

Microprocessor Architecture, Programming, and Applications with the 8085

Fifth Edition

Ramesh S. Gaonkar

FREE CD ENCLOSED!
Book not returnable if software
has been removed.

Fifth Edition

Microprocessor Architecture, Programming and Applications with the 8085

Ramesh S. Gaonkar

Microprocessor Architecture, Programming, and Applications with the 8085 provides a comprehensive treatment of the microprocessor, covering both hardware and software based on the 8085 microprocessor family.

The fifth edition, divided into three parts, presents an integrated approach to hardware and software in the context of the 8085 microprocessor.

Part I focuses on microprocessor architecture, the 8085 instruction set, and interfacing, and Part II introduces programming. Part III integrates hardware and software concepts from the earlier sections in interfacing and designing microprocessor-based products. In-depth coverage of each topic is provided, from basic concepts to industrial applications, and illustrated by numerous examples with complete schematics.

Learning of the material is reinforced by practical-application assignments.

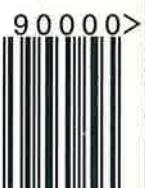
New and updated information includes:

- The most recent technological changes and a block diagram of the Microprocessor-Controlled Temperature System (MCTS).
- A programming model of the 8085 processor and its instruction set.
- Additional explanation and expansion of blocks in the MCTS.
- An additional interfacing application in the context of MCTS.
- The latest technological changes in 32- and 64-bit microprocessors.

A free Instructor's Manual (ISBN 0-13-034001-4) is available to instructors using this book for a course.

For more information about this book or any of Prentice Hall's other new technology titles, visit our website at <http://www.prenhall.com>

ISBN 0-13-019570-7



9 780130 195708

ATILIM UNIVERSITY LIBRARY



0000016924

Microprocessor Architecture, Programming, and Applications with the 8085

FIFTH EDITION

Ramesh S. Gaonkar
STATE UNIVERSITY OF NEW YORK,
O.C.C. CAMPUS AT SYRACUSE



ATILIM
UNIVERSITY
LIBRARY



*Upper Saddle River, New Jersey
Columbus, Ohio*

Library of Congress Cataloging-in-Publication Data

Gaonkar, Ramesh S.

Microprocessor architecture, programming, and applications with the 8085 / Ramesh S.

Gaonkar. — 5th ed.

p. cm.

Includes index.

ISBN 0-13-019570-7 (alk. paper).

1. Intel 8085 (Microprocessor). I. Title.

QA76.8.I2912 G36 2002

004.165—dc21

2001051377

Editor in Chief: Stephen Helba

Assistant Vice President and Publisher: Charles E. Stewart, Jr.

Assistant Editor: Delia K. Uhrec

Production Editor: Alexandrina Benedicto Wolf

Design Coordinator: Diane Ernsberger

Cover Designer: Ali Mohrman

Cover Art: Corbis Stock Market

Production Manager: Matthew Ottenweller

This book was set in Times Roman by The Clarinda Company and was printed and bound by R. R. Donnelley & Sons Company. The cover was printed by Phoenix Color Corp.

QA76.8

GAO

2002

ATLANTIC

UNIVERSITY

LIBRARY

16924



Pearson Education Ltd., London

Pearson Education Australia Pty. Limited, Sydney

Pearson Education Singapore, Pte. Ltd.

Pearson Education North Asia Ltd., Hong Kong

Pearson Education Canada, Ltd., Toronto

Pearson Educación de Mexico, S.A. de C.V.

Pearson Education-Japan, Tokyo

Pearson Education Malaysia, Pte. Ltd.

Pearson Education, Upper Saddle River, New Jersey

Copyright ©2002, 1999, 1996, 1989, 1984 by Pearson Education, Inc., Upper Saddle River, New Jersey 07458. All rights reserved. Printed in the United States of America. This publication is protected by Copyright and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permission(s), write to: Rights and Permissions Department.



10 9 8 7 6 5 4 3 2 1

ISBN 0-13-019570-7

Preface

This book was first published in 1984, and it has been in the field for the last eighteen years. The microprocessor concepts that were at the cutting edge of the technology in the 1970s and '80s have become fundamentals of the computer field. It is gratifying to see such acceptance of the integrated approach to teaching microprocessor concepts. The text is intended for introductory microprocessor courses at the undergraduate level in technology and engineering. It is a comprehensive treatment of the microprocessor, covering both hardware and software based on the 8085 microprocessor family. The text assumes a course in digital logic as a prerequisite; however, it does not assume any background in programming. At the outset, though, we need to answer the following three critical questions.

1. In the early years of the twenty-first century, is an 8-bit microprocessor an appropriate device through which to teach microprocessor concepts when 32- and 64-bit microprocessors are readily available? If we consider the worldwide sales volume of microprocessor chips, the answer is a resounding yes: 8-bit microprocessors (including single-chip microcontrollers) account for more than 90 percent of the total. The 8-bit microprocessor has already established its market in the areas of industrial control, such as machine control, process control, instrumentation, and consumer appliances; these systems that include a microprocessor are known as embedded systems or microprocessor-based products. The recent 32- and 64-bit microprocessors are used primarily in microcomputers

and workstations; they are so powerful that their applications are better suited to such tasks as high-speed data processing, CAD/CAM, multitasking, and multiuser systems. The 32- or 64-bit microprocessors are less likely to replace 8-bit microprocessors in industrial control applications.

From the teaching point of view, we are interested in teaching the basic concepts underlying a programmable device, such as buses, machine cycles, various processes of data flow (parallel, serial, interrupts, and DMA), internal register architecture, programming, and interfacing. A general-purpose 8-bit microprocessor is an ideal device to teach these concepts, especially in a rapidly changing technological environment. When students master the basic concepts, they will be able to apply those concepts in such an environment, whether it is based on a microcontroller, an 8-bit processor with a different set of instructions, or a 64-bit processor.

2. Why shouldn't we focus on the Intel high-end 32- or 64-bit processors when PCs (personal computers) are commonly available in college laboratories? This is similar to asking why shouldn't we use LSI devices to teach basic logic concepts of AND, NAND, and OR. To teach basic concepts, we need a simple processor with an adequate instruction set. The Intel high-end processors are too difficult to comprehend at the introductory level because of their complex architecture and large instruction set. They are suitable for high-level languages and handling large databases and graphics. These processors are used primarily in PCs and network servers.

3. Why teach the 8085 microprocessor? This question has several answers. One is that any 8-bit microprocessor that is commonly available will meet the teaching criteria, and another is that the 8085 is one of the most widely used microprocessors in college laboratories. It has simple architecture and an adequate instruction set that enable instructors to teach necessary programming concepts. It is inconsequential which microprocessor is selected as the focus; the concepts are easily transferable from one device to another. Having learned basic concepts with the 8085 microprocessor, students can adapt to the microcontroller environment (such as the Intel 8051 or Motorola 68HC11) or to the PC environment. Furthermore, peripheral devices (such as the 8255A, 8254, and 8259) are commonly used in the PC environment. One can argue for a microcontroller as a basis for an introductory course. However, the experiences of many institutions suggest that a microcontroller is an appropriate device for a higher-level course; at an introductory level, the pedagogy becomes quite cumbersome. Furthermore, general-purpose 8-bit processors are now being used in small systems such as graphic calculators, coffee machines, home appliances, and serving web pages over a TCP/IP network.

PEDAGOGICAL APPROACH AND TEXT ORGANIZATION

The microprocessor is a general-purpose programmable logic device. A thorough understanding of the microprocessor demands concepts and skills from two different disciplines: hardware concepts from electronics and programming skills from computer science. Hardware is the physical structure of the microprocessor, and the programming makes it come alive; one without the other is meaningless. Therefore, this text presents an integrated approach to hardware and software in the context of the 8085 microprocessor. Part I focuses on the microprocessor architecture, the 8085 instruction set, and interfacing; Part II introduces programming; and Part III integrates hardware and software concepts from the earlier sections in interfacing and designing microprocessor-based products. Each topic is covered in

depth from basic concepts to industrial applications and is illustrated by numerous examples with complete schematics. The material is supported with assignments of practical applications.

Part I has five chapters dealing with the hardware aspects of the microcomputer as a system, presented with the spiral approach that is similar to the view from an airplane that is getting ready to land. As the plane circles around, what one observes is a view without any details. As the plane starts descending, one begins to see more details. This approach is preferable because students need to use a microcomputer as a system in their laboratory work in the early stages of a course, without having an understanding of all aspects of the system. Chapter 1 presents an overview of microprocessor-based systems with an illustration of a microprocessor-controlled temperature system (MCTS). It presents the 8-bit microprocessor as a programmable device and an embedded controller, rather than a computing device or CPU used in computers. Chapter 2 introduces the instruction set of the 8085 processor. Chapters 3, 4, and 5 examine microprocessor architecture, memory, and I/O, with each chapter having increasing depth—from registers to instruction timing and interfacing.

Part II has six chapters dealing with 8085 instructions, programming techniques, program development, and software development systems. The contents are presented in a step-by-step format. A few instructions that can perform a simple task are selected. Each instruction is described fully with illustrations of its operations and its effects on the selected flags. Then these instructions are used in writing programs, accompanied by programming techniques and troubleshooting hints. Each illustrative program begins with a problem statement, provides the analysis of the problem, illustrates the program, and explains the programming steps. The chapters conclude with reviews of all the instructions discussed. The contents of Part II are presented in such a way that, in a course with heavy emphasis on hardware, students can teach themselves assembly language programming if necessary.

Part III synthesizes the hardware concepts of Part I and the software techniques of Part II. It deals with the interfacing of I/Os, with numerous industrial and practical examples. Each illustration analyzes the hardware, includes software, and describes how hardware and software work together to accomplish given objectives. Chapters 12 through 16 include various types of data transfer between the microprocessor and its peripherals such as interrupts, interfacing of data converters, I/O with handshake signals using programmable devices, and serial I/O. Chapter 14 discusses special-purpose programmable devices used primarily with the 8085 systems (such as the 8155), while Chapter 15 discusses general-purpose programmable devices (such as the 8255A, 8254, 8259, and 8237). Chapter 17 deals primarily with the project design of a single-board microcomputer that brings together all the concepts discussed in the text. Chapter 18 discusses trends in microprocessor technology ranging from recent microcontrollers to the latest general-purpose 32- and 64-bit microprocessors.

NEW AND IMPROVED FEATURES IN THE FIFTH EDITION

The fifth edition preserves the focus as described and includes the following changes and additions, suggested by reviewers and by faculty who have used the book in their classrooms:

1. Chapter 1 is revised to include the most recent technological changes and introduces a block diagram of the microprocessor-controlled temperature system (MCTS).
2. Chapter 2 introduces a programming model of the 8085 processor and its instruction set; this was Chapter 5 in previous editions.
3. Chapters 3 and 4 include additional explanation and expansion of blocks in the MCTS.
4. Part II (Chapters 6 through 11) has few changes in the content, except an 8085 simulator (on the CD packaged with the text) can be used to demonstrate the use of instructions.
5. Chapter 11 is revised to include technological changes in PC development systems.

6. Chapter 15 includes an additional interfacing application in the context of the MCTS.
7. Chapter 18 is updated to include the latest technological changes in 32- and 64-bit microprocessors.
8. Appendix D includes data sheets of additional devices such as the LM135 temperature sensor.
9. Appendix H is added to include detailed information on the 8085 simulator.

A WORD WITH FACULTY

This text is based on my teaching experience, my course development efforts, and my association with industry engineers and programmers. It is an attempt to share my classroom experiences and my observations of industrial practices. Some of my assumptions and observations of eighteen years ago appear still valid today:

1. Software (instructions) is an integral part of the microprocessor and demands emphasis equal to that of the hardware.
2. In industry, for development of microprocessor-based projects, 70 percent of the effort is devoted to software and 30 percent to hardware.
3. Technology and engineering students tend to be hardware oriented and have considerable difficulty in programming.
4. Students have difficulty in understanding mnemonics and realizing the critical importance of flags.

In the last eighteen years, numerous faculty members have shared their classroom experiences, concerns, and student difficulties with me through letters and e-mail messages. I have made every effort to incorporate those concerns and suggestions in the fifth edition. This revised edition can be used flexibly to meet the objectives of various courses at the undergraduate level. If used for a one-semester course with 50 percent hardware and 50 percent software emphasis, the following chapters are recommended: Chapters 1 through 5 for hardware lectures and Chapters 6 through 9 and selected sections

of Chapter 10 for software laboratory sessions. For interfacing, the initial sections of Chapters 12 and 16 (introduction to interrupts and serial I/O) are recommended. If the course is heavily oriented toward hardware, Chapters 1 through 5 and Chapters 12 through 17 are recommended, and necessary programs can be selected from Chapters 6 through 9. If the course is heavily oriented toward software, Chapters 1 through 11 and selected portions of Chapters 12 and 16 can be used. For a two-semester course, it is best to use the entire text.

A WORD WITH STUDENTS

The microprocessor is an exciting, challenging, and growing field; it will pervade industry for decades to come. To meet the challenges of this growing technology, you will have to be conversant with the programmable aspect of the microprocessor. Programming is a process of problem solving and communication in the language of mnemonics. Most often, hardware-oriented students find this communication process difficult. One of the questions frequently asked by students is: How do I get started in a given programming assignment? One approach to learning programming is to examine various types of programs and imitate them. You can begin by studying the illustrative program relevant to an assignment, its flowchart, its analysis, program description, and particularly the comments. Read the instructions from Appendix F as necessary and pay attention to the flags. This text is written in such a way that simple programming aspects of the microprocessor can be self-taught. Once you master the elementary programming techniques, interfacing and design become exciting and fun.

ACKNOWLEDGMENTS

My sincere thanks to my family members: my wife, Shaila, for her unwavering support and my daugh-

ters, Nelima and Vanita, for their enthusiasm and pride in my writing activities. Several persons have made valuable contributions to this text. I would like to extend my sincere appreciation to my colleagues Charles Abate, James Delaney, and John Merrill, who offered many suggestions throughout the project, and Chris Conty, who initiated the project. Similarly, I appreciate the efforts and numerous suggestions of my reviewers: John Morgan from DeVry Institute, Peter Holsberg from Mercer Community College, David Hata from Portland Community College, and David Delkar from Kansas State University. If this text reads well, the credit goes to Gnomi Gouldin and to my colleague from the English Department, Dr. Kathy Forrest, who have devoted painstaking hours to editing the rough draft. For this fifth edition, I would like to express my sincere appreciation to the following reviewers who provided me with valuable comments and suggestions: Carl Buskey, Donald E. Haley, and Michael Pelletier, Northern Essex Community College, Haverhill, MA; Tony Messuri, Youngstown State University, OH; N.K. Swain, South Carolina State University; and Christian Wolf, The Cittone Institute, Edison, NJ.

I also thank Alex Wolf, my production editor at Prentice Hall, and copy editor Sheryl Rose for their contributions to the text. I would appreciate any communications about the text from readers. Please feel free to write or send an e-mail message.

Ramesh Gaonkar
State University of New York
O.C.C. Campus at Syracuse
Syracuse, New York 13215
E-mail: gaonkarr@sunyocc.edu

Contents

PART I	MICROPROCESSOR-BASED SYSTEMS: HARDWARE AND INTERFACING	1
Chapter 1	Microprocessors, Microcomputers, and Assembly Language	3
	1.1 Microprocessors 4 □ 1.2 Microprocessor Instruction Set and Computer Languages 13 □ 1.3 From Large Computers to Single Chip Microcontrollers 20 □ 1.4 Application: Microprocessor-Controlled Temperature System (MCTS) 24	
Chapter 2	Introduction to 8085 Assembly Language Programming	31
	2.1 The 8085 Programming Model 32 □ 2.2 Instruction Classification 34 □ 2.3 Instruction, Data Format, and Storage 37 □ 2.4 How to Write, Assemble, and Execute a Simple Program 42 □ 2.5 Overview of the 8085 Instruction Set 46 □ 2.6 Writing and Hand Assembling a Program 50	
Chapter 3	Microprocessor Architecture and Microcomputer Systems	57
	3.1 Microprocessor Architecture and Its Operations 58 □ 3.2 Memory 63 □ 3.3 Input and Output (I/O) Devices 80 □ 3.4 Example of a Microcomputer System 81 □ 3.5 Review: Logic Devices for Interfacing 83 □ 3.6 Microprocessor-Based System Application: MCTS 90	
Chapter 4	8085 Microprocessor Architecture and Memory Interfacing	95
	4.1 The 8085 MPU 96 □ 4.2 Example of an 8085-Based Microcomputer 109 □ 4.3 Memory Interfacing 116 □ 4.4 Interfacing the 8155 Memory Segment 123 □ 4.5 Illustrative Example: Designing Memory for the MCTS Project 126 □ 4.6 Testing and Troubleshooting Memory Interfacing Circuits 129 □ 4.7 How Does an 8085-Based Single-Board Microcomputer Work? 132	

Chapter 5	Interfacing I/O Devices	139	
5.1 Basic Interfacing Concepts	140	□ 5.2 Interfacing Output Displays	150
□ 5.3 Interfacing Input Devices	155	□ 5.4 Memory-Mapped I/O	157
□ 5.5 Testing and Troubleshooting I/O Interfacing Circuits	163		
□ 5.6 Some Questions and Answers	164		
PART II	PROGRAMMING THE 8085	173	
Chapter 6	Introduction to 8085 Instructions	175	
6.1 Data Transfer (Copy) Operations	176	□ 6.2 Arithmetic Operations	86
□ 6.3 Logic Operations	96	□ 6.4 Branch Operations	204
Assembly Language Programs	210	□ 6.5 Writing	
□ 6.7 Some Puzzling Questions and Their Answers	215	6.6 Debugging a Program	215
Chapter 7	Programming Techniques with Additional Instructions	227	
7.1 Programming Techniques: Looping, Counting, and Indexing	228		
□ 7.2 Additional Data Transfer and 16-Bit Arithmetic Instructions	232		
□ 7.3 Arithmetic Operations Related to Memory	241	□ 7.4 Logic	
Operations: Rotate	247	□ 7.5 Logic Operations: Compare	254
□ 7.6 Dynamic Debugging	261		
Chapter 8	Counters and Time Delays	275	
8.1 Counters and Time Delays	276	□ 8.2 Illustrative Program: Hexadecimal	
Counter	282	□ 8.3 Illustrative Program: Zero-to-Nine (Modulo Ten) Counter	
285	□ 8.4 Illustrative Program: Generating Pulse Waveforms	288	
□ 8.5 Debugging Counter and Time-Delay Programs	290		
Chapter 9	Stack and Subroutines	295	
9.1 Stack	296	□ 9.2 Subroutine	305
Return Instructions	315	□ 9.3 Restart, Conditional Call, and	
		□ 9.4 Advanced Subroutine Concepts	316
Chapter 10	Code Conversion, BCD Arithmetic, and 16-Bit Data Operations	323	
10.1 BCD-to-Binary Conversion	324	□ 10.2 Binary-to-BCD Conversion	327
□ 10.3 BCD-to-Seven-Segment-LED Code Conversion	329	□ 10.4 Binary-to-ASCII and ASCII-to-Binary Code Conversion	332
□ 10.5 BCD Addition	334	□ 10.6 BCD Subtraction	337
□ 10.7 Introduction to Advanced Instructions		□ 10.8 Multiplication	342
and Applications	338	□ 10.9 Subtraction with Carry	344
Chapter 11	Software Development Systems and Assemblers	351	
11.1 Microprocessor-Based Software Development Systems	352		
□ 11.2 Operating Systems and Programming Tools	354	□ 11.3 Assemblers	
and Cross-Assemblers	359	□ 11.4 Writing Programs Using a Cross-Assembler	363

PART III	INTERFACING PERIPHERALS (I/Os) AND APPLICATIONS	371
Chapter 12	Interrupts	375
12.1	The 8085 Interrupt	376
□	12.2 8085 Vectored Interrupts	385
□	12.3 Restart as Software Instructions	393
□	12.4 Additional I/O Concepts and Processes	395
Chapter 13	Interfacing Data Converters	403
13.1	Digital-to-Analog (D/A) Converters	404
□	13.2 Analog-to-Digital (A/D) Converters	414
Chapter 14	Programmable Interface Devices: 8155 I/O and Timer; 8279 Keyboard/Display Interface	425
14.1	Basic Concepts in Programmable Devices	426
□	14.2 The 8155: Multipurpose Programmable Device	432
□	14.3 The 8279 Programmable Keyboard/Display Interface	450
Chapter 15	General-Purpose Programmable Peripheral Devices	459
15.1	The 8255A Programmable Peripheral Interface	460
□	15.2 Illustration: Interfacing Keyboard and Seven-Segment Display	479
□	15.3 Illustration: Bidirectional Data Transfer Between Two Microcomputers	488
□	15.4 The 8254 (8253) Programmable Interval Timer	494
□	15.5 The 8259A Programmable Interrupt Controller	505
□	15.6 Direct Memory Access (DMA) and the 8237 DMA Controller	514
Chapter 16	Serial I/O and Data Communication	523
16.1	Basic Concepts in Serial I/O	524
□	16.2 Software-Controlled Asynchronous Serial I/O	534
□	16.3 The 8085—Serial I/O Lines: SOD and SID	537
□	16.4 Hardware-Controlled Serial I/O Using Programmable Chips	540
Chapter 17	Microprocessor Applications	563
17.1	Interfacing Scanned Multiplexed Displays and Liquid Crystal Displays	
464	□ 17.2 Interfacing a Matrix Keyboard	573
□	17.3 Memory Design	581
□	17.4 MPU Design	589
□	17.5 Designing a System: Single-Board Microcomputer	592
□	17.6 Software Design	597
□	17.7 Development and Troubleshooting Tools	603
Chapter 18	Extending 8-Bit Microprocessor Concepts to Higher-Level Processors and Microcontrollers	607
18.1	8-Bit Microprocessors Contemporary to the 8085	608
□	18.2 Review of Microprocessor Concepts	611
□	18.3 16-Bit Microprocessors	612
□	18.4 High-End-High-Performance Processors	626
□	18.5 Single-Chip Microcontrollers	633

Appendix A	Number Systems	637
Appendix B	Introduction to the EMAC Primer	645
Appendix C	Pin Configuration of Selected Logic and Display Devices	659
Appendix D	Specifications: Data Converters and Peripheral Devices	669
Appendix E	American Standard Code for Information Interchange: ASCII Codes	735
Appendix F	8085 Instruction Set	737
Appendix G	Solutions to Selected Questions, Problems, and Programming Assignments	785
Appendix H	Introduction to 8085 Assemblers and Simulators	801
	Index	815

I

Microprocessor-Based Systems: Hardware and Interfacing

CHAPTER 1

Microprocessors, Microcomputers, and Assembly Language

CHAPTER 2

Introduction to 8085 Assembly Language Programming

CHAPTER 3

Microprocessor Architecture and Microcomputer Systems

CHAPTER 4

8085 Microprocessor Architecture and Memory Interfacing

CHAPTER 5

Interfacing I/O Devices

Part I of this book is concerned primarily with the microprocessor architecture in the context of microprocessor-based products. The microprocessor-based systems are discussed in terms of four components—microprocessor, memory, input, and output—and their communication process. The role of the programming languages, from machine language to higher-level languages, is presented in the context of the system.

The material is presented in a format similar to the view from an airplane that is getting ready to land. As the plane circles around, a passenger observes a view without any details. As the plane starts descending, the passenger begins to see the same view but with more details. In the same way, Chapter 1 presents the microprocessor from two points of view: the microprocessor as a programmable embedded device in a product and as an element of a computer system, and how it commun-

cates with memory and I/O. The chapter also discusses the role of assembly language in microprocessor-based products and presents an overview of various types of computers—from large computers to microcomputers and their applications. The chapter concludes with a block diagram and an overview of a microprocessor-controlled temperature system (MCTS) as an application of the microprocessor-based product, and the components of this block diagram are expanded and discussed in detail in appropriate chapters.

Chapter 2 introduces the 8085 assembly language and its instruction set, and Chapter 3 provides a closer look at a microcomputer system in relation to the 8085 microprocessor. Chapter 4 examines the details of the 8085 microprocessor and memory interfacing. Chapter 5 discusses the interfacing of input/output (I/O) devices.

PREREQUISITES

The reader is expected to know the following concepts:

- Number systems (binary, octal, and hexadecimal) and their conversions.

- Boolean algebra, logic gates, flip-flops, and registers.
- Concepts in combinational and sequential logic.

1

Microprocessors, Microcomputers, and Assembly Language

The microprocessor plays a significant role in the everyday functioning of industrialized societies. The microprocessor can be viewed as a programmable logic device that can be used to control processes or to turn on/off devices. On the other hand, the microprocessor can be viewed as a data processing unit or a computing unit of a computer. The **microprocessor** is a programmable integrated device that has computing and decision-making capability similar to that of the central processing unit (CPU) of a computer. Nowadays, the microprocessor is being used in a wide range of products called microprocessor-based products or systems. The microprocessor can be embedded in a larger system, can be a stand alone unit controlling processes, or it can function as the CPU of a computer called a **microcomputer**. This chapter introduces the basic structure of a microprocessor-based product and shows how the same structure is applicable to microcomputers and other large computers.

Computers. The chapter concludes with an overview of microprocessor applications in the context of the entire spectrum of various computer applications and presents a block diagram of a temperature-control system as an application of the microprocessor-based system.

The microprocessor communicates and operates in the binary numbers 0 and 1, called **bits**. Each microprocessor has a fixed set of instructions in the form of binary patterns called a **machine language**. However, it is difficult for humans to communicate in the language of 0s and 1s. Therefore, the binary instructions are given abbreviated names, called **mnenomics**, which form the **assembly language for a given microprocessor**. This chapter explains both the machine language and the assembly language of the microprocessor, known as the 8085. The advantages of assembly language are compared with high-level languages (such as BASIC, C, C++, and Java).

OBJECTIVES

- Draw a block diagram of a microprocessor-based system and explain the functions of each component: microprocessor, memory, and I/O, and their lines of communication (the bus).
- Explain the terms *SSI*, *MSI*, and *LSI*.
- Define the terms *bit*, *byte*, *word*, *instruction*, *software*, and *hardware*.
- Explain the difference between the machine language and the assembly language of a computer.
- Explain the terms *low-level* and *high-level languages*.
- Explain the advantages of an assembly language over high-level languages.
- Define the term *ASCII* code and explain the relationship between the binary code and alphanumeric characters.
- Define the term *operating system*.
- List components and peripherals of a typical personal computer (PC).
- Draw a block diagram of a microprocessor-controlled temperature system (MCTS) and identify functions of each component.

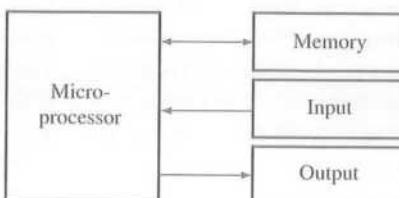
1.1

MICROPROCESSORS

A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called *memory*, accepts binary data as input and processes data according to those instructions, and provides results as output. At a very elementary level, we can draw an analogy between microprocessor operations and the functions of a human brain that process information according to instructions (understanding) stored in its memory. The brain gets input from eyes and ears and sends processed information to output “devices” such as the face with its capacity to register expression, the hands or feet. However, there is no comparison between the complexity of a human brain and its memory and the relative simplicity of a microprocessor and its memory.

A typical programmable machine can be represented with four components: microprocessor, memory, input, and output, as shown in Figure 1.1. These four components work together or interact with each other to perform a given task; thus, they comprise a system. The physical components of this system are called **hardware**. A set of instructions written for the microprocessor to perform a task is called a **program**, and a group of programs is called **software**. The machine (system) represented in Figure 1.1 can be programmed to turn traffic lights on and off, compute mathematical functions, or keep track of a guidance system. This system may be simple or sophisticated, depending on its applications, and it is recognized by various names depending upon the purpose for which it is designed. The microprocessor applications are classified primarily in two categories: **reprogrammable systems** and **embedded systems**. In reprogrammable systems, such as microcomputers, the microprocessor is used for computing and data processing. These systems include

FIGURE 1.1
A Programmable Machine



general-purpose microprocessors capable of handling large data, mass storage devices (such as disks and CD-ROMs), and peripherals such as printers; a personal computer (PC) is a typical illustration. In embedded systems, the microprocessor is a part of a final product and is not available for reprogramming to the end user. A copying machine is a typical example of an embedded system. The microprocessors used in these systems are generally categorized as: (1) **microcontrollers** that include all the components shown in Figure 1.1 on one chip, and (2) general-purpose microprocessors with discrete components shown in Figure 1.1. Embedded systems can also be viewed as products that use microprocessors to perform their operations; they are known as microprocessor-based products. Examples include a wide range of products such as washing machines, dishwashers, automobile dashboard controls, traffic light controllers, and automatic testing instruments.

BINARY DIGITS

The microprocessor operates in binary digits, 0 and 1, also known as bits. Bit is an abbreviation for the term *binary digit*. These digits are represented in terms of electrical voltages in the machine: Generally, 0 represents one voltage level, and 1 represents another. The digits 0 and 1 are also synonymous with low and high, respectively.

Each microprocessor recognizes and processes a group of bits called the *word*, and microprocessors are classified according to their word length. For example, a processor with an 8-bit word is known as an 8-bit microprocessor, and a processor with a 32-bit word is known as a 32-bit microprocessor.

A MICROPROCESSOR AS A PROGRAMMABLE DEVICE

The fact that the microprocessor is programmable means it can be instructed to perform given tasks within its capability. A piano is a programmable machine; it is capable of generating various kinds of tones based on the number of keys it has. A musician selects keys depending upon the musical score printed on a sheet. Similarly, today's microprocessor is designed to understand and execute many binary instructions. It is a multipurpose machine: It can be used to perform various sophisticated computing functions, as well as simple tasks such as turning devices on or off. A programmer can select appropriate instructions and ask the microprocessor to perform various tasks on a given set of data.

The person who designs a piano determines the frequency (tone) for a given key and the scope of the piano music. Similarly, the engineers designing a microprocessor determine a set of tasks the microprocessor should perform and design the necessary logic circuits, and provide the user with a list of the instructions the processor will understand. For example, an instruction for adding two numbers may look like a group of eight binary digits, such as 1000 0000. These instructions are simply a pattern of 0s and 1s. The user (programmer) selects instructions from the list and determines the sequence of execution for a given task. These instructions are entered or stored in storage, called *memory*, which can be read by the microprocessor.

MEMORY

Memory is like the pages of a notebook with space for a fixed number of binary numbers on each line. However, these pages are generally made of semiconductor material. Typically, each line is an 8-bit register that can store eight binary bits, and several of these registers are

arranged in a sequence called memory. These registers are always grouped together in powers of two. For example, a group of $1024(2^{10})$ 8-bit registers on a semiconductor chip is known as 1K byte of memory; 1K is the closest approximation in thousands.* The user writes the necessary instructions and data in memory through an input device (described below), and asks the microprocessor to perform the given task and find an answer. The answer is generally displayed at an output device (described below) or stored in memory.

INPUT/OUTPUT

The user can enter instructions and data into memory through devices such as a keyboard or simple switches. These devices are called **input devices**, similar to eyes and ears in a human body. The microprocessor reads the instructions from the memory and processes the data according to those instructions. The result can be displayed by a device such as seven-segment LEDs (Light Emitting Diodes) or printed by a printer. These devices are called **output devices**.

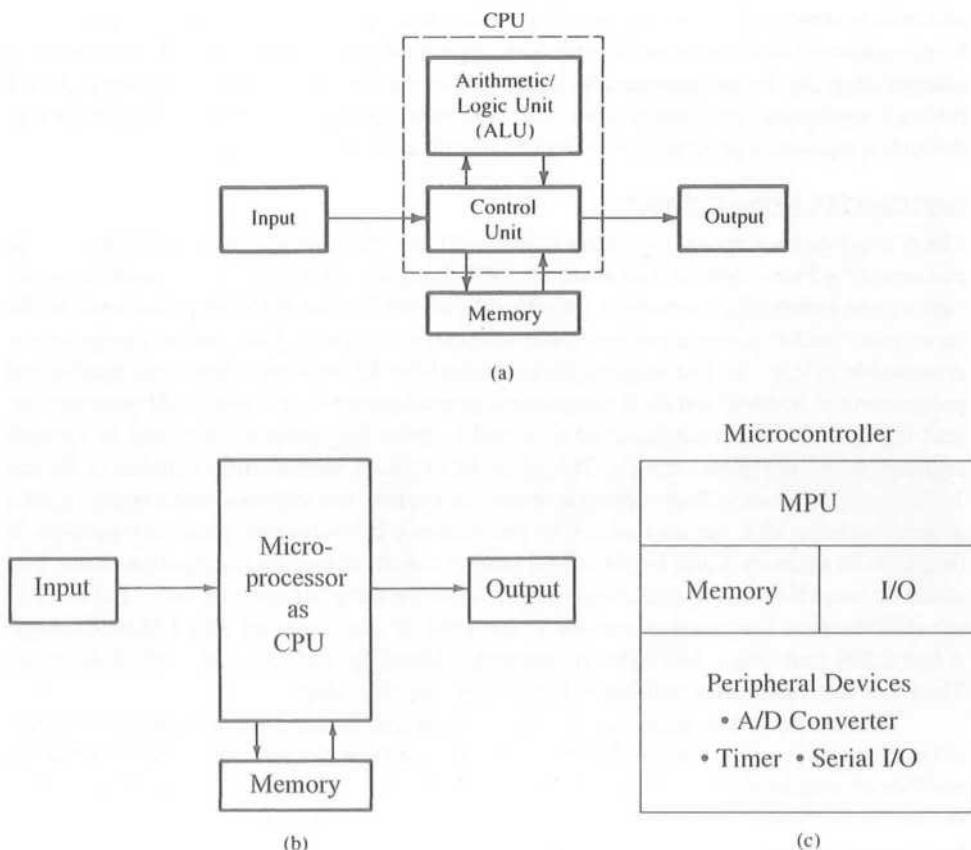
MICROPROCESSOR AS A CPU (MPU)

We can also view the microprocessor as a primary component of a computer. Traditionally, the computer is represented in block diagram as shown in Figure 1.2(a). The block diagram shows that the computer has four components: memory, input, output, and the central processing unit (CPU), which consists of the **arithmetic/logic unit (ALU)** and the **control unit**. The CPU contains various registers to store data, the ALU to perform arithmetic and logical operations, instruction decoders, counters, and control lines. The CPU reads instructions from the memory and performs the tasks specified. It communicates with **input/output devices** either to accept or to send data. These devices are also known as **peripherals**. The CPU is the **primary** and central player in communicating with devices such as memory, input, and output. However, the timing of the communication process is controlled by the group of circuits called the **control unit**.

In the late 1960s, the CPU was designed with discrete components on various boards. With the advent of integrated circuit technology, it became possible to build the CPU on a single chip; this came to be known as a microprocessor, and the traditional block diagram shown in Figure 1.2(a) can be replaced by the block diagram shown in Figure 1.2(b). A computer with a microprocessor as its CPU is known as a **microcomputer**. The terms **microprocessor** and **microprocessor unit (MPU)** are often used synonymously. MPU implies a complete processing unit with the necessary control signals. Because of the limited number of available pins on a microprocessor package, some of the signals (such as control and multiplexed signals) need to be generated by using discrete devices to make the microprocessor a complete functional unit or MPU.

As semiconductor fabrication technology became more advanced, manufacturers were able to place not only MPU but also memory and I/O interfacing circuits on a single chip; this is known as a **microcontroller** or **microcontroller unit (MCU)**. A microcontroller is essentially an entire computer on a single chip. Figure 1.2(c) shows that the microcontroller chip also includes additional devices such as an A/D converter, serial I/O, and timers (these devices are discussed in later chapters).

*In computer terminology, 1K is equal to 1024. In scientific terminology, 1k is equal to 1000.

**FIGURE 1.2**

(a) Traditional Block Diagram of a Computer. (b) Block Diagram of a Computer with the Microprocessor as CPU; and (c) Block Diagram of a Microcontroller

1.1.1 Advances in Semiconductor Technology

Since 1950, semiconductor technology has undergone unprecedented changes. After the invention of the transistor, integrated circuits (ICs) appeared on the scene at the end of the 1950s; an entire circuit consisting of several transistors, diodes, and resistors could be designed on a single chip. In the early 1960s, logic gates known as the 7400 series were commonly available as ICs, and the technology of integrating the circuits of a logic gate on a single chip became known as small-scale integration (SSI). As semiconductor technology advanced, more than 100 gates were fabricated on one chip; this was called medium-scale integration (MSI). A typical example of MSI is a decade counter (7490). Within a few years, it was possible to fabricate more than 1000 gates on a single chip; this came to be known as large-scale integration (LSI). Now we are in the era of very-large-scale integration (VLSI) and super-large-scale integration (SLSI). The lines of demarcation between these different scales of integration are rather ill defined and arbitrary. As

technology improved, more and more logic circuits were built on one chip, and they could be programmed to do different functions through hard-wired connections. For example, a counter chip can be programmed to count in Hex or decimal by providing logic 0 or 1 through appropriate pin connections. The next step was the idea of providing 0s and 1s through a register, a programmable device described in the next section.

HISTORICAL PERSPECTIVE

The microprocessor revolution began with a bold and innovative approach in logic design pioneered by Intel engineer Ted Hoff. In 1969, Intel was primarily in the business of designing semiconductor memory; it introduced a 64-bit bipolar RAM chip that year. In the same year, Intel received a contract from a Japanese company, Busicom, to design a programmable calculator. The original design called for 12 different chips with hard-wired programming. Instead, Ted Hoff suggested a general-purpose chip that could perform various logic functions, which could be activated by providing patterns of 0s and 1s through registers with appropriate timing. The group of registers used to store patterns of 0s and 1s was called memory. Thus a programmable calculator was designed successfully with a general-purpose logic device that can be programmed by storing the necessary patterns of 0s and 1s in memory. Later Intel realized that this small device had computing power that could be used for many applications. Intel coined the term “microprocessor” and in 1971 released the first 4-bit microprocessor as the 4004. It was designed with LSI technology; it had 2,300 transistors, 640 bytes of memory-addressing capacity, and a 108 kHz clock. Thus, the microprocessor revolution began with this tiny chip.

This invention has been placed on a par with that of the printing press or the internal combustion engine. Gordon Moore, cofounder of Intel Corporation, predicted that the number of transistors per integrated circuit would double every 18 months; this came to be known as “Moore’s Law.” Just twenty-five years since the invention of the 4004, we have processors that are designed with 15 million transistors, that can address one terabyte (1×10^{12}) of memory, and that can operate at 400 MHz to 1.5-GHz frequency (see Table 1.1).

The Intel 4004 was quickly replaced by the 8-bit microprocessor (the Intel 8008), which was in turn superseded by the Intel 8080. In the mid-1970s, the Intel 8080 was widely used in control applications, and small computers also were designed using the 8080 as the CPU; these computers became known as microcomputers. Within a few years after the emergence of the 8080, the Motorola 6800, the Zilog Z80, and the Intel 8085 microprocessors were developed as improvements over the 8080. The 6800 was designed with a different architecture and the instruction set from the 8080. On the other hand, the 8085 and the Z80 were designed as upward software compatible with the 8080; that is, they included all the instructions of the 8080 plus additional instructions. As the microprocessors began to acquire more and more computing functions, they were viewed more as CPUs rather than as programmable logic devices. Most microcomputers are now built with 32- and 64-bit microprocessors. Each microprocessor has begun to carve a niche for its own applications. The 8-bit microprocessors are being used as programmable logic devices in control applications, and more powerful microprocessors are being used for mathematical computing (number crunching), data processing, and computer graphics applications.

TABLE 1.1
Intel Microprocessors: Historical Perspective

Processor	Year of Introduction	Number of Transistors	Initial Clock Speed	Address Bus	Data Bus	Addressable Memory
4004	1971	2,300	108 kHz	10-bit	4-bit	640 bytes
8008	1972	3,500	200 kHz	14-bit	8-bit	16 K
8080	1974	6,000	2 MHz	16-bit	8-bit	64 K
8085	1976	6,500	5 MHz	16-bit	8-bit	64 K
8086	1978	29,000	5 MHz	20-bit	16-bit	1 M
8088	1979	29,000	5 MHz	20-bit	8-bit*	1 M
80286	1982	134,000	8 MHz	24-bit	16-bit	16 M
80386	1985	275,000	16 MHz	32-bit	32-bit	4 G
80486	1989	1.2 M	25 MHz	32-bit	32-bit	4 G
Pentium	1993	3.1 M	60 MHz	32-bit	32/64-bit	4 G
Pentium Pro	1995	5.5 M	150 MHz	36-bit	32/64-bit	64 G
Pentium II	1997	8.8 M	233 MHz	36-bit	64-bit	64 G
Pentium III	1999	9.5 M	650 MHz	36-bit	64-bit	64 G
Pentium 4	2000	42 M	1.4 GHz	36-bit	64-bit	64 G

*External 8-bit and internal 16-bit data bus

Our focus here is on using 8-bit microprocessors as programmable devices in microprocessor-based systems. The overwhelming majority of microprocessor applications use 8-bit processors or microcontrollers. The range of applications is very wide and diversified—from the auto industry to home appliances. Recent statistics suggest that the worldwide sales of 8-bit chips range from 90 percent to 95 percent of the total number of microprocessor chips sold. However, the most compelling reason to use an 8-bit processor is educational. To understand the basic concepts underlying the microprocessor device, it is easier to learn from a simple 8-bit processor than from a 64-bit processor. And these fundamental concepts are easily transferable from 8-bit processors to larger processors.

1.1.2 Organization of a Microprocessor-Based System

Figure 1.3 shows a simplified but formal structure of a microprocessor-based system or a product. Since a microcomputer is one among many microprocessor-based systems, it will have the same structure as shown in Figure 1.3. It includes three components: *microprocessor*, *I/O (input/output)*, and *memory* (read/write memory and read-only memory). These components are organized around a common communication path called a *bus*. The entire group of components is also referred to as a system or a microcomputer system, and the components themselves are referred to as sub-systems. At the outset, it is necessary to differentiate between the terms *microprocessor* and *microcomputer* because of the common misuse of these terms in popular literature. The *microprocessor* is one component of the *microcomputer*. On the other hand, the mi-

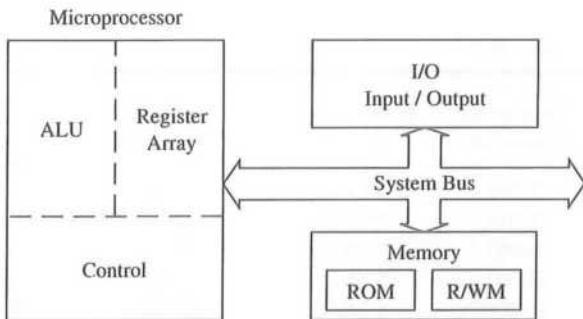


FIGURE 1.3
Microprocessor-Based System with Bus Architecture

microcomputer is a complete computer similar to any other computer, except that CPU functions of the microcomputer are performed by the microprocessor. Similarly, the term **peripheral** is used for input/output devices. The various components of a microprocessor-based product or a microcomputer are shown in Figure 1.3 and their functions are described in this section.

MICROPROCESSOR

The microprocessor is a clock-driven semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale integration (LSI) or very-large-scale integration (VLSI) technique. The microprocessor is capable of performing various computing functions and making decisions to change the sequence of program execution. In large computers, a CPU implemented on one or more circuit boards performs these computing functions. The microprocessor is in many ways similar to the CPU, but includes all the logic circuitry, including the control unit, on one chip. The microprocessor can be divided into three segments for the sake of clarity, as shown in Figure 1.3: arithmetic/logic unit (ALU), register array, and control unit.

Arithmetic/Logic Unit This is the area of the microprocessor where various computing functions are performed on data. The ALU unit performs such arithmetic operations as addition and subtraction, and such logic operations as AND, OR, and exclusive OR.

Register Array This area of the microprocessor consists of various registers identified by letters such as B, C, D, E, H, and L. These registers are primarily used to store data temporarily during the execution of a program and are accessible to the user through instructions.

Control Unit The control unit provides the necessary timing and control signals to all the operations in the microcomputer. It controls the flow of data between the microprocessor and memory and peripherals.

Now the question is: What is the relationship among the programmer's instruction (binary pattern of 0s and 1s), the ALU, and the control unit? This can be explained with the example of a full adder circuit. A full adder circuit can be designed with registers, logic gates, and a clock. The clock initiates the adding operation. Similarly, the bit pattern of an instruction initiates a sequence of clock signals, activates the appropriate logic circuits in the ALU, and performs the task. This is called microprogramming, which is done in the design stage of the microprocessor. In many ways, this is similar to the process of how our brain operates. In early childhood, we learn a word, "sit," and physical motions needed for the action are embedded in our brain. When we hear the word "sit," our brain activates a series of actions for our muscles and bones and we sit down. In this analogy, the word "sit" is like an instruction in a microprocessor, and actions initiated by our brain are like microprograms.

The bit patterns required to initiate these microprogram operations are given to the programmer in the form of the instruction set of the microprocessor. The programmer selects appropriate bit patterns from the set for a given task and enters them sequentially in memory through an input device. When the CPU reads these bit patterns one at a time, it initiates appropriate microprograms through the control unit, and performs the task specified in the instructions.

At present, various microprocessors are available from different manufacturers. Examples of widely used 8-bit microprocessors include the Intel 8085, Zilog Z80, and Motorola 68008. Earlier microcomputers were designed around the 8-bit microprocessors; now these processors are generally used in embedded systems. The recent versions of IBM personal computers are designed around the Intel 32- or 64-bit microprocessors. Single-board microcomputers such as the SDK-85 (Intel), The Primer (EMAC Inc.), and the Micro-Professor (Multitech) are commonly used in college laboratories; the SDK-85 and The Primer (described in Appendix B) are based on the 8085 microprocessor, and the Micro-Professor is based on the Z80 microprocessor.

MEMORY

Memory stores such binary information as instructions and data, and provides that information to the microprocessor whenever necessary. To execute programs, the microprocessor reads instructions and data from memory and performs the computing operations in its ALU section. Results are either transferred to the output section for display or stored in memory for later use. The memory block shown in Figure 1.3 has two sections: **Read-Only memory (ROM)** and **Read/Write memory (R/WM)**, popularly known as **Random-Access memory (RAM)**.

The ROM is used to store programs that do not need alterations. The monitor program of a single-board microcomputer is generally stored in the ROM. This program interprets the information entered through a keyboard and provides equivalent binary digits to the microprocessor. Programs stored in the ROM can only be read; they cannot be altered.

The Read/Write memory (R/WM) is also known as *user memory*. It is used to store user programs and data. In single-board microcomputers, the monitor program monitors the Hex keys and stores those instructions and data in the R/W memory. The information stored in this memory can be easily read and altered.

I/O (INPUT/OUTPUT)

The third component of a microprocessor-based system is I/O (input/output); it communicates with the outside world. I/O includes two types of devices: input and output; these I/O devices are also known as *peripherals*.

The input devices such as a keyboard, switches, and an analog-to-digital (A/D) converter transfer binary information (data and instructions) from the outside world to the microprocessor. Typically, a microcomputer used in college laboratories includes either a hexadecimal keyboard or an ASCII keyboard as an input device. The hexadecimal (Hex) keyboard has 16 data keys (0 to 9 and A to F) and some additional function keys to perform such operations as storing data and executing programs. The ASCII (the term is explained in Section 1.2) keyboard is similar to a typewriter keyboard, and it is used to enter programs in an English-like language. Although the ASCII keyboard is found in most microcomputers (PCs), single-board microcomputers generally have Hex keyboards, and microprocessor-based products such as a microwave oven have decimal keyboards.

The output devices transfer data from the microprocessor to the outside world. They include devices such as light emitting diodes (LEDs), a cathode-ray tube (CRT) or video screen, a printer, X-Y plotter, a magnetic tape, and digital-to-analog (D/A) converter. Typically, single-board microcomputers and microprocessor-based products (such as a dishwasher or a microwave oven) include LEDs, seven-segment LEDs, and alphanumeric LED displays as output devices. Microcomputers (PCs) are generally equipped with output devices such as a video screen (also called a monitor) and a printer.

SYSTEM BUS

The system bus is a communication path between the microprocessor and peripherals; it is nothing but a group of wires to carry bits. In fact, there are several buses in the system that will be discussed in the next chapter. All peripherals (and memory) share the same bus; however, the microprocessor communicates with only one peripheral at a time. The timing is provided by the control unit of the microprocessor.

1.1.3 How Does the Microprocessor Work?

Assume that a program and data are already entered in the R/W memory. (How to write and execute a program will be explained later.) The program includes binary instructions to add given data and to display the answer at the seven-segment LEDs. When the microprocessor is given a command to execute the program, it reads and executes one instruction at a time and finally sends the result to the seven-segment LEDs for display.

This process of program execution can best be described by comparing it to the process of assembling a radio kit. The instructions for assembling the radio are printed in a sequence on a sheet of paper. One reads the first instruction, then picks up the necessary components of the radio and performs the task. The sequence of the process is *read, interpret, and perform*. The microprocessor works the same way. The instructions are stored sequentially in the memory. The microprocessor fetches the first instruction

from its memory sheet, decodes it, and executes that instruction. The sequence of *fetch*, *decode*, and *execute* is continued until the microprocessor comes across an instruction to stop. During the entire process, the microprocessor uses the system bus to fetch the binary instructions and data from the memory. It uses registers from the register section to store data temporarily, and it performs the computing function in the ALU section. Finally, it sends out the result in binary, using the same bus lines, to the seven-segment LEDs.

1.1.4 Summary of Important Concepts

The functions of various components of a microprocessor-based system can be summarized as follows:

1. The microprocessor
 - reads instructions from memory.
 - communicates with all peripherals (memory and I/Os) using the system bus.
 - controls the timing of information flow.
 - performs the computing tasks specified in a program.
2. The memory
 - stores binary information, called instructions and data.
 - provides the instructions and data to the microprocessor on request.
 - stores results and data for the microprocessor.
3. The input device
 - enters data and instructions under the control of a program such as a monitor program.
4. The output device
 - accepts data from the microprocessor as specified in a program.
5. The bus
 - carries bits between the microprocessor and memory and I/Os.

MICROPROCESSOR INSTRUCTION SET AND COMPUTER LANGUAGES

1.2

Microprocessors recognize and operate in binary numbers. However, each microprocessor has its own binary words, meanings, and language. The words are formed by combining a number of bits for a given machine. The word (or word length) is defined as the number of bits the microprocessor recognizes and processes at a time. The word length ranges from four bits for small, microprocessor-based systems to 64 bits for high-speed large computers. Another term commonly used to express word length is byte. A byte is defined as a group of eight bits. For example, a 16-bit microprocessor has a word length equal to two bytes. The term nibble, which stands for a group of four bits, is found also in popular computer magazines and books. A byte has two nibbles.

Each machine has its own set of instructions based on the design of its CPU or of its microprocessor. To communicate with the computer, one must give instructions in binary language (**machine language**). Because it is difficult for most people to write programs in sets of 0s and 1s, computer manufacturers have devised English-like words to represent the binary instructions of a machine. Programmers can write programs, called **assembly language** programs, using these words. Because an assembly language is specific to a given machine, programs written in assembly language are not transferable from one machine to another. To circumvent this limitation, such general-purpose languages as BASIC and FORTRAN have been devised; a program written in these languages can be machine-independent. These languages are called **high-level languages**. This section deals with various aspects of these three types of languages: machine, assembly, and high-level. The machine and assembly languages are discussed in the context of the 8085 microprocessor.

1.2.1 Machine Language

The number of bits in a word for a given machine is fixed, and words are formed through various combinations of these bits. For example, a machine with a word length of eight bits can have 256 (2^8) combinations of eight bits—thus a language of 256 words. However, not all of these words need to be used in the machine. The microprocessor design engineer selects combinations of bit patterns and gives a specific meaning to each combination by using electronic logic circuits; this is called an **instruction**. Instructions are made up of one word or several words. The set of instructions designed into the machine makes up its machine language—a binary language, composed of 0s and 1s—that is specific to each computer. In this book, we are concerned with the language of a widely used 8-bit microprocessor, the 8085, manufactured by Intel Corporation. The primary focus here is on the microprocessor because the microprocessor determines the machine language and the operations of a microprocessor-based system.

1.2.2 8085 Machine Language

The 8085 is a microprocessor with 8-bit word length: its **instruction set** (or language) is designed by using various combinations of these eight bits. The 8085 is an improved version of the earlier processor 8080A.

An **instruction** is a binary pattern entered through an input device in memory to command the microprocessor to perform that specific function.

For example:

- 0011 1100 is an instruction that increments the number in the register called the **accumulator** by one.
- 1000 0000 is an instruction that adds the number in the register called B to the number in the accumulator, and keeps the sum in the accumulator.

The 8085 microprocessor has 246 such bit patterns, amounting to 74 different instructions for performing various operations. These 74 different instructions are called its **instruction set**. This binary language with a predetermined instruction set is called the **8085 machine language**.

Because it is tedious and error-inducive for people to recognize and write instructions in binary language, these instructions are, for convenience, written in hexadecimal code and entered in a single-board microcomputer by using Hex keys. For example, the binary instruction 0011 1100 (mentioned previously) is equivalent to 3C in hexadecimal. This instruction can be entered in a single-board microcomputer system with a Hex keyboard by pressing two keys: 3 and C. The monitor program of the system translates these keys into their equivalent binary pattern.

1.2.3 8085 Assembly Language

Even though the instructions can be written in hexadecimal code, it is still difficult to understand a program written in hexadecimal numbers. Therefore, each manufacturer of a microprocessor has devised a symbolic code for each instruction, called a **mnemonic**. (The word *mnemonic* is based on the Greek word meaning *mindful*; that is, a memory aid.) The mnemonic for a particular instruction consists of letters that suggest the operation to be performed by that instruction.

For example, the binary code 0011 1100 ($3C_{16}$ or 3CH* in hexadecimal) of the 8085 microprocessor is represented by the mnemonic INR A:

INR A INR stands for increment, and A represents the accumulator. This symbol suggests the operation of incrementing the accumulator contents by one.

Similarly, the binary code 1000 0000 (80_{16} or 80H) is represented as

ADD B ADD stands for addition, and B represents the contents in register B. This symbol suggests the addition of the contents in register B and the contents in the accumulator.

Although these symbols do not specify the complete operations, they suggest its significant part. The complete description of each instruction must be supplied by the manufacturer. The complete set of 8085 mnemonics is called the 8085 assembly language, and a program written in these mnemonics is called an assembly language program. (Again, the assembly language, or mnemonics, is specific to each microprocessor. For example, the Motorola 6800 microprocessor has an entirely different set of binary codes and mnemonics than the 8085. Therefore, the assembly language of the 6800 is far different from that of the 8085.) An assembly language program written for one microprocessor is not transferable to a computer with another microprocessor unless the two microprocessors are compatible in their machine codes.

Machine language and assembly language are microprocessor-specific and are both considered low-level languages. The machine language is in binary, and the assembly language is in English-like words; however, the microprocessor understands only the binary. How, then, are the assembly language mnemonics written and translated into machine language or binary code? The mnemonics can be written by hand on paper (or in a

*Hexadecimal numbers are shown either with the subscript 16, or as a number followed by the letter H.

notebook) and translated manually in hexadecimal code, called **hand assembly**, as explained in Section 1.25. Similarly, the mnemonics can be written electronically on a computer using a program called an Editor in the ASCII code (explained in the next section) and translated into binary code by using the program called an **assembler**.

1.2.4 ASCII Code

A computer is a binary machine; to communicate with the computer in alphabetic letters and decimal numbers, translation codes are necessary. The commonly used code is known as **ASCII**—American Standard Code for Information Interchange. It is a 7-bit code with 128 (2^7) combinations, and each combination from 00H to 7FH is assigned to either a letter, a decimal number, a symbol, or a machine command (see Appendix E). For example, hexadecimal 30H to 39H represent 0 to 9 decimal digits, 41H to 5AH represent capital letters A through Z, 20H to 2FH represent various symbols, and initial codes 00H to 1FH represent such machine commands as carriage return and line feed. In microcomputer systems, keyboards (called ASCII keyboards), video screens, and printers are typical examples of devices that use ASCII codes. When the key "9" is pressed on an ASCII keyboard, the computer receives 39H in binary, called an ASCII character, and the system program translates ASCII characters into appropriate binary numbers.

However, recent computers use many more characters than the original 128 combinations; this is called Extended ASCII. It is an 8-bit code that provides 256 (2^8) combinations; the additional 128 combinations are assigned to various graphics characters.

1.2.5 Writing and Executing an Assembly Language Program

As we explained earlier, a program is a set of logically related instructions written in a specific sequence to accomplish a task. To manually write and execute an assembly language program on a single-board computer, with a Hex keyboard for input and LEDs for output, the following steps are necessary:

1. Write the instructions in mnemonics obtained from the instruction set supplied by the manufacturer.
2. Find the hexadecimal machine code for each instruction by searching through the set of instructions.
3. Enter (load) the program in the user memory in a sequential order by using the Hex keyboard as the input device.
4. Execute the program by pressing the Execute key. The answer will be displayed by the LEDs.

This procedure is called either **manual** or **hand assembly**.

When the user program is entered by the keys, each entry is interpreted and converted into its binary equivalent by the monitor program, and the machine code is stored as eight bits in each memory location in a sequence. When the Execute command is given, the microprocessor fetches each instruction, decodes it, and executes it in a sequence until the end of the program.

The manual assembly procedure is commonly used in single-board microcomputers and is suited for small programs; however, looking up the machine codes and entering the program is tedious and subject to errors. The other process involves the use of a computer with an ASCII keyboard and an assembler.

The **assembler** is a program that translates the mnemonics entered by the ASCII keyboard into the corresponding binary machine codes of the microprocessor. Each microprocessor has its own assembler because the mnemonics and machine codes are specific to the microprocessor being used, and each assembler has rules that must be followed by the programmer. Personal Computers (PCs—see Section 1.3) are commonly available on college campuses. These computers are based on 16- or 32-bit microprocessors with different mnemonics than the 8085 microprocessor. However, the programs known as **cross-assemblers** can be used to translate the 8085 mnemonics into appropriate machine codes. (Assemblers and cross-assemblers are discussed in Chapter 11.)

1.2.6 High-Level Languages

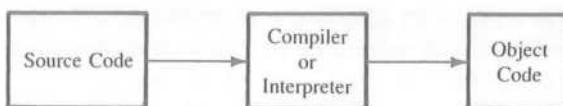
Programming languages that are intended to be machine-independent are called **high-level languages**. These include such languages as BASIC, PASCAL, C, C++, and Java, all of which have certain sets of rules and draw on symbols and conventions from English. Instructions written in these languages are known as **statements** rather than mnemonics. A program written in BASIC for a microcomputer with the 8085 microprocessor can generally be run on another microcomputer with a different microprocessor.

Now the question is: How are words in English converted into the binary languages of different microprocessors? The answer is: Through another program called either a **compiler** or an **interpreter**. These programs accept English-like statements as their input, called the *source code*. The compiler or interpreter then translates the source code into the machine language compatible with the microprocessor being used in the system. This translation in the machine language is called the *object code* (Figure 1.4). Each microprocessor needs its own compiler or an interpreter for each high-level language. The primary difference between a compiler and an interpreter lies in the process of generating machine code. The compiler reads the entire program first and translates it into the object code that is executed by the microprocessor. On the other hand, the interpreter reads one instruction at a time, produces its object code (a sequence of machine actions), and executes the instruction before reading the next instruction. M-Basic is a common example of an interpreter for BASIC language. Compilers are generally used in such languages as FORTRAN, PASCAL, C, and C++.

Compilers and interpreters require large memory space because an instruction in English requires several machine codes to translate it into binary. On the other hand, there is one-to-one correspondence between the assembly language mnemonics and the ma-

FIGURE 1.4

Block Diagram: Translation of High-Level Language Program into Machine Code



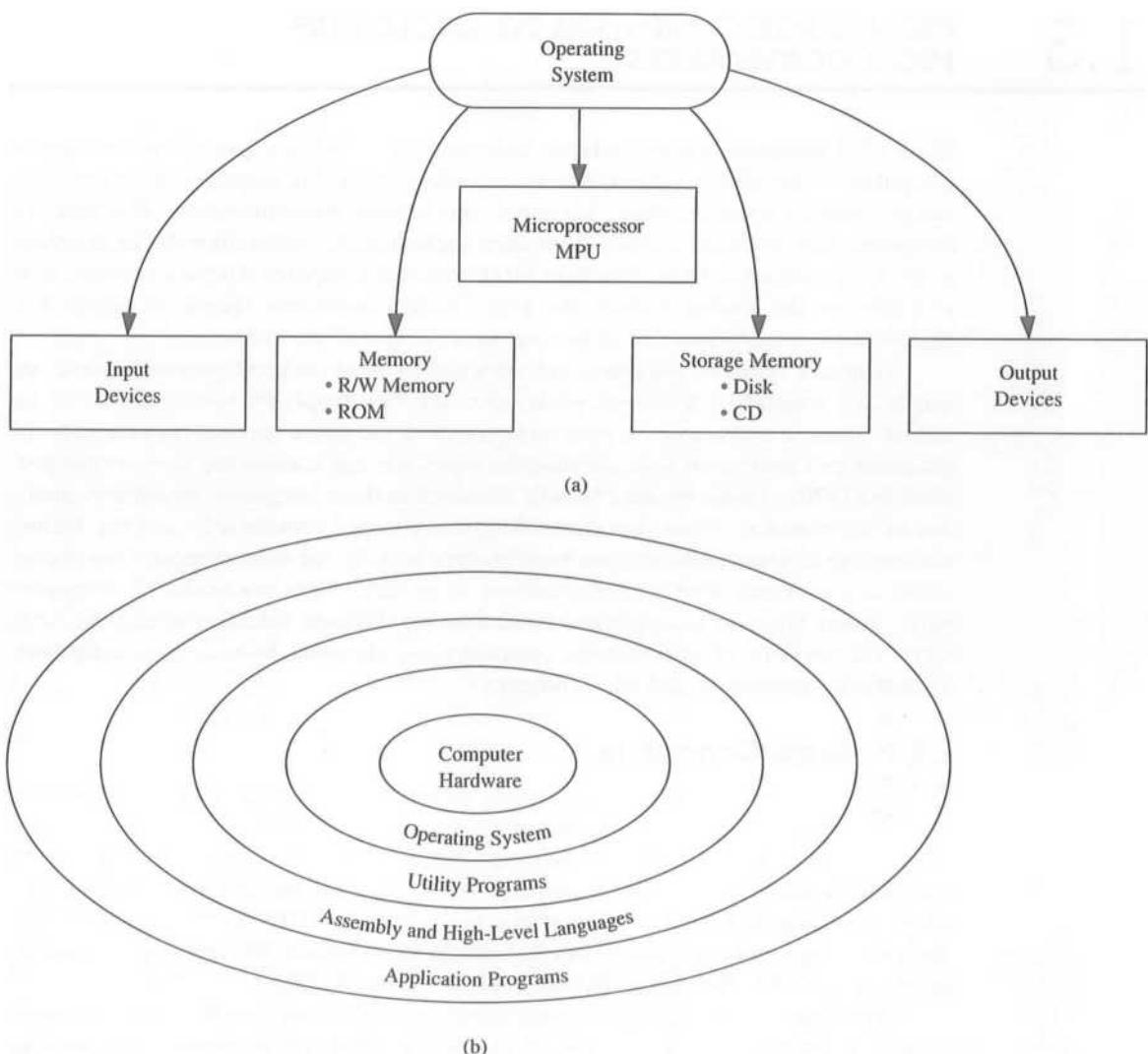
chine code. Thus, assembly language programs are compact and require less memory space. They are more efficient than the high-level language programs. The primary advantage of high-level languages is in troubleshooting (debugging) programs. It is much easier to find errors in a program written in a high-level language than to find them in a program written in an assembly language.

In certain applications such as traffic control and appliance control, where programs are small and compact, assembly language is suitable. Similarly, in such real-time applications as converting a high-frequency waveform into digital data, program efficiency is critical. In real-time applications, events and time should closely match each other without significant delay; therefore, assembly language is highly desirable in these applications. On the other hand, for applications in which programs are large and memory is not a limitation, high-level languages may be desirable. Typical examples of applications programs are word processors, video games, tax-return preparation, billing, accounting, and money management. These programs are generally written by professionals such as programmers in high-level languages. The advantage of time saved in debugging a large program may outweigh the disadvantages of memory requirements and inefficiency. Now we need to examine the relationship and the interaction between the hardware (microprocessor, memory, and I/O) and software (languages and application programs).

1.2.7 Operating Systems

The interaction between the hardware and the software is managed by a set of programs called an **operating system** of a computer; it oversees all the operations of the computer. The computer transfers information constantly between memory and various peripherals such as printer, keyboard, and video monitor. It also stores programs on disk. The operating system is responsible primarily for storing information on the disk and for the communication between the computer and its peripherals. The functional relationship between the operating system and the hardware of the computer is shown in Figure 1.5(a).

Figure 1.5(b) shows the relationship and the hierarchy among the hardware, the operating system, high-level languages, and application programs. The operating system is closest to the hardware and application programs are farthest from the hardware. When the computer is turned on, the operating system is in charge of the system and stays in the background and provides channels of communications to application programs. Each computer has its own operating system. In the 1970s, CP/M (Control Monitor Program) was a widely used operating system; it was designed for 8-bit processors such as the Z80 and 8085/8080A. In the 1980s, when 8-bit processors were replaced by 16-bit processors in personal computers (PCs), MS-DOS (Microsoft Disk Operating System) replaced CP/M. MS-DOS (also known as DOS or PC-DOS) was designed to handle 16-bit processor systems, and it became almost an industry standard for the personal computer. MS-DOS is a text-based operating system; the commands are written using a keyboard. In the 1990s, it was replaced by graphical user interface (GUI) operating systems such as Windows 3.1 and 95, which enabled the user to write commands by clicking on icons rather than using a keyboard. In recent 32- and 64-bit computers, operating systems such as UNIX, Linux, OS/2, Windows 95/98/2000, ME (Millennium), and NT are commonly used.

**FIGURE 1.5**

- (a) Operating System and Its Functional Relationship with Various Hardware Components
(b) Hierarchical Relationship between Computer Hardware and Software

1.3

FROM LARGE COMPUTERS TO SINGLE-CHIP MICROCONTROLLERS

Since 1950, advances in semiconductor technology have had an unprecedented impact on computers. In the 1960s, computers were accessible only to big corporations, universities, and government agencies. Now, “computer” has become a common word. The range of computers now available extends from such sophisticated, multimillion-dollar machines as the Cray computers to the less-than-\$1000 personal computer. All the computers now available on the market include the same basic components shown in Figure 1.3. Nevertheless, it is obvious that these computers are not all the same.

Different types of computers are designed to serve different purposes. Some are suitable for scientific calculations, while others are used simply for turning appliances on and off. Thus, it is necessary to have an overview of the entire spectrum of computer applications as a context for understanding the topics and applications discussed in this text. Until the 1970s, computers were broadly classified in three categories: mainframe, mini-, and microcomputers. Since then, technology has changed considerably, and the distinctions between these categories have been blurred. Initially, the microcomputer was recognized as a computer with a microprocessor as its CPU. Now practically all computers have various types of microprocessors performing different functions within the large CPU. For the sake of convenience, computers are classified here as large computers, medium-size computers, and microcomputers.

1.3.1 Large Computers

These are large, general-purpose, multi-user, multitasking computers designed to perform such data processing tasks as complex scientific and engineering calculations, and handling of records for large corporations or government agencies. These computers can be classified broadly into mainframes and supercomputers, and mainframes are further classified according to their sizes. The prices range from \$100,000 to millions of dollars. Typical examples of these computers include the IBM System/390 series, Cray computers (Cray-2, Y-MP), the Fujitsu GS8800, and the Hitachi MP5800.

Mainframes are high-speed computers, and their word length generally ranges from 32 to 64 bits. They are capable of addressing megabytes of memory and handling all types of peripherals and a large number of users. Supercomputers such as the Cray-2 and Y-MP are 64-bit high-performance and high-speed computers. They are the fastest computers, capable of executing billions of instructions per second, and are used primarily in research dealing with problems in areas such as global climate and high-energy physics.

1.3.2 Medium-Size Computers

In the 1960s, these computers were designed to meet the instructional needs of small colleges, the manufacturing problems of small factories, and the data processing tasks of medium-size businesses, such as payroll and accounting. These were called minicomput-

ers. These machines were slower and smaller in memory capacity than mainframes. The price range used to be anywhere from \$25,000 to \$100,000. Typical examples include such computers as the Digital Equipment PDP 11/45 and the Data General Nova.

However, current low-end mainframes and high-end microcomputers (described in the next section) overlap considerably in price, performance, and applications with traditional minicomputers. Therefore, the term **minicomputer** is mostly referred to in the historical context.

1.3.3 Microcomputers

The 4-bit and 8-bit microprocessors became available in the mid-1970s, and initial applications were primarily in the areas of machine control and instrumentation. As the price of the microprocessors and memory began to decline, the applications mushroomed in almost all areas, such as video games, word processing, and small-business applications. Early microcomputers were designed around 8-bit microprocessors. Since then, 16-, 32- and 64-bit microprocessors, such as the Intel 8086/88, 80386/486, Pentium, Pro-Pentium, Pentium 4, Motorola 68000, and the Power PC series have been introduced, and recent microcomputers are being designed around these microprocessors. Present-day microcomputers can be classified in four groups: personal (or business) computers (PC), workstations, single-board, and single-chip microcomputers (microcontrollers).

PERSONAL COMPUTERS (PC)

These microcomputers are single-user systems and are used for a variety of purposes, such as payroll, business accounts, word processing, legal and medical record keeping, personal finance, accessing Internet resources (e-mail, Web search, newsgroup), and instruction. They are also known as personal computers (PC) or desktop computers. Typically, the price ranges from \$500 to \$5000 for a single-user system. Examples include such microcomputers as the IBM Personal Computer (Aptiva series), the Hewlett-Packard Pavilion series, and the Apple Macintosh series. Figure 1.6 shows an example.

At the low end of the microcomputer spectrum, a typical configuration includes a 32-bit (or 64-bit) microprocessor, 32 to 256 MB (megabytes) of system memory, a video screen (monitor), a 3½" high-density floppy disk, a hard disk with storage capacity of more than 10 gigabytes, a CD-ROM, and a Zip disk. The **floppy disk** is a magnetic medium similar to a cassette tape except that it is round in shape, like a record. Information recorded on these disks can be accessed randomly using disk drives. Conversely, information stored on a cassette tape is accessed serially. In order to read information at the end of the tape, the user must run the entire tape through the machine. The **hard disk** is similar to the floppy disk except that the magnetic material is coated on a rigid aluminum base that is enclosed in a sealed container and permanently installed in a microcomputer. The hard disk and the floppy disks are used to store programs semipermanently, i.e., the binary information does not disappear when the power is turned off. However, the microprocessor does not have direct access to this information; it must copy this information (programs) into system memory to modify or execute these programs. The hard disk has a large storage capacity; therefore, large and frequently used programs such as compilers, interpreters, system programs, and application

FIGURE 1.6

Microcomputer with Disk Storage: IBM PC

SOURCE: Photograph courtesy of International Business Machines Corporation.



programs are stored on this disk. The floppy disk is generally used for user programs and to make backup copies.

The microcomputers are further classified according to their size, weight, and portability. They are called laptop and notebook. The laptop computer is a portable microcomputer that has a flat screen, a hard disk, and a 3½" floppy disk, and usually weighs around ten pounds. These computers can be battery operated or use AC power and are carried easily from place to place. These are called laptop (instead of desktop) because the size is small enough to place them in one's lap (if necessary). The notebook computer is a portable microcomputer of a notebook size (8½" × 11" × 2") and weighs around five pounds. A microcomputer smaller than the notebook computers, called a subnotebook, is also available.

WORKSTATIONS

These are high-performance cousins of the personal computers. They are used in engineering and scientific applications such as computer-aided design (CAD), computer-aided engineering (CAE), and computer-aided manufacturing (CAM). They generally include system memory and storage (hard disk) memory in gigabytes, and a high-resolution screen.

The workstations are designed around RISC (reduced instruction set computing) processors (described in Chapter 18). The RISC processors tend to be faster and more efficient than the processors used in personal computers. Some of the workstations have better performance than that of the low-end large computers.

SINGLE-BOARD MICROCOMPUTERS

These microcomputers (as shown in Figure 1.7) are primarily used in college laboratories and industries for instructional purposes or to evaluate the performance of a

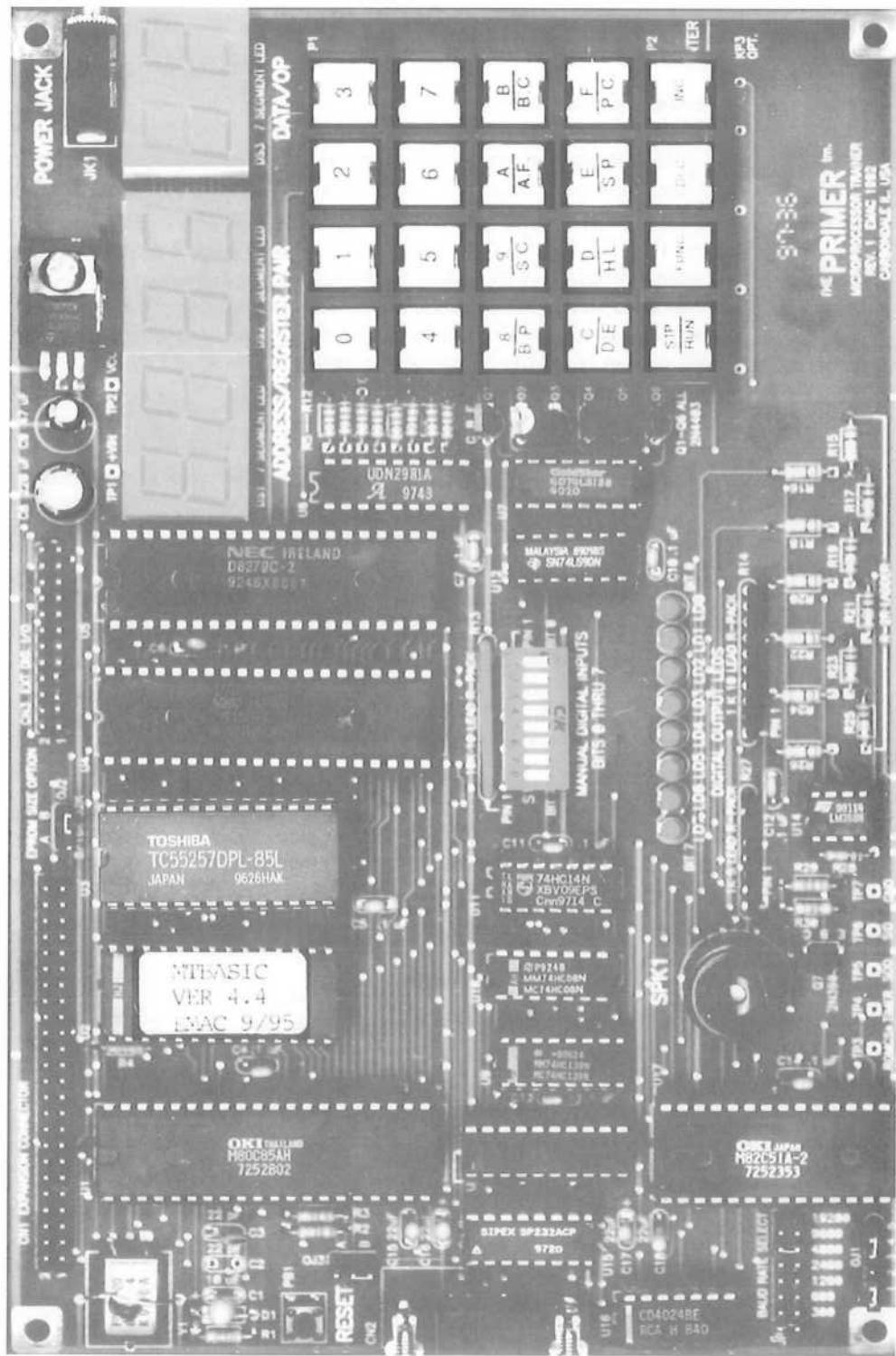


FIGURE 1.7 Single-Board Microcomputer: The Primer

given microprocessor. They can also be part of some larger systems. Typically, these microcomputers include an 8- or 16-bit microprocessor, from 256 bytes to 8K bytes of user memory, a Hex keyboard, and seven-segment LEDs as display. The interaction between the microprocessor, memory, and I/Os in these small systems is looked after by a program called a system monitor program, which is generally small in size, stored in less than 2K bytes of ROM. When a single-board microcomputer is turned on, the monitor program is in charge of the system; it monitors the keyboard inputs, interprets those keys, stores programs in memory, sends system displays to the LEDs, and enables the execution of the user programs. The function of the monitor program in a small system is similar to that of the operating system in a large system. The prices of these single-board computers range from \$100 to \$800, the average price being around \$300.

Examples of these computers include such systems as the Intel SDK 85, SDK 86, and the EMAC Primer (Figure 1.7). These are generally used to write and execute assembly language programs and to perform interfacing experiments.

SINGLE-CHIP MICROCOMPUTERS (MICROCONTROLLERS)

These microcomputers are designed on a single chip, which typically includes a microprocessor, 256 bytes of R/W memory, from 1K to 8K bytes of ROM, and several signal lines to connect I/Os. These are complete microcomputers on a chip; they are also known as microcontrollers. They are used primarily for such functions as controlling appliances and traffic lights. Typical examples of these microcomputers include such chips as the Zilog Z8, Intel MCS 51 series, Motorola 68HC11, and the Microchip Technology PIC family.

1.4

APPLICATION: MICROPROCESSOR-CONTROLLED TEMPERATURE SYSTEM (MCTS)

Based on the concepts we discussed in the previous section, let us examine a typical microprocessor-controlled temperature system. This system is expected to read the temperature in a room, display the temperature at a liquid crystal display (LCD) panel (described later), turn on a fan if the temperature is above a set point, and turn on a heater if the temperature is below a set point. We will discuss various components of this system in later chapters as we begin to learn more hardware and software concepts.

1.4.1 System Hardware

In addition to the microprocessor and memory, we need various input and output devices. The system needs a temperature sensor (described later) as an input device to sense room temperatures, and three output devices—a fan, a heater, and an LCD panel for display. Figure 1.8 represents such a system; this figure is an extension of Figure 1.3. Each component of this system is described briefly.

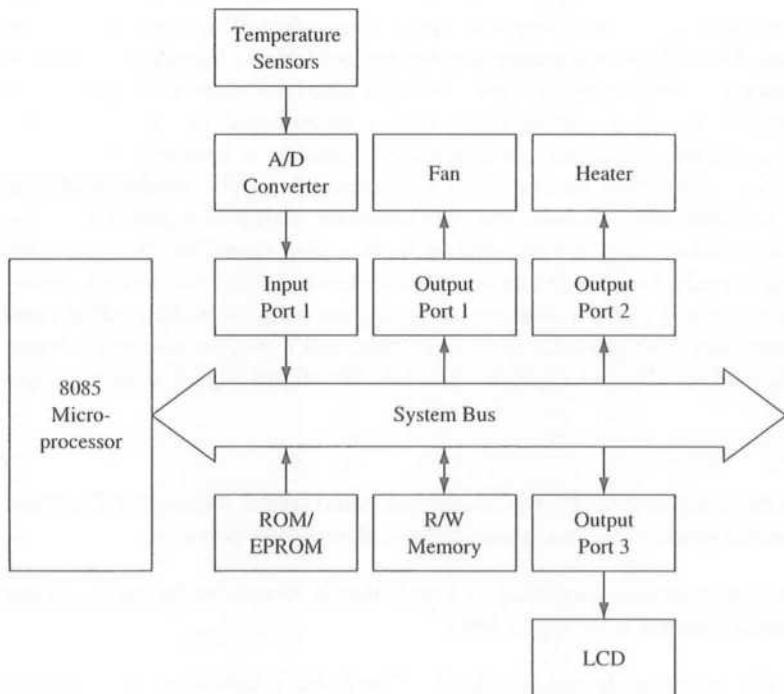


FIGURE 1.8
Microprocessor-Controlled Temperature System (MCTS)

MICROPROCESSOR

Figure 1.8 shows an 8085 processor with a system bus; we will expand this bus in various buses in the following chapter. The processor will read the binary instructions from memory and execute those instructions continuously. It will read the temperature, display it at the LCD display panel, and turn on/off the fan and the heater based on the temperature.

MEMORY

The system includes two types of memory. ROM (read-only memory) will be used to store the program, called the monitor program, that is responsible for providing the necessary instructions to the processor to monitor the system. This will be a permanent program stored in ROM and will not be altered. The R/W (read-write) memory is needed for temporary storage of data; the need for this memory will be explained in a later chapter.

INPUT

In this system, we need a device that can translate temperature (measurement of heat) into an equivalent electrical signal; a device that translates one form of energy into another

form is called a transducer. For example, a microphone is a transducer that converts sound energy into an electrical signal, and a thermocouple is a transducer that converts heat into an electrical signal. These days temperature sensors are available as integrated circuits. A **temperature sensor** is a three-terminal semiconductor electronic device (a chip similar to a transistor) that generates a voltage signal that is proportional to the temperature. However, this is an analog signal and our processor is capable of handling only binary bits. Therefore, this signal must be converted into digital bits. The **analog-to-digital (A/D converter)** performs that function. The A/D converter, shown in Figure 1.8, is also an electronic semiconductor chip that converts an input analog signal into the equivalent eight binary output signals. In microprocessor-based systems, devices that provide binary inputs (data) are connected to the processor using devices such as buffers called input ports. In our system, this A/D converter is an input port, and it will be assigned a binary number called an address. The microprocessor reads this digital signal from the input port.

OUTPUT

Figure 1.8 shows three output devices: fan, heater and liquid crystal display (LCD). These devices are connected to the processor using latches called output ports.

Fan This is an output device, identified as Port1, that is turned on by the processor when the temperature reaches a set higher limit.

Heater This is also an output device, identified as Port2, that is turned on by the processor when the temperature reaches a set lower limit.

Liquid Crystal Display (LCD) This display is made of crystal material placed between two plates in the form of a dot matrix or segments. It can display letters, decimal digits, or graphic characters. The LCD in Figure 1.8 will be used to display temperatures.

1.4.2 System Software (Programs)

The program that runs the system is called a monitor program or system software. Generally, the entire program is divided into subtasks and written as independent modules, and it is stored in ROM (or EPROM). When the system is reset, the microprocessor reads the binary command (instruction) from the first memory location of ROM and continues in sequence to execute the program.

SUMMARY

The various concepts and terms discussed in this chapter are summarized here:

Computer Structure

- Digital computer**—a programmable machine that processes binary data. It is traditionally represented by five components: CPU, ALU plus control unit, memory, input, and output.

- **CPU**—the central processing unit. The group of circuits that processes data and provides control signals and timing. It includes the arithmetic/logic unit, registers, instruction decoder, and the control unit.
- **ALU**—**the group of circuits that performs arithmetic and logic operations.** The ALU is a part of the CPU.
- **Control unit**—**the group of circuits that provides timing and signals to all operations in the computer and controls data flow.**
- **Memory**—a medium that stores binary information (instructions and data).
- **Input**—a device that transfers information from the outside world to the computer.
- **Output**—a device that transfers information from the computer to the outside world.

Scale of Integration

- **SSI** (small-scale integration)—the process of designing a few circuits on a single chip. The term refers to the technology used to fabricate discrete logic gates on a chip.
- **MSI** (medium-scale integration)—the process of designing more than a hundred gates on a single chip.
- **LSI** (large-scale integration)—the process of designing more than a thousand gates on a single chip. Similarly, the terms *VLSI* (very-large-scale integration) and *SLSI* (super-large-scale integration) are used to indicate the scale of integration.

Microcomputers

- **Microprocessor (MPU)**—a semiconductor device (integrated circuit) manufactured by using the LSI technique. It includes the ALU, register arrays, and control circuits on a single chip. The term *MPU* is also synonymous with the microprocessor (see Section 2.2 for additional details).
- **Microprocessor-based product**—a machine or product that uses a microprocessor to run or execute its operations. It is represented by three components: microprocessor, memory, and I/O (input/output).
- **Microcontroller**—a device that includes microprocessor, memory, and I/O signal lines on a single chip, fabricated using VLSI technology.
- **Microcomputer**—a computer that is designed using a microprocessor as its CPU. It includes microprocessor, memory, and I/O (input/output).
- **Bus**—a group of lines used to transfer bits between the microprocessor and other components of the computer system.
- **RAM (Random-Access memory)**—see R/Wm.
- **ROM (Read-Only memory)**—a memory that stores binary information permanently. The information can be read from this memory but cannot be altered.
- **R/WM (Read/Write memory)**—a memory that stores binary information during the operation of the computer. This memory is used as a writing pad to write user programs and data. The information stored in this memory can be read and altered easily.

Computer Languages

- **Bit**—a binary digit, 0 or 1.
- **Byte**—a group of eight bits.

- Nibble**—a group of four bits.
- Word**—a group of bits the computer recognizes and processes at a time.
- Instruction**—a command in binary that is recognized and executed by the computer to accomplish a task. Some instructions are designed with one word, and some require multiple words.
- Mnemonic**—a combination of letters to suggest the operation of an instruction.
- Program**—a set of instructions written in a specific sequence for the computer to accomplish a given task.
- Machine language**—the binary medium of communication with a computer through a designed set of instructions specific to each computer.
- Assembly language**—a medium of communication with a computer in which programs are written in mnemonics. An assembly language is specific to a given computer.
- Low-level language**—a medium of communication that is machine-dependent or specific to a given computer. The machine and the assembly languages of a computer are considered low-level languages. Programs written in these languages are not transferable to different types of machines.
- High-level language**—a medium of communication that is independent of a given computer. Programs are written in English-like words, and they can be executed on a machine using a translator (a compiler or an interpreter).
- Source code**—a program written either in mnemonics of an assembly language or in English-like statements of a high-level language (before it is assembled or compiled).
- Compiler**—a program that translates English-like words of a high-level language into the machine language of a computer. A compiler reads a given program, called a source code, in its entirety and then translates the program into the machine language, which is called an object code.
- Interpreter**—a program that translates the English-like statements of a high-level language into the machine language of a computer. An interpreter translates one statement at a time from a source code to an object code.
- Assembler**—a computer program that translates an assembly language program from mnemonics to the binary machine code of a computer.
- Manual assembly**—a procedure of looking up the machine codes manually from the instruction set of a computer and entering those into the computer through a keyboard.
- ASCII**—American Standard Code for Information Interchange. This is a 7-bit alphanumeric code with 128 combinations. Each combination is assigned to either a letter, decimal digit, a symbol, or a machine command.
- Extended ASCII**—an 8-bit code with 256 combinations. The ASCII code is extended from seven bits to eight bits to include additional graphic symbols.
- Operating system**—a set of programs that manages interaction between hardware and software. It is responsible primarily for storing information on disks and for communication between microprocessor, memory, and peripherals.
- Monitor program**—a program that interprets the input from a keyboard and converts the input into its binary equivalent.

LOOKING AHEAD

This chapter has given a brief introduction to computer organization and computer languages, with emphasis on the 8085 microprocessor and its assembly language. The last section has provided an overview of the entire spectrum of computers, including their salient features and applications. The primary focus of this book is on the architectural details of the 8085 microprocessor and its industrial applications. Heavy emphasis also is put on assembly language programming in the context of these applications. In the microcomputer field, little separation is made between hardware and software, especially in applications where assembly language is necessary. In designing a microprocessor-based product, hardware and software tasks are carried out concurrently because a decision in one area affects the planning of the other area. Some functions can be performed either through hardware or software, and a designer needs to consider both approaches. This book focuses on the tradeoffs between the two approaches as a design philosophy.

Chapter 2 introduces 8085 assembly language programming and provides an overview of the 8085 instruction set. Chapter 3 expands on the architectural concepts of microcomputers, discussed in this chapter, using an illustration of a generalized microprocessor. The chapter discusses each component of the block diagram shown in Figure 1.3. Chapter 4 focuses on the architecture details of the 8085 microprocessor and memory interfacing, and Chapter 5 on I/O interfacing.

QUESTIONS AND PROBLEMS

1. List the components of a computer.
2. Explain the functions of each component of a computer.
3. What is a microprocessor? What is the difference between a microprocessor and a CPU?
4. Explain the difference between a microprocessor and a microcomputer.
5. Explain these terms: *SSI*, *MSI*, and *LSI*.
6. Define *bit*, *byte*, *word*, and *instruction*.
7. How many bytes make a word of 32 bits?
8. Specify the number of registers in a 2K memory chip.
9. Calculate the number of registers in a 64K memory board.
10. Explain the difference between the machine language and the assembly language of the 8085 microprocessor.
11. What is an assembler?
12. What are low- and high-level languages?
13. Explain the difference between a compiler and an interpreter.
14. What are the advantages of an assembly language in comparison with high-level languages?
15. What is an ASCII code?

16. Identify the difference between the ASCII and the extended ASCII codes.
17. Find the ASCII codes for the letters "A," "Z," and "m" from the ASCII table in Appendix E.
18. What is an operating system?
19. Identify the following peripherals as input or output: scanner, digital camera, printer, keyboard, mouse.

2

Introduction to 8085 Assembly Language Programming

As defined in Chapter 1, an assembly language program is a set of instructions written in the mnemonics of a given microprocessor. These instructions are the commands to the microprocessor to be executed in the given sequence to accomplish a task. To write such programs for the 8085 microprocessor, we should be familiar with the programming model and the instruction set of the microprocessor. This chapter provides such an overview of the 8085 microprocessor.

The 8085 instruction set is classified into five different groups: data transfer, arithmetic, logic, branch, and machine control; each of these groups is illustrated with examples. The chapter also discusses the instruction format and various addressing modes. A simple problem of adding two Hex numbers is used to illustrate writing, assembling, and executing a program. The flowcharting technique and symbols are discussed in the context of the problem.

The chapter concludes with a list of selected 8085 instructions.

OBJECTIVES

- Explain the various functions of the registers in the 8085 programming model.
- Define the term *flag* and explain how the flags are affected.
- Explain the terms *operation code* (opcode) and the *operand*, and illustrate these terms by writing instructions.
- Classify the instructions in terms of their word size and specify the number of memory registers required to store the instructions in memory.
- List the five categories of the 8085 instruction set.
- Define and explain the term *addressing mode*.

- Write logical steps to solve a simple programming problem.
- Draw a flowchart from the logical steps of a given programming problem.
- Translate the flowchart into mnemonics and convert the mnemonics into Hex code for a given programming problem.

2.1

THE 8085 PROGRAMMING MODEL

A model is a conceptual representation of a real object. It can take many forms, such as text description, a drawing, or a built structure. Most of us have seen an architectural model of a building. Similarly, the microprocessor can be represented in terms of its hardware (physical electronic components) and a programming model (information needed to write programs). In Chapter 1, we described a simplified hardware model of a microprocessor as a part of the microprocessor-based system (Figure 1.3). It showed three components: ALU, register array, and control. Figure 2.1 shows a hardware model and a programming model specific to the 8085 microprocessor.

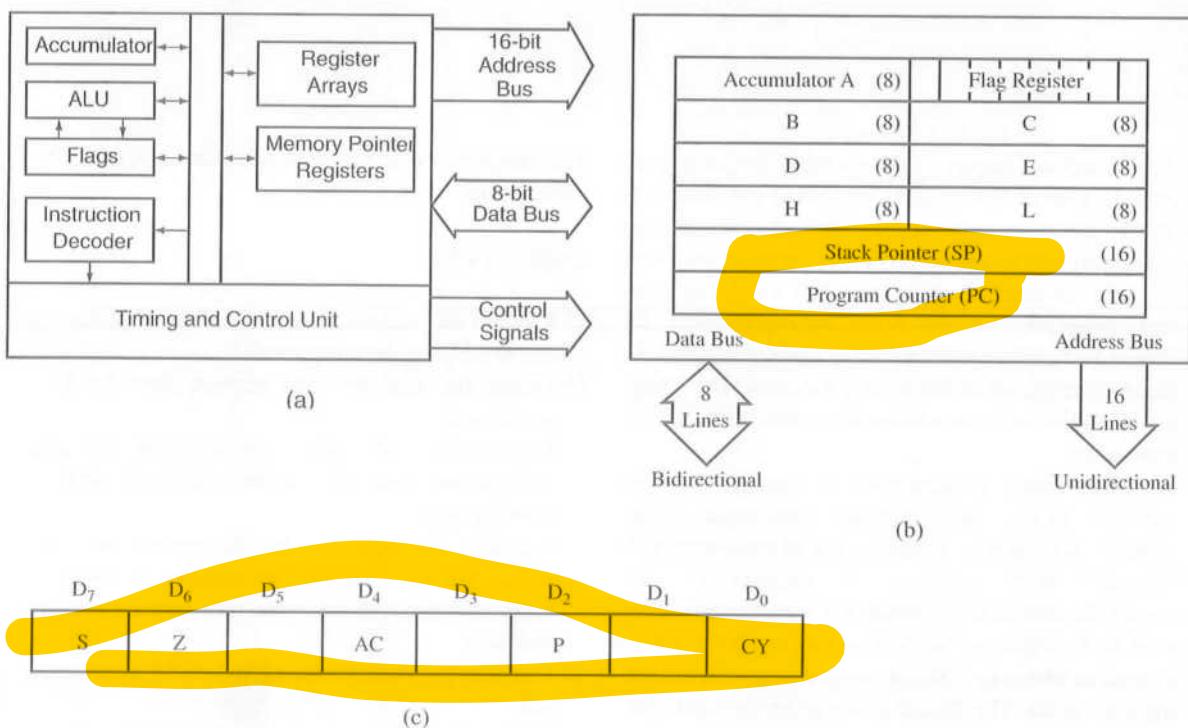


FIGURE 2.1

8085 Hardware Model (a), Programming Model (b), and Flag Register (c)

2.1.1 8085 Hardware Model

The hardware model in Figure 2.1(a) shows two major segments. One segment includes the arithmetic/logic unit (ALU) and an 8-bit register called an accumulator, instruction decoder, and flags. The second segment shows 8-bit and 16-bit registers. Both segments are connected with various internal connections called an internal bus. The arithmetic and logical operations are performed in the ALU. Results are stored in the accumulator, and flip-flops, called flags, are set or reset to reflect the results (see Figure 2.1a). There are three buses: a 16-bit unidirectional address bus, an 8-bit bidirectional data bus, and a control bus. In Chapter 1, these three buses were shown as one system bus. The 8085 processor uses the 16-bit address bus to send out memory addresses, the 8-bit data bus to transfer data, and the control bus for timing signals. The details of the hardware model are included in later chapters.

2.1.2 8085 Programming Model

The programming model consists of some segments of the ALU and the registers. This model does not reflect the physical structure of the 8085 but includes the information that is critical in writing assembly language programs. The model includes six registers, one accumulator, and one flag register, as shown in Figure 2.1(b). In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows.

REGISTERS

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B, C, D, E, H, and L, as shown in Figure 2.1(b). They can be combined as register pairs—BC, DE, and HL—to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

ACCUMULATOR

The accumulator is an 8-bit register that is part of the arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

FLAGS

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; they are listed in Table 2.1 and their bit positions in the flag register are shown in Figure 2.1(c). The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through software instructions. For example, the instruction JC (Jump On Carry) is implemented to change the sequence of a program when the CY flag is set. The thorough understanding of flags is essential in writing assembly language programs.

TABLE 2.1
The 8085 Flags

The following flags are set or reset after the execution of an arithmetic or logic operation; data copy instructions do not affect any flags. See the instruction set (Appendix F) to find how flags are affected by an instruction.

- Z—Zero: The Zero flag is set to 1 when the result is zero; otherwise it is reset.
- CY—Carry: If an arithmetic operation results in a carry, the CY flag is set; otherwise it is reset.
- S—Sign: The Sign flag is set if bit D₇ of the result = 1; otherwise it is reset.
- P—Parity: If the result has an even number of 1s, the flag is set; for an odd number of 1s, the flag is reset.
- AC—Auxiliary Carry: In an arithmetic operation, when a carry is generated by digit D₃ and passed to digit D₄, the AC flag is set. This flag is used internally for BCD (binary-coded decimal) operations; there is no Jump instruction associated with this flag.

PROGRAM COUNTER (PC) AND STACK POINTER (SP)*

These are two 16-bit registers used to hold memory addresses. The size of these registers is 16 bits because the memory addresses are 16 bits.

The microprocessor uses the PC register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading a 16-bit address in the stack pointer. The stack concept is explained in Chapter 9, "Stack and Subroutines."

This programming model will be used in subsequent chapters to examine how these registers are affected after the execution of an instruction.

2.2

INSTRUCTION CLASSIFICATION

An **instruction** is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the **instruction set**, determines what functions the microprocessor can perform. The 8085 microprocessor includes the instruction set of its predecessor, the 8080A, plus two additional instructions.

*The concept of stack memory is difficult to explain at this time; it is not necessary for the reader to understand stack memory until subroutines are discussed. It is included here only to provide continuity in the discussion of programmable registers and microprocessor operations. This concept will be explained more fully in Chapter 9.

2.2.1 The 8085 Instruction Set

The 8085 instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

DATA TRANSFER (COPY) OPERATIONS

This group of instructions copies data from a location called a source to another location, called a destination, without modifying the contents of the source. In technical manuals, the term *data transfer* is used for this copying function. However, the term *transfer* is misleading; it creates the impression that the contents of a source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

Types	Examples
<input type="checkbox"/> Between registers	Copy the contents of register B into register D.
<input type="checkbox"/> Specific data byte to a register or a memory location	Load register B with the data byte 32H.
<input type="checkbox"/> Between a memory location and a register	From the memory location 2000H to register B.
<input type="checkbox"/> Between an I/O device and the accumulator	From an input keyboard to the accumulator.

ARITHMETIC OPERATIONS

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

- Addition**—Any 8-bit number, or the contents of a register, or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g., the contents of register B cannot be added directly to the contents of register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.
- Subtraction**—Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2's complement, and the results, if negative, are expressed in 2's complement. No two other registers can be subtracted directly.
- Increment/Decrement**—The 8-bit contents of a register or a memory location can be incremented or decremented by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decremented by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e., they can be performed in any one of the registers or in a memory location.

LOGICAL OPERATIONS

These instructions perform various logical operations with the contents of the accumulator.

- **AND, OR, Exclusive-OR**—Any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, ORed, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.
- **Rotate**—Each bit in the accumulator can be shifted either left or right to the next position.
- **Compare**—Any 8-bit number, or the contents of a register, or a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.
- **Complement**—The contents of the accumulator can be complemented; all 0s are replaced by 1s and all 1s are replaced by 0s.

BRANCHING OPERATIONS

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

- **Jump**—Conditional jumps are an important aspect of the decision-making process in programming. These instructions test for a certain condition (e.g., Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called *unconditional jump*.
- **Call, Return, and Restart**—These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

MACHINE CONTROL OPERATIONS

These instructions control machine functions such as Halt, Interrupt, or do nothing.

2.2.2 Review of the 8085 Operations

The microprocessor operations related to data manipulation can be summarized in four functions:

1. copying data
2. performing arithmetic operations
3. performing logical operations
4. testing for a given condition and altering the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Arithmetic and logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some exceptions). The flags are affected according to the results.

3. Any register including memory can be used for increment and decrement.
4. A program sequence can be changed either conditionally or by testing for a given data condition.

INSTRUCTION, DATA FORMAT, AND STORAGE

2.3

An instruction is a command to the microprocessor to perform a given task on specified data. Each instruction has two parts: one is the task to be performed, called the **operation code (opcode)**, and the second is the data to be operated on, called the **operand**. The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or an 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

2.3.1 Instruction Word Size

The 8085 instruction set is classified into the following three groups according to word size or byte size.

In the 8085, “byte” and “word” are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

1. 1-byte instructions
2. 2-byte instructions
3. 3-byte instructions

ONE-BYTE INSTRUCTIONS

A 1-byte instruction includes the opcode and the operand in the same byte. For example:

Task	Opcode	Operand*	Binary Code	Hex Code
Copy the contents of the accumulator in register C.	MOV	C,A	0100 1111	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert (complement) each bit in the accumulator.	CMA		0010 1111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location.

*In the operand, the destination register C is shown first, followed by the source register A.

TWO-BYTE INSTRUCTIONS

In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. For example:

Task	Opcode	Operand	Binary Code	Hex Code			
Load an 8-bit data byte in the accumulator.	MVI	A,32H	<table border="1"> <tr><td>0011 1110</td></tr> <tr><td>0011 0010</td></tr> </table>	0011 1110	0011 0010	3E 32	First Byte Second Byte
0011 1110							
0011 0010							
Load an 8-bit data byte in register B.	MVI	B,F2H	<table border="1"> <tr><td>0000 0110</td></tr> <tr><td>1111 0010</td></tr> </table>	0000 0110	1111 0010	06 F2	First Byte Second Byte
0000 0110							
1111 0010							

These instructions would require two memory locations each to store the binary codes. The data bytes 32H and F2H are selected arbitrarily as examples.

THREE-BYTE INSTRUCTIONS

In a 3-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address. For example:

Task	Opcode	Operand	Binary Code	Hex Code*				
Load contents of memory 2050H into A.	LDA	2050H	<table border="1"> <tr><td>0011 1010</td></tr> <tr><td>0101 0000</td></tr> <tr><td>0010 0000</td></tr> </table>	0011 1010	0101 0000	0010 0000	3A 50 20	First Byte Second Byte Third Byte
0011 1010								
0101 0000								
0010 0000								
Transfer the program sequence to memory location 2085H.	JMP	2085H	<table border="1"> <tr><td>1100 0011</td></tr> <tr><td>1000 0101</td></tr> <tr><td>0010 0000</td></tr> </table>	1100 0011	1000 0101	0010 0000	C3 85 20	First Byte Second Byte Third Byte
1100 0011								
1000 0101								
0010 0000								

These instructions would require three memory locations each to store the binary codes.

These commands are in many ways similar to our everyday conversation. For example, while eating in a restaurant, we may make the following requests and orders:

1. Pass (the) butter.
2. Pass (the) bowl.
3. (Let us) eat.

*The 16-bit addresses are stored in memory locations in reversed order, the low-order byte first, followed by the high-order byte.

4. I will have combination 17 (on the menu).
5. I will have what Susie ordered.

The first request specifies the exact item; it is similar to the instruction for loading a specific data byte in a register. The second request mentions the bowl rather than the contents, even though one is interested in the contents of the bowl. It is similar to the instruction MOV C,A where registers (bowls) are specified rather than data. The third suggestion (let us eat) assumes that one knows what to eat. It is similar to the instruction Complement, which implicitly assumes that the operand is the accumulator. In the fourth sentence, the location of the item on the menu is specified and not the actual item. It is similar to the instruction: Transfer the data byte from the location 2050H. The last order (what Susie ordered) is specified indirectly. It is similar to an instruction that specifies a memory location through the contents of a register pair. (Examples of the last two types of instruction are illustrated in later chapters.)

These various ways of specifying data are called the **addressing modes**. Although microprocessor instructions require one or more words to specify the operands, the notations and conventions used in specifying the operands have very little to do with the operation of the microprocessor. The mnemonic letters used to specify a command are chosen (somewhat arbitrarily) by the manufacturer. When an instruction is stored in memory, it is stored in binary code, the only code the microprocessor is capable of reading and understanding. The conventions used in specifying the instructions are valuable in terms of keeping uniformity in different programs and in writing assemblers. The important point to remember is that the microprocessor neither reads nor understands mnemonics or hexadecimal numbers.

2.3.2 Opcode Format

To understand operation codes, we need to examine how an instruction is designed into the microprocessor. This information will be useful in reading a user's manual, in which operation codes are specified in binary format and 8-bits are divided in various groups. However, this information is not necessary to understand assembly language programming.

In the design of the 8085 microprocessor chip, all operations, registers, and status flags are identified with a specific code. For example, all internal registers are identified as follows:

Code	Registers	Code	Register Pairs
000	B	00	BC
001	C	01	DE
010	D	10	HL
011	E	11	AF OR SP
100	H		
101	L		
111	A		
110	Reserved for Memory-Related operation		

Some of the operation codes are identified as follows:

Function	Operation Code
1. Rotate each bit of the accumulator to the left by one position.	00000111 = 07H (8-bit opcode)
2. Add the contents of a register to the accumulator.	10000 SSS (5-bit opcode—3 bits are reserved for a register)

This instruction is completed by adding the code of the register. For example,

Add	:	10000
Register B	:	000
to A	:	Implicit
Binary Instruction:		<u>10000</u> <u>000</u> = 80H
		Add Reg.B

In assembly language, this is expressed as

Opcode	Operand	Hex Code
ADD	B	80H

3. MOVE (Copy) the content of register Rs (source) to register Rd (destination)	01	DDD	SSS
	2-bit Opcode for MOVE	Reg. Rd	Reg. Rs

This instruction is completed by adding the codes of two registers. For example,

Move (copy) the content:	0 1		
To register C	:	0 0 1 (DDD)	
From register A	:	1 1 1 (SSS)	
Binary Instruction	:	<u>0 1</u> <u>0 0 1</u> <u>1 1 1</u> → 4FH	
		Opcode	Operand

In assembly language, this is expressed as

Opcode	Operand	Hex Code
MOV	C,A	4F

Please note that the first register is the destination and the second register is the source—from A to C—which appears reversed for a general pattern from left to right. Typically, in the 8085 user's manual the data transfer (copy) instruction is shown as follows:

MOV rl, r2*

0	1	D	D	D	S	S	S
---	---	---	---	---	---	---	---

2.3.3 Data Format

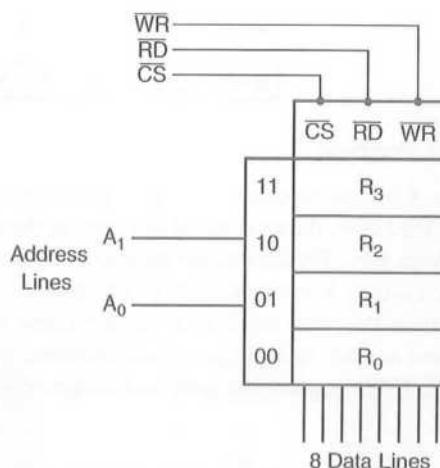
The 8085 is an 8-bit microprocessor, and it processes (copy, add, subtract, etc.) only binary numbers. However, the real world operates in decimal numbers and languages of alphabets and characters. Therefore, we need to code binary numbers into different media. Let us examine coding. What is the letter "A"? It is a symbol representing a certain sound in a visual medium that eyes can recognize. Similarly, we can represent or code groups of bits into different media. In 8-bit processor systems, commonly used codes and data formats are ASCII, BCD, signed integers, and unsigned integers. They are explained as follows.

- **ASCII Code**—This is a 7-bit alphanumeric code that represents decimal numbers, English alphabets, and nonprintable characters such as carriage return. Extended ASCII is an 8-bit code. The additional numbers (beyond 7-bit ASCII code) represent graphical characters. This code was discussed in Chapter 1 (Section 1.24).
- **BCD Code**—The term *BCD* stands for binary-coded decimal; it is used for decimal numbers. The decimal numbering system has ten digits, 0 to 9. Therefore, we need only four bits to represent ten digits from 0000 to 1001. The remaining numbers, 1010 (A) to 1111 (F), are considered invalid. An 8-bit register in the 8085 can accommodate two BCD numbers.
- **Signed Integer**—A signed integer is either a positive number or a negative number. In an 8-bit processor, the most significant digit, D₇, is used for the sign; 0 represents the positive sign and 1 represents the negative sign. The remaining seven bits, D₆–D₀, represent the magnitude of an integer. Therefore, the largest positive integer that can be processed by the 8085 at one time is 0111 1111 (7FH); the remaining Hex numbers, 80H to FFH, are considered negative numbers. However, all negative numbers in this microprocessor are represented in 2's complement format (see Appendix A.2 for additional explanation).
- **Unsigned Integers**—An integer without a sign can be represented by all the 8 bits in a microprocessor register. Therefore, the largest number that can be processed at one time is FFH. However, this does not imply that the 8085 microprocessor is limited to handling only 8-bit numbers. Numbers larger than 8 bits (such as 16-bit or 24-bit numbers) are processed by dividing them in groups of 8 bits.

Now let us examine how the microprocessor interprets any number. Let us assume that after performing some operations the result in the accumulator is 0100 0001 (41H). This number can have many interpretations: (1) It is an unsigned number equivalent to 65

*In this text, rl is specified as Rd and r2 is specified as Rs to indicate destination and source.

FIGURE 2.2
Simplified Memory Model



in decimal; (2) it is a BCD number representing 41 decimal; (3) it is the ASCII capital letter “A”; or (4) it is a group of 8 bits where bits D₆ and D₀ turn on and the remaining bits turn off output devices. The processor processes binary bits; it is up to the user to interpret the result. In our example, the number 41H can be displayed on a screen as an ASCII “A” or 41 BCD.

2.3.4 Instruction and Data Storage: Memory

Now the next question is: How do we provide this information to the processor? It is provided by another electronic storage chip called memory. In some ways, the term *memory* is a misnomer; it is a storage of binary bits. Memory chips used in most systems are nothing but 8-bit registers stacked one above the other as shown in our memory model in Figure 2.2. It includes only four registers, and each register can store 8 bits. This chip can be referred to as a 4-byte or 32 (4 × 8) bits memory chip. It has two address lines, A₀ and A₁, to identify four registers, 8 data lines to store 8 bits, and three timing or control signals: Read (RD), Write (WR), and Chip Select (CS); all control signals are designed to be active low, indicated by bars over the symbols. The processor can select this chip and identify its register, and store (Write) or access (Read) 8 bits at a time. Figure 2.2 shows only four registers to simplify the explanation; in reality, the size of a memory chip is in kilo- or megabytes. The memory addresses assigned to these registers are determined by the interfacing logic used in the system. In Chapters 3 and 4, we will discuss memory addressing and interfacing in more detail.

2.4

HOW TO WRITE, ASSEMBLE, AND EXECUTE A SIMPLE PROGRAM

A program is a sequence of instructions written to tell a computer to perform a specific function. The instructions are selected from the instruction set of the microprocessor. To write a program, divide a given problem in small steps in terms of the operations the 8085

can perform, then translate these steps into instructions. Writing a simple program of adding two numbers in the 8085 language is illustrated below.

2.4.1 Illustrative Program: Adding Two Hexadecimal Numbers

PROBLEM STATEMENT

Write instructions to load the two hexadecimal numbers 32H and 48H in registers A and B, respectively. Add the numbers, and display the sum at the LED output port PORT1.

PROBLEM ANALYSIS

Even though this is a simple problem, it is necessary to divide the problem into small steps to examine the process of writing programs. The wording of the problem provides sufficient clues for the necessary steps. They are as follows:

1. Load the numbers in the registers.
2. Add the numbers.
3. Display the sum at the output port PORT1.

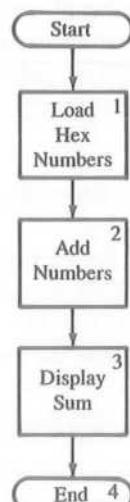
FLOWCHART

The steps listed in the problem analysis and the sequence can be represented in a block diagram, called a flowchart. Figure 2.3 shows such a flowchart representing the above steps. This is a simple flowchart, and the steps are self-explanatory. We will discuss flowcharting in the next chapter.

ASSEMBLY LANGUAGE PROGRAM

To write an assembly language program, we need to translate the blocks shown in the flowchart into 8085 operations and then, subsequently, into mnemonics. By examining the blocks, we can classify them into three types of operations: Blocks 1 and 3 are copy op-

FIGURE 2.3
Flowchart: Adding Two
Numbers



erations; Block 2 is an arithmetic operation; and Block 4 is a machine-control operation. To translate these steps into assembly and machine languages, you should review the instruction set. The translation of each block into mnemonics with comments is shown as follows:

Block 1:	MVI A,32H	Load register A with 32H
	MVI B,48H	Load register B with 48H
Block 2:	ADD B	Add two bytes and save the sum in A
Block 3:	OUT 01H	Display accumulator contents at port 01H
Block 4:	HALT	End

FROM ASSEMBLY LANGUAGE TO HEX CODE

To convert the mnemonics into Hex code, we need to look up the code in the 8085 instruction set; this is called either manual or hand assembly.

Mnemonics	Hex Code	
MVI A,32H	3E 32	2-byte instruction
MVI B,48H	06 48	2-byte instruction
ADD B	80	1-byte instruction
OUT 01H	D3 01	2-byte instruction
HLT	76	1-byte instruction

STORING IN MEMORY AND CONVERTING FROM HEX CODE TO BINARY CODE

To store the program in R/W memory of a single-board microcomputer and display the output, we need to know the memory addresses and the output port address. Let us assume that R/W memory ranges from 2000H to 20FFH, and the system has an LED output port with the address 01H. Now, to enter the program:

1. Reset the system by pushing the RESET key.
2. Enter the first memory address using Hex keys where the program should be stored. Let us assume it is 2000H.
3. Enter each machine code by pushing Hex keys. For example, to enter the first machine code, push the 3, E, and STORE keys. (The STORE key may be labeled differently in different systems.) When you push the STORE key, the program will store the machine code in memory location 2000H and upgrade the memory address to 2001H.
4. Repeat Step 3 until the last machine code, 76H.
5. Reset the system.

