a5	Shift right by 5 bit
7	1	1	0	0	0	0	0	0	0	a7	a6	Shift right by 6 bit
8	1	1	1	0	0	0	0	0	0	0	a7	Shift right by 7 bit

1.5 SEQUENTIAL LOGIC CIRCUITS

The sequential circuits are the type of digital circuits where the present output states depend on the present input states as well as on the past output states. In other words, in the sequential circuits, the outputs are function of the input logic levels as well the time when the inputs were applied. Let us understand the concept of temporal dependency of the outputs by the following circuit shown in Fig. 1.66.

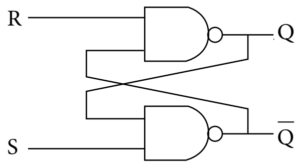


FIGURE 1.66 Example of a sequential circuit using NAND gates

In this digital circuit, there are two inputs *R* and *S* and two outputs *Q* and $\overline{Q}$. When both the inputs are at logic 0, both the outputs are at logic 1. When $R = 0$ and $S = 1$, the output $Q = 1$, and the output $\overline{Q} = 0$.

Alternately, when $R = 1$ and $S = 0$, the output $Q = 0$, and the output $\overline{Q}$= 1. But when both the inputs are at logic 1, the outputs cannot be decided directly. In this condition, the outputs will be decided based on their previous logic levels. Let us assume, previously $Q = 0$ and $\overline{Q}$= 1. Now the inputs are $R = S = 1$. Since, previously $Q = 0$, $\overline{Q}$ will remain at logic 1, and as $\overline{Q}$ remains at logic 1, Q will remain at logic 0. Therefore, the outputs hold their previous states. Similarly, if the outputs were $Q = 1$ and $\overline{Q}$= 0, now also they will remain as they are if the inputs are $R = S = 1$. The functionality of this circuit is illustrated in Table 1.33.

TABLE 1.33 Truth table of SR flip-flop

Inputs		**Previous outputs**		**Present outputs**		**State of the FF**
S	R	Q	$\overline{Q}$	Q	$\overline{Q}$	
0	0	X	X	1	1	Not allowed
0	1	X	X	0	1	Reset
1	0	X	X	1	0	Set
1	1	0	1	0	1	Hold
1	1	1	0	1	0	Hold
X— indicates don't care condition						

This example introduces the main concepts of the sequential circuit, that is, the temporal dependency of the outputs. For the present input conditions $S = R = 1$, the outputs depend on whether the past states were 10 or 01.

In order to maintain the temporal dependency of the input and output states, an extra input, which is known as clock (CLK). Depending on whether the inputs are in synchronous with the clock input, the sequential circuits are classified into two types: synchronous and asynchronous.

1.5.1 SR Flip-flop

Generally, a flip-flop is a sequential circuit with one or more inputs and two complementary outputs Q and $\overline{Q}$. SR flip-flop is a flip-flop with two inputs S and R, where S indicates set and R indicates reset. When the input $S = 1$ and $R = 0$, the flip-flop outputs $Q\overline{Q}$ = 10 and when the input $S = 0$ and $R = 1$, the flip-flop outputs $Q\overline{Q}$ = 01. Figure 1.67 shows the symbol of SR flip-flop.

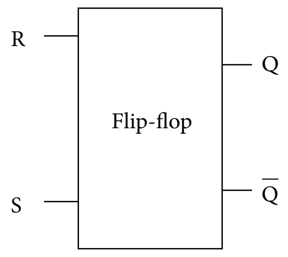


FIGURE 1.67 Symbol of SR flip-flop

The SR flip-flop can be implemented either using NAND gates or using NOR gates. The NAND-based SR flip-flop is shown in Fig. 1.66. The NOR-based SR flip-flop is shown in Fig. 1.68.

The operation of the SR flip-flop using NOR gates is illustrated in Table 1.34.

The SR flip-flop is known as SR latch as the data is latched in the logic circuit for some input combinations.

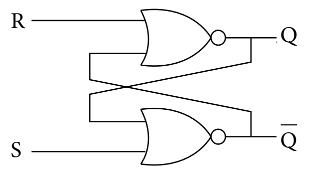


FIGURE 1.68 SR flip-flop using NOR gates

TABLE 1.34 Truth table of SR flip-flop

Inputs		Previous outputs		Present outputs		State of the FF
S	*R*	*Q*	$\overline{Q}$	*Q*	$\overline{Q}$	
0	0	1	0	1	0	Hold
0	0	0	1	0	1	Hold
0	1	*X*	*X*	0	1	Reset
1	0	*X*	*X*	1	0	Set
1	1	*X*	*X*	0	0	Not allowed
X—indicates don't care condition						

1.5.2 SR Flip-flop with Clock Input

The SR flip-flop explained before works asynchronously. There is no clock input. This makes the flip-flop outputs to change anytime the input changes. This is OK when the flip-flop is operated separately. However, in a digital circuit, a flip-flop alone does not work. There are plenty of other gates both combinational and sequential. The inputs of a particular SR flip-flop come from the outputs of some other logic. As the inputs come from different paths, they encounter different path delays and reach at the inputs at different time. As we have seen before, the output changes anytime the input changes in a flip-flop, which causes incorrect output to be latched/stored in the flip-flop.

Let us consider an examination hall where each student coming to the hall in different time. If we allow the examination to start for each student as they come in, there will be a total chaos. In order to make sure that the examination happens properly, we fix up a time at which all students must reach to the examination hall. Therefore, we need a clock and the event must be synchronized with the clock.

In the digital circuits also, we must use a clock to ensure proper operations of sequential circuit. Let us illustrate this with the following example shown in Fig. 1.69.

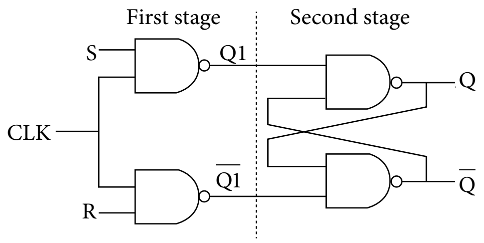


FIGURE 1.69 Clocked SR flip-flop

When the clock input CLK = 0, the outputs of the NAND gates in the first stage ($Q1$ and $\overline{Q1}$) are at logic 1. This makes outputs of NAND gates in the second stage to hold their previous states. In this condition, the outputs Q and $\overline{Q}$ do not change their values even if the inputs R and S change their values.

When the clock input CLK = 1, the outputs of the NAND gates at the first stage depends on the inputs R and S. If $S = 1$ and $R = 0$, then $Q1 = 0$ and $\overline{Q1} = 1$, which makes $Q = 1$ and $\overline{Q} = 0$. If $S = 0$ and $R = 1$, then $Q1 = 1$ and $\overline{Q1} = 0$, which makes $Q = 0$ and $\overline{Q} = 1$.

When $S = R = 0$, the outputs Q and $\overline{Q}$ hold their previous values.

When $S = R = 1$, both $Q1$ and $\overline{Q1}$ are at logic 0. This makes both outputs Q and $\overline{Q}$ to become 1. However, depending on which one becomes 1 first, the other will be decided. Therefore, this is a race condition between the outputs. Therefore, the outputs are termed as forbidden for the inputs $S = R = 1$.

1.5.3 JK Flip-Flop

The race condition of the SR flip-flop is overcome in the JK flip-flop. Figure 1.70 shows the JK flip-flop using NAND gates.

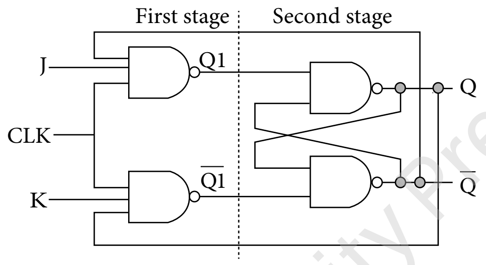


FIGURE 1.70 JK flip-flop

In the first stage, two 3-input NAND gates are used. The outputs Q and $\overline{Q}$ are fed back to the inputs at first stage. When CLK = 0, the outputs of the first stage $Q1$ and $\overline{Q1}$ are at logic 1. This makes the outputs of the flip-flop to hold their previous states. When CLK = 1, the outputs of the first stage $Q1$ and $\overline{Q1}$ depend on the inputs and previous outputs. Under CLK = 1 condition, the operation of the flip-flop is explained as follows:

1. When $J = K = 0$, $Q1 = \overline{Q1} = 1$, therefore, outputs Q and $\overline{Q}$ hold their previous states.
2. When $J = 0$, $K = 1$, $Q1 = 1$. If $Q = 1$, then $\overline{Q1} = 0$. This makes $Q = 0$ and $\overline{Q} = 1$.
3. When $J = 1$, $K = 0$, $\overline{Q1} = 1$. If $Q = 1$, then $\overline{Q} = 0$. This makes $Q = 1$ and $\overline{Q}$ remains at 0.
4. When $J = 1$ and $K = 1$, $Q1$ and $\overline{Q1}$ are decided based on Q and $\overline{Q}$. If we assume that $Q = 0$ and $\overline{Q} = 1$, then $Q1 = 0$ and $\overline{Q1} = 1$. This makes $Q = 1$ and $\overline{Q} = 0$. Thus, outputs toggle. If we assume that $Q = 1$ and $\overline{Q} = 0$, then $Q1 = 1$ and $\overline{Q1} = 0$. This makes $Q = 0$ and $\overline{Q} = 1$. Thus, also outputs toggle.

The operation of JK flip-flop is summarized in Table 1.35.

TABLE 1.35 Truth table of JK flip-flop

Inputs		Previous outputs		Present outputs		State of the FF
J	K	Q_n	$\overline{Q_n}$	Q_{n+1}	$\overline{Q_{n+1}}$	
0	0	0	1	0	1	Hold
0	0	1	0	1	0	Hold
0	1	X	X	0	1	Reset
1	0	X	X	1	0	Set
1	1	0	1	1	0	Toggle
1	1	1	0	0	1	Toggle

1.5.4 Master-slave Flip-flop

When $J = K = 1$ in JK flip-flop, the output toggles. As long as the clock pulse is at logic 1 state, the outputs keep toggling and at the end of clock pulse the outputs become unpredictable. This is known as race-around condition. One way to solve this problem is by reducing the duration of clock pulse less than the delay of the flip-flop. However, it is very difficult to design such a clock pulse with a very low duty cycle. The other way to solve the race-around condition is by using master-slave flip-flop.

