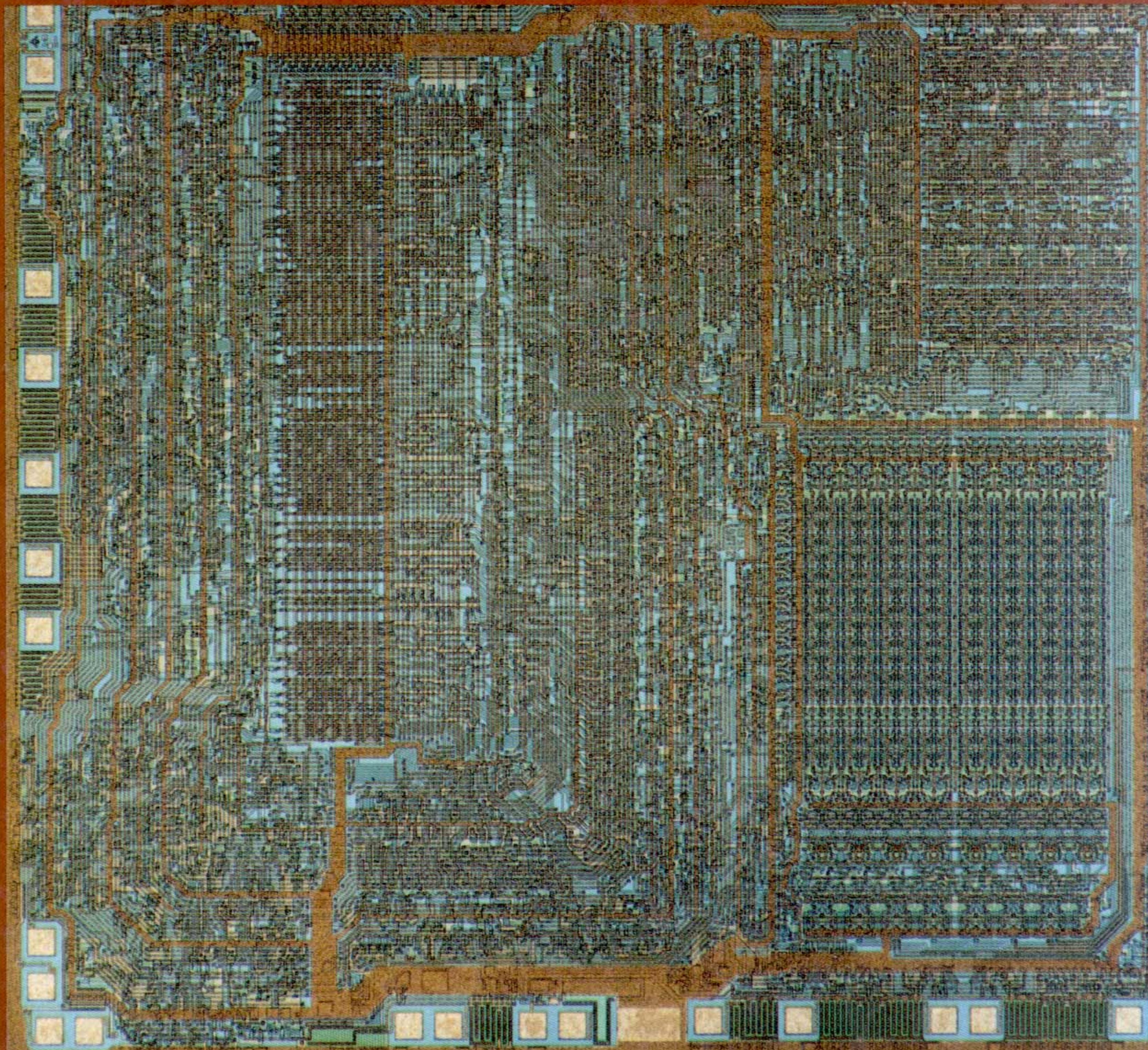


THE Z80 MICROPROCESSOR

Architecture, Interfacing,
Programming, and Design



Ramesh Gaonkar

THE Z80 MICROPROCESSOR

**Architecture, Interfacing,
Programming, and Design**

THE Z80 MICROPROCESSOR

Architecture, Interfacing,
Programming, and Design

Ramesh S. Gaonkar

ONONDAGA COMMUNITY COLLEGE
STATE UNIVERSITY OF NEW YORK

Macmillan Publishing Company
New York

Collier Macmillan Canada, Inc.
Toronto

Maxwell Macmillan International Publishing Group
New York Oxford Singapore Sydney

Cover Photo: Z80® CPU. Zilog, Inc.

Published by Merrill Publishing Company
A Bell & Howell Information Company
Columbus, Ohio 43216

This book was set in Times Roman

Administrative Editor: Steve Helba
Developmental Editor: Don Thompson
Production Coordinator: Rex Davidson
Art Coordinator: Pete Robison
Cover Designer: Cathy Watterson

Copyright © 1988 by Macmillan Publishing Company, a division of Macmillan, Inc.
All rights reserved. No part of this book may be reproduced in any form, electronic
or mechanical, including photocopy, recording, or any information storage and
retrieval system, without permission in writing from the publisher.

Library of Congress Catalog Card Number: 87-62817

International Standard Book Number: 0-675-20540-9

Printed in the United States of America

3 4 5 6 7 8 9-92 91

Client: Macmillan
Job: 12182 GaokarFM
Date: 12/19/90
File: [Macmillan 2 / file 141] (bj)
Fonts: Times (32,1014)
Proof: 5 4 3 2 1

MERRILL'S INTERNATIONAL SERIES IN ELECTRICAL AND ELECTRONICS TECHNOLOGY

ADAMSON	<i>Structured BASIC Applied to Technology</i> , 20772-X
BATESTON	<i>Introduction to Control System Technology, Second Edition</i> , 8255-2
BEACH/JUSTICE	<i>DC/AC Circuit Essentials</i> , 20193-4
BERLIN	<i>Experiments in Electronic Devices, Second Edition</i> , 20881-5
BERLIN/GETZ	<i>The Illustrated Electronic Dictionary</i> , 20451-8
	<i>Principles of Electronic Instrumentation and Measurement</i> , 20449-6
	<i>Experiments in Instrumentation and Measurement</i> , 20450-X
BOGART	<i>Electronic Devices and Circuits</i> , 20317-1
BOGART/BROWN	<i>Experiments in Electronic Devices and Circuits</i> , 20488-7
BOYLESTAD	<i>Introductory Circuit Analysis, Fifth Edition</i> , 20631-6
BOYLESTAD/KOUSOUROU	<i>Experiments in Circuit Analysis, Fifth Edition</i> , 20743-6
BREY	<i>Microprocessors and Peripherals: Hardware, Software, Interfacing, and Applications, Second Edition</i> 20883-X
BUCHLA	<i>8086/8088 Microprocessor: Architecture, Programming and Interfacing</i> , 20443-7
	<i>Experiments in Electric Circuits Fundamentals</i> , 20836-X
	<i>Experiments in Electronics Fundamentals: Circuits, Devices and Applications</i> , 20736-3
COX	<i>Digital Experiments: Emphasizing Systems and Design</i> , 20562-X
FLOYD	<i>Digital Experiments: Emphasizing Troubleshooting</i> , 20518-2
	<i>Electric Circuits Fundamentals</i> , 20756-8
	<i>Electronics Fundamentals: Circuits, Devices and Applications</i> , 20714-2
	<i>Digital Fundamentals, Third Edition</i> , 20517-4
	<i>Electronic Devices, Second Edition</i> , 20883-1
	<i>Essentials of Electronic Devices</i> , 20062-8
	<i>Principles of Electric Circuits, Second Edition</i> , 20402-X
GAONKAR	<i>Electric Circuits, Electron Flow Version</i> , 20037-7
	<i>Microprocessor Architecture, Programming, and Applications with the 8085/8080A</i> , 20159-4
GILLIES	<i>The Z80 Microprocessor: Architecture, Interfacing, Programming, and Design</i> 20540-9
HUMPHRIES	<i>Instrumentation and Measurements for Electronic Technicians</i> , 20432-1
KULATHINAL	<i>Motors and Controls</i> , 20235-3
LAMIT/LLOYD	<i>Transform Analysis and Electronic Networks with Applications</i> , 20765-7
LAMIT/WAHLER/HIGGINS	<i>Drafting for Electronics</i> , 20200-0
MARUGGI	<i>Workbook in Drafting for Electronics</i> , 20417-8
MILLER	<i>Technical Graphics: Electronics Worktext</i> , 20161-6
ROSENBLATT/FRIEDMAN	<i>The 68000 Microprocessor: Architecture, Programming, and Applications</i> , 20522-0
SCHOENBECK	<i>Direct and Alternating Current Machinery, Second Edition</i> , 20160-8
SCHWARTZ	<i>Electronic Communications: Modulation and Transmission</i> , 20473-9
STANLEY, B.H.	<i>Survey of Electronics, Third Edition</i> , 20162-4
STANLEY, W.D.	<i>Experiments in Electric Circuits, Second Edition</i> , 20403-8
TOCCI	<i>Operational Amplifiers with Linear Integrated Circuits</i> , 20090-3
	<i>Fundamentals of Electronic Devices, Third Edition</i> , 9887-4
WEBB	<i>Electronic Devices, Third Edition, Conventional Flow Version</i> , 20063-6
YOUNG	<i>Fundamentals of Pulse and Digital Circuits, Third Edition</i> , 20033-4
	<i>Introduction to Electric Circuit Analysis, Second Edition</i> , 20002-4
	<i>Programmable Controllers: Principles and Applications</i> , 20452-6
	<i>Electronic Communication Techniques</i> , 20202-7



To Gaokar, Deodhar, Bhandwalkar, and Khadapkar—
who dedicated their lives to teaching,
devoted their waking moments to helping
others, and who have been my source of
inspiration and guidance.

Preface

This text is intended for 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 Z80 microprocessor family. The text assumes a course in digital logic as a prerequisite; however, it does not assume a background in programming. This text is also suited for the second level course in curricula where the first level course is based on another microprocessor. At the outset there are two critical questions: Why teach an 8-bit microprocessor when technology is dominated by 16- and 32-bit microprocessors? And why select the Z80 microprocessor?

The first question is best answered by an analogy from the auto industry. For transportation, we have trucks, sports cars, family cars, and compact cars. Each serves a different purpose. The 8-bit microprocessors have already established their market in the areas of industrial control, such as machine control, process control, instrumentation, and consumer appliances. The 16- and 32-bit microprocessors are so powerful that their applications are better suited in such areas as high-speed data processing, CAD/CAM, multi-tasking, and multi-user systems. The 16- and 32-bit microprocessors are less likely to replace 8-bit microprocessors in industrial control applications. In many applications, even 8-bit microprocessors are utilized at less than 50 percent of their capacity. Furthermore, the basic concepts of architecture, programming, and interfacing are easier to teach with the 8-bit than with the 16-bit microprocessor.

The second question has several answers. One is that the Z80 is one of the most widely used microprocessors in industrial applications. It has simple architecture and a powerful instruction set that includes the 8085 instruction set (except for two instructions). In addition, there appears to be a resurgence of interest in the Z80, indicated by the

introduction of Z80-compatible microprocessors by major manufacturers such as National Semiconductor, Hitachi, Toshiba, and Zilog itself.

The microprocessor is a general purpose programmable logic device. A thorough understanding of the microprocessor demands the 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 function—one without the other is meaningless. Therefore, in this text, the contents are presented with an integrated approach to hardware and software in the context of the Z80 microprocessor. Part I focuses on the microprocessor architecture and interfacing, Part II introduces programming, and Part III integrates the hardware and software concepts from earlier sections in dealing with interfacing and designing microprocessor-based products. Each topic is covered in depth from basic concepts to industrial applications and illustrated by numerous examples with complete schematics. Material is supported with assignments having practical applications.

Part I consists of five chapters that deal with the hardware aspects of the microcomputer as a system, presented with the spiral approach. The material is presented in a format analogous to the view from an airplane that is getting ready to land. As the plane circles, the passenger observes a view without details. As the plane descends, the same view is seen with 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 understanding all aspects of the system. Chapter 1 presents an overview of the computer systems and discusses the microcomputer and its assembly language in the context of the entire spectrum of computers and their languages. Chapter 2 develops a generalized model of the microprocessor unit and focuses on the basic concepts related to memory and input/output (I/O). Chapter 3 examines the Z80 microprocessor in the context of the hardware and software models developed in Chapter 2. Chapters 4 and 5 are concerned with basic concepts in interfacing memory and I/O.

Part II has six chapters that deal with Z80 instructions, programming techniques, program development, and software development systems. Chapters 6 and 7 are general in nature, serving as an introduction to assembly language programming and assemblers. In the remaining chapters (Chapters 8 through 11), the contents are presented in a step-by-step format. A few instructions that can perform a simple task are selected. Each instruction is reviewed briefly by referring to the instruction set in the appendix. These instructions are then used in writing programs with explanations of 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. These chapters conclude by reviewing all the instructions discussed in those chapters. The contents in 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 software techniques of Part II. It deals with advanced topics in interfacing memory and I/Os with numerous industrial and practical examples. Each illustration analyzes the hardware and 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 dynamic memory, I/O with hand-shake signals using programmable devices, and serial I/O. Chapter 17 deals primarily with the project design of a single-board microcomputer that brings together all the concepts discussed in the text, and Chapter 18 provides a brief introduction to 16-bit microprocessors and single-chip microcontrollers. Finally, the text includes two appendices related to the instruction set. Appendix A includes the complete set of Z80 instructions explained with illustrative examples in alphabetical order so that students can easily access the instruction set with a complete explanation of each item. In addition, Appendix E summarizes all the instructions with flat information for quick reference when writing programs.

A Word to Faculty Members

This is my second textbook based on my teaching experience and my association with industry engineers and programmers. It is an attempt to share my classroom experiences and my observations in industrial practices. My assumptions and observations are similar to those of my first 8085 textbook. They are as follows:

1. It is easier to teach microprocessor concepts with an 8-bit microprocessor than with a 16-bit microprocessor. Due to their easy access on college campuses, personal computers can be used to develop programs using cross assemblers.
2. Software (instructions) is an integral part of the microprocessor and demands an emphasis equal to that of the hardware.
3. In industry, for the development of microprocessor-based projects, 70 percent of the efforts are devoted to software and 30 percent to hardware.
4. Technology and engineering students tend to be oriented toward hardware and have considerable difficulty in programming.
5. Students have difficulty in understanding mnemonics and realizing the critical importance of flags.

The text meets the objectives of courses with various emphases at the undergraduate level. 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 and interfacing lectures, and Chapters 6 through 10 and selected sections of Chapter 11 for software laboratory sessions. For additional interfacing concepts, the initial sections of Chapters 12, 13, and 15 (concepts in introduction to interrupts, programmable I/O devices, and serial I/O) are recommended. If the course is heavily oriented toward hardware, Chapters 1 through 5 and Chapters 12 through 16 are recommended, and necessary programs can be selected from Chapters 6 through 10. Interfacing laboratory sessions can be designed around the illustrations in chapters or assignments given at the end of chapters. If the course is heavily oriented toward software, Chapters 1 through 3 and 6 through 11 can be used. For a two-semester course, the entire text can be covered. The instructor's manual includes a course design, suggested weekly lecture and laboratory schedule, solutions, and selected figures to produce transparencies.

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 need to be well conversant with the programmable aspect of the microprocessor. Programming is a process of problem solving and communication in a strange language of mnemonics. Most often, hardware-oriented students find this communication process very difficult. One of the questions frequently asked by a student 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: 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 A as necessary and pay attention to the flags. This text is written in such a way that simple programming of the microprocessor can be self-taught. Once you master the elementary programming techniques, interfacing and design become exciting and fun.

ACKNOWLEDGMENTS

The primary incentive for writing this text came from the wide acceptance of my first 8085 text and the generous comments from faculty and students across the country. It would have been impossible, however, to devote the necessary time and energy to undertake such a project with a full-time teaching commitment. I wish to thank my wife, Shaila, whose ventures in the corporate world supported this undertaking and my daughters, Nelima and Vanita, who good naturedly tolerated the consequent disruptions in their young lives. In addition to my family members, several persons have made valuable contributions to this text. I would like to extend my sincere appreciation to my colleagues Peter Cahan and James Delaney for their contributions in its final phases. I cannot sufficiently emphasize the critical role played by reviewers in shaping the ideas: I have been fortunate to work directly with Peter Holsberg from Mercer Community College and David Delker at Kansas State University. Similarly, I appreciate the efforts and numerous suggestions of the other reviewers: Charles Finley, DeVry Institute of Technology, Kansas City, Missouri; Frank Gergelyi, Metropolitan Technical Institute, West Milford, New Jersey; and Kevin Greshock and Sam Watson, DeVry Institute of Technology, Lombard, Illinois.

I would like to thank Al Davis from Publication Services, who took painstaking efforts in editing the manuscript and converted my idiosyncratic prose into a readable text. Finally, I would like to thank Steve Helba, Tim McEwen, and Don Thompson for undertaking and coordinating this project through various phases, and to Rex Davidson and Pete Robison for seeing it through the production process.

Contents

PART I MICROPROCESSOR ARCHITECTURE AND INTERFACING	1
Chapter 1 Microprocessors, Microcomputers, and Assembly Language	3
1.1 Microprocessors 4 □ 1.2 From Large Computers to Single-Chip Microcomputers 11 □ 1.3 Microprocessor Instruction Set and Computer Languages 16	
Chapter 2 Microcomputer System: MPU, Memory, and I/O	25
2.1 Generalized Microprocessor Unit (MPU) 26 □ 2.2 Memory 32 □ 2.3 Input and Output (I/O) Devices 44 □ 2.4 Example of a Microcomputer System 45	
Chapter 3 Z80 Microprocessor: Programming Model and Hardware Model	51
3.1 The Z80 Programming Model 52 □ 3.2 Z80 Hardware Model 57 □ 3.3 Machine Cycles and Bus Timings 61 □ 3.4 Some Puzzling Questions and Their Answers 69 □ 3.5 Architecture of Contemporary 8-Bit Microprocessors 71	
Chapter 4 Memory Interfacing	81
4.1 Interfacing Memory 82 □ 4.2 Illustrative Example 1: Interfacing 2732 EPROM 87 □ 4.3 Illustrative Example 2: Interfacing Static R/W Memory 90 □ 4.4 Illustrative Example 3: Designing Memory 92 □ 4.5 Testing and Troubleshooting Interfacing Circuits 94 □ 4.6 Some Questions and Answers 96	

Chapter 5	Interfacing I/O Devices	101
	5.1 Interfacing Output Devices 102 □ 5.2 Illustrative Example 1: Interfacing LEDs 105 □ 5.3 Interfacing Input Devices 107 □ 5.4 Illustrative Example 2: Interfacing Input Switches 110 □ 5.5 Memory-Mapped I/O 111 □ 5.6 Illustrative Example 3: Appliance Control Using Memory-Mapped I/O Technique 113 □ 5.7 Additional Illustrative Examples: Interfacing Sensors and Motors 117 □ 5.8 Troubleshooting I/O Interfacing Circuits 120 □ 5.9 Some Questions and Answers 121	
PART II ASSEMBLY LANGUAGE PROGRAMMING: THE Z80		127
Chapter 6	Introduction to Z80 Assembly Language Programming	129
	6.1 Overview: Z80 Instruction Set 130 □ 6.2 Addressing Modes 137 □ 6.3 How to Write, Assemble, and Execute a Simple Assembly Language Program 139 □ 6.4 Flowcharting 143 □ 6.5 List of Selected Z80 Instructions 146	
Chapter 7	Software Development Systems and Assemblers	151
	7.1 Microprocessor-Based Software Development Systems □ 152	
	7.2 Assemblers 159 □ 7.3 Writing Programs Using an Assembler 162	
Chapter 8	Introduction to Z80 Instructions and Programming Techniques	169
	8.1 Data Copy (Load) Operations 170 □ 8.2 Arithmetic Operations 177 □ 8.3 Branch Operations 182 □ 8.4 Z80 Instructions Related to Index Registers 187 □ 8.5 Programming Techniques: Looping, Counting, and Indexing 189 □ 8.6 Illustrative Program 1: Block Transfer of Data Bytes 191 □ 8.7 Illustrative Program 2: Addition With Carry 193 □ 8.8 Debugging a Program 195 □ 8.9 Z80 Special Instructions 197 □ 8.10 Illustrative Program 3: Block Transfer of Data Bytes Using Z80 Special Instructions 198	
Chapter 9	Logic and Bit Manipulation Instructions	206
	9.1 Logic and Compare Operations 208 □ 9.2 Rotate (Shift) Operations and Bit Manipulation 213 □ 9.3 Illustrative Program 1: Searching for a Maximum Number 217 □ 9.4 Illustrative Program 2: Generating Square Wave Pulses 218 □ 9.5 Debugging Programs 221 □ 9.6 Z80 Special Instructions 223 □ 9.7 Illustrative Program 3: Searching for a Maximum Number Using the Instruction CPI 224	
Chapter 10	Stacks and Subroutines	233
	10.1 Stack 234 □ 10.2 Illustrative Program 1: Examining and Manipulating Flags 239 □ 10.3 Subroutine 241 □ 10.4 Illustrative Program 2: Traffic Signal Controller 245 □ 10.5 Subroutine	

Documentation and Parameter Passing	249	□	10.6 Advanced Subroutine Concepts	251	□	10.7 Illustrative Program 3: BCD Counter and Its Seven-Segment LED Display	252	□	10.8 Modular Programming and Debugging	257															
Chapter 11	Application Programs and Software Design	263																							
11.1 16-Bit Operations	264	□	11.2 Illustrative Program: Multi-precision Addition	268	□	11.3 Binary Multiplication	269	□	11.4 Binary Division	271	□	11.5 Illustrative Program: BCD to Binary Conversion	274	□	11.6 Illustrative Program: Binary to BCD Conversion	277	□	11.7 Illustrative Program: ASCII to Binary Code Conversion	279	□	11.8 Illustrative Program: Binary to ASCII Code Conversion	281	□	11.9 Software Design	282
PART III INTERFACING PERIPHERALS, PROGRAMMABLE I/O DEVICES, APPLICATIONS, AND DESIGN											289														
Chapter 12	Interrupts	293																							
12.1 Basic Concepts in Interrupt I/O	294	□	12.2 Illustration: An Implementation of the Z80 Interrupt in Mode 0	302	□	12.3 Illustration: Interfacing an A/D Converter in Interrupt Mode 1	307	□	12.4 Interrupt Mode 2	313	□	12.5 Nonmaskable Interrupt	315	□	12.6 Multiple Interrupts and Priorities	318	□	12.7 Illustration: Restart as a Software Instruction—Implementation of a Breakpoint Technique	322						
Chapter 13	Programmable Interface Devices	329																							
13.1 Basic Concepts in Programmable Devices	330	□	13.2 Z80 Parallel Input/Output Device (PIO)	333	□	13.3 Modes 0, 1, and 2 with Handshake Signals and Interrupt I/O	339	□	13.4 Mode 3: Bit Mode	344	□	13.5 Illustration: Interfacing Keyboard and Seven-Segment Display	348	□	13.6 Illustration: Bidirectional Data Transfer between Two Microcomputers Using PIO in Mode 2	359	□	13.7 The 8255A Programmable Peripheral Interface	366						
Chapter 14	Programmable Timers and Counters	385																							
14.1 Z80 CTC—Counter/Timer Circuit	386	□	14.2 Illustration: Design of a Baud (Rate) Generator Using the CTC in the Timer Mode	395	□	14.3 Illustration: Using the CTC in the Counter Mode with Interrupt	398	□	14.4 The 8253 Programmable Interval Timer	401															
Chapter 15	Serial I/O and Data Communication	411																							
15.1 Basic Concepts in Serial I/O	412	□	15.2 Software-Controlled Asynchronous Serial I/O	422	□	15.3 Programmable Communication Interface—Intel 8251A: Hardware Approach to Serial I/O	425	□	15.4 Illustration: Interfacing a RS-232 Terminal Using the 8251A in the																

Polled Mode 435	□ 15.5 Serial Input/Output Controllers: Z80 SIO and DART 439	□ 15.6 Illustration: Interfacing a RS-232 Terminal Using DART (SIO) in the Interrupt Mode 455							
Chapter 16	Advanced Topics in Memory Design and DMA Concepts		463						
	16.1 Interfacing Memory Using Wait States 464	□ 16.2 Interfacing Dynamic Memory 468	□ 16.3 Illustration: Interfacing the 2118—16K R/W Dynamic Memory—with the Z80 475	□ 16.4 Designing Memory Systems 478	□ 16.5 Direct Memory Access (DMA) and the Z80 DMA Controller 484				
Chapter 17	Designing Microprocessor-Based Products		495						
	17.1 Project Statement: Designing a Microcomputer System 496	□ 17.2 Z80 MPU Design 497	□ 17.3 Memory Design 504	□ 17.4 Designing Scanned Displays 507	□ 17.5 Interfacing a Matrix Keyboard 512	□ 17.6 Project Decisions and Software Design 519	□ 17.7 Designing Software Modules 526	□ 17.8 Development and Troubleshooting Tools 532	
Chapter 18	Trends in Microprocessor Technology		537						
	18.1 Single-Chip Microcomputers (Microcontrollers) 538	□ 18.2 16- and 32-Bit Microprocessors 541	□ 18.3 Bus Interface Standards 554						
References			559						
Appendix A	Z80 Instruction Set		561						
Appendix B	Number Systems		621						
	B.1 Number Conversion 621	□ B.2 2's Complement and Arithmetic Operations 623	□ B.3 Modulo-2 Arithmetic 629						
Appendix C	American Standard Code for Information Interchange: ASCII Codes		631						
Appendix D	Preferred Logic Symbols		633						
Appendix E	The Micro-Trainer: Z80-Based Single-Board Microcomputer for Laboratory Use		637						
Appendix F	Z80 Instruction Summary		641						
Index			647						

CHAPTER 1:

Microprocessors, Microcomputers, and
Assembly Language

CHAPTER 2:

Microcomputer System: MPU, Memory, and I/O

CHAPTER 3:

Z80 Microprocessor: Programming Model and
Hardware Model

CHAPTER 4:

Memory Interfacing

CHAPTER 5:

Interfacing I/O Devices

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

The material is presented in a format similar to the view from an airplane preparing to land. As the plane circles, one observes a view without any details. As the plane descends, one begins to see the same view but with more details. Chapter 1 presents the microprocessor from two points of view: the microprocessor as a programmable device and as an element of a computer system, and how it communicates 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. Chapter 2 describes in generalized models a microcomputer system and its three components: the

I

Microprocessor Architecture and Interfacing

microprocessor, memory, and input and output (I/O). Chapters 3, 4, and 5 examine these components in detail and discuss how memory and I/O devices interface with the Z80 microprocessor.

PREREQUISITES

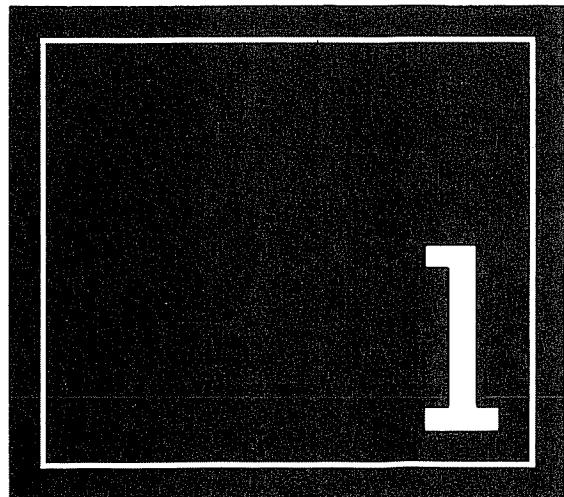
The reader is expected to know the following concepts:

- Number systems (binary, octal and hexadecimal) and their conversions.
- Boolean algebra, logic gates, flip-flops, and registers.
- Concepts in combinational and sequential logic.

Microprocessors, Microcomputers, and Assembly Language

The microcomputer plays a significant role in the everyday functioning of industrialized societies. The microcomputer is no different from any other computer in its basic structure. In the 1960s, computers were accessible and affordable only to such institutions as large corporations, universities, and government agencies. Today because of advances in semiconductor technology, the million-dollar computing capacity of the 1960s is now available for less than five dollars in an integrated circuit called the **microprocessor**. The microprocessor can be defined as a programmable logic device that can be used to control processes, to turn devices on or off, or as a data processing unit of a computer. A computer that is designed using the microprocessor is called a **microcomputer**. This chapter introduces the basic structure of a computer and shows how the same structure is applicable to microprocessor-based products. Later in the chapter, microcomputer applications in an industrial environment are presented in the context of the entire spectrum of various computer applications.

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 **mnemonics**, 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 Z80. The advantages of assembly language are compared with such English-like languages as BASIC and FORTRAN.

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.

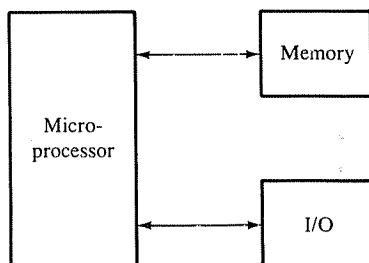
1.1 MICROPROCESSORS

A microprocessor is a multipurpose, *programmable* logic 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*. A typical programmable machine can be represented with three components: microprocessor, memory, and I/O as shown in Figure 1.1. These three 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. When the microprocessor system is used for control applications such as turning devices (or machines) on and off, it is generally known as a microcontroller. When it is used for computing or data processing, it is known as microcomputer.

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.

FIGURE 1.1
A Programmable Machine



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 16-bit word is known as a 16-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 toaster is an example of an elementary programmable machine. It can be programmed to remain on for a given length of time by adjusting a mechanical lever to a "light" or "dark" setting. The toaster is designed to understand and execute one instruction. On the other hand, the present-day microprocessor is designed to understand and execute many **binary** instructions. It can be used to perform sophisticated computing functions as well as to perform such simple control tasks as turning devices on and off. The person using a microprocessor selects appropriate instructions and asks the microprocessor to perform various tasks on a given set of data.

The engineer who designs a toaster determines the timing for light and dark toast, and the manufacturer of the toaster provides the necessary instructions to operate the toaster. Similarly, after the engineers designing a microprocessor determine a set of tasks the microprocessor should perform and design the necessary logic circuits, the manufacturer of the microprocessor provides 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 a storage device called **memory**, which can be read by the microprocessor.

MEMORY

Memory is like the page(s) 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 such devices as a keyboard or simple switches. These devices are called **input devices**. The microprocessor reads the instructions from the memory and processes the data according to those instructions. The result can be displayed by such a device as seven-segment LEDs (Light Emitting Diodes) or printed by a printer. These devices are called **output devices**.

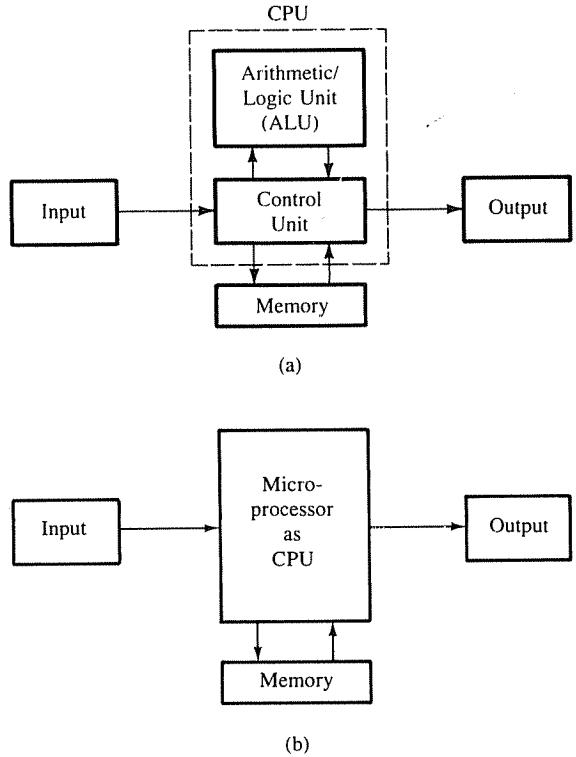
MICROPROCESSOR AS A CPU

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 ALU (Arithmetic/Logic Unit) and Control Unit. The CPU contains various registers to store data, the arithmetic/logic unit (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 such devices 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 1960s, the CPU was designed with discrete components on various boards. With the advent of the 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).

FIGURE 1.2

(a) Traditional Block Diagram of a Computer (b) Block Diagram of a Computer with the Microprocessor as CPU



1.11 Advances in Semiconductor Technology

In the last thirty years, 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 the technology moved from SSI to LSI, 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. The necessary signal patterns of 0s and 1s were stored in registers and given to the programmable chip at appropriate times; the group of registers used for storage was called memory. Because of the LSI technology, it became possible to build many computing functions and their related timing on a single chip.

The Intel 4004 was the first 4-bit programmable device that was primarily used in calculators. It was designed by Intel Corporation and became known as the 4-bit microprocessor. It 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. In terms of the instruction set, the 8080 and the 8085 are almost identical; however, the Z80 has a powerful instruction set containing twice as many instructions as the 8080. 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 16- and 32-bit microprocessors, and 64-bit microprocessors are also being used in some prototype computers. The 8-bit microprocessors are not simply being replaced by more powerful microprocessors, however; 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 the 16- and 32-bit microprocessors are being used for mathematical computing (number crunching) and data processing applications. Our focus here is in using 8-bit microprocessors as programmable devices.

1.12 Microcomputer Organization

Figure 1.3 shows a simplified but formal structure of a microcomputer. It includes four components: *microprocessor*, *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 *microcomputer* is a complete computer similar to any other computer, except that the CPU functions of the *microcomputer* are performed by the *microprocessor*. Similarly, the term **peripheral** is used for input/output devices. The various components of the *microcomputer* shown in Figure 1.3 and their functions are described in this section.

MICROPROCESSOR

The *microprocessor* is a semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale (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. Results are stored either in registers or in memory.

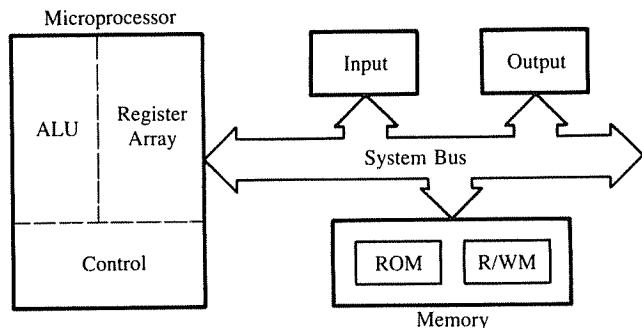


FIGURE 1.3
Microcomputer with Bus Architecture

Register Array This area of the microprocessor consists of various registers. These registers are primarily used to store data temporarily during the execution of a program. Some of the registers 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. 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 6800 and 6809. Earlier microcomputers such as the Radio Shack TRS-80, the Televideo 803, and the Kaypro 4 are designed around the Z80 microprocessor. The recent versions of IBM personal computers, Personal System/2, are designed around 16-bit and 32-bit microprocessors; the model 60 is based on the Intel 80286 (16-bit) and the model 80 is based on the Intel 80386 (32-bit). Single-board microcomputers such as the Intel SDK-85, the Motorola MEK-6800-D2, the Multitech Micro-Professor, and the CAMI Research Micro-Trainer are commonly used in college laboratories; the SDK-85 is based on the 8085 microprocessor, the MEK-6800-D2 on the 6800 microprocessor, and the Micro-Professor and the Micro-Trainer on the Z80 microprocessor.

INPUT

The input section transfers data and instructions in binary from the outside world to the microprocessor. It includes such devices as a keyboard, a teletype, and an analog-to-digital converter. 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 keyboard (explained in Section 1.3) 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, single-board microcomputers generally have Hex keyboards.

OUTPUT

The output section transfers data from the microprocessor to such output devices as light emitting diodes (LEDs), a cathode-ray tube (CRT), a printer, a magnetic tape, or another

computer. Typically, single-board computers include LEDs and seven-segment LEDs as output devices.

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 (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.

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.

1.13 How Does the Microcomputer 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 microcomputer 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.

1.14 Summary of Important Concepts

The functions of various components of a microcomputer can be summarized as follows:

1. The microprocessor

- communicates with all peripherals (memory and I/Os) using the system bus.
- controls timing of information flow.
- performs the computing tasks specified in a program.

2. The memory

- stores binary instructions and data, called programs.
- 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.

FROM LARGE COMPUTERS TO SINGLE-CHIP MICROCOMPUTERS

1.2

In the last thirty years, advances in semiconductor technology have had an unprecedented impact on computers. Thirty years ago, 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, multi-million-dollar machines as the IBM 3090 to the less-than-\$200 home 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 15 years ago, 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-sized computers, and microcomputers.

1.21 Large Computers

These are large, general-purpose computers designed to perform such data processing tasks as complex scientific and engineering calculations and handling of records for large corporations or government agencies. The price is generally beyond \$1 million and can go as high as \$10 million. Typical examples of these computers include IBM 3090 or IBM 9370 series, Burroughs 6700, and Univac 1100.

These are high speed computers, and their word lengths range from 32 to 64 bits. They are capable of addressing megabytes of memory and handling all types of peripherals. For the more expensive, the CPU alone may cost more than one million dollars. For example, the IBM 3000/81 CPU, capable of addressing 32 megabytes of memory, may cost more than \$3 million; the price of the total system may go as high as \$6 million. However, IBM also has medium-sized systems, called 4300 series, costing around \$100,000, and they are also known as **mainframe computers**.

1.22 Medium-Sized Computers

In the late 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-sized businesses, such as payroll and accounting. They were called **minicomputers**. The price range was anywhere from \$25,000 to \$100,000. Typical examples include such computers as Digital Equipment PDP 11/45 and Data General Nova.

These computers were slower than the large computers, and their word length generally ranged from 12 to 32 bits. They were capable of addressing 64K to 256K bytes of memory. Some of the larger minicomputers were known as **midicomputers**. However, these classifications are no longer valid. For example, Digital Equipment's new VAX 11 system is a 32-bit machine with megabytes of memory addressing capacity. The price ranges from \$50,000 to \$450,000. The high end of the VAX 11 system is almost in the territory of the large computers.

1.23 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, including video games, word processing, and small business applications. Early arrivals in the microcomputer market, such as Cromemco, North Star Horizon, Radio Shack TRS-80, and Apple were designed around 8-bit microprocessors. Since then, 16-bit and 32-bit microprocessors such as Intel 8086/88, 80286, and 80386, Motorola 68000, and Zilog Z8000 have been introduced, and recent microcomputers have been designed around these microprocessors. Present day microcomputers can be classified into four groups: business (or personal), home, single-board, and single-chip microcomputers.

BUSINESS MICROCOMPUTERS

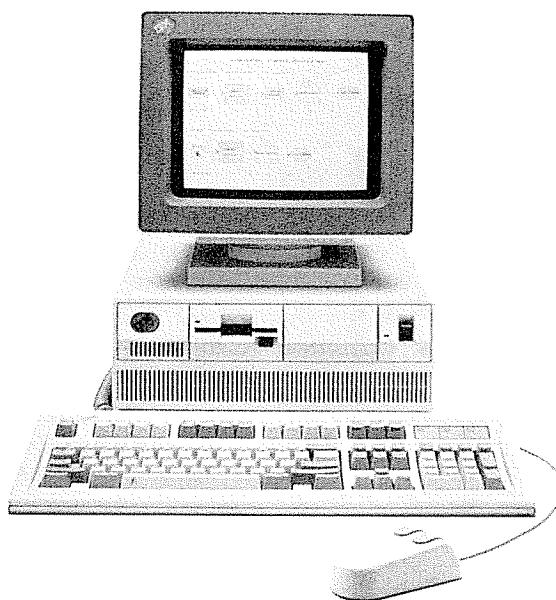
These microcomputers are being used for a variety of purposes, such as payroll, business accounts, word processing, legal and medical recordkeeping, personal finance, and instruction. They are also known as personal computers. Typically, the price ranges from \$1,000 to \$8,000 for a single-user system, and it can go higher for a multi-user system. Examples include such microcomputers as IBM Personal Computers (IBM PC, XT, AT, System/2), the AT&T 6300 series, Apple Computers, and Zenith or Compaq computers.

At the low end of the microcomputer spectrum, a typical configuration includes an 8-bit or 16-bit microprocessor, 64K (or 128K) bytes of memory, a CRT terminal, a printer, and dual disk drive for 5 $\frac{1}{4}$ -inch floppy disks. 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, while 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. Floppy disks are used to store such programs as compilers, interpreters, system programs, user programs, and data. Whenever the user needs to write a program, the necessary software is transferred from the floppy disk to the system's memory. At the high end of the microcomputer spectrum, the basic configuration remains essentially similar. It may include a 16-bit or 32-bit microprocessor, a hard disk with megabytes of storage, two floppy disks, an expensive terminal, and a printer.

FIGURE 1.4

Microcomputer with Disk Storage:
IBM Personal System/2

SOURCE: Photograph courtesy of IBM Corporation



HOME COMPUTERS

Home computers are differentiated from business microcomputers in terms of their memory storage. Typically, these computers have an 8-bit microprocessor, a CRT terminal with an ASCII typewriter, 16K to 64K memory, and a cassette tape as a storage medium. Some of these computers can be used with television as a video monitor. The prices of these computers may range from less than \$200 to \$500. Typical examples include Commodore 64, Tandy 100, and Atari 130XE. These microcomputers are used primarily for playing video games, learning simple programming, and running some instructional programs.

SINGLE-BOARD MICROCOMPUTERS

These microcomputers are used primarily in college laboratories and industries for instructional purposes or for evaluating the performance of a given microprocessor. They can also be part of some larger systems. Typically, these microcomputers include an 8-bit microprocessor, from 256 to 2K bytes of user memory, a Hex keyboard, and seven-segment LEDs for display. The system monitor programs of these computers are generally small;

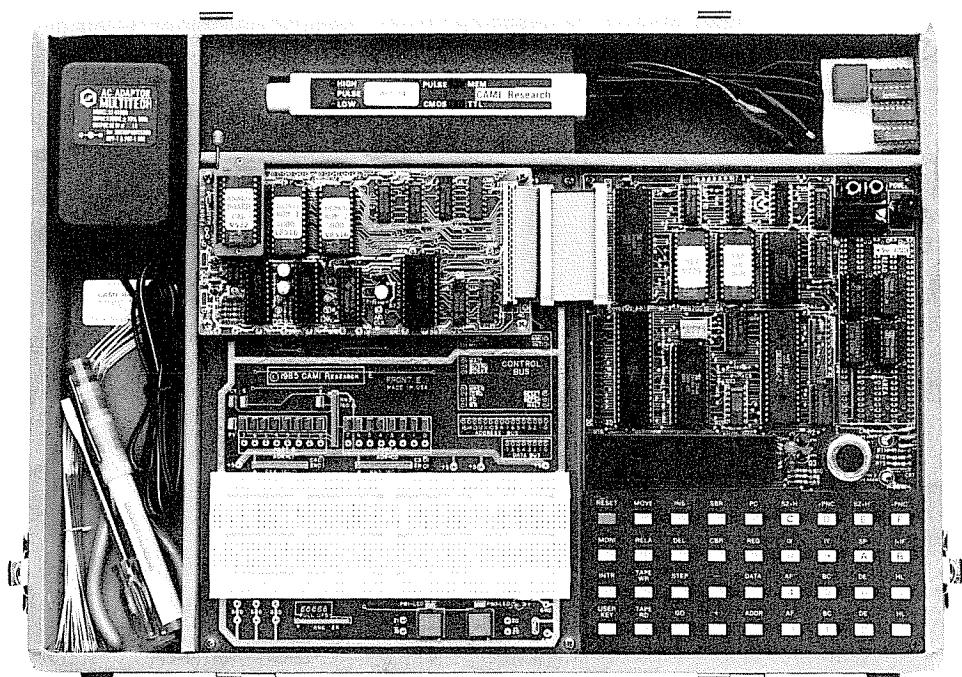


FIGURE 1.5

Single-Board Microcomputer: Micro-Trainer

SOURCE: Photograph courtesy of CAMI Research, Inc.

they are stored in less than 2K bytes of ROM. The prices of these single-board computers range from \$100 to \$800, with the average price being about \$300.

Examples of these computers include such systems as Intel SDK-85, Motorola Evaluation Kit, and CAM1 Research Micro-Trainer (Figure 1.5). These are generally used to write and execute assembly language programs and to perform interfacing experiments.

SINGLE-CHIP MICROCOMPUTERS

These microcomputers are designed on a single chip, which typically includes a microprocessor, 64 bytes of R/W memory, from 1K to 2K 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 the Zilog Z8, Intel MCS 51 and 96 series, Fairchild F8, and Motorola 6802.

The entire spectrum of computer applications is shown in Figure 1.6, and various applications and categories of the microcomputer are listed in Table 1.1.

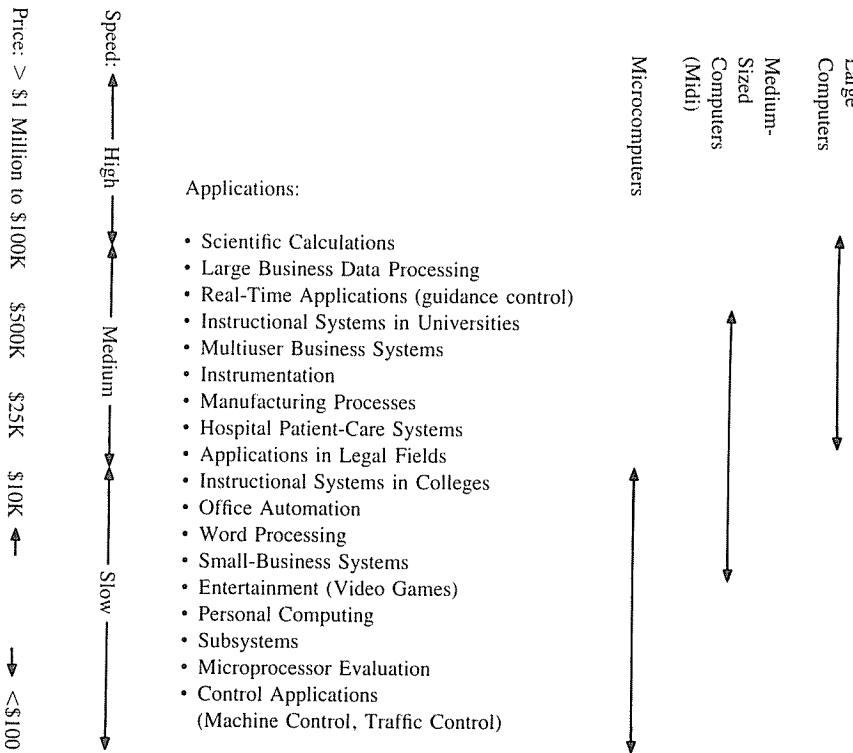


FIGURE 1.6
Applications: From Large Computers to Single-Chip Microcomputers

TABLE 1.1
Microcomputer Applications

Characteristics	Types			
	Microcomputer with Disk Storage	Microcomputer with Cassette Tape Storage	Single-Board Microcomputer	Single-Chip Microcomputer
Price range	\$1,000–8,000	\$100–\$500	\$100–\$800	<\$50
Memory size (R/W/M)	64K–512K	4K–64K	256 Bytes—2K	64–128 bytes
I/O	ASCII Keyboard, CRT	ASCII Keyboard, CRT	Hex Keyboard (Rarely ASCII) LEDs	Keyboard LEDs
Languages Used	Various Types of High-Level Languages, Assembly	High-Level, Generally BASIC	Assembly	Assembly
Applications	Small Business Applications, Word Processing, Instructional Applications	Entertainment (Video Games), Personal Computing	Evaluation of Microprocessors, Assembly Language Instruction; As a Subsystem	Industrial Control

1.3

MICROPROCESSOR INSTRUCTION SET AND COMPUTER LANGUAGES

Microprocessors recognize and operate in binary numbers. However, each microprocessor has its own binary words, instructions, meanings, and language. The words are formed by combining a number of bits for a given machine. The **word** (or word length), as defined earlier, is the number of bits the microprocessor recognizes and processes at a time. The word length ranges from 4 bits for small, microprocessor-based computers, to 32 bits for such large computers as the IBM 3800 series. Another term commonly used to express word length is **byte**. The **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 also found in popular computer magazines and books. (A byte has two nibbles.)

The **instruction** is defined as a complete task (such as Add) the microprocessor can perform; it can be made up of one or more words. Each machine has its own set of instructions based on the design of its CPU or its microprocessor. To be intelligible to the microprocessor, instructions must be written in binary language, also known as **machine language**. However, it is difficult for human beings to write programs in sets of 0s and 1s. Therefore, microprocessor manufacturers have devised Englishlike words to represent the binary instructions of a machine, and programmers can write programs using these words.

These are called **assembly language** programs. 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 so that 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 Z80 microprocessor.

1.31 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**. The set of instructions designed into the machine makes up what is called the machine language, a binary language composed of 0s and 1s. Its words, its instructions, and their meanings are specific to each computer. In this book, we are concerned with the language of the Z80 microprocessor from Zilog Corporation, a widely used microprocessor in industrial applications. The primary focus here is on the microprocessor, because it is the microprocessor that determines the machine language and the operations of a microcomputer.

1.32 Z80 Machine Language

The Z80 is a microprocessor with 8-bit word length. Its instruction set (or language) is upward compatible with that of the 8080; the Z80 has 159 instructions that include the entire 8080 set of 72 instructions. An instruction, as discussed earlier, is a binary pattern entered through an input device to command the microprocessor to perform a 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 which adds the number in the register called B to the number in the accumulator, and keeps the sum in the accumulator. |

The Z80 microprocessor has a variety of such bit patterns resulting in its 159 instructions for performing different operations, called the **instruction set**. The Z80 microprocessor also accepts data in 8-bit words as input from input devices, processes data according to the instructions written by the user, and sends out data in 8-bit words to output devices. This binary language with a predetermined instruction set is called the Z80 machine language.

However, it is tedious and conducive to error for human beings to recognize and

write instructions in binary language. Therefore, for convenience, these instructions are written in hexadecimal (or octal) code and entered into 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 into a singleboard microcomputer system with Hex keyboard by pressing two keys: 3 and C. The monitor program of the system translates these keys into their equivalent binary pattern.

1.33 Z80 Assembly Language

Even though the instructions can be written in hexadecimal code, it is still not easy to understand such a program. Therefore, each manufacturer of microprocessors has devised a symbolic code for each instruction, called a **mnemonic**. (The word *mnemonic* is based on the Greek word related to *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 (3C₁₆ or 3C_H* in hexadecimal) of the Z80 microprocessor is represented by the mnemonic INC A:

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

Similarly, the binary code 1000 0000 (80₁₆ or 80_H*) is represented as follows:

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

Even though these symbols do not specify the complete operations, they suggest the significant portions. The complete description of each instruction must be supplied by the manufacturer. The complete set of Z80 mnemonics is called the Z80 assembly language, and a program written in these mnemonics is called an assembly language program. Again, the assembly language is specific to each microprocessor. For example, the Motorola 6800 microprocessor has an entirely different set of binary codes and mnemonics from that of the Z80. 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.

The machine language and the assembly language are microprocessor-specific, and both are 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 entered into a microprocessor system and translated into binary code? In a microcomputer, the mnemonics are entered as ASCII code (explained in the next section) using the keyboard as an input device, and the translation is performed by a program called an **assembler**. In a single-

*Hexadecimal numbers are shown with the subscript H in the text.

board microcomputer, the user translates mnemonics into Hex digits by looking up the code manually in the instruction set and enters them into the system through the Hex keyboard. This is called **hand assembly**.

1.34 Alphanumeric Codes

A computer is a binary machine; in order 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 00_H to $7F_H$ is assigned to either a letter, a decimal number, a symbol, or a machine command (See Appendix C). For example, hexadecimal 30_H to 39_H represent 0 to 9, decimal digits; 41_H to $5A_H$ represent capital letters A through Z; 20_H to $2F_H$ represent various symbols; and the initial codes 00_H to $1F_H$ represent such machine commands as carriage return and line feed. Devices which use ASCII characters include ASCII terminals, teletype machines (TTY), and printers. When the key 9 is pressed on an ASCII terminal, the computer receives 39_H in binary, and the system program translates ASCII characters into appropriate binary or BCD numbers.

Another code, called EBCDIC (Extended Binary Coded Decimal Interchange Code) is widely used in IBM computers (except in IBM Personal Computers or microcomputers). This is an 8-bit code representing 256 combinations; however, several combinations are not used.

1.35 Writing and Executing an Assembly Language Program

As explained earlier, a program is a set of logically related instructions written in a specific sequence to accomplish a task. To write and execute an assembly language program manually 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.

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, the steps of looking up the machine codes and

entering the program, which are tedious and subject to errors, can be avoided by using an assembler on a microcomputer system.

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 certain rules which must be learned by the programmer. Assemblers are discussed in detail in Chapter 7.

1.36 High-Level Languages

Programming languages that are intended to be machine-independent are called high-level languages. The list includes such languages as C, FORTRAN, BASIC, PASCAL, and COBOL. These languages 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 Z80 microprocessor can generally run on another microcomputer with a different microprocessor.

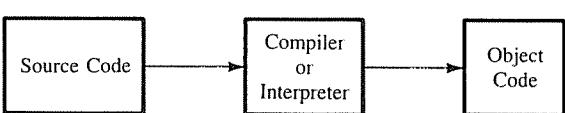
Now the question is: How do words in English get converted into the binary languages of different microprocessors? The answer lies with another program called either a **compiler** or an **interpreter**. These programs accept English-like statements as their input, called the *source code*. The compiler or interpreter then translates the source code into the machine language compatible with the microprocessor being used in the system. This translation into the machine language is called the *object code* (Figure 1.7). Each microprocessor needs its own compiler or interpreter for each high-level language. The primary difference between a compiler and an interpreter is in the process of generating machine code. The compiler reads the entire program first and then generates the object code, while the interpreter reads one instruction at a time, produces its object code, and executes the instruction before reading the next instruction. M-Basic is a common example of an interpreter for the BASIC language. Compilers are generally used in such languages as FORTRAN, COBOL, and PASCAL.

Compilers and interpreters require large memory space because each instruction in English requires several machine codes to translate that instruction into binary. On the other hand, there is a one-to-one correspondence between the assembly language mnemonics and the machine 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 programs, also known as debugging. 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 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 appli-

FIGURE 1.7

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



cations as converting a high frequency waveform into digital data, program efficiency is critical. In real-time applications, events and time should closely match with 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. The advantage of time saved in debugging a large program may outweigh the disadvantages of large memory requirements and inefficiency.

SUMMARY

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

Computer Structure

- **Digital Computer**—a programmable machine that processes binary data. It includes four 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 100 gates on a single chip.
- **LSI**—Large-Scale Integration. The process of designing more than 1,000 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**—a semiconductor device (integrated circuit) that is manufactured by using the large-scale integration technique. It includes the ALU, register arrays, and control circuits on a single chip.

- **Microcomputer**—a computer that uses a microprocessor as its CPU. It includes four components: microprocessor, memory, input, and output.
- **Bus**—a group of lines used to transfer bits between the microprocessor and other components of the computer system.
- **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 easily read and altered.

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 as a whole.
- **Instruction**—a command in binary that is recognized and executed by the computer in order 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 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).
- **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 code manually from the instruction set of a computer and entering those codes into the computer through a keyboard.
- **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 Z80 microprocessor and its assembly language. The chapter has given 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 Z80 microprocessor and its industrial applications, and on assembly language programming in the context of these applications. In the microcomputer field, there is hardly any separation 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. There are various functions that can be performed through either hardware or software, and a designer needs to consider both approaches. This book focuses on trade-off between the two approaches as a design philosophy.

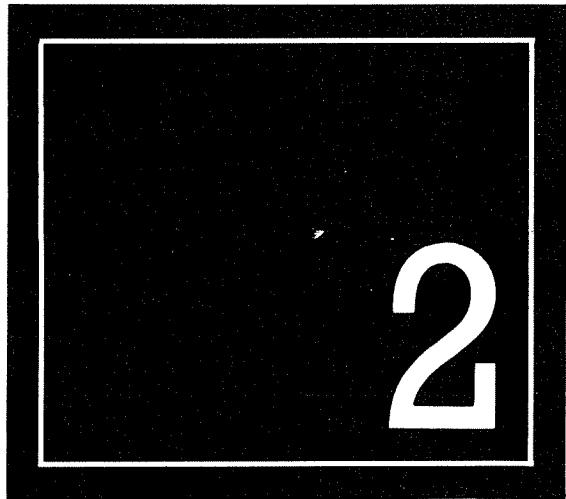
ASSIGNMENTS

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 the following terms: SSI, MSI, and LSI.
6. Define: bit, byte, word, and instruction.
7. How many bytes make a word of 32 bits?
8. Explain the difference between the machine language and the assembly language of the Z80 microprocessor.
9. What is an assembler?
10. What are low- and high-level languages?
11. Explain the difference between a compiler and an interpreter.
12. What are the advantages of an assembly language in comparison with high-level languages?

Microcomputer System: MPU, Memory, and I/O

A microcomputer system consists primarily of three components—the microprocessor unit (MPU), memory, and I/O (input/output). The MPU is the central player; it communicates with memory and I/O devices, processes data, and controls timing of all its operations. In this chapter, we will examine what the MPU does and what its requirements are. We then design a model for a generalized MPU that expands on the bus concept discussed in the previous chapter and shows signals necessary for the MPU to communicate with other devices. The model also describes the requirements for processing data and shows registers and logic circuits the MPU needs.

Memory and I/Os are integral parts of a microcomputer system. We will discuss memory in terms of its basic elements—latches and registers—and specify the requirements for a memory chip to store information and communicate with the MPU. Based on those requirements, we then develop the concepts of memory addressing and memory maps. We also discuss how the MPU addresses and communicates with I/Os.



OBJECTIVES

- List the four program-initiated operations performed by the MPU.
- Define the functions of the address bus, data bus, and control signals.
- List the externally initiated operations the MPU should respond to.
- Draw the model of a generalized MPU showing the necessary signals.
- List the types of registers the MPU needs to process data internally.
- Explain the internal organization of memory and the requirements of a memory chip to store information and communicate with the MPU.
- Explain the functions of the control signals: Chip Select (\overline{CS}), Read (\overline{RD}), and Write (WR).
- Explain how memory addresses are assigned to a memory chip and recognize the memory map of a given chip.
- List the two techniques of addressing I/O devices.
- Draw a block diagram of a microcomputer system showing the MPU, memory, I/Os, and buses.

2.1

GENERALIZED MICROPROCESSOR UNIT (MPU)

The Microprocessor Unit (MPU) is a programmable logic device with a designed set of instructions. In this section, we will examine the functions and requirements of the MPU and derive a generalized model. From the previous chapter, we can recall what the MPU does. It reads or fetches each instruction, one at a time, from memory and performs data manipulation specified by the instruction; it also reads data from input devices, and writes (or sends) data to output devices.

When the MPU is executing a program, it communicates frequently with memory and I/O devices; the process consists of fetch, decode, and execute operations. However, the question is: Can it respond to unexpected events? For example, while printing a long program, can it stop printing temporarily and read any critical data that may arrive at the input? Can it be “interrupted”? Can it wait until a peripheral is ready? For example, when memory response is too slow, can the MPU wait until memory is ready? The answer to all these questions must be affirmative.

In addition to processing data according to the instructions written in memory, the MPU needs to respond to various situations described above. External devices should be able to interrupt and request the attention of the MPU. This communication process and related operations between the MPU and the external devices (memory, I/Os) can be classified into two main categories:

- Program-initiated operations
- Peripheral (or externally) initiated operations.