Now the question is: How does the Hex code get converted into binary code? The answer lies with the Monitor program stored in Read-Only memory (or EPROM) of the micro-

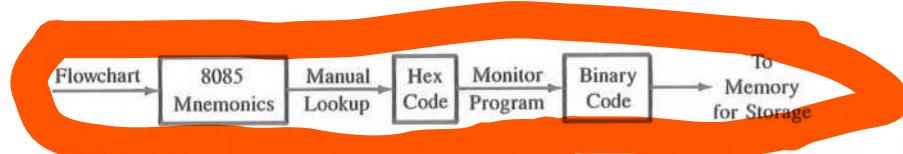


FIGURE 2.4

Manual Assembly Process

computer system. An important function of the Monitor program is to check the keys and convert Hex code into binary code. The entire process of manual assembly is shown in Figure 2.4.

In this illustrative example, the program will be stored in memory as follows:

Mnemonics	Hex Code	Memory Contents	Memory Address
MVI A,32H	3E 32	0 0 1 1 1 1 1 0 0 0 1 1 0 0 1 0	2000 2001
MVI B,48H	06 48	0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0	2002 2003
ADD B	80	1 0 0 0 0 0 0 0	2004
OUT 01H	D3 01	1 1 0 1 0 0 1 1 0 0 0 0 0 0 0 1	2005 2006
HLT	76	0 1 1 1 1 1 1 0	2007

This program has eight machine codes and will require eight memory locations to store the program. The critical concept that needs to be emphasized here is that the microprocessor can understand and execute only the binary instructions (or data); everything else (mnemonics, Hex code, comments) is for the convenience of human beings.

EXECUTING THE PROGRAM

To execute the program, we need to tell the microprocessor where the program begins by entering the memory address 2000H. Now, we can push the Execute key (or the key with a similar label) to begin the execution. As soon as the Execute function key is pushed, the microprocessor loads 2000H in the program counter, and the program control is transferred from the Monitor program to our program.

The microprocessor begins to read one machine code at a time, and when it fetches the complete instruction, it executes that instruction. For example, it will fetch the machine codes stored in memory locations 2000H and 2001H and execute the instruction MVI A,32H; thus it will load 32H in register A. The ADD instruction will add the two numbers, and the OUT instruction will display the answer 7AH ($32H + 48H = 7AH$) at the LED port. It continues to execute instructions until it fetches the HLT instruction.

RECOGNIZING THE NUMBER OF BYTES IN AN INSTRUCTION

Students who are introduced to an assembly language for the first time should hand assemble at least a few small programs. Such exercises can clarify the relationship among instruction codes, data, memory registers, and memory addressing. One of the stumbling blocks in hand

assembly is in recognizing the number of bytes in a given instruction. The following clues can be used to recognize the number of bytes in an instruction of the 8085 microprocessor.

1. One-byte instruction—A mnemonic followed by a letter (or two letters) representing the registers (such as A, B, C, D, E, H, L, M, and SP) is a one-byte instruction. Instructions in which registers are implicit are also one-byte instructions.
Examples: (a) MOV A, B; (b) DCX SP; (c) RRC
2. Two-byte instruction—A mnemonic followed by 8-bit (byte) is a two-byte instruction.
Examples: (a) MVI A, 8-bit; (b) ADI 8-bit
3. Three-byte instruction—A mnemonic followed by 16-bit (also terms such as adr or dble) is a three-byte instruction.
Examples: (a) LXI B, 16-bit (dble); (b) JNZ 16-bit (adr); (c) CALL 16-bit (adr)

In writing assembly language programs, we can assign memory addresses in a sequence once we know the number of bytes in a given instruction. For example, a three-byte instruction has three Hex codes and requires three memory locations in a sequence. In hand assembly, omitting a byte inadvertently can have a disastrous effect on program execution, as explained in the next section.

2.4.2 How Does a Microprocessor Differentiate Between Data and Instruction Code?

The microprocessor is a sequential machine. As soon as a microprocessor-based system is turned on, it begins the execution of the code in memory. The execution continues in a sequence, one code after another (one memory location after another) at the speed of its clock until the system is turned off (or the clock stops). If an unconditional loop is set up in a program, the execution will continue until the system is either reset or turned off.

Now a puzzling question is: How does the microprocessor differentiate between a code and data when both are binary numbers? The answer lies in the fact that the microprocessor interprets the first byte it fetches as an opcode. When the 8085 is reset, its program counter is cleared to 0000H and it fetches the first code from the location 0000H. In the example of the previous section, we tell the processor that our program begins at location 2000H. The first code it fetches is 3EH. When it decodes that code, it knows that it is a two-byte instruction. Therefore, it assumes that the second code, 32H, is a data byte. If we forget to enter 32H and enter the next code, 06H, instead, the 8085 will load 06H in the accumulator, interpret the next code, 48H, as an opcode, and continue the execution in sequence. As a consequence, we may encounter a totally unexpected result.

2.5

OVERVIEW OF THE 8085 INSTRUCTION SET

The 8085 microprocessor instruction set has 74 operation codes that result in 246 instructions. The set includes all the 8080A instructions plus two additional instructions (SIM and RIM, related to serial I/O). It is an overwhelming experience for a beginner to

study these instructions. You are strongly advised not to attempt to read all these instructions at one time. However, you should be able to grasp an overview of the set by examining the frequently used instructions listed below.*

The following notations are used in the description of the instructions.

R = 8085 8-bit register	(A, B, C, D, E, H, L)
M = Memory register (location)	
Rs = Register source	
Rd = Register destination	(A, B, C, D, E, H, L)
Rp = Register pair	(BC, DE, HL, SP)
() = Contents of	

1. Data Transfer (Copy) Instructions. These instructions perform the following six operations.

- Load an 8-bit number in a register
- Copy from register to register
- Copy between I/O and accumulator
- Load 16-bit number in a register pair
- Copy between register and memory
- Copy between registers and stack memory

Mnemonics	Examples	Operation
1.1 MVI R,** 8-bit	MVI B, 4FH	Load 8-bit data (byte) in a register
1.2 MOV Rd, Rs**	MOV B, A MOV C, B	Copy data from source register Rs into destination register Rd
1.3 LXI Rp,** 16-bit	LXI B, 2050H	Load 16-bit number in a register pair
1.4 OUT 8-bit (port address)	OUT 01H	Send (write) data byte from the accumulator to an output device
1.5 IN 8-bit (port address)	IN 07H	Accept (read) data byte from an input device and place it in the accumulator
1.6 LDA 16-bit	LDA 2050H	Copy the data byte into A from the memory specified by 16-bit address
1.7 STA 16-bit	STA 2070H	Copy the data byte from A into the memory specified by 16-bit address
1.8 LDAX Rp	LDAX B	Copy the data byte into A from the memory specified by the address in the register pair
1.9 STAX Rp	STAX D	Copy the data byte from A into the memory specified by the address in the register pair

*These instructions are explained and illustrated in Chapters 6 and 7. The complete instruction set is explained alphabetically in Appendix F for easy reference; the appendix also includes three lists of instruction summaries arranged according to the functions, hexadecimal sequence of machine codes, and alphabetical order.

**The letters R, Rd, Rs, Rp represent generic registers. In the 8085 instructions, these are replaced by registers such as A, B, C, D, E, H, and L or register pairs.

1.10	MOV R, M	MOV B, M	Copy the data byte into register from the memory specified by the address in HL register
1.11	MOV M, R	MOV M, C	Copy the data byte from the register into memory specified by the address in HL register

2. Arithmetic Instructions. The frequently used arithmetic operations are:

- Add
- Subtract
- Increment (Add 1)
- Decrement (Subtract 1)

Mnemonics	Examples	Operation
2.1 ADD R	ADD B	Add the contents of a register to the register to the contents of A
2.2 ADI 8-bit	ADI 37H	Add 8-bit data to the contents of A
2.3 ADD M	ADD M	Add the contents of memory to A; the address of memory is in HL register
2.4 SUB R	SUB C	Subtract the contents of a register from the contents of A
2.5 SUI 8-bit	SUI 7FH	Subtract 8-bit data from the contents of A
2.6 SUB M	SUB M	Subtract the contents of memory from A; the address of memory is in HL register
2.7 INR R	INR D	Increment the contents of a register
2.8 INR M	INR M	Increment the contents of memory, the address of which is in HL
2.9 DCR R	DCR E	Decrement the contents of a register
2.10 DCR M	DCR M	Decrement the contents of a memory, the address of which is in HL
2.11 INX Rp	INX H	Increment the contents of a register pair
2.12 DCX Rp	DCX B	Decrement the contents of a register pair

3. Logic and Bit Manipulation Instructions. These instructions include the following operations:

- AND
- OR
- X-OR (Exclusive OR)
- Compare
- Rotate Bits

Mnemonics	Examples	Operation
3.1 ANA R	ANA B	Logically AND the contents of a register with the contents of A

3.2	ANI 8-bit	ANI 2FH	Logically AND 8-bit data with the contents of A
3.3	ANA M	ANA M	Logically AND the contents of memory with the contents of A; the address of memory is in HL register
3.4	ORA R	ORA E	Logically OR the contents of a register with the contents of A
3.5	ORI 8-bit	ORI 3FH	Logically OR 8-bit data with the contents of A
3.6	ORA M	ORA M	Logically OR the contents of memory with the contents of A; the address of memory is in HL register
3.7	XRA R	XRA B	Exclusive-OR the contents of a register with the contents of A
3.8	XRI 8-bit	XRI 6AH	Exclusive-OR 8-bit data with the contents of A
3.9	XRA M	XRA M	Exclusive-OR the contents of memory with the contents of A; the address of memory is in HL register
3.10	CMP R	CMP B	Compare the contents of register with the contents of A for less than, equal to, or greater than
3.11	CPI 8-bit	CPI 4FH	Compare 8-bit data with the contents of A for less than, equal to, or greater than

4. Branch Instructions. The following instructions change the program sequence.

4.1	JMP 16-bit address	JMP 2050H	Change the program sequence to the specified 16-bit address
4.2	JZ 16-bit address	JZ 2080H	Change the program sequence to the specified 16-bit address if the Zero flag is set
4.3	JNZ 16-bit address	JNZ 2070H	Change the program sequence to the specified 16-bit address if the Zero flag is reset
4.4	JC 16-bit address	JC 2025H	Change the program sequence to the specified 16-bit address if the Carry flag is set
4.5	JNC 16-bit address	JNC 2030H	Change the program sequence to the specified 16-bit address if the Carry flag is reset
4.6	CALL 16-bit address	CALL 2075H	Change the program sequence to the location of a subroutine
4.7	RET	RET	Return to the calling program after completing the subroutine sequence

5. Machine Control Instructions. These instructions affect the operation of the processor.

5.1	HLT	HLT	Stop processing and wait
5.2	NOP	NOP	Do not perform any operation

This set of instructions is a representative sample; it does not include various instructions related to 16-bit data operations, additional jump instructions, and conditional Call and Return instructions.

2.6

WRITING AND HAND ASSEMBLING A PROGRAM

In previous sections, we discussed the 8085 instructions, recognized the number of bytes per instruction, looked at the relationship between the number of bytes of an instruction and memory registers needed for storage, and examined the processor's computing capability in the overview of the instruction set. Now let us pull together all these concepts in a simple illustrative program.

2.6.1 Illustrative Program: Subtracting Two Hexadecimal Numbers and Storing the Result in Memory

PROBLEM STATEMENT

Write instructions to subtract two bytes already stored in memory registers (also referred to as memory locations or memory addresses) 2051H and 2052H. Location 2051H holds the byte 49H and location 2051H holds the byte 9FH. Subtract the first byte, 49H, from the second byte, 9FH, and store the answer in memory location 2053H. Write instructions beginning at memory location 2030H.

PROBLEM ANALYSIS

This is a problem similar to the problem in Section 2.4. However, we need to note some specific points in this problem.

1. The data bytes to be subtracted are already stored in memory registers 2051H and 2052H. We do not need to write instructions to store these bytes. You should store these bytes by using a keyboard of your trainer, or if you are using a simulator, you should observe the numbers in those memory locations when you store them.
2. The program should be written starting at memory location 2030H. This memory location is selected arbitrarily to emphasize that you can write a program beginning at any available memory location.

3. The microprocessor performs arithmetic operations in the ALU, meaning the processor must use accumulator A in performing the subtraction.
4. The data bytes must be copied from memory into the microprocessor registers (until you learn how to perform an arithmetic operation by using the accumulator and a memory register).

WRITING MNEMONICS AND ASSEMBLING HEX CODE

The flowchart for this problem is similar to that shown in Figure 2.3. The steps in writing instructions are as follows:

1. Copy two data bytes into processor registers. By examining Section 2.5, we find two instructions, LDA and STA (instructions 1.6 and 1.7), to copy a byte from memory into A and from A into memory. There are two other instructions (MOV R, M and MOV M, R) that can copy between memory and registers, but those instructions require the concept of memory pointers, which will be discussed in Chapter 7. Now let us take a look at the instruction LDA in the 8085 Instruction Summary: Alphabetical Order, inside the back cover.
 - a. **The instruction is: LDA 16-bit.** LDA is the opcode with Hex code 3A. (For a complete description of the instruction LDA, see Appendix F—all instructions are explained in alphabetical order.) As we discussed in Section 2.4 (“Recognizing the Number of Bytes in an Instruction”), this must be a 3-byte instruction and will require three memory locations.
 - b. **The operand is a 16-bit address of the memory location from which we want to copy the byte into A.** First we want to copy a byte from memory location 2051H.
 - c. **The 3-byte code is 3A 51 20.** See Section 2.3. A 16-bit address is always written in reverse order—low-order byte followed by high-order byte. Our program begins at location 2030H; therefore, these 3 bytes will be stored in locations with Hex addresses 2030, 31, and 32. And the instruction, when executed, will copy the first byte, 49H, into A.
 - d. **Copying the second byte into A.** If we copy the second byte, 9FH, from memory location 2052H into A, we will destroy the first byte, 49H. Therefore, the previous byte, 49H, should be stored first in some other register such as B.
2. As discussed in 1d above, the second instruction should be to copy the byte from A into B. The instruction is: MOV Rd, Rs (instruction 1.2 in Section 2.5), copying from one source register Rs to another destination register Rd. The terms Rs and Rd are generic; they represent any register A through L. The instruction is:
MOV B, A with Hex code 47. Copy A into B—note the reversed order of A and B. This is a 1-byte instruction.
3. Now we can copy the second byte (9FH) from memory location 2052H into A by using the instruction
LDA 2052H with the 3-byte Hex code 3A 52 20.
4. The next step is to subtract B from A. If we look in Section 2.5 under Arithmetic Instructions, we find three instructions: 2.4, SUB R; 2.5, SUI 8-bit; and 2.6, SUB M.

The instruction is:

SUB B with Hex code 90. (Look at the inside of the back cover to find the code.) This instruction subtracts B from A and saves the result in A.

5. The result in A should be stored in memory location 2053H. The instruction is **STA 2053 with Hex code 32 53 20.**
6. Each program must be terminated; otherwise, the processor continues to fetch and execute instructions from the remaining memory registers until it gets lost or caught up in an infinite loop. This step may appear trivial, but it is essential. Therefore, the last instruction is
HALT with Hex code 76. In instructional trainers, the Restart (RST) instruction is used to pass the control of running programs back to the monitor program of the trainer.
7. Now we need to load the two data bytes 49H and 9FH in memory locations 2051H and 2052H. This step is a manual entry of the data bytes, independent of the program.
8. So far we have completed two steps: writing the program (as shown in Column 1 in Table 2.2) and entering the Hex code in the memory registers of a memory chip (as shown in Column 2 in Table 2.2). If you observe this program in a simulator (see Appendix H), it should appear as shown in Column 2. Using the analogy of putting a radio kit together, we now have a page of instructions. Now we need to find the page, begin to read, understand the instructions, and perform the task until the kit is

TABLE 2.2
Illustrative Program: Assembly

Column 1	Column 2		Column 3
Instructions	Memory Addresses	Hex Code	Comments
LDA 2051H	2030	3A	Copy the first byte, 49H, from memory location 2051H into A
	2031	51	
	2032	20	
MOV B, A	2033	47	Save the first byte in B
LDA 2052H	2034	3A	Copy the second byte, 9FH, from memory location 2052H into A
	2035	52	
	2036	20	
SUB B	2037	90	Subtract 49H from 9FH and save the result in A
STA 2053H	2038	32	Save the result in memory location 2053H
	2039	53	
	203A	20	
	203B	76/FF	End of the program
HLT or RST7	2051	49	
	2052	9F	These data bytes must be manually loaded; they are not part of writing the program
	2053	00	

built or we take a break. Entering Hex code in memory is similar to the instruction page of a radio kit. Now we need to tell the processor where our instructions begin and ask it to execute those instructions.

9. Our program begins at location 2030H. We need to let the processor know the starting location through a keyboard or simulation.
10. Now we have two choices: let the processor execute one instruction at a time, called Single Step, or execute (run) the entire program. The Single Step execution is easily observed in a simulator (see Appendix H).
11. Initially, it is strongly advisable to follow the Single Step execution. As you step through each instruction, you can observe changes in the contents of registers. When you execute SUB B, the accumulator should have 56H (9FH–49H), and when you execute STA 2053H, the contents of memory location 2053H should change from 00 to 56H.

COMMON ERRORS

When we learn a new language, we make errors such as misspelling and mispronouncing words. Similar errors happen in writing assembly language instructions. But the processor demands the exact syntax. A list of common errors is as follows, with examples from our problem:

1. LDA 2051H: Not entering the code of the 16-bit address in reversed order.
2. Forgetting to enter the code for the operand, such as 2051H.
3. MOV B, A: Assuming that this copies from B to A.
4. Incrementing the address in decimal, from 2039H to 2040H.
5. HLT: Not terminating a program.
6. Confusing the entering of Hex code in memory as executing a program.

SUMMARY

This chapter described the data manipulation functions of the 8085 microprocessor, provided an overview of the instruction set, and illustrated the execution of instructions in relation to the system's clock. The important concepts in this chapter can be summarized as follows.

- The 8085 microprocessor operations are classified into five major groups: data transfer (copy), arithmetic, logic, branch, and machine control.
- An instruction has two parts: opcode (operation to be performed) and operand (data to be operated on). The operand can be data (8- or 16-bit), address, or register, or it can be implicit. The method of specifying an operand (directly, indirectly, etc.) is called the addressing mode.
- The instruction set is classified in three groups according to the word size: 1-, 2-, or 3-byte instructions.
- To write an assembly language program, divide the given problem into small steps in terms of the microprocessor operations, translate these steps into assembly language instructions, and then translate them into the 8085 machine code.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

1. List the four categories of 8085 instructions that manipulate data.
2. Define opcode and operand, and specify the opcode and the operand in the instruction MOV H,L.
3. Write the machine code for the instruction MOV H,A if the opcode = 01₂, the register code for H = 100₂, and the register code for A = 111₂.
4. Find the machine codes and the number of bytes of the following instructions. Identify the opcodes and the operands. (Refer to the instruction set on the inside back cover.)
 - a. MVI H,47H
 - b. ADI F5H
 - c. SUB C
5. Find the Hex codes for the following instructions, identify the opcodes and operands, and show the order of entering the codes in memory.
 - a. STA 2050H
 - b. JNZ 2070H
6. Find the Hex machine code for the following instructions from the instruction set listed on the back cover, and identify the number of bytes of each instruction.

MVI B,4FH	;Load the first byte
MVI C,78H	;Load the second byte
MOV A,C	;Get ready for addition
ADD B	;Add two bytes
OUT 07H	;Display the result at port 7
HLT	;End of program

7. If the starting address of the system memory is 2000H, and you were to enter the Hex code for the instructions in Question 6, identify the memory addresses and their corresponding Hex codes.
8. Assemble the following program, starting with the memory address 2020H.

MVI A,8FH	;Load the first byte
MVI B,68H	;Load the second byte
SUB B	;Subtract the second byte
ANI 0FH	;Eliminate D ₇ -D ₄
STA 2070H	;Store D ₃ -D ₀ in memory location 2070H
HLT	;End of program

9. Assemble the following program, starting at location 2000H.

START: IN F2H	;Read input switches at port F2H
CMA	;Set ON switches to logic 1
ORA A	;Set Z flag if no switch is ON
JZ START	;Go back and read input port if all ; switches are off

10. Write logical steps to add the following two Hex numbers. Both the numbers should be saved for future use. Save the sum in the accumulator.

Numbers: A2H and 18H

11. Translate the program in Question 10 into the 8085 assembly language.
12. Data byte 28H is stored in register B and data byte 97H is stored in the accumulator. Show the contents of registers B, C, and the accumulator after the execution of the following two instructions:

MOV A,B
MOV C,A

13. In Question 6, explain the potential results of the program if the code 07H of the OUT instruction is omitted.
14. In Question 8, explain possible outcomes if the second byte 0FH of the instruction ANI 0FH is omitted.
15. Given the following three sets of Hex codes, identify the mnemonics:

(a)	(b)	(c)
3E	06	06
F2	82	4F
32	78	0E
32	32	37
20	50	78
76	20	81
	FF	00
		32
		35
		20
		76

16. Identify and explain the results of Question 15 (a) and (b).
17. In Question 15 (c), what does the code 00 represent: data, low-order address, or an opcode?
18. In Question 15 (c), explain what the program does, calculate the sum, and identify the memory location where the sum is stored.

3

Microprocessor Architecture and Microcomputer Systems

A microcomputer system consists of four components—the microprocessor, memory, and I/O (input/output)—as discussed in Chapter 1. The microprocessor manipulates data, controls the timing of various operations, and communicates with such peripherals (devices) as memory and I/O. The internal logic design of the microprocessor, called its **architecture**, determines how and when various operations are performed by the microprocessor. The system bus provides paths for the flow of binary information (data and instructions).

This chapter expands on the bus concept discussed in Chapter 1 and shows how binary information flows externally among the components of the system. The chapter deals with the internal architecture and various operations of the microprocessor in the context of the 8085. It also expands

on topics such as memory and I/O, and reviews interfacing devices, such as buffers, decoders, and latches.

OBJECTIVES

- List the four operations commonly performed by the microprocessor or the microprocessing unit (MPU).
- Define the address bus, the data bus, and the control bus, and explain their functions in reference to the 8085 microprocessor.
- Explain the functions Reset, Interrupt, Wait, and Hold.
- Explain memory organization and memory map, and explain how memory addresses are assigned to a memory chip.

- List the types of memory and their functions.
- Explain the difference between the peripheral I/O (also known as I/O-mapped I/O) and the memory-mapped I/O.
- Describe the steps in executing an instruction in a bus-oriented system.
- Define tri-state logic and explain the functions of such MSI devices as buffers, decoders, encoders, and latches.

3.1

MICROPROCESSOR ARCHITECTURE AND ITS OPERATIONS

The microprocessor is a programmable digital device, designed with registers, flip-flops, and timing elements. The microprocessor has a set of instructions, designed internally, to manipulate data and communicate with peripherals. This process of data manipulation and communication is determined by the logic design of the microprocessor, called the architecture.

The microprocessor can be programmed to perform functions on given data by selecting necessary instructions from its set. These instructions are given to the microprocessor by writing them into its memory. Writing (or entering) instructions and data is done through an input device such as a keyboard. The microprocessor reads or transfers one instruction at a time, matches it with its instruction set, and performs the data manipulation indicated by the instruction. The result can be stored in memory or sent to such output devices as LEDs or a CRT terminal. In addition, the microprocessor can respond to external signals. It can be interrupted, reset, or asked to wait to synchronize with slower peripherals. All the various functions performed by the microprocessor can be classified in three general categories:

- Microprocessor-initiated operations
- Internal operations
- Peripheral (or externally initiated) operations

To perform these functions, the microprocessor requires a group of logic circuits and a set of signals called control signals. However, early processors did not have the necessary circuitry on one chip; the complete units were made up of more than one chip. Therefore, the term *microprocessing unit (MPU)* is defined here as a group of devices that can perform these functions with the necessary set of control signals. This term is similar to the term *central processing unit (CPU)*. However, later microprocessors include most of the necessary circuitry to perform these operations on a single chip. Therefore, the terms *MPU* and *microprocessor* often are used synonymously.

The microprocessor functions listed above are explained here in relation to the 8085 MPU but without the details of the MPUs. However, the general concepts discussed here are applicable to any microprocessor. The devices necessary to make up the 8085 MPUs will be discussed in the next chapter.

3.1.1 Microprocessor-Initiated Operations and 8085 Bus Organization

The MPU performs primarily four operations:^{*}

1. Memory Read: Reads data (or instructions) from memory.
2. Memory Write: Writes data (or instructions) into memory.
3. I/O Read: Accepts data from input devices.
4. I/O Write: Sends data to output devices.

All these operations are part of the communication process between the MPU and peripheral devices (including memory). To communicate with a peripheral (or a memory location), the MPU needs to perform the following steps:

Step 1: Identify the peripheral or the memory location (with its address).

Step 2: Transfer binary information (data and instructions).

Step 3: Provide timing or synchronization signals.

The 8085 MPU performs these functions using three sets of communication lines called buses: the address bus, the data bus, and the control bus (Figure 3.1). In Chapter 1, these buses are shown as one group, called the system bus.

ADDRESS BUS

The address bus is a group of 16 lines generally identified as A₀ to A₁₅. The address bus is unidirectional: bits flow in one direction—from the MPU to peripheral devices. The MPU uses the address bus to perform the first function: identifying a peripheral or a memory location (Step 1).

In a computer system, each peripheral or memory location is identified by a binary number, called an address, and the address bus is used to carry a 16-bit address. This is sim-

^{*}Other operations are omitted here for clarity and discussed in the next chapter.

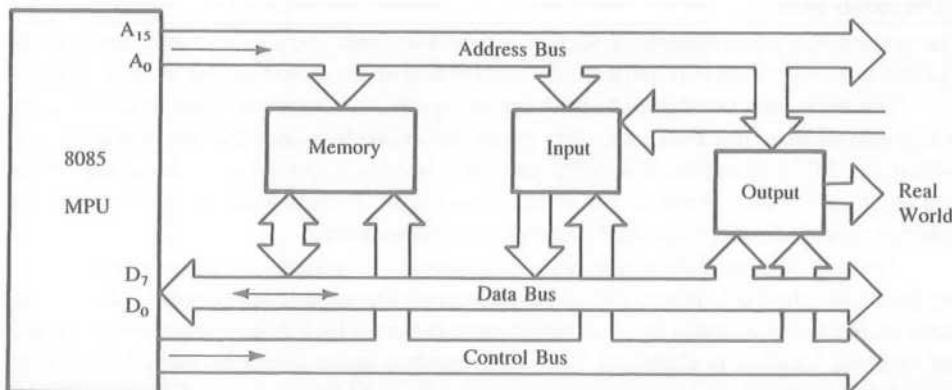


FIGURE 3.1
The 8085 Bus Structure

ilar to the postal address of a house. A house can be identified by various number schemes. For example, the forty-fifth house in a lane can be identified by the two-digit number 45 or by the four-digit number 0045. The two-digit numbering scheme can identify only a hundred houses, from 00 to 99. On the other hand, the four-digit scheme can identify ten thousand houses, from 0000 to 9999. Similarly, the number of address lines of the MPU determines its capacity to identify different memory locations (or peripherals). The 8085 MPU with its 16 address lines is capable of addressing $2^{16} = 65,536$ (generally known as 64K) memory locations. As explained in Chapter 1, 1K memory is determined by rounding off 1024 to the nearest thousand; similarly, 65,536 is rounded off to 64,000 as a multiple of 1K.

Most 8-bit microprocessors have 16 address lines. This may explain why microcomputer systems based on 8-bit microprocessors have 64K memory. However, not every microcomputer system has 64K memory. In fact, most single-board microcomputers have less than 4K of memory, even if the MPU is capable of addressing 64K memory. The number of address lines is arbitrary; it is determined by the designer of a microprocessor based on such considerations as availability of pins and intended applications of the processor. For example, the Intel 8088 processor has 20 and the Pentium processor has 32 address lines.

DATA BUS

The data bus is a group of eight lines used for data flow (Figure 3.1).^{*} These lines are **bidirectional**—data flow in both directions between the MPU and memory and peripheral devices. The MPU uses the data bus to perform the second function: transferring binary information (Step 2).

The eight data lines enable the MPU to manipulate 8-bit data ranging from 00 to FF ($2^8 = 256$ numbers). The largest number that can appear on the data bus is 11111111 (255₁₀). The 8085 is known as an 8-bit microprocessor. Microprocessors such as the Intel 8086, Zilog Z8000, and Motorola 68000 have 16 data lines; thus they are known as 16-bit microprocessors. The Intel 80386/486 have 32 data lines; thus they are classified as 32-bit microprocessors.

CONTROL BUS

The control bus is comprised of various single lines that carry synchronization signals. The MPU uses such lines to perform the third function: providing timing signals (Step 3).

The term **bus**, in relation to the control signals, is somewhat confusing. These are not groups of lines like address or data buses, but individual lines that provide a pulse to indicate an MPU operation. The MPU generates specific control signals for every operation (such as Memory Read or I/O Write) it performs. These signals are used to identify a device type with which the MPU intends to communicate.

To communicate with a memory—for example, to read an instruction from a memory location—the MPU places the 16-bit address on the address bus (Figure 3.2). The address on the bus is decoded by an external logic circuit, which will be explained later, and the memory location is identified. The MPU sends a pulse called Memory Read as the control signal. The pulse activates the memory chip, and the contents of the memory location (8-bit data) are placed on the data bus and brought inside the microprocessor.

^{*}The term *data* refers to any binary information that may include an instruction, an address, or a number.

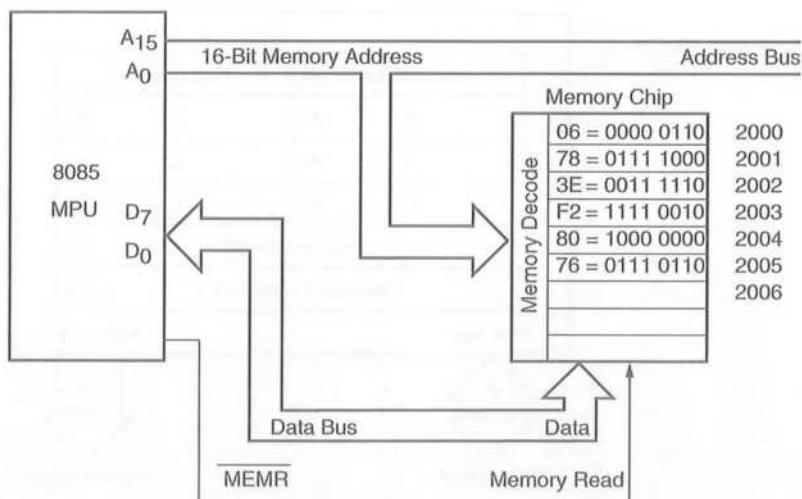


FIGURE 3.2
Memory Read Operation

What happens to the data byte brought into the MPU depends on the internal architecture of the microprocessor, which we will describe in the next section.

3.1.2 Internal Data Operations and the 8085 Registers

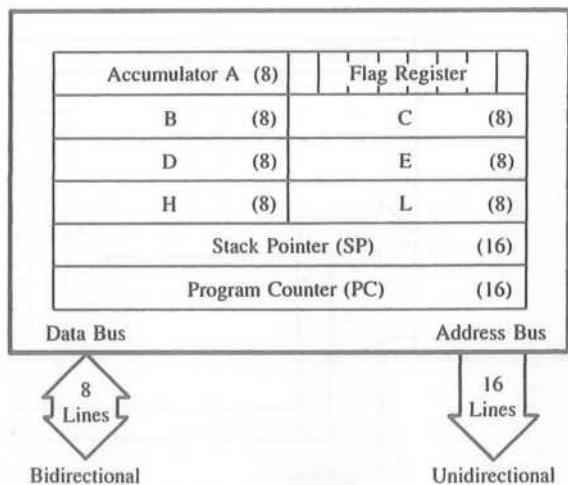
The internal architecture of the 8085 microprocessor determines how and what operations can be performed with the data. These operations are:

1. Store 8-bit data.
2. Perform arithmetic and logical operations.
3. Test for conditions.
4. Sequence the execution of instructions.
5. Store data temporarily during execution in the defined R/W memory locations called the stack.

To perform these operations, the microprocessor requires registers, an arithmetic/logic unit (ALU) and control logic, and internal buses (paths for information flow). Figure 3.3 (same as Figure 2.1(b); it is repeated here for reference) shows the programming model of the 8085 displaying the internal registers and the accumulator. The functions of these registers are described in reference to the five operations when the processor executes the following three instructions. The Hex codes of these instructions are stored in memory locations from 2000H to 2005H as shown in Figure 3.2.

2000	06	MVI B, 76H
2001	78	
2002	3E	MVI A, F2H
2003	F2	
2004	80	ADD B
2005	76	HLT

FIGURE 3.3
The 8085 Programmable Registers



When the user enters the memory address 2000H and pushes the execute key of the trainer, the processor places the address 2000H in the program counter (PC).

1. The program counter is a 16-bit register that performs the fourth operation in the list: sequencing the execution of the instructions. When the processor begins execution, it places the address 2000H on the address bus and increments the address in the PC to 2001 for the next operation. It brings the code 06, interprets the code, places the address 2001H on the address bus, and then gets byte 78H and increments the address in PC to 2002H. The processor repeats the same process for the next instruction, MVI A, F2H.
2. When the processor executes the first two instructions, it uses register B to store 78H and A to store F2H in binary (Operation 1).
3. When the processor executes the instruction ADD B in the ALU (Operation 2), it adds 78H to F2H, resulting in the sum 16AH ($78H + F2H = 16AH$). It replaces F2H by 6AH in A and sets the Carry flag as described next.
4. In our example, the addition operation generates a carry because the sum is larger than the size of the accumulator (8 bits). To indicate the carry, the processor sets the flip-flop called Carry (CY flag) to 1 and places logic 1 in the flag register at the designated bit position for the carry.
5. The fifth operation deals with the concept of the stack. The stack pointer is a 16-bit register used as a memory pointer to identify the stack, part of the R/W memory defined and used by the processor for temporary storage of data during the execution. This is fully described in Chapter 9.

3.1.3 Peripheral or Externally Initiated Operations

External devices (or signals) can initiate the following operations, for which individual pins on the microprocessor chip are assigned: Reset, Interrupt, Ready, Hold.

- Reset: When the reset pin is activated by an external key (also called a reset key), all internal operations are suspended and the program counter is cleared (it holds 0000H). Now the program execution can again begin at the zero memory address.
- Interrupt: The microprocessor can be interrupted from the normal execution of instructions and asked to execute some other instructions called a service routine (for example, emergency procedures). The microprocessor resumes its operation after completing the service routine (see Chapter 12).
- Ready: The 8085 has a pin called READY. If the signal at this READY pin is low, the microprocessor enters into a Wait state. This signal is used primarily to synchronize slower peripherals with the microprocessor.
- Hold: When the HOLD pin is activated by an external signal, the microprocessor relinquishes control of buses and allows the external peripheral to use them. For example, the HOLD signal is used in Direct Memory Access (DMA) data transfer (see Chapter 15).

These operations are listed here to provide an overview of the capabilities of the 8085. They will be discussed in Part III.

MEMORY

3.2

Memory is an essential component of a microcomputer system; it stores binary instructions and data for the microprocessor. There are various types of memory, which can be classified in two groups: prime (or main) memory and storage memory. In the last chapter, we discussed briefly two examples of prime memory: Read/Write memory (R/WM) and Read-Only memory (ROM). Magnetic tapes or disks can be cited as examples of storage memory. First, we will focus on prime memory and then, briefly discuss storage memory when we examine various types of memory.

The R/W memory is made of registers, and each register has a group of flip-flops or field-effect transistors that store bits of information; these flip-flops are called memory cells. The number of bits stored in a register is called a memory word; memory devices (chips) are available in various word sizes. The user can use this memory to hold programs and store data. On the other hand, the ROM stores information permanently in the form of diodes; the group of diodes can be viewed as a register. In a memory chip, all registers are arranged in a sequence and identified by binary numbers called memory addresses. To communicate with memory, the MPU should be able to

- select the chip,
- identify the register, and
- read from or write into the register.

The MPU uses its address bus to send the address of a memory register and uses the data bus and control lines to read from (as shown in Figure 3.2) or write into that register. In the following sections, we will examine the basic concepts related to memory: its

structure, its addressing, and its requirements to communicate with the MPU and build a model for R/W memory. However, except for slight differences in Read/Write control signals, the discussion is equally applicable to ROM.

3.2.1 Flip-Flop or Latch as a Storage Element

What is memory? It is a circuit that can store bits—high or low, generally voltage levels or capacitive charges representing 1 and 0. A flip-flop or a latch* is a basic element of memory. To write or store a bit in the latch, we need an input data bit (D_{IN}) and an enable signal (EN), as shown in Figure 3.4(a). In this latch, the stored bit is always available on the output line D_{OUT} . To avoid unintentional change in the input and control the availability of the output, we can use two tri-state* buffers on the latch, as shown in Figure 3.4(b). Now we can write into the latch by enabling the input buffer and read from it by enabling the output buffer. Figure 3.4(b) shows the Write signal as \overline{WR} and the Read signal as \overline{RD} ; these are active low signals indicated by the bar. This latch, which can store one binary bit, is called a memory cell. Figure 3.5(a) shows four such cells or latches grouped together; this is a register, which has four input lines and four output lines and can store four bits; thus the size of the memory word is four bits. The size of this register is specified either as 4-bit or 1×4 -bit, which indicates one register with four cells or four I/O lines. Figures 3.5(b) and (c) show simplified block diagrams of the 4-bit register.

In Figure 3.6, four registers with eight cells (or an 8-bit memory word) are arranged in a sequence. To write into or read from any one of the registers, a specific register should be identified or enabled. This is a simple decoding function; a 2-to-4 decoder can perform that function. However, two more input lines A_1 and A_0 , called address lines, are required to the decoder. These two input lines can have four different bit combinations (00, 01, 10, 11), and each combination can identify or enable one of the registers named as Register 0 through Register 3. Thus the Enable signal of the flip-flops in Figure 3.5 is replaced by two address lines in Figure 3.6. Figure 3.6(a) has 8-bit registers and Figure

*If you are not familiar with these devices, review Section 3.5; flip-flops (latches), tri-state buffers, and decoders are discussed briefly in Section 3.5.

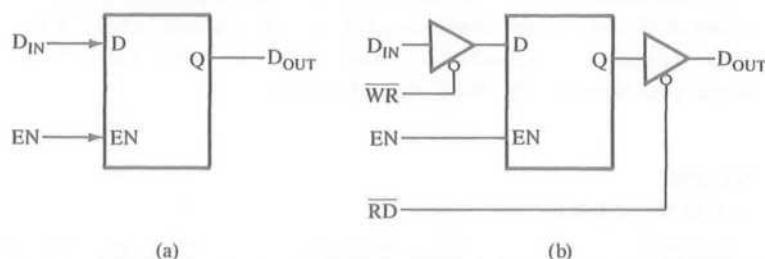


FIGURE 3.4
Latches as Storage Element: Basic Latch (a) and Latch with Two Tri-State Buffers (b)

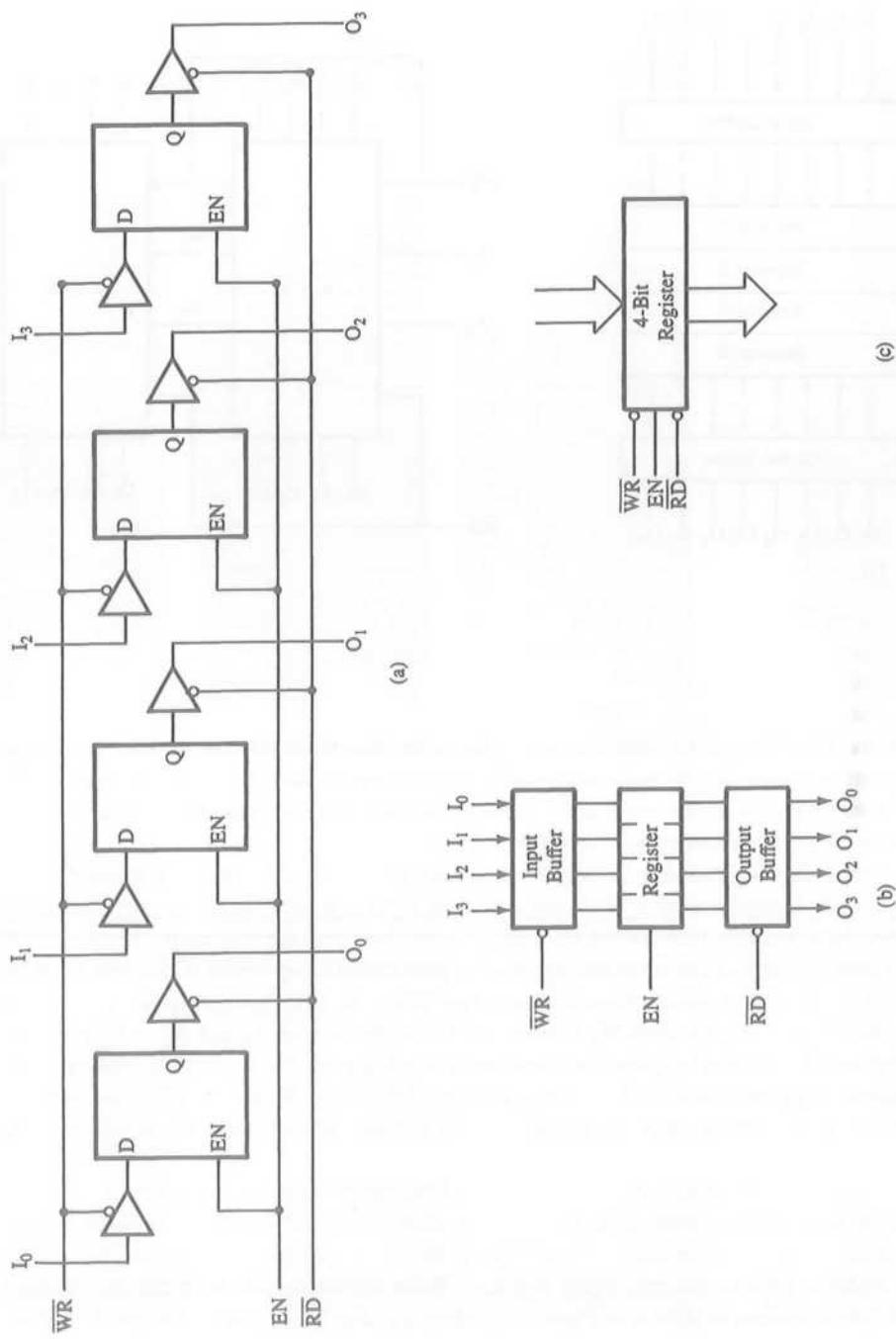


FIGURE 3.5
Four Latches as a 4-Bit Register (a) and Block Diagrams of a 4-Bit Register (b and c)

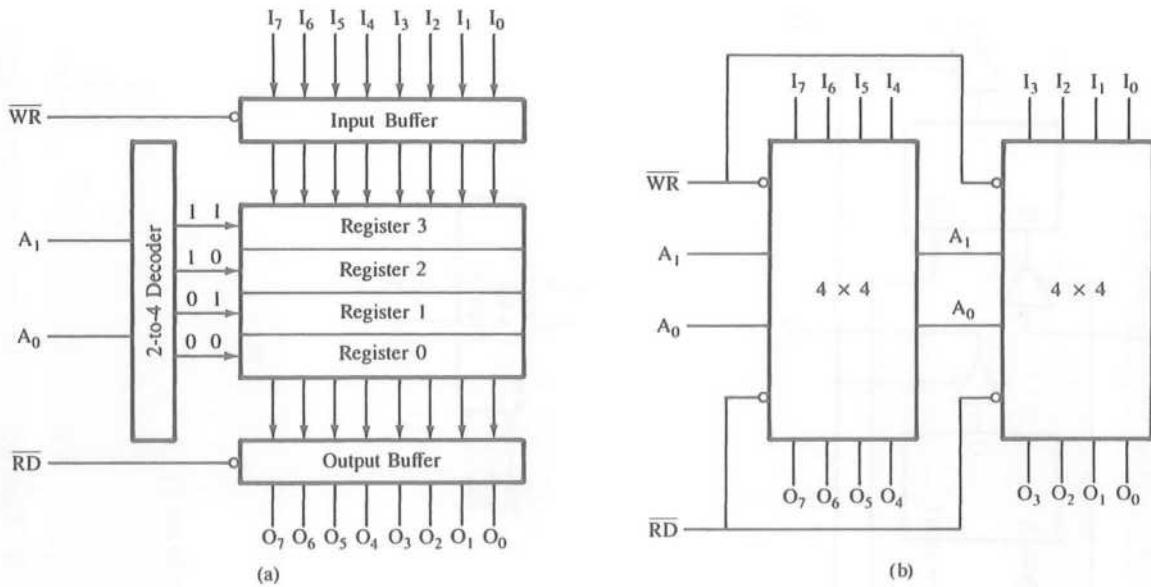


FIGURE 3.6
4 × 8-Bit Register

3.6(b) has two chips with 4-bit registers. This is an illustration of how smaller word size chips can be connected to make up an 8-bit word memory size. Now we can expand the number of registers. If we have eight registers on one chip, we need three address lines, and if we have 16 registers, we need four address lines.

An interesting problem is how to deal with more than one chip; for example, two chips with four registers each. We have a total of eight registers; therefore, we need three address lines, but one line should be used to select between the two chips. Figure 3.7(b) shows two memory chips, with an additional signal called Chip Select (\overline{CS}), and A_2 (with an inverter) is used to select between the chips. When A_2 is 0 (low), chip M_1 is selected, and when A_2 is 1 (high), chip M_2 is selected. The addresses on A_1 and A_0 will determine the registers to be selected; thus, by combining the logic on A_2 , A_1 , and A_0 , the memory addresses range from 000 to 111. The concept of the Chip Select signal gives us more flexibility in designing chips and allows us to expand memory size by using multiple chips.

Now let us examine the problem from a different perspective. Assume that we have available four address lines and two memory chips with four registers each as before. Four address lines are capable of identifying 16 (2^4) registers; however, we need only three address lines to identify eight registers. What should we do with the fourth line? One of the solutions is shown in Figure 3.8. Memory chip M_1 is selected when A_3 and A_2 are both 0; therefore, registers in this chip are identified with the addresses ranging from 0000 to 0011 (0 to 3). Similarly, the addresses of memory chip M_2 range from 1000 to

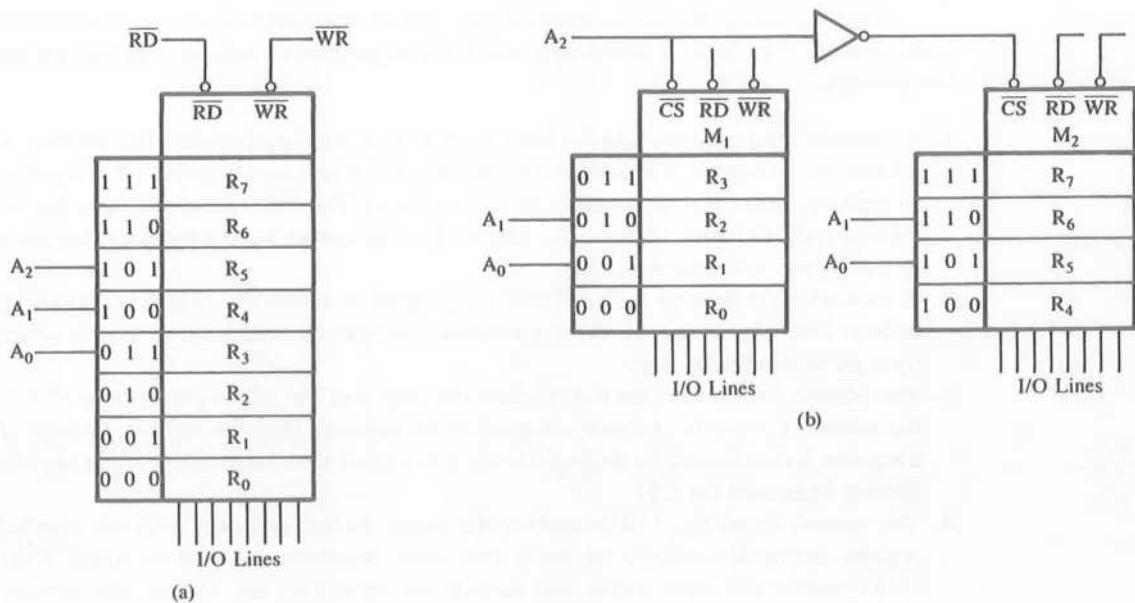


FIGURE 3.7
Two Memory Chips with Four Registers Each and Chip Select

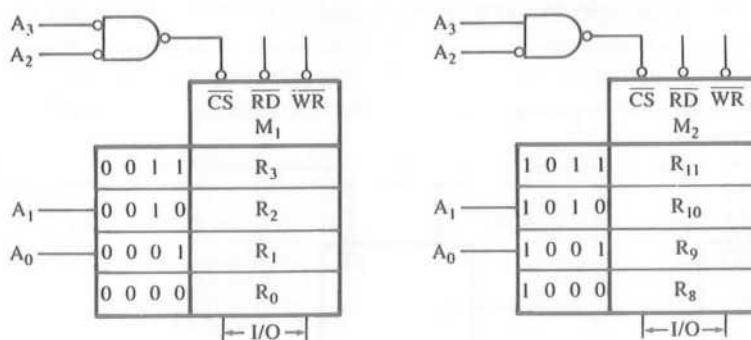


FIGURE 3.8
Addressing Eight Registers with Four Address Lines

1011 (8 to B); this chip is selected only when A_3 is 1 and A_2 is 0. In this example, we need three lines to identify eight registers: two for registers and one for Chip Select. However, we used the fourth line for Chip Select also. This is called complete or absolute decoding. Another option is to leave the fourth line as don't care; we will further explore this concept later.

After reviewing the above explanation, we can summarize the requirements of a memory chip, then build a model and match the requirements with the microprocessor bus concepts:

1. A memory chip requires address lines to identify a memory register. The number of address lines required is determined by the number of registers in a chip (2^n = Number of registers where n is the number of address lines). The 8085 microprocessor has 16 address lines. Of these 16 lines, the address lines necessary for the memory chip must be connected to the memory chip.
2. A memory chip requires a Chip Select (\overline{CS}) signal to enable the chip. The remaining address lines (from Step 1) of the microprocessor can be connected to the \overline{CS} signal through an interfacing logic.
3. The address lines connected to \overline{CS} select the chip, and the address lines connected to the address lines of the memory chip select the register. Thus the memory address of a register is determined by the logic levels (0/1) of all the address lines (including the address lines used for \overline{CS}).
4. The control signal Read (RD) enables the output buffer, and data from the selected register are made available on the output lines. Similarly, the control signal Write (WR) enables the input buffer, and data on the input lines are written into memory cells. The microprocessor can use its Memory Read and Memory Write control signals to enable the buffers and the data bus to transport the contents of the selected register between the microprocessor and memory.

A model of a typical memory chip representing the above requirements is shown in Figure 3.9. Figure 3.9(a) represents the R/W memory and Figure 3.9(b) represents the Read-Only memory; the only difference between the two as far as addressing is concerned is that ROM does not need the \overline{WR} signal. Internally, the memory cells are arranged in a matrix format—in rows and columns; as the size increases, the internal de-

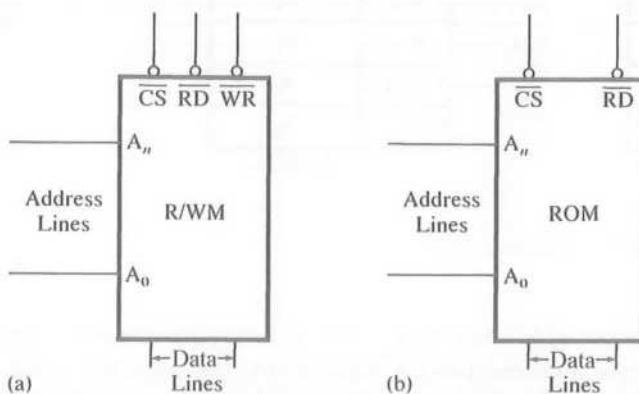


FIGURE 3.9
R/W Memory Model (a) and ROM Model (b)

coding scheme we discussed becomes impractical. For example, a memory chip with 1024 registers would require a 10-to-1024 decoder. If the cells are arranged in six rows and four columns, the internal decoding circuitry can be designed with two decoders, one for selecting a row and the other for selecting a column. However, we will not be concerned about the internal row and column arrangement because it does not affect our external interfacing logic, which is explained in the next section.

3.2.2 Memory Map and Addresses

Typically, in an 8-bit microprocessor system, 16 address lines are available for memory. This means it is a numbering system of 16 binary bits and is capable of identifying 2^{16} (65,536) memory registers, each register with a 16-bit address. The entire memory addresses can range from 0000 to FFFF in Hex. A memory map is a pictorial representation in which memory devices are located in the entire range of addresses. Memory addresses provide the locations of various memory devices in the system, and the interfacing logic defines the range of memory addresses for each memory device. The concept of memory map and memory addresses can be illustrated with an analogy of identical houses built in sequence and their postal addresses, or numbers.

Let us assume that houses are given three-digit decimal numbers, which will enable us to number one thousand houses from 000 to 999. Because it is cumbersome to direct someone to houses with large numbers, the numbering scheme can be devised with the concept of a row or block. Each block will have a hundred houses to be numbered with the last two digits from 00 to 99. Similarly, the blocks are also identified by the first decimal digit. For example, a house with the number 247 is house number 47 in block 2. With this scheme, all the houses in block 0 will be identified from 000 to 099, in block 2 from 200 to 299, and in block 9 from 900 to 999. This numbering scheme with three decimal digits is capable of giving addresses to one thousand houses from 000 to 999 (10 blocks of 100 houses each). Let us also assume that all houses are identical and have eight rooms.

The example of numbering the houses is directly applicable to assigning addresses to memory registers. In the binary number system, 16 binary digits can have 65,536 (2^{16}) different combinations. In the hexadecimal number system, 16 binary bits are equivalent to four Hex digits that can be used to assign addresses to 65,536 (0000H to FFFFH) memory registers in various memory chips. In our analogy, a memory chip is similar to a block in a housing development and a register can be viewed as a house with eight identical rooms.

Let us assume that we have a memory chip with 256 registers. Therefore, we need only 256 numbers (out of 65,536) that require eight address lines ($2^8 = 256$). Now the question is what we should do with the remaining eight address lines of the microprocessor. We can find a clue in our housing analogy. Let us assume that we have only 100 houses in block five. They will be numbered as 500 to 599; the first digit 5 remains constant and the next two digits vary from 00 to 99. Similarly, we can use the remaining eight address lines to assign fixed logic to generate a constant (fixed) number. This can be accomplished by using the remaining eight lines for the Chip Select through appropriate logic gates, as shown in Example 3.1.



As mentioned previously, in computer systems, we define 1024 as 1K; therefore a 1K-byte memory chip has 1024 registers with 8 bits each. Similarly, a group of 256 registers is defined as one *page* and each register is viewed as a *line* to write on. This is analogous to a notebook containing various pages, with each page having a certain number of lines. With this analogy, we can view 1K-byte memory as a chip with four pages ($1024/256 = 4$) with each page having 256 registers. With two Hex digits, 256 registers can be numbered from 00 to FFH; 1024 registers can be numbered with four digits from 0000 to 03FF. If we examine the high-order digits of 1K-byte memory, we find that they range from 00 to 03 representing four pages (00, 01, 02, and 03). In 8-bit microprocessor systems, this page concept is used frequently. In 16- and 32-bit microprocessor systems, the page concept (256 registers) defined here is not applicable; it is defined differently, based on the microprocessor used in a system.

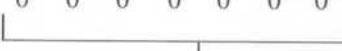
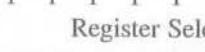
So far we have been using the term *addresses* or *address range* for a given memory chip. The term *memory map* is used generally for the entire address ranges of the memory chips in a given system. The relationship of a row of houses to the road map is similar to the relationship of memory addresses to the memory map. However, these terms are also used synonymously.

Example 3.1

Illustrate the memory address range of the chip with 256 bytes of memory, shown in Figure 3.10(a), and explain how the range can be changed by modifying the hardware of the Chip Select CS line in Figure 3.10(b).

Solution

Figure 3.10(a) shows a memory chip with 256 registers with eight I/O lines; the memory size of the chip is expressed as 256×8 . It has eight address lines (A_7-A_0), one Chip Select signal (CS) (active low), and two control signals Read (RD) and Write (WR). The eight address lines (A_7-A_0) of the microprocessor are required to identify 256 memory registers. The remaining eight lines ($A_{15}-A_8$) are connected to the Chip Select (CS) line through inverters and the NAND gate. The memory chip is enabled or selected when CS goes low. Therefore, to select the chip, the address lines $A_{15}-A_8$ should be at logic 0, which will cause the output of the NAND gate to go low. No other logic levels on the lines $A_{15}-A_8$ can select the chip. Once the chip is selected (enabled), the remaining address lines A_7-A_0 can assume any combination from 00H to FFH and identify any of the 256 memory registers through the decoder. Therefore, the memory addresses of the chip in Figure 3.10(a) will range from 0000H to 00FFH, as shown below.

$A_{15} \ A_{14} \ A_{13} \ A_{12} \ A_{11} \ A_{10} \ A_9 \ A_8$	$A_7 \ A_6 \ A_5 \ A_4 \ A_3 \ A_2 \ A_1 \ A_0$	$= 0000H$
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
		$= 00FFH$

The address lines $A_{15}-A_8$, which are used to select the chip, must have fixed logic levels, and these lines are called high-order address lines. The address lines A_7-A_0 , which

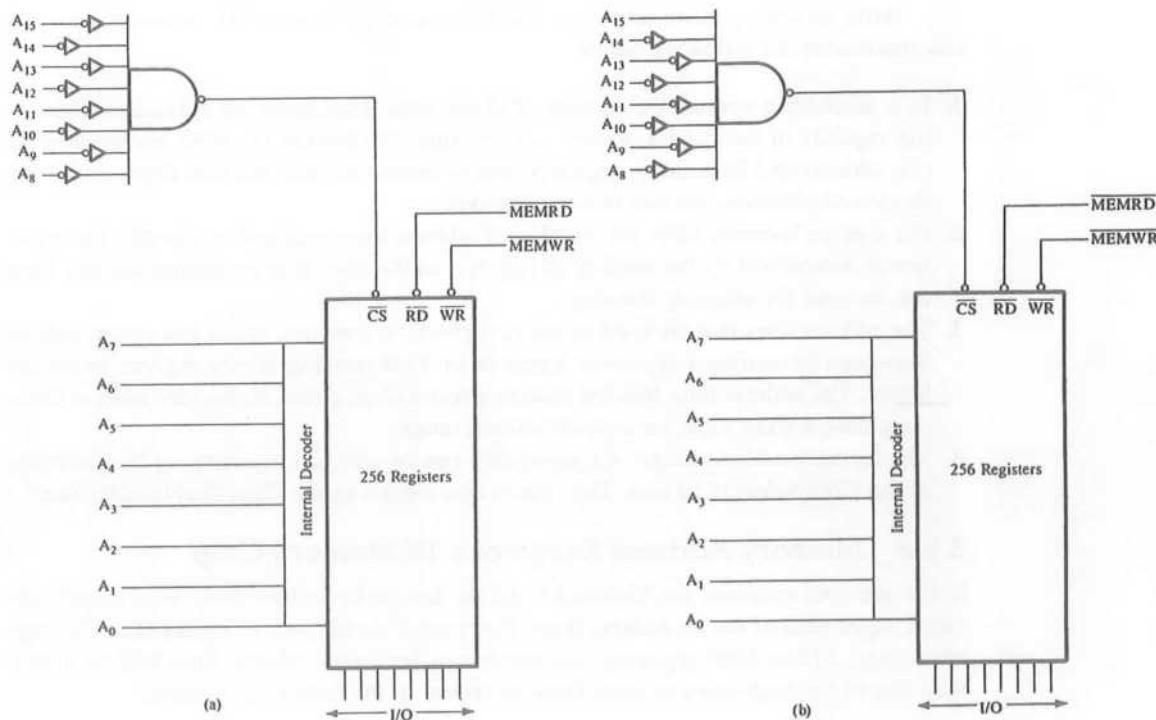


FIGURE 3.10
Memory Maps: 256 Bytes of Memory

are used to select a register, are called low-order address lines, and they can be assigned logic levels from all 0s to all 1s and any in-between combination. For example, when the address lines A₇–A₀ are all 0s, the register number 0 is selected, and when they are all 1s, the register number 255 (FFH) is selected. The Chip Select addresses are determined by the hardware (the inverters and NAND gate); therefore, the memory addresses of the chip can be changed by modifying the hardware. For example, if the inverter on line A₁₅ is removed, as shown in Figure 3.10(b), the address required on A₁₅–A₈ to enable the chip will be as follows:

$$\begin{array}{cccccccc} A_{15} & A_{14} & A_{13} & A_{12} & A_{11} & A_{10} & A_9 & A_8 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} = 80H$$

The memory address range in Figure 3.10(b) will be 8000H to 80FFH.

The memory chips in Figures 3.10(a) and (b) are the same chips. However, by changing the hardware of the Chip Select logic, the location of the memory in the map can be changed, and memory can be assigned addresses in various locations over the entire map of 0000 to FFFFH.

After reviewing the example and the previous explanation of the memory map, we can summarize the following points.

1. In a numbering system, the number of digits used determines the maximum addressing capacity of the system. Sixteen address lines (16 bits) of the 8085 microprocessor can address 65,536 memory registers; this is similar to three decimal digits providing the postal addresses for one thousand houses.
2. For a given memory chip, the number of address lines required to identify the registers is determined by the number of registers in the chip. The remaining address lines can be used for selecting the chip.
3. The address lines that are used to select registers in memory, called low-order address lines, can be assigned any logic levels (0 or 1) depending on the register being selected. The address lines that are used to select a chip, called high-order address lines, must have a fixed logic for a given address range.
4. The memory address range of a given chip can be changed by changing the hardware of the Chip Select (CS) line. This line is also known as the Chip Enable (CE) line.

3.2.3 Memory Address Range of a 1K Memory Chip

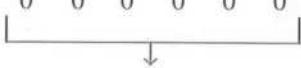
In the previous example, the high-order and the low-order address lines were equally divided, eight each of the 16 address lines. However, if a chip includes more than 256 registers (e.g., 512 or 1024 registers), the number of low-order address lines will be higher than that of the high-order address lines, as shown in the following example.

**Example
3.2**

Explain the memory address range of 1K (1024×8) memory shown in Figure 3.11 and explain the changes in the addresses if the hardware of the CS line is modified.

Solution

The memory chip has 1024 registers; therefore 10 address lines ($A_9 - A_0$) are required to identify the registers. The remaining six address lines ($A_{15} - A_{10}$) of the microprocessor are used for the Chip Select (CS) signal. In Figure 3.11, the memory chip is enabled when the address lines $A_{15} - A_{10}$ are at logic 0. The address lines $A_9 - A_0$ can assume any address of the 1024 registers, starting from all 0s to all 1s, as shown next.

$A_{15} \quad A_{14} \quad A_{13} \quad A_{12} \quad A_{11} \quad A_{10}$	$A_9 \quad A_8 \quad A_7 \quad A_6 \quad A_5 \quad A_4 \quad A_3 \quad A_2 \quad A_1 \quad A_0$	$= 0000H$
0 0 0 0 0 0	0 0 0 0 0 0	
		
Chip Select Logic	1 1 1 1 1 1 1 1 1	$= 03FFH$

The memory addresses range from 0000H to 03FFH.

By combining the high-order and the low-order address lines, we can specify the complete memory address range of a given chip. As explained in the previous example, the memory addresses of the 1K chip in Figure 3.11 can be changed to any other lo-

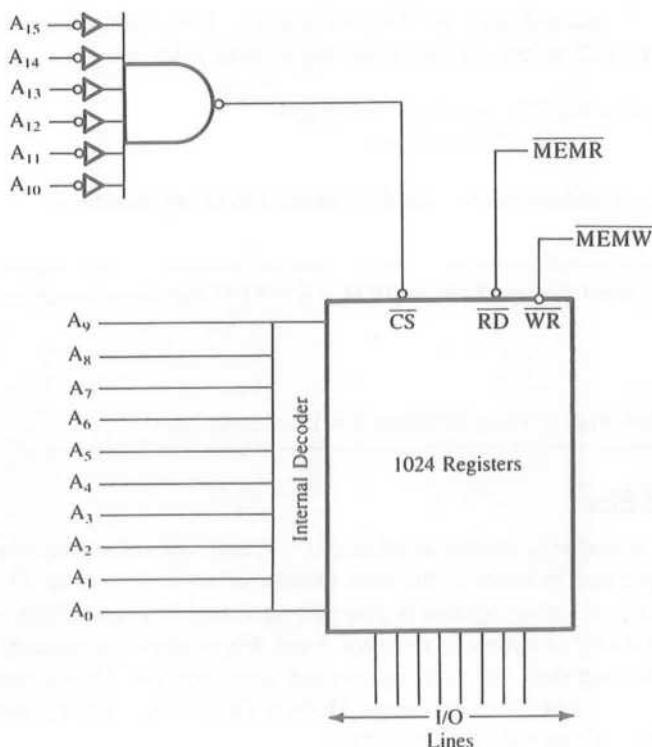


FIGURE 3.11
Memory Address Range: 1024 Bytes of Memory

cation by changing the hardware of the CS line. For example, if A₁₅ is connected to the NAND gate without an inverter, the memory addresses will range from 8000H to 83FFH.

The preceding discussion concerning the memory addresses is equally applicable to the Read-Only memory (ROM). The ROM is in many ways organized the same as the R/W memory. The primary difference between the organization of the two memories is in the control signals; the ROM requires only the Read signal from the MPU.

3.2.4 Memory Address Lines

In the last two examples, the address lines of the memory chips were given. The number of address lines necessary for a given chip can be obtained from data sheets. However, we need to know the relationship between the number of registers in a memory chip and the number of address lines. For a chip with 256 registers, we need 256 binary numbers to identify each register. Each address line can assume only two logic states (0 and 1); therefore,

we need to find the power of 2 that will give us 256 combinations. The problem can be restated as follows: Find x where $2^x = 256$. By taking the log of both sides, we get:

$$\begin{aligned}\log 2^x &= \log 256 \rightarrow x \log 2 = \log 256 \\ x &= \log 256 / \log 2 = 8\end{aligned}$$

Here x represents the number of address lines needed to obtain 256 binary numbers.

Example 3.3	Calculate the address lines required for an 8K-byte ($1024 \times 8 = 8192$ registers) memory chip.
----------------	--

Solution	Number of address lines $x = \log 8192 / \log 2 = 13$ address lines
----------	---

3.2.5 Memory Word Size

Memory devices (chips) are available in various word sizes (1, 4, and 8) and the size of a memory chip is generally specified in terms of the total number of bits it can store. On the other hand, the memory size in a given system is generally specified in terms of bytes. Therefore, it is necessary to design a byte-size memory word. For example, a memory chip of size 1024×4 has 1024 registers and each register can store four bits; thus it can store a total of 4096 ($1024 \times 4 = 4096$) bits. To design 1K-byte (1024×8) memory, we will need two chips; each chip will provide four data lines.

Example 3.4	Calculate the number of memory chips needed to design 8K-byte memory if the memory chip size is 1024×1 .
----------------	---

Solution	The chip 1024×1 has 1024 (1K) registers and each register can store one bit with one data line. We need eight data lines for byte-size memory. Therefore, eight chips are necessary for 1K-byte memory. For 8K-byte memory, we will need 64 chips. We can arrive at the same answer by dividing 8K-byte by $1K \times 1$ as follows:
----------	---

$$8192 \times 8 \div 1024 \times 1 = 64$$

So far we have been concerned primarily with finding necessary address lines and assigning addresses. In the next section, we will examine how the microprocessor communicates with memory using the address lines and control signals.

3.2.6 Memory and Instruction Fetch

The primary function of memory is to store instructions and data and to provide that information to the MPU whenever the MPU requests it. The MPU requests the information

by sending the address of a specific memory register on the address bus and enables the data flow by sending the control signal, as illustrated in the next example.

**Example
3.5**

The instruction code 0100 1111 (4FH) is stored in memory location 2005H. Illustrate the data flow and list the sequence of events when the instruction code is fetched by the MPU.

Solution

To fetch the instruction located in memory location 2005H, the following steps are performed:

1. The program counter places the 16-bit address 2005H of the memory location on the address bus (Figure 3.12).
2. The control unit sends the Memory Read control signal (MEMR, active low) to enable the output buffer of the memory chip.
3. The instruction (4FH) stored in the memory location is placed on the data bus and transferred (copied) to the instruction decoder of the microprocessor.
4. The instruction is decoded and executed according to the binary pattern of the instruction.

Figure 3.12 shows how the 8085 MPU fetches the instruction using the address, the data, and the control buses. Figure 3.12 is similar to Figure 3.2, Memory Read operation, except that Figure 3.12 shows additional details.

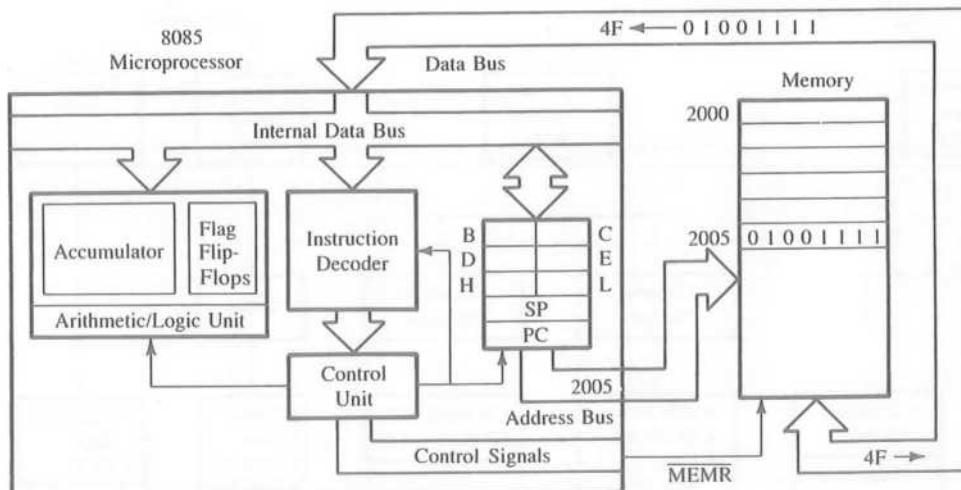


FIGURE 3.12
Instruction Fetch Operation

3.2.7 Memory Classification

As mentioned earlier, memory can be classified into two groups: prime (system or main) memory and storage memory. The R/WM and ROM are examples of prime memory; this is the memory the microprocessor uses in executing and storing programs. This memory should be able to respond fast enough to keep up with the execution speed of the microprocessor. Therefore, it should be random access memory, meaning that the microprocessor should be able to access information from any register with the same speed (independent of its place in the chip). The size of a memory chip is specified in terms of bits. For example, a 1K memory chip means it can store 1K (1024) bits (not bytes). On the other hand, memory in a system such as a PC is specified in bytes. For example, 4M memory in a PC means it has 4 megabytes of memory.

The other group is the storage memory, such as magnetic disks and tapes (see Figure 3.13). This memory is used to store programs and results after the completion of

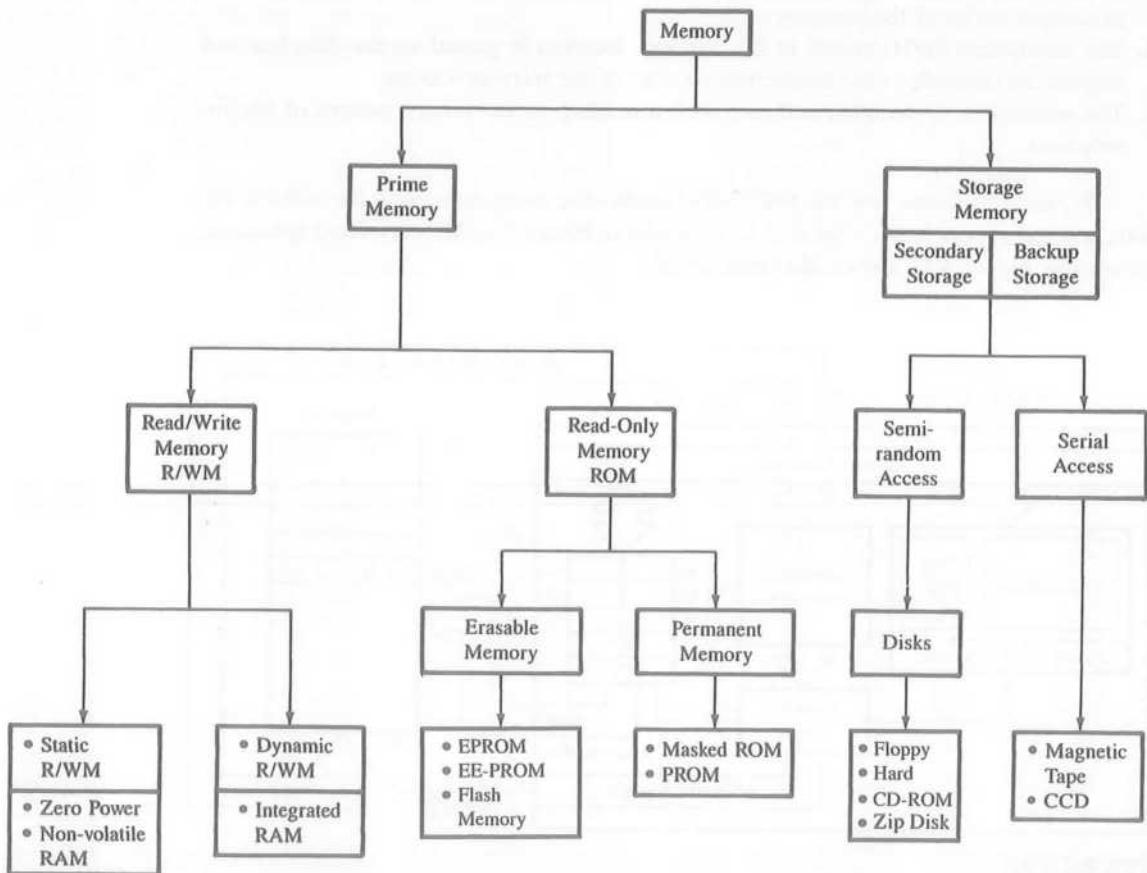


FIGURE 3.13
Memory Classification

program execution. Information stored in these memories is nonvolatile, meaning information remains intact even if the system is turned off. The microprocessor cannot directly execute or process programs stored in these devices; programs need to be copied into the R/W prime memory first. Therefore, the size of the prime memory, such as 512K or 8M (megabytes), determines how large a program the system can process. The size of the storage memory is unlimited; when one disk or tape is full, the next one can be used.

Figure 3.13 shows two groups in storage memory: secondary storage and backup storage. The secondary storage is similar to what you put on a shelf in your study, and the backup is similar to what you store in your attic. The secondary storage and the backup storage include devices such as disks, magnetic tapes, magnetic bubble memory, and charged-coupled devices, as shown in Figure 3.13. The primary features of these devices are high capacity, low cost, and slow access. A disk is similar to a record; the access to the stored information in the disk is semirandom (see Chapter 11 for additional discussion). The remaining devices shown in Figure 3.13 are serial, meaning if information is stored in the middle of the tape, it can be accessed after running half the tape. We will discuss some of these memory storage devices again in Chapter 11. In this chapter, we will focus on various types of prime memory.

Figure 3.13 shows that the prime (system) memory is divided into two main groups: Read/Write memory (R/WM) and Read-Only memory (ROM); each group includes several different types of memory, as discussed below.

R/WM (READ/WRITE MEMORY)

As the name suggests, the microprocessor can write into or read from this memory; it is popularly known as Random Access memory (RAM). It is used primarily for information that is likely to be altered, such as writing programs or receiving data. This memory is volatile, meaning that when the power is turned off, all the contents are destroyed. Two types of R/W memories—static and dynamic—are available; they are described in the following paragraphs.

Static Memory (SRAM) This memory is made up of flip-flops, and it stores the bit as a voltage. Each memory cell requires six transistors; therefore, the memory chip has low density but high speed. This memory is more expensive and consumes more power than the dynamic memory described in the next paragraph. In high-speed processors (such as Intel 486 and Pentium), SRAM known as cache memory is included on the processor chip. In addition, high-speed cache memory is also included external to the processor to improve the performance of a system.

Dynamic Memory (DRAM) This memory is made up of MOS transistor gates, and it stores the bit as a charge. The advantages of dynamic memory are that it has high density and low power consumption and is cheaper than static memory. The disadvantage is that the charge (bit information) leaks; therefore, stored information needs to be read and written again every few milliseconds. This is called refreshing the memory, and it requires extra circuitry, adding to the cost of the system. It is generally economical to use dynamic memory when the system memory size is at least 8K; for small systems, the static mem-

ory is appropriate. However, in recent years, the processor speed has reached beyond 200 MHz, and 1000 MHz processors are in the design stage. In comparison to the processor speed, the DRAM is too slow. To increase the speed of DRAM various techniques are being used. These techniques have resulted in high-speed memory chips such as EDO (Extended Data Out), SDRAM (Synchronous DRAM), and RDRAM (Rambus DRAM).

ROM (READ-ONLY MEMORY)

The ROM is a nonvolatile memory; it retains stored information even if the power is turned off. This memory is used for programs and data that need not be altered. As the name suggests, the information can be read only, which means once a bit pattern is stored, it is permanent or at least semipermanent. The permanent group includes two types of memory: masked ROM and PROM. The semipermanent group also includes two types of memory: EPROM and EE-PROM, as shown in Figure 3.13. The concept underlying the ROM can be explained with the diodes arranged in a matrix format, as shown in Figure 3.14. The horizontal lines are connected to vertical lines only through the diodes; they are not connected where they appear to cross in the diagram. Each of the eight horizontal rows can be viewed as a register with binary addresses ranging from 000 to 111; information is stored by the diodes in the register as 0s or 1s. The presence of a diode stores 1, and its absence stores 0. When a register is selected, the voltage of that line goes high, and the output lines, where diodes are connected, go high. For example, when the memory register 111 is selected, the data byte 0111 1000 (78H) can be read at the data lines D₇–D₀.

The diode representation is a simplified version of the actual MOSFET memory cell. The manufacturer of the ROM designs the MOSFET matrix according to the infor-

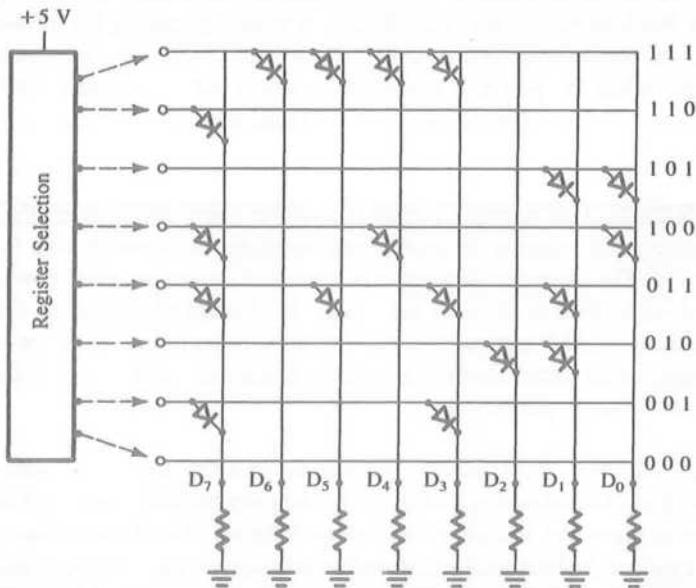


FIGURE 3.14

Functional Representation of ROM Memory Cell

mation to be stored; therefore, information is permanently recorded in the ROM, as a song is recorded on a record. Five types of ROM—masked ROM, PROM, EPROM, EEPROM, and Flash Memory—are described in the following paragraphs.

Masked ROM In this ROM, a bit pattern is permanently recorded by the masking and metalization process. Memory manufacturers are generally equipped to do this process. It is an expensive and specialized process, but economical for large production quantities.

PROM (Programmable Read-Only Memory) This memory has nichrome or poly-silicon wires arranged in a matrix; these wires can be functionally viewed as diodes or fuses. This memory can be programmed by the user with a special PROM programmer that selectively burns the fuses according to the bit pattern to be stored. The process is known as “burning the PROM,” and the information stored is permanent.

EPROM (Erasable Programmable Read-Only Memory) This memory stores a bit by charging the floating gate of an FET. Information is stored by using an EPROM programmer, which applies high voltages to charge the gate. All the information can be erased by exposing the chip to ultraviolet light through its quartz window, and the chip can be reprogrammed. Because the chip can be reused many times, this memory is ideally suited for product development, experimental projects, and college laboratories. The disadvantages of EPROM are (1) it must be taken out of the circuit to erase it, (2) the entire chip must be erased, and (3) the erasing process takes 15 to 20 minutes.

EE-PROM (Electrically Erasable PROM) This memory is functionally similar to EPROM, except that information can be altered by using electrical signals at the register level rather than erasing all the information. This has an advantage in field and remote control applications. In microprocessor systems, software update is a common occurrence. If EE-PROMs are used in the systems, they can be updated from a central computer by using a remote link via telephone lines. Similarly, in a process control where timing information needs to be changed, it can be changed by sending electrical signals from a central place. This memory also includes a Chip Erase mode, whereby the entire chip can be erased in 10 ms vs. 15 to 20 min. to erase an EPROM. However, this memory is expensive compared to EPROM or flash memory (described in the next paragraph).

Flash Memory This is a variation of EE-PROM that is becoming popular. The major difference between the flash memory and EE-PROM is in the erasure procedure. The EE-PROM can be erased at a register level, but the flash memory must be erased either in its entirety or at the sector (block) level. These memory chips can be erased and programmed at least a million times. The power supply requirement for programming these chips was around 12 V, but now chips are available that can be programmed using a power supply as low as 1.8 V. Therefore, this memory is ideally suited for low-power systems.

In a microprocessor-based product, programs are generally written in ROM, and data that are likely to vary are stored in R/WM. For example, in a microprocessor-controlled oven, programs that run the oven are permanently stored in ROM, and data such as baking period, starting time, and temperature are entered in R/W memory through

the keyboard. On the other hand, when microcomputers are used for developing software or for learning purposes, programs are first written in R/W memory, and then stored on a storage memory such as a cassette tape or floppy disk.

ADVANCES IN MEMORY TECHNOLOGY

Memory technology has advanced considerably in recent years. In addition to static and dynamic R/W memory, other options are also available in memory devices. Examples include Zero Power RAM, Nonvolatile RAM, and Integrated RAM.

The Zero Power RAM is a CMOS Read/Write memory with battery backup built internally. It includes lithium cells and voltage-sensing circuitry. When the external power supply voltage falls below 3 V, the power-switching circuitry connects the lithium battery; thus, this memory provides the advantages of R/W and Read-Only memory.

The Nonvolatile RAM is a high-speed static R/W memory array backed up, bit for bit, by EE-PROM array for nonvolatile storage. When the power is about to go off, the contents of R/W memory are quickly stored in the EE-PROM by activating the Store signal on the memory chip, and the stored data can be read into the R/W memory segment when the power is again turned on. This memory chip combines the flexibility of static R/W memory with the nonvolatility of EE-PROM.

The Integrated RAM (iRAM) is a dynamic memory with the refreshed circuitry built on the chip. For the user, it is similar to the static R/W memory. The user can derive the advantages of the dynamic memory without having to build the external refresh circuitry.

3.3

INPUT AND OUTPUT (I/O) DEVICES

Input/output devices are the means through which the MPU communicates with "the outside world." The MPU accepts binary data as input from devices such as keyboards and A/D converters and sends data to output devices such as LEDs or printers. There are two different methods by which I/O devices can be identified: one uses an 8-bit address and the other uses a 16-bit address. These methods are described briefly in the following sections.

3.3.1 I/Os with 8-Bit Addresses (Peripheral-Mapped I/O)

In this type of I/O, the MPU uses eight address lines to identify an input or an output device; this is known as peripheral-mapped I/O (also known as I/O-mapped I/O). This is an 8-bit numbering system for I/Os used in conjunction with Input and Output instructions. This is also known as I/O space, separate from memory space, which is a 16-bit numbering system. The eight address lines can have 256 (2^8 combinations) addresses; thus, the MPU can identify 256 input devices and 256 output devices with addresses ranging from 00H to FFH. The input and output devices are differentiated by the control signals; the MPU uses the I/O Read control signal for input devices and the I/O Write control signal for output devices. The entire range of I/O addresses from 00 to FF is known as an I/O map, and individual addresses are referred to as I/O device addresses or I/O port numbers.

If we use LEDs as output or switches as input, we need to resolve two issues: how to assign addresses and how to connect these I/O devices to the data bus. In a bus architecture, these devices cannot be connected directly to the data bus or the address bus; all

connections must be made through tri-state interfacing devices so they will be enabled and connected to the buses only when the MPU chooses to communicate with them. In the case of memory, we did not have to be concerned with these problems because of the internal address decoding, Read/Write buffers, and availability of CS and control signals of the memory chip. In the case of I/O devices, we need to use external interfacing devices (see Section 3.5).

The steps in communicating with an I/O device are similar to those in communicating with memory and can be summarized as follows:

1. The MPU places an 8-bit address on the address bus, which is decoded by external decode logic (explained in Chapter 4).
2. The MPU sends a control signal (I/O Read or I/O Write) and enables the I/O device.
3. Data are transferred using the data bus.

3.3.2 I/Os with 16-Bit Addresses (Memory-Mapped I/O)

In this type of I/O, the MPU uses 16 address lines to identify an I/O device; an I/O is connected as if it is a memory register. This is known as memory-mapped I/O. The MPU uses the same control signal (Memory Read or Memory Write) and instructions as those of memory. In some microprocessors, such as the Motorola 6800, all I/Os have 16-bit addresses; I/Os and memory share the same memory map (64K). In memory-mapped I/O, the MPU follows the same steps as if it is accessing a memory register.

The peripheral- and memory-mapped I/O techniques will be discussed in detail in the context of interfacing I/O devices (see Chapter 5).

EXAMPLE OF A MICROCOMPUTER SYSTEM

3.4

On the basis of the discussion in the previous sections, we can expand the microcomputer system shown in Figure 3.1 to include additional details. Figure 3.15 illustrates such a system. It shows the 8085 MPU, two types of memory (EPROM and R/WM), input and output, and the buses linking all peripherals (memory and I/Os) to the MPU.

The address lines $A_{15}-A_0$ are used to address memory, and the low-order address bus A_7-A_0 is used to identify the input and the output. The data bus D_7-D_0 is bidirectional and common to all the devices. The four control signals generated by the MPU are connected to different peripheral devices, as shown in Figure 3.15.

The MPU communicates with only one peripheral at a time by enabling the peripheral through its control signal. For example, to send data to the output device, the MPU places the device address (output port number) on the address bus, data on the data bus, and enables the output device using the control signal IOW (I/O Write). The output device latches and displays data if the output device happens to be LEDs. The other peripherals that are not enabled remain in a high impedance state called tri-state (explained later), similar to being disconnected from the system. Figure 3.15 is a simplified block diagram of the system; it does not show such details as data latching and tri-state devices (see Section 3.5).

Figure 3.16 shows an expanded version of the output section and the buses of Figure 3.15. The block diagram includes tri-state bus drivers, a decoder, and a latch. The

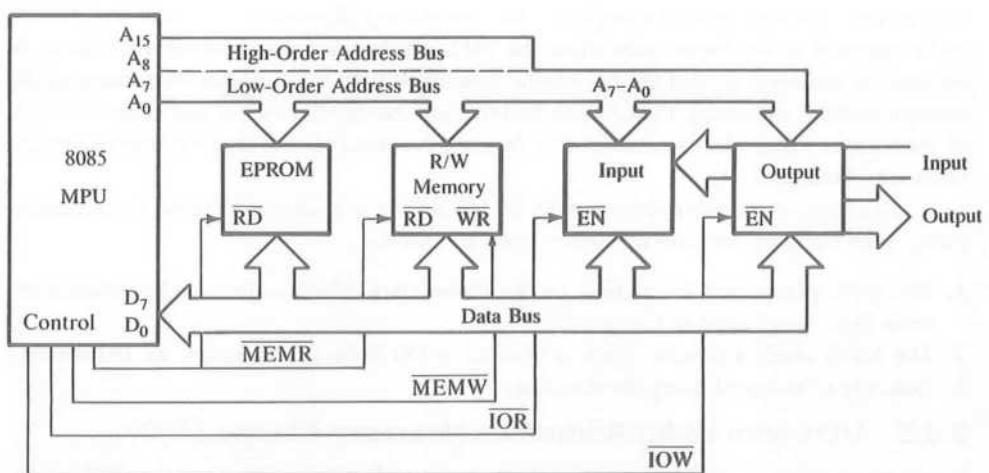


FIGURE 3.15
Example of a Microcomputer System

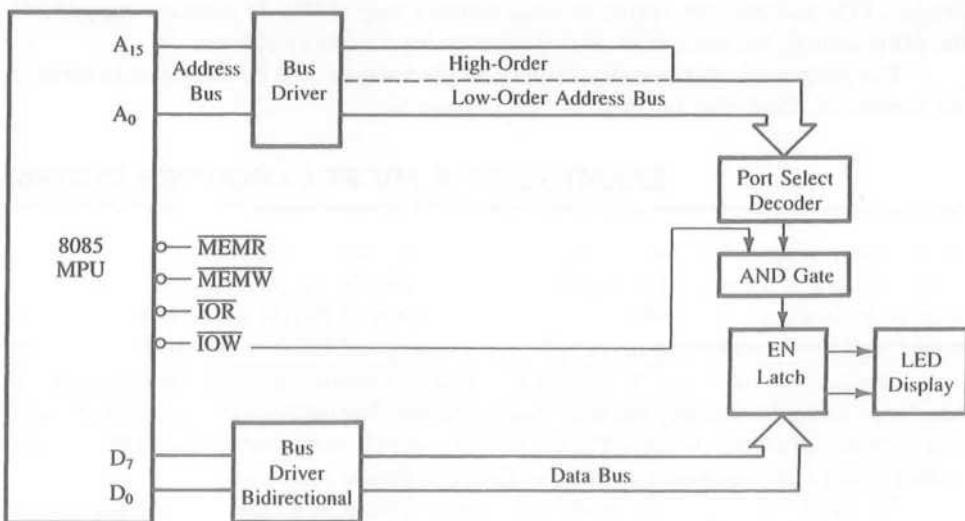


FIGURE 3.16
The Output Section of the Microcomputer System Illustrated in Figure 3.15

bus drivers increase the current driving capacity of the buses, the decoder decodes the address to identify the output port, and the latch holds data output for display. These devices are called **interfacing devices**. The interfacing devices are semiconductor chips that are needed to connect peripherals to the bus system. Before we discuss interfacing concepts, we will review these interfacing devices.

REVIEW: LOGIC DEVICES FOR INTERFACING

3.5

Several types of interfacing devices are necessary to interconnect the components of a bus-oriented system. The devices used in today's microcomputer systems are designed using medium-scale integration (MSI) technology. In addition, tri-state logic devices are essential to proper functioning of the bus-oriented system, in which the same bus lines are shared by several components. The concept underlying the tri-state logic, as well as commonly used interfacing devices, will be reviewed in the following section.

3.5.1 Tri-State Devices

Tri-state logic devices have three states: logic 1, logic 0, and high impedance. The term *Tri-State* is a trademark of National Semiconductor and is used to represent three logic states. A tri-state logic device has a third line called Enable, as shown in Figure 3.17. When this line is activated, the tri-state device functions the same way as ordinary logic devices. When the third line is disabled, the logic device goes into the high impedance state—as if it were disconnected from the system. Ordinarily, current is required to drive a device in logic 0 and logic 1 states. In the high impedance state, practically no current is drawn from the system. Figure 3.17(a) shows a tri-state inverter. When the Enable is high, the circuit functions as an ordinary inverter; when the Enable line is low, the inverter stays in the high impedance state. Figure 3.17(b) also shows a tri-state inverter with active low Enable line—notice the bubble. When the Enable line is high, the inverter stays in the high impedance state.

In microcomputer systems, peripherals are connected in parallel between the address bus and the data bus. However, because of the tri-state interfacing devices, peripherals do not load the system buses. The microprocessor communicates with one device at a time by enabling the tri-state line of the interfacing device. Tri-state logic is critical to proper functioning of the microcomputer.

3.5.2 Buffer

The **buffer** is a logic circuit that amplifies the current or power. It has one input line and one output line (a simple buffer is shown in Figure 3.18a). The logic level of the output

FIGURE 3.17
Tri-State Inverters with Active High
and Active Low Enable Lines

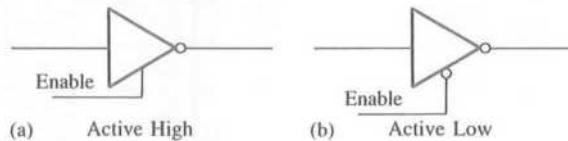
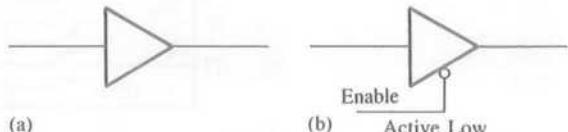


FIGURE 3.18
A Buffer and a Tri-State Buffer



is the same as that of the input; logic 1 input provides logic 1 output (the opposite of an inverter). The buffer is used primarily to increase the driving capability of a logic circuit. It is also known as a driver.

Figure 3.18b shows a tri-state buffer. When the Enable line is low, the circuit functions as a buffer; otherwise it stays in the high impedance state. The buffer is commonly used to increase the driving capability of the data bus and the address bus.

EXAMPLES OF TRI-STATE BUFFERS

The octal buffer 74LS244 shown in Figure 3.19 is a typical example of a tri-state buffer. It is also known as a **line driver** or **line receiver**. This device is commonly used as a driver for the address bus in a bus-oriented system.

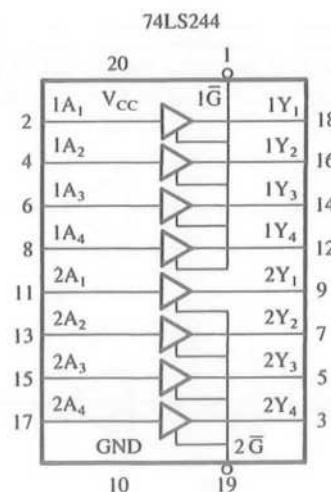
Figure 3.19 shows two groups of four buffers with noninverted tri-state output. The buffers are controlled by two active low Enable lines ($\bar{1G}$ and $\bar{2G}$). Until these lines are enabled, the output of the drivers remains in the high impedance state. Each buffer is capable of sinking 24 mA and sourcing -15 mA of current. The 74LS240 is another example of a tri-state buffer; it has tri-state inverted output.

BIDIRECTIONAL BUFFER

The data bus of a microcomputer system is bidirectional; therefore, it requires a buffer that allows data to flow in both directions. Figure 3.20 shows the logic diagram of the bidirectional buffer 74LS245, also called an **octal bus transceiver**. This is commonly used as a driver for the data bus.

The 74LS245 includes 16 bus drivers, eight for each direction, with tri-state output. The direction of data flow is controlled by the pin DIR. When DIR is high, data flow from the A bus to the B bus; when it is low, data flow from B to A. The schematic also includes an Enable signal (G), which is active low. The Enable signal and the DIR signal are ANDed to activate the bus lines. The device is designed to sink 24 mA and source -15 mA of current.

FIGURE 3.19
Logic Diagram of the 74LS244
Octal Buffer
SOURCE: Courtesy of Texas Instruments Incorporated.



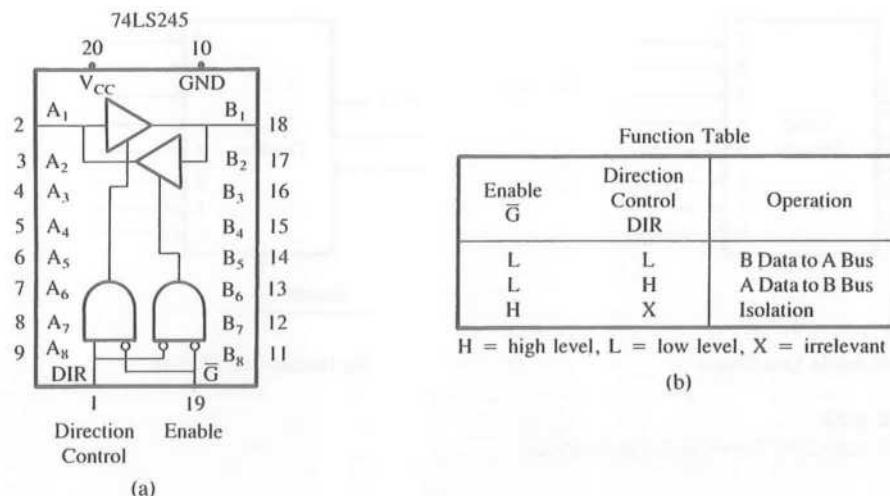


FIGURE 3.20

Logic Diagram and Function Table of the 74LS245 Bidirectional Buffer

SOURCE: Courtesy of Texas Instruments Incorporated.

3.5.3 Decoder

The decoder is a logic circuit that identifies each combination of the signals present at its input. For example, if the input to a decoder has two binary lines, the decoder will have four output lines (Figure 3.21). The two lines can assume four combinations of input signals—00, 01, 10, 11—with each combination identified by the output lines 0 to 3. If the input is 11₂, the output line 3 will be at logic 1, and the others will remain at logic 0. This is called **decoding**. Figure 3.21(a) shows a symbolic representation for a hypothetical 2-to-4 decoder. It is also called a 1-out-of-4 decoder. Various types of decoders are available; for example, 3-to-8, 4-to-16 (to decode binary inputs), and 4-to-10 (to decode BCD input). In general, decoders have active low output lines as well as Enable lines, as

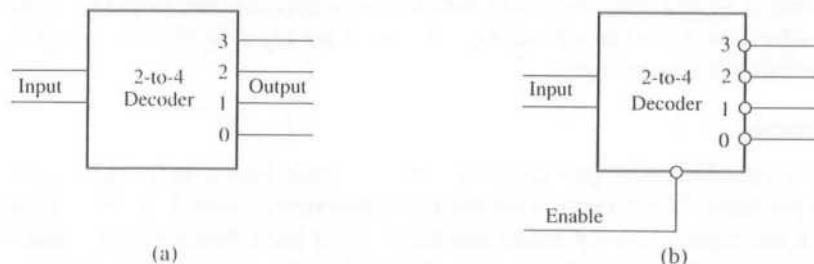


FIGURE 3.21

2-to-4 (1-out-of-4) Decoder Logic Symbol (a) 2-to-4 Decoder with Active Low Output and Enable Line (b)

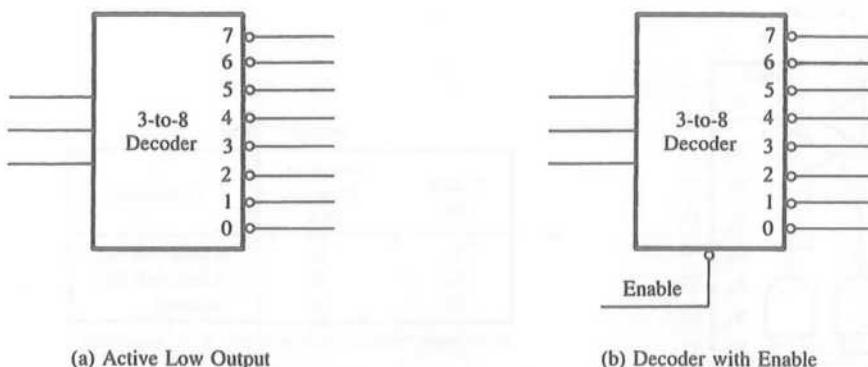


FIGURE 3.22
3-to-8 (1-out-of-8) Decoder Logic Symbol

shown in Figure 3.21(b) and Figure 3.22. The decoders shown in Figures 3.21(b) and 3.22(b) will not function unless they are enabled by a low signal.

A decoder is a commonly used device in interfacing I/O peripherals and memory. In Figure 3.16 the decoder (Port Select Decoder) is used to decode an address bus to identify the output device. Decoders are also built internal to a memory chip to identify individual memory registers.

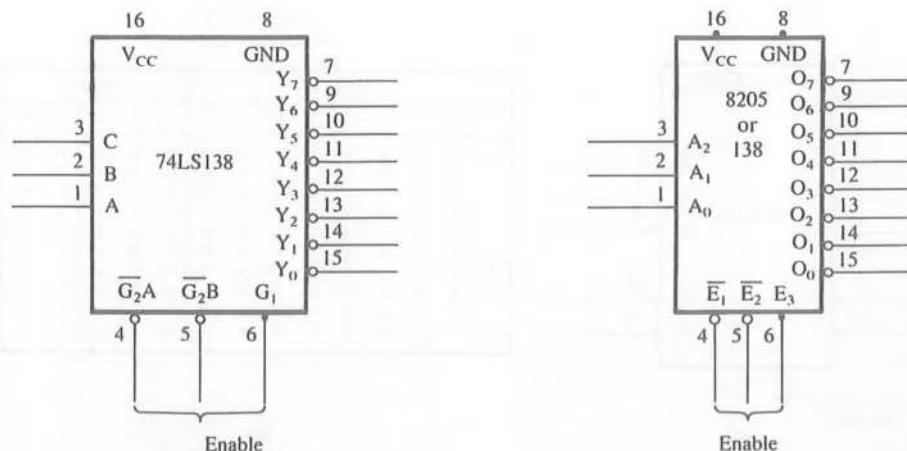
EXAMPLES OF DECODERS

Figure 3.23 shows the block diagrams of two 3-to-8 decoders, the 74LS138 and the Intel 8205. These are pin-compatible, with slight differences in their switching response and current capacity. They are also called **1-out-of-8 binary decoders** or **demultiplexers**. There are several other semiconductor manufacturers that use input/output symbols similar to those of the Intel 8205 for their 74LS138 decoder.

The 74LS138 has three input lines and eight active low output lines. It requires three Enable inputs. Two are active low and one is active high; all three Enable lines should be activated so that the device can function as a decoder. For example, if the 74LS138 is enabled ($G_2A = G_2B = 0$ and $G_1 = 1$) and if the input is 101, the output Y_5 will go low; others will remain high.

3.5.4 Encoder

The encoder is a logic circuit that provides the appropriate code (binary, BCD, etc.) as output for each input signal. The process is the reverse of decoding. Figure 3.24 shows an 8-to-3 encoder; it has eight active low inputs and three output lines. When the input line 0 goes low, the output is 000; when the input line 5 goes low, the output is 101. However, this encoder is unable to provide an appropriate output code if two or more input lines are activated simultaneously. Encoders called priority encoders can resolve the problem of simultaneous inputs.



LS138, S138
Function Table

Inputs			Outputs							
Enable	Select		Y ₀	Y ₁	Y ₂	Y ₃	Y ₄	Y ₅	Y ₆	Y ₇
G ₁	G ₂	C B A	X X X	H H H	H H H	H H H	H H H	H H H	H H H	H H H
X	H	X X X	H H H	H H H	H 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	H H H	H H H
H	L	L L L	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	L	L L 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	L	L H L	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	L	L 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	L	H L L	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	L	H L H	H H H	H H H	H H H	H L H	H H H	H L H	H H H	H H H
H	L	H H L	H H H	H H H	H H H	H H H	H H H	H H L	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	L H H

H = high level, L = low level, X = irrelevant

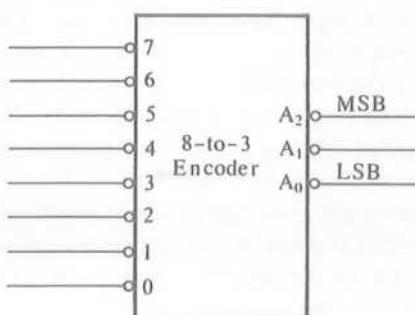
FIGURE 3.23

Logic Diagrams and Function Table of 3-to-8 Decoders

SOURCE: (a) Courtesy of Texas Instruments Incorporated. (b) Intel corporation, *MCS-80/85 Family User's Manual* (Santa Clara, Calif.: Author, 1979), pp. 6-74.

FIGURE 3.24

Logic Symbols: 8-to-3 Encoder



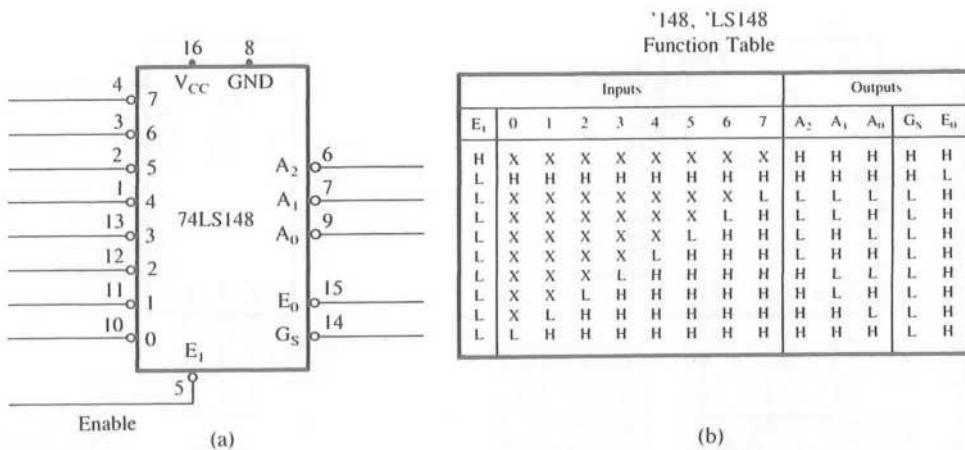


FIGURE 3.25

8-to-3 Priority Encoder—74LS148: Logic Symbol and Function Table

SOURCE: Function table courtesy of Texas Instruments, Incorporated.

Figure 3.25 shows the logic symbol of the 74LS148, an 8-to-3 priority encoder. It has eight inputs and one active low enable signal. It has five output signals—three are encoding lines and two are output-enable indicators. The output lines GS and EO can be used to encode more than eight inputs by cascading these devices. When the encoder is enabled and two or more input signals are activated simultaneously, it ignores the low-priority inputs and encodes the highest-priority input. Similarly, the output of this decoder is active low; when input signal 7 is active, the output is 000 rather than 111.

Encoders are commonly used with keyboards. For each key pressed, the corresponding binary code is placed on the data bus.

3.5.5 D Flip-Flops: Latch and Clocked

In its simplest form, a latch is a D flip-flop, as shown in Figure 3.26; it is also called a transparent latch. A typical example of a latch is the 7475 D flip-flop. In this latch, when the enable signal (G) is high, the output changes according to the input D. Figure 3.26(a) shows that the output Q of the 7475 latch changes during T_{12} and T_{34} and data bits are latched at t_2 and t_4 . On the other hand, in a positive-edge-triggered flip-flop, the output changes with the positive edge of the clock. Figure 3.26(b) shows the output of the 7474 positive-edge-triggered flip-flop. At the first positive going clock (t_1), the input is low; therefore, the output remains low until the next positive edge (t_3). At t_3 , the D input is high; therefore, the output goes high. There is no effect on the output at any other time.

A latch is used commonly to interface output devices. When the MPU sends an output, data are available on the data bus for only a few microseconds; therefore, a latch is used to hold data for display.

EXAMPLES OF LATCHES (REGISTERS)

A typical example of a transparent latch is the 74LS373 shown in Figure 3.27. This octal latch is used to latch 8-bit data.

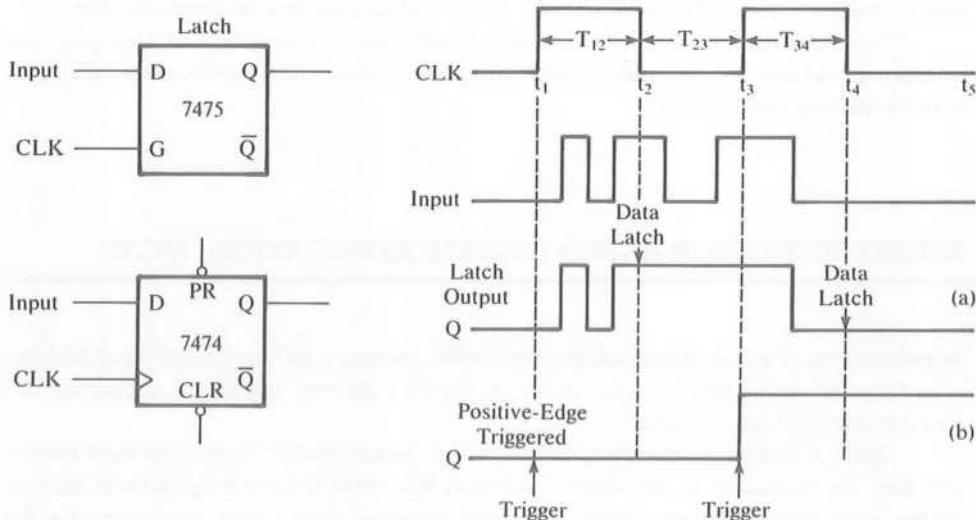
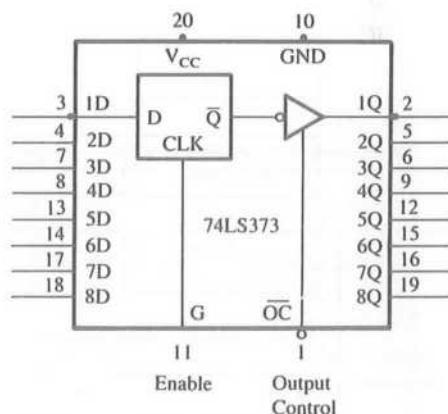


FIGURE 3.26

Output Waveforms of Latch (a) and Positive-Edge-Triggered Flip-Flop (b)

FIGURE 3.27

74LS373 D Latch: Logic Diagram and Function Table



Function Table

Output Control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q_o
H	X	X	Z

The device includes eight D latches with tri-state buffers, and it requires two input signals, Enable (G) and Output Control (OC). The Enable is an active high signal connected to the clock input of the flip-flop. When this signal goes low, data are latched from the data bus. The Output Control signal is active low, and it enables the tri-state buffers to output data to display devices. This latch can also be viewed as a register in a memory chip.

These interfacing devices are discussed briefly here as a review to explain why they are needed and how they are used in microcomputer systems; they will be discussed again with interfacing applications.

3.6

MICROPROCESSOR-BASED SYSTEM APPLICATION: MCTS

In previous sections, we discussed the 8085 MPU, memory, I/O, and interfacing devices.

The MCTS system is again shown in Figure 3.28 with additional interfacing devices discussed in this chapter

Figure 3.28 includes three output devices (fan, heater, and LCD) and one input device, and they are connected to the address and data bus through interfacing devices such as latches and a buffer. A binary address is assigned to each of these interfacing devices, and the microprocessor sends data through these latches when asked through an OUT instruction.

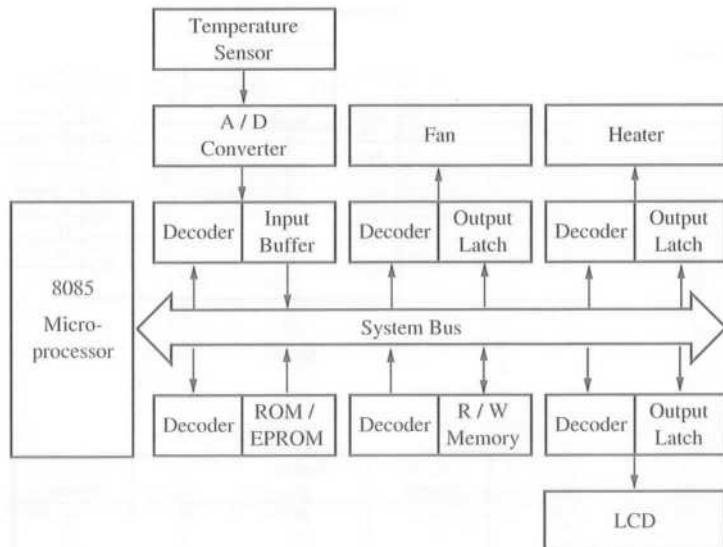


FIGURE 3.28

Micropocessor-Controlled Temperature System (MCTS) with Interfacing Devices

SUMMARY

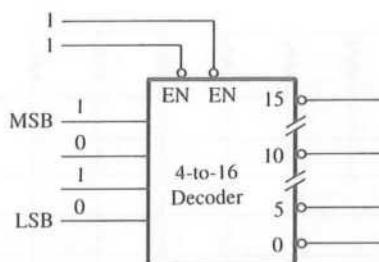
- The microprocessor (MPU) primarily performs four operations: Memory Read, Memory Write, I/O Read, and I/O Write. For each operation, it generates the appropriate control signal.
- To communicate with a peripheral (and memory), the MPU identifies the peripheral or the memory location by its address, transfers data, and provides timing signals.
- **Address Bus**—a group of lines that are used to send a memory address or a device address from the MPU to the memory location or the peripheral. The 8085 microprocessor has 16 address lines.
- **Data Bus**—a group of bidirectional lines used to transfer data between the MPU and peripherals (or memory). The 8085 microprocessor has eight data lines.
- **Control Bus**—single lines that are generated by the MPU to provide timing of various operations.
- The 8085 microprocessor has six general-purpose 8-bit registers to store data and an accumulator to perform arithmetic and logical operations.
- The data conditions, after an arithmetic or logical operation, are indicated by setting or resetting the flip-flops called flags. The 8085 includes five flags: Sign, Zero, Auxiliary Carry, Parity, and Carry.
- The 8085 has two 16-bit registers: the program counter and the stack pointer. The program counter is used to sequence the execution of a program, and the stack pointer is used as a memory pointer for the stack memory.
- The 8085 can respond to four externally initiated operations: Reset, Interrupt, Ready, and Hold.
- Memory is a group of registers, arranged in a sequence, to store bits. The 8085 MPU requires an 8-bit-wide memory word and uses the 16-bit address to select a register called a memory location.
- The memory addresses assigned to a memory chip in a system are called the memory map. The assignment of memory addresses is done through the Chip Select logic.
- Memory can be classified primarily into two groups: Read/Write memory (R/WM) and Read-Only memory (ROM). The R/W memory is volatile and can be used to read and write information. This is also called the user memory. The ROM is a non-volatile memory and the information written into this memory is permanent.
- Input/Output devices or peripherals can be interfaced with the 8085 MPU in two ways, peripheral I/O and memory-mapped I/O. In peripheral I/O, the MPU uses an 8-bit address to identify an I/O, and IN and OUT instructions for data transfer. In memory-mapped I/O, the MPU uses a 16-bit address to identify an I/O and memory-related instructions for data transfer.
- To execute an instruction, the MPU places the 16-bit address on the address bus, sends the control signal to enable the memory chip, and fetches the instruction. The instruction is then decoded and executed.
- To interconnect peripherals with the 8085 MPU, additional logic circuits, called interfacing devices, are necessary. These circuits include devices such as buffers, decoders, encoders, and latches.

