SPECIAL INDIAN EDITION

DIGITAL PRINCIPLES AND APPLICATIONS

Seventh Edition

Donald P Leach
Santa Clara University

Albert Paul Malvino
President, Malvino Inc.

Goutam Saha
Associate Professor
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology (IIT) Kharagpur

2851

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Special Indian Edition 2011

Adapted in India by arrangement with the McGraw-Hill Compaines, Inc., New York

Sales Territories: India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan

Digital Principles and Applications, 7e

First reprint 2011
DZXCRRXGRQRQR

ISBN (13 digit): 978-0-07-014170-4
ISBN (10 digit): 0-07-014170-3

Vice President and Managing Director—McGraw-Hill Education: Asia-Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Manager: Sponsoring—SEM & Tech Ed: *Shalini Jha*
Asst Sponsoring Editor: *Surabhi Shukla*
Development Editor: *Surbhi Suman*
Executive—Editorial Services: *Sohini Mukherjee*
Jr Manager—Production: *Anjali Razdan*
Dy Marketing Manager: SEM & Tech Ed: *Biju Ganesan*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

Typeset at Tej Composers, WZ 391, Madipur, New Delhi 110 063 and printed at Pashupati Printers Pvt. Ltd., 1/429/16, Gali No. 1, Friends colony, Industrial Area, G.T. Road, Shahdara, Delhi 110095

Cover Printer: SDR Printers

The McGraw-Hill Companies

Contents

Preface to the Seventh Edition (SIE)

The seventh edition of *Digital Principles and Applications* continues with the upgradation of the work started in its previous edition. The job was to build upon the strengths of one of the best introductory and authentic texts in the field of Digital Electronics—its lucid language, down-to-earth approach, detailed analysis and ready-to-use information for laboratory practices. The sixth edition sought improvement primarily by (i) strengthening the design or synthesis aspect that included advanced material, such as a simple computer design, and (ii) incorporating many new topics like Hardware Description Language, Asynchronous Sequential Circuit, Algorithm State Machine chart, Quine-McClusky algorithm, Look Ahead Carry Adder, etc.

The tremendous response to the improvements made in the sixth edition from the academic community prompted us to work on their suggestions and come out with this seventh edition.

NEW TO THIS EDITION

The seventh edition has been revised extensively and restructured to emphasize new and important concepts in Digital Principles and Applications. This edition increases the depth and breadth of the title by incorporating latest information on existing topics like *Boolean Algebra*, *Schmitt Trigger*, *555 Timer*, *Edge Triggering*, *Memory Cell*, *Computer Architecture,* and also introduces new topics like *Noise Margin*, *Error Detection and Correction*, *Universal Shift Register and Content Addressable Memory*.

The most notable change in this edition is the inclusion of two completely new features—*problem solving by multiple methods* and *laboratory experiments*—that will enable the student community develop deeper understanding of the application side of digital principles. *Problem solving by multiple methods* help students in understanding and appreciating different alternatives to reach a solution, without feeling stuck at any point of time. *Laboratory experiments* facilitate experimentation with different analysis and synthesis problems using digital integrated circuits (IC). Each experiment describes its aim, a short reference to theory, apparatus required and different work elements.

THE BASIC FEATURES

The new edition retains its appeal as a complete self-study guide for a first-level course on Digital Logic and Digital Circuits. It will serve the purpose of a textbook for undergraduate students of CSE, ECE, EEE, Electronics and Instrumentation and IT. It will also be a valuable reference for students of MCA, BCA, DOEACC 'A' Level, as well as BSc/MSc (Computer Science/IT).

The key features are:

- Presence of various applications and lab experiments considering the common digital circuit design employed in industries (e.g., LCD display and ADC0804 operation).
- In-depth coverage of important topics like clock and timing circuits, D/A-A/D conversion, register, counters and memory.
- Tutorial-based approach with section-end self test questions and problem solving through various methods.
- Useful discussion on TTL and CMOS devices and pin diagrams
- Rich Pedagogy
    - *180 Solved Examples*
    - *290 Section-end Problems*
    - *500 Chapter-end Problems*

COMPREHENSIVE WEBSITE

An important addition to this title is the accompanying website—*http://www.mhhe.com/leach/dpa7*, designed to be an exhaustive Online Learning Centre (OLC). This website contains the following:

For Students

- Downloadable codes for HDL examples in the book
- Supplementary Reading material

Besides Quine-McClusky code and HDL examples, additional information and discussion on various supplementary materials like five-variable Karnaugh Map and Petrick's Algorithm will be available here. Regular updates on different topics of Digital Electronics will be posted to keep the reader informed about recent changes in this field.

For Instructors

Instructors who have adopted this textbook can access a password-protected section that offers the following resources.

- Solution manual
- Chapterwise PowerPoint slides with diagrams and notes

ACKNOWLEDGEMENTS

I would like to acknowledge the inspiration and support I received from Prof. B N Chatterji (Retd.), Prof. R N Pal (Retd.), Prof. R V Rajakumar, Prof. A Chakraborty, Prof. P P Chakraborty, Prof. D Datta, Prof. S Bannerjee, Prof. P K Biswas, Prof. S S Pathak, Prof. S Mukhopadhyaya, Prof. A S Dhar, Prof. I Chakraborty, Prof. P Mandal, Prof. S Mahapatra, Prof. T K Bhattacharya, and all my faculty colleagues at Indian Institute of Technology (IIT) Kharagapur, especially, from the Department of Electronics and Electrical Communication Engineering and G S Sanyal School of Telecommunications. But for the paucity of space, I would have named many others.

Thanks are due to my research students—Mr S Ari, Mr Md Sahidullah, Mr Israj Ali, and Mr A Mandal for their contribution at different stages of development of the edition. I acknowledge the benefit derived from my interaction with different batches of students while teaching the Digital Electronics subject—three years at Institute of Engineering and Management, Kolkata and over six years at IIT Kharagpur.

I am grateful to the esteemed reviewers for their encouraging comments and valuable suggestions for this edition.

Sunil Mathur
Maharaja Agrasen Institute of Technology, New Delhi
V Kumar
Maharaja Surajmal Institute of Technology, New Delhi
Bijoy Bandopadhyay
University College of Science & Technology, Kolkata
Anita Kanavalli
M S Ramaiah Institute of Technology, Bangalore

I also thank the entire team of Tata McGraw Hill Education, more specifically Vibha Mahajan, Shalini Jha, Ashes Saha, Surbhi Suman, Anjali Razdan and Baldev Raj for their support.

At this point, I humbly remember all my teachers and my father (late) G N Saha who provided me a great learning environment. I also fondly recollect the contributions in my upbringing of Kharagpur Vivekananda Yuva Mahamandal, Vivekananda Study Circle, IIT Kharagpur Campus and Ramakrishna Mission. I must mention the support I always received from my family—my mother, my parents-in-law, my sisters (specially Chhordi), Chhoto Jamaibabu, and last but not the least, my wife, Sanghita, and daughter, Upasana. The effort behind this work was mine but the time was all theirs.

GOUTAM SAHA

Feedback

Due care has been taken to avoid any mistake in the print edition as well as in the OLC. However, any note on oversight as well as suggestions for further improvement sent at tmh.csefeedback@gmail.com will be gratefully acknowledged (*kindly mention the title and author name in the subject line*). Also, please report to us any piracy of the book spotted by you.

Preface

PURPOSE

The fifth edition of *Digital Principles and Applications* is completely recorgnized. It is written for the individual who wishes to learn the *principles* of digital circuits and then *apply* them to useful, meaningful design. Thus the title. The material in this book is appropriate for an introductory course in digital logic in either a computer or an electronics program. It is also appropriate for "self-study" and as a "reference" for individuals working in the field. Emphasis is given to the two most popular digital circuit (IC) families—transistor-transistor logic (TTL) and complementary metal oxide silicon (CMOS) logic. Many of these individual ICs are discussed in detail, and pinouts for more than 60 digital IC chips are summarized in Appendix 8. Standard logic symbols are used along with the new IEEE standard logic. A review of the new IEEE symbols is given in the appendix.

BACKGROUND

It is not necessary to have a background in electronics to study this text. A familiarity with Ohm's law and voltage and current in simple dc resistive circuits is helpful but not required. If you have no desire to learn about electronics, you can skip Chap. 13. To the extent possible, the remaining chapters are written to be independent of this material. If you have not studied electronics, Chap. 13 will provided the necessary background for you to converse successfully with those who have. Study it any time after Chap. 1. For "old-times" who have studied electronics, Chap. 13 will provide a good review and perhaps a new and valuable point of view. In any case, the material in Chap. 13 will certainly enhance both the knowledge and ability of anyone!

ORGANIZATION

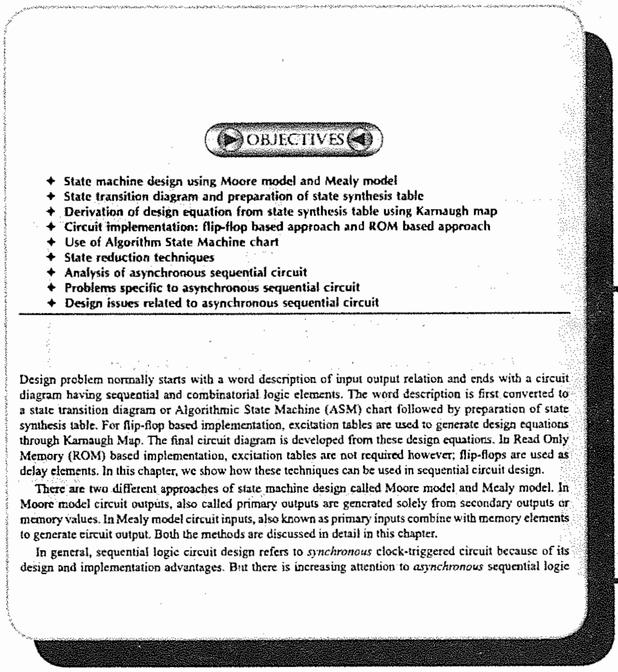
Each chapter begins with a contents that lists the subjects in each section. The contents listing is followed by a list of chapter objectives. At the end of each chapter section are review questions, called self-tests, which are intended to be a self-check of key ideas and concepts. At the end of each chapter, answers are supplied for the self-tests. A summary and a glossary are provided at the end of each chapter. In any subject area, there are many terms and concepts to be learned. The summary and glossary will provide you with the opportunity to be sure that you understand the *exact* meaning of these terms, phrases, and abbreviations, The end-of-chapter problems are arranged according to chapter sections. The problems reinforce ideas and concepts presented and allow you to apply them on your own. Solutions to selected odd-numbered problems are given at the end of the book. In addition, the appendix contains reference material that will be useful from time to time.

LABORATORY EXPERIMENTS

A complete set of experiments keyed to this text is available in a laboratory manual, *Experiments for Digital Principles*.

DONALD P. LEACH
ALBERT PAUL MALVINO

Visual Walkthrough



OBJECTIVES

- State machine design using Moore model and Mealy model
- State transition diagram and preparation of state synthesis table
- Derivation of design equation from state synthesis table using Karnaugh map
- Circuit implementation: flip-flop based approach and ROM based approach
- Use of Algorithm State Machine chart
- State reduction techniques
- Analysis of asynchronous sequential circuit
- Problems specific to asynchronous sequential circuit
- Design issues related to asynchronous sequential circuit

Design problem normally starts with a word description of input output relation and ends with a circuit diagram having sequential and combinatorial logic elements. The word description is first converted to a state transition diagram or Algorithmic State Machine (ASM) chart followed by preparation of state synthesis table. For flip-flop based implementation, excitation tables are used to generate design equations through Karnaugh Map. The final circuit diagram is developed from these design equations. In Read Only Memory (ROM) based implementation, excitation tables are not required however; flip-flops are used as delay elements. In this chapter, we show how these techniques can be used in sequential circuit design.

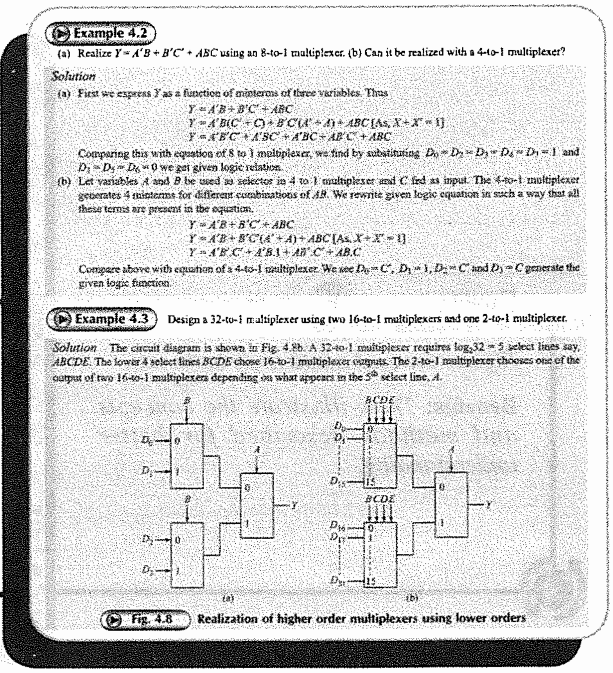
There are two different approaches of state machine design called Moore model and Mealy model. In Moore model circuit outputs, also called primary outputs are generated solely from secondary outputs or memory values. In Mealy model circuit inputs, also known as primary inputs combine with memory elements to generate circuit output. Both the methods are discussed in detail in this chapter.

In general, sequential logic circuit design refers to *synchronous* clock-triggered circuit because of its design and implementation advantages. But there is increasing attention to *asynchronous* sequential logic

Chapter Objectives

Every chapter opens with a set of chapter objectives.

Benefits: *These provide a quick look into the concepts that will be discussed in the chapter.*



Example 4.2

(a) Realize $Y = A'B + B'C' + ABC$ using an 8-to-1 multiplexer. (b) Can it be realized with a 4-to-1 multiplexer?

Solution

(a) First we express Y as a function of minterms of three variables. Thus

$$Y = A'B + B'C' + ABC$$
$$Y = A'B(C' + C) + B'C'(A' + A) + ABC \; [\text{As, } X + X' = 1]$$
$$Y = A'B'C' + A'BC' + A'BC + AB'C' + ABC$$

Comparing this with equation of 8 to 1 multiplexer, we find by substituting $D_0 = D_2 = D_3 = D_4 = D_7 = 1$ and $D_1 = D_5 = D_6 = 0$ we get given logic relation.

(b) Let variables A and B be used as selector in 4 to 1 multiplexer and C fed as input. The 4-to-1 multiplexer generates 4 minterms for different combinations of AB. We rewrite given logic equation in such a way that all these terms are present in the equation.

$$Y = A'B + B'C' + ABC$$
$$Y = A'B + B'C'(A' + A) + ABC \; [\text{As, } X + X' = 1]$$
$$Y = A'B'.C' + A'B.1 + AB'.C' + AB.C$$

Compare above with equation of a 4-to-1 multiplexer. We see $D_0 = C'$, $D_1 = 1$, $D_2 = C'$ and $D_3 = C$ generate the given logic function.

Example 4.3 Design a 32-to-1 multiplexer using two 16-to-1 multiplexers and one 2-to-1 multiplexer.

Solution The circuit diagram is shown in Fig. 4.8b. A 32-to-1 multiplexer requires $\log_2 32 = 5$ select lines say, $ABCDE$. The lower 4 select lines $BCDE$ chose 16-to-1 multiplexer outputs. The 2-to-1 multiplexer chooses one of the output of two 16-to-1 multiplexers depending on what appears in the 5th select line, A.

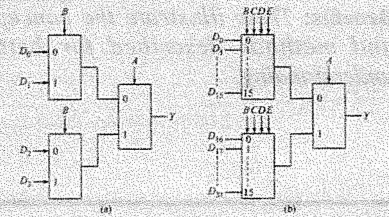
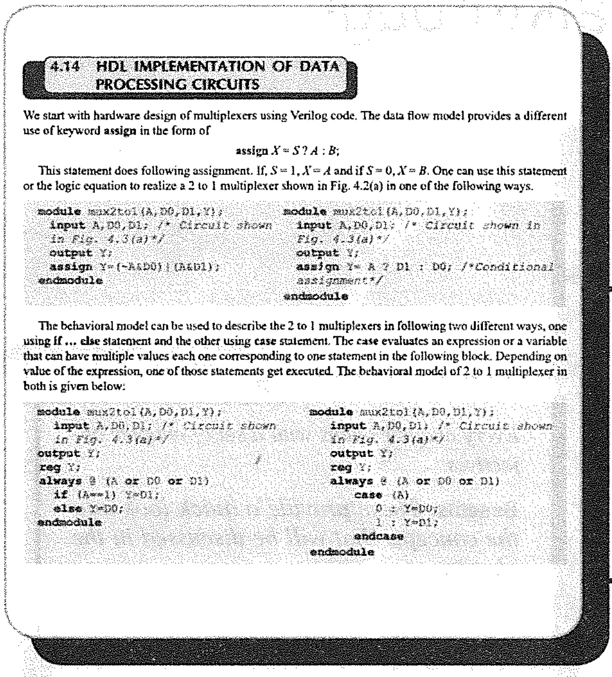


Fig. 4.8 Realization of higher order multiplexers using lower orders

Examples

Every chapter contains several worked out examples totalling to 180 in the book.

Benefits: *These will guide the students while understanding the concepts and working out the exercise problems.*



4.14 HDL IMPLEMENTATION OF DATA PROCESSING CIRCUITS

We start with hardware design of multiplexers using Verilog code. The data flow model provides a different use of keyword **assign** in the form of

assign $X = S\,?\,A : B$;

This statement does following assignment. If, $S = 1$, $X = A$ and if $S = 0$, $X = B$. One can use this statement or the logic equation to realize a 2 to 1 multiplexer shown in Fig. 4.2(a) in one of the following ways.

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown
  in Fig. 4.3(a)*/
  output Y;
  assign Y=(~A&D0)|(A&D1);
endmodule
```

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown in
  Fig. 4.3(a)*/
  output Y;
  assign Y= A ? D1 : D0; /*Conditional
  assignment*/
endmodule
```

The behavioral model can be used to describe the 2 to 1 multiplexers in following two different ways, one using **if ... else** statement and the other using **case** statement. The **case** evaluates an expression or a variable that can have multiple values each one corresponding to one statement in the following block. Depending on value of the expression, one of those statements get executed. The behavioral model of 2 to 1 multiplexer in both is given below:

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown
  in Fig. 4.3(a)*/
output Y;
reg Y;
always @ (A or D0 or D1)
  if (A==1) Y=D1;
  else Y=D0;
endmodule
```

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown
  in Fig. 4.3(a)*/
  output Y;
  reg Y;
  always @ (A or D0 or D1)
     case (A)
       0 : Y=D0;
       1 : Y=D1;
     endcase
endmodule
```

HDL Codes

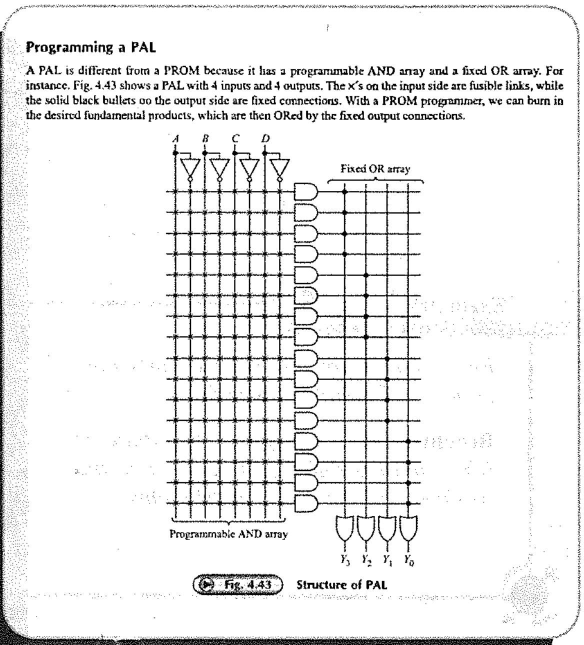
New to this edition, HDL, an interesting development in the field of hardware design, has been introduced.

Benefits: *The relevant HDL description and codes are weaved into chapters to help students implement and design digital circuits.*

Figures

Figures are used exhaustively in the text.

Benefits: *These illustrate the concepts and methods described for better understanding.*



Programming a PAL

A PAL is different from a PROM because it has a programmable AND array and a fixed OR array. For instance, Fig. 4.43 shows a PAL with 4 inputs and 4 outputs. The x's on the input side are fusible links, while the solid black bullets oo the output side are fixed connections. With a PROM programmer, we can burn in the desired fundamental products, which are then ORed by the fixed output connections.

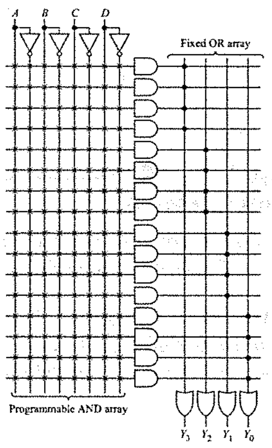
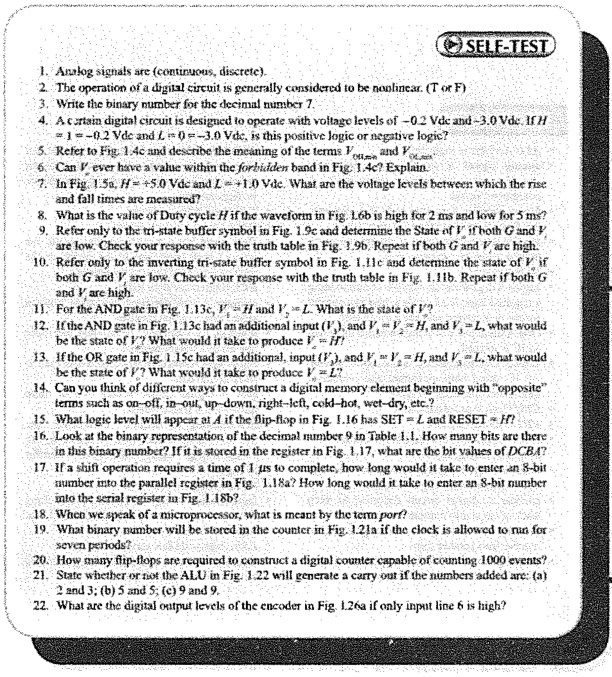


Fig. 4.43 **Structure of PAL**



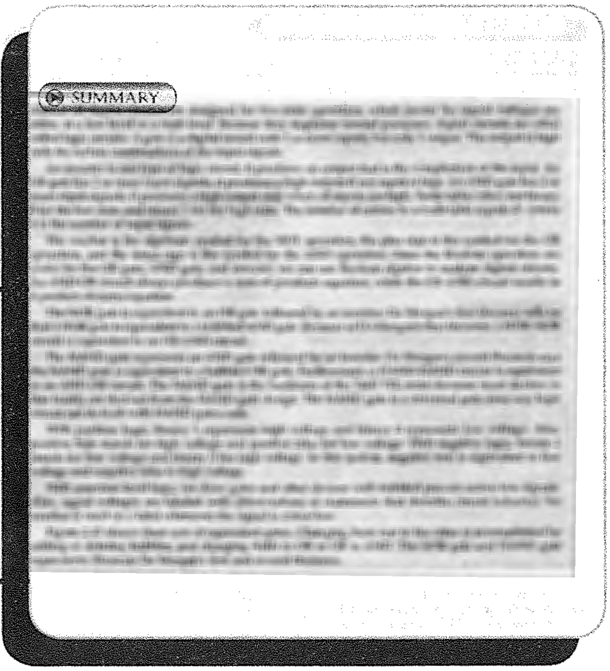
SELF-TEST

1. Analog signals are (continuous, discrete).
2. The operation of a digital circuit is generally considered to be nonlinear. (T or F)
3. Write the binary number for the decimal number 7.
4. A certain digital circuit is designed to operate with voltage levels of −0.2 Vdc and −3.0 Vdc. If $H = 1 = -0.2$ Vdc and $L = 0 = -3.0$ Vdc, is this positive logic or negative logic?
5. Refer to Fig. 1.4c and describe the meaning of the terms $V_{OH,min}$ and $V_{OL,max}$.
6. Can V_o ever have a value within the *forbidden* band in Fig. 1.4c? Explain.
7. In Fig. 1.5a, $H = +5.0$ Vdc and $L = +1.0$ Vdc. What are the voltage levels between which the rise and fall times are measured?
8. What is the value of Duty cycle H if the waveform in Fig. 1.6b is high for 2 ms and low for 5 ms?
9. Refer only to the tri-state buffer symbol in Fig. 1.9c and determine the State of V_o if both G and V_i are low. Check your response with the truth table in Fig. 1.9b. Repeat if both G and V_i are high.
10. Refer only to the inverting tri-state buffer symbol in Fig. 1.11c and determine the state of V_o if both G and V_i are low. Check your response with the truth table in Fig. 1.11b. Repeat if both G and V_i are high.
11. For the AND gate in Fig. 1.13c, $V_1 = H$ and $V_2 = L$. What is the state of V_o?
12. If the AND gate in Fig. 1.13c had an additional input (V_3), and $V_1 = V_2 = H$, and $V_3 = L$, what would be the state of V_o? What would it take to produce $V_o = H$?
13. If the OR gate in Fig. 1.15c had an additional, input (V_3), and $V_1 = V_2 = H$, and $V_3 = L$, what would be the state of V_o? What would it take to produce $V_o = L$?
14. Can you think of different ways to construct a digital memory element beginning with "opposite" terms such as on–off, in–out, up–down, right–left, cold–hot, wet–dry, etc.?
15. What logic level will appear at A if the flip-flop in Fig. 1.16 has SET = L and RESET = H?
16. Look at the binary representation of the decimal number 9 in Table 1.1. How many bits are there in this binary number? If it is stored in the register in Fig. 1.17, what are the bit values of $DCBA$?
17. If a shift operation requires a time of 1 μs to complete, how long would it take to enter an 8-bit number into the parallel register in Fig. 1.18a? How long would it take to enter an 8-bit number into the serial register in Fig. 1.18b?
18. When we speak of a microprocessor, what is meant by the term *port*?
19. What binary number will be stored in the counter in Fig. 1.21a if the clock is allowed to run for seven periods?
20. How many flip-flops are required to construct a digital counter capable of counting 1000 events?
21. State whether or not the ALU in Fig. 1.22 will generate a carry out if the numbers added are: (a) 2 and 3; (b) 5 and 5; (c) 9 and 9.
22. What are the digital output levels of the encoder in Fig. 1.26a if only input line 6 is high?

Self-Test

A section called Self-Test appears after every section in every chapter.

Benefits: *This will help students check their understanding of the concepts discussed in a section before moving on to the next section. Answers to Self-Tests are given at the end of that chapter.*



SUMMARY

[illegible]

Summary

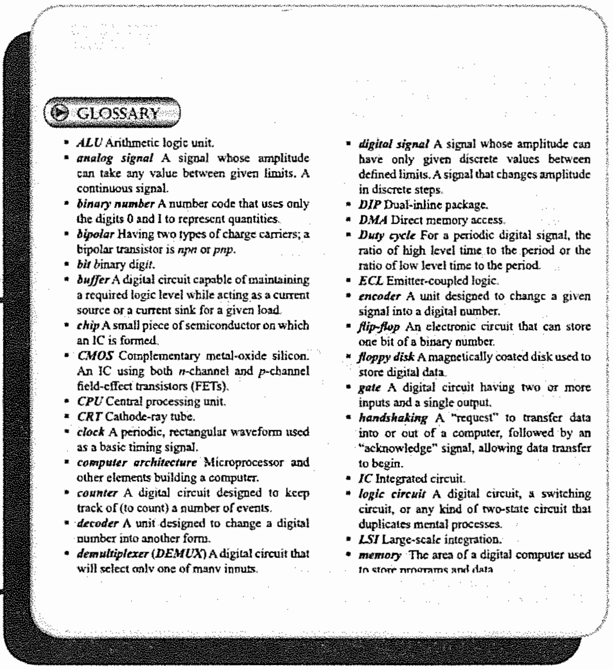
A Brief summary is provided at the end of the chapters.

Benefits: *Summary gives the essence of each chapter in brief and will be helpful for a quick review during the examinations.*

Glossary

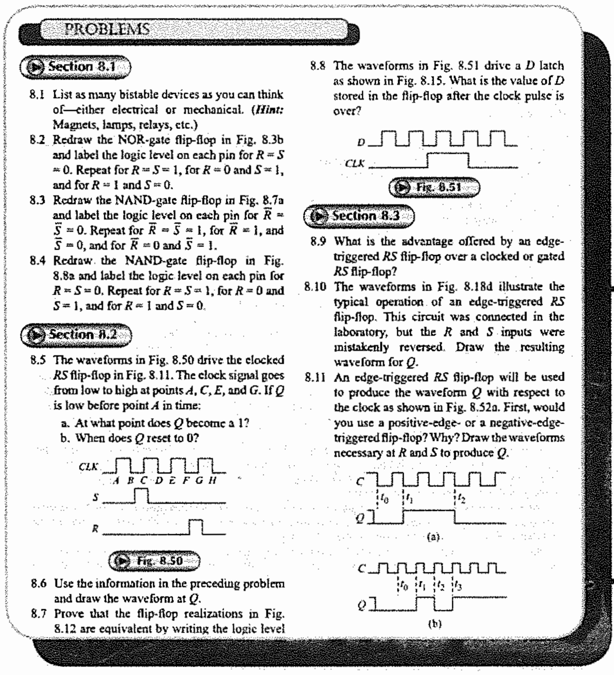
A glossary containing the important definitions and abbreviations is listed at the end of each chapter.

Benefits: *It helps in memorising the important terms discussed in the chapter.*



GLOSSARY

- ***ALU*** Arithmetic logic unit.
- ***analog signal*** A signal whose amplitude can take any value between given limits. A continuous signal.
- ***binary number*** A number code that uses only the digits 0 and 1 to represent quantities.
- ***bipolar*** Having two types of charge carriers; a bipolar transistor is *npn* or *pnp*.
- ***bit*** binary digit.
- ***buffer*** A digital circuit capable of maintaining a required logic level while acting as a current source or a current sink for a given load.
- ***chip*** A small piece of semiconductor on which an IC is formed.
- ***CMOS*** Complementary metal-oxide silicon. An IC using both *n*-channel and *p*-channel field-effect transistors (FETs).
- ***CPU*** Central processing unit.
- ***CRT*** Cathode-ray tube.
- ***clock*** A periodic, rectangular waveform used as a basic timing signal.
- ***computer architecture*** Microprocessor and other elements building a computer.
- ***counter*** A digital circuit designed to keep track of (to count) a number of events.
- ***decoder*** A unit designed to change a digital number into another form.
- ***demultiplexer (DEMUX)*** A digital circuit that will select only one of many inputs.
- ***digital signal*** A signal whose amplitude can have only given discrete values between defined limits. A signal that changes amplitude in discrete steps.
- ***DIP*** Dual-inline package.
- ***DMA*** Direct memory access.
- ***Duty cycle*** For a periodic digital signal, the ratio of high level time to the period or the ratio of low level time to the period.
- ***ECL*** Emitter-coupled logic.
- ***encoder*** A unit designed to change a given signal into a digital number.
- ***flip-flop*** An electronic circuit that can store one bit of a binary number.
- ***floppy disk*** A magnetically coated disk used to store digital data.
- ***gate*** A digital circuit having two or more inputs and a single output.
- ***handshaking*** A "request" to transfer data into or out of a computer, followed by an "acknowledge" signal, allowing data transfer to begin.
- ***IC*** Integrated circuit.
- ***logic circuit*** A digital circuit, a switching circuit, or any kind of two-state circuit that duplicates mental processes.
- ***LSI*** Large-scale integration.
- ***memory*** The area of a digital computer used to store programs and data



PROBLEMS

Section 8.1

8.1 List as many bistable devices as you can think of—either electrical or mechanical. (***Hint:*** Magnets, lamps, relays, etc.)

8.2 Redraw the NOR-gate flip-flop in Fig. 8.3b and label the logic level on each pin for $R = S = 0$. Repeat for $R = S = 1$, for $R = 0$ and $S = 1$, and for $R = 1$ and $S = 0$.

8.3 Redraw the NAND-gate flip-flop in Fig. 8.7a and label the logic level on each pin for $\bar{R} = \bar{S} = 0$. Repeat for $\bar{R} = \bar{S} = 1$, for $\bar{R} = 1$, and $\bar{S} = 0$, and for $\bar{R} = 0$ and $\bar{S} = 1$.

8.4 Redraw the NAND-gate flip-flop in Fig. 8.8a and label the logic level on each pin for $R = S = 0$. Repeat for $R = S = 1$, for $R = 0$ and $S = 1$, and for $R = 1$ and $S = 0$.

Section 8.2

8.5 The waveforms in Fig. 8.50 drive the clocked *RS* flip-flop in Fig. 8.11. The clock signal goes from low to high at points *A*, *C*, *E*, and *G*. If *Q* is low before point *A* in time:
a. At what point does *Q* become a 1?
b. When does *Q* reset to 0?

Fig. 8.50

8.6 Use the information in the preceding problem and draw the waveform at *Q*.

8.7 Prove that the flip-flop realizations in Fig. 8.12 are equivalent by writing the logic level

8.8 The waveforms in Fig. 8.51 drive a *D* latch as shown in Fig. 8.15. What is the value of *D* stored in the flip-flop after the clock pulse is over?

Fig. 8.51

Section 8.3

8.9 What is the advantage offered by an edge-triggered *RS* flip-flop over a clocked or gated *RS* flip-flop?

8.10 The waveforms in Fig. 8.18d illustrate the typical operation of an edge-triggered *RS* flip-flop. This circuit was connected in the laboratory, but the *R* and *S* inputs were mistakenly reversed. Draw the resulting waveform for *Q*.

8.11 An edge-triggered *RS* flip-flop will be used to produce the waveform *Q* with respect to the clock as shown in Fig. 8.52a. First, would you use a positive-edge- or a negative-edge-triggered flip-flop? Why? Draw the waveforms necessary at *R* and *S* to produce *Q*.

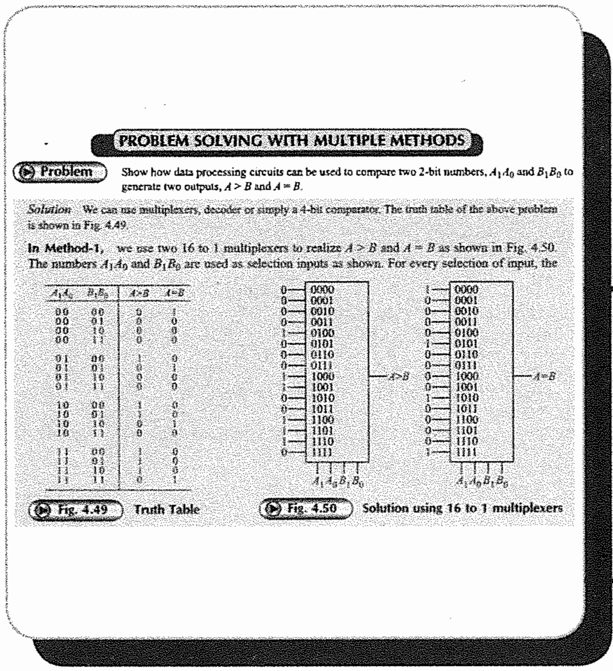
(a)

(b)

Problems

The text contains more than 250 section-end practice problems.

Benefits: *These will help the students in improving their problem-solving skills.*



PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Show how data processing circuits can be used to compare two 2-bit numbers, A_1A_0 and B_1B_0 to generate two outputs, $A > B$ and $A = B$.

Solution We can use multiplexers, decoder or simply a 4-bit comparator. The truth table of the above problem is shown in Fig. 4.49.

In Method-1, we use two 16 to 1 multiplexers to realize $A > B$ and $A = B$ as shown in Fig. 4.50. The numbers A_1A_0 and B_1B_0 are used as selection inputs as shown. For every selection of input, the

A_1A_0	B_1B_0	$A>B$	$A=B$
00	00	0	1
00	01	0	0
00	10	0	0
00	11	0	0
01	00	1	0
01	01	0	1
01	10	0	0
01	11	0	0
10	00	1	0
10	01	1	0
10	10	0	1
10	11	0	0
11	00	1	0
11	01	1	0
11	10	1	0
11	11	0	1

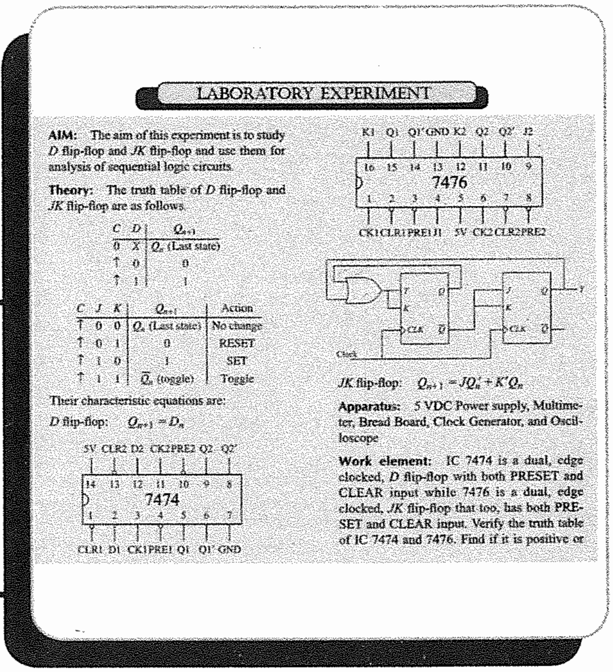
Fig. 4.49 Truth Table

Fig. 4.50 Solution using 16 to 1 multiplexers

Problem Solving with Multiple Methods

Each chapter contains numerous problems solved using multiple methods.

Benefits: *Problem solving by multiple methods helps students in understanding and appreciating different alternatives to reach a solution, without feeling stuck at any point of time.*



LABORATORY EXPERIMENT

AIM: The aim of this experiment is to study *D* flip-flop and *JK* flip-flop and use them for analysis of sequential logic circuits.

Theory: The truth table of *D* flip-flop and *JK* flip-flop are as follows.

C	D	Q_{n+1}
0	X	Q_n (Last state)
↑	0	0
↑	1	1

C	J	K	Q_{n+1}	Action
↑	0	0	Q_n (Last state)	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	$\overline{Q}_n$ (toggle)	Toggle

Their characteristic equations are:

D flip-flop: $Q_{n+1} = D_n$

5V CLR2 D2 CK2PRE2 Q2 Q2'
14 13 12 11 10 9 8
7474
1 2 3 4 5 6 7
CLR1 D1 CK1PRE1 Q1 Q1' GND
K1 Q1 Q1' GND K2 Q2 Q2' J2
16 15 14 13 12 11 10 9
7476
1 2 3 4 5 6 7 8
CK1 CLR1 PRE1 J1 5V CK2 CLR2 PRE2
J
K
CLK
Q
$\overline{Q}$
Y
Clock

JK flip-flop: $Q_{n+1} = JQ_n' + K'Q_n$

Apparatus: 5 VDC Power supply, Multimeter, Bread Board, Clock Generator, and Oscilloscope

Work element: IC 7474 is a dual, edge clocked, *D* flip-flop with both PRESET and CLEAR input while 7476 is a dual, edge clocked, *JK* flip-flop that too, has both PRESET and CLEAR input. Verify the truth table of IC 7474 and 7476. Find if it is positive or

Laboratory Experiments

Each chapter contains a lab experiment.

Benefits: *Laboratory experiments facilitate experimentation with different analysis and synthesis problems using digital integrated circuits (IC). These give a hands-on experience to the reader.*

Digital Principles

1

OBJECTIVES

- Understand the difference between analog and digital signals, recognize binary equivalents of decimal numbers 0 to 15, and be familiar with basic terminology related to digital waveforms.
- Based on input conditions, determine the output of a buffer, a tri-state buffer, an inverter, a tri-state inverter, an AND gate, and an OR gate.
- Discuss several ways of how digital information (bits) can be stored and transferred and describe some of the operations of an ALU.
- Recognize digital logic symbols and identify fundamental difference, in operation and logic levels between major IC families.

In the modern world of electronics, the term *digital* is probably most often associated with a *computer*. It certainly is difficult to think of an area of life today that is not influenced in one way or another by a digital computer. Checking and savings accounts at a bank, automobile insurance, credit card accounts, federal and state income taxes, airline tickets—the list of functions controlled by large computer systems seems almost endless! In addition to these large systems, the hand calculator, the IBM or IBM clone personal computer (PC), the Apple family of computers, and a host of other desktop computer systems are readily available at a reasonable cost to virtually anyone. The availability of such computational power can be traced directly to the development of the digital integrated circuit (IC).

The semiconductor industry provided the first commercially available families of digital ICs in the early 1960s. These devices were used to develop smaller, faster, more economical, and more powerful digital computers. They were also used in a great many other applications. Today, digital circuits and systems can be found in almost every field of electronics. In communications, the principles of digital electronics are found in satellites, telephone switching and transmission networks, and navigation systems. Digital circuits in the area of consumer electronics are found in compact discs, VCRs, and television. Process controls in industrial applications, and electronic systems used in medicine have benefited greatly from advances in digital electronics. The list will no doubt continue to expand.

An introduction to the field of digital electronics cannot cover all possible applications, but a collection of basic principles can be identified. These *digital principles* are the basis for this text, and they along with a number of *applications* are intended to provide the background for you to succeed in the modern world of digital electronics.

This chapter presents some distinctive features of the world of digital electronics. It defines digital signals and terminologies associated with digital waveform; discusses digital logic and basic operations on digital data; introduces the concept of digital IC, its signal levels and noise margin.

1.1 DEFINITIONS FOR DIGITAL SIGNALS

Analog versus Digital

Electronic circuits and systems can be conveniently divided into two broad categories generally referred to as *analog* and *digital*. Analog circuits, designed for use with small signals, can be made to work in a linear fashion. An operational amplifier (op amp) connected as an amplifier with a voltage gain of 10 is an analog circuit. The output voltage for this circuit will be a faithfully amplified version of any signal presented at its input till saturation is reached. This is *linear operation*. Digital circuits are generally used with *large signals* and are considered nonlinear. One example is a remote control circuit that switches the lights in a parking area on after sunset and turns them off at sunrise. In this case, the input signal might be a voltage representing the time of day or it might be a current taken from a light-sensing circuit. The output signal is simply *on* or *off*, which is clearly not an amplified version of the input signal. This is *nonlinear operation*.

Any quantity that changes with time either can be represented as an analog signal or it can be treated as a digital signal. For example, place a container of water at room temperature on a stove and apply heat. The measurable quantity of interest here is the change in water temperature. There are two ways to record the water temperature over a period of time. In Fig. 1.1a, the temperature is recorded *continuously*, and it changes smoothly from 20°C to 80°C. While being heated, the water temperature passes through every possible

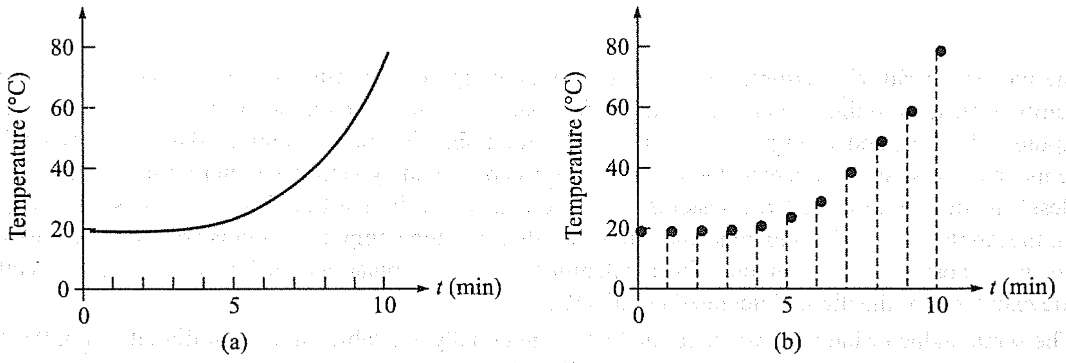


Fig. 1.1 (a) An analog (continuous) signal, (b) A digital (discrete) signal

value between 20°C and 80°C. This is an example of an analog signal. *Analog signals* are continuous and all possible values are represented.

If the water temperature is measured and recorded only once every minute, the temperature is recorded as in Fig. 1.1b. In this case, the recorded temperature is *not* continuous. Rather, it *jumps* from point to point, and there are only a finite number of values between 20°C and 80°C say, at an increment of 1°C like 20°C, 21°C, 22°C, and so on. There are exactly 11 values in this case. When a quantity is recorded as a series of distinct (discrete) points, it is said to be *sampled*. This is an example of a digital signal. *Digital signals* represent only a finite number of discrete values.

Virtually all naturally occurring physical phenomena are analog signals. Temperature, pressure, velocity, and sound, for instance, are signals that take on all possible values between given limits. These signals can be conditioned and operated on with the use of analog electronic circuits. For example, an analog power amplifier is used to amplify a music signal and drive a set of stereo speakers. Digital circuits and systems can be used to process both analog signals and digital signals. As an example, both music and speech are readily translated into digital signals for use in a digital stereo system. At the same time, digital circuits can be used to perform functions unrealizable with analog circuits—counting, for instance.

Binary System

Digital electronics today involves circuits that have exactly two possible states. A system having only two states is said to be *bi*nary (*bi* means "two"). The *binary number system* has exactly two symbols—0 and 1. As you might expect, the binary number system is widely used in digital electronics. We will consider the binary number system in much greater detail later, but for immediate reference, the first 16 binary numbers and their decimal equivalents are shown in Table 1.1.

Table 1.1

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

The operation of an electronic circuit can be described in terms of its voltage levels. In the case of a digital circuit, there are only two. Clearly one voltage is more positive than the other. The more positive voltage is the *high (H)* level, and the other is the *low (L)* level. This is immediately related to the binary number system by assigning $L = 0$ and $H = 1$. Many functions performed by digital circuits are *logical operations*, and thus the terms true (T) and false (F) are often used. Choosing $H = 1 = T$ and $L = 0 = F$ is called *positive logic*. The majority of digital systems utilize positive logic. Note that it is also possible to construct a *negative logic* system by choosing $H = 0 = F$ and $L = 1 = T$.

Today the majority of digital circuit families utilize a single +5 Vdc power supply, and the two voltage levels used are +5 Vdc and 0 Vdc. Here is a summary of the two binary states (levels) in this positive logic system.

$$+5\ \text{Vdc} = H = 1 = T$$
$$0\ \text{Vdc} = L = 0 = F$$

You can no doubt see how to extend these definitions to include terms such as *on–off go–no go, yes–no*, and so on. A lamp or a *light-emitting* diode (LED) is frequently used to indicate a digital signal. *On* (illuminated) represents 1, and *off* (extinguished) represents 0. As an example, the four LEDs in Fig. 1.2 are indicating the binary number 0101, which is equivalent to decimal 5.

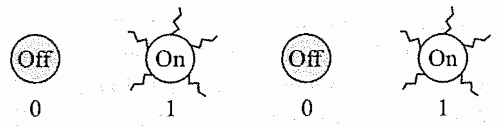


Fig. 1.2 The binary number 0101 (decimal 5)

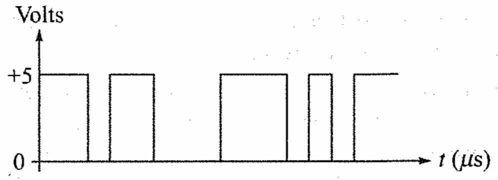


Fig. 1.3 An ideal digital signal

Ideal Digital Signals

The voltage levels in an *ideal* digital circuit will have values of either +5 Vdc or 0 Vdc. Furthermore, when the voltages change (switch) between values, they do so in zero time!

These concepts are illustrated in Fig. 1.3, which represents an arbitrary digital signal whose level changes with time. As we shall see in Sec. 1.2, *actual* digital signals depart somewhat from this ideal.

SELF-TEST

1. Analog signals are (continuous, discrete).
2. The operation of a digital circuit is generally considered to be nonlinear. (T or F)
3. Write the binary number for the decimal number 7.
4. A certain digital circuit is designed to operate with voltage levels of −0.2 Vdc and −3.0 Vdc. If $H = 1 = -0.2$ Vdc and $L = 0 = -3.0$ Vdc, is this positive logic or negative logic?

1.2 DIGITAL WAVEFORMS

The ideal digital signal represented in Fig. 1.3 has two precise voltage levels—+5 Vdc and 0 Vdc. Furthermore, the signal switches from one level to the other in zero time. In reality, modern digital circuits can produce signals that approach, but do not quite attain, this ideal behavior.

Voltage Levels

First of all, the output voltage level of any digital circuit depends somewhat on its load, as illustrated in Fig. 1.4a below. When V_0 is high, the voltage should be +5 Vdc. In this case, the digital circuit must act as a *current source* to deliver the current I_o to the load. However, the circuit may not be capable of delivering the necessary current I_o while maintaining +5 Vdc. To account for this, it is agreed that any output voltage close to +5 Vdc within a certain range will be considered high. This is illustrated in Fig. 1.4c, where any output voltage level between +5 Vdc and $V_{\text{OH,min}}$ is defined as $H = 1 = T$. The term $V_{\text{OH,min}}$ stands for the minimum value of the output voltage when high. As we will see, one popular transistor-transistor logic (TTL) family of digital circuits allows $V_{\text{OH,min}} = +3.5$ Vdc. In this case, any voltage level between +5 Vdc and +3.5 Vdc is $H = 1$.

In Fig. 1.4b. V_o is low, and the digital circuit must act as a *current sink*. That is, it must be capable of accepting a current I_o from the load and delivering it to ground. In this situation, V_o, should be 0 Vdc, but the

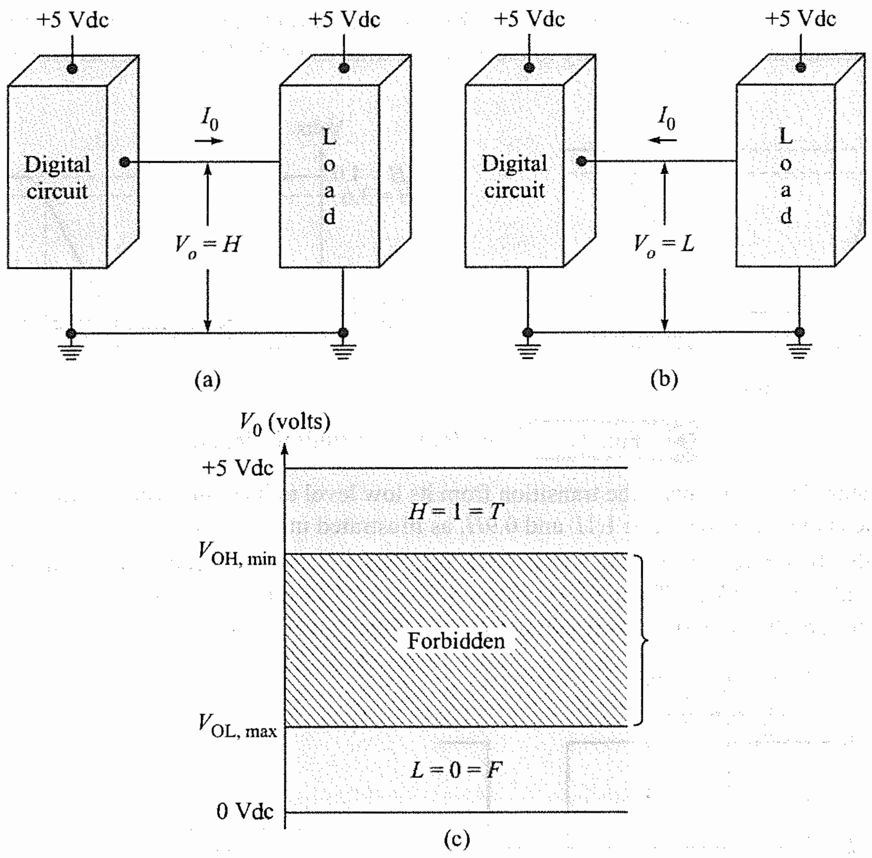


Fig. 1.4 **Loading of digital circuit**

circuit may not be capable of this. So it is agreed to accept any output voltage that is close to 0 Vdc within certain limits as the low level. This is illustrated in Fig. 1.4c, where any output voltage level between 0 Vdc and $V_{OL,max}$, is defined as $L = 0 = F$. The term $V_{OL,max}$ stands for the *maximum* value of the output voltage when low. Again, the popular TTL family mentioned above allows $V_{OL,max} = +0.1$ Vdc. Thus, any voltage level between +0.1 Vdc and 0 Vdc is $L = 0$.

The diagram in Fig. 1.4c clearly shows that the digital signals being used here require a high-level voltage somewhere in the band labeled H. A low-level voltage must be somewhere in the band labeled L. Furthermore, *no other voltage levels are permitted!*

Switching Time

If the digital circuit in Fig. 1.4 were *ideal*, it would change from high to low, or from low to high, in zero time. Thus, the output voltage would never have a value in the forbidden range. In reality, it does, in fact, require a finite amount of time for V_o to make the transition (switch) between levels. As a result, the voltage V_o versus time might appear as in Fig 1.5a. Clearly V_o does take on values in the forbidden range between the high and low band—but only for a very short time, and only while switching! When it is not switching, V_o remains within the high or the low band as required.

The time required for V_o to make the transition from its high level to its low level is defined as *fall time* t_f. For ease of measurement, it is customary to measure fall time using $0.9H$ and $1.1L$, as shown in Fig. 1.5a.

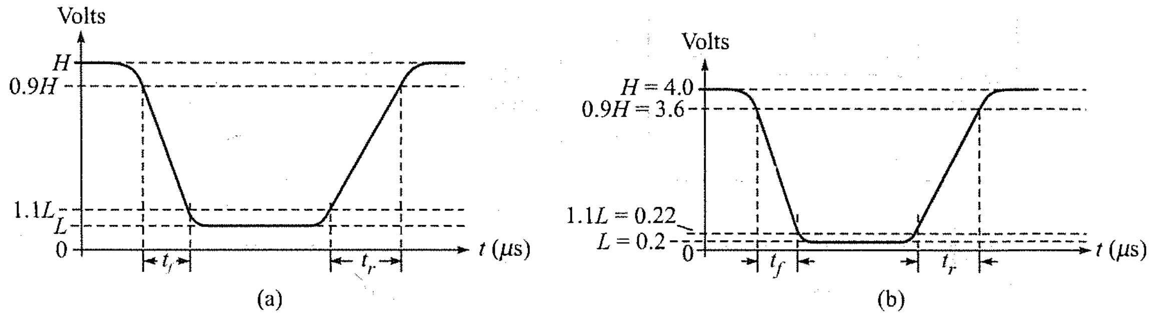


Fig. 1.5 Switching in digital circuit

The time required for V_0 to make the transition from its low level to its high level is defined as *rise time* t_r. Again, rise time is measured between $1.1L$ and $0.9H$, as illustrated in Fig. 1.5a.

Figure 1.5b illustrates how rise time and fall time are measured. For example, suppose $H = +4.0$ Vdc and $L = +0.2$ Vdc. Then, $0.9H = 0.9 \times 4.0 = +3.6$ Vdc, and $1.1L = 1.1 \times 0.2 = +0.22$ Vdc. The rise and fall times are then measured between these two voltage levels as shown.

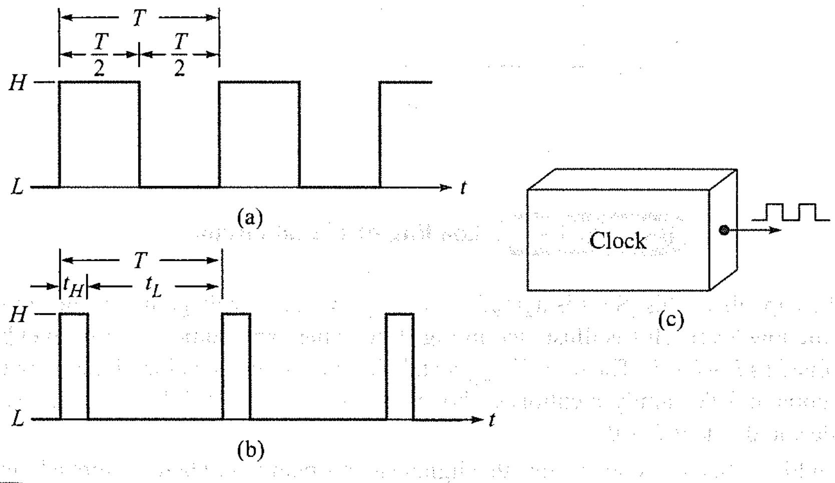


Fig. 1.6 (a) Symmetrical signal with period T, (b) Asymmetrical signal with period T, (c) System clock

Period and Frequency

There are many occasions where a symmetrical digital signal as in Fig. 1.6a will be used (clock and counter circuits for instance). The period T of this waveform is shown. This is the time over which the signal repeats itself. A rectangular waveform such as this can be produced by adding together an infinite (or at least a large number) of sinusoidal waveforms of different frequencies and amplitudes. Even though this digital signal is not sinusoidal, it is convenient to define the frequency as $f = 1/T$. As an example, if the period of this square wave is 1 μs, then its frequency is found as

$$f = \frac{1}{T} = \frac{1}{1\ \mu s} = \frac{1}{10^{-6}} = 10^6 = 1\text{ MHz}$$

The digital waveform in Fig. 1.6b is not a square wave; that is, it is not symmetrical. Even so, its frequency is still found as the reciprocal of its period i.e. $f = 1/T$.

A symmetrical signal as illustrated in Fig. 1.6a or Fig. 1.6b is frequently used as the basis for timing all operations in a digital system. As such, it is called the *clock signal*. The electronic circuit used to generate this square wave is referred to as the *system clock*, as illustrated in Fig. 1.6c. A system clock is simply an oscillator circuit having a very precise frequency. Frequency stability is provided by using a crystal as the frequency-determining element.

Duty Cycle

Duty cycle is a convenient measure of how symmetrical or how unsymmetrical a waveform is. For the waveform in Fig. 1.6b, there are two possible definitions for duty cycle.

$$\text{Duty cycle } H = \frac{t_H}{T}$$

$$\text{Duty cycle } L = \frac{t_L}{T}$$

The first definition is the fraction of time the signal is high, and the second is the fraction of time the signal is low. Either definition is acceptable, provided you clearly define which you are using. To express as a percentage, simply multiply by 100.

Note that the duty cycle for a symmetrical wave as in Fig. 1.6a is

$$\text{Duty cycle } H = \text{Duty cycle } L = \frac{T/2}{T} = 0.5 \text{ or } (50\%)$$

Example 1.1 The waveform in Fig. 1.6b has a frequency of 5 MHz, and the width of the positive pulse is 0.05 μs. What is the high duty cycle (Duty cycle H)?

Solution The period of the waveform is found as

$$T = \frac{1}{f} = \frac{1}{(5\text{ MHz})} = \frac{1}{5 \times 10^6} = 0.2\ \mu s$$

Then

$$\text{Duty cycle } H = \frac{0.05\ \mu s}{0.2\ \mu s} = 0.25 = 25\%$$

SELF-TEST

5. Refer to Fig. 1.4c and describe the meaning of the terms $V_{OH,min}$ and $V_{OL,mix}$.
6. Can V_o ever have a value within the *forbidden* band in Fig. 1.4c? Explain.
7. In Fig. 1.5a, $H = +5.0$ Vdc and $L = +1.0$ Vdc. What are the voltage levels between which the rise and fall times are measured?
8. What is the value of Duty cycle H if the waveform in Fig. 1.6b is high for 2 ms and low for 5 ms?

1.3 DIGITAL LOGIC

Generating Logic Levels

The digital voltage levels described in Fig. 1.4 can be produced using switches as illustrated in Fig. 1.7 on the next page. In Fig. 1.7a, the switch is down and $V_o = L = 0 = 0$ Vdc. When the switch is up, as in Fig. 1.7b, $V_o = H = 1 = +5$ Vdc. A switch is easy to use and easy to understand, but it must be operated by hand.

A *relay* is a switch that is actuated by applying a voltage V_i to a coil as shown in Fig. 1.7c. The coil current develops a magnetic field that moves the switch arm from one contact to the other. This is indicated with the dashed line drawn between the coil and the relay arm. For this particular relay, $V_o = L = 0$ Vdc when $V_i = 0$ Vdc. Applying a voltage V_i will actuate the relay, and then $V_o = H = +5$ Vdc. This relay could of course be connected so that its output is low when actuated.

Switches and relays were useful in the construction of early machines used for calculation and/or logic operations. In fact, they are still used to a limited extent in modern computer systems where humans must interact with a system. For instance, on–off power switches, reset, start–stop, and load–unload are functions that might require human initiation.

On the other hand, modern computers are capable of performing billions of switching operations every second! Switches and relays are clearly not capable of this performance, and they have therefore been replaced by transistors (bipolar and/or MOSFET). A *digital integrated circuit (IC)* is constructed using numerous transistors and resistors, and each is designed to perform a given logic operation. On an IC, each transistor is used as an electronic switch. Let's use the simple switch models shown in Fig. 1.7 to define some basic logic circuits. Later on, we'll take the time to look at the actual ICs and discuss circuit operation in detail.

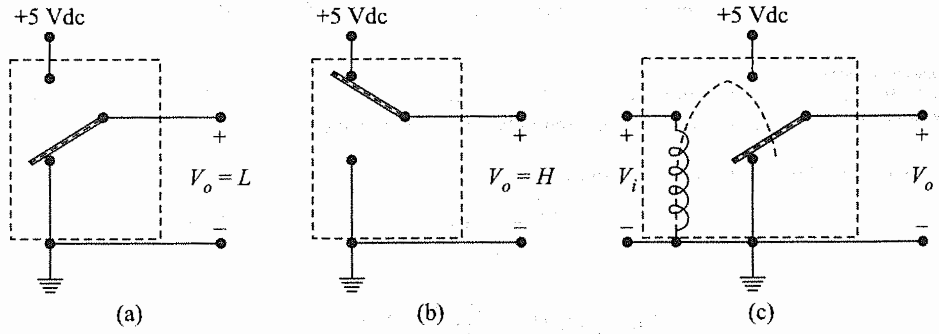
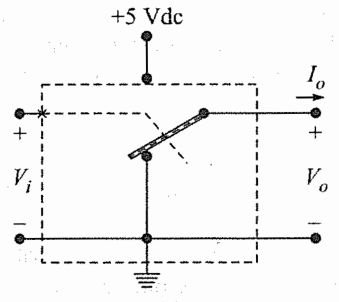


Fig. 1.7 **(a) Switch, (b) Switch, (c) Normally low relay**

The Buffer

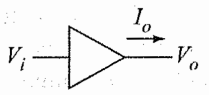
In order to deliver the necessary load current I_o in Fig. 1.4, a digital IC called a *buffer* might be used. A buffer can be thought of as an electronic switch, as shown in Fig. 1.8a. The switch is actuated by the input voltage V_i. Its operation is similar to the relay in Fig. 1.7c. When V_i is low, the switch is down, and V_o is low. On the other hand, when V_i is high, the switch moves up and V_o is high. Operation of this IC is summarized by using the truth table, or table of combinations, shown in Fig. 1.8b. There are only two possible input voltage levels (L and H), and the truth table shows the value of V_o in each case.



(a)

V_i	V_o
L	L
H	H

(b)



(c)

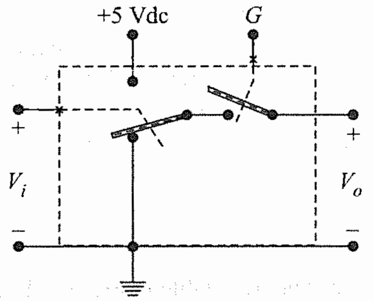
Fig. 1.8 **(a) Buffer amplifier model, (b) Truth table, (c) Symbol**

Since the buffer is capable of delivering additional current to a load, it is often called a *buffer amplifier*. The traditional amplifier symbol (a triangle) shown in Fig. 1.8c is used on schematic diagrams. If you're interested in an actual IC buffer, look in the standard TTL logic family. The 5407 or 7407 is a 14-pin IC that contains six buffers.

The Tri-State Buffer

At the input of a digital system, there may be more than one input signal of interest. Generally speaking, however, it will be necessary to connect only one signal at a time, and thus there is a requirement to connect or disconnect (switch) input signals electronically. Similarly, the output of a digital system may need to be directed to more than one destination, one at a time.

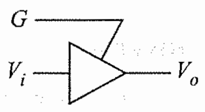
The logic circuit in Fig. 1.9a is a simple buffer with an additional switch controlled by an input labeled *G*. When *G* is low, this switch is open and the output is "disconnected" from the buffer. When *G* is high, the switch is closed and the output follows the input. That is, the circuit behaves as an ordinary buffer amplifier. In effect, the control signal *G* connects the buffer to the load or disconnects the buffer from the load.



(a)

V_i	G	V_o
L	L	Open
H	L	Open
L	H	L
H	H	H

(b)



(c)

Fig. 1.9 **A tri-state buffer: (a) Model, (b) Truth table, (c) Symbol**

The truth table in Fig. 1.9b summarizes circuit operation. Notice that when *G* is high, V_o is either high or low (two states). However, when *G* is low, the output is in effect an *open circuit* (a third state). Since there are *three* possible states for V_o, this circuit is called a *tri-state buffer*. (*Tri* stands for "three", and thus the term *three-state buffer* is often used.)

The standard symbol for a tri-state buffer such as this is shown in Fig. 1.9c. It is simply the buffer symbol with an additional input, G. Since G controls operation of the circuit, it is often referred to as the *enable input.* In the standard TTL logic family, a 54126 or a 74126 is a 14-pin IC that has four of these circuits.

The Inverter

One of the most basic operations in a digital system is *inversion*, or *negation*. This requires a circuit that will invert a digital level. This logic circuit is called an *inverter*, or sometimes a *NOT circuit*. The switch arrangement in Fig. 1.10a is an inverter. When the input to this circuit is low, the switch remains up and the output is high. When the input is high, the switch moves down and the output is low. The truth table for the inverter is given in Fig. 1.10b Clearly the output is the negative, or the inverse, of the input.

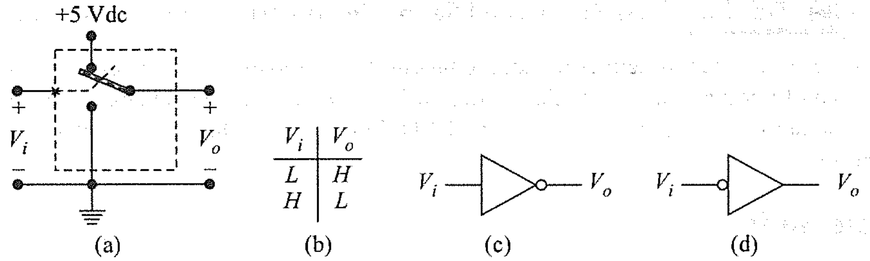


Fig. 1.10 Digital inverter: (a) Model, (b) Truth table, (c) Symbol, (d) Another symbol

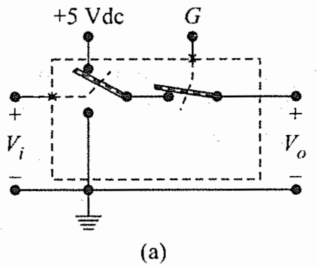
When the inverter is used as a logic circuit, H is often defined as the "true" state, while L is defined as the "false" state. In this sense, the inverter will always provide at its output a signal that is the inverse, or complement, of the signal at its input. It is thus called a *negation* or NOT circuit. This makes sense, since there are only two possible states, and therefore NOT H must be L and NOT L must be H.

The inverse or complement of a signal is shown by writing a bar above the symbol. For instance, the complement of A is written as $\bar{A}$ or A' and this is read as "A bar" A logic expression for the inverter in Fig. 1.10c is $V_o = \overline{V_i}$. It is read "V sub oh is equal to V sub eye bar."

The standard symbol for an inverter is given in Fig. 1.10c. Notice the small circle (bubble) at the output. This small circle signifies inversion, and it is used on many other logic symbols. For instance, the symbol in Fig. 1.10d has the small circle on the input side. This is still an inverter, but the circle on the input side has additional significance, which will be considered next. In the standard TTL logic family, a 5404 or 7404 is a 14-pin IC with six inverters.

The Tri-state Inverter

A *tri-state inverter* is easy to construct, as shown in Fig. 1.11a. The truth table in Fig. 1.11b shows that when G is low, the inverter is connected to the output. When G is high, the enable switch opens, and the output is disconnected from the inverter. The standard logic symbol for this tri-state inverter is given in Fig. 1.11c. The inverting amplifier symbol indicates that V_o is the inverse of V_i (the small circle is at the amplifier output). However, note the small circle at the input of the amplifier used for G. From the truth table, you can see that the switch controlled by G is closed when G is low! Thus, when G is low, the circuit is activated and output V_o is the inverse of the input V_i. Compare this with the G input to the tri-state buffer in Fig. 1.9. In this case, the switch is closed and the circuit is activated when G is high. Here, then, is the significance of the circle on the input side: *Placing a circle at the input of a logic circuit means that circuit is activated when the* *signal at that input is low*! The tri-state inverters used on the 74LS386A IC (TTL logic family) are similar to Fig. 1.11.



V_i	G	V_o
L	L	H
H	L	L
L	H	Open
H	H	Open

(b)

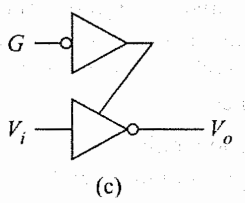


Fig. 1.11 **Inverting tri-state buffer: (a) Model, (b) Truth table, (c) Symbol**

The AND Gate

An *AND gate* is a digital circuit having two or more inputs and a single output as indicated in Fig. 1.12. The inputs to this gate are labeled V_1, V_2, V_3, ... V_n (there are n inputs), and the output is labeled V_o. The operation of an AND gate can be expressed in a number of different, but equivalent, ways. For instance,

1. If *any* input is low, V_o will be low.
2. V_o will be high only when all inputs are high.
3. $V_o = H$ only if $V_1 = H$, and $V_2 = H$, and $V_3 = H$, . . . and $V_n = H$.

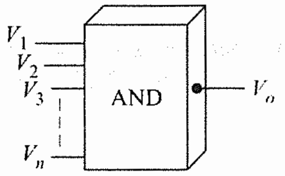
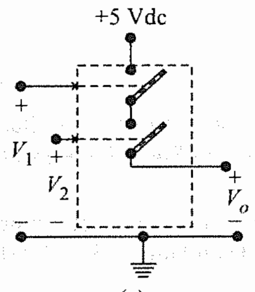


Fig. 1.12 **AND gate**

This last statement leads to the designation *AND gate*, since V_1 *and* V_2, *and* V_3, ... *and* V_n must all be high in order for V_o to be high.

A model for an AND gate having 2 inputs is shown in Fig. 1.13a. This gate can be used to make "logical" decisions; for example, "If V_1 and V_2, then V_o." As a result, it is referred to as a digital logic circuit, as are all AND gates. From the model, it is seen that $V_1 = H$ closes the upper switch, and $V_2 = H$ closes the lower switch. Clearly, $V_o = H$ only when both V_1 and V_2 are high. This can be expressed in the form of a logic equation written as

$$V_o = V_1 \text{ AND } V_2$$



V_1	V_2	V_o
L	L	L
H	L	L
L	H	L
H	H	H

(b)

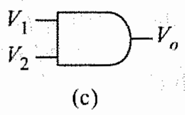


Fig. 1.13 **Two-input AND gate: (a) Model, (b) Truth table, (c) Symbol**

The operation is summarized in the truth table in Fig. 1.13b. The symbol for a 2-input AND gate is shown in Fig. 1.13c. Thus, AND is a logic operation which is realized here through a logic gate.

The OR Gate

An *OR gate* is also a digital circuit having 2 or more inputs and a single output as indicated in Fig. 1.14. The inputs to this gate are labeled V_1, V_2, V_3, … V_n, (there are n inputs), and the output is labeled V_o. The operation of an OR gate can be expressed in a number of different ways. For instance,

1. V_o will be low only when all inputs are low.
2. If any input is high, V_o will be high.
3. $V_o = H$ if V_1, *or* V_2 *or* V_3, … *or* $V_n = H$.

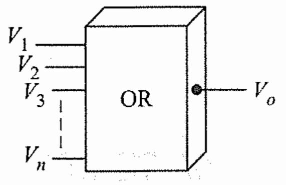


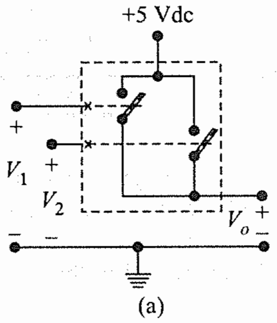
Fig. 1.14 **OR gate**

This last statement leads to the designation OR gate, since $V_o = H$ only if V_1 *or* V_2, *or* V_3, … $V_n = H$.

A model for an OR gate having 2 inputs is shown in Fig. 1.15a. This gate can be used to make "logical" decisions; for example, "If V_1 or V_2, then V_o." As a result, it is referred to as a digital logic circuit, as are all OR gates. From the model, it is seen that $V_1 = H$ closes the upper switch, and $V_2 = H$ closes the lower switch. Clearly, $V_o = H$ if either V_1 or V_2 is high. This can be expressed in the form of a "logic" equation written as

$$V_o = V_1 \text{ OR } V_2$$

The operation is summarized in the truth table in Fig. 1.15b. The symbol for a 2-input OR gate is shown in Fig. 1.15c. Thus, OR is a logic operation which is realized here through a logic gate.



V_1	V_2	V_o
L	L	L
H	L	H
L	H	H
H	H	H

(b)

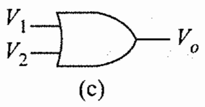


Fig. 1.15 **Two-input OR gate: (a) Model, (b) Truth table, (c) Symbol**

SELF-TEST

9. Refer only to the tri-state buffer symbol in Fig. 1.9c and determine the State of V_o if both G and V_i are low. Check your response with the truth table in Fig. 1.9b. Repeat if both G and V_i are high.
10. Refer only to the inverting tri-state buffer symbol in Fig. 1.11c and determine the state of V_o if both G and V_i are low. Check your response with the truth table in Fig. 1.11b. Repeat if both G and V_i are high.

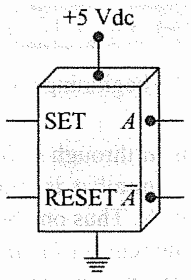
11. For the AND gate in Fig. 1.13c, $V_1 = H$ and $V_2 = L$. What is the state of V_o?
12. If the AND gate in Fig. 1.13c had an additional input (V_3), and $V_1 = V_2 = H$, and $V_3 = L$, what would be the state of V_o? What would it take to produce $V_o = H$?
13. If the OR gate in Fig. 1.15c had an additional, input (V_3), and $V_1 = V_2 = H$, and $V_3 = L$, what would be the state of V_o? What would it take to produce $V_o = L$?

1.4 MOVING AND STORING DIGITAL INFORMATION

Memory Elements

A *digital memory element* is a device or perhaps a circuit that will maintain a desired logic level at its output till it is changed by changing the input condition. The simplest memory element is the switch shown in Figs. 1.7a and b. The switch in Fig. 1.7a is placed such that its output is low, and it will remain low without any further action. Thus, it will "remember" that $V_o = L$. Since $L = 0 = 0$ Vdc, the switch can be thought of as "holding" or "storing" a logic 0. In Fig. 1.7b, $V_o = H$, and it will remain high without any further action. The switch remembers that $V_o = H$. In this case, the switch is holding or storing a logic 1, since $H = 1 = +5$ Vdc. It is easy to see that this switch can be used to store a digital level, and it will remember the stored level indefinitely.

The simplest electronic circuit used as a memory element is called a *flip-flop*. Since a flip-flop is constructed using transistors, its operation depends upon dc supply voltage(s) as seen in Fig. 1.16a. The flip-flop can be used to store a logic level (high or low), and it will retain a stored level indefinitely provided the dc supply voltage is maintained. An interruption in the dc supply voltage will result in loss of the stored logic level. When power is first applied to a flip-flop (turning the system on first thing in the morning), it will store either a high or a low. This is a "random" result, and it must be accounted for in any digital system. Generally, a signal such as MASTER RESET or power-on reset will be used to *initialize* all storage elements.



(a)

SET	RESET	A
H	L	H
L	H	L
L	L	No change
H	H	???(not allowed)

(b)

Fig. 1.16 (a) A flip-flop, (b) Truth table

The truth table in Fig. 1.16b can be used to explain the operation of this flip-flop. The two inputs are SET and RESET, and the output is A. The output labeled $\bar{A}$ is simply the *inverse* of A. Here's how it works:

1. When SET = H and RESET = L, the flip-flop is *set*, and $A = H$.
2. When SET = L and RESET = H, the flip-flop is reset, and $A = L$.

3. Holding SET = *L* and RESET = *L* disables the flip-flop and its output remains unchanged.
4. Applying SET = *H* and RESET = *H* at the same time is not allowed, since this is a request to *set* and *reset* at the same time—an impossible request!

To summarize, when the flip-flop is SET, it stores a high (a logic 1). When it is RESET, it stores a low (a logic 0). A simple flip-flop such as this is often called a *latch*, since its operation is similar to a switch. A 7475 is an IC in the TTL family that contains four similar flip-flops.

Registers

A group of flip-flops can be connected together to store more than a single logic level. For instance, the four flip-flops in Fig. 1.17 can be used to store four logic levels. As such, they could be used to store any of the ten binary numbers given in Table 1.1. As an example, if *A* is SET, *B* is RESET, *C* is SET, and *D* is RESET, this will store the binary number *DCBA* = *LHLH* = 0101, which is equivalent to decimal 5.

When we speak of decimal numbers, each position in a number is called a *decimal digit*, or simply a digit. For example, the decimal number 847 has three digits. When we speak of binary numbers, each position in the number is called a *binary digit*, or *bit*. (The term "binary digit" has been shortened to "bit.") For example, the binary number 0101 is composed of four bits; it is a 4-bit binary number. The four flip-flops in Fig. 1.17 can be used to store any 4-bit binary number.

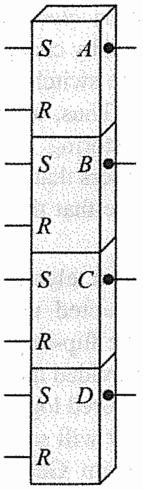


Fig. 1.17 A four-bit register

A group of flip-flops used to store a binary number is called a *register*, or sometimes a *storage register*. The register in Fig. 1.17 is a 4-bit register. There are eight flip-flops in an 8-bit register, and so on. In the TTL family, the 74198 is an 8-bit register. Clearly a register can be used to store decimal numbers in their binary equivalent form. In general, binary numbers such as this are referred to as *data*. A register is a fundamental building block in a microprocessor or digital computer, and you can now see the beginnings of how these systems are used for computation.

The register in Fig. 1.18a has 8 inputs, 1 through 8, and 8 outputs, *a* through *h*. It is constructed using eight flip-flops and some additional electronic circuits. A binary number is stored in this register by applying the appropriate level (high or low) at each input *simultaneously*. Thus one bit is "shifted" into each flip-flop in the register. The binary number is said to be shifted into the register *in parallel*, since all bits are entered at the same time. In this case, the binary number (or data) is entered in one single operation. Once a number is stored in this register, it appears immediately at the 8 outputs, *a* through *h*. A 74198 is an example of an 8-bit *parallel register*.

The register in Fig. 1.18b has a single input and a single output. It is also constructed using eight flip-flops and some additional electronic circuits. It will store an 8-bit binary number, but the number must be entered into the register one bit at a time at the input. It thus requires eight operations to store an 8-bit number. This is how it is done. The first bit of the binary number is entered in flip-flop *A* at the input. The second bit is then entered into flip-flop *A*, and at the same time the first bit in flip-flop *A* is passed along (shifted) to flip-flop

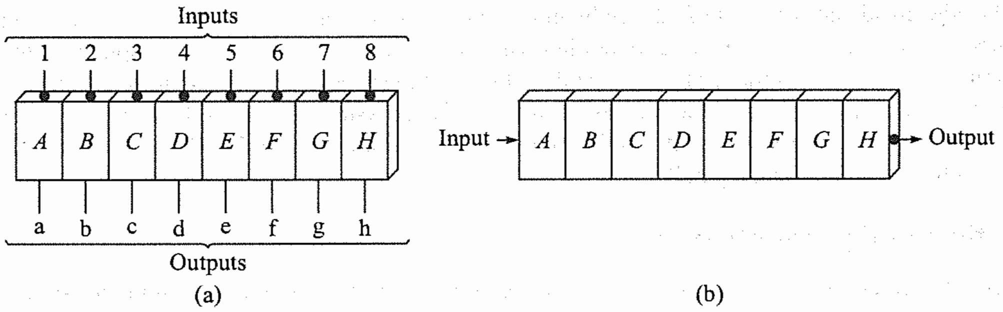


Fig. 1.18 **(a) An 8-bit parallel register, (b) An 8-bit serial register**

B. When the third bit enters *A*, the bit in *A* goes to *B* and the bit in *B* goes to *C*. This shift right process is repeated, and after eight operations, the 8-bit number will be stored in the register. Since the bits are entered one after the other in a serial fashion, this is called a *serial register*. For a stored number to be extracted from this register, the bits must he shifted through the flip-flops from left to right. The stored number will then appear at the output, one bit at a time. It requires eight operations, or eight right shifts, to extract the stored number. A 74164 TTL is an example of an 8-bit serial register (this particular IC also provides parallel outputs).

Transferring Digital Data

A register is used to enter data (binary numbers) into a microprocessor or computer. A register is also used to extract data from a computer and direct it to an external destination. Wire cables are generally the means for connecting systems. If a parallel register is used, the data is said to be shifted in parallel. The connector in this case must have one pin for each bit, and the cable must have at least one wire for each bit. An 8-bit register requires a cable having at least 8 wires, a 16-bit register must have at least 16 wires, and so on.

Data are also transferred (shifted) between registers within a digital system. Instead of drawing all 8 (or 16 or 32) wires on a schematic, it is common practice to use an arrow between the registers, as illustrated in Fig. 1.19a. The number 8 in parentheses means that there are eight wires. In this case, there are eight connections used to transfer 8 bits of data in parallel from register *A* to register *B*. The eight wires represented by this arrow are called a *data bus*. The double arrow shown in Fig. 1.19b means 16 bits of data can be shifted in parallel from *A* to *B* or from *B* to *A*. This is a 16-bit *bidirectional data bus*.

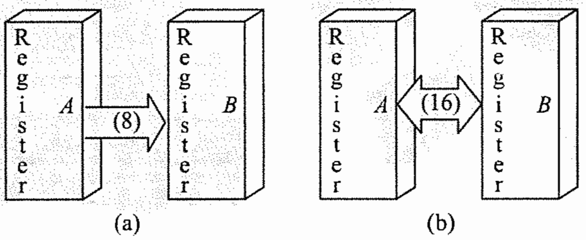


Fig. 1.19 **(a) An 8-bit data bus, (b) A 16-bit, bidirectional data bus**

On the other hand, data can be shifted serially into or out of a serial register, and only one wire (connection) is required for the data. Clearly, parallel operation will transfer data into or out of a computer system much more rapidly than serial operation. The price paid for this gain in speed is an increase in complexity, in terms of both the electronic circuits and the increased number of connections (wires in the cable). The computer connector where data is entered or extracted is frequently called a *port*. Nearly all computer systems have available both a serial port and a parallel port.

Magnetic and Optical Memory

Any memory element must be capable of storing or retaining only two logic levels, and there are numerous devices with the appropriate electronic circuits used for this purpose. One of the most common systems for memory makes use of the fact that a magnetic material can be magnetized with two different orientations. Thus, magnetizing spots on a strip of magnetic tape, or on a hard disk with a magnetic coating, or on a magnetic floppy disk are well known and widely used memory systems. In optical memory data is encoded in binary by making two different kinds of reflecting surface on a spiral track of a circular disk. A special pointed source of light falls on the surface and intensity of reflected light gives information about the data stored. A number of devices that utilize magnetic and optical storage are illustrated in Fig. 1.20.

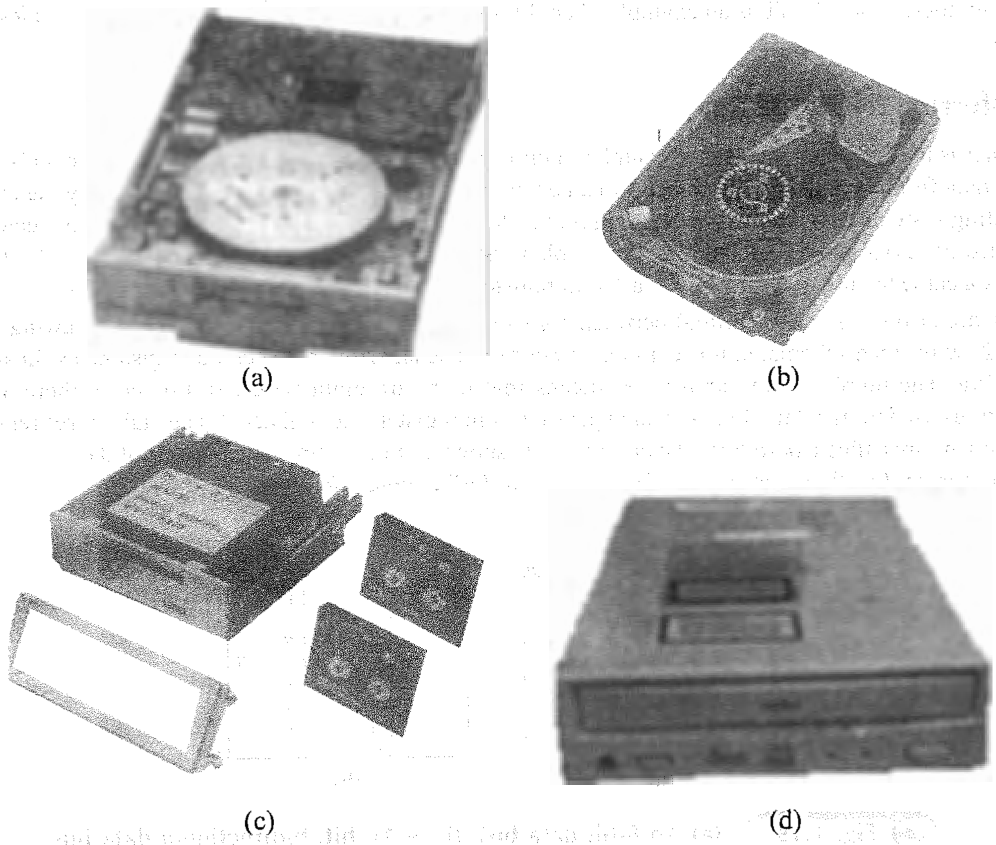


Fig. 1.20 **(a) A 3½-inch magnetic floppy disk drive, (b) Magnetic hard disk drive, (c) Magnetic tape drive, (d) An optical disk drive**

SELF-TEST

14. Can you think of different ways to construct a digital memory element beginning with "opposite" terms such as on–off, in–out, up–down, right–left, cold–hot, wet–dry, etc.?
15. What logic level will appear at A if the flip-flop in Fig. 1.16 has SET = L and RESET = H?
16. Look at the binary representation of the decimal number 9 in Table 1.1. How many bits are there in this binary number? If it is stored in the register in Fig. 1.17, what are the bit values of $DCBA$?
17. If a shift operation requires a time of 1 μs to complete, how long would it take to enter an 8-bit number into the parallel register in Fig. 1.18a? How long would it take to enter an 8-bit number into the serial register in Fig. 1.18b?
18. When we speak of a microprocessor, what is meant by the term *port*?

1.5 DIGITAL OPERATIONS

Counters

It was mentioned previously that counting is an operation easily performed by a digital circuit. A digital circuit designed to keep track of a number of events, or to count, is called a *counter.* The counter in Fig. 1.21a is constructed using a number of flip-flops (n) and additional electronic circuits. It is similar to a storage register, since it is capable of storing a binary number. The input to this counter is the rectangular waveform labeled *clock*. Each time the clock signal changes state from low to high, the counter will add one (1) to the number stored in its flip-flops. In other words, this counter will count the number of clock transitions from low to high. A clock having a small circle (bubble) in the input side would count clock transitions from high to low. This is the concept of *active low*—that is, an action occurs when the input is low.

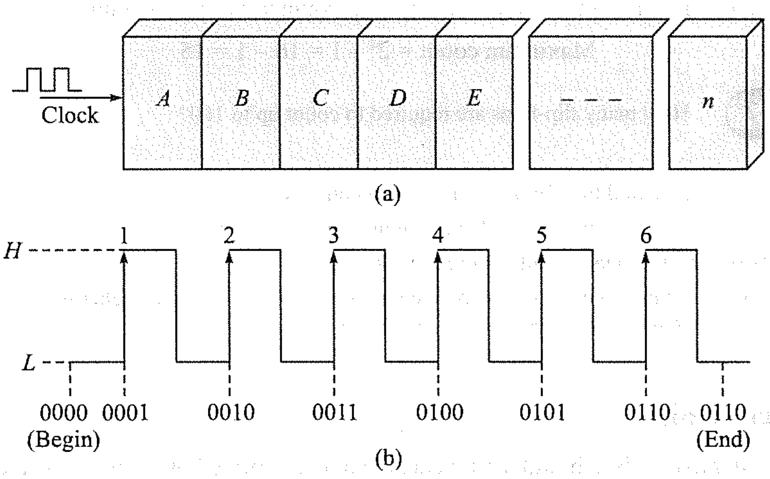


Fig. 1.21 (a) A counter constructed with n flip-flops, (b) A count of 6

As an example of how this circuit might be used, suppose that the counter consists of four flip-flops, all of which are RESET. That is, the binary number stored in the counter is 0000. The clock signal is initially held low. Now the clock is allowed to "run" for six clock periods, and then it is held low, as shown in Fig. 1.21b. After the first clock transition from low to high, the counter will advance to 0001. After the second transition, it will advance to 0010, and so on, until it will store the binary number 0110 after the sixth transition. The binary number 0110 is equal to decimal 6, and thus the counter has counted and stored the six clock transitions! The waveform in Fig. 1.21b shows the clock with the counter contents directly beneath each transition.

A four-flip-flop counter can count decimal numbers from 0 to 15. To count higher, it is necessary to add more flip-flops. It is easy to determine the maximum decimal count in terms of the number of flip-flops using the following relation

$$\text{Maximum count} = 2^n - 1 \tag{1.1}$$

where n = number of flip-flops.

The term 2^n means 2 raised to the nth power, that is, 2 multiplied by itself n times. For example,

$$\begin{aligned}
2^2 &= 2 \times 2 = 4 \\
2^3 &= 2 \times 2 \times 2 = 8 \\
2^4 &= 2 \times 2 \times 2 \times 2 = 16 \\
2^5 &= 2 \times 2 \times 2 \times 2 \times 2 = 32 \\
2^6 &= 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64 \\
2^7 &= 128 \\
2^8 &= 256 \\
2^9 &= 512 \\
2^{10} &= 1024
\end{aligned}$$

We'll spend more time on binary and decimal numbers in Chapter 5. For now, this listing of powers of 2 can be used with Eq. (1.1). For example, the four-flip-flop counter has a maximum decimal count of

$$\text{Maximum count} = 2^4 - 1 = 16 - 1 = 15$$

Example 1.2 How many flip-flops are required to count up to 100?

Solution From Eq. (1.1), we find that for $n = 6$, maximum count is $2^6 - 1 = 63$

and for $n = 7$, maximum count is $2^7 - 1 = 127$

Thus, 100 lying in between, the count would require 6 flip-flops.

The other way of solving this problem is to round the number $\log_2(N + 1)$ to next higher integer when a count of N is desired. Since, $\log_2 101 = 6.6582$, the number of flip-flops required is 7.

Arithmetic Logic Unit

An *arithmetic logic unit* (*ALU*) is a digital circuit capable of performing both *arithmetic* and *logic* operations. The basic arithmetic operations performed by an ALU are addition (+) and subtraction (–). Multiplication (×) and division (÷) of digital numbers are accomplished with other digital circuits. Logic operations will usually include inversion (NOT), AND, and OR. The ALU represented in Fig. 1.22 has two data inputs, the

A bus and the *B* bus, and the *F* bus is the resultant output. The digital levels on the *S* bus determine which operation is to be performed. Generally, each of the data buses will have the same number of bits. As an example, in the TTL family, the 74181 is an ALU similar to Fig. 1.22. Both the *A* and *B* inputs are 4-bit buses, and the output at *F* is also a 4-bit bus. For this particular circuit, the control signal at the *S* bus is also four bits. In addition, the 74181 has a number of other inputs and outputs which we will discuss in detail later.

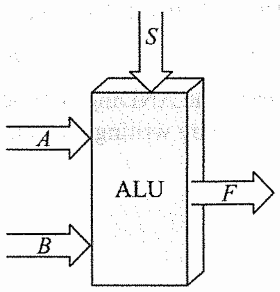


Fig. 1.22 An arithmetic logic unit (ALU)

Addition and Subtraction If the proper digital levels are applied to the inputs of the *S* bus, the ALU in Fig. 1.23 can be used to add two digital numbers. The two numbers to be added are represented by the proper logic levels at *A* and *B*, and the SUM of these two numbers will appear at output *F*. In the event the sum of the two numbers generates a carry, an *H* will appear at the CARRY OUT. To illustrate, suppose we wish to perform the addition 6 + 7 = 13. Here's how we might do it with decimal numbers.

```
(carry)←┐
        | 6
        | 7(+1)
   ──────────
      1   3
```

The digital levels illustrated in Fig. 1.23 will result in the addition of these two numbers. The equivalent decimal numbers are shown in parentheses. Notice that the CARRY IN allows this ALU to add two numbers, plus a carry.

By changing the control levels at the *S* bus, this ALU will determine the difference *A* – *B* (subtraction). In this case, the digital levels at the *F* bus represent the DIFFERENCE, rather than the SUM.

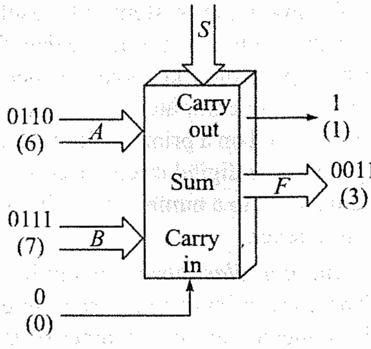


Fig. 1.23 An ALU used for addition

Logic Functions By changing the digital levels at the *S* bus, the ALU in Fig. 1.22 can be used to perform a number of different logic functions relative to the two digital inputs. The desired function appears at the *F* bus. Here are some of the possibilities:

$$F = \overline{A}$$

$$F = \overline{B}$$

$$F = A \text{ AND } B$$

$$F = A \text{ OR } B$$

The operations are carried out "bit by bit." For example,

If $A = 1010$ then $F = \overline{A} = 0101$

If $A = 1010$ and if $B = 0110$, then

$$F = A \text{ AND } B = 1010 \text{ AND } 0110 = 0010$$

In this case, the ANDing is done on the corresponding bit of each input. There are tour AND operations. It's easier to see by writing the data as follows:

$$\begin{aligned} A &= 1010 \\ &\;\;\; |||| \\ B &= 0110 \\ F &= 0010 \end{aligned}$$

The "vertical lines" between A and B show which bits are ANDed.

Comparison Comparing the magnitude of two numbers is an important logical operation. The circuit in Fig. 1.24 is a *comparator*. It is capable of comparing two digital numbers and indicating whether the magnitude of one is greater than, less than, or equal to the other. For example, if $A = 0110$ (decimal 6) and $B = 0111$ (decimal 7), then the output $A < B$ will be high. The other two outputs will be low. A 7485 in the TTL family is a 4-bit comparator similar to Fig. 1.24. Also, the 74181 ALU can be used with the same results.

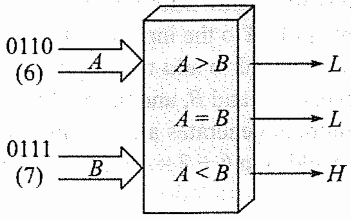


Fig. 1.24 A comparator

Input/Output

In order for any digital system to be useful, there must be some provision for entering data into the system and also some method of extracting data from the system. In the case of a computer, information is frequently entered by typing on a keyboard or perhaps by using a magnetic floppy disk. Useful information can be obtained from the computer by examining the visual displays on a cathode-ray tube (CRT) or by reading material produced on a printer. Clearly there is a requirement to connect multiple input devices, one at a time, to the system. The digital circuit used for this operation is a multiplexer. Likewise, there is a need to connect the system output to a number of different destinations, one at a time. The digital circuit used for this purpose is a demultiplexer.

The term *multiplex* means "many into one." A *multiplexer* (*MUX*) can be represented as shown in Fig. 1.25a. There are n input lines. Each line is used to shift digital data serially. There is a single output line which is connected to the computer (system) input port. Operation of the circuit can be explained by using the "switch" as a model. Each setting of the digital control levels on the C bus will connect the switch to one of the input lines. Data from that particular input is then entered into the computer. Changing the C bus levels will connect a different input. Thus, data from multiple sources can be connected to a single input port, one at a time. An example of a MUX is the 74150 in the TTL family. It has 16 input lines and a single output line.

The opposite of multiplex is *demultiplex*, which means "one into many." A *demultiplexer* (*DEMUX*) can be represented as in Fig. 1.25b. This digital circuit simply connects the single data input line to one of the n output lines, one at a time, according to the levels on the C bus. Thus serial data from the computer output port can be directed to different destinations, one at a time. An example of a DEMUX is the TTL 74154, which can be used to connect a single input to any one of 16 outputs, one at a time.

Any information entered into a digital system must be in the form of a digital number. A circuit that changes data into the required digital form is called an *encoder*. The encoder shown in Fig. 1.26a on the next

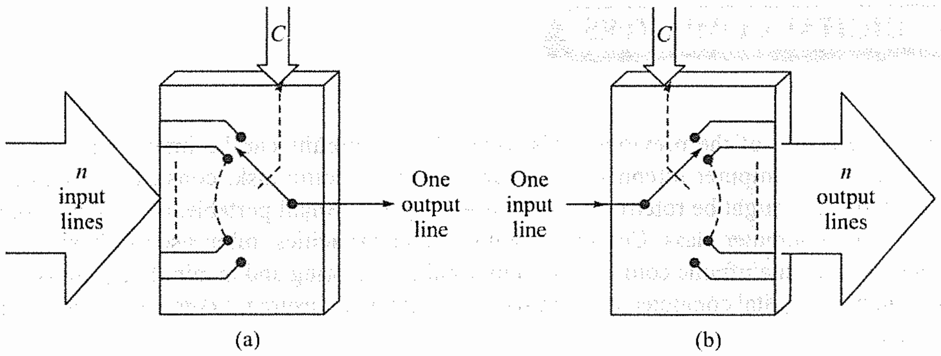


Fig. 1.25 (a) A multiplexer (MUX), (b) A demultiplexer (DEMUX)

page will change a decimal number into its binary equivalent. It may be used with a keyboard. For instance, depressing the number 4 key on a keyboard will cause input line 4 to this encoder to be high (the other inputs are all low). The result will be decimal 4, binary 0100, at the encoder output as shown.

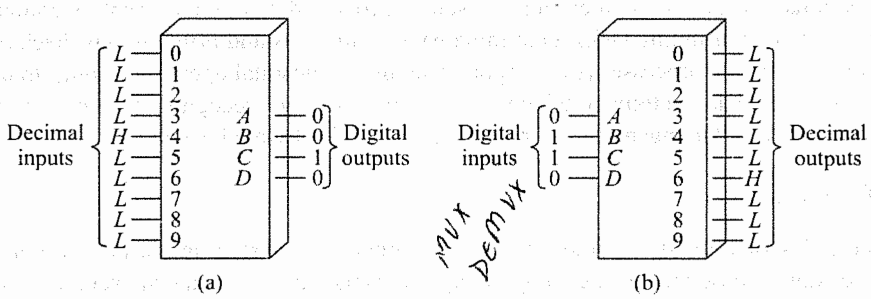


Fig. 1.26 (a) An encoder, (b) A decoder

Taking digital information from the output of a computer and changing it into another form is accomplished with a *decoder*, for example, changing the digital number 0110 (decimal 6) into its decimal form. The decoder in Fig. 1.26b will accept a 4-bit binary number and indicate its decimal equivalent between 0 (zero) and 9. As shown, the binary input 0110 will cause output line 6 to be high, while all other output lines remain low. There are many different types of encoders and decoders. A number of them will be discussed in detail in Chapter 4.

SELF-TEST

19. What binary number will be stored in the counter in Fig. 1.21a if the clock is allowed to run for seven periods?
20. How many flip-flops are required to construct a digital counter capable of counting 1000 events?
21. State whether or not the ALU in Fig. 1.22 will generate a carry out if the numbers added are: (a) 2 and 3; (b) 5 and 5; (c) 9 and 9.
22. What are the digital output levels of the encoder in Fig. 1.26a if only input line 6 is high?

1.6 DIGITAL COMPUTERS

Terms

An appropriate selection of the previously discussed digital circuits can be interconnected to construct a digital computer. A computer intended to perform a very specific task, constructed with a minimum number of components, might be referred to as a *microcomputer*. Small portable, or desktop, computers are usually in the microcomputer class. Computers with greater capacities, often used in business, are called *minicomputers*. A large mainframe computer system capable of storing and manipulating massive quantities of data, for example, a digital computer system used by a bank or an insurance company, might then be called a *maxicomputer*.

Uses

What can a digital computer be used for? Numerical computation is surely one possible use. The inclusion of an ALU with additional logic circuits provides arithmetic capabilities (addition, subtraction, multiplication, division). The logic portion of the ALU means the computer can be used to make logical decisions. Beyond these basic functions, a digital computer can be used to process data (balance bank accounts), to rapidly perform otherwise time-consuming tasks (determine payroll amounts and print out paychecks), to precisely monitor and control intricate processes (life support systems in a hospital operating room), to use speech for communication with humans (automatic telephone systems and voice recognition)—the list is almost endless, and is limited only by the ingenuity and resourcefulness of individual users!

Basic Configurations

A microcomputer designed to control a given machine, process, or system might be represented as in Fig. 1.27. The control signals produced by the computer appear as the output bus and are sent to an output device. Here, the signals are properly conditioned and sent to the mechanism being controlled. The controlled entity must then send signals indicating its present condition back to the computer via an input device and via the input bus. The computer analyzes these present condition signals, determines any necessary action, and sends required correction signals out to the system.

A microcomputer system might be designed to irrigate the lawn area of a park. Watering is to be done only at night, when the soil moisture falls below a given value. The system "sensors" in this case would be (1) a probe to detect soil moisture and (2) a light detector to distinguish between daylight and darkness. The computer would monitor signals from the two sensors, turn on the watering system whenever the soil moisture fell below a set value (but only at night), and turn the system off as the moisture increased above the desired value.

The minicomputer illustrated in Fig. 1.28 on the next page is more complicated than the microcomputer described, but it has greatly increased utility. The input bus is serviced with a MUX.

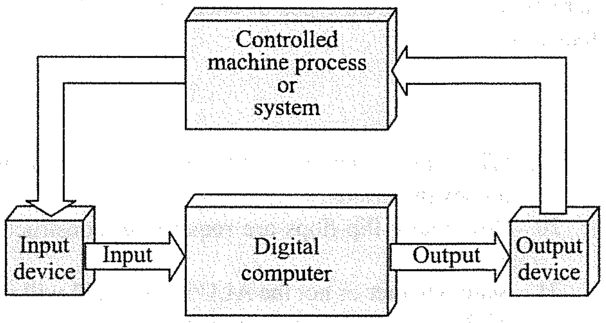


Fig. 1.27 A digital computer based system

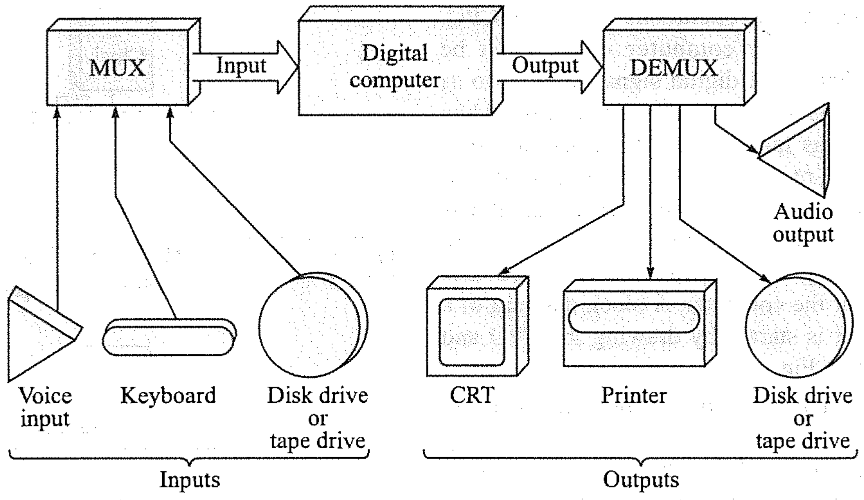


Fig. 1.28 **Block diagram of a computer system**

This allows the connection of a number of different input devices:

A keyboard for typewritten entry of alphanumeric information

A disk drive or tape drive for entering data stored in magnetic form

A microphone for voice input

The DEMUX on the output bus allows numerous possibilities for receiving information from the computer:

The familiar CRT for a visual display

A printer to provide printed material (called *hard copy*)

A disk or tape drive to record data in magnetic form

Perhaps a speaker for audio information

A minicomputer such as this can be used for many different tasks. It can be used as a word processor, for data processing, for communication via telephone (both voice and fax), for training in an educational setting, for computer games, and so on!

The block diagram in Fig. 1.28 forms the basis for larger computer systems. A larger system will have a much greater capacity to store data and will in general utilize numerous input/output units. For instance, a maxicomputer will likely have more than one printer, and perhaps even different types of printers. It will generally have a large number of users, all of whom desire access to the system at the same time. One workstation must then be provided for each user. A keyboard and a CRT are the minimum components required at each workstation. The digital circuits used to construct maxicomputer systems are necessarily more complicated than minicomputer systems, and they may operate at a much faster rate. Let's take a look inside a typical digital computer.

Basic Computer Architecture

The *central processing unit* (*CPU*) is the brain of a digital computer. It is constructed using an ALU along with a number of registers and counters. The CPU is therefore the primary center for computation and decision making. All the operations within the CPU, and indeed within the computer itself, must be carefully coordinated. A digital signal referred to as the *system clock* is used as a reference to time when specific operations take place. The clock signal is usually a periodic, rectangular waveform as illustrated in Fig. 1.6. Using a crystal in the clock circuit allows the accuracy and stability of the clock frequency, to be controlled with great precision. The clock provides a "heartbeat" for the computer. A block diagram of a digital computer is started by drawing the CPU and clock as shown in Fig. 1.29.

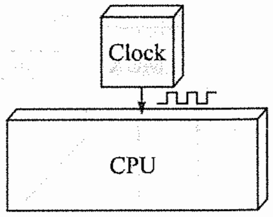


Fig. 1.29 **The "heart" and "brain" of a digital computer**

The CPU is capable of computation and decision, but it must have specific instructions telling it exactly *what* to do and *when* to do it. This set of instructions is called a *program*. A program is a detailed list of operations written by a human programmer. The programmer decides what the computer is to do and when it should be done, and then writes a list of instructions to be carried out in the proper order. The program is entered into the computer, using perhaps a keyboard, and stored in the computer *memory*. The CPU can then "fetch" from memory one instruction at a time, in the given order. It will execute the instruction and then fetch the next instruction. With this repeated fetch-and-execute cycle, the CPU will accomplish the desired task. A memory block used for program storage has been added in Fig. 1.30.

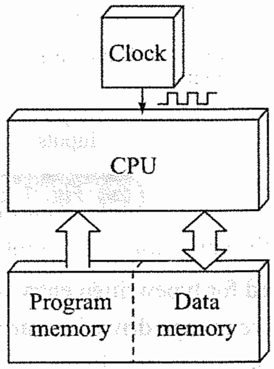
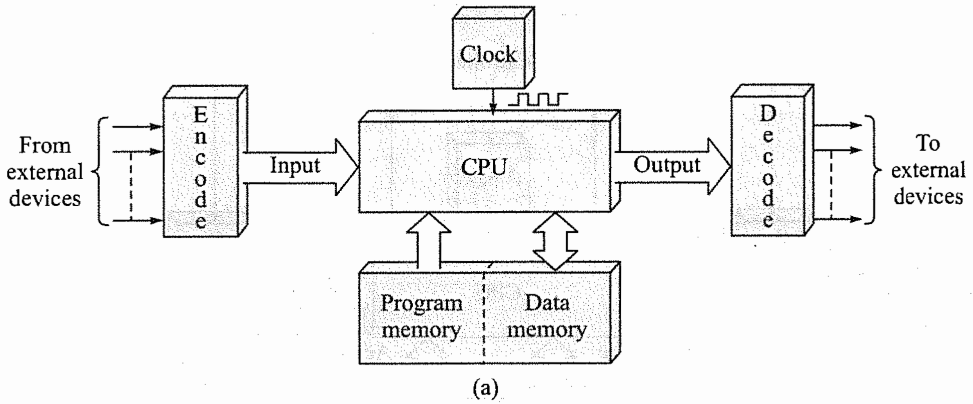


Fig. 1.30 **CPU with memory block**

A portion of the memory block in Fig. 1.30 is labeled *data*. This is the area where the information being processed by the computer is stored. In addition, this is where the CPU stores the results of computations and/or decisions made. Since the CPU takes ("reads") data from memory, as well as returns ("writes") data into memory, the memory data bus is bidirectional. By contrast, the program data bus is not bidirectional, since information on this bus is always from memory to CPU.

The CPU communicates with the "outside world" by means of the input encoders and the output decoders. The ability to multiplex inputs and demultiplex outputs may also be included in the input/output blocks that have been added in Fig. 1.31a. This configuration is sometimes quite inefficient, since all information entering or exiting the computer must pass through the CPU. The CPU operates at a much faster rate than most external devices, and it must *wait* while data are being entered or exited. A *direct memory access* (*DMA*) block is generally included to alleviate this problem. As seen in Fig. 1.31b, the DMA allows information to move directly from an input device into memory, or from memory directly to an output device. While information is being transferred via the DMA, the CPU is free to carry on its computational or logical operations. This greatly improves system efficiency as well as speed of operation.

Before data can be entered into the computer, a signal on the input request line asks the computer for "permission" to input information. For instance, depressing the enter key on a keyboard will generate an input request signal. When the CPU is ready, a signal is generated on the acknowledge line, and data will be entered



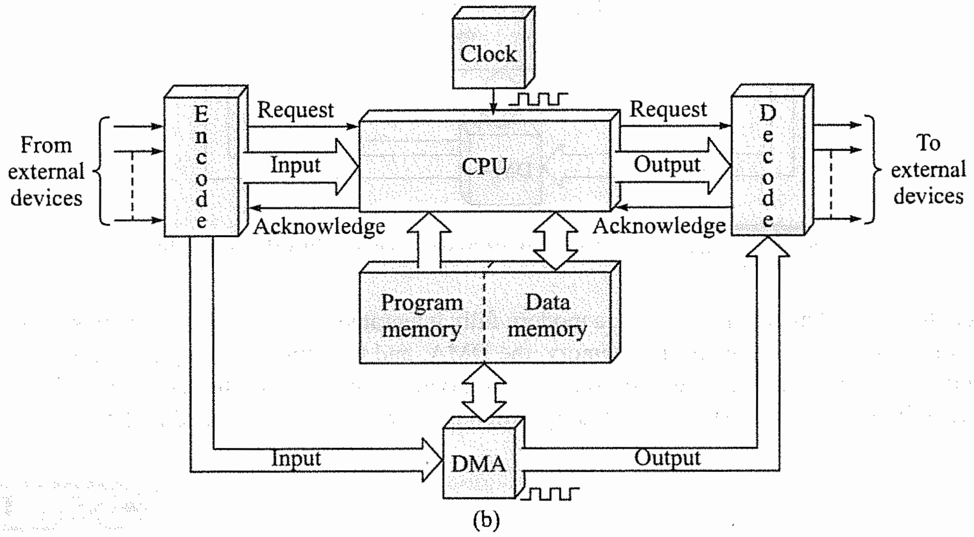


Fig. 1.31 **CPU, memory with input/output block**

via the DMA into the memory. This request-acknowledge sequence is often called *handshaking*. A similar handshaking must occur when the CPU is ready to deliver data to an external device. However, in this case, the CPU makes an output request, and the external device gives permission.

All of these blocks are operated in synchronism with the clock, but additional direction must be provided. The *controller* is the unit that decides which block "goes first" (establishes priorities), decides the order in which external devices are serviced, routes data along the various buses such that no conflicts occur, and controls the overall operation of the system. As such, the controller communicates individually with each block as illustrated in Fig. 1.32. This block diagram is representative of the architecture of many digital computers.

A *microprocessor* is often used as the basic IC around which a microcomputer or minicomputer is constructed. Numerous computers have been designed, beginning with the 8080 microprocessor. Improvements to this basic IC have led to the development of a family of microprocessor units including the 8085 and the 8086.

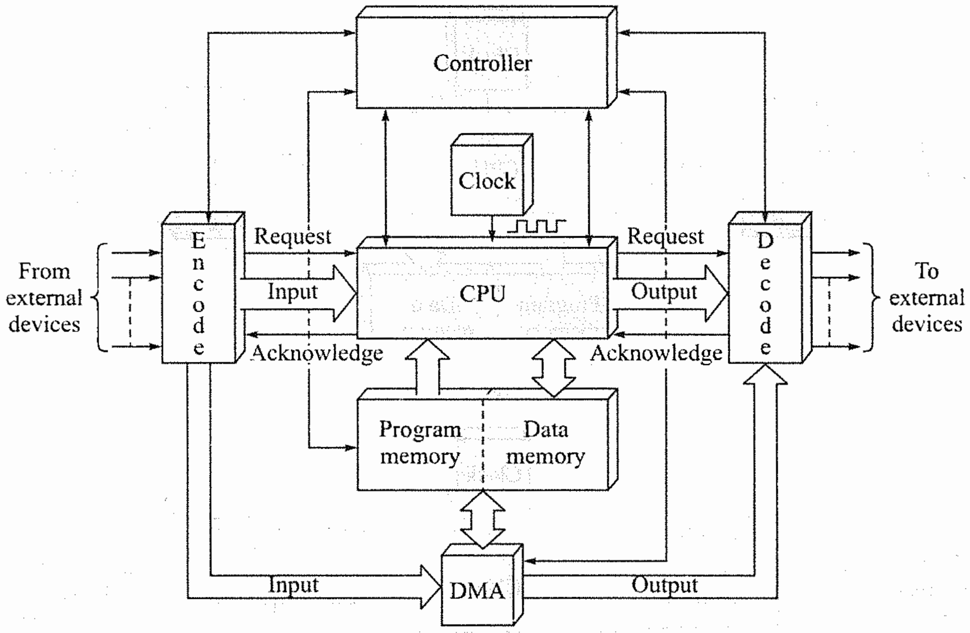


Fig. 1.32 A microprocessor-based digital computer

Referring to the block diagram in Fig. 1.32, a modern fully integrated microprocessor contains the controller, the clock, the CPU, and portions of the memory, the DMA, and the input and output blocks. The purpose of this section is to provide a general understanding of a digital computer. In the remainder of this text, the blocks used to build any digital system, including a digital computer, will be studied in detail.

SELF-TEST

23. Why is the system clock considered the heart of a digital computer system?
24. What is a computer program?
25. What functions are carried out in an ALU?
26. What is the purpose of the DMA in Fig. 1.32?

1.7 DIGITAL INTEGRATED CIRCUITS

A digital IC is constructed by an interconnection of resistors, transistors, and perhaps small capacitors, all of which have been formed on the surface of a semiconductor wafer. The entire circuit resides on a tiny piece of semiconductor material called a *chip*.

The semiconductor wafer is typically a slice of monocrystalline silicon about 0.2 mm thick and perhaps 8 to 15 cm in diameter, as illustrated in Fig. 1.33a. The wafer is divided checkerboard fashion into 1000 or so rectangular areas. Each area will become a single chip. The resistors and transistors necessary for each digital

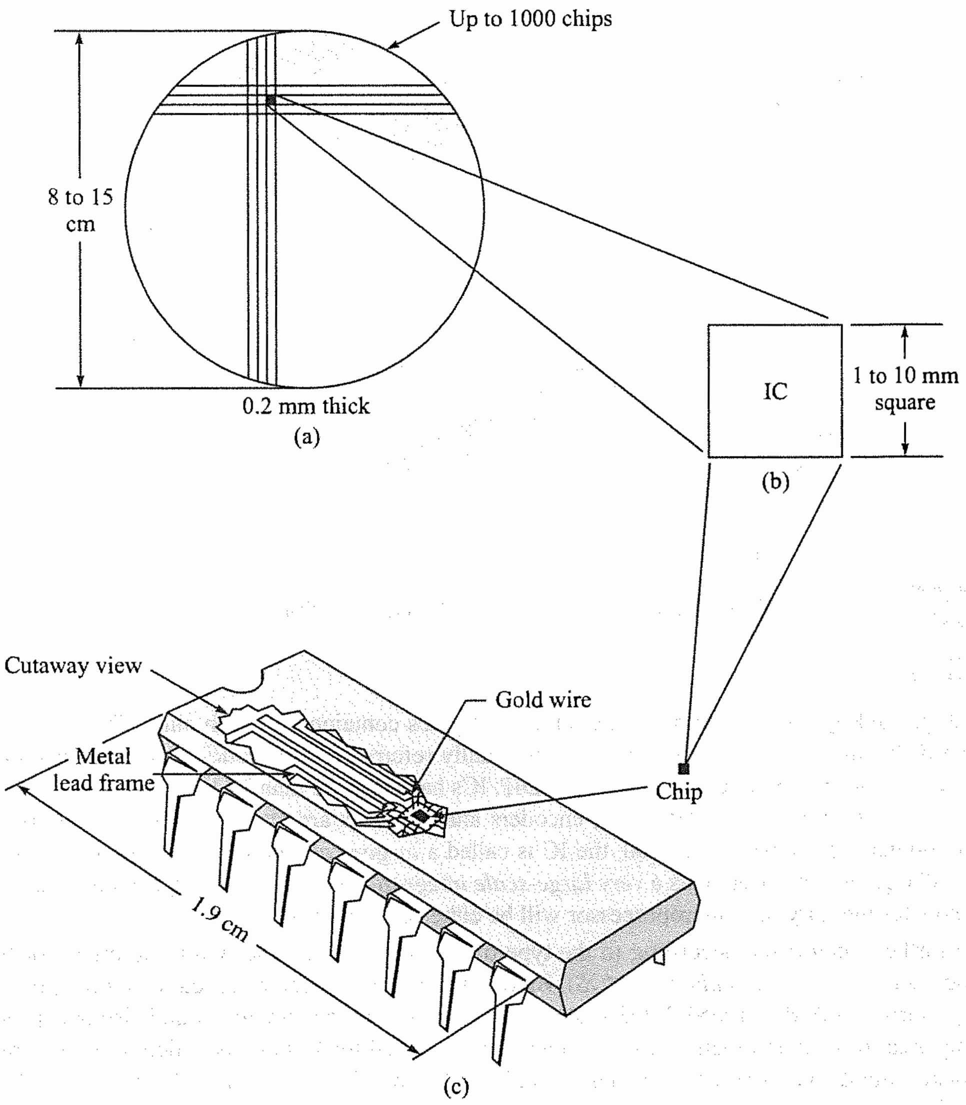


Fig. 1.33 (a) A wafer, (b) One chip, (c) Dual-inline package (DIP)

circuit are then formed on each chip by a series of semiconductor processing steps. In this fashion, identical digital circuits are manufactured simultaneously on the same silicon wafer.

After the processing steps are completed, the wafer is separated into individual chips as seen in Fig. 1.33b. Each chip is a digital circuit, for example, an inverter or an AND gate. An individual digital circuit may have only a few components, but some circuits have a few hundred components! Each chip is then mounted in a suitable package, as shown in Fig. 1.33c. The package illustrated here is a 14-pin *dual-inline package* (*DIP*). Some additional packages for ICs are shown in Fig. 1.34.

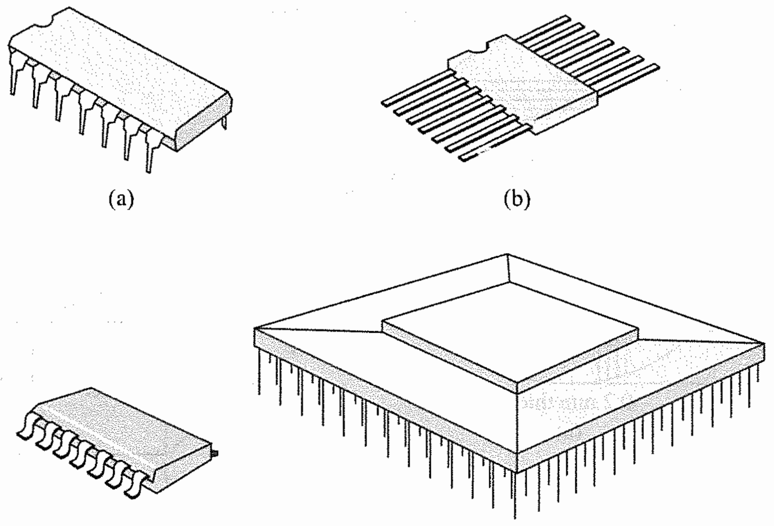


Fig. 1.34 Some IC packages: (a) DIP, (b) Flat pack, (c) Surface mount, (d) Pin-grid array

IC Families

ICs are categorized by size according to the number of gates contained on each chip. There is no absolute rule, but an IC having fewer than 10 or 12 gates is usually referred to as a *small-scale integration* (*SSI*) *IC*. For instance, a 7404 has six inverters in a 14-pin DIP. ICs having more than 12 but fewer than 100 gates are called *medium-scale integration* (*MSI*) *ICs*; encoders and decoders are examples of MSI ICs. If there are more than 100 gates but fewer than 1000, the IC is called a *large-scale integration* (*LSI*) *IC*. An IC having more than 1000 gates is referred to as a *very large-scale integration* (*VLSI*) *IC*. A large complex system such as semiconductor memory or a microprocessor will be either LSI or VLSI.

ICs are further categorized according to the type of transistors used. The two basic transistor types are *bipolar* and *metal-oxide-semiconductor* (*MOS*). Bipolar technology is faster but requires more power, and is generally preferred for SSI and MSI. MOS is slower, but requires much less power and also occupies a much smaller chip area for a given function. MOS is therefore preferred for LSI and is widely used in applications such as pocket calculators, wristwatches, hearing aids, and so on. For the moment, let's consider the overall characteristics of each digital IC family.

Bipolar Transistors

There are two important digital circuit families constructed using bipolar transistors:

- **Transistor-Transistor Logic** (TTL)
- **Emitter-Coupled Logic** (ECL)

Transistor-Transistor Logic TTL was first introduced by Texas Instruments in 1964 using the numbers 54XX and 74XX. These two families are now widely available from a number of different manufacturers. The 74XX ICs operate over a temperature range of 0°C to 75°C. The 54XX devices are more rugged; they operate over a temperature range of −55°C to +125°C. As you might expect, the 54XX devices are more expensive. Otherwise, the logical operations of these two families are the same. In each case, the XX portion of the part number refers to a specific device. For instance, "04" stands for inverter, and a 7404 is a TTL inverter. A 7411 is a TTL AND gate, and so on. When there is no danger of confusion, it is common practice to shorten the description by omitting the first two digits. Thus, a 7404 inverter is referred to as a 04, and the 7411 AND gate is designated as 11. Table 1.2 is a partial listing of some widely used 74XX ICs. Note that a 5404 is logically the same as a 7404; it is simply guaranteed to operate over a wider temperature range.

Table 1.2 **Some Standard Digital Circuits**

TTL	*ECL*	*CMOS*	*Function*
7400	—	74HC00	Quad 2-input NAND gate
7402	MC10102	74HC02	Quad 2-input NOR gate
7404		74HC04	Hex inverter
7408	MC10104	74HC08	Quad 2-input AND gate
7432	MC10103	74HC32	Quad 2-input OR gate

In the interest of higher operating speed, the 74XX family was improved with the introduction of the 74HXX (where the H stands for high speed) family of devices. The price paid for increased speed was an increase in power required to operate each gate. This led to another family of devices designed to minimize power requirements—the 74LXX (where the L stands for low power) series.

A major improvement in the TTL series came with the development of a special transistor arrangement called a *Schottky transistor*. Using this device, the 74SXX (S for Schottky) family came into being. These devices greatly improved operating speed, but again at the cost of increased power consumption. At this point, a family of devices designated 74LSXX (low-power Schottky) was developed. The 74LSXX family offers high-speed operation with minimal power consumption and today is preferred in most designs. The original 7400 also remains popular.

There are two additional families, 74ASXX (advanced Schottky) and 74ALSXX (advanced low-power Schottky), available. One might anticipate the development of other families with characteristics to match specific needs. Table 1.3 is a comparison of power consumption and speed of operation (delay time) for some TTL families.

Emitter-Coupled Logic Emitter-coupled logic (ECL) is considerably faster than any of the TTL families, but the power required for each gate is also much higher. With a propagation delay of only 2 ns, the industry standard for ECL circuits is 10,000 ECL, abbreviated 10K. The 100K (100,000) series is even faster, with a delay time of only 1 ns. Motorola markets a family of devices designated MECL 10K and MECL 10KH (Motorola Emitter Coupled Logic). Representative 10K MECL circuits are listed in Table 1.2.

MOS Transistors

There are three digital logic families constructed using MOS field-effect transistors (MOSFETs):

- **PMOS** Using *p*-channel MOSFETs
- **NMOS** Using *n*-channel MOSFETs
- **CMOS** Using both *n*-channel and *p*-channel MOSFETs

PMOS, the slowest and oldest type, is nearly obsolete today. NMOS dominates the LSI field and is widely used in semiconductor memories and microprocessors. CMOS is preferred where individual logic circuits are used and where very low power consumption is required.

Table 1.3 **TTL Power-Delay Values**

Type	*Name*	*Power, mW*	*Delay Time, ns*
74XX	Standard TTL	10	10
74HXX	High-speed TTL	22	6
74LXX	Low-power TTL	1	35
74SXX	Schottky TTL	20	3
74LSXX	Low-power Schottky	2	10

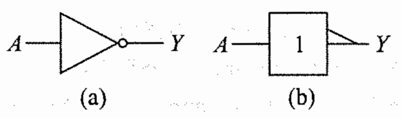
The original 4000 series of MOS devices was introduced by RCA. It was slow, was not compatible with TTL, and is rarely seen in modern designs. The 74CXX (C for CMOS) is a series of digital ICs that are manufactured using MOS technology. These devices are pin-for-pin replacements for similarly numbered 7400 TTL devices. For instance, a 7404 is an IC that contains six inverters, and the 74C04 also contains six inverters. Thus, digital circuits designed using 74XX TTL devices can also be implemented using 74CXX MOS devices. The 74CXX design will require considerably less operating power but will be restricted to lower operating speeds. The 74HCXX (where the HC stands for high-speed CMOS) family of circuits is the most widely used today (see Table 1.2).

Complete digital logic systems can be constructed using entirely 74XX devices or 74HCXX devices, and the two types of devices can even be used together, provided certain precautions are observed. The necessary precautions involve voltage levels, current requirements, and switching times; this comes under the subject of *interfacing*, which will be addressed in Chapter 13. A second family of CMOS devices that is entirely compatible with TTL circuits is the 74HCTXX series (where the H stands for high-speed, the C for CMOS, and the T for TTL-compatible).

Since the mid-1980s, two additional CMOS families have been available—the 74ACXX (advanced CMOS), and the 74ACTXX (advanced CMOS TTL-compatible) families. As with TTL, there will no doubt be further advancements to produce additional CMOS families.

Digital Logic Symbols

The Institute of Electrical and Electronics Engineers (IEEE) along with the American National Standards Institute (ANSI) have developed a new symbolic language and set of symbols to be used with digital logic circuits. These new symbols are now being used on manufacturers' data sheets along with traditional symbols. The most recent revision of *IEEE Standard Graphic Symbols for Logic Functions*, ANSI/IEEE Std 91–1984, provides for two different types of symbols. Symbols of the first type, called *distinctive-shape symbols*, are exactly as have been shown throughout this chapter. The second system, which is called the *rectangular-shape system*, uses a rectangular box with a special symbol for each type of gate. The IEEE standard does not express a preference for either shape. Most people presently involved in digital electronics seem to prefer the distinctive-shape system, and since this type of symbol is still included on data sheets, we will use these symbols in this text. Below is a brief introduction to the new rectangular symbols, which are presented along with their traditional, distinctive-shape counterparts. More detailed information is available in Appendix 7.



A	*Y*
L	*H*
H	*L*

(c)

Fig. 1.35 **(a) Standard symbol, (b) New IEEE symbol, (c) Truth table**

The standard logic symbol for an inverter is shown in Fig. 1.35a, where *Y* is the complement of

A. The new IEEE symbol is shown in part *b* of this figure. A rectangular box is used for the gate, the input is labeled *A*, and the output is labeled *Y*. The 1 inside the box signifies that the input must be active in order to have an active output. The triangle on the output line signifies that the output is active when low. Thus, when the input is active (high), the output will be active (low). The truth table is shown in Fig. 1.35c.

The 7404 is a hex inverter; that is, it is an IC that contains six inverters. The DIP package for this device is shown in Fig. 1.36a, along with the proper pin numbers on the package. Figure 1.36b shows the six standard logic symbols for the inverters. Figure 1.36c shows the new IEEE logic symbol.

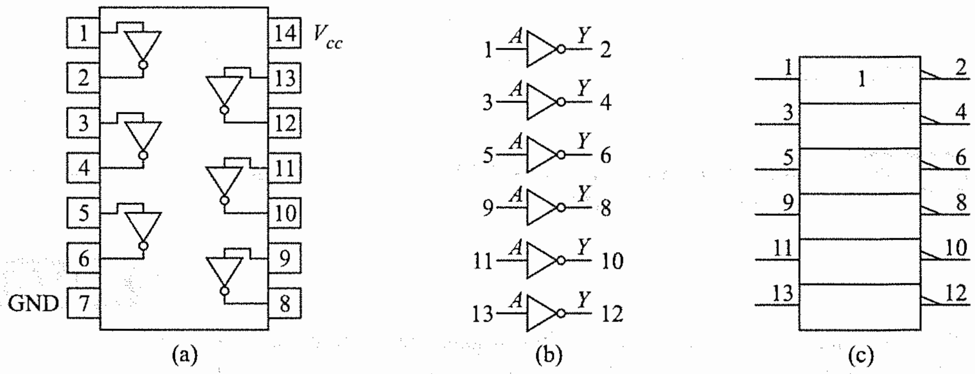


Fig. 1.36 Hex inverter, 7404: (a) Pin configuration, (b) Logic symbol, (c) Logic symbol (IEEE)

A 7411 is an IC that contains three 3-input AND gates. The DIP package and pinout for the 7411 are shown in Fig. 1.37a, and the standard logic symbols are given in Fig. 1.37b. The new IEEE symbol for the AND gate is a rectangle with the ampersand (&) symbol written in it; is used in Fig. 1.37c to show the three AND gates in the 7411.

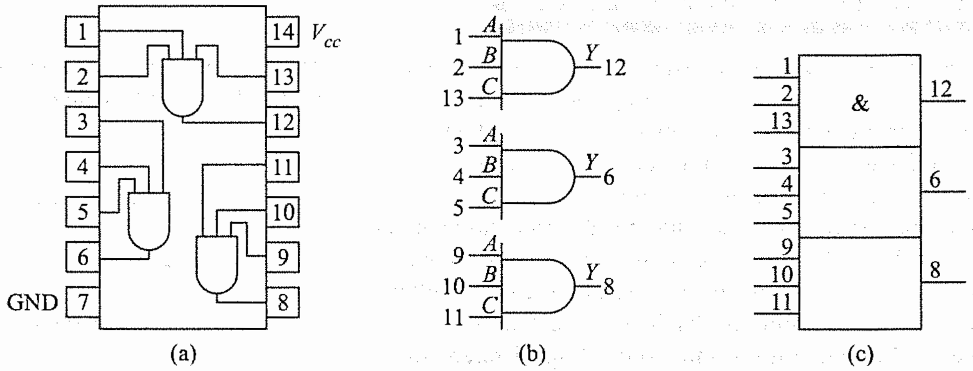


Fig. 1.37 Triple three-input AND gate, 7411: (a) Pin configuration, (b) Logic symbol, (c) Logic symbol (IEEE)

The pinout and symbols for the 7432 quad 2-input OR gate are shown in Fig. 1.38. The term *quad* means "four," and there are four gates in this DIP. The IEEE symbol for an OR gate is a rectangular box with the greater than or equal to (≥) symbol inside. This symbol means "at least one input must be high in order for the output to be high."

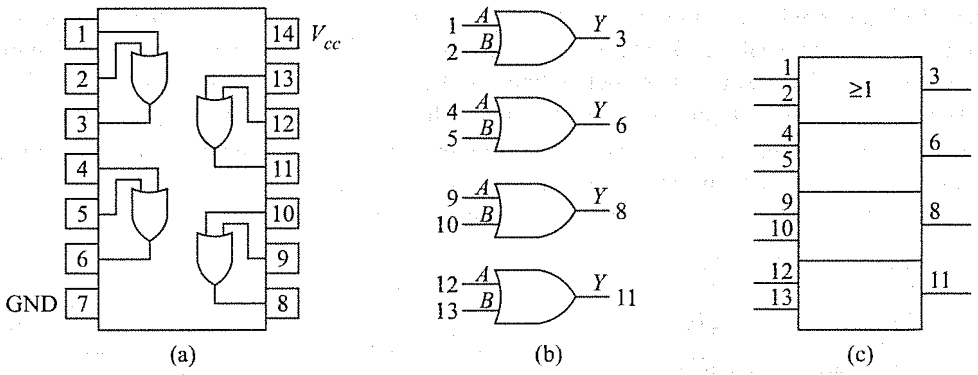


Fig. 1.38 **Quad two-input OR gate, 7432: (a) Pin configuration, (b) Logic symbol, (c) Logic symbol (IEEE)**

SELF-TEST

27. What is generally accepted as the number of gates per chip for SSI, MSI, and LSI?
28. Which is faster, TTL or ECL? Which requires more power to operate?
29. Over what temperature range will 74XX TTL operate?
30. When referring to TTL ICs, what is the meaning of quad? of hex?
31. In the 74ACTXX family of ICs, what do the letters A, C, and T stand for?
32. What is the significance of the triangle on the output line of the inverter in Fig. 1.35b?

1.8 DIGITAL IC SIGNAL LEVELS

The voltages in Fig. 1.39 are used to define the two digital logic levels, $H = 1 = T$ and $L = 0 = F$. Logic level 1 is any voltage between $+V_{CC}$ and $+V_{H,\min}$. Logic level 0 is any voltage between $+V_{L,\max}$ and 0. Voltages within the *forbidden* region are not allowed. This illustration is often called a *profile*, and it can be used to define the operation of any digital logic circuit. Each family of digital circuits has its own unique operating characteristics, and each individual circuit has an input and an output. Thus, there must be an input profile and an output profile for each family. An understanding of the correct voltage levels for each digital family is absolutely essential. Measuring logic levels in the laboratory, interconnecting different logic families, and connecting digital logic circuits with other digital circuits require a detailed knowledge of voltage levels. For now, we will consider the two most popular TTL families and one of the widely used CMOS families. Profiles for other circuits are easily obtained from manufacturers' data books.

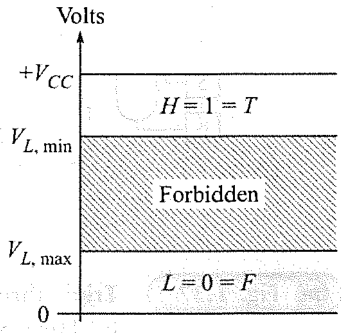


Fig. 1.39 **Logic level profile**

TTL Logic Levels

The 74XX and 74LSXX are the two most widely used TTL families, and they have identical voltage-level characteristics (there is a difference in current capabilities, however).

The input profile and the output profile for the 74XX and the 74LSXX family are shown in Fig. 1.40. From the output profile, each circuit will produce a voltage between $+V_{CC}$ = +5 Vdc and $V_{OH,min}$ = 2.4 Vdc to signify $H = 1$. A voltage level between $+V_{OL,max}$ = 0.4 Vdc and 0 Vdc will be produced to signify $L = 0$.

From the input profile, it is seen that any voltage between $+V_{CC}$ = +5 Vdc and $+V_{IH,min}$ = 2.0 Vdc is recognized as $H = 1$. Any voltage between $+V_{IL,max}$ = 0.8 Vdc and 0 Vdc is recognized as $L = 0$.

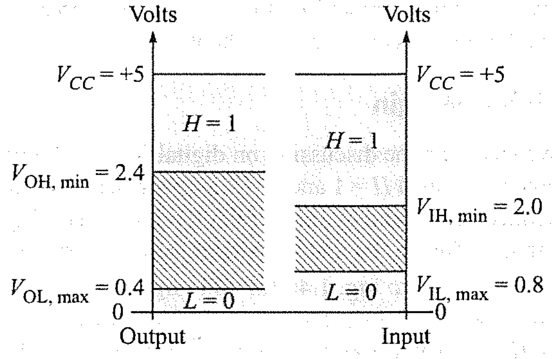


Fig. 1.40 74XX and 74LSXX profiles

Look carefully at Fig. 1.40 and note that any output voltage within the high range is *within* the input range recognized as a high. Similarly, any output voltage within the low range is *within* the input range recognized as a low. Clearly an output voltage from any 74XX circuit can be used as the input signal to any other 74XX circuit! This family of circuits is thus said to be *compatible*. This shouldn't come as a surprise, since any circuit within a family should be able to "drive" any other circuit within the same family. Similarly, any 74LSXX circuit can drive any other 74LSXX circuit—this is also a compatible family. Furthermore, the 74XX and 74LSXX families are compatible with one another.

There are, however, differences in the *number* of circuits that can be connected to the output in each family. This consideration, called *fanout*, is discussed in Chapter 13.

CMOS Logic Levels

74HCXX is the most widely used CMOS family. The input profile and the output profile for the 74HCXX family are shown in Fig. 1.41. From the output profile, each circuit will produce a voltage between $+V_{CC}$ = +5 Vdc and $+V_{OH,min}$ = 4.9 Vdc to signify $H = 1$. A voltage level between $+V_{OL,max}$ = 0.1 Vdc and 0 Vdc will be produced to signify $L = 0$.

From the input profile, it is seen that any voltage between $+V_{CC}$ = +5 Vdc and $V_{IH,min}$ = 3.5 Vdc is recognized as $H = 1$. Any voltage between $+V_{IL,max}$ = 1.5 Vdc and 0 Vdc is recognized as $L = 0$.

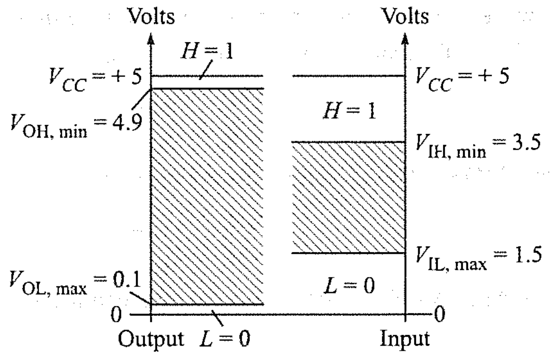


Fig. 1.41 74HCXX profiles

Look carefully at Fig. 1.41 and note that any output voltage within the high range is *within* the input range recognized as a high. Similarly, any output voltage within the low range is *within* the input range recognized as a low. Clearly an output voltage from any 74HCXX circuit can be used as the input signal to any other 74HCXX circuit! This family of circuits is thus said to be *compatible*.

By comparing the profiles in Figs. 1.40 and 1.41, you can see that a 74HCXX CMOS circuit can be used to drive any 74XX TTL circuit. However, a 74XX TTL *cannot* be used to drive a 74HCXX CMOS. The voltage levels *are not* compatible. Interconnecting different families like this is called *interfacing*. Both interfacing and fanout are considered in Chapter 13.

Noise Margin

We shall end the discussion on digital IC signal levels by introducing the concept of *noise margin*. We have noted that level $H = 1$ and $L = 0$ are represented by a range of voltages. Consider, output of a digital device is connected to input of another digital device (see Fig. 1.42) but some noise (in the form of a random voltage) can get added to the output voltage before it arrives at the input of next device.

Now, refer to Fig. 1.40 for TTL digital devices. The output for $H = 1$ can be lower than 5 V and can go as low as $V_{OH,min}$(2.4 V). The input of a TTL device is treated as $H = 1$ if it's within the range +5 V to $V_{IH,min}$(2.0 V). An addition of any random noise voltage greater than 2.0 V–2.4 V = –0.4 V (e.g., –0.3 V, –0.2 V) makes the summer output greater than 2.0 V, the minimum acceptable level $V_{IH,min}$ at input side. Thus, there exists a noise margin, the amount of noise that can get added without any possibility of logic value misinterpretation. This can be defined for $H = 1$ as

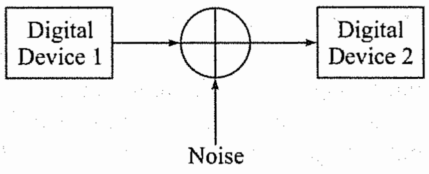


Fig. 1.42 Noise affecting output of a digital device

$$NM_H = V_{IH,min} - V_{OH,min} \tag{1.2}$$

Similarly, a noise margin for $L = 0$ exists, beyond which output of a device at $L = 0$ may fail to get identified as the same, at the input of a similar device due to noise corruption. The worst-case scenario here can make $V_{OL,max}$ plus noise go above $V_{IL,max}$. Thus noise margin for $L = 0$ can be defined as

$$NM_L = V_{IL,max} - V_{OL,max} \tag{1.3}$$

Note the polarity of the noise voltage in above two cases. For NM_H the corrupting noise voltage should be negative and less than the noise margin to cause any misinterpretation. If it is positive, there is no such issue. However, for NM_L the corrupting noise voltage should be positive when misinterpretation may occur.

For TTL 74XX and 74LSXX family,

$$NM_H = V_{IH,min} - V_{OH,min} = 2.0 - 2.4\text{ V} = -0.4\text{ V}$$
$$NM_L = V_{IL,max} - V_{OL,max} = 0.8 - 0.4 = 0.4\text{ V}$$

For CMOS 74HCXX family, from Fig. 1.41

$$NM_H = V_{IH,min} - V_{OH,min} = 3.5 - 4.9\text{ V} = -1.4\text{ V}$$
$$NM_L = V_{IL,max} - V_{OL,max} = 1.5 - 0.1 = 1.4\text{ V}$$

Note that CMOS has better noise margin characteristics over TTL.

SUMMARY

This introductory chapter in *digital principles* is intended to provide a clear concept of a digital signal in an electronic circuit or system. Both ideal and realistic digital voltage levels are presented. How these levels vary with time (waveforms) is illustrated, and the concepts of rise time, fall time, and duty cycles are defined. Simple ideal switch models are used to illustrate digital circuit operation. Symbols and operation of the following basic digital circuits are presented: buffer, tri-state buffer, inverter, tri-state inverter, AND gate, and OR gate. The flip-flop is introduced as a basic memory element, and both serial and parallel shift registers are discussed. The basic conceptual operation of a number of common MSI and LSI digital circuits is covered: encoders, decoders, multiplexers, demultiplexers, ALUs, counters, and comparators. These basic elements are then discussed in the context of their use in a simple digital computer. Finally, currently available digital ICs, including the 54/74XX TTL, 74HCXX CMOS, and MECL families, are discussed.

GLOSSARY

- ***ALU*** Arithmetic logic unit.
- ***analog signal*** A signal whose amplitude can take any value between given limits. A continuous signal.
- ***binary number*** A number code that uses only the digits 0 and I to represent quantities.
- ***bipolar*** Having two types of charge carriers; a bipolar transistor is *npn* or *pnp*.
- ***bit*** *b*inary dig*it*.
- ***buffer*** A digital circuit capable of maintaining a required logic level while acting as a current source or a current sink for a given load.
- ***chip*** A small piece of semiconductor on which an IC is formed.
- ***CMOS*** Complementary metal-oxide silicon. An IC using both *n*-channel and *p*-channel field-effect transistors (FETs).
- ***CPU*** Central processing unit.
- ***CRT*** Cathode-ray tube.
- ***clock*** A periodic, rectangular waveform used as a basic timing signal.
- ***computer architecture*** Microprocessor and other elements building a computer.
- ***counter*** A digital circuit designed to keep track of (to count) a number of events.
- ***decoder*** A unit designed to change a digital number into another form.
- ***demultiplexer (DEMUX)*** A digital circuit that will select only one of many inputs.
- ***digital signal*** A signal whose amplitude can have only given discrete values between defined limits. A signal that changes amplitude in discrete steps.
- ***DIP*** Dual-inline package.
- ***DMA*** Direct memory access.
- ***Duty cycle*** For a periodic digital signal, the ratio of high level time to the period or the ratio of low level time to the period.
- ***ECL*** Emitter-coupled logic.
- ***encoder*** A unit designed to change a given signal into a digital number.
- ***flip-flop*** An electronic circuit that can store one bit of a binary number.
- ***floppy disk*** A magnetically coated disk used to store digital data.
- ***gate*** A digital circuit having two or more inputs and a single output.
- ***handshaking*** A "request" to transfer data into or out of a computer, followed by an "acknowledge" signal, allowing data transfer to begin.
- ***IC*** Integrated circuit.
- ***logic circuit*** A digital circuit, a switching circuit, or any kind of two-state circuit that duplicates mental processes.
- ***LSI*** Large-scale integration.
- ***memory*** The area of a digital computer used to store programs and data.

- ***memory element*** Any device or circuit used to store 1 bit of a binary number.
- ***microprocessor*** An IC around which many small computer systems are constructed.
- ***MSI*** Medium-scale integration.
- ***multiplexer (MUX)*** A digital circuit that will connect a single input to any one of many possible outputs.
- ***noise margin*** Allowable level of additive noise for proper interpretation of logic value.
- ***parallel shifting*** Transferring all bits in a binary number (digital data) simultaneously.
- ***port*** A register that serves as a place to either input data to or extract data from a digital system.
- ***program*** A detailed set of instructions used to direct the operation of a computer.
- ***programmer*** A person who writes programs for digital computer systems.
- ***Schottky diode*** A special kind of diode that can be very rapidly switched on and off. The combination of a Schottky diode with a bipolar transistor is called a *Schottky transistor*.
- ***serial shifting*** Transferring each bit in a binary number (digital data), one bit at a time, one after the other.
- ***SSI*** Small-scale integration.
- ***tri-state circuit*** A digital circuit having three states—high, low, and open.
- ***truth table*** A table that shows all of the input-output possibilities of a digital circuit.
- ***TTL*** Transistor-transistor logic. The widely used 54XX/74XX family of bipolar junction transistor (BJT) integrated circuits.
- ***VLSI*** Very large-scale integration.

PROBLEMS

Section 1.1

1.1 Describe in your own words the characteristics of an analog signal and a digital signal.

1.2 Use four indicator lamps to illustrate the decimal number 3.

1.3 Demonstrate that only three indicator lamps are required to display the eight decimal numbers 0. 1, 2, 3. 4, 5, 6, and 7.

1.4 On certain wall clocks, the representation of the progression of seconds is analog, while on other clocks it is digital. What is the difference?

Section 1.2

1.5 Make a sketch similar to Fig. 1.4c to illustrate that $V_{OH,min}$ = 3.9 Vdc, $V_{QL,max}$ = 0.8 Vdc, +5 Vdc $\geq H \geq V_{OH,min}$, and 0 Vdc $\leq L \leq V_{OL,max}$.

1.6 A certain digital logic family has levels which are given as 0 Vdc $\geq H \geq -0.2$ Vdc and -2.0 Vdc $\geq L \geq -2.5$ Vdc. Make a sketch similar to Fig. 1.4c to illustrate the allowed logic levels. (This is similar to the popular ECL family of digital circuits.) Is this positive or negative logic?

1.7 The waveform in Fig. 1.6b has a duty cycle H = 20 percent, and the positive pulses occur every 500 μs. What is the width of each positive pulse?

1.8 Draw a waveform similar to Fig. 1.6b if H = +5 Vdc, L = 0 Vdc, and Duty cycle H = 90 percent. Does this resemble a series of negative pulses?

1.9 Make a sketch of an ideal symmetrical 1-MHz square wave having H = +5 Vdc and L = 0 Vdc (similar to Fig. 1.6). Directly under this waveform, sketch a nonideal waveform having the same values but with a rise time of 0.5 μs and a fall time of 0.5 μs. What has happened to the square wave?

Section 1.3

1.10 Construct a truth table for a 3-input AND gate. *Hint:* Use the binary numbers in Table 1.1.

1.11 Construct a truth table for a 3-input OR gate. *Hint:* Use the binary numbers in Table 1.1.

1.12 Determine the state of V_o in Fig. 1.43.

1.13 In order to obtain a digital signal at V_o, in Fig. 1.44, G must be high or low. What must the state of V_i and G in Fig. 1.44 be if $V_o = L$?

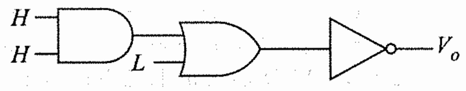


Fig. 1.43

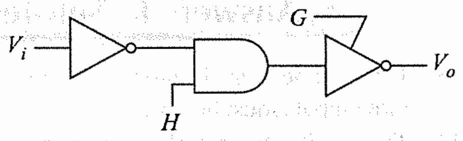


Fig. 1.44

Section 1.4

1.14 Draw a circuit that can be used to SET and RESET the flip-flop in Fig. 1.16. Use only one switch and one inverter. With the switch in one position, the flip-flop will SET. Moving the switch to the other position will RESET the flip-flop.

1.15 Explain how two 4-bit serial registers could be used to form a single 8-bit serial register. Draw a diagram to illustrate this.

1.16 A 16-bit serial register requires 500 ns for each shift operation. How much time is required to enter or extract a 16-bit binary number?

1.17 A modern computer uses a 32-bit microprocessor and shifts data in parallel between its microprocessor and the input-output port register. How many connections (wires) must be made between the microprocessor and the register if the data are truly shifted in parallel?

To reduce the number of connections, data are sometimes shifted in groups of 2. That is, the first 16 bits are shifted in parallel, and then the next 16 bits are shifted in parallel. If this is done, the number of connections is cut in half. How many connections are needed here if this is done?

Compare the times required to shift a 32-bit number for each case if one shift operation requires 250 ns.

Section 1.5

1.18 What is the largest decimal count possible with a 7-flip-flop counter?

1.19 If both the A and the B inputs to the ALU in Fig. 1.22 are 4 bits, what is the largest single-digit decimal number that can be represented? What if the buses were only three bits each?

1.20 The ALU in Fig. 1.22 is set to perform an AND function. What is F if $A = 0101$ and $B = 0100$? What is F if the ALU is changed to the OR function and A and B remain the same?

1.21 Write binary values for F and CARRY OUT shown in Fig. 1.23 if $A = 0011$ and $B = 0001$.

1.22 Determine the output logic levels for the comparator in Fig. 1.24 if

a. $A = 0111, B = 0111$
b. $A = 1000, B = 0111$
c. $A = 0011, B = 0100$

1.23 Show the proper logic levels on the decoder in Fig. 1.26b if the digital input is $ABCD$ = 1001.

Section 1.6

1.24 Give definitions for the following abbreviations:

a. ALU b. CPU
c. ECL

1.25 Discuss the term *handshaking* in reference to the block diagram in Fig. 1.32.

1.26 What are the basic blocks included in the architecture of a typical microprocessor?

1.27 Why must the bus connecting the CPU and the data memory be bidirectional?

1.28 Draw a block diagram, similar to Fig. 1.27, of a machine, process, or system (similar to the lawn-watering system discussed) that is controlled by a computer. Describe any sensors needed, and give a general discussion of the system operation.

Section 1.7

1.29 You are asked to recommend a family of TTL digital circuits that will operate at temperatures as high as 50°C and as low as –10°C. Power requirements are to be kept as low as possible, and the system is to operate at the highest possible clock frequency. What would you recommend?

1.30 A co-worker asks you to explain the functions of the TTL designated as a 7404 and also wants to know the difference between a 7404 and a 74HC04.

1.31 You are designing a system that uses 250 standard 74XX TTL gates. What is the total power required?

1.32 Repeat Prob. 1.31 using 74LSXX gates.

1.33 Draw from memory the standard symbols and the new IEEE/IEC symbols for an inverter, a 2-input AND gate, and a 2-input OR gate.

Answers to Self-tests

1. continuous
2. T
3. 0111
4. Positive logic
5. The term $V_{OH,min}$ stands for the minimum value of the output voltage when *high*. The term $V_{OL,max}$ stands for the maximum value of the output voltage when low.
6. V_o may have a value within the forbidden region only during the short time while transitioning from high to low or low to high. When not switching (static), V_o must either be in the high band or the low band.
7. $0.9H = 4.5$ Vdc, and $1.1L = 1.1$ Vdc.
8. Duty cycle $H = 2/7 = 0.286 = 28.6$ percent.
9. V_o is open in the first instance, and then it is high.
10. V_o is high in the first instance, and then it is open.
11. V_o is low.
12. V_o would be low. To produce $V_o = H$, *all* three inputs must be high.
13. V_o would be high. To produce $V_o = L$, *all* three inputs must be low.
14. The possibilities are almost endless!
15. $A = L$
16. This is a 4-bit number. $DCBA = 1001$.
17. It would take only 1 μs for the parallel register, but it would require 8 μs for the serial register.
18. A port is usually a register that serves as a place to either input data to or extract data from the microprocessor or computer.
19. 0111
20. Ten
21. a. No b. Yes c. Yes
22. 0110
23. The clock provides a periodic digital signal that is used to time computer operations.
24. It is a specific list of instructions, prepared by a programmer, that directs the actions of the computer.
25. Arithmetic computations and logical decisions.

26. The DMA allows the transfer of data directly between memory and external devices, without passing through the CPU.
27. SSI, less than 12; MSI, more than 12 but less than 100; LSI, more than 100.
28. ECL is faster but requires more power.
29. 0°C to 75°C
30. *Quad* means "four." *Hex* means "six."
31. *A*—advanced, *C*—CMOS, *T*—TTL compatible.
32. The output is active when low.

Digital Logic

2

OBJECTIVES

- Write the truth tables for, and draw the symbols for, 2-input OR, AND, NOR, and NAND gates.
- Write Boolean equations for logic circuits and draw logic circuits for Boolean equations.
- Use DeMorgan's first and second theorems to create equivalent circuits.
- Understand the operation of AND-OR-INVERT gates and expanders.

A digital circuit having one or more input signals but only one output signal is called a *gate*. In Chapter 1, the most basic gates—the NOT gate (inverter), the OR gate and the AND gate—were introduced. Connecting the basic gates in different ways makes it possible to produce circuits that perform arithmetic and other functions associated with the human brain (an ALU). Because they simulate mental processes, gates are often called *logic circuits*. A discussion of both positive and negative logic leads to the important concept of *assertion-level logic*.

Hardware description languages (HDL) are an alternative way of describing logic circuits. This uses a set of textual codes that is machine (computer) readable. The concept is relatively new and is useful for design, testing and fabrication of complex digital circuits. We'll have a soft introduction of HDL towards the end of this chapter. We'll learn it in detail in later part of this book introducing features relevant to each chapter.

2.1 THE BASIC GATES—NOT, OR, AND

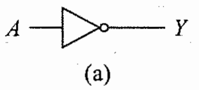
Is an action right or wrong? A motive good or bad? A conclusion true or false? Much of our thinking involves trying to find the answer to two-valued questions like these. Two-state logic had a major influence on Aristotle, who worked out precise methods for getting to the truth. Logic next attracted mathematicians, who intuitively sensed some kind of algebraic process running through all thought.

Augustus De Morgan came close to finding the link between logic and mathematics. But it was George Boole (1854) who put it all together. He invented a new kind of algebra that replaced Aristotle's verbal methods. Boolean algebra did not have an impact on technology, however, until almost a century later. In 1938 Shannon applied the new algebra to telephone switching circuits. Because of Shannon's work, engineers soon realized that Boolean algebra could be used to analyze and design computer circuits.

Three logic circuits, the *inverter*, the *OR gate*, and the *AND gate*, can be used to produce any digital system. The function of each of these gates was introduced in Chapter 1. Let's look more closely at the operation of each circuit and also at their Boolean expressions.

The Inverter (NOT Gate)

Figure 2.1 is the symbol and truth table for an inverter. In one truth table, the symbols H and L are used, while the binary numbers 0 and 1 are used in the other. The information in each table is identical, however, since we know $L = 0$ and $H = 1$. In this text, both symbols are used, hence since there is no chance for confusion. You will find both symbols used in other texts, as well as in manufacturers' data sheets. The important idea is that there are only two possible voltage levels (low and high) associated with a digital circuit. This fits nicely with the binary number system, since it has only two values (0 and 1). This is often referred to as *two-state operation*. By definition, this is *positive logic*, since the *higher* voltage level is assigned binary 1. Later in this chapter, we will consider *negative logic*, where the *higher* voltage level is assigned binary (zero).



A	Y
L	H
H	L

A	Y
0	1
1	0

(b)

Fig. 2.1 **(a) Inverter symbol, (b) Truth tables**

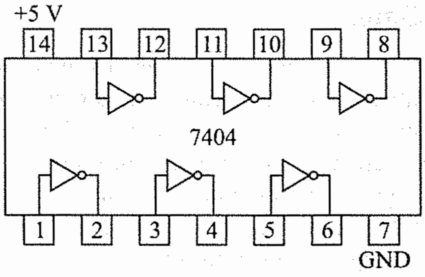


Fig. 2.2 **Pinout diagram of a 7404**

Figure 2.2 shows the pinout diagram of a 7404 hex inverter. This IC contains six inverters. After applying +5 Vdc (the supply voltage for all TTL devices) to pin 14 and grounding pin 7, you can connect any or all inverters to other TTL devices. For instance, if you only need one inverter, you can connect an input signal to pin 1 and take the output signal from pin 2; the other inverters can be left unconnected.

In Boolean algebra a variable can be either 0 or 1. The output Y of not gate is always complement of input A. In equation form

$$Y = \text{not } A \quad \text{i.e.} \quad Y = A' \quad \text{so that, if } A = 0, Y = 0' = 1 \quad \text{and if} \quad A = 1, Y = 1' = 0$$

The truth tables in Fig. 2.1 illustrate signal levels that do not change with time. However, almost all digital signals do in fact change with time, as illustrated by the waveforms in Chapter 1 (Sec. 1.2). Here are two examples that illustrate how to use the truth table information with signals that vary with time.

Example 2.1 A 1-kHz square wave drives pin 1 of a 7404 (see Fig. 2.2). What does the voltage waveform at pin 2 look like?

Solution Figure 2.3a shows what you will see on a dual-trace oscilloscope. Assuming you have set the sweep timing to get the upper waveform (pin 1), then you would see an inverted square wave on pin 2.

Example 2.2 If a 500-Hz square wave drives pin 3 of a 7404, what is the waveform on pin 4?

Solution Pins 3 and 4 are the input and output pins of an inverter (see Fig. 2.2). A glance at Fig. 2.3b shows the typical waveforms on the input (pin 3) and output (pin 4) of a 7404. Again, the output waveform is the complement of the input waveform. Because of two-state operation, rectangular waveforms like this are the normal shape of digital signals. Incidentally, a *timing diagram* is a picture of the input and output waveforms of a digital circuit. Examples of timing diagrams are shown in Figs. 2.3a and b.

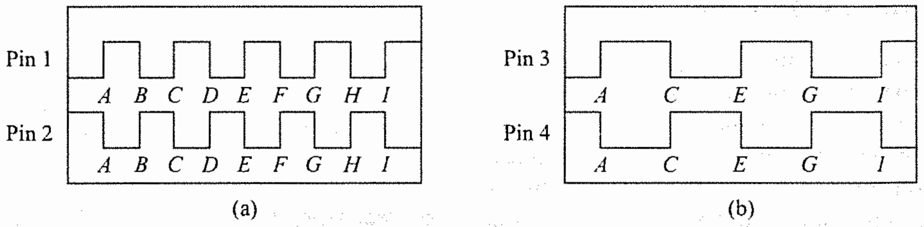
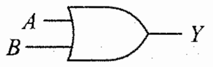


Fig. 2.3

OR Gates

An OR gate has two or more input signals but only one output signal. It is called an OR gate because the output voltage is high if any or all of the input voltages are high. For instance, the output of a 2-input OR gate is high if either or both inputs are high. Figure 2.4a shows the logic symbol of a 2-input OR gate and Fig. 2.4b its truth table.



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

(a) (b)

Fig. 2.4 **(a) OR gate, (b) Truth table**

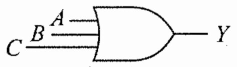
In Boolean equation form

$$Y = A \text{ OR } B, \quad \text{i.e.} \quad Y = A + B$$

so that $Y = 0 + 0 = 0$, $Y = 0 + 1 = 1$, $Y = 1 + 0 = 1$ and $Y = 1 + 1 = 1$.

The '+' sign here represents logic operation OR and not addition operation of basic arithmetic. Note that in arithmetic 1 + 1 = 2 in decimal and 1 + 1 = 10 in binary number system (Table 1.1 of Chapter 1). Binary addition is discussed in detail in Chapter 6.

Three Inputs Figure 2.5 shows a 3-input OR gate. The inputs are A, B, and C. When all inputs are low, Y is low. If A or B or C is high, Y will be high. The truth table summarizes all input possibilities. In equation form, the three input OR gate is represented as: $Y = A + B + C$.



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

(a) (b)

Fig. 2.5 **(a) Three-input OR gate, (b) Truth table**

The truth table (Fig. 2.5b) allows us to check that all input possibilities are included. Why? Because every possibility is included when the input entries follow a binary sequence. For example, the first *ABC* entry is 000, the next is 001, then 010, and so on, up to the final entry of 111. Since all binary numbers are present, all input possibilities are included.

Incidentally, the number of rows in a truth table equals 2^n, where *n* is the number of inputs. For a 2-input OR gate, the truth table has 2^2, or 4 rows. A 3-input OR gate has a truth table with 2^3, or 8 rows, while a 4-input OR gate results in 2^4, or 16 rows, and so on.

An OR gate can have as many inputs as desired. No matter how many inputs, the action of any OR gate is summarized like this: One or more high inputs produce a high output.

Logic Symbols Figure 2.6a shows the symbol for a 2-input OR gate of any design. Whenever you see this symbol, remember the output is high if either input is high.

Shown in Fig. 2.6b is the logic symbol for a 3-input OR gate. Figure 2.6c is the symbol for a 4-input OR gate. For these gates, the output is high when any input is high. The only way to get a low output is by having all inputs low.

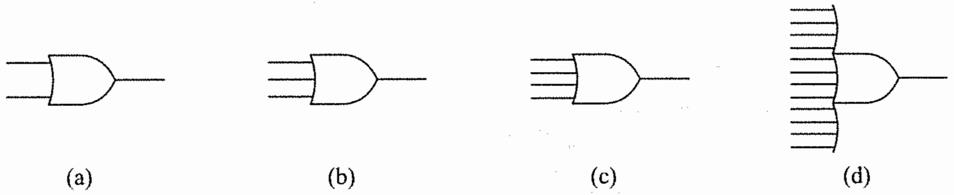


Fig. 2.6 **OR gate symbols: (a) Two-input, (b) Three-input, (c) Four input, (d) Twelve-input**

When there are many input signals, it's common drafting practice to extend the input side as needed to allow sufficient space between the input lines. For instance, Fig. 2.6d is the symbol for a 12-input OR gate. The same idea applies to any type of gate; extend the input side when necessary to accommodate a large number of input signals.

TTL OR Gates Figure 2.7 shows the pinout diagram of a 7432, a TTL quad 2-input OR gate. This digital IC contains four 2-input OR gates inside a 14-pin DIP. After connecting a supply voltage of +5 V to pin 14 and a ground to pin 7, you can connect one or more of the OR gates to other TTL devices.

Timing Diagram Figure 2.8 shows an example of a timing diagram for a 2-input OR gate. The input voltages drive pins 1 and 2 of a 7432. Notice that the output (pin 3) is low only when both inputs are low. The output is high the rest of the time because one or more input pins are high.

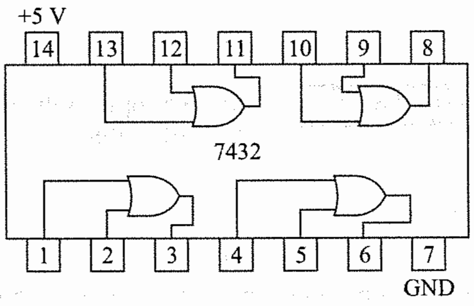


Fig. 2.7 **Pinout diagram of a 7432**

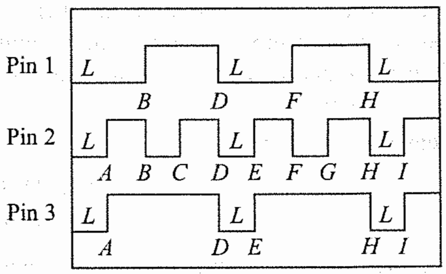


Fig. 2.8 **Timing diagram**

Example 2.3 Work out the truth table for Fig. 2.9a.

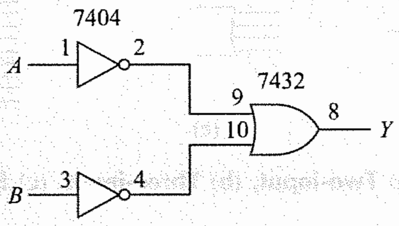
Solution With two input signals (A and B), four input cases are possible: low-low, low-high, high-low, and high-high. For convenience, let L stand for low and H for high. Then, the input possibilities are LL, LH, HL, and HH, as listed in Fig. 2.9b. Here is what happens for each input possibility.

CASE 1 A is low and B is low. With both input voltages in the low state, each inverter has a high output. This means that the OR gate has a high output, the first entry of Fig. 2.9b.

CASE 2 A is low and B is high. With these inputs the upper inverter has a high output, while the lower inverter has a low output. Since the OR gate still has a high input, the output Y is high.

CASE 3 A is high and B is low. Now, the upper inverter has a low output and the lower inverter has a high output. Again, the OR gate produces a high output, so that Y is high.

CASE 4 A is high and B is high. With both inputs high, each inverter has a low output. This time, the OR gate has all inputs in the low state, so that Y is low, as shown by the final entry of Fig. 2.9b.



(a)

A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

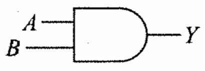
(b)

Fig. 2.9 Logic circuit and truth table of Example 2.3

Incidentally, the circuit of Fig. 2.9a uses only one-third of a 7404 and one-fourth of a 7432. The other gates in these digital ICs are not connected, which is all right because you don't have to use all of the available gates.

AND Gates

The AND gate has a high output only when all inputs are high. Figure 2.10a shows a 2-input AND gate. The truth table (Fig. 2.10b) summarizes all input-output possibilities for a 2-input AND gate. Examine this table carefully and remember the following: the AND gate has a high output only when A and B are high. In other words, the AND gate is an all-or-nothing gate; a high output occurs only when all inputs are high. This truth table uses 1s and 0s, where $1 = H$ and $0 = L$.



(a)

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

(b)

Fig. 2.10 (a) Two-input AND gate, (b) Truth table

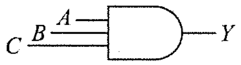
In Boolean equation form

$$Y = A \text{ AND } B, \text{ i.e. } Y = A.B \text{ or } Y = AB$$

so that, $Y = 0.0 = 0$, $Y = 0.1 = 0$, $Y = 1.0 = 0$ and $Y = 1.1 = 1$

The '.' sign here represents logic AND operation and not multiplication operation of basic arithmetic though the result are same for both.

Three Inputs Figure 2.11a shows a 3-input AND gate. The inputs are A, B, and C. When all inputs are low, Y is low. If even one input is low, Y is in the low state. The only way to get a high output is to raise all inputs to the high state (+5 V) The truth table (Fig. 2.11b) summarizes all input-output possibilities. In equation form, the three input AND gate is represented as: $Y = A.B.C = ABC$.



(a)

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

(b)

Fig. 2.11 (a) Three-input AND gate, (b) Truth table

Logic Symbols Figure 2.12a shows the symbol for a 2-input AND gate of any design. Shown in Fig. 2.12b is the logic symbol for a 3-input AND gate. Figure 2.12c is the symbol for a 4-input AND gate. Remember: For any of these gates, the output is high only if all inputs are high. As before, it's common drafting practice to extend the input sides when there are many input signals. For instance, Fig. 2.12d is the symbol for a 12-input AND gate.

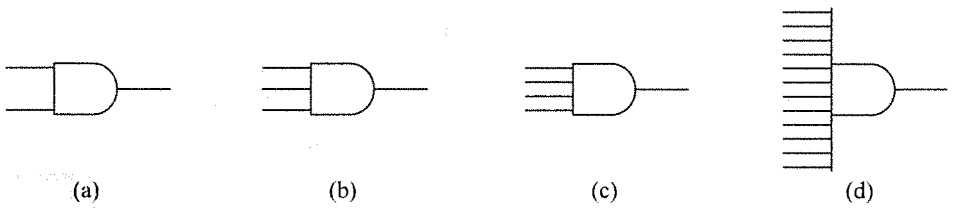


Fig. 2.12 AND gate symbols: (a) Two-input, (b) Three-input, (c) Four-input, (d) Twelve-input

TTL AND Gates Figure 2.13 shows the pinout diagram of a 7408, a TTL quad 2-input AND gate. This digital IC contains four 2-input AND gates. After connecting a supply voltage of +5V to pin 14 and a ground to pin 7, you can connect one or more of the AND gates to other TTL devices. TTL AND gates are also available in triple 3-input and dual 4-input packages. (See Appendix 3 for pinout diagrams.)

Timing Diagram Figure 2.14 shows an example of a timing diagram for a 2-input AND gate. The input voltages drive pins 1 and 2 of a 7408. Notice that the output (pin 3) is high only when both inputs are high (between C and D, G and H, etc.). The output is low the rest of the time.

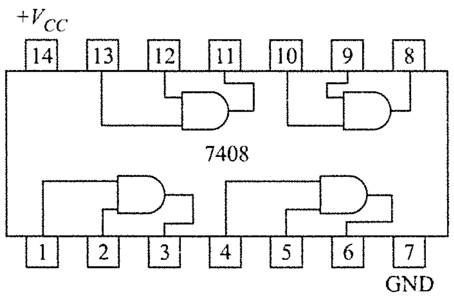


Fig. 2.13 Pinout diagram of a 7408

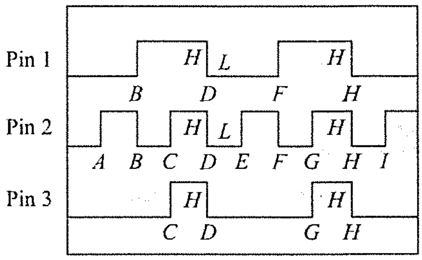
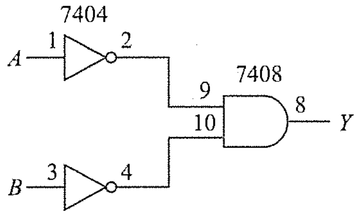


Fig. 2.14 Timing diagram

Example 2.4 Work out the truth table for Fig. 2.15a.



(a)

A	B	A'	B'	$Y = A'.B'$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

(b)

Fig. 2.15 Logic circuit and truth table of Example 2.4

Solution We get the final truth table here in slightly different way. Consider, one logic gate at a time as shown in Fig. 2.15b. The NOT gate connected to A gives A' at its output and is shown in column 3. The NOT gate connected to B gives B' at its output and is shown in column 3. Finally, the 4th column shows OR operation on column 3 and 4 to give the final output Y. Here is what happens for each input possibility.

CASE 1 A is low and B is low. With both input voltages in the low state, each inverter has a high output. This means the AND gate has a high output, the first entry of Fig. 2.16.

CASE 2 A is low and B is high. With these inputs the upper inverter has a high output, while the lower inverter has a low output. Since the AND gate produces a low output, Y is low.

CASE 3 A is high and B is low. Now, the upper inverter has a low output and the lower inverter has a high output. Again, the AND gate produces a low output, so Y is low.

CASE 4 A is high and B is high. With both inputs high, each inverter has a low output. Again, the AND gate has a low output, as shown by the final entry of Fig. 2.16.

A	B	Y
L	L	H
L	H	L
H	L	L
H	H	L

Fig. 2.16

Note that input-output relations described in Fig. 2.15b and Fig. 2.16 are same.

Example 2.5 What is the Boolean equation for the logic circuit of Fig. 2.17a?

Solution This circuit is called an AND-OR network because input AND gates drive an output OR gate. The intermediate outputs are

$$Y_3 = AB \qquad Y_6 = CD$$

The final output is

$$Y_8 = Y_3 + Y_6$$
$$Y = AB + CD$$

An equation in this form is referred to as a *sum-of-products equation*. AND-OR networks always produce sum-of-products equations.

Example 2.6 Write the Boolean equation for Fig. 2.17b.

Solution This logic circuit is called an OR-AND network because input OR gates drive an output AND gate. The intermediate outputs are $Y_8 = A + B$ and $Y_{11} = C + D$.

The final output is

$$Y_6 = Y_8 Y_{11}$$

or

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

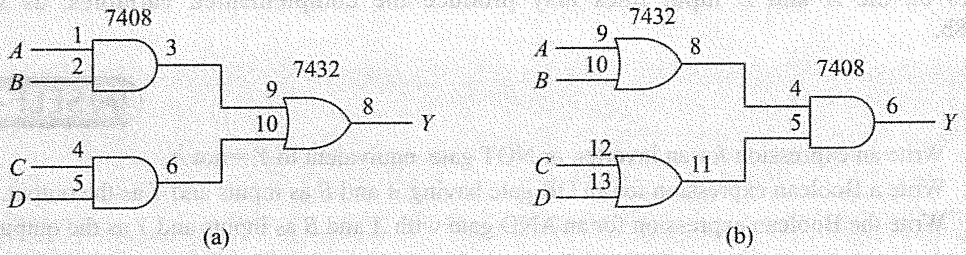


Fig. 2.17 **(a) AND-OR, (b) OR-AND network**

As shown in this equation, parentheses may be used to indicate a logical product (ANDing). Also notice that the final answer is a product of sums. OR-AND networks always produce *product-of-sums equations*.

Example 2.7 What is the logic circuit whose Boolean equation is

$$Y = \bar{A}BC + A\bar{B}C$$

Solution This is a sum-of-products equation with some of the inputs in complemented form. Figure 2.18a shows an AND-OR circuit with the foregoing Boolean equation. The upper AND gate produces a logical product of

$$Y_{12} = \bar{A}BC$$

The lower AND gate produces

$$Y_6 = A\bar{B}C$$

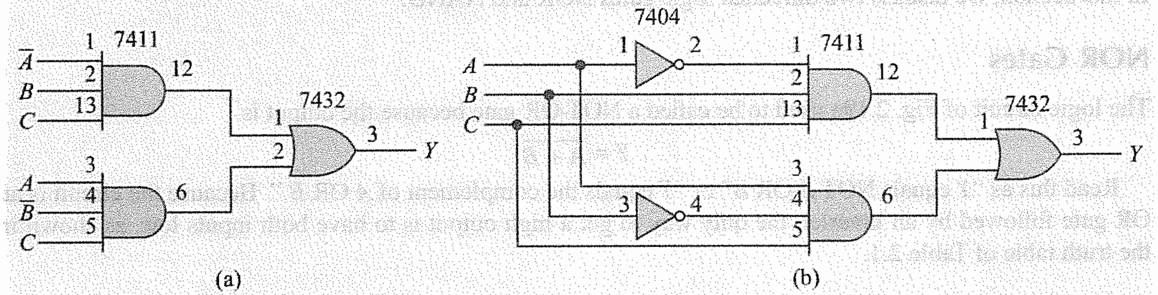


Fig. 2.18 **(a) Intermediate, (b) Final logic circuit of Example 2.7**

SELF-TEST

1. A system in which $H = 1$ and $L = 0$ is (positive, negative) logic.
2. A gate whose output is H if any input is H is an _______ gate.
3. A gate whose output is H only when all inputs are H is an _______ gate.

The final output therefore equals the sum of the Y_{12} and Y_6 products:

$$Y = \overline{A}BC + A\overline{B}C$$

The complemented inputs A and B may be produced by other circuits (discussed later). Alternatively, inverters on the A and B input lines may produce the complemented variables, as shown in Fig. 2.18b.

SELF-TEST

4. Write an expression for an inverter, or NOT gate equivalent to $Y =$ not A.
5. Write a Boolean expression for an OR gate having A and B as inputs and Y as the output.
6. Write the Boolean expression for an AND gate with A and B as inputs and Y as the output.

This example illustrates one method of logic design. Whenever you are given a sum-of-products equation, you can draw the corresponding AND-OR network using AND gates to produce the logical products and an OR gate to produce the sum.

2.2 UNIVERSAL LOGIC GATES—NOR, NAND

In the previous section we have seen how AND, OR and NOT gates can be connected together to realize any logic function. Here, we address an interesting question. Is it possible to use only one type of gate for this purpose? If possible, one needs to procure only one type of gate for his design. And more importantly, fabrication of Integrated Circuit that performs a logic operation becomes easier when gate of only one kind is used. Gates, which can perform this task, are called universal logic gates. Clearly, basic gates like AND, OR and NOT don't fit into this category for the simple reason that conversion among themselves itself are not possible. As for example, one cannot gate OR operation by using any number or combination of AND gates. In this section, we discuss two universal logic gates NOR and NAND.

NOR Gates

The logic circuit of Fig. 2.19a used to be called a NOT-OR gate because the output is

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

Read this as "Y equals NOT A OR B" or "Y equals the complement of A OR B." Because the circuit is an OR gate followed by an inverter, the only way to get a high output is to have both inputs low, as shown in the truth table of Table 2.1.

NOR Gate Symbol

The logic circuit of Fig. 2.19a has become so popular that the abbreviated symbol of Fig. 2.19b is used for it. The bubble (small circle) on the output is a reminder of the inversion that takes place after the ORing. Furthermore, the words NOT-OR are contracted to the word NOR. So from now, we will call the circuit a NOR gate and will use the symbol of Fig. 2.19b. Whenever you see this symbol, remember that the output is NOT the OR of the inputs. With a NOR gate, all inputs must be low to get a high output. If any input is high, the output is low.

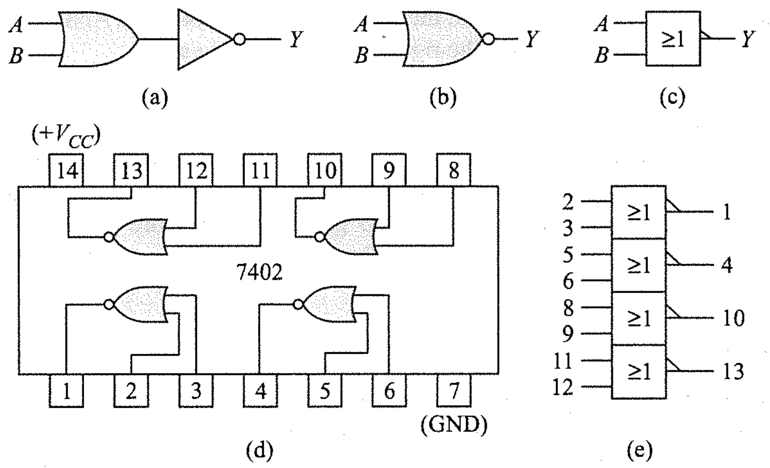


Fig. 2.19 NOR logic gate

Figure 2.19c shows the new IEEE rectangular symbol for the NOR gate. The small triangle on the output is equivalent to the bubble used on the standard symbol. The indicator ≥ inside the box means "if one or more of the inputs are high, the output is high."

The 7402 is a quad 2-input NOR gate in a 14-pin DIP as illustrated in Fig. 2.19d. The new rectangular symbol for the 7402 is shown in Fig. 2.19e.

Table 2.1 NOR Gate

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

Bubbled AND Gate

Figure 2.20a shows inverters on the input lines of an AND gate. This logic circuit is often drawn in the abbreviated form shown in Fig. 2.20b. The bubbles on the inputs are a reminder of the inversion that takes place before the AND operation. We will refer to the abbreviated drawing of Fig. 2.20b as *a bubbled AND gate*. We have already analyzed this circuit in Example 2.4 and obtained its truth table as shown in Fig. 2.15b. We find that output Y and inputs A, B are identical for bubbled AND gate and NOR gate. Therefore, these two circuits are equivalent and thus interchangeable. Given any logic circuit with NOR gates, we can replace it by bubbled AND gates and converse is also true.

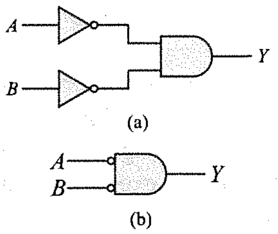


Fig. 2.20 (a) AND gate with inverted inputs, (b) Equivalent symbol

De Morgan's First Theorem

The Boolean equation for Fig. 2.19b is

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

The Boolean equation for Fig. 2.20b is

$$Y = \overline{A}\,\overline{B}$$

The first equation describes a NOR gate, and the second equation a bubbled AND gate. Since the outputs are equal for the same inputs, we can equate the right-hand members to get

$$\overline{A+B} = \overline{A}\,\overline{B} \tag{2.1}$$

This identity is known as *De Morgan's first theorem.* In words, it says the complement of a sum equals the product of the complements. This can also be proved by comparing the truth tables shown in Fig. 2.4(b) and NOR gate truth table of Table 2.1. A similar exercise that compares truth tables of three input NOR gate and three input bubbled AND gate show they are identical and we can write, $(A + B + C)' = A'B'C'$. Note that this equivalence can be extended to gates or circuits for larger number of inputs, too.

Universality of NOR Gate

Figure 2.21 shows how all other logic gates can be obtained from NOR gates. To get a NOT gate we tie inputs of NOR gate together (Fig. 2.21a) so that there is only one input to the circuit. If input is 0, then both the inputs to NOR gate are 0. Following NOR gate truth table (Table 2.1) we see output now is 1. Similarly, if input is 1, both the inputs to NOR gate are 1 that gives output 0. Therefore output of circuit, shown in Fig. 2.21a is complement of its input and thus gives NOT operation.

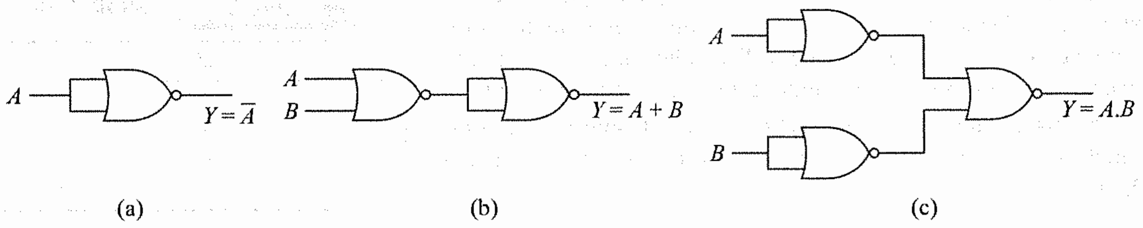


Fig. 2.21 **Universality of NOR gate (a) NOT from NOR, (b) OR from NOR, (c) AND from NOR**

Figure 2.21b shows how to get OR circuit using only NOR gates. The first NOR gate performs usual NOR operation while second NOR gate performs as NOT gate and inverts the NOR logic to OR.

To understand how we get AND circuit using only NOR gates (Fig. 2.21c) let us refer to example 2.3. The configuration is similar except the output there is generated from OR and here from NOR and of course the NOT gates are replaced by NOR equivalent. Since NOR gate is NOT operation followed by OR we invert the output of example 2.3, shown in Fig. 2.9b to get output of this circuit. Thus output of circuit in Fig. 2.21c is high only when both the inputs are high and it functions like an AND gate.

The above equivalences can be proved simply, by applying Boolean theorems and we'll discuss those theorems in next chapter. Since, we can perform all the Boolean operations using only NOR gates it is termed as universal logic gate.

Eye of the Beholder

Which brings us to a principle. Truth tables, logic circuits, and Boolean equations are different ways of looking at the same thing. Whatever we learn from one viewpoint applies to the other two. If we prove that truth tables are identical, this immediately tells us the corresponding logic circuits are interchangeable, and their Boolean equations are equivalent. When analyzing, we generally start with a logic circuit, construct its truth table, and summarize with the Boolean equation. When designing, we often start with a truth table, generate a Boolean equation, and arrive at a logic circuit.

Example 2.8 A 7402 is a quad 2-input NOR gate. This TTL IC has four 2-input NOR gates in a 14-pin DIP as shown in Appendix 3. What is the Boolean equation for the output of Fig. 2.22a?

Solution The AND gates produce AB and CD. These are ORed to get $AB + CD$. The final inversion gives

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

The circuit of Fig. 2.22a is known as an AND-OR-INVERT network because it starts with ANDing, follows with ORing, and ends with INVERTing.

The AND-OR-INVERT network is available as a separate TTL gate. For instance, the 7451 is a dual 2-input 2-wide AND-OR-INVERT gate, meaning two networks like Fig. 2.22a in a single 14-pin TTL package. Appendix 3 shows the pinout diagram. Figure 2.22b shows how we can use half of a 7451 to produce the same output as the circuit of Fig. 2.22a.

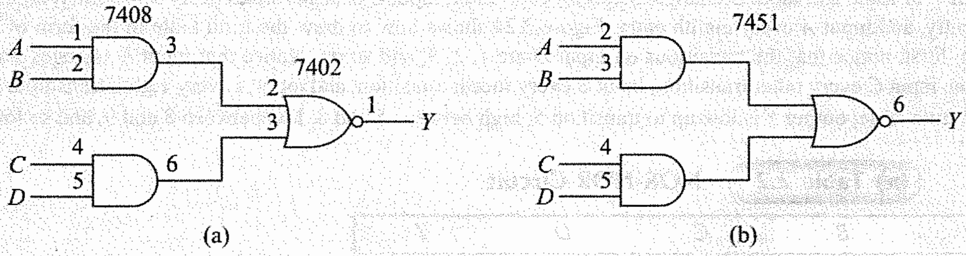


Fig. 2.22 AND-OR-INVERT network

Example 2.9 Prove that Fig. 2.23c is logically equivalent to Fig. 2.23a.

Solution De Morgan's first theorem says we can replace the final NOR gate of Fig. 2.23a by a bubbled AND gate to get the equivalent circuit of Fig. 2.23b. If you invert a signal twice, you get the original signal back again. Put another way, double inversion has no effect on the logic state; double invert a low and you still have a low; double-invert a high and you still have a high. Therefore, each double inversion in Fig. 2.23b (a pair of bubbles on the same signal line) cancels out, leaving the simplified circuit of Fig. 2.23c. Therefore, Fig. 2.23a and Fig. 2.23c are equivalent or interchangeable.

Why would anyone want to replace Fig. 2.23a by 2.23c? Suppose your shelves are full of AND gates and OR gates. If you have just run out of NOR gates and you are trying to build a NOR-NOR network like Fig. 2.23a, you can connect the OR-AND circuit of Fig. 2.23c because it produces the same output as the original circuit. In general, this idea applies to any circuit that you can rearrange with De Morgan's theorem. You can build whichever equivalent circuit is convenient.

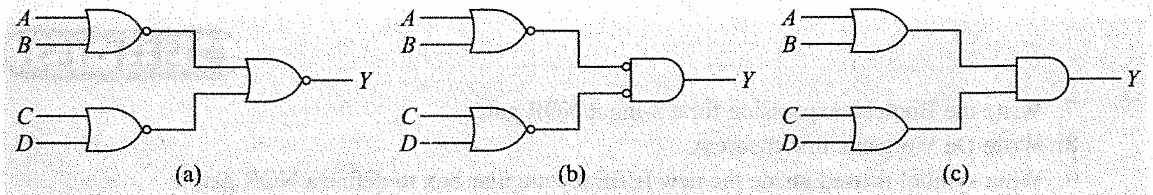


Fig. 2.23 Equivalence among logic circuits: Example 2.9

Example 2.10 What is the truth table for the NOR-NOR circuit of Fig. 2.23a?

Solution First, realize that the truth table of Fig. 2.23a is the same for 2.23b and 2.23c because all circuits are equivalent. Therefore, we can analyze whichever circuit we want to. Figure 2.23c is the easiest for most people to analyze, so let us use it in the following discussion.

Table 2.2 lists every possibility starting with all inputs low and progressing to all inputs high. Notice that the total number of entries equals 2^4 or 16. By analyzing each input possibility, we can work out the corresponding output. For instance, when all inputs are low in Fig. 2.23c. both OR gates have low outputs, so the AND gate produces a low output. This is the first entry of Table 2.2. Proceeding like this, we can determine the output for the remaining possibilities and arrive at all the entries shown in Table 2.2.

Example 2.11 Convert Table 2.2 into a timing diagram.

Solution In Table 2.2, input D changes states for each entry, input C changes states every other entry, input B every fourth entry, and input A every eighth entry. Figure 2.24 shows how to draw the truth table in the form of a timing diagram. First, notice that the transitions on input D are 1, 2, 3, and so on. Notice that input D changes states each transition, input C every other transition, input B every fourth transition, and input A every eighth transition. To agree with the truth table, output Y is low up to transition 5, high between 5 and 8, low between 8 and 9, and so forth.

Table 2.2 NOR-NOR Circuit

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

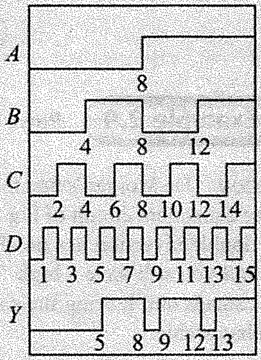


Fig. 2.24 Timing diagram

SELF-TEST

7. Write the Boolean expression for a 2-input NOR gate.
8. Write De Morgan's first theorem.
9. What symbol is used inside the new IEEE rectangular box to define a NOR gate?

NAND Gates

Originally, the logic circuit of Fig. 2.25a was called NOT-AND gate because the output is

$$Y = \overline{AB}$$

Read this as "Y equals NOT A AND B" or "Y equals the complement of A AND B." Because the circuit is an AND gate followed by an inverter, the only way to get a low output is for both inputs to be high, as shown in the truth table of Table 2.3.

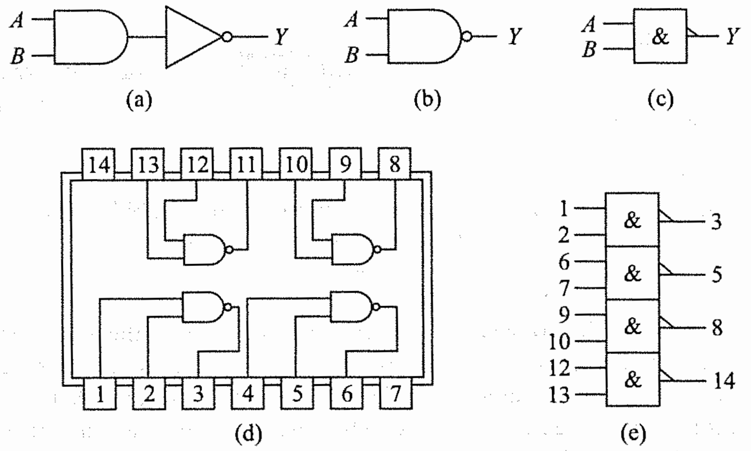


Fig. 2.25 NAND logic gate

NAND-Gate Symbol

The logic circuit of Fig. 2.25a has become so popular that the abbreviated symbol of Fig. 2.25b is used for it. The bubble on the output reminds us of the inversion after the ANDing. Also, the words NOT-AND are contracted to NAND. Whenever you see this symbol, remember that the output is NOT the AND of the inputs. With a NAND gate, all inputs must be high to get a low output. If any input is low, the output is high.

Table 2.3 NAND Gate

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

Figure 2.25c shows the new IEEE rectangular symbol for the NAND gate. The small triangle on the output is equivalent to the bubble used on the standard symbol. The indicator "&" inside the box means "the output is high only when *all* inputs are high."

The 7400 is a quad 2-input NAND gate in a 14-pin DIP as illustrated in Fig. 2.25d. The new rectangular symbol for the 7402 is shown in Fig. 2.25e.

Bubbled OR Gate

Figure 2.26a shows inverters on the input lines of an OR gate. The circuit is often drawn in the abbreviated form shown in Fig. 2.26b, where the bubbles represent inversion. We will refer to the abbreviated drawing of Fig. 2.2b as a *bubbled OR gate.* We have already analyzed this circuit in Example 2.3 and obtained its truth table in Fig. 2.9b. We see that output Y and inputs A, B are identical for bubbled OR gate and NAND gate. Therefore, these two circuits are equivalent and thus interchangeable. Given any logic circuit with NOR gates, we can replace it by bubbled AND gates and converse is also true.

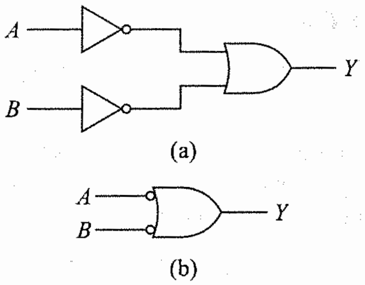


Fig. 2.26 (a) OR gate with inverted inputs, (b) Equivalent symbol

De Morgan's Second Theorem

The Boolean equation for Fig. 2.24b is

$$Y = \overline{AB}$$

The Boolean equation for Fig. 2.25b is

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

The first equation describes a NAND gate, and the second equation a bubbled OR gate. Since the outputs are equal for the same inputs, we can equate the right-hand members to get

$$\overline{AB} = \overline{A} + \overline{B} \tag{2.2}$$

This identity is known as *De Morgan's second theorem*. It says the complement of a product equals the sum of the complements. This can also be proved by comparing the truth tables shown in Fig. 2.3(b) and NAND gate truth table of Table 2.2. A similar exercise that compares truth tables of three input NAND gate and three input bubbled OR gate show they are identical and we can write, $(A.B.C)' = A' + B' + C'$. Note that this equivalence can be extended to gates or circuits with any number of inputs.

Universality of NAND Gate

Figure 2.27 shows how all other logic gates can be obtained from NAND gates and why it is called a universal logic gate. Figure 2.27a shows how we tie inputs of NAND gate together (as we had done in case of NOR gate) to get a NOT gate that has only one input. If input is 0, then both the inputs to NAND gate are 0. Following NAND gate truth table (Table 2.3) we see output now is 1. Similarly, if input is 1, both the inputs to NAND gate are 1 that gives output 0. Therefore output of circuit, shown in Fig. 2.27a is complement of its input and thus gives NOT operation.

Figure 2.27b shows how we get AND circuit using only NAND gates. The second NAND gate performs as a NOT gate and inverts the NAND logic of first NAND gate to AND logic.

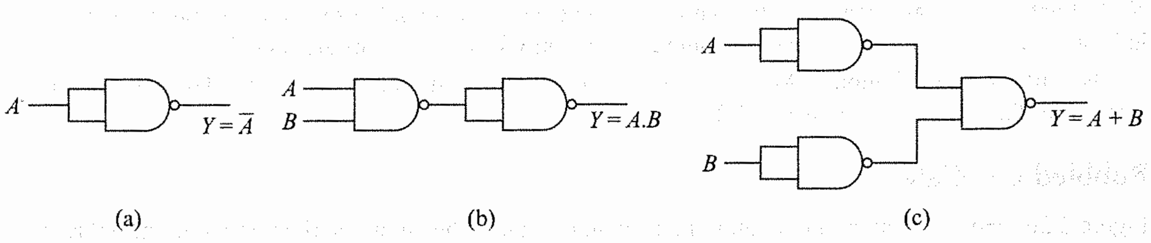


Fig. 2.27 Universality of NAND gate: (a) NOT from NAND, (b) AND from NAND, (c) OR from NAND

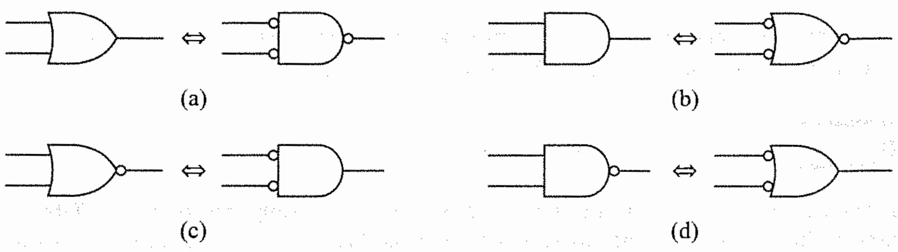


Fig. 2.28 **Useful logic equivalences**

To obtain OR logic using NAND gate we compare Fig. 2.27c circuit with Fig. 2.15a. The later gives NOR logic and has AND gate at output. The present circuit has NAND gate at output and thus inverts the output of previous circuit, from NOR to OR.

TTL NAND Gates

The NAND gate is the backbone of the 7400 TTL series because most devices in this family are derived from it. Because of its central role in TTL technology, the NAND gate has become the least expensive and most widely used TTL gate. Furthermore, the NAND gate is available in more configurations than other gates, as shown in Table 2.4. Notice that the NAND gate is available as a 2-, 3-, 4-, or 8-input gate. The other gates have fewer configurations, with the OR gate available only in 2-input form.

Table 2.4 **Standard TTL Gates**

Type	*Quad 2-Input*	*Triple 3-Input*	*Dual 4-Input*	*Single 8-Input*
NAND	7400	7410	7420	7430
NOR	7402	7427	7425	
AND	7408	7411	7421	
OR	7432			

Example 2.12 Prove that Fig. 2.29c is logically equivalent to Fig. 2.29a.

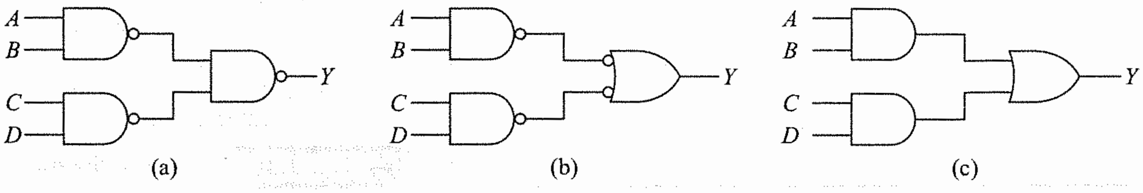


Fig. 2.29 **Equivalence of logic gates: Example 2.12**

Solution De Morgan's second theorem says we can replace the final NAND gate of Fig. 2.29a by a bubbled OR gate to get the equivalent circuit of Fig. 2.29b. Each double inversion in Fig. 2.29b cancels out, leaving the simplified circuit of Fig. 2.29c. Therefore, Figs. 2.29a and 2.29c are equivalent.

Incidentally, most people find Fig. 2.29b easy to analyze because they learn to ignore the double inversions and see only the simplified AND-OR circuit of Fig. 2.29c. For this reason, if you build a NAND-NAND Network like Fig. 2.29a, you can draw it like Fig. 2.29b. Anyone who sees Fig. 2.29b on a schematic diagram will know it is two input NAND gates driving an output NAND gate. Furthermore, when troubleshooting the circuit, they can ignore the bubbles and visualize the easy-to-analyze AND-OR circuit of Fig. 2.29c.

Example 2.13 What is the truth table for the NAND-NAND circuit of Fig. 2.29a?

Solution Let us analyze the equivalent circuit of Fig. 2.29c because it is simpler to work with. Table 2.5 lists every possibility starting with all inputs low and progressing to all inputs high. By analyzing each input possibility, we can determine the resulting output. For instance, when all inputs are low in Fig. 2.29c, both AND gates have low outputs, so the OR gate produces a low output. This is the first entry of Table 2.5. Proceeding like this, we can arrive at the output for the remaining possibilities of Table 2.5.

Example 2.14 Show a timing diagram for the NAND-NAND circuit of Fig. 2.29a.

Solution All you have to do is convert the low-high states of Table 2.5 into low-high waveforms like Fig. 2.30. First, notice that the transitions on input *D* are numbered 1, 2, 3, and so on. Input *D* changes states each transition, input *C* every other transition, input *B* every fourth transition, and input *A* every eighth transition. To agree with the truth table, output *Y* is low up to transition 3, high between 3 and 4, low between 4 and 7, and so forth.

Table 2.5 NAND-NAND Circuit

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

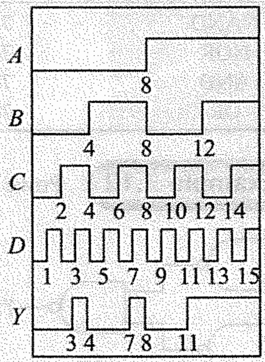


Fig. 2.30 Timing diagram

SELF-TEST

10. Write the Boolean expression for a 2 input NAND gate.
11. Write De Morgan's second theorem.
12. What symbol is used inside the IEEE rectangular box to define a NAND gate?

2.3 AND-OR-INVERT GATES

Figure 2.31a shows an AND-OR circuit. Figure 2.31b shows the De Morgan equivalent circuit, a NAND-NAND network. In either case, the Boolean equation is

$$Y = AB + CD$$

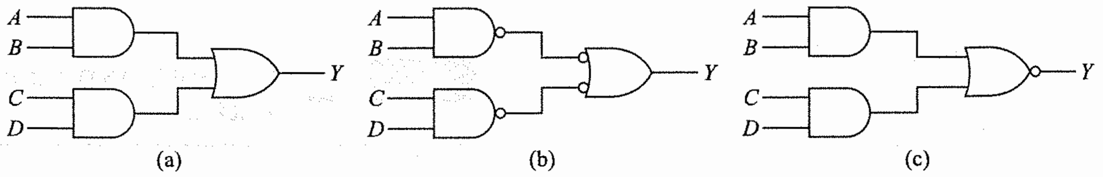


Fig. 2.31 (a) AND-OR circuit, (b) NAND-NAND circuit, (c) AND-OR-INVERT circuit

Since NAND gates are the preferred TTL gates, we would build the circuit of Fig. 2.31b. As you know, NAND-NAND circuits like this are important because with them you can build any desired logic circuit.

TTL Devices

AND-OR circuits are not easily derived from the basic NAND-gate design. But it is easy to get an AND-OR-INVERT circuit as in Fig. 2.31c. A variety of circuits like this are available as TTL chips. Because of the inversion, the output has the equation shown below.

$$Y = \overline{AB + CD} \tag{2.3}$$

Table 2.6 lists the AND-OR-INVERT gates available in the 7400 series. In this table, *2-wide* means two AND gates across, *4-wide* means four AND gates across, and so on. For instance, the 7454 is a 2-input 4-wide AND-OR-INVERT gate as in Fig. 2.32a; each AND gate has two inputs (2-input), and there are four AND gates (4-wide). Figure 2.32b shows the 7464; it is a 2-2-3-4-input 4-wide AND-OR-INVERT gate.

Table 2.6 AND-OR-INVERT Gates

Device	*Description*
7451	Dual 2-input 2-wide
7454	2-input 4-wide
7459	Dual 2-3-input 2-wide
7464	2-2-3-4-input 4-wide

Connecting the output of a 2-input 2-wide AND-OR-INVERT gate to an inverter will give us the same output as an AND-OR circuit.

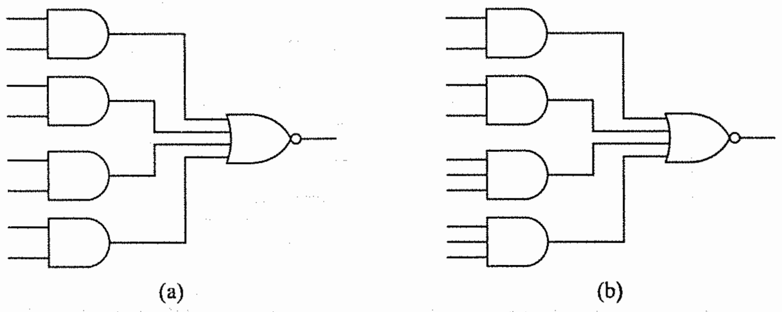


Fig. 2.32 Examples of AND-OR circuits

Expandable AND-OR-INVERT Gates

The widest AND-OR-INVERT gate available in the 7400 series is 4-wide. What do we do when we need a 6- or 8-wide circuit? One solution is to use an *expandable* AND-OR-INVERT gate.

Figure 2.33 shows the logic symbol for an expandable AND-OR-INVERT gate. There are two additional inputs, labeled *bubble* and *arrow*. Table 2.7 lists the expandable AND-OR-INVERT gates in the 7400 series.

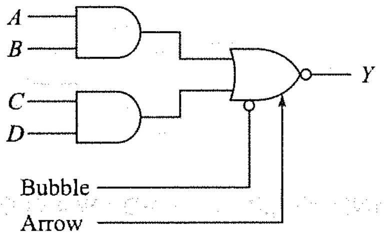


Fig. 2.33 Expandable AND-OR-INVERT gate

Table 2.7 Expandable AND-OR-INVERT Gates

Device	*Description*
7450	Dual 2-input 2-wide
7453	2-input 4-wide
7455	4-input 2-wide

Expanders

What do we connect to the arrow and bubble inputs of an expandable gate? We connect the output of an *expander* as in Fig. 2.34a. Connect bubble to bubble and arrow to arrow.

Visualize the outputs of Fig. 2.34a connected to the arrow and bubble inputs of Fig. 2.33. Figure 2.34b shows the logic circuit. This means that the expander outputs are being ORed with the signals of the AND-OR-INVERT gate. In other words, Fig. 2.34b is equivalent to the AND-OR-INVERT circuit of Fig. 2.34c.

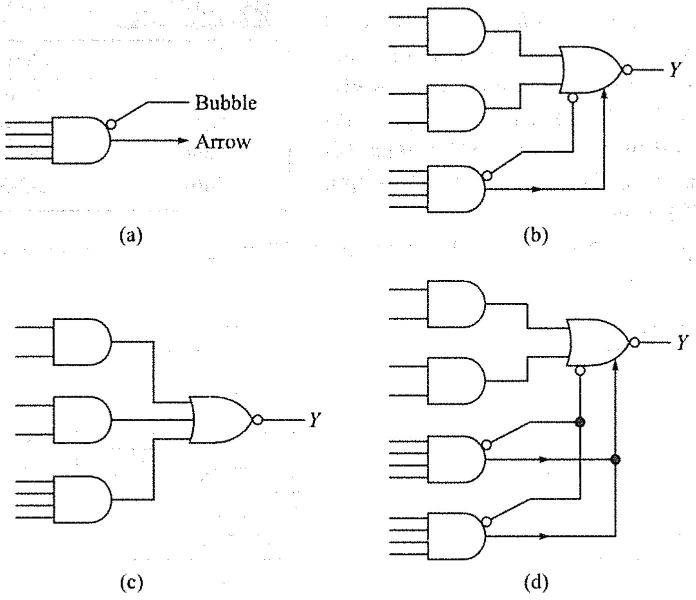


Fig. 2.34 (a) Expander, (b) Expander driving expandable AND-OR-INVERT gate, (c) AND-OR-INVERT circuit, (d) Expandable AND-OR-INVERT with two expanders

We can connect more expanders. Figure 2.34d shows two expanders driving the expandable gate. Now we have a 2-2-4-4-input 4-wide AND-OR-INVERT circuit.

The 7460 is a dual 4-input expander. The 7450, a dual expandable AND-OR-INVERT gate, is designed for use with up to four 7460 expanders. This means that we can add two more expanders in Fig. 2-34d to get a 2-2-4-4-4-4-input 6-wide AND-OR-INVERT circuit.

SELF-TEST

13. When we speak of an AND-OR-INVERT gate, what is the meaning of 2-wide?
14. What is the purpose of using an *expander* with an AND-OR-INVERT gate?

2.4 POSITIVE AND NEGATIVE LOGIC

Up to now, we have used a binary 0 for low voltage and a binary 1 for high voltage. This is called *positive logic*. People are comfortable with positive logic because it feels right. But there is another code known as *negative logic* where binary 0 stands for high voltage and binary 1 for low voltage. Even though it seems unnatural, negative logic has many uses. The following discussion introduces some of the terminology and concepts for both types of logic.

Positive and Negative Gates

An OR gate in a positive logic system becomes an AND gate in a negative logic system. Why? Look at the gate of Fig. 2.35. We have been calling it an OR gate. This is correct, provided we are using positive logic. Table 2.8 shows the *truth* for the gate of Fig. 2.35, no matter what you call it. That is, if either input is high in Fig. 2.35, the output is high.

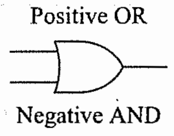


Fig. 2.35 **Meaning of symbol depends on whether you use positive or negative logic**

Table 2.8

A	*B*	*Y*
Low	Low	Low
Low	High	High
High	Low	High
High	High	High

In a positive logic system, binary 0 stands for low and binary 1 for high. So, we can convert Table 2.8 to Table 2.9. Note that *Y* is a 1 if either *A* or *B* is 1. This sounds like an OR gate. And it is, because we are using positive logic. To avoid ambiguity, we can call Fig. 2.35 a positive OR gate because it performs the OR function with positive logic. (Some data sheets describe gates as positive OR gate, positive AND gate, etc.)

In a negative logic system, binary 1 stands for low and binary 0 for high. With this code, we can convert Table 2.8 to Table 2.10. Now, watch what happens. The output *Y* is a 1 only when both *A* and *B* are 1. This sounds like an AND gate! And it is, because we are now using negative logic. In other words, gates are defined by the way they process the binary 0s and 1s. If you use binary 1 for low voltage and binary 0 for high voltage, then you have to refer to Fig. 2.35 as a negative AND gate.

As you see, the gate of Fig. 2.35 always produces a high output if either input is high. But what you call it depends on whether you see positive or negative logic. Use whichever name applies. With positive logic, call it a positive OR gate. With negative logic, call it a negative AND gate.

Table 2.9

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

Table 2.10

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

In a similar way, we can show the truth table of other gates with positive or negative logic. By analyzing the inputs and outputs in terms of 0s and 1s, you find these equivalences between the positive and negative logic:

Positive OR	↔ negative AND
Positive AND	↔ negative OR
Positive NOR	↔ negative NAND
Positive NAND	↔ negative NOR

Table 2.11 summarizes these gates and their definitions in terms of voltage levels. These definitions are always valid. If you get confused from time to time, refer to Table 2.11 to get back to the ultimate meaning of the basic gates.

Table 2.11 **Voltage Definitions of Basic Gates**

Gate	*Definition*
Positive OR/negative AND	Output is high if any input is high.
Positive AND/negative OR	Output is high when all inputs are high.
Positive NOR/negative NAND	Output is low if any input is high.
Positive NAND/negative NOR	Output is low when all inputs are high.

Assertion-Level Logic

Why do we even bother with negative logic? The reason is related to the concept of *active-low signals*. For instance, the 74150 multiplexer has an active-low input strobe; this input turns on the chip only when it is low (negative true). This is an active-low signal; it causes something to happen when it is low, rather than high. As another example, the 74154 decoder has 16 output lines; the decoded output signal is low (negative true). In other words, all output lines have a high voltage, except the decoded output line. Besides TTL devices, microprocessor chips like the 8085 have a lot of active-low input and output signals.

Many designers draw their logic circuits with bubbles on all pins with active-low signals and omit bubbles on all pins with active-high signals. This use of bubbles with active-low signals is called *assertion-level logic*. It means that you draw chips with the kind of input that causes something to happen, or with the kind of output that indicates something has happened. If a low input signal turns on a chip, you show a bubble on that input. If a low output is a sign of chip action, you draw a bubble on that output. Once you get used to assertion-level logic, you may prefer drawing logic circuits this way.

One final point. Sometimes you hear expressions such as "The inputs are asserted" or "What happens when the inputs are asserted?" An input is *asserted* when it is active. This means it may be low or high, depending on whether it is an active-low or active-high input. For instance, given a positive AND gate, all inputs must be asserted (high) to get a high output. As another example, the STROBE input of a TTL multiplexer must be asserted (low) to turn on the multiplexer. In short, you can equate the word *assert* with *activate*. You assert, or activate, the inputs of a gate or device to get something to happen.

Points to Remember

Here are some ideas that you should try to remember:

1. Positive true always represents a high voltage, and negative true always represents a low voltage.
2. If possible, draw basic gates with bubbles on active-low signal lines.
3. When a signal is active-low, use an overbar as a reminder that the signal voltage is negative true when the underlying statement is true.

Example 2.15

(a) The number stored in a register may be zero (all bits low). Show how to detect this condition.
(b) What change in (a) will detect presence of the word 10110101 in the 8-bit register?

Solution

(a) Figure 2.36 shows a design using assertion-level logic. The bits go to a bubbled AND gate (the same as positive NOR gate). When all the bits are low, output ZERO is high. Because of the inverter, the final output $\overline{\text{ZERO}}$ is active-low. Therefore, when the sum is zero, $\overline{\text{ZERO}}$ is negative true.
(b) Some of the bubbles at the input of the bubbled AND gate need to be removed. These are for locations in the code word where '1' is present, specifically S_7, S_5, S_4, S_2 and S_0.

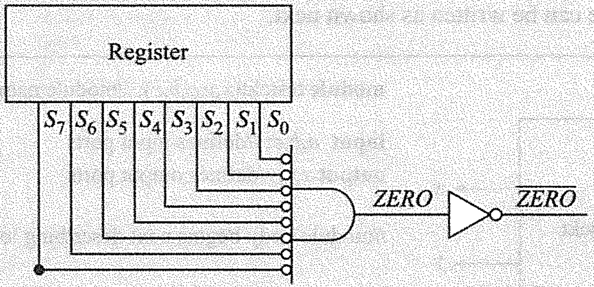


Fig. 2.36 **Assertion-level logic diagram showing the detection of zero and minus accumulator contents**

SELF-TEST

15. What is negative logic?
16. What is meant by *assertion-level logic*?

2.5 INTRODUCTION TO HDL

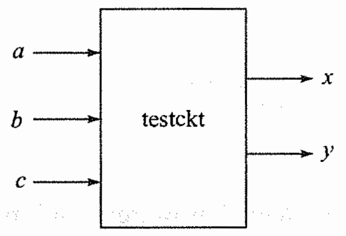
In this section, we introduce an interesting development in the field of hardware design. This is textual description of a digital circuit. Though we have already described hardware, can there be a language which is more crisp and more importantly, machine-readable? The advantage of course, is to be able to (i) describe a large complex design requiring hundreds of logic gates in a convenient manner, in a smaller space, (ii) use software test-bench to detect functional error, if any, and correct it (called *simulation*) and finally, (iii) get hardware implementation details (called *synthesis*). Hardware Description Language, more popular with its acronym HDL is an answer for that.

Currently, there are two widely used HDLs—Verilog and VHDL (Very high speed integrated circuit Hardware Description Language). Verilog is considered simpler of the two and is more popular. However, both share lot of common features and it is not too difficult to switch from one to the other. In this book, we'll deal with Verilog and shall discuss it over a span of number of chapters by introducing features relevant to that chapter. We expect by the time you finish Chapter 11, you'll have reasonable knowledge about HDL to deal with any digital logic design problem. We discuss target hardware devices on which HDL code can be directly exported in Section 13.6 of Chapter 13.

Verilog HDL

Verilog as a hardware description language has a small history. Introduced in 1980, primarily as a simulation and verification tool by Gateway Design Automation, it was later acquired by Cadence Data Systems. Put to public domain in 1990, it gained popularity and is now controlled by a group of companies and universities, called Open Verilog International. The reader with an exposure to any programming language like C will find it relatively easier to learn Verilog or any HDL.

Describing Input/Output In any digital circuit, we find there are a set of inputs and a set of outputs. Often termed as *ports*, the relationship between these input and outputs are explained within the digital circuit. To design any circuit that has say, three inputs *a*, *b*, *c* and two outputs say, *x*, *y* as shown in Fig. 2.37 the corresponding Verilog code can be written as shown next.



```
module testckt(x,y,a,b,c); //module name with port list

input  a,b,c; //defines input ports
output x,y; //defines output ports

//module body begins next describing logic relation
.

.
//module body ends
endmodule
```

Fig. 2.37 **Input/output definition in Verilog HDL for logic circuit described within black-box testckt**

Note that, **module** and **endmodule** written in bold are keywords for Verilog. A module describes a design entity with a name or identifier selected by user (here, testckt) followed by input output port list. This entity if used by another then arguments (i.e. ports) are to be passed in the same order as it appears here. The symbol '//' is used to put comments and improve readability for a human but not used by the machine, i.e. compiler. The module body describes the logic within the black box which acts on the inputs *a*, *b*, *c* and generates output *x*, *y*. Observe, where semicolon ';' is used and where not to end a statement, e.g. *endmodule* in above code does not end with semicolon.

Writing Module Body There are three different models of writing module body in Verilog HDL. Each one has its own advantage and suited for certain kind of design. We start with structural model by example of two-input OR gate described in Fig. 2.4a.

```
module or_gate(A,B,Y);
input A,B;   // defines two input port
output Y;    // defines one output port
or g1(Y,A,B);  /*Gate declaration with predefined keyword or representing
                 logic OR, g1 is optional user defined gate identifier */
endmodule
```

Verilog supports predefined gate level primitives such as **and, or, not, nand, nor, xor, xnor** etc. The syntax followed above can be extended to other gates and for 4 input OR gate it is as given next,

or (output, input 1, input 2, input 3, input 4)

For NOT gate, **not** (output, input)

Note that, Verilog can take up to 12 inputs for logic gates. Comments when extends to next line is written within /* ..... */. Identifiers in Verilog are case sensitive, begin with a letter or underscore and can be of any length.

Let us now look at description of a logic circuit shown in Fig. 2.17a that has 4 inputs and 1 output. The inputs are fed to two 2-input AND gate. AND gate outputs are fed to a 2-input OR gate to generate final output. The verilog code for this is given below. Note that, we define two intermediate variables and_op1 and and_op2 representing two AND gate outputs through keyword **wire**. Wire represents a physical wire in a circuit.

```
module fig2_24a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  wire and_op1, and_op2;  // internal connections
  and g1(and_op1,A,B); // g1 represents upper AND gate
  and g2(and_op2,C,D); // g2 represents lower AND gate
  or g3(Y,and_op1,and_op2); // g3 represents the OR gate
endmodule
```

One can see that structural model tries to replicate graphical layout design of a logic circuit. It does not matter if **or** statement in above example is written before **and** statements. This is as if one draws or connects the OR gate first on a design board and then the AND gates.

Example 2.16 Consider, the black box testckt of Fig. 2.38 has following logic circuit in it. Give Verilog structural code for the same.

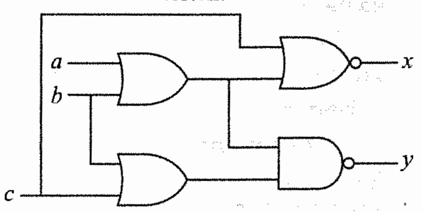


Fig. 2.38 **Logic circuit for Example 2.16**

Solution The code from the above discussion can be written as follows.

```
module testckt(a,b,c,x,y);
 input a,b,c;
 output x,y;
 wire or_op1, or_op2; /* internal connections, outputs of upper and
    lower OR gates respectively */
 or g1(or_op1,a,b); // g1 represents upper OR gate
 or g2(or_op2,b,c); // g2 represents lower OR gate
 nor g3(x,c,or_op1); // g3 represents the NOR gate
 nand g4(y,or_op1,or_op2); // g4 represents the NAND gate
endmodule
```

Preparation of Test Bench We shall discuss data flow model and behavioral model of Verilog VHDL in subsequent chapters. But, before we wind up this chapter let us see how to prepare a test bench in Verilog to simulate a digital circuit. For those of you with no programming background, this may appear little difficult. We could have postponed this discussion to a later chapter, but this gives you a feel of how simulation works or how a circuit you design can be tested. More clarity is assured as you go through discussions of subsequent chapters.

We take up the example of simulating a simple OR gate (Fig. 2.4a) for which Verilog code is already described. The test bench, creates an input in the form of a timing waveform and passes this to OR gate module through a function or procedural call (passing arguments in proper order). To generate timing waveform we use time delay available in Verilog in the form of *#n* where *n* denotes a number in decimal that gives delay in nanosecond. Input values to a variable can be provided through syntax *m'tn* where *m* represents number of digits, *t* represents type of number and *n* represents value to be provided.

The test bench used here generates all possible combinations of two inputs AB as 00,01,10 and 11 but at an interval of 20 ns. Note that, we have provided a 20 ns gate delay with **or** statement by #(20). All practical logic circuit comes with finite gate delay, i.e. output changes according to input after certain time. To change the gate delay to 10 ns we should write #(10) in **or** statement. The keyword **reg** is used to hold value of a data object in a procedural assignment. The keyword **initial** ensures sequential execution of codes following it, but once. We'll learn another keyword **always** in later chapter, which too is used for sequential execution but for infinite time.

```
 module testor; //Simulation module given a name testor
 reg A,B;       //Storage of data for passing it to module or_gate
 wire x;
 or_gate org(A,B,x); //circuit is instantiated with the name, or_gate
 initial    // Starts simulation
   begin   /* Input is generated to test the circuit through following
     statements, simulation begins*/
 A=1'b0;B=1'b0; /* 1'b0 signifies on binary digit with a value 0, AB is
assigned 00*/
    #20              // Delay of 20 ns
    A=1'b0;B=1'b1;  //After 20ns AB=01
```

```
        #20          // Another Delay of 20 ns
        A=1'b1;B=1'b0; //After 40ns from start point  AB=10
        #20
        A=1'b1;B=1'b1; //After 60ns from start point  AB=11
        #20 $finish;// the simulation terminates after 80 ns
    end
endmodule

module or_gate(A,B,x); // OR gate used as a procedure in simulation
    input A,B;      // defines two input port
    output x;       // defines one output port
    or #(20) g1(x,A,B);  /*Gate declaration with a gate delay of 20ns,
     output is effected after 20 ns*/
endmodule
```

Execution of above Verilog code generates following timing diagram. One can see that input AB, given by testor.A and testor.B (testor is module name of the test bench) is taking value 00,01,10,11 as expected and retain them for 20 ns. Output of OR gate, testor.x changes according to input but after a delay of 20 ns. For first 20 ns, OR gate output is unknown as it needs 20ns (gate delay) to respond to first appearance of input logic at A,B. Note that Verilog, in general offers four logic values in simulation 0,1, unknown (or **x**) and high impedance (or **z**). Unknown value is exhibited when input is ambiguous and high impedance is shown when a wire by mistake is left unconnected or the circuit is following tri-state logic (Chapter 14, Section 6).

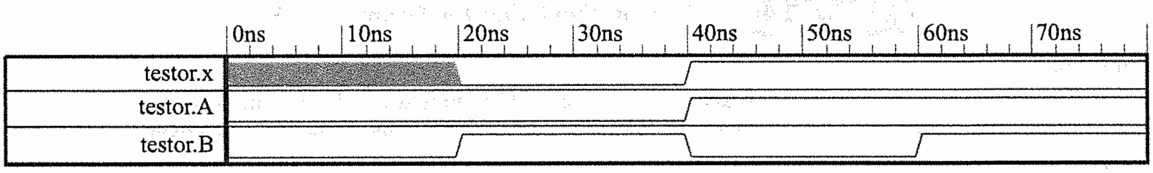


Fig. 2.39 **Verilog simulation of 2 input OR gate with 20ns gate delay**

Example 2.17 Write the statements between **begin** and **end** of a test bench for circuit described in Example 2.16 with 50 ns holding time of each input combination.

Solution Since the circuit has three inputs we need $2^3 = 8$ different combinations of inputs. Thus the statements would look like as follows:

```
begin   // Input is generated to test the circuit through following
            statements, simulation begins
    a=1'b0;b=1'b0;c=1'b0; //  ABC is assigned 000
    #50                    // Delay of 50 ns
    a=1'b0;b=1'b0;c=1'b1; //  ABC is assigned 001
    #50                    // Delay of 50 ns
    a=1'b0;b=1'b1;c=1'b0; //  ABC is assigned 010
    #50                     // Delay of 50 ns
    a=1'b0;b=1'b1;c=1'b1; //  ABC is assigned 011
    #50                     // Delay of 50 ns
    a=1'b1;b=1'b0;c=1'b0; //  ABC is assigned 100
    #50                     // Delay of 50 ns
    a=1'b1;b=1'b0;c=1'b1; //  ABC is assigned 101
    #50                     // Delay of 50 ns
    a=1'b1;b=1'b1;c=1'b0; //  ABC is assigned 110
    #50                     // Delay of 50 ns
    a=1'b1;b=1'b1;c=1'b1; //  ABC is assigned 111
    #50 $finish;// the simulation terminates after 400 ns
  end
```

Example 2.18 The following timing diagram is generated by simulation of Verilog code for 2-input, 1-output device where A and B are input and x is output. Can you (i) estimate the gate delay and (ii) identify the logic?

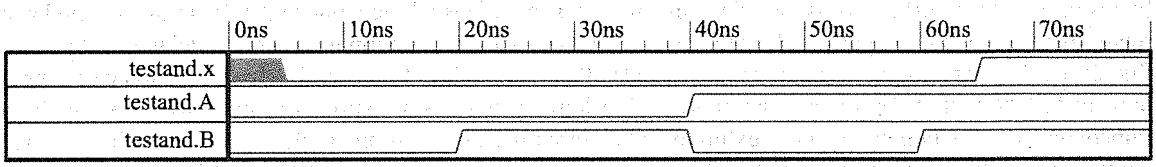


Fig. 2.40 Verilog simulation for Example 2.18

Solution

(i) The unknown value of the output is approximately half of 10 ns time scale. Hence, gate delay is 5ns.

(ii) Output goes HIGH when both the inputs go high after a delay of 5 ns. Hence, the logic underlying is AND.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Realize $Y = AB + \overline{C}$ using only one type of gate.

Solution The function to be realized involves AND, NOT and OR operations. We can realize this expression using universal logic gate.

In Method-1, we realize it using NOR gate. We realize individual logic operations like AND, NOT and OR as depicted in Fig. 2.21. The solution is given in Fig. 2.41.

In Method-2, we realize it using NAND gate. We realize individual logic operations like AND, NOT and OR as depicted in Fig. 2.27. The solution is given in Fig. 2.42.

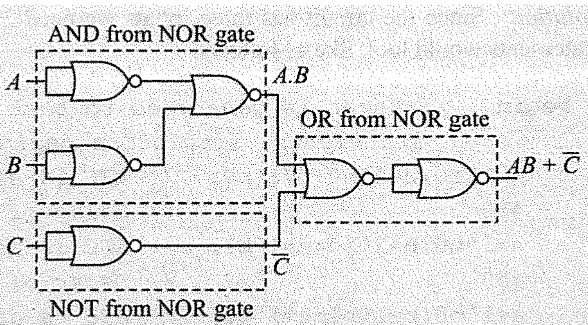


Fig. 2.41 Realization of $Y = AB + \overline{C}$ using only NOR gate

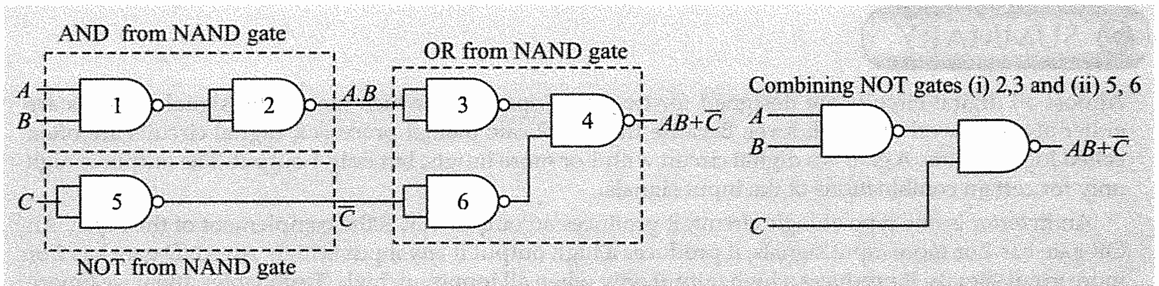


Fig. 2.42 **Realization of $Y = AB + \overline{C}$ using only NAND gate**

Note that the final logic circuit achieved involves only 2 NAND gates as two NOT gates can easily be replaced by directly drawing connection from uncomplemented input.

In Method-3, we again get a solution using NOR gate but with a different approach where we use De Morgan's first (Eq. 2.1) and second (Eq. 2.2) theorem. The objective is to have only NOR relation throughout.

Given,

$$
\begin{aligned}
Y &= AB + \overline{C} \\
&= \overline{\overline{AB + \overline{C}}} && \text{double complement} \\
&= \overline{\overline{AB}.C} && \text{from Eq. 2.1} \\
&= \overline{(\overline{A} + \overline{B}).C} && \text{from Eq. 2.2} \\
&= \overline{(\overline{A} + \overline{B})} + \overline{C} && \text{from Eq. 2.2} \\
&= \overline{\overline{\overline{(\overline{A} + \overline{B})} + \overline{C}}} && \text{double complement}
\end{aligned}
$$

Thus, two NOR operations $(\overline{A} + \overline{B})'$ and $((\overline{A} + \overline{B})' + \overline{C})'$ [$'$represents NOT operation such that $\overline{X} = X'$] require two NOR gates, and four inversions $\overline{A}$, $\overline{B}$, $\overline{C}$ and $(((\overline{A} + \overline{B})' + \overline{C})')'$ require four NOR gates totaling six NOR gates for realization. You will find that the solution is similar to what is achieved by Method-1.

In Method-4, similar to Method-3 we use De Morgan's theorems to get a solution but only with NAND gates. The objective is to have only NAND relation throughout. We need to go only up to step two of Method-3 analysis to get an all NAND realization.

Given,

$$
\begin{aligned}
Y &= AB + \overline{C} \\
&= \overline{\overline{AB + \overline{C}}} && \text{double complement} \\
&= \overline{\overline{AB}.C} && \text{from Eq. 2.1}
\end{aligned}
$$

Thus, we need two NAND gates—one for $(A.B)'$ and another for $((A.B)'.C)'$ realization. We find that the final logic circuit of Method-2 is similar to what is obtained here.

SUMMARY

Almost all digital circuits are designed for two-state operation, which means the signal voltages are either at a low level or a high level. Because they duplicate mental processes, digital circuits are often called logic circuits. A gate is a digital circuit with 1 or more inputs, but only 1 output. The output is high only for certain combinations of the input signals.

An inverter is one type of logic circuit, it produces an output that is the complement of the input. An OR gate has 2 or more input signals; it produces a high output if any input is high. An AND gate has 2 or more input signals; it produces a high output only when all inputs are high. Truth tables often use binary 0 for the low state and binary 1 for the high state. The number of entries in a truth table equals 2^n, where n is the number of input signals.

The overbar is the algebraic symbol for the NOT operation, the plus sign is the symbol for the OR operation, and the times sign is the symbol for the AND operation. Since the Boolean operators are codes for the OR gate, AND gate, and inverter, we can use Boolean algebra to analyze digital circuits. An AND-OR circuit always produces a sum-of-products equation, while the OR-AND circuit results in a product-of-sums equation.

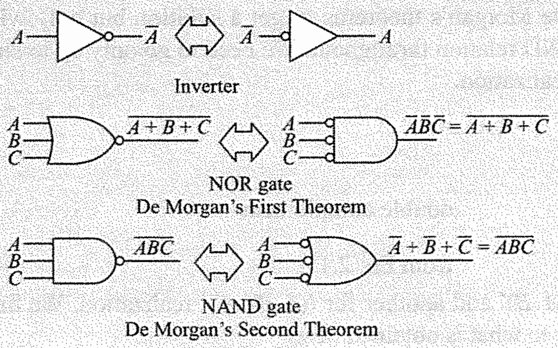
The NOR gate is equivalent to an OR gate followed by an inverter. De Morgan's first theorem tells us that a NOR gate is equivalent to a bubbled AND gate. Because of De Morgan's first theorem, a NOR-NOR circuit is equivalent to an OR-AND circuit.

The NAND gate represents an AND gate followed by an inverter. De Morgan's second theorem says the NAND gate is equivalent to a bubbled OR gate. Furthermore, a NAND-NAND circuit is equivalent to an AND-OR circuit. The NAND gate is the backbone of the 7400 TTL series because most devices in this family are derived from the NAND-gate design. The NAND gate is a universal gate since any logic circuit can be built with NAND gates only.

With positive logic, binary 1 represents high voltage and binary 0 represents low voltage. Also, positive true stands for high voltage and positive false for low voltage. With negative logic, binary 1 stands for low voltage and binary 0 for high voltage. In this system, negative true is equivalent to low voltage and negative false to high voltage.

With assertion-level logic, we draw gates and other devices with bubbled pins for active-low signals. Also, signal voltages are labeled with abbreviations of statements that describe circuit behavior. An overbar is used on a label whenever the signal is active-low.

Figure 2.43 shows three sets of equivalent gates. Changing from one to the other is accomplished by adding or deleting bubbles, and changing AND to OR or OR to AND. The NOR gate and NAND gate equivalents illustrate De Morgan's first and second theorems.



A
$\overline{A}$
$\overline{A}$
A
Inverter
A
B
C
$\overline{A+B+C}$
$\overline{A}\,\overline{B}\,\overline{C} = \overline{A+B+C}$
NOR gate
De Morgan's First Theorem
$\overline{ABC}$
$\overline{A}+\overline{B}+\overline{C} = \overline{ABC}$
NAND gate
De Morgan's Second Theorem

Fig. 2.43

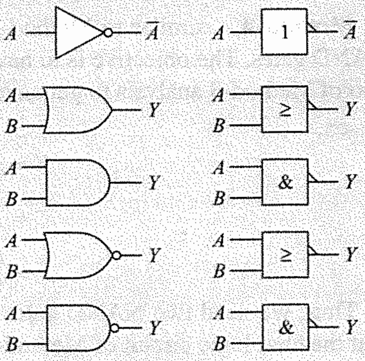


Fig. 2.44

Figure 2.44 shows the additional logic symbols for five basic gates along with the corresponding IEEE rectangular symbols.

GLOSSARY

- ***active-low*** Active-low refers to the concept in which a signal must be low to cause something to happen or to indicate that something has happened.
- ***AND gate*** A gate with 2 or more inputs. The output is high only when all inputs are high.
- ***assert*** To activate. If an input line has a bubble on it, you assert the input by making it low. If there is no bubble, you assert the input by making it high.
- ***De Morgan's first theorem*** In words, the complement of a logical sum equals the logical product of the complements. In terms of circuits, a NOR gate equals a bubbled AND gate.
- ***De Morgan's second theorem*** In words, the complement of a logical product equals the logical sum of the complements. In terms of circuits, a NAND gate is equivalent to a bubbled OR gate.
- ***gate*** A digital circuit with one or more input voltages but only one output voltage.
- ***inverter*** A gate with only one input and a complemented output.
- ***logic circuit*** A digital circuit, a switching circuit, or any kind of two-state circuit that duplicates mental processes.
- ***negative true*** A signal is negative true when the voltage is low.
- ***OR gate*** A gate with two or more inputs. The output is high when any input is high.
- ***positive true*** A signal is positive true when the voltage is high.
- ***product-of-sums equation*** A Boolean equation that is the logical product of logical sums. This type of equation applies to an OR-AND circuit.
- ***sum-of-products equation*** A Boolean equation that is the logical sum of logical products. This type of equation applies to an AND-OR circuit.
- ***timing diagram*** A picture that shows the input-output waveforms of a logic circuit.
- ***truth table*** A table that shows all of the input-output possibilities of a logic circuit.
- ***two-state operation*** The use of only two points on the load line of a device, resulting in all voltages being either low or high.

PROBLEMS

Section 2.1

2.1 What is the output in Fig. 2.45a if the input is low? The output if the input is high?

2.2 The input is low in Fig. 2.45b. Is the output low or high? Is the circuit equivalent to an inverter? If you cascade an odd number of inverters, what kind of gate is the overall circuit equivalent to?

2.3 Construct the truth table for Fig. 2.46a. After you are finished, discuss the relation between

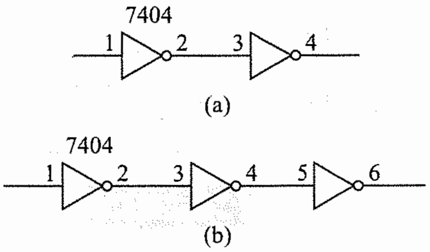


Fig. 2.45

the circuit of Fig. 2.46a and a 3-input OR gate.

2.4 Construct the truth table for Fig. 2.46b.

2.5 Construct the truth table for Fig. 2.46c.

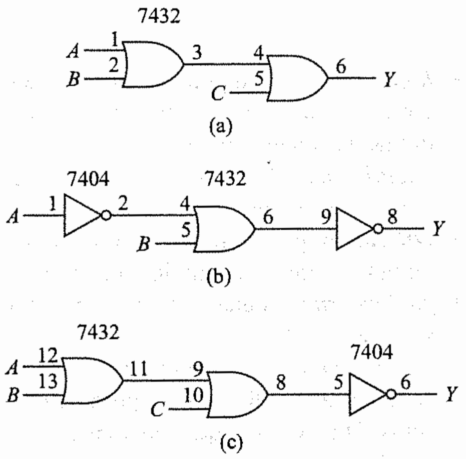


Fig. 2.46

2.6 The circuit of Fig. 2.46b has trouble. Figure 2.47 shows its timing diagram. Which of the following is the trouble:

a. Input inverter acts like OR gate.
b. Pin 6 is shorted to ground.
c. AND gate is used instead of OR gate.
d. Output inverter is faulty.

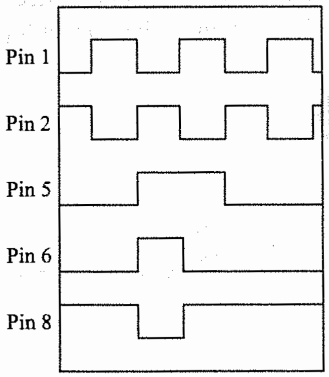


Fig. 2.47

2.7 Construct the truth table for Fig. 2.48a. Then discuss the relation between the circuit of Fig. 2.48a and a 3-input AND gate.

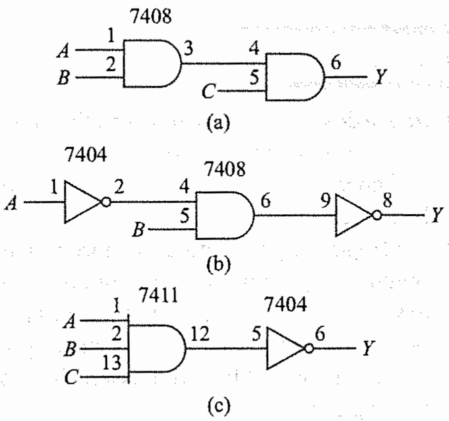


Fig. 2.48

2.8 Construct the truth table of Fig. 2.48b.

2.9 Construct the truth table for Fig. 2.48c.

2.10 Assume the circuit of Fig. 2.48b has trouble. If Fig. 2.49 is the timing diagram, which of the following is the trouble:

a. Input inverter is shorted.
b. OR gate is used instead of AND gate
c. Pin 6 is shorted to ground.
d. Pin 8 is shorted to +5 V.

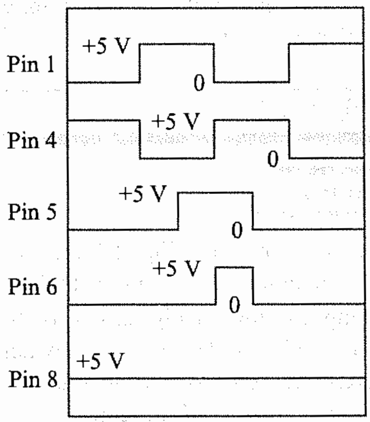


Fig. 2.49

2.11 What is the Boolean equation for the output of Fig. 2.46a?
For Fig. 2.46b? For Fig. 2.46c?

2.12 Draw the logic circuit whose Boolean equation is

$$Y = \overline{\overline{A + B} + \overline{C}}$$

2.13 Use the 7404 and the 7432 with pin numbers.
What is the Boolean equation for the output of Fig. 2.46a?

2.14 For Fig. 2.46b? For Fig. 2.46c?
Draw the logic circuit described by

$$Y = (\overline{ABC})\overline{D}$$

2.15 Use a 7404, 7408, and 7411 with pin numbers.
Draw the logic circuit given by this Boolean equation:

$$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + \overline{A}B\overline{C}$$

Use the following devices with pin numbers: 7404, 7411, and 7432.

Section 2.2

2.16 Construct the truth table for the 3-input NOR gate of Fig. 2.50a.

2.17 Construct the truth table for the 4-input NOR gate of Fig. 2.50b.

2.18 Show an equivalent NOR-NOR circuit for Fig. 2.50c. Use the 7402 and the 7427 with pin numbers.

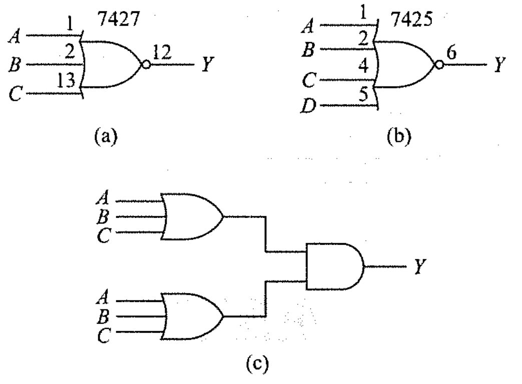


Fig. 2.50

2.19 The circuit of Fig. 2.50c has trouble. If output Y is stuck in the high state, which of the following is the trouble:

a. Either input pin of the AND gate is shorted to ground.
b. Any input of either OR gate is shorted to ground.
c. Any input of either OR gate is shorted to a high voltage.
d. AND gate is defective.

2.20 Construct the truth table for the 4-input NAND gate of Fig. 2.51a.

2.21 The inputs are $A_0, A_1, A_2, \ldots, A_7$ in Fig. 2.51b. What is the Boolean equation for the input of the NAND gate?