Two SR flip-flops are taken in cascade to form the master-slave flip-flop as shown in Fig. 1.71. The first flip-flop is called master, whereas the second one is called slave. The clock signal (CLK) directly goes to the clock input of master flip-flop and inverted clock signal ($\overline{CLK}$) goes to the clock input of slave flip-flop.

When clock signal is at logic high level (CLK = 1), the outputs of the master Q_m and ($\overline{Q_m}$) are determined depending on its inputs S_m and R_m. Under this condition, the outputs of the slave Q_s and ($\overline{Q_s}$) do not change. When clock signal goes to logic low level or ($\overline{CLK}$)=1, the outputs of slave are determined by its inputs S_s and R_s. Under this condition, the outputs of the master Q_m and ($\overline{Q_m}$) do not change.

Therefore, the master-slave flip-flop only changes its final outputs when clock signal goes from logic high to low level. Thus, it is called edge-triggered flip-flop. The master-salve flip-flop shown in Fig. 1.71 is a negative edge-triggered flip-flop as it changes its output only when clock makes $1 \rightarrow 0$ transition. There is positive edge-triggered flip-flop also which changes its output when clock makes $0 \rightarrow 1$ transition.

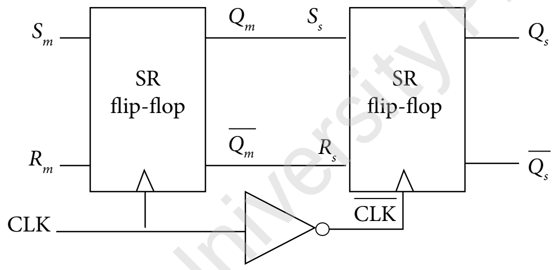


FIGURE 1.71 Master-slave flip-flop

1.5.5 D Flip-flop

A JK flip-flop is converted into a DFF or delay flip-flop with the configuration shown in Fig. 1.72. It has two inputs: one is clock input and the other one is D or data input.

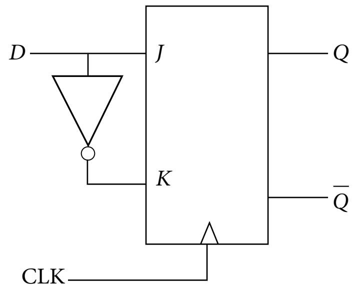


FIGURE 1.72 D flip-flop using JK flip-flop

We have seen that the Q output of JK flip-flop is logic 1 when $J = 1$ and $K = 0$, and is logic 0 when $J = 0$ and $K = 1$. In other words, we can say that $Q = J$ when $K = \overline{J}$. Thus, a NOT gate connected between J and K

input to make $K = \overline{J}$. When $D = 1, J = 1$, and $K = 0$ which makes $Q = 1$ and $\overline{Q}$=0. When $D = 0, J = 0$ and $K = 1$ which makes $Q = 0$ and $\overline{Q}$=1.

The purpose of the DFF is to work as a delay element as the output follows the input only after a delay. That is why it is also known as delay flip-flop.

1.5.6 T Flip-flop

A T flip-flop stands for toggle flip-flop. It has two inputs: one is clock input (CLK) and the other one is T input. It is also constructed from JK flip-flop as shown in Fig. 1.73.

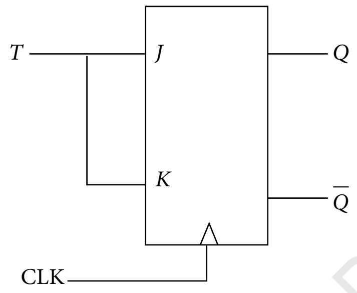


FIGURE 1.73 T flip-flop using JK flip-flop

In a JK flip-flop when both the inputs are in same logic level, that is, $J = K$, the outputs either remain in the same state or complement themselves. When $J = K = 1$, the outputs toggle, that is, $Q_{n+1} = \overline{Q_n}$. When $J = K = 0$, the outputs hold their previous state, that is, $Q_{n+1} = Q_n$. Therefore, a JK flip-flop is converted into a T flip-flop by shorting its two inputs.

1.5.7 Flip-flop Characteristics

A flip-flop is characterized by the following parameters:

Propagation Delay

It is also known as Clock-to-Q time delay, which means the propagation time delay between the clock signal and the Q output.

Set-up Time

It is the minimum time before the clock edge the inputs must arrive. Let us consider an example of examination hall to explain the concept of set-up time. Suppose an examination hall where the examination will start at 10 AM. The examinees must enter the examination hall 30 minutes before the examination starts and will not be allowed to leave the hall. So 30 minutes is the set-up time for the examination. Nobody will be allowed to appear in the examination if he/she comes after 9:30 AM. Similarly, in case of a flip-flop, the data must arrive before the clock pulse by a specified period of time and must not change their values.

Hold Time

It is the minimum time after the clock edge and the inputs must not change their values.

Maximum Clock Frequency

It is the maximum frequency of the clock signal that can be applied to the flip-flop. It depends on the propagation delay and set-up time.

Asynchronous Active Pulse Width

The minimum pulse width of the asynchronous inputs like preset and clear input signals.

Figure 1.74 illustrates propagation delay, set-up time, and hold time of a flip-flop.

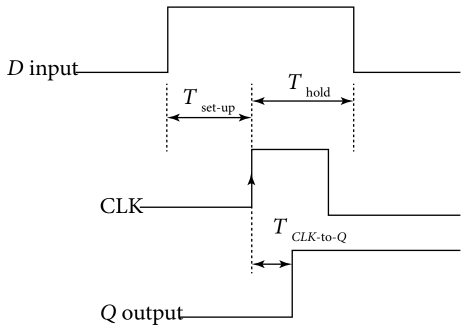


FIGURE 1.74 Illustration of propagation delay, set-up time, and hold time

1.5.8 Registers

A flip-flop can store or register a single bit. Therefore, a flip-flop is known as one-bit register. When a number of flip-flops connected in cascade, a multi-bit register is formed. In an N-bit register, there is N number of flip-flops. In an N-bit register, it is often required to shift data from one register to another. An array of flip-flops that allows the shifting of data is called a shift register.

1.5.9 Shift Register

A shift register is a chain of flip-flops where the input data is propagated through the chain by applying the clock pulses.

A 4-bit shift register is shown in Fig. 1.75. There are four positive edge-triggered DFFs connected in series. The clock input is common to all the flip-flops. At the positive edge of the clock signal, the input signal goes to the output of the first flip-flop, the output of the first flip-flop goes to the output of the second flip-flop; the output of the second flip-flop goes to the output of the third flip-flop, and so on. Figure 1.76 shows the shifting of data with the clock pulse.

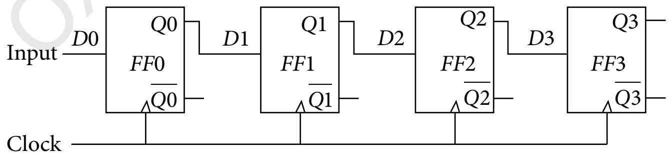


FIGURE 1.75 4-Bit shift register

Shift registers are generally of the following four types:

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

Clock pulse	Input bits	Q0	Q1	Q2	Q3
0	0 1 1 0 1	0	0	0	0
1		1	0	0	0
2		0	1	0	0
3		1	0	1	0
4		1	1	0	1
5		0	1	1	0
6			0	1	1
7				0	1
8					0

FIGURE 1.76 Propagation of data in a SISO shift register

1.5.10 SISO Shift Register

In a SISO shift register, data is entered at one edge of the chain of shift registers and is retrieved at the other end. An example of SISO shift register is depicted in Fig. 1.76.

1.5.11 PISO Shift Register

In a PISO shift register, data is entered to all the flip-flops in parallel but is retrieved serially at the output end.

A 4-bit PISO shift register with load and shift capability is shown in Fig. 1.77. It has four parallel data input lines, one *Shift*/$\overline{\textit{Load}}$ control line, clock input, and one serial data output line.

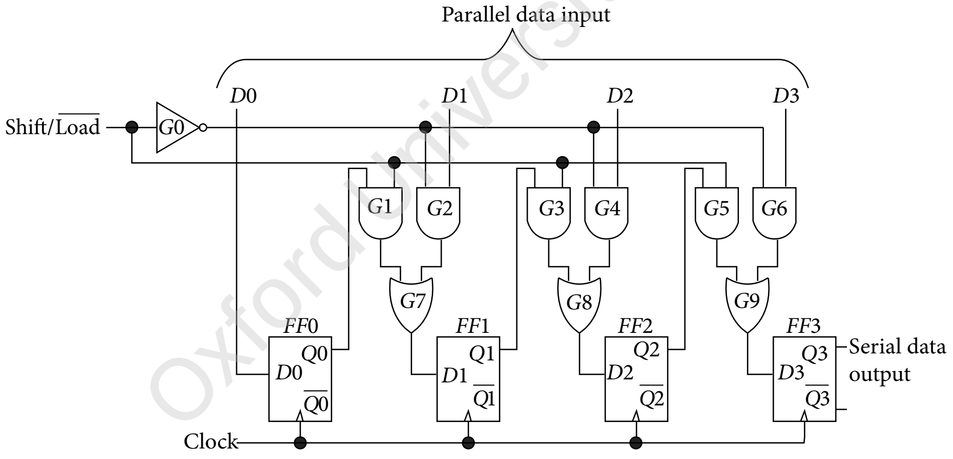


FIGURE 1.77 PISO shift register

Shift Operation

When the *Shift*/$\overline{\textit{Load}}$ control line is held at logic 1, AND gates, *G*1, *G*3, and *G*5, are enabled. With the clock pulse, data from *Q*0 goes to *D*1, *Q*1 goes to *D*2, and *Q*2 goes to *D*3. Therefore, the data is shifted serially through the chain of flip-flops.

Load Operation

When the *Shift*/$\overline{\textit{Load}}$ control line is held at logic 0, AND gates, *G*2, *G*4, and *G*6, are enabled. The four data inputs, *D*0, *D*1, *D*2, and *D*3, are applied to the data input of flip-flops, and with the clock pulse, the data is loaded into the register.

1.5.12 PIPO Shift Register

The PIPO shift register has a set of parallel data input lines and a set of parallel data output lines. The input data is entered in parallel into the register. After one clock pulse, the data is shifted at the output in parallel. An example of 4-bit PIPO shift register is shown in Fig. 1.78.