- A tri-state logic device has three states: two logic states and one high impedance state. When the device is not enabled, it remains in high impedance and does not draw any current from the system.

QUESTIONS AND PROBLEMS

1. List the four operations commonly performed by the MPU.
2. What is a bus?
3. Specify the function of the address bus and the direction of the information flow on the address bus.
4. How many memory locations can be addressed by a microprocessor with 14 address lines?
5. How many address lines are necessary to address two megabytes (2048K) of memory?
6. Why is the data bus bidirectional?
7. Specify the four control signals commonly used by the 8085 MPU.
8. Specify the control signal and the direction of the data flow on the data bus in a memory-write operation.
9. What is the function of the accumulator?
10. What is a flag?
11. Why are the program counter and the stack pointer 16-bit registers?
12. While executing a program, when the 8085 MPU completes the fetching of the machine code located at the memory address 2057H, what is the content of the program counter?
13. Specify the number of registers and memory cells in a 128×4 memory chip.
14. How many bits are stored by a 256×4 memory chip? Can this chip be specified as 128-byte memory?
15. What is the memory word size required in an 8085 system?
16. If the memory chip size is 2048×8 bits, how many chips are required to make up 16K-byte memory?
17. If the memory chip size is 1024×4 bits, how many chips are required to make up 2K (2048) bytes of memory?
18. If the memory chip size is 256×1 bits, how many chips are required to make up 1K (1024) bytes of memory?
19. What is the function of the WR signal on the memory chip?
20. How many address lines are necessary on the chip of 2K (2048) byte memory?
21. The memory map of a 4K (4096) byte memory chip begins at the location 2000H. Specify the address of the last location on the chip and the number of pages in the chip.
22. The memory address of the last location of a 1K byte memory chip is given as FBFFH. Specify the starting address.
23. The memory address of the last location of an 8K byte memory chip is FFFFH. Find the starting address.
24. In Figure 3.10(a), eliminate all the inverters and connect address lines A_8-A_{15} directly to the NAND gate. Identify the memory address range of the chip.

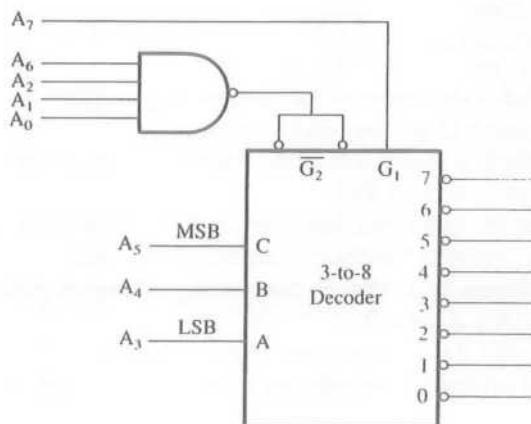
FIGURE 3.29
Logic Diagram of a 4-to-16 Decoder



25. In Figure 3.10(b), if the chip is selected and the address lines $A_7 - A_0$ have 01000111, specify the complete address in Hex of the register selected.
26. In Figure 3.11, connect A_{13} to the NAND gate without an inverter, and identify the memory map.
27. Specify the Hex address of the register selected in Question 26 if the address on the address lines $A_9 - A_0$ is 00 1111 1000.
28. How many address lines are used to identify an I/O port in the peripheral I/O and in the memory-mapped I/O methods?
29. What are tri-state devices and why are they essential in a bus-oriented system?
30. In Figure 3.19, if the input to the octal buffer is 4FH and the enable lines $1\bar{G}$ and $2\bar{G}$ are high, what is the output of the buffer?
31. In Figure 3.20, if signals G (Enable) and DIR are low, specify the direction of data flow.
32. Specify the output line of the 4-to-16 decoder that goes low if the input to the decoder is as shown in Figure 3.29.
33. In Figure 3.30, specify the output line that goes low if the input (including the enable lines) to the 3-to-8 decoder (74LS138) is

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	0	1	1	1

FIGURE 3.30
The 3-to-8 Decoder (74LS138)



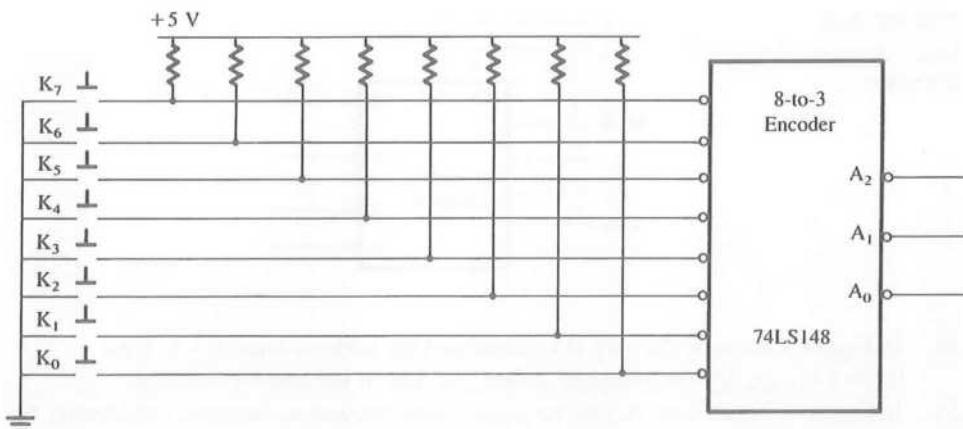


FIGURE 3.31
The 8-to-3 Encoder (74LS148)

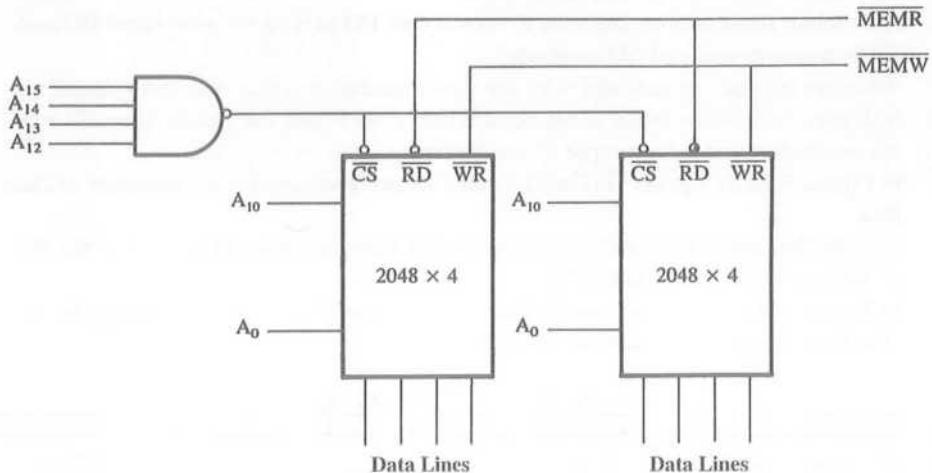


FIGURE 3.32
Memory Schematic

34. What is the output of the encoder (Figure 3.31) if the key K₆ is pushed? (See Figure 3.25 for the function table.)
35. What is a transparent latch, and why is it necessary to use a latch with output devices such as LEDs?
36. List the high-order, low-order, and don't care address lines in Figure 3.32. How many pages of memory does the chip include?
37. In Figure 3.32, identify the memory addresses, assuming the don't care address line A₁₁ at logic 0.
38. Specify the entire memory map (addresses) of the schematic shown in Figure 3.32, and explain the significance of the don't care address line on memory addresses.

4

8085 Microprocessor Architecture and Memory Interfacing

The 8085 microprocessor is a much improved version of its predecessor, the 8080A. The 8085 includes on its chip most of the logic circuitry for performing computing tasks and for communicating with peripherals. However, eight of its bus lines are **multiplexed**; that is, they are time-shared by the low-order address and data. This chapter discusses the 8085 architecture in detail and illustrates techniques for demultiplexing the bus and generating the necessary control signals.

Later, the chapter describes a typical 8085-based microcomputer designed with general-purpose memory and I/O devices; it also illustrates the bus timing signals in executing an instruction. Then, it examines the requirements of a memory chip based on the timing signals and derives the steps necessary in interfacing memory. In addition, the chapter includes illustrations of a special-purpose device, the 8155 and its interfacing.

OBJECTIVES

- Recognize the functions of various pins of the 8085 microprocessor.
- Explain the bus timings in fetching an instruction from memory.
- Explain how to demultiplex the AD₇–AD₀ bus using a latch.
- Draw a logic schematic to generate four control signals, using the 8085 IO/M, RD, and WR signals: (1) MEMR, (2) MEMW, (3) IOR, and (4) IOW. Explain the functions of these control signals.
- List the various internal units that make up the 8085 architecture, and explain their functions in decoding and executing an instruction.
- Draw the block diagram of an 8085-based microcomputer.

- List the steps performed by the 8085 microprocessor, and identify the contents of buses when an instruction is being executed.
- Analyze a memory interfacing circuit, and specify the memory addresses of a given memory device.
- Recognize partial decoding and identify fold-back (mirror) memory space.

4.1

THE 8085 MPU

The term **microprocessing unit** (MPU) is similar to the term **central processing unit** (CPU) used in traditional computers. We define the MPU as a device or a group of devices (as a unit) that can communicate with peripherals, provide timing signals, direct data flow, and perform computing tasks as specified by the instructions in memory. The unit will have the necessary lines for the address bus, the data bus, and the control signals, and would require only a power supply and a crystal (or equivalent frequency-determining components) to be completely functional.

Using this description, the 8085 microprocessor can almost qualify as an MPU, but with the following two limitations.

1. The low-order address bus of the 8085 microprocessor is **multiplexed** (time-shared) with the data bus. The buses need to be demultiplexed.
2. Appropriate control signals need to be generated to interface memory and I/O with the 8085. (Intel has some specialized memory and I/O devices that do not require such control signals.)

This section shows how to demultiplex the bus and generate the control signals after describing the 8085 microprocessor and illustrates the bus timings.

4.1.1 The 8085 Microprocessor

The 8085A (commonly known as the 8085) is an 8-bit general-purpose microprocessor capable of addressing 64K of memory. The device has forty pins, requires a +5 V single power supply, and can operate with a 3-MHz single-phase clock. The 8085A-2 version can operate at the maximum frequency of 5 MHz. The 8085 is an enhanced version of its predecessor, the 8080A; its instruction set is upward-compatible with that of the 8080A, meaning that the 8085 instruction set includes all the 8080A instructions plus some additional ones.

Figure 4.1 shows the logic pinout of the 8085 microprocessor. All the signals can be classified into six groups: (1) address bus, (2) data bus, (3) control and status signals, (4) power supply and frequency signals, (5) externally initiated signals, and (6) serial I/O ports.

ADDRESS BUS

The 8085 has 16 signal lines (pins) that are used as the address bus; however, these lines are split into two segments: A₁₅–A₈ and AD₇–AD₀. The eight signal lines, A₁₅–A₈, are

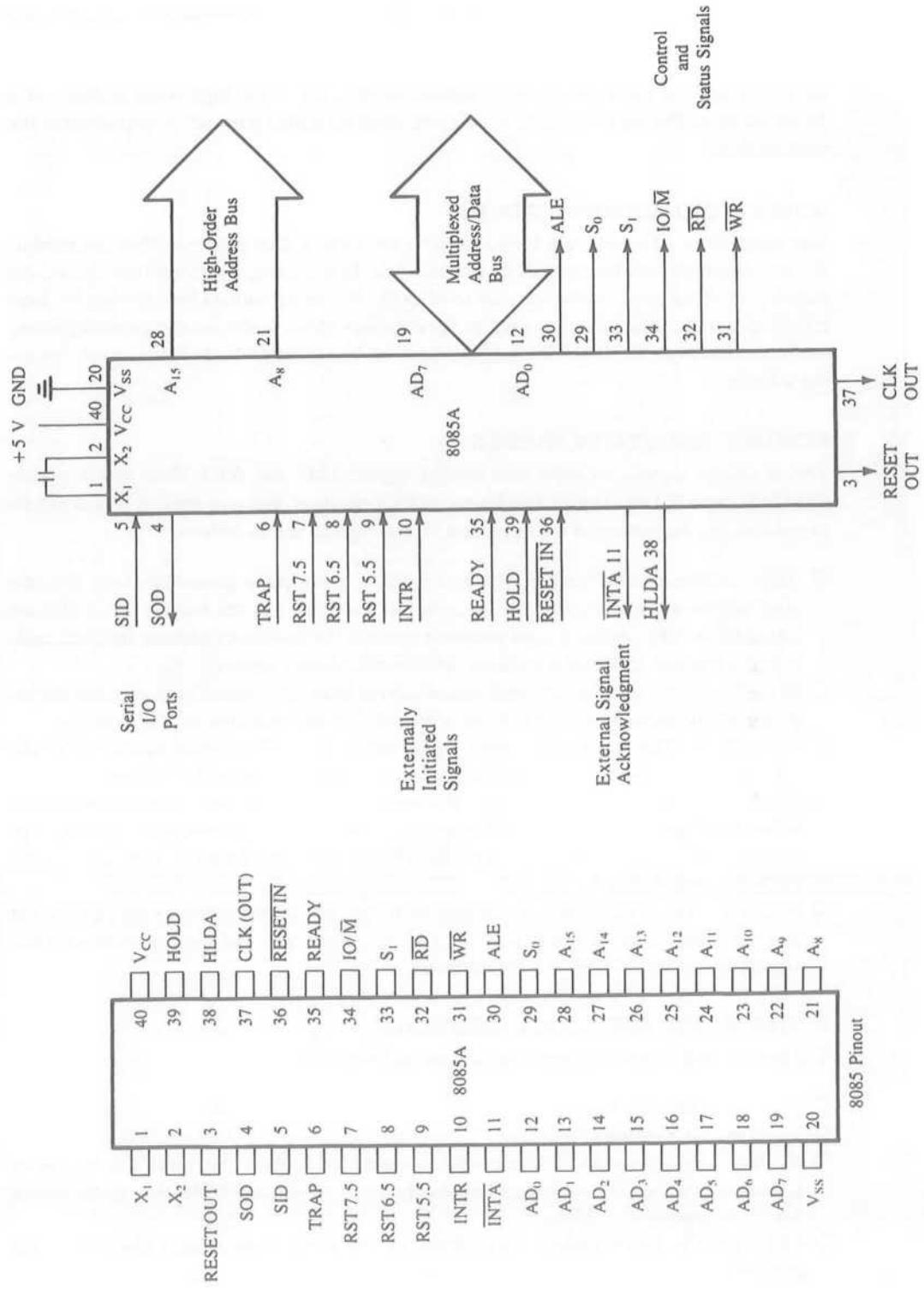


FIGURE 4.1

The 8085 Microprocessor Pinout and Signals

NOTE: The 8085A is commonly known as the 8085.

SOURCE (Pinout): Intel Corporation, *Embedded Microprocessors* (Santa Clara, Calif.: Author, 1994), pp. 1-11.

unidirectional and used for the most significant bits, called the high-order address, of a 16-bit address. The signal lines AD_7 – AD_0 are used for a dual purpose, as explained in the next section.

MULTIPLEXED ADDRESS/DATA BUS

The signal lines AD_7 – AD_0 are bidirectional: they serve a dual purpose. They are used as the low-order address bus as well as the data bus. In executing an instruction, during the earlier part of the cycle, these lines are used as the low-order address bus. During the later part of the cycle, these lines are used as the data bus. (This is also known as multiplexing the bus.) However, the low-order address bus can be separated from these signals by using a latch.

CONTROL AND STATUS SIGNALS

This group of signals includes two control signals (\overline{RD} and \overline{WR}), three status signals (IO/M , S_1 , and S_0) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation. These signals are as follows:

- ALE—Address Latch Enable: This is a positive going pulse generated every time the 8085 begins an operation (machine cycle). It indicates that the bits on AD_7 – AD_0 are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines, A_7 – A_0 .
- RD—Read: This is a Read control signal (active low). This signal indicates that the selected I/O or memory device is to be read and data are available on the data bus.
- WR—Write: This is a Write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.
- IO/M: This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low, it indicates a memory operation. This signal is combined with RD (Read) and WR (Write) to generate I/O and memory control signals.
- S_1 and S_0 : These status signals, similar to IO/\overline{M} , can identify various operations, but they are rarely used in small systems. (All the operations and their associated status signals are listed in Table 4.1 for reference.)

POWER SUPPLY AND CLOCK FREQUENCY

The power supply and frequency signals are as follows:

- V_{CC} : +5 V power supply.
- V_{SS} : Ground Reference.
- X_1 , X_2 : A crystal (or RC, LC network) is connected at these two pins. The frequency is internally divided by two; therefore, to operate a system at 3 MHz, the crystal should have a frequency of 6 MHz.
- CLK (OUT)—Clock Output: This signal can be used as the system clock for other devices.

TABLE 4.1
8085 Machine Cycle Status and Control Signals

Machine Cycle	Status			Control Signals
	IO/M	S ₁	S ₀	
Opcode Fetch	0	1	1	$\overline{RD} = 0$
Memory Read	0	1	0	$\overline{RD} = 0$
Memory Write	0	0	1	$\overline{WR} = 0$
I/O Read	1	1	0	$\overline{RD} = 0$
I/O Write	1	0	1	$\overline{WR} = 0$
Interrupt Acknowledge	1	1	1	$\overline{INTA} = 0$
Halt	Z	0	0	
Hold	Z	X	X	$\overline{RD}, \overline{WR} = Z$ and $\overline{INTA} = 1$
Reset	Z	X	X	

NOTE: Z = Tri-state (high impedance)

X = Unspecified

EXTERNALLY INITIATED SIGNALS, INCLUDING INTERRUPTS

The 8085 has five interrupt signals (see Table 4.2) that can be used to interrupt a program execution. One of the signals, INTR (Interrupt Request), is identical to the 8080A microprocessor interrupt signal (INT); the others are enhancements to the 8080A. The microprocessor acknowledges an interrupt request by the INTA (Interrupt Acknowledge) signal. (The interrupt process is discussed in Chapter 12.)

In addition to the interrupts, three pins—RESET, HOLD, and READY—accept the externally initiated signals as inputs. To respond to the HOLD request, the 8085 has one

TABLE 4.2
8085 Interrupts and Externally Initiated Signals

<input type="checkbox"/> INTR (Input)	Interrupt Request: This is used as a general-purpose interrupt; it is similar to the INT signal of the 8080A.
<input type="checkbox"/> INTA (Output)	Interrupt Acknowledge: This is used to acknowledge an interrupt.
<input type="checkbox"/> RST 7.5 (Inputs)	Restart Interrupts: These are vectored interrupts that transfer the program control to specific memory locations. They have higher priorities than the INTR interrupt. Among these three, the priority order is 7.5, 6.5, and 5.5.
<input type="checkbox"/> TRAP (Input)	This is a nonmaskable interrupt and has the highest priority.
<input type="checkbox"/> HOLD (Input)	This signal indicates that a peripheral such as a DMA (Direct Memory Access) controller is requesting the use of the address and data buses.
<input type="checkbox"/> HLDA (Output)	Hold Acknowledge: This signal acknowledges the HOLD request.
<input type="checkbox"/> READY (Input)	This signal is used to delay the microprocessor Read or Write cycles until a slow-responding peripheral is ready to send or accept data. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high.

signal called HLDA (Hold Acknowledge). The functions of these signals were previously discussed in Section 3.1.3. The RESET is again described below, and others are listed in Table 4.2 for reference.

- **RESET IN:** When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated, and the MPU is reset.
- **RESET OUT:** This signal indicates that the MPU is being reset. The signal can be used to reset other devices.

SERIAL I/O PORTS

The 8085 has two signals to implement the serial transmission: SID (Serial Input Data) and SOD (Serial Output Data). In serial transmission, data bits are sent over a single line, one bit at a time, such as the transmission over telephone lines. This will be discussed in Chapter 16 on serial I/O.

In this chapter, we will focus on the first three groups of signals; others will be discussed in later chapters.

4.1.2 Microprocessor Communication and Bus Timings

To understand the functions of various signals of the 8085, we should examine the process of communication (reading from and writing into memory) between the microprocessor and memory and the timings of these signals in relation to the system clock. The first step in the communication process is reading from memory or fetching an instruction. This can be easily understood using an analogy of how a package is picked up from your house by a shipping company such as Federal Express. The steps are as follows:

1. A courier gets the address from the office; he or she drives the pickup van, finds the street, and looks for your house number.
2. The courier rings the bell.
3. Somebody in the house opens the door and gives the package to the courier, and the courier returns to the office with the package.
4. The internal office staff disposes the package according to the instructions given by the customer.

Now let us examine the steps in the following example of how the microprocessor fetches or gets a machine code from memory.

Example 4.1

Refer to Example 3.5 in the last chapter (Section 3.2.6): Illustrate the steps and the timing of data flow when the instruction code 0100 1111 (4FH—MOV C,A), stored in location 2005H, is being fetched.

Solution

To fetch the byte (4FH), the MPU needs to identify the memory location 2005H and enable the data flow from memory. This is called the Fetch cycle. The data flow is shown in Figure 4.2, and the timings are explained below.

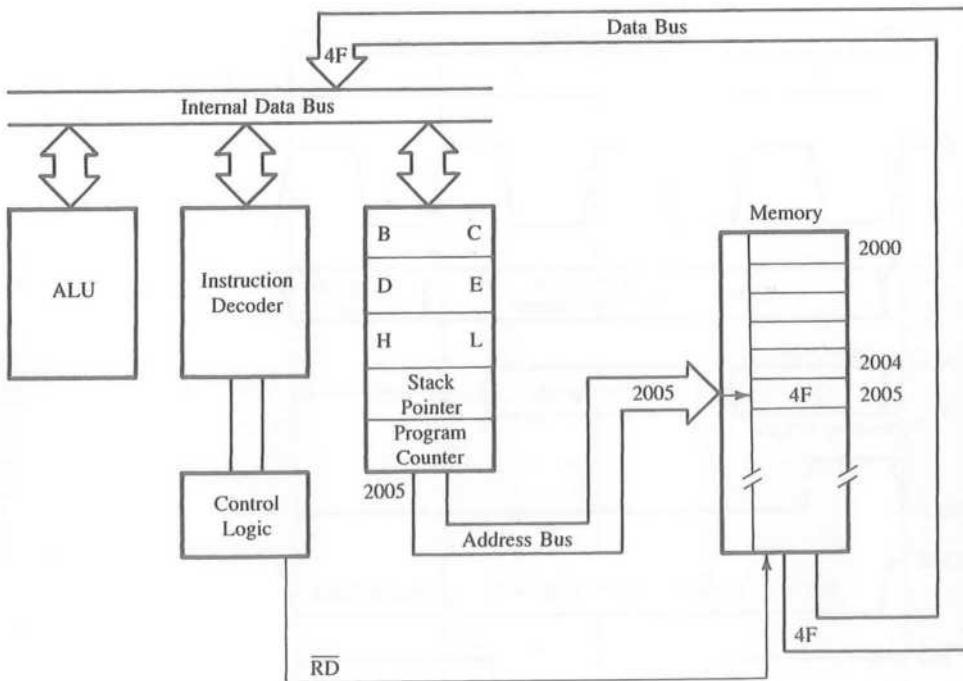


FIGURE 4.2
Data Flow from Memory to the MPU

Figure 4.3 shows the timing of how a data byte is transferred from memory to the MPU; it shows five different groups of signals in relation to the system clock. The address bus and data bus are shown as two parallel lines. This is a commonly used practice to represent logic levels of groups of lines; some lines are high and others are low. The crossover of the lines indicates that a new byte (information) is placed on the bus, and a dashed straight line indicates the high impedance state. To fetch the byte, the MPU performs the following steps:

Step 1: The microprocessor places the 16-bit memory address from the program counter (PC) on the address bus (Figure 4.2). In our analogy, this is the equivalent of our courier getting on the road to find the address.

Figure 4.3 shows that at T_1 , the high-order memory address 20H is placed on the address lines $A_{15}-A_8$, the low-order memory address 05H is placed on the bus AD_7-AD_0 , and the ALE signal goes high. Similarly, the status signal IO/M goes low, indicating that this is a memory-related operation. (For the sake of clarity, the other two status signals, S_1 and S_0 , are not shown in Figure 4.3; they will be discussed in the next section.)

Step 2: The control unit sends the control signal RD to enable the memory chip (Figure 4.2). This is similar to ringing the doorbell in our analogy of a package pickup.

The control signal \overline{RD} is sent out during the clock period T_2 , thus enabling the memory chip (Figure 4.3). The RD signal is active during two clock periods.

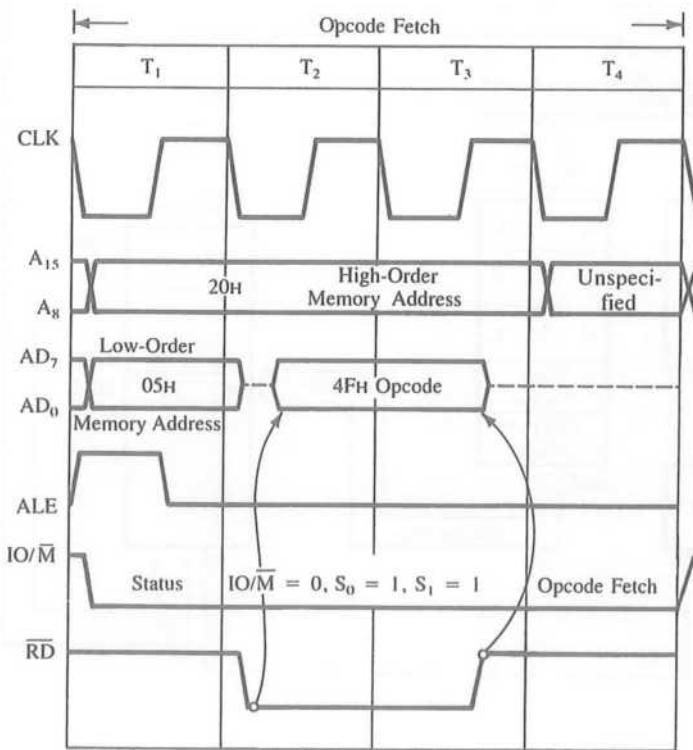


FIGURE 4.3
Timing: Transfer of Byte from Memory to MPU

Step 3: The byte from the memory location is placed on the data bus.

When the memory is enabled, the instruction byte (4FH) is placed on the bus AD₇-AD₀ and transferred to the microprocessor. The RD signal causes 4FH to be placed on bus AD₇-AD₀ (shown by the arrow), and when RD goes high, it causes the bus to go into high impedance.

Step 4: The byte is placed in the instruction decoder of the microprocessor, and the task is carried out according to the instruction.

The machine code or the byte (4FH) is decoded by the instruction decoder, and the contents of the accumulator are copied into register C. This task is performed during the period T₄ in Figure 4.3.

The above four steps are similar to the steps listed in our analogy of the package pickup.

4.1.3 Demultiplexing the Bus AD₇-AD₀

The need for demultiplexing the bus AD₇-AD₀ becomes easier to understand after examining Figure 4.3. This figure shows that the address on the high-order bus (20H) remains on the bus for three clock periods. However, the low-order address (05H) is lost after the

first clock period. This address needs to be latched and used for identifying the memory address. If the bus AD_7-AD_0 is used to identify the memory location (2005H), the address will change to 204FH after the first clock period.

Figure 4.4 shows a schematic that uses a latch and the ALE signal to demultiplex the bus. The bus AD_7-AD_0 is connected as the input to the latch 74LS373. The ALE signal is connected to the Enable (G) pin of the latch, and the Output control (\bar{OC}) signal of the latch is grounded.

Figure 4.3 shows that the ALE goes high during T_1 . When the ALE is high, the latch is transparent; this means that the output changes according to input data. During T_1 , the output of the latch is 05H. When the ALE goes low, the data byte 05H is latched until the next ALE, and the output of the latch represents the low-order address bus A_7-A_0 after the latching operation.

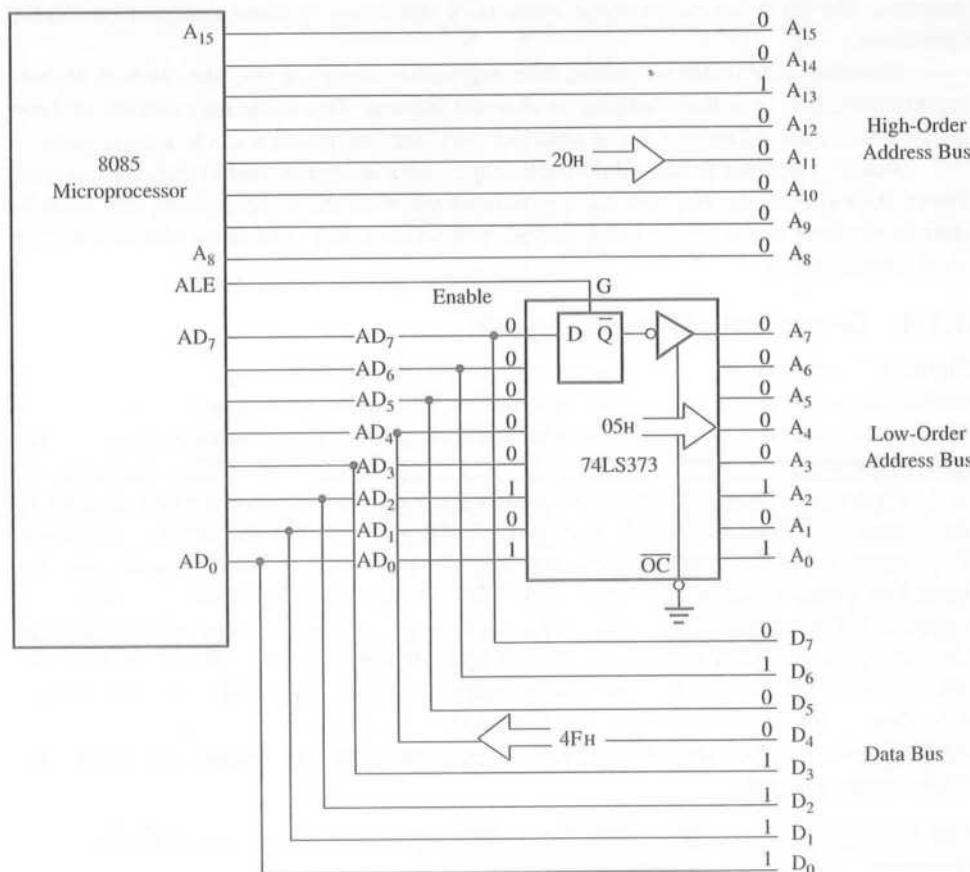


FIGURE 4.4
Schematic of Latching Low-Order Address Bus

Intel has circumvented the problem of demultiplexing the low-order bus by designing special devices such as the 8155 (256 bytes of R/W memory + I/Os), which is compatible with the 8085 multiplexed bus. These devices internally demultiplex the bus using the ALE signal (see Figures 4.18 and 4.19).

After carefully examining Figure 4.3, we can make the following observations:

1. The machine code 4FH (0100 1000) is a one-byte instruction that copies the contents of the accumulator into register C.
2. The 8085 microprocessor requires one external operation—fetching a machine code* from memory location 2005H.
3. The entire operation—fetching, decoding, and executing—requires four clock periods.

Now we can define three terms—instruction cycle, machine cycle, and T-state—and use these terms later for examining timings of various 8085 operations (Section 4.2).

Instruction cycle is defined as the time required to complete the execution of an instruction. The 8085 instruction cycle consists of one to six machine cycles or one to six operations.

Machine cycle is defined as the time required to complete one operation of accessing memory, I/O, or acknowledging an external request. This cycle may consist of three to six T-states. In Figure 4.3, the instruction cycle and the machine cycle are the same.

T-state is defined as one subdivision of the operation performed in one clock period. These subdivisions are internal states synchronized with the system clock, and each T-state is precisely equal to one clock period. The terms T-state and clock period are often used synonymously.

4.1.4 Generating Control Signals

Figure 4.3 shows the RD (Read) as a control signal. Because this signal is used both for reading memory and for reading an input device, it is necessary to generate two different Read signals: one for memory and another for input. Similarly, two separate Write signals must be generated.

Figure 4.5 shows that four different control signals are generated by combining the signals RD, WR, and IO/M. The signal IO/M goes low for the memory operation. This signal is ANDed with RD and WR signals by using the 74LS32 quadruple two-input OR gates, as shown in Figure 4.5. The OR gates are functionally connected as negative NAND gates. When both input signals go low, the outputs of the gates go low and generate MEMR (Memory Read) and MEMW (Memory Write) control signals. When the IO/M signal goes high, it indicates the peripheral I/O operation. Figure 4.5 shows that this signal is complemented using the Hex inverter 74LS04 and ANDed with the RD and WR signals to generate IOR (I/O Read) and IOW (I/O Write) control signals.

*This code is an operation code (opcode) that instructs the microprocessor to perform the specified task. The term *opcode* was explained in Chapter 2 (Section 2.3).

FIGURE 4.5
Schematic to Generate Read/Write Control Signals for Memory and I/O

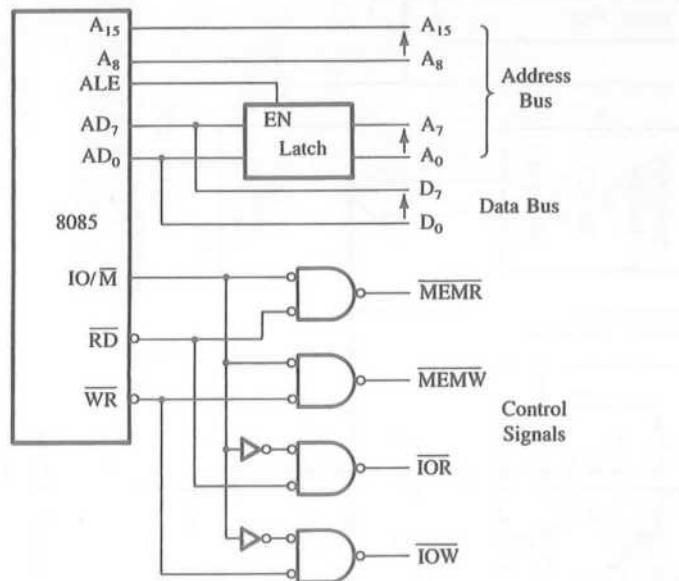
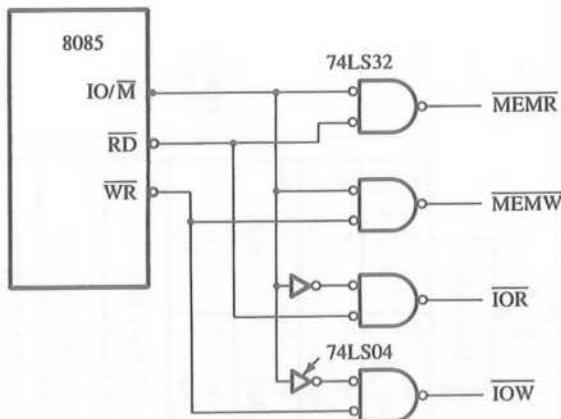


FIGURE 4.6
8085 Demultiplexed Address and Data Bus with Control Signals

To demultiplex the bus and to generate the necessary control signals, the 8085 microprocessor requires a latch and logic gates to build the MPU, as shown in Figure 4.6. This MPU can be interfaced with any memory or I/O.

4.1.5 A Detailed Look at the 8085 MPU and Its Architecture

Figure 4.7 shows the internal architecture of the 8085 beyond the programmable registers we discussed previously. It includes the ALU (Arithmetic/Logic Unit), Timing and

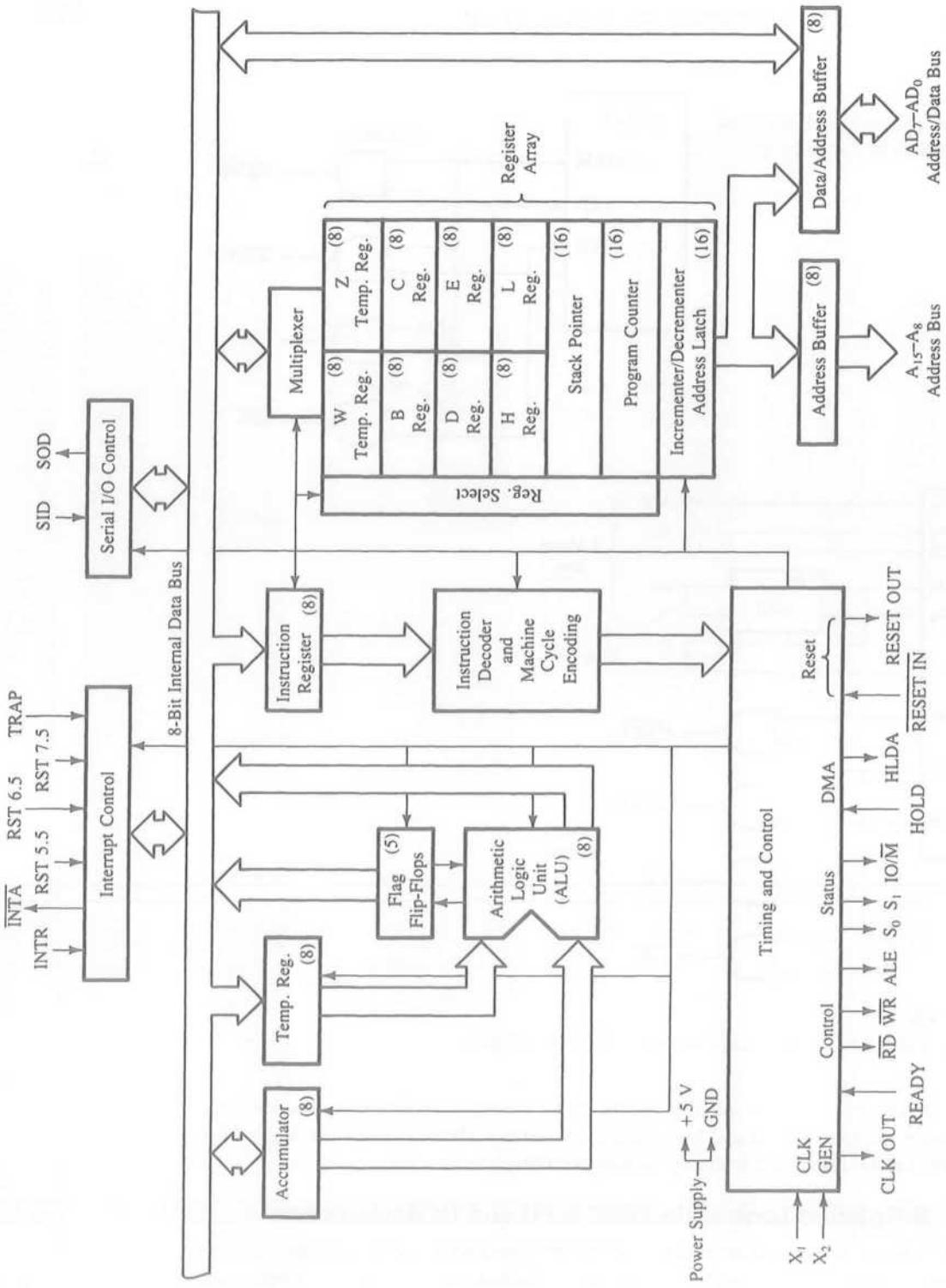


FIGURE 4.7

The 8085A Microprocessor: Functional Block Diagram

NOTE: The 8085A microprocessor is commonly known as the 8085.

SOURCE: Intel Corporation, *Embedded Microprocessors* (Santa Clara, Calif.: Author, 1994), pp. 1-11.

Control Unit, Instruction Register and Decoder, Register Array, Interrupt Control, and Serial I/O Control. We will discuss the first four units below; the last two will be discussed later in the book.

THE ALU

The arithmetic/logic unit performs the computing functions; it includes the accumulator, the temporary register, the arithmetic and logic circuits, and five flags. The temporary register is used to hold data during an arithmetic/logic operation. The result is stored in the accumulator, and the flags (flip-flops) are set or reset according to the result of the operation.

The flags are affected by the arithmetic and logic operations in the ALU. In most of these operations, the result is stored in the accumulator. Therefore, the flags generally reflect data conditions in the accumulator—with some exceptions. The descriptions and conditions of the flags are as follows:

- S—Sign flag:** After the execution of an arithmetic or logic operation, if bit D₇ of the result (usually in the accumulator) is 1, the Sign flag is set. This flag is used with signed numbers. In a given byte, if D₇ is 1, the number will be viewed as a negative number; if it is 0, the number will be considered positive. In arithmetic operations with signed numbers, bit D₇ is reserved for indicating the sign, and the remaining seven bits are used to represent the magnitude of a number. However, this flag is irrelevant for the operations of unsigned numbers. Therefore, for unsigned numbers, even if bit D₇ of a result is 1 and the flag is set, it does not mean the result is negative. (See Appendix A2 for a discussion of signed numbers.)
- Z—Zero flag:** The Zero flag is set if the ALU operation results in 0, and the flag is reset if the result is not 0. This flag is modified by the results in the accumulator as well as in the other registers.
- AC—Auxiliary Carry flag:** In an arithmetic operation, when a carry is generated by digit D₃ and passed on to digit D₄, the AC flag is set. The flag is used only internally for BCD (binary-coded decimal) operations and is not available for the programmer to change the sequence of a program with a jump instruction.
- P—Parity flag:** After an arithmetic or logical operation, if the result has an even number of 1s, the flag is set. If it has an odd number of 1s, the flag is reset. (For example, the data byte 0000 0011 has even parity even if the magnitude of the number is odd.)
- CY—Carry flag:** If an arithmetic operation results in a carry, the Carry flag is set; otherwise it is reset. The Carry flag also serves as a borrow flag for subtraction.

The bit positions reserved for these flags in the flag register are as follows:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
S	Z		AC		P		CY

Among the five flags, the AC flag is used internally for BCD arithmetic; the instruction set does not include any conditional jump instructions based on the AC flag. Of the remaining four flags, the Z and CY flags are those most commonly used.

TIMING AND CONTROL UNIT

This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communication between the microprocessor and peripherals. The control signals are similar to a sync pulse in an oscilloscope. The RD and WR signals are sync pulses indicating the availability of data on the data bus.

INSTRUCTION REGISTER AND DECODER

The instruction register and the decoder are part of the ALU. When an instruction is fetched from memory, it is loaded in the instruction register. The decoder decodes the instruction and establishes the sequence of events to follow. The instruction register is not programmable and cannot be accessed through any instruction.

REGISTER ARRAY

The programmable registers were discussed in the last chapter. Two additional registers, called temporary registers W and Z, are included in the register array. These registers are used to hold 8-bit data during the execution of some instructions. However, because they are used internally, they are not available to the programmer.

4.1.6 Decoding and Executing an Instruction

Decoding and executing an instruction after it has been fetched can be illustrated with the example from Section 4.12.

**Example
4.2**

Assume that the accumulator contains data byte 82H, and the instruction MOV C,A (4FH) is fetched. List the steps in decoding and executing the instruction.

Solution

This example is similar to the example in Section 4.12, except that the contents of the accumulator are specified. To decode and execute the instruction, the following steps are performed.

The microprocessor:

1. Places the contents of the data bus (4FH) in the instruction register (Figure 4.8) and decodes the instruction.
2. Transfers the contents of the accumulator (82H) to the temporary register in the ALU.
3. Transfers the contents of the temporary register to register C.

4.1.7 Review of Important Concepts

1. The 8085 microprocessor has a multiplexed bus AD₇–AD₀ used as the lower-order address bus and the data bus.
2. The bus AD₇–AD₀ can be demultiplexed by using a latch and the ALE signal.
3. The 8085 has a status signal IO/M and two control signals RD and WR. By ANDing these signals, four control signals can be generated: MEMR, MEMW, IOR, and IOW.

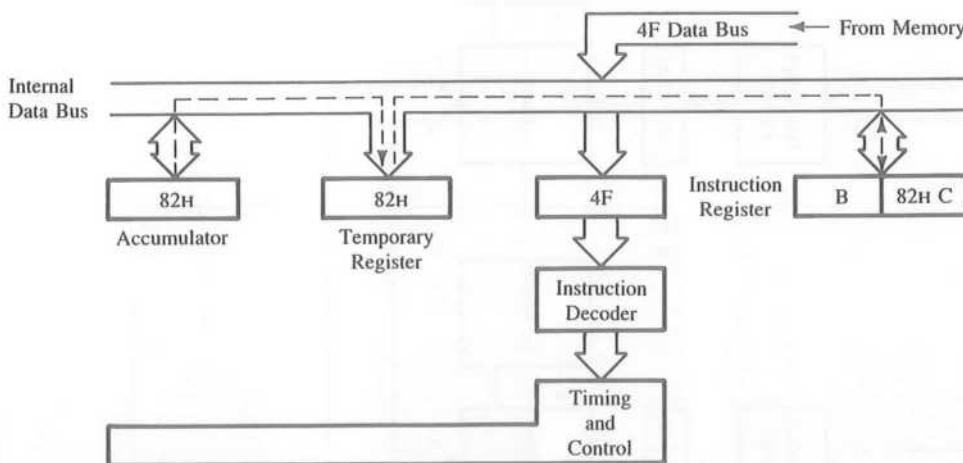


FIGURE 4.8
Instruction Decoding and Execution

4. The 8085 MPU

- transfers data from memory locations to the microprocessor by using the control signal **Memory Read (MEMR—active low)**. This is also called **reading from memory**. The term *data* refers to any byte that is placed on the data bus; the byte can be an instruction code, data, or an address.
- transfers data from the microprocessor to memory by using the control signal **Memory Write (MEMW—active low)**. This is also called **writing into memory**.
- accepts data from input devices by using the control signal **I/O Read (IOR—active low)**. This is also known as **reading from an input port**.
- sends data to output devices by using the control signal **I/O Write (IOW—active low)**. This is also known as **writing to an output port**.

5. To execute an instruction, the MPU

- places the memory address of the instruction on the address bus.
- indicates the operation status on the status lines.
- sends the MEMR control signal to enable the memory, fetches the instruction byte, and places it in the instruction decoder.
- executes the instruction.

EXAMPLE OF AN 8085-BASED MICROCOMPUTER

4.2

A general microcomputer system was illustrated in Figure 3.15, in the last chapter. After our discussion of the 8085 microprocessor and the interfacing devices, we can expand the system to include more details, as shown in Figure 4.9. The system includes interfacing devices such as buffers, decoders, and latches.

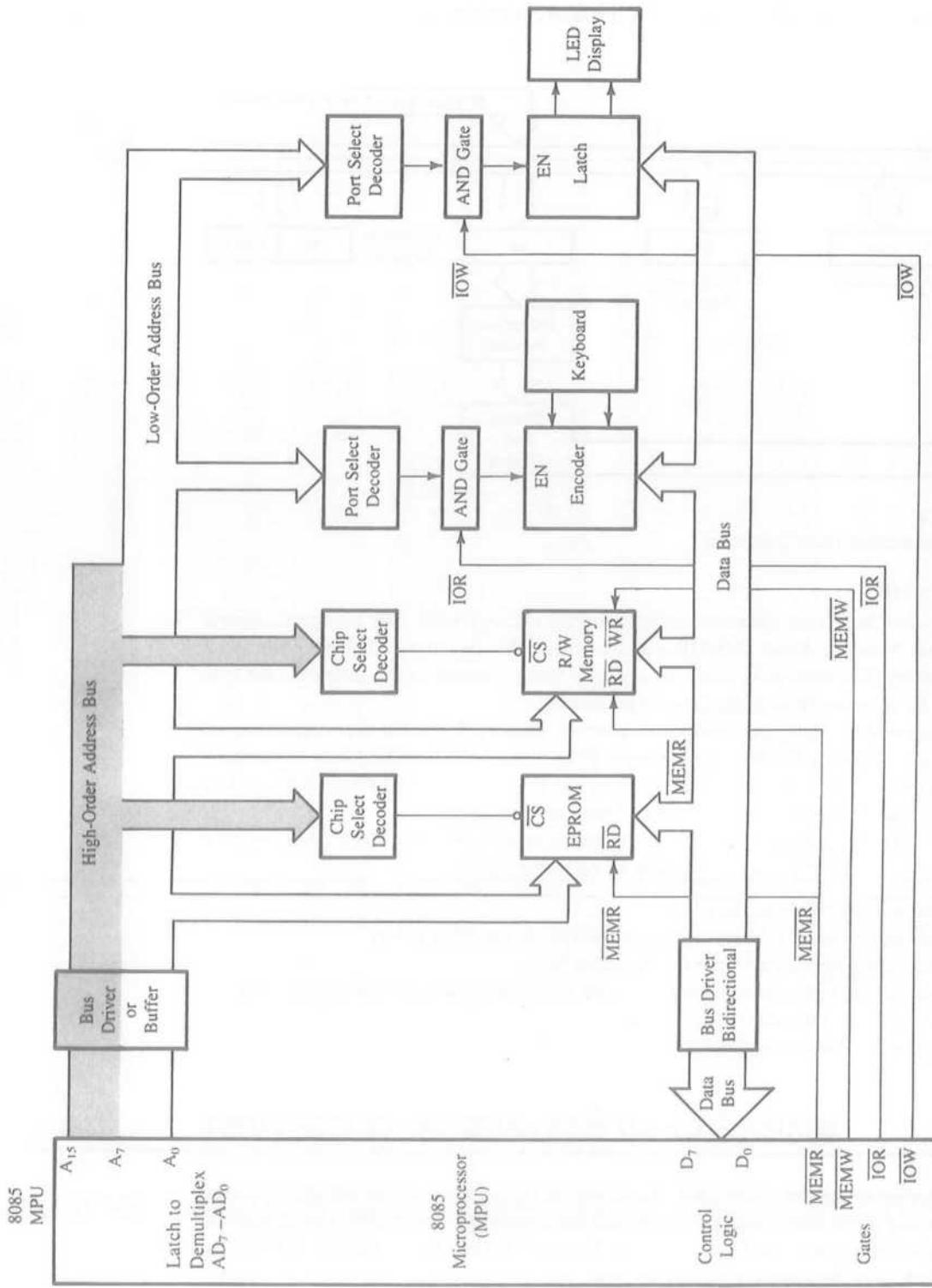


FIGURE 4.9

8085 Single-Board Microcomputer System

NOTE: The bus AD₇-AD₀ is demultiplexed using a latch and memory, and I/O control signals are generated using gates. Figure 4.6 is included in the MPU.

The 8085 MPU module (Figure 4.9) includes devices such as the 8085 microprocessor, an octal latch, and logic gates, as shown previously in Figure 4.6. The octal latch demultiplexes the bus AD_7-AD_0 using the signal ALE, and the logic gates generate the necessary control signals. Figure 4.9 shows the demultiplexed address bus, the data bus, and the four active low control signals: MEMR, MEMW, IOR, and IOW. In addition, to increase the driving capacity of the buses, a unidirectional bus driver is used for the address bus and a bidirectional bus driver is used for the data bus. Now we can examine various operations the 8085 microprocessor performs in terms of machine cycles and T-states.

4.2.1 The 8085 Machine Cycles and Bus Timings

The 8085 microprocessor is designed to execute 74 different instruction types. Each instruction has two parts: operation code, known as opcode, and operand. The opcode is a command such as Add, and the operand is an object to be operated on, such as a byte or the contents of a register. Some instructions are 1-byte instructions and some are multi-byte instructions. To execute an instruction, the 8085 needs to perform various operations such as Memory Read/Write and I/O Read/Write. However, there is no direct relationship between the number of bytes of an instruction and the number of operations the 8085 has to perform (this will be clarified later). In preceding sections, numerous 8085 signals and their functions were described. Now we need to examine these signals in conjunction with execution of individual instructions and their operations. This task may appear overwhelming at the beginning; fortunately, all instructions are divided into a few basic machine cycles and these machine cycles are divided into precise system clock periods.

Basically, the microprocessor external communication functions can be divided into three categories:

1. Memory Read and Write
2. I/O Read and Write
3. Request Acknowledge

These functions are further divided into various operations (machine cycles), as shown in Table 4.1. Each instruction consists of one or more of these machine cycles, and each machine cycle is divided into T-states.

In this section, we will focus on the first three operations listed in Table 4.1—Opcode Fetch, Memory Read, and Memory Write—and examine the signals on various buses in relation to the system clock. In the next section, we will use these timing diagrams to interface memory with the 8085 microprocessor. Similarly, we will discuss timings of other machine cycles in later chapters in the context of their applications. For example, I/O Read/Write machine cycles will be discussed in Chapter 5 on I/O interfacing, and Interrupt Acknowledge will be discussed in Chapter 12, “Interrupts.”

4.2.2 Opcode Fetch Machine Cycle

The first operation in any instruction is Opcode Fetch. The microprocessor needs to get (fetch) this machine code from the memory register where it is stored before the microprocessor can begin to execute the instruction.

We discussed this operation in Example 4.1. Figure 4.2 shows how the 8085 fetches the machine code, using the address and the data buses and the control signal. Figure 4.3 shows the timing of the Opcode Fetch machine cycle in relation to the system's clock. In this operation, the processor reads a machine code (4FH) from memory. However, to differentiate an opcode from a data byte or an address, this machine cycle is identified as the Opcode Fetch cycle by the status signals ($\overline{IO/M} = 0$, $S_1 = 1$, $S_0 = 1$); the active low $\overline{IO/M}$ signal indicates that it is a memory operation, and S_1 and S_0 being high indicate that it is an Opcode Fetch cycle.

This Opcode Fetch cycle is called the M_1 cycle and has four T-states. The 8085 uses the first three states T_1-T_3 to fetch the code and T_4 to decode and execute the opcode. In the 8085 instruction set, some instructions have opcodes with six T-states. When we study the example of the Memory Read machine cycle, discussed in the next section, we may find that these two operations (Opcode Fetch and Memory/Read) are almost identical except that the Memory Read Cycle has three T-states.

4.2.3 Memory Read Machine Cycle

To illustrate the Memory Read machine cycle, we need to examine the execution of a 2-byte or a 3-byte instruction because in a 1-byte instruction the machine code is an opcode; therefore, the operation is always an Opcode Fetch. The execution of a 2-byte instruction is illustrated in the next example.

Example 4.3

Two machine codes—0011 1110 (3EH) and 0011 0010 (32H)—are stored in memory locations 2000H and 2001H, respectively, as shown below. The first machine code (3EH) represents the opcode to load a data byte in the accumulator, and the second code (32H) represents the data byte to be loaded in the accumulator. Illustrate the bus timings as these machine codes are executed. Calculate the time required to execute the Opcode Fetch and the Memory Read cycles and the entire instruction cycle if the clock frequency is 2 MHz.

Memory Location	Machine Code	Instruction
2000H	0 0 1 1 1 1 1 0	\rightarrow 3EH
2001H	0 0 1 1 0 0 1 0	\rightarrow 32H

Solution

This instruction consists of two bytes; the first is the opcode and the second is the data byte. The 8085 needs to read these bytes first from memory and thus requires at least two machine cycles. The first machine cycle is Opcode Fetch and the second machine cycle is Memory Read, as shown in Figure 4.10; this instruction requires seven T-states for these two machine cycles. The timings of the machine cycles are described in the following paragraphs.

1. The first machine cycle M_1 (Opcode Fetch) is identical in bus timings with the machine cycle illustrated in Example 4.1, except for the bus contents.

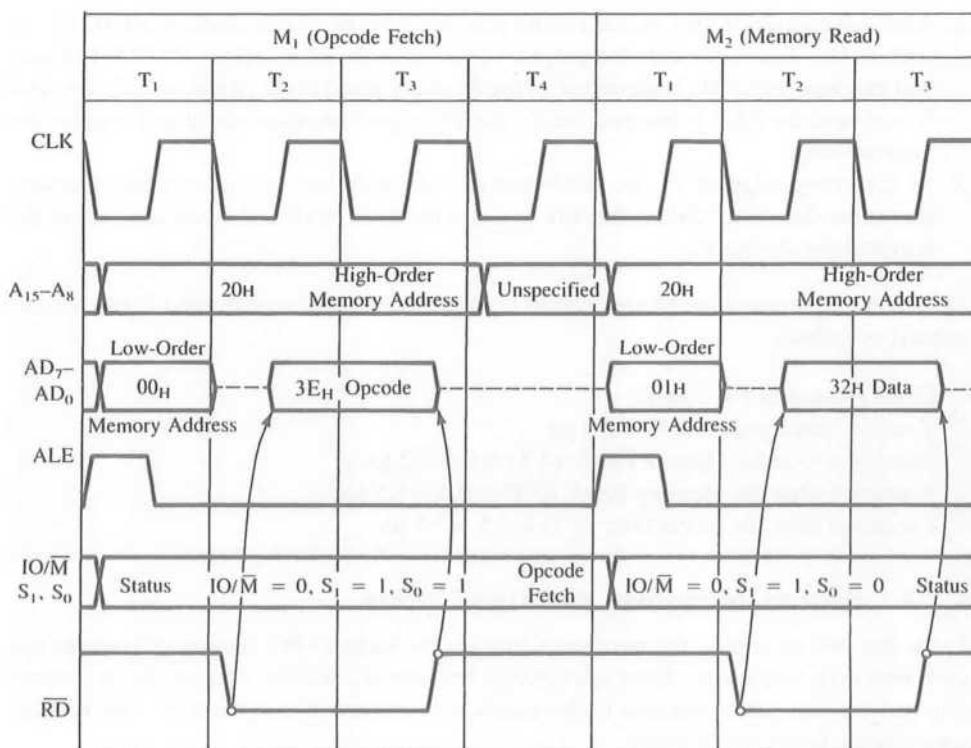


FIGURE 4.10
8085 Timing for Execution of the Instruction MVI A,32H

At T_1 , the microprocessor identifies that it is an Opcode Fetch cycle by placing 011 on the status signals* ($IO/M = 0$, $S_1 = 1$ and $S_0 = 1$). It places the memory address (2000H) from the program counter on the address bus, 20H on $A_{15}-A_8$, and 00H on AD_7-AD_0 and increments the program counter to 2001H to point to the next machine code. The ALE signal goes high during T_1 , which is used to latch the low-order address 00H from the bus AD_7-AD_0 . At T_2 , the 8085 asserts the RD control signal, which enables the memory, and the memory places the byte 3EH from location 2000H on the data bus. Then the 8085 places the opcode in the instruction register and disables the RD signal. The fetch cycle is completed in state T_3 . During T_4 , the 8085 decodes the opcode and finds out that a second byte needs to be read. After the T_3 state, the contents of the bus $A_{15}-A_8$ are unknown, and the data bus AD_7-AD_0 goes into high impedance.

*The status signals S_1 and S_0 can be used to differentiate between various machine cycles. However, they are rarely needed; the necessary control signals can be generated by using IO/M , RD , and WR .

2. After completion of the Opcode Fetch cycle, the 8085 places the address 2001H on the address bus and increments the program counter to the next address 2002H. The second machine cycle M_2 is identified as the Memory Read cycle ($IO/M = 0$, $S_1 = 1$, and $S_0 = 0$) and the ALE is asserted. At T_2 , the RD signal becomes active and enables the memory chip.
3. At the rising edge of T_2 , the 8085 activates the data bus as an input bus, memory places the data byte 32H on the data bus, and the 8085 reads and stores the byte in the accumulator during T_3 .

The execution times of the Memory Read machine cycle and the instruction cycle are calculated as follows:

- Clock frequency $f = 2$ MHz
- T-state = clock period $(1/f) = 0.5 \mu s$
- Execution time for Opcode Fetch: $(4 T) \times 0.5 = 2 \mu s$
- Execution time for Memory Read: $(3 T) \times 0.5 = 1.5 \mu s$
- Execution time for Instruction: $(7 T) \times 0.5 = 3.5 \mu s$

4.2.4 How to Recognize Machine Cycles

In the last two examples, the number of bytes is the same as the number of machine cycles. However, there is no direct relationship between the number of bytes in an instruction and the number of machine cycles required to execute that instruction. This is illustrated in the following example.

Example 4.4

Explain the machine cycles of the following 3-byte instruction when it is executed.

Opcode	Operand	Bytes	Machine Cycles	T-States	Operation
STA	2065H	3	4	13	This instruction stores (writes) the contents of the accumulator in memory location 2065H

The machine codes are stored in memory locations 2010H, 2011H, and 2012H as follows: the 16-bit address of the operand must be entered in reverse order, the low-order byte first, followed by the high-order byte.

Memory Address	Machine Code		
2010	0011	0010 → 32H	Opcode
2011	0110	0101 → 65H	Low-order address
2012	0010	0000 → 20H	High-order address

This is a 3-byte instruction; however, it has four machine cycles with 13 T-states. The first operation in the execution of an instruction must be an Opcode Fetch. The 8085 requires three T-states for each subsequent operation; thus, nine T-states are required for the remaining machine cycles. Therefore, the Opcode Fetch in this instruction must be a four T-state machine cycle, the same as the one described in the previous example. The two machine cycles following the Opcode Fetch must be Memory Read machine cycles because the microprocessor must read all the machine codes (three bytes) before it can execute the instruction. Now let us examine what the instruction does. It stores (writes) the contents of the accumulator in memory location 2065H; therefore, the last machine cycle must be Memory Write. The execution steps are as follows:

Solution

1. In the first machine cycle, the 8085 places the address 2010H on the address bus and fetches the opcode 32H.
2. The second machine cycle is Memory Read. The processor places the address 2011H and gets the low-order byte 65H.
3. The third machine cycle is also Memory Read; the 8085 gets the high-order byte 20H from memory location 2012H.
4. The last machine cycle is Memory Write. The 8085 places the address 2065H on the address bus, identifies the operation as Memory Write ($\overline{\text{IO/M}} = 0$, $S_1 = 0$, and $S_0 = 1$). It places the contents of the accumulator on the data bus $\text{AD}_7\text{--}\text{AD}_0$ and asserts the WR signal. During the last T-state, the contents of the data bus are placed in memory location 2065H.

4.2.5 Review of Important Concepts

1. In each instruction cycle, the first operation is always Opcode Fetch. This cycle can be of four to six T-states duration.
2. The Memory Read cycle requires three T-states and is in many ways similar to the Opcode Fetch cycle. Both use the same control signal (RD) and read contents from memory. However, the Opcode Fetch reads opcodes and the Memory Read reads 8-bit data or address; these two machine cycles are differentiated by the status signals.
3. When the status signal IO/M is active low, the 8085 indicates that it is a memory-related operation, and the control signal RD suggests that it is a Read operation. Both signals are necessary to read from memory; the MEMR (Memory Read) control signal is generated by ANDing these two signals (see Section 4.14). The other status signals S_1 and S_0 are generally not needed in simple systems.
4. In the Memory Write cycle, the 8085 writes (stores) data in memory, using the control signal WR and the status signal IO/M.
5. In the Memory Read cycle, the 8085 asserts the RD signal to enable memory, and then the addressed memory places data on the data bus; on the other hand, in the Memory Write cycle, the 8085 places the data byte on the data bus and then asserts the WR signal to write into the addressed memory.

6. The Memory Read and Write cycles consist of three T-states. The Memory Read and Write cycles will not be asserted simultaneously—the microprocessor cannot read and write at the same time.

4.3

MEMORY INTERFACING

Memory is an integral part of a microcomputer system, and in this chapter, our focus will be on how to interface a memory chip with the microprocessor. While executing a program, the microprocessor needs to access memory quite frequently to read instruction codes and data stored in memory; the interfacing circuit enables that access. Memory has certain signal requirements to write into and read from its registers. Similarly, the microprocessor initiates a set of signals when it wants to read from and write into memory. The interfacing process involves designing a circuit that will match the memory requirements with the microprocessor signals.

In the following sections, we will examine memory structure and its requirements and the 8085 Memory Read and Write machine cycles. Then we will derive the basic steps necessary to interface memory with the 8085. In the last chapter, we discussed a hypothetical memory chip and the concepts in addressing. In this chapter, we will illustrate memory interfacing, using memory chips such as 2732 EPROM and 6116 static R/W memory, and will discuss address decoding and memory addresses.

4.3.1 Memory Structure and Its Requirements

As discussed in Chapter 3, **Read/Write memory (R/WM)** is a group of registers to store binary information. Figure 4.11(a) shows a typical R/W memory chip; it has 2048 registers and each register can store eight bits indicated by **eight input and eight output data lines**.* The chip has 11 address lines $A_{10}-A_0$, one **Chip Select (CS)**, and two control lines: **Read (RD)** to enable the output buffer and **Write (WR)** to enable the input buffer. Figure 4.11(a) also shows the internal decoder to decode the address lines. Figure 4.11(b) shows the logic diagram of a typical EPROM (Erasable Programmable Read-Only Memory) with 4096 (4K) registers. It has 12 address lines $A_{11}-A_0$, one Chip Select (CS), and one **Read control signal**. This chip must be programmed (written into) before it can be used as a read-only memory. Figure 4.11(b) also shows a quartz window on the chip that is used to expose the chip to ultraviolet rays for erasing the program. Once the chip is programmed, the window is covered with opaque tape to avoid accidental erasing. For interfacing the R/W memory, Figure 4.11(a), and the EPROM, Figure 4.11(b), the process is similar; the only difference is that the EPROM does not require the WR signal.

*In a typical memory chip, the input and output data lines are not shown separately, but are shown as a group of eight data lines.

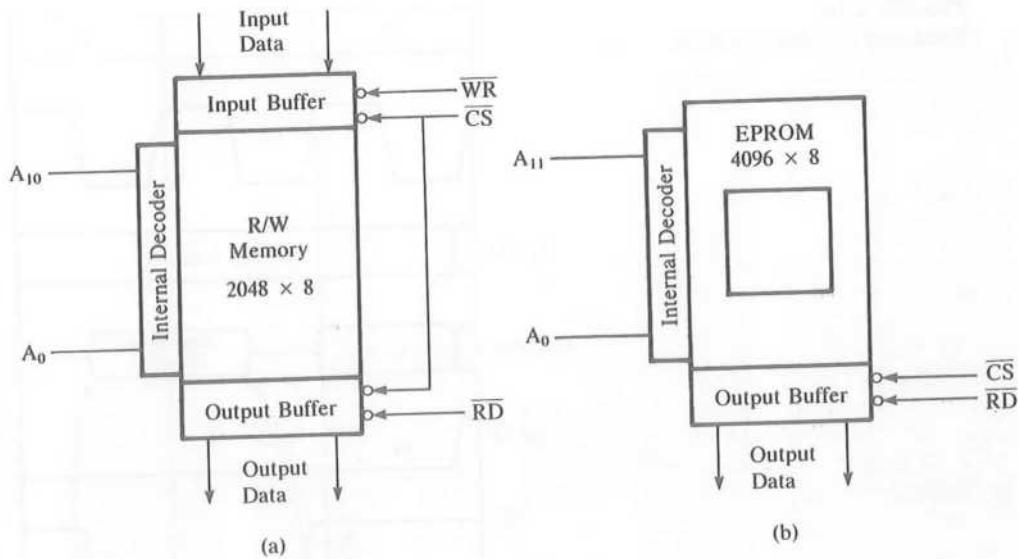


FIGURE 4.11
Typical Memory Chips: R/W Static Memory (a) and EPROM (b)

4.3.2 Basic Concepts in Memory Interfacing

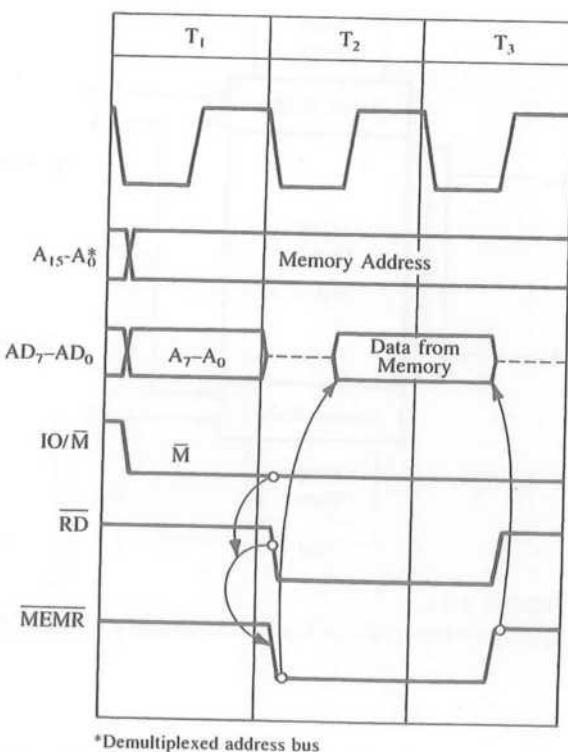
The primary function of memory interfacing is that the microprocessor should be able to read from and write into a given register of a memory chip. Recall from Chapter 3 that to perform these operations, the microprocessor should

1. be able to select the chip.
 2. identify the register.
 3. enable the appropriate buffer.

Let us examine the timing diagram of the Memory Read operation (Figure 4.12) to understand how the 8085 can read from memory. Figure 4.12 is the M₂ cycle of Figure 4.10 except that the address bus is demultiplexed. We could also use the M₁ cycle to illustrate these interfacing concepts.

1. The 8085 places a 16-bit address on the address bus, and with this address only one register should be selected. For the memory chip in Figure 4.11(a), only 11 address lines are required to identify 2048 registers. Therefore, we can connect the low-order address lines $A_{10}-A_0$ of the 8085 address bus to the memory chip. The internal decoder of the memory chip will identify and select the register for the EPROM, Figure 4.11(a).

FIGURE 4.12
Timing of the Memory Read Cycle

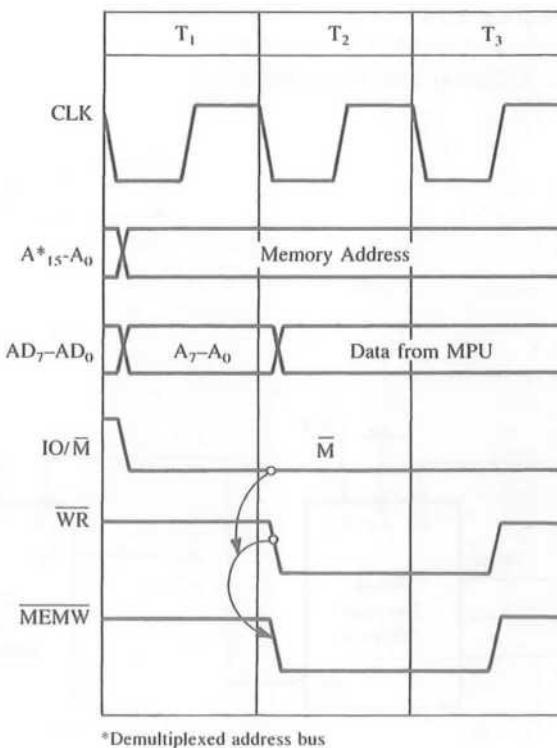


2. The remaining 8085 address lines ($A_{15}-A_{11}$) should be decoded to generate a Chip Select (CS) signal unique to that combination of address logic (illustrated in Examples 4.3 and 4.4).
3. The 8085 provides two signals— $\overline{IO/M}$ and \overline{RD} —to indicate that it is a memory read operation. The $\overline{IO/M}$ and \overline{RD} can be combined to generate the \overline{MEMR} (Memory Read) control signal that can be used to enable the output buffer by connecting to the memory signal RD .
4. Figure 4.12 also shows that memory places the data byte from the addressed register during T_2 , and that is read by the microprocessor before the end of T_3 .

To write into a register, the microprocessor performs similar steps as it reads from a register. Figure 4.13 shows the Memory Write cycle. In the Write operation, the 8085 places the address and data and asserts the $\overline{IO/M}$ signal. After allowing sufficient time for data to become stable, it asserts the Write (WR) signal. The $\overline{IO/M}$ and WR signals can be combined to generate the \overline{MEMW} control signal that enables the input buffer of the memory chip and stores the byte in the selected memory register.

To interface memory with the microprocessor, we can summarize the above steps as follows:

FIGURE 4.13
Timing of the Memory Write Cycle



1. Connect the required address lines of the address bus to the address lines of the memory chip.
2. Decode the remaining address lines of the address bus to generate the Chip Select signal, as discussed in the next section (4.3.3), and connect the signal to select the chip.
3. Generate control signals MEMR and MEMW by combining RD and WR signals with IO/M, and use them to enable appropriate buffers.

4.3.3 Address Decoding

The process of address decoding should result in identifying a register for a given address. We should be able to generate a unique pulse for a given address. For example, in Figure 4.11(b), 12 address lines ($A_{11}-A_0$) are connected to the memory chip, and the remaining four address lines ($A_{15}-A_{12}$) of the 8085 microprocessor must be decoded. Figure 4.14 shows two methods of decoding these lines: one by using a NAND gate and the other by using a 3-to-8 decoder. The output of the NAND goes active and selects the chip only when all address lines $A_{15}-A_{12}$ are at logic 1. We can obtain the same result by using O_7 of the 3-to-8 decoder, which is capable of decoding eight different input addresses. In the decoder circuit, three input lines can have eight different logic combinations from 000 to 111; each input combination can be identified by the corresponding output line if Enable

FIGURE 4.14
Address Decoding Using NAND Gate (a) and 3-to-8 Decoder (b)

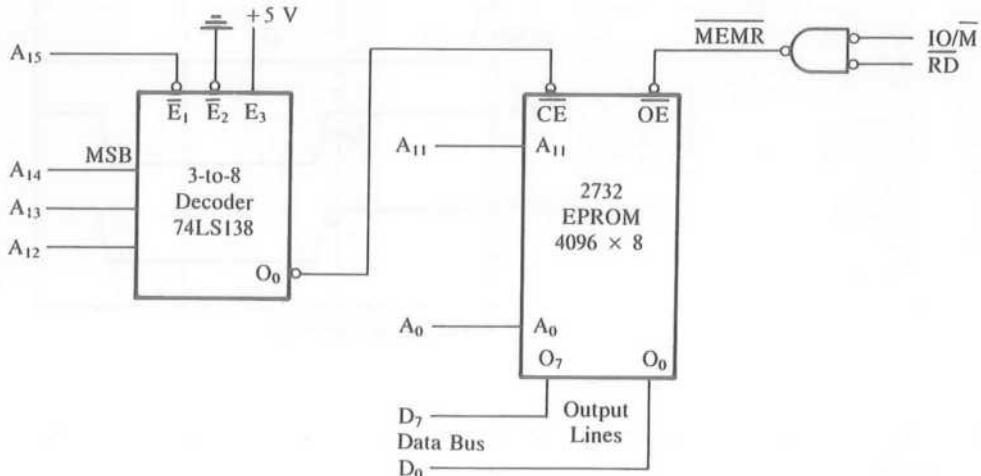
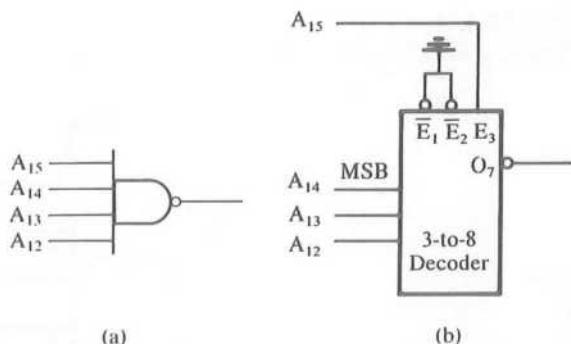


FIGURE 4.15
Interfacing the 2732 EPROM

lines are active. In this circuit, the Enable lines \bar{E}_1 and \bar{E}_2 are enabled by grounding, and A_{15} must be at logic 1 to enable E_3 . We will use this address decoding scheme to interface a 4K EPROM and a 2K R/W memory as illustrated in the next two examples.

4.3.4 Interfacing Circuit

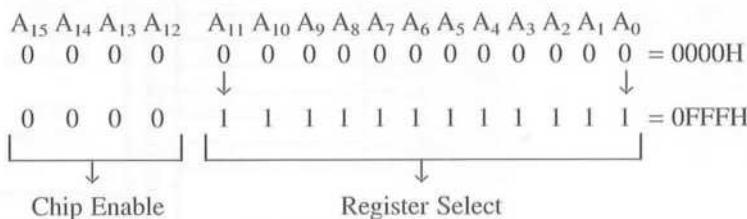
Figure 4.15 shows an interfacing circuit using a 3-to-8 decoder to interface the 2732 EPROM memory chip. It is assumed here that the chip has already been programmed, and we will analyze the interfacing circuit in terms of the same three steps outlined previously:

Step 1: The 8085 address lines A_{11} – A_0 are connected to pins A_{11} – A_0 of the memory chip to address 4096 registers.

- Step 2:** The decoder is used to decode four address lines $A_{15}-A_{12}$. The output O_0 of the decoder is connected to Chip Enable (CE). The CE is asserted only when the address on $A_{15}-A_{12}$ is 0000; A_{15} (low) enables the decoder and the input 000 asserts the output O_0 .
- Step 3:** For this EPROM, we need one control signal: Memory Read ($\overline{\text{MEMR}}$), active low. The $\overline{\text{MEMR}}$ is connected to $\overline{\text{OE}}$ to enable the output buffer; $\overline{\text{OE}}$ is the same as RD in Figure 4.11.

4.35 Address Decoding and Memory Addresses

We can obtain the address range of this memory chip by analyzing the possible logic levels on the 16 address lines. The logic levels on the address lines $A_{15}-A_{12}$ must be 0000 to assert the Chip Enable, and the address lines $A_{11}-A_0$ can assume any combinations from all 0s to all 1s. Therefore, the memory address of this chip ranges from 0000H to 0FFFH, as shown below.



We can verify the memory address range in terms of our analogy of page and line numbers, as discussed in Chapter 3, Section 3.22. The chip's 4096 bytes of memory can be viewed as 16 pages with 256 lines each. The high-order Hex digits range from 00 to 0F, indicating 16 pages—0000H to 00FFH and 0100H to 01FFH, for example.

Now, to examine how an address is decoded and how the microprocessor reads from this memory, let us assume that the 8085 places the address 0FFFH on the address bus. The address 0000 (0H) goes to the decoder, and the output line O_0 of the decoder selects the chip. The remaining address FFFH goes on the address lines of the chip, and the internal decoder of the chip decodes the address and selects the register FFFH. Thus, the address 0FFFH selects the register as shown in Figure 4.16. When the 8085 asserts the RD signal, the output buffer is enabled and the contents of the register 0FFFH are placed on the data bus for the processor to read.

Analyze the interfacing circuit in Figure 4.17 and find its memory address range.

Example
4.5

Figure 4.17 shows the interfacing of the 6116 memory chip with 2048 (2K) registers. The memory chip requires 11 address lines ($A_{10}-A_0$) to decode 2048 registers. The remaining address lines $A_{15}-A_{11}$ are connected to the decoder. However, in this circuit, the decoder is enabled by the IO/M signal in addition to the address lines A_{15} and A_{14} , and the RD and

Solution

FIGURE 4.16
Address Decoding and Reading from Memory

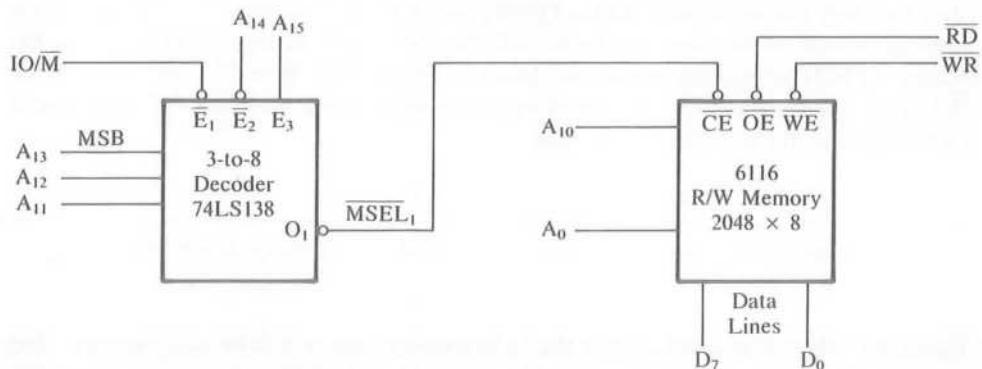
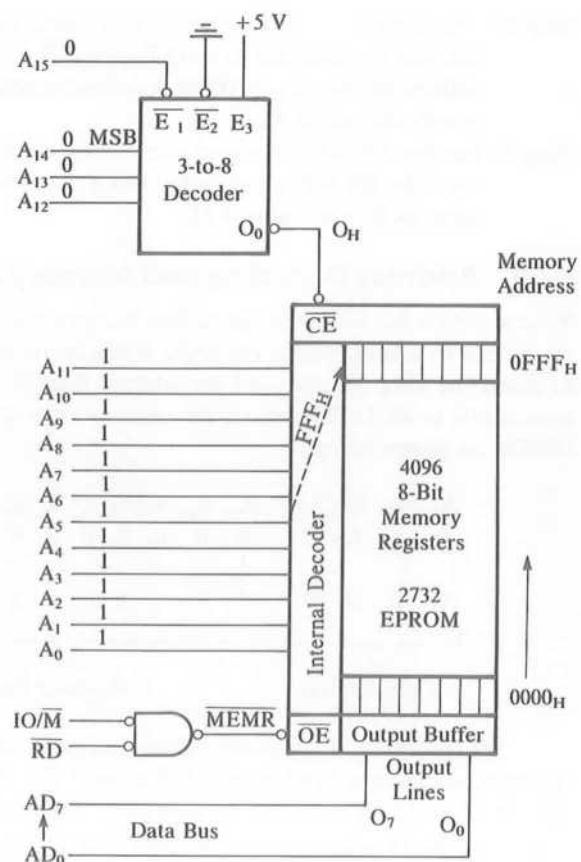
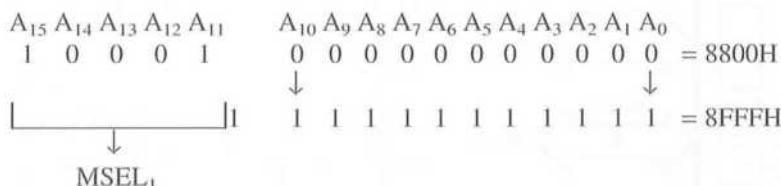


FIGURE 4.17
Interfacing R/W Memory

WR signals of the 8085 are directly connected to the memory chip. The signals MEMR and MEMW need not be generated separately; thus, this technique saves two gates. The memory address of this chip ranges from 8800H to 8FFFH, as shown below.



The output line O_1 of the decoder is connected to \overline{CE} of the memory chip, and it is identified as $MSEL_1$ because it is asserted only when IO/M is low.

The address lines A₁₅ (high) and A₁₄ (low) enable the decoder, and the input lines to the decoder—A₁₃, A₁₂, and A₁₁ (001)—activate the output O₁ (MSEL₁) to select the memory chip. The address lines A₁₀–A₀ can assume any logic combination between all 0s to all 1s, as shown above.

INTERFACING THE 8155 MEMORY SEGMENT

4.4

The SDK-85 is a single-board microcomputer designed by Intel and widely used in college laboratories. The system is designed using the 8085 microprocessor and specially compatible devices, such as the 8155/8156.

The 8155/8156 includes multiple devices on the same chip. The 8155 has 256 bytes of R/W memory, two programmable I/O ports, and a timer. The 8156 is identical to the 8155, except that its Chip Enable (CE) signal is active high. The programmable I/O ports of this device are discussed in Chapter 14. The memory section of this chip and its memory addresses in the SDK-85 system will now be discussed.

4.4.1 Interfacing the 8155 Memory Section

Figure 4.18(a) shows the block diagram of the 8155 memory section. It has eight address lines, one CE (Chip Enable) line, and five lines compatible with the control and status signals of the 8085: IO/M, ALE, RD, WR, and RESET. These control and status lines are not found in the general-purpose memory devices shown in the previous section. These lines eliminate the need for external demultiplexing of the bus AD_7 – AD_0 and for generating separate control signals for memory and I/O.

Figure 4.18(b) also shows the internal structure of the 8155 memory section. The memory section includes 256×8 memory locations and an internal latch to demultiplex the bus lines AD_7 – AD_0 . The memory section also requires a Chip Enable (CE) signal and the Memory Write (MEMW) and Memory Read (MEMR) control signals, generated internally by combining the IO/M, WR, and RD signals.

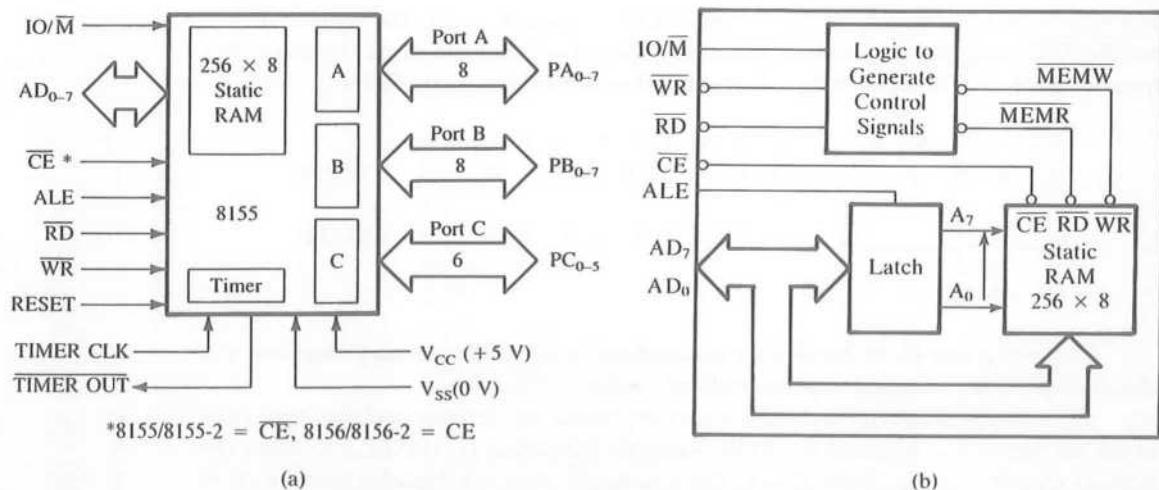


FIGURE 4.18

The 8155 Memory Section: The Block Diagram (a) and the Internal Structure (b)

SOURCE: (a) Intel Corporation, *Embedded Microprocessors* (Santa Clara, Calif.: Author, 1994), pp. 1-31.

FIGURE 4.19

Interfacing the 8155 Memory Schematic from the SDK-85 System

SOURCE: Intel Corporation, *SDK-85 User's Manual* (Santa Clara, Calif.: Author, 1978), Appendix B.

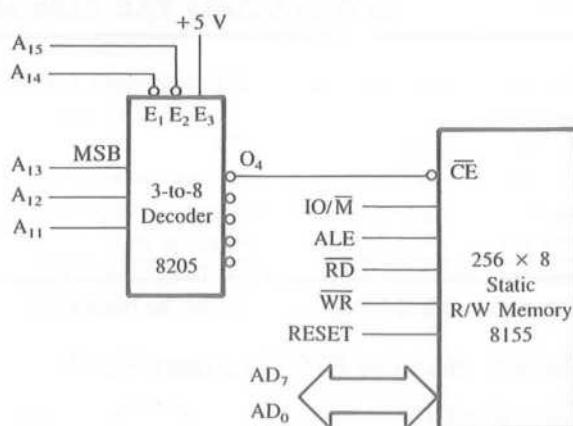


Figure 4.19 shows a schematic of the SDK-85 system of interfacing the 8155 memory section with the 8085. The 8205, a 3-to-8 decoder (identical to the 74LS138 decoder), decodes the address lines A₁₅–A₁₁, and the output line O₄ of the decoder enables the memory chip. The control and the status signals from the 8085 are connected directly to the respective signals on the memory chip. Similarly, the bus lines AD₇–AD₀ are also connected directly to the memory chip to address any one of the 256 memory locations.

Explain the decoding logic and the memory address range of the 8155 shown in Figure 4.19.

Example
4.6

The interfacing logic shows the 3-to-8 decoder; its output line 4 (O_4) is used to select the 8155. The address lines A_{11} to A_{13} are connected as input to the decoder, and the lines A_{15} and A_{14} are used as active low Enable lines. The third Enable line (active high) is permanently enabled by tying it to +5 V. The address lines A_{10} , A_9 , and A_8 are not connected; thus, they are left as don't care lines. The output line O_4 of the decoder goes low when the address lines have the following address:

Solution

$$\begin{array}{ccccccccc} A_{15} & A_{14} & A_{13} & A_{12} & A_{11} & A_{10} & A_9 & A_8 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} = 20H \text{ (assuming the don't care lines are at logic 0)}$$

The address lines AD_7 – AD_0 can assume any combination of logic levels from all 0s to all 1s. Thus, the memory addresses of the 8155 memory will range from 2000H to 20FFH as follows.

$A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8$	$A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0$
$0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$	$0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 = 2000H$
$\boxed{0 \ 0 \ 1 \ 0 \ 0} \ \downarrow \quad \boxed{0 \ 0 \ 0} \ \downarrow \quad \boxed{1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1} \ \downarrow$	$= 20FFH$
Chip Enable Don't Care Register Select	

In reality, the memory section of this 8155 uses the memory space from 2000H to 27FFH. In Figure 4.19, the address lines A_{10} , A_9 , and A_8 are not connected; thus, they are don't care lines capable of assuming any logic state 0 or 1. Three don't care address lines can be assumed to have any one of the eight combinations from 000 to 111. Thus each combination can generate one set of complete addresses. The address range given by assuming all don't care lines at logic 0 is, by convention, specified as the memory address range of the system or the primary address; the remaining address ranges are known as either foldback memory or mirror memory. In this example, the primary address range is 2000H to 20FFH and the foldback memory range is 2100H to 27FFH as shown in Table 4.3.

The Don't Care column in Table 4.3 shows all the additional logic combinations of the address lines A_{10} – A_8 from 001 to 111, thus generating seven additional address ranges called foldback (or mirror) memory. In reality, these are the same memory registers as 2000H to 20FFH. Attempting to store an instruction in location 2100H (or 2200H or 2700H) is the same as entering the instruction in location 2000H. This results in assigning eight different addresses to the same memory register. To find each address range, we should assume only one combination of the don't care lines at a time; this is particularly important when the don't care lines are not in a particular sequence. (See Problem 34 at the end of the chapter.)

TABLE 4.3

Range of Foldback (Mirror) Memory Addresses for Example 4.6

Chip Enable					Don't Care			Register Select							Hex Address	
A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	2100H
					0	0	1	1	1	1	1	1	1	1	1	21FFH
					0	1	0	0	0	0	0	0	0	0	0	2200H
					0	1	0	1	1	1	1	1	1	1	1	22FFH
					0	1	1	0	0	0	0	0	0	0	0	2300H
					0	1	1	1	1	1	1	1	1	1	1	23FFH
					1	0	0	0	0	0	0	0	0	0	0	2400H
					1	0	0	1	1	1	1	1	1	1	1	24FFH
					1	0	1	0	0	0	0	0	0	0	0	2500H
					1	0	1	1	1	1	1	1	1	1	1	25FFH
					1	1	0	0	0	0	0	0	0	0	0	2600H
					1	1	0	1	1	1	1	1	1	1	1	26FFH
					1	1	1	0	0	0	0	0	0	0	0	2700H
					1	1	1	1	1	1	1	1	1	1	1	27FFH

4.4.2 Absolute vs. Partial Decoding and Multiple Address Ranges

In Figures 4.15 and 4.17, all the high-order address lines were decoded to select the memory chip, and the memory chip is selected only for the specified logic levels on these high-order address lines; no other logic levels can select the chip. This is called absolute decoding, a desirable design practice commonly used in large memory systems.

However, in Figure 4.19, three address lines out of eight (A₁₅–A₈) were not decoded, resulting in multiple addresses; this is called partial decoding. The 8155 chip has 256 memory registers, but it occupies the memory space of 2048 locations, eight times the space of its size. In a small system where the total memory space is not needed, such a technique of partial decoding can be used. The primary advantage of such a technique is in cost saving. In Figure 4.19, we can use the same decoder for multiple size memory chips. This decoder is designed to decode 2K memory chips without generating any multiple address ranges. (See Problem 27.)

4.5

ILLUSTRATIVE EXAMPLE: DESIGNING MEMORY FOR THE MCTS PROJECT

In this section, we will approach the question of memory interfacing from a design point of view. In the previous examples, we analyzed the given schematics of memory interfacing. Now we will design an interfacing circuit for the microprocessor-controlled temperature system (MCTS) described in Chapter 1.

4.5.1 Memory Design: Problem Statement

Given the components as listed, design an interfacing circuit for the MCTS memory to meet the following specifications:

1. 74LS138: 3-to-8 decoder
2. 2732 (4K × 8): EPROM—address range should begin at 0000H and additional 4K memory space should be available for future expansion.
3. 6116 (2K × 8): CMOS R/W memory

4.5.2 Problem Analysis

1. Given one 3-to-8 decoder without any additional gates, the decoder must be enabled using the $\overline{IO/M}$, and the \overline{RD} and \overline{WR} signals must be connected directly to the memory chips as necessary.
2. The 2732 EPROM is a 4K memory chip that requires 12 address lines ($A_{11}-A_0$); therefore, only four high-order address lines ($A_{15}-A_{12}$) must be decoded. The 74LS138 decoder has three input lines and three enable lines (two active low and one active high). However, we have four high-order address lines and one $\overline{IO/M}$ signal. Therefore, three address lines can be connected as inputs, one address line and $\overline{IO/M}$ to the active low enable lines of the decoder, and the remaining enable line of the decoder must be connected high (+ 5V). The EPROM, once programmed, needs only the \overline{RD} signal. Therefore, the \overline{RD} signal from the 8085 must be connected to the Output Enable (\overline{OE}) of the EPROM.
3. To assign the starting address, 0000H, the address lines should have the following logic levels:

$A_{15} A_{14} A_{13} A_{12} \quad A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0$

0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 = 0000H											

The logic levels of $A_{15}-A_{12}$ should be zero. Therefore, A_{15} must be connected to the active low enable signal of the decoder. If the output O_0 of the decoder is used to select the EPROM memory chip, address lines $A_{14}-A_{12}$ will be at logic 0.

4. The 6116 is a 2K R/W memory chip and requires 11 address lines from A_{10} to A_0 . To use the same decoder, address line A_{11} must be left as don't care. There is no specific requirement for the starting address for this memory except that 4K memory space must be left for additional expansion. Therefore, the output O_1 of the decoder must be left for future expansion. We can use O_2 to select the 6116 by connecting it to the \overline{CE} signal. The 8085 \overline{RD} and \overline{WR} signals should be connected to the Output Enable (\overline{OE}) and Write Enable (\overline{WE}) signals, respectively, of the 6116 R/W memory chip.

4.5.3 Circuit Analysis

Figure 4.20 shows the schematic of the interfacing circuit based on the problem analysis.

- When the logic levels of $A_{15}-A_{12}$ are all 0s, and the processor asserts IO/\bar{M} to read from memory, the output O_0 goes active and selects the chip. The address range of the EPROM is as follows:

$$\begin{array}{ll}
 A_{15} A_{14} A_{13} A_{12} & A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 \\
 \\
 \boxed{0 \ 0 \ 0 \ 0} & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 = 0000H \\
 \\
 MSEL_0 & 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 = 0FFFH
 \end{array}$$

- The 6116 R/W memory is selected by the output signal of the decoder O_2 . In this circuit the address line A_{11} is at don't care logic. Assuming A_{11} is at logic 0, the address range is as follows:

$$\begin{array}{ll}
 A_{15} A_{14} A_{13} A_{12} & A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 \\
 \\
 \boxed{0 \ 0 \ 1 \ 0} & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 = 2000H \\
 \\
 MSEL_2 & 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 = 27FFH
 \end{array}$$

Assuming A_{11} is at logic 1, the foldback (or mirror) address range is as follows:

$$A_{15} A_{14} A_{13} A_{12} \quad A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0$$

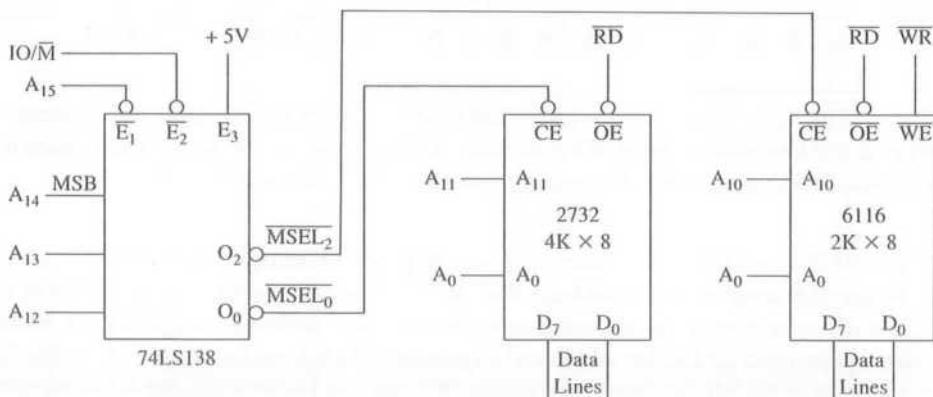


FIGURE 4.20
Schematic of Memory Design for MCTS Project

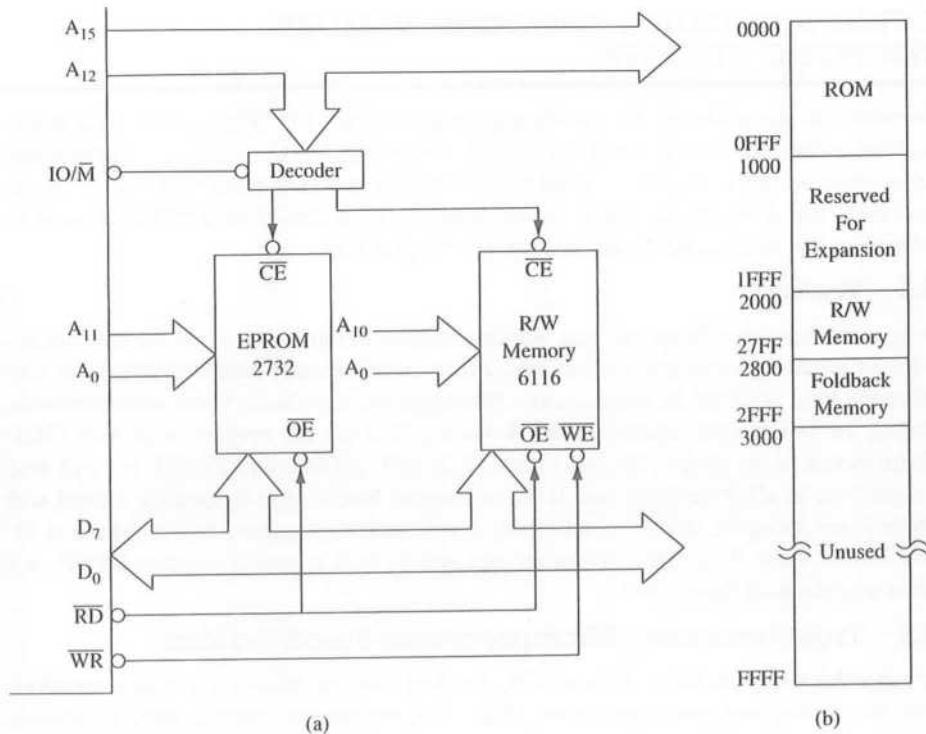


FIGURE 4.21
(a) System Memory for MCTS
(b) System Memory Map

| 0 0 1 0 | 1 0 0 0 0 0 0 0 0 0 = 2800H

MSEL₂ 1 1 1 1 1 1 1 1 1 1 = 2FFFH

4.5.4 MCTS System

We will use this memory design for our MCTS project as shown in the system diagram in Figure 4.21 (a) with the complete memory map as shown in Figure 4.21(b). The memory space from 3000H to FFFFH is unused in this system. The R/W memory will be used to store any temporary data, and EPROM will be used to store a monitor program that will control the system. In this design, we replaced the original ROM memory with the EPROM memory for two reasons: (1) The programming of ROMs needs a specialized masking process. It is done by a chip manufacturer in large quantities. (2) The EPROMs can be programmed in college laboratories, and can be erased and reprogrammed if necessary.

4.6

TESTING AND TROUBLESHOOTING MEMORY INTERFACING CIRCUITS

In Section 4.4, we analyzed the interfacing circuit of the 8155 (Figure 4.19) and determined the address range was 2000H to 20FFH. Assuming that we connected all the wires to an existing working system, we need to test whether we have memory in the given address range, and if we are unable to access memory from 2000H to 20FFH, we need to troubleshoot the interfacing circuit and correct the problem.

4.6.1 Testing

Testing a memory chip in an existing working system is fairly simple. If we have an input device such as a keyboard, we can load a byte at the memory address 2000H and verify that the byte is stored in that location. Similarly, we can check a few more locations including the last memory address, 20FFH. We can also test the multiple addresses (fold-back memory). If we access locations such as 2100H, 2200H, and 2700H, we will find the same byte in all these locations. If we attempt to load a byte in location 2800H and the byte is not accepted, this is an indication that the memory register does not exist at location 2800H. Now if we find that we are not able to load a byte in location 2000H, we need to troubleshoot the circuit.

4.6.2 Troubleshooting Microprocessor-Based Systems

To troubleshoot the circuit in Figure 4.19, the first obvious step is a visual inspection. Check the wiring and pin connections. After this preliminary check, most traditional methods used in checking analog circuits (such as an amplifier) are ineffective because the logic levels on the buses are dynamic; they constantly change depending upon the operation being performed at a given instant by the microprocessor. In troubleshooting analog circuits, a commonly used technique is signal injection, whereby a known signal is injected at the input and the output signal is verified against the expected outcome. To use this concept, we need to generate a constant and identifiable signal and check various points in relation to that signal. We can generate such a signal by asking the processor to execute a continuous loop, called diagnostic routine, as discussed below.

4.6.3 Diagnostic Routine to Generate a Steady Signal

To generate a steady signal, we need to write a continuous loop as shown below.

```
START: MVI A, 55H      ;Load a byte in the accumulator
        STA 2000H      ;Store 55H in memory location 2000H
        JMP START       ;Jump back to beginning and repeat
```

This routine has three instructions. The first instruction loads 55H in the accumulator, and the second instruction stores the byte in location 2000H. The byte 55H has no particular significance. The third instruction is an unconditional Jump instruction that takes the program execution back to the beginning. These instructions are executed continuously. Now we need to examine the machine cycles of these instructions to find an identifiable signal that is repeated at a certain interval. We can analyze the machine cy-

cles in the loop as follows (it will be helpful to have read Chapter 7 to understand the diagnostic routine).

This loop has 30 T-states and nine operations. To execute the loop once, the microprocessor asserts the RD signal eight times (the Opcode Fetch is also a Read operation) and the WR signal once. Assuming the system clock frequency is 3 MHz, the loop is executed in 9.9 μ s, and the WR signal is repeated every 9.9 μ s that can be observed on a scope. If we sync the scope on the WR pulse from the 8085, we can check the output of the decoder and memory control signals WR and RD; some of these signals are shown in Figure 4.22.

When the 8085 asserts the WR signal, the address on A₁₅-A₀ must be 2000H, and the output of the decoder must be asserted low. If it is high, it indicates that the address lines A₁₅-A₁₁ are improperly connected or that the decoder chip is faulty.

If the decoder output is low, it confirms that the decoding circuit is functioning properly. Now if we check the entire data bus in relation to the WR signal, one line at a time, we must read the data 55H. If we check the RD signal, it must be high when the WR is asserted, and we will observe eight RD signals between every two WR signals, as shown in Figure 4.22.

Now we can use the WR (MEMW) signal shown in Figure 4.22 as the reference signal. With the second probe, we check the circuit in Figure 4.19 at various points. The potential symptoms and probable conclusions are as follows.

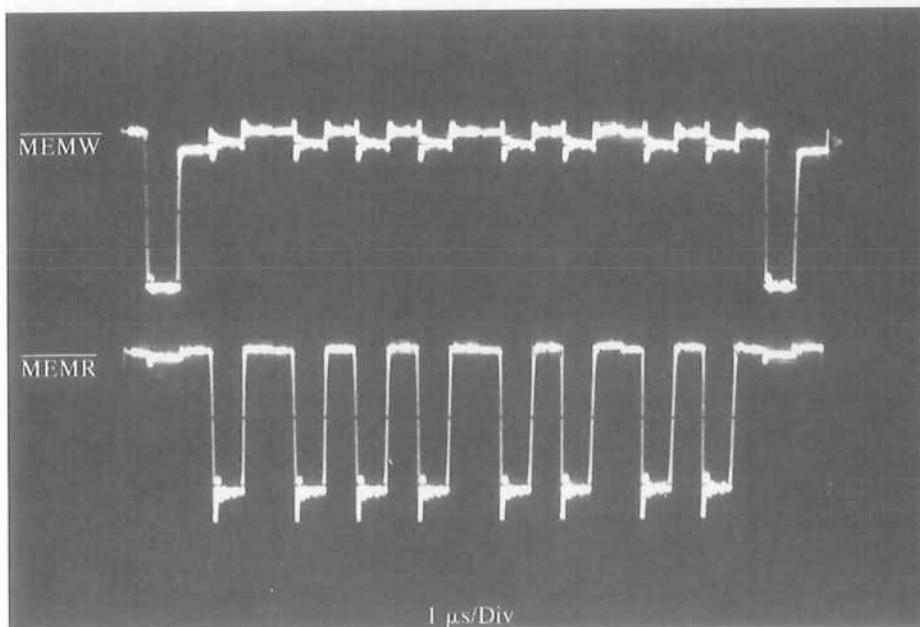


FIGURE 4.22
Timing Signals of Diagnostic Routine

1. Symptom: The data byte read from memory is different from the one that is stored (55H).
The data lines D_7 – D_0 are incorrectly connected.
2. Symptom: The signals at O_4 of the decoder and \overline{CE} of the memory are low (similar to the WR signal).
The address lines A_{15} – A_{11} are connected properly and the decoder chip is functioning correctly. Check the IO/M, ALE, and RD signals on the memory chip (assuming that we are checking the WR signal at the memory chip).
3. Symptom: The signals at O_4 of the decoder and \overline{CE} of the memory are high.
Check other outputs of the decoder. If one of the outputs is low, it is an indication that the input address lines of the decoder (A_{13} – A_{10}) are incorrectly connected. If none of the outputs is low, it is an indication that either the address lines A_{15} and A_{14} are not properly connected or the decoder chip is functioning incorrectly.
4. Symptom: The signals at O_4 of the decoder and \overline{CE} of the memory are low similar to the ground signal.
Either the V_{CC} to the decoder is not connected or the chip is functioning improperly.

4.7

HOW DOES AN 8085-BASED SINGLE-BOARD MICROCOMPUTER WORK?

Hardware is the skeleton of the computer; software is its life. The software (programs) makes the computer live; without it, the hardware is a dead piece of semiconductor material. Single-board microcomputers, such as the one shown in Figure 4.9, the system in Appendix B, or the SDK-85, have a program called Key Monitor or Key Executive permanently stored in memory. This program is stored either in EPROM or in ROM, beginning at the memory location 0000H.

When the power is turned on, the monitor program comes alive. Initially, the program counter has a random address. When the system is reset, the program counter in the 8085 is cleared, and it holds the address 0000H. The trainer system (Appendix B) includes a “power on” reset circuit, which resets the system and clears the program counter when the system is turned on. The MPU places the address 0000H on the address bus. The instruction code stored in location 0000H is fetched and executed, and the execution continues according to the instructions in the monitor program. The primary functions of the monitor program are as follows:

1. Reading the Hex keyboard and checking for a key closure. Continuing to check the keyboard until a key is pressed.
2. Displaying the Hex equivalent of the key pressed at the output port, such as the seven-segment LEDs.
3. Identifying the key pressed and storing its binary equivalent in memory, if necessary.
4. Transferring the program execution sequence to the user program when the Execute key is pressed.

The programmer enters a program in R/W memory in sequential memory locations by using the data keys (0 to F) and the function key called Enter. When the system is re-

set, the program counter is cleared, and the monitor program begins to check a key closure again. By using the keyboard, the programmer enters the first memory address where the user program is stored in R/W memory and directs the MPU to execute the program by pressing the Run key. The MPU fetches, decodes, and executes one instruction code at a time and continues to do so until it fetches the Halt instruction.

The key monitor program is a critical element in entering, storing, and executing a program. Until the Execute key is pushed, the monitor program in the EPROM (or ROM) directs all the operations of the MPU. After the Execute key is pushed, the user program directs the MPU to perform the functions written in the program.

SUMMARY

This chapter described the architecture of the 8085 microprocessor and illustrated the techniques for demultiplexing the bus AD_7 - AD_0 and generating the control signals. The bus timings of three operations—Opcode Fetch, Memory Read, and Memory Write—were examined and were used in discussing the basic concepts in memory interfacing. Several examples of memory interfacing were illustrated, and the concepts of memory addressing, absolute and partial decoding, and multiple addresses were discussed. The important concepts are summarized below.

- The 8085 microprocessor signals can be classified in six groups: address bus, data bus, control and status signals, externally initiated signals and their acknowledgment, power and frequency, and serial I/O signals.
- The data bus and the low-order address bus are multiplexed; they can be demultiplexed by using the ALE (Address Latch Enable) signal and a latch.
- The IO/M is a status signal—when it is high, it indicates an I/O operation; when it is low, it indicates a memory operation.
- The RD and WR are control signals; the RD is asserted to read from an external device, and the WR is asserted to write into an external device (memory or I/O).
- The RD and WR signals are logically ANDed with the IO/M signal to generate four active low control signals: MEMR, MEMW, IOR, and IOW.
- Each instruction of the 8085 microprocessor can be divided into a few basic operations called machine cycles, and each machine cycle can be divided into T-states.
- The frequently used machine cycles are Opcode Fetch, Memory Read and Write, and I/O Read and Write.
- When the 8085 performs any of the operations, it asserts the appropriate control signal and status signal.
- Most Opcode Fetch operations consist of four T-states, and the subsequent Memory Read or Memory Write cycles require three T-states. Some Opcode Fetch operations require six T-states.
- The Opcode Fetch and the Memory Read are operationally similar; the 8085 reads from memory in both machine cycles. However, the 8085 reads opcode during the Opcode Fetch cycle, and it reads 8-bit data during the Memory Read cycle. In the Memory Write cycle, the processor writes data into memory.

- The 8085 performs three basic steps in any of these machine cycles: It places the address on the address bus, sends appropriate control signals, and transfers data via data bus.
- To read from memory, the address of the register (to be read from) should be placed on the address lines and the RD signal must be asserted low to enable the output buffer.
- To write into memory, the address of the register (to be written into) should be placed on the address lines, a data byte should be placed on the data lines, and the WR signal must be asserted low to enable the input buffer.
- To interface a memory chip with the 8085, the necessary low-order address lines of the 8085 address bus are connected to the address lines of the memory chip. The high-order address lines are decoded to generate CS signals to enable the chip.
- In the absolute decoding technique, all the address lines that are not used for the memory chip to identify a memory register must be decoded; thus, the Chip Select can be asserted by only one address. In the partial decoding technique, some address lines are left don't care. This technique reduces hardware, but generates multiple addresses resulting in foldback memory space.

QUESTIONS AND PROBLEMS

1. Explain the functions of the ALE and IO/M̄ signals of the 8085 microprocessor.
2. Explain the need to demultiplex the bus AD₇–AD₀.
3. Figure 4.23 shows the 74LS138 (3-to-8) decoder with the three input signals: IO/M, RD, and WR from the 8085 microprocessor. Specify and name the valid output signals.
4. Explain why four output signals are invalid or meaningless in Figure 4.23.
5. Identify appropriate control signals that are generated at the output of the 2-to-4 decoder in Figure 4.24.