To perform these operations, the MPU requires a group of logic circuits, a set of signals to transfer information and control signals for timing, and clock circuitry; these constitute the architecture. Early microprocessors did not have the necessary circuitry on one chip; the complete units were made up of more than one chip. Therefore, we define here the term Microprocessor Unit (MPU) as a group of devices that can perform opera-

tions similar to those of the Central Processing Unit (CPU). For example, the 8080A MPU requires three chips to make it a functional unit. However, since later microprocessors include most of the necessary circuitry on a single chip, the terms MPU and microprocessor are often used synonymously.

2.11 Program-Initiated Operations and Buses

To communicate with memory and I/Os, the MPU performs four operations:

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

Now the question is: how does the MPU identify a memory register or an I/O device? It does so the same way we identify a house; we give a number. Because it understands only the binary numbers, the MPU identifies each memory register or I/O by a binary number called an address. The next question is: how does the MPU inform the peripherals when it is ready to read or write data? It does so by sending out appropriate timing signals called control signals before it transfers data.

The steps in performing these MPU operations can be summarized as follows (not necessarily in the order listed):

1. Identify the memory location or the peripheral with its address.
2. Transfer binary data.
3. Provide timing or synchronization signal.

Therefore, the MPU requires three sets of communication lines called buses: the first group of lines, called the address bus, to identify the memory location; the second group, called the data bus, to transfer data; and the third group, called the control lines, for timing signals. In the previous chapter (Figure 1.3), all these different signal lines were grouped together and shown as the system bus. Now we shall describe them individually.

ADDRESS BUS

As mentioned earlier, the MPU identifies each peripheral or memory location with a binary address. Now the question is: how large is this address? The answer depends upon the internal design of the microprocessor and available pins on a chip; it can be eight, 16, 20, or more bits. If the address size is 12 bits, the microprocessor can identify $4,096 (2^{12})$ different memory locations. The addressing is simply a numbering scheme to identify memory registers. For example, a two-digit decimal numbering scheme can identify only 100 items, from 00 to 99. On the other hand, a four-digit numbering scheme can identify 10,000 items, from 0000 to 9999. Thus, the number of bits (address lines) used for addressing by the MPU clearly determines the number of memory registers it can identify.

Figure 2.1 shows one group of lines as the address bus for our generalized MPU.

The arrow suggests that these lines are unidirectional—the signals flow from the MPU to peripherals because only the MPU sends out an address. The address lines are generally identified as A_0 to A_m , where m is the size of the address bus. Typically, earlier microprocessors such as the 8085, the Z80, and the 6800 have 16 address lines which are capable of addressing 65,536 (2^{16}) memory locations, commonly known as 64K memory. However, recent microprocessors such as the 8086 have 20 address lines, and the 68000 has 23 address lines.

DATA BUS

The second group of lines shown in Figure 2.1 is the data bus. These lines are used to transfer data and are bidirectional—data can flow either direction. These lines are identified as D_0 to D_n , where n signifies the size of the data bus. Again, the size of the data bus determines how large a binary number can be transferred and processed at a time and thus influences the microprocessor architecture considerably. The 8085, the Z80, and the 6800 have eight data lines and are thus called 8-bit microprocessors. On the other hand, the 8086, the Z8000, and the 68000 have 16 data lines and are called 16-bit microprocessors.

CONTROL SIGNALS (MPU INITIATED)

These are individual signal lines generated by the MPU to indicate its operations. The MPU generates a specific signal for each of its four operations—Memory Read, Memory Write, I/O Read, and I/O Write. These are timing signals that are used to enable, or activate, peripherals. For example, to fetch (or read) an instruction from a memory location, the MPU sends a timing pulse called Memory Read to enable the memory chip.

2.12 Externally Initiated Operations

There are various occasions when ongoing MPU operations need to be interrupted. For the MPU we are designing, we can classify these types of external interruptions or delays into four categories.

- Reset: Start again from the beginning. For example, if we are using a microprocessor as a timer, we should be able to reset the timer after each operation or in the middle of an operation and start again.
- Interrupt: Stop the ongoing process temporarily; do something now which is more critical, and then go back to the original process. For example, we should be able to stop printing temporarily and read data from a keyboard; then, when the MPU finishes reading that data, it can go back to printing.
- Wait: When memory response time is too slow to respond to the speed of the MPU, this signal can be used to delay the MPU operations.
- Bus Request: When the MPU operations are too slow compared to the speed of a peripheral, the peripheral can request the use of the buses. For example, when large amounts of data are to be transferred to memory, DMA (Direct Memory Access) controllers can transfer data much faster than can the MPU.

In our generalized MPU model (Figure 2.1), these externally initiated signals are shown as External Requests. To indicate its response to some of these external requests, the MPU needs additional signal lines shown as Request Acknowledge.

2.13 Clock Signals and Power

The MPU can be viewed as a complex timer. The timing is very critical in all its operations. The bits of a binary instruction are associated with the microprograms inside the chip; when the MPU executes an instruction, it releases a series of microprograms at precise time intervals. Therefore, the MPU needs circuits that generate clock signals. In addition, it needs electrical power to run all the operations.

Figure 2.1 shows all the signals necessary for our generalized MPU. Presently, because of LSI technology, most of the MPU requirements can be satisfied by single-chip microprocessors with slight variations. For example, the Z80 microprocessor has all the signals of the MPU except clock-generating circuitry, and some of its control signals need to be logically ANDed to generate the specific control signals shown in Figure 2.1.

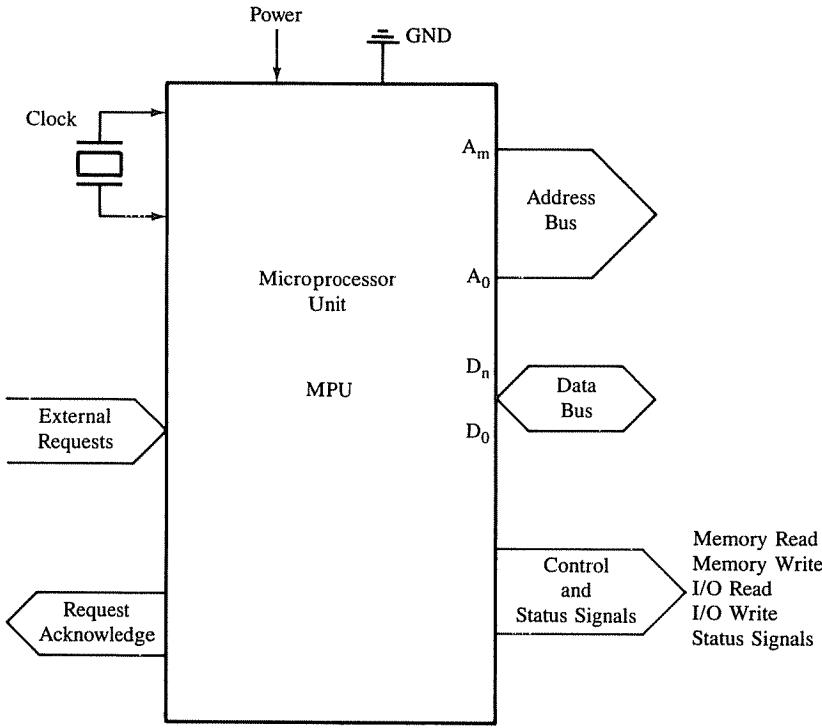


FIGURE 2.1
Generalized Microprocessor Unit (MPU)

However, the present microprocessors include all the data processing and timing circuitry on one chip; therefore, they can be viewed as MPUs. Now we shall examine what is inside the microprocessor to understand how it processes data.

2.14 Microprocessor as a Processing Unit

When the microprocessor executes instructions, it does so in a continuous sequence of fetch, decode, and execute operations. After examining these operations in more detail, we can describe the requirements of the internal architecture of our generalized microprocessor.

FETCHING AN INSTRUCTION

To fetch an instruction, the microprocessor places a memory address on the address bus and reads binary information using the data bus. Therefore, it needs a register that can hold memory addresses and increment these addresses after the fetching is completed, a sort of memory pointer.

DECODING AN INSTRUCTION

Once an instruction byte is fetched, it needs to be decoded to answer the following:

- Is it a complete instruction? If not, how many more bytes need to be fetched?
- What type of operation is required and on what data?

To perform these functions, the microprocessor needs an instruction decoder that can interpret the fetched binary information.

EXECUTING AN INSTRUCTION

The type of data manipulation the microprocessor can perform depends on its internal microprograms, that is, on its instruction set. These operations can be classified as data copy (transfer), arithmetic/logic operations, and decision making. For example, to subtract two numbers, both numbers must be loaded into registers. After the subtraction, it is necessary to indicate whether the result is positive, negative, or zero. This can be indicated by setting or resetting flip-flops called flags. To perform these arithmetic and logic operations, the microprocessor needs a group of logic circuits called Arithmetic/Logic Unit (ALU).

This description of the requirements of the microprocessor to process data can be summarized in a simplified block diagram shown in Figure 2.2. From this block diagram, we can derive a programming model for a specific microprocessor.

2.15 Review of Important Concepts

The description and the requirements of a generalized microprocessor unit can be summarized as follows (see Figure 2.3):

To communicate with memory and I/O devices, the MPU should have the following:

1. Address bus to send the address of a memory register or an I/O.

FIGURE 2.2
MPU Internal Structure

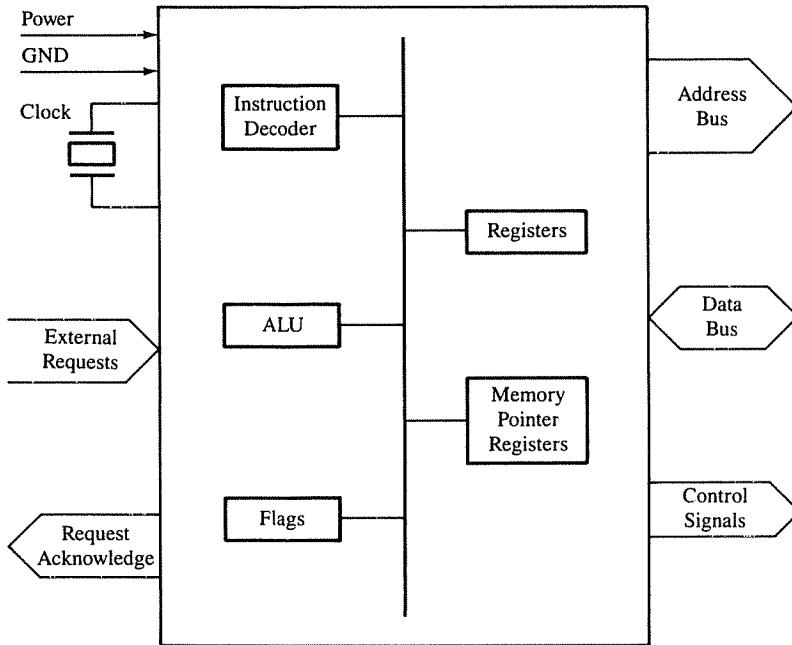
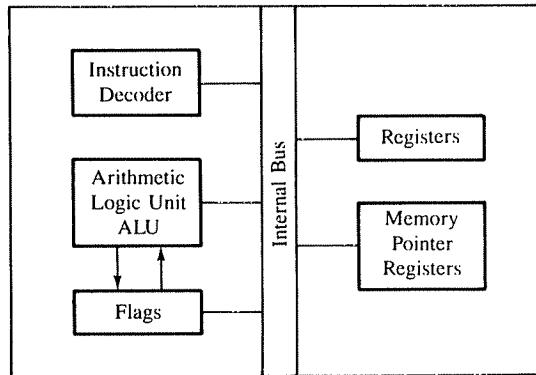


FIGURE 2.3
MPU Architecture

2. Data bus to transfer data between the MPU and memory and I/O devices.
3. Control signals to identify its operations and provide timing.
4. External Request signal lines to interrupt the MPU operations.
5. Request Acknowledge signals to respond to the requests by peripherals.
6. Clock signals to provide timing and power to operate circuits.

To process data internally, the MPU should include the following:

1. Instruction Decoder to decode the fetched binary information.
2. Registers to store binary data.
3. Registers as memory pointers for addressing memory registers.
4. ALU to perform arithmetic and logic operations.
5. Flags (flip-flops) to indicate data conditions for decision making.

2.2

MEMORY

Memory is an essential component of a microcomputer system; it stores binary instructions and data for the microprocessor. There are various types of memory, and they can be classified in two groups: prime (or main) memory and storage memory. In the last chapter, we saw two examples of prime memory: Read/Write Memory (R/WM) and Read-Only Memory (ROM). Magnetic tapes and 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 up of registers, and each register has a group of flip-flops or field-effect transistors that store bits of information. 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. The MPU uses its address bus to send the address of a memory register and uses data and control buses to read from or write into that register. In the following sections, we examine the basic concepts related to memory—its structure, its addresses, and its requirements for communication with the MPU—and build a model for R/W memory. However, the discussion is equally applicable to ROM except for slight differences in Read/Write control signals.

2.21 Flip-Flop or Latch as a Storage Element

What is memory? It is a circuit that can store bits—generally high or low voltage levels 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 and an enable signal (Figure 2.4(a)). In this latch, the stored bit is always available on the output line. If a tri-state buffer is connected to the output of the latch (as shown in Figure 2.4(b)), the stored bit can be read only when the buffer is enabled. Similarly, we can also use a tri-state buffer on the input of the latch. Now we can write into the latch (Figure 2.4(c)) by enabling the input buffer and read from it by enabling the output buffer. This latch, which can store one binary bit, is called a memory cell. Figure 2.5(a) shows four such cells or latches grouped together to form a register which has four input lines and four output lines and can store four bits. The size of this register is specified as either 4-bit or 1×4 bit, which indicates one register with four cells or four I/O lines. The number of bits stored in a register is called a memory word. Figures 2.5(b) and (c) show simplified block diagrams of the 4-bit register.

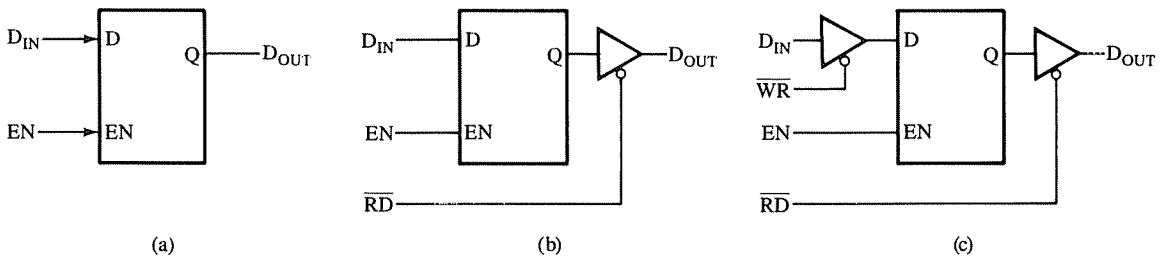


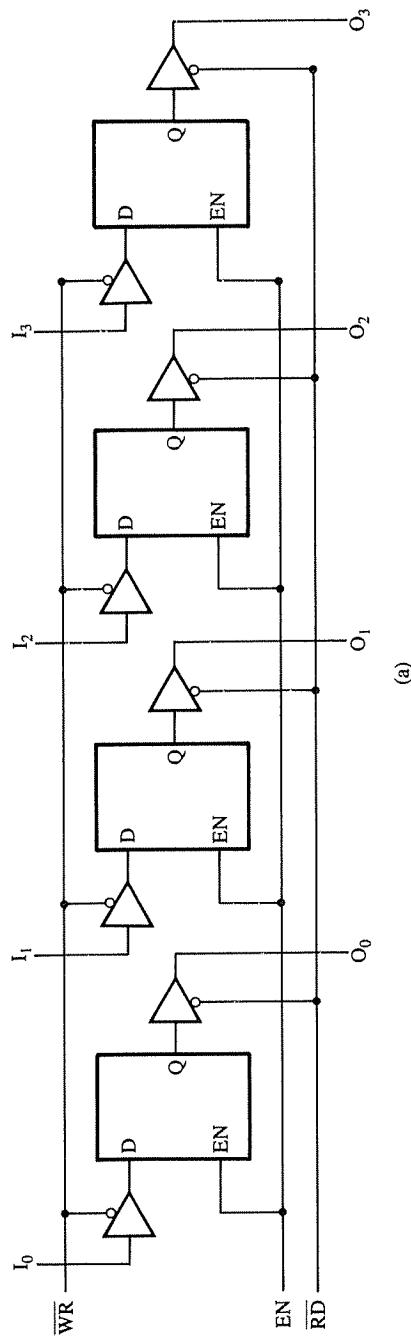
FIGURE 2.4
Latches as Storage Elements

In Figure 2.6(a), four registers with eight cells (or 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 carry 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.

In Figure 2.6(a), the chip has an 8-bit memory word, and its size can be specified as 32 bits, 4×8 bits, or 4 bytes. If we have a memory chip with a 4-bit memory word, we can combine two such chips in parallel to make an 8-bit memory word as shown in Figure 2.6(b). The address lines and RD/WR control signals ($\bar{-}$ indicates active low) will be connected in parallel, but the memory word will consist of 4 bits from each chip as shown.

Now we can expand the number of registers. If we have eight registers on one chip, we need three address lines and a 3-to-8 decoder. An interesting problem is how to deal with two chips with four registers each. We have a total of eight registers; therefore, we need three address lines. One address line, A_2 , is used to select a chip, and the address lines A_1 and A_0 are connected to both chips. Figure 2.7(b) shows that the Chip Select signal CS is active low, so that 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 sixteen (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 2.8. 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 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



(a)

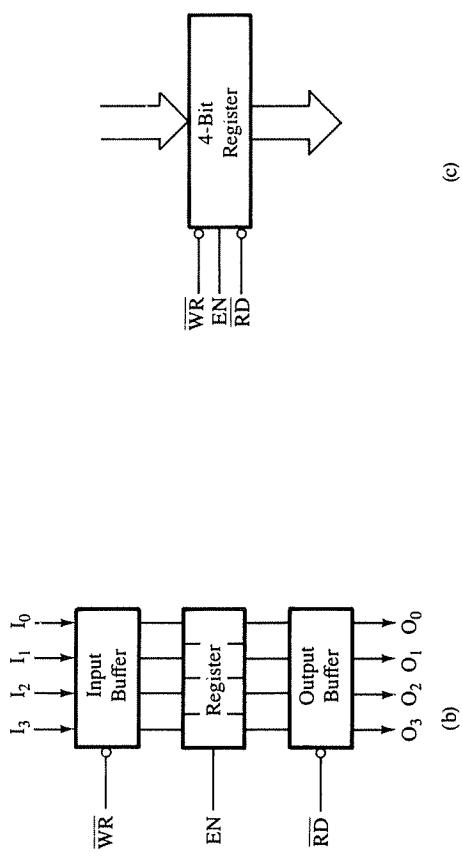
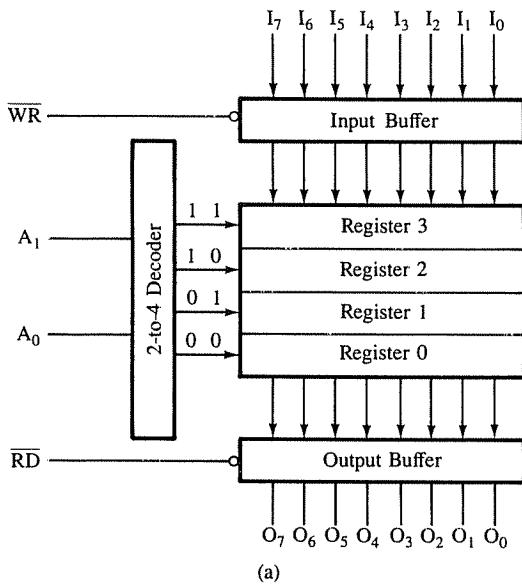
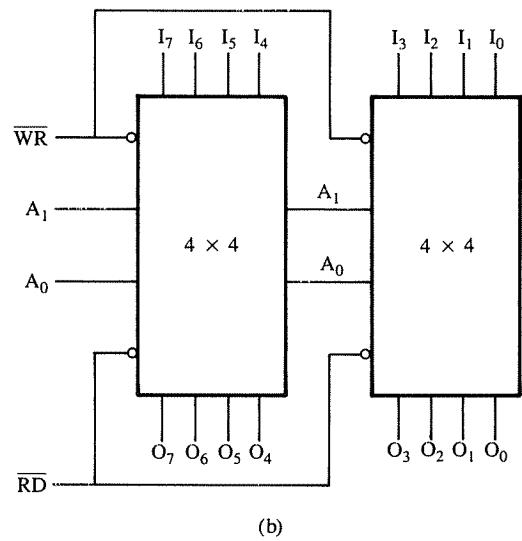


FIGURE 2.5
(a) Four Latches as a 4-Bit Register (b) and (c) Block Diagrams of a 4-Bit Register



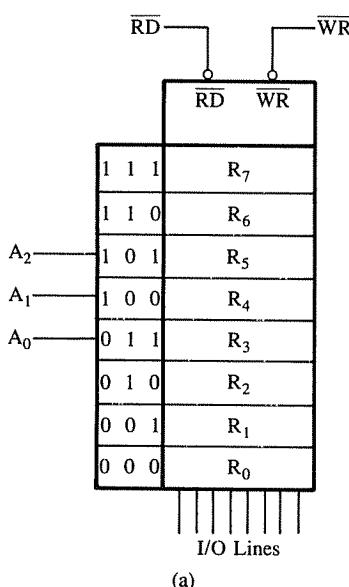
(a)



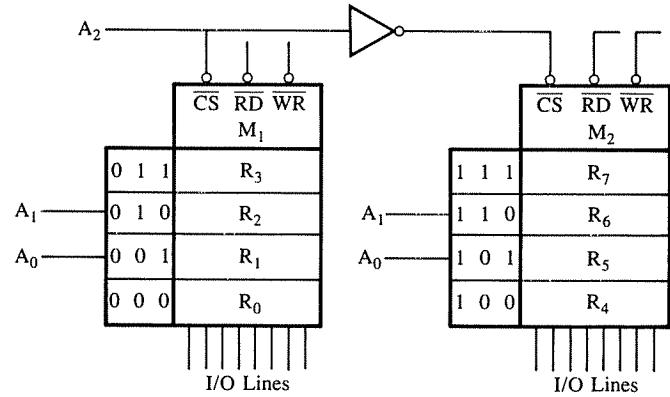
(b)

FIGURE 2.6

(a) 4 × 8 Bit Register (b) Two 4 × 4 Bit Registers



(a)



(b)

FIGURE 2.7

(a) Memory Chip with Eight Registers (b) Two Memory Chips with Four Registers Each

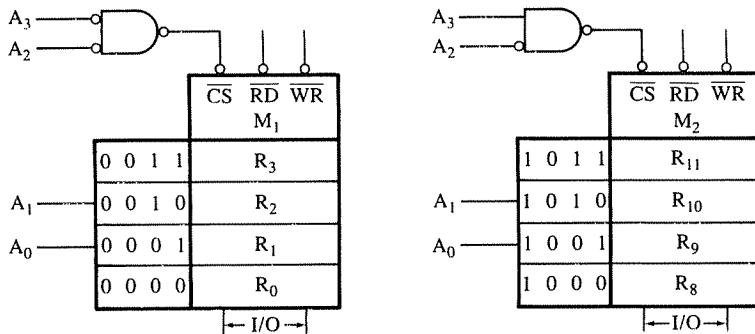


FIGURE 2.8
Addressing Eight Registers with Four Address Lines

identify eight registers, two for registers and one for Chip Select. However, we used also the fourth line for Chip Select. 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 as follows:

1. A memory chip requires address lines to identify a memory register, a Chip Select \overline{CS} signal to enable the chip, and control signals to read from and write into memory registers.
2. The number of address lines required is determined by the number of registers in a chip ($2^n = \text{Number of registers}$ where n is the number of address lines).
3. If additional address lines are available in a system, they are used to enable the Chip Select \overline{CS} signal. 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.

A model of a typical memory chip representing the requirements just stated is shown in Figure 2.9. Figure 2.9(a) represents the R/W memory and Figure 2.9(b) represents the Read-Only Memory; the only difference between the two as far as addressing is concerned is that ROM does not need a WR signal. Internally, the memory cells are arranged in a matrix format (in rows and columns), because as the size increases the internal decoding 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, however, the internal decoding circuitry can be designed with two decoders, one for selecting a row and the other for selecting a column. The internal row and column arrangement does not affect our external interfacing logic.

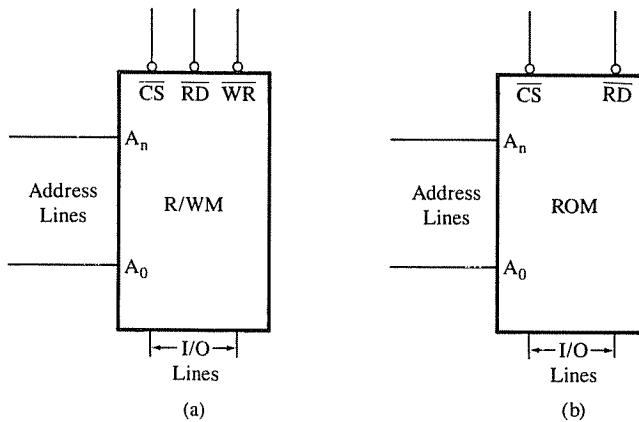


FIGURE 2.9
(a) R/W Memory Model (b) ROM Model

2.22 Memory Map

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. Memory map is like 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.

Now let us assume that we have a memory chip with 256 registers which needs only eight address lines ($2^8 = 256$). How can we assign 16-bit addresses to 256 registers? This can be accomplished by using the remaining eight lines for the Chip Select through appropriate logic gates as illustrated in the next example.

Illustrate the memory map of the chip with 256 bytes of memory, shown in Figure 2.10(a), and explain how the memory map can be changed by modifying the hardware of the Chip Select CS line in Figure 2.10(b).

Example
2.1

Solution

Figure 2.10(a) shows a memory chip with 256 registers with 8 I/O lines; the memory size of the chip is expressed as 256×8 . It has eight address lines A₇–A₀, one Chip Select CS signal (active low) and two control signals Read (RD) and Write (WR). The eight address lines (A₇–A₀) of the microprocessor are required to identify 256 memory registers. The remaining eight lines (A₁₅–A₈) 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 A₁₅–A₈ should be at logic 0, which will cause the output of the NAND gate to go low. No other logic levels on the lines A₁₅–A₈ can select the chip. Once the chip is selected (enabled), the remaining address lines A₇–A₀

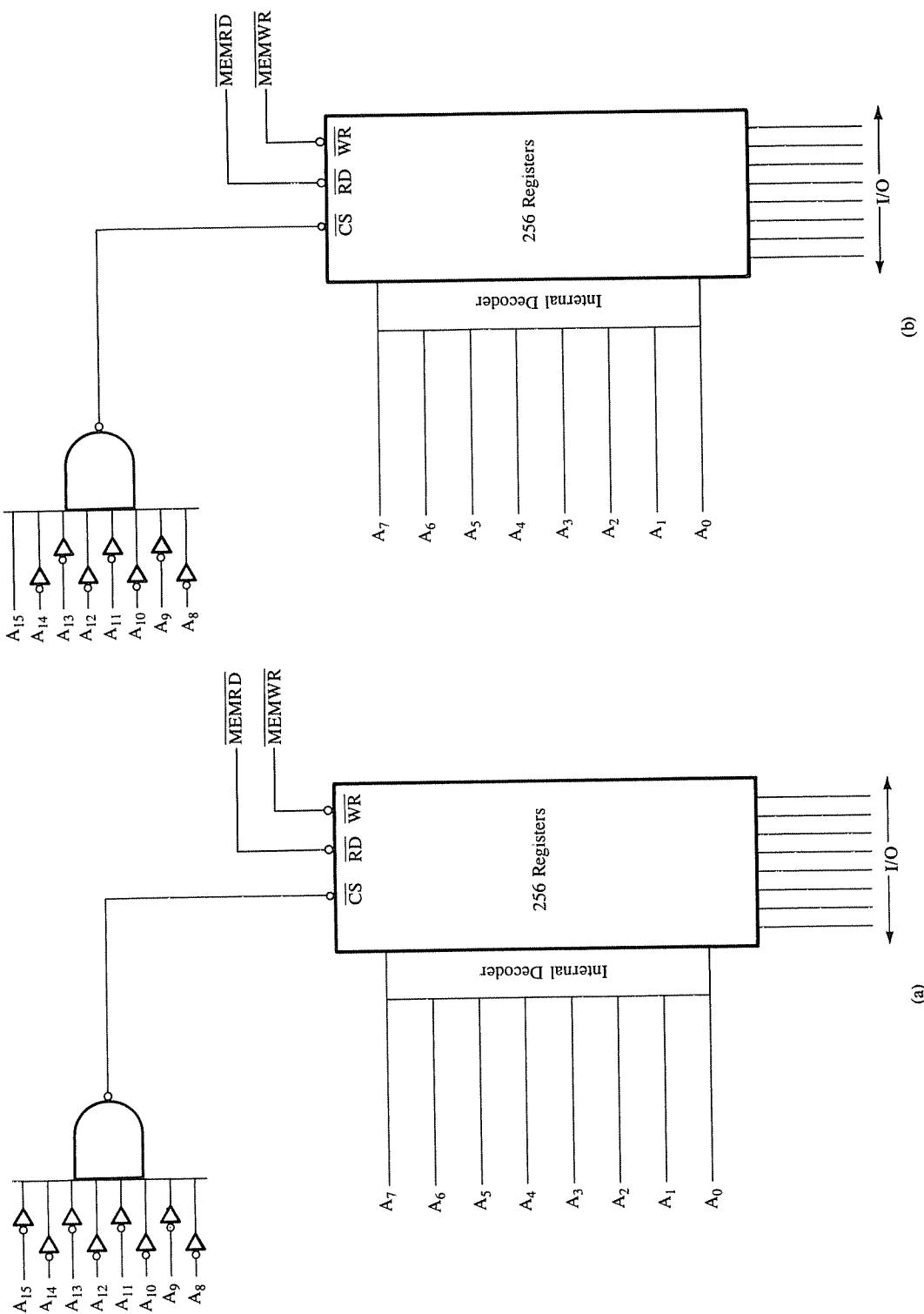


FIGURE 2.10
Memory Maps: 256 Bytes of Memory

can assume any combination from 00_H to FF_H , and identify any of the 256 memory registers through its decoder. Therefore, the memory addresses of the chip in Figure 2.10(a) will range from 0000_H to $00FF_H$ 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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$= 0000_H$
Chip Enable or Chip Select								Register Select							
↓								↓							
1								1							

The entire range of the memory addresses from 0000_H to $00FF_H$ is known as the memory map of the chip in Figure 2.10(a). The Chip Select addresses are determined by the hardware (the inverters and NAND gate); therefore, the memory map of the chip can be changed by modifying the hardware. For example, if the inverter on line A_{15} is removed as shown in Figure 2.10(b), the address required on A_{15} – A_8 to enable the chip will be as follows:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	$= 80_H$
1	0	0	0	0	0	0	0	

The memory map for Figure 2.10(b) will be 8000_H to $80FF_H$.

The memory chips in Figures 2.10(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 range of 0000 to FFFF_H.

In a memory system, a 16-bit address can be conceptually organized into two groups of Hex numbers. With two Hex digits, 256 registers can be numbered from 00_H to FF_H as shown in the previous example. This is defined as a page with 256 lines (registers) to read from or write on. Similarly, high-order Hex digits in an address can be used to number the pages from 00_H to FF_H ; thus the total range of 64K can be conceptually divided into 256 pages with each page having 256 lines. For example, the memory address $020F_H$ represents line (register) 15 on page 2, and the address $07FF_H$, represents register 255 on page 7. A memory chip with 1K (1,024) byte can be viewed as a chip with four pages. This is just a convenient way of thinking memory maps.

Another way of viewing a memory address is in terms of high-order and low-order addresses. The lines used for chip select are called high-order address lines, and the lines connected to memory address lines are called low-order address lines. Let us use an example of a four-digit (decimal) numbering system in a high-rise apartment building. Generally, the first two digits (high-order) represent a floor and the last two digits (low-order) represent an apartment number. To locate apartment 1241, we go first to the twelfth floor (similar to Chip Select in memory addressing), and then we look for the apartment 41 (similar to selecting a register). Now let us use the example of an apartment complex. Let

us assume the complex is divided into sections 1 to 9 and each section has up to 999 apartments. In this situation, the number 2451 would represent Section 2 and apartment number 451; the digit 2 is a high-order address and 451 is a low-order address. This is similar to memory addresses of 1K memory. The 1K memory chip will require 10 address lines, and the remaining six lines of the address bus will be used for the \bar{CS} . Thus, the group of six address lines will be high-order, and the remaining ten address lines will be low-order. The memory addresses will be determined by combining the logic levels of these address lines. If the number of address lines in a microprocessor is larger than 16, we will use a five-digit Hex numbering scheme.

2.23 How the MPU Writes into and Reads from Memory

To store (write) a byte into a memory location (Figure 2.11), the MPU

1. places the 16-bit address on the address bus of the memory location where a byte is to be stored. This address is decoded to select the memory chip, and the memory register is identified.
2. places the byte on the data bus.
3. sends the control signal Memory Write to enable the input buffers of the memory and then stores the byte.

To read from memory, the steps are similar.

1. The MPU places the 16-bit address on the address bus and sends the control signal Memory Read to enable the output buffer of the memory chip.

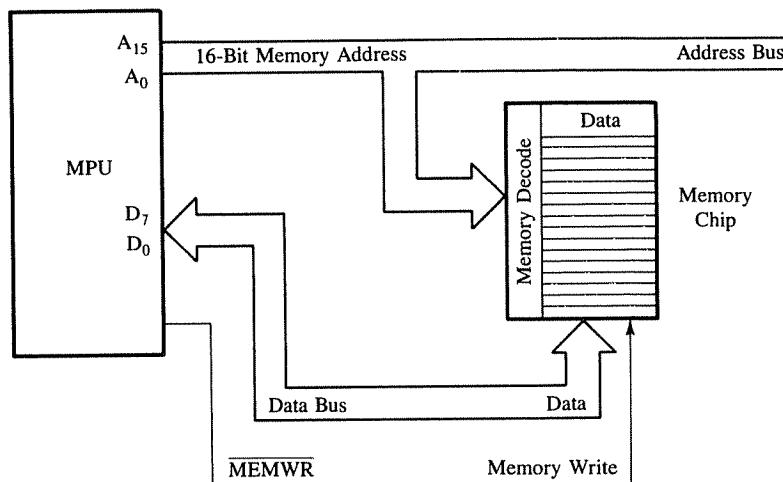


FIGURE 2.11
Memory Write Operation

2. The interfacing logic of the memory chip decodes the address and selects the appropriate memory register.
3. The memory chip places the data byte on the data bus, and the MPU reads the data byte.

2.24 Memory Classification

Memory can be classified into two groups: prime (or main) memory and storage memory. The R/WM and ROM discussed in the last section are examples of prime memory; this is the memory the microcomputer 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).

Storage memory includes examples such as magnetic disks and tapes (see Figure 2.12). This memory is used to store programs and results after the completion of program execution. Information stored in these memories is nonvolatile, meaning information remains intact even if the system is turned off. Generally, these memory devices are not a part of any system; they are made part of the system only when stored programs need to be accessed. The microprocessor cannot execute or directly process programs stored in these devices; programs must be copied into the prime memory first. Therefore, the size of the prime memory (e.g., 64K or 128K) determines how large a program the system can process. The size of the storage memory is unlimited; when one disk or tape is full, another can be used.

Figure 2.12 shows two subdivisions of storage memory: secondary storage and backup storage. The secondary storage is similar to what you put on your shelf in your study, and the backup is similar to what you store in your attic. Storage memory includes such devices as disks, magnetic tapes, magnetic bubble memory, and charged-coupled devices (CCD). The primary features of all 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 semi-random. The remaining devices shown in Figure 2.12 are serial: if information is stored in the middle of the tape, it can be accessed only after running half the tape. We will discuss some of these memory storage devices again in Chapter 7. In this chapter, we will focus on various types of prime memory.

Figure 2.12 shows that the prime memory is divided into two main groups: Read/Write Memory (R/WM) and Read-Only Memory (ROM), and each group includes several different types of memory.

R/WM (READ/WRITE MEMORY)

As the name suggests, the microprocessor can write into or read from this memory, and 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 its contents are destroyed.

Two types of R/W memories—static and dynamic—are available. Static memory is

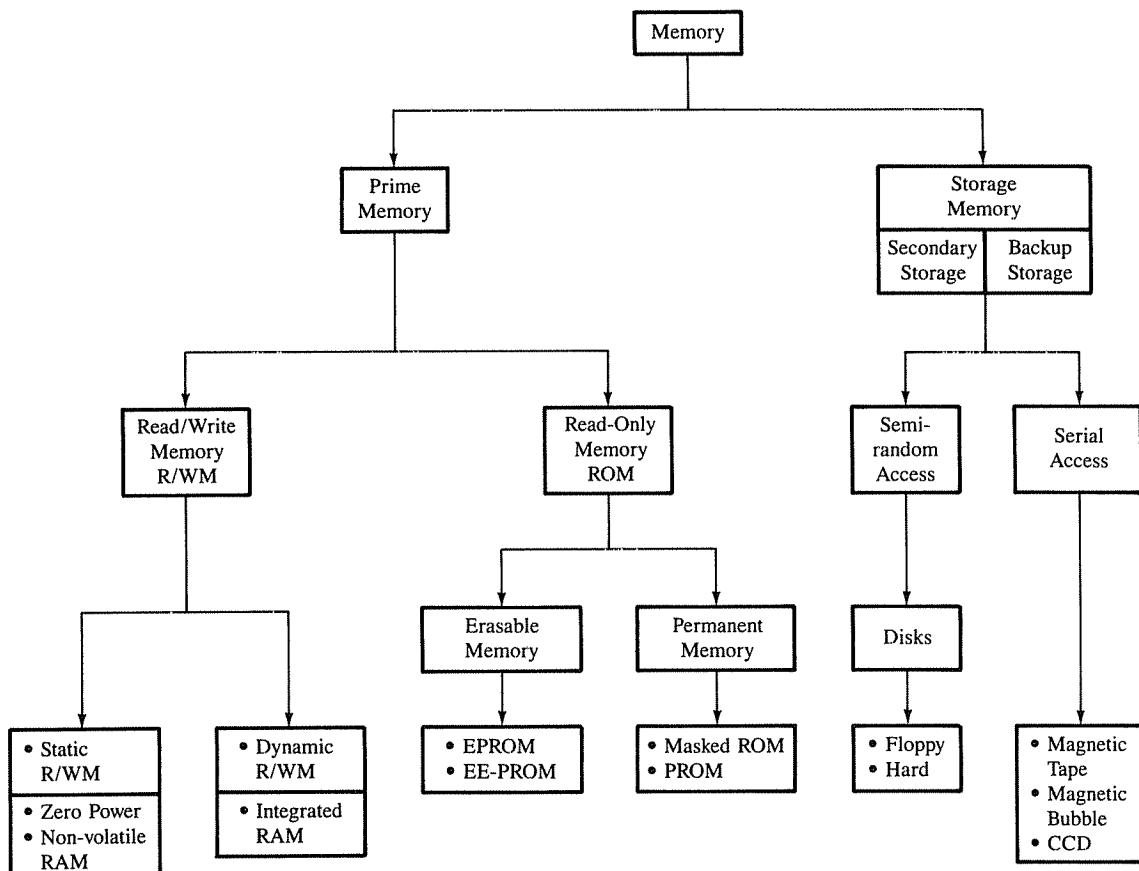


FIGURE 2.12
Memory Classification

made up of flip-flops, and it stores the bit as a voltage. Dynamic memory is made up of MOS transistor gates, and it stores the bit as a charge. The advantages of the dynamic memory are that it has higher density, lower power consumption, and is cheaper than the 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, which adds to the cost of the system. It is generally economical to use dynamic memory when the system memory size is larger than 16K; for smaller systems, the static memory is appropriate.

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 because, as the

name suggests, the information can be read only so that once a bit pattern is stored, it is permanent or at least semi-permanent. The permanent group includes two types of memory: masked ROM and PROM, and the semi-permanent group also includes two types of memory: EPROM and EE-PROM as shown in Figure 2.12.

MASKED ROM

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

PROM (PROGRAMMABLE READ-ONLY MEMORY)

This memory has nichrome or polysilicon 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 a 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.

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 in which timing information has to be changed, it can be done 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 as opposed to 15 to 20 minutes for an EPROM.

RECENT ADVANCES IN MEMORY TECHNOLOGY

Memory technology has advanced considerably in recent years. In addition to static and dynamic R/W memory, there are now more options available in memory devices. Recent examples include Zero Power RAM from MOSTEK, Non-Volatile RAM from Intel, and Integrated RAM from several manufacturers.

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 both R/W and Read-Only Memory.

The Non-Volatile RAM is a high speed static R/W Memory array backed up, bit for bit, by an 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 a STORE signal on the memory chip, and the stored data can be read into the R/W memory segment when the power is turned on again. 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 refreshing circuitry. At present, this memory is economical for a system with medium-sized memory (between 8K and 64K).

2.3

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 an MPU can identify I/O devices: one uses an 8-bit address and the other a 16-bit address. These methods are described briefly in the following sections.

2.31 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 also known as peripheral-mapped I/O. 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 00_H to FF_H . The input and output devices are differentiated by the control signals I/O Read for input devices and I/O Write for output devices. The entire range of I/O addresses from 00_H to FF_H is also known as I/O map, and individual addresses are also 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.

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 the external decode logic (explained in Chapter 5).
2. The MPU sends a control signal (I/O Read or I/O Write) to enable the I/O device.
3. Data are transferred on the data bus.

2.32 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. In memory-mapped I/O, the MPU uses the same control signals (Memory Read or Memory Write) and instructions as those of memory and follows the same steps as when it is accessing a memory register. In some microprocessors, such as the Motorola 6800, all I/Os have 16-bit addresses so that I/Os and memory share the same memory map (64K).

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

2.4

EXAMPLE OF A MICROCOMPUTER SYSTEM

In the last three sections, we discussed a generalized MPU model, prime memory and its organization model, and I/Os. The discussion can be summarized in the block diagram of a microcomputer system as shown in Figure 2.13. It includes a generalized MPU, two types of prime memory, and two I/O devices.

All address lines are used to address memory, and only the low-order address bus is used to identify I/O devices, indicating that they are connected as peripheral-mapped I/O (the details of Chip Select decoding are omitted here). The data bus is bidirectional and

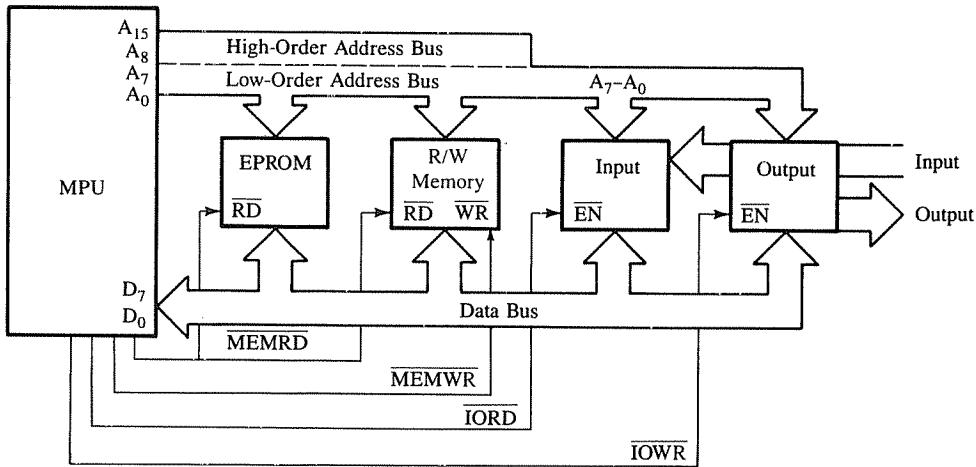


FIGURE 2.13
Example of a Microcomputer System

common to all devices. The four control signals generated by the MPU are connected to different peripherals, as shown in Figure 2.13.

HOW DOES THE SYSTEM WORK?

Let us assume that a simple program with three instructions is already written and stored in binary in R/W memory. Those instructions are

1. Read Input Port No. 20_H .
2. Display the Data at the Output Port 80_H .
3. Stop.

To execute these instructions, the MPU does the following:

1. Places the memory address bus of the instruction 1 and fetches the instruction using the control signal Memory Read (MEMRD). (The MPU may have to fetch instruction codes more than once if the instruction has more than one byte.) It decodes the instruction.
 - Reads the input port by placing the address bus 20_H , reads data using the control signal I/O Read (IORD), and stores the data in one of the registers.
2. Fetches the next instruction by placing the memory address of that instruction and the control signal MEMRD. Then, it decodes the instruction.
 - Places the port address 80_H and transfers the data using the control signal I/O Write (\overline{IOWR}).
3. Again fetches the last instruction from memory as before, decodes it, and stops.

This is a simplified description of how the system works; it excludes the details about multi-byte instructions, machine cycles, and timing.

SUMMARY

In this chapter, we examined the requirements of the Microprocessor Unit (MPU) to communicate with memory and I/O devices and to process binary data. Based on those requirements, we designed a generalized model of the MPU. We discussed memory in terms of its storage elements, namely, latches and registers and techniques of assigning addresses. The steps required for the MPU to communicate with memory and I/Os were briefly described. The important concepts are summarized as follows.

- The MPU performs four primary operations: Memory Read, Memory Write, I/O Read, and I/O Write.
- To communicate with memory and I/Os, the MPU needs three types of buses: the unidirectional address bus to send memory and I/O addresses, the bidirectional data bus to transfer data, and control signals to enable the devices.

- The MPU should have signal lines to accept and to acknowledge external requests. These requests are Reset (go back to beginning), interrupt (stop the ongoing process and attend to something urgent), wait to synchronize with slow memory, and allow the use of buses to an external device because the MPU response time is slower than that of the external device.
- To process data, the MPU should include registers to store data, memory pointers to hold memory addresses, ALU to perform arithmetic and logic operations, and flags to indicate data conditions.
- Memory is a group of registers, arranged in a sequence, to store bits. The number of cells (latches) in a register determines the size of the memory word in a chip.
- A memory chip requires address lines to identify a memory register, Chip Select signal to select the chip, and control signals to read from and write into memory registers.
- The range of memory addresses assigned to a memory chip in a system is called the memory map. The assignment of memory addresses is done through the Chip Select logic.
- An I/O device can be identified either with an 8-bit address called the peripheral-mapped I/O or with a 16-bit address called the memory-mapped I/O.
- To communicate with memory or I/O, the MPU places the address of the device on the address bus, places data on the data bus, and sends the appropriate control signal.

LOOKING AHEAD

In this chapter, we examined the microprocessor as a programmable logic device and developed a generalized model. Similarly, we discussed memory as a storage element and constructed a memory model. We examined briefly the role of I/Os as channels of communication with “the outside world.” These three elements were interconnected through a bus architecture to form a model of a microcomputer system. Then we discussed how the MPU communicates with memory and I/Os.

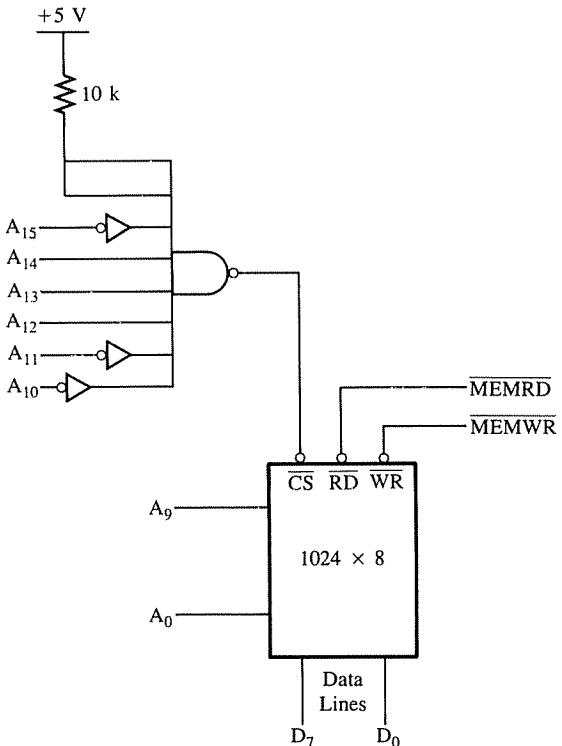
In the next three chapters, we will explore each component and its communication process separately with details and specific examples. In Chapter 3, we will examine the Z80 microprocessor in the context of our generalized model of a programmable logic device. Chapter 4 discusses memory and its interfacing, and Chapter 5 is devoted to interfacing I/O devices.

ASSIGNMENTS

1. List the four operations commonly performed by the MPU.
2. What is a bus?

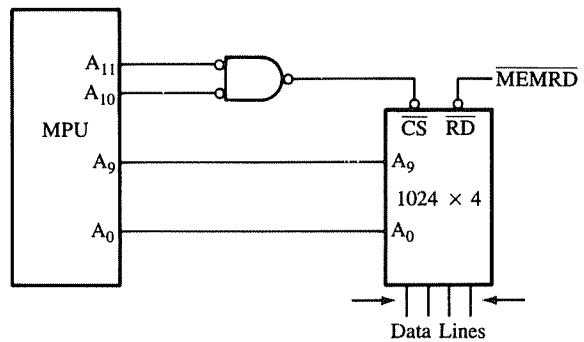
3. What is the function of the address bus?
4. How many memory locations can be addressed by the MPU with thirteen address lines?
5. How many address lines are necessary to address two megabytes (2048K) of memory?
6. What is the function of the interrupt signal and when is it used?
7. When is the bus request signal used?
8. Specify the number of registers and memory cells in a 128×4 memory chip.
9. How many bits are stored by a 256×4 memory chip? Can this chip be specified as 128-byte memory?
10. If the memory size is 1024×4 bits, how many chips are required to make up 1K-byte memory?
11. If the memory chip size is 1024×1 bits, how many chips are necessary to make up 4K (4,096) bytes of memory?
12. What is the function of the \overline{WR} signal on the memory chip?
13. How many address lines are necessary for the memory chip with 2048×8 size?

FIGURE 2.14
Identification of Memory Maps
for Assignments 17–18



14. How many address lines are necessary for the memory chip with 2048×4 size?
15. The memory map of a 4K(4,096)-byte memory chip begins at the location 8000_H . Specify the entire memory map and the number of pages in the map.
16. The memory address of the last location of an 8K-byte memory chip is $FFFF_H$. Find the starting address.
17. Identify the memory map in Figure 2.14. List the high-order and low-order address lines. How many pages of memory does the chip include?
18. In Figure 2.14, identify the memory map if the inverter of the address line A15 is eliminated and A15 is connected directly to the NAND gate.
19. Figure 2.15 shows an MPU with the address bus containing 12 address lines and the data bus with four data lines; it is interfaced with the 1K-byte memory chip. Find the memory map.
20. Specify the size of the memory word shown in Figure 2.15.

FIGURE 2.15
Identification of Memory Maps for
Assignments 19–20



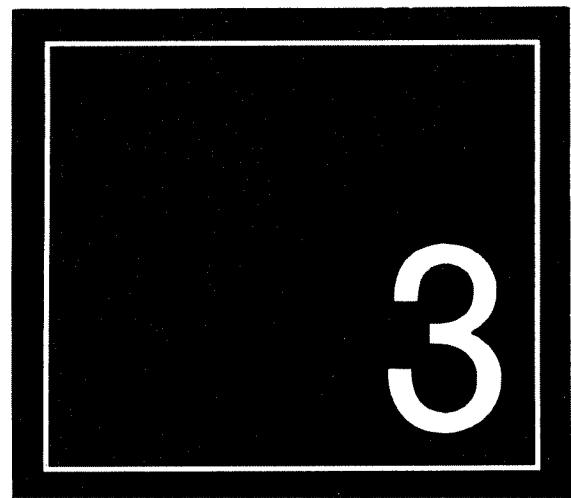
Z80

Microprocessor: Programming Model and Hardware Model

The Z80 is one of the most versatile and widely used 8-bit microprocessors, and many microcomputer systems are designed around the Z80. The Z80 chip includes most of the logic circuitry for performing computing tasks and necessary bus signals. This chapter discusses the Z80 architecture in terms of two models: the programming model and the hardware model derived from the generalized MPU discussed in the previous chapter.

We will describe the programming model first because it provides the overview of the Z80 architecture. This information is essential to hardware designers as well as programmers. The model describes the accumulator, internal 8-bit and 16-bit registers, and their functions during the execution of a program. The description also includes the details of the flags and data conditions under which they are set or reset, information very critical to the programmer.

The hardware model shows logic pinout of the chip and classifies the signals in various groups according to their functions. The model lists the operations the Z80 frequently performs and describes how the Z80 communicates with memory and I/Os by using various buses. These operations are illustrated in terms of machine cycles and logic levels of the buses in relation to the system clock.



Finally, the chapter includes the discussion of other contemporary 8-bit microprocessors in terms of the generalized model developed in the last chapter and compares them with the Z80.

OBJECTIVES

- Draw the Z80 programming model and identify the registers.
- Explain the functions of the accumulator, general-purpose registers, and alternate registers.
- Explain the functions of 16-bit registers and special-purpose registers.
- List the flags and explain the data conditions under which they are set or reset.
- List the functional groups of the Z80 signals.
- Define the address bus, the data bus, and the control signals, and explain their functions.
- List the types of external signals and explain their purposes.
- List three categories of the Z80's operations.
- Explain the terms instruction cycle, machine cycle, and T-state.
- List the steps the Z80 performs to execute the Opcode Fetch, the Memory Read, and the Memory Write cycles, and explain their functions.
- Show the bus contents and the appropriate control signals in reference to the system clock when these machine cycles are executed.
- Describe the 8085, the NSC800, and the 6800 microprocessors in terms of the generalized MPU and compare them with the Z80.

3.1 THE Z80 PROGRAMMING MODEL

In the last chapter, we developed a model to represent the internal structure of the MPU shown in Figure 2.3. We will now describe a similar model of the Z80 microprocessor; however, we will include only those components necessary for the programmer. Figure 3.1 shows such a model, which includes an **accumulator** and a **flag register**, **general-purpose register arrays**, registers used as **memory pointers**, and **special-purpose registers**. These registers and their functions are described in the following sections.

3.11 Accumulator

The accumulator is an 8-bit register that is part of the Arithmetic/Logic unit (ALU) and is also identified as register A. This register is used to store 8-bit data and to perform arithmetic and logic operations. The result of an operation performed in the ALU is also stored in the accumulator. For example, in an 8-bit addition, the instruction ADD always assumes that one of the numbers is the byte in the accumulator, and the result of the addition is stored in the accumulator by replacing the previous byte.

Figure 3.1 shows an additional accumulator called A' in the alternate register set. A' is not directly accessible to store a byte or perform an ALU operation, but the contents of A' are accessible by exchanging its contents with the contents of the accumulator A.

3.12 Flag Register

The ALU includes six flip-flops that are set or reset according to data conditions after an ALU operation, and the status of each flip-flop, also known as flags, is shown in the flag register F. The status of each of the six flags is stored in the 8-bit flag register so that they

Alternate Registers	
Accumulator A	Flags F
B	C
D	E
H	L
Index Register IX	
Index Register IY	
Stack Pointer SP	
Program Counter PC	
Interrupt Vector I	Memory Refresh R

FIGURE 3.1
The Z80 Programming Model
SOURCE: Courtesy of Mostek Corporation

FIGURE 3.2
Flag Register: Bit Identification

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	
S	Z		H			P/V	N	C

can be examined if necessary. The bit position of each flag is shown in Figure 3.2; bits D₅ and D₃ are unused.

Among the six flags, the H (Half-Carry) and N (Add/Subtract) flags are used internally by the microprocessor for **BCD (Binary Coded Decimal)** operations. These two flags cannot be tested by any instruction and are not available to the programmer for decision making. The remaining four flags—S (Sign), Z (Zero), P/V (Parity/Overflow), and C (Carry)*—can be tested in conjunction with Conditional Jump or Call instructions.

*To avoid confusion between C as a register and C as the Carry flag, we will refer to the Carry flag as CY when it does not refer specifically to bit D₀ in the flag register.

Each of these four flags has two Jumps or Call instructions associated with it: one when the flag is set and the other when the flag is reset. These flags have critical importance in the decision making process; all decisions are based on the status of these flags. For example, the instruction **JP C, 2050H** (Jump on Carry to memory location 2050_{16}) is implemented to change the sequence of a program when the Carry flag is set.

The details of these flags are described below in the order of frequency of use. They will be discussed again in the context of illustrative programs. At the outset, the descriptions of these flags may appear quite complex. However, when we begin to write programs, we will see that, in general, most flags are ignored except one or two depending upon the operations being performed. For the time being, to understand their function, you should focus on three flags: C (Carry), Z (Zero), and S (Sign).

- **C—Carry flag:** If an arithmetic operation generates a carry (in addition) or a borrow (in subtraction), the Carry flag is set; otherwise it is reset.
It is important to remember that when an arithmetic operation does not generate a carry (or borrow), the flag is reset.
The flag is also affected by such other instructions as logic and shift instructions. The details will be discussed when specific instructions are explained.
The Z80 includes instructions **SCF**—Set Carry Flag—and **CCF**—Complement Carry Flag—that can set or complement this flag independent of the previous ALU operation.
- **Z—Zero flag:** If an 8-bit operation results in zero, the Z flag is set; otherwise it is reset.
In a bit testing operation, if the bit is zero, this flag is set; otherwise it is reset.
In comparing two numbers, the Z flag is set when they are equal; otherwise it is reset.
The Z flag is also affected by special input instruction, block I/O instructions, and counting instructions.
- **S—Sign flag:** After an ALU operation, if the most significant bit D_7 is 1, the sign flag is set; otherwise it is reset. When the flag is set, you do not necessarily have a negative result. The interpretation of the Sign flag depends upon the number system (unsigned number, signed magnitude, or 2's complement) being used by the programmer. This flag can, of course, be used to indicate negative numbers, but its usage can be confusing. Therefore, it is discussed in detail in the context of the appropriate instructions. This flag is also affected by special input instructions in the Z80 set.
- **P/V—Parity/Overflow flag:** This flag is used for two purposes: to check the parity (the number of 1s in a byte) and to check an overflow in dealing with signed numbers.
In the case of parity check after an operation, if the number of 1s in the result is even (even parity), this flag is set, and if the number of 1s is odd (odd parity), the flag is reset. For example, if the result, of ANDing two bytes is 0 0 0 0 0 0 1 1, the parity flag is set to indicate even parity (two 1s). In this example, the magnitude base-ten (3_{10}) is odd; however, the odd or even number has no relationship with the odd or the even parity.

In arithmetic operations of signed numbers where bit D₇ is used to indicate sign, this flag is set to indicate an overflow condition. For example, when bit D₇ is reserved for a sign, the magnitude of a number is represented by the remaining seven bits, the maximum being 0 1 1 1 1 1 1 (+ 127₁₀). After an addition, if the sum goes beyond +127, bit D₇ changes to 1, a change that would indicate a negative result. In fact, this is an overflow condition and it is indicated by the overflow (V) flag.

This flag is also used for other functions such as block transfer, search, and interrupt.