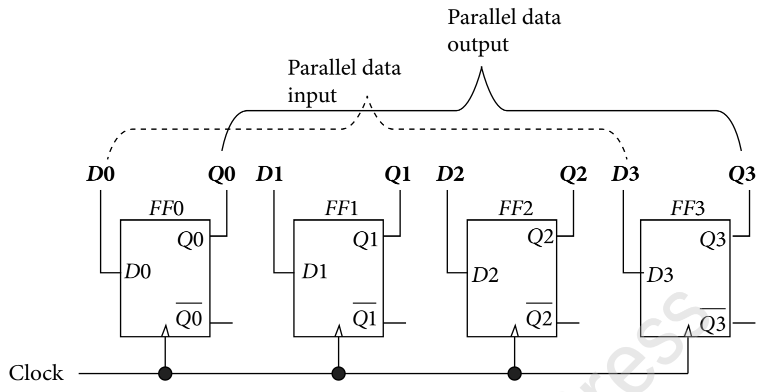


FIGURE 1.78 PIPO shift register

All four flip-flops are operated by a common clock. Four input bits *D*0 through *D*3 are applied to flip-flops *FF*0 through *FF*3, and the outputs *Q*0 through *Q*3 are taken out in parallel. When the clock pulse is applied, four inputs are stored in the flip-flops simultaneously and are available simultaneously at the outputs.

1.5.13 SIPO Shift Register

In SIPO shift register, input data is entered serially but the outputs are taken in parallel. A 4-bit SIPO shift register is shown in Fig. 1.79.

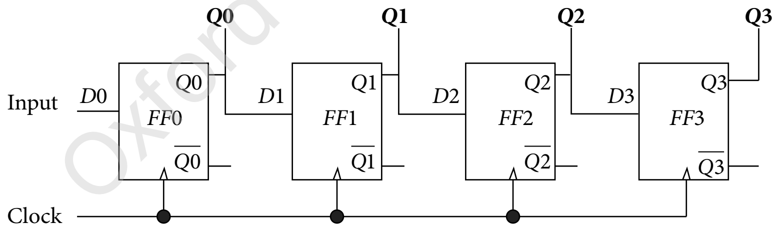


FIGURE 1.79 SIPO shift register

It works very much like a SISO shift register. Only difference is that the outputs of every flip-flop are available as primary output.

1.5.14 Counters

Counter is a sequential circuit that undergoes a sequence of predefined states by the application of clock pulses. They are constructed using flip-flops and other combinational logic gates. The number of states of a counter is determined by the number of flip-flops used to design the counter. The following are the two types of counters:

1. Asynchronous counter: In this counter, the external clock is applied to only the first flip-flop and the clock inputs of rest of the flip-flops are fed by the output of the preceding flip-flops. Asynchronous counter is also known as ripple counter.

2. Synchronous counter: In this counter, all the flip-flops are operated simultaneously/synchronously by the external clock signal.

1.5.15 Asynchronous/Ripple Counter

A 3-bit ripple counter is shown in Fig. 1.80. It is constructed using three positive edge triggered JK flip-flops. The *J* and *K* inputs are held at logic 1 to keep the flip-flops in toggle mode. In each positive transition in their clock input each flip-flop just toggles its output. The external clock is applied to the first flip-flop *FF*0 that toggles its output in every positive edge of the clock pulse as shown in Fig. 1.81.

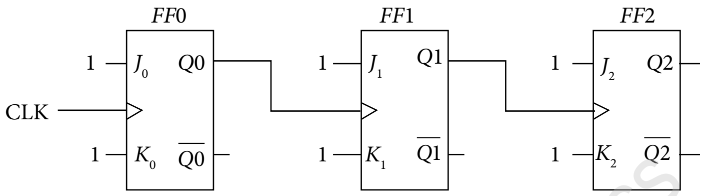


FIGURE 1.80 Three-bit ripple counter

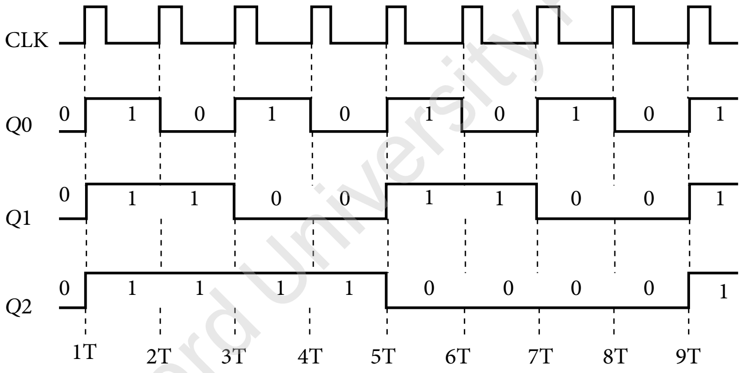


FIGURE 1.81 Waveforms of 3-bit ripple counter

Initially, let us assume that all the flip-flops are at reset state, that is, $Q2Q1Q0$ = 000. Now we apply the external clock pulse at the clock input of the first flip-flop. In the first positive edge (at time $1T$), *FF*0 changes its output *Q*0 from logic 0 to logic 1. As *Q*0 is connected to clock input of the second flip-flop, it also changes its output *Q*1 from logic 0 to logic 1. Similarly, the third flip-flop also changes its output *Q*2 from logic 0 to logic 1. Hence, after the first positive edge of external clock, the state of the counter becomes $Q2Q1Q0$ = 111.

At time $2T$, *Q*0 goes from logic 1 to logic 0. But this $1 \rightarrow 0$ transition in *Q*0 does not trigger *FF*1, and *Q*1 remains at logic 1 and therefore *Q*2 also remains at logic 1. So after the second positive edge of the clock the state of the counter becomes $Q2Q1Q0$ = 110. After the third positive edge, *Q*0 goes back to logic 1 creating a $0 \rightarrow 1$ transition at clock input of *FF*1. Thus, *FF*1 toggles and *Q*1 becomes logic 0. But *Q*2 remains at logic 1.

After the third positive edge of the clock, the state of the counter becomes $Q2Q1Q0$ = 101. In this manner, the counter goes through a sequence of states till all the flip-flops go to reset state as shown in Fig. 1.82. After the eight clock pulses, the counter goes back to its initial state of 000. Thus, this counter is also known as mod-8 counter. The sequence of states indicates that this counter acts as a binary down counter, starting from 111 state down to 000 state.

This counter can be made to count in up direction if the clock input of the second flip-flop onwards is taken from the $\overline{Q}$ output of the preceding flip-flop instead *Q* output.

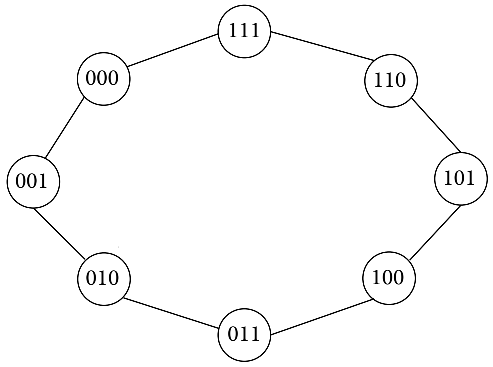


FIGURE 1.82 Sequence of states of 3-bit ripple counter

1. If flip-flops are positive edge triggered and Q output feeds the clock of the next flip-flop, the counter counts downward.
2. If flip-flops are positive edge triggered and $\overline{Q}$ output feeds the clock of the next flip-flop, the counter counts upward.
3. If flip-flops are negative edge triggered and Q output feeds the clock of the next flip-flop, the counter counts upward.
4. If flip-flops are negative edge triggered and $\overline{Q}$ output feeds the clock of the next flip-flop, the counter counts downward.

1.5.16 Synchronous Counter

Synchronous counter is another type of counter where all the flip-flops are operated by the same clock. The common clock triggers all the flip-flops simultaneously/synchronously. The state of the flip-flop is decided by the JK inputs. If $J = K = 0$, the flip-flops hold their previous states. If $J = K = 1$, the flip-flops toggle.

A 3-bit synchronous counter is shown in Fig. 1.83. There are three positive edge triggered JK flip-flops. The external clock signal is applied to all the flip-flops to be triggered simultaneously. The first flip-flop $FF0$ is in toggle mode as $J0 = K0 = 1$. Therefore, $FF0$ toggles in every positive edge of clock pulse. $FF1$ toggles only when $Q0$ becomes 1 and there is a positive edge of the clock pulse. $FF2$ toggles when both $Q0$ and $Q1$ are at logic 1 and there is a positive edge of the clock pulse.

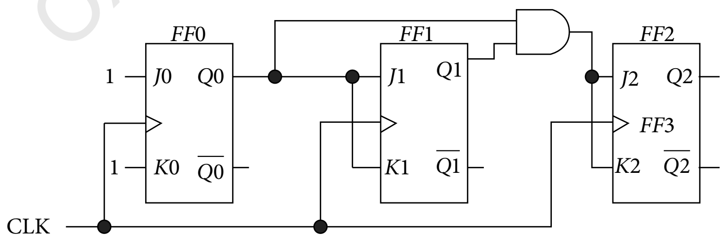


FIGURE 1.83 3-Bit synchronous counter

The timing diagram of the clock pulse and flip-flop outputs are shown in Fig. 1.84. Initially, let us assume that the flip-flops are in reset state, that is, $Q2Q1Q0$ = 000. After the first positive edge of clock pulse, $Q0$ becomes 1, and the counter goes from state 000 to 001. In the second positive edge of clock pulse, $Q1$ becomes 1 as $Q0$ was at logic 1, and now $Q0$ becomes 0. Counter goes to state 010. In this manner, the counter counts from 000 to 111. After the eighth clock pulse, it goes back to its initial state 000.

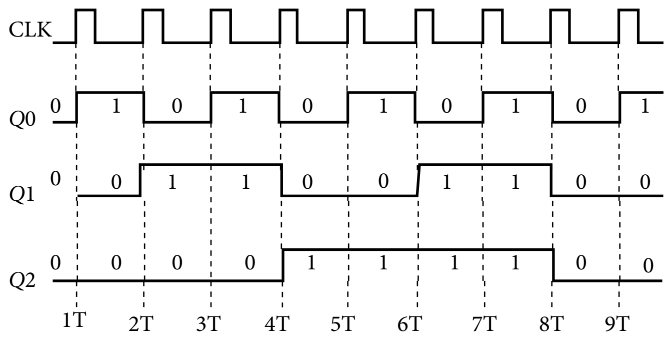


FIGURE 1.84 Waveforms of 3-bit synchronous counter

1.6 FINITE-STATE MACHINE

In general, a sequential circuit contains a number of flip-flops and some combinational logic gates. The sequential circuit undergoes a sequence of binary states. Therefore, sequential circuit is also known as finite-state machine (FSM). Figure 1.85 shows the structure of a FSM in general. It consists of two parts: a combinational logic portion and a bunch of flip-flops.