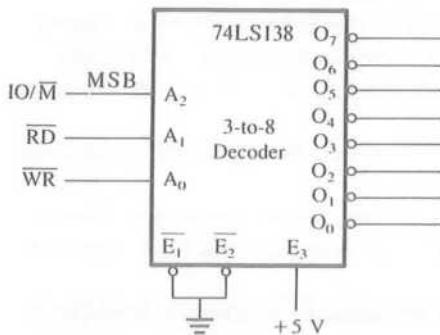


FIGURE 4.23

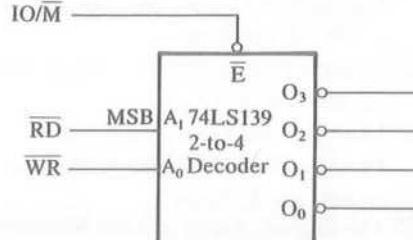


FIGURE 4.24

6. In Figure 4.4, if the 8085 places the address 20H on $A_{15}-A_8$ and 05H on AD_7-AD_0 and the ALE is high, specify the output of the latch 74LS373.
7. At T_2 (refer to Figure 4.3), the data byte 4FH is placed on AD_7-AD_0 . Specify the output of the latch in Figure 4.4. Explain your answer.
8. Specify the crystal frequency required for an 8085 system to operate at 1.1 MHz.
9. List the sequence of events that occurs when the 8085 MPU reads from memory.
10. If the 8085 adds 87H and 79H, specify the contents of the accumulator and the status of the S, Z, and CY flags.
11. If the 8085 has fetched the machine code located at the memory location 205FH, specify the contents of the program counter.
12. If the clock frequency is 5 MHz, how much time is required to execute an instruction of 18 T-states?
13. Assume that memory location 2075H has a data byte 47H. Specify the contents of the address bus $A_{15}-AD_8$ and the multiplexed bus AD_7-AD_0 when the MPU asserts the RD signal.
14. In the Opcode Fetch cycle, what are the control and status signals asserted by the 8085 to enable the memory buffer?
15. The instruction MOV B,M copies the contents of the memory location in register B. It is a 1-byte instruction with two machine cycles and seven T-states. Identify the second machine cycle and its control signal.
16. The instruction LDA 2050H copies the contents of the memory location 2050H into the accumulator. It is a 3-byte instruction with four machine cycles and 13 T-states. Identify the fourth machine cycle and its control signal.
17. In Question 16, identify the contents of the demultiplexed address bus $A_{15}-A_0$ and the data bus in the fourth machine cycle when the control signal is asserted.
18. Identify the machine cycles in the following instructions. (You should be able to identify the machine cycles even if you are not familiar with some of the instructions.)

```

SUB B      ; 1-byte, 4 T-states
            ; Subtract the contents of register B from the accumulator
ADI 47H    ; 2-byte, 7 (4, 3) T-states
            ; Add 47H to the contents of the accumulator
STA 2050H  ; 3-byte, 13 (4, 3, 3) T-states
            ; Copies the contents of the accumulator into memory
            ; location 2050H
PUSH B     ; 1-byte, 12 (6, 3, 3) T-states
            ; Copies the contents of the BC register into two stack memory
            ; locations

```

19. In Figure 4.25, connect the output line O_6 of the decoder to the \overline{CE} of the memory chip instead of O_0 , and identify the memory map.
20. In Figure 4.25, connect A_{15} to the active high enable signal E_3 of the decoder, and ground E_1 . Identify the memory map of the chip.

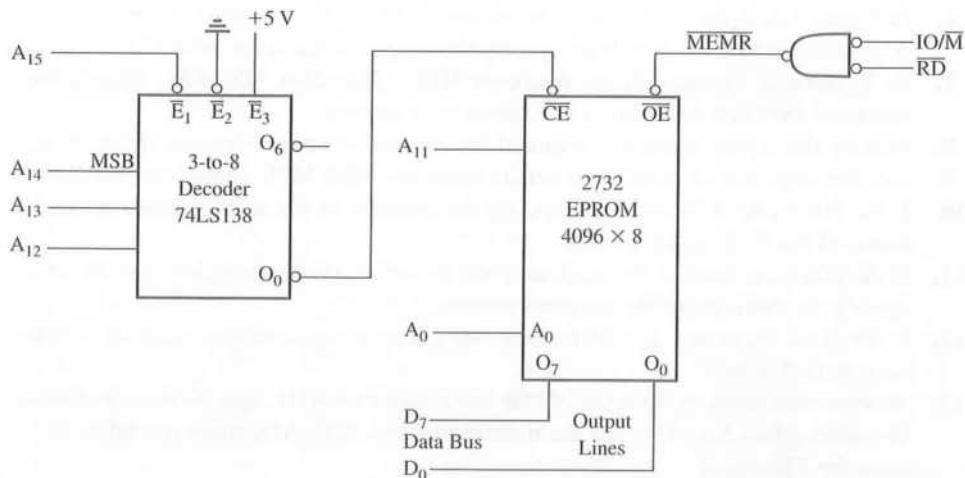


FIGURE 4.25
Interfacing the 2732 EPROM

21. Identify the actual gate you would use to generate the $\overline{\text{MEMR}}$ signal in Figure 4.25.
22. Modify the schematic in Figure 4.25 to eliminate the negative NAND gate and obtain the same memory address range without adding other components.
23. In Figure 4.26, exchange A_{15} and A_{13} and identify the memory map.
24. In Figure 4.26, if we use all the output lines (O_7-O_0) of the decoder to select eight memory chips of the same size as the 6116, what is the total range of the memory map?
25. In the SDK-85 system, the specified map of the 8155 memory is 2000H to 20FFH (see Figure 4.27). If you enter a data byte at the location 2100H, will the system

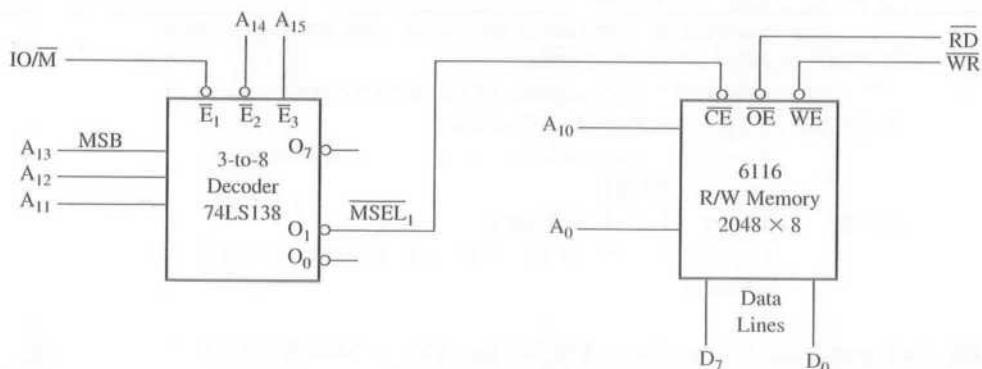


FIGURE 4.26
Interfacing R/W Memory

accept the data byte? If it accepts it, where will it store the data byte? Explain your answer.

26. In Figure 4.27, specify the memory address range if output line O_1 of the decoder 8205 is connected to the \overline{CE} signal. Specify the range of the foldback memory.
27. In Figure 4.27 specify the memory address range if output line O_7 of the decoder 8205 is connected to the \overline{CE} signal of a 2K (2048) memory chip.
28. By examining the range of the foldback memory in Figure 4.27, specify the relationship between the range of foldback memory and the number of don't care lines.
29. In Figure 4.28, specify the memory addresses of ROM1, ROM2, and R/WM1.
30. In Figure 4.28, eliminate the second decoder and connect CS_4 to CE of the R/WM1, and identify its memory map and foldback space.
31. In Figure 4.29, identify the address range of the memory chip.
32. In Figure 4.29, connect Y_1 to CE of the memory chip in place of Y_0 , and identify the address range of the memory chip.
33. In Figure 4.29, replace the 27128 (16K) memory chip with the 2764 (8K) memory chip. Identify the primary address range and the mirror (foldback) address range of the memory chip for the given decoding circuit.
34. In Question 33, the address line A_{13} was at a don't care logic state. Replace the address line A_{15} by the address line A_{13} , leave A_{15} as don't care, and identify the mirror address range.
35. In Figure 4.22, identify the MEMR signals of the Opcode Fetch machine cycles.
36. In Figure 4.22, identify the machine cycle and the Hex code read by the processor when it asserts the last MEMR signal.

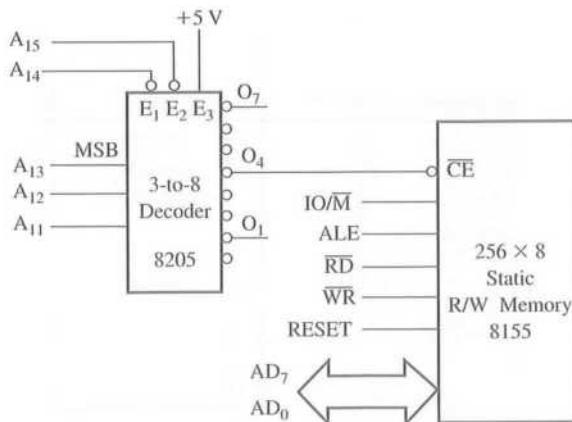


FIGURE 4.27

Interfacing the 8155 Memory
Schematic from the SDK-85 System

SOURCE: Intel Corporation, *SDK-85 User's Manual* (Santa Clara, Calif.: Author, 1978), Appendix B.

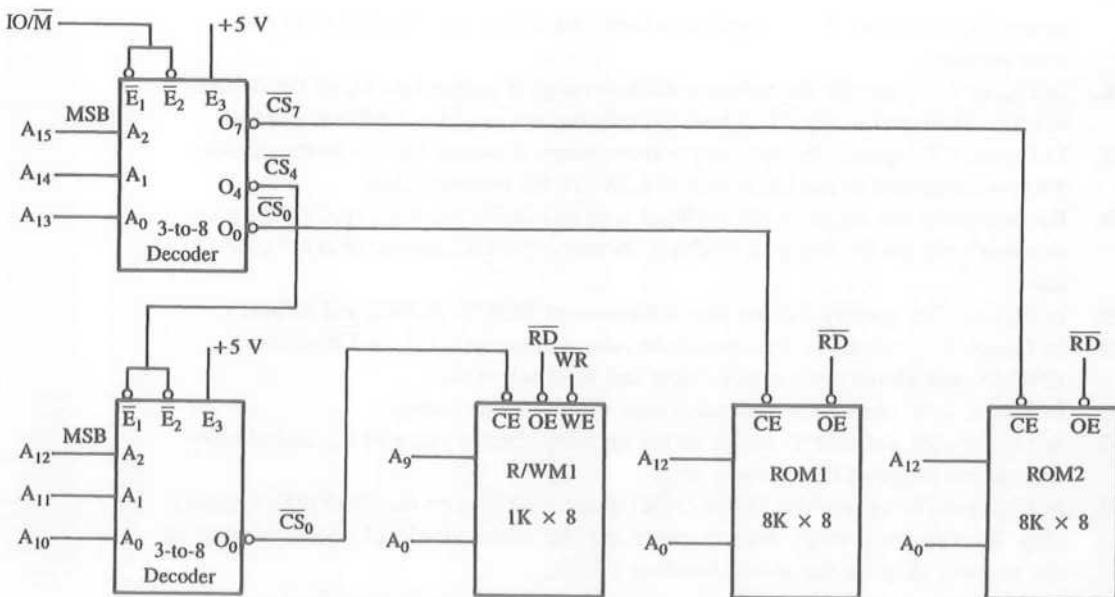
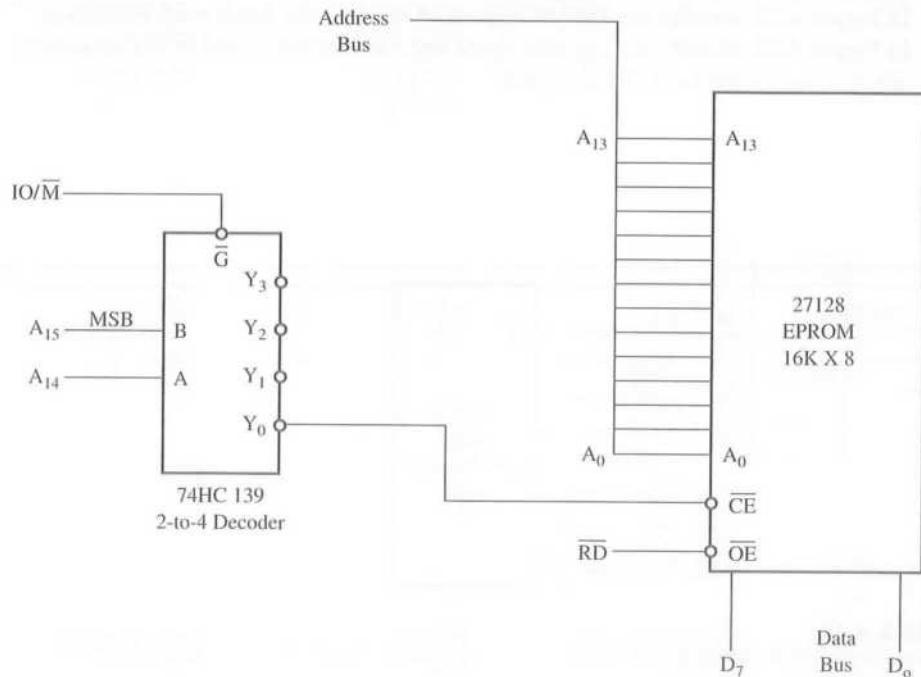


FIGURE 4.28

FIGURE 4.29
27128 (16K) EPROM Interfacing Circuit

5

Interfacing I/O Devices

The I/O devices, such as keyboards and displays, are the ears and eyes of the MPUs; they are the communication channels to the “outside world.” Data can enter (or exit) in groups of eight bits using the entire data bus; this is called the parallel I/O mode. The other method is the serial I/O, whereby one bit is transferred using one data line; typical examples include peripherals such as the CRT terminal and cassette tape. In this chapter, we will focus on interfacing I/O devices in the parallel mode and will discuss the serial mode in Chapter 16.

In the last chapter, we discussed memory interfacing. The 8085 microprocessor uses a 16-bit address bus for identifying and accessing memory registers. This results in a numbering scheme ranging from 0000H to FFFFH, also known as memory space. Similarly, the microprocessor needs to identify I/O devices with a binary number. These I/O devices can be interfaced using addresses (binary numbers) from the memory space; this is called memory-mapped I/O. Another option is to have a

separate numbering (addressing) scheme for I/O devices. The 8085 microprocessor has a separate 8-bit addressing scheme (I/O space) for I/O devices; this is called peripheral-mapped I/O (or I/O-mapped I/O), and the I/O space ranges from 00H to FFH. In the 8085-based systems, I/O devices can be interfaced using both techniques: peripheral-mapped I/O and memory-mapped I/O. In peripheral-mapped I/O, a device is identified with an 8-bit address and enabled by I/O-related control signals. On the other hand, in memory-mapped I/O, a device is identified with a 16-bit address and enabled by memory-related control signals. The process of data transfer in both is identical. Each device is assigned a binary address, called a device address or port number, through its interfacing circuit. When the microprocessor executes a data transfer instruction for an I/O device, it places the appropriate address on the address bus, sends the control signals, enables the interfacing device, and transfers data. The interfacing device is like a gate for data bits, which is

opened by the MPU whenever it intends to transfer data. In peripheral-mapped I/O, data bytes are transferred by using IN/OUT instructions, and in memory-mapped I/O, data bytes are transferred by using memory-related (LDA, STA, etc.) data transfer instructions.

To grasp the essence of interfacing techniques, first we will examine the machine cycles of I/O instructions and find out the timings when I/O data are arriving on the data bus, and then we will latch (or catch) that information. We will derive the basic concepts in interfacing I/O devices from the machine cycles. Then we will illustrate these concepts by interfacing LEDs as an output device and switches as an input device. Specifically, the chapter deals with how the 8085 selects an I/O device, what hardware chips are necessary, what software instructions are used, and how data are transferred.

OBJECTIVES

- Illustrate the 8085 bus contents and control signals when OUT and IN instructions are executed.
- Recognize the device (port) address of a peripheral-mapped I/O by analyzing the associated logic circuit.
- Illustrate the 8085 bus contents and control signals when memory-related instructions (LDA, STA, etc.) are executed.
- Recognize the device (port) address of a memory-mapped I/O by analyzing the associated logic circuit.
- Explain the differences between the peripheral-mapped and memory-mapped I/O techniques.
- Interface an I/O device to a microcomputer for a specified device address by using logic gates and MSI chips, such as decoders, latches, and buffers.

5.1

BASIC INTERFACING CONCEPTS

The approach to designing an interfacing circuit for an I/O device is determined primarily by the instructions to be used for data transfer. An I/O device can be interfaced with the 8085 microprocessor either as a peripheral I/O or as a memory-mapped I/O. In the peripheral I/O, the instructions IN/OUT are used for data transfer, and the device is identified by an 8-bit address. In the memory-mapped I/O, memory-related instructions are used for data transfer, and the device is identified by a 16-bit address. However, the basic concepts in interfacing I/O devices are similar in both methods. Peripheral I/O is described in the following section, and memory-mapped I/O is described in Section 5.4.

5.1.1 Peripheral I/O Instructions

The 8085 microprocessor has two instructions for data transfer between the processor and the I/O device: IN and OUT. The instruction IN (Code DB) inputs data from an input device (such as a keyboard) into the accumulator, and the instruction OUT (Code D3) sends the contents of the accumulator to an output device such as an LED display. These are 2-byte instructions, with the second byte specifying the address or the port number of an I/O device. For example, the OUT instruction is described as follows.

Opcode	Operand	Description
OUT	8-bit Port Address:	This is a two-byte instruction with the hexadecimal opcode D3, and the second byte is the port address of an output device. This instruction transfers (copies) data from the accumulator to the output device.

Typically, to display the contents of the accumulator at an output device (such as LEDs) with the address, for example, 01H, the instruction will be written and stored in memory as follows:

Memory Address	Machine Code	Mnemonics	Memory Contents								
2050	D3	OUT 01H	; 2050 → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table> = D3H	1	1	0	1	0	0	1	1
1	1	0	1	0	0	1	1				
2051	01		; 2051 → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table> = 01H	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1				

(Note: The memory locations 2050H and 2051H are chosen here arbitrarily for the illustration.)

If the output port with the address 01H is designed as an LED display, the instruction OUT will display the contents of the accumulator at the port. The second byte of this OUT instruction can be any of the 256 combinations of eight bits, from 00H to FFH. Therefore, the 8085 can communicate with 256 different output ports with device addresses ranging from 00H to FFH. Similarly, the instruction IN can be used to accept data from 256 different input ports. Now the question remains: How does one assign a device address or a port number to an I/O device from among 256 combinations? The decision is arbitrary and somewhat dependent on available logic chips. To understand a device address, it is necessary to examine how the microprocessor executes IN/OUT instructions.

5.1.2 I/O Execution

The execution of I/O instructions can best be illustrated using the example of the OUT instruction given in the previous section (5.1.1). The 8085 executes the OUT instruction in three machine cycles, and it takes ten T-states (clock periods) to complete the execution.

OUT INSTRUCTION (8085)

In the first machine cycle, M₁ (Opcode Fetch, Figure 5.1), the 8085 places the high-order memory address 20H on A₁₅-A₈ and the low-order address 50H on AD₇-AD₀. At the same time, ALE goes high and IO/M goes low. The ALE signal indicates the availability of the address on AD₇-AD₀, and it can be used to demultiplex the bus. The IO/M, being low, indicates that it is a memory-related operation. At T₂, the microprocessor sends the RD control signal, which is combined with IO/M (externally, see Chapter 4) to generate the MEMR signal, and the processor fetches the instruction code D3 using the data bus.

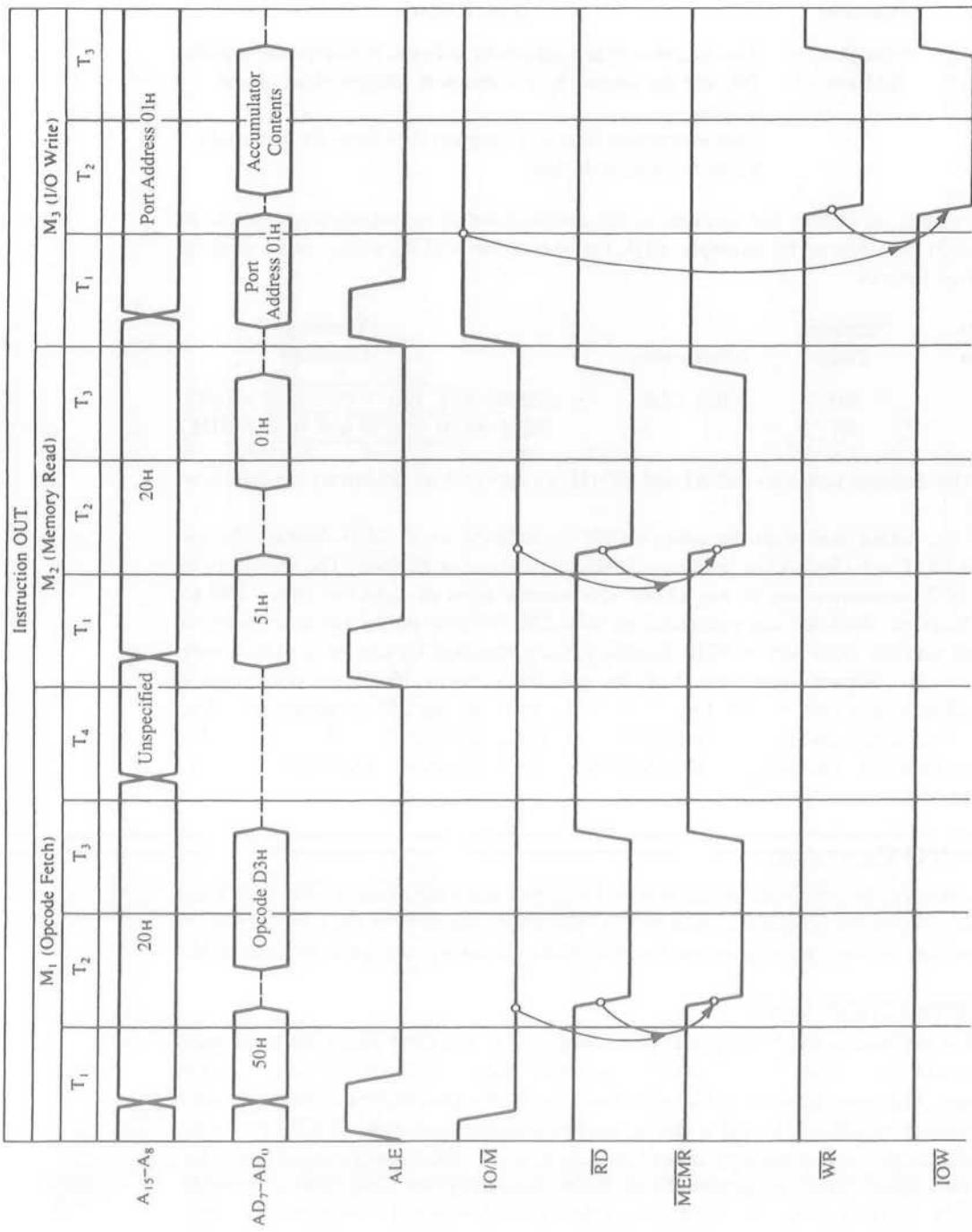


FIGURE 5.1
8085 Timing for Execution of OUT Instruction

When the 8085 decodes the machine code D3, it finds out that the instruction is a 2-byte instruction and that it must read the second byte.

In the second machine cycle, M₂ (Memory Read), the 8085 places the next address, 2051H, on the address bus and gets the device address 01H via the data bus.

In the third machine cycle, M₃ (I/O Write), the 8085 places the device address 01H on the low-order (AD₇-AD₀) as well as the high-order (A₁₅-A₈) address bus. The IO/M signal goes high to indicate that it is an I/O operation. At T₂, the accumulator contents are placed on the data bus (AD₇-AD₀), followed by the control signal WR. By ANDing the IO/M and WR signals, the IOW (see Figure 4.5) signal can be generated to enable an output device.

Figure 5.1 shows the execution timing of the OUT instruction. The information necessary for interfacing an output device is available during T₂ and T₃ of the M₃ cycle. The data byte to be displayed is on the data bus, the 8-bit device address is available on the low-order as well as high-order address bus, and availability of the data byte is indicated by the WR control signal. The availability of the device address on both segments of the address bus is redundant information; in peripheral I/O, only one segment of the address bus (low or high) is sufficient for interfacing. The data byte remains on the data bus only for two T-states, then the processor goes on to execute the next instruction. Therefore, the data byte must be latched now, before it is lost, using the device address and the control signal (Section 5.13).

IN INSTRUCTION

The 8085 instruction set includes the instruction IN to read (copy) data from input devices such as switches, keyboards, and A/D data converters. This is a two-byte instruction that reads an input device and places the data in the accumulator. The first byte is the opcode, and the second byte specifies the port address. Thus, the addresses for input devices can range from 00H to FFH. The instruction is described as

IN 8-bit This is a two-byte instruction with the hexadecimal opcode DB, and the second byte is the port address of an input device.

This instruction reads (copies) data from an input device and places the data byte in the accumulator.

To read switch positions, for example, from an input port with the address 84H, the instructions will be written and stored in memory as follows:

Memory Address	Machine Code	Mnemonics	Memory Contents								
2065	DB	IN 84H	; 2065 → <table border="1" style="display: inline-table;"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table> = DBH	1	1	0	1	1	0	1	1
1	1	0	1	1	0	1	1				
2066	84		; 2066 → <table border="1" style="display: inline-table;"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table> = 84H	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0				

(Note: The memory locations 2065H and 66H are selected arbitrarily for the illustration.)

When the microprocessor is asked to execute this instruction, it will first read the machine codes (or bytes) stored at locations 2065H and 2066H, then read the switch po-

sitions at port 84H by enabling the interfacing device of the port. The data byte indicating switch positions from the input port will be placed in the accumulator. Figure 5.2 shows the timing of the IN instruction; M₁ and M₂ cycles are identical to that of the OUT instruction. In the M₃ cycle, the 8085 microprocessor places the address of the input port (84H) on the low-order address bus AD₇–AD₀ as well as on the high-order address bus A₁₅–A₈ and asserts the RD signal, which is used to generate the I/O Read (IOR) signal. The IOR enables the input port, and the data from the input port are placed on the data bus and transferred into the accumulator.

Machine cycle M₃ (Figure 5.2) is similar to the M₃ cycle of the OUT instruction; the only differences are (1) the control signal is RD instead of WR, and (2) data flow from an input port to the accumulator rather than from the accumulator to an output port.

5.1.3 Device Selection and Data Transfer

The objective of interfacing an output device is to get information or a result out of the processor and store it or display it. The OUT instruction serves that purpose; during the M₃ cycle of the OUT instruction the processor places that information (accumulator contents) on the data bus. If we connect the data bus to a latch, we can catch that information and display it via LEDs or a printer. Now the questions are: (1) When should we enable the latch to catch that information? and (2) What should be the address of that latch? The answers to both questions can be found in the M₃ cycle (Figure 5.1). The latch should be enabled when IO/M is high and WR is active low. Similarly, the address of an output port is also on the address bus during M₃ (it is 01H in Figure 5.1). Now the task is to generate one pulse by decoding the address bus (A₇–A₀ or A₁₅–A₈) to indicate the presence of the port address we are interested in, generate a timing pulse by combining IO/M and WR signals to indicate that the data byte we are looking for is on the data bus, and use these pulses (by combining them) to enable the latch. These steps are summarized as follows. (For all subsequent discussion, the bus A₇–A₀ is assumed to be the demultiplexed bus AD₇–AD₀.)

1. Decode the address bus to generate a unique pulse corresponding to the device address on the bus; this is called the **device address pulse** or **I/O address pulse**.
2. Combine (AND) the device address pulse with the control signal to generate a device select (I/O select) pulse that is generated only when both signals are asserted.
3. Use the I/O select pulse to activate the interfacing device (I/O port).

The block diagram (Figure 5.3) illustrates these steps for interfacing an I/O device. In Figure 5.3, address lines A₇–A₀ are connected to a decoder, which will generate a unique pulse corresponding to each address on the address lines. This pulse is combined with the control signal to generate a device select pulse, which is used to enable an output latch or an input buffer.

Figure 5.4 shows a practical decoding circuit for the output device with address 01H. Address lines A₇–A₀ are connected to the 8-input NAND gate that functions as a decoder. Line A₀ is connected directly, and lines A₇–A₁ are connected through the inverters. When the address bus carries address 01H, gate G₁ generates a low pulse; otherwise, the

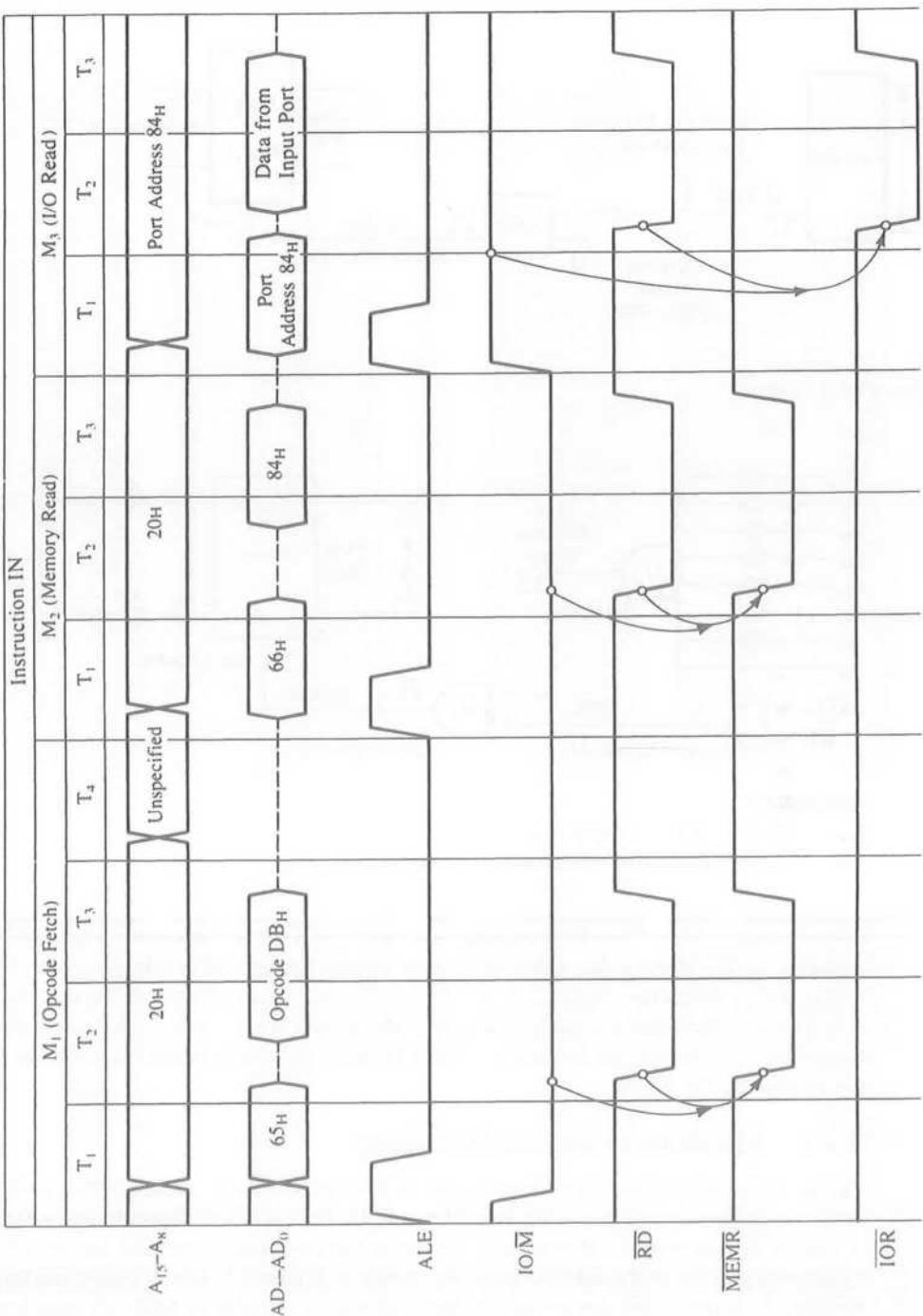


FIGURE 5.2
8085 Timing for Execution of IN Instruction

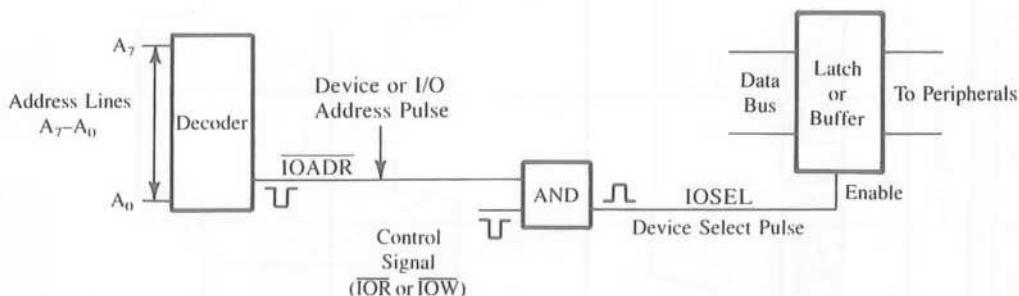


FIGURE 5.3

Block Diagram of I/O Interface

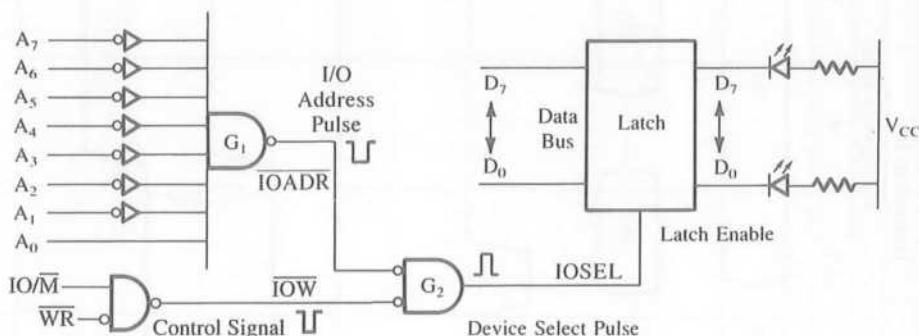


FIGURE 5.4

Decode Logic for LED Output Port

NOTE: To use this circuit with the 8085, the bus AD₇-A₀ must be demultiplexed.

output remains high. Gate G₂ combines the output of G₁ and the control signal IOW to generate an I/O select pulse when both input signals are low. Meanwhile (as was shown in the timing diagram—Figure 5.1, machine cycle M₃), the contents of the accumulator are placed on the data bus and are available on the data bus for a few microseconds and, therefore, must be latched for display. The I/O select pulse clocks the data into the latch for display by the LEDs.

5.1.4 Absolute vs. Partial Decoding

In Figure 5.4, all eight address lines are decoded to generate one unique output pulse; the device will be selected only with the address, 01H. This is called **absolute decoding** and is a good design practice. However, to minimize the cost, the output port can be selected by decoding some of the address lines, as shown in Figure 5.5; this is called **partial decoding**. As a result, the device has multiple addresses (similar to foldback memory addresses).

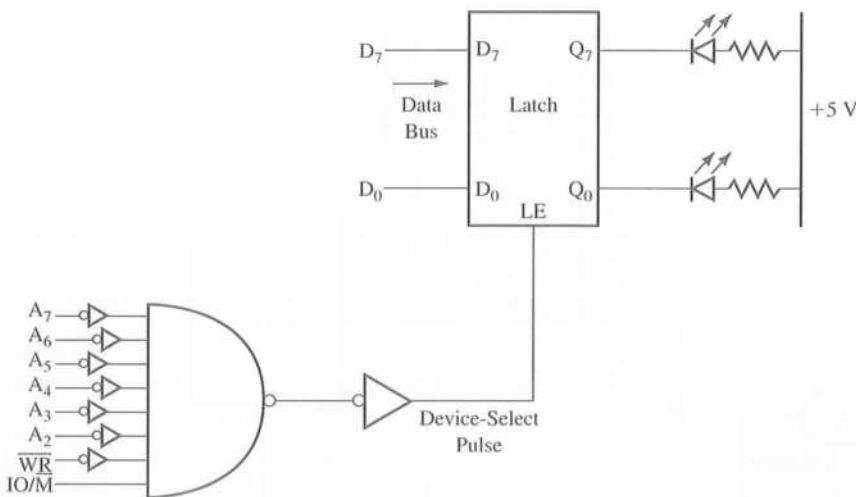


FIGURE 5.5
Partial Decoding: Output Latch with Multiple Addresses

Figure 5.5 is similar to Figure 5.4 except that the address lines A₁ and A₀ are not connected, and they are replaced by IO/M and WR signals. Because the address lines A₁ and A₀ are at don't care logic level, they can be assumed to be 0 or 1. Thus this output port (latch) can be accessed by the Hex addresses 00, 01, 02, and 03. The partial decoding is a commonly used technique in small systems. Such multiple addresses will not cause any problems, provided these addresses are not assigned to any other output ports.

5.1.5 Input Interfacing

Figure 5.6 shows an example of interfacing an 8-key input port. The basic concepts behind this circuit are similar to the interfacing concepts explained earlier.

The address lines are decoded by using an 8-input NAND gate. When address lines A₇–A₀ are high (FFH), the output of the NAND gate goes low and is combined with control signal IOR in gate G₂. When the MPU executes the instruction (IN FFH), gate G₂ generates the device select pulse that is used to enable the tri-state buffer. Data from the keys are put on the data bus D₇–D₀ and loaded into the accumulator. The circuit for the input port in Figure 5.6 differs from the output port in Figure 5.4 as follows:

1. Control signal $\overline{\text{IOR}}$ is used in place of $\overline{\text{IOW}}$.
2. The tri-state buffer is used as an interfacing port in place of the latch.
3. In Figure 5.6, data flow from the keys to the accumulator; on the other hand, in Figure 5.4, data flow from the accumulator to the LEDs.

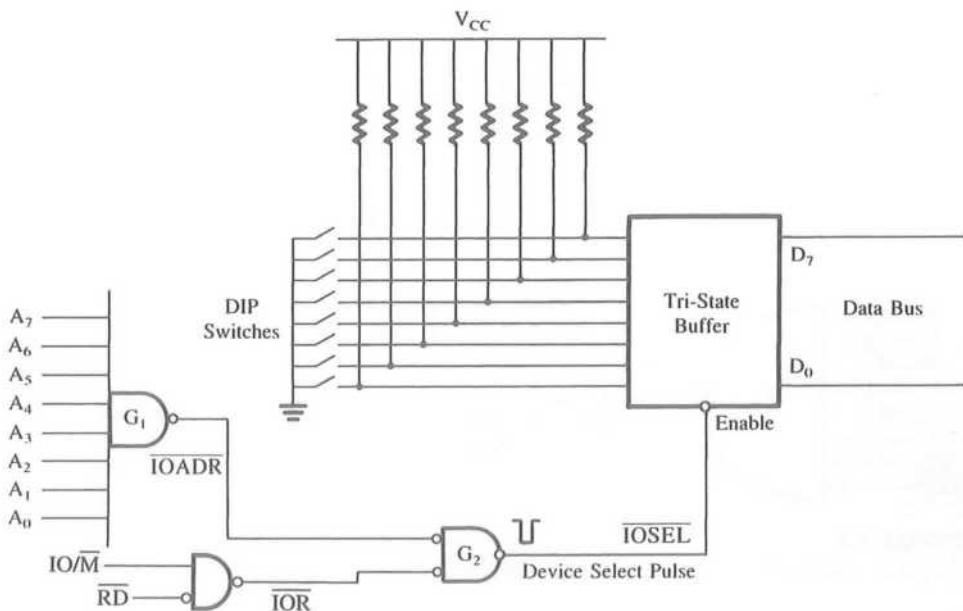


FIGURE 5.6
Decode Logic for a Dip-Switch Input Port

5.1.6 Interfacing I/Os Using Decoders

Various techniques and circuits can be used to decode an address and interface an I/O device to the microprocessor. However, all of these techniques should follow the three basic steps suggested in Section 5.1.3. Figures 5.4 and 5.6 illustrate an approach to device selection using an 8-input NAND gate. Figure 5.5 illustrates a technique using minimum hardware; this technique has the disadvantage of having multiple addresses for the same device. Figure 5.7 illustrates another scheme of address decoding. In this circuit, a 3-to-8 decoder and a 4-input NAND gate are used to decode the address bus; the decoding of the address bus is the first step in interfacing I/O devices. The address lines A₂, A₁, and A₀ are used as input to the decoder, and the remaining address lines A₇–A₃ are used to enable the decoder. The address line A₇ is directly connected to E₃ (active high Enable line), and the address lines A₆–A₃ are connected to E₁ and E₂ (active low Enable lines) using the NAND gate. The decoder has eight output lines; thus, we can use this circuit to generate eight device address pulses for eight different addresses.

The second step is to combine the decoded address with an appropriate control signal to generate the I/O select pulse. Figure 5.7 shows that the output O₀ of the decoder is logically ANDed in a negative AND gate with the IOW control signal. The output of the gate is the I/O select pulse for an output port. The third step is to use this pulse to enable the output port. Figure 5.7 shows that the I/O select pulse enables the LED latch with the output port address F8H, as shown below (A₇–A₀ is the demultiplexed low-order bus).

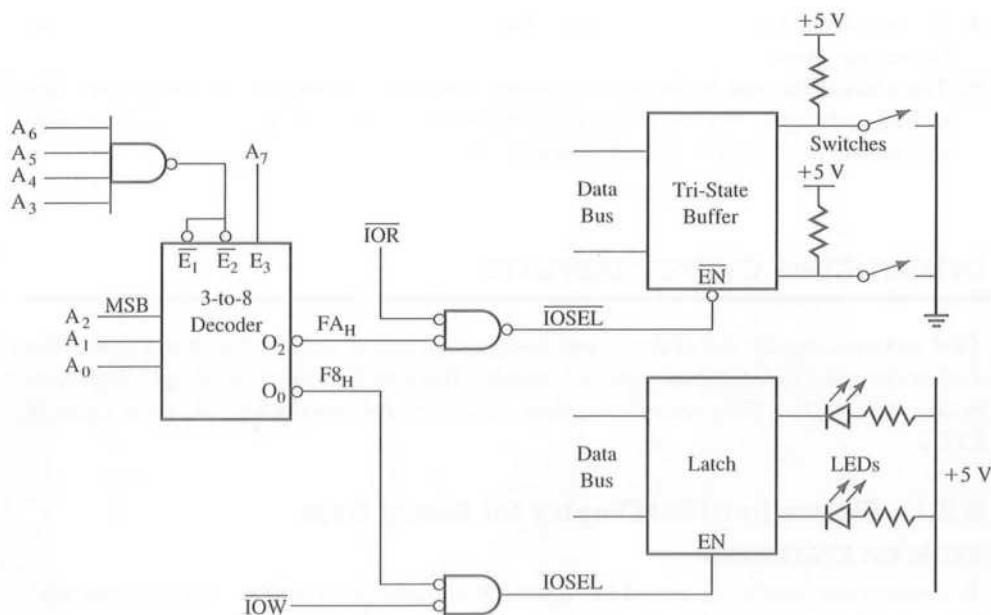
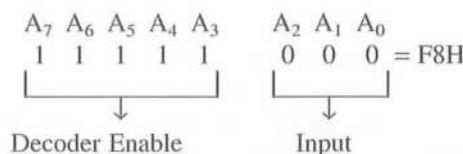


FIGURE 5.7
Address Decoding Using a 3-to-8 Decoder



Similarly, the output O_2 of the decoder is combined with the I/O Read (\overline{IOR}) signal, and the I/O select pulse is used to enable the input buffer with the address FAH .

5.1.7 Review of Important Concepts

In peripheral I/O, the basic concepts and the steps in designing an interfacing circuit can be summarized as follows:

- When an I/O instruction is executed, the 8085 microprocessor places the device address (port number) on the demultiplexed low-order as well as the high-order address bus.
- Either the high-order bus ($A_{15}-A_8$) or the demultiplexed low-order bus (A_7-A_0) can be decoded to generate the pulse corresponding to the device address on the bus.
- The device address pulse is ANDed with the appropriate control signal (\overline{IOR} or \overline{IOW}) and, when both signals are asserted, the I/O port is selected.

4. As interfacing devices, a latch is used for an output port and a tri-state buffer is used for an input port.
5. The address bus can be decoded by using either the absolute- or the linear-select decoding technique. The linear-select decoding technique reduces the component cost, but the I/O device ends up with multiple addresses.

5.2

INTERFACING OUTPUT DISPLAYS

This section concerns the analysis and design of practical circuits for data display. The section includes two different types of circuits. The first illustrates the simple display of binary data with LEDs, and the second illustrates the interfacing of seven-segment LEDs.

5.2.1 Illustration: LED Display for Binary Data

PROBLEM STATEMENT

1. Analyze the interfacing circuit in Figure 5.8(a), identify the address of the output port, and explain the circuit operation.
2. Explain similarities between (a) and (b) in Figure 5.8.
3. Write instructions to display binary data at the port.

CIRCUIT ANALYSIS

Address bus $A_7 - A_0$ is decoded by using an 8-input NAND gate. The output of the NAND gate goes low only when the address lines carry the address FFH. The output of the NAND gate is combined with the microprocessor control signal \overline{IOW} in a NOR gate (connected as negative AND). The output of NOR gate 74LS02 goes high to generate an I/O select pulse when both inputs are low (or both signals are asserted). Meanwhile, the contents of the accumulator have been put on the data bus. The I/O select pulse is used as a clock pulse to activate the D-type latch, and the data are latched and displayed.

In this circuit, the LED cathodes are connected to the \overline{Q} output of the latch. The anodes are connected to +5 V through resistors to limit the current flow through the diodes. When the data line (for example D_0) has 1, the output \overline{Q} is 0 and the corresponding LED is turned on. If the LED anode were connected to Q , its cathode would be connected to the ground. In this configuration, the D flip-flop would not be able to supply the necessary current to the LED.

Figure 5.8(b) uses the 74LS373 octal latch as an interfacing device, and both circuits (a) and (b) are functionally similar. The 74LS373 includes D-latches (flip-flops) followed by tri-state buffers (see Figure 3.27 for details). This device has two control signals: Enable (G) to clock data in the flip-flops and Output Control (\overline{OC}) to enable the buffers. In this circuit, the 74LS373 is used as a latch; therefore, the tri-state buffers are enabled by grounding the \overline{OC} signal.

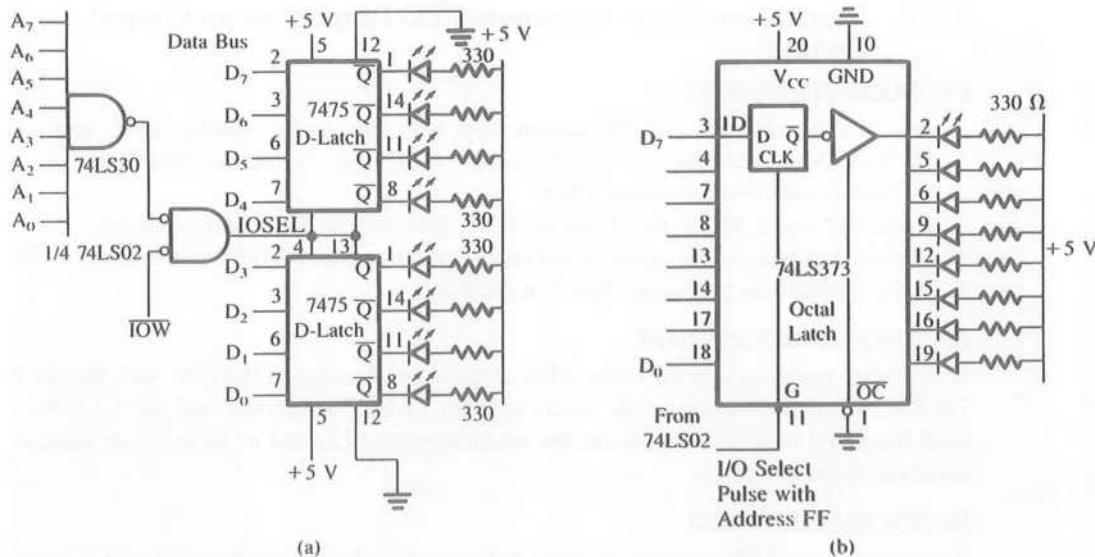


FIGURE 5.8

Interfacing LED Output Port Using the 7475 D-Type Latch (a) and Using the 74LS373 Octal D-Type Latch (b)

PROGRAM

Address (LO)	Machine Code	Mnemonics	Comments
00	3E	MVI A,DATA	;Load accumulator with data
01	DATA*		
02	D3	OUT FFH	;Output accumulator contents ; to port FFH
03	FF		
04	76	HLT	;End of program

PROGRAM DESCRIPTION

Instruction MVI A loads the accumulator with the data you enter, and instruction OUT FFH identifies the LED port as the output device and displays the data.

*Enter data you wish to display.

5.2.2 Illustration: Seven-Segment LED Display as an Output Device

PROBLEM STATEMENT

1. Design a seven-segment LED output port with the device address F5H, using a 74LS138 3-to-8 decoder, a 74LS20 4-input NAND gate, a 74LS02 NOR gate, and a common-anode seven-segment LED.
2. Given WR and IO/M signals from the 8085, generate the \overline{IOW} control signal.
3. Explain the binary codes required to display 0 to F Hex digits at the seven-segment LED.
4. Write instructions to display digit 7 at the port.

HARDWARE DESCRIPTION

The design problem specifies two MSI chips—the decoder (74LS138) and the latch 74LS373—and a common-anode seven-segment LED. The decoder and the latch have been described in previous sections; the seven-segment LED and its binary code requirement are discussed below.

SEVEN-SEGMENT LED

A seven-segment LED consists of seven light-emitting diode segments and one segment for the decimal point. These LEDs are physically arranged as shown in Figure 5.9(a). To display a number, the necessary segments are lit by sending an appropriate signal for cur-

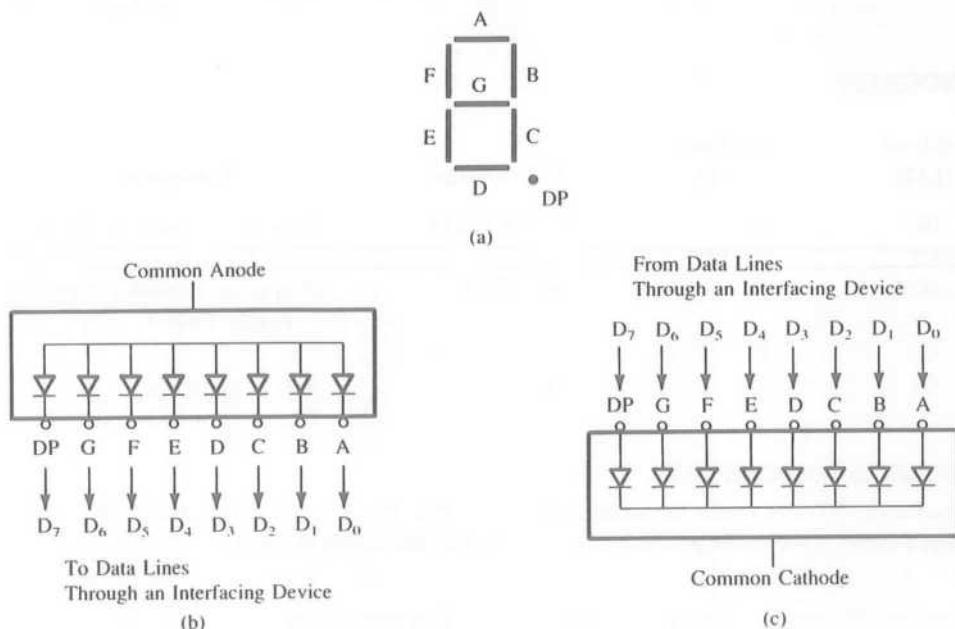


FIGURE 5.9

Seven-Segment LED: LED Segments (a); Common-Anode LED (b); Common-Cathode LED (c)

rent flow through diodes. For example, to display an 8, all segments must be lit. To display 1, segments B and C must be lit. Seven-segment LEDs are available in two types: common cathode and common anode. They can be represented schematically as in Figure 5.9(b) and (c). Current flow in these diodes should be limited to 20 mA.

The seven segments, A through G, are usually connected to data lines D_0 through D_6 , respectively. If the decimal-point segment is being used, data line D_7 is connected to DP; otherwise it is left open. The binary code required to display a digit is determined by the type of the seven-segment LED (common cathode or common anode), the connections of the data lines, and the logic required to light the segment. For example, to display digit 7 at the LED in Figure 5.10, the requirements are as follows:

1. It is a common-anode seven-segment LED, and logic 0 is required to turn on a segment.
2. To display digit 7, segments A, B, and C should be turned on.
3. The binary code should be

Data Lines	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
Bits	X	1	1	1	1	0	0	0	= 78H
Segments	NC	G	F	E	D	C	B	A	

The code for each digit can be determined by examining the connections of the data lines to the segments and the logic requirements.

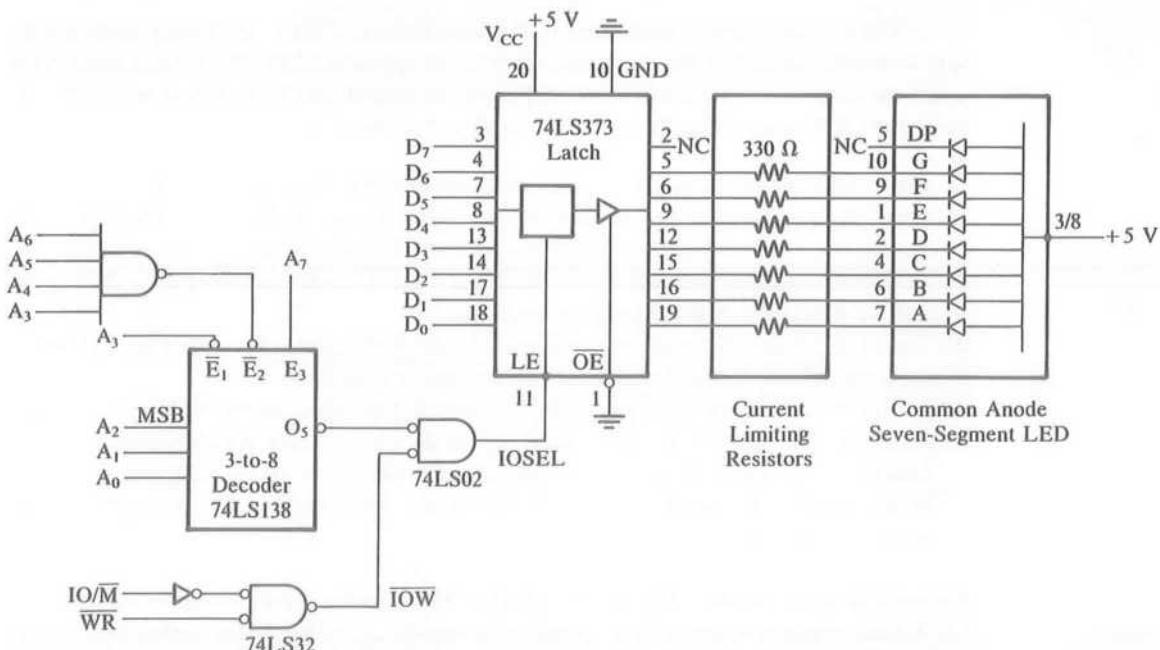


FIGURE 5.10
Interfacing Seven-Segment LED

INTERFACING CIRCUIT AND ITS ANALYSIS

To design an output port with the address F5H, the address lines A₇–A₀ should have the following logic:

A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	= F5H
1	1	1	1	0	1	0	1	

This can be accomplished by using A₂, A₁, and A₀ as input lines to the decoder. A₃ can be connected to active low enable E₁, and the remaining address lines can be connected to E₂ through the 4-input NAND gate. Figure 5.10 shows an output port with the address F5H. The output O₅ of the decoder is logically ANDed with the control signal IOW using the NOR gate (74LS02). The output of the NOR gate is the I/O select pulse that is used to enable the latch (74LS373). The control signal IOW is generated by logically ANDing IO/M and WR signals in the negative NAND gate (physically OR gate 74LS32).

Instructions The following instructions are necessary to display digit 7 at the output port:

```
MVI A,78H      ;Load seven-segment code in the accumulator
OUT F5H        ;Display digit 7 at port F5H
HLT            ;End
```

The first instruction loads 78H in the accumulator; 78H is the binary code necessary to display digit 7 at the common-anode seven-segment LED. The second instruction sends the contents of the accumulator (78H) to the output port F5H. When the 8085 executes the OUT instruction, the digit 7 is displayed at the port as follows:

1. In the third machine cycle M₃ of the OUT instruction (refer to Figure 5.1), the port address F5H is placed on the address bus A₇–A₀ (it is also duplicated on the high-order bus A₁₅–A₈, but we have used the low-order bus for interfacing in this example).
2. The address F5H is decoded by the decoding logic (decoder and 4-input NAND gate), and the output O₅ of the decoder is asserted.
3. During T₂ of the M₃ cycle (see Figure 5.1), the 8085 places the data byte 78H from the accumulator on the data bus and asserts the WR signal.
4. In Figure 5.10, when the IOW signal is asserted, the output of the NOR gate 74LS02 goes high and enables the latch 74LS373. The data byte (78H), which is already on the data bus at the input of the latch, is passed on to the output of the latch and displayed by the seven-segment LED. However, the byte is latched when the WR signal is deasserted during T₃.

Current Requirements The circuit in Figure 5.10 uses a common-anode seven-segment LED. Each segment requires 10 to 15 mA of current ($I_{D\max} = 19 \text{ mA}$) for appropriate illumination. The latch can sink 24 mA when the output is low and can supply approximately 2.6 mA when the output is high. In this circuit, the common-anode LED segments are turned on by zeros on the output of the latch. If common-cathode seven-segment LEDs were used in

in this circuit, the output of the latch would have to be high to drive the segments. The current supplied would be about 2.6 mA, which is insufficient to make the segments visible.

INTERFACING INPUT DEVICES

5.3

The interfacing of input devices is similar to that of the interfacing output devices, except with some differences in bus signals and circuit components. We will follow the same basic steps described in Section 5.1.3 and the timing diagram for the execution of the IN instruction shown in Figure 5.2.

5.3.1 Illustration: Data Input from DIP Switches

In this section, we will analyze the circuit used for interfacing eight DIP switches, as shown in Figure 5.11. The circuit includes the 74LS138 3-to-8 decoder to decode the low-order bus and the tri-state octal buffer (74LS244) to interface the switches to the data bus. The port can be accessed with the address 84H; however, it has multiple addresses, as explained below.

5.3.2 Hardware

Figure 5.11 shows the 74LS244 tri-state octal buffer used as an interfacing device. The device has two groups of four buffers each, and they are controlled by the active low sig-

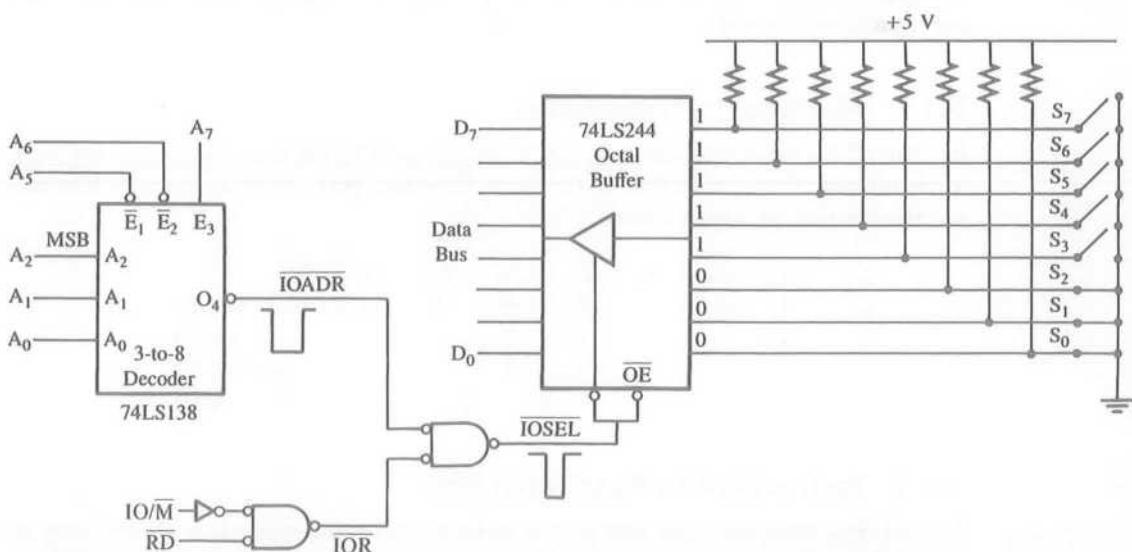


FIGURE 5.11
Interfacing DIP Switches

nals \overline{OE} . When \overline{OE} is low, the input data show up on the output lines (connected to the data bus), and when \overline{OE} is high, the output lines assume the high impedance state.

5.3.3 Interfacing Circuit

Figure 5.11 shows that the low-order address bus, except the lines A_4 and A_3 , is connected to the decoder (the 74LS138); the address lines A_4 and A_3 are left in the don't care state. The output line O_4 of the decoder goes low when the address bus has the following address (assume the don't care lines are at logic 0):

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	$= 84H$
1	0	0	0	0	1	0	0	
↓	↓	↓	↓	↓	↓	↓	↓	
Enable Lines			Don't Care			Input		

The control signal I/O Read (\overline{IOR}) is generated by ANDing the $\overline{IO/M}$ (through an inverter) and RD in a negative NAND gate, and the I/O select pulse is generated by ANDing the output of the decoder and the control signal \overline{IOR} . When the address is 84H and the control signal \overline{IOR} is asserted, the I/O select pulse enables the tri-state buffer and the logic levels of the switches are placed on the data bus. The 8085, then, begins to read switch positions during T_3 (Figure 5.2) and places the reading in the accumulator. When a switch is closed, it has logic 0, and when it is open, it is tied to +5 V, representing logic 1. Figure 5.11 shows that the switches S_7-S_3 are open and S_2-S_0 are closed; thus, the input reading will be F8H.

5.3.4 Multiple Port Addresses

In Figure 5.11, the address lines A_4 and A_3 are not used by the decoding circuit; the logic levels on these lines can be 0 or 1. Therefore, this input port can be accessed by four different addresses, as shown below.

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	$= 84H$
1	0	0	0	0	1	0	0	
			0	1				$= 8CH$
			1	0				$= 94H$
			1	1				$= 9CH$

5.3.5 Instructions to Read Input Port

To read data from the input port shown in Figure 5.11, the instruction IN 84H can be used. When this instruction is executed, during the M_3 cycle, the 8085 places the address 84H on the low-order bus (as well as on the high-order bus), asserts the RD control signal, and reads the switch positions.

MEMORY-MAPPED I/O

5.4

In memory-mapped I/O, the input and output devices are assigned and identified by 16-bit addresses. To transfer data between the MPU and I/O devices, memory-related instructions (such as LDA, STA, etc.)* and memory control signals (MEMR and MEMW) are used. The microprocessor communicates with an I/O device as if it were one of the memory locations. The memory-mapped I/O technique is similar in many ways to the peripheral I/O technique. To understand the similarities, it is necessary to review how a data byte is transferred from the 8085 microprocessor to a memory location or vice versa. For example, the following instruction will transfer the contents of the accumulator to the memory location 8000H.

Memory Address	Machine Code	Mnemonics	Comments
2050	32	STA 8000H	;Store contents of accumulator in memory location 8000H
2051	00		
2052	80		

(Note: It is assumed here that the instruction is stored in memory locations 2050H, 51H, and 52H.)

The STA is a three-byte instruction; the first byte is the opcode, and the second and third bytes specify the memory address. However, the 16-bit address 8000H is entered in the reverse order; the low-order byte 00 is stored in location 2051, followed by the high-order address 80H (the reason for the reversed order will be explained in Section 5.6). In this example, if an output device, instead of a memory register, is connected at this address, the accumulator contents will be transferred to the output device. This is called the **memory-mapped I/O technique**.

On the other hand, the instruction LDA (Load Accumulator Direct) transfers the data from a memory location to the accumulator. The instruction LDA is a 3-byte instruction; the second and third bytes specify the memory location. In the memory-mapped I/O technique, an input device (keyboard) is connected instead of a memory. The input device will have the 16-bit address specified by the LDA instruction. When the microprocessor executes the LDA instruction, the accumulator receives data from the input device rather than from a memory location. To use memory-related instructions for data transfer, the control signals Memory Read (MEMR) and Memory Write (MEMW) should be connected to I/O devices instead of IOR and IOW signals, and the 16-bit address bus ($A_{15}-A_0$) should be decoded. The hardware details will be described in Section 5.4.2).

*In addition to the instructions STA and LDA, other memory-related data transfer instructions, such as MOV M, LDAX, and STAX, also can be used for memory-mapped I/O. These instructions will not be discussed here to maintain clarity in presenting the concepts of memory-mapped I/O.

5.4.1 Execution of Memory-Related Data Transfer Instructions

The execution of memory-related data transfer instructions is similar to the execution of IN or OUT instructions, except that the memory-related instructions have 16-bit addresses. The microprocessor requires four machine cycles (13 T-states) to execute the instruction STA (Figure 5.12). The machine cycle M_4 for the STA instruction is similar to the machine cycle M_3 for the OUT instruction.

For example, to execute the instruction STA 8000H in the fourth machine cycle (M_4), the microprocessor places memory address 8000H on the entire address bus ($A_{15}-A_0$). The accumulator contents are sent on the data bus, followed by the control signal Memory Write MEMW (active low).

On the other hand, in executing the OUT instruction (Figure 5.1), the 8-bit device address is repeated on the low-order address bus (A_0-A_7) as well as on the high-order bus, and the IOW control signal is used. To identify an output device, either the low-order or the high-order bus can be decoded. In the case of the STA instruction, the entire bus must be decoded.

Device selection and data transfer in memory-mapped I/O require three steps that are similar to those required in peripheral I/O:

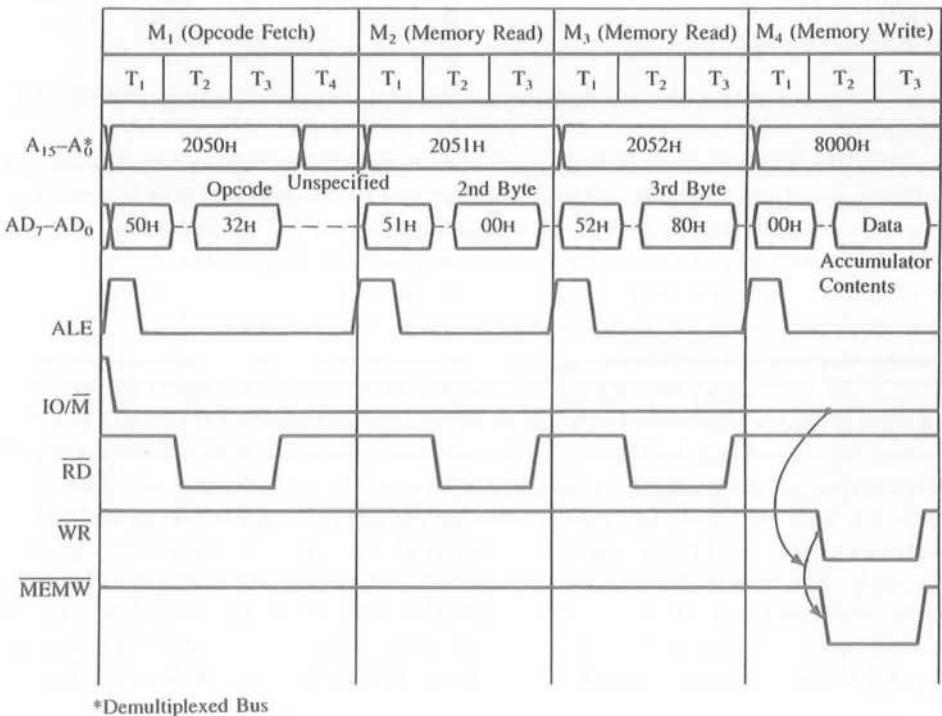


FIGURE 5.12
Timing for Execution of the Instruction: STA 8000H

1. Decode the address bus to generate the device address pulse.
2. AND the control signal with the device address pulse to generate the device select (I/O select) pulse.
3. Use the device select pulse to enable the I/O port.

To interface a memory-mapped input port, we can use the instruction LDA 16-bit, which reads data from an input port with the 16-bit address and places the data in the accumulator. The instruction has four machine cycles; only the fourth machine cycle differs from M_4 in Figure 5.12. The control signal will be RD rather than WR, and data flow from the input port to the microprocessor.

5.4.2 Illustration: Safety Control System Using Memory-Mapped I/O Technique

Figure 5.13 shows a schematic of interfacing I/O devices using the memory-mapped I/O technique. The circuit includes one input port with eight DIP switches and one output port to control various processes and gates, which are turned on/off by the microprocessor according to the corresponding switch positions. For example, switch S_7 controls the cooling system, and switch S_0 controls the exit gate. All switch inputs are tied high; therefore, when a switch is open (off), it has +5 V, and when a switch is closed (on), it has logic 0. The circuit includes one 3-to-8 decoder, one 8-input NAND gate, and one 4-input NAND gate to decode the address bus. The output O_0 of the decoder is combined with control signal MEMW to generate the device select pulse that enables the octal latch. The output O_1 is combined with the control signal MEMR to enable the input port. The eight switches are interfaced using a tri-state buffer 74LS244, and the solid state relays controlling various processes are interfaced using an octal latch (74LS373) with tri-state output.

OUTPUT PORT AND ITS ADDRESS

The various process control devices are connected to the data bus through the latch 74LS373 and solid state relays. If an output bit of the 74LS373 is high, it activates the corresponding relay and turns on the process; the process remains on until the bit stays high. Therefore, to control these safety processes, we need to supply an appropriate bit pattern to the latch.

The 74LS373 is a latch followed by a tri-state buffer, as shown in Figure 5.13. The latch and the buffer are controlled independently by the Latch Enable (LE) and Output Enable (OE). When LE is high, the data enter the latch, and when LE goes low, data are latched. The latched data are available on the output lines of the 74LS373 if the buffer is enabled by OE (active low). If OE is high, the output lines go into the high impedance state.

Figure 5.13 shows that the \overline{OE} is connected to the ground; thus, the latched data will keep the relays on/off according to the bit pattern. The LE is connected to the device

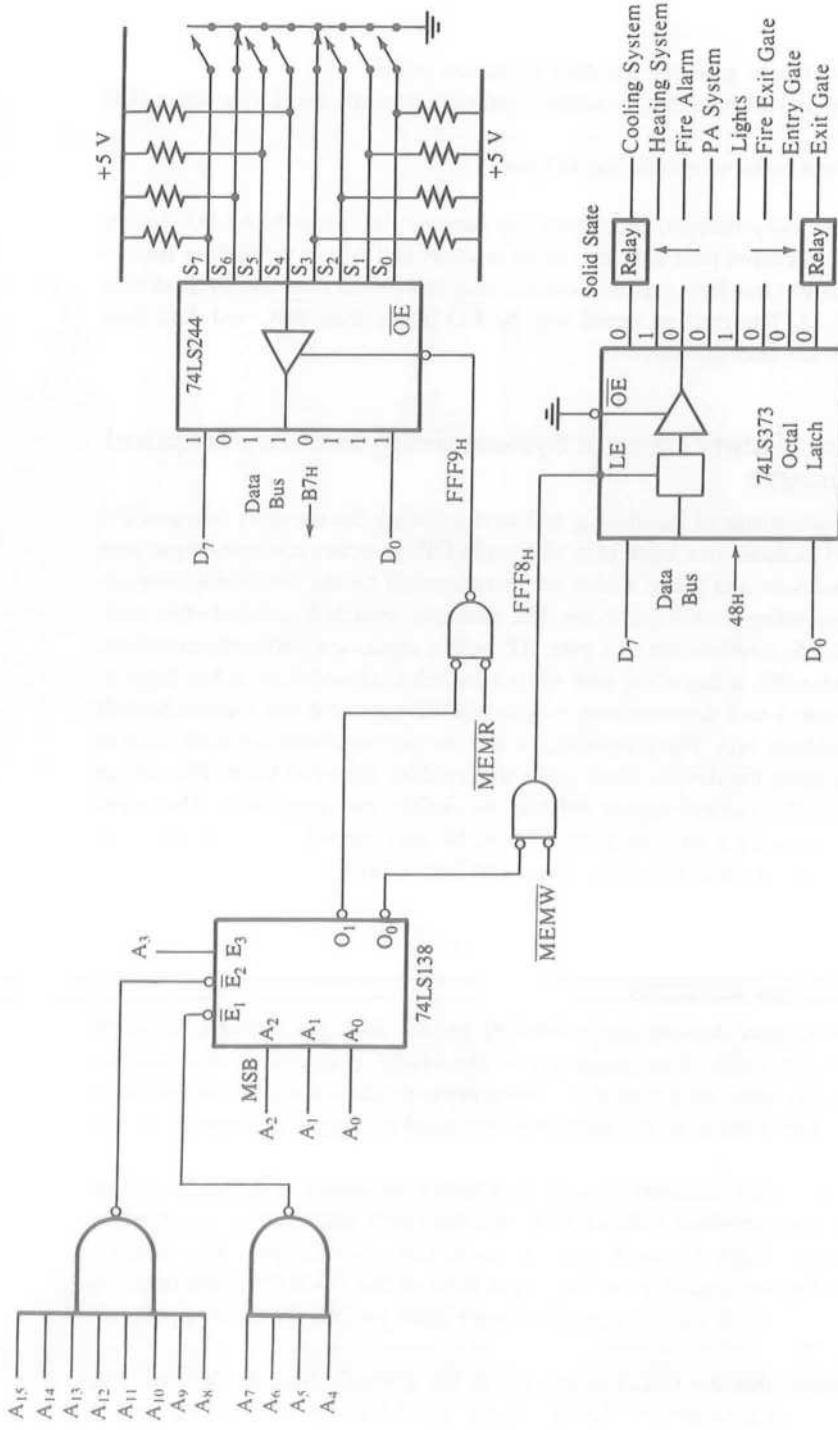
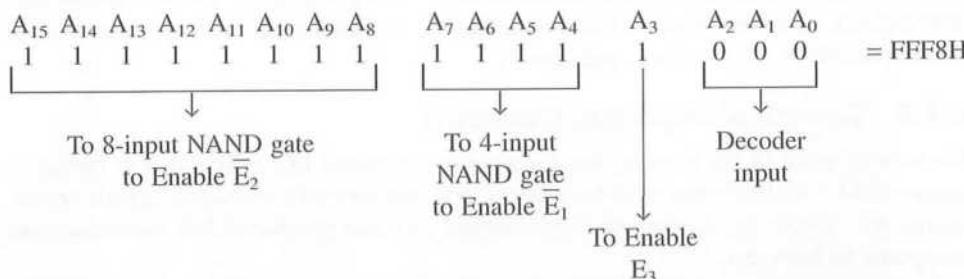


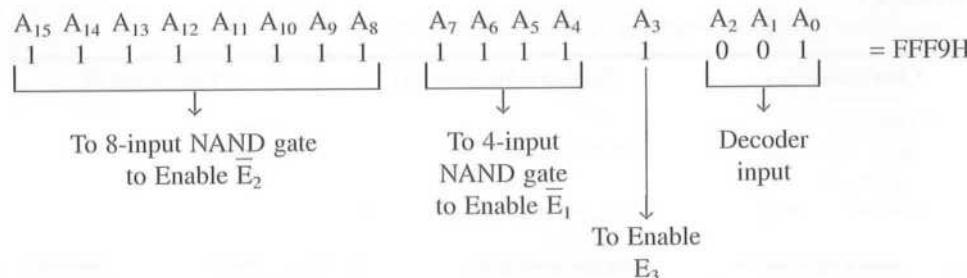
FIGURE 5.13
Memory-Mapped I/O Interfacing

select pulse, which is asserted when the output O_0 of the decoder and the control signal $\overline{\text{MEMW}}$ go low. Therefore, to assert the I/O select pulse, the output port address should be FFF8H, as shown below:



INPUT PORT AND ITS ADDRESS

The DIP switches are interfaced with the 8085 using the tri-state buffer 74LS244. The switches are tied high, and they are turned on by grounding, as shown in Figure 5.13. The switch positions can be read by enabling the signal $\overline{\text{OE}}$, which is asserted when the output O_1 of the decoder and the control signal $\overline{\text{MEMR}}$ go low. Therefore, to read the input port, the port address should be



Instructions To control the processes according to switch positions, the microprocessor should read the bit pattern at the input port and send that bit pattern to the output port. The following instructions can accomplish this task:

READ: LDA FFF9H	;Read the switches
CMA	;Complement switch reading, convert on-switch (logic 0)
	; into logic 1 to turn on appliances
STA FFF8H	;Send switch positions to output port and turn on/off appli-
	; ances
JMP READ	;Go back and read again

When this program is executed, the first instruction reads the bit pattern 1011 0111 (B7H) at the input port FFF9H and places that reading in the accumulator; this bit pattern represents the on-position of switches S₆ and S₃. The second instruction complements the

reading; this instruction is necessary because the on-position has logic 0, and to turn on solid state relays logic 1 is necessary. The third instruction sends the complemented accumulator contents ($0100\ 1000 = 48H$) to the output port FFF8H. The 74LS373 latches the data byte 0100 1000 and turns on the heating system and lights. The last instruction, JMP READ, takes the program back to the beginning and repeats the loop continuously. Thus, it monitors the switches continuously.

5.4.3 Review of Important Concepts

Memory-mapped I/O is in many ways similar to peripheral I/O, except that in memory-mapped I/O, the device has a 16-bit address, and memory-related control signals are required to identify the device. Memory-mapped I/O and peripheral I/O techniques are compared in Table 5.1.

An examination of Table 5.1 shows that the selection of the I/O technique will be determined primarily by the type of application; the advantages seem to balance the disadvantages. In systems in which 64K memory is a requirement, peripheral I/O becomes essential; on the other hand, in control applications in which the number of I/Os exceed the limit (256) and direct data manipulation is preferred, memory-mapped I/O may have an advantage.

TABLE 5.1
Comparison of Memory-Mapped I/O and Peripheral I/O

Characteristics	Memory-Mapped I/O	Peripheral I/O
1. Device address	16-bit	8-bit
2. Control signals for Input/Output	MEMR/MEMW	IOR/IOW
3. Instructions available	Memory-related instructions such as STA; LDA; LDAX; STAX; MOV M,R; ADD M; SUB M; ANA M; etc.	IN and OUT
4. Data transfer	Between any register and I/O	Only between I/O and the accumulator
5. Maximum number of I/Os possible	The memory map (64K) is shared between I/Os and system memory	The I/O map is independent of the memory map; 256 input devices and 256 output devices can be connected
6. Execution speed	13 T-states (STA,LDA) 7 T-states (MOV M,R)	10 T-states
7. Hardware requirements	More hardware is needed to decode 16-bit address	Less hardware is needed to decode 8-bit address
8. Other features	Arithmetic or logical operations can be directly performed with I/O data	Not available

TESTING AND TROUBLESHOOTING I/O INTERFACING CIRCUITS

5.5

In previous sections, we illustrated how to interface an I/O device to a working microcomputer system or add an I/O port as an expansion to the existing system. In Section 5.2.2, we designed the LED output port with the address F5H. The next step is to test and verify that we can display the digit 7 by sending the code 78H as specified in the design problem. In the first attempt, the most probable outcome will be that nothing is displayed or digit 8 is displayed irrespective of the code sent to the port.

Now we need to troubleshoot the interfacing circuit. The obvious first step is to check the wiring and the pin connections. After this preliminary check, we need to generate a constant and identifiable signal and check various points in relation to that signal. We can generate such a signal by asking the processor to execute a continuous loop, called a diagnostic routine, as discussed in Chapter 4 (Section 4.6).

5.5.1 Diagnostic Routine and Machine Cycles

We can use the same instructions for the diagnostic routine that we used in the design problem; however, to generate a continuous signal, we need to add a Jump instruction, as shown next.

Instruction	Bytes	T-States	Machine Cycles		
			M ₁	M ₂	M ₃
START: MVI A,78H	2	7 (4,3)	Opcode Fetch	Memory Read	
OUT F5H	3	10 (4,3,3)	Opcode Fetch	Memory Read	I/O Write
JMP START	3	10 (4,3,3)	Opcode Fetch	Memory Read	Memory Read

This loop has 27 T-states and eight operations (machine cycles). To execute the loop once, the microprocessor asserts the RD signal seven times (the Opcode Fetch is also a Read operation) and the WR signal once. Assuming the system clock frequency is 3 MHz, the loop is executed in 8.9 μ s, and the WR signal is repeated every 8.9 μ s that can be observed on a scope. If we sync the scope on the WR pulse from the 8085, we can check the output of the decoder, IOW, and IOSEL signals; some of these signals of a working circuit are shown in Figure 5.14.

When the 8085 asserts the WR signal, the port address F5H must be on the address bus A₇-A₀, and the output O₅ of the decoder in Figure 5.9 must be low. Similarly, the IOW must be low and the IOSEL (the output of the 74LS02) must be high. Now if we check the data bus in relation to the WR signal, one line at a time, we must read the data byte 78H. If the circuit is not properly functioning, we can check various signals in reference to the WR signal as suggested below:

1. If IOSEL is low, check IOW and O₅ of the decoder.
2. If IOW is high, check the input to the OR gate 74LS32. Both should be low.

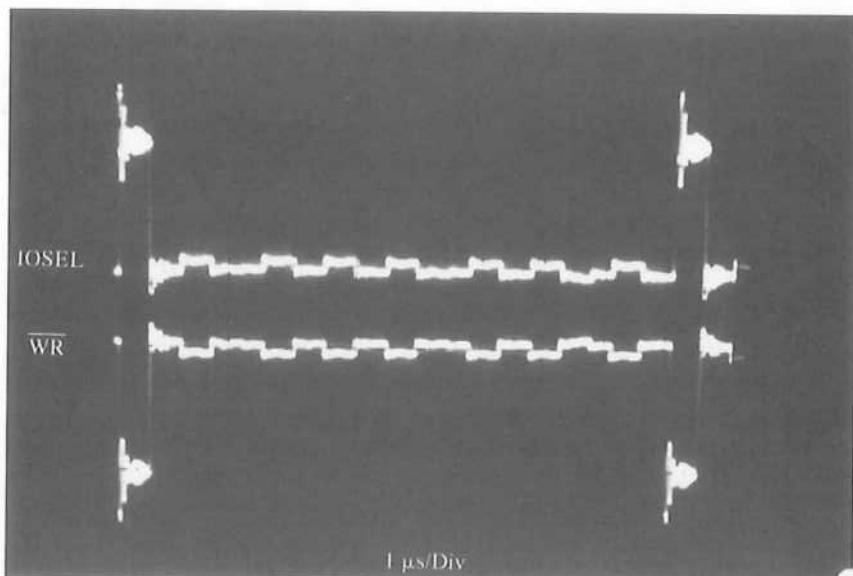


FIGURE 5.14
Timing Signals of Diagnostic Routine

3. If O_5 of the decoder is high, check all the output lines O_0 to O_7 of the decoder. If all of them are high, that means the decoder is not enabled. If one of the outputs of the decoder is low, it suggests that the input address lines are improperly connected.
4. If the decoder is not enabled, check the address lines A_4 – A_7 ; all of them must be high and the address line A_3 must be low.
5. Another possibility is that the port is enabled, but the seven-segment display is wrong. The problem must be with data lines. Try different codes to display other digits. If two data lines are interchanged, you may be able to isolate these two data lines. The final step is to check all the data lines.

5.6

SOME QUESTIONS AND ANSWERS

During the discussion of I/O interfacing, we focused on the basic concepts and avoided some details to maintain the clarity. We will attempt to answer those questions.

1. *Can an input port and an output port have the same port address?*
Yes. They will be differentiated by control signals. The RD is used to enable the input port and the WR is used to enable the output port.
2. *How will the port number be affected if we decode the high-order address lines A_{15} – A_8 rather than A_7 – A_0 ?*

The port address will remain the same because the I/O port address is duplicated on both segments of the address bus.

3. *If high-order lines are partially decoded, how can one determine whether it is peripheral I/O or memory-mapped I/O?*

To recognize the type of I/O, examine the control signal. If the control signal is IOW (or IOR), it must be a peripheral I/O, and if the control signal is MEMW (or MEMR), it must be a memory-mapped I/O (see Problem 14).

5. *In a memory-mapped I/O, how does the microprocessor differentiate between an I/O and memory? Can an I/O have the same address as a memory register?*

In memory-mapped I/O, the microprocessor cannot differentiate between an I/O and memory; it treats an I/O as if it is memory. Therefore, an I/O and memory register cannot have the same address; the entire memory map (64K) of the system has to be shared between memory and I/O.

5. *Why is a 16-bit address (data) stored in memory in the reversed order—the low-order byte first, followed by the high-order byte?*

This has to do with the design of the 8085 microprocessor. The instruction decoder or the associated microprogram is designed to recognize the second byte as the low-order byte in a three-byte instruction.

SUMMARY

In this chapter, we examined the machine cycles of the OUT and IN instructions and derived the basic concepts in interfacing peripheral-mapped I/Os. Similarly, we examined the machine cycles of memory-related data transfer instructions and derived the basic concepts in interfacing memory-mapped I/Os. These concepts were illustrated with various examples of interfacing I/O devices. The interfacing concepts can be summarized as follows.

Peripheral-Mapped I/O

- The OUT is a two-byte instruction. It copies (transfers or sends) data from the accumulator to the addressed port.
- When the 8085 executes the OUT instruction, in the third machine cycle, it places the output port address on the low-order bus, duplicates the same port address on the high-order bus, places the contents of the accumulator on the data bus, and asserts the control signal WR.
- A latch is commonly used to interface output devices.
- The IN instruction is a two-byte instruction. It copies (transfers or reads) data from an input port and places the data into the accumulator.
- When the 8085 executes the IN instruction, in the third machine cycle, it places the input port address on the low-order bus, as well as on the high-order bus, asserts the control signal, RD, and transfers data from the port to the accumulator.
- A tri-state buffer is commonly used to interface input devices.

- To interface an output or an input device, the low-order address bus A₇-A₀ (or high-order bus A₁₅-A₈) needs to be decoded to generate the device address pulse, which must be combined with the control signal IOR (or IOW) to select the device.

Memory-Mapped I/O

- Memory-related instructions are used to transfer data.
- To interface I/O devices, the entire bus must be decoded to generate the device address pulse, which must be combined with the control signal MEMR (or MEMW) to generate the I/O select pulse. This pulse is used to enable the I/O device and transfer the data.

QUESTIONS AND PROBLEMS

1. Explain why the number of output ports in the peripheral-mapped I/O is restricted to 256 ports.
2. In the peripheral-mapped I/O, can an input port and an output port have the same port address?
3. If an output and input port can have the same 8-bit address, how does the 8085 differentiate between the ports?
4. Specify the two 8085 signals that are used to latch data in an output port.
5. Specify the type of pulse (high or low) required to latch data in the 7475.
6. Are data latched in the 7475 at the leading edge, during the level, or at the trailing edge of the enable (E) signal?
7. In Figure 5.8, explain why the LED cathodes rather than anodes are connected to the latch.
8. Specify the 8085 signals that are used to enable an input port.
9. Explain why a latch is used for an output port, but a tri-state buffer can be used for an input port.
10. What are the control signals necessary in the memory-mapped I/O?
11. Can the microprocessor differentiate whether it is reading from a memory-mapped input port or from memory?
12. Identify the port address in Figure 5.15.
13. In Figure 5.15, if \overline{OE} is connected directly to the WR signal and the output of the decoder is connected to the latch enable (through an inverter), can you display a byte at the output port? Explain your answer.
14. In Figure 5.16, can you recognize whether it is the memory-mapped or the peripheral-mapped I/O?
15. In Figure 5.16, what is the port address if all the don't care address lines are assumed to be at logic 0?
16. In Figure 5.17, specify the output port address if the output signal O₁ of the decoder is connected to gate 74LS02 instead of the signal O₅.

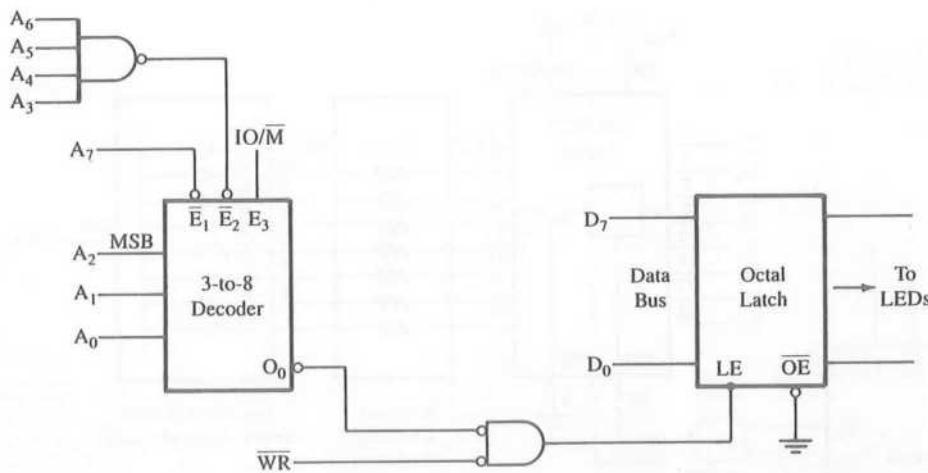


FIGURE 5.15

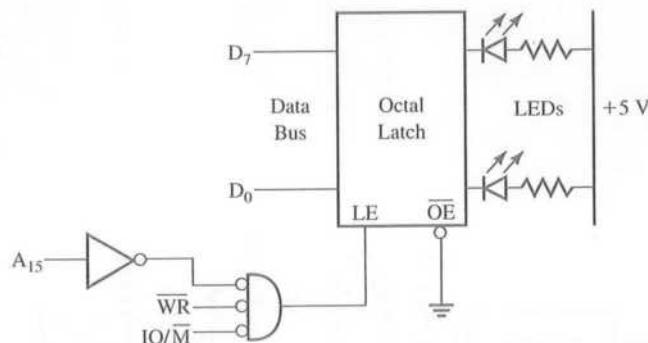


FIGURE 5.16

17. In Figure 5.17, eliminate the gate 74LS32, replace A_7 by the $\overline{IO/M}$ signal, and connect the WR signal directly to gate 74LS02. Specify the port addresses if the address line A_7 is left in don't care state.
18. In Question 17, replace A_3 by the $\overline{IO/M}$ signal instead of A_7 . Specify the port address assuming all don't care lines (including A_3) at logic 0. Replace instructions shown in Section 5.2.2 to display the digit "7" by appropriate instructions.
19. Write instructions to display digit "0" at the output port in Figure 5.17.
20. Write instructions to display letter "H" at the output port in Figure 5.17.
21. In Figure 5.18, eliminate the negative NAND gate that generates the \overline{IOR} signal, replace A_7 by the $\overline{IO/M}$ signal, and connect RD directly to the negative NAND gate. Identify all the port addresses.
22. In Figure 5.15, exchange A_7 with $\overline{IO/M}$ signal. Identify the port address assuming any don't care lines at logic 0.

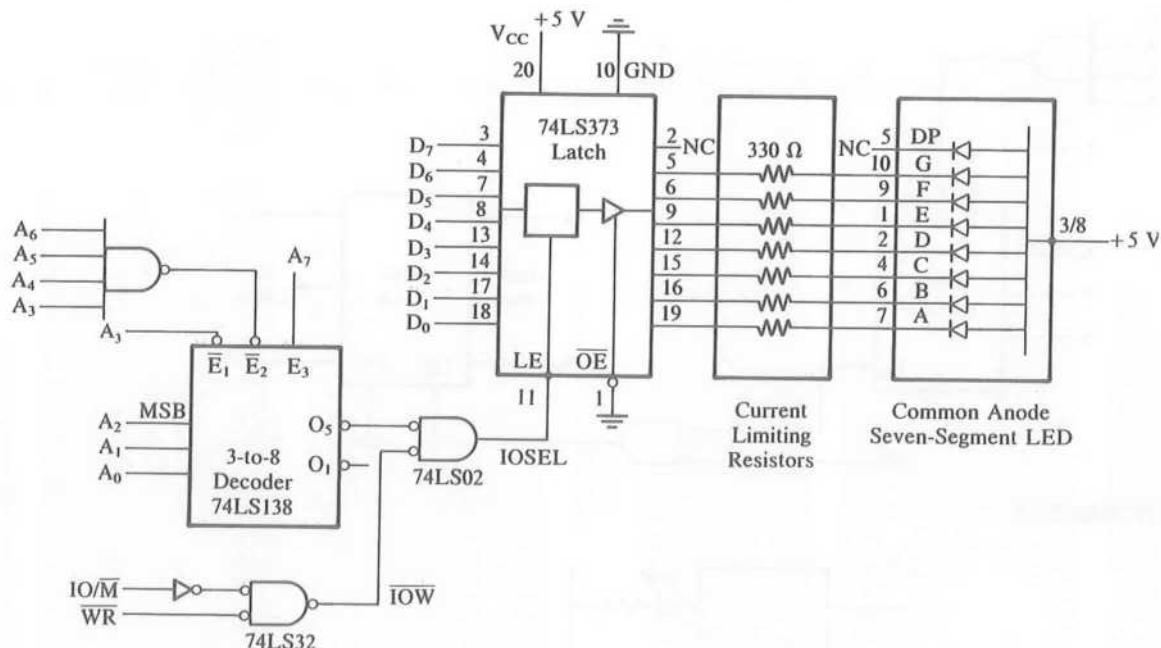


FIGURE 5.17

Interfacing Seven-Segment LED

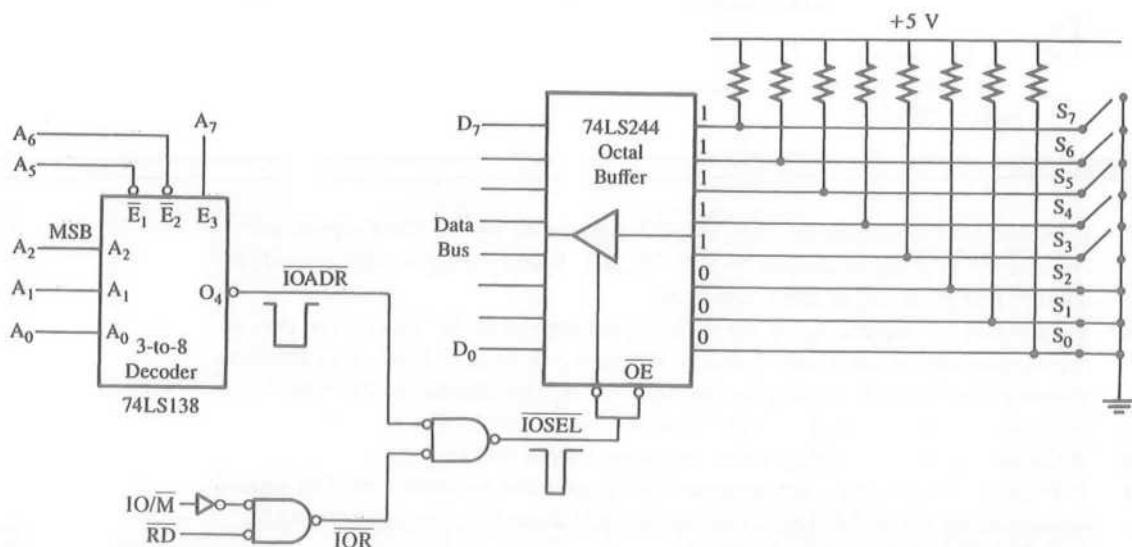


FIGURE 5.18

Interfacing DIP Switches

23. In Figure 5.19, identify ports A and B as input or output ports.
24. In Figure 5.19, what are the addresses of ports A and B?
25. In Figure 5.17, explain why the pulse width of the output O_5 of the decoder is larger than the \overline{IOW} signal.
26. The following diagnostic routine can be used to troubleshoot the interfacing circuit of an input port such as in Figure 5.11.

Instruction	Bytes	T-States	Machine Cycles		
			M ₁	M ₂	M ₃
START: IN 84H	2	10 (4,3,3)			
JMP START	3	10 (4,3,3)			

- a. Identify the machine cycles.
- b. If the system clock is 2 MHz, calculate the time required to execute the routine.
- c. Specify the number of times the \overline{RD} signal is asserted if the loop is executed once.
- d. If the loop is executed continuously, specify the time between two consecutive IO/M signals that are high.
- e. Is there a \overline{WR} pulse in the diagnostic routine? If the answer is no, what is the unique identifiable signal that can be used to sync the scope?

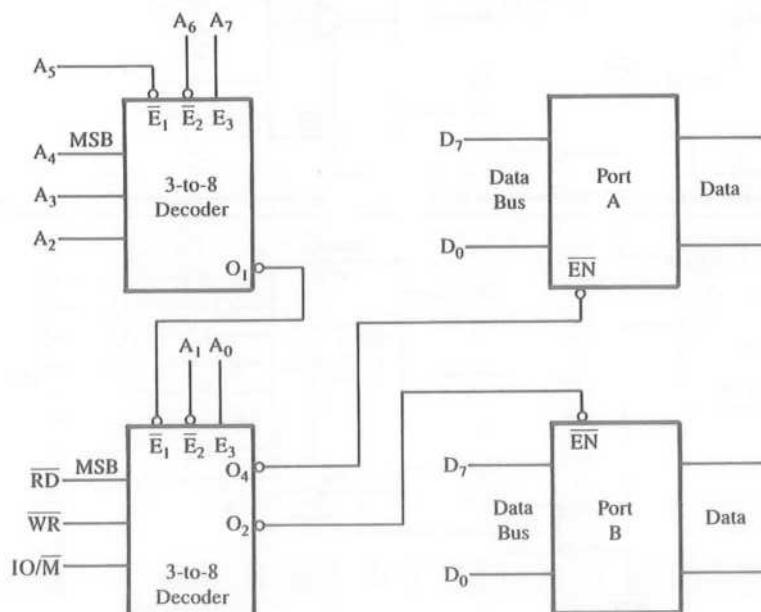


FIGURE 5.19

27. Diagnostic Routine for Figure 5.13:

Instruction	Bytes	T-States	Machine Cycles			
			M ₁	M ₂	M ₃	M ₄
START: LDA FFF9H	3	13 (4,3,3,3)				
STA FFF8H	3	13 (4,3,3,3)				
MOV B,A	1	4				
JMP START	3	10 (4,3,3)				

- a. Identify the machine cycles of each instruction.
 - b. Specify the contents of the address bus in the fourth machine cycle of the LDA instruction.
 - c. Specify the number of RD and WR signals in one loop.
 - d. If the system frequency is 2 MHz, calculate the time period between two consecutive MEMW signals.
28. In Figure 5.20, identify the addresses of the input and the output ports.

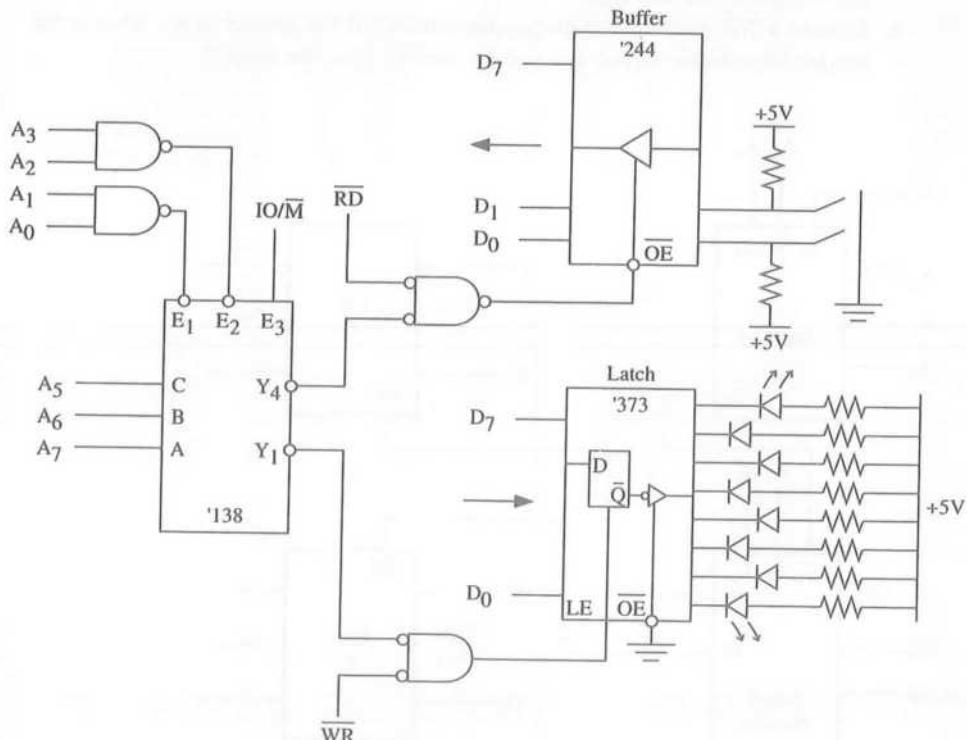


FIGURE 5.20

29. Write instructions to read the input port (Figure 5.20) and continue to read it until both switches are closed (by an operator). When both switches are closed, turn on all the LEDs.
30. In Figure 5.20, write instructions to read the input port and continue to read it until at least one switch is closed. When a switch is closed, turn on the corresponding LED.
31. In Figure 5.20, assume that both switches are normally closed. Write instructions to read the input port and continue to read the port until at least one of the switches is open. Once a switch is opened, turn on the corresponding LED.
32. In Figure 5.21, write instructions to display the number “97” at the common-anode seven-segment LED port.

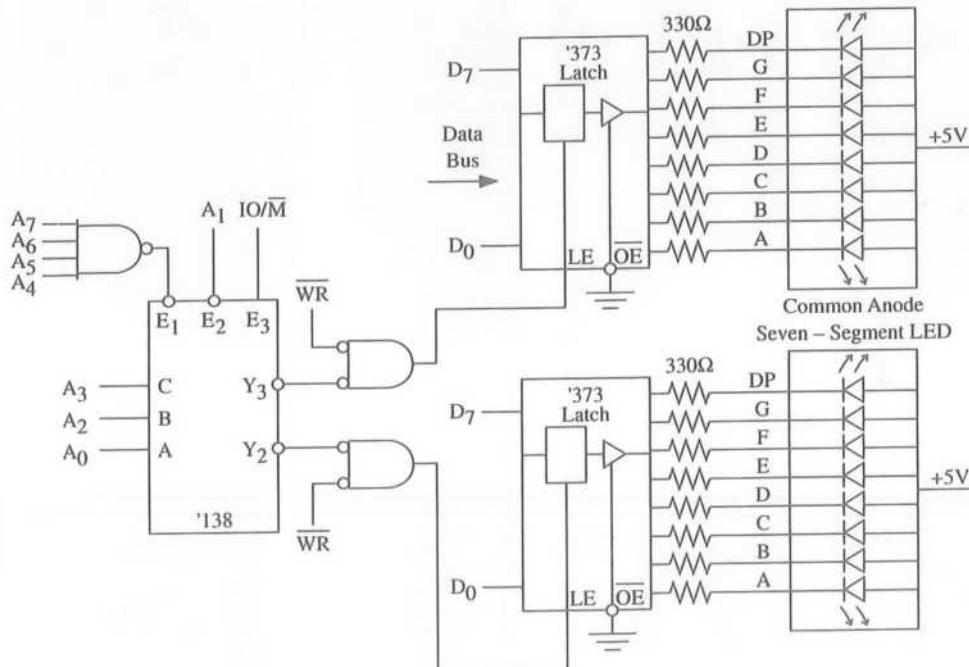


FIGURE 5.21

II

Programming the 8085

CHAPTER 6

Introduction to 8085 Instructions

CHAPTER 7

Programming Techniques with Additional Instructions

CHAPTER 8

Counters and Time Delays

CHAPTER 9

Stack and Subroutines

CHAPTER 10

Code Conversion, BCD Arithmetic, and 16-Bit Data Operations

CHAPTER 11

Software Development Systems and Assemblers

Part II of this book is an introduction to assembly language programming for the 8085. It explains commonly used instructions, elementary programming techniques, and their applications.

The content is presented in a format as if for teaching a foreign language. One approach to learning a foreign language is to begin with a few words that can form simple, meaningful, and interactive sentences. After learning a few sentences, the student begins to write a paragraph that can convey an idea in a coherent fashion; then, by sequencing a few paragraphs, begins to compose a letter. Chapters 6 to 11 are arranged in similar fashion—from simple instructions to applications.

Chapter 2 provided an overview of the 8085 instruction set. Chapters 6 and 7 are concerned primarily with the instructions that occur most frequently. The instructions are not introduced accord-

ing to the five groups as classified in Chapter 2; instead, a few instructions that can perform a simple task are selected from each group. Chapter 6 includes the discussion of instructions from each group—from data copy to branch instructions. Chapter 7 introduces elementary programming techniques such as looping and indexing. Chapter 8 uses the instructions and techniques presented in these chapters to design software delays and counters.

Chapter 9 introduces the concepts of subroutine and stack, which provide flexibility and variety for program design. Chapter 10 includes applications of the concepts presented in Chapter 9, presenting techniques for writing programs concerned with code conversions, and arithmetic routines. Chapter 11 deals with the uses of assemblers and software development systems.

PREREQUISITES

The reader is expected to know the following topics:

The 8085 architecture, especially the programming registers.

- The concepts related to memory and I/Os.
- Logic operations and binary and hexadecimal arithmetic.

6

Introduction to 8085 Instructions

A microcomputer performs a task by reading and executing the set of instructions written in its memory. This set of instructions, written in a sequence, is called a **program**. Each instruction in the program is a command, in binary, to the microprocessor to perform an operation. This chapter introduces 8085 basic instructions, their operations, and their applications.

Chapters 3 and 4 described the architecture of the 8085 microprocessor and Chapter 2 provided an overview of the instruction set and the tasks the 8085 can perform. This chapter is concerned with using instructions within the constraints and capabilities of its registers and the bus system. A few instructions are introduced from each of the five groups (Data Transfer, Arithmetic, Logical, Branch, and Machine Control) and are used to write simple programs to perform specific tasks.

The simple illustrative programs given in this chapter can be entered and executed on the single-

board microcomputers used commonly in college laboratories.

OBJECTIVES

- Explain the functions of data transfer (copy) instructions and how the contents of the source register and the destination register are affected.
- Explain the Input/Output instructions and port addresses.
- Explain the functions of the machine control instructions HLT and NOP.
- Recognize the addressing modes of the instructions.
- Draw a flowchart of a simple program.
- Write a program in 8085 mnemonics to illustrate an application of data copy instructions, and translate those mnemonics manually into their Hex codes.

- Write a program in the proper format showing memory addresses, Hex machine codes, mnemonics, and comments.
- Explain the arithmetic instructions, and recognize the flags that are set or reset for given data conditions.
- Write a set of instructions to perform an addition and a subtraction (in 2's complement).
- Explain the logic instructions, and recognize the flags that are set or reset for given data conditions.
- Write a set of instructions to illustrate logic operations.
- Explain the use of logic instructions in masking, setting, and resetting individual bits.
- Explain the unconditional and conditional Jump instructions and how flags are used by the conditional Jump instructions to change the sequence of a program.
- Write a program to illustrate an application of Jump instructions.
- List the important steps in writing and troubleshooting a simple program.

6.1

DATA TRANSFER (COPY) OPERATIONS

One of the primary functions of the microprocessor is copying data, from a register (or I/O or memory) called the source, to another register (or I/O or memory) called the destination. In technical literature, the copying function is frequently labeled as the **data transfer function**, which is somewhat misleading. In fact, the contents of the source are not transferred, but are copied into the destination register without modifying the contents of the source.

Several instructions are used to copy data (as listed in Chapter 2). This section is concerned with the following operations.

MOV : Move	Copy a data byte.
MVI : Move Immediate	Load a data byte directly.
OUT : Output to Port	Send a data byte to an output device.
IN : Input from Port	Read a data byte from an input device.

The term *copy* is equally valid for input/output functions because the contents of the source are not altered. However, the term *data transfer* is used so commonly to indicate the data copy function that, in this book, these terms are used interchangeably when the meaning is not ambiguous.

In addition to data copy instructions, it is necessary to introduce two machine-control operations to execute programs.

HLT: Halt	Stop processing and wait.
NOP: No Operation	Do not perform any operation.

These operations (opcodes) are explained and illustrated below with examples.

Instructions The data transfer instructions copy data from a source into a destination without modifying the contents of the source. The previous contents of the destination are replaced by the contents of the source.

Important Note: In the 8085 processor, data transfer instructions do not affect the flags.

Opcode	Operand	Description
MOV	Rd,Rs*	Move <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Copies data from source register Rs to destination register Rd
MVI	R,8-bit*	Move Immediate <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Loads the 8 bits of the second byte into the register specified
OUT	8-bit port address	Output to Port <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Sends (copies) the contents of the accumulator (A) to the output port specified in the second byte
IN	8-bit port address	Input from Port <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Accepts (reads) data from the input port specified in the second byte, and loads into the accumulator
HLT		Halt <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> The processor stops executing and enters wait state <input type="checkbox"/> The address bus and data bus are placed in high impedance state. No register contents are affected
NOP		No Operation <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> No operation is performed <input type="checkbox"/> Generally used to increase processing time or substitute in place of an instruction. When an error occurs in a program and an instruction needs to be eliminated, it is more convenient to substitute NOP than to reassemble the whole program

*The symbols Rd, Rs, and R are generic terms; they represent any of the 8085 8-bit registers: A, B, C, D, E, H, and L.

**Example
6.1**

Load the accumulator A with the data byte 82H (the letter H indicates hexadecimal number), and save the data in register B.

Instructions MVI A, 82H,
 MOV B,A

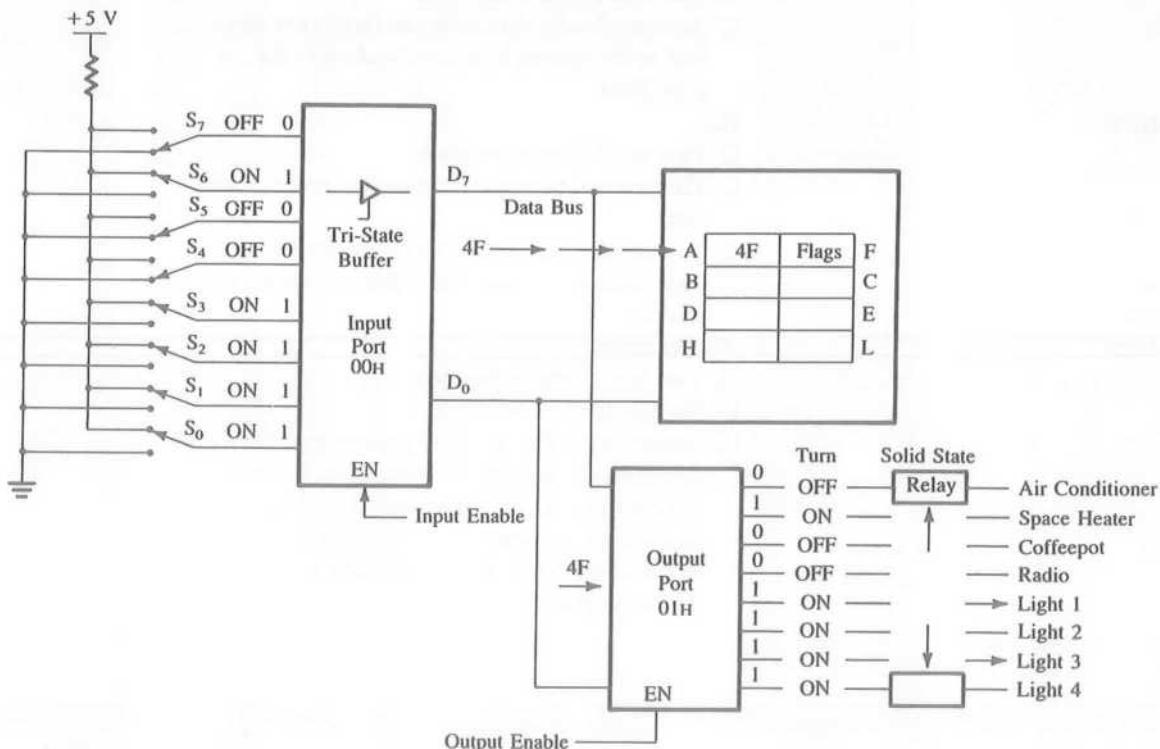
The first instruction is a 2-byte instruction that loads the accumulator with the data byte 82H, and the second instruction MOV B,A copies the contents of the accumulator in register B without changing the contents of the accumulator.

**Example
6.2**

Write instructions to read eight ON/OFF switches connected to the input port with the address 00H, and turn on the devices connected to the output port with the address 01H, as shown in Figure 6.1. (I/O port addresses are given in hexadecimal.)

Solution

The input has eight switches that are connected to the data bus through the tri-state buffer. Any one of the switches can be connected to +5 V (logic 1) or to ground (logic 0), and each switch controls the corresponding device at the output port. The microprocessor needs to read the bit pattern on the switches and send the same bit pattern to the output port to turn on the corresponding devices.

**FIGURE 6.1**

Reading Data at Input Port and Sending Data to Output Port

Instructions IN 00H
 OUT 01H
 HLT

When the microprocessor executes the instruction IN 00H, it enables the tri-state buffer. The bit pattern 4FH formed by the switch positions is placed on the data bus and transferred to the accumulator. This is called reading an input port.