- **H—Half-Carry flag:** In an arithmetic operation, this flag is affected by the carry or borrow between bits D₃ and D₄. In addition, when there is a carry from bit D₃ to D₄, the Half-Carry flag (H) is set; otherwise, it is reset. In a subtraction, when there is a borrow from bit D₄ to D₃, this flag is set; otherwise, it is reset.
The flag is used internally for BCD (Binary Coded Decimal) operations, and there are no Jump or Call instructions associated with this flag.
- **N—Add/Subtract flag:** This flag is also used internally for BCD operations to distinguish between addition and subtraction. For BCD addition, this flag is 0 and for subtraction it is set to 1.

The **alternate flag register F'** is associated with the alternate accumulator A' as shown in Figure 3.1. The contents of this register can be accessed by using the exchange instruction.

3.13 General-Purpose and Alternate Registers

The Z80 microprocessor has six programmable general-purpose registers named B, C, D, E, H, and L, as shown in Figure 3.1. These are 8-bit registers used for storing data during the program execution. They can be combined as register pairs—BC, DE, HL—to perform 16-bit operations or to hold memory addresses.

The programmer can use these registers to *load* or copy data. For example, the instruction LD B, C copies the data from register C into register B. Conceptually, these registers can be viewed as memory locations, except that they are built inside the microprocessor and identified by specific names. Some microprocessors do not have this type of register; instead, they use memory as their registers.

In addition to the general-purpose registers, the Z80 includes a similar set of six **alternate registers** designated as B', C', D', E', H', and L'. These are 8-bit registers used for exchanging data with the general-purpose registers. They are not directly available to the programmer, except through the exchange instructions.

3.14 16-Bit Registers as Memory Pointers

The Z80 microprocessor includes four 16-bit registers used to hold memory addresses; they are classified here as memory pointers. The primary function of memory is to store instructions and data, and the microprocessor needs to access memory registers to read these instructions and data. To access a memory register, the microprocessor identifies the register by using the addresses in these memory pointers.

INDEX REGISTERS (IX AND IY)

The Z80 has two 16-bit **index registers** called IX and IY. Each register is used to specify a memory address by the 16-bit address it holds and a displacement count. For example, if the IX register holds 2050_H , a higher memory address such as 2060_H can be specified by adding the displacement count of 10_H . Similarly, a lower memory address such as 2040_H can be specified by adding the negative of 10_H in 2's complement.

In addition to the index registers, the HL pair is frequently used as a memory pointer. Similarly, the BC and DE pairs can be used also as memory pointers in a limited way. However, no displacement byte can be added to the contents of these pairs.

STACK POINTER (SP)

The stack pointer is also a 16-bit register used to point to the memory location called the **stack**. The stack is a defined area of memory locations in R/W memory, and the beginning of the stack is defined by loading a 16-bit address into the stack pointer.

We will discuss the concept of the stack memory in detail when we introduce the topic of subroutines.

PROGRAM COUNTER (PC)

This register functions as a 16-bit counter. The microprocessor uses this register to sequence the execution of instructions. The program counter points to the memory address from which the next byte is to be fetched, and when the microprocessor places an address on the address bus to fetch the byte from memory, it then increments the program counter by one to point to the next memory location.

3.15 Special-Purpose Registers

The Z80 microprocessor includes two special-purpose registers generally not found in other 8-bit microprocessors. These registers are shown in Figure 3.1 as interrupt vector register (I) and the memory refresh register (R).

INTERRUPT VECTOR REGISTER (I)

This is an 8-bit register used in the interrupt process. When an external device interrupts the microprocessor with a request to do something else, the microprocessor should be directed to a 16-bit address in memory where it can find what to do next. The I register is used to store the high-order eight bits of the 16-bit address; the low-order eight bits must be supplied by the interrupting device. We will discuss the details and applications of this register in Chapter 12.

MEMORY REFRESH REGISTER (R)

The memory refresh register (R) is also an 8-bit register which is used as a 7-bit counter to provide an address of memory cells to be refreshed in dynamic memory. As mentioned in the previous chapter, information stored as a capacitive charge in dynamic memory leaks; therefore, bit information should be refreshed, meaning it should be read and stored again

every few milliseconds. Applications of the memory refresh register (R) will be discussed in detail with the topic of Interfacing Dynamic Memory.

3.16 Using the Programming Model

In this section, we will illustrate what happens to the contents of some of the registers in the microprocessor when a series of instructions is executed as shown in Example 3.1.

Write instructions in English-like statements to load the two data bytes 53_H and $C9_H$ into registers A and B respectively. Add the two bytes. Illustrate the contents of registers affected in the programming model after the execution of each instruction and the status of the Carry and Zero flags.

Example
3.1

Solution

A	53	X	F	1. Load A with 53_H .
B	C9	X	C	2. Load B with $C9_H$.

(a)

A	1C	C = 1, Z = 0		3. Add registers A and B.
B	C9	X		

(b)

FIGURE 3.3
Register Contents

Figure 3.3(a) shows the contents of registers A and B after the execution of the first two instructions. The next instruction adds the contents of registers A and B; the sum is $11C_H$. Figure 3.3(b) shows the accumulator with $1C_H$ and the CY flag set in the flag register. Please note that the flags are not affected by the Load or Copy instructions.

Z80 HARDWARE MODEL

3.2

The Z80 hardware model described in this section represents the microprocessor unit (MPU) as defined in Chapter 2. The Z80 microprocessor almost qualifies as an MPU, except that an external oscillator circuit is required to provide the operating frequency and appropriate control signals need to be generated to communicate with memory and I/O. In

the following sections, we describe the Z80 microprocessor in relation to the model we developed in the previous chapter. Then we examine the timing involved in reading an instruction from memory and generate the necessary control signals by using appropriate logic gates.

3.21 The Z80 Microprocessor

The Z80 is a general-purpose 8-bit microprocessor with 16 address lines and requires a single +5 V power supply. It is housed in a 40-pin dual-in-line (DIP) package. The different versions of Z80 microprocessors such as Z80, Z80A, Z80B, and Z80H are rated to operate at various frequencies ranging from 2.5 MHz to 8 MHz. Even though the Z80 instruction set is upward compatible with the Intel 8080 set, neither of these microprocessors are pin compatible.

Figure 3.4 shows pin configuration of the Z80 microprocessor and its hardware model with logic signals. All the signals can be classified into six groups: (1) address bus, (2) data bus, (3) control signals, (4) external requests, (5) request acknowledge and special

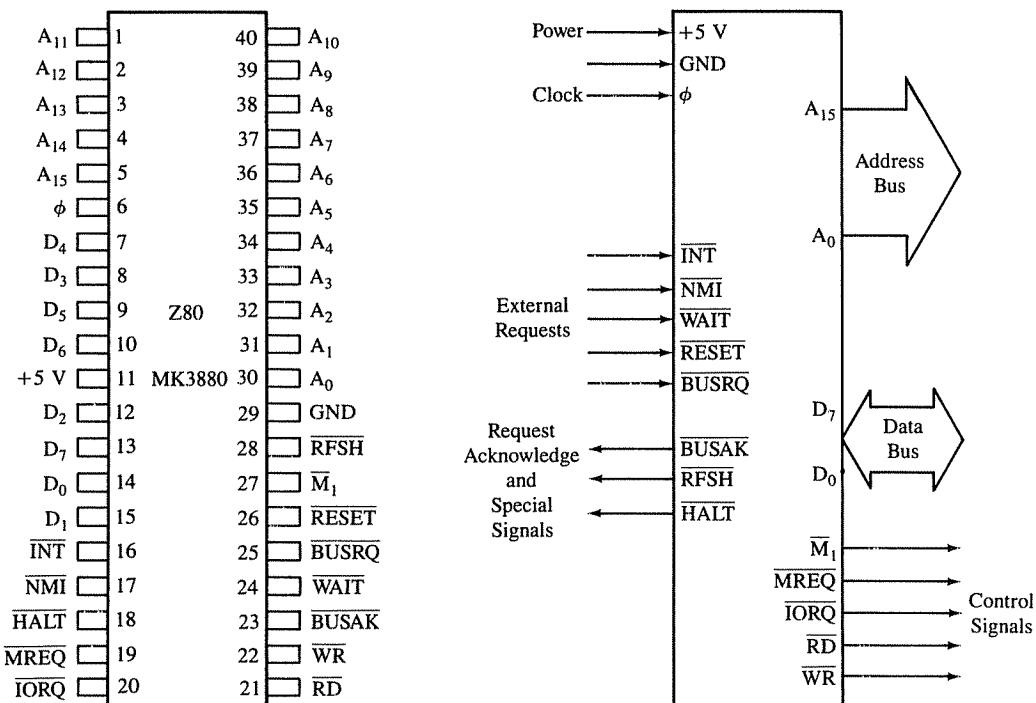


FIGURE 3.4
Z80 Microprocessor Pinout and Logic Signals

SOURCE: Courtesy of Mostek Corporation

signals, and (6) power and frequency signals. This Z80 hardware model matches the hardware model of the generalized MPU described in Chapter 2. The specific details of these signals follow.

ADDRESS BUS

The Z80 has 16 tri-state signal lines, $A_{15}-A_0$, known as the address bus. These lines are unidirectional and capable of addressing 64K (2^{16}) memory. The address bus is used to send (or place) the addresses of memory registers and I/O devices.

DATA BUS

The data bus consists of eight tri-state bidirectional lines D_7-D_0 and is used for data transfer. On these lines, data can flow in either direction—from the microprocessor to memory and I/Os or vice versa.

CONTROL SIGNALS

This group consists of five individual output lines: three can be classified as status signals indicating the nature of the operation being performed, and two as control signals to read from and write into memory or I/Os.

- M₁—Machine Cycle One: This is an active low signal indicating that an opcode is being fetched from memory. This signal is also used in an interrupt operation to generate an interrupt acknowledge signal, which will be explained in Chapter 12.
- MREQ—Memory Request: This is an active low tri-state line. This signal indicates that the address bus holds a valid address for a memory read or write operation.
- IORQ—I/O Request: This is an active low tri-state line. This signal indicates that the low-order address bus (A_7-A_0) holds a valid address for an I/O read or write operation. This signal is also generated for an interrupt operation.
- RD—Read: This is an active low tri-state line. This signal indicates that the microprocessor is ready to read data from memory or an I/O device. This signal should be used in conjunction with MREQ for the Memory Read (MEMRD) operation and with IORQ for the I/O Read (IORD) operation.
- WR—Write: This is an active low tri-state line. This signal indicates that the microprocessor has already placed a data byte on the data bus and is ready to write into memory or an I/O device. This signal should be used in conjunction with MREQ for the Memory Write (MEMWR) operation and with IORQ for the I/O Write (IOWR) operation.

EXTERNAL REQUESTS

This group includes five different input signals to the microprocessor from external sources. These signals are used to interrupt an ongoing process and to request the microprocessor to do something else.

- RESET—Reset: This is an active low signal used to reset the microprocessor. When RESET is activated, the program counter (PC), the interrupt register (I), and the memory refresh register (R) are all cleared to 0. During the reset time, the address bus and

the data bus are in high impedance state, and all control signals become inactive. This signal also disables interrupt and refresh. The RESET signal can be initiated by an external key or switch and must be active at least for three clock periods to complete the reset operation.

- INT—Interrupt Request: This is an active low signal, initiated by an I/O device to interrupt the microprocessor operation. When the microprocessor accepts the interrupt request, it acknowledges by activating the IORQ signal during the M₁ cycle. The INT signal is maskable, meaning it can be disabled through a software instruction. The interrupt process will be fully discussed in Chapter 12.
- NMI—Nonmaskable Interrupt: This is a nonmaskable interrupt; it cannot be disabled. It is activated by a negative edge-triggered signal from an external source. This signal is used primarily for implementing emergency procedures. There is no signal or pin to acknowledge this signal; it is accepted provided the Bus Request signal is inactive.
- BUSRQ—Bus Request: This is an active low signal initiated by external I/O devices such as the DMA (Direct Memory Access) controller. An I/O device can send a low signal to BUSRQ to request the use of the address bus, the data bus, and the control signals. The external device can use the buses, and when its operations are complete, it returns the control to the microprocessor. This signal is used primarily for the direct memory access technique to be discussed in Chapter 16.
- WAIT—Wait: This is an active low signal and can be used by memory or I/O devices to add clock cycles to extend the Z80 operations. This signal is used when the response time of memory or I/O devices is slower than that of the Z80. When this signal goes low, it indicates to the microprocessor that the addressed memory or I/O device is not yet ready for data transfer. As long as this signal is low, the Z80 keeps adding cycles to its operation.

REQUEST ACKNOWLEDGE AND SPECIAL SIGNALS

Among the five external requests described above, only two of the requests need acknowledgement: Bus Request and Interrupt. The interrupt is acknowledged by the IORQ signal in conjunction with the M₁ signal. The Bus Request is acknowledged by a BUSAK (Bus Acknowledge). In addition, the Z80 has two special signals: HALT and RFSH.

- BUSAK—Bus Acknowledge: This is an active low output signal initiated by the Z80 in response to the Bus Request signal. This signal indicates to the requesting device that the address bus, the data bus, and the control signals (RD, WR, MREQ, and IORQ) have entered into the high impedance state and can be used by the requesting device.
- HALT—Halt: This is an active low output signal used to indicate that the MPU has executed the HALT instruction.
- RFSH—Refresh: This is an active low signal indicating that the address bus A₆-A₀ (low-order seven bits) holds a refresh address of dynamic memory; it should be used in conjunction with MREQ to refresh memory contents.

POWER AND FREQUENCY SIGNALS

This group includes three signals as follows:

- ϕ —Clock: This pin is used to connect a single phase frequency source. The Z80 does not include a clock circuit on its chip; the circuit must be built separately.
- +5 V and GND—These pins are for a power supply and ground reference; the Z80 requires one +5 V power source.

MACHINE CYCLES AND BUS TIMINGS

3.3

The Z80 microprocessor is designed to execute 158 different instructions. 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 Z80 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 Z80 has to perform. For example, the instruction to send the contents of the accumulator to the output port 10_H is a 2-byte instruction: OUT (10H), A.

- Byte 1: OUT → This is the opcode to output data.
- Byte 2: (10H*), A → This is the operand to specify that the byte should be sent from the accumulator to Port 10_H.

But the Z80 has to perform three operations: (1) read Byte 1 from memory, (2) read Byte 2 from memory, (3) send data to port 10_H.

In the previous section, numerous Z80 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 operations called machine cycles, and these machine cycles are divided into precise *system clock periods*.

The microprocessor external communication functions can be divided into three basic 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

*A hexadecimal number in an instruction is shown as a number followed by the letter H.

shown in Table 3.1. Each instruction consists of one or more of these machine cycles, and each machine cycle is divided into T-states.

To understand various operations, we need to define three terms: instruction cycle, machine cycle, and T-state.

Instruction cycle is defined as the time required to complete the execution of an instruction. The Z80 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, accessing I/O, or acknowledging an external request. This cycle may consist of three to six T-states.

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.

In this chapter, we focus on the first three operations listed in Table 3.1—Opcode Fetch, Memory Read, and Memory Write—and examine the signals on various buses in relation to the system clock. In the next chapter, we will use these timing diagrams to interface memory with the Z80 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 and Interrupt Acknowledge will be discussed in Chapter 12.

3.31 Opcode Fetch Machine Cycle (\overline{M}_1)

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. The opcode fetch operation and its timing signals are illustrated in the example below.

TABLE 3.1
The Z80 Machine Cycles and Control Signals

Machine Cycle	\overline{M}_1	\overline{MREQ}	\overline{IORQ}	\overline{RD}	\overline{WR}
Opcode Fetch (\overline{M}_1)	0	0	1	0	1
Memory Read	1	0	1	0	1
Memory Write	1	0	1	1	0
I/O Read	1	1	0	0	1
I/O Write	1	1	0	1	0
Interrupt Acknowledge	0	1	0	1	1
Non-maskable Interrupt	0	0	1	0	1
Bus Acknowledge ($\overline{BUSAK} = 0$)	1	Z	Z	Z	Z

NOTE: Logic 0 = Active, Logic 1 = Inactive, Z = High Impedance

The accumulator of the Z80 microprocessor holds the data byte $9F_H$, and the code for instruction LD B, A (opcode) $0\ 1\ 0\ 0\ 0\ 1\ 1\ 1$ (47_H) is stored in memory location 2002_H . This is a 1-byte instruction, and when this opcode is executed, the contents of the accumulator will be copied into register B. List the sequence of events that takes place to execute this machine code and illustrate the signals on various buses in relation to the system clock.

Example 3.2

Before the Z80 can execute the opcode, it needs to fetch the code from the memory location. To fetch the opcode, the Z80 performs the following steps:

Solution

1. The Z80 places the contents of the program counter (2002_H) on the address bus, and increments the program counter to the next address, 2003_H . The program counter always points to next byte to be executed.
2. The address is decoded by the external decoding circuit and the register 2002_H is identified.
3. The Z80 sends the control signals (\overline{MREQ} and \overline{RD}) to enable the memory output buffer.
4. The contents of the memory register (opcode 47_H) are placed on the data bus and brought into the instruction decoder of the microprocessor.
5. The Z80 decodes the opcode and executes the instruction, meaning it copies the contents of the accumulator into register B.

Figure 3.5 shows how the Z80 fetches the opcode using the address and the data buses and the control signal. Figure 3.6 shows the timing of the Opcode Fetch machine cycle in relation to the system's clock. The address bus in Figure 3.6 is 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, and the crossover of the lines indicates that a new address is being placed on the address bus. The high impedance state is shown by a straight line as in the data bus (D_7-D_0). The timing details of these signals are given below.

1. Figure 3.6 shows that the Opcode Fetch cycle is completed in four clock periods or T-states. This machine cycle is also identified as the M_1 cycle.
2. At the beginning of the first clock period T_1 , the control signal $\overline{M_1}$ goes low and the contents of the program counter (2002_H) are placed on the address bus.
3. After the falling edge of T_1 , the Z80 asserts two control signals— \overline{MREQ} and \overline{RD} , both active low. The \overline{MREQ} indicates that it is a memory related operation and \overline{RD} suggests that it is a Read operation. Both signals are necessary to read from memory.
4. The internal decoder of the memory and the Chip Select circuit (not shown in Figure 3.6) decode the address and identify register 2002_H . The control signals \overline{MREQ} and \overline{RD} are used to enable the memory output buffer. The data bus, which was in high impedance state, is activated as an input bus (to the microprocessor) shortly after the leading edge of T_2 . After the falling edge of T_2 , memory places its register contents (47_H) on the data bus.

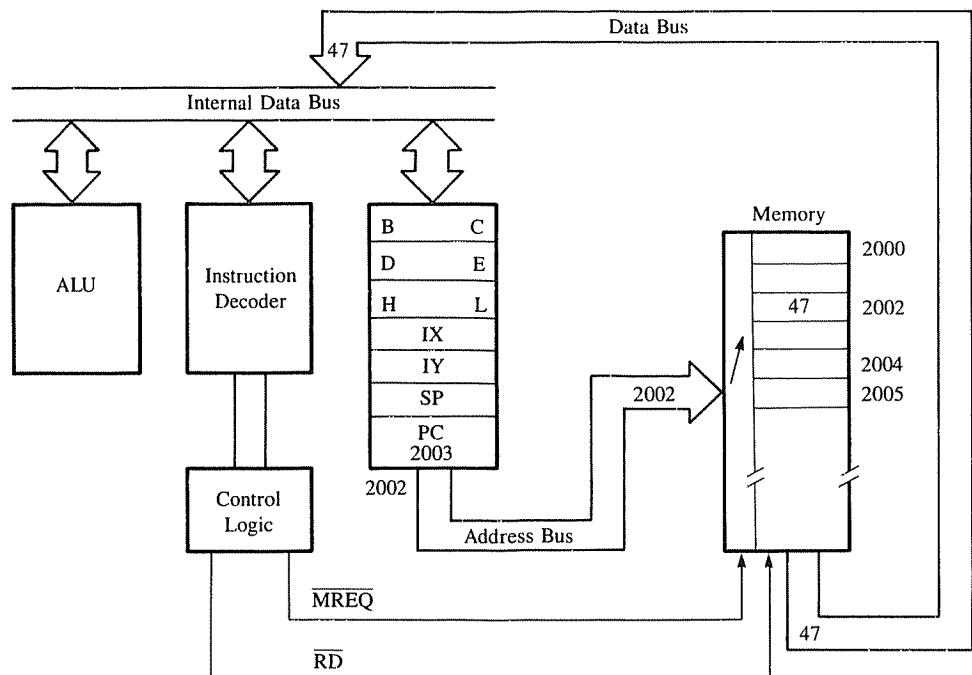


FIGURE 3.5
Z80 Memory Read Operation

5. At the leading edge of T_3 , the data on the data bus are read, and the control signals become inactive.
 6. During T_3 and T_4 , the instruction decoder in the microprocessor decodes and executes the opcode. These are internal operations and cannot be observed on the data bus.

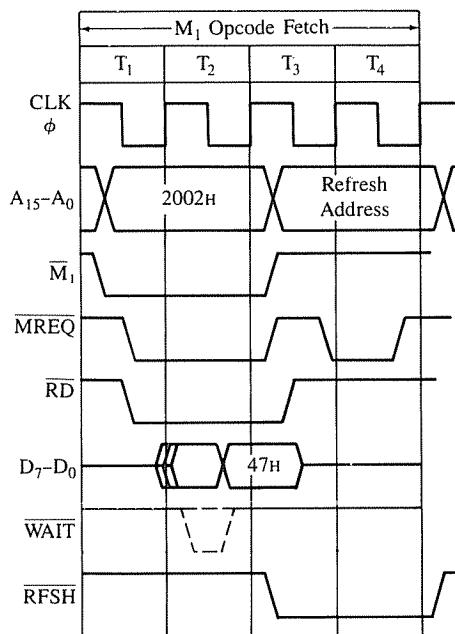
The following two steps are irrelevant to the present problem; however, they are included here as part of the M_1 cycle.

7. During T_3 and T_4 , when the Z80 is performing internal operations, the low-order address bus is used to supply a 7-bit address for refreshing dynamic memory. If the system includes dynamic memory, this operation simplifies its interfacing hardware. This aspect of the M_1 cycle will be discussed again when we illustrate interfacing of dynamic memory (Chapter 16).
 8. Figure 3.6 shows the signal called WAIT. The Z80 samples the Wait line during T_2 , and if it is forced low by an external device (such as memory or I/O), the Z80 adds Wait states (clock cycles) to extend the machine cycle and continues to add clock cycles until the Wait signal goes high again. This technique is used to interface memories with slow response time and will be discussed again in Chapter 16.

FIGURE 3.6

Z80 Opcode Fetch (M_1) and Bus Timings

SOURCE: Courtesy of Zilog Inc. (adapted).



3.32 Memory Read Machine Cycle

The second machine cycle we want to illustrate is Memory Read. As explained in the next example, this cycle is quite similar to the Opcode Fetch cycle.

Two machine codes—0 0 1 1 1 1 1 0 (3E_H) and 1 0 0 1 1 1 1 1 (9F_H)—are stored in memory locations 2000_H and 2001_H respectively, as shown below. The first machine code (3E_H) represents the opcode to load a data byte into the accumulator, and the second code (9F_H) represents the data byte to be loaded into the accumulator. Illustrate the bus timings as these machine codes are executed, and calculate the time required to execute the Opcode Fetch and the Memory Read cycles and the entire instruction cycle if the clock frequency is 4 MHz.

Example
3.3

Address	Machine Code	Instruction	Comment
2000 _H	0 0 1 1 1 1 1 0	→ 3E	LD A, 9FH ;Load 9FH in the accumulator
2001 _H	1 0 0 1 1 1 1 1	→ 9F	

This instruction consists of two bytes; the first is the opcode and the second is the data byte. The Z80 must first read these bytes from memory and thus requires at least two machine cycles. The first machine cycle is Opcode Fetch and the second machine cycle is

Solution

Memory Read, as shown in Figure 3.7. These cycles are described in the following list.

1. The first machine cycle (Opcode Fetch) is identical in bus timings with the machine cycle illustrated in Example 3.2, except for the bus contents. The address bus contains 2000_H and the data bus contains the opcode $3E_H$. When the Z80 decodes the opcode during the T_3 state, it realizes that a second byte must be read.
2. After the completion of the Opcode Fetch cycle, the Z80 places the address 2001_H on the address bus and increments the program counter to the next address, 2002_H . To differentiate the second cycle from the Opcode Fetch cycle, the M_1 signal remains inactive (high).
3. After the falling edge of T_1 of the Memory Read cycle, the control signals \overline{MREQ} and \overline{RD} are asserted. These signals along with the memory address are used to identify the register 2001_H and enable the memory chip.
4. After the leading edge of T_3 , the Z80 activates the data bus as an input bus; memory places the data byte $9F_H$ on the data bus, and the Z80 reads and stores the byte in the accumulator during T_3 .
5. After the falling edge of T_3 , both control signals become inactive (high), and at the end of T_3 , the next machine cycle begins.

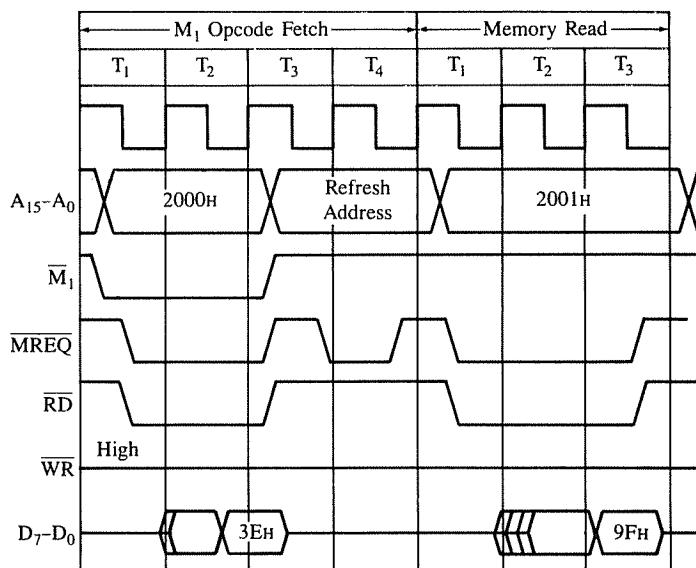


FIGURE 3.7
Memory Read Machine Cycle and Its Timings
SOURCE: Courtesy of Zilog Inc (adapted).

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

Clock Frequency $f = 4$ MHz

T-state = Clock Period ($1/f$) = $0.25 \mu\text{s}$

Execution Time for Opcode Fetch: $(4 \text{ T}) \times 0.25 = 1.0 \mu\text{s}$

Execution Time for Memory Read: $(3 \text{ T}) \times 0.25 = 0.75 \mu\text{s}$

Execution Time for Instruction: $(7 \text{ T}) \times 0.25 = 1.75 \mu\text{s}$.

3.33 Memory Write Cycle

Now we want to illustrate the third machine cycle: Memory Write. This machine cycle writes or stores data in a specified memory register as shown in the following example.

The HL register holds the address 2350_{H} , and the accumulator has the data byte $9F_{\text{H}}$. The instruction code $0111\ 0111$ (77_{H}) is stored in memory location 2003_{H} . When this code is executed, it stores the contents of the accumulator in the memory location indicated by the address in the HL register. Illustrate the bus contents and timings as this instruction is being executed.

Example
3.4

Instruction: LD (HL), A ;Copy contents of the accumulator
into memory location, the address
of which is stored in HL register.

This is a one-byte instruction with two machine cycles: Opcode Fetch and Memory Write. In the first machine cycle, the Z80 fetches the code (77_{H}), and in the second machine cycle, it copies the byte $9F_{\text{H}}$ from the accumulator into the memory location 2350_{H} . The timings of these machine cycles are shown in Figure 3.8 and explained below.

Solution

1. In the Opcode Fetch machine cycle, the Z80 places the address 2003_{H} on the address bus and gets the code 77_{H} by using the control signals MREQ and RD as in the previous examples. The program counter is also incremented to the next address, 2004_{H} .
2. During the T_3 and T_4 states, the Z80 decodes the machine code 77_{H} and prepares for the memory write operation.
3. At the beginning of the next machine cycle (Memory Write), it places the contents (2350_{H}) of the HL register on the address bus. At the falling edge of T_1 , $\overline{\text{MREQ}}$ goes low and the data byte $9F_{\text{H}}$ from the accumulator is placed on the data bus.
4. After allowing one T-state (after MREQ) to stabilize the address, the Z80 asserts the control signal Write ($\overline{\text{WR}}$), which is used to write the data byte at the address shown on the address bus.
5. After the falling edge of T_3 , both control signals become inactive, and one-half T-state later, the data bus goes into high impedance state.

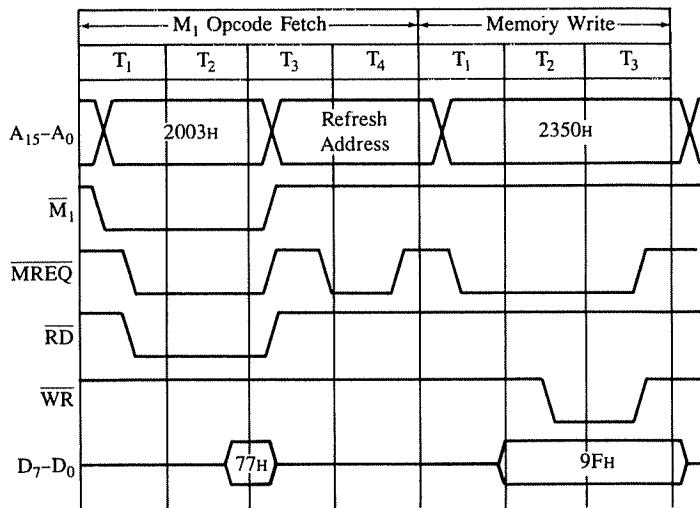


FIGURE 3.8
 Memory Write Machine Cycle and Its Timings
 SOURCE: Courtesy of Zilog Inc. (adapted)

3.34 Review of Important Concepts

1. In each instruction cycle, the first operation is always Opcode Fetch, and it is indicated by the active low \overline{M}_1 signal. This cycle can be four to six T-states in duration.
2. The Memory Read cycle is in many ways similar to the Opcode Fetch cycle. Both use the same control signals (MREQ and RD) and read contents from memory. However, the Opcode Fetch reads opcodes and the Memory Read reads 8-bit data or addresses; the two machine cycles are differentiated by the \overline{M}_1 signal.
3. The control signals MREQ and RD, are both necessary to read from memory.
4. In the Memory Write cycle, the Z80 writes (stores) data in memory using the control signals MREQ and WR.
5. In the Memory Read cycle, the Z80 asserts the MREQ and RD signals to enable memory, and then the addressed memory places data on the data bus; on the other hand, in the Memory Write cycle, the Z80 asserts the MREQ, places data byte on the data bus, and then asserts the WR signal to write into the addressed memory.
6. Generally, the Memory Read and Write cycles consist of three T-states; however, they can take four T-states in some instructions. The Memory Read and Write cycles will not be asserted simultaneously; the microprocessor cannot read and write at the same time.

3.35 Generating Control Signals

After examining the concepts summarized at the end of the previous section, we may need to generate additional control signals.

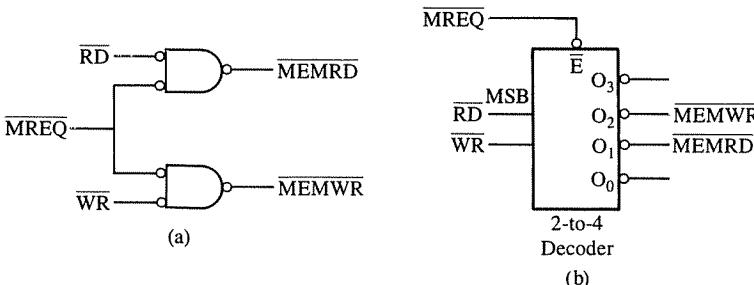


FIGURE 3.9
Generating Memory Control Signals

1. To read from memory, the MREQ and the RD signals are necessary, and to read from an input device, the IORQ and the RD are necessary; all these signals are active low. As a design practice, the MREQ is generally combined with a decoded address (discussed in Chapter 4) and RD is connected directly to the memory chip. However, control signals RD and WR can also be combined with MREQ and IORQ to generate additional signals. We can generate active low Memory Read (MEMRD) signal either by ANDing these signals in a negative NAND gate as shown in Figure 3.9(a) or by using a 2-to-4 decoder as shown in Figure 3.9(b). The decoder is enabled by the MREQ and has RD and WR signals as input. Both inputs cannot be active at the same time; when one is low, the other will remain high. When RD is active low, the input is 0 1, and the output O₁ goes active as MEMRD.
2. To write into memory, the MREQ and the WR signals are necessary, and to write a data byte to an output device the IORQ and WR signals are necessary; all these signals are active low. If necessary, we can generate active low Memory Write (MEMWR) signal by ANDing MREQ and WR signals in a negative NAND gate as shown in Figure 3.9(a) or by using the decoder as shown in Figure 3.9(b). Similarly, IORD (I/O Read) and IOWR (I/O Write) signals can be generated; this is discussed in Chapter 5.

SOME PUZZLING QUESTIONS AND THEIR ANSWERS

3.4

After reading the previous sections, the reader may have many unanswered questions. One of the primary reasons for this predicament is that the microprocessor is a programmable and complex device. It interacts with external devices such as memory and I/Os, and some questions cannot be answered until we discuss these other devices. Similarly, some questions will remain unanswered until we start using instructions and writing programs. However, there are some questions which we should answer immediately.

1. *How does the Z80 microprocessor know where to begin after the power is turned on?*

Most microcomputer systems have built-in power-on *reset circuits*, meaning that when the power is turned on, the microprocessor is reset and its program counter is cleared to the address 0000_H . The address 0000_H is placed on the address bus, and the instruction stored at that location determines what happens next.

2. How does the Z80 know what operation to perform first (Memory Read/Write or I/O Read/Write)?

The first operation is always an Opcode Fetch.

3. How does the microprocessor differentiate between an opcode and a data byte?

When the first opcode is fetched and decoded in the instruction register, the microprocessor recognizes the number of bytes that must be read from memory for the complete instruction. The instructions can range from 1-byte to 4-byte in length. Figure 3.7, for example, contains a 2-byte instruction (3E and Data), and the second byte is always considered Data. If that second byte is omitted by mistake, the Z80 will interpret whatever is in that memory location as Data. The byte after the Data will be treated as the next instruction. The microprocessor is a sequential machine; it goes from one memory location to the next unless instructed to do otherwise.

4. What is the use of the \overline{M}_1 signal? It looks as if it will not be connected to any device.

This signal serves two purposes: (1) it differentiates the Opcode Fetch cycle from other operations, and (2) it can be used to generate the Interrupt Acknowledge signal.

5. If flags are individual flip-flops, can they be observed on an oscilloscope?

No, they cannot be observed on an oscilloscope; these flip-flops are internal and not connected to any of the external pins. However, they can be examined by storing them on the *stack memory* (see Chapter 10).

6. Is the number of T-states required for a given machine cycle constant?

No. But most Opcode Fetch machine cycles require four T-states, and Memory Read/Write and I/O Read/Write machine cycles, generally, take three or four T-states. However, there are some exceptions.

7. How does one recognize the machine cycles in a given instruction?

The number of machine cycles and the T-states required for those machine cycles are listed in the instruction set. There is a repetitive pattern, and one can use the following guidelines.

- The number of machine cycles in an instruction indicates how many times the microprocessor must access memory or I/O.
- The first machine cycle in an instruction is always Opcode Fetch.
- The microprocessor must read all the bytes (codes) from memory before it can execute an instruction.

For example, a 3-byte instruction requires at least three machine cycles. The unconditional Jump instruction is a 3-byte instruction with 10 (4, 3, 3) T-states; it consists of one opcode and a 16-bit address of the jump location. Therefore, by examining the number of T-states, we can easily classify the machine cycles of the Jump instruction as one Opcode Fetch and two Memory Read.

Another example is ADD A, 32H (add a byte 32H to the contents of the accumulator). This is a 2-byte instruction with 7 (4, 3) T-states. By examining the number of bytes and the number of T-states, we can conclude that it must have two machine cycles—the first is Opcode Fetch and the second is Memory Read. The addition is performed inside the processor, and it does not need any additional information from memory or I/O.

8. How does one recognize machine cycles in an instruction when the number of bytes is not the same as the number of machine cycles?

One has to examine the number of bytes, T-states, and the operation being performed. For example, the instruction LD (2050H), A has three bytes and 13 (4, 3, 3, 3) T-states; it copies the contents of the accumulator into the memory location 2050H. The processor must read the entire instruction first; therefore, the first must be Opcode Fetch, followed by two Memory Read cycles. This accounts for ten T-states. In the remaining three states, the processor must write (copy) the contents of the accumulator into the memory location 2050H; therefore, it must be the Memory Write cycle.

ARCHITECTURE OF CONTEMPORARY 8-BIT MICROPROCESSORS

3.5

The primary reasons to discuss other 8-bit contemporary microprocessors are to examine how the MPU model developed in the last chapter matches with various microprocessors and to confirm that the underlying basic concepts remain similar even though specific details may vary from one chip to another. At present, a large number of 8-bit general-purpose microprocessors are available in the market. We will focus on three: the Intel 8085, the National Semiconductor NSC 800, and the Motorola 6800. These microprocessors are selected to illustrate various strategies used in designing the microprocessor. The recent trend in 8-bit microprocessors can be illustrated by so-called 8-bit super chips, such as the Hitachi HD64180 and the Zilog Z280. These are discussed in Chapter 18.

3.51 The Intel 8085

The Intel 8085 and its predecessor the 8080 are the most widely used 8-bit microprocessors. The 8080 MPU is composed of three chips—the 8080 microprocessor, the clock generator, and the system driver—and it needs three power supplies (+ 5 V, - 5 V, + 12 V). The 8085 is an upgraded version of the 8080; it operates with one + 5 V power supply, and one chip replaces the 8080's three chips. The 8085 is upward software compatible with the 8080; it has only two more instructions than the 8080. The programming models of both microprocessors are identical; however, the 8085 hardware model differs

significantly not only from the 8080 but also from other contemporary 8-bit microprocessors. The 8085 has a multiplexed bus (8 lines), which is used as both the 8-bit data bus and the low-order address bus. This feature allows Intel to provide additional interrupt lines.

THE 8085 HARDWARE MODEL

Figure 3.10 represents the hardware model with the logic pinout of the 8085. The six categories of the signals are address bus, data bus, control (and status) signals, external requests, request acknowledge, and power and frequency signals. In addition, the 8085 has two signals for serial I/O.

The 8085 has a 16-bit address bus; however, its low-order address bus is multiplexed with the data bus. These eight lines are time-shared by two functions; in the earlier part of a machine cycle, they are used for a low-order address, and in the later part for data. To

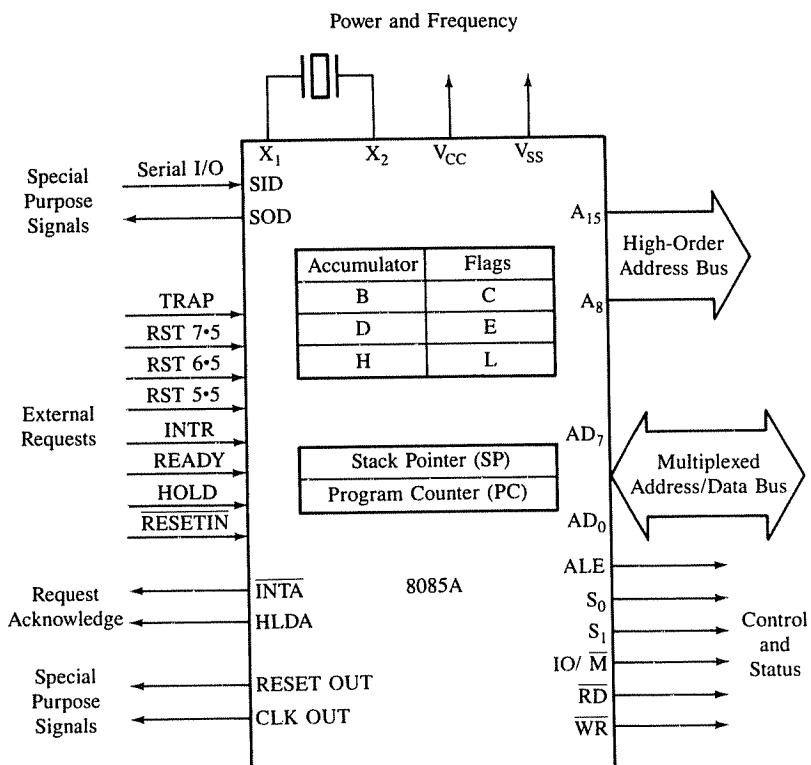


FIGURE 3.10
The Intel 8085 Microprocessor Model
SOURCE: Courtesy of Intel Corporation

interface this chip with memory (without any special features), these lines need to be demultiplexed (separated). The 8085 has a signal called ALE (Address Latch Enable), which can be used to demultiplex the bus, as shown in Figure 3.11. The ALE is asserted at the beginning of each machine cycle, when the bus has an address. Figure 3.11 shows that the ALE is used to latch the address, thus creating a separate low-order bus A₇-A₀. The Z80 does not need this signal because it has separate lines for the data and the address buses.

The 8085 has two status signals S₀ and S₁ to identify various machine cycles, and an IO/M signal to differentiate between an I/O operation and a memory operation. In contrast, the Z80 identifies the Opcode Fetch cycle by asserting M₁ and has two separate signals (MREQ and IORQ) to identify memory and I/O operations. In the 8085, the control

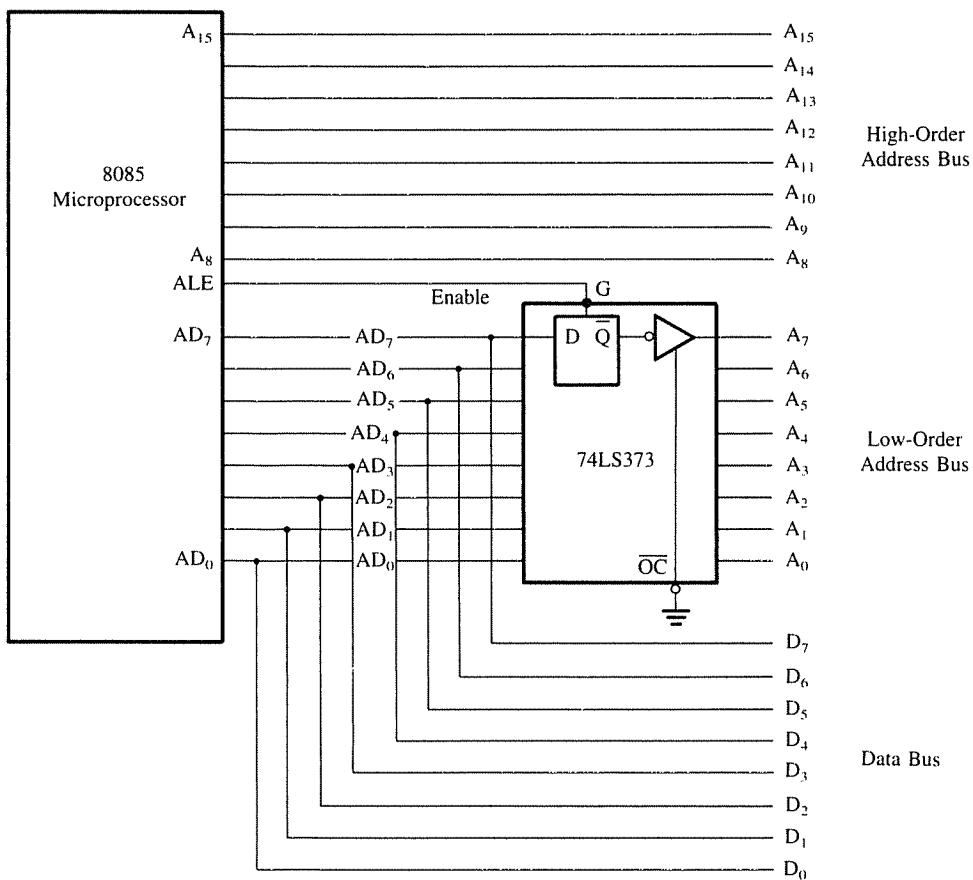


FIGURE 3.11
Demultiplexing the 8085 Bus

signals Memory Read/Write and I/O Read/Write are generated by ANDing IO/\bar{M} and control signals RD and WR.

Figure 3.10 shows that the 8085 provides five interrupt lines as external requests, out of which the TRAP is equivalent to the Z80 non-maskable interrupt. The Z80 provides various additional interrupt modes through software.

THE 8085 SOFTWARE MODEL

Figure 3.10 also shows the software model of the 8085. It includes one accumulator, a flag register, a general-purpose register array, and two 16-bit registers as memory pointers (program counter and stack pointer). This model matches very well with the requirements of the microprocessor as a processing unit (Figure 2.2). The Z80 includes all the 8085 registers in addition to an alternate set of registers, index registers, and special-purpose registers.

3.52 The National Semiconductor NSC800

The NSC800 is an 8-bit microprocessor manufactured by National Semiconductor. It is a low-power CMOS device that combines features of the 8085 and the Z80. Because its power consumption is 5 percent of that of NMOS devices, it is ideally suited for low-power or battery-operated applications.

The NSC800 has a bus structure similar to that of the 8085: a multiplexed bus with the status signals S_0 , S_1 , and IO/\bar{M} . It has a powerful interrupt scheme that combines the 8085 signals and the Z80 interrupt modes. Its software model, instruction set, and mnemonics are identical with those of the Z80.

In summary, the NSC800 combines the software capability of the Z80 with the bus structure of the 8085; its hardware and software models match with the generalized model we developed in the previous chapter.

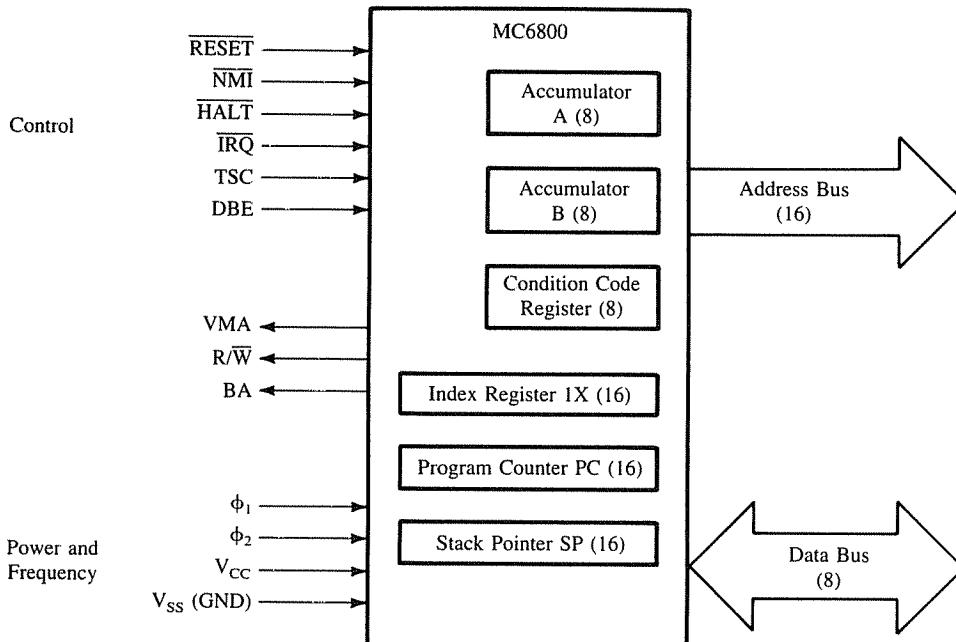
3.53 The Motorola MC6800

The MC6800 was developed at about the same time as the Intel 8080. The hardware model of this processor is similar to any other processor we have discussed, but it has a different internal architecture.

Figure 3.12 represents the 6800's architecture. It has 16 address lines, 8 data lines and fewer control (and status) signals than the Z80. The fewer control signals result from not having peripheral-mapped I/O; all I/Os are interfaced as memory-mapped I/Os. Therefore, the control signals in this processor need not differentiate between memory and I/O operations.

The other significant difference is in its internal architecture; it has two accumulators, one flag register shown as the Condition Code Register, but no general-purpose registers. This processor uses external memory for storing interim calculations and data bytes; it makes extensive use of memory referencing in its operations. The 6800 has simple timing and control signals; the clock period is the same as the machine cycle.

The 6809 is the latest improved version of the 6800 family; however, its machine code is not compatible with that of the 6800. Its internal architecture is similar to that of the

**FIGURE 3.12**

The Motorola 6800 Microprocessor Model

SOURCE: Courtesy of Motorola, Inc.

6800, except it has an additional stack pointer, an additional index register, and a register to be used for referencing memory. The basic design philosophy is the same as that of the 6800, but it has eliminated some limitations of the 6800.

3.54 Review of 8-bit Microprocessors

In the last section, we examined the architectures of three microprocessors and occasionally compared them with the Z80. Now we can easily conclude that the architectures of various 8-bit microprocessors have similar patterns and can be represented by the hardware and the software models developed in the last chapter. We can classify these processors into two categories: one group, including the Z80, the 8085, NSC800, is register-oriented; the group including the 6800 and the 6809 is memory-reference-oriented.

SUMMARY

- The Z80 microprocessor has six general-purpose 8-bit registers (B, C, D, E, H, and L) as a primary set. In addition, it includes the alternate set of these registers,

all of which can be used to exchange information with the primary set. The registers B and C, D and E, and H and L can be combined to perform some 16-bit operations.

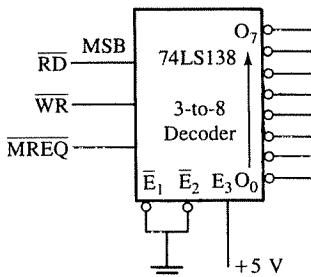
- The ALU section of the Z80 includes accumulator A and the flag register to indicate six different data conditions. It also includes the alternate accumulator A' and flag register F', which can be used to exchange information with A and F respectively.
- Four flags—Sign, Zero, Carry, and Parity/Overflow—can be used for decision making and tested in conjunction with Jump, Call and Return instructions. Two flags—Half-Carry and Add/Subtract—are used internally for BCD operations and not available for the programmer.
- The Z80 has four 16-bit registers—IX, IY, SP, and PC—used as memory pointers. Two index registers IX and IY can be used to point to any memory location, and an address and the direction (backward or forward) can be specified with a displacement byte. The stack pointer (SP) is used to specify memory locations in a defined R/W memory segment called the stack. The program counter (PC) is used to sequence the program execution; it points to the next memory address from which the machine code is to be fetched.
- The Z80 includes two 8-bit special-purpose registers: Interrupt Vector (I) and Memory Refresh (R). The I register provides the high-order 8 bits of a 16-bit address to which the program is to be directed after an interrupt. The R register is a 7-bit counter and supplies an address for refreshing memory cells of a dynamic memory.
- The Z80 signals can be classified into six groups: address bus, data bus, control signals, external requests, request acknowledge, and power and frequency signals (see Section 3.2 for definitions of these signals).
- The Z80 is designed to execute 158 instructions, and each instruction can be divided into a few basic operations called machine cycles.
- The frequently used machine cycles are Opcode Fetch, Memory Read and Write, and I/O Read and Write.
- The Opcode Fetch and Memory Read are operationally similar; the Z80 reads from memory in both machine cycles. However, the Z80 reads opcode during the Opcode Fetch cycle, and it reads 8-bit data during the Memory Read cycle. In the Memory Write cycle, the Z80 writes data into memory.
- The memory operations are differentiated from I/O operations by two control signals: MREQ and IORQ. The signal MREQ is combined with RD and WR signals to generate MEMRD and MEMWR control signals.
- The Z80 performs three basic steps in any of these machine cycles: It places an address on the address bus, sends appropriate control signals, and transfers data via the data bus.
- The contemporary 8-bit microprocessors can be classified into two categories: One group is register-oriented, and the other is memory-reference-oriented.

ASSIGNMENTS

1. How is the accumulator different from the 8-bit general-purpose registers of the Z80 microprocessor?
2. Explain the function of the alternate registers.
3. What is a flag and what is its function?
4. If the Z80 adds 87_H and 79_H , specify the contents of the accumulator and the status of the S, Z, and CY flags.
5. If the Z80 is an 8-bit microprocessor, why are the program counter and the stack pointer 16-bit registers?
6. If the Z80 has fetched the machine code located at the memory location $205F_H$, specify the contents of the program counter.
7. The index register IX holds the address 2058_H . Specify the value of the displacement byte needed to make the effective address 2097_H .
8. The index register IY holds the address 2070_H . Specify the value of the displacement byte in 2's complement needed to make the effective address 2050_H .
9. The MOS Technology 6501 microprocessor chip has 13 address lines. Specify the memory registers it is capable of addressing.
10. If the Intel 8086 microprocessor has 20 address lines, what is its capacity of memory addressing?
11. If the clock frequency is 4 MHz, how much time is required to execute an instruction of 21 T-states?
12. The instruction LD IX, (2050_H) loads 2050_H into the index register. Specify the number of bytes, machine cycles, and T-states of this instruction by checking the instruction set. Calculate the time required to execute the instruction if the system clock frequency is 6 MHz.
13. List the sequence of events that occurs when the Z80 reads from memory.
14. In the Opcode Fetch cycle, what are the control signals required to enable the memory buffer?
15. When is the data byte placed on the data bus in the Memory Write cycle?
16. The memory location 2065_H holds the opcode F9_H. If the Z80 begins the Opcode Fetch cycle by placing the address 2065_H on the address bus, specify the contents of the data bus after the falling edge of the T₂ state.
17. The instruction LD B, (HL) copies the contents of the memory location specified by the 16-bit contents in the HL register. It is a 1-byte instruction with two machine cycles. Identify the second machine cycle and its control signals.
18. Figure 3.13 shows a 3-to-8 decoder with MREQ, RD, and WR as input signals. Identify the control signals that can be generated at the outputs of the decoder.
19. Figure 3.14 shows the timings of three machine cycles. Identify the types of operations.

FIGURE 3.13

Generating Control Signals Using a 3-to-8 Decoder. Assignment 18



20. Do the three machine cycles in Figure 3.14 represent a complete instruction? Explain.
21. Examine the machine cycle M_b in Figure 3.14 and specify the memory being accessed and its contents.
22. Does the byte on the data bus in the machine cycle M_a in Figure 3.14 represent an opcode?
23. Explain what is being done in machine cycle M_c (Figure 3.14).
24. Identify the machine cycles M_a and M_b in Figure 3.15.
25. Identify the machine cycles in the following instructions:

- | | |
|---------------|-----------------------------------------------------------------------|
| SUB B | : 1-byte, 4 T-states |
| | : Subtract the contents of register B from the accumulator |
| AND 47H | : 2-byte, 7 (4, 3) T-states |
| | : Logically AND 47H with the contents of the accumulator |
| LD A, (2050H) | : 3-byte, 13 (4, 3, 3) T-states |
| | : copy the contents of the memory location 2050H into the accumulator |
| PUSH BC | : 1-byte, 10 (4, 3, 3) T-states |
| | : copy the contents of BC register into two stack memory locations |

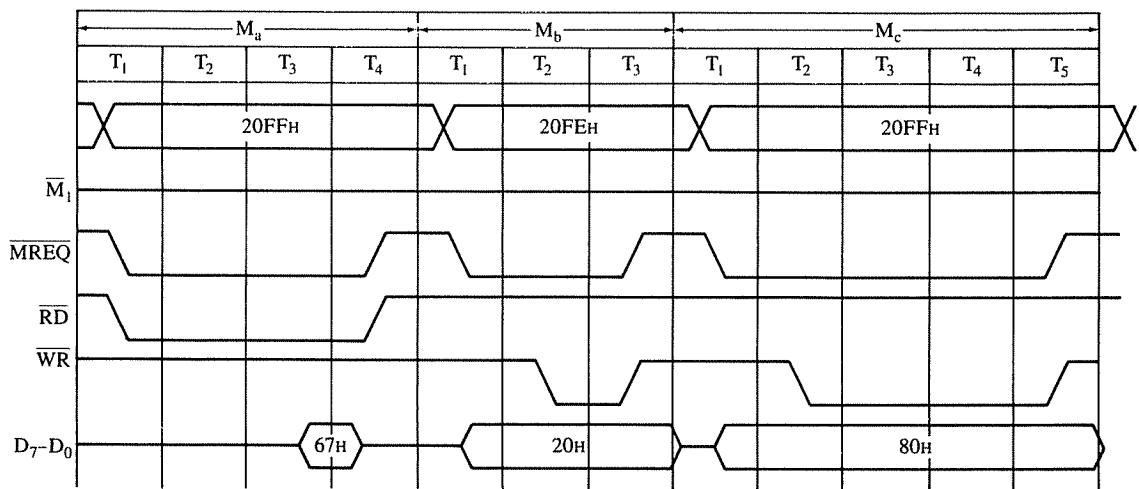


FIGURE 3.14
Machine Cycles for Assignments 19–23

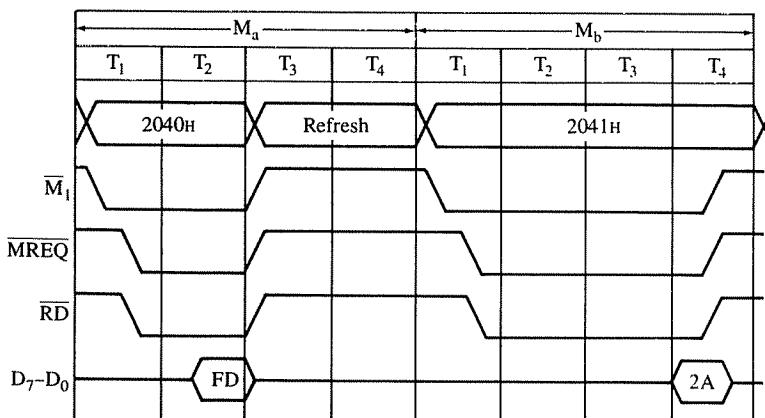


FIGURE 3.15
Machine Cycles for Assignment 24

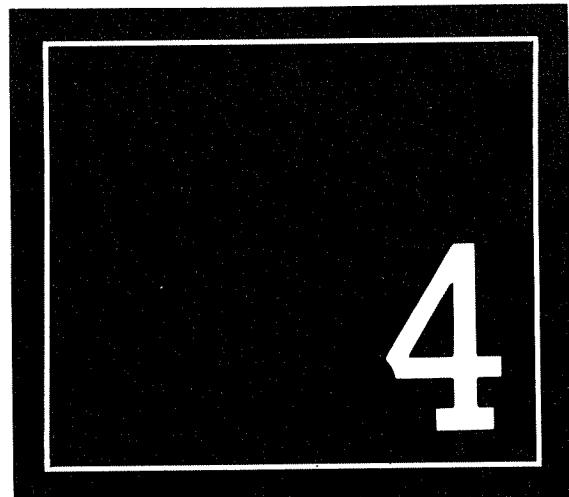
Memory Interfacing

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. We will examine memory structure and requirements to read from it and write into it. We then compare those requirements with those of the Z80 Memory Read and Write machine cycles. From that comparison, we will derive the basic steps necessary to interface memory.

This chapter illustrates two examples of interfacing memory chips, one EPROM and the other static R/W memory. The discussion includes analyses of the following: **decoding circuits, memory maps**, the concept of **foldback memory**, and **linear decoding** versus **absolute decoding**. Finally, an example of memory design is illustrated to synthesize the interfacing concepts.