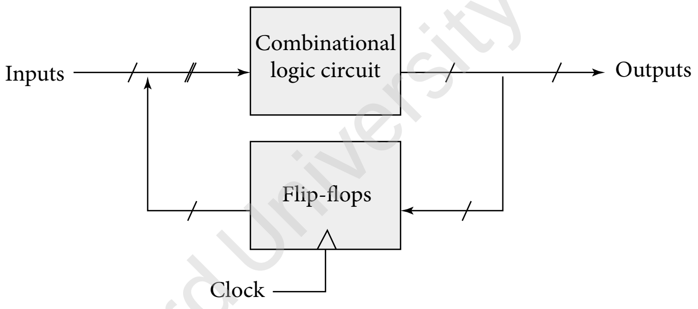


FIGURE 1.85 General FSM

Finite-state machines are classified into two following types:

1. Moore state machine—in this FSM, the outputs are determined by only the internal states.
2. Mealy state machine—in this FSM, the outputs are determined by the internal states as well as the inputs.

1.6.1 Example of FSM

Figure 1.86 shows an FSM. The outputs of an FSM are determined by its inputs, and the present state of the flip-flops. The next states of the flip-flops are also determined by the inputs and the present state of the flip-flops. The behavior of an FSM is fully specified by a graphical representation called state diagram. Figure 1.87 shows a state diagram of an FSM. It consists of a number of states that the FSM will follow in sequence by the application of inputs and the corresponding outputs of the FSM are also described.

The input and output of the FSM are x and y, respectively. There are two DFFs with outputs $Q0$ and $Q1$. The present and next states of the flip-flops are $Q1(n)Q0(n)$ and $Q1(n+1)Q0(n+1)$, respectively. From the circuit diagram, we can write the following state equations:

$$Q0(n+1) = x \oplus Q0(n) \tag{1.79}$$

$$Q1(n+1) = x \oplus \overline{Q0(n)} \tag{1.80}$$

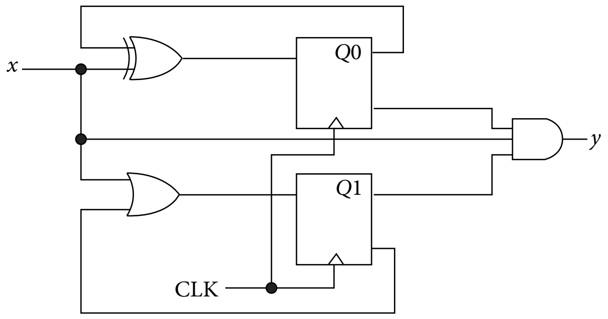


FIGURE 1.86 Example of an FSM

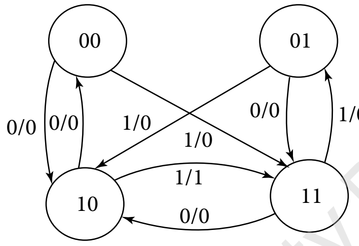


FIGURE 1.87 State diagram

The output y is expressed as follows:

$$y = x\overline{Q0(n)}Q1(n) \tag{1.81}$$

The states are represented by circles enclosed with the corresponding state value. The transitions between the states are represented by directed lines. The lines are associated with a notation like x/y where x indicates logic value of the input and y indicates the logic value of the output. For example, 1/0 indicates that input is logic 1 and output is logic 0.

From the state transition diagram, we can derive the state table as shown in Table 1.36. There are four columns for present state, input, next state, and output.

TABLE 1.36 State table

Present state		Input	Next state		Output
Q1	Q0	*x*	Q1	Q0	*y*
0	0	0	1	0	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	0	1	0
1	1	1	1	0	0

1.6.2 Design of an FSM

Let us design an FSM as described by the state diagram shown in Fig. 1.88. We can implement the FSM using four steps described as follows:

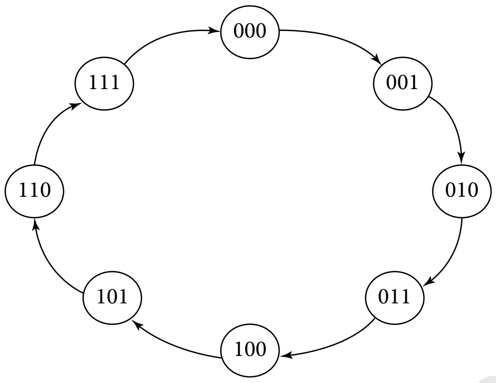


FIGURE 1.88 State diagram of a 3-bit binary up-counter

Step 1: The first design step is to derive the state table of the circuit from the given state diagram. Let us assume three bits as $Q2$, $Q1$, and $Q0$ where $Q2$ is MSB. The state table can be derived as shown in Table 1.37.

TABLE 1.37 State table of 3-bit binary up-counter

Clock pulse	Present state			Next state		
	$Q2$	$Q1$	$Q0$	$Q2$	$Q1$	$Q0$
1	0	0	0	0	0	1
2	0	0	1	0	1	0
3	0	1	0	0	1	1
4	0	1	1	1	0	0
5	1	0	0	1	0	1
6	1	0	1	1	1	0
7	1	1	0	1	1	1
8	1	1	1	0	0	0

The initial state of the circuit is '000'. After every clock pulse, the circuit goes from the present state to the next state. For example, after the first clock pulse, the circuit goes from the state '000' to '001'. After the second clock pulse, the circuit goes from the state '001' to '010'. In this manner, after the eighth clock pulse, the circuit goes back to its initial state '000'.

Step 2: Let us design the circuit using JK flip-flops. Therefore, we need to see how a JK flip-flop changes its state for the inputs J and K. This is described by its state transition table shown in Table 1.38.

In Table 1.38, $Q(n)$ is the present state and $Q(n + 1)$ is the next state. The letter 'X' indicates don't care condition.

Using the Table 1.38, we can rewrite Table 1.37 with required inputs for every state transition as shown in Table 1.39.

TABLE 1.38 State transition table for JK flip-flop

Output transitions			Required inputs	
$Q(n)$	$\rightarrow$	$Q(n+1)$	J	K
0	$\rightarrow$	0	0	X
0	$\rightarrow$	1	1	X
1	$\rightarrow$	0	X	1
1	$\rightarrow$	1	X	0

TABLE 1.39 State table of 3-bit binary up-counter

Clock pulse	Present state			Next state			Required inputs					
	$Q2$	$Q1$	$Q0$	$Q2$	$Q1$	$Q0$	$J2$	$K2$	$J1$	$K1$	$J0$	$K0$
1	0	0	0	0	0	1	0	X	0	X	1	X
2	0	0	1	0	1	0	0	X	1	X	X	1
3	0	1	0	0	1	1	0	X	X	0	1	X
4	0	1	1	1	0	0	1	X	X	1	X	1
5	1	0	0	1	0	1	X	0	0	X	1	X
6	1	0	1	1	1	0	X	0	1	X	X	1
7	1	1	0	1	1	1	X	0	X	0	1	X
8	1	1	1	0	0	0	X	1	X	1	X	1

Step 3: The next step is to obtain the Boolean expressions for J and K inputs using the Karnaugh map method as illustrated in Fig. 1.89.

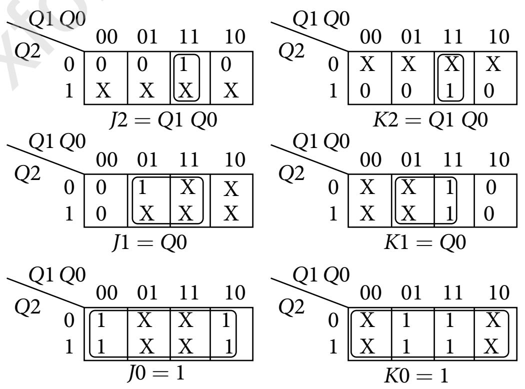


FIGURE 1.89 Karnaugh maps for J and K inputs

Step 4: Using the expressions for J and K inputs, the circuit is implemented as shown in Fig. 1.90.

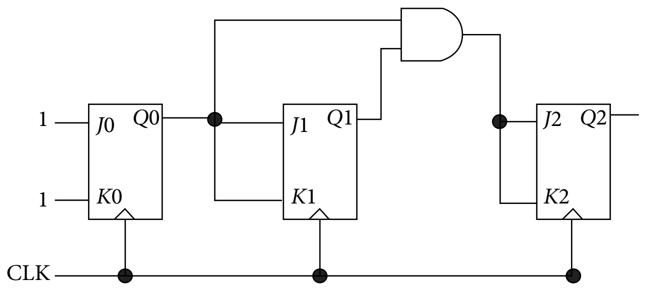


FIGURE 1.90 Circuit diagram for 3-bit up-counter

1.6.3 State Reduction

It is the technique of reducing the number of states of an FSM without disturbing the behavior of the FSM. The reduction of state will ultimately reduce the required number of logic gates and flip-flops in the design.

Sometimes, different states of an FSM are equivalent to each other and they can be combined into a single state. Two states of an FSM are equivalent *if and only if, for any input, they have identical outputs and the corresponding next states are equivalent*.

Let us consider an example of FSM represented by a finite state diagram shown in Fig. 1.91. The corresponding state table is shown in Table 1.40. The states *s*4 and *s*6 are equivalent as they have identical next state and output for both inputs. So we can replace the states *s*4 and *s*6 with *s*46 in the state table and remove one of these states.