When the microprocessor executes the next instruction, OUT 01H, it places the contents of the accumulator on the data bus and enables the output port 01H. (This is also called writing data to an output port.) The output port latches the bit pattern and turns ON/OFF the devices connected to the port according to the bit pattern. In Figure 6.1, the bit pattern 4FH will turn on the devices connected to the output port data lines D₆, D₃, D₂, D₁, and D₀; the space heater and four light bulbs. To turn off some of the devices and turn on other devices, the bit pattern can be modified by changing the switch positions. For example, to turn on the radio and the coffeepot and turn off all other devices, the switches S₄ and S₅ should be on and the others should be off. The microprocessor will read the bit pattern 0011 0000, and this bit pattern will turn on the radio and the coffeepot and turn off other devices.

The preceding explanation raises two questions:

1. What are the second bytes in the instructions IN and OUT?
2. How are they determined?

In answer to the first question, the second bytes are I/O port addresses. Each I/O port is identified with a number or an address similar to the postal address of a house. The second byte has eight bits, meaning 256 (2^8) combinations; thus 256 input ports and 256 output ports with addresses from 00H to FFH can be connected to the system.

The answer to the second question depends on the logic circuit (called interfacing) used to connect and identify a port by the system designer (see Chapter 5).

6.1.1 Addressing Modes

The above instructions are commands to the microprocessor to copy 8-bit data from a source into a destination. In these instructions, the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are, in fact, operands. The various formats of specifying the operands are called the addressing modes. The 8085 instruction set has the following addressing modes. (Each mode is followed by an example and by the corresponding piece of restaurant conversation from the analogy discussed in Chapter 2.)

1. Immediate Addressing—MVI R,Data (Pass the butter)
2. Register Addressing—MOV Rd,Rs (Pass the bowl)
3. Direct Addressing—IN/OUT Port# (Combination number 17 on the menu)
4. Indirect Addressing—Illustrated in the next chapter (I will have what Susie has)

The classification of the addressing modes is unimportant, except that it provides some clues in understanding mnemonics. For example, in the case of the MVI opcode, the letter I suggests that the second byte is data and not a register. What is important is to become familiar with the instructions. After you study the examples given in this chapter, you will see a pattern begin to emerge.

6.1.2 Illustrative Program: Data Transfer—From Register to Output Port

PROBLEM STATEMENT

Load the hexadecimal number 37H in register B, and display the number at the output port labeled PORT1.

PROBLEM ANALYSIS

This problem is similar to the illustrative program discussed in Section 2.4.1. Even though this is a very simple problem it is necessary to break the problem into small steps and to outline the thinking process in terms of the tasks described in Section 6.1.

STEPS

- Step 1: Load register B with a number.
- Step 2: Send the number to the output port.

QUESTIONS TO BE ASKED

- Is there an instruction to load the register B? YES—MVI B.
- Is there an instruction to send the data from register B to the output port? NO. Review the instruction OUT. This instruction sends data from the accumulator to an output port.
- The solution appears to be as follows: Copy the number from register B into accumulator A.
- Is there an instruction to copy data from one register to another register? YES—MOV Rd,Rs.

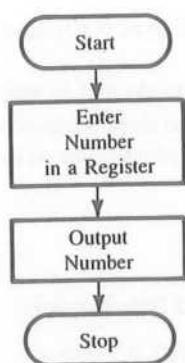
FLOWCHART

The thinking process described here and the steps necessary to write the program can be represented in a pictorial format, called a **flowchart**. Figure 6.2 describes the preceding steps in a flowchart.

Flowcharting is an art. The flowchart in Figure 6.2 does not include all the steps described earlier. Although the number of steps that should be represented in a flowchart is ambiguous, not all of them should be included. That would defeat the purpose of the flowchart. It should represent a logical approach and sequence of steps in solving the problem. A flowchart is similar to the block diagram of a hardware system or to the outline of a chapter. Information in each block of the flowchart should be similar to the heading of a paragraph. Generally, a flowchart is used for two purposes: to assist and clarify the thinking process and to communicate the programmer's thoughts or logic to others.

Symbols commonly used in flowcharting are shown in Figure 6.3. Two types of symbols—rectangles and ovals—are already illustrated in Figure 6.2. The diamond is

FIGURE 6.2
Flowchart



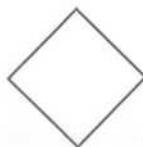
Meaning



Arrow: Indicates the direction of the program execution



Rectangle: Represents a process or an operation



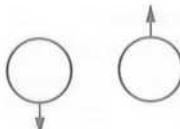
Diamond: Represents a decision-making block



Oval: Indicates the beginning or end of a program



Double-sided rectangle: Represents a predefined process such as a subroutine.



Circle with an arrow: Represents continuation (an entry or exit) to a different page

FIGURE 6.3
Flowcharting Symbols

used with Jump instructions for decision making (see Figure 6.10), and the double-sided rectangle is used for subroutines (see Chapter 9).

The flowchart in Figure 6.2 includes what steps to do and in what sequence. As a rule, a general flowchart does not include how to perform these steps or what registers are being used. The steps described in the flowchart are translated into an assembly language program in the next section.

ASSEMBLY LANGUAGE PROGRAM

Tasks	8085 Mnemonics
1. Load register B with 37H.	MVI B,37H*
2. Copy the number from B to A.	MOV A,B
3. Send the number to the output—port 01H.	OUT PORT1
4. End of the program.	HLT

TRANSLATION FROM ASSEMBLY LANGUAGE TO MACHINE LANGUAGE

Now, to translate the assembly language program into machine language, look up the hexadecimal machine codes for each instruction in the 8085 instruction set and write each machine code in the sequence, as follows:

8085 Mnemonics	Hex Machine Code
1. MVI B,37H	06
	37
2. MOV A,B	78
3. OUT PORT1	D3
	01
4. HLT	76

This program has six machine codes and will require six bytes of memory to enter the program into your system. If your single-board microcomputer has R/W memory starting at the address 2000H, this program can be entered in the memory locations 2000H to 2005H. The format generally used to write an assembly language program is shown below.

PROGRAM FORMAT

Memory Address (Hex)	Machine Code (Hex)	Instruction		Comments
Address (Hex)	Code (Hex)	Opcode	Operand	Comments
XX00 [†]	06	MVI	B,37H	;Load register B with data 37H
XX01	37			

*A number followed by the letter H represents a hexadecimal number.

[†]Enter high-order address (page number) of your R/W memory in place of XX.

XX02	78	MOV	A,B	;Copy (B) into (A)
XX03	D3	OUT	PORT1	;Display accumulator contents
XX04	PORT1*			; (37H) at Port1
XX05	76	HLT		;End of the program

This program has five columns: Memory Address, Machine Code, Opcode, Operand, and Comments. Each is described in the context of a single-board microcomputer.

Memory Address These are 16-bit addresses of the user (R/W) memory in the system, where the machine code of the program is stored. The beginning address is shown as XX00; the symbol XX represents the page number of the available R/W memory in the microcomputer, and 00 represents the line number. For example, if the microcomputer has the user memory at 2000H, the symbol XX represents page number 20H; if the user memory begins at 0300H, the symbol XX represents page 03H. Substitute the appropriate page when entering the machine code of a program.

Machine Code These are the hexadecimal numbers (instruction codes) that are entered (or stored) in the respective memory addresses through the hexadecimal keyboard of the microcomputer. The monitor program, which is stored in Read-Only memory (ROM) of the microcomputer, translates the Hex numbers into binary digits and stores the binary digits in the R/W memory.

If the system has R/W memory with the starting address at 2000H and the output port address 01H, the program will be stored as follows:

Memory Address	Memory Contents	Hex Code
2000	0 0 0 0 0 1 1 0	→ 06
2001	0 0 1 1 0 1 1 1	→ 37
2002	0 1 1 1 1 0 0 0	→ 78
2003	1 1 0 1 0 0 1 1	→ D3
2004	0 0 0 0 0 0 0 1	→ 01
2005	0 1 1 1 0 1 1 0	→ 76

Opcode (Operation Code) An instruction is divided into two parts: Opcode and Operand. Opcodes are the abbreviated symbols specified by the manufacturer (Intel) to indicate the type of operation or function that will be performed by the machine code.

Operand The operand part of an instruction specifies the item to be processed; it can be 8- or 16-bit data, a register, or a memory address.

*Enter the output port address of your system. If an output port is not available on your system, see "How to Execute a Program without an Output Port" later in this section.

An instruction, called a mnemonic or mnemonic instruction, is formed by combining an opcode and an operand. The mnemonics are used to write programs in the 8085 assembly language; and then the mnemonics in these programs are translated manually into the binary machine code by looking them up in the instruction set.

Comments The comments are written as a part of the proper documentation of a program to explain or elaborate the purpose of the instructions used. These are separated by a semi-colon (;) from the instruction on the same line. They play a critical role in the user's understanding of the logic behind a program. Because the illustrative programs in the early part of this chapter are simple, most of the comments are either redundant or trivial. The purpose of the comments in these programs is to reinforce the meaning of the instructions. In actual usage, the comments should not just describe the operation of an instruction.

HOW TO ENTER AND EXECUTE THE PROGRAM

This program assumes that one output port is available on your microcomputer system. The program cannot be executed without modification if your microcomputer has no independent output ports other than the system display of memory address and data or if it has programmable I/O ports. (See Chapter 14.) To enter the program:^{*}

1. Push the Reset key.
2. Enter the 16-bit memory address of the first machine code of your program. (Substitute the page number of your R/W memory for the letters XX and the output port address for the label PORT1.)
3. Enter and store all the machine codes sequentially, using the hexadecimal keyboard on your system.
4. Reset the system.
5. Enter the memory address where the program begins and push the Execute key.

If the program is properly entered and executed, the data byte 37H will be displayed at the output port.

HOW TO EXECUTE A PROGRAM WITHOUT AN OUTPUT PORT

If your system does not have an output port, either eliminate the instruction OUT PORT1, or substitute NOP (No Operation) in place of the OUT instruction. Assuming your system has R/W memory starting at 2000H, you can enter the program as follows:

Memory Address	Machine Code	Mnemonic Instruction
2000	06	MVI B,37H
2001	37	
2002	78	MOV A,B
2003	00	NOP

^{*}Refer to the user's manual of your microcomputer for details.

2004	00	NOP
2005	76	HLT

After you have executed this program, you can find the answer in the accumulator by pushing the Examine Register key (see your user's manual).

The program also can be executed by entering the machine code 76 in location 2003H, thus eliminating the OUT instruction.

6.1.3 Illustrative Program: Data Transfer to Control Output Devices

PROBLEM STATEMENT

A microcomputer is designed to control various appliances and lights in your house. The system has an output port with the address 01H, and various units are connected to the bits D₇ to D₀ as shown in Figure 6.4. On a cool morning you want to turn on the radio, the coffeepot, and the space heater. Write appropriate instructions for the microcomputer. Assume the R/W memory in your system begins at 2000H.

PROBLEM ANALYSIS

The output port in Figure 6.4 is a latch (D flip-flop). When data bits are sent to the output port they are latched by the D flip-flop. A data bit at logic 1 supplies approximately 5 V as output and can turn on solid-state relays.

To turn on the radio, the coffeepot, and the space heater, set D₆, D₅, and D₄ at logic 1, and the other bits at logic 0:

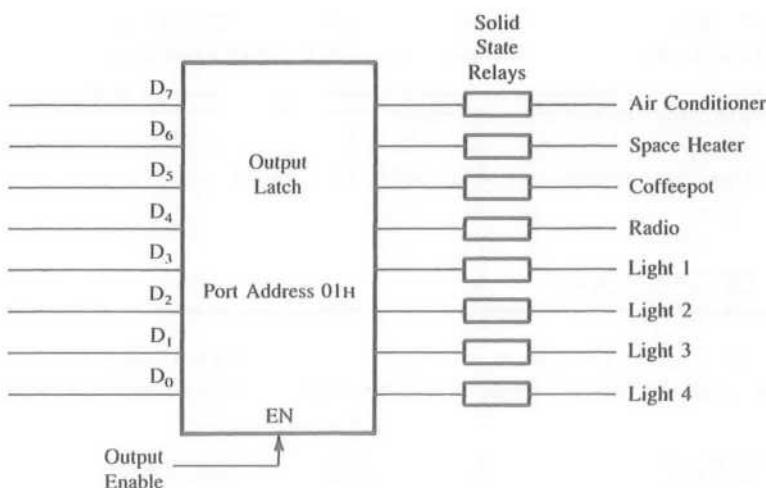


FIGURE 6.4
Output Port to Control Devices

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	1	1	0	0	0	0

$= 70H$

The output port requires 70H, and it can be sent to the port by loading the accumulator with 70H.

PROGRAM

Memory Address	Machine Code	Mnemonic Instruction	Comments
HI-LO*			
2000	3E	MVI A,70H	;Load the accumulator with the bit pattern
2001	70		; necessary to turn on the devices
2002	D3	OUT 01H	;Send the bit pattern to the port 01H, and
2003	01 [†]		; turn on the devices
2004	76	HLT	;End of the program

PROGRAM OUTPUT

This program simulates controlling of the devices connected to the output port by displaying 70H on a seven-segment LED display. If your system has individual LEDs, the binary pattern—0111 0000—will be displayed.

6.1.4 Review of Important Concepts

- Registers are used to load data directly or to save data bytes.
- In data transfer (copying), the destination register is modified but the source register retains its data.
- The 8085 transfers data from an input port to the accumulator (IN) and from the accumulator to an output port (OUT). The instruction OUT cannot send data from any other register.
- The data copy instructions do not affect the flags.

See Questions and Assignments 1–7 at the end of this chapter.

6.2

ARITHMETIC OPERATIONS

The 8085 microprocessor performs various arithmetic operations, such as addition, subtraction, increment, and decrement. These arithmetic operations have the following mnemonics.

*Change the high-order memory address 20 to the appropriate address for your system.

[†]Substitute the appropriate port address.

ADD : Add	Add the contents of a register.*
ADI : Add Immediate	Add 8-bit data.
SUB : Subtract	Subtract the contents of a register.
SUI : Subtract Immediate	Subtract 8-bit data.
INR : Increment	Increase the contents of a register by 1.
DCR : Decrement	Decrease the contents of a register by 1.

The arithmetic operations Add and Subtract are performed in relation to the contents of the accumulator. However, the Increment or the Decrement operations can be performed in any register. The instructions for these operations are explained below.

INSTRUCTIONS

These arithmetic instructions (except INR and DCR)

1. assume implicitly that the accumulator is one of the operands.
2. modify all the flags according to the data conditions of the result.
3. place the result in the accumulator.
4. do not affect the contents of the operand register.

The instructions INR and DCR

1. affect the contents of the specified register.
2. affect all flags except the CY flag.

The descriptions of the instructions (including INR and DCR) are as follows:

Opcode	Operand	Description
ADD	R [†]	Add <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Adds the contents of register R to the contents of the accumulator
ADI	8-bit	Add Immediate <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Adds the second byte to the contents of the accumulator
SUB	R [†]	Subtract <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Subtracts the contents of register R from the contents of the accumulator
SUI	8-bit	Subtract Immediate <input type="checkbox"/> This is a 2-byte instruction

*Memory-related arithmetic operations are excluded here; they are discussed in Chapter 7.

[†]R represents any of registers A, B, C, D, E, H, and L.

		<input type="checkbox"/> Subtracts the second byte from the contents of the accumulator
INR	R*	Increment <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Increases the contents of register R by 1 <i>Caution:</i> All flags except the CY are affected
DCR	R*	Decrement <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Decreases the contents of register R by 1 <i>Caution:</i> All flags except the CY are affected

6.2.1 Addition

The 8085 performs addition with 8-bit binary numbers and stores the sum in the accumulator. If the sum is larger than eight bits (FFH), it sets the Carry flag. Addition can be performed either by adding the contents of a source register (B, C, D, E, H, L, or memory) to the contents of the accumulator (ADD) or by adding the second byte directly to the contents of the accumulator (ADI).

Example
6.3

The contents of the accumulator are 93H and the contents of register C are B7H. Add both contents.

Instruction ADD C

$$\begin{array}{r}
 & \text{CY} & D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 \\
 (A) : & 93H = & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
 + & & & & & & & & & \\
 (C) : & B7H = & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
 & & \curvearrowleft 1 & \text{Carry} \\
 \text{SUM (A)} : & \boxed{1} 4AH = \boxed{2} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 & & \text{CY} & & & & & &
 \end{array}$$

Flag Status:[†] S = 0, Z = 0, CY = 1

When the 8085 adds 93H and B7H, the sum is 14AH; it is larger than eight bits. Therefore, the accumulator will have 4AH in binary, and the CY flag will be set. The result in the accumulator (4AH) is not 0, and bit D₇ is not 1; therefore, the Zero and the Sign flags will be reset.

*R represents any of registers A, B, C, D, E, H, and L.

[†]The P and AC flags are not shown here. In this chapter, the focus will be on the Sign, Zero, and Carry flags.

Add the number 35H directly to the sum in the previous example when the CY flag is set.

Example 6.4

Instruction ADI 35H

$$\begin{array}{r}
 & \text{CY} \\
 (\text{A}) : 4\text{AH} = \boxed{1} \quad 0 \ 1 \ 0 \ 0 \quad 1 \ 0 \ 1 \ 0 \\
 & + \\
 (\text{Data}) : 35\text{H} = \quad 0 \ 0 \ 1 \ 1 \quad 0 \ 1 \ 0 \ 1 \\
 (\text{A}) : 7\text{FH} = \boxed{0} \quad 0 \ 1 \ 1 \ 1 \quad 1 \ 1 \ 1 \ 1
 \end{array}$$

Flag Status: S = 0, Z = 0, CY = 0

The addition of 4AH and 35H does not generate a carry and will reset the previous Carry flag. Therefore, in adding numbers, it is necessary to count how many times the CY flag is set by using some other programming techniques (see Section 7.3.2).

Assume the accumulator holds the data byte FFH. Illustrate the differences in the flags set by adding 01H and by incrementing the accumulator contents.

Example 6.5

Instruction ADI 01H

$$\begin{array}{r}
 \text{(A)} : \quad \text{FFH} = \quad \begin{array}{ccccccccc} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \end{array} \\
 + \\
 \text{(Data)} : \quad 01H = \quad \begin{array}{ccccccccc} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \end{array} \\
 \hline
 \text{(A)} : \boxed{1} 00H = \quad \boxed{1} \quad \begin{array}{ccccccccc} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \end{array} \\
 \text{CY}
 \end{array}$$

Flag Status: S = 0, Z = 1, CY = 1

After adding 01H to FFH, the sum in the accumulator is 0 with a carry. Therefore, the CY and Z flags are set. The Sign flag is reset because D₇ is 0.

Instruction INR A

The accumulator contents will be 00H, the same as before. However, the instruction INR will not affect the Carry flag; it will remain in its previous status.

Flag Status: S = 0, Z = 1, CY = NA

FLAG CONCEPTS AND CAUTIONS

As described in the previous chapter, the flags are flip-flops that are set or reset after the execution of arithmetic and logic operations, with some exceptions. In many ways, the flags are like signs on an interstate highway that help drivers find their destinations.

Drivers may see one or more signs at a time. They may take the exit when they find the sign they are looking for, or they may continue along the interstate and ignore the signs.

Similarly, flags are signs of data conditions. After an operation, one or more flags may be set, and they can be used to change the direction of the program sequence by using Jump instructions, which will be described later. However, the programmer should be alert for them to make a decision. If the flags are not appropriate for the tasks, the programmer can ignore them.

Caution #1 In Example 6.3, the CY flag is set, and in Example 6.4, the CY flag is reset. The critical concept here is that if the programmer ignores the flag, it can be lost after the subsequent instructions. However, the flag can be ignored when the programmer is not interested in using it.

Caution #2 In Example 6.5, two flags are set. The programmer may use one or more flags to make decisions or may ignore them if they are irrelevant.

Caution #3 The CY flag has a dual function; it is used as a carry in addition and as a borrow in subtraction.

The importance of flags cannot be emphasized enough, and a thorough understanding of them is critical in writing assembly language programs.

1. Flags are flip-flops in the ALU (arithmetic/logic unit). They are affected (set or reset) by the operations in the ALU; therefore, operations, such as copy, that take place outside the ALU do not affect the flags.
2. The status of the flags is determined by the result of an operation. In most instances, the result is in the accumulator. However, in some operations, such as Increment (INR), results can be in registers other than the accumulator.
3. There is no relationship between a result and the bit positions of the flag register. In Example 6.3, the answer of the addition is 4AH with a carry. The binary answer is as follows:

Result										Flag Register							
CY	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀		D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	1	0	0	1	0	1	0	4A	S	Z						CY

Carry Flag Set to 1 because the answer is larger than eight bits; there is a carry generated out of the last bit D₇. During the addition, bits D₀ through D₆ may generate carries, but these carries do not affect the CY flag.

Misconception #1 Bit D₀ in the result (4AH) corresponds to the bit position of the Carry flag D₀ in the flag register; therefore, the Carry flag is reset.

Misconception #2 In the addition process, bits D₀ of 93H and B7H generate a carry (or other bit additions generate carries); therefore the Carry flag is set.

Zero Flag Reset to 0 because the answer is not zero. The Zero flag is set only when all eight bits in the result are 0.

Misconception #3 Bit D₆ in the result (4AH) is 1, and it corresponds to bit D₆ (Zero flag position) in the flag register. Therefore, the Z flag is set.

Sign Flag Reset to 0 because D₇ in the result is 0. The position of the sign flag in the flag register is also D₇. But it is just a coincidence. The microprocessor designer could have chosen bit D₆ for the Sign flag and bit D₇ for the Zero flag in the flag register. The Sign flag is relevant only when we are using signed numbers.

Misconception #4 If the Sign flag is set, the result must be negative.

See Questions and Programming Assignments 9 through 19 at the end of this chapter.

6.2.2 Illustrative Program: Arithmetic Operations—Addition and Increment

PROBLEM STATEMENT

Write a program to perform the following functions, and verify the output.

1. Load the number 8BH in register D.
2. Load the number 6FH in register C.
3. Increment the contents of register C by one.
4. Add the contents of registers C and D and display the sum at the output PORT1.

PROGRAM

The illustrative program for arithmetic operations using addition and increment is presented as Figure 6.5 to show the register contents during some of the steps.

PROGRAM DESCRIPTION

1. The first four machine codes load 8BH in register D and 6FH in register C (see Figure 6.5). These are Data Copy instructions. Therefore, no flags will be affected; the flags will remain in their previous status. The status of the flags is shown as X to indicate no change in their status.
2. Instruction INR C adds 1 to 6FH and changes the contents of C to 70H. The result is nonzero and bit D₇ is zero; therefore, the S and Z flags are reset. However, the CY flag is not affected by the INR instruction.
3. To add (C) to (D), the contents of one of the registers must be transferred to the accumulator because the 8085 cannot add two registers directly. Review the ADD instruction. The instruction MOV A,C copies 70H from register C into the accumulator without affecting (C). See the register contents.

Memory Address (H)	Machine Code	Instruction Opcode	Instruction Operand	Comments and Register Contents
HI-LO XX00	16	MVI	D,8BH	The first four machine codes load the registers as
01	8B			A S Z X X CY X B 6F D 8B H L
02	0E	MVI	C,6FH	
03	6F			
04	0C	INR	C	Add 01 to (C): 6F + 01 = 70H
05	79	MOV	A,C	A 70 S Z 0 0 CY X B 70 D 8B
06	82	ADD	D	
07	D3	OUT	PORT1	
08	PORT #	PORT1		A FB S Z 1 0 CY 0 B 70 D 8B
09	76	HLT		End of the program

FIGURE 6.5

Illustrative Program for Arithmetic Operations—Using Addition and Increment

4. Instruction ADD D adds (D) to (A), stores the sum in A, and sets the Sign flag as shown below:

$$\begin{array}{r}
 (A) : 70H = 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
 + \\
 (D) : 8BH = 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\
 \hline
 (A) : FBH = \boxed{0}\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \text{ (see Figure 6.5)} \\
 \text{CY}
 \end{array}$$

Flag Status: S = 1, Z = 0, CY = 0

5. The sum FBH is displayed by the OUT instruction.

PROGRAM OUTPUT

This program will display FBH at the output port. If an output port is not available, the program can be executed by entering NOP instructions in place of the OUT instruction and the answer FBH can be verified by examining the accumulator A. (Most systems have the Examine-Register operation.) Similarly, the contents of registers C and D and the flags can be verified.

By examining the contents of the registers, the following points can be confirmed:

1. The sum is stored in the accumulator.
2. The contents of the source registers are not changed.
3. The Sign (S) flag is set.

Even though the Sign (S) flag is set, this is not a negative sum. The microprocessor sets the Sign flag whenever an operation results in $D_7 = 1$. The microprocessor cannot recognize whether FBH is a sum, a negative number, or a bit pattern. It is your responsibility to interpret and use the flags. (See "Flag Concepts and Cautions" in Section 6.2.1.) In this example, the addition is not concerned with the signed numbers. With the signed numbers, bit D_7 is reserved for a sign by the programmer (not by the microprocessor), and no number larger than $+127_{10}$ can be entered.

6.2.3 Subtraction

The 8085 performs subtraction by using the method of 2's complement. (If you are not familiar with the method of 2's complement, review Appendix A2.)

Subtraction can be performed by using either the instruction SUB to subtract the contents of a source register or the instruction SUI to subtract an 8-bit number from the contents of the accumulator. In either case, the accumulator contents are regarded as the minuend (the number from which to subtract).

The 8085 performs the following steps internally to execute the instruction SUB (or SUI).

Step 1: Converts subtrahend (the number to be subtracted) into its 1's complement.

Step 2: Adds 1 to 1's complement to obtain 2's complement of the subtrahend.

Step 3: Add 2's complement to the minuend (the contents of the accumulator).

Step 4: Complements the Carry flag.

These steps are illustrated in the following example.

Register B has 65H and the accumulator has 97H. Subtract the contents of register B from the contents of the accumulator.

Example
6.6

Instruction SUB B

Subtrahend (B): 65H = 0 1 1 0 0 1 0 1

Step 1: 1's complement of 65H = 1 0 0 1 1 0 1 0
(Substitute 0 for 1 and 1 for 0)

+

Step 2: Add 01 to obtain 0 0 0 0 0 0 0 1
2's complement of 65H = 1 0 0 1 1 0 1 1
+

To subtract: 97H – 65H,

Add 97H to 2's complement of 65H = 1 0 0 1 0 1 1 1

1 1 1 1 1 Carry

Step 3:

Step 4: Complement Carry

Result (A): 32H

CY

1

 0 0 1 1 0 0 1 0

0

 0 0 1 1 0 0 1 0

Flag Status: S = 0, Z = 0, CY = 0

If the answer is negative, it will be shown in the 2's complement of the actual magnitude. For example, if the above subtraction is performed as 65H – 97H, the answer will be the 2's complement of 32H with the Carry (Borrow) flag set.

6.2.4 Illustrative Program: Subtraction of Two Unsigned Numbers

PROBLEM STATEMENT

Write a program to do the following:

1. Load the number 30H in register B and 39H in register C.
2. Subtract 39H from 30H.
3. Display the answer at PORT1.

PROGRAM

The illustrative program for subtraction of two unsigned numbers is presented as Figure 6.6 to show the register contents during some of the steps.

PROGRAM DESCRIPTION

1. Registers B and C are loaded with 30H and 39H, respectively. The instruction MOV A,B copies 30H into the accumulator (shown as register contents). This is an essential step because the contents of a register can be subtracted only from the contents of the accumulator and not from any other register.
2. To execute the instruction SUB C the microprocessor performs the following steps internally:

Step 1:

$$\begin{array}{r} 39H = 0 0 1 1 1 0 0 1 \\ 1's \text{ complement of } 39H = 1 1 0 0 0 1 1 0 \\ + \end{array}$$

Step 2:

$$\begin{array}{r} Add \ 01 = 0 0 0 0 0 0 0 1 \\ 2's \text{ complement of } 39H = 1 1 0 0 0 1 1 1 \\ + \end{array}$$

Step 3:

$$\begin{array}{r} Add \ 30H \text{ to } 2's \text{ complement of } 39H = 0 0 1 1 0 0 0 0 \\ CY \ \boxed{0} \quad \underline{1 1 1 1} \quad 0 1 1 1 \end{array}$$

Step 4: Complement carry

[1] 1 1 1 1 0 1 1 1 = F7H

Flag Status: S = 1, Z = 0, CY = 1

3. The number F7H is a 2's complement of the magnitude $(39H - 30H) = 09H$.
 4. The instruction OUT displays F7H at PORT1.

PROGRAM OUTPUT

This program will display F7H as the output. In this program, the unsigned numbers were used to perform the subtraction. Now, the question is: How do you recognize that the answer F7H is really a 2's complement of 09H and not a straight binary F7H?

The answer lies with the Carry flag. If the Carry flag (also known as the Borrow flag in subtraction) is set, the answer is in 2's complement. The Carry flag raises a second question: Why isn't it a positive sum with a carry? The answer is implied by the instruction SUB (it is a subtraction).

There is no way to differentiate between a straight binary number and 2's complement by examining the answer at the output port. The flags are internal and not easily displayed. However, a programmer can test the Carry flag by using the instruction Jump On Carry (JC) and can find a way to indicate that the answer is in 2's complement. (This is discussed in Branch instructions.)

Memory Address (H)	Machine Code	Instruction		Comments and Register Contents								
		Opcode	Operand									
HI-LO												
XX00	06	MVI	B,30H									
01	30			Load the minuend in register B								
02	0E	MVI	C,39H	Load the subtrahend in register C								
03	39			The register contents:								
04	78	MOV	A,B									
				<table border="1"> <tr> <td>A</td><td>30</td><td></td></tr> <tr> <td>B</td><td>30</td><td>39</td></tr> </table>	A	30		B	30	39		
A	30											
B	30	39										
05	91	SUB	C	 <table border="1"> <tr> <td>A</td><td>F7</td><td>S Z 1 0</td><td>CY 1</td></tr> <tr> <td>B</td><td>30</td><td></td><td>39</td></tr> </table>	A	F7	S Z 1 0	CY 1	B	30		39
A	F7	S Z 1 0	CY 1									
B	30		39									
06	D3	OUT	PORT1									
07	PORT#											
08	76	HLT										

FIGURE 6.6
Illustrative Program for Subtraction of Two Unsigned Numbers

6.2.5 Review of Important Concepts

1. The arithmetic operations implicitly assume that the contents of the accumulator are one of the operands.

2. The results of the arithmetic operations are stored in the accumulator; thus, the previous contents of the accumulator are altered.
3. The flags are modified to reflect the data conditions of an operation.
4. The contents of the source register are not changed as a result of an arithmetic operation.
5. In the Add operation, if the sum is larger than 8-bit, CY is set.
6. The Subtract operation is performed by using the 2's complement method.
7. If a subtraction results in a negative number, the answer is in 2's complement and CY (the Borrow flag) is set.
8. In unsigned arithmetic operations, the Sign flag (S) should be ignored.
9. The instructions INR (Increment) and DCR (Decrement) are special cases of the arithmetic operations. These instructions can be used for any one of the registers, and they do not affect CY, even if the result is larger than 8-bit. All other flags are affected by the result in the register used (not by the contents of the accumulator).

6.3

LOGIC OPERATIONS

A microprocessor is basically a programmable logic chip. It can perform all the logic functions of the hard-wired logic through its instruction set. The 8085 instruction set includes such logic functions as AND, OR, Ex OR, and NOT (complement). The opcodes of these operations are as follows:*

ANA:	AND	Logically AND the contents of a register.
ANI :	AND Immediate	Logically AND 8-bit data.
ORA:	OR	Logically OR the contents of a register.
ORI :	OR Immediate	Logically OR 8-bit data.
XRA:	X-OR	Exclusive-OR the contents of a register.
XRI :	X-OR Immediate	Exclusive-OR 8-bit data.

All logic operations are performed in relation to the contents of the accumulator. The instructions of these logic operations are described below.

INSTRUCTIONS

The logic instructions

1. implicitly assume that the accumulator is one of the operands.
2. reset (clear) the CY flag. The instruction CMA is an exception; it does not affect any flags.
3. modify the Z, P, and S flags according to the data conditions of the result.
4. place the result in the accumulator.
5. do not affect the contents of the operand register.

*Memory-related logic operations are excluded here; they will be discussed in the next chapter.

Opcode	Operand	Description
ANA	R	Logical AND with Accumulator <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Logically ANDs the contents of the register R with the contents of the accumulator <input type="checkbox"/> 8085: CY is reset and AC is set
ANI	8-bit	AND Immediate with Accumulator <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Logically ANDs the second byte with the contents of the accumulator <input type="checkbox"/> 8085: CY is reset and AC is set
ORA	R	Logically OR with Accumulator <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Logically ORs the contents of the register R with the contents of the accumulator
ORI	8-bit	OR Immediate with Accumulator <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Logically ORs the second byte with the contents of the accumulator
XRA	R	Logically Exclusive-OR with Accumulator <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> Exclusive-ORs the contents of register R with the contents of the accumulator
XRI	8-bit	Exclusive-OR Immediate with Accumulator <input type="checkbox"/> This is a 2-byte instruction <input type="checkbox"/> Exclusive-ORs the second byte with the contents of the accumulator
CMA		Complement Accumulator <input type="checkbox"/> This is a 1-byte instruction that complements the contents of the accumulator <input type="checkbox"/> No flags are affected

6.3.1 Logic AND

The process of performing logic operations through the software instructions is slightly different from the hardwired logic. The AND gate shown in Figure 6.7(a) has two inputs and one output. On the other hand, the instruction ANA simulates eight AND gates, as shown in Figure 6.7(b). For example, assume that register B holds 77H and the accumulator A holds 81H. The result of the instruction ANA B is 01H and is placed in the accumulator replacing the previous contents, as shown in Figure 6.7(b).

Figure 6.7(b) shows that each bit of register B is independently ANDed with each bit of the accumulator, thus simulating eight 2-input AND gates.

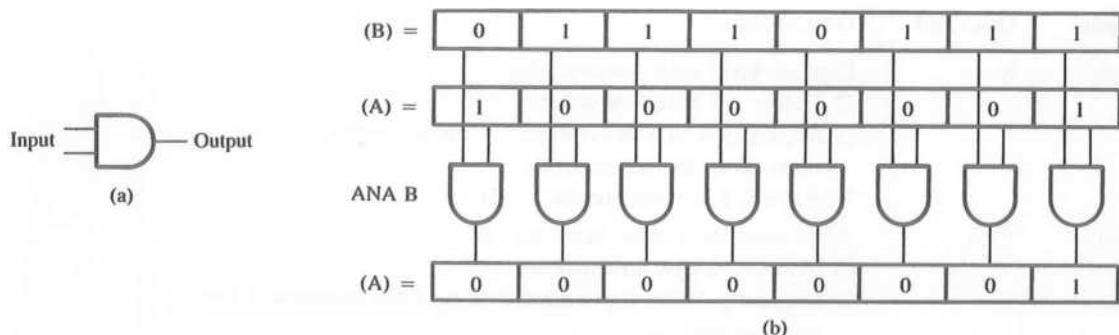


FIGURE 6.7
AND Gate (a) and a Simulated ANA Instruction (b)

6.3.2 Illustrative Program: Data Masking with Logic AND

PROBLEM STATEMENT

To conserve energy and to avoid an electrical overload on a hot afternoon, implement the following procedures to control the appliances throughout the house (Figure 6.8). Assume that the control switches are located in the kitchen, and they are available to anyone in the house. Write a set of instructions to

1. turn on the air conditioner if switch S₇ of the input port 00H is on.
2. ignore all other switches of the input port even if someone attempts to turn on other appliances.

(To perform this experiment on your single-board microcomputer, simulate the reading of the input port 00H with the instruction MVI A, 8-bit data.)

PROBLEM ANALYSIS

In this problem you are interested in only one switch position, S₇, which is connected to data line D₇. Assume that various persons in the family have turned on the switches of the air conditioner (S₇), the radio (S₄), and the lights (S₃, S₂, S₁, S₀).

If the microprocessor reads the input port (IN 00H), the accumulator will have data byte 9FH. This can be simulated by using the instruction MVI A, 9FH. However, if you are interested in knowing only whether switch S₇ is on, you can mask bits D₆ through D₀ by ANDing the input data with a byte that has 0 in bit positions D₆ through D₀ and 1 in bit position D₇.

$$\begin{array}{cccccccc} D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} = 80H$$

After bits D₆ through D₀ have been masked, the remaining byte can be sent to the output port to simulate turning on the air conditioner.

PROGRAM

Memory Address	Machine Code	Instruction Opcode	Operand	Comments
HI-LO				
XX00	3E	MVI	A,Data	;This instruction simulates the
01	9F			; instruction IN 00H
02	E6	ANI	80H	;Mask all the bits except D ₇
03	80			
04	D3	OUT	01H	;Turn on the air conditioner if
05	01			; S ₇ is on
06	76	HLT		;End of the program

PROGRAM OUTPUT

The instruction ANI 80H ANDs the accumulator data as follows:

$$\begin{array}{r}
 (A) = 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ (9FH) \\
 \text{AND} \\
 (\text{Masking Byte} = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (80H)) \\
 \hline
 (A) = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (80H)
 \end{array}$$

Flag Status: S = 1, Z = 0, CY = 0

The ANDing operation always resets the CY flag. The result (80H) will be placed in the accumulator and then sent to the output port, and logic 1 of data bit D₇ turns on the air conditioner. In this example, the output (80H) is the same as the masking data byte (80H) because switch S₇ (or data bit D₇) is on. If S₇ is off, the output will be zero.

The masking is a commonly used technique to eliminate unwanted bits in a byte. The masking byte to be logically ANDed is determined by placing 0s in bit positions that are to be masked and by placing 1s in the remaining bit positions.

6.3.3 OR, Exclusive-OR, and NOT

The instruction ORA (and ORI) simulates logic ORing with eight 2-input OR gates; this process is similar to that of ANDing, explained in the previous section. The instruction XRA (and XRI) performs Exclusive-ORing of eight bits, and the instruction CMA inverts the bits of the accumulator.

Assume register B holds 93H and the accumulator holds 15H. Illustrate the results of the instructions ORA B, XRA B, and CMA.

Example
6.7

1. The instruction ORA B will perform the following operation:

$$\text{OR} \quad \begin{array}{l} (\text{B}) = 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ (\text{93H}) \\ (\text{A}) = 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ (\text{15H}) \end{array}$$

$$\underline{\hspace{10em}} \\ (\text{A}) = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ (\text{97H})$$

Flag Status: S = 1, Z = 0, CY = 0

The result 97H will be placed in the accumulator, the CY flag will be reset, and the other flags will be modified to reflect the data conditions in the accumulator.

2. The instruction XRA B will perform the following operation.

$$\text{X-OR} \quad \begin{array}{l} (\text{B}) = 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ (\text{93H}) \\ (\text{A}) = 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ (\text{15H}) \end{array}$$

$$\underline{\hspace{10em}} \\ (\text{A}) = 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ (\text{86H})$$

Flag Status: S = 1, Z = 0, CY = 0

The result 86H will be placed in the accumulator, and the flags will be modified as shown.

3. The instruction CMA will result in

$$\text{CMA} \quad \begin{array}{l} (\text{A}) = 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ (\text{15H}) \end{array}$$

$$\underline{\hspace{10em}} \\ (\text{A}) = 1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ (\text{EAH})$$

The result EAH will be placed in the accumulator and no flags will be modified.

6.3.4 Setting and Resetting Specific Bits

At various times, we may want to set or reset a specific bit without affecting the other bits. OR logic can be used to set the bit, and AND logic can be used to reset the bit.

**Example
6.8**

In Figure 6.8, keep the radio on (D_4) continuously without affecting the functions of other appliances, even if someone turns off the switch S_4 .

Solution

To keep the radio on without affecting the other appliances, the bit D_4 should be set by ORing the reading of the input port with the data byte 10H as follows:

$$\begin{array}{l} \text{IN } 00\text{H: } (\text{A}) = D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0 \\ \text{ORI } 10\text{H: } \quad \quad \quad = 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\ \underline{\hspace{10em}} \\ (\text{A}) = D_7\ D_6\ D_5\ 1\ D_3\ D_2\ D_1\ D_0 \end{array}$$

Flag Status: CY = 0; others will depend on data.

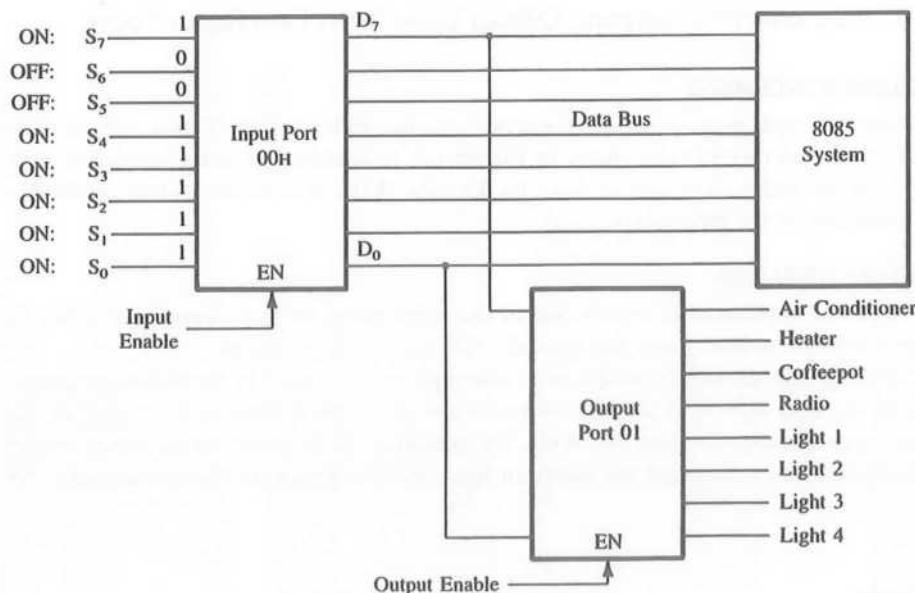


FIGURE 6.8
Input Port to Control Appliances

The instruction IN reads the switch positions shown as D₇–D₀ and the instruction ORI sets the bit D₄ without affecting any other bits.

In Figure 6.8, assume it is winter, and turn off the air conditioner without affecting the other appliances.

Example
6.9

To turn off the air conditioner, reset bit D₇ by ANDing the reading of the input port with the data byte 7FH as follows:

$$\begin{array}{r}
 \text{IN } 00\text{H}: (A) = D_7 \ D_6 \ D_5 \ D_4 \ D_3 \ D_2 \ D_1 \ D_0 \\
 \text{ANI } 7\text{FH}: \quad = 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\
 \hline
 0 \ D_6 \ D_5 \ D_4 \ D_3 \ D_2 \ D_1 \ D_0
 \end{array}$$

Solution

Flag Status: CY = 0; others will depend on the data bits.

The ANI instruction resets bit D₇ without affecting the other bits.

6.3.5 Illustrative Program: ORing Data from Two Input Ports

PROBLEM STATEMENT

An additional input port with eight switches and the address 01H (Figure 6.9) is connected to the microcomputer shown in Figure 6.8 to control the same appliances and lights from the bedroom as well as from the kitchen. Write instructions to turn on the devices from any of the input ports.

PROBLEM ANALYSIS

To turn on the appliances from any one of the input ports, the microprocessor needs to read the switches at both ports and logically OR the switch positions.

Assume that the switch positions in one input port (located in the bedroom) correspond to the data byte 91H and the switch positions in the second port (located in the kitchen) correspond to the data byte A8H. The person in the bedroom wants to turn on the air conditioner, the radio, and the bedroom light; and the person in the kitchen wants to

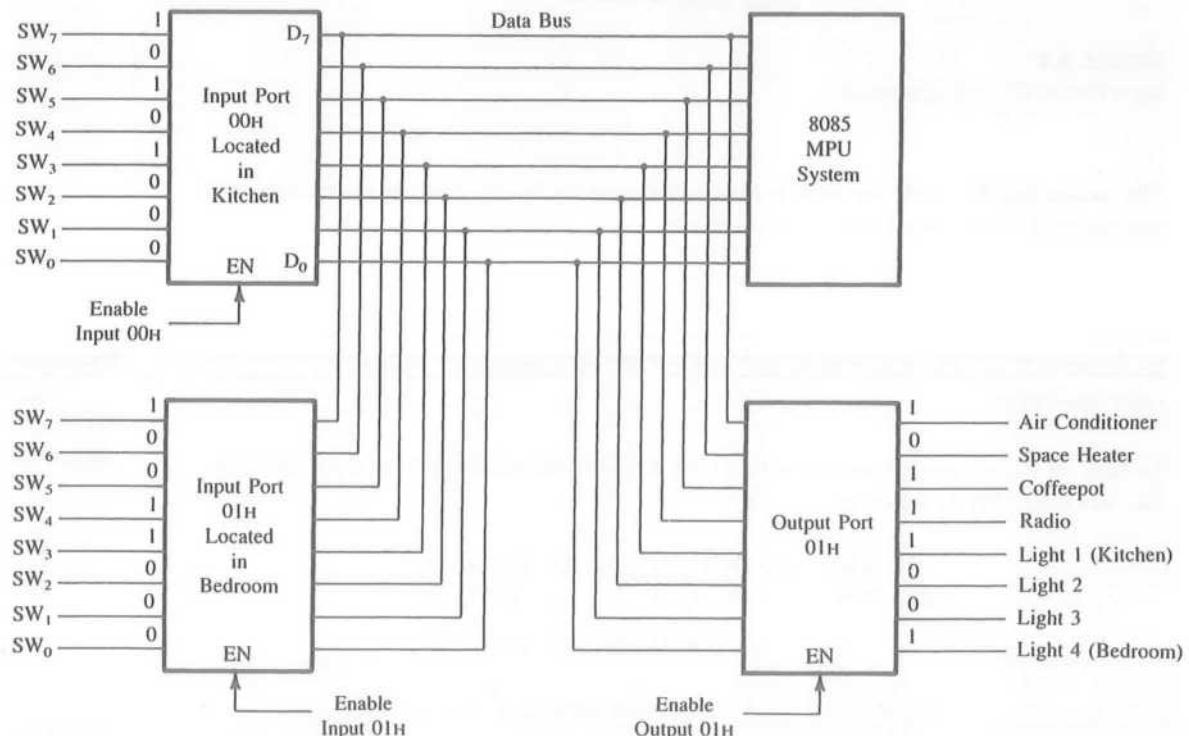


FIGURE 6.9

Two Input Ports to Control Output Devices

turn on the air conditioner, the coffeepot, and the kitchen light. By ORing these two data bytes the microprocessor can turn on the necessary appliances.

To test this program, we must simulate the readings of the input port by loading the data into registers—for example, into B and C.

PROGRAM

Memory Address	Machine Code	Instructions		Comments
		Opcode	Operand	
HI-LO				
XX00	06	MVI	B,91H	;This instruction simulates reading input port 01H
01	91			
02	0E	MVI	C,A8H	;This instruction simulates reading input port 00H
03	A8			
04	78	MOV	A,B	;It is necessary to transfer data byte from B to A to OR with C. B and C cannot be ORed directly
05	B1	ORA	C	;Combine the switch positions from registers B and C in the accumulator
06	D3	OUT	PORT1	;Turn on appliances and lights
07	PORT1			
08	76	HLT		;End of the program

PROGRAM OUTPUT

By logically ORing the data bytes in registers B and C

$$\begin{array}{r}
 (B) \rightarrow (A) = 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ (91H) \\
 (C) = 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ (A8H) \\
 \hline
 (A) = 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ (B9H)
 \end{array}$$

Flag Status: S = 1, Z = 0, CY = 0

Data byte B9H is placed in the accumulator that turns on the air conditioner, radio, coffeepot, and bedroom and kitchen lights.

6.3.6 Review of Important Concepts

1. Logic operations are performed in relation to the contents of the accumulator.
2. Logic operations simulate eight 2-input gates (or inverters).
3. The Sign, Zero (and Parity) flags are modified to reflect the status of the operation. The Carry flag is reset. However, the NOT operation does not affect any flags.
4. After a logic operation has been performed, the answer is placed in the accumulator replacing the original contents of the accumulator.

5. The logic operations cannot be performed directly with the contents of two registers.
6. The individual bits in the accumulator can be set or reset using logic instructions.

See Questions and Programming Assignments 20–29 at the end of this chapter.

6.4

BRANCH OPERATIONS

The **branch instructions** are the most powerful instructions because they allow the microprocessor to change the sequence of a program, either unconditionally or under certain test conditions. These instructions are the key to the flexibility and versatility of a computer.

The microprocessor is a sequential machine; it executes machine codes from one memory location to the next. Branch instructions instruct the microprocessor to go to a different memory location, and the microprocessor continues executing machine codes from that new location. The address of the new memory location is either specified explicitly or supplied by the microprocessor or by extra hardware. The branch instructions are classified in three categories:

1. Jump instructions
2. Call and Return instructions
3. Restart instructions

This section is concerned with applications of Jump instructions. The Call and Return instructions are associated with the subroutine technique and will be discussed in Chapter 9; Restart instructions are associated with the interrupt technique and will be discussed in Chapter 12.

The Jump instructions specify the memory location explicitly. They are 3-byte instructions: one byte for the operation code, followed by a 16-bit memory address. Jump instructions are classified into two categories: Unconditional Jump and Conditional Jump.

6.4.1 Unconditional Jump

The 8085 instruction set includes one unconditional Jump instruction. The unconditional Jump instruction enables the programmer to set up continuous loops.

INSTRUCTION

Opcode	Operand	Description
JMP	16-bit	<p>Jump</p> <ul style="list-style-type: none"><input type="checkbox"/> This is a 3-byte instruction<input type="checkbox"/> The second and third bytes specify the 16-bit memory address. However, the second byte specifies the low-order and the third byte specifies the high-order memory address

For example, to instruct the microprocessor to go to the memory location 2000H, the mnemonics and the machine code entered will be as follows:

Machine Code	Mnemonics
C3	JMP 2000H
00	
20	

Note the sequence of the machine code. The 16-bit memory address of the jump location is entered in the reverse order, the low-order byte (00H) first, followed by the high-order byte (20H). The 8085 is designed for such a reverse sequence. The jump location can also be specified using a label. While writing a program, you may not know the exact memory location to which a program sequence should be directed. In that case, the memory address can be specified with a label (or a name). This is particularly useful and necessary for an assembler. However, you should not specify both a label and its 16-bit address in a Jump instruction. Furthermore, you cannot use the same label for different memory locations. The next illustrative program shows the use of the Jump instruction.

6.4.2 Illustrative Program: Unconditional Jump to Set Up a Continuous Loop

PROBLEM STATEMENT

Modify the program in Example 6.2 to read the switch positions continuously and turn on the appliances accordingly.

PROBLEM ANALYSIS

One of the major drawbacks of the program in Example 6.2 is that the program reads switch positions once and then stops. Therefore, if you want to turn on/off different appliances, you have to reset the system and start all over again. This is impractical in real-life situations. However, the unconditional Jump instruction, in place of the HLT instruction, will allow the microcomputer to monitor the switch positions continuously.

PROGRAM

Memory Address	Machine Code	Label	Mnemonics	Comments
2000	DB	START:	IN 00H	;Read input switches
2001	00			
2002	D3		OUT 01H	;Turn on devices according to
2003	01			; switch positions
2004	C3		JMP START	;Go back to beginning and
2005	00			; read the switches again
2006	20			

PROGRAM FORMAT

The program includes one more column called *label*. The memory location 2000H is defined with the label START; therefore, the operand of the Jump instruction can be specified by the label START. The program sets up the endless loop, and the microprocessor monitors the input port continuously. The output will reflect any change in the switch positions.

6.4.3 Conditional Jumps

Conditional Jump instructions allow the microprocessor to make decisions based on certain conditions indicated by the flags. After logic and arithmetic operations, flip-flops (flags) are set or reset to reflect data conditions. The conditional Jump instructions check the flag conditions and make decisions to change or not to change the sequence of a program.

FLAGS

The 8085 flag register has five flags, one of which (Auxiliary Carry) is used internally. The other four flags used by the Jump instructions are

1. Carry flag
2. Zero flag
3. Sign flag
4. Parity flag

Two Jump instructions are associated with each flag. The sequence of a program can be changed either because the condition is present or because the condition is absent. For example, while adding the numbers you can change the program sequence either because the carry is present (JC = Jump On Carry) or because the carry is absent (JNC = Jump On No Carry).

INSTRUCTIONS

All conditional Jump instructions in the 8085 are 3-byte instructions; the second byte specifies the low-order (line number) memory address, and the third byte specifies the high-order (page number) memory address. The following instructions transfer the program sequence to the memory location specified under the given conditions:

Opcode	Operand	Description
JC	16-bit	Jump On Carry (if result generates carry and CY = 1)
JNC	16-bit	Jump On No Carry (CY = 0)
JZ	16-bit	Jump On Zero (if result is zero and Z = 1)
JNZ	16-bit	Jump On No Zero (Z = 0)
JP	16-bit	Jump On Plus (if D ₇ = 0, and S = 0)
JM	16-bit	Jump On Minus (if D ₇ = 1, and S = 1)
JPE	16-bit	Jump On Even Parity (P = 1)
JPO	16-bit	Jump On Odd Parity (P = 0)

All the Jump instructions are listed here for an overview. The Zero and Carry flags and related Jump instructions are used frequently. They are illustrated in the following examples.

6.4.4 Illustrative Program: Testing of the Carry Flag

PROBLEM STATEMENT

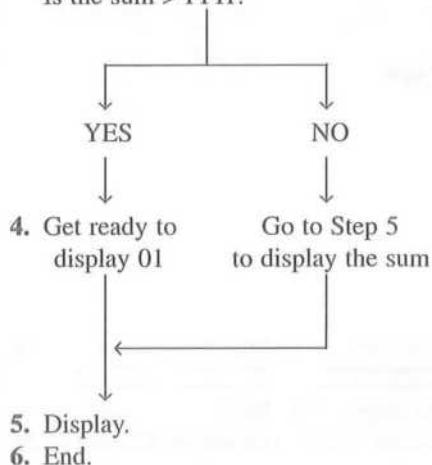
Load the hexadecimal numbers 9BH and A7H in registers D and E, respectively, and add the numbers. If the sum is greater than FFH, display 01H at output PORT0; otherwise, display the sum.

PROBLEM ANALYSIS AND FLOWCHART

The problem can be divided into the following steps:

1. Load the numbers in the registers.
2. Add the numbers.
3. Check the sum.

Is the sum > FFH?



FLOWCHART AND ASSEMBLY LANGUAGE PROGRAM

The six steps listed above can be converted into a flowchart and assembly language program as shown in Figure 6.10.

Step 3 is a decision-making block. In a flowchart, the decision-making process is represented by a diamond shape. It is important to understand how this block is translated into the assembly language program. By examining the block carefully you will notice the following:

1. The question is: Is there a Carry?
2. If the answer is no, change the sequence of the program. In the assembly language this is equivalent to Jump On No Carry—JNC.

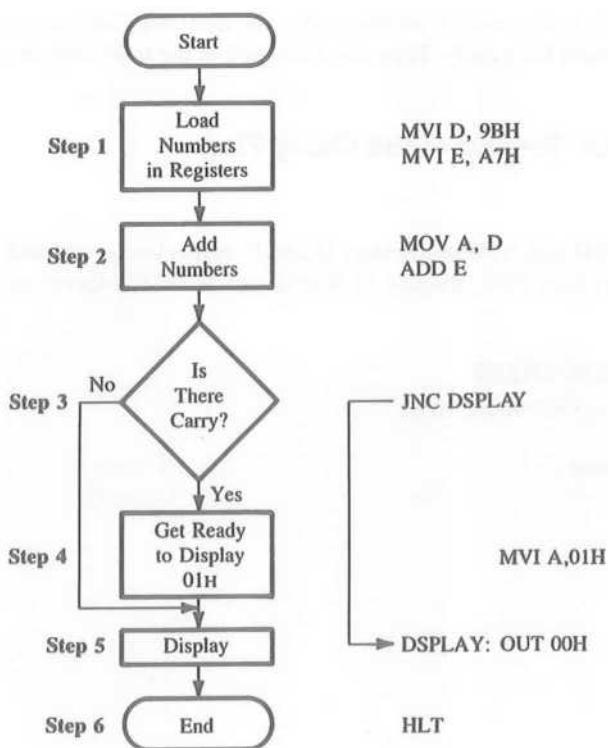


FIGURE 6.10
Flowchart and Assembly Language Program to Test Carry Flag

3. Now the next question is where to change the sequence—to Step 5. At this point the exact location is not known, but it is labeled DSPLAY.
4. The next step in the sequence is 4. Get ready to display byte 01H.
5. After completing the straight line sequence, translate Step 5 and Step 6: Display at the port and halt.

MACHINE CODE WITH MEMORY ADDRESSES

Assuming your R/W memory begins at 2000H, the preceding assembly language program can be translated as follows:

Memory Address	Machine Code	Label	Mnemonics
2000	16	START:	MVI D,9BH
2001	9B		
2002	1E		MVI E,A7H
2003	A7		

2004	7A	MOV A,D
2005	83	ADD E
2006	D2	JNC DISPLAY
2007	X	
2008	X	
2009	3E	MVI A,01H
200A	01	
200B	D3	DISPLAY: OUT 00H
200C	00	
200D	76	HLT

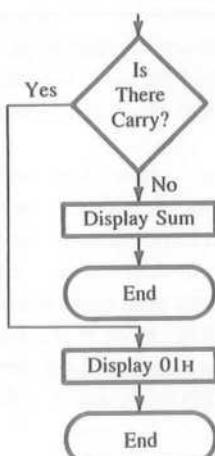
While translating into the machine code, we leave memory locations 2007H and 2008H blank because the exact location of the transfer is not known. What is known is that two bytes should be reserved for the 16-bit address. After completing the straight line sequence, we know the memory address of the label DISPLAY; i.e., 200BH. This address must be placed in the reversed order as shown:

2007	0B	Low-order: Line Number
2008	20	High-order: Page Number

USING THE INSTRUCTION JUMP ON CARRY (JC)

Now the question remains: Can the same problem be solved by using the instruction Jump On Carry (JC)? To use instruction JC, exchange the places of the answers YES and NO to the question: Is there a Carry? The flowchart will be as in Figure 6.11, and it shows that the program sequence is changed if there is a Carry. This flowchart has two end points; thus it will require a few more instructions than that of Figure 6.10. In this particular example, it is unimportant whether to use instruction JC or JNC, but in most cases the choice is made by the logic of a problem.

FIGURE 6.11
Flowchart for the Instruction Jump
On Carry



6.4.5 Review of Important Concepts

1. The Jump instructions change program execution from its sequential order to a different memory location.
2. The Jump instructions can transfer program execution ahead of the sequence (Jump Forward) or behind the sequence (Jump Backward).
3. The unconditional Jump is, generally, used to set up continuous loops.
4. The conditional Jumps are used for the decision-making process based on the data conditions of the result, reflected by the flags.
5. Arithmetic and logic instructions modify the flags according to the data of the result, and the conditional branch instructions use them to make decisions. However, the branch instructions do not affect the flags.

CAUTION

The conditional Jump instructions will not function properly unless the preceding instruction sets the necessary flag. Data Copy instructions do not affect the flags; furthermore, some arithmetic and logic instructions either do not affect the flags or affect only certain flags.

See Questions and Programming Assignments 30–40 at the end of this chapter.

6.5

WRITING ASSEMBLY LANGUAGE PROGRAMS

Communicating with a microcomputer—giving it commands to perform a task and watching it perform them—is exciting. However, one can be uneasy communicating in strange mnemonics and hexadecimal machine codes. This feeling is like the uneasiness one has when beginning to speak a foreign language. How do we learn to communicate with a microcomputer in its assembly language? By using a few mnemonics at a time such as the mnemonics for Read the switches and Display the data. This chapter has introduced a group of basic instructions that can command the 8085 microprocessor to perform simple tasks.

After we know a few instructions, how do we begin to write a program? Any program, no matter how large, begins with mnemonics. And, just as several persons contribute to the construction of a hundred-story building, so the writing of a large program is usually the work of a team. In addition, the 8085 instruction set contains only 74 different instructions, some of them used quite frequently.

In a hundred-story building, most of the rooms are similar. If one knows the basic fundamentals of constructing a room, one can learn how to tie these rooms together in a coherent structure. However, planning and forethought are critical. Before beginning to build a structure, an architectural plan must be drawn. Similarly, to write a program, one

needs to draw up a plan of logical thoughts. A given task should be broken down into small units that can be built independently. This is called the **modular design approach**.

6.5.1 Getting Started

Writing a program is equivalent to giving specific *commands* to the microprocessor in a *sequence* to *perform a task*. The italicized words provide clues to writing a program. Let us examine these terms.

- Perform a Task.* What is the task you are asking it to do?
- Sequence.* What is the sequence you want it to follow?
- Commands.* What are the commands (instruction set) it can understand?

These terms can be translated into steps as follows:

- Step 1:** Read the problem carefully.
- Step 2:** Break it down into small steps.
- Step 3:** Represent these small steps in a possible sequence with a flowchart—a plan of attack.
- Step 4:** Translate each block of the flowchart into appropriate mnemonic instructions.
- Step 5:** Translate mnemonics into the machine code.
- Step 6:** Enter the machine code in memory and execute. Only on rare occasions is a program successfully executed on the first attempt.
- Step 7:** Start troubleshooting (see Section 6.6, “Debugging a Program”).

These steps are illustrated in the next section.

6.5.2 Illustrative Program: Microprocessor-Controlled Manufacturing Process

PROBLEM STATEMENT

A microcomputer is designed to monitor various processes (conveyer belts) on the floor of a manufacturing plant, presented schematically in Figure 6.12. The microcomputer has two input ports with the addresses F1H and F2H and an output port with the address F3H. Input port F1H has six switches, five of which (corresponding to data lines D₄–D₀) control the conveyer belts through the output port F3H. Switch S₇, corresponding to the data line D₇, is reserved to indicate an emergency on the floor. As a precautionary measure, input port F2H is controlled by the foreman, and its switch, S₇', is also used to indicate an emergency. Output line D₆ of port F3H is connected to the emergency alarm.

Write a program to

1. turn on the five conveyer belts according to the ON/OFF positions of switches S₄–S₀ at port F1H.
2. turn off the conveyer belts and turn on the emergency alarm only when both switches—S₇ from port F1H and S₇' from port F2H—are triggered.
3. monitor the switches continuously.

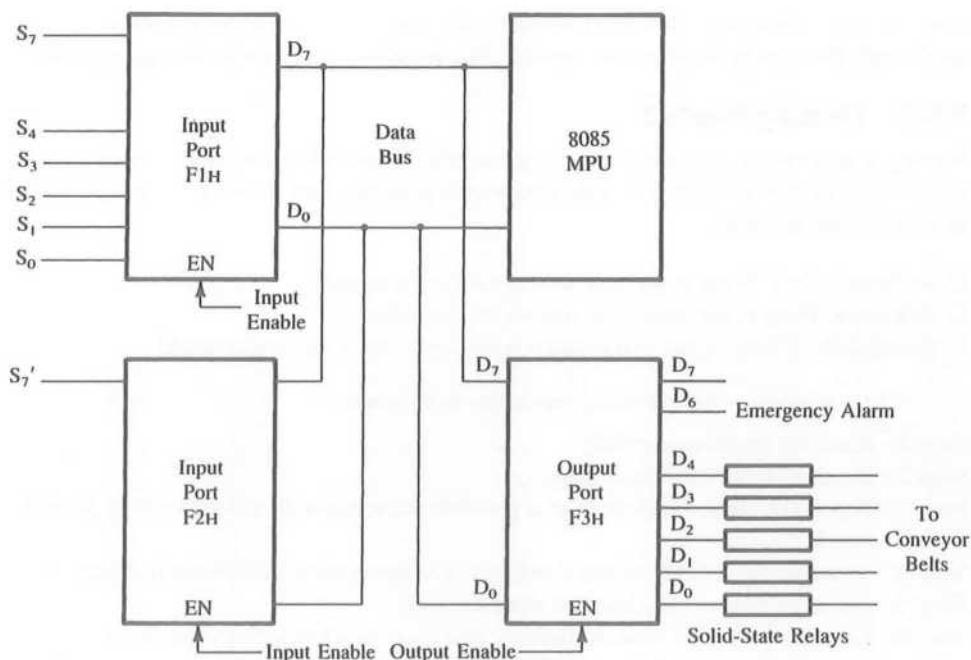


FIGURE 6.12
Input/Output Ports to Control Manufacturing Processes

PROBLEM ANALYSIS

To perform the tasks specified in the problem, the microprocessor needs to

1. read the switch positions.
2. check whether switches S₇ and S_{7'} from the ports F1H and F2H are on.
3. turn on the emergency signal if both switches are on, and turn off all the conveyer belts.
4. turn on the conveyer belts according to the switch positions S₀ through S₄ at input port F1H if both switches, S₇ and S_{7'}, are not on simultaneously.
5. continue checking the switch positions.

FLOWCHART AND PROGRAM

The five steps listed above can be translated into a flowchart and an assembly language program as shown in Figure 6.13.

6.5.3 Documentation

A program is similar to a circuit diagram. Its purpose is to communicate to others what the program does and how it does it. Appropriate comments are critical for conveying the logic behind a program. The program as a whole should be self-documented.

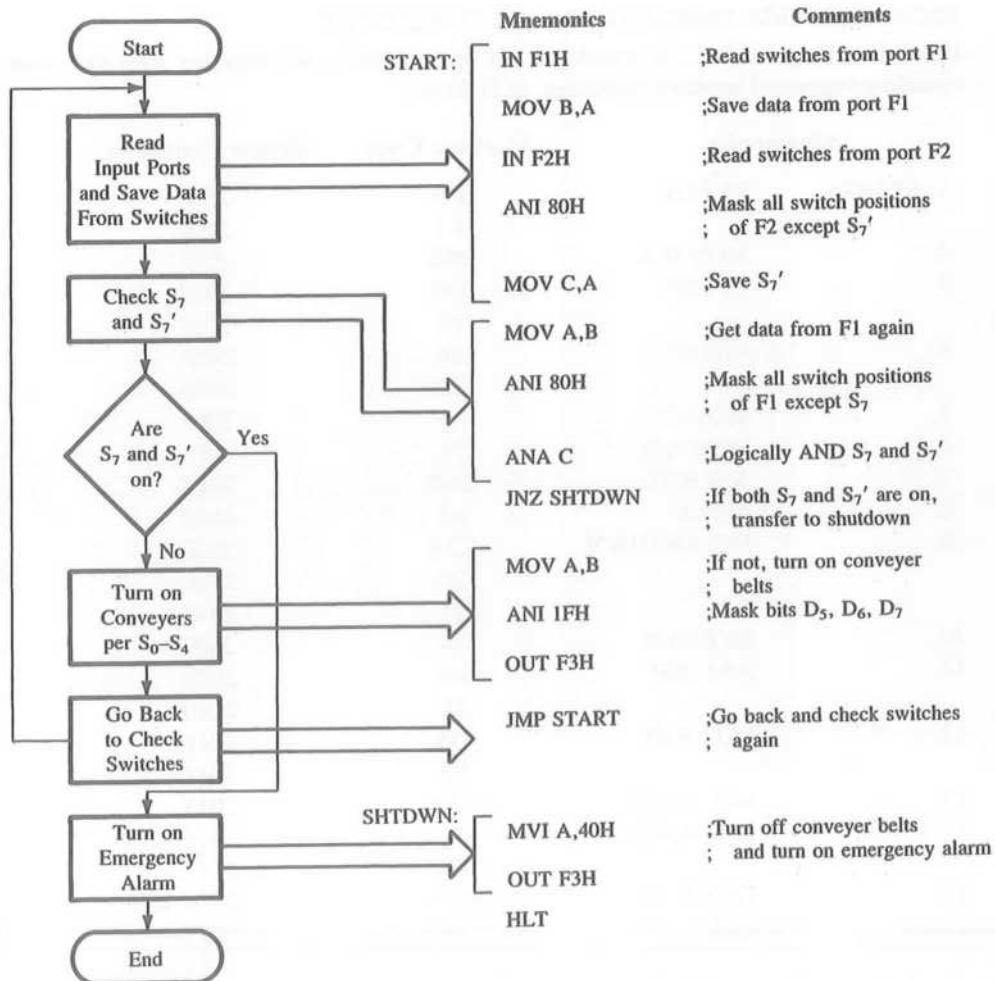


FIGURE 6.13

Flowchart and Program for Controlling Manufacturing Processes

The comments should explain what is intended; they should not explain the mnemonics. For example, the first comment in Figure 6.13 indicates that the switch positions are being read at the input port and that the reading is saved. There is no point in writing: MOV B,A means transfer the contents of the accumulator to register B. Similarly, in a schematic, one does not write the word *resistor* once a resistor is represented by a symbol. A comment can be omitted if it does not say anything more than repeat a mnemonic.

The labels START and SHTDWN indicate what actions are being taken. These are the landmarks in a program. Avoid “cute” labels. Using cute labels in a program is similar to representing a power supply in a schematic by a picture of the sun or a ground with the digit zero.

FROM ASSEMBLY LANGUAGE TO MACHINE CODE

Illustrative Program 6.5.2 is translated into its machine code, together with the corresponding suggested memory addresses, as follows:

	Mnemonics	Machine Code	Memory Addresses
1.	START: IN F1H	DB F1	2000 2001
2.	MOV B,A	78①	2002
3.	IN F2H	DB F2	2003 2004
4.	ANI 80H	E6 80	2005 2006
5.	MOV C,A	4F	2007
6.	MOV A,B	78	2008
7.	ANI 80H	E6②	2009
8.	ANA C	A1	200A
9.	JNZ SHTDWN	C2③	200B
		20	200C
		14	200D
10.	MOV A,B	78	200E
11.	ANI 1FH	E6 1F	200F 2010
12.	OUT F3H	D3 F3	2011 2012
13.	JMP START	C3④	2013
14.	SHTDWN: MVI A,40H	3E 40	2014 2015
15.	OUT F3H	D3⑤	2016
16.	HLT	76	2017

This program includes several errors, indicated by the circled numbers beside the codes. (See Assignment 41 for the debugging of this program.)

PROGRAM EXECUTION

The above machine codes can be loaded in R/W memory, starting with memory address 2000H. The execution of the program can be done two ways. The first is to execute the entire code by pressing the Execute key, and the second is to use the Single-Step key on a single-board computer. The Single-Step key executes one instruction at a time, and by using the Examine Register key, you can observe the contents of the registers and the flags as each instruction is being executed. The Single-Step and Examine Register techniques are discussed in Chapter 7 under the topic "Dynamic Debugging."

6.6

DEBUGGING A PROGRAM

Debugging a program is similar to troubleshooting hardware, but it is much more difficult and cumbersome. It is easy to poke and pinch at the components in a circuit, but, in a program, the result is generally binary: either it works or it does not work. When it does not work, very few clues alert you to what exactly went wrong. Therefore, it is essential to search carefully for the errors in the program logic, machine codes, and execution.

The debugging process can be divided into two parts: static debugging and dynamic debugging.

Static debugging is similar to visual inspection of a circuit board; it is done by a paper-and-pencil check of a flowchart and machine code. **Dynamic debugging** involves observing the output, or register contents, following the execution of each instruction (the single-step technique) or of a group of instructions (the breakpoint technique). Dynamic debugging will be discussed in the next chapter.

6.6.1 Debugging Machine Code

Translating the assembly language to the machine code is similar to building a circuit from a schematic diagram; the machine code will have errors just as would the circuit board. The following errors are common:

1. Selecting a wrong code.
2. Forgetting the second or third byte of an instruction.
3. Specifying the wrong jump location.
4. Not reversing the order of high and low bytes in a Jump instruction.
5. Writing memory addresses in decimal, thus specifying wrong jump locations.

The program for controlling manufacturing processes listed in Section 6.5.3 has several of these errors. These errors must be corrected before entering the machine code in the R/W memory of your system.

See Questions and Programming Assignments 41–43 at the end of this chapter.

SOME PUZZLING QUESTIONS AND THEIR ANSWERS

6.7

After one learns something about the microprocessor architecture, memory, I/O, the instruction set, and simple programming, a few questions still remain unanswered. These questions do not fit into any particular discussion. They just lurk in the corners of one's mind to reappear once in a while when one is in a contemplative mood. This section attempts to answer some of these unasked questions.

1. *What happens in a single-board microcomputer when the power is turned on and the Reset key is pushed?*

When the power is turned on, the monitor program stored either in EPROM or ROM comes alive. The Reset key clears the program counter, and the program counter holds the memory address 0000H. Some systems are automatically reset when the power is turned on (called power-on reset).

2. How does the microprocessor know how and when to start?

As soon as the Reset key is pushed, the program counter places the memory address 0000H on the address bus, the instruction at that location is fetched, and the execution of the Key Monitor program begins. Therefore, the Key Monitor program is stored on page 00H.

3. What is a monitor program?

In a single-board microcomputer with a Hex keyboard, the instructions are entered in R/W memory through the keyboard. The Key Monitor program is a set of instructions that continuously checks whether a key is pressed and stores the binary equivalent of a pressed key in a memory location.

4. What is an assembler?

An assembler is a program that translates the mnemonics into their machine code. It is generally not available on a single-board microcomputer.

A program can be entered in mnemonics in a microcomputer equipped with an ASCII keyboard. The assembler will translate mnemonics into the 8085 machine code and assign memory locations to each machine code, thus avoiding the manual assembly and the errors associated with it. Additional instructions can be inserted anywhere in the program, and the assembler will reassign all the new memory locations and jump locations.

5. How does the microprocessor know what operation to perform first (Read/Write memory or Read/Write I/O)?

The first operation is always a Fetch instruction.

6. How does the microprocessor differentiate among a positive number, a negative number, and a bit pattern?

It does not know the difference. The microprocessor views any data byte as eight binary digits. The programmer is responsible for providing the interpretation.

For example, after an arithmetic or logic operation, if the bits in the accumulator are

$$1\ 1\ 1\ 1\ 0\ 0\ 1\ 0 = F2H$$

the Sign flag is set because $D_7 = 1$. This does not mean it is a negative number, even if the Sign flag is set. The Sign flag indicates only that $D_7 = 1$. The eight bits in the ac-

cumulator could be a bit pattern, or a positive number larger than 127_{10} , or the 2's complement of a number.

7. If flags are individual flip-flops, can they be observed on an oscilloscope?

No, they cannot be observed on an oscilloscope. The flag register is internal to the microprocessor. However, they can be tested through conditional branch instructions, and they can be examined by storing them on the stack memory (see Chapter 9).

8. If the program counter is always one count ahead of the memory location from which the machine code is being fetched, how does the microprocessor change the sequence of program execution with a Jump instruction?

When a Jump instruction is fetched, its second and third bytes (a new memory location) are placed in the W and Z registers of the microprocessor. After the execution of the Jump instruction, the contents of the W and Z registers are placed on the address bus to fetch the instruction from a new memory location, and the program counter is loaded by updating the contents of the W and Z registers.

SUMMARY

The instructions from the 8085 instruction set introduced in this chapter are summarized below to provide an overview. After careful examination of these instructions, you will begin to see a pattern emerge from the mnemonics, the number of bytes required for the various instructions, and the tasks the 8085 can perform. Read the notations (Rs) as the contents of the source register, (Rd) as the contents of the destination register, (A) as the contents of the accumulator, and (R) as the contents of the register R.*

Instructions	Tasks	Addressing Mode
<i>Data transfer (Copy) Instructions</i>		
1. MOV Rd,Rs	Copy (Rs) into (Rd).	Register
2. MVI R,8-bit	Load register R with the 8-bit data.	Immediate
3. IN 8-bit port address	Read data from the input port.	Direct
4. OUT 8-bit port address	Write data in the output port.	Direct
<i>Arithmetic Instructions</i>		
1. ADD R	Add (R) to (A).	Register
2. ADI 8-bit	Add 8-bit data to (A).	Immediate
3. SUB R	Subtract (R) from (A).	Register
4. SUI 8-bit	Subtract 8-bit data from (A).	Immediate

*R, Rs, and Rd represent any one of the 8-bit registers—A, B, C, D, E, H, and L.

5. INR R	Increment (R).	Register
6. DCR R	Decrement (R).	Register
<i>Logic Instructions</i>		
1. ANA R	Logically AND (R) with (A).	Register
2. ANI 8-bit	Logically AND 8-bit data with (A).	Immediate
3. ORA R	Logically OR (R) with (A).	Register
4. ORI 8-bit	Logically OR 8-bit data with (A).	Immediate
5. XRA R	Logically Exclusive-OR (R) with (A).	Register
6. XRI 8-bit	Logically Exclusive-OR 8-bit data with (A).	Immediate
7. CMA	Complement (A).	
<i>Branch Instructions</i>		
1. JMP 16-bit	Jump to 16-bit address unconditionally.	Immediate
2. JC 16-bit	Jump to 16-bit address if the CY flag is set.	Immediate
3. JNC 16-bit	Jump to 16-bit address if the CY flag is reset.	Immediate
4. JZ 16-bit	Jump to 16-bit address if the Zero flag is set.	Immediate
5. JNZ 16-bit	Jump to 16-bit address if the Zero flag is reset.	Immediate
6. JP 16-bit	Jump to 16-bit address if the Sign flag is reset.	Immediate
7. JM 16-bit	Jump to 16-bit address if the Sign flag is set.	Immediate
8. JPE 16-bit	Jump to 16-bit address if the Parity flag is set.	Immediate
9. JPO 16-bit	Jump to 16-bit address if the Parity flag is reset.	Immediate
<i>Machine Control Instructions</i>		
1. NOP	No operation.	
2. HLT	Stop processing and wait.	

The set of instructions listed here is used frequently in writing assembly language programs. The important points to be remembered about these instructions are as follows:

1. The data transfer (copy) instructions copy the contents of the source into the destination without affecting the source contents.
2. The results of the arithmetic and logic operations are usually placed in the accumulator.
3. The conditional Jump instructions are executed according to the flags set after an operation. Not all instructions set the flags; in particular, the data transfer instructions do not set the flags.

See Section 2.5 for an overview of the instruction set and see inside the cover page for a complete instruction set according to the functions.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

Note: To execute the instructions in the following assignments and use the Examine Register key, the HLT instruction must be replaced by an appropriate instruction for your system. In the Intel SDK-85 system, the HLT can be replaced by the RST1 (code CFH) instruction. If the HLT instruction is used, the system must be Reset to exit from the Halt instruction; that clears the registers.

Section 6.1: Data Transfer (Copy) Operations

1. Specify the contents of the registers and the flag status as the following instructions are executed.

A	B	C	D	S	Z	CY
---	---	---	---	---	---	----

MVI A,00H
MVI B,F8H
MOV C,A
MOV D,B
HLT

2. Assemble the instructions in Assignment 1, enter the code in R/W memory of a single-board microcomputer, and execute the instructions using the single-step key. Before you begin to execute the instructions, record the initial conditions of the registers and the flags using the Examine Register key. Observe the register contents and the flags as you execute each instruction.
3. Write instructions to load the hexadecimal number 65H in register C, and 92H in the accumulator A. Display the number 65H at PORT0 and 92H at PORT1.
4. Write instructions to read the data at input PORT 07H and at PORT 08H. Display the input data from PORT 07H at output PORT 00H, and store the input data from PORT 08H in register B.
5. Specify the output at PORT1 if the following program is executed.

MVI B,82H
MOV A,B
MOV C,A
MVI D,37H
OUT PORT1
HLT

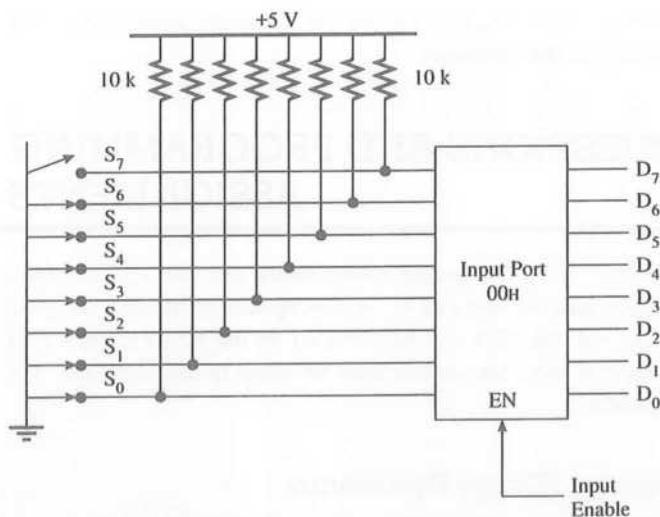


FIGURE 6.14
Input Port with Switches

6. If the switch \$S_7\$ of the input PORT0 (see Figure 6.14) connected to the data line \$D_7\$ is at logic 1 and other switches are at logic 0, specify the contents of the accumulator when the instruction IN PORT0 is executed.

```

MVI A,9FH
IN PORT0
MOV B,A
OUT PORT1
HLT

```

7. Specify the output at PORT1 and the contents of register B after executing the instructions in Assignment 6.

Section 6.2: Arithmetic Operations

8. Specify the register contents and the flag status as the following instructions are executed. Specify also the output at PORT0.

A	B	S	Z	CY	
00	FF	0	1	0	Initial Contents

```

MVI A,F2H
MVI B,7AH
ADD B
OUT PORT0
HLT

```

9. What operation can be performed by using the instruction ADD A?
10. What operation can be performed by using the instruction SUB A? Specify the status of Z and CY.
11. Specify the register contents and the flag status as the following instructions are executed.

A	C	S	Z	CY	
XX	XX	0	0	0	Initial Contents

MVI A,5EH

ADI A2H

MOV C,A

HLT

12. Assemble the program in Assignment 11, and enter the code in R/W memory of your system. Reset the system, and examine the contents of the flag register using the Examine Register key. Clear the flags by inserting 00H in the flag register, and execute each instruction using the single-step key. Verify the register contents and the flags as each instruction is being executed.
13. Write a program using the ADI instruction to add the two hexadecimal numbers 3AH and 48H and to display the answer at an output port.
14. Write instructions to
 - a. load 00H in the accumulator.
 - b. decrement the accumulator.
 - c. display the answer.
 Specify the answer you would expect at the output port.
15. The following instructions subtract two unsigned numbers. Specify the contents of register A and the status of the S and CY flags. Explain the significance of the sign flag if it is set.

MVI A,F8H

SUI 69H

16. Specify the register contents and the flag status as the following instructions are executed.

A	B	S	Z	CY	
XX	XX	X	X	X	

SUB A

MOV B,A

DCR B

INR B

SUI 01H

HLT

17. Assemble the program in Assignment 16, and execute each instruction using the Single-Step key. Verify the register contents and the flags as each instruction is being executed.
18. Write a program to
 - a. clear the accumulator.
 - b. add 47H (use ADI instruction).
 - c. subtract 92H.
 - d. add 64H.
 - e. display the results after subtracting 92H and after adding 64H.Specify the answers you would expect at the output ports.
19. Specify the reason for clearing the accumulator before adding the number 47H directly to the accumulator in Assignment 18.

Section 6.3: Logic Operations

20. What operation can be performed by using the instruction XRA A (Exclusive-OR the contents of the accumulator with itself)? Specify the status of Z and CY.
21. Specify the register contents and the flag status (S, Z, CY) after the instruction ORA A is executed.

```
MVI A,A9H  
MVI B,57H  
ADD B  
ORA A
```

22. Assemble the program in Assignment 21 by adding an End instruction (such as RST1 in Intel's SDK-85 system), and execute the program. Verify the register contents and the flags by using the Examine Register key.
23. When the microprocessor reads an input port, the instruction IN does not set any flag. If the input reading is zero, what logic instruction can be used to set the Zero flag without affecting the contents of the accumulator?
24. Specify the register contents and the flag status as the following instructions are executed.

A	B	S	Z	CY
XX	XX	X	X	X

```
XRA A  
MVI B,4AH  
SUI 4FH  
ANA B  
HLT
```

25. Assemble the instructions in Assignment 24. Execute each instruction using the Single-Step key, and verify the register contents and flags.

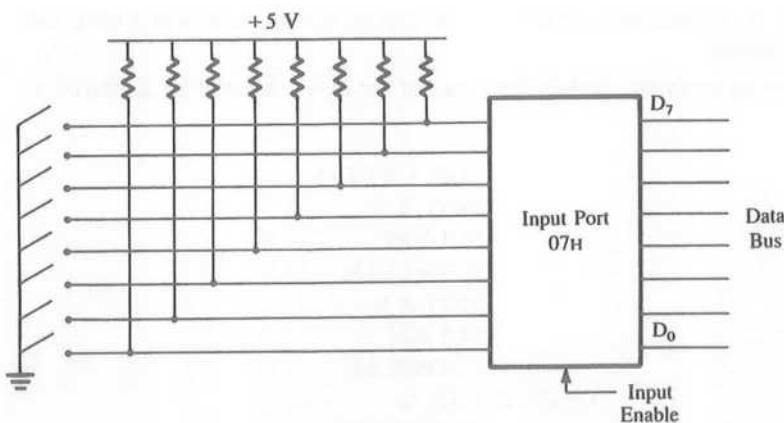


FIGURE 6.15
DIP Switch Input Port with Address 07H

26. Load the data byte A8H in register C. Mask the high-order bits (D₇–D₄), and display the low-order bits (D₃–D₀) at an output port.
27. Load the data byte 8EH in register D and F7H in register E. Mask the high-order bits (D₇–D₄) from both the data bytes, Exclusive-OR the low-order bits (D₃–D₀), and display the answer.
28. Load the bit pattern 91H in register B and 87H in register C. Mask all the bits except D₀ from registers B and C. If D₀ is at logic 1 in both registers, turn on the light connected to the D₀ position of output port 01H; otherwise, turn off the light.
29. Figure 6.15 shows an input port with an 8-key DIP switch. When all switches are off, the microprocessor reads the data FFH. When a switch is turned on (closed), it goes to logic 0 (for all switches ON, the data will be 00H). Write instructions to read the input port and, if all switches are open, set the Zero flag. (Use the instruction CMA to complement the input reading and ORA A to set the Zero flag.)

Section 6.4: Branch Operations

30. What is the output at PORT1 when the following instructions are executed?

```

MVI A,8FH
ADI 72H
JC DSPLAY
OUT PORT1
HLT
DSPLAY: XRA A
        OUT PORT1
        HLT
    
```

31. In Question 30, replace the instruction ADI 72H by the instruction SUI 67H, and specify the output.
32. In the following program, explain the range of the bytes that will be displayed at PORT2.

```

MVI A,BYTE1
MOV B,A
SUI 50H
JC DELETE
MOV A,B
SUI 80H
JC DISPLAY
DELETE: XRA A
OUT PORT1
HLT
DISPLAY: MOV A,B
OUT PORT2
HLT

```

33. Specify the address of the output port, and explain the type of numbers that can be displayed at the output port.

```

MVI A,BYTE1      ;Get a data byte
ORA A            ;Set flags
JP OUTPRT        ;Jump if the byte is positive
XRA A
OUTPRT: OUT F2H
HLT

```

34. In Question 33, if BYTE1 = 92H, what is the output at port F2H?
35. Explain the function of the following program.

```

MVI A,BYTE1      ;Get a data byte
ORA A            ;Set flags
JM OUTPRT
OUT 01H
HLT
OUTPRT: CMA      ;Find 2's complement
ADI 01H
OUT 01H
HLT

```

36. In Question 35, if BYTE1 = A7H, what will be displayed at port 01H?
37. Rewrite the Section 6.4.4 illustrative program for testing the Carry flag using the instruction JC (Jump On Carry).

38. Write instructions to clear the CY flag, to load number FFH in register B, and increment (B). If the CY flag is set, display 01 at the output port; otherwise, display the contents of register B. Explain your results.
39. Write instructions to clear the CY flag, to load number FFH in register C, and to add 01 to (C). If the CY flag is set, display 01 at an output port; otherwise, display the contents of register C. Explain your results. Are they the same as in Question 38?
40. Write instructions to load two unsigned numbers in register B and register C, respectively. Subtract (C) from (B). If the result is in 2's complement, convert the result in absolute magnitude and display it at PORT1; otherwise, display the positive result. Execute the program with the following sets of data.

Set 1: (B) = 42H, (C) = 69H

Set 2: (B) = 69H, (C) = 42H

Set 3: (B) = F8H, (C) = 23H

Section 6.6: Debugging a Program

41. In Section 6.5.3, a program for controlling manufacturing processes is examined, all the errors are marked as 1 through 5. Rewrite the program with the errors corrected.
42. To test this program, substitute the instructions IN F1H and IN F2H by loading the two data bytes 97H and 85H in registers D and E. Rewrite the program to include other appropriate changes. Enter and execute the program on your system.
43. In the program presented in Section 6.5.3, assume that an LED indicator is connected to the output line D₇ of the port F3. Modify the program to turn on the LED when switch S₇ from port F1 is turned on, even if switch S_{7'} is off.

7

Programming Techniques with Additional Instructions

A computer is at its best, surpassing human capability, when it is asked to repeat such simple tasks as adding thousands of numbers. It does this accurately with electronic speed and without showing any signs of boredom. The programming techniques—such as looping, counting, and indexing—required for repetitious tasks are introduced in this chapter.

Data needed for repetitious tasks generally are stored in the system's R/W memory. The data must be transferred (copied) from memory to the microprocessor for manipulation (processing). The instructions related to data manipulations and data transfer (copy) between memory and the microprocessor are introduced in this chapter, as well as instructions related to 16-bit data and additional logic operations. Applications of these instructions are shown in five illustrative programs. The chapter concludes with a discussion of dynamic debugging techniques.

OBJECTIVES

- Draw a flowchart of a conditional loop illustrating indexing and counting.
- List the seven blocks of a generalized flowchart illustrating data acquisitions and data processing.
- Explain the functions of the 16-bit data transfer instructions LXI and of the arithmetic instructions INX and DCX.
- Explain the functions of memory-related data transfer instructions, and illustrate how a memory location is specified using the indirect and the direct addressing modes.
- Write a program to illustrate an application of instructions related to memory data transfer and 16-bit data.
- Explain the functions of arithmetic instructions related to data in memory: ADD/SUB M. Write a program to perform arithmetic operations that generate carry.

- Explain the functions and the differences between the four instructions: RLC, RAL, RRC, and RAR. Write a program to illustrate uses of these instructions.
- Explain the functions of the Compare instructions: CMP and CPI and the flags set under various conditions. Write a program to illustrate uses of the Compare instructions.
- Explain the term *dynamic debugging* and the debugging techniques: Single Step and Breakpoint.

7.1

PROGRAMMING TECHNIQUES: LOOPING, COUNTING, AND INDEXING

The programming examples illustrated in previous chapters are simple and can be solved manually. However, the computer surpasses manual efficiency when tasks must be repeated, such as adding a hundred numbers or transferring a thousand bytes of data. It is fast and accurate.

The programming technique used to instruct the microprocessor to repeat tasks is called **looping**. A loop is set up by instructing the microprocessor to change the sequence of execution and perform the task again. This process is accomplished by using Jump instructions. In addition, techniques such as counting and indexing (described below) are used in setting up a loop.

Loops can be classified into two groups:

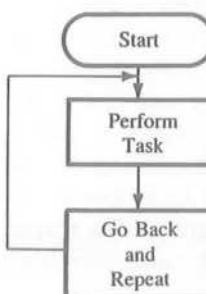
- Continuous loop—repeats a task continuously
- Conditional loop—repeats a task until certain data conditions are met

They are described in the next two sections.

7.1.1 Continuous Loop

A continuous loop is set up by using the unconditional Jump instruction shown in the flowchart (Figure 7.1).

FIGURE 7.1
Flowchart of a Continuous Loop



A program with a continuous loop does not stop repeating the tasks until the system is reset. Typical examples of such a program include a continuous counter (see Chapter 8, Section 8.2) or a continuous monitor system.

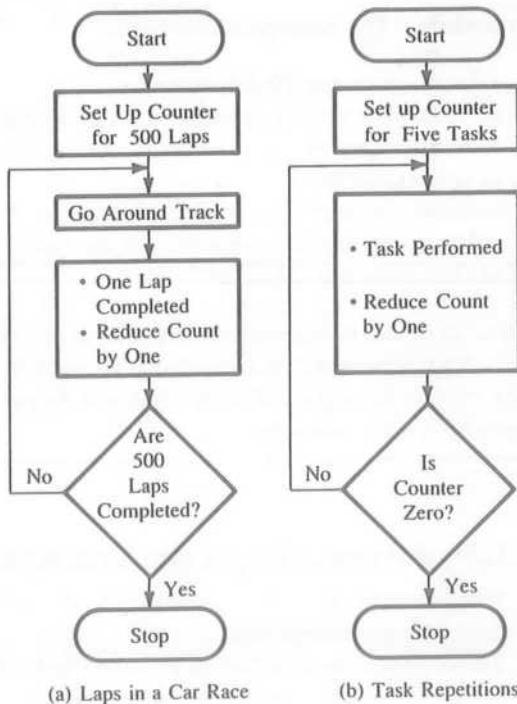
7.1.2 Conditional Loop

A conditional loop is set up by the conditional Jump instructions. These instructions check flags (Zero, Carry, etc.) and repeat the specified tasks if the conditions are satisfied. These loops usually include counting and indexing.

CONDITIONAL LOOP AND COUNTER

A counter is a typical application of the conditional loop. For example, how does the microprocessor repeat a task five times? The process is similar to that of a car racer in the Indy 500 going around the track 500 times. How does the racer know when 500 laps have been completed? The racing team manager sets up a counting and flagging method for the racer. This can be symbolically represented as in Figure 7.2(a). A similar approach is needed for the microprocessor to repeat the task five times. The microprocessor needs a counter, and when the counting is completed, it needs a flag. This can be accomplished with the conditional loop, as illustrated in the flowchart in Figure 7.2(b).

FIGURE 7.2
Flowcharts to Indicate Number of
Repetitions Completed



The computer flowchart of Figure 7.2(b) is translated into a program as follows:

1. Counter is set up by loading an appropriate count in a register.
2. Counting is performed by either incrementing or decrementing the counter.
3. Loop is set up by a conditional Jump instruction.
4. End of counting is indicated by a flag.

It is easier to count down to zero than to count up because the Zero flag is set when the register becomes zero. (Counting up requires the Compare instruction, which is introduced later.)

Conditional Loop, Counter, and Indexing Another type of loop includes indexing along with a counter. (*Indexing* means pointing or referencing objects with sequential numbers. In a library, books are arranged according to numbers, and they are referred to or sorted by numbers. This is called indexing.) Similarly, data bytes are stored in memory locations, and those data bytes are referred to by their memory locations.

**Example
7.1**

Illustrate the steps necessary to add ten bytes of data stored in memory locations starting at a given location, and display the sum. Draw a flowchart.

Procedure The microprocessor needs

- a. a counter to count 10 data bytes
- b. an index or a memory pointer to locate where data bytes are stored
- c. to transfer data from a memory location to the microprocessor (ALU)
- d. to perform addition
- e. registers for temporary storage of partial answers
- f. a flag to indicate the completion of the task
- g. to store or output the result

These steps can be represented in the form of a flowchart as in Figure 7.3.

This generalized flowchart can be used in solving many problems. Some blocks may have to be expanded with additional loops, or some blocks may need to be interchanged in their positions.

7.1.3 Review of Important Concepts

1. Programming is a logical approach to instruct the microprocessor to perform operations in a given sequence.
2. The computer is at its best in repeating tasks. It is fast and accurate.
3. Loops are set up by using the looping technique along with counting and indexing.
4. The computer is a versatile and powerful computing tool because of its capability to set up loops and to make decisions based on data conditions.

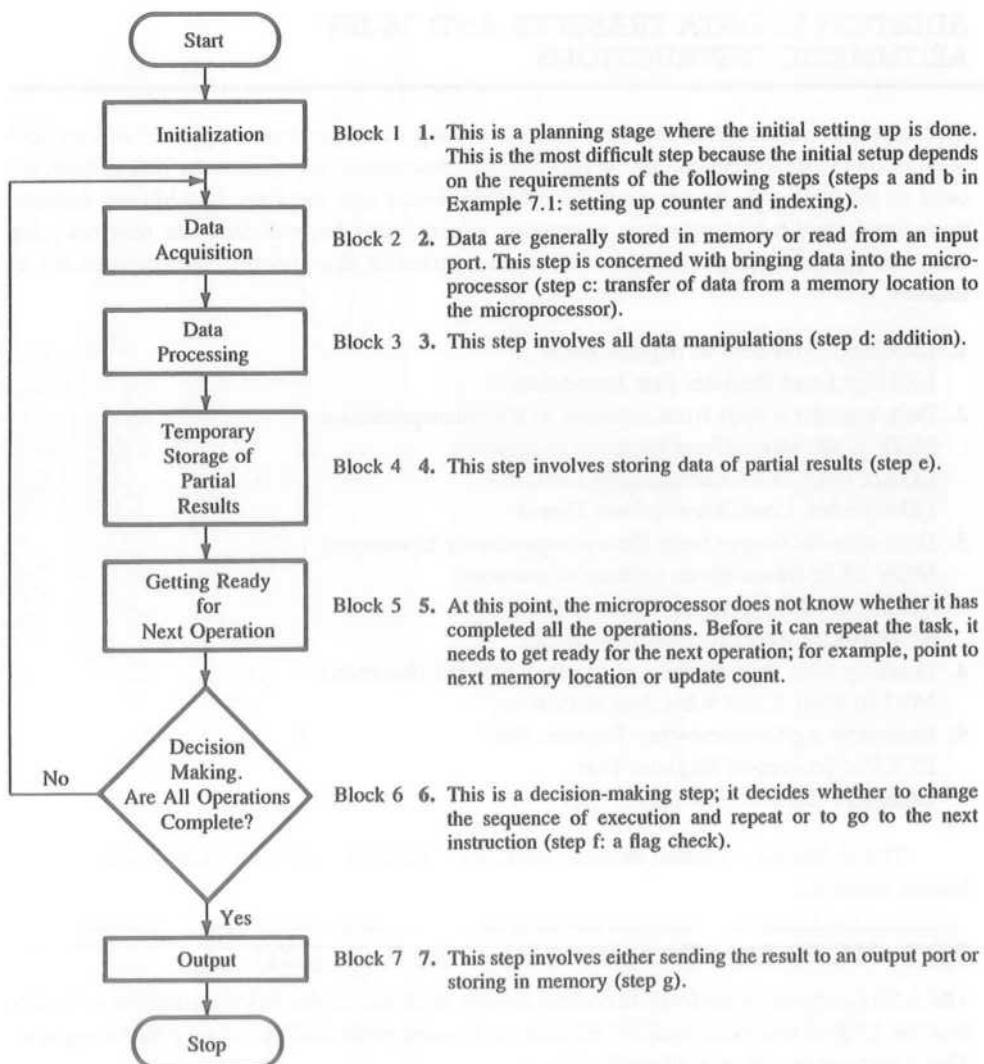


FIGURE 7.3
Generalized Programming Flowchart

LOOKING AHEAD

The programming techniques and the flowcharting introduced in this section will be illustrated with various applications throughout the chapter. Additional instructions necessary for these applications will be introduced first. The primary focus here is to analyze a given programming problem in terms of the basic building blocks of the flowchart shown in Figure 7.3.

See Questions and Programming Assignments 1–4 at the end of the chapter.

7.2

ADDITIONAL DATA TRANSFER AND 16-BIT ARITHMETIC INSTRUCTIONS

The instructions related to the data transfer among microprocessor registers and the I/O instructions were introduced in the last chapter; this section introduces the instructions related to the data transfer between the microprocessor and memory. In addition, instructions for some 16-bit arithmetic operations are included because they are necessary for using the programming techniques introduced earlier in this chapter. The opcodes are as follows:

1. Loading 16-bit data in register pairs
LXI Rp: Load Register Pair Immediate
2. Data transfer (copy) from memory to the microprocessor
MOV R,M: Move (from memory to register)
LDAX B/D: Load Accumulator Indirect
LDA 16-bit: Load Accumulator Direct
3. Data transfer (copy) from the microprocessor to memory
MOV M,R: Move (from register to memory)
STAX B/D: Store Accumulator Indirect
STA 16-bit: Store Accumulator Direct
4. Loading 8-bit data directly in memory register (location)
MVI M,8-bit: Load 8-bit data in memory
5. Incrementing/Decrementing Register Pair
INX Rp: Increment Register Pair
DCX Rp: Decrement Register Pair

The instructions related to these operations are illustrated with examples in the following sections.

7.2.1 16-Bit Data Transfer to Register Pairs (LXI)

The LXI instructions perform functions similar to those of the MVI instructions, except that the LXI instructions load 16-bit data in register pairs and the stack pointer register. These instructions do not affect the flags.

INSTRUCTIONS

Opcode	Operand	
LXI	Rp, 16-bit	Load Register Pair
LXI	B, 16-Bit	<input type="checkbox"/> This is a 3-byte instruction
LXI	D,16-bit	<input type="checkbox"/> The second byte is loaded in the low-order register of the register pair (e.g., register C)
LXI	H,16-bit	<input type="checkbox"/> The third byte is loaded in the high-order register pair (e.g., register B)

LXI SP,16-bit

- There are four such instructions in the set as shown. The operands B, D, and H represent BC, DE, and HL registers, and SP represents the stack pointer register

Write instructions to load the 16-bit number 2050H in the register pair HL using LXI and MVI opcodes, and explain the difference between the two instructions.

Example
7.2

Instructions Figure 7.4 shows the register contents and the instructions required for Example 7.2.

The LXI instruction is functionally similar to two MVI instructions. The LXI instruction takes three bytes of memory and requires ten clock periods (T-states). On the other hand, two MVI instructions take four bytes of memory and require 14 clock periods (T-states).

	Machine Code	Mnemonics	Comments
LXI	21	LXI H,2050H	;Load HL registers
H	50*		;50H in L register and
	20		;20H in H register
MVI	26	MVI H,20H	;Load 20H in register H
H	20		
	2E	MVI L,50H	;Load 50H in register L
	50		

*NOTE: The order of the LXI machine code is reversed in relation to the mnemonics; low-order byte first followed by the high-order byte. This is similar to Jump instructions.

FIGURE 7.4

Instructions and Register Contents for Example 7.2

7.2.2 Data Transfer (Copy) from Memory to the Microprocessor

The 8085 instruction set includes three types of memory transfer instructions; two use the indirect addressing mode and one uses the direct addressing mode. These instructions do not affect the flags.

1. MOV R,M: Move (from Memory to Register)
 - This is a 1-byte instruction

- It copies the data byte from the memory location into a register
 - R represents microprocessor registers A, B, C, D, E, H, and L
 - The memory location is specified by the contents of the HL register
 - This specification of the memory location is indirect; it is called the indirect addressing mode
2. LDAX B/D: Load Accumulator Indirect
- This is a 1-byte instruction
- LDAX B It copies the data byte from the memory location into the accumulator
- LDAX D The instruction set includes two instructions as shown
- The memory location is specified by the contents of the registers BC or DE
 - The addressing mode is indirect
3. LDA 16-bit: Load Accumulator Direct
- This is a 3-byte instruction
 - It copies the data byte from the memory location specified by the 16-bit address in the second and third byte
 - The second byte is a line number (low-order memory address)
 - The third byte is a page number (high-order memory address)
 - The addressing mode is direct

**Example
7.3**

The memory location 2050H holds the data byte F7H. Write instructions to transfer the data byte to the accumulator using three different opcodes: MOV, LDAX, and LDA.

Solution

Figure 7.5 shows the register contents and the instructions required for Example 7.3. All of these three instructions copy the data byte F7H from the memory location 2050H to the accumulator.

In Figure 7.5(a), register HL is first loaded with the 16-bit number 2050H. The instruction MOV A,M uses the contents of the HL register as a memory pointer to location 2050H; this is the indirect addressing mode. The HL register is used frequently as a memory pointer because any instruction that uses M as an operand can copy from and into any one of the registers.

In Figure 7.5(b), the contents of register BC are used as a memory pointer to location 2050H by the instruction LDAX B. Registers BC and DE can be used as restricted memory pointers to copy the contents of only the accumulator into memory and vice versa; however, they cannot be used to copy the contents of other registers.

Figure 7.5(c) illustrates the direct addressing mode; the instruction LDA specifies the memory address 2050H directly as a part of its operand.

After examining all three methods, you may notice that the indirect addressing mode takes four bytes and the direct addressing mode takes three bytes. The question is: Why not just use the direct addressing mode?

If only one byte is to be transferred, the LDA instruction is more efficient. But for a block of memory transfer, the instruction LDA (three bytes) will have to be repeated for

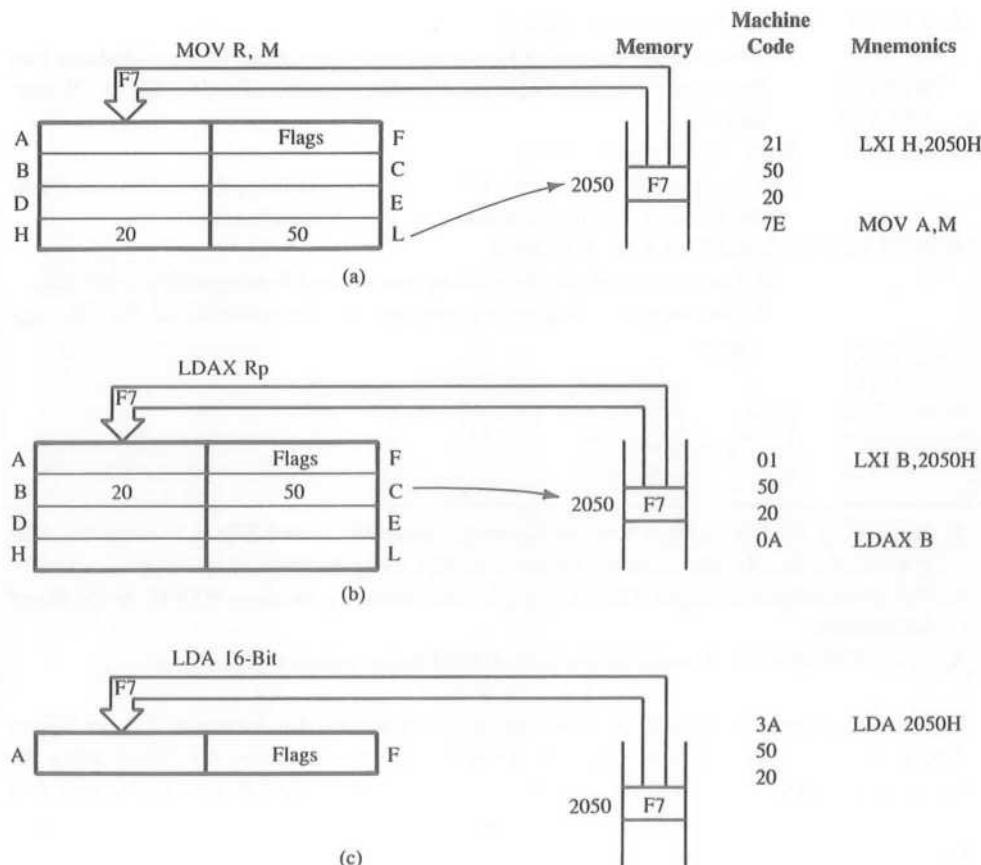


FIGURE 7.5
Instructions and Register Contents for Example 7.3

each memory. On the other hand, a loop can be set up with two other instructions, and the contents of a register pair can be incremented or decremented. This is further illustrated in Section 7.2.6.

7.2.3 Data Transfer (Copy) from the Microprocessor to Memory or Directly into Memory

The instructions for copying data from the microprocessor to a memory location are similar to those described in the previous section. These instructions are as follows:

1. MOV M,R: Move (from Register to Memory).

- This is a 1-byte instruction that copies data from a register, R, into the memory location specified by the contents of HL registers

2. STAX B/D: Store Accumulator Indirect
 - This is a 1-byte instruction that copies data from the accumulator into
STAX B the memory location specified by the contents of either BC or DE reg-
 isters
 - STAX D
3. STA 16-bit: Store Accumulator Direct
 - This is a 3-byte instruction that copies data from the accumulator into
the memory location specified by the 16-bit operand.
4. MVI M,8-bit: Load 8-bit data in memory
 - This is a two-byte instruction; the second byte specifies 8-bit data
 - The memory location is specified by the contents of the HL reg-
ister

**Example
7.4**

1. Register B contains 32H. Illustrate the instructions MOV and STAX to copy the contents of register B into memory location 8000H using indirect addressing.
2. The accumulator contains F2H. Copy (A) into memory location 8000H, using direct addressing.
3. Load F2H directly in memory location 8000H using indirect addressing.

Solution

Figure 7.6 shows the register contents and the instructions for Example 7.4. In Figure 7.6(a), the byte 32H is copied from register B into memory location 8000H by using the HL as a memory pointer. However, in Figure 7.6(b), where the DE register is used as a memory pointer, the byte 32H must be copied from B into the accumulator first because the instruction STAX copies only from the accumulator.

In Figure 7.6(c), the instruction STA copies 32H from the accumulator into the memory location 8000H. The memory address is specified as the operand; this is an illustration of the direct addressing mode. On the other hand, Figure 7.6(d) illustrates how to load a byte directly in memory location by using the HL as a memory pointer.

7.2.4 Arithmetic Operations Related to 16 Bits or Register Pairs

The instructions related to incrementing/decrementing 16-bit contents in a register pair are introduced below. These instructions do not affect flags.

1. INX Rp: Increment Register Pair
 - This is a 1-byte instruction
 - INX B It treats the contents of two registers as one 16-bit number and in-
 creases the contents by 1
 - INX D
 - INX H The instruction set includes four instructions, as shown
 - INX SP

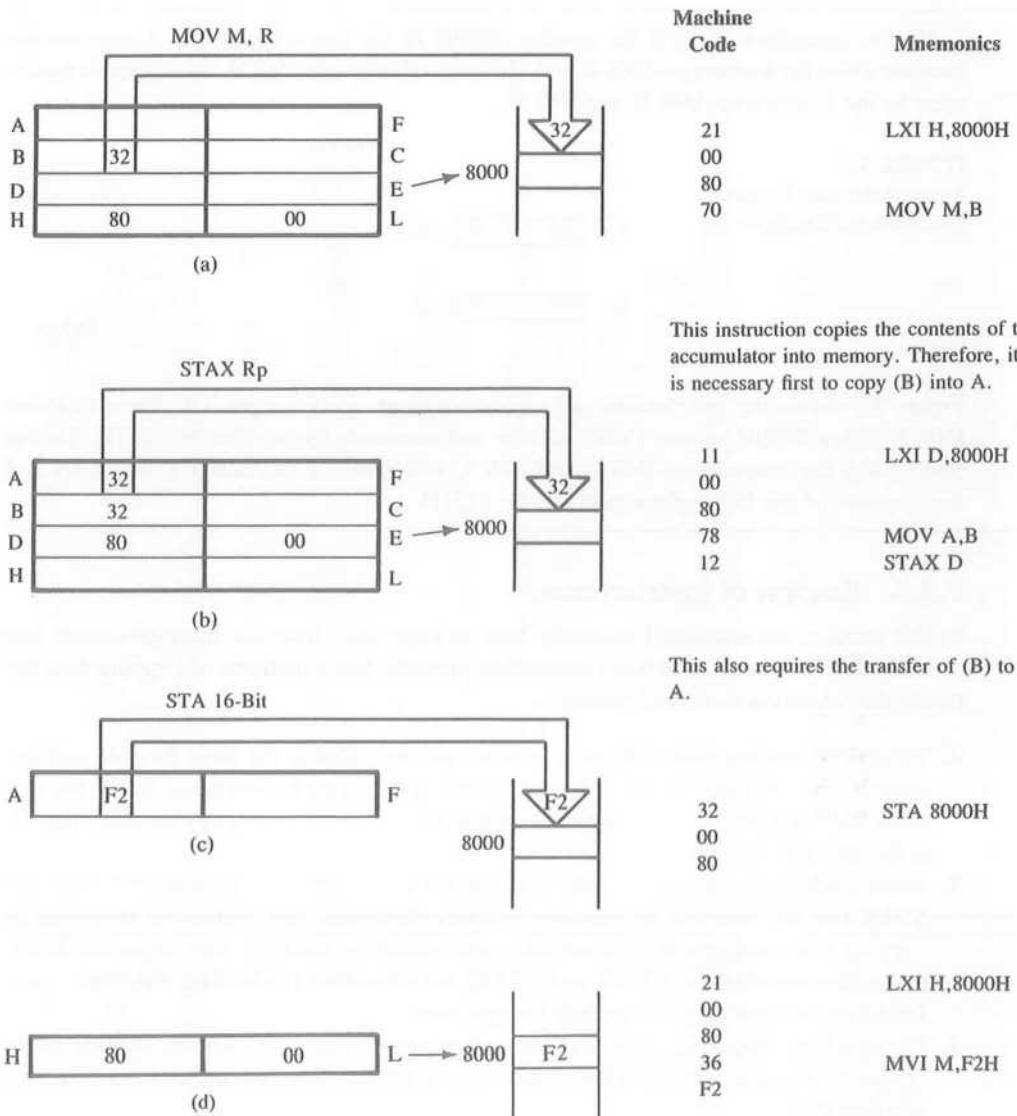


FIGURE 7.6

Instructions and Register Contents for Example 7.4

2. DCX Rp: Decrement Register Pair

- This is a 1-byte instruction

DCX B It decreases the 16-bit contents of a register pair by 1

DCX D The instruction set includes four instructions, as shown

DCX H

DCX SP

**Example
7.5**

Write the instruction to load the number 2050H in the register pair BC. Increment the number using the instruction INX B and illustrate whether the INX B instruction is equivalent to the instructions INR B and INR C.

FIGURE 7.7
Instructions and Register
Contents for Example 7.5

			Machine	Mnemonics
	B	C	Code	
			01	LXI B,2050H
	20	50	50	
			20	
	20	51	03	INX B

Solution

Figure 7.7 shows the instructions and register contents for Example 7.5. The instruction INX B views 2050H as one 16-bit number and increases the number to 2051H. On the other hand, the instructions INR B and INR C will increase (B) and (C) separately and the contents of the BC register pair will be 2151H.