OBJECTIVES

- List the requirements to read from memory.
- List the steps initiated by the Z80 to read from and write into memory.
- List the steps required to interface a memory chip with the Z80.
- Analyze given EPROM and static R/W memory interfacing circuits and specify their memory maps.
- Explain the terms absolute decoding, linear decoding, and foldback memory.
- Design a circuit to interface EPROM and R/W memory with the Z80 for given memory maps.



4.1 INTERFACING MEMORY

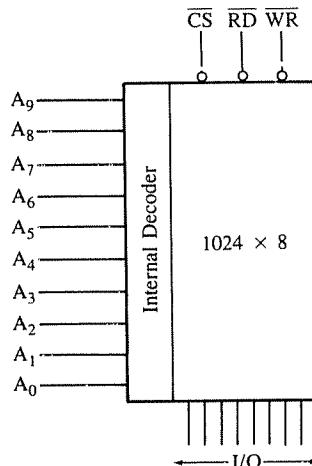
While executing a program, the microprocessor needs to access memory frequently to read instruction codes and data stored in memory, and the *interfacing circuit* enables that access. Memory has certain signal requirements for writing into and reading from its registers. Similarly, the microprocessor initiates a set of signals when it wants to communicate with memory. The interfacing process involves designing a circuit that will match the memory requirements with the microprocessor signals. In the following section, we examine memory structure and its requirements and also the Z80 Memory Read and Write machine cycles. Then we derive the basic steps necessary to interface memory with the Z80.

4.1.1 Memory Structure and Its Requirements

Read/Write Memory (R/WM) is a group of registers to store binary information. Figure 4.1 shows a typical R/W memory chip; it has 1024 registers, each of which can store eight bits indicated by eight I/O lines. The chip has ten address lines A_9 – 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 4.1 also shows the internal decoder to decode the address lines. We may recall from Chapter 2 that to read from or write into one of the memory registers certain requirements have to be met. They are as follows:

1. An address should be placed on the address lines. The low-order address lines are decoded by the internal decoder of the memory chip, and the addressed register is identified.

FIGURE 4.1
Logic Diagram: A Typical 1K
Memory Chip



2. The high-order address should be decoded to generate a Chip Select signal, and the memory chip is selected by asserting the Chip Select \overline{CS} low.
3. To read from the addressed register, the \overline{RD} should be asserted low to enable the output buffer, and then the data byte from the register will be placed on the I/O lines.
4. To write into the addressed register, the \overline{WR} should be asserted low to enable the input buffer, and then data bits from the data lines are stored into the register.

To interface this memory with the Z80 microprocessor, we need to examine the signals the microprocessor asserts when it attempts to communicate with memory.

4.12 How does the Z80 Read from or Write into Memory?

In Chapter 3, we showed the timing diagrams and the Z80 bus contents when an opcode or a data byte is fetched from memory. To read from memory, the Z80 performs the following steps, as shown in Figure 4.2(a):

1. places a 16-bit address on its address bus (shown as high- and low-order addresses).
2. asserts the \overline{MREQ} to indicate that the address bus holds a valid address.
3. asserts the \overline{RD} signal low to indicate that it wants to read.

To write into memory, the Z80 performs the following steps, as shown in Figure 4.3:

1. places a 16-bit address on the address bus.
2. asserts \overline{MREQ} and places data on the data bus.
3. asserts \overline{WR} signal.

To understand and design an interface circuit, we need to match the memory requirements with the Z80 read/write operations.

4.13 Basic Concepts in Memory Interfacing

The primary function of memory interfacing is to allow the microprocessor to read from and write into a given register of a memory chip. To perform these operations, the microprocessor should

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

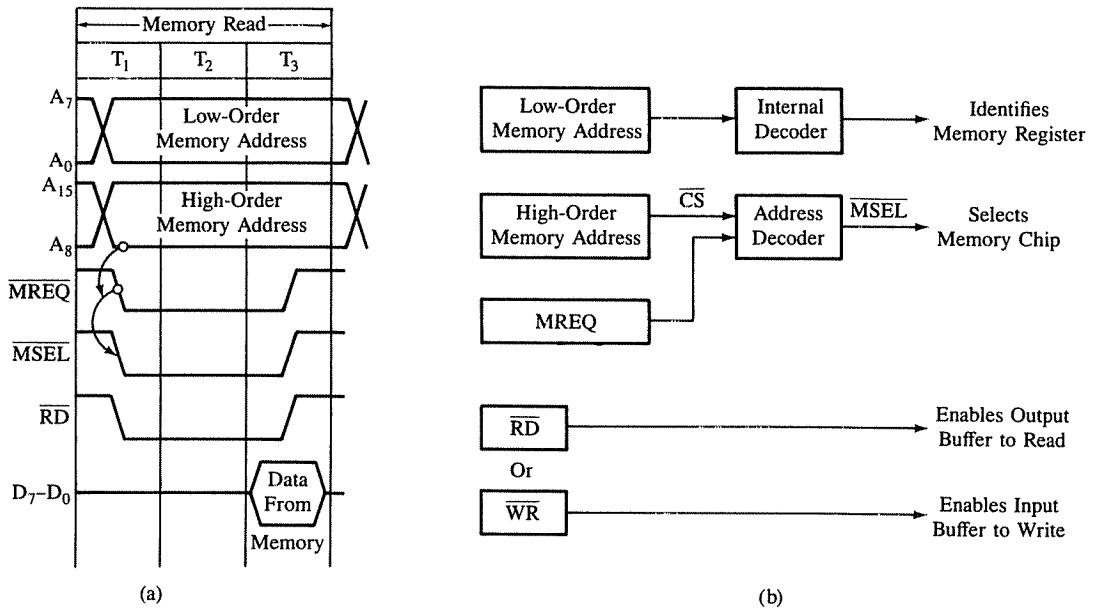


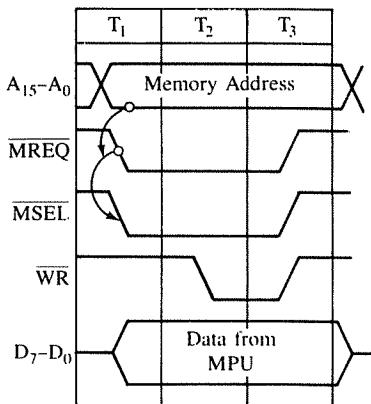
FIGURE 4.2

(a) Memory Read Timing Diagram (b) Block Diagram: Address Decoding and Memory Read/Write Operations

SOURCE: Courtesy of Zilog, Inc. (adapted)

FIGURE 4.3

Writing into Memory Register



Let us examine the timing diagram of the Memory Read operation—Figure 4.2(a)—in order to understand how the Z80 can read from memory. In Figure 4.2(a), the address bus is divided into two segments, low-order and high-order, to explain the decoding concepts.

1. The Z80 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 4.1, only ten address lines are required to identify 1,024 registers. Therefore, we can connect the low-order address lines A_9 – A_0 of the Z80 address bus to the memory chip. The internal decoder of the memory chip will identify and select the register, as shown in Figure 4.2(b).

2. The remaining Z80 address lines (A_{15} – A_{10}) should be decoded to generate a Chip-Select (CS) signal unique to that combination of address logic.
3. The Z80 provides two signals: $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$. The $\overline{\text{MREQ}}$ can be combined with the decoded address pulse ($\overline{\text{CS}}$) to generate a Memory Select (MSEL) to select the memory chip.
4. The microprocessor asserts the control signal $\overline{\text{RD}}$, enables the output buffer of memory, and reads the data byte. Figure 4.2(a) also shows that memory must place the data byte on the data bus at the beginning of T_3 .

To write into a register, the microprocessor performs similar steps. Figure 4.3 shows the Memory Write cycle. In the Write operation, the Z80 places the address and data, and asserts the $\overline{\text{MREQ}}$ signal. After allowing sufficient time for data to become stable, it asserts the Write (WR) signal. The WR signal enables the input buffer of the memory chip and stores the byte in the selected register.

An alternative to generating the MSEL signal (Step 3 in Memory Read) to select the memory chip is to generate the control signals MEMRD and MEMWR by combining the MREQ, RD, and WR as shown in Figure 4.4(a). The MEMRD can be used to enable the output buffer to read from memory; the MEMWR can be used to enable the input buffer to write into memory, and the decoded address pulse ($\overline{\text{CS}}$) can be used to select the chip as shown in Figure 4.4(b).

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

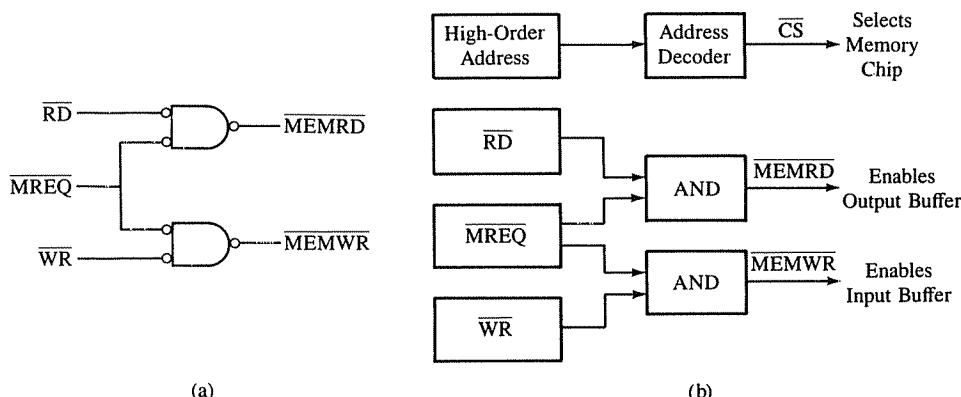


FIGURE 4.4

(a) Generating Control Signals (b) Block Diagram: Alternative Approach to Memory Read/Write Operations

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 (4.14).
3. Generate the signal Memory Select (MSEL) by combining the decoded address pulse \bar{CS} and the MREQ, and use the MSEL to select the memory chip.
4. Connect the Z80 \bar{RD} and \bar{WR} control signals to the \bar{RD} and \bar{WR} memory signals to enable memory buffers.
5. An alternative procedure is to generate control signals $\overline{\text{MEMRD}}$ and $\overline{\text{MEMWR}}$ by combining \bar{RD} and \bar{WR} signals with the $\overline{\text{MREQ}}$ and to use them to enable appropriate buffers. The decoded address pulse (\bar{CS}) is used to select the memory chip.

4.14 Address Decoding

The process of **address decoding** should result in identifying a register with a given address; we should be able to generate a unique pulse for that address. For example, in Figure 4.5(a), the output of the NAND gate goes low (active) only when the address on the address lines is $F7H$; no other address can cause the output of the gate to go low. This process is called decoding the address. We can also use a decoder for address decoding, as discussed below, or a PROM (Programmable Read-Only-Memory), as discussed in Chapter 16.

Figure 4.5(b) shows a 3-to-8 decoder and a 4-input NAND gate. The decoder has three enable lines—one active high and two active low. The enable line \bar{E}_1 is connected to address line A_3 , and \bar{E}_2 is connected to address lines A_4 – A_7 through the NAND gate. Address lines A_2 , A_1 , and A_0 are inputs to the decoder, and the enable line E_3 is tied high and is not being used here for decoding.

In this decoder circuit, three input lines can have eight different logic combinations

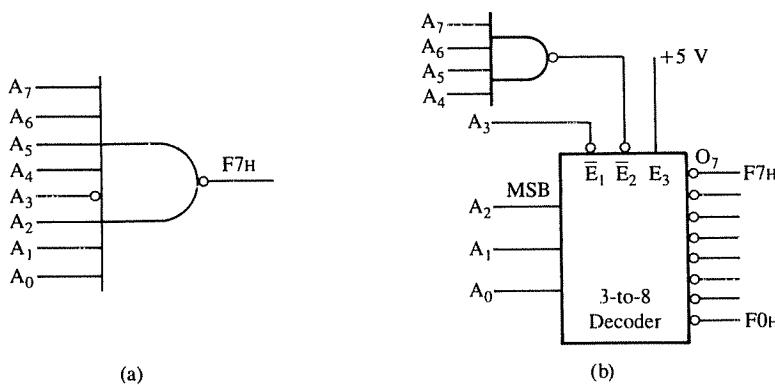


FIGURE 4.5
Address Decoding

from 000 to 111; each input combination can be identified by the corresponding output line if enable lines are active. For example, if the input is 0 0 0, O_0 goes low (others remain high), and if the input is 1 1 0, O_6 goes low. To activate the enable line \bar{E}_1 , A_3 should be low, and to activate \bar{E}_2 , address lines A_7 – A_4 should be high causing the output of the NAND gate to go low. If the input to the decoder is 1 1 1, the output line O_7 of the decoder will go low, thus decoding the address $F7_H$.

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	$= F7_H$
1	1	1	1	0	1	1	1	
↓					↓			
Enable				Input				

This 3-to-8 decoder can identify or decode eight addresses from $F0_H$ to $F7_H$ as shown in Figure 4.5(b). We will use this address decoding scheme for interfacing memory chips in the following illustrations (Sections 4.2 and 4.3).

ILLUSTRATIVE EXAMPLE 1: INTERFACING 2732 EPROM

4.2

In this section we illustrate memory interfacing with the Z80 microprocessor by using an actual chip: 2732 EPROM (Erasable Programmable Read-Only Memory). This is a memory chip commonly used in industry to develop microprocessor-based products. In this illustration, we assume that the chip has been already programmed—that is, the binary patterns representing Z80 instructions are stored in it—and we only read from it. We focus only on the interfacing concepts, interfacing logic circuit, and memory maps.

4.21 2732 EPROM

This is a 4K (4096×8) memory chip with eight data lines housed in a 24-pin package; Figure 4.6 shows the logic pinout and the pin configuration. It has twelve address lines A_{11} – A_0 to identify 4096 registers, one Chip Select signal shown as Chip Enable (\bar{CE}), and one Output Enable (\bar{OE}) signal to enable the output buffer. It operates from a single +5 V power supply in the Read mode and requires a +25 V pulse V_{pp} to program it. The signals OE and V_{pp} are multiplexed at pin 20; in the Read mode, pin 20 is used as OE and in the programming mode, it is used as V_{pp} .

The chip has a quartz window, and the information stored in this memory can be erased by exposing the window to ultraviolet light for 15 to 20 minutes. To avoid accidental erasures from direct sunlight or fluorescent lights, the window should be covered with an opaque label. Once it is erased, the chip can be used again to store a new program. The programming is done by using a circuit called EPROM programmer, which can store bits in memory registers by providing a 25 V pulse to V_{pp} .

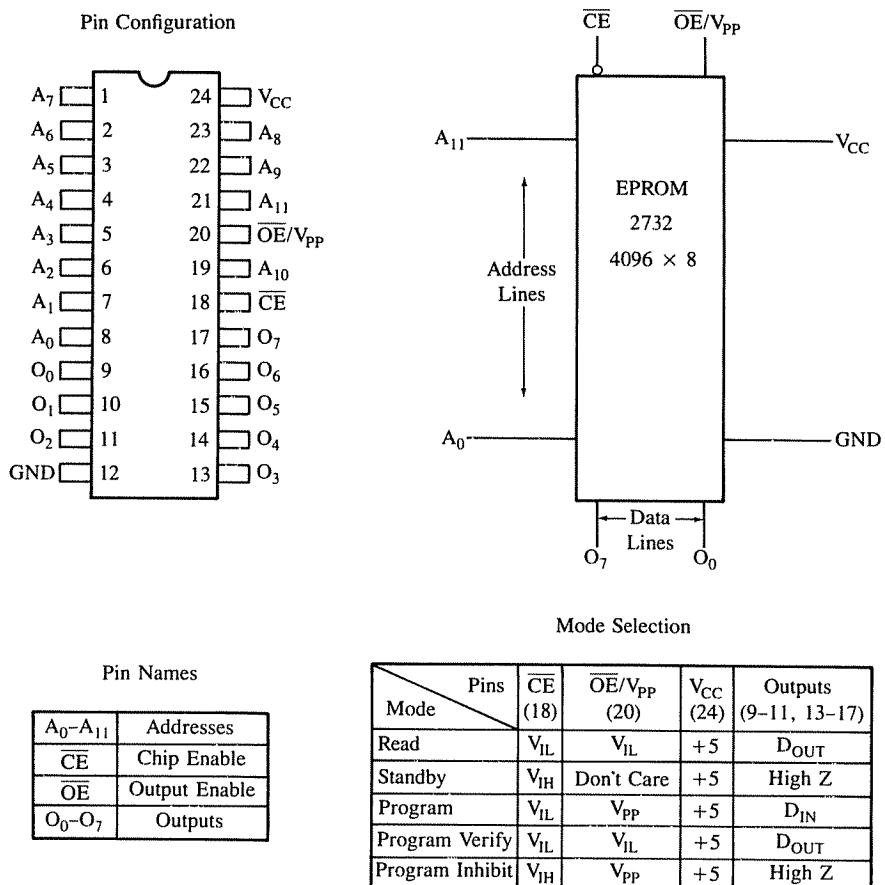


FIGURE 4.6

2732 EPROM: Pin Configuration and Logic Symbol

SOURCE: Courtesy of Intel Corporation.

4.22 Interfacing Circuit

Figure 4.7 shows a complete schematic of interfacing the 2732 with the Z80 microprocessor. We will describe this circuit in terms of the four steps required for interfacing as listed in the previous section.

Step 1: Connect the necessary address lines to the memory chip.

Figure 4.7 shows that the address lines A₁₁–A₀ are connected to the memory chip to identify 4,096 registers.

Step 2: Decode the remaining address lines and combine the \overline{MREQ} with the decoded & pulse to generate the Memory Select (MSEL) pulse.

Step 3:

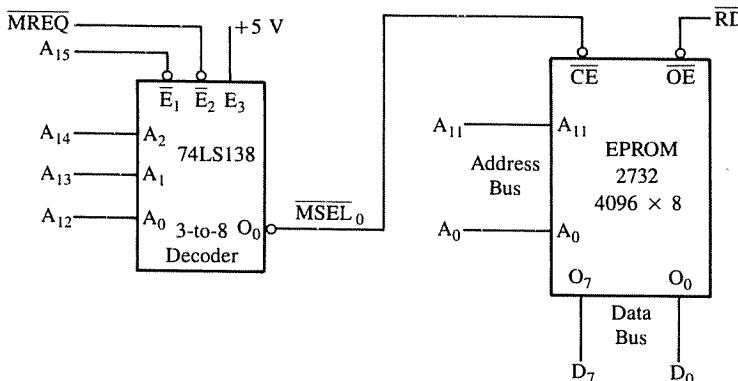


FIGURE 4.7
Schematic of Interfacing 2732 EPROM

In this schematic, two steps—the decoding of the address and generating of the Memory Select (MSEL)—are combined by using the 74LS138 3-to-8 decoder. The decoder has three inputs, three enable lines, and eight output lines. Two enable lines are active low and one is active high. Once the decoder is enabled, only one output line, corresponding to the input combination, goes active (low).

In Figure 4.7, the output O_0 of the decoder is shown as Memory Select (\overline{MSEL}_0), which is connected to the Chip Enable (\overline{CE}) of the memory, and O_0 goes active low when the address lines $A_{15}-A_{12}$ and the \overline{MREQ} are all at logic 0. The address line A_{15} and the \overline{MREQ} are used to enable the decoder (active low); the address lines A_{14} , A_{13} , and A_{12} are used as input to the decoder, and the enable line E_3 is connected to +5 V. No other logic level on these address lines can assert the \overline{MSEL}_0 signal.

Step 4: Connect the Z80 control signal to enable an appropriate buffer.

Figure 4.7 shows that the Z80 Read (\overline{RD}) is connected to the \overline{OE} signal of the memory chip. When the RD signal is asserted, the output buffer is enabled and the data byte from the selected register is placed on the data bus.

4.23 Memory Map

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 $A_{15}-A_{12}$ have to be 0 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 map of this chip ranges from 0000_{16} to $0FFF_{16}$.

We can verify the memory map in terms of our analogy of page and line numbers. The chip's 4,096 bytes of memory can be viewed as 16 pages with 256 lines each. Let us examine the high order Hex digits of the map; they range from 00 to 0F, indicating 16 pages—0000 to 00FF and 0100 to 01FF, for example.

4.3

ILLUSTRATIVE EXAMPLE 2: INTERFACING STATIC R/W MEMORY

In this example, we will use the MOSTEK MK4802 memory chip to demonstrate both Read and Write operations. To simplify the discussion, we will use the same decoding circuit as in Figure 4.7, except that the $\overline{\text{MSEL}}_4$ signal is used as the Chip Enable. This chip has 2K memory; therefore, one address line (A_{11}) will be left as “don’t care” in order to use the previous circuit. Because of the “don’t care” address line, the memory registers will have multiple addresses, and the memory chip will occupy more memory space than necessary, as explained later.

4.31 MOSTEK MK4802 Static R/W Memory

This is a 2K static R/W memory chip, organized in a 2048×8 format. It has eleven address lines ($A_{10}-A_0$), eight data lines, and three control signals: $\overline{\text{CE}}$, $\overline{\text{OE}}$, and $\overline{\text{WE}}$. We are already familiar with the first two control signals; the third signal $\overline{\text{WE}}$ (Write Enable) is active low and used to enable the input buffer of the memory. The logic pinout and the pin configuration are shown in Figure 4.8.

4.32 Interfacing Circuit

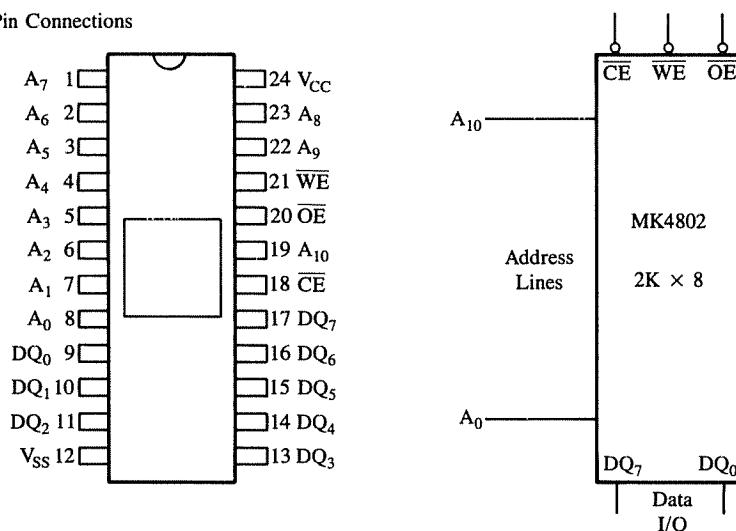
Figure 4.9 shows the interfacing circuit using the MK4802 memory chip. The decoding circuit is the same as in Figure 4.7. We will analyze this circuit in terms of the same four steps outlined previously.

Step 1: The Z80 address lines $A_{10}-A_0$ are connected to pins $A_{10}-A_0$ of the memory chip to address 2048 registers; the address line A_{11} is not necessary for the chip. The address line A_{11} can be connected to the decoder by modifying the circuit, but we have left it “don’t care” to observe its effects on the memory map.

Step 2: The Memory Select $\overline{\text{MSEL}}_4$ line (the output 0_4 of the decoder) is used as the & Chip Enable ($\overline{\text{CE}}$). The $\overline{\text{CE}}$ is asserted only when the address on $A_{15}-A_{12}$ is **Step 3: 0 1 0 0**.

Step 4: In the case of a R/W memory, we need two control signals: Read ($\overline{\text{RD}}$) and Write ($\overline{\text{WR}}$), both active low. The $\overline{\text{RD}}$ is connected to $\overline{\text{OE}}$, as in the previous illustration, to enable the output buffer. The $\overline{\text{WR}}$ is connected to $\overline{\text{WE}}$ (Write Enable) of the memory chip, and when $\overline{\text{WR}}$ is asserted low, the input buffer of the memory chip is enabled, allowing data to be written into the selected memory register.

Pin Connections



Pin Names

A ₀ -A ₁₀	Address Inputs	V _{CC}	Power (+5 V)
CE	Chip Enable	WE	Write Enable
V _{SS}	Ground	OE	Output Enable
DQ ₀ -DQ ₇	Data In/Data Out		

FIGURE 4.8

MK4802 Static R/W Memory Pin Configuration and Logic Symbol

SOURCE: Courtesy of Mostek Corporation

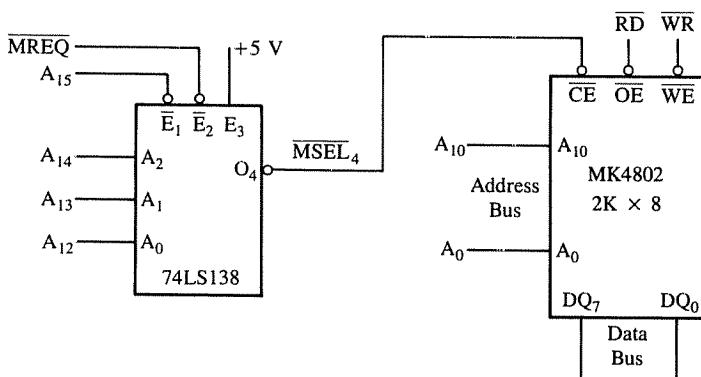


FIGURE 4.9

Schematic of Interfacing Static R/W Memory MK4802

4.33 Memory Map

Assuming the “don’t care” address line A_{11} at logic 0, the memory map of this memory chip ranges from 4000_H to $47FF_H$.

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	1	0	0	X	0	0	0	0	0	0	0	0	0	0	$= 4000_H$
<u>MSEL₄</u>				X	1	1	1	1	1	1	1	1	1	1	$= 47FF_H$

If we assume A_{11} at logic 1, the memory map ranges from 4800_H to $4FFF_H$ 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
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	$= 4800_H$
<u>MSEL₄</u>				1	1	1	1	1	1	1	1	1	1	1	$= 4FFF_H$

The entire memory map appears to be from 4000_H to $4FFF_H$ (4K memory). In reality, we have only 2K memory occupying the memory space of 4K. Because of the one “don’t care” line, each register can have two addresses. For example, the addresses 4000_H and 4800_H will select the same register. The duplicate or redundant range of the memory addresses (4800_H to $4FFF_H$) is generally known as the **foldback memory**; this memory space cannot be used by any other memory chip.

4.34 Absolute versus Linear Decoding

In Illustrative Example 1 (Figure 4.7), all the high-order address lines A_{15} – A_{12} were decoded to select the memory chip, and the memory chip is selected only for the specified logic levels (all 0s) 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. In Illustrative Example 2 (Figure 4.9), high-order address lines were partially decoded, resulting in multiple addresses. In small systems, hardware for the decoding logic can be eliminated by using individual high-order address lines to select memory chips. For example, in Figure 4.9, A_{15} can be directly connected to the memory chip, thus eliminating the decoder, and the chip is selected whenever $A_{15} = 0$. This is called linear decoding. The scheme can reduce cost, but has the drawback of multiple addresses. The linear decoding can be used in small systems, such as a microwave oven, where memory requirements are limited and further expansion is unlikely.

4.4

ILLUSTRATIVE EXAMPLE 3: DESIGNING MEMORY

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 given specifications.

4.41 Problem Statement

Given a 2K R/W (2048 \times 8) static memory chip and one 3-to-8 decoder, design memory for the beginning address 2800_H. Use the MREQ signal to enable one of the decoder lines, and the RD and WR control signals can be directly connected to the memory chip.

4.42 Problem Analysis

1. The 2K memory requires 11 address lines (A₁₀–A₀), and the remaining five address lines A₁₅–A₁₁ can be used to generate the Memory Select signal.
2. The 74LS138 decoder has three input lines and three enable lines: two active low and one active high. Out of the five address lines, three lines can be used as input to the decoder, and two address lines and the MREQ can be used to enable the decoder.
3. The Z80 control signals RD and WR should be connected to the Output Enable (OE) and Write Enable (WE) of the memory chip, respectively.
4. To assign the starting address 2800_H. The address lines should have the following logic levels:

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	1	0	0	0	0	0	0	0	0	0	0	0	= 2800 _H

MSEL₁

These logic level requirements dictate that A₁₃ and A₁₁ should be 1, and that A₁₅, A₁₄, and A₁₂, should be 0. We can connect A₁₃ to the active high enable line (E₃), and A₁₅ and MREQ to the active low enable lines (E₁ and E₂) of the decoder, respectively, and by connecting the output O₁ of the decoder to CE, we can ensure that the memory chip is selected when A₁₁ = 1.

4.43 Circuit Analysis

Figure 4.10 shows the schematic of the interfacing circuit based on the problem analysis.

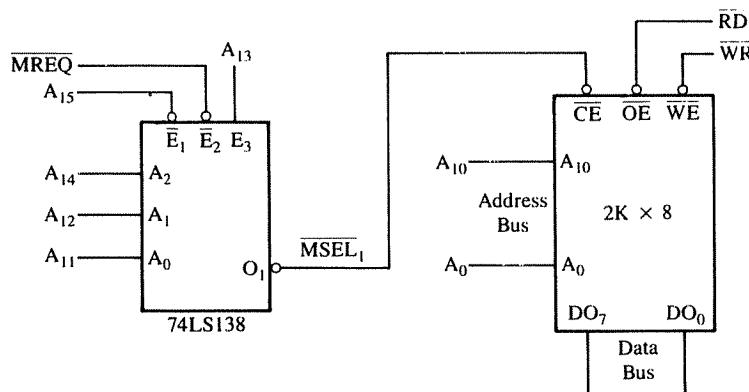


FIGURE 4.10
Schematic of 2K Memory Design

1. The output line O_1 of the decoder is connected to the Chip Enable (\overline{CE}) of the memory chip. The decoder is enabled when $A_{15} = 0$, $A_{13} = 1$, and MREQ is asserted low. The output line O_1 of the decoder is asserted only when input lines are at logic 0 0 1. The logic levels on these address lines will assign the starting address as 2800_H .
2. In this design example, the MREQ is used to enable the decoder. The output of the decoder goes low only when the MREQ is asserted; thus, the chip is enabled when the MREQ is low. The output and the input buffers of the memory chip are enabled directly by the control signals RD and WR.

4.5

TESTING AND TROUBLESHOOTING INTERFACING CIRCUITS

In the last section, we discussed how to design or interface memory for a given address. The next step is to test and verify that we can store a byte at a memory location within the address range of the memory chip and read the byte. At this point, we need to make an assumption that we have a working microcomputer system, and the memory design is an expansion of the existing system. If we are designing a system, we may need to use an in-circuit emulator to test the memory; this is discussed in Chapter 17.

To test the memory, we can simply access an address such as 2800_H through the system keyboard, store a byte, and check the address location again to verify the byte. If there is any fault in the interfacing circuit, the system is likely to show an error message, or a different byte from the one we stored will be displayed. Now we need to troubleshoot the interfacing circuit. The question is: Where do we begin? The obvious step is to check the wiring and the 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 a **diagnostic routine**, as shown.

Diagnostic Routine

START: LD A, F7H ;Load F7H into the accumulator

LD (2800H), A ;Store accumulator contents in location 2800H

JP START ;Jump back to beginning and repeat

This routine has three instructions. The first instruction loads $F7_H$ into the accumulator, and the second instruction stores the byte in the memory location 2800_H . The third instruction is a Jump instruction that takes the program control at the beginning, and these three instructions are repeated 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 loop in the machine cycles as follows (it will be helpful to have read Chapter 6 to understand the diagnostic routine):

Instruction	Bytes	T-states	Machine Cycles			
			M ₁	M ₂	M ₃	M ₄
LD A, F7H	2	7 (4, 3)	Opcode Fetch	Memory Read		
LD (2800H), A	3	13 (4, 3, 3, 3)	Opcode Fetch	Memory Read	Memory Read	Memory Write
JP START	3	10 (4, 3, 3)	Opcode Fetch	Memory Read	Memory Read	

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 2 MHz, the loop is executed in 15 μ s, and the WR signal, repeated every 15 μ s, can be observed on a scope. If we sync the scope on the WR pulse from the Z80, we can check M₁, the output of the decoder MSEL₁ and memory signals CE, WR, and RD; three of these signals are in Figure 4.11.

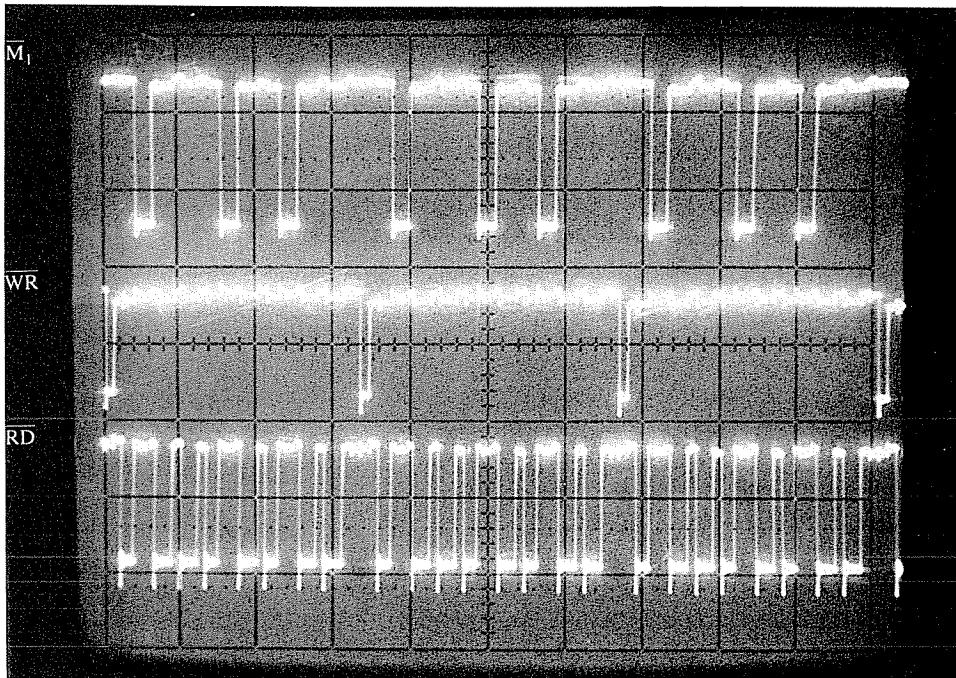


FIGURE 4.11
Timing Signals of Diagnostic Routine
SOURCE: Photograph by Gregg Texido

When the Z80 asserts the WR signal, the high-order address $A_{15}-A_{11}$ must be 0 0 1 0 1, and MSEL₁ must be asserted low. If MSEL₁ is high, it indicates that the address lines $A_{15}-A_{11}$ or MREQ are improperly connected or the decoder chip is faulty.

If MSEL₁ is low, it confirms that the decoding circuit is functioning properly. Now if we check the entire address bus and the data bus in relation to the WR signal, one line at a time, we must read the address 2800_H and the data $F7_H$. 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 4.11.

4.6

SOME QUESTIONS AND ANSWERS

In the above discussion of memory interfacing, we focused on certain aspects of the communication process between the Z80 and memory. However, in order to avoid distraction from basic concepts, we did not address several important issues. Now we will attempt to answer those questions briefly or provide references for them.

1. How do you determine whether a memory chip is too slow for a given Z80 system?

The response time of a memory chip is defined in terms of *Access Time*. This is the time delay between when the microprocessor places a memory address on the address bus and when memory places a data byte on the data bus. Typically, Access Time is 50–450 ns for static R/W memory. Similarly, the microprocessor has a timing specification: the time delay after the Z80 places an address on the address bus to when it begins to read data on the data bus. The memory access time must be less than this microprocessor time delay. This will be discussed when we consider advanced topics in memory interfacing.

2. How do you interface a memory chip with slow response time?

If the memory response time is slower than the microprocessor read time, the Memory Read cycle can be extended by using the WAIT signal. During the T_2 state of the Memory Read cycle, the Z80 samples the WAIT signal, and if it is low, the Z80 adds Wait states until the signal goes high again. Typically, one Wait state (one clock cycle) provides sufficient time for memory to place data on the data bus. Extra circuitry is necessary for adding Wait states; this is discussed in Chapter 16.

3. Why did you not include an illustrative example of dynamic memory?

The dynamic memory stores information as a capacitive charge; therefore, information needs to be refreshed every few milliseconds. In the latter part of the Opcode Fetch

cycle, the Z80 uses the low-order bus for refresh addresses. To interface the dynamic memory, additional refresh circuits that can use the refresh addresses from the Opcode Fetch cycle are necessary. This will also be discussed in Chapter 16.

SUMMARY

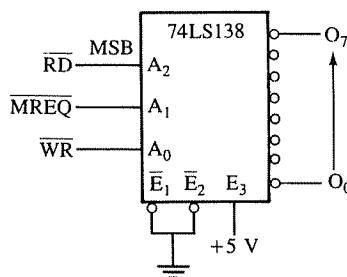
- To read from memory, the address of the register to be read from should be placed on the address lines, and the Chip Enable \overline{CE} and \overline{RD} signals must be asserted low to enable the output buffer.
- To write into memory, the address of the register to be written into should be placed on the address lines; a data byte should be placed on the data lines, and the Chip Enable \overline{CE} and \overline{WR} signals must be asserted low to enable the input buffer.
- The Z80 identifies memory operations by initiating the \overline{MREQ} signal. This signal is combined with the decoded address pulse (CS) to generate Memory Select (MSEL), which is connected to Chip Enable (\overline{CE}) signal of the memory chip. Another alternative is to use the decoded address pulse \overline{CS} to enable the memory chip and generate Memory Read (\overline{MEMRD}) and Memory Write (\overline{MEMWR}) signals by combining \overline{MREQ} , \overline{RD} , and \overline{WR} signals.
- To interface a memory chip with the Z80, the necessary low-order address lines of the Z80 address bus are connected to the address lines of the memory chip. The high-order address lines and the \overline{MREQ} are used to generate the MSEL signal, which enables the chip. The \overline{RD} signal is used to enable the output buffer, and the \overline{WR} signal is used to write into memory by enabling the input buffer.
- In the absolute decoding technique, all the address lines not used by a memory chip to identify a memory register must be decoded; thus, the Chip Select can be asserted by only one address. In the linear decoding technique, one address line can be used for \overline{CS} , and others can be left “don’t care.” This technique saves on hardware, but generates multiple addresses, which result in foldback memory space.
- To troubleshoot an interfacing circuit, a constant and identifiable signal must be generated by writing a continuous loop.

ASSIGNMENTS

1. If a memory chip is organized in a 4096×1 format, specify the number of registers in the chip and the number of bits stored by each register.
2. If $16K \times 1$ memory chips are used in a memory design, how many chips are required to design $64K$ -byte memory?
3. Specify the number of chips necessary to design $8K$ -byte memory with 1024×4 memory chips.

4. In Figure 4.7, generate the Chip Select (\overline{CS}) signal by using a 4-input NAND gate (and inverters) to decode the address lines A_{15} – A_{12} and combine \overline{CS} with the MREQ using an appropriate gate to generate the equivalent MSEL signal.
5. Generate the signal equivalent to the MSEL signal in Figure 4.7 using the 74LS139, which has two 2-to-4 decoders in the package.
6. If the first address of the $8K \times 8$ memory chip is 0800_H , what is the address of the last register?
7. In Figure 4.7, if we connect the output line O_4 (instead of O_0) of the decoder to the \overline{CE} signal, what will be the memory map of the circuit?
8. In Figure 4.7, if we use all the output lines (O_7 – O_0) of the decoder to select 8 memory chips of the same size as the 2732, what is the total range of the memory map?
9. In Figure 4.9, replace the address line A_{15} with A_{11} and find the range of the foldback memory.
10. In Figure 4.9, replace the MK4802 with a $1K \times 8$ memory chip and leave address lines A_{11} and A_{10} as “don’t care.” Find the memory map of the chip and the range of the foldback memory.
11. By examining the range of the foldback memory in Figure 4.9, and in Assignment 10, specify the relationship between the range of foldback memory and the number of “don’t care” lines.
12. In Figure 4.12, the control signals \overline{RD} , \overline{MREQ} , and \overline{WR} are used as inputs to the 3-to-8 decoder, and the decoder is enabled. Specify the output lines that can be used as MEMRD and MEMWR control signals.
13. In Figure 4.12, explain why the output line O_0 cannot be asserted low.
14. In Figure 4.13, specify the memory maps of ROM1, ROM2 and R/WM1.
15. Is there a foldback memory for any one of the chips in Figure 4.13?
16. Sketch the memory map in Figure 4.13.
17. Figure 4.14 illustrates an example of linear decoding. Specify the memory map of each chip without accessing more than one chip at a time.
18. Given a $1K$ (1024×8) EPROM memory chip and one 3-to-8 decoder, design an interfacing circuit to assign the beginning address at 0400_H . Use the 74LS32 OR gate to generate the control signal MEMRD.
19. You are given the 74LS139 (two 2-to-4 decoders) and $8K$ static R/W memory.

FIGURE 4.12
Generating Control Signals
Using the 3-to-8 Decoder



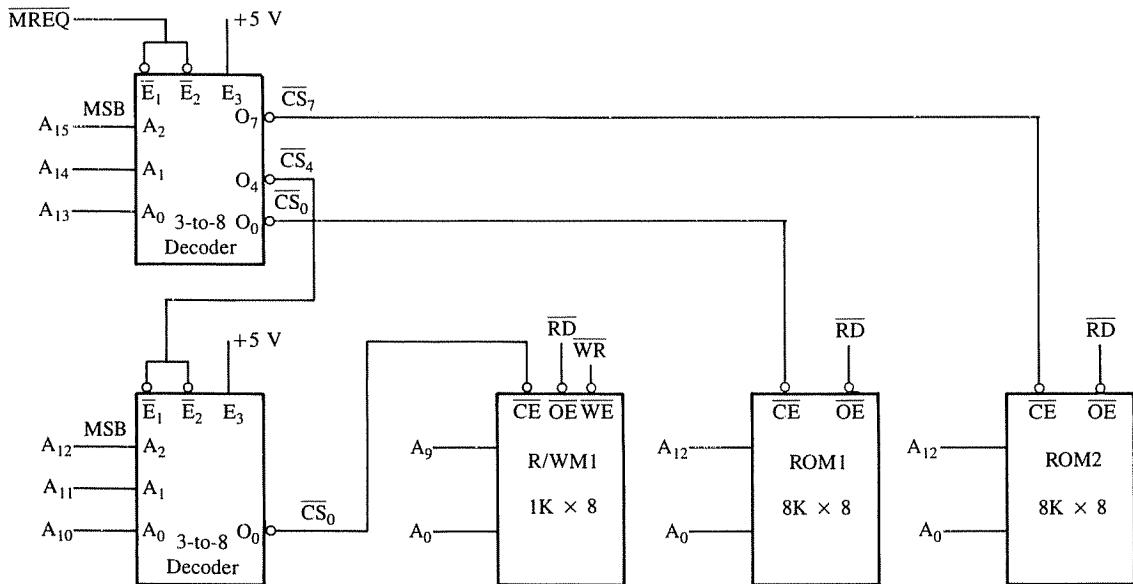


FIGURE 4.13
Schematic for Assignments 14–16

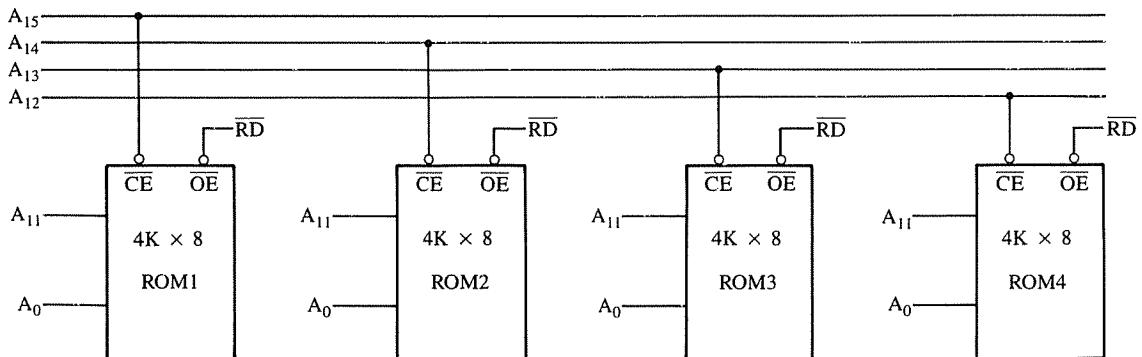


FIGURE 4.14
Memory Map with Linear Decoding

Use one decoder to assign the memory map with the starting address at 8000_H and use the other decoder to generate the MEMRD and MEMWR control signals.

20. In Section 4.5, if the diagnostic routine is executed on a system with the clock frequency 4 MHz, specify the time interval between two WR pulses.

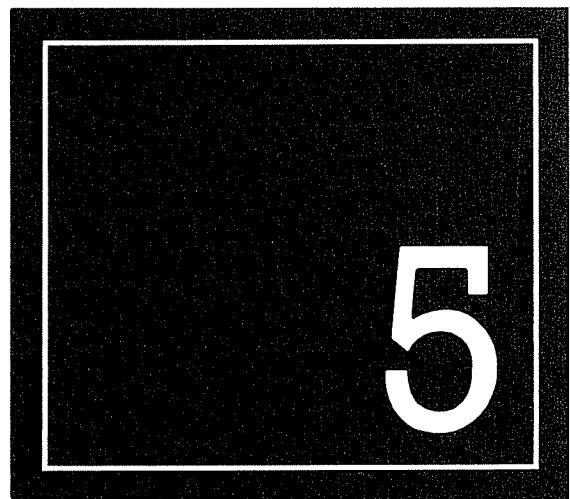
21. In the diagnostic routine, how many times is the $\overline{\text{MREQ}}$ signal asserted in one loop?
22. Specify the logic levels of the address lines A_{15} and A_{13} and the data lines D_7 and D_3 when the $\overline{\text{WR}}$ signal is asserted during the diagnostic routine.
23. How many times is the M_1 signal asserted during the execution of the diagnostic routine?
24. Sketch the waveforms of $\overline{\text{MSEL}}_1$ signals in one loop.

Interfacing I/O Devices

The I/O (Input/Output) is the third component of a microcomputer system. 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 mode is the **serial I/O**, whereby one bit is transferred using one data line; typical examples include peripherals such as CRT terminals or cassette tapes. In this chapter, we focus on interfacing I/O devices in the parallel mode; the serial mode will be discussed in Chapter 15.

In the parallel I/O mode, devices can be interfaced using two 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. 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 through its interfacing circuit. When the Z80 is programmed to transfer data, 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.

To grasp the essence of interfacing techniques,



we first examine the machine cycles of I/O instructions to determine the timings for I/O data arriving on the data bus, and then latch (or catch) that information. We derive the basic concepts of peripheral-mapped and memory-mapped I/O from the machine cycles. The peripheral-mapped I/O concepts are illustrated with two examples: interfacing LEDs as an output device and switches as input device. The memory-mapped I/O technique is illustrated with an example of appliance control. The chapter also includes additional interfacing examples that occur frequently in microprocessor-based products.

OBJECTIVES

- Illustrate the Z80 bus contents and control signals when OUT and IN instructions are executed.
- Explain the necessity of Wait states in I/O machine cycles.
- Recognize the device (port) address of a peripheral-mapped I/O by analyzing the associated logic circuit.
- 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 such MSI chips as decoders, latches, and buffers.
- Explain the concepts in interfacing analog devices such as sensors and motors.

5.1

INTERFACING OUTPUT DEVICES

In peripheral-mapped I/O, a device is identified with an 8-bit address, and I/O-related control signals are used to enable the device. The process of data transfer is in many ways similar to that of reading from or writing into a memory register. The Z80 uses the instruction IN to read (input) data from an input device and uses the instruction OUT to write (send) data to an output device. To understand interfacing of I/O devices, we need to examine the execution and machine cycles of these input/output instructions. In the next section, we will examine the execution of the OUT instruction and discuss the interfacing of output devices, and in Section 5.3, we will examine the IN instruction and discuss the interfacing of input devices.

5.11 OUT Instruction

The Z80 microprocessor has several output instructions to send (copy or write) data to an output device. It can send data from the accumulator, internal general-purpose registers, or memory registers to an output device. The Out instructions include the 8-bit address of a device as an operand. Therefore, the address can be any of the 256 8-bit binary combinations from 00_H to FF_H . Thus, an output device can be assigned any 8-bit address between 00_H and FF_H through an appropriate interfacing circuit. The address range from 00_H to FF_H is called the I/O or peripheral map, and an address can be referred to as a device address, port address, or port number. Among the several Out instructions, we will examine the machine cycles and timing of the following instruction.

Opcode	Operand	Description
OUT	(8-bit), A	<p>This is a 2-byte instruction with the hexadecimal opcode D3, and the second byte is the port address of an output device.</p> <p>This instruction transfers (copies) data from the accumulator to the output device.</p>

Typically, to display the contents of the accumulator at an output device (such as

LEDs) with the address, for example, 07_H , the instruction will be written and stored in memory as follows:

Memory Address	Machine Code	Mnemonics	Memory Contents							
2050	D3	OUT (07H), A ; 2050	<table border="1"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> </table> = $D3_H$	1	1	0	1	0	0	1
1	1	0	1	0	0	1				
2051	07	;	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> </table> = 07_H	0	0	0	0	0	1	1
0	0	0	0	0	1	1				

NOTE: The memory locations are chosen here arbitrarily for the illustration.

When the microprocessor reads and executes the machine codes written at memory registers 2050_H and 2051_H , it will transfer (copy) the byte from the accumulator to the LED port with address 07_H and display the byte. Now the question remains: How is the address 07_H assigned to the output port? To answer that question, we need to examine the machine cycles of this instruction, as shown in the next section.

5.12 Execution of OUT Instruction and Timing

The OUT instruction has three machine cycles: Opcode Fetch, Memory Read, and I/O Write. The Z80 reads the opcode and the port address from memory in the first two machine cycles and writes into the port in the third cycle. Figure 5.1 shows the timing of the OUT instruction with the port address 07_H illustrated in the previous section.

The first two machine cycles—Opcode Fetch and Memory Read—are similar to the machine cycles shown in Figure 3.7; however, in Figure 5.1, the low-order and high-order address buses are shown separately to illustrate the contents of the low-order bus in the third cycle. In the Opcode Fetch cycle, the Z80 places the address 2050_H on the address bus and fetches the opcode $D3_H$ (1 1 0 1 0 0 1 1) via the data bus. When the Z80 decodes the opcode, it realizes that the instruction consists of two bytes, and that it must read the second byte. In the second machine cycle, the Z80 places the next address, 2051_H , on the address bus and reads the port address 07_H .

In the third machine cycle, M_3 (I/O Write), the following events occur:

1. The Z80 places the port address 07_H on the low-order address bus and the contents of the accumulator on the data bus.
2. During T_2 , it asserts the \overline{IORQ} and \overline{WR} control signals; the assertion of \overline{IORQ} indicates that it is an I/O operation.
3. The Z80 automatically inserts a single Wait state T_W after T_2 to allow sufficient response time for an I/O device; this Wait state is added regardless of the WAIT signal status.
4. During T_3 , the control signals \overline{IORQ} and \overline{WR} become inactive.

To interface an output device, the information on the buses during the M_3 cycle is critical. From the beginning of T_2 until the end of T_3 , we have the port address (07_H) on the low-order address bus and the data byte to be displayed on the data bus. The availability of this information is indicated by the control signals. Now what we must do is to latch

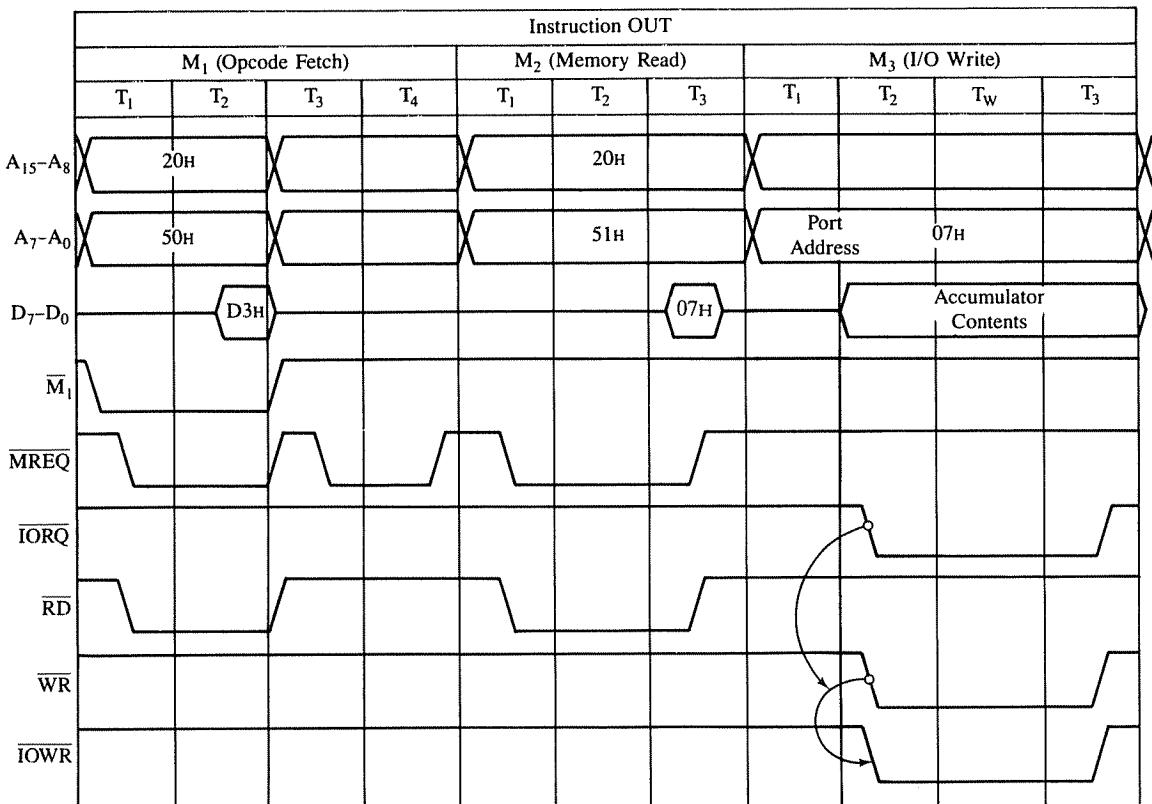


FIGURE 5.1
Z80 Timing for Execution of OUT Instruction

(catch) this information using the control signals before it disappears from the buses; we need to open the gate at that precise moment to let the data flow to the “outside world.” This is the essence of interfacing.

5.13 Basic Concepts in Interfacing Output Devices

The concepts in interfacing output devices are similar to those in interfacing memory. The steps can be listed as follows:

1. Decode the low-order address bus to generate a unique pulse corresponding to the port address on the bus; this is called the I/O address (IOADR) pulse.
2. Combine (AND) the I/O address pulse (IOADR), IORQ, and WR to generate the IOSEL (I/O select) pulse (Figure 5.2(a)). Another approach is to generate the IOWR (I/O Write) by combining IORQ and WR, and then combine IOWR with the IOADR (I/O Address) pulse to generate the IOSEL pulse (Figure 5.2(b)). The critical concept here is that the decoded address, IORQ, and WR are all necessary to latch the data at

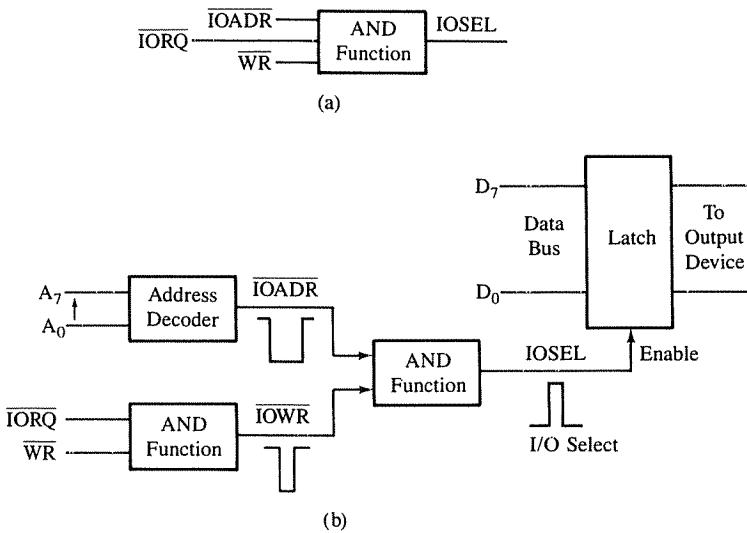


FIGURE 5.2
Block Diagram: Output Interfacing

the appropriate time; how these signals are combined is often dictated by availability of decoding devices (chips) in the system.

3. Use the IOSEL pulse to enable (activate) the output device.

Let us examine the significance of the I/O select pulse. This pulse is generated by ANDing the decoded address, $\overline{\text{IORQ}}$, and $\overline{\text{WR}}$ signals as shown in Figure 5.2(a); all these signals are active low. The assertion of this pulse indicates two pieces of information: (1) the low-order address bus has the port address (07_H), and (2) the data byte from the accumulator is on the data bus. Thus, this is the appropriate time to enable the latch (or open the gate for data). Figure 5.2(b) shows how these control signals are generally ANDed in a typical interfacing circuit to generate the IOSEL pulse and how the I/O select pulse is used to enable the output latch.

ILLUSTRATIVE EXAMPLE 1: INTERFACING LEDS

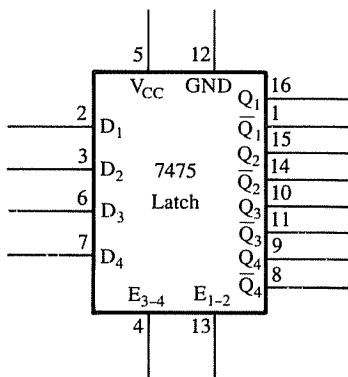
5.2

In this section, we will analyze an actual interfacing circuit with the port address 07_H to display binary data. A group of 8 LEDs will be used to indicate binary 1s and 0s and will be connected to the data bus using the 7475 latches.

5.21 Hardware

Figure 5.3 shows the logic symbols of the 7475 latch. It has four *bistable latches* controlled by the active high enable signals; E_{1-2} enables the first two latches and E_{3-4} enables

FIGURE 5.3
Logic Symbol: 7475 Latch



the remaining two. When E is high, data enter the latch and appear at the Q outputs, and Q outputs correspond to the input data. When E goes from high to low, data will be latched and will remain stable until E goes high again.

When Q output is high, it can supply (source) $400 \mu\text{A}$, and when it is low, it can sink 16 mA current. Since most LEDs require a $10\text{--}15 \text{ mA}$ current to be properly illuminated, they are connected to \bar{Q} output of the latch so that when the input is high, \bar{Q} output is low and the LED is turned on.

5.22 Interfacing Circuit

Figure 5.4 shows an interfacing circuit for the LED output port with the address 07_H . We will analyze this circuit in terms of the three steps for interfacing output devices as outlined in Section 5.13.

1. An 8-input NAND gate with five inverters is used to decode the low-order address bus $A_7\text{-}A_0$. The output of the NAND gate is asserted when the address is $0\ 0\ 0\ 0\ 0\ 1\ 1$ (07_H); thus, the NAND gate performs the decoding function to generate the I/O address (IOADR) pulse.
2. The control signals \overline{IORQ} and \overline{WR} are ANDed in a negative AND gate (physically, an OR gate) to generate the control signal \overline{IOWR} (active low). The \overline{IOWR} is again ANDed (through a NOR gate) with the I/O address pulse to generate the I/O select pulse (active high). This pulse is asserted only when the address is 07_H and the control signals \overline{IORQ} and \overline{WR} are low.
3. The I/O select pulse is used to enable the latches 7475. The data bus $D_7\text{-}D_0$ is connected to the D input and the LED cathodes are connected to the \bar{Q} output of the latch. The LED anodes are connected to the power supply $+5 \text{ V}$ through the current-limiting resistors.