2.22 Draw an equivalent NAND-NAND circuit for Fig. 2.51c. Use the 7420 and include pin numbers.

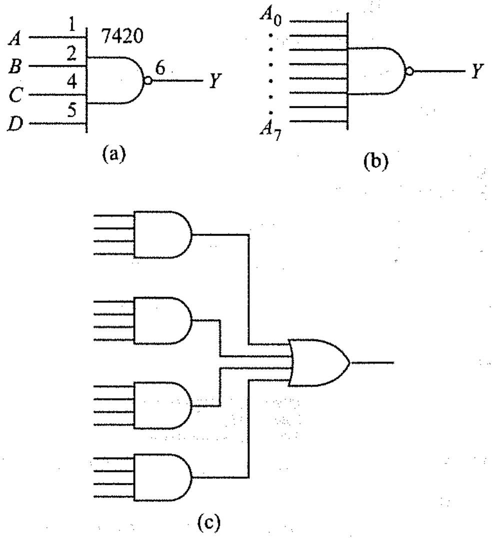


Fig. 2.51

2.23 Suppose the final output of Fig. 2.51c is stuck in the high state. Which of the following is the trouble?

a. Any input to the OR gate is shorted to a high voltage.
b. Any AND-gate input is shorted to ground.

c. Any AND-gate input is shorted to a high voltage.
d. One of the AND gates is defective because its output is always low.

Section 2.3

2.24 What is the output in Fig. 2.31c for these inputs?
a. $ABCD = 0000$ b. $ABCD = 0101$
c. $ABCD = 1100$ d. $ABCD = 1111$

2.25 Is the output Y of Fig. 2.52 low or high for these conditions?
a. Both switches open, A is low.
b. Both switches closed, A is high.
c. Left switch open, right switch closed, A is low.
d. Left switch closed, right switch open, A is high.

2.26 If all inputs are low in Fig. 2.34b, what is the output? If all inputs are high, what is the output?

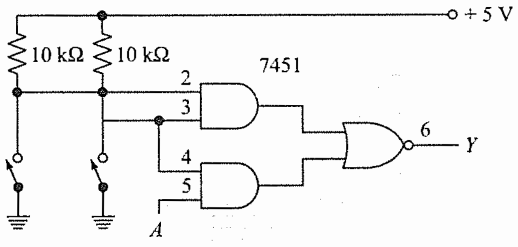


Fig. 2.52

2.27 What is the value of Y in Fig. 2.53 for each of these?
a. $ABCD = 0000$ b. $ABCD = 0101$
c. $ABCD = 1000$ d. $ABCD = 1111$

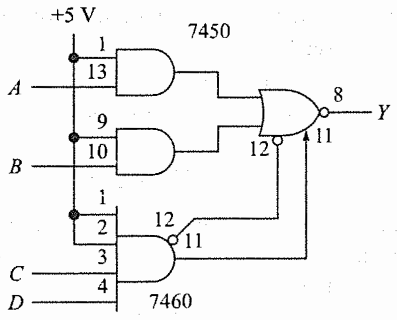


Fig. 2.53

Section 2.4

2.28 In Fig. 2.54, is each of the following an active-low or an active-high?
a. Pin 1 b. Pin 2
c. Pin 3 d. Pin 5
e. Pin 6

2.29 An 8085 microprocessor uses the following labels with assertion-level logic. Is each signal active-low or active-high?
a. HOLD b. $\overline{\text{RESET IN}}$
c. $\overline{\text{RD}}$ d. $\overline{\text{WR}}$
e. ALE f. INTR
g. $\overline{\text{INTA}}$

2.30 When switch B of Fig. 2.54 is closed, is pin 6 high or low? For this condition, is B CLOSED negative true or negative false?

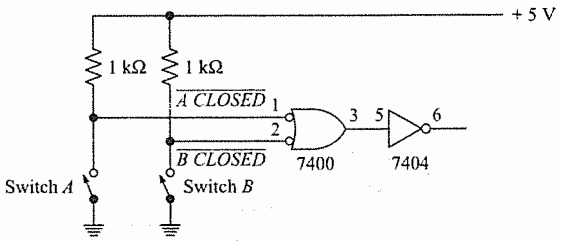


Fig. 2.54

LABORATORY EXPERIMENT

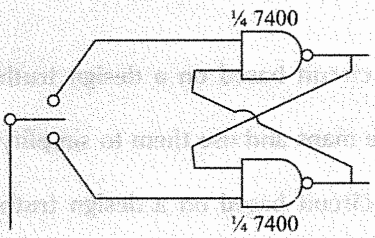
AIM: The aim of this experiment is to study basic NAND gate and implement a bounce-free switch using basic gates.

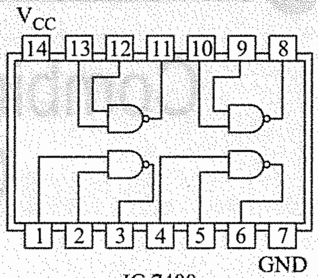
Theory: The NAND gate implements the logic

$Y = (A.B.C...)'$

where A, B, C, ... are inputs and Y is output.

The mechanical switches used for electrical connection go through several make-break situations before resting in a particular position. For digital circuits, this could amount to a series of HIGH-LOW which is particularly detrimental to sequential logic circuit. The following circuit presents a NAND based debounce switch.





IC 7400

Apparatus: +5 Volt dc power supply, multimeter, bread board, and oscilloscope

Work element: IC 7400 is a quad 2-input NAND gate. Study its pin configuration and verify the NAND truth table. Study the debounce switch circuit and find out the principle behind its working. Use two NAND gates of IC 7400 and wire it as shown in the diagram. A wire may be used as a switch, the other side of which is connected to a dc power supply or Ground. Observe voltage level at the oscilloscope, at various points of the circuit when switching is done at the input side. Discuss the result. Design and implement a NOR (IC 7402) gate based debounce switch.

Answers to Self-tests

1. positive
2. OR
3. AND
4. $Y = \overline{A}$
5. $Y = A + B$
6. $Y = A \cdot B = AB$
7. $Y = \overline{A + B}$
8. $\overline{A + B} = \overline{A} \cdot \overline{B}$
9. ≥
10. $Y = \overline{A \cdot B}$
11. $\overline{A \cdot B} = \overline{A} + \overline{B}$
12. &
13. There are two AND gates at the input.
14. It is used to increase the number of input AND gates.
15. Converse of positive logic. Here binary 0 stands for high voltage and binary 1 stands for low voltage.
16. It means drawing logic symbols to indicate the action of each signal. If the signal causes something to happen when low, it is drawn with a bubble; this is an active-low signal. If a signal causes something to happen when high, it is drawn without a bubble; this is an active-high signal.

Combinational Logic Circuits

3

OBJECTIVES

- Demonstrate the ability to use basic Boolean laws.
- Use the sum-of-products method to design a logic circuit based on a design truth table.
- Be able to make Karnaugh maps and Entered variable maps and use them to simplify Boolean expressions.
- Use the product-of-sums method to design a logic circuit based on a design truth table.
- Use Quine-McClusky tabular method for logic simplification
- Analyze hazards in logic circuit and provide solution for them.

This chapter discusses Boolean algebra and several simplification techniques. After learning the laws and theorems of Boolean algebra, you can rearrange Boolean equations to arrive at simpler logic circuits. An alternative method of simplification is based on the Karnaugh map. In this approach, geometric rather than algebraic techniques are used to simplify logic circuits. Quine-McClusky tabular method provides a more systematic reduction technique, which is preferred when a large number of variables are in consideration.

There are two fundamental approaches in logic design: the sum-of-products method and the product-of-sums method. Either method produces a logic circuit corresponding to a given truth table. The sum-of-products solution results in an AND-OR or NAND-NAND network, while the product-of-sums solution results in an OR-AND or NOR-NOR network. Either can be used, although a designer usually selects the simpler circuit because it costs less and is more reliable. A practical logic circuit can show hazard due to finite propagation delay involved in each logic gate. This gives glitches or shows multiple transitions at the output. This chapter discusses different types of hazards and ways to prevent them.

3.1 BOOLEAN LAWS AND THEOREMS

You should know enough Boolean algebra to make obvious simplifications. What follows is a discussion of the basic laws and theorems of Boolean algebra. Some of them will look familiar from ordinary algebra but others will be distinctly new.

Basic Laws

The commutative laws are

$$A + B = B + A \tag{3.1}$$
$$AB = BA \tag{3.2}$$

These two equations indicate that the order of a logical operation is unimportant because the same answer is arrived at either way. As far as logic circuits are concerned. Figure 3.la shows how to visualize Eq. (3.1). All it amounts to is realizing that the inputs to an OR gate can be transposed without changing the output. Likewise, Fig. 3.1b is a graphical equivalent for Eq. (3.2).

The associative laws are

$$A + (B + C) = (A + B) + C \tag{3.3}$$
$$A(BC) = (AB)C \tag{3.4}$$

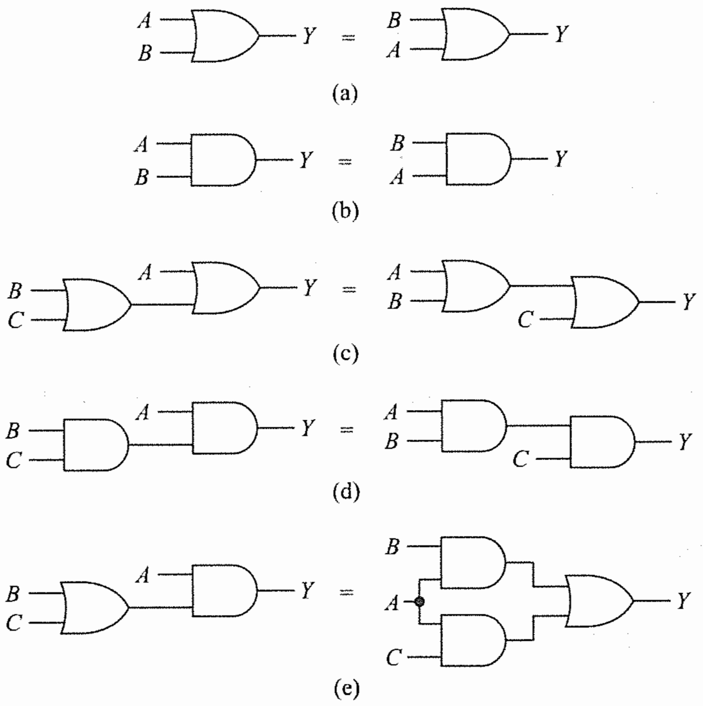


Fig. 3.1 Commutative, associative, and distributive laws

These laws show that the order of combining variables has no effect on the final answer. In terms of logic circuits, Fig. 3.1c illustrates Eq. (3.3), while Fig. 3.1d represents Eq. (3.4).

The distributive law is

$$A(B + C) = AB + AC \tag{3.5}$$

This law is easy to remember because it is identical to ordinary algebra. Figure 3.1e shows the corresponding logic equivalence. The distributive law gives you a hint about the value of Boolean algebra. If you can rearrange a Boolean expression, the corresponding logic circuit may be simpler.

The first five laws present no difficulties because they are identical to ordinary algebra. You can use these laws to simplify complicated Boolean expressions and arrive at simpler logic circuits. But before you begin, you have to learn other Boolean laws and theorems.

OR Operations

The next four Boolean relations are about OR operations. Here is the first:

$$A + 0 = A \tag{3.6}$$

This says that a variable ORed with 0 equals the variable. If you think about it, makes perfect sense. When A is 0,

$$0 + 0 = 0$$

And when A is 1,.

$$1 + 0 = 1$$

In either case, Eq. (3.6) is true.

Another Boolean relation is

$$A + A = A \tag{3.7}$$

Again, you can see right through this by substituting the two possible values of A. First when $A = 0$, Eq. (3.7) gives

$$0 + 0 = 0$$

which is true. Next, $A = 1$ results in

$$1 + 1 = 1$$

which is also true because 1 ORed with 1 produces 1. Therefore, any variable ORed with itself equals the variable.

Another Boolean rule worth knowing is

$$A + 1 = 1 \tag{3.8}$$

Why is this valid? When $A = 0$, Eq. (3.8) gives

$$0 + 1 = 1$$

which is true. Also. $A = 1$ gives

$$1 + 1 = 1$$

This is correct because the plus sign implies OR addition, not ordinary addition. In summary, Eq. (3.8) says this, if one input to an OR gate is high, the output is high no matter what the other input.

Finally, we have

$$A + \overline{A} = 1 \tag{3.9}$$

You should see this in a flash. If A is 0, $\overline{A}$ is 1 and the equation is true. Conversely, if A is 1, $\overline{A}$ is 0 and the equation still agrees. In short, a variable ORed with its complement always equals 1.

AND Operations

Here are three AND relations

$$A \cdot 1 = A \tag{3.10}$$
$$A \cdot A = A \tag{3.11}$$
$$A \cdot 0 = 0 \tag{3.12}$$

When A is 0, all the foregoing are true. Likewise, when A is 1, each is true. Therefore, the three equations are valid and can be used to simplify Boolean equations.
One more AND formula is

$$A \cdot \overline{A} = 0 \tag{3.13}$$

This one is easy to understand because you get either

$$0 \cdot 1 = 0$$

or

$$1 \cdot 0 = 0$$

for the two possible values of A. In words, Eq. (3.13) indicates that a variable ANDed with its complement always equals zero.

Double Inversion and De Morgan's Theorems

The *double-inversion rule* is

$$\overline{\overline{A}} = A \tag{3.14}$$

which shows that the double complement of a variable equals the variable. Finally, there are the De Morgan theorems discussed in Chapter 2:

$$\overline{A + B} = \overline{A}\,\overline{B} \tag{3.15}$$
$$\overline{AB} = \overline{A} + \overline{B} \tag{3.16}$$

You already know how important these are. The first says a NOR gate and a bubbled AND gate are equivalent. The second says a NAND gate and a bubbled OR gate are equivalent.

Duality Theorem

The *duality theorem* is one of those elegant theorems proved in advanced mathematics. We will state the theorem without proof. Here is what the duality theorem says. Starting with a Boolean relation, you can derive another Boolean relation by

1. Changing each OR sign to an AND sign
2. Changing each AND sign to an OR sign
3. Complementing any 0 or 1 appearing in the expression

For instance, Eq. (3.6) says that

$$A + 0 = A$$

The dual relation is

$$A \cdot 1 = A$$

This dual property is obtained by changing the OR sign to an AND sign, and by complementing the 0 to get a 1.

The duality theorem is useful because it sometimes produces a new Boolean relation. For example, Eq. (3.5) states that

$$A(B + C) = AB + AC$$

By changing each OR and AND operation, we get the dual relation

$$A + BC = (A + B)(A + C) \tag{3.17}$$

This is new, not previously discussed. (If you want to prove it, construct the truth table for each side of the equation. The truth tables will be identical, which means the Boolean relation is true.)

Covering and Combination

The covering rule, where one term covers the condition of the other term so that the other term becomes redundant, can be represented in dual form as

$$A + AB = A \tag{3.18}$$

and

$$A(A + B) = A \tag{3.19}$$

The above can be easily proved from basic laws because,

$$A + AB = A \cdot 1 + AB = A(1 + B) = A \cdot 1 = A$$

and

$$A(A + B) = A \cdot A + AB = A + AB = A$$

The combining rules are,

$$AB + A\overline{B} = A \tag{3.20}$$

and in its dual form

$$(A + B)(A + \overline{B}) = A \tag{3.21}$$

Eq. (3.20) can easily be proved as $B + \overline{B} = 1$

Expanding left hand side of Eq. (3.21)

$$\begin{aligned} A \cdot A + A \cdot B + A \cdot \overline{B} + B \cdot \overline{B} &= A + A(B + \overline{B}) + 0 \\ &= A + A \cdot 1 = A + A = A = \text{right hand side} \end{aligned}$$

Consensus Theorem

The *consensus theorem* finds a redundant term which is a *consensus of two other terms*. The idea is that if the consensus term is true, then any of the other two terms is true and thus it becomes redundant. This can be expressed in dual form as

$$AB + \overline{A}C + BC = AB + \overline{A}C \tag{3.22}$$

$$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C) \tag{3.23}$$

In the first expression, BC is the consensus term and thus redundant. This is because if $BC = 1$, then both $B = 1$ and $C = 1$ and any of the other two terms AB or $\overline{A}C$ must be one as either $A = 1$ or $\overline{A} = 1$. Similarly, in the second expression, $(B + C)$ is the consensus term and if this term is 0 then both $B = 0$ and $C = 0$. This makes one of the other two sum terms 0 as either $A = 0$ or $\overline{A} = 0$.

SUMMARY OF BOOLEAN RELATIONS

For future reference, here are some Boolean relations and their duals:

$A + B = B + A$	$AB = BA$
$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
$A + 0 = A$	$A \cdot 1 = A$
$A + 1 = 1$	$A \cdot 0 = 0$
$A + A = A$	$A \cdot A = A$
$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
$\overline{\overline{A}} = A$	$\overline{A} = A$
$\overline{A + B} = \overline{A}\,\overline{B}$	$\overline{AB} = \overline{A} + \overline{B}$
$A + AB = A$	$A(A + B) = A$
$A + \overline{A}B = A + B$	$A(\overline{A} + B) = AB$
$AB + A\overline{B} = A$	$(A + B)(A + \overline{B}) = A$
$AB + \overline{A}C + BC = AB + \overline{A}C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$

Example 3.1 Prove that, $A(A' + C)(A'B + C)(A'BC + C') = 0$

Solution

$$\begin{aligned}
\text{LHS} &= (AA' + AC)(A'B + C)(A'BC + C') && \text{: distributive law}\\
&= AC(A'B + C)(A'BC + C') && \text{: since, } XX' = 0\\
&= (AC \cdot A'B + AC \cdot C)(A'BC + C') && \text{: distributive law}\\
&= AC(A'BC + C') && \text{: since, } XX' = 0\\
&= AC \cdot A'BC + AC \cdot C' && \text{: distributive law}\\
&= 0 = \text{RHS} && \text{: since, } XX' = 0
\end{aligned}$$

Example 3.2 Simplify, $Y = (A + B)(A'(B' + C'))' + A'(B + C)$

Solution

$$\begin{aligned}
Y &= (A + B)((A + (B' + C')') + A'(B + C) && \text{: De Morgan's theorem}\\
&= (A + B)(A + BC) + A'(B + C) && \text{: De Morgan's theorem}\\
&= (AA + ABC + AB + BBC) + A'(B + C)\\
&= (A + AB + ABC + BC) + A'(B + C)\\
&= A(1 + B + BC) + BC + A'(B + C)\\
&= A + BC + A'(B + C)\\
&= (A + A'(B + C)) + BC\\
&= A + B + C + BC\\
&= A + B + C(1 + B)\\
&= A + B + C
\end{aligned}$$

Example 3.3 A *logic clip* is a device that you can attach to a 14- or 16-pin DIP. This troubleshooting tool contains 16 light-emitting diodes (LEDs) that monitor the state of the pins. When a pin voltage is high, the corresponding LED lights up. If the pin voltage is low, the LED is dark.

Suppose you have built the circuit of Fig. 3.2a, but it doesn't work correctly. When you connect a logic clip to the 7408, you get the readings of Fig. 3.2b (a black circle means an LED is off, and a white one means it's on). When you connect the clip to the 7432, you get the indications of Fig. 3.2c. Which of the gates is faulty?

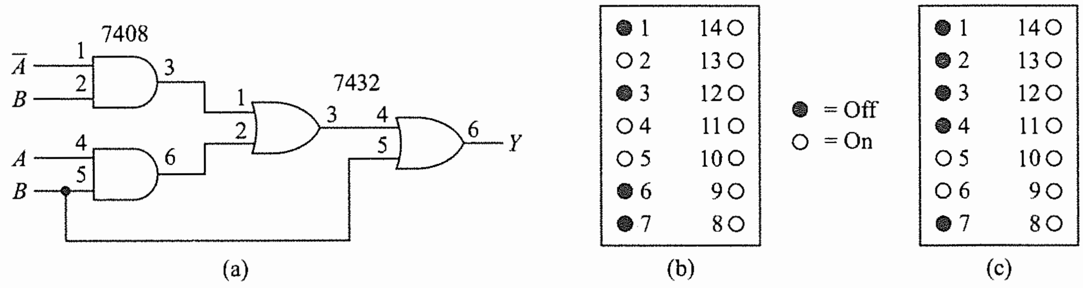


Fig. 3.2

Solution When you use a logic clip, all you have to do is look at the inputs and output to isolate a faulty gate. For instance, Fig. 3.2b applies to a 7408 (quad 2-input AND gate). The First AND gate (pins 1 to 3) is all right because

Pin 1—low
Pin 2—high
Pin 3—low

A 2-input AND gate is supposed to have a low output if any input is low.

The second AND gate (pins 4 to 6) is defective. Why? Because

Pin 4—high
Pin 5—high
Pin 6—low

Something is wrong with this AND gate because it produces a low output even though both inputs are high.

If you check Fig. 3.2c (the 7432), all OR gates are normal. For instance, the first OR gate (pins 1 to 3) is all right because it produces a low output when the 2 inputs are low. The second OR gate (pins 4 to 6) is working correctly since it produces a high output when 1 input is high.

SELF-TEST

1. All the rules for Boolean algebra are exactly the same as for ordinary algebra. (T or F)
2. Expand using the distributive law: $Y = A(B + C)$.
3. Simplify: $Y = \overline{A}Q + AQ$.

3.2 SUM-OF-PRODUCTS METHOD

Figure 3.3 shows the four possible ways to AND two input signals that are in complemented and uncomplemented form. These outputs are called *fundamental products*. Table 3.1 lists each fundamental product next to the input conditions producing a high output. For instance, $\overline{A}\,\overline{B}$ is high when A and B are low; $\overline{A}B$ is high when A is low and B is high; and so on. The fundamental products are also called *minterms*. Products $A'B'$, $A'B$, AB', AB are represented by m_0, m_1, m_2, and m_3 respectively. The suffix i of m_i comes from decimal equivalent of binary values (Table 3.1) that makes corresponding product term high.

Table 3.1 Fundamental Products for Two Inputs

A	B	*Fundamental Product*
0	0	$\overline{A}\,\overline{B}$
0	1	$\overline{A}B$
1	0	$A\overline{B}$
1	1	AB

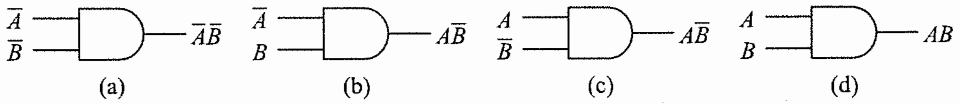


Fig. 3.3 ANDing two variables and their complements

The idea of fundamental products applies to three or more input variables. For example, assume three input variables: A, B, C and their complements. There are eight ways to AND three input variables and their complements resulting in fundamental products of

$$\overline{A}\,\overline{B}\,\overline{C}, \overline{A}\,\overline{B}C, \overline{A}B\overline{C}, \overline{A}BC, A\overline{B}\,\overline{C}, A\overline{B}C, AB\overline{C}, ABC$$

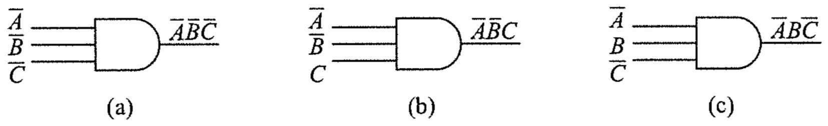


Fig. 3.4 Examples of ANDing three variables and their complements

The above three variable minterms can alternatively be represented by m_0, m_1, m_2, m_3, m_4, m_5, m_6, and m_7 respectively. Note that, for n variable problem there can be 2^n number of minterms. Figure 3.4a shows the first fundamental product, Fig. 3.4b the second, and Fig. 3.4c the third. (For practice, draw the gates for the remaining fundamental products.) for twice variable case.

Table 3.2 summarizes the fundamental products by listing each one next to the input condition that results in a high output. For instance, when $A = 1$, $B = 0$ and $C = 0$, the fundamental product results in an output of

$$Y = A\overline{B}\,\overline{C} = 1 \cdot \overline{0} \cdot \overline{0} = 1$$

Table 3.2 Fundamental Products for Three Inputs

A	B	C	*Fundamental Products*
0	0	0	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	$\overline{A}\,\overline{B}C$
0	1	0	$\overline{A}B\overline{C}$
0	1	1	$\overline{A}BC$
1	0	0	$A\overline{B}\,\overline{C}$
1	0	1	$A\overline{B}C$
1	1	0	$AB\overline{C}$
1	1	1	ABC

Sum-of-Products Equation

Here is how to get the sum-of-products solution, given a truth table like Table 3.3. What you have to do is locate each output 1 in the truth table and write down the fundamental product. For instance, the first output 1 appears for an input of $A = 0$, $B = 1$, and $C = 1$. The corresponding fundamental product is $\overline{A}BC$. The next output 1 appears for $A = 1$, $B = 0$, and $C = 1$. The corresponding fundamental product is $A\overline{B}C$. Continuing like this, you can identify all the fundamental products, as shown in Table 3.4. To get the sum-of-products equation, all you have to do is OR the fundamental products of Table 3.4:

$$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \tag{3.24}$$

Alternate representation of Table 3.3,

$$Y = F(A, B, C) = \Sigma\, m\,(3, 5, 6, 7)$$

where 'Σ' symbolizes summation or logical OR operation that is performed on corresponding minterms and $Y = F(A, B, C)$ means Y is a function of three Boolean variables A, B and C. This kind of representation of a truth table is also known as *canonical sum form*.

Table 3.3 Design Truth Table

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

Table 3.4 Fundamental Products for Table 3.3

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	$1 \rightarrow \overline{A}BC$
1	0	0	0
1	0	1	$1 \rightarrow A\overline{B}C$
1	1	0	$1 \rightarrow AB\overline{C}$
1	1	1	$1 \rightarrow ABC$

Logic Circuit

After you have a sum-of-products equation, you can derive the corresponding logic circuit by drawing an AND-OR network, or what amounts to the same thing, a NAND-NAND network. In Eq. (3.24) each product is the output of a 3-input AND gate. Furthermore, the logical sum Y is the output of a 4-input OR gate. Therefore, we can draw the logic circuit as shown in Fig. 3.5. This AND-OR circuit is one solution to the design problem that we started with. In other words, the AND-OR circuit of Fig. 3.5 has the truth table given by Table 3.3.

We cannot build the circuit of Fig. 3.5 because a 4-input OR gate is not available as a TTL *chip* (a synonym for integrated circuit). But a 4-input NAND gate is. Figure 3.6 shows the logic circuit as a NAND-NAND circuit with TTL pin numbers. Also notice how the inputs come from a *bus*, a

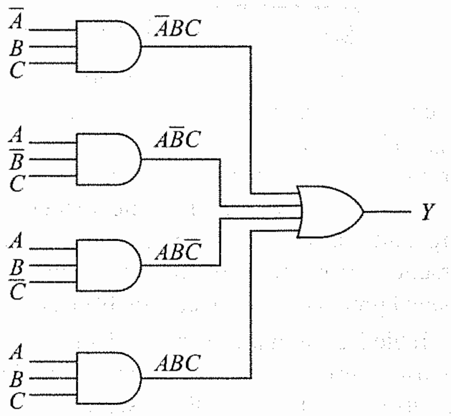


Fig. 3.5 AND-OR solution

group of wires carrying logic signals. In Fig. 3.6, the bus has six wires with logic signals A, B, C, and their complements. Microcomputers are bus-organized, meaning that the input and output signals of the logic circuits are connected to buses.

Example 3.4 Suppose a three-valuable truth table has a high output for these input conditions: 000, 010, 100, and 110. What is the sum-of-products circuit?

Solution Here are the fundamental products:

$000 : \overline{A}\,\overline{B}\,\overline{C}$

$010 : \overline{A}B\overline{C}$

$100 : A\overline{B}\,\overline{C}$

$110 : AB\overline{C}$

When you OR these products, you get

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C}$$

The circuit of Fig. 3.6 will work if we reconnect the input lines to the bus as follows:

$\overline{A}$: pins 1 and 3

$\overline{B}$: pins 2 and 10

$\overline{C}$: pins 13, 5, 11, and 13

A : pins 9 and 1

B : pins 4 and 2

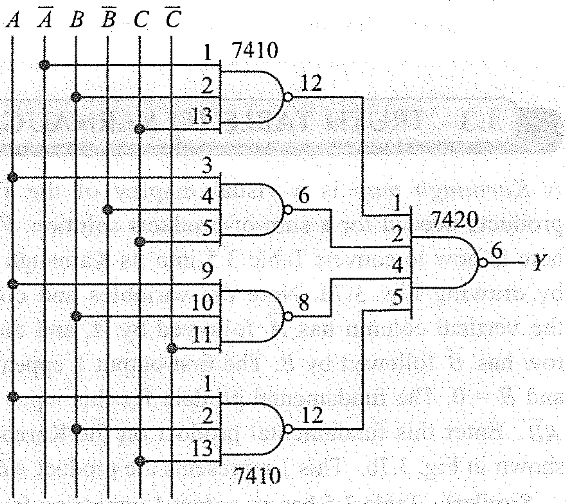


Fig. 3.6 **Combinational logic circuit**

Example 3.5 Simplify the Boolean equation in Example 3.4 and describe the logic circuit.

Solution The Boolean equation is

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C}$$

Since $\overline{C}$ is common to each term, factor as follows:

$$Y = (\overline{A}\,\overline{B} + \overline{A}B + A\overline{B} + AB)\overline{C}$$

Again, factor to get

$$Y = [\overline{A}(\overline{B} + B) + A(\overline{B} + B)]\overline{C}$$

Now, simplify the foregoing as follows:

$$Y = [\overline{A}(1) + A(1)]\overline{C} = (\overline{A} + A)\overline{C}$$

or

$$Y = \overline{C}$$

This final equation means that you don't even need a logic circuit. All you need is a wire connecting input $\overline{C}$ to output Y.

The lesson is clear. The AND-OR (NAND-NAND) circuit you get with the sum-of-products method is not necessarily as simple as possible. With algebra, you often can factor and reduce the sum-of-products equation to arrive at a simpler Boolean equation, which means a simpler logic circuit. A simpler logic circuit is preferred because it usually costs less to build and is more reliable.

SELF-TEST

4. How many fundamental products are there for two variables? How many for three variables?
5. The AND-OR or the NAND-NAND circuit obtained with the sum-of-products method is always the simplest possible circuit. (T or F)

3.3 TRUTH TABLE TO KARNAUGH MAP

A *Karnaugh map* is a visual display of the fundamental products needed for a sum-of-products solution. For instance, here is how to convert Table 3.5 into its Karnaugh map. Begin by drawing Fig. 3.7a. Note the variables and complements: the vertical column has $\bar{A}$ followed by A, and the horizontal row has $\bar{B}$ followed by B. The first output 1 appears for $A = 1$ and $B = 0$. The fundamental product for this input condition is $A\bar{B}$. Enter this fundamental product on the Karnaugh map as shown in Fig. 3.7b. This 1 represents the product $A\bar{B}$ because the 1 is in row A and column $\bar{B}$.

Table 3.5

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

Similarly, Table 3.5 has an output 1 appearing for inputs of $A = 1$ and $B = 1$. The fundamental product is AB, which can be entered on the Karnaugh map as shown in Fig. 3.7c. The final step in drawing the Karnaugh map is to enter 0s in the remaining spaces (see Fig. 3.7d).

In terms of decimal equivalence each position of Karnaugh map can be drawn as shown in Fig. 3.7b. Note that, Table 3.5 can be written using minterms as $Y = \Sigma\, m(2, 3)$ and Fig. 3.7e represents that.

$$\begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & & \\ A & & \end{array} \qquad \begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & 0 & 1 \\ A & 2 & 3 \end{array} \qquad \begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & & \\ A & 1 & \end{array} \qquad \begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & & \\ A & 1 & 1 \end{array} \qquad \begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & 0 & 0 \\ A & 1 & 1 \end{array}$$

(a) (b) (c) (d) (e)

Fig. 3.7 **Constructing a Karnaugh map**

Three-Variable Maps

Here is how to draw a Karnaugh map for Table 3.6 or for logic equation, $Y = F(A, B, C) = \Sigma m(2,6,7)$. First, draw the blank map of Fig. 3.8a. The vertical column is labeled $\bar{A}\bar{B}$, $\bar{A}B$, AB, and $A\bar{B}$. With this order, only one variable changes from complemented to uncomplemented form (or vice versa) as you move downward. In terms of decimal equivalence of each position the Karnaugh map is as shown in Fig. 3.8b. Note how minterms in the equation gets mapped into corresponding positions in the map.

Table 3.6

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

Next, look for output 1s in Table 3.6. Output 1s appear for ABC inputs of 010, 110 and 111. The fundamental products for these input conditions are $\bar{A}B\bar{C}$, $AB\bar{C}$, and ABC Enter 1s for these products on the Karnaugh map (Fig. 3.8b).

The final step is to enter 0s in the remaining spaces (Fig. 3.8c).

(a)

	$\bar{C}$	C
$\bar{A}\bar{B}$		
$\bar{A}B$		
AB		
$A\bar{B}$		

(b)

	$\bar{C}$	C
$\bar{A}\bar{B}$	0	1
$\bar{A}B$	2	3
AB	6	7
$A\bar{B}$	4	5

(c)

	$\bar{C}$	C
$\bar{A}\bar{B}$		
$\bar{A}B$	1	
AB	1	1
$A\bar{B}$		

(d)

	$\bar{C}$	C
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	1	0
AB	1	1
$A\bar{B}$	0	0

Fig. 3.8 Three-variable Karnaugh map

Four-Variable Maps

Many digital computers and systems process 4-bit numbers. For instance, some digital chips will work with nibbles like 0000, 0001, 0010, and so on. For this reason, logic circuits are often designed to handle four input variables (or their complements). This is why you must know how to draw a four-variable Karnaugh map.

Here is an example. Suppose you have a truth table like Table 3.7. Start by drawing a blank map like Fig. 3.9a. Notice the order. The vertical column is $\bar{A}\bar{B}, \bar{A}B, AB,$ and $A\bar{B}$. The horizontal row is $\bar{C}\bar{D}, \bar{C}D, CD,$ and $C\bar{D}$. In terms of decimal equivalence of each position the Karnaugh map is as shown in Fig. 3.9b. In Table 3.7, you have output 1s appearing for $ABCD$ inputs of 0001, 0110, 0111, and 1110. The fundamental products for these input conditions are $\bar{A}\bar{B}\bar{C}D, \bar{A}BC\bar{D}, \bar{A}BCD,$ and $ABC\bar{D}$. After entering 1s on the Karnaugh map, you have Fig. 3.9c. The final step of filling in 0s results in the complete map of Fig. 3.9d.

Table 3.7

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

SELF-TEST

6. What is a Karnaugh map?
7. How many entries are there on a four-variable Karnaugh map?

Entered Variable Map

As the name suggests, in *entered variable map* one of the input variable is placed inside Karnaugh map. This is done separately noting how it is related with output. This reduces the Karnaugh map size by one degree,

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$				
$\bar{A}B$				
AB				
$A\bar{B}$				

(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	3	2
$\bar{A}B$	4	5	7	6
AB	12	13	15	14
$A\bar{B}$	8	9	11	10

(b)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	3	2
$\bar{A}B$	4	5	7	6
AB	12	13	15	14
$A\bar{B}$	8	9	11	10

(c)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	0	1	1
AB	0	0	0	1
$A\bar{B}$	0	0	0	0

(d)

Fig. 3.9 **Constructing a four-variable Karnaugh map**

i.e. a three variable problem that requires $2^3 = 8$ locations in Karnaugh map will require $2^{(3-1)} = 4$ locations in entered variable map. This technique is particularly useful for mapping problems with more than four input variables.

We illustrate the technique by taking a three variable example, truth table of which is shown in Table 3.6. Let's choose C as *map entered variable* and see how output Y varies with C for different combinations of other two variables A and B. Fig. 3.10a shows the relation drawn from Table 3.6. For $AB = 00$ we find $Y = 0$ and is not dependent on C. For $AB = 01$ we find Y is complement of C thus we can write $Y = C'$. Similarly, for $AB = 10$, $Y = 0$ and for $AB = 11$, $Y = 1$. The corresponding entered variable map is shown in Fig. 3.10b. If we choose A as map entered variable we have table shown in Fig. 3.10c showing relation with Y for various combinations of BC; the corresponding entered variable map is shown in Fig. 3.10d.

A	B	Y
0	0	0
0	1	$\bar{C}$
1	0	0
1	1	1

(a)

	$\bar{B}$	B
$\bar{A}$	0	$\bar{C}$
A	0	1

(b)

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

(c)

	$\bar{C}$	C
$\bar{B}$	0	0
B	1	A

(d)

Fig. 3.10 **Entered variable map of truth table shown in Table 3.6**

3.4 PAIRS, QUADS, AND OCTETS

Look at Fig. 3.11a. The map contains a pair of 1s that are horizontally adjacent (next to each other). The first 1 represents the product $ABCD$; the second 1 stands for the product $ABC\bar{D}$. As we move from the first 1 to the second 1, only one variable goes from uncomplemented to complemented form (D to $\bar{D}$); the other variables don't change form (A, B and C remain uncomplemented). Whenever this happens, you can *eliminate the variable that changes form*.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	0	0	1	1
$A\bar{B}$	0	0	0	0

(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	0	0	1	1
$A\bar{B}$	0	0	0	0

(b)

Fig. 3.11 **Horizontally adjacent 1s**

Proof

The sum-of-products equation corresponding to Fig. 3.11a is

$$Y = ABCD + ABC\overline{D}$$

which factors into

$$Y = ABC(D + \overline{D})$$

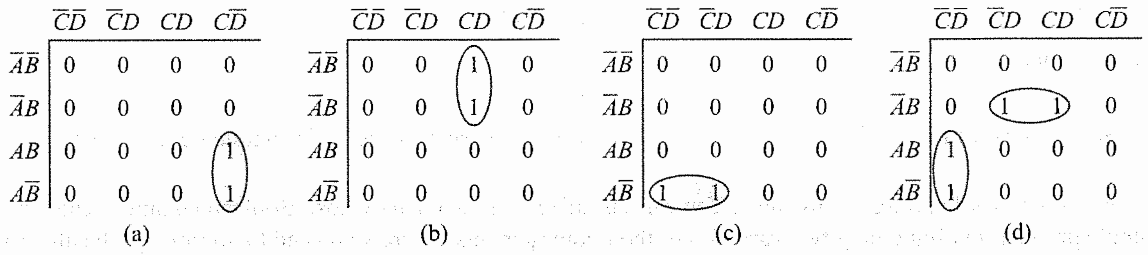
Since D is ORed with its complement, the equation simplifies to

$$Y = ABC$$

In general, a pair of horizontally adjacent 1s like those of Fig. 3.11a means the sum-of-products equation will have a variable and a complement that drop out as shown above.

For easy identification, we will encircle two adjacent 1s as shown in Fig. 3.11b. Two adjacent 1s such as these are called a pair. In this way, we can tell at a glance that one variable and its complement will drop out of the corresponding Boolean equation. In other words, an encircled pair of 1s like those of Fig. 3.11b no longer stand for the ORing of two separate products, $ABCD$ and $ABC\overline{D}$. Rather, the encircled pair is visualized as representing a single reduced product ABC.

Here is another example. Figure 3.12a shows a pair of 1s that are vertically adjacent. These 1s correspond to $ABC\overline{D}$ and $A\overline{B}C\overline{D}$. Notice that only one variable changes from uncomplemented to complemented form (B to $\overline{B}$). Therefore, B and $\overline{B}$ can be factored and eliminated algebraically, leaving a reduced product of $AC\overline{D}$.



$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$
$\overline{A}B$
AB
$A\overline{B}$
(a) (b) (c) (d)

Fig. 3.12 Examples of pairs

More Examples

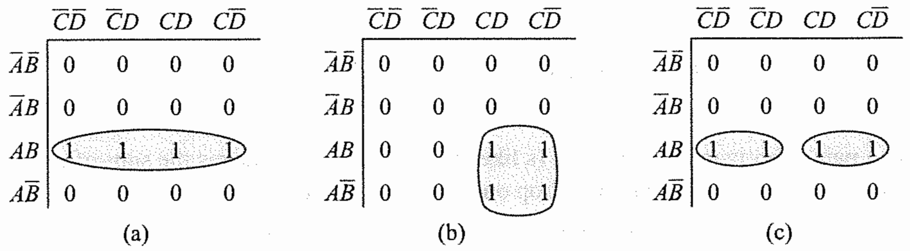
Whenever you see a pair of horizontally or vertically adjacent 1s, you can eliminate the variable that appears in both complemented and uncomplemented form. The remaining variables (or their complements) will be the only ones appearing in the single-product term corresponding to the pair of 1s. For instance, a glance at Fig. 3.12b indicates that B goes from complemented to uncomplemented form when we move from the upper to the lower 1; the other variables remain the same. Therefore, the encircled pair of 1s in Fig. 3.12b, represents the product $\overline{A}CD$. Likewise, given the pair of 1s in Fig. 3.12c, the only change is from $\overline{D}$ to D. So the encircled pair of 1s stands for the product $A\overline{B}\,\overline{C}$.

If more than one pair exists on a Karnaugh map, you can OR the simplified products to get the Boolean equation. For instance, the lower pair of Fig. 3.12d represents the simplified product $A\overline{C}\overline{D}$; the upper pair stands for $\overline{A}BD$. The corresponding Boolean equation for this map is

$$Y = A\overline{C}\overline{D} + \overline{A}BD$$

The Quad

A *quad* is a group of four 1s that are horizontally or vertically adjacent. The 1s may be end-to-end, as shown in Fig. 3.13a, or in the form of a square, as in Fig. 3.13b. When you see a quad, always encircle it because it leads to a simpler product. In fact, a quad eliminates *two variables and their complements*.



(a)	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	1	1	1	1
$A\bar{B}$	0	0	0	0

(b)	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	0	0	1	1
$A\bar{B}$	0	0	1	1

(c)	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	1	1	1	1
$A\bar{B}$	0	0	0	0

Fig. 3.13 **Examples of quads**

Here is why a quad eliminates two variables and their complements. Visualize the four 1s of Fig. 3.13a as two pairs (see Fig. 3.13c). The first pair represents $AB\bar{C}$; the second pair stands for ABC. The Boolean equation for these two pairs is

$$Y = AB\bar{C} + ABC$$

This factors into

$$Y = AB(\bar{C} + C)$$

which reduces to

$$Y = AB$$

So, the quad of Fig. 3.13a represents a product whose two variables and their complements have dropped out.

A similar proof applies to any quad. You can visualize it as two pairs whose Boolean equation leads to a single product involving only two variables or their complements. There's no need to go through the algebra each time. Merely step through the different 1s in the quad and determine which two variables go from complemented to uncomplemented form (or vice versa); these are the variables that drop out.

For instance, look at the quad of Fig. 3.13b. Pick any 1 as a starting point. When you move horizontally, D is the variable that changes form. When you move vertically, B changes form. Therefore, the remaining variables (A and C) are the only ones appearing in the simplified product. In other words, the simplified equation for the quad of Fig. 3.13b is

$$Y = AC$$

The Octet

Besides pairs and quads, there is one more group to adjacent 1s to look for: the *octet*. This is a group of eight 1s like those of Fig. 3.14a on the next page. An octet like this eliminates *three variables and their complements*. Here's why. Visualize the octet as two quads (see Fig. 3.14b). The equation for these two quads is

$$Y = A\bar{C} + AC$$

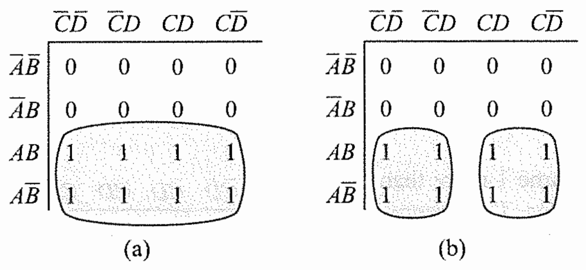

$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 0 0 0 0
AB 1 1 1 1
$A\overline{B}$ 1 1 1 1
(a)
$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 0 0 0 0
AB 1 1 1 1
$A\overline{B}$ 1 1 1 1
(b)

Fig. 3.14 **Example of octet**

After factoring,

$$Y = A(\overline{C} + C)$$

But this reduces to

$$Y = A$$

So the octet of Fig. 3.14a means three variables and their complements drop out of the corresponding product.

A similar proof applies to any octet. From now on don't bother with the algebra. Merely step through the 1s of the octet and determine which three variables change form. These are the variables that drop out.

SELF-TEST

8. On a Karnaugh map, two adjacent 1s are called a ____.
9. On a Karnaugh map, an octet contains how many 1s?

3.5 KARNAUGH SIMPLIFICATIONS

As you know, a pair eliminates one variable and its complement, a quad eliminates two variables and their complements, and an octet eliminates three variables and their complements. Because of this, after you draw a Karnaugh map, encircle the octets first, the quads second, and the pairs last. In this way, the greatest simplification results.

An Example

Suppose you have translated a truth table into the Karnaugh map shown in Fig. 3.15a. First, look for octets. There are none. Next, look for quads. When you find them, encircle them. Finally, look for and encircle pairs. If you do this correctly, you arrive at Fig. 3.15b.

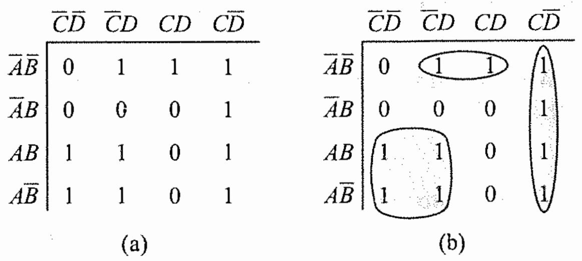

$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 1 1 1
$\overline{A}B$ 0 0 0 1
AB 1 1 0 1
$A\overline{B}$ 1 1 0 1
(a)
$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 1 1 1
$\overline{A}B$ 0 0 0 1
AB 1 1 0 1
$A\overline{B}$ 1 1 0 1
(b)

Fig. 3.15 **Encircling octets, quads and pairs**

The pair represents the simplified product $\overline{A}\overline{B}D$, the lower quad stands for $A\overline{C}$, and the quad on the right represents $C\overline{D}$. By ORing these simplified products, we get the Boolean equation corresponding to the entire

Karnaugh map:

$$Y = \overline{A}\overline{B}D + A\overline{C} + C\overline{D} \quad (3.25)$$

Overlapping Groups

You are allowed to use the same 1 more than once. Figure 3.16a illustrates this idea. The I representing the fundamental product $AB\overline{C}D$ is part of the pair and part of the octet. The simplified equation for the overlapping groups is

$$Y = A + B\overline{C}D \quad (3.26)$$

It is valid to encircle the 1s as shown in Fig. 3.16b, but then the isolated 1 results in a more complicated equation:

$$Y = A + \overline{A}B\overline{C}D$$

So, always overlap groups if possible. That is, use the 1s more than once to get the largest groups you can.

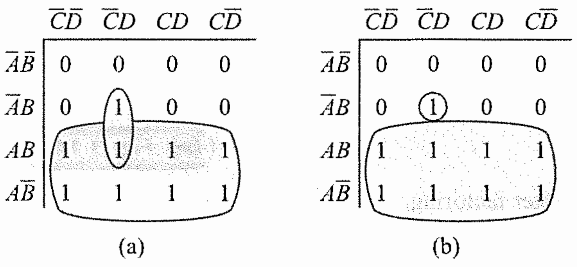

$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 0 1 0 0
AB 1 1 1 1
$A\overline{B}$ 1 1 1 1
(a)
(b)

Fig. 3.16 Overlapping groups

Rolling the Map

Another thing to know about is rolling. Look at Fig. 3.17a on the next page. The pairs result in this equation:

$$Y = B\overline{C}\overline{D} + BC\overline{D} \quad (3.27)$$

Visualize picking up the Karnaugh map and rolling it so that the left side touches the right side. If you are visualizing correctly, you will realize the two pairs actually form a quad. To indicate this, draw half circles around each pair, as shown in Fig. 3.17b. From this viewpoint, the quad of Fig. 3.17b has the equation

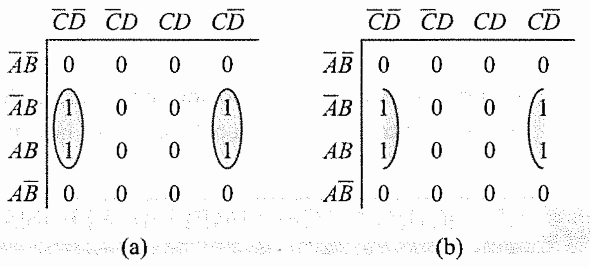

$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 1 0 0 1
AB 1 0 0 1
$A\overline{B}$ 0 0 0 0
(a)
(b)

Fig. 3.17 Rolling the Karnaugh map

$$Y = B\overline{D} \quad (3.28)$$

Why is rolling valid? Because Eq. (3.27) can be algebraically simplified to Eq. (3.28). The proof starts with Eq. (3.27):

$$Y = B\overline{C}\overline{D} + BC\overline{D}$$

This factors into

$$Y = B\overline{D}(\overline{C} + C)$$

which reduces to

$$Y = B\overline{D}$$

But this final equation is the one that represents a rolled quad like Fig. 3.17b. Therefore, 1s on the edges of a Karnaugh map can be grouped with 1s on opposite edges.

More Examples

If possible, roll and overlap to get the largest groups you can find. For instance, Fig. 3.18a shows an inefficient way to encircle groups. The octet and pair have a Boolean equation of

$$Y = \overline{C} + BC\overline{D}$$

You can do better by rolling and overlapping as shown in Fig. 3.18b; the Boolean equation now is

$$Y = \overline{C} + B\overline{D}$$

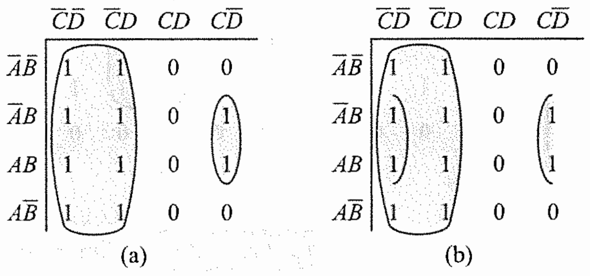


Fig. 3.18 Rolling and overlapping

Here is another example. Figure 3.19a shows an inefficient grouping of 1s; the corresponding equation is

$$Y = \overline{C} + \overline{A}C\overline{D} + A\overline{B}C\overline{D}$$

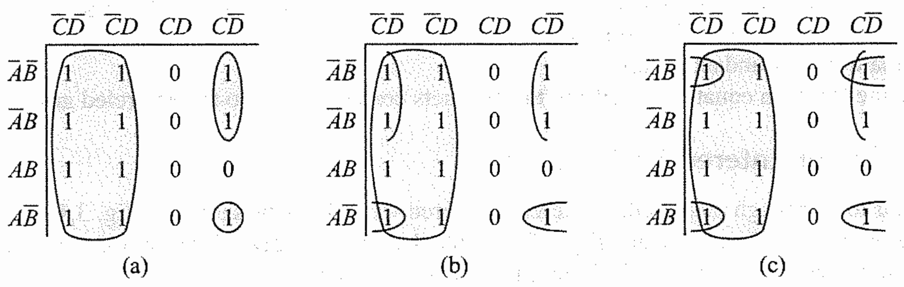


Fig. 3.19 Different ways of encircling groups

If we roll and overlap as shown in Fig. 3.19b, the equation is simpler:

$$Y = \overline{C} + \overline{A}\overline{D} + A\overline{B}\overline{D}$$

It is possible to group the 1s as shown in Fig. 3.19c. The equation now becomes

$$Y = \overline{C} + \overline{A}\overline{D} + \overline{B}\overline{D} \tag{3.29}$$

Compare this with the preceding equation. As you can see, the equations are comparable in simplicity. Either grouping (Fig. 3.19b or c) is valid; therefore, you can use whichever you like.

Eliminating Redundant Groups

After you have finished encircling groups, eliminate any *redundant group*. This is a group whose 1s are already used by other groups. Here is an example. Given Fig. 3.20a, encircle the quad to get Fig. 3.20b. Next, group the remaining 1s into pairs by overlapping (Fig. 3.20c). In Fig. 3.20c, all the 1s of the quad are used by the pairs. Because of this, the quad is redundant and can be eliminated to get Fig. 3.20d. As you see, all the 1s are covered by the pairs. Figure 3.20d contains one less product than Fig. 3.20c; therefore, Fig. 3.20d is the most efficient way to group the 1s.

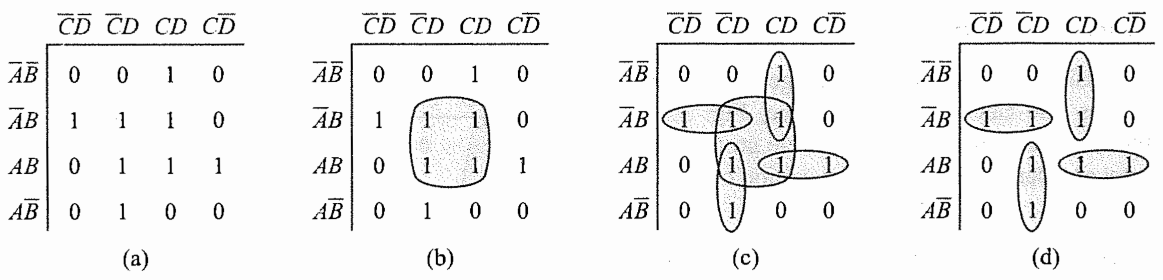


Fig. 3.20 **Eliminating an unnecessary group**

Conclusion

Here is a summary of the Karnaugh-map method for simplifying Boolean equations:

1. Enter a 1 on the Karnaugh map for each fundamental product that produces a 1 output in the truth table. Enter 0s elsewhere.
2. Encircle the octets, quads, and pairs. Remember to roll and overlap to get the largest groups possible.
3. If any isolated 1s remain, encircle each.
4. Eliminate any redundant group.
5. Write the Boolean equation by ORing the products corresponding to the encircled groups.

Simplification of Entered Variable Map

This is similar to Karnaugh map method. Refer to entered variable maps shown in Fig. 3.10. The groupings for these are as shown in Fig. 3.21a and Fig. 3.21b. Note that in Fig. 3.21a C' is grouped with 1 to get a larger group as 1 can be written as $1 = 1 + C'$. Similarly A is grouped with 1 in Fig. 3.21b.

Next, the product term representing each group is obtained by including map entered variable in the group as an additional ANDed term. Thus, group 1 of Fig. 3.21a gives $B.(C') = BC'$ and group 2 gives $AB.(1) = AB$ resulting $Y = BC' + AB$.

In Fig. 3.21b, group 1 gives product term $B.(A) = AB$ and group 2 gives $BC'.(1) = BC'$ so that $Y = BC' + AB$. The final expression is same for both as they represent the same truth table (Table 3.6).

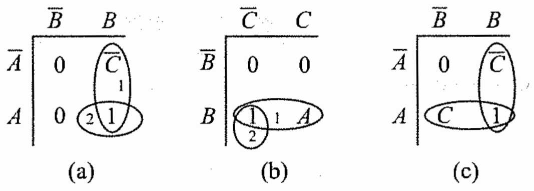


Fig. 3.21 **Simplification of entered variable map**

Note that, entered variable map shown Fig. 3.21c for a different truth table (Take it as an exercise to prepare that truth table) has only two product terms and doesn't need a separate coverage of 1. This is because one can write $1 = C + C'$ and C is included in one group while C' in other. The output of this map can be written as $Y = AC + BC'$.

Example 3.6 What is the simplified Boolean equation for the following logic equation expressed by minterms?

$$Y = F(A, B, C, D) = \Sigma m(7, 9, 10, 11, 12, 13, 14, 15)$$

Solution We know, each minterm makes corresponding location in Karnaugh map 1 and thus Fig. 3.22a represents the given equation. There are no octets, but there is a quad as shown in Fig. 3.22b. By overlapping, we can find two more quads (see Fig. 3.22c). We can encircle the remaining 1 by making it part of an overlapped pair (Fig. 3.22d). Finally, there are no redundant groups.

The horizontal quad of Fig. 3.22d corresponds to a simplified product AB. The square quad on the right corresponds to AC, while the one on the left stands for AD. The pair represents BCD. By ORing these products, we get the simplified Boolean equation:

$$Y = AB + AC + AD + BCD \tag{3.30}$$

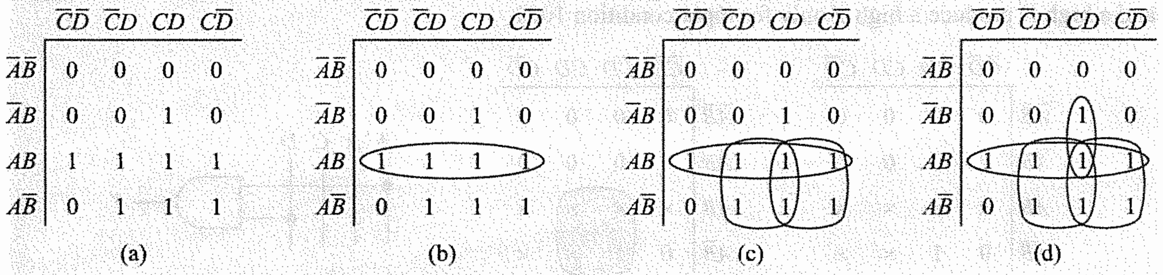


Fig. 3.22 **Using the Karnaugh map**

SELF-TEST

10. Write the sum-of-product terms for the entries in Fig. 3.18. Use Boolean algebra to simplify the expression.

3.6 DON'T-CARE CONDITIONS

In some digital systems, certain input conditions never occur during normal operation; therefore, the corresponding output never appears. Since the output never appears, it is indicated by an X in the truth table. For instance, Table 3.8 on the next page shows a truth table where the output is low for all input entries from 0000 to 1000, high for input entry 1001, and an X for 1010 through 1111. The X is called a *don't-care condition*. Whenever you see an X in a truth table, you can let it equal either 0 or 1, whichever produces a simpler logic circuit.

Figure 3.23a shows the Karnaugh map of Table 3.8 with don't-cares for all inputs from 1010 to 1111. These don't-cares are like wild cards in poker because you can let them stand for whatever you like. Figure 3.23b shows the most efficient way to encircle the 1. Notice two crucial ideas. First, the 1 is included in a

Table 3.8 **Truth Table with Don't-Care Conditions**

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

quad, the largest group you can find if you visualize all X's as 1s. Second, after the 1 has been encircled, all X's outside the quad are visualized as 0s. In this way, the X's are used to the best possible advantage. As already mentioned, you are free to do this because don't-cares correspond to input conditions that never appear.

The quad of Fig. 3.23b results in a Boolean equation of

$$Y = AD$$

The logic circuit for this is an AND gate with inputs of A and D, as shown in Fig. 3.23c. You can check this logic circuit by examining Table 3.8. The possible inputs are from 0000 to 1001; in this range a high A and a high D produce a high Y only for input condition 1001.

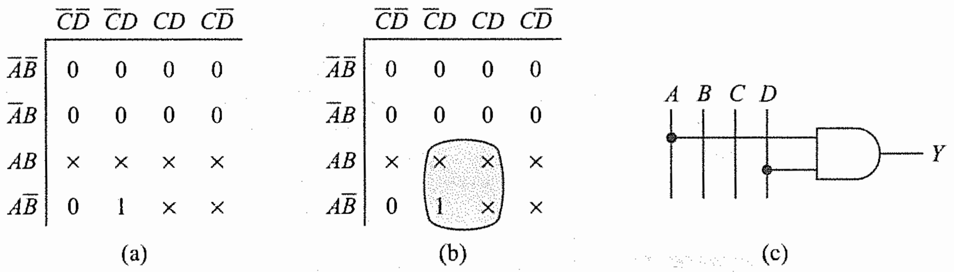


Fig. 3.23 **Don't-care conditions**

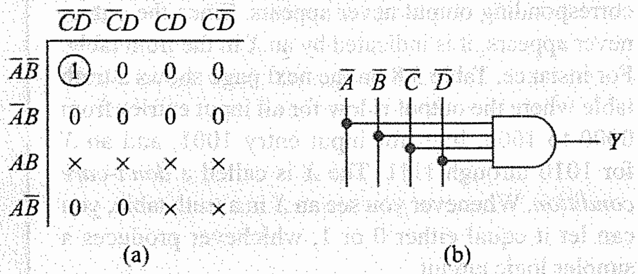
Remember these ideas about don't-care conditions:

1. Given the truth table, draw a Karnaugh map with 0s, 1s, and don't-cares.
2. Encircle the actual 1s on the Karnaugh map in the largest groups you can find by treating the don't-cares as 1s.
3. After the actual 1s have been included in groups, disregard the remaining don't cares by visualizing them as 0s.

Example 3.7 Suppose Table 3.8 has high output for an input of 0000, low output, for 0001 to 1001, and don't cares for 1010 to 1111. What is the simplest logic circuit with this truth table?

Solution The truth table has a 1 output only for the input condition 0000. The corresponding fundamental product is $\bar{A}\bar{B}\bar{C}\bar{D}$. Figure 3.24a shows the Karnaugh map with a 1 for the fundamental product, 0s for inputs 0001 to 1001, and X's for inputs 1010 to 1111. In this case, the don't-cares are of no help. The best we can do is to encircle the isolated 1, while treating the don't-cares as 0s. So, the Boolean equation is

$$Y = \bar{A}\bar{B}\bar{C}\bar{D}$$



$\bar{C}\bar{D}$ $\bar{C}D$ CD $C\bar{D}$
$\bar{A}\bar{B}$ 1 0 0 0
$\bar{A}B$ 0 0 0 0
AB × × × ×
$A\bar{B}$ 0 0 × ×
(a)
$\bar{A}$ $\bar{B}$ $\bar{C}$ $\bar{D}$
Y
(b)

Fig. 3.24 **Decoding 0000**

Figure 3.24b shows the logic circuit. The 4-input AND gate produces a high output only for the input condition $A = 0$, $B = 0$, $C = 0$, and $D = 0$.

Example 3.8 Give the simplest logic circuit for following logic equation where d represents don't-care condition for following locations.

$$F(A, B, C, D) = \Sigma m(7) + d(10, 11, 12, 13, 14, 15)$$

Solution Figure 3.25a is the Karnaugh map. The most efficient encircling is to group the 1s into a pair using the don't-care as shown. Since this is the largest group possible, all remaining don't cares are treated as 0s. The equation for the pair is

$$Y = BCD$$

and Fig. 3.25b is the logic circuit. This 3.input AND gate produces a high output only for an input of $A = 0$, $B = 1$, $C = 1$, and $D = 1$ because the input possibilities range only from 0000 to 1001.

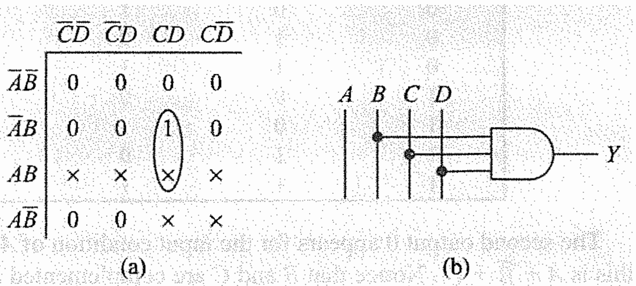


Fig. 3.25 **Decoding 0111**

SELF-TEST

11. What is meant by a don't-care condition on a Karnaugh map? How is it indicated?
12. How can using don't-cares aid circuit simplification?

3.7 PRODUCT-OF-SUMS METHOD

With the sum-of-products method the design starts with a truth table that summarizes the desired input-output conditions. The next step is to convert the truth table into an equivalent sum-of-products equation. The final step is to draw the AND-OR network or its NAND-NAND equivalent.

The product-of-sums method is similar. Given a truth table, you identify the fundamental sums needed for a logic design. Then by ANDing these sums, you get the product-of-sums equation corresponding to the truth table. But there are some differences between the two approaches. With the sum-of-products method, the fundamental product produces an output 1 for the corresponding input condition. But with the product-of-sums method, the fundamental sum produces an output 0 for the corresponding input condition. The best way to understand this distinction is with an example.

Converting a Truth Table to an Equation

Suppose you are given a truth table like Table 3.9 and you want to get the product-of-sums equation. What you have to do is locate each output 0 in the truth table and write down its fundamental sum. In Table 3.9, the first output 0 appears for $A = 0$, $B = 0$, and $C = 0$. The fundamental sum for these inputs is $A + B + C$. Why? Because this produces an output zero for the corresponding input condition:

$$Y = A + B + C = 0 + 0 + 0 = 0$$

Table 3.9

A	B	C	Y	*Maxterm*
0	0	0	$0 \rightarrow A+B+C$	M_0
0	0	1	1	M_1
0	1	0	1	M_2
0	1	1	$0 \rightarrow A+\overline{B}+\overline{C}$	M_3
1	0	0	1	M_4
1	0	1	1	M_5
1	1	0	$0 \rightarrow \overline{A}+\overline{B}+C$	M_6
1	1	1	1	M_7

The second output 0 appears for the input condition of $A = 0$, $B = 1$, and $C = 1$. The fundamental sum for this is $A + \overline{B} + \overline{C}$. Notice that B and C are complemented because this is the only way to get a logical sum of 0 for the given input conditions:

$$Y = A + \overline{B} + \overline{C} = 0 + \overline{1} + \overline{1} = 0 + 0 + 0 = 0$$

Similarly, the third output 0 occurs for $A = 1$, $B = 1$, and $C = 0$; therefore, its fundamental sum is $\overline{A} + \overline{B} + C$:

$$Y = \overline{A} + \overline{B} + C = \overline{1} + \overline{1} + 0 = 0 + 0 + 0 = 0$$

Table 3.9 shows all the fundamental sums needed to implement the truth table. Notice that each variable is complemented when the corresponding input variable is a 1; the variable is uncomplemented when the corresponding input variable is 0. To get the product-of-sums equation, all you have to do is AND the fundamental sums:

$$Y = (A + B + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C) \tag{3.31}$$

This is the product-of-sums equation for Table 3.9.

As each product term was called minterm in SOP representation in POS each sum term is called *maxterm* and is designated by M_i as shown in Table 3.9. Equation 3.31 in terms of maxterm can be represented as

$$Y = F(A, B, C) = \Pi M(0, 3, 6)$$

where 'Π' symbolizes product, i.e. AND operation. This kind of representation of a truth table is also known as *canonical product form*.

Logic Circuit

After you have a product-of-sums equation, you can get the logic circuit by drawing an OR-AND network, or if you prefer, a NOR-NOR network. In Eq. (3.31) each sum represents the output of a 3-input OR gate. Furthermore, the logical product Y is the output of a 3-input AND gate. Therefore, you can draw the logic circuit as shown in Fig. 3.26.

A 3-input OR gate is not available as a TTL chip. So, the circuit of Fig. 3.26 is not practical. With De Morgan's first theorem, however, you can replace the OR-AND circuit of Fig. 3.26 by the NOR-NOR circuit of Fig. 3.27.

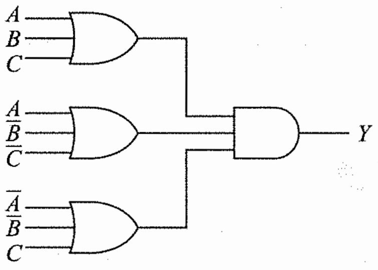


Fig. 3.26 Product-of-sums circuit

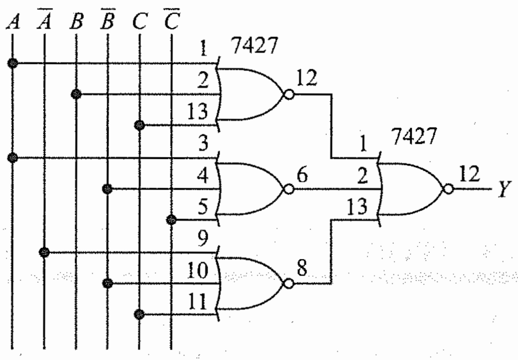


Fig. 3.27

Conversion between SOP and POS

We have seen that SOP representation is obtained by considering ones in a truth table while POS comes considering zeros. In SOP, each one at output gives one AND term which is finally ORed. In POS, each zero gives one OR term which is finally ANDed. Thus SOP and POS occupy complementary locations in a truth table and one representation can be obtained from the other by

(i) identifying complementary locations,
(ii) changing minterm to maxterm or reverse, and finally
(iii) changing summation by product or reverse.

Thus Table 3.9 can be represented as

$$Y = F(A, B, C) = \Pi M(0, 3, 6) = \Sigma m(1, 2, 4, 5, 7)$$

Similarly Table 3.4 can be represented as

$$Y = F(A, B, C) = \Sigma m(3, 5, 6, 7) = \Pi M(0, 1, 2, 4)$$

This is also known as *conversion between canonical forms.*

Example 3.9 Suppose a truth table has a low output for the first three input conditions: 000, 001, and 010. If all other outputs are high, what is the product-of-sums circuit?

Solution The product-of-sums equation is

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

The circuit of Fig. 3.27 will work if we reconnect the input lines as follows:

A : pins 1, 3, and 9
B : pins 2 and 4
C : pins 13 and 11
$\overline{B}$: pin 10
$\overline{C}$: pin 5

SELF-TEST

13. A product-of-sums expression leads to what kind of logic circuit?
14. Explain how to convert the complementary NAND-NAND circuit into its dual NOR-NOR circuit.

3.8 PRODUCT-OF-SUMS SIMPLIFICATION

After you write a product-of-sums equation, you can simplify it with Boolean algebra. Alternatively, you may prefer simplification based on the Karnaugh map. There are several ways of using the Karnaugh map. One can use a similar technique as followed in SOP representation but by forming largest group of zeros and then replacing each group by a sum term. The variable going in the formation of sum term is inverted if it remains constant with a value 1 in the group and it is not inverted if that value is 0. Finally, all the sum terms are ANDed to get simplest POS form. We illustrate this in Examples 3.11 and 3.12. In this section we also present an interesting alternative to above technique.

Sum-of-Products Circuit

Suppose the design starts with a truth table like Table 3.10. The first thing to do is to draw the Karnaugh map in the usual way to get Fig. 3.28a. The encircled groups allow us to write a sum-of-products equation:

$$Y = \overline{A}\overline{B} + AB + AC$$

Figure 3.28b shows the corresponding NAND-NAND circuit.

Complementary Circuit

To get a product-of-sums circuit, begin by complementing each 0 and 1 on the Karnaugh map of Fig. 3.28a. This results in the complemented map shown in Fig. 3.28c. The encircled 1s allow us to write the following sum-of-products equation:

$$\overline{Y} = \overline{A}B + A\overline{B}\overline{C}$$

Why is this $\overline{Y}$ instead of Y? Because complementing the Karnaugh map is the same as complementing the output of the truth table, which means the sum-of-products equation for Fig. 3.28c is for $\overline{Y}$ instead of Y.

Figure 3.28d shows the corresponding NAND-NAND circuit for $\overline{Y}$. This circuit does not produce the desired output; it produces the complement of the desired output.

Finding the NOR-NOR Circuit

What we want to do next is to get the product-of-sums solution, the NOR-NOR circuit that produces the

Table 3.10

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

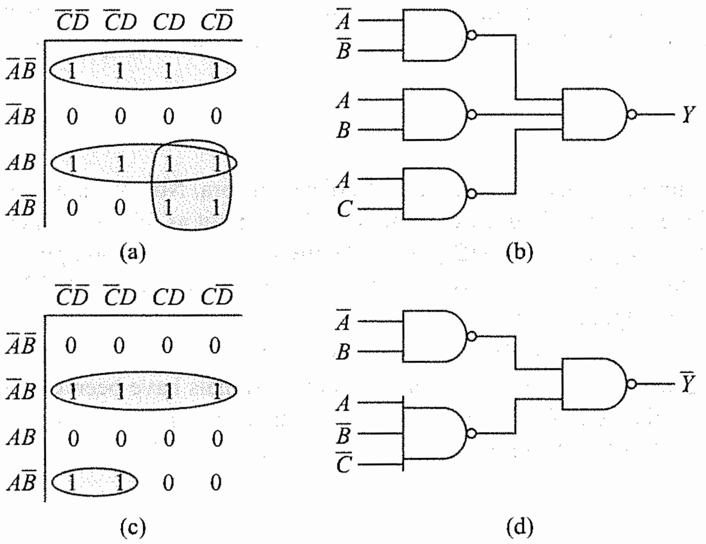


Fig. 3.28 **Deriving the sum-of-products circuit**

original truth table of Table 3.10. De Morgan's first theorem tells us NAND gates can be replaced by bubbled OR gates; therefore, we can replace Fig. 3.28d by Fig. 3.29a. A bus with each variable and its complement is usually available in a digital system. So, instead of connecting $\overline{A}$ and B to a bubbled OR gate, as shown in Fig. 3.29a, we can connect A and $\overline{B}$ to an OR gate, as shown in Fig. 3.29b. In a similar way, instead of connecting A, $\overline{B}$, and $\overline{C}$ to a bubbled OR gate, we have connected $\overline{A}$, B, and C to an OR gate. In short, Fig. 3.29b is equivalent to Fig. 3.29a.

The next step toward a NOR-NOR circuit is to convert Fig. 3.29b into Fig. 3.29c, which is done by sliding the bubbles to the left from the output gate to the input gates. This changes the input OR gates to NOR gates. The final step is to use a NOR gate on the output to produce Y instead of $\overline{Y}$, as shown in the NOR-NOR circuit of Fig. 3.29d.

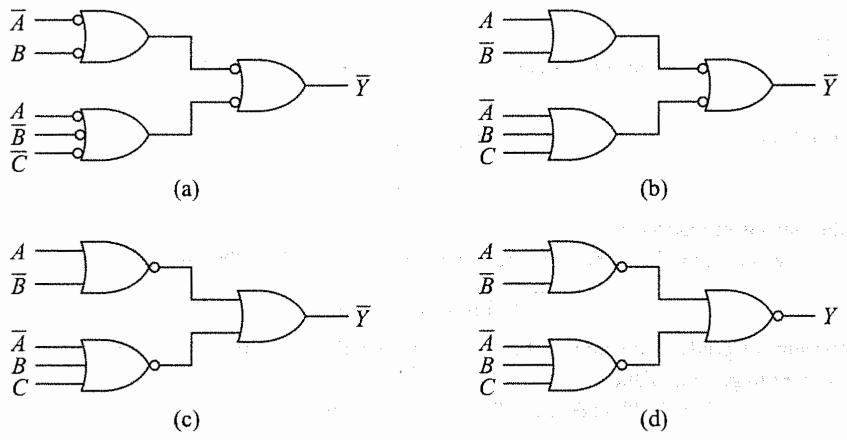


Fig. 3.29 **Deriving the product-of-sums circuit**

From now on, you don't have to go through every step in changing a complementary NAND-NAND circuit to an equivalent NOR-NOR circuit. Instead, you can apply the duality theorem as described in the following.

Duality

An earlier section introduced the duality theorem of Boolean algebra. Now we are ready to apply this theorem to logic circuits. Given a logic circuit, we can find its dual circuit as follows: Change each AND gate to an OR gate, change each OR gate to an AND gate, and complement all input-output signals. An equivalent statement of duality is this: Change each NAND gate to a NOR gate, change each NOR gate to a NAND gate, and complement all input-output signals.

Compare the NOR-NOR circuit of Fig. 3.29d with the NAND-NAND circuit of Fig. 3.28d. NOR gates have replaced NAND gates. Furthermore, all input and output signals have been complemented. This is an application of the duality theorem. From now on, you can change a complementary NAND-NAND circuit (Fig. 3.28d) into its dual NOR-NOR circuit (Fig. 3.29d) by changing all NAND gates to NOR gates and complementing all signals.

Points to Remember

Here is a summary of the key ideas in the preceding discussion:

1. Convert the truth table into a Karnaugh map. After grouping the 1s, write the sum-of-products equation and draw the NAND-NAND circuit. This is the sum-of-products solution for Y.
2. Complement the Karnaugh map. Group the 1s, write the sum-of-products equation, and draw the NAND-NAND circuit for $\overline{Y}$. This is the complementary NAND-NAND circuit.
3. Convert the complementary NAND-NAND circuit to a dual NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals. What remains is the product-of-sums solution for Y.
4. Compare the NAND-NAND circuit (Step 1) with the NOR-NOR circuit (Step 3). You can use whichever circuit you prefer, usually the one with fewer gates.

Example 3.10 Show the sum-of-products and product-of-sums circuits for the Karnaugh map of Fig. 3.30a.

Solution The Boolean equation for Fig. 3.30a on the next page is

$$Y = A + BC\overline{D}$$

Figure 3.30b is the sum-of-products circuit.

After complementing and simplifying the Karnaugh map, we get Fig. 3.30c. The Boolean equation for this is

$$\overline{Y} = \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{A}D$$

Figure 3.30d is the sum-of-products circuit for the $\overline{Y}$. As shown earlier, we can convert the dual circuit into a NOR-NOR equivalent circuit to get Fig. 3.30e.

The two design choices are Fig. 3.30b and 3.30e. Figure 3.30b is simpler.

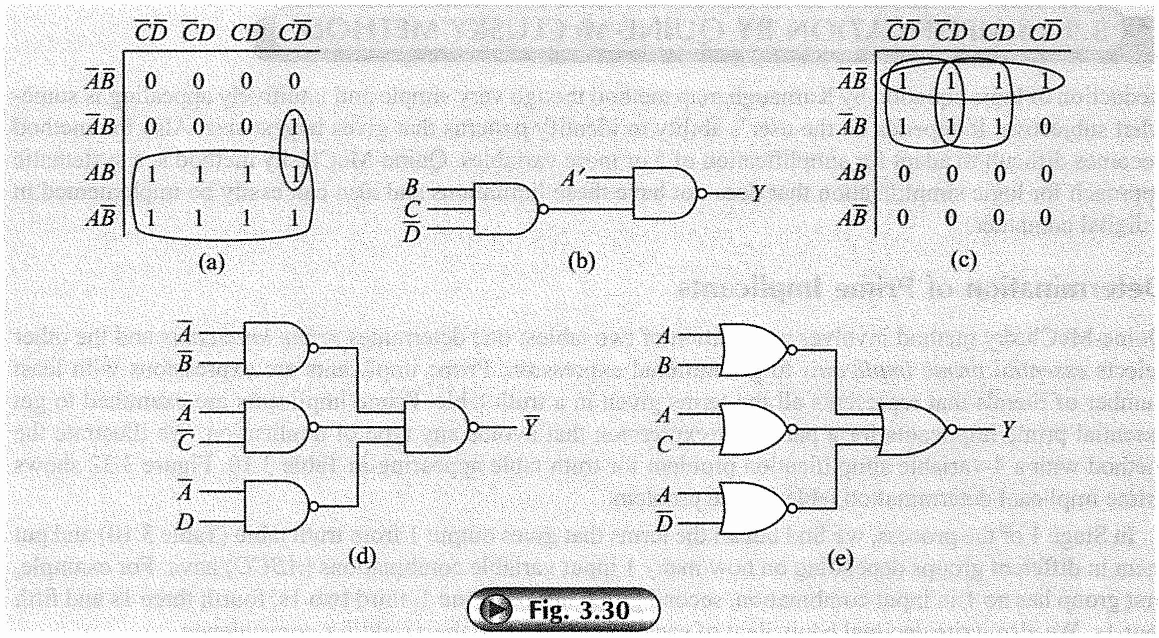


Fig. 3.30

Example 3.11 Give simplest POS form of Karnaugh map shown in Fig. 3.30a by grouping zeros.

Solution Refer to grouping of zeros as shown in Fig. 3.31a. Three groups cover all the zeros that give three sum terms. The first group has A' and C' constant within the group that gives sum term $(A + C)$. Group 2 has A' and D constant giving sum term $(A + D')$. Group 3 has A' and B' constant generating $(A + B)$ as sum term.

The final solution is thus product of these three sum terms and expressed as

$$Y = (A + B)\,(A + C)\,(A + D')$$

Note that, the above relation can be realized by OR-AND circuit or NOR-NOR (Fig. 3.30e) circuit.

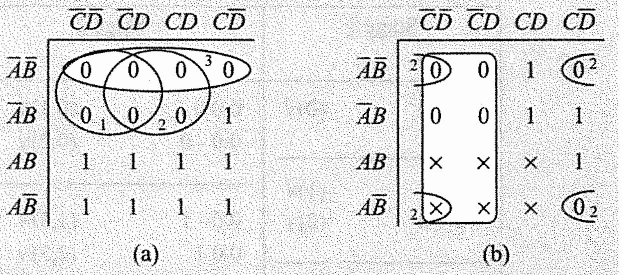


Fig. 3.31 **Simplification by grouping zeros**

Example 3.12 Give simplest POS form of Karnaugh map shown in Fig. 3.31b by grouping zeros.

Solution In a Karnaugh map if don't care conditions exist, we may consider them as zeros if that gives larger group size. This in turn reduces number of literals in the sum term. Refer to grouping of zeros in Fig. 3.31b. We require minimum two groups that includes all the zeros and are also largest in sizes. In group 1, only C' is constant that gives only one literal in sum term as C. Group 2 has B' and D' constant giving sum term $(B + D)$. The final solution is thus product of these two sum terms and expressed as

$$Y = C\,(B + D)$$

3.9 SIMPLIFICATION BY QUINE-McCLUSKY METHOD

Reduction of logic equation by Karnaugh map method though very simple and intuitively appealing is somewhat subjective. It depends on the user's ability to identify patterns that gives largest size. Also the method becomes difficult to adapt for simplification of 5 or more variables. Quine-McClusky method is a systematic approach for logic simplification that does not have these limitations and also can easily be implemented in a digital computer.

Determination of Prime Implicants

Quine-McClusky method involves preparation of two tables; one determines *prime implicants* and the other selects *essential prime implicants* to get minimal expression. Prime implicants are expressions with least number of literals that represents all the terms given in a truth table. Prime implicants are examined to get essential prime implicants for a particular expression that avoids any type of duplication. We illustrate the method with a 4-variable simplification problem for truth table appearing in Table 3.10. Figure 3.32 shows prime implicant determination table for the problem.

In Stage 1 of the process, we find out all the terms that gives output 1 from truth table (Table 3.10) and put them in different groups depending on how many 1 input variable combinations (*ABCD*) have. For example, first group has no 1 in input combination, second group has only one 1, third two 1s, fourth three 1s and fifth four 1s. We also write decimal equivalent of each combination to their right for convenience.

In Stage 2, we first try to combine first and second group of Stage 1, on a member to member basis. The rule is to see if only one binary digit is differing between two members and we mark that position by

Stage 1		Stage 2		Stage 3	
ABCD		*ABCD*		*ABCD*	
0 0 0 0	(0)√	0 0 0 -	(0,1)√	0 0 - -	(0,1,2,3)
		0 0 - 0	(0,2)√	0 0 - -	(0,2,1,3)
0 0 0 1	(1)√	0 0 - 1	(1,3)√	- 0 1 -	(2,10,3,11)
0 0 1 0	(2)√	0 0 1 -	(2,3)√	1 - 1 -	(10,11,14,15)
		- 0 1 0	(2,10)√	1 - 1 -	(10,14,11,15)
0 0 1 1	(3)√	- 0 1 1	(3,11)√	1 1 - -	(12,13,14,15)
1 0 1 0	(10)√	1 0 1 -	(10,11)√	1 1 - -	(12,14,13,15)
1 1 0 0	(12)√	1 - 1 0	(10,14)√		
		1 1 0 -	(12,13)√		
1 0 1 1	(11)√	1 1 - 0	(12,14)√		
1 1 0 1	(13)√	1 - 1 1	(11,15)√		
1 1 1 0	(14)√	1 1 - 1	(13,15)√		
1 1 1 1	(15)√	1 1 1 -	(14,15)√		

Fig. 3.32 Determination of prime implicants

'–'. This means corresponding variable is not required to represent those members. Thus (0) of first group combines with (1) of second group to form (0,1) in Stage 2 and can be represented by $A'B'C'$ (0 0 0 –). The logic of this representation comes from the fact that minterm $A'B'C'D'$ (0) and $A'B'C'D$ (1) can be combined as $A'B'C'(D' + D) = A'B'C'$. We proceed in the same manner to find rest of the combinations in successive groups of Stage 1 and table them in Fig. 3.32. Note that, we need not look beyond successive groups to find such combinations as groups that are not adjacent, differ by more than one binary digit. Also note that each combination of Stage 2 can be represented by three literals. All the members of particular stage, which finds itself in at least one combination of next stage are tick (√) marked. This is followed for Stage 1 terms as well as terms of other stages.

In Stage 3, we combine members of different groups of Stage 2 in a similar way. Now it will have two '–' elements in each combination. This means each combination requires two literals to represent it. For example (0,1,2,3) is represented by $A'B'$ (0 0 – –). There are three other groups in Stage 3; (2,10,3,11) represented by $B'C$, (10,14,11,15) by AC and (12,13,14,15) by AB. Note that, (0,2,1,3), (10,11,14,15) and (12,14,13,15) get represented by $A'B$, AC and AB respectively and do not give any new term.

There is no Stage 4 for this problem as no two members of Stage 3 has only one digit changing among them. This completes the process of determination of prime implicants. The rule is all the terms that are not ticked at any stage is treated as prime implicants for that problem. Here, we get four of them from Stage 3, namely $A'B'$, $B'C$, AC, AB and none from previous stage as all the terms there are ticked (√).

Selection of Prime Implicants

Once we are able to determine prime implicants that covers all the terms of a truth table we try to select essential prime implicants and remove redundancy or duplication among them. For this, we prepare a table as shown in Table 3.11 that along the row lists all the prime implicants and along columns lists all minterms. The cross-point of a row and column is ticked if the term is covered by corresponding prime implicant. For example, terms 0 and 1 are covered by $A'B'$ only while 2 and 3 are covered by both $A'B'$ and $B'C$ and the corresponding cross-points are ticked. This way we complete the table for rest of the terms.

Table 3.11

	0	*1*	*2*	*3*	*10*	*11*	*12*	*13*	*14*	*15*
$A'B'$ (0,1,2,3)	√	√	√	√						
$B'C$ (2,3,10,11)			√	√	√	√				
AC (10,11,14,15)					√	√			√	√
AB (12,13,14,15)							√	√	√	√

Selection of essential prime implicants from this table is done in the following way. We find minimum number of prime implicants that covers all the minterms. We find $A'B'$ and AB cover terms that are not covered by others and they are essential prime implicants. $B'C$ and AC among themselves cover 10,11 which are not covered by others. So, one of them has to be included in the list of essential prime implicants making it three. And the simplified representation of truth table given in Table 3.10 is one of the following

$$Y = A'B' + B'C + AB \text{ or } Y = A'B' + AC + AB$$

Simplification of the same truth table by Karnaugh map method is shown in Fig. 3.28a and we see the results are the same.

Now, how do you compare the complexity of this approach with Karnaugh map groupings? Yes, this method is more tedious and monotonous compared to Karnaugh map method and people don't prefer it for simplification problems with smaller number of variables. However, as we have mentioned before, for simplification problems with large number of variables Quine-McClusky method can offer solution and Karnaugh map does not.

Example 3.13 Give simplified logic equation of Table 3.6 by Quine-McClusky method.

Solution Tables that determine prime implicants and selects essential prime implicants are shown in Figs. 3.33a and 3.33b respectively. We find both the prime implicants are essential prime implicants. The simplified logic equation thus is expressed as

$$Y = AB + BC'$$

Note that, we got the same expression by simplification entered variable map shown in Figs. 3.22a and 3.22b.

Stage 1		Stage 2	
A B C		*A B C*	
0 1 0	(2)√	- 1 0	(2,6)
1 1 6	(6)√	1 1 -	(6,7)
1 1 1	(7)√		

(a)

	2	6	7
BC' (2,6)	√	√	
AB (6,7)		√	√

(b)

Fig. 3.33 **Simplification by Quine-McClusky method for Example 3.14**

SELF-TEST

15. What is a prime implicant?
16. What are the advantages of Quine-McClusky method?

3.10 HAZARDS AND HAZARD COVERS

In past few sections we have discussed in detail various simplification techniques that give minimal expression for a logic equation which in turn requires minimum hardware for realization of that. It may sound off-beat, but due to some practical problems, in certain cases we may prefer to include more terms than given by simplification techniques. The discussion so far considered gates generating outputs instantaneously. But practical circuits always offer finite propagation delay though very small, in nanosecond order. This gives rise to several *hazards* and *hazard covers* are additional terms in an equation that prevents occurring of them. In this section, we discuss this problem and its solution.

Static-1 Hazard

This type of hazard occurs when $Y = A + A'$ type of situation appears for a logic circuit for certain combination of other inputs and A makes a transition $1 \rightarrow 0$. An $A + A'$ condition should always generate 1 at the output, i.e. static-1. But the NOT gate output (Fig. 3.34a) takes finite time to become 1 following $1 \rightarrow 0$ transition of A. Thus for the OR gate there are two zeros appearing at its input for that small duration, resulting a 0 at its output (Fig. 3.34b). The width of this zero is in nanosecond order and is called a glitch. For combinational circuits it may go unnoticed but in sequential circuit, more particularly in asynchronous sequential circuit (discussed in Chapter 11) it may cause major malfunctioning.

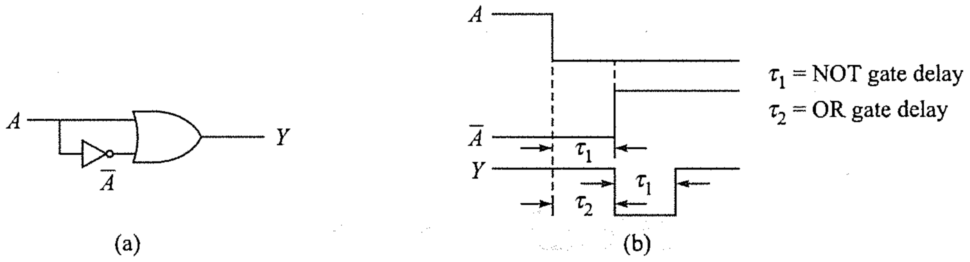
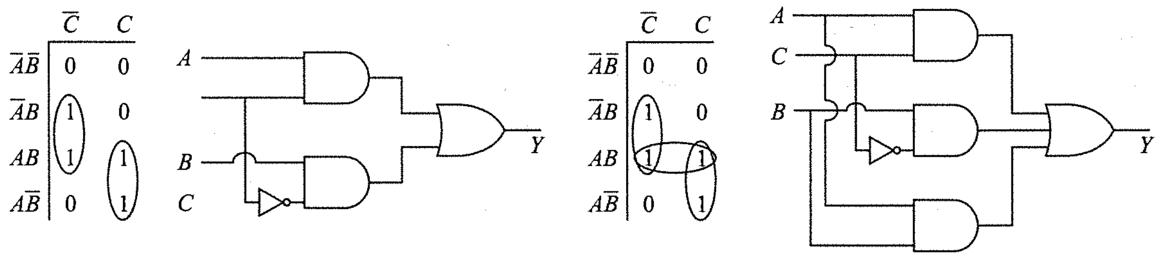


Fig. 3.34 Static-1 hazard

To discuss how we cover static-1 hazard let's look at one example. Refer to Karnaugh map shown in Fig. 3.35a, which is minimally represented by $Y = BC' + AC$. The corresponding circuit is shown in Fig. 3.35b. Consider, for this circuit input $B = 1$ and $A = 1$ and then C makes transition $1 \rightarrow 0$. The output shows glitch as discussed above. Consider another grouping for the same map in Fig. 3.35c. This includes one additional term AB and now output $Y = BC' + AC + AB$. The corresponding circuit diagram is shown in Fig. 3.35d. This circuit though require more hardware than minimal representation, is hazard free. The additional term AB ensures $Y = 1$ for $A = 1, B = 1$ through the third input of final OR gate and a $1 \rightarrow 0$ transition at C does not affect output. Note that, there is no other hazard possibility and inclusion of hazard cover does not alter the truth table in anyway.



(a) $Y = B\overline{C} + AC$ (b) Circuit with static-1 hazard (c) $Y = B\overline{C} + AC + AB$ (d) Hazard free circuit

Fig. 3.35 Static-1 hazard and its cover

Again, a NAND gate with A and A' connected at its input for certain input combination will give static-1 hazard when A makes a transition $0 \rightarrow 1$ and requires hazard cover.

Static-0 Hazard

This type of hazard occurs when $Y = A.A'$ kind of situation occurs in a logic circuit for certain combination of other inputs and A makes a transition $0 \rightarrow 1$. An $A.A'$ condition should always generate 0 at the output, i.e. static-0. But the NOT gate output (Fig. 3.36a) takes finite time to become 0 following a $0 \rightarrow 1$ transition of A. Thus for final AND gate there are two ones appearing at its input for a small duration resulting a 1 at its output (Fig. 3.36b). This $Y = 1$ occurs for a very small duration (few nanosecond) but may cause malfunctioning of sequential circuit.

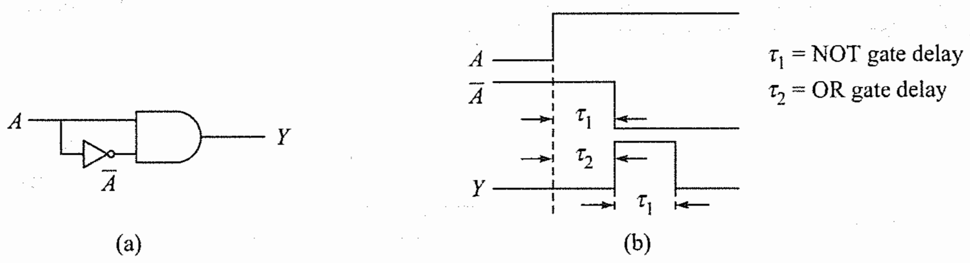
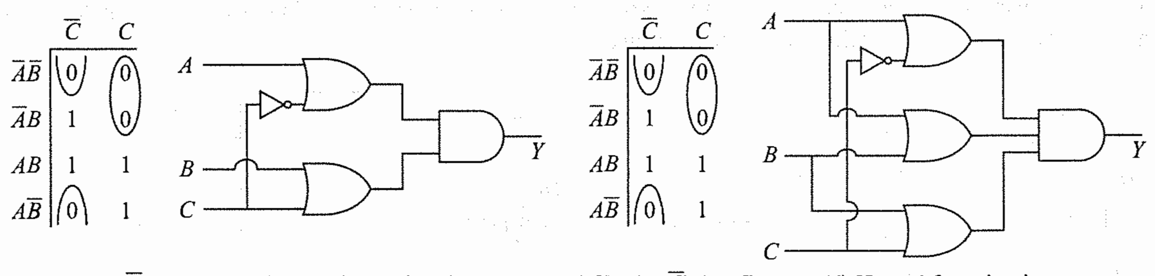


Fig. 3.36 Static-0 hazard

Again, we take an example to discuss how we can prevent static-0 hazard. We use the same truth table as shown in Fig. 3.35a but form group of 0s such that a POS form results. Figure 3.37a shows the minimal cover in POS form that gives $Y = (B + C)(A + C')$ and corresponding circuit in Fig. 3.37b. But if $B = 0$, $A = 0$ and C makes a transition $0 \rightarrow 1$ there will be static-0 hazard occurring at output. To prevent this we add one additional group, i.e. one more sum term $(A + B)$ as shown in Fig. 3.37c and the corresponding circuit is shown in Fig. 3.37d. The additional term $(A + B)$ ensures $Y = 0$ for $A = 0$, $B = 0$ through the third input of final AND gate and a $0 \rightarrow 1$ transition at C does not affect output. Again note that for this circuit there is no other hazard possibility and inclusion of hazard cover does not alter the truth table in anyway.



(a) $Y = (B+\bar{C})(A+C)$ (b) Circuit with static-0 hazard (c) $Y = (B+\bar{C})(A+C)(A+B)$ (d) Hazard free circuit

Fig. 3.37 Static-1 hazard and its cover

Also note, a NOR gate with A and A' connected at its input for certain input combination will give static-0 hazard when A makes a transition $1 \rightarrow 0$ and requires hazard cover.

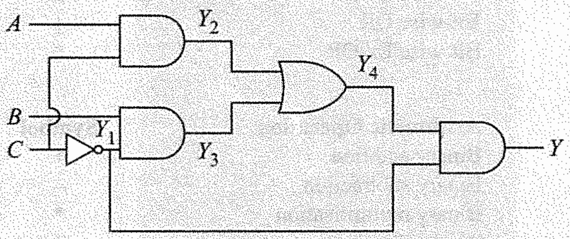
Dynamic Hazard

Dynamic hazard occurs when circuit output makes multiple transitions before it settles to a final value while the logic equation asks for only one transition. An output transition designed as $1 \rightarrow 0$ may give $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ when such hazard occurs and a $0 \rightarrow 1$ can behave like $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$. The output of logic equation in dynamic hazard degenerates into $Y = A + A'.A$ or $Y = (A + A').A$ kind of relations for certain combinations of the other input variables. As shown by these equations, these occur in multilevel circuits having implicit static-1 and/or static-0 hazards. Providing covers to each one of them dynamic hazard can be prevented.

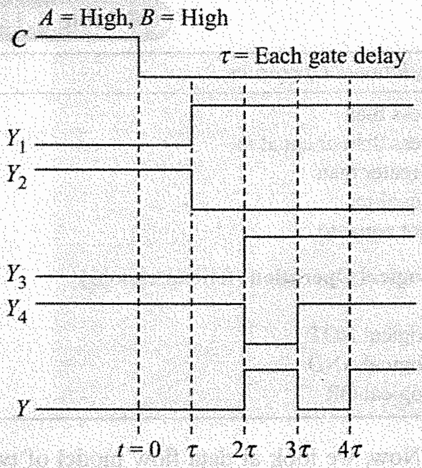
Example 3.14 Check if the circuit shown in Fig. 3.38a exhibit dynamic hazard. Show how output varies with time if dynamic hazard occurs. Consider all the gates have equal propagation delay of τ nanosecond. Also mention how the hazard can be prevented.

Solution The logic circuit can be written in the form of equation as $Y = (A.C + B.C').C'$. Clearly for $A = 1, B = 1$ we get $Y = (C + C').C'$ which shows potential dynamic hazard with an implicit static-1 hazard. Figure 3.38b shows how a transition $1 \rightarrow 0$ at input C for $AB = 11$ causes dynamic hazard at the output.

The hazard can be prevented by using an additional two input AND gate fed by input A and B and replacing two input OR gate by a three input OR gate. The additional (third) input of OR gate will be fed by output of the new AND gate.



(a)



(b)

Fig. 3.38 **Example of dynamic hazard**

SELF-TEST

17. What is static-0 hazard?
18. What is dynamic hazard?

3.11 HDL IMPLEMENTATION MODELS

We continue our discussion of Verilog HDL description for a digital logic circuit from Chapter 2, Section 2.5. We have seen how structural gate level modeling easily maps a digital circuit and replicates graphical symbolic representation. We have also seen how a simple test bench can be prepared to test a designed circuit. There, we generated all possible combinations of input variables and passed it to a circuit to be tested by providing realistic gate delays. We'll follow similar test bench but more ways to describe a digital circuit in this and subsequent chapters.

Dataflow Modeling

Gate level modeling, though very convenient to get started with an HDL, consumes more space in describing a circuit and is unsuitable for large, complex design. Verilog provides a keyword **assign** and a set of operators (partial list given in Table 3.11, some operations will be explained in later chapters) to describe a circuit through its behavior or function. Here, we do not explicitly need to define any gate structure using **and**, **or** etc. and it is not necessary to use intermediate variables through **wire** showing gate level interconnections. Verilog compiler handles this while compiling such a model. All **assign** statements are concurrent, i.e. order in which they appear do not matter and also continuous, i.e. any change in a variable in the right hand side will immediately effect left hand side output.

Table 3.12 A Partial List of Verilog Operator

Relational Operation	*Symbol*	*Bit-wise Operation*	*Symbol*
Less than	<	Bit-wise NOT	~
Less than or equal to	<=	Bit-wise AND	&
Greater than	>	Bit-wise OR	|
Equal to	==	Bit-wise Ex-OR	^
Not equal to	!=		
Logical Operation (for expressions)	**Symbol**	**Arithmetic Operation**	**Symbol**
		Binary addition	+
Logical NOT	!	Binary subtraction	–
Logical AND	&&	Binary multiplication	*
Logical OR	||	Binary division	/

Now, we look at data flow model of two circuits shown in Fig. 2.17a and Fig. 2.38. We compare these codes with gate level model code presented in Section 2.5 and note the advantage. We see that data flow model resembles a logic equation and thus gives a more crisp representation.

```
module fig2_24a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  assign Y=(A&B)|(C&D);
  //One statement is enough
endmodule
```

```
module testckt(a,b,c,x,y);
  input a,b,c;
  output x,y;
  assign x=~((a|b)|c); // NOR through NOT-OR
  assign y=~((a|b)&(b|c)); /* NAND by NOT-
  AND*/
endmodule
```

Behavioral Modeling

In a behavioral model, statements are executed sequentially following algorithmic description. It is ideally suited to describe a sequential logic circuit. However, it is also possible to describe combinatorial circuits with this but may not be a preferred model in most of the occasions. It always uses **always** keyword followed by a sensitivity list. The procedural statements following **always** is executed only if any variable within sensitivity list changes its value. Procedure assignment or output variables within **always** must be of register type, defined by **reg** which unlike **wire** is not continuously updated but only after a new value is assigned to it. Note that, **wire** variables can only be read and not assigned to in any procedural block, also it cannot store any value and must be continuously driven by output or assign statement.

Now, let us try to write behavioral code for circuit given in Fig. 2.17a. We note that, $Y = AB + CD$, i.e. $Y = 1$ if $AB = 11$ or if $CD = 11$, otherwise $Y = 0$. We use **if...else if...else** construct to describe this circuit. Here, the conditional expression after **if**, if true executes one set of instructions else executes a different set following **else** or none at all.

```
module fig2_24a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  reg Y;       /* Y is output after procedural assignment within always
                 block, hence as reg.*/
always @ (A or B or C or D) //A,B,C,D form sensitivity list, note keyword
                                  or
  if ((A==1)&&(B == 1)) // If A,B both are 1
     Y=1;                // Assignment through equal sign not keyword assign
  else if ((C==1)&&(D == 1)) // if C,D both are 1
     Y=1;
  else                   // for all other combinations of A,B,C,D
     Y=0;
endmodule
```

You can compare how logic circuit described in Fig. 2.17a is realized in Verilog HDL following three different models two of which are described in this chapter and one in previous chapter. One might find data flow model more convenient to use for combinatorial circuits. We'll learn more about it in subsequent chapters.

Example 3.15 Realize the truth table shown in Karnaugh Map of Fig. 3.19 using data flow model.

Solution The simplified logic equation of this is given by equation 3.29 as $Y = C' + A'D' + B'D'$. We use this in **assign** statement to get HDL description.

```
module fig3_20a(A,B,C,D,Y);
 input A,B,C,D;
 output Y;
 assign Y= ~C | (~A & ~D) | (~B & ~D); // ~ operator has higher precedence
endmodule
```

Note that, ~ operator has higher precedence over & and |; while & and | are at same level. To avoid confusion and improve readability it is always advised to use parentheses (...) that has second highest precedence below bit select [...].

The test bench for all the examples described in this chapter can be prepared in a manner similar to what is described in Chapter 2. A simpler HDL representation to prepare a test bench will be discussed in Chapter 6.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Get a minimized expression for $Y = F(A, B, C) = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}BC + A\overline{B}C$

Solution We can solve this using Boolean Algebra, Karnaugh Map, Entered Variable Map and QM Algorithm.

In Method-1 We take help of Boolean Algebra for minimization. We see that $\overline{A}\,\overline{B}C$ can be combined with all three terms using distributive law (Eq. 3.5)

Since, in Boolean algebra $X = X + X + X$ (extending Eq. 3.7) we can write

$$Y = \overline{A}\,\overline{B}\,\overline{C} + (\overline{A}\,\overline{B}C + \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}C) + \overline{A}BC + A\overline{B}C$$

From associative law (Eq. 3.3)

$$Y = (\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C) + (\overline{A}\,\overline{B}C + \overline{A}BC) + (\overline{A}\,\overline{B}C + A\overline{B}C)$$

From distributive law (Eq. 3.5)

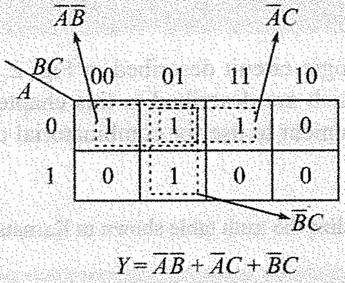
$$Y = \overline{A}\,\overline{B}(\overline{C} + C) + \overline{A}C(\overline{B} + B) + \overline{B}C(\overline{A} + A)$$

From Eq. 3.9, since $X + \overline{X} = 1$

$$Y = \overline{A}\,\overline{B} \cdot 1 + \overline{A}C \cdot 1 + \overline{B}C \cdot 1$$
$$= \overline{A}\,\overline{B} + \overline{A}C + \overline{B}C \quad \text{(since, } X \cdot 1 = X \text{ from Eq. 3.10)}$$

In Method-2, we use Karnaugh Map for minimization. Fig. 3.39 shows the solution by this method.

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



$\overline{A}\,\overline{B}$
$\overline{A}C$
BC
A
00
01
11
10
0
1
1
1
1
0
1
0
1
0
0
$\overline{B}C$
$Y = \overline{A}\,\overline{B} + \overline{A}C + \overline{B}C$

Fig. 3.39 **Solution using Karnaugh Map**

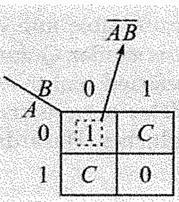
Note how one term is common in three groups formed and the similarity with Method-1 solution.

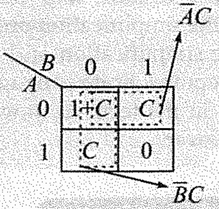
In Method-3, we use Entered Variable Map for minimization. Figure 3.40 shows the solution by this method.

Since $1 = C + \overline{C}$, we need a separate group for $AB = 00$ as $\overline{C}$ is not explained by other two groups. We use C embedded in 1 to make other two groups bigger and reduce the number of literals, and thus minimize the expression.

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

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





$Y = \overline{A}\overline{B} + \overline{A}C + \overline{B}C$

Fig. 3.40 **Solution using Entered Variable Map**

In Method-4, we use QM algorithm for minimization. Fig. 3.41 shows prime implicants and essential prime implicants. The final solution is arrived at by combining essential prime implicants.

Stage 1 ABC			Stage 2 ABC	
000	(0)	√	00–	(0, 1)
001	(1)	√	0–1	(1, 3)
			–01	(1, 5)
011	(3)	√		
101	(5)	√		

Prime implicants only from stage 2.
They are:
00–($A'B'$), 0–1 ($A'C$) and –01 ($B'C$)

	0	1	3	5
$A'B'$	√	√		
$A'C$		√	√	
$B'C$		√		√

All are essential
$Y = A'B' + A'C + B'C$

Fig. 3.41 **Solution using QM Algorithm**

SUMMARY

Every Boolean equation has a dual form obtained by changing OR to AND, AND to OR, 0 to 1, and 1 to 0. With Boolean algebra you may be able to simplify a Boolean equation, which implies a simplified logic circuit.

Given a truth table, you can identify the fundamental products that produce output 1s. By ORing these products, you get a sum-of-products equation for the truth table. A sum-of-products equation always results in an AND-OR circuit or its equivalent NAND-NAND circuit.

The Karnaugh method of simplification starts by converting a truth table into a Karnaugh map. Next. You encircle all the octets, quads, and pairs. This allows you to write a simplified Boolean equation and to draw a simplified logic circuit. When a truth table contains don't-cares, you can treat the don't-cares as 0s or 1s, whichever produces the greatest simplification.

One way to get a product-of-sums circuit is to complement the Karnaugh map and write the simplified Boolean equation for $\overline{Y}$. Next, you draw the NAND-NAND circuit for $\overline{Y}$. Finally, you change the NAND-NAND circuit into a NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals.

Entered variable map maps a truth table into lower dimension space compared to Karnaugh map though the simplification procedure is similar. Quine-McClusky method provides a step-by-step approach for logic simplification and is a preferred tool that involves large number of variables. Practical digital circuit requires finite propagation delay to transfer information from input to output. This often leads to hazards in the form of unwanted glitches. Hazards are prevented by using additional gates serving as hazard cover.

GLOSSARY

- ***chip*** An integrated circuit. A piece of semiconductor material with a micro-miniature circuit on its surface.
- ***consensus theorem*** A theorem that simplifies a Boolean equation removing a redundant consensus theorem.
- ***don't-care condition*** An input-output condition that never occurs during normal operation. Since the condition never occurs, you can use an X on the Karnaugh map. This X can be a 0 or a 1, whichever you prefer.
- ***dual circuit*** Given a logic circuit, you can find it dual as follows. Change each AND (NAND) gate to an OR (NOR) gate, change each OR (NOR) gate to an AND (NAND) gate, and complement all input-output signals.
- ***Entered variable map*** an alternative to Karnaugh map where a variable is placed as output.
- ***Hazard*** unwanted glitches due to finite propagation delay of logic circuit.
- ***Hazard cover*** additional gates in logic circuit preventing hazard.
- ***Quine-McClusky method*** a tabular method for logic simplification.
- ***logic clip*** A device attached to a 14- or 16-pin DIP. The LEDs in this troubleshooting tool indicate the logic states of the pins.
- ***Karnaugh map*** A drawing that shows all the fundamental products and the corresponding output values of a truth table.
- ***octet*** Eight adjacent 1s in a 2×4 shape on a Karnaugh map.
- ***overlapping groups*** Using the same 1 more than once when looping the 1s of a Karnaugh map.
- ***pair*** Two horizontally or vertically adjacent 1s on a Karnaugh map.
- ***product-of-sums equation*** The logical product of those fundamental sums that produce output 1s in the truth table. The corresponding logic circuit is an OR-AND circuit, or the equivalent NOR-NOR circuit.
- ***quad*** Four horizontal, vertical, or rectangular 1s on a Karnaugh map.
- ***redundant group*** A group of 1s on a Karnaugh map that are all part of other groups. You can eliminate any redundant group.
- ***sum-of-products equation*** The logical sum of those fundamental products that produce output 1s in the truth table. The corresponding logic circuit is an AND-OR circuit, or the equivalent NAND-NAND circuit.

PROBLEMS

Section 3.1

3.1 Draw the logic circuit for

$$Y = A\overline{B}C + ABC$$

Next, simplify the equation with Boolean algebra and draw the simplified logic circuit.

3.2 Draw the logic circuit for

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

Use Boolean algebra to simplify the equation. Then draw the corresponding logic circuit.

3.3 In Fig. 3.42a, the output NAND gate acts like a 2-input gate because pins 10 and 11 are tied together. Suppose a logic clip is connected to the 7410. Which of the three gates is defective if the logic clip displays the data of Fig. 3.42b?

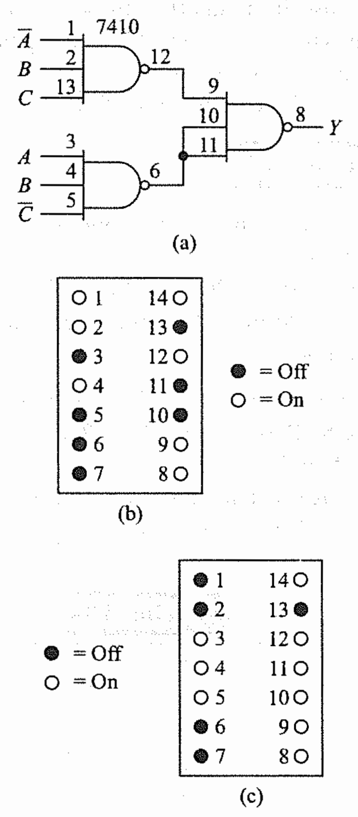


Fig. 3.42

3.4 If a logic clip displays the states of Fig. 3.42c for the circuit of Fig. 3.42a, which of the gates is faulty?

3.5 The circuit of Fig. 3.42a has trouble. If Fig. 3.43 is the timing diagram, which of the following is the trouble:

a. Upper NAND gate is defective.
b. Pin 6 is shorted to +5 V.
c. Pin 9 is grounded.
d. Pin 8 is shorted to +5 V.

Section 3.2

3.6 What is the sum-of-products circuit for the truth table of Table 3.11?

3.7 Simplify the sum-of-products equation in Prob. 3.6 as much as possible and draw the corresponding logic circuit.

3.8 A digital system has a 4-bit input from 0000 to 1111. Design a logic circuit that produces a high output whenever the equivalent decimal input is greater than 13.

3.9 We need a circuit with 2 inputs and 1 output. The output is to be high only when 1 input is high. If both inputs are high, the output is to be low. Draw a sum-of-products circuit for this.

Section 3.3

3.10 Draw the Karnaugh map for Table 3.11.

3.11 Draw the Karnaugh map for Table 3.13.

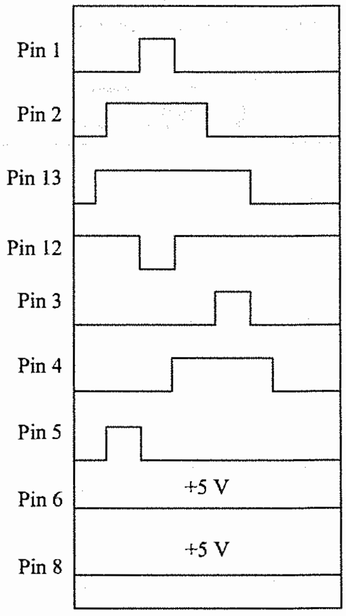


Fig. 3.43

3.12 Show Karnaugh map for equation $Y = F(A, B, C) = \Sigma m(1, 2, 3, 6, 7)$

3.13 Show Karnaugh map for equation $Y = F(A, B, C, D) = \Sigma m(1, 2, 3, 6, 8, 9, 10, 12, 13, 14)$

Section 3.4

3.14 Draw the Karnaugh map for Table 3.11. Then encircle all the octets, quads, and pairs you can find.

3.15 Repeat Prob. 3.14 for Table 3.14.

Section 3.5

3.16 What is the simplified Boolean equation for the Karnaugh map of Table 3.13? The logic circuit?

3.17 Given Table 3.14, use Karnaugh simplification and draw the simplified logic circuit.

3.18 Table 3.15 on the next page shows a special code known as the *Gray code*. For each binary input *ABCD*, there is a corresponding Gray-code output. What is the simplified sum-of-products equation for Y_3? For Y_2? For Y_1? For Y_0? Draw a logic circuit that converts a 4-bit binary input to a Gray-code output.

Section 3.6

3.19 Suppose the last six entries of Table 3.11 are changed to don't-cares. Using the Karnaugh map, show the simplified logic circuit.

3.20 Assume the first six entries of Table 3.13 are changed to don't-cares. What is the simplified logic circuit?

3.21 Suppose the inputs 1010 through 1111 only appear when there is trouble in a digital system. Design a logic circuit that detects the presence of any nibble input from 1010 to 1111.

Section 3.7

3.22 Draw the unsimplified product-of-sums circuit for Table 3.11.

3.23 Repeat Prob. 3.20 for Table 3.13.

3.24 Draw a NOR-NOR circuit for this Boolean expression:

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

3.25 Give SOP form of $Y = F(A, B, C, D) = \Pi M(0, 3, 4, 5, 6, 7, 11, 15)$

3.26 Draw Karnaugh map of $Y = F(A, B, C, D) = \Pi M(0, 1, 3, 8, 9, 10, 14, 15)$

Table 3.13

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

Table 3.14

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

Table 3.15 **Gray Code**

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
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

Section 3.8

3.27 What is the simplified NOR-NOR circuit for Table 3.11?

3.28 Draw the simplified NOR-NOR circuit for Table 3.13.

3.29 Figure 3.44 shows all the input waveforms for the timing diagram of Fig. 3.30e. Draw the waveform for the output Y.

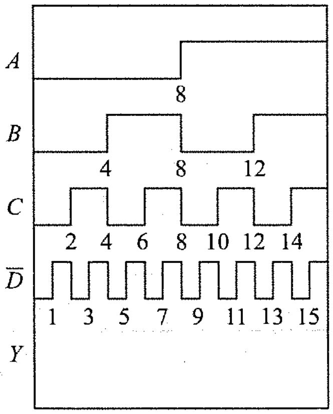


Fig. 3.44

3.30 You are given the following Boolean equation

$$Y = \overline{A}B\overline{C}D + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}C\overline{D}$$

Show the simplified NAND-NAND circuit for this. Also, show the simplified NOR-NOR circuit.

3.31 Table 3.16 is the truth table of *full adder*, a logic circuit with two outputs called the *CARRY* and the *SUM*. What is the simplified NAND-NAND circuit for the *CARRY* output? For the *SUM* output?

3.32 Repeat Prob. 3.27 using NOR-NOR circuits

Table 3.16 **Full-Adder Truth Table**

A	B	C	*Carry*	*Sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.33 Simplify to give POS form by grouping zeros in Karnaugh map for equation given in problem 3.27.

3.34 Simplify to give POS form by grouping zeros in Karnaugh map for equation given in problem 3.28.

Sections 3.9 and 3.10

3.35 Get simplified expression of $Y = F(A, B, C, D) = \Sigma\, m(1, 2, 8, 9, 10, 12, 13, 14)$ using Quine-McClusky method.

3.36 Get simplified expression of $Y = F(A, B, C, D, E) = \Sigma\, m(0, 1, 2, 3, 4, 5, 12, 13, 14, 26, 27, 28, 29, 30)$ using Quine-McClusky method.

3.37 For the following Karnaugh map give SOP and POS form that do not show static-0 or static-1 hazard.

	$\bar{C}$	C
$\bar{A}\bar{B}$	1	1
$\bar{A}B$	0	0
AB	1	0
$A\bar{B}$	1	0

3.38 Verify with timing diagram if the following circuit shows dynamic hazard.

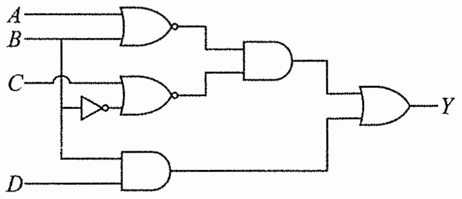


Fig. 3.45

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to verify De Morgan's theorems

Theory: De Morgan's two theorems are

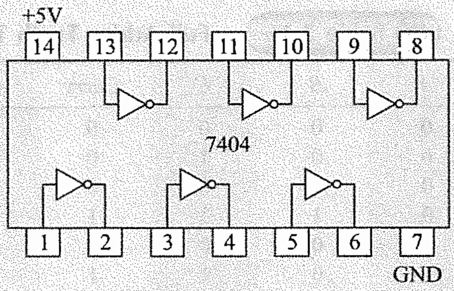
$$(A + B)' = A' \cdot B'$$

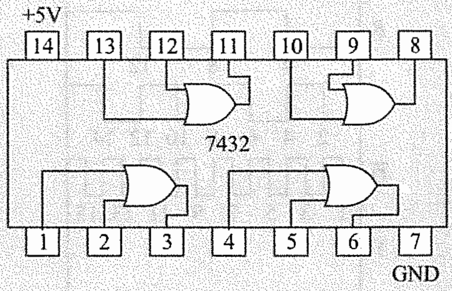
and $$(A \cdot B)' = A' + B'$$

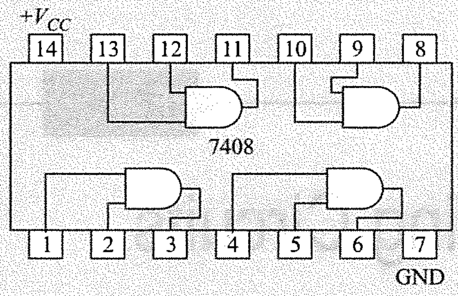
NAND gate and NOR gate can be used to generate the left hand side of the two equations while NOT gate, AND gate and OR gate can be used to generate the right hand side.

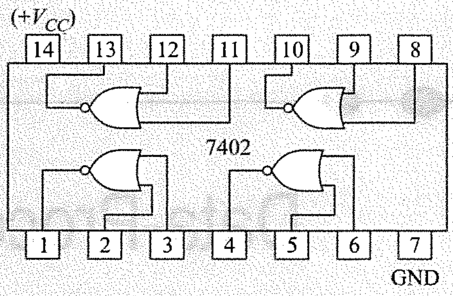
Apparatus: 5 V DC Power supply, Multi-meter, and Bread Board

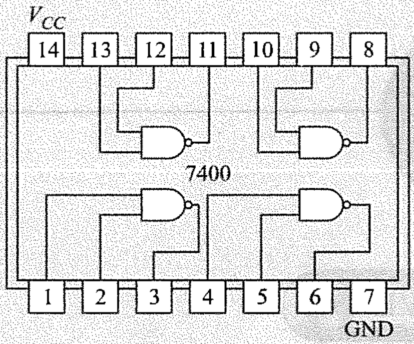
Work element: Verify the truth table of IC 7404, 7408, 7432, 7402 and 7400. Interconnect them in such a manner so that right hand sides of the equations are implemented. Find its truth table. Compare it with truth table of NOR and NAND gates.

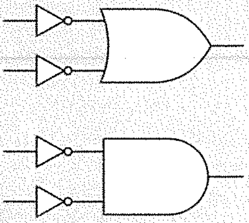












Answers to Self-tests

1. False
2. $Y = AB + AC$
3. $Y = Q$
4. Four, eight
5. False
6. A Karnaugh map is a visual display of the fundamental products needed for a sum-of-products solution.
7. Sixteen
8. Pair
9. Eight
10. $Y = \overline{A}B\overline{C}\overline{D} + AB\overline{C}\overline{D} + \overline{A}BC\overline{D} + ABC\overline{D}$. Simplify as $Y = B\overline{D}$
11. A don't-care condition is an input condition that never occurs during normal operations, and it is indicated with an X.
12. An X can be used to create pairs, quads, octets, etc.
13. A product-of-sums expression leads directly to an OR-AND circuit.
14. Change all NAND gates to NOR gates, and complement all signals (see Example 3.10).
15. Prime implicants are expressions with least number of literals that represents all the terms given in a truth table.
16. Systematic, step-by-step approach that can be implemented in a digital computer and providing solution for any number of variables.
17. A logic high pulse of very short duration when output should be at logic low.
18. Dynamic hazard occurs when circuit output makes multiple transitions before it settles while the logic equation asks for only one transition.

Data-Processing Circuits

4

OBJECTIVES

- Determine the output of a multiplexer or demultiplexer based on input conditions.
- Find, based on input conditions, the output of an encoder or decoder.
- Draw the symbol and write the truth table for an exclusive-OR gate.
- Explain the purpose of parity checking.
- Show how a magnitude comparator works.
- Describe a ROM, PROM, EPROM, PAL, and PLA.

This chapter is about logic circuits that process binary data. We begin with a discussion of multiplexers, which are circuits that can select one of many inputs. Then you will see how multiplexers are used as a design alternative to the sum-of-products solution. This will be followed by an examination of a variety of circuits, such as demultiplexers, decoders, encoders, exclusive-OR gates, parity checkers, magnitude comparator, and read-only memories. The chapter ends with a discussion of programmable logic arrays and relevant HDL concepts.

4.1 MULTIPLEXERS

Multiplex means *many into one*. A *multiplexer* is a circuit with many inputs but only one output. By applying control signals, we can steer any input to the output. Thus it is also called a *data selector* and control inputs are termed select inputs. Figure 4.1a illustrates the general idea. The circuit has n input signals, m control signals and 1 output signal. Note that, m control signals can select at the most 2^m input signals thus $n \le 2^m$.

The circuit diagram of a 4-to-1 multiplexer is shown in Fig. 4.1c and its truth table in Fig. 4.1b. Depending on control inputs A, B one of the four inputs D_0 to D_3 is steered to output Y.

Let us write the logic equation of this circuit. Clearly, it will give a SOP representation, each AND gate generating a product term, which finally are summed by OR gate. Thus,

$$Y = A'B'.D_0 + A'B.D_1 + AB'.D_2 + AB.D_3$$

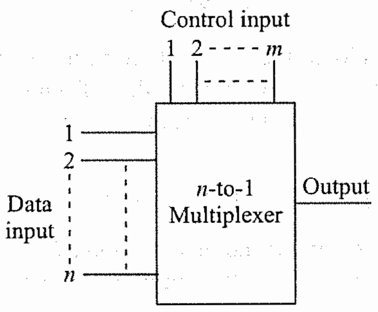
If $A = 0, B = 0,$ $\quad Y = 0'0'.D_0 + 0'.0.D_1 + 0.0'.D_2 + 0.0.D_3$

or, $\quad Y = 1.1.D_0 + 1.0.D_1 + 0.1.D_2 + 0.0.D_3$

or, $\quad Y = D_0$

In other words, for $AB = 00$, the first AND gate to which D_0 is connected remains active and equal to D_0 and all other AND gate are inactive with output held at logic 0. Thus, multiplexer output Y is same as D_0. If $D_0 = 0$, $Y = 0$ and if $D_0 = 1$, $Y = 1$.

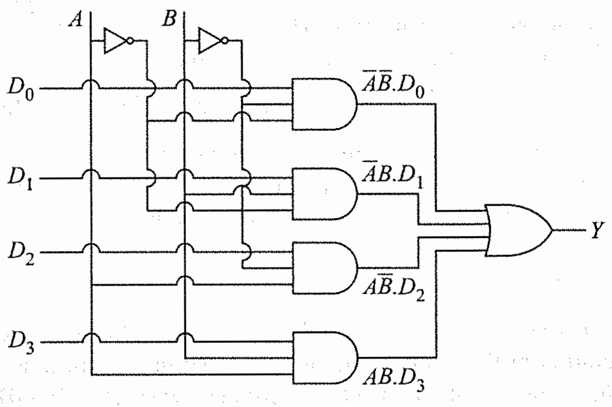
Similarly, for $AB = 01$, second AND gate will be active and all other AND gates remain inactive. Thus, output $Y = D_1$. Following same procedure we can complete the truth table of Fig. 4.1b.



(a)

A	B	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b)



(c)

Fig. 4.1 **(a) Multiplexer block diagram, (b) 4-to-1 multiplexer truth table, (c) Its logic circuit**

Now, if we want 5-to-1 multiplexer how many select lines are required? There is no 5th combination possible with two select lines and hence we need a third select input. Note that, with three we can select up to $2^3 = 8$ data inputs. Commercial multiplexers ICs come in integer power of 2, e.g. 2-to-1, 4-to-1, 8-to-1, 16-to-1 multiplexers. With this background, let us look at a 16-to-1 multiplexer circuit, which may look complex but follows same logic as that of a 4-to-1 multiplexer.

16-to-1 Multiplexer

Figure shows a 16-to-l multiplexer. The input bits are labeled D_0 to D_{15}. Only one of these is transmitted to the output. Which one depends on the value of $ABCD$, the control input. For instance, when

$$ABCD = 0000$$

the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit D_0 is transmitted to the output, giving

$$Y = D_0$$

If D_0 is low, Y is low; if D_0 is high, Y is high. The point is that Y depends only on the value of D_0.
If the control nibble (group of 4-bits) is changed to

$$ABCD = 1111$$

all gates are disabled except the bottom AND gate. In this case, D_{15} is the only bit transmitted to the output, and

$$Y = D_{15}$$

As you can see, the control nibble determines which of the input data bits is transmitted to the output.
Thus we can write output as

$$Y = A'B'C'D'.D_0 + A'B'C'D.D_1 + A'B'CD'.D_2 + \dots + ABCD'.D_{14} + ABCD.D_{15}$$

At this point can we answer, how would an 8 to 1 multiplexer circuit look like? First of all we need three select lines for 8 data inputs. And there will be 8 AND gates each one having four inputs; three from select lines and one from data input. The final output is generated from an OR gate which takes input from 8 AND gates. The equation for this can be written as

$$Y = A'B'C'.D_0 + A'B'C.D_1 + A'BC'.D_2 + A'BC.D_3 + AB'C'.D_4 + AB'C.D_5 + ABC'.D_6 + ABC.D_7$$

Thus, for $ABC = 000$, multiplexer output $Y = D_0$; other AND gates and corresponding data inputs D_1 to D_7 remain inactive. Similarly, for $ABC = 001$, multiplexer output $Y = D_1$, for $ABC = 010$, multiplexer output $Y = D_2$ and finally, for $ABC = 111$, multiplexer output $Y = D_7$.

The 74150

Try to visualize the 16-input OR gate of Fig. 4.2 changed to a NOR gate. What effect does this have on the operation of the circuit? Almost none. All that happens is we get the complement of the selected data bit rather than the data bit itself. For instance, when $ABCD = 0111$, the output is

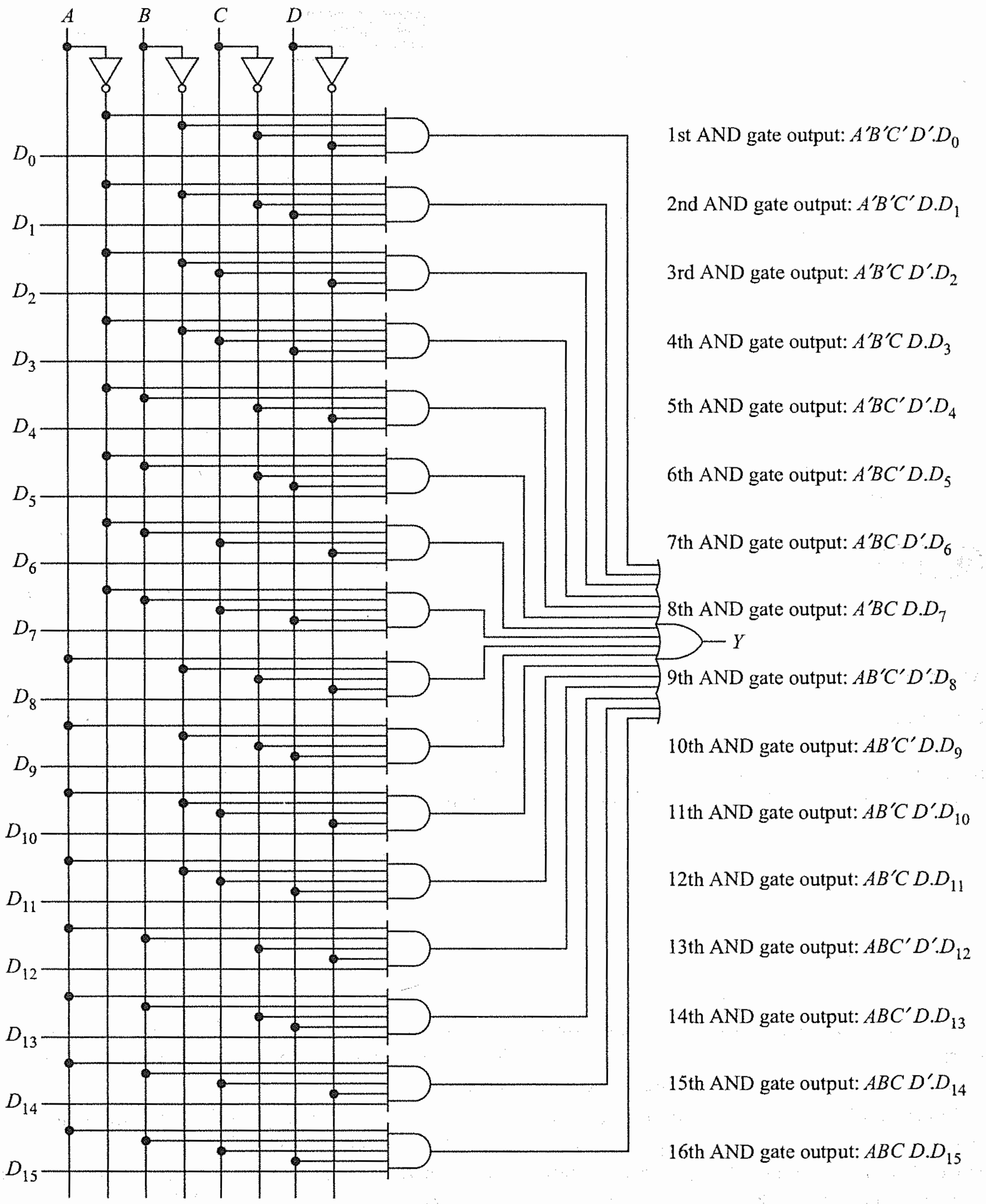
$$Y = \overline{D_7}$$

This is the Boolean equation for a typical transistor-transistor logic (TTL) multiplexer because it has an inverter on the output that produces the complement of the selected data bit.

The 74150 is a 16-to-1 TTL multiplexer with the pin diagram shown in Fig. 4.3. Pins 1 to 8 and 16 to 23 are for the input data bits D_0 to D_{15}. Pins 11, 13, 14, and 15 are for the control bits $ABCD$. Pin 10 is the output; and it equals the complement of the selected data bit. Pin 9 is for the STROBE, an input signal that disables or enables the multiplexer. As shown in Table 4.1, a low strobe enables the multiplexer, so that output Y equals the complement of the input data bit:

$$Y = \overline{D_n}$$

where n is the decimal equivalent of $ABCD$. On the other hand, a high strobe disables the multiplexer and forces the output into the high state. With a high strobe, the value of $ABCD$ doesn't matter.



A
B
C
D
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
D_8
D_9
D_{10}
D_{11}
D_{12}
D_{13}
D_{14}
D_{15}
1st AND gate output: $A'B'C'D'.D_0$
2nd AND gate output: $A'B'C'D.D_1$
3rd AND gate output: $A'B'CD'.D_2$
4th AND gate output: $A'B'CD.D_3$
5th AND gate output: $A'BC'D'.D_4$
6th AND gate output: $A'BC'D.D_5$
7th AND gate output: $A'BCD'.D_6$
8th AND gate output: $A'BCD.D_7$
Y
9th AND gate output: $AB'C'D'.D_8$
10th AND gate output: $AB'C'D.D_9$
11th AND gate output: $AB'CD'.D_{10}$
12th AND gate output: $AB'CD.D_{11}$
13th AND gate output: $ABC'D'.D_{12}$
14th AND gate output: $ABC'D.D_{13}$
15th AND gate output: $ABCD'.D_{14}$
16th AND gate output: $ABCD.D_{15}$

Fig. 4.2 **Sixteen-to-one multiplexer**

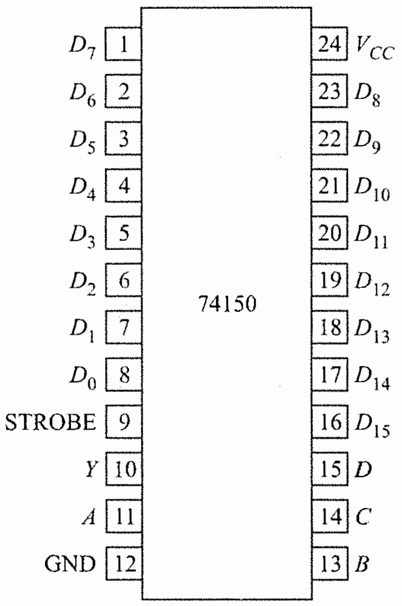


Fig. 4.3 Pinout diagram of 74150

Table 4.1 74150 Truth Table

Strobe	*A*	*B*	*C*	*D*	*Y*
L	*L*	*L*	*L*	*L*	$\overline{D_0}$
L	*L*	*L*	*L*	*H*	$\overline{D_1}$
L	*L*	*L*	*H*	*L*	$\overline{D_2}$
L	*L*	*L*	*H*	*H*	$\overline{D_3}$
L	*L*	*H*	*L*	*L*	$\overline{D_4}$
L	*L*	*H*	*L*	*H*	$\overline{D_5}$
L	*L*	*H*	*H*	*L*	$\overline{D_6}$
L	*L*	*H*	*H*	*H*	$\overline{D_7}$
L	*H*	*L*	*L*	*L*	$\overline{D_8}$
L	*H*	*L*	*L*	*H*	$\overline{D_9}$
L	*H*	*L*	*H*	*L*	$\overline{D_{10}}$
L	*H*	*L*	*H*	*H*	$\overline{D_{11}}$
L	*H*	*H*	*L*	*L*	$\overline{D_{12}}$
L	*H*	*H*	*L*	*H*	$\overline{D_{13}}$
L	*H*	*H*	*H*	*L*	$\overline{D_{14}}$
L	*H*	*H*	*H*	*H*	$\overline{D_{15}}$
H	*X*	*X*	*X*	*X*	*H*

Multiplexer Logic

Digital design usually begins with a truth table. The problem is to come up with a logic circuit that has the same truth table. In Chapter 3, you saw two standard methods for implementing a truth table: the sum-of-products and the product-of-sums solutions. The third method is the *multiplexer solution*. For example, to use a 74150 to implement Table 4.2. Complement each Y output to get the corresponding data input:

$$D_0 = \overline{1} = 0$$
$$D_1 = \overline{0} = 1$$
$$D_2 = \overline{1} = 0$$

and so forth, up to

$$D_{15} = \overline{1} = 0$$

Next, wire the data inputs of 74150 as shown in Fig. 4.4, so that they equal the foregoing values. In other words, D_0 is grounded, D_1 is connected to +5 V, D_2 is grounded, and so forth. In each of these cases, the data input is the complement of the desired Y output of Table 4.2.

Figure 4.4 is the multiplexer design solution. It has the same truth table given in Table 4.2. If in doubt, analyze it as follows for each input condition. When $ABCD$ = 0000, D_0 is the selected input in Fig. 4.4. Since D_0 is low, Y is high. When $ABCD$ = 0001, D_1 is selected. Since D_1 is high, Y is low. If you check the remaining input possibilities, you will see that the circuit has the truth table given in Table 4.2.

Bubbles on Signal Lines

Data sheets often show inversion bubbles on some of the signal lines. For instance, notice the bubble on pin 10, the output of Fig. 4.4. This bubble is a reminder that the output is the complement of the selected data bit.

Table 4.2

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

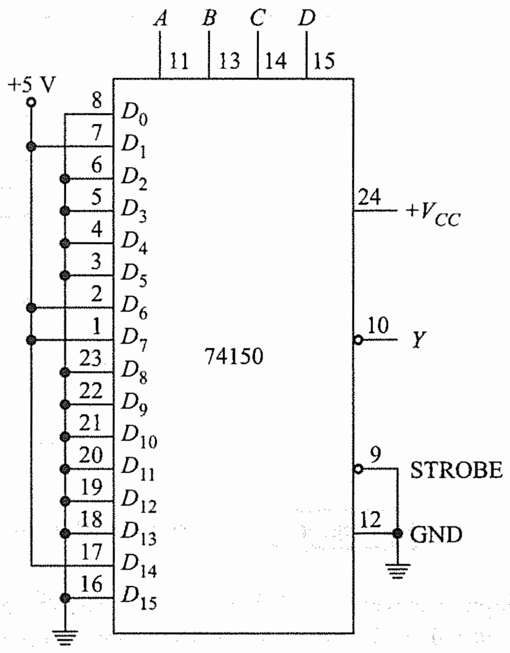


Fig. 4.4 **Using a 74150 for multiplexer logic**

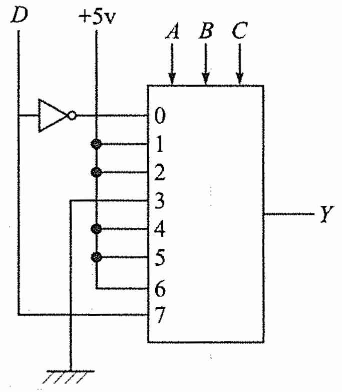
Also notice the bubble on the STROBE input (pin 9). As discussed earlier, the multiplexer is active (enabled) when the STROBE is low and inactive (disabled) when it is high. Because of this, the STROBE is called an *active-low signal*; it causes something to happen when it is low rather than when it is high. Most schematic diagrams use bubbles to indicate active-low signals. From now on, whenever you see a bubble on an input pin, remember that it means the signal is active-low.

Universal Logic Circuit

Multiplexer sometimes is called *universal logic circuit* because a 2^n-to-1 multiplexer can be used as a design solution for any n variable truth table. This we have seen for realization of a 4 variable truth table by 16-to-1 multiplexer in Fig. 4.5. Here, we show how this truth table can be realized using an 8-to-1 multiplexer. Let's consider A,B and C variables to be fed as select inputs. The fourth variable D then has to be present as data input. The method is shown in Fig. 4.5a. The first three rows map the truth table in a different way, similar to the procedure we adopted in entered variable map (Section 3.3). We write all the combinations of 3 select inputs in first row along different columns. Now corresponding to each value of 4th variable D, truth table

ABC	000	001	010	011	100	101	110	111
$D = 0$	1	1	1	0	1	1	1	0
$D = 1$	0	1	1	0	1	1	1	1
Y	D'	1	1	0	1	1	1	D
8-to-1 MUX data input	$D_0 = D'$	$D_1 = 1$	$D_2 = 1$	$D_3 = 0$	$D_4 = 1$	$D_5 = 1$	$D_6 = 1$	$D_7 = D$

(a)



(b)

Fig. 4.5 **A four variable truth table realization using 8-to-1 multiplexer**

output Y is written in 2nd and 3rd row. The 4th row writes Y as a function of D. In fifth row we assign data input values for 8-to-1 multiplexer simply copying Y values obtained in previous row. This is because for each select variable combination a multiplexer transfers a particular input to its output. In 8-to-1 multiplexer, $ABC = 000$ selects D_0, $ABC = 001$ selects D_1 and so on. The corresponding circuit is shown in Fig. 4.5b.

Note that, we can choose any of the four variables (A,B,C,D) of truth table to feed as input to 8-to-1 multiplexer but then mapping in first three rows of Fig. 4.5a will change. The rest of the procedure will remain same. We show an alternative to this technique for a new problem in Example 4.2.

Nibble Multiplexers

Sometimes we want to select one of two input nibbles. In this case, we can use a nibble multiplexer like the one shown in Fig. 4.6. The input nibble on the left is $A_3A_2A_1A_0$ and the one on the right is $B_3B_2B_1B_0$. The control signal labeled *SELECT* determines which input nibble is transmitted to the output. When SELECT is low, the four NAND gates on the left are activated; therefore,

$$Y_3Y_2Y_1Y_0 = A_3A_2A_1A_0$$

When SELECT is high, the four NAND gates on the right are active, and

$$Y_3Y_2Y_1Y_0 = B_3B_2B_1B_0$$

Figure 4.7a on the next page shows the pinout diagram of a 74157, a nibble multiplexer with a SELECT input as previously described. When SELECT is low, the left nibble is steered to the output. When SELECT

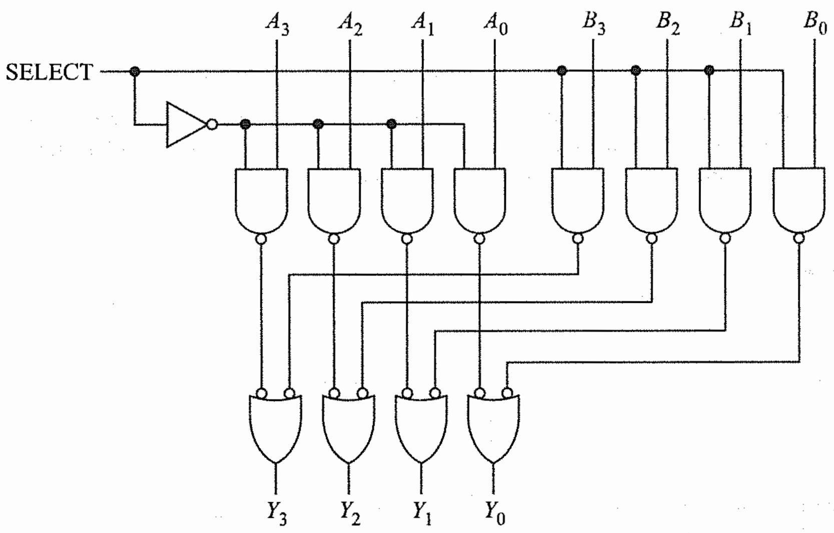


Fig. 4.6 Nibble multiplexer

is high, the right nibble is steered to the output. The 74157 also includes a strobe input. As before, the strobe must be low for the multiplexer to work properly. When the strobe is high, the multiplexer is inoperative.

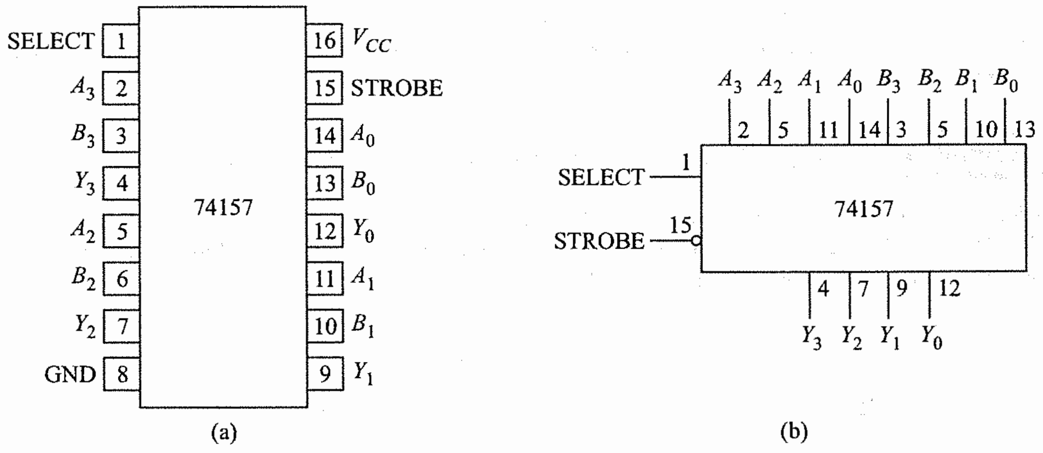


Fig. 4.7 Pinout diagram of 74157

Figure 4.7b shows how to draw a 74157 on a schematic diagram. The bubble on pin 15 tells us that STROBE is an active-low input.

Example 4.1 Show how 4-to-1 multiplexer can be obtained using only 2-to-1 multiplexer.

Solution

Logic equation for 2-to-1 Multiplexer: $Y = A'.D_0 + A.D_1$

Logic equation for 4-to-1 Multiplexer: $Y = A'B'.D_0 + A'B.D_1 + AB'.D_2 + AB.D_3$

This can be rewritten as, $Y = A'(B'.D_0 + B.D_1) + A(B'.D_2 + B.D_3)$

Compare this with equation of 2-to-1 multiplexer. We need two 2-to-1 multiplexer to realize two bracketed terms where B serves as select input. The output of these two multiplexers can be sent to a third multiplexer as data inputs where A serves as select input and we get the 4-to-1 multiplexer. Figure 4.8a shows circuit diagram for this.

Example 4.2

(a) Realize $Y = A'B + B'C' + ABC$ using an 8-to-1 multiplexer. (b) Can it be realized with a 4-to-1 multiplexer?

Solution

(a) First we express Y as a function of minterms of three variables. Thus

$$Y = A'B + B'C' + ABC$$
$$Y = A'B(C' + C) + B'C'(A' + A) + ABC \text{ [As, } X + X' = 1]$$
$$Y = A'B'C' + A'BC' + A'BC + AB'C' + ABC$$

Comparing this with equation of 8 to 1 multiplexer, we find by substituting $D_0 = D_2 = D_3 = D_4 = D_7 = 1$ and $D_1 = D_5 = D_6 = 0$ we get given logic relation.

(b) Let variables A and B be used as selector in 4 to 1 multiplexer and C fed as input. The 4-to-1 multiplexer generates 4 minterms for different combinations of AB. We rewrite given logic equation in such a way that all these terms are present in the equation.

$$Y = A'B + B'C' + ABC$$
$$Y = A'B + B'C'(A' + A) + ABC \text{ [As, } X + X' = 1]$$
$$Y = A'B'.C' + A'B.1 + AB'.C' + AB.C$$

Compare above with equation of a 4-to-1 multiplexer. We see $D_0 = C'$, $D_1 = 1$, $D_2 = C'$ and $D_3 = C$ generate the given logic function.

Example 4.3

Design a 32-to-1 multiplexer using two 16-to-1 multiplexers and one 2-to-1 multiplexer.

Solution The circuit diagram is shown in Fig. 4.8b. A 32-to-1 multiplexer requires $\log_2 32 = 5$ select lines say, $ABCDE$. The lower 4 select lines $BCDE$ chose 16-to-1 multiplexer outputs. The 2-to-1 multiplexer chooses one of the output of two 16-to-1 multiplexers depending on what appears in the 5th select line, A.

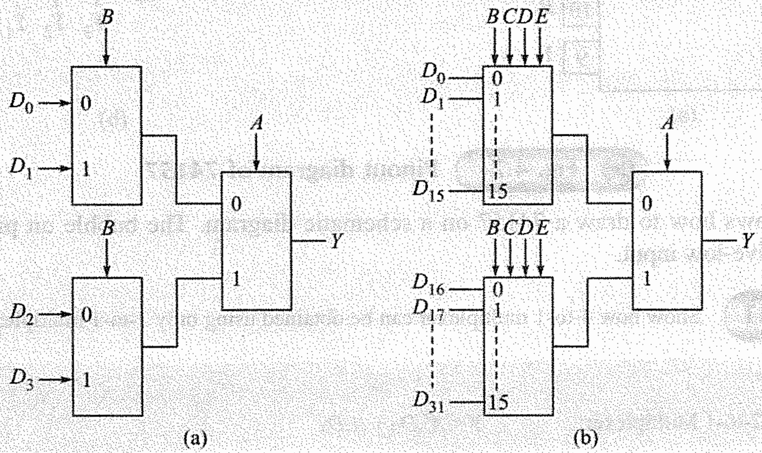


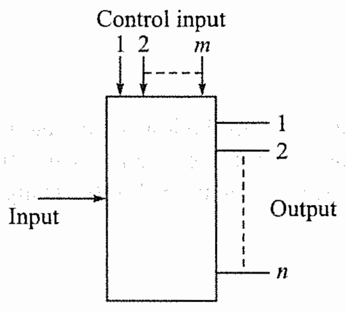
Fig. 4.8 Realization of higher order multiplexers using lower orders

SELF-TEST

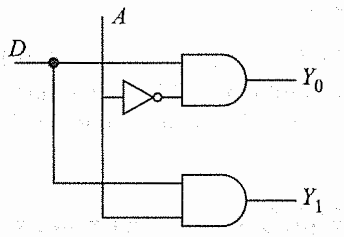
1. A circuit with many inputs but only one output is called a _____.
2. What is the significance of the bubble on pin 10 of the multiplexer in Fig. 4.5?

4.2 DEMULTIPLEXERS

Demultiplex means *one into many*. A *demultiplexer* is a logic circuit with one input and many outputs. By applying control signals, we can steer the input signal to one of the output lines. Figure 4.9a illustrates the general idea. The circuit has 1 input signal, m control or select signals and n output signals where $n \leq 2^m$. Figure 4.9b shows the circuit diagram of a 1-to-2 demultiplexer. Note the similarity of multiplexer and demultiplexer circuits in generating different combinations of control variables through a bank of AND gates. Figure 4.9c lists some of the commercially available demultiplexer ICs. Note that a demultiplexer IC can also behave like a decoder. More about this will be discussed in next section.



(a)



(b)

IC No.	DEMUX Type	Decoder Type
74154	1-to-16	4-to-16
74138	1-to-8	3-to-8
74155	1-to-4	2-to-4

(c)

Fig. 4.9 **(a) Demultiplexer block diagram, (b) Logic circuit of 1-to-2 demultiplexer, (c) Few commercially available ICs**

1-to-16 Demultiplexer

Figure 4.10 shows a 1-to-16 demultiplexer. The input bit is labeled D. This data bit (D) is transmitted to the data bit of the output lines. But which one? Again, this depends on the value of $ABCD$, the control input. When $ABCD = 0000$, the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit D is transmitted only to the Y_0 output, giving $Y_0 = D$. If D is low, Y_0 is low. If D is high, Y_0 is high. As you can see, the value of Y_0 depends on the value of D. All other outputs are in the low state. If the control nibble is changed to $ABCD = 1111$, all gates are disabled except the bottom AND gate. Then, D is transmitted only to the Y_{15} output, and $Y_{15} = D$.

The 74154

The 74154 is a 1-to-16 demultiplexer with the pin diagram of Fig. 4.11. Pin 18 is for the input DATA D, and pins 20 to 23 are for the control bits $ABCD$. Pins 1 to 11 and 13 to 17 are for the output bits Y_0 to Y_{15}. Pin 19 is for the STROBE, again an active-low input. Finally, pin 24 is for V_{CC} and pin 12 for ground.

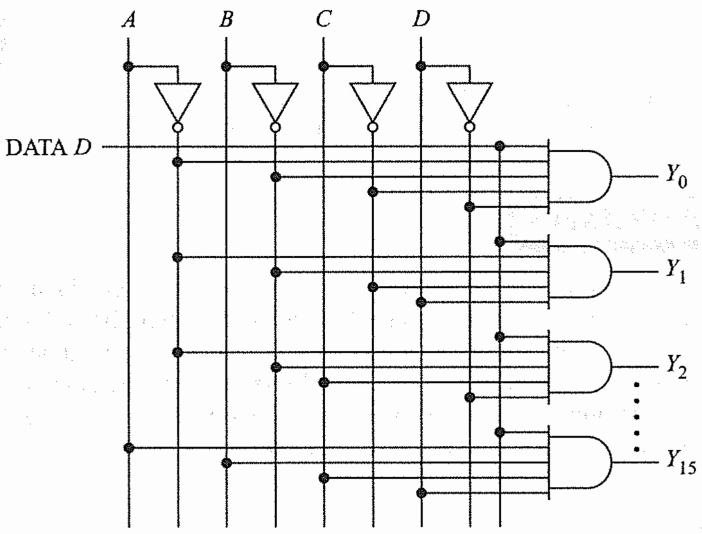


Fig. 4.10 **1-to-16 demultiplexer**

Table 4.3 shows the truth table of a 74154. First, notice the STROBE input. It must be low to activate the 74154. When the STROBE is low, the control input *ABCD* determines which output lines are low when the DATA input is low. When the DATA input is high, all output lines are high. And, when the STROBE is high, all output lines are high.

Table 4.3 **74154 Truth Table**

Strobe	*Data*	*A*	*B*	*C*	*D*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

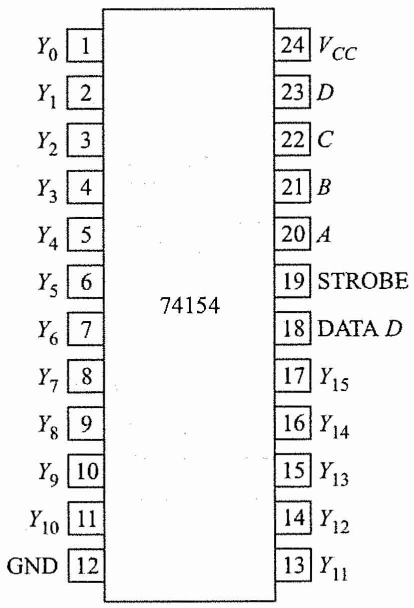


Fig. 4.11 Pinout diagram of 74154

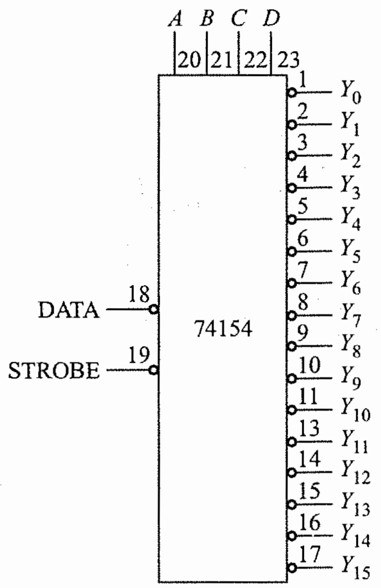


Fig. 4.12 Logic diagram of 74154

Figure 4.12 shows how to draw a 74154 on a schematic diagram. There is one input DATA bit (pin 18) under the control of nibble *ABCD*. The DATA bit is automatically steered to the output line whose subscript is the decimal equivalent of *ABCD*. Again, the bubble on the STROBE pin indicates an active-low input. Notice that DATA is inverted at the input (the bubble on pin 18) and again on any output (the bubble on each output pin). With this double inversion, DATA passes through the 74154 unchanged.

Example 4.4 In Fig. 4.13a, what does the Y_{12} output equal for each of the following conditions:

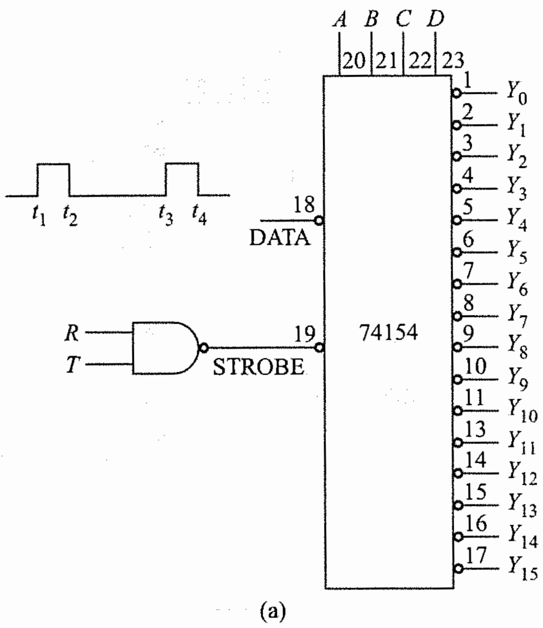
a. R is high, T is high, $ABCD = 0110$.
b. R is low, T is high, $ABCD = 1100$.
c. R is high, T is high, $ABCD = 1100$.

Solution

a. Since R and T are both high, the STROBE is low and the 74154 is active. Because $ABCD = 0110$, the input data is steered to the Y_6 output line (pin 7). The Y_{12} output remains in the high state (see Table 4.3).
b. Here, the STROBE is high and the 74154 is inactive. The Y_{12} output is high.
c. With R and T both high, the STROBE is low and the 74154 is active. Since $ABCD = 1100$, the two pulses are steered to the Y_{12} output (pin 14).

Example 4.5 Show how two 1-to-16 demultiplexers can be connected to get a 1-to-32 demultiplexer.

Solution Figure 4.13b shows the circuit diagram. A 1-to-32 demultiplexer has 5 select variables *ABCDE*. Four of them (*BCDE*) are fed to two 1-to-16 demultiplexer. And the fifth (*A*) is used to select one of these two multiplexer through strobe input. If $A = 0$, the top 714154 is chosen and *BCDE* directs data to one of the 15 outputs of that IC. If $A = 1$, the bottom IC is chosen and depending on value of *BCDE* data is directed to one of the 15 outputs this IC.



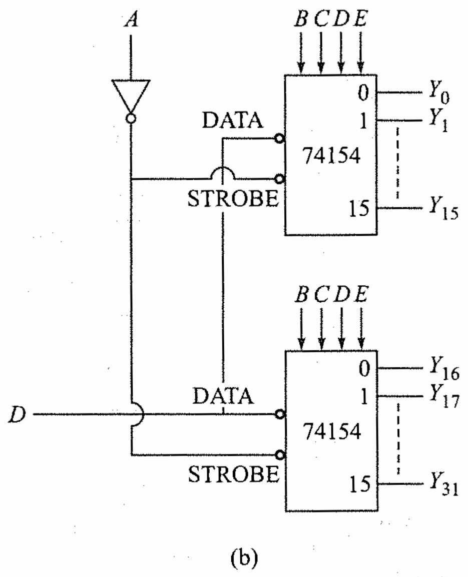


Fig. 4.13

SELF-TEST

3. A logic circuit with one input and many outputs is called a ____.
4. For the 74154 demultiplexer, what must the logic levels *ABCD* be in order to steer the DATA input signal to output line Y_{10}?
5. If *ABCD* = *LHLH*, DATA = *L*, and STROBE = *H*, what will the logic level be at Y_5 on a 74154?

4.3 1-OF-16 DECODER

A *decoder* is similar to a demultiplexer, with one exception—there is no data input. The only inputs are the control bits *ABCD*, which are shown in Fig. 4.14. This logic circuit is called a *l-of-16 decoder* because only 1 of the 16 output lines is high. For instance, when *ABCD* is 0001, only the Y_1 AND gate has all, inputs high; therefore, only the Y_1 output is high. If *ABCD* changes to 0100 only the Y_4 AND gate has all inputs high; as a result, only the Y_4 output goes high.

If you check the other *ABCD* possibilities (0000 to 1111), you will find that the subscript of the high output always equals the decimal equivalent of *ABCD*. For this reason, the circuit is sometimes called a *binary-to-decimal decoder*. Because it has 4 input lines and 16 output lines, the circuit is also known as a *4-line to 16-line decoder*.

Normally, you would not build a decoder with separate inverters and AND gates as shown in Fig. 4.14. Instead, you would use an IC such as the 74154. The 74154 is called a *decoder-demultiplexer*, because it can be used either as a decoder or as a demultiplexer.

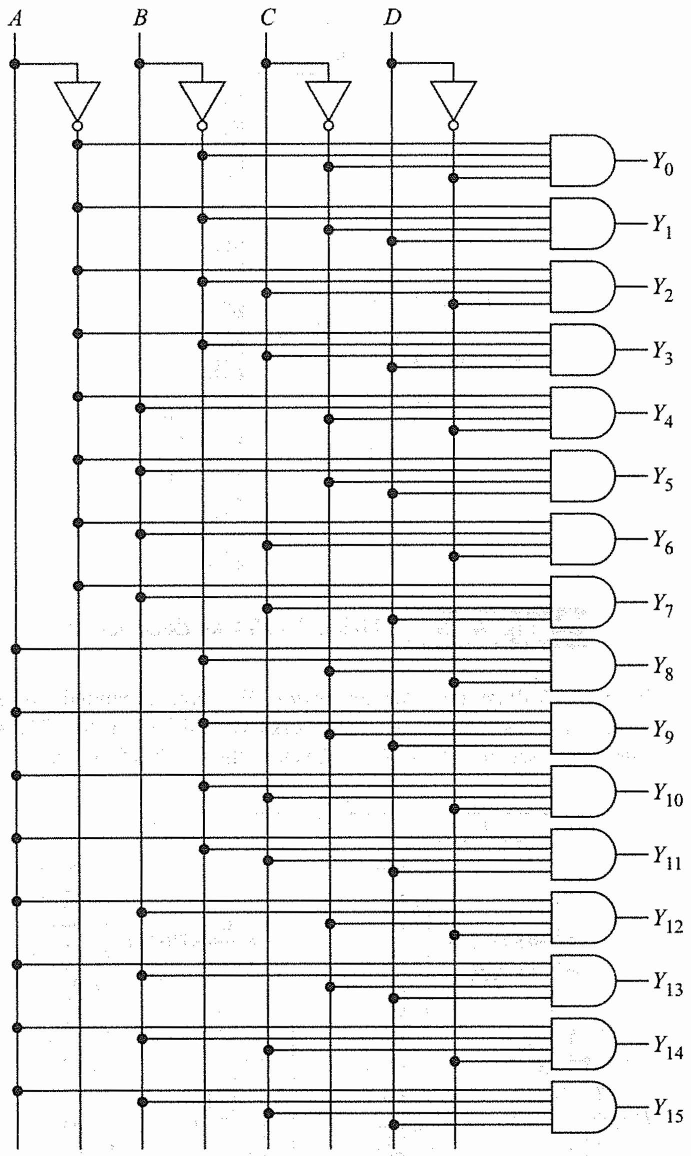


Fig. 4.14 **1-of-16 decoder**

You saw how to use a 74154 as a demultiplexer in Sec. 4.2. To use this same IC as a decoder, all you have to do is ground the DATA and STROBE inputs as shown in Fig. 4.15. Then, the selected output line is in the low state (see Table 4.3). This is why bubbles are shown on the output lines. They remind us that the output line is low when it is active or selected. For instance, if the binary input is

$$ABCD = 0111$$

then the Y_7 output is low, while all other-outputs are high.

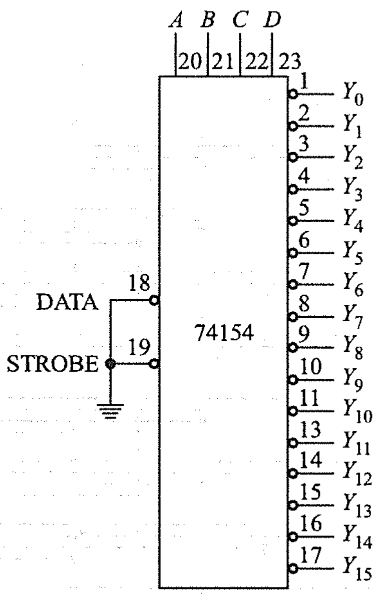


Fig. 4.15 **Using 74154 as decoder**

Example 4.6 Figure 4.16 illustrates *chip expansion*. We have expanded two 74154s to get a 1-of-32 decoder. Here is the way the circuit works. Bit X drives the first 74154, and the complement of X drives the second 74154. When X is low, the first 74154 is active and the second is inactive.

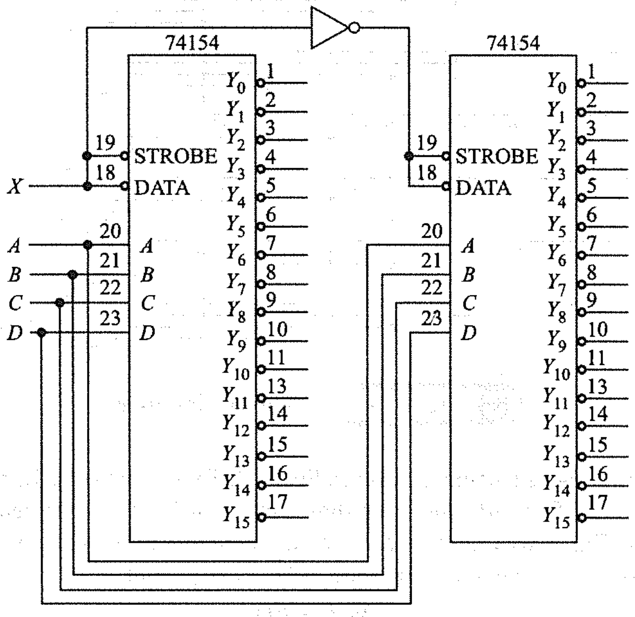


Fig. 4.16 **Chip expansion**

The *ABCD* input drives both decoders but only the first is active; therefore, only one output line on the first decoder is in the low state.

On the other hand, when *X* is high, the first 74154 is disabled and the second one is enabled. This means the *ABCD* input is decoded into a low output from the second decoder. In effect, the circuit of Fig. 4.16 acts like a l-of-32 decoder.

In Fig. 4.16, all output lines are high, except the decoded output line. The bubble on each output line tells anyone looking at the schematic diagram that the active output line is in the low state rather than the high state. Similarly, the bubbles on the STROBE and DATA inputs of each 74154 indicate active-low inputs.

Example 4.7 Show how using a 3-to-8 decoder and multi-input OR gates following Boolean expressions can be realized simultaneously.

$$F_1(A, B, C) = \Sigma m(0, 4, 6);\ F_2(A, B, C) = \Sigma m(0, 5);\ F_2(A, B, C) = \Sigma m(1, 2, 3, 7)$$

Solution Since at the decoder output we get all the minterms we use them as shown in Fig. 4.17 to get the required Boolean functions.

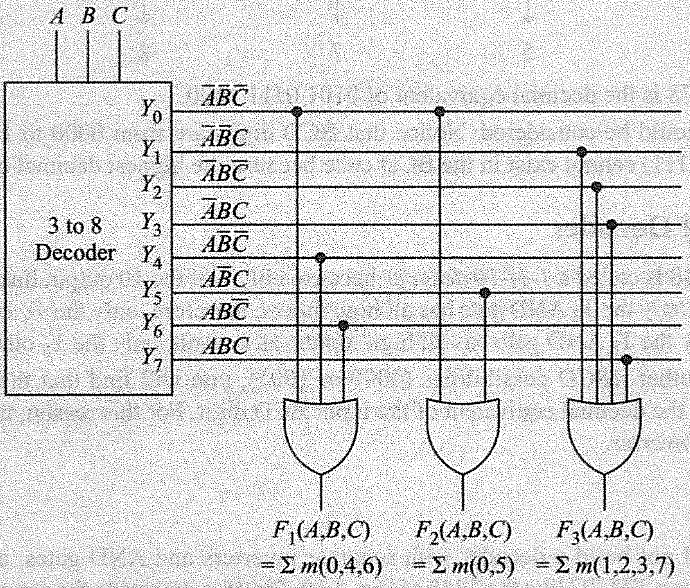


Fig. 4.17 **Solution for Example 4.7**

SELF-TEST

6. What is the significance of the bubbles on the outputs of the 74154 in Fig. 4.15?
7. In Fig. 4.15, *ABCD* = *HLHL*. What are the logic levels of the outputs?

4.4 BCD-TO-DECIMAL DECODERS

BCD is an abbreviation for *binary-coded decimal*. The BCD code expresses each digit in a decimal number by its nibble equivalent. For instance, decimal number 429 is changed to its BCD form as follows:

$$\begin{array}{ccc} 4 & 2 & 9 \\ \downarrow & \downarrow & \downarrow \\ 0100 & 0010 & 1001 \end{array}$$

To anyone using the BCD code, 0100 0010 1001 is equivalent to 429.

As another example, here is how to convert the decimal number 8963 to its BCD form:

$$\begin{array}{cccc} 8 & 9 & 6 & 3 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 1000 & 1001 & 0110 & 0011 \end{array}$$

Again, we have changed each decimal digit to its binary equivalent.

Some early computers processed BCD numbers. This means that the decimal numbers were changed into BCD numbers, which the computer then added, subtracted, etc. The final answer was converted from BCD back to decimal numbers.

Here is an example of how to convert from the BCD form back to the decimal number:

$$\begin{array}{ccc} 0101 & 0111 & 1000 \\ \downarrow & \downarrow & \downarrow \\ 5 & 7 & 8 \end{array}$$

As you can see, 578 is the decimal equivalent of 0101 0111 1000.

One final point should be considered. Notice that BCD digits are from 0000 to 1001. All combinations above this (1010 to 1111) cannot exist in the BCD code because the highest decimal digit being coded is 9.

BCD-to-Decimal Decoder

The circuit of Fig. 4.18 is called a *1-of-10 decoder* because only 1 of the 10 output lines is high. For instance, when *ABCD* is 0011, only the Y_3 AND gate has all high inputs; therefore, only the Y_3 output is high, If *ABCD* changes to 1000, only the Y_8 AND gate has all high inputs; as a result, only the Y_8 output goes high.

If you check the other *ABCD* possibilities (0000 to 1001), you will find that the subscript of the high output always equals the decimal equivalent of the input BCD digit. For this reason, the circuit is also called a *BCD-to-decimal converter*.

The 7445

Typically, you would not build a decoder with separate inverters and AND gates, as shown in Fig. 4.18. Instead, you would use a TTL IC like the 7445 of Fig. 4.19. Pin 16 connects to the supply voltage V_{CC} and pin 8 is grounded. Pins 12 to 15 are for the BCD input (*ABCD*), while pins 1 to 7 and 9 to 11 are for the outputs. This IC is functionally equivalent to the one in Fig. 4.18, except that the active output line is in the low state. All other output lines are in the high state, as shown in Table 4.4. Notice that an invalid BCD input (1010 to 1111) forces all output lines into the high state.

Example 4.8 The decoded outputs of a 7445 can be connected to light-emitting diodes (LEDs), as shown in Fig. 4.20. If each resistance is 1 kΩ and each LED has a forward voltage drop of 2 V, how much current is there through a LED when it is conducting? (See Chapter 13 for a discussion of LEDs.)

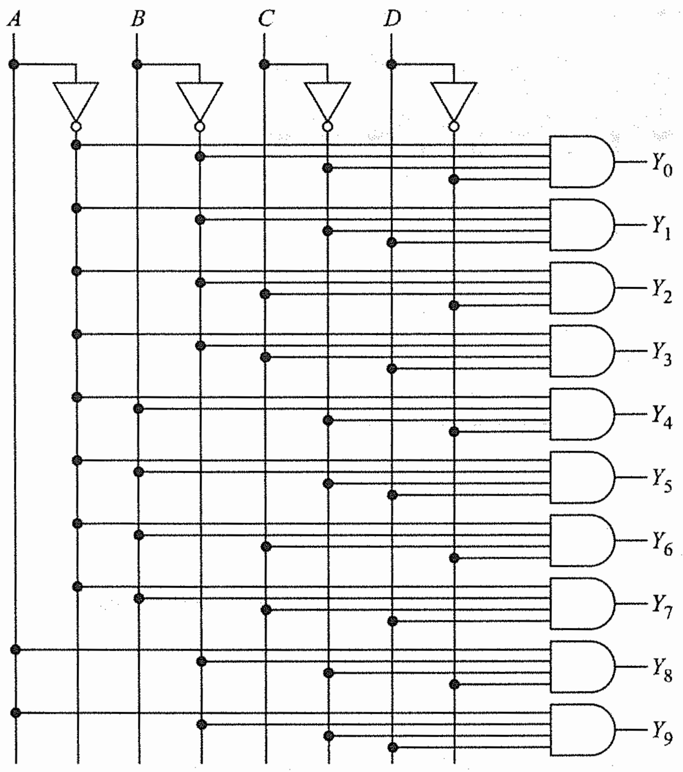


Fig. 4.18 1-of-10 decoder

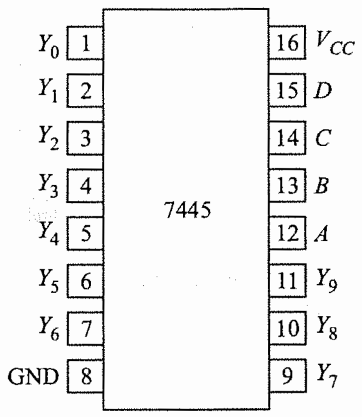


Fig. 4.19 Pinout diagram of 7445

Table 4.4 7445 Truth Table

	Inputs				*Outputs*									
No.	*A*	*B*	*C*	*D*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9
0	*L*	*L*	*L*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
1	*L*	*L*	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
2	*L*	*L*	*H*	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
3	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*
4	*L*	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*
5	*L*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*
6	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*
7	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*
8	*H*	*L*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*
9	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*
	H	*L*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*

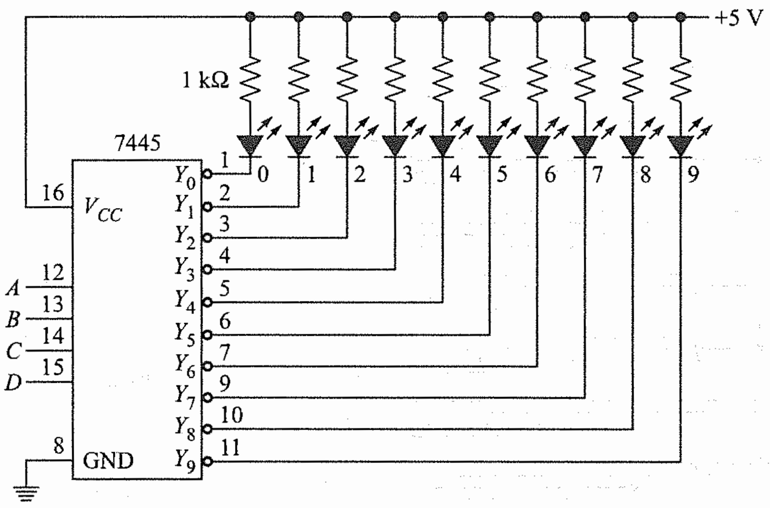


Fig. 4.20 Circuit for Example 4.7

Solution When an output is in the low state, you can approximate the output voltage as zero. Therefore, the current through a LED is

$$I = \frac{5\text{ V} - 2\text{ V}}{1\text{ k}\Omega} = 3\text{ mA}$$

Example 4.9 The LEDs of Fig. 4.20 are numbered 0 through 9. Which of the LEDs is lit for each of the following conditions:

a. $ABCD = 0101$.
b. $ABCD = 1001$.
c. $ABCD = 1100$.

Solution

a. When $ABCD = 0101$, the decoded output line is Y_5. Since Y_5 is approximately grounded, LED 5 lights up. All other LEDs remain off because the other outputs are high.
b. When $ABCD = 1001$, LED 9 is on:
c. $ABCD = 1100$ is an invalid input. Therefore, none of the LEDs is on because all output, lines are high (see Table 4.4).

SELF-TEST

8. What does the abbreviation BCD stand for?
9. What is a LED?

4.5 SEVEN-SEGMENT DECODERS

A LED emits radiation when forward-biased. Why? Because free electrons recombine with holes near the junction. As the free electrons fall from a higher energy level to a lower one, they give up energy in the form of heat and light. By using elements like gallium, arsenic, and phosphorus, a manufacturer can produce LEDs that emit red, green, yellow, blue, orange and infrared (invisible) light. LEDs that produce visible radiation are useful in test instruments, pocket calculators, etc.

Seven-Segment Indicator

Figure 4.21a shows a *seven-segment indicator*, i.e. seven LEDs labeled *a* through *g*. By forward-biasing different LEDs, we can display the digits 0 through 9 (see Fig. 4.21b). For instance, to display a 0, we need to light up segments *a*, *b*, *c*, *d*, *e*, and *f*. To light up a 5, we need segments *a*, *c*, *d*, *f*, and *g*.

Seven-segment indicators may be the common-anode type where all anodes are connected together (Fig. 4.22a) or the common-cathode type where all cathodes are connected together (Fig. 4.22b). With the common-anode type of Fig. 4.22a, you have to connect a current-limiting resistor between each LED and ground. The size of this resistor determines how much current flows through the LED. The typical LED current is between 1 and 50 mA. The common-cathode type of Fig. 4.22b uses a current-limiting resistor between each LED and $+V_{CC}$.

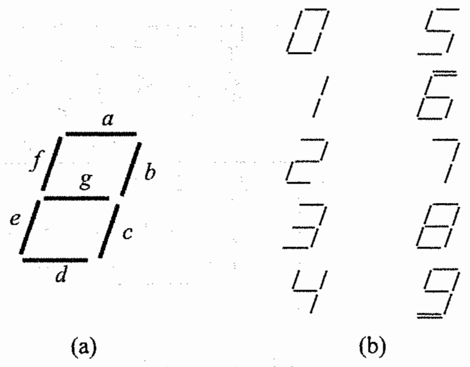


Fig. 4.21 Seven-segment indicator

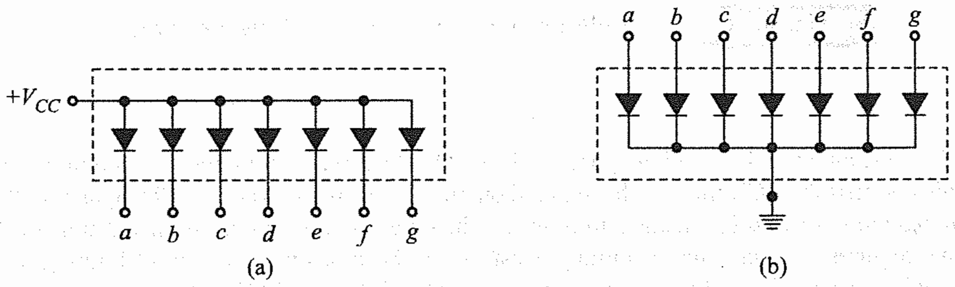


Fig. 4.22 (a) Common-anode type, (b) Common-cathode type

The 7446

A seven-segment *decoder-driver* is an IC decoder that can be used to drive a seven-segment indicator. There are two types of decoder-drivers, corresponding to the common-anode and common-cathode indicators. Each decoder-driver has 4 input pins (the BCD input) and 7 output pins (the *a* through *g* segments).

Figure 4.23a shows a 7446 driving a common-anode indicator. Logic circuits inside the 7446 convert the BCD input to the required output. For instance, if the BCD input is 0111, the internal logic (not shown) of the 7446 will force LEDs *a*, *b*, and *c* to conduct. As a result, digit 7 will appear on the seven-segment indicator.

Notice the current-limiting resistors between the seven-segment indicator and the 7446 of Fig. 4.23a. You have to connect these external resistors to limit the current in each segment to a safe value between 1 and 50 mA, depending on how bright you want the display to be.

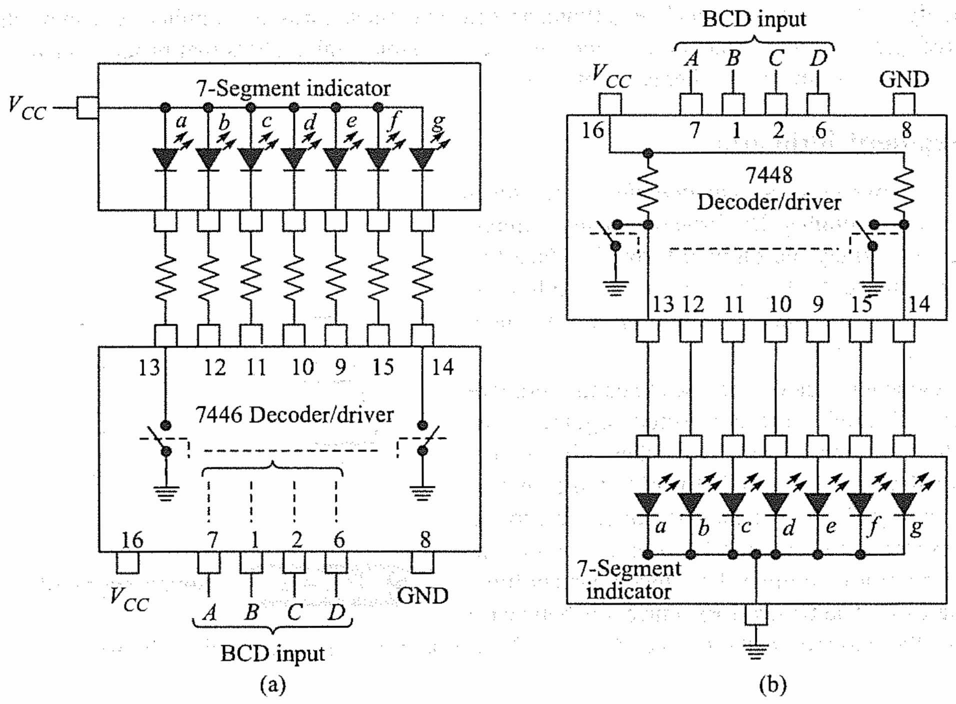


Fig. 4.23 **(a) 7446 decoder-driver, (b) 7448 decoder-driver**

The 7448

Figure 4.23b is the alternative decoding approach. Here, a 7448 drives a common-cathode indicator. Again, internal logic converts the BCD input to the required output. For example, when a BCD input of 0100 is used, the internal logic forces LEDs *b*, *c*, *f*, and *g* to conduct. The seven-segment indicator then displays a 4. Unlike the 7446 that requires external current-limiting resistors, the 7448 has its own current-limiting resistors on the chip. A switch symbol is used to illustrate operation of the 7446 and 7448 in Fig. 4.23. Switching in the actual IC is of course accomplished using bipolar junction transistors (BJTs).

SELF-TEST

10. Sketch the segments in a seven-segment indicator.
11. Each segment of a seven-segment indicator is what type of device?

4.6 ENCODERS

An encoder converts an active input signal into a coded output signal. Figure 4.24 illustrates the general idea. There are *n* input lines, only one of which is active. Internal logic within the encoder converts this active input to a coded binary output with *m* bits.

Decimal-to-BCD Encoder

Figure 4.25 shows a common type of encoder—*the decimal-to-BCD encoder*. The switches are push-button switches like those of a pocket calculator. When button 3 is pressed, the C and D OR gates have high inputs; therefore, the output is

$$ABCD = 0011$$

If button 5 is pressed, the output becomes

$$ABCD = 0101$$

When switch 9 is pressed,

$$ABCD = 1001$$

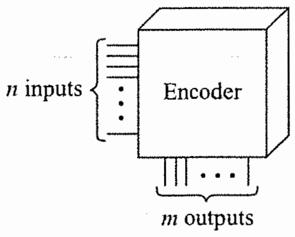


Fig. 4.24 Encoder

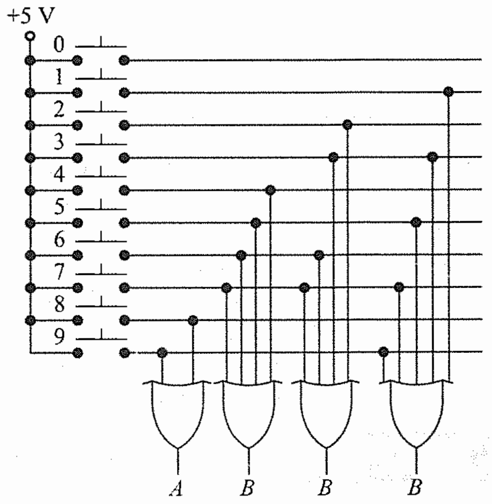


Fig. 4.25 Decimal-to-BCD encoder

The 74147

Figure 4.26a is the pinout diagram for a 74147, a decimal-to-BCD encoder. The decimal input, X_1 to X_9, connect to pins 1 to 5, and 10 to 13. The BCD output comes from pins 14, 6, 7, and 9. Pin 16 is for the supply voltage, and pin 8 is grounded. The label NC on pin 15 means *no connection* (the pin is not used).

Figure 4.26b shows how to draw a 74147 on a schematic diagram. As usual, the bubbles indicate active-low inputs and outputs. Table 4.5 is the truth table of a 74147. Notice the following. When all X inputs are high, all outputs are high. When X_9 is low, the $ABCD$ output is $LHHL$ (equivalent to 9 if you complement the bits). When X_8 is the only low input, $ABCD$ is $LHHH$

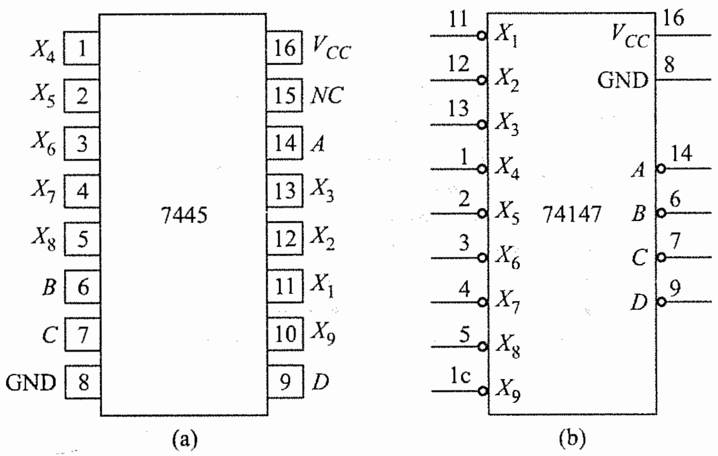


Fig. 4.26 (a) Pinout diagram of 74147, (b) Logic diagram

Table 4.5 **74147 Truth Table**

Inputs									Outputs			
X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	A	B	C	D
H	H	H	H	H	H	H	H	H	H	H	H	H
X	X	X	X	X	X	X	X	L	L	H	H	L
X	X	X	X	X	X	X	L	H	L	H	H	H
X	X	X	X	X	X	L	H	H	H	L	L	L
X	X	X	X	X	L	H	H	H	H	L	L	H
X	X	X	X	L	H	H	H	H	H	L	H	L
X	X	X	L	H	H	H	H	H	H	L	H	H
X	X	L	H	H	H	H	H	H	H	H	L	L
X	L	H	H	H	H	H	H	H	H	H	L	H
L	H	H	H	H	H	H	H	H	H	H	H	L

(equivalent to 8 if the bits are complemented). When X_7 is the only low input, *ABCD* becomes *HLLL* (equivalent to 7 if the bits are complemented). Continue like this through the rest of the truth table and you can see that an active-low decimal input is being converted to a complemented BCD output.

Incidentally, the 74147 is called a *priority encoder* because it gives priority to the highest-order input. You can see this by looking at Table 4.5. If all inputs X_1 through X_9 are low, the highest of these, X_9, is encoded to get an output of *LHHL*. In other words, X_9 has priority over all others. When X_9 is high, X_8 is next in line of priority and gets encoded if it is low. Working your way through Table 4.5, you can see that the highest active-low from X_9 to X_0 has priority and will control the encoding.

Example 4.10 What is the *ABCD* output of Fig. 4.27 when button 6 is pressed?

Solution When all switches are open, the X_1 to X_9 inputs are pulled up to the high state (+5 V). A glance at Table 4.5 indicates that the *ABCD* output is *HHHH* at this time.

When switch 6 is pressed, the X_6 input is grounded. Therefore, all *X* inputs are high, except for X_6. Table 4.5 indicates that the *ABCD* output is *HLLH*, which is equivalent to 6 when the output bits are complemented.

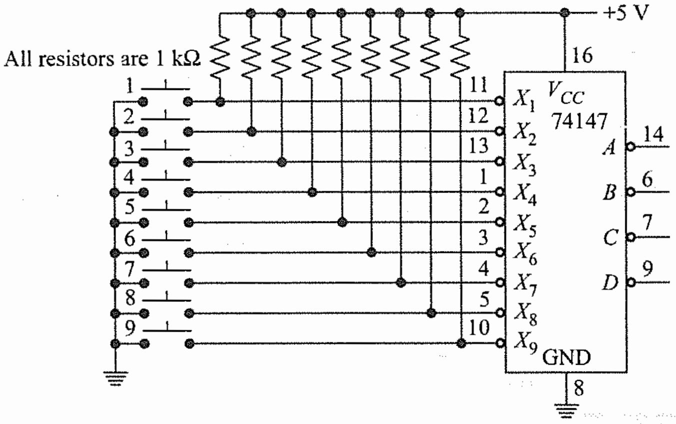


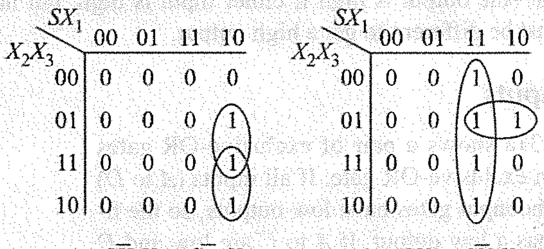
Fig. 4.27

Example 4.11 Design a priority encoder the truth table of which is shown in Fig. 4.28a. The order of priority for three inputs is $X_1 > X_2 > X_3$. However, if the encoder is not enabled by S or all the inputs are inactive the output $AB = 00$.

Solution Figure 4.28b and Fig. 4.28c show the Karnaugh map for output A and B respectively. Note that, we have used a different notation for input variables in these maps. Compare this with notations presented in previous chapters. You will find a variable with prime is presented by 0 and if it is not primed is represented by 1. Then taking groups of 1s we get the design equations as shown in the figure. The logic circuits for output A and B can be directly drawn from these equations.

Intput				Output	
S	X_1	X_2	X_3	A	B
0	×	×	×	0	0
1	1	×	×	0	1
1	0	1	×	1	0
1	0	0	1	1	1
1	0	0	0	0	0

(a)



$A = S\bar{X}_1X_3 + S\bar{X}_1X_2$ (b)

$B = SX_1 + S\bar{X}_2X_3$ (c)

Fig. 4.28 **Design of a priority encoder**

SELF-TEST

12. What is an encoder?
13. What type of encoder is the TTL 74147?

4.7 EXCLUSIVE-OR GATES

The *exclusive-OR gate* has a high output only when an odd number of inputs is high. Figure 4.29 shows how to build an exclusive-OR gate. The upper AND gate forms the product $\bar{A}B$, while the lower one produces $A\bar{B}$. Therefore, the output of the OR gate is

$$Y = \bar{A}B + A\bar{B}$$

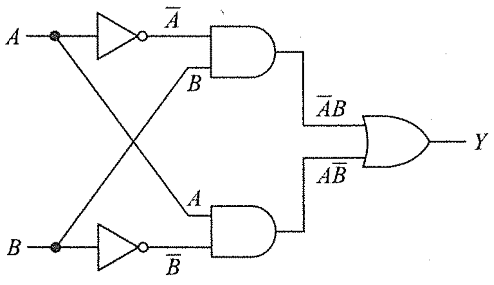


Fig. 4.29 **Exclusive-OR gate**

Here is what happens for different inputs. When A and B are low, both AND gates have low outputs; therefore, the final output Y is low. If A is low and B is high, the upper AND gate has a high output, so the OR gate has high output. Likewise, a high A and a low B result in a final output that is high. If both inputs are high, both AND gates have low outputs and the final output is low.

Table 4.6 shows the truth table for a 2-input exclusive-OR gate. The output is high when A or B is high, but not when both are high. This is why the circuit is known as an exclusive-OR gate. In other words, the output is a 1 *only* when the inputs are different.

Table 4.6 **Exclusive-OR Truth Table**

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

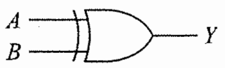


Fig. 4.30 **Logic symbol for exclusive-OR gate**

Figure 4.30 shows the symbol for a 2-input exclusive-OR gate. Whenever you see this symbol, remember the action—the output is high if either input is high, but not when both are high. Stated another way, the inputs must be different to get a high output.

Four Inputs

Figure 4.31a shows a pair of exclusive-OR gates driving an exclusive-OR gate. If all inputs (*A* to *D*) are low, the input gates have low outputs, so the final gate has a low output. If *A* to *C* are low and *D* is high, the upper gate has a low output, the lower gate has a high output, and the output gate has a high output.

If we continue analyzing the circuit operation for the remaining input possibilities, we can work out Table 4.7. Here is an important property of this truth table. Each *ABCD* input with an odd number of 1s produces an output 1. For instance, the first *ABCD* entry to produce an output 1 is 0001; it has an odd

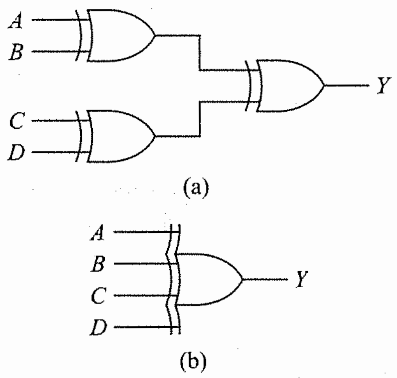


Fig. 4.31 **Four-input exclusive OR gate**

Table 4.7 **4-Input Exclusive-OR Gate**

Comment	*A*	*B*	*C*	*D*	*Y*
Even	0	0	0	0	0
Odd	0	0	0	1	1
Odd	0	0	1	0	1
Even	0	0	1	1	0
Odd	0	1	0	0	1
Even	0	1	0	1	0
Even	0	1	1	0	0
Odd	0	1	1	1	1
Odd	1	0	0	0	1
Even	1	0	0	1	0
Even	1	0	1	0	0
Odd	1	0	1	1	1
Even	1	1	0	0	0
Odd	1	1	0	1	1
Odd	1	1	1	0	1
Even	1	1	1	1	0

number of 1s. The next *ABCD* entry to produce an output 1 is 0010; again, an odd number of 1s. An output 1 also occurs for these *ABCD* inputs: 0100, 0111, 1000, 1011,1101, and 1110, each having an odd number of 1s.

Figure 4.31a illustrates the logic for a 4-input exclusive-OR gate. In this book, we will use the abbreviated symbol given in Fig. 4.31b to represent a 4-input exclusive-OR gate. When you see this symbol, remember the action—the gate produces an output 1 when the *ABCD* input has an odd number of 1s.

Any Number of Inputs

Using 2-input exclusive-OR gates as building blocks, you can produce exclusive-OR gates with any number of inputs. For example, Fig. 4.32a shows a pair of exclusive-OR gates. There are 3 inputs and 1 output. If you analyze this circuit, you will find it produces an output 1 only when the 3-bit input has an odd number of 1s. Figure 4.32b shows an abbreviated symbol for a 3-input exclusive-OR gate.

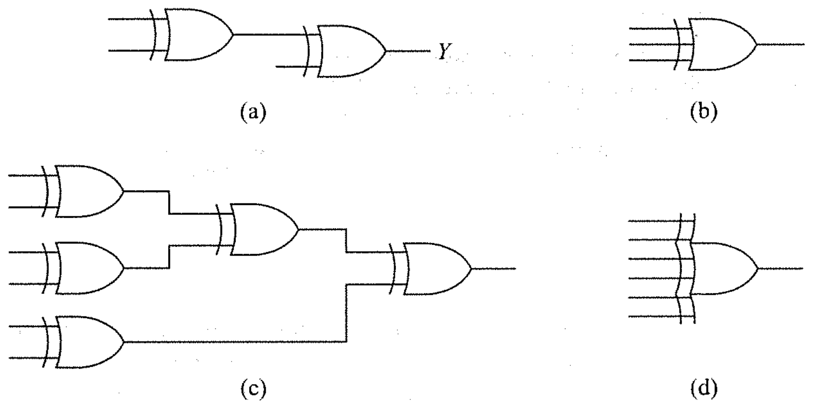


Fig. 4.32 **Exclusive-OR gate with several inputs**

As another example, Fig. 4.32c shows a circuit with 6 inputs and 1 output. Analysis of the circuit shows that it produces an output 1 only when the 6-bit input has an odd number of 1s. Figure 4.32d shows an abbreviated symbol for a 6-input exclusive-OR gate.

In general, you can build an exclusive-OR gate with any number of inputs. Such a gate always produces an output 1 only when the *n*-bit input has an odd number of 1s.

SELF-TEST

14. When is the output of an exclusive-OR gate high?
15. Draw the logic symbol for an exclusive-OR gate.

4.8 PARITY GENERATORS AND CHECKERS

Even parity means an *n*-bit input has an even number of 1s. For instance, 110011 has even parity because it contains four 1s. *Odd parity* means an *n*-bit input has an odd number of 1s. For example, 110001 has odd parity because it contains three 1s.

Here are two more examples:

1111 0000 1111 0011 even parity
1111 0000 1111 0111 odd parity

The first binary number has even parity because it contains ten 1s; the second binary number has odd parity because it contains eleven 1s. Incidentally, longer binary numbers are much easier to read if they are split into nibbles, or groups of four, as done here.

Parity Checker

Exclusive-OR gates are ideal for checking the parity of a binary number because they produce an output 1 when the input has an odd number of 1s. Therefore, an even-parity input to an exclusive-OR gate produces a low output, while an odd-parity input produces a high output.

For instance, Fig. 4.33 shows a 16-input exclusive-OR gate. A 16-bit number drives the input. The exclusive-OR gate produces an output 1 because the input has odd parity (an odd number of 1s). If the 16-bit input changes to another value, the output becomes 0 for even-parity numbers and 1 for odd-parity numbers.

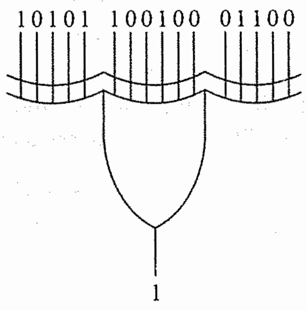


Fig. 4.33 Exclusive-OR gate with 16 inputs

Parity Generation

In a computer, a binary number may represent an instruction that tells the computer to add, subtract, and so on; or the binary number may represent data to be processed like a number, letter, etc. In either case, you sometimes will see an extra bit added to the original binary number to produce a new binary number with even or odd parity.

For instance, Fig. 4.34 shows this 8-bit binary number:

$$X_7X_6X_5X_4 \quad X_3X_2X_1X_0$$

Suppose this number equals 0100 0001. Then, the number has even parity, which means the exclusive-OR gate produces an output of 0. Because of the inverter,

$$X_8 = 1$$

and the final 9-bit output is 1 0100 0001. Notice that this has odd parity.

Suppose we change the 8-bit input to 0110 0001. Now, it has odd parity. In this case, the exclusive-OR gate produces an output 1. But the inverter produces a 0, so that the final 9-bit output is 0 0110 0001. Again, the final output has odd parity.

The circuit given in Fig. 4.34 is called an *odd-parity generator* because it always produces a 9-bit output number with odd parity. If the 8-bit input has even parity, a 1 comes

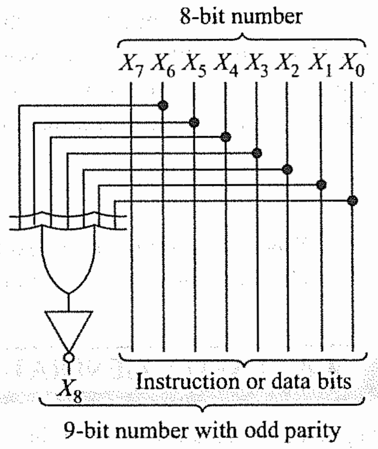

8-bit number
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
X_8
Instruction or data bits
9-bit number with odd parity

Fig. 4.34 Odd-parity generation

out of the inverter to produce a final output with odd parity. On the other hand, if the 8-bit input has odd parity, a 0 comes out of the inverter, and the final 9-bit output again has odd parity. (To get an even-parity generator, delete the inverter.)

Application

What is the practical application of parity generation and checking? Because of transients, noise, and other disturbances, 1-bit errors sometimes occur when binary data is transmitted over telephone lines or other communication paths. One way to check for errors is to use an odd-parity generator at the transmitting end and an odd-parity checker at the receiving end. If no 1-bit errors occur in transmission, the received data will have odd parity. But if one of the transmitted bits is changed by noise or any other disturbance, the received data will have even parity.

For instance, suppose we want to send 0100 0011. With an odd-parity generator like Fig. 4.34, the data to be transmitted will be 0 0100 0011. This data can be sent over telephone lines to some destination. If no errors occur in transmission, the odd-parity checker at the receiving end will produce a high output, meaning the received number has odd parity. On the other hand, if a 1-bit error does creep into the transmitted data, the odd-parity checker will have a low output, indicating the received data is invalid.

One final point should be made. Errors are rare to begin with. When they do occur, they are usually 1-bit errors. This is why the method described here catches almost all of the errors that occur in transmitted data.

The 74180

Figure 4.35 shows the pinout diagram for a 74180, which is a TTL parity generator-checker. The input data bits are X_7 to X_0; these bits may have even or odd parity. The even input (pin 3) and the odd input (pin 4) control the operation of the chip as shown in Table 4.8. The symbol Σ stands for *summation*. In the left input column of Table 4.8, Σ of H's (highs) refers to the parity of the input data X_7 to X_0. Depending on how you set up the values of the even and odd inputs, the Σ even and Σ odd outputs may be low or high.

For instance, suppose even input is high and odd input is low. When the input data has even parity (the first entry of Table 4.8), the Σ even output is high and the Σ odd output is low. When the input data has odd parity, the Σ even output is low and the Σ odd output is high.

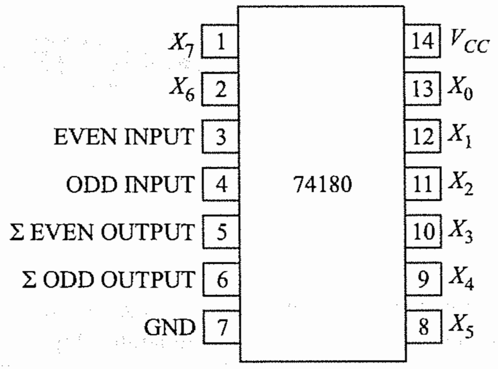


Fig. 4.35 Pinput diagram of 74180

Table 4.8 74180 Truth Table

Inputs			*Outputs*	
Σ of H's at X_7 to X_0	*Even*	*Odd*	*Σ even*	*Σ odd*
Even	*H*	*L*	*H*	*L*
Odd	*H*	*L*	*L*	*H*
Even	*L*	*H*	*L*	*H*
Odd	*L*	*H*	*H*	*L*
X	*H*	*H*	*L*	*L*
X	*L*	*L*	*H*	*H*

If you change the control inputs, you change the operation. Assume that the even input is low and the odd input is high. When the input data has even parity, the Σ even output is low and the Σ odd output is high. When the input data has odd parity, the Σ even output is high and the Σ odd output is low.

The 74180 can be used to detect even or odd parity. It can also be set up to generate even or odd parity.

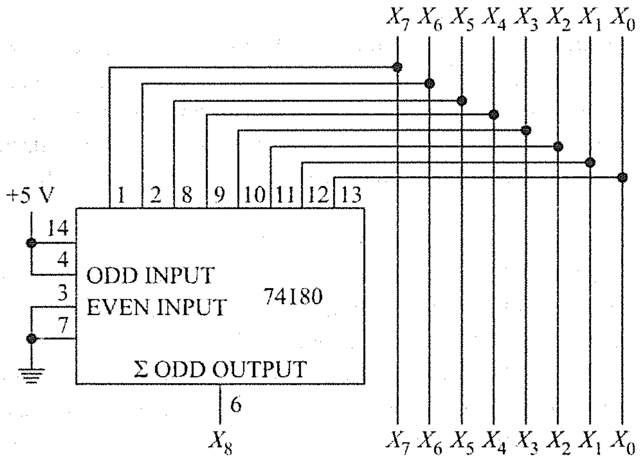

X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
+5 V
1 2 8 9 10 11 12 13
14
4
3
7
ODD INPUT
EVEN INPUT
74180
Σ ODD OUTPUT
6
X_8
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0

Fig. 4.36 **Using a 74180 to generate odd parity**

Example 4.12 Show how to connect a 74180 to generate a 9-bit output with odd parity.

Solution Figure 4.36 shows one solution. The ODD INPUT (pin 4) is connected to +5 V, and the EVEN INPUT (pin 3) is grounded. Suppose the input data $X_7 \ldots X_0$ has even parity. Then, the third entry of Table 4.8 tells us the Σ ODD OUTPUT (pin 6) is high. Therefore, the 9-bit number $X_8 \ldots X_0$ coming out of the circuit has odd parity.

On the other hand, suppose $X_7 \ldots X_0$ has odd parity. Then the fourth entry of Table 4.8 says that the Σ odd output is low. Again, the 9-bit number $X_8 \ldots X_0$ coming out at the bottom of Fig. 4.36 has odd parity.

The following conclusion may be drawn. Whether the input data has even or odd parity, the 9-bit number being generated in Fig. 4.36 always has odd parity.

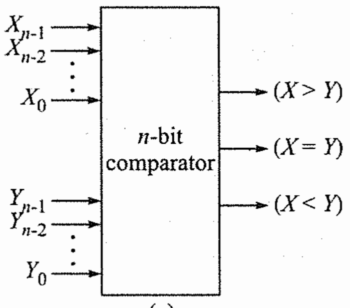
SELF-TEST

16. What does it mean to say that an n-bit binary number has *even* parity?
17. Exclusive-OR gates are useful as parity generators. (T or F)

4.9 MAGNITUDE COMPARATOR

Magnitude comparator compares magnitude two n-bit binary numbers, say X and Y and activates one of these three outputs $X = Y$, $X > Y$ and $X < Y$. Figure 4.37a presents block diagram of such a comparator. Fig. 4.37b presents truth table when two 1-bit numbers are compared and its circuit diagram is shown in Fig. 4.37c. The logic equations for the outputs can be written as follows, where G, L, E stand for greater than, less than and equal to respectively.

$$(X > Y): G = XY' \quad (X < Y): L = X'Y \qquad (X = Y): E = X'Y' + XY = (XY' + X'Y)' = (G + L)'$$



Input		Output		
X	Y	$X>Y$	$X=Y$	$X<Y$
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

(b)

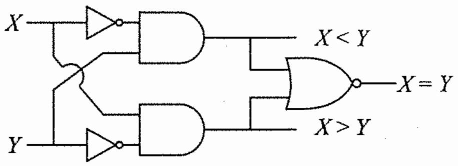


Fig. 4.37 **(a) Block diagram of Magnitude comparator, (b) Truth table, (c) Circuit for 1-bit comparator**

Now, how can we design a 2-bit comparator? We can form a 4-variable (X: X_1X_0 and Y: Y_1Y_0) truth table and get logic equations through any simplification technique. But this procedure will become very complex if we try to design a comparator for 3-bit numbers or more. Here, we discuss a simple but generic procedure for 2-bit comparator design, which can easily be extended to make any n-bit magnitude comparator. We shall use the truth table of 1-bit comparator that generates greater than, less than and equal terms.

Let's first define bit-wise greater than terms (G):	$G_1 = X_1Y_1'$,	$G_0 = X_0Y_0'$
Then, bit-wise less than term (L):	$L_1 = X_1'Y_1$,	$L_0 = X_0'Y_0$
Therefore, bit-wise equality term (E):	$E_1 = (G_1 + L_1)'$,	$E_0 = (G_0 + L_0)'$

From above definitions we can easily write 2-bit comparator outputs as follows.

$$(X = Y) = E_1.E_0 \qquad (X > Y) = G_1 + E_1.G_0 \qquad (X < Y) = L_1 + E_1.L_0$$

The logic followed in arriving at these equations is this; $X = Y$ when both the bits are equal. $X > Y$ if MSB of X is higher ($G_1 = 1$) than that of Y. If MSB is equal, given by $E_1 = 1$, then LSB of X and Y is checked and if found higher ($G_0 = 1$) the condition $X > Y$ is fulfilled. Similar logic gives us the $X < Y$ term.

Thus for any two n-bit numbers X: $X_{n-1}\,X_{n-2}\ldots X_0$ and Y: $Y_{n-1}\,Y_{n-2}\ldots Y_0$

We can write,
$$(X = Y) = E_{n-1}\,E_{n-2}\ldots E_0$$
$$(X > Y) = G_{n-1} + E_{n-1}G_{n-2} + \ldots + E_{n-1}\,E_{n-2}\ldots\,E_1G_0$$
$$(X < Y) = L_{n-1} + E_{n-1}L_{n-2} + \ldots + E_{n-1}\,E_{n-2}\ldots\,E_1L_0$$

where E_i, G_i and L_i represent for ith bit $X_i = Y_i$, $X_i > Y_i$ and $X_i < Y_i$ terms respectively.

The block diagram of IC 7485, which compares two 4-bit numbers is shown in Fig. 4.38a. This is a 16 pin IC and all the pin numbers are mentioned in this functional diagram. Note that the circuit has three additional inputs in the form of $(X = Y)_{\text{in}}$, $(X > Y)_{\text{in}}$ and $(X < Y)_{\text{in}}$. What is the use of them? They are used when we need to connect more than one IC 7485 to compare numbers having more than 4-bits. But these inputs are of lower priority. They can decide the output only when 4-bit numbers fed to this IC are equal. For example, if $X =$ 0100 and $Y =$ 0011, $(X > Y)_{\text{out}}$ will be high and other outputs will be low irrespective of the value appearing at $(X = Y)_{\text{in}}$, $(X > Y)_{\text{in}}$ and $(X < Y)_{\text{in}}$. When IC 7485 is not used in cascade we keep $(X = Y)_{\text{in}} = 1$, $(X > Y)_{\text{in}} = 0$ and $(X < Y)_{\text{in}} = 0$.

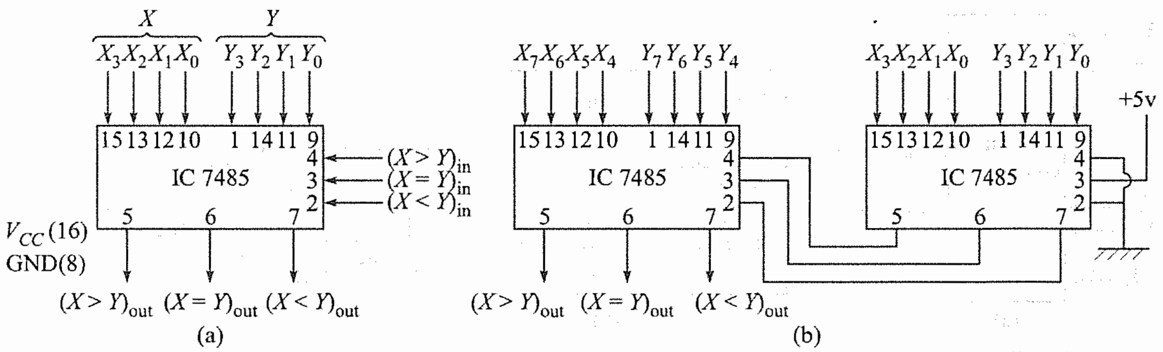


Fig. 4.38 (a) Functional diagram of IC 7485, (b) 8-bit comparator from two 4-bit comparators

Example 4.13 Show how two IC 7485 can be used to compare magnitude of two 8-bit numbers.

Solution Refer to Fig. 4.38b for solution. The numbers to compare are X: $X_7 X_6 \ldots X_0$ and Y: $Y_7 Y_6 \ldots Y_0$. We need two IC 7485s each one comparing 4 bits. The most significant bits (suffix 7,6,5,4) are given higher priority and the final output is taken from that IC 7485 which compares them.

SELF-TEST

18. How many outputs a magnitude comparator generates?
19. How many IC 7485s are needed to compare two 12-bit numbers?

4.10 READ-ONLY MEMORY

A *read-only memory* (which is abbreviated ROM and rhymes with Mom) is an IC that can store thousands of binary numbers representing computer instructions and other fixed data. A good example of fixed data is the unchanging information in a mathematical table. Since the numerical data do not change, they can be stored in a ROM, included in a computer system, and used as a "look-up" table when needed. Some of the smaller ROMs are also used to implement truth tables. In other words, we can use a ROM instead of sum-of-products circuit to generate any Boolean function.

Diode ROM

Suppose we want to build a circuit that stores the binary numbers shown in Table 4.9. To keep track of where the numbers are stored, we will assign *addresses*. For instance, we want to store 0111 at address 0, 1000 at address 1, 1011 at address 2, and so forth. Figure 4.39 shows one way to store the nibbles given in Table 4.9. When the switch is in position 0 (address 0), the upper row of diodes are conducting current (they act as closed

Table 4.9 Diode ROM

Address	*Nibble*
0	0111
1	1000
2	1011
3	1100
4	0110
5	1001
6	0011
7	1110

switches). (See Chapter 14 for a discussion of diodes.) The output of the ROM is thus

$$Y_3 Y_2 Y_1 Y_0 = 0111$$

When the switch is moved to position 1, the second row is activated and

$$Y_3 Y_2 Y_1 Y_0 = 1000$$

As you move the switch to the remaining positions or addresses, you get a $Y_3 \ldots Y_0$ output that matches the nibbles given in Table 4.9.

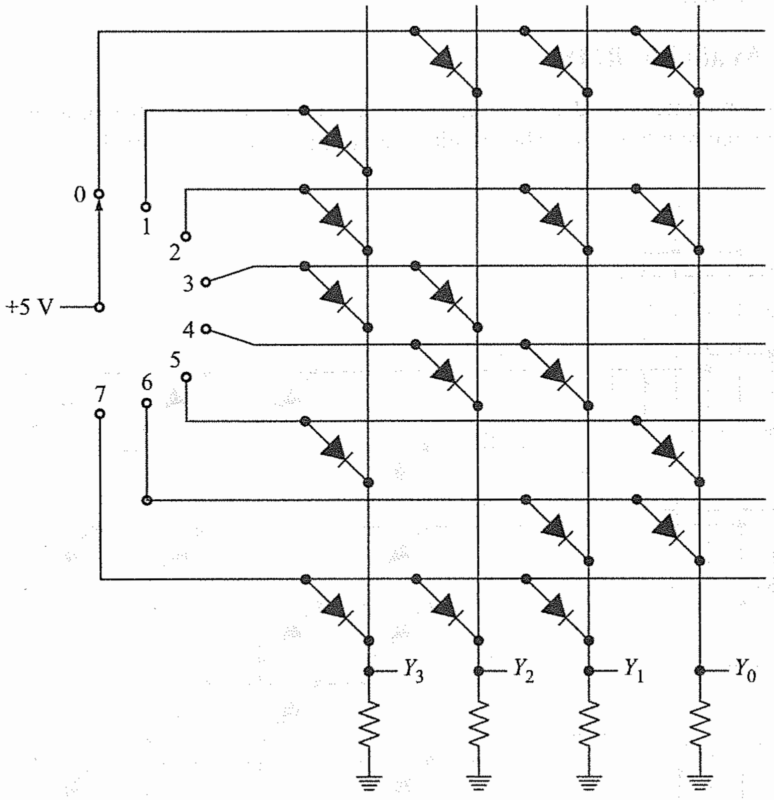


Fig. 4.39 Diode ROM

On-Chip Decoding

Rather than switch-select the addresses as shown in Fig. 4.39, a manufacturer uses *on-chip decoding*. Figure 4.40 illustrates the idea. The 3-input pins (A, B, and C) supply the binary address of the stored number. Then, a 1-of-8 decoder produces a high output to one of the diode rows. For instance, if

$$ABC = 100$$

the 1-of-8 decoder applies a high voltage to the $A\overline{B}\overline{C}$ line, and the ROM output is

$$Y_3 Y_2 Y_1 Y_0 = 0110$$

If you change the binary address to

$$ABC = 110$$

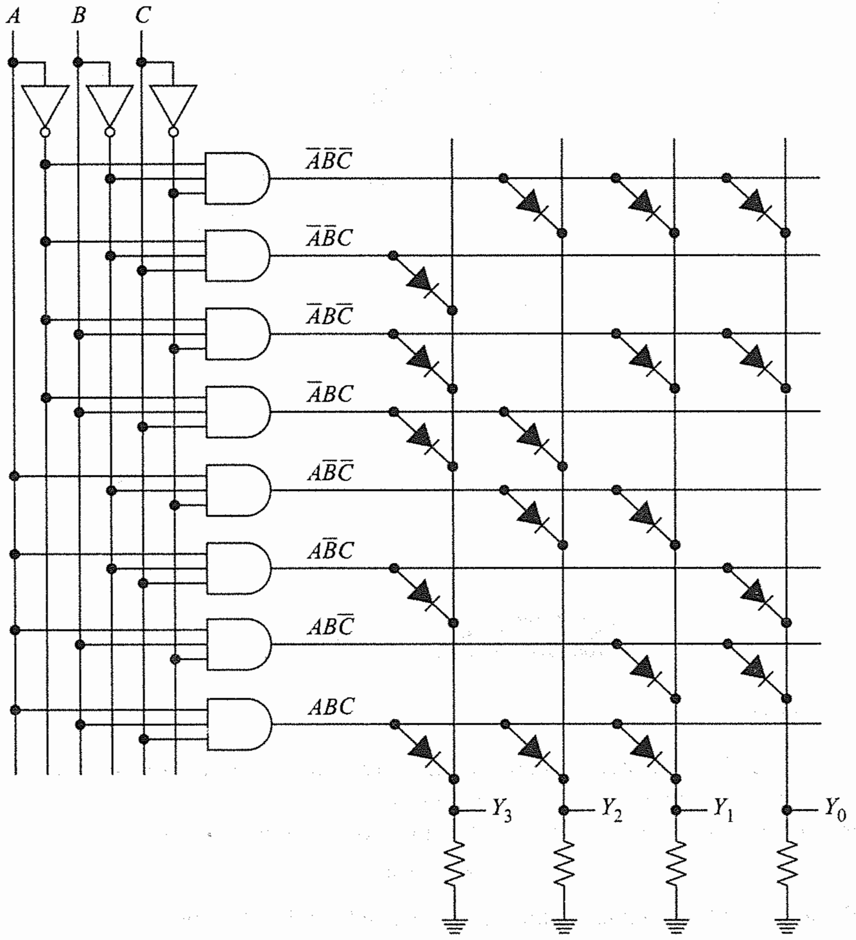
the ROM output changes to

$$Y_3 Y_2 Y_1 Y_0 = 0011$$

With on-chip decoding, n inputs can select 2^n memory locations (stored numbers). For instance, we need 3 address lines to access 8 memory locations, 4 address lines for 16 memory locations, 8 address lines for 256 memory locations, and so on.

Commercially Available ROMs

A binary number is sometimes called a *word*. In a computer, binary numbers or words represent instructions, alphabet letters, decimal numbers, etc. The circuit given in Fig. 4.40 is a 32-bit ROM organized as 8 words



A
B
C
$\overline{A}\overline{B}\overline{C}$
$\overline{A}\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
Y_3
Y_2
Y_1
Y_0

Fig. 4.40 On-chip decoding

with 4 bits at each address (an 8×4 ROM). The ROM given in Fig. 4.40 is for instructional purposes only because you would not build this circuit with discrete components. Instead, you would select a commercially available ROM. For instance, here are some TTL ROMs:

7488: 256 bits organized as 32×8

74187: 1024 bits organized as 256×4

74S370: 2048 bits organized as 512×4

As you can see, the 7488 can store 32 words of 8 bits each, the 74187 can store 256 words of 4 bits each, and the 74S370 can store 512 words of 4 bits each. If you want to store bytes (words with 8 bits), then you can parallel the 4-bit ROMs. For example, two parallel 74187s can store 256 words of 8 bits each.

One way to change the stored numbers of a ROM is by adding or removing diodes. With discrete circuits, you would have to solder or unsolder diodes to change the stored nibbles. With integrated circuits, however, you can send a list of the data to be stored to an IC manufacturer, who then produces a *mask* (a photographic template of the circuit). This mask is used in the mass production of your ROMs. As a rule, ROMs are used only for large production runs (thousands or more) because of manufacturing costs.

Generating Boolean Functions

Because the AND gates of Fig. 4.40 produce all the fundamental products and the diodes OR some of these products, the ROM is generating four Boolean functions as follows:

$$Y_3 = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + ABC \tag{4.1}$$
$$Y_2 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + ABC \tag{4.2}$$
$$Y_1 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C} + ABC \tag{4.3}$$
$$Y_0 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}C + AB\overline{C} \tag{4.4}$$

This means that you can use a ROM instead of a logic circuit to implement a truth table.

For instance, suppose you start with a truth table like the one in Table 4.10. There are four outputs: Y_3, Y_2, Y_1, and Y_0. A sum-of-products solution would lead to four AND-OR circuits, one for Eq. (4.1), a second for Eq. (4.2), and so on. The ROM solution is different. With a ROM you have to store the binary numbers of Table 4.9 (same as Table 4.10) at the indicated addresses. When this is done, the ROM given in Fig. 4.40 is equivalent to a sum-of-products circuit. In other words, you can use the ROM instead of an AND-OR circuit to generate the desired truth table.

Table 4.10 Truth Table

A	B	C	Y_3	Y_2	Y_1	Y_0
0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	1	1	1	1	0	0
1	0	0	0	1	1	0
1	0	1	1	0	0	1
1	1	0	0	0	1	1
1	1	1	1	1	1	0

Programmable ROMs

A *programmable ROM (PROM)* allows the user instead of the manufacturer to store the data. An instrument called a *PROM programmer* stores the words by "burning in." Here is an example of how a PROM programmer works. Originally, all diodes are connected at the cross points. For instance, in Fig. 4.40 there would be a total of 32 diodes (8 rows and 4 columns). Each of these diodes has a *fusible link* (a small fuse). The PROM programmer sends destructively high currents through all diodes to be removed. In this way, only the desired diodes remain connected after programming a PROM. Programming like this is permanent because the data cannot be erased after it has been burned in.

Here are some commercially available PROMs:

74S188: 256 bits organized as 32×8

74S287: 1024 bits organized as 256×4

74S472: 4096 bits organized as 512×8

PROMs such as these are useful for small production runs. For instance, if you are building only a few hundred units (or maybe even just one), you would choose a PROM rather than a ROM.

Since PROMs are useful in many applications, manufacturers produce these chips in high volume. Furthermore, the PROM is a universal logic solution. Why? Because the AND gates generate all the fundamental products; the user can then OR these products as needed to generate any Boolean output. One disadvantage of PROMs is the limit on number of input variables; typically, PROMs have 8 inputs or less.

Simplified Drawing of a PROM

It is cumbersome to draw large PROMs as illustrated in Fig. 4.41, because of the large number of diodes. An alternative, streamlined drawing procedure for PROMs like the one in Fig. 4.40 is shown in Fig. 4.41. In this simplified drawing, the solid black bullets indicate connections to the AND-gate inputs. Each bullet represents a *fixed* connection that cannot be changed. Furthermore, each AND gate has 3 inputs, indicated by the bullet on its input line. Similarly, each OR gate has 8 inputs, as indicated by the $\times$'s on its input line, but each $\times$ is a fusible link that can be removed.

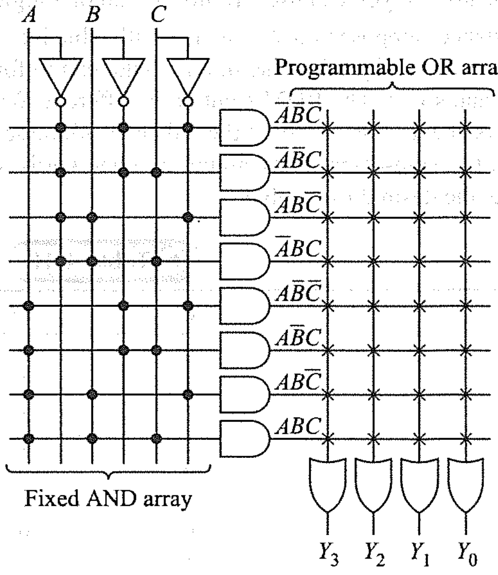
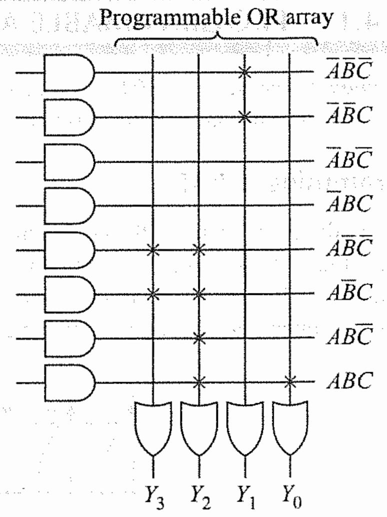


Fig. 4.41 **Streamlined drawing of PROM**

Notice that the input side of Fig. 4.41 is a fixed AND array, meaning the inputs to the AND gates are not programmable in a PROM. On the other hand, the output side of the circuit is programmable because each connection at the input of each OR gate is a fusible link. A fixed AND array and a programmable OR array are characteristic of all PROMs. To begin with, every AND-gate output is connected to every OR-gate input. Since the AND gates produce all eight possible combinations of the input variables A, B and C, it is possible to produce any Boolean function at the OR-gate outputs.

Programming a PROM

Generating a Boolean function at the output of a PROM is accomplished by *fusing* (melting) fusible links at the input to the OR gates in Fig. 4.41. For example, suppose we want to generate the function $Y_0 = ABC$. Simply fuse (melt) 7 of the AND-gate outputs connected to the Y_0 OR-gate input and leave the single AND-gate output ABC connected. A portion of Fig. 4.41 is shown in Fig. 4.43 with the proper fusible link remaining for Y_0.



Programmable OR array
$\overline{A}\overline{B}\overline{C}$
$\overline{A}\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
Y_3
Y_2
Y_1
Y_0

Fig. 4.42 **Boolean function from PROM**

As a second example, suppose we want to generate the function $Y_1 = \overline{A}\overline{B}$. We must include all terms containing $\overline{A}\overline{B}$, since

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

The two top fusible links must be included, while the remaining six are broken, as shown in Fig. 4.42. Continuing in this fashion, you can see that $Y_2 = A$ and $Y_3 = A\overline{B}$.

Erasable PROMs

The *erasable PROM (EPROM)* uses metal-oxide-semiconductor field-effect transistors (MOSFETs). Data is stored with an EPROM programmer. Later, data can be erased with ultraviolet light. The light passes through a quartz window in the IC package. When it strikes the chip, the ultraviolet light releases all stored charges. The effect is to wipe out the stored contents. In other words, the EPROM is ultraviolet-light-erasable and electrically reprogrammable.

Here are some commercially available EPROMs:

2716: 16,384 bits organized as 2048×8

2732: 32,768 bits organized as 4096×8

The EPROM is useful in project development. With an EPROM, the designer can modify the contents until the stored data is perfect. When the design is finalized, the data can be burned into PROMs (small production runs) or sent to an IC manufacturer who produces ROMs (large production runs).

SELF-TEST

20. What is a ROM?
21. What does it mean to say that a particular ROM is "512×8"?
22. What is a PROM?

4.11 PROGRAMMABLE ARRAY LOGIC

Programmable array logic (PAL) is a programmable array of logic gates on a single chip. PALs are another design solution, similar to a sum-of-products solution, product-of-sums solution, and multiplexer logic.

Programming a PAL

A PAL is different from a PROM because it has a programmable AND array and a fixed OR array. For instance. Fig. 4.43 shows a PAL with 4 inputs and 4 outputs. The ×'s on the input side are fusible links, while the solid black bullets on the output side are fixed connections. With a PROM programmer, we can burn in the desired fundamental products, which are then ORed by the fixed output connections.

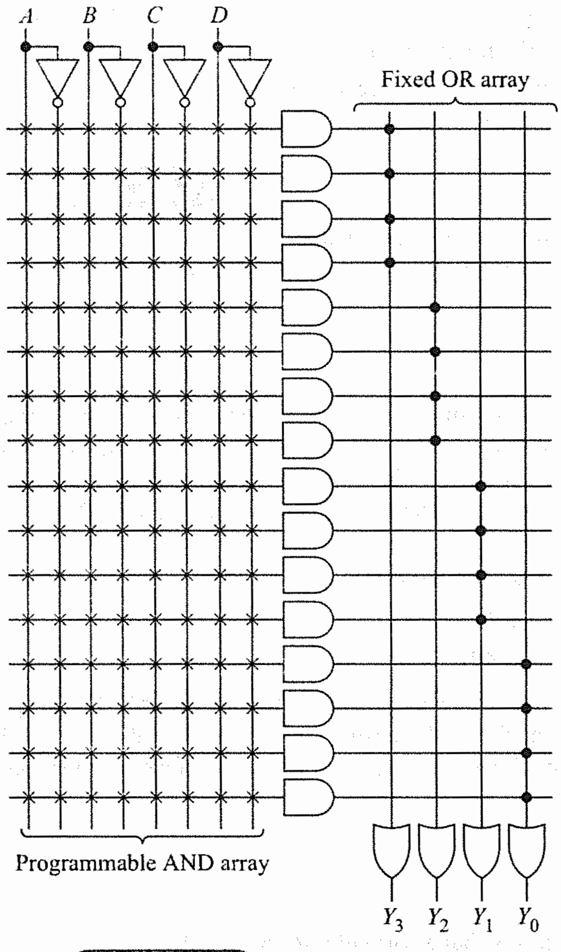


Fig. 4.43 **Structure of PAL**

Here is an example of how to program a PAL. Suppose we want to generate the following Boolean functions:

$$Y_3 = \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + \bar{A}BCD + ABC\bar{D} \quad (4.5)$$
$$Y_2 = \bar{A}BC\bar{D} + \bar{A}BCD + ABCD \quad (4.6)$$
$$Y_1 = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + AB\bar{C} \quad (4.7)$$
$$Y_0 = ABCD \quad (4.8)$$

Start with Eq. (4.5). The first desired product is $\bar{A}B\bar{C}D$. On the top input line of Fig. 4.44 we have to remove the first ×, the fourth ×, the fifth ×, and the eighth ×. Then the top AND gate has an output of $\bar{A}B\bar{C}D$.

By removing ×s on the next three input lines, we can make the top four AND gates produce the fundamental products of Eq. (4.5). The fixed OR connections on the output side imply that the first OR gate produces an output of

$$Y_3 = \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + \bar{A}BCD + ABC\bar{D}$$

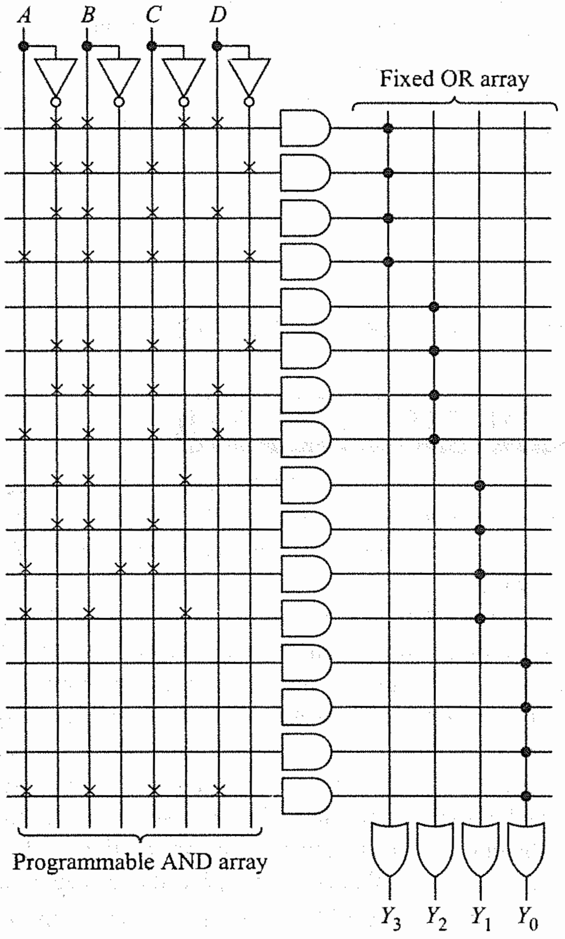


Fig. 4.44 **Example of programming a PAL**

Similarly, we can remove ×s as needed to generate Y_2, Y_1, and Y_0. Figure 4.44 shows how the PAL looks after the necessary ×s have been removed. If you examine this circuit, you will see that it produces the Y outputs given by Eqs. (4.5) to (4.8).

Commercially Available PALs

The PAL given in Fig. 4.43 is hypothetical. Commercially available PALs typically have more inputs. For instance, here is a sample of some TTL PALs available from National Semiconductor Corporation:

10H8; 10 input and 8 output AND-OR

16H2: 6 input and 2 output AND-OR

14L4: 14 input and 4 output AND-OR-INVERT

For these chip numbers, H stands for active-high output and L for active-low output. The 10H8 and the 16H2 produce active-high outputs because they are AND-OR PALs. The 14L4, on the other hand, produces an active-low output because it is an AND-OR-INVERT circuit (one that has inverters at the final outputs).

Unlike PROMs, PALs are not a universal logic solution. Why? Because only some of the fundamental products can be generated and ORed at the final outputs. Nevertheless, PALs have enough flexibility to produce all kinds of complicated logic functions. Furthermore, PALs have the advantage of 16 inputs compared to the typical limit of 8 inputs for PROMs.

SELF-TEST

23. What is a PAL?
24. A PAL has an AND array and an OR array. Which one is fixed and which is programmable?

4.12 PROGRAMMABLE LOGIC ARRAYS

Programmable logic arrays (*PLAs*), along with ROMs and PALs, are included in the more general classification of ICs called *programmable logic devices* (*PLDs*). Figure 4.45 illustrates the basic operation of these three PLDs. In each case, the input signals are presented to an array of AND gates, while the outputs are taken from an array of OR gates.

The input AND-gate array used in a PROM is *fixed* and cannot be altered, while the output OR-gate array is *fusible-linked*, and can thus be programmed. The PAL is just the opposite: The output OR-gate array is fixed, while the input AND-gate array is fusible-linked and thus programmable. The PLA is much more versatile than the PROM or the PAL, since both its AND-gate array and its OR-gate array are fusible-linked and programmable. It is also more complicated to utilize since the number of fusible links are doubled.

A PLA having 3 input variables (*ABC*) and 3 output variables (*XYZ*) is illustrated in Fig. 4.46. Eight AND gates are required to decode the 8 possible input states. In this case, there are three OR gates that can be used to generate logic functions at the output. Note that there could be additional OR gates at the output if desired. Programming the PLA is a two-step process that combines procedures used with the PROM and the PAL.

As an example, suppose it is desired to use a PLA to recognize each of the 10 decimal digits represented in binary form and to correctly drive a 7-segment display. The 7-segment indicator was presented in Sec. 4.5. To begin with, the PLA must have 4 inputs, as shown in Fig. 4.47a. Four bits (*ABCD*) are required to represent the 10 decimal numbers (see Table 1.1). There must be 7 outputs (*abcdefg*), 1 output to drive each

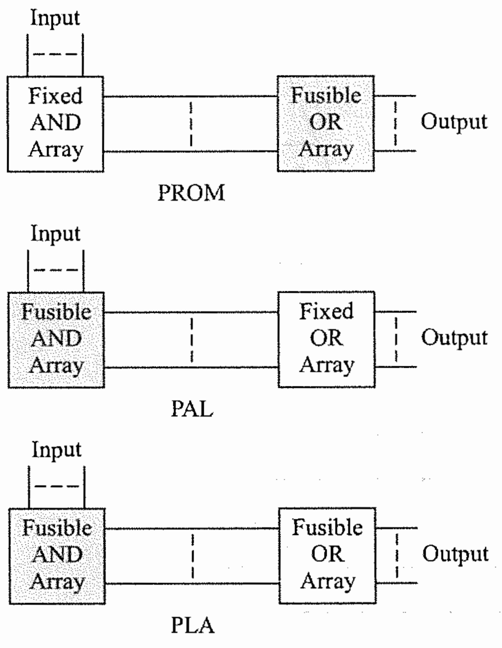


Fig. 4.45

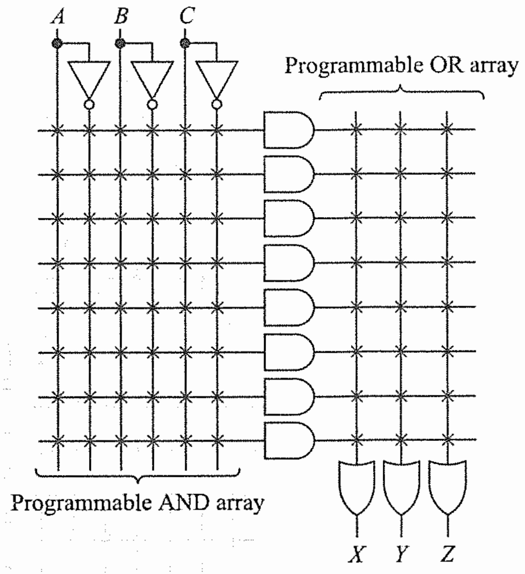


Fig. 4.46

of the 7 segments of the indicator. Let's assume that our PLA is capable of driving the 7-segment indicator directly. (This is not always a valid assumption, and a buffer amplifier may be needed to supply the proper current for the indicator.)

To begin with, all fusible links are good. The circuit in Fig. 4.47b shows the remaining links after programming. The input AND-gate array is programmed (fusible links are removed) such that each AND gate decodes one of the decimal numbers. Then, with the use of Fig. 4.47c, links are removed from the output OR-gate array such that the proper segments of the indicator are illuminated. For instance, when $ABCD = LHLH$, segments *afgcd* are illuminated to display the decimal number 5. You should take the time to examine the other nine digits to confirm proper operation.

One final point. Many PLDs are programmable only at the factory. They must be ordered from the manufacturer with specific programming instructions. There are, however, PLDs that can be programmed by the user. These are said to be *field-programmable*, and the letter F is often used to indicate this fact. For instance, the Texas Instruments TIFPLA840 is a field-programmable PLA with 14 input variables, 32 AND gates, and 6 OR gates; it is described as a 14 × 32 × 6 FPLA.

SELF-TEST

25. What is a PLA?
26. How does a PLA differ from a PAL?
27. In Fig. 4.47, $ABCD = LLHH$. What segments are activated?

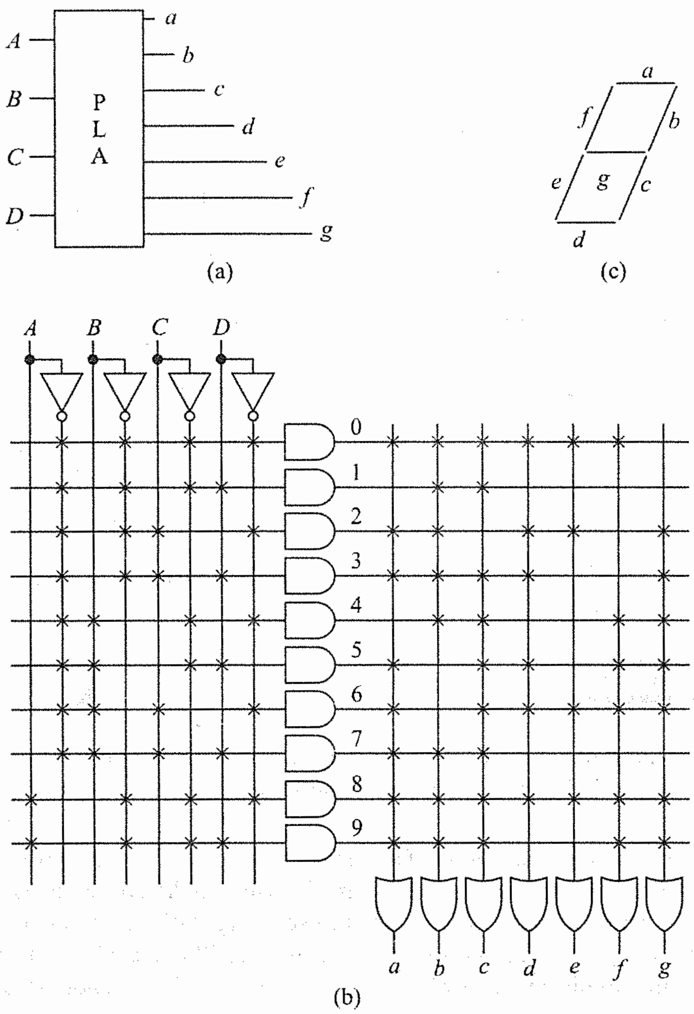


Fig. 4.47 7-segment decoder using PLA

4.13 TROUBLESHOOTING WITH A LOGIC PROBE

Chapter 3 introduced the logic clip, a device that connects to a 14 or 16-pin IC. The logic clip contains 16 LEDs that monitor the state of the pins. When a pin voltage is high, the corresponding LED lights up. When the pin voltage is low, the LED is dark.

Figure 4.48 shows a *logic probe*, which is another troubleshooting tool you will find helpful in diagnosing faulty circuits. When you touch the probe tip to the output node as shown, the device lights up for a high state and goes dark for a low state. For instance, if either A or B, or both, are low, then Y is high and the probe lights up. On the other hand, if A and B are both high, Y is low and the probe is dark.

Among other things, the probe is useful for locating short circuits that occur in manufacturing. For example, during the stuffing and soldering of printed-circuit boards, an undesirable splash of solder may connect two adjacent traces (conducting lines). Known as a *solder bridge*, this kind of trouble can short-circuit a node to the ground or to the supply voltage. The node is then stuck in a low or high state. The probe helps you to find short-circuited nodes because it stays in one state, no matter how the inputs are changing.

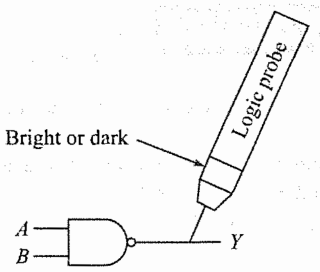


Fig. 4.48 Using a logic probe

4.14 HDL IMPLEMENTATION OF DATA PROCESSING CIRCUITS

We start with hardware design of multiplexers using Verilog code. The data flow model provides a different use of keyword **assign** in the form of

assign $X = S\ ?\ A : B$;

This statement does following assignment. If, $S = 1, X = A$ and if $S = 0, X = B$. One can use this statement or the logic equation to realize a 2 to 1 multiplexer shown in Fig. 4.2(a) in one of the following ways.