7.2.5 Review of Instructions

In this section, we examined primarily how to copy data from the microprocessor into memory and vice versa. The 8085 instruction provides three methods of copying data between the microprocessor and memory:

1. Indirect addressing using HL as a memory pointer: This is the most flexible and frequently used method. The HL register can be used to copy between any one of the registers and memory. Any instruction with the operand M automatically assumes the HL is the memory pointer.
2. Indirect addressing using BC and DE as memory pointers: The instructions LDAX and STAX use BC and DE as memory pointers. However, this method is restricted to copying from and into the accumulator and cannot be used for other registers. In addition, the mnemonics (LDAX and STAX) are somewhat misleading; therefore, careful attention must be given to their interpretation.
3. Direct addressing using LDA and STA instructions: These instructions include memory address as the operand. This method is also restricted to copying from and into the accumulator.

In addition to the above data copy instructions, we discussed two instructions, INX and DCX, concerning the register pairs. The critical feature of these instructions is that they do not affect the flags.

7.2.6 Illustrative Program: Block Transfer of Data Bytes

PROBLEM STATEMENT

Sixteen bytes of data are stored in memory locations at XX50H to XX5FH. Transfer the entire block of data to new memory locations starting at XX70H.

Data(H) 37, A2, F2, 82, 57, 5A, 7F, DA, E5, 8B, A7, C2, B8, 10, 19, 98

PROBLEM ANALYSIS

The problem can be analyzed in terms of the blocks suggested in the flowchart (Figure 7.8). The steps are as follows:

The flowchart in Figure 7.8 includes five blocks; these blocks are identified with numbers referring to the blocks in the generalized flowchart in Figure 7.3. This problem is not concerned with data manipulation (processing); therefore, the flowchart does not require Blocks 3 and 4 (data processing and temporary storage of partial results). The problem simply deals with the transferring of the data bytes from one location to another location in memory; therefore, the Store Data Byte block is equivalent to the Output block in the generalized flowchart.

Block 1 is the initialization block; this block sets up two memory pointers and one counter. Block 5 is concerned with updating the memory pointers and the counter. The

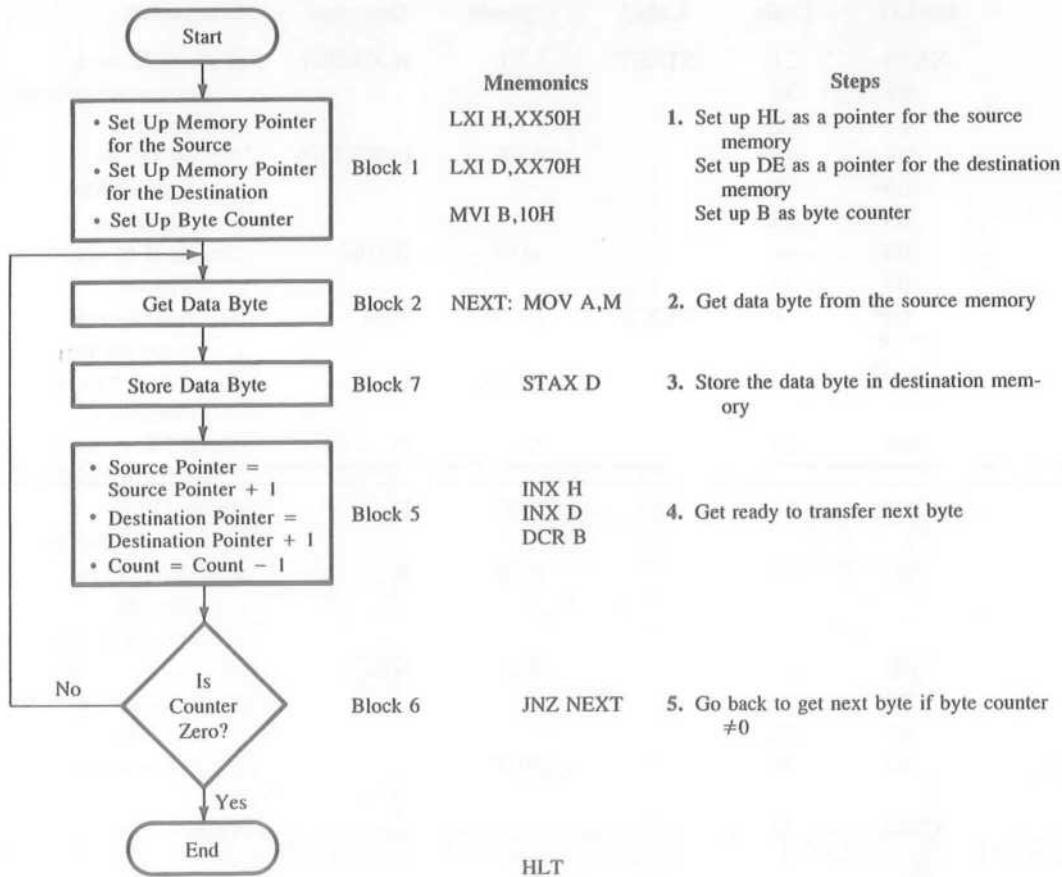


FIGURE 7.8
Flowchart for Block Transfer of Data Bytes

statements shown in the block appear strange if they are read as algebraic equations; however, they are not algebraic equations. The statement Pointer = Pointer + 1 means the new value is obtained by incrementing the previous value by one.

The statements in the flowchart correspond one-to-one with the mnemonics. In large programs, such details in the flowchart are impractical as well as undesirable. However, these details are included here to show the logic flow in writing programs. In Figure 7.8, some of the details can be eliminated very easily from the flowchart. For example, Blocks 2 and 7 can be combined in one statement; such as, Transfer Data Byte from Source to Destination. Similarly, Block 5 can be reduced to one statement; such as, Update memory Pointers and Counter.

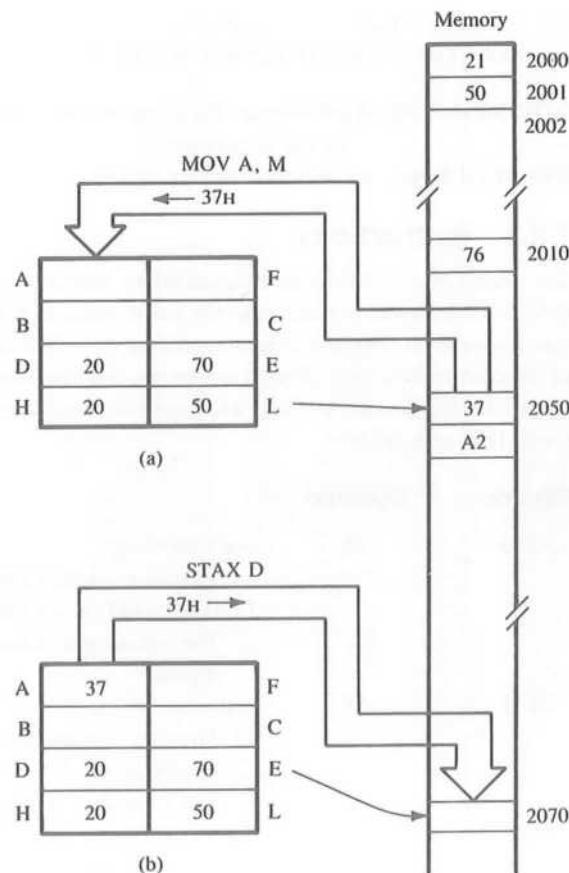
PROGRAM

Memory

Address HI-LO	Hex Code	Label	Instructions		Comments
			Opcode	Operand	
XX00	21	START:	LXI	H,XX50H	;Set up HL as a
01	50				; pointer for source
02	XX				; memory
03	11		LXI	D,XX70H	;Set up DE as
04	70				; a pointer for
05	XX				; destination
06	06		MVI	B,10H	;Set up B to count
07	10				; 16 bytes
08	7E	NEXT:	MOV	A,M	;Get data byte from
					; source memory
09	12		STAX	D	;Store data byte at
					; destination
0A	23		INX	H	;Point HL to next
					; source location
0B	13		INX	D	;Point DE to
					; next destination
0C	05		DCR	B	;One transfer is
					; complete,
0D	C2		JNZ	NEXT	; decrement count
0E	08				;If counter is not 0,
0F	XX				; go back to transfer
10	76		HLT		; next byte
					;End of program
XX50	37				;Data
↓	↓				
XX5F	98				

FIGURE 7.9

Data Transfer from Memory to Accumulator (a), Then to New Memory Location (b)



PROGRAM EXECUTION AND OUTPUT

To execute the program, substitute the page number of your system's R/W memory in place of XX, enter the program and the data, and execute it. To verify the proper execution, check the memory locations from XX70H to XX7FH.

Let us assume the system R/W user memory starts at 2000H. Figure 7.9(a) shows how the contents of the memory location 2050H are copied into the accumulator by the instruction `MOV A,M`; the HL register points to location 2050 and instruction `MOV A,M` copies 37H into A. Figure 7.9(b) shows that the DE register points to the location 2070H and the instruction `STAX D` copies (A) into the location 2070H.

See Questions and Programming Assignments 5–21 at the end of this chapter.

ARITHMETIC OPERATIONS RELATED TO MEMORY

7.3

In the last chapter, the arithmetic instructions concerning three arithmetic tasks—Add, Subtract, and Increment/Decrement—were introduced. These instructions dealt with

microprocessor register contents or numbers. In this chapter, instructions concerning the arithmetic tasks related to memory will be introduced:

ADD M/SUB M: Add/Subtract the contents of a memory location to/from the contents of the accumulator.

INR M/DCR M: Increment/Decrement the contents of a memory location.

7.3.1 Instructions

The arithmetic instructions referenced to memory perform two tasks: one is to copy a byte from a memory location to the microprocessor, and the other is to perform the arithmetic operation. These instructions (other than INR and DCR) implicitly assume that one of the operands is (A); after an operation, the previous contents of the accumulator are replaced by the result. All flags are modified to reflect the data conditions (see the exceptions: INR and DCR).

Opcode	Operand	
ADD	M	Add Memory <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> It adds (M) to (A) and stores the result in A <input type="checkbox"/> The memory location is specified by the contents of HL register
SUB	M	Subtract Memory <input type="checkbox"/> This is a 1-byte instruction <input type="checkbox"/> It subtracts (M) from (A) and stores the result in A <input type="checkbox"/> The memory location is specified by (HL)
INR	M	This is a 1-byte instruction <input type="checkbox"/> It increments the contents of a memory location by 1, not the memory address <input type="checkbox"/> The memory location is specified by (HL) <input type="checkbox"/> All flags except the Carry flag are affected
DCR	M	This is a 1-byte instruction <input type="checkbox"/> It decrements (M) by 1 <input type="checkbox"/> The memory location is specified by (HL) <input type="checkbox"/> All flags except the Carry flag are affected

**Example
7.6**

Write instructions to add the contents of the memory location 2040H to (A), and subtract the contents of the memory location 2041H from the first sum. Assume the accumulator has 30H, the memory location 2040H has 68H, and the location 2041H has 7FH.

Solution

Before asking the microprocessor to perform any memory-related operations, we must specify the memory location by loading the HL register pair. In the example illustrated in Figure 7.10, the contents of the HL pair 2040H specify the memory location. The instruction ADD M adds 68H, the contents of memory location 2040H, to the contents of

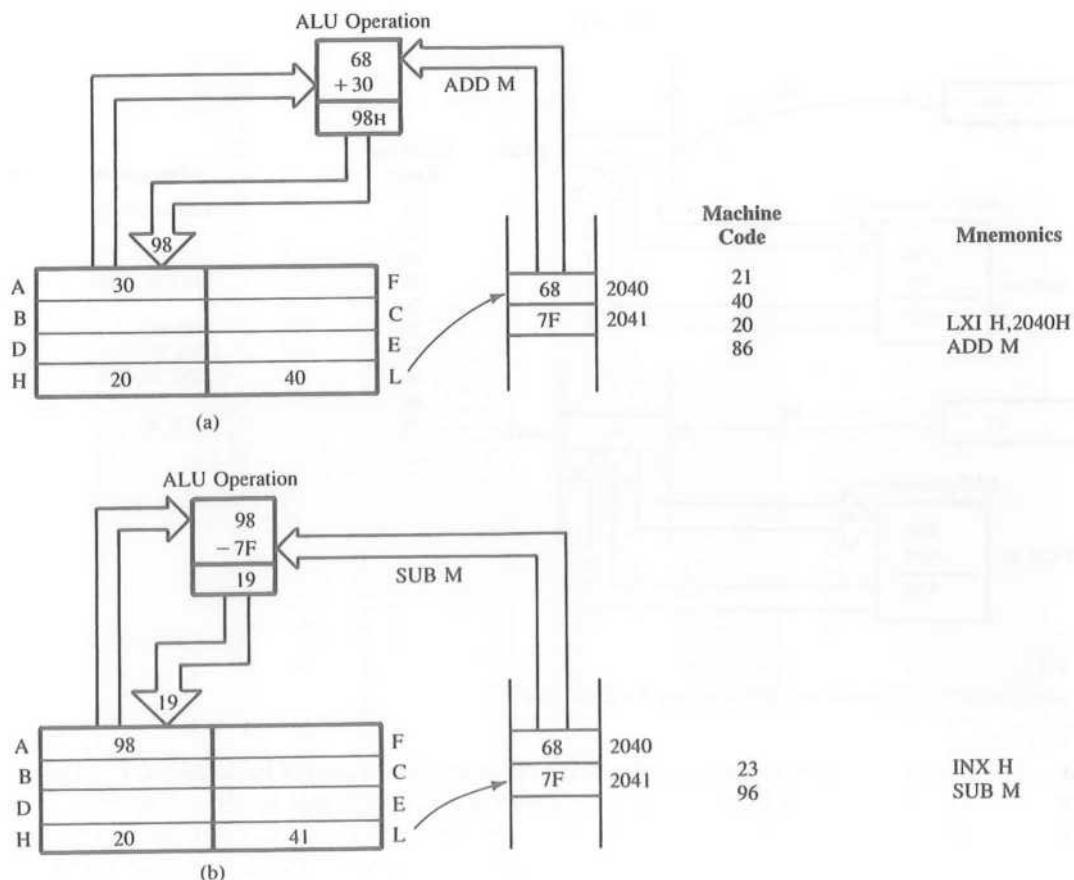


FIGURE 7.10

Register and Memory Contents and Instructions for Example 7.6

the accumulator (30H). The instruction INX H points to the next memory location, 2041H, and the instruction SUB M subtracts the contents (7FH) of memory location 2041H from the previous sum.

Write instructions to

Example
7.7

1. load 59H in memory location 2040H, and increment the contents of the memory location.
2. load 90H in memory location 2041H, and decrement the contents of the memory location.

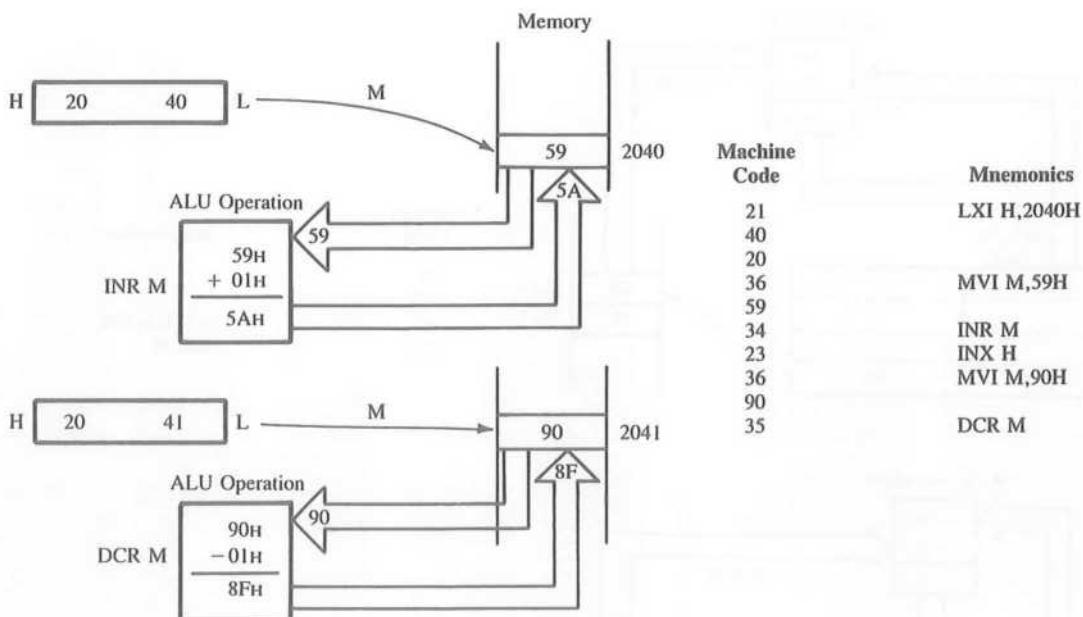


FIGURE 7.11
Register and Memory Contents, and Instructions for Example 7.7

Solution

Figure 7.11 shows register contents and the instructions required for Example 7.7. The instruction MVI M loads 59H in the memory location indicated by (HL). The instruction INR M increases the contents, 59H, of the memory location to 5AH. The instruction INX H increases (HL) to 2041H. The next two instructions load and decrement 90H.

7.3.2 Illustrative Program: Addition with Carry

PROBLEM STATEMENT

Six bytes of data are stored in memory locations starting at XX50H. Add all the data bytes. Use register B to save any carries generated, while adding the data bytes. Display the entire sum at two output ports, or store the sum at two consecutive memory locations, XX70H and XX71H.

Data(H) A2, FA, DF, E5, 98, 8B

PROBLEM ANALYSIS

This problem can be analyzed in relation to the general flowchart in Figure 7.3 as follows:

- Because of the memory-related arithmetic instructions just introduced in this section, two blocks in the general flowchart—data acquisition and data processing—can be combined in one instruction.

2. The fourth block—temporary storage of partial results—is unnecessary because the sum can be stored in the accumulator.
3. The data processing block needs to be expanded to account for carry.

In the first block (Block 1) of the flowchart in Figure 7.12, the accumulator and the carry register (for example, register B) must be cleared in order to use them for arithmetic operations; otherwise, residual data will cause erroneous results.

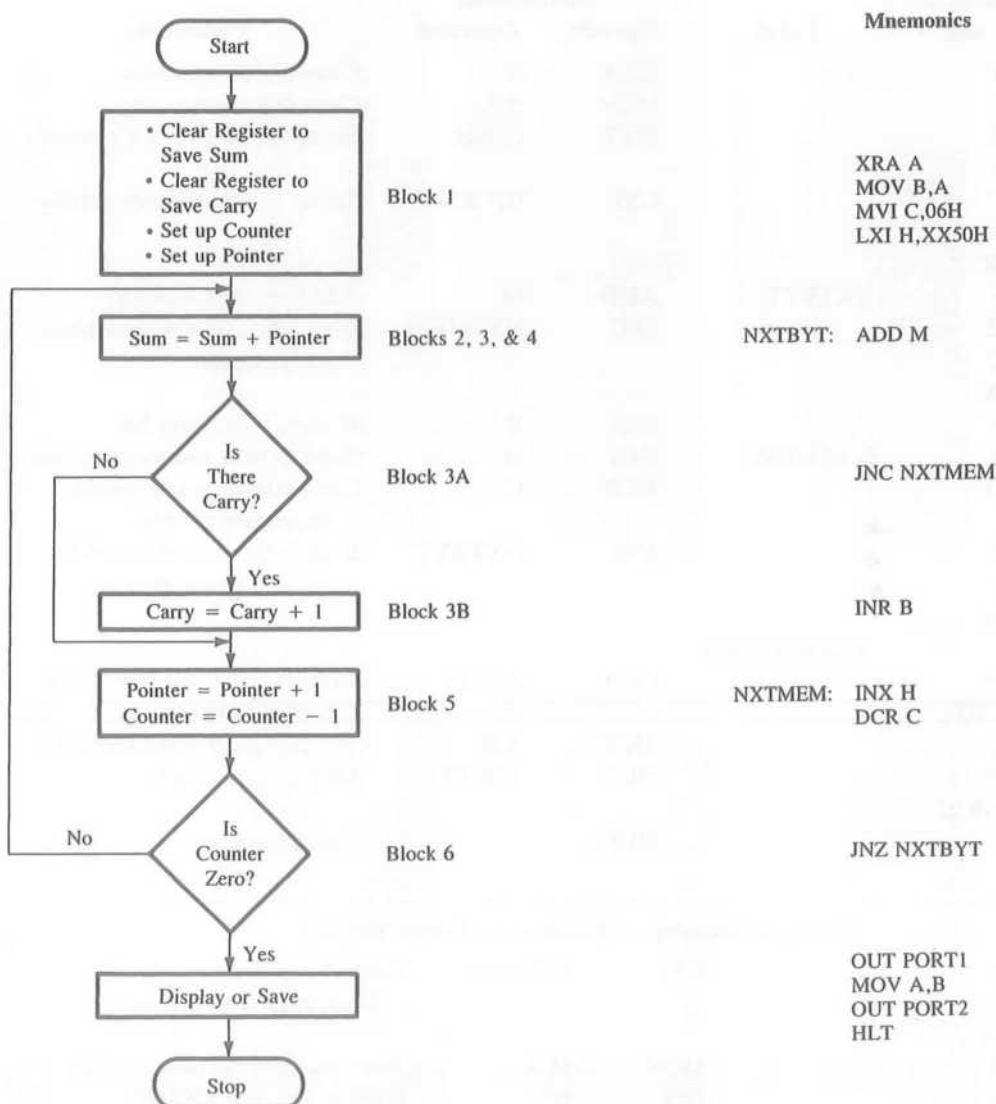


FIGURE 7.12
Flowchart for Addition with Carry

After the addition, it is necessary to check whether that operation has generated a carry (Block 3A). If a carry is generated, the carry register is incremented by one (Block 3B); otherwise, it is bypassed. The instruction ADC (Add with Carry) is inappropriate for this operation. (See Appendix F for the description of the instruction ADC.)

PROGRAM

Memory

Address HI-LO	Machine Code	Label	Instructions		Comments
			Opcode	Operand	
XX00	AF		XRA	A	;Clear (A) to save sum
01	47		MOV	B,A	;Clear (B) to save carry
02	0E		MVI	C,06H	;Set up register C as a counter
03	06				
04	21		LXI	H,XX50H	;Set up HL as memory pointer
05	50				
06	XX				
07	86	NXTBYT:	ADD	M	;Add byte from memory
08	D2		JNC	NXTMEM	;If no carry, do not increment
09	0C				; carry register
0A	XX				
0B	04		INR	B	;If carry, save carry bit
0C	23	NXTMEM:	INX	H	;Point to next memory location
0D	0D		DCR	C	;One addition is completed; ; decrement counter
0E	C2		JNZ	NXTBYT	;If all bytes are not yet added, ; go back to get next byte
0F	07				
10	XX				
		;Output Display			
11	D3		OUT	PORT1	;Display low-order byte of the
12	PORT1				; sum at PORT1
13	78		MOV	A,B	;Transfer carry to accumulator
14	D3		OUT	PORT2	;Display carry digits
15	PORT2				
16	76		HLT		;End of program

;Storing in Memory—Alternative to Output Display

11	21	LXI	H,XX70H	;Point to the memory
12	70			; location to store answer
13	XX			
14	77	MOV	M,A	;Store low-order byte at XX70H
15	23	INX	H	;Point to location XX71H
16	70	MOV	M,B	;Store carry bits
17	76	HLT		;End of program

```
50    A2      ;Data Bytes
51    FA
52    DF
53    E5
54    98
55    8B
```

PROGRAM DESCRIPTION AND OUTPUT

In this program, register B is used as a carry register, register C as a counter to count six data bytes, and the accumulator to add the data bytes and save the partial sum.

After the completion of the summation, the high-order byte (bits higher than eight bits) of the sum is saved in register B and the low-order byte is in the accumulator. Both are displayed at two different ports, or they can be stored at the memory locations XX70H and 71H.

See Questions and Programming Assignments 22–31 at the end of this chapter.

LOGIC OPERATIONS: ROTATE

7.4

In the last chapter, the logic instructions concerning the four operations AND, OR, Ex-OR, and NOT were introduced. This chapter introduces instructions related to rotating the accumulator bits. The opcodes are as follows:

- RLC: Rotate Accumulator Left
- RAL: Rotate Accumulator Left Through Carry
- RRC: Rotate Accumulator Right
- RAR: Rotate Accumulator Right Through Carry

7.4.1 Instructions

This group has four instructions; two are for rotating left and two are for rotating right. The differences between these instructions are illustrated in the following examples.

1. RLC: Rotate Accumulator Left

- Each bit is shifted to the adjacent left position. Bit D₇ becomes D₀.
- CY flag is modified according to bit D₇.

Assume the accumulator contents are AAH and CY = 0. Illustrate the accumulator contents after the execution of the RLC instruction twice.

Example
7.8

Figure 7.13 shows the contents of the accumulator and the CY flag after the execution of the RLC instruction twice. The first RLC instruction shifts each bit to the left by one position, places bit D₇ in bit D₀ and sets the CY flag because D₇ = 1. The accumulator byte AAH becomes 55H after the first rotation. In the second rotation, the byte is again AAH, and the CY flag is reset because bit D₇ of 55H is 0.

Solution

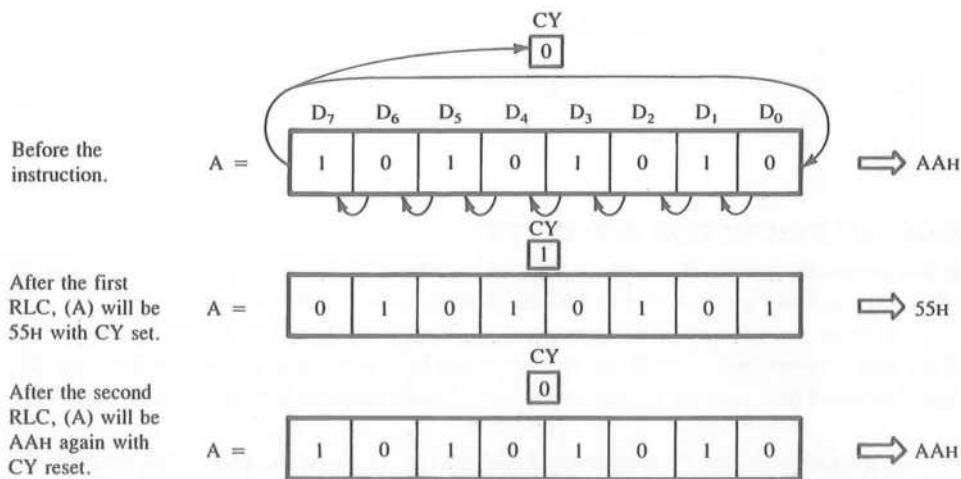


FIGURE 7.13
Accumulator Contents after RLC

2. RAL: Rotate Accumulator Left Through Carry

- Each bit is shifted to the adjacent left position. Bit D₇ becomes the carry bit and the carry bit is shifted into D₀.
- The Carry flag is modified according to bit D₇.

Example 7.9

Assume the accumulator contents are AAH and CY = 0. Illustrate the accumulator contents after the execution of the instruction RAL twice.

Solution

Figure 7.14 shows the contents of the accumulator and the CY flag after the execution of the RAL instruction twice. The first RAL instruction shifts each bit to the left by one position, places bit D₇ in the CY flag, and the CY bit in bit D₀. This is a 9-bit rotation; CY is assumed to be the ninth bit of the accumulator. The accumulator byte AAH becomes 54H after the first rotation. In the second rotation, the byte becomes A9H, and the CY flag is reset.

Examining these two examples, you may notice that the primary difference between these two instructions is that (1) the instruction RLC rotates through eight bits, and (2) the instruction RAL rotates through nine bits.

3. RRC: Rotate Accumulator Right

- Each bit is shifted right to the adjacent position. Bit D₀ becomes D₇.
- The Carry flag is modified according to bit D₀.

4. RAR: Rotate Accumulator Right Through Carry

- Each bit is shifted right to the adjacent position. Bit D₀ becomes the carry bit, and the carry bit is shifted into D₇.

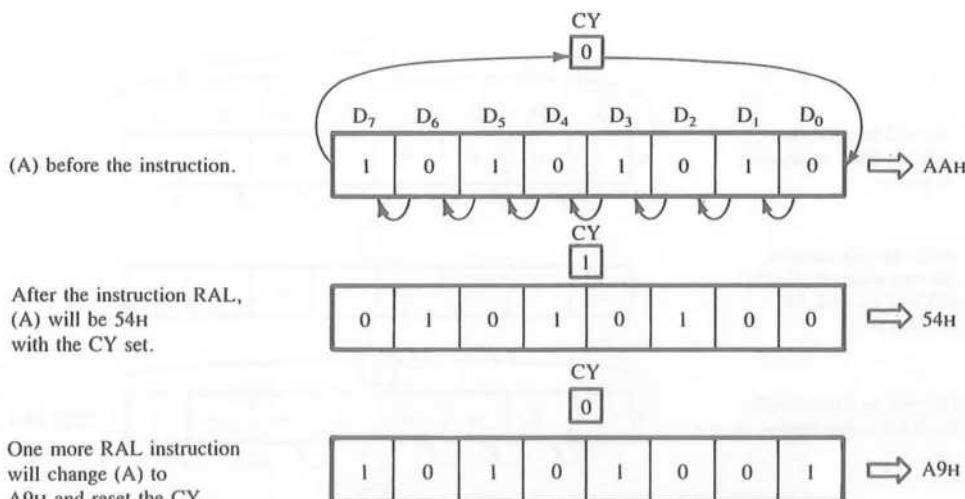


FIGURE 7.14
 Accumulator Contents after RAL

Assume the contents of the accumulator are 81H and CY = 0. Illustrate the accumulator contents after the RRC and RAR instructions.

Example
7.10

Figure 7.15 shows the changes in the contents of the accumulator (81H) when the RRC instruction is used and when the RAR instruction is used. The 8-bit rotation of the RRC instruction changes 81H into C0H, and the 9-bit rotation of the RAR instruction changes 81H into 40H.

Solution

APPLICATIONS OF ROTATE INSTRUCTIONS

The rotate instructions are primarily used in arithmetic multiply and divide operations and for serial data transfer.

For example, if (A) is 0000 1000 = 08H,

- By rotating 08H right: (A) = 0000 0100 = 04H
 This is equivalent to dividing by 2
- By rotating 08H left: (A) = 0001 0000 = 10H
 This is equivalent to multiplying by 2 ($10H = 16_{10}$)

However, these procedures are invalid when logic 1 is rotated left from D₇ to D₀ or vice versa. For example, if 80H is rotated left, it becomes 01H. Applications of serial data transfer are discussed in Chapter 16.

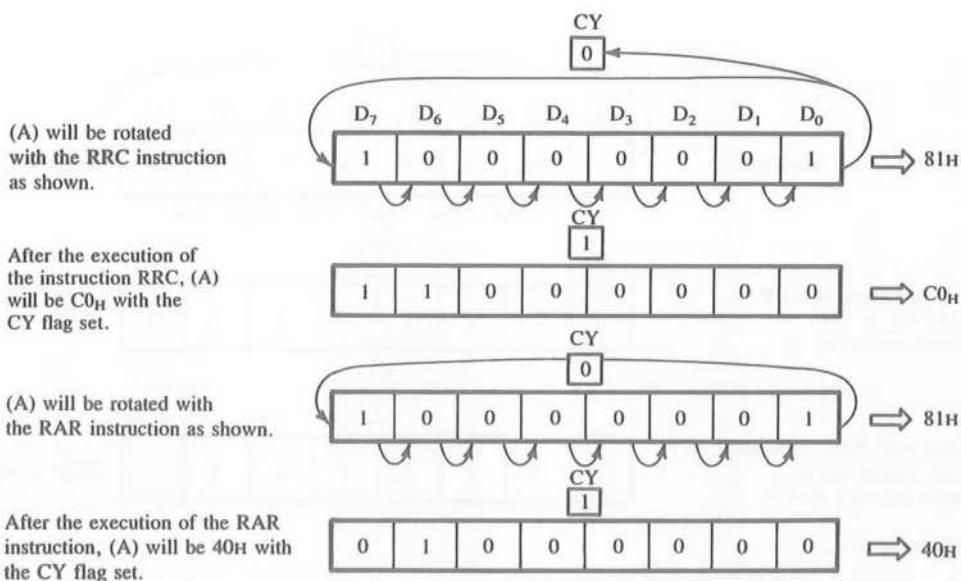


FIGURE 7.15
Rotate Right Instructions

7.4.2 Illustrative Program: Checking Sign with Rotate Instructions

PROBLEM STATEMENT

A set of ten current readings is stored in memory locations starting at XX60H. The readings are expected to be positive ($<127_{10}$). Write a program to

- check each reading to determine whether it is positive or negative.
- reject all negative readings.
- add all positive readings.
- output FFH to PORT1 at any time when the sum exceeds eight bits to indicate overload; otherwise, display the sum. If no output port is available in the system, go to step 5.
- store FFH in the memory location XX70H when the sum exceeds eight bits; otherwise, store the sum.

Data(H) 28, D8, C2, 21, 24, 30, 2F, 19, F2, 9F

PROBLEM ANALYSIS

This problem can be divided into the following steps:

- Transfer a data byte from the memory location to the microprocessor, and check whether it is a negative number. The sign of the number can be verified by rotating bit D₇ into the Carry position and checking for CY. (See Assignment 38 at the end of this chapter to verify the sign of a number with the Sign flag.)
- If it is negative, reject the data and get the next data byte.

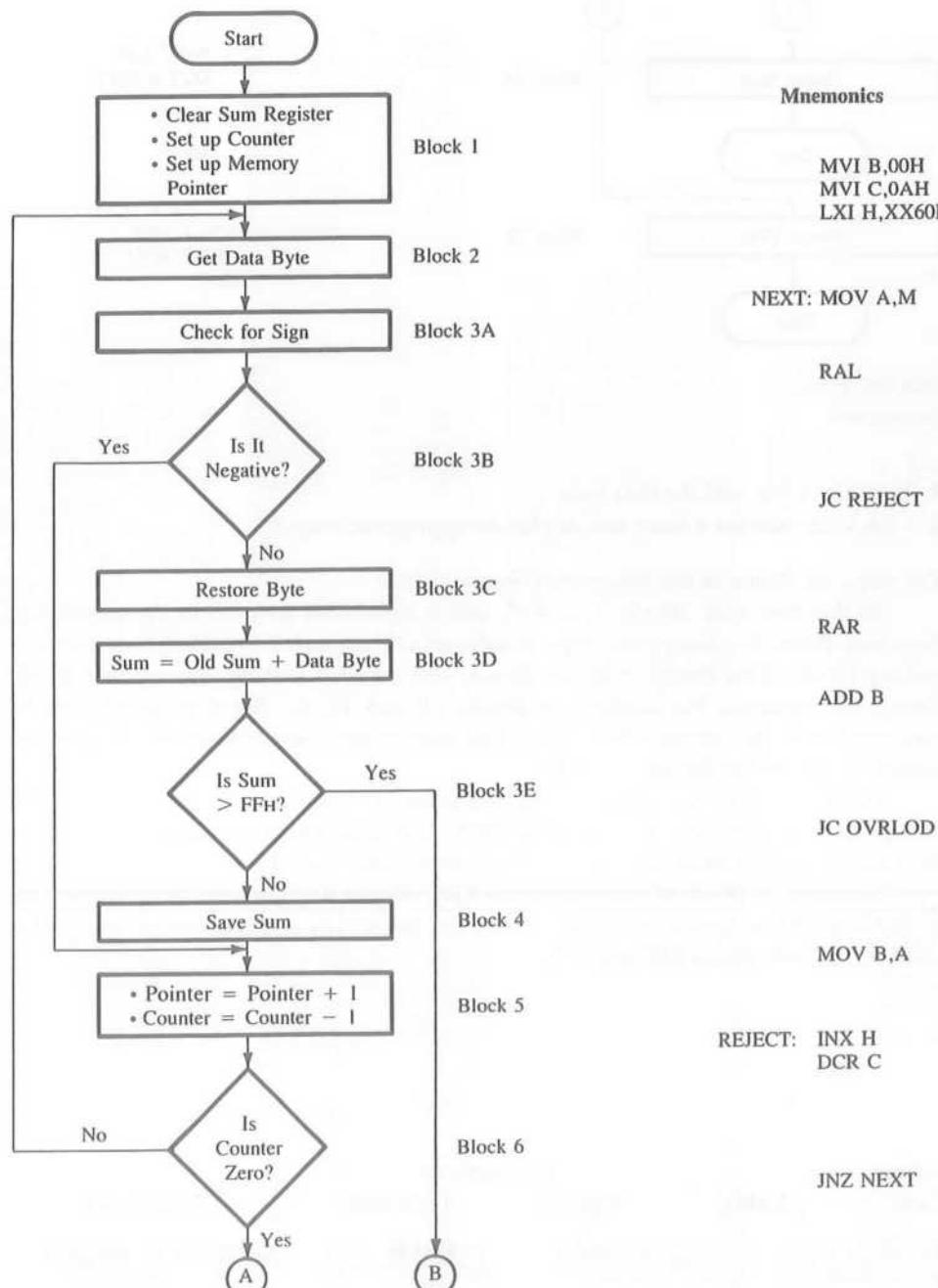


FIGURE 7.16

Flowchart for Checking Sign with Rotate Instructions (*continued on next page*)

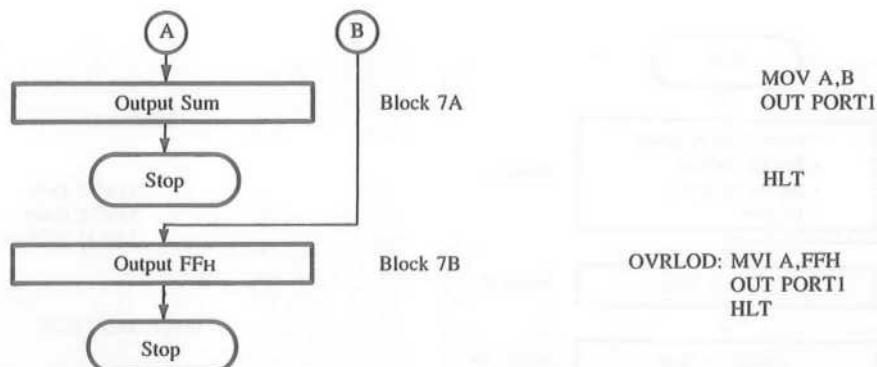


FIGURE 7.16
(continued)

3. If it is positive, add the data byte.
4. Check the sum for a carry and display an appropriate output.

The steps are shown in the flowchart in Figure 7.16.

In this flowchart, Blocks 1, 2, 4, 5, and 6 are similar to those in the generalized flowchart. Block 3—data processing—is substantially expanded by adding two decision-making blocks. This flowchart is first drawn with decision-making answers that do not change the sequence. For example, in Blocks 3B and 3E, the flowchart should first be continued with the answers NO. Then find appropriate locations where the program should be directed to the answers YES.

In this program, the sign of a data byte cannot be checked with the instruction JM (Jump On Minus) because the instruction MOV A,M does not set any flags. (However, the flags can be set by ORing the accumulator contents with itself.)

Similarly, in Block 3C, the instruction RRC (Rotate Right) cannot be used when the instruction RAL is used to rotate left. However, the program can be written using RLC and RRC in both places (3A and 3C).

PROGRAM

Memory

Address

HI-LO

Machine

Code

Label

Instructions

Opcode

Operand

Comments

XX00	06		MVI	B,00H	;Clear (B) to save sum
01	00				
02	0E		MVI	C,0AH	;Set up register C ; as a counter

03	0A				
04	21		LXI	H,XX60H	;Set up HL as memory ; pointer
05	60				
06	XX				
07	7E	NEXT:	MOV	A,M	;Get byte
08	17		RAL		;Shift D ₇ into CY
09	DA		JC	REJECT	;If D ₇ = 1, reject byte and ; go to increment ; pointer
0A	12				
0B	XX				
0C	1F		RAR		;If byte is positive, re- ; store it
0D	80		ADD	B	;Add previous sum to (A)
0E	DA		JC	OVRLOD	;If sum >FFH, it is over- ; load;
0F	1C				; turn on emergency
10	XX				
11	47		MOV	B,A	;Save sum
12	23	REJECT:	INX	H	;Point to next reading
13	0D		DCR	C	;One reading is checked; ; decrement counter
14	C2		JNZ	NEXT	;If all readings are not ; checked, go back to ; transfer next byte
15	07				
16	XX				
		;Output Display Section			
17	78		MOV	A,B	
18	D3		OUT	PORT1	;Display sum
19	PORT1				
1A	76		HLT		;End of program
1B	00		NOP		;To match Jump location, ; OVRLOD in memory ; storage
1C	3E	OVRLOD:	MVI	A,FFH	;It is an overload
1D	FF				
1E	D3		OUT	PORT1	;Display overload signal ; at PORT1
1F	PORT1				
20	76		HLT		

;Storing Result in Memory—Alternative to Output Display					
18	32		STA	XX70H	;Store sum in memory ; XX70H
19	70				
1A	XX				
1B	76				
1C	3E	OVRLOD:	MVI	A,FFH	;Store overload signal in
1D	FF				; memory XX70
1E	32		STA	XX70H	
1F	70				
20	XX				
21	76		HLT		

XX60	28	;Current Readings
61	D8	
62	C2	
63	21	
64	24	
65	30	
66	2F	
67	19	
68	F2	
69	9F	

PROGRAM DESCRIPTION AND OUTPUT

In this program, register C is used as a counter to count ten bytes. Register B is used to save the sum. The sign of the number is checked by verifying whether D₇ is 1 or 0. If the Carry flag is set to indicate the negative sign, the program rejects the number and goes to Block 5, Getting Ready for Next Operation.

The program should reject the data bytes D8, C2, F2, and 9F, and should add the rest. The answer displayed should be E5.

See Questions and Programming Assignments 32–40 at the end of this chapter.

7.5

LOGIC OPERATIONS: COMPARE

The 8085 instruction set has two types of Compare operations: CMP and CPI.

- CMP: Compare with Accumulator
- CPI: Compare Immediate (with Accumulator)

The microprocessor compares a data byte (or register/memory contents) with the contents of the accumulator by subtracting the data byte from (A), and indicates

whether the data byte is $\geq \leq$ (A) by modifying the flags. However, the contents are not modified.

7.5.1 Instructions

1. CMP R/M: Compare (Register or Memory) with Accumulator

- This is a 1-byte instruction.
- It compares the data byte in register or memory with the contents of the accumulator.
- If $(A) < (R/M)$, the CY flag is set and the Zero flag is reset.
- If $(A) = (R/M)$, the Zero flag is set and the CY flag is reset.
- If $(A) > (R/M)$, the CY and Zero flags are reset.
- When memory is an operand, its address is specified by (HL).
- No contents are modified; however, all remaining flags (S, P, AC) are affected according to the result of the subtraction.

2. CPI 8-bit: Compare Immediate with Accumulator

- This is a 2-byte instruction, the second byte being 8-bit data.
- It compares the second byte with (A).
- If $(A) <$ 8-bit data, the CY flag is set and the Zero flag is reset.
- If $(A) =$ 8-bit data, the Zero flag is set, and the CY flag is reset.
- If $(A) >$ 8-bit data, the CY and Zero flags are reset.
- No contents are modified; however, all remaining flags (S, P, AC) are affected according to the result of the subtraction.

Write an instruction to load the accumulator with the data byte 64H, and verify whether the data byte in memory location 2050H is equal to the accumulator contents. If both data bytes are equal, jump to memory location BUFFER.

Example

7.11

Solution

Figure 7.17 illustrates Example 7.11.

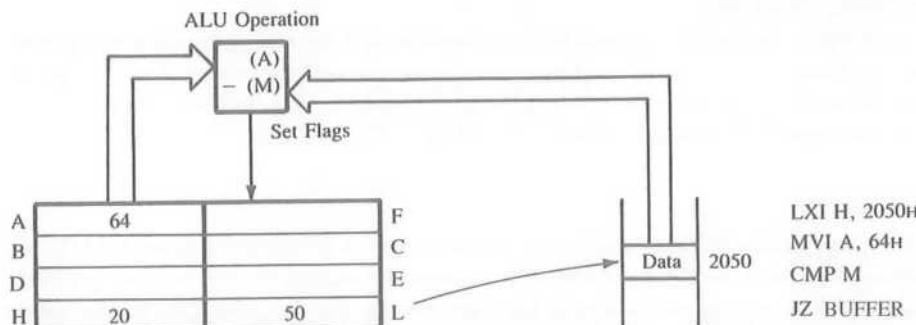


FIGURE 7.17
Compare Instructions

In these instructions, the instruction CMP M selects the memory location pointed out by the HL register (2050H) and compares the contents of that location with (A). If they are equal, the Zero flag is set, and the program jumps to location BUFFER.

Let us assume the data byte in memory location 2050H is 9AH. The microprocessor compares 64H with 9AH by subtracting 9AH from 64H as shown below.

$$\begin{array}{r}
 (A) = 64H \quad 0110\ 0100 \\
 + \\
 \text{2's Complement of 9AH} \quad 0110\ 0110 \\
 \hline
 \text{CY} \quad 0\ 1100\ 1010 \\
 \\
 \text{Complement CY} \quad 1\ 1100\ 1010
 \end{array}$$

The result CAH will modify the flags as S = 1, Z = 0, and CY = 1; however, the original byte in the accumulator 64H will not be changed.

7.5.2 Illustrative Program: Use of Compare Instruction to Indicate End of Data String

PROBLEM STATEMENT

A set of current readings is stored in memory locations starting at XX50H. The end of the data string is indicated by the data byte 00H. Add the set of readings. The answer may be larger than FFH. Display the entire sum at PORT1 and PORT2 or store the answer in the memory locations XX70 and XX71H.

Data(H) 32, 52, F2, A5, 00

PROBLEM ANALYSIS

In this problem, the number of data bytes is variable, and the end of the data string is indicated by loading 00H. Therefore, the counter technique will not be useful to indicate the end of the readings. However, by comparing each data byte with 00H, the end of the data can be determined. This is shown in the flowchart in Figure 7.18.

FLOWCHART

The flowchart shows that after a data byte is transferred, it is first checked for 00H (Block 6). This operation is similar to checking for a negative number in the previous problem. However, this is also an exit point. If the byte is zero, the program goes to the output Block 7. The other significant change is in Block 5 where the unconditional Jump brings the sequence back into the program.

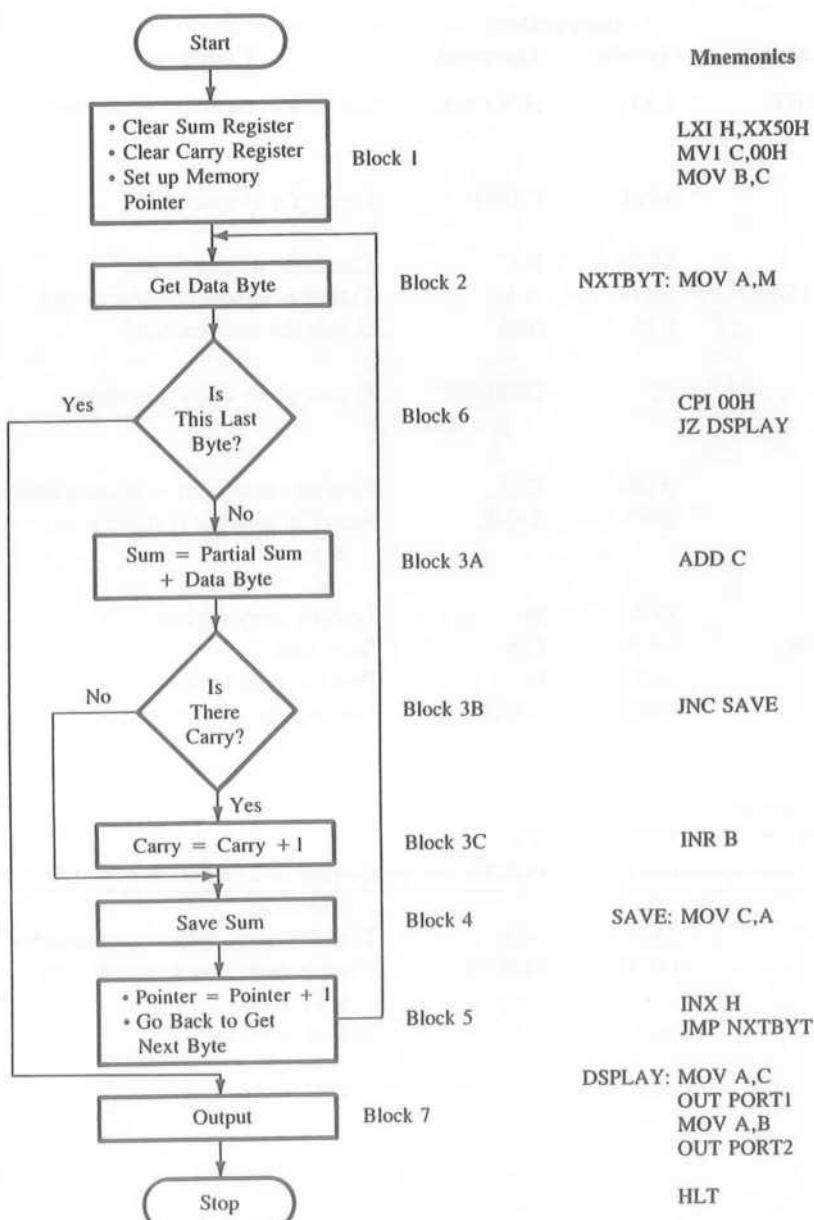


FIGURE 7.18

Flowchart for Compare Instruction to Check End of Data String

PROGRAM

Memory

Address HI-LO	Machine Code	Label	Instruction		Comments
			Opcode	Operand	
XX00	21	START:	LXI	H,XX50H	;Set up HL as memory pointer
01	50				
02	XX				
03	0E		MVI	C,00H	;Clear (C) to save sum
04	00				
05	41		MOV	B,C	;Clear (B) to save carry
06	7E	NXTBYT:	MOV	A,M	;Transfer current reading to (A)
07	FE		CPI	00H	;Is this the last reading?
08	00				
09	CA		JZ	DSPLAY	;If yes; go to display section
0A	16				
0B	XX				
0C	81		ADD	C	;Add previous sum to accumulator
0D	D2		JNC	SAVE	;Skip CY register if there is no
0E	11				; carry
0F	XX				
10	04		INR	B	;Update carry register
11	4F	SAVE:	MOV	C,A	;Save sum
12	23		INX	H	;Point to next reading
13	C3		JMP	NXTBYT	;Go back to get next reading
14	06				
15	XX				
		;Output Display			
16	79	DSPLAY:	MOV	A,C	
17	D3		OUT	PORT1	;Display low-order byte of sum
18	PORT1				; at PORT1
19	78		MOV	A,B	;Transfer carry bits to accumulator
1A	D3		OUT	PORT2	;Display high-order byte of sum
1B	PORT2				; at PORT2
1C	76		HLT		;End of program

		;Storing Result in Memory—Alternative to Output Display			
16	21	DSPLAY:	LXI	H,XX70	;Point index to XX70H location
17	70				
18	XX				
19	71		MOV	M,C	;Store low-order byte of sum ; in XX70H
1A	23		INX	H	;Point index to XX71H
1B	70		MOV	M,B	;Store high-order byte
1C	76		HLT		;End of program

50	32	;Data—Current Readings		
51	52			
52	F2			
53	A5			
54	00			

PROGRAM DESCRIPTION AND OUTPUT

This program adds the first four readings, which results in the sum 21BH. The low-order byte 1BH is saved in register C, and the high-order byte 02H is stored in the carry register B. These are each displayed at different output ports by the OUT instruction or stored in the memory locations XX70 and XX71.

See Questions and Programming Assignments 41–56 at the end of the chapter.

7.5.3 Illustrative Program: Sorting

PROBLEM STATEMENT

A set of three readings is stored in memory starting at XX50H. Sort the readings in ascending order.

Data(H) 87, 56, 42

PROBLEM ANALYSIS

In this problem, three readings should be arranged in ascending order: 42, 56, and 87. The number of readings is kept small to simplify the explanation. However, the programming technique used here, called bubble sort, can be applied to a large set of data. Ordering data in a given sequence such as ascending, descending, or alphabetical order is a common application in programming.

The technique involved is comparing two bytes at a time and placing them in the proper sequence. For example, in our data set, we will compare the first two bytes (87H and 56H), and if the first byte is larger than the second byte, we will exchange their mem-

ory locations to arrange them in ascending order; otherwise, we will keep them in the same locations. We will follow the same procedure for the second and third bytes. The number of comparisons necessary is always $N - 1$ where N is the number of data bytes. Therefore, we need a counter of 2 for one complete comparison. Table 7.1 shows the memory locations and their contents after each pass (execution of one cycle). Two exchanges occur in the first pass, and the bytes are sequenced as 56, 42, and 87. In the second pass, only one exchange occurs. The microprocessor should continue these comparisons until no exchanges occur. We need to devise a technique or set up a reminder for the processor to recognize that no exchanges occur in a given pass. This is called setting a flag. In daily life, we write a note to remind ourselves. Similarly, we can use any register to write a binary note to the microprocessor. For example, we can write 1 (or any number such as FFH) in register D when an exchange takes place; otherwise, register D will be cleared.

PROGRAM

START:	LXI H, XX50H	;Set up HL as a memory pointer for bytes
	MVI D, 00	;Clear register D to set up a flag
	MVI C, 02	;Set register C for comparison count
CHECK:	MOV A, M	;Get data byte
	INX H	;Point to next byte
	CMP M	;Compare bytes
	JC NXTBYT	;If (A) < second byte, do not exchange
	MOV B, M	;Get second byte for exchange
	MOV M, A	;Store first byte in second location
	DCX H	;Point to first location
	MOV M, B	;Store second byte in first location
	INX H	;Get ready for next comparison
	MVI D, 01	;Load 1 in D as a reminder for exchange
NXTBYT:	DCR C	;Decrement comparison count
	JNZ CHECK	;If comparison count $\neq 0$, go back
	MOV A, D	;Get flag bit in A
	RRC	;Place flag bit D_0 in Carry
	JC START	;If flag is 1, exchange occurred
	HLT	;Start the next pass
		;End of sorting

TABLE 7.1
Execution of Sort Program

Memory Locations	Initial Bytes	First Pass	Second Pass	Third Pass
XX50	87	56	42	42
XX51	56	42	56	56
XX52	42	87	87	87
Reg. D	0	1	1	0

PROGRAM DESCRIPTION AND OUTPUT

During the initialization phase, register HL is set up as a pointer where readings are stored; register D is cleared to set up a flag when an exchange occurs; and register C is initialized for a count of two because we have three data bytes to sort. Next is the comparison phase. The programs gets the first byte (87H) in A and compares that with the second byte (56H). In this comparison, the carry flag is reset; therefore, the next set of instructions places 56H in location XX50H and 87H in location XX51H. The flag in register D is set, the comparison counter in register C is decremented by 1, and the program returns to location CHECK for the next comparison. In this second comparison, 87H is compared with 42H, and again the bytes are exchanged. The flag (remainder) in D is set again, but now the comparison counter is zero. Therefore, the program falls through the first loop and checks the status of the flag in D by rotating the flag bit into Carry. Because Carry is set, the program goes back to the beginning and starts the process again from location XX50H. In the second pass, 56H and 42H are exchanged; therefore, the process is repeated a third time. In this pass, every comparison generates a carry. The program jumps to location NXTBYT, and the flag in register D remains zero. Thus, the program places the bytes in ascending order: 42H, 56H, and 87H.

DYNAMIC DEBUGGING

7.6

After you have completed the steps in the process of static debugging (described in the previous chapter), if the program still does not produce the expected output, you can attempt to debug the program by observing the execution of instructions. This is called **dynamic debugging**.

7.6.1 Tools for Dynamic Debugging

In a single-board microcomputer, techniques and tools commonly used in dynamic debugging are

- Single Step
- Register Examine
- Breakpoint

Each will be discussed below; the Single-Step and Register Examine keys were discussed briefly in the previous chapter.

SINGLE STEP

The Single-Step key on a keyboard allows you to execute one instruction at a time, and to observe the results following each instruction. Generally, a single-step facility is built with a hard-wired logic circuit. As you push the Single-Step key, you will be able to observe addresses and codes as they are executed. With the single-step technique you will be able to spot

- incorrect addresses
- incorrect jump locations for loops
- incorrect data or missing codes

To use this technique effectively, you will have to reduce loop and delay counts to a minimum number. For example, in a program that transfers 100 bytes, it is meaningless to set the count to 100 and single-step the program 100 times. By reducing the count to two bytes, you will be able to observe the execution of the loop. (If you reduce the count to one byte, you may not be able to observe the execution of the loop.) By single-stepping the program, you will be able to infer the flag status by observing the execution of Jump instructions. The single-step technique is very useful for short programs.

REGISTER EXAMINE

The Register Examine key allows you to examine the contents of the microprocessor register. When appropriate keys are pressed, the monitor program can display the contents of the registers. This technique is used in conjunction either with the single-step or the breakpoint facilities discussed below.

After executing a block of instructions, you can examine the register contents at a critical juncture of the program and compare these contents with the expected outcomes.

BREAKPOINT

In a single-board computer, the breakpoint facility is, generally, a software routine that allows you to execute a program in sections. The breakpoint can be set in your program by using RST instructions. (See "Interrupts," Chapter 12.) When you push the Execute key, your program will be executed until the breakpoint, where the monitor takes over again. The registers can be examined for expected results. If the segment of the program is found satisfactory, a second breakpoint can be set at a subsequent memory address to debug the next segment of the program. With the breakpoint facility you can isolate the segment of the program with errors. Then that segment of the program can be debugged with the single-step facility. The breakpoint technique can be used to check out the timing loop, I/O section, and interrupts. (See Chapter 12 for how to write a breakpoint routine.)

7.6.2 Common Sources of Errors

Common sources of errors in the instructions and programs illustrated in this chapter are as follows:

1. Failure to clear the accumulator when it is used to add numbers.
2. Failure to clear the carry registers or keep track of a carry.
3. Failure to update a memory pointer or a counter.
4. Failure to set a flag before using a conditional Jump instruction.
5. Inadvertently clearing the flag before using a Jump instruction.
6. Specification of a wrong memory address for a Jump instruction.
7. Use of an improper combination of Rotate instructions.

8. Specifying the Jump instruction on a wrong flag. This is a very common error with the Compare instructions.

See Questions and Programming Assignments 57–59 at the end of the chapter

SUMMARY

In this chapter, programming techniques—such as looping, counting, and indexing—were illustrated using memory-related data transfer instructions, 16-bit arithmetic instructions, and logic instructions. Techniques used commonly in debugging a program—single step, register examine, and breakpoint—were discussed; and common sources of errors were listed.

Review of Instructions

The instructions introduced and illustrated in this chapter are summarized below for an overview.

Instructions	Task	Addressing Mode
<i>Data Transfer (Copy) Instructions</i>		
1. LXI rp,16-bit	Load 16-bit data in a register pair.	Immediate
2. MOV R,M	Copy (M) into (R).	Indirect
3. MOV M,R	Copy (R) into (M).	Indirect
4. LDAX B/D	Copy the contents of the memory, indicated by the register pair, into the accumulator.	Indirect
5. STAX B/D	Copy (A) into the memory, indicated by the register pair.	Indirect
6. LDA 16-bit	Copy (M) into (A), memory specified by the 16-bit address.	Direct
7. STA 16-bit	Copy (A) into memory, specified by the 16-bit address.	Direct
8. MV1 M,8-bit	Load 8-bit data in memory; the memory address is specified by (HL).	Indirect
<i>Arithmetic Instructions</i>		
1. ADD M	Add (M) to (A).	Indirect
2. SUB M	Subtract (M) from (A).	Indirect
3. INR M	Increment the contents of (M) by 1.	Indirect
4. DCR M	Decrement the contents of (M) by 1.	Indirect
<i>Logic Instructions</i>		
1. RLC	Rotate each bit in the accumulator to the left.	

2. RAL	Rotate each bit in the accumulator to the left through the Carry.	
3. RRC	Rotate each bit in the accumulator to the right.	
4. RAR	Rotate each bit in the accumulator to the right through the Carry.	
5. CMP R	Compare (R) with (A).	Register
6. CMP M	Compare (M) with (A).	Indirect
7. CPI 8-bit	Compare 8-bit data with (A).	Immediate

LOOKING AHEAD

The instructions that have been introduced in this and the previous chapter make up the major segment of the instruction set. Applications of these instructions in designing counters and time delays are illustrated in the next chapter. The other group of instructions critical for assembly language programming is related to the subroutine technique, illustrated in Chapter 9.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

If data bytes are given in the programming assignments, enter those data bytes manually in the respective memory locations before executing the programs. If an assignment calls for an output port and your microcomputer does not have an independent output port, store the results in memory. The high-order byte of a memory address is shown as XX. Substitute the high-order byte that is appropriate to the trainer in your laboratory.

Section 7.1

1. You are given a set of 100 resistors with 10 k value and asked to test them for 10 percent tolerance. Reject all the resistors that are outside the tolerance and give the final number of resistors you found within the tolerance. Draw a flowchart.
2. You are given a big grocery list for a party and asked to buy the items starting from number 60. The maximum amount you can spend is \$125. Draw a flowchart.
3. In the receiving department of a manufacturing company, auto parts are placed in a sequential order from 0000 to 4050. These parts need to be transferred to the bins on the factory floor, marked with the starting number 8000. You have an intelligent robot that can read the bin numbers and transfer the parts. Draw a flowchart to instruct the robot to transfer 277 parts starting from bin number 1025 in the receiving department to the starting bin number 8060 on the floor.

4. A newly hired librarian was given a set of new books on computers and English literature. The stack of computer books was placed above the English books. The librarian was asked to find the total amount spent on computer books; the prices were marked on the back cover of each book. Draw a flowchart representing the process.

Section 7.2

5. Specify the memory location and its contents after the following instructions are executed.

```
MVI B,F7H
MOV A,B
STA XX75H
HLT
```

6. Show the register contents as each of the following instructions is being executed.

A	B	C	D	E	H	L
---	---	---	---	---	---	---

```
MVI C,FFH
LXI H,XX70H
LXI D,XX70H
MOV M,C
LDAX D
HLT
```

7. In Question 6, specify the contents of the accumulator and the memory location XX70H after the execution of the instruction LDAX D.
 8. Identify the memory locations that are cleared by the following instructions.

```
MVI B,00H
LXI H,XX75H
MOV M,B
INX H
MOV M,B
HLT
```

9. Specify the contents of registers A, D, and HL after execution of the following instructions.

```
LXI H,XX90H ;Set up register HL as a memory pointer
SUB A      ;Clear accumulator
MVI D,0FH   ;Set up register D as a counter
LOOP: MOV M,A ;Clear memory
       INX H    ;Point to the next memory location
```

```
DCR D      ;Update counter  
JNZ LOOP   ;Repeat until (D) = 0  
HLT
```

10. Explain the result after the execution of the program in Question 9.
11. Rewrite the instructions in Question 9 using the register BC as a memory pointer.
12. Explain how many times the following loop will be executed.

```
LXI B,0007H  
LOOP: DCX B  
      JNZ LOOP
```

13. Explain how many times the following loop will be executed.

```
LXI B,0007H  
LOOP: DCX B  
      MOV A,B  
      ORA C  
      JNZ LOOP
```

14. The following instructions are intended to clear ten memory locations starting from the memory address 0009H. Explain why a large memory block will be erased or cleared and the program will stay in an infinite loop.

```
LXI H,0009H  
LOOP: MVI M,00H  
      DCX H  
      JNZ LOOP  
      HLT
```

15. The following block of data is stored in the memory locations from XX55H to XX5AH. Transfer the data to the locations XX80H to XX85H in the reverse order (e.g., the data byte 22H should be stored at XX85H and 37H at XX80H).
Data(H) 22, A5, B2, 99, 7F, 37
16. Data bytes are stored in memory locations from XX50H to XX5FH. To insert an additional five bytes of data, it is necessary to shift the data string by five memory locations. Write a program to store the data string from XX55H to XX64H. Use any sixteen bytes of data to verify your program.

Hint: This is a block transfer of data bytes with overlapping memory locations. If the data transfer begins at location XX50H, a segment of the data string will be destroyed.

17. A system is designed to monitor the temperature of a furnace. Temperature readings are recorded in 16 bits and stored in memory locations starting at XX60H. The high-order byte is stored first and the low-order byte is stored in the next consecutive memory location. However, the high-order byte of all the temperature readings is constant.

Write a program to transfer low-order readings to consecutive memory locations starting at XX80H and discard the high-order bytes.

Temperature Readings (H) 0581, 0595, 0578, 057A, 0598

18. A string of six data bytes is stored starting from memory location 2050H. The string includes some blanks (bytes with zero value). Write a program to eliminate the blanks from the string. (*Hint:* To check a blank, set the Zero flag by using the ORA. Use two memory pointers: one to get a byte and the other to store the byte.)
Data(H) F2, 00, 00, 4A, 98, 00
19. Write a program to add the following five data bytes stored in memory locations starting from XX60H, and display the sum. (The sum does not generate a carry. Use register pair DE as a memory pointer to transfer a byte from memory into a register.)
Data(H) 1A, 32, 4F, 12, 27
20. Write a program to add the following data bytes stored in memory locations starting at XX60H and display the sum at the output port if the sum does not generate a carry. If a result generates a carry, stop the addition, and display 01H at the output port.
Data(H) First Set: 37, A2, 14, 78, 97
Second Set: 12, 1B, 39, 42, 07
21. In Assignment 20, modify the program to count the number of data bytes that have been added and display the count at the second output port.

Section 7.3

22. Specify the contents of memory locations XX70H to XX74H after execution of the following instructions.

```

LXI H,XX70H      ;Set up HL as a memory pointer
MVI B,05H        ;Set up register B as a counter
MVI A,01
STORE: MOV M,A    ;Store (A) in memory
        INR A
        INX H
        DCR B
        JNZ STORE
        HLT
    
```

23. Identify the contents of the registers, the memory location (XX55H), and the flags as the following instructions are executed.

A	H	L	S	Z	CY	M
						XX55H

```

LXI H,XX55H
MVI M,8AH
MVI A,76H
ADD M
STA XX55H
HLT
    
```

24. Assemble and execute the instructions in Question 23, and verify the contents.
25. Identify the contents of the memory location XX65H and the status of the flags S, Z, and CY when the instruction INR M is executed.

```
LXI H,XX65H  
MVI M,FFH  
INR M  
HLT
```

26. Repeat the Illustrative Program: Addition with Carry (Section 7.3.2) using the DE register as a memory pointer and memory location XX40H as a counter. (*Hints:* Use the instruction MVI M to load the memory location XX40H with a count, and DCR M to decrement the counter.)
27. The temperatures of two furnaces are being monitored by a microcomputer. A set of five readings of the first furnace, recorded by five thermal sensors, is stored at the memory location starting at XX50H. A corresponding set of five readings from the second furnace is stored at the memory location starting at XX60H. Each reading from the first set is expected to be higher than the corresponding reading from the second set. For example, the temperature reading at the location 54H (T_{54}) is expected to be higher than the temperature reading at the location 64H (T_{64}).

Write a program to check whether each reading from the first set is higher than the corresponding reading from the second set. If all readings from the first set are higher than the corresponding readings from the second set, turn on the bit D_0 of the output PORT1. If any one of the readings of the first set is lower than the corresponding reading of the second set, stop the process and output FF as an emergency signal to the output PORT1.

Data(H) First Set: 82, 89, 78, 8A, 8F
Second Set: 71, 78, 79, 82, 7F

28. Repeat Assignment 27 with the following modification. Check whether any two readings are equal, and if they are equal, turn on bit D_7 of PORT1 and continue checking. (*Hint:* Check for the Zero flag when two readings are equal.)

Data(H) First Set: 80, 85, 8F, 82, 87
Second Set: 71, 74, 7A, 82, 77

29. A set of eight data bytes is stored in memory locations starting from XX70H. Write a program to add two bytes at a time and store the sum in the same memory locations, low-order sum replacing the first byte and a carry replacing the second byte. If any pair does not generate a carry, the memory location of the second byte should be cleared.

Data(H) F9, 38, A7, 56, 98, 52, 8F, F2

30. A set of eight data bytes is stored in memory locations starting from XX70H. Write a program to subtract two bytes at a time and store the result in a sequential order in memory locations starting from XX70H.

Data(H) F9, 38, A7, 56, 98, A2, F4, 67

31. In Assignment 30, if any one of the results of the subtraction is in the 2's complement, it should be discarded.

Section 7.4

32. Specify the contents of the accumulator and the status of the CY flag when the following instructions are executed.

a. MVI A,B7H b. MVI A,B7H
 ORA A ORA A
 RLC RAL

33. Identify the register contents and the flags as the following instructions are being executed.

A S Z CY

MVI A,80H

ORA A

RAR

34. Specify the contents of the accumulator and the CY flag when the following instructions are executed.

a. MVI A,C5H b. MVI A,A7H
 ORA A ORA A
 RAL RAR
 RRC RAL

35. The accumulator in the following set of instructions contains a BCD number. Explain the function of these instructions.

MVI A,79H
ANI F0H
RRC
RRC
RRC
RRC

36. Calculate the decimal value of the number in the accumulator before and after the Rotate instructions are executed, and explain the mathematical functions performed by the instructions.

a. MVI A,18H b. MVI A,78H
 RLC RRC
 RRC

37. Explain the mathematical function that is performed by the following instructions.

MVI A,07H
RLC
MOV B,A
RLC
RLC
ADD B

38. Repeat the Illustrative Program in Section 7.4.2 ("Checking Sign with Rotate Instructions") with the following modifications:
- Check the sign of a number by using the instruction JM (Jump On Minus), instead of the instruction RAL. (*Hint:* Set the flags by using the instruction ORA A.)
 - If the sum of the positive readings exceeds eight bits, continue the addition, save generated carry, and display the total sum at two different ports.
- Data(H)** 48, 72, 8F, 7F, 6B, F2, 98, 7C, 67, 19
39. In Assignment 38, in addition to modifications a and b, count the number of positive readings in the set and display the count at PORT3.
40. A set of eight data bytes is stored in the memory location starting at XX50H. Check each data byte for bits D₇ and D₀. If D₇ or D₀ is 1, reject the data byte; otherwise, store the data bytes at memory locations starting at XX60H.
- Data(H)** 80, 52, E8, 78, F2, 67, 35, 62

Section 7.5

41. Identify the contents of the accumulator and the flag status as the following instructions are executed.

A	S	Z	CY
---	---	---	----

MVI A,7FH
 ORA A
 CPI A2H

42. Identify the register contents and the flag status as the following instructions are executed.

A	S	Z	CY
---	---	---	----

LXI H,2070H
 MVI M,64H
 MVI A,8FH
 CMP M

43. Identify the bytes from the following set that will be displayed at PORT1, assuming one byte is loaded into the accumulator at a time.

Data(H) 58, 32, 7A, 87, F2, D7

MVI A,BYTE
MVI B,64H
MVI C,C8H
CMP B
JC REJECT
CMP C

```
JNC REJECT
OUT PORT1
HLT
REJECT: SUB A
OUT PORT1
HLT
```

44. In Question 43, identify the range of numbers in decimal that will be displayed at PORT1.
45. The following program reads one data byte at a time. Identify the data bytes from the following set that will transfer the program to location ACCEPT.
Data(H) 19, 20, 64, 8F, D8, F2

```
IN PORT1
MVI B,20H
CMP B
JC REJECT
JM REJECT
STA 2070H
JMP ACCEPT
REJECT: JMP INVALID
```

46. In Question 45, identify the range of numbers in decimal that will transfer the program to location INVALID.
47. Repeat the Illustrative Program: Use of Compare Instruction to Indicate the End of a Data String (Section 7.5.2), but include the following modifications: Clear register D and use CMP D to check a byte in the memory location. If a byte is not zero, add the byte and continue adding; otherwise, go to the output.
48. A set of eight readings is stored in memory starting at location XX50H. Write a program to check whether a byte 40H exists in the set. If it does, stop checking and display its memory location; otherwise, output FFH.
Data(H) 48, 32, F2, 38, 37, 40, 82, 8A
49. Refer to Assignment 48. Write a program to find the highest reading in the set, and display the reading at an output port.
50. Refer to Assignment 48. Write a program to find the lowest reading in the set, and display the reading at the output port.
51. A set of ten bytes is stored in memory starting with the address XX50H. Write a program to check each byte, and save the bytes that are higher than 60_{10} and lower than 100_{10} in memory locations starting from XX60H.
Data(H) 6F, 28, 5A, 49, C7, 3F, 37, 4B, 78, 64
52. In Assignment 51, in addition to saving the bytes in the given range, display at PORT1 the number of bytes saved.

53. A string of readings is stored in memory locations starting at XX70H, and the end of the string is indicated by the byte 0DH. Write a program to check each byte in the string, and save the bytes in the range of 30H to 39H (both inclusive) in memory locations starting from XX90H.

Data(H) 35, 2F, 30, 39, 3A, 37, 7F, 31, 0D, 32

54. In Assignment 53, display the number of bytes accepted from the string between 30H and 39H.

55. A bar code scanner scans the boxes being shipped from the loading dock and records all the codes in computer memory; the end of the data is indicated by the byte 00. The code 1010 0011 (A3H) is assigned to 19" television sets. Write a program to count the number of 19" television sets that were shipped from the following data set.

Data(H) FA, 67, A3, B8, A3, A3, FA, 00

56. Sort the following set of marks scored by ten students in a circuit course in descending order.

Data(H) 63, 41, 56, 62, 48, 5A, 4F, 4C, 56, 56

Section 7.6

57. The following program adds the number of bytes stored in memory locations starting from XX00H and saves the result in memory. Read the program and answer the questions given below.

LXI H,XX00H	;Set up HL as a data pointer
LXI D,0000H	;Set up D as a byte counter
	; and E as a Carry register
NEXT: ADD M	;Add byte
JNC SKIP	;If the result has no carry, do not
	; increment Carry register
INR E	
SKIP: DCR D	;Update byte counter
JNZ NEXT	;Go to next byte
LXI H,XX90H	
MOV M,A	;Save the result
INX H	
MOV M,E	
HLT	

- Assuming the byte counter is set up appropriately, specify the number of bytes that are added by the program.
 - Specify the memory locations where the result is stored.
 - Identify the two errors in the program.
58. The following program checks a set of six signed numbers and adds the positive numbers. The numbers are stored in memory locations starting from

XX60H. The final result is expected to be less than FFH and stored in location XX70H.

Data(H) First Set: 20, 87, F2, 37, 79, 17
 Second Set: A2, 15, 3F, B7, 47, 9A

MVI B,00H	;Clear (B) to save result
MVI C,06H	;Set up register C to count six numbers
LXI H,XX60H	;Set up HL as a memory pointer
NEXT: MOV A,M	;Get a byte
RAL	;Place D ₇ into CY flag
JC REJECT	;If D ₇ = 1, reject the byte
RAR	;Restore the byte
ADD B	;Add the previous sum
MOV B,A	;Save the sum
REJECT: INX H	;Next location
DCR B	;Update the byte counter
JNZ NEXT	;Go back to get the next byte
STA XX70H	;Save the result
HLT	

- a. Calculate the answers for the given data sets.
 - b. Assemble and execute the program for the first data set, and verify the answer.
 - c. Execute the program for the second data set, and verify the answer.
 - d. Debug the program using the Single-Step and Examine Register techniques if the result is different from the expected answer.
59. The following program adds five bytes stored in memory locations starting from 2055H. The result is stored in locations 2060H and 2061H.

Data(H) First Set: 17, 1B, 21, 7F, 9D
 Second Set: F2, 87, A9, B8, C2

2000	21 55 20	LXI H,2055H	;Set HL as a memory pointer
2003	AF	XRA A	;Clear accumulator
2004	01 05 00	LXI B,0005H	;Clear B to save carries and set up C as ; a byte counter
2007	8E	ADC M	;Add byte
2008	D2 0C 20	JNC 200CH	;Skip CY register if no carry
200B	04	INR B	;Update CY register
200C	0D	DCR C	;Update counter
200D	23	INX H	;Next memory location
200E	C2 20 08	JNZ 2008H	;Go back to add next byte
2011	32 60 20	STA 2060H	;Store low-order sum
2014	78	MOV A,B	;Get high-order sum
2015	32 61 20	STA 2061	;Store high-order sum
2018		HLT	

- a. Calculate the answers for the given data sets.
- b. Enter the program and the first data set. Execute the program and verify the answer. Debug the program if necessary.
- c. Execute the program for the second data set, and verify the answer.
- d. Debug the program using the Single-Step and Examine Register techniques if the result is different from the expected answer.

8

Counters and Time Delays

This chapter deals with the designing of counters and timing delays through software (programming). Two of the programming techniques discussed in the last chapter—looping and counting—are used to design counters and time delays. The necessary instructions have already been introduced in the previous two chapters.

A counter is designed by loading an appropriate count in a register. A loop is set up to decrement the count for a down-counter* or to increment the count for an up-counter.[†] Similarly, a timing delay is designed by loading a register with a delay count

and setting up a loop to decrement the count until zero. The delay is determined by the clock period of the system and the time required to execute the instructions in the loop.

Counters and time delays are important techniques. They are commonly used in applications such as traffic signals, digital clocks, process control, and serial data transfer.

OBJECTIVES

- Write instructions to set up time delays, using one register, a register pair, and a loop-within-a-loop technique.
- Calculate the time delay in a given loop.
- Draw a flowchart for a counter with a delay.
- Design an up/down counter for a given delay.
- Write a program to turn on/off specific bits at a given interval.

*A down-counter counts in descending order.

[†]An up-counter counts in ascending order.

8.1

COUNTERS AND TIME DELAYS

Designing a counter is a frequent programming application. Counters are used primarily to keep track of events; time delays are important in setting up reasonably accurate timing between two events. The process of designing counters and time delays using software instructions is far more flexible and less time consuming than the design process using hardware.

COUNTER

A counter is designed simply by loading an appropriate number into one of the registers and using the INR (Increment by One) or the DCR (Decrement by One) instructions. A loop is established to update the count, and each count is checked to determine whether it has reached the final number; if not, the loop is repeated.

The flowchart shown in Figure 8.1 illustrates these steps. However, this counter has one major drawback; the counting is performed at such high speed that only the last count can be observed. To observe counting, there must be an appropriate time delay between counts.

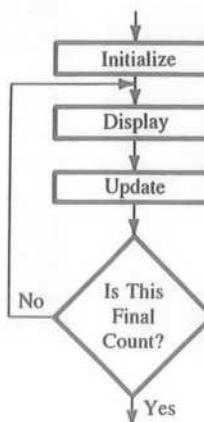
TIME DELAY

The procedure used to design a specific delay is similar to that used to set up a counter. A register is loaded with a number, depending on the time delay required, and then the register is decremented until it reaches zero by setting up a loop with a conditional Jump instruction. The loop causes the delay, depending upon the clock period of the system, as illustrated in the next sections.

8.1.1 Time Delay Using One Register

The flowchart in Figure 8.2 shows a time-delay loop. A count is loaded in a register, and the loop is executed until the count reaches zero. The set of instructions necessary to set up the loop is also shown in Figure 8.2.

FIGURE 8.1
Flowchart of a Counter



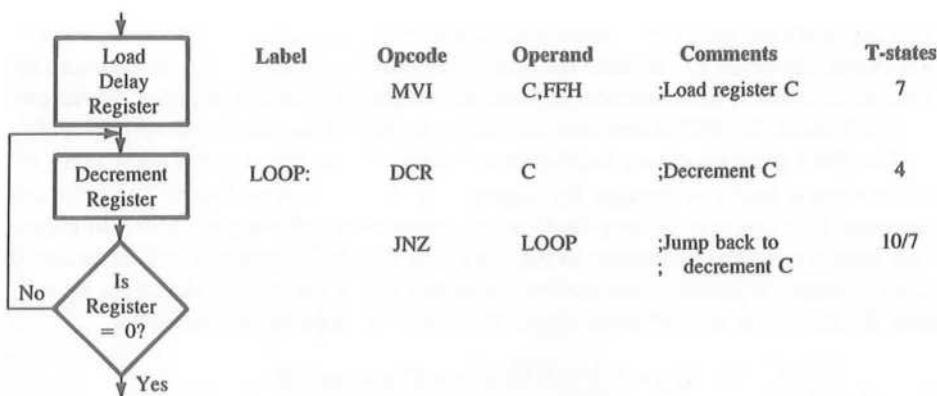


FIGURE 8.2
Time Delay Loop: Flowchart and Instructions

The last column in Figure 8.2 shows the T-states (clock periods) required by the 8085 microprocessor to execute each instruction. (See Appendix F for the list of the 8085 instructions and their T-states.) The instruction MVI requires seven clock periods. An 8085-based microcomputer with 2 MHz clock frequency will execute the instruction MVI in 3.5 μ s as follows:

$$\begin{aligned}
 &\text{Clock frequency of the system } f = 2 \text{ MHz} \\
 &\text{Clock period } T = 1/f = 1/2 \times 10^{-6} = 0.5 \mu\text{s} \\
 &\text{Time to execute MVI} = 7 \text{ T-states} \times 0.5 \\
 &\qquad\qquad\qquad = 3.5 \mu\text{s}
 \end{aligned}$$

However, if the clock frequency of the system is 1 MHz, the microprocessor will require 7 μ s to execute the same instruction. To calculate the time delay in a loop, we must account for the T-states required for each instruction and for the number of times the instructions are executed in the loop.

In Figure 8.2, register C is loaded with the count FFH (255_{10}) by the instruction MVI, which is executed once and takes seven T-states. The next two instructions, DCR and JNZ, form a loop with a total of 14 (4 + 10) T-states. The loop is repeated 255 times until register C = 0.

The time delay in the loop T_L with 2 MHz clock frequency is calculated as

$$T_L = (T \times \text{Loop T-states} \times N)$$

where T_L = Time delay in the loop

T = System clock period

N = Equivalent decimal number of the hexadecimal count loaded in the delay register

$$\begin{aligned}
 T_L &= (0.5 \times 10^{-6} \times 14 \times 255) \\
 &= 1785 \mu\text{s} \\
 &\approx 1.8 \text{ ms}
 \end{aligned}$$



In most applications, this approximate calculation of the time delay is considered reasonably accurate. However, to calculate the time delay more accurately, we need to adjust for the execution of the JNZ instruction and add the execution time of the initial instruction.

The T-states for JNZ instruction are shown as 10/7. This can be interpreted as follows: The 8085 microprocessor requires ten T-states to execute a conditional Jump instruction when it jumps or changes the sequence of the program and seven T-states when the program falls through the loop (goes to the instruction following the JNZ). In Figure 8.2, the loop is executed 255 times; in the last cycle, the JNZ instruction will be executed in seven T-states. This difference can be accounted for in the delay calculation by subtracting the execution time of three states. Therefore, the adjusted loop delay is

$$\begin{aligned}T_{LA} &= T_L - (3 \text{ T-states} \times \text{Clock period}) \\&= 1785.0 \mu\text{s} - 1.5 \mu\text{s} = 1783.5 \mu\text{s}\end{aligned}$$

Now the total delay must take into account the execution time of the instructions outside the loop. In the above example, we have only one instruction (MVI C) outside the loop. Therefore, the total delay is

$$\text{Total Delay} = \frac{\text{Time to execute instructions}}{\text{outside loop}} + \frac{\text{Time to execute}}{\text{loop instructions}}$$

$$\begin{aligned}T_D &= T_O + T_{LA} \\&= (7 \times 0.5 \mu\text{s}) + 1783.5 \mu\text{s} = 1787 \mu\text{s} \\&\approx 1.8 \text{ ms}\end{aligned}$$

The difference between the loop delay T_L and these calculations is only 2 μs and can be ignored in most instances.

The time delay can be varied by changing the count FFH; however, to increase the time delay beyond 1.8 ms in a 2 MHz microcomputer system, a register pair or a loop within a loop technique should be used.

8.1.2 Time Delay Using a Register Pair

The time delay can be considerably increased by setting a loop and using a register pair with a 16-bit number (maximum FFFFH). The 16-bit number is decremented by using the instruction DCX. However, the instruction DCX does not set the Zero flag and, without the test flags, Jump instructions cannot check desired data conditions. Additional techniques, therefore, must be used to set the Zero flag.

The following set of instructions uses a register pair to set up a time delay.

Label	Opcode	Operand	Comments	T-states
LOOP:	LXI	B,2384H	;Load BC with 16-bit count	10
	DCX	B	;Decrement (BC) by one	6
	MOV	A,C	;Place contents of C in A	4
	ORA	B	;OR (B) with (C) to set Zero flag	4
	JNZ	LOOP	;If result ≠ 0, jump back to LOOP	10/7

In this set of instructions, the instruction LXI B,2384H loads register B with the number 23H, and register C with the number 84H. The instruction DCX decrements the entire number by one (e.g., 2384H becomes 2383H). The next two instructions are used only to set the Zero flag; otherwise, they have no function in this problem. The OR instruction sets the Zero flag only when the contents of B and C are simultaneously zero. Therefore, the loop is repeated 2384H times, equal to the count set in the register pair.

TIME DELAY

The time delay in the loop is calculated as in the previous example. The loop includes four instructions: DCX, MOV, ORA, and JNZ, and takes 24 clock periods for execution. The loop is repeated 2384H times, which is converted to decimals as

$$\begin{aligned} 2384H &= 2 \times (16)^3 + 3 \times (16)^2 + 8 \times (16)^1 + 4(16)^0 \\ &= 9092_{10} \end{aligned}$$

If the clock period of the system = 0.5 μ s, the delay in the loop T_L is

$$\begin{aligned} T_L &= (0.5 \times 24 \times 9092_{10}) \\ &\approx 109 \text{ ms (without adjusting for the last cycle)} \end{aligned}$$

$$\begin{aligned} \text{Total Delay } T_D &= 109 \text{ ms} + T_O \\ &\approx 109 \text{ ms (The instruction LXI adds only } 5 \mu\text{s.)} \end{aligned}$$

A similar time delay can be achieved by using the technique of two loops, as discussed in the next section.

8.1.3 Time Delay Using a Loop within a Loop Technique

A time delay similar to that of a register pair can also be achieved by using two loops; one loop inside the other loop, as shown in Figure 8.3(a). For example, register C is used in the inner loop (LOOP1) and register B is used for the outer loop (LOOP2). The following instructions can be used to implement the flowchart shown in Figure 8.3(a).

MVI B,38H	7T
LOOP2: MVI C,FFH	7T
LOOP1: DCR C	4T
JNZ LOOP1	10/7T
DCR B	4T
JNZ LOOP2	10/7T

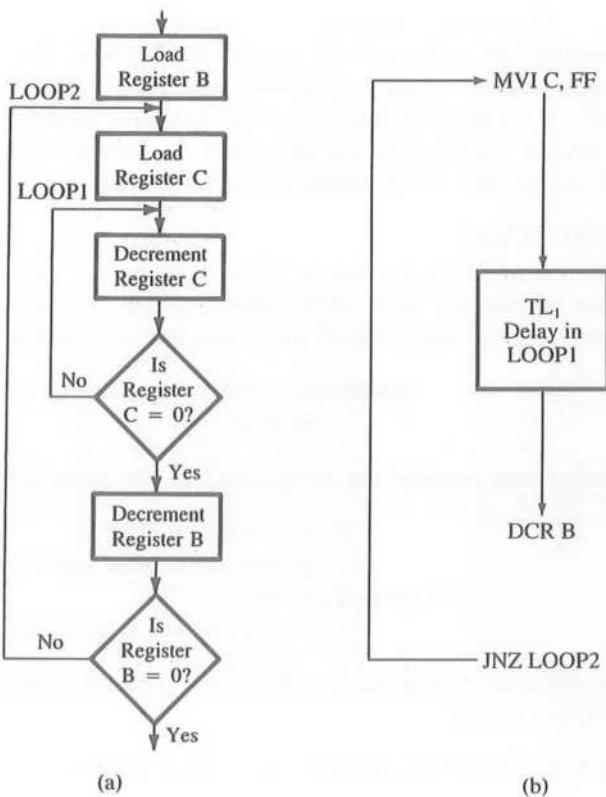
DELAY CALCULATIONS

The delay in LOOP1 is $T_{L1} = 1783.5 \mu$ s. These calculations are shown in Section 8.1.1. We can replace LOOP1 by T_{L1} , as shown in Figure 8.3(b). Now we can calculate the delay in LOOP2 as if it is one loop; this loop is executed 56 times because of the count (38H) in register B:

$$\begin{aligned} T_{L2} &= 56(T_{L1} + 21 \text{ T-states} \times 0.5 \mu\text{s}) \\ &= 56(1783.5 \mu\text{s} + 10.5 \mu\text{s}) \\ &= 100.46 \text{ ms} \end{aligned}$$

FIGURE 8.3

Flowchart for Time Delay with Two Loops



The total delay should include the execution time of the first instruction (MVI B,7T); however, the delay outside these loops is insignificant. The time delay can be increased considerably by using register pairs in the above example.

Similarly, the time delay within a loop can be increased by using instructions that will not affect the program except to increase the delay. For example, the instruction NOP (No Operation) can add four T-states in the delay loop. The desired time delay can be obtained by using any or all available registers.

8.1.4 Additional Techniques for Time Delay

The disadvantages in using software delay techniques for real-time applications in which the demand for time accuracy is high, such as digital clocks, are as follows:

1. The accuracy of the time delay depends on the accuracy of the system's clock.
2. The microprocessor is occupied simply in a waiting loop; otherwise it could be employed to perform other functions.
3. The task of calculating accurate time delays is tedious.

In real-time applications, timers (integrated timer circuits) are commonly used. The Intel 8254 (described in Chapter 15) is a programmable timer chip that can be interfaced with the microprocessor and programmed to provide timings with considerable accuracy. The disadvantages of using the hardware chip include the additional expense and the need for an extra chip in the system.

8.1.5 Counter Design with Time Delay

To design a counter with a time delay, the techniques illustrated in Figures 8.1 and 8.2 can be combined. The combined flowchart is shown in Figure 8.4.

The blocks shown in the flowchart are similar to those in the generalized flowchart in Figure 7.3. The block numbers shown in Figure 8.4 correspond to the block numbers in the generalized flowchart. Compare Figure 8.4 with Figure 7.3 and note the following points about the counter flowchart (Figure 8.4):

1. The output (or display) block is part of the counting loop.
2. The data processing block is replaced by the time-delay block.
3. The save-the-partial-answer block is eliminated because the count is saved in a counter register, and a register can be incremented or decremented without transferring the count to the accumulator.

The flowchart in Figure 8.4 shows the basic building blocks. However, the sequence can be changed, depending upon the nature of the problem, as shown in Figures 8.5(a) and 8.5(b).

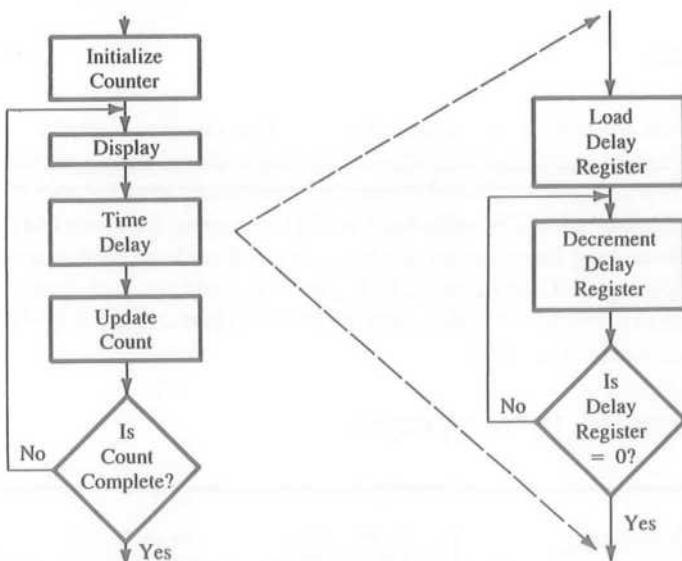


FIGURE 8.4
Flowchart of a Counter with a Time Delay

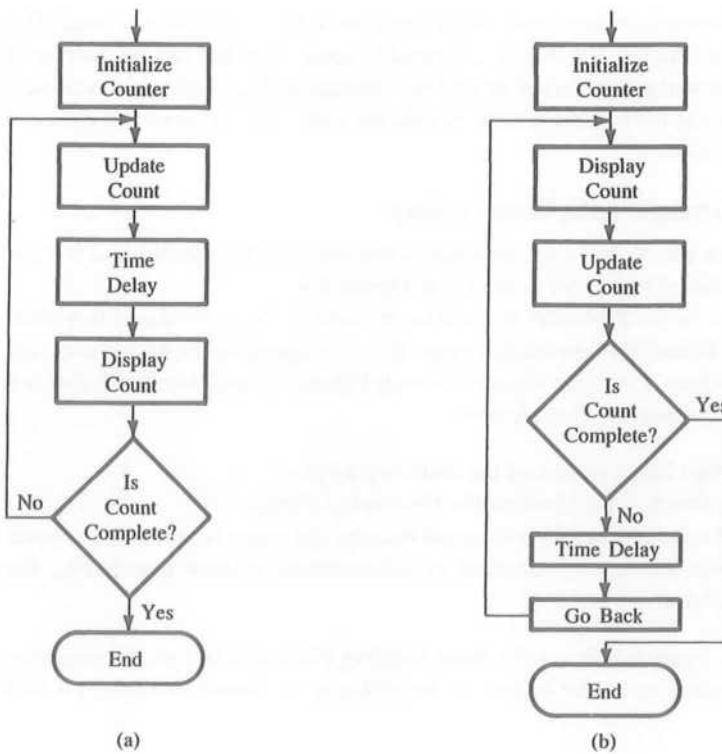


FIGURE 8.5
Variations of Counter Flowchart

The flowchart in Figure 8.4 displays the count after initialization. For example, an up-counter starting at zero can be initialized at 00H using the logic shown in Figure 8.4. However, the flowchart in Figure 8.5(a) updates the counter immediately after the initialization. In this case, an up-counter should be initialized at FFH to display the count 00H.

Similarly, the decision-making block differs slightly in these flowcharts. For example, if a counter was counting up to 9, the question in Figure 8.4 would be: Is the count 10? In Figure 8.5(a) the question would be: Is the count 9? The flowchart in Figure 8.5(b) illustrates another way of designing a counter.

8.2

ILLUSTRATIVE PROGRAM: HEXADECIMAL COUNTER

PROBLEM STATEMENT

Write a program to count continuously in hexadecimal from FFH to 00H in a system with a 0.5 μ s clock period. Use register C to set up a one millisecond (ms) delay between each count and display the numbers at one of the output ports.

PROBLEM ANALYSIS

The problem has two parts; the first is to set up a continuous down-counter, and the second is to design a given delay between two counts.

1. The hexadecimal counter is set up by loading a register with an appropriate starting number and decrementing it until it becomes zero (shown by the outer loop in the flowchart, Figure 8.6). After zero count, the register goes back to FF because decrementing zero results in a (-1), which is FF in 2's complement.
2. The one millisecond (ms) delay between each count is set up by using the procedure explained previously in Section 8.1.1—Time Delay Using One Register. Figure 8.2 is identical with the inner loop of the flowchart shown in Figure 8.6. The delay calculations are shown later.

FLOWCHART AND PROGRAM

The flowchart in Figure 8.6 shows the two loops discussed earlier—one for the counter and another for the delay. The counter is initialized in the first block, and the count is displayed

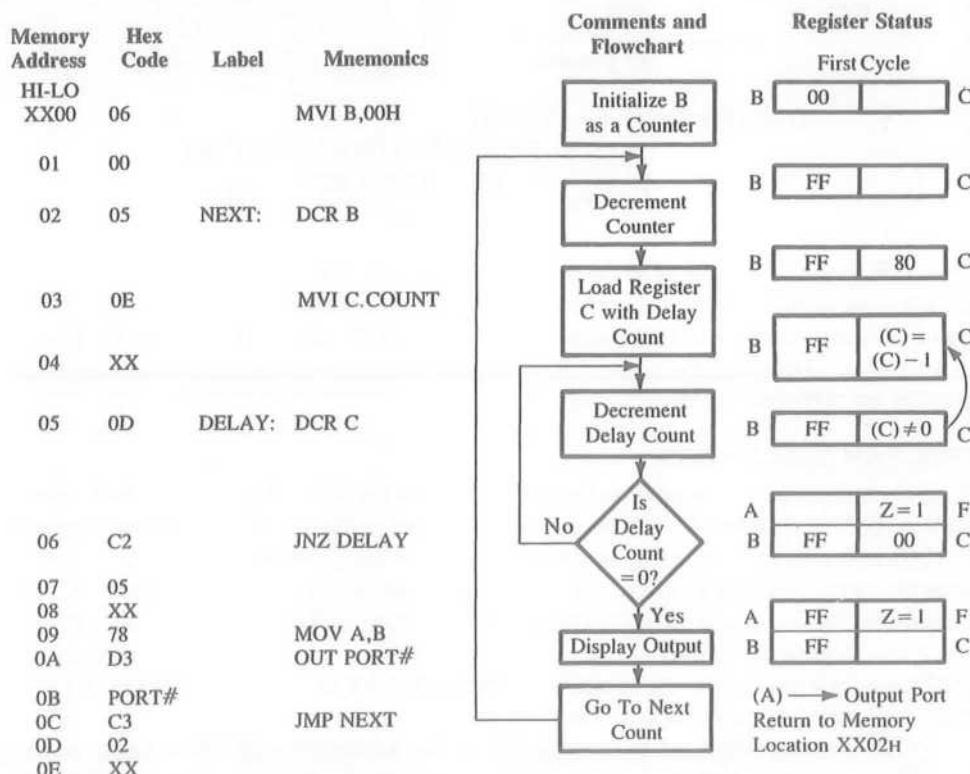


FIGURE 8.6

Program and Flowchart for a Hexadecimal Counter

in the outer loop. The delay is accomplished in the inner loop. This flowchart in Figure 8.6 is similar to that in Figure 8.5(a). You should study the flowchart carefully to differentiate between the counter loop and the delay loop because they may at first appear to be similar.

Delay Calculations The delay loop includes two instructions: DCR C and JNZ with 14 T-states. Therefore, the time delay T_L in the loop (without accounting for the fact that JNZ requires seven T-states in the last cycle) is

$$\begin{aligned}T_L &= 14 \text{ T-states} \times T \text{ (Clock period)} \times \text{Count} \\&= 14 \times (0.5 \times 10^{-6}) \times \text{Count} \\&= (7.0 \times 10^{-6}) \times \text{Count}\end{aligned}$$

The delay outside the loop includes the following instructions:

DCR B	4T	Delay outside
MVI C,COUNT	7T	the loop: $T_O = 35 \text{ T-states} \times T$
MOV A,B	4T	$= 35 \times (0.5 \times 10^{-6})$
OUT PORT	10T	$= 17.5 \mu\text{s}$
JMP	10T	
35 T-states		

$$\begin{aligned}\text{Total Time Delay } T_D &= T_O + T_L \\1 \text{ ms} &= 17.5 \times 10^{-6} + (7.0 \times 10^{-6}) \times \text{Count} \\ \text{Count} &= \frac{1 \times 10^{-3} - 17.5 \times 10^{-6}}{7.0 \times 10^{-6}} \approx 140_{10}\end{aligned}$$

Therefore, the delay count 8CH (140_{10}) must be loaded in register C to obtain 1 ms delay between each count. In this problem, the calculation of the delay count does not take into consideration that the JNZ instruction takes seven T-states in the last cycle, instead of 10 T-states. However, the count will remain the same even if the calculations take into account the difference of three T-states.

PROGRAM DESCRIPTION

Register B is used as a counter, and register C is used for delay. Register B initially starts with number 00H; when it is decremented by the instruction DCR, the number becomes FFH. (Verify this by subtracting one from zero in 2's complement.) Register C is loaded with the delay count 8CH to provide a 1 ms delay in the loop. The instruction DCR C decrements the count and the instruction JNZ (Jump On No Zero) checks the Zero flag to see if the number in register C has reached zero. If the number is not zero, instruction JNZ causes a jump back to the instruction labeled **DELAY** in order to decrement (C) and, thus, the loop is repeated 140_{10} times.

The count is displayed by moving (B) to the accumulator and then to the output port. The instruction JMP causes an unconditional jump for the next count in register B, forming a continuous loop to count from FFH to 00H. After the count reaches zero, (B) is decremented, becoming FFH, and the counting cycle is repeated.

PROGRAM OUTPUT

When the program is executed, the actual output seen may vary according to the device used as the output for the display. The eye cannot see the changes in a count with a 1 ms delay. If the output port has eight LEDs, the LEDs representing the low-order bits will appear to be on continuously, and the LEDs with high-order bits will go on and off according to the count. If the output port is a seven-segment display, all segments will appear to be on; a slight flicker in the display can be noticed. However, the count and the delay can be measured on an oscilloscope.

ILLUSTRATIVE PROGRAM: ZERO-TO-NINE (MODULO TEN)* COUNTER

8.3

PROBLEM STATEMENT

Write a program to count from 0 to 9 with a one-second delay between each count. At the count of 9, the counter should reset itself to 0 and repeat the sequence continuously. Use register pair HL to set up the delay, and display each count at one of the output ports. Assume the clock frequency of the microcomputer is 1 MHz.

Instructions Review the following instructions.

- LXI: Load Register Pair Immediate
- DCX: Decrement Register Pair
- INX: Increment Register Pair

These instructions manipulate 16-bit data by using registers in pairs (BC, DE, and HL). However, the instructions DCX and INX do not affect flags to reflect the outcome of the operation. Therefore, additional instructions must be used to set the flags.

PROBLEM ANALYSIS

The problem is similar to that in the Illustrative Program for a hexadecimal counter (Section 8.2) except in two respects: the counter is an up-counter (counts *up* to the digit 9), and the delay is too long to use just one register.

1. The counter is set up by loading a register with the appropriate number and incrementing it until it reaches the digit 9. When the counter register reaches the final count, it is reset to zero. This is an additional step compared to the Illustrative Program of Section 8.2 in which the counter resets itself. (Refer to the flowchart in Figure 8.7 to see how each count is checked against the final count.)

*The counter goes through ten different states (0 to 9) and is called a *modulo ten* counter.

2. The 1-second delay between each count is set up by using a register pair, as explained in Section 8.1.2. The delay calculations are shown later.

FLOWCHART AND PROGRAM

The flowchart in Figure 8.7 shows three loops: one for the counter (outer) loop on the left, the second for the delay (inner) loop on the left, and the third to reset the counter (the loop on the right). This flowchart is similar to the flowchart in Figure 8.4. The flowchart indicates that the number is displayed immediately after the initialization; this is different from the flowchart of Figure 8.6, in which the number is displayed at the end of the program.

Memory Address	Hex Code	Label	Mnemonics	T	Comments and Flowchart
HI-LO XX00					
01 06		START:	MVI B,00H		
02 00					
03 D3		DISPLAY:	OUT PORT#	10 } T ₀	
04 PORT#					
05 21			LXI H,16-Bit	10 }	
06 LO*					
07 HI					
08 2B		LOOP:	DCX H	6 }	
09 7D			MOV A,L	4 }	
0A B4			ORA H	4 }	
				T _L : 24 T-states	
0B C2			JNZ LOOP	10/7 }	
0C 08					
0D XX†					
0E 04			INR B	4 }	
0F 78			MOV A,B	4 }	
10 FE			CPI 0AH	7 }	
11 0A				T ₀	
12 C2			JNZ DISPLAY	10/7 }	
13 03					
14 XX†					
15 CA			JZ START		
16 00					
17 XX					
					;End of the count, start again

*Enter 16-bit delay count in place of LO and HI, appropriate to the clock period in your system.

†Enter high-order address (page number) of your R/W memory.

FIGURE 8.7

Program and Flowchart for a Zero-to-Nine Counter

The counter is incremented at the end of the program and checked against count 0AH (final count + 1). If the counter has not reached number 0AH, the count is displayed (outer loop on the left); otherwise, it is reset by the loop on the right (Figure 8.7).

PROGRAM DESCRIPTION

Register B is used as a counter, and register pair HL is used for the delay. The significant differences between this program and the Illustrative Program for a hexadecimal counter (Section 8.2) are as follows:

1. Register pair HL contains a 16-bit number that can be manipulated in two ways: first, as a 16-bit number, and second, as two 8-bit numbers. The instruction DCX views the HL register as one register with a 16-bit number. On the other hand, the instructions MOV A,L and ORA H treat the contents of the HL registers as two separate 8-bit numbers.
2. In the delay loop, the sequence is repeated by the instruction JNZ (Jump on No Zero) until the count becomes zero, thus providing a delay of 1 second. However, the instruction DCX does not set the Zero flag. Therefore, the instruction JNZ would be unable to recognize when the count has reached zero, and the program would remain in a continuous loop.

In this program, the instruction ORA is used to set the Zero flag. The purpose here is to check when the 16-bit number in register pair HL has reached zero. This is accomplished by ORing the contents of register L with the contents of register H. However, the contents of two registers cannot be ORed directly. The instruction MOV A,L loads the accumulator with (L), which is then ORed with (H) by the instruction ORA H. The Zero flag is set only when both registers are zero.

3. The instruction CPI,0AH (Compare Immediate with 0AH) checks the contents of the counter (register B) in every cycle. When register B reaches the number 0AH, the program sequence is redirected to reset the counter without displaying the number 0AH.
4. The time required to reset the counter to zero, indicated by the right-hand loop in the flowchart, is slightly different from the time delay between each count set by the left-hand outer loop.

Delay Calculations The major delay between two counts is provided by the 16-bit number in the delay register HL (the inner loop in the flowchart). This delay is set up by using a register pair, as explained in Section 8.1.2.

$$\begin{aligned} \text{Loop Delay } T_L &= 24 \text{ T-states} \times T \times \text{Count} \\ 1 \text{ second} &= 24 \times 1.0 \times 10^{-6} \times \text{Count} \\ \text{Count} &= \frac{1}{24 \times 10^{-6}} = 41666 = \text{A2C2H} \end{aligned}$$

The delay count A2C2H in register HL would provide approximately a 1-second delay between two counts. To achieve higher accuracy in the delay, the instructions outside the loop starting from OUT PORT# to JNZ DISPLAY and the difference of three states in the last execution of JNZ must be accounted for in the delay calculations.

The instructions outside the loop are: OUT, LXI, INR, MOV, CPI, and JNZ (DSPLAY). These instructions require 45 T-states; therefore, the delay count is calculated as follows:

$$\begin{aligned}\text{Total Delay } T_D &= T_O + T_L \\ 1 \text{ second} &= (45 \times 1.0 \times 10^{-6}) + (24 \times 1.0 \times 10^{+6} \times \text{Count}) \\ \text{Count} &\approx 41665\end{aligned}$$

The difference between the two delay counts calculated above is of very little significance in many applications.

PROGRAM OUTPUT

In an LED output port, each LED will be lit according to the binary count representing 0 to 9. In a seven-segment display, the continuous sequence of the numbers from 0 to 9 can be observed very easily because of the 1-second delay between each count. However, it will be difficult to observe a steady pattern on the oscilloscope (except on a storage scope) because the reset cycle time is slightly different from the delay time. (See Assignment 3 at the end of this chapter to run this program on your system.)

8.4

ILLUSTRATIVE PROGRAM: GENERATING PULSE WAVEFORMS

PROBLEM STATEMENT

Write a program to generate a continuous square wave with the period of 500 μ s. Assume the system clock period is 325 ns, and use bit D₀ to output the square wave.

Instructions The following instructions should be reviewed and the differences between various rotate instructions observed:

- RAL: Rotate Accumulator Left Through Carry
- RAR: Rotate Accumulator Right Through Carry
- RLC: Rotate Accumulator Left
- RRC: Rotate Accumulator Right

The first two instructions, RAL and RAR, use the Carry flag as the ninth bit, and the accumulator can be viewed as a 9-bit register. The last two instructions, RLC and RRC, rotate the accumulator contents through eight positions.

In addition to using these instructions, you should review the concept of masking with the ANI instruction.

PROBLEM ANALYSIS

In this problem, the period of the square wave is 500 μ s; therefore, the pulse should be on (logic 1) for 250 μ s and off (logic 0) for the remaining 250 μ s. The alternate pattern of 0/1 bits can be provided by loading the accumulator with the number AAH (1010 1010)

and rotating the pattern once through each delay loop. Bit D₀ of the output port is used to provide logic 0 and 1; therefore, all other bits can be masked by ANDing the accumulator with the byte 01H. The delay of 250 µs can be very easily obtained with an 8-bit delay count and one register.

PROGRAM

Memory Address	HEX HI-LO Code	Label	Mnemonics	Comments
XX00	16			
01	AA		MVI D,AA	;Load bit pattern AAH
02	7A	ROTATE:	MOV A,D	;Load bit pattern in A
03	07		RLC	;Change data from AAH to ; 55H and vice versa
04	57		MOV D,A	;Save (A)
05	E6		ANI 01H	;Mask bits D ₇ -D ₁
06	01			
07	D3		OUT PORT1	;Turn on or off the lights
08	PORT1			
09	06		MVI B,COUNT (7T)	;Load delay count for 250 µs
0A	COUNT			
0B	05	DELAY:	DCR B (4T)	;Next count
0C	C2		JNZ DELAY (10/7T)	;Repeat until (B) = 0
0D	0B			
0E	XX			
0F	C3		JMP ROTATE (10T)	;Go back to change logic level
10	02			
11	XX			

PROGRAM DESCRIPTION

Register D is loaded with the bit pattern AAH (1010 1010), and the bit pattern is moved into the accumulator. The bit pattern is rotated left once and saved again in register D. The accumulator contents must be saved because the accumulator is used later in the program.

The next instruction, ANI, ANDs (A) to mask all but bit D₀, as illustrated below.

(A)	→1	0	1	0	1	0	1	0
After RLC	→0	1	0	1	0	1	0	1
AND with 01H	→0	0	0	0	0	0	0	1
Remaining contents →0		0	0	0	0	0	0	1

This shows that 1 in D₀ provides a high pulse that stays on 250 µs because of the delay. In the next cycle of the loop, bit D₀ is at logic 0 because of the Rotate instruction, and the output pulse stays low for the next 250 µs.

Delay Calculations In this problem, the pulse width is relatively small (250 μ s); therefore, to obtain a reasonably accurate output pulse width, we should account for all the T-states. The total delay should include the delay in the loop and the execution time of the instructions outside the loop.

1. The number of instructions outside the loop is seven; it includes six instructions before the loop beginning at the symbol ROTATE and the last instruction JMP.

$$\text{Delay outside the Loop: } T_O = 46 \text{ T-states} \times 325 \text{ ns} = 14.95 \mu\text{s}$$

2. The delay loop includes two instructions (DCR and JNZ) with 14 T-states except for the last cycle, which has 11 T-states.

$$\begin{aligned}\text{Loop Delay: } T_L &= 14 \text{ T-states} \times 325 \text{ ns} \times (\text{Count} - 1) + 11 \text{ T-states} \times 325 \text{ ns} \\ &= 4.5 \mu\text{s} (\text{Count} - 1) + 3.575 \mu\text{s}\end{aligned}$$

3. The total delay required is 250 μ s. Therefore, the count can be calculated as follows:

$$\begin{aligned}T_D &= T_O + T_L \\ 250 \mu\text{s} &= 14.95 \mu\text{s} + 4.5 \mu\text{s} (\text{Count} - 1) + 3.575 \mu\text{s} \\ \text{Count} &= 52.4_{10} = 34H\end{aligned}$$

Program Output The output of bit D₀ can be observed on an oscilloscope; it should be a square wave with a period of 500 μ s.

8.5

DEBUGGING COUNTER AND TIME-DELAY PROGRAMS

The debugging techniques discussed in Chapters 6 and 7 can be used to check errors in a counter program. The following is a list of common errors in programs similar to those illustrated in this chapter.

1. Errors in counting T-states in a delay loop. Typically, the first instruction—to load a delay register—is mistakenly included in the loop.
2. Errors in recognizing how many times a loop is repeated.
3. Failure to convert a delay count from a decimal number into its hexadecimal equivalent.
4. Conversion error in converting a delay count from decimal to hexadecimal number or vice versa.
5. Specifying a wrong Jump location.
6. Failure to set a flag, especially with 16-bit Decrement/Increment instructions.
7. Using a wrong Jump instruction.
8. Failure to display either the first or the last count.
9. Failure to provide a delay between the last and the last-but-one count.

Some of these errors are illustrated in the following program.

8.5.1 Illustrative Program for Debugging

The following program is designed to count from 100_{10} to 0 in Hex continuously, with a 1-second delay between each count. The delay is set up by using two loops—a loop within a loop. The inner loop is expected to provide approximately 100 ms delay, and it is repeated ten times, using the outer loop to provide a total delay of 1 second. The clock period of the system is 330 ns. The program includes several deliberate errors. Recognize the errors as specified in the following assignment.

	Mnemonics	T-states
1.	MVI A,64H	7
2.	DISPLAY: OUT PORT1	10
3.	LOOP2: MVI B,10H	7
4.	LOOP1: LXI D,DELAY	10
5.	DCX D	6
6.	NOP	4
7.	NOP	4
8.	MOV A,D	4
9.	ORA E	4
10.	JNZ LOOP1	10/7
11.	DCR B	4
12.	JZ LOOP2	10/7
13.	DCR A	4
14.	CPI 00H	7
15.	JNZ DISPLAY	10/7

DELAY CALCULATIONS

$$\text{Delay in LOOP1} = \text{Loop T-states} \times \text{Count} \times \text{Clock period} (330 \times 10^{-9})$$

$$100 \text{ ms} = 32 \text{ T} \times \text{Count} \times 330 \times 10^{-9}$$

$$\begin{aligned}\text{DELAY COUNT} &= \frac{100 \times 10^{-3}}{32 \times 330 \times 10^{-9}} \\ &= 9470\end{aligned}$$

This delay calculation ignores the initial T-states in loading the count and the difference of T-states in the last execution of the conditional Jump instruction.