At the beginning of T_2 in the third machine cycle shown in Figure 5.1, the control signals \overline{IORQ} and WR are asserted, and the I/O select pulse (Figure 5.4) goes high if the address is 07_H . When the I/O select pulse goes high, the data on the data bus enters the

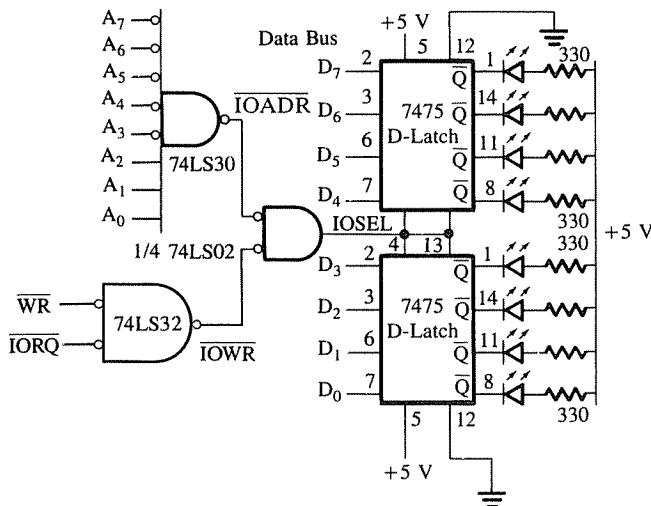


FIGURE 5.4
Schematic: Interfacing LED Output Port

latches. During T_3 , when the control signals become inactive, the I/O select pulse goes low, and the data are latched. The logic 1s on the data lines turn on the corresponding LEDs because when a data bit is high, the \bar{Q} output is low and the LED is turned on.

INSTRUCTIONS

To display data, for example $97H$, at this LED port, instructions are as follows:

LD A, 97H ; Load accumulator with the specified byte

OUT (07H), A ; Display the accumulator contents at port $07H$

The first instruction (LD A) stores the second byte $97H$ in the accumulator, and the OUT instruction sends the byte ($97H$) from the accumulator to the LED port $07H$. When the I/O select pulse is asserted, the byte $97H$ enters the latch and is displayed by the LEDs. When it goes low (inactive), the byte is latched and continues to be displayed by the LEDs.

INTERFACING INPUT DEVICES

5.3

The interfacing of input devices is almost identical to that of interfacing output devices, but with some differences in bus signals and circuit components. In this discussion, we will assume that you are familiar with the basic concepts of interfacing (Section 5.13) and

describe only the additional details. First, we examine the execution and timing of the IN instruction and discuss the interfacing of input devices in relation to the timing diagram.

5.31 IN Instruction

The Z80 instruction set includes several instructions to read (copy) data from such input devices as switches, keyboards, and A/D data converters. These instructions can read an input device and place the data into the accumulator, Z80 registers, or memory registers. These are two-byte instructions; the first byte is the opcode, and the second byte specifies port address. Although there are numerous ways of specifying the port address, it is always eight bits long. Thus, the addresses for input devices can range from 00_H to FF_H . Among the several Input instructions available, we will examine the machine cycles and timing of the following instruction.

Opcode	Operand	Description
IN	A, (8-bit)	<p>This is a two-byte instruction with the hexadecimal opcode DB, and the second byte is the port address of an Input device.</p> <p>This instruction reads (copies) data from an input device and places the data byte into the accumulator.</p>

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

Memory Address	Machine Code	Mnemonics	Memory Contents																
2065	DB	IN A, (84H) ; 2065	<table border="1"> <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> <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> = DB _H	1	1	0	1	1	0	1	1	1	0	0	0	0	1	0	0
1	1	0	1	1	0	1	1												
1	0	0	0	0	1	0	0												
2066	84	;	2066 = 84 _H																

NOTE: The memory locations 2065_H and 2066_H are selected arbitrarily for the illustration.

When the microprocessor is asked to execute these instructions, it will first read the machine codes (or bytes) stored at locations 2065_H and 2066_H , then read the switch positions at port 84_H by enabling the interfacing device of the port. The data byte indicating switch positions from the input port will be placed in the accumulator. To design an interfacing circuit with the port address 84_H , we now need to examine the machine cycles and execution timing of the IN instruction.

5.32 Execution of IN Instruction and Its Timing

The IN instruction has three machine cycles: Opcode Fetch, Memory Read, and I/O Read. In the first two machine cycles, the Z80 reads the opcode DB and the port address 84_H (see example in previous section). These cycles are identical to the first two machine cycles of

the OUT instruction shown in Figure 5.1. In the third machine cycle, the Z80 reads a data byte from the input port as follows (Figure 5.5):

1. The port address $84H$ is placed on the low-order address bus at the beginning of the machine cycle M_3 (I/O Read).
2. During T_2 , the control signals \overline{IORQ} and \overline{RD} are asserted, and one Wait state is inserted automatically after T_2 .
3. During T_3 , the Z80 reads the data bus and then causes the control signals (\overline{IORQ} and \overline{RD}) to go inactive.

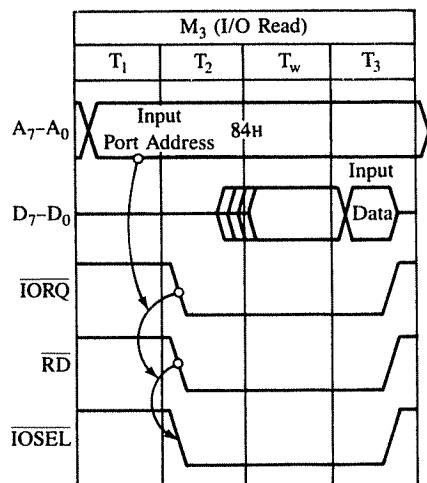
5.33 Basic Concepts in Interfacing Input Devices

To interface an input port with the address $84H$, we need to logically AND the information on the address bus with the control signals and enable the input port. The steps are as follows:

1. Decode the low-order bus to generate the I/O address pulse.
2. Combine the I/O address pulse with the control signals \overline{IORQ} and \overline{RD} to generate the signal I/O Select (IOSEL, Figure 5.5). Another approach is to combine \overline{IORQ} and \overline{RD} to generate an \overline{IORD} signal and then to combine the \overline{IORD} with the I/O address pulse to generate the I/O select pulse.
3. Enable the input interfacing device using the I/O select pulse.

These steps are identical to those listed for interfacing output devices; the only differences are (1) the control signal is \overline{RD} instead of \overline{WR} , and (2) data flow from an input port to the accumulator rather than from the accumulator to an output port.

FIGURE 5.5
Z80 Timing for Execution of IN Instruction



5.4

ILLUSTRATIVE EXAMPLE 2: INTERFACING INPUT SWITCHES

In this section, we will analyze the circuit used for interfacing eight DIP switches as shown in Figure 5.6. 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 84_H ; however, it also has multiple addresses.

5.41 Hardware

Figure 5.6 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 signal \overline{OE} . When \overline{OE} is low, the input data appear on the output lines, and when \overline{OE} is high, the output lines assume high impedance state.

5.42 Interfacing Circuit

Figure 5.6 shows that the low-order address bus (with the exception of lines A_4 and A_3) is connected to the decoder 74LS138; the address lines A_4 and A_3 are left in “don’t care” state. The output line O_4 of the decoder goes low when the address bus has the following address (we assume the “don’t care” lines are at logic 0):

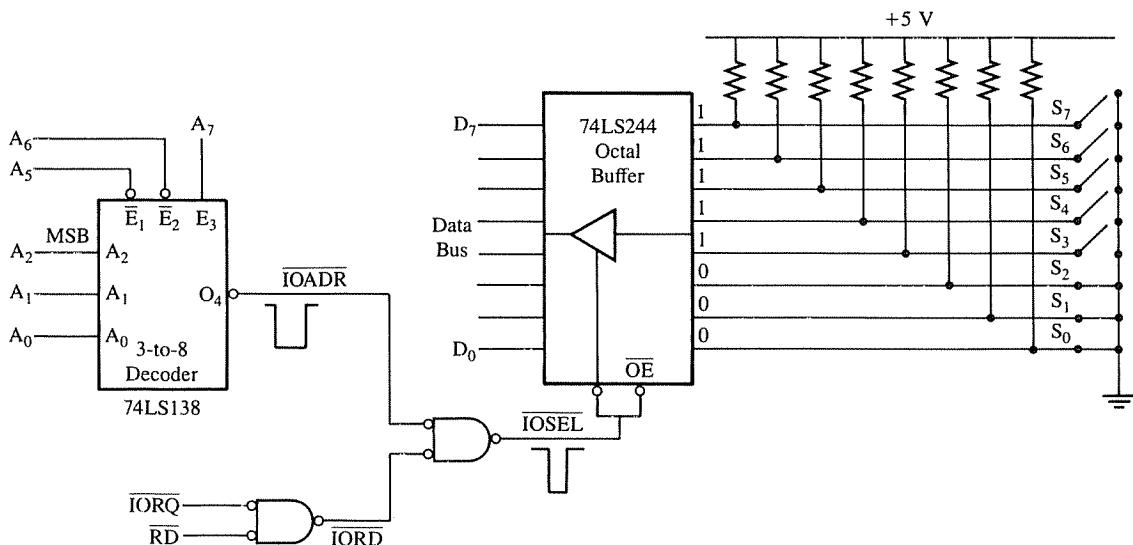
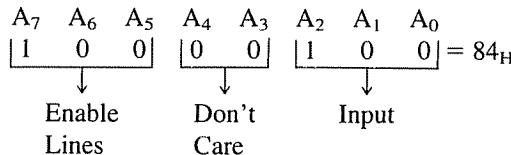


FIGURE 5.6
Schematic: Interfacing Input Switches



The control signal I/O Read ($\overline{\text{IORQ}}$) is generated by ANDing the $\overline{\text{IORQ}}$ and $\overline{\text{RD}}$ in a negative NAND gate, and the I/O select pulse is generated by ANDing the output of the decoder with the control signal $\overline{\text{IORQ}}$. When the address is 84_{H} and the control signals $\overline{\text{IORQ}}$ and $\overline{\text{RD}}$ are 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 Z80 then reads switch positions during T_3 (Figure 5.5) and places the data byte into the accumulator. When a switch is closed, it has logic 0, and when it is open, it is tied to $+5\text{ V}$, representing logic 1. Figure 5.6 shows that the switches S_7 – S_3 are open and S_2 – S_0 are closed; thus, the input reading will be $F8_{\text{H}}$.

5.43 Multiple Port Addresses

In Figure 5.6, the address lines A_4 and A_3 are not used by the decoding circuit; the logic levels on these lines can be either 0 or 1. Therefore, this input port can be accessed by four different addresses, as shown.

A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	= 84_{H}
1	0	0	0	0	1	0	0	= $8C_{\text{H}}$
				0				= 94_{H}
				1				= $9C_{\text{H}}$
				1				

5.44 Instructions to Read Input Port

To read data from the input port shown in Figure 5.6, the instruction IN A, (84H) can be used. When this instruction is executed, the Z80 places the address 84_{H} on the low-order bus, asserts the control signals, and reads the switch positions.

MEMORY-MAPPED I/O

5.5

In memory-mapped I/O, the input and output devices are assigned and identified by 16-bit addresses. To transfer data between the microprocessor and I/O devices, memory-related instructions (such as LD A, (16-bit)) and memory control signals (such as MREQ) are used. The microprocessor communicates with an I/O device as if it were one of the memory locations.

5.51 Memory-Related Data Transfer Instructions

To understand the memory-mapped I/O technique, we need to examine how a data byte is transferred from the Z80 to a memory location or vice versa. For example, the following instruction will transfer (copy) the contents of the accumulator to the memory location 8000_H . It is assumed here that the instruction is stored in memory locations 2050_H , 51_H , and 52_H .

Memory Address	Machine Code	Mnemonics	Comments
2050	32	LD (8000H), A	;Store contents of accumulator
2051	00		in memory location 8000_H
2052	80		

This is a 3-byte instruction; the first byte is the opcode, and the second and the third byte specify the memory address. However, the 16-bit address 8000_H is entered in the reverse order; the low-order byte 00 is stored in location 2051, followed by the high-order address 80_H (the reason for the reversed order will be explained in Section 5.7). 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.

Similarly, the instruction LD A, (4000H) will transfer the contents of the memory location 4000_H to the accumulator. To assign this address for a memory-mapped input port, we can interface an input device (for example, a keyboard) instead of memory by using the memory-related control signals (MREQ and RD). When the processor executes the instruction, the accumulator receives data from the input device rather than from a memory register 4000_H .

5.52 Execution of Memory-Related Data Transfer Instructions

The execution of memory-related instructions discussed in the previous section is similar to the execution of I/O instructions (Sections 5.1 and 5.3), except that the memory-related instructions have 16-bit addresses.

Figure 5.7 shows the execution of the instruction LD (8000H), A. It has four machine cycles; in the first three machine cycles, the Z80 reads the three bytes. The fourth machine cycle M_4 (Memory Write) is similar to the machine cycle M_3 of the OUT instruction. In this machine cycle, the Z80 places the 16-bit address 8000_H on the address bus and the accumulator contents on the data bus. This is followed by the assertion of the control signals MREQ and WR. The information available during M_4 can be used to interface a memory-mapped output port with the 16-bit address 8000_H .

In memory-mapped I/O, I/O selection and data transfer require steps similar to those required in peripheral-mapped I/O:

1. Decode the entire address bus $A_{15}-A_0$ (rather than just A_7-A_0).
2. Combine the control signals MREQ, WR, and the decoded pulse from Step 1 to gen-

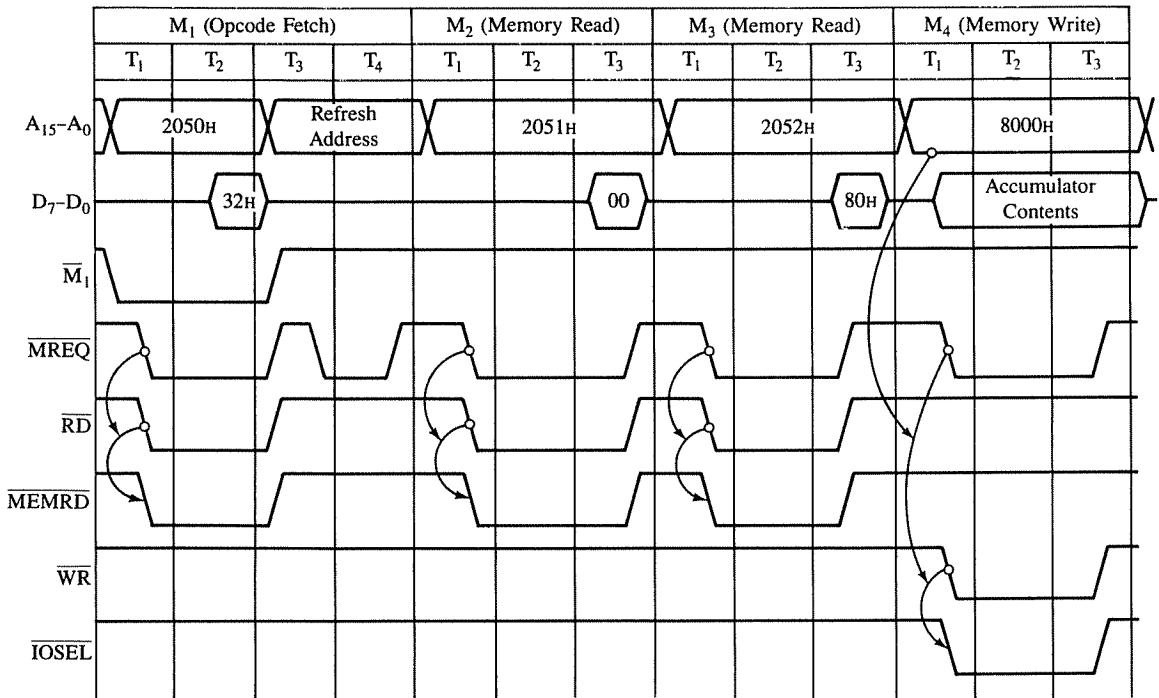


FIGURE 5.7
Z80 Timing for Execution of Instruction LD (8000H), A

erate a pulse similar to the MSEL pulse, which will be used to select an I/O rather than memory.

3. Use the I/O select pulse (actually MSEL) to enable the I/O port.

To interface a memory-mapped input port, the steps are similar to those of the memory-mapped output port. We can use the instruction LD A, (16-bit), which reads data from an input port with the 16-bit address and places it in the accumulator. The instruction has four machine cycles; only the fourth machine cycle differs from M₄ in Figure 5.7. The control signal will be RD rather than WR, and data flow from the input port to the microprocessor.

ILLUSTRATIVE EXAMPLE 3: APPLIANCE CONTROL USING MEMORY-MAPPED I/O TECHNIQUE

5.6

Figure 5.8 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

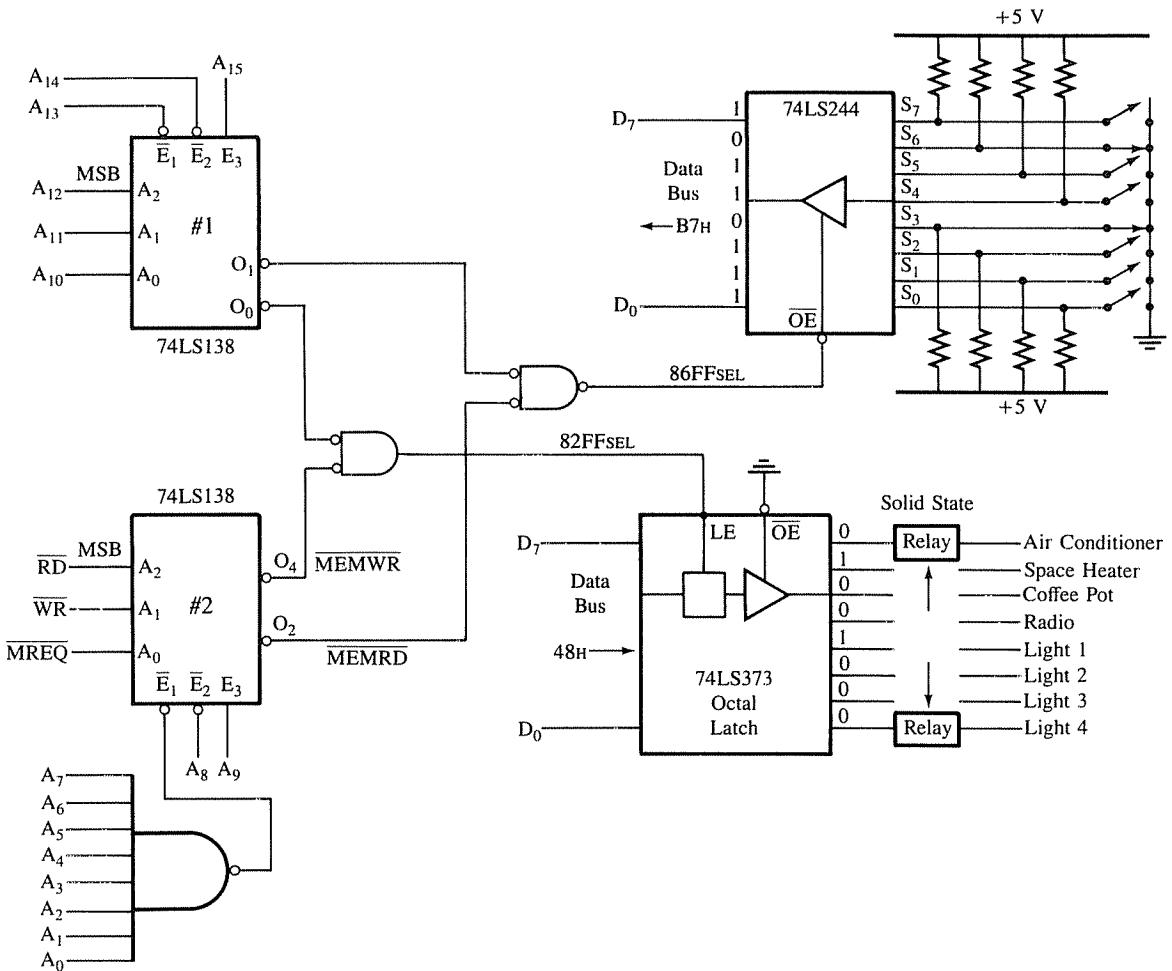


FIGURE 5.8
Schematic: Interfacing I/O Devices with Memory-Mapped I/O

to control the appliances. The appliances are turned on and off by the microprocessor according to the corresponding switch positions. For example, the switch S_7 controls the air conditioner and the switch S_0 controls Light 4. 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 two 3-to-8 decoders and one 8-input NAND gate to decode the address bus and generate the control signals. The eight switches are interfaced using a tri-state buffer 74LS244, and the appliances are interfaced using an octal latch (74LS373) with tri-state output.

5.61 Control Signals

In a memory-mapped I/O circuit, the control signals required are MREQ (Memory Request) and Read (RD) or Write (WR). In this circuit (Figure 5.8), they are used as inputs to a 3-to-8 decoder (labelled #2) to generate additional control signals. The enable lines of the decoder are controlled by the address lines. Assuming the decoder is enabled by the appropriate address, we need to analyze the input and identify the output lines of the decoder that can be used as control signals.

To assert the Memory Write (MEMWR) signal, the input should be MREQ = 0, WR = 0, and RD = 1 (RD and WR cannot be active at the same time). With this input, the output line O₄ goes active and generates the MEMWR signal.

To assert the Memory Read (MEMRD) signal, the input should be MREQ = 0, WR = 1, and RD = 0. With this input to the decoder, the output O₂ goes active and generates the MEMRD signal.

5.62 Output Port and Its Address

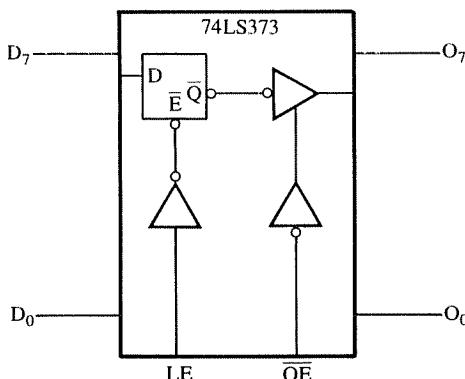
The appliances are connected to the data bus through the latch 74LS373 and solid state relays. If an output bit of the 74LS373 is high, it activates the corresponding relay and turns on the appliance, which remains on as long as the bit stays high. Therefore, to control the appliances, we need to supply the appropriate bit pattern to the latch.

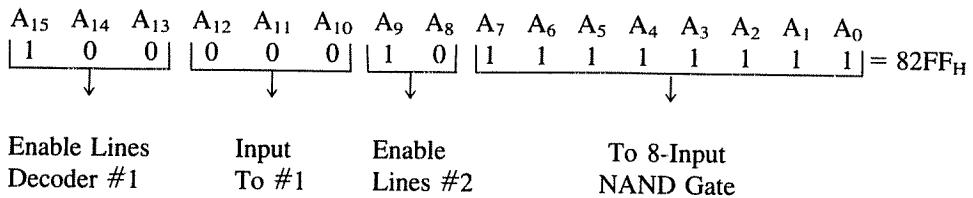
The 74LS373 is a latch followed by a tri-state buffer, as shown in Figure 5.9. The latch and the buffer are controlled independently by Latch Enable (LE) and Output Enable (OE). When LE is high, the data enter the latch, and when LE goes low, data are latched. The latched data are available on the output lines of the 74LS373 if the buffer is enabled by OE (active low). If OE is high, the output lines go into high impedance state.

Figure 5.8 shows that the OE is connected to the ground; thus, the latched data will keep the relays on or off according to the bit pattern. The LE is connected to the device select pulse, which is asserted when the output O₀ of decoder #1 and the control signal MEMWR go low. Therefore, to assert the I/O select pulse, the output port address should be 82FF_H.

FIGURE 5.9

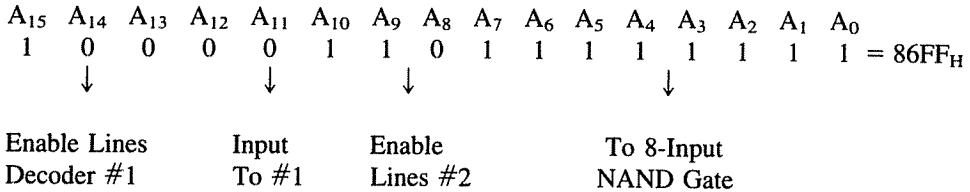
Logic Symbol: 74LS373 Octal Latch





5.63 Input Port and Its Address

The DIP switches are interfaced with the Z80 using the tri-state buffer 74LS244. The switches are tied high and are turned on by grounding as shown in Figure 5.8. The switch positions can be read by enabling the signal OE, which is asserted when both the output O₁ of decoder #1 and the control signal MEMRD go low. Therefore, to read the input port, the port address should be 86FF_H.



5.64 Instructions

To control the appliances 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: LD A, (86FFH)	;Read the switches
CPL	;Complement switch reading, convert "on" switch (logic 0) into logic 1 to turn on appliances
LD (82FF), A	;Send switch positions to output port and turn appliances on or off
JP READ	;Go back and read again

When this program is executed, the first instruction reads the bit pattern 1 0 1 1 0 1 1 1 (B7_H) at the input port 86FF_H 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 logic 1 is necessary to turn on solid state relays. The third instruction sends the complemented accumulator contents (0 1 0 0 1 0 0 0 = 48_H) to the output port 82FF_H. The 74LS373 latches the data byte 0 1 0 0 1 0 0 0 and turns on the space heater and Light 1. The last instruction, JP READ, takes the program back to the beginning and repeats the loop continuously in order to monitor the switches.

ADDITIONAL ILLUSTRATIVE EXAMPLES: INTERFACING SENSORS AND MOTORS

5.7

In previous examples, we illustrated the interfacing of I/O devices that were primarily binary devices (on/off). We now extend the concepts to interface analog devices such as temperature sensors and motors. In interfacing analog devices, the basic procedure remains similar to that of interfacing binary devices; the MPU identifies the device through a binary port address and enables it with an appropriate control signal. However, we need to find a way to detect and to convert the analog signal into the binary format and vice versa. The analog signal is generally handled in two ways: one is to detect the signal when it reaches a predetermined level, and the other is to convert it into binary format proportional to its magnitude. The predetermined level of the analog signal can be detected by using a comparator circuit, and the binary equivalent can be obtained by using an **A/D (Analog-to-Digital) data converter**. In this section, we will focus on interfacing circuits that can detect the predetermined level of analog signals and defer the discussion of interfacing data converters to Chapter 13.

Figure 5.10 shows the interfacing of a temperature sensor. This circuit is designed to detect (through an input port) whether the temperature has risen to 100°C, and at that temperature it turns on the dc motor of a water pump. The dc motor is interfaced with the MPU through an output port.

5.71 Hardware: Temperature Sensor LM135 and Comparator LM311

Figure 5.10 shows the LM135 used as a temperature sensor, and its output is connected as one of the inputs to the comparator LM311. The LM135 is an integrated circuit, designed to sense changes in temperature; its output voltage changes 10 mV/°C. It is rated to operate over a temperature range from -55°C to $+150^{\circ}\text{C}$, and the current range $400\mu\text{A}$ to 5 mA . At 25°C , the output of the sensor is typically 2.98 V, and it increases 10mV/C; therefore, at 100°C , it can reach 3.73 V (2.98 V + 750 mV).

The LM311 is a voltage comparator that can be operated from a +5 V power supply. The comparator compares two voltages at its input terminals, and if the difference between the two voltages is less than or equal to -10 mV , its output remains at the saturation voltage of about 0.75 V; otherwise, the output is near the power supply voltage.

The output of the sensor is connected to the positive terminal of the comparator, and its negative terminal is set to 3.73 V. At temperatures lower than 100°C the output voltage of the sensor is less than 3.73 V; thus, the comparator output remains at 0.75 V (logic 0). When the temperature reaches 100°C , the output of the sensor is 3.73 V, and the comparator output goes to +4.5 V (logic 1). The output of the comparator is connected to the tri-state buffer 74LS244, which serves as an input port to the MPU.

5.72 Interfacing Circuit for the Sensor

Figure 5.10 shows that the 74LS138 (3-to-8) decoder is used for address decoding. This decoding circuit is identical to the circuit shown in Figure 5.6; thus, the outputs of the decoder are asserted for port addresses ranging from 80_{H} to 87_{H} ("don't care" lines are

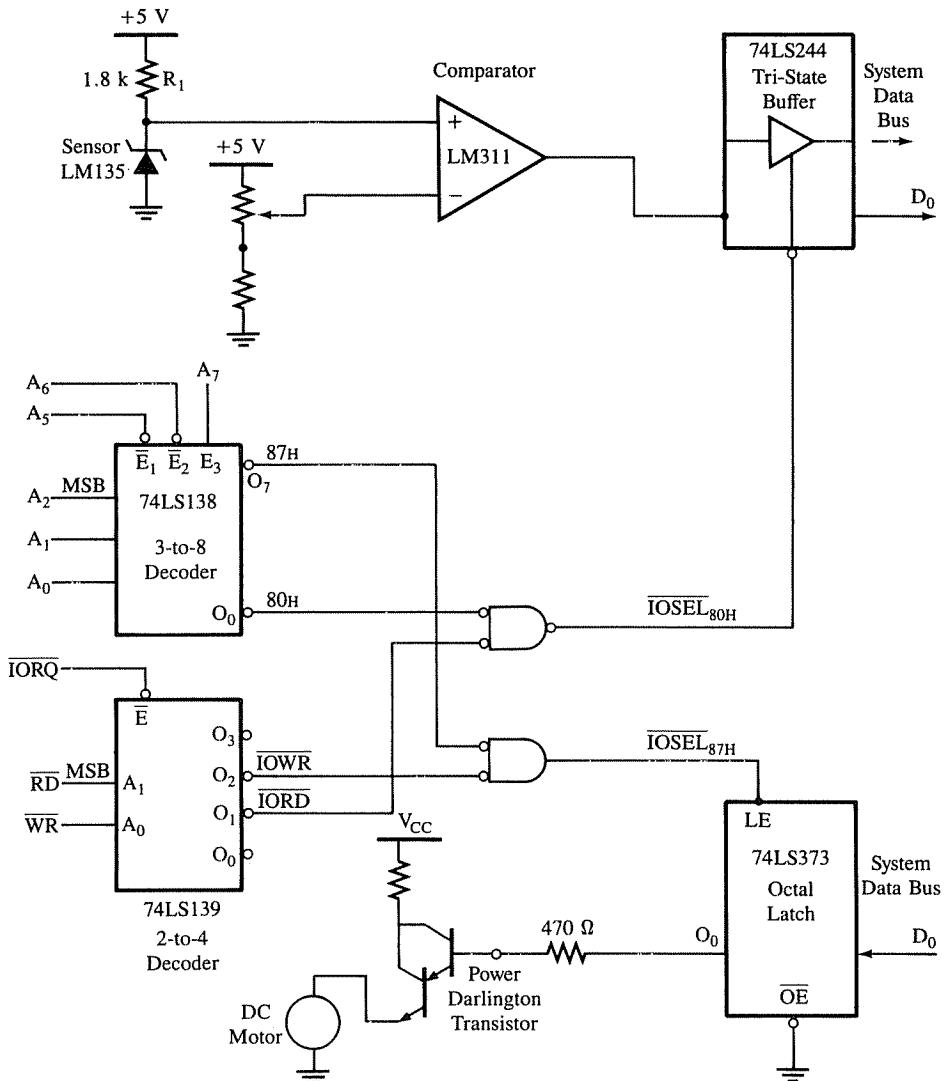


FIGURE 5.10
Interfacing Analog Signals

assumed to be at logic 0). The control signals \overline{IORD} (I/O Read) and \overline{IOWR} (I/O Write) are generated by using the 74LS139 (2-to-4) decoder; which is enabled by the \overline{IORQ} signal. When the MPU intends to read, it asserts the \overline{IORQ} and RD signals. The input of the 2-to-4 decoder becomes 0 1, and the output O_1 goes active low to assert the \overline{IORD} (I/O Read) control signal. The \overline{IORD} is logically ANDed with the decoded address 80_H to generate the

$\overline{\text{IOSEL}}_{80\text{H}}$ (I/O Select) signal, which enables the input buffer 74LS244 to read the output of the comparator. The output voltage of the comparator is connected to the data line D_0 through the buffer, and the MPU can monitor the temperature by monitoring the data line D_0 .

5.73 Interfacing Circuit for the DC Motor

The dc motor is interfaced with the MPU through the latch 74LS373; the output bit O_0 of the latch can drive the dc motor by turning on the transistor (Darlington pair). The logic level of bit O_0 of the latch is controlled by the data line D_0 . The port address of the latch (87H) is determined by the 3-to-8 decoder; the output line O_7 of the decoder is ANDed with the control signal $\overline{\text{IOWR}}$ to generate the $\overline{\text{IOSEL}}_{87\text{H}}$, which enables the latch 74LS373. When the temperature reaches 100°C , the MPU sends logic 1 to the latch (port 87H) to turn on the motor, and when the temperature is less than 100°C , the motor is turned off by the logic 0.

5.74 Instructions

```
START: IN A, (80H)      ;Read the output of the comparator  
                      AND 00000001B ;Save logic of  $D_0$  and eliminate  $D_1$  through  $D_7$   
                      OUT (87H), A    ;Turn on motor if  $D_0 = 1$  or turn off if  $D_0 = 0$   
                      JP START       ;Go back and read the output of the comparator
```

5.75 Program and Circuit Description

The first instruction IN A, (80H) enables the buffer 74LS244, reads the entire data bus D_7 – D_0 , and places the byte in the accumulator. However, we are interested in the logic level of only bit D_0 ; it has the output of the comparator. Therefore, the next instruction ANDs the contents of the accumulator with the byte 01H in order to eliminate bits D_1 – D_6 and save the logic level of bit D_0 . When the temperature exceeds 100°C , the output of the comparator is about $+5\text{ V}$, and the MPU reads logic 1 on the data line D_0 . When the temperature is lower than 100°C , the comparator output is about 0.7 V , and the MPU reads logic 0 on the data line D_0 . The next instruction OUT turns on the transistor if $D_0 = 1$ or turns off the transistor if $D_0 = 0$. When the transistor is on, it supplies the necessary current for the motor to run, and when the transistor is off, the motor is turned off. The last instruction JP takes the program back to the beginning and continuously monitors the changes in the output of the comparator.

5.76 Additional Sensors and Output Devices

Figure 5.10 illustrates one example of interfacing a sensor and driving a dc motor. We can extend the same concepts to other sensing and output devices. In Figure 5.10, we used only one data line D_0 to monitor the output of the comparator. We can connect additional sensors such as light detectors, level detectors, and smoke detectors to the remaining data

lines, and instructions can monitor all the sensors in a sequence. Similarly, we can connect output devices such as speakers, alarms, and lights by using solid state relays to the remaining output lines of the latch.

5.8

TROUBLESHOOTING I/O INTERFACING CIRCUITS

In the last several sections, we discussed the interfacing of I/O devices and instructions to test them. In Illustrative Example 1 (Figure 5.4), the test program includes two instructions that load the byte 97_H into the accumulator and output the byte to port 07_H. If we execute these instructions and no change is observed at the output port, we must implement the troubleshooting technique similar to that which we used for troubleshooting memory interfacing circuits in the last chapter. After checking the wiring and the pin connections, we can write a diagnostic routine and execute it in a continuous loop to generate a constant and identifiable signal, and then check various points in relation to that signal.

DIAGNOSTIC ROUTINE AND MACHINE CYCLES

We can use the same instructions for the diagnostic routine that we used in Illustrative Example 1; however, to generate a continuous signal, we need to add a Jump instruction, as shown.

Instruction	Bytes	T-states	Machine Cycles		
			M ₁	M ₂	M ₃
START: LD A, 97H	2	7 (4, 3)	Opcode Fetch	Memory Read	
OUT (07H), A	3	11 (4, 3, 4)	Opcode Fetch	Memory Read	I/O Write
JP START	3	10 (4, 3, 3)	Opcode Fetch	Memory Read	Memory Read

This loop has 28 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 2 MHz, the loop is executed in 14 μ s, and the WR signal, repeated every 14 μ s, can be observed on a scope. If we sync the scope on the WR pulse from the Z80, we can check the output of the 8-input NAND gate (IOADR), IOWR, and IOSEL signals; WR and IOSEL signals of a working circuit are shown in Figure 5.11.

When the Z80 asserts the WR signal, the port address 07H must be on the address bus A7–A0, and the output of the NAND gate must be low. Similarly, the IOWR must be low and the IOSEL must be high. Now if we check the data bus in relation to WR signal, one line at a time, we must read the data byte 97H.

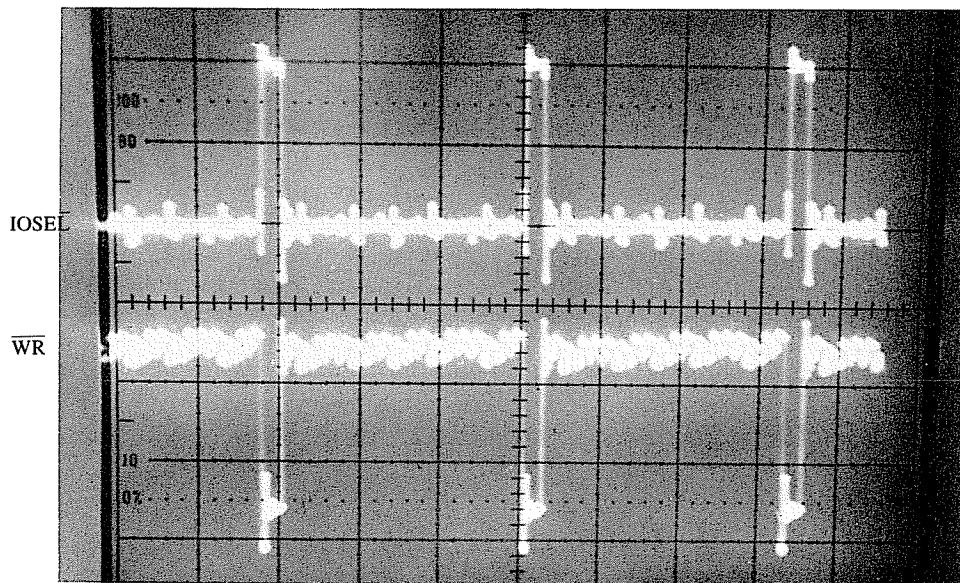


FIGURE 5.11
Timing Signals of Diagnostic Routine

SOME QUESTIONS AND ANSWERS

5.9

During the discussion of interfacing I/O devices, we focused on the basic concepts and avoided some details in order to simplify the presentation. We will now attempt to answer some of those questions.

1. *What are the other I/O instructions in the Z80 instruction set, and how do they differ from the I/O instructions discussed here?*

The Z80 instruction set includes six output instructions of which we discussed only one. The remaining five instructions perform various types of output functions: for example, output a byte from any of the registers or from a memory location, or output a block of memory. In these instructions, register C is used to specify the port address and register B can be used as a counter.

2. *What are the contents of the high-order bus ($A_{15}-A_8$) during the M_3 cycle of the IN/OUT instructions?*

The contents of the high-order bus during the M_3 cycle of the I/O instructions, illustrated in Sections 5.1 and 5.3, are generally irrelevant to the interfacing of I/O devices. For the I/O instructions discussed, the contents of the accumulator are placed on

the A₁₅–A₈ bus. However, in other I/O instructions where the contents of register C are used to specify a port address, the contents of register B are placed on the high-order bus.

3. Why is one Wait state automatically inserted when an I/O instruction is executed?

When an I/O instruction is being executed, the control signal IORQ is asserted during T₂ of the M₃ cycle. This does not leave sufficient time for the Z80 to sample the WAIT line. Therefore, a slow-responding I/O device would not be able to decode its address and activate the WAIT line if necessary. Adding one Wait cycle allows the device to activate the WAIT signal for additional Wait states.

4. In a memory-mapped I/O, what is the reason for not automatically inserting a Wait state?

In the Memory Read/Write cycles, the MREQ is asserted during T₁; therefore, there is sufficient time to sample the WAIT line during T₂ state.

5. In a memory-mapped I/O, how does the microprocessor differentiate between I/O and memory, and can an I/O device have the same address as a memory register?

In the memory-mapped I/O, the microprocessor cannot differentiate between an I/O device and memory; it treats an I/O device as if it is memory. Therefore, an I/O device 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.

6. 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?

In the Z80 microprocessor, the instruction decoder and the associated microprogram are designed to recognize the second byte as the low-order byte in a 3-byte instruction.

SUMMARY

In this chapter, we have examined the machine cycles of the OUT and IN instructions and derived the basic concepts for interfacing peripheral-mapped I/Os. Similarly, we examined the machine cycles of memory-related data transfer instructions and derived the basic concepts for interfacing memory-mapped I/Os. These concepts were illustrated with three examples of interfacing I/O devices and one example of interfacing an analog signal. The interfacing concepts can be summarized as follows.

Peripheral-Mapped I/O

- The OUT is a 2-byte instruction and copies (transfers or sends) data from the accumulator to the addressed port.
- When the Z80 executes the OUT instruction, in the third machine cycle it places the

output port address on the low-order bus, places data on the data bus, and asserts the control signals IORQ and WR.

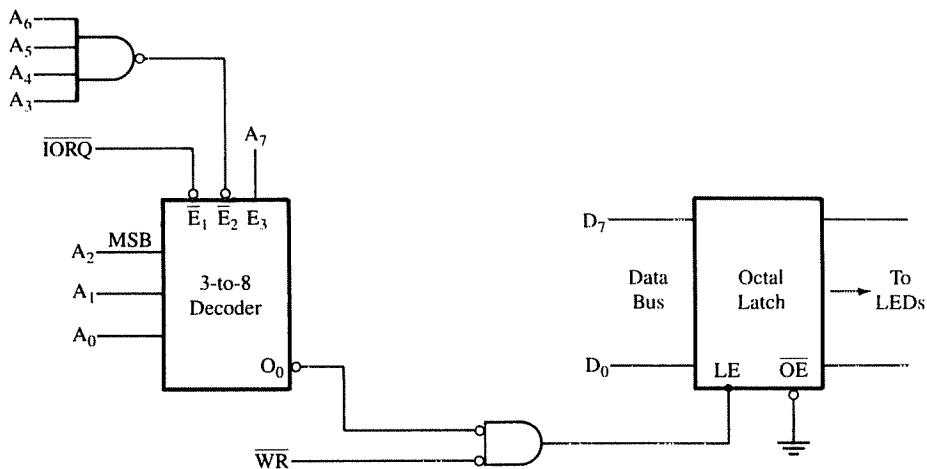
- A latch is generally used to interface output devices.
- The IN instruction is a two-byte instruction and copies (transfers or reads) data from an input port and places the data into the accumulator.
- When the Z80 executes the IN instruction, in the third machine cycle it places the input port address on the low-order bus, asserts the control signals IORQ and RD, and transfers data from the port to the accumulator.
- A tri-state buffer is generally used to interface input devices.
- To interface an output or an input device, the low-order address bus needs to be decoded to generate the device address pulse, which must be combined with control signals IORQ and RD (or WR) 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 signals MREQ and WR or RD to generate the I/O select pulse. Data are transferred by using this pulse to enable the I/O device.

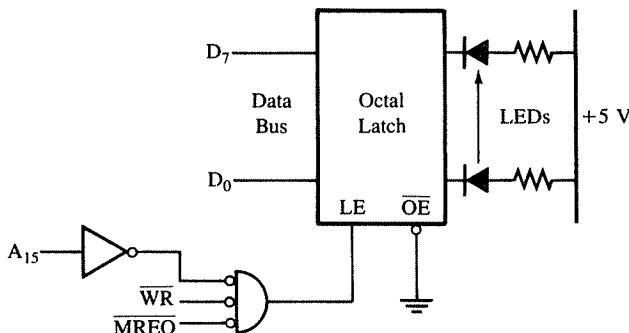
ASSIGNMENTS

1. Explain why the number of output ports in peripheral-mapped I/O is restricted to 256 ports?
2. In 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 Z80 differentiate between the ports?
4. Specify the two control signals required to latch data in an output port.
5. Specify the type of pulse 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. If the control signals WR and IORQ are asserted at the same time, can data be latched using only the control signal WR?
8. If the answer to the previous question is yes, what are potential problems with the interfacing circuit?
9. In Figure 5.4, explain why the LED cathodes rather than anodes are connected to the latch.
10. Specify the control signals required to enable an input port.
11. Explain why a latch is used for an output port, but a tri-state buffer can be used for an input port.
12. What are the control signals necessary in memory-mapped I/O?
13. Can the microprocessor differentiate whether it is reading from a memory-mapped input port or from memory?

**FIGURE 5.12**

Schematic for Assignments 15–16

14. In Figure 5.10, connect the output of the comparator to data line D₇ and also drive the transistor with bit D₇. Make the necessary changes in the instructions.
15. Identify the port address in Figure 5.12.
16. In Figure 5.12, if \overline{OE} is connected directly to the \overline{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.
17. In Figure 5.13, determine whether it is the memory-mapped or the peripheral-mapped I/O.
18. In Figure 5.13, what is the port address if all the “don’t care” address lines are assumed to be at logic 0?

**FIGURE 5.13**

Schematic for Assignments 17–18

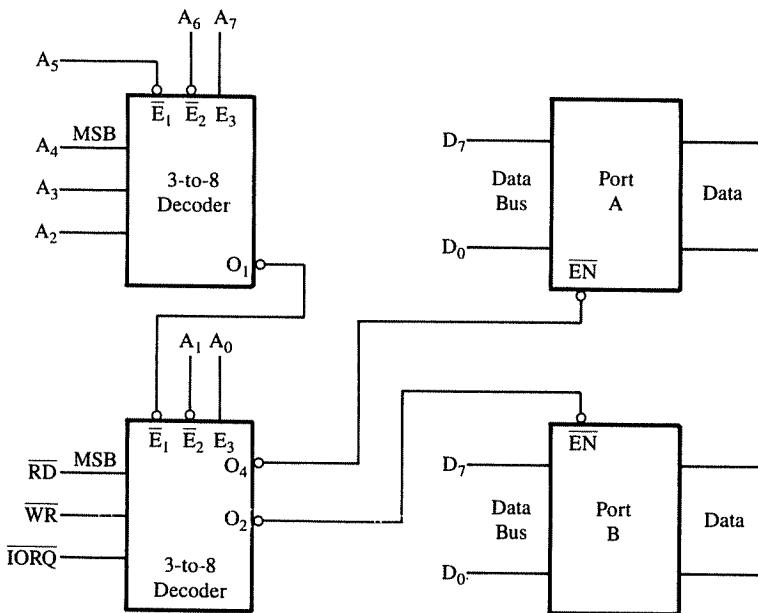


FIGURE 5.14
Schematic for Assignments 19–20

19. In Figure 5.14, are ports A and B input or output ports?
 20. In Figure 5.14, what are the addresses of ports A and B?
 21. In Figure 5.15, identify two output lines of decoder #2 that can be used as control signals and explain their functions. Explain why other output lines cannot be used as control signals.
 22. In Figure 5.15, specify the I/O addresses.
 23. In Figure 5.16, the decoder 74LS155 and an 8-input NAND gate are used to decode the address bus and generate the control signals. The decoder has two input lines A_1 and A_0 and four enable lines (pins 1, 2, 14, and 15). When pins 14 and 15 (active low) are enabled, the four output lines of the “b” group decode the input signal, and when pins 1 (active high) and 2 (active low) are enabled, the four output lines of “a” group decode the input signals. Identify the addresses that can assert the output lines of the decoder and specify their I/O functions.
 24. Sketch the waveforms of the M_1 cycles in the diagnostic routine (Section 5.8).
 25. Write a similar diagnostic routine to test the circuit in Figure 5.6.
 26. Is there a WR pulse in your diagnostic routine of 25? If the answer is no, what is the unique identifiable signal that can be used to sync the scope?

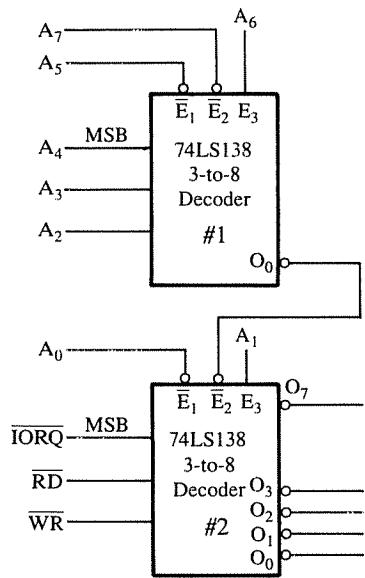


FIGURE 5.15
Schematic for Assignments 21–22

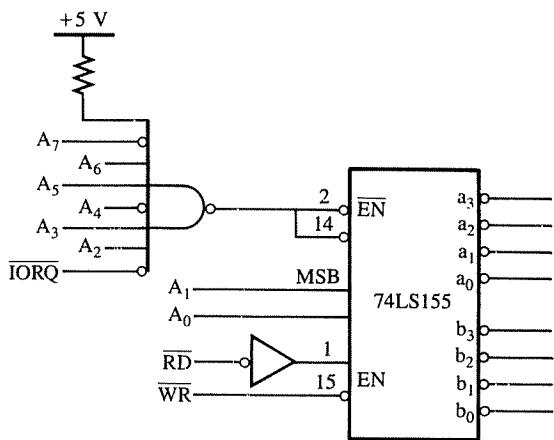


FIGURE 5.16
Schematic for Assignment 23

CHAPTER 6

Introduction to Z80 Assembly Language Programming

CHAPTER 7

Software Development Systems and Assemblers

CHAPTER 8

Introduction to Z80 Instructions and Programming Techniques

CHAPTER 9

Logic and Bit Manipulation Instructions

CHAPTER 10

Stack and Subroutines

CHAPTER 11

Application Programs and Software Design

Part II of this book is an introduction to Z80 assembly language programming. It explains commonly used instructions, elementary programming techniques and their applications, and the modular approach to software design.

The content is presented in a format similar to one for learning 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, one begins writing paragraphs that can convey ideas in a coherent fashion; then, by sequencing a few paragraphs, one can compose a letter. Chapters 6 to 11 are arranged in similar fashion—from simple instructions to applications.

Chapter 6 provides an overview of the Z80 instruction set and its capability, and Chapter 7 presents software development systems and Z80 assemblers. Chapters 8 and 9 are concerned primarily with the Z80 instructions that occur most frequently. The instructions are not introduced according to the six groups as classified in Chapter 6; instead, a few instructions that can perform simple tasks are selected from each group. Chapter 8

II

Assembly Language Programming: The Z80

includes the discussion of instructions from three groups—data copy, arithmetic and branch—and their various applications. Chapter 9 introduces logic and bit manipulation instructions and their applications. Chapter 10 introduces the concepts of subroutine and stack, which provide flexibility and variety for program design. Chapter 11 synthesizes the programming concepts presented in earlier chapters by illustrating application programs and demonstrates the modular approach to software design.

PREREQUISITES

The reader is expected to know the following topics:

- The Z80 architecture, especially the programming registers.
- The concepts related to memory and I/Os.
- Logic operations, and binary and hexadecimal arithmetic.

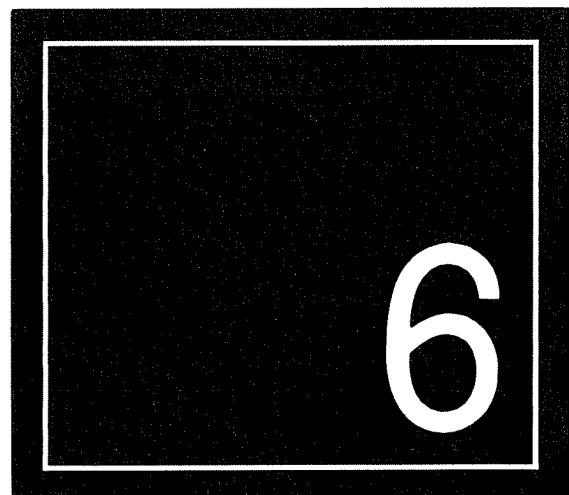
Introduction to Z80 Assembly Language Programming

An assembly language program is a set of instructions, written in the mnemonics of a given microprocessor, and in a sequence appropriate to a specified task. To write such programs, we should be familiar with the capabilities of the microprocessor and its instruction set. This chapter provides such an overview of the Z80 microprocessor.

The Z80 instruction set is classified into six categories, and each category is explained with examples. The chapter also discusses the instruction format and various addressing modes. Writing, assembling, and executing a program are illustrated by a simple problem of adding two Hex numbers. The **flowcharting** technique and symbols are discussed in the context of the illustrative program. The chapter concludes with a list of selected Z80 instructions.

OBJECTIVES

- Explain the terms operation code (opcode) and 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 six categories of the Z80 instruction set.
- Define and explain the term **addressing mode**.
- Write logical steps needed to solve a simple programming problem.
- Draw a **flowchart** from the logical steps of a given programming problem.
- Write mnemonics from the flowchart and convert the mnemonics into Hex code for a given programming problem.



6.1

OVERVIEW: Z80 INSTRUCTION SET

The instruction set of a microprocessor determines the capability of its operations, the power of its data manipulation, and the ease of programming it. For example, the Z80 instruction set includes an instruction that can copy contents from one block of memory locations to another. For most other 8-bit microprocessors, the programmer needs to write a program to perform the same function. Although it is necessary to have an overall view of the instruction set, our intent here is merely to acquaint you with the overall operations and capability of the Z80 microprocessor. As you progress through the chapters of Part II, you will be exposed to various instructions in more detail along with their applications.

The Z80 microprocessor has 158 instruction types; it includes all the instructions of the Intel 8080 microprocessor and all but two of the 8085. As discussed in Chapter 1, each instruction has two parts: one is the task to be performed (such as Load, Add, and Jump), called the operation code (opcode); and the second identifies the data to be operated on, called the operand. First, we will examine various formats of these instructions in terms of number of bytes and then their classification according to their function.

6.11 Instruction Format

An instruction is a command to the microprocessor to perform a given task on specified data. The size of Z80 instructions ranges from one to four bytes; thus, the number of memory registers (locations) required to write (or store) them varies. For example, to write a 3-byte instruction into memory requires three memory locations. Most opcodes (operation codes) are specified in one byte; however, some specialized opcodes require two bytes. The operand (or data) can be specified in the following ways: 8-bit data, 16-bit data, registers, register pairs, and memory addresses. The Z80 instruction set can be classified into four groups according to the length of an instruction: 1-byte to 4-byte instructions. Because the Z80 is an 8-bit microprocessor, the terms “byte” and “word” are used synonymously.

1-BYTE INSTRUCTIONS

In a 1-byte instruction, the opcode and the operand are included in the same byte as shown in the following examples.

Task	Opcode	Operand	Binary Code
Copy the contents of register B into the accumulator A.	LD	A, B	01 111 000 (78H)
Add the contents of register B to the contents of A.	ADD	A, B	10000 000 (80H)

These are 1-byte instructions performing two different tasks. In the first instruction, the opcode LD is specified by the first two bits (01) and the operand registers A and B are

specified by the remaining six bits. (The accumulator A is represented by 111 and register B by 000.) In the second instruction, the ADD is a 5-bit (10000) opcode, and the operand B is specified by the remaining three bits (000). These bits are associated with the internal microoperations of the microprocessor.

2-BYTE INSTRUCTIONS

In a 2-byte instruction, the first byte specifies the opcode and the second byte specifies the operand (with exceptions of some Z80 two-opcode instructions).

Task	Opcode	Operand	Binary Code
Load register B with the hexadecimal number 32. (Opcode for LD B is 06H)	LD	B, 32H*	0000 0110 (06H) Byte 1 0011 0010 (32H) Byte 2

3-BYTE INSTRUCTIONS

In a 3-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address or data in a reversed order: low-order byte followed by the high-order byte. For example:

Task	Opcode	Operand	Binary Code
Copy data from memory address 2080 _H into the accumulator.	LD	A, (2080H)	0011 1010 (3AH) Byte 1 1000 0000 (80H) Byte 2 0010 0000 (20H) Byte 3

4-BYTE INSTRUCTIONS

The descriptions given above for 2- and 3-byte instructions are valid for the instructions compatible with the 8080 instructions. The Z80 instruction set, however, includes numerous special-purpose instructions that are not compatible with the 8080 instruction set. An 8-bit microprocessor can have a maximum of 256 bit combinations; thus its instruction set is limited to 256 operation codes. The 8080 has already used 242 combinations for its 72 different instructions leaving only 14 combinations unused. However, the Z80 microprocessor needs many more combinations to use its additional registers (two index registers, alternate registers, interrupt vector, and refresh). This problem was resolved by designing 2-byte opcodes: unused opcodes combined with instruction opcodes. Z80 4-byte instructions are generally associated with index registers, as is the following example.

Task	Opcode	Operand	Binary Code
Load index register IX with 16-bit address 2000 _H .	LD	IX, 2000H	1101 1101 (DDH) Byte 1 0010 0001 (21H) Byte 2 0000 0000 (00H) Byte 3 0010 0000 (20H) Byte 4

Now we can discuss various instructions according to their functional classification.

*In an instruction, hexadecimal number is shown as the number followed by capital H, and in the text, the number is shown with the subscript _H.

6.12 Z80 Instruction Set

The Z80 instruction set can be divided into six major categories as follows:

1. Data Copy (Transfer) or Load Operations
2. Arithmetic Operations
3. Logic Operations
4. Bit Manipulation
5. Branch Operations
6. Machine Control Operations.

DATA COPY OR LOAD OPERATIONS

Copying data is one of the major functions the microprocessor needs to perform. The Z80 has numerous instructions that copy data from one location, called **source**, to another location, called **destination**, without modifying the contents of the source. In technical manuals, this function is quite often referred to as data transfer. However, since the term *data transfer* creates the impression that the contents of the source are destroyed, we prefer the term *data copy*. In this text, we have also used the terms Load, Read, and Write, all of which are data copy operations.

Figure 6.1 shows various categories of data copy operations. The Z80 has several instructions associated with each category, each of which, with its subdivisions, is listed below with examples of instructions.