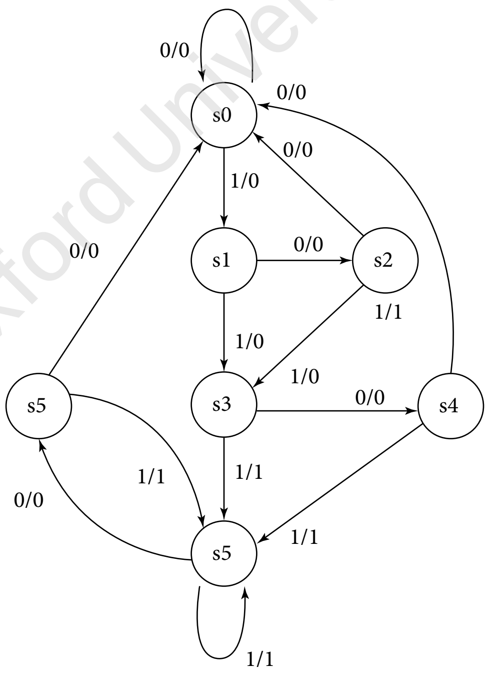


FIGURE 1.91 FSM with seven states

Now if we look at Table 1.41 we find that the states *s*3 and *s*5 are equivalent as they have identical next state and same output for both inputs. So we replace states *s*3 and *s*5 by *s*35 and remove one of them from the state

TABLE 1.40 State table of the FSM shown in Fig. 1.91

Present state	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$s0$	$s0$	$s1$	0	0
$s1$	$s2$	$s3$	0	0
$s2$	$s0$	$s3$	0	0
$s3$	$s4$	$s5$	0	1
$s4$	$s0$	$s5$	0	1
$s5$	$s6$	$s5$	0	1
$s6$	$s0$	$s5$	0	1

TABLE 1.41 State table of the FSM after removal of one state

Present state	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$s0$	$s0$	$s1$	0	0
$s1$	$s2$	$s3$	0	0
$s2$	$s0$	$s3$	0	0
$s3$	$s46$	$s5$	0	1
$s46$	$s0$	$s5$	0	1
$s5$	$s46$	$s5$	0	1

table. After this, we do not find any other equivalent states in Table 1.42. Finally, we rename the equivalent states $s35$ as $s3$ and $s46$ as $s4$ and rewrite the state table in Table 1.43 and redraw the state diagram in Fig. 1.92.

TABLE 1.42 State table of the FSM after removal of two states

Present state	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$s0$	$s0$	$s1$	0	0
$s1$	$s2$	$s35$	0	0
$s2$	$s0$	$s35$	0	0
$s35$	$s46$	$s35$	0	1
$s46$	$s0$	$s35$	0	1

TABLE 1.43 State table of the FSM after renaming of equivalent states

Present state	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$s0$	$s0$	$s1$	0	0
$s1$	$s2$	$s3$	0	0
$s2$	$s0$	$s3$	0	0
$s3$	$s4$	$s3$	0	1
$s4$	$s0$	$s3$	0	1

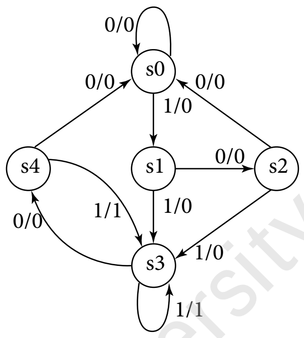


FIGURE 1.92 FSM with five states

1.6.4 State Encoding

Determining the binary representations of the sates of an FSM is the state encoding problem. State encoding determines the size of the design and speed of the design. Encoding length is the number of bits required to represent the states.

The simplest encoding method is to encode a state by setting a corresponding bit to 1 and setting the remaining bits to 0. This is known as 1-hot state encoding.

The minimum length codes uses $n_b = log_2 n_s$ bits to represent each state where n_s is the number of states. This code assigns states in binary counting order.

Another encoding technique is to use the Gray code. Gray code one advantage in that there is only one change required in going from one state to the next state.

The three different encoding techniques for state assignment of Fig. 1.92 are illustrated in Table 1.44.

TABLE 1.44 Different state encoding techniques

State	1-hot code	Binary code	Gray code
$s0$	00001	000	000
$s1$	00010	001	001
$s2$	00100	010	011
$s3$	01000	011	010
$s4$	10000	100	110

1.6.5 State Assignment

We have discussed the technique to reduce the number of states of an FSM. In the previous section, we have found that a seven-state FSM is converted into a five-state FSM using the state reduction technique. We have also learned that how each of the states can be encoded into a binary pattern. Now question arises which state should be assigned to which binary pattern? For example, the state $s0$ can be assigned to any one of the possible binary patterns.

The appropriate choice of the binary patterns to the states has an impact on reducing the required number of logic gates to implement the logic circuit. The minimum number of DFFs required to implement the logic circuit is related to the number of states in the logic circuit which is given as follows:

$$2^n \geq m, \tag{1.82}$$

where n is the number of DFFs and m is the number of state. For example, if there are six states in a state machine, it will require minimum three DFFs. If there are four states in a state machine, it will require minimum two DFFs. Now it is possible to assign these four states to the binary patterns in 24 numbers of ways. The number of combinations of binary patterns to be assigned to m number of states can be expressed as follows:

$$N = \frac{2^n!}{(2^n - m)!} \tag{1.83}$$

For example, in four-state FSM, $N = 2^2!/(2^2 - 4)! = 24$ number of combinations are possible to assign the four states.

1.6.6 Moore Machine

A Moore machine is an FSM in which the output depends only on the present state. It does not depend on the inputs. A three-state Moore machine is shown in Fig. 1.93.

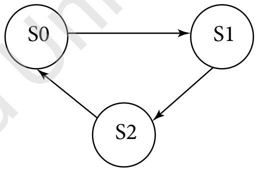


FIGURE 1.93 State diagram of a typical Moore machine

1.6.7 Mealy Machine

Mealy machine is another FSM in which the output depends on the present state as well as on the input. A three-state Mealy machine is described by state diagram as shown in Fig. 1.94. It has one input x and one output y.

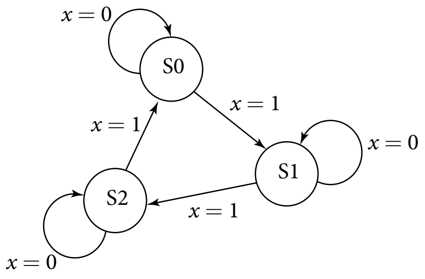


FIGURE 1.94 State diagram of a Mealy machine

1.7 MEMORY

Memory is a logic circuit that is used to store binary information. Normally, flip-flops or latches are used as storage elements. In dynamic memory, binary information is stored in capacitors.

1.7.1 Units of Memory

The smallest unit of memory is bit. It is either logic 1 or logic 0. A group of 8-bits is known as one byte and a group of 4-bits is known as one nibble. Typically, memory size is specified in kilobytes, megabytes, gigabytes, or in terabytes as shown in Table 1.45.

TABLE 1.45 Typical units of memory

Memory size	Actual amount in byte	Actual amount in bit
1 kilobyte = 1 kB	1024	8×1024
1 megabyte = 1 MB	1024×1024	$8\times1024\times1024$
1 gigabyte = 1 GB	$1024\times1024\times1024$	$8\times1024\times1024\times1024$
1 terabyte = 1 TB	$1024\times1024\times1024\times1024$	$8\times1024\times1024\times1024\times1024$

A word is a group of bits that holds complete unit of information.

1.7.2 Architecture of Memory

A unit that stores one bit is known as a memory cell. A memory is an array of cells as illustrated in Fig. 1.95.

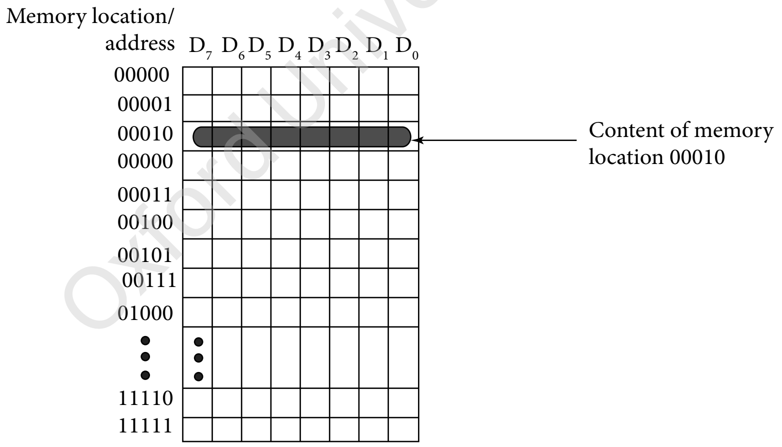


FIGURE 1.95 Memory array architecture

1. Address—The address of memory is the location where the binary data is stored. One location stores one word of information. If 2^n words can be stored in a memory, then its address size is n-bit. For example, a memory with 32 words has 5 address bits.
2. Data—The binary information stored in a memory location is called data or one memory word. It can be one byte, two bytes, three bytes, or n-bytes (n is an integer) of binary information.
3. Read—A memory read operation means data retrieval from the memory. To read the memory, a memory location is to be provided along the read control signal.

4. Write—A memory write operation means storing data into the memory. To write into the memory, a memory location, data to be written, and write control are needed. Write operation is also called programming.

1.7.3 Types of Memories

Memories are broadly classified into the following four types:

1. Sequential access memory (SAM)
2. Random access memory (RAM)
3. Read only memory (ROM)
4. Content addressable memory (CAM)

In SAM, the data is read sequentially from the memory. Thus, read access time varies from one memory location to the other memory location.

In RAM, the data is read randomly from any memory location. The read access time is equal for all memory locations. RAM supports both read and write operations. Thus, it is also known as read write memory (RWM). RAMs are of two types: static RAM (SRAM) and dynamic RAM (DRAM). In SRAM, data is stored in a latch that stores data for an indefinite amount of time as long as power supply is on. In DRAM, data is stored on the parasitic stray capacitors in which data is lost after a period of time. Therefore, in DRAM data needs to be refreshed after regular interval of time.

Read only memory has only provision of memory read, and it does not support write operation. It is one-time programmable.

The CAM operates differently than RAM or ROM. In CAM, address of a memory is accessed for a given search data. It compares search data against a table of stored data and returns the address of the matching data. The search operation performed by CAM is much faster than the software search. Therefore, CAMs are used to replace software in search-intensive applications such as Internet routers, data compression, and database acceleration.

1.8 CONTROL LOGIC CIRCUITS

Control logic circuit controls the operation of a digital circuit. For example, an adder/subtractor is controlled by a control signal to either add two operands or subtract one operand from the other. In a bidirectional shift register, the shifting of data is controlled by a control signal. The circuit associated with the controlling operation is the control logic circuit. In a processor, there is a control unit that provides timing and control signals to process different operations depending on the instructions.