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown
  in Fig. 4.3(a)*/
  output Y;
  assign Y=(~A&D0)|(A&D1);
endmodule
```

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown in
  Fig. 4.3(a)*/
  output Y;
  assign Y= A ? D1 : D0; /*Conditional
  assignment*/
endmodule
```

The behavioral model can be used to describe the 2 to 1 multiplexers in following two different ways, one using **if ... else** statement and the other using **case** statement. The **case** evaluates an expression or a variable that can have multiple values each one corresponding to one statement in the following block. Depending on value of the expression, one of those statements get executed. The behavioral model of 2 to 1 multiplexer in both is given below:

```
module mux2to1(A,D0,D1,Y);
   input A,D0,D1; /* Circuit shown
   in Fig. 4.3(a)*/
output Y;
reg Y;
always @ (A or D0 or D1)
   if (A==1) Y=D1;
   else Y=D0;
endmodule
```

```
module mux2to1(A,D0,D1,Y);
   input A,D0,D1; /* Circuit shown
   in Fig. 4.3(a)*/
   output Y;
   reg Y;
   always @ (A or D0 or D1)
      case (A)
         0 : Y=D0;
         1 : Y=D1;
      endcase
endmodule
```

Example 4.14 Design a 4 to 1 multiplexer, shown in Fig. 4.1(c) using conditional **assign** and **case** statements.

Solution The codes are given next. We have used nested condition for **assign** statement. If A = 1, condition (B ? D3 : D2) is evaluated. Then if B = 1, Y = D3. And this is what is given in Fig. 4.1(c). Similarly, the other combinations of A and B are evaluated and Y is assigned a value from D2 to D0. For **case** statement we concatenated A and B by using operator {...} and generated four possible combinations. For a particular value of AB, statement corresponding to one of them gets executed.

```
module mux4to1(A,B,D0,D1,D2,D3,Y);
  input A,B,D0,D1,D2,D3;
  output Y; /* Circuit shown
  in Fig. 4.1(c)*/
  assign  Y = A ?( B ? D3 : D2):(B ?
  D1 : D0);
endmodule
```

```
module mux4to1(A,B,D0,D1,D2,D3,Y);
  input A,B,D0,D1,D2,D3;
  output Y;
  reg Y;
  always @ (A or B or D0 or D1 or D2
  or D3)
    case ({A,B}) /*Concatenation of
    A and B, A is MSB*/
     0: Y=D0; /*Two binary digit can
        generate*/
     1: Y=D1; /*four different values
        0,1,2,3 for*/
     2: Y=D2; /*binary combination
        00,01,10*/
     3: Y=D3; //and 11 respectively
  endcase
endmodule
```

BUS Representation in HDL

We introduce concept of BUS or vector representation in HDL description through design of a 1 to 4 demultiplexer that can also serve as a 2 to 4 decoder. The data input of former is treated as enable input of later. We consider S as a select input defined by two binary digits S[1] and S[0]. Output Y is 4 bit long, one of which goes high for a particular combination of select inputs if data(enable) input is high. The Verilog code for this demultiplexer/decoder is given below:

```
module demux1to4(S,D,Y);
  input [1:0] S;
  input  D;
  output [3:0] Y;
  reg [3:0] Y;
  always @ (S or D)
     case ({D,S}) //Concatenation of D  and S to give 3 bits, D is MSB
       3'b100 : Y= 4'b0001; /*Binary representation, refer to Section 2-5.
      If D=1, S=00, Y=0001*/
       3'b101 : Y= 4'b0010; // if D=1, S=01, Y=0010
       3'b110 : Y= 4'b0100; // if D=1, S=10, Y=0100
       3'b111 : Y= 4'b1000; // if D=1, S=11, Y=1000
       default : Y= 4'b0000; //For other combinations D=0, then Y=0000
     endcase
endmodule
```

Example 4.15 A verilog HDL code for a digital circuit is given as follows. Can you describe the function it performs? Can it be related to any logic circuit discussed in this chapter?

```
module unknown(A,B,C,Y);
  input [3:0] A,B;
  input  [2:0] C;
  output [2:0] Y;
  reg [2:0] Y;
  always @ (A or B or C)
      if (A<B) Y=3'b001;
      else if (A>B) Y=3'b010;
      else Y=C;
endmodule
```

Solution The circuit described by the HDL compares two 4-bit numbers A and B and generates a 3 bit output Y. It has also a 3 bit input C. If A is less than B, output $Y = 001$ and does not depend on C. Similarly, if A is greater than B, $Y = 010$ irrespective of C. But if these two conditions are not met, i.e. if $A = B$ then $Y = C$.

If we consider three bits of Y represent (starting from MSB) $A = B$, $A > B$ and $A < B$ respectively then, this circuit represents a 4-bit magnitude comparator where C represents comparator output of previous stage that is of lower significance. If numbers of this stage are equal then the value at C that represents equal, greater than, less than condition of previous stage numbers is reflected by Y. This is similar to IC 7485 discussed in Section 4.9.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Show how data processing circuits can be used to compare two 2-bit numbers, A_1A_0 and B_1B_0 to generate two outputs, $A > B$ and $A = B$.

Solution We can use multiplexers, decoder or simply a 4-bit comparator. The truth table of the above problem is shown in Fig. 4.49.

In Method-1, we use two 16 to 1 multiplexers to realize $A > B$ and $A = B$ as shown in Fig. 4.50. The numbers A_1A_0 and B_1B_0 are used as selection inputs as shown. For every selection of input, the

A_1A_0	B_1B_0	$A>B$	$A=B$
0 0	0 0	0	1
0 0	0 1	0	0
0 0	1 0	0	0
0 0	1 1	0	0
0 1	0 0	1	0
0 1	0 1	0	1
0 1	1 0	0	0
0 1	1 1	0	0
1 0	0 0	1	0
1 0	0 1	1	0
1 0	1 0	0	1
1 0	1 1	0	0
1 1	0 0	1	0
1 1	0 1	1	0
1 1	1 0	1	0
1 1	1 1	0	1

Fig. 4.49 **Truth table**

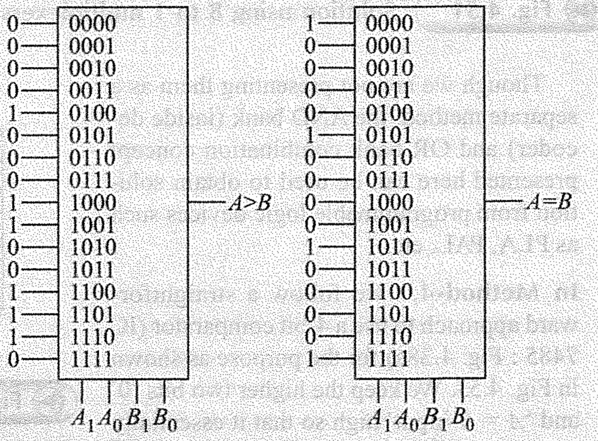


Fig. 4.50 **Solution using 16 to 1 multiplexers**

corresponding data input goes to the output. The input assignment comes straight from the truth table in Fig. 4.49 for the two cases.

In Method-2, we use two 8 to 1 multiplexers to realize $A > B$ and $A = B$ as shown in Fig. 4.51. The numbers A_1A_0 and B_1 are used as selection inputs while B_0 is part of the data input. We form pair of combinations of the truth table for constant $A_1A_0B_1$ and B_0 variable. This helps to find out how output varies with B_0.

In Method-3, we use one 4 to 16 decoder and two multi-input OR gates to realize $A > B$ and $A = B$ as shown in Fig. 4.52. We sum selected minterms, as required from the truth table, from the set of all the minterms generated by the decoder.

$A_1A_0B_1$	B_0	$A>B$	$A=B$
0 0 0	0	0 (0)	1 (B_0')
0 0 0	1	0	0
0 0 1	0	0 (0)	0 (0)
0 0 1	1	0	0
0 1 0	0	1 (B_0')	0 (B_0)
0 1 0	1	0	1
0 1 1	0	0 (0)	0 (0)
0 1 1	1	0	0
1 0 0	0	1 (1)	0 (0)
1 0 0	1	1	0
1 0 1	0	0 (0)	1 (B_0')
1 0 1	1	0	0
1 1 0	0	1 (1)	0 (0)
1 1 0	1	1	0
1 1 1	0	1 (B_0')	0 (B_0)
1 1 1	1	0	1

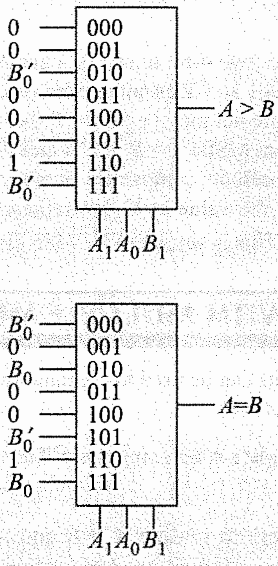


Fig. 4.51 **Solution using 8 to 1 multiplexers**

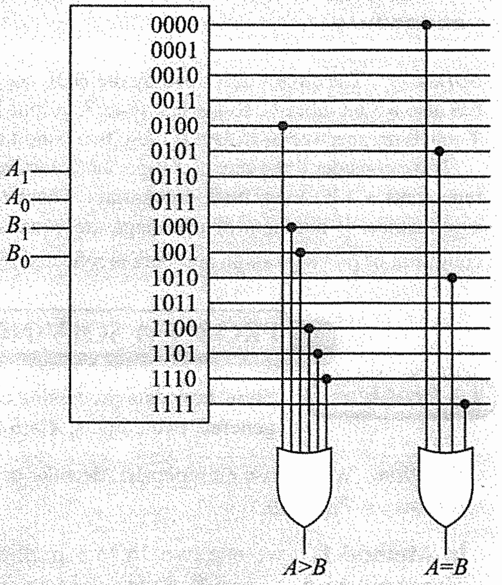


Fig. 4.52 **Solution using 4 to 16 decoder**

Though we are not presenting them as a separate method, the AND bank (inside decoder) and OR bank combination concept presented here can be used to obtain solution from programmable logic devices such as PLA, PAL, etc.

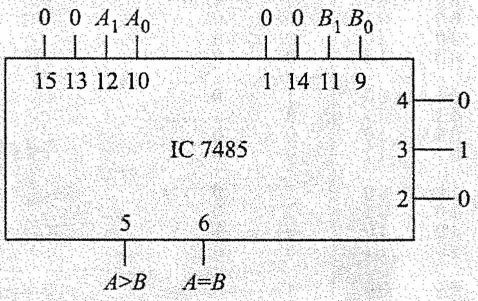


Fig. 4.53 **Solution using 4-bit comparator**

In Method-4, we follow a straightforward approach to use a 4-bit comparator (IC 7485 : Fig. 4.38a) for the purpose as shown in Fig. 4.53. We keep the higher two bits '0' and '$A = B$ input' high so that it essentially becomes a 2-bit comparator generating all three outputs $A > B$, $A = B$ and $A < B$ of which only first two are useful here.

SUMMARY

A multiplexer is a circuit with many inputs but only one output. The 16-to-1 multiplexer has 16 input bits, 4 control bits, and 1 output bit. The 4 control bits select and steer 1 of the 16 inputs to the output. The multiplexer is a universal logic circuit because it can generate any truth table.

A demultiplexer has one input and many outputs. By applying control signals, we can steer the input signal to one of the output lines. A decoder is similar to a demultiplexer, except that there is no data input. The control bits are the only input. They are decoded by activating one of the output lines.

BCD is an abbreviation for binary-coded decimal. The BCD code expresses each digit in a decimal number by its nibble equivalent. A BCD-to-decimal decoder converts a BCD input to its equivalent decimal value. A seven-segment decoder converts a BCD input to an output suitable for driving a seven-segment indicator.

An encoder converts an input signal into a coded output signal. An example is the decimal-to-BCD encoder. An exclusive-OR gate has a high output only when an odd number of inputs are high. Exclusive-OR gates are useful in parity generators-checkers.

Magnitude comparators are useful in comparing two binary numbers. It generates three outputs that give if one number is greater, equal or less than the other number. Cascading magnitude comparators we can compare two numbers of any size.

A ROM is a read-only memory. Smaller ROMs are used to implement truth tables. ROMs are expensive because they require a mask for programming. PROMs are user-programmable and ideal for small production runs. EPROMs are not only user-programmable, but they are also erasable and reprogrammable during the design and development cycle. PALs are chips that are programmable arrays of logic. Unlike the PROM with its fixed AND array and programmable OR array, a PAL has programmable AND array and a fixed OR array. The PAL has the advantage of having up to 16 inputs in commercially available devices. In the PLA both the AND array and the OR array are programmable. The PLA is a much more versatile programmable logic device (PLD) IC than the PROM or the PAL.

GLOSSARY

- ***active low*** The low state is the one that causes something to happen rather than the high state.
- ***BCD*** A binary-coded decimal.
- ***data selector*** A synonym for multiplexer.
- ***decoder*** A circuit that is similar to a demultiplexer, except there is no data input. The control input bits produce one active output line.
- ***demultiplexer*** A circuit with one input and many outputs.
- ***EPROM*** An erasable programmable read-only memory. With this device, the user can erase the stored contents with ultraviolet light and electrically store new data. EPROMs are useful during project development where programs and data are being perfected.
- ***even parity*** A binary number with an even number of 1s.
- ***exclusive-OR gate*** A gate that produces a high output only when an odd number of inputs is high.
- ***LED*** A light-emitting diode.
- ***logic probe*** A troubleshooting device that indicates the state of a signal line.
- ***Magnitude comparator*** compares two binary numbers and signals if one is greater, equal or less than other.
- ***multiplexer*** A circuit with many inputs but only one output.
- ***odd parity*** A binary number with an odd number of 1s.
- ***PAL*** A programmable array logic (sometimes written PLA, which stands for programmable

logic array). In either case, it is a chip with a programmable AND array and a fixed OR array.

- ***parity generation*** An extra bit that is generated and attached to a binary number, so that the new number has either even or odd parity.
- ***PLA*** A programmable logic array.
- ***PLD*** A programmable logic device.
- ***PROM*** A programmable read-only memory. A type of chip that allows the user to program it with a PROM programmer that burns fusible links at the diode cross points. Once the data is stored, the programming is permanent. PROMs are useful for small production runs.
- ***ROM*** A read-only memory. An IC that can store many binary numbers at locations called addresses. ROMs are expensive to manufacture and are used only for large production runs where the cost of the mask can be recovered by sales.
- ***strobe*** An input that disables or enables a circuit.

PROBLEMS

Section 4.1

4.1 In Fig. 4.2, if $ABCD$ = 1001, what does Y equal?

4.2 In Fig. 4.4, if $ABCD$ = 1100, what does Y equal?

4.3 We want to implement Table 3.12 of the preceding chapter using multiplexer logic. Show a circuit, similar to the one in Fig. 4.4, that can do the job.

4.4 Show how to connect a 74150 to implement this Boolean equation:

$$Y = \overline{A}B\overline{C}D + A\overline{B}C\overline{D} + ABC\overline{D}$$

4.5 Draw a circuit with four 74150s that has a truth table like the one in Table 4.11.

4.6 Table 4.12 shows the Gray code. Show how four 74150s may be connected to convert from binary to Gray code. Show how the same can be realized by four 74151 ICs (8-to-1 multiplexer).

Table 4.11

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	1	0
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	1
1	1	1	0	1	0	1	0
1	1	1	1	0	0	1	0

Table 4.12 Gray Code

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
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

Section 4.2

4.7 In Fig. 4.12, if $ABCD$ = 0101, which is the active output line when the strobe is high? When it is low?

4.8 Input signals R and T are low in Fig. 4.13. Which is the active output line when $ABCD$ = 0011? To have the Y_9 output line active, what input signals do you need?

4.9 Suppose a logic probe shows that pin 19, given in Fig. 4.13, is always high. Which of the following may possibly cause trouble:

a. Pin 20 is grounded.

b.Pin 18 has a sine wave instead of pulses.

c. The R input is grounded.

d.The T input is connected to +5 V.

Section 4.3

4.10 Are the output signals of Fig. 4.15 active low or active high? For the IC to decode the $ABCD$ input, does the strobe have to be low or high?

4.11 In Fig. 4.16, suppose X = 1 and $ABCD$ = 0110. Which is the active chip and which is the active output line?

4.12 Design a circuit that realizes following two functions using a decoder and two OR gates.

$$F_1(A,B) = \Sigma\, m(0,3) \quad \text{and}$$
$$F_2(A,B) = \Sigma\, m(1,2)$$

4.13 Design a circuit that realizes following three functions using a decoder and three OR gates.

$$F_1(A,B,C) = \Sigma\, m(1,3,7),$$
$$F_2(A,B,C) = \Sigma\, m(2,3,5) \quad \text{and}$$
$$F_3(A,B,C) = \Sigma\, m(0,1,5,7)$$

Section 4.4

4.14 Convert the following decimal numbers into their BCD equivalents:

a. 32 b. 634

c. 4898

4.15 Convert the following BCD numbers into their decimal equivalents:

a. 0110 0111

b. 1000 0001 0011

c. 0111 0010 0101 1001

4.16 In Fig. 4.18, what is the high output line when $ABCD$ = 0101?

4.17 In Fig. 4.20, which is the low output when $ABCD$ = 0111?

4.18 Figure 4.54 shows a group of chips numbered 0 through 9. Each chip has an active-low STROBE input. Which chip is active for each of these conditions:

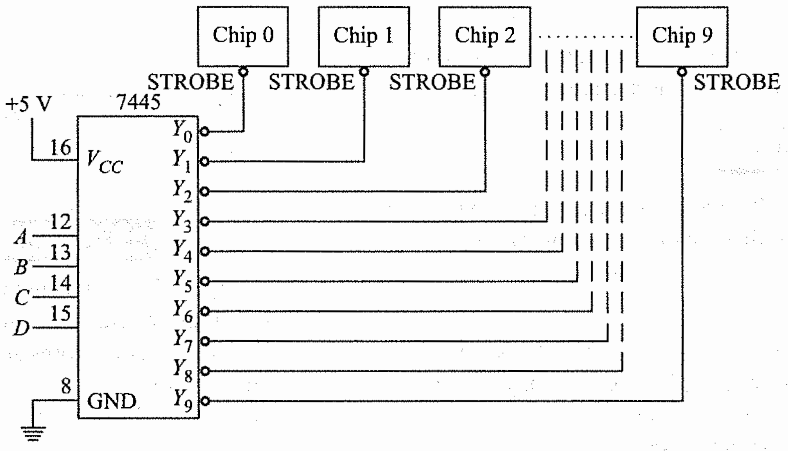


Fig. 4.54

a. $ABCD = 0000$.
b. $ABCD = 0010$.
c. $ABCD = 1001$.

4.19 The $ABCD$ input of Fig. 4.54 initially equals 1111. For this condition, all output waveforms start high in the timing diagram of Fig. 4.55. Another circuit not shown is supposed to produce the following input values of $ABCD$: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, and 1001.

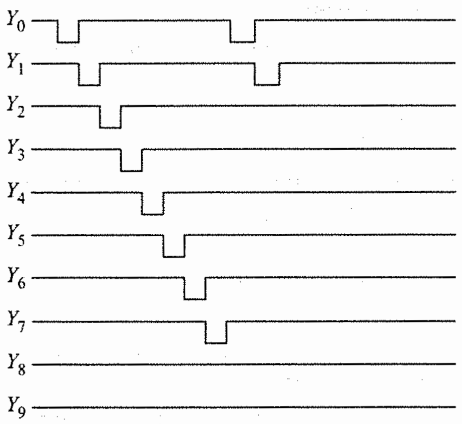


Fig. 4.55

The timing diagram tells us that something is wrong with the logic circuit of Fig. 4.54. Which of the following is a possible trouble:

a. Pin 16 is not connected to the supply voltage.
b. Pin 8 is open.
c. Pin 12 is short-circuited to the ground.
d. Pin 15 is short-circuited to +5 V.

Section 4.5

4.20 In Fig. 4.21, which of the segments have to be active to display the following digits:

a. 2
b. 6
c. 8

4.21 In Fig. 4.23a, $V_{CC} = +5$ V, all resistors are 1 kΩ, and each LED has a voltage drop of 2 V. Approximately how much current is there through an active segment?

Section 4.6

4.22 In Fig. 4.25, what is the output when button 7 is pressed? When button 3 is pressed?

4.23 In Fig. 4.27, if button 8 is pressed, which is the input pin that goes into the low state? What does the $ABCD$ output equal?

Section 4.7

4.24 In Fig. 4.32d, what does Y equal for each of the following inputs:

a. 000110
b. 011001
c. 011111
d. 111100

4.25 In Fig. 4.33, what does Y equal for the following inputs:

a. 1111	0000	1111	0000
b. 0101	1010	1100	0111
c. 1110	1011	1101	0001
d. 0001	0101	0011	0110

4.26 In Fig. 4.56, the 8-bit register is a logic circuit that stores byte $A_7 \ldots A_0$. What does byte $Y_7 \ldots Y_0$ equal for each of these conditions:

a. $A_7 \ldots A_0 = 1000\ 0111$ and INVERT = 0.
b. $A_7 \ldots A_0 = 0011\ 1100$ and INVERT = 1.
c. $A_7 \ldots A_0 = 1111\ 0000$ and INVERT = 0.
d. $A_7 \ldots A_0 = 1110\ 0001$ and INVERT = 1.

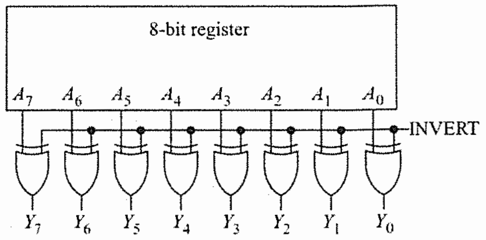


Fig. 4.56

4.27 In Fig. 4.57 on the next page, each register is a logic circuit that stores a 6-bit number. The left register stores $A_5 \ldots A_0$ and the right register stores $B_5 \ldots B_0$. What value does the output signal labeled $EQUAL$ have for each of these:

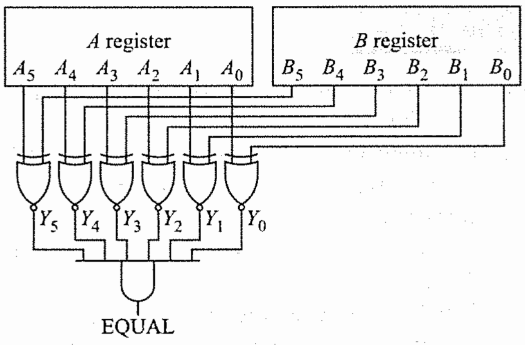

A register
A_5 A_4 A_3 A_2 A_1 A_0
B register
B_5 B_4 B_3 B_2 B_1 B_0
Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
EQUAL

Fig. 4.57

a. $A_7 \ldots A_0$ is less than $B_7 \ldots B_0$.
b. $A_7 \ldots A_0$ equals $B_7 \ldots B_0$.
c. $A_7 \ldots A_0$ is greater than $B_7 \ldots B_♦$

Sections 4.8 and 4.9

4.28 In Fig. 4.58, what does X_8 equal for each of the following $X_7 \ldots X_0$ inputs:

a. 0000 1111 b. 1111 0001
c. 1010 1110 d. 1011 1100

4.29 In Fig. 4.58, what changes can you make to get a 9-bit output with even parity?

4.30 In Fig. 4.58, assuming the circuit is working all right, what will the logic probe indicate for each of the following:

a. Input data has even parity.
b. Input data has odd parity.
c. Pins 3 and 4 are grounded.

4.31 Write the $(X > Y)$ equation for a 4-bit comparator.

4.32 Show how magnitude of two 10-bit numbers can be compared using IC 7485.

4.33 Suppose a ROM has 8 input address lines. How many memory locations does it have?

4.34 Two 74S370s are connected in parallel. To address all memory locations, how many bits must the binary address have?

4.35 In Fig. 4.40, if $ABC = 011$, what does $Y_3Y_2Y_1Y_0$ equal?

4.36 Draw a ROM circuit similar to the one in Fig. 4.40 that produces these outputs:

$Y_3 = \overline{A}B\overline{C}$

$Y_2\ A\overline{B}C + ABC$

$Y_1 = A\overline{B}C + \overline{A}BC + ABC$

$Y_0 = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} + ABC$

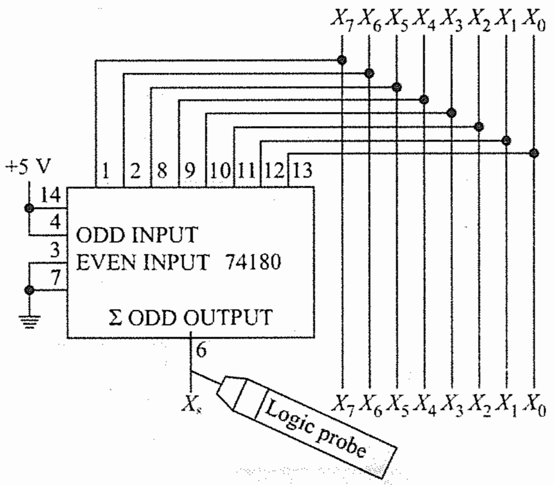


Fig. 4.58

Section 4.10

4.37 Draw a PROM circuit similar to the one in Fig. 4.41 that generates the Y_3 to Y_0 output given in Table 4.11.

4.38 What is the Boolean equation for Y_3 in Fig. 4.59 on the previous page? For Y_2? For Y_1? For Y_0?

4.39 Draw a 4-input and 4-output PAL circuit that has the truth table of Table 4.11.

Section 4.11

4.40 Write the Boolean expression for the output Y_3 in Fig. 4.42.

4.41 The input to the PLA in Fig. 4.47 is $ABCD$ = 0011. What segments of the indicator are illuminated and what decimal number is displayed? What if $ABCD$ = 1001? What about 1111?

4.42 Will there be any ambiguity if segment g of the 7-segment indicator in Fig. 4.47 is defective (burned out)? What numbers are displayed?

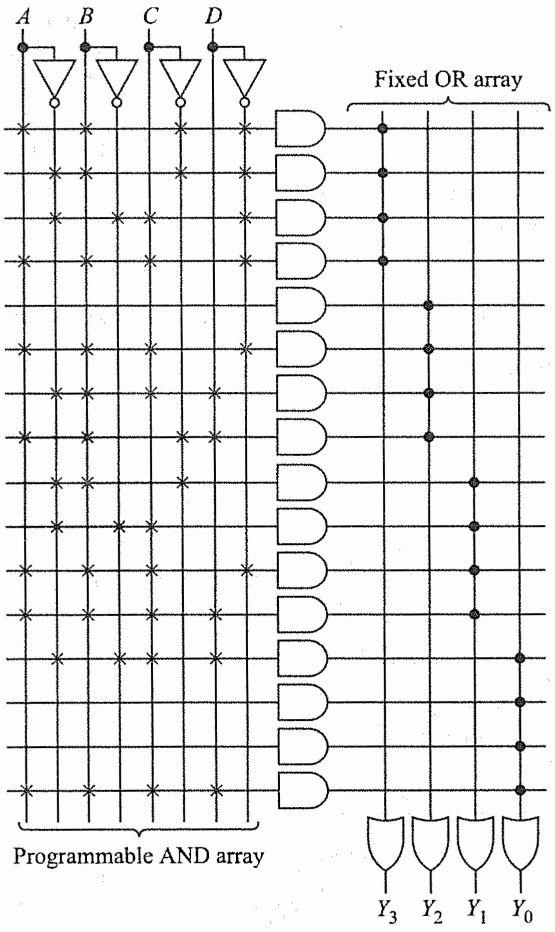
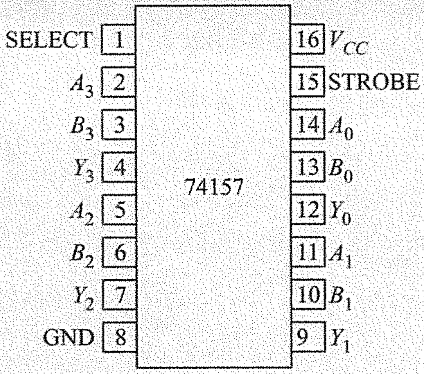


Fig. 4.59

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to display one of two BCD numbers in a 7-segment display.

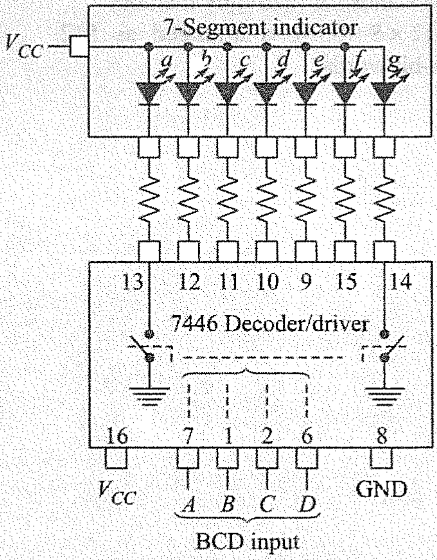
Theory: The two BCD numbers can be selected by activating select line of a multiplexer.



The multiplexer output then is one of the two BCD numbers. These outputs can be connected to four inputs of a 7-segment decoder/driver. The outputs of this driver can be connected to a 7-segment display to display the decimal equivalent of the BCD number selected.

Apparatus: 5 V DC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of multiplexer IC 74157. Note that STROBE is an input and find its use. The common select line applies to all four 2-to-1 multiplexers. Verify the truth table of IC7446 for BCD inputs. Select



resistance values to be connected in the range 220 to 1000 ohm. This is to ensure that entire power supply voltage does not drop across the LED of display. Interconnect properly all the different units and verify.

Answers to Self-tests

1. Multiplexer
2. It means that Y is active low.
3. Demultiplexer
4. $ABCD = HLHL$.
5. It will be high since STROBE is high.
6. The outputs are active low.
7. Y_{10} is low; all other outputs are high.
8. BCD stands for binary-coded decimal.
9. LED stands for light-emitting diode.
10. See Fig. 4.21.
11. Each segment is an LED.
12. An encoder converts an active input signal into a coded output signal, for instance, decimal to binary.
13. The TTL 74147 is a decimal-to-binary encoder.

14. The output for an exclusive-OR gate is high only when an *odd* number of inputs are high.
15. See Fig. 4.30.
16. There are an *even* number of 1s (highs).
17. True.
18. Three: $X = Y$, $X > Y$ and $X < Y$.
19. Three.
20. ROM stands for read-only memory.
21. A 512×8 ROM is arranged as 512 eight-bit words.
22. PROM stands for programmable read-only memory.
23. PAL stands for programmable array logic.
24. The AND array is programmable; the OR array is fixed.
25. PLA stands for programmable logic array.
26. In a PLA, both the AND array and the OR array are programmable.
27. Decimal 3; segments *abcdg*

Number Systems and Codes

5

OBJECTIVES

- ✦ Convert decimal numbers to binary and convert binary numbers to decimal
- ✦ Convert binary and decimal numbers to octal and convert octal numbers to binary and decimal
- ✦ Convert binary and decimal numbers to hexadecimal and convert hexadecimal numbers to binary and decimal
- ✦ Describe the ASCII code, excess-3 code, and Gray code
- ✦ Understand Error Detection and Correction Code

5.1 BINARY NUMBER SYSTEM

The *binary number system* is a system that uses only the digits 0 and 1 as codes. All other digits (2 to 9) are thrown away. To represent decimal numbers and letters of the alphabet with the binary code, you have to use different strings of binary digits for each number or letter. The idea is similar to the Morse code, where strings of dots and dashes are used to code all numbers and letters. What follows is a discussion of decimal and binary counting.

Decimal Odometer

To understand how to count with binary numbers, it helps to review how an odometer (miles indicator of a car) counts with decimal numbers. When a car is new, its odometer starts with

00000

After 1 km the reading becomes

00001

Successive kms produce 00002, 00003, and so on, up to

00009

A familiar thing happens at the end of the tenth km. When the units wheel turns from 9 back to 0, a tab on this wheel forces the tens wheel to advance by 1. This is why the numbers change to

00010

Reset-and-Carry

The units wheel has reset to 0 and sent a carry to the tens wheel. Let's call this familiar action *reset and carry*. The other wheels of an odometer also reset and carry. For instance, after 999 kms the odometer shows

00999

What does the next km do? The units wheel resets and carries, the tens wheel resets and carries, the hundreds wheel resets and carries, and the thousands wheel advances by 1, to get

01000

Binary Odometer

Visualize a binary odometer as a device whose wheels have only two digits, 0 and 1. When each wheel turns, it displays 0, then 1, then back to 0, and the cycle repeats. A four-digit binary odometer starts with

0000 (zero)

After 1 mile, it indicates

0001 (one)

The next mile forces the units wheel to reset and carry, so the numbers change to

0010 (two)

The third mile results in

0011 (three)

After 4 miles, the units wheel resets and carries, the second wheel resets and carries, and the third wheel advances by 1:

0100 (four)

Table 5.1 shows all the binary numbers from 0000 to 1111, equivalent to decimal 0 to 15. Study this table carefully and practice counting from 0000 to 1111 until you can do it easily. Why? Because all kinds of logic circuits are based on counting from 0000 to 1111.

The word *bit* is the abbreviation for binary digit. Table 5.1 is a list of 4-bit number from 0000 to 1111. When a binary number has 4 bits, it is sometimes called a *nibble*. Table 5.1 shows 16 nibbles (0000 to 1111). A binary number with 8 bits is known as a *byte*; this has become the basic unit of data used in computers. You will learn

Table 5.1 4-Digit Binary Numbers

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

more about bits, nibbles, and bytes in later chapters. For now memorise these definitions:

$$\text{bit} = \text{X}$$
$$\text{nibble} = \text{XXXX}$$
$$\text{byte} = \text{XXXXXXXX}$$

where the X may be a 0 or a 1.

SELF-TEST

1. What is the binary number for decimal 13?
2. What is the decimal equivalent of binary 1001?
3. How many binary digits (bits) are required to represent decimal 15?

5.2 BINARY-TO-DECIMAL CONVERSION

Table 5.1 lists the binary numbers from 0000 to 1111. But how do you convert larger binary numbers into their decimal values? For instance, what does binary 101001 represent in decimal numbers? This section shows how to convert a binary number quickly and easily into its decimal equivalent.

Positional Notation and Weights

We can express any decimal *integer* (a whole number) in units, tens, hundreds, thousands, and so on. For instance, decimal number 2945 may be written as

$$2945 = 2000 + 900 + 40 + 5$$

In powers of 10, this becomes

$$2945 = 2(10^3) + 9(10^2) + 4(10^1) + 5(10^0)$$

The decimal number system is an example of *positional notation*, each digit position has a *weight* or value. With decimal numbers, the weights are units, tens, hundreds, thousands, and so on. The sum of all the digits multiplied by their weights gives the total amount being represented. In the foregoing example, the 2 is multiplied by a weight of 1000, the 9 by a weight of 100, the 4 by a weight of 10, and the 5 by a weight of 1; the total is

$$2000 + 900 + 40 + 5 = 2945$$

Binary Weights

In a similar way, we can rewrite any binary number in terms of weights. For instance, binary number 111 becomes

$$111 = 100 + 10 + 1 \tag{5.1}$$

In decimal numbers, this may be rewritten as

$$7 = 4 + 2 + 1 \tag{5.2}$$

Writing a binary number as shown in Eq. (5.1) is the same as splitting its decimal equivalent into units, 2s, and 4s as indicated by Eq. (5.2). In other words, each digit position in a binary number has a weight. The least significant digit (the one on the right) has a weight of 1. The second position from the right has a weight of 2; the next, 4; and then 8, 16, 32, and so forth. These weights are in ascending powers of 2; therefore, we can write the foregoing equation as

$$7 = 1(2^2) + 1(2^1) + 1(2^0)$$

Table 5.2 **Binary System**

Bit Position	*Weight*
1 (Right most)	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Whenever you look at a binary number, you can find its decimal equivalent as follows:

1. When there is a 1 in a digit position, add the weight of that position.
2. When there is a 0 in a digit position, disregard the weight of that position. For example, binary number 101 has a decimal equivalent of

$$4 + 0 + 1 = 5$$

As another example, binary number 1101 is equivalent to

$$8 + 4 + 0 + 1 = 13$$

Still another example is 11001, which is equivalent to

$$16 + 8 + 0 + 0 + 1 = 25$$

Streamlined Method

We can streamline binary-to-decimal conversion by the following procedure:

1. Write the binary number.
2. Directly under the binary number write 1, 2, 4, 8, 16 … , working from right to left.
3. If a zero appears in a digit position, cross out the decimal weight for that position.
4. Add the remaining weights to obtain the decimal equivalent.

As an example of this approach, let us convert binary 101 to its decimal equivalent:

STEP 1 1 0 1
STEP 2 4 2 1
STEP 3 4 $\cancel{2}$ 1
STEP 4 4 + 1 = 5

As another example, notice how quickly 10101 is converted to its decimal equivalent:

$$\begin{array}{ccccc} 1 & 0 & 1 & 0 & 1 \\ 16 & \cancel{8} & 4 & \cancel{2} & 1 \rightarrow 21 \end{array}$$

Fractions

So far, we have discussed binary *integers* (whole numbers). How are binary fractions converted into corresponding decimal equivalents? For instance, what is the decimal equivalent of 0.101? In this case, the weights of digit positions to the right of the binary point are given by $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$, and so on. In powers of 2, the weights are

$$2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad \text{etc.}$$

or in decimal form:

0.5 0.25 0.125 0.0625 etc.

Here is an example. Binary fraction 0.101 has a decimal equivalent of

0.1 0 1

0.5 + 0 + 0.125 = 0625

Another example, the decimal equivalent of 0.1101 is

0.1 1 0 1

0.5 + 0.25 + 0 + 0.0625 = 0.8125

Mixed Numbers

For *mixed* numbers (numbers with an integer and a fractional part), handle each part according to the rules just developed. The weights for a mixed number are

etc. 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} etc.

↑

Binary point

For future reference, Table 5.3 lists powers of 2 and their decimal equivalents and the numbers of K and M. The abbreviation K stands for 1024. Therefore, 1K means 1024. 2K stands for 2048, 4K represents 4096, and so on. The abbreviation M stands for 1,048,576, which is equivalent to 1024K (1024 × 1024 = 1,048,576). A memory chip that stores 4096 bits is called a "4K memory." A digital device might have a memory capacity of 4,194,304 bytes. This would be referred to as a "4-megabyte (Mb) memory."

Table 5.3 **Powers of 2**

Powers of 2	*Decimal Equivalent*	*Abbreviation*
2^0	1	
2^1	2	
2^2	4	
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1,024	1K
2^{11}	2,048	2K
2^{12}	4,096	4K
2^{13}	8,192	8K
2^{14}	16,384	16K
2^{15}	32,768	32K
2^{16}	65,536	64K
2^{17}	131,072	128K
2^{18}	262,144	256K
2^{19}	524,288	512K
2^{20}	1,048,576	1,024K = 1M
2^{21}	2,097,152	2,048K = 2M
2^{22}	4,194,304	4,096K = 4M

Example 5.1 Convert binary 110.001 to a decimal number.

Solution

1	1	0	0	0	1
4	2	~~1~~	~~0.5~~	~~0.25~~	0.125 → 6.125

Example 5.2 What is the decimal value of binary 1011.11?

Solution

1	0	1	1	1	1
8	~~4~~	2	1	0.5	0.25 → 11.75

Example 5.3 A computer has a 2 Mb memory. What is the decimal equivalent of 2 Mb?

Solution

$$2 \times 1{,}048{,}576 = 2{,}097{,}152$$

This means that the computer can store 2,097,152 bytes in its memory.

SELF-TEST

4. What is the decimal equivalent of 10010?
5. What is the binary equivalent of 35?
6. A binary number has 9 bits. What is the binary weight of the most significant bit?

5.3 DECIMAL-TO-BINARY CONVERSION

One way to convert a decimal number into its binary equivalent is to reverse the process described in the preceding section. For instance, suppose that you want to convert decimal 9 into the corresponding binary number. All you need to do is express 9 as a sum of powers of 2, and then write 1s and 0s in the appropriate digit positions:

$$9 = 8 + 0 + 0 + 1$$
$$\rightarrow 1001$$

As another example:

$$25 = 16 + 8 + 0 + 0 + 1$$
$$\rightarrow 11001$$

Double Dabble

A popular way to convert decimal numbers to binary numbers is the *double-dabble method.* In the double-dabble method you progressively divide the decimal number by 2, writing down the remainder after each division. The remainders, taken in reverse order, form the binary number. The best way to understand the method is to go through an example step by step. Here is how to convert decimal 13 to its binary equivalent

Step 1 Divide 13 by 2, writing your work like this:

$$2\overset{6}{\overline{)13}} \qquad 1 \rightarrow \text{(first remainder)}$$

The quotient is 6 with a remainder of 1.

Step 2 Divide 6 by 2 to get

$$\begin{array}{ll} 2\overset{3}{\overline{)6}} & 0 \rightarrow \text{(second remainder)} \\ 2\overline{)13} & 1 \rightarrow \text{(first remainder)} \end{array}$$

This division gives 3 with a remainder of 0.

Step 3 Again you divide by 2:

$$\begin{array}{rl}
1 & \\
2\overline{)3} & 0 \rightarrow \text{(second remainder)} \\
2\overline{)6} & 1 \rightarrow \text{(first remainder)} \\
2\overline{)13} & 1 \rightarrow \text{(first remainder)}
\end{array}$$

Here you get a quotient of 1 with a remainder of 1.

Step 4 One more division gives

$$\begin{array}{rcl}
0 & & \\
2\overline{)1} & 1 & \mapsto \text{(fourth remainder)} \\
2\overline{)3} & 1 & \\
2\overline{)6} & 0 & \text{Read down} \\
2\overline{)13} & 1 & \downarrow
\end{array}$$

In this final division 2 does not divide into 1; thus, the quotient is 0 with a remainder of 1.

Whenever you arrive at a quotient of 0 with a remainder of 1, the conversion is finished. The remainders when read downward give the binary equivalent. In this example, binary 1101 is equivalent to decimal 13.

There is no need to keep writing down 2 before each division because you are always dividing by 2. Here is an efficient way to show the conversion of decimal 13 to its binary equivalent:

$$\begin{array}{rcl}
0 & 1 & \\
1 & 1 & \\
3 & 0 & \text{Read down} \\
6 & 1 & \downarrow \\
2\overline{)13} & &
\end{array}$$

Fractions

As far as fractions are concerned, you *multiply* by 2 and record a carry in the integer position. The carries read downward are the binary fraction. As an example, 0.85 converts to binary as follows:

$$\begin{array}{ll}
0.85 \times 2 = 1.7 = 0.7 \text{ with a carry of } 1 & \\
0.7 \times 2 = 1.4 = 0.4 \text{ with a carry of } 1 & \\
0.4 \times 2 = 0.8 = 0.8 \text{ with a carry of } 0 & \text{Read down} \\
0.8 \times 2 = 1.6 = 0.6 \text{ with a carry of } 1 & \\
0.6 \times 2 = 1.2 = 0.2 \text{ with a carry of } 1 & \\
0.2 \times 2 = 0.4 = 0.4 \text{ with a carry of } 0 & \downarrow
\end{array}$$

Reading the carries downward gives binary fraction 0.110110. In this case, we stopped the conversion process after getting six binary digits. Because of this, the answer is an approximation. If more accuracy is needed, continue multiplying by 2 until you have as many digits as necessary for your application.

Useful Equivalents

Table 5.4 shows some decimal-binary equivalences. This will be useful in the future. The table has an important property that you should be aware of. Whenever a binary number has all 1s (consists of only 1s), you can find its decimal equivalent with this formula:

$$\text{Decimal} = 2^{n-1}$$

where n is the number of bits. For instance, 1111 has 4 bits; therefore, its decimal equivalent is

$$\text{Decimal} = 2^4 - 1 = 16 - 1 = 15$$

Table 5.4 Decimal-Binary Equivalences

Decimal	*Binary*
1	1
3	11
7	111
15	1111
31	1 1111
63	11 1111
127	111 1111
255	1111 1111
511	1 1111 1111
1,023	11 1111 1111
2,047	111 1111 1111
4,095	1111 1111 1111
8,191	1 1111 1111 1111
16,383	11 1111 1111 1111
32,767	111 1111 1111 1111
65,535	1111 1111 1111 1111

As another example, 1111 1111 has 8 bits, so

$$\text{Decimal} = 2^8 - 1 = 256 - 1 = 255$$

BCD-8421 and BCD-2421 Code

Binary Coded Decimal (BCD) refers to representation of digits 0–9 in decimal system by 4-bit unsigned binary numbers. The usual method is to follow 8421 encoding which employs conventional route of weight placements like 8 representing the weight of the 4th place (as $2^{4-1} = 8$), 4, i.e. 2^{3-1} of the 3rd place, 2, i.e. 2^{2-1} of the 2nd place and 1, i.e. 2^{1-1} of the 1st place. The 2421 code is similar to 8421 code except for the fact that the weight assigned to 4th place is 2 and not 8. The decimal numbers 0–9 in these two codes then can be represented as shown in Table 5.5.

Table 5.5 BCD-8421 and BCD-2421 Code

Decimal	*BCD-8421*	*BCD-2421*
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1011
6	0110	1100
7	0111	1101
8	1000	1110
9	1001	1111

As an example, decimal number 29 in BCD-8421 is written as 00101001 (0010 representing 2 and 1001 representing 9) while in BCD-2421, it is written as 00101111 (0010 representing 2 and 1111 representing 9).

Example 5.4 Convert decimal 23.6 to a binary number.

Solution Split decimal 23.6 into an integer of 23 and a fraction of 0.6, and apply double dabble to each part.

$$\begin{array}{rr|l}
0 & 1 & \\
1 & 0 & \\
2 & 1 & \text{Read down} \\
5 & 1 & \\
11 & 1 & \\
2\overline{)23} & & \downarrow
\end{array}$$

and

$$\begin{array}{l|l}
0.6 \times 2 = 1.2 = 0.2 \text{ with a carry of } 1 & \\
0.2 \times 2 = 0.4 = 0.4 \text{ with a carry of } 0 & \\
0.4 \times 2 = 0.8 = 0.8 \text{ with a carry of } 0 & \text{Read down} \\
0.8 \times 2 = 1.6 = 0.6 \text{ with a carry of } 1 & \\
0.6 \times 2 = 0.2 = 0.2 \text{ with a carry of } 1 & \downarrow
\end{array}$$

The binary number is 10111.10011. This 10-bit number is an approximation of decimal 21.6 because we terminated the conversion of the fractional part after 5 bits.

Example 5.5 A digital computer processes binary numbers that are 32 bits long. If a 32-bit number has all 1s, what is its decimal equivalent?

Solution

$$\begin{aligned}
\text{Decimal} &= 2^{32} - 1 = (2^8)(2^8)(2^8)(2^8) - 1 \\
&= (256)(256)(256)(256) - 1 = 4{,}294{,}967{,}295
\end{aligned}$$

SELF-TEST

7. What is double dabble?
8. A binary number is composed of twelve 1s. What is its decimal equivalent?
9. What is the binary number for decimal 255?

5.4 OCTAL NUMBERS

The base of a number system equals the number of digits it uses. The decimal number system has a base of 10 because it uses the digits 0 to 9. The binary number system has a base of 2 because it uses only the digits 0 and 1. The *octal number system* has a base of 8. Although we can use any eight digits, it is customary to use the first eight decimal digits:

0, 1, 2, 3, 4, 5, 6, 7

(There is no 8 or 9 in the octal number code.) These digits, 0 through 7, have exactly the same physical meaning as decimal symbols; that is, 2 stands for ●●, 5 symbolizes ●●●●●, and so on.

Octal Odometer

The easiest way to learn how to count in octal numbers is to use an *octal odometer*. This hypothetical device is similar to the odometer of a car, except that each display wheel contains only eight digits, numbered 0 to 7. When a wheel turns from 7 back to 0, it sends a carry to the next-higher wheel.

Initially, an octal odometer shows

0000 (zero)

The next 7 kms produces readings of

0001	(one)
0002	(two)
0003	(three)
0004	(four)
0005	(five)
0006	(six)
0007	(seven)

At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

0010 (eight)

The next 7 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, and 0017. Once again, the least-significant wheel has run out of digits. So the next km results in a reset and carry:

0020 (sixteen)

Subsequent kms produce readings of 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0030, 0031, and so on.

You should have the idea by now. Each km advances the least-significant wheel by one. When this wheel runs out of octal digits, it resets and carries. And so on for the other wheels. For instance, if the odometer reading is 6377, the next octal number is 6400.

Octal-to-Decimal Conversion

How do we convert octal numbers to decimal numbers? In the octal number system each digit position corresponds to a power of 8 as follows:

$$8^3\ 8^2\ 8^1\ 8^0\ \underset{\substack{\uparrow \\ \text{Octal point}}}{.}\ 8^{-1}\ 8^{-2}\ 8^{-3}$$

Therefore, to convert from octal to decimal, multiply each octal digit by its weight and add the resulting products. Note that $8^0 = 1$.

For instance, octal 23 converts to decimal like this:

$$2(8^1) + 3(8^0) = 16 + 3 = 19$$

Here is another example. Octal 257 converts to

$$2(8^1) + 5(8^1) + 7(8^0) = 128 + 40 + 7 = 175$$

Decimal-to-Octal Conversion

How do you convert in the opposite direction, that is, from decimal to octal? *Octal dabble*, a method similar to double dabble, is used with octal numbers. Instead of dividing by 2 (the base of binary numbers), you divide by 8 (the base of octal numbers) writing down the remainders after each division. The remainders in reverse order form the octal number. As an example, convert decimal 175 as follows:

$$\begin{array}{rl} 0 & \\ 8\overline{)2} & 2 \rightarrow \text{(third remainder)} \\ 8\overline{)21} & 5 \rightarrow \text{(second remainder)} \\ 8\overline{)175} & 7 \rightarrow \text{(first remainder)} \end{array}$$

You can condense these steps by writing

$$\begin{array}{rcl} 0 & 2 & \\ 2 & 5 & \text{Read down} \\ 21 & 7 & \downarrow \\ 8\overline{)175} & & \end{array}$$

Thus decimal 175 is equal to octal 257.

Fractions

With decimal fractions, multiply instead of divide, writing the carry into the integer position. An example of this is to convert decimal 0.23 into an octal fraction.

$$\begin{array}{ll} 0.23 \times 8 = 1.84 = 0.84 & \text{with a carry of 1} \\ 0.84 \times 8 = 6.72 = 0.72 & \text{with a carry of 6} \quad \text{Read down} \\ 0.72 \times 8 = 5.76 = 0.76 & \text{with a carry of 5} \downarrow \\ \text{etc.} & \end{array}$$

The carries read downward give the octal fraction 0.165. We terminated after three places; for more accuracy, we would continue multiplying to obtain more octal digits.

Octal-to-Binary Conversion

Because 8 (the base of octal numbers) is the third power of 2 (the base of binary numbers), you can convert from octal to binary as follows: change each octal digit to its binary equivalent. For instance, change octal 23 to its binary equivalent as follows:

$$\begin{array}{cc} 2 & 3 \\ \downarrow & \downarrow \\ 010 & 011 \end{array}$$

Here, each octal digit converts to its binary equivalent (2 becomes 010, and 3 becomes 011). The binary equivalent of octal 23 is 010 011, or 010011. Often, a space is left between groups of 3 bits; this makes it easier to read the binary number.

As another example, octal 3574 converts to binary as follows:

$$\begin{array}{cccc} 3 & 5 & 7 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 011 & 101 & 111 & 100 \end{array}$$

Hence binary 011101111100 is equivalent to octal 3574. Notice how much easier the binary number is to read if we leave a space between groups of 3 bits: 011 101 111 100.

Mixed octal numbers are no problem. Convert each octal digit to its equivalent binary value. Octal 34.562 becomes

$$\begin{array}{cccccc} 3 & 4 & . & 5 & 6 & 2 \\ \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow \\ 011 & 100 & . & 101 & 110 & 010 \end{array}$$

Binary-to-Octal Conversion

Conversion from binary to octal is a reversal of the foregoing procedures. Simply remember to group the bits in threes, starting at the binary point; then convert each group of three to its octal equivalent (0s are added at each end, if necessary). For instance, binary number 1011.01101 converts as follows:

$$\begin{array}{lccccc} 1011.01101 \rightarrow & 001 & 011. & 011 & 010 \\ & \downarrow & \downarrow & \downarrow & \downarrow \\ & 1 & 3 & 3 & 2 \end{array}$$

Start at the binary point and, working both ways, separate the bits into groups of three. When necessary, as in this case, add 0s to complete the outside groups. Then convert each group of three into its binary equivalent. Therefore:

$$1011.01101 = 13.32$$

The simplicity of converting octal to binary and vice versa has many advantages in digital work. For one thing, getting information into and out of a digital system requires less circuitry because it is easier to read and print out octal numbers than binary numbers. Another advantage is that large decimal numbers are more easily converted to binary if first converted to octal and then to binary, as shown in Example 5.6.

Example 5.6 What is the binary equivalent of decimal 363?

Solution One approach is double dabble. Another approach is octal dabble, followed by octal-to-binary conversion. Here is how the second method works:

$$\begin{array}{r|l} 0 & 5 \\ 5 & 5 \\ 45 & 3 \\ 8\overline{)363} & \end{array} \quad \downarrow \text{ Read down}$$

Next, convert octal 553 to its binary equivalent:

$$\begin{array}{ccc} 5 & 5 & 3 \\ \downarrow & \downarrow & \downarrow \\ 101 & 101 & 011 \end{array}$$

The double-dabble approach would produce the same answer, but it is tedious because you have to divide by 2 nine times before the conversion terminates.

SELF-TEST

10. What are the digits used in the octal number system?
11. What is the octal number for binary 111? What is the decimal number for binary 111?

5.5 HEXADECIMAL NUMBERS

Hexadecimal numbers are used extensively in microprocessor work. To begin with, they are much shorter than binary numbers. This makes them easy to write and remember. Furthermore, you can mentally convert them to binary whenever necessary.

The hexadecimal number system has a base of 16. Although any 16 digits may be used, everyone uses 0 to 9 and A to F as shown in Table 5.6. In other words, after reaching 9 in the hexadecimal system, you continue counting as follows:

A, B, C, D, E, F

Table 5.6 **Hexadecimal Digits**

Decimal	*Binary*	*Hexadecimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Hexadecimal Odometer

The easiest way to learn how to count in hexadecimal numbers is to use a *hexadecimal odometer*. This hypothetical device is similar to the odometer of a car, except that each display wheel has 16 digits, numbered 0 to F. When a wheel turns from F back to 0, it sends a carry to the next higher wheel.

Initially, a hexadecimal odometer shows

0000 (zero)

The next 9 kms produces readings of

0001	(one)
0002	(two)
0003	(three)
0004	(four)
0005	(five)
0006	(six)
0007	(seven)
0008	(eight)
0009	(nine)

The next 6 kms gives

000A	(ten)
000B	(eleven)
000C	(twelve)
000D	(thirteen)
000E	(fourteen)
000F	(fifteen)

At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

0010 (sixteen)

The next 15 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, 0017, 0018, 0019, 001A, 001B, 001C, 001D, 001E, and 001F. Once again, the least significant wheel has run out of digits. So, the next km results in a reset and carry:

0020 (thirty-two)

Subsequent kms produce readings of 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0028, 0029, 002A, 002B, 002C, 002D, 002E, and 002F.

You should have the idea by now. Each km advances the least-significant wheel by one. When this wheel runs out of hexadecimal digits, it resets and carries, and so on for the other wheels. For instance, here are three more examples:

Number	Next number
835C	835D
A47F	A480
BFFF	C000

Hexadecimal-to-Binary Conversion

To convert a hexadecimal number to a binary number, convert each hexadecimal digit to its 4-bit equivalent using the code given in Table 5.5. For instance, here's how 9AF converts to binary:

$$\begin{tikzcd} 9 \arrow[d] & A \arrow[d] & F \arrow[d] \\ 1001 & 1010 & 1111 \end{tikzcd}$$

As another example, C5E2 converts like this:

$$\begin{tikzcd} C \arrow[d] & 5 \arrow[d] & E \arrow[d] & 2 \arrow[d] \\ 1100 & 0101 & 1110 & 0010 \end{tikzcd}$$

Binary-to-Hexadecimal Conversion

To convert in the opposite direction, from binary to hexadecimal, again use the code from Table 5.5. Here are two examples. Binary 1000 1100 converts as follows:

$$\begin{tikzcd} 1000 \arrow[d] & 1100 \arrow[d] \\ 8 & C \end{tikzcd}$$

Binary 1110 1000 1101 0110 converts like this:

$$\begin{tikzcd} 1110 \arrow[d] & 1000 \arrow[d] & 1101 \arrow[d] & 0110 \arrow[d] \\ E & 8 & D & 6 \end{tikzcd}$$

In both these conversions, we start with a binary number and wind up with the equivalent hexadecimal number.

Hexadecimal-to-Decimal Conversion

How do we convert hexadecimal numbers to decimal numbers? In the hexadecimal number system each digit position corresponds to a power of 16. The weights of the digit positions in a hexadecimal number are as follows

$$\begin{array}{cccccccc} 16^3 & 16^2 & 16^1 & 16^0 & . & 16^{-1} & 16^{-2} & 16^{-3} \\ &&&& \uparrow &&& \end{array}$$

Hexadecimal point

Therefore, to convert from hexadecimal to decimal, multiply each hexadecimal digit by its weight and add the resulting products. Note that $16^0 = 1$.

Here's an example. Hexadecimal F8E6.39 converts to decimal as follows:

$$\begin{aligned} \text{F8E6} &= \text{F}(16^3) + 8(16^2) + \text{E}(16^1) + 6(16^0) + 3(16^{-1}) + 9(16^{-2}) \\ &= 15(16^3) + 8(16^2) + 14(16^1) + 6(16^0) + 3(16^{-1}) + 9(16^{-2}) \\ &= 61{,}440 + 2048 + 224 + 6 + 0.1875 + 0.0352 \\ &= 63{,}718.2227 \end{aligned}$$

Decimal-to-Hexadecimal Conversion

One way to convert from decimal to hexadecimal is the *hex dabble*. The idea is to divide successively by 16, writing down the remainders. Here's a sample of how it's done. To convert decimal 2479 to hexadecimal, the first division is

$$\begin{array}{rl} 154 & \qquad 15 \rightarrow \text{F} \\ 16\overline{)2479} & \end{array}$$

In this first division, we get a quotient of 154 with a remainder of 15 (equivalent to F). The next step is

$$\begin{array}{rl} 9 & \qquad 10 \rightarrow \text{A} \\ 154 & \qquad 15 \rightarrow \text{F} \\ 16\overline{)2479} & \end{array}$$

Here we obtain a quotient of 9 with a remainder of 10 (same as A). The final step is

$$\begin{array}{rll} 0 & \qquad 9 \rightarrow 9 & \Big| \\ 9 & \qquad 10 \rightarrow \text{A} & \Big| \qquad \text{Read down} \\ 154 & \qquad 15 \rightarrow \text{F} & \downarrow \\ 16\overline{)2479} & & \end{array}$$

Therefore, hexadecimal 9AF is equivalent to decimal 2479.

Notice how similar hex dabble is to double dabble. Notice also that remainders greater than 9 have to be changed to hexadecimal digits (10 becomes A, 15 becomes F, etc.).

Using Appendix 1*

A typical microcomputer can store up to 65,535 bytes. The decimal addresses of these bytes are from 0 to 65,535. The equivalent binary addresses are from

```
0000    0000    0000    0000
1111    1111    1111    1111
```

The first 8 bits are called the *upper byte*, and the second 8 bits are the *lower byte*.

If you have to do many conversions between binary, hexadecimal, and decimal, learn to use Appendix 1. It has four headings: *binary, hexadecimal, upper byte,* and *lower byte*. For any decimal number between 0 and 255, you would use the binary, hexadecimal, and lower byte columns. Here is the recommended way to use Appendix 1. Suppose you want to convert binary 0001 1000 to its decimal equivalent. First, mentally convert to hexadecimal:

$$0001\ 1000 \rightarrow 18 \quad \text{(mental conversion)}$$

Next, look up hexadecimal 18 in Appendix 1 and read the corresponding decimal value from the lower-byte column:

$$18 \rightarrow 24 \quad \text{(look up in Appendix 1)}$$

For another example, binary 1111 0000 converts like this:

$$1111\ 0000 \rightarrow \text{F0} \rightarrow 240$$

The reason for mentally converting from binary to hexadecimal is that you can more easily locate a hexadecimal number in Appendix 1 than a binary number. Once you have the hexadecimal equivalent, you can read the lower-byte column to find the decimal equivalent.

When the decimal number is greater than 255, you have to use both the upper byte and the lower byte in Appendix 1. For instance, suppose you want to convert this binary number to its decimal equivalent:

$$1110\ 1001\ 0111\ 0100$$

First, convert the upper byte to its decimal equivalent as follows:

$$1110\ 1001 \rightarrow \text{E9} \rightarrow 59{,}648 \quad \text{(upper byte)}$$

Second, convert the lower byte to its decimal equivalent:

$$0111\ 0100 \rightarrow 74 \rightarrow 116 \quad \text{(lower byte)}$$

Finally, add the upper and lower bytes to obtain the total decimal value:

$$59{,}648 + 116 = 59{,}764$$

Therefore, binary 1110 1001 0111 0100 is equivalent to decimal 59,764.

Once you get used to working with Appendix 1, you will find it to be a quick and easy way to convert between the number systems. Because it covers the decimal numbers from 0 to 65,535, Appendix 1 is extremely useful for microprocessors where the typical memory addresses are over the same decimal range.

* A number of hand calculators will convert binary, octal, decimal and hexadecimal numbers.

Example 5.7 A computer memory can store thousands of binary instructions and data. A typical microprocessor has 65,536 addresses, each storing a byte. Suppose that the first 16 addresses contain these bytes:

```
0011   1100
1100   1101
0101   0111
0010   1000
1111   0001
0010   1010
1101   0100
0100   0000
0111   0111
1100   0011
1000   0100
0010   1000
0010   0001
0011   1010
0011   1110
0001   1111
```

Convert these bytes to their hexadecimal equivalents.

Solution Here are the stored bytes and their hexadecimal equivalents:

Memory contents	Hexadecimal equivalents
0011 1100	3C
1100 1101	CD
0101 0111	57
0010 1000	28
1111 0001	F1
0010 1010	2A
1101 0100	D4
0100 0000	40
0111 0111	77
1100 0011	C3
1000 0100	84
0010 1000	28
0010 0001	21
0011 1010	3A
0011 1110	3E
0001 1111	1F

What is the point of this example? When discussing the contents of a computer memory, we can use either binary numbers or hexadecimal numbers. For instance, we can say that the first address contains 0011 1100, or we can say that it contains 3C. Either way, we obtain the same information. But notice how much easier it is to say, write, and think 3C than it is to say, write, and think 0011 1100. In other words, hexadecimal numbers are much easier for people to work with.

Example 5.8 Convert the hexadecimal numbers of the preceding example to their decimal equivalents.

Solution The first address contains 3C, which converts like this:

$$3(16^1) + C(16^0) = 48 + 12 = 60$$

Even easier, look up the decimal equivalent of 3C in Appendix 1, and you get 60. Either by powers of 16 or with reference to Appendix 1, we can convert the other memory contents to get the following:

Memory contents	Hexadecimal equivalents	Decimal equivalents
0011 1100	3C	60
1100 1101	CD	205
0101 0111	57	87
0010 1000	28	40
1111 0001	F1	241
0010 1010	2A	42
1101 0100	D4	212
0100 0000	40	64
0111 0111	77	119
1100 0011	C3	195
1000 0100	84	132
0010 1000	28	40
0010 0001	21	33
0011 1010	3A	58
0011 1110	3E	62
0001 1111	1F	31

Example 5.9 Convert decimal 65,535 to its hexadecimal and binary equivalents.

Solution Use hex dabble as follows:

$$\begin{array}{rl|l}
0 & 15 \rightarrow F & \\
15 & 15 \rightarrow F & \text{Read down} \\
255 & 15 \rightarrow F & \\
4095 & 15 \rightarrow F & \downarrow \\
16\overline{)65{,}535} & &
\end{array}$$

Therefore, decimal 65,535 is equivalent to hexadecimal FFFF.

Next, convert from hexadecimal to binary as follows:

$$\begin{array}{cccc}
F & F & F & F \\
\downarrow & \downarrow & \downarrow & \downarrow \\
1111 & 1111 & 1111 & 1111
\end{array}$$

This means that hexadecimal FFFF is equivalent to binary 1111 1111 1111 1111.

Example 5.10 Show how to use Appendix 1 to convert decimal 56,000 to its hexadecimal and binary equivalents.

Solution The first thing to do is to locate the largest decimal number equal to 56.000 or less in Appendix 1. The number is 55,808, which converts like this:

$$55{,}808 \rightarrow \text{DA} \quad \text{(upper byte)}$$

Next, you need to subtract this upper byte from the original number:

$$56{,}000 - 55{,}808 = 192 \quad \text{(difference)}$$

This difference is always less than 256 and represents the lower byte, which Appendix 1 converts as follows:

$$192 \rightarrow \text{C0}$$

Now, combine the upper and lower byte to obtain

$$\text{DAC0}$$

which you can mentally convert to binary:

$$\text{DAC0} \rightarrow 1101\ 1010\ 1100\ 0000$$

Example 5.11 Convert Table 5.4 into a new table with three column headings: "Decimal," "Binary," and "Hexadecimal."

Solution This is easy. Convert each group of bits to its hexadecimal equivalent as shown in Table 5.7.

Table 5.7 **Decimal-Binary-Hexadecimal Equivalences**

Decimal	*Binary*	*hexadecimal*
1	1	1
3	11	3
7	111	7
15	1111	F
31	11111	1F
63	111111	3F
127	1111111	7F
255	11111111	FF
511	111111111	1FF
1,023	1111111111	3FF
2,047	11111111111	7FF
4,095	111111111111	FFF
8,191	1111111111111	1FFF
16,383	11111111111111	3FFF
32,767	111111111111111	7FFF
65,535	1111111111111111	FFFF

SELF-TEST

12. What are the symbols used in hexadecimal numbers?
13. What is the binary equivalent of hexadecimal 3C?
14. What is the decimal equivalent of hexadecimal 3C?

5.6 THE ASCII CODE

To get information into and out of a computer, we need to use some kind of *alphanumeric* code (one for letters, numbers, and other symbols). At one time, manufacturers used their own alphanumeric codes, which led to all kinds of confusion. Eventually, industry settled on an input-output code known as the *American Standard Code for Information Interchange* (ASCII, pronounced ask'-ee). This code allows manufacturers to standardize computer hardware such as keyboards, printers, and video displays.

Using the Code

The ASCII code is a 7-bit code whose format is

$$X_6X_5X_4X_3X_2X_1X_0$$

where each X is a 0 or a 1. Use Table 5.8 to find the ASCII code for the uppercase and lowercase letters of the alphabet and some of the most commonly used symbols. For example, the table shows that the capital letter A has an $X_6X_5X_4$ of 100 and an $X_3X_2X_1X_0$ of 0001. The ASCII code for A is, therefore,

1000001

For easier reading, we can leave a space as follows:

100 0001 (A)

The letter a is coded as

110 0001 (a)

More examples are

110 0010 (b)

Table 5.8 ASCII Code

	$X_6X_5X_4$					
$X_3X_2X_1X_0$	*010*	*011*	*100*	*101*	*110*	*111*
0000	SP	0	@	P		p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	'	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K		k	
1100	,	<	L		l	
1101	–	=	M		m	
1110	•	>	N		n	
1111	/	?	O		o	

110 0011	(c)
110 0100	(d)

and so on.

Also, study the punctuation and mathematical symbols. Some examples are

010 0100	($)
010 1011	(+)
011 1101	(=)

In Table 5.7, SP stands for space (blank). Hitting the space bar of an ASCII keyboard sends this into a microcomputer:

010 0000 (space)

Parity Bit

The ASCII code is used for sending digital data over telephone lines. As mentioned in the preceding chapter, 1-bit errors may occur in transmitted data. To catch these errors, a parity bit is usually transmitted along with the original bits. Then a parity checker at the receiving end can test for even or odd parity, whichever parity has been prearranged between the sender and the receiver. Since ASCII code uses 7 bits, the addition of a parity bit to the transmitted data produces an 8-bit number in this format:

$$\underset{\substack{\uparrow\\ \text{Parity bit}}}{X_7}X_6X_5X_4 \;\; X_3X_2X_1X_0$$

This is an ideal length because most digital equipment is set up to handle bytes of data.

EBCDIC as Alphanumeric Code

There exists few others but relatively less used alphanumeric codes. The EBCDIC is an abbreviation of Extended Binary Coded Decimal Interchange Code. It is an eight-bit code and primarily used in IBM make devices. Here, the binary codes of letters and numerals come as an extension of BCD code. The bit assignments of EBCDIC are different from the ASCII but the character symbols are the same.

Example 5.12 With an ASCII keyboard, each keystroke produces the ASCII equivalent of the designated character. Suppose that you type PRINT X. What is the output of an ASCII keyboard?

Solution The sequence is as follows: P (101 0000), R (101 0010), I (100 1001), N (100 1110), T (101 0100), space (010 000), X (101 1000).

Example 5.13 A computer sends a message to another computer using an odd-parity bit. Here is the message in ASCII code, plus the parity bit:

1100 1000
0100 0101
0100 1100
0100 1100
0100 1111

What do these numbers mean?

Solution First, notice that each 8-bit number has odd parity, an indication that no 1-bit errors occurred during transmission. Next, use Table 5.7 to translate the ASCII characters. If you do this correctly, you get a message of HELLO.

SELF-TEST

15. What is the ASCII code?
16. What symbol is represented by the ASCII code 100 0000?
17. What ASCII code is used for the percent sign, %?

5.7 THE EXCESS-3 CODE

The *excess-3 code* is an important 4-bit code sometimes used with binary-coded decimal (BCD) numbers. To convert any decimal number into its excess-3 form, add 3 to each decimal digit, and then convert the sum to a BCD number.

For example, here is how to convert 12 to an excess-3 number. First, add 3 to each decimal digit:

$$\begin{array}{r} 1 \\ +3 \\ \hline 4 \end{array} \qquad \begin{array}{r} 1 \\ +3 \\ \hline 5 \end{array}$$

Second, convert the sum to BCD form:

$$\begin{array}{cc} 4 & 5 \\ \downarrow & \downarrow \\ 0100 & 0101 \end{array}$$

So, 0100 0101 in the excess-3 code stands for decimal 12.

Take another example; convert 29 to an excess-3 number:

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \\ \downarrow \\ 0101 \end{array} \qquad \begin{array}{r} 9 \\ +3 \\ \hline 12 \\ \downarrow \\ 1100 \end{array}$$

After adding 9 and 3, do *not* carry the 1 into the next column; instead, leave the result intact as 12, and then convert as shown. Therefore, 0101 1100 in the excess-3 code stands for decimal 29.

Table 5.9 shows the excess-3 code. In each case, the excess-3 code number is 3 greater than the BCD equivalent. Such coding helps in BCD arithmetic as 9's complement of any excess-3 coded number can be obtained simply by complementing each bit. Take for example decimal number 2. Its 9's complement is 9 – 2 = 7. Excess-3 code of 2 is 0101. Complementing each bit we get 1010 and its decimal equivalent is 7. To convert BCD to excess-3 we need an adder and for the reverse we need a subtractor. These circuits are discussed in the next chapter. Incidentally, if you need an integrated circuit (IC) that converts from excess 3 to decimal, look at the data sheet of a 7443. This transistor-transistor logic (TTL) chip has four input lines for the excess-3 input and 10 output lines for the decoded decimal output.

Table 5.9 Excess-3 Code

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

5.8 THE GRAY CODE

The advantage of such coding will be understood from this example. Let an object move along a track and move from one zone to another. Let the presence of the object in one zone is sensed by sensors ABC. If consecutive zones are binary coded then zone-0 is represented by $ABC = 000$, zone-1 by $ABC = 001$, zone-2 by $ABC = 010$ and so on, as shown in Fig. 5.1a. Now consider, the object moves from zone-1 to zone-2. Both BC has to change to sense that movement. Suppose, sensor B (may be an electro-mechanical switch) reacts slightly late than sensor C. Then, initially $ABC = 000$ is sensed as if the object has moved in the other direction from zone-1 to zone-0. This problem can be more prominent if the object moves from zone-3 ($ABC = 011$) to zone-4 ($ABC = 100$) when all three sensors has to change its value. Note that, if zones are gray coded (Fig. 5.1b) such problem does not appear as between two consecutive zones only one sensor changes its value.

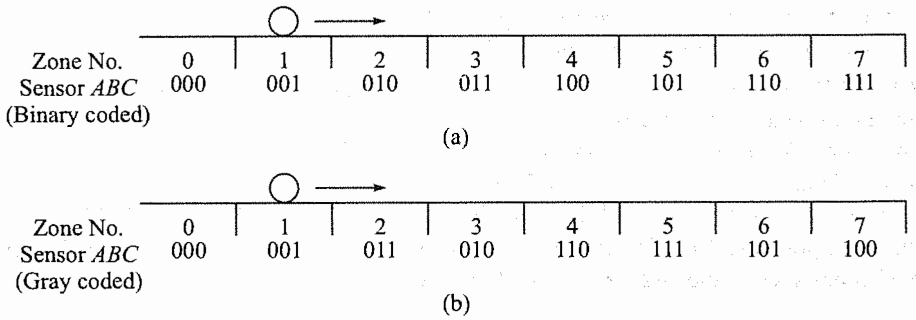

Zone No. 0 1 2 3 4 5 6 7
Sensor ABC 000 001 010 011 100 101 110 111
(Binary coded)
(a)
Zone No. 0 1 2 3 4 5 6 7
Sensor ABC 000 001 011 010 110 111 101 100
(Gray coded)
(b)

Fig. 5.1 **Object moving along a track with sensors: (a) Binary coded, (b) Gray coded**

The disadvantage with gray code is that it is not good for arithmetic operation. However, comparing truth tables of binary coded numbers and gray coded numbers (Table 5.18) we can design binary to gray converter as shown in Fig. 5.2a and gray to binary converter as shown in Fig. 5.2b. Let's see how these circuits work by taking one example each.

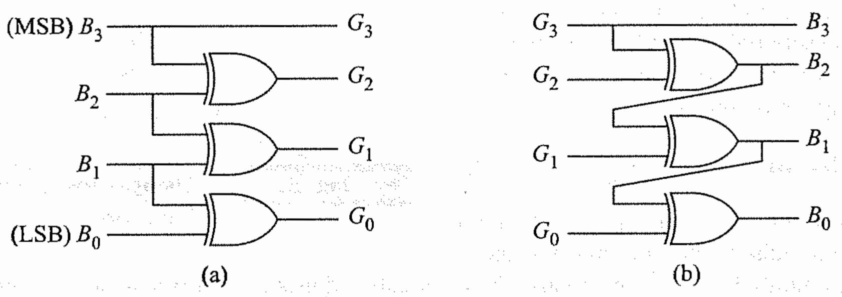


Fig. 5.2 **(a) Binary to Gray converter, (b) Gray to Binary converter**

Consider, a binary number $B_3B_2B_1B_0 = 1011$. Following the relation shown in Fig. 5.2a we get, $G_3 = B_3 = 1$, $G_2 = B_3 \oplus B_2 = 1 \oplus 0 = 1$, $G_1 = B_2 \oplus B_1 = 0 \oplus 1 = 1$ and $G_0 = B_1 \oplus B_0 = 1 \oplus 1 = 0$, i.e. $G_3G_2G_1G_0 = 1110$ and we can verify the same from truth table.

Similarly, for a gray coded number say, $G_3G_2G_1G_0$ = 0111 from Fig. 5.2b we get, $B_3 = G_3 = 0$, $B_2 = G_3 \oplus G_2 = 0 \oplus 1 = 1$, $B_1 = B_2 \oplus G_1 = 1 \oplus 1 = 0$ and $B_0 = B_1 \oplus G_0 = 0 \oplus 1 = 1$, i.e. $B_3B_2B_1B_0 = 0101$. Again this conversion can be verified from Table 5.10 that shows the *Gray code*, along with the corresponding binary numbers. Each Gray-code number differs from any adjacent number by a single bit. For instance, in going from decimal 7 to 8, the Gray-code numbers change from 0100 to 1100; these numbers differ only in the most significant bit. As another example, decimal numbers 13 and 14 are represented by Gray-code numbers 1011 and 1001; these numbers differ in only one digit position (the second position from the right). So, it is with the entire Gray code; every number differs by only 1 bit from the preceding number.

Table 5.10 **Gray Code**

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

Besides the excess-3 and Gray codes, there are other binary-type codes. Appendix 5 lists some of these codes for future reference. Incidentally, the BCD code is sometimes referred to as the *8421 code* because the weights of the digit positions from left to right are 8, 4, 2, and 1. As shown in Appendix 5, there are many other weighted codes such as the 7421, 6311, 5421, and so on.

5.9 TROUBLESHOOTING WITH A LOGIC PULSER

Figure 5.3 shows a typical *logic pulser*, a troubleshooting tool that generates a brief voltage pulse when its push-button switch is pressed. Because of its design, the logic pulser (on the left) senses the original state of the node and produces a voltage pulse of the opposite polarity. When this happens, the logic probe (on the right) blinks, indicating a temporary change of output state.

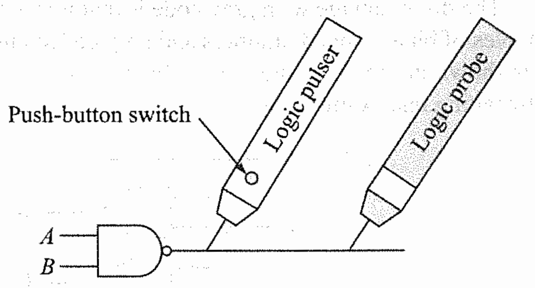


Fig. 5.3 **Using a logic pulser and a logic probe**

Thévenin Circuit

Figure 5.4a shows the Thévenin equivalent circuit for a typical logic pulser. The Thévenin voltage is a pulse with an amplitude of 5 V; the polarity automatically adjusts to the original state of the test node. As shown, the Thévenin resistance or output impedance is only 2Ω. This Thévenin resistance is representative; the exact value depends on the particular logic pulser being used. Typically, a TTL gate has an output resistance between 12Ω (low state) and 70Ω (high state). When a logic pulser drives the output of a NAND gate, the equivalent circuit appears as shown in Fig. 5.4b. Because of the low output impedance (2Ω) of the logic pulser, most of the voltage pulse appears across the load (12 to 70Ω). Therefore, the output is briefly driven into the opposite voltage state.

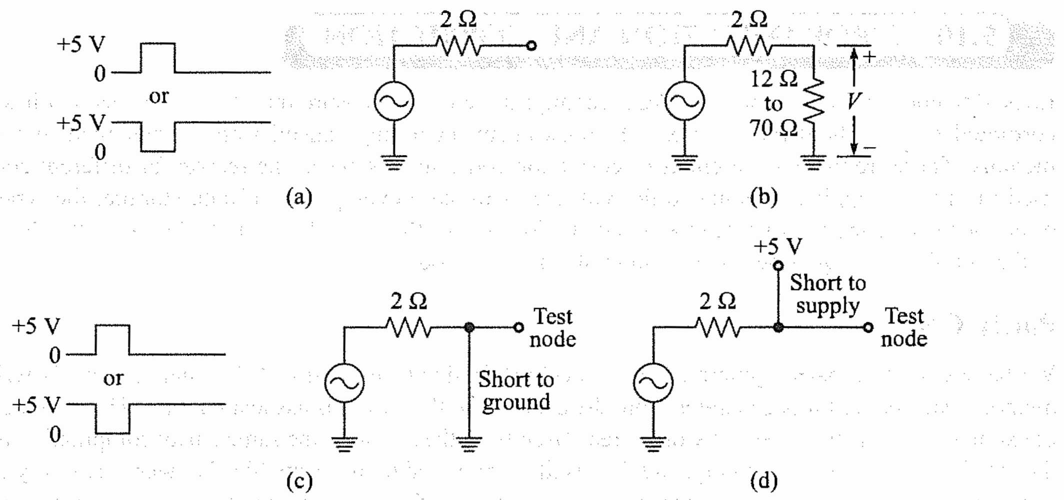


Fig. 5.4 **(a) Thévenin equivalent of logic pulser, (b) Logic pulser driving NAND-gate output, (c) Node stuck in high state**

Testing Any Node

You can use a logic pulser to drive any node in a circuit, whether input or output. Almost always, the load impedance of the node being driven is larger than the output impedance of the logic pulser. For this reason, the logic pulser can usually change the state of any node in a logic circuit. Also, the pulse width is kept very short (fractions of a microsecond) to avoid damaging the circuit being tested. (*Note: Power dissipation* is what damages ICs. A brief voltage pulse produces only a small power dissipation.)

Stuck Nodes

When is a logic pulser unable to change the state of a node? When the test node is shorted to ground or to the supply voltage. For instance, Fig. 5.4c shows the test node shorted to ground. In this case, all the voltage pulse is dropped across the internal impedance of the logic pulser; therefore the test node is stuck at 0 V, the low state.

On the other hand, the test node may be shorted to the supply voltage as shown in Fig. 5.4d. Most power supplies are regulated and have impedances in fractions of 1 Ω. For this reason, most of the voltage pulse is again dropped across the output impedance of the logic pulser, which means that the test node is stuck at +5V.

Finding Stuck Nodes

If a circuit is faulty, you can use a logic pulser and logic probe to locate stuck nodes. Here's how. Touch both the logic pulser and the logic probe to a node as shown in Fig. 5.3. If the node is stuck in either state, the logic pulser will be unable to change the state. So, if the logic probe does not blink, you have a stuck node. Then, you can look for solder bridges on any trace connected to the stuck node, or possibly replace the IC having the stuck node.

5.10 ERROR DETECTION AND CORRECTION

Error Detection and Correction (EDAC) techniques are used to ensure that data is correct and has not been corrupted, either by hardware failures or by noise occurring during transmission or a data read operation from memory. There are many different error correction codes in existence. The reason for different codes being used in different applications has to do with the historical development of data storage, the types of data errors occurring, and the overhead associated with each of the error-detection techniques. We discuss some of the popular techniques here with details of Hamming code.

Parity Code

We have discussed parity generation and checking in detail in Section 4.8. When a word is written into memory, each parity bit is generated from the data bits of the byte it is associated with. This is done by a tree of exclusive-OR gates. When the word is read back from the memory, the same parity computation is done on the data bits read from the memory, and the result is compared to the parity bits that were read. Any computed parity bit that does not match the stored parity bit indicates that there was at least one error in that byte (or in the parity bit itself). However, parity can only detect an odd number of errors. If even number of errors occur, the computed parity will match the read parity, so the error will go undetected. Since memory errors are rare if the system is operating correctly, the vast majority of errors will be single-bit errors, and will be detected.

Unfortunately, while parity allows for the detection of single-bit errors, it does not provide a means of determining which bit is in error, which would be necessary to correct the error. Thus the data needs to be read again if an error is detected. Error Correction Code (ECC) is an extension of the parity concept.

Checksum Code

This is a kind of error detection code used for checking a large block of data. The checksum bits are generated by summing all the codes of a message and are stored with data. Usually the block of data summed is of length 512 or 1024 and the checksum results are stored in 32 bits that allow overflow. When data is read, the summing operation is again done and checksum bits generated are matched with the stored one. If they are unequal, then an error is detected. Obviously, it can fool the detection system if error occurring at one place is compensated by the other.

Cycle Redundancy Code (CRC)

CRC code is a more robust error checking algorithm than the previous two. The code is generated in the following manner. Take a binary message and convert it to a polynomial, then divide it by another predefined polynomial called the *key*. The remainder from this division is the CRC. This is stored with the message. Upon reading the data, memory controller does the same operation, i.e. divides the message by the same key and compares with CRC stored. If they differ, then the data has been wrongly read or stored. Not all keys are equally good. The longer the key, the better is the error checking. On the other hand, the calculations with long keys can get quite complex. Two of the polynomials commonly used are:

$$\text{CRC–16} = x_{16} + x_{15} + x_2 + 1$$

$$\text{CRC–32} = x_{32} + x_{26} + x_{23} + x_{22} + x_{16} + x_{12} + x_{11} + x_{10} + x_8 + x_7 + x_5 + x_4 + x_2 + x + 1$$

Usually, series of exclusive-OR gates are used to generate CRC code. We shall see in the next chapter that the sum term arising out of addition is essentially an exclusive-OR operation.

Hamming Code

Introduced in 1950 by R W Hamming, this scheme allows one bit in the word to be corrected, but is unable to correct events where more than one bit in the word is in error. These multi-bit errors can only be detected, not corrected, and therefore will cause a system to malfunction. Hamming code uses parity bits discussed before but in a different way. For n number of data bits, if number of parity bits required here is m, then

$$2^m \geq m + n + 1$$

In the memory word, (i) all bit positions that are of the form 2^i are used as parity bits (like 1, 2, 4, 8, 16, 32...) and (ii) the remaining positions are used as data bits (like 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 17, 18...) Thus code will be in the form of

P1 P2 D3   P4 D5 D6 D7   P8 D9 D10 D11 ...

where P1, P2, P4, P8... are parity bits and D3, D5, D6, D7... are data bits.

We discuss Hamming code generation with an example. Consider the 7-bit data to be coded is 0110101. This requires 4 parity bits in position 1, 2, 4 and 8 so that Hamming coded data becomes 11-bit long. To calculate the value of P1, we check parity of zeroth binary locations of data bits. This is shown in 3rd row of Fig. 5.5 for this example. Zeroth locations are the places where address ends with a 1. These are D3, D5, D9 and D11 for 7-bit data. Since we have total odd number of 1s in these 4 positions P1 = 1. This is calculated as done in case of parity generation (refer to Section 4.8) by series of exclusive-OR gates through the equation

$$P1 = D3 \oplus D5 \oplus D9 \oplus D11$$

Similarly for P2, we check locations where we have 1 in address of the 1st bit, i.e. D3, D6, D7, D10 and D11. Since there are even number of 1s, P2 = 1. Proceeding in similar manner and examining parity of 2nd and 3rd position, we get P4 = 0 and P8 = 0.

	0001 P1	0010 P2	0011 D3	0100 P4	0101 D5	0110 D6	0111 D7	1000 P8	1001 D9	1010 D10	1011 D11
Data word (without parity)			0		1	1	0		1	0	1
P1	1		0		1		0		1		1
P2		0	0			1	0			0	1
P4				0	1	1	0				
P8								0	1	0	1
Data with parity	1	0	0	0	1	1	0	0	1	0	1

Fig. 5.5 Calculation of Parity Bits

Next we discuss how error in a Hamming coded data is detected and if it is in single bit, how it is corrected. We continue with the previous example and consider that the data is incorrectly read in position D11 so that 11-bit coded data is 10001100100. Figure 5.6 describes the detection mechanism. First of all, we check the parity of zeroth position and find it to be even. Since P1 = 1, the parity check fails and this is equivalent to generating a parity bit at the output (last column) following the equation

$$\text{Parity P1 check bit} = D3 \oplus D5 \oplus D9 \oplus D11 \oplus P1$$

This is similar to parity checker in Section 4.8. Note that, in addition to data bits, we have also included the corresponding parity bit to the input of exclusive-OR gate tree. Proceeding similarly for other positions, we find that except for P4 all other parity checks fail. Note that, even a single failure detects an error. However, to correct the error, we use the output of last column 1011 (in the order P8 P4 P2 P1) and find its decimal equivalent which is 11. So the data of location 11, which is D11 needs to be corrected.

	P1	P2	D3	P4	D5	D6	D7	P8	D9	D10	D11	Parity check	Parity bit
Received data word	1	0	0	0	1	1	0	0	1	0	0		
P1	1		0		1		0		1		0	Fail	1
P2		0	0			1	0			0	0	Fail	1
P4				0	1	1	0					Pass	0
P8								0	1	0	0	Fail	1

Fig. 5.6 **Error detection and correction**

Note that, this method detects error in more than one position unlike the first method but overhead is more. In simple parity method, we add 1 additional bit for 7-bit data whreas it is 4 in this method. Also note, by further increasing this overhead, error in more than one position can also be corrected. However, more than one-bit error is unlikely for memory read. With overhead for one-bit correction, if there occurs error in more than one-bit positions, then the data needs to be read once again from the memory.

SELF-TEST

18. Can parity code detect even number of errors?
19. What is the full form of CRC?
20. What is the advantage of Hamming code?
21. What is error detection-correction overhead?

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Add two gray coded numbers 0100 and 0111 and express the result in gray code.

Solution Since gray coded numbers are not suitable for arithmetic operations, we have to convert the numbers to some other form, perform the addition and then convert the result to gray code. We first show how it can be done through lookup tables. It would require storage of large lookup tables, if the numbers are large in value. Next, we show the converter-based approach which only needs the implementation of conversion equations.

In Method-1, we take help of first two columns of Table 5.9 and convert these two numbers to decimal, add the decimal numbers and then again use the table to get corresponding gray coded number. This is shown in Fig. 5.7a.

In Method-2, we take help of last two columns of Table 5.9 and convert these two numbers to binary, perform binary addition and then again use the table to get corresponding gray coded number. This is shown in Fig. 5.7b.

In Method-3, we take help of gray to binary conversion relation shown in Fig. 5.2b and convert these two numbers to binary, perform binary addition and then use binary to gray conversion relation shown in Fig. 5.2a to get corresponding gray coded number. This is shown in Fig. 5.7c.

Using Table 5.9

Gray	Decimal
0100	7
0111	+5
	12

Decimal	Gray
12	1010

(a) Addition using Method-1

Using Table 5.9

Gray	Binary
0100	0111
0111	+0101
	1100

Binary	Gray
1100	1010

(b) Addition using Method-2

From Fig. 5.2b

Gray to Binary Conversion:	$B_3 = G_3$	$B_2 = B_3 \oplus G_2$	$B_1 = B_2 \oplus G_1$	$B_0 = B_1 \oplus G_0$	Binary
For $G_3G_2G_1G_0 = 0100$:	$B_3 = 0$	$B_2 = 0 \oplus = 1$	$B_1 = 1 \oplus 0 = 1$	$B_0 = 1 \oplus 0 = 1$	0111
For $G_3G_2G_1G_0 = 0110$:	$B_3 = 0$	$B_2 = 0 \oplus 1 = 1$	$B_1 = 1 \oplus 1 = 0$	$B_0 = 0 \oplus 1 = 1$	+0101
					1100

From Fig. 5.2a

Binary to Gray Conversion:	$G_3 = B_3$	$G_2 = B_3 \oplus B_2$	$G_1 = B_2 \oplus B_1$	$G_0 = B_1 \oplus B_0$	Gray
For $B_3\,B_2\,B_1\,B_0 = 1100$:	$B_3 = 1$	$B_2 = 1 \oplus = 0$	$B_1 = 1 \oplus 0 = 1$	$B_0 = 0 \oplus 0 = 0$	1010

(c) Addition using Method-3

Fig. 5.7

SUMMARY

To convert from binary to decimal numbers, add the weight of each bit position (1, 2, 4, 8, ...) when there is a 1 in that position. With fractions, the binary weights are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$, ..., and so on. To convert from decimal to binary, use double dabble for integers and the multiply-by-2 method for fractions.

The base of a number system equals the number of digits it uses. The decimal number system has a base of 10, while the binary number system has a base of 2. The octal number system has a base of 8. A useful model for counting is the octal odometer. When a display wheel turns from 7 back to 0, it sends a carry to the next-higher wheel.

Hexadecimal numbers have a base of 16. The model for counting is the hexadecimal odometer, whose wheels reset and carry beyond F. Hexadecimal numbers are easy to convert mentally into their binary equivalents. For this reason, people prefer using hexadecimal numbers because they are much shorter than the corresponding binary numbers.

The ASCII code is an alphanumeric code widely used for transferring data into and out of a computer. This 7-bit code is used to represent alphabet letters, numbers, and other symbols. The excess-3 code and the Gray code are two other codes that are used.

A logic pulser can temporarily change the state of a node under test. If the original state is low, the logic pulser drives the node briefly into the high state. If the state is high, the logic pulser drives the node briefly

into the low state. The output impedance of a logic pulser is so low that it can drive almost any normal node in a logic circuit. When a node is shorted to ground or to the supply voltage, the logic pulser is unable to change the voltage level; this is a confirmation of the shorted condition.

Parity code, Checksum code, and CRC code have been discussed for error detection code and Hamming code for error detection and correction. These techniques are used to ensure that data is correct and has not been corrupted, either by hardware failures or by noise occurring during transmission or a data read operation from memory.

GLOSSARY

- ***base*** The number of digits or basic symbols in a number system. The decimal system has a base of 10 because it uses 10 digits. Binary has a base of 2, octal a base of 8, and hexadecimal a base of 16.
- ***binary*** Refers to a number system with a base of 2, that is, containing two digits.
- ***bit*** An abbreviated form of binary digit. Instead of saying that 10110 has five binary digits, we can say that it has 5 bits.
- ***byte*** A binary number with 8 bits.
- ***checksum code*** A error detection code generating sum of a block of data.
- ***CRC code*** Cyclic Redundancy Code is a polynomial key based error detection code.
- ***digit*** A basic symbol used in a number system. The decimal system has 10 digits, 0 through 9.
- ***error detection and correction*** A method of detection of error in a group of bits and correction of the same.
- ***hamming code*** A parity bit based error detection and correction code.
- ***hexadecimal*** Refers to number system with a base of 16. The hexadecimal system has digits 0 through 9, followed by A through F.
- ***logic pulser*** A troubleshooting device that generates brief voltage pulses. The typical logic pulser has a push-button switch that produces a single pulse for each closure. More advanced logic pulsers can generate a pulse train with a specified number of pulses.
- ***nibble*** An binary number with 4 bits.
- ***octal*** Refers to a number system with a base of 8, that is, one that uses 8 digits. Normally, these are 0, 1, 2, 3, 4, 5, 6, and 7.
- ***parity code*** An error detection code using one additional parity bit.
- ***weight*** Refers to the decimal value of each digit position of a number. For decimal numbers, the weights are 1, 10, 100, 1000, …, working from the decimal point to the left. For binary numbers the weights are 1, 2, 4, 8, … to the left of the binary point. With octal numbers, the weights become 1, 8, 64, … to the left of the octal point.

PROBLEMS

Section 5.1

5.1 What is the binary number that follows 01101111?

5.2 How many bits are there in 2 bytes?

5.3 How many nibbles are there in each of these:
a. 1001
b. 11110000

c. 110011110000
d. 1111000011001001

Section 5.2

5.4 Give the decimal equivalents for each of the following binary numbers:
a. 110101 b. 11001.011

5.5 Convert the following binary numbers to their decimal equivalents:
a. 1011 1100 b. 1111 1111

5.6 What is the decimal equivalent of 1000 1100 1011 0011?

5.7 A computer has 128K of memory. How many bytes does this represent?

Section 5.3

5.8 Convert the following decimal numbers to binary numbers: 24, 65, and 106.

5.9 What binary number does decimal 268 stand for?

5.10 Convert decimal 108.364 to a binary number.

5.11 Calculate the binary equivalent for 5280.

Section 5.4

5.12 Convert the following octal numbers to decimal equivalents:
a. 65 b. 216
c. 4073

5.13 What is the decimal equivalent of octal 325.736?

5.14 Convert these decimal numbers to octal numbers:
a. 4096 b. 65535

5.15 What is the octal equivalent of decimal 324.987?

5.16 Convert the following octal numbers to binary numbers: 34, 567, 4673.

5.17 Convert the following binary numbers to octal numbers:
a. 10101111
b. 1101.0110111
c. 1010011.101101

Section 5.5

5.18 What are the hexadecimal numbers that follow each of these:
a. ABCD b. 7F3F
c. BEEF

5.19 Convert the following hexadecimal numbers to binary numbers:
a. E5 b. B4D
c. 7AF4

5.20 Convert these binary numbers into hexadecimal numbers:
a. 1000 1100 b. 0011 0111
c. 1111 0101 0110

5.21 Convert hexadecimal 2F59 to its decimal equivalent.

5.22 What is the hexadecimal equivalent of decimal 62359?

5.23 Give the value of $Y_3Y_2Y_1Y_0$ in Fig. 5.8 for each of these:
a. All switches are open
b. Switch 4 is closed
c. Switch A is closed
d. Switch F is closed

5.24 A computer has the following hexadecimal contents stored at the addresses shown:

Address	Hexadecimal contents
2000	D5
2001	AA
2002	96
2003	DE
2004	AA
2005	EB

What are the binary contents at each address?

Section 5.6

5.25 Give the ASCII code for each of these:
a. 7 b. W
c. f d. y

5.26 Suppose that you type LIST with an ASCII keyboard. What is the binary output as you strike each letter.

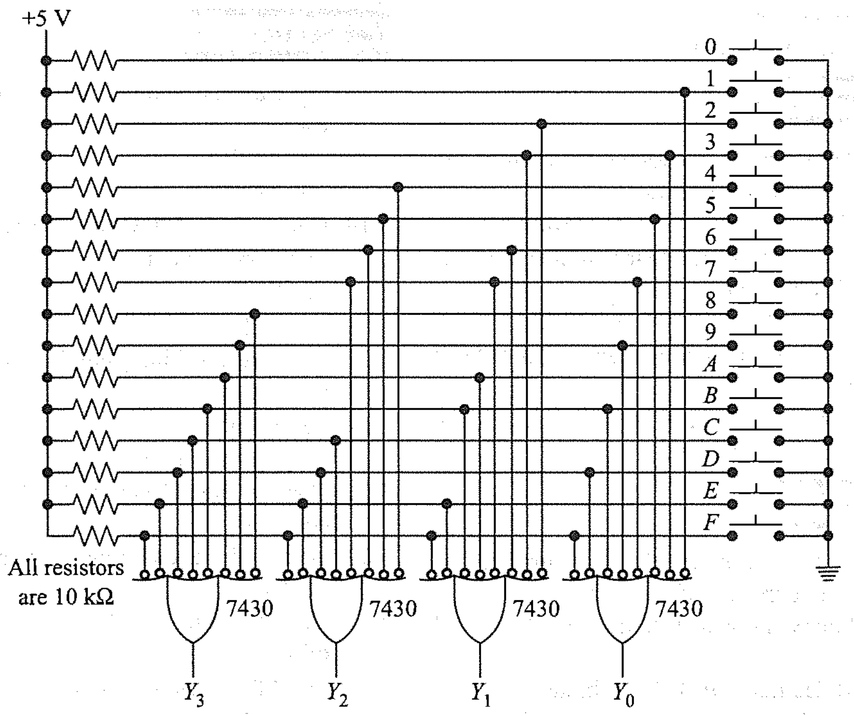


Fig. 5.8

5.27 In Example 5.15, a computer sends the word HELLO to another computer. The characters are coded in ASCII with an odd-parity bit. Here is how the word is stored in the memory of the receiving computer:

Address	Alphanumeric	Hexadecimal contents
2000	H	C8
2001	E	45
2002	L	4C
2003	L	4C
2004	O	4F

The transmitting computer then sends the word GOODBYE. Show how this word is stored in the receiving computer. Use a starting address of 2000 and include a parity bit.

Section 5.7

5.28 Express decimal 5280 in excess-3 code.

5.29 Here is an excess-3 number:

0110 1001 1100 0111

What is the decimal equivalent?

Section 5.8

5.30 What is the Gray code for decimal 8?

5.31 Convert Gray number 1110 to its BCD equivalent.

Section 5.9

5.32 Figure 5.9 shows the decimal-to-BCD encoder discussed in Sec. 4.6. Answer the following questions:

a. If all switches are open and the logic pulser is inactive, what voltage level does the logic probe indicate?
b. If switch 6 is closed and the logic pulser is inactive, what does the logic probe indicate?

c. If all switches are open and the logic pulser is activated, what does the logic probe do?

5.33 The push-button switch of the logic pulser shown in Fig. 5.9 is pressed. Suppose that the logic probe is initially dark and remains dark. Indicate which of the following are possible sources of trouble:

a. 74147 defective
b. Pin 9 shorted to ground
c. Pin 9 shorted to +5 V
d. Pin 10 shorted to ground

5.34 The instruction register shown in Fig. 5.10 on the next page is a logic circuit that stores a 16-bit number, $I_{15} \cdots I_0$. The first 4 bits, $I_{15} \cdots I_{12}$, are decoded by a 4 to 16-line decoder. Determine whether the logic probe indicates low, high, or blink for each of these conditions:

a. $I_{15} \cdots I_{12} = 0000$ and logic pulser inactive
b. $I_{15} \cdots I_{12} = 1000$ and logic pulser inactive
c. $I_{15} \cdots I_{12} = 1000$ and logic pulser active
d. $I_{15} \cdots I_{12} = 1111$ and logic pulser active

5.35 The logic pulser and logic probe shown in Fig. 5.10 are used to check the pins of the 7404 for stuck states. Suppose pin 8 is stuck in the high state. Indicate which of the following are possible sources of trouble:

a. No supply voltage anywhere in circuit
b. Pin 1 of IC2 shorted to ground
c. Pin 2 of IC4 shorted to the supply voltage
d. Pin 3 of IC5 shorted to ground
e. Pin 4 of IC8 shorted to the supply voltage

Section 5.10

5.36 Find Hamming code of data 11001.
5.37 Find Hamming code of data 1000111.
5.38 If an error occurs in the 3rd data bit, how will it be corrected for data of problem 5.37?
5.39 How many parity bits are needed to Hamming code (a) 16-bit data and (b) 24-bit data.

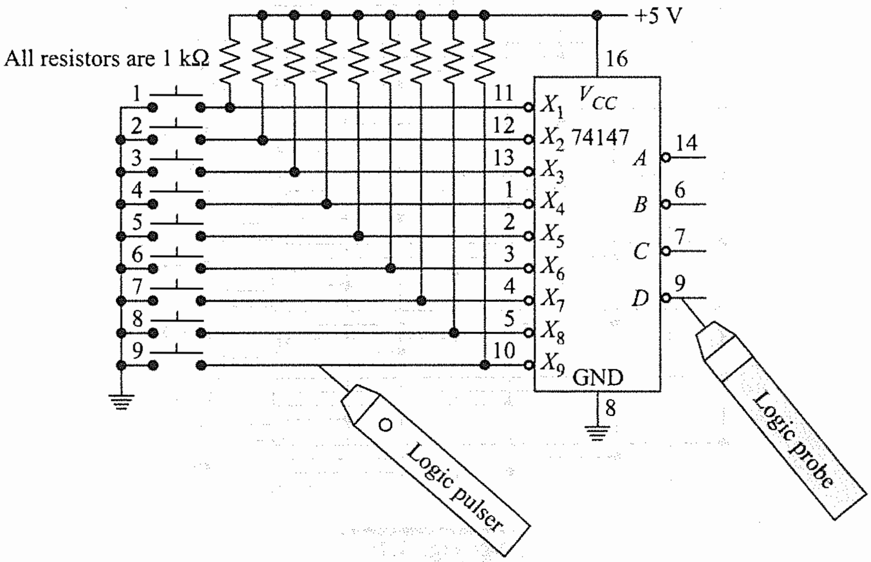


Fig. 5.9

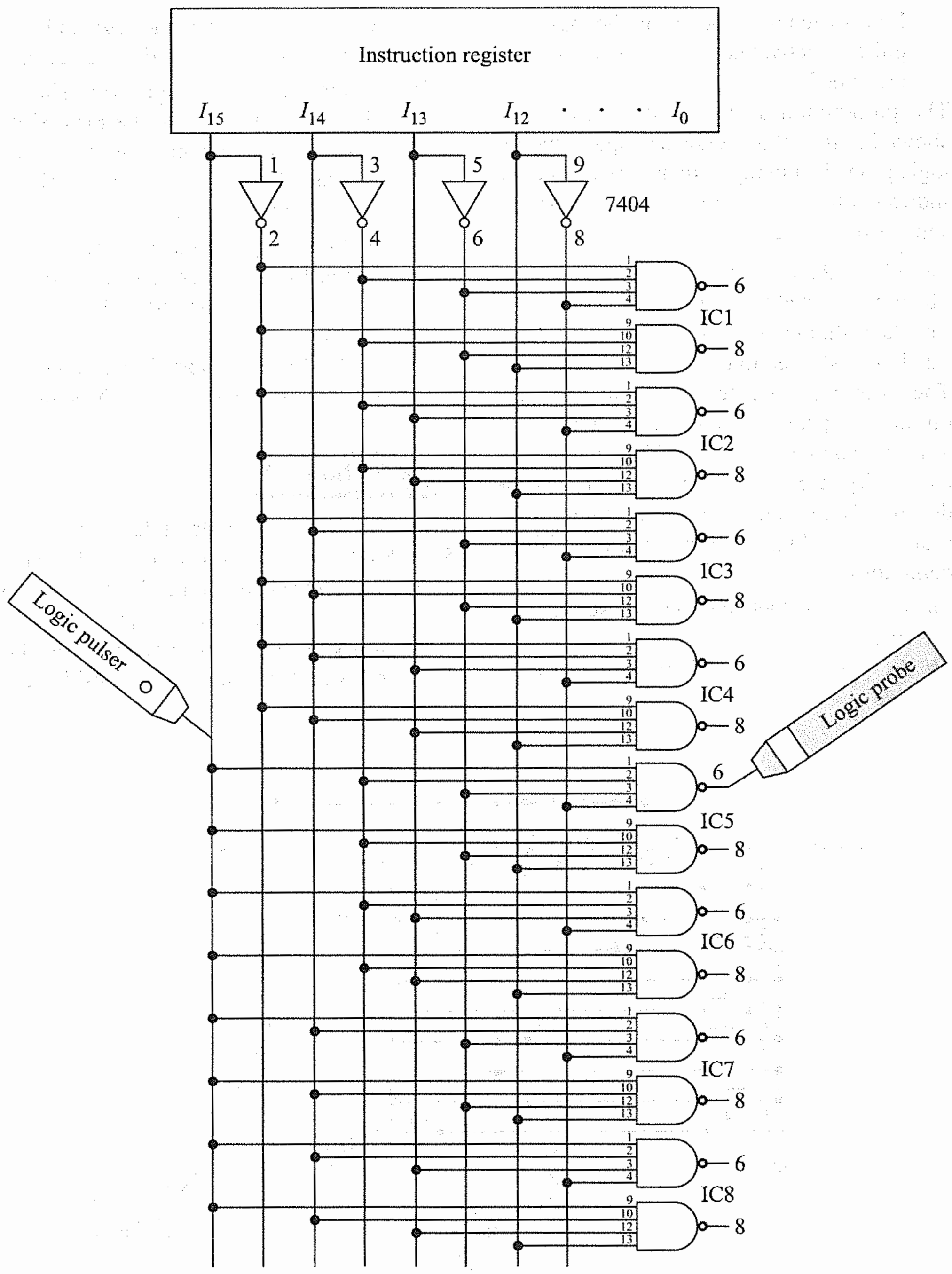

Instruction register
I_{15}
I_{14}
I_{13}
I_{12}
I_0
7404
IC1
IC2
IC3
IC4
IC5
IC6
IC7
IC8
Logic pulser
Logic probe

Fig. 5.10

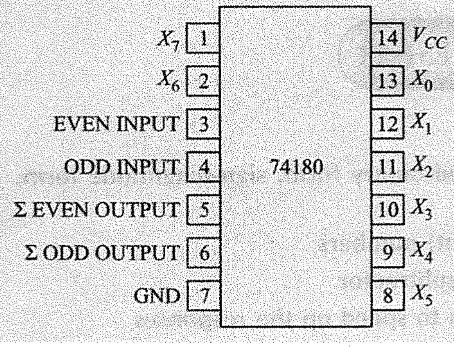
LABORATORY EXPERIMENT

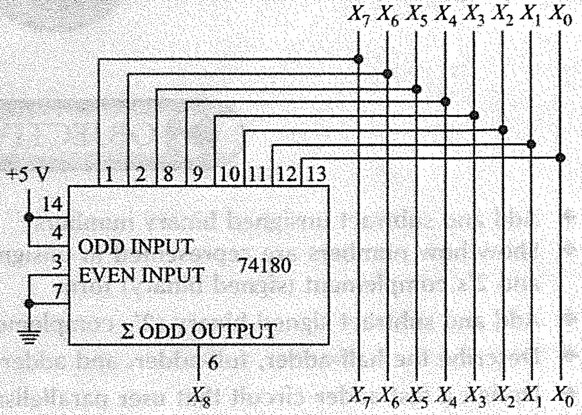
AIM: The aim of this experiment is to generate and check parity code.

Theory: The parity code is obtained by exclusive-OR of data bits. The even parity makes the number of 1s even after the addition of the parity code while odd parity maintains it as odd. The parity in the bit streams, even or odd, is also checked by exclusive-OR of incoming data. Thus the same circuit can be used both for parity generation and checking after appropriate configuration.

Apparatus: 5 VDC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of IC 74180, the 8-bit parity generator/checker. Connect it as shown to use it as parity generator. Submit 5 different numbers and check the parity of the coded data, i.e. data plus parity bit. Configure it in such a way that it becomes a parity checker and then check the parity of these 5 numbers. IC 7486 is a quad 2-input exclusive-OR gate with pin configuration similar to 7400 or 7408. Use this to generate parity and compare the result with 74180. Finally, find how 7- and 9-bit long data can be parity coded.





X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
+5 V
1 2 8 9 10 11 12 13
14
4
3
7
ODD INPUT
EVEN INPUT
74180
Σ ODD OUTPUT
6
X_8
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0

Answers to Self-tests

1. 1101
2. 9
3. Four
4. 18
5. 100011
6. $2^9 = 512$
7. *Double dabble* is a method for converting decimal numbers to binary numbers.
8. 4,095
9. 1111 1111
10. 0, 1, 2, 3, 4, 5, 6, and 7
11. 7 (octal) and 7 (decimal)
12. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F
13. 0011 1100
14. 60
15. ASCII stands for American Standard Code for Information Interchange, a code used to represent alphanumeric information.
16. @
17. 010 0101
18. No
19. Cycle Redundancy Code
20. It can detect as well as correct one-bit error.
21. Additional bits to be included with data bits for this purpose.

Arithmetic Circuits

6

OBJECTIVES

- Add and subtract unsigned binary numbers
- Show how numbers are represented in unsigned binary form, sign-magnitude form, and 2's complement (signed binary) form
- Add and subtract signed binary (2's complement) numbers
- Describe the half-adder, full-adder, and adder-subtractor
- Design a fast adder circuit that user parallelism to speed up the responses
- Describe how an Arithmetic Logic Unit can be operated
- Explain the means by which multiplication and division are performed on typical 8-bit microprocessors

Circuits that can perform binary addition and subtraction are constructed by combining logic gates. These circuits are used in the design of the arithmetic logic unit (ALU). The electronic circuits are capable of very fast switching action, and thus an ALU can operate at high clock rates. For instance, the addition of two numbers can be accomplished in a matter of nanoseconds! This chapter begins with binary addition and subtraction, then presents two different methods for representing negative numbers. You will see how an exclusive OR gate is used to construct a half-adder and a full-adder. You will see how to construct an 8-bit adder-subtracter using a popular IC. A technique to design a fast adder is discussed in detail followed by discussion on a multifunctional device called Arithmetic Logic Unit or ALU. Finally, an outline to perform binary multiplication and division is also presented.

6.1 BINARY ADDITION

Numbers represent physical quantities. Table 6.1 shows the decimal digits and the corresponding amount of pebbles. Digit 2 stands for two pebbles (●●), 5 for five pebbles (●●●●●), and so on. Addition represents the combining of physical quantities. For instance:

$$2 + 3 = 5$$

symbolizes the combining of two pebbles with three pebbles to obtain a total of five pebbles. Symbolically, this is expressed

●● + ●●● = ●●●●●

Table 6.1 The Decimal Digits

Pebbles	*Symbol*
None	0
●	1
●●	2
●●●	3
●●●●	4
●●●●●	5
●●●●●●	6
●●●●●●●	7
●●●●●●●●	8
●●●●●●●●●	9

Four Cases to Remember

Computer circuits don't process decimal numbers; they process binary numbers. Before you can understand how a computer performs arithmetic, you have to learn how to add binary numbers. Binary addition is the key to binary subtraction, multiplication, and division. So, let's begin with the four most basic cases of binary addition:

$$0 + 0 = 0 \qquad (6.1)$$

$$0 + 1 = 1 \qquad (6.2)$$

$$1 + 0 = 1 \qquad (6.3)$$

$$1 + 1 = 10 \qquad (6.4)$$

Equation (6.1) is obvious; so are Eqs. (6.2) and (6.3) because they are identical to decimal addition. The fourth case, however, may bother you. If so, you don't understand what Eq. (6.4) represents in the physical world. Equation (6.4) represents the combining of one pebble and one pebble to obtain a total of two pebbles:

● + ● = ●●

Since binary 10 stands for ●●, the binary equation

$$1 + 1 = 10$$

makes perfect sense. From now on, remember that numbers, whether binary, decimal, octal, or hexadecimal are codes for physical amounts. If you're in doubt about the meaning of a numerical equation, convert the numbers to pebbles to see if the two sides of the equation are equal.

Subscripts

The foregoing discussion brings up the idea of *subscripts*. Since we already have discussed four kinds of numbers (decimal, binary, octal, and hexadecimal), we have four different ways to code physical quantities. How do we know which code is being used? In other words, how can we tell when 10 is a decimal, binary, octal, or hexadecimal number?

Most of the time, it's clear from the discussion which kind of numbers are involved. For instance, if we have been discussing nothing but binary numbers for page after page, you can count on the next 10 being binary 10, which represents •• in the physical world. On the other hand, if a discussion uses more than one type of number, it may be helpful to use subscripts for the base as follows:

$$\begin{aligned} 2 &\rightarrow \text{binary} \\ 8 &\rightarrow \text{octal} \\ 10 &\rightarrow \text{decimal} \\ 16 &\rightarrow \text{hexadecimal} \end{aligned}$$

For instance, 11_2 represents binary 11, 23_8 stands for octal 23, 45_{10} for decimal 45, and $F4_{16}$ for hexadecimal F4. With the subscripts in mind, the following equations should make perfect sense:

$$\begin{aligned} 1_2 + 1_2 &= 10_2 \\ 7_8 + 1_8 &= 10_8 \\ 9_{10} + 1_{10} &= 10_{10} \\ F_{16} + 1_{16} &= 10_{16} \end{aligned}$$

Larger Binary Numbers

Column-by-column addition applies to binary as well as decimal numbers. For example, suppose you have this problem in binary addition:

$$\begin{array}{r} 11100 \\ +11010 \\ \hline ? \end{array}$$

Start by adding the least-significant column to get

$$\begin{array}{r} 11100 \\ +11010 \\ \hline 0 \end{array}$$

Next, add the bits in the second column as follows:

$$\begin{array}{r} 11100 \\ +11010 \\ \hline 10 \end{array}$$

The third column gives

$$\begin{array}{r} 11100 \\ +11010 \\ \hline 110 \end{array}$$

The fourth column results in

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ & +11010 \\ \hline & 0110 \end{array}$$

Notice the carry into the final column; this carry occurs because 1 + 1 = 10. As in decimal addition, you write down the 0 and carry the 1 to the next-higher column.

Finally, the last column gives

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ & +\,11010 \\ \hline & 110110 \end{array}$$

In the last column, 1 + 1 + 1 = 10 + 1 = 11.

8-Bit Arithmetic

That's all there is to binary addition. If you can remember the four basic rules, you can add column by column to find the sum of two binary numbers, regardless of how long they may be. In first-generation microcomputers (Apple II, TRS-80, etc.), addition is done on two 8-bit numbers such as

$$\begin{array}{r} A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\,B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline ? \end{array}$$

The most-significant bit (MSB) of each number is on the left, and the least-significant bit (LSB) is on the right. For the first number, A_7 is the MSB and A_0 is the LSB. For the second number, B_7 is the MSB and B_0 is the LSB. Try to remember the abbreviations MSB and LSB because they are used frequently in computer discussions.

Example 6.1 Add these 8-bit numbers: 0101 0111 and 0011 0101. Then, show the same numbers in hexadecimal notation.

Solution This is the problem:

$$\begin{array}{r} 0101 \quad 0111 \\ +0011 \quad 0101 \\ \hline ? \end{array}$$

If you add the bits in each column as previously discussed, you will obtain

$$\begin{array}{r} 0101 \quad 0111 \\ +0011 \quad 0101 \\ \hline 1000 \quad 1100 \end{array}$$

Many people prefer hexadecimal notation because it's a faster code to work with. Expressed in hexadecimal numbers, the foregoing addition is

$$\begin{array}{r} 57 \\ +35 \\ \hline 8C \end{array}$$

Often, the letter H is used to signify hexadecimal numbers, so the foregoing addition may be written as

$$\begin{array}{r} 57H \\ +35H \\ \hline 8CH \end{array}$$

Example 6.2 Add these 16-bit numbers: 0000 1111 1010 1100 and 0011 1000 0111 1111. Show the corresponding hexadecimal and decimal numbers.

Solution Start at the right and add the bits, column by column:

Binary	Hexadecimal	Decimal
0000 1111 1010 1100	0FACH	4,012
+ 0011 1000 0111 1111	+ 387FH	+ 14,463
0100 1000 0010 1011	482BH	18,475

(*Note:* Remember Appendix 1; it takes most of the work out of conversions between number systems.)

Example 6.3 Repeat Example 6.2, showing how a first-generation microcomputer does the addition.

Solution First-generation microcomputers like the Apple II have an 8-bit *microprocessor* (a digital IC that performs binary arithmetic on 8-bit numbers). To add 16-bit numbers, a first-generation microcomputer adds the lower 8 bits in one operation and then the upper 8 bits in another operation.

Here is how it works for numbers of the preceding example. The original problem is

```
   Upper bytes      Lower bytes
        ↓                ↓
   0000  1111       1010  1100
 + 0011  1000       0111  1111
 ------------------------------
                ?
```

The microcomputer starts by adding the lower bytes:

```
   1010 1100
 + 0111 1111
 -----------
  10010 1011
```

Notice the carry into the final column. The microcomputer will store the lower byte (0010 1011). Then, it will do another addition of the upper bytes, plus the carry, as follows:

```
            1 ← Carry
   0000  1111
 + 0011  1000
 ------------
   0100  1000
```

The microcomputer then stores the upper byte. To output the total answer, the microcomputer pulls the upper and lower sums out of its memory to get

0100 1000 0010 1011

which is equivalent to 482BH or 18,475, the same as we found in the preceding example.

SELF-TEST

1. Write the four rules for binary addition.
2. What kind of number is 179FH?
3. What is the meaning of 111_2? Of 111_{10}?

6.2 BINARY SUBTRACTION

Let's begin with four basic cases of binary subtraction:

$$0 - 0 = 0 \quad (6.5)$$
$$1 - 0 = 1 \quad (6.6)$$
$$1 - 1 = 0 \quad (6.7)$$
$$10 - 1 = 1 \quad (6.8)$$

Equations (6.5) to (6.7) are easy to understand because they are identical to decimal subtraction. The fourth case will disturb you if you have lost sight of what it really means. Back in the physical world, Eq. (6.4) represents

●● – ● = ●

Two pebbles minus one pebble equals one pebble.

For larger binary numbers, subtract column by column, the same as you do with decimal numbers. This means that you sometimes have to borrow from the next-higher column. Here is an example:

$$\begin{array}{r} 1101 \\ -1010 \\ \hline ? \end{array}$$

Subtract the L'SBs to get

$$\begin{array}{r} 1101 \\ -1010 \\ \hline 1 \end{array}$$

To subtract the bits of the second column, borrow from the next-higher column to obtain

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 1 \end{array}$$

In the second column from the right, subtract as follows: 10 – 1 = 1, to get

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 11 \end{array}$$

Then subtract the remaining columns:

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 0011 \end{array}$$

After you get used to it, binary subtraction is no more difficult than decimal subtraction. In fact, it's easier because there are only four basic cases to remember.

Example 6.4 Show the binary subtraction of 125_{10} from 200_{10}.

Solution First, use Appendix 1 to convert the numbers as follows:

$$200 \rightarrow \text{C8H} \rightarrow 1100\ \ 1000$$
$$125 \rightarrow \text{7DH} \rightarrow 0111\ \ 1101$$

So, here is the problem:

$$\begin{array}{r} 1100\ \ 1000 \\ -0111\ \ 1101 \\ \hline ? \end{array}$$

Column-by-column subtraction gives:

$$\begin{array}{r} 1100\ \ 1000 \\ -0111\ \ 1101 \\ \hline 0100\ \ 1011 \end{array}$$

In hexadecimal notation, the foregoing appears as

$$\begin{array}{r} \text{C8H} \\ -\text{7DH} \\ \hline \text{4BH} \end{array}$$

SELF-TEST

4. Write the four rules for binary subtraction.

6.3 UNSIGNED BINARY NUMBERS

In some applications, all data is either positive or negative. When this happens, you can forget about + and – signs, and concentrate on the *magnitude* (absolute value) of numbers. For instance, the smallest 8-bit number is 0000 0000, and the largest is 1111 1111. Therefore, the total range of 8-bit numbers is

0000 0000 (00H)

to

1111 1111 (FFH)

This is equivalent to a decimal 0 to 255. As you can see, we are not including + or – signs with these decimal numbers.

With 16-bit numbers, the total range is

0000 0000 0000 0000 (0000H)

to

1111 1111 1111 1111 (FFFFH)

which represents the magnitude of all decimal numbers from 0 to 65,535.

Data of the foregoing type is called *unsigned binary* because all of the bits in a binary number are used to represent the magnitude of the corresponding decimal number. You can add and subtract unsigned binary numbers, provided certain conditions are satisfied. The following examples will tell you more about unsigned binary numbers.

Limits

First-generation microcomputers can process only 8 bits at a time. For this reason, there are certain restrictions you should be aware of. With 8-bit unsigned arithmetic, all magnitudes must be between 0 and 255. Therefore, each number being added or subtracted must be between 0 and 255. Also, the answer must fall in the range of 0 to 255. If any magnitudes are greater than 255, you should use 16-bit arithmetic, which means operating on the lower 8 bits first, then on the upper 8 bits (see Example 6.3).

Overflow

In 8-bit arithmetic, addition of two unsigned numbers whose sum is greater than 255 causes an *overflow*, a carry into the ninth column. Most microprocessors have a logic circuit called a *carry flag*; this circuit detects a carry into the ninth column and warns you that the 8-bit answer is invalid (see Example 6.7).

Example 6.5 Show how to add 150_{10} and 85_{10} with unsigned 8-bit numbers.

Solution With Appendix 1, we obtain

$$150 \rightarrow 96\text{H} \rightarrow 1001\ 0110$$
$$85 \rightarrow 55\text{H} \rightarrow 0101\ 0101$$

Next, we can add these unsigned numbers to get

```
  1001 0110        96H
+0101 0101       +55H
-------------------------
  1110 1011        EBH
```

Again, Appendix 1 gives

$$1110\ 1011 \rightarrow \text{EBH} \rightarrow 235$$

Example 6.6 Show how to subtract 85_{10} from 150_{10} with unsigned 8-bit numbers.

Solution Use the same binary numbers as in the preceding example, but subtract to get

```
  1001 0110        96H
-0101 0101       -55H
-------------------------
  0100 0001        41H
```

Again, Appendix 1 gives

$$0100\ 0001 \rightarrow 41\text{H} \rightarrow 65$$

Example 6.7 In the two preceding examples, everything was well behaved because both decimal answers were between 0 and 255. Now, you will see how an overflow can occur to produce an invalid 8-bit answer.

Show the addition of 175_{10} and 118_{10} using unsigned 8-bit numbers.

Solution

$$\begin{array}{r} 175 \\ +118 \\ \hline 293 \end{array}$$

The answer is greater than 255. Here is what happens when we try to add 8-bit numbers:

Appendix 1 gives

$$175 \rightarrow \text{BFH} \rightarrow 1010\ \ 1111$$
$$118 \rightarrow 76\text{H} \rightarrow 0111\ \ 0110$$

An 8-bit microprocessor adds like this:

$$\begin{array}{rr} & 1010\ \ 1111 & \quad \text{AFH} \\ & +0111\ \ 0110 & \quad +\ 76\text{H} \\ \hline \text{Overflow} \rightarrow & 1\ 0010\ \ 0101 & \quad 125\text{H} \end{array}$$

With 8-bit arithmetic, only the lower 8 bits are used. Appendix 1 gives

$$0010\ \ 0101 \rightarrow 25\text{H} \rightarrow 37$$

As you see, the 8-bit answer is wrong. It is true that if you take the overflow into account, the answer is valid, but then you no longer are using 8-bit arithmetic. The point is that somebody (the programmer) has to worry about the possibility of an overflow and must take steps to correct the final answer when an overflow occurs. If you study assembly-language programming, you will learn more about overflows and what to do about them.

In summary, 8-bit arithmetic circuits can process decimal numbers between 0 and 255 only. If there is any chance of an overflow during an addition, the programmer has to write instructions that look at the carry flag and use 16-bit arithmetic to obtain the final answer. This means operating on the lower 8 bits, and then the upper 8 bits and the overflow (as done in Example 6.3).

SELF-TEST

5. What is the carry flag in a microprocessor?
6. What is the largest decimal number that can be represented with an 8-bit unsigned binary number?

6.4 SIGN-MAGNITUDE NUMBERS

What do we do when the data has positive and negative values? The answer is important because it determines how complicated the arithmetic circuits must be. The negative decimal numbers are –1, –2, –3, and so on. The magnitude of these numbers is 1, 2, 3, and so forth. One way to code these as binary numbers is to convert the magnitude to its binary equivalent and prefix the sign. With this approach, the sequence –1, –2, and –3 becomes –001, –010, and –011. Since everything has to be coded as strings of 0s and 1s, the + and – signs also have to be represented in binary form. For reasons given soon, 0 is used for the + sign and 1 for the – sign. Therefore, –001, –010, and –011 are coded as 1001, 1010, and 1011.

The foregoing numbers contain a sign bit followed by magnitude bits. Numbers in this form are called *sign-magnitude numbers*. For larger decimal numbers, you need more than 4 bits. But the idea is still the same: the MSB always represents the sign, and the remaining bits always stand for the magnitude. Here are some examples of converting sign-magnitude numbers:

$$+7 \rightarrow 0000\ \ 0111$$
$$-16 \rightarrow 1001\ \ 0000$$
$$+25 \rightarrow 0000\ \ 0000\ \ 0001\ \ 1001$$
$$-128 \rightarrow 1000\ \ 0000\ \ 1000\ \ 0000$$

Range of Sign-Magnitude Numbers

As you know, the unsigned 8-bit numbers cover the decimal range of 0 to 255. When you use sign-magnitude numbers, you reduce the largest magnitude from 255 to 127 because you need to represent both positive and negative quantities. For instance, the negative numbers are

1000 0001 (–1)

to

1111 1111 (–127)

The positive numbers are

0000 0001 (+1)

to

0111 1111 (+127)

The largest magnitude is 127, approximately half of what is for unsigned binary numbers. As long as your input data is in the range of –127 to +127, you can use 8-bit arithmetic. The programmer still must check sums for an overflow because all 8-bit answers are between –127 and +127.

If the data has magnitudes greater than 127, then 16-bit arithmetic may work. With 16-bit numbers, the negative numbers are from

1000 0000 0000 0001 (–1)

to

1111 1111 1111 1111 (–32, 767)

and the positive numbers are from

0000 0000 0000 0001 (+1)

to

0111 1111 1111 1111 (+32, 767)

Again, you can see that the largest magnitude is approximately half that of unsigned binary numbers. Unless you actually need + and – signs to represent your data, you are better off using unsigned binary.

The main advantage of sign-magnitude numbers is their simplicity. Negative numbers are identical to positive numbers, except for the sign bit. Because of this, you can easily find the magnitude by deleting the sign bit and converting the remaining bits to their decimal equivalents. Unfortunately, sign-magnitude numbers have limited use because they require complicated arithmetic circuits. If you don't have to add or subtract the data, sign-magnitude numbers are acceptable. For instance, sign-magnitude numbers are often used in *analog-to-digital* (A/D) conversions (explained in a latter chapter).

SELF-TEST

7. What is the decimal number range that can be represented with an 8-bit sign-magnitude binary number?
8. In sign-magnitude form, what is the decimal value of 1000 1101? Of 0000 1101?

6.5 2'S COMPLEMENT REPRESENTATION

There is a rather unusual number system that leads to the simplest logic circuits for performing arithmetic. Known as *2's complement representation,* this system dominates microcomputer architecture and programming.

1's Complement

The 1's complement of a binary number is the number that results when we complement each bit. Figure 6.1 shows how to produce the 1's complement with logic circuits. Since each bit drives an inverter, the 4-bit output is the 1's complement of the 4-bit input. For instance, if the input is

$$X_3X_2X_1X_0 = 1000$$

the 1's complement is

$$\overline{X}_3\overline{X}_2\overline{X}_1\overline{X}_0 = 0111$$

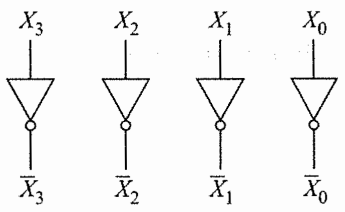


Fig. 6.1 **Inverters produce the 1's complement.**

The same principle applies to binary numbers of any length: complement each bit to obtain the 1's complement. More examples of 1's complements are

$$1010 \rightarrow 0101$$
$$1110\ \ 1100 \rightarrow 0001\ \ 0011$$
$$0011\ \ 1111\ \ 0000\ \ 0110 \rightarrow 1100\ \ 0000\ \ 1111\ \ 1001$$

2's Complement

The 2's complement is the binary number that results when we add 1 to the 1's complement. As a formula:

$$2\text{'s complement} = 1\text{'s complement} + 1$$

For instance, to find the 2's complement of 1011, proceed like this:

$$1011 \rightarrow 0100 \qquad (1\text{'s complement})$$
$$0100 + 1 = 0101 \qquad (2\text{'s complement})$$

Instead of adding 1, you can visualize the next reading on a binary odometer. So, after obtaining the 1's complement 0100, ask yourself what comes next on a binary odometer. The answer is 0101.

Here are more examples of the 2′s complements:

Number →	1′s complement →	2′s complement
1110 1100 →	0001 0011 →	0001 0100
1000 0001 →	0111 1110 →	0111 1111
0011 0110 →	1100 1001 →	1100 1010

Back to the Odometer

The binary odometer is a marvelous way to understand 2′s complement representation. By examining the numbers of a binary odometer, we can see how the typical microcomputer represents positive and negative numbers. With a binary odometer, all bits eventually reset to 0s. Some readings before and after a complete reset look like this:

1000	(–8)
1001	(–7)
1010	(–6)
1011	(–5)
1100	(–4)
1101	(–3)
1110	(–2)
1111	(–1)
0000	(0)
0001	(+1)
0010	(+2)
0011	(+3)
0100	(+4)
0101	(+5)
0110	(+6)
0111	(+7)

Binary 1101 is the reading 3 miles before reset, 1110 occurs 2 miles before reset, and 1111 indicates 1 mile before reset. Then, 0001 is the reading 1 mile after reset, 0010 occurs 2 miles after reset, and 0011 indicates 3 miles after reset.

Positive and Negative Numbers

"Before" and "after" are synonymous with "negative" and "positive." Figure 6.2 illustrates this idea with the number line of basic algebra: 0 marks the origin, positive numbers are on the right, and negative numbers are on the left. The odometer readings are the binary equivalents of decimal numbers: 1000 is the binary equivalent of –8, 1001 stands for –7, 1010 stands for –6, and so on.

The odometer readings in Fig. 6.2 demonstrate how positive and negative numbers are stored in a typical microcomputer. Here are two important ideas to notice about these odometer readings. First, the MSB is the sign bit: 0 represents a + sign, and 1 stands for a – sign. Second, the negative numbers in Fig. 6.2 are the 2′s complements of the positive numbers, as you can see in the following:

Magnitude	Positive	Negative
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	—	1000

Except for the last entry, the positive and negative numbers are 2′s complements of each other.

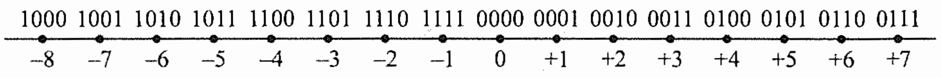


Fig. 6.2 Representing decimal numbers as 2′s complements

In other words, you can take the 2′s complement of a positive binary number to find the corresponding negative binary number. For instance:

$$3 \rightarrow 0011$$
$$-3 \leftarrow 1101$$

After taking the 2′s complement of 0011, we get 1101, which represents –3. The principle also works in reverse:

$$-7 \rightarrow 1001$$
$$+7 \leftarrow 0111$$

After taking the 2′s complement of 1001, we obtain 0111, which represents +7.

What does the foregoing mean? It means that taking the 2′s complement is equivalent to *negation*, changing the sign of the number. Why is this important? Because it's easy to build a logic circuit that produces the 2′s complement. Whenever this circuit takes the 2′s complement, the output is the negative of the input. This key idea leads to an incredibly simple arithmetic circuit that can add and subtract.

In summary, here are the things to remember about 2′s complement representation:

1. Positive numbers always have a sign bit of 0, and negative numbers always have a sign bit of 1.
2. Positive numbers are stored in sign-magnitude form.
3. Negative numbers are stored as 2′s complements.
4. Taking the 2′s complement is equivalent to a sign change.

Converting to and from 2′s Complement Representation

We need a fast way to express numbers in 2′s complement representation. Appendix 2 lists all 8-bit numbers in positive and negative form. You will come to love this Appendix if you have to work a lot with negative numbers. By reading either the positive or negative column, you can quickly convert from decimal to the 2′s complement representation, or vice versa.

Here are some examples of using Appendix 2 to convert from decimal to 2′s complement representation:

$$+23 \rightarrow 17\text{H} \rightarrow 0001\ \ 0111$$
$$-48 \rightarrow \text{D0H} \rightarrow 1101\ \ 0000$$
$$-93 \rightarrow \text{A3H} \rightarrow 1010\ \ 0011$$

Of course, you can use Appendix 2 in reverse. Here are examples of converting from 2′s complement representation to decimal:

$$0111\ \ 0111 \rightarrow 77\text{H} \rightarrow +119$$
$$1110\ \ 1000 \rightarrow \text{E8H} \rightarrow -24$$
$$1001\ \ 0100 \rightarrow 94\text{H} \rightarrow -108$$

A final point. Look at the last two entries in Appendix 2. As you see, +127 is the largest positive number in 2′s complement representation, and −128 is the largest negative number. Similarly, in the 4-bit odometer discussed earlier, +7 was the largest positive number, and −8 was the largest negative number. The largest negative number has a magnitude that is one greater than the largest positive number. This *slight asymmetry* of 2′s complement representation has no particular meaning, but it is something to keep in mind when we discuss overflows.

Example 6.8 A first-generation microcomputer stores 1 byte at each address or memory location, Show how the following decimal numbers are stored with the use of 2′s complement representation: +20, −35, +47, −67, −98, +112, and −125. The first byte starts at address 2000.

Solution With Appendix 2, we have

Address	Binary contents	Hexadecimal contents	Decimal contents
2000	0001 0100	14H	+20
2001	1101 1101	DDH	−35
2002	0010 1111	2DH	+47
2003	1011 1101	BDH	−67
2004	1001 1110	9EH	−98
2005	0111 0000	70H	+112
2006	1000 0011	83H	−125

The computer actually stores binary 0001 0100 at address 2000. Instead of saying 0001 0100, however, we may prefer to say that it stores 14H. To anyone who knows the hexadecimal code, 14H, means the same thing as 0001 0100, but 14H is much easier to say. To the person on the street who knows only the decimal code, we would say that +20 is stored at address 2000.

As you see, understanding computer operation requires knowledge of the different codes being used. Get this into your head, and you are on the way to understanding how computers work.

Example 6.9 Express −19,750 in 2′s complement representation. Then show how this number is stored starting at address 2000. Use hexadecimal notation to compress the data.

Solution The number −19,750 is outside the range of Appendix 2, so we have to fall back on Appendix 1. Start by converting the magnitude to binary form. With Appendix 1, we have

$$19{,}750 \rightarrow 4\text{D}26\text{H} \rightarrow 0100\ \ 1101\ \ 0010\ \ 0110$$

Now, take the 2′s complement to obtain the negative value:

$$1011\ 0010\ 1101\ 1001 + 1 = 1011\ 0010\ 1101\ 1010$$

This means that

$$-19{,}750 \rightarrow 1011\ 0010\ 1101\ 1010$$

In hexadecimal notation, this is expressed

$$1011\ 0010\ 1101\ 1010 \rightarrow \text{B2DAH}$$

The memory of a first-generation microcomputer is organized in bytes. Each address or memory location contains 1 byte. Therefore, a first-generation microcomputer has to break a 16-bit number like B2DA into 2 bytes: an upper byte of B2 and a lower byte of DA. The lower byte is stored at the lower address and the upper byte, at the next-higher address like this:

Address	Binary contents	Hexadecimal contents
2000	1101 1010	DA
2001	1011 0010	B2

The same approach, lower byte first and upper byte second, is used with first-generation microcomputers such as the Apple II and TRS-80.

SELF-TEST

9. What is the 1′s complement representation of 1101 0110?
10. What is the 2′s complement representation of 1101 0110?

6.6 2′S COMPLEMENT ARITHMETIC

Early computers used sign-magnitude numbers for positive and negative values. This led to complicated arithmetic circuits. Then an engineer discovered that 2′s complement representation could simplify arithmetic *hardware*. (This refers to the electronic, magnetic, and mechanical devices of a computer.) Since then, 2′s complement representation has become a universal code for processing positive and negative numbers.

Help from the Binary Odometer

Addition and subtraction can be visualized in terms of a binary odometer. When you add a positive number, this is equivalent to advancing the odometer reading. When you add a negative number, this has the effect of turning the odometer backward. Likewise, subtraction of a positive number reverses the odometer, but subtraction of a negative number advances it. As you read the following discussion of addition and subtraction, keep the binary odometer in mind because it will help you to understand what's going on.

Addition

Let us take a look at how binary numbers are added. There are four possible cases: both numbers positive, a positive number and a smaller negative number, a negative number and a smaller positive number, and both numbers negative. Let us go through all four cases for a complete coverage of what happens when a computer adds numbers.

Case 1 Both positive. Suppose that the numbers are +83 and +16. With Appendix 2, these numbers are converted as follows:

$$\begin{array}{r}
+83 \rightarrow 0101\ \ 0011 \\
+16 \rightarrow 0001\ \ 0000
\end{array}$$

Then, here is how the addition appears:

$$\begin{array}{rr}
+83 & 0101\ \ 0011 \\
+16 & +\ 0001\ \ 0000 \\
\hline
99 & 0110\ \ 0011
\end{array}$$

Nothing unusual happens here. Column-by-column addition produces a binary answer of 0110 0011. Mentally convert this to 63H. Now, look at Appendix 2 to get

$$63\text{H} \rightarrow 99$$

This agrees with the decimal sum.

Case 2 Positive and smaller negative. Suppose that the numbers are +125 and –68. With Appendix 2, we obtain

$$\begin{array}{r}
+125 \rightarrow 0111\ \ 1101 \\
-68 \rightarrow 1011\ \ 1100
\end{array}$$

The computer will fetch these numbers from its memory and send them to an adding circuit. The numbers are then added column by column, including the sign bits to get

$$\begin{array}{rr}
125 & 0111\ \ 1101 \\
+\ (-68) & +1011\ \ 1100 \\
\hline
57 & 1\ 0011\ \ 1001 \rightarrow 0011\ \ 1001
\end{array}$$

With 8-bit arithmetic, you *disregard the final carry* into the ninth column. The reason is related to the binary odometer, which ignores final carries. In other words, when the eighth wheel resets, it does not generate a carry because there is no ninth wheel to receive the carry. You can convert the binary answer to decimal as follows:

$$\begin{array}{rl}
0011\ \ 1001 \rightarrow 39\text{H} & \text{(mental conversion)} \\
39\text{H} \rightarrow +57 & \text{(look in Appendix 2)}
\end{array}$$

Case 3 Positive and larger negative. Let's use +37 and –115. Appendix 2 gives these 2's complement representations:

$$\begin{array}{r}
+37 \rightarrow 0010\ \ 0101 \\
-115 \rightarrow 1000\ \ 1101
\end{array}$$

Then the addition looks like this:

$$\begin{array}{rr}
+37 & 0010\ \ 0101 \\
+(-115) & +1000\ \ 1101 \\
\hline
-78 & 1011\ \ 0010
\end{array}$$

Next, verify the binary answer as follows:

$$1011\ 0010 \rightarrow \text{B2H} \qquad \text{(mental conversion)}$$
$$\text{B2H} \rightarrow -78 \qquad \text{(look in Appendix 2)}$$

Incidentally, mentally converting to hexadecimal before reference to the appendix is an optional step. Most people find it easier to locate B2H in Appendix 2 than 1011 0010. It only saves a few seconds, but it adds up when you have to do a lot of binary-to-decimal conversions.

Case 4 Both negative. Assume that the numbers are –43 and –78. In 2′s complement representation, the numbers are

$$-43 \rightarrow 1101\ \ 0101$$
$$-78 \rightarrow 1011\ \ 0010$$

The addition is

$$\begin{array}{r} -43 \\ +\ (-78) \\ \hline -121 \end{array} \qquad \begin{array}{r} 1101\ \ 0101 \\ +\ 1011\ \ 0010 \\ \hline 1\ 1000\ \ 0111 \rightarrow 1000\ \ 0111 \end{array}$$

Again, we ignore the final carry because it's meaningless in 8-bit arithmetic. The remaining 8 bits convert as follows:

$$1000\ \ 0111 \rightarrow 83\text{H}$$
$$83\text{H} \rightarrow -121$$

This agrees with the answer we obtained by direct decimal addition.

Conclusion

We have exhausted the possibilities. In every case, 2′s complement addition works. In other words, as long as positive and negative numbers are expressed in 2′s complement representation, an adding circuit will automatically produce the correct answer. (This assumes the decimal sum is within the –128 to +127 range. If not, you get an overflow, which we will discuss later.)

Subtraction

The format for subtraction is

$$\begin{array}{r} \text{Minuend} \\ -\ \text{Subtrahend} \\ \hline \text{Difference} \end{array}$$

There are four cases: both numbers positive, a positive number and a smaller negative number, a negative number and a smaller positive number, and both numbers negative.

The question now is *how can we use an adding circuit* to do subtraction. By trickery, of course. From algebra, you already know that adding a negative number is equivalent to subtracting a positive number. If we take the 2′s complement of the subtrahend, addition of the complemented subtrahend gives the correct answer. Remember that the 2′s complement is equivalent to negation. One way to remove all doubt about this critical idea is to analyze the four cases that can arise during a subtraction.

Case 1 Both positive. Suppose that the numbers are +83 and +16. In 2's complement representation, these numbers appear as

$$+83 \rightarrow 0101\ \ 0011$$
$$+16 \rightarrow 0001\ \ 0000$$

To subtract +16 from +83, the computer will send the +16 to a 2's complementer circuit to produce

$$-16 \rightarrow 1111\ \ 0000$$

Then it will add +83 and –16 as follows:

$$\begin{array}{r} 83 \\ +(-16) \\ \hline 67 \end{array} \qquad \begin{array}{r} 0101\ \ 0011 \\ +\ 1111\ \ 0000 \\ \hline 1\ 0100\ \ 0011 \rightarrow 0100\ \ 0011 \end{array}$$

The binary answer converts like this:

$$0100\ \ 0011 \rightarrow 43\text{H}$$
$$43\text{H} \rightarrow +67$$

Case 2 Positive and smaller negative. Suppose that the minuend is +68 and the subtrahend is –27. In 2's complement representation, these numbers appear as

$$+68 \rightarrow 0100\ \ 0100$$
$$-27 \rightarrow 1110\ \ 0101$$

The computer sends –27 to a 2's complementer circuit to produce

$$+27 \rightarrow 0001\ \ 1011$$

Then it adds +68 and +27 as follows:

$$\begin{array}{r} +68 \\ +27 \\ \hline 95 \end{array} \qquad \begin{array}{r} 0100\ \ 0100 \\ +0001\ \ 1011 \\ \hline 0101\ \ 1111 \end{array}$$

The binary answer converts to decimal as follows:

$$0101\ \ 1111 \rightarrow 5\text{FH}$$
$$5\text{FH} \rightarrow +95$$

Case 3 Positive and larger negative. Let's use a minuend of +14 and a subtrahend of –108. Appendix 2 gives these 2's complement representations:

$$+14 \rightarrow 0000\ \ 1110$$
$$-108 \rightarrow 1001\ \ 0100$$

The computer produces the 2's complement of –108:

$$+108 \rightarrow 0110\ \ 1100$$

Then it adds the numbers like this:

$$\begin{array}{r} 14 \\ +108 \\ \hline 122 \end{array} \qquad \begin{array}{r} 0000\ \ 1110 \\ +0110\ \ 1100 \\ \hline 0111\ \ 1010 \end{array}$$

The binary answer converts to decimal like this:

$$0111\ \ 1010 \rightarrow 7AH$$
$$7AH \rightarrow +122$$

Case 4 Both negative. Assume that the numbers are –43 and –78. In 2′s complement representation, the numbers are

$$-43 \rightarrow 1101\ \ 0101$$
$$-78 \rightarrow 1011\ \ 0010$$

First, take the 2′s complement of –78 to get

$$+78 \rightarrow 0100\ \ 1110$$

Then add to obtain

$$\begin{array}{r} -43 \\ +78 \\ \hline 35 \end{array} \qquad \begin{array}{r} 1101\ \ 0101 \\ +\ 0100\ \ 1110 \\ \hline 1\ 0010\ \ 0011 \rightarrow 0010\ \ 0011 \end{array}$$

Then

$$0010\ \ 0011 \rightarrow 23H$$
$$23H \rightarrow +35$$

Overflow

We have covered all cases of addition and subtraction. As shown, 2′s complement arithmetic works and is the standard method used in microcomputers. In 8-bit arithmetic, the only thing that can go wrong is a sum outside the range of –128 to +127. When this happens, there is an overflow into the sign bit, causing a sign change. With the typical microcomputer, the programmer has to write instructions that check for this change in the sign bit.

Let's take a look at overflow problems. Assume that both input numbers are in the range of –128 to +127. If a positive and a negative number are being added, an overflow is impossible because the answer is always less than the larger of the two numbers being added. Trouble can arise only when the arithmetic circuit adds two positive numbers or two negative numbers. Then, it is possible for the sum to be outside the range of –128 to +127. (Subtraction is included in the foregoing discussion because the arithmetic circuit adds the complemented subtrahend.)

Case 1 Two positive numbers. Suppose that the numbers being added are +100 and +50. The decimal sum is +150, so an overflow occurs into the MSB position. This overflow forces the sign bit of the answer to change. Here is how it looks:

$$\begin{array}{r} 100 \\ +\ 50 \\ \hline 150 \end{array} \qquad \begin{array}{r} 0110\ \ 0100 \\ +\ 0011\ \ 0010 \\ \hline 1001\ \ 0110 \end{array}$$

The sign bit is negative, despite the fact that we added two positive numbers. Therefore, the overflow has produced an incorrect answer.

Case 2 Two negative numbers. Suppose that the numbers are –85 and –97. Then

$$\begin{array}{r} -85 \\ +\ (-97) \\ \hline 182 \end{array} \qquad \begin{array}{r} 1010\ \ 1011 \\ +\ 1001\ \ 1111 \\ \hline 1\ 0100\ \ 1010 \rightarrow 0100\ \ 1010 \end{array}$$

The 8-bit answer is 0100 1010. The sign bit is positive, but we know that the right answer should contain a negative sign bit because we added two negative numbers.

What to Do with an Overflow

Overflows are a software problem, not a hardware problem. (*Software* means a program or list of instructions telling the computer what to do.) The programmer must test for an overflow after each addition or subtraction. A change in the sign bit is easy to detect. All the programmer does is include instructions that compare the sign bits of the two numbers being added. When these are the same, the sign bit of the answer is compared to either of the preceding sign bits. If the sign bit is different, more instructions tell the computer to change the processing to 16-bit arithmetic. You will learn more about overflows, 16-bit arithmetic, and related topics if you study assembly-language programming.

Example 6.10 How would an 8-bit microcomputer process this:

$$\begin{array}{r} 18{,}357 \\ -\ 12{,}618 \\ \hline ? \end{array}$$

Solution It would use *double-precision arithmetic*, synonymous with 16-bit arithmetic. This arithmetic is used with 16-bit numbers in this form:

$$X_{15}X_{14}X_{13}X_{12}\quad X_{11}X_{10}X_9X_8\quad X_7X_6X_5X_4\quad X_3X_2X_1X_0$$

Numbers like these have an upper byte $X_{15} \cdots X_8$ and a lower byte $X_7 \cdots X_0$. To perform 16-bit arithmetic, an 8-bit microcomputer has to operate on each byte separately. The idea is similar to Example 6.3, where the lower bytes were added and then the upper bytes.

Here is how it is done. With Appendix 1, we have

$$18{,}357 \rightarrow 47B5H \rightarrow 0100\ \ 0111\ \ 1011\ \ 0101$$
$$12{,}618 \rightarrow 314AH \rightarrow 0011\ \ 0001\ \ 0100\ \ 1010$$

The 2's complement of 12,618 is

$$-12{,}618 \rightarrow CEB6H \rightarrow 1100\ \ 1110\ \ 1011\ \ 0110$$

The addition is carried out in two steps of 8-bit arithmetic. First, the lower bytes are added:

$$\begin{array}{r} 1011\ \ 0101 \\ +\ 1011\ \ 0110 \\ \hline 1\ 0110\ \ 1011 \end{array} \rightarrow X_8X_7X_6X_5X_4\ X_3X_2X_1X_0$$

The computer will store $X_7 \cdots X_0$. The carry X_8 is used in the addition of the upper bytes.

Now, the computer adds the upper bytes plus the carry as follows:

$$\begin{array}{r} 1 \leftarrow X_8 \\ 0100\ \ 0111 \\ +\ 1100\ \ 1110 \\ \hline 1\ 0001\ \ 0110 \rightarrow 0001\ \ 0110 \end{array}$$

To obtain the final answer, the two 8-bit answers are combined:

0001 0110 0110 1011

Notice that the MSB is 0, which means that the answer is positive. With Appendix 1, we can convert this answer to decimal form:

$$0001\ 0110\ 0110\ 1011 \rightarrow 166BH \rightarrow +5739$$

SELF-TEST

11. What is the standard method for doing binary arithmetic in nearly all microprocessors?
12. How is 2's complement representation used to perform subtraction?

6.7 ARITHMETIC BUILDING BLOCKS

We are on the verge of seeing a logic circuit that performs 8-bit arithmetic on positive and negative numbers. But first we need to cover three basic circuits that will be used as building blocks. These building blocks are the half-adder, the full-adder, and the controller inverter. Once you understand how these work, it is only a short step to see how it all comes together, that is, how a computer is able to add and subtract binary numbers of any length.

Half-Adder

When we add two binary numbers, we start with the least-significant column. This means that we have to add two bits with the possibility of a carry. The circuit used for this is called a *half-adder*. Figure 6.3 shows how to build a half-adder. The output of the exclusive–OR gate is called the *SUM*, while the output of the AND gate is the *CARRY*. The AND gate produces a high output only when both inputs are high. The exclusive-OR gate produces a high output if either input, but not both, is high. Table 6.2 shows the truth table of a half-adder.

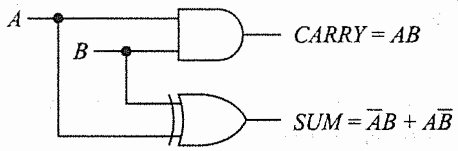


Fig. 6.3 Half-adder

Table 6.2 Half-adder Truth Table

A	*B*	*CARRY*	*SUM*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

When you examine each entry in Table 6.2, you are struck by the fact that a half-adder performs binary addition.

As you see, the half-adder mimics our brain processes in adding bits. The only difference is the half-adder is about a million times faster than we are.

Full-Adder

For the higher-order columns, we have to use a *full-adder*, a logic circuit that can add 3 bits at a time. The third bit is the carry from a lower column. This implies that we need a logic circuit with three inputs and two outputs, similar to the full-adder shown in Fig. 6.4a. (Other designs are possible. This one is the simplest.)

Table 6.3 shows the truth table of a full-adder. You can easily check this truth table for its validity. For instance, CARRY is high in Fig. 6.4a when two or more of the ABC inputs are high; this agrees with the CARRY column in Table 6.3. Also, when an odd number of high ABC inputs drives the exclusive-OR gate, it produces a high output; this verifies the SUM column of the truth table.

Table 6.3 Full-Adder Truth Table

A	*B*	*C*	*CARRY*	*SUM*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

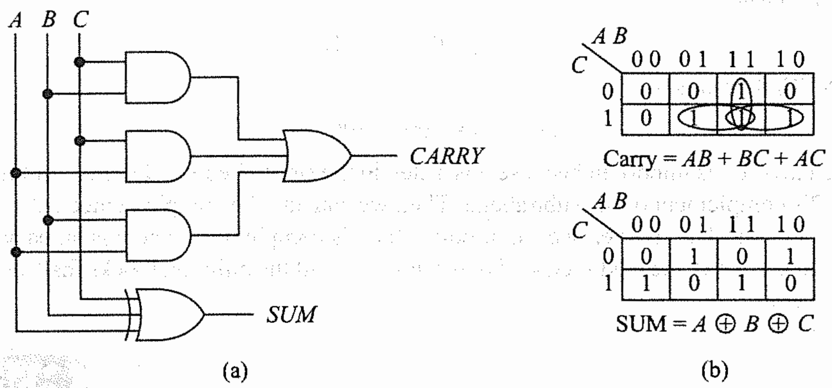


Fig. 6.4 (a) Full-adder, (b) Karnaugh map of Table 6.3

When you examine each entry in Table 6.3, you can see that a full-adder performs binary addition on 3 bits.

From this truth table we get Karnaugh map as shown in Fig. 6.4b that gives following logic equations,

$$\text{CARRY} = AB + BC + AC \quad \text{and} \quad \text{SUM} = A \oplus B \oplus C.$$

A general representation of full-adder which adds i-th bit A_i and B_i of two numbers A and B and takes carry from (i–1)th bit could be

$$C_i = A_iB_i + B_iC_{i-1} + A_iC_{i-1} \text{ or } C_i = A_iB_i + (A_i + B_i)C_{i-1} \quad \text{and} \quad S_i = A_i \oplus B_i \oplus C_{i-1}$$

where, C_i and S_i are carry and sum bits generated from the full adder. The second representation of C_i has an interesting meaning. The first term gives, if both A_i and B_i are 1 then $C_i = 1$. The second term gives if any of A_i or B_i is 1 and if there is carry from previous stage, i.e. $C_{i-1} = 1$ then also $C_i = 1$. That this is the case, we can verify from full adder truth table and this understanding is useful in design of fast adder in Section 6.9.

Controlled Inverter

Figure 6.5 shows a *controlled inverter*. When INVERT is low, it transmits the 8-bit input to the output; when INVERT is high, it transmits the l's complement. For instance, if the input number is

$$A_7 \cdots A_0 = 0110 \quad 1110$$

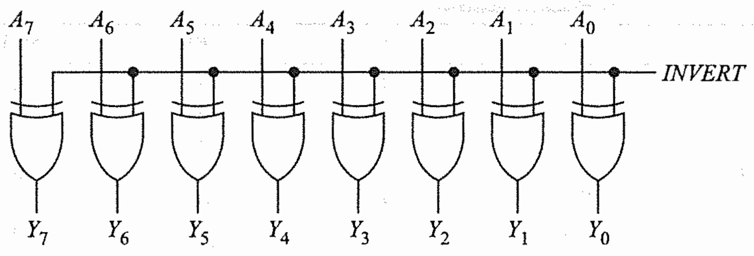


Fig. 6.5 Controlled inverter

a low INVERT produces

$$Y_7 \cdots Y_0 = 0110 \quad 1110$$

But a high INVERT results in

$$Y_7 \cdots Y_0 = 1001 \quad 0001$$

The controlled inverter is important because it is a step in the right direction. During a subtraction, we first need to take the 2's complement of the subtrahend. Then we can add the complemented subtrahend to obtain the answer. With a controlled inverter, we can produce the 1's complement. There is an easy way to get the 2's complement, discussed in the next section. So, we now have all the building blocks: half-adder, full-adder, and controlled inverter.

SELF-TEST

13. What are the inputs and outputs of a half-adder?
14. What are the inputs and outputs of a full-adder?
15. The SUM output of a full-adder is easily implemented using an exclusive-OR gate. (T or F)

6.8 THE ADDER-SUBTRACTER

We can connect full-adders as shown in Fig. 6.6 to add or subtract binary numbers. The circuit is laid out from right to left, similar to the way we add binary numbers. Therefore, the least-significant column is on the right, and the most-significant column is on the left. The boxes labeled FA are full-adders. (Some adding circuits use a half-adder instead of a full-adder in the least-significant column.)

The CARRY OUT from each full-adder is the CARRY IN to the next-higher full-adder. The numbers being processed are $A_7 \cdots A_0$ and $B_7 \cdots B_0$, and the answer is $S_7 \cdots S_0$. With 8-bit arithmetic, the final carry is ignored for reasons given earlier. With 16-bit arithmetic, the final carry is the carry into the addition of the upper bytes.

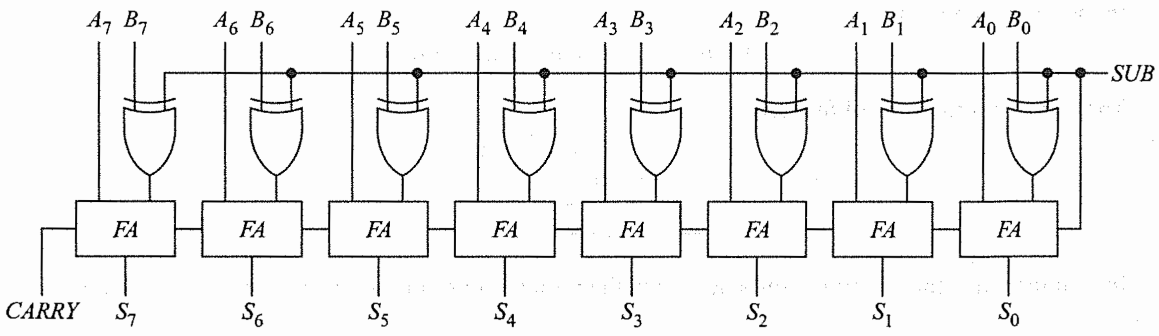


Fig. 6.6 Binary adder-subtracter

Addition

Here is how an addition appears:

$$\begin{array}{rl} & A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ + & B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline & S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

During an addition, the *SUB* signal is deliberately kept in the low state. Therefore, the binary number B_7 … B_0 passes through the controlled inverter with no change. The full-adders then produce the correct output sum. They do this by adding the bits in each column, passing carries to the next higher column, and so on. For instance, starting at the LSB, the full-adder adds A_0, B_0, and SUB. This produces a SUM of S_0 and a CARRY OUT to the next-higher full-adder. The next-higher full-adder then adds A_1, B_1, and the CARRY IN to produce S_1 and a CARRY OUT. A similar addition occurs for each of the remaining full-adders, and the correct sum appears at the output lines.

For instance, suppose that the numbers being added are +125 and –67. Then, $A_7 \cdots A_0 = 0111\ \ 1101$ and $B_7 \cdots B_0 = 1011\ \ 1101$. This is the problem:

$$\begin{array}{r} 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline ? \end{array}$$

Since SUB = 0 during an addition, the CARRY IN to the least-significant column is 0:

$$\begin{array}{r} 0 \leftarrow \text{SUB} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline ? \end{array}$$

The first full-adder performs this addition:

$$0 + 1 + 1 = 0 \quad \text{with a carry of } 1$$

The CARRY OUT of the first full-adder is the CARRY IN to the second full-adder:

$$\begin{array}{r} 1 \leftarrow \text{Carry} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 0 \end{array}$$

In the second column

$$1 + 0 + 0 = 1 \quad \text{with a carry of } 0$$

The carry goes to the third full-adder:

$$\begin{array}{r} 0 \leftarrow \text{Carry} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 10 \end{array}$$

In a similar way, the remaining full-adders add their 3 input bits until we arrive at the last full-adder:

$$\begin{array}{r} 1 \qquad\quad \leftarrow \text{Carry} \\ 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 0011\ \ 1010 \end{array}$$

When the CARRY IN to the MSB appears, the full-adder produces

$$1 + 0 + 1 = 0 \quad \text{with a carry of } 1$$

The addition process ends with a final carry:

$$\begin{array}{r} 0111\ \ 1101 \\ +1011\ \ 1101 \\ \hline 1\ 0011\ \ 1010 \end{array}$$

During 8-bit arithmetic, this last carry is ignored as previously discussed; therefore, the answer is

$$S_7 \cdots S_0 = 0011\ \ 1010$$

This answer is equivalent to decimal +58, which is the algebraic sum of the numbers we started with: +125 and –67.

Subtraction

Here is how a subtraction appears:

$$\begin{array}{r} A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\ B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

During a subtraction, the SUB signal is deliberately put into the high state. Therefore, the controlled inverter produces the 1's complement of $B_7 \cdots B_0$. Furthermore, because SUB is the CARRY IN to the first full-adder, the circuit processes the data like this:

$$\begin{array}{r} 1 \leftarrow \text{SUB} \\ A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\ \overline{B}_7\overline{B}_6\overline{B}_5\overline{B}_4 \quad \overline{B}_3\overline{B}_2\overline{B}_1\overline{B}_0 \\ \hline S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

When $A_7 \cdots A_0 = 0$, the circuit produces the 2's complement of $B_7 \cdots B_0$ because 1 is being added to the 1's complement $B_7 \cdots B_0$. When $A_7 \cdots A_0$ does not equal zero, the effect is equivalent to adding $A_7 \cdots A_0$ and the 2's complement of $B_7 \cdots B_0$.

Here is an example. Suppose that the numbers are +82 and +17. Then $A_7 \cdots A_0 = 0101\ \ 0010$ and $B_7 \cdots B_0 = 0001\ \ 0001$. The controlled inverter produces the l's complement of B, which is 1110 1110. Since SUB = 1 during a subtraction, the circuit performs the following addition:

$$\begin{array}{r} 1 \leftarrow \text{SUB} \\ 0101\ \ 0010 \\ +\ 1110\ \ 1110 \\ \hline 1\ 0100\ \ 0001 \end{array}$$

For 8-bit arithmetic, the final carry is ignored as previously discussed; therefore, the answer is

$$S_7 \cdots S_0 = 0100\ \ 0001$$

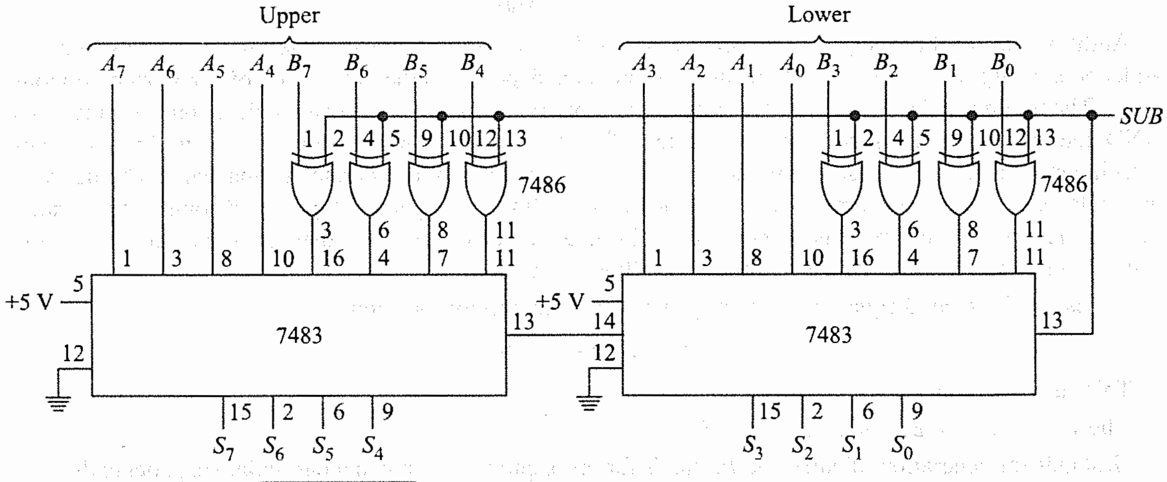

Upper
Lower
A_7 A_6 A_5 A_4 B_7 B_6 B_5 B_4
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
SUB
7486
7486
7483
7483
+5 V
+5 V
S_7 S_6 S_5 S_4
S_3 S_2 S_1 S_0

Fig. 6.7 **Two 7483s can add or subtract bytes**

This answer is equivalent to decimal +65, which is the algebraic difference between the numbers we started with: +82 and +17.

Example 6.11 Show how to build an 8-bit adder-subtracter with TTL circuits.

Solution The 7483 is a TTL circuit with four full-adders. This means that it can add nibbles. To add bytes, we need to use two 7483s as shown in Fig. 6.7. The CARRY OUT (pin 14) of the lower 7483 is used as the CARRY IN (pin 13) to the upper 7483. This allows the two 7483s to add 8-bit numbers. Two 7486s form the controlled inverter needed for subtraction.

The 74LS83, 74283, and 72LS283 are all TTL 4-bit adder ICs. They are pin-for-pin compatible, except that the '2.83 and 'LS283 have $+V_{CC}$ on pin 16 and GROUND on pin 8. The 74HC283 is the CMOS version of the same 4-bit adder.

The 74181, 74LS181, and 74LS381 are TTL ALUs, and the 74HC381 is the CMOS equivalent. Each is capable of adding two 4-bit binary numbers as well as performing numerous other logic operations.

6.9 FAST ADDER

Fast adder is also called *parallel adder* or *carry look ahead adder* because that is how it attains high speed in addition operation. Before we go into that circuit, let's see what limits the speed of an adder. Consider, the worst case scenario when two four bit numbers A: 1111 and B: 0001 are added. This generates a carry in the first stage that propagates to the last stage as shown next.

$$\begin{array}{lr} \text{Carry:} & 111 \\ A: & 1111 \\ B: & 0001 \\ \hline & 10000 \end{array}$$

Addition such as these (Fig. 6.6) is called *serial addition* or *ripple carry addition.* It also reveals from the adder equation (given in Section 6.8) result of every stage depends on the availability of carry from previous stage. The minimum delay required for carry generation in each stage is two gate delays, one coming from AND gates (1st level) and second from OR gate (2nd level). For 32-bit serial addition there will be 32 stages working in serial. In worst case, it will require $2 \times 32 = 64$ gate delays to generate the final carry. Though each gate delay is of nanosecond order, serial addition definitely limits the speed of high speed computing. Parallel adder increases the speed by generating the carry in advance (look ahead) and there is no need to wait for the result from previous stage. This is achieved by following method.

Let us use the second equation for carry generation from previous section, i.e.

$$C_i = A_iB_i + (A_i + B_i)C_{i-1}$$

This can be written as, $C_i = G_i + P_iC_{i-1}$

where, $G_i = A_iB_i$ and $P_i = A_i + B_i$

G_i stands for generation of carry and P_i stands for propagation of carry in a particular stage depending on input to that stage. As explained in previous section, if $A_iB_i = 1$, then *i*th stage will generate a carry, no matter previous stage generates it or not. And if $A_i + B_i = 1$ then this stage will propagate a carry if available from previous stage to next stage. Note that, all G_i and P_i are available after one gate delay once the numbers A and B are placed.

Starting from LSB, designated by suffix 0 if we proceed iteratively we get,

$C_0 = G_0 + P_0.C_{-1}$ [C_{-1} will normally be 0 if we are not using it as subtractor or cascading it.]

$C_1 = G_1 + P_1.C_0 = G_1 + P_1.(G_0 + P_0.C_{-1}) = G_1 + P_1.G_0 + P_1P_0.C_{-1}$ [Substituting C_0]

Similarly,

$C_2 = G_2 + P_2.C_1 = G_2 + P_2.(G_1 + P_1.G_0 + P_1P_0.C_{-1})$ [Substituting C_1]

$= G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1}$

$C_3 = G_3 + P_3.C_2 = G_3 + P_3(G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1})$ [Substituting C_2]

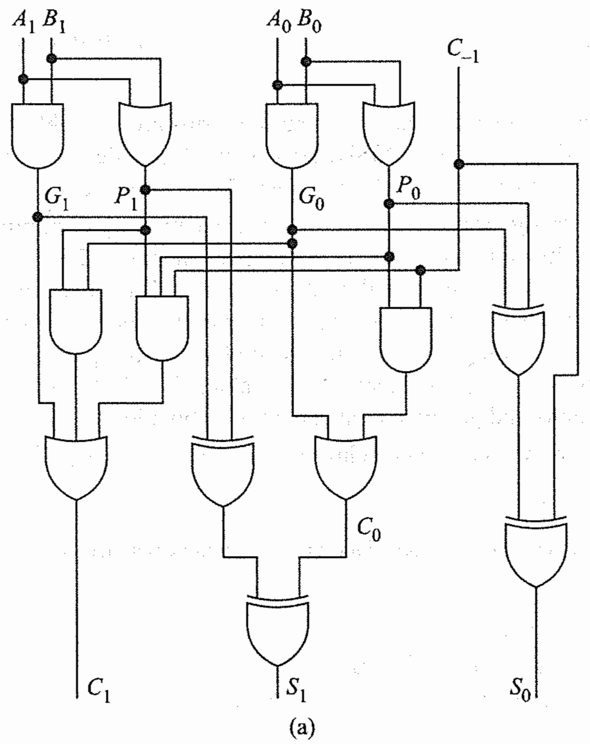
$= G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0 + P_3P_2P_1P_0.C_{-1}$

etc.

The equations look pretty complicated. But do we gain in any way? Note that, these equations can be realized in hardware using multi-input AND and OR gates and in two levels. Now, for each carry whether C_0 or C_3 we require only two gate delays once the G_i and P_i are available. We have already seen they are available after 1 gate delay. Thus parallel adder (circuit diagram for 2-bit is shown in Fig. 6.8a) generates carry within

1 + 2 = 3 gate delays. Note that, after the carry is available at any stage there are two more gate delays from Ex-OR gate to generate the sum bit as we can write $S_i = G_i \oplus P_i \oplus C_{i-1}$.

Thus serial adder in worst case requires at least $(2n + 2)$ gate delays for n-bit addition and parallel adder requires only 3 + 2 = 5 gate delays for that. One can imagine the gain for higher values of n. However, there is a caution. We cannot increase n indiscriminately for parallel adder as every logic gate has a capacity to accept at most a certain number of inputs, termed *fan-in*. This is a characteristic of the logic family to which the gate belongs. More about this is discussed in Chapter 14. The other disadvantage of parallel adder is increased hardware complexity for large n. In Fig. 6.8b we present functional diagram and pin connections of a popular fast adder, IC 74283.



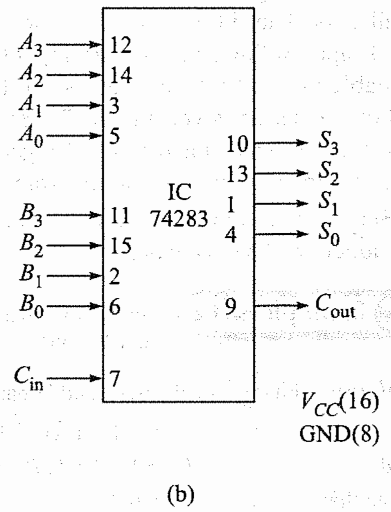


Fig. 6.8 **(a) Logic circuit for 2-bit fast adder, (b) Functional diagram of IC 74283**

Now, how do we add two 8-bit numbers using IC 74283? Obviously, we need two such devices and C_{out} of LSB adder will be fed as C_{in} of MSB unit. This way each individual 4-bit addition is done parallely but between two ICs carry propagates by rippling. To avoid carry ripple between two ICs and get truly parallel addition the following approach can be useful. Let each individual 4-bit adder unit generate two additional outputs Group Carry Generate (G_{3-0}) and Group Carry Propagate (P_{3-0}). They are defined as follows

$$G_{3-0} = G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0$$
$$P_{3-0} = P_3P_2P_1P_0$$

so that

$$C_3 = G_{3-0} + P_{3-0}\,C_{-1} \qquad \text{[From equation of } C_3 \text{ in previous discussion]}$$

Now, let us see how this is useful in 8-bit parallel addition. For the 4-bit adder adding MSB taking C_3 as carry input, we can similarly write

$$C_7 = G_{7-4} + P_{7-4}\,C_3 \qquad [C_3 \text{ is equivalent to } C_{-1} \text{ input for this adder}]$$

where

$$G_{7-4} = G_7 + P_7G_6 + P_7P_6.G_5 + P_7\,P_6P_5.G_4$$
$$P_{7-4} = P_7P_6P_5P_4$$

Thus $$C_7 = G_{7-4} + P_{7-4}(G_{3-0} + P_{3-0}\,C_{-1}) \qquad [\text{substituting } C_3]$$

or $$C_7 = G_{7-4} + P_{7-4}G_{3-0} + P_{7-4}P_{3-0}\,C_{-1}$$

What do we get from above equation? Group carry generation and propagation terms are available from respective adder blocks (G_{3-0}, P_{3-0} from LSB and G_{7-4}, P_{7-4} from MSB) after 3 and 2 gate delays respectively. This comes from the logic equations that define them with G_i, P_i available after 1 gate delay.

Once these group-carry terms are available, we can generate C_7 from previous equation by designing a small Look Ahead Carry (LAC) Generator circuit. This requires a bank of AND gates (here one 2 input and one 3 input) followed by a multi-input OR gate (here, three input) totaling 2 gate delays. Thus final carry is available in 3 + 2 = 5 gate delays and this indeed is what we were looking for in parallel addition. In next section we discuss a versatile IC 74181 that while performing 4-bit addition generates this group carry generation and propagation terms. LAC generator circuits are also commercially available; IC 74182 can take up to four pairs of group carry terms from four adder units and generate final carry for 16 bit addition.

Before we go to next section can you answer after how many gate delays the sum bits ($S_{15}\ldots S_0$) of 16 bit fast adders will be available?

Example 6.12 Show how final carry is generated for a parallel adder when two numbers added are A: 1111 and B: 0001.

Solution First it calculates, G_i and P_i parallely.

$G_0 = 1.1 = 1$, $G_1 = 1.0 = 0$, $G_2 = 1.0 = 0$, $G_3 = 1.0 = 0$.

and $P_0 = 1 + 1 = 1$, $P_1 = 1 + 0 = 1$, $P_2 = 1 + 0 = 1$, $P_3 = 1 + 0 = 1$.

Note that $C_{-1} = 0$.

Then substituting these in equation of C_3 we get final carry as

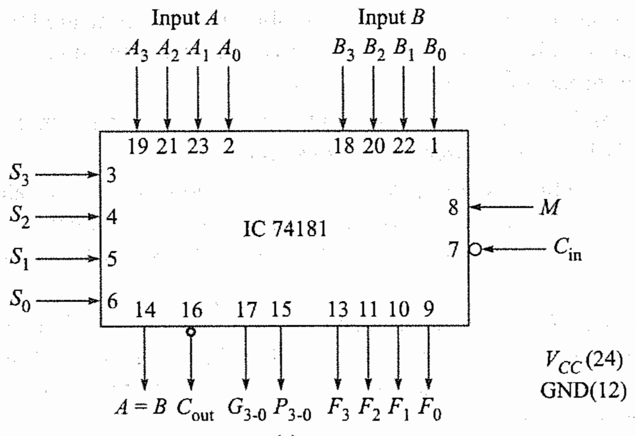
$$\begin{aligned} C_3 &= G_3 + P_3G_2 + P_3P_2.G_1 + P_3\,P_2P_1.G_0 + P_3P_2P_1P_0.C_{-1} \\ &= 0 + 1.0 + 1.1.0 + 1.1.1.1 + 1.1.1.1.0 \\ &= 0 + 0 + 0 + 1 + 0 \\ &= 1 \end{aligned}$$

SELF-TEST

16. What is the savings in time in a parallel adder?
17. What is the maximum number of inputs for an OR gate in a 4-bit parallel adder?

6.10 ARITHMETIC LOGIC UNIT

Arithmetic Logic Unit, popularly called ALU is multifunctional device that can perform both arithmetic and logic function. ALU is an integral part of central processing unit or CPU of a computer. It comes in various forms with wide range of functionality. Other than normal addition, subtraction it can also perform increment, decrement operations. As logic unit it performs usual AND, OR, NOT, EX-OR and many other complex logic functions. It also comes with PRESET and CLEAR options, invoking which all the function outputs are made 1 and 0 respectively. Normally, a mode selector input (M) decides whether ALU performs a logic operation or an arithmetic operation. In each mode different functions are chosen by appropriately activating a set of selection inputs.



(a)

$S_3\,S_2\,S_1\,S_0$	$M = 1$ (Logic Function)	$M = 0$ (Arithmetic Function) $C_{in} = 1$ (For $C_{in} = 0$, add 1 to F)
0 0 0 0	$F = A'$	$F = A$
0 0 0 1	$F = (A + B)'$	$F = A + B$
0 0 1 0	$F = A'B$	$F = A + B'$
0 0 1 1	$F = 0$	$F =$ minus 1
0 1 0 0	$F = (AB)'$	$F = A$ plus (AB')
0 1 0 1	$F = B'$	$F = (A + B)$ plus (AB')
0 1 1 0	$F = A \oplus B$	$F = A$ minus B minus 1
0 1 1 1	$F = AB'$	$F = AB'$ minus 1
1 0 0 0	$F = A' + B$	$F = A$ plus (AB)
1 0 0 1	$F = (A \oplus B)'$	$F = A$ plus B
1 0 1 0	$F = B$	$F = (A + B')$ plus (AB)
1 0 1 1	$F = AB$	$F = AB$ minus 1
1 1 0 0	$F = 1$	$F = A$ plus A
1 1 0 1	$F = A + B'$	$F = (A + B)$ plus A
1 1 1 0	$F = A + B$	$F = (A + B')$ plus A
1 1 1 1	$F = A$	$F = A$ minus 1

(b)

Fig. 6.9 **(a) Functional representation of ALU IC 74181, (b) Its truth table**

In this section, we take up one very popular discrete ALU device from TTL family for discussion. IC 74181 is a 4-bit ALU that can generate 16 different kinds of outputs in each mode selected by four selection inputs S_3, S_2, S_1 and S_0. The functional diagram of this IC with pin numbers and corresponding truth table is shown in Fig. 6.9(a) and Fig. 6.9(b) respectively. Note that this truth table considers data inputs A and B are active high. A similar but different truth table is obtained if data is considered as active low.

Well, the truth table is pretty exhaustive though one might wonder what could be the utility of functions like $(A+B)$ plus AB'. But a careful observation shows one important function missing, that of a comparator. Is it truly so? No, it can be obtained in an indirect way. The C_{out} is activated (active low) by addition as well as subtraction because subtraction is carried out by 2's complement addition. Note that, if the result of an arithmetic operation is negative it will be available in 2's complement form. The $A = B$ output is activated when all the function outputs are 1, i.e. $F_3 \ldots F_0 = 1111$. Output $A = B$, together with C_{out} can give functions like $A > B$ and $A < B$. Note that $A = B$ is an open collector output; thus when more than 4-bits are to be compared this output of different ALU devices are wire-ANDed, simply by knotting outputs together to get the final result. To know more about open collector gates refer to Section 14.5 of Chapter 14.

The outputs C_{out}, $G_{3\text{-}0}$ and $P_{3\text{-}0}$ are useful when addition and subtraction of more than 4-bits are performed using more than one IC 74181 as discussed in previous sections.

Logic operations are done bit-wise by making $M = 1$ and choosing appropriate select inputs. Note that, carry is inhibited for $M = 1$. Let us see how AND operation between two 4-bit numbers 1101 and 0111 is to be performed. Enter input $A_3..A_0 = 1101$ and $B_3..B_0 = 0111$. Make $S_3..S_0 = 1011$ and of course $M = 1$ to choose logic function. The output is shown as $F_3..F_0 = 0011$.

For arithmetic operations $M = 0$ to be chosen and we have to appropriately place C_{in} (active low), if any. For example, if we want to add decimal numbers 6 with 4 we have to place 0110 for 6 at A and 0100 for 4 at B. Then with $S_3..S_0 = 1001$ (from truth table) and $C_{in} = 1$ (active low) the output generated is $F_3..F_0 = 1010$ which is decimal equivalent of 10.

Example 6.13

(a) Show how $A > B$ output can be generated in IC 74181 ALU. (b) Also show how $A \geq B$ condition can be checked.

Solution

(a) If $A > B$, then function A minus B will be positive and the result will not be in 2's complement form and more importantly will generate a carry. Refer to discussion and examples in Section 6.6 on 2's complement arithmetic. The final result for such subtraction is obtained by disregarding the carry. But here by checking output carry whether active we can conclude if $A > B$.
Thus to check $A > B$ put $M = 0$ (arithmetic operation), $S_3..S_0 = 0110$ (gives A minus B), $C_{in} = 0$ ($C_{in} = 1$ gives A minus B minus 1) and check carry is generated, i.e. $C_{out} = 0$ (active), which gives $A > B$.

(b) A similar reasoning shows by making $C_{in} = 1$ in above and checking if $C_{out} = 0$ we can verify $A \geq B$ condition.

Example 6.14

Show how bits of input A shifted to left by one unit appear at output F in IC 74181.

Solution We know shifting to left by one unit is equivalent to multiplication by two. Again multiplication by two can be achieved by adding the number with itself once. Thus make data input at A, $M = 0$ (arithmetic operation), $C_{in} = 1$ and $S_3..S_0 = 1100$ (gives A plus A) and we have A shifted by 1 unit to left at function output F.

SELF-TEST

18. What is an ALU?
19. How do you CLEAR all outputs of IC 74181?

6.11 BINARY MULTIPLICATION AND DIVISION

Typical 8-bit microprocessors like the 6502 and the 8085 use software multiplication and division. In other words, multiplication is done with addition instructions and division with subtraction instructions. Therefore, an adder-subtracter is all that is needed for addition, subtraction, multiplication, and division.

For example, multiplication is equivalent to repeated addition. Given a problem such as

$$8 \times 4 = ?$$

the first number is called the *multiplicand* and the second number, the *multiplier*. Multiplying 8 by 4 is the same as adding 8 four times:

$$8 + 8 + 8 + 8 = ?$$

One way to multiply 8 by 4 is to program a computer to add 8 until a total of four 8s have been added. This approach is known as programmed multiplication by repeated addition.

There are other software solutions to multiplication and division that you will learn about if you study assembly-language programming.

There are ICs available that will multiply two binary numbers. For instance, the 74284 and the 74285 will produce an 8-bit binary number that is the product of two 4-bit binary numbers. These ICs are very fast, and the total multiplication time is only about 40 nanoseconds (ns)!

SELF-TEST

20. Explain how one can do division of binary numbers.

6.12 ARITHMETIC CIRCUITS USING HDL

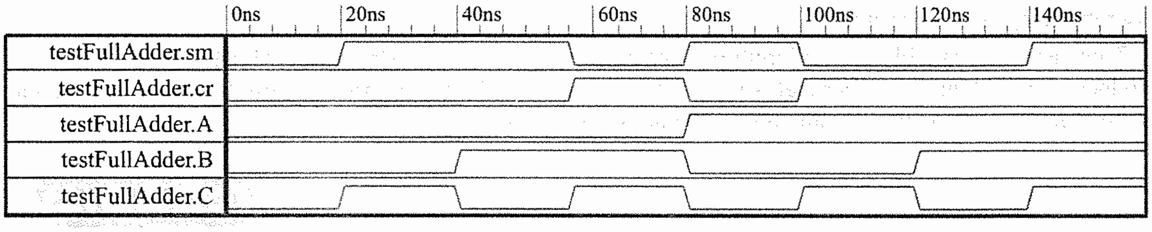
We first describe a full adder circuit and create a test bench to test it. Please refer to discussion of Section 6.7. If A and B are the binary digits to be added and C is the Carry input then output Sum and Carry (represented by sm and cr in following Verilog code) is expressed by equations

$$\text{Sum: sm} = AB + BC + CA \quad \text{and} \quad \text{Carry: cr} = A \oplus B \oplus C$$

We have used a test bench that generates all possible combinations of A, B and C by arithmetic addition and takes less space than test bench described in Chapter 2. The output sum (sm) and carry (cr) for this is shown in simulation waveform. One can see this verifies truth table of a full adder.

```
module testFullAdder;
reg A,B,C;
wire sm, cr;
fulladder fa1(A,B,C,sm,cr);// Circuit instantiated with fa1
initial // simulation begins
  begin
   {A,B,C} = 3'b000; //Initialization A=0,B=0,C=0
         repeat (7) //repeats following statement seven times
   #20 {A,B,C}={A,B,C} + 3'b001; //delay of 20 ns and then increment by 1
   #20 $finish; // simulation ends after generating 8 combinations of ABC
  end            //total time of simulation 7x20+20=160 ns
endmodule

module fulladder(A,B,C,sm,cr); // Description of fulladder Circuit
input A,B,C;
output sm,cr;
assign sm = (A&B)|(B&C)|(C&A);
assign cr = A^B^C;
endmodule
```



Example 6.15 Show Verilog design of 4-bit ripple carry adder.

Solution The code is given as follows. The one in the left hand side ensures ripple carry addition while the one in the right depends on the compiler. Based on considerations like speed, cost and other constraints Verilog compiler implements a 4-bit ripple carry adder in different manner.

```
module adder4bit (sum,cout,a,b,cin); //4-bit ripple carry adder
input [3:0] a,b; //Two 4 bit data to be added
input cin; //Input carry
output [3:0] sum; //4-bit sum output to be generated
output cout; //output carry
wire [2:0] cint; /*internal carry
generated in first three fulladder
fulladder*/
fa0(a[0],b[0],cin,sum[0],cint[0]);
// instantiates fulladder
```

```
/* 4-bit adder, compiler decides
if ripple carry */

module adder4bit
(sum,cout,a,b,cin);
input [3:0] a,b;
```

```
fulladder
fa1(a[1],b[1],cint[0],sum[1],cint[1]);
fulladder
fa2(a[2],b[2],cint[1],sum[2],cint[2]);
fulladder
fa3(a[3],b[3],cint[2],sum[3],cout);
endmodule

module fulladder(A,B,C,sm,cr);
input A,B,C;
output sm,cr;
assign sm = (A&B)|(B&C)|(C&A);
assign cr = A^B^C;
endmodule
```

```
input cin;
output [3:0] sum;
output cout;
assign {cout,sum} = a + b + cin;
endmodule
```

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Show how a half-adder can be realized.

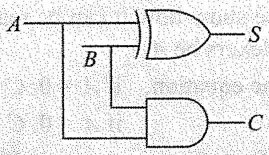
Solution The half-adder truth table and logic equations are reproduced from Section 6.7 in Fig. 6.10a.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$C = AB$

$S = A \oplus B$
$= AB' + A'B$

(a)

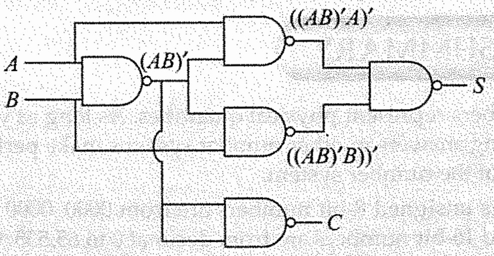


(b)

$C = A{\cdot}B = ((A{\cdot}B)')'$: double complement

$S = A{\cdot}B' + A'B$
$= (A{\cdot}B' + A'B)''$: double complement
$= ((A{\cdot}B')'{\cdot}(A'B)')'$: from Eq. 2.1
$= ((A'{+}B){\cdot}(A{+}B')')'$: from Eq. 2.2
$= (A'B'{+}AB)'$: since $X'X = 0$
$= (A'B')'{\cdot}(AB)'$: from Eq. 2.1
$= (A{+}B){\cdot}(AB)'$: from Eq. 2.2
$= (AB)'A{+}(AB)'B$
$= ((AB)'A{+}(AB)'B)''$: double complement
$= (((AB)'A)'{\cdot}((AB)'B))')'$: from Eq. 2.1

(c)



(d)

Fig. 6.10 **(a) Truth table and logic equation for half-adder, (b) Realization using AND and exclusive-OR gate, (c) Derivation of AND and exclusive-OR relation, (d) Realization using only NAND gates**

In Method-1, the logic relations can directly be realized using AND and exclusive-OR gate as shown in Fig. 6.10b.

In Method-2, we show how it can be realized using only one type of basic gates, say NAND gate. The derivation is shown in Fig. 6.10c and the realization is shown in Fig. 6.10d.

It is left as an exercise to find how it can be realized using only NOR gates.

In Method-3, we show how it can be realized using two 4 to 1 multiplexers. We make use of the truth table to assign data inputs to the multiplexers while A and B are used as select inputs. The realization is shown in Fig. 6.11a.

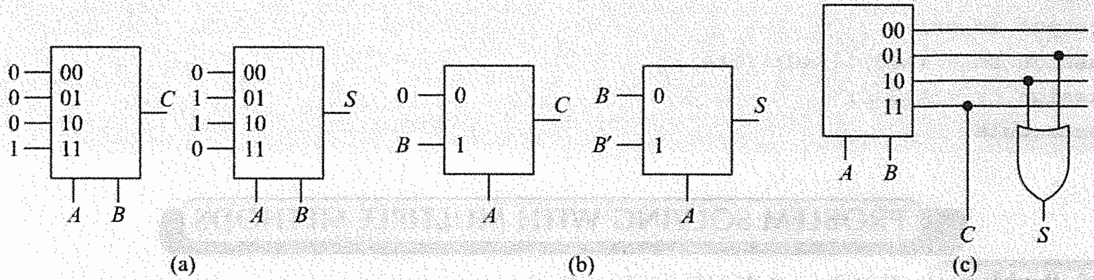


Fig. 6.11 Realization using (a) 4 to 1 multiplexers, (b) 2 to 1 multiplexers, (c) Decoder, OR gate

In Method-4, we show how it can be realized using two 2 to 1 multiplexers. Let the only selected input to the multiplexers be A.

We note from the equation, if $A = 0, C = 0$ and if $A = 1, C = B$

if $A = 0, C = B$ and if $A = 1, C = B'$

The realization from these is shown in Fig. 6.11b where B is used in the data input.

In Method-5, we show how it can be realized using a 2 to 4 decoder and OR gate. The decoder generates all the four minterms $A'B'$, $A'B$, AB' and AB. Carry output is generated directly from AB. Sum output is generated OR-ing $A'B$ and AB'. The realization is shown in Fig. 6.11c.

SUMMARY

Numbers represent physical quantities. As long as you know the number code being used, those strange-looking answers in other number systems make perfect sense. Subscripts can be used as a reminder of the base of the number system.

The unsigned 8-bit numbers are from 0000 0000 to 1111 1111, equivalent to decimal 0 to 255. The unsigned 16-bit numbers are from decimal 0 to 65,535. Overflows occur when a sum exceeds the range of the number system. With 8-bit arithmetic, an overflow occurs when the unsigned sum exceeds 255.

Sign-magnitude numbers use the MSB as a sign bit, with 0 for the + sign and 1 for the – sign. The rest of the bits are for the magnitude of the number. For this reason, 8-bit numbers cover the decimal range of –127 to +127, while 16-bit numbers cover –32,767 to +32,767.

The 2's complement representation is the most widespread code for positive and negative numbers. Positive numbers are coded as sign-magnitude numbers, and negative numbers are coded as 2's complements. The key feature of this number system is that taking the 2's complement of a number is equivalent to changing its sign. This characteristic allows us to subtract numbers by adding the 2's complement of the subtrahend. The advantage is simpler arithmetic hardware.

The half-adder has two inputs and two outputs; it adds 2 bits at a time. The full-adder has three inputs and two outputs; it adds 3 bits at a time. By connecting a controlled inverter and full-adders, we can build an adder-subtracter. This circuit can perform addition, subtraction, multiplication, and division.

A fast adder brings parallelism in addition process, more specifically by generating the carry using extra hardware through a look ahead logic. An Arithmetic Logic Unit is a versatile device, which can generate many useful arithmetic and logic functions with appropriate selection of inputs. Cascading of these devices is usually possible for working with larger sized numbers.

GLOSSARY

- ***arithmetic logic unit*** A device that can perform both arithmetic and logic function based on select inputs.
- ***full-adder*** A logic circuit with three inputs and two outputs. The circuit adds 3 bits at a time, giving a sum and a carry output.
- ***half-adder*** A logic circuit with two inputs and two outputs. It adds 2 bits at a time, producing a sum and a carry output.
- ***hardware*** The electronic, magnetic, and mechanical devices used in a computer or digital system.
- ***LSB*** Least-significant bit.
- ***look ahead carry*** Carry that need not ripple from one stage to other and is obtained through a look ahead logic after the binary numbers are placed in adder unit; useful in fast addition.
- ***magnitude*** The absolute or unsigned value of a number.
- ***microprocessor*** A digital IC that combines the arithmetic and control sections of a computer.
- ***MSB*** Most-significant bit.
- ***parallel addition*** A method of binary addition where carry generation at a particular stage does not depend on availability of carry from previous stage.
- ***overflow*** An unwanted carry that produces an answer outside the valid range of the numbers being represented.
- ***ripple carry*** Carry that ripples from one stage to other in serial addition.
- ***serial addition*** A method of binary addition where carry sequentially propagates from one stage to next stage.
- ***software*** A program or programs. The instructions that tell a computer how to process the data.
- ***2's complement*** The binary number that results when 1 is added to the 1's complement.

PROBLEMS

Section 6.1

6.1 Give the sum in each of the following:

a. $3_8 + 7_8 = ?$ b. $5_8 + 6_8 = ?$

c. $4_{16} + C_{16} = ?$ d. $8_{16} + F_{16} = ?$

6.2 Work out each of these binary sums:

a. 0000 1111 + 0011 0111

b. 0001 0100 + 0010 1001

c. 0001 1000 1111 0110 +
0000 1111 0000 1000

6.3 Show the binary addition of 750_{10} and 538_{10} using 16-bit numbers.

Section 6.2

6.4 Subtract the following: 0100 1111 – 0000 0101.

6.5 Show this subtraction in binary form: $47_{10} - 23_{10}$.

Section 6.3

6.6 Indicate which of the following produces an overflow with 8-bit unsigned arithmetic:

a. $45_{10} + 78_{10}$ b. $34_8 + 56_8$
c. $CF_{16} + 67_{16}$

Section 6.4

6.7 Express each of the following in 8-bit sign-magnitude form:

a. +23 b. +123
c. –56 d. –107

6.8 Convert each of the following sign-magnitude numbers into decimal equivalents:

a. 00110110
b. 1010 1110
c. 1111 1000
d. 1000 1100 0111 0101

Section 6.5

6.9 Express the 1's complement of each of the following in hexadecimal notation:

a. 23H b. 45H
c. C9H d. FDH

6.10 What is the 2's complement of each of these:

a. 0000 1111
b. 0101 1010
c. 1011 1110
d. 1111 0000 1111 0000

6.11 Use Appendix 2 to convert each of the following to 2's complement representation:

a. +78 b. –23
c. –90 d. –121

6.12 Decode the following numbers into decimal values, using Appendix 2:

a. FCH b. 34H
c. 9AH d. B4H

Section 6.6

6.13 Show the 8-bit addition of these decimal numbers in 2's complement representation:

a. +45, +56 b. +89, –34
c. +67, –98

6.14 Show the 8-bit subtraction of these decimal numbers in 2's complement representation:

a. +54, +65 b. +68, –43
c. +16, –38 d. –28, –65

Section 6.7

6.15 Suppose that FD34H is the input to a 16-bit controlled inverter. What is the inverted output in hexadecimal notation? In binary?

Section 6.8

6.16 Expressed in hexadecimal notation, the two input numbers in Fig. 6.6 are 7FH and 4DH. What is the output when SUB is low?

6.17 The input numbers in Fig. 6.6 are 0001 0010 and 1011 1111. What is the output when SUB is high?

Section 6.9

6.18 Show how two IC 74283s can be connected to add two 8-bit numbers. Find the worst case delay.

6.19 Show how a parallel adder generates sum and carry bits while adding two numbers 1001 and 1011. What is the final result?

Section 6.10

6.20 How $A < B$ function is performed in IC 74181?

6.21 Show how 7 can be subtracted from 13 using IC 74181.

Section 6.11

6.22 Describe a program that multiplies 9×7 using repeated addition.

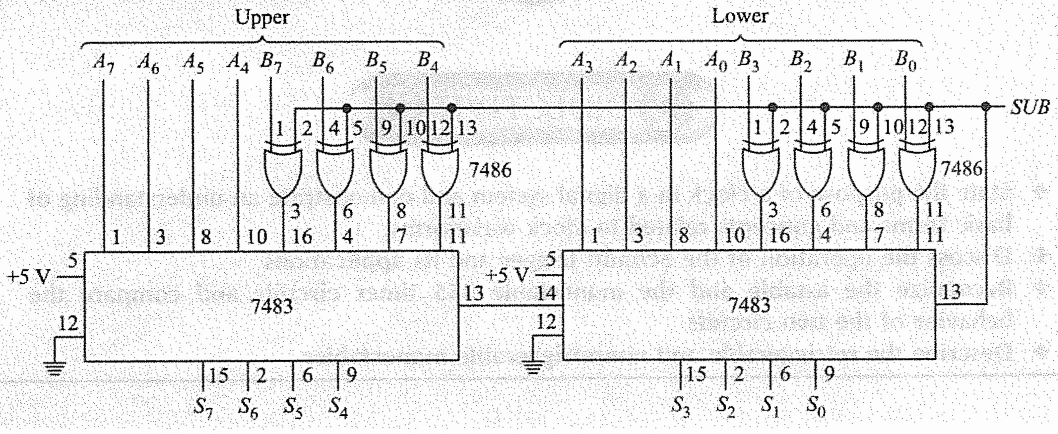
LABORATORY EXPERIMENT

AIM: The aim of this experiment is to perform addition and subtraction of two 8-bit data.

Theory: Two's complement arithmetic complements the number to be subtracted and adds one to it. Then this is added with the other number to perform subtraction. The addition is straight forward. IC 7483 is a 4-bit full adder with carry in input at pin 13. The exclusive-OR gate is useful to find complement of a binary number.

Apparatus: 5 VDC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of IC 7483. Connect two 7483 to perform 8-bit addition as shown. The exclusive-OR gate passes the same data to adder when *SUB* = 0 and a complement of the data when *SUB* = 1 where assertion of *SUB* stands for subtraction. Add and subtract five pair of numbers and compare with theoretical result.


Upper
Lower
A_7 A_6 A_5 A_4 B_7 B_6 B_5 B_4
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
SUB
7486
7483
+5 V
S_7 S_6 S_5 S_4
S_3 S_2 S_1 S_0

Answers to Self-tests

1. See Eqs. (6.1) through (6.4).
2. This is the hexadecimal number 179F.
3. Binary 111; decimal 111
4. See Eqs. (6.5) through (6.8).
5. It is used to indicate an overflow.
6. 255
7. –127 to +127
8. –13; +13
9. 0010 1001
10. 0010 1010
11. 2's complement
12. Take the complement of the subtrahend and add it to the minuend.
13. Inputs: *A* and *B*; outputs: SUM and CARRY
14. Inputs: *A*, *B*, and CARRY IN; outputs: SUM and CARRY
15. T; see Fig. 6.4.
16. Parallel adder requires 5 gate delays and serial adder ($2n + 2$) gate delays for n-bit addition.
17. Five.
18. ALU is short form of Arithmetic Logic Unit, a digital hardware that can perform both arithmetic and logic operations.
19. Substitute $M = 1$ and $S_3..S_0 = 0011$.
20. It realized only by repeatedly substracting one number from the other.

Clocks and Timing Circuits

7

OBJECTIVES

- ✦ State the purpose of a clock in a digital system and demonstrate an understanding of basic terms and concepts related to clock waveforms
- ✦ Discuss the operation of the Schmitt trigger and its applications
- ✦ Recognize the astable and the monostable 555 timer circuits and compare the behavior of the two circuits
- ✦ Describe the retriggerable and nonretriggerable monostables

The heart of every digital system is the system clock. The system clock provides the heartbeat without which the system would cease to function. In this chapter we consider the characteristics of a digital clock signal as well as some typical clock circuits. *Schmitt triggers* are used to produce nearly ideal digital signals from otherwise noisy or degraded signals. *Propagation delay* is the time required for a signal to pass from the input of a circuit to its output. You will see how to utilize logic gate propagation delay time to construct a pulse-forming circuit. A *monostable* is a basic digital timing circuit that is used in a wide variety of timing applications. We consider a number of different commercially available monostable circuits and examine some common applications.

7.1 CLOCK WAVEFORMS

Up to this point, we have been considering *static* digital logic levels, that is, voltage levels that do not change with time. However, all digital computer systems operate by "stepping through" a series of logical operations. The system signals are therefore changing with time: they are *dynamic*. The concept of a system *clock* was introduced in Chapter. 1. It is the clock signal that advances the system logic through its sequence of steps. The square wave shown in Fig. 7.1a is a typical clock waveform used in a digital system. It should be noted that the clock need not be the perfectly symmetrical waveform shown. It could simply be a series of positive (or negative) pulses as shown in Fig. 7.1b. This waveform could of course be considered an asymmetrical square wave with a duty cycle other than 50 percent. The main requirement is that the clock be perfectly periodic, and stable.

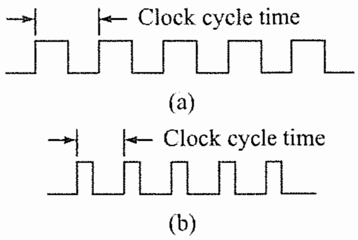


Fig. 7.1 **Ideal clock waveforms**

Notice that each signal in Fig. 7.1 defines a basic timing interval during which logic operations must be performed. This basic timing interval is defined as the *clock cycle time*, and it is equal to one period of the clock waveform. Thus all logic elements must complete their transitions in less than one clock cycle time.

Synchronous Operation

Nearly all of the circuits in a digital system (computer) change states in *synchronism* with the system clock. A change of state will either occur as the clock transitions from low to high or as it transitions from high to low. The low-to-high transition is frequently called the *positive transition* (*PT*), as shown in Fig. 7.2. The PT is given emphasis by drawing a small arrow on the *rising edge* of the clock waveform. A circuit that changes state at this time is said to be *positive-edge-triggered*. The high-to-low transition is called the *negative transition* (*NT*), as shown in Fig. 7.2. The NT is emphasized by drawing a small arrow on the *falling edge* of the clock waveform. A circuit that changes state at this time is said to be *negative-edge-triggered*. Virtually all circuits in a digital system are either positive-edge-triggered or negative-edge-triggered, and thus are synchronized with the system clock. There are a few exceptions. For instance, the operation of a push button (RESET) by a human operator might result in an instant change of state that is not in synchronism with the clock. This is called an *asynchronous operation*.

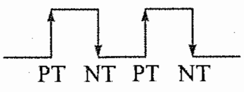


Fig. 7.2

Example 7.1 What is the clock cycle time for a system that uses a 500-kHz clock? An 8-MHz clock?

Solution The clock cycle is simply one period of the clock. For the 500-kHz clock,

$$\text{Cycle time} = \frac{1}{500 \times 10^3} = 2\ \mu\text{s}$$

For the 8-MHz clock,

$$\text{Cycle time} = \frac{1}{8 \times 10^6} = 125\ \text{ns}$$

Characteristics

The clock waveform drawn above the time line in Fig. 7.3a is a perfect, ideal clock. What exactly are the characteristics that make up an ideal clock? First, the clock levels must be absolutely stable. When the clock is high, the level must hold a steady value of +5 V, as shown between points *a* and *b* on the time line. When the clock is low, the level must be an unchanging 0 V, as it is between points *b* and *c*. In actual practice, the stability of the clock is much more important than the absolute value of the voltage level. For instance, it might be perfectly acceptable to have a high level of +4.8 V instead of +5.0 V, provided it is a *steady, unchanging*, +4.8 V.

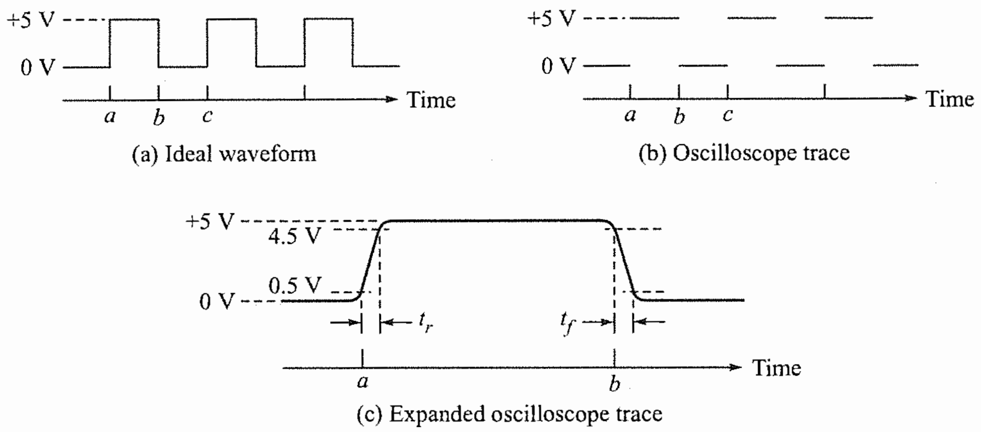


Fig. 7.3 Clock waveforms

The second characteristic deals with the time required for the clock levels to change from high to low or vice versa. The transition of the clock from low to high at point *a* in Fig. 7.3a is shown by a vertical line segment. This implies a time of zero; that is, the transition occurs instantaneously—it requires zero time. The same is true of the transition time from high to low at point *b* in Fig. 7.3a. Thus an ideal clock has zero transition time.

A nearly perfect clock waveform might appear on an oscilloscope trace as shown in Fig. 7.3b. At first glance this would seem to be two horizontal traces composed of line segments. On closer examination, however, it can be seen that the waveform is exactly like the ideal waveform in Fig. 7.3a if the vertical segments are removed. The vertical segments might not appear on the oscilloscope trace because the transition times are so small (nearly zero) and the oscilloscope is not capable of responding quickly enough. The vertical segments can usually be made visible by either increasing the oscilloscope "intensity," or by reducing the "sweep time."

Figure 7.3c shows a portion of the waveform in Fig. 7.3b expanded by reducing the "sweep time" such that the transition times are visible. Clearly it requires some time for the waveform to transition from low to high—this is defined as the rise time t_r. Remember, the time required for transition from high to low is defined as the fall time t_f. It is customary to measure the rise and fall times from points on the waveform referred to as the *10* and *90 percent* points. In this case, a 100 percent level change is 5.0 V, so 10 percent of this is 0.5 V and 90 percent is 4.5 V. Thus the rise time is that time required for the waveform to travel from 0.5 up to 4.5 V. Similarly, the fall time is that time required for the waveform to transition from 4.5 down to 0.5 V.

Finally, the third requirement that defines an ideal clock is its frequency stability. The frequency of the clock should be steady and unchanging over a specified period of time. Short-term stability can be specified by requiring that the clock frequency (or its period) not be allowed to vary by more than a given percentage over a short period of time—say, a few hours. Clock signals with short-term stability can be derived from straightforward electronic circuits as shown in the following sections.

Long-term stability deals with longer periods of time—perhaps days, months, or years. Clock signals that have long-term stability are generally derived from rather special circuits placed in a heated enclosure (usually called an "oven") in order to guarantee close control of temperature and hence frequency. Such circuits can provide clock frequencies having stabilities better than a few parts in 10^9 per day.

Propagation Delay Time

Propagation delay t_p is the time between a PT (or an NT) at the input of a digital circuit and the resulting change at the output. For all practical purposes, the time difference between fifty percent level of the input and corresponding output waveforms is used to calculate propagation delay. The box in Fig. 7.4 on the next page represents any TTL logic gate in the 74LSXX family. Notice that the waveform at the output is delayed in time from the input waveform, t_{pLH} is the delay time when the output is transitioning from low to high. t_{pHL} is the delay time when the output is transitioning from high to low. At temperatures below 75°C, t_{pHL} is only slightly larger than t_{pLH}. For the 74LSXX devices, we will simply assume they are equal, and for simplicity let's define propagation delay as

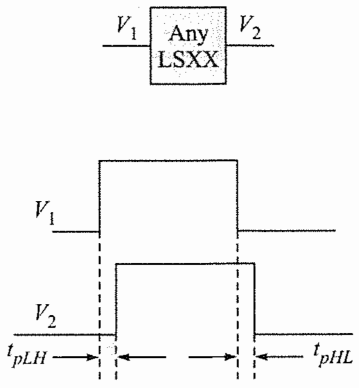


Fig. 7.4

$$\text{Propagation delay} \equiv t_p \approx t_{pLH} \approx t_{pHL}$$

The Texas Instruments data book gives a typical value of $t_p \approx 9$ ns for 74LSXX devices. For comparison, the high-speed CMOS has slightly longer delay times. For example, the 74HC04 inverter has $t_p = 24$ ns, which is typical.

Example 7.2 The total propogation delay through a 74HC04 inverter is known to be 24 ns. What is the maximum clock frequency that can be used with this device?

Solution An alternative way of posing the question is: How fast can the inverter operate? Remember, the circuit must complete any change of state within one clock cycle time. So,

$$\text{Clock cycle time} \geq t_p$$

The maximum clock frequency is then

$$\text{Frequency} = \frac{1}{t_p} = \frac{1}{24 \times 10^{-9}} = 41.7 \text{ MHz}$$

Pulse-Forming Circuits

It is sometimes necessary to use a series of narrow pulses in place of the rectangular clock waveform. Two such waveforms are shown in Fig. 7.5. The positive pulses occurring at the leading edge of the clock will define the PTs, while the negative pulses occurring at the falling edge will define the NTs. By taking advantage of the propagation delay time through a gate, it becomes a simple matter to change the rectangular clock into a series of pulses. There are numerous circuits that will change the clock into a pulse train, and here are two possibilities.

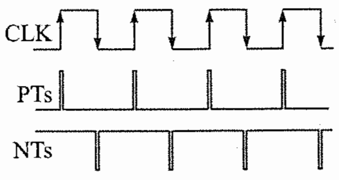


Fig. 7.5

In Fig. 7.6a, the clock (CLK) is applied to a NAND gate and an AND gate at the same time. The output of the NAND gate (A) is delayed by t_p. The output of the AND gate (PT) is high only when both its inputs are high. This is shown as the shaded region on the waveforms in Fig. 7.6b. The output (PT) is also delayed by t_p through the AND gate, and it appears as a positive pulse. Each output pulse (PT) is delayed by t_p from the leading edge of CLK, and each pulse has a width equal to t_p. Any digital circuit that incorporates the pulse-forming block in Fig 7.6a is said to be positive-edge-triggered, since it will change states in synchronism

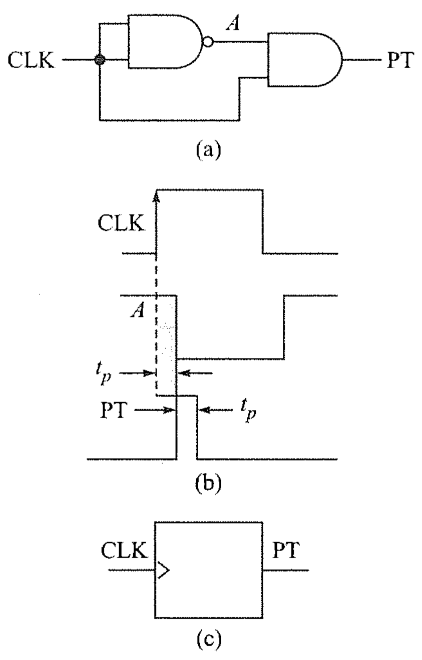


Fig. 7.6

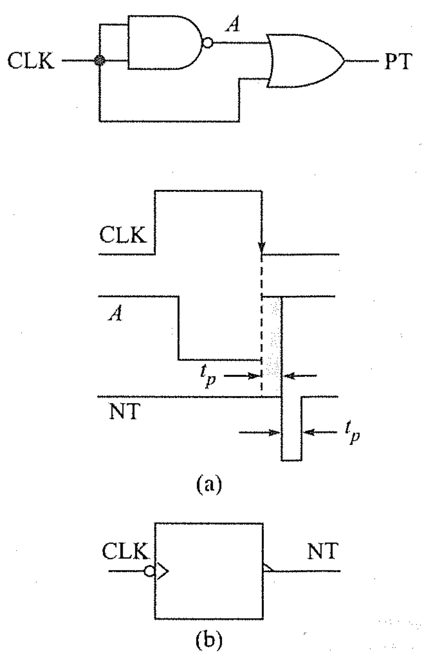


Fig. 7.7

with the PT of the clock. The box in Fig. 7.6c is a general symbol for a positive-edge-triggered circuit. The small triangle inside the box is called a *dynamic input indicator*, which simply means the circuit is sensitive to PTs.

In Fig. 7.7a, CLK is applied to a NAND gate and an OR gate simultaneously. The output of the NAND-gate (A) is delayed by t_p. The output of the OR gate (NT) is low only when both its inputs are low. This is the shaded region on the waveforms. The output (NT) is also delayed by t_p through the OR gate, and it appears as a negative pulse. Each output pulse has a width of t_p and each is delayed by t_p from the falling edge of CLK. Any digital circuit that incorporates this pulse-forming circuit is said to be negative-edge-triggered since it will change states in synchronism with the NT of the clock. The box in Fig. 7.7b is a general symbol for a negative-edge-triggered circuit. The small triangle is the dynamic input indicator, and the bubble shows that the input is active-low. The small triangle on the output indicates that NT is normally high, and is active when low. This triangle has the exact same meaning as the bubble. In fact, the IEEE standard uses these symbols interchangeably. You will see both symbols used in industry and on manufacturers' data sheets. Just remember, they both mean the same thing—active low!

It should be obvious that an inverter at the output of the AND gate in Fig. 7.6a will produce a series of negative pulses that synchronize with the leading edge of CLK. Similarly, an inverter at the output of the OR gate in Fig. 7.7a will produce a series of positive pulses in synchronization with the falling edge of CLK. These circuits, or variations of them, are used extensively with edge-triggered flip-flops—the subject of the next chapter. If you care to look ahead at the flip-flop symbols, you will see the dynamic indicator and the bubbled dynamic indicator used extensively.

SELF-TEST

1. Explain the meaning of positive-edge-triggered and negative-edge-triggered.
2. What is a dynamic input indicator?
3. What is the logic symbol for an input sensitive to NTs?

7.2 TTL CLOCK

A 7404 hexadecimal inverter can be used to construct an excellent TTL-compatible clock, as shown in Fig. 7.8. This clock circuit is well known and widely used. Two inverters are used to construct a two-stage amplifier with an overall phase shift of 360° between pins 1 and 6. Then a portion of the signal at pin 6 is fed back by means of a crystal to pin 1, and the circuit oscillates at a frequency determined by the crystal. Since the feedback element is a crystal, the frequency of oscillation is very stable. Here's how the oscillator works.

Inverter 1 has a 330-Ω feedback resistor (R_1) connected from output (pin 2) to input (pin 1). This forms a current-to-voltage amplifier with a gain of $A_1 = V_o / I_i = -R_1$. In this case, the gain is $A_1 = -330$ V/A, where the negative sign shows 180° of phase shift. For instance, an increase of 1 mA in I_i, will cause a negative-going voltage of 1 mA × 330 = 330 mV at V_o.

Inverter 2 is connected exactly as is inverter 1. Its gain is $A_2 = -R_2$. The two amplifiers are then ac-coupled with 0.01-μF capacitor to form an amplifier that has an overall gain of $A = A_1 \times A_2 = R_1R_2$. Notice that the overall gain has a positive sign, which shows 360° of phase shift. In this case, $A = 330 \times 330 = 1.09 \times 10^5$ V/A. For instance, an increase of 45 μA at I_i will result in a positive-going voltage of 5.0 V at pin 6 of inverter 2. Now, if a portion of the signal at pin 6 is fed back to pin 1, it will augment I_i (positive feedback) and the circuit will oscillate.

A series-mode crystal is used as the feedback element to return a portion of the signal at pin 6 to pin 1. The crystal acts as a series RLC circuit, and at resonance it ideally appears as a low-resistance element with no phase shift. The feedback signal must therefore be at resonance, and the two inverters in conjunction with the crystal form an oscillator operating at the crystal resonant frequency.

With the feedback element connected, the overall gain is sufficient to drive each inverter between saturation and cutoff, and the output signal is a periodic waveform as shown in Fig. 7.8. Typically, the output clock signal will transition between 0 and +5 V, will have rise and fall times of less than 10 ns, and will be essentially a square wave. The frequency of this clock signal determined by the crystal, and values between 1 and 20 MHz are common.

Inverter 3 is used as an output buffer amplifier and is capable of driving a load of 330 Ω in parallel with 100 pF while still providing rise and fall times of less than 10 ns.

Example 7.3 A TTL clock circuit as shown in Fig. 7.8 is said to provide a 5-MHz clock frequency with a stability better than 5 parts per million (ppm) over a 24-h time period. What are the frequency limits of the clock?

Solution A stability of 5 parts per million means that a 1-MHz clock will have a frequency of 1,000,000 plus or minus 5 Hz. So, this clock will have a frequency of 5,000,000 plus or minus 25 Hz. Over any 24-h period the clock frequency will be somewhere between 4,999,975 and 5,000,025 Hz.

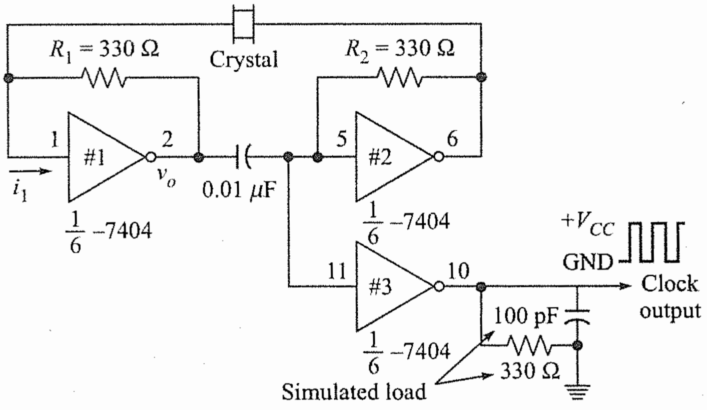


Fig. 7.8 TTL clock circuit

SELF-TEST

4. Why must the crystal in Fig. 7.8 be a series mode and not a parallel mode?
5. Are the 100-pF and 330-Ω loads necessary in Fig. 7.8?

7.3 SCHMITT TRIGGER

A *Schmitt trigger* is an electronic circuit that is used to detect whether a voltage has crossed over a given reference level. It has two stable states and is very useful as a signal-conditioning device. Given a sinusoidal waveform, a triangular wave, or any other periodic waveform, the Schmitt trigger will produce a rectangular output that has sharp leading and trailing edges. Such fast rise and fall times are desirable for all digital circuits.

Figure 7.9 shows the transfer function (V_o versus V_i) for any Schmitt trigger. The value of V_i that causes the output to jump from low to high is called the *positive-going threshold voltage* V_{T+}. The value of V_i causing the output to switch from high to low is called the *negative-going threshold voltage* V_{T-}.

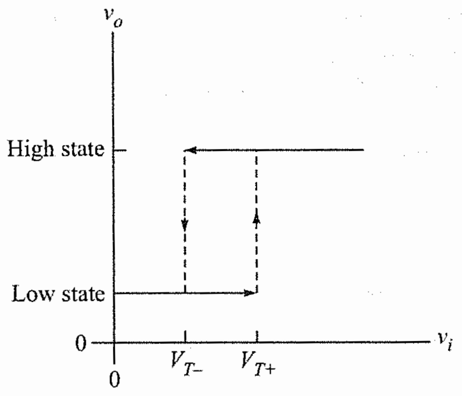


Fig. 7.9 Schmitt-trigger transfer characteristic

The output voltage is either high or low. When the output is low, it is necessary to raise the input to slightly more than V_{T+} to produce switching action. The output will then switch to the high state and remain there until the input is reduced to slightly below V_{T-}. The output will then switch back to the low state. The arrows and the dashed lines show the switching action.

The difference between the two threshold voltages is known as *hysteresis*. It is possible to eliminate hysteresis by circuit design, but a small amount of hysteresis is desirable because it ensures rapid switching action over a wide temperature range. Hysteresis can also be a very beneficial feature. For instance, it can be used to provide noise immunity in certain applications (digital modems for example).

The TTL 7414 is a hex Schmitt-trigger inverter. The *hex* means there are six Schmitt-trigger circuits in one DIP. In Fig. 7.10a, the standard logic symbol for one of the Schmitt-trigger inverters in a 7414 is shown along with a typical transfer characteristic. Because of the inversion, the characteristic curve is reversed from that shown in Fig. 7.9. Looking at the curve in Fig. 7.10b, when the input exceeds 1.7 V, the output will switch to the low state. When the input falls below 0.9 V, the output will switch back to the high state. The switching action is shown by the arrows and the dashed lines.

The TTL 74132 is a quad 2-input NAND gate that employs Schmitt-trigger with a similar hysteresis characteristics as described before for 7414. Figure 7.10c shows the standard logic symbol for one Schmitt-trigger NAND gate.

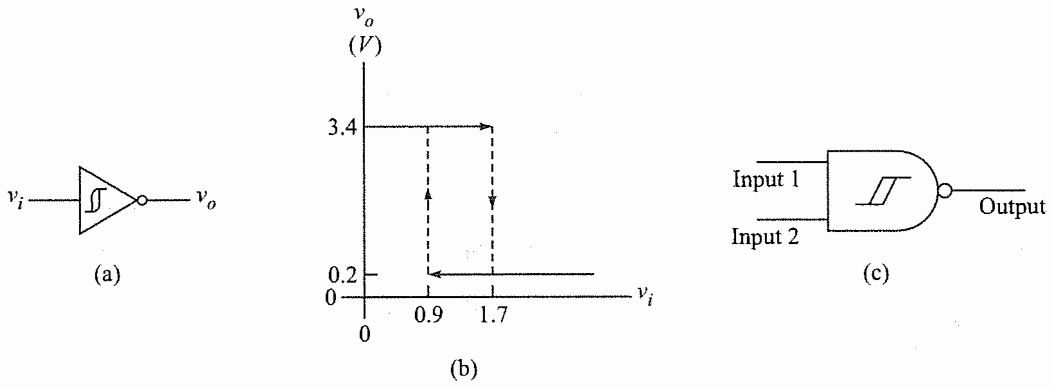


Fig. 7.10 **(a) Logic symbol of Schmitt-trigger inverter, (b) 7414 hysteresis characteristics, and (c) Logic symbol of Schmitt-trigger 2-input NAND gate**

Example 7.4 A sine wave with a peak of 2 V drives one of the inverters in a 7414. Sketch the output voltage.

Solution When the sinusoid exceeds 1.7 V, the output goes from high to low. The output stays in the low state until the input sinusoid drops below 0.9 V. Then the output jumps back to the high state. Figure 7.11 shows the input and output waveforms. This illustrates the signal-conditioning action of the Schmitt-trigger inverter. It has changed the sine wave into a rectangular pulse with fast rise and fall times. The same action would occur for any other periodic waveform.

Noisy Signals

The hysteresis characteristic of a Schmitt trigger is very useful in changing noisy signals, or signals with slow rise times, into more nearly ideal digital signals. A noisy signal is illustrated in Fig. 7.12a. Applying this signal to the input of a 7404 inverter will produce *multiple* pulses at its output, as shown in Fig. 7.12b. Each time the input signal crosses the threshold of the 7404, it will respond, and the multiple output transitions are the result. When used with an edge-triggered circuit, this will produce numerous unwanted PTs and NTs. The

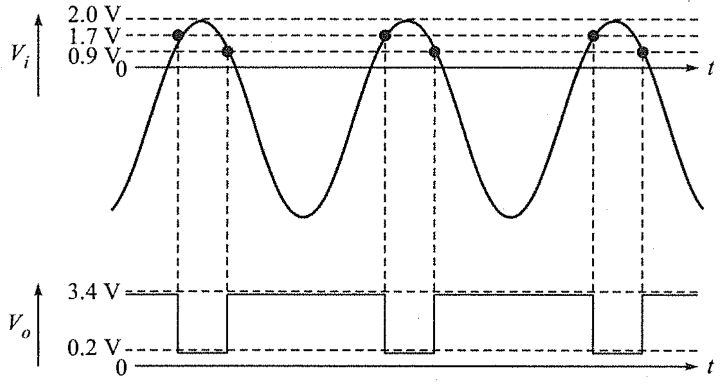


Fig. 7.11

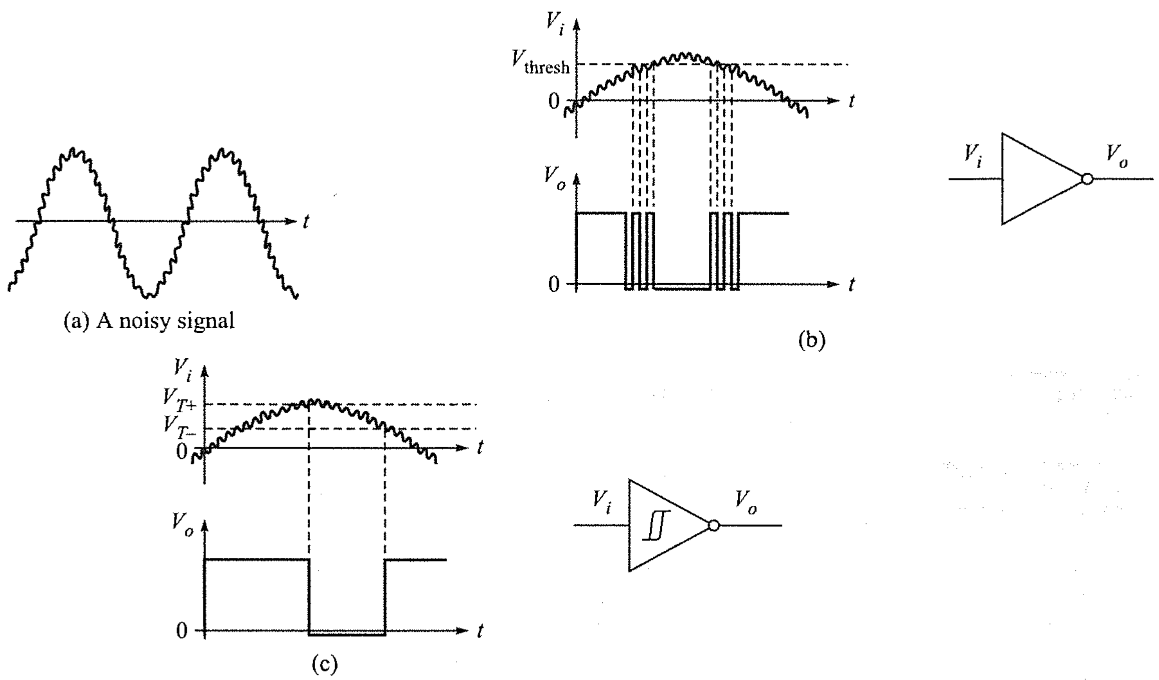


Fig. 7.12

Schmitt trigger will eliminate these multiple transitions, as shown in Fig. 7.12c. When the input rises above V_{T+}, the output will go low. However, the output will not again change state until the input falls below V_{T-}. Thus, multiple triggering is avoided! A Schmitt trigger is occasionally incorporated in an IC, for instance, the 74121, which is discussed in the next section.

SELF-TEST

6. What is the meaning of hysteresis when applied to a Schmitt trigger?
7. What is the difference between an inverting and a noninverting Schmitt trigger?
8. Schmitt triggers can be used as simple inverters. What is another good application for a Schmitt trigger?

7.4 555 TIMER—ASTABLE

The 555 timer is a TTL-compatible integrated circuit (IC) that can be used as an oscillator to provide a clock waveform. It is basically a switching circuit that has two distinct output levels. With the proper external components connected, neither of the output levels is stable. As a result, the circuit continuously switches back and forth between these two unstable states. In other words, the circuit oscillates and the output is a periodic, rectangular waveform. Since neither output state is stable, this circuit is said to be *astable* and is often referred to as a *free-running multivibrator* or *astable multivibrator*. The frequency of oscillation as well as the duty cycle are accurately controlled by two external resistors and a single timing capacitor. The internal circuit diagram of LM 555 timer is shown in Fig. 7.13(a). Note that the two comparators inside have two different reference voltages $V_{CC}/3$ and $2V_{CC}/3$ for comparisons, if $V_{CC}/3$ is the voltage between pin 1 and 8. Also note how they are connected to + and – input of the comparator. The Set Reset flip-flop sets or resets the output based on these comparator outputs in its usual operation. If required, it can be separately reset by asserting pin 4. More about this flip-flop will be discussed in next chapter. In this section, we show how 555 can be connected to get an astable multivibrator and in next section, we will discuss how it can be used in monostable mode.

The logic symbol for an LM555 timer connected as an oscillator is shown in Fig. 7.13. The timing capacitor C is charged toward $+V_{CC}$ through resistors R_A and R_B. The charging time t_1 is given as

$$t_1 = 0.693(R_A + R_B)C$$

This is the time during which the output is high as shown in Fig. 7.13.

The timing capacitor C is then discharged toward ground (GND) through the resistor R_B. The discharge time t_2 is given as

$$t_2 = 0.693R_BC$$

This is the time during which the output is low, as shown in Fig. 7.13.

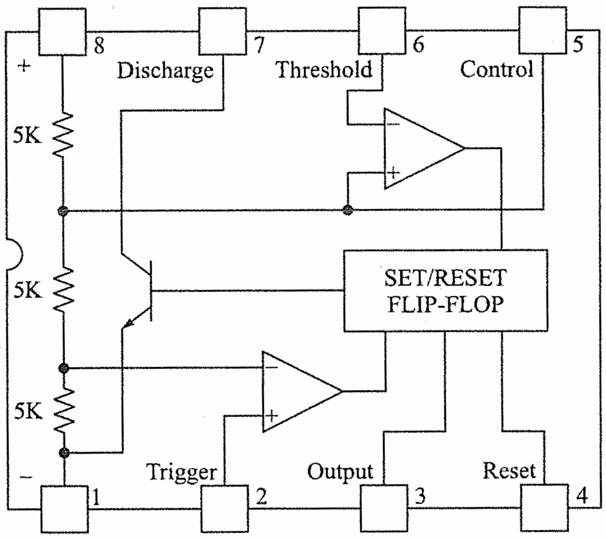
The period T of the resulting clock waveform is the sum of t_1 and t_2. Thus

$$T = t_1 + t_2 = 0.693(R_A + 2R_B)C$$

The frequency of oscillation is then found as

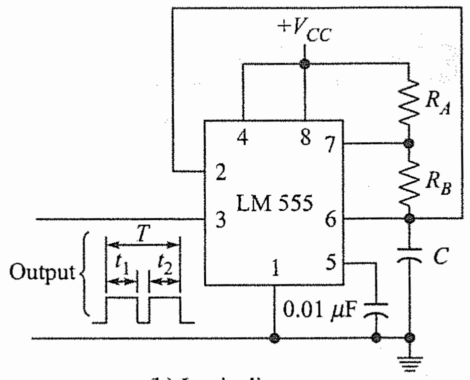
$$f = \frac{1}{T} = \frac{1.44}{(R_A + 2R_B)C}$$

Example 7.5 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = R_B = 1\ \text{k}\Omega$ and $C = 1000$ pF.

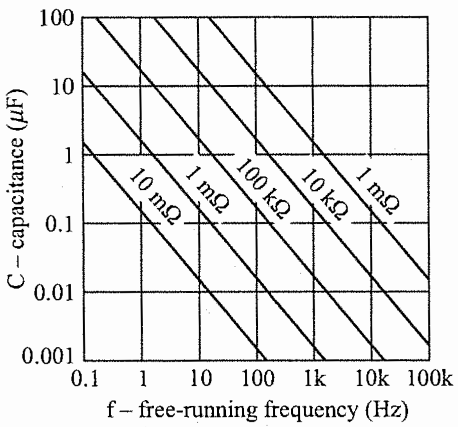


(a)

$$t_1 = 0.693\,(R_A + R_B)\,C$$
$$t_2 = 0.693\,R_BC$$
$$t = \frac{1.44}{(R_A + 2R_B)\,C}$$
$$Duty\ cycle = \frac{t_2}{t_1 + t_2} = \frac{R_B}{R_A + 2R_B}$$



(b) Logic diagram



(c) Nomograph

Fig. 7.13 **(a) Internal diagram of LM 555, (b) LM 555 in astable mode, (c) Nomograph**

Solution Using the relationship given above, we obtain

$$f = \frac{1.44}{[1000 + 2(1000)] \times 10^{-9}} = 480\ \text{kHz}$$

The output of the 555 timer when connected this way is a periodic rectangular waveform but not a square wave. This is because t_1 and t_2 are unequal, and the waveform is said to be asymmetrical. A measure of the asymmetry of the waveform can be stated in terms of its duty cycle. Here we define the duty cycle to be the ratio of t_2 to the period.

Thus

$$\text{Duty cycle} = \frac{t_2}{t_1 + t_2}$$

As defined, the duty cycle is always a number between 0.0 and 1.0 but is often expressed as a percent. For instance, if the duty cycle is 0.45 (or 45 percent), the signal is at GND level 45 percent of the time and at high level 55 percent of the time.

Example 7.6

(a) Given $R_B = 750\ \Omega$, determine values for R_A and C in Fig. 7.13 to provide a 1.0-MHz clock that has a duty cycle of 25 percent.

(b) What change in the circuit shown in Fig. 7.13 gives duty cycle approximately 50%?

Solution

(a) A 1-MHz clock has a period of 1 μs. A duty cycle of 25 percent requires $t_1 = 0.75\ \mu$s and $t_2 = 0.25\ \mu$s. Solving the expression

$$\text{Duty cycle} = \frac{R_B}{R_A + 2R_B}$$

for R_A yields

$$R_A = \frac{R_B}{\text{Duty cycle}} = -2R_B = \frac{750}{0.25} - 2 \times 750 = 1500\ \Omega$$

Solving $t_2 = 0.693R_BC$ for C yields

$$C = \frac{t_2}{0.693R_B} = \frac{0.25 \times 10^{-6}}{0.693 \times 750} = 480\ \text{pF}$$

(b) Connect a diode across R_B pointing from pin 7 to 6 so that it conducts while charging capacitor C and make $R_A = R_B$. Then while charging, R_B is bypassed as diode is forward biased but discharging is through R_B as diode remains reverse biased and does not conduct. Thus we get same charging and discharging current. Neglecting small voltage drop across forward biased diode we approximately gate 50% duty cycle.

The nomogram given in Fig. 7.13b can be used to estimate the free-running frequency to be achieved with various combinations of external resistors and timing capacitors. For example, the intersection of the resistance line 10 k$\Omega = (R_A + 2R_B)$ and the capacitance line 1.0 μF gives a free-running frequency of just over 100 Hz. It should be noted that there are definite constraints on timing component values and the frequency of oscillation, and you should consult the 555 data sheets.

SELF-TEST

9. What is an astable circuit?
10. A 555 timer can be connected to form an oscillator. (T or F)
11. The oscillation frequency in astable 555 is (directly, inversely) proportional to the external timing capacitor.

7.5 555 TIMER—MONOSTABLE

With only minimal changes in wiring, the 555 timer discussed in Sec. 7.4 can be changed from a free-running oscillator (astable) into a switching circuit having one stable state and one quasistable state. The resulting *monostable* circuit is widely used in industry for many different timing applications. The normal mode of operation is to trigger the circuit into its quasistable state, where it will remain for a predetermined length of time. The circuit will then switch itself back (regenerate) into its stable state, where it will remain until it receives another input trigger pulse. Since it has only one stable state, the circuit is characterized by the term *monostable multivibrator*, or simply *monostable*.

The standard logic symbol for a monostable is shown in Fig. 7.14a. The input is labeled *TRIGGER*, and the output is Q. The complement of the Q output may also be available at $\overline{Q}$. The input trigger circuit may be sensitive to either a PT or an NT. In this case, it is negative-edge-triggered. Usually the output at Q is low when the circuit is in its stable state.

A typical set of waveforms showing the proper operation of a monostable circuit is shown in Fig. 7.14b. In this case, the circuit is sensitive to an NT at the trigger input, and the output is low when the circuit rests in its stable state. Once triggered, Q goes high and remains high for a predetermined time t and then switches back to its stable state until another NT appears at the trigger input.

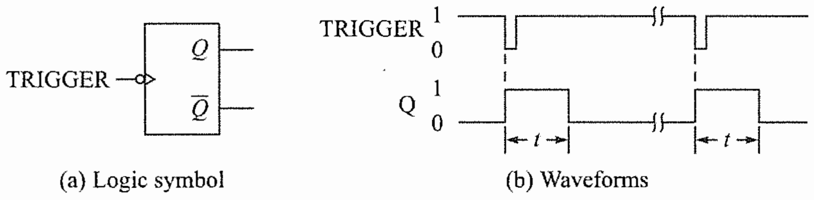


Fig. 7.14 **Monostable circuit**

A 555 timer wired as a monostable switching circuit (sometimes called a *one-shot*) is shown in Fig. 7.15 on the next page. In its stable state, the timing capacitor C is completely discharged by means of an internal transistor connected to C at pin 7. In this mode, the output voltage at pin 3 is at ground potential.

A negative pulse at the trigger input (pin 2) will cause the circuit to switch to its quasistable state. The output at pin 3 will go high and the discharge transistor at pin 7 will turn off, thus allowing the timing capacitor to begin charging toward V_{CC}.

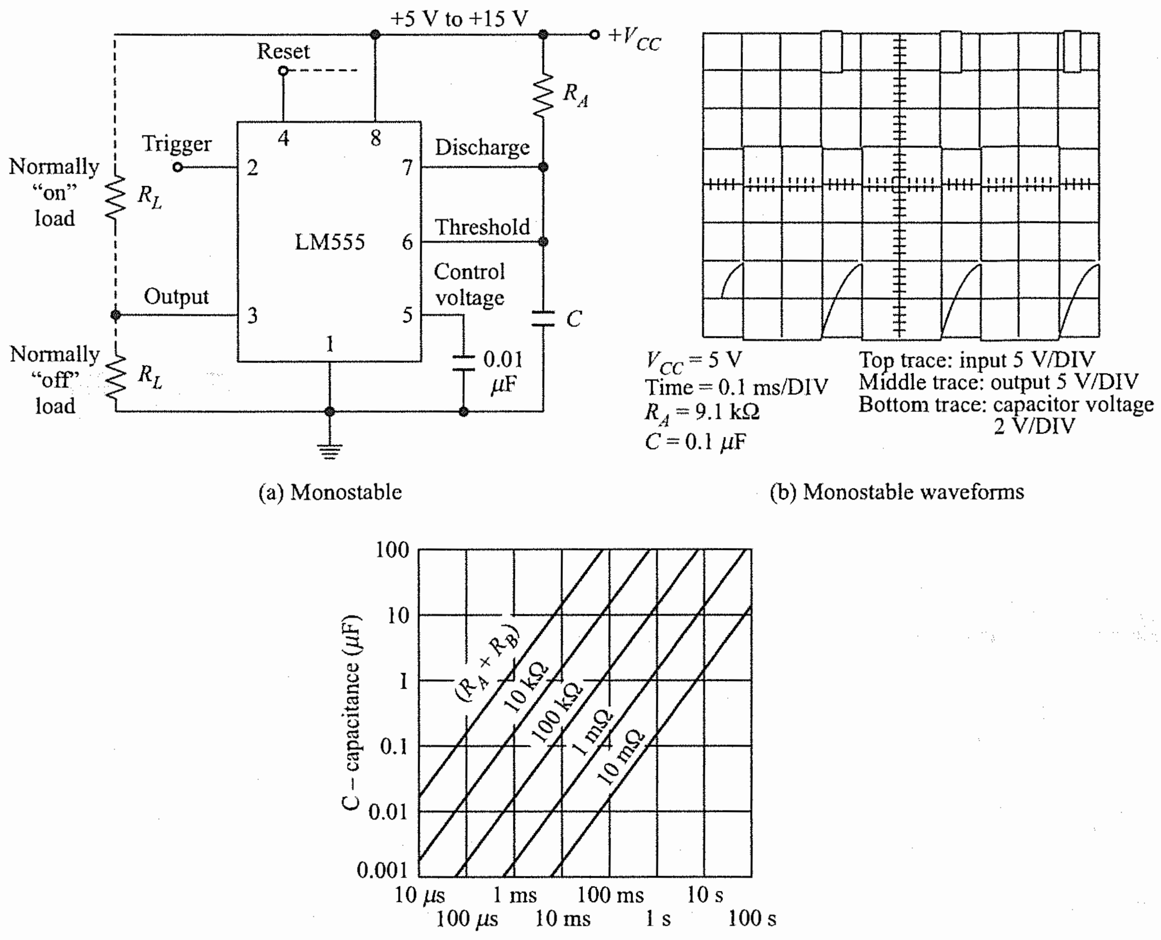
When the voltage across C reaches $^2/_3\ V_{CC}$, the circuit will regenerate back to its stable state. The discharge transistor will again turn on and discharge C to GND, the output will go back to GND, and the circuit will remain in this state until another pulse arrives at the trigger input. A typical set of waveforms is shown in Fig. 7.15b.

The output of the monostable can be considered a positive pulse with a width

$$t = 1.1\ R_A C$$

Take care to note that the input voltage at the trigger input must be held at $+V_{CC}$, and that a negative pulse should then be applied when it is desired to trigger the circuit into its quasistable or timing mode.

Example 7.7 Find the output pulse width for the timer in Fig. 7.15 given $R_A = 10\ \text{k}\Omega$ and $C = 0.1\ \mu\text{F}$.



(a) Monostable

(b) Monostable waveforms

(c) Time delay, $t = 1.1\, R_A C$

Fig. 7.15 **LM555 connected as a monostable circuit**

Solution The pulse width is found as

$$t = 1.1(R_A C) = 1.1(10^4 \times 10^{-7}) = 1.1 \text{ ms}$$

Example 7.8 Find the value of C necessary to change the pulse width in Example 7.7 to 10 ms.

Solution The timing equation can be solved for C as

$$C = \frac{t}{1.1 R_A} = \frac{10^{-2}}{1.1 \times 10^4}$$

$$= 0.909\ \mu\text{F}$$

The nomograph shown in Fig. 7.15c can be used to obtain a quick, if not very accurate, idea of the sizes of R_A or C required for various pulse-width times. You can quickly check the validity of the results of Example 7.8 by following the $R_A = 10$ kΩ line up to the $C = 0.1$ μF line and noting that pulse-width time is about 1 ms.

Once the circuit is switched into its quasistable state (the output is high), the circuit is immune to any other signals at its trigger input. That is, the timing cannot be interrupted and the circuit is said to be *nonretriggerable*. However, the timing can be interrupted by the application of a negative signal at the reset input on pin 4. A voltage level going from $+V_{CC}$ to GND at the reset input will cause the timer to immediately switch back to its stable state with the output low. As a matter of practicality, if the reset function is not used, pin 4 should be tied to $+V_{CC}$ to prevent any possibility of false triggering.