DEBUGGING QUESTIONS

1. Examine LOOP1. Is the label LOOP1 at the appropriate location? What is the effect of the present location on the program?
2. What is the appropriate place for the label LOOP1?
3. Is the delay count accurate?
4. What is the effect of instruction 8 (MOV A,D) on the count?
5. Should instruction 3 be part of LOOP2?
6. Is the byte in register B (instruction 3) accurate?
7. Calculate T-states in the outer loop using the appropriate place for the label LOOP2.
(Do not include the T-states of LOOP1.)

8. Is there any need for instruction 14 (CPI)?
9. Is there any need for an additional instruction, such as number 16?
10. What is the effect of instruction 12 (JZ) on the program?
11. Calculate the total delay in LOOP2 inclusive of LOOP1 if the byte in register B = 0AH.
12. Assuming instruction 8 is necessary, make appropriate changes in instructions 1 and 13.
13. Calculate the time delay between the display of two consecutive counts.
14. Will this program display the last count, assuming the other errors are corrected?

SUMMARY

- Counters and time delays can be designed using software.
- Time delays are designed simply by loading a count in a register or a register pair and decrementing the count by setting a loop until the count reaches zero. Time delay is determined by the number of T-states in a delay loop, the clock frequency, and the number of times the loop is repeated.
- Counters are designed using techniques similar to those used for time delays. A counter design generally includes a delay loop.
- In the 8085 microprocessor, 8-bit registers can be combined as register pairs (B and C, D and E, and H and L) to manipulate 16-bit data. Furthermore, the contents of each register can be examined separately even if registers are being used as register pairs.
- Sixteen-bit instructions such as DCX and INX do not affect flags; therefore, some other technique must be used to set flags.

QUESTIONS AND PROGRAMMING ASSIGNMENTS*

1. In Figure 8.2, calculate the loop delay T_L if register C contains 00H and the system clock frequency is 3.072 MHz. Adjust the delay calculations to account for seven T-states of the JNZ instruction in the last iteration.
2. In Section 8.1.2, load register pair BC with 8000H, and calculate the loop delay T_L if the system clock frequency is 3.072 MHz (ignore three T-state difference of the last cycle).
3. In Section 8.1.2, load register pair BC with 0000H and calculate the total delay T_D if the system clock period is 325 ns (adjust for the last cycle).

*The illustrative programs shown in this chapter and these assignments can be verified on a single-board microcomputer with a display output port.

4. Calculate the delay in the following loop, assuming the system clock period is 0.33 μ s:

8085		
Label	Mnemonics	T-states
	LXI B,12FFH	10
DELAY:	DCX B	6
	XTHL	16
	XTHL	16
	NOP	4
	NOP	4
	MOV A,C	4
	ORA B	4
	JNZ DELAY	10/7

5. In Figure 8.3, load register C with 00H and register B with C8H. Calculate the loop delay in LOOP1 and LOOP2 (clock period = 325 ns).
6. Specify the number of times the following loops are executed.
- | | | |
|-------------------|-------------------|---------------------|
| a. MVI A,17H | b. MVI A,17H | c. LXI B,1000H |
| LOOP: ORA A | LOOP: RAL | LOOP: DCX B |
| RAL | ORA A | NOP |
| JNC LOOP | JNC LOOP | JNZ LOOP |
7. Specify the number of times the following loops are executed.
- | | | |
|--------------------|---------------|-------------------|
| a. LOOP: MVI B,64H | b. ORA A | c. MVI A,17H |
| NOP | MVI B,64H | LOOP: ORA A |
| DCR B | LOOP: DCR B | RRC |
| JNZ LOOP | JNC LOOP | JNC LOOP |
8. Calculate the time delay in the Illustrative Program for a hexadecimal counter (Section 8.2), assuming a count of CFH in register C.
9. Recalculate the delay in the Illustrative Program for a zero-to-nine counter (Section 8.3) using the clock frequency of your system.
10. Calculate the COUNT to obtain a 100 μ s loop delay, and express the value in Hex. (Use the clock frequency of your system.)

T-states	
	MVI B,COUNT
LOOP: NOP	4
NOP	4
DCR B	4
JNZ LOOP	10/7

11. In Section 8.1.2, calculate the value of the 16-bit number that should be loaded in register BC to obtain the loop delay of 250 ms if the system clock period is 325 ns. Does the value change if it is calculated with seven T-states for the JNZ instruction in the last cycle?

12. Calculate the 16-bit count to be loaded in register DE to obtain the loop delay of two seconds in LOOP2 (use the clock period of your system and ignore the execution time of the first instruction MVI B).

MVI B,14H	(7)	;Count for outer loop
LOOP2: LXI D,16-bit	(10)	;Count for LOOP1
LOOP1: DCX D	(6)	
MOV A,D	(4)	
ORA E	(4)	
JNZ LOOP1	(10/7)	
DCR B	(4)	
JNZ LOOP2	(10/7)	

Use the clock frequency of your system in the following assignments.

13. Write a program to count from 0 to 20H with a delay of 100 ms between each count. After the count 20H, the counter should reset itself and repeat the sequence. Use register pair DE as a delay register. Draw a flowchart and show your calculations to set up the 100 ms delay.
14. Design an up-down counter to count from 0 to 9 and 9 to 0 continuously with a 1.5-second delay between each count, and display the count at one of the output ports. Draw a flowchart and show the delay calculations.
15. Write a program to turn a light on and off every 5 seconds. Use data bit D₇ to operate the light.
16. Write a program to generate a square wave with period of 400 μ s. Use bit D₀ to output the square wave.
17. Write a program to generate a rectangular wave with a 200 μ s on-period and a 400 μ s off-period.
18. A railway crossing signal has two flashing lights run by a microcomputer. One light is connected to data bit D₇ and the second light is connected to data bit D₆. Write a program to turn each signal light alternately on and off at an interval of 1 second.

9

Stack and Subroutines

The stack is a group of memory locations in the R/W memory that is used for temporary storage of binary information during the execution of a program. The starting memory location of the stack is defined in the main program, and space is reserved, usually at the high end of the memory map. The method of information storage resembles a stack of books. The contents of each memory location are, in a sense, “stacked”—one memory location above another—and information is retrieved starting from the top. Hence, this particular group of memory locations is called the *stack*. This chapter introduces the stack instructions in the 8085 set.

The latter part of this chapter deals with the subroutine technique, which is frequently used in programs. A subroutine is a group of instructions that performs a subtask (e.g., time delay or arithmetic operation) of repeated occurrence. The subroutine is written as a separate unit, apart from the main program, and the microprocessor transfers the program execution from the main program to

the subroutine whenever it is called to perform the task. After the completion of the subroutine task, the microprocessor returns to the main program. The subroutine technique eliminates the need to write a subtask repeatedly; thus, it uses memory more efficiently. Before implementing the subroutine technique, the stack must be defined; the stack is used to store the memory address of the instruction in the main program that follows the subroutine call.

The stack and the subroutine offer a great deal of flexibility in writing programs. A large software project is usually divided into subtasks called **modules**. These modules are developed independently as subroutines by different programmers. Each programmer can use all the microprocessor registers to write a subroutine without affecting the other parts of the program. At the beginning of the subroutine module, the register contents of the main program are stored on the stack, and these register contents are retrieved before returning to the main program.

This chapter includes two illustrative programs: The first illustrates the use of the stack-related instructions to examine and manipulate the flags; and the second illustrates the subroutine technique in a traffic-signal controller.

OBJECTIVES

- Define the stack, the stack pointer (register), and the program counter, and describe their uses.
- Explain how information is stored and retrieved from the stack using the instructions PUSH and POP and the stack pointer register.
- Demonstrate how the contents of the flag register can be displayed and how a given flag can be set or reset.
- Define a subroutine and explain its uses.
- Explain the sequence of a program execution when a subroutine is called and executed.
- Explain how information is exchanged between the program counter and the stack, and identify the contents of the stack pointer register when a subroutine is called.
- List and explain conditional Call and Return instructions.
- Illustrate the concepts in the following subroutines: multiple calling, nesting, and common ending.
- Compare similarities and differences between PUSH/POP and CALL/RET instructions.

9.1 STACK

The **stack** in an 8085 microcomputer system can be described as a set of memory locations in the R/W memory, specified by a programmer in a main program. These memory locations are used to store binary information (bytes) temporarily during the execution of a program.

The beginning of the stack is defined in the program by using the instruction LXI SP, which loads a 16-bit memory address in the stack pointer register of the microprocessor. Once the stack location is defined, storing of data bytes begins at the memory address that is one less than the address in the stack pointer register. For example, if the stack pointer register is loaded with the memory address 2099H (LXI SP,2099H), the storing of data bytes begins at 2098H and continues in reversed numerical order (decreasing memory addresses such as 2098H, 2097H, etc.). Therefore, as a general practice, the stack is initialized at the highest available memory location to prevent the program from being destroyed by the stack information. The size of the stack is limited only by the available memory.

Data bytes in the register pairs of the microprocessor can be stored on the stack (two at a time) in reverse order (decreasing memory address) by using the instruction PUSH. Data bytes can be transferred from the stack to respective registers by using the instruction POP. The stack pointer register tracks the storage and retrieval of the information. Because two data bytes are being stored at a time, the 16-bit memory address in the stack pointer register is decremented by two; when data bytes are retrieved, the address is incremented by two. An address in the stack pointer register indicates that the next two memory locations (in descending numerical order) can be used for storage.

The stack is shared by the programmer and the microprocessor. The programmer can store and retrieve the contents of a register pair by using PUSH and POP instructions. Similarly, the microprocessor automatically stores the contents of the program counter when a subroutine is called (to be discussed in the next section). The instructions necessary for using the stack are explained below.

INSTRUCTIONS

Opcode	Operand
--------	---------

LXI	SP,16-bit	<input type="checkbox"/> Load Stack Pointer <input type="checkbox"/> Load the stack pointer register with a 16-bit address. The LXI instructions were discussed in Chapter 7
PUSH	Rp	Store Register Pair on Stack <input type="checkbox"/> This is a 1-byte instruction
PUSH	B	<input type="checkbox"/> It copies the contents of the specified register pair on the stack as described below
PUSH	D	<input type="checkbox"/> The stack pointer register is decremented, and the contents of the high-order register (e.g., register B) are copied in the location shown by the stack pointer register
PUSH	H	<input type="checkbox"/> The stack pointer register is again decremented, and the contents of the low-order register (e.g., register C) are copied in that location
PUSH	PSW	<input type="checkbox"/> The operands B, D, and H represent register pairs BC, DE, and HL, respectively <input type="checkbox"/> The operand PSW represents Program Status Word, meaning the contents of the accumulator and the flags
POP	Rp	Retrieve Register Pair from Stack <input type="checkbox"/> This is a 1-byte instruction
POP	B	<input type="checkbox"/> It copies the contents of the top two memory locations of the stack into the specified register pair
POP	D	<input type="checkbox"/> First, the contents of the memory location indicated by the stack pointer register are copied into the low-order register (e.g., register L), and then the stack pointer register is incremented by 1
POP	H	<input type="checkbox"/> The contents of the next memory location are copied into the high-order register (e.g., register H), and the stack pointer register is again incremented by 1
POP	PSW	

All three of these instructions belong to the data transfer (copy) group; thus, the contents of the source are not modified, and no flags are affected.

In the following set of instructions (illustrated in Figure 9.1), the stack pointer is initialized, and the contents of register pair HL are stored on the stack by using the PUSH instruction. Register pair HL is used for the delay counter (actual instructions are not

Example
9.1

shown); at the end of the delay counter, the contents of HL are retrieved by using the instruction POP. Assuming the available user memory ranges from 2000H to 20FFH, illustrate the contents of various registers when PUSH and POP instructions are executed.

Solution

In this example, the first instruction—LXI SP,2099H—loads the stack pointer register with the address 2099H (Figure 9.1). This instruction indicates to the microprocessor that memory space is reserved in the R/W memory as the stack and that the locations beginning at 2098H and moving upward can be used for temporary storage. This instruction also suggests that the stack can be initialized anywhere in the memory; however, the stack location should not interfere with a program. The next instruction—LXI H—loads data in the HL register pair, as shown in Figure 9.1.

When instruction PUSH H is executed, the following sequence of data transfer takes place. After the execution, the contents of the stack and the register are as shown in Figure 9.2.

1. The stack pointer is decremented by one to 2098H, and the contents of the H register are copied to memory location 2098H.
2. The stack pointer register is again decremented by one to 2097H, and the contents of the L register are copied to memory location 2097H.
3. The contents of the register pair HL are not destroyed; however, HL is made available for the delay counter.

After the delay counter, the instruction POP H restores the original contents of the register pair HL, as follows. Figure 9.3 illustrates the contents of the stack and the registers following the POP instruction.

1. The contents of the top of the stack location shown by the stack pointer are copied in the L register, and the stack pointer register is incremented by one to 2098H.
2. The contents of the top of the stack (now it is 2098H) are copied in the H register, and the stack pointer is incremented by one.

Memory Location	Mnemonics	Register Contents					
		A	B	C	D	E	F
2000	LXI SP,2099H;						
2003	LXI H,42F2H;						
2006	PUSH H						
2007	DELAY COUNTER						
200F	↓						
2010	POP H						

;Store contents of register HL on the stack
;The register pair HL can be used by the Delay Counter if necessary
;Load HL registers with the contents of the two top locations of the stack

FIGURE 9.1
Instructions and Register Contents in Example 9.1

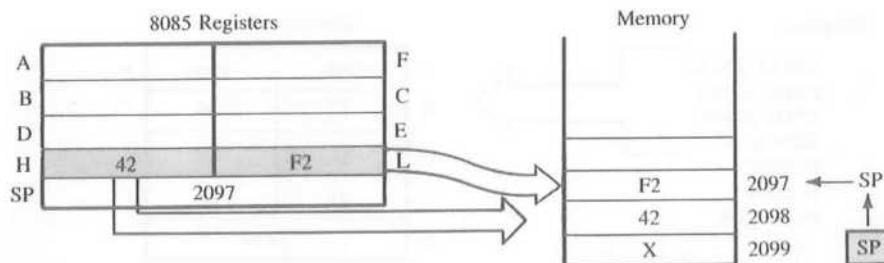


FIGURE 9.2
Contents on the Stack and in the Registers After the PUSH Instruction

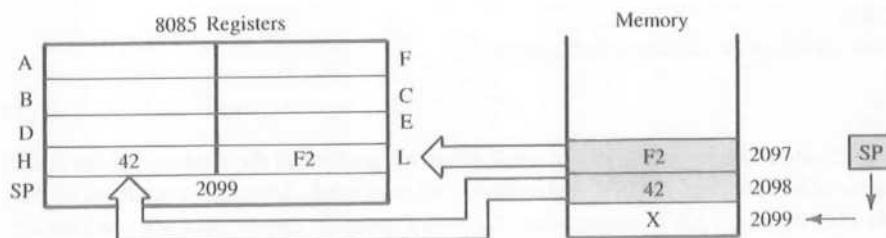


FIGURE 9.3
Contents on the Stack and in the Registers After the POP Instruction

3. The contents of memory locations 2097H and 2098H are not destroyed until some other data bytes are stored in these locations.

The available user memory ranges from 2000H to 23FFH. A program of data transfer and arithmetic operations is stored in memory locations from 2000H to 2050H, and the stack pointer is initialized at location 2400H. Two sets of data are stored, starting at locations 2150H and 2280H (not shown in Figure 9.4). Registers HL and BC are used as memory pointers to the data locations. A segment of the program is shown in Figure 9.4.

Example
9.2

1. Explain how the stack pointer can be initialized at one memory location beyond the available user memory.
 2. Illustrate the contents of the stack memory and registers when PUSH and POP instructions are executed, and explain how memory pointers are exchanged.
 3. Explain the various contents of the user memory.
1. The program initializes the stack pointer register at location 2400H, one location beyond the user memory (Figure 9.4). This procedure is valid because the initialized location is never used for storing information. **The instruction PUSH first decrements the stack pointer register, and then stores a data byte.**

Solution

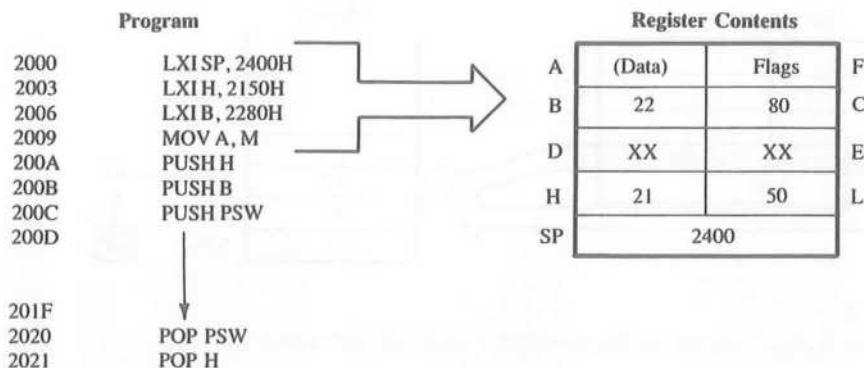


FIGURE 9.4
Instructions and Register Contents in Example 9.2

- Figure 9.5 shows the contents of the stack pointer register and the contents of the stack locations after the three PUSH instructions are executed. After the execution of the PUSH (H, B, and PSW) instructions, the stack pointer moves upward (decreasing memory locations) as the information is stored. Thus the stack can grow upward in the user memory even to the extent of destroying the program.

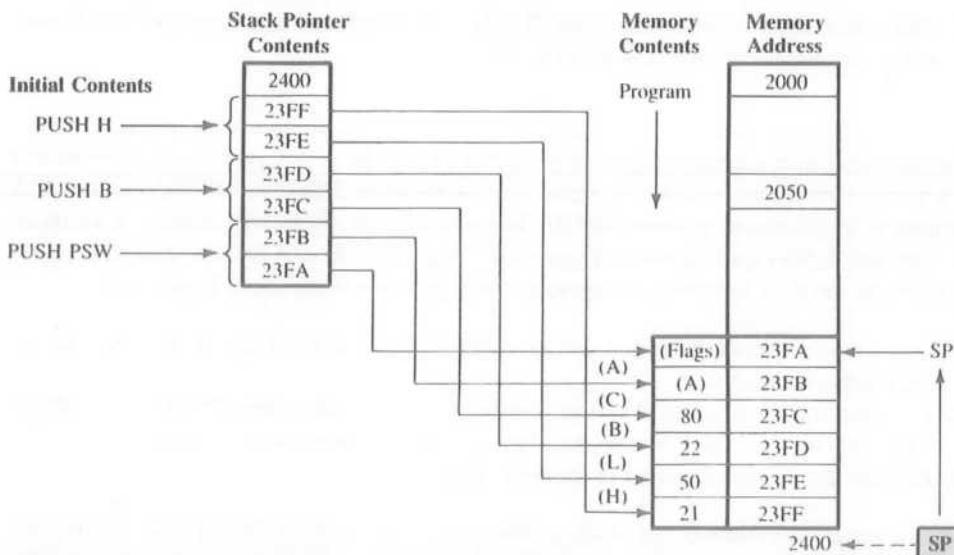


FIGURE 9.5
Stack Contents After the Execution of PUSH Instructions

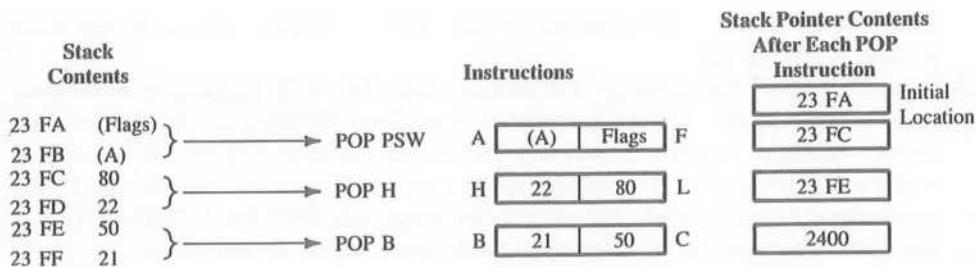
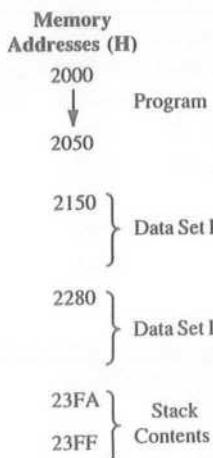


FIGURE 9.6
Register Contents After the Execution of POP Instructions

Figure 9.6 shows how the contents of various register pairs are retrieved. To restore the original contents in the respective registers, follow the sequence Last-In-First-Out (LIFO). In the example, the register contents were pushed on the stack in the order of HL, BC, and PSW. The contents should have been restored in the order of PSW, BC, and HL. However, the order is altered in this example to demonstrate how register contents are exchanged.

The instruction POP PSW copies the contents of the two top locations to the flag register and the accumulator, respectively, and increments the stack pointer by two to 23FCH. The next instruction, POP H, takes the contents of the top two locations (23FC and 23FD), and copies them in registers L and H, respectively, while incrementing the stack pointer by two to 23FEH. The instruction POP B copies the contents of the next two locations in registers C and B, incrementing the stack pointer to 2400H. By reversing the positions of two instructions, POP H and POP B, the contents of the BC pair are exchanged with those of the HL pair. It is important to remember that the instruction POP H does not restore the original contents of the HL

FIGURE 9.7
R/W Memory Contents



- pair; instead it copies the contents of the top two locations shown by the stack pointer in the HL pair.
3. Figure 9.7 shows the sketch of the memory map. The R/W memory includes three types of information. The user program is stored from 2000H to 2050H. The data are stored, starting at locations 2150H and 2280H. The last section of the user memory is initialized as the stack where register contents are stored as necessary, using the PUSH instructions. In this example, the memory locations from 23FFH to 23FAH are used as the stack, which can be extended up to the locations of the second data set.
-

9.1.1 Review of Important Concepts

The following points can be summarized from the preceding examples:

1. Memory locations in R/W memory can be employed as temporary storage for information by initializing (loading) a 16-bit address in the stack pointer register; these memory locations are called the stack. The terms *stack* and *stack pointer* appear similar; however, they are not the same. The stack is memory locations in R/W memory; the stack pointer is a 16-bit register in the 8085 microprocessor.
2. Read/Write memory generally is used for three different purposes:
 - a. to store programs or instructions;
 - b. to store data;
 - c. to store information temporarily in defined memory locations called the stack during the execution of the program.
3. The stack space grows upward in the numerically decreasing order of memory addresses.
4. The stack can be initialized anywhere in the user memory map. However, as a general practice, the stack is initialized at the highest user memory location so that it will be less likely to interfere with a program.
5. A programmer can employ the stack to store contents of register pairs by using the instruction PUSH and can restore the contents of register pairs by using the instruction POP. The address in the stack pointer register always points to the top of the stack, and the address is decremented or incremented as information is stored or retrieved.
6. The contents of the stack pointer can be interpreted as the address of the memory location that is already used for storage. The retrieval of bytes begins at the address in the stack pointer; however, the storage begins at the next memory location (in the decreasing order).
7. The storage and retrieval of data bytes on the stacks should follow the LIFO (Last-In-First-Out) sequence. Information in stack locations is not destroyed until new information is stored in those locations.

9.1.2 Illustrative Program: Resetting and Displaying Flags

PROBLEM STATEMENT

Write a program to perform the following functions:

1. Clear all the flags.
2. Load 00H in the accumulator, and demonstrate that the Zero flag is not affected by the data transfer instruction.
3. Logically OR the accumulator with itself to set the Zero flag, and display the flag at PORT1 or store all the flags on the stack.

PROBLEM ANALYSIS

The problem concerns examining the Zero flag after instructions have been executed. There is no direct way of observing the flags; however, they can be stored on the stack by using the instruction PUSH PSW. The contents of the flag register can be retrieved in any one of the registers by using the instruction POP, and the flags can be displayed at an output port. In this example, the result to be displayed is different from the result to be stored on the stack memory. To display only the Zero flag, all other flags should be masked; however, the masking is not necessary to store all the flags.

PROGRAM

Memory Address	Machine Code	Instructions	Comments
XX00	31	LXI SP,XX99H	;Initialize the stack
01	99		
02	XX		
03	2E	MVI L,00H	;Clear L
04	00		
05	E5	PUSH H	;Place (L) on stack
06	F1	POP PSW	;Clear flags
07	3E	MVI A,00H	;Load 00H
08	00		
09	F5	PUSH PSW	;Save flags on stack
0A	E1	POP H	;Retrieve flags in L
0B	7D	MOV A,L	
0C	D3	OUT PORT0	;Display flags
0D	PORT0		
0E	3E	MVI A,00H	;Load 00H again
0F	00		
10	B7	ORA A	;Set flags and reset CY, AC
11	F5	PUSH PSW	;Save flags on stack
12	E1	POP H	;Retrieve flags in L
13	7D	MOV A,L	
14	E6	ANI 40H	;Mask all flags except Z
15	40		
16	D3	OUT PORT1	
17	PORT1		
18	76	HLT	;End of program

 Storing in Memory: Alternative to Output Display

XX0A	3E	MVI A,00H	;Load 00H again
0B	00		
0C	B7	ORA A	;Set flags and reset CY and AC
0D	F5	PUSH PSW	;Save flags on stack
0E	76	HLT	;End of program

Program Description The stack pointer register is initialized at XX99H. The instruction MVI L clears (L), and (L) is placed on the stack, which is subsequently placed into the flag register to clear all the flags.

To verify the flags after the execution of the MVI A instruction, the PUSH and POP instructions are used in the same way as these instructions were used to clear the flags, and the flags are displayed at PORT0. Similarly, the Zero flag is displayed at PORT1 after the instructions MVI and ORA.

Program Output Data transfer (copy) instructions do not affect the flags; therefore, no flags should be set after the instruction MVI A, even if (A) is equal to zero. PORT0 should display 00H. However, the instruction ORA will set the Zero and the Parity flags to reflect the data conditions in the accumulator, and it also resets the CY and AC flags. In the flag register, bit D₆ represents the Z flag, and the ANI instruction masks all flags except the Z flag. PORT1 should display 40H as shown below.

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	1	0	0	0	1	0	0
S	Z	AC	P		CY		

= 40H

Storing Output in Memory If output ports are not available, the results can be stored in the stack memory. The machine code (F5) at memory location XX09H saves the flags affected by the instruction MVI A,00H. Then the instructions can be modified starting from memory location XX0AH. The alternative set of instructions is shown above; it sets the flags (using ORA instruction), and saves them on the stack without masking. The result (44H) includes the parity flag. The contents of the stack locations should be as shown in Figure 9.8.

Instructions	Stack Memory	Contents	
	XX99		Stack Pointer Initialization
XX09 PUSH PSW:	XX98	(A) = 00H	
	XX97	(F) = 00H	
XX0D PUSH PSW:	XX96	(A) = 00H	
	XX95	(F) = 44H	

FIGURE 9.8
Output Stored in Stack Memory

SUBROUTINE

9.2

A **subroutine** is a group of instructions written separately from the main program to perform a function that occurs repeatedly in the main program. For example, if a time delay is required between three successive events, three delays can be written in the main program. To avoid repetition of the same delay instructions, the subroutine technique is used. Delay instructions are written once, separately from the main program, and are called by the main program when needed.

The 8085 microprocessor has two instructions to implement subroutines: CALL (call a subroutine), and RET (return to main program from a subroutine). The CALL instruction is used in the main program to call a subroutine, and the RET instruction is used at the end of the subroutine to return to the main program. When a subroutine is called, the contents of the program counter, which is the address of the instruction following the CALL instruction, is stored on the stack and the program execution is transferred to the subroutine address. When the RET instruction is executed at the end of the subroutine, the memory address stored on the stack is retrieved, and the sequence of execution is resumed in the main program. This sequence of events is illustrated in Example 9.3.

INSTRUCTIONS

Opcode	Operand	
CALL	16-bit memory address of a subroutine	<p>Call Subroutine Unconditionally</p> <ul style="list-style-type: none"><input type="checkbox"/> This is a 3-byte instruction that transfers the program sequence to a subroutine address<input type="checkbox"/> Saves the contents of the program counter (the address of the next instruction) on the stack<input type="checkbox"/> Decrements the stack pointer register by two<input type="checkbox"/> Jumps unconditionally to the memory location specified by the second and third bytes. The second byte specifies a line number and the third byte specifies a page number<input type="checkbox"/> This instruction is accompanied by a return instruction in the subroutine
RET		<p>Return from Subroutine Unconditionally</p> <ul style="list-style-type: none"><input type="checkbox"/> This is a 1-byte instruction<input type="checkbox"/> Inserts the two bytes from the top of the stack into the program counter and increments the stack pointer register by two<input type="checkbox"/> Unconditionally returns from a subroutine

The conditional Call and Return instructions will be described later in the chapter.

**Example
9.3**

Illustrate the exchange of information between the stack and the program counter for the following program if the available user memory ranges from 2000H to 23FFH.

Memory Address		
2000	LXI SP,2400H	;Initialize the stack pointer at 2400H
↓	↓	
2040	CALL 2070H	;Call the subroutine located at 2070H. This is
2041		; a 3-byte instruction
2042		
2043	NEXT INSTRUCTION	;The address of the next instruction following ; the CALL instruction
↓	↓	
205F	HLT	;End of the main program
2070	First Subroutine Instruction	;Beginning of the subroutine
↓	↓	
207F	RET	;End of the subroutine
2080	↓	
↓	Other Subroutines	
2398	Empty Space	
23FF	↓	
2400		;The stack is initialized at 2400H

Solution

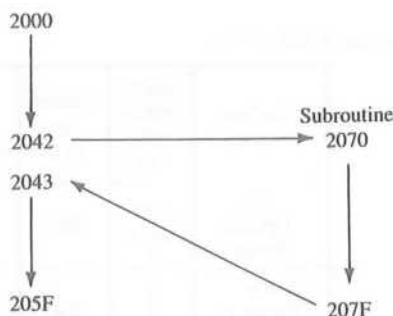
After reviewing the above program note the following points:

1. The available user memory is from 2000H to 23FFH (1024 or 1K bytes); however, the stack pointer register is initialized at 2400H, one location beyond the user memory. This allows maximum use of the memory because the actual stack begins at 23FFH. The stack can expand up to the location 2398H without overlapping with the program.
2. The main program is stored at memory locations from 2000H to 205FH.
3. The CALL instruction is located at 2040H to 2042H (3-byte instruction). The next instruction is at 2043H.
4. The subroutine begins at the address 2070H and ends at 207FH.

PROGRAM EXECUTION

The sequence of the program execution and the events in the execution of the CALL and subroutine are shown in Figures 9.9 and 9.10.

FIGURE 9.9
Subroutine Call and Program Transfer



The program execution begins at 2000_{16} , continues until the end of $\text{CALL } 2042_{16}$, and transfers to the subroutine at 2070_{16} . At the end of the subroutine, after executing the RET instruction, it comes back to the main program at 2043_{16} and continues.

CALL EXECUTION

Memory Address	Machine Code	Mnemonics	Comments
2040	CD	$\text{CALL } 2070_{16}$;Call subroutine located at the memory
2041	70		; location 2070_{16}
2042	20		
2043	NEXT	INSTRUCTION	

The sequence of events in the execution of the **CALL** instruction by the 8085 is shown in Figure 9.10. The instruction requires five machine cycles and eighteen T-states. The sequence of events in each machine cycle is as follows.

1. M_1 —Opcode Fetch: In this machine cycle, the contents of the program counter (2040_{16}) are placed on the address bus, and the instruction code CD is fetched using the data bus. At the same time, the program counter is upgraded to the next memory address, 2041_{16} . After the instruction is decoded and executed, the stack pointer register is decremented by one to $23FF_{16}$.
2. M_2 and M_3 —Memory Read: These are two Memory Read cycles during which the 16-bit address (2070_{16}) of the CALL instruction is fetched. The low-order address 70_{16} is fetched first and placed in the internal register Z. The high-order address 20_{16} is fetched next, and placed in register W. During M_3 , the program counter is upgraded to 2043_{16} , pointing to the next instruction.
3. M_4 and M_5 —Storing of Program Counter: At the beginning of the M_4 cycle, the normal operation of placing the contents of the program counter on the address bus is suspended; instead, the contents of the stack pointer register $23FF_{16}$ are placed on the address bus. The high-order byte of the program counter ($PCH = 20_{16}$) is placed on the data bus and stored in the stack location $23FF_{16}$. At the same time, the stack pointer register is decremented to $23FE_{16}$.

During machine cycle M_5 , the contents of the stack pointer $23FE_{16}$ are placed on the address bus. The low-order byte of the program counter ($PCL = 43_{16}$) is placed on the data bus and stored in stack location $23FE_{16}$.

Instruction: CALL 2070H

Machine Cycles	Stack Pointer (SP) 2400	Address Bus (AB)	Program Counter (PCH) (PCL)	Data Bus (DB)	Internal Registers (W)(Z)
M ₁ Opcode Fetch	23FF (SP-1)	2040	20 41	CD Opcode	—
M ₂ Memory Read		2041	20 42	70 Operand	→ 70
M ₃ Memory Read	23FF	2042	20 43	20 Operand	→ 20
M ₄ Memory Write	23FE (SP-2)	23FF	20 43	20 (PCH)	
M ₅ Memory Write	23FE	23FE	20 43	43 (PCL)	(20)(70)
M ₁ Opcode Fetch of Next Instruction		2070 → 2071			(2070) (W)(Z)

Memory Address	Code (H)
2040	CD
2041	70
2042	20

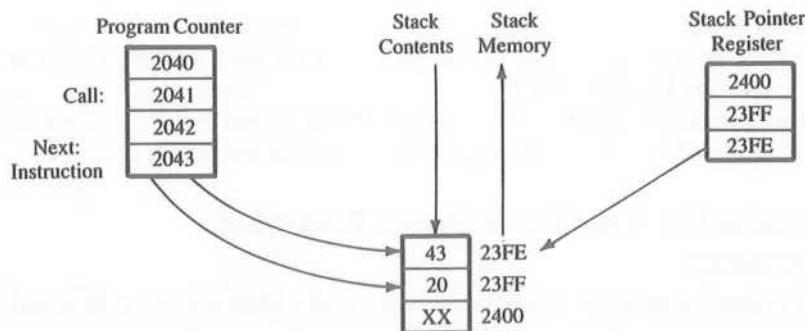
FIGURE 9.10
Data Transfer During the Execution of the CALL Instruction

4. Next Instruction Cycle: In the next instruction cycle, the program execution sequence is transferred to the CALL location 2070H by placing the contents of W and Z registers (2070H) on the address bus. During M₁ of the next instruction cycle, the program counter is upgraded to location 2071 (W,Z + 1).

In summary: After the CALL instruction is fetched, the 16-bit address (the operand) is read during M₂ and M₃ and stored temporarily in W/Z registers. (Examine the contents of the address bus and the data bus in Figure 9.10.) In the next two cycles, the contents of the program counter are stored on the stack. This is the address where the microprocessor will continue the execution of the program after the completion of the subroutine. Figure 9.11 shows the contents of the program counter, the stack pointer register, and the stack during the execution of the CALL instruction.

RET EXECUTION

At the end of the subroutine, when the instruction RET is executed, the program execution sequence is transferred to the memory location 2043H. The address 2043H was

**FIGURE 9.11**

Contents of the Program Counter, the Stack Pointer, and the Stack During the Execution of the CALL Instruction

stored in the top two locations of the stack (23FEH and 23FFH) during the CALL instruction. Figure 9.12 shows the sequence of events that occurs as the instruction RET is executed.

M_1 is a normal Opcode Fetch cycle. However, during M_2 the contents of the stack pointer register are placed on the address bus, rather than those of the program counter. Data byte 43H from the top of the stack is fetched and stored in the Z register, and the

Instruction: RET

Memory Address	Code (H)	Machine Cycles	Stack Pointer (23FE)	Address Bus (AB)	Program Counter	Data Bus (DB)	Internal Registers (W) (Z)
207F	C9	M_1 Opcode Fetch	23FE	207F	2080	C9 Opcode	
		M_2 Memory Read	23FF	23FE		43 (Stack)	43
		M_3 Memory Read	2400	23FF		20 (Stack-1)	20
		M_1 Opcode Fetch of Next Instruction		2043 (W) (Z)	2044		2043 (W) (Z)

Contents of Stack Memory:

23FE	43
23FF	20

FIGURE 9.12

Data Transfer During the Execution of the RET Instruction

stack pointer register is upgraded to the next location, 23FFH. During M₂, the next byte—20H—is copied from the stack and stored in register W, and the stack pointer register is again upgraded to the next location, 2400H.

The program sequence is transferred to location 2043H by placing the contents of the W/Z registers on the address bus at the beginning of the next instruction cycle.

9.2.1 Illustrative Program: Traffic Signal Controller

PROBLEM STATEMENT

Write a program to provide the given on/off time to three traffic lights (Green, Yellow, and Red) and two pedestrian signs (WALK and DON'T WALK). The signal lights and signs are turned on/off by the data bits of an output port as shown below:

Lights	Data Bits	On Time
1. Green	D ₀	15 seconds
2. Yellow	D ₂	5 seconds
3. Red	D ₄	20 seconds
4. WALK	D ₆	15 seconds
5. DON'T WALK	D ₇	25 seconds

The traffic and pedestrian flow are in the same direction; the pedestrian should cross the road when the Green light is on.

PROBLEM ANALYSIS

The problem is primarily concerned with providing various time delays for a complete sequence of 40 seconds. The on/off times for the traffic signals and pedestrian signs are as follows:

Time Sequence in Seconds	DON'T WALK	WALK	Red	Yellow	Green	Hex Code				
0	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀		
(15) ↓									=	41H
15	0	1	0	0	0	0	0	1	=	84H
(5) ↓										
20	1	0	0	0	1	0	0	0	=	90H
(20) ↓										
40	1	0	0	1	0	0	0	0	=	

The Green light and the WALK sign can be turned on by sending data byte 41H to the output port. The 15-second delay can be provided by using a 1-second subroutine and

a counter with a count of 15_{10} . Similarly, the next two bytes, 84H and 90H, will turn on/off the appropriate lights/signs as shown in the flowchart (Figure 9.13). The necessary time delays are provided by changing the values of the count in the counter.

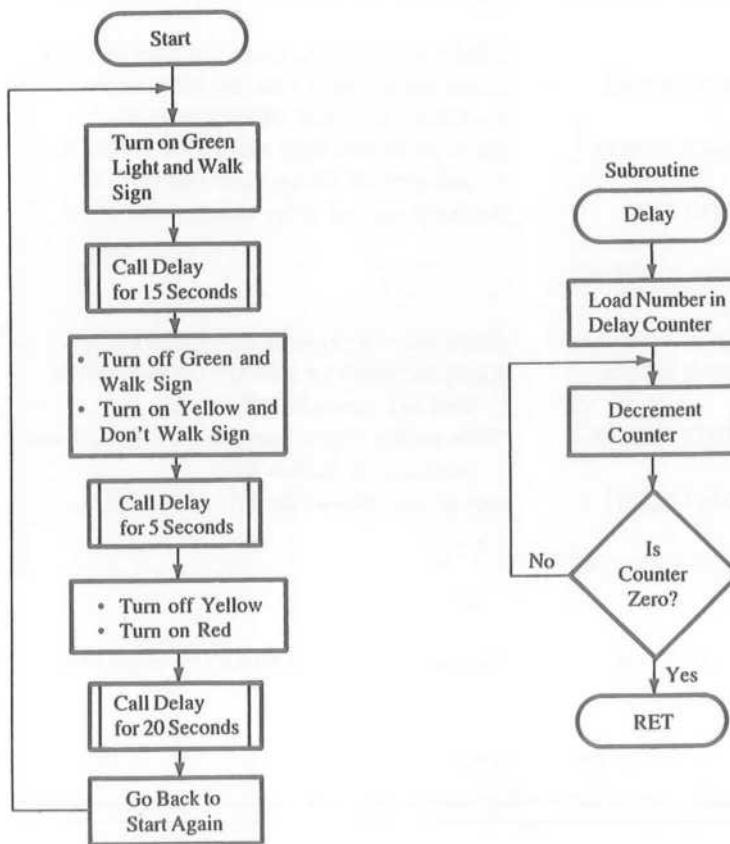


FIGURE 9.13
Flowchart for Traffic Signal Controller

PROGRAM

Memory

Address	Code	Mnemonics	Comments
XX00	31		
01	99	LXI SP,XX99	;Initialize stack pointer at location XX99H
02	XX		;High-order address (page) of user memory
03	3E	START: MVI A,41H	;Load accumulator with the bit pattern for
04	41		; Green light and WALK sign
05	D3	OUT PORT#	;Turn on Green light and WALK sign

06	POR#		
07	06	MVI B,0FH	;Use B as a counter to count 15seconds.
08	0F		;
09	CD	CALL DELAY	B is decremented in the subroutine
0A	50		;Call delay subroutine located at XX50H
0B	XX		
0C	3E	MVI A,84H	;High-order address (page) of user memory
0D	84		;Load accumulator with the bit pattern for
0E	D3	OUT PORT#	;
0F	POR#		Yellow light and DON'T WALK
10	06	MVI B,05	;Turn on Yellow light and DON'T WALK
11	05		;
12	CD	CALL DELAY	and turn off Green light and WALK
13	50		;Set up 5-second delay counter
14	XX		
15	3E	MVI A,90H	;High-order address of user memory
16	90		;Load accumulator with the bit pattern for
17	D3	OUT PORT#	;
18	POR#		Red light and DON'T WALK
19	06	MVI B,14H	;Turn on Red light, keep DON'T WALK on,
1A	14		;
1B	CD	CALL DELAY	and turn off Yellow light
1C	50		;Set up the counter for 20-second delay
1D	XX		
1E	C3	JMP START	;Go back to location START to repeat the
1F	03		;
20	XX		sequence

:DELAY: This is a 1-second delay subroutine that provides delay

; according to the parameter specified in register B

:Input: Number of seconds is specified in register B

:Output: None

:Registers Modified: Register B

XX50	D5	DELAY: PUSH D	;Save contents of DE and accumulator
51	F5	PUSH PSW	
52	11	SECOND: LXI D,COUNT	;Load register pair DE with a count for
53	LO		;
54	HI		1-second delay
55	1B	Loop: DCX D	;Decrement register pair DE
56	7A	MOV A,D	
57	B3	ORA E	;OR (D) and (E) to set Zero flag
58	C2	JNZ LOOP	;Jump to Loop if delay count is not equal to 0
59	55		
5A	XX		

5B	05	DCR B	;End of 1 second delay; decrement the counter
5C	C2	JNZ SECOND	;Is this the end of time needed? If not, go
5D	52		; back to repeat 1-second delay
5E	XX		;High-order memory address of user memory
5F	F1	POP PSW	;Retrieve contents of saved registers
60	D1	POP D	
61	C9	RET	;Return to main program

PROGRAM DESCRIPTION

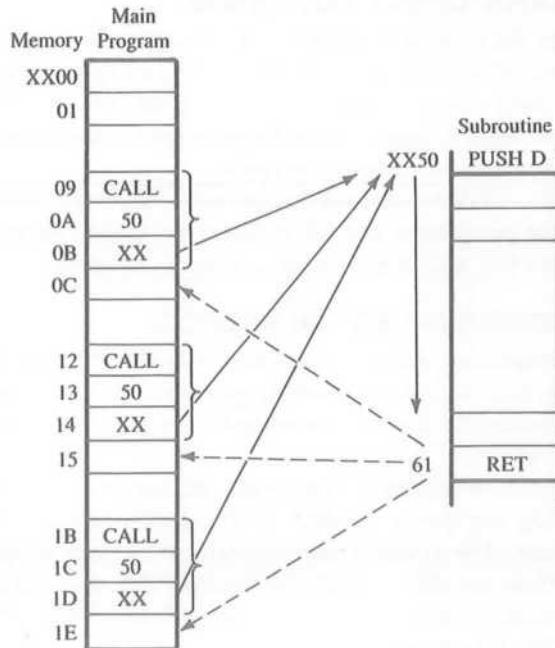
The stack pointer register is initialized at XX99H so that return addresses can be stored on the stack whenever a CALL instruction is used. As shown in the flowchart this program loads the appropriate bit pattern in the accumulator, sends it to the output port, and calls the delay routine. Register B is loaded in the main program and used in the subroutine to provide appropriate timing.

The DELAY subroutine is similar to the delays discussed in Chapter 8 except it requires the instruction RET at the end of the routine.

This example illustrates the type of subroutine that is called many times from various locations in the main program, as illustrated in Figure 9.14.

In this program, the subroutine is called from the locations XX09, 0A, and 0BH. The return address, XX0C, is stored on the stack, and the stack pointer is decremented by two to location XX97H. At the end of the subroutine, the contents of the top two loca-

FIGURE 9.14
Multiple-Calling for a Subroutine



tions of the stack (XX0C) are retrieved, the stack pointer register is incremented by two to the original location (XX99H), and the main program is resumed. This sequence is repeated two more times in the main program, as shown in Figure 9.14. This is called a multiple-calling subroutine.

9.2.2 Subroutine Documentation and Parameter Passing

In a large program, subroutines are scattered all over the memory map and are called from many locations. Various information is passed between a calling program and a subroutine, a procedure called **parameter passing**. Therefore, it is important to document a subroutine clearly and carefully. The documentation should include at least the following:

1. Functions of the subroutine
2. Input/Output parameters
3. Registers used or modified
4. List of other subroutines called by this subroutine

The delay subroutine in the traffic-signal controller program shows one example of subroutine documentation.

FUNCTIONS OF THE SUBROUTINE

It is important to state clearly and precisely what the subroutine does. A user should understand the function without going through the instructions.

INPUT/OUTPUT PARAMETERS

In the delay subroutine illustrated in Section 9.2.1, the information concerning the number of seconds is passed from the main program to the subroutine by loading an appropriate count in register B. In this example, a register is used to pass the parameter. The parameters passed to a subroutine are listed as **inputs**, and parameters returned to calling programs are listed as **outputs**.

When many parameters must be passed, R/W memory locations are used to store the parameters, and HL registers are used to point to parameter locations. Similarly, the stack is also used to store and pass parameters.

REGISTERS USED OR MODIFIED

Registers used in a subroutine also may be used by the calling program. Therefore, it is necessary to save the register contents of the calling program on the stack at the beginning of the subroutine and to retrieve the contents before returning from the subroutine.

In the delay subroutine, the contents of registers DE, the accumulator, and the flag register are pushed on the stack because these registers are used in the subroutine. The contents are restored at the end of the routine using the LIFO method. However, the contents of registers that pass parameters should not be saved on the stack because this could cause irrelevant information to be retrieved and passed on to the calling program.

LIST OF SUBROUTINES CALLED

If a subroutine is calling other subroutines, the user should be provided with a list. The user can check what parameters need to be passed to various subroutines and what registers are modified in the process.

RESTART, CONDITIONAL CALL, AND RETURN INSTRUCTIONS

9.3

In addition to the unconditional CALL and RET instructions, the 8085 instruction set includes eight Restart instructions and eight conditional Call and Return instructions.

9.3.1 Restart (RST) Instructions

RST instructions are 1-byte Call instructions that transfer the program execution to a specific location on page 00H. They are executed the same way as Call instructions. When an RST instruction is executed, the 8085 stores the contents of the program counter (the address of the next instruction) on the top of the stack and transfers the program to the Restart location. These instructions are generally used in conjunction with the interrupt process discussed in Chapter 12. These instructions are listed here to emphasize that they are Call instructions and not necessarily always associated with the interrupts. The list of eight RST instructions is as follows:

RST 0	Call 0000H	RST 4	Call 0020H
RST 1	Call 0008H	RST 5	Call 0028H
RST 2	Call 0010H	RST 6	Call 0030H
RST 3	Call 0018H	RST 7	Call 0038H

9.3.2 Conditional Call and Return Instructions

The conditional Call and Return instructions are based on four data conditions (flags): Carry, Zero, Sign, and Parity. The conditions are tested by checking the respective flags. In case of a conditional Call instruction, the program is transferred to the subroutine if the condition is met; otherwise, the main program is continued. In case of a conditional Return instruction, the sequence returns to the main program if the condition is met; otherwise, the sequence in the subroutine is continued. If the Call instruction in the main program is conditional, the Return instruction in the subroutine can be conditional or unconditional. The conditional Call and Return instructions are listed for reference.

CONDITIONAL CALL

CC	Call subroutine if Carry flag is set (CY = 1)
CNC	Call subroutine if Carry flag is reset (CY = 0)
CZ	Call subroutine if Zero flag is set (Z = 1)
CNZ	Call subroutine if Zero flag is reset (Z = 0)
CM	Call subroutine if Sign flag is set (S = 1, negative number)
CP	Call subroutine if Sign flag is reset (S = 0, positive number)

- CPE Call subroutine if Parity flag is set ($P = 1$, even parity)
 CPO Call subroutine if Parity flag is reset ($P = 0$, odd parity)

CONDITIONAL RETURN

- RC Return if Carry flag is set ($CY = 1$)
 RNC Return if Carry flag is reset ($CY = 0$)
 RZ Return if Zero flag is set ($Z = 1$)
 RNZ Return if Zero flag is reset ($Z = 0$)
 RM Return if Sign flag is set ($S = 1$, negative number)
 RP Return if Sign flag is reset ($S = 0$, positive number)
 RPE Return if Parity flag is set ($P = 1$, even parity)
 RPO Return if Parity flag is reset ($P = 0$, odd parity)

9.4

ADVANCED SUBROUTINE CONCEPTS

In Section 9.2, one type of subroutine (multiple-calling of a subroutine by a main program) was illustrated. Other types of subroutine techniques, such as nesting and multiple-ending, are briefly illustrated below.

9.4.1 Nesting

The programming technique of a subroutine calling another subroutine is called **nesting**. This process is limited only by the number of available stack locations. When a subroutine calls another subroutine, all return addresses are stored on the stack. Nesting is illustrated in Figure 9.15.

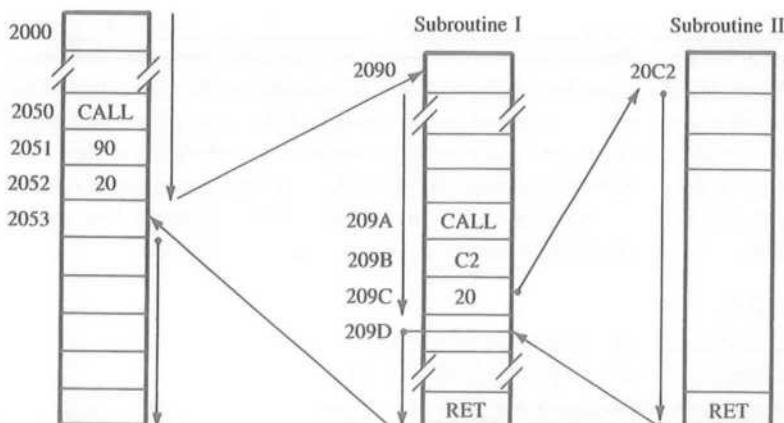
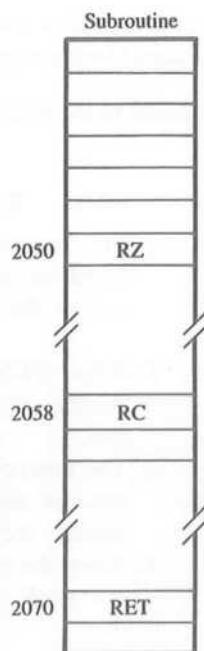


FIGURE 9.15
 Nesting of Subroutines

FIGURE 9.16
Multiple-Ending Subroutine



The main program in Figure 9.15 calls the subroutine from location 2050H. The address of the next instruction, 2053H, is placed on the stack, and the program is transferred to the subroutine at 2090H. Subroutine I calls Subroutine II from location 209AH. The address 209DH is placed on the stack, and the program is transferred to Subroutine II. The sequence of execution returns to the main program, as shown in Figure 9.15.

9.4.2 Multiple-Ending Subroutines

Figure 9.16 illustrates three possible endings to one CALL instruction. The subroutine has two conditional returns (RZ—Return on Zero, and RC—Return on Carry) and one unconditional return (RET). If the Zero flag (Z) is set, the subroutine returns from location 2050H. If the Carry flag (CY) is set, it returns from location 2058H. If neither the Z nor the CY flag is set, it returns from location 2070H. This technique is illustrated in Chapter 10, Section 10.4.2.

SUMMARY

To implement a subroutine, the following steps are necessary:

1. The stack pointer register must be initialized, preferably at the highest memory location of the R/W memory.

2. The CALL (or conditional Call) instruction should be used in the main program accompanied by the RET (or conditional Return) instruction in the subroutine.

The instructions CALL and RET are similar to the instructions PUSH and POP. The similarities and differences are as follows:

CALL and RET

1. When CALL is executed, the microprocessor automatically stores the 16-bit address of the instruction next to CALL on the stack.
2. When CALL is executed, the stack pointer register is decremented by two.
3. The instruction RET transfers the contents of the top two locations of the stack to the program counter.
4. When the instruction RET is executed, the stack pointer is incremented by two.
5. In addition to the unconditional CALL and RET instructions, there are eight conditional CALL and RETURN instructions.

PUSH and POP

1. The programmer uses the instruction PUSH to save the contents of a register pair on the stack.
2. When PUSH is executed, the stack pointer register is decremented by two.
3. The instruction POP transfers the contents of the top two locations of the stack to the specified register pair.
4. When the instruction POP is executed, the stack pointer is incremented by two.
5. There are no conditional PUSH and POP instructions.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

1. Check the appropriate answer in the following statements.
 - a. A stack is
 - (1) an 8-bit register in the microprocessor.
 - (2) a 16-bit register in the microprocessor.
 - (3) a set of memory locations in R/W/M reserved for storing information temporarily during the execution of a program.
 - (4) a 16-bit memory address stored in the program counter.
 - b. A stack pointer is
 - (1) a 16-bit register in the microprocessor that indicates the beginning of the stack memory.
 - (2) a register that decodes and executes 16-bit arithmetic expressions.
 - (3) the first memory location where a subroutine address is stored.
 - (4) a register in which flag bits are stored.

- c. When a subroutine is called, the address of the instruction following the CALL instruction is stored in/on the
 - (1) stack pointer.
 - (2) accumulator.
 - (3) program counter.
 - (4) stack.
 - d. When the RET instruction at the end of a subroutine is executed,
 - (1) the information where the stack is initialized is transferred to the stack pointer.
 - (2) the memory address of the RET instruction is transferred to the program counter.
 - (3) two data bytes stored in the top two locations of the stack are transferred to the program counter.
 - (4) two data bytes stored in the top two locations of the stack are transferred to the stack pointer.
 - e. Whenever the POP H instruction is executed,
 - (1) data bytes in the HL pair are stored on the stack.
 - (2) two data bytes at the top of the stack are transferred to the HL register pair.
 - (3) two data bytes at the top of the stack are transferred to the program counter.
 - (4) two data bytes from the HL register that were previously stored on the stack are transferred back to the HL register.
 - f. The instruction RST 7 is a:
 - (1) restart instruction that begins the execution of a program.
 - (2) one-byte call to the memory address 0038H.
 - (3) one-byte call to the memory address 0007H.
 - (4) hardware interrupt.
2. Read the following program and answer the questions given below.

Line No.	Mnemonics
1	LXI SP,0400H
2	LXI B,2055H
3	LXI H,22FFH
4	LXI D,2090H
5	PUSH H
6	PUSH B
7	MOV A,L
20	POP H

- a. What is stored in the stack pointer register after the execution of line 1?

- b. What is the memory location of the stack where the first data byte will be stored?
- c. What is stored in memory location 03FEH when line 5 (PUSH H) is executed?
- d. After the execution of line 6 (PUSH B), what is the address in the stack pointer register, and what is stored in stack memory location 03FDH?
- e. Specify the contents of register pair HL after the execution of line 20 (POP H).
- 3. The following program has a delay subroutine located at location 2060H. Read the program and answer the questions given at the end of the program.

Memory Locations	Mnemonics	
2000	LXI SP,20CDH	;Main Program
2003	LXI H,00008H	
2006	MVI B,0FH	
2008	CALL 2060H	
200B	OUT 01H	
	↓	
	DCR B	
	↓	
	CONTD	
2060	PUSH H	:Delay Subroutine
2061	PUSH B	
	↓	
	MVI B,05H	
	LXI H,COUNT	
	↓	
	POP B	
	POP H	
	RET	

- a. When the execution of the CALL instruction located at 2008H–200AH is completed, list the contents stored at 20CCH and 20CBH, the contents of the program counter, and the contents of the stack pointer register.
- b. List the stack locations and their contents after the execution of the instructions PUSH H and PUSH B in the subroutine.
- c. List the contents of the stack pointer register after the execution of the instruction PUSH B located at 2061H.
- d. List the contents of the stack pointer register after the execution of the instruction RET in the subroutine.
- 4. Explain the functions of the following routines:
 - a. LXI SP,209FH b. LXI SP,STACK
 - MVI C,00H PUSH B
 - PUSH B PUSH D
 - POP PSW POP B
 - RET POP D
 - RET

5. Read the following program and answer the questions.

2000 LXI SP,2100H	DELAY: 2064 PUSH H
2003 LXI B,0000H	2065 PUSH B
2006 PUSH B	2066 LXI B,80FFH
2007 POP PSW	LOOP: 2069 DCX B
2008 LXI H,200BH	206A MOV A,B
200B CALL 2064H	206B ORA C
200E OUT 01H	206C JNZ LOOP
2010 HLT	206F POP B
	2070 RET

- a. What is the status of the flags and the contents of the accumulator after the execution of the POP instruction located at 2007H?
- b. Specify the stack locations and their contents after the execution of the CALL instruction (not the Call subroutine).
- c. What are the contents of the stack pointer register and the program counter after the execution of the CALL instruction?
- d. Specify the memory location where the program returns after the subroutine.
- e. What is the ultimate fate of this program?
6. Write a program to add the two Hex numbers 7A and 46 and to store the sum at memory location XX98H and the flag status at location XX97H.
7. In Assignment 6, display the sum and the flag status at two different output ports.
8. Write a program to meet the following specifications:
 - a. Initialize the stack pointer register at XX99H.
 - b. Clear the memory locations starting from XX90H to XX9FH.
 - c. Load register pairs B, D, and H with data 0237H, 1242H, and 4087H, respectively.
 - d. Push the contents of the register pairs B, D, and H on the stack.
 - e. Execute the program and verify the memory locations from XX90H to XX9FH.
9. Write a program to clear the initial flags. Load data byte FFH into the accumulator and add 01H to the byte FFH by using the instruction ADI. Mask all the flags except the CY flag and display the CY flag at PORT0 (or store the results on the stack). Repeat the program by replacing the ADI instruction with the INR instruction and the byte 01H with the NOP instruction. Display the flag at PORT1 (or store the results on the stack). Explain the results.
10. Write a 20 ms time delay subroutine using register pair BC. Clear the Z flag without affecting any other flags in the flag register and return to the main program.
11. Write a program to control a railway crossing signal that has two alternately flashing red lights, with a 1-second delay on time for each light.
12. Write a program to simulate a flashing yellow light with 750 ms on time. Use bit D₇ to control the light. (*Hint:* To simulate flashing, the light must be turned off.)

10

Code Conversion, BCD Arithmetic, and 16-Bit Data Operations

In microcomputer applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) keyboard is a commonly used input device for disk-based microcomputer systems. Similarly, alphanumeric characters (letters and numbers) are displayed on a CRT (cathode ray tube) terminal using the ASCII code. However, inside the microprocessor, data processing is usually performed in binary. In some instances, arithmetic operations are performed in BCD numbers. Therefore, data must be converted from one code to another code. The programming techniques used for code conversion fall into four general categories:

1. Conversion based on the position of a digit in a number (BCD to binary and vice versa).
2. Conversion based on hardware consideration (binary to seven-segment code using table look-up procedure).

3. Conversion based on sequential order of digits (binary to ASCII and vice versa).
4. Decimal adjustment in BCD arithmetic operations. (This is an adjustment rather than a code conversion.)

This chapter discusses these techniques with various examples written as subroutines. The subroutines are written to demonstrate industrial practices in writing software, and can be verified on single-board microcomputers. In addition, instructions related to 16-bit data operations are introduced and illustrated.

OBJECTIVES

Write programs and subroutines to

- Convert a packed BCD number (0–99) into its binary equivalent.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.

- Select an appropriate seven-segment code for a given binary number using the table look-up technique.
- Convert a binary digit (0 to F) into its ASCII Hex code and vice versa.
- Decimal-adjust 8-bit BCD addition and subtraction.
- Perform such arithmetic operations as multiplication and subtraction using 16-bit data related instructions.
- Demonstrate uses of instructions such as DAD, PCHL, XTHL, and XCHG.

10.1 BCD-TO-BINARY CONVERSION

In most microprocessor-based products, data are entered and displayed in decimal numbers. For example, in an instrumentation laboratory, readings such as voltage and current are maintained in decimal numbers, and data are entered through a decimal keyboard. The system-monitor program of the instrument converts each key into an equivalent 4-bit binary number and stores two BCD numbers in an 8-bit register or a memory location. These numbers are called **packed BCD**. Even if data are entered in decimal digits, it is inefficient to process data in BCD numbers because, in each 4-bit combination, digits A through F are unused. Therefore, BCD numbers are generally converted into binary numbers for data processing.

The conversion of a BCD number into its binary equivalent employs the principle of *positional weighting* in a given number.

For example: $72_{10} = 7 \times 10 + 2$.

The digit 7 represents 70, based on its second position from the right. Therefore, converting 72_{BCD} into its binary equivalent requires multiplying the second digit by 10 and adding the first digit.

Converting a 2-digit BCD number into its binary equivalent requires the following steps:

1. Separate an 8-bit packed BCD number into two 4-bit unpacked BCD digits: BCD_1 and BCD_2 .
2. Convert each digit into its binary value according to its position.
3. Add both binary numbers to obtain the binary equivalent of the BCD number.

Example
10.1

Convert 72_{BCD} into its binary equivalent.

Solution

$$72_{10} = 0111\ 0010_{BCD}$$

Step 1: $0111\ 0010 \rightarrow 0000\ 0010$ Unpacked BCD_1
 $\rightarrow 0000\ 0111$ Unpacked BCD_2

Step 2: Multiply BCD_2 by 10 (7×10)

Step 3: Add BCD_1 to the answer in Step 2

The multiplication of BCD_2 by 10 can be performed by various methods. One method is multiplication with repeated addition: add 10 seven times. This technique is illustrated in the next program.

10.1.1 Illustrative Program: 2-Digit BCD-to-Binary Conversion

PROBLEM STATEMENT

A BCD number between 0 and 99 is stored in an R/W memory location called the **Input Buffer (INBUF)**. Write a main program and a conversion subroutine (BCDBIN) to convert the BCD number into its equivalent binary number. Store the result in a memory location defined as the **Output Buffer (OUTBUF)**.

PROGRAM

START:	LXI SP,STACK	;Initialize stack pointer
	LXI H,INBUF	;Point HL index to the Input Buffer memory location where BCD
		; number is stored
	LXI B,OUTBUF	;Point BC index to the Output Buffer memory where binary
		; number will be stored
	MOV A,M	;Get BCD number
	CALL BCDBIN	;Call BCD to binary conversion routine
	STAX B	;Store binary number in the Output Buffer
	HLT	;End of program

BCDBIN: ;Function: This subroutine converts a BCD number into its binary equivalent
;Input: A 2-digit packed BCD number in the accumulator
;Output: A binary number in the accumulator
;No other register contents are destroyed

Example: Assume BCD number is 72:			
PUSH B	;Save BC registers	A	0111 0010 → 72 ₁₀
PUSH D	;Save DE registers	B	0111 0010 → 72 ₁₀
MOV B,A	;Save BCD number	A	0000 0010 → 02 ₁₀
ANI 0FH	;Mask most significant four bits	C	0000 0010 → 02 ₁₀
MOV C,A	;Save unpacked BCD ₁ in C	A	0111 0010 → 72 ₁₀
MOV A,B	;Get BCD again	A	0111 0000 → 70M ₁₀
ANI F0H	;Mask least significant four bits		
JZ BCD ₁	;If BCD ₂ = 0, the result is only BCD ₂		
RRC	;Convert most significant four		
RRC	; bits into unpacked BCD ₂		
RRC			
RRC			
MOV D,A	;Save BCD ₂ in D	A	0000 0111 → 07 ₁₀
XRA A	;Clear accumulator	D	0000 0111 → 07 ₁₀

SUM:	MVI E,0AH ;Set E as multiplier of 10 ADD E ;Add 10 until (D) = 0 DCR D ;Reduce BCD_2 by one JNZ SUM ;Is multiplication complete? ;If not, go back and add again	E 0000 1010 → 0AH Add E as many times as (D) After adding E seven times A contains: 0100 0110 C +0000 0010 A 0100 1000 → 48H
BCD1:	ADD C ;Add BCD1 POP D ;Retrieve previous contents POP B RET	

PROGRAM DESCRIPTION

1. In writing assembly language programs, the use of labels is a common practice. Rather than writing a specific memory location or a port number, a programmer uses such labels as INBUF (Input Buffer) and OUTBUF (Output Buffer). Using labels gives flexibility and ease of documentation.
2. The main program initializes the stack pointer and two memory indexes. It brings the BCD number into the accumulator and passes that parameter to the subroutine.
3. After returning from the subroutine, the main program stores the binary equivalent in the Output Buffer memory.
4. The subroutine saves the contents of the BC and DE registers because these registers are used in the subroutine. Even if this particular main program does not use the DE registers, the subroutine may be called by some other program in which the DE registers are being used. Therefore, it is a good practice to save the registers that are used in the subroutine, unless parameters are passed to the subroutine. The accumulator contents are not saved because that information is passed on to the subroutine.
5. The conversion from BCD to binary is illustrated in the subroutine with the example of 72_{BCD} converted to binary.

The illustrated multiplication routine is easy to understand; however, it is rather long and inefficient. Another method is to multiply BCD_2 by shifting, as illustrated in Assignments 3 and 4 at the end of this chapter.

PROGRAM EXECUTION

To execute the program on a single-board computer, complete the following steps:

1. Assign memory addresses to the instructions in the main program and in the subroutine. Both can be assigned consecutive memory addresses.
2. Define STACK: the stack location with a 16-bit address in the R/W memory (such as 2099H).
3. Define INBUF (Input Buffer) and OUTBUF (Output Buffer): two memory locations in the R/W memory (e.g., 2050H and 2060H).
4. Enter a BCD byte in the Input Buffer (e.g., 2050H).
5. Enter and execute the program.

6. Check the contents of the Output Buffer memory location (2060H) and verify your answer.

See Assignments 1 through 4 at the end of this chapter.

BINARY-TO-BCD CONVERSION

10.2

In most microprocessor-based products, numbers are displayed in decimal. However, if data processing inside the microprocessor is performed in binary, it is necessary to convert the binary results into their equivalent BCD numbers just before they are displayed. Results are quite often stored in R/W memory locations called the **Output Buffer**.

The conversion of binary to BCD is performed by dividing the number by the powers of ten; the division is performed by the subtraction method.

For example, assume the binary number is

$$1\ 1\ 1\ 1\ 1\ 1\ 1_2 \text{ (FFH)} = 255_{10}$$

To represent this number in BCD requires twelve bits or three BCD digits, labeled here as BCD_3 (MSB), BCD_2 , and BCD_1 (LSB),

$$= 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1 \\ \text{BCD}_3\quad \text{BCD}_2\quad \text{BCD}_1$$

The conversion can be performed as follows:

Step 1: If the number is less than 100, go to Step 2; otherwise, divide by 100 or subtract 100 repeatedly until the remainder is less than 100. The quotient is the most significant BCD digit, BCD_3 .

Step 2: If the number is less than 10, go to Step 3; otherwise divide by 10 repeatedly until the remainder is less than 10. The quotient is BCD_2 .

Step 3: The remainder from Step 2 is BCD_1 .

	Example	Quotient
255		
-100	= 155	1
-100	= 55	1
	$\text{BCD}_3 = 2$	
55		
-10	= 45	1
-10	= 35	1
-10	= 25	1
-10	= 15	1
-10	= 05	1
	$\text{BCD}_2 = 5$	
	$\text{BCD}_1 = 5$	

These steps can be converted into a program as illustrated next.

10.2.1 Illustrative Program: Binary-to-Unpacked-BCD Conversion

PROBLEM STATEMENT

A binary number is stored in memory location BINBYT. Convert the number into BCD, and store each BCD as two unpacked BCD digits in the Output Buffer. To perform this task, write a main program and two subroutines: one to supply the powers of ten, and the other to perform the conversion.

PROGRAM

This program converts an 8-bit binary number into a BCD number; thus it requires 12 bits to represent three BCD digits. The result is stored as three unpacked BCD digits in three Output-Buffer memory locations.

START:	LXI SP,STACK ;Initialize stack pointer
	LXI H,BINBYT ;Point HL index where binary number is stored
	MOV A,M ;Transfer byte
	CALL PWRTEM ;Call subroutine to load powers of 10
	HLT
PWRTEM:	;this subroutine loads the powers of 10 in register B and calls the binary-to-BCD ; conversion routine
	;Input: Binary number in the accumulator
	;Output: Powers of ten and stores BCD ₁ in the first Output-Buffer memory
	;Calls BINBCD routine and modifies register B
	LXI H,OUTBUF ;Point HL index to Output-Buffer memory
	MVI B,64H ;Load 100 in register B
	CALL BINBCD ;Call conversion
	MVI B,0AH ;Load 10 in register B
	CALL BINBCD
	MOV M,A ;Store BCD ₁
	RET
BINBCD:	;This subroutine converts a binary number into BCD and stores BCD ₂ and BCD ₃ in the ; Output Buffer.
	;Input: Binary number in accumulator and powers of 10 in B
	;Output: BCD ₂ and BCD ₃ in Output Buffer
	;Modifies accumulator contents
NXTBUF:	MVI M,FFH ;Load buffer with (0 – 1)
	INR M ;Clear buffer and increment for each subtraction
	SUB B ;Subtract power of 10 from binary number
	JNC NXTBUF ;Is number > power of 10? If yes, add 1 to buffer memory
	ADD B ;If no, add power of 10 to get back remainder

```
INX H      ;Go to next buffer location
RET
```

PROGRAM DESCRIPTION

This program illustrates the concepts of the **nested subroutine** and the **multiple-call subroutine**. The main program calls the PWRTEM subroutine; in turn, the PWRTEM calls the BINBCD subroutine twice.

1. The main program transfers the byte to be converted to the accumulator and calls the PWRTEM subroutine.
2. The subroutine PWRTEM supplies the powers of ten by loading register B and the address of the first Output-Buffer memory location, and calls conversion routine BINBCD.
3. In the BINBCD conversion routine, the Output-Buffer memory is used as a register. It is incremented for each subtraction loop. This step also can be achieved by using a register in the microprocessor. The BINBCD subroutine is called twice, once after loading register B with 64H (100_{10}), and again after loading register B with 0AH (10_{10}).
4. During the first call of BINBCD, the subroutine clears the Output Buffer, stores BCD_3 , and points the HL registers to the next Output-Buffer location. The instruction ADD B is necessary to restore the remainder because one extra subtraction is performed to check the borrow.
5. During the second call of BINBCD, the subroutine again clears the output buffer, stores BCD_2 , and points to the next buffer location. BCD_3 is already in the accumulator after the ADD instruction, which is stored in the third Output-Buffer memory by the instruction MOV M,A in the PWRTEM subroutine.

This is an efficient subroutine; it combines the functions of storing the answer and finding a quotient. However, two subroutines are required, and the second subroutine is called twice for a conversion.

See Assignments 5 through 8 at the end of this chapter.

BCD-TO-SEVEN-SEGMENT-LED CODE CONVERSION

10.3

When a BCD number is to be displayed by a seven-segment LED, it is necessary to convert the BCD number to its seven-segment code. The code is determined by hardware considerations such as common-cathode or common-anode LED; the code has no direct relationship to binary numbers. Therefore, to display a BCD digit at a seven-segment LED, the **table look-up technique** is used.

In the table look-up technique, the codes of the digits to be displayed are stored sequentially in memory. The conversion program locates the code of a digit based on its

magnitude and transfers the code to the MPU to send out to a display port. The table look-up technique is illustrated in the next program.

10.3.1 Illustrative Program: BCD-to-Common-Cathode-LED Code Conversion

PROBLEM STATEMENT

A set of three packed BCD numbers (six digits) representing time and temperature are stored in memory locations starting at XX50H. The seven-segment codes of the digits 0 to 9 for a common-cathode LED are stored in memory locations starting at XX70H, and the Output-Buffer memory is reserved at XX90H.

Write a main program and two subroutines, called UNPAK and LEDCOD, to unpack the BCD numbers and select an appropriate seven-segment code for each digit. The codes should be stored in the Output-Buffer memory.

PROGRAM

```

LXI SP,STACK      ;Initialize stack pointer
LXI H,XX50H      ;Point HL where BCD digits are stored
MVI D,03H        ;Number of digits to be converted is placed in D
CALL UNPAK       ;Call subroutine to unpack BCD numbers
HLT              ;End of conversion

UNPAK:           ;This subroutine unpacks the BCD number in two single digits
;Input: Starting memory address of the packed BCD numbers in HL registers
;       Number of BCDs to be converted in register D
;Output: Unpacked BCD into accumulator and output
;       Buffer address in BC
;Calls subroutine LEDCOD

LXI B,BUFFER     ;Point BC index to the buffer memory
NXTBCD:          MOV A,M      ;Get packed BCD number
                 ANI F0H      ;Masked BCD1
                 RRC         ;Rotate four times to place BCD2 as unpacked single-digit BCD
                 RRC
                 RRC
                 RRC
                 CALL LEDCOD   ;Find seven-segment code
                 INX B       ;Point to next buffer location
                 MOV A,M      ;Get BCD number again
                 ANI 0FH      ;Separate BCD1
                 CALL LEDCOD   ;Find seven-segment code
                 INX B       ;Point to next BCD
                 DCR D       ;One conversion complete, reduce BCD count
                 JNZ NXTBCD   ;If all BCDs are not yet converted, go back to convert next BCD
                 RET

```

LEDCOD: ;This subroutine converts an unpacked BCD into its seven-segment-LED code
;Input: An unpacked BCD in accumulator
;Memory address of the buffer in BC register
;Output: Stores seven-segment code in the output buffer

PUSH H	:Save HL contents of the caller
LXI H,CODE	:Point index to beginning of seven-segment code
ADD L	:Add BCD digit to starting address of the code
MOV L,A	:Point HL to appropriate code
MOV A,M	:Get seven-segment code
STAX B	:Store code in buffer
POP H	
RET	

CODE: 3F :Digit 0: Common-cathode codes
06 :Digit 1
5B :Digit 2
4F :Digit 3
66 :Digit 4
6D :Digit 5
7D :Digit 6
07 :Digit 7
7F :Digit 8
6F :Digit 9
00 :Invalid Digit

PROGRAM DESCRIPTION/OUTPUT

1. The main program initializes the stack pointer, the HL register as a pointer for BCD digits, and the counter for the number of digits; then it calls the UNPAK subroutine.
2. The UNPAK subroutine transfers a BCD number into the accumulator and unpacks it into two BCD digits by using the instructions ANI and RRC. This subroutine also supplies the address of the buffer memory to the next subroutine, LEDCOD. The subroutine is repeated until counter D becomes zero.
3. The LEDCOD subroutine saves the memory address of the BCD number and points the HL register to the beginning address of the code.
4. The instruction ADD L adds the BCD digit in the accumulator to the starting address of the code. After storing the sum in register L, the HL register points to the seven-segment code of that BCD digit.
5. The code is transferred to the accumulator and stored in the buffer.

This illustrative program uses the technique of the nested subroutine (one subroutine calling another). Parameters are passed from one subroutine to another; therefore, you should be careful in using Push instructions to store register contents on the stack. In

addition, the LEDCOD subroutine does not account for a situation if by adding the register L a carry is generated. (See Assignment 12.)

See Assignments 9–12 at the end of this chapter.

10.4

BINARY-TO-ASCII AND ASCII-TO-BINARY CODE CONVERSION

The American Standard Code for Information Interchange (known as ASCII) is used commonly in data communication. It is a seven-bit code, and its 128 (2^7) combinations are assigned different alphanumeric characters (see Appendix E). For example, the hexadecimal numbers 30H to 39H represent 0 to 9 ASCII decimal numbers, and 41H to 5AH represent capital letters A through Z; in this code, bit D₇ is zero. In serial data communication, bit D₇ can be used for parity checking (see Chapter 16, Serial I/O and Data Communication).

The ASCII keyboard is a standard input device for entering programs in a microcomputer. When an ASCII character is entered, the microprocessor receives the binary equivalent of the ASCII Hex number. For example, when the ASCII key for digit 9 is pressed, the microprocessor receives the binary equivalent of 39H, which must be converted to the binary 1001 for arithmetic operations. Similarly, to display digit 9 at the terminal, the microprocessor must send out the ASCII Hex code (39H). These conversions are done through software, as in the following illustrative program.

10.4.1 Illustrative Program: Binary-to-ASCII Hex Code Conversion

PROBLEM STATEMENT

An 8-bit binary number (e.g., 9FH) is stored in memory location XX50H.

1. Write a program to
 - a. Transfer the byte to the accumulator.
 - b. Separate the two nibbles (as 09 and 0F).
 - c. Call the subroutine to convert each nibble into ASCII Hex code.
 - d. Store the codes in memory locations XX60H and XX61H.
2. Write a subroutine to convert a binary digit (0 to F) into ASCII Hex code.

MAIN PROGRAM

```
LXI SP,STACK      ;Initialize stack pointer
LXI H,XX50H       ;Point index where binary number is stored
LXI D,XX60H       ;Point index where ASCII code is to be stored
MOV A,M           ;Transfer byte
MOV B,A           ;Save byte
RRC               ;Shift high-order nibble to the position of low-order
RRC               ; nibble
```

```

    RRC
    RRC
    CALL ASCII      ;Call conversion routine
    STAX D          ;Store first ASCII Hex in XX60H
    INX D          ;Point to next memory location, get ready to store
                  ; next byte
    MOV A,B          ;Get number again for second digit
    CALL ASCII
    STAX D
    HLT

ASCII:   ;This subroutine converts a binary digit between 0 and F to ASCII Hex
        ; code
        ;Input: Single binary number 0 to F in the accumulator
        ;Output: ASCII Hex code in the accumulator
    ANI 0FH          ;Mask high-order nibble
    CPI 0AH          ;Is digit less than 1010?
    JC CODE          ;If digit is less than 1010, go to CODE to add 30H
    ADI 07H          ;Add 7H to obtain code for digits from A to F
CODE:   ADI 30H          ;Add base number 30H
        RET

```

PROGRAM DESCRIPTION

1. The main program transfers the binary data byte from the memory location to the accumulator.
2. It shifts the high-order nibble into the low-order nibble, calls the conversion subroutine, and stores the converted value in the memory.
3. It retrieves the byte again and repeats the conversion process for the low-order nibble.

In this program, the masking instruction ANI is used once in the subroutine rather than twice in the main program as illustrated in the program for BCD-to-Common-Cathode-LED Code Conversion (Section 10.3.1).

10.4.2 Illustrative Program: ASCII Hex-to-Binary Conversion

PROBLEM STATEMENT

Write a subroutine to convert an ASCII Hex number into its binary equivalent. A calling program places the ASCII number in the accumulator, and the subroutine should pass the conversion back to the accumulator.

SUBROUTINE

```

ASCBIN:  ;This subroutine converts an ASCII Hex number into its binary
        ; equivalent
        ;Input: ASCII Hex number in the accumulator

```

;Output: Binary equivalent in the accumulator

SUI 30H	;Subtract 0 bias from the number
CPI 0AH	;Check whether number is between 0 and 9
RC	;If yes, return to main program
SUI 07H	;If not, subtract 7 to find number between A and F
RET	

PROGRAM DESCRIPTION

This subroutine subtracts the ASCII weighting digits from the number. This process is exactly opposite to that of the Illustrative Program that converted binary into ASCII Hex (Section 10.4.1). However, this program uses two return instructions, an illustration of the multiple-ending subroutine.

See Assignments 13 and 14 at the end of this chapter.

10.5 BCD ADDITION

In some applications, input/output data are presented in decimal numbers, and the speed of data processing is unimportant. In such applications, it may be convenient to perform arithmetic operations directly in BCD numbers. However, the addition of two BCD numbers may not represent an appropriate BCD value. For example, the addition of 34_{BCD} and 26_{BCD} results in $5AH$, as shown below:

$$\begin{array}{r} 34_{10} = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0_{BCD} \\ + 26_{10} = 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0_{BCD} \\ \hline 60_{10} = 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0 \rightarrow 5AH \end{array}$$

The microprocessor cannot recognize BCD numbers; it adds any two numbers in binary. In BCD addition, any number larger than 9 (from A to F) is invalid and needs to be adjusted by adding 6 in binary. For example, after 9, the next BCD number is 10; however, in Hex it is A. The Hex number A can be adjusted as a BCD number by adding 6 in binary. The BCD adjustment in an 8-bit binary register can be shown as follows:

$$\begin{array}{r} A = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 \\ + 6 = 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 \\ \hline 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \rightarrow 10_{BCD} \end{array}$$

Any BCD sum can be adjusted to proper BCD value by adding 6 when the sum exceeds 9. In case of packed BCD, both BCD_1 and BCD_2 need to be adjusted; if a carry is generated by adding 6 to BCD_1 , the carry should be added to BCD_2 , as shown in the following example.

Add two packed BCD numbers: 77 and 48.

Example
10.2

Addition:

$$\begin{array}{r}
 77 = 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1 \\
 + 48 = 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \\
 \hline
 125 = 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \\
 \quad \quad \quad + 0\ 1\ 1\ 0 \\
 \hline
 \text{CY} \boxed{1} \quad 0\ 1\ 0\ 1
 \end{array}$$

Solution

The value of the least significant four bits is larger than 9. Add 6.

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

The value of the most significant four bits is larger than 9. Add 6 and the carry from the previous adjustment.

In this example, the carry is generated after the adjustment of the least significant four bits for the BCD digit and is again added to the adjustment of the most significant four bits.

A special instruction called DAA (Decimal Adjust Accumulator) performs the function of adjusting a BCD sum in the 8085 instruction set. This instruction uses the Auxiliary Carry flip-flop (AC) to sense that the value of the least four bits is larger than 9 and adjusts the bits to the BCD value. Similarly, it uses the Carry flag (CY) to adjust the most significant four bits. However, the AC flag is used internally by the microprocessor; this flag is not available to the programmer through any Jump instruction.

INSTRUCTION

DAA: Decimal Adjust Accumulator

- This is a 1-byte instruction
- It adjusts an 8-bit number in the accumulator to form two BCD numbers by using the process described above
- It uses the AC and the CY flags to perform the adjustment
- All flags are affected

It must be emphasized that instruction DAA

- adjusts a BCD sum.
- does not convert a binary number into BCD numbers.
- works only with addition when BCD numbers are used; does not work with subtraction.

10.5.1 Illustrative Program: Addition of Unsigned BCD Numbers

PROBLEM STATEMENT

A set of ten packed BCD numbers is stored in the memory location starting at XX50H.

1. Write a program with a subroutine to add these numbers in BCD. If a carry is generated, save it in register B, and adjust it for BCD. The final sum will be less than 9999_{BCD}.

2. Write a second subroutine to unpack the BCD sum stored in registers A and B, and store them in the output-buffer memory starting at XX60H. The most significant digit (BCD_4) should be stored at XX60H and the least significant digit (BCD_1) at XX63H.

PROGRAM

```

START:    LXI SP,STACK      ;Initialize stack pointer
          LXI H,XX50H      ;Point index to XX50H
          MVI C,COUNT      ;Load register C with the count of BCD numbers to be added
          XRA A            ;Clear accumulator
          MOV B,A            ;Clear register B to save carry
NXTBCD:   CALL BCDADD      ;Call subroutine to add BCD numbers
          INX H            ;Point to next memory location
          DCR C            ;One addition of BCD number is complete, decrement the counter
          JNZ NXTBCD        ;If all numbers are added go to next step, otherwise go back
          LXI H,XX63H      ;Point index to store  $BCD_1$  first
          CALL UNPAK        ;Unpack the BCD stored in the accumulator
          MOV A,B            ;Get ready to store high-order BCD— $BCD_3$  and  $BCD_4$ 
          CALL UNPAK        ;Unpack and store  $BCD_3$  and  $BCD_4$  at XX61H and XX60
          HLT

BCDADD:   ;This subroutine adds the BCD number from the memory to the accumulator and decimal-
          ; adjusts it. If the sum is larger than eight bits, it saves the carry and decimal-adjusts the
          ; carry sum
          ;Input: The memory address in HL register where the BCD number is stored
          ;Output: Decimal-adjusted BCD number in the accumulator and the carry in register B

          ADD M            ;Add packed BCD byte and adjust it for BCD sum
          DAA
          RNC              ;If no carry, go back to next BCD
          MOV D,A            ;If carry is generated, save the sum from the accumulator
          MOV A,B            ;Transfer CY sum from register B and add 01
          ADI 01H
          DAA              ;Decimal-adjust BCD from B
          MOV B,A            ;Save adjusted BCD in B
          MOV A,D            ;Place  $BCD_1$  and  $BCD_2$  in the accumulator
          RET

UNPAK:   ;This subroutine unpacks the BCD in the accumulator and the carry register and stores
          ; them in the output buffer
          ;Input: BCD number in the accumulator, and the buffer address in HL registers
          ;Output: Unpacked BCD in the output buffer

          MOV D,A            ;Save BCD number
          ANI 0FH           ;Mask high-order BCD

```

```

MOV M,A      ;Store low-order BCD
DCX H        ;Point to next memory location
MOV A,D      ;Get BCD again
ANI F0H      ;Mask low-order BCD
RRC          ;Convert the most significant four bits into unpacked BCD
RRC
RRC
RRC
MOV M,A      ;Store high-order BCD
DCX H        ;Point to the next memory location
RET

```

PROGRAM DESCRIPTION

1. The expected maximum sum is 9090, which requires two registers. The main program clears the accumulator to save BCD_1 and BCD_2 , clears register B to save BCD_3 and BCD_4 , and calls the subroutine to add the numbers. The BCD bytes are added until the counter C becomes zero.
2. The BCDADD subroutine is an illustration of the multiple-ending subroutine. It adds a byte, decimal-adjusts the accumulator and, if there is no carry, returns the program execution to the main program. If there is a carry, it adds 01 to the carry register B by transferring the contents to the accumulator and decimal-adjusting the contents. The final sum is stored in registers A and B.
3. The main program calls the UNPAK subroutine, which takes the BCD number from the accumulator (e.g., 57_{BCD}), unpacks it into two separate BCDs (e.g., 05_{BCD} and 07_{BCD}), and stores them in the output buffer. When a subroutine stores a BCD number in memory, it decrements the index because BCD_1 is stored first.

See Assignments 15–16 at the end of this chapter.

BCD SUBTRACTION

10.6

When subtracting two BCD numbers, the instruction DAA cannot be used to decimal-adjust the result of two packed BCD numbers; the instruction applies only to addition. Therefore, it is necessary to devise a procedure to subtract two BCD numbers. Two BCD numbers can be subtracted by using the procedure of 100's complement (also known as 10's complement), similar to 2's complement. The 100's complement of a subtrahend can be added to a minuend as illustrated:

For example, $82 - 48 (= 34)$ can be performed as follows:

100's complement of subtrahend	52	(100 – 48 = 52)
Add minuend	+ 82	
	1/34	

The sum is 34 if the carry is ignored. This is similar to subtraction by 2's complement. However, in an 8-bit microprocessor, it is not a simple process to find 100's complement of a subtrahend (100_{BCD} requires twelve bits). Therefore, in writing a program, 100's complement is obtained by finding 99's complement and adding 01.

10.6.1 Illustrative Program: Subtraction of Two Packed BCD Numbers

PROBLEM STATEMENT

Write a subroutine to subtract one packed BCD number from another BCD number. The minuend is placed in register B, and the subtrahend is placed in register C by the calling program. Return the answer into the accumulator.

SUBROUTINE

```
SUBBCD:    ;This subroutine subtracts two BCD numbers and adjusts the result to
           ;   BCD values by using the 100's complement method
           ;Input: A minuend in register B and a subtrahend in register C
           ;Output: The result is placed in the accumulator
           MVI A,99H
           SUB C      ;Find 99's complement of subtrahend
           INR A      ;Find 100's complement of subtrahend
           ADD B      ;Add minuend to 100's complement of subtrahend
           DAA        ;Adjust for BCD
           RET
```

See Assignments 17 and 18 at the end of this chapter.

10.7

INTRODUCTION TO ADVANCED INSTRUCTIONS AND APPLICATIONS

The instructions discussed in the last several chapters deal primarily with 8-bit data (except LXI). However, in some instances data larger than eight bits must be manipulated, especially in arithmetic manipulations and stack operations. Even if the 8085 is an 8-bit microprocessor, its architecture allows specific combinations of two 8-bit registers to form 16-bit registers. Several instructions in the instruction set are available to manipulate 16-bit data. These instructions will be introduced in this section.

10.7.1 16-Bit Data Transfer (Copy) and Data Exchange Group

- LHLD: Load HL registers direct
- This is a 3-byte instruction
 - The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number)

- Transfers the contents of the specified memory location to L register
 Transfers the contents of the next memory location to H register
SHLD: Store HL registers direct
 This is a 3-byte instruction
 The second and third bytes specify a memory location (the second byte is a line number and the third byte is a page number)
 Stores the contents of L register in the specified memory location
 Stores the contents of H register in the next memory location
XCHG: Exchange the contents of HL and DE
 This is a 1-byte instruction
 The contents of H register are exchanged with the contents of D register, and the contents of L register are exchanged with the contents of E register

Memory locations 2050H and 2051H contain 3FH and 42H, respectively, and register pair DE contains 856FH. Write instructions to exchange the contents of DE with the contents of the memory locations.

Example
10.3

Memory			
Before Instructions:	D	85 6F	E
		3F	2050
		42	2051

Instructions

Machine		Mnemonics	
Code			
2A	LHLD 2050H		3F 2050
50			42 2051
20		H 42 3F L	
EB	XCHG	D 42 3F E	3F 2050
		H 85 6F L	42 2051
22	SHLD 2050H		6F 2050
50			85 2051
20		H 85 6F L	

10.7.2 Arithmetic Group

Operation: Addition with Carry

- ADC R These instructions add the contents of the operand, the carry, and the accumulator. All flags are affected
 ADC M
 ACI 8-bit

**Example
10.4**

Registers BC contain 2793H, and registers DE contain 3182H. Write instructions to add these two 16-bit numbers, and place the sum in memory locations 2050H and 2051H.

Before instructions:	B	27	93	C
	D	31	82	E

Instructions

MOV A,C	A	93	F	93H
ADD E	A	15	F	+ 82H
MOV L,A	H	15	L	1/15H
MOV A,B				27H
ADC D				+ 31H
MOV H,A	H	59	L	59H
SHLD 2050H				

Operation: Subtraction with Carry

- SBB R These instructions subtract the contents of the operand and the borrow from the contents of the accumulator
- SBB M
- SBI 8-bit

**Example
10.5**

Registers BC contain 8538H and registers DE contain 62A5H. Write instructions to subtract the contents of DE from the contents of BC, and place the result in BC.

Instructions

MOV A,C	(B)	85	38	(C)
SUB E			-	
MOV C,A	(D)	62	A5	(E)
MOV A,B		-1	1/93	
SBB D	(B)	22	93	(C)
MOV B,A				

Operation: Double Register ADD

- DAD Rp Add register pair to register HL
 - This is a 1-byte instruction
- DAD B Adds the contents of the operand (register pair or stack pointer) to the contents of HL registers
- DAD D
- DAD H The result is placed in HL registers
- DAD SP The Carry flag is altered to reflect the result of the 16-bit addition. No other flags are affected
 - The instruction set includes four instructions

Write instructions to display the contents of the stack pointer register at output ports.

Example
10.6

Instructions

LXI H,0000H	;Clear HL
DAD SP	;Place the stack pointer contents in HL
MOV A,H	;Place high-order address of the stack pointer in the accumulator
OUT PORT1	
MOV A,L	;Place low-order address of the stack pointer in the accumulator
OUT PORT2	

The instruction DAD SP adds the contents of the stack pointer register to the HL register pair, which is already cleared. This is the only instruction in the 8085 that enables the programmer to examine the contents of the stack pointer register.

10.7.3 Instructions Related to the Stack Pointer and the Program Counter

XTHL: Exchange Top of the Stack with H and L

- The contents of L are exchanged with the contents of the memory location shown by the stack pointer, and the contents of H are exchanged with the contents of memory location of the stack pointer +1.

Write a subroutine to set the Zero flag and check whether the instruction JZ (Jump on Zero) functions properly, without modifying any register contents other than flags.

Example
10.7

Subroutine

CHECK:	PUSH H	
	MVI L,FFH	;Set all bits in L to logic 1
	PUSH PSW	;Save flags on the top of the stack
	XTHL	;Set all bits in the top stack location
	POP PSW	;Set Zero flag
	JZ NOEROR	
	JMP ERROR	
NOEROR:	POP H	
	RET	

The instruction PUSH PSW places the flags in the top location of the stack, and the instruction XTHL changes all the bits in that location to logic 1. The instruction POP PSW sets all the flags. If the instruction JZ is functioning properly, the routine returns to the calling program; otherwise, it goes to the ERROR routine (not shown). This example shows that the flags can be examined, and they can be set or reset to check malfunctions in the instructions.

SPHL: Copy H and L registers into the Stack Pointer Register

- The contents of H specify the high-order byte and the contents of L specify the low-order byte
- The contents of HL registers are not affected

This instruction can be used to load a new address in the stack pointer register. (For an example, see SPHL in the instruction set, Appendix F.)

PCHL: Copy H and L registers into the Program Counter

- The contents of H specify the high-order byte and the contents of L specify the low-order byte

**Example
10.8**

Assume that the HL registers hold address 2075H. Transfer the program to location 2075H.

Solution

The program can be transferred to location 2075H by using Jump instructions. However, PCHL is a 1-byte instruction that can perform the same function as the Jump instruction. (For an illustration, see the instruction PCHL in the instruction set, Appendix F.)

This instruction is commonly used in monitor programs to transfer the program control from the monitor program to the user's program (see Chapter 17, Section 17.5).

10.7.4 Miscellaneous Instruction

CMC: Complement the Carry Flag (CY)

If the Carry flag is 1, it is reset; and if it is 0, it is set

STC: Set the Carry Flag

These instructions are used in bit manipulation, usually in conjunction with rotate instructions. (See the instruction set in Appendix F for examples.)

10.8

MULTIPLICATION

Multiplication can be performed by repeated addition; this technique is used in BCD-to-binary conversion. It is, however, an inefficient technique for a large multiplier. A more efficient technique can be devised by following the model of manual multiplication of decimal numbers. For example,

$$\begin{array}{r} & & 108 \\ & \times & 15 \\ \hline \text{Step 1:} & (108 \times 5) = & 540 \\ \text{Step 2:} & \text{Shift left and add } (108 \times 1) = & + 108 \\ & & \hline & & 1620 \end{array}$$

In this example, the multiplier multiplies each digit of the multiplicand, starting from the farthest right, and adds the product by shifting to the left. The same process can be applied in binary multiplication.

10.8.1 Illustrative Program: Multiplication of Two 8-Bit Unsigned Numbers

PROBLEM STATEMENT

A multiplicand is stored in memory location XX50H and a multiplier is stored in location XX51H. Write a main program to

1. transfer the two numbers from memory locations to the HL registers.
2. store the product in the Output Buffer at XX90H.

Write a subroutine to

1. multiply two unsigned numbers placed in registers H and L.
2. return the result into the HL pair.

MAIN PROGRAM

```

LXI SP,STACK
LHLD XX50H      ;Place contents of XX50 in L register and contents of
                  ; XX51 in H register
XCHG            ;Place multiplier in D and multiplicand in E
CALL MLTPLY     ;Multiply the two numbers
SHLD XX90H      ;Store the product in locations XX90 and 91H
HLT

```

Subroutine

MLTPLY: This subroutine multiplies two 8-bit unsigned numbers
;Input: Multiplicand in register E and multiplier in register D
;Output: Results in HL register

MLTPLY:	MOV A,D	;Transfer multiplier to accumulator
	MVI D,00H	;Clear D to use in DAD instruction
	LXI H,0000H	;Clear HL
	MVI B,08H	;Set up register B to count eight rotations
NXTBIT:	RAR	;Check if multiplier bit is 1
	JNC NOADD	;If not, skip adding multiplicand.
	DAD D	;If multiplier is 1, add multiplicand to HL and place ; partial result in HL

NOADD:	XCHG	;Place multiplicand in HL
	DAD H	;And shift left
	XCHG	;Retrieve shifted multiplicand
	DCR B	;One operation is complete, decrement counter
	JNZ NXTBIT	;Go back to next bit
	RET	

PROGRAM DESCRIPTION

1. The objective of the main program is to demonstrate use of the instructions LHLD, SHLD, and XCHG. The main program transfers the two bytes (multiplier and multiplicand) from memory locations to the HL registers by using the instruction LHLD, places them in the DE register by the instruction XCHG, and places the result in the Output Buffer by the instruction SHLD.
2. The multiplier routine follows the format—add and shift to the left—illustrated at the beginning of Section 10.8. The routine places the multiplier in the accumulator and rotates it eight times until the counter (B) becomes zero. The reason for clearing D is to use the instruction DAD to add register pairs.
3. After each rotation, when a multiplier bit is 1, the instruction DAD D performs the addition, and DAD H shifts bits to the left. When a bit is 0, the subroutine skips the instruction DAD D and just shifts the bits.

10.9

SUBTRACTION WITH CARRY

The instruction set includes several instructions specifying arithmetic operations with carry (for example, add with carry or subtract with carry, Section 10.7.2). Descriptions of these instructions convey an impression that these instructions can be used to add (or subtract) 8-bit numbers when the addition generates carries. In fact, in these instructions when a carry is generated, it is added to bit D_0 of the accumulator in the next operation. Therefore, these instructions are used primarily in 16-bit addition and subtraction, as shown in the next program.

10.9.1 Illustrative Program: 16-Bit Subtraction

PROBLEM STATEMENT

A set of five 16-bit readings of the current consumption of industrial control units is monitored by meters and stored at memory locations starting at XX50H. The low-order byte is stored first (e.g., at XX50H), followed by the high-order byte (e.g., at XX51H). The corresponding maximum limits for each control unit are stored starting at XX90H. Subtract each reading from its specified limit, and store the difference in place of the readings. If any reading exceeds the maximum limit, call the indicator routine and continue checking.

MAIN PROGRAM

LXI D, 2050H	;Point index to readings
LXI H, 2080H	;Point index to maximum limits
MVI B,05H	;Set up B as a counter
NEXT: CALL SBTRAC	;Point to next location
INX D	;Point to next location
INX H	
DCR B	
JNZ NEXT	
HLT	

Subroutine

;SBTRAC: This subroutine subtracts two 16-bit numbers

;Input: The contents of registers DE point to reading locations

; The contents of registers HL point to maximum limits

;Output: The results are placed in reading locations, thus destroying the initial readings

;The comment section illustrates one example, assuming the following data:

Memory Contents

The first current reading = 6790H	2050 = 90H	LSB
	2051 = 67H	MSB
Maximum limit = 7000 H	2090 = 00H	LSB
	2091 = 70H	MSB

;Illustrative Example

SBTRAC:	MOV A,M;(A)	; (A) = 00H LSB of maximum limit
	XCHG	; (HL) = 2050H
	SUB M	; (A) = 0000 0000 2's complement of 90H
		; (M) = <u>0111 0000</u> Borrow flag is set to
		1 0111 0000 indicate the result is in
		2's complement.
	MOV M,A	; Store at 2050H
	XCHG	; (HL) = 2090H
	INX H	; (HL) = 2091H
	INX D	; (DE) = 2051H
	MOV A,M	; (A) = 70H MSB of the maximum limit
	XCHG	; (HL) = 2051H
	SBB M	; (A) = 0111 0000 (70H)
		; (M) = 1001 1001 2's complement of 67H
		; (CY) = 1 Borrow flag
	CC INDIKET	; Call Indicate subroutine if reading is higher than the
		; maximum limit
	MOV M,A	
	RET	

PROGRAM DESCRIPTION

This is a 16-bit subtraction routine that subtracts one byte at a time. The low-order bytes are subtracted by using the instruction SUB M. If a borrow is generated, it is accounted for by using the instruction SBB M (Subtract with Carry) for high-order bytes. In the illustrative example, the first subtraction (00H – 90H) generates a borrow that is subtracted from high-order bytes. The instruction XCHG changes the index pointer alternately between the set of readings and the maximum limits.

SUMMARY

The following code conversion and 16-bit arithmetic techniques were illustrated in this chapter:

- BCD-to-binary (Section 10.1)
- Binary-to-BCD (Section 10.2)
- BCD to seven-segment-LED code (Section 10.3)
- Binary-to-ASCII code and ASCII-to-binary (Section 10.4)
- BCD addition and subtraction (Sections 10.5 and 10.6)
- Multiplication of two 8-bit unsigned numbers and 16-bit subtraction with carry (Section 10.7)

Review of Instructions

The instructions introduced and illustrated in this section are summarized below.

16-Bit Data Transfer (Copy) and Data Exchange Instructions

- LHLD 16-bit Load HL registers direct
- SHLD 16-bit Store HL registers direct
- XCHG Exchange the contents of HL with DE
- XTHL Exchange the top of the stack with HL
- SPHL Copy HL registers into the stack pointer
- PCHL Copy HL registers into the program counter

Arithmetic Instructions Used in 16-Bit Operations

Addition: The following instructions add the contents of the operand, the carry, and the accumulator.

- ADC R Add register contents with carry
- ADC M Add memory contents with carry
- ACI 8-bit Add immediate 8-bit data with carry

Subtraction: The following instructions subtract the contents of the operand and the borrow from the contents of the accumulator.

SBB R	Subtract register contents with borrow
SBB M	Subtract memory contents with borrow
SBI 8-bit	Subtract immediate 8-bit data with borrow

LOOKING AHEAD

This chapter included various types of code conversion techniques and illustrated applications of more advanced instructions. The illustrative programs were written as independent modules, similar to the circuit boards of an electronic system. Now, what is needed is to link these modules to perform a specific task.

However, single-board microcomputer systems are unsuitable for writing and coding programs with more than 50 instructions. Coding becomes cumbersome, modifications become tedious, and troubleshooting becomes next to impossible. To write a large program, therefore, it is necessary to have access to an assembler and a disk-based system, which will be discussed in the next chapter.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

Section 10.1: BCD-to-Binary Conversion

1. Rewrite the BCDBIN subroutine to include storing results in the Output Buffer. Eliminate unnecessary PUSH and POP instructions.
2. Modify the program (Section 10.1.1) to convert a set of numbers of 2-digit BCD numbers into their binary equivalents and store them in the Output Buffer. The number of BCD digits in the set is specified by the main program in register D and passed on as a parameter to the subroutine.
3. Rewrite the multiplication section using the RLC (Rotate Left) instruction. *Hints:* Rotating left once is equivalent to multiplying by two. To multiply a digit by ten, rotate left three times and add the result of the first rotation (times 10 = times 8 + times 2).
4. In Assignment 3, multiplication is performed by rotating the high-order digit to the left. However, in the BCDBIN subroutine, just before the multiplication section, the high-order digit BCD_2 is shifted right to place it in the low-order position. Rewrite the subroutine to combine the two operations. *Hint:* Rotating BCD_2 right once from the high-order position is equivalent to multiplying it by eight.

Section 10.2: Binary-to-BCD Conversion

5. Assume the STACK is defined as 20B8H in the illustrative program to convert binary to unpacked BCD (Section 10.2.1). Specify the stack addresses and their (symbolic) contents when the BINBCD subroutine is called the second time.
6. Rewrite the main program to supply the powers of ten in registers B and C and to store converted BCD numbers in the Output Buffer. Modify the BINBCD subroutine to accommodate the changes in the main program and eliminate the PWRTEM subroutine.
7. Rewrite the program to convert a given number of binary data bytes into their BCD equivalents, and store them as unpacked BCDs in the Output Buffer. The number of data bytes is specified in register D in the main program. The converted numbers should be stored in groups of three consecutive memory locations. If the number is not large enough to occupy all three locations, zeros should be loaded in those locations.
8. A set of ten BCD readings is stored in the Input Buffer. Convert the numbers into binary and add the numbers. Store the sum in the Output Buffer; the sum can be larger than FFH.

Section 10.3: BCD-to-Seven-Segment-LED Code Conversion

9. List the common-cathode and the common-anode seven-segment-LED look-up table to include hexadecimal digits from 0 to F. (See Figure 4.9 for the diagram.)
10. A set of data is stored as unpacked (single-digit) BCD numbers in memory from XX50H to XX5FH. Write a program to look up the common-cathode-LED code for each reading and store the code from XX55H to XX64H (initial data can be eliminated).
11. Design a counter to count continuously from 00H to FFH with 500 ms delay between each count. Display the count at PORT1 and PORT2 (one digit per port) with a common-anode seven-segment-LED code.
12. Modify the LEDCOD subroutine to account for a carry when the instruction ADD L is executed. For example, if the starting address of the CODE table is 02F9, the codes of digits larger than six will be stored on page 03H. The subroutine given in the illustrative program to convert BCD to LED code (Section 10.3.1) does not account for such a situation. *Hint:* Check for the carry (CY) after the addition, and increment H whenever a carry is generated.

Section 10.4: Binary-to-ASCII Code Conversion

13. A set of ASCII Hex digits is stored in the Input-Buffer memory. Write a program to convert these numbers into binary. Add these numbers in binary, and store the result in the Output-Buffer memory.
14. Extend the program in Assignment 13 to convert the result from binary to ASCII Hex code.

Section 10.5: BCD Addition

15. Write a counter program to count continuously from 0 to 99 in BCD with a delay of 750 ms between each count. Display the count at an output port.
16. Modify the illustrative program for the addition of unsigned numbers (Section 10.5.1) to convert the unpacked BCD digits located at XX60 to XX63 into ASCII characters, and store them in the Output Buffer.

Section 10.6: BCD Subtraction

17. Design a down-counter to count from 99 to 0 in BCD with 500 ms delay between each count. Display the count at an output port. *Hint:* Check for the low-order digit; when it reaches zero, adjust the next digit to nine.
18. Write a program to subtract a 2-digit BCD number from another 2-digit BCD number; the numbers are stored in two consecutive memory locations. Rather than using the 100's complement method, subtract one BCD digit at a time, and decimal-adjust each digit after subtraction. (The instruction DAA cannot be used in subtraction.) Display the result at an output port. Verify that if the subtrahend is larger than the minuend, the result will be negative and will be displayed as the 100's complement.

Section 10.7: Advanced Instructions and Applications

19. A set of 16-bit readings is stored in memory locations starting at 2050H. Each reading occupies two memory locations: the low-order byte is stored first, followed by the high-order byte. The number of readings stored is specified by the contents of register B. Write a program to add all the readings and store the sum in the Output-Buffer memory. (The maximum limit of a sum is 24 bits.)
20. In Assignment 19, save the contents of the stack pointer from the main program, point the stack pointer to location 2050H, and transfer the readings to registers by using the POP instruction. Add the readings as in Assignment 19; however, retrieve the original contents of the stack pointer after the addition is completed.
21. Assume that the monitor program stores a memory address in the DE registers. When a Hex key is pressed to enter a new memory address, the keyboard subroutine places the 4-bit code of the key pressed in the accumulator. Write a subroutine to shift out the most significant four bits of the old address and to insert the new code from the accumulator as the least significant four bits in register E.
Hint: Place the memory address in the HL registers, and use the instruction DAD H four times; this will shift all bits to the left by four positions and will clear the least significant four bits.
22. A pair of 32-bit readings is stored in groups of four consecutive memory locations; the memory location with the lowest memory address in each group contains the least significant byte. Write a program to add these readings; if a carry is generated, call an error routine.

11

Software Development Systems and Assemblers

A **software development system** is a computer that enables the user to develop programs (**software**) with the assistance of other programs. The development process includes writing, modifying, testing, and debugging of the user programs. Programs such as Editor, Assembler, Loader (or Linker), and Debugger enable the user to write programs in mnemonics, translate mnemonics into binary code, and debug the binary code. All the activities of the computer—hardware and software—are directed by another program, called the **operating system** of the computer.

This chapter describes a microprocessor-based software development system, its hardware, and related programs. It also describes widely used operating systems and assemblers and illustrates the use of an assembler to write assembly language programs.

OBJECTIVES

- Describe the components of a software development system.
- List various types of floppy disks, and explain how information is stored on the disk.
- Define the operating system of a microcomputer, and explain its function.
- Explain the functions of programs such as Editor, Assembler, Loader (or Linker), and Debugger.
- List the advantages of an assembler over manual assembly.
- List assembler directives, and explain their functions.
- Write assembly language programs with appropriate directives.

11.1 MICROPROCESSOR-BASED SOFTWARE DEVELOPMENT SYSTEMS

A **software development system** is simply a computer that enables the user to write, modify, debug, and test programs. In a microprocessor-based development system, a microcomputer is used to develop software for a particular microprocessor. Generally, the microcomputer has a large R/W memory (typically, 32M to 256M), disk storage, and a video terminal with an ASCII keyboard. The system includes programs that enable the user to develop software in either assembly language or high-level languages. This text will focus on developing programs in the **8085 assembly language**.

Conceptually, this type of microcomputer is similar to a single-board microcomputer except that it has features that can assist in developing large programs. Programs are accessed and stored under a file name (title), and they are written by using other programs such as text editors and assemblers. The system (I/Os, files, programs, etc.) is managed by a program called the **operating system**. The hardware and software features of a typical software development system are described in the next sections.

11.1.1 System Hardware and Storage Memory

Figure 11.1 shows a typical software development system; it includes an ASCII keyboard, a CRT terminal, an MPU board with 32M to 256M R/W memory and disk controllers, and disk drives. The disk controller is an interfacing circuit through which the MPU can access a disk and provide Read/Write control signals. The disk drives have Read/Write elements, which are responsible for reading and writing data on the disk. At present, most

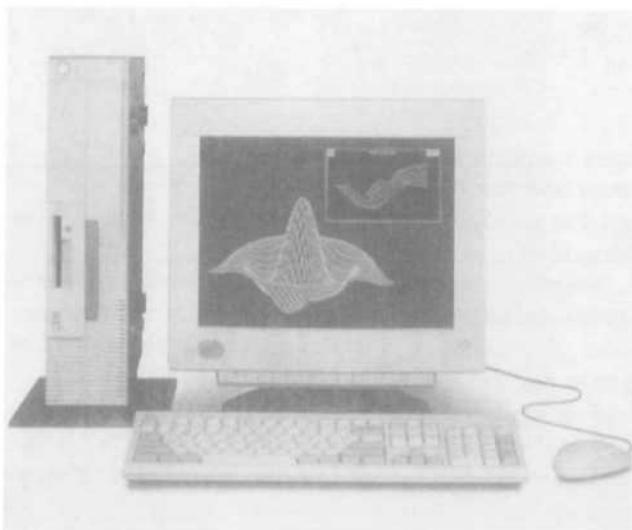


FIGURE 11.1
Typical software development system
SOURCE: Courtesy of International Business Machines

systems are generally equipped with a 3½-inch disk, a hard disk, and CD-ROM (described later), and some systems also include a high-capacity Zip disk for storage. A high-density 3½-inch disk stores 1.44M bytes of information. The storage capacity of a typical hard disk in a PC (personal computer) is 20 G (giga) bytes or higher.

FLOPPY DISK

Figure 11.2(a) shows a typical 3½-inch disk housed in a hard plastic coating. It is made of a thin magnetic material (iron oxide) that can store logic 0s and 1s in the form of magnetic directions. The surface of the disk is divided into a number of concentric tracks, and each track is divided into sectors, as shown in Figure 11.2(b). Data are stored on concentric circular tracks on both sides (known as doubled-sided). The large hole in the center (on the back side, not shown in the figure) is locked by the disk drive when it spins the disk at a constant speed (approximately 300 rpm). The oblong segment, shown as media, is the read/write (R/W) segment; this is the only segment of the surface that comes in contact with the R/W head of the controller. A piece of metal called the shutter normally covers this recording area. When the disk is inserted into a disk drive, the shutter slides over, and the recording surface of the disk is exposed to the R/W head. At the edge of the disk, there is a notch called the *write protect notch*. If the disk notch is open, data cannot be written on the disk; the disk is “write protected.”

Each sector and track (Figure 11.2b) is assigned a binary address using a program (FORMAT); this is called *formatting* a disk. The MPU can access any information on the disk with the sector and the track addresses; however, the access is semirandom. To go from one track to another track, the access is random. Once the track is found, the sector is located serially by counting the sectors. Once data bytes are located, they are transferred to the system’s R/W memory. These various functions in data transfer between a floppy disk and the system are performed by the disk controller and controlled by an operating system, described in Section 11.2.

The **Zip disk** is also becoming common in PC systems. It is slightly larger than the 3½" disk, but stores 100 MB to 250 MB of data and uses a drive similar to a hard drive. This disk is used primarily for backup storage and shipping large files.