A simple adder/subtractor circuit with a control input is shown in Fig. 1.96. The circuit has two parts: an adder section and a control section. The circuit adds to numbers *A* and *B* if the control input $C_{\text{in}} = 0$ and subtracts *B*

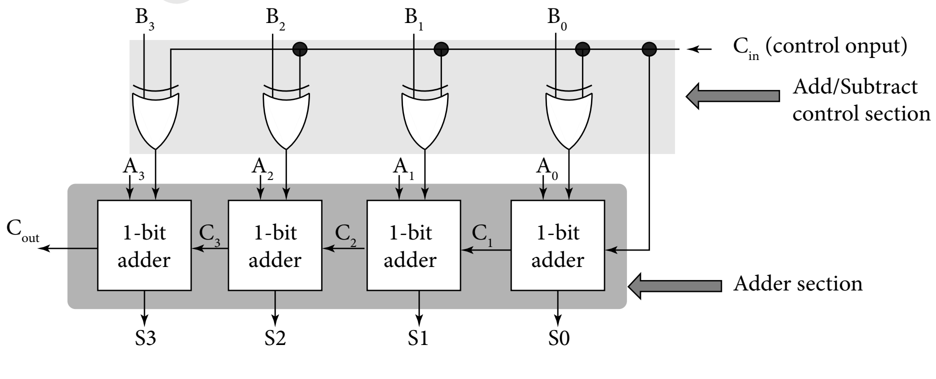


FIGURE 1.96 Adder/subtractor circuit with a control input

from A if the control input $C_{in} = 1$. Thus, with the help of control section, the adder circuit can add or subtract depending on the control input. More complex control circuits are discussed in later chapters of the book.

1.9 ALGORITHMIC STATE MACHINE

Algorithmic state machine (ASM) is a logic circuit comprised both sequential and combinational logic parts. The main task of ASM is to control a digital system to perform the steps of a procedure or an algorithm. The behavior or functionality of an ASM is described by a chart called ASM chart. ASM chart has three basic elements:

1. State box
2. Decision box
3. Conditional box

A state box is represented by a rectangle with one input and one output. The operation of state box is written inside the rectangle. The name of the state is specified at the top-left corner of the box. The binary code assigned to a state box is written at the top-right corner of the box. Figure 1.97 shows a state box.

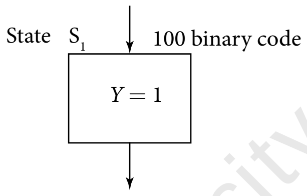


FIGURE 1.97 State box $S1$

A decision box is represented by a diamond-shaped box or rhombus with one input and two output branches. The condition is written inside the rhombus. If the condition is true (or logic 1), the operation control follows one exit path (Exit path 1), and if the condition is false (or logic 0), the operation control follows another exit path (Exit path 2). Figure 1.98 shows a decision box.

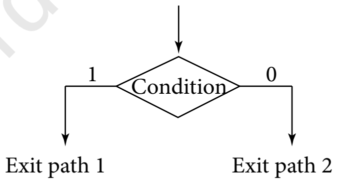


FIGURE 1.98 Decision box

A conditional box is represented by an oval-shaped box with one input coming from the exit path of a decision box and one output. Figure 1.99 shows a conditional box.

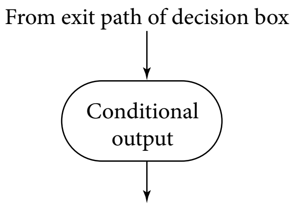


FIGURE 1.99 Conditional box

1.9.1 State Diagram vs. ASM Chart

A state diagram can be translated into an ASM chart or vice-versa. Let us consider a state diagram shown in Fig. 1.100. The state machine has three states: $S2$, $S1$, $S0$ and one input x. The operation starts with the state $S0$.

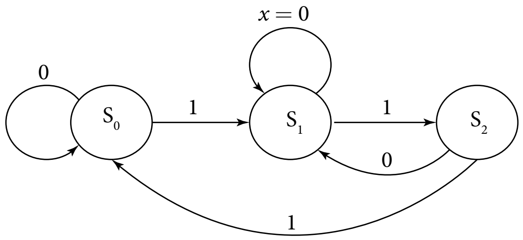


FIGURE 1.100 State diagram

1. At state $S0$, if the input $x = 1$, the next state is $S1$, else the next state is $S0$ itself.
2. At state $S1$, if the input $x = 1$, the next state is $S2$, else the next state is $S1$ itself.
3. At state $S2$, if the input $x = 1$, the next state is $S0$, else the next state is $S1$.
   A corresponding ASM chart can be derived as shown in Fig. 1.101.

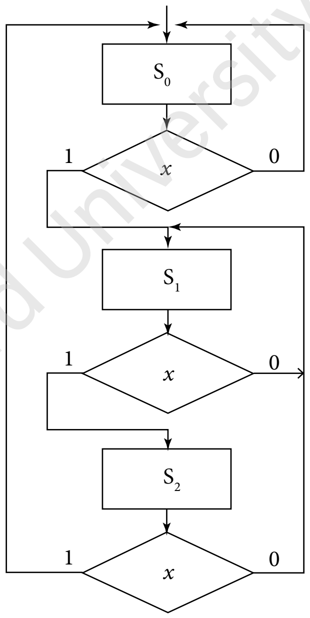


FIGURE 1.101 ASM chart

1.9.2 Realization of ASM Chart

An ASM chart is very much like a state diagram to represent the operation of state machine. Therefore, the sequential circuit implementation technique can be used to realize ASM chart of low or medium complexity. However, for ASM chart with a large number of states some special method is used.

EXAMPLE 1.17 Realize the ASM chart shown in Fig. 1.101.

Solution

There are three states in the ASM chart. Therefore, we need at least 2-bits to represent each state. Hence, there will be two flip-flops in the circuit. Let us assume that flip-flops are DFF with outputs $Q1$ and $Q0$. The state table for the ASM chart can be written as shown in Table 1.46.

TABLE 1.46 State table for Example 1.17

Input	Present state		Next state	
x	$Q1(n)$	$Q0(n)$	$Q1(n+1)$	$Q0(n+1)$
0	0	0	0	0
1	0	0	0	1
0	0	1	0	1
1	0	1	1	0
0	1	0	0	1
1	1	0	0	0

The Boolean expression for the outputs of the next state can be obtained using K-map method as follows:

$$Q_1(n+1) = xQ_0(n) \tag{1.84}$$

$$Q_0(n+1) = \overline{x}Q_1(n) + \overline{x}Q_0(n) + x\overline{Q_1(n)}\,\overline{Q_0(n)} \tag{1.85}$$

Figure 1.102 realizes the ASM chart shown in Fig. 1.101.

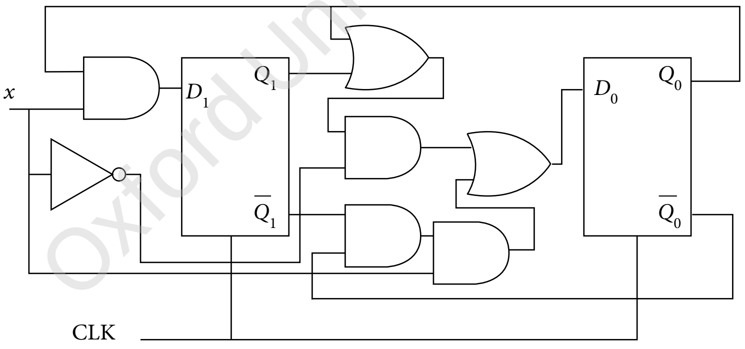


FIGURE 1.102 Logic circuit that realizes ASM chart shown in Fig. 1.101

1.9.3 Linked State Machine

A state machine with a large number of states and large complexity is often broken into smaller state machines. These smaller state machines are linked together to form the complete state machine.

SUMMARY

1. NOT, OR, AND, NOR, NAND, XOR, and XNOR are the basic logic gates.
2. NAND and NOR are the universal gates. Any Boolean expression can be realized using either of the universal gates.
3. Multiplexer has many inputs to single output signal flow depending on select lines, whereas demultiplexer has single input to many output signal flow.
4. Combinational logic circuit does not have any clock input.
5. Sequential logic circuits are operated using clock input as well as data inputs.
6. In JK flip-flop, there is no forbidden state.
7. D flip-flop is used as a unit delay element in digital filter architecture.
8. Algorithmic state machine is a generic representation of an FSM at the algorithmic level.
9. Control logic is an important part of a digital logic circuit that controls the operation of the circuit.

EXERCISES

Fill in the Blanks

1.1 Universal logic gates are ________.
(a) AND & OR
(b) NOT & OR
(c) NAND & NOR
(d) XOR & NOR

1.2 A full-adder is a ________ adder.
(a) 2-bit
(b) 3-bit
(c) 4-bit
(d) 1-bit

1.3 ________ gates are not associative.
(a) NAND & NOR
(b) AND & OR
(c) XOR & XNOR
(d) NOT & OR

1.4 The outputs of sequential logic circuits depend on ________.
(a) present inputs
(b) present inputs and past outputs
(c) present and past inputs
(d) present inputs and future outputs

1.5 Both AND array and OR array are programmable in ________.
(a) PLA
(b) PAL
(c) ROM
(d) PLD

Multiple-choice Type Questions

1.1 XOR logic can be implemented using only
(a) AND gates
(b) NAND gates
(c) OR gates
(d) NOT gates

1.2 The minimum number of NOR gates required to implement XNOR logic is
(a) four
(b) two
(c) three
(d) five

1.3 JK flip-flop has
(a) all valid input combinations
(b) two valid input combinations
(c) three valid input combinations
(d) one valid input combinations

1.4 Master-slave flip-flop is
(a) edge-triggered flip-flop
(b) level-sensitive flip-flop
(c) both (a) and (b)
(d) none of these

1.5 Multiplexer has
(a) many input lines and single output line
(b) one input line and many output lines
(c) many input lines and many output lines
(d) all of these

1.6 Parity generator circuits are used in
(a) transmitter
(b) receiver
(c) both transmitter and receiver
(d) none of these

1.7 In synchronous circuits, all flip-flops are operated by
(a) common clock
(b) different clock
(c) multiple clock
(d) no clock

1.8 The sum-of-product (SOP) form of logical expression is most suitable for designing logic circuits using only
(a) XOR gates
(b) AND gates
(c) NAND gates
(d) NOR gates

1.9 Which of the following flip-flops is used as a latch?
(a) JK flip-flop
(b) D flip-flop
(c) RS flip-flop
(d) T flip-flop

1.10 The initial state of a mod-16 down counter is 0110. The state after 37 clock pulse will be
(a) 0000
(b) 0110
(c) 0101
(d) 0001