SELF-TEST

12. What is a monostable?
13. A 555 timer can be connected as a one-shot. (T or F)
14. Is the stable output state of a 555 timer connected in a monostable mode high or low?

7.6 MONOSTABLES WITH INPUT LOGIC

The basic monostable circuit discussed in the previous section provides an output pulse of predetermined width in response to an input trigger. Logic gates have been added to the inputs of a number of commercially available monostable circuits to facilitate the use of these circuits as general-purpose delay elements. The 74121 *nonretriggerable* and the 74123 *retriggerable monostables* are two such widely used circuits.

The logic inputs on either of these circuits can be used to allow triggering of the device on either a high-to-low transition (NT) or on a low-to-high transition (PT). Whenever the value of the input logic equation changes from false to true, the circuit will trigger. Take care to note that a transition from false to true must occur, and simply holding the input logic equation in the true state will have no effect.

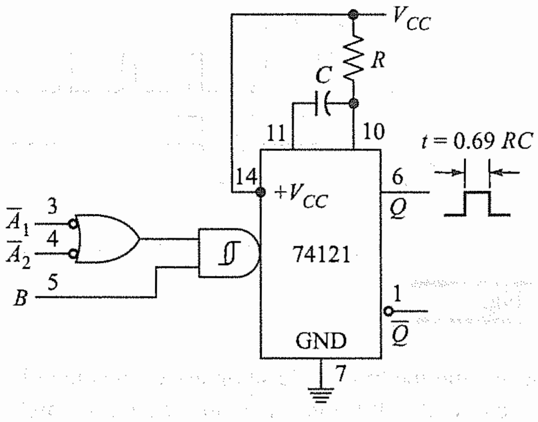
The logic diagram, truth table, and typical waveforms for a 74121 are given in Fig. 7.16. The inputs to the 74121 are $\overline{A}_1$, $\overline{A}_2$, and B. The trigger input to the monostable appears at the output of the AND gate. Here's how the gates work:

1. If B is held high, an NT at either $\overline{A}_1$ or $\overline{A}_2$ will trigger the circuit (see Fig. 7.16c). This corresponds to the bottom two entries in the truth table.
2. If either $\overline{A}_1$ or $\overline{A}_2$, or both are held low, a PT at B will trigger the circuit (see Fig. 7.16d).

This corresponds to the top two entries in the truth table. A logic equation for the trigger input can be written as

$$T = (A_1 + A_2)B\overline{Q}$$

Note that for T to be true (high), either A_1 or A_2 must be true—that is, either $\overline{A}_1$ or $\overline{A}_2$ at the gate input must be low. Also, since $\overline{Q}$ is low during the timing cycle (in the quasistable state), it is not possible for a transition to occur at T during this time. The logic equation for T must be low if $\overline{Q}$ is low. In other words, once the monostable has been triggered into its quasistable state, it must time out and switch back to its stable state before it can be triggered again. This circuit is thus nonretriggerable.



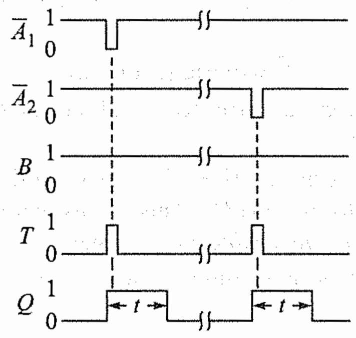
(a) Logic diagram

$\overline{A}_1$	$\overline{A}_2$	B	Result
L	X	↑	Trigger
X	L	↑	Trigger
↓	H	H	Trigger
H	↓	H	Trigger

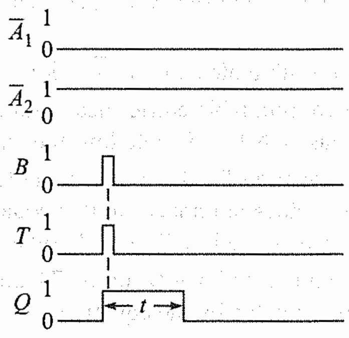
Note: Triggering can occur only when $\overline{Q}$ is H (not in timing cycle)

L = Low
H = High
X = Don't care
↑ = Low to high transition
↓ = High to low transition

(b) Truth table



(c) Negative triggering



(d) Positive triggering

Fig. 7.16 **74121 nonretriggerable monostable**

The output pulse width at Q is set according to the values of the timing resistor R and capacitor C as

$$t = 0.69RC$$

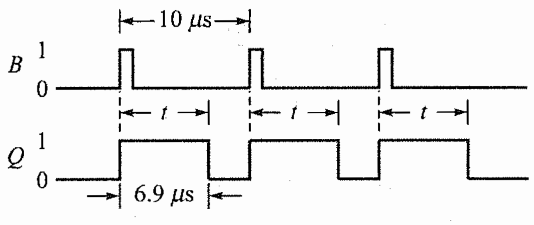
For instance, if $C = 1\ \mu\text{F}$ and $R = 10\ \text{k}\Omega$, the output pulse width will be $t = 0.69 \times 10^4 \times 10^{-6} = 6.9$ ms.

Example 7.9 The 74121 monostable in Fig. 7.16 is connected with $R = 1\ \text{k}\Omega$ and $C = 10{,}000$ pF. Pins 3 and 4 are tied to GND and a series of positive pulses are applied to pin 5. Describe the expected waveform at pin 6, assuming that the input pulses are spaced by (a) 10 μs and (b) 5 μs.

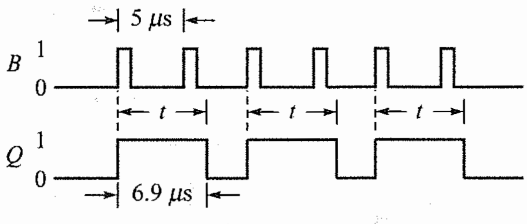
Solution The circuit is connected such that positive pulses applied to pin 5 will trigger it. The output pulse width at pin 6 will be $t = 0.69 \times 10^3 \times 10^{-3} = 6.9\ \mu$s.

(a) The monostable will trigger and time out for every input pulse appearing at B, as shown in Fig. 7.17a.
(b) Since the monostable is *not* retriggerable, it will trigger once and time out for every other input pulse as shown in Fig. 7.17b.

The logic diagram and truth table for a 74123 retriggerable monostable are given in Fig. 7.18. There are actually two circuits in each 16-pin DIP, and the pin numbers are given for one of them. The input logic is



(a) Triggers on every pulse at B



(b) Triggers on every other pulse at B

Fig. 7.17

simpler than for the 74121. The inputs are $\overline{A}$, B, and $\overline{R}$, and the truth table summarizes the operation of the circuit. The first entry in the truth table shows that the circuit will trigger if $\overline{R}$ and B are both high, and an NT occurs at $\overline{A}$.

The second truth table entry states the circuit will trigger if $\overline{A}$ is held low, $\overline{R}$ is held high, and a PT occurs at B.

In the third truth table entry, if $\overline{A}$ is low and B is high, a PT at $\overline{R}$ will trigger the circuit.

The last two truth table entries deal with direct reset of the circuit. Irrespective of the values of $\overline{A}$ or B, if the $\overline{R}$ input has an NT, or is held low, the circuit will immediately reset.

The logic equation for the trigger input to the monostable can be written $T = AB\overline{R}$. Notice that the state of the output Q does not appear in this equation (as it does for the 74121). This means that this circuit will trigger *every time* there is a PT at T. In other words, this is a *retriggerable* monostable!

The output pulse width at Q for the 74123 is set by the values of the timing resistor R and the capacitor C. It can be approximated by the equation

$$t = 0.33RC$$

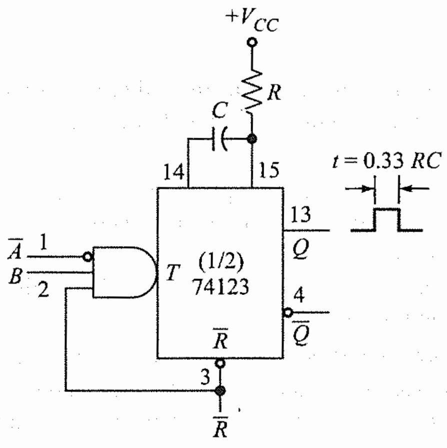
The waveforms in Fig. 7.19c show a series of negative pulses used to trigger the 74123. Notice carefully that the circuit triggers (Q goes high) at the first high-to-low transition on $\overline{A}$, but that the next two negative pulses on $\overline{A}$ retrigger the circuit and the timing cycle t does not begin until the very last trigger!

Example 7.10 The 74123 in Fig. 7.18 is connected with $\overline{A}$ at GND, $\overline{R}$ at $+V_{CC}$, R = 10 kΩ, and C = 10,000 pF. Describe the expected waveform at Q, assuming that a series of positive pulses are applied at B and the pulses are spaced at (a) 50 μs and (b) 10 μs.

Solution The output pulse width will be about

$$t = 0.33 \times 10^4 \times 10^{-8} = 33\ \mu s$$

a. The circuit will trigger and time out with every pulse as shown in Fig. 7.19a.
b. The circuit will trigger with the first pulse and then retrigger with every following pulse. The timing cycle will be reset with every input pulse, and Q will simply remain high since the circuit will never be allowed to time out (see Fig 7.19b). If the pulses at B are stopped, Q will be allowed to time out and will go low 33 μs after the last pulse at B.

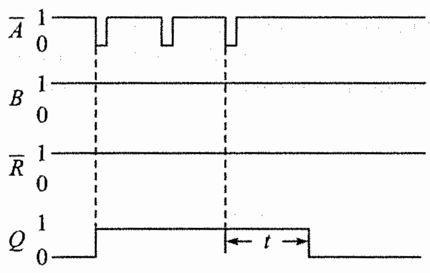


(a) Logic diagram

$\overline{A}$	B	$\overline{R}$	Q
↑	H	H	Trigger
L	↑	H	Trigger
L	H	↑	Trigger
X	X	L	Reset
X	X	↓	Reset

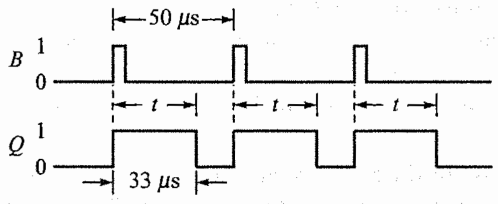
H = High
L = Low
X = Don't care
↑ = Low to high transition
↓ = High to low transition

(b) Truth table

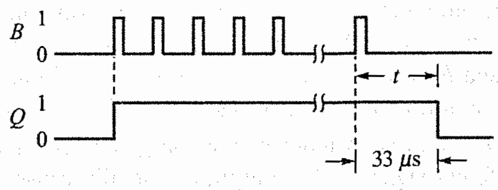


(c) Waveforms

Fig. 7.18 74123



(a)



(b)

Fig. 7.19

SELF-TEST

15. The 74121 is a (retriggerable, nonretriggerable) monostable.
16. The input logic used with a 74121 utilizes a Schmitt trigger. (T or F)
17. The output pulse width of a 74121 is RC multiplied by ________.

7.7 PULSE-FORMING CIRCUITS

The monostable circuits discussed in the previous sections have pulse-width times that are predictable to around 10 percent. As such, they do not represent precise timing circuits, but they do offer good short-term stability and are useful in numerous timing applications.

One such application involves the production of a pulse that occurs after a given event with a predictable time delay. For instance, suppose that you are required to generate a 1-ms pulse exactly 2 ms after the operation of a push-button switch. Look at the waveforms in Fig. 7.20b. If the operation of the switch occurs when the waveform labeled *SWITCH* goes high, the desired pulse is shown as OUTPUT. In this case, the delay time t_1 will be set to 2 ms, and the time of the pulse width t_2 will be 1 ms.

The two monostables in the 74123 shown in Fig. 7.20a are connected to provide a delayed pulse. The first circuit provides the delay time as $t_1 = 0.33R_1 \times C_1$, while the second circuit provides the output pulse width as $t_2 = 0.33R_2 \times C_2$. The PT at the INPUT triggers the first circuit into its quasistable state, and its output at $\overline{Q_1}$ goes low. After timing out t_1, $\overline{Q_1}$ goes high, and this transition triggers the second circuit into its quasistable state. The OUTPUT thus goes high until the second circuit times out t_2, and then it returns low.

Example 7.11 The input to the circuit in Fig. 7.20a is changed to a 100-kHz square wave. It is desired to produce a 1-μs pulse 2 μs after every positive transition of the input as shown in Fig. 7.21. Find the proper timing capacitor values, given that both timing resistors are set at 500 Ω.

Solution The capacitor value for the pulse width is found using $t = 0.33\ RC$. Thus:

$$C = \frac{10^{-6}}{0.33 \times 500} = 6000\ \text{pF}$$

The pulse delay capacitor is twice this value, or 0.012 μF.

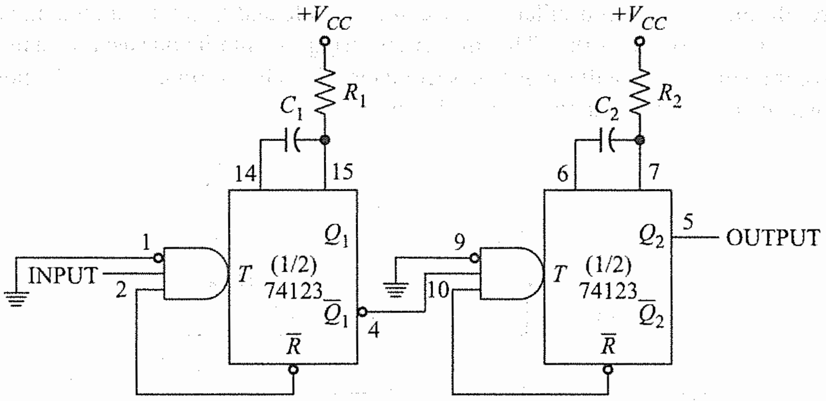
Glitches

Whenever two or more signals at the inputs of a gate are undergoing changes at the same time, an undesired signal may appear at the gate output—this undesired signal is called a *glitch*. For example, in Fig. 7.22a, the gate output at X should be low except during the time when $A = B = C = 1$ as shown. However, there is the possibility of a glitch appearing at the output at two different times. At time T_1, if C happens to go high before A and B go low, a narrow positive spike will appear at the gate output—a glitch! Similarly, a glitch could occur at time T_2 if B happens to go high before A goes low.

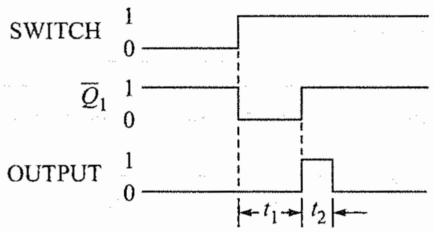
A glitch is an unwanted signal generated usually because of different propagation delay times through different signal paths, and they generally cause random errors to occur in a digital system. They are to be avoided at all costs, and a logic circuit designer must take them into account. One method of avoiding glitches in the instance shown in Fig. 7.22a is to use a strobe pulse.

It is a simple matter to use a pulse delay circuit such as the one shown in Fig. 7.20 to generate a strobe pulse. Consider using the waveform A in Fig. 7.22a as the input to the pulse delay circuit, and set the monostable times to generate a strobe pulse at the midpoint of the positive half cycle of A, as shown in Fig. 7.22b. If the inputs to the AND gate are now A, B, C, and the strobe pulse, the output will occur only when $A = B = C = 1$, and a strobe pulse occurs. The glitches are completely eliminated!

An interesting variation of the pulse delay circuit in Fig. 7.20 is shown in Fig. 7.23a. Here, we have simply connected the $\overline{Q}$ output of the second circuit back to the input of the first circuit. This is a form of positive



(a) A 74123 with DIP pin numbers



(b) Delayed pulse at *OUTPUT*

Fig. 7.20 **Delayed pulse generator**

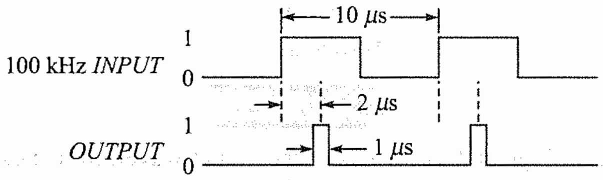
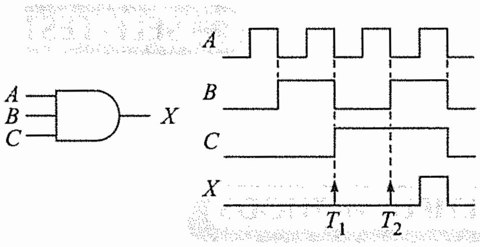
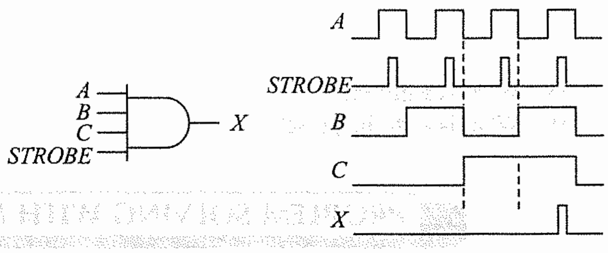


Fig. 7.21



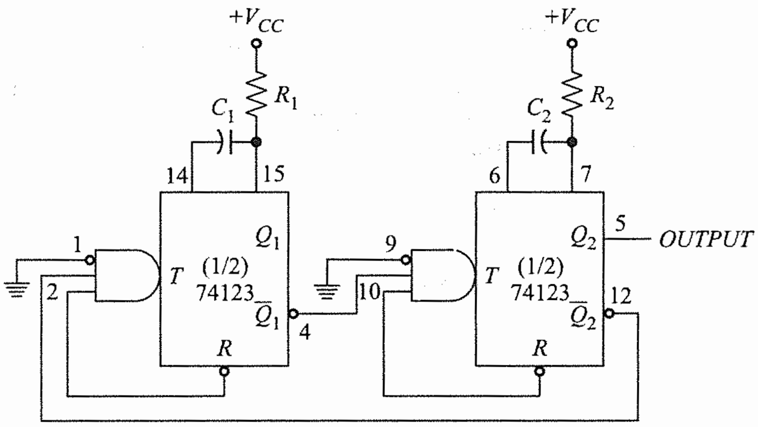
(a) Glitches at T_1 and T_2



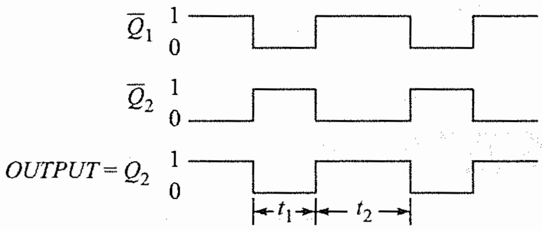
(b) Use of *STROBE* to remove glitches

Fig. 7.22

feedback. As a result, the circuit will oscillate—it becomes astable and generates a rectangular waveform as shown in Fig. 7.23. Here's how it works. The first circuit triggers into its quasistable state. When it times out t_1, the positive transition at $\overline{Q}_1$ will trigger the second circuit. When it times out t_2, the positive transition at $\overline{Q}_2$ will retrigger the first circuit and the cycle will repeat.



(a) Two monostable circuits connected to form an astable free-running oscillator



(b) Waveforms

Fig. 7.23

Independent adjustment of high and low levels of the output waveform is possible by setting the delay times of each individual monostable. Take care to note that since each circuit is edge-triggered, if a transition is missed by either circuit, oscillation will cease!

SELF-TEST

18. What is a glitch?
19. What is a strobe pulse?

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Design a 100 kHz pulse generator with 40 percent duty cycle.

Solution We can use 555 timer working in astable mode to generate this. Also, we can use monostable circuits 74121 or 74123 and positive feedback for this.

$$\text{Time period, } T = \frac{1}{10^5} \text{ sec.} = 10\ \mu\text{s}$$

If t_L are t_H are the times within T, during which pulse remain LOW and HIGH respectively

$$\text{Duty cycle} = \frac{t_L}{t_L + t_H} = \frac{t_L}{T} = 0.4$$

Thus, $t_L = 0.4 \times 10 = 4\ \mu\text{s}$

$t_H = T - t_L = 10 - 4 = 6\ \mu\text{s}$

In Method-1, we show the calculation required for 555 based pulse generator that uses a circuit as shown in Fig.7.13a.

$$t_L = 0.693\ R_B C$$

$$t_H = 0.693\ (R_A + R_B)C$$

Taking ratio, $\dfrac{t_H}{t_L} = \dfrac{R_A + R_B}{R_B} = 1 + \dfrac{R_A}{R_B} = \dfrac{6}{4}$ Thus, $R_B = 2R_A$

If we choose, $R_A = 1000\ \Omega$ then $R_B = 2000\ \Omega$

Substituting this in say, t_L calculation: $4 \times 10^{-6} = 0.693 \times 2000 \times C$

or, $C = 2.9$ nF

In Method-2, we show the calculation required for 74123 based pulse generator that uses a circuit as shown in Fig.7.23a.

$$t_L = 0.33\ R_1 C_1 = 4\ \mu\text{s}$$

$$t_H = 0.33\ R_2 C_2 = 6\ \mu\text{s}$$

Select say $C_1 = C_2 = C = 1$ nF. Then, $R_1 = 12$ kΩ and $R_2 = 18$ kΩ

In Method-3, we show the calculation required for 74121 based pulse generator that uses a circuit similar to Fig.7.23a where retriggerable 74123 is replaced by non-retriggerable 74121. From Section 7.6,

$$t_L = 0.69\ R_1 C_1 = 4\ \mu\text{s}$$

$$t_H = 0.69\ R_2 C_2 = 6\ \mu\text{s}$$

Select say $C_1 = C_2 = C = 1$ nF. Then $R_1 = 5.8$ kΩ and $R_2 = 8.7$ kΩ

SUMMARY

A system clock signal is a periodic waveform (usually a square wave) that has stable high and low levels, very short rise and fall times, and good frequency stability. A circuit widely used to generate a good, stable, TTL-compatible clock waveform is the crystal-controlled circuit shown in Fig. 7.8.

A Schmitt trigger is a switching circuit having two input threshold voltage levels. It exhibits hysteresis, and is useful in cleaning up noisy signals.

The 555 timer is a digital timing circuit that can be connected as either a monostable or an astable circuit. It is widely used in a number of different applications. The 74121 and 74123 monostable circuits both have logic circuits at their inputs that increase the number of possible applications.

A pulse delay circuit, and a free-running astable with adjustable duty cycle are only a few of the many circuits that can be constructed with the use of basic monostable circuits.

GLOSSARY

- ***astable*** Having two output states, neither of which is stable.
- ***asynchronous*** Referring to random events, not coordinated closely with a system clock.
- ***clock*** A periodic waveform (usually a square wave) that is used as a synchronizing signal in a digital system.
- ***clock cycle time*** The time period of a clock signal.
- ***clock stability*** A measure of the frequency stability of a waveform; usually given in parts per million (ppm).
- ***contact bounce*** Opening and closing of a set of contacts as a result of the mechanical bounce that occurs when the device is switched.
- ***dynamic input indicator*** A small triangle used on an input signal line to indicate sensitivity to signal transitions—edge triggering.
- ***fall time*** The time required for a signal to transition from 90 percent of its maximum value down to 10 percent of its maximum.
- ***glitch*** Very narrow positive or negative pulse that appears as an unwanted signal.
- ***monostable*** A circuit that has two output states, only one of which is stable.
- ***NT*** Negative transition.
- ***negative-edge trigger*** An input sensitive to high-to-low signal transitions.
- ***one-shot*** Another term for a monostable circuit.
- ***PT*** Positive transition.
- ***positive-edge trigger*** An input sensitive to low-to-high signal transitions.
- ***propagation delay time*** The time required for a signal to propagate through a circuit, input to output.
- ***rise time*** The time required for a signal to transition from 10 percent of its maximum value up to 90 percent of its maximum.
- ***Schmitt trigger*** A bistable circuit used to produce a rectangular output waveform.
- ***TTL clock*** A circuit that generates a clock waveform that is compatible with standard TTL logic circuits.
- ***10 percent point*** A point on a rising or falling waveform that is equal to 0.1 times its highest value.
- ***90 percent point*** A point on a rising or falling waveform that is equal to 0.9 times its highest value.
- ***555 timer*** A digital timing circuit that can be connected as either an astable or a monostable circuit.

PROBLEMS

Section 7.1

7.1 Calculate the clock cycle time for a system that uses a clock that has a frequency of:

a. 10 MHz b. 6 MHz
c. 750 kHz

7.2 What is the clock frequency if the clock cycle time is 250 ns?

7.3 What is the maximum clock frequency that can be used with a logic gate having a propagation delay of 75 ns?

7.4 You are selecting logic gates that will be used in a system that has a clock frequency of 8 MHz. What is the maximum allowable propagation delay?

7.5 What would be the 10 and 90 percent points on the waveform in Fig. 7.3c if the amplitude goes from 0 to +4.5 V?

Section 7.2

7.6 Find the upper and lower frequency limits of a 5-MHz clock signal that has a stability of 10 ppm.

7.7 A TTL clock uses a series-mode crystal having a resonant frequency of 3.5 MHz. The circuit provides a 24-h stability of 8 ppm. Calculate the oscillator frequency limits.

7.8 The TTL clock shown in Fig. 7.8 uses a crystal that has frequency of 7.5 MHz. Draw the clock output waveform if $+V_{CC}$ is set at +5 V. What is the stability in ppm if the upper limit on the clock frequency is 7,499,900 Hz?

7.9 The NAND gate in Example 7.2 has a propagation delay of 50 ns, and A is a 15-MHz clock. Make a careful sketch of the waveform at the Y output. Assume that B is always high. (***Hint***: Be sure to consider the propagation delay time.)

Section 7.3

7.10 Draw the input and output waveforms for the Schmitt trigger in Fig. 7.10, assuming that the input voltage is $V = 3.0 \cos 1000t$.

7.11 Draw carefully the waveforms at points A, B, and C in Fig. 7.24.

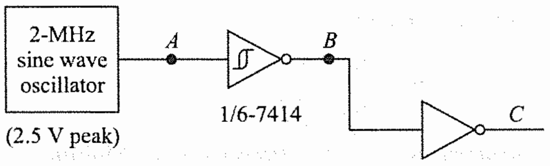


Fig. 7.24

7.12 Draw the transfer curve for a Schmitt trigger if $V_{T+} = +1.0$ V, $V_{T-} = -1.0$ V, high state = +5 Vdc, and low state = 0 Vdc.

7.13 Draw the output voltage for the Schmitt trigger in Prob. 7.12 if $V_i = 2 \sin \omega t$ V.

Section 7.4

7.14 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = R_B = 47$ kΩ and $C = 1000$ pF. Calculate the values of t_1 and t_2, and carefully sketch the output waveform.

7.15 Determine the frequency of oscillation for the 555 timer in Fig. 7.13, given $R_A = 5000\ \Omega$, $R_B = 7500\ \Omega$, and $C = 1500$ pF. Calculate values for t_1 and t_2, and carefully sketch the output waveform.

7.16 Use the nomogram in Fig. 7.13b to find $(R_A + 2R_B)$, given $C = 0.1\ \mu$F and that the desired frequency is 1 kHz. Check the results by using the formula given for the frequency.

7.17 Calculate the duty cycle for the circuit in Prob. 7.13. For Prob. 7.14.

7.18 Derive the expression

$$\text{Duty cycle} = R_B/(R_A + 2R_B)$$

7.19 It is desired to have a duty cycle of 25 percent for the circuit in Prob. 7.15. Find the correct values for the two resistors.

Section 7.5

7.20 Calculate the output pulse width for the timer in Fig. 7.15 for a 4,7-kΩ resistor and a 1.5-μF capacitor.

7.21 Calculate the output pulse width for the circuit in Problem 7.20, assuming that the resistor is halved.

7.22 Calculate the output pulse width for the circuit in Problem 7.20, assuming that the capacitor is doubled.

7.23 Find the capacitor value necessary to generate a 15-ms pulse width for the monostable in Fig. 7.15, given $R_A = 100$ kΩ.

7.24 A 500-Hz square wave is used as the trigger input for the circuit described in Example 7.7.

Make a careful sketch of the input and output waveforms (similar to those in Fig. 7.15b).

7.25 Repeat Prob. 7.24, assuming that the trigger input is changed to a 1-kHz square wave.

Section 7.6

7.26 In the 74121 in Fig. 7.16, $R = 47$ kΩ and $C =$ 10,000 pF. Calculate the output pulse width.

7.27 Redraw the 74121 logic diagram in Fig. 7.16a and show how to connect the circuit such that it will trigger on the positive transitions of a square wave. For $R = 51$ kΩ, determine a value of C such that the output pulse will have a width of 750 μs.

7.28 Repeat Prob. 7.27, but make the circuit trigger on negative transitions of the square wave.

7.29 Using the circuit described in Prob. 7.27, make a careful sketch of the input and output waveforms, assuming the input square wave has a frequency of:

a. 1 kHz b. 5 kHz

7.30 In the 74123 in Fig. 7.19, $R = 47$ kΩ and $C =$ 10,000 pF. Calculate the output pulse width.

7.31 Redraw the 74123 logic circuit shown in Fig. 7.18 and show how to connect the circuit such that it will trigger on positive transitions of a square wave. Given $R = 51$ kΩ, determine a value of C such that the output pulse will have a width of 750 μs.

7.32 Repeat Problem 7.31, but make the circuit trigger on negative transitions of the square wave.

7.33 Using the circuit described in Prob. 7.31, make a careful sketch of the input and output waveforms if the input square wave has a frequency of:

a. 1 kHz b. 5 kHz

Section 7.7

7.34 The input to the circuit in Fig. 7.20 is a 250-kHz square wave. Determine the proper timing capacitor values to generate a string of positive-going, 0.1-μs pulses, delayed by 2.0 μs from the rising edges of the input square wave. Assume $R_1 = R_2 = 1$ kΩ.

7.35 Draw the waveforms, input and output, for the circuit in Fig. 7.20, given that both timing resistors are 470 Ω, $C_1 = 0.1$ μF, $C_2 = 0.01$ μF, and the input waveform has a frequency of 20 kHz.

7.36 Show how to use the circuit in Fig. 7.20 to generate a 0.2-μs strobe pulse centered on the positive half cycle of a 200-kHz square wave (similar to Fig. 7.22b). Draw the complete circuit and calculate all timing resistor and capacitor values. Assume $R_1 = R_2 = 1$ kΩ.

7.37 Calculate values for the timing resistors and capacitors in Fig. 7.23 to generate a clock waveform that has:

a. A frequency of 100 kHz and a duty cycle of 25 percent

b. A frequency of 500 kHz and a duty cycle of 50 percent

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to implement a 100 kHz pulse generator with 40 percent duty cycle.

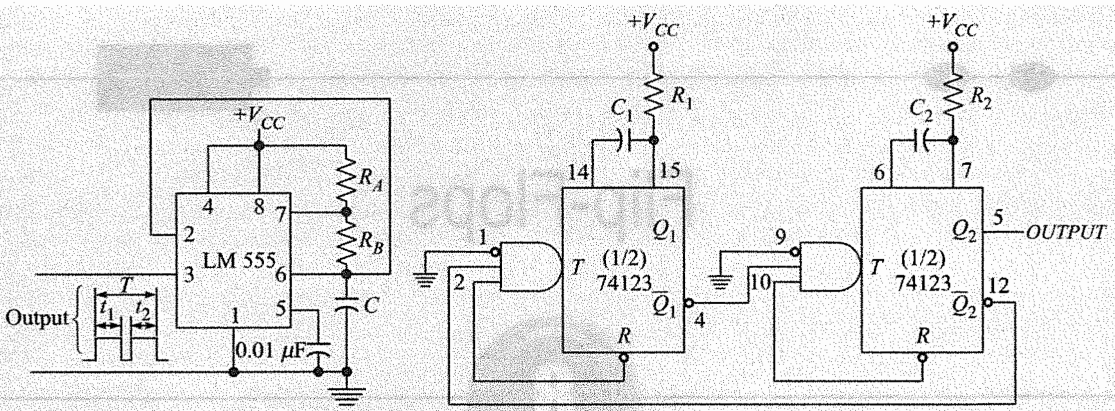
Theory: Refer to Fig. 7.13b. The 555 based pulse generator follows the following two relations.

$$t_L = 0.693\ R_B C$$
$$t_H = 0.693(R_A + R_B)C$$

Refer to Fig. 7.23a. 74123 essentially is a monostable and can be used in positive feedback. It follows the relation

$$t_L = 0.33\ R_1 C_1$$
$$t_H = 0.33\ R_2 C_2$$

Apparatus: 5 VDC Power supply, Multimeter, Bread Board, and Oscilloscope.



Work element: Study the working of IC 555 and 74123, and understand the different input outputs. From above relations, calculate the resistance and capacitance values. See the waveform in oscilloscope. Calculate duty cycle from the oscilloscope reading and compare with theoretical value. Conduct similar exercise for 74123 based circuit as shown. Repeat the experiment with other combinations of resistance and capacitance values.

Answers to Self-tests

1. An input is sensitive to PTs, and the circuit output changes synchronously with PTs. The circuit output changes in synchronism with NTs.
2. It means that a circuit input is sensitive to PTs. (See Fig. 7.6b.)
3. The logic symbol for an input sensitive to NTs is a bubble in front of a dynamic input indicator. (See Fig. 7.7b.)
4. A series mode offers low impedance at resonance, thus providing positive feedback for oscillation. A parallel mode offers high impedance at resonance, and thus provides insufficient feedback to produce oscillation.
5. Unnecessary. They simply simulate a load condition.
6. It means that the circuit has two input threshold voltage levels—an upper threshold and a lower threshold. By contrast, a simple inverter has only a single threshold voltage level.
7. Noninverting: the input and output are both high (or both low) at the same time (no phase shift). Inverting: 180° phase shift between input and output.
8. Schmitt triggers can be used to clean up a noisy signal or to change a signal having a slow rise time into one having a fast rise time.
9. A circuit has two output states, neither of which is stable.
10. True
11. Inversely
12. A circuit has two output states, one of which is stable.
13. True
14. The stable state is low.
15. Nonretriggerable
16. True
17. 0.69
18. Glitches are the unwanted pulses appearing at the output of a gate when two or more inputs change state simultaneously.
19. A strobe pulse is a pulse timed to eliminate glitches.

Flip-Flops

8

OBJECTIVES

- ✦ Describe the operation of the basic RS flip-flop and explain the purpose of the additional input on the gated (clocked) RS flip-flop
- ✦ Show the truth table for the edge-triggered. RS flip-flop, edge-triggered D flip-flop, and edge-triggered JK flip-flop
- ✦ Discuss some of the timing problems related to flip-flops
- ✦ Draw a diagram of a JK master-slave flip-flop and describe its operation
- ✦ State the cause of contact bounce and describe a solution for this problem
- ✦ Describe characteristic equations of Flip-Flops and analysis techniques of sequential circuit
- ✦ Describe excitation table of Flip-Flops and explain conversion of Flip-Flops as synthesis example

The outputs of the digital circuits considered previously are dependent entirely on their inputs. That is, if an input changes state, output may also change state. However, there are requirements for a digital device or circuit whose output will remain unchanged, once set, even if there is a change in input level(s). Such a device could be used to store a binary number. A flip-flop is one such circuit, and the characteristics of the most common types of flip-flops used in digital systems are considered in this chapter. Flip-flops are used in the construction of registers and counters, and in numerous other applications. The elimination of switch contact bounce is a clever application utilizing the unique operating characteristics of flip-flops. In a sequential logic circuit flip-flops serve as key memory elements. Analysis of such circuits are done through truth tables or characteristic equations of flip-flops. The analysis result is normally presented through state table or state transition diagram and also through timing diagram. Conversion of flip-flop from one kind to another can be posed as a synthesis problem where flip-flop excitation tables are very useful.

8.1 *RS* FLIP-FLOPS

Any device or circuit that has two stable states is said to be *bistable*. For instance, a toggle switch has two stable states. It is either up or down, depending on the position of the switch as shown in Fig. 8.la. The switch is also said to have *memory* since it will remain as set until someone changes its position.

A *flip-flop* is a bistable electronic circuit that has two stable states—that is, its output is either 0 or +5 Vdc as shown in Fig. 8.1b. The flip-flop also has memory since its output will remain as set until something is done to change it. As such, the flip-flop (or the switch) can be regarded as a memory device. In fact, any bistable device can be used to store one binary digit (bit). For instance, when the flip-flop has its output set at 0 Vdc, it can be regarded as storing a logic 0 and when its output is set at +5 Vdc, as storing a logic 1. The flip-flop is often called a *latch*, since it will hold, or latch, in either stable state.

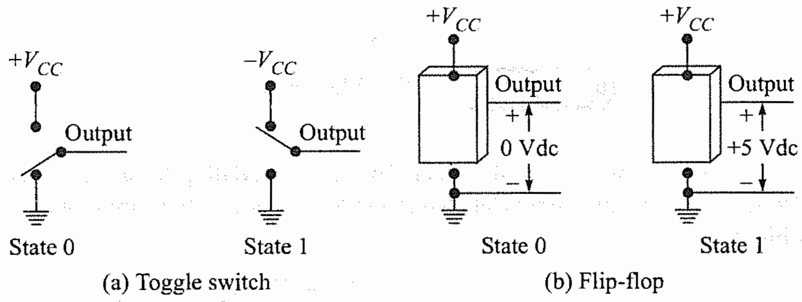


Fig. 8.1 Bistable devices

Basic Idea

One of the easiest ways to construct a flip-flop is to connect two inverters in series as shown in Fig. 8.2a. The line connecting the output of inverter *B* (*INV B*) back to the input of inverter *A* (*INV A*) is referred to as the *feedback line*.

For the moment, remove the feedback line and consider V_1 as the input and V_3 as the output as shown in Fig. 8.2b. There are only two possible signals in a digital system, and in this case we will define $L = 0 = 0$ Vdc and $H = 1 = +5$ Vdc. If V_1 is set to 0 Vdc, then V_3 will also be 0 Vdc. Now, if the feedback line shown in Fig. 8.2b is reconnected, the ground can be removed from V_1, and V_3, will remain at 0 Vdc. This is true since once the input of INV A is grounded, the output of INV B will go low and can then be used to hold the input of INV A low by using the feedback line. This is one stable state—$V_3 = 0$ Vdc.

Conversely, if V_1 is +5 Vdc, V_3 will also be +5 Vdc as seen in Fig. 8.2c. The feedback line can again be used to hold V_1 at +5 Vdc since V_3 is also at +5 Vdc. This is then the second stable state— $V_3 = +5$ Vdc.

NOR-Gate Latch

The basic flip-flop shown in Fig. 8.2a can be improved by replacing the inverters with either NAND or NOR gates. The additional inputs on these gates provide a convenient means for application of input signals to

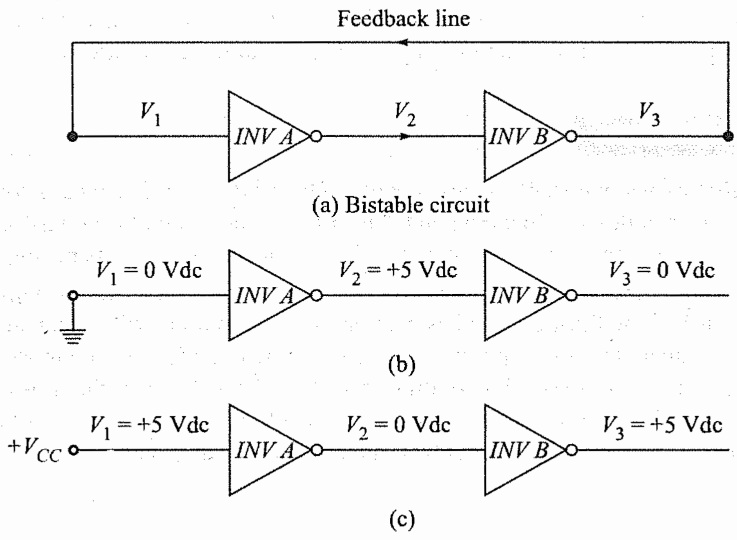


Fig. 8.2 **Bistable circuit**

switch the flip-flop from one stable state to the other. Two 2-input NOR gates are connected in Fig. 8.3a to form a flip-flop. Notice that if the two inputs labeled *R* and *S* are ignored, this circuit will function exactly as the one shown in Fig. 8.2a.

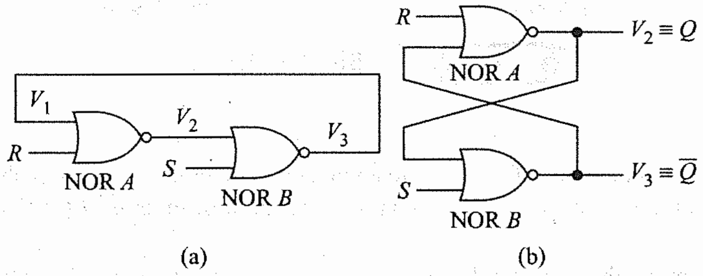


Fig. 8.3 **NOR-gate flip-flop**

This circuit is redrawn in a more conventional form in Fig. 8.3b. The flip-flop actually has two outputs, defined in more general terms as Q and $\overline{Q}$. It should be clear that regardless of the value of Q, its complement is $\overline{Q}$. There are two inputs to the flip-flop defined as *R* and *S*. The input/output possibilities for this *RS* flip-flop are summarized in the truth table in Fig. 8.4. To aid in understanding the operation of this circuit, recall that an $H = 1$ at any input of a NOR gate forces its output to an $L = 0$.

1. The first input condition in the truth table is $R = 0$ and $S = 0$. Since a 0 at the input of a NOR gate has no effect on its output, the flip-flop simply remains in its present state; that is, Q remains unchanged.
2. The second input condition $R = 0$ and $S = 1$ forces the output of NOR gate *B* low. Both inputs to NOR gate *A* are now low, and the NOR-gate output must be high. Thus a 1 at the *S* input is said to *SET* the flip-flop, and it switches to the stable state where $Q = 1$.

3. The third input condition is $R = 1$ and $S = 0$. This condition forces the output of NOR gate A low, and since both inputs to NOR gate B are now low, the output must be high. Thus a 1 at the R input is said to RESET the flip-flop, and it switches to the stable state where $Q = 0$ (or $\overline{Q} = 1$).
4. The last input condition in the table, $R = 1$ and $S = 1$, is forbidden, as it forces the outputs of both NOR gates to the low state. In other words, both $Q = 0$ and $\overline{Q} = 0$ at the same time! But this violates the basic definition of a flip-flop that requires Q to be the complement of $\overline{Q}$, and so it is generally agreed never to impose this input condition. Incidentally, if this condition is for some reason, imposed and the next input is $R = 0$, $S = 0$ then the resulting state Q depends on propagation delays of two NOR gates. If delay of gate A is less, i.e. it acts faster, then $Q = 1$ else it is 0. Such dependence makes the job of a design engineer difficult, as any replacement of a NOR gate will make Q unpredictable. That's why $R = 1$, $S = 1$ is forbidden and truth table entry is ?.

R	S	Q	Action
0	0	Last state	No change
0	1	1	SET
1	0	0	RESET
1	1	?	Forbidden

Fig. 8.4 **Truth table for a NOR-gate *RS* flip-flop**

It is also important to remember that TTL gate inputs are quite noise-sensitive and therefore should never be left unconnected (floating). Each input must be connected either to the output of a prior circuit, or if unused, to GND or $+V_{CC}$.

Example 8.1 Use the pinout diagram for a 54/7427 triple 3-input NOR gate and show how to connect a simple *RS* flip-flop.

Solution One possible arrangement is shown in Fig. 8.5. Notice that pins 3 and 4 are tied together, as are pins 10 and 11; thus no input leads are left unconnected and the two gates simply function as 2-input gates. The third NOR gate is not used. (It can be a spare or can be used elsewhere.)

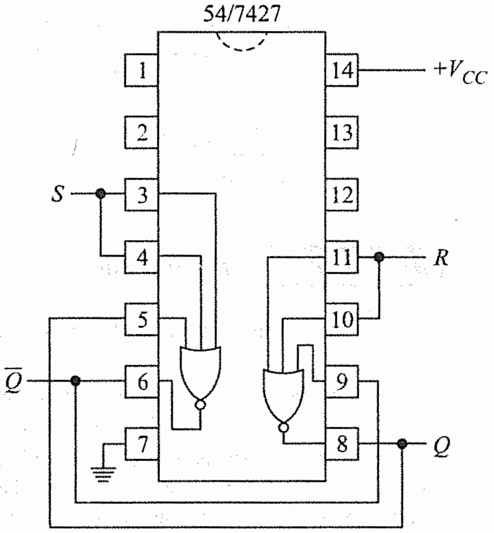
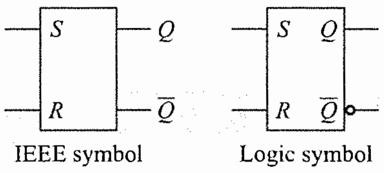


Fig. 8.5 **54/7427**

The standard logic symbols for an *RS* flip-flop are shown in Fig. 8.6 along with its truth table. The truth table is necessary since it describes exactly how the flip-flop functions.



(a)

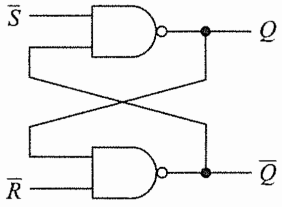
R	*S*	*Q*
0	0	Last state
0	1	1
1	0	0
1	1	? (Forbidden)

(b) Truth table

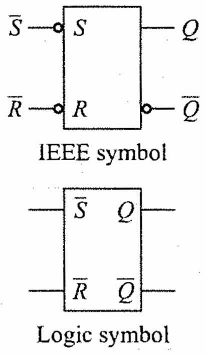
Fig. 8.6 ***RS* flip-flop**

NAND-Gate Latch

A slightly different latch can be constructed by using NAND gates as shown in Fig. 8.7. The truth table for this NAND-gate latch is different from that for the NOR-gate latch. We will call this latch an $\overline{R}\,\overline{S}$ flip-flop. To understand how this circuit functions, recall that *a low on any input to a NAND gate will force its output high*. Thus a low on the $\overline{S}$ input will set the latch ($Q = 1$ and $\overline{Q} = 0$). A low on the $\overline{R}$ input will reset it ($Q = 0$). If both $\overline{R}$ and $\overline{S}$ are high, the flip-flop will remain in its previous state. Setting both $\overline{R}$ and $\overline{S}$ low simultaneously is forbidden since this forces both Q and $\overline{Q}$ high.



(a) NAND gate latch



(b)

$\overline{R}$	$\overline{S}$	*Q*
1	1	Last state
1	0	1
0	1	0
0	0	? (Forbidden)

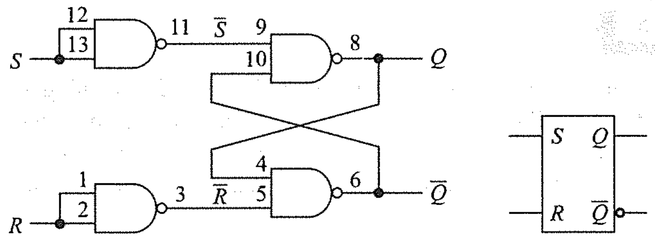
(c) Truth table

Fig. 8.7 **$\overline{R}\,\overline{S}$ flip-flop**

Example 8.2 Show how to convert the $\overline{R}\,\overline{S}$ flip-flop in Fig. 8.7 into an *RS* flip-flop.

Solution By placing an inverter at each input as shown in Fig. 8.8, the 2 inputs are now *R* and *S*, and the resulting circuit behaves exactly as the *RS* flip-flop in Fig. 8.6. A single 54/7400 (quad 2-input NAND gate) is used.

Simple latches as discussed in this section can be constructed from NAND or NOR gates or obtained as medium-scale integrated circuits (MSI). For instance, the 74LS279 is a *quad* $\overline{R}\,\overline{S}$ *latch*. The pinout and truth table for this circuit are given in Fig. 8.9. Study the truth table carefully, and you will see that the latches behave exactly like the $\overline{R}\,\overline{S}$ flip-flop discussed above.

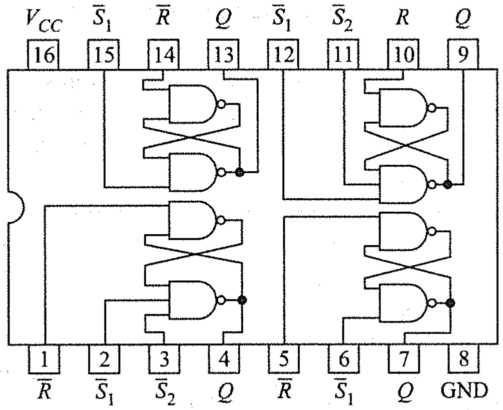


(a) 54/7400 (b) Logic symbol

R	S	Q
0	0	Last state
0	1	1
1	0	0
1	1	? (Forbidden)

(c)

Fig. 8.8 **An RS flip-flop (latch)**



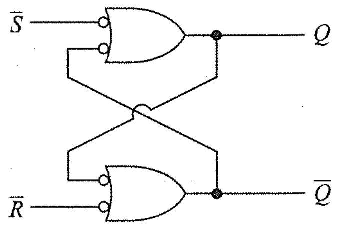
$\overline{S}_1$	$\overline{S}_2$	$\overline{R}$	Q
0	0	0	? Forbidden
0	X	1	1
X	0	1	1
1	1	0	0
1	1	1	? Forbidden

X = Don't care

(a) Pinout 74S279A (b) Truth table

Fig. 8.9 **Quad SET-RESET latch**

The NOR-gate flip-flop in Fig. 8.3 is seen to be an active-high circuit because an $H = 1$ at either the S or R input is required to change the output Q. On the other hand, the NAND-gate flip-flop in Fig. 8.7 can be considered an active-low circuit because an $L = 0$ at either input is required to change Q. The NAND gates in Fig. 8.7 can be changed to bubbled-input OR gates as shown in Fig. 8.10. This circuit is equivalent to the NAND-gate latch in Fig. 8.7 and functions in exactly the same way. However, the bubbled inputs more clearly express circuit operation.



$\overline{R}\,\overline{S}$ flip-flop

Fig. 8.10 **Bubbled OR-gate equivalent of Fig. 8.7**

SELF-TEST

1. What do the letters R and S stand for in the term "RS latch"?
2. A 74LS279 is a quad latch. What does quad mean?
3. Why is the NAND-gate latch considered active-low?

8.2 GATED FLIP-FLOPS

Two different methods for constructing an *RS* flip-flop were discussed in Sec. 8.1. The NOR-gate realization in Fig. 8.3b is an exact equivalent of the NAND-gate realization in Fig. 8.8a, and they both have the exact same symbol and truth table as given in Fig. 8.6. Both of these *RS* flip-flops, or latches, are said to be *transparent*; that is, any change in input information at *R* or *S* is transmitted immediately to the output at *Q* and $\overline{Q}$ according to the truth table.

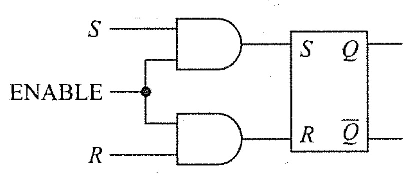
Clocked *RS* Flip-Flops

The addition of two AND gates at the *R* and *S* inputs as shown in Fig. 8.11 will result in a flip-flop that can be enabled or disabled. When the ENABLE input is low, the AND gate outputs must both be low and changes in neither *R* nor *S* will have any effect on the flip-flop output *Q*. The latch is said to be *disabled.*

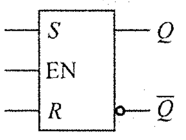
When the ENABLE input is high, information at the *R* and *S* inputs will be transmitted directly to the outputs. The latch is said to be *enabled.* The output will change in response to input changes as long as the ENABLE is high. When the ENABLE input goes low, the output will retain the information that was present on the input when the high-to-low transition took place.

In this fashion, it is possible to *strobe* or *clock* the flip-flop in order to store information (set it or reset it) at any time, and then hold the stored information for any desired period of time. This flip-flop is called a *gated* or *clocked RS flip-flop*. The proper symbol and truth table are given in Fig. 8.11b. Notice that there are now three inputs—*R*, *S*, and the ENABLE or CLOCK input, labeled *EN*. Notice also that the truth-table output is not simply *Q*, but Q_{n+1}. This is because we must consider two different instants in time: the time before the ENABLE goes low Q_n and the time just after ENABLE goes low Q_{n+1}. When EN = 0, the flip-flop is disabled and *R* and *S* have no effect; thus the truth table entry for *R* and *S* is *X* (don't care).

Example 8.3 Explain the meaning of Q_n the truth table in Fig. 8.11b.



(a) Logic diagram



EN	*S*	*R*	Q_{n+1}
1	0	0	Q_n (no change)
1	0	1	0
1	1	0	1
1	1	1	? (Illegal)
0	*X*	*X*	Q_n (no change)

(b) IEEE symbol and truth table

Fig. 8.11 **Clocked *RS* flip-flop**

Solution For the flip-flop to operate properly, there must be a PT on the EN input. While EN is high, the information on *R* and *S* causes the latch to set or reset. Then when EN transitions back to low, this information is retained in the latch. When this NT occurred, both *R* and *S* inputs were low (0), and thus there was no change of state. In other words, the value of *Q* at time $n + 1$ is the same as it was at time *n*. Remember that time *n* occurs just before the NT on EN, and time $n + 1$, occurs just after the transition.

The logic diagrams shown in Fig. 8.12a and b illustrate two different methods for realizing a clock *RS* flip-flop. Both realizations are widely used in medium- and large-scale integrated circuits, and you will find them easy to recognize. You might like to examine the logic diagrams for a 54LS109 or a 54LS74, for instance.

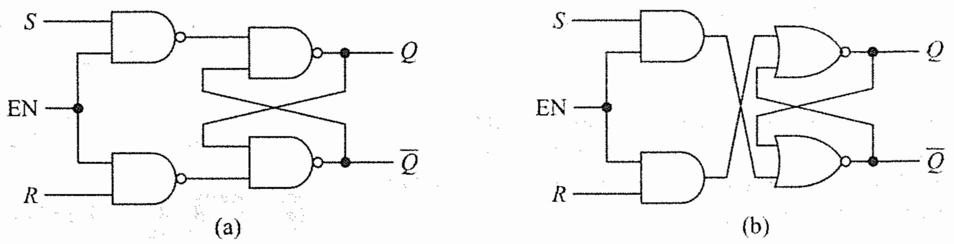


Fig. 8.12 **Two different realizations for a clocked *RS* flip-flop**

Example 8.4 Figure 8.13 shows the input waveforms R, S, and EN applied to a clocked RS flip-flop. Explain the output waveform Q.

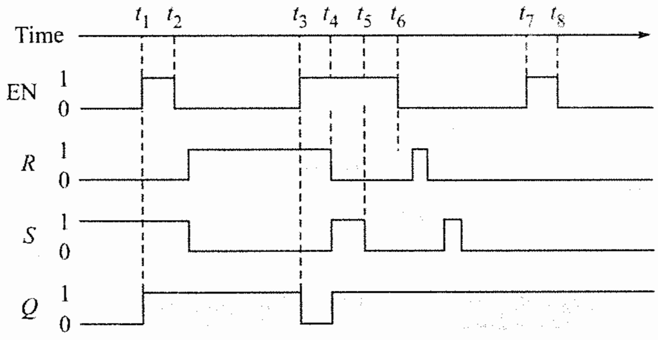


Fig. 8.13

Solution Between t_2 and t_3 both R and S change states, but since EN is low, the flip-flop is still disabled and Q remains at 1.

Between t_3 and t_6, the flip-flop will respond to any change in R and S since EN is high. Thus at t_3 Q goes low, and at t_4 it goes back high. No change occurs at t_5. At t_6 the value $Q = 1$ is latched and no changes in Q occur between t_6 and t_7 even though both R and S change.

Between t_7, and t_8 no change in Q occurs since both R and S are low. Initially, the flip-flop is reset ($Q = 0$). At time t_1 EN goes high; the flip-flop is now enabled, and it is immediately set ($Q = 1$) since $R = 0$ and $S = 1$. At time t_2 EN goes low and the flip-flop is disabled and latches in the stable state $Q = 1$.

Clocked D Flip-Flops

The RS flip-flop has two data inputs, R and S. To store a high bit, you need a high S; to store a low bit, you need a high R. Generation of two signals to drive a flip-flop is a disadvantage in many applications. Furthermore, the forbidden condition of both R and S high may occur inadvertently. This has led to the D flip-flop, a circuit that needs only a single data input.

Figure 8.14 shows a simple way to build a D (Data) flip-flop. This flip-flop is disabled when EN is low, but is *transparent* when EN is high. The action of the circuit is straightforward, as follows. When EN is low, both AND gates are disabled; therefore, D can change value without affecting the value of Q. On the other hand, when EN is high, both AND gates are enabled. In this case, Q is forced to equal the value of D. When EN again goes low, Q retains or stores the last value of D.

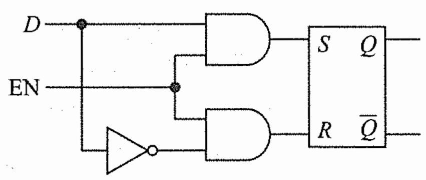
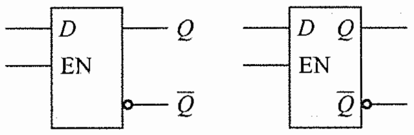


Fig. 8.14 **A *D* Flip-flop**

There are many ways to design D flip-flops. In general, a D flip-flop is a bistable circuit whose D input is transferred to the output when EN is high. Figure 8.15 shows the logic symbols used for any type of D flip-flop.

In this section we're talking about the kind of D flip-flop in which Q can follow the value of D while EN is high. In other words, if the data bit changes while EN is high, the last value of D before EN return low is the value of D that is stored. This kind of D flip-flop is often called a D latch.

Figure 8.15b shows the truth table for a D latch. While (EN) is low, D is a don't care (X); Q will remain latched in its last state. When EN is high, Q takes on the last value of D. If D is changing while EN is high, it is the last value of D that is stored.



(a) D flip-flop logic symbol

EN	D	Q_{n+1}
0	X	Q_n (last state)
1	0	0
1	1	1

(b) Truth table

Fig. 8.15 ***D* Flip-flop logic symbol**

The idea of data storage is illustrated in Fig. 8.16. Four D latches are driven by the same clock signal. When the clock goes high, input data is loaded into the flip-flops and appears at the output. Then when the clock goes low, the output retains the data. For instance, suppose that the data input is

$$D_3D_2D_1D_0 = 0111$$

When the clock goes high, this word is loaded into the D latches, resulting in an output of

$$Q_3Q_2Q_1Q_0 = 0111$$

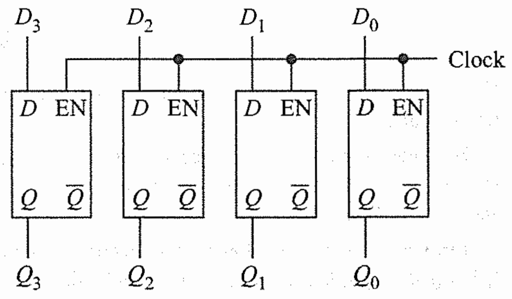


Fig. 8.16 **Storing a 4-bit word**

After the clock goes low, the output data is retained, or stored. As long as the clock is low, the D values can change without affecting the Q values.

The 7475 in Fig. 8.17 is a TTL MSI circuit that contains four D latches; it's called a *quad bistable latch*. The 7475 is ideal for handling 4-bit nibbles of data. If more than one 7475 is used, words of any length can be stored.

SELF-TEST

4. What does an entry X mean in a flip-flop truth table?
5. What could you do to disable the flip-flop in Fig. 8.11?
6. Which flip-flop is easier to use, the RS of the D, as a clocked or gated latch to store data?

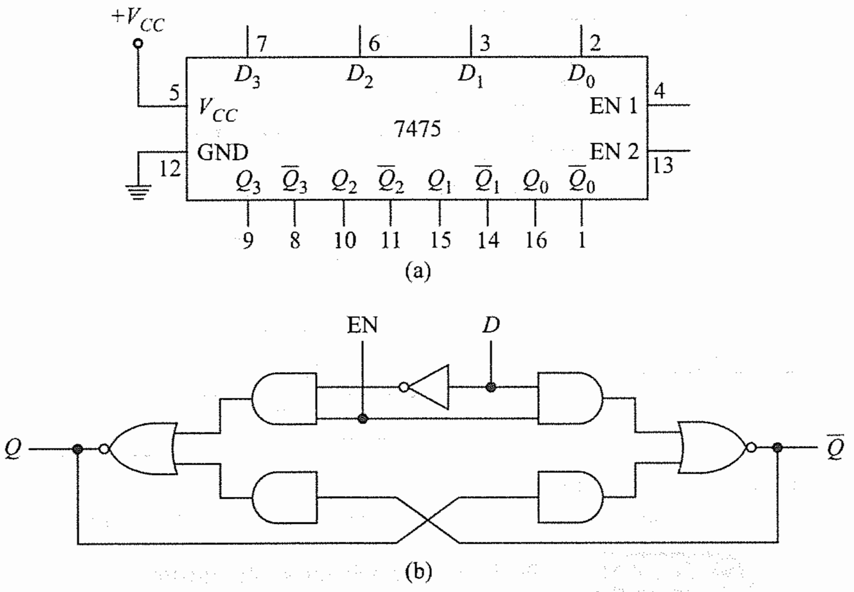


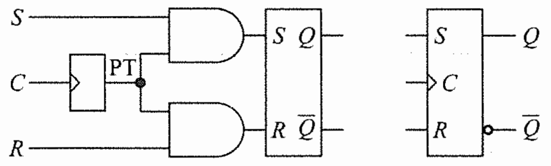
Fig. 8.17 4-bit bistable latch: (a) Pinout, (b) Logic diagram (each latch)

8.3 EDGE-TRIGGERED *RS* FLIP-FLOPS

The simple latch-type flip-flops presented in Sec. 8.1 are completely *transparent*; that is, the output Q immediately follows *any* change of state at the input (R, S, or D). The gated or clocked RS and D flip-flops in Sec. 8.2 might be considered *semitransparent*. That is, the output Q will change state immediately provided that the EN input is high. If any of these flip-flops are used in a synchronous system, care must be taken to ensure that all flip-flop inputs change state in synchronism with the clock. One way of resolving the problem for gated flip-flops is to allow changes in R, S, and D input levels only when EN is low (or require fixed levels at R, S, and D any time EN is high). At the very least, these are highly inconvenient restrictions, and at the worst they may in fact be impossible to realize. From the previous chapter, we know that virtually all digital systems operate in a synchronous mode. Thus the *edge-triggered flip-flop* was developed to overcome these rather severe restrictions.

Positive-Edge-Triggered *RS* Flip-Flops

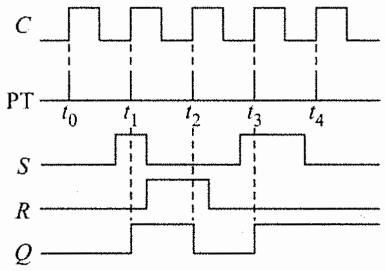
In Fig. 8.18a, the clock (C) is applied to a positive pulse-forming circuit (discussed in Sec. 7.1). The PTs developed are then applied to a gated RS flip-flop. The result is a positive-edge-triggered RS flip-flop, with the IEEE symbol given in Fig. 8.18b. The small triangle inside the symbol (dynamic input indicator) indicates that Q can change state only with PTs of the clock (C). Each PT of the clock in Fig. 8.18c produces a very narrow PT that is applied to the AND gates. The AND gates are active only while the PT is high (perhaps 25 ns), and thus Q can change state only during this short time period. In this manner Q changes state in synchronism with the PTs of the clock.



(a) Logic diagram (b) IEEE symbol

C	S	R	Q_{n+1}	Action
↑	0	0	Q_n	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	?	Illegal

(c) Truth table



(d) Positive-edge-triggered *RS* flip-flop

Fig. 8.18 **Positive-edge-triggered *RS* flip-flop**

This flip-flop is easy to use in any synchronous system! Another way of expressing its behavior is to say the flip-flop is transparent only during PTs; it is not transparent for the remainder of the time. In other words, *S* and *R* inputs affect *Q* only while the positive pulse is high, and they need to be static only during this very short time.

The truth table for the edge-triggered *RS* flip-flop is given in Fig. 8.18c. The small vertical arrows under *C* (clock) mean that changes of state (*Q*) occur according to the *R* and *S* levels, but only during PTs of the clock. Look at the waveforms in Fig. 8.18d. Note that when *Q* changes state, it does so in exact synchronism with PTs of the clock *C*.

Example 8.5 Use the positive-edge-triggered *RS* flip-flop truth table to explain *Q* changes of state with time in Fig. 8.18d.

Solution Here's what happens at each point in time:

Time t_0: $S = 0, R = 0$, no change in Q(Q remains 0)

Time t_1: $S = 1, R = 0$, Q changes from 0 to 1

Time t_2: $S = 0, R = 1$, Q resets to 0

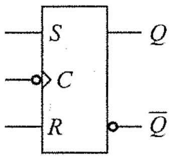
Time t_3: $S = 1, R = 0$, Q sets to 1

Time t_4: $S = 0, R = 0$, no change in Q(Q remains 1)

Notice that either *R* or *S*, or both, are allowed to change state at any time, whether *C* is high or low. The only time both *R* and *S* must be stable (unchanging) is during the short PTs of the clock.

Negative-Edge-Triggered *RS* Flip-Flops

The symbol in Fig. 8.19a is for a negative-edge-triggered *RS* flip-flop. The truth table in Fig. 8.19b shows that *Q* changes state according to the *R* and *S* inputs, but only during NTs of the clock. On the IEEE symbol, the small bubble on the clock input (*C*) means active-low. This bubble, along with the dynamic input indicator,



(a) IEEE symbol

C	S	R	Q_{n+1}	Action
↓	0	0	Q_n	No change
↓	0	1	0	RESET
↓	1	0	1	SET
↓	1	1	?	Illegal

(b) Truth table

Fig. 8.19 Negative-edge-triggered *RS* flip-flop

means negative-edge triggering. This flip-flop behaves exactly like the positive-edge-triggered *RS* flip-flop, except that changes in output Q are synchronized with NTs of the clock (C).

Example 8.6 Use the negative-edge-triggered *RS* flip-flop truth table to explain Q changes of state with time in Fig. 8.20.

Solution Here's what happens at each point in time:

Time t_0: $S = 0, R = 0$, no change in Q (Q remains 0)
Time t_1: $S = 1, R = 0$, Q changes from 0 to 1
Time t_2: $S = 0, R = 1$, Q resets to 0
Time t_3: $S = 1, R = 0$, Q sets to 1
Time t_4: $S = 0, R = 0$, no change in Q (Q remains 1)

Notice that either R or S, or both, are allowed to change state at any time, whether C is high or low. The only time both R and S must be stable (unchanging) is during the short NTs of the clock.

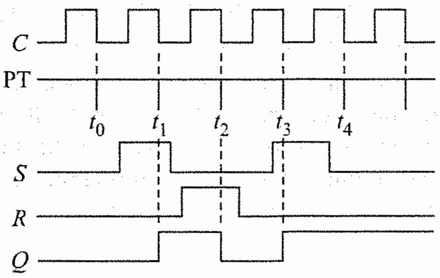


Fig. 8.20

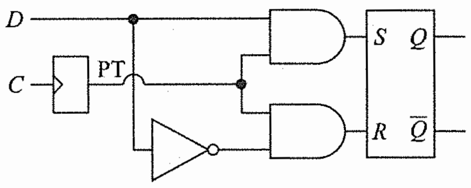
SELF-TEST

7. What does it mean to say that a flip-flop is transparent?
8. What is positive-edge triggering?
9. How does an *RS* latch differ from an edge-triggered *RS* flip-flop?

8.4 EDGE-TRIGGERED *D* FLIP-FLOPS

Although the *D* latch is used for temporary storage in electronic instruments, an even more popular kind of *D* flip-flop is used in digital computers and systems. This kind of flip-flop samples the data bit at a unique point in time.

Figure 8.21 shows a positive pulse-forming circuit at the input of a *D* latch. The narrow positive pulse (PT) enables the AND gates for an instant. The effect is to activate the AND gates during the PT of C, which is equivalent to sampling the value of D for an instant. At this unique point in time, D and its complement hit the flip-flop inputs, forcing Q to set or reset (unless Q already equals D). Again, this operation is called *edge triggering* because the flip-flop responds only when the clock is in transition between its two voltage states. The triggering in Fig. 8.21 occurs on the positive-going edge of the clock; this is why it's referred to as *positive-edge triggering*.



(a) Circuit diagram

C	D	Q_{n+1}
0	X	Q_n (last state)
$\uparrow$	0	0
$\uparrow$	1	1

(c) Truth table

Fig. 8.21 Positive-edge-triggered *D* flip-flop

The truth table in Fig. 8.21b summarizes the action of a positive-edge-triggered *D* flip-flop. When the clock is low, *D* is a don't care and *Q* is latched in its last state. On the leading edge of the clock (PT), designated by the up arrow, the data bit is loaded into the flip-flop and *Q* takes on the value of *D*.

When power is first applied, flip-flops come up in random states. To get some computers started, an operator has to push a RESET button. This sends a CLEAR or RESET signal to all flip-flops. Also, it's necessary in some digital systems to preset (synonymous with set) certain flip-flops.

Figure 8.22 shows how to include both functions in a *D* flip-flop. The edge triggering is the same as previously described. Depressing the RESET button will set *Q* to 1 with the first PT of the clock. *Q* will remain high as long as the button is held closed. The first PT of the clock after releasing the button will set *Q* according to the *D* input. Furthermore, the OR gates allow us to slip in a high PRESET or a high CLEAR when desired. A high PRESET forces *Q* to equal 1; a high CLEAR resets *Q* to 0.

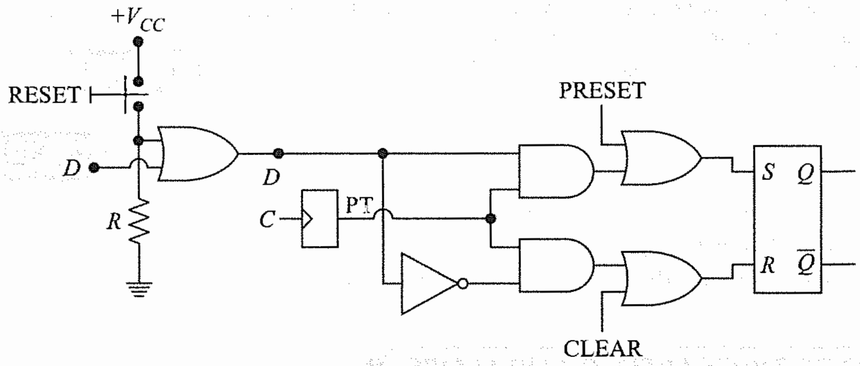


Fig. 8.22 PRESET and CLEAR functions

The PRESET and CLEAR are called *asynchronous inputs* because they activate the flip-flop independently of the clock. On the other hand, the *D* input is a synchronous input because it has an effect only with PTs of the clock.

Figure 8.23a is the IEEE symbol for a positive-edge-triggered *D* flip-flop. The clock input has a small triangle to serve as a reminder of edge triggering. When you see this symbol, remember what it means; the *D* input is sampled and stored on PTs of the clock.

Sometimes, triggering on NTs of the clock is better suited to the application. In this case, an internal inverter can complement the clock pulse before it reaches the AND gates. Figure 8.23b is the symbol for a negative-edge-triggered *D* flip-flop. The bubble and triangle symbolize the negative-edge triggering.

Figure 8.23c is another commercially available D flip-flop (the 54/74175 or 54/74LS175). Besides having positive-edge triggering, it has an inverted CLEAR input This means that a low CLR resets it. The 54/74175 has four of these D flip-flops in a single 16-pin dual in-line package (DIP), and it's referred to as a *quad D-type flip-flop with clear.*

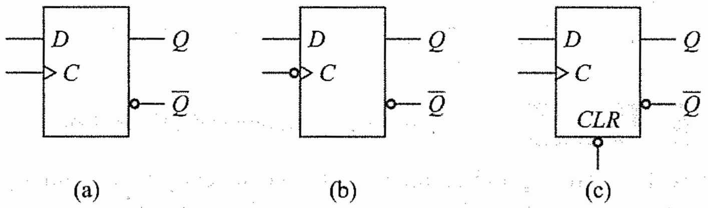


Fig. 8.23 ***D* flip-flop symbols: (a) Positive-edge-triggered, (b) Negative-edge-triggered, (c) Positive-edge-triggered with active low clear**

SELF-TEST

10. The C input to the D flip-flop in Fig. 8.21 is held low. What effect does the D input have?
11. To preset the flip-flop in Fig. 8.22, what level is required at the preset input. What is the resulting state of Q?

8.5 EDGE-TRIGGERED *JK* FLIP-FLOPS

Setting $R = S = 1$ with an edge-triggered RS flip-flop forces both Q and $\overline{Q}$ to the same logic level. This is an *illegal* condition, and it is not possible to predict the final state of Q. The JK flip-flop accounts for this illegal input, and is therefore a more versatile circuit. Among other things, flip-flops can be used to build counters. Counters can be used to count the number of PTs or NTs of a clock. For purposes of counting, the JK flip-flop is the ideal element to use. There are many commercially available edge-triggered JK flip-flops. Let's see how they function.

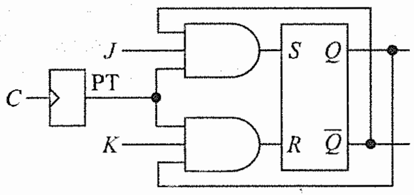
Positive-Edge-Triggered *JK* Flip-Flops

In Fig. 8.24, the pulse-forming box changes the clock into a series of positive pulses, and thus this circuit will be sensitive to PTs of the clock. The basic circuit is identical to the previous positive-edge-triggered RS flip-flop, with two important additions:

1. The Q output is connected back to the input of the lower AND gate.
2. The $\overline{Q}$ output is connected back to the input of the upper AND gate.

This cross-coupling from outputs to inputs changes the RS flip-flop into a JK flip-flop. The previous S input is now labeled J, and the previous R input is labeled K. Here's how it works:

1. When J and K are both low, both AND gates are disabled. Therefore, clock pulses have no effect. This first possibility is the initial entry in the truth table. As shown, when J and K are both 0s, Q retains its last value.
2. When J is low and K is high, the upper gate is disabled, so there's no way to set the flip-flop. The only possibility is reset. When Q is high, the lower gate passes a RESET pulse as soon as the next positive



(a) One way to implement a *JK* flip-flop

C	*J*	*K*	Q_{n+1}	Action
↑	0	0	Q_n (last state)	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	$\overline{Q}_n$ (toggle)	Toggle

(b) Truth table

Fig. 8.24 A positive-edge-triggered *JK* flip-flop

clock edge arrives. This forces *Q* to become low (the second entry in the truth table). Therefore, $J = 0$ and $K = 1$ means that the next PT of the clock resets the flip-flop (unless *Q* is already reset).

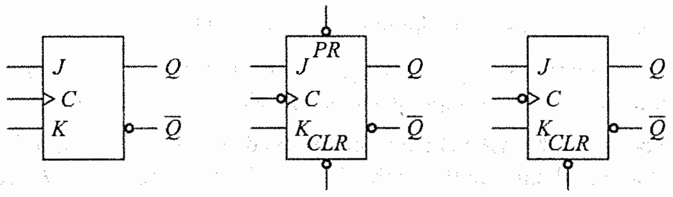
3. When *J* is high and *K* is low, the lower gate is disabled, so it's impossible to reset the flip-flop. But you can set the flip flop as follows. When *Q* is low, $\overline{Q}$ is high; therefore, the upper gate passes a SET pulse on the next positive clock edge. This drives *Q* into the high state (the third entry in the truth table). As you can see, $J = 1$ and $K = 0$ means that the next PT of the clock sets the flip-flop (unless *Q* is already high).
4. When *J* and *K* are both high (notice that this is the forbidden state with an *RS* flip-flop), it's possible to set or reset the flip-flop. If *Q* is high, the lower gate passes a RESET pulse on the next PT. On the other hand, when *Q* is low, the upper gate passes a SET pulse on the next PT. Either way, *Q* changes to the complement of the last state (see the truth table). Therefore, $J = 1$ and $K = 1$ mean the flip-flop will *toggle* (switch to the opposite state) on the next positive clock edge.

Propagation delay prevents the *JK* flip-flop from racing (toggling more than once during a positive clock edge). Here's why. In Fig. 8.24, the outputs change after the PT of the clock. By then, the new *Q* and $\overline{Q}$ values are too late to coincide with the PTs driving the AND gates. For instance, if $t_p = 20$ ns, the outputs change approximately 20 ns after the leading edge of the clock. If the PTs are narrower than 20 ns, the returning *Q* and $\overline{Q}$ arrive too late to cause false triggering.

Figure 8.25a shows a symbol for a *JK* flip-flop of any design. When you see this on a schematic diagram, remember that on the next PT of the clock:

1. *J* and *K* low: no change of *Q*.
2. *J* low and *K* high: *Q* is reset low.
3. *J* high and *K* low: *Q* is set high.
4. *J* and *K* both high: *Q* toggles to opposite state.

You can include OR gates in the design to accommodate PRESET and CLEAR as was done earlier. Figure 8.25b gives the symbol for a *JK* flip-flop with *PR* and *CLR*. Notice that it is negative-edge-triggered and requires a low *PR* to set it or a low *CLR* to reset it.



(a) Basic symbol (b) 74LS76A (c) 74LS73A

Fig. 8.25 *JK* flip-flop symbols

Figure 8.25c is another commercially available *JK* flip-flop. It is negative-edge-triggered and requires a low *CLR* to reset it. The output *Q* reacts immediately to a *PR* or *CLR* signal. Both *PR* and *CLR* are asynchronous, and they override all other input signals.

Example 8.7 Toggle flip-flop, popularly known as *T* flip-flop has following input-output relation. When input $T = 0$, the output Q does not change its state. For $T = 1$, the output Q toggles its value. Derive *T* flip-flop from *JK* flip-flop.

Solution From Fig. 8.24b we find for input $J = K = 0$, the output $Q_{n+1} = Q_n$, i.e. output does not change its state. And for $J = K = 1$, the output $Q_{n+1} = Q_n'$, i.e. output toggles. Thus, if we tie *J* and *K* inputs of *JK* flip-flop together and make a common input $T = J = K$, the resulting circuit will behave as *T* flip-flop.

SELF-TEST

12. What is the primary difference between a *JK* and an *RS* flip-flop?
13. How could you change an edge-triggered *RS* flip-flop into an edge-triggered *JK* flip-flop?

8.6 FLIP-FLOP TIMING

Diodes and transistors cannot switch states immediately. It always takes a small amount of time to turn a diode on or off. Likewise, it takes time for a transistor to switch from saturation to cutoff, and vice versa. For bipolar diodes and transistors, the switching time is in the nanosecond region.

Switching time is the main cause of propagation delay, designated t_p. This represents the amount of time it takes for the output of a gate or flip-flop to change states after the input changes. For instance, if the data sheet of an edge-triggered *D* flip-flop lists $t_p = 10$ ns, it takes about 10 ns for *Q* to change states after *D* has been sampled by the clock edge. This propagation delay time has been used to construct the "pulse-forming circuit" used with edge-triggered flip-flops. When flip-flops are used to construct counters, the propagation delay is often small enough to be ignored.

Stray capacitance at the *D* input (plus other factors) makes it necessary for data bit *D* to be at the input before the clock edge arrives. The setup time t_{setup} is the minimum amount of time that the data bit must be present before the clock edge hits. For instance, if a *D* flip-flop has a setup time of 15 ns, the data bit to be stored must be at the *D* input at least 15 ns before the clock edge arrives; otherwise, the manufacturer does not guarantee correct sampling and storing.

Furthermore, data bit *D* has to be held long enough for the internal transistors to switch states. Only after the transition is assured can we allow data bit *D* to change. Hold time t_{hold} is the minimum amount of time that data bit *D* must be present after the PT of the clock. For example, if $t_{\text{setup}} = 15$ ns and $t_{\text{hold}} = 5$ ns, the data bit has to be at the *D* input at least 15 ns before the clock edge arrives and held at least 5 ns after the clock PT.

Example 8.8 Typical waveforms for setting a 1 in a positive-edge-triggered flip-flop are shown in Fig. 8.26. Discuss the timing.

Solution The lower line in Fig. 8.26 is the time line with critical times marked on it. Prior to t_1, the data can be a 1 or a 0, or can be changing. This is shown by drawing lines for both high and low levels on *D*. From time t_1 to t_2, the

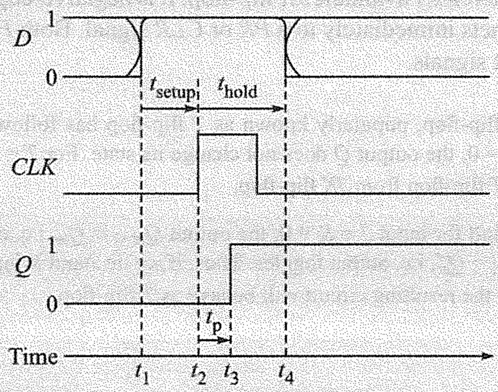


Fig. 8.26

data line D must be held steady (in this case a 1). This is the setup time t_{setup}. Data is shifted into the flip-flop at time t_2 but does not appear at Q until time t_3. The time from t_2 to t_3 is the propagation time t_p. In order to guarantee proper operation, the data line must be held steady from time t_2 until t_4; this is the hold time t_{hold}. After t_4, D is free to change states—shown by the double lines.

8.7 EDGE TRIGGERING THROUGH INPUT LOCK OUT

We have seen how edge triggering of flip-flops can be achieved by pulse forming circuit (Section 7.1). This requires application of a very narrow pulse which is generated using differential propagation delays of two signal flow paths while the flip-flops themselves are level triggered. An alternate way of achieving edge triggering is to implement a kind of lock out of the input so that it is not able to enforce a change at output which itself is level triggered. This is to say that the effect of change in input is allowed only at the edge and not after the edge. Let us see how this is possible by implementing a positive edge triggered D type flip-flop. This requires three NAND latches as shown in Fig. 8.27 with one NAND gate (number 3) having three inputs and the rest are all two input NAND gates. Note that for a NAND gate output to be 0, all the inputs must be at 1, else the output is 1. The output latch behaves like an SR flip flop where no change in output occurs if $S = 1, R = 1$.

Now, if the clock input is held at 0 then irrespective of what is present at D input, the NAND logic makes both $S = 1$, $R = 1$ and thus there could be no change in the output. If Clock = 1 then SR can always change if other inputs of NAND gates 2 and 3 change and thus the output is essentially level triggered. We will now explain how input lock out makes the circuit as a whole a positive edge triggered circuit.

Consider the case when Clock = 0 and $D = 0$ (Fig. 8.27a). Since, for a NAND gate, 0 is the forcing input, the intermediate outputs are $S = 1$, $R = 1$ and $A = 1$ which make $B = 0$. Now, clock makes a transition from 0 $\rightarrow$ 1. $D = 0$ forces $A = 1$ and $B = 0$ keeps $R = 1$. Thus, after this transition, $S = 0$, $R = 1$, $A = 1$ and $B = 0$. This makes $Q = 0$ irrespective of the previous state and one can see that the value at D, i.e. 0 is transferred to Q after the clock trigger. Next, we see if at Clock = 1, D is changed, then whether Q is changed. This is shown in Fig. 8.27b as a follow-up of Fig. 8.27a. Before D makes a transition Clock = 1, $D = 0$ and intermediate outputs $S = 0$, $R = 1$, $A = 1$, $B = 0$ and $Q = 0$. When D goes to 1, 4th NAND gate is only directly affected as D is not connected elsewhere. However, the output A of this gate does not change as it is kept held at 1 by the other input coming from $S = 0$. Thus, $S = 0$, $R = 1$, $A = 1$, $B = 0$ and $Q = 0$. This is the lock out of input we were referring to. Note that clock going from 1 to 0 does not change Q as that transition makes $S = 1$, $R = 1$.

Next, consider the case when Clock = 0 and $D = 1$. This is shown in Fig. 8.27c. $S = 1$, $D = 1$ make $A = 0$ which in turn makes $B = 1$. Now, clock makes a transition from $0 \rightarrow 1$. $A = 0$ maintains $S = 1$. Both the inputs of 2nd NAND gate being 1, $R = 0$. $S = 1$, $R = 0$ make $Q = 1$ irrespective of previous state and thus after positive clock trigger, the logic value of D arrives at Q for $D = 1$ case, too. With Clock = 1, if input D changes from 1 to 0, will the output Q change? This 4th possibility is shown in Fig. 8.27d. $D = 0$ makes $A = 1$ but $R = 0$ maintains $B = 1$ and $S = 1$. Thus, after the transition, SR remains at where it was and input D remaines locked out, i.e. unable to effect any change in the output at Clock = 1.

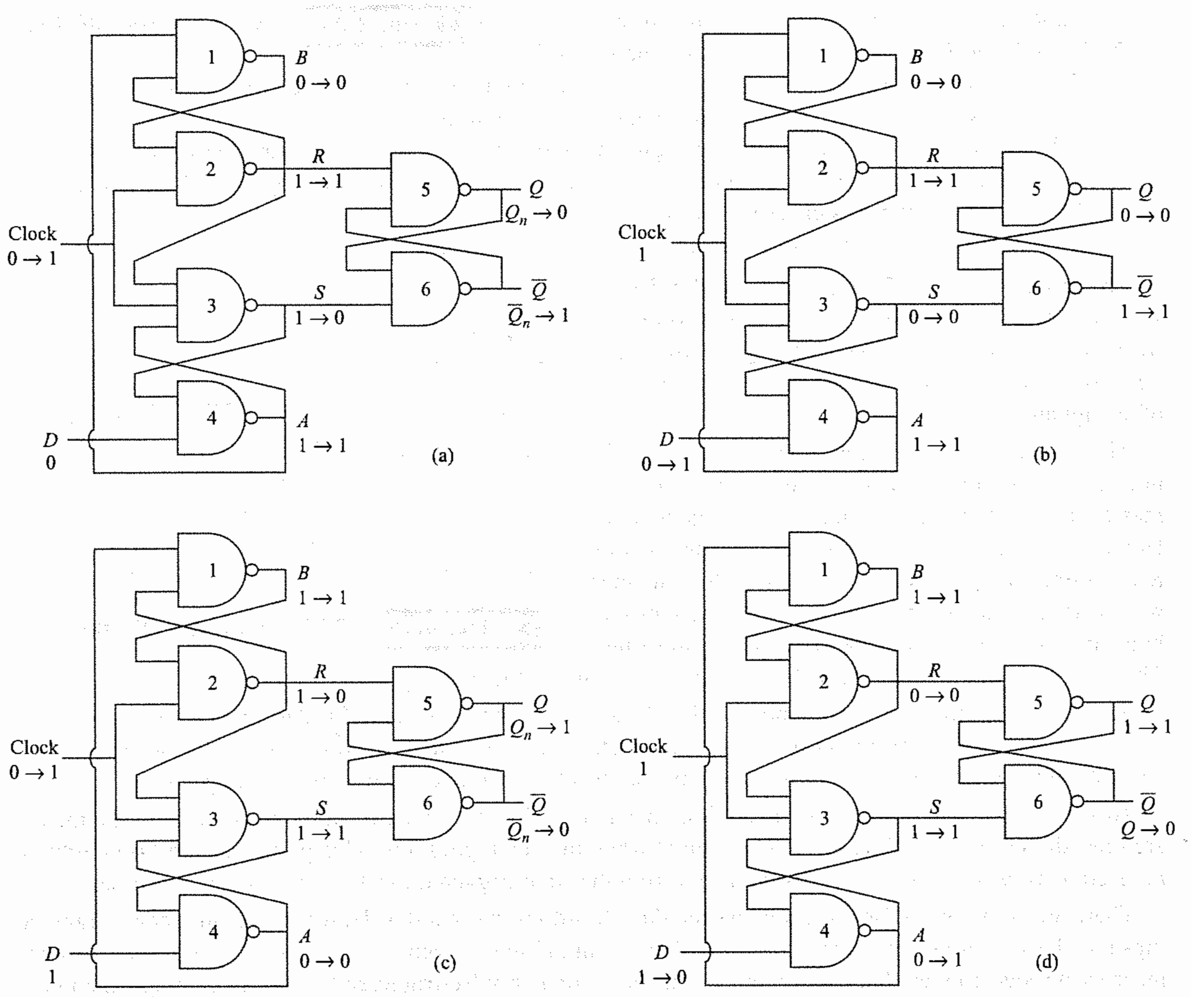


Fig. 8.27 **Positive edge triggering of *D* type flip-flop through input lock out**

8.8 *JK* MASTER-SLAVE FLIP-FLOPS

Figure 8.28 shows one way to build a *JK* master-slave flip-flop. Here's how it works.

1. To begin with, the master is positive-level-triggered and the slave is negative-level-triggered. Therefore, the master responds to its J and K inputs before the slave. If $J = 1$ and $K = 0$, the master sets on the positive clock transition. The high Q output of the master drives the J input of the slave, so on the negative clock transition, the slave sets, copying the action of the master.
2. If $J = 0$ and $K = 1$, the master resets on the PT of the clock. The high $\overline{Q}$ output of the master goes to the K input of the slave. Therefore, the NT of the clock forces the slave to reset. Again, the slave has copied the master.
3. If the master's J and K inputs are both high, it toggles on the PT of the clock and the slave then toggles on the clock NT. Regardless of what the master does, therefore, the slave copies it: if the master sets, the slave sets; if the master resets, the slave resets.
4. If $J = K = 0$, the flip-flop is disabled and Q remains unchanged.

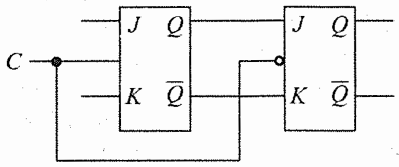
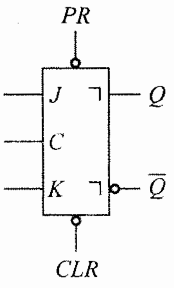


Fig. 8.28 Master-slave flip-flop

The symbol for a 7476 master-slave flip-flop is shown in Fig. 8.29. Either it can be preset to $Q = H$ by taking PR low, or it can be reset to $Q = L$ by taking CLR low. These two inputs take precedence over all other signals!

There is something different however. First of all, notice that the clock (C) is not edge-triggered. The master does in fact change state when C goes high. However, while the clock is high, any change in J or K will immediately affect the master flip-flop. In other words, the master is transparent while the clock is high, and thus J and K must be static during this time. The truth table in Fig. 8.29b reveals this action by means of the pulse symbol (⎍).



(a) Symbol

C	J	K	Q_{n+1}	Action
⎍	L	L	Q_n	No change
⎍	L	H	L	RESET
⎍	H	L	H	SET
⎍	H	H	$\overline{Q}_n$	Toggle

(b) Truth table

Fig. 8.29 7476 JK master flip-flop.

Second, the symbol ⌉ appearing next to the Q and the $\overline{Q}$ outputs is the IEEE designation for a *postponed output*. In this case, it means Q does not change state until the clock makes an NT. In other words, the contents of the master are shifting into the slave on the clock NT, and at this time Q changes state.

To summarize: The master is set according to J and K while the clock is high; the contents of the master are then shifted into the slave (Q changes state) when the clock goes low. This particular flip-flop might be referred to as *pulse-triggered*, to distinguish it from the edge-triggered flip-flops previously discussed.

There are numerous pulse-triggered master-slave flip-flops in use today. However, because edge-triggered flip-flops have overcome the restriction of holding J and K static when the clock is high, most new designs incorporate edge-triggered devices. Some of the more popular pulse-triggered flip-flops you might encounter include the 7473, 7476, and 7478. Their more modern, edge-triggered counterparts include the 74LS73A, the 74LS76A, and the 74LS78A.

Example 8.9 The *JK* master-slave flip-flop in Fig. 8.29 has its J and K inputs tied to $+V_{CC}$ and a series of pulses (actually a square wave) are applied to its C input. Describe the waveform at Q.

Solution Since $J = K = 1$, the flip-flop simply toggles each time the clock goes low. The wave-form at Q has a period twice that of the C waveform. In other words, the frequency of Q is only one-half that of C. This circuit acts as a frequency divider—the output frequency is equal to the input frequency divided by 2. Note that Q changes state on NTs of the clock. The waveforms are given in Fig. 8.30.

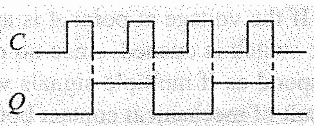


Fig. 8.30

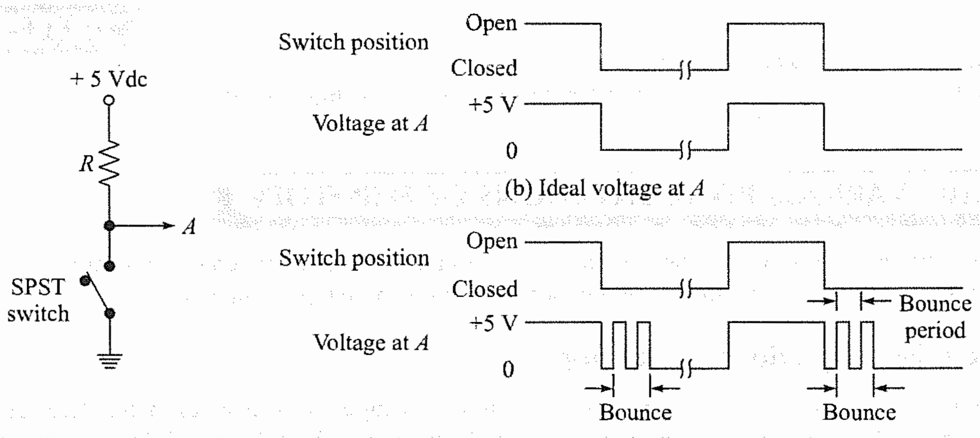
SELF-TEST

14. What is the main difference between an edge-triggered and a pulse-triggered *JK* flip-flop?
15. Explain the operation of the master-slave flip-flop in Fig. 8.29.

8.9 SWITCH CONTACT BOUNCE CIRCUITS

In nearly every digital system there will be occasion to use mechanical contacts for the purpose of conveying an electrical signal; examples of this are the switches used on the keyboard of a computer system. In each case, the intent is to apply a high logic level (usually +5 Vdc) or a low logic level (0 Vdc). The single-pole–single-throw (SPST) switch shown in Fig. 8.31a is one such example. When the switch is open, the voltage at point A is +5 Vdc; when the switch is closed, the voltage at paint A is 0 Vdc. Ideally, the voltage waveform at A should appear as shown in Fig. 8.31b as the switch is moved from open to closed, or vice versa.

In actuality, the waveform at point A will appear more or less as shown in Fig. 8.31c, as the result of a phenomenon known as *contact bounce*. Any mechanical switching device consists of a moving contact arm restrained by some sort of a spring system. As a result, when the arm is moved from one stable position to the other, the arm bounces, much as a hard ball bounces when dropped on a hard surface. The number of bounces that occur and the period of the bounce differ for each switching device. Notice carefully that in this particular instance, even though actual physical contact bounce occurs each time the switch is opened or closed, contact bounce appears in the voltage level at point A only when the switch is closed.



(a)

(b) Ideal voltage at A

(c) Voltage at A showing contact bounce

Fig. 8.31

If the voltage at point A is applied to the input of a TTL circuit, the circuit will respond properly when the switch is opened, since no contact bounce occurs. However, when the switch is closed, the circuit will respond as if multiple signals were applied, rather than the single-switch closure intended—the undesired result of mechanical contact bounce. There is a need here for some sort of electronic circuit to eliminate the contact bounce problem.

A Simple *RS* Latch Debounce Circuit

The RS latch in Fig. 8.32 will remove any contact bounce due to the switch. The output (Q) is used to generate the desired switch signal.

When the switch is moved to position H, $R = 0$ and $S = 1$. Bouncing occurs at the S input due to the switch. The flip-flop "sees" this as a series of high and low inputs, settling with a high level. The flip-flop will immediately be set with $Q = 1$ at the first high level on S. When the switch bounces, losing contact, the input signals are $R = S = 0$, therefore the flip-flop remains set ($Q = 1$). When the switch regains contact, $R = 0$ and $S = 1$; this causes an attempt to again set the flip-flop. But since the flip-flop is already set, no changes occur at Q. The result is that the flip-flop responds to the first, and only to the first, high level at its S input, resulting in a "clean" low-to-high signal at its output (Q).

When the switch is moved to position L, $S = 0$ and $R = 1$. Bouncing occurs at the R input due to the switch. Again, the flip-flop "sees" this as a series of high and low inputs. It simply responds to the *first* high level, and ignores all following transitions. The result is a "clean" high-to-low signal at the flip-flop output. The waveforms in Fig. 8.32b illustrate the behavior.

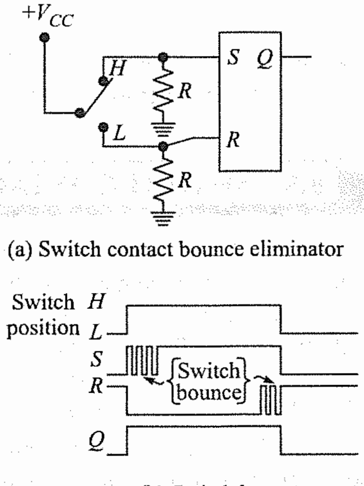


Fig. 8.32 Debounce circuit

SELF-TEST

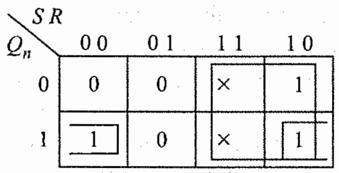
16. What is *switch contact bounce*?
17. Why is switch contact bounce important to account for in a digital system?

8.10 VARIOUS REPRESENTATIONS OF FLIP-FLOPS

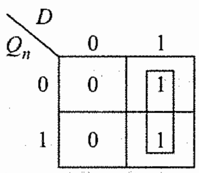
There are various ways a flip-flop can be represented, each one suitable for certain application. Considering basic flip-flop truth table as starting point, this section derives these representations.

Characteristic Equations of Flip-flops

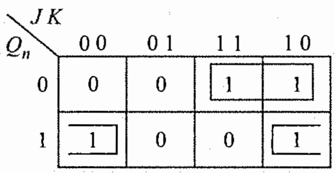
The characteristic equations of flip-flops are useful in analyzing circuits made of them. Here, next output Q_{n+1} is expressed as a function of present output Q_n and input to flip-flops. Karnaugh Map can be used to get the optimized expression and truth table of each flip-flop is mapped into it. This is shown in Fig. 8.33 for all types of flip-flops. The logic equations are presented in SOP form by forming largest group of 1's for each



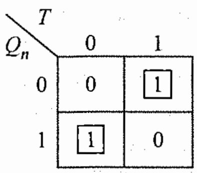
(a) $Q_{n+1} = S + \overline{R}\,Q_n$



(b) $Q_{n+1} = D$



(c) $Q_{n+1} = J\overline{Q}_n + \overline{K}\,Q_n$



(d) $Q_{n+1} = T\overline{Q}_n + \overline{T}\,Q_n$

Fig. 8.33 **Characteristic equations of (a) *SR* flip-flop, (b) *D* flip-flop, (c) *JK* flip-flop, (d) *T* flip-flop**

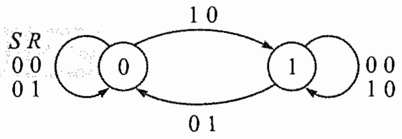
flip-flop. For *SR* flip-flop, since $S = R = 1$ input is not allowed we have don't care states in corresponding locations in Karnaugh Map. This means, it does not matter if Q_{n+1} is 0 or 1 if $SR = 11$ as such a combination at input side will never arise.

The equation for *SR* flip-flop and all others thus can be represented in a summarized form as

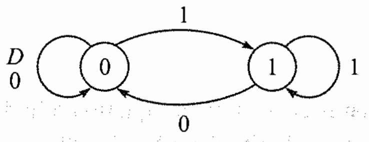
SR flip-flop:	$Q_{n+1} = S + R'Q_n$
JK flip-flop:	$Q_{n+1} = JQ_n' + K'Q_n$
D flip-flop:	$Q_{n+1} = D$
T flip-flop:	$Q_{n+1} = TQ_n' + T'Q_n$

Flip-Flops as Finite State Machine

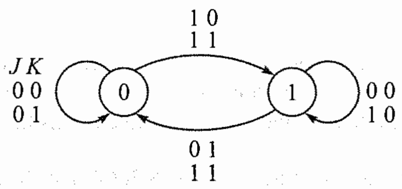
In a sequential logic circuit the value of all the memory elements at a given time define the *state* of that circuit at that time. Finite State Machine (FSM) concept offers a better alternative to truth table in understanding progress of sequential logic with time. For a complex circuit a truth table is difficult to read as its size becomes too large. In FSM, functional behavior of the circuit is explained using finite number of states. State transition diagram is a very convenient tool to describe an FSM. In Fig. 8.34 all the flip-flops are represented as finite state machine through their state transition diagrams.



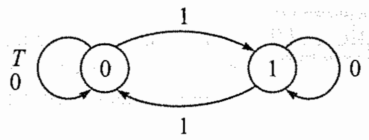
(a) *SR* flip-flop



(b) *D* flip-flop



(c) *JK* flip-flop



(d) *T* flip-flop

Fig. 8.34 **State transition diagram of (a) *SR* flip-flop, (b) *D* flip-flop, (c) *JK* flip-flop, (d) *T* flip-flop**

Let us see how state transition diagram for *SR* flip-flop is developed from its truth table or characteristic equation. Each flip-flop can be at either of 0 or 1 state defined by its stored value at any given time. Application of input may change the stored value, i.e. state of the flip-flop. This is shown by directional arrow and the corresponding input is written alongside. If *SR* flip-flop stores 0, then for $SR = 00$ or 01 the stored value does not change. For $SR = 10$, flip-flop output changes to 1. Note that, $SR = 11$ is not allowed in *SR* flip-flop. When *SR* flip-flop stores 1, application of $SR = 00$ or 10 does not change its value and only when $SR = 01$, output changes to 0. State transitions on application of all possible combination of inputs at every state are shown in Fig. 8.34(a) for *SR* flip-flop. The state transition diagrams are developed in a similar way for *D*, *JK*, *T* flip-flops and are shown in Figs. 8.34 (b), (c), (d) respectively. We see, the timing relation implicit in flip-flop truth tables are brought to the forefront by FSM concept and state transition diagram.

Flip-Flop Excitation Table

In synthesis or design problem excitation tables are very useful and its importance is analogous to that of truth table in analysis problem. Excitation table of a flip-flop is looking at its truth table in a reverse way. Here, flip-flop input is presented as a dependent function of transition $Q_n \rightarrow Q_{n+1}$ and comes later in the table. This is derived from flip-flop truth table or characteristic equation but more directly from its state transition diagram. Figure 8.35 gives a summary presentation of excitation tables of all the flip-flops.

$Q_n \rightarrow$	Q_{n+1}	*S*	*R*	*J*	*K*	*D*	*T*
0	0	0	×	0	×	0	0
0	1	1	0	1	×	1	1
1	0	0	1	×	1	0	1
1	1	×	0	×	0	1	0

Fig. 8.35 **Excitation table of flip-flops**

From Fig. 8.34(a), one can see if present state is 0 application of $SR = 0\times$ does not alter its value where '×' denotes don't care condition in *R* input. State 0 to 1 transition occurs when $SR = 10$ is present at the input side while state 1 to 0 transition occurs if $SR = 01$. Present state 1 is maintained if $SR = 0$, i.e. $SR = 00$ or $SR = 01$. This is shown in Fig. 8.35 along *SR* column. Excitation table for other flip-flops are obtained in a similar way.

Note that, *JK* flip-flop has maximum number of don't care '×' conditions and *D* flip-flop input simply follows the value to which transition is made.

SELF-TEST

18. What is characteristic equation of a flip-flop?
19. What is a Finite State Machine?
20. How is excitation table different from flip-flop truth table?

Example 8.10 A fictitious flip-flop with two inputs *A* and *B* functions like this. For $AB = 00$ and 11 the output becomes 0 and 1 respectively. For $AB = 01$, flip-flop retains previous output while output complements for $AB = 10$. Draw the truth table and excitation table of this flip-flop.

Solution The truth table and corresponding excitation tables are presented in Figs. 8.36(a) and (b) respectively. For $0 \rightarrow 0$ transition we see *AB* need to be 00 or 01. Hence, we write $AB = 0\times$ in that place and similarly for other transitions.

A	B	Q_{n+1}
0	0	0
0	1	Q_n
1	0	$\overline{Q}_n$
1	1	1

(a)

$Q_n \to Q_{n+1}$	A	B
$0 \to 0$	0	×
$0 \to 1$	1	×
$1 \to 0$	×	0
$1 \to 1$	×	1

(b)

Fig. 8.36 Solution for example 8.10: (a) Truth table, (b) Excitation table

8.11 ANALYSIS OF SEQUENTIAL CIRCUITS

A sequential logic circuit contains flip-flops as memory elements and may also contain logic gates as combinatorial circuit elements. Analysis of a circuit helps to explain its performance. We may use truth tables of each building block or corresponding equations for this purpose. In this section, we look at important issues in an analysis problem through an example. In subsequent chapters, more analysis examples will be taken up.

Example 8.11 Consider, the sequential circuit shown in Fig. 8.37. It has only input *CLK* in the form of fixed frequency binary pulses that triggers both the flip-flops. An output X is generated from flip-flop outputs as shown. Analysis of this circuit will give how flip-flop values (or states) and more importantly output X change with input *CLK*. The steps are as follows.

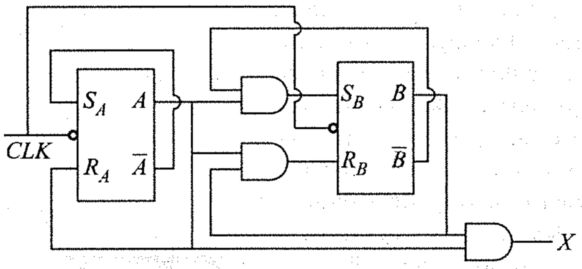


Fig. 8.37 A sequential logic circuit for analysis purpose

Note from the circuit diagram flip-flop input relations: $S_A = A_n'$, $R_A = A_n$ and $S_B = A_nB_n'$, $R_B = A_nB_n$.

Next, using characteristic equation of *SR* flip-flop (Section 8.9) we can write,

for flip-flop A

$$\begin{aligned} A_{n+1} &= S_A + R_A'A_n \\ &= A_n' + A_n'A_n \text{ (Substituting } S_A = A_n' \text{ and } R_A = A_n) \\ &= A_n' \end{aligned}$$

and for flip-flop B

$$\begin{aligned} B_{n+1} &= S_B + R_B'B_n \\ &= A_nB_n' + (A_nB_n)'B_n \text{ (Substituting } S_B = A_nB_n' \text{ and } R_B = A_nB_n) \\ &= A_nB_n' + (A_n' + B_n')B_n \text{ (Following De Morgan's Theorem)} \end{aligned}$$

$$= A_n B_n' + A_n' B_n$$
$$= A_n \oplus B_n$$

Now the output from the given circuit, $X_n = A_n B_n$

The equation shows that present (given by time index n) values of A and B flip-flop, also called *states* of the sequential circuit determine present output and next (given by time index $n + 1$) flip-flop values or state of the circuit. Thus, if present state is $B_n = 0$, $A_n = 0$ then present output $X_n = A_n B_n = 0.0 = 0$ and at the end of first clock cycle we get next state is $B_{n+1} = 0 \oplus 0 = 0$, $A_{n+1} = 0' = 1$. In next clock cycle present state is nothing but next state of previous cycle or $B_n = 0$, $A_n = 1$. The output now is generated as $X_n = 0.1 = 0$ and next state is determined as $B_{n+1} = 0 \oplus 1 = 1$, $A_{n+1} = 1' = 0$. Continuing this exercise we arrive at *state analysis table* also called state table as shown in Table 8.1.

Table 8.1 State Analysis Table for Analysis Example

Present State		*Present Input*				*Next State*		*Present Output*
B_n	A_n	$S_B = A_n B_n'$	$R_B = A_n B_n$	$S_A = A_n'$	$R_B = A_n$	$B_{n+1} = A_n \oplus B_n$	$A_{n+1} = A_n'$	$X = A_n B_n$
0	0	0	0	1	0	0	1	0
0	1	1	0	0	1	1	0	0
1	0	0	0	1	0	1	1	0
1	1	0	1	0	1	0	0	1
0	0	0	0	1	0	0	1	0
0	1	...	...				...	Repeats

We find that the states as well as output of the above circuit repeat after every four clocking periods and at every fourth clock period the output remains 1 for one clock period. The circuit thus behaves like a counter that counts number of clock pulses that has arrived at its input and signals when there is a count of four. A pictorial presentation of the performance of the circuit showing state transitions with each clock is shown in Fig. 8.38. The values within the circle follow syntax: $B_n A_n / X_n$. Flip-flop outputs defining current state is shown to the left of '/' and current output appears at right. Such circuit where output is directly derived from current state only and not from current inputs are called Moore circuit. If current inputs are also used in output forming logic it is called Mealy circuit. More about these are discussed in Chapter 11. Often, a state transition diagram of a sequential circuit serves better than the word description and is presented as final output of an analysis exercise.

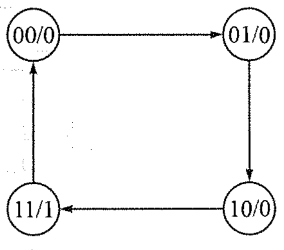


Fig. 8.38 State transition diagram of the sequential circuit given in Fig. 8.37

Analysis of a sequential circuit can also be done through timing diagram where all the input, output and if necessary intermediate variables are plotted against some reference signal say, clock input. The timing diagram obtained by analyzing circuit of Fig. 8.37 is shown in Fig. 8.39. The method followed is given next.

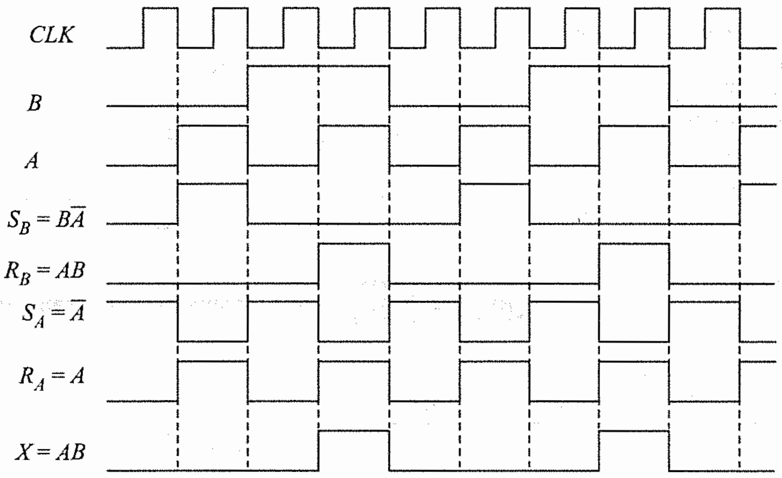


Fig. 8.39 **Timing diagram of the circuit given in Fig. 8.36**

We start with an initial state $B = 0, A = 0$ and note that this state can only change when negative edge of the clock comes. The next state values of B and A are dependent on current inputs S_B, R_B and S_A, R_A at the time of clock trigger. As done before, these input values are derived following relations given in the circuit diagram, i.e. $S_A = A'$, $R_A = A$ and $S_B = AB'$, $R_B = A_B$ (suffix n can be ignored). For $B = 0, A = 0$ we get $S_A = 1, R_A = 0, S_B = 0$ and $R_B = 1$ and these values can change only when B and A change, i.e. in next clock cycle. Thus above values of SR inputs of two flip-flops continue till next negative edge of the clock. For $S_B = 0$ and $R_B = 1$, at the negative edge of clock B remains at 0 (from truth table of SR flip-flop). Similarly for $S_A = 1$, $R_A = 0$ flip-flop A moves to 1. Thus we get B and A value of next clock cycle. Following above relation we now calculate SR input values of these flip-flops as $S_A = 0$, $R_A = 1$, $S_B = 1$ and $R_B = 0$ and these again remain constant up to next negative edge of the clock. Here as $S_B = 1$ and $R_B = 0$, B moves to 1 and as $S_A = 0$, $R_A = 1$, A moves to 0 and remains constant till next clock trigger. SR inputs are again calculated and this process is continued for subsequent clock cycles. In each of these clock cycles we calculate and draw the output following relation $X = AB$. The timing diagram shows the states get repeated as $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$, and so on. Repetition occurs after every fourth clock cycle. The output $X = AB$, accordingly shows repetition as $0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 0$ and remains high for one clock period every time flip-flop output becomes $B = 1, A = 1$.

A detailed analysis of various configurations of counter and its timing diagram will be presented in Chapter 10.

SELF-TEST

21. What is analysis of sequential circuit?
22. Which of truth table and excitation table is useful for analysis of a sequential circuit?

Example 8.12 Explain the function of the circuit shown in Fig. 8.40 through state transition diagram.

Solution The D flip-flop input can be written as $D = X \oplus Q_n$ and output $Y = XQ_n'$. Figure 8.41(a) shows the state table and Fig. 8.41(b) its state transition diagram. Note that, the circuit follows Mealy model and at any given state output is generated from input to that state. Thus, outputs are shown by the side of the input in state transition diagram to right of the input and is separated by a '/' sign.

On careful observation, we can see something interesting in above circuit. If we ignore Y, then the D flip-flop block with Ex-OR gate as connected behaves like a T flip-flop where $T = X$.

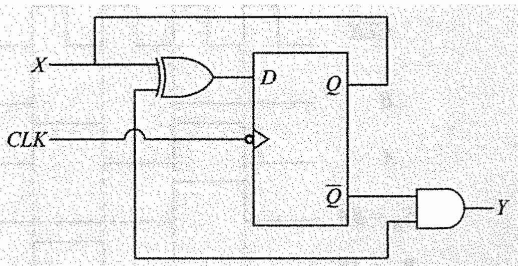
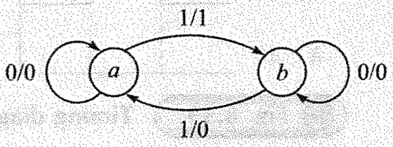


Fig. 8.40 State transition diagram of Example 8.11

Q_n	X	$D = X \oplus Q_n$	Q_{n+1}	$Y = X\overline{Q}_n$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	0
1	1	0	0	0

(a)



(b)

Fig. 8.41 Solution to Example 8.11: (a) State table, (b) State transition diagram

8.12 CONVERSION OF FLIP-FLOPS: A SYNTHESIS EXAMPLE

Knowledge about how flip-flop of one type can be converted to another may be useful on various count. Say, when we have designed the circuit with one type and for implementation we get a different type from the store or the market. Redesign of the problem with available type of flip-flops may take considerable amount of time if the circuit is very complex. Instead one can convert the available type using few basic gate to the type in which design is done and implement the existing design.

Conversion of JK to SR, D, T is fairly straightforward as we see from their respective truth tables or characteristic equations. For example, one need not do anything extra to replace SR flip-flop from a design if SR flip-flop is not available, by JK flip-flop. This is because their truth tables are same except for input combination 11, which in design with SR flip-flop is taken care of not to appear in the input side. Hence, replacing SR with JK flip-flop does not pose any problem. However, the reverse is not true. In design with JK flip-flop there remains possibility of 11 appearing at input side and that combination of input is forbidden for SR flip-flop. Again, comparing truth tables or characteristic equations of JK and D flip-flops we see that putting an inverter from J to K ($K = J'$) we get D flip-flop from JK flip-flop where $J = D$. T flip-flop can be obtained from JK flip-flop by making $T = J = K$.

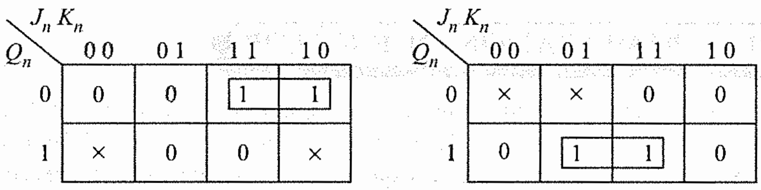
We show here how to convert an SR flip-flop to a JK flip-flop through a systematic approach, as a general methodology for synthesis or design of sequential logic circuit. A detailed study on various design problems and related issues are presented in Chapter 11.

In step one of this method, we look into JK flip-flop truth table and specifically note, $Q_n \rightarrow Q_{n+1}$ transitions for a given combination of inputs JK and present state Q_n. Since the synthesis element is SR flip-flop we shall refer to its excitation table to identify *SR* input combination for a required $Q_n \rightarrow Q_{n+1}$ transition. Table 8.2 shows truth table of *JK* flip-flop as well as necessary *SR* inputs for $Q_n \rightarrow Q_{n+1}$ transitions. Such tables are also known as *state synthesis table*.

Table 8.2 **State Synthesis Table for *SR* to *JK* Flip-Flop Conversion**

J_n	K_n	Q_n	$\rightarrow Q_{n+1}$	S_n	R_n
0	0	0	0	0	×
0	0	1	1	×	0
0	1	0	0	0	×
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	×	0
1	1	0	1	1	0
1	1	1	0	0	1

The next step is to write *SR* inputs as a function of *JK* inputs and present state Q_n. Karnaugh Map derived from Table 8.2 for *SR* inputs are shown in Fig. 8.42 along with their design equations.



$S_n = J_n \overline{Q}_n$ $\qquad$ $R_n = K_n Q_n$

Fig. 8.42 **Karnaugh Map and Design equations for *SR* inputs**

The final synthesized circuit developed from these equations, are shown in Fig. 8.43. The functional block within dotted lines made up of an *SR* flip-flop and two AND gates, behave like a *JK* flip-flop.

Thus conversion between flip-flops, in simple cases can be done comparing their respective truth tables. For other cases, the steps shown above can be followed. Refer to Example 8.13 and Problems 8.30 to 8.32.

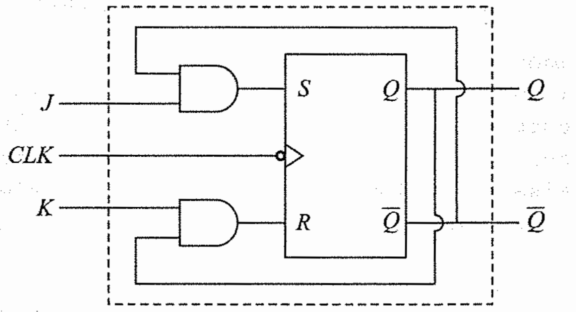


Fig. 8.43 **Conversion of *SR* flip-flop to *JK* flip-flop.**

SELF-TEST

23. Why flip-flop conversion is needed?
24. What is the basic difference between analysis and synthesis steps?
25. What is the difference between state analysis table and state synthesis table?

Example 8.13 Show how a D flip-flop can be converted to SR flip-flop.

Solution Note characteristic equation of two flip-flops.

For SR flip-flop: $Q_{n+1} = S + R'Q_n$ and for D flip-flop: $Q_{n+1} = D$

Thus with $D = S + R'Q_n$ we get circuit shown in Fig. 8.44 which behaves like an SR flip-flop but made from a D flip-flop and basic logic gates. Method as shown in Section 8.11 also gives same solution.

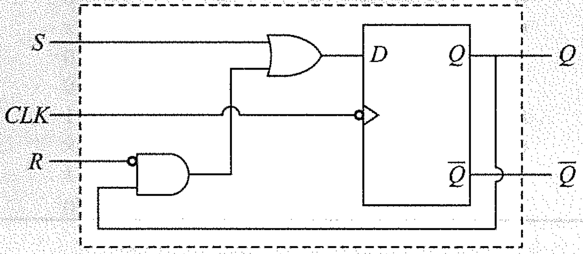


Fig. 8.44 Solution for Example 8.11. *D* flip-flop converted to *SR* flip-flop

8.13 HDL IMPLEMENTATION OF FLIP-FLOP

We continue our discussions on HDL from earlier chapters and in this section we look at how to represent a flip-flop using Verilog HDL. As discussed before, behavioral model is preferred for sequential circuit and **always** keyword is used in all these circuits. Since, sequential logic design also includes combinatorial design at some places we may use dataflow model for that. To start with let us see how a D latch (Fig. 8.15) and SR latch (Fig. 8.11) are expressed in HDL. We have used characteristic equation corresponding flip-flops given in Section 8.9. The explanation of the codes are simple. If EN = 1, output changes according to equation and if EN = 0, output does not change, i.e. remains latched to previous value.

```
module DLatch(D,EN,Q);
input D,EN;
output Q;
reg Q;
always @ (EN or D)
   if (EN) Q=D;

  //from characteristic equation
endmodule
```

```
module SRLatch(S,R,EN,Q);
input S,R,EN;
output Q;
reg Q;
always @ (EN or S or R)
   if (EN) Q=S|(~R&Q);
//from characteristic equation

endmodule
```

Next we discuss how to describe a clocked flip-flop. The following Verilog code describes a D flip-flop with positive edge trigger, negative edge trigger and positive edge trigger with reset (CLR) given in Figs. 8.23 (a), (b) and (c) respectively. Here, the CLR input is active low, i.e. it clears the output ($Q = 0$) when CLR is 0. We use keywords **posedge** and **negedge** for this. With keyword **always** it ensures execution of always block once every clock cycle at corresponding edge. For asynchronous CLR we use a particular nomenclature of Verilog HDL. The **always** sensitivity list (after @) contains any number of edge statements including clock and asynchronous inputs. The **always** block puts all asynchronous conditions in the beginning through else or **else if** and the *last* **else** statement responds to clock transition.

```
module DFFpos(D,C,Q);
input D,C; //C is clock
output Q;
reg Q;
always@(posedge C)
  Q=D;
endmodule
```

```
module DFFneg(D,C,Q);
input D,C; //C is clock
output Q;
reg Q;
always@(negedge C)
  Q=D;
endmodule
```

```
module DFFpos_clr(D,C,CLR,Q);
input D,C,CLR; //C is clock
output Q;
reg Q;
always@(posedge C or negedge CLR)
   if (~CLR) Q=1'b0;
   //Q stores 1 binary bit 0
  else Q=D;
endmodule
```

Example 8.14 Write a Verilog code that converts an *D* flip-flop to an *SR* flip-flop following Fig. 8.43 of Section 8.11.

Solution The code is given as follows. See how combinatorial logic part of the circuit is expressed by **assign** statement.

```
module SRFFneg(S,R,C,Q);
input S,R,C; //C is clock
output Q;
wire DSR;
assign DSR = S|(~R&Q); //combinatorial logic shown in fig.8.45
DFFneg D1(DSR,C,Q);  //instantiates negative edge triggered D FF
endmodule

module DFFneg(D,C,Q);
input D,C; //C is clock
output Q;
reg Q;
always @ (negedge C)
    Q=D;
endmodule
```

Example 8.15 Explain the use of following Verilog code in test bench preparation of sequential logic circuit.

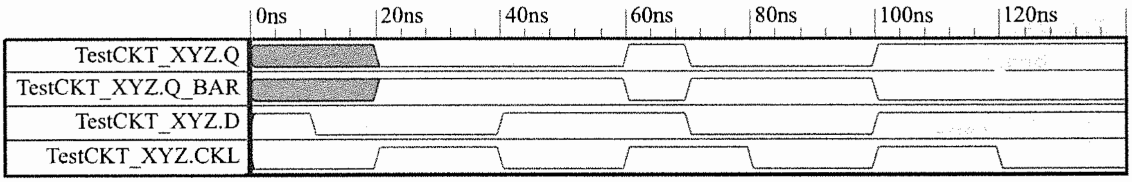
```
Initial
    begin
        clk = 1'b0;
        repeat (20)
        #50 clk = ~clk;
    end
```

Solution The keyword **initial** says following code is run for once. The variable 'clk' is of 1 binary digit and is initialized with 0 at time = 0. Keyword **repeat** ensures repetition of following statement 20 times. In that statement, variable clk is complemented after a delay of 50 ns. Thus, clk toggles between 1 and 0 every 50 ns and for 20 times generating 10 cycles of 50 + 50 = 100 ns duration each. In a test bench, clk can be fed as clock input to simulate a sequential circuit for a finite duration. The number of clock pulse generated can be changed by changing number after **repeat** and clock period can be changed by changing delay after # sign.

Example 8.16 The Verilog code given in first column generates output given in second column and corresponding timing waveform is given next. Draw the digital circuit diagram from Verilog code and explain the output.

```
module CKT_XYZ(Q,Q_BAR,D,CLK);
output Q,Q_BAR;
input D,CLK;
wire X,Y;
nand U1  (X,D,CLK) ;                        0   CLK = 0, D = 1, Q = x
nand U2  (Y,X,CLK) ;                       10   CLK = 0, D = 0, Q = x
nand U3  (Q,Q_BAR,X);                      20   CLK = 1, D = 0, Q = 0
nand U4  (Q_BAR,Q,Y);                      40   CLK = 0, D = 1, Q = 0
endmodule                                  60   CLK = 1, D = 1, Q = 1

module TestCKT_XYZ;                        70   CLK = 1, D = 0, Q = 0
wire Q, Q_BAR;                             80   CLK = 0, D = 0, Q = 0
reg D, CLK;                               100   CLK = 1, D = 1, Q = 1
CKT_XYZ  xyz(Q, Q_BAR, D, CLK);           120   CLK = 0, D = 1, Q = 1
initial
begin
 $monitor($time, "CLK = %b, D= %b,
  Q= %b\n",CLK,D,Q);
  D=1;CLK=0;
  #10 D = 0;    #30 D = 1;   #30 D = 0;
  #30 D = 1;
 #40 $finish; /* the module will terminate after
  140ns*/
end
always
 #20 CLK = ~CLK;
endmodule
```



Solution The circuit diagram from the structural model given in the code is shown in Fig. 8.45. The test bench displays in the monitor time elapsed and CLK, D, Q (in binary) through first statement after **begin**. D initially 1 toggles after a delay of 10ns, 30 ns, 30 ns, 30ns. Simulation stops after further 40 ns taking a total time of 10 + 30 + 30 +30 + 40 = 140ns. Clock toggles at every 20 ns starting with a value 0.

The circuit shows that if CLK = 0, U1 and U2 outputs are 1 irrespective of other inputs and Q, Q_BAR remains latched to previous value through cross coupled U3 and U4. When CLK = 1, D can change U1 output such that $X = D'$ also final output $Q = D$. The timing diagram shows $Q\_BAR = Q'$. Thus the circuit behaves like a high level triggered D Flip-Flop.

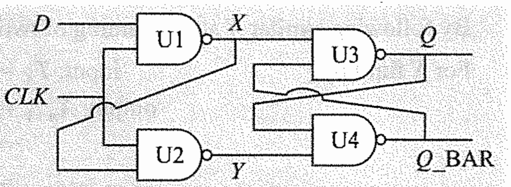


Fig. 8.45 Circuit diagram of Verilog code given in Example 8.16

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Analyze the circuit shown in Fig. 8.46 and find the output Y. Consider that the flip-flops are initially reset.

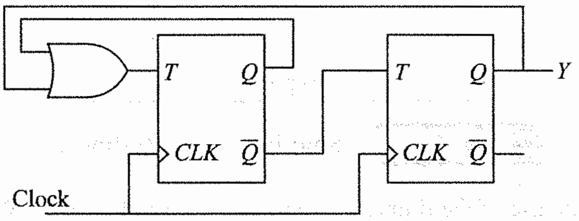


Fig. 8.46 A T flip-flop based circuit for analysis purpose

Solution We follow three different methods to analyze the circuit and identify the performance of Y.

In Method-1, we use state table approach. We make use of the fact that a T flip-flop does not change its state if $T = 0$ but it toggles when $T = 1$ at the clock trigger.

Let us name the first flip-flop as X and its input and output as T_X and X respectively. Similarly, let the second flip-flop be named Y and its input is T_Y while its output is already assigned as Y. Then, the state table is shown in Fig. 8.47. We find that the circuits move from states 00, 01, 10, 00, ... repetitively and the output Y goes HIGH once in three cycles and remains HIGH for one clock period.

Clock cycle	X_n	Y_n	$T_X = X_n + Y_n$	$T_Y = X_n'$	X_{n+1}	Y_{n+1}
0	0	0	0	1	0	1
1	0	1	1	1	1	0
2	1	0	1	0	0	0
3	0	0	... repeats ...			

Fig. 8.47 State Table to analyze circuit diagram of Fig. 8.47

In Method-2, we make use of the characteristic equation of T flip-flop.

From Section 8.9, we know that $Q_{n+1} = TQ_n' + T'Q_n$

By following similar X and Y naming of two flip-flops as in Method-1, we find that

For X flip-flop, input: $T_X = X_n + Y_n$

$$\begin{aligned}\text{output: } X_{n+1} &= (X_n + Y_n)X_n' + (X_n + Y_n)'X_n \\ &= Y_nX_n' + X_n'Y_n'X_n \\ &= Y_nX_n'\end{aligned}$$

For Y flip-flop, input: $T_Y = X_n'$

$$\begin{aligned}\text{output: } Y_{n+1} &= X_n'Y_n' + (X_n')'Y_n \\ &= X_n'Y_n' + X_nY_n\end{aligned}$$

The final solution is shown in Fig. 8.48.

X_n	Y_n	$X_{n+1} = Y_nX_n'$	$Y_{n+1} = X_n'Y_n' + X_nY_n$
0	0	0	1
0	1	1	0
1	0	0	0
0	0	… repeats	…

Fig. 8.48 Solution using Method-2

In Method-3, we make use of the timing diagram as shown in Fig. 8.49. We note that the flip-flops are positive edge triggered. The T input just before the positive edge decides output of the flip-flop in next cycle.

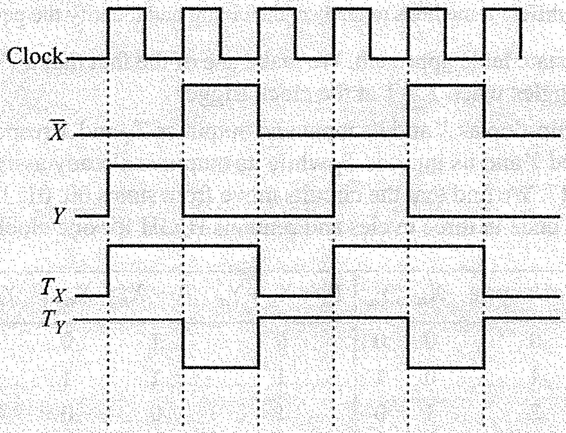


Fig. 8.49 Solution using Method-3

We start with initial $XY = 00$. Then we draw T_X by ORing X and Y waveforms and T_Y by inverting Y waveform. T_X and T_Y before positive edge decide value of X and Y respectively in next clock cycle (from T flip-flop truth table).

SUMMARY

A flip-flop is an electronic circuit that has two stable states. It is said to be bistable. A basic *RS* flip-flop, or latch can be constructed by connecting two NAND gates or two NOR gates in series with a feedback connection. A signal at the set input of an *RS* flip-flop will force the *Q* output to become a 1, while a signal at the reset input will force *Q* to become a 0.

A simple *RS* flip-flop or latch is said to be transparent—that is, its output changes state whenever a signal appears at the *R* or *S* inputs. An *RS* flip-flop can be modified to form a clocked *RS* flip-flop whose output can change states only in synchronism with the applied clock.

An *RS* flip-flop can also be modified to form a *D* flip-flop. In a *D* latch, the stored data may be changed while the clock is high. The last value of *D* before the clock returns low is the data that is stored. With edge-triggered *D* flip-flops, the data is sampled and stored on either the positive or negative clock edge.

The values of *J* and *K* determine what a *JK* flip-flop does on the next clock edge. When both are low, the flip-flop retains its last state. When *J* is low and *K* is high, the flip-flop resets. When *J* is high and *K* is low, the flip-flop sets. When both are high, the flip-flop toggles. In this last mode, the *JK* flip-flop can be used as a frequency divider.

There are various ways to represent a flip-flop like truth table, characteristic equation, state transition diagram or excitation table. Flip-flop treated as a finite state machine highlights its functional aspect. Analysis of a sequential circuit helps to understand performance of a given circuit in a systematic manner and through synthesis we develop circuit diagram for a specified problem.

GLOSSARY

- ***asynchronous*** Independent of clocking. The output can change without having to wait for a clock pulse.
- ***bistable*** Having two stable states.
- ***bistable multivibrator*** Another term for an *RS* flip-flop.
- ***buffer register*** A group of memory elements, often flip-flops, that can store a binary word.
- ***characteristic equation*** logic expression describing a flip-flop.
- ***edge triggering*** A circuit responds only when the clock is in transition between its two voltage states.
- ***finite state machine*** functional description of sequential circuit.
- ***flip-flop*** An electronic circuit that has two stable states.
- ***hold time*** The minimum amount of time that data must be present after the clock trigger arrives.
- ***latch*** Another term for an *RS* flip-flop.
- ***Mealy model*** output is dependent both on curıent state and input to the circuit.
- ***Moore model*** output is dependent only on current state of the circuit.
- ***propagation delay*** The amount of time it takes for the output to change states after an input trigger.
- ***setup time*** The minimum amount of time required for data inputs to be present before the clock arrives.
- ***state*** the set of memory values at any given time for a sequential logic circuit.
- ***synchronous*** When outputs change states in time with a clock. A clock signal must be present in order for the outputs to change states.
- ***transparent*** The condition that exists when the flip-flop output changes immediately after its inputs (*R*, *S*, *J*, *K*, *D*) change state.

PROBLEMS

Section 8.1

8.1 List as many bistable devices as you can think of—either electrical or mechanical. (***Hint:*** Magnets, lamps, relays, etc.)

8.2 Redraw the NOR-gate flip-flop in Fig. 8.3b and label the logic level on each pin for $R = S = 0$. Repeat for $R = S = 1$, for $R = 0$ and $S = 1$, and for $R = 1$ and $S = 0$.

8.3 Redraw the NAND-gate flip-flop in Fig. 8.7a and label the logic level on each pin for $\overline{R} = \overline{S} = 0$. Repeat for $\overline{R} = \overline{S} = 1$, for $\overline{R} = 1$, and $\overline{S} = 0$, and for $\overline{R} = 0$ and $\overline{S} = 1$.

8.4 Redraw the NAND-gate flip-flop in Fig. 8.8a and label the logic level on each pin for $R = S = 0$. Repeat for $R = S = 1$, for $R = 0$ and $S = 1$, and for $R = 1$ and $S = 0$.

Section 8.2

8.5 The waveforms in Fig. 8.50 drive the clocked *RS* flip-flop in Fig. 8.11. The clock signal goes from low to high at points *A*, *C*, *E*, and *G*. If *Q* is low before point *A* in time:

a. At what point does *Q* become a 1?
b. When does *Q* reset to 0?

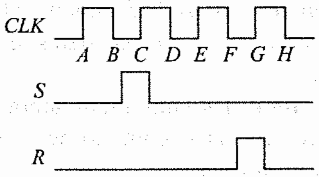


Fig. 8.50

8.6 Use the information in the preceding problem and draw the waveform at *Q*.

8.7 Prove that the flip-flop realizations in Fig. 8.12 are equivalent by writing the logic level present on every pin when $R = S = 0$ and the clock is high. Repeat for $R = S = 1$, for $R = 1$ and $S = 0$, and for $R = 0$ and $S = 1$. Describe what happens when the clock is low.

8.8 The waveforms in Fig. 8.51 drive a *D* latch as shown in Fig. 8.15. What is the value of *D* stored in the flip-flop after the clock pulse is over?

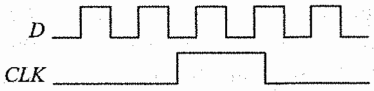


Fig. 8.51

Section 8.3

8.9 What is the advantage offered by an edge-triggered *RS* flip-flop over a clocked or gated *RS* flip-flop?

8.10 The waveforms in Fig. 8.18d illustrate the typical operation of an edge-triggered *RS* flip-flop. This circuit was connected in the laboratory, but the *R* and *S* inputs were mistakenly reversed. Draw the resulting waveform for *Q*.

8.11 An edge-triggered *RS* flip-flop will be used to produce the waveform *Q* with respect to the clock as shown in Fig. 8.52a. First, would you use a positive-edge- or a negative-edge-triggered flip-flop? Why? Draw the waveforms necessary at *R* and *S* to produce *Q*.

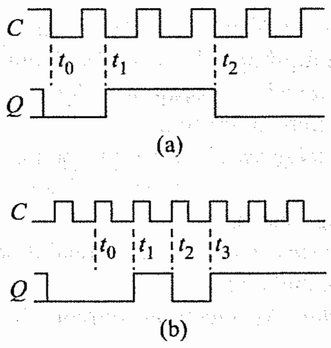


Fig. 8.52

8.12 An edge-triggered *RS* flip-flop will be used to produce the waveform *Q* with respect to the clock as shown in Fig. 8.52b. First, would you use a positive-edge- or a negative-edge-triggered flip-flop? Why? Draw the waveforms necessary at R and S to produce Q.

Section 8.4

8.13 A positive-edge-triggered D flip-flop has the input waveforms shown in Fig. 8.51. What is the value of Q after the clock pulse?

8.14 A negative-edge-triggered D flip-flop is driven by the waveforms shown in Fig. 8.51. What is the value of D stored in the flip-flop?

8.15 A D flip-flop has the following data sheet information: setup time = 5 ns; hold time = 10 ns; propagation time = 15 ns.

a. How far ahead of the triggering clock edge must the data be applied?
b. How long after the clock edge must the data be present to ensure correct storage?
c. How long after the clock edge before the output changes?

Section 8.5 and Section 8.6

8.16 Redraw the JK flip-flop in Fig. 8.23a. Connect $J = K = 1$. (This can be done by connecting the J and K inputs to $+V_{CC}$) Now, begin with $Q = 1$, and show what logic level results on each pin after one positive clock pulse. Allow one more positive clock pulse and show the resulting logic level on every pin.

8.17 In the JK flip-flop in Fig. 8.25a, $J = K = 1$. A 1-MHz square wave is applied to its C input. It has a propagation delay of 50 ns. Draw the input square wave and the output waveform expected at Q. Be sure to show the propagation delay time.

8.18 Repeat Prob. 8.17, but use the flip-flop in Fig. 8.25c.

8.19 In Prob. 8.17, what is the period of the clock? What are the period and frequency of the output waveform at Q?

8.20 Repeat Prob. 8.17, assuming that the C input has a frequency of 10 MHz.

Section 8.7

8.21 Draw two flip-flops like the one shown in Fig. 8.25b and show how to connect them such that a 500-kHz square wave applied to pin 1 will result in a 125-kHz square wave at pin 11. Give a complete wiring diagram (show each pin connection).

8.22 Explain the meaning of the symbol ⌉ in Fig. 8.29a.

8.23 What is the significance of the symbol ⎍ in the truth table of Fig. 8.29?

8.24 Why do most modern designs incorporate edge-triggered JK flip-flops rather than pulse-triggered JK flip-flops?

Section 8.8

8.25 Show how to use a simple $\overline{R}\,\overline{S}$ latch to eliminate switch contact bounce (see Fig. 8.32a).

8.26 There is contact bounce present with the SPDT switch in Fig. 8.53 just as with the SPST switch discussed in Fig. 8.31. However, the RS latch used in Fig. 8.53 will remove all contact bounce, and V_o will be *high* with the switch in position 1 and *low* with the switch in position 2. Explain exactly how this debounce circuit works. You might use waveforms as an aid. Incidentally, the 54/74279 can be used to construct four of these circuits.

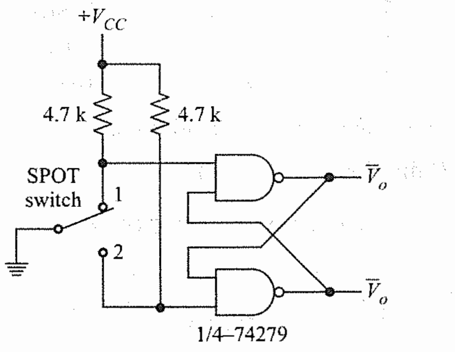


Fig. 8.53 Debounce circuit

Section 8.9 and Section 8.10

8.27 (a) Derive the characteristic equation and (b) draw state transition diagram of the fictitious flip-flop described in Example 8.10.

8.28 Explain the difference between Mealy and Moore model of sequential circuit.

8.29 Analyze the following circuit and explain what it does.

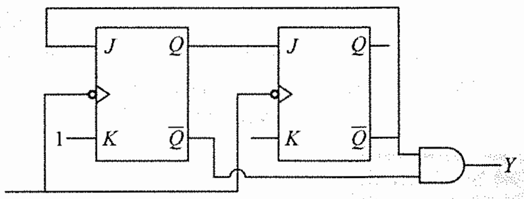


Fig. 8.54

Section 8.11

8.30 Show how to convert D flip-flop to JK flip-flop.

8.31 Convert T flip-flop to D flip-flop.

8.32 Convert SR flip-flop to T flip-flop.

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to study D flip-flop and JK flip-flop and use them for analysis of sequential logic circuits.

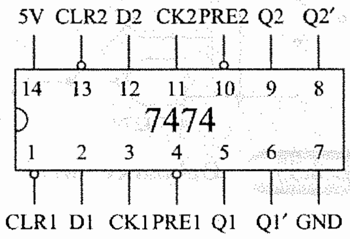
Theory: The truth table of D flip-flop and JK flip-flop are as follows.

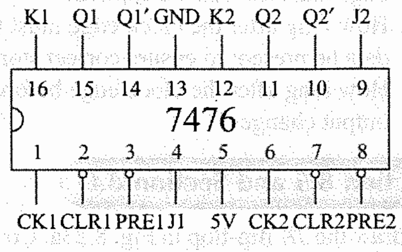
C	D	Q_{n+1}
0	X	Q_n (Last state)
↑	0	0
↑	1	1

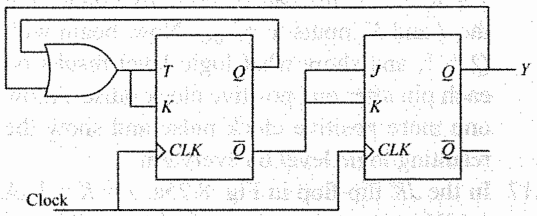
C	J	K	Q_{n+1}	Action
↑	0	0	Q_n (Last state)	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	$\overline{Q}_n$ (toggle)	Toggle

Their characteristic equations are:

D flip-flop: $Q_{n+1} = D_n$







JK flip-flop: $Q_{n+1} = JQ_n' + K'Q_n$

Apparatus: 5 VDC Power supply, Multimeter, Bread Board, Clock Generator, and Oscilloscope

Work element: IC 7474 is a dual, edge clocked, D flip-flop with both PRESET and CLEAR input while 7476 is a dual, edge clocked, JK flip-flop that too, has both PRESET and CLEAR input. Verify the truth table of IC 7474 and 7476. Find if it is positive or negative edge triggered. Appreciate the function of PRESET and CLEAR if it is asynchronous or synchronous with clock. The clock may be available from clock generator or you may use 555 based pulse generator developed in laboratory experiment of previous chapter. Connect 7476 and 7432 (OR gate) as shown, so that the analysis circuit is realized. Use CLEAR to initialize both the flip-flops to 00. Then apply clock, and see the clock and *Y* in a dual trace oscilloscope. Use 7474 to prepare an *SR* flip-flop as shown in Fig. 8.43 and find its truth table.

Answers to Self-tests

1. *R* stands for RESET ($Q = L$). *S* stands for SET ($Q = H$).
2. Quad means "four." There are four flip-flops in this IC.
3. A NAND-gate latch is considered active-low because a low input signal is required to change *Q*.
4. *X* means don't care—this input at this time has no effect.
5. Simply hold the EN input low (at 0 Vdc).
6. The *D* flip-flop is easier to use because it requires only one input (*D*).
7. It means the output responds immediately to input signals.
8. A circuit is activated by the leading edge of the clock.
9. The latch is transparent. The edge-triggered flip-flop only changes state in synchronism with the clock.
10. None. The flip-flop is disabled with *C* held low.
11. PRESET is active high. *A* high level at PRESET will set *Q* high.
12. The *JK* flip-flop has an additional input condition—$J = K = H$. This causes the flip-flop to toggle with the clock. The $R = S = H$ input condition is not allowed with an *RS* flip-flop.
13. Cross-couple the outputs back to the input AND gates.
14. The *J* and *K* inputs are transparent in a pulse-triggered flip-flop. Thus, *J* and *K* must be static while the clock is high.
15. While *C* is high, the master is SET-RESET according to the *J* and *K* inputs. When *C* goes low, the contents of the master shift into the slave, and *Q* is SET-RESET accordingly.
16. Switch contact bounce is the bouncing that occurs when a mechanical, spring-actuated device is operated.
17. The bouncing action produces multiple PTs and NTs, which may introduce unintentional signals!
18. Logic relation showing next state as a function of current state and current inputs.
19. That explains the functional behavior of a sequential circuit through finite number states and its transition from one state to another.
20. It is truth table written in a reverse way such that inputs are shown dependent on a particular state transition.
21. Finding what a given circuit does.
22. Truth table.
23. By this one need not redesign the whole circuit if flip-flop one kind is not available.
24. In analysis, problem begins with a circuit diagram and ends in state transition diagram or performance description. It uses flip-flop truth table or characteristic equation in this process. In synthesis, the path is reverse and we use excitation table instead of truth table.
25. In state analysis table, input of the flip-flops used in the circuit is written first followed by state transition whereas in state synthesis table it is other way.

Registers

9

OBJECTIVES

- ✦ **Understand serial in–serial out shift registers and be familiar with the basic features of the 74LS91 register**
- ✦ **Understand serial in–parallel out shift registers and be familiar with the basic features of the 74164 register**
- ✦ **Understand parallel in–serial out shift registers and be familiar with the basic features of the 74166 register**
- ✦ **Understand parallel in–parallel out shift registers and be familiar with the basic features of the 74174 and 7495A registers**
- ✦ **Understand working of Universal shift register with the basic features of the 74194 register.**
- ✦ **State various uses of shift registers**

A register is a very important digital building block. A data register is often used to momentarily store binary information appearing at the output of an encoding matrix. A register might be used to accept input data from an alphanumeric keyboard and then present this data at the input of a microprocessor chip. Similarly, registers are often used to momentarily store binary data at the output of a decoder. For instance, a register could be used to accept output data from a microprocessor chip and then present this data to the circuitry used to drive the display on a CRT screen. Thus registers form a very important link between the main digital system and the input-output channels. A universal asynchronous receiver transmitter (UART) is a chip used to exchange data in a microprocessor system. The UART is constructed using registers and some control logic.

A binary register also forms the basis for some very important arithmetic operations. For example, the operations of complementation, multiplication, and division are frequently implemented by means of a register. A shift register can also be connected to form a number of different types of counters. Shift registers as sequence generator and sequence detector and also as parallel to serial converters offers very distinct advantages.

The many different applications of registers, along with the myriad of techniques for using them, are simply too numerous to be discussed here. Our intent is to study the detailed operation of the four basic types of shift registers. With this knowledge, you will have the ability to study and understand exactly how a shift register is used in any specific application encountered.

9.1 TYPES OF REGISTERS

A register is simply a group of flip-flops that can be used to store a binary number. "There must be one flip-flop for each bit in the binary number. For instance, a register used to store an 8-bit binary number must have eight flip-flops. Naturally the flip-flops must be connected such that the binary number can be entered (shifted) into the register and possibly shifted out. A group of flip-flops connected to provide either or both of these functions is called a *shift register.*

The bits in a binary number (let's call them the data) can be moved from one place to another in either of two ways. The first method involves shifting the data 1 bit at a time in a serial fashion, beginning with either the most significant bit (MSB) or the least significant bit (LSB). This technique is referred to as *serial shifting*. The second method involves shifting all the data bits simultaneously and is referred to as *parallel shifting.*

There are two ways to shift data into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic register types as shown in Fig. 9.1—serial in–serial out, serial in–parallel out, parallel in–serial out, and parallel in–parallel out. All of these configurations are commercially available as TTL MSI/LSI circuits. For instance:

Serial in–serial out—54/74LS91, 8 bits

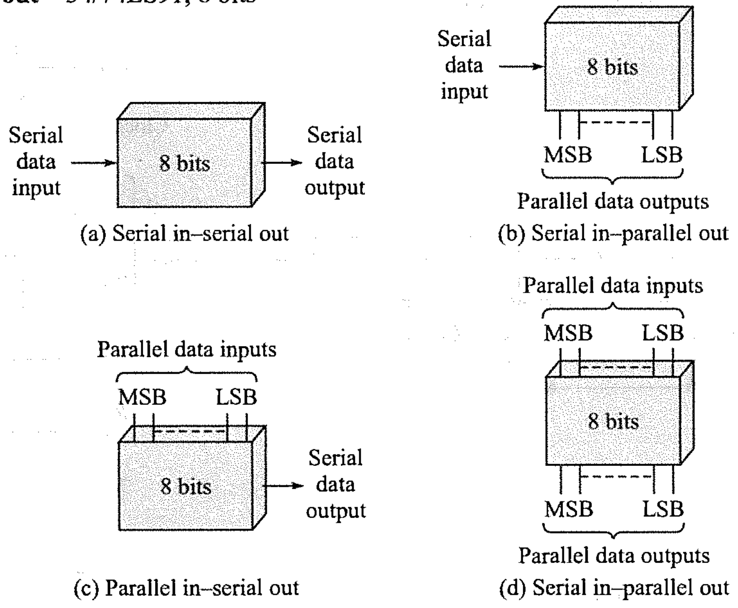


Fig. 9.1 **Shift register types**

Serial in–parallel out—54/74164, 8 bits

Parallel in–serial out—54/74165, 8 bits

Parallel in–parallel out—54/74198, 8 bits

We now need to consider the methods for shifting data in either a serial or parallel fashion. Data shifting techniques and methods for constructing the four different types of registers are discussed in the following sections.

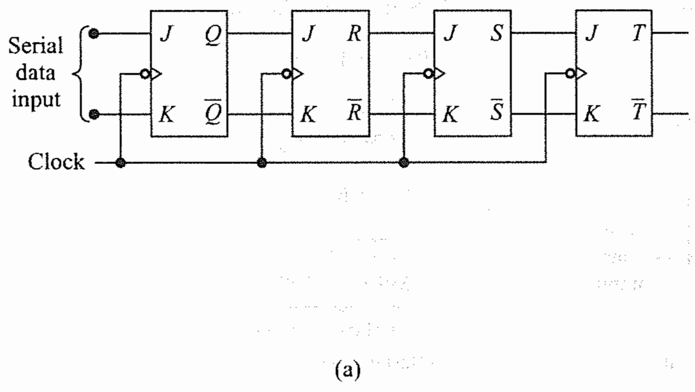
9.2 SERIAL IN–SERIAL OUT

In this section we discuss how data is serially entered or exited from a shift register. The flip-flops used to construct registers are usually edge-triggered *JK*, *SR* or *D* types. We begin our discussion with shift registers made from *D* type flip-flops and then extend the idea to other types.

Consider four *D* flip-flops connected as shown in Fig. 9.2a forming 4-bit shift register. A common clock provides trigger at its negative edge to all the flip-flops. As output of one *D* flip-flop is connected to input of the next at every clock trigger data stored in one flip-flop is transferred to the next. For this circuit transfer takes place like this $Q \rightarrow R$, $R \rightarrow S$, $S \rightarrow T$ and *serial data input* is transferred to *Q*. Let us see how actual data transfer takes place by an example.

Assume, all the flip-flops are initially cleared. Let a binary waveform, as shown along *D* of Fig. 9.2b be fed to *serial data input* of the shift register. Corresponding *Q*, *R*, *S*, *T* are also shown in the figure.

At clock edge *A*, flip-flop *Q* has input 0 from serial data in *D*, flip-flop *R* has input 0 from output of *Q*, flip-flop *S* has input 0 from output of *R* and flip-flop *T* has input 0 from output of *S*. When clock triggers, these inputs get transferred to corresponding flip-flop outputs simultaneously so that *QRST* = 0000. Thus at clock trigger, values at *DQRS* is transferred to *QRST*.



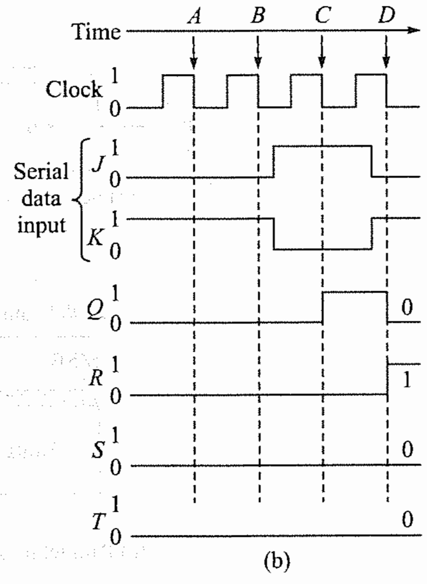


Fig. 9.2 4-bit serial input shift register

At clock edge *B*, serial data in = 0, i.e. $DQRS = 0000$. So after NT at B, $QRST = 0000$. Serial data becomes 1 in next clock cycle.

At clock edge *C*, $DQRS = 1000$ and after NT $QRST = 1000$. Serial data goes to 0 in next clock cycle such that **at clock edge *D***, $DQRS = 0100$ and after NT $QRST = 0100$. Example 9.1 will give another illustration of such data transfer.

A shift register made up of JK or SR flip-flops has non-inverting output Q of one flip-flop connected to J or S input of next flip-flop and inverting output Q' connected to K or R input respectively. For the first flip-flop, between J and K (or S and R) an inverter is connected and J (or S) input is treated as serial data in. Note that, in this configuration both JK and SR flip-flops effectively act like a D flip-flop.

Example 9.1 Show how a number 0100 is entered serially in a shift register shown in Fig. 9.2a using state table.

Solution Figure 9.3 presents the state table. The timing diagram corresponding to this is discussed in this section. Note how the data flow across the flip-flops is highlighted by arrow direction.

Clock	Serial input	Q	R	S	T
0	0	0	0	0	0
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4		0	0	1	0

Fig. 9.3 Data transfer through serial input in a shift register

Example 9.2 Draw the waveforms to shift the number 0100 into the shift register shown in Fig. 9.3 on the next page.

Solution The waveforms for this register will appear exactly as in Fig. 9.2 provided the waveform labeled K is eliminated and waveform J is labeled D.

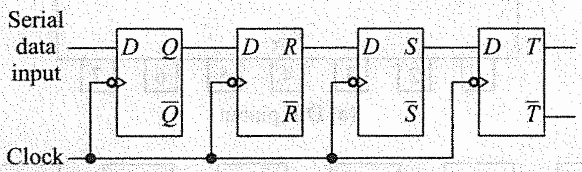


Fig. 9.3a 4-bit serial input shift register

At this point, we have developed the ideas for shifting data into a register in serial form; the serial data input can be classified as either JK or D, depending on the flip-flop type used to construct the register. Now, how about shifting data out of the register?

Let's take another look at the register in Fig. 9.3a, and suppose that it has the 4-bit number $QRST = 1010$ stored in it. If a clock signal is applied, the waveforms shown in Fig. 9.4 will be generated. Here's what happens:

Before Time *A* The register stores the number $QRST = 1010$. The LSB (a 0) appears at T.

At Time *A* The entire number is shifted one flip-flop to the right. A 0 is shifted into Q and the LSB is shifted out the right end and lost. The register holds the bits $QRST = 0101$, and the second LSB (a 1) appears at T.

At Time *B* The bits are all shifted one flip-flop to the right, a 0 shifts into Q, and the third LSB (a 0) appears at T. The register holds $QRST = 0010$.

At Time C The bits are all shifted one flip-flop to the right, a 0 shifts into Q, and the MSB (a 1) appears at T. The register holds $QRST = 0001$.

At Time *D* The MSB is shifted out the right end and lost, a 0 shifts into Q, and the register holds $QRST = 0000$.

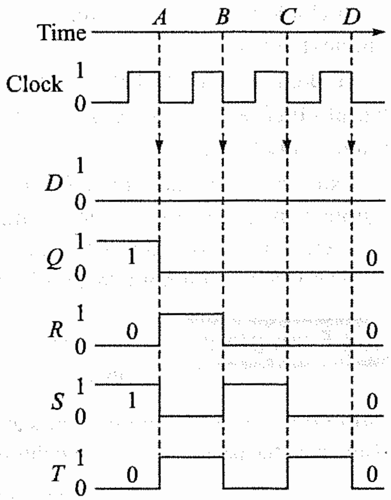
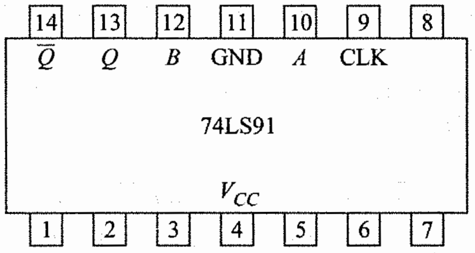


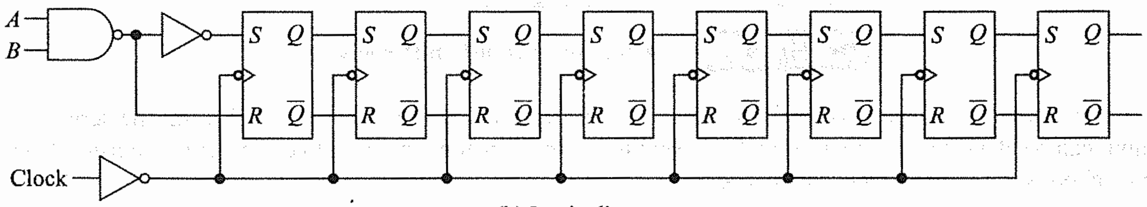
Fig. 9.4

To summarize, we have caused the number stored in the register to appear at T (this is the register output) 1 bit at a time, beginning with the LSB, in a serial fashion, over a time period of four clock cycles. In other words, the data stored was shifted out of the register at flip-flop T in a serial fashion. Thus, not only is this a serial-input shift register, it is also a serial-output shift register. It is important to realize that the stored number is shifted out of the right end of the register and lost after four clock times. Notice that the complement of the output data stream is also available at $\overline{T}$.

The pinout and logic diagram for a 74LS91 shift register are shown in Fig. 9.5. This is an 8-bit TTL MSI chip. There are eight RS flip-flops connected to provide a serial input as well as a serial output.



(a) DIP pinout



(b) Logic diagram

Fig. 9.5 **74LS91 8-bit shift register**

The clock input at each flip-flop is negative-edge-trigger-sensitive. However, since the applied clock signal is passed through an inverter, data will be shifted on the positive edges of the input clock pulses.

The inverter connected between *R* and *S* on the first flip-flop means that this circuit functions as a *D*-type flip-flop. So, the input to the register is a single line on which the data to be shifted into the register appears serially. The data input is applied at either *A* (pin 10) or *B* (pin 12). Notice that a data level at *A* (or *B*) is complemented by the NAND gate and then applied to the *R* input of the first flip-flop. The same data level is complemented by the NAND gate and then complemented again by the inverter before it appears at the *S* input. So, a 1 at input *A* will set the first flip-flop (in other words, this 1 is shifted into the first flip-flop) on a positive clock transition.

The NAND gate with inputs *A* and *B* simply provides a gating function for the input data stream if desired. If gating is not desired, simply connect pins 10 and 12 together and apply the input data stream to this connection.

Example 9.3 Examine the logic levels at the input of a 74LS91 and show how a 1 and then a 0 are shifted into the register.

Solution The input logic and the first flip-flop are redrawn in Fig. 9.6a, and a 1 is applied at the data input *A*. The *R* input is 0, the *S* input is 1, and the flip-flop will clearly be set when the clock goes high. In other words, the 1 at the data input will shift into the flip-flop. In Fig. 9.6b, a 0 is applied at the data input *A*. The *R* input is 1, the *S* input is 0, and the flip-flop will be reset when the clock goes high. The input 0 is thus shifted into the flip-flop.

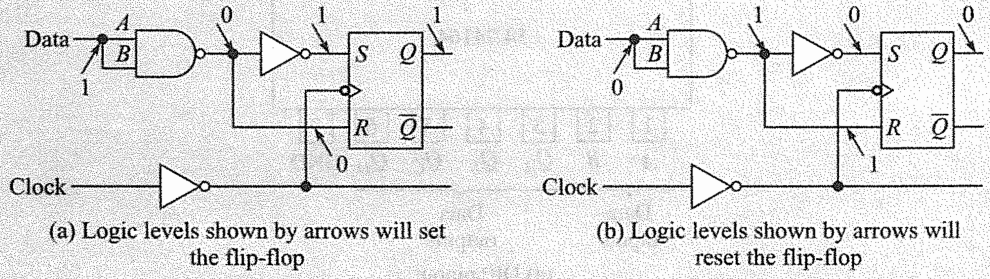

Data
A
B
1
0
1
1
S
Q
R
Q̄
0
Clock
(a) Logic levels shown by arrows will set the flip-flop
Data
A
B
0
1
0
0
S
Q
R
Q̄
1
Clock
(b) Logic levels shown by arrows will reset the flip-flop

Fig. 9.6 Example 9.2

SELF-TEST

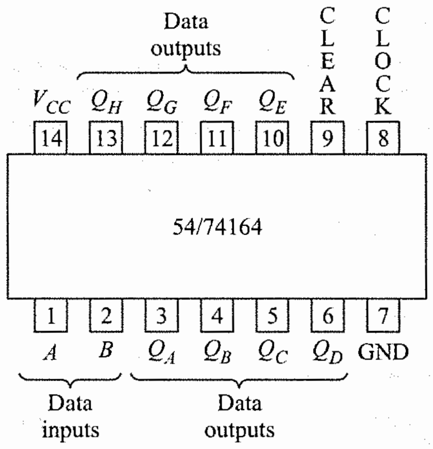
1. What is the largest decimal number that can be stored (in binary form) in a 74LS91 register?
2. Is a 74LS91 register sensitive to PTs or to NTs?

9.3 SERIAL IN–PARALLEL OUT

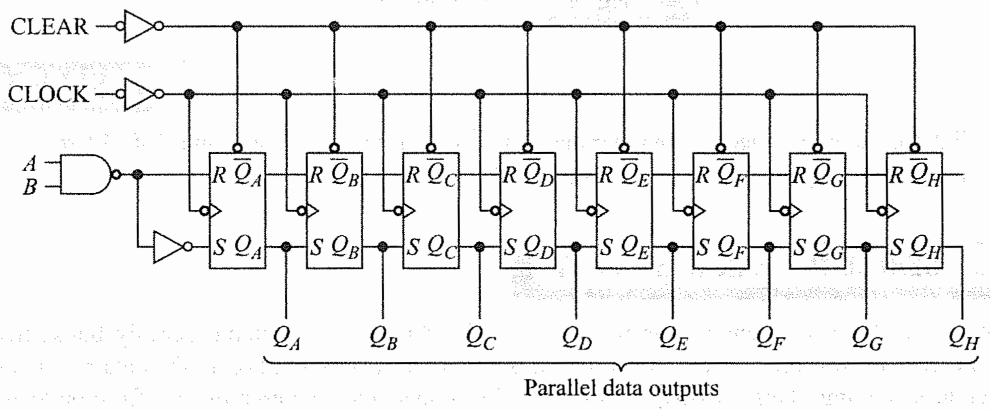
The second type of register mentioned in Sec. 9.1 is one in which data is shifted in serially, but shifted out in parallel. In order to shift the data out in parallel, it is simply necessary to have all the data bits available as outputs at the same time. This is easily accomplished by connecting the output of each flip-flop to an output pin. For instance, an 8-bit shift register would have eight output lines—one for each flip-flop in the register. The basic configuration is shown in Fig. 9.1b.

The 54/74164 is an 8-bit serial input–parallel output shift register. The pinout and logic diagram for this device are given in Fig. 9.7. It is constructed by using *RS* flip-flops having clock inputs that are sensitive to NTs. A careful examination of the logic diagram in Fig. 9.7b will reveal that this register is exactly like the 74LS91 discussed in the previous section—with two exceptions: (1) the true side of each flip-flop is available as an output—thus all 8 bits of any number stored in the register are available simultaneously as an output (this is a parallel data output); and (2) each flip-flop has an asynchronous clear input. Thus a low level at the clear input to the chip (pin 9) is applied through an amplifier and will reset (clear) every flip-flop. Notice that this is an asynchronous signal and can be applied at any time, without regard to the clock waveform and also that this signal is level sensitive. As long as the clear input to the chip is held low, the flip-flop outputs will all remain low. (The register will contain all zeros!)

Shifting data into the register in a serial fashion is exactly the same as the previously discussed 74LS91. Data at the serial inputs may be changed while the clock is either low or high, but the usual setup and hold times must be observed. The data sheet for this device gives setup time as 30 ns minimum and hold time as



(a) DIP pinout



(b) Logic diagram

Fig. 9.7 **54/74164 8-bit shift register**

0.0 ns. Since data are shifted into the register on PTs, the data input line must be stable from 30 ns before the PT until the clock transition is complete.

Let's take a look at the gated serial inputs *A* and *B*. Suppose that the serial data is connected to *A*; then *B* can be used as a control line. Here's how it works:

***B* is Held High** The NAND gate is enabled and the serial input data passes through the NAND gate inverted. The input data is shifted serially into the register.

***B* is Held Low** The NAND-gate output is forced high, the input data stream is inhibited, and the next positive clock transition will shift a 0 into the first flip-flop. Each succeeding positive clock transition will shift another 0 into the register. After eight clock pulses, the register will be full of zeros!

Example 9.4 How long will it take to shift an 8-bit number into a 54164 shift register if the clock is set at 10 MHz?

Solution A minimum of eight clock periods will be required since the data is entered serially. One clock period is 100 ns, so it will require 800 ns minimum.

Example 9.5 For the register in Example 9.4, when must the input data be stable? When can it be changed?

Solution The data must be stable from 30 ns before a positive clock transition until the positive transition occurs. This leaves 70 ns during which the data may be changing (see Fig. 9.8).

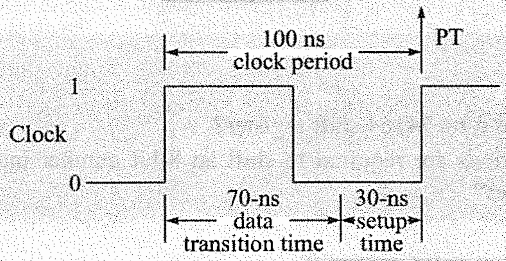


Fig. 9.8 **Example 9.5**

The waveforms shown in Fig. 9.9 show the typical response of a 54/74164. The serial data is input at *A* (pin 1), while a gating control signal is applied at *B* (pin 2). The first clear pulse occurs at time *A* and simply resets all flip-flops to 0.

The clock begins at time *B*, but the first PT does nothing since the control line is low. At time *C* the control line goes high, and the first data bit (a 0) is shifted into the register at time *D*.

The next 7 data bits are shifted in, in order, at times *E*, *F*, *G*, *H*, *I*, *J* and *K*. The clock remains high after time *K*, and the 8-bit number 0010 1100 now resides in the register and is available on the eight output lines. This assumes that the LSB was shifted in first and appears at Q_H. Notice that the clock *must be stopped* after its positive transition at time *K*, otherwise shifting will continue and the data bits will be lost.

Finally, another clear pulse occurs at time *L*, the flip-flops are all reset to zero, and another shift sequence may begin. Incidentally, the register can be cleared by holding the control line at *B* low and allowing the clock to run for eight PTs. This simply shifts eight 0s into the register.

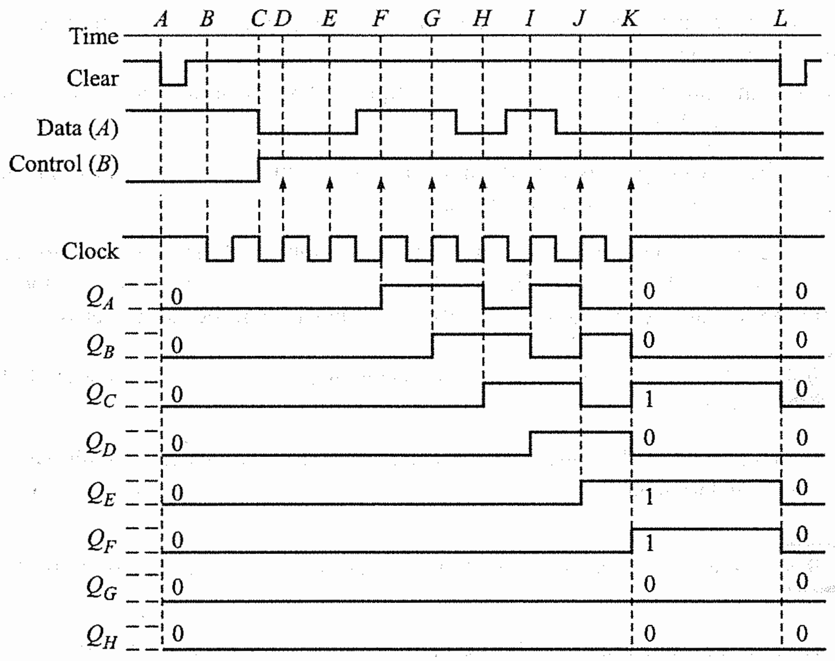


Fig. 9.9

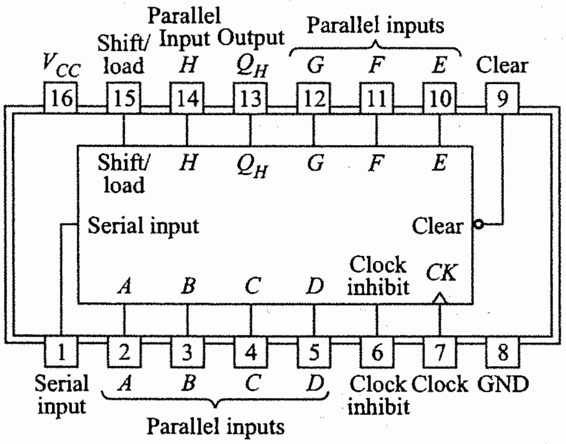
SELF-TEST

3. What is the setup time for a 74164 shift register?
4. How many clock periods are required to shift an 8-bit number into a 74164 register? To extract an 8-bit number?

9.4 PARALLEL IN–SERIAL OUT

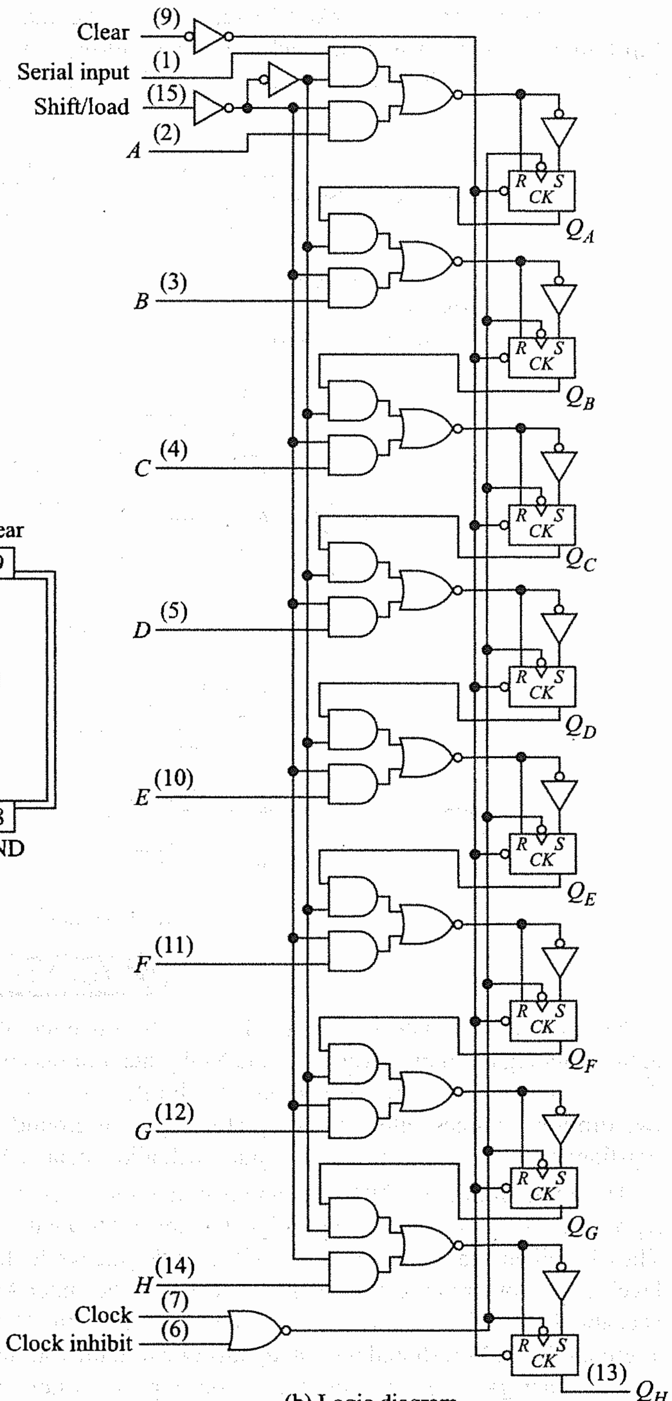
In prior sections, the ideas necessary for shifting data into and out of a register in serial have been developed. We can now use these same ideas to develop methods for the parallel entry of data into a register. There are a number of different techniques for the parallel entry of data, but we shall concentrate our efforts on commercially available TTL. At first glance, the logic diagrams for some of the shift registers seem rather formidable (see, for instance, the block diagram for the 54/74166); but they aren't really. The 54/74166, for instance, is an 8-bit shift register, and the same circuit is repeated eight times. So, it's necessary to study only one of the eight circuits, and that's what we'll do here.

The pinout and logic block diagram for a 54/74166 are given in Fig. 9.10. The functional description given on the TTL data sheet says that this is an 8-bit shift register, capable of either serial or parallel data entry, and serial data output. Notice that there are eight *RS* flip-flops, each with some attached logic circuitry. Let's analyze one of these circuits by starting with the *RS* flip-flops and then adding logic blocks to accomplish our needs.



Positive logic: see description

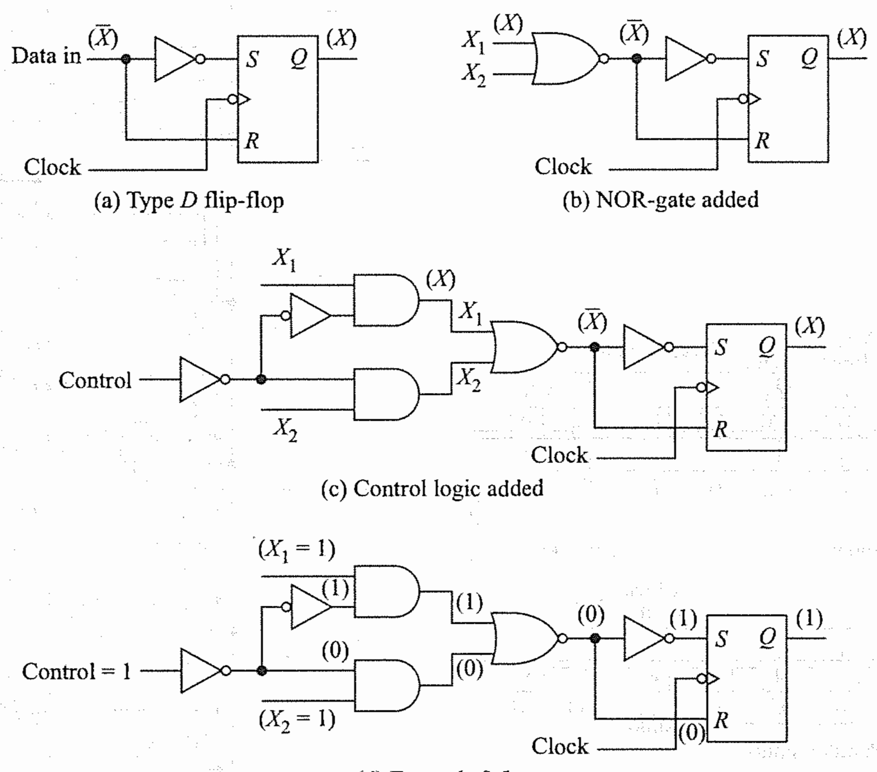
(a) Pinout



(b) Logic diagram

Fig. 9.10 54/74166

First recognize that the clocked *RS* flip-flop and the attached inverter given in Fig. 9.11a form a type *D* flip-flop. If a data bit X is to be clocked into the flip-flop, the complement of X must be present at the input. For instance, if $\overline{X} = 0$, then $R = 0$ and $S = 1$, and a 1 will be clocked into the flip-flop when the clock transitions.



(a) Type *D* flip-flop (b) NOR-gate added (c) Control logic added (d) Example 9.5

Fig. 9.11

Now, add a NOR gate as shown in Fig. 9.11b. If one leg of this NOR gate is at ground level, a data bit X at the other leg is simply inverted by the NOR gate. For instance, if $X = 1$, then at the output of the NOR gate $\overline{X} = 0$, allowing a 1 to be clocked into the flip-flop. This NOR gate offers the option of entering data from two different sources, either X_1 or X_2. Holding X_2 at ground will allow the data at X_1 to be shifted into the flip-flop; conversely, holding X_1 at ground will allow data at X_2 to be shifted in.

The addition of two AND gates and two inverters as shown in Fig. 9.11c will allow the selection of data X_1 or data X_2. If the control line is high, the upper AND gate is enabled and the lower AND gate is disabled. Thus X_1 will appear at the upper leg of the NOR gate while the lower leg of the NOR gate will be at ground level. On the other hand, if the control line is low, the upper AND gate is disabled while the lower AND gate is enabled. This allows X_2 to appear at the lower leg of the NOR gate while the upper leg of the NOR gate is at ground level. You should now study this circuit until your understanding is crystal clear! Consider writing 0 or 1 at each gate leg in response to various inputs. To summarize:

CONTROL is High Data bit at X_1 will be shifted into the flip-flop at the next clock transition.

CONTROL is Low Data bit at X_2 will be shifted into the flip-flop at the next clock transition.

Example 9.6 For the circuit in Fig. 9.11c, write the logic levels present on each gate leg if CONTROL = 1, $X_1 = 1$, and $X_2 = 1$.

Solution The correct levels are given in parentheses in Fig. 9.11d. The data value 1 at X_1 is shifted into the flip-flop when the clock transitions.

A careful examination will reveal that exactly eight of the circuits given in Fig. 9.11c are connected together to form the 54/74166 shift register shown in Fig. 9.10. The only question is: how are they connected? The answer is: they are connected to allow two different operations: (1) the parallel entry of data and (2) the operation of shifting data serially through the register from the first flip-flop Q_A toward the last flip-flop Q_H.

If the data input labeled X_2 in Fig. 9.11c is brought out individually for each flip-flop, these eight inputs will serve as the parallel data entry inputs for an 8-bit number *ABCD EFGH*. These eight inputs are labeled *A, B, C, D, E, F, G,* and *H* in Fig. 9.10. The control line is labeled shift/load. Holding this shift/load control line low will enable the lower AND gate for each flip-flop, and the 8-bit number will be LOADED into the flip-flops with a single clock transition—PARALLEL input.

Holding the shift/load control line high will enable the upper AND gate for each flip-flop. If the input from this upper AND gate receives its data from the prior flip-flop in the register, each clock transition will shift a data bit from one flip-flop into the following flip-flop—proceeding in a direction from Q_A toward Q_H. In other words, data will be shifted through the register serially! In the first flip-flop in the register, the upper AND-gate input is labeled serial input. Thus data can also be entered into this register in a serial fashion. To summarize:

Shift/Load is Low A single clock transition loads 8 bits of data (*ABCD EFGH*) into the register in parallel.

Shift/Load is High Clock transitions will shift data through the register serially, with entering data applied at the SERIAL INPUT.

Notice that the clock is applied through a two-input NOR gate. When clock inhibit is held low, the clock signal passes through the NOR gate inverted. Since the register flip-flops respond to NTs, data will shift into the register on the PTs of the clock. When clock inhibit is high, the NOR-gate output is held low, and the clock is prevented from reaching the flip-flops. In this mode, the register can be made to stop and hold its contents.

A low level at the clear input can be applied at any time without regard to the clock, and it will immediately reset all flip-flops to 0. When not in use, it should always be held high.

The truth table in Fig. 9.12 summarizes the operation of the 54/74166

Inputs						Internal Levels		Outputs
Clear	Shift/ load	Clock inhibit	Clock	Serial	Parallel $A \dots H$	Q_A and Q_B		Q_H
L	X	X	X	X	X	L	L	L
H	X	L	L	X	X	Q_{AO}	Q_{BO}	Q_{HO}
H	L	L	$\uparrow$	X	$a \dots h$	a	b	h
H	H	L	$\uparrow$	H	X	H	Q_{An}	Q_{Gn}
H	H	L	$\uparrow$	L	X	L	Q_{An}	Q_{Gn}
H	X	H	$\uparrow$	X	X	Q_{AO}	Q_{BO}	Q_{HO}

X = Irrelevant, H = High level, L = Low level
$\uparrow$ = Positive transition
$a \dots h$ = Steady state input level at $A \dots H$ respectively
Q_{AO}, Q_{BO} = Level at Q_A, Q_B . . . before steady state
Q_{An}, Q_{Gn} = Level of Q_A or Q_B before most recent transition ( ) $\uparrow$

Fig. 9.12 **54/74166 truth table**

8-bit shift register. You should study this table in conjunction with the logic diagram to understand clearly how the register can be used.

Example 9.7 Which entry in the truth table in Fig. 9.12 accounts for the parallel entry of data?

Solution The third entry from the top; clear is high, shift/load is low, clock inhibit is low, a positive clock transition occurs, and the serial data input is irrelevant.

SELF-TEST

5. What is the purpose of the shift/load line on the 74166?
6. Is data shifted into and out of a 74166 register on clock PTs or on clock NTs?

9.5 PARALLEL IN–PARALLEL OUT

The fourth type of register discussed in the introductory section of this chapter is designed such that data can be shifted either into or out of the register in parallel. In fact, simply adding an output line from each flip-flop in the 54/74166 discussed in the previous section would meet the parallel in–parallel out requirements. [It would, of course, require a larger dual in-line package (DIP)—say, a 24-pin package.]

The 54/74174

The 74174 in Fig. 9.13 is an example of a parallel in–parallel out register. The Texas Instruments data sheet refers to it as a hex *D*-type flip-flop with clear. It is simply a parallel arrangement of six *D*-type flip-flops. Each flip-flop is negative-edge-triggered, and thus a PT will shift data into the register. The six data bits, D_1 through D_6 are all shifted into the register in parallel. The stored data is immediately available, in parallel, at the outputs, Q_1 through Q_6. This type of register is simply used to store data, and is sometimes called a *data register*, or *data latch*. Notice that it is not possible to shift stored data either to the right or to the left. A low level at the clear input will immediately reset all flip-flops low. The clear input is asynchronous—that is, it can be done at any time and it takes precedence over all other inputs.

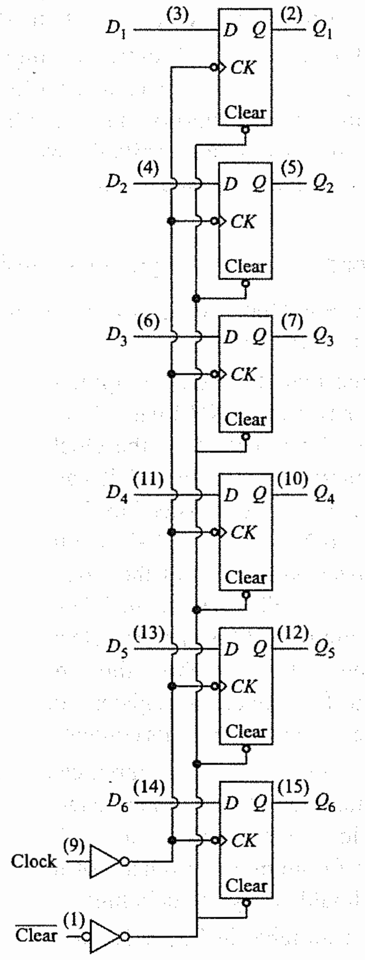


Fig. 9.13 54/74174

Example 9.8 The 74LS174 data sheet gives a setup time of 20 ns and a hold time of 5 ns. What is the minimum required width of the data input levels (D_1 … D_6) for the 74LS174 in Fig. 9.13?

Solution The data inputs must be steady at least 20 ns before the PT of the clock, and they must be held for a minimum of 5 ns after the PT. Thus, the data input levels must be held steady for a minimum of 25 ns (see Fig. 8.24 for comparison).

The 54/74198

The 54/74198 is an 8-bit TTL MSI having both parallel input and parallel output capability. The DIP pinout for this device is given in Fig. 9.14 on the next page. It uses positive edge-triggered flip-flops, as indicated by the small triangle at pin 11. Notice that a 24-pin package is required since 16 pins are needed just for the input and output data lines. Not only does this chip satisfy the parallel input-output requirements; it can also be used to shift data through the register in either direction—referred to as *shift right* and *shift left*. All the registers previously discussed have the ability to shift right, that is, to shift data serially from the data input flip-flop toward the right, or from a flip-flop Q_A toward flip-flop Q_B. We now need to consider how to shift left.

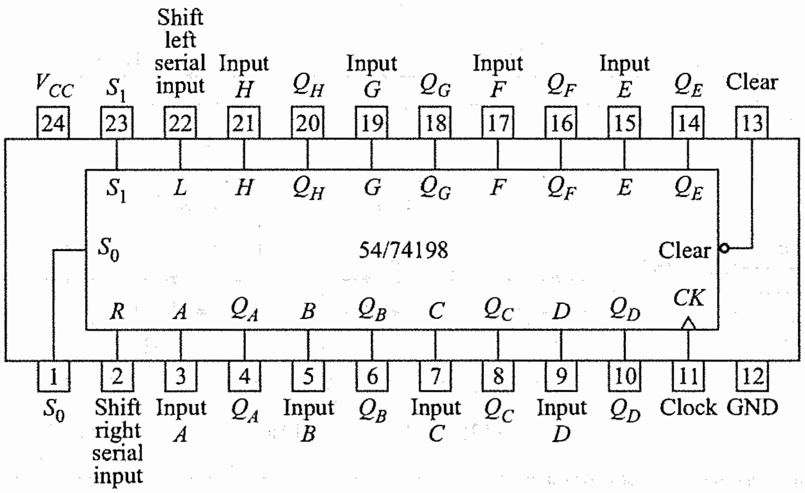
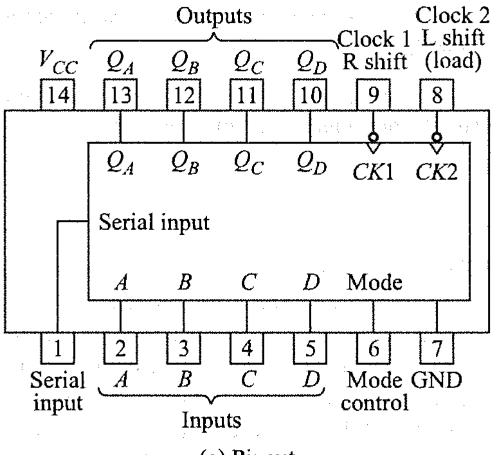


Fig. 9.14 **54/74198, 8-bit shift register. Parallel input–parallel output**

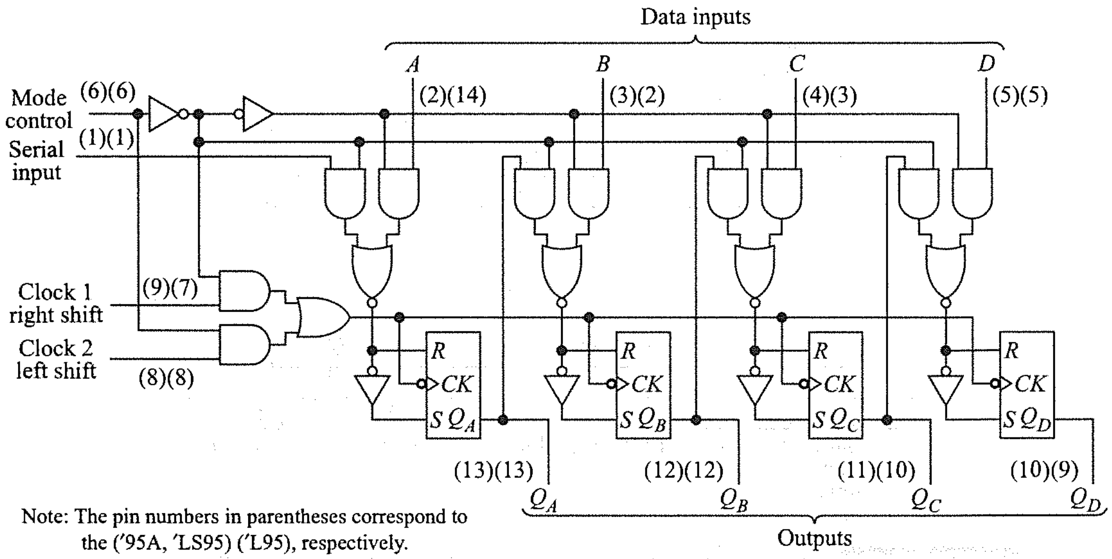
There are a number of 4-bit parallel in–parallel out shift registers available since they can be conveniently packaged in a 16-pin DIP. An 8-bit register can be created by either connecting two 4-bit registers in series or by manufacturing the two 4-bit registers on a single chip and placing the chip in a 24-pin package (such as the 54/74198). Let's analyze a typical 4-bit register, say, a 54/7495A.

The data sheet for the 54/7495A describes it as a 4-bit parallel-access shift register. It also has serial data input and can be used to shift data to the right (from Q_A toward Q_B) and in the opposite direction—to the left. The DIP pinout and logic diagram are given in Fig. 9.15. The basic flip-flop and control logic used here are exactly the same as used in the 54/74164 as shown in Fig. 9.11c.

The parallel data outputs are simply the Q sides of each of the four flip-flops in the register. In fact, note that the output Q_D could be used as a serial output when data is shifted from left to right through the register (right shift).



(a) Pinout



Note: The pin numbers in parentheses correspond to the ('95A, 'LS95) ('L95), respectively.

(b) Logic diagram

Fig. 9.15 **54/7495A**

When the mode control line is held high, the AND gate on the right input to each NOR gate is enabled while the left AND gate is disabled. The data at inputs, A, B, C and D will then be loaded into the register on a negative transition of the clock—this is parallel data input.

When the mode control line is low, the AND gate on the right input to each NOR gate is disabled while the left AND gate is enabled. The data input to flip-flop Q_A is now at serial input; the data input to Q_B is Q_A and so on down the line. On each clock NT, a data bit is entered serially into the register at the first flip-flop Q_A, and each stored data bit is shifted one flip-flop to the right (toward the last flip-flop Q_D). This is the serial input of data (at serial input), and also the right-shift operation.

In order to effect a shift-left operation, the input data must be connected to the D data input as shown in Fig. 9.16 below. It is also necessary to connect Q_D to C, Q_C to B, and Q_B to A as shown in Fig. 9.16. Now, when the mode control line is held high, data bit will be entered into flip-flop Q_D, and each stored data bit will be shifted one flip-flop to the left on each clock NT. This is also serial input of data (but at input D) and is the left-shift operation. Notice that the connections described here can either be hard wired or can be made by means of logic gates.

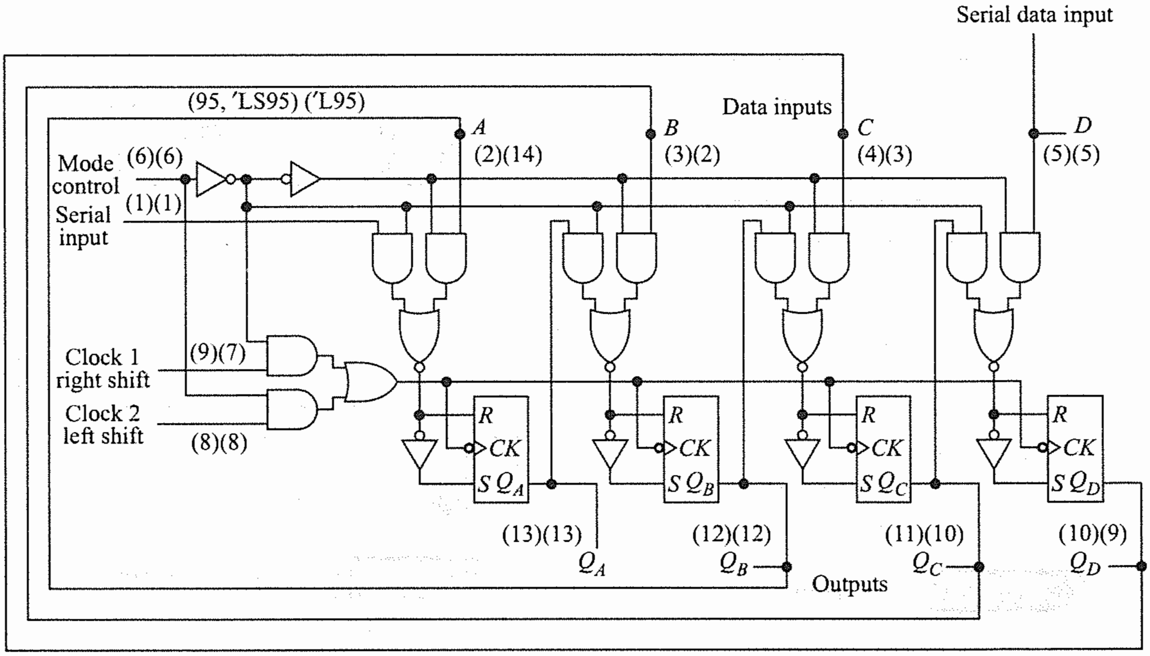


Fig. 9.16 **54/7495A wired for shift left**

There are two clock inputs—clock 1 and clock 2. This is to accommodate requirements where the clock used to shift data to the right is separate from the clock used to shift data to the left. If such a requirement is unnecessary, simply connect clock 1 and clock 2 together. The clock signal will then pass through the AND-OR gate combination noninverted, and the flip-flops will respond to clock NTs.

Example 9.9 Draw the waveforms you would expect if the 4-bit binary number 1010 were shifted into a 54/7495A in parallel.

Solution The mode control line must be high. The data input lines must be stable for more than 10 ns prior to the clock NTs (setup time for the data sheet information). A single clock NT will enter the data. (The waveforms are given in Fig. 9.17.) If the clock is stopped after the transition time T, the levels on the input data lines may be changed. However, if the clock is not stopped, the input data line levels must be maintained.

At this point, it simply cannot be overemphasized that the input control lines to any shift register *must be controlled at all times*! Remember, the register will do something every time there is a clock transition. What it does is entirely dependent on the levels applied at the control inputs. If you do not account for input control levels, you simply cannot account for the behavior of the register!

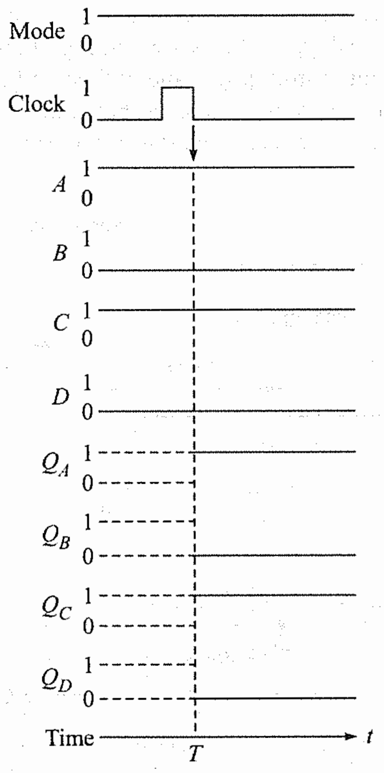


Fig. 9.17 **Example 9.8**

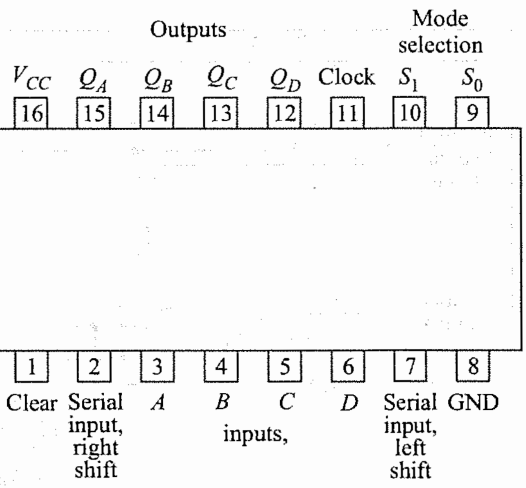


Fig. 9.18 **74194 pinout**

SELF-TEST

7. How can the 7495A, a 4-bit register, be used to store 8-bit numbers?
8. Why does the 7495A have two separate clock inputs?

9.6 UNIVERSAL SHIFT REGISTER

In Section 9.1, we have seen that for basic types of shift register, the following operations are possible—serial in–serial out, serial in–parallel out, parallel in–serial out, and parallel in–parallel out. Serial in or serial out again can be made possible by shifting data in any of the two directions, left shift ($Q_A \leftarrow Q_B \leftarrow Q_C \leftarrow Q_D \leftarrow$ Data in) and right shift (Data in $\rightarrow Q_A \rightarrow Q_B \rightarrow Q_C \rightarrow Q_D$). A universal shift register can perform all the four operations and is also bidirectional in nature. 7495A, described in previous section, is quite versatile except for the fact that it is in-built for right shift; the left shift is achieved through parallel loading (Fig. 9.16) and thus requires external wiring.

The 74194 is a 4-bit universal shift register in 16 pin package with pinout diagram as shown in Fig. 9.18. A, B, C and D are four parallel inputs, and Q_A, Q_B, Q_C and Q_D are corresponding parallel outputs. There are two separate inputs for serial data for left and right shift. In addition, there are two mode control inputs which select the mode of operation for the universal shift register according to Table 9.1. The subscript n and $n + 1$ represent two consecutive states and in between them, there is a clock trigger. In the function table, next state $Q_{A,\ n+1}$ takes the value $Q_{B,n}$ at clock-trigger which means whatever was the value of Q_B at n-th state becomes the value of Q_A at $(n + 1)$-th state.

To understand how this universal shift register is implemented, refer to logic circuit diagram of 74194 in Fig. 9.19. You may identify four 4 to 1 multiplexer blocks in the circuit (one is shown with dotted lines). Two selection inputs of each of these four multiplexers, understandably, are mode selection inputs S_1S_0. For $S_1S_0 = 00$, the second AND gate output which is nothing but the previous value of the corresponding flip-flop is transferred to the output. Thus, the flip-flop output does not change and this is the 'Hold' mode. For $S_1S_0 = 01$, the fourth AND gate output is transferred which corresponds to 'Shift right'. For $S_1S_0 = 10$, the first AND gate output is transferred which corresponds to 'Shift left'. Finally, for $S_1S_0 = 11$, the third AND gate output is selected which effects parallel 'Load' synchronized with clock. The input 'Clear' is active low and resets all the flip-flops asynchronously when activated. Note that the 'Clock' is positive edge-triggered due to two inversions (bubble) in the circuit diagram.

The 74299 is an 8-bit universal shift register in 20 pin package with a similar function table as the 74194. To save number of pins, the input and output pins are made common here. This is achieved by tristating and using additional control input that make these pins bidirectional.

Table 9.1 Function Table of 74194

Mode	*Control*	*Function*	*Next State (n+1-th state)*			
S_1	S_0		$Q_{A,n+1}$	$Q_{B,n+1}$	$Q_{C,n+1}$	$Q_{D,n+1}$
0	0	Hold	$Q_{A,n}$	$Q_{B,n}$	$Q_{C,n}$	$Q_{D,n}$
0	1	Shift right	Data in (Pin 2)	$Q_{A,n}$	$Q_{B,n}$	$Q_{C,n}$
1	0	Shift left	$Q_{B,n}$ (Pin 7)	$Q_{C,n}$	$Q_{D,n}$	Data in
1	1	Load	A	B	C	D

9.7 APPLICATIONS OF SHIFT REGISTERS

Shift registers are used in almost every sphere of a digital logic system. In this section we discuss few such applications. Shift register can be used to count number of pulses entering into a system as ring counter or switched-tail counter. As ring counter it can generate various control signals in a sequential manner. Shift register can also generate a prescribed sequence repetitively or detect a particular sequence from data input. It can also help in reduction of hardware by converting parallel data feed to serial one. Serial adder is one such application discussed in this section.

Ring Counter

Let's begin with a simple serial shift register such as the 54/74164. One of the most logical applications of feedback might be to connect the output of the last flip-flop Q_H back to the D input of the first flip-flop A (Fig. 9.20a). Notice that the A and B data inputs are connected together. Now, suppose that all flip-flops are reset and the clock is allowed to run. What will happen? The answer is, nothing will happen since the D input to the first flip-flop is low (the input at A and B). Therefore, every time the clock goes high, the zero

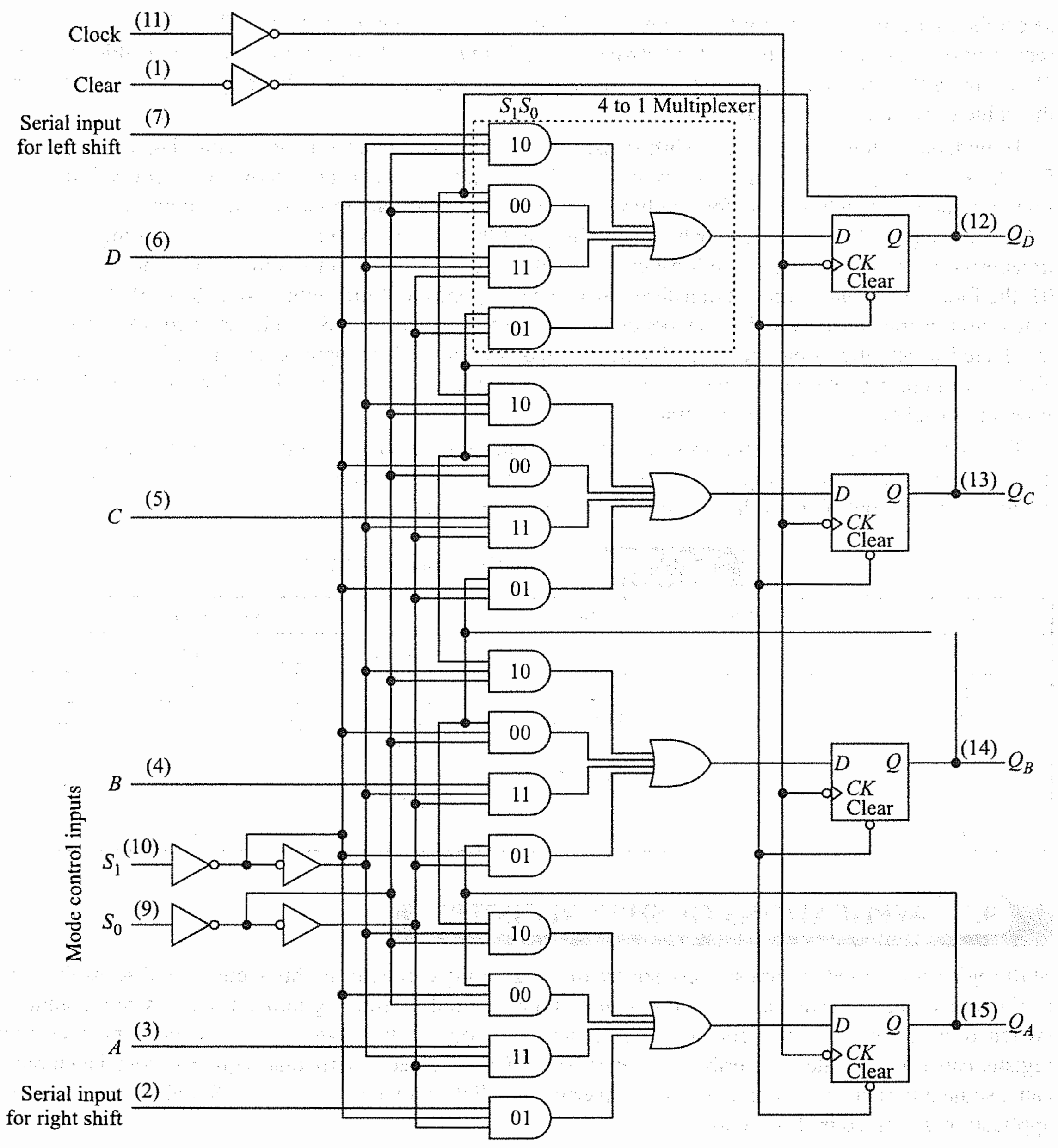
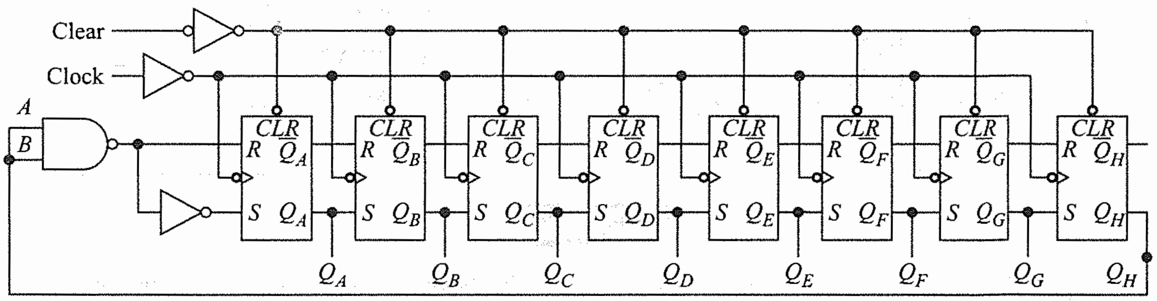
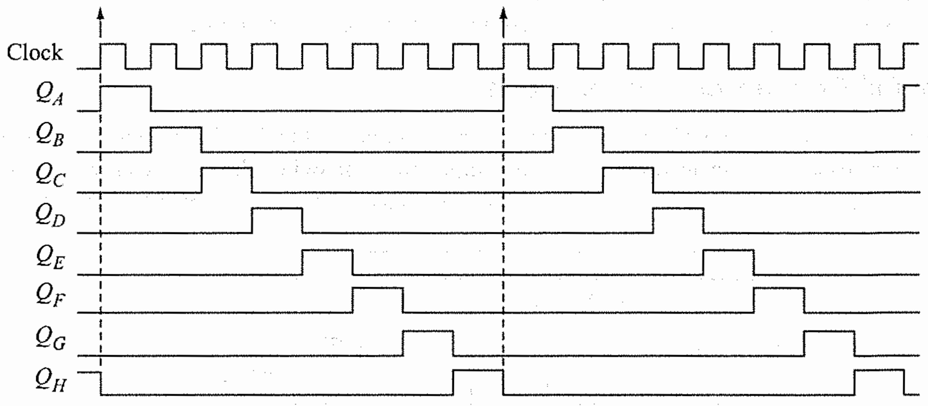


Fig. 9.19 **74194, 4-bit universal shift register**

in each flip-flop will be shifted into the next flip-flop, while the zero in the last flip-flop *H* will travel around the feedback loop and shift into the first flip-flop *A*. In other words, all the flip-flops are in a reset state, each clock PT resets them again, and each flip-flop output simply remains low. Consider the register as a tube full of zeros (ping-pong balls) that shift round and round the register, moving ahead one flip-flop with each clock PT.



(a) 54/74164 8-bit shift register with feedback line from Q_H to A-B



(b) Waveforms when register has a single one, and seven zeros

Fig. 9.20 Ring counter

In an effort to obtain some action, suppose that Q_A is high and all other flip-flops are low, and then allow the clock to run. On the very first clock PT, the 1 in *A* will shift into *B* and *A* will be reset, since the 0 in *H* will shift into *A*. All other flip-flops will still contain 0s. The second clock pulse will shift the 1 from *B* to *C*, while *B* resets. The third clock PT will shift the 1 from *C* to *D*, and so on. Thus this single 1 will shift down the register, traveling from one flip-flop to the next flip-flop each time the clock goes high. When it reaches flip-flop *H*, the next clock PT will shift it into flip-flop *A* by means of the feedback connection. Again, consider the register as a tube full of ping-pong balls, seven "white" ones (0s) and one "black" one (a 1). The ping-pong balls simply circulate around the register in a clockwise direction, moving ahead one flip-flop with each clock PT. This configuration is frequently referred to as a *circulating register* or a *ring counter*. The waveforms present in this ring counter are given in Fig. 9.20b.

Waveforms of this type are frequently used in the control section of a digital system. They are ideal for controlling events that must occur in a strict time sequence—that is, event *A*, then event *B*, then *C*, and so on. For instance, the logic diagram in Fig. 9.21 shows how to generate RESET, READ, COMPLEMENT, and WRITE (a fictitious set of control signals) as a set of control pulses that occur one after the other sequentially. The control signals are simply the outputs of flip-flips *A*, *B*, *D*, and *E* as shown in Fig. 9.20.

There is, however, a problem with such ring counters. In order to produce the waveforms shown in Fig. 9.20, the counter should have one, and only one, 1 in it. The chances of this occurring naturally when power is first applied are very remote indeed. If the flip-flops should all happen to be in the reset state when power

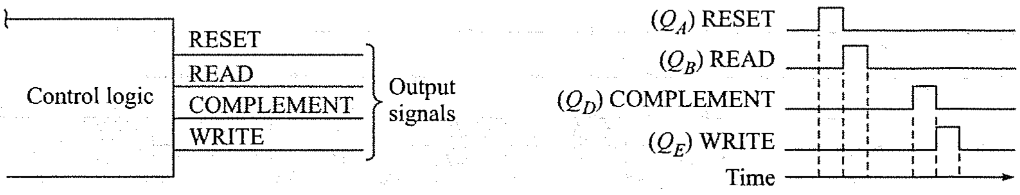
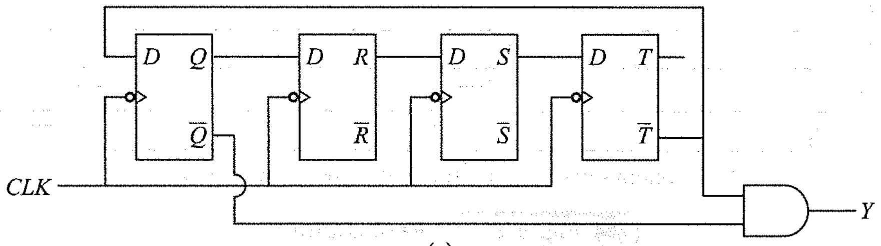


Fig. 9.21

is first applied, it will not work at all, as we saw previously. On the other hand, if some of the flip-flops come up in the set state while the remainder come up in the reset state, a series of complex waveforms of some kind will be the result. Therefore, it is necessary to preset the counter to the desired state before it can be used. Example 9.10 shows one scheme how to do presetting when power is first applied.

Switched-Tail Counter or Johnson Counter

We have seen in ring counter what happens if non-inverting output of the first flip-flop is fed back to first flip-flop of the shift register. If we instead feed inverting output back (or switch the tail) as shown in Fig. 9.22a for a 4-bit shift register we get *switched tail counter*, also known as *twisted tail counter* or *Johnson counter*. The



(a)

Clock	Serial in = T	Q R S T	$Y = Q'T'$
0	1	0 0 0 0	1
1	1	1 0 0 0	0
2	1	1 1 0 0	0
3	1	1 1 1 0	0
4	0	1 1 1 1	0
5	0	0 1 1 1	0
6	0	0 0 1 1	0
7	0	0 0 0 1	0
8	1	0 0 0 0	1
9	1	1 0 0 0	0 repeats

(b)

Fig. 9.22 (a) 4-bit switched tail counter, (b) Its state table circuit is explained through state table similar to Fig. 9.3 of Section 9.2. Assume all the flip-flops are cleared in the beginning. Then all the flip-flop inputs have 0 except the first one, *serial data* in which is complement of the last flip-flop, i.e. 1. When clock trigger occurs flip-flop stores $QRST$ as 1000. This makes 1100 at the input of $QRST$ when the next clock trigger comes and that gets transferred to output at NT. Proceeding this we complete state table of Fig. 9.22b. Note that output $Y = Q'T'$ and state of the circuit repeats every eighth clock cycle. Thus this 4-bit shift register circuit can count 8 clock pulses or called modulo-8 counter.

Following above logic and preparing state table for any N-bit shift register we see switched-tail configuration can count up to $2N$ number of clock pulse and gives modulo-$2N$ counter. The output Y, derived similarly by AND operation of first and last flip-flop inverting outputs gives a logic high at every $2N$-th clock cycle. This two-input AND gate which decodes states repeating in the memory units to generate output that signals counting of a given number of clock pulses is called decoding gate. For switched-tail counter of any modulo number we need only a 2-input AND gate. Observing the state sequences in Fig. 9.22b we find logic relation like $Y = QR'$ or $Y = RS'$ or $Y = ST'$, etc. can also be used for decoding purpose as they generate $Y = 1$ only once during $2N$ clock cycles. Note that for ring counter we don't need any decoding gate and clock pulse count can directly be obtained from any one flip-flop output. We shall discuss other counter design techniques in Chapter 10, which require less number of flip-flops for a particular modulo number. But, there decoding complexity increases with increasing number of flip-flops. For example, a modulo-8 counter is possible to design with $\log_2 8 = 3$ number of flip-flops but we need a 3 input AND gate to decode the counter. Similarly, modulo-16 counter requires 4 flip-flops and 4 input AND gate for decoding.

There is another important issue related with ring counter and switched tail counter. An n-bit register has 2^n different combination of states. But, the counter is to be initialized with one of the valid state of the counting sequence on which the design is based. Otherwise, the counter will follow a completely different state sequence (mutually exclusive) and decoding will not be proper. Solve Problem 9.25 to get an idea on what happens if circuit in Fig. 9.22a is initialized with a word outside the state sequence appearing in Fig. 9.22b.

Sequence Generator and Sequence Detector

Sequence generator is useful in generating a sequence pattern repetitively. It may be the synchronizing bit pattern sent by a digital data transmitter or it may be a control word directing repetitive control task. Sequence detector checks binary data stream and generates a signal when a particular sequence is detected.

Figure 9.23a gives the basic block diagram of a sequence generator where shift register is presented as pipe full of data and each flip-flop represents one compartment of it. The leftmost flip-flop is connected to serial data in and rightmost provides serial data out. The clock is implied and data transfer takes place only when a clock trigger arrives. Note that the shift register is connected like a ring counter and with triggering of clock the binary word stored in the clock comes out sequentially from serial out but does not get lost as it is fed back as serial in to fill the register all over again. Sequence generated for binary word 1011 is shown in the figure and for any n-bit long sequence to be generated for this configuration we need to store the sequence in an n-bit shift register.

The circuit that can detect a 4-bit binary sequence is shown in Fig. 9.23b. It has one register to store the binary word we want to detect from the data stream. Input data stream enters a shift register as serial data in and leaves as serial out. At every clocking instant, bit-wise comparisons of these two registers are done through Ex-NOR gate as shown in the figure. Two input Ex-NOR gives logic high when both inputs are low or both of them are high, i.e. when both are equal. The final output is taken from a four input AND gate, which becomes 1 only when all its inputs are 1, i.e. all the bits are matched. Figure 9.23b shows a situation

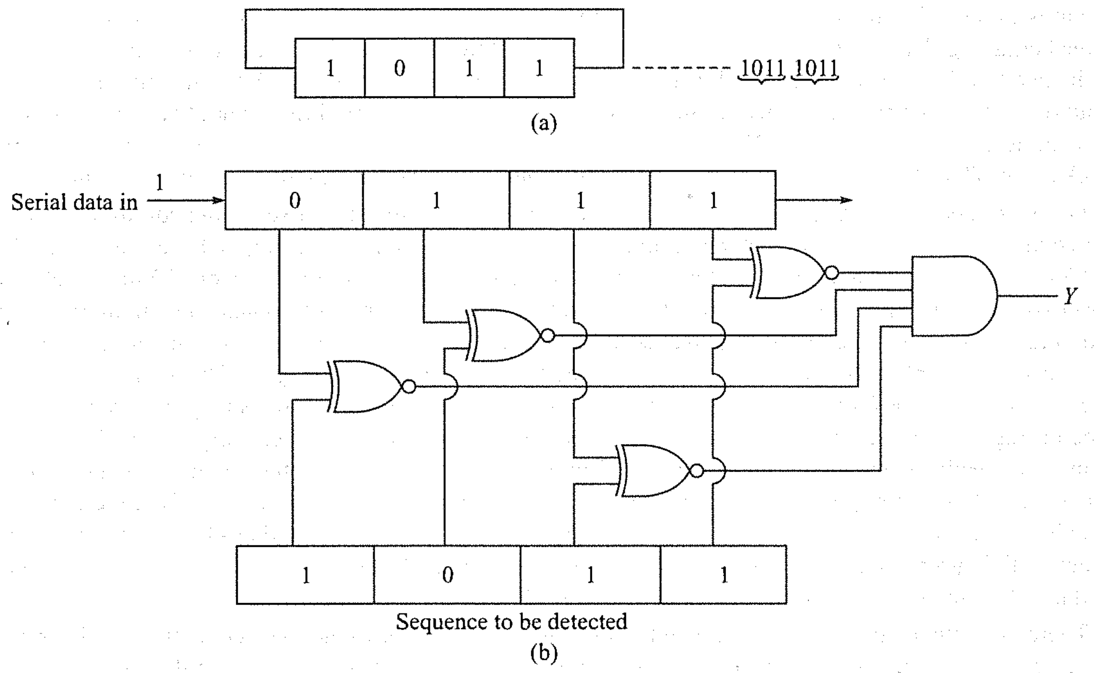


Fig. 9.23 **(a) 4-bit sequence generator, (b) 4-bit programmable sequence detector**

when data received so far is 0111 and word to be matched is 1011. The first two bits are mismatched and corresponding Ex-NOR outputs are low, so also final output Y. Now, as the next bit in the serial data stream is 1 when a clock trigger comes the first flip-flop of the shift-register stores 1 and 011 gets shifted to 2nd to 3rd flip-flops. With this both registers store 1011 and the first flip-flop of the shift-register stores 1 and 011 gets shifted to 2nd to 3rd flip-flops and $Y = 1$ completing sequence detection.

Note that Fig. 9.23b can be used as a *programmable sequence detector*, i.e. if we want to change the binary word to be detected we simply load that in the bottom register. For a fixed sequence detector, we can reduce hardware cost by removing bottom register and directly connect Ex-NOR input to $+V_{CC}$ or GND depending on whether we need a 1 or a 0 to be detected in a particular position.

Serial Adder

The addition operation and full adder (FA) circuit is discussed in detail in Chapter 6. We have seen for 8-bit addition we need 8 FA units (Fig. 6.6). There the addition is done in parallel. Using shift register we can convert this parallel addition to serial one and reduce number of FA units to only one. The benefit of this technique is more pronounced if the hardware unit that's needed to be used in parallel is very costly. Figure 9.24 shows how serial addition takes place in a time-multiplexed manner and also provides a snapshot of the register values at 3rd clock cycle.

Two 8-bit numbers, to be added ($A_7A_6...A_1A_0$ and $B_7B_6...B_1B_0$) are loaded in two 8-bit shift registers A and B. The LSB of each number appears in the rightmost position in two registers. Serial data out of A and B are fed to data inputs of full adder. The carry-in is fed from its own carry output delayed by one clock period

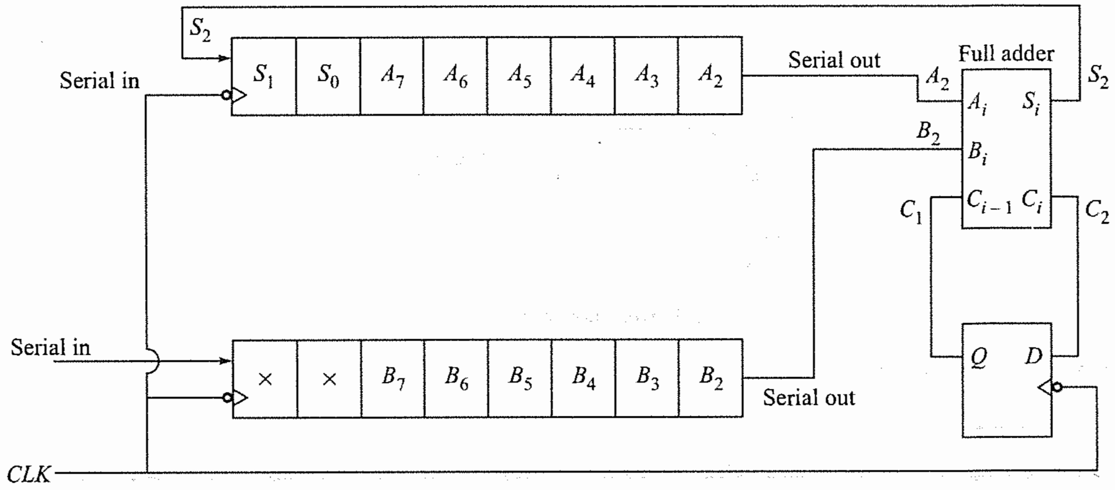


Fig. 9.24 **Serial addition of two 8-bit numbers (Register values shown are at 3rd clock cycle)**

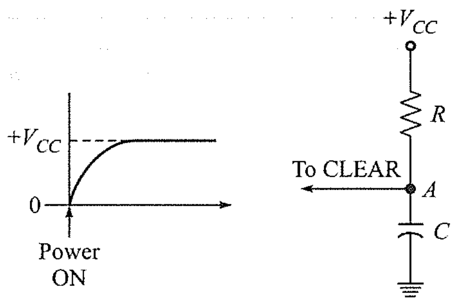
by a D flip-flop, which is initially cleared. Both registers and D flip-flop are triggered by same clock. The sum (S) output of FA is fed to serial data in of Shift Register A.

The serial addition takes place like this. The LSBs of two numbers (A_0 and B_0) appearing at serial out of respective registers are added by FA during 1st clock cycle and generate sum (S_0) and carry (C_0). S_0 is available at serial data input of register A and C_0 at input of D flip-flop. At NT of clock shift registers shift its content to right by one unit. S_0 becomes MSB of A and C_0 appears at D flip-flop output. Therefore in the second clock cycle FA is fed by second bit (A_1 and B_1) of two numbers and previous carry (C_0). In second clock cycle, S_1 and C_1 are generated and made available at serial data in of A register and input of D flip-flop respectively. At NT of clock S_1 becomes MSB of A and S_0 occupies next position. A_2 and B_2 now appear at FA data input and carry input is C_1. In 3rd clock cycle, S_2 and C_2 are generated and they get transferred similarly to register and flip-flop. This process goes on and is stopped by inhibiting the clock after 8 clock cycles. At that time shift register A stores the sum bits, S_7 in leftmost (MSB) position and S_0 in rightmost (LSB) position. The final carry is available at D flip-flop output.

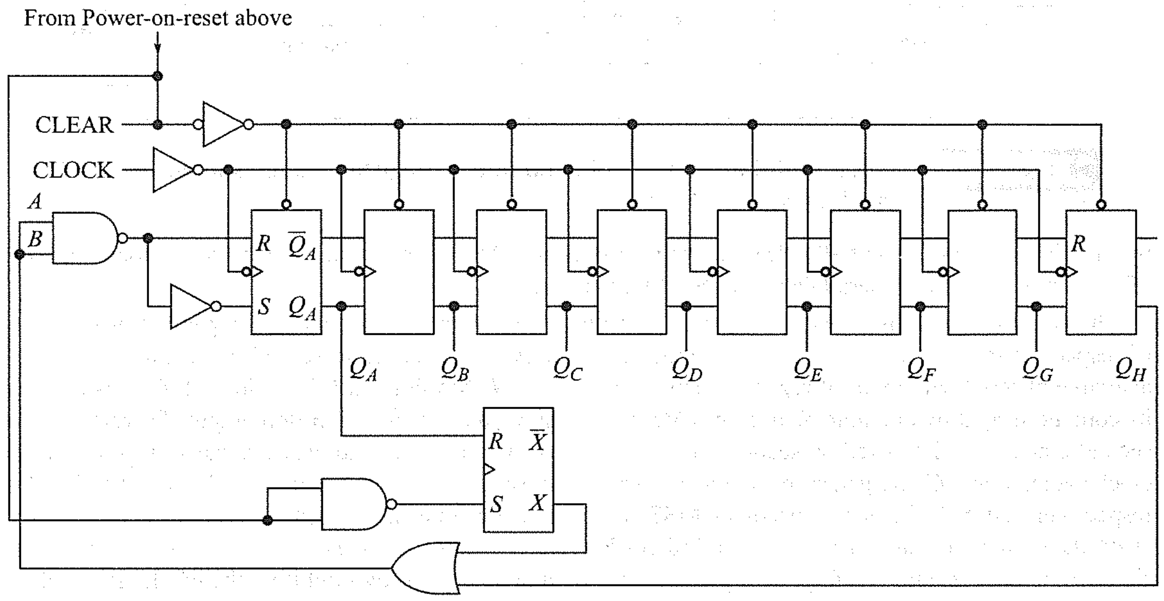
The limitation of this scheme is that the final addition result is delayed by eight clock cycles. In parallel adder the result is obtained almost instantaneously, after nanosecond order propagation delay of combinatorial circuit. However, using a high frequency clock the delay factor can be reduced considerably.

Example 9.10 The register in Fig. 9.20 can easily be cleared to all 0s by using the clear input. Show one method for setting a single 1 and the remaining 0s in the register.

Solution The simple power-on-reset circuit in Fig. 9.25a on the next page is widely used to generate the equivalent of a narrow negative pulse that occurs when power ($+V_{CC}$) is first applied to the system. Before the application of power, the voltage across the capacitor is zero. When $+V_{CC}$ is applied, the capacitor voltage charges toward $+V_{CC}$ with an RC time constant, and then remains at $+V_{CC}$ as long as the system power remains, as seen by the waveform in the figure. If point A is then connected to the clear input of the 54/74164, all flip-flops will automatically be reset to 0s when $+V_{CC}$ is first applied.



(a) Power-on-reset circuit



(b)

Fig. 9.25 54/74164 with logic to preset a single 1 and seven 0s

The logic added in the feedback path in Fig. 9.25b will now cause a single I to be set into the register. Here's how it works:

1 The power-on-reset pulse is inverted and used to initially set flip-flop *X*. This causes the output of the OR gate to be a 1, and the first clock PT will shift this 1 into Q_A.
2. When Q_A goes high, this will reset flip-flop *X*. At this point, the register contains a 1 in Q_A, and 0's in all other flip-flops. *X* will remain low as long as power is applied, and the data from Q_H will pass through the OR gate directly to the data input *AB*. The single 1 and the seven 0s will now shift around the register, advancing one position with each clock transition as desired.

Since the ring counter in Fig. 9.20 can function with more than a single 1 in it, it might be desirable to operate in this fashion at some time or other. It can, for example, be used to generate a more complex control waveform. Suppose, for instance, that the waveform shown in Fig. 9.26 were needed. This waveform could easily be generated by simply presetting the counter in Fig. 9.20 with a 1 in A, a 1 in C, and all the other flip-flops reset. Notice that it is really immaterial where the two 1s are set initially. It is necessary only to ensure that they are spaced one flip-flop apart.

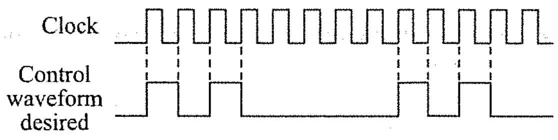


Fig. 9.26

Example 9.11 How would you preset the ring counter in Fig. 9.20 to obtain a square-wave output which is one-half the frequency of the clock? How about one-fourth the clock frequency?

Solution It is necessary only to preset a 1 in every other flip-flop, while the remaining flip-flops are all reset. This will generate a waveform at each flip flop output that is high for one clock period and then low for one clock period. The period of the output waveform is then two clock periods; therefore, the frequency is one-half the clock frequency. An output signal at one-fourth the clock frequency can be generated by presetting the shift register with two 1s, then two 0s, then two 1s, and then two 0s.

SELF-TEST

9. What is a ring counter?
10. What is a power-on-reset circuit used for?
11. What is a switched tail counter?
12. How does a serial adder work?

9.8 REGISTER IMPLEMENTATION IN HDL

In this section, we see how to describe a register using HDL. The parallel in parallel out register, primarily used for storage purpose is described for IC 741174 (Fig. 9.15) in Verilog code in the first column. We use vector notation for convenience. When Clear is activated (active LOW) all 6 outputs of Q are reset.

In second column, we show code for shift right register shown in Fig. 9.5 where T is the final output and Q, R, S are internal outputs. Since they are outputs of always block they have to be defined as **reg** and not **wire**. Note that, we use a new assignment operator <= within **always** block which unlike = operator executes all associated statements concurrently. If we had used = instead of <=, the D input through sequential execution would have reached final output in one clock cycle (unlike 4 clock cycles required in 4-bit shift register), also all the flip-flops within the register will have same value that of serial data input. Often, use of = operator is called blocking mode operation and use of <= is called non blocking mode. In column 3, we show a 4-bit serial in parallel out right shift register where all the flip-flop outputs are available externally. We use vector notation for convenience wherever possible.

```
module Reg74174(D,Clock,     module SR1(D,Clock,T);      module SR2(D,Clock,Q);
   Clear,Q);                 input Clock,D;  // Use      input Clock,D; //Clear as
input Clock, Clear;          output T; // Clear as in    output [3:0] Q; //in 74174
input [5:0] D;               reg  T; // LHS to           reg  [3:0] Q;
output [5:0] Q;                 initialize                   //to initialize
reg [5:0] Q;                 reg Q,R,S; //internal
```

```
always @ (negedge Clock
   or negedge Clear)
if (~Clear) Q=6'b0;
 //Q stores 6 binary 0
else Q=D;
endmodule
```

```
always @ (negedge Clock)

  begin
    Q <= D;
    R <= Q;
    S <= R;
    T <= S;
  end
endmodule
```

```
always @ (negedge
        Clock)
  begin
    Q[0] <= D;
    Q[1] <= Q[0];
    Q[2] <= Q[1];
    Q[3] <= Q[2];
  end
endmodule
```

Example 9.12 Write Verilog code for switched tail counter shown in Fig. 9.24.

Solution The code is similar to Shift Register description given above in second column. The serial data input here is taken from inverse of final flip-flop output. Output is generated from decoding logic $Y = Q'T'$.

```
module STC(Clock,Clear,Y); //Switched Tail Counter
input Clock, Clear;
output  Y;
reg Q,R,S,T; //internal outputs of flip-flops
assign Y= (~Q)&(~T);
always @ (negedge Clock)
begin
  if (~Clear) Q=6'b0; //Q stores 6 binary 0
  else
    begin
    Q <= ~T;  //Tail is switched and connected to input
    R <= Q;
    S <= R;
    T <= S;
  end
endmodule
```

PROBLEM SOLVING WITH MULTIPLE METHODS

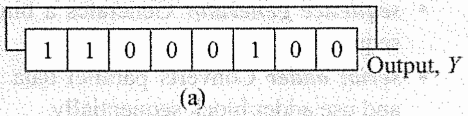
Problem Design an 8-bit sequence generator that generates the sequence 11000100 repetitively using shift register.

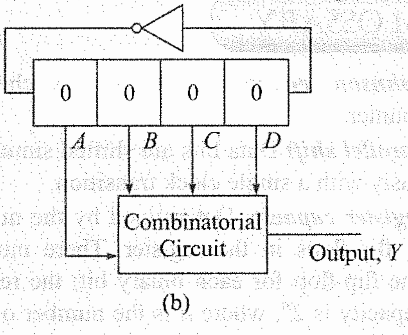
Solution We use ring counter and switched-tail counter derived from shift registers for this purpose.

In Method-1, we load an 8-bit ring counter as shown in Fig. 9.27a with the given sequence and at the output, the sequence will be repetitively generated.

In Method-2, we consider a modulo-8 switched-tail counter developed from 4-bit shift register. Let it be initially loaded with 0000. Then the 8 repetitive states of the counter will be as shown in Fig.

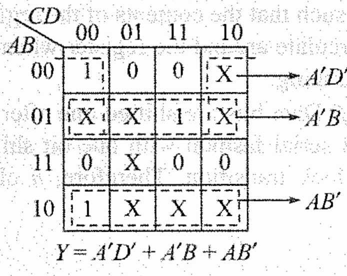
9.27b and is reproduced in Fig. 9.27c. We then design a combinatorial circuit which for each of the state generates one bit of the sequence. The Karnaugh Map for this is shown in Fig. 9.27d. Note that the unused states can be considered as 'don't care'. The logic equation of the combinatorial circuit realized as Fig. 9.27b can be written as $Y = A'D' + A'B + AB'$





Counter A	B	C	D	Sequence Generator Output, Y
0	0	0	0	1
1	0	0	0	1
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
0	1	1	1	1
0	0	1	1	0
0	0	0	1	0
0	0	0	0	...repeats...

(c)



AB \ CD	00	01	11	10
00	1	0	0	X
01	X	X	1	X
11	0	X	0	0
10	1	X	X	X

$Y = A'D' + A'B + AB'$

(d)

Fig. 9.27 **(a) Solution with Method-1, (b)-(d) Solution with Method-2, (b) Targeted realization, (c) Counter sequence vs. sequence generator output, (d) Karnaugh Map to generate logic equation**

Note that Method-2 can be used with any other types of counter and is not restricted to shift register based counter. This is shown with Example 10.15 in next Chapter.

SUMMARY

Shift registers are important digital building blocks that can be used to store binary data. They can accept data bits in either a serial or a parallel format and can, likewise, deliver data in either serial or parallel. There are thus four basic register types: serial input–serial output, serial input–parallel output, parallel input–serial output, and parallel input–parallel output.

In one application, a register can be used to change data from a serial format into a parallel format, or vice versa. As such, shift registers can be regarded as *data format changers*. The UART is a good example of a data changer. There are a great many other shift register applications—arithmetic operations, logic operations, and counters, to name only a few. Our intent has not been to discuss all the possible applications of shift registers, but rather to consider in detail how each type of register functions. With this knowledge, one can then discover the many and varied practical applications in existing digital designs.

GLOSSARY

- ***Johnson counter*** Refer to switched-tail counter.
- ***parallel shift*** Data bits are shifted simultaneously with a single clock transition.
- ***register capacity*** Determined by the number of flip-flops in the register. There must be one flip-flop for each binary bit; the register capacity is 2^n, where n is the number of flip-flops.
- ***ring counter*** A basic shift register with direct feedback such that the contents of the register simply circulate around the register when the clock is running.
- ***serial shift*** Data bits are shifted one after the other in a serial fashion with one bit shifted at each clock transition. Therefore, n clock transitions are needed to shift an n-bit binary number.
- ***sequence detector*** Detects a binary word from input data stream.
- ***sequence generator*** Generates a binary data sequence.
- ***serial adder*** Converts parallel data to serial and use adder block sequentially.
- ***switched tail counter*** Shift register with inverting output of last flip-flop fed to first flip-flop input. For n-bit shift register can give modulo $2N$ counter.
- ***shift register*** A group of flip-flops connected in such a way that a binary number can be shifted into or out of the flip-flops.
- ***UART*** Universal asynchronous receiver-transmitter.

PROBLEMS

Section 9.1

9.1 Determine the number of flip-flops needed to construct a shift register capable of storing:

a. A 6-bit binary number
b. Decimal numbers up to 32
c. Hexadecimal numbers up to F

9.2 A shift register has eight flip-flops. What is the largest binary number that can be stored in it? Decimal number? Hexadecimal number?

9.3 Name the four basic types of shift registers, and draw a block diagram for each.

Section 9.2

9.4 Draw the waveforms to shift the binary number 1010 into the register in Fig. 9.2.

9.5 Draw the waveforms to shift the binary number 1001 into the register in Fig. 9.3.

9.6 The register in Fig. 9.2 has 0100 stored in it. Draw the waveforms for four clock transitions, assuming that both J and K are low.

9.7 Draw the waveforms showing how the decimal number 68 is shifted into the 54/74LS91 in Fig. 9.5. Show eight clock periods.

9.8 The hexadecimal number AB is stored in the 54/74LS91 in Fig. 9.5. Show the waveforms at the output, assuming that the clock is allowed to run for eight cycles and that $A = B = 0$.

Section 9.3

9.9 How long will it take to shift an 8-bit binary number into the 54/74164 in Fig. 9.7 if the clock is:

a. 1 MHz b. 5 MHz

9.10 For the 54/74164 in Fig. 9.7, B is high, clear is high, a 1-MHz clock is used to shift the decimal number 200 into the register at A. Draw all the waveforms (such as in Fig. 9.9).

9.11 On the basis of information in Example 9.5, what is the maximum frequency of the clock if the minimum data transition time is 30 ns?

9.12 In Fig. 9.9, if control is taken low at time K, will the data stored in the register remain even if the clock is allowed to run? Explain.

Section 9.4

9.13 For the circuit in Fig. 9.11c, write the logic levels on each gate leg, given:

a. Control $= 1, X_1 = 0, X_2 = 1$
b. Control $= 0, X_1 = 0, X_2 = 1$

9.14 Redraw the 54/74166 in Fig. 9.10 showing only those gates used to shift data into the register in parallel. If a gate is disabled, don't draw it.

9.15 Redraw the 54/74166 in Fig. 9.10 showing only those gates used to shift data into the register in serial. If a gate is disabled, don't draw it.

9.16 Explain the operation of the 54/74166 for each of the six truth table entries in Fig. 9.12.

9.17 Draw all the input and output waveforms for the 54/74166 in Fig. 9.10, assuming that the decimal number 190 is shifted into the register in:

a. Parallel b. Serial

Section 9.5

9.18 Redraw the 54/7495A shift register in Fig. 9.15 showing only those gates used to shift data into the register in parallel. If a gate is disabled, don't draw it.

9.19 Repeat Prob. 9.18, assuming that the data is shifted in serially.

9.20 Draw the waveforms necessary to enter, and shift to the right a single 1 through the shift register in Fig. 9.15.

9.21 Repeat Prob. 9.20, but do a left shift. (See Fig. 9.16.)

Section 9.6

9.22 Draw the waveforms that would result if the circulating register (ring counter) in Fig. 9.20 had alternate 1s and 0s stored in it and a 1-MHz clock were applied.

9.23 The register in Fig. 9.20 can easily be cleared to all 0s by using the clear input. See if you can design logic circuitry to set the register with alternating 1s and 0s.

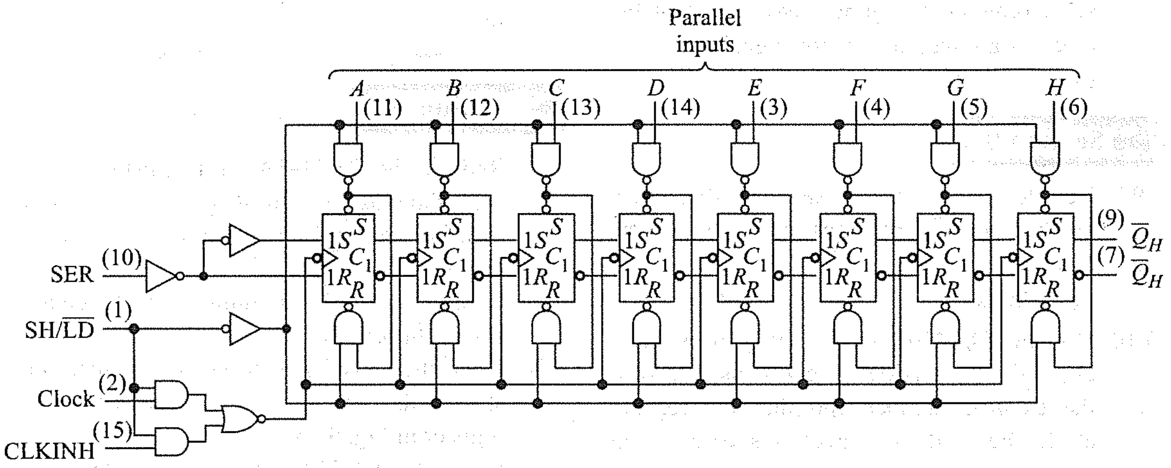
9.24 Explain the operation of the 54/74165 shift register. Redraw one of the eight flip flops along with its two NAND gates, and analyze:

a. Parallel data entry
b. Shift right
c. Serial data entry

The logic diagram is given in Fig. 9.28 on the next page.

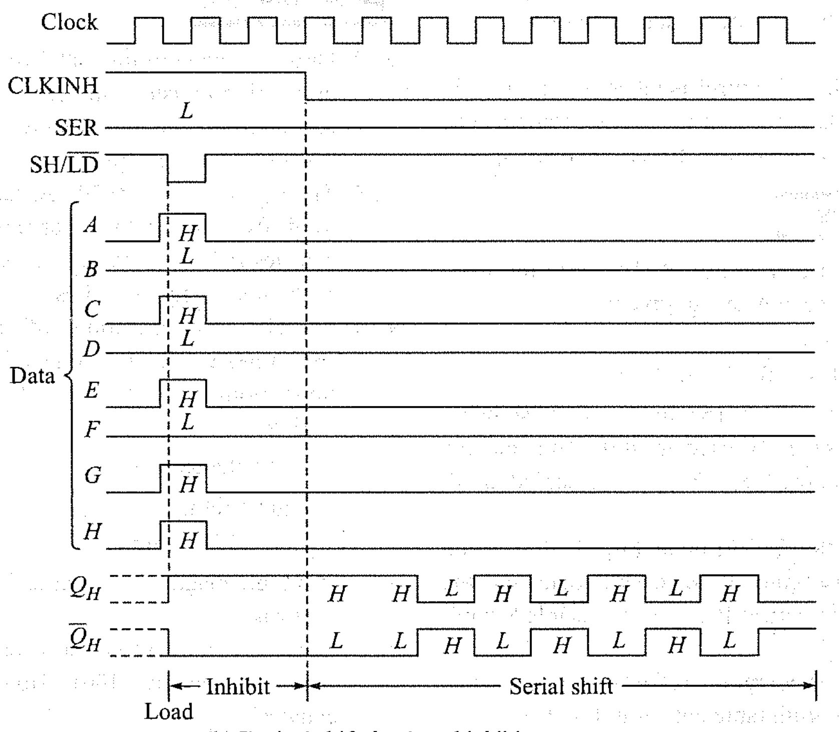
9.25 Show how modulo-8 switched tail counter works if is initialized with '1001'. How to decode this counter?

9.26 Show the circuit diagram for an 8-bit sequence detector which has to detect a fixed pattern '10011110' from incoming binary data stream.



Pin numbers shown are for *J* and *N* packages.