HARD DISK

Another type of storage memory used with computers is called a hard disk. The hard disk is similar to the floppy disk except that the magnetic material is coated on a rigid aluminum base and enclosed in a sealed container. In general, the disk is permanently fastened in a dust-free drive mechanism. While it is highly precise and reliable, the hard disk requires sophisticated controller circuitry. The hard disk is more stable mechanically than the floppy disk; therefore, it can be spun at a higher rate (1000 to 3600 rpm) resulting in a faster (by almost ten times) readout rate than that of the floppy disk. Hard disks are available in various sizes and their storage capacity is quite large—in the order of gigabytes.

CD-ROM (COMPACT DISK READ-ONLY MEMORY)

A CD-ROM is an optical disk that uses a laser beam to store digital information that can be read with a laser diode. The disk is immune to dust and mechanical wear because of

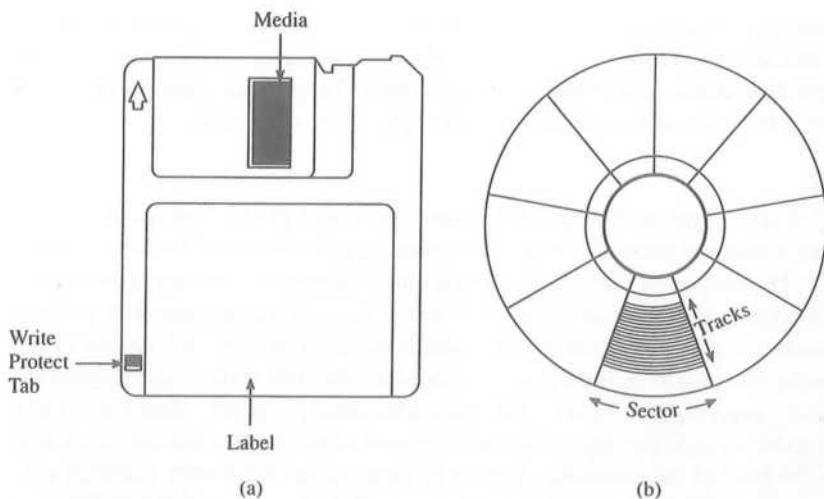


FIGURE 11.2

A Typical 3 1/2-Inch Floppy Disk (a), Its Sectors and Tracks (b)

its optical nature. This is the same technology used in audio CD recording. The CD-ROM comes in various sizes and stores a huge amount of data. Because of its huge storage capacity, the disk can store text, graphics, and audio information, and because it is removable, information can be very easily distributed widely as audio CDs. CD-ROM technology enables the computer to combine and display various types of information using its video screen and attached speakers; this has come to be known as multimedia.

DVD (DIGITAL VIDEO/VERSATILE DISK)

This is also an optical disk storage medium, and it is likely to replace CD-ROM, video and audio tapes in the coming decade. It is essentially a larger and faster CD that can hold a large amount of binary information that includes video, audio, text, and data. The capacity of the CD-ROM is 650 MB vs. the DVD's capacity of 4.7 GB (the equivalent of a two-hour movie) to 17 GB. The DVD spin rate is three times faster than that of a CD. The data transfer rate from a DVD-ROM disk at 1× speed is roughly equivalent to a 9× CD-ROM drive.

11.2

OPERATING SYSTEMS AND PROGRAMMING TOOLS

The **operating system** of a computer is a group of programs that manages or oversees all the operations of the computer. The computer transfers information constantly among peripherals such as a floppy disk, printer, keyboard, and video monitor. It also stores user programs under file names on a disk. (A **file** is defined as related records stored as a single entity.) The operating system is responsible primarily for managing the files on the

disk and the communication between the computer and its peripherals. The functional relationship between the operating system and the computer's various subsystems is shown in Chapter 1, Figure 1.5. From the user's point of view, the operating system is an access or a pathway to run application programs; it is a user interface with the computer.

11.2.1 Operating Systems

Each computer has its own operating system. We focus briefly here on four operating systems, known respectively as MS-DOS (Microsoft Disk Operating System), OS/2 (Operating System 2), Windows 95/98/2000, and UNIX. In the 1970s when most microcomputers were designed using 8-bit microprocessors, the CP/M (Control Program/Monitor) operating system was in common use. The early 1980s began the era of PCs (personal computers) based on 16-bit microprocessors (such as the Intel 8088/8086) with the addressing capability of 1M-byte memory. The CP/M was replaced by MS-DOS in PC-compatible microcomputers. MS-DOS and CP/M are in many ways similar, except that MS-DOS is designed to handle 16-bit microprocessors and 640K memory, and the CP/M was designed for 8-bit microprocessors and 64K memory. In the 1990s, 32-bit processors with gigabytes of memory addressing capacity are being used widely in microcomputers, and therefore, MS-DOS is being replaced by newer operating systems such as Microsoft Windows 95/98/2000), IBM OS/2 (Operating System/2), and UNIX (and UNIX compatibles).

MS-DOS OPERATING SYSTEM

In 16-bit microcomputers, such as IBM PC, XT, and AT, MS-DOS was so widely used that it became the industry standard. Initially, when it was installed on the IBM PC, it was known as PC-DOS; the terms MS-DOS and PC-DOS are interchangeable. MS-DOS is designed to handle 16-bit data word and 640K system memory and disks with quad (high)-density disk format with memory capacity of 720K and 1,200K. Similarly, it can support a hard disk and includes a hierarchical file directory.

OS/2 (OPERATING SYSTEM 2)

OS/2 (and its various versions) is a 32-bit single-user operating system designed by IBM to exploit various powerful features of recent 32-bit (and 64-bit) microprocessors. It has removed the memory limitation of DOS (640K bytes); it can assign 512M bytes of virtual memory space (using storage space from a disk drive) to each application program. Furthermore, it is compatible with DOS and Windows application programs. A few of the important features of OS/2 are as follows. It supports:

- multitasking
- telecommunications
- multimedia

It is a multitasking operating system, meaning the user can run multiple applications concurrently. For example, if the user is using a word processor, OS/2 can send or receive a fax, or send a message (e-mail) electronically in the background. In addition to

32-bit computing capability, it is designed to support various telecommunications features such as electronic mail, fax, and telephone voice mail. Furthermore, it is also well suited for the multimedia applications of CD-ROM.

WINDOWS 95/98/2000 AND ME (MILLENIUM)

This is a 32-bit single-user operating system designed by Microsoft, and it is by far the most widely used operating system in personal computers. It has a graphical interface and it supports the use of a mouse, icons, and menus. It is also a multitasking operating system, similar to OS/2. It includes many of the features of OS/2 described above. Windows 98/2000/ME are upgraded versions of Windows 95.

UNIX-BASED OPERATING SYSTEMS

Unix is a multiuser, multitasking operating system. Initially, it was designed for mainframe computers, but is now used on various machines ranging from powerful microcomputers to supercomputers. It is independent of any particular hardware structure. It is widely used in engineering, scientific, and research environments and is not limited by any memory constraints. Linux, a variation of UNIX, is gaining popularity in PC systems. It is free and its source code is freely distributed. It is ideally suited for Internet and networking environments.

Solaris is a Unix-based operating system designed by Sun Microsystems and is widely used in high-end microcomputers such as workstations and network servers. In the network environment, Solaris is capable of handling 64 computers and is being upgraded to handle 128 computers. One of the major problems with Unix is that there are too many versions in the market; however, several companies have begun to accept Solaris as a standard operating system for high-end microcomputers.

11.2.2 Tools for Developing Assembly Language Programs

In addition to the operating system of a computer, various programs called utility programs are necessary to develop assembly language programs. These utility programs enable the user to perform such functions as copying, printing, deleting, and renaming files. In MS-DOS, commands such as Copy, Del, Dir are given by using a keyboard, and in Windows operating systems, graphical icons and a mouse are used to perform these tasks. The program-development utilities enable the user to write, assemble, and test assembly language programs; they include programs such as Editor, Assembler, Linker (or Loader), and Debugger. The descriptions of various programs and the assembly process described below may vary in details depending upon an operating system and its utility programs.

EDITOR

The Editor is a program that allows the user to enter, modify, and store a group of instructions or text under a file name. The Editor programs can be classified in two groups: line editors and full-screen editors. Line editors, such as EDIT in MS-DOS, work with and manage one line at a time. Full-screen editors (also known as word processors), such as MS Word and WordPerfect, manage the full screen or a paragraph at a time. To write

text, the user must call the Editor under control of the operating system. As soon as the Editor program is transferred from the disk to the system memory, the program control is transferred from the operating system to the Editor program. The Editor has its own commands, and the user can enter and modify text by using those commands. At the completion of writing a program, the exit command of the Editor program will save the program on the disk under the file name and will transfer the program control to the operating system. This file is known as a source file or a source program. If you use editors such as WordPerfect or MS Word, programs should be saved in ASCII format under a filename; otherwise, the assembler may not be able to recognize the text. If the source file is intended to be a program in the 8085 assembly language, the user should follow the syntax of the assembly language and the rules of the assembler that are described next.

The Editor program is not concerned with whether one is writing a letter or an assembly language program. The full-screen editors are convenient for writing either line- or paragraph-oriented text; they automatically adjust lines as words are typed, and the text can be modified or erased with ease.

ASSEMBLER

The Assembler is a program that translates source code or mnemonics into the binary code, called object code, of the microprocessor and generates a file called the Object file. This function is similar to manual assembly, whereby the user looks up the code for each mnemonic in the listing. In addition to translating mnemonics, the Assembler performs various functions, such as error checking and memory allocations. The Assembler is described in more detail in Section 11.3.

LOADER

The Loader (or Linker) is a program that takes the Object file generated by the Assembler program and generates a file in binary code called the COM file or the EXE file. The COM (or EXE) file is the only executable file—i.e., the only file that can be executed by the microcomputer. To execute the program, the COM file is called under the control of the operating system and executed. In different assemblers, the COM file may be labeled by other names.

DEBUGGER

The Debugger is a program that allows the user to test and debug the Object file. The user can employ this program to perform the following functions:

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program, and display register contents after the execution.
- Trace the execution of the specified segment of the program, and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program; i.e., convert the object code into the source code or mnemonics.

11.2.3 Cross-Assemblers

PCs and their compatibles are widely used on college campuses, and we can use PCs to develop (assemble) 8085 assembly language programs by using a program called Cross-Assembler. PCs are designed around Intel processors that have different mnemonics from those of the 8085; thus, we need a program that can translate the 8085 mnemonics, but operate under the PC microprocessor. Such a program is called a **cross-assembler**. For example, the 8085 cross-assembler from 2500 AD Software Inc. has two programs: one is an assembler named X8085 and the other is a linker with the file name LINK. After assembling a program, the Hex file (described later) can be directly transferred to R/W memory of your 8085 single-board microcomputer by using a download program. Thus, programs and/or hardware-related laboratory experiments can be easily performed.

Writing and assembling a program using a cross-assembler such as X8085 on the PC is described as follows.

Step 1: Call an Editor program (such as EDIT in MS-DOS, WordPerfect, or MS Word in Windows) and write an assembly language program in 8085 mnemonics.

This is called a **Source file**. The format of the program is similar to the hand-written programs written in earlier chapters except that it should include **assembler directives** (described in Section 11.3). If the cross-assembler you are using runs under MS-DOS, the name of this file should be limited to eight characters and the extension to three characters. Some assemblers require that the extension be ASM. As an example, the Source file can be named as LAB1.ASM. Save this file as LAB1.ASM and exit from the Editor program. If you use a word processor to write this program, the file must be saved in **ASCII format** (check various options of saving a file in your word processor). Today, cross-assemblers are also available that run under the Windows environment. However, the assembly process remains the same.

Step 2: Call a **cross-assembler** such as X8085 and assemble the Source file LAB1.ASM. The X8085 generates an intermediate binary file called an *object file*, such as LAB1.OBJ, and provides a list of errors. Go back and repeat Step 1 to correct errors. Repeat Step 2 and reassemble the program. Repeat Steps 1 and 2 until the cross-assembler gives a message of zero errors. The cross-assembler also generates a List (LAB1.LST) file that includes memory addresses, machine codes in Hex, labels, and comments. This file is used primarily for documentation.

Step 3: Call a **Link** program and use the intermediate file LAB1.OBJ to generate either an executable file such as LAB1.COM or Hex file LAB1.HEX. However, the executable COM file in 8085 machine code is rather meaningless—the file cannot be executed on the PC because the machine codes of the PC are different from that of the 8085 processor. Assuming you have a single-board microcomputer and a program (called Download) that has the capability of transferring the machine code from PC into R/W memory of the microcomputer, you can use the Hex file and copy the machine codes in proper memory locations of the single-board microcomputer. This is similar to entering or storing a program in the single-board memory by hand.

Step 4: Execute the program on the single-board machine. If you are unable to observe the expected output on a display of the single-board, you must conclude that the program has logic errors (not assembly syntax errors). You can debug the program by using techniques such as single-step and/or break point described in Chapter 7 (Section 7.6). Once you find an error, you must **go back to Step 1 and repeat Steps 1 through 4.** (If errors are simple and obvious, and no additional memory locations are needed, the machine code can be corrected in the single-board microcomputer.)

After the end of the assembly process, you will have the following files on your PC disk.

- **ASM file** This is the Source file written by the user using an Editor. The filename can be one to eight characters long with an extension of a maximum of three characters. The filename and the extension are separated by a dot. As described before, the filename can be LAB1.ASM; the extension ASM suggests that this is an assembly language file.
- **OBJ file** This is the intermediate binary file generated by the cross-assembler.
- **LST file** This is the list file generated by the assembler program for documentation purposes. It contains memory locations, Hex code, mnemonics, and comments.
- **HEX file** This is generated by the Link program and contains program code in hexadecimal notations. This file can be used for debugging the program and to transfer files from one system to another.
- **COM file** This is the executable file generated by the Link program, and it contains binary code. However, this file cannot be executed on the PC. This type of a file can be executed if you were to write an assembly language program for the microprocessor in the PC. However, in such a situation, you would not use a cross-assembler; you would use the assembler for the PC microprocessor.

A summary of steps in writing, assembling, and executing a program using a cross-assembler is shown in the flowchart (Figure 11.3).

ASSEMBLERS AND CROSS-ASSEMBLERS

11.3

The assembler,* as previously described, is a program that translates assembly language mnemonics or source code into binary executable code. This translation requires that the source program be written strictly according to the specified syntax of the assembler. The assembly language source program includes three types of statements:

1. The program statements in 8085 mnemonics that are to be translated into binary code.
2. Comments that are reproduced as part of the program documentation.
3. Directives to the assembler that specify items such as starting memory locations, label definitions, and required memory spaces for data.

*The following description is equally applicable to cross-assemblers.

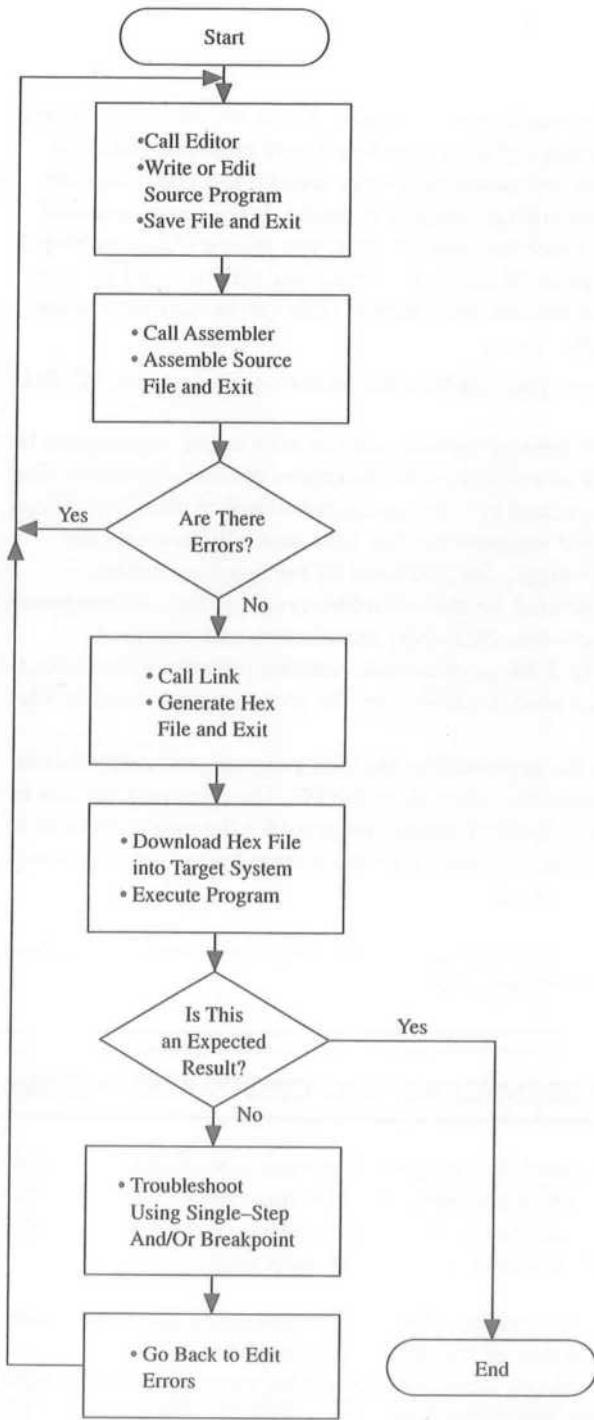


FIGURE 11.3
Flowchart: Program Assembly and Execution

The first two types of statements have been used throughout Part II of this book; the format of these statements as they appear in an assembly language source program is identical to the format used here. The third type—directive statements—and their functions will be described in the next sections.

11.3.1 Assembly Language Format

A typical assembly language programming statement is divided into four parts called fields: *label*, *operation code* (opcode), *operand*, and *comments*. These fields are separated by *delimiters* as shown in Table 11.1.

The assembler statements have a free-field format, which means that any number of blanks can be left between the fields. As mentioned before, comments are optional but are generally included for good documentation. Similarly, a label for an instruction is also optional, but its use greatly facilitates specifying jump locations. As an example, a typical assembly language statement is written as follows:

Label	Opcode	Operand	Comments
START:	LXI	SP,20FFH	;Initialize stack pointer

Delimiters include the colon following START, the space following LXI, the comma following SP, and the semicolon preceding the comment.

TABLE 11.1
Typical Delimiters Used in
Assembler Statements

Delimiter	Placement
1. Colon	After label (optional)
2. Space	Between an opcode and an operand
3. Comma	Between two operands
4. Semicolon	Before the beginning of a comment

11.3.2 Assembler and Cross-Assembler Directives

The **assembler directives** are the instructions to the assembler concerning the program being assembled; they also are called *pseudo instructions* or *pseudo opcodes*. These instructions are neither translated into machine code nor assigned any memory locations in the object file. Some of the important assembler directives for an assembler or a cross-assembler are listed and described here.

Assembler Directives	Example	Description
1. ORG (Origin)	ORG 2000H	The next block of instructions should be stored in memory locations starting at 2000H.
2. END	END	End of assembly. The HLT instruction suggests the end of a program, but that does not necessarily mean it is the end of the assembly.

3.	EQU (Equate)	PORT1 EQU 01H INBUF EQU 2099H STACK EQU INBUF+1	The value of the term PORT1 is equal to 01H. Generally, this means the PORT1 has the port address 01H. The value of the term INBUF is 2099H. This may be the memory location used as Input Buffer. The equate can be expressed by using the label of another equate. This example defines the stack as next location of INBUF.
4.	DB (Define Byte)	DATA: DB A2H,9FH	Initializes an area byte by byte. Assembled bytes of data are stored in successive memory locations until all values are stored. This is a convenient way of writing a data string. The label is optional.
5.	DW (Define Word)	DW 2050H	Initializes an area two bytes at a time.
6.	DS (Define Storage)	OUTBUF: DS 4	Reserves a specified number of memory locations. In this example, four memory locations are reserved for OUTBUF.

11.3.3 Cross-Assembler Format

As mentioned earlier, a cross-assembler is a program that can be used to translate 8085 mnemonics by a computer that has a microprocessor other than the 8085. The format of a cross-assembler is almost identical to that of an assembler with a few variations. Programmers who write these programs generally tend to follow a similar format. Occasionally, personal preferences of these programmers or some added features may cause some variations. For example, the cross-assembler from 2500AD Software Inc. has pseudo opcodes similar to those in the previous section. The directives such as LWORD (Long Word: 32-bit) or BLKW (Block Word for storage) can be cited as additional features. From the user point of view, however, the process of assembling code is almost identical. An example of how to write a program by using a cross-assembler is illustrated in Section 11.4.

11.3.4 Advantages of the Assembler/Cross-Assembler

The **assembler** is a tool for developing programs with the assistance of the computer. Assemblers are absolutely essential for writing industry-standard software; manual assembly is too difficult for programs with more than 50 instructions. The assembler performs many functions in addition to translating mnemonics, and it has several advantages over manual assembly. The salient features of the assembler are as follows:

1. The assembler translates mnemonics into binary code with speed and accuracy, thus eliminating human errors in looking up the codes.
2. The assembler assigns appropriate values to the symbols used in a program. This facilitates specifying jump locations.
3. It is easy to insert or delete instructions in a program; the assembler can reassemble the entire program quickly with new memory locations and modified addresses for jump locations. This avoids rewriting the program manually.
4. The assembler checks syntax errors, such as wrong labels and expressions, and provides error messages. However, it cannot check logic errors in a program.
5. The assembler can reserve memory locations for data or results.
6. The assembler can provide files for documentation.
7. A Debugger program can be used in conjunction with the assembler to test and debug an assembly language program.

The above comments, except number 7, are equally applicable to cross-assemblers.

WRITING PROGRAMS USING A CROSS-ASSEMBLER*

11.4

This section deals primarily with writing programs using a cross-assembler written for IBM PCs. The following examples are taken from previous chapters in which the programs were assembled manually. An assembler source program is identical to a program the user writes with paper and pencil, except that the assembler source program includes assembler directives.

11.4.1 Illustrative Program: Unconditional Jump to Set Up a Loop

The following program is taken from Chapter 6 (Section 6.4.2). Its source program is rewritten here for the assembler.

SOURCE PROGRAM

;This program monitors the switch positions of the input port and turns on/off devices
; connected to the output port.

PORT0	EQU 00H	;Input port address
PORT1	EQU 01H	;Output port address
	ORG 2000H	;Start assembling the program from location 2000H
START:	IN PORT0	;Read input switches
	OUT PORT1	;Turn on devices
	IMP START	;Go back and read switches again
	END	

*The terms *assembler* and *cross-assembler* are used synonymously.

This program illustrates the following assembler directives:

- ORG The object code will be stored starting at the location 2000H.
- EQU The program defines two equates, PORT0 and PORT1. In this program it would have been easier to write port addresses directly with the instructions. The equates are essential in development projects in which hardware and software design are done concurrently. In such a situation, equates are convenient. Equates are also useful in long programs because it is easy to change or define port addresses by defining equates.
- Label The program illustrates one label: START
- END The end of assembly.

TWO-PASS ASSEMBLER

To assemble the program, the assembler scans through the program twice; this is known as a **two-pass assembler**. In the first pass, the first memory location is determined from the ORG statement, and the counter known as the location counter is initialized. Then the assembler scans each instruction and records locations in the address column of the first byte of each instruction; the location counter keeps track of the bytes in the program. The assembler also generates a symbol table during the first pass. When it comes across a label, it records the label and its location. In the second pass, each instruction is examined, and mnemonics and labels are replaced by their machine codes in Hex notation as shown below.

Pass 1

Address Hex	Machine Code Hex	Label Opcode	Operand	Symbol Table
2000		START: IN	PORT0	PORT0 00H
2002		OUT	PORT1	PORT1 01H
2004		JMP	START	START 2000H

Pass 2

2000	DB00
2002	D301
2004	C30020

ASSEMBLED LIST FILE

The file lists the memory addresses of the first byte of each instruction.

;This program monitors the switch positions of the input port and turns on/off devices
; connected to the output port

0000 =	PORT0 EQU 00H	;Input port address
0001 =	PORT1 EQU 01H	;Output port address
2000	ORG 2000H	;Start assembling program from location ; 2000H
2000 DB00	START: IN PORT0	;Read input switches
2002 D301	OUT PORT1	;Turn on devices
2004 C30020	JMP START	;Go back and read switches again
2007	END	

11.4.2 Illustrative Program: Addition with Carry

The following program is from Chapter 7 (Section 7.3.2). The program adds six bytes of data stored in memory locations starting at 2050H and stores the sum in the Output Buffer memory in two consecutive memory locations.

Data(H) A2,FA,DF,E5,98,8B.

SOURCE PROGRAM

;Addition with Carry

INBUF	EQU 2050H	;Input Buffer location
COUNTR	EQU 06H	;Number of bytes to add
	ORG 2000H	
	XRA A	;Clear accumulator
	MOV B,A	;Set up B for carry
	MVI C,COUNTR	;Set up C to count bytes
	LXI H,INBUF	;Point to data address
NXTBYT:	ADD M	
	JNC NXTMEM	
	INR B	;If there is carry, add 1
NXTMEM:	INX H	;Point to next data byte
	DCR C	;One is added, decrement counter
	JNZ NXTBYT	;Get next byte if all bytes not yet added
	LXI H,OUTBUF	;Point index to output buffer
	MOV M,A	;Store low-order byte of the sum
	INX H	
	MOV M,B	;Store high-order byte of the sum
	HLT	
OUTBUF:	DS 2	;Reserve two memory locations
	ORG 2050H	;Assemble next instructions starting at 2050H
DATA:	DB 0A2H,0FAH,0DFH,0E5H,98H	
	DB 8BH	
	END	

This program illustrates two more assembler directives—DB (Define Byte) and DS (Define Storage)—and the use of the ORG statement to store data starting at location 2050H. A data byte or an address that begins with Hex digits (A through F) should be preceded by 0; otherwise, the assembler cannot interpret it as a Hex number (see the data string above). The assembled program is shown below.

PRINT FILE

;Addition with Carry

2050 =	INBUF	EQU 2050H	;Input Buffer location
0006 =	COUNTR	EQU 06H	;Number of bytes to add
2000		ORG 2000H	
2000 AF		XRA A	;Clear accumulator
2001 47		MOV B,A	;Set up B for carry
2002 0E06		MVI C,COUNTR	;Set up C to count bytes
2004 215020		LXI H,INBUF	;Point to data address
2007 86	NXTBYT	ADD M	
2008 D20C20		JNC NXTMEM	
200B 04		INR B	;If there is carry, add 1
200C 23	NXTMEM:	INX H	;Point to next data byte
200D 0D		DCR C	;Decrement counter
200E C20720		JNZ NXTBYT	;Get next byte if all bytes ; not yet added
2011 211820		LXI H,OUTBUF	;Point index to Output
2014 77		MOV M,A	;Store low-order byte
2015 23		INX H	
2016 70		MOV M,B	;Store high-order byte
2017 76		HLT	
2018	OUTBUF:	DS 2	;Reserve two memory ; locations
2050		ORG 2050H	;Assemble next instructions ; starting at 2050H
2050 A2FADFE598	DATA:	DB 0A2H,0FAH,0DFH,0E5H,98H	
2055 8B		DB 8BH	
2056		END	

The list file shows the memory addresses of the first byte of each instruction; this is a typical printout of an assembly language program. The file also shows that two memory locations (2018H and 2019H) are reserved for OUTBUF, and six locations are used to store data starting from 2050H.

ERROR MESSAGES

In addition to translating the mnemonics into object code, the assembler also gives error messages. The two types of error messages are terminal error messages and source program error messages. In the first case, the assembler is not able to complete the assembly. In the second case, the assembler is able to complete the assembly, but it lists the errors.

11.4.3 Illustration of a List File Using a Cross-Assembler

The following program, which adds two Hex bytes, was assembled on an IBM PC-compatible microcomputer by using the cross-assembler from 2500AD Software Inc. The source program shown below was written using an Editor.

SOURCE PROGRAM

;This program adds two Hex bytes and stores the sum in memory

OUTBUF	ORG 2000H	;Begin assembly at 2000H
START:	EQU 2050H	;Address to store sum
	MVI B,32H	;Load first byte
	MVI C,0A2H	;Load second byte
	MOV A,C	
	ADD B	;Add two bytes
	STA OUTBUF	;Store sum in buffer
	HLT	;End of program
	END	;End of assembly

The source program was assembled by calling the X8085, the file name of the cross-assembler. This cross-assembler generates two files: object (.OBJ) and list (.LST) files. The object file is used as an input to the program called LINK1 to generate the executable file (.TSE). The object file can be assembled at any starting memory location. The LINK1 program can relocate a file at a desired location or combine several files and assign appropriate memory locations. However, the 8085 processor executable file, also known as a COM file, cannot be executed on the PC. The LINK2 program can generate a Hex file that can be used to download the machine code to a single-board computer. The list file and the messages generated by the cross-assembler are shown below.

List File:

2500AD 8085 CROSS-ASSEMBLER-VERSION 3.01a.

INPUTFILENAME: ADDHEX.ASM
OUTPUTFILENAME: ADDHEX.OBJ

```

1      ;This program adds two Hex bytes
2      ; and stores the sum in memory
3
4      2000          ORG 2000H    ;Begin assembly at 2000H
5
6      50 20        OUTBUF:    EQU 2050H    ;Address to store sum

```

```
7  
8 2000 06 32      START: MVI B,32H ;Load first byte  
9  
10 2002 0E A2      MVI C,0A2H ;Load second byte  
11  
12 2004 79      Mov A,C  
13  
14 2005 80      ADD B      ;Add two bytes  
15  
16 2006 32 50 20    STA OUTBUF ;Store sum in buffer  
17  
18 2009 76      HLT      ;End of program  
19  
20 200A          END      ;End of assembly
```

```
***** SYMBOLIC REFERENCE TABLE *****
```

```
OUTBUF =2050      START 2000
```

```
LINES ASSEMBLED: 20      ASSEMBLY ERRORS: 0
```

Precautions in Writing Programs Assembler and cross-assembler programs are available from various software companies; for the most part, they follow similar formats. However, we suggest the following precautions in writing assembly language programs.

1. Some assemblers do not allow free format, meaning the unnecessary spaces are not tolerated.
2. The letter following a number specifies the type of a number. A hexadecimal number is followed by the letter H, an octal by the letter O or Q, a binary by the letter B. A number without a letter is interpreted as a decimal number.
3. Any Hex number that begins with A through F must be preceded by zero; otherwise, the assembler interprets the number as a label and gives an error message because it cannot find the label.
4. Some assemblers must have a colon after a label.
5. In some assemblers, Equate statements must begin in column 1.

SUMMARY

A software development system and an assembler are essential tools for writing large assembly language programs. These tools facilitate the writing, assembling, testing, and debugging of assembly language programs.

A disk-based microcomputer, its operating system, and assembler programs can serve as a development system. All the operations of the computer are managed and directed by the operating system of the computer. The Assembler and other utility programs assist the user in developing software. The Editor allows the user to enter text; the Assembler translates mnemonics into machine code and provides error messages. The Debugger assists in debugging the program. The 8085 code can be assembled by using a program called a cross-assembler on a system that has a processor other than the 8085.

The program assembled using the Assembler or a cross-assembler is in many ways similar to that of the hand assembly program except that the program written for the Assembler includes assembler directives concerning how to assemble the program. The Assembler has many advantages over manual assembly; without the Assembler, it would be extremely difficult to develop industry-standard software.

QUESTIONS AND PROGRAMMING ASSIGNMENTS

1. Check the appropriate answer in the following statements:
 - a. The process of accessing information on a floppy disk is
 - (1) random.
 - (2) serial.
 - (3) semirandom.
 - b. The operating system of a computer is defined as
 - (1) hardware that operates the floppy disk.
 - (2) a program that manages files on the disk.
 - (3) a group of programs that manages and directs hardware and software in the system.
 - c. The Editor is
 - (1) an assembly language program that reads and writes information on the disk.
 - (2) a high-level language program that allows the user to write programs.
 - (3) a program that allows the user to write, modify, and store text in the computer system.
 - d. The Assembler is
 - (1) a compiler that translates statements from high-level language into assembly language.
 - (2) a program that translates mnemonics into binary code.
 - (3) an operating system that manages all the programs in the system.
 - e. A file is
 - (1) a group of related records stored as a single entity.
 - (2) a program that transfers information between the system and the floppy disk.
 - (3) a program that stores data.

- f. A disk controller is
- a program that manages the files on the disk.
 - a circuit that interfaces the disk with the microcomputer system.
 - a mechanism that controls the spinning of the disk.
2. Assemble the following program at the starting location 0100H, and list the errors in the source file. Assume that the subroutine BCDBIN is written separately.

```
ORG 0100
LXI SP,STACK      ;Initialize stack pointer
LXI H,INBUF       ;Output buffer
LXI B,OUTBUF      ;Output buffer
MVI D,0AH          ;Initialize counter
NEXT:   MOV A,M      ;Get byte
        CALL BCDBIN    ;Call BCD to binary routine
        STAX B          ;Store result
        DCR D
        JNZ NEXT
INBUF:  DW
OUTBUF: DW
        HLT
```

3. Rewrite the illustrative program in Section 10.3.1 for a BCD to common-cathode-LED code conversion to assemble it with an assembler or a cross-assembler.

III

Interfacing Peripherals (I/Os) and Applications

CHAPTER 12

Interrupts

CHAPTER 13

Interfacing Data Converters

CHAPTER 14

Programmable Interface Devices

CHAPTER 15

General-Purpose Programmable Peripheral Devices

CHAPTER 16

Serial I/O and Data Communication

CHAPTER 17

Microprocessor Applications

CHAPTER 18

Extending 8-Bit Microprocessor Concepts to Higher-Level Processors and Microcontrollers

Part III of this book is concerned with the interfacing of peripherals (I/Os) and design processes of microcomputer-based systems. The primary objectives of Part III are to:

1. Examine the concepts and processes of data transfer, such as interrupts, Direct Memory Access (DMA), and serial I/O, between the microprocessor and peripherals.
2. Apply the concepts of data transfer for interfacing peripherals with the microprocessor.
3. Synthesize the concepts of microprocessor architecture, software, and interfacing by designing a simple microprocessor-based system and by discussing the process of troubleshooting.

The primary function of the **microprocessor** is to accept data from input devices such as keyboards and A/D converters, read instructions from memory, process data according to the instructions, and send the results to output devices such as LEDs, printers, and video monitors. These input and output devices are called either **peripherals** or **I/Os**. Designing logic circuits (hardware) and writing instructions (software) to enable the microprocessor to communicate with these peripherals is called **interfacing**, and the logic circuits are called **I/O ports** or **interfacing devices**.

The microprocessor (or more precisely, MPU) communicates with the peripherals in either of two formats: **asynchronous** or **synchronous**.

Similarly, it transfers data in either of two modes: **parallel I/O** or **serial I/O**. The 8085 identifies peripherals either as **memory-mapped I/O** or **peripheral I/O** on the basis of their interfacing logic circuits (Chapter 4). Data transfer between the microprocessor and its peripherals can take place under various conditions, as shown in the chart. The modes, the techniques, the instructions, and the conditions of data transfer are briefly described in the following paragraphs and summarized in the chart.

FORMATS OF DATA TRANSFER: SYNCHRONOUS AND ASYNCHRONOUS

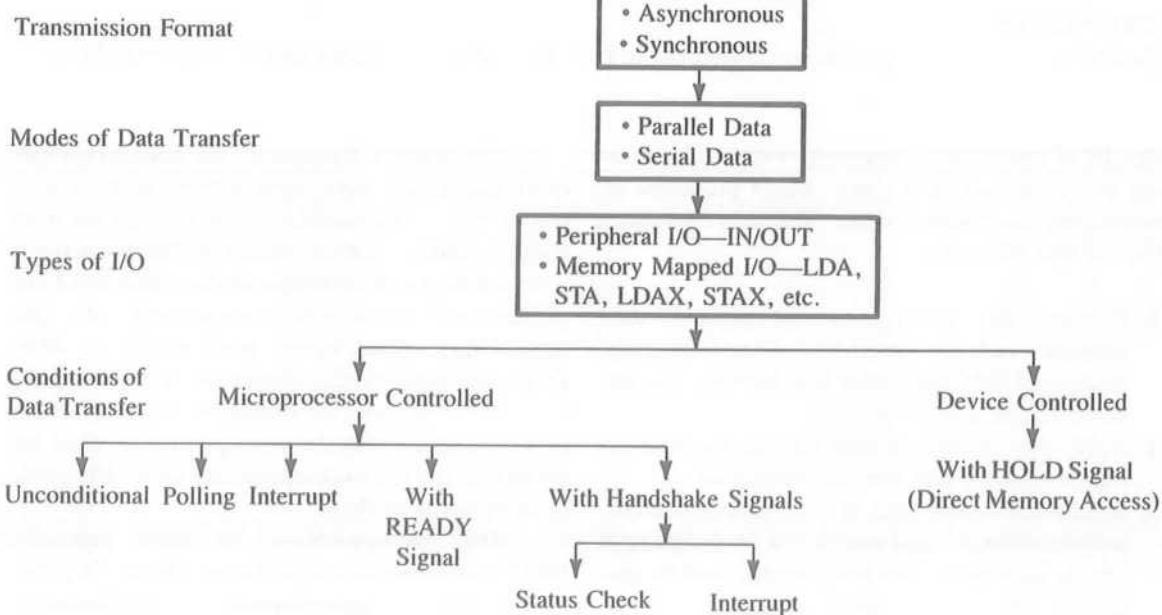
Synchronous means at the same time; the transmitter and receiver are synchronized with the same clock. *Asynchronous* means at irregular intervals. The synchronous format is used in high-speed data transmission and the asynchronous format is used for low-speed data transmission. Data transfer be-

tween the microprocessor and the peripherals is primarily asynchronous.

MODES OF DATA TRANSFER: PARALLEL AND SERIAL

The microprocessor receives (or transmits) binary data in either of two modes: parallel or serial. In the parallel mode, the entire word (4-bit, 8-bit, or 16-bit) is transferred at one time. In the 8085, an 8-bit word is transferred simultaneously over the eight data lines. The devices commonly used for parallel data transfer are keyboards, seven-segment LEDs, data converters, and memory.

In the serial mode, data are transferred one bit at a time over a single line between the microprocessor and a peripheral. For data transmission from the microprocessor to a peripheral, a word is converted into a stream of eight bits; this is called parallel-to-serial conversion. For reception, a stream of eight bits is converted into a parallel word; this is



called serial-to-parallel conversion. The serial I/O mode is commonly used with peripherals such as CRT terminals, printers (also used in parallel I/O mode), cassette tapes, and modems for telephone lines.

TYPES OF I/O: PERIPHERAL AND MEMORY-MAPPED

In peripheral I/O, a peripheral is identified with an 8-bit address. The 8085 has two instructions—IN and OUT—to implement data transfer between the microprocessor and peripherals. These are 2-byte instructions; the second byte specifies the address or the port number of a peripheral. The instruction IN transfers (copies) data from an input device to the accumulator, and the instruction OUT transfers data from the accumulator to an output device.

In memory-mapped I/O, a peripheral is connected as if it were a memory location, and it is identified with a 16-bit address. Data transfer is implemented by using memory-related instructions such as STA; LDA; MOV M,R; and MOV R,M.

CONDITIONS OF DATA TRANSFER

The process of data transfer between the microprocessor and the peripherals is controlled either by the microprocessor or by the peripherals, as shown in the chart. Data transfer is generally implemented under the microprocessor control when the peripheral response is slow relative to that of the microprocessor.

MICROPROCESSOR-CONTROLLED DATA TRANSFER

Most peripherals respond slowly in comparison with the speed of the microprocessor. Therefore, it is necessary to set up conditions for data transfer so that data will not be lost during the transfer. Microprocessor-controlled data transfer can take place under five different conditions: unconditional, polling (also known as status check), interrupt, with READY signal, and with handshake signals. These conditions are described briefly.

Unconditional Data Transfer In this form of data transfer, the microprocessor assumes that a peripheral is always available. For example, to display data at an LED port, the microprocessor simply enables the port, transfers data, and goes on to execute the next instruction.

Data Transfer with Polling (Status Check) In this form of data transfer, the microprocessor is kept in a loop to check whether data are available; this is called polling. For example, to read data from an input keyboard in a single-board microcomputer, the microprocessor can keep polling the port until a key is pressed.

Data Transfer with Interrupt In this condition, when a peripheral is ready to transfer data, it sends an interrupt signal to the microprocessor. The microprocessor stops the execution of the program, accepts the data from the peripheral, and then returns to the program. In the interrupt technique, the processor is free to perform other tasks rather than being held in a polling loop.

Data Transfer with READY Signal When peripheral response time is slower than the execution time of the microprocessor, the READY signal can be used to add T-states, thus extending the execution time. This process provides sufficient time for the peripheral to complete the data transfer. The technique is commonly used in a system with slow memory chips.

Data Transfer with Handshake Signals In this data transfer, signals are exchanged between the microprocessor and a peripheral prior to actual data transfer; these signals are called handshake signals. The function of handshake signals is to ensure the readiness of the peripheral and to synchronize the timing of the data transfer. For example, when an A/D converter is used as an input device, the microprocessor needs to wait because of the slow conversion time of the converter. At the end of the conversion, the A/D converter sends the Data Ready (DR), also known as End of Conversion, signal to the mi-

croprocessor. Upon receiving the DR signal, the microprocessor reads the data and acknowledges by sending a signal to the converter that the data have been read. During the conversion period, the microprocessor keeps checking the DR signal; this technique is called the status check with handshake signals. This status check method is functionally similar to the polling method and achieves the same results.

Rather than using the handshake signals for the status check, the signals can be used to implement data transfer with interrupt. In the above example of the A/D converter, the DR signal can be used to interrupt the microprocessor.

Handshake signals prevent the microprocessor from reading the same data more than once, from a slow device, and from writing new data before the device has accepted the previous data.

PERIPHERAL-CONTROLLED DATA TRANSFER

The last category of data transfer shown in the chart is device-controlled I/O. This type of data transfer is employed when the peripheral is much faster than the microprocessor. For example, in the case of Direct Memory Access (DMA), the DMA controller sends a HOLD signal to the microprocessor, the microprocessor releases its data bus and the address bus to the DMA controller, and data are transferred at high speed without the intervention of the microprocessor.

CHAPTER TOPICS

In Chapter 5, we discussed the basic concepts of unconditional data transfer in parallel I/O. It included examples of interfacing simple devices, such as LEDs and switches. The remaining processes of data transfer shown in the chart are discussed in Chapters 12 through 16.

Chapter 12 deals primarily with interrupts. It includes the 8085 interrupts with several examples.

Chapter 13 is concerned with the interfacing of data converters. After reviewing the basic concepts underlying data converters, the chapter presents examples of interfacing data converters.

Chapter 14 deals with programmable interface devices commonly used in small microprocessor-based systems. These devices can be set up to perform I/O tasks by writing instructions in their control registers, thus the title programmable devices. The chapter includes several illustrations of interfacing using programmable devices from the Intel family, such as the 8155/8156 (Memory with I/O and Timer) and the 8279 (Programmable Keyboard Display Interface).

Chapter 15 is an extension of the topics examined in Chapter 14. It includes general-purpose programmable interface devices, such as the 8255A (Programmable Peripheral Interface), the 8254 (Timer), the 8259A (Interrupt Controller), and the 8237 (DMA Controller).

Chapter 16 deals with serial I/O. It includes discussion of the software approach and the hardware approach to serial I/O. The hardware approach is illustrated with the example of interfacing a terminal using the 8251A (Programmable Communication Interface).

Chapter 17 is concerned with the process of designing a microprocessor-based product. The primary objective of this chapter is to synthesize the concepts, using both hardware and software, discussed in all previous chapters. The chapter includes several interfacing projects and a design project.

Chapter 18 introduces concepts in 16-bit microprocessors using the Intel 8088/86 microprocessor family and suggests the trends in microprocessor technology. It also describes other 8-bit microprocessors and microcontrollers such as the Zilog Z80, Intel 8051, and Motorola 68HC11. The chapter concludes with descriptions of the latest 32- and 64-bit microprocessors.

PREREQUISITES

- Basic concepts of microprocessor architecture, memory, and I/Os (see Part I).
- Familiarity with the 8085 instruction set and programming techniques (see Part II).

12

Interrupts

The interrupt I/O is a process of data transfer whereby an external device or a peripheral can inform the processor that it is ready for communication and it requests attention. The process is initiated by an external device and is asynchronous, meaning that it can be initiated at any time without reference to the system clock. However, the response to an interrupt request is directed or controlled by the microprocessor.

The interrupt requests are classified in two categories: maskable interrupt and nonmaskable interrupt. The 8085 microprocessor includes four maskable interrupts and one nonmaskable interrupt. Among the four maskable interrupts, one is nonvectored, which requires external hardware to supply a Call location to restart the execution. The other three are vectored to specific locations (explained later in this chapter).

The microprocessor can ignore or delay a maskable interrupt request if it is performing some critical task; however, it has to respond to a nonmaskable request immediately. In many ways, the maskable interrupt is like a telephone, which can be kept off the hook if one is not interested in receiving any messages. The nonmaskable interrupt is like a smoke detector, which should be attended to if set off.

The interrupt process allows the microprocessor to respond to these external requests for attention or service on a demand basis and leaves the microprocessor free to perform other tasks. On the other hand, in the polled or the status check I/O, the microprocessor remains in a loop, doing nothing, until the device is ready for data transfer.

This chapter first describes the nonvectored interrupt process. It includes a discussion of how

multiple interrupts are implemented with one interrupt line and how priorities are determined. The remaining three vectored interrupts and the nonmaskable interrupt are described later in the chapter. Two examples of the interrupt I/O are illustrated: a clock timer with the 60 Hz power line as the interrupting source, and a software breakpoint routine. The chapter also includes a brief explanation of the interrupt controller, the 8259, and the I/O process called Direct Memory Access (DMA).

OBJECTIVES

- Explain an interrupt process and the difference between a nonmaskable and a maskable interrupt.
- Explain the instructions EI, DI, and RST and their functions in the 8085 interrupt process.

- List the eight steps to initiate and implement the 8085 interrupt.
- Design and implement an interrupt with a given RST instruction.
- Explain how to connect multiple interrupts with the INTR interrupt line and how to determine their priorities using logic circuits.
- List the 8085 vectored interrupts, nonmaskable interrupt and their vectored memory locations.
- Explain the instructions SIM and RIM, and illustrate how to use them for the 8085 interrupts.
- Explain how to use an RST instruction to implement a software breakpoint.
- Explain features of the programmable interrupt controller, the 8259A, and the Direct Memory Access (DMA) data transfer.

12.1 THE 8085 INTERRUPT

The 8085 interrupt process is controlled by the Interrupt Enable flip-flop, which is internal to the processor and can be set or reset by using software instructions. If the flip-flop is enabled and the input to the interrupt signal INTR (pin 10) goes high, the microprocessor is interrupted. This is a maskable interrupt and can be disabled. The 8085 has a nonmaskable and three additional vectored interrupt signals as well. The best way to describe the 8085 interrupt process is to compare it to a telephone with a blinking light instead of a ring.

Assume that you are reading an interesting novel at your desk, where there is a telephone. For you to receive and respond to a telephone call, the following steps should occur:

1. The telephone system should be enabled, meaning that the receiver should be on the hook.
2. You should glance at the light at certain intervals to check whether someone is calling.
3. If you see a blinking light, you should pick up the receiver, say hello, and wait for a response. Once you pick up the phone, the line is busy, and no more calls can be received until you replace the receiver.
4. Assuming that the caller is your roommate, the request may be: It is going to rain today. Will you please shut all the windows in my room?
5. You insert a bookmark on the page you are reading.
6. You replace the receiver on the hook.
7. You shut your roommate's windows.
8. You go back to your book, find your mark, and start reading again.

Steps 6 and 7 may be interchanged, depending on the urgency of the request. If the request is critical and you do not want to be interrupted while attending to the request, you are likely to attend to the request first, then put the receiver back on the hook. The 8085 interrupt process can be described in terms of those eight steps.

Step 1: The interrupt process should be enabled by writing the instruction EI in the main program. This is similar to keeping the phone receiver on the hook. The instruction EI sets the Interrupt Enable flip-flop. The instruction DI resets the flip-flop and disables the interrupt process.

Instruction EI (Enable Interrupt)

- This is a 1-byte instruction.
- The instruction sets the Interrupt Enable flip-flop and enables the interrupt process.
- System reset or an interrupt disables the interrupt process.

Instruction DI (Disable Interrupt)

- This is a 1-byte instruction.
- The instruction resets the Interrupt Enable flip-flop and disables the interrupt.
- It should be included in a program segment where an interrupt from an outside source cannot be tolerated.

Step 2: When the microprocessor is executing a program, it checks the INTR line during the execution of each instruction.

Step 3: If the line INTR is high and the interrupt is enabled, the microprocessor completes the current instruction, disables the Interrupt Enable flip-flop and sends a signal called INTA—Interrupt Acknowledge (active low). The processor cannot accept any interrupt requests until the interrupt flip-flop is enabled again.

Step 4: The signal INTA is used to insert a restart (RST) instruction (or a Call instruction) through external hardware. The RST instruction is a 1-byte call instruction (explained below) that transfers the program control to a specific memory location on page 00H and restarts the execution at that memory location after executing Step 5.

Step 5: When the microprocessor receives an RST instruction (or a Call instruction), it saves the memory address of the next instruction on the stack. This is similar to inserting a bookmark. The program is transferred to the CALL location.

Step 6: Assuming that the task to be performed is written as a subroutine at the specified location, the processor performs the task. This subroutine is known as a service routine.

Step 7: The service routine should include the instruction EI to enable the interrupt again. This is similar to putting the receiver back on the hook.

Step 8: At the end of the subroutine, the RET instruction retrieves the memory address where the program was interrupted and continues the execution. This is similar

to finding the page where you were interrupted by the phone call and continuing to read.

We will elaborate further on the restart instructions, additional hardware mentioned in Step 4, and multiple interrupts.

12.1.1 RST (Restart) Instructions

The 8085 instruction set includes eight RST (Restart) instructions listed in Section 9.3. These are 1-byte Call instructions that transfer the program execution to a specific location on page 00H, as listed in Table 12.1. The RST instructions are executed in a similar way to that of Call instructions. The address in the program counter (meaning the address of the next instruction to an RST instruction) is stored on the stack before the program execution is transferred to the RST call location. When the processor encounters a Return instruction in the subroutine associated with the RST instruction, the program returns to the address that was stored on the stack. In case of a hardware interrupt, we will use an RST instruction to restart the program execution.

To implement Step 4 in the interrupt process, insert one of these instructions in the microprocessor by using external hardware and the signal INTA (Interrupt Acknowledge), as shown in Figure 12.1.

In Figure 12.1, the instruction RST 5 is built using resistors and a tri-state buffer. Figure 12.2 shows the timing of the 8085 Interrupt Acknowledge machine cycle. In response to the INTR (Interrupt Request) high signal, the 8085 sends the INTA (Interrupt Acknowledge) low signal, which is used to enable the buffer, and the RST instruction is placed on the data bus during M_1 . During M_1 , the program counter holds the memory address of the next instruction, which should be stored on the stack so that the program can continue after the service routine. During M_2 , the address of the stack pointer minus one ($SP - 1$) location is placed on the address bus, and the high-order address of the program counter is stored on the stack. During M_3 , the low-order address of the program counter is stored in the next location ($SP - 2$) of the stack.

The machine cycle M_1 of the Interrupt Acknowledge is identical with the Opeode Fetch cycle, with two exceptions. The INTA signal is sent out instead of the RD signal,

TABLE 12.1
Restart Instructions

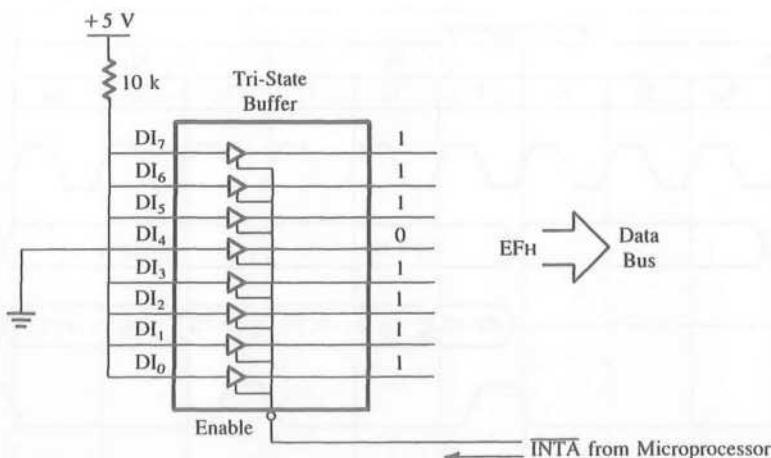


FIGURE 12.1

A Circuit to Implement the Instruction RST 5

and the status lines (IO/M, S₀ and S₁) are 1 1 1 instead of 0 1 1 (see Figure 12.2). During M₁, the RST 5 is decoded, a 1-byte Call instruction to location 0028H. The machine cycles M₂ and M₃ are Memory Write cycles that store the contents of the program counter on the stack, and then a new instruction cycle begins.

In this next instruction cycle, the program is transferred to location 0028H. The service routine is written somewhere else in memory, and the Jump instruction is written at 0028H to specify the address of the service routine. All these steps are illustrated in the following example.

12.1.2 Illustration: An Implementation of the 8085 Interrupt

PROBLEM STATEMENT

1. Write a main program to count continuously in binary with a one-second delay between each count.
2. Write a service routine at XX70H to flash FFH five times when the program is interrupted, with some appropriate delay between each flash.

MAIN PROGRAM

Memory Address	Label	Mnemonics	Comments
XX00		LXI SP,XX99H	;Initialize stack pointer
03		EI	;Enable interrupt process
04		MVI A,00H	;Initialize counter
06	NXTCNT:	OUT PORT1	;Display count
08		MVI C,01H	;Parameter for 1-second delay

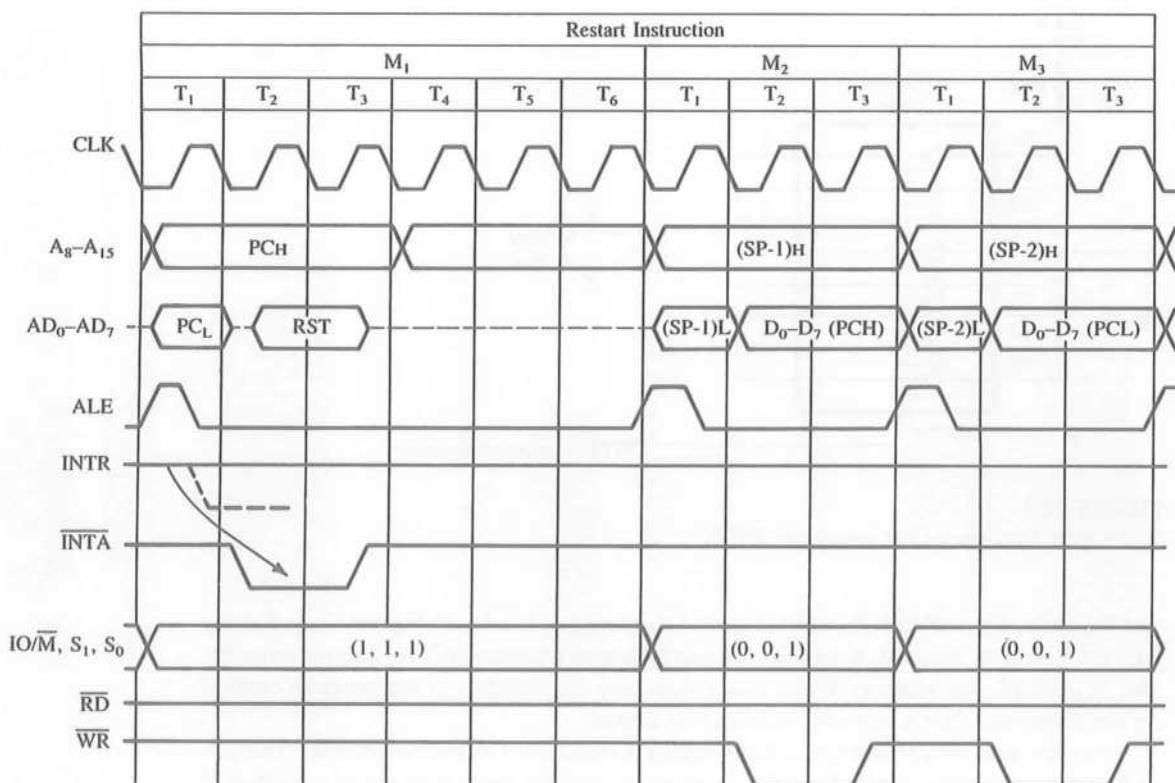


FIGURE 12.2

8085 Timing of the Interrupt Acknowledge Machine Cycle and Execution of an RST Instruction

SOURCE: Intel Corporation, *MCS 80/85 Student Study Guide* (Santa Clara, Calif.: Author, 1979), pp. 2-90.

0A	CALL DELAY	;Wait one second
0D	INR A	;Next count
0E	JMP NXTCNT	;Continue

Delay Routine Use delay subroutine illustrated in Chapter 9, Section 9.2.1.

Service Routine

Memory Address	Label	Mnemonics	Comments
XX70	SERV:	PUSH B	;Save contents
71		PUSH PSW	
72		MVI B,0AH	;Load register B for five flashes and ; five blanks

74		MVI A,00H	;Load 00 to blank display
76	FLASH:	OUT PORT1	
78		MVI C,01H	;Parameter for 1-second delay
7A		CALL DELAY	
7D		CMA	;Complement display count
7E		DCR B	;Reduce count
7F		JNZ FLASH	
82		POP PSW	
83		POP B	
84		EI	;Enable interrupt process
85		RET	;Service is complete; go back to ; main program

DESCRIPTION OF THE INTERRUPT PROCESS

1. The main program initializes the stack pointer at XX99H and enables the interrupts. The program will count continuously from 00H to FFH, with a delay of one second between each count.
2. To interrupt the processor, push the switch. The INTR line goes high.
3. Assuming the switch is pushed when the processor is executing the instruction OUT at memory location XX06H, the following sequence of events occurs:^{*}
 - a. The microprocessor completes the execution of the instruction OUT.
 - b. It senses that the line INTR is high, and that the interrupt is enabled.
 - c. The microprocessor disables the interrupt, stops execution, and sends out a control signal INTA (Interrupt Acknowledge).
 - d. The INTA (active low) enables the tri-state buffer, and the instruction EFH is placed on the data bus.
 - e. The microprocessor saves the address XX08H of the next instruction (MVI C,01H) on the stack at locations XX98H and XX97H, and the program is transferred to memory location 0028H. The locations 0028-29-2AH should have the following Jump instruction to transfer the program to the service routine.

JMP XX70H

(However, you do not have access to write at 0028H in the monitor program. See the next section.)

4. The program jumps to the service routine at XX70H.
5. The service routine saves the registers that are being used in the subroutine and loads the count ten in register B to output five flashes and also five blanks.
6. The service routine enables the interrupt before returning to the main program.
7. When the service routine executes the RET instruction, the microprocessor retrieves the memory address XX08H from the top of the stack and continues the binary counting.

^{*}It is assumed here that the Hold signal is inactive; Hold has a higher priority than any interrupt.

STACK OPERATION DURING THE INTERRUPT PROCESS

The stack is initialized at location XX99H, and the processor is interrupted when the instruction OUT PORT1 is being executed. After the processor acknowledges the interrupt request, and when it receives the RST 5 instruction (which is a 1-byte CALL), it places the address in the program counter (XX08H—the address of the next instruction, MVI C, 01H) on the stack, and the stack pointer is decremented by two locations to the address XX97H. The processor jumps to location 0028H and finds the Jump instruction to location XX70H, the address of the interrupt service routine. The service routine has two PUSH instructions; therefore, the processor stores registers BC, A, and Flags on the stack and decrements the address in the stack pointer to XX93H. The next subroutine call is CALL DELAY. The processor places registers DE, A, and Flags on the stack. It decrements the stack pointer by four locations to the address XX8FH. At the end of the DELAY subroutine, when registers A, Flags, and DE are retrieved by the POP instructions (see DELAY subroutine—Section 9.2), the stack pointer is incremented to XX93H. Again, at the end of the service routine, when registers A, Flags, and BC are retrieved by the POP instructions, the stack pointer is incremented to location XX97H. When the processor executes the last instruction, RET, it finds the address XX08H and transfers the program execution to the main program at location XX08H.

TESTING INTERRUPT ON A SINGLE-BOARD COMPUTER SYSTEM

Step 3e in the above description assumes that you are designing the system and have access to locations in EPROM or ROM on page 00H. In reality, you have no direct access to restart locations if the system has already been designed. Then how do you transfer the program control from a restart location to the service routine?

In single-board microcomputers, some restart locations are usually reserved for users, and the system designer provides a Jump instruction at a restart location to jump somewhere in R/W memory. For example, in Intel's SDK-85 system, R/W memory begins at page 20H, and you may find the following instruction in the monitor program at memory location 0028H:

0028 JMP 20C2H

If instruction RST 5 is inserted as shown in Figure 12.3, it transfers the program to location 0028H, and the monitor transfers the program from 0028H to location 20C2H. To implement the interrupt shown in Figure 12.3, you need to store the Jump instruction as shown below:

20C2	C3	JMP SERV
20C3	70	
20C4	20	

This instruction will transfer the program to the service routine located at 2070H.

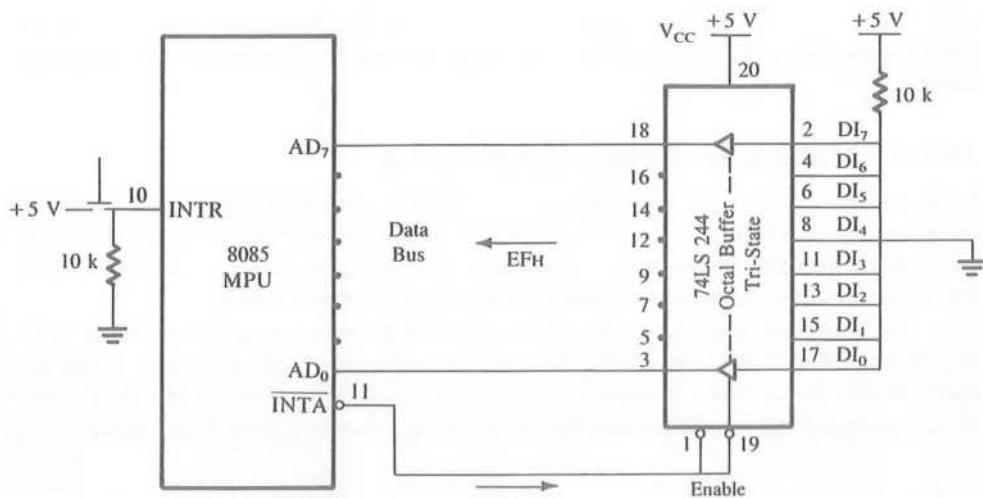


FIGURE 12.3

Schematic to Implement the 8085 Interrupt

ISSUES IN IMPLEMENTING INTERRUPTS

In the above illustration, some questions remain unanswered:

1. Is there a minimum pulse width required for the INTR signal?

The microprocessor checks INTR, one clock period before the last T-state of an instruction cycle. In the 8085, the Call instructions require 18 T-states; therefore, the INTR pulse should be high at least for 17.5 T-states. In a system with 3 MHz clock frequency (such as the SDK-85 system), the input pulse to INTR should be at least 5.8 μ s long.

2. How long can the INTR pulse stay high?

The INTR pulse can remain high until the interrupt flip-flop is set by the EI instruction in the service routine. If it remains high after the execution of the EI instruction, the processor will be interrupted again, as if it were a new interrupt. In Figure 12.3, the manual push button will keep the INTR high for more than 20 ms; however, the service routine has a delay of 1 second, and the EI instruction is executed at the end of the service routine.

3. Can the microprocessor be interrupted again before the completion of the first interrupt service routine?

The answer to this question is determined by the programmer. After the first interrupt, the interrupt process is automatically disabled. In the Illustrative Program in Section 12.1.2, the service routine enables the interrupt at the end of the service routine; in this case, the microprocessor cannot be interrupted before the completion of this routine. If in-

struction EI is written at the beginning of the routine, the microprocessor can be interrupted again during the service routine. (See Experimental Assignment 1 at the end of this chapter.)

12.1.3 Multiple Interrupts and Priorities

In the previous section, we illustrated how to implement the interrupt for one peripheral using one line (INTR). Now we will expand our discussion to include how to use INTR for multiple peripherals and how to determine priorities among these peripherals when two or more of the peripherals request interrupt service simultaneously.

The schematic in Figure 12.4 implements multiple interrupting devices using an 8-to-3 priority encoder that determines the priorities among interrupting devices. If you examine the instruction code for eight RST instructions, you will notice that bits D₅, D₄, and D₃ change in a binary sequence and that the others are always at logic 1 (see Table 12.1).

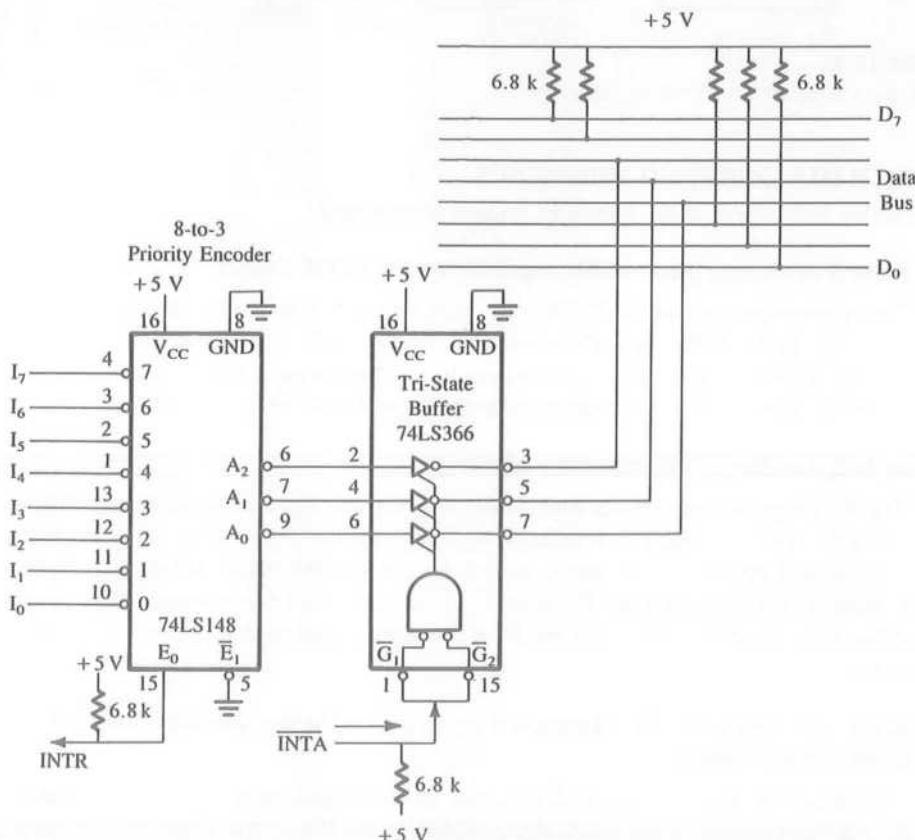


FIGURE 12.4
Multiple Interrupts Using a Priority Encoder

The encoder provides appropriate combinations on its output lines A_0 , A_1 , and A_2 , which are connected to data lines D_3 , D_4 , and D_5 through a tri-state buffer. The eight inputs to the encoder are connected to eight different interrupting devices.

When an interrupting device requests service, one of the input lines goes low, which makes line E_0 high and interrupts the microprocessor. When the interrupt is acknowledged and the signal INTA enables the tri-state buffer, the code corresponding to the input is placed on lines D_5 , D_4 , and D_3 . For example, if the interrupting device on line I_5 goes low, the output of the encoder will be 010. This code is inverted by the buffer 74LS366 and combined with other high data lines. Thus, the instruction 1110 1111 (EFH) is placed on the data bus. This is instruction RST 5. Similarly, any one of the RST instructions can be generated and placed on the data bus. If there are simultaneous requests, the priorities are determined by the encoder; it responds to the higher-level input, ignoring the lower-level input. One of the drawbacks of this scheme is that the interrupting device connected to the input I_7 always has the highest priority. The interrupt scheme shown in Figure 12.4 also can be implemented by using a special device called a Priority Interrupt Controller—8214. This device includes a status register and a priority comparator in addition to an 8-to-3 priority encoder. Today, however, this device is being replaced by a more versatile one called a Programmable Interrupt Controller—8259A (described briefly later in this chapter).

8085 VECTORED INTERRUPTS

12.2

The 8085 has five interrupt inputs (Figure 12.5). One is called INTR (discussed in the previous section), three are called RST 5.5, 6.5, and 7.5, respectively, and the fifth is called TRAP, a nonmaskable interrupt. These last four (RSTs and TRAP) are automatically vectored (transferred) to specific locations on memory page 00H without any external hardware. They do not require the INTA signal or an input port; the necessary hardware is already implemented inside the 8085. These interrupts and their call locations are as follows:

Interrupts		Call Locations
1. TRAP	→	0024H
2. RST 7.5	→	003CH
3. RST 6.5	→	0034H
4. RST 5.5	→	002CH

The TRAP has the highest priority, followed by RST 7.5, 6.5, 5.5, and INTR, in that order; however, the TRAP has a lower priority than the Hold signal used for DMA (Section 12.4.2).

12.2.1 TRAP

TRAP, a nonmaskable interrupt known as NMI, is analogous to the smoke detector described earlier. It has the highest priority among the interrupt signals, it need not be en-

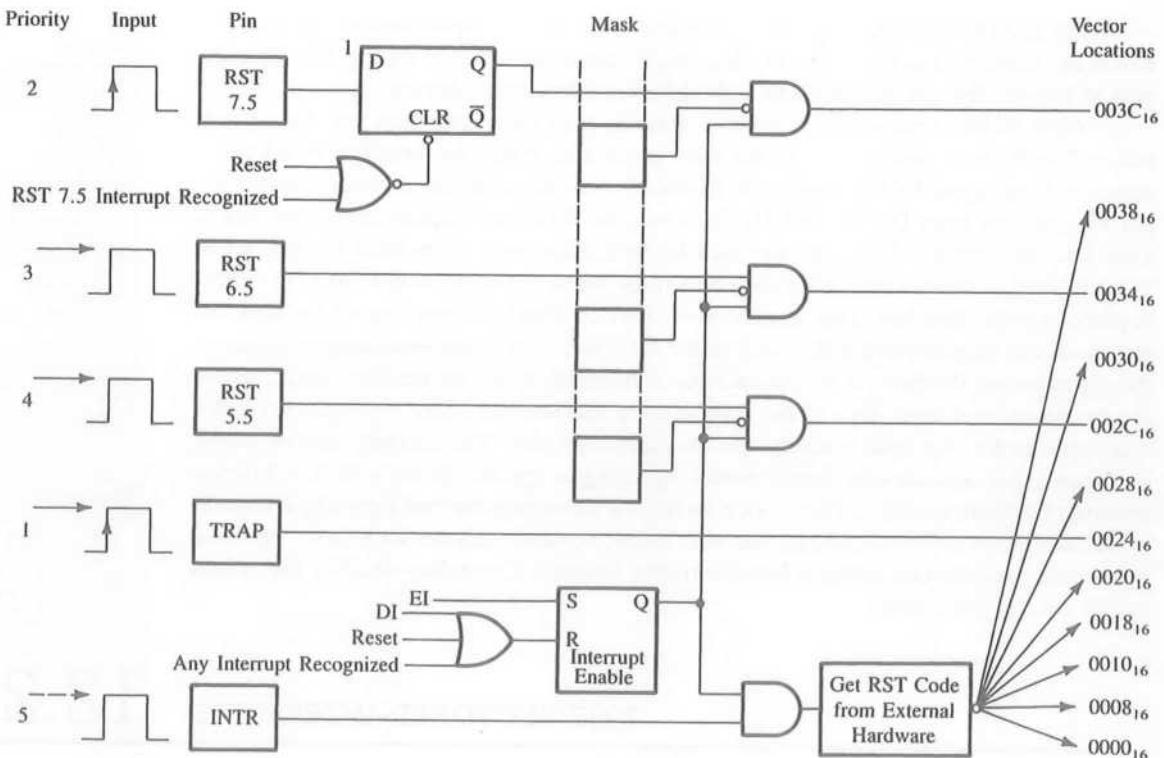


FIGURE 12.5

The 8085 Interrupts and Vector Locations

SOURCE: Intel Corporation, *MCS 80/85 Student Study Guide* (Santa Clara, Calif.: Author, 1979).

abled, and it cannot be disabled. It is level- and edge-sensitive, meaning that the input should go high and stay high to be acknowledged. It cannot be acknowledged again until it makes a transition from high to low to high.

Figure 12.5 shows that when this interrupt is triggered, the program control is transferred to location 0024H without any external hardware or the interrupt enable instruction EI. TRAP is generally used for such critical events as power failure and emergency shut-off.

12.2.2 RST 7.5, 6.5, and 5.5

These maskable interrupts (shown in Figure 12.5) are enabled under program control with two instructions: EI (Enable Interrupt) described earlier, and SIM (Set Interrupt Mask) described below:

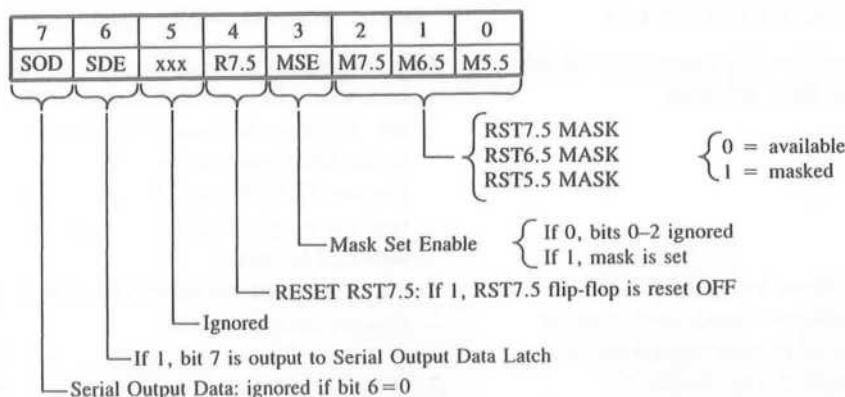


FIGURE 12.6

Interpretation of the Accumulator Bit Pattern for the SIM Instruction

SOURCE: Intel Corporation, *Assembly Language Programming Manual* (Santa Clara, Calif.: Author, 1979), pp. 3–59.

Instruction SIM: Set Interrupt Mask. This is a 1-byte instruction and can be used for three different functions (Figure 12.6).

- One function is to set mask for RST 7.5, 6.5, and 5.5 interrupts. This instruction reads the content of the accumulator and enables or disables the interrupts according to the content of the accumulator. **Bit D₃ is a control bit and should = 1 for bits D₀, D₁, and D₂ to be effective.** Logic 0 on D₀, D₁, and D₂ will enable the corresponding interrupts, and logic 1 will disable the interrupts.
- The second function is to reset RST 7.5 flip-flop (Figure 12.6). Bit D₄ is additional control for RST 7.5. If D₄ = 1, RST 7.5 is reset. This is used to override (or ignore) RST 7.5 without servicing it.
- The third function is to implement serial I/O (discussed in Chapter 16). Bits D₇ and D₆ of the accumulator are used for serial I/O and do not affect the interrupts. Bit D₆ = 1 enables the serial I/O and bit D₇ is used to transmit (output) bits.

Here we are concerned with RST 7.5, 6.5, and 5.5 interrupts and not with serial I/O.

The mnemonic SIM is confusing. **The wording—Set Interrupt Mask—implies that the instruction masks the interrupts.** However, **the instruction must be executed in order to use the interrupts.** The process required to enable these interrupts can be likened to a switchboard controlling three telephone extensions in a company. Let us assume these phone extensions are assigned to the president (RST 7.5), the vice president (RST 6.5), and the manager (RST 5.5), in that priority, and are monitored by their receptionist according to the instructions given. The protocols of placing a telephone call to one of the executives and of interrupting the microprocessor using RST 7.5, 6.5, and 5.5 can be compared as follows:

Placing a Telephone Call

1. The switchboard is functional and all telephone lines are open.
2. All executives leave instructions on the receptionist's desk as to whether they wish to receive any phone calls.
3. The receptionist reads the instructions.
4. The receptionist is on duty and sends calls through for whoever is available.
5. The receptionist is busy typing.
Phone calls can be received directly according to previous instructions.
6. No calls for the president now. Call back later.

This analogy can be extended to the interrupt INTR, which is viewed as one telephone line shared by eight engineers with a switchboard operator (external hardware) who rings the appropriate extension.

The entire interrupt process (except TRAP) is disabled by resetting the Interrupt Enable flip-flop (Figure 12.5). The flip-flop can be reset in one of the three ways:

- Instruction DI
- System Reset
- Recognition of an Interrupt Request

Figure 12.5 shows that these three signals are ORed and the output of the OR gate is used to reset the flip-flop.

TRIGGERING LEVELS

These interrupts are sensitive to different types of triggering as listed below:

- RST 7.5** This is positive-edge sensitive and can be triggered with a short pulse. The request is stored internally by the D flip-flop (Figure 12.5) until the microprocessor responds to the request or until it is cleared by Reset or by bit D₄ in the SIM instruction.
- RST 6.5** and **RST 5.5** These interrupts are level-sensitive, meaning that the triggering level should be on until the microprocessor completes the execution of the current instruction. If the microprocessor is unable to respond to these requests immediately, they should be stored or held by external hardware.

Interrupting the 8085 (Figure 12.6)

1. The interrupt process is enabled. The instruction EI sets the Interrupt Enable flip-flop, and one of the inputs to the AND gates is set to logic 1 (Figure 12.5). These AND gates activate the program transfer to various vectored locations.
2. An appropriate bit pattern is loaded into the accumulator.
3. If bit D₃ = 1, the respective interrupts are enabled according to bits D₂–D₀.
4. RST 7.5, 6.5, and 5.5 are being monitored.
5. If bit D₃ = 0, bits D₂–D₀ have no effect on previous conditions.
6. Bit D₄ = 1; this resets RST 7.5.

Enable all the interrupts in an 8085 system.

Example
12.1

Instructions

```
EI          ;Enable interrupts
MVI A,08H   ;Load bit pattern to enable RST 7.5, 6.5, and 5.5
SIM         ;Enable RST 7.5, 6.5, and 5.5
```

Bit D₃ = 1 in the accumulator makes the instruction SIM functional, and bits D₂, D₁, and D₀ = 0 enable the interrupts 7.5, 6.5, and 5.5.

Reset the 7.5 interrupt from Example 12.1.

Example
12.2

Instructions

```
MVI A,18H   ;Set D4 = 1
SIM         ;Reset 7.5 interrupt flip-flop
```

PENDING INTERRUPTS

Because there are several interrupt lines, when one interrupt request is being served, other interrupt requests may occur and remain pending. The 8085 has an additional instruction called RIM (Read Interrupt Mask) to sense these pending interrupts.

Instruction RIM: Read Interrupt Mask. This is a 1-byte instruction that can be used for the following functions.

- To read interrupt masks. This instruction loads the accumulator with 8 bits indicating the current status of the interrupt masks (Figure 12.7).
- To identify pending interrupts. Bits D₄, D₅, and D₆ (Figure 12.7) identify the pending interrupts.
- To receive serial data. Bit D₇ (Figure 12.7) is used to receive serial data.

Assuming the microprocessor is completing an RST 7.5 interrupt request, check to see if RST 6.5 is pending. If it is pending, enable RST 6.5 without affecting any other interrupts; otherwise, return to the main program.

Example
12.3

Instructions

```
RIM          ;Read interrupt mask
MOV B,A     ;Save mask information
ANI 20H     ;Check whether RST 6.5 is pending
```

```

JNZ NEXT
EI
RET      ;RST 6.5 is not pending, return to main program
NEXT:   MOV A,B ;Get bit pattern; RST 6.5 is pending
        ANI 0DH ;Enables RST 6.5 by setting D1 = 0
        ORI 08H ;Enable SIM by setting D3 = 1
        SIM
        JMP SERV ;Jump to service routine for RST 6.5

```

The instruction RIM checks for a pending interrupt. Instruction ANI 20H masks all the bits except D₅ to check pending RST 6.5. If D₅ = 0, the program control is transferred to the main program. D₅ = 1 indicates that RST 6.5 is pending. Instruction ANI 0DH sets D₁ = 0 (RST 6.5 bit for SIM), instruction ORI sets D₃ = 1 (this is necessary for SIM to be effective), and instruction SIM enables RST 6.5 without affecting any other interrupts. The JMP instruction transfers the program to the service routine (SERV) written for RST 6.5.

The RIM instruction loads the accumulator with the following information:

7	6	5	4	3	2	1	0
SID	17.5	16.5	15.5	IE	M7.5	M6.5	M5.5

FIGURE 12.7

Interpretation of the Accumulator Bit Pattern for the RIM Instruction

SOURCE: Intel Corporation, *Assembly Language Programming Manual* (Santa Clara, Calif.: Author, 1979), pp. 3-49.

12.2.3 Illustration: Interrupt-Driven Clock

PROBLEM STATEMENT

Design a 1-minute timer using a 60 Hz power line as an interrupting source. The output ports should display minutes and seconds in BCD. At the end of the minute, the output ports should continue displaying one minute and zero seconds.

HARDWARE DESCRIPTION

This 1-minute timer is designed with a 60 Hz AC line. The circuit (Figure 12.8) uses a step-down transformer, the 74121 monostable multivibrator, and interrupt pin RST 6.5. After the interrupt, program control is transferred to memory location 0034H in the monitor program.

The AC line with 60 Hz frequency has a period of 16.6 ms; that means it can provide a pulse every sixtieth of a second with 8.3 ms pulse width, which is too long for the interrupt. The interrupt flip-flop is enabled again within 6 µs in the timer service routine;

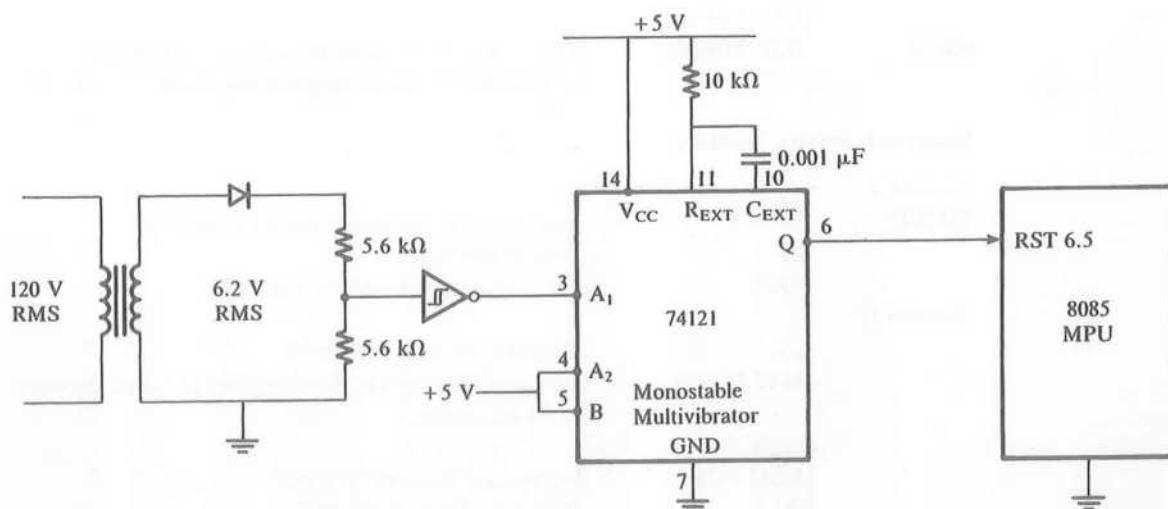


FIGURE 12.8

Schematic of Interrupt-Driven Timer Clock

therefore, the pulse should be turned off before the EI instruction in the service routine is executed. The 74121 monostable multivibrator is used to provide appropriate pulse width. Another option is to use the 7474 positive-edge triggered flip-flop (see Problem 4 at the end of this chapter).

Monitor Program

0034	JMP RWM	;This is RST 6.5; go to location in user memory ; to give Restart access to the user
------	---------	---

Main Program

	LXI SP,STACK	;Initialize stack pointer
	RIM	;Read mask
	ORI 08H	;Bit pattern to enable RST 6.5
	SIM	;Enable RST 6.5
	LXI B,0000H	;Set up register B for minutes and register C for ; seconds
	MVI D,3CH	;Set up register D to count 60_{10} interrupts
	EI	;Enable interrupts
DISPLAY:	MOV A,B	
	OUT PORT1	;Display minutes at PORT1
	MOV A,C	
	OUT PORT2	;Display seconds at PORT2

	JMP DSPLAY	
RWM:	JMP TIMER	;This is RST 6.5 vector location 0034H; go to ; ; TIMER routine to upgrade the clock
		Interrupt Service Routine
	;Section I	
TIMER:	DCR D	;One interrupt occurred; reduce count by 1
	EI	;Enable interrupts
	RNZ	;Has 1 second elapsed? If not, return
	;Section II	
	DI	;No other interrupts allowed
	MVI D,3CH	;1 second is complete; load register D again to count ; ; 60 interrupts
	MOV A,C	
	ADD 01H	;Increment "Second" register
	DAA	;Decimal-adjust "Seconds"
	MOV C,A	;Save "BCD" seconds
	CPI 60H	
	EI	
	RNZ	;Is time = 60 seconds? If not, return
	;Section III	
	DI	;Disable interrupts
	MVI C,00H	;60 seconds complete, clear "Second" register
	INR B	;Increment "Minutes"
	RET	;1 minute elapsed

PROGRAM DESCRIPTION

The main program clears registers B and C to store minutes and seconds, respectively; enables the interrupts; sets up register D to count 60 interrupts; and displays the starting time in minutes (00) and seconds (00). Instruction SIM enables RST 6.5 according to the bit pattern in the accumulator.

When the first pulse interrupts the processor, program control is transferred to memory location 0034H, as mentioned earlier. (Check location 0034H in your monitor program; you may find a Jump instruction to transfer the control to a memory location in R/W memory. Write a Jump instruction at that location to locate the service routine labeled TIMER.)

In the service routine (Section I), register D is decremented every second, the interrupt is enabled, and the program is returned to the main routine. This is repeated 60 times. After the sixtieth interrupt, counter D goes to zero and the program enters Section II. In this section, counter D is reloaded, the "second" register is incremented and adjusted for BCD, and the program is returned to the main routine. In this section, instruction DI is used as a precaution to avoid any interrupts from other sources. For the next 60 interrupts, the program remains in Section I. When Section II is repeated 60 times, the program goes to Section III, where the "minute" register is incremented and the "second" register is

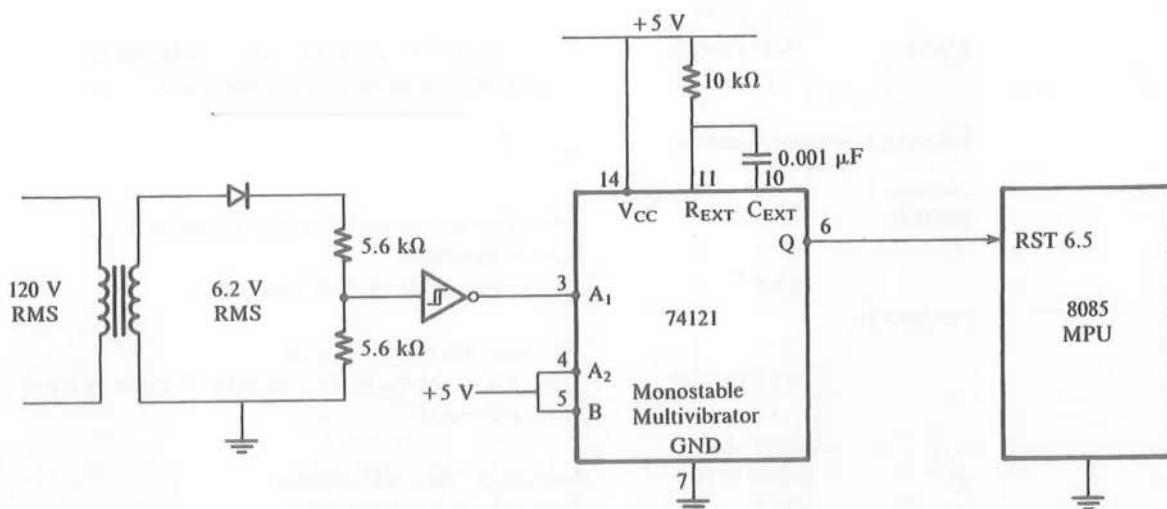


FIGURE 12.8

Schematic of Interrupt-Driven Timer Clock

therefore, the pulse should be turned off before the EI instruction in the service routine is executed. The 74121 monostable multivibrator is used to provide appropriate pulse width. Another option is to use the 7474 positive-edge triggered flip-flop (see Problem 4 at the end of this chapter).

Monitor Program

0034	JMP RWM	;This is RST 6.5; go to location in user memory ; to give Restart access to the user
------	---------	---

Main Program

	LXI SP,STACK	;Initialize stack pointer
	RIM	;Read mask
	ORI 08H	;Bit pattern to enable RST 6.5
	SIM	;Enable RST 6.5
	LXI B,0000H	;Set up register B for minutes and register C for ; seconds
	MVI D,3CH	;Set up register D to count 60_{10} interrupts
	EI	;Enable interrupts
DISPLAY:	MOV A,B	
	OUT PORT1	;Display minutes at PORT1
	MOV A,C	
	OUT PORT2	;Display seconds at PORT2

	JMP DSPLAY	
RWM:	JMP TIMER	;This is RST 6.5 vector location 0034H; go to ; ; TIMER routine to upgrade the clock
	Interrupt Service Routine	
	;Section I	
TIMER:	DCR D	;One interrupt occurred; reduce count by 1
	EI	;Enable interrupts
	RNZ	;Has 1 second elapsed? If not, return
	;Section II	
	DI	;No other interrupts allowed
	MVI D,3CH	;1 second is complete; load register D again to count ; ; 60 interrupts
	MOV A,C	
	ADD 01H	;Increment "Second" register
	DAA	;Decimal-adjust "Seconds"
	MOV C,A	;Save "BCD" seconds
	CPI 60H	
	EI	
	RNZ	;Is time = 60 seconds? If not, return
	;Section III	
	DI	;Disable interrupts
	MVI C,00H	;60 seconds complete, clear "Second" register
	INR B	;Increment "Minutes"
	RET	;1 minute elapsed

PROGRAM DESCRIPTION

The main program clears registers B and C to store minutes and seconds, respectively; enables the interrupts; sets up register D to count 60 interrupts; and displays the starting time in minutes (00) and seconds (00). Instruction SIM enables RST 6.5 according to the bit pattern in the accumulator.

When the first pulse interrupts the processor, program control is transferred to memory location 0034H, as mentioned earlier. (Check location 0034H in your monitor program; you may find a Jump instruction to transfer the control to a memory location in R/W memory. Write a Jump instruction at that location to locate the service routine labeled TIMER.)

In the service routine (Section I), register D is decremented every second, the interrupt is enabled, and the program is returned to the main routine. This is repeated 60 times. After the sixtieth interrupt, counter D goes to zero and the program enters Section II. In this section, counter D is reloaded, the "second" register is incremented and adjusted for BCD, and the program is returned to the main routine. In this section, instruction DI is used as a precaution to avoid any interrupts from other sources. For the next 60 interrupts, the program remains in Section I. When Section II is repeated 60 times, the program goes to Section III, where the "minute" register is incremented and the "second" register is

cleared. To avoid further interrupts, the interrupt is disabled and the program is returned to the main routine where one minute and zero seconds are displayed continuously.

In this particular program, the service routine does not save any register contents by using PUSH instructions before starting the service routine. However, in most service routines, register contents must be saved because the interrupt is asynchronous and can occur at any time.

In Section 12.2.3: "Illustration: Interrupt Driven Clock," assume that various segments of the program are assigned the memory locations as follows: (a) the Main Program begins at location 2000H, (b) the Interrupt Service Routine (ISR) begins at location 2050H, and (c) RWM = 20C8H. Answer the following questions:

Example
12.4

1. Specify the memory location where the processor is directed when it acknowledges an interrupt request.
 2. Specify the contents of the memory locations 0034H, 35, and 36H, and how the codes are written in these locations.
 3. Specify the contents of memory locations 20C8H, C9H, and CAH, and how the codes are written in these locations.
 4. Specify the memory location where the program returns after the execution of instructions RNZ or RET.
-
1. The interrupting signal is connected to RST 6.5; therefore, the processor is directed to location 0034H.
 2. The contents of memory locations in Hex: 0034 = C3, 0035 = C8, and 0036 = 20. These codes are part of the Jump instruction directing the program to location 20C8H. These codes must be written by the system (trainer) designer because the user does not have write access to these EPROM or ROM locations. The memory locations starting at 0000 must be assigned to some system program such as a monitor program.
 3. The contents of memory locations in Hex: 20C8 = C3, 20C9 = 50, and 20CA = 20. These codes are part of the Jump instruction directing the program to the ISR location 2050H and must be entered by the user.
 4. The processor will be interrupted at various locations in the main program; therefore, the processor returns to the address of the next instruction where it is interrupted.

Solution

RESTART AS SOFTWARE INSTRUCTIONS

12.3

External hardware is necessary to insert an RST instruction when an interrupt is requested to INTR. However, the fact that RST is a software instruction is quite often overlooked or misunderstood. RST instructions are commonly used to set up software breakpoints as a debugging technique. A breakpoint is a Restart (RST) instruction in a program where the execution of the program stops temporarily and program control is transferred to the RST location. The program should be transferred from the RST location to the

breakpoint service routine to allow the user to examine register or memory contents when specified keys are pressed. After the breakpoint routine, the program should return to executing the main program at the breakpoint. The breakpoint procedure allows the user to test programs in segments. For example, if RST 6 is written in a program, the program execution is transferred to location 0030H; it is equivalent to a 1-byte call instruction. This can be used to write a software breakpoint routine, as illustrated next.

12.3.1 Illustrative Program: Implementation of Breakpoint Technique

PROBLEM STATEMENT

Implement a breakpoint facility at RST 5 for user. When the user writes RST 5 in the program, the program should

1. be interrupted at the instruction RST 5.
2. display the accumulator content and the flags when Hex key A (1010_2) is pressed.
3. exit the breakpoint routine and continue execution when the Zero key (0000_2) is pressed.

Assume that when a keyboard routine (KBRD) is called, it returns the binary key code of the key pressed in the accumulator.

PROBLEM ANALYSIS

The breakpoint routine should display the accumulator contents and the flags when a user writes the RST instruction in a program. The technique used to display register contents after executing a segment of the user's program is as follows:

1. Store the register contents on the stack.
2. Assign (arbitrarily) a key from the keyboard for the accumulator display. (In this problem Hex key A from the keyboard is assigned to display the accumulator contents.)
3. Wait for the key to be pressed, and retrieve the contents from the stack by manipulating the stack pointer when the key is pressed.
4. Assign a key to return to the user's program. (In this problem, it is the Zero key.)

This approach assumes that a keyboard subroutine can be called from your monitor program and that the codes associated with each key are known.

BREAKPOINT SUBROUTINE

;BRKPNT: This is a breakpoint subroutine; it can be implemented with the instruction
; RST 5. It displays the accumulator and the flags when the A key is pressed
; and returns to the calling program when the Zero key is pressed

;Input: None

;Output: None

;Does not modify any register contents

;Calls: KBRD subroutine. The KBRD is a keyboard subroutine that checks a key
pressed.

```

; The routine identifies the key and places its binary code in the accumulator
BRKPNT: PUSH PSW      ;Save registers
        PUSH B
        PUSH D
        PUSH H
KYCHK:  CALL KBRD    ;Check for a key
        CPI 0AH     ;Is it key A?
        JNZ RETKY   ;If not, check Zero key
        LXI H,0007H  ;Load stack pointer displacement count; see program
                      ; description
        DAD SP      ;Place memory address in HL, where (A) is stored
        MOV A,M
        OUT PORT1   ;Display accumulator contents
        DCX H       ;Point HL to the location of the flags
        MOV A,M
        OUT PORT2   ;Display flags
        JMP KYCHK   ;Go back and check next key
RETKY:  CPI 00H     ;Is it Zero key?
        JNZ KYCHK   ;If not, go and check key program
        POP H       ;Retrieve registers
        POP D
        POP B
        POP PSW
        RET

```

PROGRAM DESCRIPTION

The breakpoint routine saves all the registers on the stack, and the address in the stack pointer is decremented accordingly. (In this particular problem, registers BC and DE need not be saved. These registers are saved here for the assignments given at the end of the chapter.) The accumulator contents are stored in the seventh memory location from the top of the stack, and the flags in the sixth memory location.

When key A is pressed, the HL register adds seven to the stack pointer contents and places that address in the HL register (DAD SP), without modifying the contents of the stack pointer. This is an important point; if the stack pointer is varied, appropriate contents may not be retrieved with POP and RET instructions.

The subroutine displays the accumulator and the flags at the two output ports and returns to the main program.

12.4 ADDITIONAL I/O CONCEPTS AND PROCESSES

The 8085 interrupt I/O, described earlier, is limited because of its single interrupt pin and hardware requirements to determine interrupt priorities. To circumvent these limitations, a programmable interrupt controller such as the 8259A is used to implement and extend

the capability of the 8085 interrupt. Another I/O process, Direct Memory Access (DMA), is commonly used for high-speed data transfer. This I/O process is implemented also by using a programmable device such as DMA controller 8257. A programmable device is generally a multifunction chip, and the microprocessor can specify and/or modify its functions by writing appropriate bits in the control register of the device. The concepts of programmable devices are discussed in Chapter 14. The intent here is to maintain the continuity in the discussion of the interrupt I/O and to introduce the concept of Direct Memory Access (DMA). The interrupt controller and the DMA process will be described briefly in the next sections—see Chapter 15 for further details.

12.4.1 Programmable Interrupt Controller: The 8259A

The 8259A is a programmable interrupt-managing device, specifically designed for use with the interrupt signals (INTR/INT) of the 8085 microprocessor. The primary features of the 8259A are as follows:

1. It manages eight interrupt requests.
2. It can vector an interrupt request anywhere in the memory map through program control without additional hardware for restart instructions. However, all eight requests are spaced at the interval of either four locations or eight locations.
3. It can solve eight levels of interrupt priorities in a variety of modes.
4. With additional 8259A devices, the priority scheme can be expanded to 64 levels.

One of the major limitations of the 8085 interrupt scheme is that all requests are vectored to memory locations on page 00H, which is reserved for ROM or EPROM, and access to these locations is difficult after a system has been designed. In addition, the process of determining priorities is limited, and extra hardware is required to insert Restart instructions. The 8259A overcomes these limitations and provides many more flexible options. It can be employed with such 16-bit Intel microprocessors as the 8086/8088 as well.

The 8259A block diagram (Figure 12.9) includes control logic, registers for interrupt requests, priority resolver, cascade logic, and data bus. The registers manage interrupt requests; the priority resolver determines their priority. The cascade logic is used to connect additional 8259A devices.

8259A INTERRUPT OPERATION

Implementing interrupts in the simplest format without cascading requires two specific instructions. The instructions are written by the MPU in the device registers (explained in Chapter 15). The first instruction specifies features such as mode and/or memory space between two consecutive interrupt levels. The second instruction specifies high-order memory address. After these instructions have been written, the following sequence of events should occur:

1. One or more interrupt request lines go high requesting the service.
2. The 8259A resolves the priorities and sends an INT signal to the MPU.

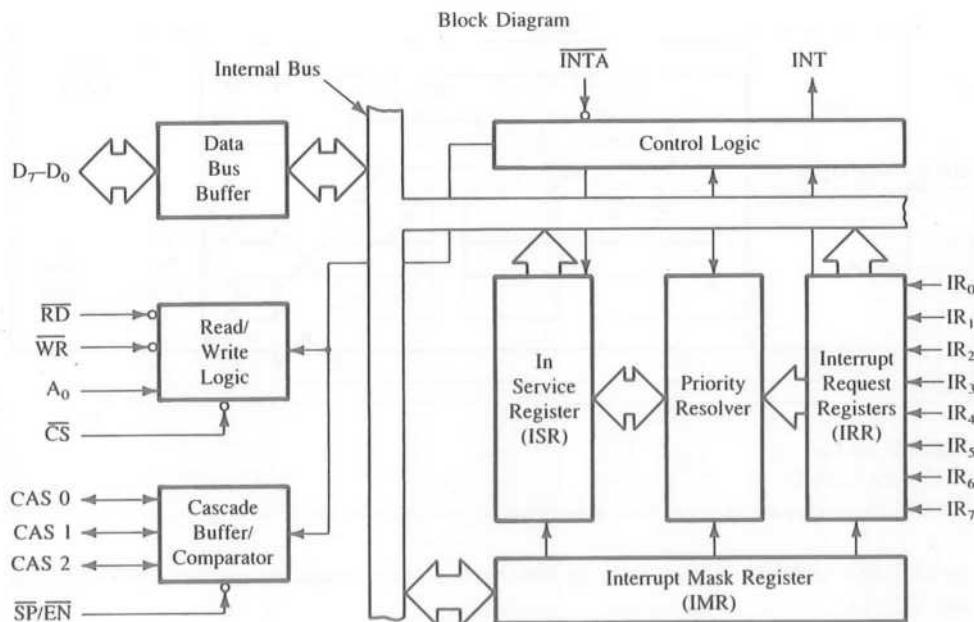


FIGURE 12.9

The 8259A Block Diagram

SOURCE: Intel Corporation, *Peripheral Components* (Santa Clara, Calif.: Author, 1993), pp. 3–171.

3. The MPU acknowledges the interrupt by sending INTA.
4. After the INTA has been received, the opcode for the CALL instruction (CDH) is placed on the data bus.
5. Because of the CALL instruction, the MPU sends two more INTA signals.
6. At the first INTA, the 8259A places the low-order 8-bit address on the data bus and, at the second INTA, it places the high-order 8-bit address of the interrupt vector. This completes the 3-byte CALL instruction.
7. The program sequence of the MPU is transferred to the memory location specified by the CALL instruction.

The 8259A includes additional features such as reading the status and changing the interrupt mode during a program execution.

12.4.2 Direct Memory Access (DMA)

The Direct Memory Access (DMA) is a process of communication or data transfer controlled by an external peripheral. In situations in which the microprocessor-controlled data transfer is too slow, the DMA is generally used; e.g., data transfer between a floppy disk and R/W memory of the system.

The 8085 microprocessor has two pins available for this type of I/O communication: HOLD (Hold) and HLDA (Hold Acknowledge). Conceptually, this is an im-

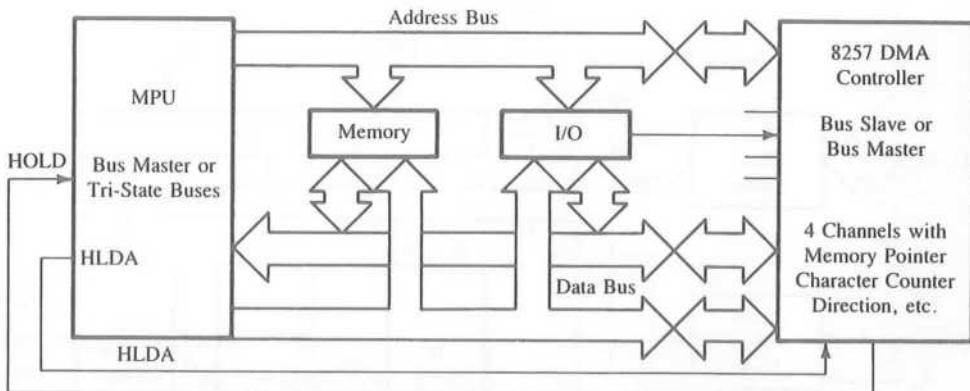


FIGURE 12.10
DMA Data Transfer

SOURCE: Intel Corporation, *MCS 80/85 Student Study Guide* (Santa Clara, Calif.: Author, 1979), pp. 2-21.

portant I/O technique; it introduces two new signals available on the 8085—HOLD and HLDA.

- HOLD—Hold. This is an active high input signal to the 8085 from another master requesting the use of the address and data buses. After receiving the HOLD request, the MPU relinquishes the buses in the following machine cycle. All buses are tri-stated and a Hold Acknowledge (HLDA) signal is sent out. The MPU regains the control of buses after HOLD goes low.
- HLDA—Hold Acknowledge. This is an active high output signal indicating that the MPU is relinquishing the control of the buses.

Typically, an external peripheral such as a DMA controller sends a request—a high signal—to the HOLD pin (Figure 12.10). The processor completes the execution of the current machine cycle; floats (high impedance state) the address, the data, and the control lines; and sends the Hold Acknowledge (HLDA) signal. **The DMA controller takes control of the buses and transfers data directly between source and destination, thus by-passing the microprocessor. At the end of data transfer, the controller terminates the request by sending a low signal to the HOLD pin, and the microprocessor regains control of the buses.** Typically, DMA controllers are programmable LSI chips. One such chip, the Intel 8237, is described in Chapter 15.

SUMMARY

The 8085 interrupts and their requirements are listed in Table 12.2 in the order of their priority; TRAP has the highest priority and INTR has the lowest priority. It must be emphasized that an interrupt can be recognized only if the HOLD signal is inactive. Some of the important features of the 8085 interrupts are summarized in Table 12.2.

TABLE 12.2
Summary of Interrupts in 8085

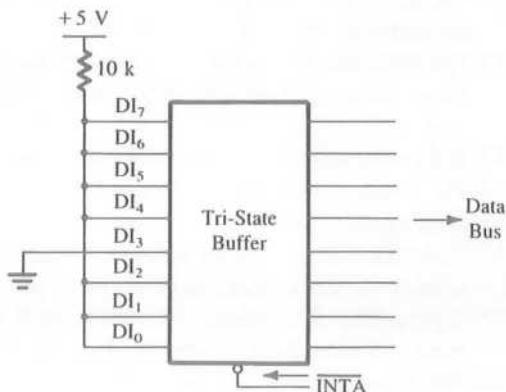
Interrupts	Type	Instructions	Hardware	Trigger	Vector
TRAP	Nonmaskable	<input type="checkbox"/> Independent of EI and DI	No external hardware	Level- and Edge-sensitive	0024H
RST 7.5	Maskable	<input type="checkbox"/> Controlled by EI and DI <input type="checkbox"/> Unmasked by SIM	No external hardware	Edge-sensitive	003CH
RST 6.5	Maskable	<input type="checkbox"/> Controlled by EI and DI <input type="checkbox"/> Unmasked by SIM	No external hardware	Level-sensitive	0034H
RST 5.5	Maskable	<input type="checkbox"/> Controlled by EI and DI <input type="checkbox"/> Unmasked by SIM	No external hardware	Level-sensitive	002CH
INTR 8085	Maskable	<input type="checkbox"/> Controlled by EI and DI	RST code from external hardware	Level-sensitive	038H ↑ 0000H

- The interrupt is an asynchronous process of communication with the microprocessor, initiated by an external peripheral.
- The 8085 has a maskable interrupt that can be enabled or disabled using the instructions EI and DI, respectively. This interrupt is labeled here as an 8085 nonvectored interrupt.
- The 8085 has eight RST instructions that are equivalent to 1-byte Calls to specific locations on memory page 00H.
- For the nonvectored interrupt, the RST instructions (0 to 7) are implemented using external hardware and the INTA signal.
- The 8085 has four additional interrupt inputs, one nonmaskable and three maskable. These three interrupts are implemented without any external hardware and are known as RST 7.5, 6.5, and 5.5.
- The instruction SIM is necessary to implement the interrupts 7.5, 6.5, and 5.5.
- The instruction RIM can be used to check whether any interrupt requests are pending.
- The RST instructions are software commands and can be used in a program to jump to their vectored locations on memory page 00H.
- A programmable interrupt controller such as the 8259A is used commonly to implement and extend the capability of the 8085 interrupt.
- The Direct Memory Access (DMA) is a process of high-speed data transfer under the control of external devices such as a DMA controller.

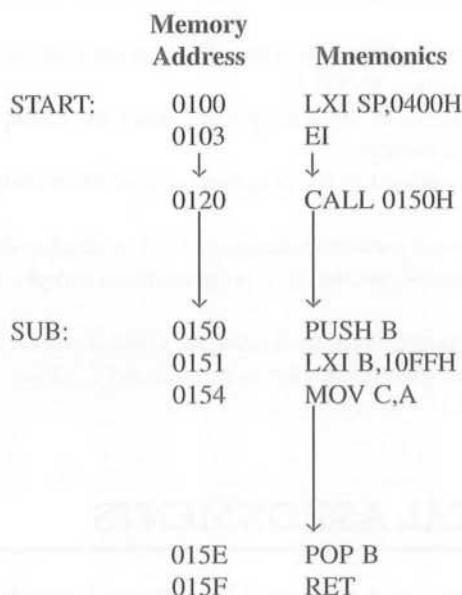
QUESTIONS, PROBLEMS, AND PROGRAMMING ASSIGNMENTS

1. Check whether the following statements are true or false.
 - a. If the 8085 microprocessor is interrupted while executing a 3-byte instruction (assuming the interrupt is enabled), the processor will acknowledge the interrupt request immediately, even before the completion of the instruction. (T/F)
 - b. When an 8085 system is Reset, all the interrupts including the TRAP are disabled. (T/F)
 - c. When the 8085 microprocessor acknowledges an interrupt, it disables the interrupt system (except TRAP). (T/F)
 - d. Instruction EI (Enable Interrupt) is necessary to implement the TRAP interrupt, but external hardware and the SIM instruction are unnecessary. (T/F)
 - e. If instruction RST 4 is written in a program, the program will jump to location 0020H without any external hardware. (T/F)
 - f. If a DMA request is sent to the microprocessor with a high signal to the HOLD pin, the microprocessor acknowledges the request after completing the present cycle. (T/F)
 - g. Instruction RIM is used to disable the interrupts 7.5, 6.5, and 5.5. (T/F)
 - h. The execution of instructions MVI A,10H, and SIM will enable all three interrupts (7.5, 6.5, and 5.5). (T/F)
2. a. Identify the RST instruction in Figure 12.11.
 b. Specify the restart memory location when the microprocessor is interrupted.
 c. If the instruction in the monitor program at 0030 is JMP 20BFH and the service routine is written at 2075H, what instruction is necessary (at 20BFH) to locate the service routine?
3. The main program is stored beginning at 0100H. The main program (at 0120H) has called the subroutine at 0150H, and when the microprocessor is executing the

FIGURE 12.11
Schematic for an Interrupt



instruction at location 0151 (LXI), it is interrupted. Read the program, then answer the questions that follow:



- a. Specify the contents of stack location 03FFH after the CALL instruction.
 - b. Specify the stack locations where the contents of registers B and C are stored.
 - c. When the program is interrupted, what is the memory address stored on the stack?
4. Redraw the schematic in Figure 12.8 with the following changes, and modify the service routine accordingly.
 - a. Replace monostable multivibrator 74121 by positive-edge trigger flip-flop 7474.
 - b. Design an output port, and use bit D₇ to clear the flip-flop.
 - c. Modify Section I of the timer routine to clear the flip-flop at an appropriate place.
 5. Answer the following questions in reference to Figure 12.4.
 - a. What is the instruction placed on the data bus when input line I₆ of the encoder goes low, thus requesting the interrupt service?
 - b. If three input lines (I₂, I₄, and I₅) go low simultaneously, explain how the priority is determined among the three requests, and specify the instruction that is placed on the data bus.
 6. A program is stored in memory from 2000H to 205FH. To check the first segment of the program up to location 2025H, a breakpoint routine call is inserted at location 2026H. (Refer to the Illustrative Program in Section 12.3.1 for the breakpoint subroutine.) If the stack pointer is initialized at 2099H, answer the following questions.

- a. Specify the contents of memory locations 2098H and 2097H.
 - b. Specify the memory locations where the accumulator contents and the flags are stored when the microprocessor executes instruction PUSH PSW in the breakpoint routine.
 - c. Specify the memory locations where HL register contents are stored after executing the instruction PUSH H.
 - d. Specify the contents of the stack pointer when the breakpoint routine returns from the KBRD routine.
 - e. What address is placed in the program counter when instruction RET is executed?
7. Modify the breakpoint routine in Section 12.3.1 to display the memory location where the breakpoint is inserted in a program (for example, location 2026H in Question 6a).
 8. Modify the breakpoint routine to display the contents of the BC, DE, and HL registers when the user pushes the Hex keys 1, 2, and 3. (The respective Hex codes are 01, 02, and 03.)

EXPERIMENTAL ASSIGNMENTS

1.
 - a. Build the circuit shown in Figure 12.3. Enter and execute the program given in the illustration.
 - b. Verify the interrupt process by pushing the Interrupt key.
 - c. Replace the instruction EI at location XX03 by the NOP instruction. Push the Interrupt key and verify whether an interrupt request can be accepted by the microprocessor.
 - d. Interrupt the processor; when the processor is in the middle of the service routine, push the Interrupt key again. Explain why the processor does not accept any interrupts during the service routine.
 - e. In the routine SERV, write instruction EI at the beginning of the service routine. Push the Interrupt key, and explain your observation. (You may notice interesting results because the manual key keeps INTR high too long.)
2.
 - a. Build the circuit shown in Figure 12.8 and implement the interrupt-driven clock.
 - b. Rewrite the program to simulate a 5-minute egg timer.
 - c. Modify the program in Experimental Assignment 2a to flash FF with some appropriate delay to indicate the completion of the 5-minute period.