1.11 In a D type latch when enable input is high and $D = 1$, the output will be
(a) 0
(b) 1
(c) don't care
(d) blocked

1.12 The frequency of the pulse at point A in Fig. 1.103 is

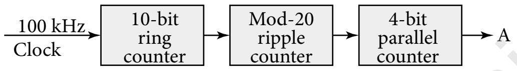


FIGURE 1.103 Diagram for Problem 12

(a) 10 kHz
(b) 31.25 kHz
(c) 50 kHz
(d) 5 kHz

1.13 An example of weighted code is
(a) Excess-3 code
(b) ASCII code
(c) Hamming code
(d) 8421 code

1.14 The minimum number of NAND gates required to design a full-adder is
(a) 5
(b) 9
(c) 6
(d) 10

1.15 A decoder with enable input can be used as
(a) encoder
(b) parity generator
(c) NAND gate
(d) demultiplexer

1.16 The output of a logic gate 1 when all its inputs are at logic 0. The gate is either
(a) NAND or XOR gate
(b) NOR or XOR gate
(c) AND or XNOR gate
(d) NOR or XNOR gate

1.17 JK flip-flop has
(a) one stable state
(b) two stable states
(c) no stable state
(d) none of these

1.18 The operation which is cumulative but not associative is
(a) AND
(b) XOR
(c) NAND
(d) NOT

1.19 The number of XOR gates required for conversion of 11011 to its equivalent gray code is
(a) 2
(b) 4
(c) 3
(d) 5

1.20 A message is 010101. For even parity generator, the parity bit to be added to the message is
(a) 0
(b) 1
(c) 0 and 1
(d) none of these

1.21 The output Y in the circuit shown in Fig. 1.104 is always 1 when

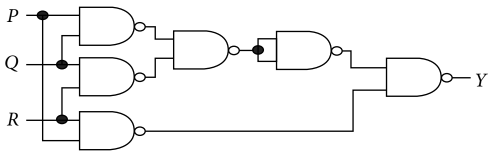


FIGURE 1.104 Diagram for Problem 21

(a) two or more of the inputs are 0
(b) two or more of the inputs are 1
(c) any odd number of the inputs is 0
(d) any odd number of the inputs is 1

1.22 The logic function implemented by the circuit shown in Fig. 1.105 is

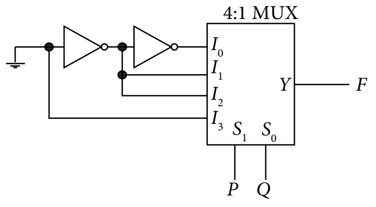


FIGURE 1.105 Diagram for Problem 22

(a) F = AND(P, Q)
(b) F = OR(P, Q)
(c) F = XNOR(P, Q)
(d) F = XOR(P, Q)

1.23 When the output Y in the circuit shown in Fig. 1.106 is 1, it implies that data has

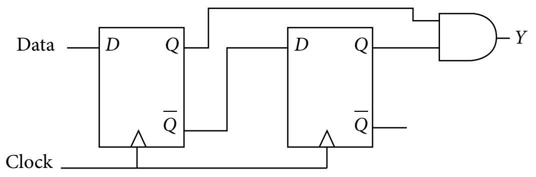


FIGURE 1.106 Diagram for Problem 23

(a) changed from 0 to 1
(b) changed from 1 to 0
(c) changed in either direction
(d) not changed

1.24 Two D flip-flops are connected as a synchronous counter that goes through the following sequence. $00 \to 11 \to 01 \to 10 \to 00 \to \cdots$ The connections to the inputs D_A and D_B are
(a) $D_A = Q_B, D_B = Q_A$
(b) $D_A = \overline{Q_A}, D_B = \overline{Q_B}$
(c) $D_A = (Q_A\overline{Q_B} + \overline{Q_A}Q_B, D_B = Q_A$
(d) $D_A = (Q_AQ_B + \overline{Q_AQ_B}), D_B = \overline{Q_B}$

1.25 Match the logic gates in column A with their equivalent in column B in Fig. 1.107.

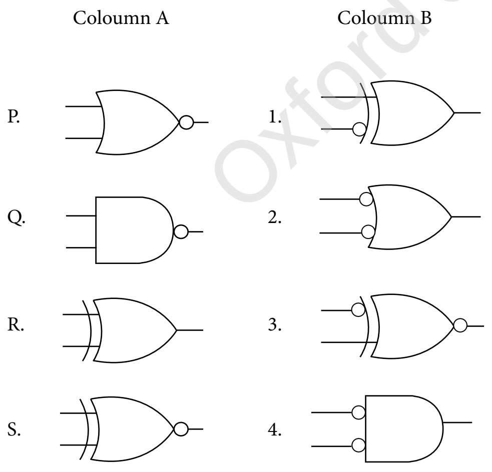


FIGURE 1.107 Diagram for Problem 25

(a) P-2, Q-4, R-1, S-3
(b) P-4, Q-2, R-1, S-3
(c) P-2, Q-4, R-3, S-1
(d) P-4, Q-2, R-3, S-1

1.26 For the output F to be 1 in the logic circuit shown in Fig. 1.108, the input combination should be

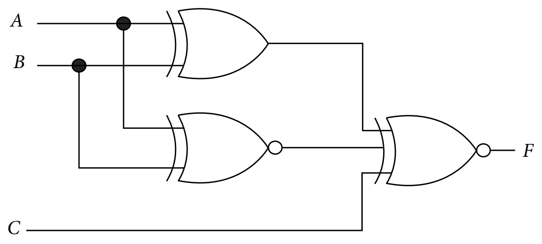


FIGURE 1.108 Diagram for Problem 26

(a) $A = 1, B = 1, C = 0$
(b) $A = 1, B = 0, C = 0$
(c) $A = 0, B = 1, C = 0$
(d) $A = 0, B = 0, C = 1$

1.27 Assuming that all flip-flops are in reset state initially, the count sequence observed at QA in the circuit shown in Fig. 1.109 is

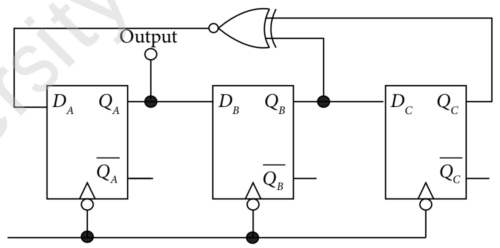


FIGURE 1.109 Diagram for Problem 27

(a) 0010111...
(b) 0001011...
(c) 0101111...
(d) 0110100...

1.28 The Boolean function realized by the logic circuit shown in Fig. 1.110 is

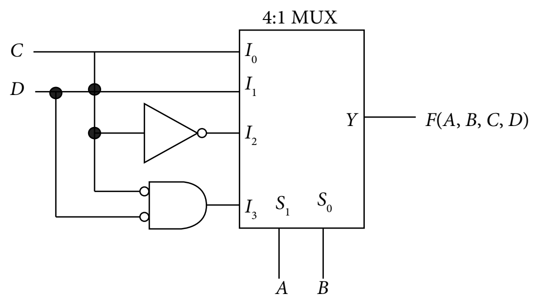


FIGURE 1.110 Diagram for Problem 28

(a) $F = \sum m(0, 1, 3, 5, 9, 10, 14)$

(b) $F = \sum m(0, 1, 3, 5, 9, 10, 14)$
(c) $F = \sum m(1, 2, 4, 5, 11, 14, 15)$
(d) $F = \sum m(2, 3, 5, 7, 8, 9, 12)$

1.29 If $X = 1$ in the following logic expression, then

$$X + Z\{\overline{Y} + (\overline{Z} + X\overline{Y})\}\{\overline{X} + \overline{Z}(X + Y)\} = 1 \quad (1.86)$$

(a) $Y = Z$
(b) $Y = \overline{Z}$
(c) $Z = 1$
(d) $Z = 0$

1.30 What are the minimum numbers of 2-to-1 multiplexers required to generate a two-input AND gate and a two-input XOR gate?
(a) 1 and 2
(b) 1 and 3
(c) 1 and 1
(d) 2 and 2

1.31 In the two-latch circuits shown in Fig. 1.111, the inputs ($P1$, $P2$) for both the latches are first made (0, 1) and then, after a few seconds, made (1, 1). The corresponding stable outputs ($Q1$, $Q2$) are

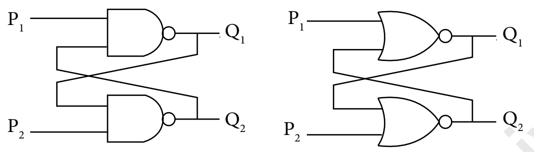


FIGURE 1.111 Diagram for Problem 31

(a) NAND: first (0,1) then (0,1) NOR: first (1,0) then (0,0)
(b) NAND: first (1,0) then (1,0) NOR: first (1,0) then (1,0)
(c) NAND: first (1,0) then (1,0) NOR: first (1,0) then (0,0)
(d) NAND: first (1,0) then (1,1) NOR: first (0,1) then (0,1)

1.32 What are the counting states ($Q1$, $Q2$) for the counter shown below?

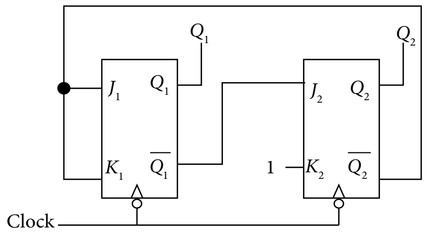


FIGURE 1.112 Diagram for Problem 32

(a) $11, 10, 00, 11, 10, \cdots$
(b) $01, 10, 11, 00, 01, \cdots$
(c) $00, 11, 01, 10, 00, \cdots$
(d) $01, 10, 00, 01, 10, \cdots$