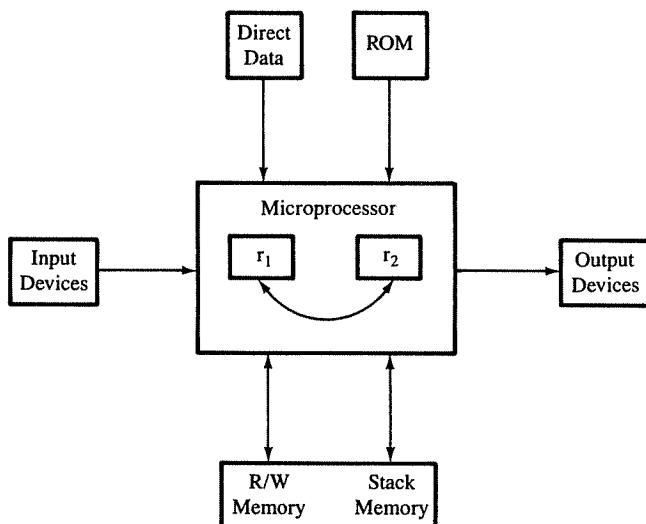


FIGURE 6.1
Types of Data Copy Operations

Data Copy Operations	Examples
1. From one register into another register.	Copy the contents of register B into the accumulator. LD A, B; LD means Load
2. (a) Specific data byte into a register or a memory location.	Load register B with the hexadecimal number 32. LD B, 32H
(b) Specific 16-bit data into a register pair.	Load register pair HL with hexadecimal number 2050. LD HL, 2050H
3. From a memory location into a register or vice versa.	Copy data from memory location 2080 _H into the accumulator. LD A, (2080H)
4. (a) From an input port into the accumulator.	Read data from input port 01 _H and copy into the accumulator. IN A, (01H)
(b) From the accumulator into an output port.	Write (send) the contents of the accumulator into port 07 _H . OUT (07H), A
5. From microprocessor registers into stack memory locations and vice versa.	Copy the contents of register pair BC into defined stack memory locations. PUSH BC
6. Exchange contents between registers. (This is a slightly different operation from data copy; this is a data exchange.)	Exchange the contents of general purpose registers (BC, DE, HL) with alternate registers. EXX

General characteristics of these data copy instructions can be listed as follows:

1. In data copy operations, the contents of the source are copied into the destination without affecting the contents of the source (except in Exchange instructions).
2. In an operand, the destination is specified first, followed by the source. For example, in the instruction LD A, B the source is register B and the destination is the accumulator. This may appear backward because the flow is generally assumed to be from left to right.
3. The memory and I/O addresses are enclosed in parentheses.
4. In some instructions, operand is implicit (for example, EXX).
5. These instructions do not affect flags.

ARITHMETIC OPERATIONS

The Z80 instruction set includes four types of arithmetic operations: addition, subtraction, increment/decrement, and 1's and 2's complement. In 8-bit arithmetic operations, the

accumulator is generally assumed to be one of the operands (with the exception of increment/decrement instructions).

- **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. The result of the addition is stored in the accumulator, and the flags are affected by the result. No two other 8-bit registers can be added directly; for example, the contents of register B cannot be added directly to the contents of register C.

Examples: Add the contents of register B to the contents of the accumulator. → ADD A, B

Add the byte 97_H to the contents of the accumulator. → ADD A, 97H

- **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. The subtraction is performed in 2's complement, and the result is stored in the accumulator. The result modifies the flags, and if the result is negative, it is expressed in 2's complement. The following mnemonics indicate that the accumulator is implicitly assumed as one of the operands.

Examples: Subtract the contents of register C from the contents of the accumulator. → SUB C

Subtract the byte 47_H from the contents of the accumulator. → SUB 47H

- **Increment/Decrement.** The 8-bit contents of a register (including the accumulator) or a memory location can be incremented or decremented by 1. Similarly, the 16-bit contents of a register pair (such as HL) can be incremented or decremented by 1. Unlike Add and Subtract, these operations can be performed in any of the registers. The instructions related to 8-bit contents affect flags (except Carry); on the other hand, instructions related to 16-bit contents do not affect any flags.

Examples: Increment the contents of register B. → INC B

Decrement the contents of register pair BC. → DEC BC

- **1's and 2's Complement.** The contents of the accumulator can be complemented (1's or 2's complement), and the result is stored in the accumulator. Some flags are affected by the result. These instructions assume that the operand is the accumulator.

Examples: Complement the contents of the accumulator (this is equivalent to 1's complement). → CPL

Subtract the contents of the accumulator from zero (this is equivalent to 2's complement). → NEG

LOGIC OPERATIONS

The instructions related to logic operations can be divided into three groups: logic functions (AND, OR, etc), bit rotations or shifts, and comparisons (less than, greater than, and equal to) of data bytes.

- **Logic Functions.** Any 8-bit number, the contents of a register, or the contents of a memory location can be ANDed, ORed, or Exclusive ORed with the contents of the accumulator. The result is stored in the accumulator, and the flags are affected by the result.

Examples: Logically AND the contents of register B with the → AND B
contents of the accumulator.

Exclusive OR the contents of register B with the → XOR B
contents of the accumulator.

- **Shift and Rotate.** Each bit in the accumulator, in the registers, or in memory can be shifted either left or right by one position.

Examples: Rotate the contents of the accumulator → RRA
right through Carry flag.

Rotate left the contents of register B. → RLC B

- **Compare.** Any 8-bit number, the contents of a register, or memory can be compared for equality, greater than, or less than with the contents of the accumulator. The result of the comparison is indicated by appropriate flags.

Examples: Compare the contents of register B with the → CP B
contents of the accumulator.

Compare the data byte 97_H with the contents → CP 97H
of the accumulator.

BIT MANIPULATION

The bit manipulation instructions can be classified into two groups: bit test and bit set/reset.

- **Bit Test**—Any one of the eight bits in a register, accumulator, or memory can be verified as 0 or 1, and the Z flag will be modified accordingly.

Example: Check bit D₇ in register B. → BIT 7, B

- **Bit Set/Reset**—Any one of the eight bits in a register, accumulator, or memory can be set or reset.

Examples: Set bit D₅ in the accumulator. → SET 5, A

Reset bit D₂ in register B. → RES 2, B

BRANCHING OPERATIONS

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

- **Jump.** The sequence of program execution can be altered either conditionally or unconditionally. When a conditional Jump instruction is used, the microprocessor checks the specified flag, and if the condition is true, the execution sequence is altered; otherwise, the next instruction is executed. The destination location to which the program should be directed can be specified directly or relative to the contents of the program counter. These instructions are critical to the decision making process in programming.

Examples: After an operation (such as an addition), → JP C, 2050H
if CY flag is set, jump to location 2050_H.

If Zero flag is not set, jump forward → JR NZ, 0FH
by 15 locations.

- **Call/Return.** These instructions change the sequence of a program by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions check for appropriate flags.

Examples: Go to subroutine located at 2050_H. → CALL 2050H
Go to Subroutine located at 2070_H → CALL Z, 2070H
if Z flag is set.

- **Restart.** These instructions are used to change the program sequence to one of eight restart locations on memory page 00. The instructions are generally used with interrupts.

Example: Call location 0028_H. → RST 28H

MACHINE CONTROL OPERATIONS

These instructions control microprocessor operations such as Halt and Interrupt.

Examples: Suspend execution of instruction. → HALT
Disable interrupts by resetting the → DI
Interrupt Enable flip-flops.

6.13 Review of Important Concepts

Our intent here is to give you an overall view of the instruction set and the capability of the Z80 microprocessor. The Z80 has 158 instructions with 694 opcodes. These numbers can be overwhelming and intimidating to a beginner. Fortunately, as you begin to use instructions, a logical pattern will begin to emerge. At this point, the important concepts to remember are as follows:

1. Each instruction has two parts: opcode and operand. The opcode specifies the task, and the operand specifies either data or where data are located.

2. Instructions can be classified into four groups according to their word length: one to four bytes.
3. In an instruction, when the data source and the destination are explicitly specified, the destination is shown first and the source second.
4. When an operand is a 16-bit address (or data), it is stored in memory in a reversed order: the low-order byte first, followed by the high-order byte.
5. Instructions are stored in memory in binary format; the microprocessor neither reads nor understands mnemonics or hexadecimal numbers.
6. The number of memory locations required to store an instruction is determined by the word length. For example, a 3-byte instruction would require three memory locations.

ADDRESSING MODES

6.2

The addressing mode is a way of specifying an operand or pointing to a data location. The Z80 microprocessor has ten addressing modes, as shown in Table 6.1. The first three are explained here as illustrations, and the others will be explained in later chapters.

In Section 6.12, we listed various categories of data copy operations. Data can be loaded directly into registers (or memory) or can be copied from registers and memory, including I/O ports. Here are the addressing modes of these data copy operations.

Addressing Modes	Examples						
1. Immediate	<p>In this mode, the byte following the opcode is the operand.</p> <p>Example: LD A, 32H → Load 32_H into the accumulator</p> <table style="margin-left: 40px;"> <tr> <td>3E</td> <td>Opcode</td> </tr> <tr> <td>32</td> <td>Operand</td> </tr> </table>	3E	Opcode	32	Operand		
3E	Opcode						
32	Operand						
2. Immediate Extended	<p>In this mode, two bytes following the opcode constitute the operand; the second byte is low-order and the third byte is high-order.</p> <p>Example: LD HL, 2050H → Load 2050_H into the HL pair.</p> <table style="margin-left: 40px;"> <tr> <td>21</td> <td>Opcode</td> </tr> <tr> <td>50</td> <td>Operand: Low-order</td> </tr> <tr> <td>20</td> <td>Operand: High-order</td> </tr> </table>	21	Opcode	50	Operand: Low-order	20	Operand: High-order
21	Opcode						
50	Operand: Low-order						
20	Operand: High-order						
3. Register	<p>In this mode, a data byte is copied from one register to another register, and both registers are specified in the instruction.</p> <p>Example: LD A, B → Copy the contents of register B into A.</p>						

At this point, you are not familiar with the instructions set; therefore, you should avoid the details of the addressing modes given in Table 6.1. As we begin to use various

TABLE 6.1

Z80 Addressing Modes	Explanation	Example
1. Immediate :	The byte following the opcode is the operand. This mode is used to load 8-bit data into a register. Load 97 _H into register B	LD B, 97H
2. Immediate : Extended	The two bytes following the opcode are the operands. This mode is used to load 16-bit data or address into a register pair. Load 8045 _H into register pair BC	LD BC, 8045H
3. Register :	The operand register is included as a part of the opcode. This mode is used to copy data from one Z80 register into another register. Copy data from register A into B	LD B, A
4. Implied :	This refers to operations in which the opcode implies one or more Z80 registers as containing the operands. For example, instructions for logic operations imply that the accumulator is one of the operands and that the result is stored in the accumulator. Logically AND register B with A	AND B
5. Register Indirect	This mode is used to copy data between the MPU and memory; the 16-bit contents in a register pair are used as a memory pointer. Copy the contents of memory location 2060 _H into register B. Register HL contains the address 2060 _H .	LD, B, (HL)
6. Extended :	The two bytes following the opcode specify the jump location. Jump to location 2080 _H .	JP 2080H.
7. Relative :	In this mode, the second byte specifies the displacement value in a signed 2's complement for a jump location. Jump forward 20 locations from the address of the next instruction.	JR 14H
8. Indexed :	In this mode, the byte following the opcode specifies a displacement value that is added to one of the index registers to form a memory pointer. The index register IX contains 2060 _H ; increment the contents of memory location 2070 _H .	INC (IX + 10H)
9. Bit :	This mode is used for bit operation (manipulation). In this mode, instruction specifies a bit from a register or a memory location using one of the three addressing modes (register, register indirect, or indexed). Set bit D ₇ in register B.	SET 7, B
10. Page Zero :	The instruction set includes eight restart (one-byte call) instructions on memory page zero. In this mode, the memory location can be specified by using the low-order byte, and the high-order byte is assumed to be 00 _H . Call restart memory location 0028 _H .	RST 28H

instructions in following chapters, we will discuss the appropriate addressing modes. As you become more familiar with the instruction set, you will be able to choose an appropriate addressing mode for a given task.

HOW TO WRITE, ASSEMBLE AND EXECUTE A SIMPLE ASSEMBLY LANGUAGE PROGRAM

6.3

An assembly language program is a sequence of instructions written in mnemonics to perform a specific task. These instructions are selected from the instruction set of the microprocessor being used. To write a program, we need to divide a given problem into small steps and translate these steps into the operations the Z80 can perform. For example, the Z80 does not have an instruction that can multiply two binary numbers, but it can add. Therefore, the multiplication problem can be written as a series of additions.

After writing the instructions in mnemonics, you should translate them into binary machine code; this process of translation is called **assembling the code**. Quite often, this process involves intermediate steps, such as translating mnemonics into Hex code and then into binary code. The code assembly can be done manually, as described in this chapter, or using an assembler (a program that translates mnemonics into machine code), as described in the next chapter.

To execute a program, the binary code should be entered and stored in the R/W memory of a microcomputer so that the microprocessor can read and execute the binary instructions written in memory. In a single-board microcomputer the instructions are, generally, entered using a Hex keyboard. This is one of the reasons why we translate mnemonics into Hex code as an intermediate step rather than into binary code directly. When the Hex code is entered, the keyboard program, residing in the microcomputer system, translates the Hex code into binary code. The steps required to write, assemble, and execute a program are illustrated in the next section.

6.31 Illustrative Program: Adding Two Hexadecimal Numbers

PROBLEM STATEMENT

Write instructions to load the two hexadecimal numbers 32_H and $A2_H$ into registers B and C 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 in order 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 into 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 6.2 shows such a flowchart representing those steps. This is a simple flowchart, and the steps are self-explanatory. We will discuss flowcharting in the next section.

ASSEMBLY LANGUAGE PROGRAM

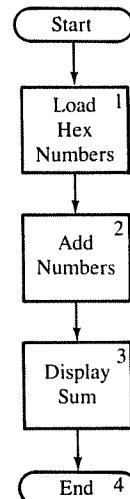
To write an assembly language program, we need to translate the blocks shown in the flowchart into Z80 operations and then into mnemonics. By examining the blocks, we can classify them into three types of Z80 operations: Blocks 1 and 3 are copy operations; Block 2 is an arithmetic operation; and Block 4 is a machine control operation. The translation of each block into mnemonics with comments is shown below.

Block 1:	LD B, 32H	;Load register B with 32H.
	LD C, A2H	;Load register C with A2H.
	LD A, C	;Copy contents of C into accumulator to perform addition. B and C cannot be added directly.
Block 2:	ADD A, B	;Add two bytes and save the sum in A.
Block 3:	OUT (01H), A	;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 Z80 instruction set; this is called either manual or hand assembly. The Hex code is as follows:

FIGURE 6.2
Flowchart for Adding Two Numbers



Mnemonics	Hex Code	
LD B, 32H	06 32	2-byte instruction
LD C, A2H	0E A2	2-byte instruction
LD A, C	79	1-byte instruction
ADD A, B	80	1-byte instruction
OUT (01H), A	D3 01	2-byte instruction
HALT	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 map and the output port address. Let us assume that R/W memory ranges from 2000_H to $20FF_H$, and the system has an LED output port with the address 01_H . To enter the program, the following steps are necessary:

1. Reset the system by pushing the RESET key.
2. Using Hex keys, enter the first memory address at which the program should be stored. Let us assume it is 2000_H .
3. Enter each machine code by pushing Hex keys. For example, to enter the first machine code push 0, 6, and STORE keys. (The STORE key may be labelled differently in different systems.) When you push the STORE key, the program will store the machine code in memory location 2000_H and upgrade the memory address to 2001_H .
4. Repeat Step 3 until the last machine code 76_H .
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 the Read-Only Memory (or EPROM) of the microcomputer 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 6.3.

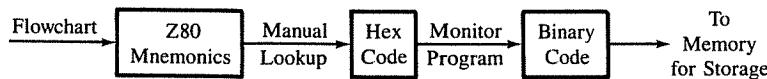


FIGURE 6.3
Assembling the Code

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

Mnemonics	Hex Code	Memory Contents	Memory Address
LD B, 32H	06	0 0 0 0 0 1 1 0	2000
	32	0 0 1 1 0 0 1 0	2001
LD C, A2H	0E	0 0 0 0 1 1 1 0	2002
	A2	1 0 1 0 0 0 1 0	2003
LD A, C	79	0 1 1 1 1 0 0 1	2004
ADD A, B	80	1 0 0 0 0 0 0 0	2005
OUT (01H), A	D3	1 1 0 1 0 0 1 1	2006
	01	0 0 0 0 0 0 0 1	2007
HALT	76	0 1 1 1 0 1 1 0	2008

This program has nine machine codes and will require nine memory locations to store the program. The critical concept to be emphasized here is that the microprocessor can understand and execute only the binary instructions (or data); everything else (mnemonics, Hex code, comments) are for the convenience of those who write and use the assembly language programs.

EXECUTING THE PROGRAM

To execute the program, we need to tell the microprocessor where the program begins by entering the memory address 2000_H . Then, 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, Z80 loads 2000_H into 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 2000_H and 2001_H and execute the instruction LD B, 32H; thus it will load 32_H into register B. It continues to execute instructions until it fetches the HALT instruction.

6.32 Program Documentation Or Writing Format

Program documentation is an important aspect of writing programs. The documentation should be able to communicate what the program does and the logic underlying the program, so that it can be debugged and modified if necessary. For our illustrative program, a writing format based on assembler files (discussed in the next chapter) is shown here.

Memory Address	Hex Code	Label	Instruction (Opcode)	(Operand)	Comments
2000	06	START:	LD	B, 32H	; Load first byte
2001	32				
2002	0E		LD	C, A2H	; Load second byte to be added
2003	A2				
2004	79		LD	A, C	; Copy one of the bytes into A
2005	80		ADD	A, B	; Add two bytes
2006	D3		OUT	(01H), A	; Display the result
2007	01				
2008	76		HALT		; End

This writing format has five columns: Memory Address, Hex Code, Label, Instruction (Opcode and Operand), and Comments. Each column is described below in the context of a single-board computer.

Memory Addresses These are 16-bit addresses of the system's R/W memory in which the binary code of the user program is stored. In the illustration, we assumed that the R/W memory in our system begins at the address 2000_H , and we chose to store the program starting at the location 2000_H ; we could have chosen any other available memory block to store our program.

Hex Codes These are the hexadecimal codes of the Z80 mnemonics we looked up in the instruction set; they were entered in memory using the Hex keyboard of the single-board microcomputer system. The key monitor program of the system translates these Hex codes and stores the binary equivalents in the proper memory locations.

Labels They are used to identify a memory location. The program has one label: START. This label is used for documentation; it indicates the beginning of the program. The labels are used to identify memory locations and will be especially useful for Jump instructions when we use assemblers to write programs (discussed in the next chapter).

Instructions These are the Z80 mnemonics representing the microprocessor operations. Each instruction is divided into two parts: opcode and operand.

Comments The comments are written as a part of the proper documentation of a program to explain or elaborate the purpose of the instruction used. They thus play a critical role in the user's understanding of the logic behind a program. Because the illustrative program is very simple, the comments shown are either redundant or trivial, but in general comments should not merely describe the meaning of mnemonics.

FLOWCHARTING

6.4

A flowchart is a graphic representation of the logic and sequence of tasks to be performed. A flowchart should assist in clarifying one's thinking process and communicate the programmer's approach and logic in writing the program.

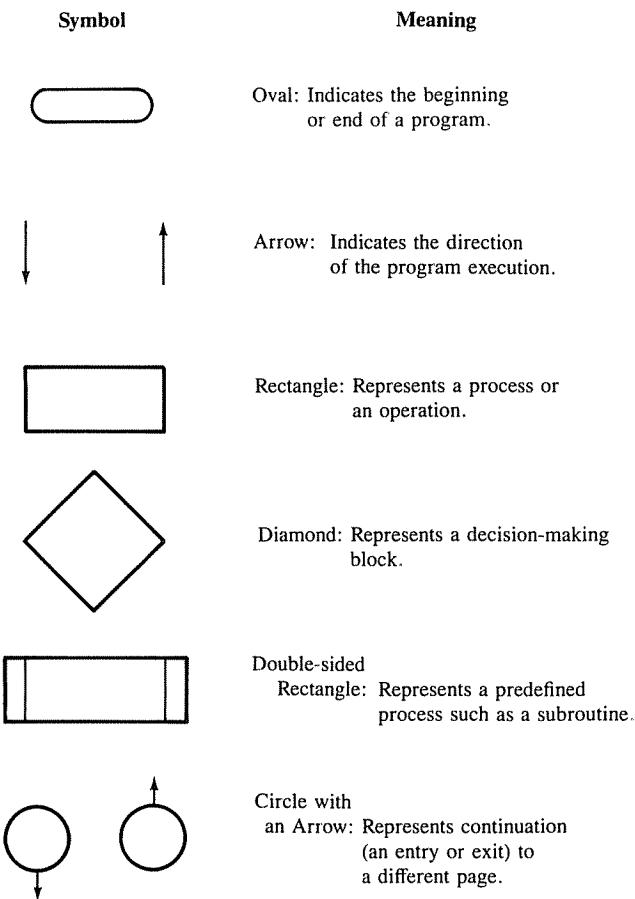


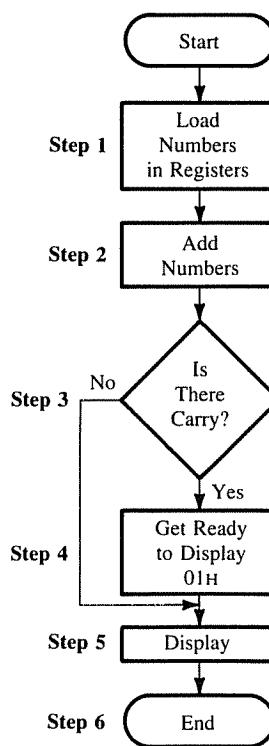
FIGURE 6.4
Flowcharting Symbols

Flowcharting is an art; how much detail it should include requires a subjective decision. At one level, the flowchart includes only the functions to be performed without any reference to a particular microprocessor; at another level the functions of registers being used are specified in detail. However, it should not duplicate the instructions in the program in a graphic format; this would defeat the whole purpose of drawing the flowchart. It should simply represent a logical approach and sequence of steps in solving the problem.

The six symbols commonly used in flowcharting are shown in Figure 6.4. We have already used three symbols in Figure 6.2. The fourth symbol, shown by the diamond shape, represents the decision-making block. It is used when data conditions need to be checked and the program sequence has to be altered. This symbol is illustrated in Figure

FIGURE 6.5

Flowchart: Adding Two Hex Numbers and Checking Carry



6.5. The fifth symbol, a double-sided rectangle, represents a predetermined process such as a subroutine (discussed in Chapter 10). The last symbol, a circle with an arrow, is used to show continuation of the flowchart to a different column or to a different page.

Draw a flowchart to represent the following problem. Load two Hex bytes into Z80 registers and add the bytes. If the sum is larger than 8 bits, display 01_H as the overload condition at port PORT7; otherwise, display the sum at the output port.

Example
6.1

The problem can be divided into the following steps.

Solution

1. Load bytes into Z80 registers.
2. Add the bytes.
3. Check the sum.
4. If the sum > FF_H, display 01_H at the output port.
5. If the sum < FF_H, display the sum at the output port.

The steps listed in Example 6.1 and the sequence can be represented by the flowchart shown in Figure 6.5. The first two blocks can be easily understood. The third block, shown by the diamond shape, is a decision-making block. In this block the result is checked by examining the CY flag, and the program execution is altered accordingly. If the CY flag is set, the result is larger than FF_H and the program execution goes to the next block. It loads 01_H and displays it at the output port. If the answer to the question in the decision block is "No," the sum is less than FF. The program sequence is then altered; it bypasses Block 4 and displays the sum at the output port.

An interesting question is: Can we interchange the answers "Yes" and "No" at the decision-making block? That is, can the program sequence be changed if CY is set? This is given as a problem at the end of the chapter; you may find that the resulting flowchart will have two end points.

6.5

LIST OF SELECTED Z80 INSTRUCTIONS

The Z80 instruction set includes 158 instructions resulting in 694 machine codes. The following list is a representative sample of each group described in Section 6.1. The purpose of the list is to show you the overall capability of the Z80 and some logical patterns in its instruction. You should not study these instructions in detail; instead, you should search for logical patterns. Once you recognize logical patterns, you will be able to recognize the function of an instruction even if you have not seen it before.

Most instructions are compatible with the 8080 instruction set, with a few exceptions. Notations used in the description of the instructions include

r = Z80 8-bit Register	rp = Register Pair
r_s = Register Source	rx = Index Registers
r_d = Register Destination	d = Displacement Byte
m = Memory	b = Bit
() = contents of 16-bit Memory Address or 8-bit I/O Address	

1. Data Copy (Load) Instructions

Mnemonics	Bytes	Tasks
<i>Data (8 bits and 16 bits) copy or load in registers</i>		
LD r_d, r_s	1	Copy data from source register r_s into destination register r_d .
LD r , 8-bit	2	Load 8-bit into a register.
LD rp , 16-bit	3	Load 16-bit into register pair.
LD rx , 16-bit	4	Load 16-bit data into index register.

Data copy between registers and memory

LD A, (16-bit)	3	Load accumulator from memory; the address is specified by 16-bit operand.
LD (16-bit), A	3	Load memory from accumulator; the memory address is specified by 16-bit operand.
LD A, (rp)	1	Load accumulator from memory; the memory address is specified by contents of register pair.
LD (rp), A	1	Load memory from accumulator; the memory address is given by the contents of register pair.
LD r, (HL)	1	Load register from memory; the address is specified by 16-bit contents in HL.
LD (HL), r	1	Load memory from register; the address is specified by 16-bit contents in HL.
LD r, (rx + d)	3	Copy memory contents into register r; the memory address is obtained by adding the contents of index register and the displacement byte d.
LD (rx + d), r	3	Copy register contents into memory address shown by index register and the displacement (rx + d)

2. Arithmetic Instructions*

ADD A, r	1	Add register contents to accumulator.
ADD A, 8-bit	2	Add 8-bit data to accumulator.
ADD A, (HL)	1	Add memory contents to accumulator; the memory address is specified by the contents in HL.
SUB r	1	Subtract contents of register from accumulator.
SUB 8-bit	2	Subtract 8-bit data from accumulator.
SUB (HL)	1	Subtract memory contents from accumulator; the memory address is specified by the contents of HL.
INC r	1	Increment the contents of a register.
INC (HL)	1	Increment the contents of memory; the memory address is specified by the contents of HL.
INC rp	1	Increment 16-bit contents in a register pair.
DEC r	1	Decrement the contents of a register.
DEC (HL)	1	Decrement the contents of memory; the memory address is specified by the contents of HL.
DEC rp	1	Decrement 16-bit contents in a register pair.

*Instructions used for 16-bit addition and subtraction are not shown here.

3. Logic Instructions*

AND r	1	Logically AND the contents of a register with the accumulator.
AND 8-bit	2	Logically AND 8-bit data with accumulator.
AND (HL)	1	Logically AND the contents of memory with accumulator; the memory address is specified by the contents of HL.
CP r	1	Compare the contents of register with accumulator for less than, equal to, or greater than.
CP 8-bit	1	Compare 8-bit data with accumulator for less than, equal to, or greater than.
CP (HL)	1	Compare the contents of memory with accumulator for less than, equal to, or greater than. The memory address is specified by the contents of HL.

4. Bit Rotation

RLCA	1	Rotate each bit in the accumulator to the left position.
RLA	1	Rotate each bit in the accumulator including the carry C to the left position.
RRCA	1	Rotate each bit in the accumulator to the right position.
RRA	1	Rotate each bit in the accumulator including the carry C to the right position.

5. Branch Instructions†

JP 16-bit	3	Change the program sequence (Jump) to memory location specified by the 16-bit address.
JP Z, 16-bit	3	Change the program sequence (Jump) to memory location specified by the 16-bit address if the Zero (Z) flag is set.
JP NZ, 16-bit	3	Change the program sequence (Jump) to memory location specified by the 16-bit address if the Zero (Z) flag is reset.
JP C, 16-bit	3	Change the program sequence (Jump) to memory location specified by the 16-bit address if the Carry (C) flag is set.
JP NC, 16-bit	3	Change the program sequence (Jump) to memory location specified by the 16-bit address if the Carry (C) flag is reset.
CALL 16-bit	3	Change the program sequence to the location of the subroutine.
RET	1	Return to the calling program after completing the subroutine sequence.

*The Z80 instruction set includes similar instructions for logically ORing and Exclusive ORing with mnemonics OR and XOR respectively.

†The Z80 set also includes conditional Call and Return instructions.

6. Machine Control Instructions

HALT	1	Suspend execution and wait.
NOP	1	Do not perform any operation.

7. Bit Rotation*

RLC r	2	Rotate each bit in register r to the left.
RL r	2	Rotate each bit in register r to the left, including Carry flag.
SLA r	2	Shift each bit in register r to the left.

8. Bit Manipulation†

BIT b, r	2	Test bit b in register r, affecting the Z flag.
SET b, r	2	Set bit b in register r. ("b" represents bit position 0 to 7)
RES b, r	2	Reset bit b in register r.

9. Z80 Special (Conditional) Repetitive Instructions.

The Z80 instruction set includes several instructions that are automatically repeated until a specified register becomes zero. These instructions are quite efficient in dealing with block transfer or counter applications. Some of these instructions are as follows:

CPDR	2	Compare memory contents specified by HL with the accumulator. Increment HL, decrement BC, and repeat until BC = 0. or A = contents of memory specified by HL.
DJNZ d	2	Decrement B, and if B ≠ 0, jump to memory address obtained by adding displacement byte to the program counter.
INDR	2	Read input port indicated by the C register, and store the byte in memory specified by HL register. Decrement B and HL, and continue until B = 0.
OTDR	2	Output the contents of memory specified by HL to port indicated by the C register. Decrement B and HL, and continue until B = 0.

SUMMARY

This chapter has provided an overview of the Z80 instruction set and the capability of the Z80 microprocessor. The important concepts and topics discussed in this chapter can be summarized as follows:

*The Z80 set includes similar instructions to shift right as well as to rotate bits in any memory location.

†Similarly, any bit in a memory location can be set or reset.

- The Z80 microprocessor operations are classified into six major groups: data copy (load), arithmetic, logic, bit manipulation, 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 8- or 16-bit data, an address, register, register pair, or it can be implicit.
- The method of specifying an operand is called the addressing mode.
- The instruction set is classified into four groups according to the word size: 1-, 2-, 3-, and 4-byte instructions.
- To write a simple assembly language program, the problem should be divided into small steps in terms of microprocessor operations, and these steps should be translated into Z80 mnemonics. Then, the Hex code is assembled by looking up the code in the instruction list; this is called either hand or manual assembly.
- To enter a program in memory of a single-board microcomputer, Hex keys are used to enter the code, which is converted into binary code by the Key Monitor program of the system and stored in R/W memory. This binary code can then be read and executed by the microprocessor.

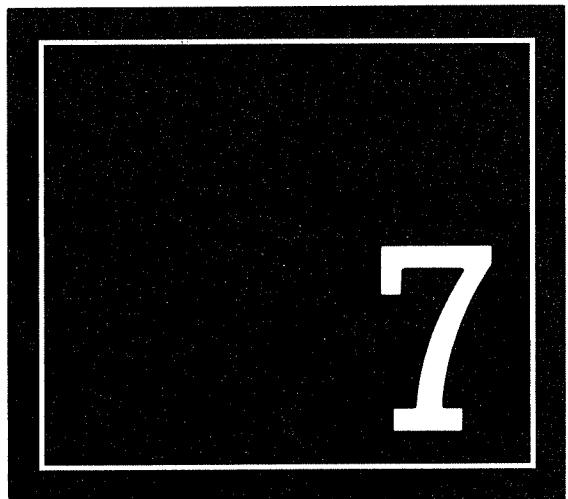
ASSIGNMENTS

1. List the six types of operations the Z80 performs.
2. Define opcode and operand, and specify the opcode and the operand in the instruction LD A, B.
3. Explain the instruction LD A, B. Specify the data source and destination.
4. If the instruction LD A, B is stored in memory location 2005_H, what are the contents of the memory register?
5. Explain the instruction SUB H. List the operand implicit in the instruction.
6. Write mnemonics to load F8_H into register C and show the Hex codes with the memory address starting at 1800_H.
7. Write logical steps to load the following three Hex numbers (2F, 47, and 7A) into Z80 registers B, C, and D, respectively. Add the numbers and save the sum in register H.
8. Translate the steps in the previous question into Z80 assembly language.
9. Redraw the flowchart in Figure 6.5 by interchanging the answers of the decision block. For example, the program sequence will be altered if the answer is "Yes." (Hint: The flowchart can have two End statements.)
10. Draw a flowchart to represent the following problem. Load two numbers into Z80 registers, and subtract the second number from the first number. If the result generates a borrow, display FF_H at the output port of the system; otherwise display the second number.

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. In the previous chapter, we discussed how to write a simple assembly language program and translate its mnemonics into Hex code manually. In this chapter, we will develop assembly language programs with the help of four other programs: Editor, Assembler, Linking Loader, and Debugger. These programs enable the user to write programs in mnemonics, translate mnemonics into Hex and binary code, and debug the code. All the activities of the computer—hardware and software—are directed by another program, called the **operating system**.

This chapter describes a microprocessor-based software development system, its hardware, and related programs. It also describes such widely used operating systems as CP/M and MS-DOS, and illustrates the use of the 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 accessed from the disk.
- Define the operating system of a microcomputer, and explain its function.
- Explain the functions of these programs: Editor, Assembler, Linking Loader, and Debugger.
- List the advantages of the assembler over manual assembly.
- List the assembler directives, and explain their functions.
- Write assembly language programs with appropriate directives.

7.1 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 (64K or higher), disk storage, and a video terminal with a typewriter-like keyboard. The keyboard enables the user to write programs in alphanumeric (alphabet and number) characters, which are translated into ASCII (American Standard Code for Information Interchange) binary code; the keyboard (or the terminal) is known as ASCII keyboard (or terminal). 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 **Z80 assembly language**.

Conceptually, this type of microcomputer is similar to a single-board microcomputer except that it has additional features that can assist in developing large programs. Programs are accessed and stored under a file name (title), and they are written by using such other programs as text editors and assemblers. The system (I/Os, files, programs, etc.) is managed by a program called the operating system. The various hardware and software features of a typical software development system are described in the next sections.

7.11 System Hardware and Storage Memory

Figure 7.1 shows a typical software development system; it includes an ASCII keyboard, a CRT terminal, an MPU board with at least 64K R/W memory and disk controllers, and two 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 that are responsible for reading and writing data on the disk. Three types of floppy disks are in use: 8-inch, 5 $\frac{1}{4}$ -inch, and 3 $\frac{1}{2}$ -inch; at present, systems with 5 $\frac{1}{4}$ -inch disks seem to be the most commonly used. A 5 $\frac{1}{4}$ -inch single-density disk can store about 90K bytes of data; the storage capacity can be doubled by using double-density disks, and quadrupled (to 360K) by using both sides of the disks. Recently, manufacturers have improved on storage density, and now 5 $\frac{1}{4}$ -inch high-density disks with 1.2 megabyte storage capacity are available.

FIGURE 7.1

A Typical Software Development System: AT&T PC 6300 Plus

SOURCE: Photograph Courtesy of AT&T

**FLOPPY DISK**

A **floppy disk**—Figure 7.2 (a)—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, each track divided into sectors, as shown in Figure 7.2(b). The large hole in the center of the disk is locked by the disk drive when it spins the disk. The small hole shown in Figure 7.2(a) is known as the indexing hole. The disk drive uses this hole as a reference to count the sectors. The oblong cutout, called the head slot, is the reading/recording segment; this is the only segment of the surface that comes in contact with the R/W head. At the edge of the disk, near the head slot, is a notch called the Write Protect notch. In the 5 1/4-inch disk, if this notch is covered, data cannot be written on the disk; the disk is then “Write Protected.”

Floppy disks are further classified as either soft-sectored or hard-sectored. The disk shown in Figure 7.2(b) is a soft-sectored disk; it has one hole as a reference to the first sector, and the other sectors are formatted by using software. In the hard-sectored disk, now almost obsolete, each sector is identified with a separate hole.

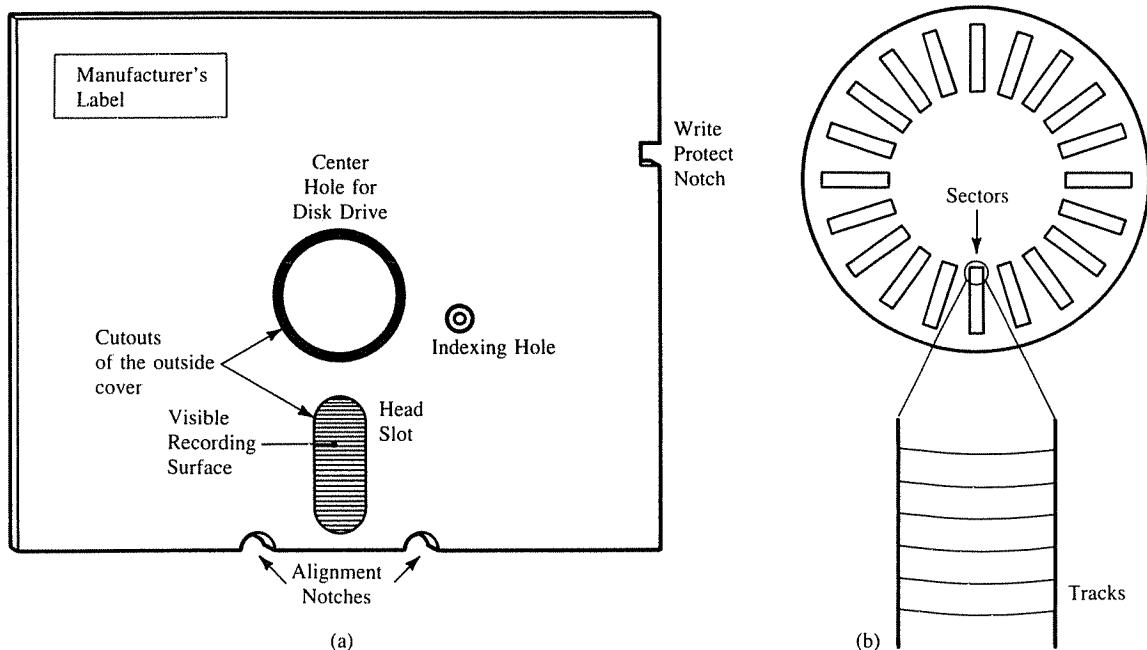


FIGURE 7.2

(a) A Typical 5 1/4-Inch Floppy Disk and (b) Its Sectors and Tracks

Each sector and track is assigned a binary address. The MPU can access any information on the disk with the sector and the track addresses; however, the access is semi-random. To go from one track to another, the access is random. Once the track is found, the system waits for the index hole and then locates the sector serially by counting the sectors. Once data bytes are located, they are transferred to the system's R/W memory. These data transfer functions between a floppy disk and the system are performed by the **disk controller** and controlled by the **operating system**, also known as the Disk Operating System (DOS), described in Section 7.12.

HARD DISK

Another type of storage memory used with computers is called a **hard disk** or **Winchester 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. While it is highly precise and reliable, the hard disk requires sophisticated controller circuitry; it is thus relatively expensive. However, its storage capacity is quite large. Hard disks are available in various sizes; 3 1/2-inch, 5 1/4-inch, 8-inch, and 14-inch. Storage capacity can range from several megabytes to several gigabytes.

7.12 Operating Systems and CP/M

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 such peripherals 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 instructions or 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 Figure 7.3.

Each computer has its own operating system. CP/M (Control Program/Monitor for Microcomputers) is by far the most widely used operating system for the microcomputers designed around the Z80 and the 8085/8080 microprocessors. The CP/M design is, for the most part, independent of the machine, so that microcomputer manufacturers can adapt it to their own designs with minimum changes. To illustrate the operation of a software development system, CP/M is briefly described here in reference to a system with 64K R/W memory.

CP/M

This operating system is divided into three components: BIOS (Basic Input/Output System), BDOS (Basic Disk Operating System), and CCP (Console Command Processor).

BIOS This program consists of input/output routines; it manages data transfer between the microprocessor and various peripherals. This section of CP/M is accessible to the user. Each manufacturer writes a BIOS specifically to the hardware design in a particular system.

BDOS This program directs the activities of the disk controller and manages the file allocation on the disk. The BDOS program allocates memory space under a file name.

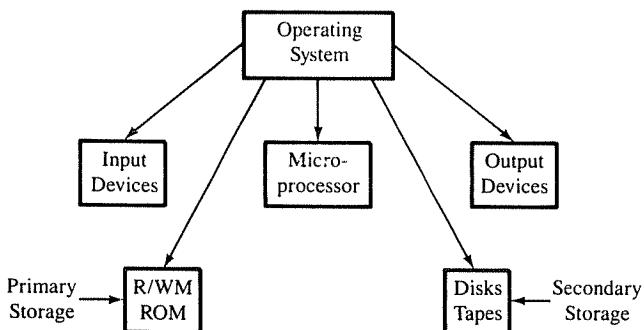


FIGURE 7.3
Operating System and Its Functional Relationship with Various Components of a Computer System

CCP This program reads and interprets the CP/M commands from the keyboard. These commands include such operations as listing the programs on the disk, copying, erasing, and renaming a file. CCP also transfers the program control from CP/M to user or other programs.

When CP/M is loaded into a system's R/W memory, it occupies 6K to 12K of memory at the highest available locations, as shown in Figure 7.4. In addition, the first 256 locations (from 0000 to 00FF_H) are reserved for system parameters. The rest of the R/W memory (approximately 52K to 58K) is available for the user. Once the operating system is loaded into R/W memory, the user can write, assemble, test, and debug programs by using utility programs, which are described in the next section.

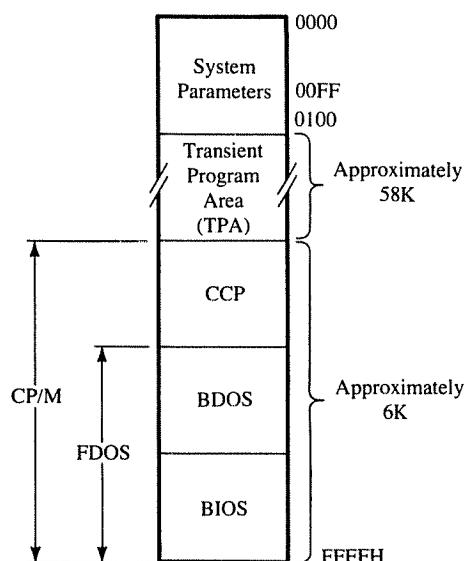
7.13 Tools for Developing Assembly Language Programs

The CP/M operating system includes programs called **utility programs**. These programs can be classified in two categories: (1) file management utilities, and (2) program development utilities. The file management utilities are programs that enable the user to perform such functions as copying, printing, erasing, and renaming files. The program development utilities enable the user to write, assemble, and test assembly language programs; they include programs such as Editor, Assembler, Linking Loader, and Debugger.

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. To write text, the user must call the Editor under CP/M control. As soon as the Editor program is transferred from the disk to the system memory, the program control is transferred from CP/M to the Editor program. The Editor

FIGURE 7.4
CP/M Memory Map with 64K
R/W Memory



has its own commands, with which the user can enter and modify text. Some Editor programs such as "Word Star," "Word Perfect," and "Tele Write" are easy to use. At the completion of writing a program, the exit command of the editor will save the program on the disk under the file name, and will transfer the program control back to CP/M. This file is known as a source file or a source program.

The Editor program is not concerned with whether one is writing a letter or an assembly language program. If the source file is intended to be a program in the Z80 assembly language, the user should follow the syntax of the assembly language and the rules of the assembler.

Z80 ASSEMBLER

Several Assemblers are available commercially as Z80 Assemblers. The following description is accurate for the Microsoft M80 Assembler. The assembler is a program that translates the source file into modules of the Z80 code and generates two files: one is called the print (PRN) or listing file and the other is called the relocatable (REL) file. In addition to translating mnemonics, the Assembler performs such functions as error checking and memory allocations.

The print file includes the source file plus the memory addresses and the Hex code of each instruction. This file is used primarily for documentation and may look like the hand-assembled file shown in the last chapter. The relocatable file is an intermediate file, generated to create two more files: a Hex file and an object file, which is necessary to combine different modules (or programs) and relocate the modules from one block of memory to another. The Assembler is described in more detail in Section 7.2.

LINKING LOADER

The Linking Loader is a program that uses the REL file generated by the Assembler to generate a binary code file called the COM file or object code; it can also generate a Hex file. The COM file is the only file that can be executed by the microcomputer. To execute the program, the COM file is called under CP/M control and executed. The HEX file is used for debugging the code and transferring files from one system to another. This transfer of files among different systems is called either downloading or uploading of files.

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; for example, convert the object code into the source code or mnemonics.

In the M80 Assembler, translating mnemonics into binary code is a two-step process: first, the source file is converted into the REL (Relocatable) file by the Assembler program; then, the REL file is converted into the binary object (COM) file by the Linking Loader program. This is called **program assembly**. Additional files, such as Hex and PRN files, can be also generated using these programs. In addition, the Editor generates the back-up (BAK) file. The BAK file is generated when the user calls the source file for reediting; the BAK file is the copy of the previous file before the user begins to reedit. The BAK file is generated as a precautionary measure, in the event that the user may wish to go back to the previous file. At the completion of the assembly process, the CP/M user will have the following files:

- Source File: This is the source file written by the user. Under CP/M, a filename can be one to eight characters long with an extension of a maximum three characters. The filename and the extension are separated by a dot. For example, the file name can be **DELAY1.ASM**; the extension **ASM** suggests that it names an assembly language file.
- REL File: This is a relocatable binary file generated by the assembler without any specific reference to the user memory. This file is used to generate a COM file and relocate the entire program for storage to specified memory locations.
- PRN File: This is the print 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 loader program and contains program code in hexadeciml notations. This file can be used for debugging the program and transferring files from one system to another.
- COM File: This is the executable file generated by the linking loader program, and it contains binary code.
- BAK File: When the source file is called for reediting, the previous file is saved as the BAK file.

7.14 MS-DOS Operating System

The CP/M operating system is designed for 8-bit microprocessors; however, recent disk-based microcomputers, such as the IBM PC, XT, and AT, are designed around 16-bit microprocessors. In these microcomputers, the **MS-DOS** (Microsoft Disk Operating System) is so widely used that it has become the industry standard. The MS-DOS is in many ways similar to the CP/M, except that it is capable of handling 16-bit data words and large size (1 Megabyte) system memory. Similarly, it is designed to handle disks with quad (high) density disk format with memory capacity of 720K and 1,200K. The latest version of MS-DOS is geared towards handling communication between multi-user systems.

The MS-DOS operating system, installed on IBM PCs, is divided into four components: ROM-BIOS, IBMBIO, IBMDOS, and COMMAND; these are COM files. In a typical 1Mbyte (1,024K-byte) system, the memory space is divided into 16 blocks from 0 to F, each being 64K memory; the Hex address ranges from 00000 to FFFFFH. Generally, the lowest addresses in the 0 block are reserved for system software, the highest block F is

used for ROM-BIOS, and approximately ten blocks (640K) are reserved as the user memory. The remaining blocks are used for such varied purposes as video display and BIOS extensions.

ROM-BIOS This program is functionally similar to BIOS in the CP/M; it is called ROM-BIOS because it is generally installed in Read-Only Memory. The primary function of this program is to communicate with I/O devices when it receives commands from a user's program. The IBMBIO program is an extension of the ROM-BIOS; the IBMBIO program allows modifications in the BIOS programs and additions of new peripherals.

IBMDOS This program directs the activities of the disk controller and also contains DOS service routines; these service routines include such programs as DIR (Directory), FORMAT (Formatting disk), and COPY (Copying files). These programs are also included in CP/M, but they are generally stored on a disk, and are not part of the system.

COMMAND This program reads and interprets the commands from the keyboard and differentiates between the DOS services (such as COPY) and the utility programs such as DEBUG.

In summary, the DOS operating system is conceptually similar to the CP/M operating system, but it is capable of handling large memory size, large word size, and many more functions than the CP/M.

MS-DOS AND CROSS-ASSEMBLERS

The MS-DOS is an operating system designed primarily for 16-bit microprocessors. Now the question is: Why are we discussing it in the context of an 8-bit microprocessor such as the Z80? The answer lies with the widespread use of IBM PCs or their compatibles on college campuses. A 16-bit microprocessor is not ideally suited for learning about microprocessors; they are too complex for the control type applications. But we can use these 16-bit machines to develop (write) Z80 assembly language programs by using a program called cross-assembler. This program translates Z80 mnemonics into appropriate Z80 machine codes. From the user's point of view, it makes no difference whether he/she uses a Z80-based system or any other system with a Z80 cross-assembler. After assembling a program, the Hex file can be directly transferred to R/W memory of your Z80 single-board microcomputer by using a download program. Thus, hardware related laboratory experiments can be easily performed.

ASSEMBLERS

7.2

The assembler, as defined before, is a program that translates assembly language mnemonics or source code into binary executable code. Here, we are using the term **assembler** to include all the utility programs (such as Assembler, Linker Loader) necessary for the

assembly process. This translation process 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 Z80 mnemonics, which are to be translated into binary code.
2. *Comments*, which are reproduced as a part of the program documentation.
3. *Directives* to the assembler that specify such items as starting memory locations, label definitions, and required memory spaces for data.

The first two types of statements have been used in the program of adding two Hex numbers in the last chapter. The format of these statements as they appear in an assembly language source program is identical to the format used here. The third type—directives—and their functions will be described in Section 7.22.

7.21 Assembly Language Format

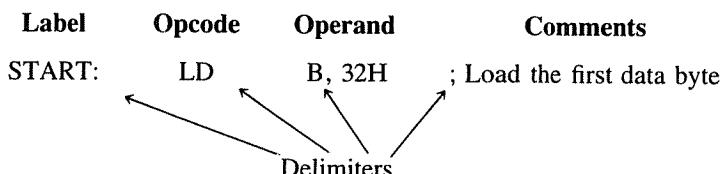
A typical assembly language programming statement is divided into four parts, called **fields**: *label*, *operation code* (opcode), *operand*, and *comments*. These fields are separated by **delimiters** for the CP/M assembler, as shown in Table 7.1.

TABLE 7.1
Delimiters Used in Assembler Statements

Delimiter	Placement
Colon	After label
Space	Between an opcode and an operand
Comma	Between two operands*
Semicolon	Before the beginning of a comment

*Some assemblers may not tolerate space between comma and the operand.

The **assembler statements** have a free-field format, which means that any number of blanks can be left between fields. Comments are optional but are generally included for good documentation. 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:



Delimiters include the colon following START, the space following LD, the comma following B, and the semicolon preceding the comment.

7.22 Assembler Directives

The **assembler directives** are the instructions to the assembler concerning the program being assembled; they are also called *pseudo operations* or *pseudo-ops*. These instructions are neither translated into machine code nor assigned any memory locations in the object file. Some of the important assembler directives for the Z80 assembler are listed and described here.

Assembler Directives	Example		Description
1. ORG (Origin)	ORG	0100H	The next block of instructions should be stored in memory locations starting from 0100 _H .
2. END	END		End of assembly. The HALT instruction suggests the end of a program, but that does not necessarily mean the end of assembly.
3. EQU (Equate)	PORT1	EQU 01H	The value of the term PORT1 is equal to 01 _H . Generally, this means the PORT1 has the port address 01 _H .
	INBUF	EQU 2099H	The value of the term INBUF is 2099 _H . This may be the memory location used as input buffer.
	OUTBUF	EQU INBUF + 4	The equate can be expressed by using the label of another equate. This example defines the OUTBUF memory location in terms 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 of two bytes at a time. This statement reserves two locations for 2050 _H .

6. DS (Define Storage)	OUTBUF: DS 4	Reserves a specified number of memory locations. In this example, four memory locations are reserved for OUTBUF.
------------------------------	--------------	------------------------------------------------------------------------------------------------------------------

7.23 Advantages of the 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 quite time-consuming for programs larger 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 quickly reassemble the entire program 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.

7.3

WRITING PROGRAMS USING AN ASSEMBLER

This section deals primarily with writing programs using an assembler. The illustrative example is simple and has been selected to demonstrate the use of assemblers. 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.

To illustrate how to write a source program, we selected the Z80 assembler called MACRO-80 (M80), developed by Microsoft. The example is taken from the last chapter, where it was assembled using manual assembly. The source program is written using an Editor under the file name PROGRAM1.MAC. To assemble the program using the assembler M80, the file name must have the extension .MAC and it should include the pseudo-op .Z80 at the beginning.

7.31 Illustrative Program: Addition of Two Hexadecimal Numbers

This illustrative program is the same one we discussed in the last chapter. The problem statement is repeated here for convenience; refer to Section 6.31 for analysis.

PROBLEM STATEMENT

Write instructions to load the two hexadecimal numbers 32_H and $A2_H$ into registers B and C, respectively. Add the numbers, and display the sum at the LED output port PORT1.

SOURCE PROGRAM

;This program adds two Hex bytes and displays the sum.

```
.Z80
      ORG 2000H      ;Begin assembly at 2000H
PORT1  EQU 01H      ;Output port address
START: LD B, 32H    ;Load first byte
        LD C, 0A2H*  ;Load second byte to be added
        LD A, C      ;Copy one of the bytes into A
        ADD A, B     ;Add two bytes
        OUT (PORT1), A ;Display the result
        HALT         ;End
END
```

This program illustrates the following assembler directives:

ORG

The object code will be stored starting at the location 2000_H .

EQU

The program defines one equate: PORT1. In this program it would have been easier to write the port address directly with the instructions. However, equates are essential in development projects where hardware and software design are done concurrently, and they are also useful in long programs because they make it easy to change or redefine port addresses.

Label

The program illustrates one label: START. This label represents the memory location 2000_H . In this illustration, the label is not particularly useful; generally, labels are used to specify Jump and Call addresses. In writing assembly language programs, it is convenient to identify a Jump or Call address by a label because absolute addresses are not known in the beginning. This is especially true when programmers in a team are assigned various tasks. Also, if any changes (deletions and additions) are made in the source program, the assembler will reassign all label addresses when it is reassembled. In manual assembly, the entire program must be rewritten with new addresses if any changes are to be made.

End

The end of assembly.

*Any Hex number that begins with A through F must be preceded by zero.

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.

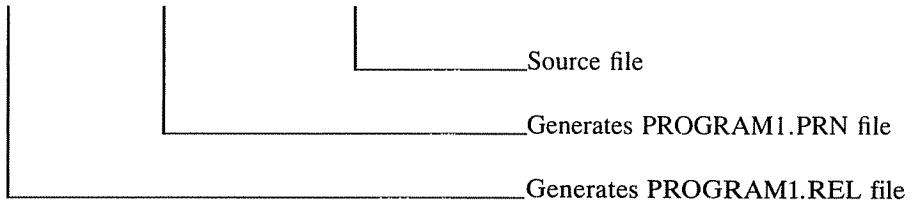
ASSEMBLED PRINT FILE

To create the REL file and the PRN file from a source file, the following command format is necessary when the prompt * appears after calling M80:

RELFILe, PRNFILE = SOURCE FILE

Therefore, to generate the relocatable object file and the print file from the source file PROGRAM1.MAC, the command is

PROGRAM1,PROGRAM1 = PROGRAM1



In this command, any file names can be given to the REL and the PRN files. For example, if we were to substitute TEST1 for the first word PROGRAM1, the assembler will generate a TEST1.REL file from the PROGRAM1 source file.

The PRN file generated from the source program has five columns: memory addresses, Hex codes, labels, mnemonics and comments. It lists the memory addresses of the first byte of each instruction with its Hex codes on the same line. For example, the listing shows that the first memory address is 2000 and the first two Hex codes are 06 32; the next address is 2002. The memory address 2001, then, holds the Hex byte 32. In addition to the program listing, the PRN file includes the list of symbols, equates, and error messages.

Error Messages In addition to translating the mnemonics into object code, the assembler also gives error messages. These messages are of two types: *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 lists the errors, but it is able to complete the assembly.

PRINT (PRN) FILE

;This program adds two Hex bytes and displays the sum.

```

.Z80
        ORG 2000H      ; Begin assembly at 2000H
0001      PORT1 EQU 01H      ; Output port address
2000'    06 32  START: LD   B, 32H      ; Load first byte
2002'    0E A2          LD   C, 0A2H      ; Load second byte to be added
2004'    79          LD   A, C      ; Copy one of the bytes into A
2005'    80          ADD  A, B      ; Add two bytes
2006'    D3 01          OUT  (PORT1), A      ; Display the result
2008'    76          HALT      ; End
        END

```

Macros:

Symbols:

PORT1 0001 START 2000'

No Fatal error(s)

GENERATING COM AND HEX FILES

The Linking Loader program (L80) generates COM and HEX files. To create the COM file and the HEX file from the PROGRAM1.REL file, the following command format must be used after calling L80:

PROGRAM1, PROGRAM1/N/X/E

This command will generate PROGRAM1.COM and PROGRAM1.HEX files, save them on the disk, and exit to the operating system.

Precautions in Writing Programs Assembler programs are available from various software companies, and for the most part, they follow a similar format. However, we suggest the following precautions in writing assembly language programs.

1. The M80 Assembler allows the free format in writing the source code; however, some assemblers (especially cross-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 letter "O," a binary by 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 mnemonic and gives an error message because it does not understand the mnemonic.
4. Some assemblers require a colon after a label.
5. When a 16-bit address is used in a mnemonic (such as Jump to 2050H), the M80 prints the address as 2050; however, it is stored in the reversed order in memory. Some assemblers print the address as 50 20.

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, and the Assembler translates mnemonics into machine code and provides error messages. The Debugger assists in debugging the program.

The program thus assembled is in many ways similar to that of the hand assembly program except that the program written for the assembler includes assembler directives, which are instructions 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.

ASSIGNMENTS

Check the appropriate answer in **1–10**.

1. The process of accessing information on a floppy disk is
 - a. random.
 - b. serial.
 - c. semi-random.
2. The operating system of a computer is defined as
 - a. hardware that operates the floppy disk.
 - b. a program that manages files on the disk.
 - c. a group of programs that manages and directs hardware and software in the system.
3. The Editor is
 - a. an assembly language program that reads and writes information on the disk.
 - b. a high-level language program that allows the user to edit programs.
 - c. a program that allows the user to write, modify, and store text in the computer system.
4. The Assembler is
 - a. a compiler that translates statements from high-level language into assembly language.
 - b. a program that translates mnemonics into binary code.
 - c. an operating system that manages all the programs in the system.

5. A file is
 - a. a group of related records stored as a single entity.
 - b. a program that transfers information between the system and the floppy disk.
 - c. a program that stores data.
6. The COM file
 - a. consists of Hex digits and is used for communication.
 - b. is the only file that can be interpreted and executed by the microprocessor.
 - c. consists of Z80 mnemonics.
7. The Hex file generated by the M80 Assembler is used primarily
 - a. to reduce the memory requirement for storing files.
 - b. to transfer a file from one system to another.
 - c. to transfer a file between a floppy disk and the system's R/W memory.
8. A hard-sectorized disk is
 - a. a floppy disk in which each sector is identified with a hole.
 - b. a hard disk that stores information on an aluminum-based magnetic surface.
 - c. a double-density, double-sided floppy disk.
9. A disk controller is
 - a. a program that manages the files on the disk.
 - b. a circuit that interfaces the disk with the microcomputer system.
 - c. a mechanism that controls the spinning of the disk.
10. The MS-DOS is
 - a. an operating system that is designed primarily to handle the communication between the 16-bit microprocessor and its peripherals.
 - b. an updated version of the CP/M operating system.
 - c. an application program that handles communication between various systems.
11. Assemble the following program with the starting address 0100_H, and print the PRN file. The address of the output port PORT7 is 07_H.

```
START: LD B, 32H      ;Load B with first data byte
        LD C, 0A2H      ;Load C with second data byte
        LD A, C          ;Copy (C) into A for addition
        ADD A, B          ;Add two bytes
        JP NC, DISPLAY   ;If sum < FFH, display sum at PORT7
        LD A, 01H          ;If sum > FFH, load 01 to display
                           ; as over load
DISPLAY: OUT (PORT7), A ;Display result at PORT7
        HALT
        END
```


Introduction to Z80 Instructions and Programming Techniques

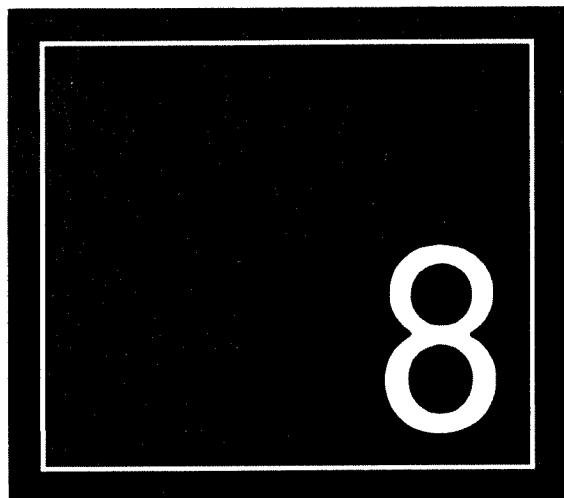
When a microcomputer is asked to execute a program stored in its memory, it reads one instruction at a time and performs the task specified by the instruction. Each instruction in the program is a command, in binary, to the microprocessor to perform an operation. In Chapter 6, we examined briefly the Z80 instruction set and its capability. In this chapter, we will introduce a few selected instructions and illustrate them with examples. These instructions are selected from three groups: data copy, arithmetic, and branch operations.