(a) Logic diagram (positive logic)



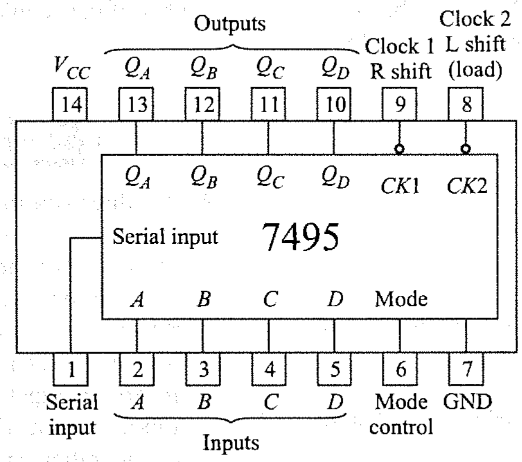
(b) Typical shift, load, and inhibit sequences

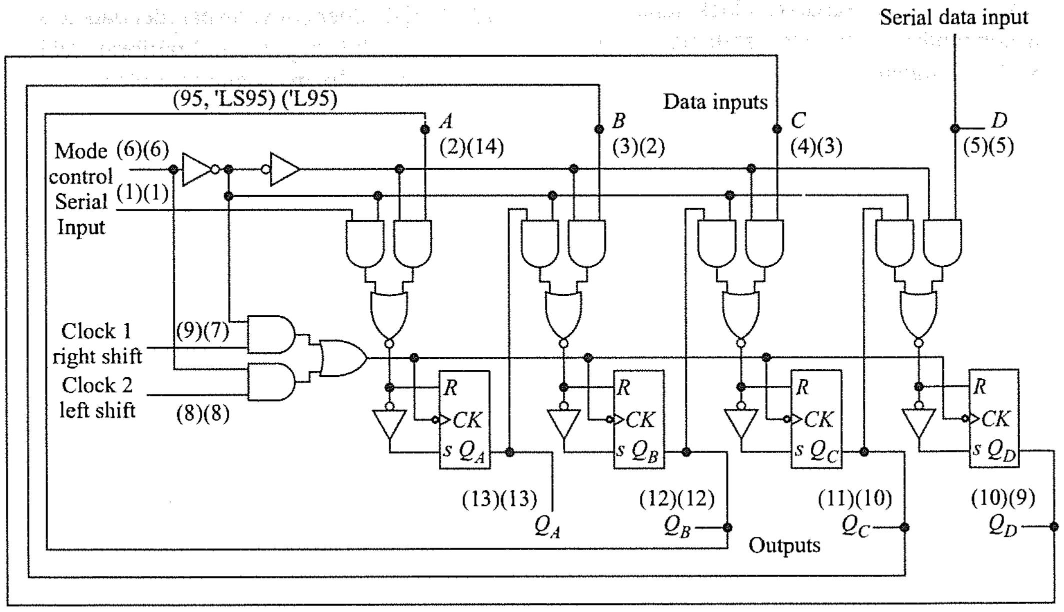
Fig. 9.28 54/74165, 8-bit shift register

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to study Shift Register and use it to get Ring Counter and Johnson Counter.

Theory: The shift register is a special kind of register, i.e. group of memory units where binary data can be shifted from one unit to another in a sequential manner. The loading of shift register may be done serially or parallely. In serial loading as many number of clock cycles are required as the size of the register to load it fully. In parallel loading, all the memory units are loaded simultaneously in one clock cycle. The data within the register can be





made to shift in left, right or both directions. Feedback from uncomplemented final output to the serial input can make the shift register work as Ring Counter while feedback from complemented output allows it to work as Johnson Counter.

Apparatus: 5 V DC Power supply, Multimeter, Bread Board, Clock Generator, and Oscilloscope

Work element: IC 7495 is a 4-bit shift register with a mode control that allows parallel loading or right shift. Verify the truth table of this. The register can be made to work like a left shift register using parallel loading as shown in the figure. Verify the left shift operation. Use feedback to make it work like (i) Ring Counter and then (ii) Johnson Counter. Initialize appropriately and verify the counter output w.r.t. clock in a dual trace oscilloscope. Find how counting sequence varies on different initializations and comment on the modulo number of the counter.

Answers to Self-tests

1. 255
2. PTs
3. 30 ns
4. Eight; one
5. The shift/load line on the 74166 allows either serial or parallel data entry.
6. PTs
7. Two 7495As connected in series will store 8 bit numbers.
8. The 7495A has separate clock inputs to accommodate separate shift-right and shift-left signals.
9. A direct-feedback shift register—the contents of the register circulate around the register when the clock is running.
10. A power-on-reset circuit is used to preset flip-flops to any desired states.
11. Switched tail counter is a shift register with inverting output of last flip-flop fed to first flip-flop input. For n-bit shift register this configuration can give modulo $2N$ counter.
12. Serial adder converts parallel data to serial using shift register and performs addition sequentially using an adder block.

Counters

10

OBJECTIVES

- Describe the basic construction and operation of an asynchronous counter
- Determine the logic circuit needed to decode a given state from the output of a given counter
- Describe the synchronous counter and its advantages
- See how the modulus of a counter can be reduced by skipping one or more of its natural counts
- Understand how to design counter as a finite state machine

A counter is probably one of the most useful and versatile subsystems in a digital system. A counter driven by a clock can be used to count the number of clock cycles. Since the clock pulses occur at known intervals, the counter can be used as an instrument for measuring time and therefore period or frequency. There are basically two different types of counters—synchronous and asynchronous.

The ripple counter is simple and straightforward in operation and its construction usually requires a minimum of hardware. It does, however, have a speed limitation. Each flip-flop is triggered by the previous flip-flop, and thus the counter has a cumulative settling time. Counters such as these are called serial, or asynchronous.

An increase in speed of operation can be achieved by use of a parallel or synchronous counter. Here, every flip-flop is triggered by the clock (in synchronism), and thus settling time is simply equal to the delay time of a single flip-flop. The increase in speed is usually obtained at the price of increased hardware.

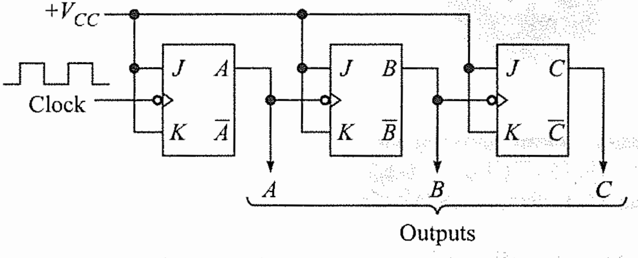
Serial and parallel counters are used in combination to compromise between speed of operation and hardware count. Serial, parallel, or combination counters can be designed such that each clock transition advances the contents of the counter by one; it is then operating in a count-up mode. The opposite is also possible; the counter then operates in the count-down mode. Furthermore, many counters can be either "cleared" so that every flip-flop contains a zero, or preset such that the contents of the flip-flops represent any desired binary number.

Now, let's take a look at some of the techniques used to construct counters.

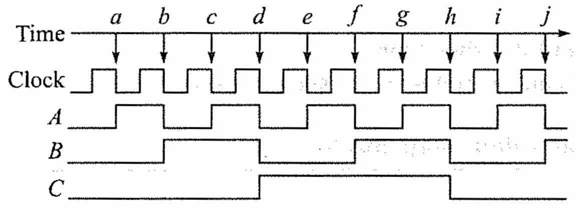
10.1 ASYNCHRONOUS COUNTERS

Ripple Counters

A binary ripple counter can be constructed using clocked *JK* flip-flops. Figure 10.1 shows three negative-edge-triggered, *JK* flip-flops connected in cascade. The system clock, a square wave, drives flip-flop *A*. The output of *A* drives *B*, and the output of *B* drives flip-flop *C*. All the *J* and *K* inputs are tied to $+V_{CC}$. This means that each flip-flop will change state (toggle) with a negative transition at its clock input.



(a) Three-bit binary ripple counter



(b) Waveforms

Negative clock transitions	C	B	A	State or count
---	0	0	0	0
a	0	0	1	1
b	0	1	0	2
c	0	1	1	3
d	1	0	0	4
e	1	0	1	5
f	1	1	0	6
g	1	1	1	7
h	0	0	0	0

(c) Truth table

Fig. 10.1

When the output of a flip-flop is used as the clock input for the next flip-flop, we call the counter a *ripple counter*, or *asynchronous counter*. The *A* flip-flop must change state before it can trigger the *B* flip-flop, and the *B* flip-flop has to change state before it can trigger the *C* flip-flop. The triggers move through the flip-flops like a ripple in water. Because of this, the overall propagation delay time is the sum of the individual delays. For instance, if each flip-flop in this three-flip-flop counter has a propagation delay time of 10 ns, the overall propagation delay time for the counter is 30 ns.

The waveforms given in Fig. 10.1b show the action of the counter as the clock runs. Let's assume that the flip-flops are all initially reset to produce 0 outputs. If we consider *A* to be the least-significant bit (LSB) and *C* the most-significant bit (MSB), we can say the contents of the counter is *CBA* = 000.

Every time there is a clock NT, flip-flop *A* will change state. This is indicated by the small arrows (↓) on the time line. Thus at point *a* on the time line, *A* goes high, at point *b* it goes back low, at *c* it goes back high, and so on. Notice that the waveform at the output of flip-flop *A* is one-half the clock frequency.

Since A acts as the clock for B, each time the waveform at A goes low, flip-flop B will toggle. Thus at point b on the time line, B goes high; it then goes low at point d and toggles back high again at point f. Notice that the waveform at the output of flip-flop B is one-half the frequency of A and one-fourth the clock frequency.

Since B acts as the clock for C, each time the waveform at B goes low, flip-flop C will toggle. Thus C goes high at point d on the time line and goes back low again at point h. The frequency of the waveform at C is one-half that at B, but it is only one-eighth the clock frequency.

Example 10.1 What is the clock frequency in Fig. 10.1 if the period of the waveform at C is 24 μs?

Solution Since there are eight clock cycles in one cycle of C, the period of the clock must be 24/8 = 3 μs. The clock frequency must then be $1/(3 \times 10^{-6}) = 333$ kHz.

Notice that the output condition of the flip-flops is a binary number equivalent to the number of clock NTs that have occurred. Prior to point a on the time line the output condition is $CBA = 000$. At point a on the time line the output condition changes to $CBA = 001$, at point b it changes to $CBA = 010$, and so on. In fact, a careful examination of the waveforms will reveal that the counter content advances one count with each clock NT in a "straight binary progression" that is summarized in the truth table in Fig. 10.1c.

Because each output condition shown in the truth table is the binary equivalent of the number of clock NTs, the three cascaded flip-flops in Fig. 10.1 comprise a 3-bit binary ripple counter. This counter can be used to count the number of clock transitions up to a maximum of seven. The counter begins at count 000 and advances one count for each clock transition until it reaches count 111. At this point it resets back to 000 and begins the count cycle all over again. We can say that this ripple counter is operating in a count-up mode.

Since a binary ripple counter counts in a straight binary sequence, it is easy to see that a counter having n flip-flops will have 2^n output conditions. For instance, the three-flip-flop counter just discussed has $2^3 = 8$ output conditions (000 through 111). Five flip-flops would have $2^5 = 32$ output conditions (00000 through 11111), and so on. The largest binary number that can be represented by n cascaded flip-flops has a decimal equivalent of $2^n - 1$. For example, the three-flip-flop counter reaches a maximum decimal number of $2^3 - 1$. The maximum decimal number for five flip-flops is $2^5 - 1 = 31$, while six flip-flops have a maximum count of 63.

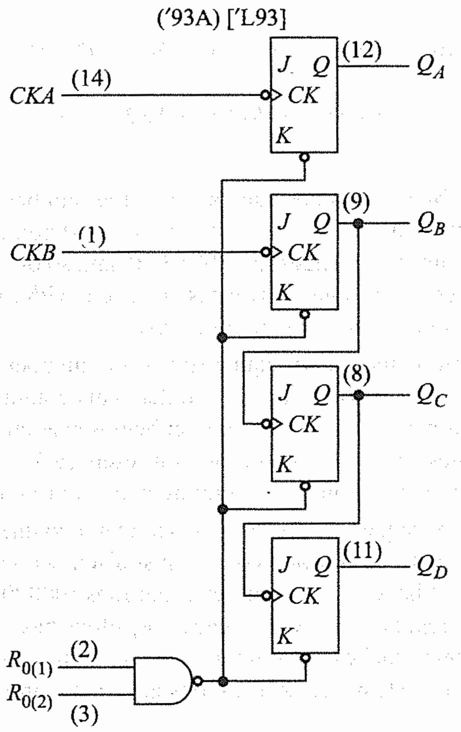
A three-flip-flop counter is often referred to as a modulus-8 (or mod-8) counter since it has eight states. Similarly, a four-flip-flop counter is a mod-16 counter, and a six-flip-flop counter is a mod-64 counter. The *modulus* of a counter is the total number of states through which the counter can progress.

Example 10.2 How many flip-flops are required to construct a mod-128 counter? A mod-32? What is the largest decimal number that can be stored in a mod-64 counter?

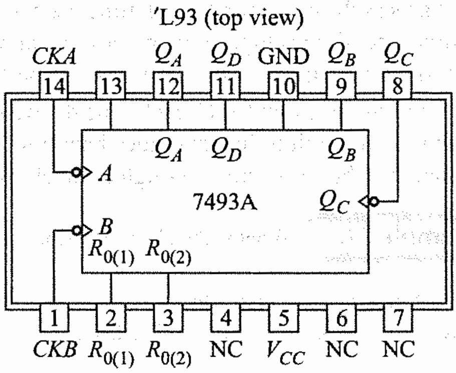
Solution A mod-128 counter must have seven flip-flops, since $2^7 = 128$. Five flip-flops are needed to construct a mod-32 counter. The largest decimal number that can be stored in a six-flip-flop counter (mod-64) is 111111 = 63. Note carefully the difference between the modulus (total number of states) and the maximum decimal number.

The 54/7493A

The logic diagram, DIP pinout, and truth table for a 54/7493A are given in Fig. 10.2. This TTL MSI circuit is a 4-bit binary counter that can be used in either a mod-8 or a mod-16 configuration. If the clock is applied at input CKB, the outputs will appear at Q_B, Q_C, and Q_D, and this is a mod-8 binary ripple counter exactly like that in Fig. 10.1. In this case, flip-flop Q_A is simply unused.



(a) Logic diagram



Positive logic: see function tables
NC – No internal connection

(b) DIP pinout

'L93A, 'L93, 'LS93 Count sequence

Count	Output			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H
10	H	L	H	L
11	H	L	H	H
12	H	H	L	L
13	H	H	L	H
14	H	H	H	L
15	H	H	H	H

(c) Truth table

Fig. 10.2 7493A

On the other hand, if the clock is applied at input *CKA* and flip-flop Q_A is connected to input *CKB*, we have a mod-16, 4-bit binary ripple counter. The outputs are Q_A, Q_B, Q_C, and Q_D. The proper truth table for this connection is given in Fig. 10.2c.

All the flip-flops in the 7493A have direct reset inputs that are active low. Thus a high level at both reset inputs of the NAND gate, $R_{0(1)}$ and $R_{0(2)}$, is needed to reset all flip-flops simultaneously. Notice that this reset operation will occur without regard to the clock.

Example 10.3 Draw the correct output waveforms for a 7493A connected as a mod-16 counter.

Solution The correct waveforms are shown in Fig. 10.3. The contents of the counter is 0000 at point *a* on the time line. With each negative clock transition, the counter is advanced by one until the counter contents are 1111 at point *b* on the time line. At point *c*, the counter resets to 0000, and the counting sequence repeats. Clearly, this is a mod-16 counter, since there are 16 states (0000 through 1111), and the maximum decimal number that can be stored in the flip-flops is decimal 15 (1111).

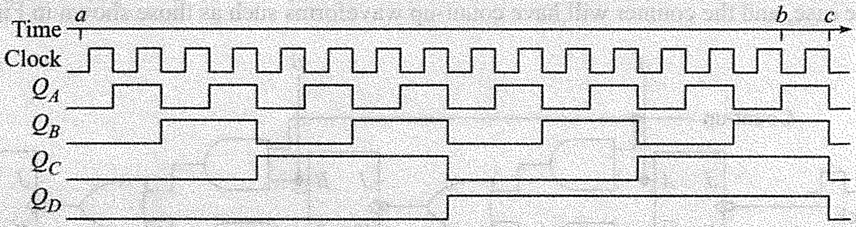
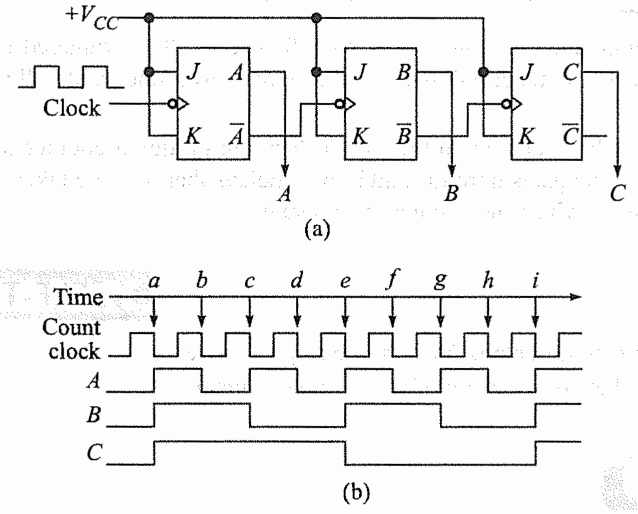


Fig. 10.3 **Waveforms for a mod-16, 7493A**

An interesting and useful variation of the 3-bit ripple counter in Fig. 10.1 is shown in Fig. 10.4. The system clock is still used at the clock input to flip-flop A, but the complement of A, $\overline{A}$, is used to drive flip-flop B, likewise; $\overline{B}$ is used to drive flip-flop C. Take a look at the resulting waveforms.



Count	C	B	A
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1
0	0	0	0
7	1	1	1

(c)

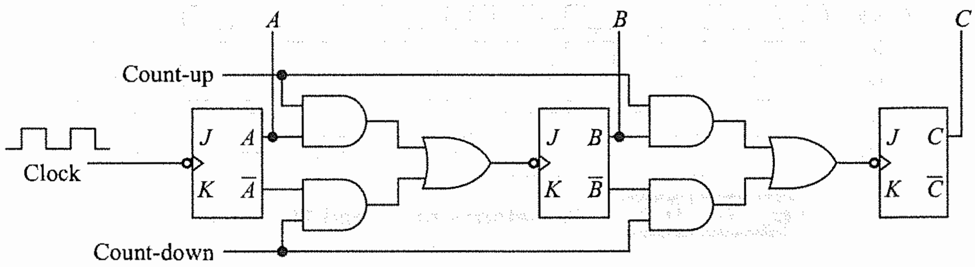
Fig. 10.4 **A down counter**

Flip-flop A simply toggles with each negative clock transition as before. But flip-flop B will toggle each time *A goes high*! Notice that each time A goes high, $\overline{A}$ goes low, and it is this negative transition on $\overline{A}$ that triggers B. On the time line, B toggles at points *a*, *c*, *e*, *g* and *i*.

Similarly, flip-flop C is triggered by $\overline{B}$ and so C will toggle each time B goes high. Thus C toggles high at point *a* on the time line, toggles back low at point *e* and goes back high again at point *i*.

The counter contents become $ABC = 111$ at point *a* on the time line, change to 110 at point *b*, and change to 101 at point *c*. Notice that the counter contents are reduced by one count with each clock transition! In other words, the counter is operating in a count-down mode. The results are summarized in the truth table in Fig. 10.4c. This is still a mod-8 counter, since it has eight discrete states, but it is connected as a *down counter*.

A 3-bit asynchronous *up-down counter* that counts in a straight binary sequence is shown in Fig. 10.5. It is simply a combination of the two counters discussed previously. For this counter to progress through a count-up sequence, it is necessary to trigger each flip-flop with the true side of the previous flip-flop (as opposed to the complement side). If the count-down control line is low and the count-up control line high, this will be the case, and the counter will have count-up waveforms such as those shown in Fig. 10.1.



Note: The J and K inputs are all tied to $+V_{CC}$.
The counter outputs are A, B, and C.

Fig. 10.5 **3-bit binary up-down counter**

On the other hand, if count-down is high and count-up is low, each flip-flop will be triggered from the complement side of the previous flip-flop. The counter will then be in a count-down mode and will progress through the waveforms as shown in Fig. 10.4.

This process can be continued to other flip-flops down the line to form an up-down counter of larger moduli. It should be noticed, however, that the gates introduce additional delays that must be taken into account when determining the maximum rate at which the counter can operate.

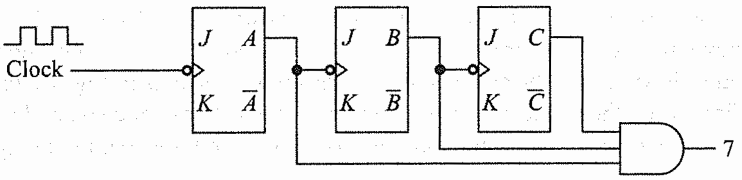
SELF-TEST

1. What is the largest binary number representable by a mod-6 ripple counter?
2. How many flip-flops are required to construct a mod-1024 ripple counter?

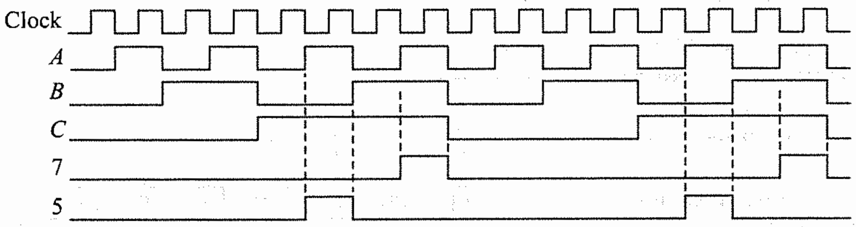
10.2 DECODING GATES

A decoding gate can be connected to the outputs of a counter in such a way that the output of the gate will be high (or low) only when the counter contents are equal to a given state. For instance, the decoding gate connected to the 3-bit ripple counter in Fig. 10.6a will decode state 7 ($CBA = 111$). Thus the gate output will be high only when $A = 1$, $B = 1$, and $C = 1$ and the waveform appearing at the output of the gate is labeled 7. The Boolean expression for this gate can be written $7 = CBA$. A comparison with the truth table for this counter (in Fig. 10.1) will reveal that the condition $CBA = 111$ is true only for state 7.

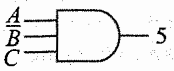
The other seven states of the counter can be decoded in a similar fashion. It is only necessary to examine the truth table for the counter and then the proper Boolean expression for each gate can be written. For instance, to decode state 5, the truth table reveals that $CBA = 101$ is the unique state. For the gate output to be high during this time, we must use C, $\overline{B}$, and A at the gate inputs. Notice carefully that if $B = 0$, then $\overline{B}$ = 1! The correct Boolean expression is then $5 = C\overline{B}A$, and the desired gate is that given in Fig. 10.6c. The waveform is again that given in Fig. 10.6b and is labeled 5.



(a) Decoding gate for state 7



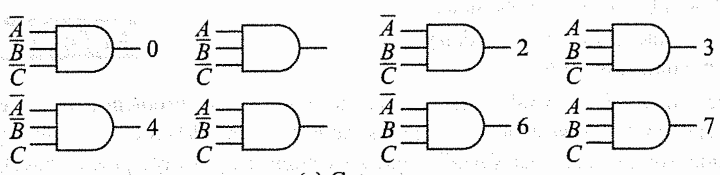
(b) Waveforms



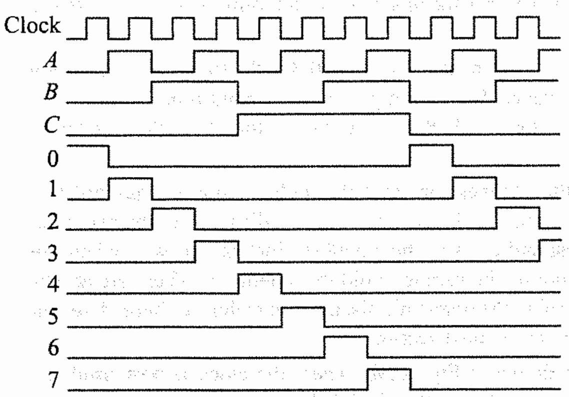
(c) Gate to decode state 5

Fig. 10.6

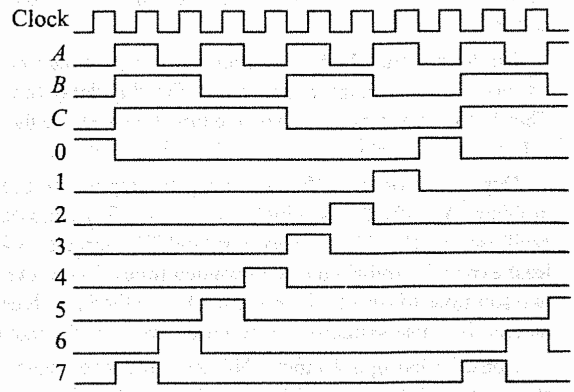
All eight gates necessary to decode the eight states of the 3-bit counter in Fig. 10.1 are shown in Fig. 10.7a. The gate outputs are shown in Fig. 10.7b. These decoded waveforms are a series of positive pulses that occur in a strict time sequence and are very useful as control signals throughout a digital system. If we consider state 0 as the first event, then state 1 will be the second, state 2 the third, and so on, up to state 7. Clearly the counter is counting upward in decimal notation from 0 to 7 and then beginning over again at 0.



(a) Gates



(b) Count-up mode



(c) Count-down mode

Fig. 10.7 **Decoding gates for a 3-bit binary ripple counter**

If these eight gates are connected to the up-down counter shown in Fig. 10.5, the decoded waveforms will appear exactly as shown in Fig. 10.7b, provided the counter is operating in the count-up mode. If the counter is operated in the count-down mode, the decoded waveforms will appear as in Fig. 10.7c. In this case, if state 0 is considered the first event, then state 7 is the second event, then state 6, and so on, down to state 1. Clearly the counter is counting downward in decimal notation from 7 to 0 and then beginning again at 7.

Example 10.4 Show how to use a 54LS11, triple 3-input AND gate to decode states 1, 4, and 6 of the counter in Fig. 10.5.

Solution The logic diagram and pinout for a 54LS11 is given in Fig. 10.8. The correct Boolean expressions for the desired states are $1 = \overline{C}\,\overline{B}A$, $4 = C\overline{B}\,\overline{A}$, and $6 = CB\overline{A}$. Wiring from the counter flip-flop outputs to the chip is given in Fig. 10.8.

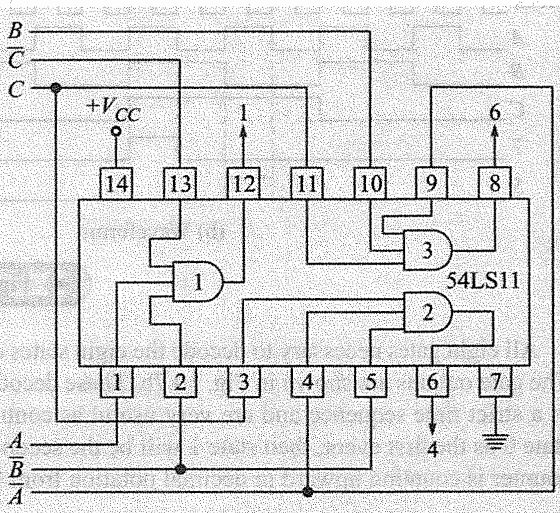


Fig. 10.8

Let's take a more careful look at the waveforms generated by the counter in Fig. 10.5 as it operates in the count-up mode. The clock and each flip-flop output are redrawn in Fig. 10.9, and the propagation delay time of each flip-flop is taken into account. Notice carefully that the clock is the trigger for flip-flop A, and the A waveform is thus delayed by t_p from the negative clock transition. For reference purposes, the complement of A, $\overline{A}$, is also shown. Naturally it is the exact mirror image of A.

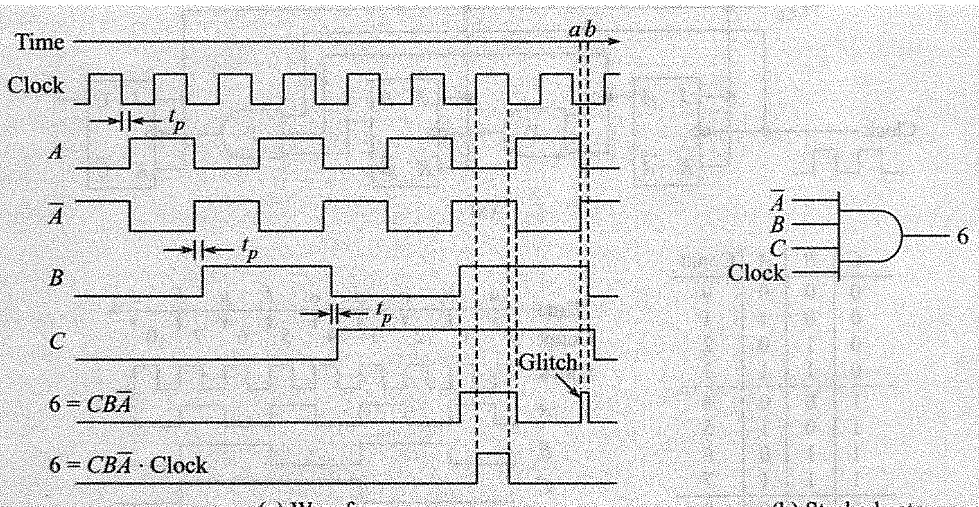
Since A acts as the trigger for B, the B waveform is delayed by one flip-flop delay time from the negative transition of A. Similarly, the C waveform is delayed by t_p from each negative transition of B.

At first glance, these delay times would seem to offer no more serious problem than a speed limitation for the counter, but a closer examination reveals a much more serious problem. When the decoding gates in Fig. 10.7 are connected to this counter (or, indeed, when decoding gates are connected to any ripple counter), glitches may appear at the outputs of one or more of the gates. Consider, for instance, the gate used to decode state 6. The proper Boolean expression is $6 = CB\overline{A}$. So, in Fig. 10.9 the correct output waveform for this gate is high only when $C = 1$, $B = 1$, and $\overline{A} = 1$.

But look at the glitch that occurs when the counter progresses from state 7 to state 0. On the time line, A goes low ($\overline{A}$ goes high) at point a. Because of flip-flop delay time, however, B does not go low until point b on the time line! Thus between points a and b on the time line we have the condition $C = 1$, $B = 1$, and $A = 1$—therefore, the gate output is high, and we have a glitch! Look at the waveform $6 = CB\overline{A}$.

Depending on how the decoder gate outputs are used, the glitches (or unwanted pulses) may or may not be a problem. Admittedly the glitches are only a few nanoseconds wide and may even be very difficult to observe on an oscilloscope. But TTL is *very fast*, and TTL circuits will respond to even the smallest glitches—usually when you least expect it, and always at unwanted times! Therefore, you must beware to avoid this condition. There are at least two solutions to the glitch problem. One method involves strobing the gates; we discuss that technique here. A second method is to use synchronous counters; we consider that topic in the next section.

Consider using a 4-input AND gate to decode state 6 as shown in Fig. 10.9b, where the clock is now used as a strobe. An examination of the waveforms in this figure clearly reveals that the clock is low between points a and b on the time line. Since the clock must be high for the gate output to be high, the glitch cannot possibly occur! On the other



(a) Waveforms (b) Strobed gate

Fig. 10.9

hand, the clock is high when $C = 1$, $B = 1$, and $\overline{A} = 1$, and the waveform for 6 appears exactly as it should. Notice that the width of the positive pulse at 6 is exactly the width of the positive portion of the clock. Look at the waveform $6 = CB\overline{A}$ clock. This technique can be applied to the other seven decoding gates for this counter (or for any other counter), and the decoded output waveforms will be glitch-free.

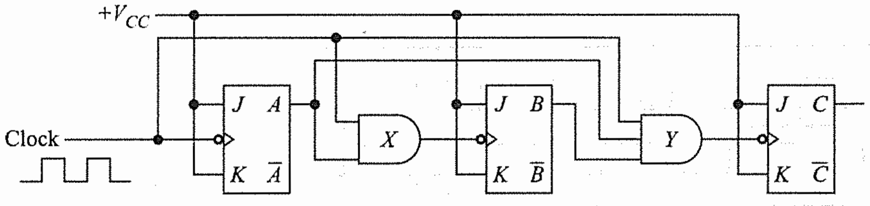
SELF-TEST

3. What is the Boolean expression for an AND gate needed to decode state 3 of the counter in Fig. 10.6?
4. What is the primary cause of glitches that sometimes occur at the output of a decoding gate used with a ripple counter? What is one method to eliminate these glitches?

10.3 SYNCHRONOUS COUNTERS

The ripple counter is the simplest to build, but there is a limit to its highest operating frequency. As previously discussed, each flip-flop has a delay time. In a ripple counter these delay times are additive, and the total "settling" time for the counter is approximately the delay time times the total number of flip-flops. Furthermore, there is the possibility of glitches occurring at the output of decoding gates used with a ripple counter. The first problem fully and the second problem, to some extent can be overcome by the use of a synchronous parallel counter. The main difference here is that every flip-flop is triggered in synchronism with the clock. Note that *strobing* as the solution to glitches has been discussed before in a separate subsection of Section 7.7 of Chapter 7.

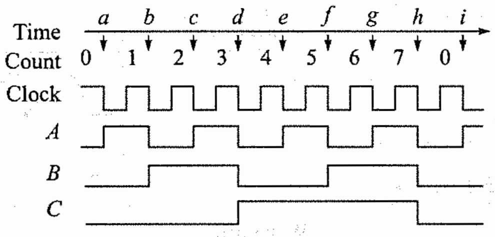
The construction of one type of parallel binary counter is shown in Fig. 10.10, along with the truth table and the waveforms for the natural count sequence. Since each state corresponds to an equivalent binary number (or count), we refer to each state as a count from now on. The basic idea here is to keep the J and K



(a)

C	B	A	Count
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7
0	0	0	0

(b)



(c)

Fig. 10.10 **Mod-8 binary counter with parallel clock input**

inputs of each flip-flop high, such that the flip-flop will toggle with any clock NT at its clock input. We then use AND gates to gate every second clock to flip-flop B, every fourth clock to flip-flop C, and so on. This logic configuration is often referred to as "steering logic" since the clock pulses are gated or steered to each individual flip-flop.

The clock is applied directly to flip-flop A. Since the JK flip-flop used responds to a negative transition at the clock input and toggles when both the J and K inputs are high, flip-flop A will change state with each clock NT.

Whenever A is high, AND gate X is enabled and a clock pulse is passed through the gate to the clock input of flip-flop B. Thus B changes state with every other clock NT at points b, d, f, and h on the time line. Since, there is an additional AND gate delay for the clock at B flip-flop in comparison to A flip-flop, it is not a parallel counter in a strict sense of the term.

Since AND gate Y is enabled and will transmit the clock to flip-flop C only when both A and B are high, flip-flop C changes state with every fourth clock NT at points d and h on the time line.

Examination of the waveforms and the truth table reveals that this counter progresses upward in a natural binary sequence from count 000 up to count 111, advancing one count with each clock NT. This is a mod-8 parallel or synchronous binary counter operating in the count-up mode.

Let's see if this counter configuration has cured the glitch problem discussed previously. The waveforms for this counter are expanded and redrawn in Fig. 10.11, and we have accounted for the individual flip-flop propagation times. Study these waveforms carefully and note the following:

1. The clock NT is the mechanism that toggles each flip-flop.
2. Therefore, whenever a flip-flop changes state, it toggles at exactly the same time as all the other flip-flops—in other words, all the flip-flops change states in synchronism!
3. As a result of the synchronous changes of state, it is not possible to produce a glitch at the output of a decoding gate, such as the gate for 6 shown in Fig. 10.11. Therefore, the decoding gates need not be strobed. All the decoding gates in Fig. 10.7 can be used with this counter without fear of glitches!

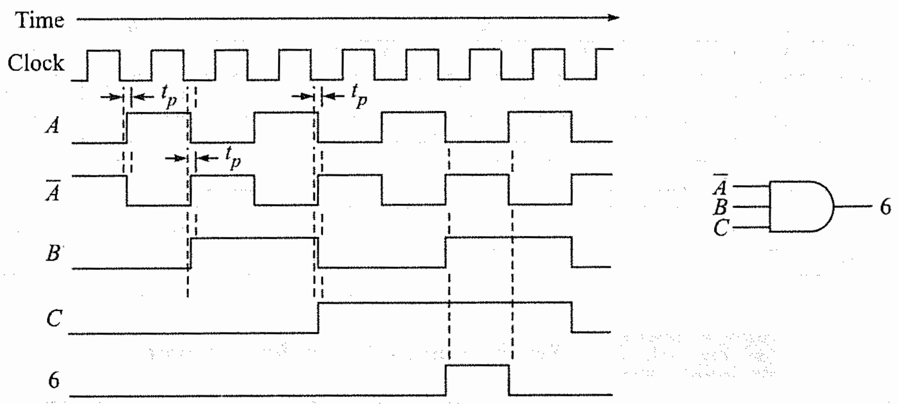
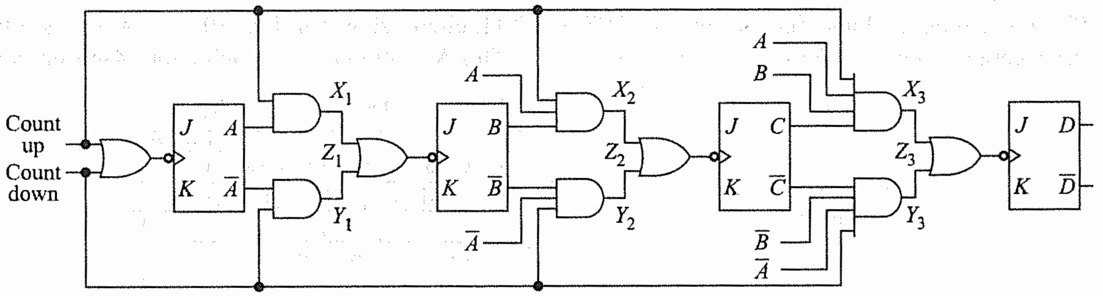


Fig. 10.11

You should take time to compare these waveforms with those generated by the ripple counter as shown in Fig. 10.9.

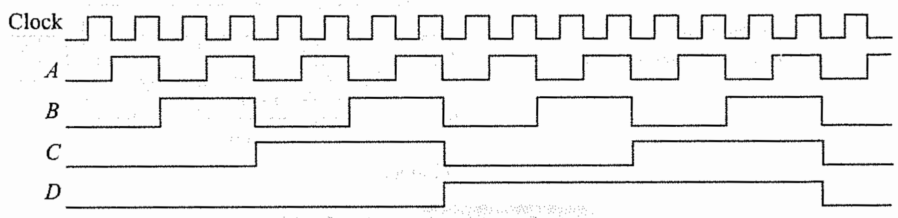
A parallel up-down counter can be constructed in a fashion similar to that shown in Fig. 10.12. In any parallel counter, the time at which any flip-flop changes state is determined by the states of all previous flip-flops in the counter. In the count-up mode, a flip-flop must toggle every time all previous flip-flops are in a 1 state, and the clock makes a transition. In the count-down mode, flip-flop toggles must occur when all prior flip-flops are in a 0 state.

The counter in Fig. 10.12 is a synchronous 4-bit up-down counter. To operate in the count-up mode, the system clock is applied at the count-up input, while the count-down input is held low. To operate in the count-down mode, the system clock is applied at the count-down input while holding the count-up input low.

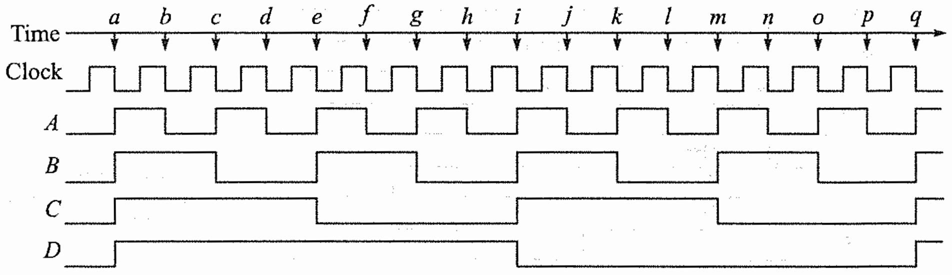


Note : All J and K inputs are tied to $+V_{CC}$.

(a) Logic diagram



(b) Count up waveforms



(c) Count down waveforms

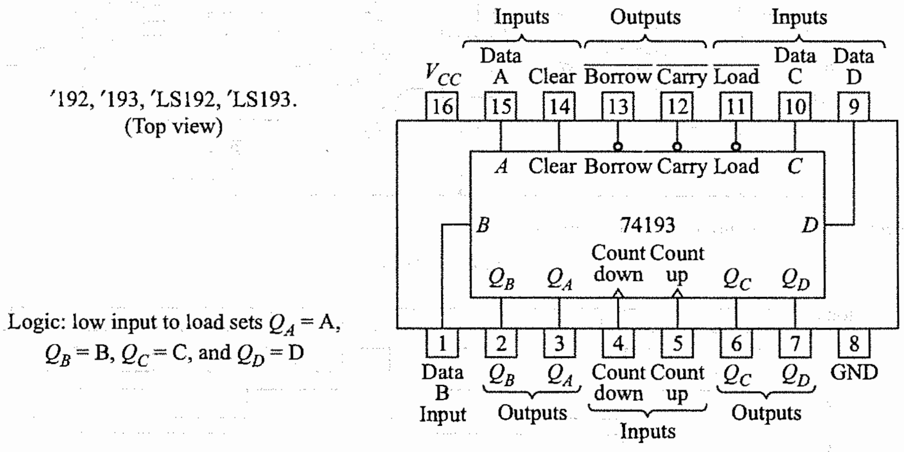
Fig. 10.12 Synchronous, 4-bit up-down counter

Holding the count-down input low (at ground) will disable AND gates Y_1, Y_2, and Y_3. The clock applied at count-up will then go directly into flip-flop A and will be steered into the other flip-flops by AND gates X_1, X_2, and X_3. This counter will then function exactly as the previously discussed parallel counter shown in Fig. 10.10. The only difference here is that this is a mod-16 counter that advances one count with each clock NT, beginning with 0000 and ending with 1111. The correct waveforms are shown in Fig. 10.12b.

If the count-up line is held low, the upper AND gates X_1, X_2, and X_3 are disabled. The clock applied at input count-down will go directly into flip-flop A and be steered into the following flip-flops by AND gates Y_1, Y_2, and Y_3.

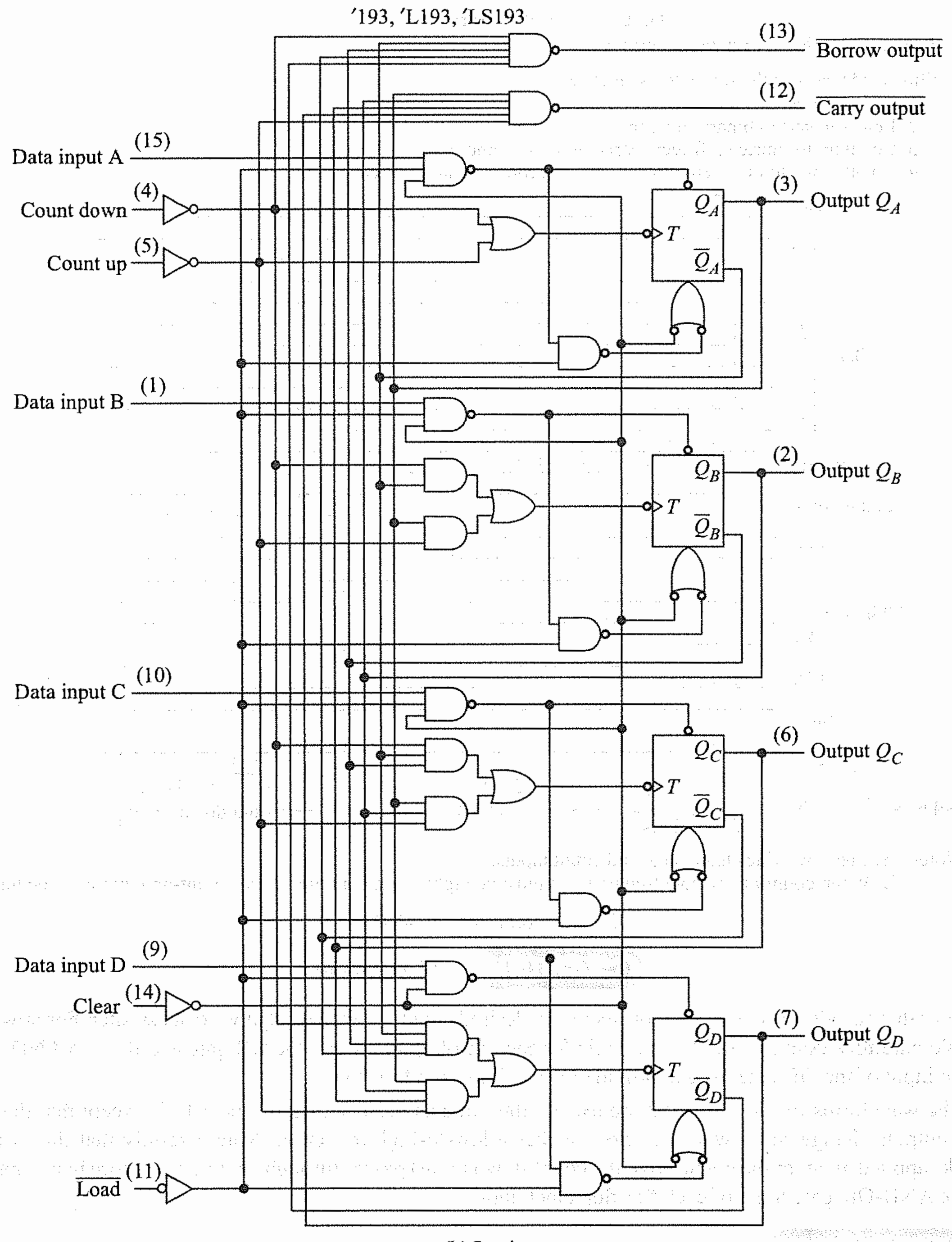
Flip-flop A will toggle each time there is a clock NT as shown in Fig. 10.12c. Each time $\overline{A}$ is high, AND gate Y_1 will be enabled and the clock NT will toggle flip-flop B at points a, c, e, g, and so on. Whenever both $\overline{A}$ and $\overline{B}$ are high, AND gate Y_2 is enabled, and thus a clock will be steered into flip-flop C at points a, e, i, m, and q. Similarly, AND gate Y_3 will steer a clock into flip-flop D only when $\overline{A}$, $\overline{B}$, and $\overline{C}$ are all high. Thus flip-flop D will toggle at points a and i on the time line. The waveforms in Fig. 10.12c clearly show that the counter is operating in a count-down mode, progressing one count at a time from 1111 to 0000.

If you examine the logic diagram for the 54/74193 TTL circuit shown in Fig. 10.13, you will see that it uses steering logic just like the counter in Fig. 10.12. This MSI circuit is a synchronous 4-bit up-down



(a) Pinout

Fig. 10.13 54/74193



(b) Logic

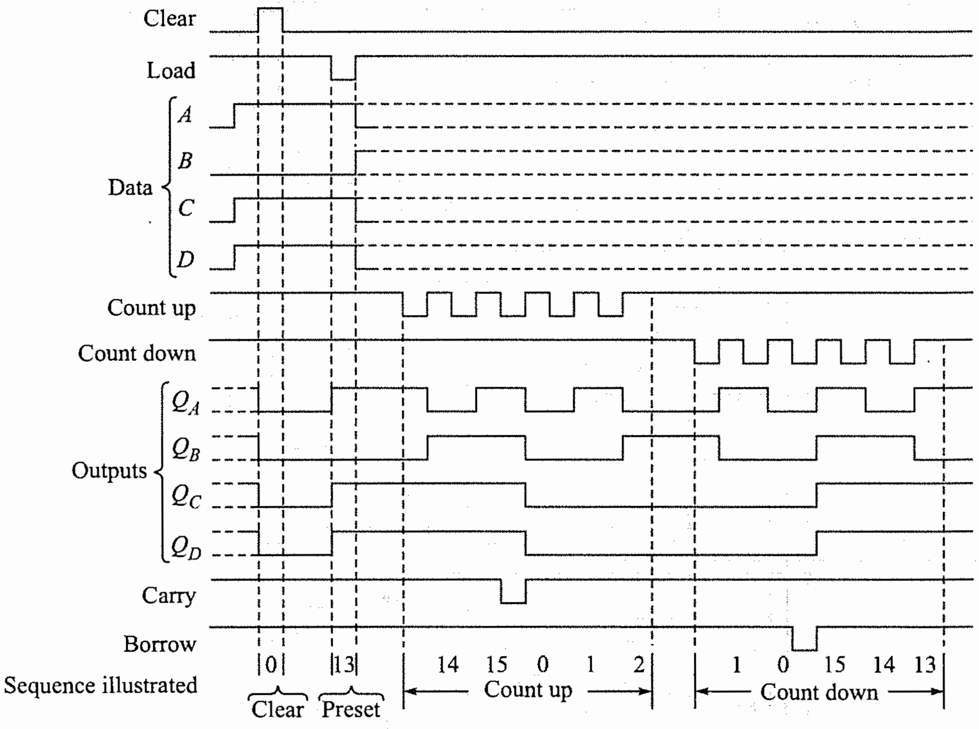
Fig. 10.13 **(Continued)**

′193, ′L193, ′LS193 Binary counters

Typical clear, load, and count sequences

Illustrated below is the following sequence.

1. Clear outputs to zero.
2. Load (preset) to binary thirteen.
3. Count up to fourteen, fifteen, carry, zero, one, and two.
4. Count down to one, zero, borrow, fifteen, fourteen, and thirteen.



Notes: A. Clear overrides load, data, and count inputs.
B. When counting up, count-down input must be high; when counting down, count-up input must be high.

(c) Waveforms for 54/74193

Fig. 10.13 **(Continued)**

counter that can also be cleared and preset to any desired count—attributes that we discuss later. For now, you should carefully examine the steering logic for each flip-flop and study the OR gate and the two AND gates at the input of the OR gate used to provide the clock to each flip-flop.

The waveforms for the 54/74193 are exactly the same as those shown in Fig. 10.12, except that the flip-flop outputs change states when the clock makes a low-to-high transition. Note carefully that the external clock (applied at either the count-up or the count-down input) passes through an inverter before being applied to the AND-OR-gate logic of each flip-flop clock input.

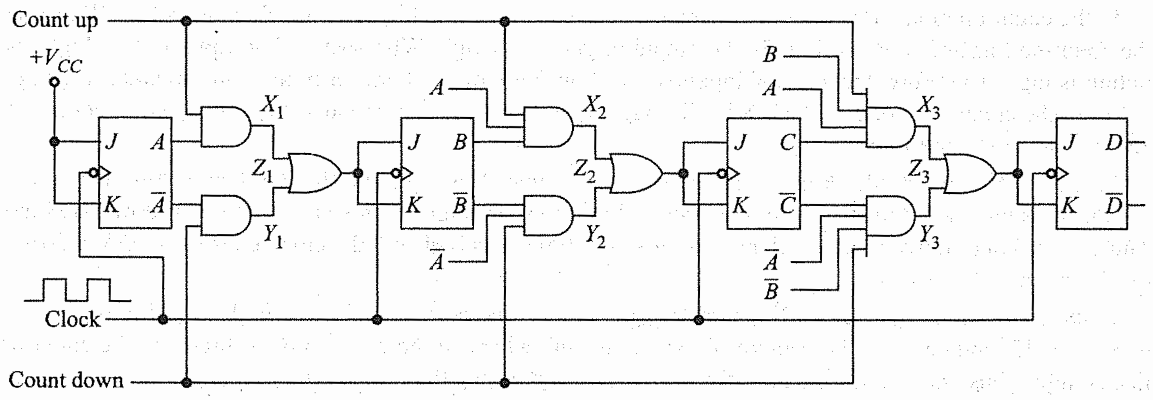
Example 10.5 Write a Boolean expression for the AND gate connected to the lower leg of the OR gate that drives the clock input to flip-flop Q_D in the 54/74193.

Solution The correct expression is

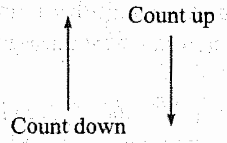
$$x = \overline{(\text{count-up clock})}\,(Q_A)(Q_B)(Q_C)$$

A parallel up-down counter can be formed by using a slightly different logic scheme, as shown in Fig. 10.14. Remember that in a parallel counter, the time at which any flip-flop changes state is determined by the states of all previous flip-flops in the counter. In the count-up mode, a flip-flop must toggle every time all previous flip-flops are in a 1 state, and the clock makes a transition. In the count-down mode, flip-flop toggles must occur when all prior flip-flops are in a 0 state.

This particular counter works in an *inhibit mode*, since each flip-flop changes state on a clock NT provided its J and K inputs are both high; a change of state will not occur when the J and K inputs are low. We



(a)



D	C	B	A	Count
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15
0	0	0	0	0

(b)

Fig. 10.14 **Parallel up-down counter**

might consider this is "look-ahead logic," since the mode of operation occurs in a strict time sequence as follows:

1. Establish a level on the J and K inputs (low or high)
2. Let the clock transition high to low
3. Look at the flip-flop output to determine whether it toggled.

To understand the logic used to implement this counter, refer to the truth table shown in Fig. 10.14b.

A is required to change state each time the clock goes low, and flip-flop A therefore has both its J and K inputs held in a high state. This is true in both the count-up and count-down modes, and thus no other logic is necessary for this flip-flop.

In the count-up mode, B is required to change state each time A is high and the clock goes low. Whenever the count-up line and A are both high, the output of gate X_1 is high. Whenever either input to Z_1 is high, the output is high. Therefore, the J and K inputs to flip-flop B are high whenever both count-up and A are high. Then, in the count-up mode, a clock NT will toggle B each time A is high, such as in going from count 1 to count 2, or 3 to 4, and so on.

In the count-down mode, B must change state each time $\overline{A}$ is high and the clock goes low. The output of gate Y_1 is high, and thus the J and K inputs to flip-flop B are high whenever $\overline{A}$ and count-down are high. Thus, in the count-down mode, B changes state every time $\overline{A}$ is high and the clock goes low—going from 0 to 15, or from 14 to 13, etc.

In the count-up mode, a clock NT must toggle C every time both A and B are high (transitions 3 to 4, 7 to 8, 11 to 12, and 15 to 0). The output of gate X_2 is high whenever both A and B are high and the count-up line is high. Thus, the J and K inputs to flip-flop C are high during these times and C changes state during the needed transitions.

In the count-down mode, C is required to change state whenever both $\overline{A}$ and $\overline{B}$ are high. The output of gate Y_2 is high any time both $\overline{A}$ and $\overline{B}$ are high, and the count-down line is high. Thus the J and K inputs to flip-flop C are high during these times, and C then changes state during the required transitions—that is, 0 to 15, 12 to 11, 8 to 7, and 4 to 3.

In the count-up mode, D must toggle every time $\overline{A}$, $\overline{B}$, and $\overline{C}$ are all high. The output of gate X_3 is high, and thus the J and K inputs to flip-flop D are high whenever A, B, and C and count-up are all high. Thus D changes state during the transitions from 7 to 8 and from 15 to 0.

In the count-down mode, a clock NT must toggle D whenever A, B, and C are all high. The output of gate Y_3 is high, and thus the J and K inputs to flip-flop D are high whenever $\overline{A}$, $\overline{B}$, and $\overline{C}$ and count-down are all high. Thus D changes state during the transitions from 0 to 15 and from 8 to 7. The count-up and count-down waveforms for this counter are exactly like those shown in Fig. 10.12.

Take a look at the logic diagram for the 54/74191 TTL MSI circuit shown in Fig. 10.15. This is a synchronous up-down counter. A careful examination of the AND-OR-gate logic used to precondition the J and K inputs to each flip-flop will reveal that this counter uses look-ahead logic exactly like the counter in Fig. 10.14. Additional logic allows one to clear or preset this counter to any desired count, and we study these functions later. For now, carefully compare the logic diagram with the counter in Fig. 10.14 to be certain you understand its operation.

Notice carefully that the clock input passes through an inverter before it is fed to the individual flip-flops. Thus the outputs of the four master-slave flip-flops will change states only on low-to-high transitions of the input clock. Typical waveforms are given in Fig. 10.15. Incidentally, these are precisely the same waveforms one would expect when using the 54/74193 discussed previously.

Example 10.6 Write a Boolean expression for the 4-input AND gate connected to the lower leg of the OR gate that conditions the J and K inputs to the Q_D flip-flop in a 54/74191.

Solution The correct logic expression is

$$x = \overline{(\text{down}-\text{up})}\,(Q_A)(Q_B)(Q_C)\,\overline{(\text{enable})}$$

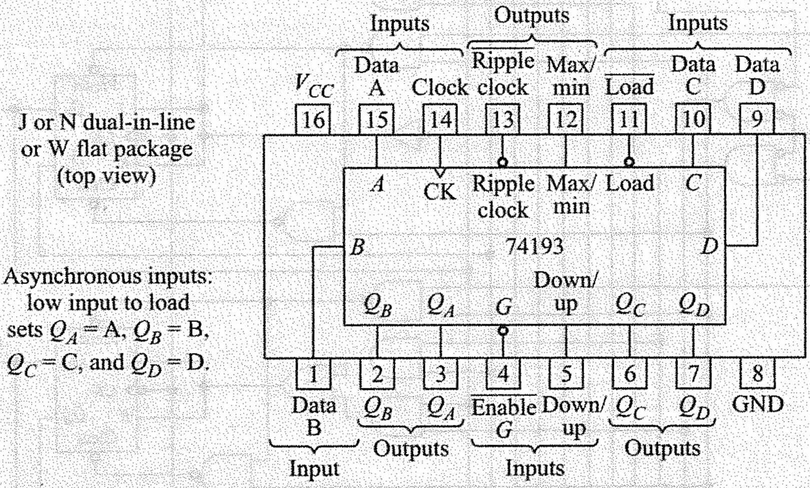


Fig. 10.15 **54/74191 (continued on next page)**

SELF-TEST

5. How does a parallel (synchronous) counter differ from a serial (asynchronous) counter?
6. Why are decoding gate glitches eliminated in a synchronous counter?
7. Does the 74193 change state with PTs or with NTs?

10.4 CHANGING THE COUNTER MODULUS

Counter Modulus

At this point, we have discussed asynchronous (ripple) counters and two different types of synchronous (parallel) counters, all of which have the ability to operate in either a count-up or count-down mode. All of these counters progress one count at a time in a strict binary progression, and they all have a modulus given by 2^n, where n indicates the number of flip-flops. Such counters are said to have a "natural count" of 2^n.

A mod-2 counter consists of a single flip-flop; a mod-4 counter requires two flip-flops, and it counts through four discrete states. Three flip-flops form a mod-8 counter, while four flip-flops form a mod-16 counter. Thus we can construct counters that have a natural count of 2, 4, 8, 16, 32, and so on by using the proper number of flip-flops.

It is often desirable to construct counters having a modulus other than 2, 4, 8, and so on. For example, a counter having a modulus of 3, or 5, would be useful. A small modulus counter can always be constructed

′191, ′LS191 Binary counters

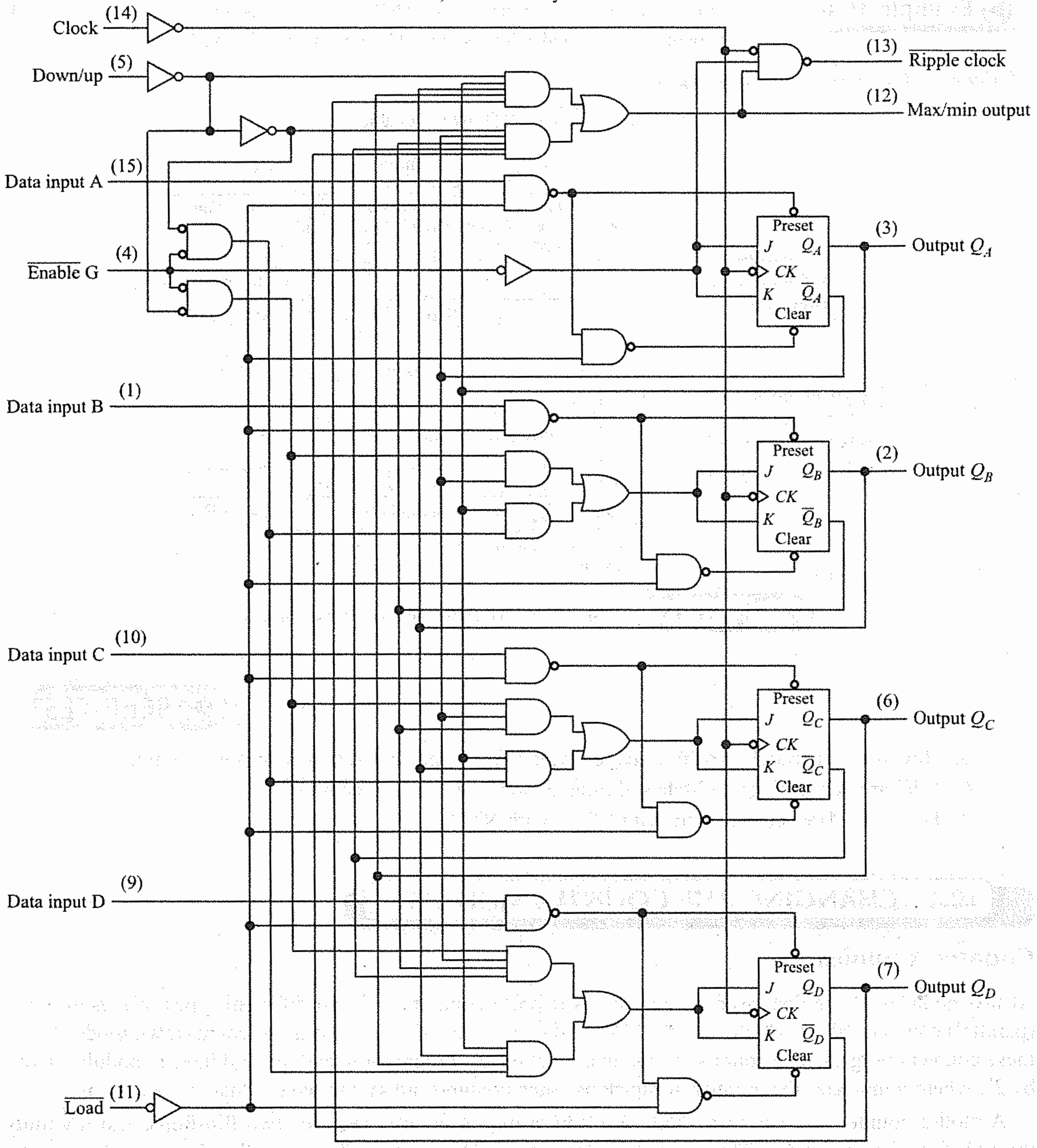


Fig. 10.15 (Continued)

'191, 'LS191 Binary counters

Typical load, count and inhibit sequences

Illustrated below is the following sequence.

1. Load (preset) to binary thirteen.
2. Count up to fourteen, fifteen (maximum), zero, one, and two.
3. Inhibit
4. Count down to one, zero (minimum), fifteen, fourteen, and thirteen.

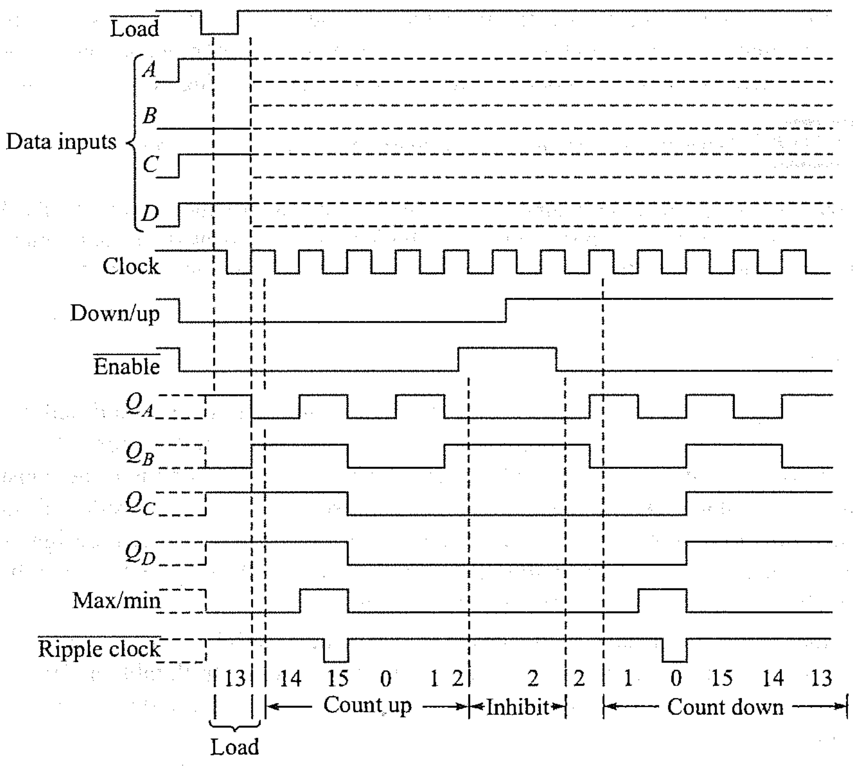


Fig. 10.15 **(Continued)**

from a larger modulus counter by skipping states. Such counters are said to have a *modified count.* It is first necessary to determine the number of flip-flops required. The correct number of flip-flops is determined choosing the lowest natural count that is greater than the desired modified count. For example, a mod-7 counter requires three flip-flops, since 8 is the lowest natural count greater than the desired modified count of 7.

Example 10.7 Indicate how many flip-flops are required to construct each of the following counters: (a) mod-3, (b) mod-6, and (c) mod-9.

Solution

a. The lowest natural count greater than 3 is 4. Two flip-flops provide a natural count of 4. Therefore, it requires at least two flip-flops to construct a mod-3 counter.

b. Construction of a mod-6 counter requires at least three flip-flops, since 8 is the lowest natural count greater than 6.

c. A mod-9 counter requires at least four flip-flops, since 16 is the lowest natural count greater than 9.

A single flip-flop has a natural count of 2; thus we could use a single flip-flop to construct a mod-2 counter, and that's all. However, a two flip-flop counter has a natural count of 4. Skipping one count will lead to a mod-3 counter. So, two flip-flops can be used to construct either a mod-4 or mod-3 counter.

Similarly, a three-flip-flop counter has a natural count of 8, but by skipping counts we can use three flip-flops to construct a counter having a modulus of 8, 7, 6, or 5. Note that counters having a modulus of 4 or 3 could also be constructed, but these two counters can be constructed by using only two flip-flops.

Example 10.8 What modulus counters can be constructed with the use of four flip-flops?

Solution A four-flip-flop counter has a natural count of 16. We can thus construct any counter that has a modulus between 16 and 2, inclusive. We might choose to use four flip-flops only for counters having a modulus between 16 and 9, since only three flip-flops are required for a modulus of less than 8, and only two are required for a modulus of less than 4.

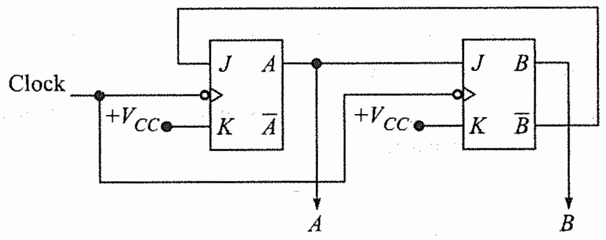
A Mod-3 Counter

There are a great many different methods for constructing a counter having a modified count. A counter can be synchronous, asynchronous, or a combination of these two types; furthermore, there is the decision of which count to skip. For instance, if a mod-6 counter using three flip-flops is to be constructed, which two of the eight discrete states should be skipped? Our purpose here is not to consider all possible counter configurations and how to design them; rather, we devote our efforts to one or two designs widely used in TTL MSI. A mod-3 counter is considered in this section and a mod-5 in the next section, and then we consider the use of presettable counters to achieve any desired modulus.

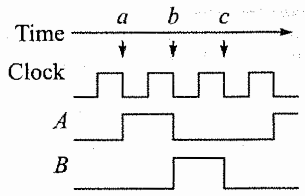
The two flip-flops in Fig. 10.16 have been connected to provide a mod-3 counter. Since two flip-flops have a natural count of 4, this counter skips one state. The waveforms and the truth table in Fig. 10.16 show that this counter progresses through the count sequence 00, 01, 10, and then back to 00. It clearly skips count 11. Here's how it works:

1. Prior to point *a* on the time line, $A = 0$ and $B = 0$. A negative clock transition at *a* will cause:
   a. A to toggle to a 1, since its J and K inputs are high
   b. B to reset to 0 (it's already a 0), since its J input is low and its K input is high
2. Prior to point *b* on the time line, $A = 1$, and $B = 0$. A negative clock transition at *b* will cause:
   a. A to toggle to a 0, since its J and K inputs are high
   b. B to toggle to a 1, since its J and K inputs are high
3. Prior to point *c* on the time line, $A = 0$ and $B = 1$. A negative clock transition at *c* will cause:
   a. A to reset to 0 (it's already 0), since its J input is low and its K input is high
   b. B to reset to 0 since its J input is low and its K input is high
4. The counter has now progressed through all three of its states, advancing one count with each negative clock transition.

This two-flip-flop mod-3 counter can be considered as a logic building block as shown in Fig. 10.16d. It has a clock input and outputs at A and B. It can be considered as a divide-by-3 block, since the output



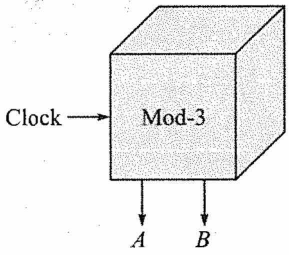
(a) Logic diagram



(b) Waveforms

B	A	Count
0	0	0
0	1	1
1	0	2
0	0	0

(c) Truth table



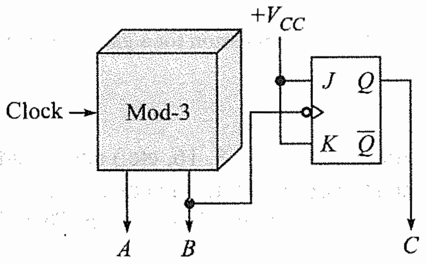
(d) Logic block

Fig. 10.16 **Mod-3 counter**

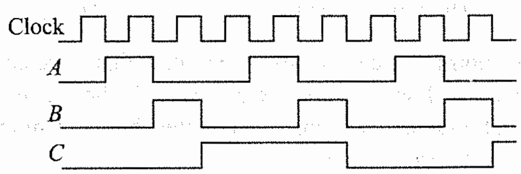
waveform at *B* (or at *A*) has a period equal to three times that of the clock—in other words, this counter divides the clock frequency by 3. Notice that this is a synchronous counter since both flip-flops change state in synchronism with the clock.

A Mod-6 Counter

If we consider a basic flip-flop to be a mod-2 counter, we see that a mod-4 counter (two flip-flops in series) is simply two mod-2 counters in series. Similarly, a mod-8 counter is simply a 2 × 2 × 2 connection, and so on. Thus a great number of higher-modulus counters can be formed by using the product of any number of lower-modulus counters. For instance, suppose that we connect a flip-flop at the B output of the mod-3 counter in Fig. 10.16. The result is a (3 × 2 = 6) mod-6 counter as shown in Fig. 10.17. The output of the single flip-flop is labeled *C*. Notice that it is a symmetrical waveform, and it also has a frequency of one-sixth that of the input clock. Also, this can no longer be considered a synchronous counter since flip flop *C* is triggered by flip-flop *B*; that is, the flip-flops do not all change status in synchronism with the clock.



(a) 3 × 2 Mod-6 counter

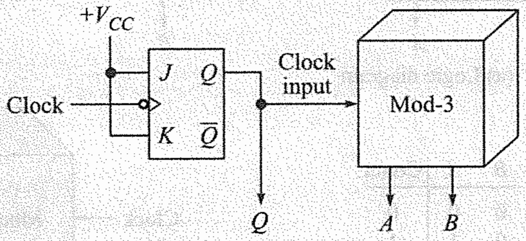


(b) Waveforms

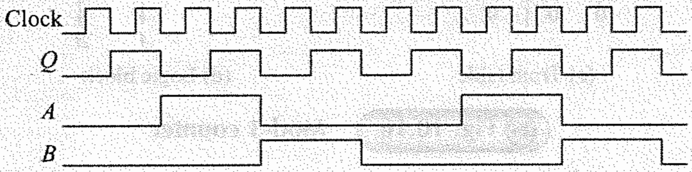
Fig. 10.17

Example 10.9 Draw the waveforms you would expect from the mod-6 counter by connecting a single flip-flop in front of the mod-3 counter in Fig. 10.16.

Solution The resulting counter is a 2×3 = mod-6 counter that has the waveforms shown in Fig. 10.18. Notice that B now has a period equal to six clock periods, but it is not symmetrical.



(a) 2×3 Mod-6 counter



(b) Waveforms

Fig. 10.18

The 54/7492A

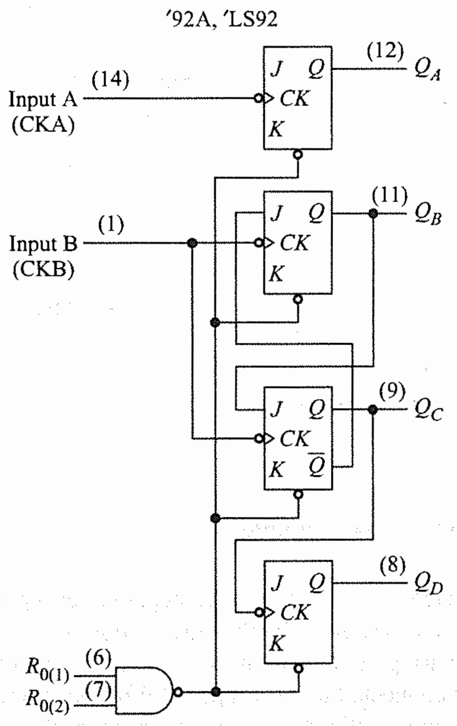
The 54/7492A ('92A) in Fig. 10.19 is a TTL divide-by-12, MSI counter. A careful examination of the logic diagram will reveal that flip-flops Q_B, Q_C, and Q_D are exactly the same as the 3×2 counter in Fig. 10.17. Thus if the clock is applied to input B of the '92A and the outputs are taken at Q_B, Q_C, and Q_D, this is a mod-6 counter.

On the other hand, if the clock is applied at input A and Q_A is connected to input B, we have a $2 \times 3 \times 2$ mod-12 counter. The proper truth table for the mod-12 configuration is given in Fig. 10.19b. Again, this must be considered as an asynchronous counter since all flip-flops do not change states at the same time. Thus there is the possibility of glitches occurring at the outputs of any decoding gates used with the counter.

Example 10.10 Use the truth table for the '92A to write a Boolean expression for a gate to decode count 8.

Solution The correct expression is "8" = $Q_D Q_{\overline{C}} Q_B Q_{\overline{A}}$.

At this point, we can construct counters that have any natural count (2, 4, 8, 16, etc.) and, in addition, a mod-3 counter. Furthermore, we can cascade these counters in any combination, such as 2×2, 2×3, 3×4, and so on. So far we can construct counters having a modulus of 2, 3, 4, 6, 8, 9, 12, and so on. Therefore, let's consider next a mod-5 counter.

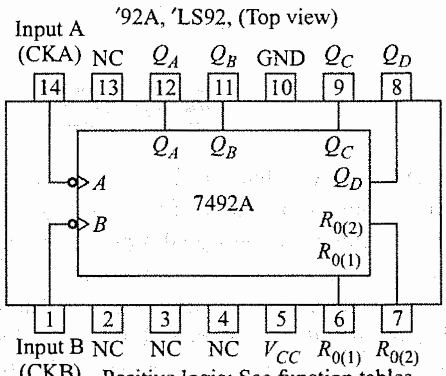


(a) Logic

'92A, 'LS92 Count sequence
(See Note C)

Count	Output			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	H	L	L	L
7	H	L	L	H
8	H	L	H	L
9	H	L	H	H
10	H	H	L	L
11	H	H	L	H

(b) Truth table



Positive logic: See function tables.

Note: Output Q_A connected to input B

(c) Pinout

Fig. 10.19 54/7492A

SELF-TEST

8. How many flip-flops are required to construct a mod-12 counter?
9. Three flip-flops are available. What modulus counters could be constructed?

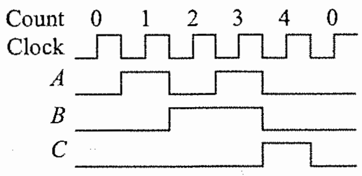
10.5 DECADE COUNTERS

A Mod-5 Counter

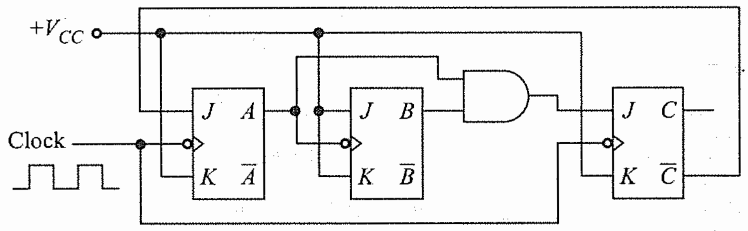
The three-flip-flop counter shown in Fig. 10.20 has a natural count of 8, but it is connected in such a way that it will skip over three counts. It will, in fact, advance one count at a time, through a strict binary sequence, beginning with 000 and ending with 100; therefore, it is a mod-5 counter. Let's see how it works.

C	B	A	Count
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
0	0	0	0

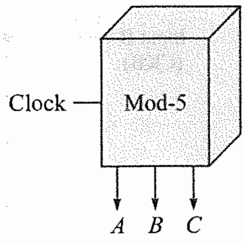
(a)



(b)



(c)



(d) Logic block

Fig. 10.20 **Mod-5 binary counter**

The waveforms show that flip-flop A changes state each time the clock goes negative, except during the transition from count 4 to count 0. Thus, flip-flop A should be triggered by the clock and must have an inhibit during count 4—that is, some signal must be provided during the transition from count 4 to count 0. Notice that $\overline{C}$ is high during all counts except count 4. If $\overline{C}$ is connected to the J input of flip-flop A, we will have the desired inhibit signal. This is true since the J and K inputs to flip-flop A are both true for all counts except count 4; thus the flip-flop triggers each time the clock goes negative. However, during count 4, the J side is low and the next time the clock goes negative the flip-flop will be prevented from being set. The connections which cause flip-flop A to progress through the desired sequence are shown in Fig. 10.20.

The desired waveforms (Fig. 10.20b) show that flip-flop B must change state each time A goes negative. Thus the clock input of flip-flop B will be driven by A (Fig. 10.20c).

If flip-flop C is triggered by the clock while the J input is held low and the K input high, every clock pulse will reset it. Now, if the J input is high only during count 3, C will be high during count 4 and low during all other counts. The necessary levels for the J input can be obtained by ANDing flip-flops A and B. Since A and B are both high only during count 3, the J input to flip-flop C is high only during count 3. Thus, when the clock goes negative during the transition from count 3 to count 4, flip-flop C will be set. At all other times, the J input to flip-flop C is low and is held in the reset state. The complete mod-5 counter is shown in Fig. 10.20.

In constructing a counter of this type, it is always necessary to examine the omitted states to make sure that the counter will not malfunction. This counter omits states 5, 6, and 7 during its normal operating sequence. There is however, a very real possibility that the counter may set up in one of these omitted (illegal) states when power is first applied to the system. It is necessary to check the operation of the counter when starting from each of the three illegal states to ensure that it progresses into the normal count sequence and does not become inoperative.

Begin by assuming that the counter is in state 5 (*CBA* = 101). When the next clock pulse goes low, the following events occur:

1. Since $\overline{C}$ is low, flip-flop *A* resets. Thus *A* changes from a 1 to a 0.
2. When *A* goes from a 1 to a 0, flip-flop *B* triggers and *B* changes from a 0 to a 1.
3. Since the *J* input to flip-flop *C* is low, flip-flop *C* is reset and *C* changes from a 1 to a 0.
4. Thus the counter progresses from the illegal state 5 to the legal state 2 (*CBA* = 010) after one clock.

Now, assume that the counter starts in the illegal state 6 (*CBA* = 110). On the next negative clock transition, the following events occur:

1. Since $\overline{C}$ is low, flip-flop *A* is reset. Since *A* is already a 0, it just remains a 0.
2. Since *A* does not change, flip-flop *B* does not change and *B* remains a 1.
3. Since the *J* input to flip-flop *C* is low, flip-flop *C* is reset and *C* changes from a 1 to a 0.
4. Thus the counter progresses from the illegal state 6 to the legal state 2 after one clock transition.

Finally, assume that the counter begins in the illegal state 7 (*CBA* = 111). On the next negative clock transition, the following events occur:

1. Since $\overline{C}$ is low, flip-flop *A* is reset and *A* changes from a 1 to a 0.
2. Since *A* changes from a 1 to a 0, flip-flop *B* triggers and *B* changes from a 1 to a 0.
3. The *J* input to flip-flop *C* is high; therefore, flip-flop *C* toggles from a 1 to a 0.
4. Thus the counter progresses from the illegal count 7 to the legal count 0 after one clock transition.

None of the three illegal states will cause the counter to malfunction, and it will automatically work itself out of any illegal state after only one clock transition.

A Mod-10 Counter

This mod-5 counter configuration can be considered as a logic block as shown in Fig. 10.20d and can be used in cascade to construct higher-modulus counters. For instance, a 2 × 5 or a 5 × 2 will form a mod-10 counter, or *decade counter*.

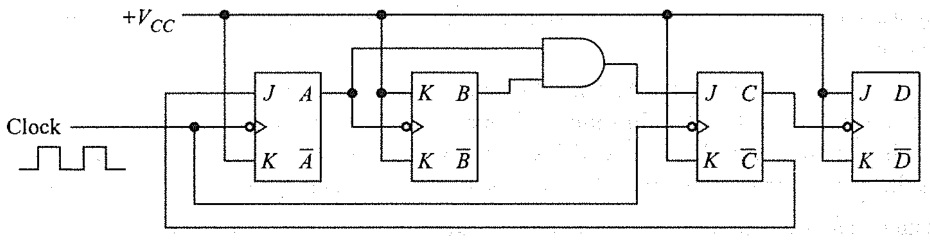
Example 10.11 Show a method for constructing a 5 × 2 (mod-10) decade counter.

Solution A decade counter can be constructed by using the mod-5 counter in Fig. 10.20 and adding an additional flip-flop, labeled *D*, as shown in Fig. 10.21. The appropriate waveforms and truth table are included. Notice that the counter progresses through a *biquinary* count sequence and does not count in a straight binary sequence.

A decade counter could be formed just as easily by using the mod-5 counter in Fig. 10.20 in conjunction with a flip-flop, but connected in a 2 × 5 configuration as shown in Fig. 10.22. The truth table for this configuration, and the resulting waveforms are shown. This is still a mod-10 (decade) counter since it still has 10 discrete states. Notice that this counter counts in a straight binary sequence from 0000 up to 1001, and then back to 0000.

The 7490A

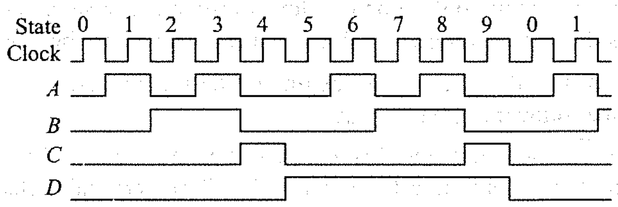
The 54/7490A is a TTL MSI decade counter. Its logic diagram, truth table, and pinout are given in Fig. 10.23. A careful examination will reveal that flip-flops Q_B, Q_C, and Q_D form a mod-5 counter exactly like the one in Fig. 10.20. Notice, however, that flip-flop Q_D in the '90A is an *RS* flip-flop that has a direct connection from its *Q* output back to its *R* input. The net result in this case is that Q_D behaves exactly like a *JK* flip-flop.



(a)

D	C	B	A	State
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
1	0	0	0	5
1	0	0	1	6
1	0	1	0	7
1	0	1	1	8
1	1	0	0	9
0	0	0	0	0

(b)

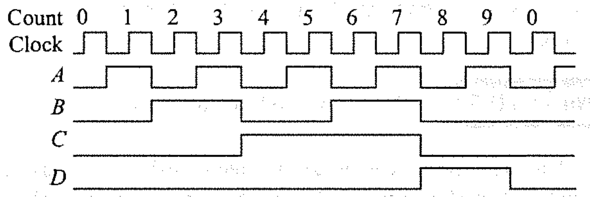


(c)

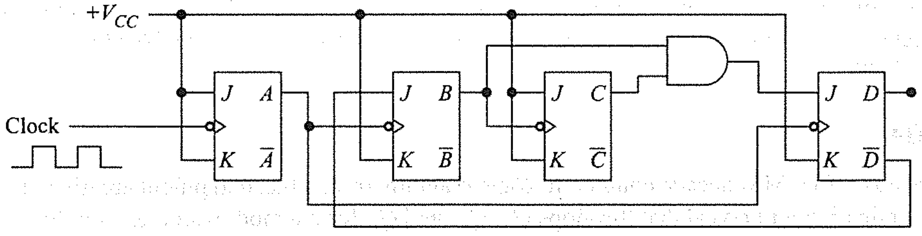
Fig. 10.21 A decade counter

D	C	B	A	Count
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
0	0	0	0	0

(a)

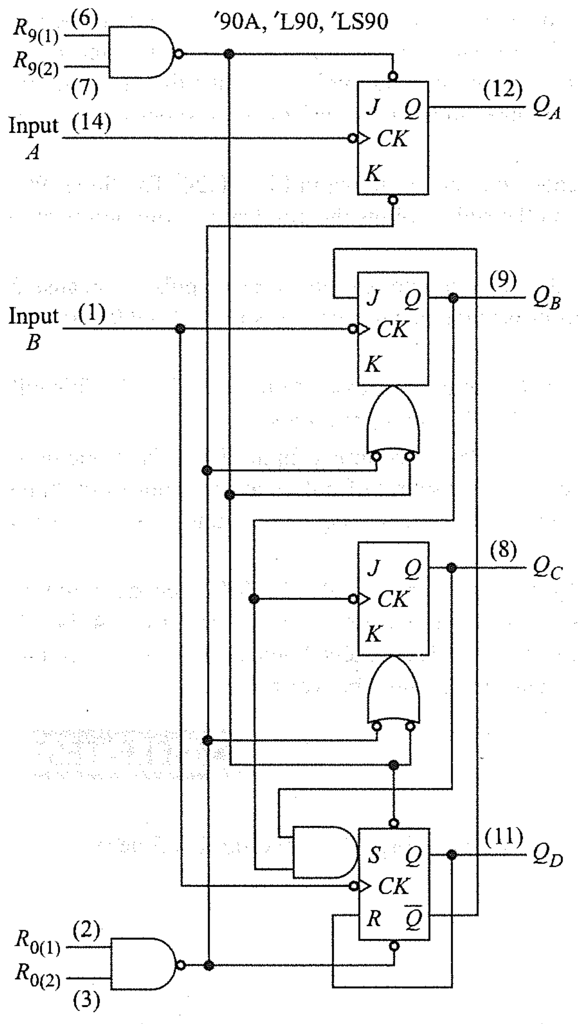


(b)



(c)

Fig. 10.22 A decade counter



(a) Logic

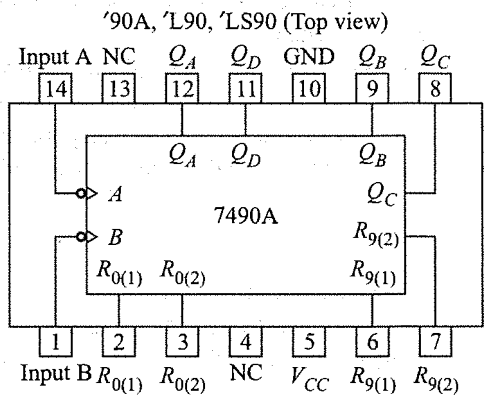
'90A, 'L90, 'LS90
BCD count sequence
(See note A)

Count	Output			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H

'90A, 'L90, 'LS90
Bi-quinary (5-2)
(See note B)

Count	Output			
	Q_A	Q_D	Q_C	Q_B
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	H	L	L	L
6	H	L	L	H
7	H	L	H	L
8	H	L	H	H
9	H	H	L	L

(b) Truth table



Positive logic: See function tables

Note: A output Q_A connected to input B
B output Q_D connected to input A

(c) Pinout

Fig. 10.23 **54/7490A**

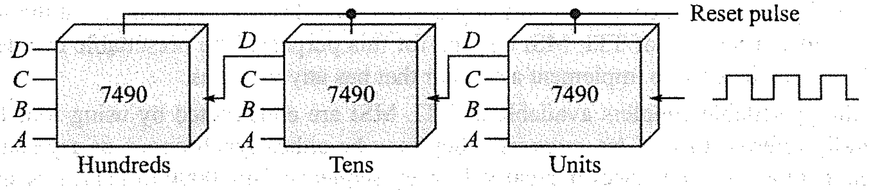


Fig. 10.24 **Cascaded 7490's can count to 999**

If the system clock is applied at input A and Q_A is connected to input B, we have a true binary decade counter exactly as in Fig. 10.22. On the other hand, if the system clock is applied at input B and Q_D is connected to input A, we have the biquinary counter as discussed in Example 10.11. Take time to study the logic diagram and the truth table for the '90A; it is widely used in industry, and the time spent will be well worth your while.

An interesting application using three 54/7490A decade counters is shown in Fig. 10.24. The three '90A counters are connected in series such that the first one (on the right) counts the number of input pulses at its clock input. We call it a *units counter*.

The middle '90A will advance one count each time the units counter counts 10 input pulses, because D from the units counter will have a single negative transition as that counter progresses from count 9 to 0. This middle block is then called *tens counter*.

The left '90A will advance one count each time the tens counter progresses from count 9 to 0. This will occur once for every 100 input pulses. Thus this block is called the *hundreds counter*.

Now the operation should be clear. This logic circuit is capable of counting input pulses from one up to 999. The procedure is to reset all the '90As and then count the number of pulses at the input to the units counter. This cascaded arrangement is widely used in digital voltmeters, frequency counters, etc., where a decimal count is needed.

It should be pointed out that the 54/7490A is only one of a number of TTL MSI decade counters. In particular, the 54/74176 is another popular asynchronous decade counter, and the 54/74160, 54/74162, 54/74190, and 54/74192 are all popular synchronous decade counters. Each has particular attributes that you should consider, and a study of their individual data sheets would be worthwhile.

SELF-TEST

10. What is a decade counter?
11. What is the difference between the 5 × 2 decade counter in Fig. 10.21 and the 2 × 5 decade counter in Fig. 10.22?

10.6 PRESETTABLE COUNTERS

Up to this point we have discussed the operation of counters that progress through a natural binary count sequence in either a count-up or count-down mode and have studied two counters that have a modified count—a mod-3 and a mod-5. With these basic configurations, and with cascaded combinations of these basic units, it is possible to construct counters having moduli of 2, 3, 4, 5, 6, 7, 8, 9, 10, and so on. The ability to quickly and easily construct a counter having any desired modulus is so important that the semiconductor industry has provided a number of TTL MSI circuits for this purpose. The presettable counter is the basic building block that can be used to implement a counter that has any modulus.

Nearly all the presettable counters available as TTL MSI are constructed by using four flip-flops, and they are generally referred to as *4-bit counters*. They may be either synchronous or asynchronous. When connected such that the count advances in a natural binary sequence from 0000 to 1111, it is simply referred to as a *binary counter*. For instance, the 54/74161 and the 54/74163 are both synchronous binary counters that operate in a count-up mode. The 54/74191 and the 54/74193 are also synchronous binary counters, but they can operate in either a count-down or count-up mode.

Since the decade counter is a very important and useful configuration, many of the "basic 4-bit counters are internally connected to provide a modified count of 10—a mod-10 or decade counter. For instance, the 54/74160 and the 54/74162 are synchronous decade counters that operate in the count-up mode. The 54/74190 and the 54/74192 are also synchronous decade counters but they can operate in either a count-up or count-down mode.

The counters mentioned above are all TTL MSI circuits, and as such we have little control over the internal logic used to implement each counter. Our concern is directed at how each unit can be used in a digital system. Thus we consider each of these counters as a logic block, and our efforts are concentrated on inputs, outputs, and control signals. Even so, the logic block diagram is given for each counter, since a knowledge of the internal logic gives a depth of understanding that is invaluable in practical applications.

Synchronous Up Counters

The pinout and logic diagram for a 54/74163 synchronous 4-bit counter are given in Fig. 10.25. The pinout contains a logic block diagram for this unit. The power requirements are $+V_{CC}$ and GROUND on pins 16 and 8, respectively. The "clock" is applied on pin 2, and you will notice from the diagram that the outputs change states on positive clock transitions (PTs).

The four flip-flop outputs are Q_A, Q_B, Q_C, and Q_D, while the CARRY output on pin 15 can be used to enable successive counter stages (e.g. in a units, tens, hundreds application).

The two ENABLE inputs (P on pin 7 and T on pin 10) are used to control the counter. If either ENABLE input is low, the counter will cease to advance; both of these inputs must be high for the counter to count.

A low level on the $\overline{\text{CLEAR}}$ input will reset all flip-flop outputs low at the very next clock transition, regardless at the levels on the ENABLE inputs. This is called a *synchronous reset* since it occurs at a positive clock transition. On the other hand, note that the 54/74161 has an asynchronous clear, since it occurs immediately when the $\overline{\text{CLEAR}}$ input goes low, regardless of the levels on the CLOCK, ENABLE, or $\overline{\text{LOAD}}$ inputs.

When a low level is applied to the $\overline{\text{LOAD}}$ input, the counter is disabled, and the very next positive clock transition will set the flip-flops to agree with the levels present on the four data inputs (D, C, B, and A). For instance, suppose that the data inputs are $DCBA$ = 1101, and the $\overline{\text{LOAD}}$ input is taken low. The very next positive clock transition will load these data into the counter and the outputs will become $Q_DQ_CQ_BQ_A$ = 1101. This is a very useful function when it is desired to have the counter begin counting from a predetermined count.

For the counter to count upward in its normal binary count sequence, it is necessary to hold the ENABLE inputs (P and T), the $\overline{\text{LOAD}}$ input, and the CLEAR input all high. Under these conditions, the counter will advance one count for each positive clock transition, progressing from count 0000 up to count 1111 and then repeating the sequence. Since the flip-flops are clocked synchronously, the outputs change states simultaneously and there are no counting spikes or glitches associated with the counter outputs. The state diagram given in Fig. 10.26a show the normal count sequence, where each box corresponds to one count (or state) and the arrows show how the counter progresses from one state to the next.

The count length can be very easily modified by making use of the synchronous $\overline{\text{CLEAR}}$ input. It is a simple matter to use a NAND gate to decode the maximum count desired, and use the output of this NAND gate to clear the counter synchronously to count 0000. The counter will then count from 0000 up to the maximum desired count and then clear back to 0000. This is the technique that can be used to construct a counter that has any desired modulus.

SN54163, SN74163 Synchronous binary counters

SN54161, SN74161 Synchronous binary counters are similar; however, the CLEAR is asynchronous as shown for the SN54160, 74160 decade counters at left.

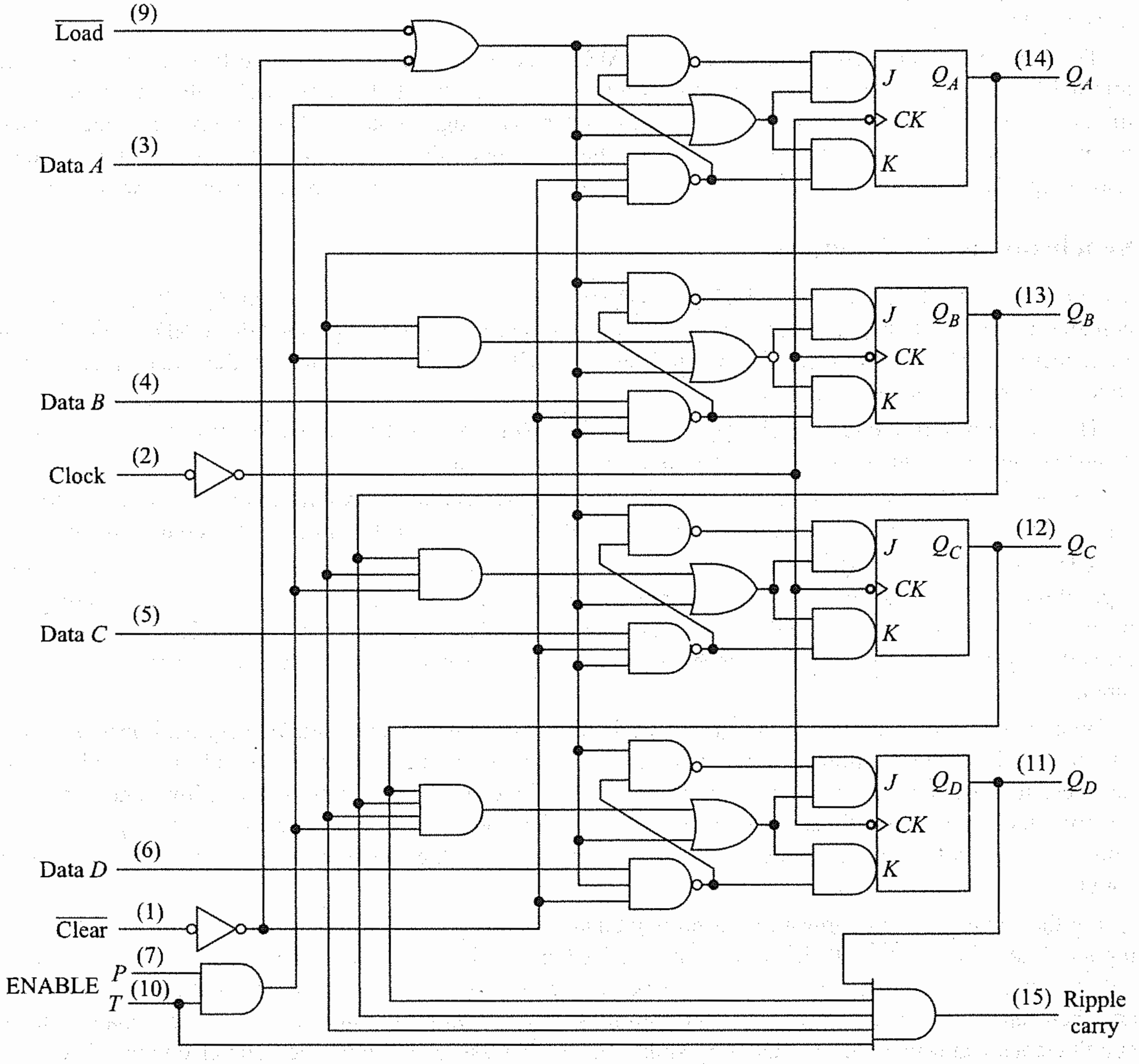
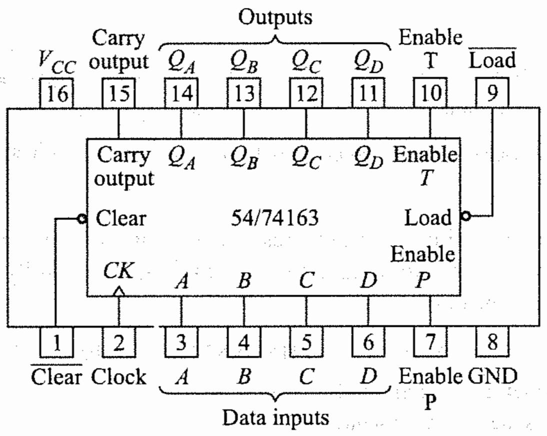


Fig. 10.25 54/74161 and 54/74163

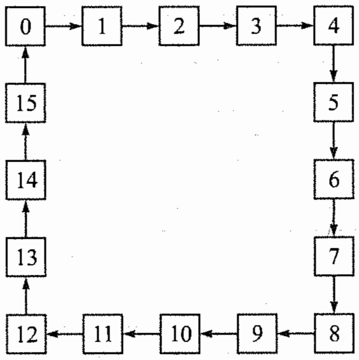
For instance, if a maximum count of 9 is desired, we connect the inputs of the NAND gate to decode count $9 = DCBA = 1001$. We then have a mod-10 counter, since the count sequence is from 0000 up to 1001. The NAND gate used to decode count 9 along with the modified state diagram are shown in Fig. 10.26b and c, respectively. Notice that it was necessary to use two inverters to obtain $Q_{\overline{B}}$ and $Q_{\overline{C}}$. The modified

J or *N* dual-in-line or W flat package (Top view)

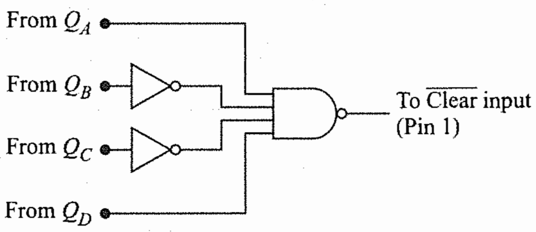


Positive logic: See description.

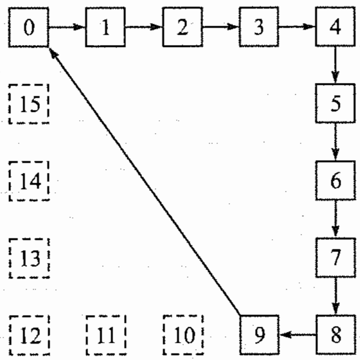
Fig. 10.25 **(Continued)**



(a) Mod-16 counter state diagram



(b) Gate to decode count 9 (1001)



(c) Modified state diagram for Mod-10 counter

Fig. 10.26

state diagram has solid boxes for states in the modified, mod-10 counter, and dashed boxes for omitted states.

Example 10.12 What are the NAND-gate inputs in Fig. 10.26b if this figure is to be used to construct a mod-12 counter?

Solution The counter must progress from 0000 up to 1011 (decimal 11); the NAND-gate inputs must then be Q_D, Q_C, Q_B, and Q_A.

A set of typical waveforms showing the clear, preset, count, and inhibit operations for a 54/74163 (and 54/74161) is given in Fig. 10.27. You should take time to study them carefully until you understand exactly how these four operations are controlled.

SN54161, SN54163, SN74161, SN74163 Synchronous binary counters

Typical clear, preset, count, and inhibit sequences

Illustrated below is the following sequence.

1. Clear outputs to zero.
2. Preset to binary twelve.
3. Count to thirteen, fourteen, fifteen, zero, one, and two.
4. Inhibit.

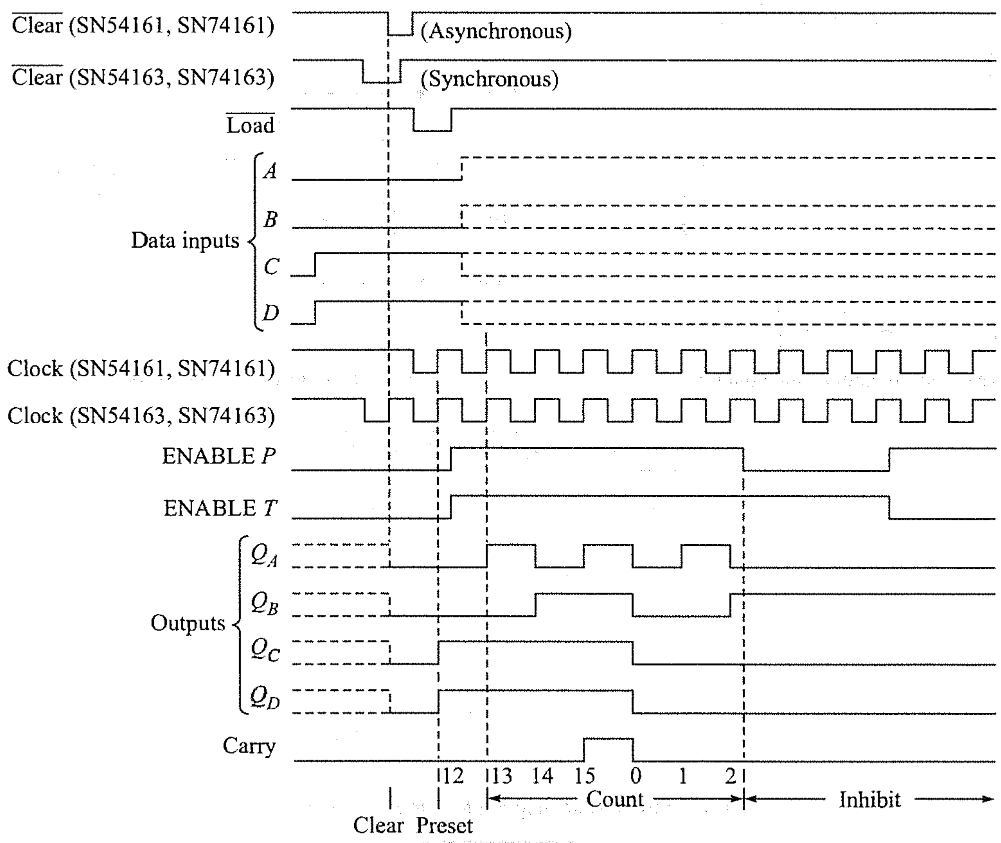
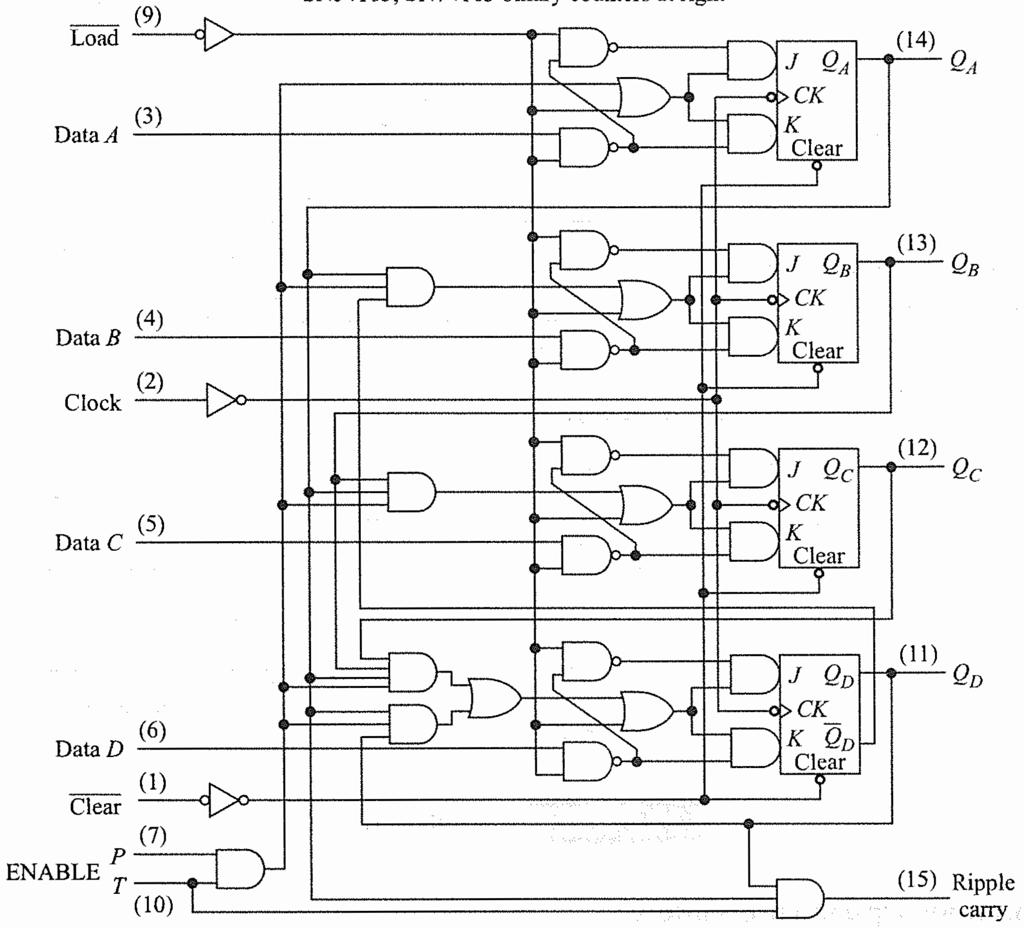


Fig. 10.27

The logic diagram and a typical set of waveforms for the 54/74160 and the 54/74162 are given in Fig. 10.28. (The pinout is identical for the previously given 54/74163.) These two counters have been modified internally and are decade counters. Other than that, the input, output, and control lines for these two counters are identical with the previously discussed 54/74163 and 54/74161. These counters advance one count with each positive clock transition, progressing from 0000 to 1001 and back to 0000. The state diagram for these two units would appear exactly as shown in Fig. 10.26c; this is the state diagiam for a mod-10 or decade counts.

SN54160, SN74160 Synchronous decade counters

SN54162, SN74162 Synchronous decade counters are similar; however, the clear is synchronous as shown for the SN54163, SN74163 binary counters at right



(a)

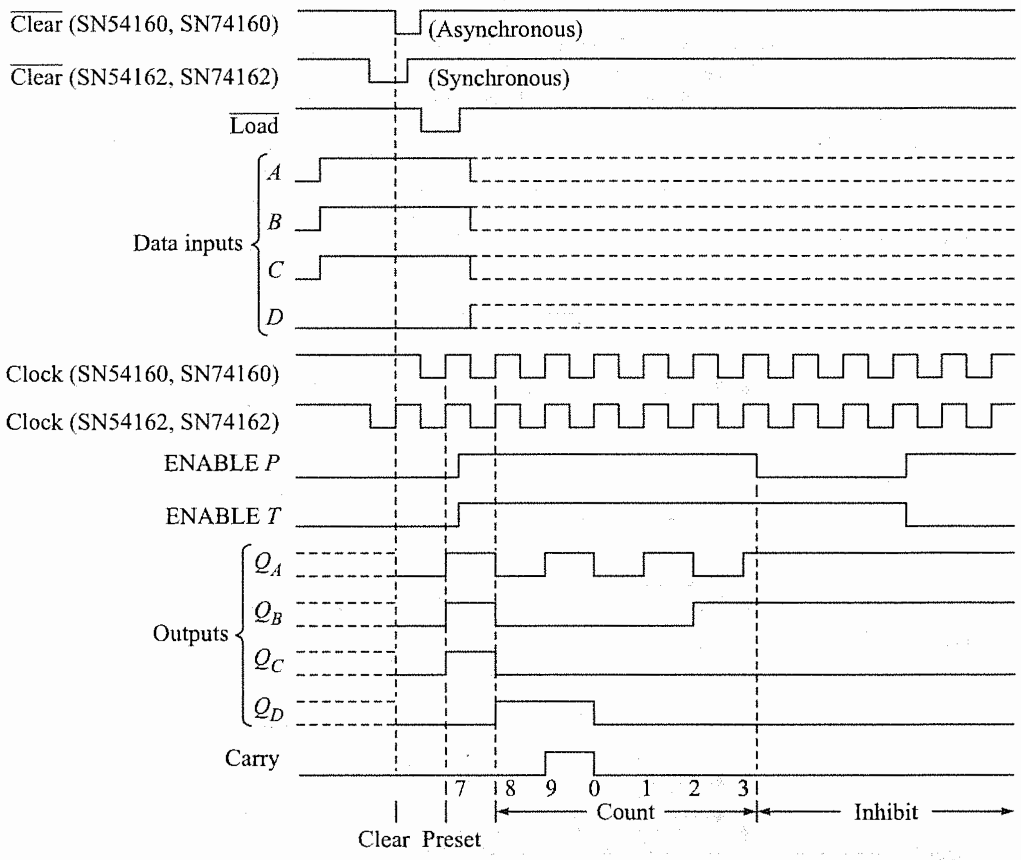
Fig. 10.28 54/74160 (continued on next page)

SN54160, SN54162, SN74160, SN74162 Synchronous binary counters

Typical clear, preset, count, and inhibit sequences

Illustrated below is the following sequence.

1. Clear outputs to zero.
2. Preset to BCD seven.
3. Count to eight, nine zero, one, two, and three.
4. Inhibit.



(b)

Fig. 10.28 (Continued)

Synchronous Up-Down Counters

The 54/74193 is a 4-bit synchronous up-down binary counter. It has a master reset input and can be reset to any desired count with the parallel load inputs. The logic symbol for this TTL MSI is shown in Fig. 10.29a. Pin $\overline{PL}$ is a control input for loading data into pins P_A, P_B, P_C, and P_D. When the device is used as a counter, these four pins are left open and $\overline{PL}$ must be held high. Pin MR is the master reset, and it is normally held low. (A high level on MR will reset all flip-flops.)

Outputs TC_U and TC_D are to be used to drive the following units, such as in a cascade arrangement. The clock inputs are CP_U and CP_D. Placing the clock on CP_U will cause the counter to count up, and placing the clock on CP_D will cause the counter to count down. Notice that the clock should be connected to either CP_U or CP_D, but not both, and the unused input should be held high. The outputs of the counter are Q_A, Q_B, Q_C and Q_D.

A state diagram is a simple drawing which shows the stable states of the counter, as well as how the counter progresses from one count to the next. The state diagram for the 54/74193 is shown in Fig. 10.29b. Each box represents a stable state, and the arrows indicate the count sequence for both count-up and count-down operations. This is a 4-bit counter, and clearly there are 16 stable states, numbered 0, 1, 2, ..., 15.

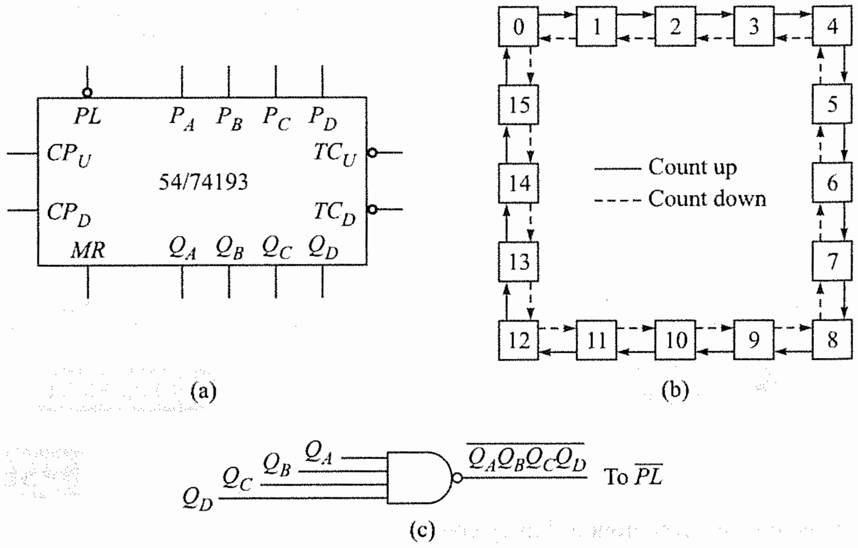


Fig. 10.29 **4-bit binary counter (presettable)**

The 54/74193 has a parallel-data-entry capability which permits the counter to be preset to the number present on the parallel-data-entry inputs (P_A, P_B, P_C, and P_D). Whenever the parallel load input ($\overline{PL}$) is low, the data present at these four inputs is shifted into the counter; that is, the counter is preset to the number held by $P_DP_CP_BP_A$.

Now, here is another technique for modifying the count. Simply use a NAND gate to detect any of the stable states, say, state 15 (1111), and use this gate output to take $\overline{PL}$ low. The only time $\overline{PL}$ will be low is when Q_D, Q_C, Q_B, and Q_A are all high, or state 15(1111). At this time, the counter will be preset to the data $P_DP_CP_BP_A$.

For example, suppose that $P_DP_CP_BP_A = 1001$ (the number 9). When the clock is applied, the counter will progress naturally to count 15(1111). At this time, $\overline{PL}$ will go low and the number 9 (1001) will be shifted into the counter. The counter will then progress through states 9, 10, 11, 12, 13, and 14, and at count 15 it will again be preset to 9.

The count sequence is easily shown by the state diagram in Fig. 10.30 on the next page. Notice that count 15 (1111) is no longer a stable state; it is the short time during which the counter is preset. The stable states in this example are 9, 10, 11, 12, 13, and 14. This is, then, a mod-6 counter. Notice that this technique is asynchronous since the preset action is not in synchronism with the clock. Therefore, you should be aware that counting spikes or glitches may be associated with the outputs of this presetting arrangement.

Example 10.13 Suppose that the counter just discussed is still preset to 1001 (the number 9) but the clock is applied to count down rather than count up. What are the counting states? What is the modulus?

Solution The counter will count down to 15, then preset back to 9, and repeat. The resulting state diagram is given in Fig. 10.31. The modulus is clearly 10.

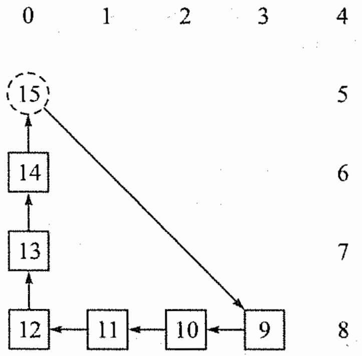


Fig. 10.30

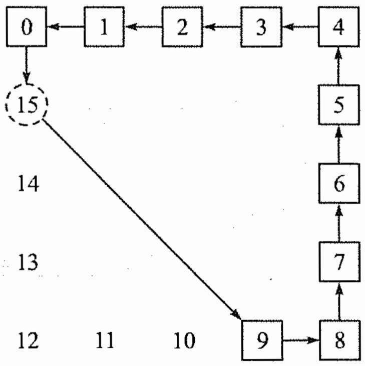


Fig. 10.31

SELF-TEST

12. Name two popular synchronous binary counters.
13. What is the difference between the 74161 binary counter and the 74191 binary counter?
14. What is the modulus of the 74160 counter?
15. Can a 74160 counter be used to count down?

10.7 COUNTER DESIGN AS A SYNTHESIS PROBLEM

Section 8.11 of Chapter 8 presents a systematic approach towards sequential logic circuit design using FSM concept. In this section, we consider counter as a state machine and discuss counter design steps through an example.

Let us try to design a modulo-6 counter, the counting states (memory values) of which are shown in state transition diagram of Fig. 10.32. We need three memory elements or flip-flops for this as with n flip-flop we can get at most 2^n number of different counting states.

Now with three flip-flop, 8 different states are possible but in our design states 110 and 111 are not used in the counting

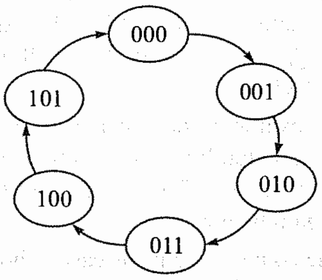


Fig. 10.32 State sequence of a modulo-6 counter

sequence. To start with we shall assume the counter is always initialized with one of the valid states and not 110 or 111. We decide to use three *JK* flip-flops labeled *A*, *B* and *C* as memory element for this design.

The next step to be taken is to form a state synthesis table as shown in Table 10.1. In this, the first column represents current state of the counter and second column, as shown in the next state of the counter state transition diagram. We fill up next three columns using excitation table of *JK* flip-flop given in Fig. 8.34 of Chapter 8. Excitation table gives inputs need to be present when clock triggers a certain $Q_n \to Q_{n+1}$ transition of the flip-flop. In the first row, we see both *C* and *B* make transition $0 \to 0$ and hence corresponding *JK* inputs should be 0× from excitation table. For flip-flop *A*, transition is $0 \to 1$ and input should be 1×. This is continued to fill up other five rows of input columns for three flip-flops.

Table 10.1 State Table for Design of Modulo-6 Counter Given in Fig. 10.32

C_n	B_n	A_n	C_{n+1}	B_{n+1}	A_{n+1}	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	1	0	×	0	×	1	×
0	0	1	0	1	0	0	×	1	×	×	1
0	1	0	0	1	1	0	×	×	0	1	×
0	1	1	1	0	0	1	×	×	1	×	1
1	0	0	1	0	1	×	0	0	×	1	×
1	0	1	0	0	1	×	1	0	×	×	1

Our next objective is to get logic equation for each flip-flop input as a function of present state of the counter. We use Karnaugh Map for this as shown in Fig. 10.33. Note that values corresponding to unused states 110 and 111 appear as don't care '×'. We have not shown Karnaugh Map for J_A and K_A as it is obvious from Table 10.1 that $J_A = K_A = 1$.

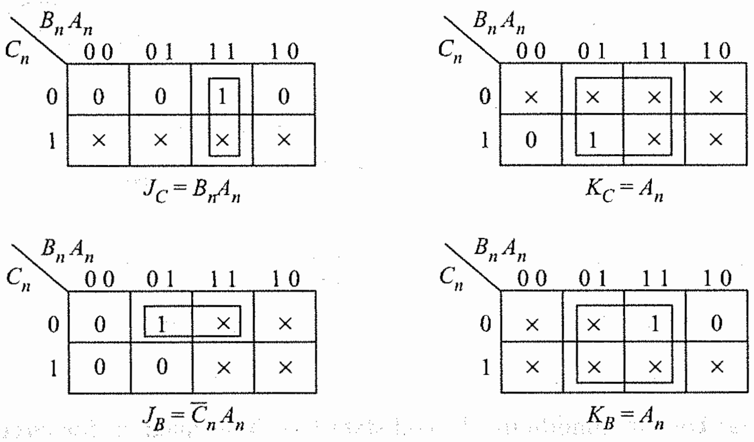


Fig. 10.33 Derivation of design equations from Karnaugh Map

The final step is to draw the circuit diagram from these design equations, which is shown in Fig. 10.34. The decoding output is obtained from a three input AND gate which goes high every time the counter goes to a valid state *CBA* = 000 and that occurs in every 6th clock cycle.

Note that the method we have explained is a general one and can be used to design counter of any modulo number and that can follow any given counting sequence. An *irregular counter* is the one which does not follow any regular binary sequence but has N number of distinct states and thus qualifies as a modulo-N counter. In Example 10.15, we present a modulo-4 irregular counter.

One question can be raised at this point for the above circuit. What happens if the circuit for any reason goes to one of the unused state? Does it come back to any of the valid counting state or in the worst case gets locked as shown in Fig. 10.35a? Initializing the designed circuit with 110 or 111 unused state we find that they get back to counting sequence as shown in Fig. 10.35b. However, a designer may not leave unused states to chances and want them to follow certain course if the circuit accidentally enters into one of them. Example 10.14 shows how to handle unused states in a counter design problem.

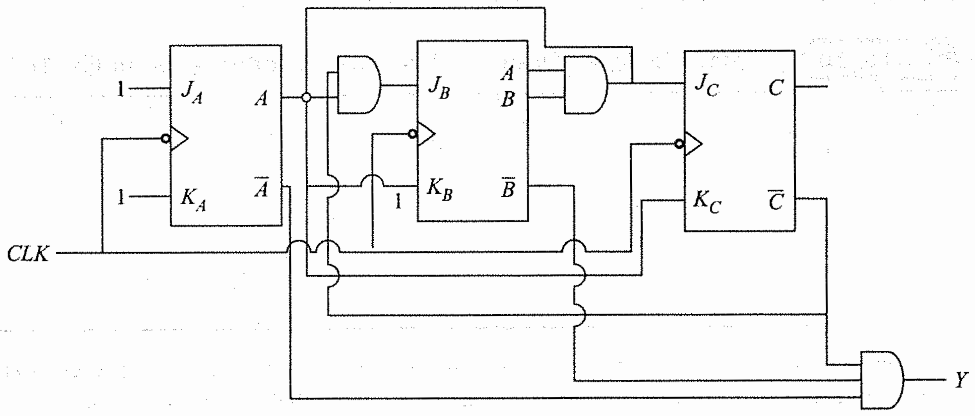


Fig. 10.34 **Circuit diagram of modulo-6 synchronous counter described in Fig. 10.32**

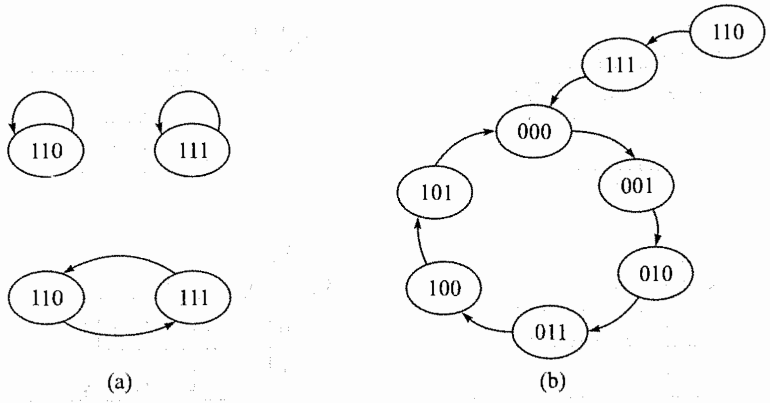


Fig. 10.35 **(a) Lock-in conditions, (b) Full state transition diagram for circuit in Fig. 10.34**

Example 10.14 Design a *self-correcting* modulo-6 counter as described in Fig. 10.32 in which all the unused state leads to state $CBA = 000$.

Solution For this we have to add two more rows as given next for two unused states in state Table 10.1.

C_n	B_n	A_n	C_{n+1}	B_{n+1}	A_{n+1}	J_C	K_C	J_B	K_B	J_C	K_C
1	1	0	0	0	0	×	1	×	1	0	×
1	1	1	0	0	0	×	1	×	1	×	1

Accordingly, Karnaugh Map giving design equations changes to as given in Fig. 10.36.

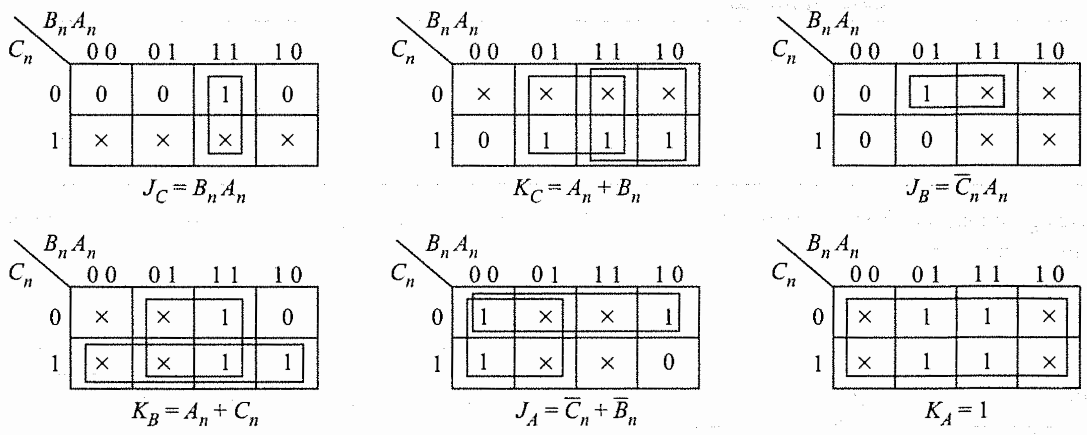

$J_C = B_n A_n$
$K_C = A_n + B_n$
$J_B = \overline{C}_n A_n$
$K_B = A_n + C_n$
$J_A = \overline{C}_n + \overline{B}_n$
$K_A = 1$

Fig. 10.36 Design equations for Example 10.14

Note the difference between Fig. 10.33 and 10.36. Unused states 110 and 111 can no longer be considered as don't care. This type of design is called self-correcting as the circuit comes out on its own from an invalid state to a valid counting state sequence. The final circuit diagram from design equations are shown in Fig. 10.37.

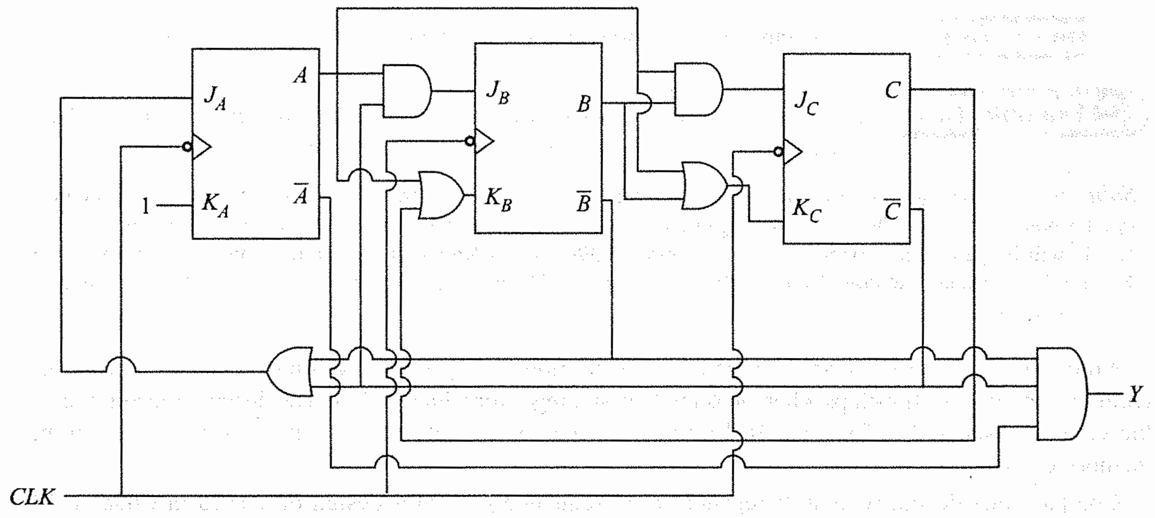


Fig. 10.37 Circuit diagram for Example 10.14

Example 10.15 Design a modulo-4 irregular counter with following counting sequence using D flip-flop.

$$\begin{tikzcd} 00 \arrow[r] & 10 \arrow[r] & 11 \arrow[r] & 01 \arrow[lll, bend left] \end{tikzcd}$$

Solution Using state excitation table of D flip-flop (Fig. 8.34), the state table can be formed as shown in Table 10.2.

Table 10.2 State Table for Design of Irregular Counter

B_n	A_n	B_{n+1}	A_{n+1}	D_B	D_A
0	0	1	0	1	0
0	1	0	0	0	0
1	0	1	1	1	1
1	1	0	1	0	1

Design equations from Karnaugh Map can be derived as shown in Fig. 10.38(a), and corresponding logic circuit is shown in Fig. 10.38(b).

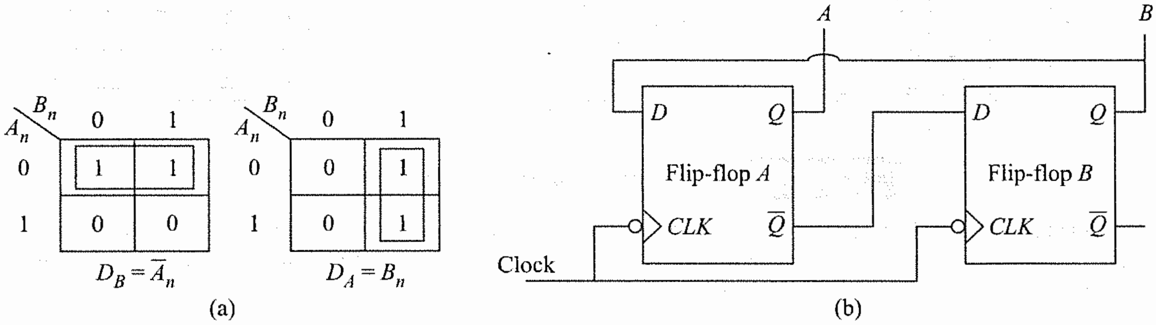


Fig. 10.38 (a) Deriving design equations for Example 10.15, (b) Circuit diagram

Example 10.16 Show how a modulo-4 counter designed with two flip-flops can generate a repetitive sequence of binary word '1101' with minimum number of memory elements?

Solution Let the counting sequence of two flip-flops B and A be $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$ ..., i.e. a modulo-4 synchronous up counter. The corresponding output is $1 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 1$.... As shown in Fig. 10.39(a) the sequence '1101' will be generated repetitively by Y. Figure 10.39(b) gives Karnaugh Map representation of Y and we get $Y = A + B'$. A standard modulo-4 up counter and an 2-input OR gate connected as shown in Fig. 10.39(c) generates the given sequence.

Note that for N-bit sequence generator we need modulo-N counter. Modulo-N synchronous counter requires m number of flip-flops where m is the lowest integer for which $2^m \geq N$. The design procedure remains the same as discussed in Example 10.16. Output Y now is a function of m state variables representing m memory elements.

Compare this design with shift register based sequence generator design discussed in Chapter 9 that requires N number of memory elements for N-bit sequence generator. Though shift register based design does not require any combinatorial circuit to generate output logic the overall hardware cost is more and it is more pronounced for large N.

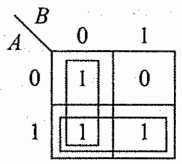
A similar design for sequence detector circuit with minimum number of flip-flops is discussed in Chapter 11.

SELF-TEST

16. What is lock-out of a counter?
17. For 48-bit sequence generator what is the minimum number of memory elements required?

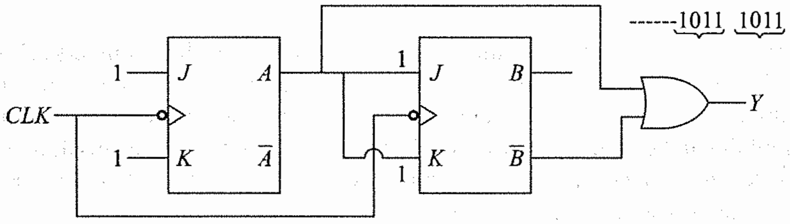
A	B	Y
0	0	1
0	1	1
1	0	0
1	1	1

(a)



$Y = A + \overline{B}$

(b)



(c)

Fig. 10.39 Sequence generator circuit using synchronous counter, (a) State Table, (b) Output equation, (c) Circuit diagram

10.8 A DIGITAL CLOCK

A very interesting application of counters and decoding arises in the design of a digital clock. Suppose that we want to construct an ordinary clock which will display hours, minutes, and seconds. The power supply for this system is the usual 60-Hz 120-Vac commercial power. Since the 60-Hz frequency of most power systems is very closely controlled, it is possible to use this signal as the basic clock frequency for our system. Note that in several countries commercial power supply is 50-Hz and not 60-Hz. There one can use standard variable frequency signal generator, set at 60-Hz, as input.

In order to obtain pulses occurring at a rate of one each second, it is necessary to divide the 60-Hz power source by 60. If the resulting 1-Hz waveform is again divided by 60, a one-per-minute waveform is the result. Dividing this signal by 60 then provides a one-per-hour waveform. This, then, is the basic idea to be used in forming a digital clock.

A block diagram showing the functions to be performed is given in Fig. 10.40. The first divide-by-60 counter simply divides the 60-Hz power signal down to a 1-Hz square wave. The second divide-by-60 counter changes state once each second and has 60 discrete states. It can, therefore, be decoded to provide signals to display seconds. This counter is then referred to as the seconds counter.

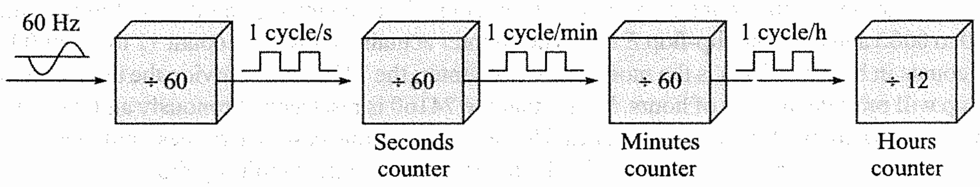


Fig. 10.40 Block diagram of digital clock

The third divide-by-60 counter changes state once each minute and has 60 discrete states. It can thus be decoded to provide the necessary signals to display minutes. This counter is then the minutes counter.

The last counter changes state once each 60 minutes (once each hour). Thus, if it is a divide-by-12 counter, it will have 12 states that can be decoded to provide signals to display the correct hour. This, then, is the hours counter.

As you know, there are a number of ways to implement a counter. What is desired here is to design the counters in such a way as to minimize the hardware required. The first counter must divide by 60, and it need not be decoded. Therefore, it should be constructed in the easiest manner with the minimum number of flip-flops.

For instance, the divide-by-60 counter could be implemented by cascading counters ($12 \times 5 = 60$, or $10 \times 6 = 60$, etc.). The TTL MSI 7490 decade counter can be used as a divide-by-10 counter, and the TTL MSI 7492 can be used as a divide-by-6 counter. Cascading these two will provide a divide-by-60 counter as shown in Fig. 10.41. The amplifier at the input provides a 60-Hz square wave of the proper amplitude to drive the 7490. The 7492 is connected as a divide-by-12 counter, but only outputs Q_A, Q_B, and Q_C are used. In this fashion, the 7492 operates essentially as a divide-by-6 counter.

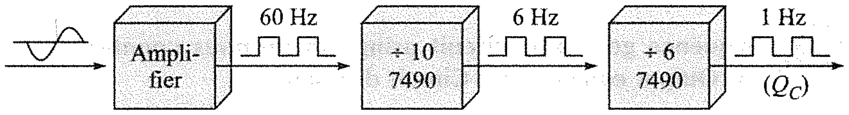


Fig. 10.41 Divide-by-60 counter

The seconds counter in the system also divides by 60 and could be implemented in the same way. However, the seconds counter must be decoded. We are interested in decoding this counter to represent each of the 60 s in 1 min. This can most easily be accomplished by constructing a mod-10 counter in series with a mod-6 counter for the divide-by-60 counter. The mod-10 counter can then be decoded to represent the units digit of seconds, and the mod-6 counter can be decoded to represent the tens digits of seconds.

Since both the 7490 and the 7492 count in straight 8421 binary, a 7447 decoder-driver can be used with each to drive two 7-segment indicators, as shown in Fig. 10.42. Notice that the 7492 is connected as a divide-by-12 counter, but only outputs Q_A, Q_B, and Q_C are used to drive the 7447 decoder-driver.

The minutes counter is exactly the same as the seconds counter, except that it is driven by the one-per-minute square wave from the output of the seconds counter, and its output is a one-per-hour square wave, as shown in Fig. 10.42.

The divide-by-12 hours counter must be decoded into 12 states to display hours. This can be accomplished by connecting a mod-10 (54/74160) decade counter in series with a single flip-flop E as shown in Fig. 10.43. This forms a divide-by-20 ($10 \times 2 = 20$) counter. Feedback is then used to form a mod-12 counter.

The hours counter must count through states 00, 01, 02, ... , 11, and then back to 00. The NAND gate in Fig. 10.43 will go low as the counter progresses from count 11 to count 12, and this will immediately clear the 74160 to 0000 and reset the flip-flop E to 0. The counter actually skips from count 11 to count 00 omitting the eight counts in between. This is the mod-12 hours counter; the 74160 will provide the units of hours while the flip-flop will provide the tens of hours. Notice that the 74160 is reset asynchronously and there might then be glitches at the outputs of the decoding gates. However, this is one case where these glitches will have no effect, since they are too narrow to cause a visible indication on the light emitting diodes (LEDs).

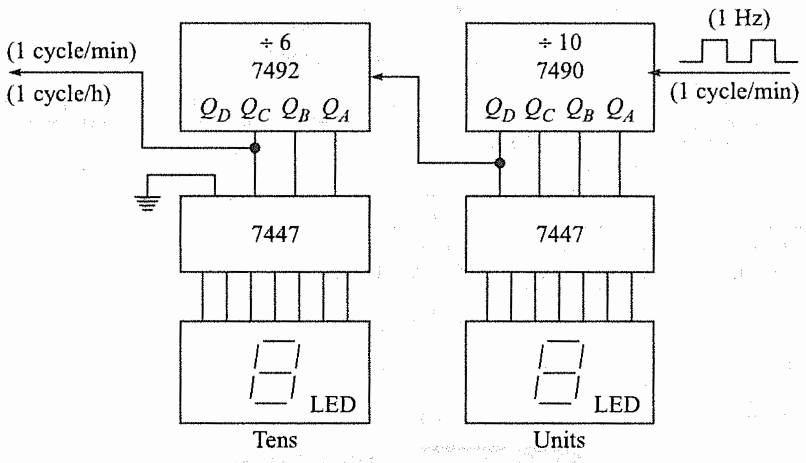


Fig. 10.42 **A 10 × 6 mod-60 counter with units and tens decoding**

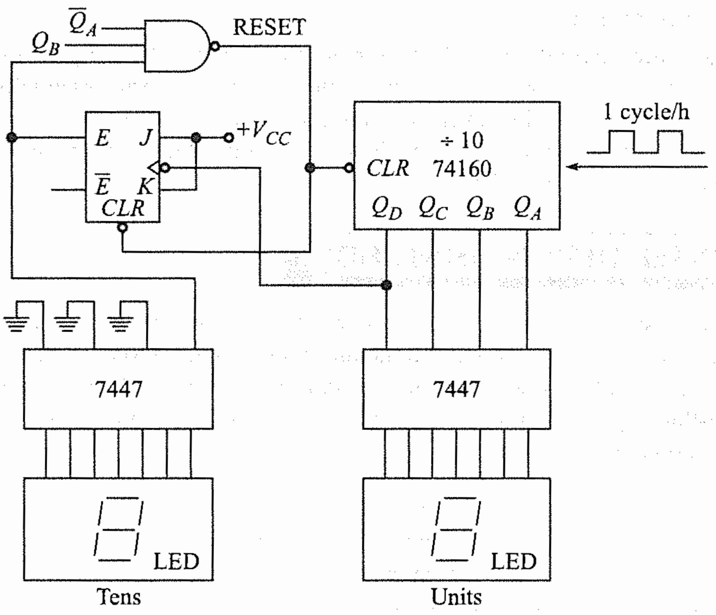


Fig. 10.43 **Mod-12 hours counter**

Finally, some means must be found to set the clock because the flip-flops will assume random states when the power is turned off and then turned back on again. Setting the clock can be quite easily accomplished by means of the SET push-buttons shown in Fig. 10.44. Depressing the SET HOURS button causes the hours counter to advance at a one-count-per-second rate, and thus this counter can be set to the desired hour. The minutes counter can be similarly set by depression of the SET MINUTES button.

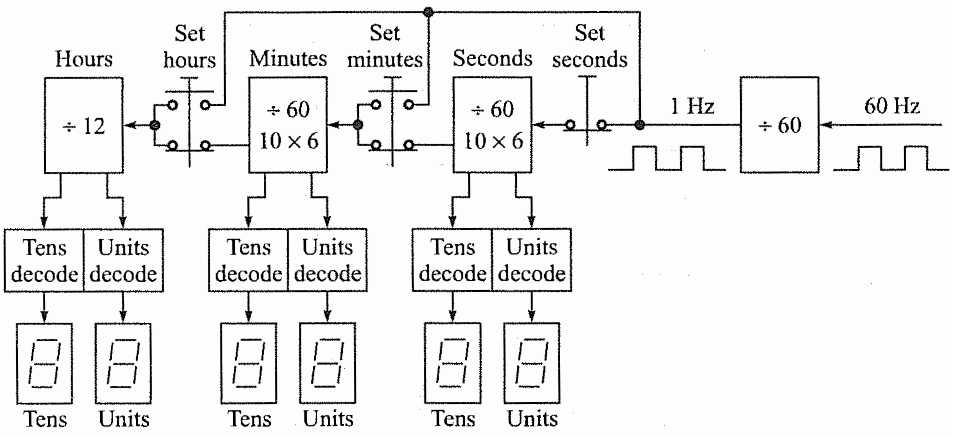


Fig. 10.44 **Digital clock**

Depression of the SET SECONDS button removes the signal from the seconds counter, and the clock can thus be brought into synchronization.

By means of large-scale integration (LSI), it is possible to construct a digital clock entirely on one semiconductor chip. Such units are commercially available, and they perform essentially the function shown in the logic diagram in Fig. 10.43 (the seven-segment indicators are, of course, separate). The National Semiconductor 5318 is one such commercially available LSI digital clock. It is available in a 24-pin dual in-line (DIP) package measuring 0.54 × 1.25 in.

10.9 COUNTER DESIGN USING HDL

Counter design in HDL is straight forward if one uses arithmetic operator + and – that corresponds to binary addition and subtraction respectively. We show a modulo-8 up counter design in the example given in first column. It is left to the compiler to decide which flip-flop is to be used. If one wants to ensure use of a particular type of flip-flop say, *JK* then the code should be written in a manner shown in second column for modulo-3 up counter shown in Fig. 10.16a.

```
module UC(Clock, Reset,Q);
input Clock, Reset;
output  [2:0] Q;
  //modulo 8 requires 3 flip-flop
reg [2:0] Q;
always @ (negedge Clock
    or negedge Reset)
  if (~Reset) Q=3'b0;
  else Q = Q+1;
endmodule
```

```
 module UCJK(A,B,Clock,Reset);
 input Clock,Reset;
 output  A,B; //modulo-3 requires 2 flip-flop
wire JA,JB,KA,KB;
 assign JA=~B;

assign KA=1'b1;
assign JB=A;
assign KB=1'b1;
JKFF JK1(A,JA,KA,Clock,Reset); //instantiates JKFF
JKFF JK2(B,JB,KB,Clock,Reset); //instantiates JKFF
endmodule
```

```
module JKFF(Q,J,K,Clock,Reset);
input J,K,Clock,Reset;
 output Q;
 reg Q;
always @ (negedge Clock or negedge Reset)
   if(~Reset) Q=1'b0;
   else Q <=  (J&~Q) | (~K&Q);
endmodule
```

Example 10.17 Design a modulo-8 up down counter which counts in upward direction if input MODE = 0, else counts in downward direction. It should also have a parallel load facility. When *PL* = 1, a 3-bit number *D* is asynchronously loaded to the counter. The counter counts at the negative edge of CLOCK and its output is represented by *Q*.

Solution The Verilog HDL code for the problem is given below. We have used a new keyword integer to hold a value temporarily. This helps us in writing both up and down count in one single statement that responds to clock within **always** block.

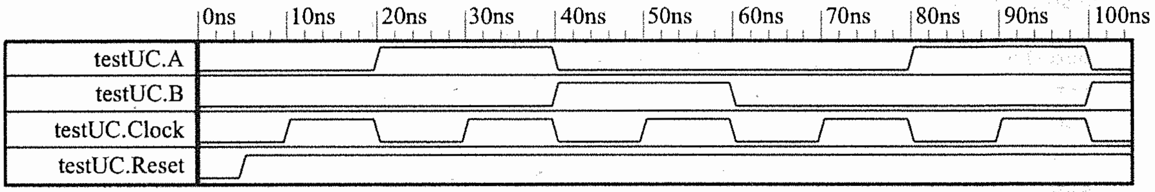
```
module UDCPL(CLOCK,PL,MODE,D,Q);    //Up Down Counter
input CLOCK,PL,MODE;                  //with parallel load
input [2:0] D;
output  [2:0] Q; //modulo 8 requires 3 flip-flop
reg [2:0] Q;
integer updown; //updown will be +1 or -1 depending on MODE
always @ (negedge CLOCK)
   begin
     if (MODE) updown=-1; //If MODE=1, counts downward
     else updown=1;
     if (PL) Q=D;  //If PL=1, parallel loading takes place
     else Q=Q+updown;  //Last else statement responds to clock
   end
Endmodule
```

Example 10.18 The code in the first column when executed with modulo-3 *JK* counter, UCJK described in this section generates monitor output as shown in column 2 and timing diagram as shown below. Show how the test bench verifies Verilog code UCJK is that of a modulo-3 counter.

```
module testUC();
 reg Clock,Reset;                                          0 Clock= 0, A=0 B=0
 wire A,B;                                                10 Clock= 1, A=0 B=0
 UCJK M3(A,B,Clock,Reset); /*instantiates                 20 Clock= 0, A=1 B=0
   modulo-3 JK FF  counter*/                              30 Clock= 1, A=1 B=0
 initial                                                  40 Clock= 0, A=0 B=1
      begin                                               50 Clock= 1, A=0 B=1
```

```
    Clock = 0; // initial value of clock
    Reset = 0; //initial value of Reset=0
    #5 Reset =1; //Reset becomes 1 after 5 second
    #100 $finish; // Terminate simulation after 100ns
 end
 always
     begin
    #10 Clock = ~Clock;   //Clock toggles every 10 ns
     end
 initial
    begin
    $monitor($time, " Clock= %b,A= %b  B=%b\n",
      Clock,A,B);
    end
endmodule
```

```
 60 Clock= 0, A=0 B=0
 70 Clock= 1, A=0 B=0
 80 Clock= 0, A=1 B=0
 90 Clock= 1, A=1 B=0
100 Clock= 0, A=0 B=1
```



Solution The test bench, given in the code above runs the simulation for 5 + 100 = 105ns duration. At every 10 ns clock toggles giving a 10 + 10 = 20ns clock cycle time. So we have 5 negative edges of the clock (1 to 0 transition) from start at 20, 40, 60, 80 and 100ns. If we look at the timing diagram and monitor output we see *JK* flip flop output changes value at negative edges as *BA* = 00 (initially reset by 'Reset'), 01, 10, 00, 01, 10 etc. These show that three counting states 00, 01, 10 get repeated. Hence, the code of module UCJK behaves like a modulo-3 counter.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Design a self correcting modulo-3 down counter.

Solution We need 2 flip-flops, say *B* and *A* for this purpose which has 4 states. Let the down counter count like 10 → 01 → 00 → 10... and undesired state 11 corrects itself to 10. The excitation table of Fig. 8.35 is used for the design purpose.

In Method-1, we use *SR* flip-flop for design purpose. Figure 10.45a shows the state table and in the second column, necessary inputs for the two *SR* flip-flops are given. Figure 10.45b shows the use of Karnaugh Map to get the design equations.

In Method-2, we use *JK* flip-flop for design purpose. The fourth column shows necessary inputs for the two *JK* flip-flops. Figure 10.45c shows the use of Karnaugh Map to get the design equations.

In Method-3, we use *D* flip-flop for design purpose. The third column shows necessary inputs for the two *D* flip-flops. Fig. 10.45d shows the use of Karnaugh Map to get the design equations.

In Method-4, we use T flip-flop for design purpose. The last column shows necessary inputs for the two D flip-flops. Figure 10.45e shows the use of Karnaugh Map to get the design equations.

Present State B_nA_n	Next State $B_{n+1}A_{n+1}$	S_BR_B S_AR_A	D_B D_A	J_BK_B J_AK_A	T_B T_A
0 0	1 0	1 0 0 X	1 0	1 X 0 X	1 0
0 1	0 0	0 X 0 1	0 0	0 X X 1	0 1
1 0	0 1	0 1 1 0	0 1	X 1 1 X	1 1
1 1	1 0	X 0 0 1	1 0	X 0 X 1	0 1

(a)

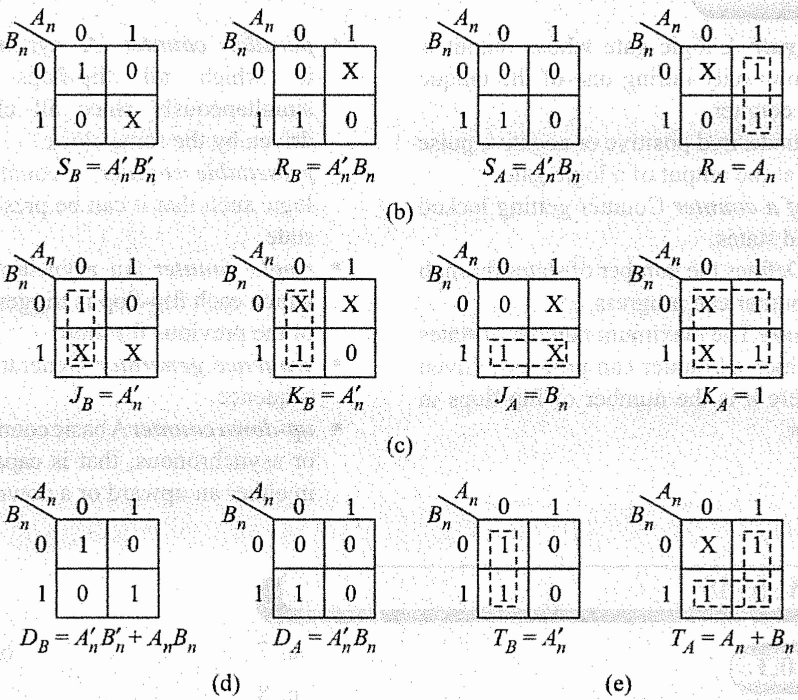


Fig. 10.45 (a) State table for the self correcting modulo-3 counter and required inputs, (b) Design with SR flip-flops, (c) Design with *JK* flip-flops, (d) Design with *D* flip-flops, (e) Design with *T* flip-flops

SUMMARY

A counter has a natural count of 2^n, where n is the number of flip-flops in the counter. Counters of any modulus can be constructed by incorporating logic which causes certain states to be skipped over or omitted. One technique for skipping counts is to steer the clock pulses to certain flip-flops at the proper time—this is called steering logic. A second technique is to precondition the logic inputs to each flip-flop in order to omit certain states. This is called look-ahead logic.

Logic can be included such that the counter can operate in either a count-up or count-down mode. Furthermore, logic gates can be designed to uniquely decode each state of a counter.

Higher-modulus counters can be easily constructed by using combinations of lower-modulus counters. Such configurations represent a compromise between speed and hardware count.

The digital clock is an interesting application that illustrates some of the methods employing counters and decoders.

GLOSSARY

- ***decoding gate*** A logic gate whose output is high (or low) only during one of the unique states of a counter.
- ***glitch*** An undesired positive or negative pulse appearing at the output of a logic gate.
- ***lock out of a counter*** Counter getting locked into unused states.
- ***modulus*** Defines the number of states through which a counter can progress.
- ***natural count*** The maximum number of states through which a counter can progress. Given by 2^n, where n is the number of flip-flops in the counter.
- ***parallel counter*** A synchronous counter in which all flip-flops change states simultaneously since all clock inputs are driven by the same clock.
- ***presettable counter*** A counter incorporating logic such that it can be preset to any desired state.
- ***ripple counter*** An asynchronous counter in which each flip-flop is triggered by the output of the previous flip-flop.
- ***sequence generator*** Generates a binary data sequence.
- ***up-down counter*** A basic counter, synchronous or asynchronous, that is capable of counting in either an upward or a downward direction.

PROBLEMS

Section 10.1

10.1 Draw the logic diagram, truth table, and waveforms for a two-flip-flop ripple counter similar to that in Fig. 10.1.

10.2 Draw the logic diagram, truth table, and waveforms for a three-flip-flop ripple counter that uses *JK* flip-flops sensitive to a clock PT.

10.3 What is the clock frequency if the period of *B* in Fig. 10.1 is 1000 ns?

10.4 Determine the number of possible states in a counter composed of the following number of flip-flops:

a. 7 b. 10

c. 8

10.5 See if you can draw the waveforms for a 10-flip-flop ripple counter. What difficulties do you encounter?

10.6 What is the largest decimal number that can be stored in each counter in Prob. 10.4?

10.7 Draw the waveforms at Q_B, Q_C, and Q_D for a 7493A, assuming that a 1-MHz clock is applied at input *B*.

10.8 Draw the logic diagram, truth table, and waveforms for a two-flip-flop ripple counter operating in the count-down mode.

Section 10.2

10.9 Draw the gates necessary to decode the 16 states of a 7493A operating as in Fig. 10.3.

10.10 Assume that the clock for the ripple counter in Fig. 10.1 is a 1-MHz square wave and each flip-flop has a delay time of 0.25 μs. Carefully draw the waveforms for the clock and each flip-flop and the output decoded signals. Do you see any sources of difficulty?

10.11 Use the waveforms in Fig. 10.9 and study the remaining seven decoding gates in Fig. 10.7. Show whether glitches will appear by drawing the decoded waveform for each gate.

Section 10.3

10.12 Draw the logic diagram, truth table, and waveforms for the synchronous counter in Fig. 10.13 in the count-up mode.

10.13 Repeat Prob. 10.12, but in the count-down mode.

10.14 Write a Boolean expression for the AND gate connected to the upper leg of the OR gate that drives the clock input to flip-flop Q_D in a 74193.

10.15 Draw a complete set of waveforms for the 74191 in Fig. 10.15 operating in the count-up mode.

10.16 Repeat Prob. 10.15, but operating in the count-down mode.

Section 10.4

10.17 Determine the number of flip-flops that would be required to build the following counters:

a. Mod-6 b. Mod-11

c. Mod-15 d. Mod-19

e. Mod-31

10.18 Draw decoding gates and all waveforms for the mod-3 counter in Fig. 10.16.

10.19 Draw decoding gates and all waveforms for the mod-6 counter in Fig. 10.17.

10.20 Draw decoding gates and all waveforms for the counter in Fig. 10.18.

10.21 Draw the logic diagram, truth table, and waveforms for a mod-9 counter using two mod-3 counters connected in series.

Section 10.5

10.22 Draw decoding gates and all waveforms for the decade counter in Fig. 10.21.

10.23 Draw decoding gates and all waveforms for the counter in Fig. 10.22.

10.24 Draw waveforms for Q_B, Q_C, and Q_D, assuming that the clock is applied to input B of a 7490A.

10.25 Show how an AND gate might be used in Fig. 10.24 to count an unknown number of pulses that occur during a known time interval. This is the basic idea used in a frequency counter.

Section 10.6

10.26 Draw the logic block for a 74163 and show how to construct a mod-13 counter. Use the same technique as in Fig. 10.26. Draw the state diagram.

10.27 Repeat Prob. 10.26 for a mod-11 counter and then a mod-7 counter.

10.28 Draw the waveforms expected in Prob. 10.26.

10.29 Draw the logic block for a 74162 and show how to construct a mod-7 counter. Use the same technique as in Fig. 10.26. Draw the state diagram.

10.30 Use a 74193 presettable counter to implement a mod-8 counter. List the omitted states .and normal count sequence; draw a complete logic diagram. Draw the set of waveforms you would expect, showing the clock and the four outputs. Remember that the output transitions occur on positive clock transitions.

Section 10.7

10.31 Design a modulo-3 counter using D flip-flop that counts as $01 \rightarrow 10 \rightarrow 11$. The unused state 00 goes to 01 at next clock trigger.

10.32 Design a modulo-5 counter using D flip-flop the unused states of which go to one of the valid counting state at next clock trigger.

10.33 Design a circuit using *JK* flip-flop that behaves both as a modulo-5 and modulo-3 counter depending on how it I initialized.

10.34 Design a modulo-8 counter (a) using *SR* flip-flop and (b) using *T* flip-flop.

10.35 Design a sequence generator with minimum number of flip-flops that generates sequence '110001' repetitively.

10.36 Design a sequence generator with minimum number of flip-flops that generates sequence '10110001' repetitively.

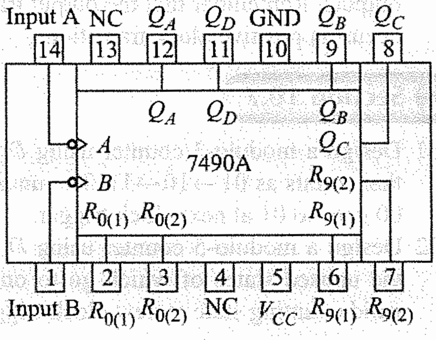
LABORATORY EXPERIMENT

AIM: The aim of this experiment is to study counters and design a modulo-N counter.

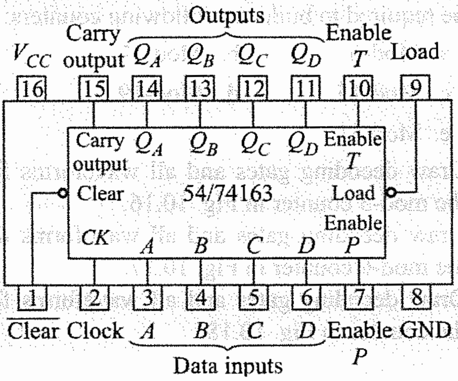
Theory: A counter counts events happening in certain form at its input. It consists of a bank of flip-flops and also may have a combinatorial logic circuit. In ripple counter, output of one flip-flop triggers another flip-flop. In synchronous counter, all the flip-flops are triggered simultaneously by a common clock. A modulo-*N* counter generates an output at every *n* pulse occurring at its input which is usually a clock signal. With *m* number of flip-flops, a maximum of 2^m modulo number can be achieved. The CLEAR input clears a counter which can be used to have lower modulo numbers from originally designed higher modulo number counters. The LOAD input, if available, allows parallel loading of a set of data from where counting sequence can begin. This can also be used to get a lower modulo number or some specific counting states.

Apparatus: 5V DC Power supply, Multimeter, Bread Board, Clock Generator, and Oscilloscope

Work element: IC 7490 has two separate counters, a modulo-2 and a synchronous modulo-5 counter which can work independently with two different clocks being connected to input *A* and input *B*. They can be combined to work as modulo-10 (coming from 5 × 2) counter if output of one is used as clock input of other. The pair R_0 performs NAND operation and then clears (active low) the modulo-2 counter. The pair R_9 functions similarly but for modulo-5 counter. Verify 7490 truth table for individual counters and the combination. Use *R* inputs to get modulo numbers different from 2, 5 and 10. IC 74163 is modulo-16 counter with synchronous clear and data input load facility. Verify the truth table of 74163. Understand the function of carry output. Show in how many different ways 74163 can be connected to get a decade (modulo-10) counter. Use two 74163 to obtain a modulo 50 and then a modulo 100 counter.



Input A NC Q_A Q_D GND Q_B Q_C
14 13 12 11 10 9 8
Q_A Q_D Q_B Q_C
A
B
7490A
$R_{9(2)}$
$R_{0(1)}$ $R_{0(2)}$ $R_{9(1)}$
1 2 3 4 5 6 7
Input B $R_{0(1)}$ $R_{0(2)}$ NC V_{CC} $R_{9(1)}$ $R_{9(2)}$



Outputs
Carry output
Enable T
V_{CC} Q_A Q_B Q_C Q_D Load
16 15 14 13 12 11 10 9
Carry output Q_A Q_B Q_C Q_D Enable T
Clear 54/74163 Load
Enable P
CK A B C D P
1 2 3 4 5 6 7 8
Clear Clock A B C D Enable P GND
Data inputs

Answers to Self-tests

1. 63
2. Ten
3. State 3 = $\overline{C}BA$
4. The primary cause of such glitches is flip-flop propagation time; one way to eliminate them is to use the clock as a "strobe."
5. In a parallel counter, all flip-flops change state in synchronism with the clock.
6. The glitches are eliminated because all gate inputs are synchronized—that is, they are all delayed from the clock by the same amount.
7. PTs
8. Four
9. 8. 7, 6, 5, 4, 3, or 2. However, mod-4 and mod-3 require only two flip-flops, and mod-2 is a single flip-flop.
10. A decade counter has 10 states—a mod-10 counter.
11. The decade counter in Fig. 10.21 has symmetrical output at *D*, but does not count in straight binary. The decade counter in Fig. 10.22 does not have symmetrical output at *D* and does count in straight binary.
12. 74161, 74163, 74191, 74193
13. The 74191 can count up or down.
14. The 74160 is a mod-10 counter.
15. The 74160 can only count up. (The 74190 can count down.)
16. Lock-out of a counter occurs when the counter remains locked into unused states and does not function properly.
17. Six.

Design of Synchronous and Asynchronous Sequential Circuits

11

OBJECTIVES

- **State machine design using Moore model and Mealy model**
- **State transition diagram and preparation of state synthesis table**
- **Derivation of design equation from state synthesis table using Karnaugh map**
- **Circuit implementation: flip-flop based approach and ROM based approach**
- **Use of Algorithm State Machine chart**
- **State reduction techniques**
- **Analysis of asynchronous sequential circuit**
- **Problems specific to asynchronous sequential circuit**
- **Design issues related to asynchronous sequential circuit**

Design problem normally starts with a word description of input output relation and ends with a circuit diagram having sequential and combinatorial logic elements. The word description is first converted to a state transition diagram or Algorithmic State Machine (ASM) chart followed by preparation of state synthesis table. For flip-flop based implementation, excitation tables are used to generate design equations through Karnaugh Map. The final circuit diagram is developed from these design equations. In Read Only Memory (ROM) based implementation, excitation tables are not required however; flip-flops are used as delay elements. In this chapter, we show how these techniques can be used in sequential circuit design.

There are two different approaches of state machine design called Moore model and Mealy model. In Moore model circuit outputs, also called primary outputs are generated solely from secondary outputs or memory values. In Mealy model circuit inputs, also known as primary inputs combine with memory elements to generate circuit output. Both the methods are discussed in detail in this chapter.

In general, sequential logic circuit design refers to *synchronous* clock-triggered circuit because of its design and implementation advantages. But there is increasing attention to *asynchronous* sequential logic circuit, as its response is not limited by the clock frequency. But there are too many operational constraints that makes design of asynchronous circuit very complex. Except for time-critical applications synchronous circuit always remains a preferred choice for sequential logic design.

We divide this chapter in two parts. Part A presents a systematic approach towards synchronous system design while Part B is devoted to asynchronous circuit.

PART A : DESIGN OF SYNCHRONOUS SEQUENTIAL CIRCUIT

11.1 MODEL SELECTION

There are two distinct models by which a synchronous sequential logic circuit can be designed. In *Moore model* (Fig. 11.1a) the output depends only on present state and not on input. In *Mealy model* (Fig. 11.1b), the output is derived from present state as well as input. The option to include input in output generation logic gives certain advantage to Mealy model. Usually it requires less number of states and thereby less hardware to solve any problem. Also, the output is generated one clock cycle earlier. However, there is one important disadvantage associated with such circuit. The input transients, glitches etc. (if any) are directly conveyed to the output. Also if we want output transitions to be synchronized while input can change any time Mealy model is not preferred. In Moore model, the output remains stable over entire clock period and changes only when there occurs a state change at clock trigger based on input available at that time.

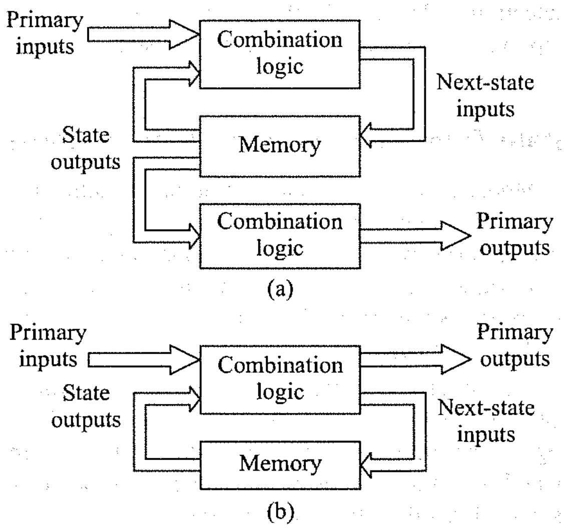


Fig. 11.1 **(a) Moore model, (b) Mealy model of sequential logic system**

In Section 11.2, we shall discuss how *conversion* from one model to other can be done through state diagram representation. Depending on application requirements we choose one of these two models or a mixed model where a part of the circuit follows Mealy model and the other Moore model.

We address all design related issues of synchronous sequential logic by solving a binary sequence detector problem in a step-by-step manner. We use both Moore model and Mealy model for this problem and note the pros and cons of each approach. Note that, any other design problem can be attempted in the same way. The solution presented in subsequent sections is particular to this problem but the approach is general in nature. The sequence detector problem is stated next.

The Problem Design a sequence detector that receives binary data stream at its input, *X* and signals when a combination '011' arrives at the input by making its output, *Y* high which otherwise remains low. Consider, data is coming from left, i.e. the first bit to be identified is 1, second 1 and third 0 from the input sequence.

11.2 STATE TRANSITION DIAGRAM

The first step in a sequential logic synthesis problem is to convert this word description to State transition diagram or Algorithm State Machine (ASM) Chart. ASM chart is discussed in Section 11.6. In this section, let us see how we arrive at state transition diagram following Moore and Mealy model. We use the problem presented in Section 11.1 for demonstration.

State Definitions: Moore Model

Since, the output is generated only from the state variables let us see how many of them are necessary. Let the detector circuit be at state *a* when initialized. State *a* can also be considered as one where none of the bit in input sequence is properly detected or the starting point of detection. Then if 1st bit is detected properly the circuit should be at a different state say, *b*. Similarly, we need two more states say, *c* and *d* to represent detection of 2nd and 3rd bit in proper order. When the detector circuit is at state *d*, output *Y* is asserted and kept high as long as circuit remains in state *d* signaling sequence detection. For other states detector output, $Y = 0$.

State Transition Diagram: Moore Model

In Moore model each state and output is defined within a circle in state transition diagram in the format *s*/*Y* where *s* represents a symbol or memory values identified with a state and *Y* represents the output of the circuit. An arrow sign marks state transition following an input value 0 or 1 that is written along the path. Note that *X* represents the binary data input from which sequence '011' is to be detected.

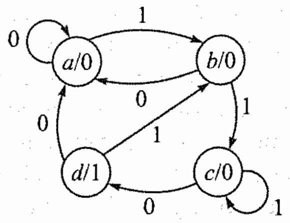


Fig. 11.2a State transition diagram of sequence detector: Moore model

Figure 11.2a shows the state transition diagram following Moore model. We arrive at it by following logic. The circuit is initialized with state *a*. If input data $X = 1$, the first bit of the sequence to be detected is considered detected and the circuit goes to state *b*. If $X = 0$ then it remains at state *a* to check next bit that arrives. If at state *b*, the circuit receives $X = 1$, then first two bit of the pattern is considered detected and it moves to state *c*. But at state *b*, if it receives $X = 0$ (i.e input sequence is '01') then detection has to start afresh as we need all three bits of ' 011' to match. Thus, the detector goes back to initial state *a*. At state *c*, if the circuit receives $X = 0$ then input bit stream is '011' and the circuit goes to state *d* and signals detection of pattern at state *d*. However, at *c* if $X = 1$, the detector is in a situation where it has received '111' in order. It stays at *c* so that if next arriving bit, $X = 0$ it should signal sequence detection. At state *d* if the circuit continues sequence detection job, receiving $X = 1$ it goes to state *b*. That ensures detection of '011' second time in input '011011'. For $X = 0$ the circuit goes to initial state *a* signifying not a single bit has been detected properly subsequent to previous detection.

State Definitions: Mealy Model

Since, the output can be derived using state as well as input we need three different states for 3-bit sequence detector circuit following Mealy model. The three states say, *a*, *b*, *c* represents none, 1st bit and 2nd bit detection. When the circuit is at state *c* if the input is as per the pattern the output is generated in state *c* itself with proper logic combination of input. Note the difference with Moore model where output is generated one clock cycle later in state *d* and also requires one additional state.

State Transition Diagram: Mealy Model

Here, the output is written by the side of input along arrow path in the format *X*/*Y*, where *X* and *Y* represent input and output respectively. Figure 11.2b shows state transition diagram of the given problem following Mealy model. The explanation is as follows.

The circuit is initialized with state *a*. If it receives input $X = 0$, it stays at *a* else goes to state *b* that signifies first bit is detected properly. In both the cases output, $Y = 0$ signifying no detection. At state *b*, if $X = 0$, the circuit returns to initial state *a*, i.e. no bit in given order is detected and if $X = 1$, goes to state *c*, signifying two bits in order are detected. In both the cases $Y = 0$. Now when at *c*, if input received is 0 then all the three bits of the pattern are received properly and sequence detection can be signaled through $Y = 1$. Also the circuit goes to initial state *a* and prepares for a new set of detection. At state *c*, if $X = 1$ then the sequence received is '111'. An arrival of 0 in next clock can make the detection '110' possible. So, at state *c* if $X = 1$ it is considered as two bits, '11' have been detected properly and the circuit remains at state *c*. The ouput at that time is $Y = 0$ since sequence is not fully detected.

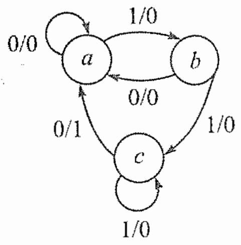


Fig. 11.2b State transition diagram of sequence detector: Mealy model

Conversion of Models

Conversion between Mealy and Moore models can take place as shown in Fig. 11.3 where, T_1, T_2, T_3 represent paths leading to state *a*. The path T_4 leads from state *a* when input is 1. If input is 0, state *a* leads to state *b* and there are no other paths reaching *b*. The rule of conversion is as follows. If all the transitions in a Mealy model to a particular state are associated with only one type of output then in corresponding Moore model that output becomes state output (Fig. 11.3a). If there is more than one output in Mealy model we need as many intermediate state variables, as shown in Fig. 11.3b. In Fig. 11.3c it is shown how to treat transitions that loop within a particular state. The reverse of this is applied in converting Moore model to Mealy model.

As an example, let us look at equivalence between two models of the sequence detector problem shown in Fig. 11.2a and Fig. 11.2b. In Mealy model we have paths leading to state *a*, have two different types of outputs. So state *a* of Mealy model get divided in two as *a* and *d* in Moore model. Since there is a loop in state *a* itself for one input, conversion rule shown in Fig. 11.3c is applicable. For other states there is no such conflict and a direct conversion is possible following Fig. 11.3a.

Now that we know one model can be obtained from other, i.e. logical equivalence exists between the two, we let application constraints (as discussed in Section 11.1) decide which one is to be chosen for a particular problem.

SELF-TEST

1. What is a state transition diagram?
2. How does state transition diagram of a Moore Machine differ from Mealy machine?

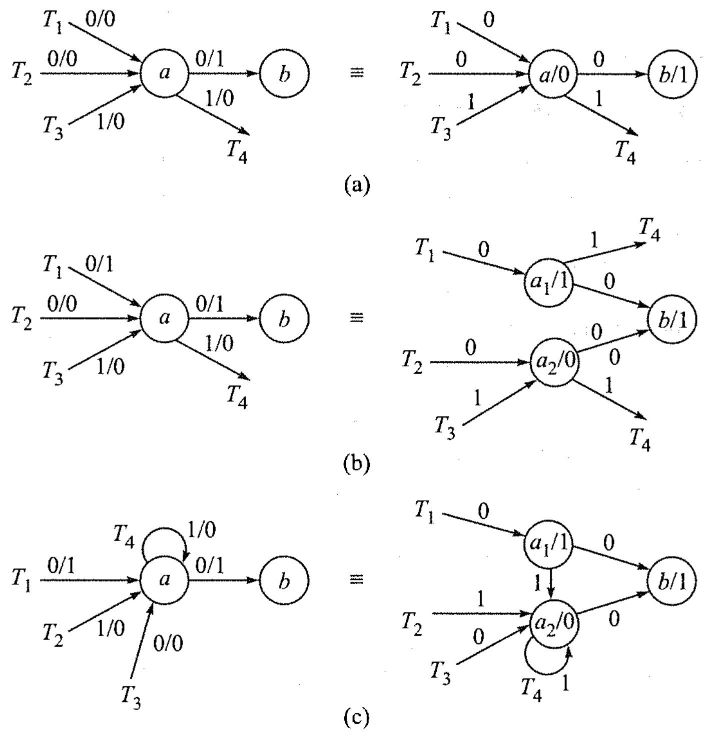


Fig. 11.3 Conversion between Mealy and Moore model

11.3 STATE SYNTHESIS TABLE

The next step in design process is to develop *state synthesis table*, also called *circuit excitation table* or simply *state table* from state transition diagram. Note that for m number of memory elements we can have up to 2^m number of different states in a circuit. Once we decide how many memory elements are to be used, we go for state assignment.

Often, we need to exercise state reduction technique before state assignment to remove redundancy in state description. Redundancy may come while converting word description of a complex problem to state transition diagram. We shall discuss state reduction techniques in Section 11.7.

State Assignment

Here, we allocate each state a binary combination of memory values. For the given problem, both Moore and Mealy models require minimum two flip-flops (say A and B) to define their states (4 for Moore and 3 for Mealy). Let the state assignment be as follows.

$$a: B = 0,\quad A = 0 \qquad b: B = 0,\quad A = 1 \qquad c: B = 1,\quad A = 0 \qquad d: B = 1,\quad A = 1$$

Note that Mealy model does not use state d. Assignment can be done in any order, e.g. we can make a: $B = 0, A = 1$ and b: $B = 0, A = 0$ and proceed with the design. However, one set of state assignment may give simpler final logic circuit over other. Though there is no definite state assignment rule that gives minimum hardware for an implementation, logical adjacency between transition states often helps. In Problem 11.7 to 11.10, we shall see how a different state assignment for this sequence detector problem asks for different hardware requirement.

State Synthesis Table

The next design step is to decide what kind of memory elements are to be used for our design. Flip-flops are commonly used for this purpose. A ROM based implementation is discussed in Section 11.5. When we use flip-flops we take note of the fact that there are different types of them available. Each flip-flop has a unique characteristic equation and excitation table (Section 8.9). In synthesis problem we have to find out how flip-flop inputs are to be connected and how final output is generated from flip-flop output. For this, we use state synthesis table that gives the input requirement of all flip-flops for a given state transition diagram. Before we prepare this table we should decide which flip-flop we are going to use. We normally prefer *JK* flip-flop as it has maximum number of don't care states in its excitation table and that leads to simpler design equations. We design the given sequence detector circuit using *JK* flip-flops.

Moore Model

State synthesis table obtained from state transition diagram of Moore model (Fig. 11.2a) and excitation table of *JK* flip-flop (Fig. 8.34) is shown in Table 11.1. It has eight rows as for each of the four possible states there can be two different types of inputs. The table is prepared as follows. When the circuit is at state 00, i.e. *a* and receives $X = 0$ it remains at state 00 and output in this state $Y = 0$. Since both B and A flip-flop makes $0 \rightarrow 0$ transition both the flip-flops should have input 0× from excitation table. This way first four columns of the table (present state, input, next state, output) are filled from state transition diagram and last two columns (B and A flip-flop inputs) from flip-flop excitation table.

Table 11.1 State Synthesis Table for Moore Model

Present State		*Present Input*	*Next State*		*Output*				
B_n	A_n	X_n	B_{n+1}	A_{n+1}	Y_n	J_B	K_B	J_A	K_A
0	0	0	0	0	0	0	×	0	×
0	0	1	0	1	0	0	×	1	×
0	1	0	0	0	0	0	×	×	1
0	1	1	1	0	0	1	×	×	1
1	0	0	1	1	0	×	0	1	×
1	0	1	1	0	0	×	0	0	×
1	1	0	0	0	1	×	1	×	1
1	1	1	0	1	1	×	1	×	0

Mealy Model

Since, Mealy Model requires three states for this problem we have six rows in state synthesis table as in each state there can be two different types of input $X = 0$ or $X = 1$. Table 11.2 represents state synthesis table for Mealy model. The method remains the same as Moore model but we use state transition diagram (Fig. 11.2b) corresponding to Mealy model from Section 11.2.

Table 11.2 State Synthesis Table for Mealy Model

Present State		*Present Input*	*Next State*		*Present Output*				
B_n	A_n	X_n	B_{n+1}	A_{n+1}	Y_n	J_B	K_B	J_A	K_A
0	0	0	0	0	0	0	×	0	×
0	0	1	0	1	0	0	×	1	×
0	1	0	0	0	0	0	×	×	1
0	1	1	1	0	0	1	×	×	1
1	0	0	0	0	1	×	1	0	×
1	0	1	1	0	0	×	0	0	×

11.4 DESIGN EQUATIONS AND CIRCUIT DIAGRAM

In this section, we discuss how to get final circuit diagram from state synthesis table (Section 11.3) through design equation. In design equation we express flip-flop inputs as a function of present state, i.e. memory values (here, *B* and *A*) and present input (here, *X*). This ensures proper transfer of the circuit to next state. The design equations also give output (here, *Y*) equation in terms of state variables or memory elements in Moore model and state variables together with input in Mealy model. We normally use Karnaugh map technique to get a simplified form of these relations.

Moore Model

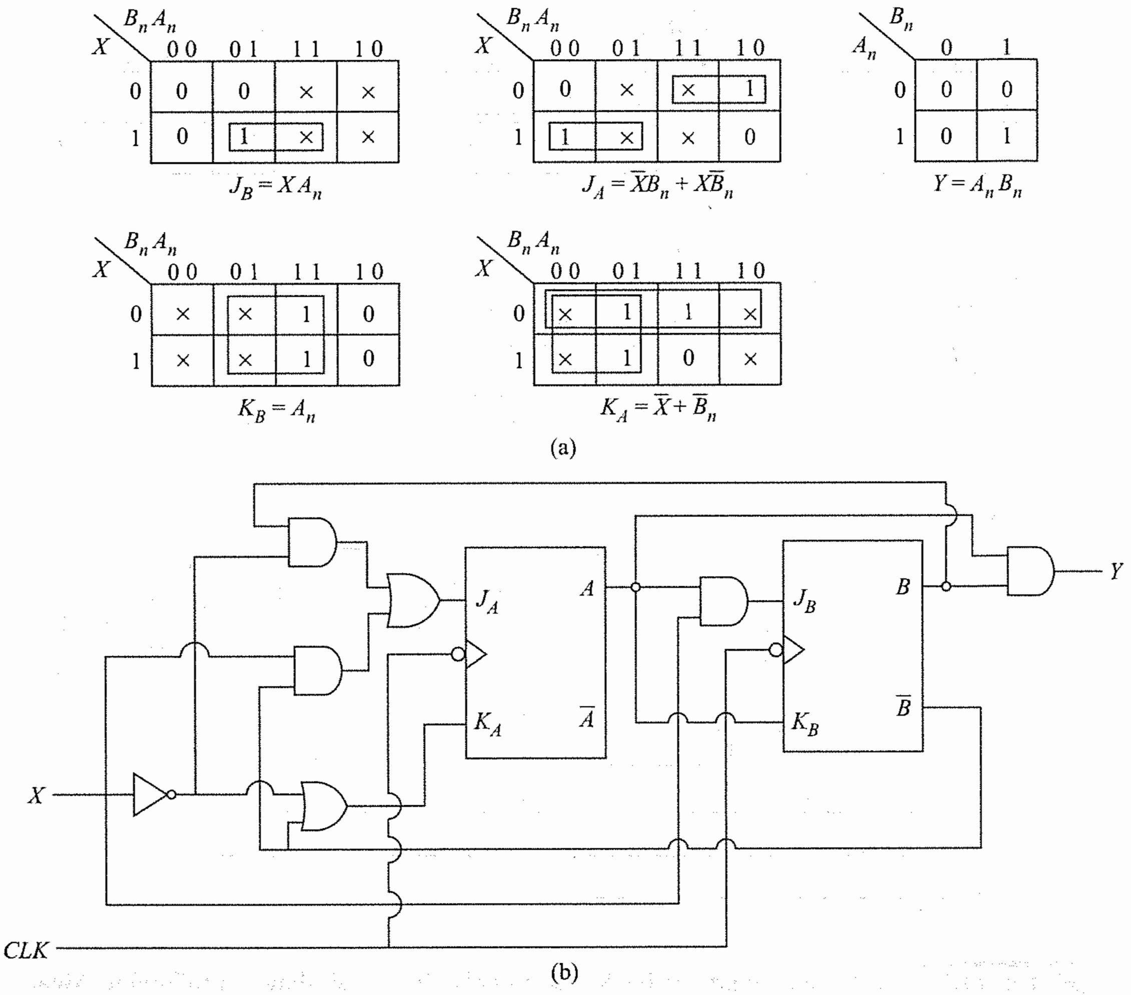
Figure 11.4a presents Karnaugh map developed from state synthesis Table 11.1 and also shows corresponding design equations. Figure 11.4b shows the sequence detector circuit diagram developed from these equations. This is done in the following manner. Equation $J_B = XA$ requires *J* input of flip-flop *A* to be fed from a two input AND gate, inputs to which are *X* and *A*. The other inputs and output are obtained in similar way. Note that, output is generated by AND operation on two flip-flop outputs and does not use *X*.

Mealy Model

Using state synthesis table corresponding to Mealy model (Table 11.2) we can fill six positions in each Karnaugh map (Fig. 11.5a). Locations $B_nA_nX = 110$ and $B_nA_nX = 111$ are filled with don't care(×) conditions as such a combination never occur in the detector circuit if properly initialized. The design equations are obtained from these Karnaugh maps from which circuit diagram is drawn as shown in Fig. 11.5b. Note that in this circuit, output directly uses input information.

SELF-TEST

3. What is an excitation map?
4. Is there any difference in hardware requirement between Moore and Mealy machine?
5. In the sequence detector circuit designed here, show the output in each clock cycle by completing Table 11.3.



$J_B = XA_n$
$J_A = \overline{X}B_n + X\overline{B}_n$
$Y = A_nB_n$
$K_B = A_n$
$K_A = \overline{X} + \overline{B}_n$
(a)
X
CLK
Y
(b)

Fig. 11.4 (a) Design equations for Moore model, (b) Circuit diagram following Moore model

Table 11.3

Clock Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Input	0	1	0	1	1	0	1	0	1	1	0	1	1	0	1
Moore output															
Mealy output															

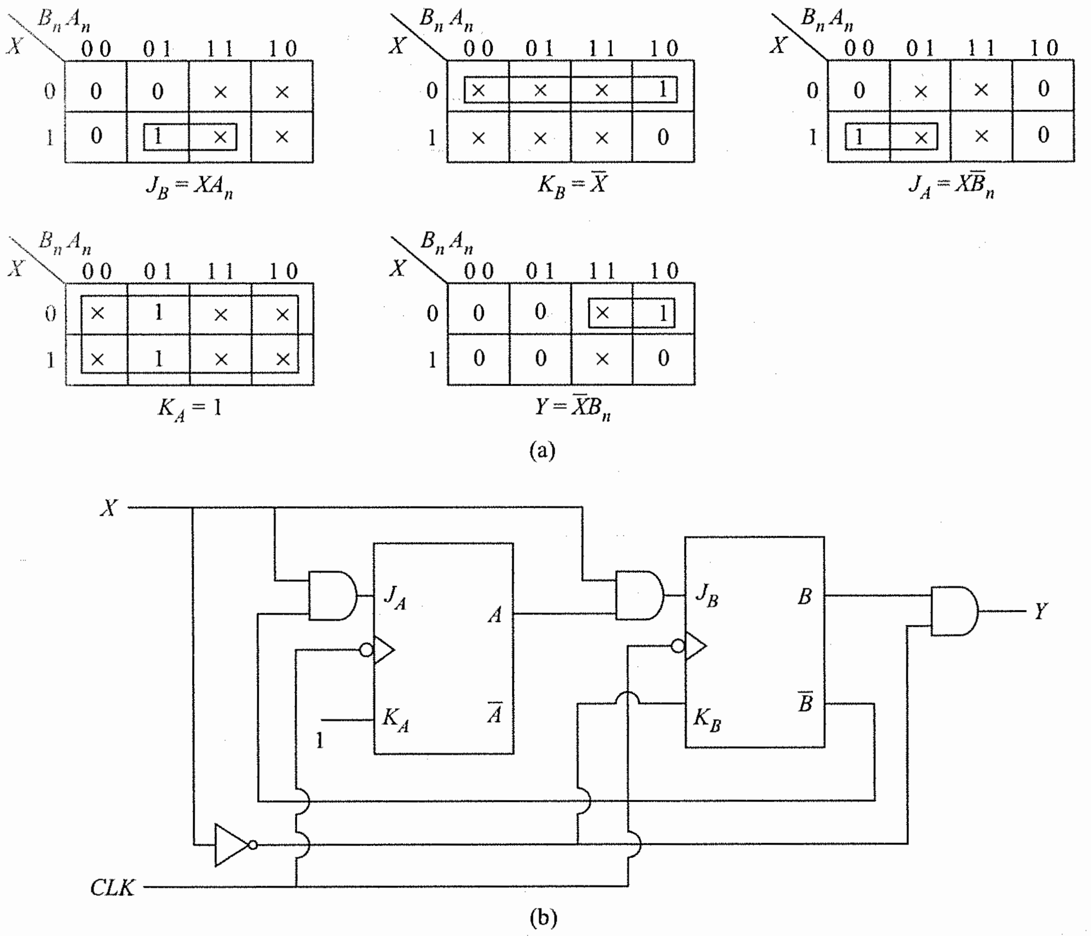


Fig. 11.5 **(a) Design equations for Mealy models, (b) Circuit diagram following Mealy model**

11.5 IMPLEMENTATION USING READ ONLY MEMORY

In this section we present an interesting solution to sequential logic problem using Read Only Memory (ROM) which though called memory is a combinatorial circuit. The output of ROM is immediately available when a particular location in memory is addressed. The detailed description of ROM, its architecture, type and operation is given in Sections 4.9 and 13.5.

For our design purpose we need to have a ROM that has as many memory locations as the number of rows in a state synthesis table. Note that, each row is uniquely identified by present state and present input. This present state and input combination through an address decoder points to memory spaces in ROM. In each location of ROM we store the next state value of the circuit.

We also need a bank of Delay flip-flops, the number is same as the number of state variables or memory elements. Next state information for each state variable that is stored in ROM is fed to these D flip-flop inputs. At clock trigger they appear at the output of the flip-flop. Now that the circuit has advanced by one clock cycle these D flip-flop outputs serve as present state information and fed to ROM address decoder. Together with present input they point to another location in memory that has next state information. ROM being a combinatorial circuit these next state values are immediately available to D flip-flop inputs and the cycle goes on. The final output is generated from state variables in Moore model and also uses direct input in Mealy model.

Moore Model

For the sequence detector problem we need 8×2 ROM as there are 8 rows in state synthesis Tables 11.1 and 11.2. The circuit diagram is shown in Fig. 11.6. The 3 to 8 address decoder is fed by B, A and X. The output of decoder is 000, 001, 010 .. in same order as they appear in state Table 11.1. For example, when $BAX = 000$ next state is 00 from state table and we store 00 in ROM corresponding to decoder output 000. Similarly, next memory values stored in ROM are 01, 00, 10, 11... in order from state table. D flip-flop connections are explained before and output is generated following logic equation $Y = AB$.

The circuit functions like this. The D flip-flops are initially cleared, i.e. $BA = 00$. If $X = 0$, the first location in ROM corresponding to $BAX = 000$ is selected and ROM output = 00 and at clock trigger next state remains

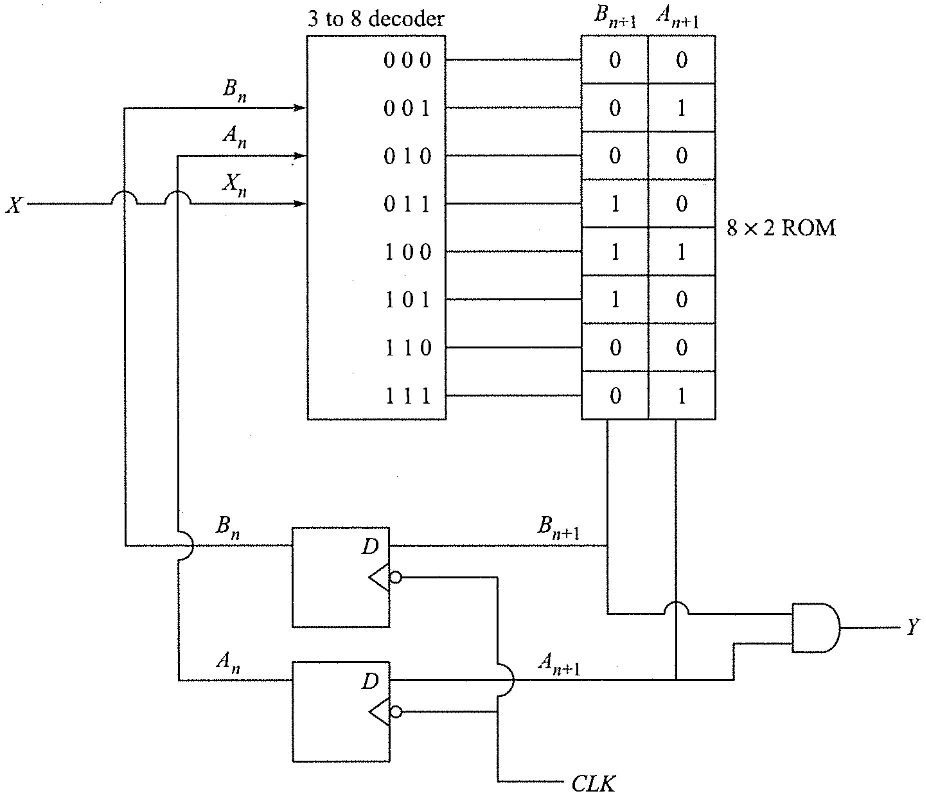


Fig. 11.6 ROM based implementation of sequence detector: Moore model

at $BA = 00$. Thus 00 state remains at 00 for $X = 0$. If $X = 1$, then $BAX = 001$ location in ROM is selected which stores 01, i.e. the circuit (D flip-flops) goes to $BA = 01$ state with clock trigger. For $BA = 01$, if $X = 0$ then $BAX = 010$ location in ROM is selected which stores 00 that means with NT of clock the circuit goes to state 00 or initial state. If $BA = 01$ and $X = 1$ then $BAX = 011$ location of ROM is selected which stores 10. Thus next state becomes 10 with NT. Now at $BA = 10$, if $X = 0$ then $BAX = 100$ location is selected which stores 11 and next state becomes 11. If we have recorded input values we see when 100 location in ROM is selected in ROM the pattern '011' has arrived in proper order. Stored ROM data is immediately available in the same clock cycle and we can generate circuit output from this signaling detection.

Thus, compared to previous implementation here sequence detection signal comes one cycle earlier. Also note that the design process is very straightforward. We don't need to remember flip-flop excitation table or simplify design equation, which gives different circuit for different problem. Here, for all problems circuit remains same only the content of ROM changes. The output logic may also be different but we have an option to store the output as 3rd bit in the ROM and we then don't need any output logic equation to be realized by basic gates. Refer to Problems 11.15 and 11.16.

Mealy Model

ROM based solution of Mealy model uses state synthesis table in the same way as Moore model ROM locations are selected by present state and input as appears in state table and next state value fills corresponding ROM locations (Fig. 11.7). Delay flip-flop banks are used in the same way but final output is generated from

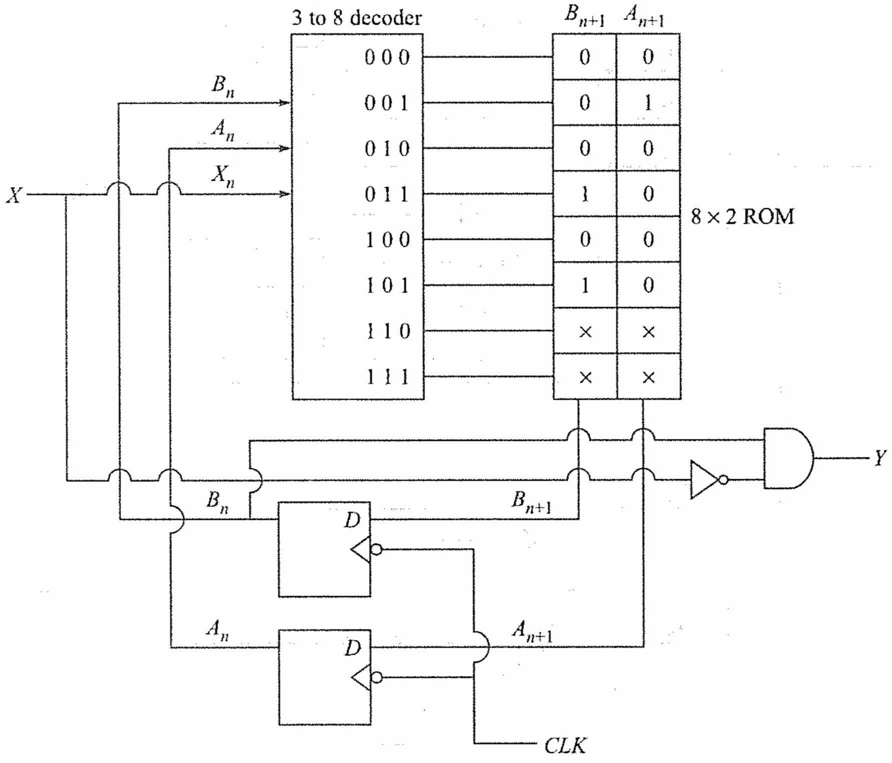


Fig. 11.7 ROM based implementation of sequence detector: Mealy model

D flip-flop outputs (representing present state) and data input. In Moore model we have used ROM outputs directly to generate sequence detector output. Note that ROM of size 6×2 is sufficient for Mealy model and last two locations of 8×2 ROM are not used.

Example 11.1 Give design equations for the synchronous sequential logic circuit that has two inputs X and Y. The output Z of this circuit is generated according to the timing diagram shown in Fig.11.8.

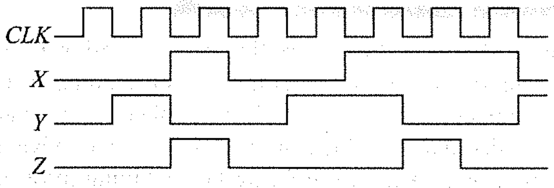
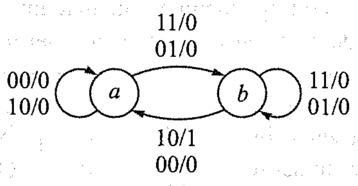


Fig. 11.8 **Timing diagram for Example 11.1**

Solution Instead of word description we have timing diagram explaining the problem. On careful observation we find Z remains high for one clock period when Y goes from high to low and if at that time (Y low) the other input X remains at logic high. Thus if we adopt a Mealy model, the circuit needs one memory element that remembers if previous state of Y was high for any $X = 1$, $Y = 0$ input. The state transition diagram, state synthesis table and design equations are shown in Figs. 11.9(a), (b) and (c) respectively. The design has been done with D flip-flop in which D input simply follows next state. Refer to excitation table of Fig. 8.34.



(a)

A_n	X	Y	A_{n+1}	D_n	Z
0	0	0	0	0	0
	0	1	1	1	0
	1	0	0	0	0
	1	1	1	1	0
1	0	0	0	0	0
	0	1	1	1	0
	1	0	0	0	1
	1	1	1	1	0

(b)

A_n \ XY	00	01	11	10
0	0	1	1	0
1	0	1	1	0

$D_n = Y$

A_n \ XY	00	01	11	10
0	0	0	0	0
1	0	0	0	1

$Z = X\overline{Y}A_n$

(c)

Fig. 11.9 **(a) State transition diagram, (b) State synthesis table, (c) Design equations for Example 11.1**

SELF-TEST

6. How use of flip-flop is different in ROM based implementation?
7. Complete the table as shown in Q. 5 for ROM based Moore and Mealy models.

11.6 ALGORITHMIC STATE MACHINE

Algorithmic State Machine (ASM) is a flow chart like representation (ASM Chart) of the algorithm a state machine performs. State Transition diagrams though more compact in representation has certain disadvantages. For relatively more complex problem where number of inputs and states are higher the state diagram space becomes so crowded that it is difficult to read. The other advantage of ASM chart is that, it handles implementation issues with greater ease offering better timing information. In ASM chart, square boxes represents a state. If a state generates an unconditional output (Moore model) it can be specified within the square box. A diamond shaped box represents decision to be taken and normally the variable or the condition that is tested is placed inside it with a question mark. There are two exit paths of this decision box since the decision is binary in nature. For Mealy model, oval shaped boxes are used to describe the output that depends on present state as well as the present input. Circles are used to denote start, stop of the algorithm and also the connector point of an ASM chart when it becomes too large and needs to be drawn at more than one place. Entry and exit of each ASM block is shown by arrow headed connecting link.

We take a new example and discuss its design using ASM chart. ASM chart for sequence detector problem of previous section is shown in Example 11.4.

Vending Machine Problem

The task is to design a synchronous logic control unit of a vending machine. The machine can take only two types of coins of denomination 1 and 2 in any order. It delivers only one product that is priced Rs. 3. On receiving Rs. 3 the product is delivered by asserting an output $D = 1$ which otherwise remains 0. If it gets Rs. 4 then product is delivered by asserting X and also a coin return mechanism is activated by output $Y = 1$ to return a Re. 1 coin. There are two sensors to sense the denomination of the coins that give binary output as shown in the following table. The clock speed is much higher than human response time, i.e. no two coins can be deposited in same clock cycle.

I	J	Coin
0	×	No Coin dropped
1	0	One Rupee
1	1	Two Rupees

ASM Chart

The ASM chart is prepared following Mealy model and is shown in Fig. 11.10. The initial state when no coin is deposited is designated as state a. Note that, sensor output $I = 0$ indicates no coin is deposited. At every clock trigger I is tested and if found 0 the circuit retraces its path to state a and obviously none of X and Y is asserted, i.e. no product is delivered or coin returned. If $I = 1$, the controller tests J. If $J = 0$ it goes to state b that represents Re. 1 is received and if $J = 1$, goes to state c indicating Rs. 2 is received. The controller remains at state b if no further coin is deposited found by checking I. Now, if $I = 1$ and $J = 0$, the machine has received two Re. 1 coins in succession and should move to state c. But $I = 1$ and $J = 1$ means a Rs. 2 coin is received following Re. 1 totaling Rs. 3 the cost of the product. Hence, the product is delivered by asserting $X = 1$ and the circuit

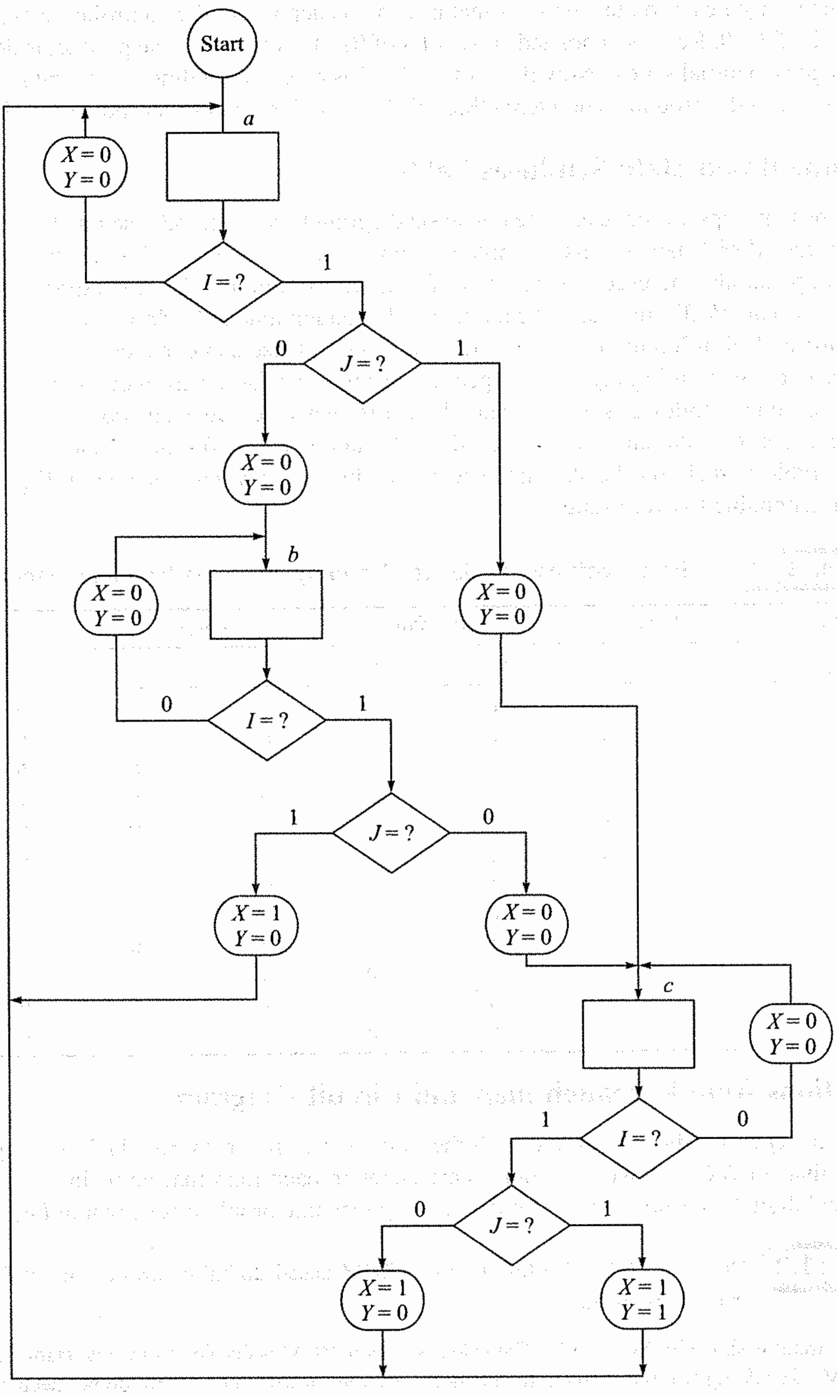


Fig. 11.10 ASM chart for vending machine problem: Mealy model

goes to initial state. At state c if on testing $I = 1$ that is a coin is deposited, the controller tests J to ascertain if it is Re. 1 or Rs. 2. If $J = 0$, Re. 1 is deposited and a total of Rs. 3 is received. The product is delivered by $X = 1$ and the circuit goes to initial state a. Now if $J = 1$ then Rs. 2 is received totaling Rs. 4. Then Re. 1 is returned by asserting $Y = 1$, also the product is delivered through $X = 1$ and the controller moves to initial state a.

State Assignment and State Synthesis Table

The subsequent design steps are same as state transition diagram based method discussed before. We prepare state table from this ASM Chart. In this example we show how to use D flip-flop as the memory element though JK flip-flop can also be used. As expected, filling up of columns that corresponds to D input in a given state is easier than JK flip-flop, also the number of Karnaugh map to be drawn for each flip-flop is half that of JK flip-flop as D flip-flop has only one data input. But all these come at a cost of increased hardware complexity. This example will highlight this aspect of design issue for synchronous sequential circuit. The state assignment is done as follows. Since there are three different states we need two flip-flops (say, B and A) to represent them. Let $BA = 00$ represent state a, $BA = 01$ state b, $BA = 10$ state c. State $BA = 11$ is not used in this problem. Table 11.4 shows the state table corresponding to ASM chart shown in Fig. 11.10 and also the D inputs corresponding to every state.

Table 11.4 **State Synthesis Table for Vending Machine Problem: Mealy Model**

Present	*State*	*Input*		*Next State*		*Output*		D_B	D_A
B_n	A_n	I	J	B_{n+1}	A_{n+1}	X	Y		
		0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
		1	0	0	1	0	0	0	1
		1	1	1	0	0	0	1	0
		0	0	0	1	0	0	0	1
0	1	0	1	0	1	0	0	0	1
		1	0	1	0	0	0	1	0
		1	1	0	0	1	0	0	0
		0	0	1	0	0	0	1	0
1	0	0	1	1	0	0	0	1	0
		1	0	0	0	1	0	0	0
		1	1	0	0	1	1	0	0

Design Equations from Karnaugh map and Circuit Diagram

Karnaugh maps for each flip-flop input and both the outputs are shown in Fig. 11.11a along with design equations. Note that for $BA = 11$ we have don't care states in each map that helps in minimizing design equation. The final digital controller circuit for the vending machine problem is shown in Fig. 11.11b.

Example 11.2 Draw the Delay flip-flop-Decoder-ROM based digital controller circuit for the vending machine problem.

Solution The circuit is shown in Fig. 11.12a. The values stored in ROM is derived from state stable. Here we have used higher ROM size adding two more bits in the memory location for each address but do not need any basic gate for output logic. However, we can use logic gates as shown in sequence detector problem and reduce two columns corresponding to X and Y in ROM.

$IJ \backslash B_nA_n$	00	01	11	10
00	0	0	×	1
01	0	0	×	1
11	1	0	×	0
10	0	1	×	0

$D_B = \bar{I}B_n + I\bar{J}A_n + IJ\bar{B}_n\bar{A}_n$

$IJ \backslash B_nA_n$	00	01	11	10
00	0	1	×	0
01	0	1	×	0
11	0	0	×	0
10	1	0	×	0

$D_A = \bar{I}A_n + I\bar{J}\bar{B}_n\bar{A}_n$

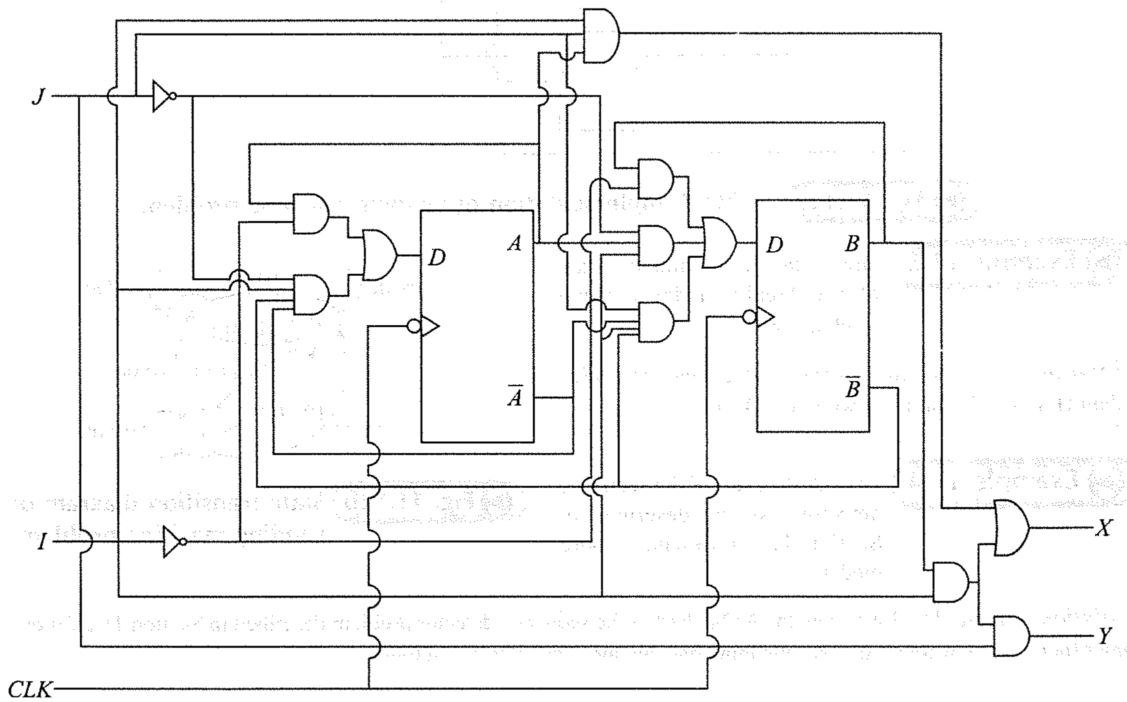
$IJ \backslash B_nA_n$	00	01	11	10
00	0	0	×	0
01	0	0	×	0
11	0	1	×	1
10	0	0	×	1

$X = IB_n + IJA_n$

$IJ \backslash B_nA_n$	00	01	11	10
00	0	0	×	0
01	0	0	×	0
11	0	0	×	1
10	0	0	×	0

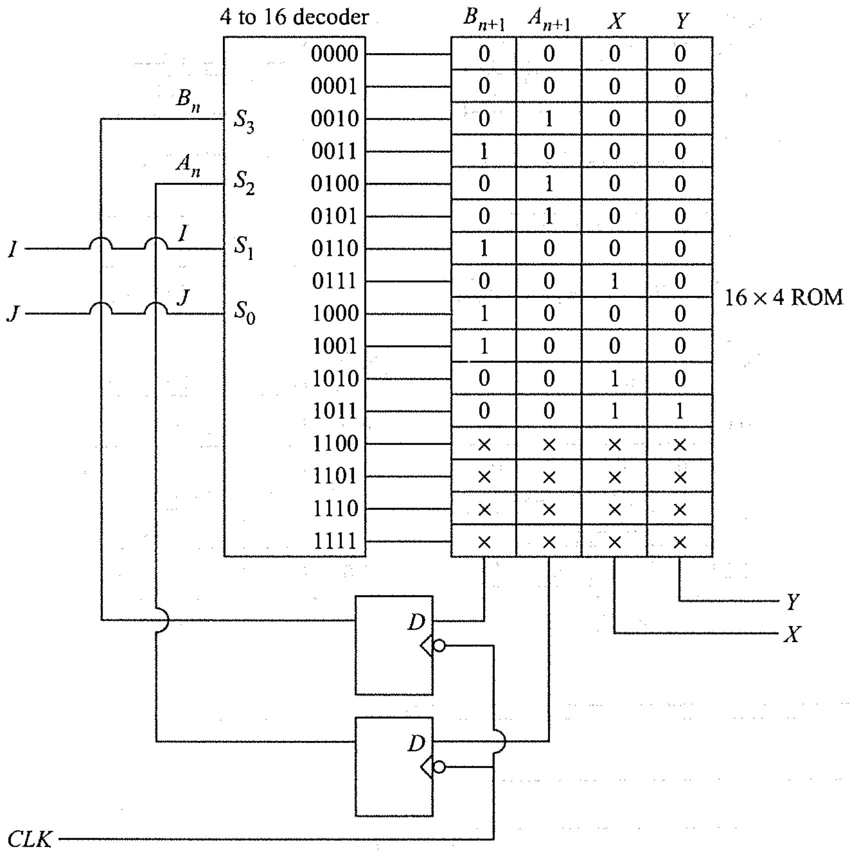
$Y = IJB_n$

(a)



(b)

Fig. 11.11 (a) Design equation, (b) Circuit diagram for vending machine problem: Mealy model



	B_{n+1}	A_{n+1}	X	Y
0000	0	0	0	0
0001	0	0	0	0
0010	0	1	0	0
0011	1	0	0	0
0100	0	1	0	0
0101	0	1	0	0
0110	1	0	0	0
0111	0	0	1	0
1000	1	0	0	0
1001	1	0	0	0
1010	0	0	1	0
1011	0	0	1	1
1100	×	×	×	×
1101	×	×	×	×
1110	×	×	×	×
1111	×	×	×	×

Fig. 11.12a **ROM implementation of vending machine problem**

Example 11.3 Draw state transition diagram of the Mealy model vending machine problem.

Solution The diagram can easily be drawn from ASM chart (Fig. 11.10) and is shown in Fig. 11.12b.

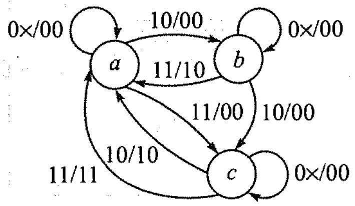


Fig. 11.12b **State transition diagram of vending machine problem**

Example 11.4 Draw ASM chart of the sequence detector problem described in Section 11.2 following Moore model.

Solution In Fig. 11.13 we show the ASM chart of the sequence detector problem described in Section 11.2 following Moore model where X denotes the input data bit and Y the detector output.

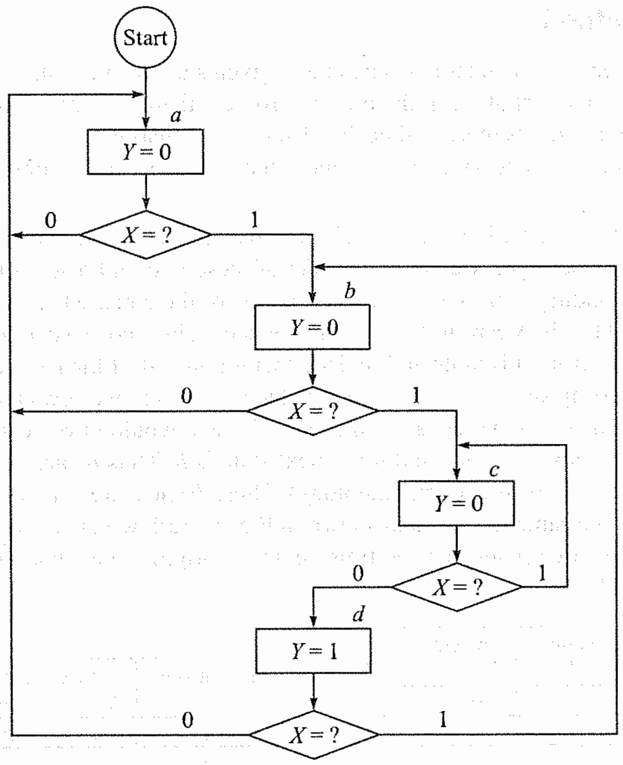


Fig. 11.13 **ASM chart of sequence detector problem: Moore model**

Note the similarity between Moore model state transition diagram of Fig. 11.2a and ASM chart shown here. Once we arrive at the ASM Chart the rest of the design procedure starting from state assignment up to final circuit diagram is same as what is discussed in Section 11.6. ASM Chart for the Mealy model sequence detector is left as exercise for the reader.

11.7 STATE REDUCTION TECHNIQUE

In design of sequential logic circuit state reduction techniques play an important role, more so for complex problems. While converting problem statement to state transition diagram or state table we may use more number of states than necessary. On removing redundant states the clarity of the problem is enhanced. This also offers simpler solution and less hardware to implement a circuit. We explain two state reduction techniques through an example.

Let the state transition diagram drawn following a Mealy model is as shown in Fig. 11.14. The goal is to identify and remove redundant states, if any and obtain the reduced state diagram.

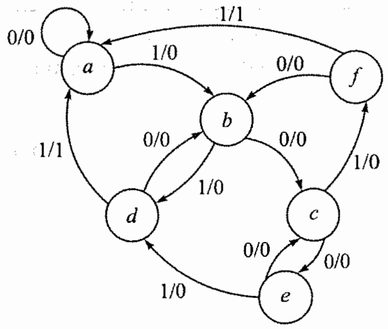


Fig. 11.14 **A state transition diagram**

Row Elimination Method

In this method, we first prepare a state table where at any given state the next state and present output(s) are written for each combination of input(s). In the present problem there are only two possible values of input $X = 0$ and $X = 1$. For 2 input circuits there will be $2^2 = 4$ such combinations in this table. Now, two states are considered equivalent if they move to same or equivalent state for every input combination and also generate same output.

Figure 11.15a shows the state table for Fig. 11.14 and we see that states *b* and *e* are equivalent as next state and output are same. Therefore, we can retain one of these two and discard the other. Let us retain *b* and eliminate row corresponding to present state *e* and in rest of the table, wherever *e* appears we replace it by *b* and get table of Fig. 11.15b. A careful look on this reduced table shows state *d* and *f* are equivalent. We retain *d* and eliminate row *f* from this table and replace *f* with *d* in rest of the rows and get Fig. 11.15c. This table places us in an interesting situation as far as equivalence between two rows are concerned. For states *b* and *c* except for next state at $X = 0$ the rest are same. Now *b* and *c* would have been equivalent if these next states are equivalent. For *b*, next state is *c* and for *c*, next state is *b*. Thus *bc* are equivalent if next states *cb* are equivalent which can always be true (from tautology). Thus, *b* and *c* are equivalent and state *b* is retained and row *c* is eliminated in the same manner and shown in Fig. 11.15d, which cannot be further reduced. The final reduced state table has three states reduced from six in the original state diagram and final reduced state diagram is shown in Fig. 11.15e.

Present state	Next state		Present output	
	$X=0$	$X=1$	$X=0$	$X=1$
a	*a*	*b*	0	0
$\surd b$	*c*	*d*	0	0
c	*e*	*f*	0	0
d	*b*	*a*	0	1
$\surd e$	*c*	*d*	0	0
f	*b*	*a*	0	1

(a) Original table

Present state	Next state		Present output	
	$X=0$	$X=1$	$X=0$	$X=1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*b*	*f*	0	0
$\surd d$	*b*	*a*	0	1
$\surd f$	*b*	*a*	0	1

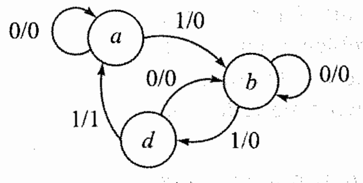
(b) After one row elimination

Present state	Next state		Present output	
	$X=0$	$X=1$	$X=0$	$X=1$
a	*a*	*b*	0	0
$\surd b$	*c*	*d*	0	0
$\surd c$	*b*	*d*	0	0
d	*b*	*a*	0	1

(c) After two row elimination

Present state	Next state		Present output	
	$X=0$	$X=1$	$X=0$	$X=1$
a	*a*	*b*	0	0
b	*b*	*d*	0	0
d	*b*	*a*	0	1

(d) Final reduced table after three row elimination

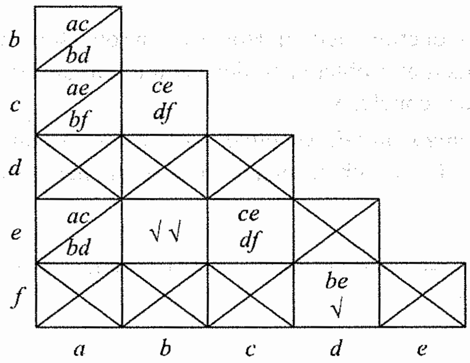


(e)

Fig. 11.15 (a)–(d) Tables showing row elimination steps, (e) Reduced state diagram

Implication Table Method

Implication table provides a more systematic approach towards solution of a complex state reduction problem. For n states in the initial description we have n–1 rows in implication table and as many number of columns. Refer to implication table of Fig. 11.16a for the given state reduction problem. The cross-point in an implication table is the location where a row and a column meet. Here, the conditions for equivalence between the states crossing each other are tested. We use state table of Fig. 11.15a derived from state transition diagram to fill up implication table. The steps to be followed are given next.



(a)

$e : e$
$d : e\,(df)$
$c : e\,(df)\,(ce) \equiv (df)\,(ce)$
$b : (df)\,(ce)\,(bc) \equiv (df)\,(bce)$
$a : a\,(df)\,(bce)$

$\text{P} = (df)\,(bce)\,(a)$

(b)

Fig. 11.16 Implication table method of state reduction: (a) Implication table, (b) Partition table

In **Step 1**, we identify the states, which cannot be equivalent, as their outputs do not match. This we denote by putting a double-cross in respective cross points. In this problem state d and f only have output = 1 for X = 1 unlike other states. Thus, intersection of d and f with others except themselves are double crossed.

In **Step 2**, for other cross points, we write necessary conditions for equivalence of intersecting states. As an example, let us look at intersection of states a and b. To get the necessary condition we refer to rows starting with a and b in state table of Fig. 11.15a. We find that at X = 0, a stays at a while b goes to c and at X = 1, a goes to b while b goes to d. Thus, a and b can only be equivalent if next states a and c are equivalent and also if b and d are equivalent. This is written at cross point of a and b in implication table. Note that output of a and b match, else, it would have got a double cross in Step 1. We similarly fill up other cross points and note that b and e are equivalent and does not require any equivalence between other states and a double tick mark is placed at that cross point.

In **Step 3**, we use relationships obtained in Steps 1 and 2, specially the ones represented by double cross and double tick mark and check if any other cross points can be crossed or ticked. Since df equivalence depends only on equivalence be which is true, they are equivalent and that cross point can be ticked. Similarly, ac cannot be equivalent, as it requires bf to be equivalent which is not true. Hence, ac intersection is crossed.

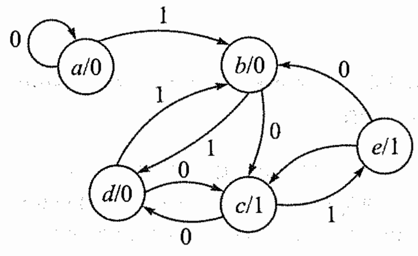
In **Step 4**, we keep repeating Step 3 and cross or tick (if possible) as many cross points in the implication table as possible. We see ab and ae cross points can be crossed as they need ac to be equivalent which is crossed in the previous step. With no further crossing and ticking possible the implication table is fully prepared and we go to Step 5.

In **Step 5**, we check pairwise equivalence starting from rightmost column e of implication table. Since, the only cross point, representing ef equivalence along column e is crossed there is no equivalence possible at column *e* and we write *e* in Fig. 11.16b in the first place. In column *d*, we find *df* are equivalent and along *d* in Fig. 11.16b the same is written. In column *c*, we find *ce* are equivalent as *df* is equivalent. In column *b*, equivalence between *be* and *bc* can be observed as *ce* and *df* are already considered equivalent and the same are written along *c* and *b*. Note that if *p* and *q* are equivalent and so are *q* and *r* then *p* and *q* are equivalent. In that case, *pqr* can form one group and any one of its members can represent the group. Since, column *a* does not give any equivalent pair the final partition table is represented as P = (*df*) (*bce*) (*a*) and has three partitions. Then three states are sufficient to solve this problem each representing one partition. If *d* represents any of *df* and *b* represents *bce* in Fig. 11.16b we get reduced state table as shown in Fig. 11.15d and corresponding reduced state diagram is shown in Fig. 11.15e.

Note that the final result is same by both the state reduction method. However, in row elimination method one has to draw many tables for a complex state reduction problem and depend a lot on observation power. The implication method being more systematic is more conclusive.

We shall discuss state reduction technique for *incompletely specified state table* in connection with asynchronous sequential circuit design in Section 11.10. In such problems some of the next states or output remains unspecified and treated as don't care condition.

Example 11.5 Reduce state transition diagram (Moore Model) of Fig. 11.17a by (i) row elimination method and (ii) implication table method



(a)

Present state	Next state $X=0$	$X=1$	Present output
a	*a*	*b*	0
√*b*	*c*	*d*	0
c	*d*	*e*	1
√*d*	*c*	*b*	0
e	*b*	*c*	1

(b) Original table

Present state	Next state $X=0$	$X=1$	Present output
a	*a*	*b*	0
b	*c*	*b*	0
√*c*	*b*	*e*	1
√*e*	*b*	*c*	1

(c) Table after elimination of one row

Present state	Next state $X=0$	$X=1$	Present output
a	*a*	*b*	0
b	*c*	*b*	0
c	*b*	*c*	1

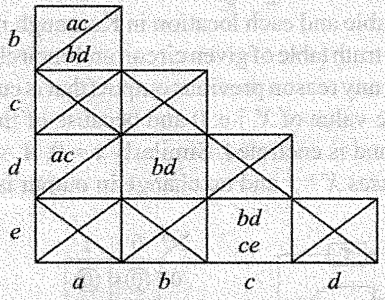
(d) Final reduced table after elimination of two rows

Fig. 11.17 **Reduction by row elimination method**

Solution

(i) Refer to state table of Fig. 11.17b obtained from state transition diagram. Comparing row *b* and *d* we see they are equivalent because that needs no other consideration except equivalence between themselves. Retaining *b* and replacing *d* by *b* in rest of the table we get Table of Fig. 11.17c. There we find *c* and *e* are equivalent and we retain *c* and replacing *e* by *c* get Fig. 11.17d. We see no further reduction is possible and final reduced state table that has three states.

(ii) Refer to state transition diagram and state table developed from it. Implication table is shown in Fig. 11.18. The non-compliance of output makes cross-points *de, be, ae, cd, bc, ac* non-equivalent and hence double crossed. From this we find *ab* and *ad* cannot be equivalent as that requires *ac* to be equivalent which is not true. Finally moving columnwise starting from *d* we get partition table and final partition P has three groups. Hence, the number of states is reduced to 3 from 5 by this technique.



d : *d*
c : *d* (*ce*)
b : *d* (*ce*) (*bd*) ≡ (*ce*) (*bd*)
a : (*ce*) (*bd*) *a*
P = (*ce*) (*bd*) (*a*)

Fig. 11.18 Reduction by implication table method

SELF-TEST

8. What is an ASM chart?
9. What is an implication table?
10. What is a partition table?
11. What is the usefulness of state reduction technique?

PART B: ASYNCHRONOUS SEQUENTIAL CIRCUIT

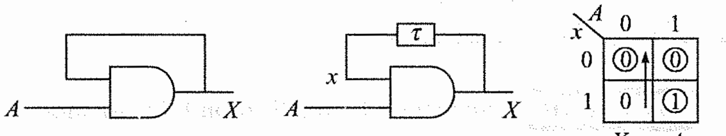
Asynchronous Sequential Circuit, also called Event Driven Circuit does not have any clock to trigger change of state. State changes are triggered by change in input signal. In clock driven circuit all the memory elements change their states together. In spite of all the advantages it offers, there are certain limitations with such circuit. The most important being the speed of operation. This is limited by the clock frequency since, state change can only take place at time $t = nT$, where T = Time period of clock signal and inverse of frequency and n is an integer. If the input changes in a manner that warrants change in the state, it cannot do that immediately and wait till the next clock trigger comes. Asynchronous sequential circuit is a solution to this however, design of such circuit is very complex and has several constraints to be taken care of, which is not required for synchronous circuit. Here, we shall discuss *fundamental mode* of operation of asynchronous sequential circuit where output change depends on change in input level. There is another type of such circuit called *pulse mode* where output change is affected by edge of the input pulse.

11.8 ANALYSIS OF ASYNCHRONOUS SEQUENTIAL CIRCUIT

As we have already noted memory is the most important element in sequential logic circuit. In synchronous system we use clock driven flip-flops which we cannot work here. This is done through feedback similar to basic latch portion of a flip-flop. Before we discuss that let us see how a two input AND gate and two input NAND gate behave with output fed back to one of the input. We shall use Karnaugh map for the analysis.

AND Gate

The two input AND gate with output fed back as one input is shown in Fig. 11.19a. The circuit can be redrawn as shown in Fig. 11.19b that includes the effect of propagation delay of the gate (say, τ), the finite time after which a gate reacts to its input. Thus, if X is current output obtained following logic relation and x is the feedback output we write, $x = X(t-\tau)$. The truth table is also called state table and each location in Karnaugh map a state of the asynchronous sequential circuit. Figure. 11.19c shows the truth table of given circuit and encircled states indicate stable condition of the circuit. For example, if $A = 0$ and by any reason previous output (that is currently fed back) $x = 0$ then, $X = x.A = 0.0 = 0$. After time $t = \tau$, x takes the value of X, i.e. 0 and because of that output X does not change. Thus, $x = 0, A = 0$ represents a stable state and is encircled. Similarly $x = 0, A = 1$ position and $x = 1, A = 1$ positions are also stable as in each of these cases $X = x$ and no change in output is necessary.



(a) AND gate (b) AND gate with propagation delay (c) Truth table

Fig. 11.19 Two input AND gate with output feedback

Let us now consider the following case. The circuit is at $x = 1, A = 1$, a stable state. Now A is made 1 and held at that value. How does the circuit react? First of all, following Karnaugh map the circuit moves one step left, i.e from $x = 1, A = 1$ to $x = 1, A = 0$ position because x, the feedback input takes finite time, τ to react. At this position $X = 0$. Therefore after time τ, x becomes 0, i.e. we move up by one position in Karnaugh map to $x = 0, A = 0$ position. Here, $X = 0$ and thus $x = X$ and as long as A does not change the circuit remains in this position, a stable one. Thus, we find $x = 1, A = 0$ position is unstable.

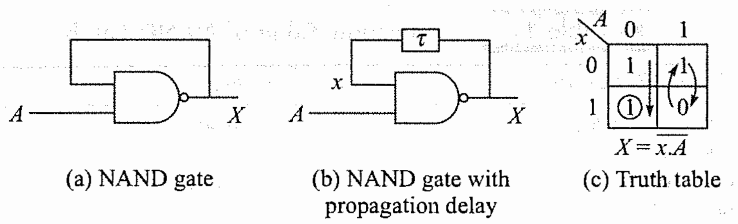
We make an important observation from this discussion which is universally true for asynchronous sequential circuit. *For any state, if $x = X$ then the circuit is stable and if $x \neq X$ it is unstable.*

NAND Gate

We extend the above observation to feedback NAND circuit shown in Fig. 11.20(a) and arrive at the Truth Table given in Fig. 11.20(c). It is interesting to note that for $A = 1$ there is no stable state and $x = X'$ for both $x = 0$ and $x = 1$. Thus there is oscillation between $x = 0, A = 1$ and $x = 1, A = 1$ state.

Two Input NAND Latch

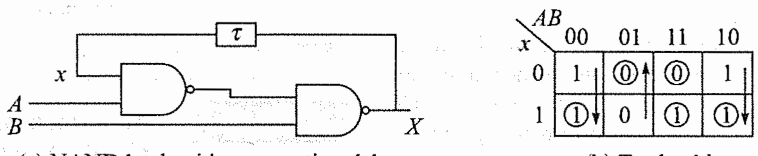
In analysis of asynchronous sequential circuit there is an important constraint to be followed. Though there can be more than one input feeding the circuit, *at a time only one input variable can change*. The other input



(a) NAND gate (b) NAND gate with propagation delay (c) Truth table

Fig. 11.20 Two input NAND gate with output feedback

can change only when the circuit is stabilized following the previous input change. The time required to stabilize the circuit is in the order of propagation delay of a gate, i.e. in nanosecond order. Similarly, if there are two or more output variables *only one output variable can change at any time instant,* as propagation delays in different paths are different. While analyzing the NAND latch given in Fig.11.21a we shall keep this in mind.



(a) NAND latch with propagation delay (b) Truth table

Fig. 11.21 Two input NAND latch

The stable and unstable states are arrived at (Fig.11.21b) following discussion in preceding section, i.e. for any given combination of x, A, B if, $X = x$, the circuit is stable otherwise not. Stables states are encircled and arrows show the movements from transient states. Now let us see how input changes affect the output. For each input combination the circuit has at least one stable state and this stable state will be the starting point of our discussion in each case.

Input AB Change from 00 to 01 The circuit moves from xAB = 100, a stable position to xAB = 101 (Note, x takes a time τ to react to a new set of input) which is unstable and then moves to xAB = 001, a stable state that has output 0. Therefore, a 00→01 transition in AB has output X making 1→0 transition.

Input AB Changes from 00 to 10 The circuit moves from xAB = 100, a stable position to xAB = 110, another stable state that has output 1. Therefore, a 00→10 transition in AB does not alter the value of output, X = 1.

Note that AB cannot change from 00 to 11 as there will be a finite delay, however small it may be between A and B change. Thus, the transition path of AB is either 00→01→11 (then output changes as 1→0→0) or 00→10→11 (output changes as 1→1→1) depending on which of A or B changes earlier. Therefore, output is 0 or 1 depending on intermediate value and in asynchronous logic design such transitions are not allowed.

Following this procedure, we look at other possible transitions of state (xAB) for input change and get transition Table 11.5. Note that, at AB = 11, there are two stable states xAB = 011 and xAB = 111. Transition of AB, 01→11 reaches xAB = 011 state while 10→11 reaches xAB = 111. Thus looking at output of the circuit when AB = 11 (also called idle input that does not force change) one can tell whether AB = 01 or 10 before AB becomes 11. Thus at AB = 11, the circuit generates output $x = X$ from memory or it has latched the

Table 11.5 **Transition Table of NAND Latch**

Input AB	*State(xAB) transition*		*Output X*		*Remark*
00→01	100→101→001		1→0→0		At $AB = 00$,
00→10	100→110		1→1		stable $x = 1$,
01→00	001→000→100		0→1→1		At $AB = 01$,
01→11	001→011		0→0		stable $x = 0$,
10→00	110→100,		1→1		At $AB = 10$,
10→11	110→111		1→1		stable $x = 1$,
11→01	011→001,	111→101→001	0→0,	1→0→0	At $AB = 11$,
11→10	011→010→101,	111→110	0→1→1,	1→1	stable $x = 0, 1$.

before AB becomes 11. Thus at $AB = 11$, the circuit generates output $x = X$ from memory or it has latched the information of previous input combination.

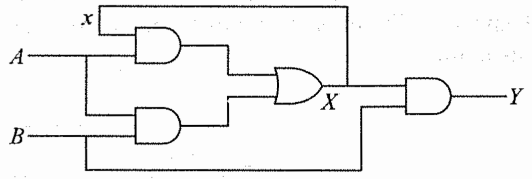


Fig. 11.22 **An asynchronous sequential circuit: Mealy model**

Example 11.6 (i) Analyze the Mealy model asynchronous sequential circuit of Fig. 11.22 and show its stable state and corresponding outputs. (ii) Give the state diagram of this circuit.

Solution To analyze the circuit we consider $x = X(t - \tau)$ where τ is the cumulative propagation delay from input side up to X. For all possible combinations of xAB we get X and Y following logic relation shown in the circuit and prepare Karnaugh map of Fig. 11.23a. States where $X = x$ are stable and encircled. Outputs corresponding to each state and input combination are shown beside. (ii) Since, there are two stable states $x = 0$ and $x = 1$ the state diagram can be drawn from Table 11.5 by considering all possible input combinations for each state as shown in Fig. 11.23b. Note that the output is dependent on inputs as well as state and is shown along the transition path beside the input.

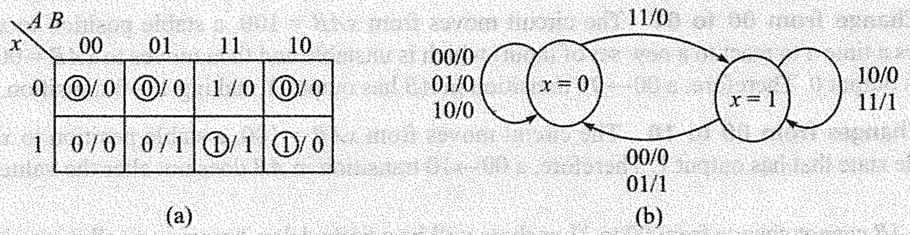


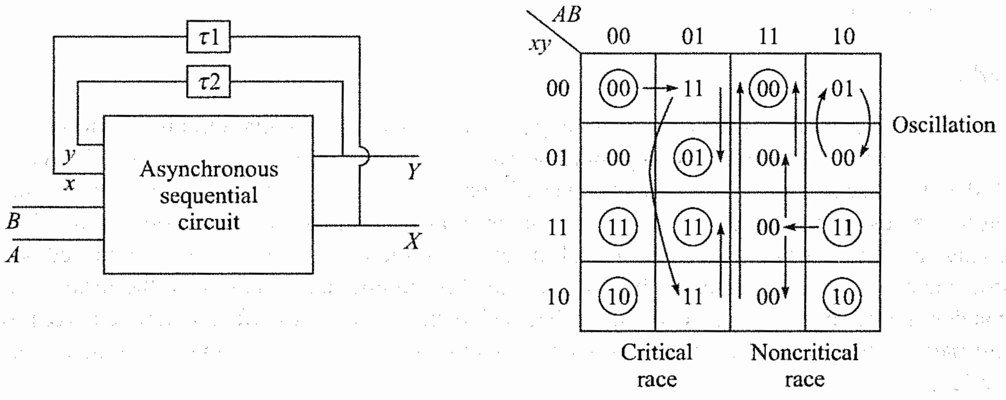
Fig. 11.23 **(a) Karnaugh map, (b) State diagram for asynchronous circuit shown in Fig. 11.22**

SELF-TEST

12. What is fundamental mode of operation of asynchronous sequential circuit?
13. If there are more than one input to such a circuit what constraint is imposed on them?

11.9 PROBLEMS WITH ASYNCHRONOUS SEQUENTIAL CIRCUITS

Before we go for design of asynchronous sequential circuit we would like to look into some important design related issues. These are non-issues in synchronous circuit where external clock trigger arrives after all the inputs are stabilized. Asynchronous circuit responds to all the transient values and problems like oscillation, critical race, hazards can cause major problem unless they are addressed at design stage. To explain these problems we take help of Truth Table shown in Fig.11.24 where the circuit has two external inputs A, B and two outputs X, Y. Both the outputs are fed back to the input side in the form of x and y but with different propagation delays. Thus x, y cannot change simultaneously but with time delays $\tau 1$ and $\tau 2$ respectively and we can write $x = X(t-\tau 1)$ and $y = Y(t-\tau 2)$.



(a) Block diagram (b) Truth table

Fig. 11.24 **(a) Block diagram, (b) Truth table of a 2-input, 2-output circuit**

Refer to the discussion in Section 11.8. The stable states are encircled in the circuit where $xy = XY$. But there are certain major problems with this truth table which we discuss in a future section.

Oscillation

Consider, the stable state $xyAB = 0000$, where $x = X$ and $y = Y$. If input AB changes from 00 to 10, the circuit goes to $xyAB = 0010$ state and then output $XY = 01$. This is a transient state because $xy \neq XY$. After time $\tau 2$, y takes the value of $Y = 1$ and the circuit goes to $xyAB = 0110$ where output $XY = 00$. This again is a transient state and after another propagation delay of $\tau 2$, the circuit goes to $xyAB = 0010$. Thus the circuit oscillates between state 0010 and 0110 and the output Y oscillates between 0 and 1 with a time gap $\tau 2$. In asynchronous sequential circuits for any given input, transitions between two unstable states like these are to be avoided to remove oscillation.

Critical Race

Next we discuss race condition that could be a major problem in asynchronous sequential circuit. This occurs when an input change tries to modify more than one output. In the truth table of Fig.11.24b, consider the stable state $xyAB = 0000$. Now, if AB changes to 01 the circuit moves to $xyAB = 0001$ where $XY = 11$. Now depending which of $\tau1$ and $\tau2$ is lower, xy moves from 00 to either 01 or 10. If $\tau1$ is lower, x changes earlier and the circuit goes to $xyAB$ = 1001 which is a unstable state with output XY = 11 and $xy \neq XY$. The circuit next moves to state $xyAB$ = 1101 which is a stable state and final output XY = 11. If $\tau2$ is lower, y changes earlier and the circuit goes to $xyAB$ = 0101, a stable state and the final output is 01. Thus, depending on propagation delays in feedback path, the circuit settles at two different states generating two different set of outputs. Such a situation is called critical race condition and is to be avoided in asynchronous sequential circuit.

Race can be non-critical too, in which case its presence does not pose any problem for the circuit behavior. In the truth table, consider stable state $xyAB$ = 1110. If input AB changes to 11, the circuit goes to $xyAB$ = 1111 where output XY = 00. Note that both the output variables are supposed to change which cannot happen. Again depending on propagation delays xy becomes either 01 or 10. If xy = 01 then the circuit moves to $xyAB$ = 0111 and then to 0011 and settles there. If xy = 10 then the transition path is 1111→1011→0011. In both the cases final state is 0011 and output is 00. Since, the race condition does not lead to two different state it is termed as non-critical race.

Hazards

Static and dynamic hazards causes malfunctioning of asynchronous sequential circuit. Situations like $Y = A + A'$ or $Y = AA'$ are to be avoided for any input output combination with the help of hazard covers in truth table. A detailed discussion on how to avoid hazard appears in Section 3.9. In circuit with feedback even when these hazards are adequately covered there can be another problem called *essential hazard.* This occurs when change in input does not reach one part of the circuit while from other part one output fed back to the input side becomes available. Essential hazard is avoided by adding delay, may be in the form of additional gates that does not change the logic level, in the feedback path. This ensures effect of input change propagates to the all parts of the circuit and then only feed back output, generated from that input-change makes its presence felt.