1.33 Which of the following Boolean expression correctly represents the relation between P, Q, R, and $M1$?

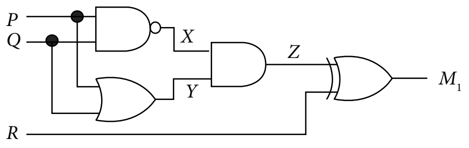


FIGURE 1.113 Diagram for Problem 33

(a) $M1$ = (P OR Q) XOR R
(b) $M1$ = (P AND Q) XOR R
(c) $M1$ = (P NOR Q) XOR R
(d) $M1$ = (P XOR Q) XOR R

1.34 For the circuit shown in Fig. 1.114, $I_0 - I_3$ are inputs to the 4:1 multiplexer. R (MSB) and S are control bits.

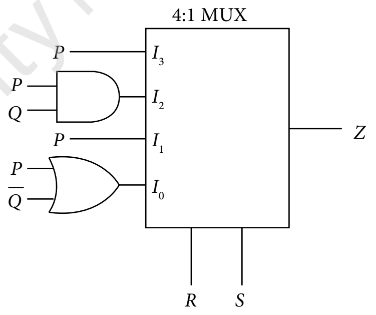


FIGURE 1.114 Diagram for Problem 34

The output Z can be represented by
(a) $PQ + P\overline{Q}S + \overline{Q}\,\overline{R}\,\overline{S}$
(b) $P\overline{Q} + PQ\overline{R} + \overline{P}\,\overline{Q}\,\overline{S}$
(c) $P\overline{Q}\,\overline{R} + \overline{P}QR + PQRS + \overline{Q}\,\overline{R}\,\overline{S}$
(d) $PQ\overline{R} + PQR\overline{S} + P\overline{Q}\,\overline{R}S + \overline{Q}\,\overline{R}\,\overline{S}$

1.35 For each of the positive edge-triggered JK flip-flop used in Fig. 1.115, the propagation delay is ΔT.

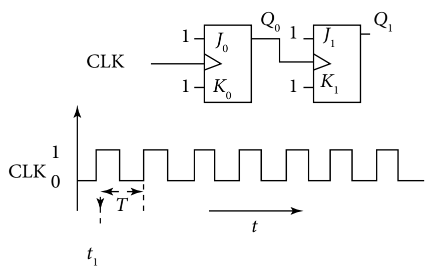


FIGURE 1.115 Diagram for Problem 35

Which of the following waveforms shown in Fig. 1.116 correctly represent the output at $Q1$?

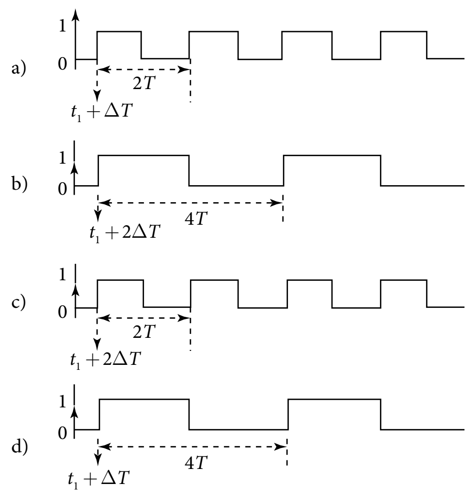


FIGURE 1.116 Waveforms for Problem 35

1.36 For the circuit shown in Fig. 1.117, D has a transition from 0 to 1 after CLK changes from 1 to 0. Assume gate delays to be negligible.

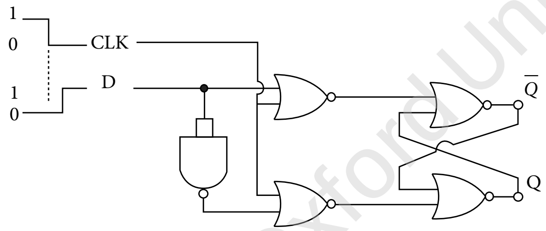


FIGURE 1.117 Waveforms for Problem 36

Which of the following statement is true?
(a) Q goes to 1 at the CLK transition and stays at 1.
(b) Q goes to 0 at the CLK transition and stays at 0.
(c) Q goes to 1 at the CLK transition and goes to 0 when D goes to 1.
(d) Q goes to 0 at the CLK transition and goes to 1 when D goes to 1.

1.37 The Boolean function $Y = AB + CD$ is to be realized using only two-input NAND gates. The minimum number of gate required is
(a) 2
(b) 3
(c) 4
(d) 5

1.38 In the circuit shown in Fig. 1.118, X is given by

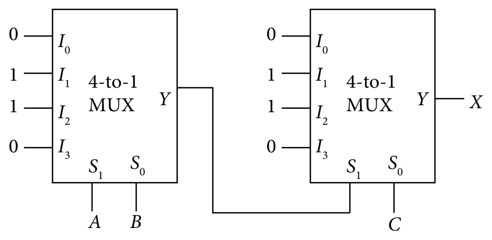


FIGURE 1.118 Waveforms for Problem 38

(a) $X = A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}\,\overline{B}C + ABC$
(b) $X = \overline{A}B\overline{C} + A\overline{B}C + AB\overline{C} + \overline{A}\,\overline{B}\,\overline{C}$
(c) $X = AB + BC + AC$
(d) $X = \overline{A}\,\overline{B} + \overline{B}\,\overline{C} + \overline{A}\,\overline{C}$

1.39 The following binary values were applied to the X and Y inputs of the NAND latch shown in Fig. 1.119 in the sequence indicated below: X = 0, Y = 1; X = 0, Y = 0; X = 1, Y = 1. The corresponding stable P, Q outputs will be

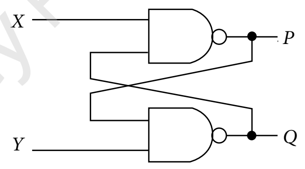


FIGURE 1.119 Waveforms for Problem 39

(a) $P = 1$, $Q = 0$; $P = 1$, $Q = 0$; $P = 1$, $Q = 0$ or $P = 0$, $Q = 1$
(b) $P = 1$, $Q = 0$; $P = 0$, $Q = 1$ or $P = 0$, $Q = 1$; $P = 0$, $Q = 1$
(c) $P = 1$, $Q = 0$; $P = 1$, $Q = 1$; $P = 1$, $Q = 0$ or $P = 0$, $Q = 1$
(d) $P = 1$, $Q = 0$; $P = 1$, $Q = 1$; $P = 1$, $Q = 1$

1.40 For the circuit shown in Fig. 1.120, the counter state ($Q1Q0$) follows the sequence

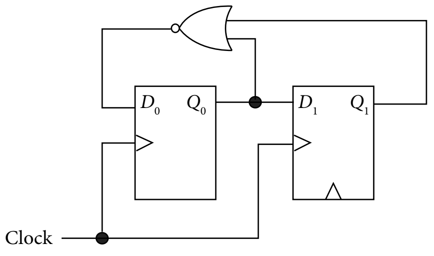


FIGURE 1.120 Waveforms for Problem 40

(a) 00, 01, 10, 11, 00, $\cdots$
(b) 00, 01, 10, 00, 10, $\cdots$
(c) 00, 01, 11, 00, 01, $\cdots$
(d) 00, 10, 11, 00, 10, $\cdots$

State True or False

1.1 Barrel shifter is a sequential circuit.

1.2 In Moore state machine, the outputs are determined by only the internal states.

1.3 Setup time is the time after the clock transition when data must not change.

1.4 The outputs toggle in a T flip-flop when its T-input is at logic 1.

1.5 Ripple counter is an asynchronous counter.

Short-answer Type Questions

1.1 Design a 4:16 decoder using 3:8 decoders.

1.2 Implement two-input XOR function using minimum number of two-input NAND gates.

1.3 Design a full-subtractor using 4:1 multiplexer.

1.4 Perform the conversion from SR to JK flip-flop.

1.5 Realize a full-subtractor using only NOR gates.

1.6 Draw a BCD adder circuit to add two BCD numbers maximum up to 9. The outputs of this adder should also be in BCD.

1.7 Construct a 2-bit comparator using only decoder.

1.8 What is the main difference between a latch and a flip-flop?

1.9 Design a full-adder using 3:8 decoder and other logic gates.

1.10 Explain the race-around condition of JK flip-flop.

Long-answer Type Questions

1.1 (a) Design a mod-10 synchronous binary up-counter using JK flip-flop and other necessary gates. (b) Calculate the propagation delay for 4-bit synchronous binary up-counter when JK flip-flops are connected in series and parallel. Given that the propagation delay of the flip-flop is 30 ns and the other logic gates have equal propagation delay of 20 ns.

1.2 (a) Draw the circuit for a 4-bit Johnson counter using D flip-flop and explain its operation. Draw its timing diagram. How does the timing diagram differ from that of a Ring counter? (b) Perform the conversion from D flip-flop to JK flip-flop.

1.3 (a) Distinguish between ROM, PLA, and PAL. (b) Design a combinational circuit using an 8 × 4 ROM that accepts a 3-bit number and generates an output binary number equal to the square of the input number. (c) Draw a logic diagram of master-slave JK flip-flop. Why is it called so?

1.4 (a) Write down the difference between combinational logic circuit and sequential logic circuit. (b) Design a mod-14 asynchronous up/down counter using JK flip-flop.

1.5 (a) Design a combinational circuit that accepts a BCD input and generates Excess-3 as output using ROM. (b) Design and explain the operation of a 4-bit universal register.

1.6 (a) Write down the excitation table of JK and D flip-flops. Derive the excitation equation for these flip-flops. (b) Design a full-subtractor using a full-adder and NOT gates.

1.7 Obtain the ASM chart for the following state transitions: (a) If $x = 0$, control goes from state $S1$ to state $S2$; if $x = 1$, generate a conditional operation and go from $S1$ to $S2$. (b) If $x = 1$, control goes from $S1$ to $S2$ and then to $S3$; if $x = 0$, control goes from $S1$ to $S3$. (c) Start from state $S1$; then: if $xy = 00$, go to state $S2$; if $xy = 01$, go to state $S3$; if $xy = 10$, go to state $S1$; otherwise, go to state $S3$.

1.8 Design a mod-8 up/down counter.

ANSWERS

Fill in the Blanks

1.1	1.2	1.3	1.4	1.5
(c)	(b)	(a)	(b)	(a)

Multiple-choice Type Questions

1.1	1.2	1.3	1.4	1.5	1.6
(b)	(a)	(a)	(a)	(a)	(a)
1.7	1.8	1.9	1.10	1.11	1.12
(a)	(c)	(c)	(d)	(b)	(b)
1.13	1.14	1.15	1.16	1.17	1.18
(d)	(b)	(d)	(d)	(b)	(c)
1.19	1.20	1.21	1.22	1.23	1.24
(b)	(b)	(b)	(d)	(a)	(d)
1.25	1.26	1.27	1.28	1.29	1.30
(d)	(d)	(d)	(d)	(d)	(a)
1.31	1.32	1.33	1.34	1.35	1.36
(c)	(c)	(d)	(a)	(b)	(a)
1.37	1.38	1.39	1.40		
(b)	(a)	(c)	(b)		

True or False

1.1	1.2	1.3	1.4	1.5
(f)	(t)	(f)	(t)	(t)