A computer is at its best, relative to human capability, when it is asked to repeat such simple tasks as adding or copying. The programming techniques—such as **looping**, **indexing**, and **counting**—necessary to perform such tasks are introduced and illustrated with two programs. This chapter also includes a brief discussion of debugging programs.

Finally, a group of special Z80 instructions that perform multiple tasks are introduced with illustrative examples.

OBJECTIVES

- Explain the functions of data copy instructions and how the contents of the source register and the destination registers are affected.



- List four types of data copy operations and explain the term *addressing mode*.
- Explain how a memory address is specified to copy data from and to a memory register.

- Explain how data are transferred from and to I/O devices.
- Explain the functions of arithmetic instructions (ADD, SUB, INC, DEC) and how flags are affected by these instructions.
- Write a set of commands using data copy and arithmetic instructions to perform a given task.
- Explain the functions of unconditional and conditional jump instructions and how they are used for decision making.
- Draw a flowchart of a conditional loop to illustrate the indexing and counting techniques.
- List the seven blocks of a generalized flowchart illustrating data acquisitions and data processing.
- Write a program to copy data from one block of memory to another block including the case of overlapping blocks.
- Write a program to perform arithmetic operations on given data stored in memory.
- List the types of errors that frequently occur in writing assembly language programs and in hand assembling the code. Recognize the errors in a given program.
- List Z80 special instructions and explain how they provide more flexibility and improve efficiency in writing Z80 programs.
- Modify the previously written programs using the Z80 special instructions.

8.1 DATA COPY (LOAD) OPERATIONS

In this section, we focus on three types of **data copy operations**: data copy related to internal registers, memory, and I/Os. Instructions frequently used are illustrated below, and the Z80 block transfer instruction will be discussed later in the chapter. In addition, one machine control instruction—HALT—is introduced; this instruction is necessary to indicate the end of a program.

8.11 Data Copy (Load) among Registers

In this group, we have three types of instructions: data copy from one register to another, loading 8-bit data into a register and loading 16-bit data into a register pair.*

Opcode	Operand	Bytes	Addressing Mode	Description
LD	rd, rs	1	Register	Copy data from source register rs to destination register rd. In this mode, the operand is a part of the opcode.
LD	r, 8-bit	2	Immediate	Load 8-bit data of the second byte into the specified register. In this mode, the second byte is the operand.

*Appendix A includes complete descriptions of these instructions in alphabetical order with illustrative examples.

LD	rp, 16-bit	3	Immediate Extended	Load 16 bits into the specified register pair. In this mode, two bytes following the opcode are the operands.
LD	rx, 16-bit	4	Immediate Extended	Load 16 bits into the specified index register.
HALT		1		This is a machine control instruction. The processor stops executing and enters into Wait state.

General Characteristics

1. Copy (Load) instructions do not affect flags.
2. The operands of copy instructions specify a destination register first, followed by a source register; they are separated by a comma.
3. The data byte is copied without modifying the contents of a source register.
4. A 16-bit operand is stored in two consecutive memory locations in the reversed order: the low-order byte first, followed by the high-order byte.
5. The instructions related to the index registers IX and IY have two-byte opcodes.

Write instructions to load $97H$ into the accumulator, $2050H$ into HL registers, and $2075H$ into the index register IX. Copy the contents of the accumulator into register C and the contents of register H into register B. Write the HALT instruction at the end of the sequence. Enter the machine codes of these instructions in R/W memory starting from $2000H$, and show the contents of each register after the execution.

Example
8.1

Solution

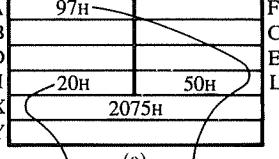
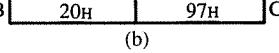
Memory Address	Hex Code	Opcode	Operand	Register Contents
2000	3E	LD	A, 97H	
2001	97			
2002	21	LD	HL, 2050H	
2003	50			
2004	20			
2005	DD	LD	IX, 2075H	
2006	21			
2007	75			
2008	20			
2009	79	LD	C, A	
200A	44	LD	B, H	
200B	76	HALT		

FIGURE 8.1
Instructions and Register Contents

Description

1. The first instruction LD A, 97_H is a 2-byte instruction; the opcode 3E and the operand 97 are stored in the first two memory locations. This instruction loads 97_H into the accumulator (Figure 8.1(a)).
2. The second instruction is a 3-byte instruction that loads 16-bit data (2050_H) into the HL registers (Figure 8.1(a)). The low-order byte (50_H) is stored first in memory location 2003_H, followed by the high-order byte (20_H).
3. The third instruction (IX, 2075_H) is a 4-byte instruction; it has a 2-byte opcode (DD and 21). This instruction loads 16-bit data (2075_H) into the index register IX.
4. The remaining two instructions are 1-byte instructions; they copy data from one register to another as shown in Figure 8.1(b). It is important to note that the copy operations do not destroy the contents of the source registers. Figure 8.1 shows that registers A and H retain their contents after the copy operations.
5. The last instruction (HALT) is a machine control instruction; it forces the machine into the Wait state.

8.12 Data Copy Between Z80 Registers and Memory

To copy data from and into memory, the 16-bit address of a selected memory register must be specified, and this memory address can be specified in various ways: for example, using the HL register, register pairs, or a direct 16-bit address. Methods using index registers are discussed after the discussion of 2's complement arithmetic because the index registers include a displacement byte, which is expressed as a signed 2's complement number. In Z80 mnemonics, the memory address is enclosed in parentheses, as shown in the following list.

Opcode	Operand	Bytes	Addressing Modes	Description
LD	r, (HL)	1	Register Indirect	Copy contents of memory into register r. The memory address is specified indirectly by the number in the HL register; therefore, this is called register indirect addressing.
LD	(HL), r	1	Register Indirect	Copy contents of register r into memory.
LD	(HL), 8-bit	2	Register Indirect & Immediate	Copy 8-bit data into memory. This mode is combination of indirect and immediate addressing.

Note: In these three instructions, the memory address is specified by the contents of register HL, and register r can be any one of the general-purpose registers.

LD	A, (rp)	1	Register Indirect	Copy contents of memory into accumulator.
LD	(rp), A	1	Register Indirect	Copy contents of accumulator into memory.

Note: In the preceding two instructions, the memory address is shown by the contents of a register pair (BC or DE). However, these instructions can copy data from and into the accumulator only.

LD	A, (16-bit)	3	Extended	Copy contents of memory into accumulator.
LD	(16-bit), A	3	Extended	Copy contents of accumulator into memory.

Note: In these instructions, the memory address is the 16-bit operand, and these instructions can copy data from and into the accumulator only.

General Characteristics

1. No flags are affected by these data copy operations.
2. Memory-related data copy operations can be recognized by the parentheses around the operand.
3. Register HL is a versatile memory pointer; a data byte can be copied from any memory location to any general-purpose register and vice versa. In addition, HL can be used to load a byte directly into memory.
4. A 16-bit direct address and other register pairs (BC and DE) can be used as memory pointers to copy data from a memory location into the accumulator and vice versa. However, these memory pointers cannot be used to copy data between general-purpose registers and memory.

The memory location 2050_H contains the data byte 37_H . Write instructions to copy the byte from the memory location into the accumulator. Illustrate three different ways of transferring the byte from memory to the microprocessor and list the associated machine codes.

Example
8.2

Solution

1. The first method of copying a byte from memory into the microprocessor is by using HL register as a memory pointer; this is an illustration of indirect addressing. First, we need to load the memory address into the HL register and then use the contents of HL as a memory pointer (Figure 8.2(a)).
2. The second method of copying a byte from memory into the microprocessor is by using BC or DE as a memory pointer; this is also the indirect addressing (Figure 8.2(b)).
3. The third technique is to use the direct extended addressing (Figure 8.2(c)).

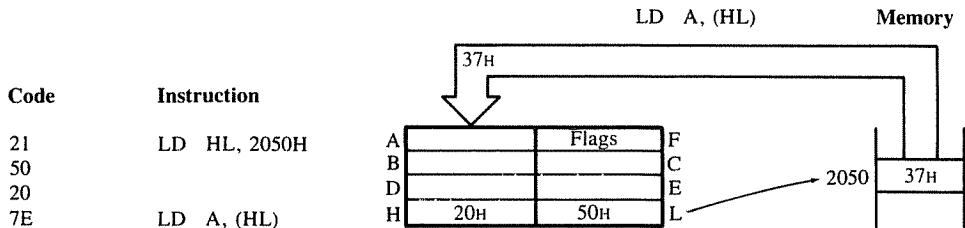


FIGURE 8.2
(a) Indirect Addressing Using HL

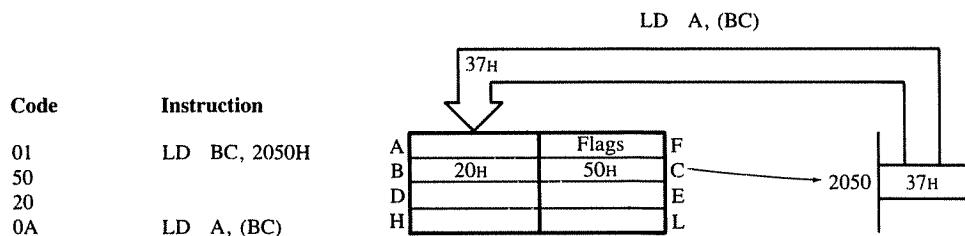


FIGURE 8.2
(b) Indirect Addressing Using BC

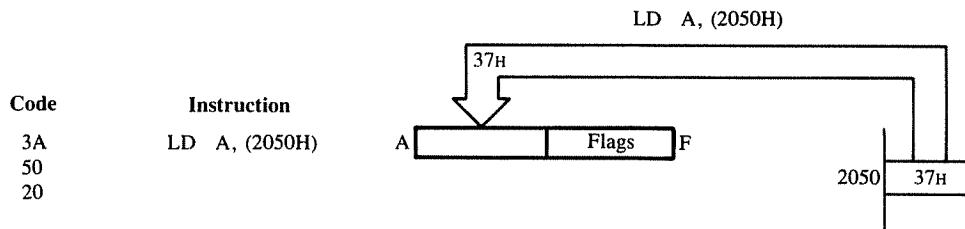


FIGURE 8.2
(c) Extended Addressing

**Example
8.3**

The memory location 2040_{H} contains the data byte $F2_{\text{H}}$. Copy the data byte $F2_{\text{H}}$ from the memory location 2040_{H} into 2070_{H} using memory pointers. Then, clear the memory location 2040_{H} . Enter the machine codes of these instructions in memory locations starting from 2000_{H} . Describe how data copy operations are performed.

Solution

Memory Address	Hex Code	Opcode	Operand	Comments
2000	21	LD	HL, 2040H	; Set up HL as memory pointer for 2040H
2001	40			
2002	20			

2003	01	LD	BC, 2070H	; Set up BC as memory pointer
2004	70			for 2070H
2005	20			
2006	7E	LD	A, (HL)	; Copy data (F2H) into accumulator
2007	02	LD	(BC), A	; Copy data into memory (2070H)
2008	36	LD	(HL), 00	; Clear location 2040H
2009	00			
200A	76	HALT		

Description

1. The first two instructions load registers HL and BC with the numbers 2040_H and 2070_H respectively. These are not memory-related data copy instructions because the operands do not have any parentheses.
2. The next two instructions copy the data byte ($F2H$) stored in memory location 2040_H into the accumulator and from the accumulator into location 2070_H (see Figure 8.3).
3. The next instruction LD (HL), 00 is a 2-byte instruction; it clears the memory location 2040_H by loading 00 into the memory location pointed to by the HL register.

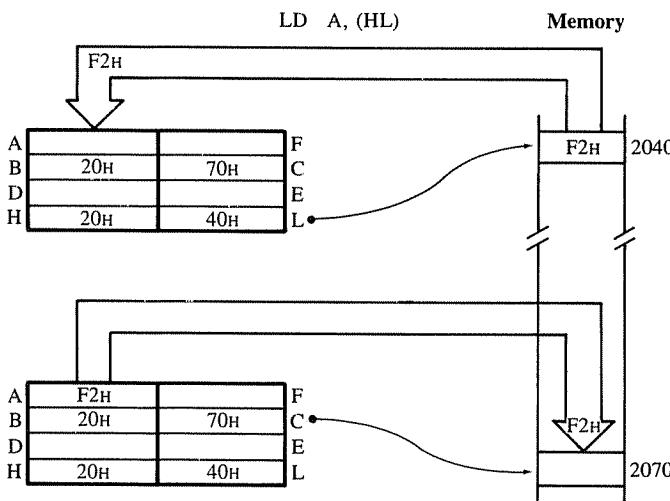


FIGURE 8.3
Data Copy between Microprocessor and Memory

8.13 Data Copy Between Accumulator and I/Os

In the Z80 instruction set, input and output devices are identified by 8-bit addresses. The set includes several instructions that can read data from an input device (also known as input port) and write data into an output device (or output port). Two of these I/O instructions are described here:

Opcode	Operand	Bytes	Description
IN	A, (8-bit)	2	Read data from an input port into the accumulator.
OUT	(8-bit), A	2	Write data to an output port from the accumulator.

General Characteristics

1. These I/O instructions do not affect flags. (Some Z80 I/O instructions do affect flags; they are discussed later.)
2. The I/O instructions have 8-bit operands; thus, the Z80 is capable of addressing 256 input and 256 output ports.
3. The 8-bit I/O addresses are enclosed in parentheses similar to those of memory addresses.

Example 8.4 Read the switches connected to the input port 01_{H} (Figure 8.4). Display the reading at the LED output port 07_{H} and store it in memory location 2060_{H} .

Solution Instructions are as follows:

Opcode	Operand	Comments
IN	A, (01H)	; Read input switches
OUT	(07H), A	; Display switch reading at output port
LD	(2060H), A	; Store switch reading in memory
HALT		

Description

1. Figure 8.4 shows that the switch positions at the input port 01_{H} provide the reading $0\ 1\ 0\ 0\ 1\ 1\ 1\ 1$ ($4F_{H}$). The first instruction reads the switch positions and places the reading in the accumulator.
2. The OUT instruction sends the accumulator contents to the output port 07_{H} and displays the corresponding LEDs (Figure 8.4).
3. The last instruction stores the accumulator contents in memory location 2060_{H} .

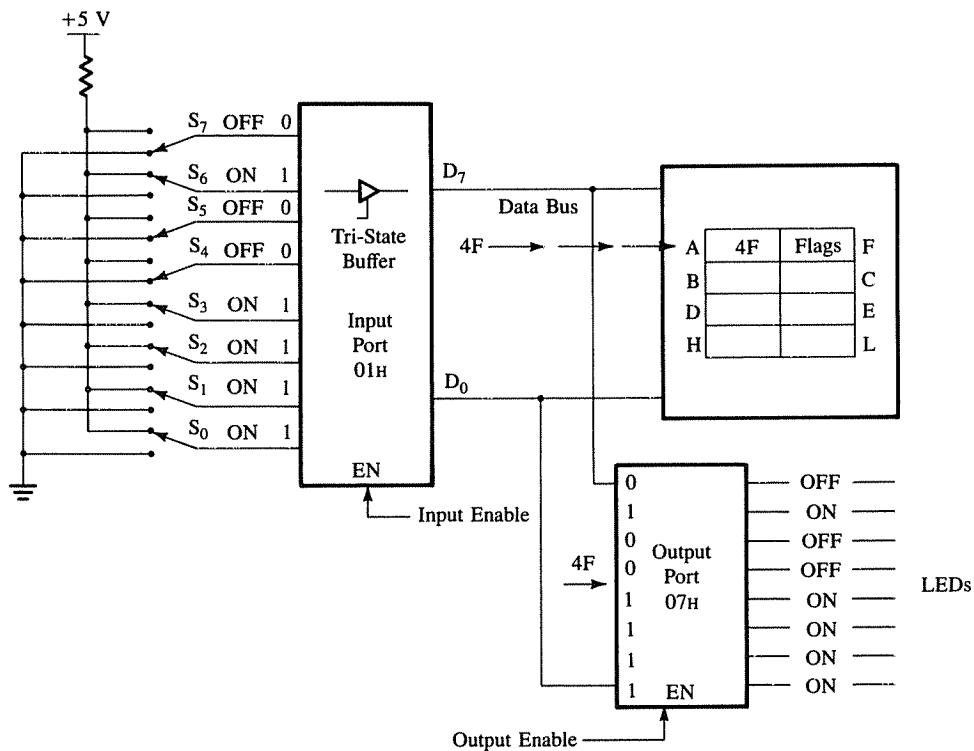


FIGURE 8.4
Reading Data at Input Port and Sending Data to Output Port

ARITHMETIC OPERATIONS

8.2

The Z80 microprocessor performs various **arithmetic operations** such as *addition*, *subtraction*, *increment/decrement*, and *1's* and *2's complement*. Most of these operations are concerned with 8-bit operands. The instruction set also includes some 16-bit operations which will be discussed in later chapters. (See Appendix A for complete alphabetical listing of the Z80 instruction set and how flags are affected by the instructions.)

8.21 Addition and Subtraction

The addition and subtraction operations are performed in relation to contents of the accumulator. We focus here on three types of operands: register contents, 8-bit data, and memory contents.

Opcode	Operand	Bytes	Description
ADD	A, r	1	Add contents of register r to the contents of the accumulator, and store the result in the accumulator.
ADD	A, 8-bit	2	Add 8-bit data directly to the accumulator.
ADD	A, (HL)	1	Add memory contents to the accumulator.
SUB	r	1	Subtract contents of register r from the accumulator.
SUB	8-bit	2	Subtract 8-bit data from the accumulator.
SUB	(HL)	1	Subtract memory contents from the accumulator.

General Characteristics:

These Arithmetic Instructions

1. assume that the accumulator is one of the operands.
2. modify all the flags according to the result of the operation.
3. place the result in the accumulator.
4. do not affect the contents of the operand register or memory.

8.22 Increment/Decrement Instructions

The following instructions are a special type of arithmetic instructions; they increment or decrement the contents of the operand by one. These instructions are generally used in counting and indexing.

Opcode	Operand	Bytes	Description
INC	r	1	Increment the contents of register r.
INC	(HL)	1	Increment the contents of memory.
INC	rp	1	Increment the contents of register pair rp (Register pairs are BC, DE, HL, and SP).
DEC	r	1	Decrement the contents of register r.
DEC	(HL)	1	Decrement the contents of memory.
DEC	rp	1	Decrement the contents of register pair rp.

General Characteristics

1. In these instructions, the operand can be any of the 8-bit registers r, memory, or register pairs r_p . The result is stored back into the same operand register.
2. The instructions dealing with 8-bit registers affect all the flags except the Carry (CY) flag.
3. The instructions dealing with register pairs do not affect any flags. This is important to remember when a register pair is used as a 16-bit counter.

8.23 1's and 2's Complement Instructions

The Z80 instruction set includes the following instructions that perform complement operations with the contents of the accumulator. The addressing mode is implied; the accumulator is implied as the operand.

Opcode	Operand	Bytes	Description
CPL		1	Invert each bit of the accumulator. This can also be classified as the NOT function. No flags (except H and N) are affected.
NEG		2	Subtract the contents of the accumulator from 00; this is equivalent to taking 2's complement of the number in the accumulator. This instruction affects all the flags.

Load two unsigned numbers $F2_H$ and 68_H in registers B and C respectively, and store $A2_H$ in memory location 2065_H , using the HL register as a memory pointer. Subtract 68_H from $F2_H$, complement the result, and add $A2_H$ from memory. Store the final answer in memory location 2066_H . Show register contents and the status of S (Sign), Z (Zero), and CY (Carry) flags as each instruction is being executed.

Example
8.5

Solution

Instructions	Mnemonics	Register Contents					Flags		
		A	B	C	H	L	S	Z	CY
1. LD BC, F268H		X	F2	68	X	X			No change
2. LD HL, 2065H		X			20	65			
3. LD (HL), A2H	A2 2065	X							
4. LD A, B		F2							
5. SUB C (F2 - 68) →		8A					1	0	0
6. CPL (Invert 8A) →		75							No change
7. ADD A, (HL) (75 + A2)		17			20	66	0	0	1
8. INC HL									No change
9. LD (HL), A	17 2066								
HALT		17	F2	68	20	66	0	0	1

Description

1. The first instruction loads register BC with the given bytes. This could be achieved by using two separate load instructions for each register, but loading a register pair is slightly more efficient.
2. The second instruction sets up HL as a memory pointer for location 2065_H , and the third instruction loads $A2_H$ into the memory location indicated by HL.

3. To subtract C from B, it is necessary to copy the contents of B into the accumulator (Instruction 4).
4. Instructions 1 through 4 are all data copy instructions; they do not affect flags. All the flags will remain in their initial conditions before the program is executed.
5. Instruction 5 performs the subtraction in 2's complement and places $8A_H$ in the accumulator as shown below. The subtraction method using 2's complement involves three steps: (1) Find 2's complement of the subtrahend, (2) Add the 2's complement to the minuend, and (3) Complement CY. (Refer to Appendix B if you are unfamiliar with the technique.)

Register C = 68_H →	0 1 1 0	1 0 0 0	Subtrahend	
2's Com. of 68_H →	1 0 0 1	1 0 0 0	2's Complement of Subtrahend	
Accumulator = $F2_H$ →	+ 1 1 1 1	0 0 1 0		
	1	1 0 0 0	1 0 1 0	Sum
Complement CY →	0 1 0 0 0	1 0 1 0	$\rightarrow 8A_H$ Final Result	
				Flags: S = 1, Z = 0, and CY = 0

The result of this subtraction sets the Sign flag and resets the Zero and Carry flags. However, the result is not a negative number. After an arithmetic operation, if bit $D_7 = 1$, the Sign flag is set. In this subtraction, the Sign flag should be ignored because data bytes are not signed numbers (see further discussion in Section 8.24).

6. The instruction CPL inverts the contents of the accumulator $8A_H$; the result is 75_H . This instruction does not affect any flags, so the flags set by the previous instruction are preserved.

$1 0 0 0 1 0 1 0$ ($8A_H$) → 0 1 1 1 0 1 0 1 (75_H)

7. Instruction 7 adds $A2_H$ from the memory location pointed to by HL to the accumulator contents (75_H). The result is 117_H . The instruction places 17_H into the accumulator, sets the CY flag, and resets the S and Z flags.

Accumulator = 75_H →	0 1 1 1	0 1 0 1	Flags: S = 0, Z = 0, CY = 1	
Memory (2065_H) = $A2_H$ →	+ 1 0 1 0	0 0 1 0		
C →	1	0 0 0 1		0 1 1 1

8. Instruction 8 increments HL to point to the next location 2066_H , and the next instruction stores the result in the memory location 2066_H .

8.24 Flags and Decision Making

As described in Chapter 3, the Z80 architecture includes six flags, which are flip-flops that are set or reset after the execution of arithmetic and logic operations, with some exceptions. Four of the flags (S, Z, P/V, CY) can be used by the programmer for decision

making in conjunction with Jump and Call instructions; the remaining two (H, N) are used internally by the microprocessor for BCD arithmetic. The thorough understanding of flags is critical to writing assembly language programs.

In many ways the flags are like signs on an interstate highway that help drivers in decision making. A driver sees one or more signs at a time, but continues along the highway ignoring the signs until the appropriate sign is found, and then he or she changes direction or takes an exit. Flags function similarly as signs of data conditions. After an operation, one or more flags is set (or reset) and can be used to change the direction of program sequence by using Jump instructions (discussed in the next section). The following illustrations from Example 8.5 may clarify some of the critical issues.

1. In Example 8.5, Instruction 5 sets the Sign flag and resets the other flags. However, the sign flag can be ignored because the numbers loaded into registers are unsigned numbers. The Sign flag should be considered only when the programmer is dealing with signed numbers.
2. Instruction 7 sets the Carry flag and resets the other flags. If the programmer is adding numbers and is interested in finding the total, the Carry flag must be used to test for a sum larger than an 8-bit number.
3. Another important observation that can be made after the execution of Instruction 7 is that the flags set by Instruction 5 are altered by Instruction 7. Thus, if the programmer is interested in making a decision based on the Sign flag, it should be made before that flag is altered by another operation.

8.25 Signed Numbers and Flags

The microprocessor is incapable of understanding a + or - sign unless the sign is represented in the form of binary digits. Therefore, in 8-bit microprocessors, bit D₇ is reserved for the sign by the user when signed numbers are used in arithmetic operations. For a positive number, bit D₇ is 0, and for a negative number, D₇ is set to 1; the remaining seven bits represent the magnitude of a number. If a number is negative, it is represented in 2's complement. In an 8-bit microprocessor, the largest positive number is 0111 1111 (7F_H = +127₁₀), and the largest negative number is 1000 0000 (80_H = -128₁₀).

The Z80 microprocessor has two flags to indicate the status of the arithmetic results in signed numbers: Sign and Overflow. After an arithmetic (or logical) operation, if bit D₇ = 1, the Sign flag is set, and if D₇ = 0, the Sign flag is reset. However, this flag can be misleading when the result of an addition exceeds the magnitude 7F_H or that of the subtraction exceeds 80_H. These conditions are known as overflow and are indicated by the P/V flag.

The P/V flag is a dual-purpose flag; in logical operations it indicates the parity, and in arithmetic operations it indicates the overflow (we have discussed this flag in Chapter 3). In arithmetic operations, the P/V flag is used to indicate an overflow. If the sum of an addition of two positive numbers exceeds 7F, bit D₇ becomes 1, indicating a negative number. However, the Z80 sets the P/V flag to indicate the error in the result. The critical point to remember is that the Z80 does not know whether the numbers are signed, unsigned, or just individual digits. The interpretation of the flags is the responsibility of the user.

**Example
8.6**

Add two signed numbers: $+29_{\text{H}}$ and $+76_{\text{H}}$. Indicate the status of the flags S, P/V, and CY if the operation is performed by the Z80 microprocessor. Explain how the flags are affected if the numbers are unsigned.

Solution

$$\begin{array}{r}
 +29_{\text{H}} = 0010\ 1001 \\
 + \\
 +76_{\text{H}} = 0111\ 0110 \\
 \hline
 +9F_{\text{H}} = 1001\ 1111
 \end{array}$$

CY = 0 because the sum is less than FF_{H} ,

S = 1 because $D_7 = 1$, and

P/V = 1 because the sum exceeds $7F_{\text{H}}$.

In this addition of two positive numbers, the sign flag erroneously indicates that the sum is negative; however, the overflow flag (P/V) suggests that the result has an overflow from bit D_6 and that the result is therefore inaccurate. The user must check the P/V flag and correct the sum.

If these numbers were unsigned numbers, the result and the status of the flags would not be altered. The interpretation of the result would therefore be different; the user should ignore the S and P/V flags and check for the CY flag. In this example, the sum is $9F_{\text{H}}$ with no carry.

8.3

BRANCH OPERATIONS

The **branch instructions** and their associated flags are the key to the power of a computer or its microprocessor. These instructions can change the sequence of execution based on certain data conditions indicated by the flags; thus, they are decision-making instructions.

The branch instructions are classified into three categories, as listed in Chapter 6: (1) *Jump instructions*, (2) *Call and Return instructions*, and (3) *Restart instructions*. In this chapter, we concentrate on Jump instructions.

8.31 Jump Instructions

The Jump instructions can be divided into two groups: *absolute jump* and *relative jump*. In case of absolute jump, the operand specifies the 16-bit address to which the program sequence should be transferred; these are 3-byte instructions. The relative jump instruc-

tions are 2-byte instructions and contain an operand that specifies 8-bit displacement, forward or backward (in 2's complement), in relation to the address of the Jump instruction; these instructions are discussed in the next Section.

The absolute jump instructions can be further classified into two groups: *unconditional* and *conditional jump*. The conditional jump instructions are implemented based on the status of four flags: S (Sign), Z (Zero), CY (Carry), and P/V (Parity/Overflow). Two instructions are associated with each flag: one for when the flag is set and the other for when it is reset. The list of Jump instructions is as follows:

Opcode	Operand	Bytes	Description
JP	16-bit	3	Jump unconditional to memory location specified by the 16-bit operand.
JP	C, 16-bit	3	Jump on carry to 16-bit address (CY = 1).
JP	NC, 16-bit	3	Jump on no carry to 16-bit address (CY = 0).
JP	Z, 16-bit	3	Jump on zero to 16-bit address (Z = 1).
JP	NZ, 16-bit	3	Jump on no zero to 16-bit address (Z = 0).
JP	M, 16-bit	3	Jump on minus to 16-bit address (S = 1).
JP	P, 16-bit	3	Jump on positive to 16-bit address (S = 0).
JP	PE, 16-bit	3	Jump on parity even to 16-bit address (P/V = 1).
JP	PO, 16-bit	3	Jump on parity odd to 16-bit address (P/V = 0).

General Characteristics

1. The Jump (JP) instructions are 3-byte instructions. The second byte specifies the low-order address and the third byte specifies the high-order address.
2. A conditional jump instruction checks for the appropriate flag. If the condition is true, the program sequence is changed to the memory location specified by the operand; otherwise, the execution continues to the next instruction.
3. The Jump instructions do not affect any flags.

Write instructions to load two Hex bytes BYTE1 and BYTE2 into registers B and C, respectively, and add the bytes. If the sum is larger than 8 bits, display 00H as the overload condition at output port PORT1 and clear the memory location OUTBUF; otherwise, store the sum in memory location OUTBUF. Draw a flowchart and assemble the program starting at location 2000_H. The data bytes and the labels are defined as follows:

Example
8.7

BYTE1 = 9A_H, BYTE2 = A7_H, PORT1 = 01_H, and OUTBUF = 2050_H

This problem is similar to Example 6.1 with some variations in display and data storage. A flowchart and instructions are as follows:

Solution

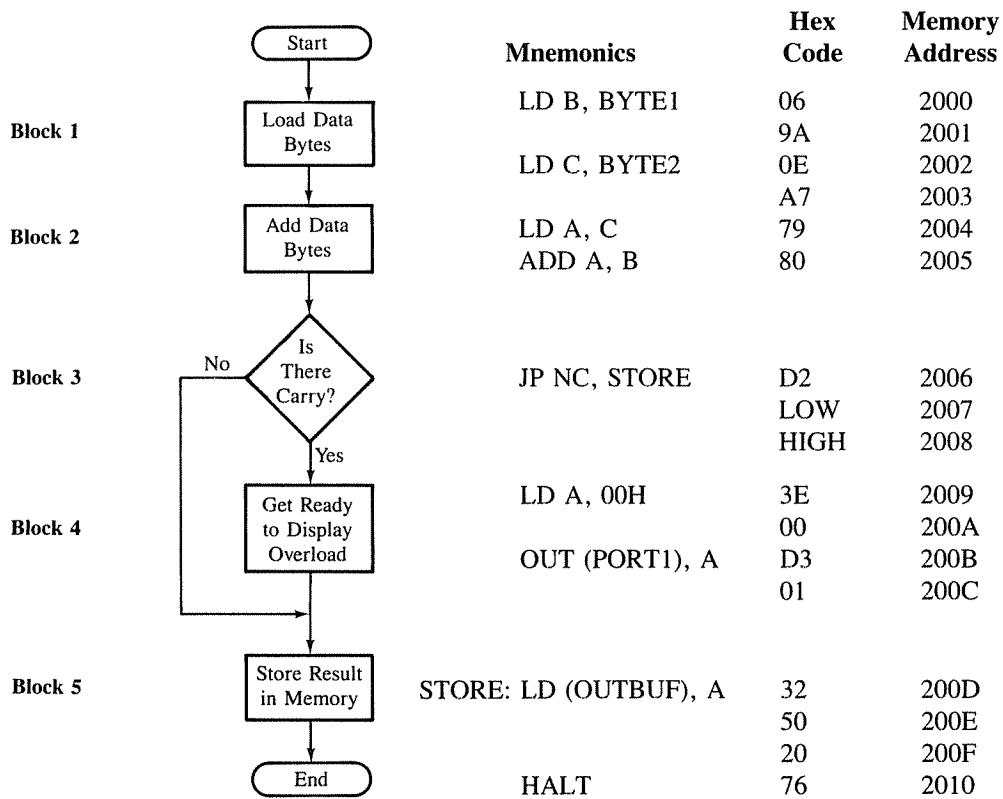


FIGURE 8.5
Flowchart

Program Description

1. The first four instructions (Block 1 and 2) are similar to those we used previously and do not need any additional explanation.
2. Block 3 is concerned with decision making, and it is important to understand how this block is translated into Hex code. Initially, to assemble the decision-making block (JP NC, STORE), we do not know the address of the jump location. Therefore, we just label the address as STORE and leave the two memory locations (2007_H and 2008_H) for the address to be filled in later.
3. Now we can assemble the straight line segment of the flowchart (Block 4 and Block 5), the instructions shown in these blocks are self-explanatory. The critical point to remember in entering memory addresses is that the 16-bit number is entered in the reversed order—low-order byte first, followed by the high-order byte.
4. After completing the translation of Blocks 4 and 5, we can specify the address of the Jump location STORE (200DH), and fill in the blanks for LOW and HIGH bytes; $0D_H$ is entered in location 2007_H and 20_H in location 2008_H .

Write instructions to read incoming data from input port INPORT, count the number of readings, and add the readings. When the sum exceeds FF_H , stop reading the port, store the number of readings added in memory location OUTBUF, and display 01 at the output port OUTLED to indicate the overload.

Example
8.8

Solution

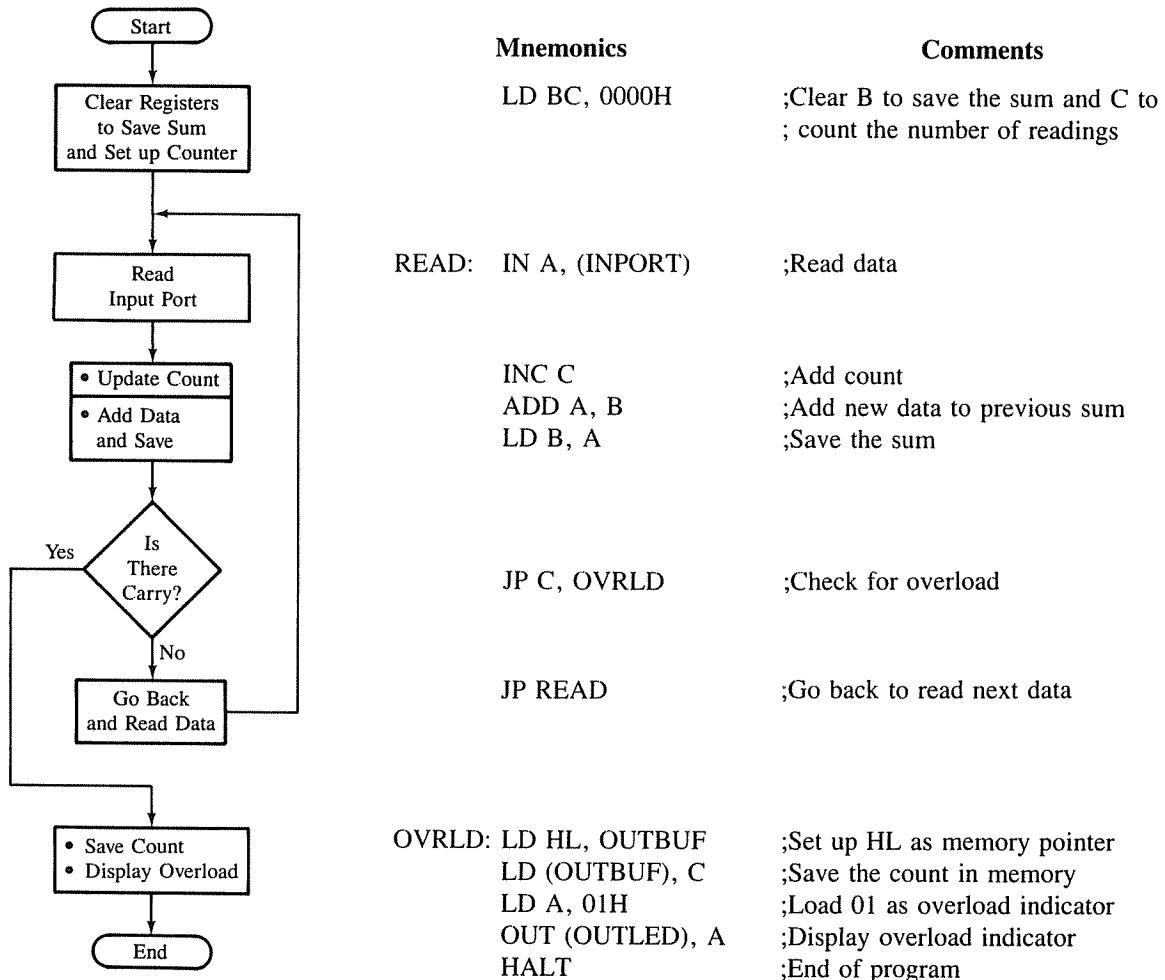


FIGURE 8.6
Flowchart

Program Description

1. This program uses two labels—READ AND OVRLD—to specify jump memory locations. Similarly, I/O ports are shown with labels: IMPORT and OUTLED. To assemble this program, these labels must be replaced by appropriate addresses.
2. Register B is used to save the sum and register C is used to count the number of readings added.
3. Initially, registers B and C are cleared. If they are not cleared in the first operation the sum and the count will have the residual contents of registers B and C.
4. The IN instruction reads the input port, and register C counts the number of data bytes read. The two following instructions add the data bytes and save the result in register B.
5. If the addition does not generate a carry, the READ loop is repeated. When the addition generates a carry, the microprocessor sets the CY flag to indicate an overload. The program jumps to location OVRLD, whereby the count is saved in memory location OUTBUF, and the overload is indicated by displaying 01_H at the output port.

8.32 Relative Jump Instructions

The Z80 instruction set includes two types of relative jump instructions: *unconditional* and *conditional*. The new address to which the program sequence is redirected is specified by an 8-bit offset (displacement) value relative to the Jump instruction. The displacement can be positive (**forward jump**), specified by the seven bits D₆-D₀ (the MSB D₇ = 0), or negative (**backward jump**) specified in 2's complement. The total offset values range from -126 to +129 bytes (explained in Example 8.9). The list of relative Jump instructions is as follows (d = displacement).

Mnemonics	Bytes	Description
JR d	2	:Jump relative unconditionally
JR Z, d	2	:Jump relative if Z = 1
JR NZ, d	2	:Jump relative if Z = 0
JR C, d	2	:Jump relative if CY = 1
JR NC, d	2	:Jump relative if CY = 0

Note: There are no Relative Jump instructions based on Sign and Parity flags.

General Characteristics

1. These are 2-byte instructions, therefore more efficient than 3-byte absolute jump instructions in terms of memory space and, in some situations, execution time.
2. Relative jumps are limited to 256 memory locations.
3. No flags are affected by these instructions.

The unconditional relative jump instruction is stored in memory locations 2100 and 2101_H, as shown below. Find the memory address of the forward jump location if the displacement byte is 7F_H, and find the memory address for the backward jump if the displacement byte is 9C_H.

Example
8.9

2100 18 JR d ;Jump relative to given offset
 2101 Offset d
 2102 Next Opcode

Solution

- When the jump instruction is executed, the program counter (PC) contains the address 2102 (PC always points to the next machine code to be fetched). By adding the displacement byte to the program counter, the address of the jump location becomes $(2102_H + 7F_H = 2181_H)$.

For an 8-bit displacement byte, 7F_H is the largest offset value for a forward jump. Therefore, relative to the memory location of the first code of the jump instruction, the maximum displacement is 7F_H plus two memory locations of the instruction. The decimal equivalent of 81_H (7F + 2) is 129; thus, the positive range extends to 129 memory locations.

- If the displacement byte is 9C_H, it must be in 2's complement because $D_7 = 1$. The memory address for the jump location is calculated by adding the displacement byte to the program counter using the 2's complement procedure.

$$\begin{array}{l}
 \text{Program Counter:} \quad \begin{array}{cccc} 2 & 1 & 0 & 2 \end{array} \\
 \text{Displacement Byte} \quad \begin{array}{c} + 9 \quad C \\ \hline \end{array} \\
 \text{in 2's Complement:} \quad \begin{array}{c} 9 \quad E \\ \hline \end{array} \\
 \\
 \text{Complement Cy and} \quad \begin{array}{cccc} 1 & 9 & E \\ \hline 2 & 0 & 9 & E \end{array} \\
 \text{subtract from } 21_H: \quad \begin{array}{c} \\ \\ \end{array}
 \end{array}$$

Memory address of the jump location is 209E_H.

Z80 Instructions Related To Index Registers

8.4

The Z80 microprocessor includes two 16-bit index registers IX and IY, and they are used primarily as memory pointers. In the previous sections, we discussed instructions concerning data copy, arithmetic, and branch operations. The Z80 can perform these operations with the contents of memory registers using the index registers.

The following group shows data copy, arithmetic, and unconditional jump instructions related to the IX registers; there is an identical set for the IY register.

Opcode	Operand	Bytes	Description
LD	IX, 16-bit	4	Load 16-bit data into IX register (this instruction was discussed in Section 8.11)
LD	(IX + d), 8-bit	4	Load 8-bit into memory location IX + d*
LD	r, (IX + d)	3	Copy from memory IX + d into register r
LD	(IX + d), r	3	Copy from register r into memory IX + d
ADD	A, (IX + d)	3	Add contents of memory IX + d to A
SUB	(IX + d)	3	Subtract contents of memory IX + d from A
INC	IX	2	Increment 16-bit contents of IX
INC	(IX + d)	3	Increment contents of memory IX + d
DEC	IX	2	Decrement 16-bit contents of IX
DEC	(IX + d)	3	Decrement contents of memory IX + d

General Characteristics

1. Index registers IX and IY are used as memory pointers. The memory address is calculated by adding the displacement byte (also known as offset) to the contents of the index register. The displacement byte is an 8-bit number; it can be either positive or negative. The magnitude of a positive offset is specified by the seven bits D₆-D₀, and the positive sign is indicated by bit D₇ being 0. For a negative offset, the displacement byte is expressed in 2's complement (illustrated in Example 8.10). The total offset ranges from + 127 to - 128 memory locations.
2. When the operand is memory, it is specified by enclosing the memory address in the parentheses (as in any other memory-related instructions), and when the operand is the index register, it is written without parentheses.
3. The instructions listed above follow the same pattern as discussed in the previous sections.
4. These instructions have 2-byte opcodes; therefore, the number of bytes in index-related instructions ranges from two to four bytes.

Example 8.10	Set up index registers IX and IY as memory pointers to locations 2050 _H and 2150 _H respectively. Load data bytes 32 _H into location 2090 _H and 97 _H into 2110 _H using the index registers. Add the bytes and save the sum in the accumulator.
---------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Solution

Mnemonics	Descriptions
LD IX, 2050H	;Point index IX to location 2050H
LD IY, 2150H	;Point index IY to location 2150H

*d is an offset value added to the contents of the index register to obtain in the memory location (see Example 8.10)

LD (IX + 40H), 32H	;Load byte into location (2050H + 40H) = 2090H
LD (IY + C0H), 97H	;Load byte into location 2110H
	Offset is (2110H - 2150H) 40H locations
	backward. 2's complement of 40H = C0H.
LD A, (IX + 40H)	;Copy first byte (32H) into A
ADD A, (IY + C0H)	;Add second byte
HALT	

The memory addresses are calculated by adding the offset to the low-order byte of the index register.

$$\begin{array}{r}
 \text{IX} + 40 = 20 \quad \text{IY} + \text{C}0 = 21 \\
 + \quad \frac{40}{90_H} \rightarrow 2090_H \quad + \quad \frac{C0}{10_H} \\
 \hline
 \end{array}$$

Because the second operation is a 2's complement addition, the carry is complemented.

PROGRAMMING TECHNIQUES: LOOPING, COUNTING, AND INDEXING

8.5

The examples illustrated in the previous sections are simple and can be solved manually. However, a computer is at its best, surpassing human capability, when it has to repeat such tasks as adding a large set of numbers or copying bytes from one block of memory locations to another. It is fast and accurate.

To perform a given repetitive task, commonly used techniques are looping, counting, and indexing. To add data bytes stored in memory, for example, the following steps are necessary.

1. Define the task to be repeated: **looping.**

A loop is set up by using either a conditional Jump or an unconditional Jump as illustrated in Examples 8.7 and 8.8.

2. Specify how many times the task is to be repeated: **Counting.**

The counter is set by loading a count (number of times the task is to be repeated) into a register or a register pair, and the counting is done by decrementing the count every time the loop is repeated. The counter can also be set up to count from 0 to the final count using increment instructions.

3. Specify the location of the data: **Indexing.**

The starting location of the data can be specified by loading the memory address into a register pair and using the register pair as a memory pointer or index.

4. Indicate the end of the repetitive task: **Setting Flags.**

The end of repetition is indicated by the flag of the conditional jump instruction. When the condition is true, the loop is repeated, and when the condition is false, the loop execution is terminated, and the execution goes to the next instruction in memory.

These steps are further clarified in Example 8.11.

Example 8.11

Draw a general flowchart to add ten bytes of data stored in memory starting at a given location, and display the sum. Explain the blocks in the flowchart.

Solution

To draw a flowchart, the problem must be divided into steps as follows:

1. Set up a counter to count the number of bytes.
Set up a memory pointer (index) to locate where data bytes are stored.
Clear a register if necessary (either to store partial results or count the number of carries).
2. Transfer data from memory to the microprocessor.
3. Perform addition, checking for carry.
4. Save the partial result.
5. Update the counter and the memory pointer for the next operation.
6. Check the flag to indicate the completion of the task. If the condition is true, repeat the task; otherwise go to the next instruction.
7. Display or store the result.

These steps and their sequence can be represented in the form of a flowchart as shown in Figure 8.7.

Blocks

1. Initialization

This is a planning stage where all initial conditions and requirements are defined. In our example, this block should set up a counter, memory index (pointer), carry register, and temporary storage register.

2. Data Acquisition

Data are generally stored in memory or read from an input port. This step is concerned with bringing data into the microprocessor.

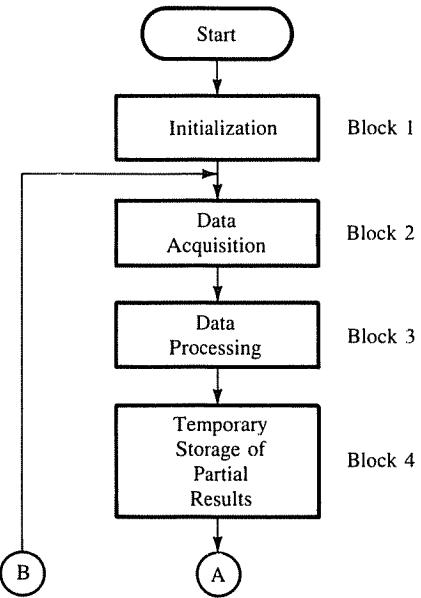
3. Data Processing

This step involves data manipulation, such as arithmetic or logical operations. In the example, we add a data byte, check for a carry, and update carry register if necessary.

4. Temporary Storage

This step involves storing of partial results so that the previous result will not be destroyed by the next data processing operation.

FIGURE 8.7
Generalized Programming Flowchart



5. Getting Ready for Next Operation

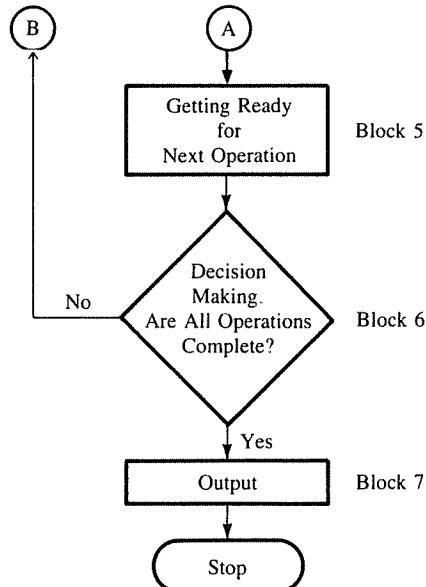
Before we can check whether the task is completed, we need to update the initial conditions; the index and the counter should be incremented or decremented.

6. Decision Making

In this step, the flag is checked. if the condition is true, the loop is repeated; otherwise, the program goes to the next block to display the result.

7. Output

In this Block, the result is either sent to an output port or stored in memory.



LOOKING AHEAD

In the previous sections, we introduced three groups of instructions: data copy, arithmetic, and branching. These instructions were illustrated with examples. In the last section, we discussed the programming techniques with the generalized flowchart (Figure 8.7). Now we will illustrate two programs using the instructions and the programming techniques that were introduced and discussed. We will attempt to analyze the programming problems in terms of the blocks shown in Figure 8.7 and modify these blocks if necessary.

ILLUSTRATIVE PROGRAM 1: BLOCK TRANSFER OF DATA BYTES

8.6

In practical applications, data transfer from one memory block to another is a common occurrence. This illustrative program demonstrates how to copy data bytes from one block of memory to another using the instructions discussed previously.

8.61 Problem Statement

One hundred bytes of data are stored in a block of memory with the starting location labelled as SOURCE. Transfer all data to a new block starting with the location labelled as OUTBUF (Output Buffer). When the data transfer is complete, display 01 at the output port PORT0.

8.62 Problem Analysis

We can analyze this problem in terms of the generalized flowchart (Figure 8.7).

1. **Initialization:** In this problem, we need one counter to count 100 bytes and two memory pointers: one for SOURCE memory and the other for OUTBUF memory.

2. *Data Acquisition:* In this problem, when a data byte is transferred from memory to the microprocessor, it is immediately transferred to a new memory location. There is no data processing; thus, we can eliminate Blocks 3 and 4.

The flowchart is shown in Figure 8.8; Blocks 5, 6 and 7 are identical to the blocks shown in Figure 8.7.

8.63 Program and Flowchart

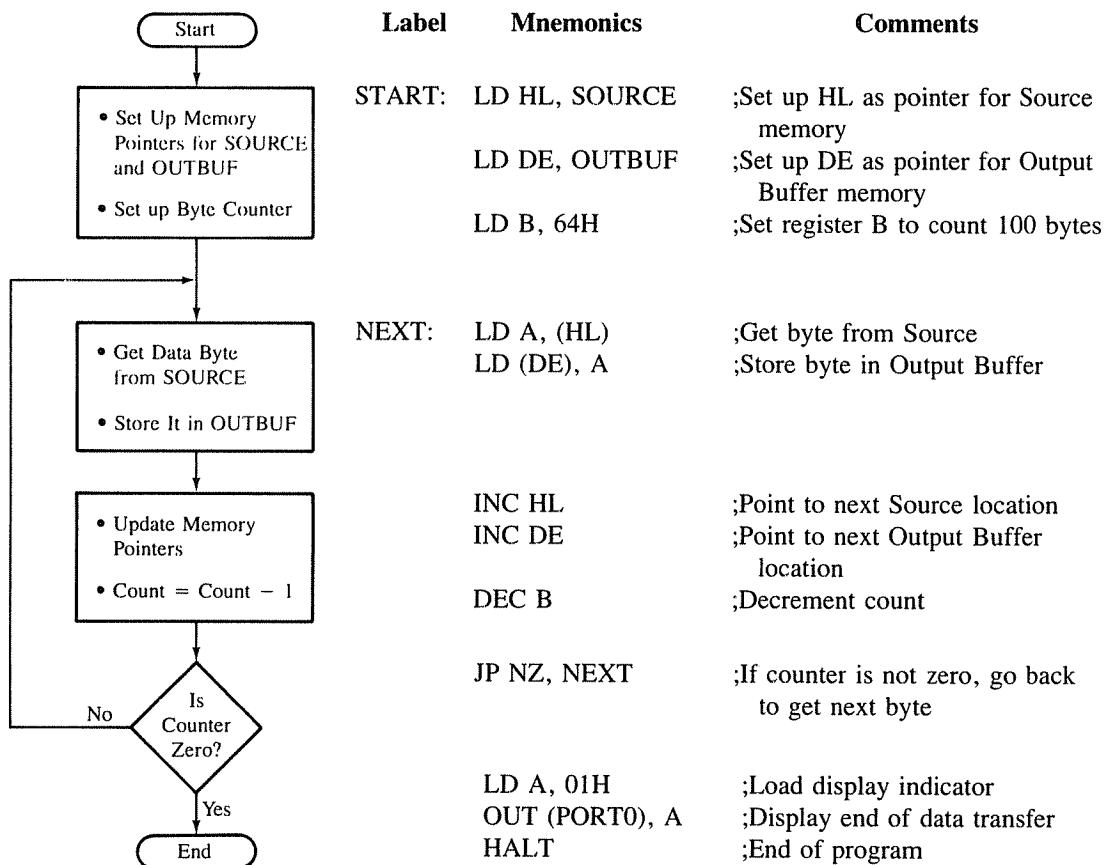


FIGURE 8.8

Flowchart: Block Transfer of Data Bytes

8.64 Program Description and Execution

In this program, several labels are used to specify memory locations and I/O ports; this is a common industrial practice. When an assembler is used to write programs, labeling provides convenience and flexibility. In manual assembly, labels make it easy to read a

program. In this problem, we need to specify or define absolute values of the labels SOURCE, OUTBUF, AND PORT0, as well as the label START, the location where the program begins. The memory address of the label NEXT depends on the starting address of the program, and in manual assembly, it can be calculated by counting the number of bytes of each instruction written before NEXT. If we assume the starting address of the program is 2000_H , the memory address of NEXT will be 2008_H .

The flowchart in Figure 8.8 is similar to the generalized flowchart of Figure 8.7. In the first block, registers HL and DE are used as memory pointers and register B as a counter to count 100 bytes. In the next block, a byte is transferred from SOURCE memory to the accumulator using HL as the memory pointer, and the same byte is stored in OUTBUF memory using DE as the memory pointer.

The statements shown in the next block update the memory pointers and the counter. These statements may appear strange as algebraic equations; in fact, they are not algebraic statements but value assignments. The statement $\text{Count} = \text{Count} - 1$ means the new value is obtained by decrementing the previous value at the completion of one loop. It is important to remember that updating should be done before the decision making because once the Jump instruction finds the Zero flag not set, the program execution will go back to location NEXT. This loop is repeated until the counter $B = 0$, and then the data transfer is indicated by displaying 01 at PORT0: this is shown as End in the flowchart.

When you execute the program on a single-board system, the successful completion of data transfer can be checked by verifying the contents of some locations in SOURCE memory and the corresponding memory locations in OUTBUF memory.

ILLUSTRATIVE PROGRAM 2: ADDITION WITH CARRY

8.7

The following program adds the number of bytes stored in memory and counts the number of carries generated. The maximum sum can be up to 16-bit.

8.71 Problem Statement

Add the following ten data bytes stored in memory with the starting address INBUF (Input Buffer). Store the sum in two memory locations; the low-order byte of the sum should be stored in OUTBUF and the high-order byte in OUTBUF + 1.

Data (H): A2, 37, 4F, 97, 22, 6B, 75, 8E, 9A, C7.

8.72 Problem Analysis

This problem is similar to Example 8.11 and can be very easily analyzed in terms of the blocks shown in the generalized flowchart in Figure 8.7.

1. In the initialization block, we need to set up a counter to count ten bytes, a memory pointer for INBUF, and registers to save the partial sum and carries. We use the accumulator for addition. The memory pointer for OUTBUF is not necessary until the data processing is completed; thus, the memory pointer used for INBUF can be also used for OUTBUF.

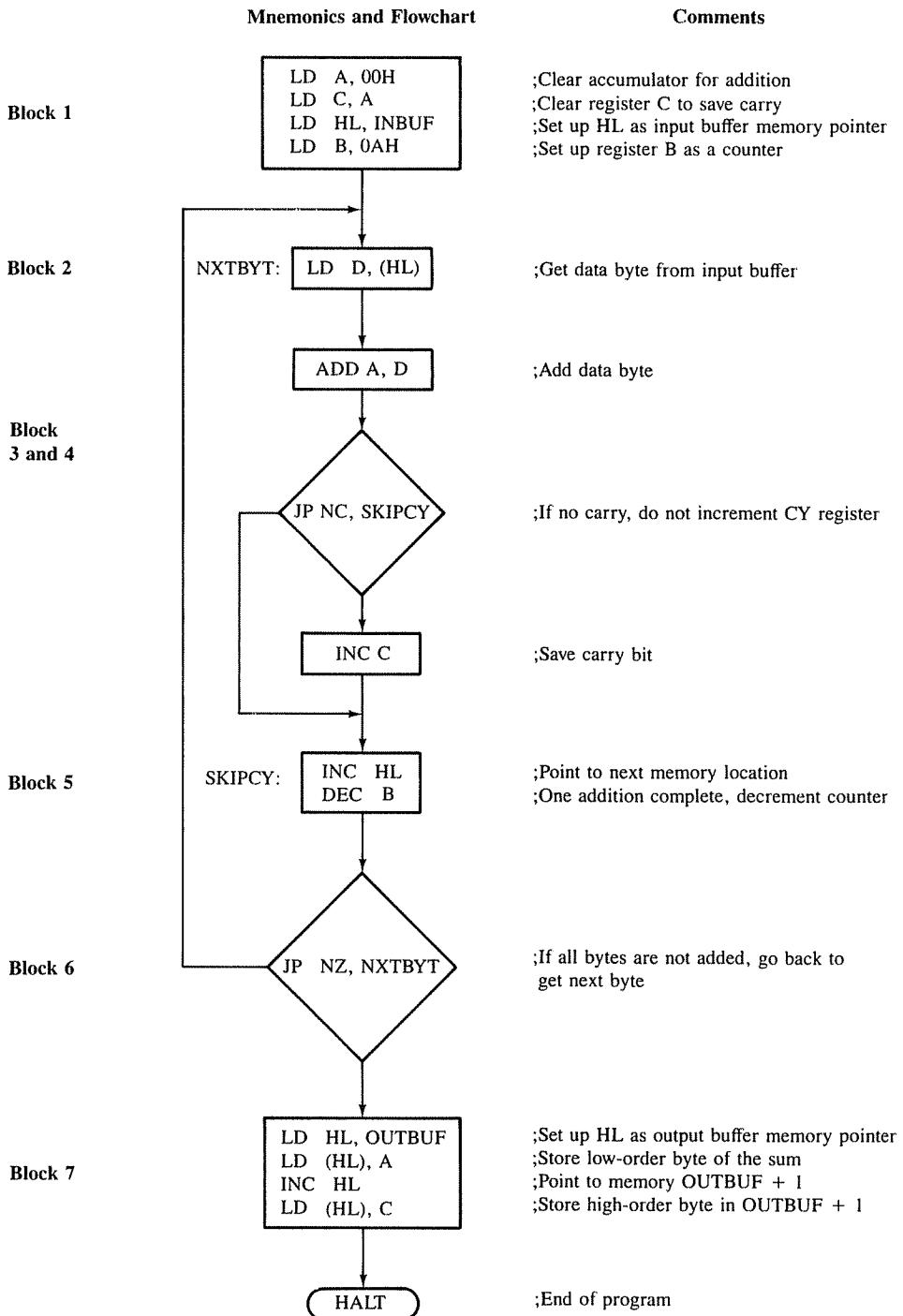


FIGURE 8.9
Adding Ten Bytes: Program and Flowchart

2. In this problem, the data processing block needs to be expanded because of the carries. Whenever a carry is generated after an addition, the carry register will be incremented; thus, the high-order byte of the sum will be saved in the carry register and the low-order byte will be in the accumulator.