Example 11.7 In an asynchronous sequential circuit, the state variable outputs of X and Y are related with primary inputs A and B and its own feedback x and y as shown in Karnaugh map of Fig. 11.25. Can the circuit face any problem in its operation?

Solution Yes, the circuit may face problem in its operation. When the circuit is at stable state $xyAB$ = 1111 and input AB changes from 11→10 the circuit oscillates between $xyAB$ = 1110 and $xyAB$ = 1010. Also there can be a critical race problem if at stable state $xyAB$ = 0001, input AB change from 01 to 00. The circuit may settle at $xyAB$ = 0100 or $xyAB$ = 1000 depending on which of x and y changes first at the feedback path. Non-critical race situation occurs if at stable state $xyAB$ = 0010 the input AB change from 10 to 00.

xy \ AB	00	01	11	10
00	11	$\textcircled{00}$	11	$\textcircled{00}$
01	$\textcircled{01}$	11	11	$\textcircled{01}$
11	10	11	$\textcircled{11}$	10
10	$\textcircled{10}$	$\textcircled{10}$	11	11

Fig. 11.25 **Karnaugh map for Example 11.7**

SELF-TEST

14. What is racing? What is the difference between critical and non-critical race?
15. What is essential hazard?

11.10 DESIGN OF ASYNCHRONOUS SEQUENTIAL CIRCUIT

The discussions in previous sections show several design constraints for asynchronous sequential circuit. This makes the design of such circuit complex and cumbersome and if benefits like speed are not of critical importance, synchronous design is preferred to asynchronous design. In this section we explain the design steps of asynchronous sequential circuit through an example. The problem we attempt to solve is described next.

The Problem

A digital logic circuit is to be designed that has two inputs A, B and one output X. X goes high if at $A = 1$, B makes a transition $1 \rightarrow 0$. X remains high as long as this $A = 1, B = 0$ are maintained. If any of A or B changes at this time output X goes low. It becomes high again when at $A = 1$, B goes from 1 to 0. The timing diagram corresponding to this problem is shown in Fig. 11.26.

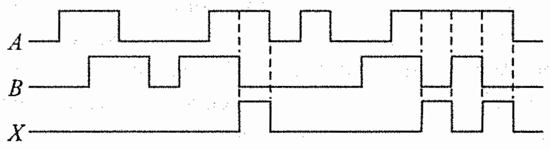


Fig. 11.26 Timing diagram of the problem

State Transition Diagram

From the problem statement we first develop a state transition diagram, say using Moore model. The state symbol and output at that state is shown together within a circle in this diagram (Fig. 11.27). Let the initial state be considered as a when $AB = 00$ with output 0. As long as AB remains 00, the circuit remains at a. Note that in synchronous sequential circuit between two clock trigger input might change but the state of the circuit remains same. Here, as soon as one of A or B changes the circuit may immediately move to different states. Note that A and B cannot change together, a constraint we have to adhere to in asynchronous design. At state a, if input $AB = 01$, the circuit goes to state b and if input $AB = 10$ it goes to c ($00 \rightarrow 11$ prohibited). Both b and c generate output 0 as they have not yet fulfilled the condition stated in the problem for assertion of output. The circuit remains at b for $AB = 01$. If AB changes to 11, the circuit moves to state d with output $X = 0$ and if AB becomes 00 the circuit goes back to a. Similarly the circuit stays at c if input stays at 10 but goes to d receiving 11 and to a receiving 00. Note that at state d, the input $AB = 11$ and now if $B \rightarrow 0$ then condition for output $X = 1$ is fulfilled and next state e for $AB = 10$ shows output as 1. The circuit remains at state e as long as $AB = 10$. It goes back to state d if AB becomes 11 because if B again goes to 0 output should be high. However at e, if AB changes to 00 the circuit goes to initial state a as AB becoming 10 following 00 will not assert output.

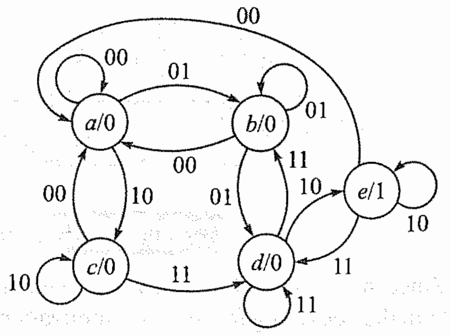


Fig. 11.27 State transition diagram of the problem

Primitive Table

The next step is to form state table from state transition diagram. In this table if all the rows representing a state has only one stable state for all possible input combinations and it is termed as *primitive table*, or *primitive flow table* or simply *flow table*. Often we can skip step one and directly go to primitive table from problem statement. Primitive table prepared from state transition diagram is shown in table Fig. 11.28.

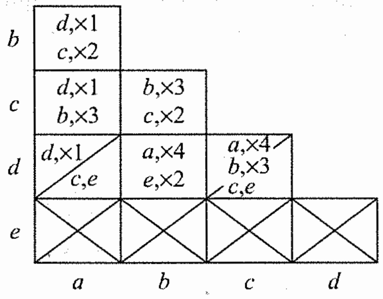
Note that each row in this table has one don't care state. The don't care state in each row comes which asks for both the input variables to change to move from stable state, a condition not allowed in asynchronous sequential logic. The don't care states have been given suffix like 1, 2 which is not a must. However, this helps in next step where we check state redundancy.

	AB 00	01	11	10	*X*
a	$\textcircled{a}$	*b*	×1	*c*	0
b	*a*	$\textcircled{b}$	*d*	×2	0
c	*a*	×3	*d*	$\textcircled{c}$	0
d	×4	*b*	$\textcircled{d}$	*e*	0
e	*a*	×5	*d*	$\textcircled{e}$	1

Fig. 11.28 Primitive table for the problem

State Reduction

It is always useful to check state redundancy before going for actual circuit design. Removing redundant states helps in generating the circuit in a simpler way and with less hardware. We use implication table for this example to remove redundant state, if there is any. The implication table, drawn from primitive table and state reduction is shown in Fig. 11.29. For preparation of implication table refer to discussion in Section 11.7. Note that in asynchronous design, when there are don't care states in state table, this is called incompletely specified table. For this, state reduction can be done as follows.



b	*d*,×1 *c*,×2			
c	*d*,×1 *b*,×3	*b*,×3 *c*,×2		
d	$\cancel{d,\times 1\ c,e}$	*a*,×4 *e*,×2	$\cancel{a,\times 4\ b,\times 3\ c,e}$	
e	╳	╳	╳	╳
	a	*b*	*c*	*d*

d : *d*
c : *cd*
b : *cd* (*bc*) ≡ *d* (*bc*) ×2 ≡ *c*, ×3 ≡ *b*
a : *d* (*bc*) (*ab*) (*ac*) ≡ (*abc*) ×1 ≡ *d*
P = (*abc*) (*d*) (*e*)

Fig. 11.29 State reduction by implication table

Since state *e* cannot be equivalent with any of *a*, *b*, *c* and *d* (output being different) we put double cross right in the beginning for row *e* of implication table. Next we find *a* and *d* cannot be equivalent as that requires *c* and *e* to be equivalent which is not. Similarly, *c* and *d* cannot be equivalent and we cross these two places in row *d* of implication table with single line. Now let us try to find equivalence by moving along columns. In column *d* and *c* there is no equivalence possible. In column *b*, equivalence between either of *b* and *c* or *b* and *d* is possible. But, both (*bc* and *bd*) equivalences are not possible as it requires don't care state ×2 to be made equivalent to *c* and *e* while *ce* themselves are not equivalent. In column *a*, we see (*c*, ×2) equivalence may make *a* and *b* equivalent. Therefore, from column *b*, we get (*bc*) equivalence by making *c*, ×2 and *b*, ×3 equivalent. In column *a*, by assigning ×1 to *d* we can make (*ab*) and (*ac*) equivalent and there is no conflict with assignment of don't care states. Since, (*bc*) (*ab*) (*ac*) ≡ (*abc*) partition table has three different groups (*abc*), (*d*) and (*e*). Thus the states are reduced to 3 from original 5. Let state *a* represent the group (*abc*). Now the reduced state table is as shown in Fig. 11.30a and reduced state transition diagram in Fig. 11.30b.

State Assignment

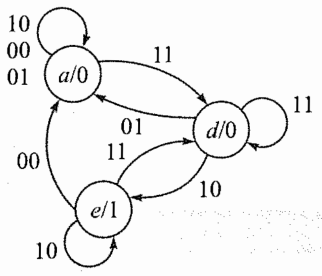
This step in asynchronous sequential circuit design has to be done very carefully so that a valid state transition does not require two or more output variables to change simultaneously which may lead to racing problem. In this problem there are three states in the reduced state diagram which needs two variables to represent them. Figure 11.30 shows that we cannot avoid two variables changing together in one or more occasions for the reduced state transition diagram. If {*a,d,e*} is represented by {00,01,10} it occurs twice for $d \rightarrow e$ and $e \rightarrow d$ transitions in state transition diagram. A representation of {00,01,11} requires two variables to change only once when $e \rightarrow a$ transition occurs. The solution to this may be found if the unused fourth combination of two variable representation is used as a dummy state, say ϕ. We include ϕ between *e* and *a*. Note that, if one dummy state is not enough we may need to use a third variable to represent the states that will make $2^3-3 = 5$ dummy variable available for this purpose. Let us represent the states in this problem by two variables *PQ* in the following way

a: 00 *d*: 01 *e*: 11 ϕ: 10

The modified state diagram and state table with dummy variable ϕ =10 included are shown in Fig. 11.31. Note that ϕ is an unstable state and before the input can change it goes to next stable state *a*. We represent state variables by *P* and *Q*, the corresponding feedback variables are represented by *p* and *q* respectively.

	AB				
	00	01	11	10	*X*
a	*a*	*a*	*d*	*a*	0
d	×4	*a*	*d*	*e*	0
e	*a*	×5	*d*	*e*	1

(a)



(b)

Fig. 11.30 Reduced (a) State table, (b) State transition diagram

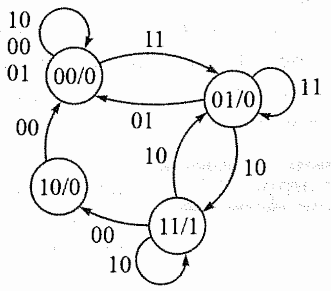


Fig. 11.31 Modified state transition diagram

Design Equations and Circuit Diagram

We use Karnaugh map to get expression of state variables *P* and *Q* as a function of input *A,B* and feedback variables *p* and *q*. The equations derived from Karnaugh map are shown in Fig. 11.32. The equation of output *X* is generated from *P* and *Q* as we use Moore model. The final circuit is developed from these equations and is shown in Fig. 11.33.

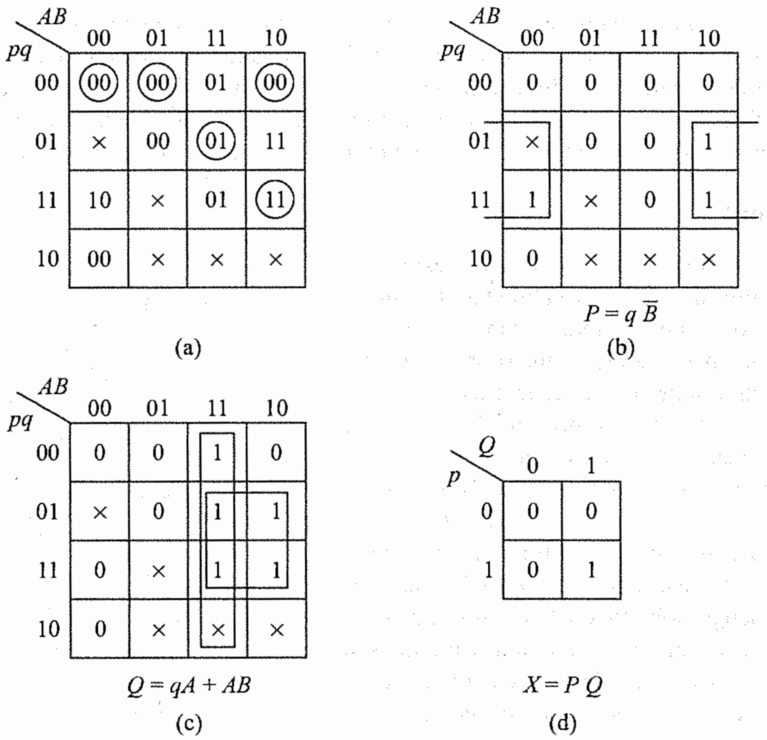


Fig. 11.32 (a) Reduced state diagram from Fig. 11.27, (b)–(d) Karnaugh map and design equations

It is left to the reader to analyze this circuit and verify the timing diagram shown along with the problem statement. Now, that we have seen all the steps in asynchronous sequential logic design we are in a position to appreciate how complex the process is compared to synchronous sequential logic design. Thus the later is preferred if issues like speed, clock skew, etc. are not of critical importance.

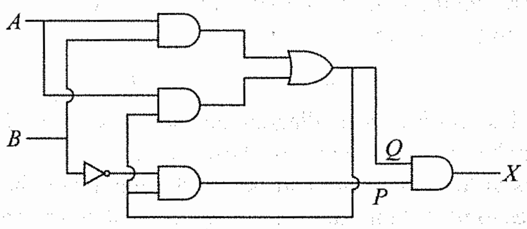


Fig. 11.33 Circuit diagram of asynchronous sequential logic for the problem in Fig. 11.23

Example 11.8 Design an asynchronous sequential logic circuit for state transition diagram shown in Fig. 11.34.

Solution Let the two input variables be termed A and B in order. Figure 11.35a shows state table through Karnaugh map. Since the state transition diagram has two states we need one ($\log_2 2$) output feedback serving as memory. Let the output variable be termed X and its feedback x. If we represent current state a as $x = 0$ and b as $x = 1$ then output

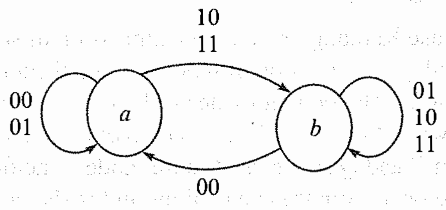


Fig. 11.34 State transition diagram for Example 11.8

X can be expressed as shown in Fig. 11.35b. The asynchronous sequential logic circuit drawn from design equation is shown in Fig. 11.35c.

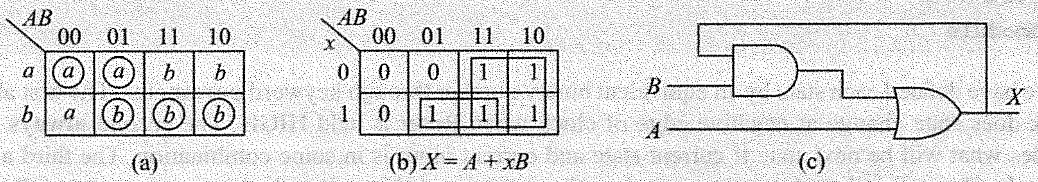


Fig. 11.35 **Solution for Example 11.8**

SELF-TEST

16. What is a primitive flow table?
17. What is an incompletely specified table?
18. What is a dummy variable?
19. What is the advantage and disadvantage of asynchronous over synchronous sequential circuits?

11.11 FSM IMPLEMENTATION IN HDL

We shall conclude our discussion on HDL showing how one can represent a Finite State Machine (FSM) in HDL. We take up the Mealy Model shown in Fig. 11.2b as illustration and the code is given next. Note that, we have introduced two variables Clock and Reset, which is not explicit in the figure. The clock is used for synchronous state transition of the circuit (at negative edge) and active low asynchronous Reset is used to initialize the circuit to state *a*.

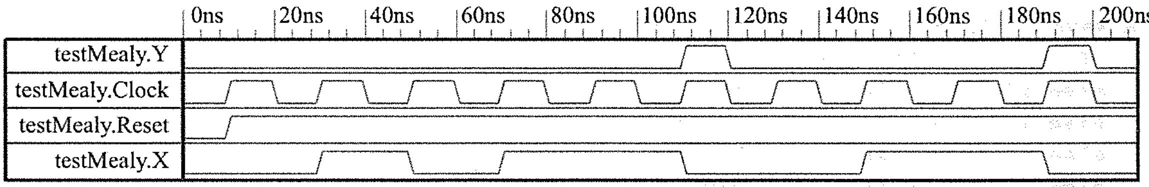
```
module MealyFSM(X, Clock,Reset,Y);
input X,Clock,Reset;
output  Y;
reg Y;
reg [1:0] PS,NS; //PS represents Present State, NS Next State
parameter a=2'b00, b=2'b01, c=2'b10;  //States shown in fig. given binary value
always @ (negedge Clock or negedge Reset) //Reset or state change
  if (~Reset) PS=a;
  else PS=NS;
always @ (PS or  X) //Determines next state
  if (PS==a && X==0) NS=a;
  else if (PS==a && X==1)  NS=b;
  else if (PS==b && X==0)  NS=a;
  else if (PS==b && X==1)  NS=c;
  else if (PS==c && X==0)  NS=a;
  else if (PS==c && X==1)  NS=c;
```

```
always @ (PS or  X)  //Determines output which is dependent both on present
   if (PS==c && X==0) Y=1; //state and present input as Mealy Model
   else Y=0;
Endmodule
```

We have defined each state by an equivalent binary number through keyword **parameter**. The first **always** block does state change at negative edge of clock when Reset is held HIGH. The second **always** block decides what will be next state if current state and current input is in some combination. The third always block decides output based on current state and current input. Note that all these assignments follow the conditions stated in FSM Mealy model of Fig. 11.2b.

Now let us try to test this circuit by feeding an input *X* = 01011001101(first value of *X* fed is 0 as if data is coming from right) by creating a test bench and appending it to above code. Let us verify whether the circuit can detect pattern '110' (as data is considered to come from right) and generate appropriate output. The following test bench can generate such pattern and the output is plotted against clock and input *X* in the subsequent timing diagram, obtained from Verilog simulation.

```
module testMealy();
reg Clock, Reset, X;
wire Y;
initial
begin  Reset = 0; //Initial value of reset=0, this resets the FSM and PS=a.
X=0;  // input is 0 at start
#10 Reset = 1;
#20 X=1;  #20 X=0;  #20 X=1;  #20 X=1;  //Change in input data at
#20 X=0;  #20 X=0;  #20 X=1;  #20 X=1;  //odd multiple of 10 ns
#20 X=0;  #20 X=1;                        //starting from 30 ns.
end
// Clock generator follows
initial
    begin
   Clock = 1'b0;
   repeat (21)
   #10 Clock = ~Clock; //Clock inverts at every 10ns so that
    end      //negative edge of clock comes at even multiple of 10 ns
MealyFSM MFSM(X,Clock,Reset,Y);
endmodule
```



Find from the timing diagram $X = 0$ up to 30 ns from start and remains 1 till 50 ns and so on. Clock negative edge comes at 20 ns, 40 ns etc. When X remains 1 at two negative edges of clock and a 0 follows like in between 110–120 ns $Y = 1$ and similarly between 190–200 ns. $Y = 0$ elsewhere and this is what the FSM is supposed to perform, i.e. detect '110' (data from right, if from left '011') that is input bit 1, followed by another 1 followed by 0.

Example 11.9 Give Verilog HDL description of the Moore Model shown in Fig. 11.2a

Solution

```
module MooreFSM(X, Clock,Reset,Y);
input X,Clock,Reset;
output  Y; reg Y;
reg [1:0] PS,NS; //PS represents Present State, NS Next State
parameter a=2'b00, b=2'b01, c=2'b10, d=2'b11; //Four states given binary value
always @ (negedge Clock or negedge Reset) //Reset or state change
   if (~Reset) PS=a;
   else PS=NS;
always @ (PS or  X) //Determines next state
   if (PS==a && X==0) NS=a;
   else if (PS==a && X==1)  NS=b;
   else if (PS==b && X==0)  NS=a;
   else if (PS==b && X==1)  NS=c;
   else if (PS==c && X==0)  NS=d;
   else if (PS==c && X==1)  NS=c;
   else if (PS==d && X==0)  NS=a;
   else if (PS==d && X==1)  NS=b;
always @ (PS)  //Determines output which is dependent only  on
   if (PS==d) Y=1; //present state due to Moore Mealy Model
   else Y=0;
endmodule
```

We conclude our discussion on HDL here. The objective had been to make one get started with basics of HDL design. Dedicated books and courses deal with this subject in greater details. One should note that free or student version of Verilog compliler has limited ability and trial full versions are free only for the trial period. Also the hardware device on which the design is exported is not cheap. It is thus not useful for simple design problem except for finding functional error through simulation. But it definitely is cheaper and convenient if one considers a large complex design problem. The hardware devices commonly used to load HDL codes are discussed in Section 13.6 of Chapter Memory.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem A part of the simplistic digital control unit of a hypothetical Automatic Teller Machine (ATM) works like this. The ATM senses ATM card insertion by assertion of an input I. A correct typing of Personal Identification Number (PIN) is sensed by P. Transaction is done by asserting

TR and card return by *CR* is if $P = 1$. If the process is cancelled by pressing push button 'Return' that asserts *R*, transaction does not take place but card is returned. If not cancelled, the user gets two more opportunities to enter PIN. But third incorrect entry locks the card by asserting *CL*.

Solution The input and output variables assertions are as follows.

Input:

Card inserted $I = 1$, Card not inserted $I = 0$.

'Return' button pressed $R = 1$, 'Return' button not pressed $R = 0$.

PIN correctly entered $P = 1$, PIN incorrectly entered $P = 0$.

Output:

Card to be returned $CR = 1$, Card not to be returned $CR = 0$.

Transaction allowed $TR = 1$, Transaction not allowed $TR = 0$.

Card to be locked $CL = 1$, Card not to be locked $CL = 0$.

In Method-1, we make use of Moore Model and ASM chart which is shown in Fig. 11.36.

We find there are six rectangular boxes or six states (*a* to *f*). Thus, it requires three flip-flops which can handle up to eight states. The three inputs and three present states of flip-flops would require total $3 + 3 = 6$ variable Karnaugh Map for design. The ROM-Delay flip-flop approach may thus be preferred.

Let the state assignments of three flip-flops, say *C*, *B*, and *A* be as follows

$a : CBA = 000,\quad b : CBA = 001,\quad c : CBA = 010,$

$d : CBA = 011,\quad e : CBA = 100,\quad f : CBA = 101.$

Since, it is a Moore Model, the output corresponding to each state are

$a : CR = 0, TR = 0, CL = 0 \quad b : CR = 0, TR = 0, CL = 0$

$c : CR = 0, TR = 0, CL = 0 \quad d : CR = 0, TR = 0, CL = 1$

$e : CR = 1, TR = 0, CL = 0 \quad f : CR = 1, TR = 1, CL = 0$

Then the state transition table can be as given in Fig. 11.37.

Three present states and three inputs call for a 6 to 64 decoder. Three next states require a storage of three bits in each ROM address. Thus this implementation will require 3 flip-flops, one 6 to 64 decoder and one 64×3 ROM.

The outputs are derived from present state, i.e. flip-flops in a Moore Model. The combinatorial circuit required for this is arrived from Karnaugh Maps presented in Fig. 11.38.

The final realization is shown in Fig. 11.39.

Note that the requirement for decoder, ROM, etc. can be reduced noting that not all 64 combinations are required for the solution, e.g., if $I = 0$, no matter what the other values are, the circuit remains in state *a*.

In Method-2, this is ASM chart based approach targeting a Mealy Model. The output is asserted as soon as the condition is fulfilled. Figure 11.40 shows the ASM chart.

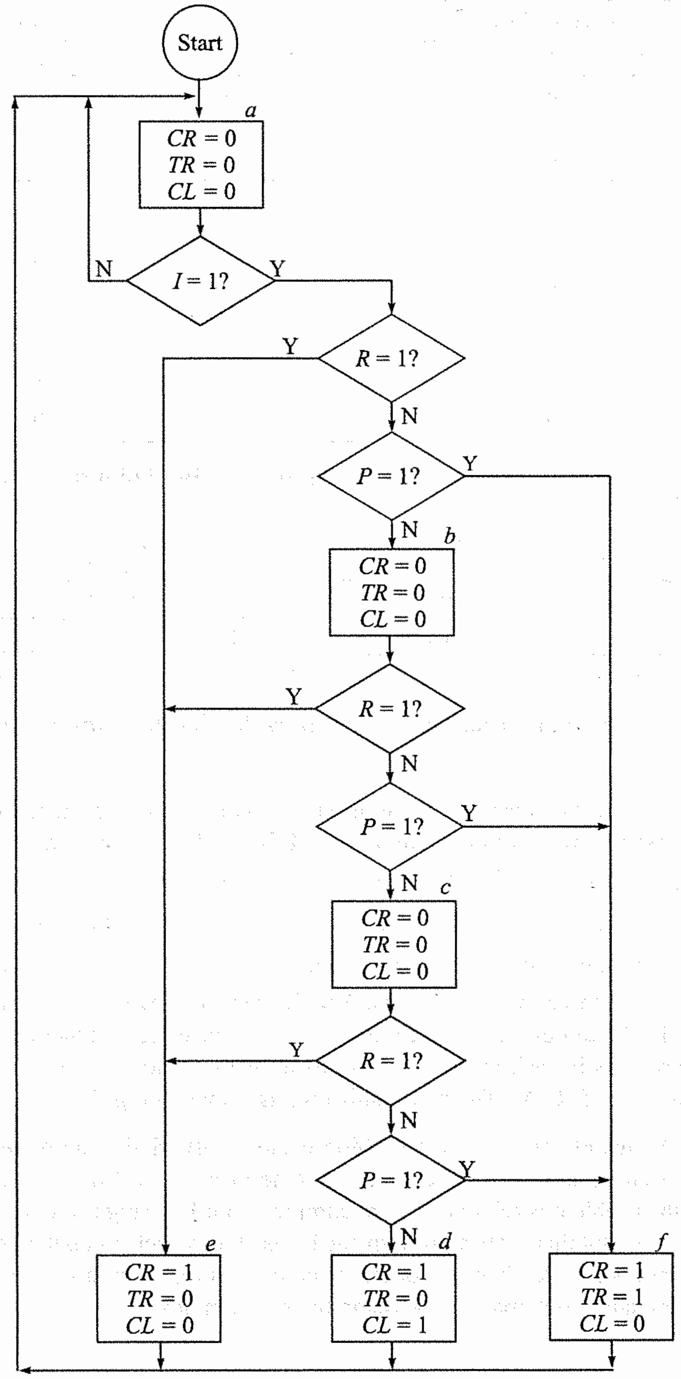


Fig. 11.36 ASM chart for Moore Model: Solution using Method-1

	Present State			Input			Next State		
	C_n	B_n	A_n	I	R	P	C_{n+1}	B_{n+1}	A_{n+1}
a :	0	0	0	0	X	X	0	0	0
a :	0	0	0	1	0	0	0	0	1
b :	0	0	1	1	1	X	1	0	0
b :	0	0	1	1	0	1	1	0	1
b :	0	0	1	1	0	0	0	1	0
c :	0	1	0	1	1	X	1	0	0
c :	0	1	0	1	0	1	1	0	1
c :	0	1	0	1	0	0	0	1	1
d :	0	1	1	X	X	X	0	0	0
e :	1	0	0	X	X	X	0	0	0
f :	1	0	1	X	X	X	0	0	0

Fig. 11.37 State Transtion Table for Moore Model: Solution using Method-1

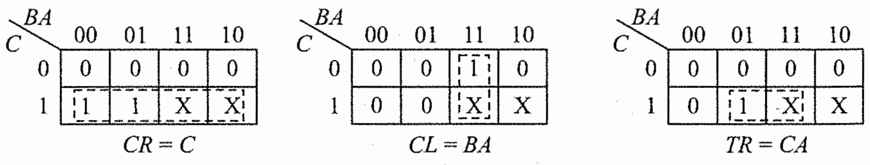


Fig. 11.38 Combinatorial Logic for Moore Model: Solution using Method-1

We find there are three rectangular boxes or three states (*a* to *c*). Thus it requires two flip-flops which can handle up to four states. We continue with the ROM-Delay flip-flop approach.

Let the state assignments of two flip-flops, say *B* and *A* be as follows

a: $BA = 00$, $\qquad b$: $BA = 01$, $\qquad c$: $BA = 10$

Then the state transition table can be as given in Fig. 11.41.

Two present states and three inputs call for a 5 to 32 decoder. Two next states require storage of two bits and in each ROM address. The three outputs can also be directly generated from ROM which require additional three bits in each location. Thus this implementation will require 2 flip-flops, one 5 to 32 decoder and one 32 × 5 ROM. The final realization is shown in Fig. 11.42.

Other Methods: As already mentioned, both Moore and Mealy Model can be realized by flip-flops and combinatorial circuits, as done for vending machine problem in Fig. 11.11. But this becomes cumbersome when the problem is relatively more complex with Karnaugh Map requiring solution for more than four variables. But then, QM algorithm can be used for which computer code also exists. The state transition diagram is avoided here as for larger number of input variables, every node will have too many branches spreading out, making the diagram very complex.

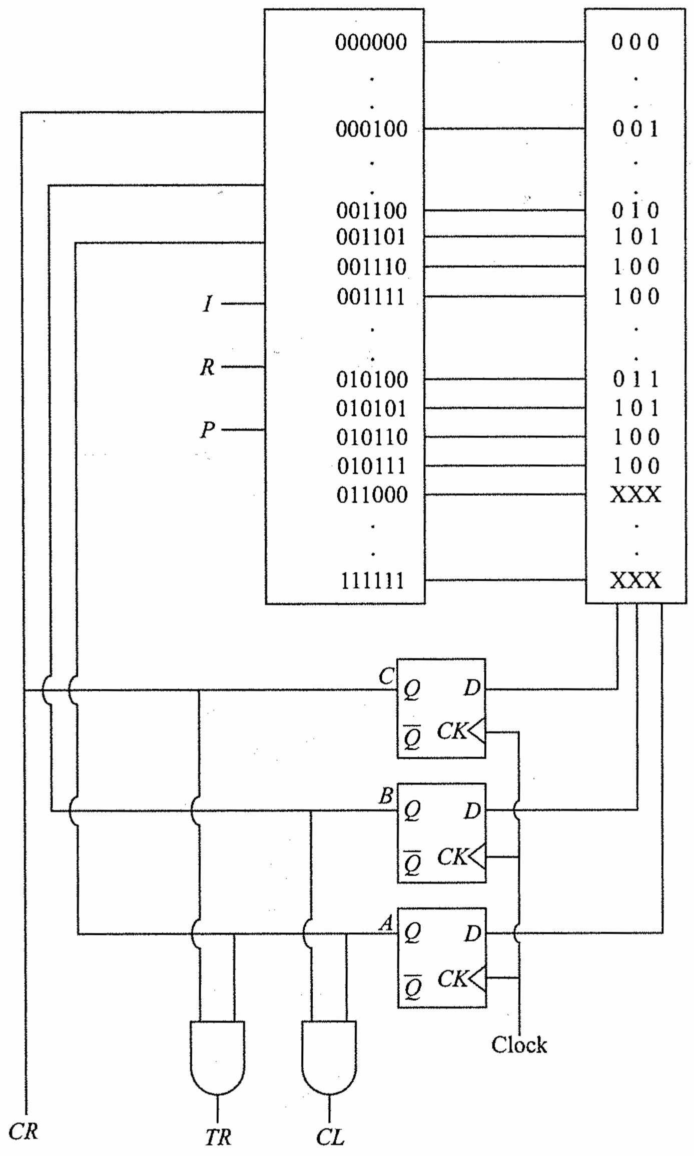


Fig. 11.39 Realization using ROM: Solution using Method-1

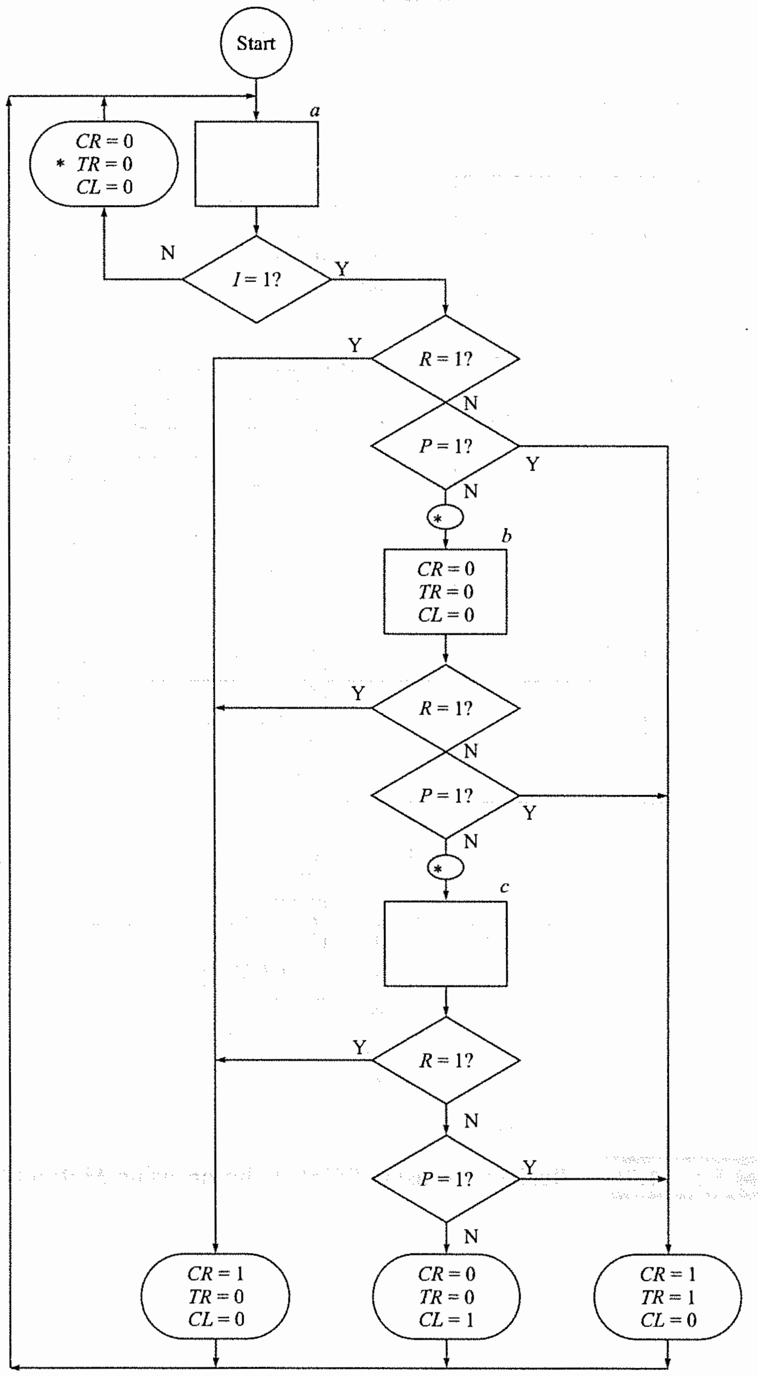


Fig. 11.40 **ASM chart for Mealy Model: Solution using Method-2, *represents *CR* = 0, *TR* = 0, *CL* = 0**

Present State			Input			Next State		Output		
	B_n	A_n	I	R	P	B_{n+1}	A_{n+1}	CR	TR	CL
a :	0	0	0	X	X	0	0	0	0	0
a :	0	0	1	0	0	0	1	0	0	0
a :	0	0	1	0	1	0	0	1	1	0
a :	0	0	1	1	X	0	0	1	0	0
b :	0	1	1	0	0	1	0	0	0	0
b :	0	1	1	0	1	0	0	1	1	0
b :	0	1	1	1	X	0	0	1	0	0
c :	1	0	1	0	0	0	0	0	0	1
c :	1	0	1	0	1	0	0	1	1	0
c :	1	0	1	1	X	0	0	1	0	0

Fig. 11.41 State Table for Mealy Model: Solution using Method-2

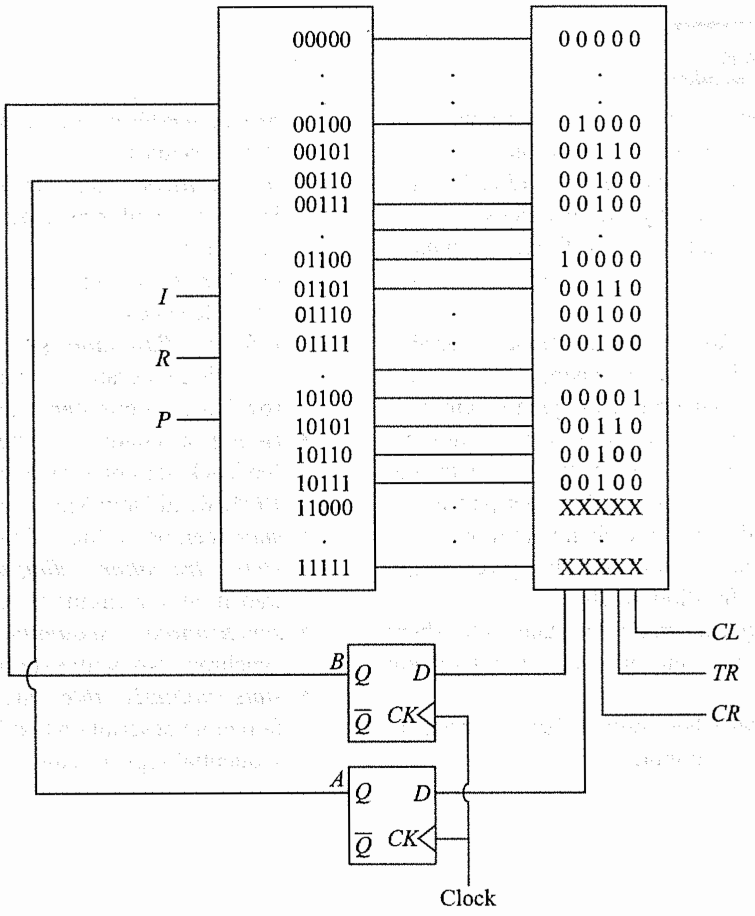


Fig. 11.42 Realization using ROM: Solution using Method-1

SUMMARY

The most popular sequential logic circuit design uses an external clock for triggering and the state changes occur synchronously with clock. In Moore model, such design output is derived directly from state outputs, also called secondary outputs. In Mealy model, output is generated from primary or actual input to circuits and secondary inputs. Word or timing description of a problem is first converted to state transition diagram or ASM chart followed by state synthesis table. In one implementation circuit can be realized using any type of flip-flop using flip-flop excitation table and excitation map. In other ROM and delay flip-flops are used where ROM stores the next state information. State reduction techniques are used to remove redundant states thereby can reduce the circuit complexity. Asynchronous system is not dependent on any external clock thus operates at a higher speed. But the circuit reacts to any and every change in the inputs and are prone to problems like racing, oscillations and different types of hazards. Design of such circuits are much more difficult compared to synchronous circuit and are attempted only for time critical applications where a very fast response to any input change is required.

GLOSSARY

- ***ASM chart*** a flow chart describing state transition with timing information
- ***asynchronous sequential circuit*** not synchronous with any external clock
- ***critical race*** leads to two different outputs of circuit depending on which feedback variable changes earlier
- ***dummy variable*** an additional variable preventing simultaneous change of two state variables in asynchronous sequential circuit
- ***essential hazard*** a condition when following an input change, one feedback variable tries to change the output before the other part of the circuit could respond to change in input.
- ***excitation map*** relationship that gives design equation for flip-flop inputs
- ***incompletely specified table*** state table where some of the next state or output or both remain unspecified.
- ***Mealy model*** where output depends both on state variable and input
- ***Moore model*** where output depends only on state variables
- ***non critical race*** leads to same output irrespective of propagation delay in a race condition
- ***oscillation*** circuit moving between two unstable states
- ***primitive flow table*** that directly maps state transition diagram in a state table where each row has only one stable state.
- ***racing*** a condition when more than one feedback variables try to change its value
- ***ROM*** Read Only Memory
- ***state*** memory values of a sequential circuit
- ***state transition diagram*** depicts state transition of a circuit pictorially
- ***synchronous sequential circuit*** works synchronously with external clock trigger
- ***state synthesis table*** state transition and flip-flop input description that leads to synthesis of sequential logic circuit

PROBLEMS

Section 11.1 and Section 11.2

11.1 Draw state transition diagram of synchronous sequential logic circuit using Mealy model that detects three consecutive zeros from an input data stream, X and signals detection by making output, $Y = 1$.

11.2 Convert Mealy model of Problem 11.1 to Moore model using conversion rules.

11.3 Using Moore model draw state transition diagram of a serial parity checker circuit. If the number of '1's received at input X is even, parity checker output, $Y = 0$. If odd number of '1's are received at X then $Y = 1$.

11.4 Convert Moorem model of Problem 11.3 to a Mealy model.

11.5 Using Moore model draw state transition diagram of the circuit that generates a single pulse of width equal to clock period when enabled by $E = 1$. The circuit is reset by $E = 0$ at any stage.

11.6 Draw state transition diagram of sequence detector circuit that detects '1101' from input data stream using both Mealy and Moore model.

Section 11.3 and Section 11.4

11.7 For Mealy model state transition diagram of sequence detector problem shown in Fig. 11.2b use following state assignment and get corresponding state synthesis table for JK flip-flop based solution.

a: $B = 0, A = 0$ b: $B = 0, A = 1$

c: $B = 1, A = 1$

11.8 For Mealy model state transition diagram of sequence detector Problem shown in Fig. 11.2b use following state assignment and get corresponding state synthesis table for JK flip-flop based solution.

a: $B = 0, A = 0$ b: $B = 0, A = 1$

c: $B = 1, A = 1$ d: $B = 1, A = 0$

11.9 Give design equations for Problem 11.7. Compare this with solution given in Section 11.4.

11.10 Give design equations for Problem 11.8. Compare this with solution given in Section 11.4.

11.11 Give design equations for Problem 11.1 for implementation with D flip-flops.

11.12 Implement circuit diagram for Problem 11.1 using JK flip-flops.

11.13 How many memory elements are necessary for Mealy and Moore models in sequence detector Problem 11.6.

11.14 Implement (a) parity checker circuit of Problem 11.3 and (b) single pulse generator circuit of Problem 11.5.

Section 11.5 to 11.7

11.15 Show how using an additional column in ROM the combinatorial circuit of Fig. 11.7 for sequence detector problem can be dispensed with.

11.16 Implement ROM based solution for Problem 11.6 where output is directly derived from ROM.

11.17 In vending machine problem of Section 11.6 we want to add an additional function. We give the customer an option to get back the coins he has deposited if he finds himself short of money or changes his mind midway. However, this function does not work if the cost of the product is reached. A push button switch, P is used for this which when pressed generates $P = 1$ and returns the coin deposited thus far by activating $C = 1$. Show what changes in ASM chart of Fig. 11.10 are necessary for this.

11.18 Draw ASM chart for Problem 11.5 and implement the circuit using ROM.

11.19 Find the minimum number of states necessary to represent following state table both by row elimination and implication table method.

Table 11.6

Present State	*Next State*		*Present Output*	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
a	*f*	*d*	0	1
b	*c*	*f*	1	1
c	*f*	*b*	1	1
d	*e*	*g*	1	1
e	*a*	*d*	1	1
f	*g*	*b*	0	1
g	*a*	*d*	0	1

11.20 Reduce following state table using implication table method.

Table 11.7

Present State	*Next State*		*Present Output*	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
a	*h*	*c*	1	0
b	*c*	*d*	0	1
c	*h*	*b*	0	0
d	*f*	*h*	0	0
e	*c*	*f*	0	1
f	*f*	*g*	0	0
g	*g*	*c*	1	0
h	*a*	*c*	1	0

Section 11.8

11.21 State the condition of stability in asynchronous sequential logic.

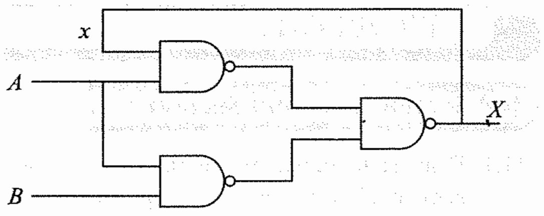
11.22 One of the two inputs of a two input NOR gate is fed back from the output. Write its state stable and encircle stable states, if any.

11.23 For state table in Problem 11.30 show the stable states, if any.

11.24 For state table in Problem 11.30 show how the circuit behaves when $xy = 11$ and A changes as $1 \rightarrow 0$.

11.25 There are three inputs A, B and C to an asynchronous sequential logic system. If $ABC = 111$ at any given time write the allowed combination of inputs that can follow.

11.26 Draw state table of adjacent asynchronous sequential logic circuit.



Section 11.9

11.27 When does oscillation occur in an asynchronous sequential logic circuit?

11.28 How can essential hazard be prevented in asynchronous sequential logic circuit?

11.29 There are two inputs A, B and three feedback outputs x, y and z of an asynchronous sequential logic system. If $xyzAB = 10011$ gives a stable state and input AB changes as $11 \rightarrow 10$, which of the following next state does not give racing problem – 10110, 00110, 11010, 00010 and 11110?

11.30 Find out potential problems in following state table where A is input and x and y are output feedbacks.

$A \backslash xy$	00	01	11	10
0	01	00	10	10
1	00	01	11	01

Section 11.10

11.31 The T flip-flop has a single input T, and single output Q. For $T = 0$, output does not change. For $T = 1$, output complements and remains at that value as long as $T = 1$. Draw its (a) state diagram and (b) primitive flow table.

11.32 For Problem 11.31, use state reduction technique to check if a reduced flow table is possible.

11.33 Find design equations for Problem 11.31 after appropriate state assignment.

11.34 Design a parity generator using asynchronous sequential logic that gives output = 1 when it receives odd number of pulses and output = 0 if the number of pulses received is even. (**Hint:** State transition diagram is same as Problem 11.31.)

11.35 Draw state transition diagram of a modulo-3 counter for asynchronous sequential logic. The counter counts number of pulses appearing at its input and generates output = 1 when three pulses arrive else output = 0.

11.36 Design modulo-3 counter stated in Problem 11.35 using asynchronous sequential logic.

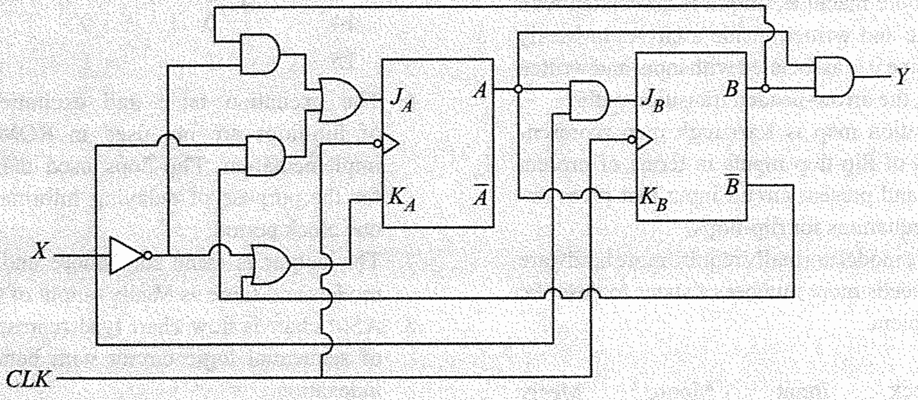
11.37 We require a circuit which will suppress narrow positive spikes on a signal line. The output of the circuit will be an inverted and slightly delayed version of the input minus the spikes. Construct the primitive flow table and show one state assignment scheme.

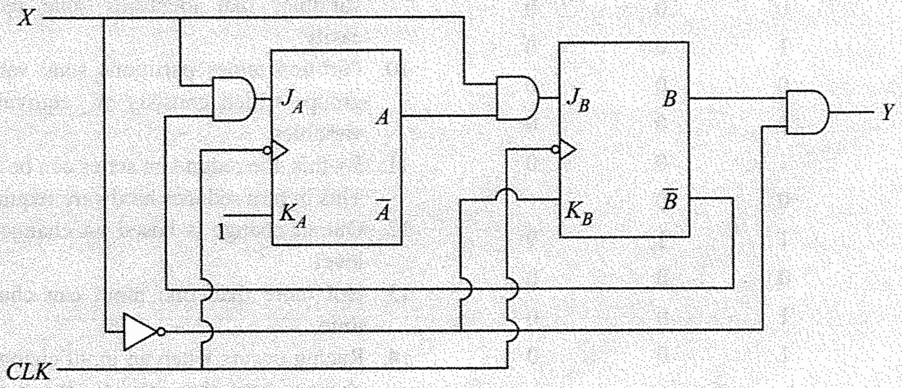
11.38 Get design equations for Problem 11.37 and implement the circuit. Verify how it does noise suppression.

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to implement a Moore Model and a Mealy Model for a sequence detector that detects a sequence '110' from the incoming data stream.

Theory: The Moore Model generates final output solely from flip-flop states while Mealy Model can use input data too. Moore Model, usually takes more hardware but in





Mealy Model, unwanted fluctuations in input get directly reflected at the output. Section 11.4 of the book explains the design of these models and reproduces the circuits to be implemented.

Apparatus: +5V DC Power supply, Multimeter, Bread Board, Clock Generator, and Oscilloscope.

Work element: Connect the circuit as shown. You can manually give input, or a counter can be used to generate a repetitive sequence that contains '110'. Check the output. For manual input, check the performance for both the Models when debounce switch is used and when it is not used. Learn to debug the circuit by verifying outputs of individual building blocks say flip-flops and logic gates for different inputs.

Answers to Self-tests

1. State transition diagram is a visual description of how state of sequential circuit change in each clock cycle.
2. In Moore machine, output is associated with a state and written inside a circle. In Mealy machine it is associated with input and written along the arrow-headed transition path.
3. Excitation map is Karnaugh map representation of flip-flop inputs in terms of present state and present circuit input that gives design equations for flip-flops.
4. Moore model normally require more hardware as it needs more number of states to describe a problem.
5.

Clock Cycle	Input	Moore output	Mealy output
1	0	0	0
2	1	0	0
3	0	0	0
4	1	0	0
5	1	0	0
6	0	0	1
7	1	1	0
8	0	0	0
9	1	0	0
10	1	0	0
11	0	0	1
12	1	1	0
13	1	0	0
14	0	0	1
15	1	1	0

6. The excitation table and excitation map of flip-flops are not used in ROM based implementation. Flip-flops used there only for the purpose of delaying information by one clock period.
7. The output is same for Moore and Mealy models and same as Mealy output of Q. 5.
8. ASM chart is flow chart type representation of sequential logic circuit with better time indexation.
9. Implication table is a mapping of state variables that identifies state redundancy easily.
10. Partition tables partitions state variables in groups which consists of equivalent state variables.
11. By this, the redundant states can be removed. This in turn reduces hardware requirement.
12. Output change is based on change in input level.
13. Not more than one input can change at a time.
14. Racing occurs when an input change tries to change more than one feedback variables.

Depending on which changes earlier the circuit may stabilize in two different states of the circuit resulting critical race. If it stabilizes in same state it is called Non-critical race.

15. This is the condition when following an input change, one feedback variable tries to change the output before the other part of the circuit could respond to change in input.
16. This directly maps state transition diagram in a state table where each row has only one stable state.
17. State table where some of the next state or output or both remain unspecified.
18. An additional variable, which is not required as such but prevents simultaneous change of two state variables in asynchronous sequential circuit.
19. Asynchronous circuit does not depend on clock trigger, hence faster. But there are several practical constraints that makes design of such circuit very complex.

D/A Conversion and A/D Conversion

12

OBJECTIVES

- **Be able to do calculations related to variable resistor and binary ladder networks**
- **Recall some of the sections of a typical D/A converter and calculate D/A resolution**
- **Understand A/D conversion using the simultaneous, counter, continuous, and dual-slope methods**
- **Discuss the accuracy and resolution of A/D converters**

Digital-to-analog (D/A) and analog-to-digital (A/D) conversion form two very important aspects of digital data processing. Digital-to-analog conversion involves translation of digital information into equivalent analog information. As an example, the output of a digital system might be changed to analog form for the purpose of driving a pen recorder. Similarly, an analog signal might be required for the servomotors which drive the cursor arms of a plotter. In this respect, a D/A converter is sometimes considered a decoding device.

The process of changing an analog signal to an equivalent digital signal is accomplished by the use of an A/D converter. For example, an A/D converter is used to change the analog output signals from transducers (measuring temperature, pressure, vibration, etc.) into equivalent digital signals. These signals would then be in a form suitable for entry into a digital system. An A/D converter is often referred to as an *encoding device* since it is used to encode signals for entry into a digital system.

Digital-to-analog conversion is a straightforward process and is considerably easier than A/D conversion. In fact, a D/A converter is usually an integral part of any A/D converter. For this reason, we consider the D/A conversion process first.

12.1 VARIABLE, RESISTOR NETWORKS

The basic problem in converting a digital signal into an equivalent analog signal is to change the n digital voltage levels into one equivalent analog voltage. This can be most easily accomplished by designing a resistive network that will change each digital level into an equivalent binary weighted voltage (or current).

Binary Equivalent Weight

As an example of what is meant by *binary equivalent weight*, consider the truth table for the 3-bit binary signal shown in Fig. 12.1. Suppose that we want to change the eight possible digital signals in this figure into equivalent analog voltages. The smallest number represented is 000, let us make this equal to 0 V. The largest number is 111: let us make this equal to +7 V. This then establishes the range of the analog signal to be developed. (There is nothing special about the voltage levels chosen; they were simply selected for convenience.)

2^2	2^1	2^0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Fig. 12.1

Now, notice that between 000 and 111 there are seven discrete levels to be defined. Therefore, it will be convenient to divide the analog signal into seven levels. The smallest incremental change in the digital signal is represented by the least-significant bit (LSB), 2^0. Thus we would like to have this bit cause a change in the analog output that is equal to one-seventh of the full-scale analog output voltage. The resistive divider will then be designed such that a 1 in the 2^0 position will cause $+7 \times \frac{1}{7} = +1$ V at the output.

Since $2^1 = 2$ and $2^0 = 1$, it can be clearly seen that the 2^1 bit represents a number that is twice the size of the 2^0 bit. Therefore, a 1 in the 2^1 bit position must cause a change in the analog output voltage that is twice the size of the LSB. The resistive divider must then be constructed such that a 1 in the 2^1 bit position will cause a change of $+7 \times \frac{2}{7} = +2$ V in the analog output voltage.

Similarly, $2^2 = 4 = 2 \times 2^1 = 4 \times 2^0$, and thus the 2^2 bit must cause a change in the output voltage equal to four times that of the LSB. The 2^2 bit must then cause an output voltage change of $+7 \times \frac{4}{7} = +4$ V.

The process can be continued, and it will be seen that each successive bit must have a value twice that of the preceding bit. Thus the LSB is given a binary equivalent weight of $\frac{1}{7}$ or 1 part in 7. The next LSB is given a weight of $\frac{2}{7}$, which is twice the LSB, or 2 parts in 7. The MSB (in the case of this 3-bit system) is given a weight of $\frac{4}{7}$, which is 4 times the LSB or 4 parts in 7. Notice that the sum of the weights must equal 1. Thus $\frac{1}{7}+\frac{2}{7}+\frac{4}{7}=\frac{7}{7} = 1$. In general, the binary equivalent weight assigned to the LSB is $1/(2^n - 1)$, where n is the number of bits. The remaining weights are found by multiplying by 2, 4, 8, and so on. Remember,

$$\text{LSB weight} = \frac{1}{(2^n - 1)}$$

Example 12.1 Find the binary equivalent weight of each bit in a 4-bit system.

Solution The LSB has a weight of $1/(2^4 - 1) = 1/(16 - 1) = \frac{1}{15}$, or 1 part in 15. The second LSB has a weight of $2 \times \frac{1}{15} = \frac{2}{15}$. The third LSB has a weight of $4 \times \frac{1}{15} = \frac{4}{15}$, and the MSB has a weight of $8 \times \frac{1}{15} = \frac{8}{15}$. As a check, the sum of the weights must equal 1. Thus $\frac{1}{15}+\frac{2}{15}+\frac{4}{15}+\frac{8}{15}=\frac{15}{15} = 1$. The binary equivalent weights for 3-bit and 4-bit systems are summarized in Fig. 12.2.

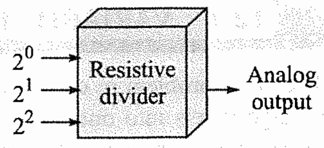
Bit	Weight
2^0	1/7
2^1	2/7
2^2	4/7
Sum	7/7

(a)

Bit	Weight
2^0	1/15
2^1	2/15
2^2	4/15
2^3	8/15
Sum	15/15

(b)

Fig. 12.2 **Binary equivalent weights**



(a)

Digital input			Analog output
0	0	0	+0 V
0	0	1	+1 V
0	1	0	+2 V
0	1	1	+3 V
1	0	0	+4 V
1	0	1	+5 V
1	1	0	+6 V
1	1	1	+7 V

(b)

Fig. 12.3

Resistive Divider

What is now desired is a resistive divider that has three digital inputs and one analog output as shown in Fig. 12.3a. Assume that the digital input levels are 0 = 0 V and 1 = +7 V. Now, for an input of 001, the output will be +1 V. Similarly, an input of 010 will provide an output of +2 V and an input of 100 will provide an output of +4 V. The digital input 011 is seen to be a combination of the signals 001 and 010. If the +1 V from the 2^0 bit is added to the +2 V from the 2^1 bit, the desired +3 V output for the 011 input is achieved. The other desired voltage levels are shown in Fig. 12.3b; they, too, are additive combinations of voltages.

Thus the resistive divider must do two things in order to change the digital input into an equivalent analog output voltage:

1. The 2^0 bit must be changed to +1 V, and 2^1 bit must be changed to +2 V, and 2^2 bit must be changed to +4 V.
2. These three voltages representing the digital bits must be summed together to form the analog output voltage.

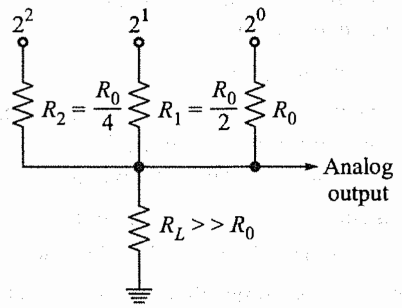


Fig. 12.4

A resistive divider that performs these functions is shown in Fig. 12.4. Resistors R_0, R_1, and R_2 form the divider network. Resistance R_L represents the load to which the divider is connected and is considered to be large enough that it does not load the divider network.

Assume that the digital input signal 001 is applied to this network. Recalling that 0 = 0 V and 1 = +7 V, you can draw the equivalent circuit shown in Fig. 12.5. Resistance R_L is considered large and is neglected. The analog output voltage V_A can be most easily found by use of Millman's theorem, which states that the voltage appearing at any node in a resistive network is equal to the summation of the currents entering the node (found by assuming that the node voltage is zero) divided by the summation of the conductances connected to the node. In equation form, Millman's theorem is

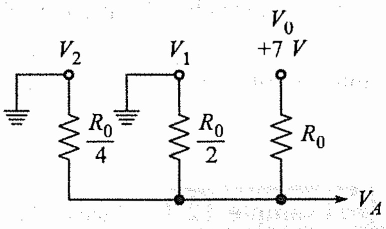


Fig. 12.5

$$V = \frac{V_1/R_1 + V_2/R_2 + V_3/R_3 + \cdots}{1/R_1 + 1/R_2 + 1/R_3 + \cdots}$$

Applying Millman's theorem to Fig. 12.5, we obtain

$$V_A = \frac{V_0/R_0 + V_1/(R_0/2) + V_2/(R_0/4)}{1/R_0 + 1/(R_0/2) + 1/(R_0/4)}$$

$$= \frac{7/R_0}{1/R_0 + 2/R_0 + 4/R_0} = \frac{7}{7} = +1\ V$$

Drawing the equivalent circuits for the other 7-input combinations and applying Millman's theorem will lead to the table of voltages shown in Fig. 12.3 (see Prob. 12.3).

To summarize, a resistive divider can be built to change a digital voltage into an equivalent analog voltage. The following criteria can be applied to this divider:

1. There must be one input resistor for each digital bit.
2. Beginning with the LSB, each following resistor value is one-half the size of the previous resistor.
3. The full-scale output voltage is equal to the positive voltage of the digital input signal. (The divider would work equally well with input voltages of 0 and $-V$.)
4. The LSB has a weight of $1/(2^n - 1)$, where n is the number of input bits.
5. The change in output voltage due to a change in the LSB is equal to $V/(2^n - 1)$, where V is the digital input voltage level.
6. The output voltage V_A can be found for any digital input signal by using the following modified form of Millman's theorem:

$$V_A = \frac{V_0 2^0 + V_1 2^1 + V_2 2^2 + V_3 2^3 + \cdots + V_{n-1} 2^{n-1}}{2^n - 1} \tag{12.1}$$

where $V_0, V_1, V_2, V_3, \ldots, V_{n-1}$ are the digital input voltage levels (0 or V) and n is the number of input bits.

Example 12.2 For a 5-bit resistive divider, determine the following: (a) the weight assigned to the LSB; (b) the weight assigned to the second and third LSB; (c) the change in output voltage due to a change in the LSB, the second LSB, and the third LSB; (d) the output voltage for a digital input of 10101. Assume 0 = 0 V and 1 = +10 V.

Solution

(a) The LSB weight is $1/(2^5 - 1) = 1/31$.

(b) The second LSB weight is 2/31, and the third LSB weight is 4/31.

(c) The LSB causes a change in the output voltage of 10/31 V. The second LSB causes an output voltage change of 20/31 V, and the third LSB causes an output voltage change of 40/31 V.

(d) The output voltage for a digital input of 10101 is

$$V_A = \frac{10 \times 2^0 + 0 \times 2^1 + 10 \times 2^2 + 0 \times 2^3 + 10 \times 2^4}{2^5 - 1}$$

$$= \frac{10(1 + 4 + 16)}{32 - 1} = \frac{210}{31} = +6.77\ \text{V}$$

This resistive divider has two serious drawbacks. The first is the fact that each resistor in the network has a different value. Since these dividers are usually constructed by using precision resistors, the added expense becomes unattractive. Moreover, the resistor used for the MSB is required to handle a much greater current than that used for the LSB resistor. For example, in a 10-bit system, the current through the MSB resistor is approximately 500 times as large as the current through the LSB resistor (see Prob. 12.5). For these reasons, a second type of resistive network, called a *ladder*, has been developed.

SELF-TEST

1. What is the LSB weight of a 6-bit resistive ladder?
2. What is the value of V_A in Example 12.2 if the MSB is 0?

12.2 BINARY LADDERS

The *binary ladder* is a resistive network whose output voltage is a properly weighted sum of the digital inputs. Such a ladder, designed for 4 bits, is shown in Fig. 12.6. It is constructed of resistors that have only two values and thus overcomes one of the objections to the resistive divider previously discussed. The left end of the ladder is terminated in a resistance of $2R$, and we shall assume for the moment that the right end of the ladder (the output) is open-circuited.

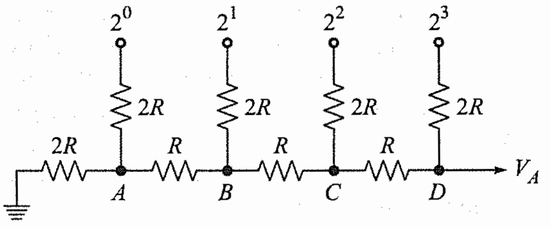


Fig. 12.6 **Binary ladder**

Let us now examine the resistive properties of the network, assuming that all the digital inputs are at ground. Beginning at node A, the total resistance looking into the terminating resistor is $2R$. The total resistance looking out toward the 2^0 input is also $2R$. These two resistors can be combined to form an equivalent resistor of value R as shown in Fig. 12.7.

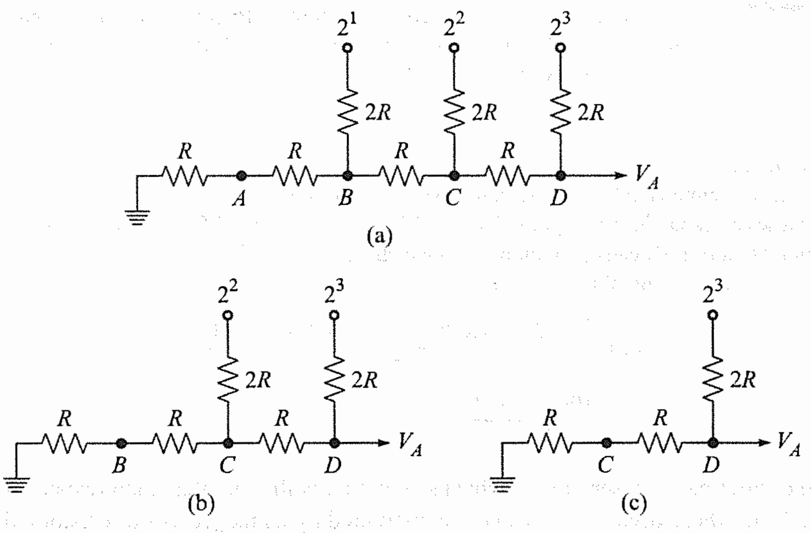


Fig. 12.7

Now, moving to node B, we see that the total resistance looking into the branch toward node A is $2R$, as is the total resistance looking out toward the 2^1 input. These resistors can be combined to simplify the network as shown in Fig. 12.7.

From Fig. 12.7, it can be seen that the total resistance looking from node C down the branch toward node B or out the branch toward the 2^2 input is still $2R$. The circuit in Fig. 12.7 can then be reduced to the equivalent as shown in Fig. 12.7.

From this equivalent circuit, it is clear that the resistance looking back toward node C is $2R$, as is the resistance looking out toward the 2^3 input.

From the preceding discussion, we can conclude that the total resistance looking from any node back toward the terminating resistor or out toward the digital input is $2R$. Notice that this is true regardless of whether the digital inputs are at ground or $+V$. The justification for this statement is the fact that the internal impedance of an ideal voltage source is 0 Ω, and we are assuming that the digital inputs are ideal voltage sources.

We can use the resistance characteristics of the ladder to determine the output voltages for the various digital inputs. First, assume that the digital input signal is 1000. With this input signal, the binary ladder can be drawn as shown in Fig. 12.8a. Since there are no voltage sources to the left of node D, the entire network to the left of this node can be replaced by a resistance of $2R$ to form the equivalent circuit shown in Fig. 12.8b. From this equivalent circuit, it can be easily seen that the output voltage is

$$V_A = V \times \frac{2R}{2R + 2R} = \frac{+V}{2}$$

Thus a 1 in the MSB position will provide an output voltage of $+V/2$.

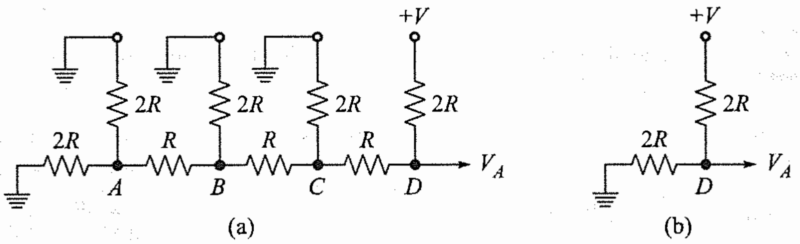


Fig. 12.8 **(a) Binary ladder with a digital input of 1000, (b) Equivalent circuit for a digital input of 1000**

To determine the output voltage due to the second MSB, assume a digital input signal of 0100. This can be represented by the circuit shown in Fig. 12.9a. Since there are no voltage sources to the left of node C, the entire network to the left of this node can be replaced by a resistance of $2R$, as shown in Fig. 12.9b. Let us now replace the network to the left of node C with its Thévenin equivalent by cutting the circuit on the jagged line shown in Fig. 12.9b. The Thévenin equivalent is clearly a resistance R in series with a voltage source $+V/2$. The final equivalent circuit with the Thévenin equivalent included is shown in Fig. 12.9c. From this circuit, the output voltage is clearly

$$V_A = \frac{+V}{2} \times \frac{2R}{R + R + 2R} = \frac{+V}{4}$$

Thus the second MSB provides an output voltage of $+V/4$.

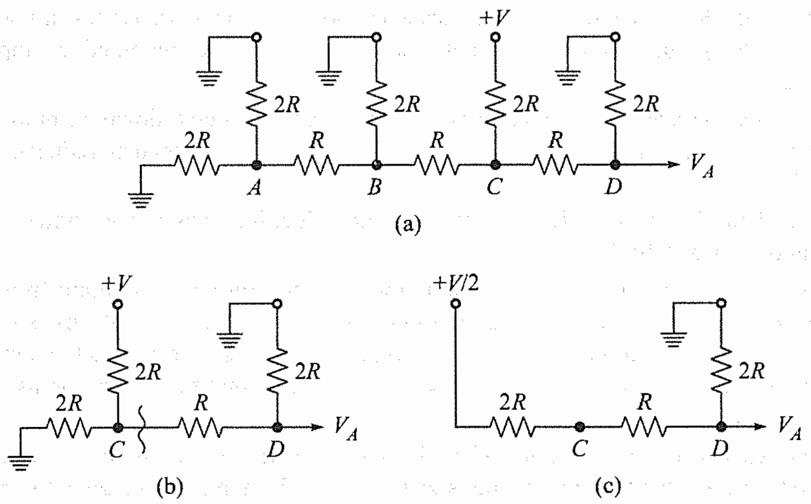


Fig. 12.9 **(a) Binary ladder with a digital input of 0100, (b) Partially reduced equivalent circuit, (c) Final equivalent circuit using Thévenin's theorem**

This process can be continued, and it can be shown that the third MSB provides an output voltage of $+V/8$, the fourth MSB provides an output voltage of $+V/16$, and so on. The output voltages for the binary ladder are summarized in Fig. 12.10; notice that each digital input is transformed into a properly weighted binary output voltage.

Example 12.3 What are the output voltages caused by each bit in a 5-bit ladder if the input levels are 0 = 0 V and 1 = +10 V?

Solution The output voltages can be easily calculated by using Fig. 12.10. They are

First MSB $V_A = \frac{V}{2} = \frac{+10}{2} = +5\text{ V}$

Second MSB $V_A = \frac{V}{4} = \frac{+10}{4} = +2.5\text{ V}$

Third MSB $V_A = \frac{V}{8} = \frac{+10}{8} = +1.25\text{ V}$

Fourth MSB $V_A = \frac{V}{16} = \frac{+10}{16} = +0.625\text{ V}$

LSB = fifth MSB $V_A = \frac{V}{32} = \frac{+10}{32} = +0.3125\text{ V}$

Bit position	Binary weight	Output voltage
MSB	1/2	$V/2$
2d MSB	1/4	$V/4$
3d MSB	1/8	$V/8$
4th MSB	1/16	$V/16$
5th MSB	1/32	$V/32$
6th MSB	1/64	$V/64$
7th MSB	1/128	$V/128$
⋮	⋮	⋮
Nth MSB	$1/2^N$	$V/2^N$

Fig. 12.10 **Binary ladder output voltages**

Since this ladder is composed of linear resistors, it is a linear network and the principle of superposition can be used. This means that the total output voltage due to a combination of input digital levels can be found by simply taking the sum of the output levels caused by each digital input individually.

In equation form, the output voltage is given by

$$V_A = \frac{V}{2} + \frac{V}{4} + \frac{V}{8} + \frac{V}{16} + \cdots + \frac{V}{2^n} \tag{12.2}$$

where n is the total number of bits at the input.

This equation can be simplified somewhat by factoring and collecting terms. The output voltage can then be given in the form

$$V_A = \frac{V_0 2^0 + V_1 2^1 + V_2 2^2 + V_3 2^3 + \cdots + V_{n-1} 2^{n-1}}{2^n} \tag{12.3}$$

where V_0, V_1, V_3, ..., V_{n-1} are the digital input voltage levels. Equation (12.3) can be used to find the output voltage from the ladder for any digital input signal.

Example 12.4 Find the output voltage from a 5-bit ladder that has a digital input of 11010. Assume that $0 = 0$ V and $1 = +10$ V.

Solution By Eq. (12.3):

$$V_A = \frac{0 \times 2^0 + 10 \times 2^1 + 0 \times 2^2 + 10 \times 2^3 + 10 \times 2^4}{2^5}$$

$$= \frac{10(2+8+16)}{32} = \frac{10 \times 26}{32} = +8.125 \text{ V}$$

This solution can be checked by adding the individual bit contributions calculated in Example 12.3.

Notice that Eq. (12.3) is very similar to Eq. (12.1), which was developed for the resistive divider. They are, in fact, identical with the exception of the denominators. This is a subtle but very important difference. Recall that the full-scale voltage for the resistive divider is equal to the voltage level of the digital input 1. On the other hand, examination of Eq. (12.2) reveals that the full-scale voltage for the ladder is given by

$$V_A = V\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{2^n}\right)$$

The terms inside the brackets form a geometric series whose sum approaches 1, given a sufficient number of terms. *However, it never quite reaches 1.* Therefore, the full-scale output voltage of the ladder approaches V in the limit, but never quite reaches it.

Example 12.5 What is the full-scale output voltage of the 5-bit ladder in Example 12.4?

Solution The full-scale voltage is simply the sum of the individual bit voltages. Thus

$$V = 5 + 2.5 + 1.25 + 0.625 + 0.3125 = +9.6875 \text{ V}$$

To keep the ladder in perfect balance and to maintain symmetry, the output of the ladder should be terminated in a resistance of $2R$. This will result in a lowering of the output voltage, but if the $2R$ load is maintained constant, the output voltages will still be a properly weighted sum of the binary input bits. If the load is varied, the output voltage will not be a properly weighted sum, and care must be exercised to ensure that the load resistance is constant.

Terminating the output of the ladder with a load of $2R$ also ensures that the input resistance to the ladder seen by each of the digital voltage sources is constant. With the ladder balanced in this manner, the resistance looking into any branch from any node has a value of $2R$. Thus the input resistance seen by any input digital source is $3R$. This is a definite advantage over the resistive divider, since the digital voltage sources can now all be designed for the same load.

Example 12.6 Suppose that the value of R for the 5-bit ladder described in Example 12.3 is 1000 Ω. Determine the current that each input digital voltage source must be capable of supplying. Also determine the full-scale output voltage, assuming that the ladder is terminated with a load resistance of 2000 Ω.

Solution The input resistance into the ladder seen by each of the digital sources is $3R = 3000$ Ω. Thus, for a voltage level of +10 V, each source must be capable of supplying $I = 10/(3 \times 10^3) = 3\frac{1}{3}$ mA (without the $2R$ load resistor, the resistance looking into the MSB terminal is actually $4R$). The no-load output voltage of the ladder has already been determined in Example 12.5. This open-circuit output voltage along with the open-circuit output resistance can be used to form a Thevénin equivalent circuit for the output of the ladder. The resistance looking back into the ladder is clearly $R = 1000$ Ω. Thus the Thévenin equivalent is as shown in Fig. 12.11. From this figure, the output voltage is

$$V_A = +9.6875 \times \frac{2R}{2R + R} = +6.4583 \text{ V}$$

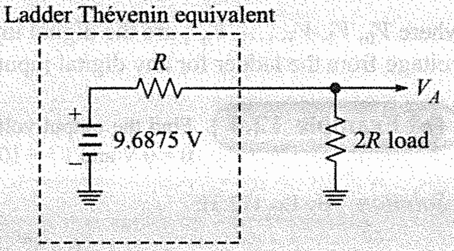


Fig. 12.11 Example 12.6

The operational amplifier (OA) shown in Fig. 12.12a is connected as a unity-gain noninverting amplifier. It has a very high input impedance, and the output voltage is equal to the input voltage. It is thus a good buffer amplifier for connection to the output of a resistive ladder. It will not load down the ladder and thus will not disturb the ladder output voltage V_A; V_A will then appear at the output of the OA.

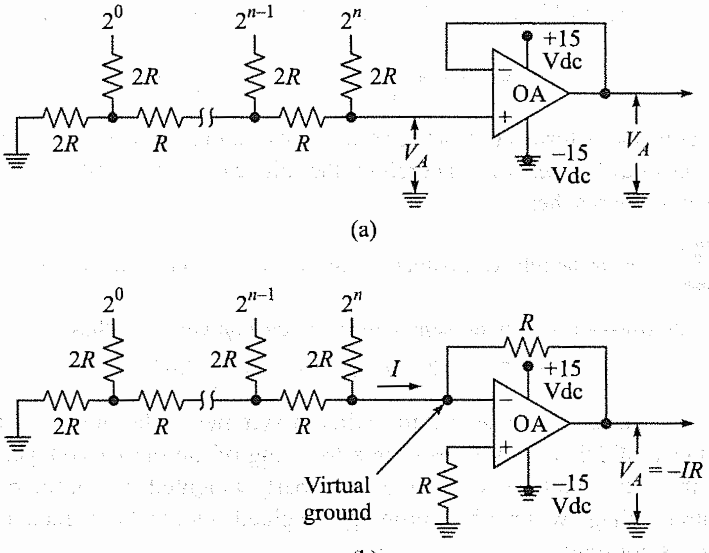


Fig. 12.12

Connecting an OA with a feedback resistor R as shown in Fig. 12.12b results in an amplifier that acts as an inverting current-to-voltage amplifier. That is, the output voltage V_A is equal to the negative of the input current I multiplied by R. The input impedance to this amplifier is essentially 0 Ω: thus, when it is connected to an R-$2R$ ladder, the connecting point is virtually at ground potential. In this configuration, the R-$2R$ ladder will produce a current output I that is a binary weighted sum of the input digital levels. For instance, the MSB produces a current of $V/2R$. The second MSB produces a current of $V/4V$, and so on. But the OA multiplies these currents by $-R$, and thus V_A is

$$V_A = (-R)\left(\frac{V}{2R} + \frac{V}{4R} + \cdots\right) = -\frac{V}{2} - \frac{V}{4} - \cdots$$

This is exactly the same expression given in Eqs. (12.2) and (12.3) except for the sign. Thus the D/A converter in Fig. 12.12a and b will provide the same output voltage V_A except for sign. In Fig. 12.12a, the R-$2R$ ladder and OA are said to operate in a voltage mode, while the connection in Fig. 12.12b is said to operate in a current mode.

SELF-TEST

3. If the ladder in Example 12.4 is increased to 6 bits, what is the output voltage due to the sixth bit alone?
4. If the ladder in Example 12.4 is increased to 6 bits, what is its full-scale output voltage?

12.3 D/A CONVERTERS

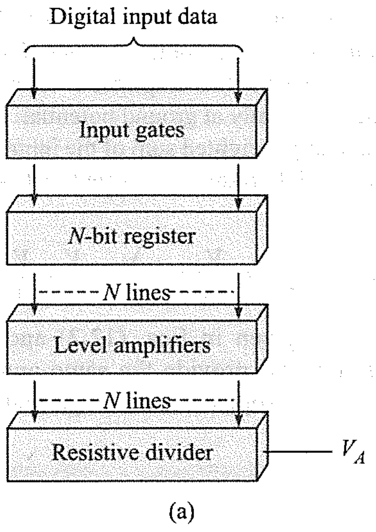
Either the resistive divider or the ladder can be used as the basis for a digital-to-analog (D/A) converter. It is in the resistive network that the actual translation from a digital signal to an analog voltage takes place. There is, however, the need for additional circuitry to complete the design of the D/A converter.

As an integral part of the D/A converter there must be a register that can be used to store the digital information. This register could be any one of the many types discussed in previous chapters. The simplest register is formed by use of *RS* flip-flops, with one flip-flop per bit. There must also be level amplifiers between the register and the resistive network to ensure that the digital signals presented to the network are all of the same level and are constant. Finally, there must be some form of gating on the input of the register such that the flip-flops can be set with the proper infomation from the digital system. A complete D/A converter in block-diagram form is shown in Fig. 12.13a.

Let us expand on the block diagram shown in this Fig. 12.13a by drawing the complete schematic for a 4-bit D/A converter as shown in Fig. 12.13b. You will recognize that the resistor network used is of the ladder type.

The level amplifiers each have two inputs: one input is the +10 V from the precision voltage source, and the other is from a flip-flop. The amplifiers work in such a way that when the input from a flip-flop is high, the output of the amplifier is at +10 V. When the input from the flip-flop is low, the output is 0 V.

The four flip-flops form the register necessary for storing the digital information. The flip-flop on the right represents the MSB, and the flip-flop on the left represents the LSB. Each flip-flop is a simple *RS* latch and requires a positive level at the *R* or *S* input to reset or set it. The gating scheme for entering information into the register is straightforward and should be easy to understand. With this particular gating scheme, the flip-flops need not be reset (or set) each time new information is entered. When the READ IN line goes high, only



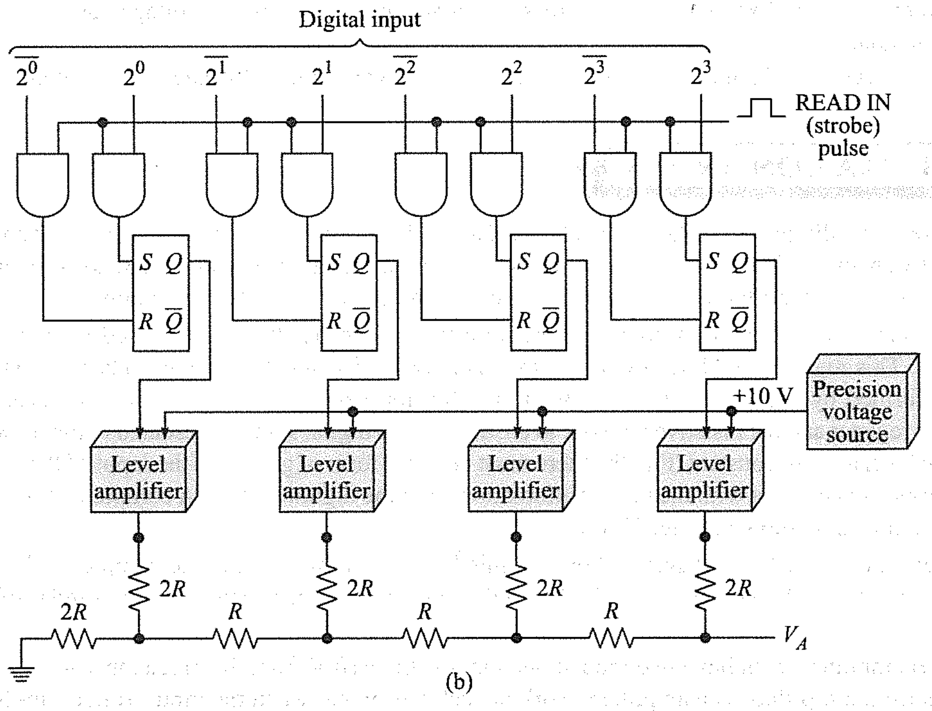


Fig. 12.13 4-bit D/A converter

one of the two gate outputs connected to each flip-flop is high, and the flip-flop is set or reset accordingly. Thus data are entered into the register each time the READ IN (strobe) pulse occurs. *D* flip-flops could be used in place of the *RS* flip-flops.

Multiple Signals

Quite often it is necessary to decode more than one signal—for example, the X and Y coordinates for a plotting board. In this event, there are two ways in which to decode the signals.

The first and most obvious method is simply to use one D/A converter for each signal. This method, shown in Fig. 12.14a, has the advantage that each signal to be decoded is held in its register and the analog output voltage is then held fixed. The digital input lines are connected in parallel to each converter. The proper converter is then selected for decoding by the select lines.

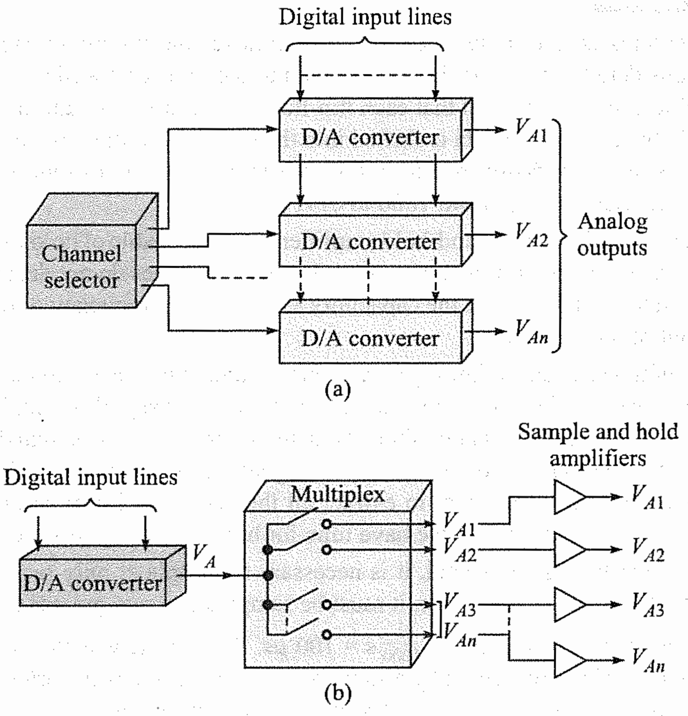


Fig. 12.14 **Decoding a number of signals: (a) Channel selection method, (b) Multiplex method**

The second method involves the use of only one D/A converter and switching its output. This is called *multiplexing*, and such a system is shown in Fig. 12.14b. The disadvantage here is that the analog output signal must be held between sampling periods, and the outputs must therefore be equipped with sample-and-hold amplifiers.

Sample and Hold Circuit

An OA connected as in Fig. 12.15a is a unity-gain noninverting voltage amplifier—that is, $V_o = V_i$. Two such OAs are used with a capacitor in Fig. 12.15b to form a sample-and-hold amplifier. When the switch is closed, the capacitor charges to the D/A converter output voltage. When the switch is opened, the capacitor holds the voltage level until the next sampling time. The operational amplifier provides a large input impedance so as not to discharge the capacitor appreciably and at the same time offers gain to drive external circuits.

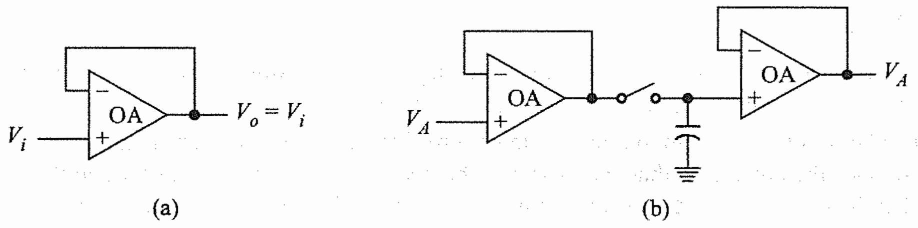


Fig. 12.15 (a) Unity gain amplifier, (b) Sample-and-hold circuit

When the D/A converter is used in conjunction with a multiplexer, the maximum rate at which the converter can operate must be considered. Each time data is shifted into the register, transients appear at the output of the converter. This is due mainly to the fact that each flip-flop has different rise and fall times. Thus a settling time must be allowed between the time data is shifted into the register and the time the analog voltage is read out. This settling time is the main factor in determining the maximum rate of multiplexing the output. The worst case is when all bits change (e.g., from 1000 to 0111).

Naturally, the capacitors on the sample-and-hold amplifiers are not capable of holding a voltage indefinitely; therefore, the sampling rate must be sufficient to ensure that these voltages do not decay appreciably between samples. The sampling rate is a function of the capacitors as well as the frequency of the analog signal which is expected at the output of the converter.

At this point, you might be curious to know just how fast a signal must be sampled in order to preserve its integrity. Common sense leads to the conclusion that the more often the signal is sampled, the less the sample degrades between samples. On the other hand, if too few samples are taken, the signal degrades too much (the sample-and-hold capacitors discharge too much), and the signal information is lost. We would like to reduce the sampling rate to the minimum necessary to extract all the necessary information from the signal. The solution to this problem involves more than we have time for here, but the results are easy enough to apply.

First, if the signal in question is sinusoidal, it is necessary to sample at only *twice* the signal frequency. For instance if the signal is a 5-kHz sine wave, it must be sampled at a rate greater than or equal to 10 kHz. In other words, a sample must be taken every $\frac{1}{1000}$ s = 100 μs. What if the waveform is not sinusoidal? Any waveform that is periodic can be represented by a summation of sine and cosine terms, with each succeeding term having a higher frequency. In this case, it will be necessary to sample at a rate equal to twice the highest frequency of interest.

D/A Converter Testing

Two simple but important tests that can be performed to check the proper operation of the D/A converter are the *steady-state accuracy test* and the *monotonicity test.*

The steady-state accuracy test involves setting a known digital number in the input register, measuring the analog output with an accurate meter, and comparing with the theoretical value.

Checking for monotonicity means checking that the output voltage increases regularly as the input digital signal increases. This can be accomplished by using a counter as the digital input signal and observing the analog output on an oscilloscope. For proper monotonicity, the output waveform should be a perfect staircase waveform, as shown in Fig. 12.16. The steps on the staircase waveform must be equally spaced and of the exact same amplitude. Missing steps, steps of different amplitude, or steps in a downward fashion indicate malfunctions.

The monotonicity test does not check the system for accuracy, but if the system passes the test, it is relatively certain that the converter error is less than 1 LSB. Converter accuracy and resolution are the subjects of the next section.

A D/A converter can be regarded as a logic block having numerous digital inputs and a single analog output as seen in Fig. 12.16b. It is interesting to compare this logic block with the potentiometer shown in Fig. 12.16c. The analog output voltage of the D/A converter is controlled by the digital input signals while the analog output voltage of the potentiometer is controlled by mechanical rotation of the potentiometer shaft. Considered in this fashion, it is easy to see how a D/A converter could be used to generate a voltage waveform (sawtooth, triangular, sinusoidal, etc.). It is, in effect, a digitally controlled voltage generator!

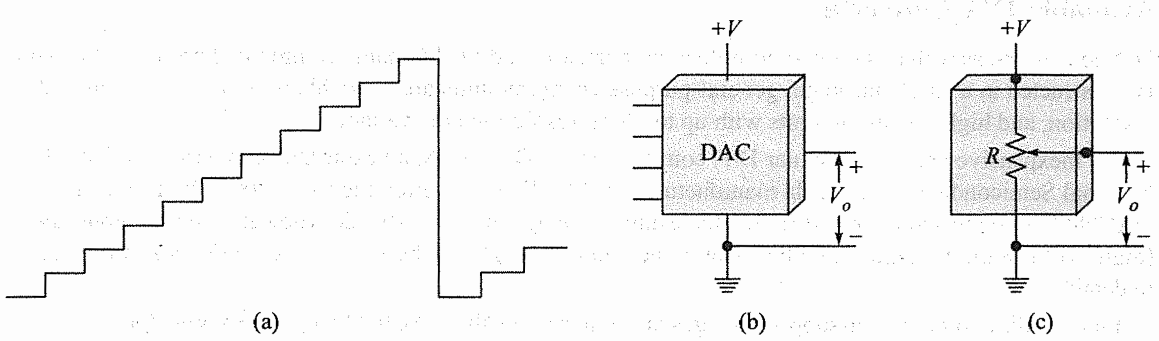


Fig. 12.16 **Correct output voltage waveform for monotonicity test**

Example 12.7 Suppose that in the course of a monotonicity check on the 4-bit converter in Fig. 12.13 the waveform shown in Fig. 12.17 is observed. What is the probable malfunction in the converter?

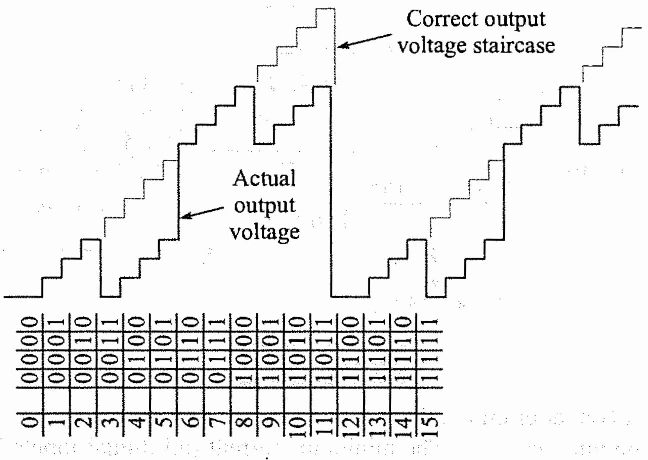


Fig. 12.17 **Irregular output voltage for Example 12.7**

Solution There is obviously some malfunction since the actual output waveform is not continuously increasing as it should be. The actual digital inputs are shown directly below the wave-form. Notice that the converter functions correctly up to count 3. At count 4, however, the output should be 4 units in amplitude. Instead, it drops to 0. It remains 4 units below the correct level until it reaches count 8. Then, from count 8 to 11, the output level is correct. But again at count 12 the output falls 4 units below the correct level and remains there for the next four levels. If you examine the waveform carefully, you will note that the output is 4 units below normal during the time when the 2^2 bit is supposed to be high. This then suggests that the 2^2 bit is being dropped (i.e., the 2^2 input to the ladder is not being held high). This means that the 2^2-level amplifier is malfunctioning or the 2^2 AND gate is not operating properly. In any case, the monotonicity check has clearly shown that the second MSB is not being used and that the converter is not operating properly.

Available D/A Converters

D/A converters, as well as sample-and-hold amplifiers, are readily obtainable commercial products. Each unit is constructed in a single package; general-purpose economy units are available with 6-, 8-, 10-, and 12-bit resolution, and high-resolution units with up to 16-bit resolution are available.

An inexpensive and very popular D/A converter is the DAC0808, an 8-bit D/A converter available from National Semiconductor. Motorola manufactures an 8-bit D/A converter, the MC1508/1408. In Fig. 12.18, a DAC0808 is connected to provide a full-scale output voltage of $V_o = +10$ Vdc when all 8 digital inputs are 1s (high). If the 8 digital inputs are all 0s (low), the output voltage will be $V_o = 0$ Vdc. Let's look at this circuit in detail.

First of all, two dc power-supply voltages are required for the DAC0808: $V_{CC} = +5$ Vdc and $V_{EE} = -15$ Vdc. The 0.1-μF capacitor is to prevent unwanted circuit oscillations, and to isolate any variations in V_{EE}. Pin 2 is ground (GND), and pin 15 is also referenced to ground through a resistor.

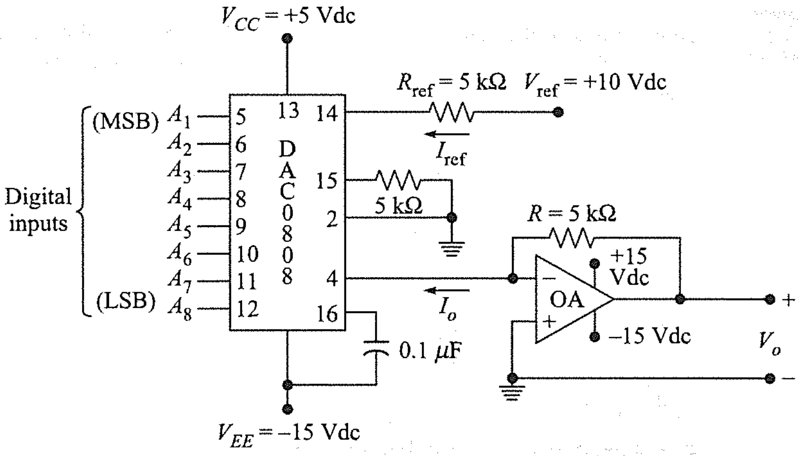


Fig. 12.18

The output of the D/A converter on pin 4 has a very limited voltage range (+0.5 to −0.6 V). Rather, it is designed to provide an output current I_o. The minimum current (all digital inputs low) is 0.0 mA, and the maximum current (all digital inputs high), is I_{ref}. This reference current is established with the resistor at pin 14 and the reference voltage as

$$I_{ref} = V_{ref}/R_{ref} \tag{12.4}$$

The D/A converter output current I_o is given as

$$I_o = I_{\text{ref}}\left(\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\cdots+\frac{A8}{256}\right) \tag{12.5}$$

where $A_1, A_2, A_3, \ldots, A_8$ are the digital input levels (1 or 0).

The OA is connected as a current-to-voltage converter, and the output voltage is given as

$$V_o = I_o \times R \tag{12.6}$$

Substituting Eqs. (12.4) and (12.5) into Eq. (12.6),

$$V_o = V_{\text{ref}} / R_{\text{ref}} \times \left(\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\cdots+\frac{A8}{256}\right) \times R \tag{12.7}$$

If we set the OA feedback resistor R equal to R_{ref}, then

$$V_o = V_{\text{ref}}\left(\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\cdots+\frac{A8}{256}\right) \tag{12.8}$$

Let's try out Eq. (12.8). Suppose all digital inputs are 0s (all low). Then

$$\begin{aligned} V_o &= V_{\text{ref}} \times \left(\frac{0}{2}+\frac{0}{4}+\frac{0}{8}+\cdots+\frac{0}{256}\right) \\ &= V_{\text{ref}} \times 0 = 0.0 \text{ Vdc} \end{aligned}$$

Now, suppose all digital inputs are 1s (all high). Then

$$\begin{aligned} V_o &= V_{\text{ref}} \times \left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{256}\right) \\ &= (V_{\text{ref}}) \times \left(\frac{255}{256}\right) = 0.996 \times V_{\text{ref}} \end{aligned}$$

Since V_{ref} in Fig. 12.18 is +10 Vdc, the output voltage is seen to have a range between 0.0 and +9.96 Vdc. It doesn't quite reach +10 Vdc, but this is characteristic of this type of circuit. This circuit is essentially the current-mode operation discussed in the previous section and illustrated in Fig. 12.12b.

Example 12.8 In Fig. 12.16, $A1$ is high, $A2$ is high, $A5$ is high and $A7$ is high. The other digital inputs are all low. What is the output voltage V_o?

Solution

$$V_o = 10 \times \left(\frac{1}{2}+\frac{1}{4}+\frac{1}{32}+\frac{1}{128}\right) = 10 \times 0.789 = 7.89 \text{ V}$$

SELF-TEST

5. What is a monotonicity test?
6. What would be the full-scale output voltage in Fig. 12.18 if V_{ref} were changed to +5 Vdc?

12.4 D/A ACCURACY AND RESOLUTION

Two very important aspects of the D/A converter are the resolution and the accuracy of the conversion. There is a definite distinction between the two, and you should clearly understand the differences.

The accuracy of the D/A converter is primarily a function of the accuracy of the precision resistors used in the ladder and the precision of the reference voltage supply used. Accuracy is a measure of how close the actual output voltage is to the theoretical output value.

For example, suppose that the theoretical output voltage for a particular input should be +10 V. An accuracy of 10 percent means that the actual output voltage must be somewhere between +9 and +11 V. Similarly, if the actual output voltage were somewhere between +9.9 and +10.1 V, this would imply an accuracy of 1 percent.

Resolution, on the other hand, defines the smallest increment in voltage that can be discerned. Resolution is primarily a function of the number of bits in the digital input signal; that is, the smallest increment in output voltage is determined by the LSB.

In a 4-bit system using a ladder, for example, the LSB has a weight of $\frac{1}{16}$. This means that the smallest increment in output voltage is $\frac{1}{16}$ of the input voltage. To make the arithmetic easy, let us assume that this 4-bit system has input voltage levels of +16 V. Since the LSB has a weight of $\frac{1}{16}$, a change in the LSB results in a change of 1 V in the output. Thus the output voltage changes in steps (or increments) of 1 V. The output voltage of this converter is then the staircase shown in Fig. 12.16 and ranges from 0 to +15V in 1-V increments. This converter can be used to represent analog voltages from 0 to +15 V, but it cannot resolve voltages into increments smaller than 1 V. If we desired to produce +4.2 V using this converter, therefore, the actual output voltage would be +4.0 V. Similarly, if we desired a voltage of +7.8 V, the actual output voltage would be +8.0 V. It is clear that this converter is not capable of distinguishing voltages finer than 1 V, which is the resolution of the converter.

If we wanted to represent voltages to a finer resolution, we would have to use a converter with more input bits. As an example, the LSB of a 10-bit converter has a weight of $\frac{1}{1024}$. Thus the smallest incremental change in the output of this converter is approximately $\frac{1}{1000}$ of the full-scale voltage. If this converter has a +10-V full-scale output, the resolution is approximately $+10 \times \frac{1}{1000} = 10$ mV. This converter is then capable of representing voltages to within 10 mV.

Example 12.9 What is the resolution of a 9-bit D/A converter which uses a ladder network? What is this resolution expressed as a percent? If the full-scale output voltage of this converter is +5 V, what is the resolution in volts?

Solution The LSB in a 9-bit system has a weight of $\frac{1}{512}$. Thus this converter has a resolution of 1 part in 512. The resolution expressed as a percentage is $\frac{1}{512} \times 100$ percent $\cong 0.2$ percent. The voltage resolution is obtained by multiplying the weight of the LSB by the full-scale output voltage. Thus the resolution in volts is $\frac{1}{512} \times 5 \cong 10$ mV.

Example 12.10 How many bits are required at the input of a convener if it is necessary to resolve voltages to 5 mV and the ladder has +10 V full scale?

Solution The LSB of an 11-bit system has a resolution of $\frac{1}{2048}$. This would provide a resolution at the output of $\frac{1}{2048} \times +10 \cong +5$ mV.

It is important to realize that resolution and accuracy in a system should be compatible. For example, in the 4-bit system previously discussed, the resolution was found to be 1 V. Clearly it would be unjustifiable to construct such a system to an accuracy of 0.1 percent. This would mean that the system would be accurate to 16 mV but would be capable of distinguishing only to the nearest 1 V.

Similarly, it would be wasteful to construct the 11-bit system described in Example 12.19 to an accuracy of only 1 percent. This would mean that the output voltage would be accurate only to 100 mV, whereas it is capable of distinguishing to the nearest 5 mV.

SELF-TEST

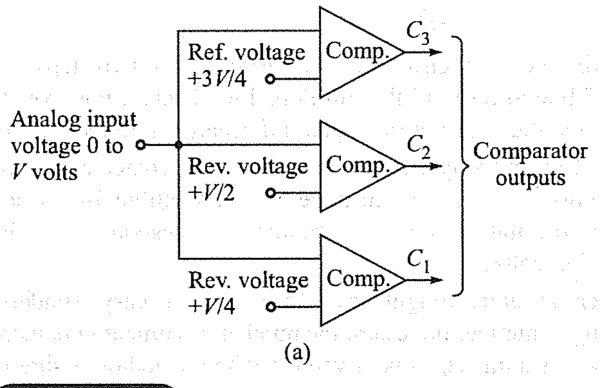
7. What is the resolution of the DAC0808 in Fig. 12.18?

12.5 A/D CONVERTER—SIMULTANEOUS CONVERSION

The process of converting an analog voltage into an equivalent digital signal is known as *analog-to-digital (A/D) conversion*. This operation is somewhat more complicated than the converse operation of D/A conversion. A number of different methods have been developed, the simplest of which is probably the simultaneous method. This is also known as *A/D converter, flash type,* the reason for which will be clear shortly.

The simultaneous method of A/D conversion is based on the use of a number of comparator circuits. One such system using three comparator circuits is shown in Fig. 12.19 below. The analog signal to be digitized serves as one of the inputs to each comparator. The second input is a standard reference voltage. The reference voltages used are $+V/4$, $+V/2$, and $+3V/4$. The system is then capable of accepting an analog input voltage between 0 and $+V$.

If the analog input signal exceeds the reference voltage to any comparator, that comparator turns on. (Let's assume that this means that the output of the comparator goes high.) Now, if all the comparators are off, the analog input signal must be between 0 and $+V/4$. If C_1 is high (comparator C_1 is on) and C_2 and C_3 are low, the input must be between $+V/4$ and $+V/2$ V. If C_1 and C_2 are high while C_3 is low, the input must be between $+V/2$ and $+3V/4$. Finally, if all comparator outputs are high, the input signal must be between $+3V/4$ and $+V$. The comparator output levels for the various ranges of input voltages are summarized in Fig. 12.19.



Input voltage	Comparator output		
	C_1	C_2	C_3
0 to $+V/4$	Low	Low	Low
$+V/4$ to $+V/2$	High	Low	Low
$+V/2$ to $+3V/4$	High	High	Low
$+3V/4$ to $+V$	High	High	High

(b)

Fig. 12.19 **Simultaneous A/D conversion: (a) Logic diagram, (b) Comparator outputs for input voltage ranges**

Examination of Fig. 12.19 reveals that there are four voltage ranges that can be detected by this converter. Four ranges can be effectively discerned by two binary digits (bits). The three comparator outputs can then be fed into a coding network to provide 2 bits which are equivalent to the input analog voltage. The bits of the coding network can then be entered into a flip-flop register for storage. The complete block diagram for such an A/D converter is shown in Fig. 12.20.

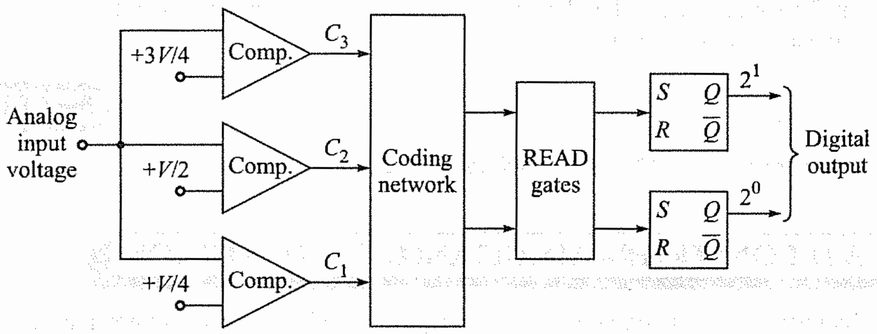


Fig. 12.20 **2-bit simultaneous A/D converter**

In order to gain a clear understanding of the operation of the simultaneous A/D converter, let us investigate the 3-bit converter shown in Fig. 12.21a. Notice that in order to convert the input signal to a digital signal having 3 bits, it is necessary to have seven comparators (this allows a division of the input into eight ranges). For the 2-bit converter, remember that three comparators were necessary for defining four ranges. In general, it can be said that $2^n - 1$ comparators are required to convert to a digital signal that has n bits. Some of the comparators have inverters at their outputs since both C and $\overline{C}$ are needed for the encoding matrix.

The encoding matrix must accept seven input levels and encode them into a 3-bit binary number (having eight possible states). Operation of the encoding matrix can be most easily understood by examination of the table of outputs in Fig. 12.22.

The 2^2 bit is easiest to determine since it must be high (the 2^2 flip-flop must be set) whenever C_4 is high.

The 2^1 line must be high whenever C_2 is high and $\overline{C}_4$ is high, or whenever C_6 is high. In equation form, we can write $2^1 = C_2\overline{C}_4 + C_6$.

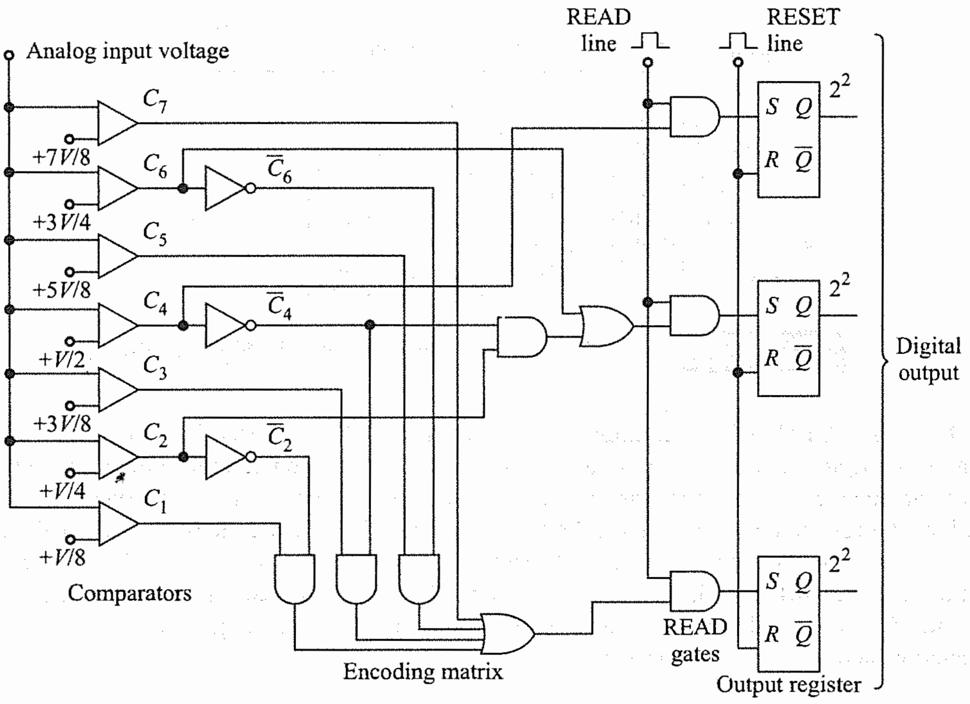
The logic equation for the 2^0 bit can be found in a similar manner; it is

$$2^0 = C_1\overline{C}_2 + C_3\overline{C}_4 + C_5\overline{C}_6 + C_7$$

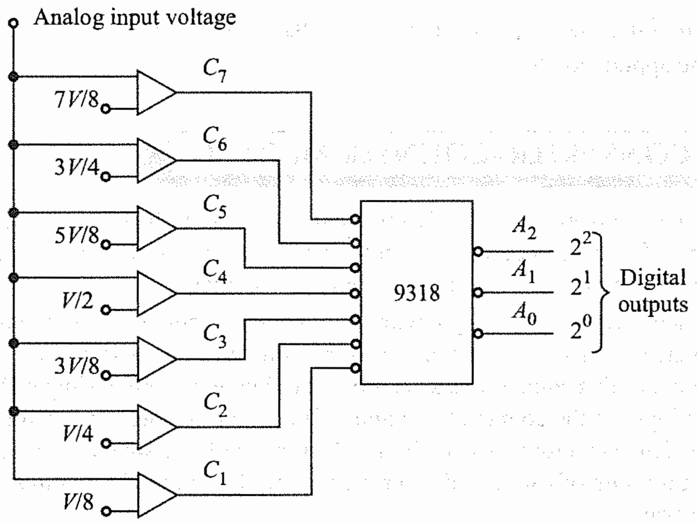
The transfer of data from the encoding matrix into the register must be carried out in two steps. First, a positive reset pulse must appear on the RESET line to reset all the flip-flops low. Then, a positive READ pulse allows the proper READ gates to go high and thus transfer the digital information into the flip-flops.

Interestingly, a convenient application for a 9318 priority encoder is to use it to replace all the digital logic as shown in Fig. 12.21b. Of course, the inputs $C_1, C_2, \ldots, C_7$ must be TTL-compatible. In essence, the output of the 9318 is a digital number that reflects the highest-order zero input; this corresponds to the lowest reference voltage that still exceeds the input analog voltage.

The construction of a simultaneous A/D converter is quite straightforward and relatively easy to understand. However, as the number of bits in the desired digital number increases, the number of comparators increases very rapidly ($2^n - 1$), and the problem soon becomes unmanageable. Even though this method is simple and is capable of extremely fast conversion rates, here are preferable methods for digitizing numbers having more than 3 or 4 bits. Because it is so fast, this type of converter is frequently called a *flash converter*.



(a)



(b)

Fig. 12.21 **3-bit simultaneous A/D converter: (a) Logic diagram, (b) Using a 9318 priority encoder**

Input voltage	Comparator for level							Binary output		
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	2^2	2^1	2^0
0 to $V/8$	Low	Low	Low	Low	Low	Low	Low	0	0	0
$V/8$ to $V/4$	High	Low	Low	Low	Low	Low	Low	0	0	1
$V/4$ to $3V/8$	High	High	Low	Low	Low	Low	Low	0	1	0
$3V/8$ to $V/2$	High	High	High	Low	Low	Low	Low	0	1	1
$V/2$ to $5V/8$	High	High	High	High	Low	Low	Low	1	0	0
$5V/8$ to $3V/4$	High	High	High	High	High	Low	Low	1	0	1
$3V/4$ to $7V/8$	High	High	High	High	High	High	Low	1	1	0
$7V/8$ to V	High	High	High	High	High	High	High	1	1	1

Fig. 12.22 **Logic table tor the converter in Fig. 12.19(a)**

The Motorola MC10319 is an example of an 8-bit flash A/D converter. The input has 256 parallel comparators connected to a precision voltage divider network. The comparator outputs are fed to latches and then to an encoder network that captures the digital signal in Gray code. Gray code is used to ensure that small input errors do not result in large digital signal errors. The Gray code is then decoded into straight binary and presented to the outputs, which are tri-state TTL = compatible. The flash A/D converter is capable of operation with a 25-MHz clock! It comes in a 24-pin DIP and requires two dc supply voltages—typically +5 Vdc and –5 Vdc. Possible applications include radar signal processing, video displays, high-speed instrumentation, and television broadcasting.

SELF-TEST

8. Why is a simultaneous A/D converter called a flash converter?
9. What is one application for a flash converter?

12.6 A/D CONVERTER–COUNTER METHOD

A higher-resolution A/D converter using only one comparator could be constructed if a variable reference voltage were available. This reference voltage could then be applied to the comparator, and when it became equal to the input analog voltage, the conversion would be complete.

To construct such a converter, let us begin with a simple binary counter. The digital output signals will be taken from this counter, and thus we want it to be an n-bit counter, where n is the desired number of bits. Now let us connect the output of this counter to a standard binary ladder to form a simple D/A converter. If a clock is now applied to the input of the counter, the output of the binary ladder is the familiar staircase waveform shown in Fig. 12.16. This waveform is exactly the reference voltage signal we would like to have for the comparator! With a minimum of gating and control circuitry, this simple D/A converter can be changed into the desired A/D converter.

Figure 12.23 shows the block diagram for a counter-type A/D converter. The operation of the counter is as follows. First, the counter is reset to all 0s. Then, when a convert signal appears on the START line, the gate opens and clock pulses are allowed to pass through to the input of the counter. The counter advances through its normal binary count sequence, and the staircase waveform is generated at the output of the ladder. This waveform is applied to one side of the comparator, and the analog input voltage is applied to the other side. When the reference voltage equals (or exceeds) the input analog voltage, the gate is closed, the counter stops, and the conversion is complete. The number stored in the counter is now the digital equivalent of the analog input voltage.

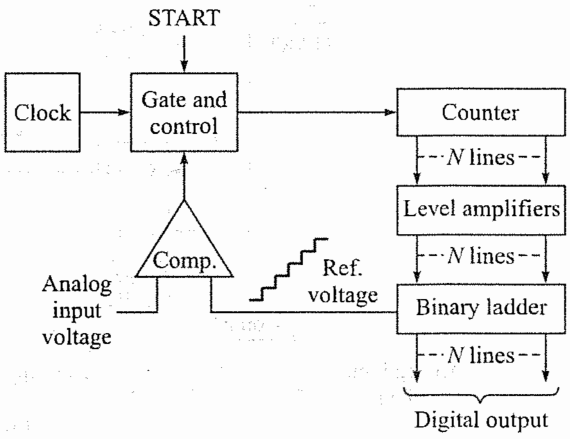


Fig. 12.23 **Counter type A/D converter**

Notice that this converter is composed of a D/A converter (the counter, level amplifiers, and the binary ladder), one comparator, a clock, and the gate and control circuitry. This can really be considered as a closed-loop control system. An error signal is generated at the output of the comparator by taking the difference between the analog input signal and the feedback signal (staircase reference voltage). The error is detected by the control circuit, and the clock is allowed to advance the counter. The counter advances in such a way as to reduce the error signal by increasing the feedback voltage. When the error is reduced to zero, the feedback voltage is equal to the analog input signal, the control circuitry stops the clock from advancing the counter, and the system comes to rest.

The counter-type A/D converter provides a very good method for digitizing to a high resolution. This method is much simpler than the simultaneous method for high resolution, but the conversion time required is longer. Since the counter always begins at zero and counts through its normal binary sequence, as many as 2^n counts may be necessary before conversion is complete. The average conversion time is, of course, $2^n/2$ or 2^{n-1} counts.

The counter advances one count for each cycle of the clock, and the clock therefore determines the conversion rate. Suppose, for example, that we have a 10-bit converter. It requires 1024 clock cycles for a full-scale count. If we are using a 1-MHz clock, the counter advances 1 count every microsecond. Thus, to count full scale requires $1024 \times 10^{-6} = 1.024$ ms. The converter reaches one-half full scale in half this time, or in 0.512 ms. The time required to reach one-half full scale can be considered the *average* conversion time for a large number of conversions.

Example 12.11 Suppose that the converter shown in Fig. 12.23 is an 8-bit converter driven by a 500-kHz clock. Find (a) the maximum conversion time; (b) the average conversion time; (c) the maximum conversion rate.

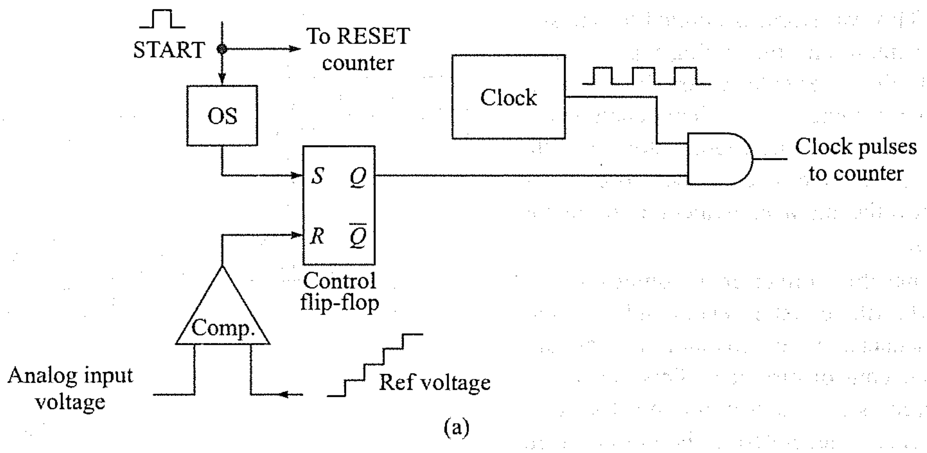
Solution

(a) An 8-bit converter has a maximum of $2^8 = 256$ counts. With a 500-kHz clock, the counter advances at the rate of 1 count each 2 μs. To advance 256 counts requires $256 \times 2 \times 10^{-6} = 512 \times 10^{-6} = 512\ \mu$s.

(b) The average conversion time is one-half the maximum conversion time. Thus it is $1/2 \times 0.512 \times 10^{-3} = 0.256$ ms.

(c) The maximum conversion rate is determined by the longest conversion time. Since the converter has a maximum conversion time of 0.512 ms, it is capable of making at least $1/(0.512 \times 10^{-3}) \cong 1953$ conversions per second.

Figure 12.24 shows one method of implementing the control circuitry for the converter shown in Fig. 12.23. The waveforms for one conversion are also shown. A conversion is initiated by the receipt of a START signal.



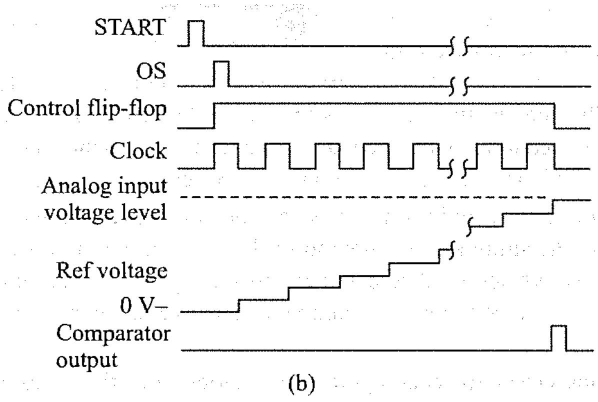


Fig. 12.24 **Control of the A/D converter in Fig. 12.21**

The positive edge of the START pulse is used to reset all the flip-flops in the counter and to trigger the one-shot. The output of the one-shot sets the control flip-flop, which makes the AND gate true and allows clock pulses to advance the counter.

The delay between the RESET pulse to the flip-flops and the beginning of the clock pulses (ensured by the one-shot) is to ensure that all flip-flops are reset before counting begins. This is a definite attempt to avoid any racing problems.

With the control flip-flop set, the counter advances through its normal count sequence until the staircase voltage from the ladder is equal to the analog input voltage. At this time, the comparator output changes state, generating a positive pulse which resets the control flip-flop. Thus the AND gate is closed and counting ceases. The counter now holds a digital number which is equivalent to the analog input voltage. The converter remains in this state until another conversion signal is received.

If a new start signal is generated immediately after each conversion is completed, the converter will operate at its maximum rate. The converter could then be used to digitize a signal as shown in Fig. 12.25a Notice that the conversion times in digitizing this signal are not constant but depend on the amplitude of the input signal. The analog input signal can be reconstructed from the digital information by drawing straight

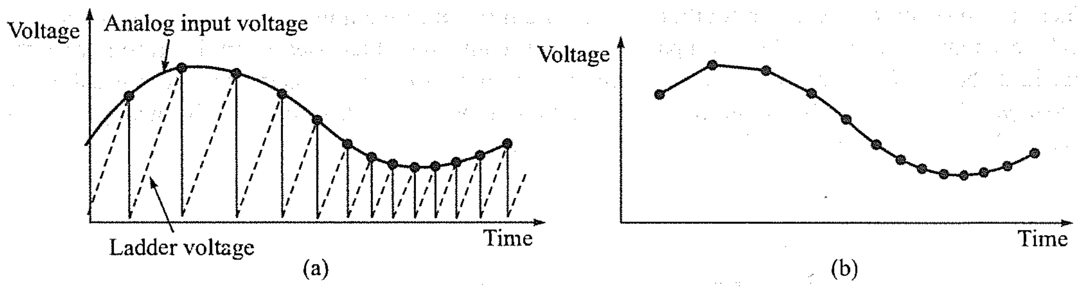


Fig. 12.25 (a) Digitizing an analog voltage. (b) Reconstructed signal from the digital data.

lines from each digitized point to the next. Such a reconstruction is shown in Fig. 12.25b; it is, indeed, a reasonable representation of the original input signal. In this case, it is important to note that the conversion times are smaller than the transient time of the input waveform.

On the other hand, if the transient time of the input waveform approaches the conversion time, the reconstructed output signal is not quite so accurate. Such a situation is shown in Fig. 12.26a and b. In this case, the input waveform changes at a rate faster than the converter is capable of recognizing. Thus the need for reducing conversion time is apparent.

SELF-TEST

10. The A/D converter in Fig. 12.23 has 8 bits and is driven by a 2-MHz clock. What is the maximum conversion time?
11. What is the average conversion time for the converter in question 10?

12.7 CONTINUOUS A/D CONVERSION

An obvious method for speeding up the conversion of the signal as shown in Fig. 12.26 is to eliminate the need for resetting the counter each time a conversion is made. If this were done. the counter would not begin at zero each time, but instead would begin at the value of the last converted point. This means that the counter would have to be capable of counting either up or down. This is no problem; we are already familiar with the operation of up-down counters.

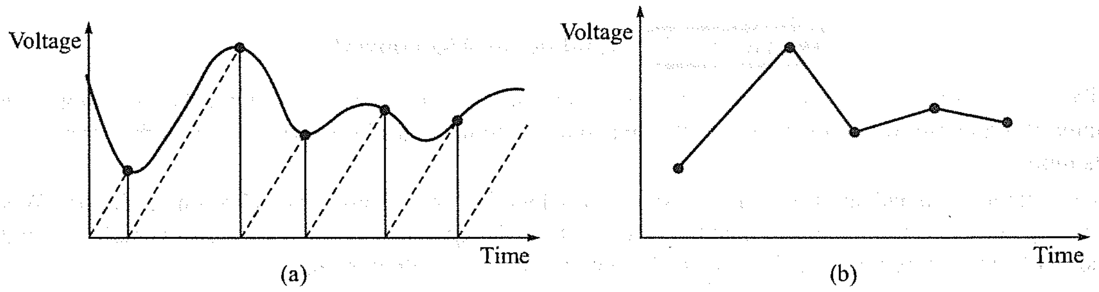
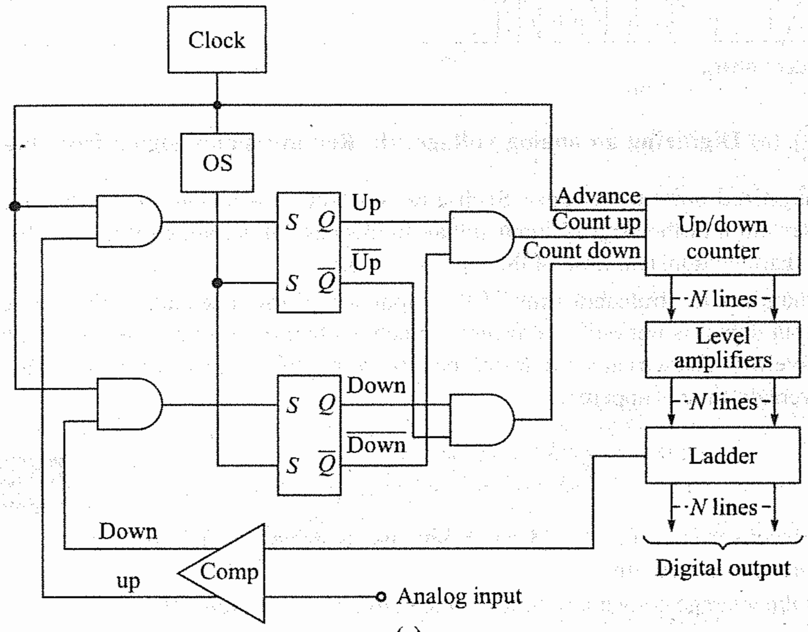
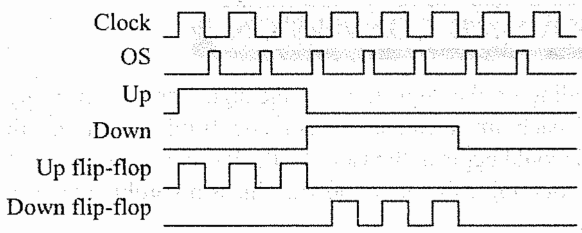


Fig. 12.26 (a) Digitizing an analog voltage, (b) Reconstructed signal from the digital data

There is, however, the need for additional logic circuitry, since we must decide whether to count up or down by examining the output of the comparator. An A/D converter which uses an up-down counter is shown in Fig. 12.27 below. This method is known as *continuous conversion*, and thus the converter is called a *continuous-type A/D converter*. Since the converter's digital output always tries to track the analog input to the converter, this is also known as *A/D converter-tracking type*.



(a)



(b)

Fig. 12.27 Continuous A/D converter

The D/A portion of this converter is the same as those previously discussed, with the exception of the counter. It is an up-down counter and has the up and down count control lines in addition to the advance line at its input.

The output of the ladder is fed into a comparator which has two outputs instead of one as before. When the analog voltage is more positive than the ladder output, the *up* output of the comparator is high. When the analog voltage is more negative than the ladder output, the *down* output is high.

If the *up* output of the comparator is high, the AND gate at the input of the *up* flip-flop is open, and the first time the clock goes positive, the *up* flip-flop is set. If we assume for the moment that the *down* flip-flop is reset, the AND gate which controls the *count-up* line of the counter will be true and the counter will advance one count. The counter can advance only one count since the output of the one-shot resets both the *up* and the *down* flip-flops just after the clock goes low. This can then be considered as one count-up conversion cycle.

Notice that the AND gate which controls the *count-up* line has inputs of *up* and $\overline{down}$. Similarly, the count-down line AND gate has inputs of *down* and $\overline{up}$. This could be considered an exclusive-OR arrangement and ensures that the count-down and count-up lines cannot both be high at the same time.

As long as the *up* line out of the comparator is high, the converter continues to operate one conversion cycle at a time. At the point where the ladder voltage becomes more positive than the analog input voltage, the *up* line of the comparator goes low and the *down* line goes high. The converter then goes through a count-down conversion cycle. At this point, the ladder voltage is within 1 LSB of the analog voltage, and the converter oscillates about this point. This is not desirable since we want the converter to cease operation and not jump around the final value. The trick here is to adjust the comparator such that its outputs do not change at the same time.

We can accomplish this by adjusting the comparator such that the *up* output will not go high unless the ladder voltage is more than 1/2 LSB below the analog voltage. Similarly, the *down* output will not go high unless the ladder voltage is more than 1/2 LSB above the analog voltage. This is called *centering on the LSB* and provides a digital output which is within 1/2 LSB.

A waveform typical of this type of converter is shown in Fig. 12.28. You can see that this converter is capable of following input voltages that change at a much faster rate.

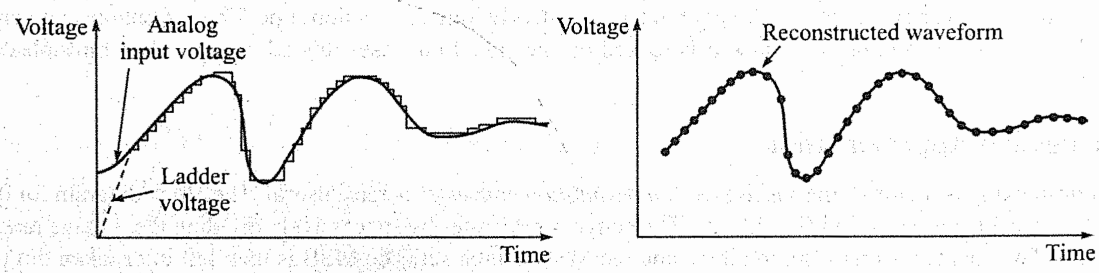


Fig. 12.28 **Continuous A/D conversion**

Example 12.12 Quite often, additional circuitry is added to a continuous converter to ensure that it cannot count off scale in either direction. For example, if the counter contained all 1s, it would be undesirable to allow it to progress through a count-up cycle, since the next count would advance it to all 0s. We would like to design the logic necessary to prevent this.

Solution The two limit points which must be detected are all 1s and all 0s in the counter. Suppose that we construct an AND gate having the 1 sides of all the counter flip-flops as its inputs. The output of this gate will be true whenever the counter contains all 1s. If the gate is then connected to the reset side of the *up* flip-flop, the counter will be unable to count beyond all 1s.

Similarly, we might construct an AND gate in which the inputs are the 0 sides of all the counter flip-flops. The output of this gate can be connected to the reset side of the *down* flip-flop, and the counter will then be unable to count beyond all 0s. The gates are shown in Fig. 12.29.

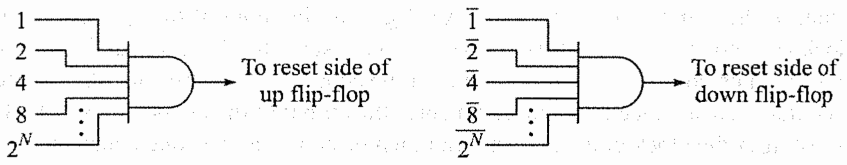


Fig. 12.29 Count-limiting gates for the converter in Fig. 12.25

SELF-TEST

12. How does the continuous-type A/D converter differ from the simple counter-type A/D converter?
13. What advantage does the continuous-type A/D converter offer over the counter-type A/D converter?

12.8 A/D TECHNIQUES

There are a variety of other methods for digitizing analog signals—too many to discuss in detail. Nevertheless, we shall take the time to examine two more techniques and the reasons for their importance.

Probably the most important single reason for investigating other methods of conversion is to determine ways to reduce the conversion time. Recall that the simultaneous converter has a very fast conversion time. The counter converter is simple logically but has a relatively long conversion time. The continuous converter has a very fast conversion time once it is locked on the signal but loses this advantage when multiplexing inputs.

Successive Approximation

If multiplexing is required, the *successive-approximation converter* is most useful. The block diagram for this type of converter is shown in Fig. 12.30a. The converter operates by successively dividing the voltage ranges in half. The counter is first reset to all 0s, and the MSB is then set. The MSB is then left in or taken out (by resetting the MSB flip-flop) depending on the output of the comparator. Then the second MSB is set in, and a comparison is made to determine whether to reset the second MSB flip-flop. The process is repeated down to the LSB, and at this time the desired number is in the counter. Since the conversion involves operating on one flip-flop at a time, beginning with the MSB, a ring counter may be used for flip-flop selection.

The successive-approximation method thus is the process of approximating the analog voltage by trying 1 bit at a time beginning with the MSB. The operation is shown in diagram form in Fig. 12.30b. It can be seen from this diagram that each conversion takes the same time and requires one conversion cycle for each bit. Thus the total conversion time is equal to the number of bits, n, times the time required for one conversion cycle. One conversion cycle normally requires one cycle of the clock. As an example, a 10-bit converter operating with a 1-MHz clock has a conversion time of $10 \times 10^{-6} = 10^{-5} = 10\ \mu s$.

When dealing with conversion times this short, it is usually necessary to take into account the other delays in the system (e.g. switching time of the multiplexer, settling time of the ladder network, comparator delay, and settling time).

All the logic blocks inside the dashed line in Fig. 12.30a, or some equivalent arrangement, are frequently constructed on a single MSI chip; this chip is called a *successive-approximation register* (SAR). For example, the Motorola MC6108 shown in Fig. 12.28c is an 8-bit microprocessor-compatible A/D converter that includes an SAR, D/A conversion capabilities, control logic, and buffered digital outputs, in a 28-pin DIP.

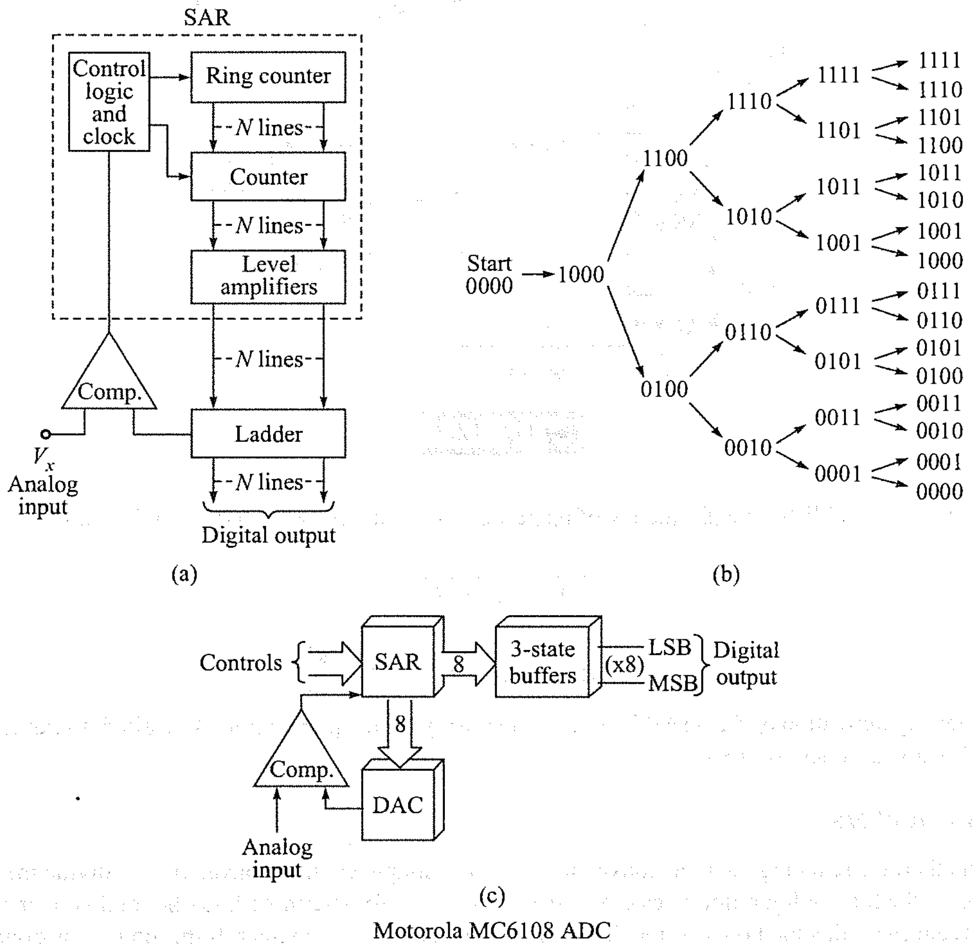


Fig. 12.30 **Successive approximation converter**

The ADC0804

The ADC0804 is an inexpensive and very popular A/D converter which is available from a number of different manufacturers, including National Semiconductor. The ADC0804 is an 8-bit CMOS microprocessor compatible successive-approximation A/D converter that is supplied in a 20-pin DIP. It is capable of digitizing an analog input voltage within the range 0 to +5 Vdc, and it only requires a single dc supply voltage—usually +5 Vdc. The digital outputs are both TTL- and CMOS-compatible.

The block diagram of an ADC0804 is shown in Fig. 12.31. In this case, the controls are wired such that the converter operates continuously. This is the so-called *free-running mode*. The 10-kΩ resistor, along with the

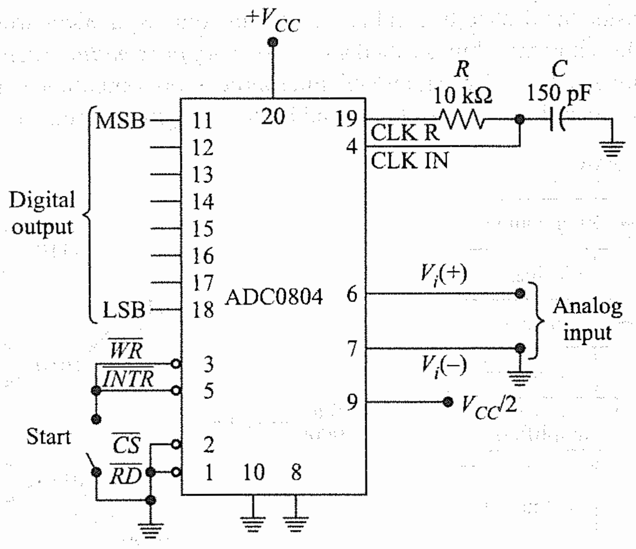


Fig. 12.31

150-pF capacitor, establishes the frequency of operation according to $f \approx 1/1.1(RC)$. In this case,

$$f \approx \frac{1}{1.1 \times (10\ \text{k}\Omega \times 150\ \text{pF})}$$

$$= \frac{1}{1.1 \times (10^4 \times 1.5 \times 10^{-12})} = 607\ \text{kHz}$$

A momentary activation of the START switch is necessary to begin operation. A detailed discussion of the ADC0804 is given in Section 15.4.

Section Counters

Another method for reducing the total conversion time of a simple counter converter is to divide the counter into sections. Such a configuration is called a *section counter*. To determine how the total conversion time might be reduced by this method, assume that we have a standard 8-bit counter. If this counter is divided into two equal counters of 4 bits each, we have a section converter. The converter operates by setting the section containing the four LSBs to all 1s and then advancing the other sections until the ladder voltage exceeds the input voltage. At this point the four LSBs are all reset, and this section of the counter is then advanced until the ladder voltage equals the input voltage.

Notice that a maximum of $2^4 = 16$ counts is required for each section to count full scale. Thus this method requires only $2 \times 2^4 = 2^5 = 32$ counts to reach full scale. This is a considerable reduction over the $2^8 = 256$ counts required for the straight 8-bit counter. There is, of course, some extra time required to set the counters initially and to switch from counter to counter during the conversion. This logical operation time is very small, however, compared with the total time saved by this method.

This type of converter is quite often used for digital voltmeters, since it is very convenient to divide the counters by counts of 10. Each counter is then used to represent one of the digits of the decimal number appearing at the output of the voltmeter. We discuss this subject in detail in the next chapter.

SELF-TEST

14. What does SAR stand for in Fig. 12.30c?
15. What is an ADC0804?

12.9 DUAL-SLOPE A/D CONVERSION

Up to this point, our interest in different methods of A/D conversion has centered on reducing the actual conversion time. If a very short conversion time is not a requirement, there are other methods of A/D conversion that are simpler to implement and much more economical. Basically, these techniques involve comparison of the unknown input voltage with a reference voltage that begins at zero and increases linearly with time. The time required for the reference voltage to increase to the value of the unknown voltage is directly proportional to the magnitude of the unknown voltage, and this time period is measured with a digital counter. This is referred to as a *single-ramp method*, since the reference voltage is sloped like a ramp. A variation on this method involves using an operational amplifier integrating circuit in a dual-ramp configuration. The dual-ramp method is very popular, and widely used in digital voltmeters and digital panel meters. It offers good accuracy, good linearity, and very good noise-rejection characteristics.

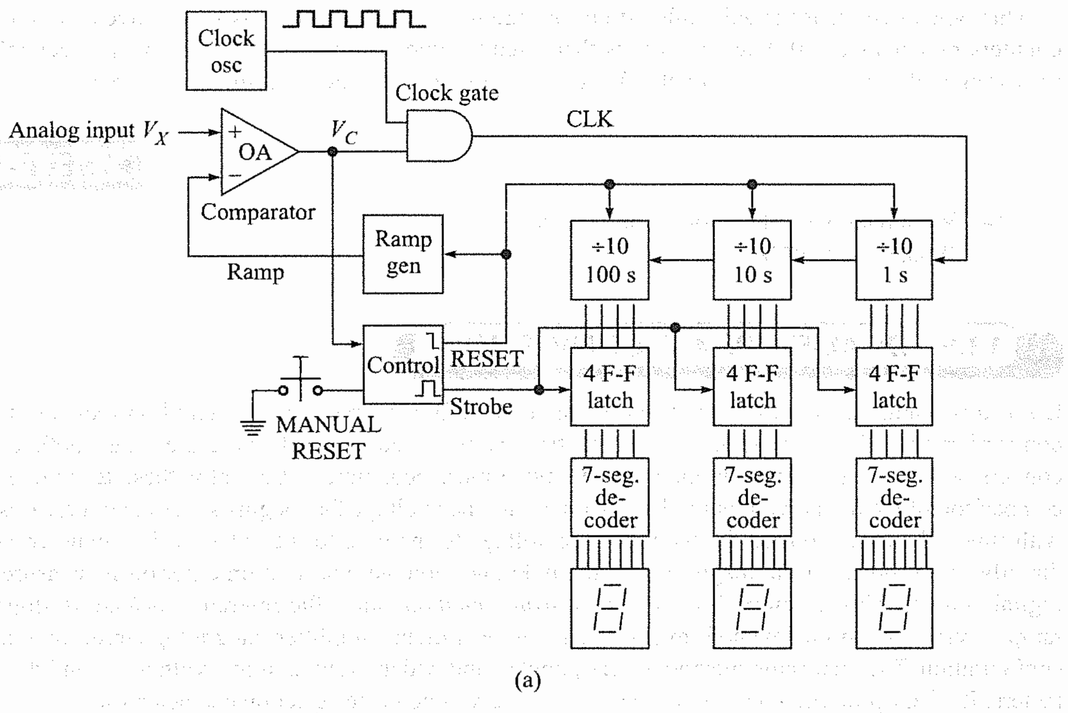
Single-Ramp A/D Converter

Let's take a look at the single-ramp A/D converter in Fig. 12.32. The heart of this converter is the *ramp generator*. This is a circuit that produces an output voltage ramp as shown in Fig. 12.33a. The output voltage begins at zero and increases linearly up to a maximum voltage V_m. It is important that this voltage be a straight line—that is, it must have a constant slope. For instance, if $V_m = 1.0$ Vdc, and it takes 1.0 ms for the ramp to move from 0.0 up to 1.0 V, the slope is 1 V/ms, or 1000 V/s.

This ramp generator can be constructed in a number of different ways. One way might be to use a D/A converter driven by a simple binary counter. This would generate the staircase waveform previously discussed and shown in Fig. 12.16a. A second method is to use an operational amplifier (OA) connected as an integrator as shown in Fig. 12.33b. For this circuit, if V_i is a constant, the output voltage is given by the relationship $V_o = (V_i/RC)t$. Since V_i, R, and C are all constants, this is the equation of a straight line that has a slope (V_i/RC) as shown in Fig. 12.33a. Now that we have a way to generate a voltage ramp and we understand its characteristics, let's return to the converter in Fig. 12.32.

We assume that the clock is running continuously and that any input voltage V_X that we wish to digitize is positive. If it is not, there are circuits that we can use to adjust for negative input signals. The three decade counters are connected in cascade, and their outputs can be strobed into three 4-flip-flop latch circuits. The latches are then decoded by seven-segment decoders to drive the LED displays as units, tens, and hundreds of counts. We can begin a conversion cycle by depressing the MANUAL RESET switch.

Refer carefully to the logic diagram and the waveforms in Fig. 12.32. MANUAL RESET generates a RESET pulse that clears all the decade counters to 0s and resets the ramp voltage to zero. Since V_X is positive and RAMP begins at zero, the output of the comparator OA, V_c, must be high. This voltage enables the



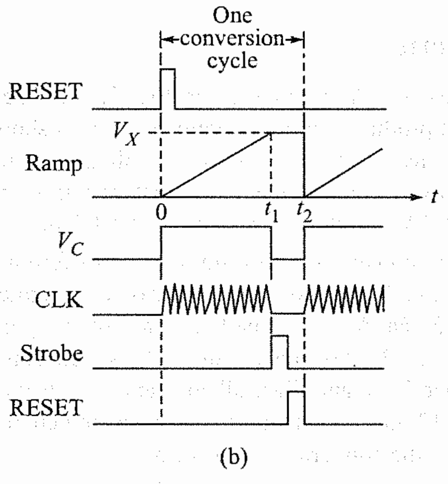
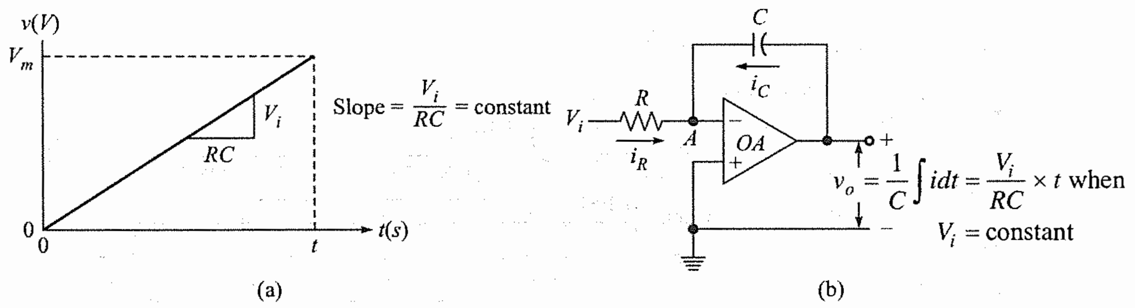


Fig. 12.32 **Single-slope A/D converter**

CLOCK gate allowing the clock, CLK, to be applied to the decade counter. The counter begins counting upward, and the RAMP continues upward until the ramp voltage is equal to the unknown input V_X.

At this point, time t_1, the output of the comparator V_c goes low, thus disabling the CLOCK gate and the counters cease to advance. Simultaneously, this negative transition on V_c generates a STROBE signal in the CONTROL box that shifts the contents of the three decade counters into the three 4-flip-flop latch circuits.



v(V)
V_m
V_i
RC
Slope = $\frac{V_i}{RC}$ = constant
0
0
t
t(s)
(a)
C
i_C
R
V_i
A
i_R
OA
$v_o = \frac{1}{C}\int i dt = \frac{V_i}{RC} \times t$ when
V_i = constant
(b)

Fig. 12.33 An integrating circuit

Shortly thereafter, a reset pulse is generated by the CONTROL box that resets the RAMP and clears the decade counters to 0s, and another conversion cycle begins. In the meantime, the contents of the previous conversion are contained in the latches and are displayed on the seven-segment LEDs.

As a specific example, suppose that the clock in Fig. 12.32 is set at 1.0 MHz and the ramp voltage slope is 1.0 V/ms. Note that the decade counters have the ability to store and display any decimal number from 000 up to 999. From the beginning of a conversion cycle, it will require 999 clock pulses (999 μs) for the counters to advance full scale. During this same time period, the ramp voltage will have increased from 0.0 V up to 999 mV. So, this circuit as it stands will display the value of any input voltage between 0.0 V and 999 mV.

In effect, we have a digital voltmeter! For instance, if V_X = 345 mV, it will require 345 clock pulses for the counter to advance from 000 to 345, and during the same time period the ramp will have increased to 345 mV. So, at the end of the conversion cycle, the display output will read 345—we supply the units of millivolts.

One weakness of the single-slope A/D converter is its dependency on an extremely accurate ramp voltage. This in turn is strongly dependent on the values of *R* and *C* and variations of these values with time and temperature. The dual-slope A/D converter overcomes these problems.

Dual-Slope A/D Converter

The logic diagram for a basic dual-slope A/D converter is given in Fig. 12.32. With the exception of the ramp generator and the comparator, the circuit is similar to the single-slope A/D converter in Fig. 12.32. In this case, the integrator forms the desired ramp—in fact, two different ramps—as the input is switched first to the unknown input voltage V_X and then to a known reference voltage V_r. Here's how it works.

We begin with the assumptions that the clock is running, and that the input voltage V_X is positive. A conversion cycle begins with the decade counters cleared to all 0s, the ramp reset to 0.0 V, and the input switched to the unknown input voltage V_X. Since V_X is positive, the integrator output V_c will be a negative ramp. The comparator output V_g is thus positive and the clock is allowed to pass through the CLOCK GATE to the counters. We allow the ramp to proceed for a fixed time period t_1, determined by the count detector for time t_1. The actual voltage V_c at the end of the fixed time period t_1 will depend on the unknown input V_X, since we know that $V_c = -(V_x/RC) \times t_1$ for an integrator.

When the counter reaches the fixed count at time t_1, the CONTROL unit generates a pulse to clear the decade counters to all 0s and switch the integrator input to the negative reference voltage V_r. The integrator will now begin to generate a ramp beginning at $-V_c$ and increasing steadily upward until it reaches 0.0 V. All this time, the counter is counting, and the conversion cycle ends when V_c = 0.0 V since the CLOCK GATE

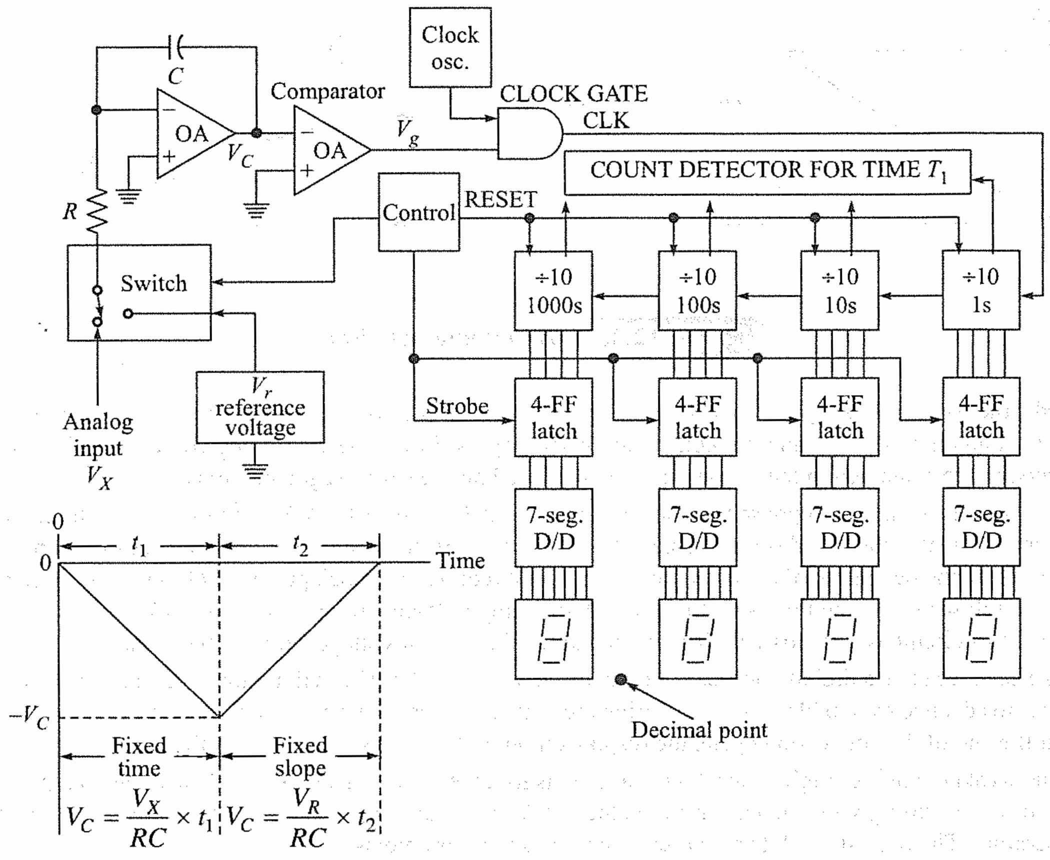


Fig. 12.34 **Dual-slope A/D converter**

is now disabled. The equation for the positive ramp is $V_c = (V_r/RC) \times t_2$. In this case, the slope of this ramp (V_r/RC) is constant, but the time period t_2 is variable.

In fact, since the integrator output voltage begins at 0.0 V, integrates down to $-V_c$, and then integrates back up to 0.0 V, we can equate the two equations given for V_c. That is:

$$\frac{V_X}{RC} \times t_1 = \frac{V_r}{RC} \times t_2$$

The value RC will cancel from both sides, leaving

$$V_X = V_r \times \frac{t_2}{t_1}$$

Since V_r is a known reference voltage and t_1 is a predetermined time, clearly the unknown input voltage is directly proportional to the variable time period t_2. However, this time period is exactly the contents of the decade counters at the end of a conversion cycle! The obvious advantage here is that the RC terms cancel from both sides of the equation above—in other words, this technique is free from the absolute values of either R or C and also from variations in either value.

As a concrete example, let's suppose that the clock in Fig. 12.34 is 1.0 MHz, the reference voltage is –1.0 Vdc, the fixed time period t_1 is 1000 μs, and the RC time constant of the integrator is set at $RC = 1.0$ ms. During the time period t_1, the integrator voltage V_c, will ramp down to –1.0 Vdc if $V_X = 1.0$ V. Then, during time t_2, V_c will ramp all the way back up to 0.0 V, and this will require a time of 1000 μs, since the slope of this ramp is fixed at 1.0 V/ms. The output display will now read 1000, and with placement of a decimal as shown, this reads 1.000 V.

Another way of expressing the operation of this A/D converter is to solve the equation $V_X = V_r(t_2/t_1)$ for t_2, since t_2 is the digital readout. Thus $t_2 = (V_X/V_r)t_1$. If the same values as given above are applied, an unknown input voltage $V_X = 2.75$ V will be digitized and the readout will be $t_2 = (2.75/1.0)1000 = 2750$, or 2.75 V, using the decimal point on the display. Notice that we have used $t_1 = 1000$, the number of clock pulses that occur during the time period t_1. Likewise, t_2 is the number of clock pulses that occur during the time period t_2.

SELF-TEST

16. Is a single-ramp A/D converter slower or faster than a successive-approximation A/D converter?
17. What is the greatest weakness of a single-ramp A/D converter?
18. What advantage does the dual-slope A/D converter offer over the single-ramp A/D converter?

12.10 A/D ACCURACY AND RESOLUTION

Since the A/D converter is a closed-loop system involving both analog and digital systems, the overall accuracy must include errors from both the analog and digital positions. In determining the overall accuracy it is easiest to separate the two sources of error.

If we assume that all components are operating properly, the source of the digital error is simply determined by the resolution of the system. In digitizing an analog voltage, we are trying to represent a continuous analog voltage by an equivalent set of digital numbers. When the digital levels are converted back into analog form by the ladder, the output is the familiar staircase waveform. This waveform is a representation of the input voltage but is certainly not a continuous signal. It is, in fact, a discontinuous signal composed of a number of discrete steps. In trying to reproduce the analog input signal, the best we can do is to get on the step which most nearly equals the input voltage in amplitude.

The simple fact that the ladder voltage has steps in it leads to the digital error in the system. The smallest digital step, or quantum, is due to the LSB and can be made smaller only by increasing the number of bits in the counter. This inherent error is often called the *quantization error* and is commonly ±1 bit. If the comparator is centered, as with the continuous converter, the quantization error can be made ±1/2 LSB.

The main source of analog error in the A/D converter is probably the comparator. Other sources of error are the resistors in the ladder, the reference-voltage supply ripple, and noise. These can, however, usually be made secondary to the sources of error in the comparator.

The sources of error in the comparator are centered around variations in the dc switching point. The dc switching point is the difference between the input voltage levels that cause the output to change state. Variations in switching are due primarily to offset, gain, and linearity of the amplifier used in the comparator. These parameters usually vary slightly with input voltage levels and quite often with temperature. It is these changes which give rise to the analog error in the system.

An important measure of converter performance is given by the differential linearity. *Differential linearity* is a measure of the variation in voltage-step size that causes the converter to change from one state to the next: It is usually expressed as a percent of the average step size. This performance characteristic is also a function of the conversion method and is best for the converters having counters that count continuously. The counter-type and continuous-type converters usually have better differential linearity than do the successive-approximation-type converters. This is true since the ladder voltage is always approaching the analog voltage from the same direction in the one case. In the other case, the ladder voltage is first on one side of the analog voltage and then on the other. The comparator is then being used in both directions, and the net analog error from the comparator is thus greater.

The next logical question that might be asked is: what should be the relative order of magnitudes of the analog and digital errors? As mentioned previously, it would be difficult to justify construction of a 15-bit converter that has an overall error of 1 percent. In general, it is considered good practice to construct converters having analog and digital errors of approximately the same magnitudes. There are many arguments for and against this, and any final argument would have to depend on the situation. As an example, an 8-bit converter would have a quantization error of $\frac{1}{256} \cong 0.4$ percent. It would then seem reasonable to construct this converter to an accuracy of 0.5 percent in an effort to achieve an overall accuracy of 1.0 percent. This might mean constructing the ladder to an accuracy of 0.1 percent, the comparator to an accuracy of 0.2 percent, and so on, since these errors are all accumulative.

Example 12.13 What overall accuracy could one reasonably expect from the construction of a 10-bit A/D converter?

Solution A 10-bit converter has a quantization error of $\frac{1}{1024} \cong 0.1$ percent. If the analog portion can be constructed to an accuracy of 0.1 percent, it would seem reasonable to strive for an overall accuracy of 0.2 percent.

SUMMARY

Digital-to-analog conversion, the process of converting digital input levels into an equivalent analog output voltage, is most easily accomplished by the use of resistance networks. The binary ladder has been found to have definite advantages over the resistance divider. The complete D/A converter consists of a binary ladder (usually) and a flip-flop register to hold the digital input information.

The simultaneous method for A/D conversion is very fast but becomes cumbersome for more than a few bits of resolution. The counter-type A/D converter is somewhat slower but represents a much more reasonable solution for digitizing high-resolution signals. The continuous-converter method, the successive-approximation method, and the section-counter method are all variations of the basic counter-type A/D converter which lead to a much faster conversion time. A dual-slope A/D converter is somewhat slower than the previously discussed methods but offers excellent accuracy in a relatively inexpensive circuit. Dual-slope A/D converters are widely used in digital voltmeters.

The D/A converter and A/D converter logic circuits given in this chapter are all drawn in logic block diagram form and can all be constructed by simply connecting these commercially available logic blocks. For instance, a D/A converter can be constructed by connecting resistors that have values of R and $2R$, or an A/D converter can be constructed by connecting the various inverters, gates, flip-flops, and so on; however, you must realize that these units are now readily available as MSI circuits. The only really practical and economical way to build D/A converters or A/D converters is to make use of these commercially available circuits; this is exactly the subject pursued in the next chapter.

GLOSSARY

- ***A/D conversion*** The process of converting an analog input voltage to a number of equivalent digital output levels.
- ***A/D converter flash type*** Effects fast and simultaneous conversion of analog data to digital through number of comparators.
- ***A/D converter tracking type*** Effects tracking of analog input through its continuous comparison with converter's digital output.
- ***binary equivalent weight*** The value assigned to each bit in a digital number, expressed as a fraction of the total. The values are assigned in binary fashion according to the sequence 1, 2, 4, 8, … , 2^n, where n is the total number of bits.
- ***D/A conversion*** The process of converting a number of digital input signals to one equivalent analog output voltage.
- ***differential linearity*** A measure of the variation in size of the input voltage to an A/D converter which causes the converter to change from one state to the next.
- ***Millman's theorem*** A theorem from network analysis which states that the voltage at any node in a resistive network is equal to the sum of the currents entering the node divided by the sum of the conductances connected to the node, all determined by assuming that the voltage at the node is zero.
- ***monotonicity*** A consistent increase in output in response to a consistent increase in input (voltage or current).
- ***quantization error*** The error inherent in any digital system due to the size of the LSB.
- ***sample and hold circuit*** Samples analog voltage signal and holds briefly to facilitate analog to digital conversion.
- ***SAR*** Sequential approximation register, used in a sequential A/D converter.

PROBLEMS

Section 12.1

12.1 What is the binary equivalent weight of each bit in a 6-bit resistive divider?

12.2 Draw the schematic for a 6-bit resistive divider.

12.3 Verify the voltage output levels for the network in Fig. 12.4, using Millman's theorem. Draw the equivalent circuits.

12.4 Assume that the divider in Prob. 12.2 has +10 V full-scale output, and find the following:

a. The change in output voltage due to a change in the LSB

b. The output voltage for an input of 110110

12.5 A 10-bit resistive divider is constructed such that the current through the LSB resistor is 100 μA. Determine the maximum current that will flow through the MSB resistor.

Sections 12.2, 12.3 and 12.4

12.6 What is the full-scale output voltage of a 6-bit binary ladder if 0 = 0 V and 1 = +10 V? Of an 8-bit ladder?

12.7 Find the output voltage of a 6-bit binary ladder with the following inputs:

a. 101001 b. 111011

c. 110001

12.8 Check the results of Prob. 11-7 by adding the individual bit contributions.

12.9 What is the resolution of a 12-bit D/A converter which uses a binary ladder? If the full-scale output is +10 V, what is the resolution in volts?

12.10 How many bits are required in a binary ladder to achieve a resolution of 1 mV if full scale is +5 V?

Section 12.5

12.11 How many comparators are required to build a 5-bit simultaneous A/D converter?

12.12 Redesign the encoding matrix and READ gates in Fig. 12.20, using NAND gates.

12.13 Assuming that the input reference voltage is $V = 10.0$ Vdc, determine the digital output of the A/D converter in Fig. 12.21a for an input voltage of:

a. 1.25 V b. 3.33 V
c. 8.05 V

Section 12.6

12.14 Find the following for a 12-bit counter-type A/D converter using a 1-MHz clock:

a. Maximum conversion time
b. Average conversion time
c. Maximum conversion rate

12.15 What clock frequency must be used with a 10-bit counter-type A/D converter if it must be capable of making at least 7000 conversions per second?

12.16 Design additional control circuitry for Fig. 12.24 such that the A/D converter in Fig. 12.23 will continue to make conversions after an initial START pulse is applied.

Sections 12.7 and 12.8

12.17 What is the conversion time of a 12-bit successive-approximation-type A/D converter using a 1-MHz clock?

12.18 What is the conversion time of a 12-bit section-counter-type A/D converter using a 1-MHz clock? The counter is divided into three equal sections.

Section 12.9

12.19 For the integrator in Fig. 12.31, show that the output voltage is given by $V_o = \{V_i/RC\}t$, assuming that the input voltage V_i is a constant. [*Hint*: Using Kirchhoff's current law at node A, the resistor current i_R is equal to the capacitor current i_C, but $i_R = V_i/R$ and $i_C = q/t = (V_oC)/t$.]

12.20 Design the control logic for the CONTROL box in Fig. 12.32 to generate the proper control signals shown in that figure.

12.21 Calculate a value for C in Fig. 12.33 to obtain a fixed slope $V_i/(RC) = 1000$ V/s, given $V_i = 1.0$ Vdc and $R = 100$ kΩ.

12.22 Can you design an amplifier such that the output is always positive and is equal to the magnitude of the input voltage? In other words, the input can be either $+V_i$ or $-V_i$ but in either case, the output will be $+V_i$.

12.23 Design the CONTROL logic for the converter in Fig. 12.34.

Section 12.10

12.24 What overall accuracy could you reasonably expect from a 12-bit A/D converter?

12.25 Discuss the overall acceptable accuracy of a 10-bit A/D converter in terms of quantization error, ladder accuracy, comparator accuracy, converter accuracy, and other factors.

Answers to Self-tests

1. 1/63
2. 30/15 = 2 V
3. 0.15625 V
4. 9.84375 V
5. A monotonicity test checks to see that the D/A output voltage increases regularly as the input digital signals increase.
6. +5 Vdc
7. Resolution = 10/256 = 39.06 mV
8. Its conversion time is very fast.
9. Possibilities include radar signal processing, video displays, high-speed instrumentation, and television broadcasting.
10. 128 μs

11. 64 μs
12. The continuous type A/D converter uses an up-down counter.
13. The continuous type A/D converter is faster than the counter-type A/D converter.
14. SAR stands for successive-approximation register.
15. The ADC0804 is an 8-bit CMOS successive-approximation A/D converter.
16. Slower
17. One major weakness of the single-slope A/D converter is that it is extremely sensitive to variations in ramp voltage and hence to errors in ramp voltage.
18. The *RC* time constant cancels out, making the conversion much less sensitive to variations in ramp voltage accuracy.

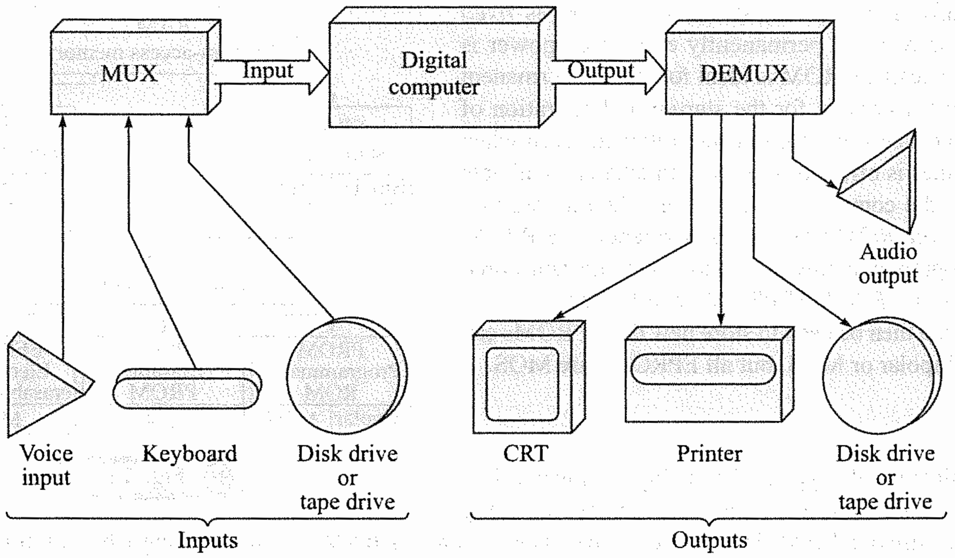
Memory

13

OBJECTIVES

- List the various forms of magnetic and optical memory and explain how each works
- Discuss memory addressing techniques
- Describe ROM, PROM, and EPROM and their characteristics and differences
- Compare the advantages and disadvantages of SRAM and DRAM and be familiar with the basic features of SRAM and DRAM chips
- Describe how content addressable memory works

The ability to store information (to remember) is an important requirement in a digital system. Circuits and/or systems designed specifically for data storage are referred to as memory. In the simplest application, the memory may be a flip-flop, or perhaps a number of flip-flops connected to form a register. In a larger system, such as a microcomputer, the memory may be composed of semiconductor memory chips. Semiconductor memories are composed of bipolar transistors or MOS transistors on an integrated circuit (IC), and are available in two general categories—read-only memory (ROM) and random-access memory (RAM). ROM and RAM memories can be constructed to store impressive amounts of data entirely within a computer system. Both programmed instructions and data are stored in a computer by means of ROM and RAM. But really large amounts of data (such as banking or insurance records) are generally stored using magnetic memory techniques. Magnetic memory includes the recording of digital information on magnetic tape, hard disks, and floppy disks. Magnetic storage systems are quite sophisticated and are usually externally accessed, as shown in Chapter 1 in Fig. 1.28 and repeated here for reference. However, in last two decades there has been tremendous growth in optical memory devices like compact disk, digital versatile disk etc. that gives low cost high capacity alternative storage solution.



13.1 BASIC TERMS AND IDEAS

Semiconductor Memory

Recent advances in semiconductor technology have provided a number of reliable and economical MSI and LSI memory circuits. The typical semiconductor memory consists of a rectangular array of *memory cells*, fabricated on a silicon wafer, and housed in a convenient package, such as a DIP. The basic memory cell is typically a transistor flip-flop or a circuit capable of storing charge and is used to store 1 bit of information. Memories are usually classified as either bipolar, metal oxide semiconductor (MOS), or complementary metal oxide semiconductor (CMOS) according to the type of transistor used to construct the individual memory cells. The total number of cells in a memory determine its *capacity*. For instance, a 1024 bipolar memory chip is a semiconductor memory that has 1024 memory cells, each cell consisting of a flip-flop constructed with the use of bipolar transistors. *Chip* is a term commonly used to refer to a semiconductor memory device. In general, faster operation is obtained with a bipolar memory chip, but greater packing density and thus reduced size and cost, as well as lower power requirements, are characteristics of MOS and CMOS memory chips.

Characteristics

The two general categories of memory, RAM and ROM, can be further divided as illustrated in Fig. 13.1. A dc power supply is required to energize any semiconductor memory chip. Once dc power is applied to a *static RAM (SRAM)*, the SRAM retains stored information indefinitely, without any further action. A *dynamic RAM (DRAM)*, on the other hand, does not retain stored data indefinitely; any stored data must be stored again (refreshed) periodically. Both SRAMs and DRAMs are used to construct the memory inside a microcomputer or minicomputer (see Fig. 1.34 in Chapter 1). DRAMs are used as the bulk of the memory, and high-speed SRAMs are used for a smaller, rapid-access type of memory known as *cache memory*. The cache is used to momentarily store selected data in order to improve computer speed of operation. SRAMs can be either bipolar or MOS, but all DRAMs are MOS.

The information (data) stored in a ROM is *fixed* and will be retained permanently even if dc power is removed. Clearly, a ROM is ideal for storing permanent instructions necessary for the startup and operation of a computer. These instructions are retained, even when the computer is off, and become immediately available each time the computer is turned on. Data stored in a *pogrammable ROM* (PROM) is permanent—a PROM can be programmed only once! However, the data stored in an *erasable PROM* (EPROM) can be "erased"; the EPROM can then be used to store new data. PROMs can be either bipolar or MOS, but all EPROMs are MOS.

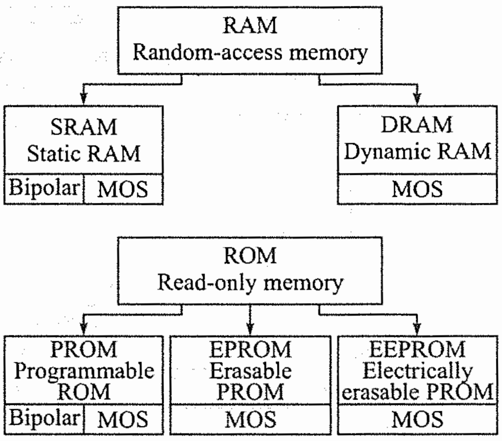


Fig. 13.1

RAM

A block diagram of a typical RAM chip is shown in Fig. 13.2a. An application in which data changes frequently calls for the use of a RAM. The logic circuitry associated with a RAM will allow a single bit of information to be stored in any of the memory cells—this is the write operation. There is also logic circuitry that will detect whether a 0 or a 1 is stored in any particular cell—this is the read operation. The fact that a bit can be written (stored) in any cell or read (detected) from any cell suggests the description *random access*. A control signal, usually called *chip-select* or *chip-enable*, is used to enable or disable the chip. In the read mode, data from the selected memory cells is made available at the output. In the write mode, information at the data input is written into (stored in) the selected cells. The address lines determine the cells written into or read from. Since each cell is a transistor circuit, a loss of dc power means a loss of data—a RAM that has this type of memory cell is said to provide *volatile storage*.

ROM

A typical ROM chip is shown in Fig. 13.2b. An application in which the data does not change dictates the use of a ROM. For instance, a "lookup table" that stores the values of mathematical constants such as trigonometric functions or a fixed program such as that used to find the square root of a number could be stored in a ROM. The content of a ROM is fixed during manufacturing, perhaps by metallization or by the

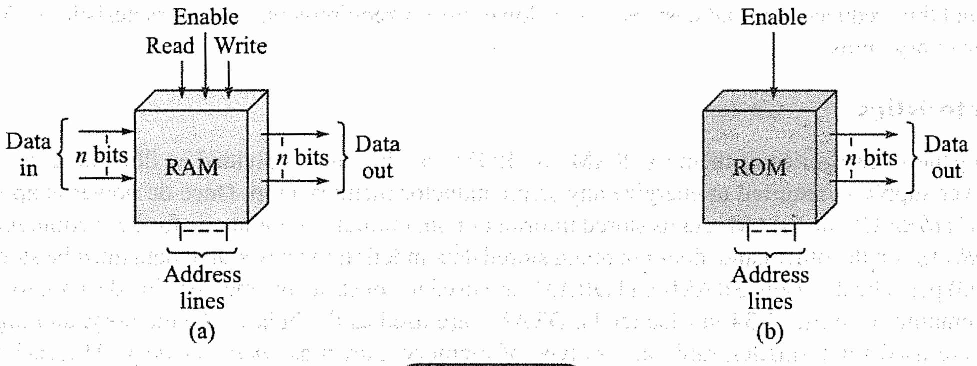


Fig. 13.2

presence or absence of a working transistor in a memory cell, by opening or shorting the gate structure, or by the oxide-layer thickness. A ROM is still random access, since there are logic circuitry and address lines to select any desired cell in the memory. When enabled, data from the selected cells is made available at the output. There is, of course, no write mode. Since data is permanently stored in each cell, a loss of power does not cause a loss of data, and thus a ROM provides *nonvolatile data storage.*

An application in which the data does not change but the required data will not be available until a later time suggests the use of a PROM, where the stored data can be set in the memory by writing into the PROM at the user's convenience. An application in which the data may change from time to time might call for the use of an EPROM.

Example 13.1 State the most likely type of semiconductor memory for each application: (a) main memory in a hand calculator; (b) storing values of logarithms; (c) storing prices of vegetable produce; (d) emergency stop procedures for an industrial mill now in the design stage.

Solution (a) RAM; (b) ROM; (c) EPROM; (d) PROM.

SELF-TEST

1. What is the operational difference between an SRAM and a DRAM?
2. What is an EPROM?
3. What is a cache memory?

13.2 MAGNETIC MEMORY

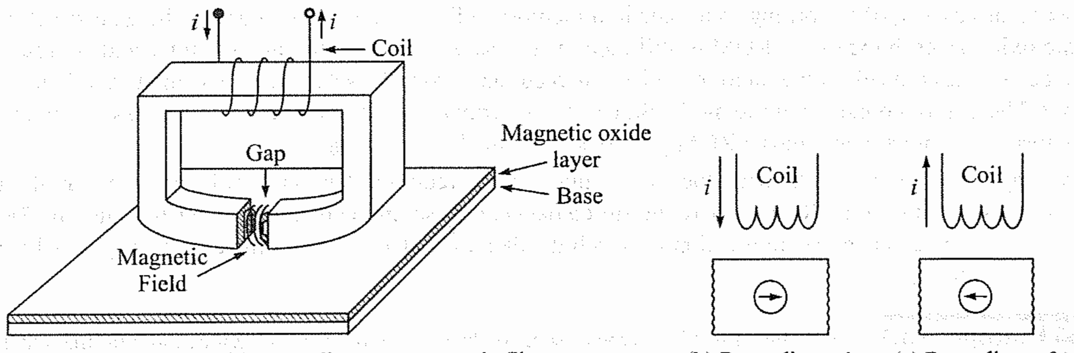
Magnetic tape, floppy disks, and hard disks are all capable of storing large quantities of digital data. A hard disk drive and a floppy disk drive are important components in nearly all microcomputer and minicomputer systems. Large reels of magnetic tape are economical and widely used mass storage components in large computer systems. The basic principle involved in each case is the magnetization of small spots in a thin film of magnetic material.

Magnetic Recording

Magnetic tape is produced by the deposition of a thin film of magnetic material on a long strip of plastic, which is then wound on a reel. Magnetic material deposited on a rigid disk forms the basis of a *hard disk;* the same material on a *semirigid* disk is used to construct a *floppy disk.* Digital information is recorded on any of these surfaces in essentially the same fashion.

A current *i* in the coil shown in Fig. 13.3a or, the next page will produce a magnetic field across the gap. A portion of this field will extend into the magnetic material below the gap, and the material will be magnetized with a fixed orientation. When the current is removed, a magnetized spot remains, as shown in Fig. 13.3b. Thus, information has been stored. If the current is reversed in direction, a spot will again be magnetized, but with the opposite fixed orientation, as shown in Fig. 13.3c. Clearly this is a binary system, and it can be used to store binary information. For example, one could "define" Part b of Fig. 13.3 as 1 (high) and Part c as 0 (low). Introducing current *i* to record a 0 or a 1 is *writing* (or recording or storing) data.

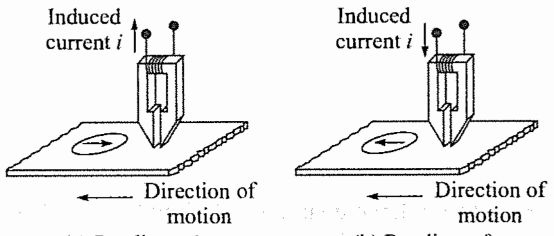
Now if a fixed, magnetized spot with a given orientation is moved past a gap as shown in Fig. 13.4a on the next page, a current with the direction shown will be induced in the coil. But if a magnetized



(a) Recording on a magnetic film (b) Recording a 1 (c) Recording a 0

Fig. 13.3

spot with the opposite orientation is moved past the gap, a current will be induced in the opposite direction, as shown in Fig. 13.4b. Detecting the orientation of the magnetized spot by measuring the induced current is *reading* information (1 or 0).



(a) Reading a 1 (b) Reading a 0

Fig. 13.4

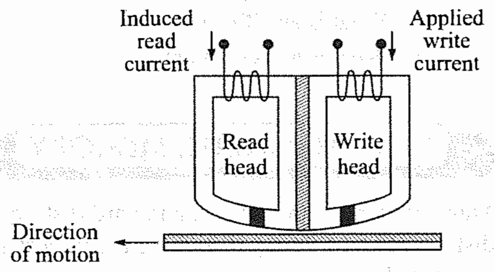
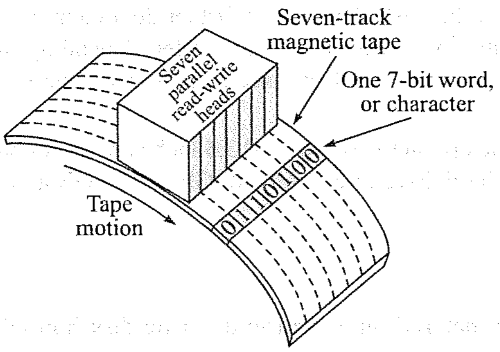


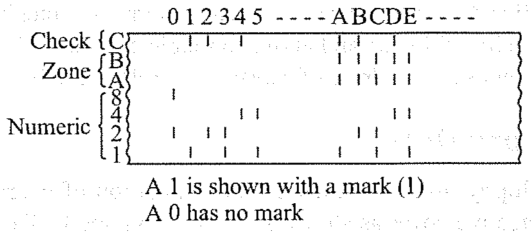
Fig. 13.5 **Dual read-write head**

The same magnetic read-write head in Fig. 13.3a can be used to write digital data or to read digital data. However, the dual read-write head in Fig. 13.5 is more common. Here's why. The tape or disk is moved under the heads in the direction shown. At time t_1, a spot is recorded under the write head. A short time later, at time t_2, this spot passes under the read head. It can then be read out and a check can be made to ensure that the correct data was in fact recorded.

Magnetic Tape

Either seven or nine dual read-write heads are connected in parallel for use with magnetic tape as illustrated in Fig. 13.6a. As the tape moves past the heads, data is read or written, 7 (or 9) bits at a time. In the 7-bit system, alphanumeric information is recorded in coded form, and there is 1 parity bit (even or odd). There are numerous coding schemes, but a portion of a commonly used IBM code is shown in Fig. 13.6b. In the 9-bit system, a data word is composed of 8 bits, and the ninth bit is for parity (either even or odd). Data can be stored in coded form or in straight binary form.


Seven-track magnetic tape
One 7-bit word, or character
Seven parallel read-write heads
Tape motion
0 1 1 1 0 1 1 0 1 0
(a) Magnetic tape recording



(b) A common 7-bit code

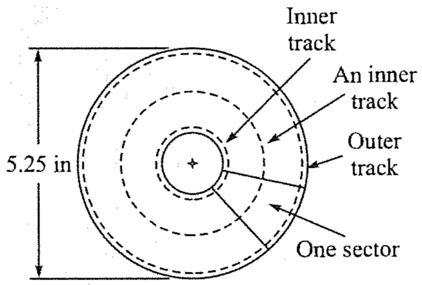
Fig. 13.6

Data storage on a magnetic tape is *sequential*. That is, data is stored one word after another, in sequence. To recover (read) data from the tape requires sequential searching. Clearly, the storage (or recovery) of data in a sequential system such as this requires considerably more time than storage (or recovery) using RAM. Tape is said to have a longer *access time* than RAM. Typical access times are measured in seconds, compared with nanosecond access times for RAMs.

Hard Disks

Magnetic material deposited on a rigid disk (usually aluminum) is the basis for a hard disk system. One or more of these disks are mounted in an enclosure similar to that shown in Fig. 13.7a. The hard disks used in small computer systems are typically 3.5 in. or 5.25 in. in diameter. Hard disk drives with 40 to 400 gigabyte capacities are common in microcomputer systems. The disk is rotated at speeds between 3600 to 7200 rpm and in high end servers up to 15000 rpm resulting in typical access time of 16 ms to 3.6 ms. Because of the relatively short access times and the high storage density, hard disks are widely used in all computer systems.

(a)



(b) Hard disk

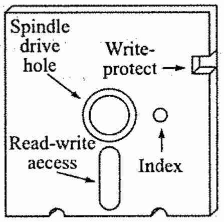
Fig. 13.7 **Hard disk system**

Information is stored in tracks (concentric rings) around the disk. The disk is further divided into sectors (pie-shaped sections), as shown in Fig. 13.7b. The number of tracks and sectors differ, depending on the computer system and on the individual manufacturer. The smaller hard disks used in microcomputer systems typically have 300 or so tracks.

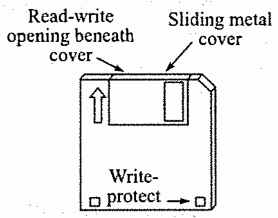
Besides internal Hard Disks, a modern computer has the option to use external 3.5 inch hard drives having capacity of 80 GB and above, portable external 2.5 inch hard drive of capacity 40 GB to 120 GB and palm size pocket hard drive of capacity 2.5 GB or 5 GB.

Floppy Disks

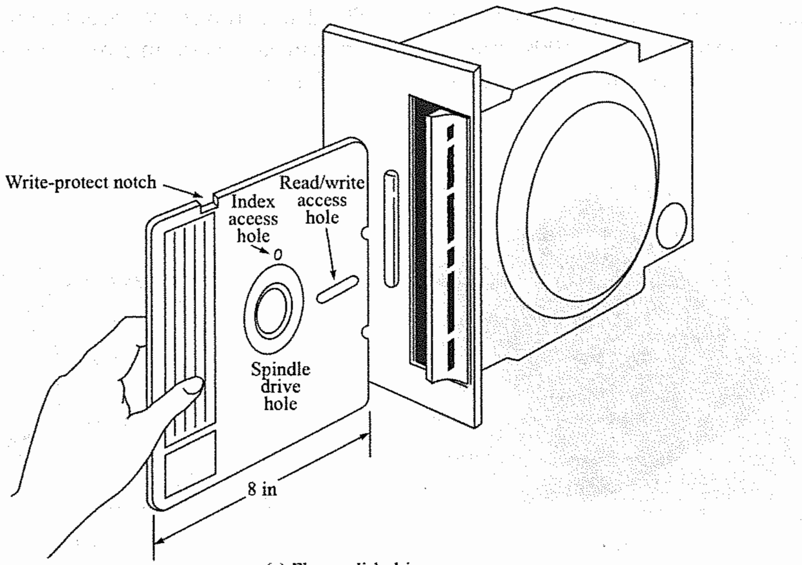
A floppy disk is formed by the deposition of magnetic material on a semirigid plastic disk housed in a protective cover as shown in Figs. 13.8a and b. The read-write opening provides access for the read-write head, and the index access hole allows the use of a photosensor to establish a reference position. When the write-protect notch is covered, data cannot be recorded on the disk, preventing accidental loss of data. Double-sided high-density 5.25-in disks have a capacity of 1.2 MB. Double-sided high-density 3.5-in disks have a capacity of 2.88 MB.



(a) 5.25-in floppy disk (minifloppy)



(b) 3.5-in floppy disk (microfloppy)



(c) Floppy disk drive

Fig. 13.8

As with hard disks, data is stored in tracks and the disk is divided into sectors. In IBM format, 5.25-in disks have 40 tracks per side and 3.5-in disks have 77 tracks per side. The IBM standard for sectors is nine.

The floppy disk is portable, and it must be inserted into a disk drive as shown in Fig. 13.8c. The drive unit consists of a single read-write head, read-write and control electronics, a drive mechanism, and a track-positioning mechanism. The spindle drive rotates the magnetic disk at a speed of 360 rpm. Access time is thus somewhat higher than the hard disk, being about 80 ms on average.

Note that, all the numbers that refer to maximum capacity, speed, given in this section or at other places are improving day by day by rapid technological advancements in this field.

SELF-TEST

4. Does the code in Fig. 13.6b have even or odd parity?
5. Magnetic tape provides inexpensive storage of large quantities of digital data. Why not use it, instead of RAM, in a microcomputer?
6. How can binary information be recorded on magnetic film?

13.3 OPTICAL MEMORY

Introduced in 1982 jointly by Philips and Sony for storing digital audio data, Compact Disk (CD) found its way into computer storage in 1985. There was no looking back since then and today we find different types of CDs flooding the market where binary data is optically coded. The memory capacity of a CD is in the range of 650–700 MB, i.e. nearly 500 times more than 1.44 MB magnetic floppy disk. Both come in movable data storage category with almost same price tag but data integrity in optical disk is maintained over much longer period of time. Its newer variety called Digital Versatile Disk (DVD) can store data from 4.7 GB to 17.1 GB depending on configuration and make. Thus, the growth in optical storage media has been spectacular in last two decades. In this section we'll first discuss how CDs store binary data, what differentiates one type of CD from the other and then we'll look into DVD features.

CD ROM

CD ROM or CD Read Only Memory devices are mass produced in factory using a stamp press technology. CD ROM drives uses LASER (Light Amplification by Stimulated Emission of Radiation) technology to read data from it. A semiconductor LASER generates a high intensity light wave of stable wavelength ≈ 780 nm. A lens system is used to direct the LASER towards the disk over approximately 1 micron diameter spot. Refer to simplified diagram of Fig. 13.9a. The intensity of the reflected light from metallic reflection layer, received by photo sensors gives the information of binary data is stored in CD. There are two different surfaces called *pit* and *land* from which reflection occurs. The pit is approximately 0.12 micron deep compared to land and reflected intensities are about 25% and more than 70% respectively (Fig. 13.9b). Every time laser beam travels from land to pit or pit to land there is a change in intensity of reflected light. This change is read as binary digit 1 and a constant intensity reflected light is interpreted as zero. The pit width is such that there is at least 2 and at most 10 zeroes between every 1. This is achieved by converting every 8-bit byte into a 14-bit value, a process called Eight to Fourteen Modulation (EFM). Such arrangement makes it easy for the read laser to detect bits and also helps in synchronization of internal clock as CDs are essentially self clocking. Data corresponding to a small portion of the track is shown above label in Fig. 13.9a.

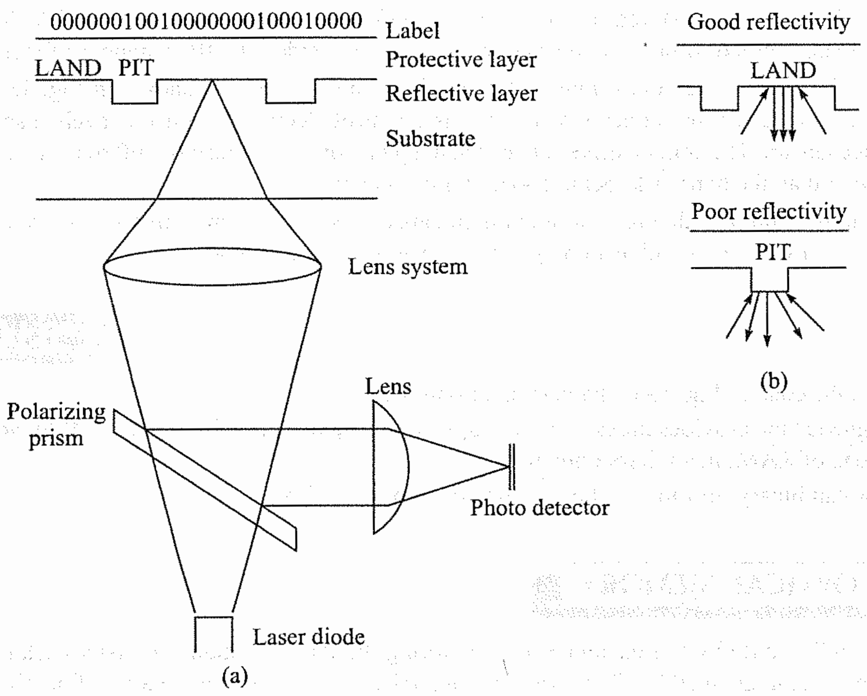


Fig. 13.9 **A compact disk reading system**

A compact disk normally comes in 12 cm diameter. The 1.2 mm thickness has four distinct parts. They are (a) label layer, (b) protective layer, (c) reflective layer, and (d) a transparent substrate layer on which land and pits are formed. The high reliability of CD comes from protection of data on one side by 10–20 microns thick protective lacquer layer and label and on the other side a tough approximately 1.2 mm thick polycarbonate layer. Thus, data integrity is maintained for years against most normal physical abuses and also for the fact that it is not susceptible to magnetic fields or radiations. Note that, small scratches on the surface of CD do not directly erase the data, but create additional areas of light scattering. This can make things difficult for drive's electronic, which is also much less sensitive to radial scratches than to the circumferential ones. The other reason that increases reliability of data stored in a CD is the ability to use efficient error correction codes. The data is stored in the form of a spiral of around 20000 windings totaling approximately 4.5 km of length and contains nearly 2 billion shallow pits on its surface. A macroscopic view of a part of CD is presented in Fig. 13.10.

You definitely have seen some kind of symbol like 48X, 52X etc. written on a CD. What is that? It gives the speed at which data is read from CD where 1X stands for 150 KB/Sec. Earlier versions of CD ROM drive below 12X were built on constant linear velocity (CLV) where motor had variable speed to maintain CLV. Present-day drives are based on constant angular velocity (CAV) where motor rotates at constant speed and this requires less seek time to access data from CD.

CD-R

CD-R or CD-Recordable allows user to write data but once. The CD-R drive has laser unit which uses higher intensity light wave for write operations than read. CD-R disk does not have pits and lands but a photosensitive organic dye (between reflective layer and polycarbonate substrate) that write laser heats

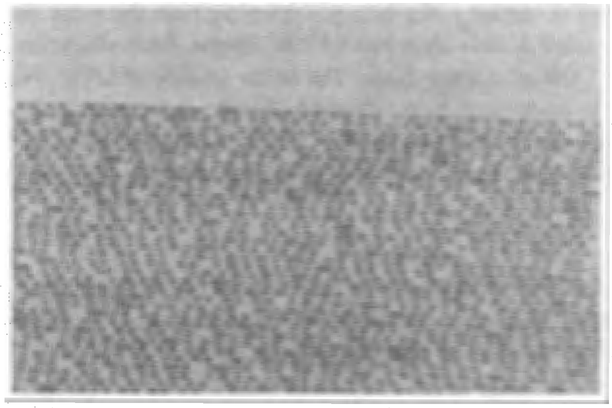


Fig. 13.10 A macroscopic view of a part of compact disk surface

to approximately 250°C. This melts or chemically decomposes the dye to form a depression mark in the recording layer in appropriate places. The places burnt have lower reflectivity of light. Thus read laser gets two different intensities on reflected light while reading the disk, similar to read operation of a CD ROM. Earlier versions of CD-R called WORM, abbreviation of write once read many times required data to be written in only one session or one go. Now it is possible to write data in CD-R in multiple sessions till it is completely filled. The writing speed of CD-R is much slower than the read speed.

CD-RW

CD-RW or CD Read Write, previously known as CD-Erasable gives user facility to write and erase data many times, unlike CD-R. CD-RW uses an active layer of Ag-In-Sb-Te (silver-indium-antimony-tellurium) alloy that has a polycrystalline structure making it reflective (reflectivity 25%). Writing data on disk uses highest power of laser that heats up selected spots to 500°–700°C. At this temperature the chemical structure liquefies losing its polycrystalline structure and on cooling solidifies to an amorphous state that has reduced reflectivity of 15%. The read process is like CD-ROM and CD-R that notes the difference in reflectivity of the reflecting surface.

To reverse the phase or erase data, the laser operates at a lower power setting and heats the active material to nearly 200 °C. This reverses the material from its amorphous to its polycrystalline state and then becomes reflective again. According to manufacturers, in a CD-RW the rewrite operations can be done 1000 times or more. The main drawback of CD-RW is very low reflectivity of the material and the difference between two levels is also not much. This often limits readability of these devices. Note that CD-Recordable drives often come with three different speed ratings, one speed for write-once operations, one for re-write operations, and one for read-only operations. The speeds are typically listed in that order, e.g. 12X/10X/32X. This means CPU and media-permitting, CD drive can write to CD-R disks at 12X speed, write to CD-RW disks at 10X speed and read from CD disks at 32X speed.

DVD

Digital Versatile Disk or Digital Video Disk, popular as DVD resemble compact disk in dimension and look but contains much higher storage space. DVD driver uses smaller wavelength (635 nm or 650 nm) and lower numerical aperture of lens system to read smaller dimension land and pits. Each side can have two layers from which data is read and in certain disks data is written on both the sides. Single sided, single layer has capacity of 4.7 GB, single sided double layer has 8.5 GB, double sided single layer has 9.4 GB while double sided double layer has 17.1 GB of storage space. The better quality of DVD output compared to CD comes from better channel coding, error correction scheme and of course a higher data transfer rate. Note that in DVD terminology, 1X refers to 1.32 MB/Sec unlike 150 KB/Sec of CD. Like CD, different varieties of DVD like DVD-R, DVD-RW, DVD-RAM are being developed and entering the market.

Handling Tips

Before we complete this section let us provide you some useful tips for handling CDs, the most used movable storage device of these days. Of course, the list is not exhaustive and the requirements come mainly from maintaining a good reflecting surface and physical balance of the CD.

(i) Handle disks by the outer edge or the center hole. Don't touch the surface of the disk to avoid leaving fingerprints and oil behind. Keep the disk free of dirt.

(ii) Clean dirt, smudges, and liquids from disks by wiping with a clean cotton fabric in a straight line radially outward from the center of the disk. Don't wipe in circles. The error correction codes on the disk can handle only small interruptions like scratch that travels across the spiral. You may clean stubborn dirt and foreign substances with 99% isopropyl alcohol or 99% methyl alcohol. First apply the cleaner to a cotton, then rub the cloth across the disk, taking care not to get any fluid on the label side of the disk.

(iii) Label the disk with a non-solvent-based felt-tip permanent marker. Beware of permanent markers that contain strong solvents. The use of adhesive labels is not recommended. If you use a label, don't try to remove or reposition it.

(iv) Never ever bend the disk. Flexing the disk can cause stress patterns to form in the polycarbonate, and if you stretch it far enough the reflective and recording layers get deformed. Store disks vertically. Over a long period, gravity will warp the disk if it's left flat.

(v) Store disks in a cool, dry, dark environment in which the air is clean to avoid corrosion. Keep it away from areas that are excessively hot or damp and also from direct sunlight and other ultraviolet light sources. Do not expose the disk to rapid changes in temperature or humidity.

SELF-TEST

7. What is the wavelength of laser used for reading CD-ROM?
8. What is the reflectivity of CD-ROM and CD-RW surfaces?
9. What is the capacity of a single sided double layer DVD ROM?

13.4 MEMORY ADDRESSING

Cell Selection

Addressing is the process of selecting one of the cells in a memory to be written into or to be read from. In order to facilitate selection, memories are generally arranged by placing cells in a rectangular arrangement of rows and columns as shown in Fig. 13.11a. In this particular case, there are m rows and n columns, for a total of $n \times m$ cells in the memory.

The control circuitry that accompanies the basic memory array is designed such that if one and only one row line is activated and one and only one column line is activated, the memory cell at the intersection of these two lines is selected. For instance, in Fig. 13.11b, if row A is activated and column B is activated, the cell at the intersection of this row and column is selected—that is, it can be read from or written into. For convenience, this cell is then called AB, corresponding to the row and the column selected. This designation is defined as the *address* of the cell. The activation of a line (row or column) is achieved by placing a logic 1 (or perhaps a logic 0) on it.

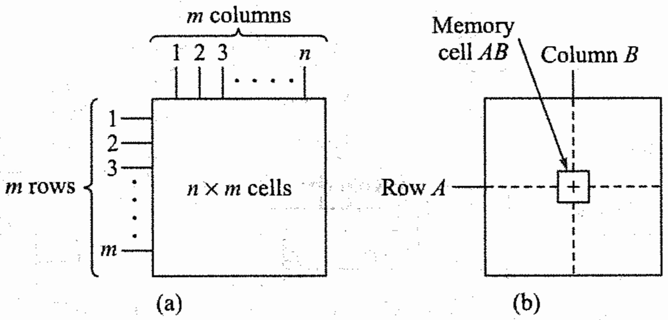


Fig. 13.11 **(a) A rectangular array of $m \times n$ cells, (b) Selecting the cell at memory address *AB***

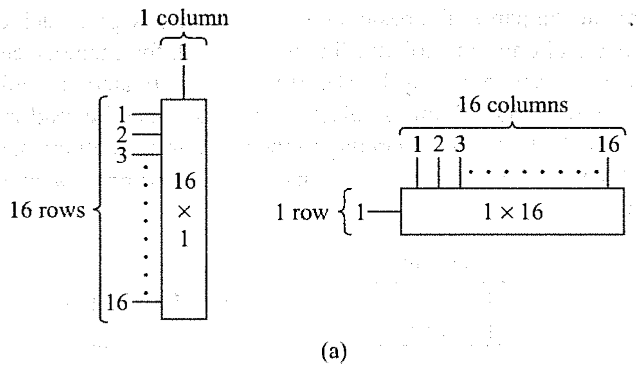
Matrix Addressing

Let's take a little time to consider the various possible configurations for a rectangular array of memory cells. The different rectangular arrays of 16 cells are shown in Fig. 13.12. In each of the five cases given, there are exactly 16 cells. The 16×1 and the 1×16 arrangements in Fig. 13.12a are really equivalent; likewise, the 8×2 and the 2×8 are essentially the same. So, there are really only three different configurations, each of which contain the exact same number of cells.

For any of the three configurations, the selection of a single cell still requires a single row and a single column to define a unique address. In Fig. 13.12a, a total of 17 address lines must be used—16 rows and 1 column, or 1 row and 16 columns. The minimum requirement in either case is really only 16 lines. However, either arrangement in Fig. 13.12b requires only 10 address lines—8 rows and 2 columns, or 2 rows and 8 columns. Clearly the best arrangement is given in Fig. 13.12c, since this configuration only requires 8 address lines—4 rows and 4 columns!

In general, the arrangement that requires the fewest address lines is a square array of n rows and n columns for a total memory capacity of $n \times n = n^2$ cells. It is exactly for this reason that the square configuration is so widely used in industry. This arrangement of n rows and n columns is frequently referred to as *matrix addressing*. In contrast, a single column that has n rows (such as the 16×1 array of cells) is frequently called *linear addressing*, since selection of a cell simply means selection of the corresponding row, and the column is always used.

For instance, a 74S201 is a 256-bit bipolar RAM, arranged in a 256×1 array. The IEEE symbol for the 74S201 ('S201) is given in Fig. 13.13 on the next page. Eight address lines ($A_0, A_1, \ldots, A_7$) are required to select one of the 256 cells. There are three chip select lines ($\overline{S}_1$, $\overline{S}_2$ and $\overline{S}_3$), all of which must be low in order to activate (select) the chip. When the $R/\overline{W}$ line is high, the data bit at input D is stored at the selected address. When the $R/\overline{W}$ line is low, the *complement* of the bit at the selected address appears at the $\overline{Q}$ output.



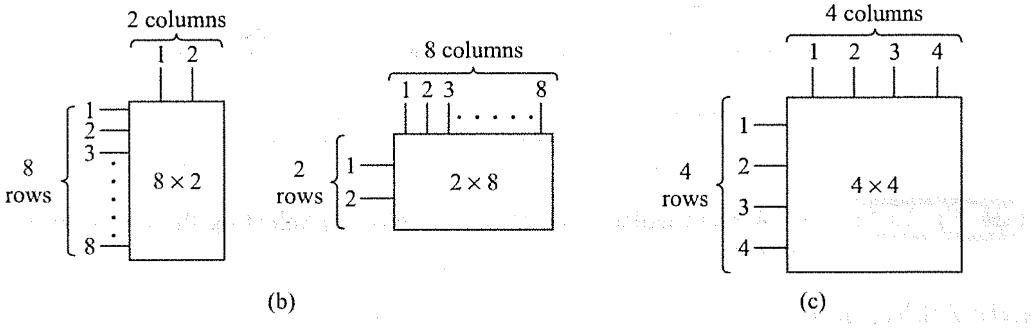


Fig. 13.12

The small triangle (▽) at the $\overline{Q}$ output means that the output is three-state (tri-state). We'll use this chip in Sec. 13.5.

Address Decoding

Take another look at the 4 × 4 memory in Fig. 13.12c. To select a single cell, we must activate one and only one row, and one and only one column. This suggests the use of two 1 of 4 binary to decimal decoders as shown in Fig. 13.14. Consider the selection of the cell at address 43 (row 4 and column 3). If $A_4 = 1$ and $A_3 = 1$, the decoder will hold the row 4 line high while all other row lines will be low. Similarly, if $A_2 = 1$ and $A_1 = 0$, the decoder will hold column 3 high and all other column lines low. Thus an input $A_4A_3A_2A_1 = 1110$ will select cell 43. We can consider A_4A_3 as a row address of 2 bits and A_2A_1 as a column address of 2 bits. Taken together, any cell in the array can be uniquely specified by the 4-bit address $A_4A_3A_2A_1$. As another example, the address $A_4A_3A_2A_1 = 0110$ selects the cell at row 2 and column 3 (address 23).

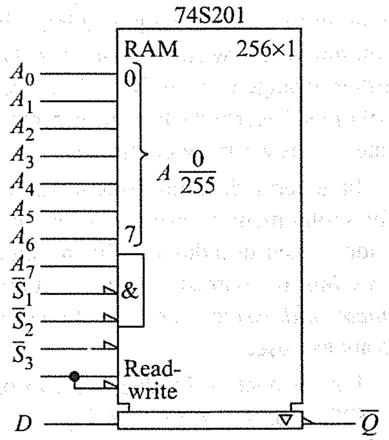


Fig. 13.13

The address decoders shown in Fig. 13.14 further reduce the number of address lines needed to uniquely locate a memory cell, and they are almost always included on the memory chip. Recall that a binary-to-decimal decoder having n binary inputs will select one of 2^n output lines. For instance, a decoder that has 3 binary inputs will have $2^3 = 8$ outputs, or a decoder having 4 inputs will have 16 outputs, and so on.

In general, an address of B bits can be used to define a square memory of 2^B cells, where there are $B/2$ bits for the rows and $B/2$ bits for the columns, as shown in Fig. 13.15. Notice that the total number of address bits B must be an even integer (2,4,6,8, ...). Since the input to each decoder is $B/2$ bits, the output of each decoder must be $2^{B/2}$ lines. So the capacity of the memory must be $2^{B/2} \times 2^{B/2} = 2^B$. For instance, an address of 12 bits can be used in this way for a memory that has $2^{12} = 4096$ bits. There will be 6 address bits providing $2^6 = 64$ rows and likewise 6 address bits providing 64 columns. The memory will then be arranged as a square array of $64 \times 64 = 4096$ memory cells.

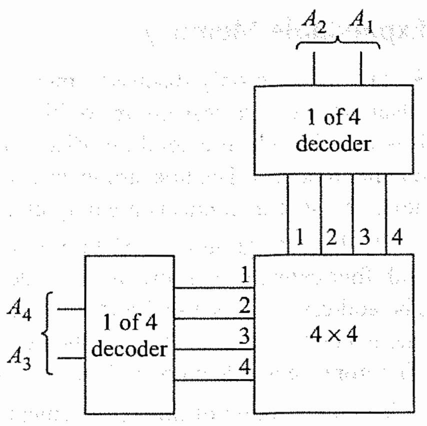


Fig. 13.14

You may have noticed that most commercially available memories have capacities like 1024, 2048, 4096, 16,384, and so on. The reason for this is now clear—all of these numbers are clearly integer powers of 2! Incidentally, a memory having 1024 bits is usually referred to as a 1K memory (1000 bits) simply for convenience. Similarly, a memory advertised as 16K really has 16,384 bits. 4K is really 4096, and so on.

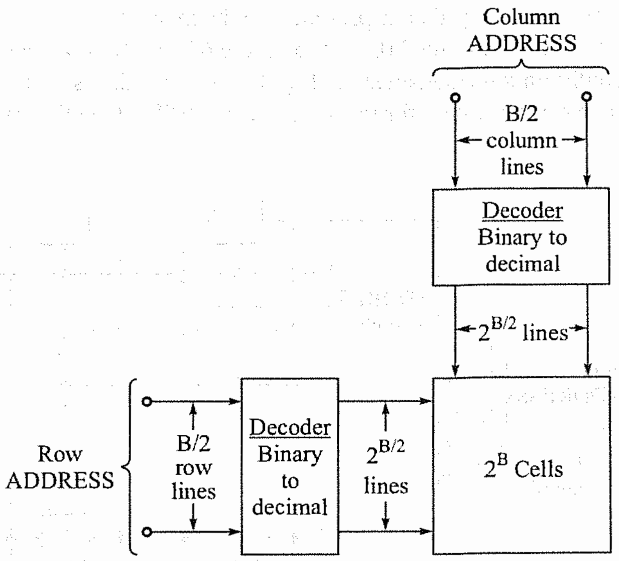


Fig. 13.15

Example 13.2 What would be the structure of the binary address for a memory system having a capacity of 1024 bits?

Solution Since $2^{10} = 1024$, there would have to be 10 bits in the address word. The First 5 bits could be used to designate one of the required 32 rows, and the second 5 bits could be used to designate one of the required 32 columns. Notice that $32 \times 32 = 1024$.

Example 13.3 For the memory system described in the previous example, what is the decimal address for the binary address 10110 01101? What is the address in hexadecimal?

Solution The first 5 bits are the row address. Thus row = 10110 = 22. The second 5 bits are the column address. So, column = 13. The decimal address is thus 22 13. In hexadecimal, this same address is 16 0D.

Expandable Memory

So far, we have only discussed memories that provide access to a single cell or bit at a time. It is often advantageous to access groups of bits—particularly groups of 4 bits (a nibble) and groups of 8 bits (a byte). It is not difficult to extend our discussion here to accommodate such requirements. There are at least two popular methods. The first simply accesses groups of cells on the same memory chip, and we discuss this idea next. The second connects memory chips in parallel, and we consider this technique in a following section.

The logic diagram for a 64-bit (16 × 4) bipolar memory is given in Fig. 13.16. There are 16 rows of cells with four cells in each row; thus the description (16 × 4). Each cell is a bipolar junction transistor flip-flop. The address decoder has 4 address bits and thus 16 select lines—one for each row. In this case, each select line is connected to all four of the cells in a row. So, each select line will now select four cells at a time. Therefore, each select line will select a 4-bit word (a nibble), rather than a single cell.

You might think of this arrangement as a "stack" of sixteen 4-bit registers. This is really a form of linear addressing, since the 4 address bits, when decoded, select one of the sixteen 4-bit registers. In any case, when data is read from this memory it appears at the four data output lines $\overline{D}_1$, $\overline{D}_2$, $\overline{D}_3$, and $\overline{D}_4$ as a 4-bit data word. Similarly, data is presented to the memory for storage as a 4-bit data word at input lines I_1, I_2, I_3, and I_4. The 74S89 and the 74LS189 both are 64-bit (16 × 4) bipolar scratch pad memories arranged in exactly this configuration (look ahead in Fig. 13.22). The idea is easily extended to memories that access a word of 8 bits (a byte) at a time—for instance, the TBP18S030 ROM discussed in the next section.

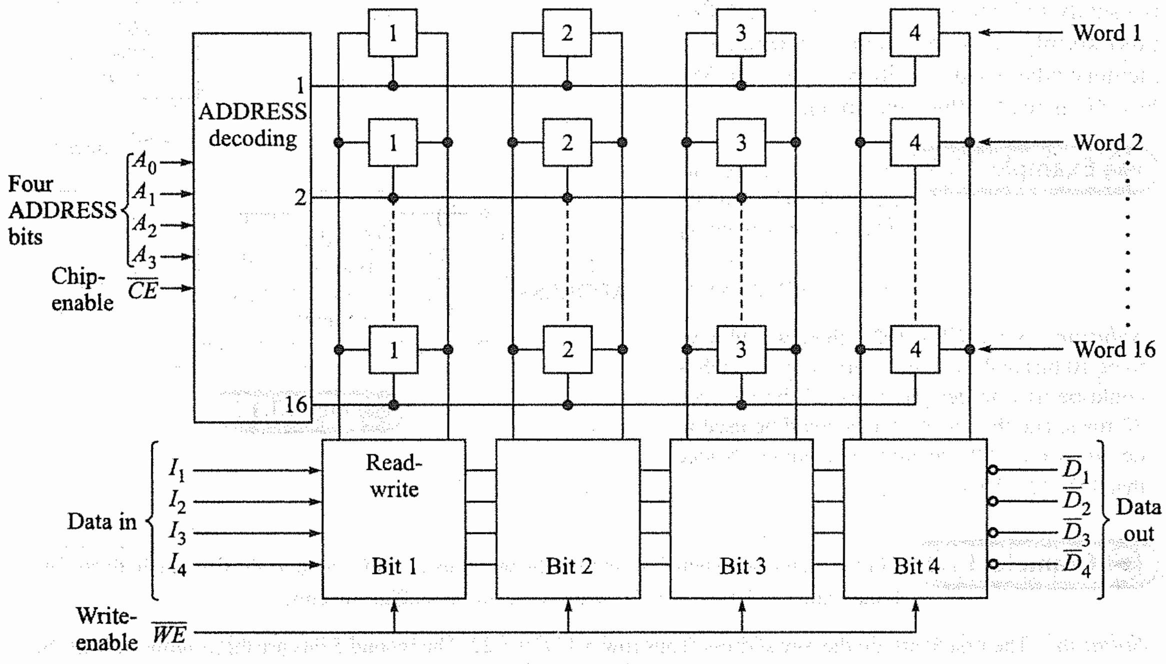


Fig. 13.16 64-bit (16 × 4) memory

SELF-TEST

10. What binary address will select cell 145 (decimal) in the 74S201 in Fig. 13.13?
11. The address applied in Fig. 13.14 is $A_4A_3A_2A_1$ = 1010. What cell is being accessed?

13.5 ROMs, PROMs, AND EPROMs

Having gained an understanding of memory addressing, let's turn our attention to the operation of a ROM. The term ROM is generally reserved for memory chips that are programmed by the manufacturer. Such a chip is said to be *mask-programmable*, in contrast to a PROM, which is said to be *field-programmable*—that is, it can be programmed by the user. EPROMs can be programmed, erased, and programmed again; they are clearly much more versatile chips than PROMs. Refer to Section 4.9 of Chapter 4 for a detailed discussion on its internal circuitry.

Programming

What exactly does programming a ROM, PROM or EPROM involve? It simply involves writing, or storing, a desired pattern of 0s and 1s (data). Each cell in the memory chip can store either a 1 or a 0. As supplied from the manufacturer, most chips have a 0 stored in each cell. The chip is then programmed by entering 1s in the appropriate cells. For instance, the content of every 4-bit word in a 64 × 4 chip is initially 0000. If the desired content of a word is to be 0110, then the two inner bit positions will be altered to 1s during programming.

In the case of a ROM, you must supply the manufacturer with the exact memory contents to be stored in the ROM. The Texas Instruments TMS4732 is a ROM having 4096 eight-bit words (a 4096 × 8 ROM). The logic diagram is given in Fig. 13.17. The 8-bit word length makes this NMOS (*n*-channel MOS) chip ideal for microprocessor applications. Texas Instruments will store user-specified data during manufacturing. The user must supply data storage requirements in accordance with detailed instructions given on the TMS4732 data sheet.

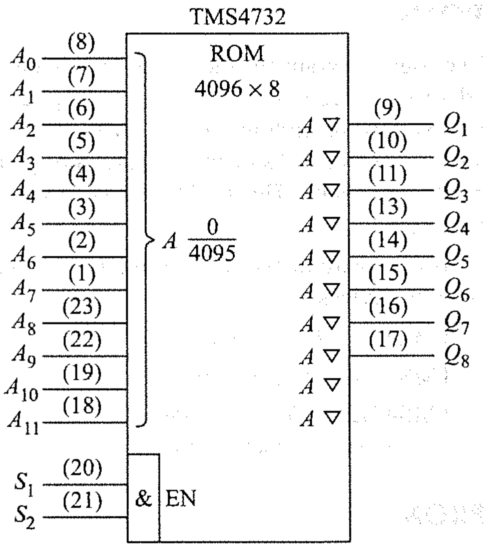


Fig. 13.17

The Texas Instruments TBP18S030 is a bipolar memory chip arranged as thirty-two 8-bit words (256 bits). The logic diagram for this user-programmable PROM chip is given in Fig. 13.18. Basically, the programming is done by applying a current pulse to each output terminal where a logic 1 must appear (be stored). The current pulse will destroy an *existing fuse link*. When the fuse link is present, the transistor circuit in that cell stores a 0. After the fuse link is destroyed, the circuit stores a 1. A typical programming sequence might be:

1. Apply the proper dc power supply voltage(s) to the chip; in the case of the TBP18S030, +5 Vdc. Disable the chip (the enable input is high).

2. Apply the address of the word to be programmed (A_0, A_1, A_2, A_3, A_4). For instance, to programm the word at address 14H (hex 14). apply

$$A_0A_1A_2A_3A_4 = 10100$$

3. To store the word $Q_0Q_1Q_2Q_3Q_4Q_5Q_6Q_7$ = 00101000:
   a. Ground output Q_2 and connect all other outputs to +5 Vdc through a 3.9-kΩ resistor. Raise the +5-Vdc supply to +9.25 Vdc and momentarily enable the chip. This will program a 1 in bit position Q_2.
   b. Repeat (a) for bit position Q_4. This will program a 1 in bit position Q_4.
4. Repeat steps 2 and 3 for each word to be programmed.

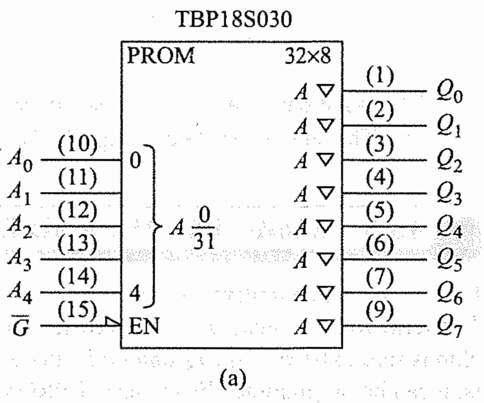


Fig. 13.18 (a) Logic symbol

ROMs

The logic diagram for the Texas Instruments TMS4732, a 4096 × 8 ROM, is given in Fig. 13.17. Twelve address bits are required, A_0, A_1, ... , A_{11} (2^{12} + 4096). There are two chip-enable inputs, S_1 and S_2. Both S_1 and S_2 must be high in order to enable the chip. Each of the eight data output lines is a three-state line (the small ∇ symbol). As mentioned previously, this chip is ideal for microprocessor applications because of the 8-bit word length. This ROM is mask-programmable, and data must be specified for the manufacturer before purchase.

Texas Instruments offers a number of other ROMs with larger memory capacity, all of which are LSI NMOS devices.

TMS4664:	8192 × 8-bit
TMS4764:	8192 × 8 bit
TMS47128:	16,384 × 8-bit
TMS47256:	32,768 × 8-bit

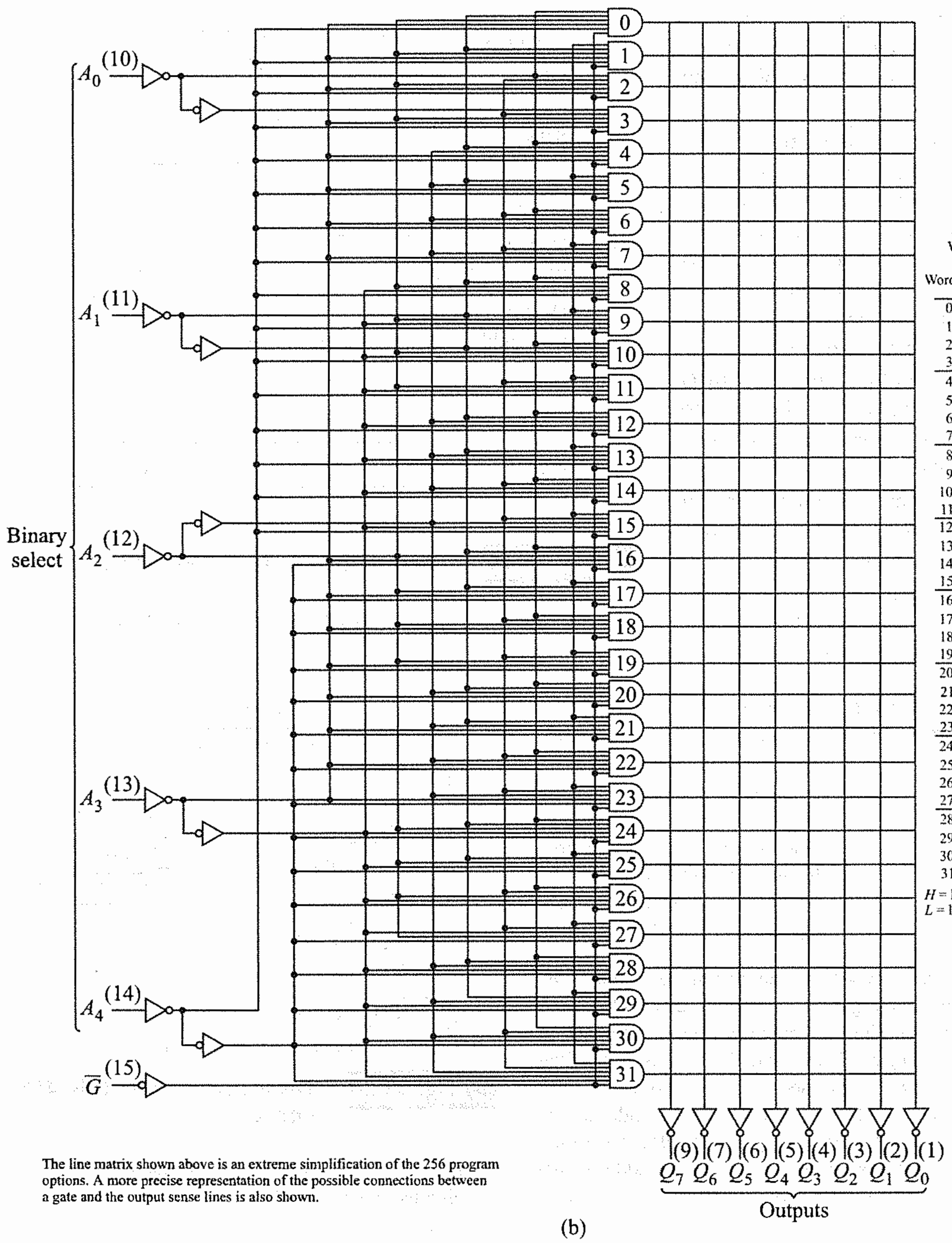
PROMs

The TBP18S030 is a 256-bit (32 × 8) PROM arranged as a stack of thirty-two 8-bit words. The 74S288 is an equivalent designation. As shown in Fig. 13.18, the 5 row address bits are labeled A_0, A_1, A_2, A_3, A_4 and the 8 output bits in a word are labeled Q_0, Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7.

Input $\overline{G}$ is used to enable or disable the entire set of 32 input decoding gates. When $\overline{G}$ is high, all the address decoding gates are inhibited and the memory chip is disabled, causing the eight output data bit lines to be high. When G is low, the data at the outputs will correspond to the 8-bit word in memory selected by the input address. On most memory chips there is a chip-enable or chip-select input line that performs the same function as $\overline{G}$.

Using the TBP18S030 PROM is relatively simple. First, since the logic circuits are TTL, a supply voltage and ground connections must be made. The data sheet calls for a nominal supply voltage of $+V_{CC}$ = 5.0 Vdc

Functional block diagram and word selection



Word select table

Word	Inputs				
	A_4	A_3	A_2	A_1	A_0
0	L	L	L	L	L
1	L	L	L	L	H
2	L	L	L	H	L
3	L	L	L	H	H
4	L	L	H	L	L
5	L	L	H	L	H
6	L	L	H	H	L
7	L	L	H	H	H
8	L	H	L	L	L
9	L	H	L	L	H
10	L	H	L	H	L
11	L	H	L	H	H
12	L	H	H	L	L
13	L	H	H	L	H
14	L	H	H	H	L
15	L	H	H	H	H
16	H	L	L	L	L
17	H	L	L	L	H
18	H	L	L	H	L
19	H	L	L	H	H
20	H	L	H	L	L
21	H	L	H	L	H
22	H	L	H	H	L
23	H	L	H	H	H
24	H	H	L	L	L
25	H	H	L	L	H
26	H	H	L	H	L
27	H	H	L	H	H
28	H	H	H	L	L
29	H	H	H	L	H
30	H	H	H	H	L
31	H	H	H	H	H

H = high level
L = low level

The line matrix shown above is an extreme simplification of the 256 program options. A more precise representation of the possible connections between a gate and the output sense lines is also shown.

(b)

Fig. 13.18 **(b) TBP18S030 PROM (74S288)**

on pin 16, with ground connected to pin 8. The inputs and outputs are all TTL-compatible. The eight data outputs are three-state (note the symbol ∇ at each output).

Now, all that is required is to apply the correct input address to read a desired 8-bit word and then take the $\overline{G}$ input line (select line) low. The TBP18S030 data sheet states an access time t_p of 12 ns (typical) and 25 ns (maximum). So, an 8-bit data word will be available at the outputs $Q_0 \dots Q_7$ within 25 ns after the falling edge of $\overline{G}$, as shown in Fig. 13.19a. The address lines should, of course, be stable during the time data is being read out of the memory. There are two output lines in Fig. 13.19a showing that a data line may transition low to high, or high to low. To save time and space, this idea is usually conveyed in a single waveform as seen in Fig. 13.19b—this single waveform is the equivalent of the two output waveforms above it in Fig. 13.19a.

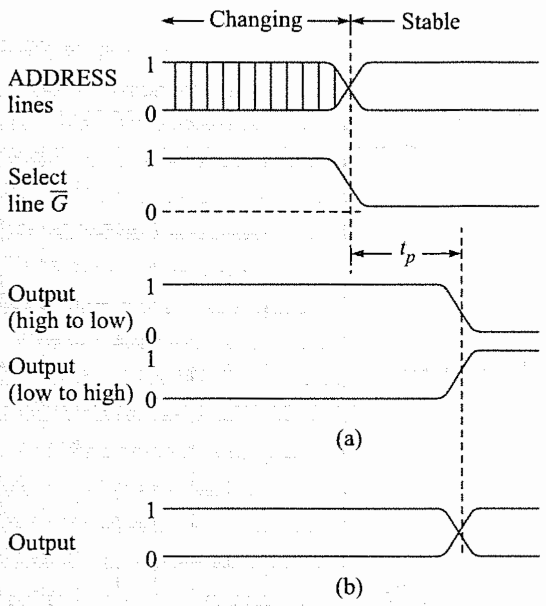


Fig. 13.19 Access time to for a TPB18S030 PROM

Example 13.4 The TTL LSI TBP24S10 is advertised as a 1024-bit PROM. Since $2^{10} = 1024$, it would seem to require 10 address bits, but the data sheet shows only 8 bits of address. Can you explain how the memory on this chip must be organized by looking at the logic diagram in Fig. 13.20?

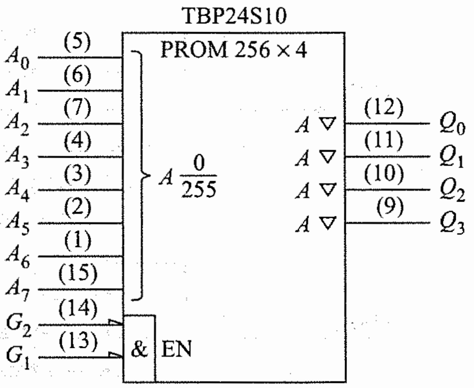


Fig. 13.20 Texas Instruments TPB24S10 PROM

Solution There are 4 bits appearing at the output of the chip, so it must be organized as 256 words of 4 bits each ($256 \times 4 = 1024$). The 1024 memory cells are arranged in a square consisting of 32 rows and 32 columns. Five of the address bits (*DEFGH*) are used to select one of the 32 rows ($2^5 = 32$). The 32 columns are divided into eight groups of 4 bits each. So, it is only necessary to select one of the eight groups, and this can be done with three address lines (*ABC*), since $2^3 = 8$. As an example, the address *HGFEDCBA* = 10110 110 will select row 10110 = 22 and the 4 bits (four columns) in group 110 = 6.

EPROMs

One disadvantage of a PROM is that once it is programmed, the contents are stored in that memory chip permanently—it can't be changed; a mistake in programming the chip can't be corrected. The EPROM overcomes this difficulty.

The EPROM has a structure and addressing scheme similar to those of the previously discussed PROM, but it is constructed using MOS devices rather than bipolar devices. Many MOS EPROMs are TTL-compatible, and even the technique used to program the chip is similar to that used with a bipolar memory. The only difference is really the mechanism for permanently storing a 1 or a 0 in an MOS memory cell.

The current pulse used to store a 1 when programming a bipolar PROM is used to destroy ("burn out") a connection on the chip. The same technique is used to program an MOS-type EPROM, but the current pulse is now applied for a period of time (usually a few milliseconds) in order to store a fixed charge in that particular memory cell. This stored charge will cause the cell to store a logic 1.

The interesting thing about this phenomenon is that the charge can be removed (or erased), and the cell will now contain a logic 0! Furthermore, the process can be repeated. The memory cells are "erased" by shining an ultraviolet light through a quartz window onto the top of the chip. The light bleeds off the charge and all cells will now contain 0s. The requirements for programming and erasing an EPROM vary widely from chip to chip, and data sheet information must be consulted in each individual case.

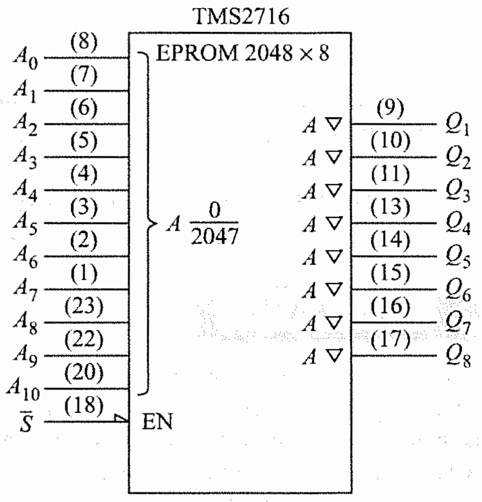


Fig. 13.21 2716 EPROM

The logic diagram for a 2716, a 16K (2K × 8) EPROM, is given in Fig. 13.21. There are actually 2048 words, 8 bits each, for a total storage capacity of 16,384 bits. The chip is completely TTL-compatible and has an access time of less than 450 ns. The 11 address bits will uniquely select one of 2048, 8-bit words (2^{11} = 2048), and the selected word will appear at the data output lines if chip-select is low. The 2732 is a 32K (4K × 8) EPROM that is pin-compatible with the 2716—it simply has twice the memory storage. Likewise, the 2764 is a 64K (8K × 8) EPROM.

As a matter of fact, the logic diagram in Fig. 13.21a is virtually the same for any ROM, PROM, or EPROM. Essentially the only variation is the total number of address inputs to accommodate the number of bits in the cell matrix. As an example, a CMOS EPROM with essentially the same logic diagram is the 27C512. But this chip has 16 address input lines and a 534,288-bit cell matrix, organized as 65,536 words, each 8 bits in length. This is most impressive when you consider that there are over a half-million bits of information stored in a 28-pin package that measures less than 1.5 in. in length and about 0.5 in. in width!

EEPROM, Flash Memory

Electrically Erasable Programmable Read Only Memory (EEPROM) is similar to EPROM as far as writing into memory is considered, i.e. effecting a current pulse to store charge. The erasing, however, is different and is done by removing the charge and sending a pulse of opposite polarity. There are two types of EEPROM—parallel and serial. Parallel EEPROM is faster, costlier and comes in 28xx family. Their pinout and functioning is similar to 27xx EPROM family. Serial EEPROM is slower, cheaper, uses lower number of pins and comes in 24xx family.

Flash memory is a further advancement on EEPROM. This, too, writes and erases data electrically—can be both parallel and serial type. The number of write/erase cycle is finite and often, there is a separate management scheme to take note of this. There is an internal voltage generation block that takes single voltage supply and generates different voltages required for writing and erasing. Different manufacturers have created different standards for flash memory chip which differ in pinout, memory organization, etc. Intel family chips are 28Fxxx while AMD chips are numbered as 29Fxxx.

SELF-TEST

12. What does it mean to say that a chip is mask-programmable?
13. What is the meaning of the small triangle on each output line of the TMS4732 in Fig. 13.17?
14. The 74S288 in Fig. 13.18 is programmed with 1011 0001 in word 20. The desired content at this word address is 1011 1001. Can this be corrected?

13.6 RAMs

The basic difference between a RAM and a ROM is that data can be written into (stored in) a RAM at any address as often as desired. Naturally data can be read from any address in either a RAM or a ROM, and the addressing and read cycles for both devices are similar. The characteristics of both bipolar and MOS "static" RAMs are discussed in this section.

A static RAM (SRAM) uses a flip-flop as the basic memory cell (either bipolar or MOS) and once a bit is stored in a flip-flop, it will remain there as long as power is available to the chip—essentially forever—thus the term "static." On the other hand, the basic memory cell in a "dynamic" RAM (DRAM) utilizes stored charge in conjunction with an MOS device to store a bit of information. Since this stored charge will not remain for long periods of time, it must periodically be recharged (refreshed), and thus the term "dynamic" RAM. Both static and dynamic RAMs are "volatile" memory storage devices, since a loss of power supply voltages means a loss of stored data. In this section we discuss SRAMs in detail that explains how a RAM unit works and how several RAM chips can be combined together to expand the memory capacity, For this purpose we'll use mostly the TTL devices however a brief discussion of MOS based SRAM and DRAM will also be presented.

The 7489

The 7489 shown in Fig. 13.22 is a TTL LSI 64-bit RAM, arranged as 16 words of 4 bits each. Holding the memory-enable ($\overline{ME}$) input low will enable the chip for either a read or a write operation, and the four data address lines will select which one of the sixteen 4-bit word positions to read from or write into. Then, if the write-enable ($\overline{WE}$) is held low, the 4 bits present at the data inputs (D_1, D_2, D_3, D_4) will be stored in the selected address. Conversely, if $\overline{WE}$ is high, the data currently stored in the memory address will be presented to the four data output lines ($\overline{Q}_1$, $\overline{Q}_2$, $\overline{Q}_3$, $\overline{Q}_4$). Incidentally, the outputs are open-collector transistors, and a pull-up resistor from each output up to $+V_{CC}$ is normally required. The operations for this chip are summarized in the truth table in Fig. 13.22.

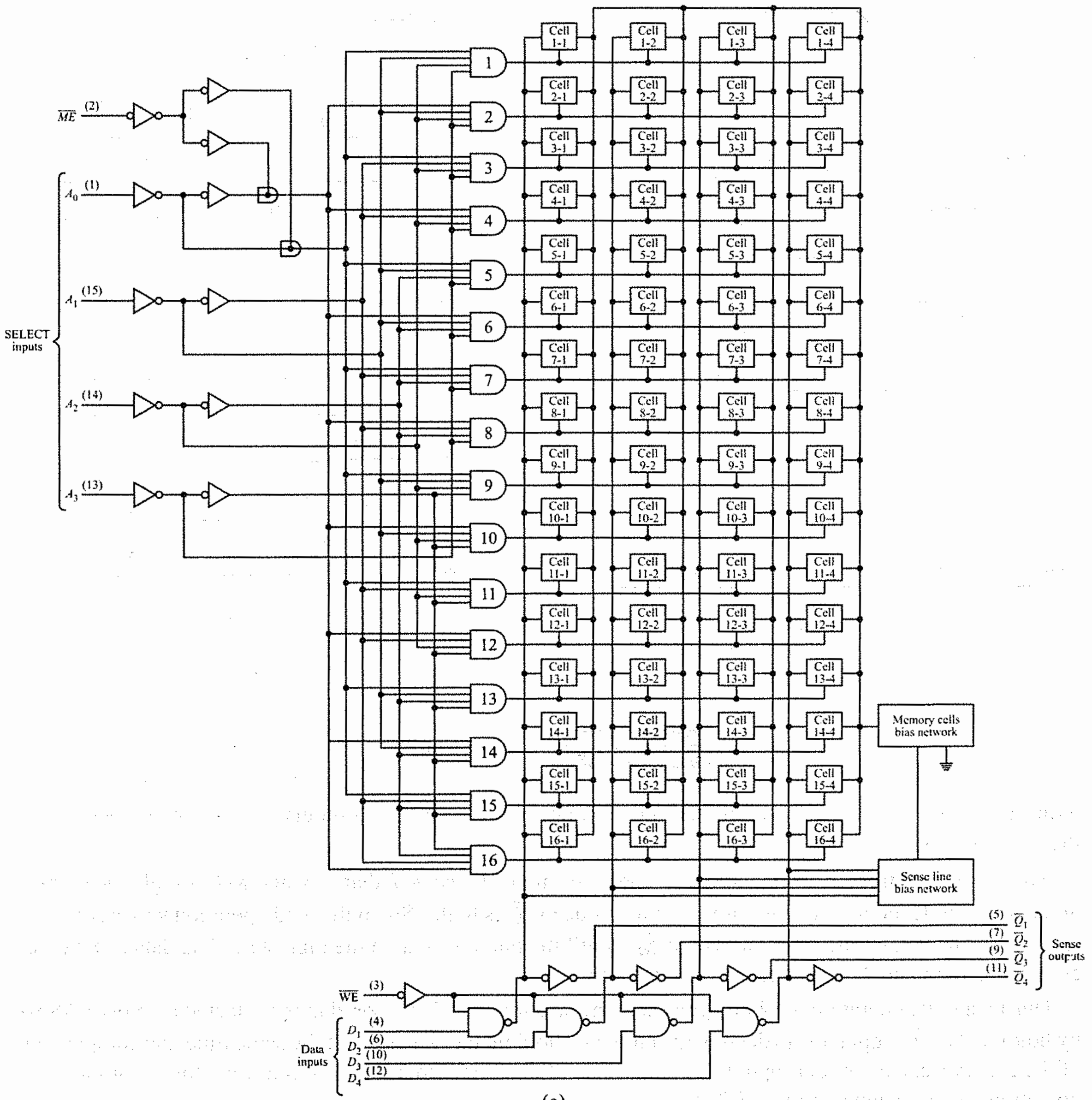
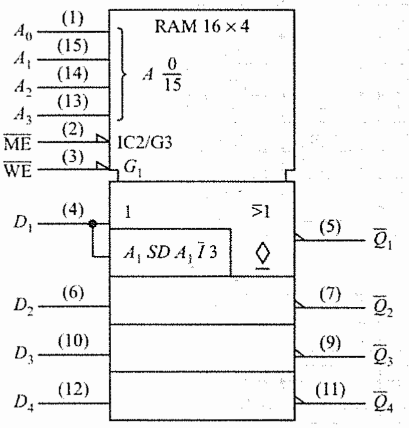


Fig. 13.22 **7489, 64-bit RAM**

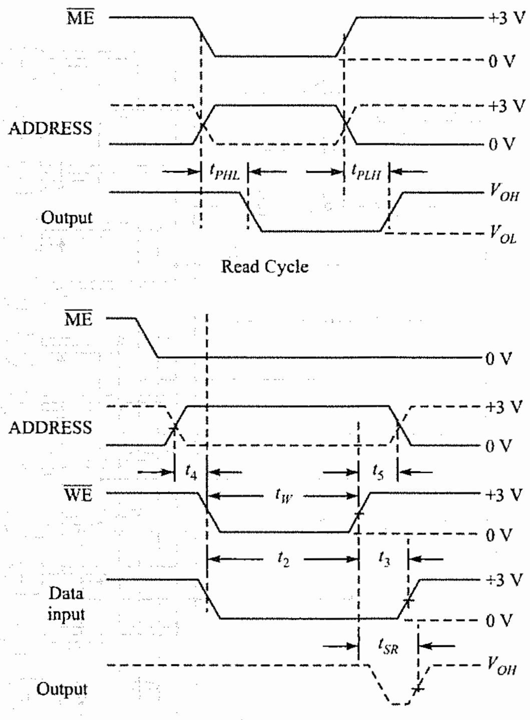
The read operation is no different from that for a ROM. For this chip, simply hold $\overline{ME}$ low and $\overline{WE}$ high, and select the desired address. The 4-bit data word then appears at the "sense" outputs. The timing for a read operation is shown by the waveforms in Fig. 13.22d. The propagation delay time t_{PHL} is that period of time from the fall of $\overline{ME}$ until stable data appears at the outputs—the data sheet gives a maximum value of 50 ns, with 33 ns typical. Naturally the address input lines must be stable during the entire read operation, beginning



(b)

$\overline{ME}$	$\overline{WE}$	Operation	Condition of outputs
L	*L*	Write	Complement of data inputs
L	*H*	Read	Complement of selected word
H	*L*	Inhibit storage	Complement of data inputs
H	*H*	Do nothing	High

(c)



(d)

Fig. 13.22 (Continued)

with the fall of $\overline{ME}$. Notice carefully that the data appearing at the four outputs will be the *complement* of the stored data word!

You will notice from the truth table that when the chip is *deselected*, that is, when $\overline{ME}$ is high, the outputs all go to a high level, provided we are in a read mode ($\overline{WE}$ is high). So, in the read operation waveforms, the time t_{PLH} is the delay time from the rise of $\overline{ME}$ until the outputs assume the high state. The data sheet gives 50 ns maximum and 26 ns typical for this delay time.

During a write operation the 4 bits present at the data inputs will be stored in the selected memory address by holding the $\overline{ME}$ input low (selecting the chip) and holding the $\overline{WE}$ low. At the same time, the complement of the data present at the four input lines will appear at the four output lines. Timing waveforms for the write operation are also shown in Fig. 13.22d.

Let's look carefully at the timing requirements for the write cycle. First, the $\overline{WE}$ must be held low for a minimum period of time in order to store information in the memory cells—this is given as time t_W on the waveforms, and the data sheet calls for 40 ns minimum. Memory-enable selects the chip when low, and is allowed to go low coincident with or before a write operation is called for by $\overline{WE}$ going low.

Next, the data to be written into memory must be stable at the data inputs for a minimum period of time before $\overline{WE}$ and, also for a minimum period of time after $\overline{WE}$. The time period prior to $\overline{WE}$ is called the *data-setup time* t_2. This time is measured from the end of the write-enable signal back to the point where the data must be stable. The data sheet calls for 40 ns, and in this case it is the same as t_W. Also, the data inputs must be held stable for a period of time after $\overline{\text{WE}}$ rises—this is called the *data-hold time* t_3, and the data sheet calls for 5 ns minimum.

The address lines must also be stable for a period of time before as well as after the $\overline{\text{WE}}$ signal. The time period before $\overline{\text{WE}}$ is called the *address-setup* or *select-setup time* t_4. This time is measured from the fall of $\overline{\text{WE}}$ back to the point where the input address lines must be stable; the data sheet calls for 0.0 ns minimum. In other words, the address lines are allowed to become stable coincident with or before $\overline{\text{WE}}$ goes low. The address lines must also be stable for a period of time after the rise of $\overline{\text{WE}}$; this is called the *address-hold* or *select-hold* time t_5, and the data sheet calls for 5 ns minimum.

Finally, after a write operation, if the chip is deselected ($\overline{\text{ME}}$ goes high), the outputs will return to a high state. The maximum time for this to occur is the sense-recovery time t_{SR}, given as 70 ns maximum on the data sheet.

The operation of a 7489 ns straightforward and easy to understand; therefore it is a good chip to study in elementary discussions of RAMs. It can be used to construct memories having larger capacities by connecting chips in parallel, but it's not too practical when we wish to consider memories of 16K, 32K, ... , 256K, 512K, and so on. Nevertheless, the time spent studying this chip is well invested since the fundamentals of addressing and the read and write operations are essentially the same for all static RAMs. So, with these fundamentals in mind, let's take a look at some chips that have more memory capacity.

The 74S201

The block diagram in Fig. 13.23 can be used to describe the operation of most SRAMs. Most of these are constructed with n address lines that will uniquely select only one of the 2^n cells in the memory array—that is, selection is 1 bit at a time. There will be a chip-enable control (CE), a write-enable (WE), and a provision for a single input data bit (D_i) and a single output data bit (D_o).

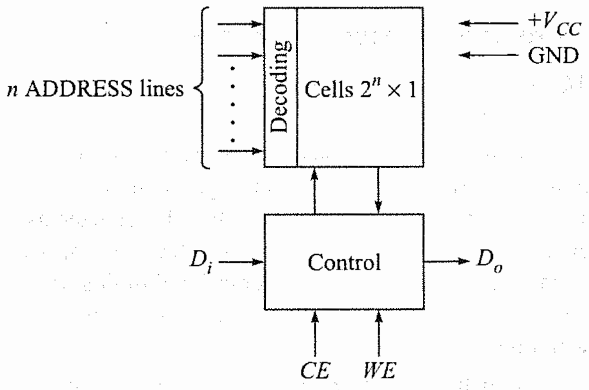
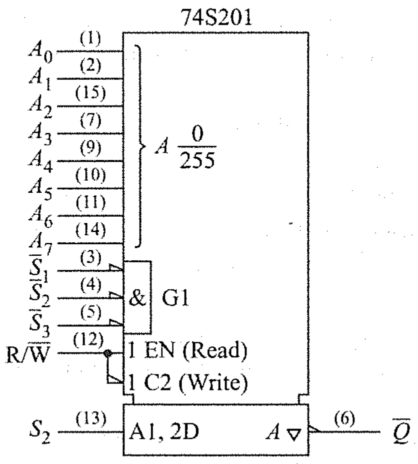


Fig. 13.23 Generalized block diagram for a static RAM

For instance, the 74S201 in Fig. 13.24 is a 256-bit RAM, organized as 256 words, each 1 bit in length. The 256 cells are arranged in a square array of 16 rows and 16 columns. The 8 address bits ($2^8 = 256$) are divided into 4 bits that are decoded to select one of the 16 rows and 4 bits that are decoded to select one of the 16 columns. There is a single input data bit (D), a single output data bit ($\overline{Q}$), and a read-write line (R/$\overline{\text{W}}$). There are three memory-enable inputs ($\overline{S}_1$, $\overline{S}_2$, $\overline{S}_3$), and all three of them must be low to select or enable the chip.

The truth table shows that if the chip is enabled, a write cycle is initiated by holding R/$\overline{\text{W}}$ low, or a read cycle can be initiated by holding R/$\overline{\text{W}}$ high. If any or all of the read-write inputs are high, the chip is inhibited and the output goes to a high-impedance state. Naturally the proper timing must be observed as defined by the timing waveforms; you will see that the timing requirements are very similar to the previously described 7489.



(a) Logic symbol

Write-enable pulse width (minimum)		65 ns
Setup time	ADDRESS before write	65 ns
	Data before end of write	65 ns
	Chip-select before end of write	65 ns
Hold time	ADDRESS after write	0 ns
	Data after write	0 ns
	Chip-select after write	0 ns

(b) Recommended timing

Fig. 13.24

Example 13.5 Using the information in Fig. 13.24, determine how long the address lines for the 74S201 must be held before $R/\overline{W}$ goes low and after $R/\overline{W}$ goes high.

Solution The setup time, address to write-enable, is 0.0 ns. The hold time, address from write-enable, is 0.0 ns. Therefore, the address lines must be stable from the fall of $R/\overline{W}$ until the rise of $R/\overline{W}$.

Formation of Memory Banks

Memory bank is the concept of increasing memory's capacity by connecting more than one memory block in series, parallel or both.

Now that we understand the operation of the 74S201 (abbreviated as '201), it is a simple matter to use multiple '201 chips to construct larger memories. For instance, we can connect four '201 chips in parallel as shown in Fig. 13.25 to construct a RAM organized as 256 words, each 4 bits in length. Connecting eight '201 chips in parallel will form a memory having 8-bit words, and so on. The nice thing about connecting chips in parallel like this is that the control and timing are exactly the same as if there were only a single chip. The only difference is that there are 4 data bits in and 4 data bits out (or 8 in and 8 out), all of which are in parallel with one another.

As a matter of fact, even larger memories can be constructed by connecting basic chips such as the '201 in both series and parallel. For instance, thirty-two '201 chips are connected in a 4 × 8 matrix in Fig. 13.26 to form a memory having one thousand, twenty-four 8-bit words. This configuration requires a 10-bit address: 2 bits can be used to select one of the four rows of eight '201s, and the remaining 8 bits will be wired in parallel to all the chips; they will work exactly as for the two-hundred fifty-six 4-bit word memory in Fig. 13.25. This concept can be continued, of course, but it becomes somewhat impractical with larger memory requirements, especially since there are MOS chips readily available with greater memory capacity.

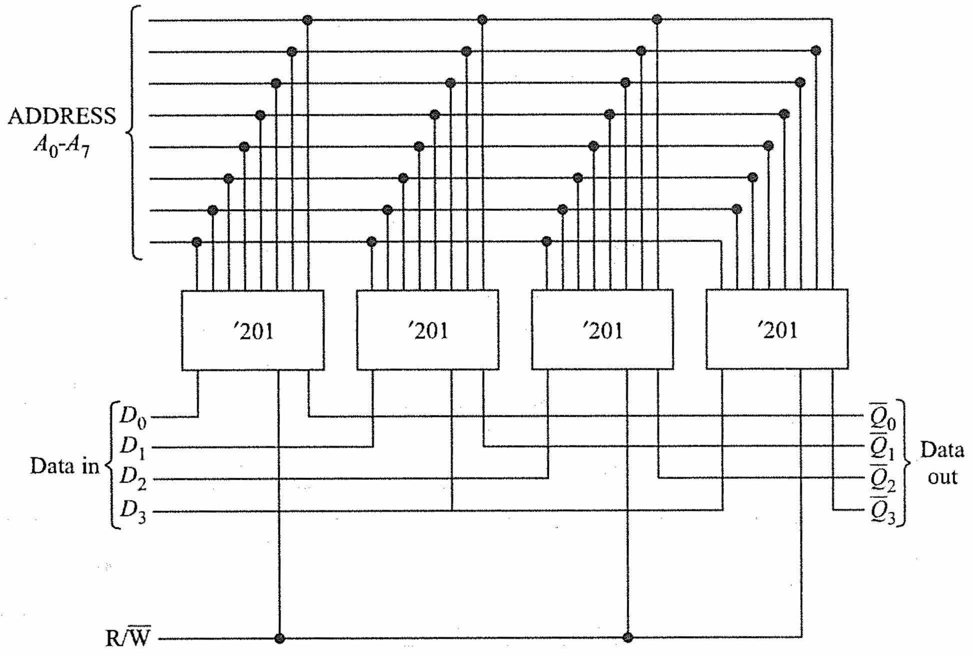


Fig. 13.25 **Four 74S201's arranged as a 1024-bit memory having 256**

SRAMs

A very popular and widely used MOS memory chip is the 2114. This is an SRAM having 4096 bits arranged as 1024 words of 4-bits each. The organization of this chip is quite similar to that of the 7489 shown in Fig. 13.22, but notice that the 2114 is sixteen times larger! Nearly all SRAMs larger than 1024 bits are MOS types.

The basic memory is arranged as 64 rows and 64 columns for a total of $64 \times 64 = 4096$ bits. Six address bits (A_3 through A_8) are used to select one of the 64 rows ($2^6 = 64$). The 64 columns are divided into 16 groups of 4-bit words, and four address bits (A_0, A_1, A_2 and A_9) are used to select one of these 16 groups. A 10-bit address will then select a single 4-bit word from 64 rows and 16 columns, to provide a memory of $64 \times 16 = 1024$ four-bit words.

A basic SRAM cell or latch is shown in Fig. 13.27a. It consists of a back-to-back inverter that latches on to a particular state of logic 0 or 1. The two pass transistors enable writing and reading from two bit lines. Both the transistors are 'on' for both reading and writing when this cell is selected after decoding the address. The bit lines during writing operations write its value into the latch while during reading sense the latch state. A typical SRAM requires six transistors per bit of memory—two pass transistors and two transistors each of the inverter. However, some implementations use only a single transistor per inverter with a total requirement of four transistors per bit.

DRAMs

A typical DRAM is essentially the same as the previously discussed SRAM chip, with the exception of the required refresh cycle. The 4116 is a widely used 16K (16,384 × 1) DRAM available from a number of

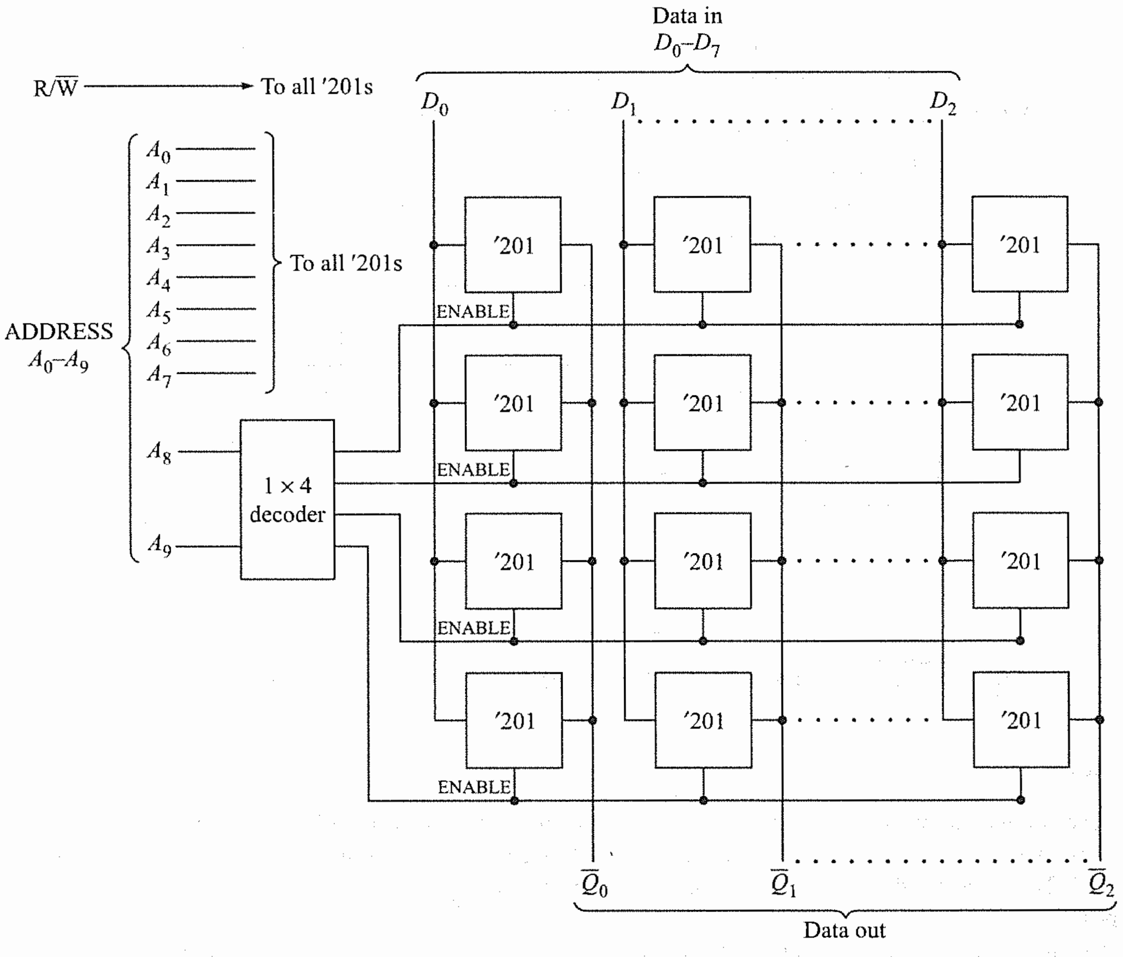


Fig. 13.26

different sources, such as Mostek (MK4116), Motorola (MCM4116), and Texas Instruments (TMS4116). Note that, 4116 must be refreshed at least once in every 2 ms. There is another widely used DRAM, the 4164, organized as a 64,536 × 1 chip. The operation of this chip is quite similar to 4116. The 64K DRAM is available under following part numbers: Texas Instruments TMS4164, Motorola MCM6665, Intel 2164, etc.

A basic DRAM cell is shown in Fig. 13.27b. A capacitor is used as a storage element, as it can store electrical charge but for a limited amount of time. This requires periodic *refresh* to replenish the charge and thus the RAM is always *dynamic*. The cell is selected by turning on the pass transistor. The bit line is used both for writing and reading (sensing). A faster writing ability requires the capacitor to be charged faster which in turn discharges the capacitor quickly requiring quicker refresh cycle. Compared to SRAM, DRAM uses less number of components that require less space, increasing the *packing density*. Also it is much less expensive. But SRAM scores over DRAM on a very important area. DRAM has a very high access time due

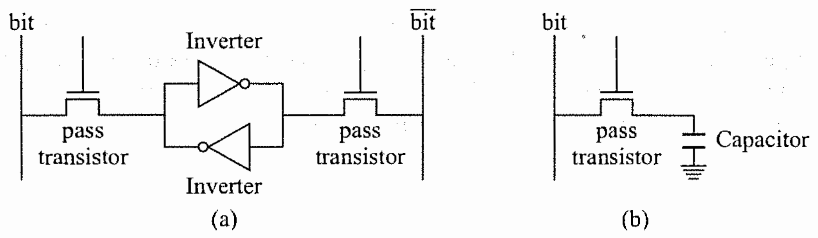


Fig. 13.27 (a) A basic SRAM cell, and (b) A basic DRAM cell

to high latency. This is why even if DRAM is used as a computer's main memory to make it cheap, SRAM is used as a *cache memory* for faster access instruction and data. Cache memory uses *locality* feature where a set of sequential instructions and data are found in contiguous locations in memory. A cache controller brings this memory block from main memory to cache (which is SRAM) for speedier operation of the computer.

13.7 SEQUENTIAL PROGRAMMABLE LOGIC DEVICES

We have discussed programmable devices (like PLA, PAL) for combinatorial circuits. In this section, we discuss similar devices available for sequential logic circuits. Architecture-wise, they additionally have memory elements like flip-flops. The simplest of the three widely used variety is called Simple Programmable Logic Devices (SPLD or simply PLD). Similar technology but using larger number of logic gates, suitable to address more complex sequential logic problems is called Complex Programmable Logic Devices (CPLD). The third type uses slightly different technology but of much higher capacity is known as Field Programmable Gate Array (FPGA). The term; High Capacity Programmable Logic Devices (HCPLD) is also used to refer to CPLD and FPGA together. Note that, Hardware Descriptions Languages (HDL), like Verilog, can be used to program these devices.

PLD

Refer to discussions of Section 4.10 and 4.11 on PAL and PLA and corresponding figures (Fig. 4.44, Fig. 4.47). Note that, output is taken from OR gates following AND plane generating combinatorial logic functions in SOP form. Each OR gate with added circuitry (as shown in Fig. 13.28) that includes a flip-flop, multiplexer and tri-state output forms a *macrocell* of PLD. The flip-flop can store the OR gate output indefinitely and is triggered by a Clock. Multiplexer selects either OR gate or flip-flop output which is also fed back to AND plane for internal use. The output buffer when enabled by *Enable* makes multiplexer output available to external world through output pin, else output pin is held at high impedance state.

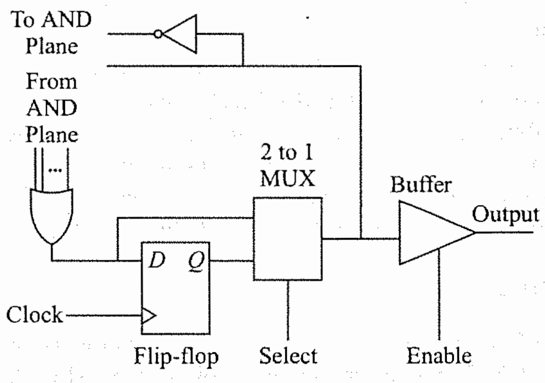


Fig. 13.28 A macrocell of PLD

Each PLD typically has 8–10 macrocells. Advanced Micro Devices (AMD) manufactured SPLDs 16R8 and 22V10 are PAL based. The name "16R8" means that the PAL has a maximum of 16 inputs (there are 8 dedicated inputs and 8 input/outputs which can be configured either as input or output), and a maximum of 8 outputs. The "R" stands for PAL outputs registered as *D* flip-flop. Similarly, the "22V10" has a maximum of 22 inputs and 10 outputs and "V" stands for versatility of the output. The other manufacturers of popular SPLDs are Altera, Lattice, Cypress and Philips-Signetics.

CPLD

Simple PLDs can handle 10–20 logic equations. Thus, for more complex circuit design one needs to physically connect few such units. This problem is solved by the advent of CPLD, which consists of a number of PLD like blocks (Fig. 13.29). The blocks are interconnected among themselves through programmable switches present in interconnection block. This means it needs two levels of programming: one for programming PLD block the other for programming the switches. Input and output pins of a CPLD chip are routed through I/O blocks. Here, the macrocell has a two input Ex-OR gate after the OR gate. Depending on the value present in the other (Control) input, Ex-OR gate sends complemented or uncomplemented OR output to flip-flop and multiplexer. Commercial CPLDs can have up to 50 PLD blocks. Higher density is not supported by CPLD architecture and FPGAs are used for that. Transistors are used as programmable switches for CPLDs (and also for many SPLDs) by placing it between two wires in a way that facilitates implementation of wired-AND functions. EPROM used as switches does not support in-circuit programming but EEPROM does that. The advantage with them is that both are non-volatile in nature. Re-programmability feature of CPLD is a very useful advantage.

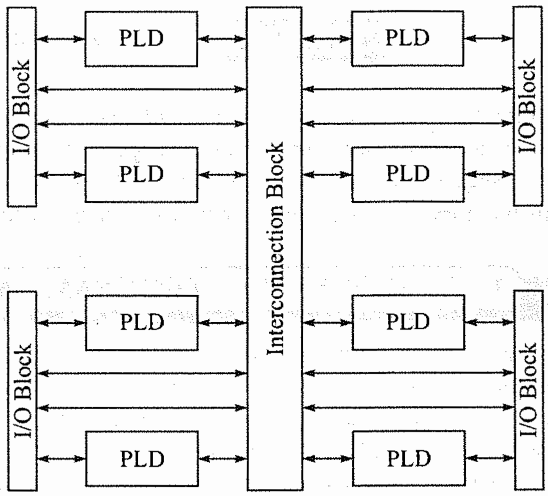


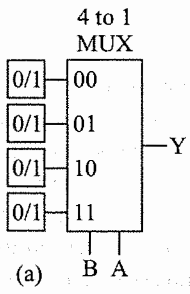
Fig. 13.29 **A typical block diagram representation of CPLD**

The applications of CPLDs can be found in reasonably complex designs, like graphics controller, UARTs, cache control and many others. Circuits that can exploit wide AND/OR gates, and do not need too many flip-flops are suited for CPLD implementation.

AMD offered CPLD family, Mach 1 to Mach 5 comprises multiple PAL-like blocks: Mach 1 and 2 consist of optimized 22V16 PALs, Mach 3 and 4 comprise several optimized 34V16 PALs and Mach 5 is similar but offers enhanced speed performance. Mach chips are based on EEPROM technology, Xilinx offers XC7000 and XC9500 where each chip consists of a collection of SPLD-like blocks with 9 macrocells in each. Altera has developed three families of chips that fit within the CPLD category: MAX 5000, MAX 7000, and MAX 9000. The other manufacturers of CPLD are Lattice, Cypress, etc.

FPGA

FPGA consists of an array of circuit elements called logic blocks, which unlike AND-OR combination of CPLD has programmable look up table (LUT). The look up table can generate any logic combination for the variables involved. A multiplexer based 2-variable look up table is shown in Fig. 13.30a. Based on what value (0 or 1) is stored at input this can generate any of the $2^4 = 16$ possible functions of A, B as



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

(b)

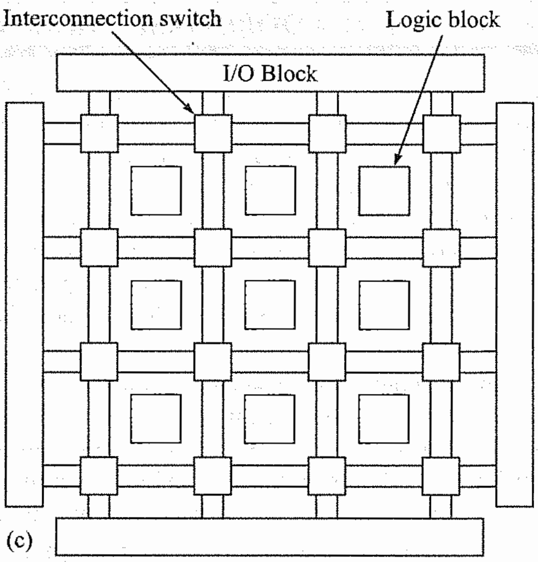


Fig. 13.30 **(a) A two variable programmable look up unit, (b) Generation of any two variable logic Y = *f* (A, B) by placing appropriate combination as input to 4 to 1 Multiplexer, (c) A typical structure of FPGA**

Y = *f* (A, B). Few examples are shown in the table of Fig. 13.30b. A typical FPGA structure is shown in Fig. 13.30c which is a two dimensional array of logic blocks interconnected by horizontal and vertical wires. The interconnection switches in interconnection blocks are either SRAM or antifuse type. Antifuses are modified CMOS based, normally open circuit but provides low resistance when programmed. Antifuses are not reprogrammable and nonvolatile. SRAM switches are reprogrammable but volatile.

FPGAs have gained rapid acceptance and growth because they can be applied to a very wide range of applications like device controllers, communication encoding and filtering, small to medium sized systems with SRAM blocks and many more. The other important applications of FPGAs are prototyping of designs (later to be implemented in custom made integrated circuits) and also for emulation of large hardware systems.

In Xilinx XC4000 SRAM based FPGA, each configurable logic block (also called CLB) can generate logic functions of up to nine inputs and has two flip-flops. Each of the interconnecting horizontal or vertical channel contains some short wire segments that span a single CLB, longer segments that span two CLBs, and very long segments that span the entire length or width of the chip. In this series, XC4003E has 100 CLBs while XC40250XV has as high as 8464 CLBs. Xilinx also offers XC2000, XC3000, XC5000 and XC8100(antifuse). The other manufacturers providing commercial FPGAs are Altera: FLEX8000 and FLEX10000, Actel: Act1, Act2 and Act3, Quicklogic: pASIC, pASIC2, etc. The SRAM based FPGAs, normally comes with EPROMs that stores bit-streams. This gets loaded every time power is switched on and programs the FPGA.

13.8 CONTENT ADDRESSABLE MEMORY

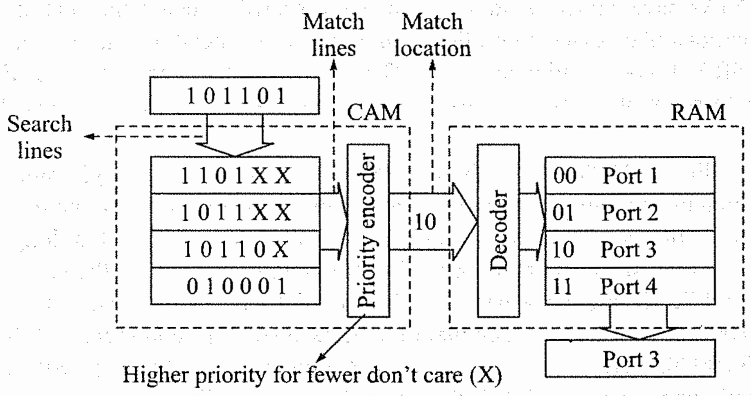
Content addressable memory (CAM) uses a completely different kind of addressing scheme from what has been discussed so far. It is designed to be faster to serve specific applications where speed is an issue. Take for example functioning of a network like the Internet. There, a message such as an e-mail or a web page is transferred by first breaking up the message into small data packets of a few hundred bytes, and then, sending each data packet individually through the network. These packets are routed from the source, through the intermediate nodes of the network (called routers), and reassembled at the destination to reproduce the original message. The function of a router is to compare the destination address of a packet to all possible routes and choose the appropriate one. A CAM is a good choice for implementing this lookup operation due to its fast search capability.

CAM compares input search data against a table of stored data, and returns the address of the matching data. Thus, it makes use of the content or data itself to find specific address by implementing a lookup table function using dedicated comparison circuitry. This is to reduce address decoding delays of conventional RAM by making the address meaningful and not an arbitrary one like RAM. A basic CAM cell serves two basic functions—bit storage, like RAM and in addition, bit comparison.

Let us try to understand how a CAM works from a packet forwarding example of a router. Figure 13.31a shows a simple routing table where output port assignment is shown for a range of destination addresses available from the relevant portion of input data stream. This table is a part of the CAM as shown in Fig. 13.31b. When an input data arrives with destination address 101101, the matching of the content occurs for 2nd and 3rd row, i.e. output port 2 and 3 both are eligible to transmit this data. A priority encoder following this lookup table decides which one is to be selected when a match occurs at more than one place. It follows a specific priority scheme. In this example, higher priority is given for the match that has lower number of don't care (X) states. The logic behind this is to keep a port free or available—to the extent possible—that can handle more number of destination addresses. Thus, match location 10 is the output of the priority encoder. The decoder to RAM takes this 10 as the address and selects port 3 as the selected output port.

Destination address	Output port
1101XX	1
1011XX	2
10110X	3
010001	4

(a)



(b)

Fig. 13.31 (a) Routing table, (b) CAM implementing address lookup

Extending the above example, generally speaking, an input word or *tag* from the incoming data is first stored in a *Search Data Register*. Its content, i.e. the tag, is then broadcasted as *search word* over *search lines*. In a typical CAM, there could be 36 to 144 bits in search lines while the table size could be as high as 32K entries (up to 15 bits of address space). Each stored word has a match line and whenever a match with a word occurs, the corresponding match line is activated. A priority encoder selects a match location based on some priority rule when there are more than one match line in activated state. This effectively reduces a larger space of input search word to a smaller space of output match location. The matching logic corresponding to CAM tags could be *binary* or *ternary*. The binary CAM requires exact match of all the binary locations and returns corresponding match lines while ternary CAM in addition, allows matching with don't care (X) bits. As we have seen in the network routing example, ternary CAM is more useful but also more complex to manage. Often, a CAM comes with a *hit* flag to indicate if there is no matching location in CAM.

SELF-TEST

15. What is the organization of the 7489, 64-bit RAM?
16. What is the organization of the 2114 SRAM?
17. What is a DRAM?
18. What is the organization of a 4116 DRAM?
19. How is combinatorial logic generated in FPGA?

SUMMARY

This entire chapter has been devoted to the study of memories. The use of magnetic and optical memory is discussed first. Next we considered the various rectangular arrays of memory cells on a chip and found that a square array containing the same number of rows and columns requires the fewest number of address lines.

Programmable, erasable-programmable, and plain read-only memories (PROMs, EPROMs, and ROMs) are used to store data in appiications where the data changes not at all, or only infrequently. These memory chips are available as either bipolar or MOS, but the MOS devices offer much greater capacity per chip.

Random-access memories (RAMs) are also available as either bipolar or MOS devices and are used to store data that must be readily available and may be changed frequently. The dynamic random-access memory (DRAM) offers the greater advantage of more storage capacity on a chip but has the disadvantage of requiring refreshing. Careful attention to timing requirements is absolutely essential with the use of any memory chip.

The basic memory cell on a bipolar chip is a simple latch using cross-coupled bipolar junction transistors. The same is true for a static MOS or CMOS memory chip, except that the transistors used are MOS or CMOS, respectively. A dynamic memory, on the other hand, uses a capacitor and one or more MOS transistors to store charge and, therefore, a single bit.

We have not undertaken an exhaustive study of all the memory chips available, but the chips discussed in detail are representative of the most popular ones in present use.

GLOSSARY

- ***access time*** In general, the delay time measured fron chip-enable (or address) until valid data appears at the output.
- ***address*** Selection of a cell in a memory array for a read or a write operation.
- ***cache*** Small, fast SRAM, a faster memory as an adjunct to slower main memory.
- ***CAM*** Content Addressable Memory.
- ***capacity*** The total number of bits that can be stored in a memory.
- ***chip*** A semiconductor circuit on a single silicon die.
- ***CD-ROM*** Compact Disk Read Only Memory, a kind of movable optical storage media that has higher capacity compared to magnetic counterpart.
- ***CD-R*** Compact Disk Recordable, a kind of optical memory on which data can be written but once.
- ***CD-RW*** Compact Disk Rewritable, a kind of optical memory on which data can be written and erased many times.
- ***DRAM*** Dynamic RAM.
- ***DVD*** Digital Versatile Disk or Digital Video Disk, a very high density optical memory.
- ***dynamic memory*** A memory whose contents must be restored periodically.
- ***EPROM*** An erasable-programmable read-only memory.
- ***EEPROM*** Electrically Erasable Programmable Read Only Memory.
- ***field-programmable*** Referring to a PROM that can be programmed by the user.
- ***flash memory*** A kind of nonvolatile memory which can be written and erased electrically.
- ***Floppy disk*** A movable low capacity magnetic storage media.
- ***Hard disk*** A high capacity magnetic storage media, integral part of modern computer.
- ***mask-programmable*** Referring to a PROM that can be programmed only by the manufacturer.
- ***matrix addressing*** Selection of a single cell in a rectangular array of cells by choosing the intersection of a single row and a single column.
- ***memory bank*** Connects more than one memory block to increase the capacity of the memory.
- ***memory cell*** The circuit used to store a single bit of information in a semiconductor memory chip.
- ***nonvolatile storage*** A method whereby a loss of power will not result in a loss of stored data.
- ***packing density*** Number of memory bits packed in per unit space.
- ***pass transistor*** A MOS transistor that passes information in either direction when it is turned on.
- ***PROM*** Programmable read-only memory.
- ***RAM*** Random-access memory.
- ***read operation*** The act of detecting the contents of a memory.
- ***refresh cycle*** Periodic refresh of DRAM.
- ***ROM*** read-only memory.
- ***static memory*** A memory capable of storing data indefinitely, provided there is no loss of power.
- ***SRAM*** Static RAM.
- ***volatile storage*** A method of storing information whereby a loss of power will result in a loss of the data stored.
- ***write operation*** The act of storing information in a memory.

PROBLEMS

Section 13.1

13.1 State the most appropriate memory type to use for each of the following:

a. The working memory in a small computer
b. The memory used to store permanent programs in a small computer
c. A memory used to store development programs in a small computer

13.2 Explain the difference between an EPROM and a PROM.

13.3 Explain the term volatile memory.

13.4 Explain why an EPROM is or is not a volatile memory.

13.5 A memory chip has a read and a write input. Is the chip ROM or RAM?

13.6 What is the difference between a memory cell and a memory word?

13.7 Why is a ROM considered nonvolatile memory?

13.8 What is the difference between an SRAM and a DRAM?

Section 13.2

13.9 What is the advantage of using a read-write head in magnetic recording systems?

13.10 Look at the code in Fig. 13.6b and determine the proper recording for each of the following:

a. The decimal number 4
b. The letter D
c. The decimal number 8
d. The decimal number 7

13.11 Why is recording on magnetic tape not considered random access?

13.12 A 2400-ft reel of 1/2-in magnetic tape has a data storage density of 6250, 7-bit characters per inch. Assuming no gaps and no lost space, what is the maximum storage capacity of the tape?

13.13 A 2400-ft reel of 1/2-in magnetic tape has a rewind speed of 300 in/s. How much time is required to rewind the tape from mid-position to its beginning? Neglect start and stop times.

Section 13.3

13.14 Why data integrity of optical memory is better than magnetic memory?

13.15 What is the data transfer rate of a 52X CD-ROM drive?

13.16 Briefly explain Read, Write, Erase process of CD-RW media.

13.17 What is the data transfer rate of 8X DVD-ROM drive?

13.18 Show the different possible rectangular arrangements for a memory that contains 32 memory cells. How many rows and columns for each case?

13.19 How many address lines are required for each case in Prob. 13.14?

13.20 What is the required address $A_4A_3A_2A_1$ to select cell 21 in Fig. 13.10c?

13.21 Determine how many address bits are required for a memory that has the following number of bits:

a. 1024	b. 4098
c. 256	d. 16,384

13.22 A memory chip available from Advanced Micro Devices is the Am9016, advertised as a 16K memory. How many bits of storage are there? How many address lines are required to access one bit at a time?

13.23 What address must be applied to the 74S89 in Fig. 13.20 to select the 4-bit word stored in row 14? Give the address in both binary and hexadecimal.

Section 13.4

13.24 What is the required address in both binary and hexadecimal to select the 8-bit word in row 27 of the TBP18S030 in Fig. 13.16?

13.25 Show a method for scanning the contents of a TBP18S030 beginning with word 1, then word 2, and so on up to word 32, and then repeating. (***Hint***: Try using a five-flip-flop binary counter for the address *ABCDE*, or use a mod-5 counter with decoding gates, or maybe a shift counter, or ...)

13.26 Write a Boolean expression for address row 15 in the TBP18S030 in Fig. 13.16.

13.27 Define the term mask-programmable.

13.28 Draw a set of timing waveforms for a TBP18S030 similar to Fig. 13.17, assuming an access time of 35 ns.

13.29 Redraw Fig. 13.30 and show exactly how to set the switches to program the 8-bit word at address 110 101. Explain exactly what must be done to program the word 1010 0011 at this address. Connecting switch *P* to an output will program a 1 at that cell.

13.30 In a manufacturing process, the pressure (P) in a pipe is related to the fluid in the pipe (F) according to the relation $P = 3F + 2$. Rather than compute values in real time, it is decided to store precomputed data in a PROM. In this case, F has only integer values between 0 and 4, so the computed values are found as shown in the accompanying table. Here's how the data is stored: Each integer value of F (0, 1, 2, 3, and 4) represents an address in the PROM. The value of P is stored in binary form at the proper address. For instance, when $F = 2$, $P = 1000$ is stored at row address 2.

a. What is the value of P when $F = 4$?

b. Draw a PROM having 4-bit words, and show how all data are stored.

F	*P*	*P(binary)*
0	2	0010
1	5	0101
2	8	1000
3	11	1011
4	14	1110

13.31 Repeat part (b) of Prob. 13.30 if the relation is changed to $P = 2F + 1$.

13.32 Design a ROM to be used as a look-up table for the relation $L = S^2 - 2S + 3$, where $0 \le S \le 6$, and S has only integer values.

Section 13.5

13.33 Show how to connect 7489s in series to construct a memory that has thirty-two 4-bit words.

13.34 Show how to connect 7489s in parallel to construct a memory that contains sixteen 8-bit words.

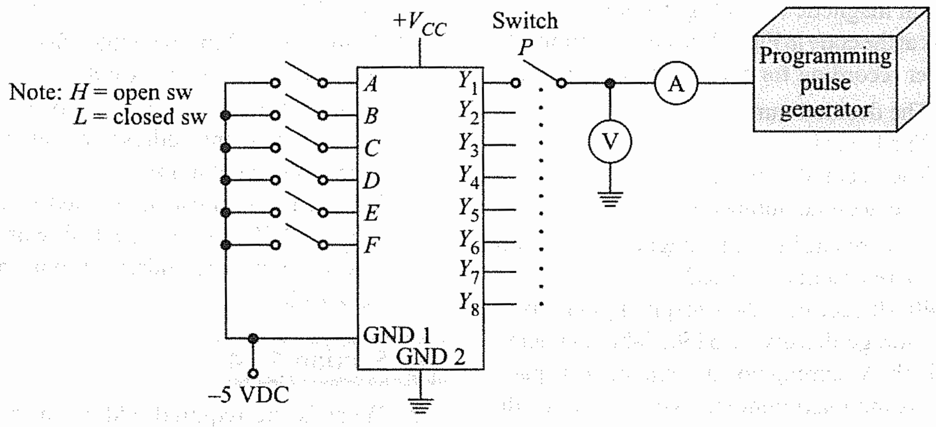


Fig. 13.32

13.35 Design the logic circuits, to provide a read and a write cycle for a 7489.

13.36 Refer to the 74S201 information in Fig. 13.26 and determine the following:

a. Minimum write-enable pulse width

b. Setup time, address to write-enable

c. Hold time, data from write-enable

13.37 Draw the logic diagram for a 256-word 8-bit memory using '201s.

Answers to Self-tests

1. A DRAM must be refreshed periodically.
2. EPROM stands for erasable-programmable read-only memory.
3. Cache memory is a small high-speed SRAM used inside a computer to speed up operation.
4. Even
5. Tape access time is too long!
6. Binary information is recorded on as magnetic film by magnetizing spots with two different orientations.
7. 780 nm.
8. CD-ROM 25% and more than 70%. CD-RW 15% and 25%.
9. 8.5 GB.
10. 145 (decimal) = 1001 0001 = $A_7A_6A_5A_4A_3A_2A_1A_0$
11. The cell at address 22—row 2 and column 2.
12. It refers to a ROM whose contents are established during the manufacturing process.
13. The triangle is the symbol for a three-state output.
14. It can be corrected by simply programming (adding) a 1 at word position Q_3. Note that you can add a 1 by programming (this is destroying a fuse link), but you cannot remove a programmed 1, since this would require replacing a fuse link.
15. Sixteen 4-bit words.
16. 1024, 4-bit words.
17. DRAM stands for dynamic random-access memory.
18. 16,384 × 1 bits
19. Through multiplexer-based look up table.

Digital Integrated Circuits

14

OBJECTIVES

- Explain how diodes and transistors can be used as electronic switches
- Demonstrate an understanding of TTL devices, their parameters, how to drive them, and how to use them to drive external loads
- Be familiar with CMOS-devices and characteristics
- Understand TTL-to-CMOS and CMOS-to-TTL interfacing

In 1964 Texas Instruments introduced transistor-transistor logic (TTL), a widely used family of digital devices. TTL is fast, inexpensive, and easy to use. In this chapter we discuss several types of TTL: standard, high-speed, low-power, Schottky, and low-power Schottky. You will learn about open-collector and tri-state devices because these are used to build buses, the backbone of modern computers and digital systems. Since TTL uses active-low as well as active-high signals, negative logic may be used as well as positive logic. Complementary metal-oxide semiconductor (CMOS) devices are chips that combine *p*-channel and *n*-channel MOSFETs in a push-pull arrangement. Because the input current of a MOSFET is much smaller than that of a bipolar transistor, cascaded CMOS devices have very low power dissipation compared with TTL devices. This low dissipation explains why CMOS circuits are used in battery-powered equipment such as pocket calculators, digital wristwatches, and portable computers.

Since a knowledge of the subjects covered here is not prerequisite to any other chapter in this text, the material can be studied in part or in whole, at any time. An understanding of Ohm's law and familiarity with basic dc circuits are the only background needed.

14.1 SWITCHING CIRCUITS

The semiconductor devices used in digital integrated circuits (ICs) include diodes, *bipolar junction transistors (BJTs)* and *metal-oxide-semiconductor field-effect transistors (MOSFETs)*. The most popular transistor-transistor logic (TTL) in use includes the 7400 and the 74LS00 families; resistors, diodes, and BJTs are the elements used to construct these circuits. The 74C00 and the 74HC00 are the most widely used families constructed using MOSFETs. These two families of circuits are referred to as CMOS, since they use two different types of MOSFETs. In Chapter 1, we used the term *electronic switch* (see Fig. 1.7). Virtually all digital ICs in use today are silicon, so let's see how a silicon diode or transistor is used as an electronic switch.

The Semiconductor Diode

The symbol for a semiconductor diode (sometimes called a *pn* junction) is shown in Fig. 14.la. The diode behaves like a *one-way* switch. That is, it will allow an electric current in one direction but not the other. We will use *conventional current flow* rather than *electron flow*. Figure 14.1b shows the direction of current through a diode—this is the *forward* direction. When conducting current, a silicon (Si) diode will have a nominal voltage of 0.7 V across its terminals as shown in Fig. 14.1b. In this condition, the diode is said to be *forward-biased*. Notice that the triangle in the diode symbol points in the direction of forward current—an easy memory crutch! It is not possible to pass current through the diode in the other direction—the *reverse* direction. When *reverse-biased*, the diode will act as an open switch as illustrated in Fig. 14.lc. To summarize:

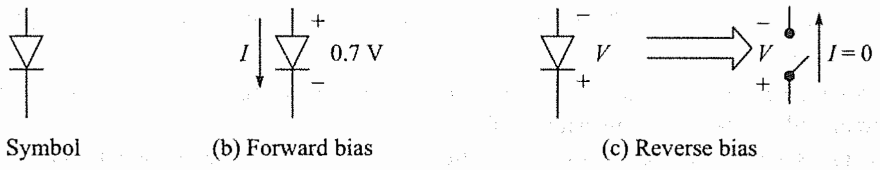


Fig. 14.1 **Semiconductor diode**

1. When forward-biased, the diode conducts current, and the voltage across the diode terminals is about 0.7 Vdc.
2. When reverse-biased, the diode will not conduct current. The voltage across the diode terminals depends on the external circuit.

Example 14.1 For each diode in Fig. 14.2, determine whether the diode is forward- or reverse-biased. Determine the diode current I in each case.

Solution

(a) The current direction is from +5 Vdc to ground, and thus the diode is forward-biased. The voltage across the diode terminals is 0.7 Vdc, and the diode current is found as

$$I = (5 - 0.7)/1\ \text{k}\Omega = 4.3/1\ \text{k}\Omega = 4.3\ \text{mA}$$

(b) The current direction is from +12 Vdc to ground, thus the diode is reverse-biased. The diode current is then I = 0.0 mA. There is no voltage across the 10-kΩ resistor, and thus the voltage across the diode terminals is 12 Vdc.

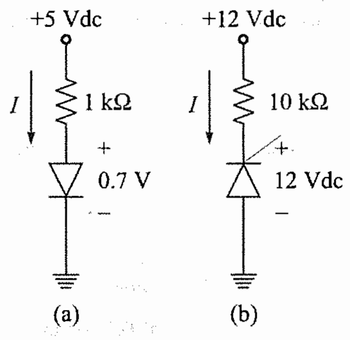
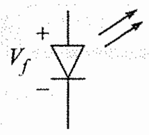


Fig. 14.2

LEDs

The symbol for a light-emitting diode (LED) is shown in Fig. 14.3a. The arrows indicate light emission capability. The operation of an LED is similar to that of an ordinary diode. When forward-biased, it emits light in the visible spectrum and is thus used as an indicator. However, the voltage across the diode terminals when forward-biased (V_f) is somewhat greater than 0.7 Vdc. Typical LED forward voltages given in Fig. 14.3b show that V_f varies with the color of the emitted light. The color of the emitted light depends on the elements added to the semiconductor material during manufacturing.



(a) Symbol

	Red	Yellow	Green
V_f(V)	1.6	2.2	2.4

(b) Typical forward voltages

Fig. 14.3

Example 14.2 The diode in Fig. 14.2a is replaced with a *red* LED. What is the diode current?

Solution The diode is forward-biased, and the voltage across its terminals is about 1.6 Vdc (Fig. 14.3b). The diode current is then

$$I = (5 - 1.6)/1\text{ k}\Omega = 3.4/1\text{ k}\Omega = 3.4\text{ mA}$$

BJTs

The bipolar junction transistor (BJT) is available in two polarities (*npn* and *pnp*), as shown by the symbols in Fig. 14.4a. The BJT terminals are named *collector*, *emitter*, and *base*, as indicated. In Fig. 14.4b the BJT behaves as an electronic switch. The switch is activated by applying a voltage between base and emitter. Here's how it works:

1. The voltage between base and emitter is zero. The switch is open, and no current is allowed between collector and emitter. The transistor is off.
2. A voltage is applied between base and emitter. The switch is closed and a current is allowed between collector and emitter. The transistor is on. The voltage between emitter and collector (across a closed switch) is zero!

Since the BJT is available in two polarities—*npn* and *pnp*—the polarity of the applied base-emitter voltage must be as shown in Fig. 14.4c. For the *npn*, the base must be more positive than the emitter. The opposite is true for the *pnp*. This base-emitter voltage is applied across a forward-biased *pn* junction (a diode) and is thus limited to about 0.7 Vdc. Care must be taken not to exceed 0.7 Vdc, or the BJT may be destroyed.

The current through the *npn* transistor must be from collector to emitter as shown in Fig. 14.4c. For the *pnp*, current must be from emitter to collector. Notice that the current is in the direction of the arrow on the

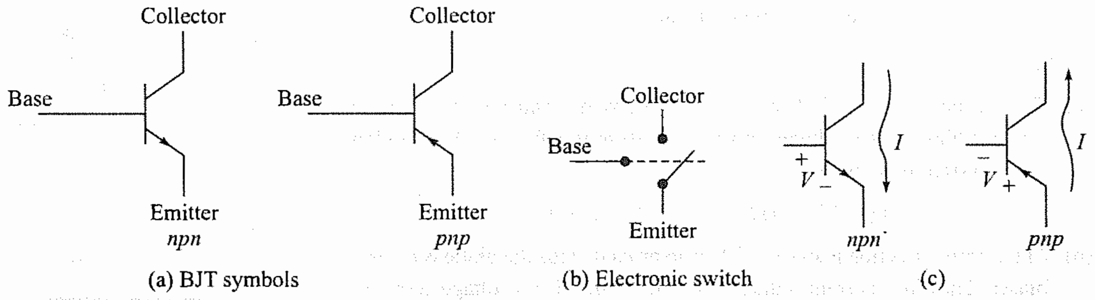


Fig. 14.4

emitter—a good memory crutch! These polarities and current directions are important—*you should make every effort to commit them to memory!*

Example 14.3

a. Determine the current I and the voltage V_2 for the circuit in Fig. 14.5a if (i) $V_1 = 0$ Vdc, and (ii) $V_1 = +5$ Vdc.
b. Repeat part (a) for the circuit in Fig. 14.5b.

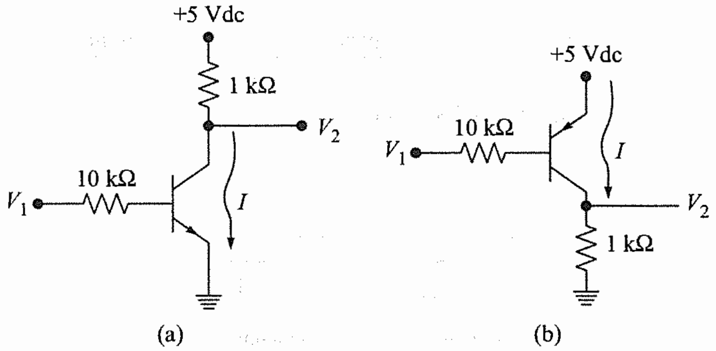


Fig. 14.5

Solution

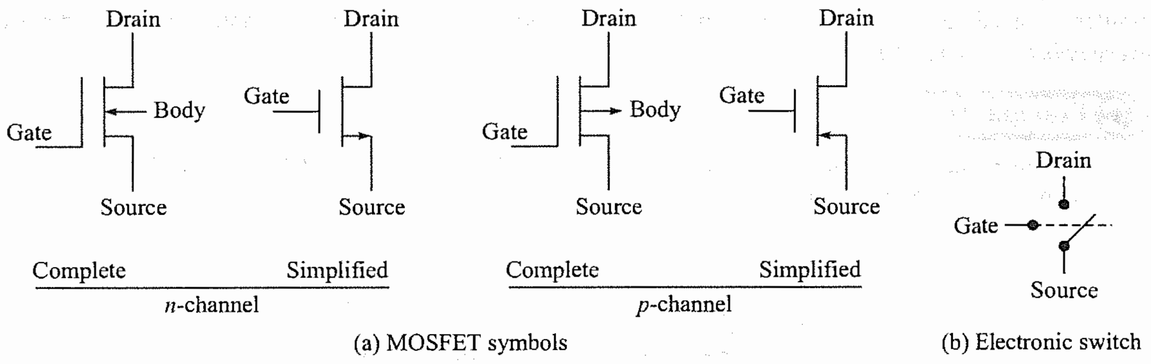
a. $V_1 = 0$ Vdc. There is no current in the 10-kΩ resistor. Thus the voltage base-emitter is zero. The BJT is off (switch is open). The BJT current and the current in the 1-kΩ resistor is zero. The voltage V_2 is +5 Vdc.
   $V_1 = +5$ Vdc. The base is more positive than the emitter—the BJT is on (switch is closed). V_2 is zero. The BJT current is $I = 5$ mA.
b. V_1 is 0 Vdc. The base is more negative than the emitter—the BJT is on (switch closed). V_2 is +5 Vdc. The BJT current is $I = 5$ mA.
   V_1 is +5 Vdc. There is no current in the 10-kΩ resistor. The base is at +5 Vdc, and so is the emitter. Thus the voltage base-emitter is zero, and the BJT is off (switch open). The current $I = 0$ mA, and $V_2 = 0$ Vdc.

Let's look carefully at the results from Example 14.3. For both circuits, when $V_1 = 0$ Vdc, $V_2 = +5$ Vdc. Also, when $V_1 = +5$ Vdc, $V_2 = 0$ Vdc. Clearly V_2 is always the *inverse* of V_1—in other words, each circuit in Fig. 14.5 is an inverter! Either of these circuits can be used to implement the basic inverter introduced in Chapter 1 (Fig. 1.10).

MOSFETs

MOSFETs are available in two polarities (n-channel and p-channel) as shown by the symbols in Fig. 14.6a. MOSFETs operate as "depletion" or "enhancement" mode devices; the transistors in Fig. 14.6 are enhancement types. The MOSFET terminals are named *gate, source, drain, and body* as indicated. When the body is connected to the source, as is often the case with ICs, the simplified symbols are used. The MOSFET also behaves as the electronic switch in Fig. 14.6b. The switch is activated by applying a voltage between gate and source. Here's how it works:

1. The voltage between gate and source is zero. The switch is open, and no current is allowed between source and drain. The transistor is off.



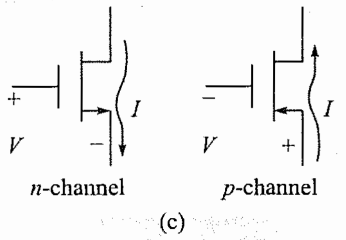


Fig. 14.6

2. A voltage is applied between gate and source. The switch is closed, and a current is allowed between source and drain. The transistor is on. The voltage between source and drain (across a closed switch) is zero!

Since the MOSFET is available in two polarities—*n*-channel and *p*-channel—the polarity of the applied gate-source voltage must be as shown in Fig. 14.6c. For the *n*-channel transistor, the gate must be more positive than the source. The opposite is true for the *p*-channel transistor. The current through the *n*-channel transistor must be from drain to source as shown in Fig. 14.6c. For the *p*-channel transistor, current must be from source to drain. Notice that the current is in the direction of the small arrow on the drain—a good memory crutch! These polarities and current directions are important—*you should make every effort to commit them to memory!*

Example 14.4 An *n*-channel MOSFET can be used to construct a simple inverter as shown in Fig. 14.7. Determine the current I and the output voltage V_2 if (a) $V_1 = 0$ Vdc, and (b) $V_1 = +5$ Vdc.

Solution

(a) $V_1 = 0$ Vdc. There is no current in the 100-kΩ resistor. Thus the gate-source voltage is zero. The MOSFET is off (the switch is open). The MOSFET current and the current in the 10-kΩ resistor are zero. The voltage V_2 is +5 Vdc.

(b) $V_1 = +5$ Vdc. The gate is more positive than the source—the MOSFET is on (the switch is closed). V_2 is zero. The MOSFET current is $I = 0.5$ mA. This circuit could also be used to implement the basic inverter introduced in Chapter 1 (Fig. 1.10).

The small size of a MOSFET on an IC is one of the great advantages of ICs constructed using MOSFETs. The 10-kΩ resistor in Fig. 14.7 requires a large area on an IC compared to a MOSFET. A second MOSFET can be used in place of this resistor, as shown in Fig. 14.8. In this case, transistor Q_1 has its gate connected directly to its drain. When it is connected in this fashion, its behavior is similar to a resistor but shows little non-linearity compared to passive load, Q_1 is called an *active load*, and this circuit is a simple inverter.

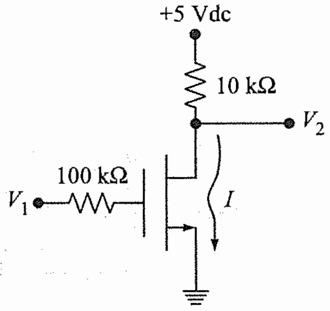


Fig. 14.7

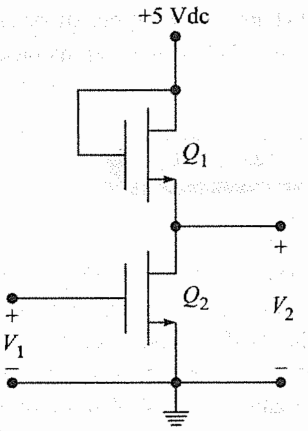


Fig. 14.8 An inverter with active load

Complementary Metal-Oxide-Semiconductor (CMOS) FETs

ICs constructed entirely with *n*-channel MOSFETs are called *NMOS* ICs. ICs constructed entirely with *p*-channel MOSFETs are called *PMOS* ICs. The 74C00 and 74HC00 families are constructed using *both* *n*-channel and *p*-channel MOSFETs. Since *n*-channel and *p*-channel MOSFETs are considered *complementary* devices, these ICs are referred to as *CMOS* ICs.

A CMOS inverter is shown in Fig. 14.9. Ideally, the characteristics of the *n* channel are closely matched with the *p* channel. This circuit is the basis for the 74C00 and 74HC00 families. Here's how it works:

1. V_1 = 0 Vdc. Q_n is off and Q_p is on. V_2 = +5 Vdc.
2. V_1 = +5 Vdc. Q_n is on and Q_p is off. V_2 = 0 Vdc.

Note that in the steady state (while not switching), one of the transistors is *always off.* As a result, the current I = 0 mA. When switching between states, both transistors are on for a very short time because of the rise or fall time of V_1. This is the only time the current I is nonzero. This is the reason CMOS is used in applications where dc power supply current must be held to a minimum—watches, pocket calculators, etc. *A word of warning:* If the input V_1 is held at +5 Vdc/2 = 2.5 Vdc, *both* transistors will be on. This is an almost direct short between +5 Vdc and ground, and it won't be long before both transistors expire! So don't impose this condition on a CMOS IC.

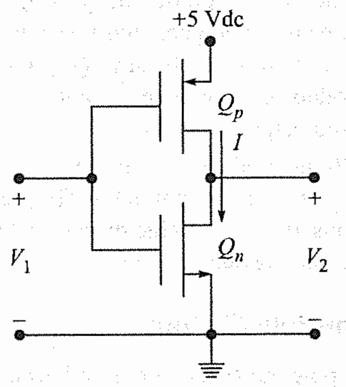


Fig. 14.9 A CMOS inverter

SELF-TEST

1. How does an LED differ from an ordinary silicon diode?
2. What are the two types of BJTs? What is the complement of an *n*-channel MOSFET?
3. An *npn* BJT is on when its base is more (positive, negative) than its emitter.
4. What is an active load in an NMOS IC?

14.2 7400 TTL

Standard TTL

Figure 14.10 shows a TTL NAND gate. The *multiple-emitter* input transistor is typical of the gates and other devices in the 7400 series. Each emitter acts like a diode; therefore, Q_1 and the 4-kΩ resistor act like a 2-input AND gate. The rest of the circuit inverts the signal so that the overall circuit acts like a 2-input NAND gate. The output transistors (Q_3 and Q_4) form a *totem-pole connection* (one *npn* in series with another); this kind of output stage is typical of most TTL devices. With a totem-pole output stage, either the upper or lower transistor is on. When Q_3 is on, the output is high; when Q_4 is on, the output is low.

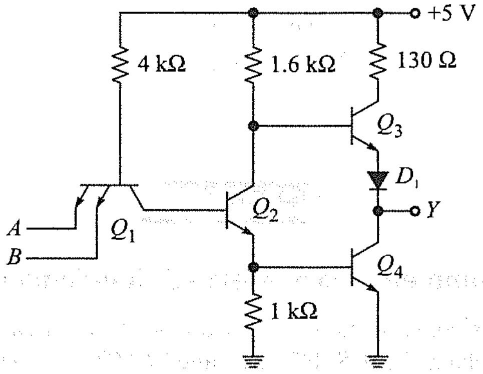


Fig. 14.10 Two-input TTL NAND gate

The input voltages A and B are either low (ideally grounded) or high (ideally +5 V). If A or B is low, the base of Q_1 is pulled down to approximately 0.7 V. This reduces the base voltage of Q_2 to almost zero. Therefore, Q_2 cuts off. With Q_2 open, Q_4 is off, and the Q_3 base is pulled high. The emitter of Q_3 is only 0.7 V below the base, and thus the Y output is pulled up to a high voltage.

On the other hand, when A and B are both high voltages, the emitter diodes of Q_1 stop conducting, and the collector diode goes into forward conduction. This forces Q_2 to turn on. In turn, Q_4 goes on and Q_3 turns off, producing a low output. Table 14.1 summarizes all input and output conditions.

Table 14.1 Two-Input NAND Gate

A	*B*	*C*
0	0	1
0	1	1
1	0	1
1	1	0

Without diode D_1 in the circuit, Q_3 will conduct slightly when the output is low. To prevent this, the diode is inserted; its voltage drop keeps the base-emitter diode of Q_3 reverse-biased. In this way, only Q_4 conducts when the output is low.

Totem-Pole Output

Totem-pole transistors are used because they produce a *low output impedance*. Either Q_3 acts as an emitter follower (high output), or Q_4 is on (low output). When Q_3 is conducting, the output impedance is approximately 70 ohms (Ω); when Q_4 is on, the output impedance is only 12 Ω (this can be calculated from information on the data sheet). Either way, the output impedance is low. This means the output voltage can change quickly from one state to the other because any stray output capacitance is rapidly charged or discharged through the low output impedance.

Propagation Delay Time and Power Dissipation

Two quantities needed for later discussion are power dissipation and propagation delay time. A standard TTL gate has a power dissipation of about 10 milliwatts (mW). It may vary from this value because of signal levels, tolerances, etc. but on the average it is 10 mW per gate. The *propagation delay time* is the time it takes for the output of a gate to change after the inputs have changed. The propagation delay time of a TTL gate is approximately 10 nanoseconds (ns).

Device Numbers

By varying the design of Fig. 14.10 manufacturers can alter the number of inputs and the logic function. With only few exceptions, the multiple-emitter inputs and the totem-pole outputs are used for different TTL devices. Table 14.2 lists some of the 7400 series TTL gates. For instance, the 7400 is a chip with four 2-input NAND gates in one package. Similarly, the 7402 has four 2-input NOR gates, the 7404 has six inverters, and so on.

Table 14.2 **Standard TTL**

Device Number	*Description*
7400	Quad 2-input NAND gates
7402	Quad 2-input NOR gates
7404	Hex inverter
7408	Quad 2-input AND gates
7410	Triple 3-input NAND gates
7411	Triple 3-input AND gates
7420	Dual 4-input NAND gates
7421	Dual 4-input AND gates
7425	Dual 4-input NOR gates
7427	Triple 3-input NOR gates
7430	8-input NAND gate
7486	Quad 2-input XOR gates

5400 Series

Any device in the 7400 series works over a temperature range of 0 to 70°C and over a supply range of 4.75 to 5.25 V. This is adequate for commercial applications. The 5400 series, developed for the military applications, has the same logic functions as the 7400 series, except that it works over a temperature range of −55 to 125°C and over a supply range of 4.5 to 5.5 V. Although 5400 series devices can replace 7400 series devices, they are rarely used commercially because of their much higher cost.

High-Speed TTL

The circuit of Fig. 14.10 is called *standard TTL*. By decreasing the resistances a manufacturer can lower the internal time constants; this decreases the propagation delay time. The smaller resistances, however, increase the power dissipation. This design variation is known as *high-speed TTL*. Devices of this type are numbered 74H00, 74H01, 74H02, and so on. A high-speed TTL gate has a power dissipating around 22 mW and a propagation delay time of approximately 6 ns.

Low-Power TTL

By increasing the internal resistances a manufacturer can reduce the power dissipation of TTL gates. Devices of this type are called *low-power TTL* and are numbered 74L00, 74L01, 74L02, etc. These devices are slower than standard TTL because of the larger internal time constants. A low-power TTL gate has a power dissipation of 1 mW and a propagation delay time of about 35 ns.

Schottky TTL

With standard TTL, high-speed TTL, and low-power TTL, the transistors are switched on with excessive current, causing a surplus of carriers to be stored in the base. When you switch a transistor from on to off, you have to wait for the extra carriers to flow out of the base. The delay is known as *saturation delay time.*

One way to reduce saturation delay time is with *Schottky TTL*. The idea is to fabricate a Schottky diode along with each bipolar transistor of a TTL circuit, as shown in Fig. 14.11. Because the Schottky diode has a forward voltage of only 0.25 to 0.4 V, it prevents the transistor from saturating fully. This virtually eliminates saturation delay time, which means better switching speed. These devices are numbered 74S00, 74S01, 74S02, and so forth.

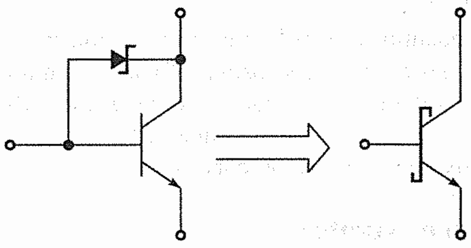


Fig. 14.11 **Schottky diode prevents transistor saturation**

Schottky TTL devices are very fast, capable of operating reliably at 100 megahertz (MHz). The 74S00 has a power dissipation around 20 mW per gate and a propagation delay time of approximately 3 ns.

Low-Power Schottky TTL

By increasing internal resistances as well as using Schottky diodes, manufacturers have come up with a compromise between low power and high speed: *low-power Schottky TTL*. Devices of this type are numbered 74LS00, 74LS01, 74LS02, etc. A low-power Schottky gate has a power dissipation of around 2 mW and a propagation delay time of approximately 10 ns, as shown in Table 14.3.

The Winner

Low-power Schottky TTL is the best compromise between power dissipation and saturation delay time. In other words, of the five TTL types listed in Table 14.3, low-power Schottky TTL has emerged as the favorite of digital designers. It is used for almost everything. When you must have more output current, you can fall back on standard TTL. Or, if your application requires faster switching speed, then Schottky TTL is useful. Low-power and high-speed TTL are rarely used. if at all.

SELF-TEST

5. Draw the symbol for a Schottky transistor.
6. Which TTL family offers the lowest power and the fastest operation?

14.3 TTL PARAMETERS

7400 series devices are guaranteed to work reliably over a temperature range of 0 to 70°C and over a supply range of 4.75 to 5.25 V. In the discussion that follows, *worst case* means that the *parameters* (input current, output voltage, and so on) are measured under the worst conditions of temperature and voltage. This means maximum temperature and minimum voltage for some parameters, minimum temperature and maximum voltage for others, or whatever combination produces the worst values.

Floating Inputs

When a TTL input is high (ideally +5 V), the emitter current is approximately zero (Fig. 14.12a). When a TTL input is *floating* (unconnected, as shown in Fig. 14.12b), no emitter current is possible because of the open circuit. Therefore, a floating TTL input is equivalent to a high output. Because of this, you sometimes see unused TTL inputs left unconnected; an open input allows the rest of the gate to function properly.

There is a disadvantage to floating inputs. When you leave an input open, it acts as a small antenna. Therefore, it will pick up stray electromagnetic noise voltages. In some environments, the noise pickup is large enough to cause erratic operation of logic circuits. For this reason, most designers prefer to connect unused TTL inputs to the supply voltage.

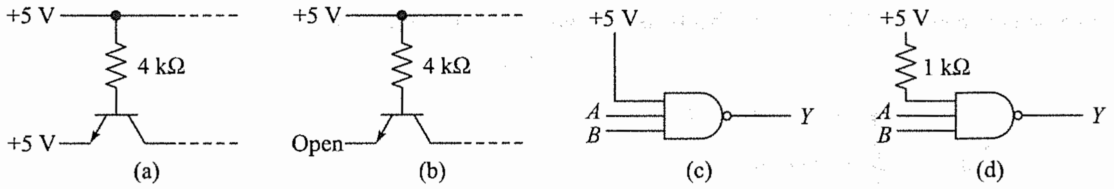


Fig. 14.12 **(a) High input, (b) Open is equivalent to high input, (c) Direct connection to supply voltage, (d) High input through a pull-up resistor**

For instance, Fig. 14.12c shows a 3-input NAND gate. The top input is unused, so it is connected to +5 V. A direct connection like this is all right with most Schottky devices (74S and 74LS) because their inputs can withstand supply overvoltages caused by switching transients. Since the top input is always high, it has no effect on the output. (*Note*: You don't ground the unused TTL input of Fig. 14.12c because then the output would remain stuck high, no matter what the values of A and B.)

Figure 14.12d shows an indirect connection to the supply through a resistor. This type of connection is used with standard, low-power, and high-speed TTL devices (74, 74L, and 74H). These older TTL devices have an absolute maximum input rating of +5.5 V. Beyond this level, the ICs may be damaged. The resistor is called a *pull-up resistor* because it serves to pull the input voltage up to a high. Most transients on the supply voltage are too short to charge the input capacitance through the pull-up resistor. Therefore, the input is protected against temporary overvoltages.

Worst-Case Input Voltages

Figure 14.13a shows a TTL inverter with an input voltage of V_i and an output voltage of V_o. When V_i is 0 V (grounded), it is in the low state and is designated V_{IL}. With TTL devices, we can increase V_{IL} to 0.8 V and still have a low-state input because the output remains in the high state. In other words, the low-state input voltage V_{IL} can have any value from 0 to 0.8 V. TTL data sheets list the worst-case low input as

$$V_{IL,\max} = 0.8 \text{ V}$$

If the input voltage is greater than this, the output state is unpredictable.

However, suppose V_i is 5 V in Fig. 14.13a. This is a high input and can be designated V_{IH}. This voltage can decrease all the way down to 2 V without changing the output state. In other words, the high-stage input V_{IH}

is from 2 to 5 V; any input voltage in this range produces a low output voltage. Data sheets list the worst-case high input as

$$V_{IH,\min} = 2\text{ V}$$

When the input voltage is less than this, the output state is again unpredictable.

Figure 14.13b summarizes these ideas. As you see, any input voltage less than 0.8 V is a valid low-state input. Any input greater than 2 V is a valid high-state input. Any input between 0.8 and 2 V is indeterminate because there is no guarantee that it will produce the correct output voltage.

Worst-Case Output Voltages

Ideally, the low output state is 0 V, and the high output state is 5 V. We cannot attain these ideal values because of internal voltage drops inside TTL devices. For instance, when the output voltage is low in Fig. 14.13a, Q_4 is saturated and has a small voltage drop across it. With TTL devices, any output voltage from 0 to 0.4 V is considered a low output and is designated V_{OL}. This means the low-state output V_{OL} of a TTL device may have any value between 0 and 0.4 V. Data sheets list the worst-case low output as

$$V_{OL,\max} = 0.4\text{ V}$$

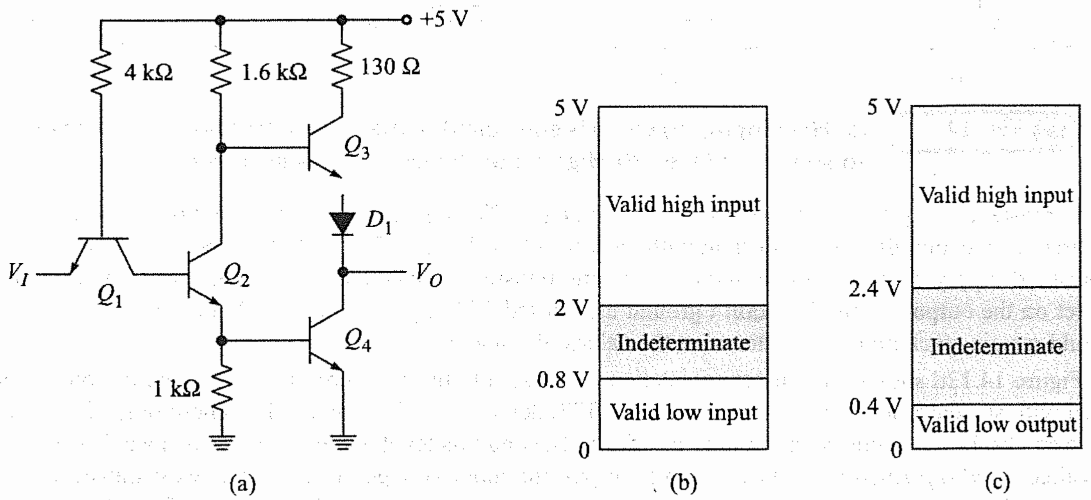


Fig. 14.13 **(a) TTL inverter, (b) TTL input profile, (c) TTL output profile**

When the output is high, Q_3 acts as an emitter follower. Because of the voltage drop across Q_3, D_1, and the 130-Ω resistor, the output voltage will be less than supply voltage. With TTL devices, the high-state output voltage is designated V_{OH}; it has a value between 2.4 and 3.9 V, depending on the supply voltage, temperature, and load. TTL data sheets list the worst-case high output as

$$V_{OH,\min} = 2.4\text{ V}$$

Figure 14.13c summarizes the output states. As shown, any output voltage less than 0.4 V is a valid low-state output, any output voltage greater than 2.4 V is a valid high-state output, and any output between 0.4 and 2.4 V is indeterminate under worst-case conditions.

Profiles and Windows

The input characteristics of Fig. 14.13b are called the TTL input *profile.* Furthermore, each rectangular area in Fig. 14.13b can be thought of as a *window.* There is a low window (0 to 0.8 V), an indeterminate window (0.8 to 2.0 V), and a high window (2.0 to 5 V).

Similarly, Fig. 14.13c is the TTL output profile. Here you see a low window from 0 to 0.4 V, an indeterminate window from 0.4 to 2.4 V, and a high window from 2.4 to 5 V.

Values to Remember

We have discussed the low and high states for the input and output voltages. Here they are again as a reference for future discussions:

$$V_{IL,\max} = 0.8\text{ V}$$
$$V_{IH,\min} = 2\text{ V}$$
$$V_{OL,\max} = 0.4\text{ V}$$
$$V_{OH,\min} = 2.4\text{ V}$$

These are the worst-case values shown in Fig. 14.13b and c. On the input side, a voltage has to be less than 0.8 V to qualify as a low-state input, and it must be more than 2 V to be considered a high-state input. On the output side, the voltage has to be less than 0.4 V to be a low-state output and more than 2.4 V to be a high-state output.

Compatibility

TTL devices are *compatible* because the low and high output windows fit inside the low and high input windows. Therefore the output of any TTL device is suitable for driving the input of another TTL device. For instance, Fig. 14.14a shows one TTL device driving another. The first device is called a *driver* and the second a *load.*

Figure 14.14b shows the output stage of the TTL driver connected to the input stage of the TTL load. The driver output is shown in the low state. Since any input less than 0.8 V is a low-state input, the driver output (0 to 0.4 V) is compatible with the load input requirements.

Similarly, Fig. 14.14c shows high TTL output. The driver output (2.4 to 3.9 V) is compatible with the load input requirements (greater than 2 V).

Sourcing and Sinking

When a standard TTL output is low (Fig. 14.14b), an emitter current of approximately 1.6 milliamperes (mA) (worst case) exists in the direction shown. The conventional flow is from the emitter of Q_1 to the collector of Q_4. Because it is saturated, Q_4 acts as a *current sink*; conventional current flows through Q_4 to ground like water flowing down a sink.

However, when the standard TTL output is high (Fig. 14.14c), a reverse emitter current of 40 microamperes (μA) (worst-case) exists in the direction shown. Conventional current flows out of Q_3 to the emitter of Q_1. In this case, Q_3 is acting as a *source.*

Data sheets list the worst-case input currents:

$$I_{IL,\max} = -1.6\text{mA} \quad I_{IH,\max} = 40\ \mu\text{A}$$

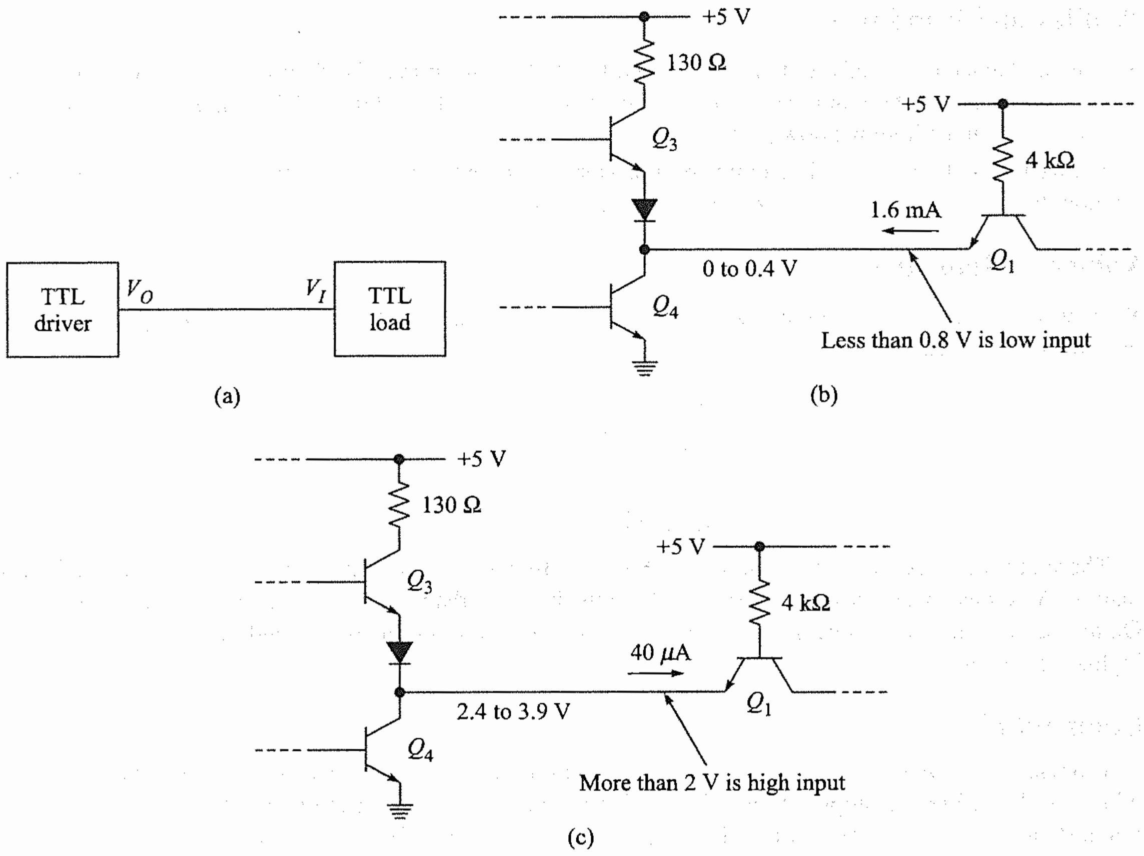


Fig. 14.14 **Sourcing and sinking current**

The minus sign indicates that the conventional current is out of the device; a plus sign means the conventional current is into the device. All data sheets use this notation, so do not be surprised when you see minus currents. The previous data tells us the maximum input current is 1.6 mA (outward) when an input is low and 40 μA (inward) when an input is high.

Noise Immunity

In the worst case, there is a difference of 0.4 V between the driver output voltages and required load input voltages. For instance, the worst-case low values are

$$V_{OL,\max} = 0.4 \text{ V} \quad \text{driver output}$$
$$V_{IL,\max} = 0.8 \text{ V} \quad \text{load input}$$

Similarly, the worst-case high values are

$$V_{OH,\min} = 2.4 \text{ V} \quad \text{driver output}$$
$$V_{IH,\min} = 2 \text{ V} \quad \text{load input}$$

In either case, the difference is 0.4 V. This difference is called *noise immunity*. It represents built-in protection against noise. Note that the concept of noise margin has been introduced in section 1.8 of Chapter 1.

Why do we need protection against noise? The connecting wire between a TTL driver and load is equivalent to a small antenna that picks up stray noise signals. In the worst case, the low input to the TTL load is

$$V_{IL} = V_{OL} + V_{\text{noise}} = 0.4\text{ V} + V_{\text{noise}}$$

and the high-stage input is

$$V_{IH} = V_{OH} - V_{\text{noise}} = 2.4\text{ V} - V_{\text{noise}}$$

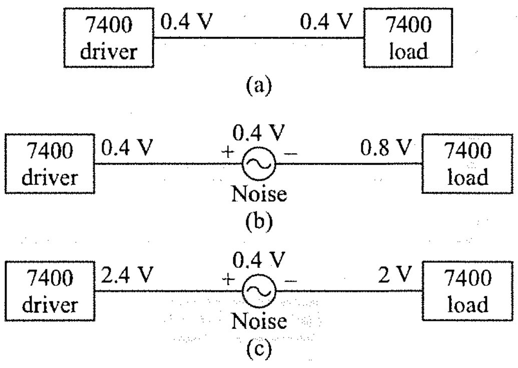


Fig. 14.15 **(a) TTL driver and load, (b) False triggering into high state, (c) False triggering into low state**

In most environments, the induced noise voltage is less than 0.4 V, and we get no false triggering of the TTL load.

For instance, Fig. 14.15a shows a low output from the TTL driver. If no noise voltage is induced on the connecting wire, the input voltage to the TTL load is 0.4 V, as shown. In a noisy environment, however, it is possible to have 0.4 V of induced noise on the connecting wire for either the low state (Fig. 14.15b) or the high state (Fig. 14.15c). Either way, the TTL load has an input that is on the verge of being unpredictable. The slightest additional noise voltage may produce a false change in the output state of the TTL load.

Standard Loading

A TTL device can source current (high output) or sink current (low output). Data sheets of standard TTL devices indicate that any 7400 series device can sink up to 16 mA, designated

$$I_{OL,\max} = 16\text{ mA}$$

and can source up to 400 μA, designated

$$I_{OH,\max} = -400\ \mu\text{A}$$

(Again, a minus sign means that the conventional current is out of the device, and a plus sign means that it is into the device.) As discussed earlier, the worst-case TTL input currents are

$$I_{IL,\max} = -1.6\text{ mA} \quad I_{IH,\max} = 40\ \mu\text{A}$$

Since the maximum output currents are 10 times larger than the input currents, we can connect up to 10 TTL emitters to any TTL output.

As an example, Fig. 14.16a shows a low output voltage (worst case). Notice that a single TTL driver is connected to 10 TTL loads (only the input emitters are shown). Here you see the TTL driver sinking 16 mA, the sum of the 10 TTL load currents. In the low state, the output voltage is guaranteed to be 0.4 V or less. If you try connecting more than 10 emitters, the output voltage may rise above 0.4 V under worst-case conditions. If this happens, the low-state operation is no longer reliable. Therefore, 10 TTL loads are the maximum that a manufacturer allows for guaranteed low-state operation.

Figure 14.16b shows a high output voltage (worst case) with the driver sourcing 400 μA for 10 TTL loads of 40 μA each. For this source current, the output voltage is guaranteed to be 2.4 V or greater under worst-case conditions. If you try to connect more than 10 TTL loads, you will exceed $I_{OH,\max}$ and high-state operation becomes unreliable.

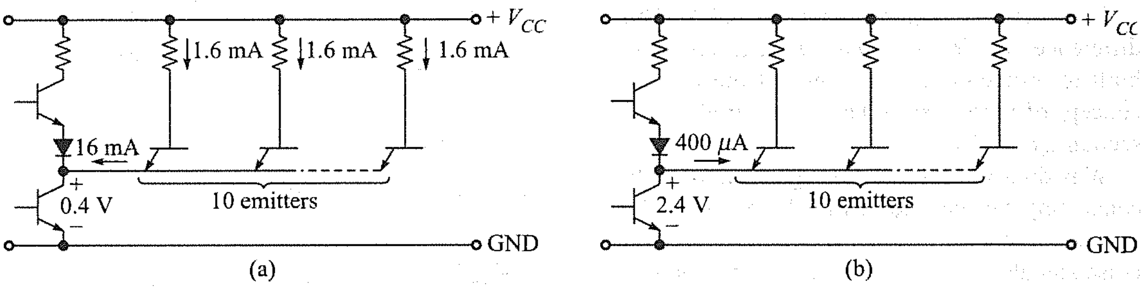


Fig. 14.16 (a) Low-state fanout, (b) High-state fanout

Loading Rules

Figure 14.17 shows the output-input profiles for different types of TTL. The output profiles are on the left, and the input profiles are on the right. These profiles are a concise summary of the voltages and currents for each TTL type. Start with the profiles of Fig. 14.17a; these are for standard TTL. On the left, you see the profile of output characteristics. The high output window is from 2.4 to 5 V with up to 400 μA of source current; the low output window is from 0 to 0.4 V with up to 16 mA of sink current. On the right, you see the input profile of a standard TTL device. The high window is from 2 to 5 V with an input current of 40 μA, while the low window is from 0 to 0.8 V with an input current of 1.6 mA.

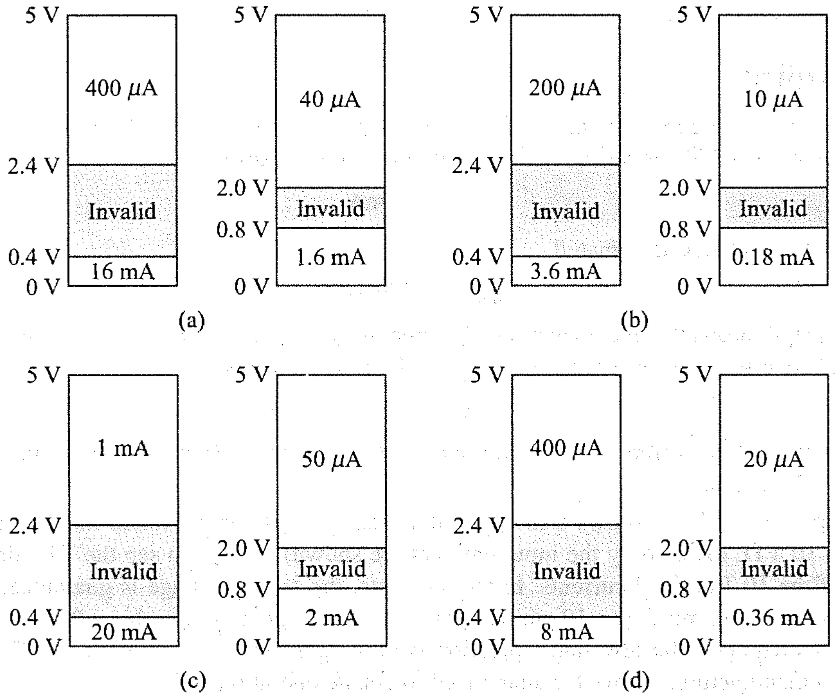


Fig. 14.17 TTL output-input profiles: (a) Standard TTL, (b) Low-power TTL, (c) Schottky TTL, (d) Low-power Schottky TTL

Standard TTL devices are compatible because the low and high output windows fit inside the corresponding input window. In other words, 2.4 V is always large enough to be a high input to a TTL load, and 0.4 V is always small enough to be a low input. Furthermore, you can see at a glance that the available source current is 10 times the required high-state input current, and the available sink current is 10 times the required low-state input current. The maximum number of TTL loads that can be reliably driven under worst-case conditions is called the *fanout*. With standard TTL, the fanout is 10 because one TTL driver can drive 10 TTL loads.

The remaining figures all have identical voltage windows. The output states are always 0 to 0.4 and 2.4 to 5 V, while the input states are 0 to 0.8 and 2 to 5 V. For this reason, all the TTL types are compatible; this means you can use one type of TTL as a driver and another type as a load.

The only differences in the TTL types are the currents. You can see in Fig. 14.17a to *d* that the input and output currents differ for each TTL type. For instance, a low-power Schottky TTL driver (see Fig. 14.17d) can source 400 μA and sink 8 mA; a low-power Schottky load requires input currents of 20 μA (high state) and 0.36 mA (low state). These numbers are different from standard TTL (Fig. 14.17a) with its output currents of 400 μA and 16 mA and its input currents of 40 μA and 1.6 mA.

Incidentally, notice that the profiles of high-speed TTL are omitted in Fig. 14.17 because Schottky TTL has replaced high-speed TTL, in virtually all applications. If you need high-speed TTL data, consult manufacturers' catalogs.

By analyzing Fig. 14.17*a* to *d* (plus the data sheets for high-speed TTL), we can calculate the fanout for all possible combinations. Table 14.4 summarizes these fanouts, which are useful if you ever have to mix TTL types.

Read Table 14.3 as follows. The TTL types have been abbreviated; 74 stands for 7400 series (standard), 74H for 74H00 series (high speed), and so forth. Drivers are on the left, and loads are on the right. Pick the driver, pick the load, and read the fanout at the intersection of the two. For instance, the fanout of a standard device (74) driving low-power Schottky devices (74LS) is 20. As another example, the fanout of a low-power device (74L) driving high-speed devices (74H) is only 1.

Table 14.3 Fanouts

TTL Driver	*TTL Load*				
	74	*74H*	*74L*	*74S*	*74LS*
74	10	8	40	8	20
74H	12	10	50	10	25
74L	2	1	20	1	10
74S	12	10	100	10	50
74LS	5	4	40	4	20

SELF-TEST

7. Why should TTL gate inputs never be left floating?
8. What is a TTL input profile?
9. What is the delay time for a 74LS04?

14.4 TTL OVERVIEW

Let's take a look at the logic functions available in the 7400 series. This overview will give you an idea of the variety of gates and circuits found in the TTL family. As a guide, Appendix 3 lists some of the 7400 series devices.

NAND Gates

The NAND gate is the backbone of the 7400 series. All devices in this series are derived from the 2-input NAND gate shown in Fig. 14.10. To produce 3-, 4-, and 8-input NAND gates, the manufacturer uses 3-, 4-, and 8-emitter transistors. Because they are so basic, NAND gates are the least expensive devices in the 7400 series.

NOR Gates

To get other logic functions the manufacturer modifies the basic NAND-gate design. For instance. Fig. 14.18 shows a 2-input NOR gate. Here Q_5 and Q_6 have been added to basic NAND-gate design. Since Q_2 and Q_6 are in parallel, we get the OR function, which is followed by inversion to get the NOR function.

When A and B are both low, the bases of Q_1 and Q_5 are pulled low; this cuts off Q_2 and Q_6. Then Q_3 acts as an emitter-follower, and we get a high output.

If A or B is high, Q_1 and Q_5 are cut off, forcing Q_2 or Q_6 to turn on. When this happens, Q_4 saturates and pulls the output down to a low voltage.

With more transistors, a manufacturer can produce 3- and 4-input NOR gates. (*Note*: A TTL 8-input NOR gate is not available.)

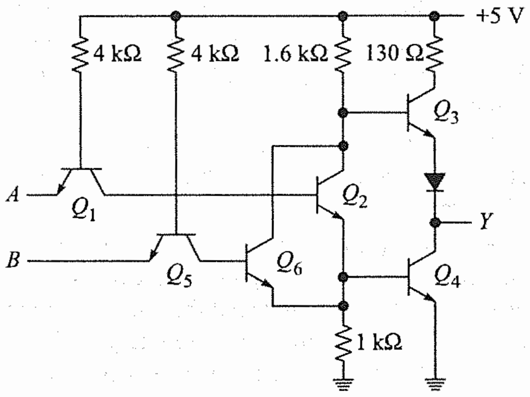


Fig. 14.18 TTL NOR gate

AND and OR Gates

To produce the AND function, another inverting stage is inserted in the basic NAND-gate design. The extra inversion converts the NAND gate to an AND gate. The available TTL AND gates are the 7408 (quad 2-input), 7411 (triple 3-input), and 7421 (dual 4-input).

Similarly, another inverting stage can be inserted in the NOR gate of Fig. 14.18; this converts the NOR gate to an OR gate. The only available TTL OR gate is the 7432 (quad 2-input).

Buffer Drivers

All IC buffer can source and sink more current than a standard TTL gate. As an example, the 7437 is a quad 2-input NAND buffer, meaning four 2-input NAND gates optimized to get high output currents. Each gate has the following worst-case currents:

$$I_{IL} = -1.6 \text{ mA} \qquad I_{IH} = 40\ \mu\text{A}$$
$$I_{OL} = 48 \text{ mA} \qquad I_{OH} = -1.2 \text{ mA}$$

The input currents are the same as those of a 7400 (standard TTL NAND gate), but the output currents are 3 times as high. This means that a 7437 can drive heavier loads. In other words, the fanout of a 7437 is 3 times that of a 7400. Appendix 3 includes several other buffer-drivers.

Hex schmitt-trigger (discussed in Chapter 7) inverter IC 7414 is shown in Fig. 14.19a.

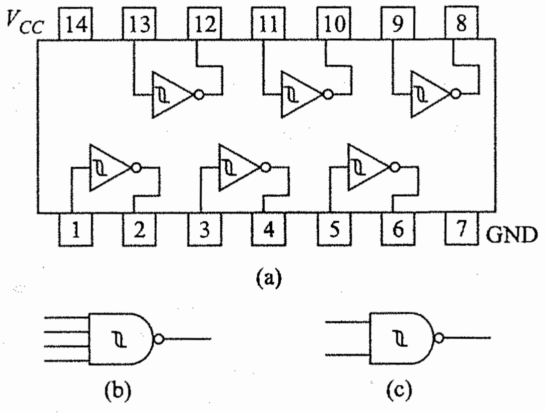


Fig. 14.19 (a) Hex Schmitt-trigger inverters, (b) 4-input NAND Schmitt trigger, (c) 2-input NAND Schmitt trigger

AND-or-INVERT Gates

Figure 14.20 shows the schematic diagram of an AND-OR-INVERT circuit. Here, Q_1, Q_2, Q_3 and Q_4 form the basic 2-input NAND gate of the 7400 series. By adding Q_5 and Q_6, we convert the basic NAND gate to an AND-OR-INVERT gate. Both Q_1 and Q_5 act as 2-input AND gates; Q_2 and Q_6 produce ORing and inversion. Because of this, the circuit is logically equivalent to Fig. 14.20b. This circuit represents 2-input, 2-wide AND-OR-INVERT gate. It is 2-input as each AND gate has 2 inputs and it is 2-wide because there are 2 AND gates at input stage. Figure 14.21a shows the schematic diagram of an expandable AND-OR-INVERT gate. As the name suggests, many such gates put together can expand the width at the input side. The difference between this and preceding AND-OR-INVERT gate (Fig. 14.20) is the collector and emitter pin brought outside the package.

Since Q_2 and Q_6 are the key to the ORing operation, we are being given access to the internal ORing function. By connecting other gates to these new inputs, we can expand the width of the AND-OR-INVERT gate.

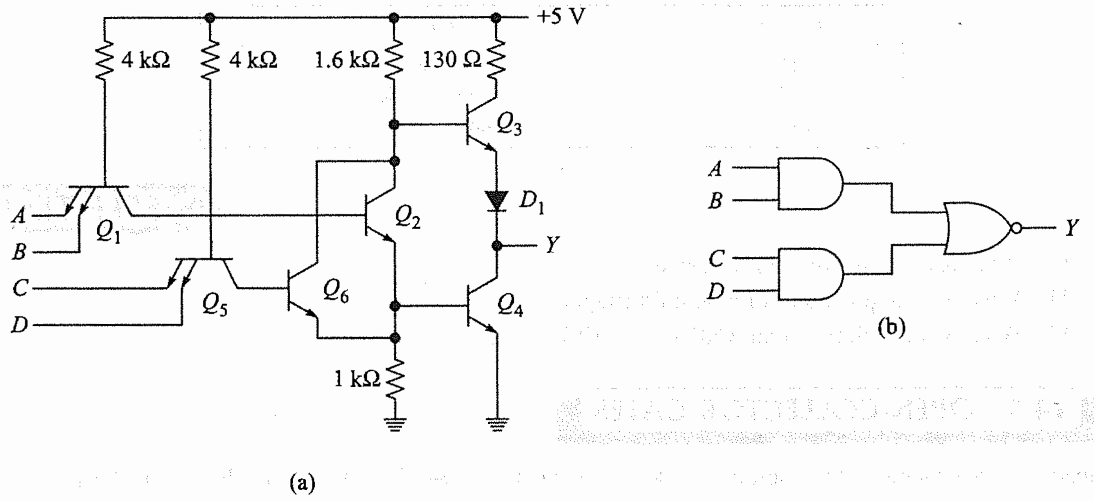
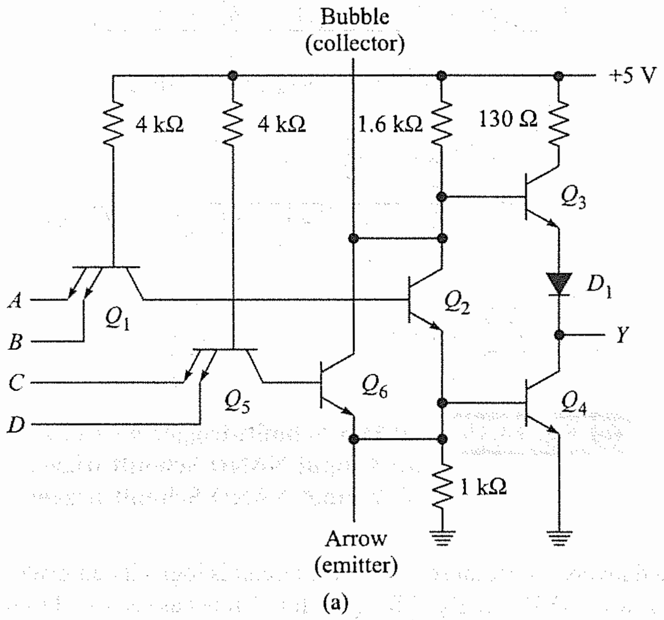


Fig. 14.20 (a) AND-OR-INVERT schematic diagram, (b) Circuit



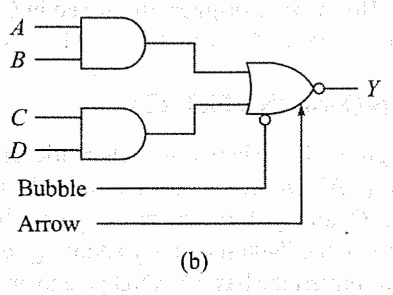


Fig. 14.21 **(a) Expandable AND-OR-INVERT gate, (b) Logic symbol**

Figure 14.21b shows the logic symbol for an expandable AND-OR-INVERT gate. The arrow input represents the emitter and the bubble stands for the collector. Table 14.4 lists the expandable AND-OR-INVERT gates in the 7400 series.

Table 14.4 **Expandable AND–OR–INVERT Gates**

Device	*Description*
7450	Dual 2-input 2-wide
7453	2-input 4-wide
7455	4-input 2-wide

SELF-TEST

10. What is the value of a buffer-driver?
11. What is an application for a Schmitt trigger?
12. What is the "width" of an AND-OR-INVERT gate?

14.5 OPEN-COLLECTOR GATES

Instead of a totem-pole output, some TTL devices have an *open-collector output*. This means they use only the lower transistor of a totem-pole pair. Figure 14.22a shows a 2-input NAND gate with an open-collector output. Because the collector of Q_4 is open, a gate like this will not work properly until you connect an external pull-up resistor, shown in Fig. 14.22b.

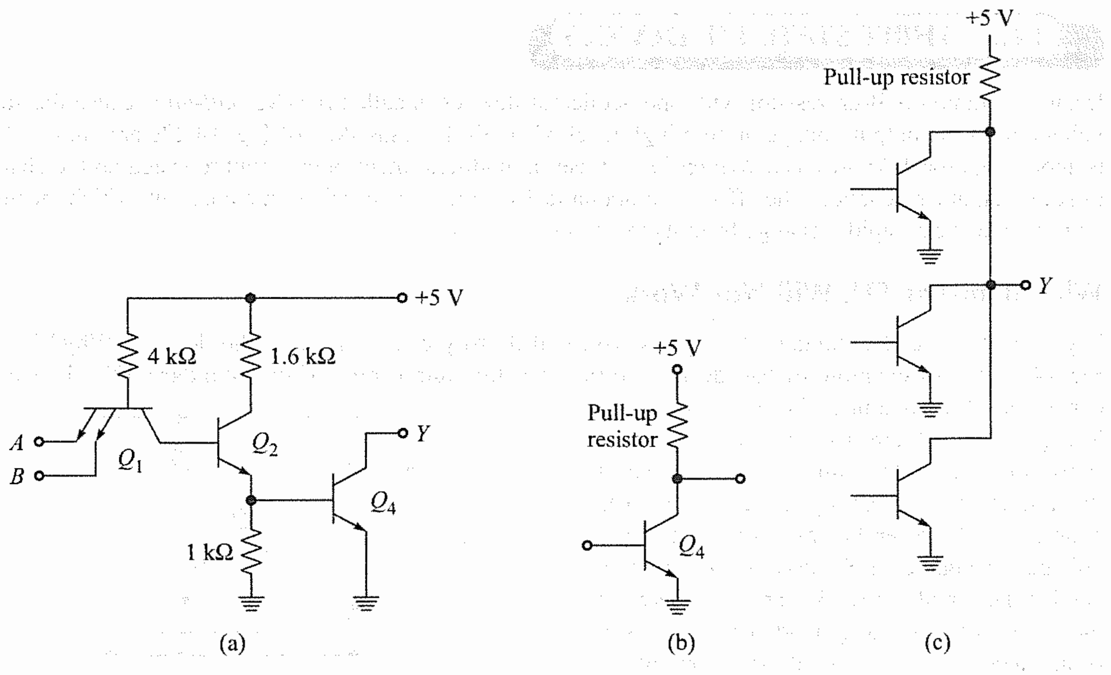


Fig. 14.22 **Open-collector TTL: (a) Circuit, (b) Pull-up resistor, (c) Open-collector outputs connected to a pull-up resistor**

The outputs of open-collector gates can be wired together and connected to a common pull-up resistor. For instance, Fig. 14.22c shows three TTL devices connected to the pull-up resistor. This is known as *wire-OR* (some called it wire-AND). A connection like this has the advantage of combining the output of three devices without using a final OR gate (or AND gate). The combining is done by a direct connection of the three outputs to the lower end of the common pull-up resistor. This is very useful when many devices are wire-ORed together. For instance, in some systems the outputs of 16 open-collector devices are connected to a pull-up resistor.

The big disadvantage of open-collector gates is their slow switching speed. Why is it slow? Because the pull-up resistance is a few kilohms, which results in a relatively long time constant when it is multiplied by the stray output capacitance. The slow switching speed of open-collector TTL devices is worst when the output goes from low to high. Imagine all three transistors going into cutoff in Fig. 14.22c. Then any capacitance across the output has to charge through the pull-up resistor. This charging produces a relatively slow exponential rise between the low and high state.

SELF-TEST

13. What must be connected to the output of an open-collector TTL gate?
14. Open-collector gates have (slower, faster) switching times.

14.6 THREE-STATE TTL DEVICES

Using a common pull-up resistor with open-collector devices is called *passive pull-up* because the supply voltage pulls the output voltage up to a high level when all the transistors of Fig. 14.22c are cut off. There is another approach known as *active pull-up*. It uses a modified totem-pole output to speed up the charging of stray output capacitance. The effect is to dramatically lower the charging time constant, which means the output voltage can rapidly change from its low to its high stage.

Why Standard TTL Will Not Work

If you try to wire-OR standard TTL gates, you will destroy one or more of the devices. Why? Look at Fig. 14.23 for an example of bad design. Notice that the output pins of two standard TTL devices are connected. If the output of the second device is low, Q_4 is on and appears approximately like a short circuit. If, at the same time, the output of the first device is in the high state, then Q_1 acts as an emitter follower that tries to pull the output voltage to a high level. Since Q_1 and Q_4 are both conducting heavily, only 130 Ω remains between the supply voltage and ground. The final result is an excessive current that destroys one of the TTL devices.

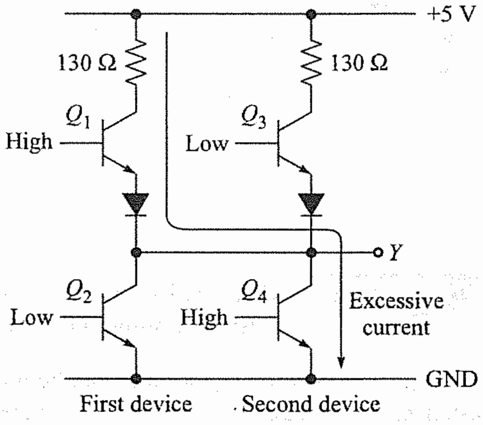


Fig. 14.23 Direct connection of TTL outputs produces excessive current

Low DISABLE Input

As you have seen, wire-ORing standard TTL devices will not work because of destructive currents in the output stages. This inability to wire-OR ordinary totem-pole devices is what led to *three-state (tri-state) TTL*, a new breed of totem-pole devices introduced in the early 1970s. With three-state gates, we can connect totem-pole outputs directly without destroying any devices. The reason for wanting to use totem-pole outputs is to avoid the loss of speed that occurs with open-collector devices.

Figure 14.24 shows a simplified drawing for a three-state inverter. When DISABLE is low, the base and collector of Q_6 are pulled low. This cuts off Q_7 and Q_8. Therefore, the second emitter of Q_1 and the cathode of D_1 are floating. For this condition, the rest of the circuit acts as an inverter: a low A input forces Q_2 and Q_5 to cut off, while Q_3 and Q_4 turn on, producing a high output. On the other hand, a high A input forces Q_2 to turn on, which drives Q_5 on and produces a low output. Table 14.5 summarizes the operation for low DISABLE.

Table 14.5 Three-State Inverter

Disable	*A*	*Y*
0	0	1
0	1	0
1	*X*	Hi-Z

High DISABLE Input

When DISABLE is high, the base and collector of Q_6 go high, which turns on Q_7 and Q_8. Ideally, the collector of Q_8 is pulled down to ground. This causes the base and collector of Q_1 to go low, cutting off Q_2 and Q_5.

Also Q_3 is off because of the clamping action of D_1. In other words, the base of Q_3 is only 0.7 V above ground, which is insufficient to turn on Q_3 and Q_4.

With both Q_4 and Q_5 off, the Y output is floating. Ideally, this means that the Thévenin impedance looking back into the Y output approaches infinity. Table 14.5 summarizes the action for this high-impedance state. As shown, when DISABLE is high, input A is a don't care because it has no effect on the Y output. Furthermore, because of the high output impedance, the output line appears to be disconnected from the rest of the gate. In effect, the output line is floating.

In conclusion, the output of Fig. 14.24 can be in one of three states: low, high, or floating.

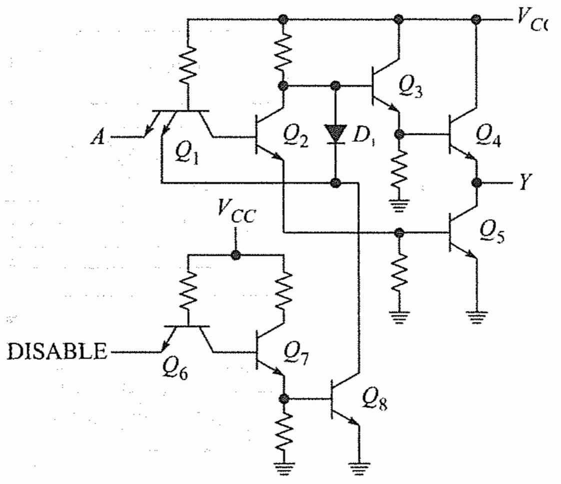


Fig. 14.24 Three-state inverter

Logic Symbol

Figure 14.25a is an equivalent circuit for the three-state inverter. When DISABLE is low, the switch is closed and the circuit acts as an ordinary inverter. When DISABLE is high, the switch is open and the Y output is floating or disconnected.

Figure 14.25b shows the logic symbol for a three-state inverter. When you see this symbol, remember that a low DISABLE results in normal inverter action, but a high DISABLE floats the Y output.

Three-State Buffer

By modifying the design, we can produce a three-state buffer, whose logic symbol is shown in Fig. 14.25c. When DISABLE is low, the circuit acts as a noninverting buffer, so that $Y = A$. But when DISABLE is high, the output floats. The three-state buffer is equivalent to an ordinary switch. When DISABLE is low, the switch is closed. When DISABLE is high, the switch is open.

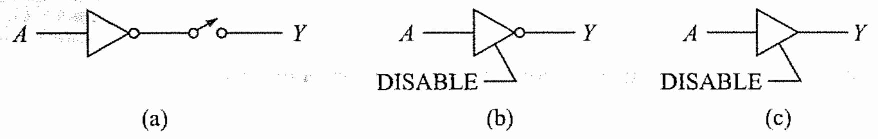


Fig. 14.25 Three-state logic diagrams: (a) Equivalent circuit of inverter, (b) Logic symbol of inverter, (c) Logic symbol of buffer

The 74365 is an example of a commercially available three-state hex noninverting buffer. This IC contains six buffers with three-state outputs. It is ideal for organizing digital components around a *bus*, a group of wires that transmits binary numbers between registers.

Bus Organization

Figure 14.26 shows some registers connected to a common bus. The three-state buffers control the flow of binary data between the registers. For instance, if we want the contents of register A to appear on the bus, all we have to do is make DISABLE low for register A but high for registers B and C. Then all the three-state

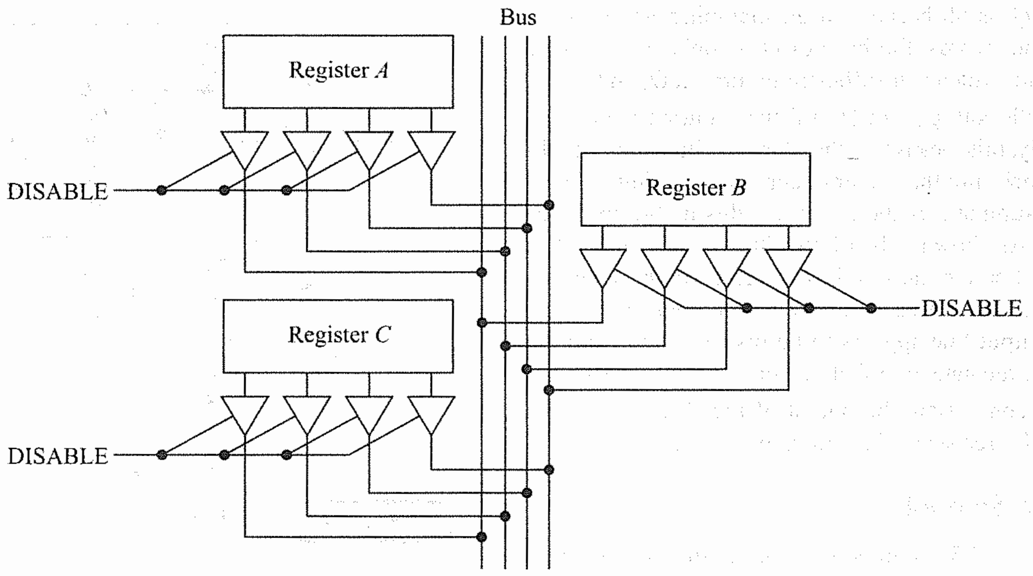


Fig. 14.26 **Three-state bus control**

switches on register A are closed, while all other three-state switches are open. As a result, only the contents of register A appear on the bus.

The idea in any bus-organized system is to make DISABLE high for all registers except the register whose contents are to appear on the bus. In this way, dozens of registers can time-share the same transmission path. Not only does this reduce the amount of wiring, but also it has simplified the architecture and design of computers and other digital systems. Refer to simple computer design discussed in Chapter 16.

SELF-TEST

15. Why are three-state gates used in conjunction with computer buses?

14.7 EXTERNAL DRIVE FOR TTL LOADS

To drive a TTL load with an external source, you need to satisfy the TTL input requirements for voltage and current. For standard TTL in the low state, this means an input voltage between 0 and 0.8 V with a current of approximately 1.6 mA. In the high state, the voltage has to be from 2 to 5 V with a current of approximately 40 μA. Let us take a brief look at some of the ways to drive a TTL load.

Switch Drive

Figure 14.27 shows the preferred method for driving a TTL input from a switch. With the switch open, the input is pulled up to +5 V. In the worst case, only 40 μA of input current exists. Therefore, the voltage appearing at the input pin is slightly less than the supply voltage because of the small voltage drop across the pull-up resistor:

$$V_i = 5\text{ V} - (40\ \mu\text{A})(1\text{ k}\Omega) = 4.96\text{ V}$$

This is well above the minimum requirement of 2 V, which is fine because it means that the noise immunity is excellent.

When the switch is closed, the input is pulled down to ground. In the worst case, the input current is 1.6 mA. This sink current creates no problem because it flows through the closed switch to ground. The noise immunity is fine because the input voltage is 0 V, well below the maximum allowable value of 0.8 V.

Size of Pull-Up Resistance

A pull-up resistance of 1 kΩ is nominal. You can use other values. Here are some of the factors to consider when you are selecting a pull-up resistor. In Fig. 14.27, the current drain with a closed switch is

$$I = \frac{5\text{ V}}{1\text{ k}\Omega} = 5\text{ mA}$$

The smaller the pull-up resistance, the larger the current drain. At some point, too much current drain becomes a problem for the power supply, so you have to use a resistance that is large enough to keep the current drain to tolerable levels.

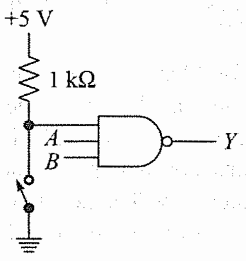


Fig. 14.27 **Switch drive for TTL input**

On the other hand, too large a pull-up resistance causes speed problems. The worst case occurs when the switch is opened. For instance, if the input capacitance is 10 picofarads (pF) in Fig. 14.27, the time constant is

$$RC = (1\text{ k}\Omega)(10\text{ pF}) = 10\text{ ns}$$

The larger the pull-up resistance, the larger the time constant. A larger time constant means a slower switching speed because the input capacitance has to charge through the pull-up resistance.

Pull-up resistances between 1 and 10 kΩ are typical. They result in current drains and time constants that are acceptable in most applications.

Transistor Drive

Figure 14.28a shows another way to drive a TTL laid. This time, we are using a transistor switch to control the state of the TTL input. When V_i is low, the transistor is off and is equivalent to an open switch. Then the TTL input is pulled up to +5 V through a resistance of 1 kΩ. When V_i is high, the transistor is on and is equivalent to a closed switch. In this case, it easily sinks the 1.6 mA of input current.

The transistor inverts the control signal V_i. If this is objectionable, you can insert an inverter as shown in Fig. 14.28b. Now, the double inversion produces an in-phase control signal at the TTL input.

Operational Amplifier Drive

Sometimes, you want to use the output of an operational amplifier (OA) to control a TTL input. Because OAs typically use split-supply voltages of +15 and −15 V, you have to be careful how you connect to the TTL

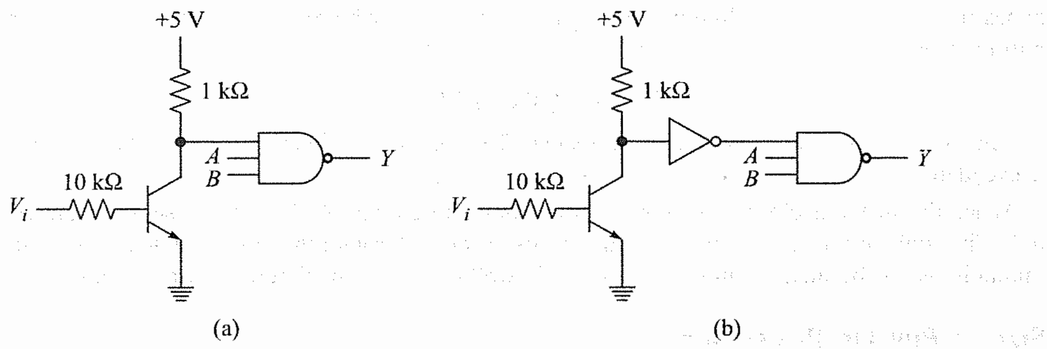


Fig. 14.28 (a) Transistor drive for TTL input, (b) Inverter eliminates transistor inversion

load. Figure 14.29 shows one way to use the output of a 741 to control a TTL input. The output of the OA ideally swings from +15 to −15V. The positive swing closes the transistor switch, producing a TTL input of approximately 0 V. The negative swing drives the transistor into cutoff, producing a TTL input of +5V.

Notice the diode in the base circuit. It protects the base against excessive reverse voltage. The data sheet of a 2N3904 indicates an absolute maximum base-emitter voltage rating of

$$V_{BE,\max} = -6\text{ V}$$

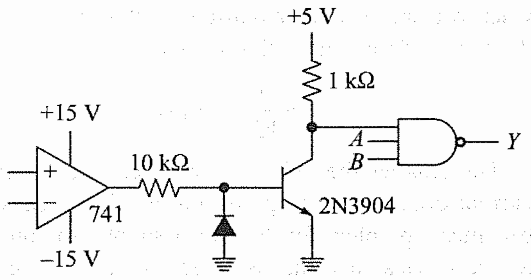


Fig. 14.29 Op amp and transistor drive for TTL input

Since the negative output of the OA approaches −15 V, we need to use a protective diode as shown between the base and ground. This diode clamps the base voltage at approximately −0.7 V on the negative swing.

Comparator Drive

Figure 14.30a shows the schematic diagram for a typical comparator, an IC that detects when the input voltage is positive or negative. Notice two things. First, a supply voltage of +15 V is typically used with this kind of device. Second, the comparator has an open-collector output transistor, Q_5. This sink transistor can be connected to any supply voltage.

Figure 14.30b shows how to connect an LM339 (typical comparator) to a TTL load. Because of the open-collector output, we can connect the output pin of the comparator to a supply voltage of +5 V through a pull-up resistance of 1 kΩ. When V_i is positive, the sink transistor goes off and the TTL input is pulled high. When V_i is negative, the sink transistor goes on and the TTL output is pulled low.

SELF-TEST

16. What factors influence the size of the resistor in Fig. 14.29?
17. What is the purpose of the diode in Fig. 14.31?

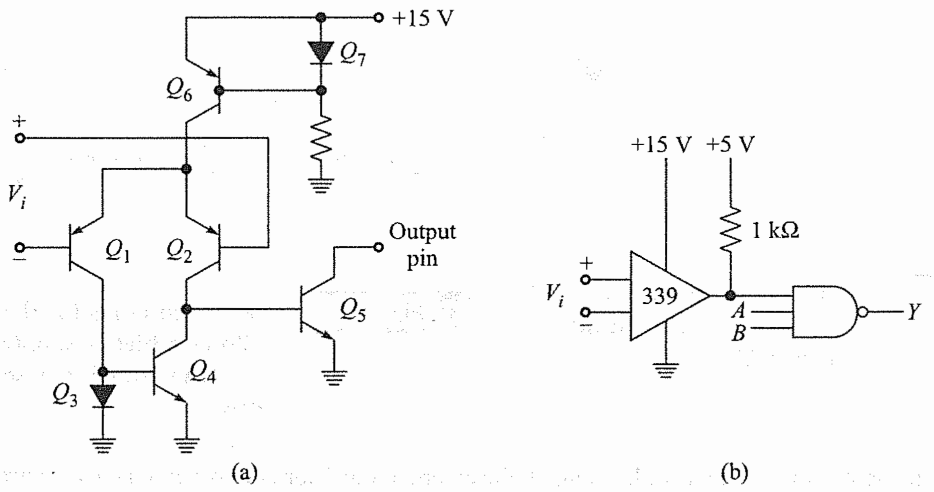


Fig. 14.30 (a) Schematic diagram of comparator, (b) Interfacing an LM339 to a TTL input

14.8 TTL DRIVING EXTERNAL LOADS

Because standard TTL can sink up to 16 mA, you can use a TTL driver to control an external load such as a relay, an LED, etc. Figure 14.31a illustrates the idea. When the TTL output is high, there is no load current. But when the TTL output is low, the lower end of R_L is ideally grounded. This sets up a load current of approximately

$$I_L = \frac{5\text{ V}}{R_L}$$

Since standard TTL can sink a maximum of 16 mA, the load resistance is limited to a minimum value of about

$$R_L = \frac{5\text{ V}}{16\text{ mA}} = 312\ \Omega$$

Driving an LED

Figure 14.31b is another example. Here a TTL circuit drives an LED. When the TTL output is high, the LED is dark. When the TTL output is low, the LED lights up. If the LED voltage drop is 2 V, the LED current for a low TTL output is approximately

$$I_L = \frac{5\text{ V} - 2\text{ V}}{270\ \Omega} = 11.1\text{ mA}$$

Supply Voltage Different from +5 V

If you need to use a supply voltage different from +5 V, you can use an open-collector TTL device. For instance, Fig. 14.32a on the next page shows an open-collector gate driving a load resistor that is returned to +15 V. Since an open-collector device can sink a maximum of 16 mA, the minimum load resistance in Fig. 14.32a is slightly less than 1 kΩ.

If you want more than 16 mA of load current, you can use an external transistor, as shown in Fig. 14.32b.

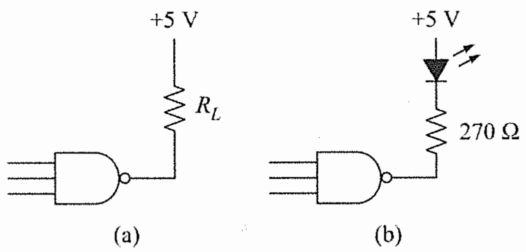


Fig. 14.31 **(a) TTL output drives load resistor, (b) TTL output drives LED**

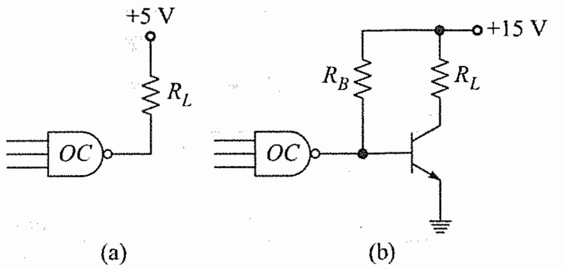


Fig. 14.32 **(a) Open-collector device allows a higher supply voltage, (b) Transistor increases current drive**

When the open-collector device has a low output, the external transistor goes off and the load current is zero. When the device has a high output, the external transistor goes on and the load current is maximum.

SELF-TEST

18. Could the resistor R_L in Fig. 14.34 be replaced with a red LED?

14.9 74C00 CMOS

National Semiconductor Corporation pioneered the 74C00 series, a line of CMOS circuits that are pin-for-pin and function-for-function compatible with TTL devices of similar numbers. For instance, the 74C00 is a quad 2-input NAND gate, the 74C02 is a quad 2-input NOR gate, and so on. This CMOS family contains a variety of small-scale integration (SSI) and medium-scale integration (MSI) chips that allow you to replace many TTL designs by the comparable CMOS designs. This is useful if you are trying to build battery-powered equipment. The 74HC00 series is the high-speed CMOS family.

NAND Gate

Figure 14.33 shows a CMOS NAND gate. The complementary design of input and output stages is typical of CMOS devices. Notice that Q_1 and Q_2 form one complementary connection; Q_3 and Q_4 form another. Visualize these transistors as switches. Then a low A input will close Q_1 and open Q_2; a high A input will open Q_1 and close Q_2. Similarly, a low B input will open Q_3, and close Q_4; a high B input will close Q_3 and open Q_4.

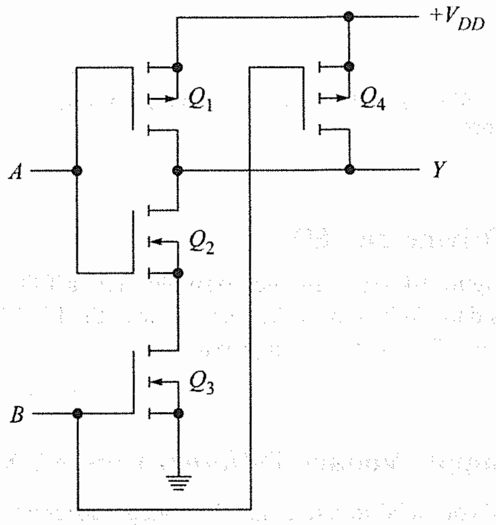


Fig. 14.33 **CMOS NAND gate**

In Fig. 14.33, the Y output is pulled up to the supply voltage when either Q_1 or Q_4 is conducting. The output is pulled down to ground only when Q_2 and Q_3 are conducting. If you keep this in mind, it simplifies the following discussion.

Case 1 Here A is low and B is low. Because A is low, Q_1 is closed. Therefore, Y is pulled high through the small resistance of Q_1.

Case 2 Now A is low and B is high. Since A is still low, Q_1 remains closed and Y stays in the high state.

Case 3 The A input is high and the B is low. Because B is low, Q_4 is closed. This pulls Y up to the supply voltage through the small resistance of Q_4.

Case 4 The A is high, and the B is high. When both inputs are high, Q_2 and Q_3 are closed, pulling the output down to ground.

Table 14.6 summarizes all input-output possibilities. As you can see, this is the truth table of a positive NAND gate. The output is low only when all inputs are high. To produce the positive AND function, we can connect the output of Fig. 14.33 to a CMOS inverter.

Table 14.6 CMOS NAND Gate

A	B	Y
Low	Low	High
Low	High	High
High	Low	High
High	High	Low

NOR Gate

Figure 14.34 shows a CMOS NOR gate. The output goes high only when Q_1 and Q_2 are closed. The output goes low if either Q_3 or Q_4 is closed. There are four possible cases:

Case 1 The A is low, and the B is low. For both inputs low, Q_1 and Q_2 are closed. Therefore, Y is pulled high though the small series resistance of Q_1 and Q_2.

Case 2 The A is low, and the B is high. Because B is high, Q_3 is closed, pulling the output down to ground.

Case 3 The A is high, and the B is low. With A high, Q_4 is closed. The closed Q_4 pulls the output low.

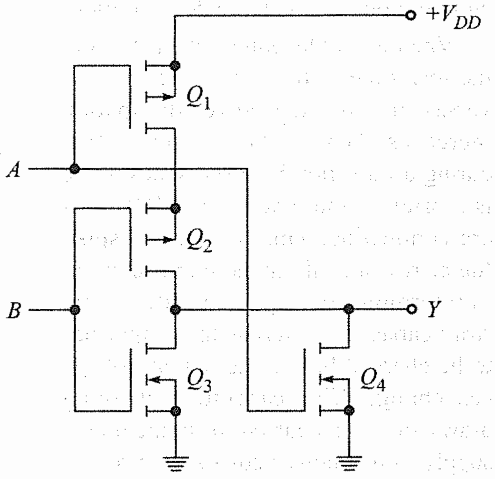


Fig. 14.34 CMOS NOR gate

Table 14.7

A	B	Y
Low	Low	High
Low	High	Low
High	Low	Low
High	High	Low

Case 4 The A is high, and the B is high. Since A is still high, Q_4 is still closed and the output remains low.

Table 14.7 summarizes these possibilities. As you can see, this is the truth table of a positive NOR gate. The output is low when any input is high.

Propagation Delay Time

A standard CMOS gate has a propagation delay time t_p of approximately 25 to 100 ns, with the exact value depending on the power supply voltage and other factors. As you recall, t_p is the time it takes for the output of a gate to change after its inputs have changed. When two or more CMOS gates are cascaded, you have to add the propagation delay times to get the total. For instance, if you cascade three CMOS gates each with a t_p of 50 ns, then the total propagation delay time is 150 ns.

Power Dissipation

The *static power dissipation* of a device is its average power dissipation when the output is constant. The static power dissipation of a CMOS gate is in nanowatts. For instance, a 74C00 has a power dissipation of approximately 10 nanowatts (nW) per gate. This dissipation equals the product of supply voltage and leakage current, both of which are dc quantities.

When a CMOS output changes from the low state to the high state (or vice versa), the average power dissipation increases. Why? The reason is that during a transition between states, there is a brief period when both MOSFETs are conducting. This produces a spike (quick rise and fall) in the supply current. Furthermore, during a transition, any stray capacitance across the output has to be charged before the output voltage can change. This capacitive charging draws additional current from the power supply. Since power equals the product of supply voltage and device current, the instantaneous power dissipation increases, which means the average power dissipation is higher.

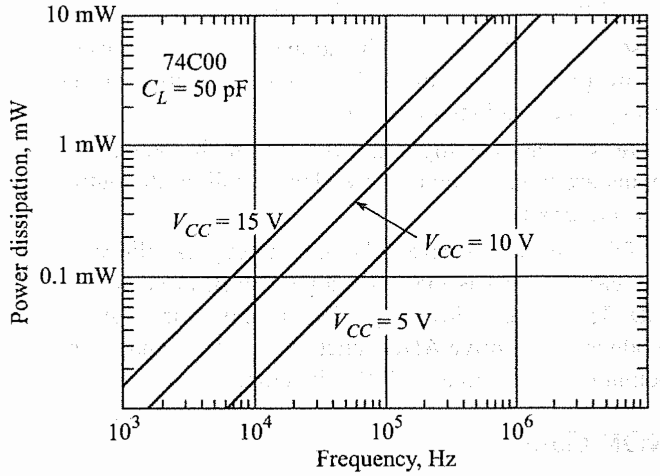


Fig. 14.35 Active power dissipation of a 74C00

The average power dissipation of a CMOS device whose output is continuously changing is called the *active power dissipation*. How large is the active power dissipation? This depends on the frequency at which the output is switching states. When the operating frequency increases, the current spikes occur more often and active power dissipation increases. Figure 14.35 shows the active power dissipation of a 74C00 versus frequency for a load capacitance of 50 pF. As you see, the power dissipation per gate increases with frequency and supply voltage. For frequencies in the megahertz region, the gate dissipation approaches or exceeds 10 mW (TTL gate dissipation). For CMOS to have an advantage over TTL, you operate CMOS devices at lower frequencies.

Another way to reduce power dissipation is to decrease the supply voltage. But this has adverse effects because it increases propagation time and decreases noise immunity. Although CMOS devices can work over a range of 3 to 15 V, the best compromise for speed, noise immunity, and overall performance is a supply voltage from 9 to 12 V. From now on, we assume a supply voltage of 10 V, unless otherwise specified.

Incidentally, notice the use of V_{CC} rather than V_{DD} for the supply voltage. This is a carryover from TTL circuits. You will find V_{CC} on the data sheets for 74C00 devices.

54C00 Series

Any device in the 74C00 series works over a temperature range of −40 to +85°C. This is adequate for most commercial applications. The 54C00 series (for military applications) works over a temperature range of −55 to +125°C. Although 54C00 devices can be substituted for 74C00 devices, they are rarely used commercially because of their much higher cost.

74HC00 Devices

The main disadvantage of CMOS devices is their relatively long propagation delay times. This places a limit on the maximum operating frequency of system. The 74HC00 series is a CMOS series of devices that are pin-for-pin and function-for-function compatible with TTL devices. These devices have the advantage of higher speed (less propagation delay time).

74HCT00 Devices

These are also high-speed CMOS circuits designed to be directly compatible with TTL devices. That is, they can be connected *directly* to any TTL circuit. Interfacing TTL and CMOS devices is discussed in Secs. 14.11 and 14.12.

CD4000 Series

RCA was the first to introduce CMOS devices. The original devices were numbered from CD4000 upward. This 4000 series was soon replaced by the 4000A series (called conventional) and the 4000B series (called the buffered type). The 4000A and B series are widely used; they have many functions not available in the 74C00 series. The main disadvantage of 4000 devices is their lack of pin-for-pin and function-for-function compatibility with TTL.

SELF-TEST

19. What kind of transistors are shown in Fig. 14.33?
20. Why is the NOR gate in Fig. 14.34a CMOS device?

14.10 CMOS CHARACTERISTICS

74C00 series devices are guaranteed to work reliably over a temperature range of −40 to +85°C and over a supply range of 3 to 15 V. In the discussion that follows, *worst case* means the parameters are measured under the worst conditions of temperature and voltage.

Floating Inputs

When a TTL input is floating, it is equivalent to a high input. You can use a floating TTL input to simulate a high input; but as already pointed out, it is better to connect unused TTL inputs to the supply voltage. This prevents the floating leads from picking up stray noise in the environment.

If you try to float a CMOS input, however, not only do you set up a possible noise problem, but, much worse, you produce excessive power dissipation. Because of the insulated gates, a floating input allows the gate voltage to drift into the linear region. When this happens, excessive current can flow through push-pull stages.

The absolute rule with CMOS devices, therefore, is to *connect all input pins*. Most of or all the inputs are normally connected to signal lines. If you happen to have an input that is unused, connect it to ground or the supply voltage, whichever prevents a stuck output state. For instance, with a positive NOR gate you should ground an unused input. Why? Because returning the unused NOR input to the supply voltage forces the output into a stuck low state. On the other hand, grounding an unused NOR input allows the other inputs to control the output.

With a positive NAND gate, you should connect an unused input to the supply voltage. If you try grounding an unused NAND input, you disable the gate because its output will stick in the high state. Therefore, the best thing to do with an unused NAND input is to tie it to the supply voltage. A direct connection is all right: CMOS inputs can withstand the full supply voltage.

Easily Damaged

Because of the thin layer of silicon dioxide between the gate and the substrate, CMOS devices have a very high input resistance, approximately infinite. The insulating layer is kept as thin as possible to give the gate more control over the drain current. Because this layer is so thin, it is easily destroyed by excessive gate voltage.

Aside from directly applying an excessive gate voltage, you can destroy the thin insulating layer in more subtle ways. If you remove or insert a CMOS device into circuit while the power is on, transient voltages caused by inductive kickback and other effects may exceed the gate voltage rating. Even picking up a CMOS IC may deposit enough charge to exceed its gate voltage rating.

One way to protect against overvoltages is to include zener diodes across the input. By setting the zener voltage below the breakdown voltage of the insulating layer, manufacturers can prevent the gate voltage from becoming destructively high. Most CMOS ICs include this form of zener protection.

Figure 14.36 shows a typical *transfer characteristic* (input-output graph) of a CMOS inverter. When the input voltage is in the low state, the output voltage is in the high state. As the input voltage increases, the output remains in the high state until a threshold is reached. Somewhere near an input voltage of $V_{CC}/2$, the output will switch to the low state. Then any input voltage greater than $V_{CC}/2$ holds the output in the low state.

This transfer characteristic is an improvement over TTL. Why? Because the indeterminate region is much smaller. As you can see, the input voltage has to be nearly equal to $V_{CC}/2$ before the CMOS output switches states. This implies that the noise immunity of CMOS devices ideally approaches $V_{CC}/2$. Typically, noise immunity is 45 percent of V_{CC}.

Also notice how much better defined the low and high output states are. When CMOS loads are used, the CMOS source and sink transistors have almost no voltage drop because there is almost no input

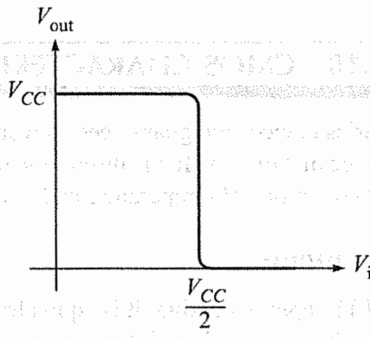


Fig. 14.36 Typical transfer characteristic of a CMOS gate

current to a CMOS load. Therefore, the static currents are extremely small. For this reason, the high output voltage is approximately equal to V_{CC}, and the low output voltage is approximately at ground. Stated another way, the logic swing between the low and high output states approximately equals the supply voltage, a considerable advantage for CMOS over TTL.

Compatibility

CMOS devices are compatible with one another because the output of any CMOS device can be used as the input to another CMOS device, as shown in Fig. 14.37a. For instance, Fig. 14.37b shows the output stage of a CMOS driver connected to the input stage of a CMOS load. The supply voltage is +10 V. Ideally, the input switching level is +5 V. Since the CMOS driver has a low output, the CMOS load has a high input.

Similarly, Fig. 14.37c shows a high CMOS driver output. This is more than enough voltage to drive the CMOS load with a high-state input. In fact, the noise immunity typically approaches 4.5 V (from 45 percent of V_{CC}). Any noise picked up on the connecting line between devices would need a peak value of more than 4.5 V to cause unwanted switching action.

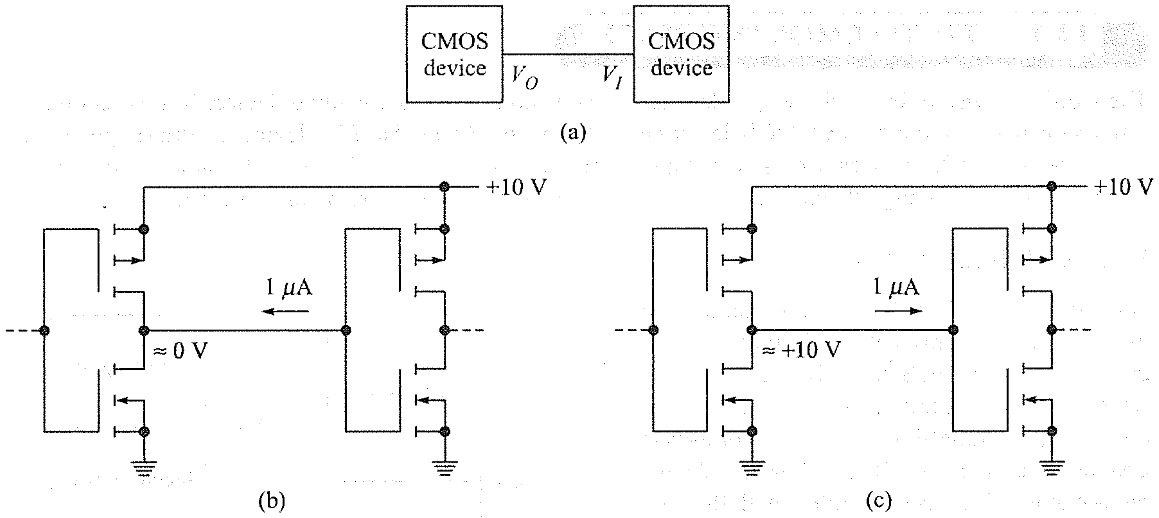


Fig. 14.37 (a) Output of CMOS device can drive input of another CMOS device, (b) Sink current, (c) Source current

Sourcing and Sinking

When a standard CMOS driver output is low (Fig. 14.37b), the input current to the CMOS load is only 1 microampere (μA) (worst case shown on data sheet). The input current is so low because of the insulated gates. This means that the CMOS driver has to sink only 1 μA. Similarly, when the driver output is high (Fig. 14.37c), the CMOS driver is sourcing 1 μ.A.

In symbols, here are the worst-case input currents for CMOS devices:

$$I_{IL,\max} = -1\ \mu\text{A} \quad I_{IH,\max} = 1\ \mu\text{A}$$

We use these values to calculate the fanout.

Fanout

The fanout of CMOS devices depends on the kind of load being driven. In Sec. 14.11, we discuss CMOS devices driving TTL devices. Now we want to concentrate on CMOS driving CMOS. Data sheets for 74C00 series devices give the following output currents for CMOS driving CMOS:

$$I_{OL,\max} = 10\ \mu\text{A} \quad I_{OH,\max} = -10\ \mu\text{A}$$

Since the worst-case input current of a CMOS device is only 1 μA, a CMOS device can drive up to 10 CMOS loads. Therefore, you can use a fanout of 10 for CMOS-to-CMOS connections. This value is reliable under all operating conditions.

SELF-TEST

21. Why must care be taken when using CMOS devices?
22. What is the transfer characteristic of a CMOS inverter?

14.11 TTL-TO-CMOS INTERFACE

The word *interface* refers to the way a driving device is connected to a loading device. In this section, we discuss methods for interfacing CMOS devices to TTL devices. Recall that TTL devices need a supply voltage of 5 V, while CMOS devices can use any supply voltage from 3 to 15 V. Because the supply requirements differ, several interfacing schemes may be used. Here are a few of the more popular methods.

Supply Voltage at 5 V

One approach to TTL/CMOS interfacing is to use +5 V as the supply voltage for both the TTL driver and the CMOS load. In this case, the worst-case TTL output voltages (Fig. 14.38a) are almost compatible with the worst-case CMOS input voltages (Fig. 14.38b). *Almost*, but not quite. There is no problem with the TTL low-state window (0 to 0.4 V) because it fits inside the CMOS low-state window (0 to 1.5 V). This means the CMOS load always interprets the TTL low-state drive as a low.

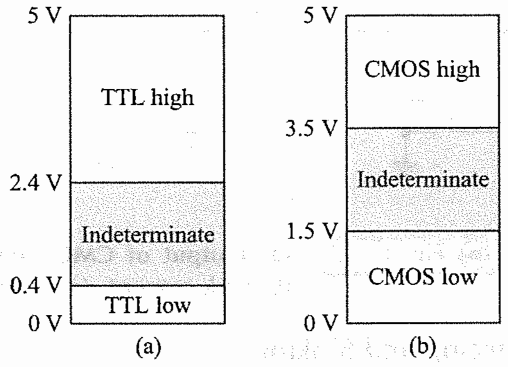


Fig. 14.38 (a) TTL output profile, (b) CMOS input profile

The problem is in the TTL high state, which can be as low as 2.4 V (see Fig. 14.38a). If you try using a TTL high-state output as the input to a CMOS device, you get indeterminate action. The CMOS device needs at least 3.5 V for a high-state input (Fig. 14.38b). Because of this, you cannot get reliable operation by connecting a TTL output directly to a CMOS input. You have to do something extra to make the two different devices compatible.

What do you do? The standard solution is to use a pull-up resistor between the TTL driver and the CMOS load, as shown in Fig. 14.39. What effect does the pull-up resistor have? It has almost no effect on the low state, but it does raise the high state to approximately +5 V. For instance, when the TTL output is low, the lower end of the 3.3 kΩ is grounded (approximately). Therefore, the TTL driver sinks a current of roughly

$$I = \frac{5\text{ V}}{3.3\text{ k}\Omega} = 1.52\text{ mA}$$

When the TTL output is in the high state, however, the output voltage is pulled up to +5 V. Here is how it happens. As before, the upper totem-pole transistor actively pulls the output up to +2.4 V (worst case). Because of the pull-up resistor, however, the output rises above +2.4 V, which forces the upper totem-pole transistor into cutoff. The pull-up action is now passive because the supply voltage is pulling the output voltage up to +5 V through the pull-up resistor.

The gate capacitance of the CMOS load has to be charged through the pull-up resistor. This slows down the switching action. If speed is important, you can decrease the pull-up resistance. The minimum resistance is determined by the maximum sink current of the TTL device: $I_{OL,\max} = 16$ mA. In the worst case the supply voltage may be as high as 5.25 V, so the minimum resistance is

$$R_{\min} = \frac{5.25\text{ V}}{16\text{ mA}} = 328\ \Omega$$

The nearest standard value is 330 Ω, which you should consider the absolute minimum value for the pull-up resistor. And you would use this only if switching speed were critical. In many applications, a pull-up resistance of 3.3 kΩ is fine.

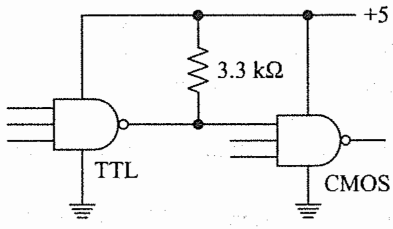


Fig. 14.39 TTL driver and CMOS load

Incidentally, the other inputs of the TTL driver and CMOS load (Fig. 14.39) are connected to signal lines not shown. Also, the use of 3-input gates is arbitrary. You can interface gates with any number of inputs. If more than one TTL chip is being interfaced to the CMOS load, connect each TTL driver to a separate pull-up resistor and CMOS input.

Different Supply Voltages

CMOS performance deteriorates at lower voltages because the propagation delay time increases and the noise immunity decreases. Therefore, it is better to run CMOS devices with a supply voltage between 9 and 12 V. One way to use a higher supply voltage is with an open-collector TTL driver (Fig. 14.40). Recall that the output stage of an open collector TTL device consists only of a sink transistor with a floating collector. In Fig. 14.40. this open collector is connected to a supply voltage of +12 V through a pull-up resistance of 6.8 kΩ. Likewise, the CMOS device now has a supply voltage of +12 V.

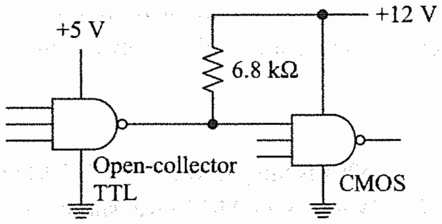


Fig. 14.40 Open-collector TTL driver allows higher CMOS supply voltage

When the TTL output is low, we can visualize a ground on the lower end of the pull-up resistor. Therefore, the TTL device has to sink approximately

$$I_{sink} = \frac{12 \text{ V}}{6.8 \text{ k}\Omega} = 1.76 \text{ mA}$$

When the TTL output is high, the open-collector output rises passively to +12 V. In either case, the TTL outputs are compatible with the CMOS input states.

The passive pull-up in Fig. 14.40 produces slower switching action than before. For instance, with a gate input capacitance of 10 pF, the pull-up time constant is

$$RC = (6.8 \text{ k}\Omega)(10 \text{ pF}) = 68 \text{ ns}$$

If this is a problem, reduce the pull-up resistance to its minimum allowable value of

$$R_{min} = \frac{12 \text{ V}}{16 \text{ mA}} = 750 \ \Omega$$

Then the pull-up time constant decreases to

$$RC = (750 \ \Omega)\ (10 \text{ pF}) = 7.5 \text{ ns}$$

CMOS Level Shifter

Figure 14.41 shows a 40109, called a *level shifter*. The input stage of the chip uses a supply voltage of +5 V, while the output stage uses +12 V. In other words, the input stage interfaces with TTL, and the output stage interfaces with CMOS.

In Fig. 14.41, a standard TTL device drives the level shifter. This produces active TTL pull-up to at least +2.4 V. Beyond this level, the pull-up resistor takes over and raises the voltage to +5 V, which ensures a valid high-state input to the level shifter. The output side of the level shifter connects to +12V (this can be changed to any voltage from 3 to 15 V). Since the CMOS load runs off of +12 V, it has better propagation delay time and noise immunity.

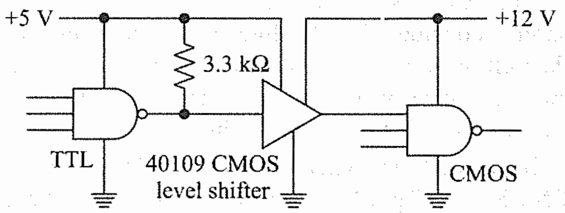


Fig. 14.41 **CMOS level shifter allows the use of 5-V and 12-V supplies**

In summary, TTL has to run off of +5 V, but CMOS does better with a supply voltage of +12 V. This is the reason for using a level shifter between the TTL driver and the CMOS load.

SELF-TEST

23. What is the purpose of the 3.3-kΩ resistor in Fig. 14.41?
24. Where is a CMOS level shifter used?

14.12 CMOS-TO-TTL INTERFACE

In this section, we discuss methods for interfacing CMOS devices to TTL devices, Again, the problem is to shift voltage levels until the CMOS output states fall inside the TTL input windows. Specifically, we have to make sure that the CMOS low-state output is always less than 0.8 V, the maximum allowable TTL low-state input voltage. Also, the CMOS high-state output must always be greater than 2 V, the minimum allowable TTL high-state input voltage.

Supply Voltage at 5 V

One approach is to use +5 V as the supply voltage for the driver and the load, as shown in Fig. 14.42. A direct interface like this forces you to use a low-power Schottky TTL load (or two low-power TTL loads). Why? Because a low-power Schottky device has these worst-case input currents:

$$I_{IL,\max} = -360\ \mu\text{A} \quad I_{IH,\max} = 20\ \mu\text{A}$$

Data sheets for 74C00 devices list these worst-case output currents for CMOS driving TTL:

$$I_{OL,\max} = 360\ \mu\text{A} \quad I_{OH,\max} = -360\ \mu\text{A}$$

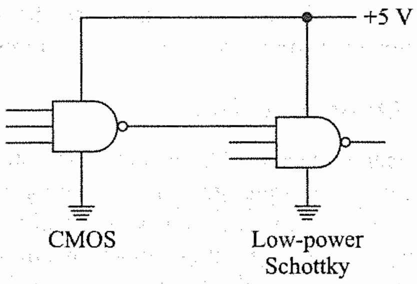


Fig. 14.42 CMOS driver and low-power Schottky TTL load

This tells us that a CMOS drive can sink 360 μA in the low state, exactly the input current for a low-power Schottky TTL devices. On the other hand, the CMOS driver can source 360 μA, which is more than enough to handle the high-state input current (only 20 μA). So the sink current limits the CMOS/74LS fanout to 1.

CMOS can also drive low-power TTL devices. The limiting factor again is the sink current. Low-power TTL has a worst-case low-state input current of 180 μA. Since a CMOS driver can sink 360 μA, it can drive two low-power TTL devices. Briefly stated, the CMOS/74L fanout is 2.

CMOS cannot drive standard TTL directly because the latter requires a low-state input current of −1.6 mA, for too much current for a CMOS device to sink without entering the TTL indeterminate region. The problem is that the sink transistor of a CMOS device is equivalent to a resistance of approximately 1.11 kΩ (worst case). The CMOS output voltage equals the product of 1.6 mA and 1.11 kΩ, which is 1.78 V. This is too large to be low-state TTL input.

Using a CMOS Buffer

Figure 14.43 shows how to get around the fanout limitation just discussed. The CMOS driver now connects directly to a CMOS buffer, a chip with larger output currents. For instance, a 74C902 is a hex buffer, or six CMOS buffers in a single package. Each buffer has these worst-case output currents:

$$I_{OL,\max} = 3.6\ \text{mA} \quad I_{OH,\max} = 800\ \mu\text{A}$$

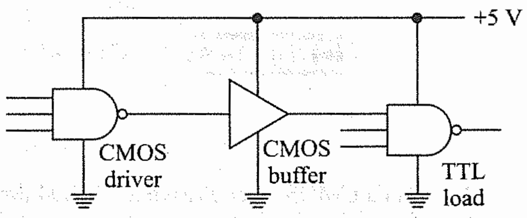


Fig. 14.43 CMOS buffer can drive standard TTL load

Since a standard TTL load has a low-state input current of 1.6 mA and a high-state input current of 40 μA, a 74C902 can drive two standard TTL loads. If you use one-sixth of a 74C902 in Fig. 14.43, the CMOS/TTL fanout is 2. Other available buffers are the CD4049A (inverting), CD4050A (noninverting), 74C901 (inverting), etc.

Different Supply Voltages

CMOS buffers like the 74C902 can use a supply voltage of 3 to 15V and an input voltage of −0.3 to 15 V. The input voltage can be greater than the supply voltage without damaging the device. For instance, you can use a high-state input of +12 V even though the supply voltage is only +5 V.

Figure 14.44 shows how to use the previous idea to our advantage. Here, the supply pin of the CMOS driver is connected to +12 V. On the other hand, the supply pin of the CMOS buffer is connected to +5 V to produce the TTL interface. Therefore, the input to the CMOS buffer will be as much as +12 V, even if its supply voltage is only +5 V. The fanout of this interface is still two standard TTL loads.

Open-Drain Interface

Recall open-collector TTL devices. The output stage consists of a sink transistor with a floating collector. Similar devices exist in the CMOS family. Known as *open-drain devices*, these have an output stage consisting only of a sink MOSFET. An example is the 74C906, a hex open-drain buffer.

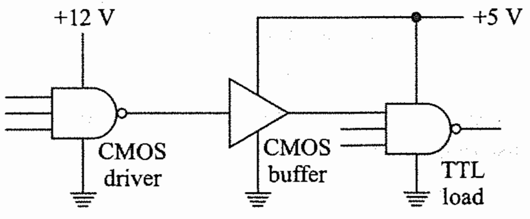


Fig. 14.44 **CMOS driver runs better with 12-V supply**

Figure 14.45 shows how an open-drain CMOS buffer can be used as an interface between a CMOS driver and a TTL load. The supply voltage for most of the buffer is +12 V. The open drain, however, is connected to a supply voltage of +5 V through a pull-up resistance of 3.3 kΩ. This has the advantage that both the CMOS driver and the CMOS buffer run off of +12 V, except for the open-drain output which provides the TTL interface.

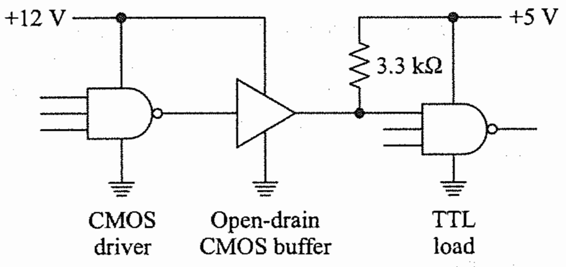


Fig. 14.45 **Open-drain CMOS buffer increases sink current**

SELF-TEST

25. Can a CMOS circuit drive a 74LS04 directly? What about a 7404?
26. CMOS output voltage levels are well within the profile of TTL input voltage levels. Why can't the CMOS drive the TTL directly?

14.13 CURRENT TRACERS

Figure 14.46a shows a solder bridge shorting a node to ground. When you trigger the logic pulser, the logic probe remains dark because the node is stuck in the low state. A logic pulser and logic probe will help you locate stuck nodes, but they cannot tell you the exact location of the short.

Figure 14.46b shows a *current tracer*, a troubleshooting tool than can detect current in a wire or circuit-board trace. Although it touches the wire, the current tracer does not make electric contract. Inside its

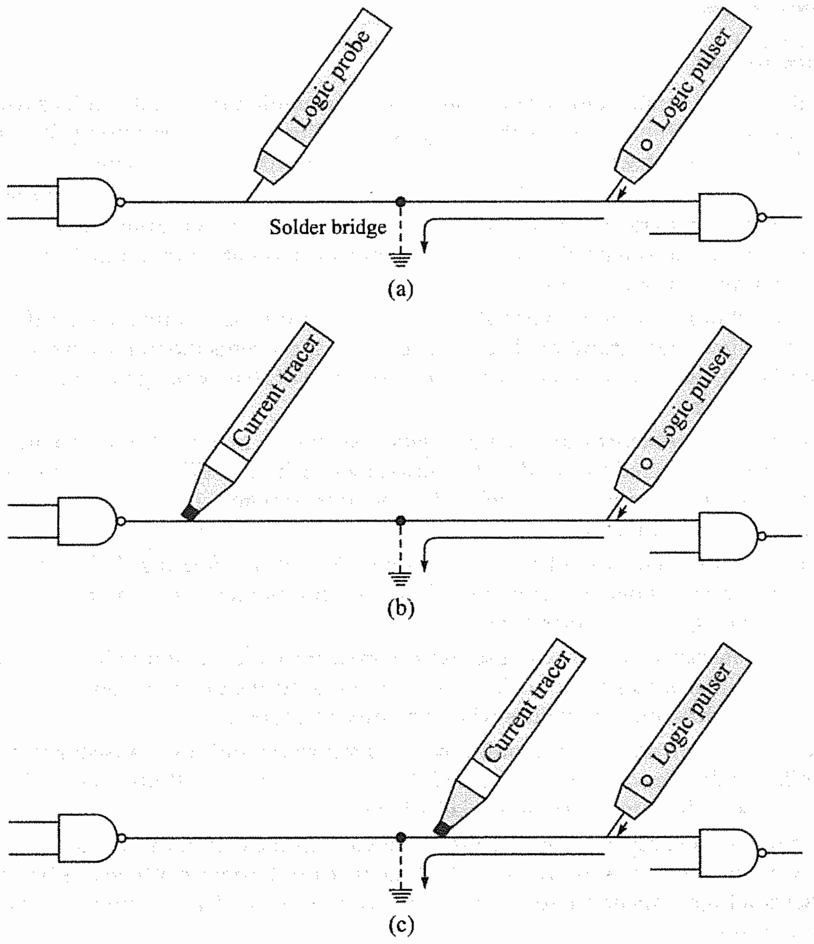


Fig. 14.46 **(a) Solder bridge shorts node to ground, (b) Current tracer will not detect any current, (c) Current tracer will detect current**

blunt insulated tip is a small pick-up coil than can detect the magnetic field around a wire carrying current. Therefore, if there is any current through the wire, the current tracer lights up.

In Fig. 14.46b, each time you trigger the logic pulser, a conventional current flows through its tip to ground along the path shown. The current tracer will not detect this current because it is touching another part of the wire.

If you move the current tracer between the ground and the logic pulser as shown in Fig. 14.46c, the current tracer will light up. The critical position for the current tracer is directly over the short. Move left, and the current tracer goes out. Move right, and it turns on. When this happens, you know the current tracer is directly over the trouble. During troubleshooting, a visual check at this critical location usually reveals the nature of the trouble (a solder bridge, in this discussion).

That's the basic idea behind a current tracer. You use the logic pulser and logic probe to find stuck nodes. Then you use the current tracer to locate the exact position along a trace where the short is.

SUMMARY

A chip is a small piece of semiconductor material with microminiature circuits on its surface. Small-scale integration (SSI) refers to chips with less than 12 gates. Medium-scale integration (MSI) means 12 to 100 gates per chip. Large-scale integration (LSI) refers to more than 100 gates on a chip.

The 7400 series is a line of standard TTL chips. This bipolar family contains a variety of compatible SSI and MSI devices. One way to recognize TTL design is the multiple-emitter input transistors and the totem-pole output transistors. The standard TTL chip has a power dissipation of about 10 mW per gate and a propagation delay time of around 10 ns.

By including a Schottky diode in parallel with the collector-base terminals, manufacturers produce Schottky TTL. This eliminates saturation delay time because it prevents the transistors from saturating on. Numbered from 74S00, these devices have a power dissipation of 20 mW per gate and a propagation delay time of approximately 3 ns.

By increasing internal resistances and including Schottky diodes, manufacturers can produce low-power Schottky TTL devices (numbered from 74LS00). A low-power Schottky TTL gate has a power dissipation of around 2 mW per gate and a propagation delay time of approximately 10 ns. Low-power Schottky TTL is the most widely used of the TTL types.

A floating TTL input is equivalent to a high input. Do not use floating TTL inputs when you are operating in an electrically noisy environment. Floating inputs may pick up enough noise voltage to produce unwanted changes in the output states.

A standard TTL gate can sink 16 mA and source 400 mA. Since the maximum input currents are 1.6 mA (low state) and 40 μA (high state), standard TTL has a fanout of 10, meaning that one standard TTL gate can drive 10 others. Fanout has different values when you mix TTL types.

Open-collector devices have only the pull-down transistor; the pull-up transistor is omitted. Because of this, open-collector devices can be wire—ORed through a common pull-up resistor. This connection is inherently slow because the time constant is relatively long.

Three-state devices have replaced open-collector devices in most applications because they are much faster. These newer devices have a control input that can turn off the device. When this happens, the output floats and presents a high impedance to whatever it is connected to. Three-state devices are widely used for connecting to buses.

A CMOS inverter uses complementary MOSFETs in a push-pull arrangement. The key advantage of CMOS devices is the low power dissipation. The main disadvantage is the slow switching speed.

The 74C00 series is a line of CMOS circuits that are pin-for-pin and function-for-function compatible with TTL devices. The static power dissipation of 74C00 devices is approximately 10 nanowatts (nW) per gate. Active power dissipation is higher because of the current spikes during transitions. Lower supply voltages increase the propagation delay time and noise immunity. Higher supply voltages increase the power dissipation. The best compromise is a supply voltage from 9 to 12 V. The 74HC00 series is a line of high-speed CMOS devices. The CD4000 series is another line of CMOS devices with many functions not available in the 74C00 series.

CMOS devices are guaranteed to work reliably over a temperature range of –40 to +85°C and a supply range of 3 to 15 V. Unused inputs should be returned to the supply voltage or to ground, depending on which connection prevents a stuck output. A floating CMOS input is poor design because it produces large power dissipation. CMOS devices have a fanout of 10 when driving other CMOS devices. By using level shifting, CMOS devices can be interfaced with TTL devices.

The 74HCT00 series is completely TTL-compatible, and special interfacing is not required.

GLOSSARY

- ***active load*** A transistor that acts as a load for another transistor.
- ***active power dissipation*** The power dissipation of a device under switching conditions. It differs from static power dissipation because of the large current spikes during output transitions.
- ***bipolar*** Having two types of charge carriers: free electrons and holes.
- ***bus*** A group of wires that transmits binary data. The data bus of a first-generation microcomputer has eight wires, each carrying 1 bit. This means that the data bus can transmit 1 byte at a time. Typically, the byte represents an instruction or data word that is moved from one register to another.
- ***chip*** A small piece of semiconductor material. Sometimes chip refers to an IC device including its pins.
- ***CMOS inverter*** A push-pull connection of *p*- and *n*-channel MOSFETs.
- ***compatibility*** Ability of the output of one device to drive the input of another device.
- ***interface*** The way a driving device is connected to a loading device. All the circuitry between the output of a device and the input of another device.
- ***fanout*** The maximum number of TTL loads that a TTL device can reliably drive.
- ***low-power Schottky TTL*** A modification of standard TTL in which larger resistances and Schottky diodes are used. The larger resistances decrease the power dissipation, and the Schottky diodes increase the speed.
- ***noise immunity*** The amount of noise voltage that causes unreliable operation. With TTL it is 0.4 V. As long as the noise voltages induced on connecting lines are less than 0.4 V, the TTL devices will work reliably.
- ***saturation delay time*** The time delay encountered when a transistor tries to come out of the saturation region. When the base drive switches from high to low, a transistor cannot instantaneously come out of hard saturation; extra carriers must first flow out of the base region.
- ***Schmitt trigger*** A digital circuit that produces a rectangular output. The input waveform may be sinusoidal, triangular, distorted, and so on. The output is always rectangular.
- ***sink*** A place where something is absorbed. When saturated, the lower transistor in a totem-pole output acts as a current sink because conventional charges flow through the transistor to ground.
- ***source*** The upper transistor of a totem-pole output acts as a source because conventional flow is out of the emitter into the load.
- ***standard TTL*** The basic TTL design. It has a power of dissipation of 10 mW per gate and a propagation delay time of 10 ns.
- ***three-state TTL*** A modified TTL design that allows us to connect outputs directly. Earlier computers used open-collector devices with their buses, but the passive pull-up severely limited the operating speed. By replacing open-collector devices with three-state devices, we can significantly reduce the switching time needed to change from the low state and the output state. The result is faster data changes on the bus, which is equivalent to speeding up the operation of a computer.

PROBLEMS

Section 14.1

14.1 For each assigned circuit in Fig. 14.47, determine the indicated current I and/or voltage V.

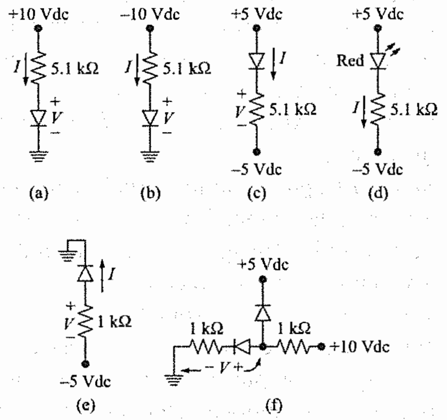


Fig. 14.47

14.2 From memory, draw the symbols for: diode, LED, *npn* BJT, *pnp* BJT.

14.3 Make a truth table for the circuit in Fig. 14.5a.

14.4 From memory, draw the simplified symbols for an *n*-channel MOSFET and a *p*-channel MOSFET.

14.5 For each assigned circuit in Fig. 14.48, determine the indicated current I and/or voltage V.

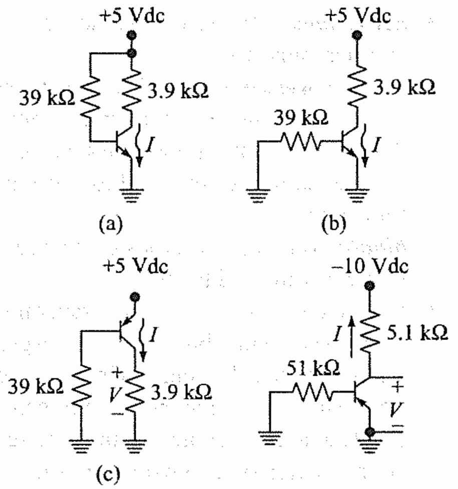


Fig. 14.48

14.6 Draw the circuit for a CMOS inverter.

14.7 For each assigned circuit in Fig. 14.49 on the next page, determine the indicated current I and/or voltage V.

Section 14.2

14.8 Figure 14.10 shows typical resistance values at room temperature. Here A is high, and B is grounded. Allowing 0.7 V for the base-emitter

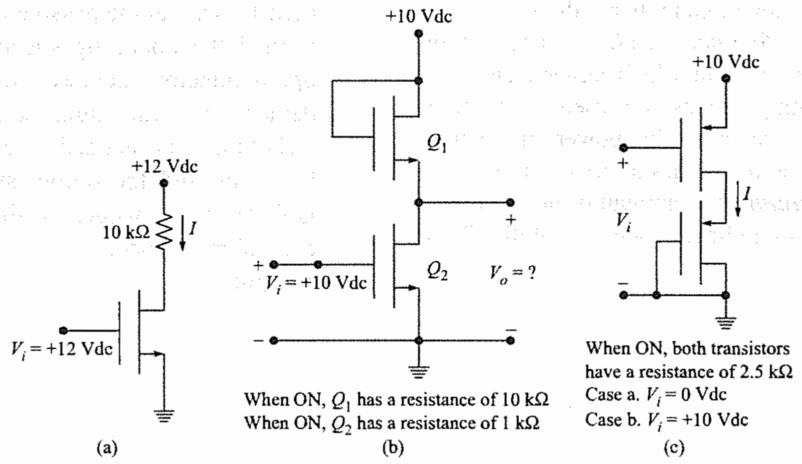


Fig. 14.49

voltage, how much current is there through the 4kΩ?

14.9 Suppose you need a TTL device with a power dissipation of less than 5 mW per gate and a delay time of less than 20 ns. What TTL type would you choose?

14.10 Use the values of Table 14.3 to calculate the total propagation delay time of three cascaded gates for each of the following TTL types:

a. Low-power
b. Low-power Schottky
c. Standard
d. High-speed
e. Schottky

Section 14.3

14.11 What is the fanout of a 74S00 device when it drives low-power TTL loads?

14.12 What is the fanout of a low-power Schottky device driving standard TTL devices?

14.13 What is the fanout of a standard TTL device driving a 74LS device?

14.14 The output of a 74LS04 is connected to the inputs of two 7404s, one 7400, and three 7410s. It seems to malfunction occasionally. What might be the problem?

14.15 What would be a simple "fix" for the circuit in Prob. 14.14?

14.16 A zero-rise-time pulse is applied to the input of a 74LS04. Its output drives a 74LS10. What is the delay time from the rising edge of the input pulse to the rising edge of the 74LS10 output?

Section 14.4

14.17 What is the fanout of a 7437 buffer when it drives standard TTL loads?

14.18 The input to a 7414 hex Schmitt trigger is a 2-V-peak sinewave. Sketch both the input and output voltages.

14.19 What is the output in Fig. 14.20 for these inputs?

a. $ABCD = 0000$
b. $ABCD = 0101$
c. $ABCD = 1100$
d. $ABCD = 1111$

14.20 Is the output Y of Fig. 14.50 low or high for these conditions?

a. Both switches open, A is low
b. Both switches closed, A is high
c. Left switch open, right switch closed, A is low
d. Left switch closed, right switch open, A is high

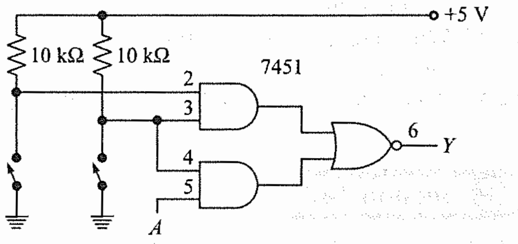


Fig 14.50

14.21 What is the value of Y in Fig. 14.51 for each of these?

a. $ABCD = 0000$
b. $ABCD = 0101$
c. $ABCD = 1000$
d. $ABCD = 1111$

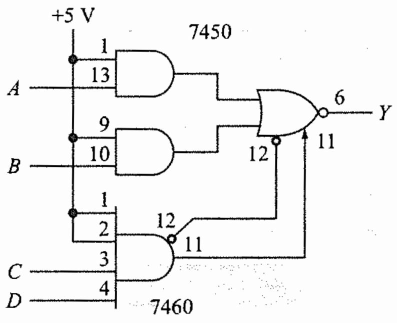


Fig. 14.51

Section 14.5

14.22 In Fig. 14.22a, $I_{OL,\max} = 16$ mA. If three open-collector gates like these are wire-ORed together as shown in Fig. 14.22c, what is the minimum value of pull-up resistance needed to avoid destroying any device?

14.23 Suppose the total output capacitance is 20 pF in Fig. 14.22c. If the pull-up resistance equals 3.6 kΩ, what does the charging time constant equal?

Section 14.6

14.24 You want the contents of register *B* to appear on the bus of Fig. 14.28. What are the necessary disable values?

14.25 What are the three output conditions of a three-state gate?

14.26 Draw the logic symbol for a three-state inverter.

Section 14.7

14.27 In Fig. 14.52, what does output *Y* equal when each switch is open? When either switch is closed?

14.28 What is the current drain through the pull-up resistors when both switches are closed in Fig. 14.52? What is the time constant for each input when the switches are open?

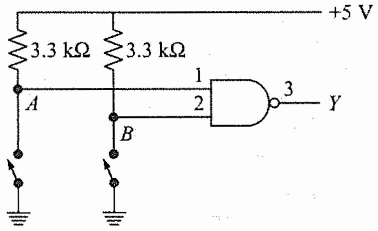


Fig. 14.52

14.29 In Fig. 14.53, what does the output *Y* equal when either switch is open? When both are closed? This is not a preferred method of driving TTL loads. Try to figure out two reasons why this circuit is not as good as the circuit shown in Fig. 14.52.

Section 14.8

14.30 In Fig. 14.54a, the TTL output voltage is 0.4 V, and the LED voltage is 2 V. What is the sink current when the LED is lighted?

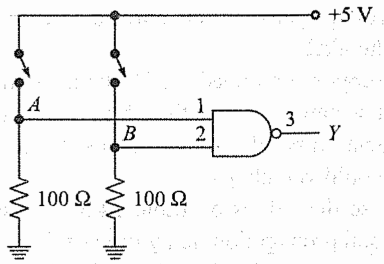


Fig. 14.53

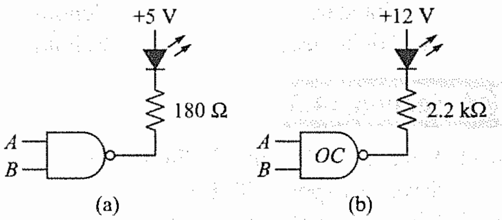


Fig. 14.54

14.31 What is the LED current in Fig. 14.54b if the LED voltage drop is 2 V and the TTL output is high? If the TTL output is 0.4 V, what is the LED current?

14.32 When switch *B* of Fig. 14.55 is closed, is the LED on or off? For this condition, what is the current in the LED?

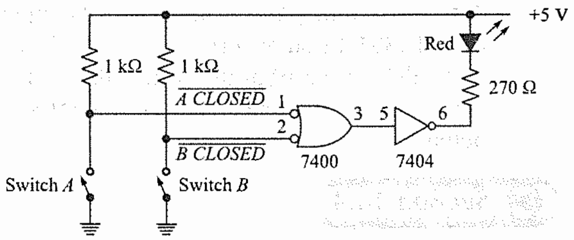


Fig. 14.55

Section 14.9

14.33 Three CMOS devices are cascaded. If each has a propagation delay time of 100 ns, what is the total propagation delay time?

14.34 A 74C00 has a load capacitance of 50 pF. If the supply voltage is 10 V, what is the active power dissipation per gate at each of the following frequencies?

a. kHz b. 10 kHz
c. 100 kHz

14.35 Explain the difference between 74C00, 74HC00, and 74HCT00 devices.

14.36 What are CD4000 series ICs?

Section 14.10

14.37 Figure 14.56a shows how to drive a CMOS device from a switch. As you see, the input does not float in either state. Is the output low or high when the switch is open? Is it low or high when the switch is closed?

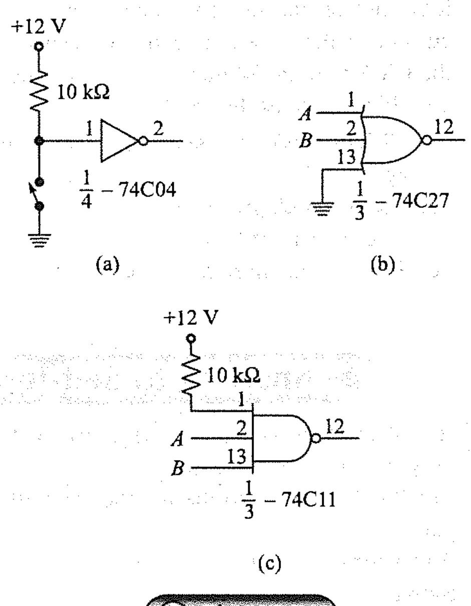


Fig. 14.56

14.38 Pin 13 is an unused input in Fig. 14.56b. As you see, it is grounded. Is the output low or high for each of these conditions?

a. *A* and *B* both low
b. *A* low and *B* high
c. *A* high and *B* low
d. *A* and *B* both high

14.39 If pin 13 is returned to the supply voltage instead of grounded in Fig. 14.56b, the circuit is useless as a NOR gate. Why?

14.40 Pin 1 is an unused input in Fig. 14.56c. As you see, it is returned to the supply voltage through a pull-up resistor. Is the output low or high for each of these conditions?

a. *A* and *B* both low
b. *A* low and *B* high
c. *A* high and *B* low
d. *A* and *B* both high

14.41 If pin 1 is grounded instead of returned to the supply voltage in Fig. 14.56c, the circuit is useless as a NAND gate. Why?

Section 14.11

14.42 If the CMOS input is low in Fig. 14.57, what is the sink current in the TTL driver? If the CMOS input is high, how much voltage drop is there across the 1.5 kΩ if the gate current is 1 μA (worst case)?

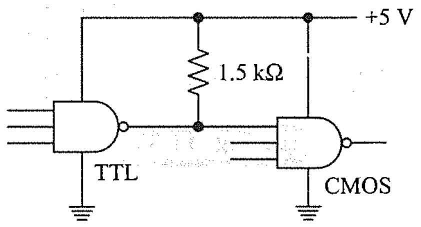


Fig. 14.57

14.43 Ideally, how much current does the open-collector driver of Fig. 14.58 have to sink when its output is low?

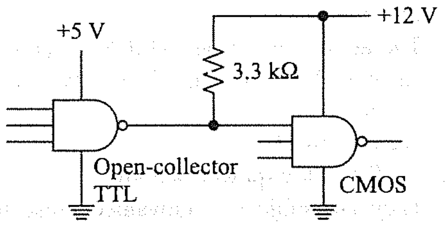


Fig. 14.58

14.44 What is the smallest acceptable value for the 3.3-kΩ resistor in Fig. 14.39 if the TTL device is a 7410 (look at maximum sink current)? What if the 7410 were replaced with a 74LS10?

14.45 The TTL in Fig. 14.40 is a 74LS10. Could a second CMOS circuit be added at the output of the 74LS10 without violating any loading rules? How many could be added?

Section 14.12

14.46 Ideally, what is the sink current *I* in Fig. 14.59? If the TTL load has a high-state input current of 40 μA, what is the voltage drop across the 2.2 kΩ?

14.47 If the input capacitance of the TTL load is 10 pF, what does the pull-up time constant equal in Fig. 14.59?

14.48 What is the maximum sink current for the 74C906 in Fig. 14.59?

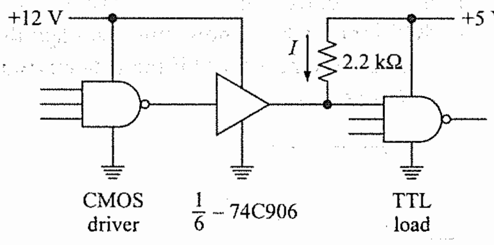


Fig. 14.59

Section 14.13

14.49 In Fig. 14.57, a logic probe indicates that the lower end of the pull-up resistor is stuck in the low state. Using a logic pulser and current tracer, you detect a current in the wire between this resistor and the TTL output. Which of the following is a possible trouble?

a. 1.5 kΩ shorted
b. 1.5 kΩ open
c. Open trace between the TTL output and the resistor
d. TTL sink transistor shorted

14.50 The lower end of the pull-up resistor (Fig. 14.57) is stuck in the high state. With a logic pulser and current tracer, you detect a current in the wire between this resistor and the CMOS input. Which of the following is a possible source of the trouble?

a. CMOS input trace shorted to supply voltage
b. CMOS input grounded
c. TTL output trace open
d. TTL output shorted to ground

Answers to Self-tests

1. The voltage across a forward-biased LED is larger.
2. *npn* and *pnp*. A *p*-channel MOSFET is the complement of an *n*-channel MOSFET.
3. Positive.
4. The active load in an NMOS IC is an *n*-channel MOSFET used in place of a resistor.
5. See Fig. 14.11.
6. 74LS00—low-power Schottky.
7. They are subject to unwanted noise that may switch the circuit.
8. It specifies acceptable high and low input voltage levels.
9. From Table 14.3, it is 10 ns.
10. It can sink or source more current than a standard gate.
11. It will produce an output waveform with very fast rise and fall times.
12. The "width" refers to the number of AND gates.
13. A resistor to +5 Vdc
14. Slower
15. They are used to facilitate connecting two or more gate outputs in parallel.
16. The factors are dc power supply current and switching time.
17. Its purpose is to protect the base-emitter of the transistor from excessive voltage.
18. No, you must also have a resistor in series with the LED to limit current; otherwise, when the transistor is on, the diode and/or the transistor will burn out.

19. Q_1 and Q_4 are *p*-channel MOSFETs. Q_2 and Q_3 are *n*-channel MOSFETs.
20. The NOR gate is CMOS because it uses both *n*- and *p-channel transistors.*
21. The thin oxide layer connected to the gate is easily damaged by static electricity.
22. The transfer characteristic of a CMOS inverter is a plot of input voltage versus output voltage (Fig. 14.38).
23. Its purpose is to raise the minimum TTL high output level above the lowest allowable CMOS high input level.
24. A level shifter is used between a TTL gate driving a CMOS gate; it is used to make their high and low levels compatible.
25. Yes, no
26. The CMOS has current sink and source limitations.

Applications

15

OBJECTIVES

- Understand the multiplexing techniques used with LED displays
- List and describe the main sections of a frequency counter
- Explain how a time measurement circuit can be designed
- Be familiar with the basic features of the ADC0804 A/D converter
- Be familiar with the basic features of the ADC3511 microprocessor compatible A/D converter
- Discuss how to construct a digital voltmeter using the National Semi-conductor ADD3501 chip

This chapter is intended to tie together many of the fundamental ideas presented previously by considering some of the more common digital circuit design encountered in industry. The multiplexing of digital LED displays is considered first since it requires the use of a number of different TTL circuits studied in detail in prior chapters. Digital instruments that can be used to measure time and frequency are considered next, and the concept of display multiplexing is applied here.

A number of applications using the popular ADC0804 are presented. An intergrating-type converter, the microprocessor-compatible ADC3511, is studied in detail. Then a similar converter, the ADC3501, is used to construct a digital voltmeter. In most of the applications considered, specific TTL part numbers have been specified, but in the interest of clarity, detailed designs including pin numbers have not been provided. However, it is a simple matter to consult the appropriate data sheets for this information.

In some cases, a specific part number has not been assigned; an example of this is the 1-MHz clock: oscillator shown in Fig. 15.14, or a divide-by-10 counter in the same figure. In such cases, it is left to you to select any one of a number of divide-by-10 circuits, or to choose an oscillator circuit such as discussed in a previous chapter, on the basis of availability, cost, ease of use, compatibility with the overall system, and other factors.

15.1 MULTIPLEXING DISPLAYS

The decimal outputs of digital instruments such as digital voltmeters (DVMs) and frequency counters are often displayed using seven-segment indicators. Such indicators are constructed by using a fluorescent bar, a liquid crystal bar, or a LED bar for each segment. LED-type indicators are convenient because they are directly compatible with TTL circuits, do not require the higher voltages used with fluorescents, and are generally brighter than liquid crystals. On the other hand, LEDs do generally require more power than either of the other two types, and *multiplexing* is a technique used to reduce indicator power requirements.

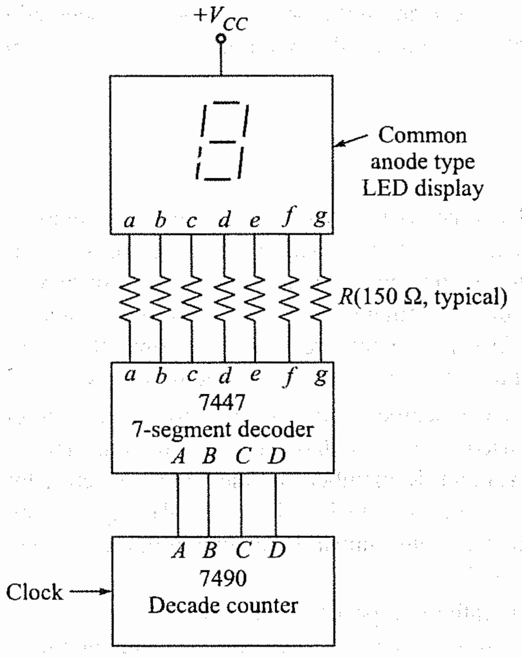
The circuit in Fig. 15.1a is a common-anode LED-type seven-segment indicator used to display a single decimal digit. The 7447 BCD to seven-segment decoder is used to drive the indicator, and the four inputs to the 7447 are the four-flip-flop outputs of the 7490 decade counter. Remember that the 7447 has active low outputs, so the equivalent circuit of an illuminated segment appears as in Fig. 15.lb. A 1-Hz square wave applied at the clock input of the 7490 will cause the counter to count upward, advancing one count each second, and the equivalent decimal number will appear on the display.

A similar single decimal digit display using a common-cathode-type LED indicator is shown in Fig. 15.2a. The seven-segment decoder used here is the 7448; its outputs are active high, and they are intended to drive buffer amplifiers since their output current capabilities are too small to drive LEDs directly. The seven *npn* transistors simply act as switches to connect $+V_{CC}$ to a segment. When an output of the 7448 is high, a transistor is on, and current is supplied to a LED segment. The equivalent circuit for an illuminated segment is shown in Fig. 15.2b. When an output of the 7448 is low, the transistor is off, and there is no segment current and thus no illumination.

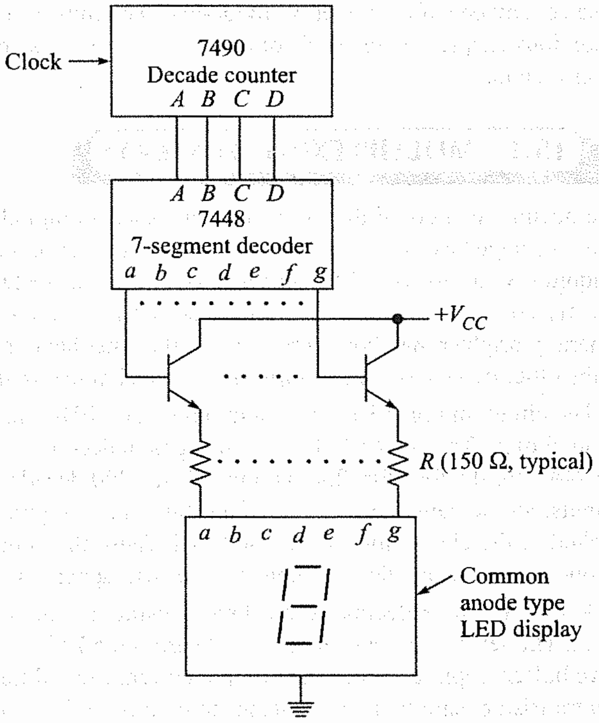
Let's take a look at the power required for the single-digit display in Fig. 15.la. A segment is illuminated whenever an output of the 7447 goes low (essentially to ground). If we assume a 2-Vdc drop across an illuminated segment (LED), a current $I = (5 - 2)/150 = 20$ mA is required to illuminate each segment. The largest current is required when the number 8 is displayed, since this requires all segments to be illuminated. Under this condition, the indicator will require $7 \times 20 = 140$ mA. The 7447 will also require about 64 mA, so a maximum of around 200 mA is required for this single digit display. An analysis of the display circuit in Fig. 15.2 will yield similar results.

A digital instrument that has a four-digit decimal display will require four of the circuits in Fig. 15.1 and thus has a current requirement of $4 \times 200 = 800$ mA. A six-digit instrument would require 1200 mA, or 1.2 A, just for the displays! Clearly these current requirements are much too large for small instruments, but they can be greatly reduced using multiplexing technique.

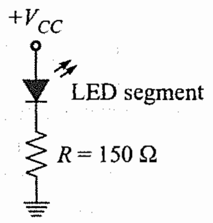
Basically, multiplexing is accomplished by applying current to each display digit in short, repeated pulses rather than continuously. If the pulse repetition rate is sufficiently high, your eye will perceive a steady illumination without any flicker. (For instance, hardly any flicker is noticeable with indicators illuminated using 60 Hz.) The single-digit display in Fig. 15.3a has +5 Vdc (and thus current) applied through a *pnp* transistor that acts as a switch. When DIGIT is high, the transistor (switch) is off, and the indicator current is zero. When DIGIT is low, the transistor is on, and a number is displayed. If the waveform in Fig. 15.3b is used as DIGIT, the transistor will be on and the segment will display a number for only 1 out of every 4 ms. Even though the display is not illuminated for 3 out of 4 ms, the illumination will appear to your eye as if it



(a) Single decimal-digit display

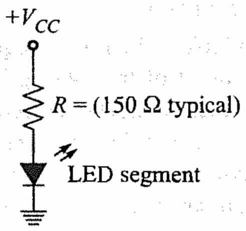


(a)



(b) Equivalent circuit for an illuminated segment

Fig. 15.1



(b) Equivalent circuit for an illuminated segment

Fig. 15.2

were continuous. Since the display is illuminated with a pulse that occurs once every 4 ms, the repetition rate (RR) is given as $RR = 1/0.004 = 250$ Hz. As a guideline, any RR greater than around 50 or 60 Hz will provide steady illumination without any perceptible flicker. The great advantage here is that this single-digit display requires only one-fourth the current of a continuously illuminated display. This then is the great advantage of multiplexing!

Let's see how to multiplex the four-digit display in Fig. 15.4a. Assume that the four BCD inputs to each digit are unchanging. If the four waveforms in Fig. 15.4b are used as the four DIGIT inputs, each digit will be illuminated for one-fourth of the time and extinguished for three-fourths of the time. Looking at the time line, we see that digit 1 is illuminated during time t_1, digit 2 during time t_2, and so on. Clearly, $t_1 = t_2 = t_3 = t_4$

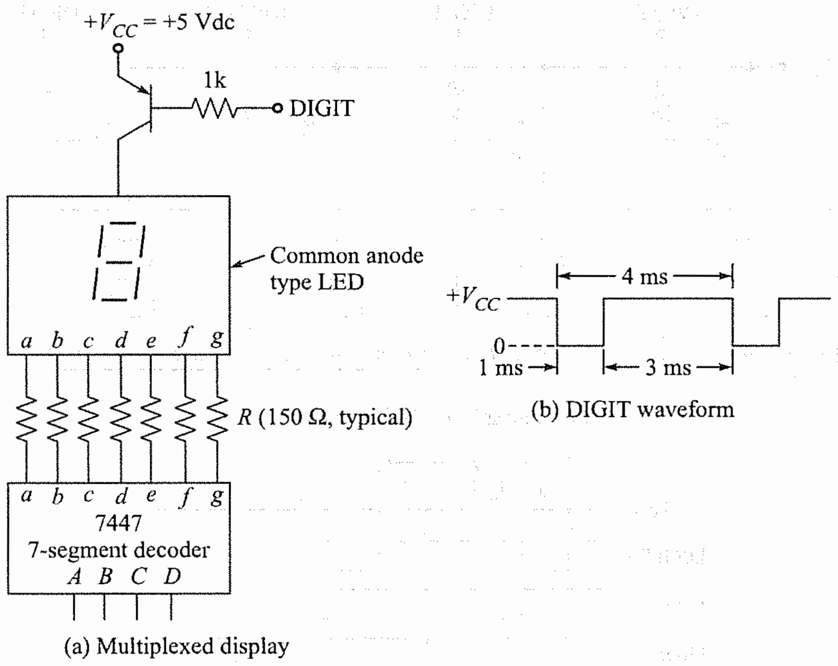


Fig. 15.3

$= T/4$. The repetition rate is given as $RR = 1/T$, and if the rate is sufficient, no flicker will appear. For instance, if $t_1 = 1$ ms, then $T = 4$ ms, and $RR = 1/0.004 = 250$ Hz.

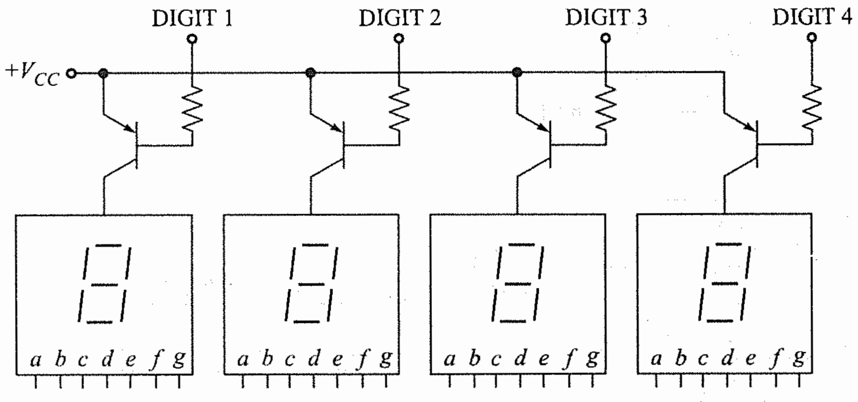
Now, here is an important concept; an illuminated digit requires 200 mA, and since only one digit is illuminated at a time, the current required from the $+V_{CC}$ supply is always 200 mA. Therefore, we are illuminating four indicators but using the current required of only a single indicator. In fact, in multiplexing displays in this way, the power supply current is simply the current required of a single display, no matter how many displays are being multiplexed!

Example 15.1 Explain the timing for a six-digit display that has a repetition rate of 125 Hz.

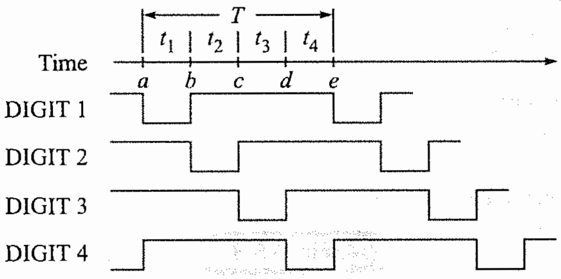
Solution An RR of 125 Hz means that all digits must be serviced once every $\frac{1}{125} = 8$ ms. Dividing the time equally among the six digits means that each digit will be on for $\frac{8}{6} = 1.33$ ms and off for 6.67 ms. Note that as the pulse width is decreased, the display brightness will also decrease. It may thus become necessary to increase the peak current through each segment by reducing the size of the resistors R in Figs. 15.1 and 15.2.

Example 15.2 The circuit in Fig. 15.3 show how to multiplex a common-anode-type display. Show how to multiplex a common-cathode-type display.

Solution The *npn* transistor in Fig. 15.5 is used as a switch between the cathode of the display and ground. When the transistor is on, current is allowed to pass through a segment for illumination. When the transistor is off, no current is allowed, and the segment cannot illuminate. The DIGIT waveform is shown in Fig. 15.5b. Notice that a positive pulse is required to turn the transistor on, and the display will be illuminated for 1 out of every 4 ms.

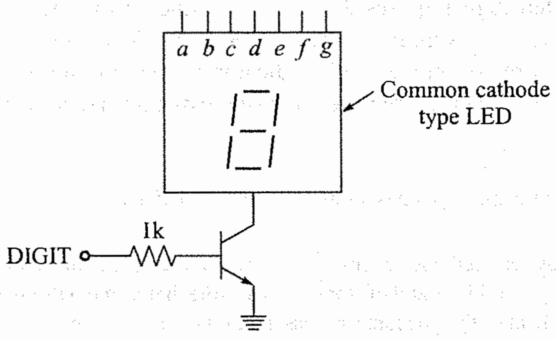


(a) A multiplexed 4-digit display

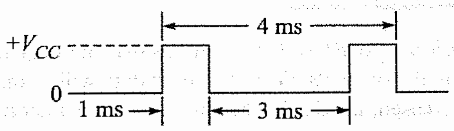


(b) Control waveforms

Fig. 15.4



(a)



(b)

Fig. 15.5

The flip-flop outputs of a 7490 decade counter are used to drive the seven-segment decoder-driver in Figs. 15.1 and 15.2, and as long as the counter is counting, the displays will be changing states. It is often more desirable to periodically strobe the contents of the counter into a four-flip-flop latch and use these latches to drive the seven-segment decoder-driver. Then the four BCD inputs to the decoder-driver as well as the display will be steady all of the time except when new data is being strobed in. The circuit shown in Fig. 15.6 is a complete single-digit display that will indicate the decimal equivalent of the binary number stored in the 7475 quad D-type latch by the positive STROBE pulse. It is also capable of being multiplexed by use of the DIGIT input.

Example 15.3 What are some possible methods for generating the DIGIT control waveforms shown in Fig. 15.4b?

Solution Reflecting back on topics covered in previous chapters, a number of different methods come to mind—for instance:

1. A two-flip-flop counter with four decoding gates
2. A four-flip-flop ring counter
3. A two-flip-flop shift counter with four decoding gates
4. A 1-of-4 low-output multiplexer

Can you think of any others?

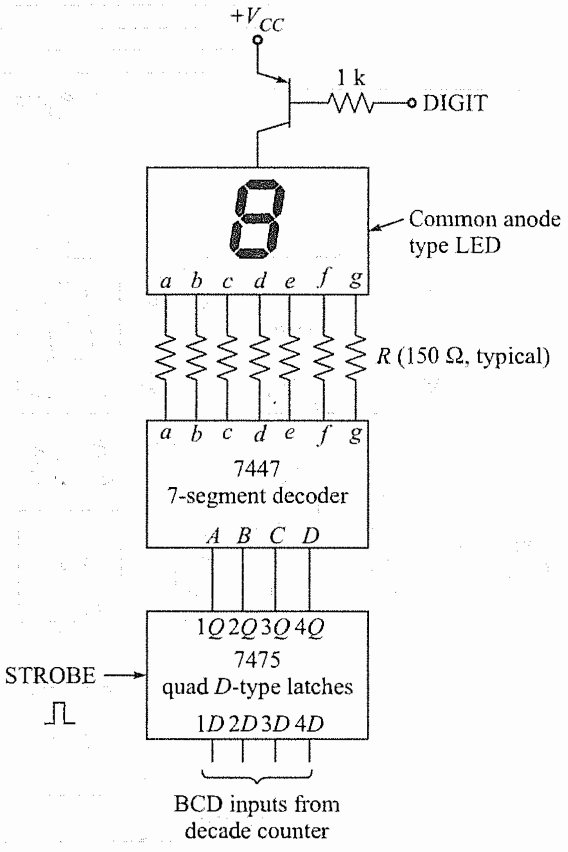


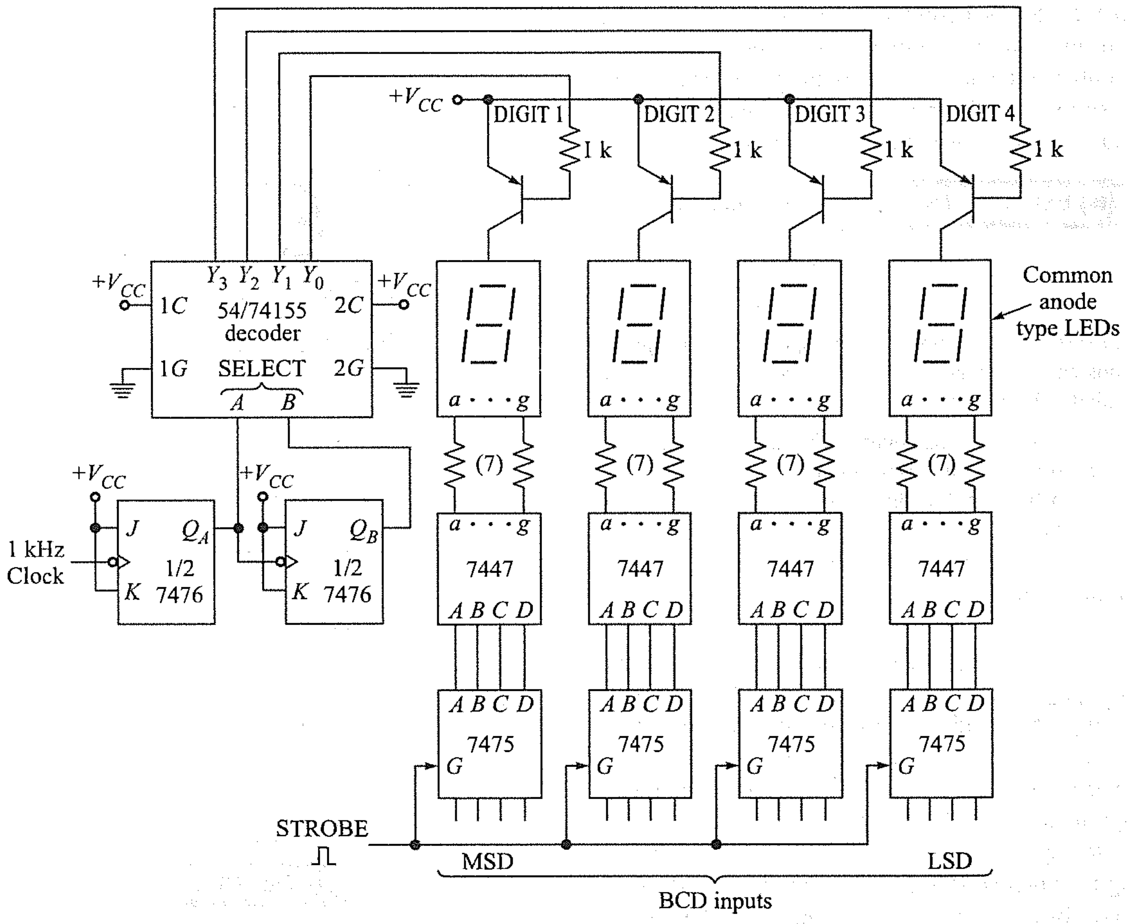
Fig. 15.6

The four-digit display in Fig. 15.7a on the next page uses four of the decimal digit displays in Fig. 15.6, and they are multiplexed to reduce power supply requirements. Notice that DIGIT 1 controls the LED on the left and this is the most-significant digit (MSD). The right display is controlled by DIGIT 4, and this is the least-significant digit counter (LSD). If we assume that the decimal point for this display to be at the right, the LSD is the units digit, and the MSD is the thousands digit. This circuit is capable of displaying decimal numbers from 0000 up to 9999. The 54/74155 is a dual 2-line to 4-line decoder-demultiplexer, and it is driven by a two-flip-flop binary counter called the *multiplexing counter*. As this multiplexing counter progresses through its four states, one and only one of the 54/74155 output lines will go low for each counter state. As a result, the DIGIT control waveforms exactly like those shown in Fig. 15.4b will be developed. You might like to review the operation of decoder-demultiplexers as discussed in Chapter 4.

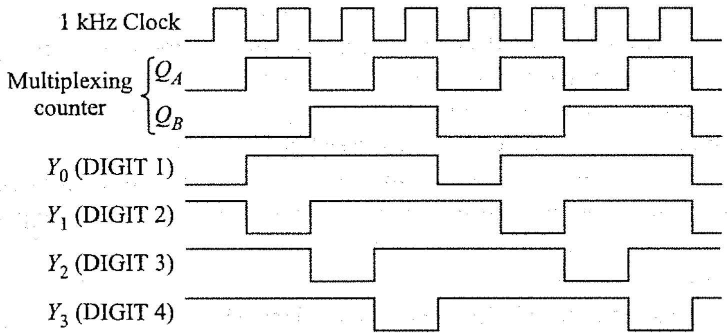
A savings in components as well as power can often be realized if the four inputs ($ABCD$) to the seven-segment decoder in Fig. 15.6 are multiplexed along with the DIGIT control. The four-digit display in Fig. 15.8 uses two 54/74153 dual 4- to 1-line multiplexers to apply the four outputs of each 7475 sequentially to a single seven-segment decoder. Here's how it works.

The BCD input data is stored in four 7475 D-type latches labeled 1, 2, 3, and 4. Latch 1 stores the MSD, and latch 4 stores the LSD. The 4-bit binary number representing the MSD is labeled $1_A 1_B 1_C 1_D$. For instance, if the MSD = 7, then $1_A 1_B 1_C 1_D = 0111$.

Each 74153 contains two multiplexers, and the four multiplexers are labeled A, B, C, and D. The A and B SELECT lines of the two multiplexers are connected in parallel and are driven by the multiplexing counter



(a) Four-decimal-digit multiplexed display



(b) Waveforms

Fig. 15.7

(exactly as in Fig. 15.7). When the SELECT inputs are $AB = 00$, the number 1 line of each multiplexer will be connected to its output. So, the multiplexer outputs connected to the 7447 decoder will be $1_A1_B1_C1_D$, which is the binary number for the MSD. This binary number is decoded by the 7447 and applied to all the LED displays in parallel. However, at this same time, DIGIT 1 is selected by the 74155 decoder, so the MSD will be displayed in the leftmost LED display. All the other displays will be turned off.

Now, when the multiplexing counter advances to count $AB = 01$, the number 2 line of each multiplexer will be selected, and the binary number applied to the 7447 will be $2_A2_B2_C2_D$, which is the next MSD (the hundreds digit). The decoded output of the 7447 is again applied to all the displays in parallel, but DIGIT 2 is the only LOW DIGIT line, so the "hundreds" digit is now displayed. (Again, all other displays are turned off during this time.)

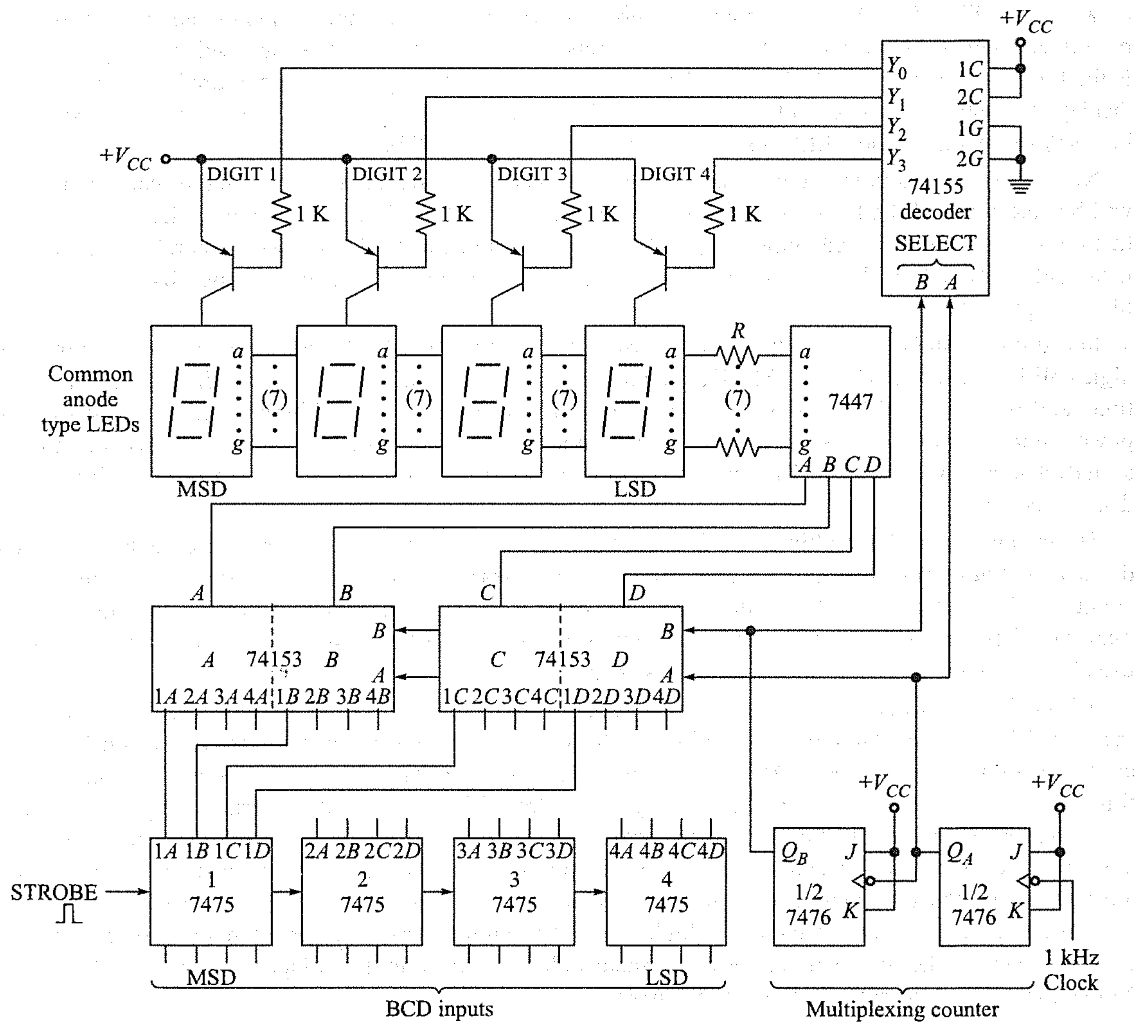
In a similar fashion, the tens digit will be displayed when the SELECT inputs are $AB = 10$, and the units digit will be displayed when the SELECT inputs are $AB = 11$. Notice that only one digit is displayed at a time, and the $RR = 250$ Hz, so no flicker will be apparent. Again, we are illuminating four digits but the power supply current is the same as for a single, continuously illuminated digit. At the same time, there is a modest saving of two chips. The savings in components increases as the number of decimal digits in the display increases.

The techniques used to multiplex the four-digit display in Fig. 15.8 on the next page are easily expanded to displays that have more than four decimal digits. It is necessary only to increase the size of the multiplexing counter and to replace the 74153 multiplexer with one that has a greater number of inputs. It is also a simple matter to alter the design to accommodate common-cathode-type LEDs instead of the common-anode types used here. (See the problems at the end of this chapter.)

All the display circuits discussed here are frequently constructed and used, but you should be aware that there are LSI chips available that have all the multiplexing accomplished on a single chip; examples of this are the National Semiconductor MM74C925, 926, 927, and 928. The MM74C925 shown in Fig. 15.9 is a four-digit counter with multiplexed seven-segment output drivers. The only external components needed are the seven-segment indicators and seven current-limiting resistors. In fact, a four-digit counter is even included on the chip! A positive pulse on the RESET input will reset the 4-bit counter, and then the counter will advance once with each negative transition of CLOCK. A negative pulse on LATCH ENABLE will then latch the contents of the counter into the four 4-bit latches. The four numbers stored are then multiplexed, decoded, and displayed on the four external seven-segment indicators. A simplified diagram is given in Fig. 15.9b. Notice that this is a common-cathode-type display.

15.2 FREQUENCY COUNTERS

A frequency counter is a digital instrument that can be used to measure the frequency of any periodic waveform. The fundamental concepts involved are illustrated in the block diagram in Fig. 15.10. The counter and display unit are exactly as described in Sec. 15.1. A GATE ENABLE signal that has a known period t is generated with a clock oscillator and a divider circuit and is applied to one leg of an AND gate. The unknown signal is applied to the other leg of the AND gate and acts as the clock for the counter. The counter will advance one count for each transition of the unknown signal, and at the end of the known time period, the contents of the counter will equal the number of periods of the unknown signal that have occurred during t. In other words, the counter contents will be proportional to the frequency of the unknown signal. For instance, suppose that the gate signal is exactly 1 s and the unknown input signal is a 750-Hz square wave. At the end of 1 s, the counter will have counted up to 750, which is exactly the frequency of the input signal.

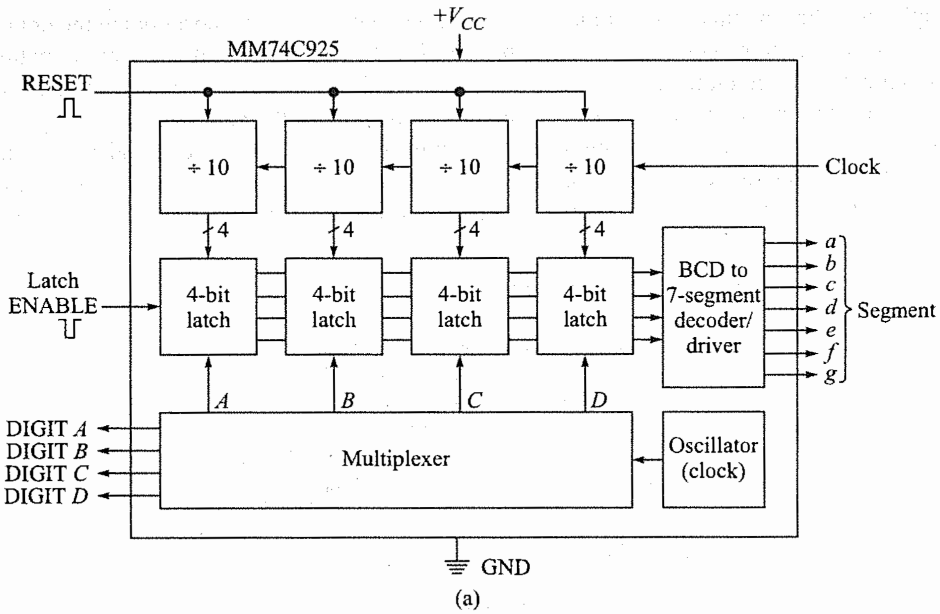


Note: For clarity, only the connections for the *MSD* are shown (1*A*, 1*B*, 1*C*, 1*D*), but the other 12 connections must be made between the input latches and the 74153s.

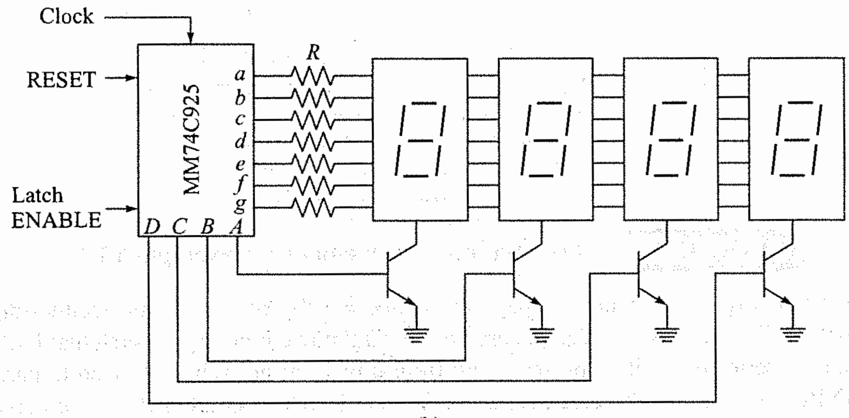
Fig. 15.8

Example 15.4 Suppose that the unknown input signal in Fig. 15.10 is a 7.50-kHz square wave. What will the display indicate if the GATE ENABLE time is $t = 0.1$ s? What if $t = 1$, and then 10 s?

Solution When $t = 0.1$ s, the counter will count up to 7500 (transitions per second) × 0.1 (second) = 750. When $t = 1$ s, the counter will display 7500 (transitions per second) × 1 (second) = 7500. When $t = 10$ s, the counter will display 7500 (transitions per second) × 10 (seconds) = 75,000. For this last case, we would have to have a five-decimal-digit display.



(a)



(b)

Fig. 15.9

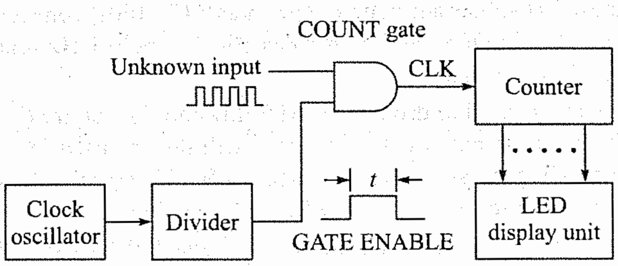


Fig. 15.10 **Basic frequency counter**

In Example 15.4, the contents of the counter are always a number that is proportional to the unknown input frequency. In this case, the proportionality constant is either 10, 1, or $\frac{1}{10}$. So, it is a simple matter to insert a decimal point between the indicators such that the unknown frequency is displayed directly. Figure 15.11 shows how the decimal point moves in a five-decimal-digit display as the gate width is changed. In the top display, the unknown frequency is the display contents multiplied by 10, so the decimal point is moved one place to the right. The middle display provides the actual unknown frequency directly. In the bottom display, the contents must be divided by 10 to obtain the unknown frequency, so the decimal point is moved one place to the left.

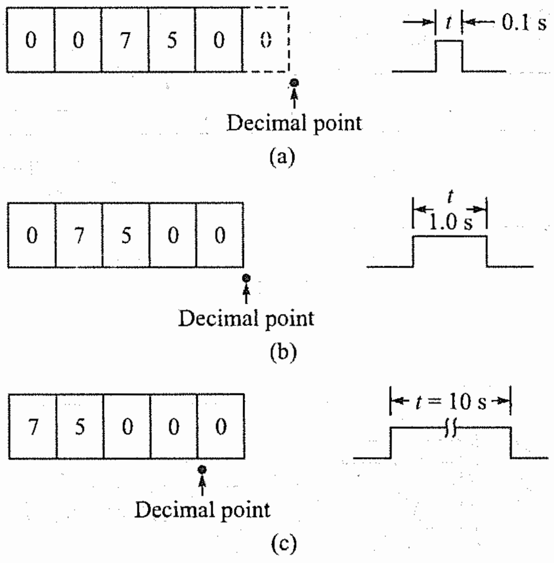


Fig. 15.11 **Decimal point movement for Example 15.4**

The logic diagram in Fig. 15.12 on the next page shows one way to construct a four-decimal-digit frequency counter. The AMPLIFIER block is intended to condition the unknown input signal such that INPUT is a TTL-compatible signal—a series of positive pulses going from 0 to +5 V dc. When allowed to pass through the COUNT gate, INPUT will act as the clock for the COUNTER. The COUNTER can be constructed from four decade counters such as 54/74160s, and it can then be connected to a multiplexed LED DISPLAY such as the one shown in Fig. 15.8. Or, COUNTER and DISPLAY can be combined in a single chip such as the MM74C925 shown in Fig. 15.9.

The DIVIDER is composed of six decade counters (such as 54/74160s) connected in series. Its input is a 100-kHz square wave from OSC CLOCK, and it provides 10-, 1-, and 0.1-Hz square wave outputs that are used to generate the ENABLE-gate signal.

When the 1-Hz square wave is used to drive the GATE flip-flop, its output, Q, is a 0.5-Hz square wave. Output Q will be high for exactly 1s and low for 1s, and it will thus be used for the ENABLE-gate signal. Notice that the 10-Hz signal will generate a 0.1-s gate and the 0.1-Hz signal will generate a 10-s gate. Let's use the waveforms in Fig. 15.12 to see exactly how the circuit functions.

A measurement period begins when the GATE flip-flop is toggled high—labeled *START* on the time line. INPUT now passes through the COUNT gate and advances the COUNTER. (Let's assume that the counter is initially at 0000.) At the end of the ENABLE-gate time t, the GATE flip-flop toggles low, the COUNTER ceases to advance, and this negative transition of Q triggers the 74121 one-shot. Simultaneously, $\overline{Q}$ goes high, and this will strobe the contents of COUNTER into the DISPLAY latches. There is a propagation delay time of 30 ns minimum through the 74121, and then a negative RESET pulse appears at its output, $\overline{X}$. This propagation delay assures that the contents of COUNTER are strobed into DISPLAY before COUNTER is reset. The RESET pulse from the '121 has an arbitrary width of 1 μs, as set by its R and C timing components. The end of the RESET pulse is the end of one measurement period, labeled END on the time line.

For a 1.0-s gate, the decimal point will be at the right of the units digit, and the counter will be capable of counting up to 9999 full scale, with an accuracy of plus or minus one count (i.e. 1 part in 10^4). With a 10-s gate, the decimal point is between the units and the tens digits, and with a 0.1-s gate, the decimal point is one place to the right of the units digit.

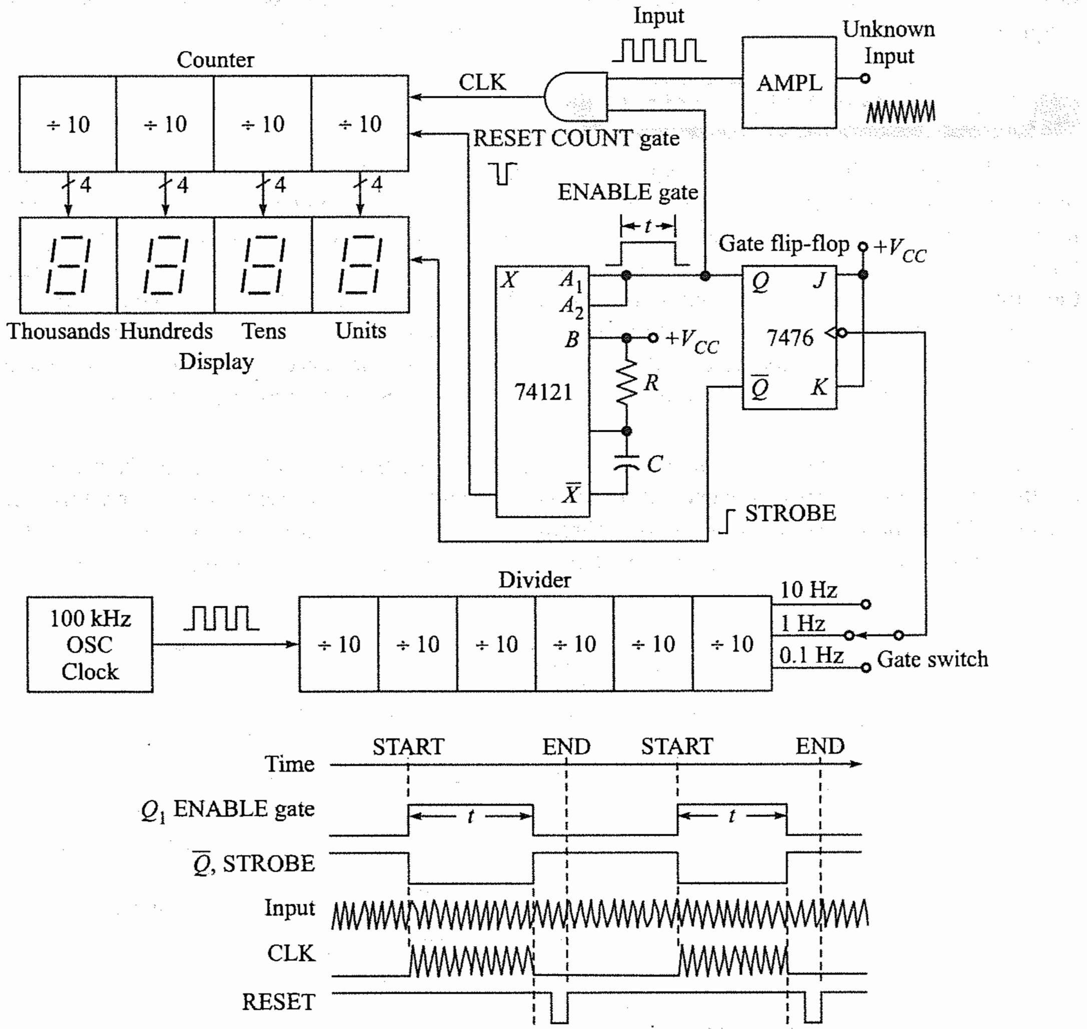


Fig. 15.12 **Four-decimal-digit frequency counter**

The CLOCK oscillator is set at 100 kHz, and this provides an accuracy on the ENABLE-gate time of 1 part in 10^4 with the 0.1-s gate. Thus the accuracy here is compatible with that of the COUNTER.

Example 15.5 Explain what would happen if the instrument in Fig. 15.12 were set on a 1-s gate time and the input signal were 12 kHz.

Solution Assuming that the counter began at 0000, the display would read 200 at the end of the first measurement period. It would then read 400, then 600, and so on at the end of succeeding periods. This is because the counter capacity is exceeded each time, and it simply recycles through 0000.

The design in Fig. 15.12 shows one method for constructing a frequency counter using readily available TTL chips, but you should be aware that there are numerous chips available that have all, or nearly all, of this design on a single chip, for instance, the Intersil ICM7226A. You will be asked to do a complete design of a frequency counter based on Fig. 15.12 in one of the problems at the end of this chapter.

15.3 TIME MEASUREMENT

With only slight modifications, the frequency counter in Fig. 15.10 can be converted into an instrument for measuring time. The logic block diagram in Fig. 15.13 illustrates the fundamental ideas used to construct an instrument that can be used to measure the period of any periodic waveform. The unknown voltage is passed through a conditioning amplifier to produce a periodic waveform that is compatible with TTL circuits and is then applied to a *JK* flip-flop. The output of this flip-flop is used as the ENABLE-gate signal, since it is high for a time *t* that is exactly equal to the time period of the unknown input voltage. The oscillator and divider provide a series of pulses that are passed through the count gate and serve as the clock for the counter. The contents of the counter and display unit will then be proportional to the time period of the unknown input signal.

For instance, if the unknown input signal is a 5-kHz sine wave and the clock pulses from the divider are 0.1 μs in width and are spaced every 1.0 μs, the counter and display will read 200. Clearly this means 200 μs, since 200 of these 0.1-μs pulses will pass through the COUNT gate during the 200 μs that ENABLE-gate signal is high. Naturally the counter and the display have an accuracy of plus or minus one count.

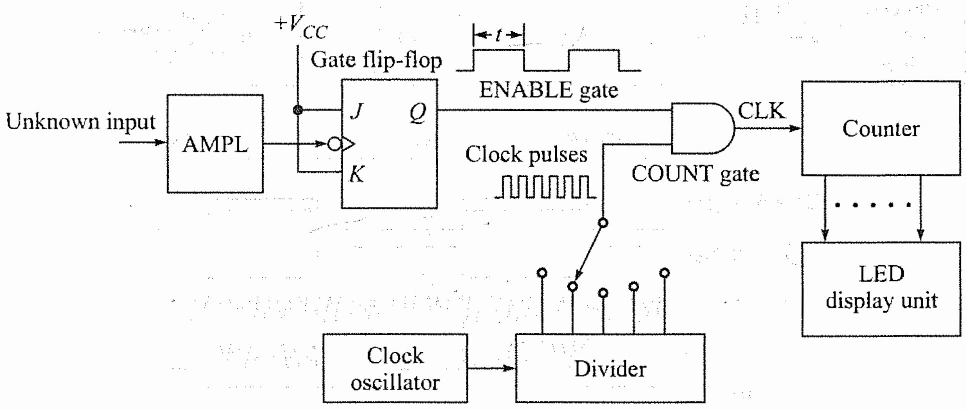


Fig. 15.13 **Instrument to measure time period**

Example 15.6 Suppose that the counter and the display unit in Fig. 15.13 have five-decimal-digit capacity and the divider switch is set to provide a 100-kHz square wave that will be used as clock pulses. What will the display read after one ENABLE-gate time t, if the unknown input is a 200-Hz square wave?

Solution Assume that the counter and the display are initially at 00000. A 200-Hz input signal will produce an ENABLE-gate time of $t = \frac{1}{200} = 5000$ μs. The 100-kHz square wave used as the clock is essentially a series of positive pulses spaced by 10 μs. Therefore, during the gate time t, the counter will advance by, $\frac{5000}{10} = 500$ counts, and this is what will be viewed in the displays. Since each clock pulse represents 10 μs, the display should be read as $500 \times 10 = 5000$ μs—this is the time period of the unknown input.

Example 15.7 Explain the meaning of an accuracy of plus or minus one count applied to the measurement in Example 15.6.

Solution An accuracy of plus or minus one count means that the display could read 499, 500, or 501 after the measurement period. This means that the period as measured could be 4990, 5000, or 5010 μs—in other words, 5000 plus or minus 10 μs. Since a single count represents a clock period of 10 μs, this instrument can be used for measurement only within this limit. For more precise measurement, say, to within 1 μs, the clock pulses would have to be changed from 10- to 1-μs spacing.

The circuit in Fig. 15.14 is a four-decimal-digit instrument for measuring the time period of a periodic waveform. It is essentially the same as the frequency instrument in Fig. 15.12 with only slight modifications. First, the CLOCK has been increased to 1 MHz, and DIVIDER is composed of a buffer amplifier and three decade counters. This will provide clock pulses for COUNTER with 1-, 10-, and 100-μs as well as 1-ms spacing. The unknown input is conditioned by AMPLIFIER and is then applied to the GATE flip-flop to generate the ENABLE-gate signal. STROBE and RESET are generated and applied as before. Notice that a single instrument for measuring both frequency and period could be easily designed by using a 1-MHz clock with a divider that has seven decade counters and some simple mechanical switches.

Example 15.8 Explain the DISPLAY ranges for the four-decimal-digit period measurement instrument in Fig. 15.14.

Solution With CLOCK pulses switched to the 1-μs position, each count of COUNTER represents 1 μs. Therefore, it has a full scale of 9999 ± 1 μs. On 10 μs, it has a full scale of $9999 \times 10 = 99{,}990 \pm 10$ μs. Full scale on the 0.1-ms position is $9999 \times 0.1 = 999.9 \pm 0.1$ ms. Full scale on the 1-ms position is 9999 ± 1 ms.

An interesting variation on the instrument in Fig. 15.14 is to use it to measure the time elapsed between two events. There will be two input signals, the first of which sets the ENABLE gate high and begins the count period. The second signal (or event) resets the ENABLE gate low and completes the time period. One method for handling this problem is to use the first event to set a flip-flop and then use the second event to reset it. Of course, both input signals must be first conditioned such that they are TTL-compatible. You are given the opportunity to design such an instrument in one of the problems at the end of the chapter.

15.4 USING THE ADC0804

Stand-Alone Operation

The ADC0804 was briefly introduced in Sec. 12.8. Figure 12.29 shows how to connect the ADC0804 for stand-alone operation, and it is repeated here for convenience. The recommended supply voltage is $V_{CC} = +5$

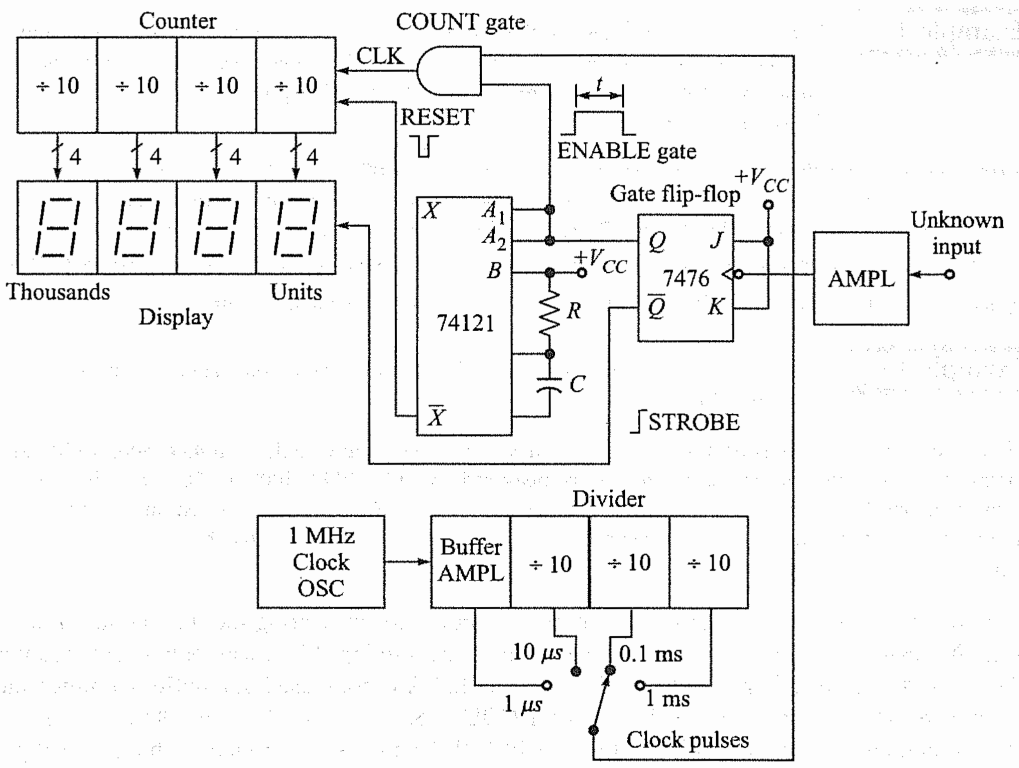


Fig. 15.14 **Four-decimal-digit period measurement instrument**

Vdc. The external resistor R and capacitor C establish the frequency of the internal clock according to

$$f \cong 1/(1.1\,RC) \tag{15.1}$$

Pin 9 is an input for an external reference voltage V_{ref}. If pin 9 is left open, the reference voltage is set internally at $V_{CC}/2$. The analog input voltage is applied between pins 6 and 7. With pin 7 connected to ground, the allowable input voltage range is from 0.0 to +5.0 V.

This ADC is designed for use with the 8080A CPU (central processing unit) chip set, composed of the 8080A microprocessor, the 8228 system controller, and the 8224 clock. It can also be used directly with the 8048 MPU (microprocessor unit). The inputs $\overline{\text{WR}}$, $\overline{\text{INTR}}$, $\overline{\text{CS}}$ and $\overline{\text{RD}}$ are microprocessor control signals. In the stand-alone mode of operation, $\overline{\text{WR}}$ and $\overline{\text{INTR}}$ are connected directly to ground. $\overline{\text{CS}}$ and $\overline{\text{RD}}$ are momentarily grounded with a push-button switch to initiate a conversion. The converter will digitize the analog voltage present at the input at the instant the push button is depressed. It will then continue to convert additional analog input voltage levels at approximately 100-μs intervals.

The digitized value of an input voltage is presented as an 8-bit binary number on pins 11 through 18, with pin 11 the most significant bit (MSB). An input voltage of 0.0 V has a digitized output of 0000 0000 (OOH). The digital output 1111 1111 (FFH) represents a full-scale input of +5.0 V. The digitized output is accurate to ±1 LSB. Since the full-scale input of 5.0 V is represented by $2^8 = 256$ bits, 1 bit (the LSB) is equivalent to an analog voltage of 5.0 V/256 = 19.53 mV.

Example 15.9 Refer to the ADC0804 in Fig. 12.29, repeated below for your reference.

(a) What is the digital output for an analog input of 2.5 V?
(b) The digital output is 0010 0010 (22H). What is the analog input?

Solution

(a) 2.5 V is one-half full scale. The digital output is then 1000 0000 ±1 bit ($2^7 = 128$). As a check, 128 × 19.53 mV = 2.5 V
(b) $(2^5 + 2^1) \times 19.53$ mV = (32 + 2) × 19.53 = 0.664 V.

Span Adjust

As shown in Fig. 12.29, the ADC0804 functions nicely for analog input voltages between 0.0 and +5.0 V. But what if the input voltage range is only from 0.0 to 2.0 V? In this case, we would like the full-scale input to be 2.0 V rather than 5.0 V. Fortunately, this is quite easy to do with the ADC0804! Simply connect an external reference voltage V_{ref} to pin 9 that is *one-half* the desired full-scale input voltage. Another term for the full-scale input voltage range is *span*. In this case, set V_{ref} = 2.0 V/2 = 1.0 Vdc. In general terms,

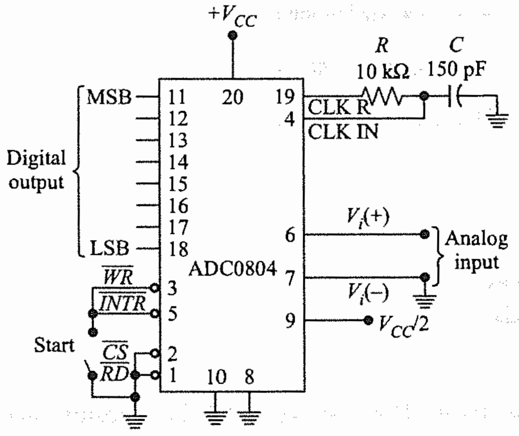
$$V_{ref} = \text{full-scale analog input voltage}/2 = \text{span}/2 \tag{15.2}$$

In Fig. 15.15, a simple resistive divider is used to generate the reference voltage V_{ref}. Here's an expression to use with this divider:

$$V_{ref} = V_{CC} \frac{R_2 + R/2}{R_1 + R_2 + R} \tag{15.3}$$

As an example, let's apply Eq. (15.3) using the circuit values given in Fig. 15.15.

$$V_{ref} = +5\text{ Vdc}\,\frac{1\text{ k}\Omega + 0.25\text{ k}\Omega}{1\text{ k}\Omega + 4.7\text{ k}\Omega + 0.5\text{ k}\Omega} = 1.01\text{ Vdc}$$



Stand-alone operation (Fig. 12.29 repeated)

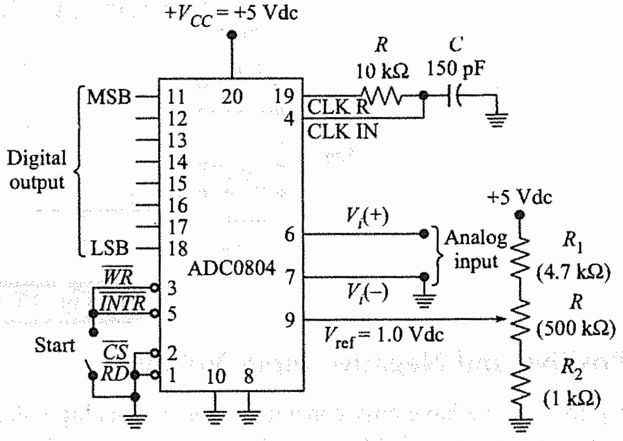


Fig. 15.15

In this case, $V_{ref} = 1.0$ Vdc. The 500-Ω potentiometer will allow fine adjustment. So the full-scale analog input voltage is then 0.0 to 2.0 V. An input voltage of 2.0 V will convert to a digital output 1111 1111 (FFH). The LSB is equivalent to 2.0 V/256 ≡ 7.8 mV.

Zero Shift

The ADC0804 can also accommodate analog input voltages that are offset from zero. For instance, suppose we wish to digitize an analog signal that is always between the limits +1.5 V and +4.0 V, as illustrated in Fig. 15.16a. The span of this signal is (4.0 – 1.5) V = 2.5 V. So we would use Eq. (15.2) to find

$$V_{ref} = \frac{\text{span}}{2} = \frac{2.5 \text{ volts}}{2} = 1.25 \text{ Vdc}$$

This reference voltage (1.25 Vdc) is applied to pin 9.

Now we connect pin 7 to the *lower limit* of the input voltage. This lower limit is called the *OFFSET*. In general terms,

$$\text{OFFSET at } V_{i\overline{v}} = \text{analog input lower limit} \tag{15.4}$$

In Fig. 15.16b, we have used two voltage dividers, one for $V_{ref} = 1.25$ Vdc and one for $V_{i-} = 1.5$ Vdc. For this circuit an analog input voltage of 1.5 V will be digitized as 0000 0000. An input of 4.0 V will convert to 1111 1111. This LSB is then equivalent to 2.5 V/256 ≅ 9.77 mV.

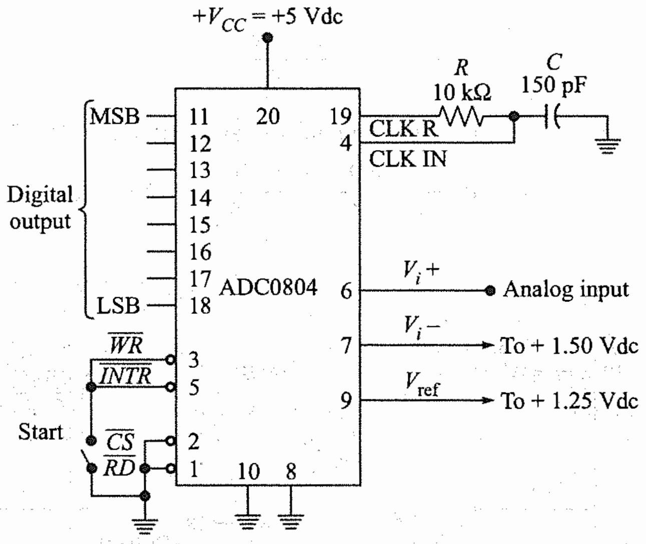


Fig. 15.16

Positive and Negative Input Voltages

Up to now, we have only considered positive analog voltage levels. How can we handle both positive and negative input signals? For instance, suppose we wish to digitize analog voltages that vary from –5 to +5 Vdc. One solution is given in Fig. 15.17. The technique is to use a resistive voltage divider (*R* and *R*) at the input pin 6. Pin 7 is connected to ground.

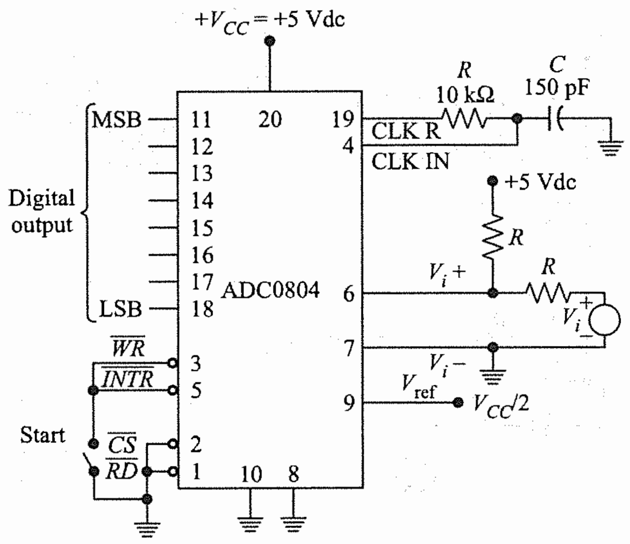


Fig. 15.17

A little thought will reveal the following:

1. $V_i = -5$ V. Then $V_{i+} = 0.0$ V. The digitized output is 0000 0000 (00H).
2. $V_i = 0.0$ V. Then $V_{i+} = +2.5$ V. The digitized output is 1000 0000 (80H). This is mid scale.
3. $V_i = +5$ V. Then $V_{i+} = +5.0$ V. The digitized output is 1111 1111 (FFH). This is full scale.

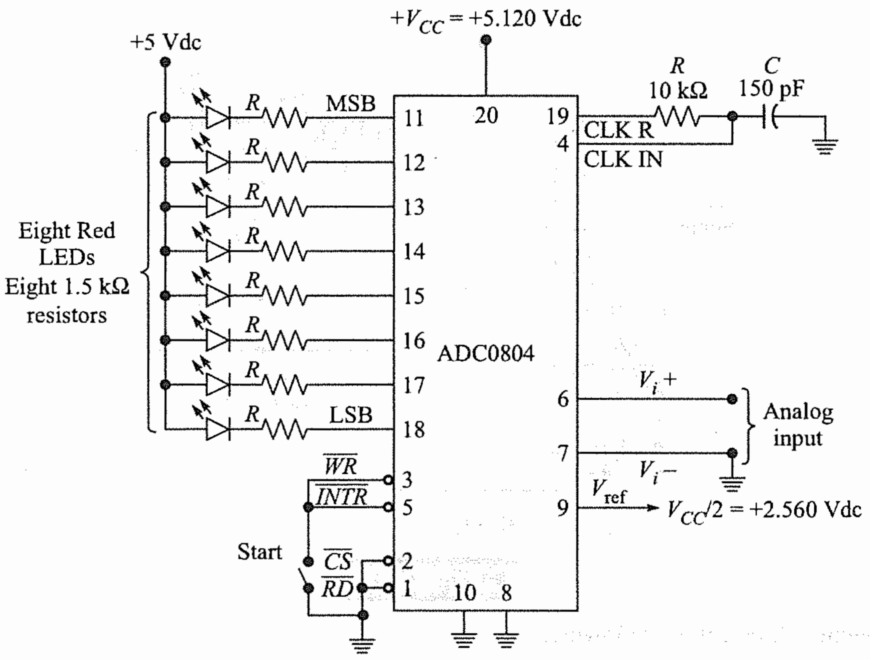
The span at V_{i+} is clearly 5.0 V. OFFSET is not required, since the voltage at V_{i+} varies between 0.0 V and +5.0 V. Notice that a *negative* input voltage, V_i, always produces a 0 for the MSB of the digital output (with the possible exception of 0.0). A *positive* input voltage, V_i, always produces a 1. for the MSB of the digital output (again, with the possible exception of 0.0). In this case, the LSB is equivalent to 10 V/256 = 39.01 mV.

Testing

When using an ADC0804, it may become necessary to test it for proper operation, for example, before initial installation or perhaps to troubleshoot a suspected malfunction. There are many different testing procedures for A/D converters, some of which are quite complex and computer-controlled. However, a rapid and simple test is to apply a known analog input voltage while monitoring the digital outputs. The test circuit in Fig. 15.18 on the next page can be used for this purpose. Notice that the dc supply voltage has been adjusted carefully to a value $V_{CC} = 5.120$ Vdc. Also, V_{ref} has been set at $V_{CC}/2 = 2.560$ Vdc. These values have been chosen so that the LSB has a weight of 5.120 V/256 = 20 mV. This eliminates any round off error and makes the arithmetic easier!

A checkerboard-type test is used to activate each output. Here's how to do it:

1. Apply an input voltage to produce the digital output 1010 1010 (AAH). The required input is (128 + 32 + 8 + 2) 20 mV = 3.400 V.
2. Apply an input voltage to produce the digital output 0101 0101 (55H). The required input is (64 + 16 + 4 + 1) 20 mV = 1.700 V.



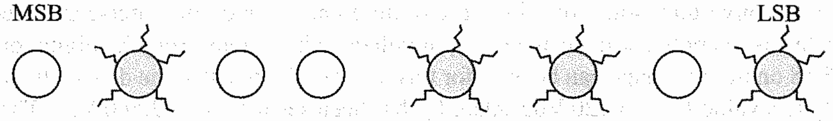
Note: Illuminated LED ≡ output = low = 0
Extinguished LED ≡ output = high = 1

Fig. 15.18

These two tests will activate all eight outputs in both states. This is not a comprehensive test, but it will detect any faults in the outputs, and it will thus give a reasonable degree of confidence in the operation of the A/D converter. Note carefully that a digital output 1 (high) will extinguish the LED. A low ouput (a 0) will illuminate an LED. So,

Illuminated LED ≡ low ≡ 0
Extinguished LED ≡ high ≡ 1

For example, the output 1011 0010 is "seen" as



SELF-TEST

1. Why are the external R and C in Fig. 12.29 needed?
2. What is the purpose of the START button in Fig. 12.29?
3. The digital output or an ADC0804 is 1100 0011. What is this in hexadecimal?
4. For the ADC0804 what do the terms *span* and *OFFSET* mean?

15.5 MICROPROCESSOR-COMPATIBLE A/D CONVERTERS

A fundamental requirement in many digital data acquisition systems is an A/D converter that is simple, reliable, accurate, inexpensive, and readily usable with a minicomputer or microprocessor. The National Semiconductor ADC3511 is a single-chip A/D converter constructed with CMOS technology that has $3\frac{1}{2}$-digit BCD outputs designed specifically for use with a microprocessor, and it is available for less than \$9! The 3511 uses an integrating-type conversion technique and is considerably slower than flash-type or SAR-type A/D converters. It is quite useful in digitizing quantities such as temperature, pressure, or displacement, where fewer than five conversions per second are adequate. The pinout and logic block diagram for an ADC3511 are shown in Fig. 15.19.

Only a single +5-Vdc power supply is required, and the 3511 is completely TTL-compatible. This A/D converter is a very high precision analog device, and great care must be taken to ensure good grounding, power supply regulation, and decoupling. It is important that a single GROUND point be eatablished at pin 13, to eliminate any ground loop currents. Voltage V_{CC} on pin 1 is used to apply +5-Vdc power. A 10-μF 10-Vdc capacitor is connected between pin 2 and GROUND; this capacitor, and the internal 100-Ω resistor shown on the logic block diagram are used to decouple the dc power used for the analog and digital circuits. Voltage V_{SS} on pin 22 should also be connected directly to GROUND.

The conversion rate of the chip is established by a resistor R connected between pins 17 and 18 and a capacitor C connected between pin 17 and GROUND. The clock frequency developed by these two components is given by $f = 0.6/RC$ and should be set between 100 and 640 kHz. This is the clock signal used to advance the internal counters thal finally store the digitized value of the analog input voltage.

The analog signal to be digitized is applied between $+V_i$ and $-V_i$, pins 11 and 10, respectively. Negative signals are handled automatically by the converter through the switching network at the input to the comparator. A conversion is initiated with a low-to-high transition of START CONVERSION on pin 7. The waveform, CONVERSION COMPLETE, on pin 6 will go low at the beginning of a conversion cycle and then return high at the end of a conversion cycle. Connecting pin 7 to $+V_{CC}$ will cause the chip to continuously convert the analog input signal. The using edge of the waveform on pin 6 indicates that new digital information has been transferred to the digit latches and is available for output.

The digitized analog signal is contained in the converter as four BCD digits. The LSD, or units digit, is $D_1C_1B_1A_1$, the tens digit is $D_2C_2B_2A_2$, the hundreds digit is $D_3C_3B_3A_3$, and the MSD is $C_4B_4A_4$. All digits can store the BCD equivalent of decimal 0, 1, 2, ... , 9, except the MSD. The MSD can have values of only decimal 0 or decimal 1. It is for this reason that the 3511 is called a $3\frac{1}{2}$ digit device—the MSD is referred to as a *half-digit*. The 16 × 4 MUX is used to multiplex one digit (4 bits) at a time to the outputs according to the input signals D_0 and D_1 as given in Fig. 15.20. For instance, when $D_1D_0 = 00$, the LSD appears on the four output lines 23, 22, 21, and 20. A low-to-high transition on DIGIT LATCH ENABLE (DLE), pin 19, will latch the inputs D_1D_0, and the selected digit will remain on the four output pins until DLE returns low. The polarity of the digitized input analog signal will also appear on pin 8, SIGN. The 3511 has a full-scale count of 1999, and if this count is exceeded, an overflow condition occurs and the four digit outputs will indicate *EEEE*.

The heart of the analog-to-digital conversion consists of the comparator, the *D*-type flip-flop, and an *RC* network that is periodically switched between a reference voltage V_{ref} and ground. When the output of the *D*-type flip-flop, Q, is high, the transistor designated as SW_1 is on, and the other transistor designated as SW_2 is off. Under this condition, the capacitor C charges through R toward the reference voltage (usually +2.00

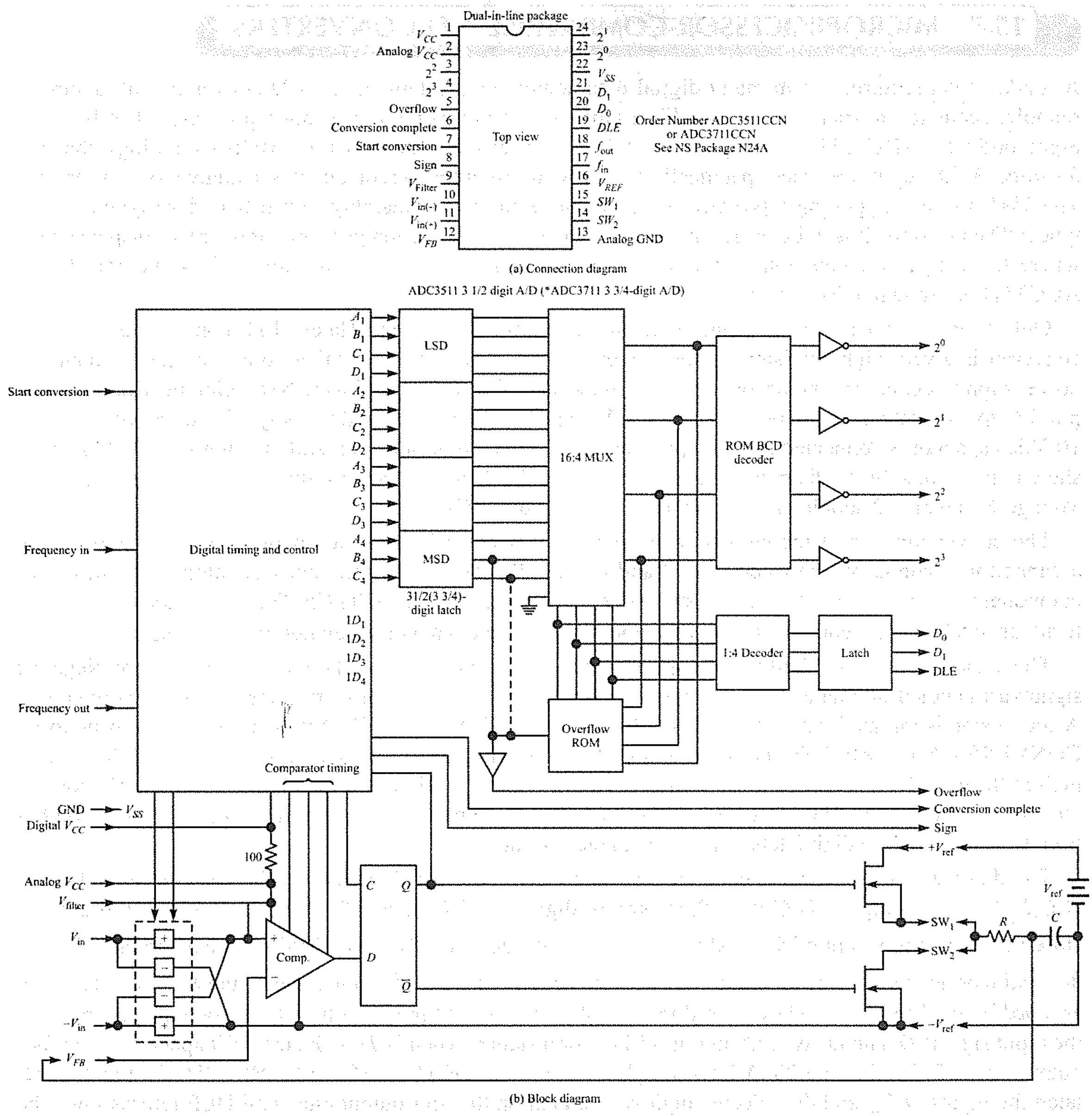


Fig. 15.19 National Semiconductor ADC3511 (3711)

Vdc), and the capacitor voltage V_{fb}, is fed back to the negative input terminal of the comparator. When V_{fb} exceeds the analog input voltage, the comparator output switches low, and the next clock pulse will set Q low. When Q is low, SW_1 is off and SW_2 is on. The capacitor now discharges through resistor R toward 0.0 Vdc, As soon as V_{fb} discharges below the analog input voltage, the comparator output will switch back to a high state, and this process will repeat.

These components form a closed-loop system that will oscillate—that is, a rectangular waveform as shown in Fig. 15.21 will be produced at SW_1 and SW_2 (pins 15 and 14).

The duty cycle of this waveform is given as

$$\text{Duty cycle} = \frac{t_c}{t_c + t_d}$$

and its dc value is given as

$$V_{dc} = V_{ref} \times \text{duty cycle}$$

This dc voltage will appear at V_{fb} and the closed-loop system will adjust itself such that

$$V_i = V_{dc} = V_{ref} \times \text{duty cycle}$$

or

$$\frac{V_i}{V_{ref}} = \text{duty cycle} = \frac{t_c}{t_c + t_d}$$

DIGIT SELECT Inputs			
DLE	D_1	D_0	Selected DIGIT
L	*L*	*L*	DIGIT 0 (LSD)
L	*L*	*H*	DIGIT 1
L	*H*	*L*	DIGIT 2
L	*H*	*H*	DIGIT 3 (MSD)
H	*X*	*X*	No change

L = Low logic level
H = High logic level
X = Irrelevent-logic level
The value of the selected digit is presented at the 2^3, 2^2, 2^1, and 2^0 outputs in BCD format.

Fig. 15.20 ADC 3511 (3711) control levels

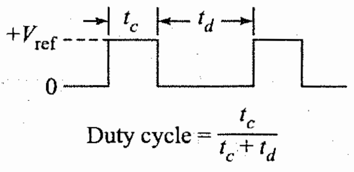


Fig. 15.21 Waveforms at SW_1 and SW_2 (pins 15 and 14 respectively) for the ADC3511

The maximum allowable value for the analog input voltage is V_{ref}. When the input is equal to V_{ref}, the duty cycle must be equal to 1.0 ($t_d = 0$) and Q is always high. If the input analog signal is 0.0. the duty cycle must be zero ($t_c = 0$), and Q is always low. For an analog input voltage between 0.0 and $+V_{ref}$, the duty cycle is some value between 0.0 and 1.0.

The waveform Q at the output of the D-type flip-flop has exactly the same duty cycle as V_{fb}, and it is used to gate a counter in the converter. The counter can only advance when Q is high, and the gating is arranged such that for a duty cycle of 1.0, the counter will count full scale (1999), and for a duty cycle of 0, the counter will count 0000. For any duty cycle between 0.0 and 1.0, the counter will count a proportional amount between 0000 and 1999. In fact, the exact COUNT relationship is given as

$$\text{COUNT} = N \times \frac{V_i}{V_{ref}}$$

where N is the full-scale count of 2000.

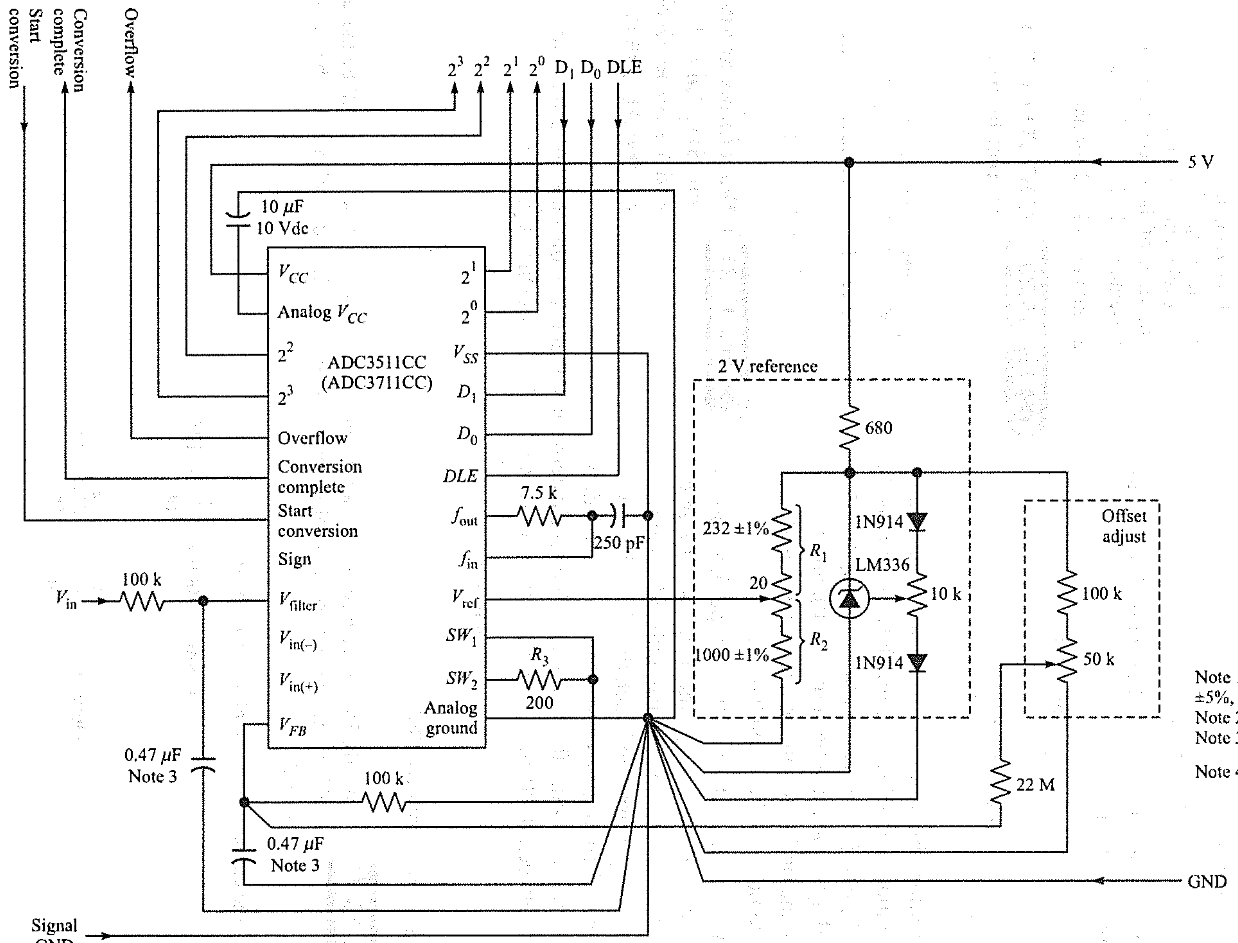
Example 15.10 An ADC3511 is connected with a reference voltage of +2.0 Vdc. What will be the count held in the counter for an analog input voltage of 1.25 Vdc? What must be the duty cycle?

Solution Using the expression given above, we obtain

$$\text{Count} = 2000 \times \frac{1.25}{2.00} = 1250$$

The duty cycle must be

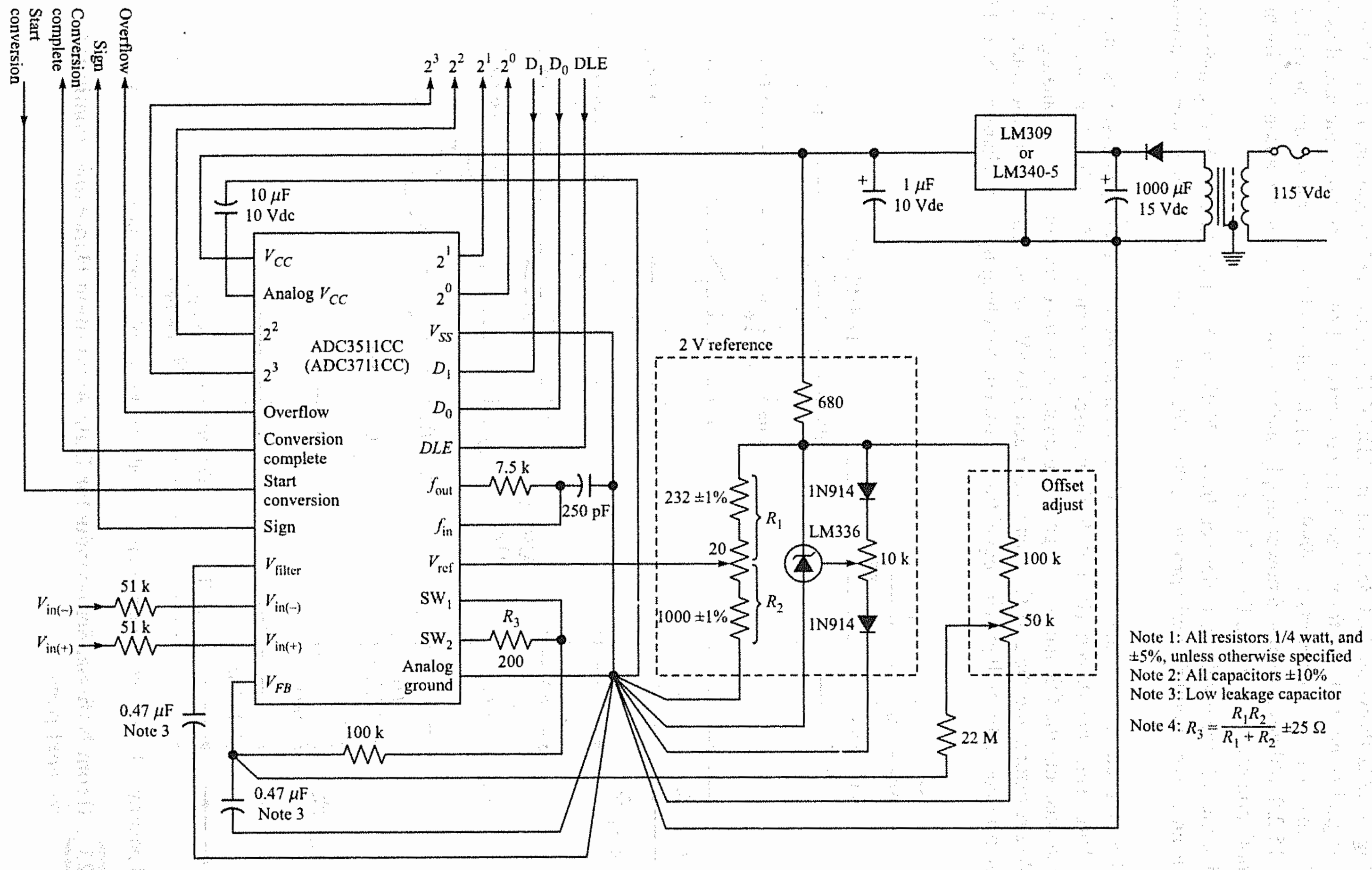
$$\text{Duty cycle} = \frac{V_i}{V_{ref}} = \frac{1.25}{2.00} = 0.625$$



Note 1: All resistors 1/4 watt, and ±5%, unless otherwise specified
Note 2: All capacitors ±10%
Note 3: Low leakage capacitor
Note 4: $R_3 = \dfrac{R_1 R_2}{R_1 + R_2}$ ±2 Ω

(a) 3 1/2-digit A/D; +1999 counts, +2.000 volts full scale (3 3/4 digit A/D; +3999 counts, +2.000 volts full scale)

Fig. 15.22 **(From National Semiconductor Data Acquisition Handbook; continued on next page)**



Note 1: All resistors 1/4 watt, and ±5%, unless otherwise specified
Note 2: All capacitors ±10%
Note 3: Low leakage capacitor
Note 4: $R_3 = \frac{R_1R_2}{R_1 + R_2} \pm 25\ \Omega$

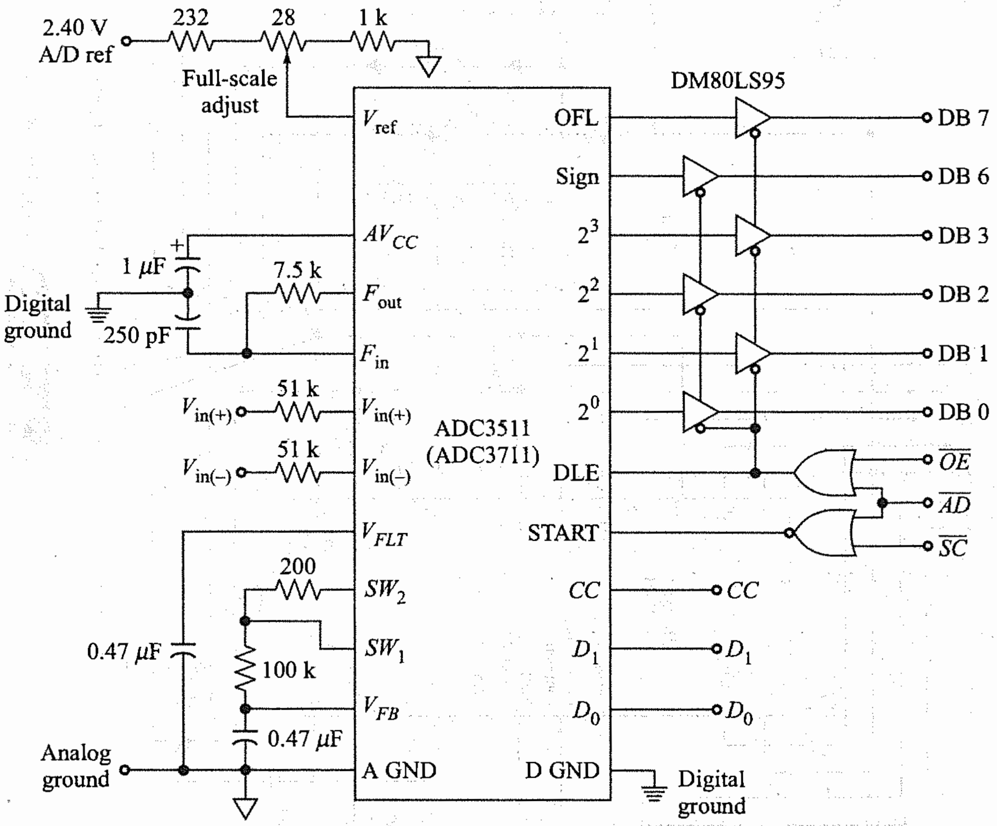
(b) 3 1/2-digit A/D; ±1999 counts, ±2.000 volts full scale (3 3/4 digit A/D; ±3999 counts, ±2.000 volts full scale)

Fig. 15.22 **(Continued)**

The circuit in Fig. 15.22a shows an ADC3511 (or an ADC3711) connected to convert 0.0 to +2.00 Vdc into an equivalent digital signal in BCD form. The 3511 converts to 1999 counts full scale and thus has a 1-bit resolution of 1 mV. The 3711 converts to 3999 counts full scale and has a 1-bit resolution of 0.5 mV. The circuit in Fig. 15.22b utilizes an isolated power supply such that the converter can automatically handle input voltages of both polarities—from +2.0 to –2.0 Vdc.

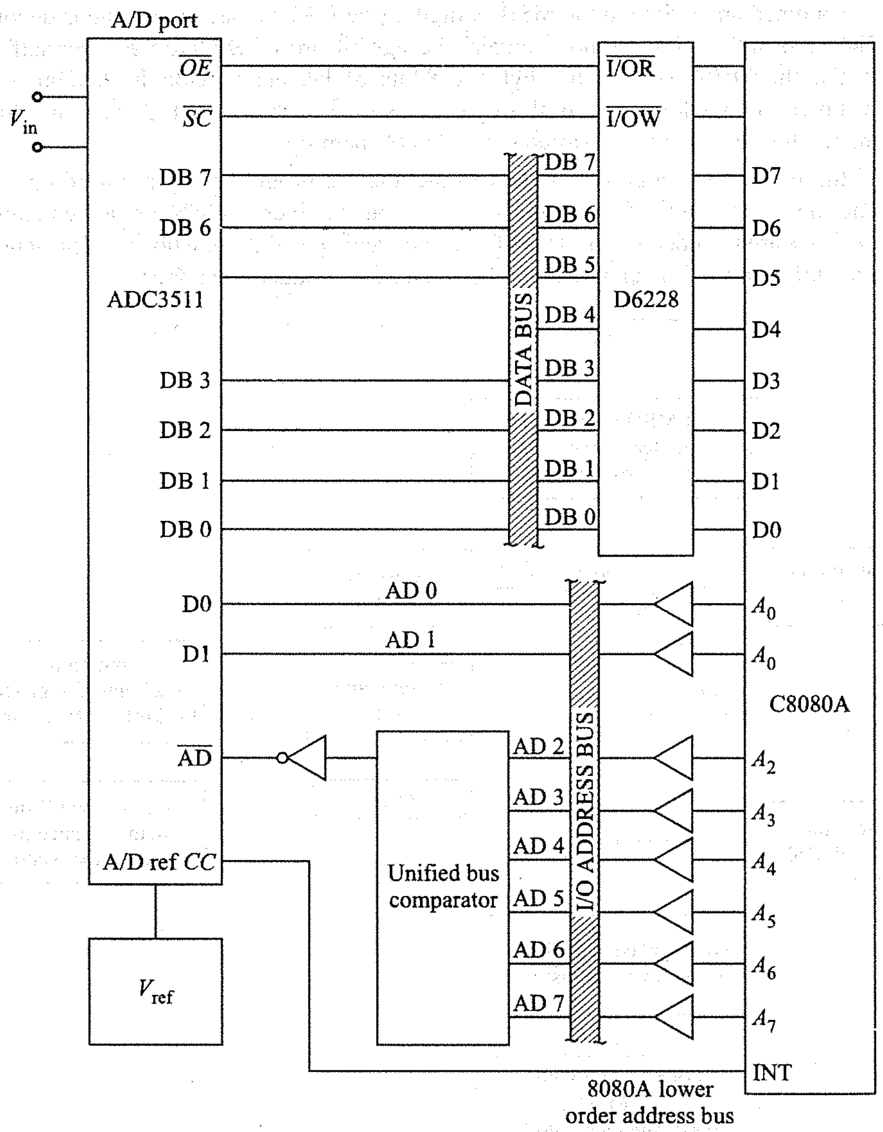
For both circuits, the reference voltage is derived from a National Semiconductor LM336, indicated by dotted lines. This is an active circuit that will provide 2.000 Vdc with a very low thermal drift of around 20 ppm/° C.

A complete circuit used to interface the ADC3511 with an 8080A microprocessor is shown in Fig. 15.23 below. Three-state bus drivers (DM80LS95) are used between the 3511 digital outputs and the microprocessor data bus, and the OR-gate–NOR-gate combination is used for control. The analog input is balanced with 51-kΩ resistors, and the 200-Ω resistor connected to SW_1 is chosen to equal the source resistance of the voltage reference; this will provide equal time constants for charging or discharging the 0.47-μF capacitor.



(a) Dual polarity A/D requires that inputs are isolated from the supply. Input range is ± 1.999 V

Fig. 15.23 **(From National Semiconductor Data Acquisition Handbook; continued on next page)**



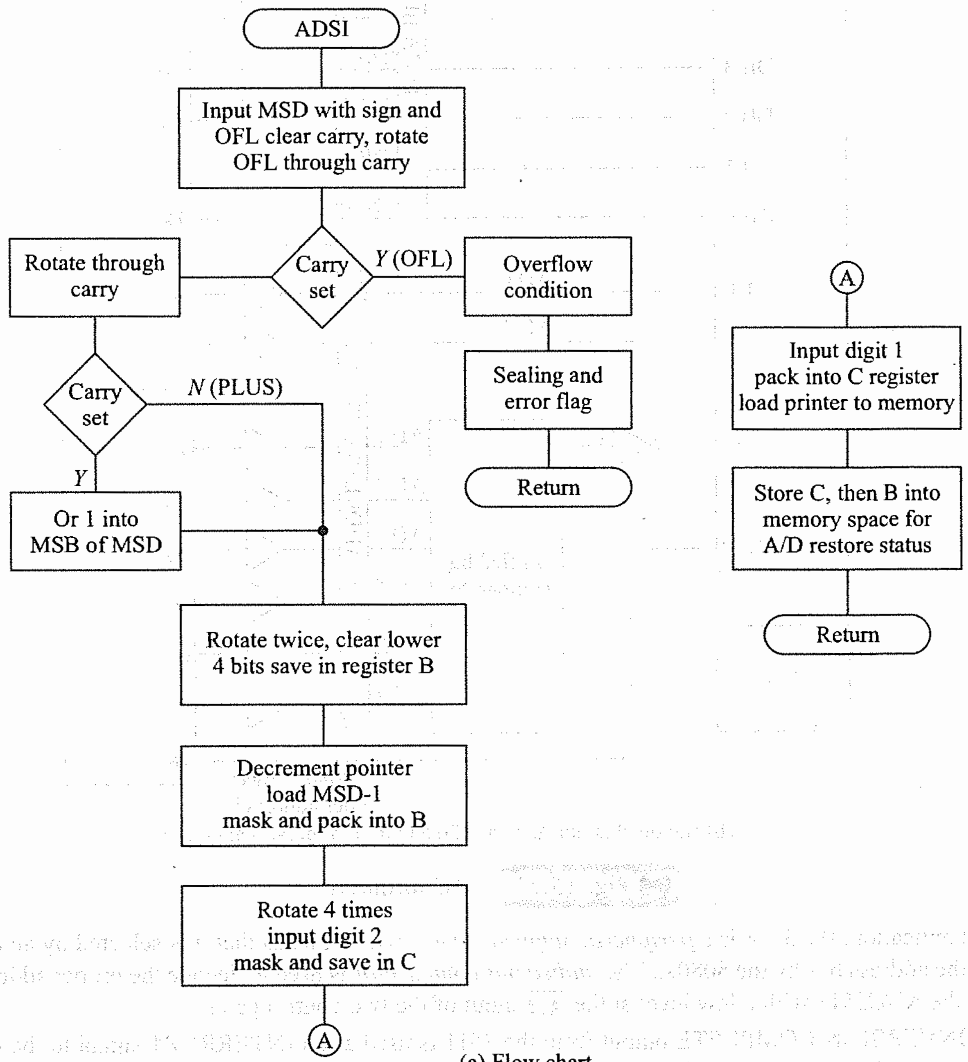
(b) Single channel A/D interface with peripheral mapped I/O

Fig. 15.23 **(Continued)**

In this application, the 3511 is a *peripheral mapped device*, which means that it is selected by an address placed on the address bus by the 8080A. The *unified bus comparator* is used to decode the proper address bits and select the ADC3511 with a low level at the $\overline{AD}$ input of the two control gates.

The CONVERSION COMPLETE output from the 3511 is used as an INTERRUPT signal to the 8080A, telling it that a digitized value is available to be read into the microprocessor. The receipt of an INTERRUPT signal causes the 8080A to read in the MSD (4 bits), the overflow (OFL), and the SIGN. If an overflow condition exists (OFL is high), an error signal is generated and the 8080A returns to its prior duties. Otherwise, the SIGN bit is examined and stored in the MSB of digit 4 (the LSD); a negative value is denoted by a 1 in this position. The 4 bits of the LSD, that now contains the sign bit, are shifted into the upper half of the 8080A data byte. (Note that the 8080A works with 1 byte, i.e. 8 bits, of data at a time on the data bus.) The 4 bits of digit 3 are then shifted into the lower half of this byte. In a similar fashion, digits 2 and 1 are shifted into the second byte, and the four digits are now stored in the 8080A memory.

It is beyond the intent and scope of this text to include the programming required on the 8080A to interface with the ADC3511, but the flow chart and service routine given in the National Semiconductor *Data Acquisition Handbook* are included in Fig. 15.24 for the convenience of those who might presently utilize the circuit. Additional information is available in the National Semiconductor handbook.


ADSI
Input MSD with sign and OFL clear carry, rotate OFL through carry
Carry set
Y (OFL)
Overflow condition
Rotate through carry
Carry set
N (PLUS)
Y
Or 1 into MSB of MSD
Sealing and error flag
Return
Rotate twice, clear lower 4 bits save in register B
Decrement pointer load MSD-1 mask and pack into B
Rotate 4 times input digit 2 mask and save in C
A
A
Input digit 1 pack into C register load printer to memory
Store C, then B into memory space for A/D restore status
Return

(a) Flow chart

Fig. 15.24 (From National Semiconductor Data Acquisition Handbook; continued on next page)

```
Label   Opcode  Operand   Comment                  Label  Opcode  Operand    Comment
ADIS:   PUSH    PSW       :A/D interrupt                  IN      ADD 2      :delay
                           service                        RAL                :rotate
        PUSH    H         :save                           RAL                :into
        PUSH    B         current status                  RAL                :upper
        IN      ADD 4     :input A/D digit 4              RAL                :4 bits
        IN      ADD 4     :delay                          ANI     FO         :mask lower bits
        ORA               :RESET carry                    MOV     C, A       :save in C
        RAL               :rotate OFL through             IN      ADD 1      :in digit 1
                          carry                           IN      ADD 1      :delay
        JC      OFL       :overflow condition             ANI     OF         :mask upper bits
        RAL               :rotate sign through            OR      C          :pack
                          carry                           MOV     C, A       :save in C
        JC      PLUS      :positive input                 LXI     H, ADMS    :load printer to A/D
        ORI     20H       :OR 1 into MSB                                      memory space
                           negative input
PLUS:   RAL               :shift                          MOV     M, C       :save C in memory
        RAL               :into position                  INX     H          :point next
        ANI     FO        :make lower bits                MOV     M, B       :save B in memory
        MOV     BA        :save in B                      OUT     ADD 1      :start new
        IN      ADD 3     :input digit 3                                      conversion
        IN      ADD 3     :delay                          POP     B          :restore
        ANI     OF        :mask higher bits               POP     H          :previous
        OR      B         :pack into B                    POP     PSW        :status
        MOV     B, A      :save in B                      EI                 :ENABLE interrupts
        IN      ADD 2     :input digit 2                  RET                :return to main program
```

(b) Routine 1, single channel interrupt service routine

Fig. 15.24 (Continued)

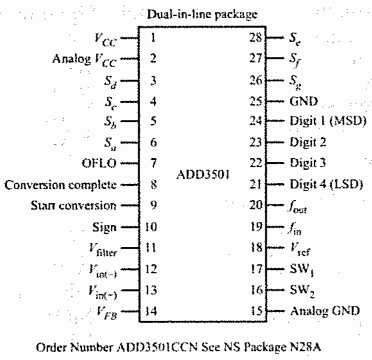
Example 15.11 Explain why it is acceptable to place the sign bit of a voltage digitized by the ADC3511 (or 3711) in the MSB of the MSD.

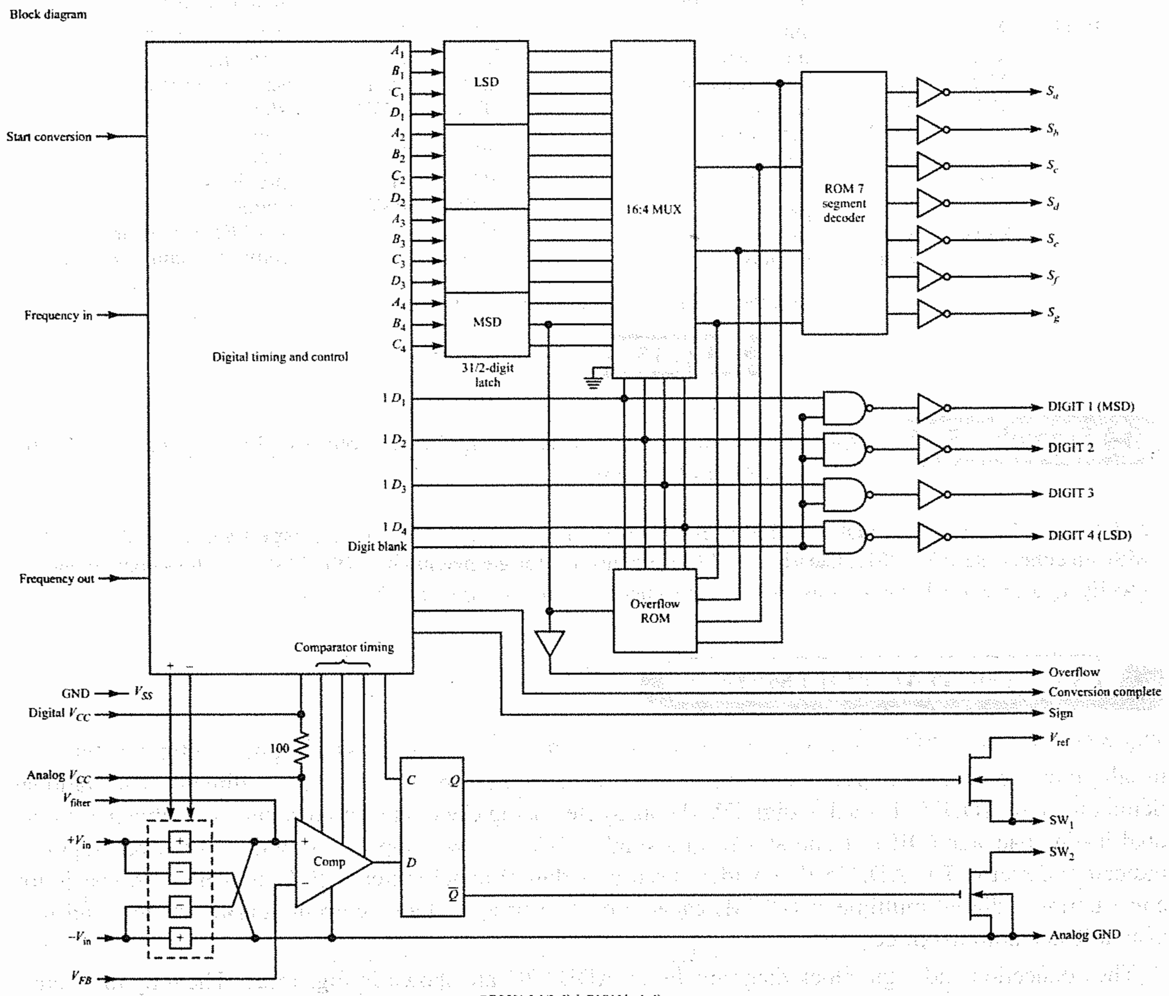
Solution The full-scale count for the 3511 is 1999 and for the 3711, is 3999. So, the largest value possible for the MSD in either case is 3 = 0011. Clearly the MSB is not needed for the magnitude of the MSD. It is thus convenient to specify a positive number when this bit is a 0 and a negative number when this bit is a 1.

15.6 DIGITAL VOLTMETERS

The ADC3511 (or 3711) discussed in the previous section can be used as a digital voltmeter, but it is usually more convenient to have a circuit that will drive seven-segment LED displays directly. The National Semiconductor ADD3501 is a $3\frac{1}{2}$-digit DVM constructed using CMOS technology and available in a single dual in-line package (DIP). It operates from a single +5-Vdc power supply and will drive seven-segment indicators directly. The ADD3501 is widely used as a digital panel meter (DPM) as well as the basis for constructing a digital multimeter (DMM) capable of measuring voltage, current, and resistance, and it is available at a nominal price.

The connection and logic block diagrams for an ADD3501 are shown in Fig. 15.25. The only difference between this device and the ADC3511 are the outputs. There are seven segment outputs , S_a, S_b, … , S_g, and the four digit outputs, DIGIT 1, … , DIGIT 4. These outputs are fully multiplexed and are designed to drive a





ADD3501 3 1/2-digit DVN block diagram

Fig. 15.25 National Semiconductor ADC3501

common-cathode-type LED display directly. All the other inputs and controls are identical to the previously discussed ADC3511. The 3501 has a full-scale count of 1999 for a full-scale analog input voltage of +2.00 Vdc. A resolution of 1 bit thus corresponds to 1 mV of input voltage.

The circuit shown in Fig. 15.26 on next page shows how to use an ADD3501 as a digital voltmeter that has a full-scale analog input voltage of +2.00 Vdc. The LM309 is a voltage regulator used to reduce jitter problems caused by switching. The NSB5388 is a $3\frac{1}{2}$-digit 0.5-in common-cathode LED display. The LM336 is an active circuit which is used to provide the 2.00-Vdc reference voltage. When using this configuration, it is important to keep all ground leads connected to a single, central point as shown in Fig. 15.26, and care must be taken to prevent high currents from flowing in the analog V_{CC} and ground wires. National Semiconductor has carefully designed the circuit to synchronize the multiplexing and the A/D conversion operations in an effort to eliminate switching noise due to power supply transients.

Example 15.12 What is the purpose of the 7.5-kΩ resistor and the 250-pF capacitor connected to pins 19 and 20 of the ADD3501 in Fig. 15.26?

Solution These two components establish the internal oscillator frequency used as the clock frequency in the converter according to the relationship $f_i = 0.6/RC$. In this case, $f_i = 320$ kHz.

The DVM in Fig. 15.27 on page 589 is modified slightly in order to accommodate analog input voltages of either polarity, and also of different magnitudes. Power for the circuit is obtained from the 115-Vac power line through an isolation transformer, and the analog input is now applied at $V_{\text{in}}(+)$ and $V_{\text{in}}(-)$.

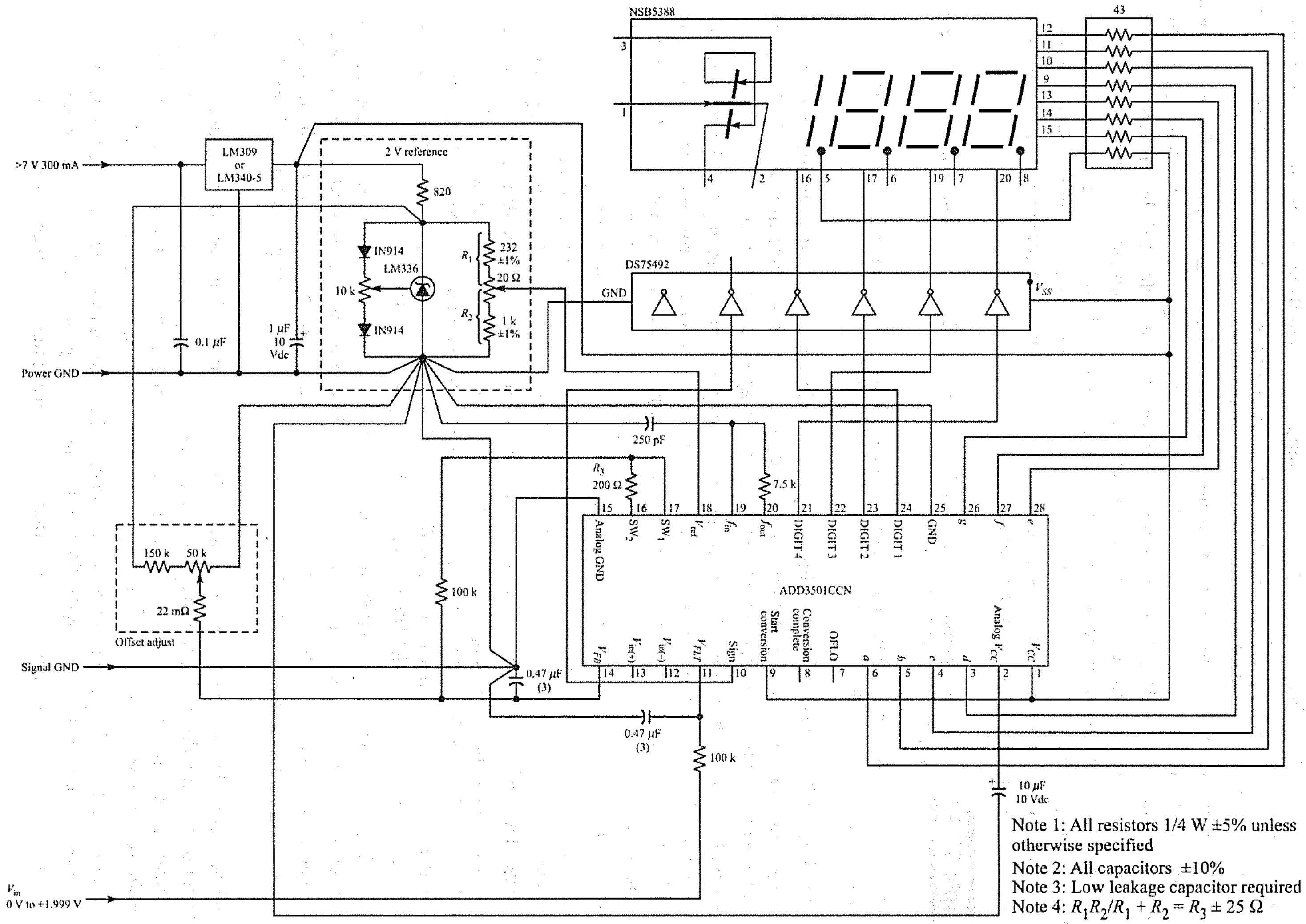
Scaling the analog input voltage for different ranges is accomplished by changing the feedback resistor between SW_1 on pin 17 and V_{FB} on pin 14, or using a simple resistance divider across the analog input. First look at the 2.00-Vdc range, since this is the normal full-scale range for the 3501. In this position, the range switch connects 100 kΩ as the feedback resistor, and the analog input goes directly to pin 13 [V_{i+}]. Also notice that the decimal point is between the 1 and the 8, giving 1.999 Vdc as a full-scale reading.

On the 0.2-Vdc scale, the range switch still applies the analog input voltage directly to pin 13. but the reference voltage at SW_1 is reduced by a factor of 10 by a resistive voltage divider before being used as a feedback voltage. The resistive divider is composed of a 90-kΩ resistor R_1, and a 10-kΩ resistor R_2. The voltage developed at the node connecting these two resistors is 0.1 V_{ref}, and so the full-scale voltage is also reduced by a factor of 10. The 90-kΩ resistor R_3 is used to keep the charging time constant essentially the same on all ranges. The time constant is given as $RC = 100\ \text{k}\Omega \times 0.47\ \mu\text{F}$. Notice that the decimal-point position has moved to pin 7 on the NSB5388 to give a full-scale reading of 199.9 mV.

On the 200-Vdc position, the range switch puts back the original feedback resistor, but the analog input voltage is reduced by a factor of 100 with a resistive voltage divider composed of 9.9-MΩ and a 100-kΩ resistor. The analog input to the 3501 is thus still 2.00 Vdc full scale even though the actual input signal is 200 Vdc full scale. The decimal point will be placed on pin 7 of the NSB5388 to give a full-scale reading of 199.9 Vdc.

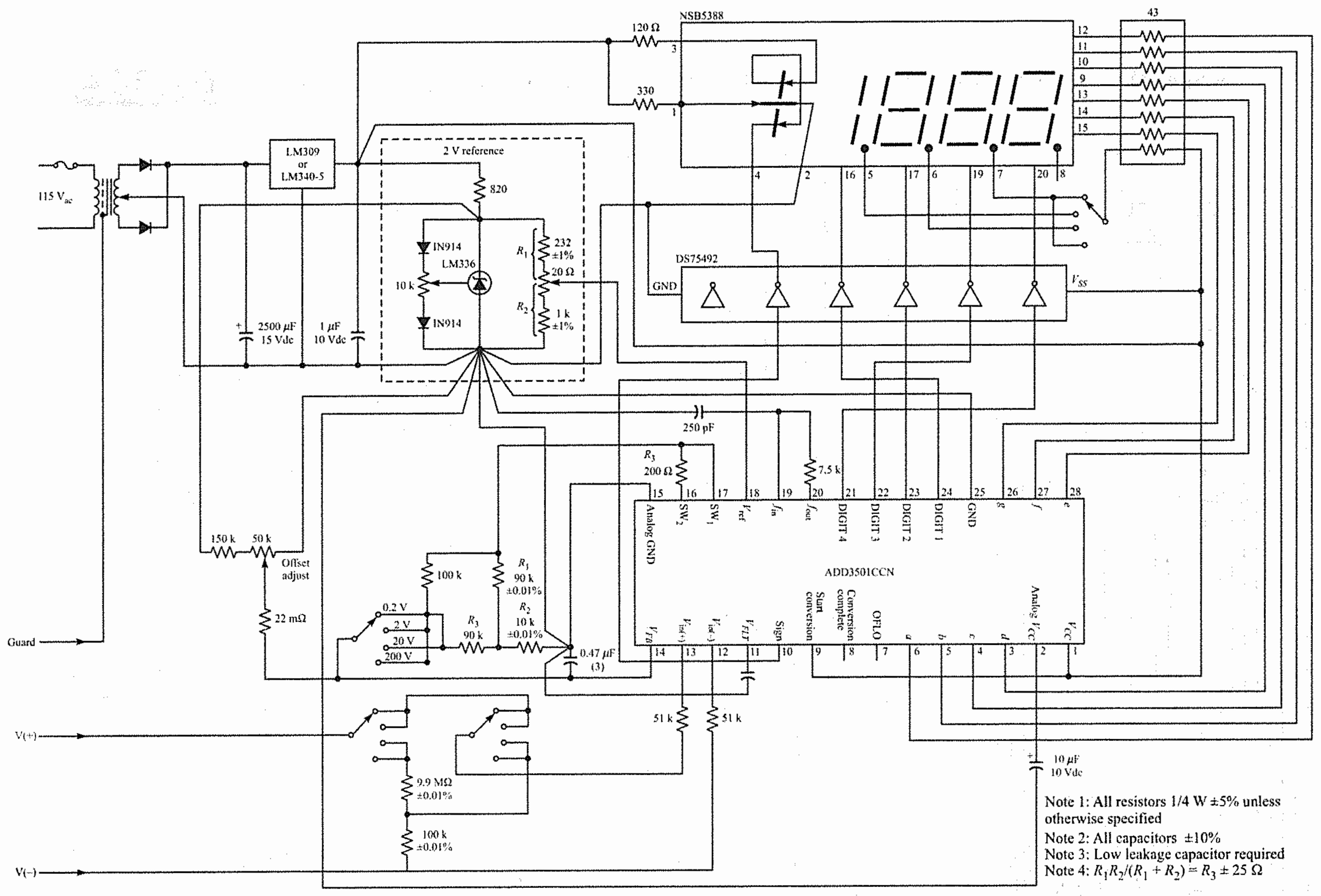
On the 20-Vdc full-scale position, the range switch still uses the input voltage divider to reduce the input signal by a factor of 100, but the feedback resistor is also used to effectively increase the full scale by a factor of 10. The net result is that the 3501 will count full scale when the analog input voltage is 20 Vdc. Notice that the decimal point is now applied to pin 6 of the NSB5388 to give a full-scale reading of 19.99 Vdc.

The circuit shown in Fig. 15.28 on page 590 is a complete DMM taken from the National Semiconductor *Data Acquisition Handbook*. It utilizes the ADD3501 and is capable of measuring both dc and ac currents and voltages as well as resistances. The ranges and accuracies of the instrument are given in Fig. 15.29.



Note 1: All resistors 1/4 W ±5% unless otherwise specified
Note 2: All capacitors ±10%
Note 3: Low leakage capacitor required
Note 4: $R_1R_2/R_1 + R_2 = R_3 \pm 25\ \Omega$

Fig. 15.26 $3\frac{1}{2}$ digit DPM, +1.999 Vdc full scale, (National Semiconductor)



Note 1: All resistors 1/4 W ±5% unless otherwise specified
Note 2: All capacitors ±10%
Note 3: Low leakage capacitor required
Note 4: $R_1R_2/(R_1 + R_2) = R_3 \pm 25\ \Omega$

Fig. 15.27 $3\frac{1}{2}$ digit DVM, Four decade; ±0.2, +0.2, ±20, and ±200 Vdc (National Semiconductor)

Technical specifications

DC volts	< ± 1% accuracy
ranges	2 V, 20 V, 200 V, 2 kV
input impedance	2 V range, > 10 MΩ
	20 V to 2 kV range, 10 MΩ
AC RMS volts	< ± 1% accuracy
ranges	2 V, 20 V, 200 V, 2 kV
	(40 to 5 kHz sinewave)
DC amps	< ± 1% accuracy
ranges	200 μA, 2 mA, 20 mA, 200mA, 2A
Ohms	< ± 1% accuracy
ranges	200 Ω, 2kΩ, 20kΩ, 200kΩ, 2 MΩ

Note 1: All V_{cc} connections should use a single V_{cc} point and all ground/analog ground connection should use a single ground/analog ground point.

Note 2: All resistors are 1/4 watt unless otherwise specified

Note 3: All capacitors are ±10%

Note 4: All op amps have a 0.1 μF capacitor connected across the V+ and V– supplier

Note 5: All diodes are 1N914

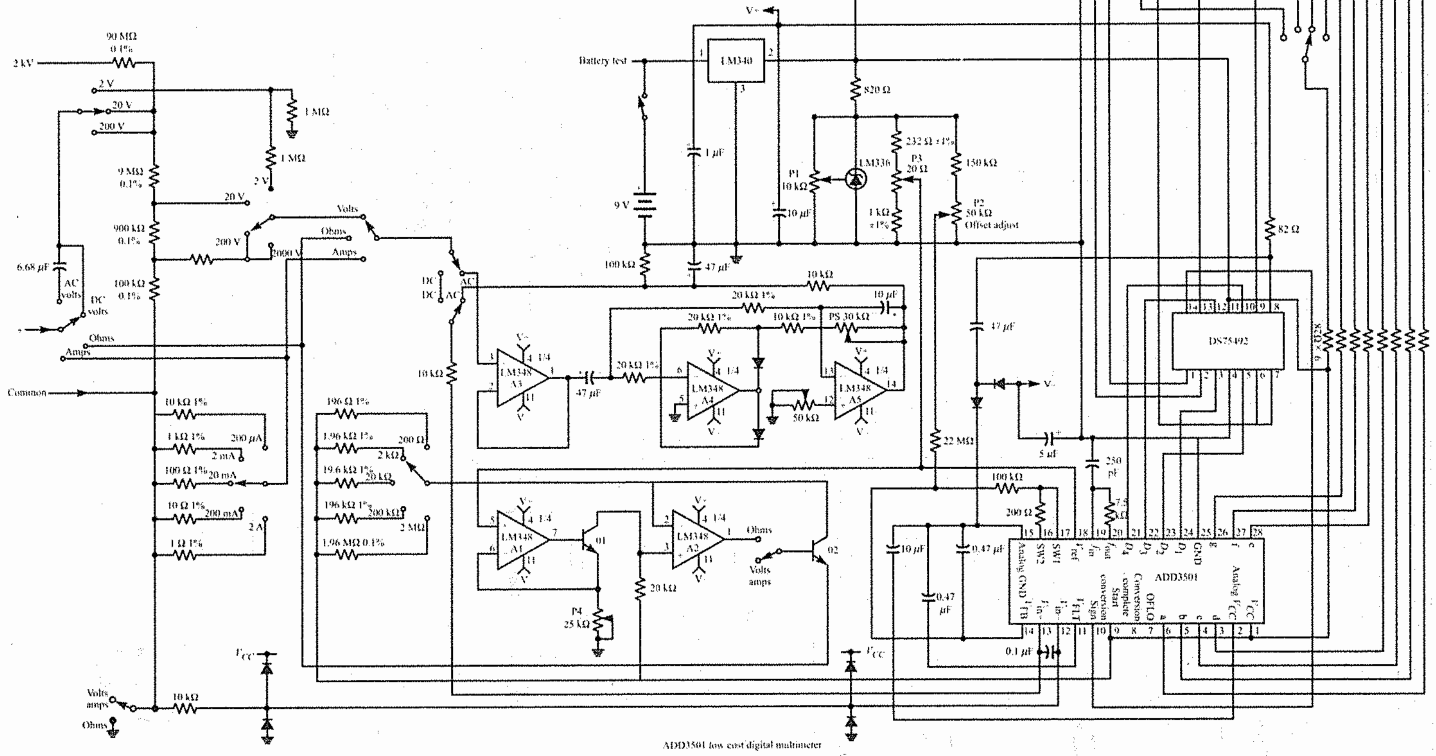


Fig. 15.28 A low-cost DMM using the ADD 3501 (National Semiconductor Data Acquisition Handbook)

The different dc and ac voltage ranges are accommodated by a resistive voltage divider at the analog input. Alternating-current voltages are measured by using the three operational amplifiers A_3, A_4, and A_5 to develop a dc voltage that is proportional to the root-mean-square (RMS) value of the ac input voltage.

Measurement mode	Range					Frequency response	Accuracy	Overrange display
	0.2	2.0	20	200	2000			
DC volts	–	V	V	V	V	–	≤ 1% FS	± OFLO
AC volts	–	V_{RMS}	V_{RMS}	V_{RMS}	V_{RMS}	40 Hz to 5 kHz	≤ 1% FS	+ OFLO
DC amps	mA	mA	mA	mA	mA	–	≤ 1% FS	± OFLO
AC amps	mARMS	mARMS	mARMS	mARMS	mARMS	40 Hz to 5 kHz	≤ 1% FS	+ OFLO
Ohms	kΩ	kΩ	kΩ	kΩ	kΩ	–	≤ 1% FS	+ OFLO

Fig. 15.29 **Performance of the DMM in Fig. 15.28**

A series of current-sensing resistors are used to measure either dc or ac current. The current to be measured is passed through one of the sensing resistors, and the DMM digitizes the voltage developed across the resistor.

The DMM measures resistance by applying a known current from an internal current source (operational amplifiers A_1 and A_2) to the unknown resistance and then digitizing the resulting voltage developed.

For those interested in pursuing this subject, complete details for the construction and calibration of this DMM are given in the *National Semiconductor Data Acquisition Handbook.*

SUMMARY

The primary objective of this chapter is to demonstrate the use of many of the most fundamental principles discussed throughout the text by considering some of the more common digital circuit configurations encountered in industry. Multiplexing of LED displays, time and frequency measurement, and use of digital voltmeters of all types are widely used throughout industry. Although our coverage is by no means comprehensive, it will serve as an excellent introduction to industrial practices.

The problems at the end of this chapter will also provide a good transition into industry. They are in general longer than previously assigned problems. All the necessary information required to work a given problem may not be given—this is intentional since it will require you to seek information from industrial data sheets. However, the problems are more of a design nature, and usually deal with a practical, functional circuit that can be used to accomplish a given task; as such, they are much more interesting and satisfying to solve.

PROBLEMS

In order to solve some of these problems, you may have to consult product data sheets that are not included in this text. It is intended that you discover a source for such information.

15.1 Pick one of the solutions suggested in Example 15.3 and do a detailed design, including part numbers and pin numbers.

15.2 Design a four-decimal-digit multiplexed display like the one in Fig. 15.7, but use common-cathode-type LEDs. Use a basic circuit like the one in Fig. 15.2, but you will now need to generate DIGIT waveforms that have positive pulses.

15.3 How often is each digit in Fig. 15.7 serviced, and for what period of time is it illuminated? Extinguished?

15.4 Design a multiplexed display like the one in Fig. 15.8 having eight decimal digits. Use a three-flip-flop multiplexing counter and four 74151 multiplexers.

15.5 Specify a ROM that could be used in place of the 7447 in Fig. 15.8. Draw a circuit, showing exactly how to connect it.

15.6 Design a four-decimal-digit display using 54/74143 and common-anode LEDs.

15.7 Using Fig. 15.12 as a pattern, design a four-digit frequency counter using 54/74143s and 54/74160s. Use a 1.0-MHz clock, and provide 0.1-, 1.0-, and 10.0-s gates. Specify the frequency range for each gate.

15.8 Following Fig. 15.12 as a guide, design a four-digit frequency counter using National Semiconductor MM74C925.

15.9 Design a circuit to measure "elapsed time" between two events in time—for instance, the time difference between a pulse occurring on one signal followed by a pulse occurring on another signal. Use as much of Fig. 15.14 as possible, but consider using a set-reset flip-flop in conjunction with the two input signals.

15.10 Combine the circuits in Figs. 15.12 and 15.14 into a single instrument. Use a 1.0-MHz clock and seven decade counters, define the scales and readouts carefully.

15.11 What is the internal clock frequency of the ADC0804 in Fig. 12.29 if the capacitor C is changed to 100 pF?

15.12 In stand-alone operation, how often does the ADC0804 do an A/D conversion?

15.13 Assuming that V_{CC} = +5.0 Vdc, determine the digital outputs of an ADC0804 for analog inputs of:

a. 1.25 V, b. 1.0 V

c. 4.4 V

15.14 Assuming that V_{CC} = +5.0 Vdc, in Fig. 12.29, determine the analog input voltages that will produce a digital output of:

a. 1000 1100 b. 25H

c. 0001 1000?

15.15 The ADC0804 in Fig. 12.29 is to be used with an analog signal that varies between 0.0 and +3.3 V. Determine a new value for V_{ref}.

15.16 The ADC0804 in Fig. 15.16b is to be used with an analog signal that varies between +2.2 and +3.3 V. Determine a new value for V_{ref} and V_{i-}.

15.17 Use the ADC0804 and design a stand-alone circuit to digitize an analog voltage that ranges between +0.25 and +5.0 V.

15.18 Use the ADC0804 and design a stand-alone circuit to digitize an analog voltage that ranges between −2.5 and +2.5 V.

15.19 Design a resistive voltage divider to use with the ADC3511 such that it will digitize an analog input voltage of 20 Vdc as full-scale voltage input. What is the resolution in millivolts for this design?

15.20 Design a voltage divider such that the DVM in Fig. 15.27 will measure full-scale voltages of 2.0, 20.0 and 200.0 Vdc without changing the feedback resistor. Leave the feedback resistor at 100 kΩ. Draw the complete design. Is it possible to achieve a full scale of 0.2 Vdc for this circuit without changing the feedback resistor?

Answers to Self-tests

1. R and C are needed to set the internal clock frequency.
2. Depressing the START button begins the A/D conversion process.
3. C3H
4. Span is the range of input voltage. OFFSET is the lowest value of analog input voltage.

A Simple Computer Design

16

OBJECTIVES

- Determine hardware requirement in design of a simple computer
- Discuss use of Register Transfer Language in computer design
- Design control unit of a simple computer
- Discuss how to program the simple computer in solving various problems

In this chapter, we demonstrate how the knowledge that you gathered in this book can take you to the next higher level, where you can start designing a digital computer. A digital computer is capable of computation and taking decision based on binary coded instructions stored inside it. The central processing unit (CPU), also known as the brain of the computer sequentially fetches these instructions, decodes it and then executes it by performing some action through available hardware. In this chapter, we'll design a simple computer, which has a limited instruction set but is capable enough to solve variety of arithmetic and logic problems. The technique you learn in developing this simple machine will be useful when you go for a full-fledged computer design in some higher-level courses.

We begin the chapter by defining a small problem, which our simple computer should be able to solve. Next, we spell out different hardware components required as building blocks. We'll also discuss a simple hardware operation description language, called Register Transfer Language (RTL) useful for state machine design. Through RTL we'll describe all the operations of our simple computer. Then we'll design the control unit that will coordinate all these operations. Finally, we will discuss how to program this simple computer to solve the problem we started with and many other arithmetic and logic problems.

16.1 BUILDING BLOCKS

In Section 1.6 of Chapter 1, we have broadly seen the kind of components required for designing a computer. In this chapter, we address how to design central processing unit of a simple computer that interacts with a small memory module. Before we proceed further let us define a problem that our computer is supposed to solve. This is not the only problem it can handle. Depending on how we program it, we will be able to solve different arithmetic and logic problems and that is shown towards the end of this chapter through examples. The purpose of defining a problem is to choose specific hardware components that will serve as building block of our simple computer.

The Problem

Add 10 numbers stored in consecutive locations of memory. Subtract from this total a number, stored in 11th location of memory. Multiply this result with 2 and store it in 12th location of memory. All the numbers brought from memory lie between 0 and 9.

Memory

Since, the problem says the numbers or data to be fetched from memory and we also know that programs, i.e. binary coded instructions are also stored in memory, let us divide the memory used in our computer in two parts. One part stores the program or series of instructions the computer executes sequentially and this is known as *program memory*. The other part houses data that program uses and is also used for storing result. This is called *data memory*. From the given problem we find, we need 12 memory locations for data storage. We expect our computer won't need more than 20 instructions to complete the given task hence, a memory with 32 locations (integer power of 2) can be selected for our computer.

Now we try to decide how many bits of information we store in each address location. Usually, bits in memory locations are stored in multiple of 8 called *byte*. Let's see if our job can be done with 8 bits. Each memory location stores data between 0 and 9 on which program operates and thus require only 4 bits. The final result at most can be $10 \times 9 \times 2 = 180$ which requires 8-bit for representation. So the data memory can be of 8-bits with which we can represent decimal number up to $2^8 - 1 = 255$.

Let's now see the requirement of program memory. There, in each location, certain number of bits are allocated that defines the instruction to be executed. This is called operation code or in short, *opcode*. The rest of the bits can be used for referring the memory location from which data is to be brought or stored, if required by the instruction. Since, 32 memory locations require $\log_2 32 = 5$ bits for memory referencing we'll have $8 - 5 = 3$ bits for opcode specification giving $2^3 = 8$ different opcodes (Fig. 16.1). We'll see that 8 instructions are sufficient for the given kind of task in our limited ability computer. Hence, one important hardware component of our computer gets decided. The memory to be used is of size 32×8.

The above mode of addressing memory for data is called *direct addressing*. If the address mentioned in the instruction contains address from which actual data is to be brought it is called *indirect addressing*. If after opcode, in place of address actual data is made available, it is called *immediate addressing*. Note that, in immediate addressing data cannot be more than 5 bits as 3 bits gets used in opcode. Also note that the instruction like this is called single byte instruction. If an instruction requires 2 bytes to be fetched from program memory it is called 2-

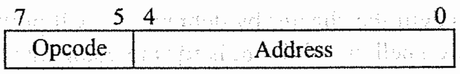


Fig. 16.1 Three Most Significant Bits (MSB) are opcode and five Least Significant Bits (LSB) are address

byte instruction. Obviously, in 2-byte instruction number of opcodes or memory addressing capability can be more than a single byte instruction.

Register Array

The computer needs a set of registers to perform its operation. Let us define them and assign task to each one of them for our simple computer. Note that, we are using a 32 × 8 memory module.

Memory Address Register (*MAR*) is a 5-bit register that stores the address of the memory location referred in a particular instruction. The output of this is fed to a 5-to-32 address decoder. Each output of the decoder points to a location in the memory. All memory referenced instruction loads memory address in *MAR*.

Memory Data Register (*MDR*) is an 8-bit register that stores the memory output when a memory read operation is performed. During memory write operation it stores the value that gets written to the memory. Thus it can also be called a memory buffer. In arithmetic or logic operation when more than one operand is required by ALU, one operand in our simple machine comes from *MDR*.

Program Counter (*PC*) is a 5-bit counter that stores the address of the memory location from which next instruction is fetched. At power on, our machine *PC* is reset so that its content is all zero. Thus location 00000 has to be a part of program memory and this is also the starting address from which program execution begins. Since, in our simple machine all the instructions are single byte instruction, every time an opcode is fetched we'll increment *PC* by one, and thus *PC* will point to location of the next opcode.

Instruction Register (*IR*) is a 3-bit register, which retains the opcode till it is properly executed in one or more clock cycles. Since all memory read and write operations are done through *MDR*, after an instruction is read from memory, 3 MSB that contains the opcode are transferred to *IR*.

Accumulator (*ACC*) is a multi-purpose register that always stores one operand of an arithmetic or logic operation. The result of this operation, i.e. ALU output is also stored in *ACC*. Functions like shifting of bits to left or right are also carried on *ACC*. Thus, in our simple computer *ACC* is a shift register with parallel load facility.

Timing Counter (*TC*) is a synchronous parallel load counter that stores and updates the timing information. The timing counter output is decoded to generate different timing signal, which in turn triggers different events in execution of an instruction. The counter is reset synchronously with clock once an instruction is fully executed. If an instruction is conceived as a *macro operation* then series of sequential steps necessary to carry out the instruction in the computer is called *micro operations*. In our simple computer, we are not expecting more than 8 micro operations for any macro-operations and hence a 3-bit counter is sufficient. Later if we see, we need more than 8 micro operations we'll change it to a 4-bit counter. Note that, a *master clock* (also called system clock) to which all the state changes of the computer are synchronized, triggers this counter. Also note, *TC* has power on reset facility, i.e. when the computer is switched on it stores 000.

Start/Stop Flag (*S*) is a flip-flop which when set, stops execution of the program. This we do in our simple computer by inhibiting the master clock. Like program counter, this also has a power-on-reset facility so that when the computer is switched on the master clock is not inhibited.

Other Important Hardware

Arithmetic Logic Unit (ALU) is a versatile combinatorial circuit that can perform a large range of arithmetic and logic operations. Since the data is 8-bit long, we use an 8-bit ALU. The control input value decides the function ALU executes at a particular time. ALU can accept up to two operands at a time, one from *ACC* and the other from *MDR*. The ALU output is stored in *ACC*. If addition operation generates a carry output from ALU, that can be stored in a flip-flop, often called carry flag (*CY*). Since, in our problem numbers are small in magnitude the 8-bit ALU doesn't generate carry output and we don't need *CY* flag for our simple computer. Note that, ALU cannot perform multiply and division operation for which we use special hardware or some indirect technique.

Instruction Decoder (ID) is a 3-to-8 decoder, which takes input from *IR* and thereby decodes the opcode. In our simple computer there are 8 different opcodes, each one making one of the decoder output ($D_0, D_1, \ldots, D_7$) high. This in turn initiates specific micro operations necessary to execute that opcode in subsequent clock cycles.

Timing Sequence Decoder (TSD) is again a 3-to-8 decoder that takes input from *TC* and provides necessary timing information in the form of decoded output ($T_0, T_1, \ldots, T_7$) for a micro operation to be executed.

BUS is a group of wires that serve as a shared common path for data transfer of all the devices connected to it. With this, we do not need a separate device to device connection which increases the number of wiring specially when large number of devices are used in a system. Since, the largest group of binary data that is transferred in our computer is 8-bit, the bus used is an 8-bit bus.

BUS Selector (BS) is a multiplexer, which decides which one of all the connected devices is in transmission mode, i.e. has placed data in the BUS. Note that, if more than one device try to send data simultaneously, there will be a conflict producing erroneous result. However, in our computer we may allow more than one device connected to BUS to receive data from BUS. We'll see shortly that only *PC*, *ACC*, *MDR* and ALU want to transmit or place data on the BUS. *MAR* and *IR* only receive data and other hardware give control signal and don't do data transfer. Thus, BS has to select one of the four devices and uses eight (each one for one bit) 4-to-1 multiplexer type of device. We can also use tri-stated output for bus connection (Section 14.6 of Chapter 14) that will reduce the current loading on the device when it is not selected.

From this discussion we can draw the *data path* of our simple computer as shown in Fig. 16.2. Here, by data we mean address, opcode as well as operand and they move from/to memory, register, ALU, etc. Of course, we need another set of path to send control signal to various hardware to carry out microoperations. This is called *control path* and we'll design it when we define the instruction set for our computer.

Generally speaking, *address bus* is the group of wires that transfer address information, *data bus* is another group that transfers data and *control bus* transfers control information. Often, address information and data are transferred through a common bus and a control logic decides which is to be transferred and when. You might have noticed that in our simple computer design, we have used a common address and data bus. More about control bus will appear in Section 16.4 where we discuss the design of control unit.

Find in Fig. 16.2 the direction of arrow that shows the direction of data flow. Note that, *IR* and *MAR* can only receive data from BUS; *PC* can only send data by BUS; *ACC* and *MDR* can do both;

Memory data transfer takes place only via *MDR* and operands of ALU come from *ACC* and *MDR* and result is sent via BUS.

Example 16.1 In a particular configuration each memory location contains 16-bit data. In program memory, if 4 MSB contains opcode and rest contains address of memory locations give (a) Number of opcodes (b) Size of memory (c) Size of *PC*, *IR*, *ACC*, *MAR* and *MDR*.

Solution

(a) Number of opcodes = $2^4 = 16$ (Maximum)

(b) Number of address bits = $16 - 4 = 12$. No. of memory locations = $2^{12} = 2^2 \cdot 2^{10} = 4K$. So size of memory is $4K \times 16$.

(c) Size of *PC* and *MAR* = No. of address bits = 12. Size of *IR* = Size of opcode = 4. Size of *ACC* and *MDR* = No. of data bits = 16

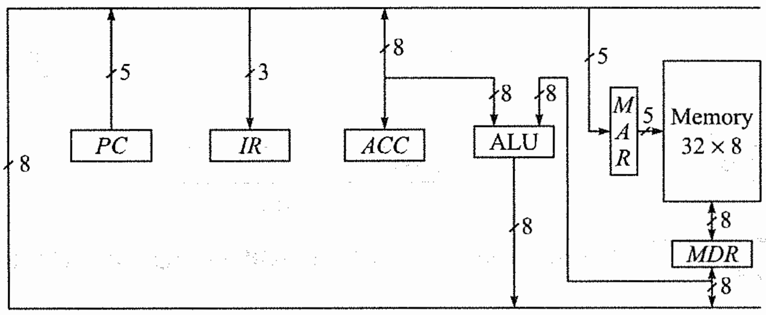


Fig. 16.2 Data path of the simple computer

SELF-TEST

1. What is the highest integer in decimal that we can store in 16-bit data field?
2. What is an opcode?
3. What is the function of program counter?
4. What is indirect addressing?

16.2 REGISTER TRANSFER LANGUAGE

Before we go for design of control path and the control unit as a whole we have to define macro operations and then we need to break up each macro operation in series of micro operations at register level. Register Transfer Language (RTL) gives a simple tool through which these micro operations can be expressed and then control unit can be designed from that. The basic structure of this language is

$$X : A \leftarrow B$$

This means, if condition X is TRUE, i.e. $X = 1$ then content of register B is transferred to register A. X can be a single logic variable or a logic expression like $xy \equiv x\&y$, $x + y \equiv x \mid y$, etc. In RTL we distinguish logic operation 'OR' from arithmetic operation 'addition' by assigning symbols '$\mid$' and '+' respectively. The logic AND is expressed by symbol '&'. However, if the '+' sign appears left to ':' in an RTL statement it means logical OR and '.' refers to logical AND. This is so because to left of ':' only logical operators can reside. Often AND, OR, NOT are expressed by '$\wedge$', '$\vee$', '~' respectively. Also note, this register transfer destroys the previous content of A but not that of B. Both the register A and B now have the same value. If register transfer takes via BUS

$$A \leftarrow B \equiv BUS \leftarrow B, \quad A \leftarrow BUS$$

Since, BUS is not a register but a group of wire this means B getting access to BUS through BUS selector (BS) and the whole event takes place in one clock cycle. Figure 16.3 pictorially depicts register transfer without and with BUS.

To write anything to memory, in our simple computer we have to place the address information in MAR and the data to be written in MDR. Thus, memory write operation in RTL is expressed as

$$X : \mathrm{M}[MAR] \leftarrow MDR$$

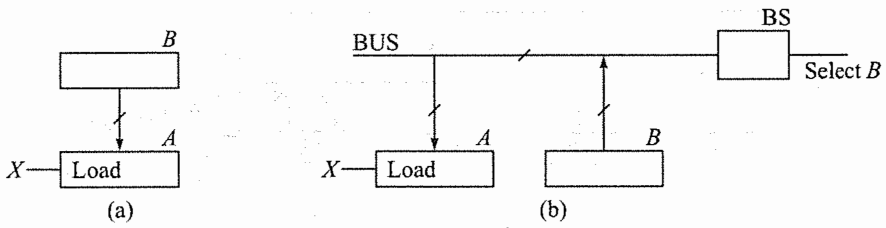


Fig. 16.3 Register transfer A → B: (a) without BUS, (b) with BUS

Similarly, memory read operation is also done through *MAR* and *MDR* and RTL expression is

$$X : MDR \leftarrow \mathrm{M}[MAR]$$

If certain bits of a register are to be addressed we use RTL as follows:

$$X : IR \leftarrow MDR[7{:}5]$$

The statement above refers to transfer of three most significant bits of *MDR* to *IR*, a 3-bit register when $X = 1$.

The arithmetic and logic operations of ALU that bring operands from *ACC* and *MDR* and store the result in *ACC* can be expressed in RTL in the following way

$X : ACC \leftarrow ACC \,\&\, MDR$	[logic AND]
$X : ACC \leftarrow ACC \mid MDR$	[logic OR]
$X : ACC \leftarrow ACC \oplus MDR$	[logic EX-OR]
$X : ACC \leftarrow ACC'$	[logic NOT]
$X : ACC \leftarrow ACC + MDR$	[arithmetic addition]
$X : ACC \leftarrow ACC - MDR$	[arithmetic subtraction]
$X : ACC \leftarrow ACC + 1$	[increment by 1]
etc.	

And finally if data is to be shifted in a register say by 1 bit to left we can write

$$X : ACC[7{:}1] \leftarrow ACC[6{:}0],\ ACC[0] \leftarrow 0$$

If such left shift occurs through carry the statement will be

$$X : ACC[7{:}1] \leftarrow ACC[6{:}0],\ ACC[0] \leftarrow CY$$

Normally, we come across these four kinds of micro operations namely (i) Inter-Register transfer (ii) Arithmetic operation (iii) Logic operation and (iv) Shift operation. Note that, left shift without carry can also be obtained by addition operation as shown in Example 6.14 of Chapter 6. Figure 16.4 shows how addition operation $A \leftarrow A + B$ takes place through ALU and BUS.

Note that, *TC* and *PC* can increment by 1 without taking help of ALU as they are designed to be parallel load up counters. For more complex processor unit where 2 byte, 3 byte instructions are possible we can have an adder unit accessible by *PC*.

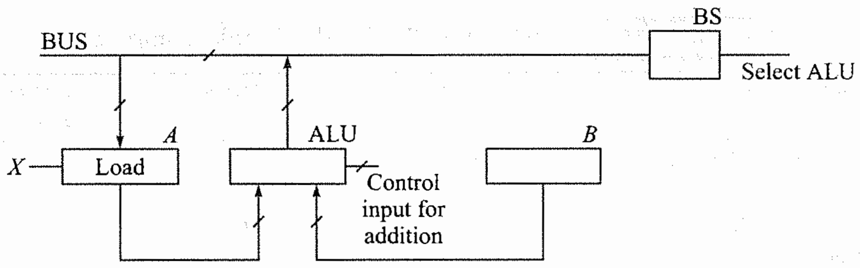


Fig. 16.4 **The micro operation $A \rightarrow A + B$**

Example 16.2 Explain what the following RTL statements perform

$$T_1 : MDR \leftarrow ACC$$
$$T_2 : ACC \leftarrow ACC'$$
$$T_3 : ACC \leftarrow ACC \,\&\, MDR$$

Solution The first statement says if $T_1 = 1$, content of ACC is transferred to MDR. The second statement says if $T_2 = 1$, content of ACC is complemented. The third statement says if $T_3 = 1$, bit-wise AND operation is performed on ACC and MDR and the result is stored in ACC. Since content of MDR and ACC were complement of one another before this statement is executed, by AND operation all the bits of ACC become zero, i.e. ACC is reset by these three statements irrespective of its initial content.

Note that, T_1, T_2 and T_3 can be output of a timing sequencer, which become active one after another in consecutive clock cycles. This way, ACC can be cleared in three clock cycles by above RTL statements.

SELF-TEST

5. What is RTL?
6. What is to be changed in Fig. 16.4 to perform $A \leftarrow A \,\&\, B$?

16.3 EXECUTION OF INSTRUCTIONS, MACRO AND MICRO OPERATIONS

In a computer, *execution of instructions* is carried through macro operations which again can be subdivided into micro operations. In this section, we first define the macro operations that we want to be executed in the computer we are designing. Next, we'll discuss micro operations necessary to execute each macro operation and it will be expressed through RTL. Remember that we have assigned only 3-bits as opcode and hence we can define $2^3 = 8$ instructions or macro operations with them. Table 16.1 lists all the instructions, corresponding mnemonics (easy to remember short forms), opcodes and 3-to-8 decoder (ID) output when IR is loaded with this opcode.

Table 16.1 Instruction Set for the Simple Computer

Macro operation performed	*Instruction mnemonic*	*Opcode*	*Instruction decoder (ID) output activated*
Load data from a specified memory location to *ACC*	LDA	0 0 0	D_0
Store *ACC* data in a specified memory location	STA	0 0 1	D_1
Halts execution of the program	HLT	0 1 0	D_2
Perform bitwise AND operation of *ACC* with data of a specified memory location and store result in *ACC*	AND	0 1 1	D_3
Perform bitwise NOT operation of *ACC*	NOT	1 0 0	D_4
Perform 1-bit left shift of *ACC* with $ACC[0] \leftarrow 0$	SHL	1 0 1	D_5
Perform addition operation of *ACC* with data of a specified memory location and store result in *ACC*	ADD	1 1 0	D_6
Subtract from *ACC*, data of a specified memory location and store result in *ACC*	SUB	1 1 1	D_7

Instruction Cycles

To carry out each instruction or macro operation the computer has to go through three distinct phases or cycles. In fetch cycle it brings the instruction or opcode from the program memory. In decoding phase it decodes the opcode and finally the execution is done in execute cycle. These cycles together known as *instruction cycle* are again repeated for next instruction. It is understandable that fetch and decode phase will be same for all instructions in our simple computer as we have only single byte directly addressed instructions. However, the execution cycle will be different for different instructions depending on the tasks the instruction wants to perform.

Fetch Cycle

An instruction cycle begins with *fetch cycle* when *TC* is reset to 0. Then, only T_0 output of TSD will be high and rest low. As told before *PC* contains the address of the location from which next instruction is to be fetched, content of *PC* is loaded into *MAR* in T_0.

At the next trigger of master clock *TC* is incremented by 1 so that T_1 becomes high and other outputs of TSD are low. In this clock cycle, content of memory from location specified by *MAR* (through 5-to-32 address decoder attached to memory) is loaded to *MDR*. *PC* now can be incremented to point to address of next location in program memory, which stores next instruction.

In the next clock cycle *TC* generates $T_2 = 1$ when opcode from 3 MSB of *MDR* is transferred to *IR* and 5 LSB to *MAR*. Content of *IR* is used for decoding opcode in decode phase. Content of *MAR* will be useful in execute phase if the opcode makes some memory reference, the address of which remain available at *MAR*. In RTL the above operations can be represented as

$$T_0 : MAR \leftarrow PC$$
$$T_1 : MDR \leftarrow \mathrm{M}[MAR],\ \ PC \leftarrow PC+1$$
$$T_2 : IR \leftarrow MDR[7:5],\ \ MAR \leftarrow MDR[4:0]$$

Decode Cycle

In *decode cycle* we decode the opcode fetched from program memory. Since at T_2, register *IR* is loaded with opcode and 3-to-8 decoder (ID) that decodes the opcode is a combinatorial circuit, we finish decoding in T_2 itself. In RTL we express it as

$$T_2 : D_0 \ldots D_7 \leftarrow \text{DECODE } (IR)$$

Often, the 3rd statement of previously mentioned fetch cycle that loads *IR* with new opcode is considered a part of decode cycle or fetch-decode together is called fetch cycle.

Execute Cycle

Micro operations for each instruction are different and we list them first and then give the explanation.

LDA	$D_0T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_0T_4 : ACC \leftarrow MDR,\ \ TC \leftarrow 0$
STA	$D_1T_3 : MDR \leftarrow ACC$
	$D_1T_4 : \mathrm{M}[MAR] \leftarrow MDR,\ \ TC \leftarrow 0$
HLT	$D_2T_3 : S \leftarrow 1,\ \ TC \leftarrow 0$
AND	$D_3T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_3T_4 : ACC \leftarrow ACC \,\&\, MDR\ \ TC \leftarrow 0$
NOT	$D_4T_3 : ACC \leftarrow ACC',\ \ TC \leftarrow 0$
SHL	$D_5T_3 : ACC[7:1] \leftarrow ACC[6:0], ACC[0] \leftarrow 0, TC \leftarrow 0$
ADD	$D_6T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_6T_4 : ACC \leftarrow ACC + MDR,\ \ TC \leftarrow 0$
SUB	$D_7T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_7T_4 : ACC \leftarrow ACC - MDR, TC \leftarrow 0$

A quick overview of the above list shows, at the completion of each instruction cycle (fetch-decode-execute) *TC* is reset by which the computer goes to T_0 state and fetch cycle for next instruction begins. Note that, a detailed discussion on execution of the program at register level for every clock trigger appears in Section 16.5.

In operations like LDA, AND, ADD, SUB data is brought from memory, address of which is available in *MAR*. In executing STA the *MAR* content denotes the location where data is to be stored in memory.

Macro operations AND, NOT, ADD, SUB use ALU. When HLT is executed *S* flag is set which stops execution of the program. This flag is cleared through power-on-reset.

Now let's pick up one macro operation (say, LDA) and see how it gets executed through its constituent micro operations. From Table 16.1 we find instruction LDA transfers content of a specified memory location to *ACC*. If the opcode fetched in fetch cycle is 000 it refers to LDA operation. In decode phase, opcode 000 makes $D_0 = 1$ and the other outputs of ID are all zero. This is so till *IR* is refreshed or receives another opcode in the next fetch cycle, state T_2. Till then D_0 gives output 1.

The computer enters execution phase at state T_3 ($T_3 = 1$). Now as $D_0 = 1$, condition $D_0T_3 = 1$ and data is read from memory and loaded in *MDR*. Note that, the address of memory location from which data is to be brought was made available to *MAR* in state T_2. Also note that, memory content cannot directly be loaded into *ACC* (refer to data path shown in Fig. 16.8) and is to be done through *MDR*. In next clock cycle, i.e. when $D_0T_4 = 1$ the content of *MDR* is transferred to *ACC* via BUS and the macro operation is complete. We reset *TC* and let the computer begin a new instruction cycle. This analysis can be extended to explain execution of other instructions.

At this point we make an important observation that all the instruction executions are completed within 5 clock cycles (T_0 to T_4) and hence a 3-bit counter, which can count up to 8 is sufficient as *TC* in our simple computer.

SELF-TEST

7. What is a fetch cycle?
8. Why *TC* is reset every time an instruction is executed?

16.4 DESIGN OF CONTROL UNIT

The control unit is primarily a combinatorial circuit that supplies necessary controls inputs to all the important hardware elements of the computer. This takes timing information from computer master clock and is thus responsible for providing necessary *timing and control* information. The path through which these signals travel to reach different parts of a computer is called *control path*. Often we assign a group of wires, called *control bus* as shared path for this. The control logic is arrived at from (i) basic computer architecture we have adopted in the beginning, (ii) conditions appearing at left hand side of symbol ':' in RTL statements for our simple computer, given in previous section, and (iii) certain other issues, e.g. power-on-reset, control variables need to be activated for intended operation of a particular hardware, etc.

Loading Registers

Let us first see when parallel load control of *IR* is to be activated. We find from discussion of previous section, only during T_2 it is loaded. So TSD (Timing counter decoder) output T_2 can be directly connected as parallel load control input of *IR*. Every time T_2 is active this loads three MSBs of BUS (data path is such, refer to Fig. 16.2), which at that time holds *MDR* value, into *IR*. Obviously, at that time BUS selector (BS) should place content of *MDR* into BUS. This we'll discuss while designing control for BS.

What happens if we allow loading of *IR* say, in every clock cycle instead of above? Whenever there is some data made available in BUS by any hardware 3 MSB of that will be loaded into *IR*; ID (decoder) will immediately change and execution corresponding to a different opcode, not the intended one, may begin. You can understand it'll be all chaos without any sense. Thus we return sanity to our simple machine by loading *IR* only when opcode is fetched, i.e. in T_2 and we can write logic relation

$$\text{LOAD}_{IR} = T_2$$

We see, *MDR* is loaded during T_1, D_0T_3, D_1T_3, D_3T_3, D_6T_3, D_7T_3 and corresponding condition is

$$\text{LOAD}_{MDR} = T_1 + (D_0 + D_1 + D_3 + D_6 + D_7)T_3$$

Proceeding in same manner we can write, $\text{LOAD}_{MAR} = T_0 + T_2$ and

$$\text{LOAD}_{ACC} = D_4T_3 + (D_0 + D_3 + D_6 + D_7)T_4$$

Memory Read/Write

Memory read signal is invoked by:	$\text{READ}_\text{M} = T_1 + (D_0 + D_3 + D_6 + D_7)T_3$
Memory write signal is invoked by:	$\text{WRITE}_\text{M} = D_1T_4$

ALU Control

Control variables of ALU activated for addition:	$\text{ALU}_\text{ADD} = D_6T_4$
Control variables of ALU activated for subtraction:	$\text{ALU}_\text{SUB} = D_7T_4$
Control variables of ALU activated for logic AND:	$\text{ALU}_\text{AND} = D_3T_4$
Control variables of ALU activated for logic NOT:	$\text{ALU}_\text{NOT} = D_4T_3$

BUS Controller

BUS controller gives access	to *ACC* by	$\text{BUS}_{ACC} = D_1T_3$,
	to *PC* by	$\text{BUS}_{PC} = T_0$,
	to *MDR* by	$\text{BUS}_{MDR} = T_2 + D_0T_4$
and	to ALU by	$\text{BUS}_\text{ALU} = D_4T_3 + (D_3 + D_6 + D_7)T_4$

Thus, selection inputs of eight 4-to-1 multiplexers that places data from one of these four devices *ACC*, *PC*, *MDR* and *ALU* on BUS should become active when corresponding conditions mentioned by above logic equations are met.

Other Control Signal

The condition for setting START/STOP flag *S* is:	$\text{SET}_S = D_2T_3$ [*S* is power on reset]
The condition for shift left operation of *ACC* is:	$\text{SHIFT\_LEFT}_{ACC} = D_5T_3$
The signal that triggers increment of *PC*:	$\text{INCREMENT}_{PC} = T_1$
Timing counter *TC* is synchronously reset by:	$\text{RESET}_{TC} = (D_2 + D_4 + D_5)\,T_3 + (D_0 + D_1 + D_3 + D_6 + D_7)\,T_4$

Finally, the master clock remains enabled if flag *S* is not set. Thus $\text{ENABLE}_{CLOCK} = S'$

Based on these equations the control unit of our simple computer can be made. We show the control circuit of *ACC*, *TC* and *TSD*, BS in following three examples. Refer to problems of Section 4 of this chapter for more circuits. Together they make the control unit of our simple computer.

Example 16.3 Show using circuit diagrams the control inputs to *ACC*.

Solution The parallel load shift register *ACC* in the simple computer designed shifts data to left while serial data in is 0 (GND). It also loads parallel data from BUS. The conditions for these two operations are shown above in the form of logic equations. The corresponding diagram is shown in Fig. 16.5.

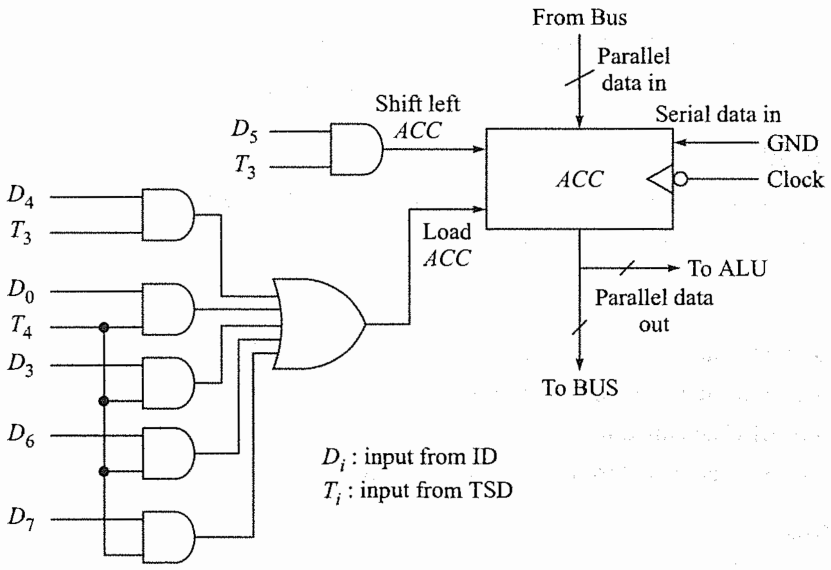


Fig. 16.5 Control for ACC

Example 16.4 Show using diagrams control inputs to *TC* and its connection to TSD.

Solution The timing counter *TC* is a mod-8 up counter with parallel load facility. When RESET$_{TC}$ is activated according to control logic discussed in this section, 000 is synchronously loaded and up count resumes. The required circuit diagram is shown in Fig. 16.6.

Example 16.5 Show using diagram how bus controller works.

Solution The controller developed from control equations discussed in this section is shown in Fig. 16.7. This is developed on multiplexer logic like Fig. 4.5 of Chapter 5. A tri-state bus control can also be designed similar to diagram shown in Fig. 14.26 of Chapter 14. There the *DISABLE* control input will be fed by complement of respective BUS activate signal, i.e. complement of BUS$_{ACC}$, BUS$_{PC}$ etc.

Note that, *PC* does not access bit 5 to 7 of BUS as it has only 5 bit binary information that is transferred to *MAR* via bit 0 to 4 of the BUS. Hence, for 3 MSB there is one AND gate less and the OR gate is of 3 input.

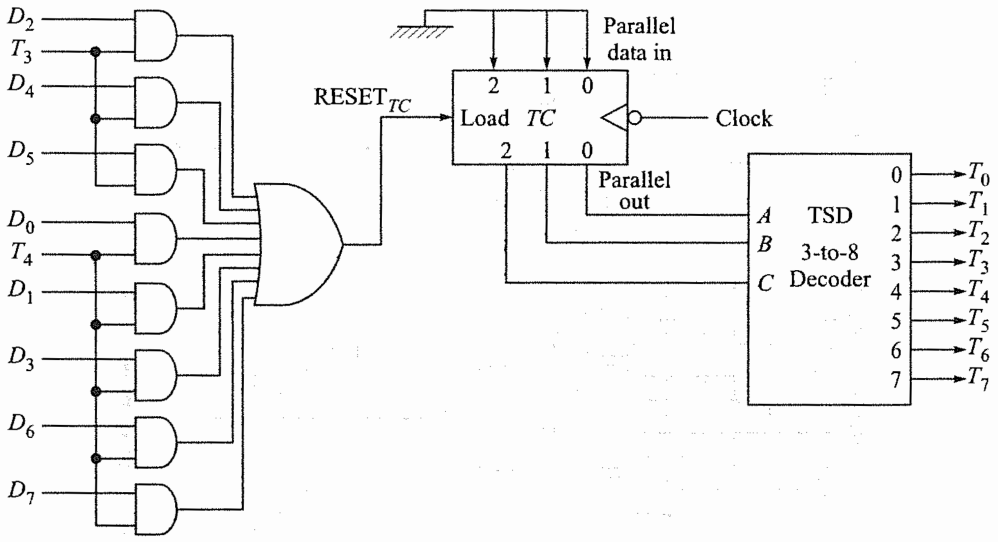


Fig. 16.6 **Control of Timing Counter, TC and its connection with Timing Sequence Detector, TSD**

SELF-TEST

9. How long instruction decoder outputs $D_7 \ldots D_0$ remain constant?
10. What are the instructions of this simple computer in which ALU places data on BUS?
11. Which instruction sets flag F?

16.5 PROGRAMMING COMPUTER

Now that our simple computer is ready with hardware and instruction sets let us see what computer program can solve the problem with which we started designing our simple machine. In Table 16.2 we present the program in mnemonics along with comments on job done by each instruction. Program in binary code as exists in 32 × 8 memory module will be shown after that.

Thus we need 14 instructions (Table 16.2), all single byte to solve the problem in our simple computer. We need 12 memory locations for storing numbers. So 14 + 12 = 26 bytes of our 32 byte memory are used for this problem. For bigger sized problems we need bigger memory and for more complex problem additional instruction sets and, of course, more complex computer architecture.

Now let us see how program and data remain stored in memory in binary numbers. We know that due to power-on-reset *PC* is always initialized with 00000, the first location of the memory (Refer Table 16.3) where first instruction of the program is to be stored. We use first 14 locations (address 00000 to 01101) of memory to store instructions. If we store data used in the program, i.e. 11 numbers in next consecutive locations then addresses 01110 to 11000 get filled. The location 11001, i.e. 26th location of memory can be used to store the result. Note that, multiplication is achieved by left shifting *ACC* and thus we don't need to store any multiplicand for that. If the 10 numbers to be added are say, 5, 2, 1, 3, 8, 6, 5, 2, 7, 4 and the number

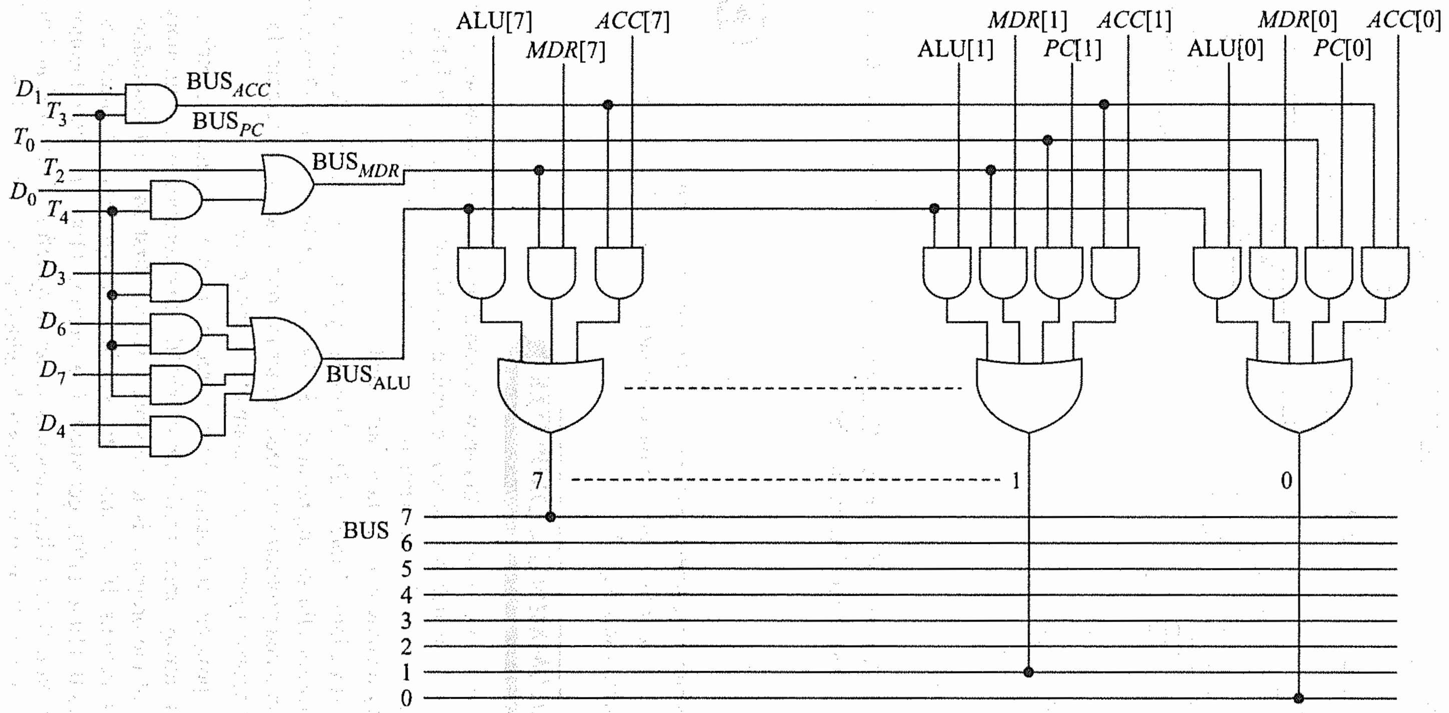


Fig. 16.7 **Control over bus by different devices using multiplexer logic**

Table 16.2 **Program to Solve Given Problem with Comments**

Instruction Number	*Instruction pnemonic*	*Comment*
1	LDA *addr*1	loads 1st number to *ACC*, the address of which follows opcode
2	ADD *addr*2	fetches 2nd number from memory and adds to 1st, stores the sum in *ACC*
3	ADD *addr*3	similarly adds 3rd number
4	ADD *addr*4	adds 4th number
5	ADD *addr*5	adds 5th number
6	ADD *addr*6	adds 6th number
7	ADD *addr*7	adds 7th number
8	ADD *addr*8	adds 8th number
9	ADD *addr*9	adds 9th number
10	ADD *addr*10	adds 10th number, now sum of 10 numbers remain available in *ACC*
11	SUB *addr*11	fetches 11th no. from memory, subtracts it from sum of 10 nos., stores result in *ACC*
12	SHL	shifts *ACC* to left by 1 bit, equivalent to multiplication by 2
13	STA *addr* 12	stores content of *ACC* in memory in the address available after opcode
14	HTL	halts the computer

subtracted is say, 9 then we can fill up first 25 locations of memory as shown in Fig. 16.14. The 26th location, before the program is run, may contain anything but after the program is run will contain the end result, i.e. 68 expressed in binary. Memory content is often shown in hexadecimal instead of binary. Refer to Problems 16.19 and 16.20 for this.

Program Execution

Now let us see how the program gets executed in first few instruction cycles. We note the change in the value of the registers along with ID and TSD in each clock cycle since the program begins. Table 16.4 shows sequential progress of our simple computer with every trigger of system clock. As told before *PC*, *TC* and *S* are power on reset. They all contain zero in the beginning when the computer is switched on.

In first clock cycle, the machine is in T_0 state given by TSD that decodes *TC*. At T_0, content of *PC* that contains the starting address of the program is copied to *MAR*. Corresponding micro operation is shown in rightmost column of Table 16.4. *TC* is incremented by 1.

In next clock cycle, *TC* and *PC* are incremented by 1, data from memory is loaded to *MDR* that contains the first instruction.

In 3rd clock cycle *TC* is incremented, *IR* gets the opcode and *MAR* gets address for first data. Note that decoding of *IR* is also done in same clock cycle that makes D_0 high, as opcode is 000 (LDA). This completes the fetch cycle, which is common for all instructions.

In executing LDA instruction the first state is T_3 state. Here, data from 15th location of Memory (*MAR* = 01110) which contains 00000101, decimal equivalent of 5 is loaded to *MDR*. In T_4 state this data is transferred to *ACC* and macro operation LDA is fully executed. This completes the first instruction cycle. Note that timing counter (*TC*) is to be reset after execution of data transfer from *MDR* to *ACC* and that begins the next instruction fetch.

Table 16.3 Program and Data Section of the Memory

Memory location number	*Memory address in binary*	*Memory content*	*Comment*
1	00000	00001110	Program section begins. Loads 1st no. from location 01110 to *ACC*. 3MSB 000: Load
2	00001	11001111	3MSB110: ADD, 5LSB 01111: Address of 2nd operand
3	00010	11010000	.
4	00011	11010001	.
5	00100	11010010	.
6	00101	11010011	First 14 locations, i.e. memory address 00000 to 01101 contain instructions.
7	00110	11010100	Here, three MSBs always refer to opcode. Five LSBs refer to memory
8	00111	11010101	address for instructions LDA, ADD, SUB, STA. For instructions
9	01000	11010110	SHL and HLT, five LSBs can be anything as they are not referred
10	01001	11010111	anywhere.
11	01010	11111000	.
12	01011	10100000	.
13	01100	00111001	.
14	01101	01000000	Halts computer. Program section ends.
15	01110	00000101	The data section starts. Stores 1st number, 5 expressed in binary
16	01111	00000010	2nd no. 2 in binary
17	10000	00000001	.
18	10001	00000011	.
19	10010	00001000	.
20	10011	00000110	.
21	10100	00000101	.
22	10101	00000010	.
23	10110	00000111	.
24	10111	00000100	.
25	11000	00001001	Stores 11th number, 9 that is subtracted from the sum of 10 nos.
26	11001	xxxxxxxx	After the program is run it becomes 01000100, i.e. 68 in decimal.
27	11010	xxxxxxxx	UNUSED
28	11011	xxxxxxxx	UNUSED
29	11100	xxxxxxxx	UNUSED
30	11101	xxxxxxxx	UNUSED
31	11110	xxxxxxxx	UNUSED
32	11111	xxxxxxxx	UNUSED

The fetch cycle is repeated in clock cycle 6 to 8. Since the instruction fetched is ADD (opcode 110) corresponding micro operations are performed in clock cycles 9 and 10 followed by next instruction fetch, starting again at 11th clock cycle. This continues till we reach 14th instruction HLT which when executed, sets *S* flag. This inhibits the system clock output in our design; thus content of all registers and memory will remain unchanged after that till the computer is switched off.

Table 16.4 **Execution of the Program at Register Level**

Clock Cycle	*TC$*	*TSD*	*PC$*	*MAR*	*MDR*	*IR*	*ID*	*ACC*	*S*	*Micro operation performed after clock trigger*
1	000	T_0	00000	00000	xxxxxxxx	xxx	x	xxxxxxxx	0	*MAR* ← *PC*
2	001	T_1	00000	00000	00001110	xxx	x	xxxxxxxx	0	*MDR* ← M[*MAR*], *PC*←*PC*+1
3	010	T_2	00001	01110	00001110	000	D_0	xxxxxxxx	0	*IR*←*MDR*[7:5], *MAR*←*MDR*[4:0]
4	011	T_3	00001	01110	00000101	000	D_0	xxxxxxxx	0	*MDR* ← M[*MAR*]
5	100	T_4	00001	01110	00000101	000	D_0	00000101	0	*ACC* ← *MDR*, *TC* ← 0
6	000	T_0	00001	00001	00000101	000	D_0	00000101	0	*MAR* ← *PC*
7	001	T_1	00001	00001	11001111	000	D_0	00000101	0	*MDR* ← M[*MAR*], *PC*←*PC*+1
8	010	T_2	00010	01111	11001111	110	D_6	00000101	0	*IR* ← *MDR*[7:5], *MAR* ←*MDR*[4:0]
9	011	T_3	00010	01111	00000010	110	D_6	00000101	0	*MDR* ← M[*MAR*]
10	100	T_4	00010	01111	00000010	110	D_6	00000111	0	*ACC* ←*ACC* + *MDR*, *TC*←0
11	000	T_0	00010	00010	00000010	110	D_6	00000111	0	*MAR* ← *PC*
12	001	T_1	00010	00010	11010000	110	D_6	00000111	0	*MDR* ← M[*MAR*], *PC*←*PC*+1
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...

$*PC*, *TC* (also *TSD*) values shown are the ones before clock trigger while for other registers this is what appears after clock trigger (taking care from *TC* and *PC*).

Concluding Remark

Before we conclude our computer design exercise let us see what we have achieved and what more is needed to make this computer fully functional. We have designed a simple processor comprising register arrays, flag, small memory, BUS and control unit. In short, we have designed a *central processing unit* (CPU) that connects to a small memory module and is able to execute programs built on a small instruction set.

What we have not discussed is how data is entered into computer from an external device say, keypad and also how it displays data in some output device say, a monitor. We also have not discussed interesting and important issues like

(i) handling of jump instructions (CPU jumps to an address to fetch an instruction),
(ii) use of subroutines (program under program that is used and called many times),
(iii) memory management (slow-fast, addressing mode details),
(iv) interrupt handling (devices of different priorities asking attention and service of CPU),
(v) pipelined CPU (doing jobs in parallel if there is no conflict to enhance computer speed, e.g. execution phase of present instruction in parallel with fetch of next instruction), so on and so forth. These are covered in detail in titles related to modern computer design courses and an interested reader can refer to the same.

We shall conclude this chapter by revisiting *computer architecture* introduced in Section 1.6, carrying forward the discussions of this chapter. Figure 16.8 represents a basic 8-bit computer. The 8-bit data bus is bidirectional in nature i.e. CPU is capable of both reading and writing from/to a location defined by 16-bit address which is a total of 2^{16} = 64K locations. Individual RAM and ROM are of size 16K and this requires 14 bits for addressing. The two MSBs are sent to a 2 to 4 address decoder which generates four Chip Select (CS)

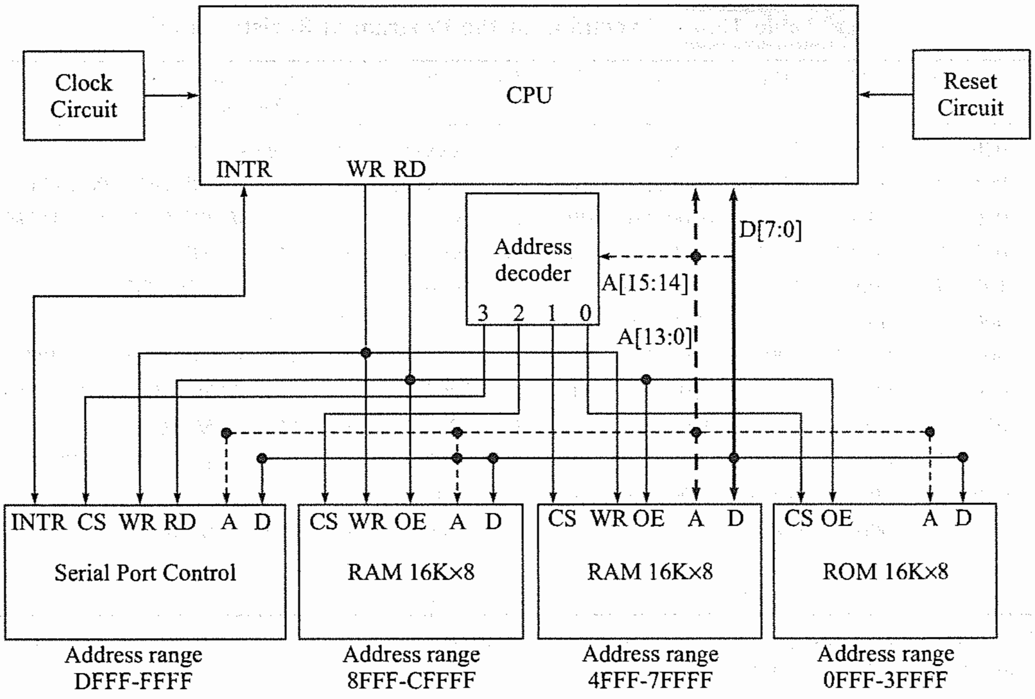


Fig. 16.8 **Basic architecture of an 8-bit computer**

signals, connected to each of the memory and output module generating unique address ranges as specified in the bottom of the figure. The calculation is as follows. For the ROM, A[15:14] is always 00 and thus all possible values of A[13:0] generate address ranges 0000 0000 0000 0000 to 0011 1111 1111 1111, i.e. 0000 to 3FFF in hex. Similarly, for the first RAM block, A[15:14] is always 01 and thus all possible values of A[13:0] generate address ranges 0100 0000 0000 0000 to 0111 1111 1111 1111, i.e. 4000 to 7FFF etc. CPU read is enabled by activating the control signal, RD (Read). This, in turn, requests outputs of the devices from which data is to be read to be enabled through OE (Output Enable) or through RD, if it is a serial input-output port, following which CPU takes the value from data bus. The control signal WR (Write) is activated to enable CPU writing to devices. Note that WR and RD should not be activated simultaneously. The timing of these control signals are also important so that data, chip select and address are properly stabilized to avoid false reading and writing operations. The ROM is not writable and usually contains sequence of instructions required for booting. This is usually used during *power on* of the computer and also in between, if the computer is asked to stop all operations and start afresh. The other time a computer may be asked to stop its usual fetch-decode-execute operations, but only temporarily, is when an *interrupt* is invoked. Then the computer's present state is stored in a designated memory space called *stack*. The computer comes back to its usual operating state once the interrupt is served usually through a *interrupt service routine* (ISR). There could be both software and hardware interrupts. The serial port control block shows how a hardware interrupt can ask service from CPU by activating INTR. Note that the *maskable interrupts* can be masked (disabled) by writing into a control register while *non-maskable interrupts* cannot be disabled. Reset is a non-maskable interrupt and care should be taken in the design of a computer so that corresponding ISR is in place before an interrupt is invoked.

Example 16.6 How many clock cycles are needed to execute the program shown in Table 16.2?

Solution The calculation of clock cycles is as follows.

One LDA	:	5
Nine ADD	:	$9 \times 5 = 45$
One SUB	:	5
One STA	:	5
One SHL	:	4
One HLT	:	4
TOTAL =		68 clock cycles

Example 16.7 Write a program for this computer that adds two positive integers, available in memory locations *addr*1 and *addr*2, multiplies the sum by 5 and stores final result in location *addr*3. Consider, the numbers are small enough not to cause any overflow, i.e. data at every stage require less than 8-bits for its representation.

Solution Addition of two numbers is straightforward and can be done using LDA and ADD instructions as done before. For multiplication with 5 we have to use an indirect technique. Two left shift give multiplication by 4 and one more addition will make it multiplication by 5. Alternatively, 5 ADD operations will also give multiplication by 5. The program can be written as follows.

```
LDA addr1
ADD addr2
STA addr3
SHL
SHL
ADD addr3
STA addr3
HLT
```

Note that, we have used *addr*3 as intermediate storage of addition result. Since in the computer designed there is no instruction to place data on a register from *ACC* (and also retrieve the same) we had to use memory. Storing intermediate results in registers speeds up the process but here we are limited by the architecture and instruction set available. Also note, we could have used any other available memory location for intermediate storage.

Example 16.8 Write a program for this computer that performs bit-wise Ex-OR operation on two numbers available in memory locations *addr*1 and *addr*2. The result is to be stored in location *addr*3.

Solution Our designed computer can perform only two kinds of logic operations AND and NOT. Therefore, we break Ex-OR logic of two numbers, say A and B in such a way that there is only AND and NOT operator.

$$Y = A \oplus B = AB' + A'B = ((AB')'.(A'B)')' \quad \text{[From DeMorgan's Theorem]}$$

Thus the program can be written as shown next. The logic operation performed by each instruction is shown as comment after semicolon.

```
LDA     addr1     ; A
NOT               ; A'
AND     addr2     ; A'B
NOT               ; (A'B)'
STA     addr3
LDA     addr2     ; B
NOT               ; B'
AND     addr1     ; AB'
NOT               ; (AB')'
AND     addr3     ; (AB')'.(A'B)'
NOT               ; ((AB')'.(A'B)')'
STA     addr3
HLT
```

SELF-TEST

12. What happens to the program shown in Table 16.2 or Table 16.3 if HLT instruction is not provided?

SUMMARY

A computer stores program or binary coded instructions in its program memory. The central processing unit comprising set of registers and a control unit sequentially fetches this program, decodes it and executes the same. To accomplish this, an instruction, also called macro operation is broken into series of micro operations. Register Transfer Language is a very convenient tool to express each of these micro operations. A simple computer is designed in this chapter that has eight instructions and can perform logic operations like AND, NOT and arithmetic operations like addition and subtraction. It can also load data from memory and store data in memory. The data path and control unit of the computer is designed using hardware discussed in earlier chapters of the book. The programming technique for this computer for various arithmetic and logic problems is also demonstrated.

GLOSSARY

- ***accumulator*** A multipurpose register that stores one operand of all arithmetic and logic operations and also for memory referenced data transfers.
- ***address bus*** Group of wires that transfer address information.
- ***arithmetic logic unit*** A combinatorial circuit that can perform various types of arithmetic and logic functions decided by a set of selection inputs.
- ***bus*** A group of wire providing shared common path between number of devices.
- ***central processing unit*** The brain of computer that controls the operations of a computer.
- ***computer architecture*** Organization of a digital computer.

- ***control bus*** Group of wires that transfer control information.
- ***control path*** The path through which control signals travel to different devices and make them perform their assigned tasks.
- ***data bus*** Group of wires that transfer data.
- ***data memory*** The part of memory that contains data.
- ***data path*** The path through which data moves from one device to another in a computer.
- ***flag*** A single flip-flop that stores binary outcome of a certain operation.
- ***instruction register*** A register that contains the opcode or binary code of an instruction.
- ***interrupt*** An event that asks computer's immediate attention.
- ***interrupt service routine*** A set of computer instructions that serves an interrupt.
- ***macro operation*** An instruction that a computer executes in a complete instruction cycle, consists of series of micro operations.
- ***maskable interrupt*** Interrupt that can be disabled.
- ***memory address register*** A register that contains address of memory location for all memory referenced instructions.
- ***memory data register*** A register that acts as buffer between memory and rest of the circuit, storing data that moves to and from memory.
- ***micro operation*** The basic operation, a computer performs at register level.
- ***non-maskable interrupt*** Interrupt that cannot be disabled.
- ***opcode*** The binary code of an instruction.
- ***program*** Series of instructions that accomplishes a task in a computer.
- ***program counter*** A register that stores address of next instruction.
- ***program memory*** The part of memory that contains instruction.
- ***register transfer language*** A language, which expresses register transfer and condition for that.
- ***stack*** A memory block usually used for storing a computer's present state when interrupt is invoked.
- ***system clock*** The clock providing basic unit of clock cycle from which trigger of all sequential operations are derived.

PROBLEMS

Section 16.1

16.1 For memory configured as in Fig. 16.2, if immediate addressing is allowed what is the maximum value of number (in decimal) that can be loaded through instruction fetch?

16.2 What is the minimum size required for *MAR* if memory addressed has size 1K × 16?

16.3 For a more complex computer design, 75 different instructions are required. What size of *IR* would you likely choose?

16.4 Draw data path of the computer described next. The computer in addition to what is described in Section 16.2 has two more registers *P* and *Q*, which can transfer data to/from *ACC* via BUS. It also has a *CY* flag that stores ALU overflow and a *Z* flag that is set when all the bits of *ACC* are zero.

Section 16.2

16.5 What does the following statement mean $(X + Y) : A \leftarrow B$

16.6 Explain the meaning of $XY : A \leftarrow B$

16.7 Give the final content of *ACC* when following statements are executed

$$T_1 : ACC \leftarrow ACC \oplus MDR$$
$$T_2 : ACC \leftarrow ACC'$$

16.8 State what the following statement performs for the computer described in problem 16.8

$$T_1 : ACC \leftarrow ACC + MDR$$

$CY \,\&\, T_2 : P \leftarrow ACC$

$CY' \,\&\, T_2 : Q \leftarrow ACC$

Section 16.3

16.9 Show how shift left operation with carry is executed.

16.10 Show how shift right operation with carry can be executed.

16.11 Consider the first instruction of the simple computer is replaced by MVI that moves immediate data (immediate addressing) to *ACC*. Write micro operations for this instruction. What change in hardware is required for this?

16.12 Consider the first instruction of the simple computer is replaced by LDI that moves indirect data (indirect addressing) to *ACC*. Write micro operations for this instruction. Does it require any change in hardware?

Section 16.4

16.13 What does $\text{LOAD}_{MAR} = T_0 + T_2$ mean?

16.14 Explain if there will be any problem if by mistake the control unit is developed on logic equation $\text{LOAD}_{MAR} = T_0 + T_2 + T_4$.

16.15 Show using diagrams control inputs to ALU. Consider IC 74181 (Section 6.10, Chapter 6) is used as ALU.

16.16 Show using diagrams control inputs to Memory.

Section 16.5

16.17 How many clock cycles are required to execute program given for Example 16.7?

16.18 How many clock cycles are required to execute program given for Example 16.8?

16.19 If the two numbers used in Example 16.7 are 5 and 8, and data section immediately follows program section show the memory values in binary after the program is executed? How will it be represented in hexadecimal?

16.20 If the two numbers used in Example 16.8 are $F2_{16}$ and $D6_{16}$ and data section immediately follows program section show the memory values in binary after the program is executed?

16.21 Write a program that compares two binary data located in *addr*1 and *addr*2 of memory by making all the bits of *addr*3 one if two numbers are exactly equal.

16.22 Write a program that executes following where Data_{addr} refers to data corresponding to address *addr*.

$$\text{Data}_{addr7} = (\text{Data}_{addr1} + \text{Data}_{addr2} + \text{Data}_{addr3} + \text{Data}_{addr4}) \times 3 - (\text{Data}_{addr5} + \text{Data}_{addr6}) \times 2$$

Answers to Self-tests

1. $2^{16} - 1 = 65535$
2. A computer operation coded in a group of binary digits.
3. To store address from which next instruction is fetched.
4. The instruction fetches address of location in which address for operand exists.
5. Register Transfer Language.
6. Control input to ALU.
7. A part of instruction cycle that fetches instruction from memory that after decoding gets executed in execute cycle.
8. Resetting *TC* a new instruction cycle can begin.
9. Till it is loaded in next fetch cycle.
10. ADD, SUB, AND, NOT.
11. HLT.
12. It goes on executing by loading next content of memory which houses first number and since three MSB are 000 opcode decodes it as an LDA instruction.

Appendix 1: Binary-Hexadecimal-Decimal Equivalents

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
0000 0000	00	0	0
0000 0001	01	256	1
0000 0010	02	512	2
0000 0011	03	768	3
0000 0100	04	1,024	4
0000 0101	05	1,280	5
0000 0110	06	1,536	6
0000 0111	07	1,792	7
0000 1000	08	2,048	8
0000 1001	09	2,304	9
0000 1010	0A	2560	10
0000 1011	0B	2,816	11
0000 1100	0C	3,072	12
0000 1101	0D	3,328	13
0000 1110	0E	3,584	14
0000 1111	0F	3,840	15
0001 0000	10	4,096	16
0001 0001	11	4,352	17
0001 0010	12	4,608	18
0001 0011	13	4,864	19
0001 0100	14	5,120	20
0001 0101	15	5,376	21
0001 0110	16	5,632	22
0001 0111	17	5,888	23
0001 1000	18	6,144	24
0001 1001	19	6,400	25
0001 1010	1A	6,656	26
0001 1011	1B	6,912	27
0001 1100	1C	7,168	28
0001 1101	1D	7,424	29
0001 1110	1E	7,680	30

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
0001 1111	1F	7,936	31
0010 0000	20	8,192	32
0010 0001	21	8,448	33
0010 0010	22	8,704	34
0010 0011	23	8,960	35
0010 0100	24	9,216	36
0010 0101	25	9,472	37
0010 0110	26	9,728	38
0010 0111	27	9,984	39
0010 1000	28	10,240	40
0010 1001	29	10,496	41
0010 1010	2A	10,752	42
0010 1011	2B	11,008	43
0010 1100	2C	11,264	44
0010 1101	2D	11,520	45
0010 1110	2E	11,776	46
0010 1111	2F	12,032	47
0011 0000	30	12,288	48
0011 0001	31	12,544	49
0011 0010	32	12,800	50
0011 0011	33	13,056	51
0011 0100	34	13,312	52
0011 0101	35	13,568	53
0011 0110	36	13,824	54
0011 0111	37	14,080	55
0011 1000	38	14,336	56
0011 1001	39	14,592	57
0011 1010	3A	14,848	58
0011 1011	3B	15,104	59
0011 1100	3C	15,360	60
0011 1101	3D	15,616	61
0011 1110	3E	15,872	62
0011 1111	3F	16,128	63
0100 0000	40	16,384	64
0100 0001	41	16,640	65
0100 0010	42	16,896	66
0100 0011	43	17,152	67
0100 0100	44	17,408	68
0100 0101	45	17,664	69
0100 0110	46	17,920	70
0100 0111	47	18,176	71
0100 1000	48	18,432	72
0100 1001	49	18,688	73
0100 1010	4A	18,944	74
0100 1011	4B	19,200	75

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
0100 1100	4C	19,456	76
0100 1101	4D	19,712	77
0100 1110	4E	19,968	78
0100 1111	4F	20,224	79
0101 0000	50	20,480	80
0101 0001	51	20,736	81
0101 0010	52	20,992	82
0101 0011	53	21,248	83
0101 0100	54	21,504	84
0101 0101	55	21,760	85
0101 0110	56	22,016	86
0101 0111	57	22,272	87
0101 1000	58	22,528	88
0101 1001	59	22,784	89
0101 1010	5A	23,040	90
0101 1011	5B	23,296	91
0101 1100	5C	23,552	92
0101 1101	5D	23,808	93
0101 1110	5E	24,064	94
0101 1111	5F	24,320	95
0110 0000	60	24,576	96
0110 0001	61	24,832	97
0110 0010	62	25,088	98
0110 0011	63	25,344	99
0110 0100	64	25,600	100
0110 0101	65	25,856	101
0110 0110	66	26,112	102
0110 0111	67	26,368	103
0110 1000	68	26,624	104
0110 1001	69	26,880	105
0110 1010	6A	27,136	106
0110 1011	6B	27,392	107
0110 1100	6C	27,648	108
0110 1101	6D	27,904	109
0110 1110	6E	28,160	110
0110 1111	6F	28,416	111
0111 0000	70	28,672	112
0111 0001	71	28,928	113
0111 0010	72	29,184	114
0111 0011	73	29,440	115
0111 0100	74	29,696	116
0111 0101	75	29,952	117
0111 0110	76	30,208	118
0111 0111	77	30,464	119
0111 1000	78	30,720	120

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
0111 1001	79	30,976	121
0111 1010	7A	31,232	122
0111 1011	7B	31,488	123
0111 1100	7C	31,744	124
0111 1101	7D	32,000	125
0111 1110	7E	32,256	126
0111 1111	7F	32,512	127
1000 0000	80	32,768	128
1000 0001	81	33,024	129
1000 0010	82	33,280	130
1000 0011	83	33,536	131
1000 0100	84	33,792	132
1000 0101	85	34,048	133
1000 0110	86	34,304	134
1000 0111	87	34,560	135
1000 1000	88	34,816	136
1000 1001	89	35,072	137
1000 1010	8A	35,328	138
1000 1011	8B	35,584	139
1000 1100	8C	35,840	140
1000 1101	8D	36,096	141
1000 1110	8E	36,352	142
1000 1111	8F	36,608	143
1001 0000	90	36,864	144
1001 0001	91	37,120	145
1001 0010	92	37,376	146
1001 0011	93	37,632	147
1001 0100	94	37,888	148
1001 0101	95	38,144	149
1001 0110	96	38,400	150
1001 0111	97	38,656	151
1001 1000	98	38,912	152
1001 1001	99	39,168	153
1001 1010	9A	39,424	154
1001 1011	9B	39,680	155
1001 1100	9C	39,936	156
1001 1101	9D	40,192	157
1001 1110	9E	40,448	158
1001 1111	9F	40,704	159
1010 0000	A0	40,960	160
1010 0001	A1	41,216	161
1010 0010	A2	41,472	162
1010 0011	A3	41,728	163
1010 0100	A4	41,984	164
1010 0101	A5	42,240	165

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
1010 0110	A6	42,496	166
1010 0111	A7	42,752	167
1010 1000	A8	43,008	168
1010 1001	A9	43,264	169
1010 1010	AA	43,520	170
1010 1011	AB	43,776	171
1010 1100	AC	44,032	172
1010 1101	AD	44,288	173
1010 1110	AE	44,544	174
1010 1111	AF	44,800	175
1011 0000	B0	45,056	176
1011 0001	B1	45,312	177
1011 0010	B2	45,568	178
1011 0011	B3	45,824	179
1011 0100	B4	46,080	180
1011 0101	B5	46,336	181
1011 0110	B6	46,592	182
1011 0111	B7	46,848	183
1011 1000	B8	47,104	184
1011 1001	B9	47,360	185
1011 1010	BA	47,616	186
1011 1011	BB	47,872	187
1011 1100	BC	48,128	188
1011 1101	BD	48,384	189
1011 1110	BE	48,640	190
1011 1111	BF	48,896	191
1100 0000	C0	49,152	192
1100 0001	C1	49,408	193
1100 0010	C2	49,664	194
1100 0011	C3	49,920	195
1100 0100	C4	50,176	196
1100 0101	C5	50,432	197
1100 0110	C6	50,688	198
1100 0111	C7	50,944	199
1100 1000	C8	51,200	200
1100 1001	C9	51,456	201
1100 1010	CA	51,712	202
1100 1011	CB	51,968	203
1100 1100	CC	52,224	204
1100 1101	CD	52,480	205
1100 1110	CE	52,736	206
1100 1111	CF	52,992	207
1101 0000	D0	53,248	208
1101 0001	D1	53,504	209
1101 0010	D2	53,760	210

Binary	*Hexadecimal*	*Upper Byte*	*Lower Byte*
1101 0011	D3	54,016	211
1101 0100	D4	54,272	212
1101 0101	D5	54,528	213
1101 0110	D6	54,784	214
1101 0111	D7	55,040	215
1101 1000	D8	55,296	216
1101 1001	D9	55,552	217
1101 1010	DA	55,808	218
1101 1011	DB	56,064	219
1101 1100	DC	56,320	220
1101 1101	DD	56,576	221
1101 1110	DE	56,832	222
1101 1111	DF	57,088	223
1110 0000	E0	57,344	224
1110 0001	E1	57,600	225
1110 0010	E2	57,856	226
1110 0011	E3	58,112	227
1110 0100	E4	58,368	228
1110 0101	E5	58,624	229
1110 0110	E6	58,880	230
1110 0111	E7	59,136	231
1110 1000	E8	59,392	232
1110 1001	E9	59,648	233
1110 1010	EA	59,904	234
1110 1011	EB	60,160	235
1110 1100	EC	60,416	236
1110 1101	ED	60,672	237
1110 1110	EE	60,928	238
1110 1111	EF	61,184	239
1111 0000	F0	61,440	240
1111 0001	F1	61,696	241
1111 0010	F2	61,952	242
1111 0011	F3	62,208	243
1111 0100	F4	62,464	244
1111 0101	F5	62,720	245
1111 0110	F6	62,976	246
1111 0111	F7	63,232	247
1111 1000	F8	63,488	248
1111 1001	F9	63,744	249
1111 1010	FA	64,000	250
1111 1011	FB	64,256	251
1111 1100	FC	64,512	252
1111 1101	FD	64,768	253
1111 1110	FE	65,024	254
1111 1111	FF	65,280	255

Appendix 2: 2′s Complement Representation

Positive			*Negative*		
Decimal	*Hexadecimal*	*Binary*	*Binary*	*Hexadecimal*	*Decimal*
0	00H	0000 0000	0000 0000	00H	–0
1	01H	0000 0001	1111 1111	FFH	–1
2	02H	0000 0010	1111 1110	FEH	–2
3	03H	0000 0011	1111 1101	FDH	–3
4	04H	0000 0100	1111 1100	FCH	–4
5	05H	0000 0101	1111 1011	FBH	–5
6	06H	0000 0110	1111 1010	FAH	–6
7	07H	0000 0111	1111 1001	F9H	–7
8	08H	0000 1000	1111 1000	F8H	–8
9	09H	0000 1001	1111 0111	F7H	–9
10	0AH	0000 1010	1111 0110	F6H	–10
11	0BH	0000 1011	1111 0101	F5H	–11
12	0CH	0000 1100	1111 0100	F4H	–12
13	0DH	0000 1101	1111 0011	F3H	–13
14	0EH	0000 1110	1111 0010	F2H	–14
15	0FH	0000 1111	1111 0001	F1H	–15
16	10H	0001 0000	1111 0000	F0H	–16
17	11H	0001 0001	1110 1111	EFH	–17
18	12H	0001 0010	1110 1110	EEH	–18
19	13H	0001 0011	1110 1101	EDH	–19
20	14H	0001 0100	1110 1100	ECH	–20
21	15H	0001 0101	1110 1011	EBH	–21
22	16H	0001 0110	1110 1010	EAH	–22
23	17H	0001 0111	1110 1001	E9H	–23
24	18H	0001 1000	1110 1000	E8H	–24
25	19H	0001 1001	1110 0111	E7H	–25
26	1AH	0001 1010	1110 0110	E6H	–26
27	1BH	0001 1011	1110 0101	E5H	–27

Positive			Negative		
Decimal	*Hexadecimal*	*Binary*	*Binary*	*Hexadecimal*	*Decimal*
28	1CH	0001 1100	1110 0100	E4H	−28
29	1DH	0001 1101	1110 0011	E3H	−29
30	1EH	0001 1110	1110 0010	E2H	−30
31	1FH	0001 1111	1110 0001	E1H	−31
32	20H	0010 0000	1110 0000	E0H	−32
33	21H	0010 0001	1101 1111	DFH	−33
34	22H	0010 0010	1101 1110	DEH	−34
35	23H	0010 0011	1101 1101	DDH	−35
36	24H	0010 0100	1101 1100	DCH	−36
37	25H	0010 0101	1101 1011	DBH	−37
38	26H	0010 0110	1101 1010	DAH	−38
39	27H	0010 0111	1101 1001	D9H	−39
40	28H	0010 1000	1101 1000	D8H	−40
41	29H	0010 1001	1101 0111	D7H	−41
42	2AH	0010 1010	1101 0110	D6H	−42
43	2BH	0010 1011	1101 0101	D5H	−43
44	2CH	0010 1100	1101 0100	D4H	−44
45	2DH	0010 1101	1101 0011	D3H	−45
46	2EH	0010 1110	1101 0010	D2H	−46
47	2FH	0010 1111	1101 0001	D1H	−47
48	30H	0011 0000	1101 0000	D0H	−48
49	31H	0011 0001	1100 1111	CFH	−49
50	32H	0011 0010	1100 1110	CEH	−50
51	33H	0011 0011	1100 1101	CDH	−51
52	34H	0011 0100	1100 1100	CCH	−52
53	35H	0011 0101	1100 1011	CBH	−53
54	36H	0011 0110	1100 1010	CAH	−54
55	37H	0011 0111	1100 1001	C9H	−55
56	38H	0011 1000	1100 1000	C8H	−56
57	39H	0011 1001	1100 0111	C7H	−57
58	3AH	0011 1010	1100 0110	C6H	−58
59	3BH	0011 1011	1100 0101	C5H	−59
60	3CH	0011 1100	1100 0100	C4H	−60
61	3DH	0011 1101	1100 0011	C3H	−61
62	3EH	0011 1110	1100 0010	C2H	−62
63	3FH	0011 1111	1100 0001	C1H	−63
64	40H	0100 0000	1100 0000	C0H	−64
65	41H	0100 0001	1011 1111	BFH	−65
66	42H	0100 0010	1011 1110	BEH	−66
67	43H	0100 0011	1011 1101	BDH	−67
68	44H	0100 0100	1011 1100	BCH	−68
69	45H	0100 0101	1011 1011	BBH	−69

Positive			*Negative*		
Decimal	*Hexadecimal*	*Binary*	*Binary*	*Hexadecimal*	*Decimal*
70	46H	0100 0110	1011 1010	BAH	−70
71	47H	0100 0111	1011 1001	B9H	−71
72	48H	0100 1000	1011 1000	B8H	−72
73	49H	0100 1001	1011 0111	B7H	−73
74	4AH	0100 1010	1011 0110	B6H	−74
75	4BH	0100 1011	1011 0101	B5H	−75
76	4CH	0100 1100	1011 0100	B4H	−76
77	4DH	0100 1101	1011 0011	B3H	−77
78	4EH	0100 1110	1011 0010	B2H	−78
79	4FH	0100 1111	1011 0001	B1H	−79
80	50H	0101 0000	1011 0000	B0H	−80
81	51H	0101 0001	1010 1111	AFH	−81
82	52H	0101 0010	1010 1110	AEH	−82
83	53H	0101 0011	1010 1101	ADH	−83
84	54H	0101 0100	1010 1100	ACH	−84
85	55H	0101 0101	1010 1011	ABH	−85
86	56H	0101 0110	1010 1010	AAH	−86
87	57H	0101 0111	1010 1001	A9H	−87
88	58H	0101 1000	1010 1000	A8H	−88
89	59H	0101 1001	1010 0111	A7H	−89
90	5AH	0101 1010	1010 0110	A6H	−90
91	5BH	0101 1011	1010 0101	A5H	−91
92	5CH	0101 1100	1010 0100	A4H	−92
93	5DH	0101 1101	1010 0011	A3H	−93
94	5EH	0101 1110	1010 0010	A2H	−94
95	5FH	0101 1111	1010 0001	A1H	−95
96	60H	0110 0000	1010 0000	A0H	−96
97	61H	0110 0001	1001 1111	9FH	−97
98	62H	0110 0010	1001 1110	9EH	−98
99	63H	0110 0011	1001 1101	9DH	−99
100	64H	0110 0100	1001 1100	9CH	−100
101	65H	0110 0101	1001 1011	9BH	−101
102	66H	0110 0110	1001 1010	9AH	−102
103	67H	0110 0111	1001 1001	99H	−103
104	68H	0110 1000	1001 1000	98H	−104
105	69H	0110 1001	1001 0111	97H	−105
106	6AH	0110 1010	1001 0110	96H	−106
107	6BH	0110 1011	1001 0101	95H	−107
108	6CH	0110 1100	1001 0100	94H	−108
109	6DH	0110 1101	1001 0011	93H	−109
110	6EH	0110 1110	1001 0010	92H	−110
111	6FH	0110 1111	1001 0001	91H	−111

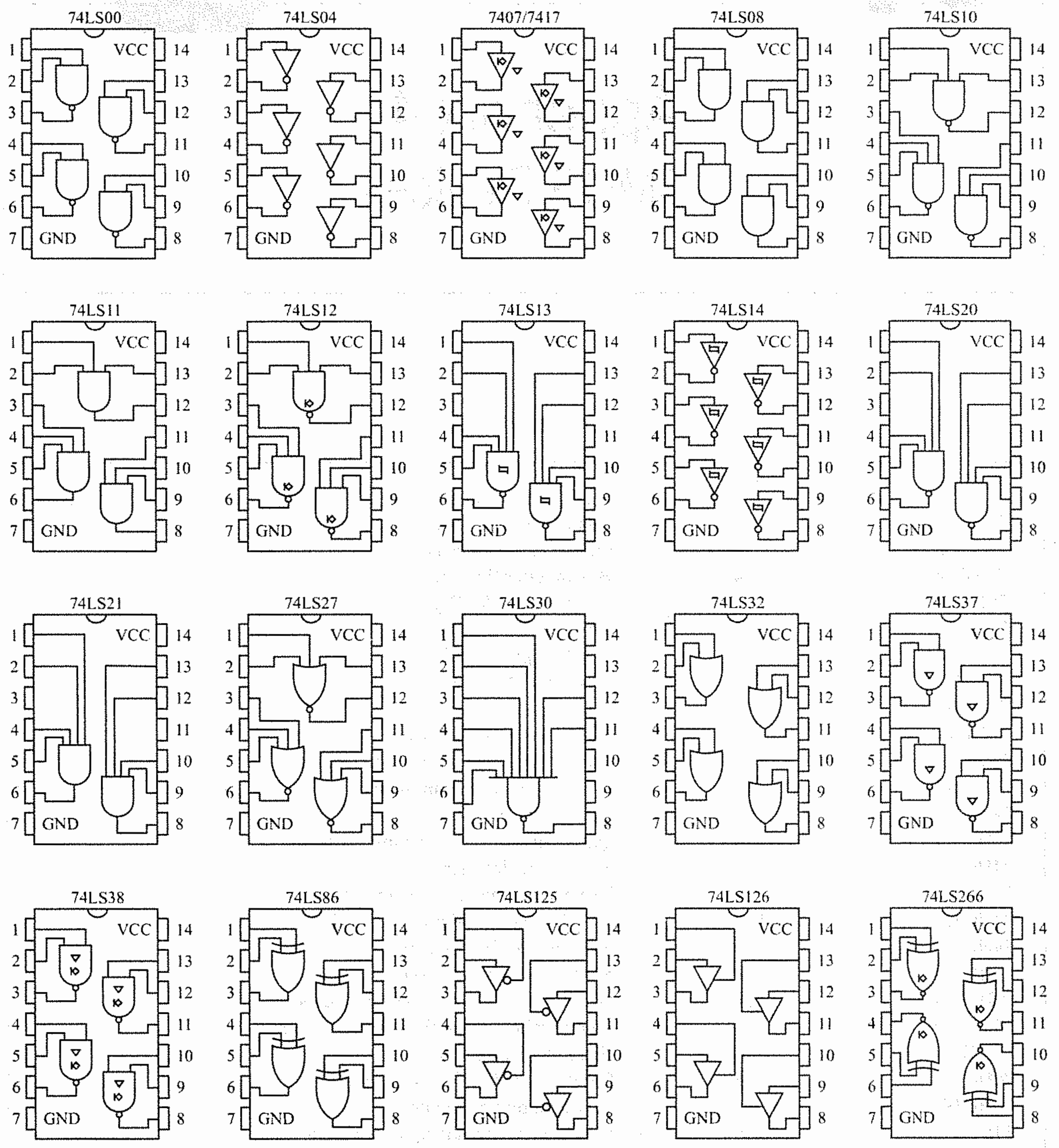
	Positive			*Negative*	
Decimal	*Hexadecimal*	*Binary*	*Binary*	*Hexadecimal*	*Decimal*
112	70H	0111 0000	1001 0000	90H	−112
113	71H	0111 0001	1000 1111	8FH	−113
114	72H	0111 0010	1000 1110	8EH	−114
115	73H	0111 0011	1000 1101	8DH	−115
116	74H	0111 0100	1000 1100	8CH	−116
117	75H	0111 0101	1000 1011	8BH	−117
118	76H	0111 0110	1000 1010	8AH	−118
119	77H	0111 0111	1000 1001	89H	−119
120	78H	0111 1000	1000 1000	88H	−120
121	79H	0111 1001	1000 0111	87H	−121
122	7AH	0111 1010	1000 0110	86H	−122
123	7BH	0111 1011	1000 0101	85H	−123
124	7CH	0111 1100	1000 0100	84H	−124
125	7DH	0111 1101	1000 0011	83H	−125
126	7EH	0111 1110	1000 0010	82H	−126
127	7FH	0111 1111	1000 0001	81H	−127
128	—	—	1000 0000	80H	−128

Appendix 3: TTL Devices

Number	*Function*
7400	Quad 2-input NAND gates
7401	Quad 2-input NAND gates (open collector)
7402	Quad 2-input NOR gates
7403	Quad 2-input NOR gates (open collector)
7404	Hex inverters
7405	Hex inverters (open collector)
7406	Hex inverter buffer-driver
7407	Hex buffer-drivers
7408	Quad 2-input AND gates
7409	Quad 2-input AND gates (open collector)
7410	Triple 3-input NAND gates
7411	Triple 3-input AND gates
7412	Triple 3-input NAND gates (open collector)
7413	Dual Schmitt triggers
7414	Hex Schmitt triggers
7416	Hex inverter buffer-drivers
7417	Hex buffer-drivers
7420	Dual 4-input NAND gates
7421	Dual 4-input AND gates
7422	Dual 4-input NAND gates (open collector)
7423	Expandable dual 4-input NOR gates
7425	Dual 4-input NOR gates
7426	Quad 2-input TTL-MOS interface NAND gates
7427	Triple 3-input NOR gates
7428	Quad 2-input NOR buffer
7430	8-input NAND gate
7432	Quad 2-input OR gates
7437	Quad 2-input NAND buffers
7438	Quad 2-input NAND buffers (open collector)
7439	Quad 2-input NAND buffers (open collector)
7440	Dual 4-input NAND buffers
7441	BCD-to-decimal decoder-Nixie driver
7442	BCD-to-decimal decoder
7443	Excess 3-to-decimal decoder
7444	Excess Gray-to-decimal
7445	BCD-to-decimal decoder-driver
7446	BCD-to-seven segment decoder-drivers (30-V output)
7447	BCD-to-seven segment decoder-drivers (15-V output)
7448	BCD-to-seven segment decoder-drivers
7450	Expandable dual 2-input 2-wide AND-OR-INVERT gates
7451	Dual 2-input 2-wide AND-OR-INVERT gates
7452	Expandable 2-input 4-wide AND-OR gates
7453	Expandable 2-input 4-wide AND-OR-INVERT gates
7454	2-input 4-wide AND-OR-INVERT gates
7455	Expandable 4-input 2-wide AND-OR-INVERT gates
7459	Dual 2-3 input 2-wide AND-OR-INVERT gates
7460	Dual 4-input expanders
7461	Triple 3-input expanders
7462	2-2-3-3 input 4-wide expanders
7464	2-2-3-4 input 4-wide AND-OR-INVERT gates
7465	4-wide AND-OR-INVERT gates (open collector)
7470	Edge-triggered *JK* flip-flop
7472	*JK* master-slave flip-flop
7473	Dual *JK* master-slave flip-flop
7474	Dual *D* flip-flop
7475	Quad latch

Number	*Function*	*Number*	*Function*
7476	Dual *JK* master-slave flip-flop	74162	Synchronous 4-bit counter
7480	Gates full adder	74163	Synchronous 4-bit counter
7482	2-bit binary full adder	74164	8-bit serial shift register
7483	4-bit binary full adder	74165	Parallel-load 8-bit serial shift register
7485	4-bit magnitude comparator	74166	8-bit shift register
7486	Quad EXCLUSIVE-OR gate	74173	4-bit three-state register
7489	64-bit random-access read-write memory	74174	Hex *F* flip-flop with clear
7490	Decade counter	74175	Quad *D* flip-flop with clear
7491	8-bit shift register	74176	35-MHz presettable decade counter
7492	Divide-by-12 counter	74177	35-MHz presettable binary counter
7493	4-bit binary counter	74179	4-bit parallel-access shift register
7494	4-bit shift register	74180	8-bit odd-even parity generator-checker
7495	4-bit right-shift–left-shift register	74181	Arithmetic-logic unit
7496	5-bit parallel-in–parallel-out shift register	74182	Look-ahead carry generator
74100	4-bit bistable latch	74184	BCD-to-binary converter
74104	*JK* master-slave flip-flop	74185	Binary-to-BCD converter
74105	*JK* master-slave flip-flop	74189	Three-state 64-bit random-access memory
74107	Dual *JK* master-slave flip-flop	74190	Up-down decade counter
74109	Dual *JK* positive-edge-triggered flip-flop	74191	Synchronous binary up-down counter
74116	Dual 4-bit latches with clear	74192	Binary up-down counter
74121	Monostable multivibrator	74193	Binary up-down counter
74122	Monostable multivibrator with clear	74194	4-bit directional shift register
74123	Monostable multivibrator	74195	4-bit parallel-access shift register
74125	Three-state quad bus buffer	74196	Presettable decade counter
74126	Three-state quad bus buffer	74197	Presettable binary counter
74132	Quad Schmitt trigger	74198	8-bit shift register
74136	Quad 2-input EXCLUSIVE-OR gate	74199	8-bit shift register
74141	BCD-to-decimal decoder-driver	74221	Dual one-shot Schmitt trigger
74142	BCD counter-latch-driver	74251	Three-state 8-channel multiplexer
74145	BCD-to-decimal decoder-driver	74259	8-bit addressable latch
74147	10/4 priority encoder	74276	Quad *JK* flip-flop
74148	Priority encoder	74279	Quad debouncer
74150	16-line-to-1-line multiplexer	74283	4-bit binary full adder with fast carry
74151	8-Channel digital multiplexer	74284	Three-state 4-bit multiplexer
74152	8-Channel data selector-multiplexer	74285	Three-state 4-bit multiplexer
74153	Dual 4/1 multiplexer	74365	Three-state hex buffers
74154	4-line-to-16-line decoder-demultiplexer	74366	Three-state hex buffers
74155	Dual 2/4 demultiplexer	74367	Three-state hex buffers
74156	Dual 2/4 demultiplexer	74368	Three-state hex buffers
74157	Quad 2/1 data selector	74390	Individual clocks with flip-flops
74160	Decade counter with asynchronous clear	74393	Dual 4-bit binary counter
74161	Synchronous 4-bit counter		

TTL CIRCUITS



Appendix 4:
CMOS Devices

74HC00 Series

Part No.	*Pins*	*Function*
74HC00	14	Quad 2-input NAND gate
74HC02	14	Quad 2-input NOR gate
74HC04	14	Hex inverter (buffered)
74HC08	14	Quad 2-input AND gate
74HC14	14	Hex inverting Schmitt trigger
74HC20	14	Dual 4-input NAND gate
74HC30	14	8-input NAND gate
74HC32	14	Quad 2-input OR gate
74HC42	16	BCD-to-decimal decoder
74HC74	14	Dual *D* flip-flop with preset and clear
74HC85	16	4-bit magnitude comparator
74HC123	16	Dual monostable multivibrator
74HC132	14	Quad 2-input NAND Schmitt trigger
74HC138	16	3- to 8-line decoder
74HC139	16	Expandable dual 2- to 4-line decoder
74HC154	S-24	4- to 16-line decoder (use 24SLP socket)
74HC161	16	Synchronous binary counter
74HC163	16	Synchronous binary counter
74HC164	14	8-bit serial in–parallel out shift register
74HC165	16	8-bit parallel in–serial out shift register
74HC174	16	Hex *D* Flip-Flop with clear
74HC175	16	Quad *D* type flip-flop with clear
74HC191	16	Up-down binary counter
74HC192	16	Synchronous decade up-down counter
74HC193	16	Synchronous binary up-down counter
74HC221	16	Dual monostable multivibrator
74HC240	20	Inverting octal tri-state buffer
74HC244	20	Octal tri-state buffer

74HC00 Series

Part No.	*Pins*	*Function*
74HC245	20	Octal tri-state transceiver
74HC257	16	Quad 2-channel tri-state multiplexer
74HC273	20	Octal *D* flip-flop
74HC367	16	Tri-state hex buffer
74HC373	20	Tri-state octal *D*-type latch
74HC374	20	Tri-state octal *D*-type flip-flop
74HC390	16	Dual 4-bit decade counter
74HC393	14	Dual 4-bit binary counter
74HC541	20	Octal buffer–line driver (tri-state)
74HC573	20	Tri-state octal *D*-type latch
74HC574	20	Tri-state octal *D*-type flip-flop
74HC595	16	8-bit serial to-parallel shift register latch
74HC688	20	8-bit magnitude comparator (equality detector)
74HC942	20	Full duplex low-speed 300-baud modem chip
74HC943	20	Full duplex 300-baud modem chip
74HC4017	16	Decade counter-divider with 10 decoded outputs
74HC4020	16	14-stage binary counter
74HC4040	16	12-stage binary counter
74HC4046	16	CMOS phase-lock loop
74HC4060	16	14-stage binary counter
74HC4066	14	Quad analog switch
74HC4514	S-24	4- to 16-line decoder with latch (use 24SLP socket)
74HC4538	16	Dual retriggerable monostable multivibrator

Appendix 5: Codes

Table A5.1

Decimal	*BCD*	*Binary*
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
...	......	...
98	1001 1000	1100010
99	1001 1001	1100011
100	0001 0000 0000	1100100
101	0001 0000 0001	1100101
102	0001 0000 0010	1100110
...	....	...
578	0101 0111 1000	1001000010
...	....	......

Table A5.2 4-Bit BCD Codes

Decimal	*7421*	*6311*	*5421*	*5311*	*5211*
0	0000	0000	0000	0000	0000
1	0001	0001	0001	0001	0001
2	0010	0011	0010	0011	0011
3	0011	0100	0011	0100	0101
4	0100	0101	0100	0101	0111
5	0101	0111	1000	1000	1000
6	0110	1000	1001	1001	1001
7	1000	1001	1010	1011	1011
8	1001	1011	1011	1100	1101
9	1010	1100	1100	1101	1111

Table A5.3 More 4-Bit BCD Codes

Decimal	*4221*	*3321*	*2421*	$84\bar{2}\bar{1}$	$74\bar{2}\bar{1}$
0	0000	0000	0000	0000	0000
1	0001	0001	0001	0111	0111
2	0010	0010	0010	0110	0110
3	0011	0011	0011	0101	0101
4	1000	0101	0100	0100	0100
5	0111	1010	1011	1011	1010
6	1100	1100	1100	1010	1001
7	1101	1101	1101	1001	1000
8	1110	1110	1110	1000	1111
9	1111	1111	1111	1111	1110

Table A5.4 5-Bit BCD Codes

Decimal	*2-out-of-5*	*63210*	*Shift-Counter*	*86421*	*51111*
0	00011	00110	00000	00000	00000
1	00101	00011	00001	00001	00001
2	00110	00101	00011	00010	00011
3	01001	01001	00111	00011	00111
4	01010	01010	01111	00100	01111
5	01100	01100	11111	00101	10000
6	10001	10001	11110	01000	11000
7	10010	10010	11100	01001	11100
8	10100	10100	11000	10000	11110
9	11000	11000	10000	10001	11111

Table A5.5 More than 5-Bit BCD Codes

Decimal	*50 43210*	*543210*	*9876543210*
0	01 00001	000001	0000000001
1	01 00010	000010	0000000010
2	01 00100	000100	0000000100
3	01 01000	001000	0000001000
4	01 10000	010000	0000010000
5	10 00001	100001	0000100000
6	10 00010	100010	0001000000
7	10 00100	100100	0010000000
8	10 01000	101000	0100000000
9	10 10000	110000	1000000000

Table A5.6 Excess-3 Code

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

Table A5.7 Gray Code

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

Appendix 6: BCD Codes

Hollerith Code

Punched cards are rarely, if ever, used today, but the *Hollerith code* is included here for historical and reference purposes. In this code the numbers 0 through 9 are represented by a single punch in a vertical column. For example, a hole punched in the fifth row of column 12 represents a 5 in that column. The letters of the alphabet are represented by two punches in any one column. The letters A and I are represented by a zone punch in row 12 and a punch in rows 1 through 9. The letters J through R are represented by a zone punch in row 11 and a punch in rows 1 through 9. The letters S through Z are represented by a zone punch in row 0 and a punch in rows 2 through 9. Thus, any of the 10 decimal digits and any of the 26 letters of the alphabet can be represented in a binary fashion by punching the proper holes in the card. In addition, a number of special characters can be represented by punching combinations of holes in a column which are not used for the numbers or letters of the alphabet.

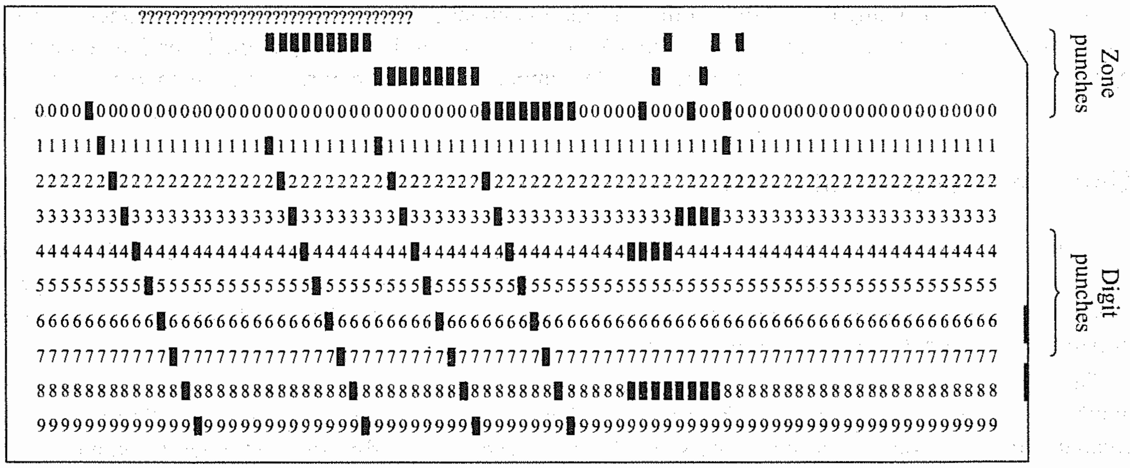


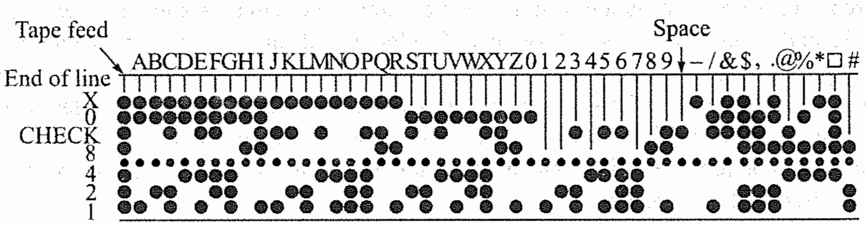
Fig. A6.1 Standard punched card using Hollerith code

An easy device for remembering the alphabetic characters is the phrase "JR is 11." Notice that the letters J through R have an 11 punch, those before have a 12 punch, and those after have a 0 punch. It is also necessary to remember that S begins on a 2 and not a 1.

Eight-Hole Code

Again, this information is for historical and reference purposes. There are a number of codes for punching data in paper tape, but one of the most widely used is the *eight-hole code.* Holes, representing data, are punched in eight parallel channels which run the length of the tape. (The channels are labeled 1, 2, 4, 8, parity, 0, *X*, and end of line.) Each character—numeric, alphabetic, or special—occupies one column of eight positions across the width of the tape.

Numbers are represented by punches in one or more channels labeled 0, 1, 2, 4, and 8, and each number is the sum of the punch positions. For example, 0 is represented by a single punch in the 0 channel; 1 is represented by a single punch in the 1 channel; 2 is a single punch in channel 2; 3 is a punch in channel 1 and a punch in channel 2, etc. Alphabetic characters are represented by a combination of punches in channels *X*, 0, 1, 2, 4, and 8. Channels *X* and 0 are used much as the zone punches in punched cards. For example, the letter *A* is designated by punches in channels *X*, 0, and 1. The special characters are represented by combinations of punches in all channels which are not used to designate either numbers or letters. A punch in the end-of-line channel signifies the end of a block of information, or the end of record. This is the only time a punch appears in this channel.



As a means of checking the validity of the information punched on the tape, the parity channel is used to ensure that each character is represented by an *odd* number of holes. For example, the letter *C* is represented by punches in channels *X*, 0, 1, and 2. Since an odd number of holes is required for each character, the code for the letter C also has a punch in the parity channel, and thus a total of five punches is used for this letter.

Universal Product Code (UPC)

The Universal Product Code (UPC) symbol in Fig. A6.3 is an example of a machine-readable label that appears on virtually every kind of retail grocery product. It is the result of an industrywide attempt to improve productivity through the use of automatic checkstand equipment. The standard symbol consists of a number of parallel light and dark bars of variable widths.

The symbol is designed around a 10-digit numbering system, 5 digits being assigned as an identification number for each manufacturer and the remaining 5 digits being used to identify a specific product, e.g. creamed corn, pea soup, or catsup. Each symbol can be read by a fixed-position scanner, as on a conveyor belt, or by a hand-held wand. The code numbers are printed on each symbol under the bars as a convenience in the event of equipment failure.

The American Standard Code for Information Exchange*

	000	001	010	011	100	101	110	111
0000	*NULL*	① DC_0	*b̸*	0	@	*P*		
0001	*SOM*	DC_1	!	1	*A*	*Q*		
0010	*EOA*	DC_2	”	2	*B*	*R*		
0011	*EOM*	DC_3	#	3	*C*	*S*		
0100	*EOT*	DC_4 (Stop)	$	4	*D*	*T*		
0101	*WRU*	ERR	%	5	*E*	*U*		
0110	*RU*	*SYNC*	&	6	*F*	*V*		
0111	*BELL*	*LEM*	’	7	*G*	*W*		
1000	FE_0	S_0	(	8	*H*	*X*	Unassigned	
1001	*HT* / *SK*	S_1	)	9	*I*	*Y*		
1010	*LF*	S_2	*	:	*J*	*Z*		
1011	V_{TAB}	S_3	+	;	*K*	[		
1100	*FF*	S_4	,	<	*L*	\		ACK
1101	*CR*	S_5	–	=	*M*	]		②
1110	*SO*	S_6	*	>	*N*	↑		*ESC*
1111	*SI*	S_7	/	?	*O*	←		*DEL*

Example $\boxed{100}\boxed{0001}$ = *A*

b_1----------b_1

The abbreviations used in the figure mean:

NULL	Null idle	*CR*	Carriage return
SOM	Start of message	*SO*	Shift out
EOA	End of address	*SI*	Shift in
EOM	End of message	DC_0	Device control ! Reserved for data Link escape
EOT	End of transmission	DC_1—DC_2	Device control
WRU	"Who are you?"	*ERR*	Error
RU	"Are you . . . ?"	*SYNC*	Synchronous idle
BELL	Audible signal	*LEM*	Logical end of media
FE	Format effector	SO_0—SO_7	Separator (information)
HT	Horizontal tabulation		Word separator (blank, normally nonprinting)
SK	Skip (punched card)	*ACK*	Acknowledge
LF	Line feed	2	Unassigned control
V/TAB	Vertical tabulation	*ESC*	Escape
FF	Form feed	*DEL*	Delete idle

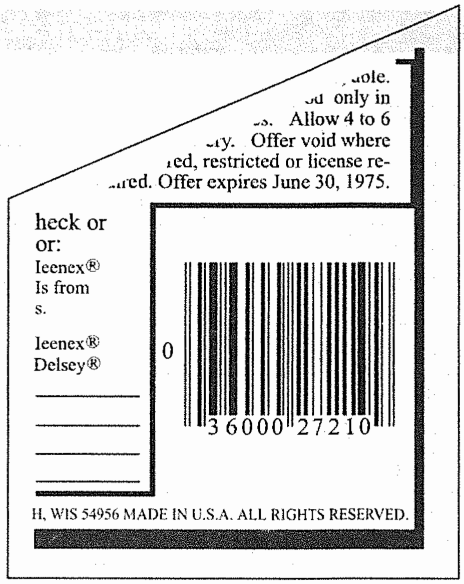
*Reprinted from *Digital Computer Fundamentals* by Thomas C. Bartee. Copyright 1960, 1966 by McGraw-Hill, Inc. Used with permission of McGraw-Hill Book Company.

Fig. A6.2 **American Standard Code for Information Exchange**

Specifications

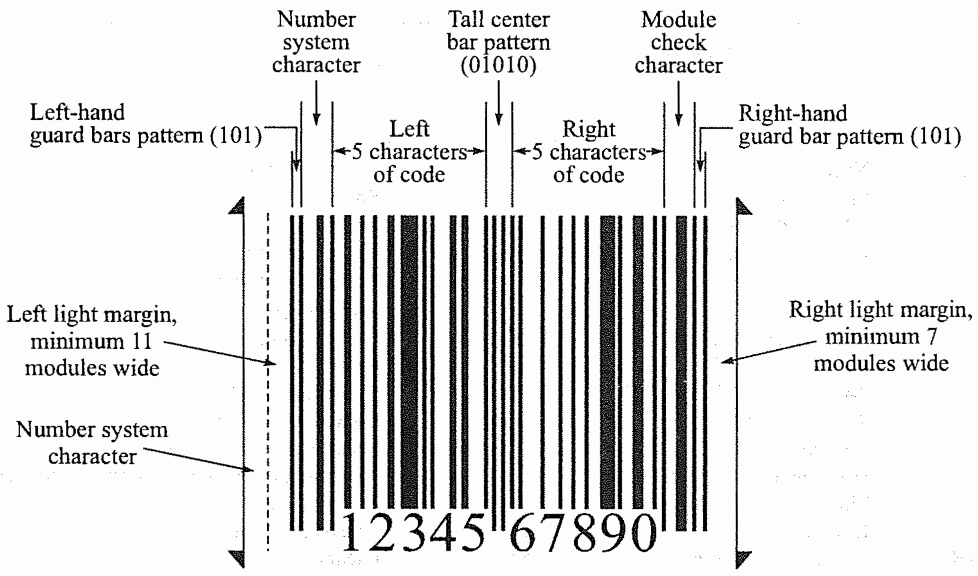
Each 10-digit symbol is rectangular in shape and consists of exactly 30 dark and 29 light vertical bars, as seen in Fig. A6.4. Each digit is represented by two dark bars and two light spaces. To account for the variable widths of the bars, each digit or character is broken down into seven modules. A module can be either dark or light, and each dark bar is made up of 1, 2, 3, or 4 dark modules. An example is shown in Fig. A6.5. Notice that each digit or character has exactly seven modules, and each digit has two dark bars and two light spaces. Dark modules are 1s while light modules are 0s.

Characters are encoded differently at the left and at the right of center. A left-side character begins with a light space and ends with a dark bar and always consists of either three or five dark modules (odd parity). A right-side character begins with a dark bar and ends with a light space and always has either two or four dark modules (even parity). The encoding for each character is summarized in Fig. A6.6.



, ..ole.
..d only in
... Allow 4 to 6
..y. Offer void where
..ed, restricted or license re-
..red. Offer expires June 30, 1975.
heck or
or:
leenex®
ls from
s.
leenex®
Delsey®
0
3 6000 27210
H, WIS 54956 MADE IN U.S.A. ALL RIGHTS RESERVED.

Fig. A6.3 UPC symbol from box of Kleenex tissues. Registered trademark of Kimberly-Clark Corp., Neenah, Wis



Number system character
Tall center bar pattern (01010)
Module check character
Left-hand guard bars pattern (101)
Left 5 characters of code
Right 5 characters of code
Right-hand guard bar pattern (101)
Left light margin, minimum 11 modules wide
Right light margin, minimum 7 modules wide
Number system character
12345 67890

Fig. A6.4 UPC standard symbol

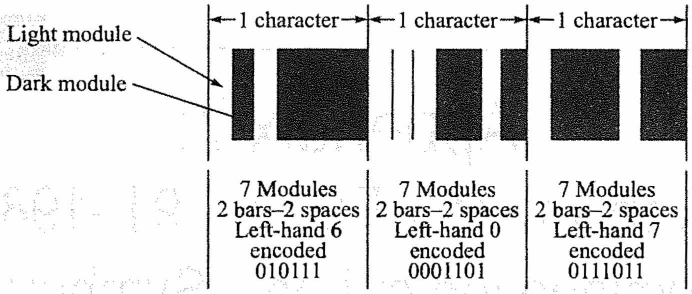


Fig. A6.5 UPC character construction

Decimal number	Left characters (odd parity)	Right characters (even parity)
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	0011100
5	0110001	0001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

Fig. A6.6 UPC character encoding

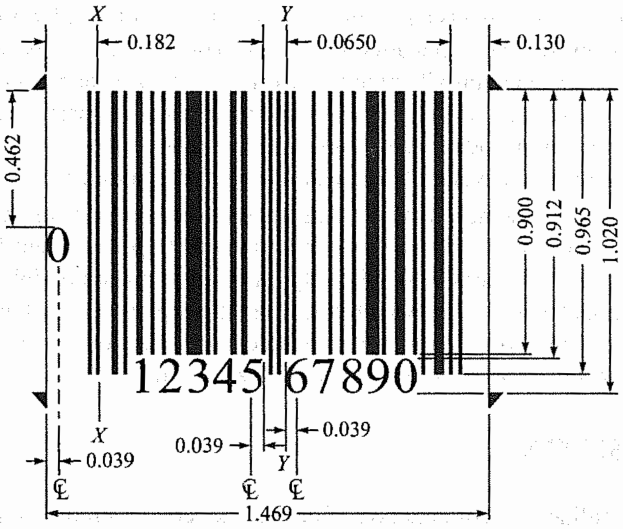


Fig. A6.7 UPC symbol dimensions

Appendix 7: Overview of IEEE Std. 91–1984, Explanation of Logic Symbols*

INTRODUCTION

The International Electrotechnical Commission (IEC) has been developing a very powerful symbolic language that can show the relationship of each input of a digital logic circuit to each output without showing explicitly the internal logic. At the heart of the system is dependency notation.

The system was introduced in the United States in a rudimentary form in IEEE/ANSI Standard Y32.14-1973. Lacking at that time a complete development of dependency notation, it offered little more than a substitution of rectangular shapes for the familiar distinctive shapes for representing the basic functions of AND, OR, negation, etc. This is no longer the case.

Internationally, Working Group 2 of IEC Technical Committee TC-3 has prepared a new document (Publication 617-12) that consolidates the original work started in the mid-1960s and published in 1972 (Publication 117-15) and the amendments and supplements that have followed. Similarly, for the USA, IEEE Committee SCC 11.9 has revised the publication IEEE Std 91/ANSI Y32.14. Now numbered simply IEEE Std 91-1984, the IEEE standard contains all of the IEC work that has been approved, and also a small amount of material still under international consideration. Texas Instruments is participating in the work of both organizations and this document introduces new logic symbols in accordance with the new standards. When changes are made as the standards develop, future editions will take those changes into account.

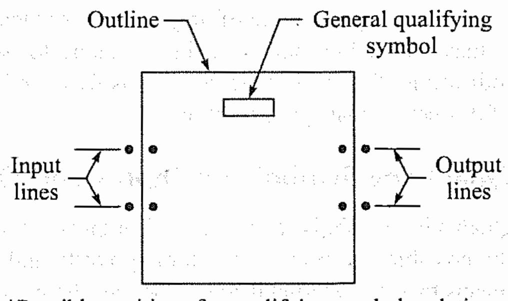
The following explanation of the new symbolic language is necessarily brief and greatly condensed from what the standards publications will contain. This is not intended to be sufficient for those people who will be developing symbols for new devices. It is primarily intended to make possible the understanding of the symbols used in this data book and is somewhat briefer than the explanation that appears in several of Texas Instruments data books on digital logic. However, it includes a new section that explains several symbols for actual devices in detail. This has proven to be a powerful learning aid.

SYMBOL COMPOSITION

A symbol comprises an outline or a combination of outlines together with one or more qualifying symbols. The shape of the symbols is not significant. As shown in Fig. A7.1, general qualifying symbols are used to

*Courtesy of Texas Instruments Incorporated.

tell exactly what logical operation is performed by the elements. Table A7.1 shows general qualifying symbols defined in the new standards. Input lines are placed on the left and output lines are placed on the right. When an exception is made to that convention, the direction of signal flow is indicated by an arrow.



*Possible positions for qualifying symbols relating to inputs and outputs

Fig. A7.1 **Symbol composition**

QUALIFYING SYMBOLS*

General Qualifying Symbols

Table A7.1 shows general qualifying symbols defined by IEEE Standard 91. These characters are placed near the top center or the geometric center of a symbol or symbol element to define the basic function of the device represented by the symbol or of the element.

Table A7.1 **General Qualifying Symbols**

Symbol	Description	
&	AND gate or function.	
≥ 1	OR gate or function. The symbol was chosen to indicate that at least one active input is needed to activate the output.	
= 1	Exclusive OR. One and only one input must be active to activate the output.	
1	The one input must be active.	
▷ or ◁	A buffer or element with more than usual output capability (symbol is oriented in the direction of signal flow).	
⎍	Schmitt trigger; element with hysteresis.	
X/Y	Coder, code converter, level converter.	
	The following are examples of subsets of this general class of qualifying symbol:	
	BCD/7-SEG	BCD to 7-segment display driver.
	TTL/MOS	TTL to MOS level converter.
	CMOS/PLASMA DISP	Plasma-display driver with CMOS-compatible inputs.
	MOS/LED	Light-emitting-diode driver with MOS-compatible inputs.
	CMOS/VAC FLUOR DISP	Vacuum-fluorescent display driver with CMOS-compatible inputs.
	CMOS/EL DISP	Electroluminescent display driver with CMOS-compatible inputs.
	TTL/GAS DISCH DISPLAY	Gas-discharge display driver with TTL-compatible inputs.
SRGm	Shift register. m is the number of bits	

* Symbols on pages 639 through 641 can be found in Texas Instruments, *High-Speed CMOS Logic Data Book*, pp. 6-5 to 6-10.

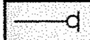
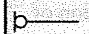
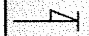
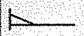
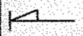
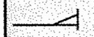
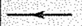
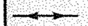
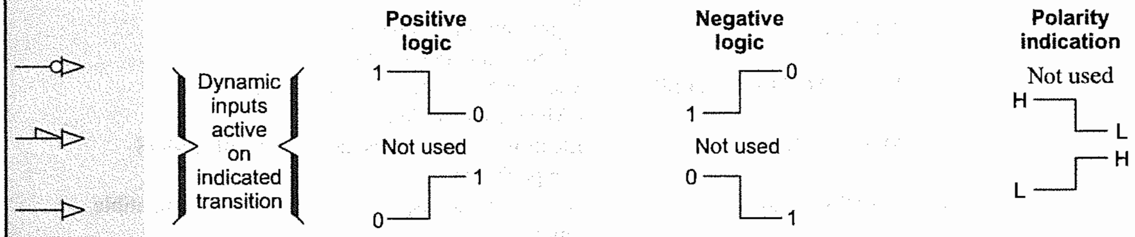
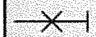
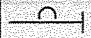
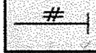
X/Y is the general qualifying symbol for identifying coders, code converters, and level converters. X and Y may be used in their own right to stand for some code or either or both may be replaced by some other indication of the code or level such as BCD or TTL. As might be expected, interface circuits often make use of this set of qualifying symbols.

Qualifying Symbols for Inputs and Outputs

Qualifying symbols for inputs and outputs are shown in Table A7.2 and will be familiar to most users with the possible exception of the logic polarity and analog signal indicators. The older logic negation indicator means that the external 0 state produces the internal 1 state. The internal 1 state means the active state. Logic negation may be used in pure logic diagrams; in order to tie the external 1 and 0 logic states to the levels H (high) and L (low), a statement of whether positive logic (1 = H, 0 = L) or negative logic (1 = L, 0 = H) is being used is required or must be assumed. Logic polarity indicators eliminate the need for calling out the logic convention and are used in the symbology for actual devices. The presence of the triangle polarity indicator indicates that the L logic level will produce the internal 1 state (the active state) or that, in the case of an output, the internal 1 state will produce the external L level. Note how the active direction of transition for a dynamic input is indicated in positive logic, negative logic, and with polarity indication.

When nonstandardized information is shown inside an outline, it is usually enclosed in square brackets [like these]. The square brackets are omitted when associated with a nonlogic input, which is indicated by an X superimposed on the connection line outside the symbol.

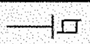
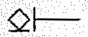
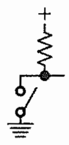
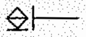
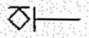
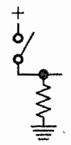
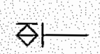
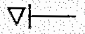
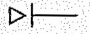
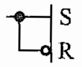
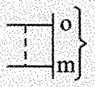
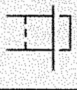
Table A7.2 Qualifying Symbols for Inputs and Outputs

	Logic negation at input. External 0 produces internal 1.
	Logic negation at output. Internal 1 produces external 0.
	Active-low input. Equivalent to ——○ in positive logic.
	Active-low output. Equivalent to ○—— in positive logic.
	Active-low input in the case of right-to-left signal flow.
	Active-low output in the case of right-to-left signal flow.
	Signal flow from right to left. If not otherwise indicated, signal flow is from left to right.
	Bidirectional signal flow.
 Positive logic Negative logic Polarity indication Dynamic inputs active on indicated transition 1 0 Not used H L	
	Nonlogic connection. A label inside the symbol will usually define the nature of this pin.
	Input for analog signals (on a digital symbol).
	Input for digital signals (on an analog symbol).

Symbols Inside the Outline

Table A7.3 shows some symbols used inside the outline. Note particularly that open-collector (open-drain), open-emitter (open-source), and three-state outputs have distinctive symbols. Also note that an EN input affects all of the outputs of the element and has no effect on inputs. An EN input affects all the external outputs of the element in which it is placed, plus the external outputs of any elements shown to be influenced by that element. It has no effect on inputs. When an enable input affects only certain outputs, affects outputs located outside the indicated influence of the element in which the enable input is placed, and/or affects one or more inputs, a form of dependency notation will indicate this. The effects of the EN input on the various types of outputs are shown.

Table A7.3 Symbols Inside the Outline

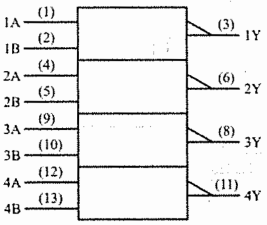
	Bi-threshold input (input with hysteresis).	
	*npn* open-collector or similar output that can supply a relatively low-impedance L level when not turned off. Requires external pull-up. Capable of positive-logic wired-AND connection.	
	Passive-pull-up output is similar to *npn* open-collector output but is supplemented with a built-in passive pull-up.	
	*npn* open-emitter or similar output that can supply a relatively low-impedance H level when not turned off. Requires external pull-down. Capable of positive-logic wired-OR connection.	
	Passive-pull-down output is similar to *npn* open-emitter output but is supplemented with a built-in passive pull-down.	
	3-state output.	
	Output with more than usual output capability (symbol is oriented in the direction of signal flow).	
—┤EN	Enable input	
	When at its internal 1-state, all outputs are enabled.	
	When at its internal 0-state, open-collector, open-emitter, and three-state outputs are at external high-impedance state, and all other outputs (i.e., totem-poles) are at the internal 0-state.	
J, K, R, S, T	Usual meanings associated with flip-flops (e.g., R = reset, T = toggle).	
—┤D	Data input to a storage element equivalent to: 	
—┤→m —┤←m	Shift right (left) inputs, m = 1, 2, 3, etc. If m = 1, it is usually not shown.	
	Binary grouping, m is highest power of 2. Produces a number equal to the sum of the weights of the active inputs.	
	Input line grouping . . . indicates two or more terminals used to implement a single logic input, e.g., differential inputs.	

It is particularly important to note that a D input is always the data input of a storage element. At its internal 1 state, the D input sets the storage element to its 1 state, and at its internal 0 state it resets the storage element to its 0 state.

The binary grouping symbol is explained more fully in a later section. Binary-weighted inputs are arranged in order and the binary weights of the least-significant and the most-significant lines are indicated by numbers. In this document weights of input and output lines will be represented by powers of 2 usually only when the binary grouping symbol is used, otherwise decimal numbers will be used. The grouped inputs generate an internal number on which a mathematical function can be performed or that can be an identifying number for dependency notation. This number is the sum of the weights (1, 2, 4 ... , $2n$) of those inputs standing at their 1 states. A frequent use is in addresses for memories.

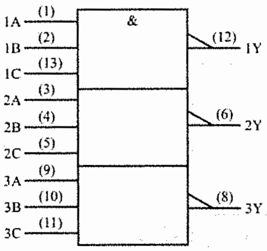
Reversed in direction, the binary grouping symbol can be used with outputs. The concept is analogous to that for the inputs, and the weighted outputs will indicate the internal number assumed to be developed within the circuit.

Appendix 8:
Pinout Diagrams



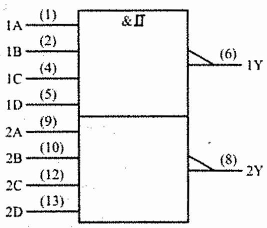
Positive logic: $Y = \overline{AB}$

00



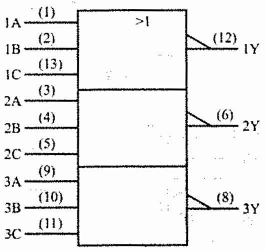
Positive logic: $Y = \overline{ABC}$

10



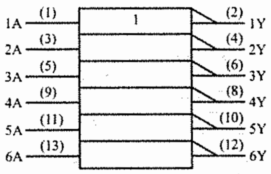
Positive logic: $Y = \overline{ABCD}$

13



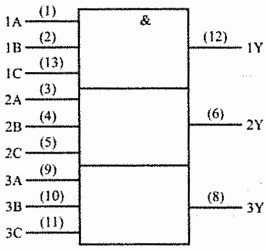
Positive logic: $Y = \overline{A+B+C}$

27



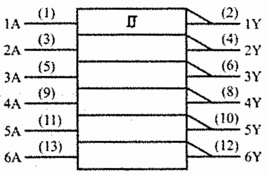
Positive logic: $Y = \overline{A}$

04



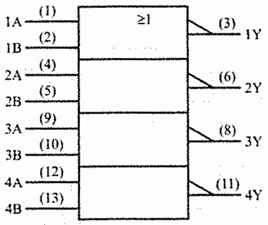
Positive logic: Y = ABC

11



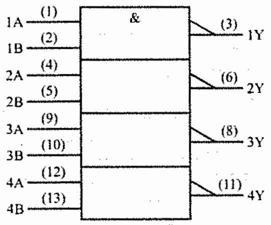
Positive logic: $Y = \overline{A}$

14



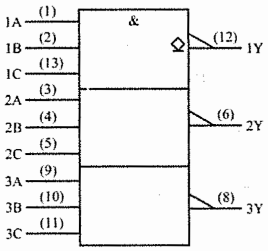
Positive logic: Y = A+B

32



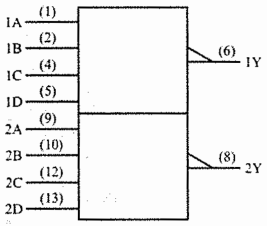
Positive logic: Y = AB

08



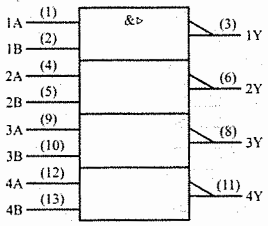
Positive logic: $Y = \overline{ABC}$

12



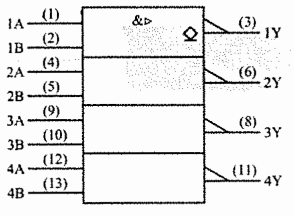
Positive logic: $Y = \overline{ABCD}$

20



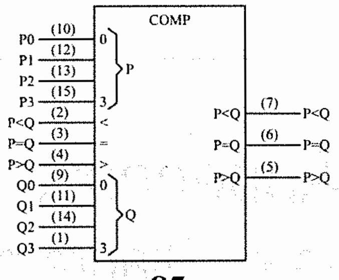
Positive logic: $Y = \overline{AB}$

37

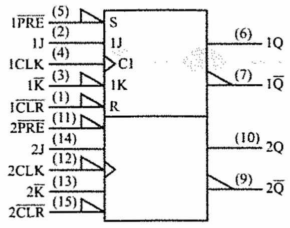


Positive logic: $Y = \overline{AB}$

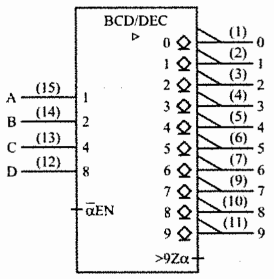
38



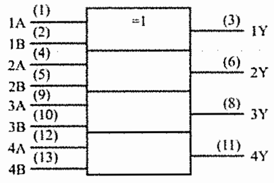
85



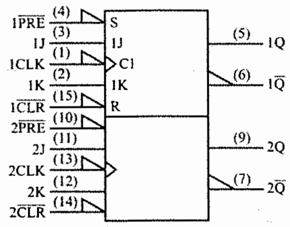
109



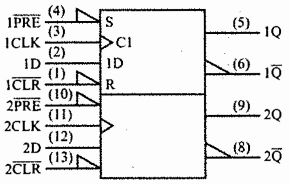
45



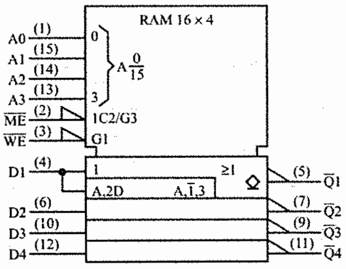
86



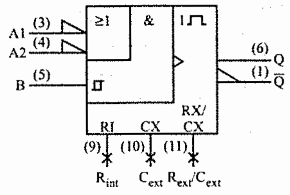
112



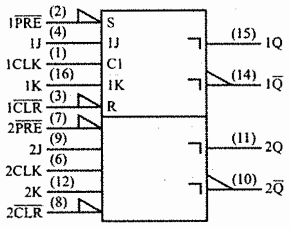
74



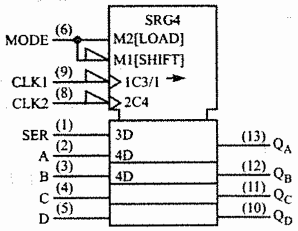
89



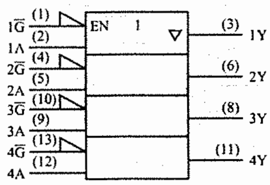
121



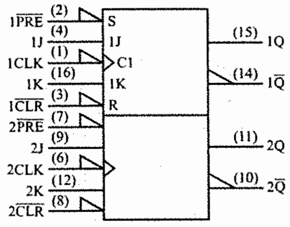
76



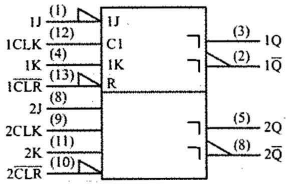
95



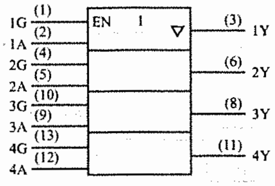
125



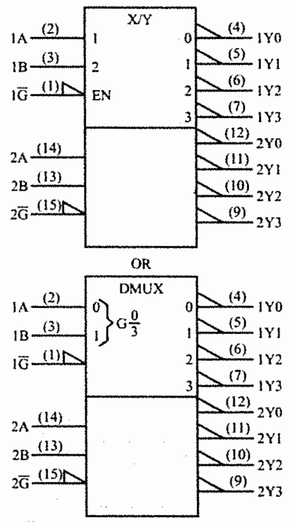
LS76A



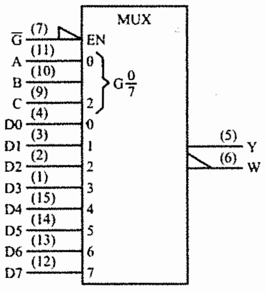
107



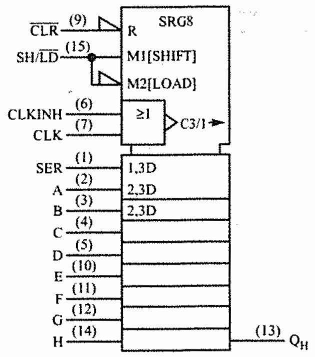
126



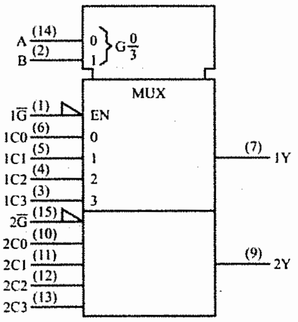
139



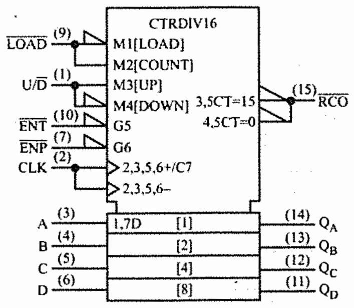
151



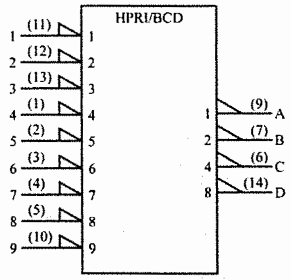
166



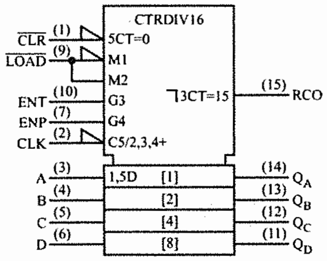
153



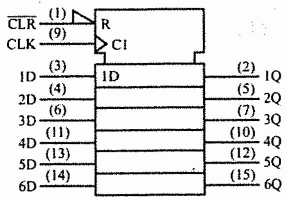
169



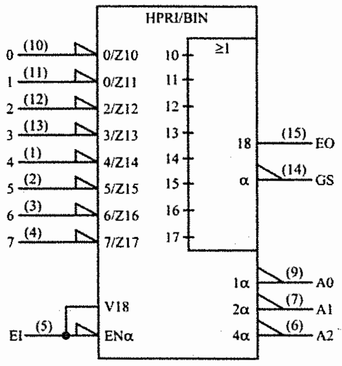
147



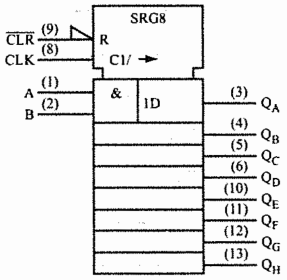
163



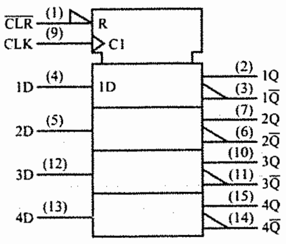
174



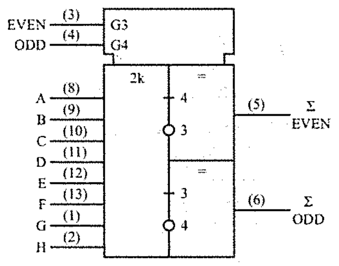
148



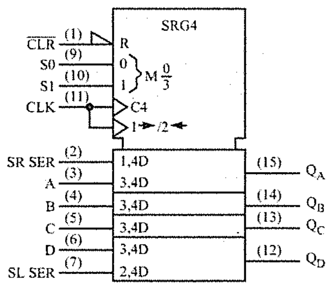
164



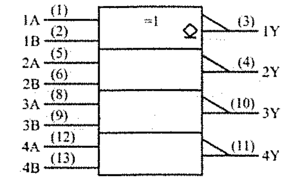
175



180

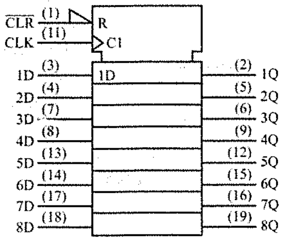


194

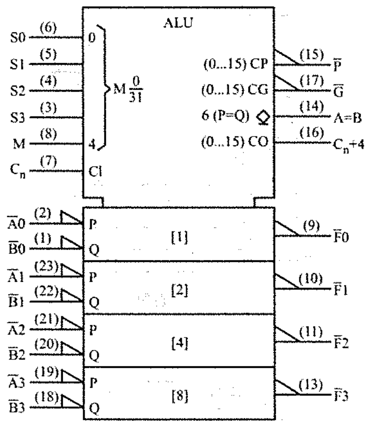


Positive logic: $Y = \overline{A \oplus B} = AB + \overline{A}\overline{B}$

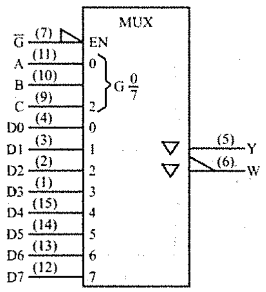
266



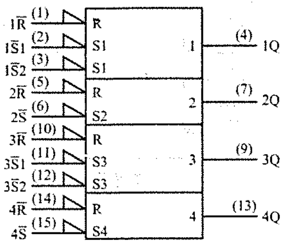
273



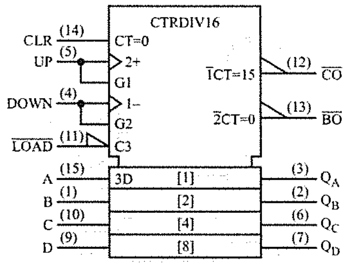
181



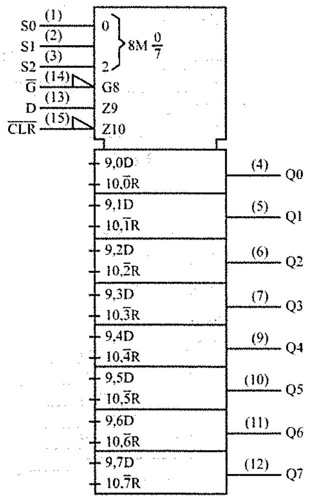
251



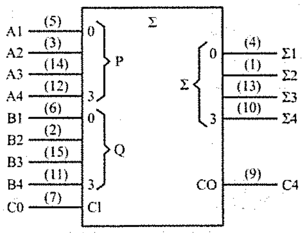
279



193



259



283

Appendix 9: Answers to Selected Odd-Numbered Problems

Chapter 1

1.1 See Sec. 1.1.

1.3

Lamps	*	0	1	0	1	0	1	0	1
	*	0	0	1	1	0	0	1	1
	*	0	0	0	0	1	1	1	1
	No.	0	1	2	3	4	5	6	7

1.5 See Fig. 1.4c.

1.7 $t_H = 100\ \mu s$

1.9 The nonideal waveform is nearly triangular in shape!

1.11

V_1	L	H	L	H	L	H	L	H
V_2	L	L	H	H	L	L	H	H
V_3	L	L	L	L	H	H	H	H
V_0	L	H	H	H	H	H	H	H

1.13 G must be high to have a signal at V_o. $G = H$ and $V_i = L$.

1.15 Simply connect the output of the first 4-bit register to the input of the second 4-bit register.

1.17 32 connections. 16 connections. True parallel shifting requires a single operation of 250 ns. Shifting 16 bits twice requires 500 ns.

1.19 Nine; seven

1.21 $F = 0100$; CARRY OUT = 0

1.23 Only line 9 is high.

1.25 Handshaking is a request to transfer data into or out of the computer. It is a request to transfer data, followed by an acknowledge, allowing data transfer to begin.

1.27 Because data to be operated on is taken from memory into the CPU and results are moved back to memory for storage.

1.29 54LSXX

1.31 2.5 W

1.33 See Figs. 1.36, 1.37, and 1.38.

Chapter 2

2.1 Low; high

2.3 The truth table is:

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

This is the truth table of a 3-input OR gate. Therefore, a cascade of two 2-input OR gates is equivalent to a 3-input OR gate.

2.5 The truth table is

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

2.7 The truth table is

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

2.9 The truth table is

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

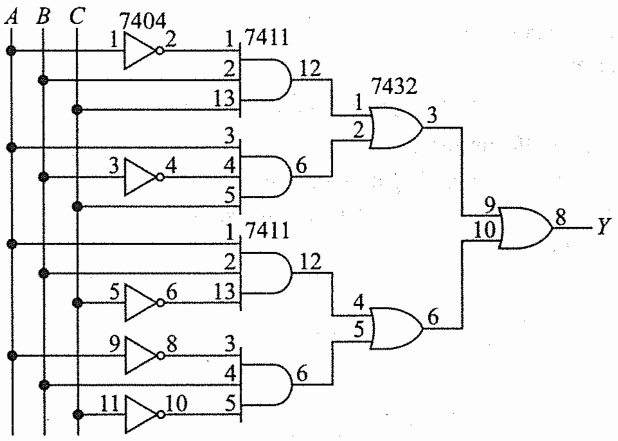
2.11 The Boolean equations are

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

2.13 The Boolean equations are

$$Y = ABC$$
$$Y = \overline{AB}$$
$$Y = \overline{ABC}$$

2.15 The logic circuit.



2.17 Here is a summary of the truth table. Y equals 1 when $ABCD = 0000$; Y equals 0 for all other $ABCD$ inputs. There are 16 $ABCD$ inputs, starting with 0000 and ending with 1111.

2.19 d

2.21 $Y = \overline{A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7}$

2.23 a

2.25 a. Low b. High c. High d. Low

2.27 a. 1 b. 0 c. 0 d. 0

2.29 Active-low: b., c., d., and g.; active-high: a., e. and f.

Chapter 3

3.1 Draw an AND-OR circuit with two AND gates and one OR gate. The upper AND gate has inputs of A, $\overline{B}$, and C. The lower AND gate has inputs of A, B, and C. The simplified logic circuit is an AND gate with inputs of A and C.

3.3 The lower input gate

3.5 d

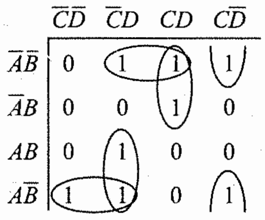
3.7 $Y = \overline{A}\,\overline{C}D + A\overline{B}C + AB\overline{C}$, which means an AND-OR circuit that ORs the foregoing logical products.

3.9 $Y = \overline{A}B + A\overline{B}$, which implies an AND-OR circuit that ORs the foregoing products.

3.11 Figure (a) shows the Karnaugh map.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	1	1
$\bar{A}B$	0	0	1	0
AB	0	1	0	0
$A\bar{B}$	1	1	0	1

(a)



(b)

3.13 Fig. 3.16a

3.15 Figure (b) shows the Karnaugh map.

3.17 The simplified equation is

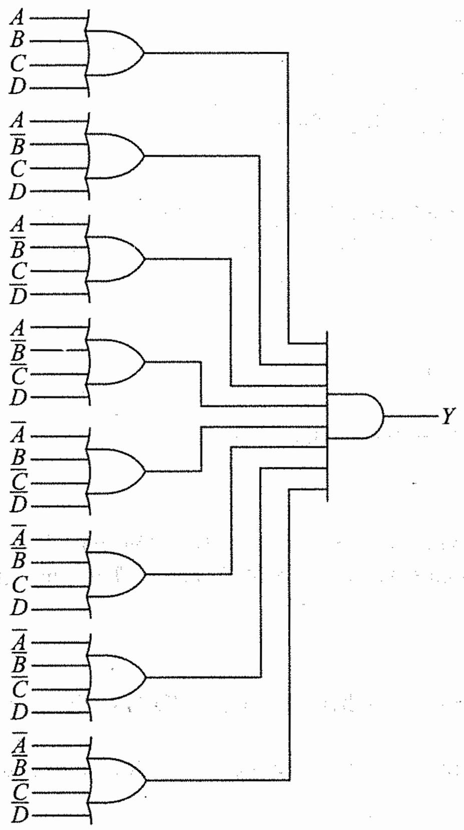
$$Y = \bar{A}\,\bar{B}D + \bar{A}CD + A\bar{B}\,\bar{C} + A\bar{C}D + \bar{B}C\bar{D}$$

The corresponding AND-OR circuit has five AND gates driving an OR gate.

3.19 The simplified logic circuit is an AND gate with inputs of $\bar{A}$, $\bar{C}$, and D.

3.21 $Y = AB + AC$; use an AND-OR circuit to produce this equation.

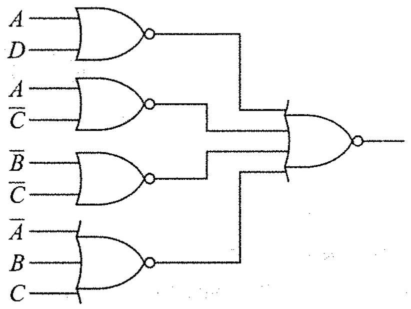
3.23 The unsimplified logic circuit.


A B C D
A $\bar{B}$ C D
A $\bar{B}$ $\bar{C}$ $\bar{D}$
A $\bar{B}$ $\bar{C}$ D
$\bar{A}$ B $\bar{C}$ $\bar{D}$
$\bar{A}$ $\bar{B}$ C D
$\bar{A}$ $\bar{B}$ $\bar{C}$ D
$\bar{A}$ $\bar{B}$ $\bar{C}$ $\bar{D}$
Y

3.25 $Y = F(A, B, C, D) = \Sigma\, m(1, 2, 8, 9, 10, 12, 13, 14)$

3.27 The map and the circuit.

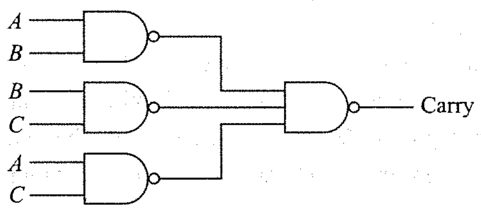
	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	0	1	1
$\overline{A}B$	1	0	1	1
AB	0	0	1	1
$A\overline{B}$	1	1	0	0



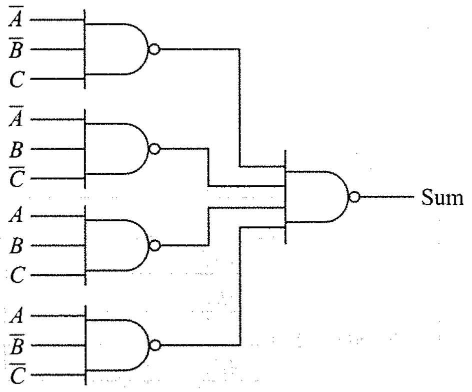
3.29 The Y waveform is low between 0 and 7. Then, it is high between 7 and 16.

3.31 The simplified NAND-NAND circuits.

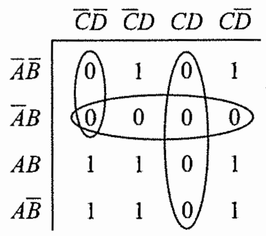
	$\overline{C}$	C
$\overline{A}\overline{B}$	0	0
$\overline{A}B$	0	1
AB	1	1
$A\overline{B}$	0	1



	$\overline{C}$	C
$\overline{A}\overline{B}$	0	1
$\overline{A}B$	1	0
AB	0	1
$A\overline{B}$	1	0



3.33



	$\bar C\bar D$	$\bar C D$	CD	$C\bar D$
$\bar A\bar B$	0	1	0	1
$\bar A B$	0	0	0	0
AB	1	1	0	1
$A\bar B$	1	1	0	1

$Y = (A + \bar B)(\bar C + \bar D)(A + C + D)$

3.35 $Y = AC' + AD' + B'C'D + B'CD'$

3.37 SOP: $Y = A'B' + AC' + B'C'$ and POS: $Y = (A + B')(A' + C')(B' + C')$

Chapter 4

4.1 Y equals D_9.

4.3 Connect the data inputs as follows: +5 V—D_0, D_4, D_5, D_6, D_{11}, D_{12}, D_{14}, and D_{15}; ground—D_1, D_2, D_3, D_7, D_8, D_9, D_{10}, and D_{13}.

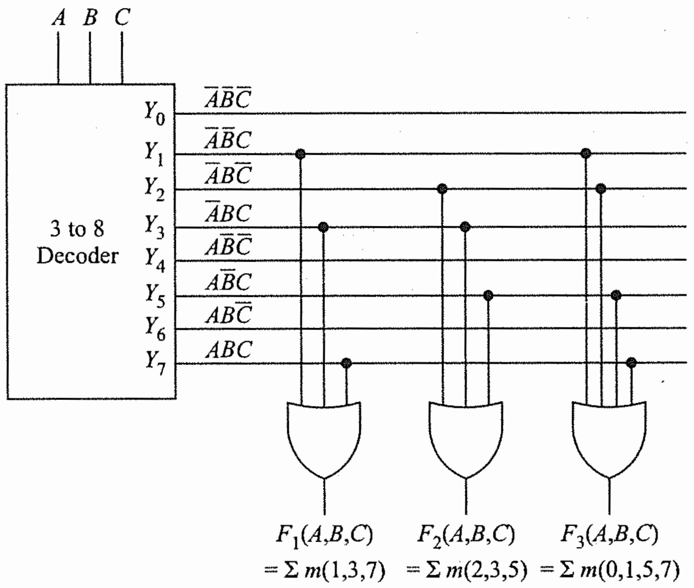
4.5 Y_3 multiplexer: ground D_1, D_6, D_7, and D_{14}; all other data inputs high.
Y_2 multiplexer: ground D_3, D_8, and D_{13}, all other data inputs high.
Y_1 multiplexer: ground D_0, D_1, D_{14}, and D_{15}, all other data inputs high.
Y_0 multiplexer: ground D_8, D_9, and D_{13}, all other data inputs high.

4.7 None; Y_5

4.9 c

4.11 The chip on the right; Y_6

4.13



A B C
3 to 8 Decoder
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7
$\bar A\bar B\bar C$
$\bar A\bar B C$
$\bar A B\bar C$
$\bar A BC$
$A\bar B\bar C$
$A\bar B C$
$AB\bar C$
ABC
$F_1(A,B,C) = \Sigma\, m(1,3,7)$
$F_2(A,B,C) = \Sigma\, m(2,3,5)$
$F_3(A,B,C) = \Sigma\, m(0,1,5,7)$

4.15 a. 67 b. 813 c. 7259

4.17 Y_7

4.19 c

4.21 Approximately 3 mA

4.23 Pin 5; 0111

4.25 a. 0 b. 1 c. 0 d. 1

4.27 a. 0 b. 1 c. 0

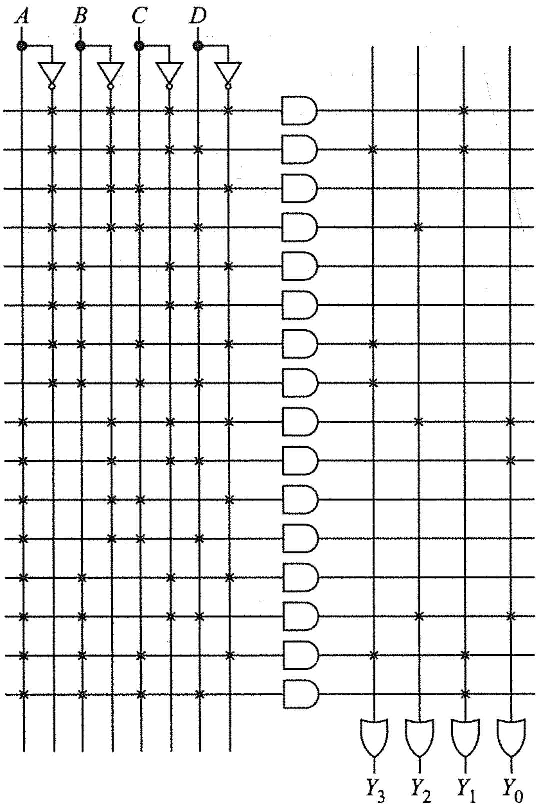
4.29 $(X > Y) = G_3 + E_3G_2 + E_3E_2G_1 + E_3E_2E_1G_0$

4.31 Ground pin 4 (after disconnecting from +5 V) and connect pin 3 to +5 V (after disconnecting from ground).

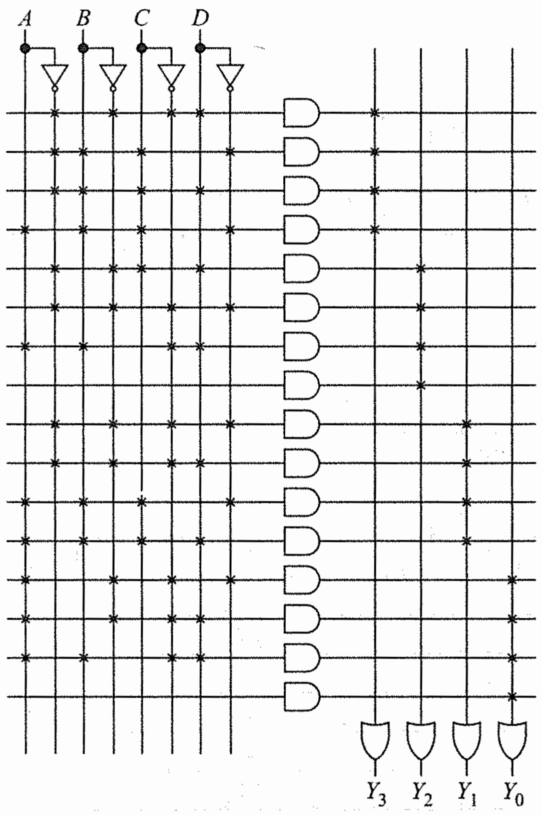
4.33 256

4.35 1100

4.37 The PROM.



4.39 The PAL circuit.



4.41 3; 9; 15 (illegal)

Chapter 5

5.1	01110000			
5.3	a. 1	b. 2	c. 3	d. 4
5.5	a. 188	b. 255		
5.7	131,072			
5.9	100001100			
5.11	1010010100000			
5.13	011 010 101.111 011 110			
5.15	504.771			
5.17	a. 257	b. 15.331	c. 123.55	
5.19	a. 1110 0101	b. 1011 0100 1101	c. 0111 1010 1111 0100	
5.21	12,121			
5.23	a. 0000	b. 0100	c. 1010	d. 1111
5.25	a. 011 0111	b. 101 0111	c. 110 0110	d. 111 1001

5.27

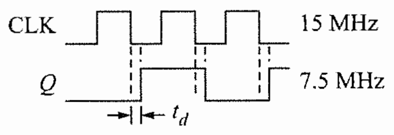
Address	Alphanumeric	Hex contents
2000	G	C7
2001	O	4F
2002	O	4F
2003	D	C4
2004	B	C2
2005	Y	D9
2006	E	45

5.29 3694
5.31 0001 0001
5.33 a, b, and d
5.35 e
5.37 11100001111
5.39 Five for both (a) and (b)

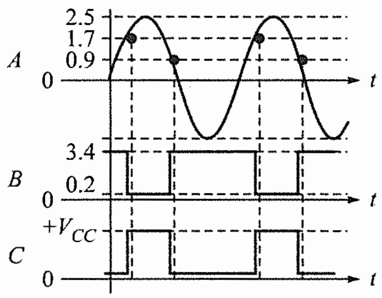
Chapter 6

6.1 a. 12_8 b. 13_8 c. 10_{16} d. 17_{16}
6.3 0000 0101 0000 1000
6.5 0001 1000
6.7 a. 0001 0111 b. 0111 1011 c. 1011 1000 d. 1110 1011
6.9 a. DCH b. BAH c. 36H d. O2H
6.11 a. 0100 1110 b. 1110 1001 c. 1010 0110 d. 1000 0111
6.13 a.
$$\begin{array}{r} 0010\ 1101 \\ +\ 0011\ 1000 \\ \hline 0110\ 0101 \end{array}$$
b.
$$\begin{array}{r} 0101\ 1001 \\ +\ 1101\ 1110 \\ \hline 0011\ 0111 \end{array}$$
c.
$$\begin{array}{r} 0100\ 0011 \\ +\ 1001\ 1110 \\ \hline 1110\ 0001 \end{array}$$
6.15 02CBH, 0000 0010 1100 1011
6.17 Binary 0101 0011, or decimal 83
6.19 $G_0 = 1.1 = 1, G_1 = 1.0 = 0, G_2 = 0.0 = 0, G_3 = 1.1 = 1.$
$P_0 = 1 + 1 = 1, P_1 = 1 + 0 = 1, P_2 = 0 + 0 = 0, P_3 = 1 + 1 = 1$ and $C_{-1} = 0.$
Substituting these in corresponding equations $C_0 = 1, C_1 = 1, C_2 = 0, C_3 = 1.$
Using $S_i = G_i \oplus P_i \oplus C_{i-1}$ $S_0 = 0, S_1 = 0, S_2 = 1, S_3 = 0.$
Final result : $C_3S_3S_2S_1S_0 = 10100$
6.21 Substitute $M = 0$ and $S_3..S_0 = 0110, C_{in} = 0, A = 1101$ and $B = 0111.$

Chapter 7

7.1 a. 100 ns b. 167 ns c. 1.33 µs
7.3 13.3 MHz
7.5 0.45/4.05
7.7 3.5 MHz plus or minus 28 Hz
7.9


7.11



7.15 48 kHz. $t_1 = 13\ \mu s$, $t_2 = 7.8\ \mu s$

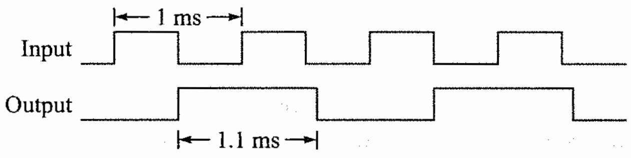
7.17 33.3 percent, 37.5 percent

7.19 $R_A + R_B = 15\ k\Omega$. $R_A = 3.75\ k\Omega$, $R_B = 11.25\ k\Omega$

7.21 3.88 ms

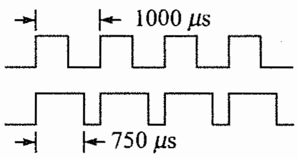
7.23 0.136 μF

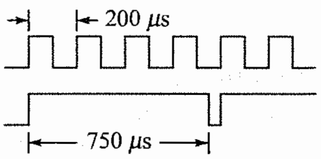
7.25



7.27 Connect as in Example 7.8. 21.3 nF.

7.29 a. b.

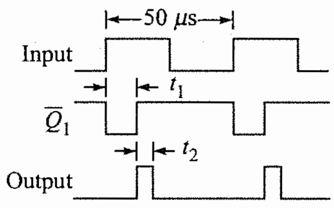




7.31 Connect $\overline{A}_1$ to GND and apply input to B_2. $C = 44.6$ nF.

7.33 Same as Prob. 7.29.

7.35



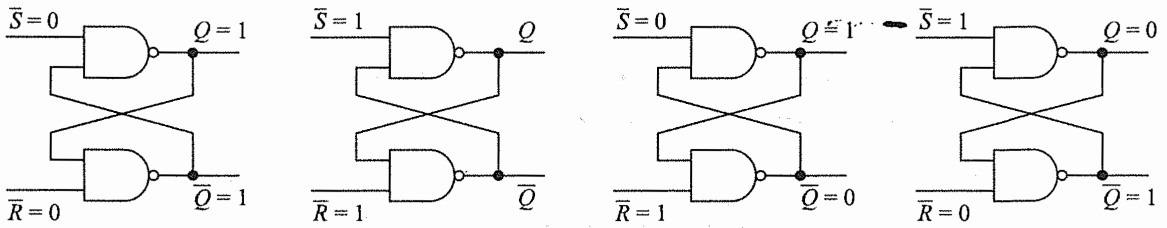
7.37 Let $R_1 = R_2 = 1\ k\Omega$.

a. $t_1 = 2.5\ \mu s$, $t_2 = 7.5\ \mu s$, $C_1 = 7500$ pF, $C_2 = 22{,}500$ pF.

b. $t_1 = t_2 = 1\ \mu s$, $C_1 = C_2 = 3000$ pF.

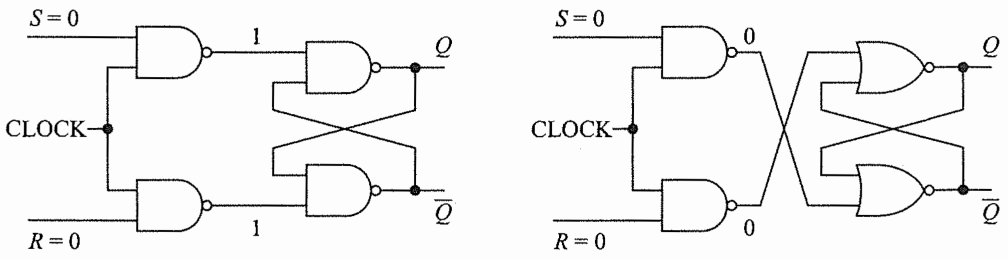
Chapter 8

8.3



8.5 a. C b. G

8.7 When the clock is low, the flip-flop is insensitive to levels on either R or S input. (Only first case is shown here.)



8.9 The R and S inputs do not need to be held static while the clock is high.

8.11 Use negative-edge-triggering.

t_0: S is low, R is high

t_1: S is high, R is low

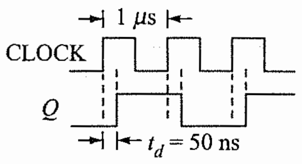
t_2 and after: S is low, R is high.

After t_2, both R and S can be low.

8.13 Low

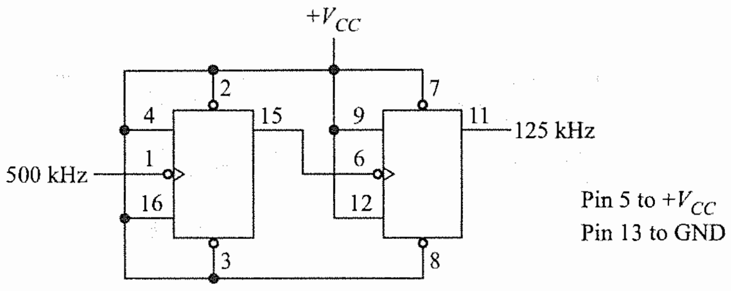
8.15 a. 5 ns b. 10 ns c. 15 ns

8.17



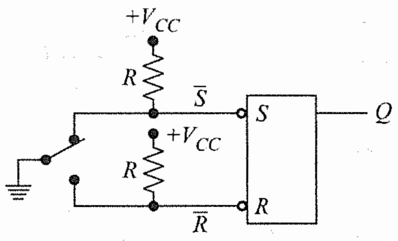
8.19 Clock period = 1 μs. Period of $Q = 2\ \mu$s ($f = 500$ kHz)

8.21



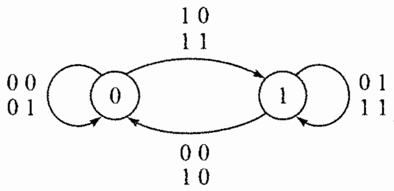
8.23 The pulse symbol shows that the flip-flop is pulse-triggered.

8.25



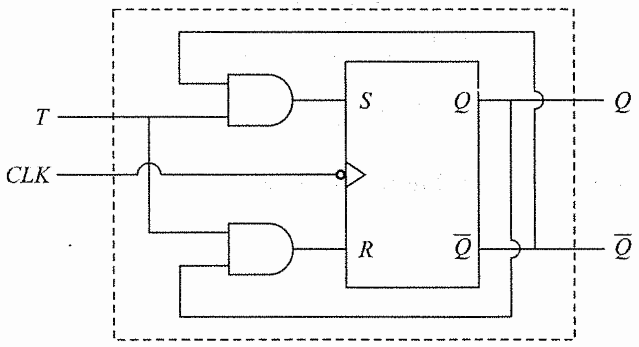
8.27 (a) $Q_{n+1} = BQ_n + AQ_n'$

(b)



8.29 This is a modulo-3 counter with state sequence 00→01→10→00... and corresponding output, Y changes as 1→0→0→1...

8.31

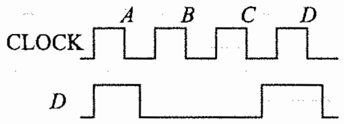


Chapter 9

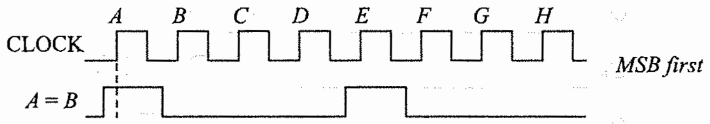
9.1 a. 6 b. 6 c. 4

9.3 See Fig. 9.1

9.5



9.7

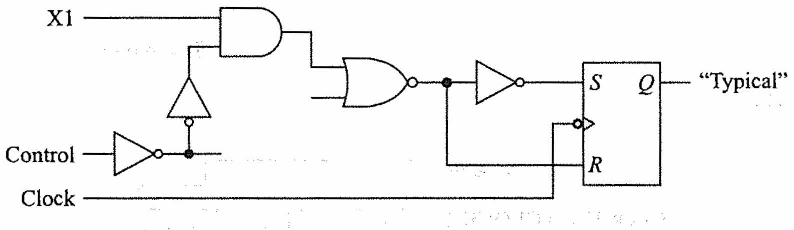


9.9 a. 8 μs b. 1.6 μs

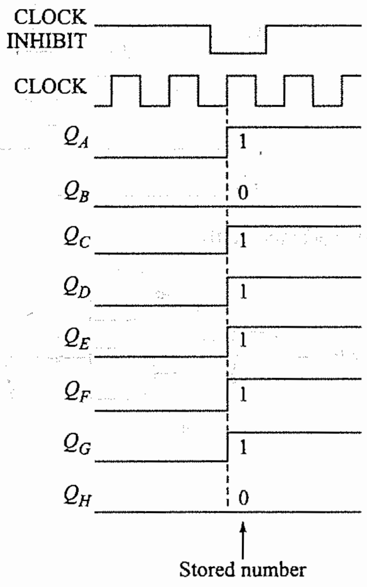
9.11 16.7 MHz

9.13 a. $R = 1, S = 0, Q = 0$ b. $R = 0, S = 1, Q = 1$

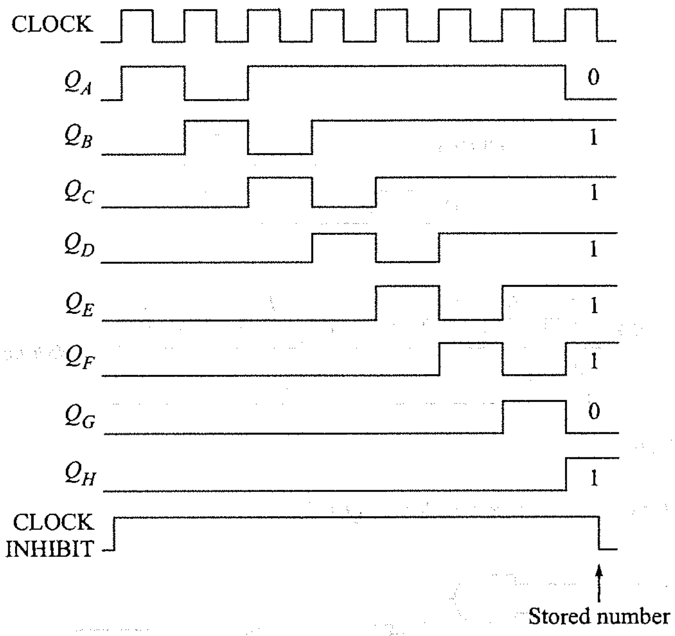
9.15



9.17 MSB first, shift/load is low, *ABCD EFGH* = 1011 1110.

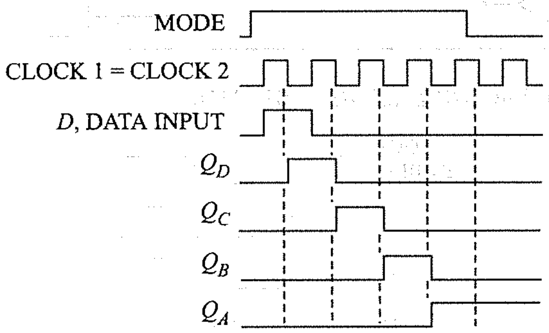


MSB first, shift/load is high.

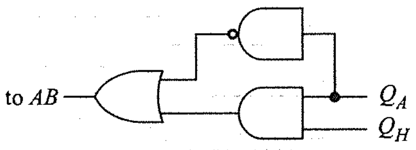


9.19 Same as 9.15.

9.21

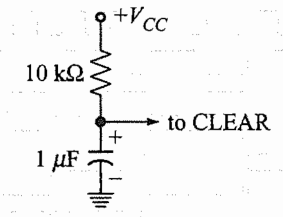


9.23 For alternate 1s and 0s, replace feedback with:



(a)

Include a power-on-CLEAR circuit like:



(b)

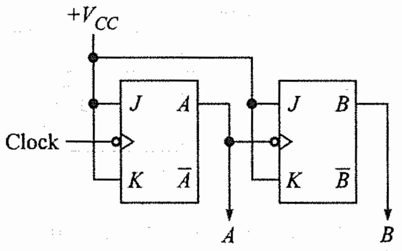
This will CLEAR all flip-flops to zeros when power is first applied.

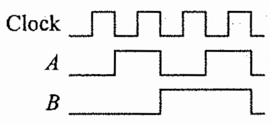
9.25 Decoder output $Y = R'S'$

Clock	*Serial in* $= T'$	Q	R	S	T	$Y = R'S'$
0	0	1	0	0	1	1
1	1	0	1	0	0	0
2	1	1	0	1	0	0
3	0	1	1	0	1	0
4	1	0	1	1	0	0
5	0	1	0	1	1	0
6	0	0	1	0	1	0
7	1	0	0	1	0	0
8	0	1	0	0	1	1
9	1	0	1	0	0	0
						repeats

Chapter 10

10.1





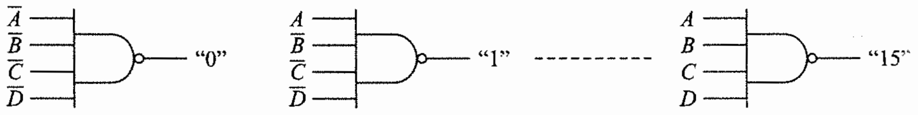
Clock	B	A
0	0	0
1	0	1
2	1	0
3	1	1
0	0	0

10.3 4 MHz

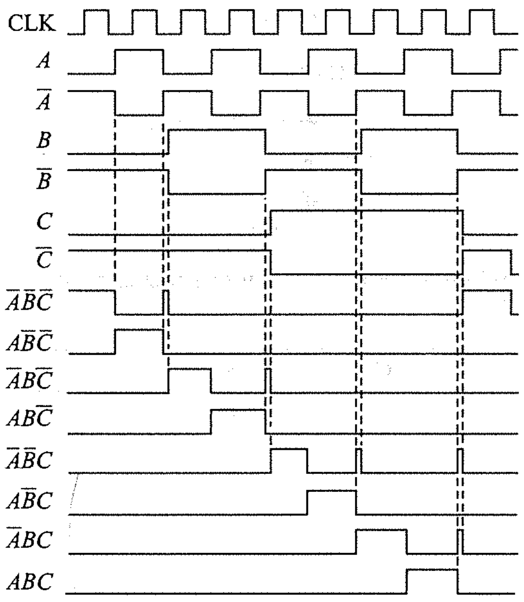
10.5 Difficult to get clock and first few flip-flop outputs on the same page with the last few flip-flop outputs.

10.7 Same as Fig. 10.3 where period of QB is 2 μs.

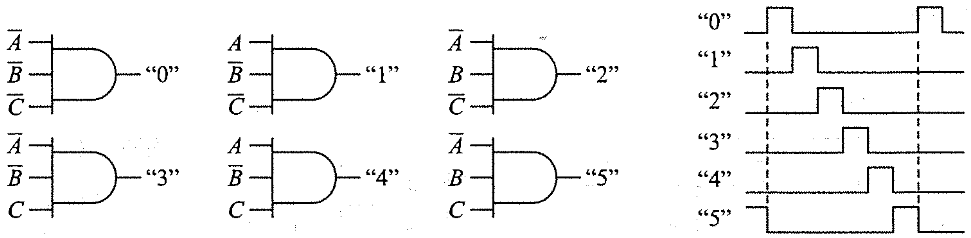
10.9 Sixteen 4-input NAND-gates_with inputs, $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, $A\overline{B}\,\overline{C}\,\overline{D}$, $\overline{A}B\overline{C}\,\overline{D}$, $AB\overline{C}\,\overline{D}$, $\overline{A}\,\overline{B}C\overline{D}$, $A\overline{B}C\overline{D}$, $\overline{A}BC\overline{D}$, $ABC\overline{D}$, $\overline{A}\,\overline{B}\,\overline{C}D$, $A\overline{B}\,\overline{C}D$, $\overline{A}B\overline{C}D$, $AB\overline{C}D$, $\overline{A}\,\overline{B}CD$, $A\overline{B}CD$, $\overline{A}BCD$, and $ABCD$.



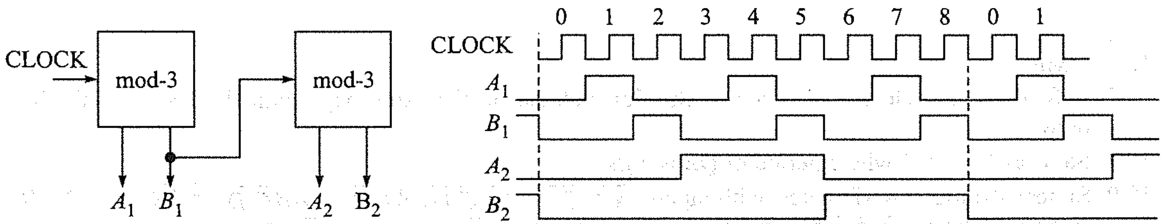
10.11


CLK
A
$\overline{A}$
B
$\overline{B}$
C
$\overline{C}$
$\overline{A}\overline{B}\overline{C}$
$A\overline{B}\overline{C}$
$\overline{A}B\overline{C}$
$AB\overline{C}$
$\overline{A}\overline{B}C$
$A\overline{B}C$
$\overline{A}BC$
ABC

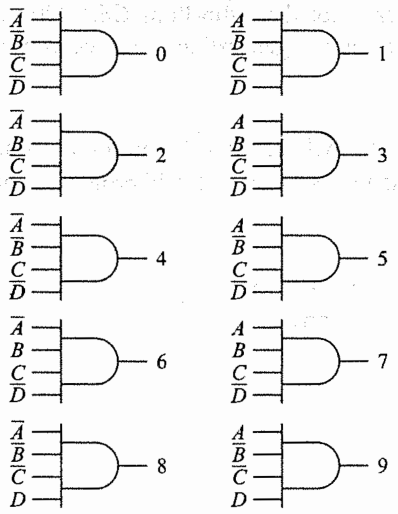
10.13 Same as Fig. 10.12c, except transitions occur on low-to-high clock.
10.15 As in Fig. 10.15.
10.17 a. 3 b. 4 c. 4 d. 5 e. 5
10.19


$\overline{A}$ $\overline{B}$ $\overline{C}$ "0"
A $\overline{B}$ $\overline{C}$ "1"
$\overline{A}$ B $\overline{C}$ "2"
$\overline{A}$ $\overline{B}$ C "3"
A $\overline{B}$ C "4"
$\overline{A}$ B C "5"
"0"
"1"
"2"
"3"
"4"
"5"

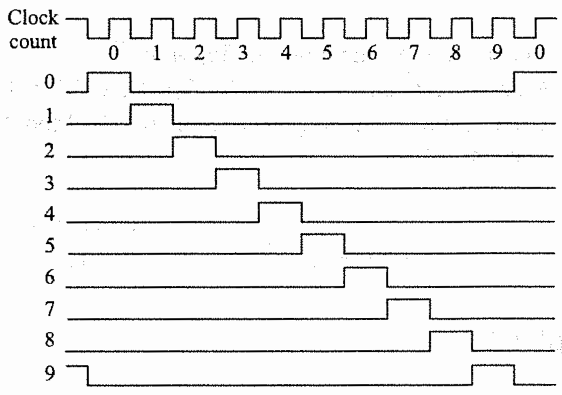
10.21



10.23

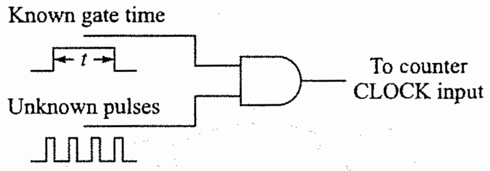


Decoding gates

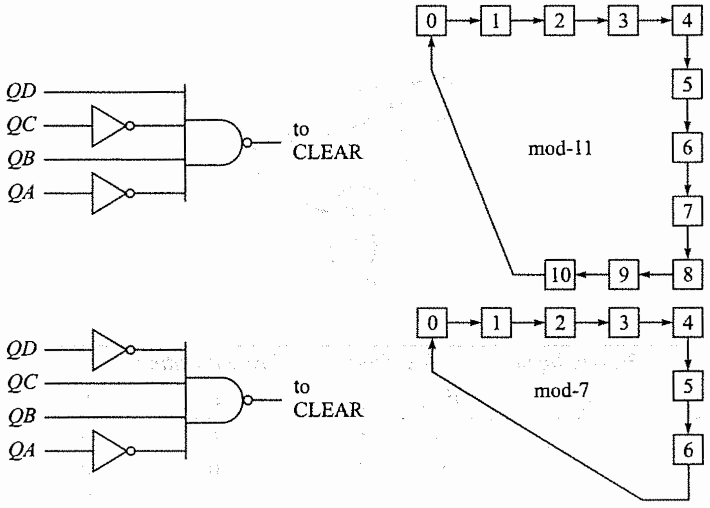


Decode-counter outputs

10.25



10.27



10.29 Like Prob. 10.27.

10.31 Draw the circuit from design equations $D_A = A \otimes B$, $D_B = A' + B$

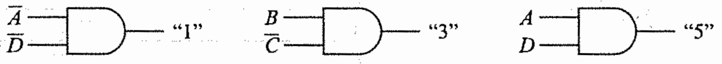
10.33 Draw the circuit from design equations $J_A = B + C'$, $K_A = B' + C'$; $J_B = A$, $K_B = A$; $J_C = AB$, $K_C = A'B'$

10.35 Design a modulo-6 counter (Section 10.7) with state sequence for three flip-flops *CBA* 000 → 001 → 010 → 011 → 100 → 101 → 000... Then draw circuit for output generating sequence as $Y = C + A'B'$

10.39 Reconnect the *J* input on flip-flop *A* to $\overline{D}$.

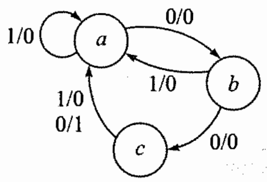
10.41 Mod-5 illegal states are 2(010), 5(101), and 7(111). AND gate will detect $\overline{A}B$ and force counter to $CB\overline{A}$. Count 7 will progress to count 6(110). Mod-3 illegal state is 3(11), which will progress naturally to count 2(10).

10.43

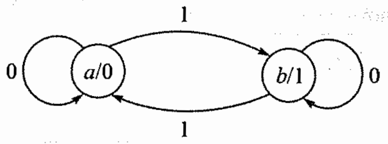


Chapter 11

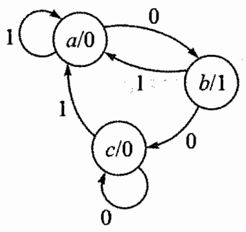
11.1



11.3



11.5



11.7

Present State		*Present Input*	*Next State*		*Present Output*				
B_n	A_n	X_n	B_{n+1}	A_{n+1}	Y_n	J_B	K_B	J_A	K_A
0	0	0	0	0	0	0	×	0	×
0	0	1	0	1	0	0	×	1	×
0	1	0	0	0	0	0	×	×	1
0	1	1	1	1	0	1	×	×	0
1	1	0	0	0	1	×	1	×	1
1	1	1	1	1	0	×	0	×	0

11.9

$X \backslash B_nA_n$	00	01	11	10
0	0	0	×	×
1	0	1	×	×

$J_B = XA_n$

$X \backslash B_nA_n$	00	01	11	10
0	×	×	1	×
1	×	×	0	×

$K_B = \overline{X}$

$X \backslash B_nA_n$	00	01	11	10
0	0	×	×	×
1	1	×	×	×

$J_A = X$

$X \backslash B_nA_n$	00	01	11	10
0	×	1	1	×
1	×	0	0	×

$K_A = \overline{X}$

$X \backslash B_nA_n$	00	01	11	10
0	0	0	1	×
1	0	0	0	×

$Y = XB_n$

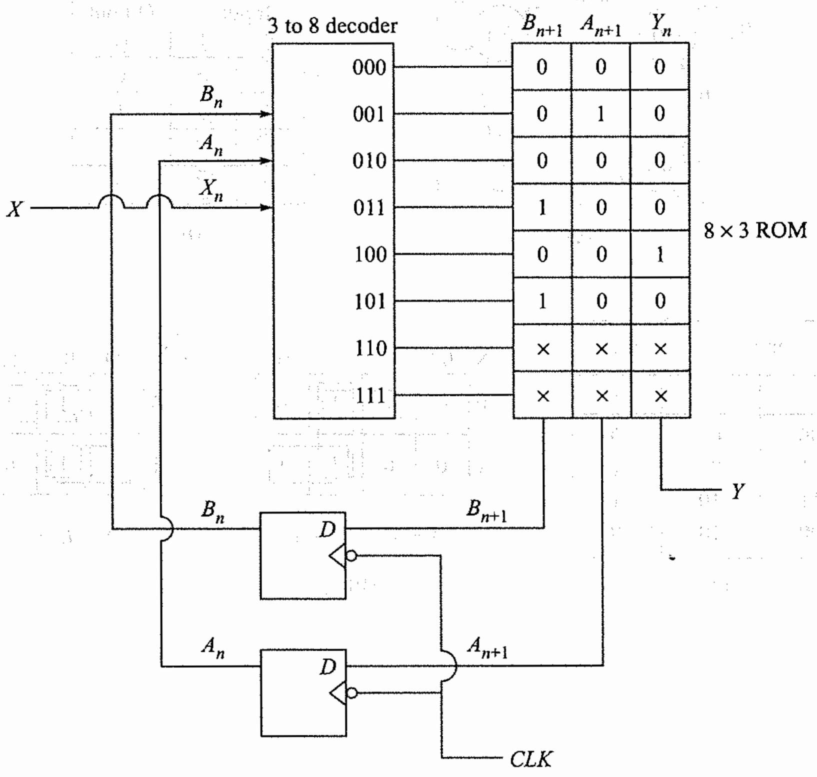
Compared to solution given in Section 11.4, this requires one AND gates less for J_A input.

11.11 Two for Mealy and three for Moore circuit.

11.13 Two flip-flops (B and A) are required. Three states assigned as (BA) 00, 01, 11 representing no, 1st, 2nd bit detection respectively. Then,

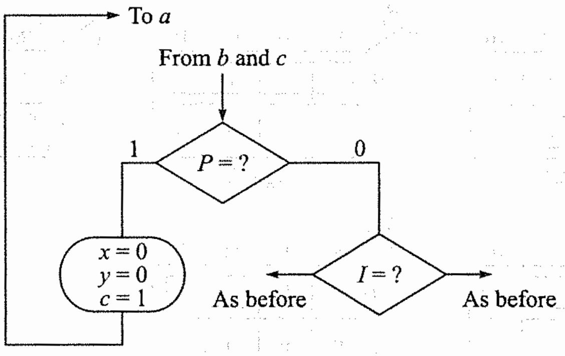
$$D_B = X'B'A \qquad D_A = X'B' \qquad \text{Output } Y = X'BA$$

11.15

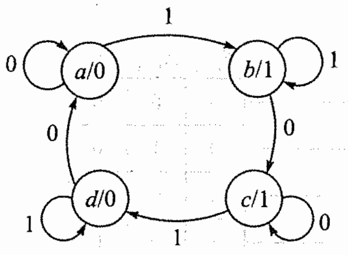


Decoder output	B_{n+1}	A_{n+1}	Y_n
000	0	0	0
001	0	1	0
010	0	0	0
011	1	0	0
100	0	0	1
101	1	0	0
110	×	×	×
111	×	×	×

11.17



11.19 Three.
11.21 Current output is same as previous output fed back at the input side.
11.23 Stable states $Axy = 010, 100, 101, 111$.
11.25 $ABC = 110, 101, 011$.
11.27 When the circuit moves between two transient states following a particular input transition.
11.29 10110, 11010, 00010.
11.31



(a)

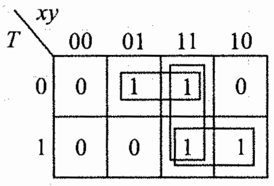
	Input T 0	Input T 1	Output Q
a	ⓐ	b	0
b	c	ⓑ	1
c	ⓒ	d	1
d	a	ⓓ	0

(b)

11.33

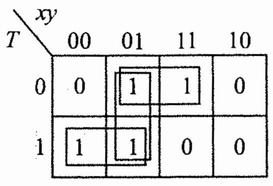
	Input T		Output
xy	0	1	Z
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

(a)



$X = \bar{T}y + Tx + xy$

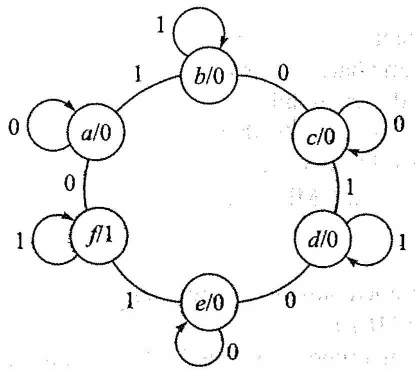
(b)



$Y = T\bar{x} + \bar{T}y + \bar{x}y$

(c)

11.35



11.37

Present state	Input A		Output
(xy)	0	1	Z
a (10)	$\textcircled{a}$	b	1
b (00)	–	c	1
c (01)	–	d	0
d (11)	a	$\textcircled{d}$	0

Chapter 12

12.1 $\frac{1}{63}, \frac{2}{63}, \frac{4}{63}, \frac{8}{63}, \frac{16}{63}, \frac{32}{63}$

12.5 51.2 mA

12.7 a. 0.641 V b. 0.923 V c. 0.766 V

12.9 1 part in 4096; 2.44 mV

12.11 31

12.13 $A_2A_1A_0$ = a. 001 b. 010 c. 110

12.15 7 MHz

12.17 12 μs, not counting delay and control times.

12.19 $i_R = i_C$, $V_i/R = V_oC/t$, $\therefore V_o = \dfrac{V_i}{RC} \times t$

12.21 0.01 μF

12.25 They should all be comparable

Chapter 13

13.1 a. RAM b. ROM c. EPROM

13.3 Loss of power results in loss of memory.
13.5 RAM
13.7 ROM data storage is permanent.
13.9 For accuracy check. Read immediately after write.
13.11 Data is recorded and retrieved sequentially.
13.13 One-half tape length = 14,400 in; 14,400/300 = 48 s.
13.15 Maximum 52×150 KB/Sec = 7.8 MB/Sec.
13.17 Maximum 8×1.32 MB/Sec = 10.56 MB/Sec.
13.19 $DCBA = 1101$
13.21 Five flip-flop binary counter.
13.23 A PROM that can be programmed only by the supplier.
13.25 a. Apply address $FEDCBA = 110101$.
b. Apply a current pulse, one at a time to outputs Y_1, Y_2, Y_6, and Y_8.
13.27 $P = 2F + 1$

F	P	P(Binary)			
0	1	0	0	0	1
1	3	0	0	1	1
2	5	0	1	0	1
3	7	0	1	1	1
4	9	1	0	0	1

13.29 Two identical chips connected together as follows:
a. Select inputs (A to A, B to B, C to C, D to D)
b. Data inputs (D_1 to D_1, D_2 to D_2, D_3 to D_3, D_4 to D_4)
c. Sense outputs (S_1 to S_1, S_2 to S_2, S_3 to S_3, S_4 to S_4)
Now, *ME* and *WE* are used to select one chip or the other.
13.31 You must accomplish the following:
READ: a. *ME* goes low for a given time period.
b. The SELECT inputs (address) must be stable while *ME* is low.
Then DATA is valid at the outputs for the time shown in Fig. 13.22d.
WRITE: a. *ME* must be low for a given time period.
b. The select inputs (address) and the data inputs must be stable while *ME* is low.
c. *WE* must go low for a time t_w as in Fig. 13.22c.
13.33 Draw two circuits exactly like Fig. 13.25, one below the other. Now, connect:
a. The ADDRESS lines in parallel
b. The two *ME* lines together
c. The two *WE* lines together
The four DATA IN and DATA OUT lines from the upper circuit can be considered the 4 LSBs, and those from the lower circuit are the 4 MSBs.
13.35 150 ns, 100 ns
13.37 +5 Vdc, −5 Vdc, +12 Vdc
13.39 False
13.41 RAM chips

Chapter 14

14.1 a. 1.82 mA, 0.7 V   b. 0 mA, −10 V   c. 1.82 mA, 9.3 V
d. 1.65 mA   e. 0 mA, 0 V   f. 5.35 V

14.3

V_1	V_2
0	+5
+5	0

14.5 a. 1.28 mA   b. 0 mA   c. 1.28 mA   d. −10 Vdc
14.7 a. 1.2 mA   b. 0.91 V   c. Either case: $I = 0$ mA
14.9 Low-power Schottky; see Table 14.3.
14.11 100
14.13 20
14.15 Change 74LS04 to 7404.
14.17 30
14.19 a. 1   b. 1   c. 0   d. 0
14.21 a. 1   b. 0   c. 0   d. 0
14.23 Time constant = RC = (3.6 kΩ)(20 pF) = 72 ns
14.25 High, low, high impedance (open)
14.27 Low; high
14.29 High; low
14.31 With high TTL output, $I \approx 0$, ideally with TTL output 0.4 V,

$$I = \frac{12\text{ V} - 2\text{ V} - 0.4\text{ V}}{2.2\text{ k}\Omega} = 4.36\text{ mA}$$

14.33 300 ns
14.37 Low; high
14.39 Connecting pin 13 to the supply voltage will produce a permanent high input. This will force the output to stick in the low state, regardless of what values A and B have.
14.41 Grounding pin 1 will force the output to remain permanently in the high state, no matter what the values of A and B.
14.43 3.6 mA
14.45 Yes; 4 maximum
14.47 Time constant = RC = (2.2 kΩ)(10 pF) = 22 ns
14.49 Choice d, shorted sink transistor

Chapter 15

Solutions for the problems at the end of this chapter are not unique—that is, there are many different designs that will satisfy each requirement. Nevertheless, typical designs for each problem can be found in the literature and are therefore not included here. It is intended that you search application notes and other publications supplied by manufacturers in order to satisfy a particular design requirement. This will provide the opportunity to see numerous different applications, and at the same time challenge you to improve on existing logic configurations. Here are some suggested references:

Intersil, Inc., Cupertino, Calif.: *Data Sheets and Application Notes.*
Motorola Semiconductor Products, Inc., Technical Information Center, Phoenix, Ariz.: *Linear Integrated Circuits Data Book*, 1979.

National Semiconductor Corporation, Santa Clara, Calif.: *Data Acquisition Handbook*, 1978; *Linear Applications Handbook*, 1980.
Texas Instruments, Data Book Marketing, P.O. Box 225558, Dallas, TX 75222-5558:
Interface Circuits Data Book; *Linear Circuits Data Book* (3 volumes); *Optoelectronics and Image Sensing Data Book*; *TTL Logic Data Book*.

15.11 909 kHz

15.13 a. 0100 0000 b. 0011 0011 c. 1100 0001

15.15 +1.65 Vdc

15.17 $V_{i-} = +0.25$ Vdc; $V_{ref} = +2.375$ Vdc

Chapter 16

16.1 Five bits are available for data. Maximum number that can be loaded is $2^5-1 = 31$.

16.3 6-bit opcode gives $2^6 = 64$ different instructions and 7-bit opcode gives up to $2^7 = 128$ instructions. Thus we have to assign 7-bit for opcode and in turn need a 7-bit register for *IR*.

16.5 *B* is copied to *A* when any of *X* and *Y* is true.

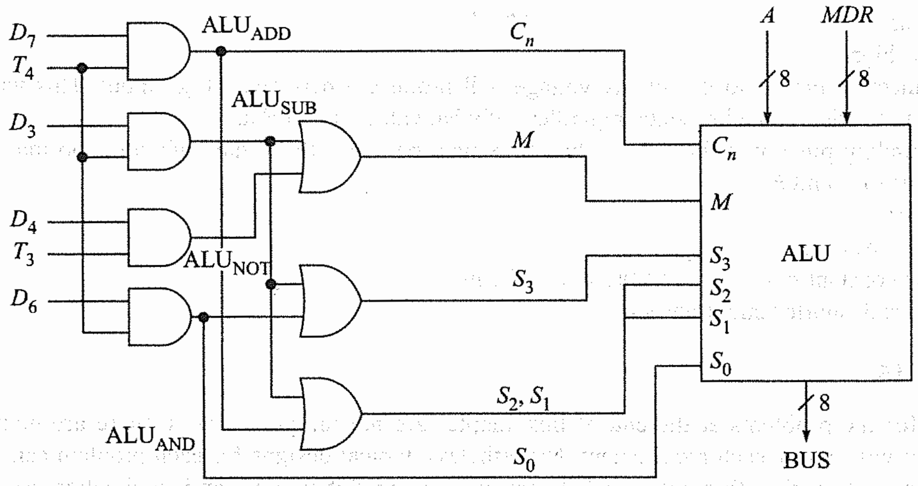
16.7 The bit positions where *ACC* and *MDR* were same will have 1 and rest 0.

16.9 $ACC[7:1] \leftarrow ACC[6:0], ACC[0] \leftarrow CY$

16.11 $D_0T_3 : ACC[4:0] \leftarrow MDR[4:0], ACC[7:5] \leftarrow 0, TC \leftarrow 0$
Parallel load should allow 0 to enter 3 MSB of *ACC* which otherwise receives all 8 bits from BUS. Three 2-to-1 multiplexer with D_0T_3 as selection can be used for this so that $D_0T_3 = 1$ selects 0 and $D_0T_3 = 0$ selects BUS.

16.13 Register *MAR* will be loaded with data available on BUS when TSD generates T_0 or T_2.

16.15



16.17 37 clock cycles.

16.19

Memory address in binary	*Memory content in binary*	*Memory content in hexadecimal*
00000	00001000	08
00001	11001001	C9
00010	00101010	2A
00011	10100000	A0
00100	10100000	A0
00101	11001010	CA
00110	00101010	2A
00111	01000000	40
01000	00000101	05
01001	00001000	08
01010	01000001	41
01011	UNUSED	UNUSED
....	.....	.....
11111	UNUSED	UNUSED

16.21

```
LDA    addr1
NOT
AND    addr2
NOT
STA    addr3
LDA    addr2
NOT
AND    addr1
NOT
AND    addr3
STA    addr3
HLT
```

Index