8.73 Program Description and Execution

The comments written in the program (see Figure 8.9) explain the function of each instruction, and the flowchart drawn around the instructions shows the sequence of execution.

In the initialization block, the accumulator and register C are cleared for use in arithmetic operations; otherwise, residual data would cause erroneous results. However, register D need not be specifically cleared because the first load instruction replaces its residual data.

This program has two types of loops; one loop repeats the addition-related instructions if the counter is not zero, and the second loop skips the carry counter if there is no carry. The instruction ADC (Add with Carry) is inappropriate for this problem; this instruction can be used for 16-bit addition (See Appendix A).

In the output block, the HL register is used again as a memory pointer for the output buffer memory. After all bytes have been added, the low-order byte of the result, which is in the accumulator, is stored in the memory location OUTBUF, and the high-order byte (carries) in register C is stored in the next memory location OUTBUF + 1.

8.8

DEBUGGING A PROGRAM

Debugging a program is similar to troubleshooting hardware, but is much more difficult and cumbersome. When a program 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 code, and execution.

The debugging procedure can be divided into two parts: **static debugging** and **dynamic debugging**. Static debugging is similar to visual inspection of a circuit board; it is the paper-and-pencil check of a flowchart and machine code. Dynamic debugging involves observing outputs, register contents, and flags following the execution of either instruction (the single-step technique) or a group of instructions (the breakpoint technique).

8.81 Static Debugging of Machine Code

Translating the assembly language into the machine code is similar to building a circuit from a schematic, in that the machine code will have errors just as would the circuit board. If an assembler is used to translate the code, most of the errors involved in hand assembly can be eliminated. The following errors are common in manual assembly:

1. Selecting a wrong code.
2. Forgetting the second byte 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 debugging problems given in the Assignments section at the end of the chapter will illustrate some of these errors.

8.82 Dynamic Debugging

Dynamic debugging is concerned with observations of data after executing an instruction or set of instructions. These observations may include verifying output displays, checking flags, examination of register contents, and tracing execution flow. The process is similar to that of the signal-injection technique in troubleshooting hardware, which involves injecting a signal into a hardware system and checking signals at various points against the expected outputs. Similarly, in debugging programs, we execute a few instructions and check register contents or outputs against the expected results. The commonly used techniques and tools are (1) **Single Step**, (2) **Register Examine**, and (3) **Breakpoint**.

SINGLE STEP

The single step technique allows us to execute one instruction at a time and to observe the results following each instruction. As we advance through each instruction, we will be able to observe memory addresses and codes as they are executed. With the single step technique, we can spot

- Incorrect addresses.
- Incorrect jump locations for loops.
- Incorrect data or missing codes.

This technique is generally used in conjunction with the Register Examine facility (described below), and it is very useful for short programs (50-100 machine codes). For larger programs, the technique is cumbersome and time consuming.

REGISTER EXAMINE

The Register Examine facility allows us to examine the contents of the microprocessor registers and the flags. We can examine registers after the execution of each instruction or after the execution of a group of instructions and compare the contents with the expected outcomes.

BREAKPOINT

The breakpoint technique allows us to check the program in segments. We can set a breakpoint at the end of a program segment or multiple breakpoints at various memory locations. When the microprocessor is asked to execute the program, it executes the codes until it comes across the first breakpoint, where it returns the control to the breakpoint subroutine in the system. At this point, we can examine the registers for expected results. If the segment of the program is found satisfactory, the program can be executed up to the next breakpoint. With the breakpoint technique, we can isolate the segments of the pro-

grams with errors and debug those segments with the single step technique. The breakpoint technique is generally used to check out timing loops, I/O sections, and interrupts.

COMMON SOURCES OF ERRORS

In addition to the errors mentioned in Section 8.81, here is a list of errors of common occurrence in the types of programs discussed in this chapter.

- Failure to clear the accumulator when it is used to add data.
- Failure to clear registers when they are used to store partial results or carries.
- Failure to update an index or a counter.
- Failure to set a flag before using a conditional Jump instruction or use of an inappropriate flag.
- Inadvertently changing a flag before using a Jump instruction.

Z80 SPECIAL INSTRUCTIONS

8.9

The Z80 instruction set includes some instructions that perform more than one task. These instructions improve programming efficiency considerably. Some of these instructions are as follows:

Mnemonics	Description
DJNZ d	Decrement B and Jump Relative on no zero (Z = 0) The instruction decrements register B, and if B \neq 0, it jumps to memory address specified by the offset value d.
LDI	Load and Increment The instruction copies a data byte from the memory location shown by HL into the memory location pointed to by DE. Registers HL and DE are incremented and BC is decremented.
LDIR	Load, Increment, and Repeat This is similar to the instruction LDI, except that it is repeated until BC = 0.
LDD	Load and Decrement The instruction copies a data byte from the memory location shown by HL into the memory location pointed to by DE. Registers HL, DE, and BC are decremented.
LDDR	Load, Decrement, and Repeat This instruction is similar to LDD, except that it is repeated until BC = 0.

Example 8.12 Modify the illustrative program Addition With Carry (Section 8.7) using the instruction DJNZ and the offset value.

Solution The following mnemonics are repeated from a segment of the program in Figure 8.9; we assume that the segment is stored in memory locations starting from 2008_H.

Location	Label	Mnemonics	Comments
2008	NXTBYT:	LD D, (HL)	;Get data byte from input buffer
2009		ADD A, D	;Add data byte
200A		JP NC, SKIPCY	;If no carry, do not save CY
200D		INC C	;Save carry bit
200E	SKIPCY:	INC HL	;Point to next memory location
200F		DJNZ F7H	;Decrement counter B, and if B ≠ 0, jump to location 2008 to get the next byte

Program Description and Calculation of the Offset Value In this program, the instruction DJNZ replaces two instructions—DEC B and JP NZ, NXTBYT—from the program in Figure 8.9. The instruction DJNZ assumes that register B is used as a counter. When the Z80 executes the 2-byte instruction DJNZ, the program counter holds the address 2011_H. The offset value for the jump location NXTBYT (2008_H) is obtained as follows:

Program Counter:	2	0	1	1
Jump Location:	2	0	0	8
2's Complement of 09 _H :	$\begin{array}{r} 09_H \\ - 2008_H \\ \hline F7_H \end{array}$ $(0\ 0\ 0\ 0\ 1\ 0\ 0\ 1)$ $(1\ 1\ 1\ 1\ 0\ 1\ 1\ 1)$			

8.10

ILLUSTRATIVE PROGRAM 3: BLOCK TRANSFER OF DATA BYTES USING Z80 SPECIAL INSTRUCTIONS

This program transfers data from one memory block to another using the Z80 instruction LDIR.

8.101 Problem Statement

Modify the illustrative program (Section 8.63) using the instruction LDIR. The problem statement from the previous program is as follows: Transfer 100 bytes from the memory block SOURCE to the memory block OUTBUF and indicate the end of data transfer by displaying 01 at PORT0.

8.102 Problem Analysis

To use the instruction LDIR, the HL register should be used to point to memory SOURCE and the DE register to the destination OUTBUF. Even if the total number of bytes to be transferred is an 8-bit number (64_H), the register BC should be used as the counter with the 16-bit count (0064_H).

8.103 Program

Label	Mnemonics	Comments
START:	LD HL, SOURCE	;Set up HL as pointer for SOURCE memory
	LD DE, OUTBUF	;Set up DE as pointer for OUTBUF memory
	LD BC, 0064H	;Specify the number of bytes in BC
	LDIR	;Transfer data byte from SOURCE to OUTBUF and repeat until BC = 0
	LD A, 01H	;Load display indicator
	OUT (PORT0), A	;Display end of data transfer
	HALT	;End of program

8.104 Program Description

In this program, the instruction LDIR is the workhorse; it replaces several instructions from the illustrative program in Section 8.63. The instruction performs three operations: (1) copies a data byte from the memory location pointed to by the HL register into the memory location indicated by the DE register, (2) updates memory pointers (HL and DE) and the counter (BC), and (3) makes the decision to repeat or terminate the loop based on the count in the BC register. When all 100 bytes are copied into new locations, the counter BC becomes zero, and the program goes on to display 01 at PORT0.

SUMMARY

In this chapter, we illustrated a group of instructions from the Z80 set frequently used in writing programs. Instructions were selected from three groups: data copy, arithmetic, and branch. These instructions range from 1-byte to 4-byte in length.

General characteristics of these instructions are as follows:

1. The data copy and load instructions copy the contents of the source into the destination without affecting the source contents. They do not affect the flags.
2. The arithmetic instructions (with some exceptions) assume one of the operands is the accumulator, and the result of an operation is usually stored in the accumulator. Most of these instructions affect the flags.
3. The conditional Jump instructions are decision-making instructions and executed according to the status of the flags. Not all instructions affect the flags; in particu-

lar, the data copy instructions and 16-bit increment/decrement instructions do not affect the flags.

4. The Z80 microprocessor includes two index registers (IX and IY), which are used primarily as memory pointers. The instructions related to index registers have 2-byte opcodes and perform data copy and arithmetic operations with the contents of memory registers.

Programming techniques such as looping, counting, and indexing were discussed and a generalized flowchart was illustrated. Two illustrative programs were discussed in the context of this generalized flowchart.

Finally, some Z80 special instructions were introduced. These instructions perform multiple tasks; thus, they improve programming efficiency.

ASSIGNMENTS

Note: In the following assignments use your own data if data are not given.

Section 8.1

1. Write mnemonics to load 39_H into register B and 92_H into register D. Save the contents of B in register L and display the contents of D at PORT1.
 2. Write instructions to load 47_H into register B and $F2_H$ into register C using one instruction. Store the contents of C in memory location 2080_H and display the contents of B at PORT1. Assemble the Hex code and store the code in memory locations starting from 2000_H .
 3. Write instructions to load $A2_H$ into register D and 2080_H into register HL. Copy the contents of D into memory location 2080_H .
 4. Write instructions to load $A7_H$ into register D and 2055_H into register BC. Copy the contents of D using BC as a memory pointer.
 5. The memory location 2040_H contains 98_H and 2070_H contains $F7_H$. Write instructions to exchange the contents of these memory locations and assemble the Hex code.
 6. Specify the register contents and the flag statuses after execution of the following instructions (show only changes):

Registers						Flags	
A	B	C	H	L	Z	CY	
34	7F	FF	01	00	0	1	(Initial Conditions)
LD A, 00H							
LD BC, 8058H							
LD B, A							
LD HL, 2040H							
LD L, C							
LD (HL), A							
HALT							

7. Write instructions to read the input port 80_H and output the reading to the port 05_H (See Figure 8.10). What appliances will be turned on with this output?
8. Write comments to explain the functions of the following instructions:

```

LD HL, 2065H
LD (HL), 00H
HALT

```

Section 8.2

9. What are the contents of the accumulator after the execution of the instruction $SUB A$? Specify the status of the Z and CY flags.
10. Write the instructions to load FF_H into the accumulator and increment A. Specify the status of the S, Z, and CY flags after the execution of the increment instruction.
11. In the previous assignment (#10), replace the increment instruction with the instruction $ADD A, 01H$ and explain how the flags S, Z, and CY are affected after the addition.

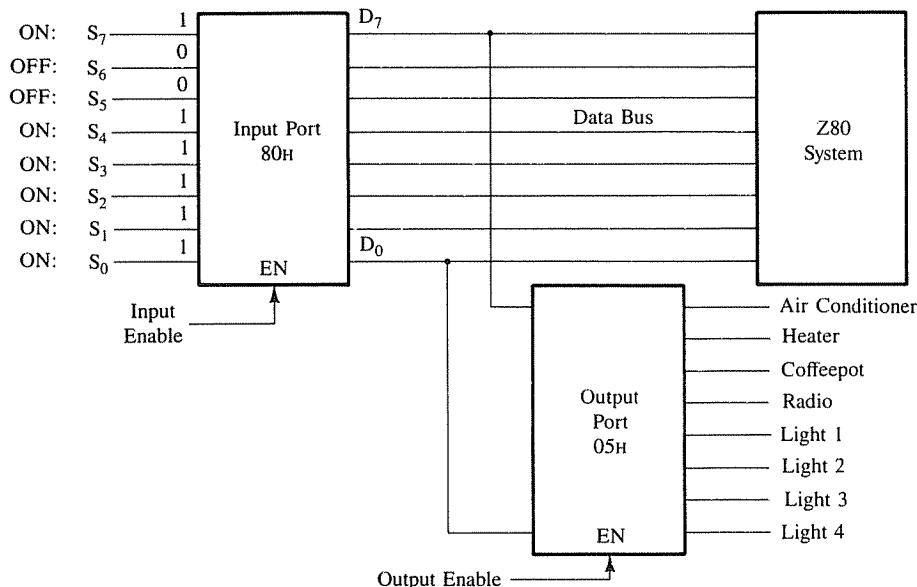


FIGURE 8.10
Appliance Control

12. Specify the register contents and the flag status after the execution of the following instructions. What is being displayed at PORT1?

	Register Contents			Flags		
	A	B	C	S	Z	CY
	FF	77	89	1	0	1 (Initial Conditions)
SUB A						
LD B, A						
ADD A, A9H						
LD C, 57H						
ADD A, C						
DEC A						
OUT (PORT1), A						
HALT						

13. Write instructions to load $40FF_H$ into register HL and increment HL. Specify the contents of register HL.
14. Register HL contains $20FF_H$. What are the contents of register HL if the byte 01_H is added (not incremented) to register L? What are the statuses of the S, Z, and CY flags after the addition?
15. Write instructions to perform the operations listed in 14, and assemble the code showing memory addresses.
16. Show the contents of the registers and the memory locations that are affected after the execution of the following instructions. Explain the difference between the two INC instructions shown below.

```
LD (209FH), FFH
LD HL, 209FH
INC (HL)
INC HL
HALT
```

17. Find the results of the following operations and explain the difference between the two results.

SUB A	SUB A
LD HL, 971FH	LD HL, 971FH
LD BC, 8F9CH	LD BC, 8F9CH
ADD A, L	ADD A, L
ADD A, C	ADD A, C
ADD A, B	SUB H
SUB H	ADD A, B
HALT	HALT

Section 8.3

18. Load $48A2_H$ into register BC. Subtract the contents of C from B. If the answer is in 2's complement, display 01_H at PORT1; otherwise, display the result. Assemble the code and execute the program.
19. Execute the program in 18 by loading $F247_H$ in register BC.
20. Three data bytes are stored in memory locations 2050 , 2051 , and 2052_H . Write instructions to subtract the bytes stored in memory locations 2050 and 2051 from the byte stored in location 2052_H . If the answer is in 2's complement, display FF_H at PORT1; otherwise, display the answer. Execute the instructions with the following set of data in Hex.

Set 1: $2050 = 32$, $2051 = 78$, $2052 = F9$

Set 2: $2050 = 67$, $2051 = 98$, $2052 = F9$

21. The Relative Jump instruction $JR NZ, 68H$ is stored in memory locations $20A7_H$ and $20A8_H$. Calculate the jump location.
22. If the opcode of the Relative Jump instruction $JR NC, 8FH$ is located at memory location 2050_H , calculate the jump location.
23. Assemble the code in Illustrative Program 1 (Section 8.63) and replace the instruction $JP NZ, NEXT$ with the appropriate Relative Jump instruction and offset.
24. In Illustrative Program 2 (Figure 8.9), assemble the code and replace the Jump instructions $JP NC, SKIPCY$; and $JP NZ, NXTBYT$ with the appropriate Relative Jump instructions and their offsets.

Section 8.4

25. Rewrite the instructions in Figures 8.2 (a), (b), and (c) using the index registers IX and IY as memory pointers.
26. Write instructions to load 2070_H into the IY index register. Using the register IY as a memory pointer with appropriate offsets, store the bytes $A2_H$ and 32_H in memory locations $204F_H$ and $209F_H$, respectively.
27. Calculate the value of the memory pointer if register IX contains 2000_H with the displacement byte 80_H .
28. Calculate the values of two memory pointers if register IY contains $20FF_H$ and it is combined with the displacement bytes $7F_H$ and $8F_H$.
29. Assuming the index register IX contains 2050_H , explain the difference between the instructions $INC IX$ and $INC (IX + 0)$.
30. Rewrite Illustrative Program 1 (Section 8.6), Block Transfer of Data Bytes, using the index registers as memory pointers.

Section 8.5

31. Draw a flowchart to add the numbers stored in memory location INBUF (Input Buffer). When the result generates a carry, subtract the last byte and display the sum.

32. Modify the above program to count and display the number of bytes added (excluding the last one).
33. You are given a long grocery list and asked to buy the items from number 20 to 47. Any item that costs more than \$10.00 should be excluded. Add up the total cost and show the total expenses. Draw a flowchart for performing these tasks.
34. Modify the above flowchart to include a ceiling of \$100 on total expenses.
35. Draw a flowchart to add the string of numbers stored in memory locations BUFFER. The end of the string is indicated by the number 00. Display the sum.

Section 8.6

36. The following block of data is stored in memory locations INBUF. Transfer the data to the locations OUTBUF in the reverse order.
Data (H) 47, 97, F2, 9C, A2, 98
37. Ten bytes are stored in memory locations starting from INBUF. To insert an additional five bytes at the beginning locations, it is necessary to shift the first ten bytes by five locations. Write a program to shift the data string by five memory locations.
38. Ten 16-bit readings are stored in memory locations SOURCE; the low-order byte is stored first, followed by the high-order byte. Write a program to copy the low-order bytes only to a new location BUFFER.
39. Given the initial conditions in 38, eliminate the high-order readings and store the low-order readings in consecutive memory locations SOURCE.

Section 8.7

40. Draw a flowchart to modify Illustrative Program 2 (Section 8.7) to include the instruction Jump On Carry instead of Jump On No Carry (JP NC, SKIPCY). You may have to use an additional Jump instruction, and the flowchart may have to be altered significantly.
41. Modify Illustrative Program 2 (Section 8.7) using the DE register as a memory pointer instead of HL.
42. Modify Illustrative Program 2 (Section 8.7) using the DE register as a memory pointer and a memory location as a counter (instead of register B).
43. Write a program to add the following string of data bytes until a carry is generated. When the Carry flag is set, subtract the last byte added and display the sum at PORT1.
Data (H) 89, 32, 2B, 7A, B5, 68, 2F, . . .
44. Modify the previous program to count the number of bytes added (excluding the byte that generates the carry) and display the count at PORT2.
45. Ten 16-bit readings are stored in memory locations SOURCE; the low-order byte first, followed by the high-order byte. Write a program to add the low-order bytes. Display the sum at two different ports and store the sum in two memory locations OUTBUF and OUTBUF + 1.
46. Two sets of data, ten bytes each, are stored in memory locations INBUF1 and

INBUF2. Subtract each data byte stored at INBUF2 from the corresponding data byte at INBUF1. Add the remainders, and if the sum of the remainders generates a carry, display FFH at PORT1; otherwise, display the sum at PORT1.

Section 8.8

47. Find the errors in the following instructions.

a. The following instructions add two Hex bytes (06 and 52) and display the sum at PORT7.

2000	06	LD B, 06H	;Load data bytes
2001	06		
2002	0E	LD C, 52H	
2003	52		
2004	80	ADD A, B	;Add data bytes
2005	81	ADD A, C	
2006	D3	OUT (07H), A	;Display the sum
2007	76	HALT	

b. The following instructions add five bytes stored in memory locations starting from 2050_H . The sum will be less than FF_H .

2000	9F	SUB A	;Clear A
2001	21	LD HL, 2050H	;Set up HL as memory index
2002	20		
2003	50		
2004	78	LD B, 05H	;Set up B as a counter
2005	05		
2006	86	ADD A, (HL)	;Add byte
2007	23	INC HL	;Point to next byte
2008	05	DEC B	;Reduce count
2009	D2	JP NZ, 2004H	;If B \neq / 0, get next byte
2010	04		
2011	20		
2012	76	HALT	;End of program

c. The following program transfers a 100_H bytes of data starting from the memory location 2100_H to a new location starting from 2800_H .

LD HL, 2100H	;Set up HL as index for Source
LD BC, 2800H	;Set up BC as index for new memory
LD DE, 0100H	;Set up DE as counter
NEXT: LD A, (HL)	;Get byte

```
LD (BC), A      ;Transfer byte to new memory
INC HL          ;Update indexes and counter
INC BC
DEC DE
JP NZ, NEXT    ;If transfer is not complete, go back
                ;and get next byte
HALT          ;End of data transfer
```

Section 8.9

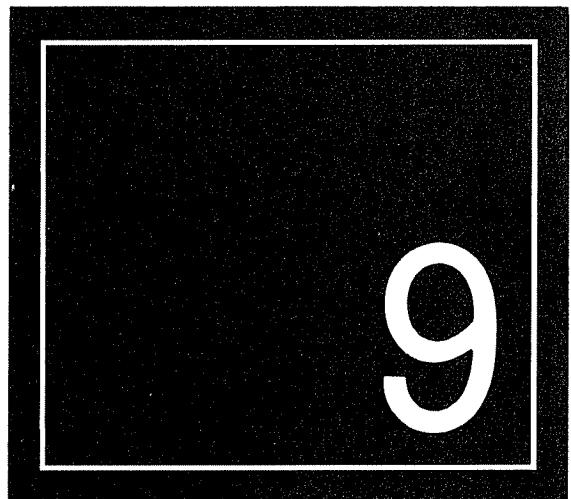
48. A data set with 512 bytes is stored in memory locations with the starting address INBUF1 (2100_H). Shift the entire data set by 256 locations with the starting address INBUF2 on the next page (2200_H). Use the instruction LDDR.
49. Rewrite Illustrative Program 2 (Section 8.7) using the instructions DJNZ and JR NC.
50. Rewrite Illustrative Program 1 (Section 8.6) using the index registers IX and IY as memory pointers and the instruction DJNZ.

Logic and Bit Manipulation Instructions

The microprocessor is a programmable logic device; it can perform all the logic functions of hardware gates, such as AND, OR, and Ex-OR (exclusive-OR). It can compare two bytes and indicate the comparison (less than, equal to, or greater than) by setting appropriate flags. In addition, it can rotate and shift bytes, and manipulate individual bits.

In this chapter, Z80 instructions related to logic and compare operations and bit manipulation are introduced. These instructions are illustrated with examples, and their applications are shown in two illustrative programs, one of which demonstrates how to design time delays using software instructions. This chapter also includes a section on debugging, which lists errors that commonly occur in writing these types of programs and suggests debugging techniques using a counter program.

Finally, this chapter introduces Z80 special instructions related to the compare operations. These special instructions perform multiple tasks such as comparing two bytes, updating registers that are used as memory pointers and a counter, and making a decision to change the program sequence. For example, one of the Compare instructions can search for a specific byte in a given memory block.



OBJECTIVES

- Explain how logic instructions (AND, OR, and XOR) perform their operations and how flags are affected by these instructions.
- Write a set of instructions to illustrate logic operations and explain how these instructions are used in masking, setting, and resetting bits.
- Explain how the Compare instructions perform a comparison and modify flags to indicate the comparison of two bytes.
- Explain the Rotate instructions and their effects on the contents of the accumulator and the CY flag.
- Write a set of instructions (programs) to illustrate the use of Compare and Rotate instructions.
- Explain how a specific bit in a register or memory can be checked and set or reset by using bit manipulation instructions.
- Write a program to set/reset specific bits at a given interval.
- Explain the Z80 special instructions related to search and compare operations.
- Write a set of instructions to illustrate these Z80 special instructions and explain their advantages.

9.1 LOGIC AND COMPARE OPERATIONS

The microprocessor is basically a programmable logic chip. It can perform all the logic functions of the hard-wired logic through its instruction set. However, the logic operations are slightly different from the hard-wired logic. The AND gate shown in Figure 9.1(a) has two inputs and one output. On the other hand, in an 8-bit microprocessor, the AND instruction simulates eight 2-input AND gates. Figure 9.1(b) shows ANDing the contents of register B with the contents of the accumulator. Register B contains 77_{16} and the accumulator has 81_{16} . After ANDing, the result (01_{16}) is stored back into the accumulator. The other logic functions are performed similarly. In the following sections, we discuss instructions related to AND, OR, and XOR logic functions; the instruction related to the NOT function was discussed as 1's complement in the last chapter.

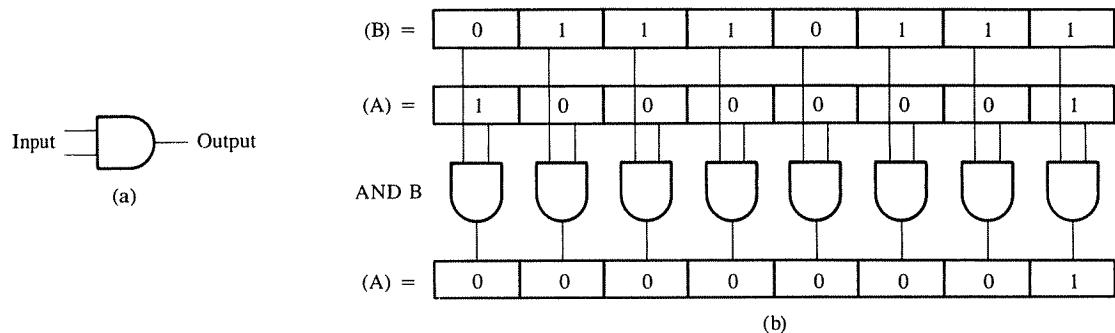


FIGURE 9.1

(a) AND Gate, and (b) a Simulated AND Instruction

9.11 Logic AND, OR, and XOR Instructions

Logic instructions are examples of implied addressing; the implied operand is the 8-bit word of the accumulator. The other operand can be an 8-bit data word or the contents of a register or memory. When the second operand is a memory location, it can be specified either by the 16-bit number in the HL register or the 16-bit number in an index register with an offset byte. (See Appendix A for complete descriptions with examples of these instructions.)

Opcode	Operand	Bytes	Description
AND	r	1	AND contents of a register with the accumulator
AND	8-bit	2	AND 8-bit data with the accumulator
AND	(HL)	1}	AND contents of memory with the accumulator
AND	(IX + d)*	3}	AND contents of memory with the accumulator
OR	r	1	OR contents of a register with the accumulator
OR	8-bit	2	OR 8-bit data with the accumulator
OR	(HL)	1}	OR contents of memory with the accumulator
OR	(IX + d)*	3}	OR contents of memory with the accumulator
XOR	r	1	Exclusive OR contents of a register with the accumulator
XOR	8-bit	2	Exclusive OR 8-bit data with the accumulator
XOR	(HL)	1}	Exclusive OR contents of memory with the accumulator
XOR	(IX + d)*	3}	Exclusive OR contents of memory with the accumulator

General Characteristics

These logic instructions

1. implicitly assume that the accumulator is one of the operands.
2. reset (clear) Carry (CY) flag and modify S, Z, and P/V 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 or memory.

Figure 9.2 shows an input port (PORT1) with three switches connected to data lines D₀, D₁, and D₂; when a switch is on, it provides logic 1 to the respective data line. Write instructions to read the port and save the reading in memory location INBUF.

Example
9.1

Solution

```

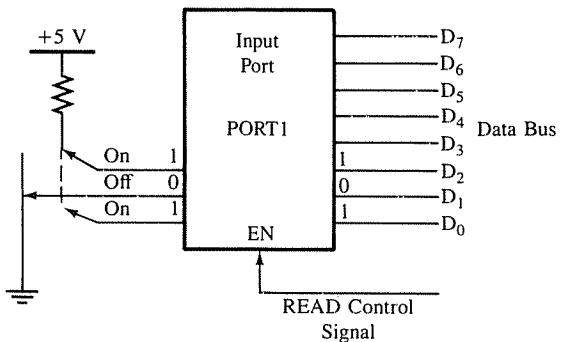
IN A, (PORT1)      ;Read the switch positions
AND 07H            ;Mask data bits D3-D7
LD (INBUF), A      ;Store the reading in memory INBUF
HALT

```

*The similar instructions related to the index register IY are not shown here.

FIGURE 9.2

Reading Switches from an Input Port



The first instruction reads the switch positions at PORT1. Even if only three switches are connected, the reading will be 8-bit data; bits D₃-D₇ will be random. Therefore, bits D₃-D₇ should be masked or eliminated without affecting the switch positions. This is accomplished by ANDing the input reading with an appropriate masking byte (07_H). The masking byte is obtained by placing 0s in bit positions that are to be masked and by placing 1's in bit positions where switches are connected. The masking is performed as follows, and the switch positions (1 0 1) are stored in memory INBUF.

	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
Input Reading in Accumulator:	X	X	X	X	X	1	0	1
Masking Byte (07 _H):	0	0	0	0	0	1	1	1
Result in Accumulator:	0	0	0	0	0	1	0	1
Flag Status:	S = 0, Z = 0, CY = 0							

Example 9.2

A microcomputer with two input ports and one output port is designed to monitor various processes (conveyor belts) on the floor of a manufacturing plant (Figure 9.3). The input port F1_H with seven switches is located at the north end of the floor, and the input port F2_H, also with seven switches, is located at the south end of the floor. The port F1_H is used to start and stop the conveyor belts in a normal situation, and if necessary, a belt can be stopped or prevented from starting by sending logic 1 through the corresponding switch at port F2_H. (The data line D₇ of the output port is connected to an emergency signal and is not a part of this example.)

Write instructions to

1. Turn on and off the seven conveyor belts according to ON/OFF positions of switches S₆-S₀ at port F1_H.
2. Stop the conveyor belt if the corresponding switch is on at port F2_H.
3. Monitor the switches continuously.

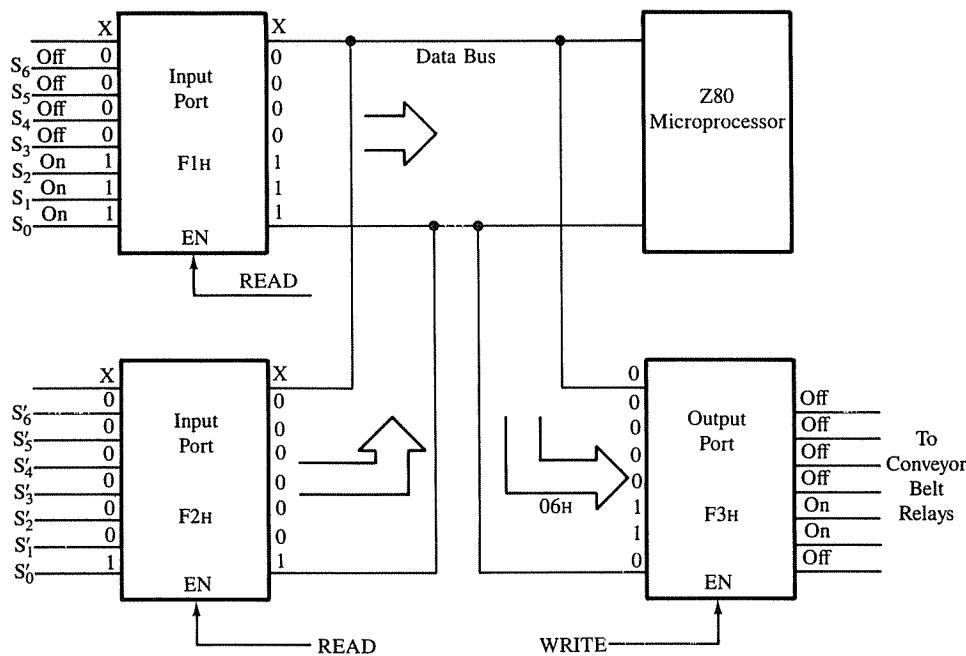


FIGURE 9.3
Microprocessor-Controlled Conveyor Belts

Solution

START: IN A, (F1H)	;Read switches at F1H	X 0 0 0 0 1 1 1 (A)
AND A, 7FH	;Mask bit D ₇	AND 0 1 1 1 1 1 1 1
LD B, A	;Save reading from F1H	0 0 0 0 0 1 1 1 (A) → (B)
IN A, (F2H)	;Read switches at F2H	X 0 0 0 0 0 0 1 (A)
AND A, 7FH	;Mask bit D ₇	AND 0 1 1 1 1 1 1 1
XOR B	;Check switches that are on at F1 and F2	0 0 0 0 0 0 0 1 (A)
OUT (F3H), A	;Turn on/off ap- propriate con- veyor belts	XOR 0 0 0 0 0 1 1 1 (B)
		0 0 0 0 0 1 1 0 (A)

JP START ;Go back and
read switches
again

The three switches S_0 , S_1 , and S_2 at port $F1_H$ are turned on as shown in Figure 9.3. Initially, these switch positions are read, bit D_7 is masked, and the reading is saved in register B. In port $F2_H$, the switch S_0' is on, which means somebody from the south end of the floor wants to stop the belt connected to line D_0 . This reading, after bit D_7 is masked again, is Exclusive-ORed with the reading from the port $F1_H$. Because the switch S_0' is on, the output at port $F3_H$ is 0 0 0 0 0 1 1 0, which turns off the first conveyor belt.

9.12 Compare Instructions

The Compare instructions test a byte for less than, equal to, or greater than the contents of the accumulator, and the comparison is indicated by the flags without affecting the operands. The instructions can test the contents of a register, memory, or 8-bit data against the contents of the accumulator. The instructions are as follows:

Opcode	Operand	Bytes	Description
CP	r	1	Compare the contents of a register with the accumulator
CP	8-bit	2	Compare 8-bit data with the accumulator
CP	(HL)	1}	Compare the contents of memory with the accumulator
CP	(IX + d)	3}	

General Description These instructions compare the operand (data byte, register contents, or memory contents) with the contents of the accumulator by subtracting the operand from the accumulator. However, no contents are modified; the comparison is indicated by setting the flags.

1. If $(A) < \text{operand}$, the Carry flag is set and the Zero flag is reset.
2. If $(A) = \text{operand}$, the Zero flag is set and the Carry flag is reset.
3. If $(A) > \text{operand}$, the Carry and Zero flags are reset.
4. Other flags are also affected according to the result of the subtraction.
5. When the operand is memory, the address is specified by the contents of HL register or index registers with an offset byte.

Example 9.3

Write instructions to compare the byte in memory location 2050_H with 80_H . If the byte is equal to 80_H , jump to location CHECK, and if it is higher than 80_H , jump to OVRLD to indicate the circuit overload.

Solution

```
LD  HL, 2050H ;Set up HL as memory pointer
LD  A, 80H    ;Load comparison byte
CP  (HL)    ;Compare memory byte with 80H
JR  Z, CHECK ;If memory byte = 80H, begin CHECK procedures
JR  C, OVRLD ;Indicate overload
```

A Relative Jump instruction can be used with a label; the assembler will automatically calculate the offset value. However, in manual assembly, the magnitude of the offset must be calculated by the user.

ROTATE (SHIFT) OPERATIONS AND BIT MANIPULATION

9.2

The rotate instructions shift each bit either to the right or to the left. These instructions are used primarily for mathematical operations and serial I/O, where one bit is transmitted over a single line. The Z80 has a set of rotate instructions that can rotate bits not only in the accumulator but in any register and memory location.

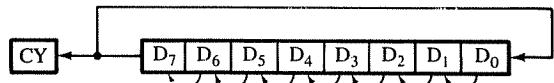
Another group of instructions that makes the Z80 one of the most attractive microprocessors in control applications is bit manipulation. The Z80 can test, set, or reset any bit in an 8-bit register or memory. In other microprocessors, the user must write a set of instructions to test a bit in a register or memory.

In this section, we first introduce the Rotate instructions dealing with the accumulator bits and then discuss the Rotate instructions dealing with registers and memory. Finally, we examine the bit manipulation instructions.

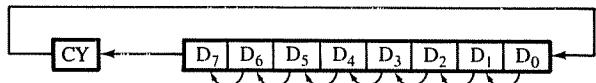
9.21 Rotate Instructions (Accumulator)

The following Rotate instructions deal with the bits in the accumulator. The rotate operations can be classified into two groups: Rotate Left and Rotate Right. Each group can be further classified into (1) 8-bit rotation and (2) 9-bit rotation through Carry. In 8-bit rotation, each bit of the accumulator is shifted to the adjacent position. In this operation, the Carry flag is affected by the rotation, but it is not a part of the rotation. On the other hand, in 9-bit rotation, the C flag is one of the bits in the rotation. These instructions are shown in Figure 9.4.

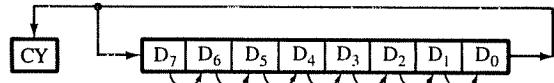
RLCA :Rotate accumulator left.
Carry is affected by D₇.



RLA :Rotate accumulator left through Carry.



RRCA :Rotate accumulator right.
Carry is determined by D₀.



RRA :Rotate accumulator right through Carry.

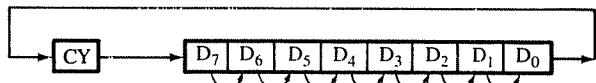


FIGURE 9.4
Rotate Instructions

Example 9.4

The accumulator contains 81_H with the Carry flag reset. Illustrate the contents of the accumulator and the status of the Carry flag after the execution of each rotate instruction.

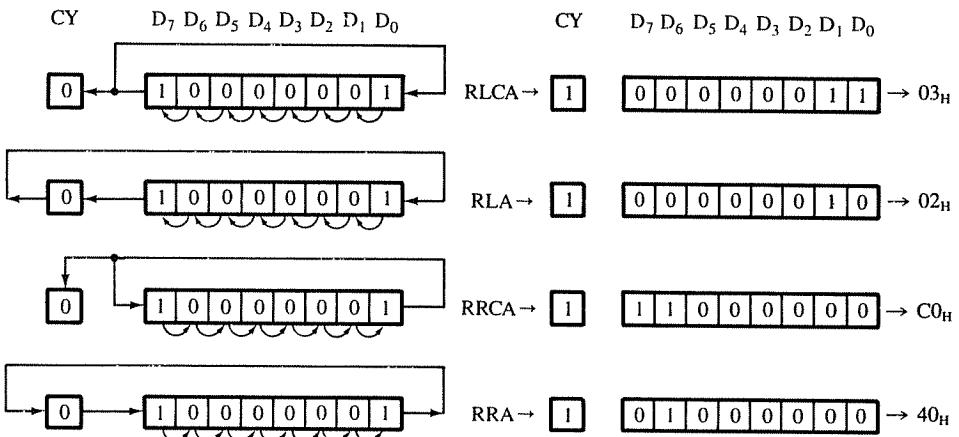


FIGURE 9.5
Rotate Instructions and Accumulator Contents

Solution

Figure 9.5 shows how the byte 81_H in the accumulator is changed after various rotate instructions. The first two instructions rotate bits to the left; however, RLCA is an 8-bit rotation and RLA is a 9-bit rotation. In instruction RLCA, bit D₇ is rotated into bit D₀; yet, in instruction RLA, CY is rotated into bit D₀. Both instructions modify the CY flag according to bit D₇. Similarly, RRCA is an 8-bit and RRA a 9-bit rotate right instruction.

The memory location 2050_H contains a 4-bit number. Write instructions to multiply the number by ten and store the result in the same memory location.

Example 9.5

Solution

LD HL, 2050H	;Set up memory pointer
LD A, (HL)	;Get the number
RLCA	;Multiply by 2
LD B, A	;Save result to use it later
RLCA	;Multiply by 4
RLCA	;Multiply by 8
ADD A, B	;To multiply by 10, add multiply-by-2
LD (HL), A	;Save result in memory
HALT	

Rotating bits to the left by one position is equivalent to multiplying the number by two, and rotating to the right is equivalent to dividing by two. For example, if the number is 01_H , the instruction Rotate Left makes it 02_H . This technique is valid until bit D_7 does not rotate 1 into bit D_0 . For example, rotating the number 80_H left makes it 01_H . However, this number can be divided by two by rotating it right.

In the above instructions, the number is multiplied by eight by rotating the accumulator contents three times. Adding the result of multiply-by-two to the result of multiply-by-eight is equivalent to multiply-by-ten.

9.22 Rotate and Shift Instructions (Registers and Memory)

The Z80 instruction set has several rotate instructions; these instructions can rotate bits in a register or memory. This is unlike the 8080 set, which restricts rotation to the accumulator. In addition, the Z80 has shift instructions which shift bits in a given direction. In the following instructions, r represents any register (A, B, C, D, E, H, or L), and m stands for a memory location in R/W memory. In these instructions, a memory location can be specified by HL or an index register. The instructions are as follows:

Opcode	Operand	Description
RLC	r or m	Rotate bits left in a register or memory.
RL	r or m	Rotate bits left through Carry in a register or memory.
RRC	r or m	Rotate bits right in a register or memory.
RR	r or m	Rotate bits right through Carry in a register or memory.
SLA	r or m	Shift bits left through CY in a register or memory and insert 0 in bit position D_0 .
SRL	r or m	Shift bits right through CY in a register or memory and insert 0 in bit position D_7 .

The set also includes the instructions SRA, RLD, and RRD. (See Appendix A for their complete description.)

General Characteristics

1. In these instructions, the memory address can be specified either by using the HL register or an index register with an offset.
2. Flags S, Z, and P/V are modified according to data conditions. The CY flag is determined by D₇ in left rotation (or shift) and by D₀ in right rotation (or shift).
3. The Shift instructions (SLA and SRA) differ from the rotate instructions in their operations. In Shift instructions, bits are shifted into the next position and 0s are inserted from the other direction. (See Appendix A for complete description of these instructions.)

Example 9.6

In a Key Monitor program, register B is used to store binary codes of data keys of the Hex keyboard. When a key is pressed, the accumulator receives the 4-bit binary code, and it is stored as the low-order four bits in register B. When a new key is pressed, the previous 4-bit code in register B is shifted to the left and the new key code is stored as the low-order four bits. Write instructions to store a new key code in register B.

Solution

```

SLA B      ;Shift low-order key code to left
SLA B      ;and clear bit positions D3–D0 in B
SLA B
SLA B
OR  B      ;Store new key code as low-order bits
LD B, A

```

9.23 Bit Manipulation

The bit manipulation group has three types of instructions: Bit Test, Bit Set, and Bit Reset. These instructions can test, set, or reset a bit in a register or memory. The instructions are as follows:

Opcode	Operand	Description
BIT	b, r or m	Test bit b in register or memory. If bit is 0, set Z flag, and if it is 1, reset Z flag.
SET	b, r or m	Set bit b in register or memory.
RES	b, r or m	Reset bit b in register or memory.

General Characteristics

1. The operand “b” represents any bit from D₇ to D₀; it is specified as a number between 0 and 7.
2. The memory address can be specified by using either the HL register or an index register.
3. The Set/Reset instructions do not affect any flags.

Write instructions to read a byte from PORT1, reset bit D₇ and store the reading in memory INBUF.

Example
9.7

Solution

```
IN A, (PORT1)      ;Read PORT1
RES 7, A          ;Eliminate the parity bit
LD (INBUF), A      ;Store reading in memory
```

To cite a practical use for these instructions, bit D₇ is used to indicate the parity in ASCII characters; therefore, to process these characters, bit D₇ must be eliminated (this will be discussed in Chapter 15).

ILLUSTRATIVE PROGRAM 1: SEARCHING FOR A MAXIMUM NUMBER

9.3

This program searches for a maximum number in a given set of data bytes stored in memory. It compares two numbers at a time, saves the higher number, and continues the process until the end of the data set.

9.31 Problem Statement

A set of ten readings is stored in memory locations starting from INBUF. Write a program to find the highest reading in the set, and store that reading in memory OUTBUF.

9.32 Problem Analysis

1. Initialization: In this problem, we need one counter to count ten readings and a memory pointer for the INBUF memory. In addition, we need one register and the accumulator for comparison.
2. Data Processing: This block involves comparing two numbers and saving the larger one for the next comparison. This process is continued until the counter is zero.

9.33 Program

```
START: XOR A          ;Begin with minimum reading (00)
       LD B, 0AH        ;Set up register B as a counter
       LD HL, INBUF      ;Set up HL as memory pointer for INBUF
NEXT:  CP (HL)         ;Compare memory reading with accumulator
       JP NC, SKIP       ;If reading is lower, do not save
       LD A, (HL)         ;Save reading
SKIP:  INC HL          ;Point to next memory location
       DEC B             ;One comparison complete, decrement count
       JP NZ, NEXT        ;Get next reading if counter ≠ 0
```

LD HL, OUTBUF	;Set up HL as memory pointer for OUTBUF
LD (HL), A	;Save the highest reading
HALT	;End of program

9.34 Program Description

In this program, the new concept is a comparison of two numbers; otherwise, the remaining program is similar to the programs in the previous chapter.

This program begins by clearing the accumulator and then compares the reading in memory INBUF with the accumulator. If the data byte in the accumulator (A) is larger than the data byte in memory (HL), the CY flag is reset, and the program does not save the byte. If the data byte in memory is larger than (A), the CY flag is set, and the byte is saved for the next comparison. This process is continued until all the readings are compared.

9.4

ILLUSTRATIVE PROGRAM 2: GENERATING SQUARE WAVE PULSES

The microprocessor can be used as a **function generator** to produce various types of waveforms using time delays and appropriate hardware. This program generates a square wave for a given frequency by turning a bit of an output port on or off at the specified time interval.

9.41 Problem Statement

Write a program to generate a square wave with period of 500 μ s if the system frequency is 2 MHz. Use bit D₀ of the output port PORT1 to display the waveform.

9.42 Problem Analysis

This problem is somewhat different from the previous data transfer or arithmetic programs. It involves turning bit D₀ on or off every 250 μ s; Figure 9.6 shows the flowchart.

The initialization block is simple; it includes the loading of bit pattern into the accumulator, but does not even require a counter or a memory pointer. The bit manipulation block is similar to the data processing block; it gets ready appropriate bits for an output, and the output block turns bit D₀ on or off. The time delay block provides appropriate delay for the output pulse.

TIME DELAYS

A **time delay** is generated by loading a general-purpose register with an appropriate count and setting up a loop to decrement the count until it reaches zero. The delay is determined by the clock period of the system, the number of instructions in the loop, and the number of times the loop is repeated. A typical set of instructions representing the time delay is shown here, and Figure 9.7 shows the flowchart for these instructions.

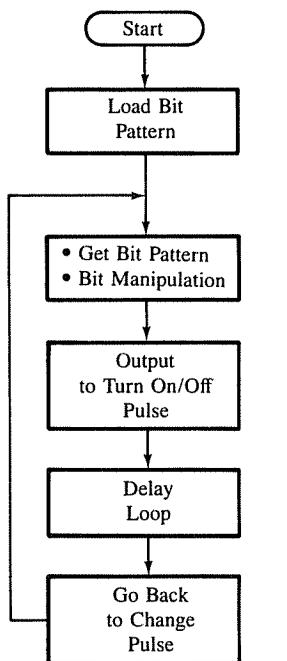


FIGURE 9.6
Flowchart: Square Wave Generation

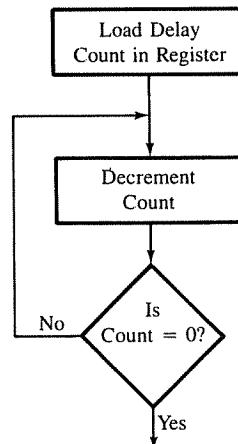


FIGURE 9.7
Flowchart: Time Delay

Mnemonics	T-states	Comments
LD B, 64H	7	;Delay Count
LOOP:DEC B	4	;Delay Loop
JP NZ, LOOP	10	

To calculate the time delay in this loop, we need to examine the T-states in the loop; one T-state is equivalent to one clock period of the system. For example, the instruction DEC B has four T-states, so the Z80 executes the instruction in four clock periods. The loop includes two instructions with 14 T-states, and the loop is repeated 100 ($64_{16} = 100$) times; the first instruction LD B, 64H is not part of the loop. Therefore, the loop delay, T_L , is calculated by the formula $T_L = (T_C \times L_T \times N_{10})$. In our example,

$$\begin{aligned}
 T_C &= \text{System clock period } (f = 2 \text{ MHz}; T = 1/f = 0.5 \text{ } \mu\text{s}) \\
 L_T &= \text{Loop T-states (14)} \\
 N_{10} &= \text{Count in decimal } (64_{16} = 100) \\
 T_L &= (0.5 \times 10^{-6} \times 14 \times 100) \\
 &= 700 \text{ } \mu\text{s}.
 \end{aligned}$$

To calculate the total delay, we need to include the execution time outside the loop. In this example, one instruction (LD B, 64H) is outside the loop. It has seven T-states and will require 3.5 μ s.

$$\begin{aligned} \text{Total Delay } T_D &= T_o + T_L \\ \text{where } T_o &= \text{Delay outside the loop} \\ T_L &= \text{Loop delay} \\ T_D &= 700 \mu\text{s} + 3.5 \mu\text{s} \\ &= 703.5 \mu\text{s} \end{aligned}$$

In most applications, the delay outside the loop is insignificant and can be ignored. However, in our illustrative program, the delay outside the loop is quite significant, as discussed in Section 9.44. We can now write the program for generating square wave.

9.43 Program

	Mnemonics	T-states	Comments
1.	START: LD C, 01010101B		;Load bit pattern
2.	ROTATE: LD A, C	(4)	;Place bit pattern in A
3.	RLCA	(4)	;Change bit pattern for next output
4.	LD C, A	(4)	;Save bit pattern
5.	AND 01H	(7)	;Mask bits D ₇ –D ₁
6.	OUT (PORT1),A	(11)	;Change pulse voltage
7.	LD B, COUNT	(7)	;Load register B with a delay count
8.	LOOP: DEC B	(4)	;Set up delay loop
9.	JP NZ, LOOP	(10)	
10.	JP ROTATE	(10)	;Go back to change pulse voltage

9.44 Program Description

The bit pattern is selected for this program in such way that when it is rotated, it provides logic 0 and 1 alternately at bit D₀. The bit pattern is masked by ANDing with the byte 01 to eliminate bits D₇–D₁. Bit D₀ is turned on and off at the interval of 250 μ s to generate a square wave with the period of 500 μ s. In this program, we are concerned with the delay between two consecutive outputs and not just the delay in the loop. The loop count for the total delay of 250 μ s is calculated as follows:

1. The delay loop consists of two instructions (8 and 9) with a total of 14 T-states; however, the number of times the loop is repeated (COUNT) needs to be calculated.

$$\begin{aligned} \text{Delay in the loop } T_L &= (0.5 \times 10^{-6} \times 14 \times \text{Count}) \\ &= 7 \times 10^{-6} \times \text{Count} \end{aligned}$$

2. In this problem, we are concerned with not just the delay in the loop, but how long the pulse stays on. After the execution of the OUT instruction (6), all the delay caused by the instructions in addition to the delay loop should be accounted for until the OUT instruction is executed again. The instructions that are executed once are 2 through 7 and 10. Therefore,

T-states outside the loop = 47.

$$\text{Delay outside the loop } T_O = (0.5 \times 10^{-6} \times 47) = 23.5 \mu\text{s}$$

3. Total delay $T_D = T_o + T_L$

$$250 \mu\text{s} = 23.5 + 7 \times 10^{-6} \times \text{Count}$$

$$\text{Count} = (250 - 23.5)/7 \approx 32$$

In this program, the delay outside the loop is quite significant, ten percent of the total delay. The loop will be repeated 32 (20_H) times.

DEBUGGING PROGRAMS

9.5

The debugging techniques discussed in the previous chapter can be used to check errors in programs similar to those discussed in this chapter. Common sources of errors in these types of programs are as follows:

1. Failure to update a memory pointer or a counter.
2. Failure to set a flag before using a conditional Jump instruction. This is especially true with 16-bit increment/decrement instructions.
3. Failure to save partial results.
4. Specifying Jump instruction on a wrong flag. This error occurs frequently with the Compare instructions.
5. Use of wrong Rotate instruction or improper combination of Rotate instructions.
6. Errors in counting T-states in a delay loop. Typically, the first instruction—to load a delay register—is mistakenly included in the loop.
7. Errors in recognizing how many times a loop is repeated.
8. Failure to convert a delay count from a decimal number into its hexadecimal equivalent or vice versa.
9. Conversion error from decimal to hexadecimal number or vice versa.
10. Specifying wrong jump location, thus possibly setting up an infinite loop.

9.51 Illustrative Program for Debugging

The following program is designed as a Hex counter to

1. Count from 00 to 20_H continuously.
2. Provide 450 ms delay between two consecutive counts.
3. Illustrate the loop within the loop technique. The inner loop LOOP1 provides 2.25 ms of delay.

The program includes several errors. Recognize the errors by answering the questions below. After correcting those errors, execute the program. If the program still does not give you the expected output, debug the program using the single-step and the break point techniques. For delay calculations, the system's clock is assumed to be 2 MHz.

Mnemonics	T-states	Comments
1. START: LD A, 00H		;Load initial count
2. DSPLAY: OUT (PORT1), A	(11)	;Display count
3. LOOP2: LD B, C7H	(7)	;Load count for outer loop
4. LD C, COUNT1	(7)	;Load count for inner loop
5. LOOP1: DEC C	(4)	;Decrement COUNT1
6. NOP	(4)	;Add T-states for delay
7. JP NZ, LOOP1	(10)	;Repeat LOOP1 until C = 0
8. DEC B	(4)	;Decrement count in outer loop
9. JP Z, LOOP2	(10)	;Repeat LOOP2 until B = 0
10. INC A	(4)	;Next Hex count
11. CP 20H	(7)	;Is Hex count = 20H?
12. JP NC, DSPLAY	(10)	;Go back to display count
13. JP DSPLAY		;Start again

DEBUGGING QUESTIONS

1. Is the jump location for instruction 7 appropriate?
2. The COUNT1 for LOOP1 is as follows:

$$\begin{aligned}
 T_{IL} &= T_c \times (\text{T-states}) \times \text{COUNT1} \\
 2.25 \text{ ms} &= 0.5 \text{ } \mu\text{s} \times 25 \times \text{COUNT1} \\
 \text{COUNT1} &= 180
 \end{aligned}$$

Find the error in the calculations, and recalculate COUNT1.

3. The total delay (T) between two consecutive outputs is the sum of the initial delay (T_I), the outer loop delay (T_{OL}), and the inner loop delay (T_{LI}). The inner loop (LOOP1) is repeated 199 ($C7H$) times because of the outer loop. Thus:

$$T_{LI} = 199 \times 2.25 \text{ ms} = 447.75 \text{ ms}$$

The instructions in the outer loop are 3, 4, 8, and 9, which require a total of 28 T-states. Since these instructions are repeated 199 times:

$$T_{OL} = (0.5 \text{ } \mu\text{s}) \times 28 \times 199 = 2.78 \text{ ms}$$

The instructions executed once before the next display are 2, 3, 10, 11, and 12—a total of 39 T-states.

$$T_I = 0.5 \text{ } \mu\text{s} \times 39 = 19 \text{ } \mu\text{s}$$

Is there any error in these calculations, assuming the appropriate COUNT1?

4. Is the jump location for instruction 9 appropriate? What is the effect of the present location on LOOP2?
5. Execute the program, and observe the output. Write your educated guesses about the output.
6. Set up a breakpoint after instruction 8 and execute the program. Examine the registers.
7. Single-step the remaining instructions and examine the contents of registers if necessary. Explain your observations.
8. Correct the errors and run the program again.

Z80 SPECIAL INSTRUCTIONS

9.6

In this section, we introduce additional Z80 special instructions. The Z80 has Compare instructions that are capable of searching for a given byte in memory; some instructions can search through 64K bytes of memory.

Instructions

CPI Compare and Increment

This instruction compares the contents of the memory location specified by register HL with the contents of the accumulator. Register HL is incremented and register BC is decremented.

CPIR Compare, Increment, and Repeat

This instruction is similar to the previous instruction CPI, except that the instruction is repeated until BC = 0 or the contents of memory are equal to the contents of the accumulator.

CPD Compare and Decrement

This instruction compares the contents of the memory location specified by register HL with the contents of the accumulator. Registers HL and BC are decremented.

CPDR Compare, Decrement, and Repeat

This instruction is similar to the previous instruction CPD, except that the instruction is repeated until BC = 0 or the contents of memory are equal to the contents of the accumulator.

General Characteristics

1. These are 2-byte instructions.
2. The Zero (Z) flag is set when a match is found, meaning the memory byte is the same as the accumulator byte.
3. The Sign (S) flag is set if the memory byte is larger than the accumulator byte.
4. The Parity/Overflow (P/V) flag is reset when BC = 0.
5. The Carry flag is not affected.

**Example
9.8**

The input buffer memory (INBUF) contains 256 bytes of data. Search for the byte (character) 24H in the input buffer. If it is found, jump to location START; otherwise, jump to location ERROR.

Solution

LD HL, INBUF	;Set up HL as memory pointer
LD BC, 0100H	;Set up BC as a counter
LD A, 24H	;Load the byte to be searched
CPIR	;Search for 24H in the input buffer
JR Z, START	;Character found, start the process
JR ERROR	;Display error message

The instruction CPIR will be repeated until it finds the character 24H. When it finds the character, the Z flag is set, the loop is terminated, and the program jumps to location START. If there is no match, the instruction is repeated until BC = 0, and the P/V flag is reset. This flag can be used for decision making if necessary. (The P/V flag is not used in this example.)

9.7

ILLUSTRATIVE PROGRAM 3: SEARCHING FOR A MAXIMUM NUMBER USING THE INSTRUCTION CPI

This program searches for a maximum number in a given set of data bytes stored in memory; this is similar to Illustrative Program 1 in Section 9.3. It compares two numbers at a time using the instruction CPI and saves the higher number, and the process is continued until the end of the data set.

9.71 Problem Statement

A set of ten readings is stored in memory locations starting from INBUF. Write a program to find the highest reading in the set, and store that reading in memory OUTBUF.

9.72 Problem Analysis

1. Initialization: To use the multi-tasking instruction CPI (Compare and Increment), BC should be used as a 16-bit counter even if the number of readings is not higher than 255, and HL should be used as the memory pointer for the INBUF memory.
2. Data Processing: This block involves comparing two numbers and saving the larger one for the next comparison. This process is continued until the counter is zero.

9.73 Program

START: XOR A	;Begin with minimum reading (00)
LD BC, 000AH	;Set up register B as a counter
LD HL, INBUF	;Set up HL as memory pointer for INBUF

NEXT: CPI	;Compare memory reading with accumulator
JP P, SKIP	;If new reading is lower, do not save
LD A, (HL)	;Save this reading for next comparison
SKIP: JP PE, NEXT	;Get next reading if counter $\neq 0$
LD HL, OUTBUF	;Set up HL as memory pointer for OUTBUF
LD (HL), A	;Save the highest reading
HALT	;End of program

9.74 Program Description

This program is similar to the program given in Section 9.3, except that it uses the instruction CPI, which eliminates the need to update the memory pointer and the counter. The other significant differences are in flags. The S flag is used to jump to location SKIP (JP P, SKIP) instead of the Carry flag (JP NC), and the P/V flag is used to go to location NEXT instead of the Z flag. These changes are necessary because the CPI instruction does not affect the Carry flag, and it sets the Z flag when a match is found in comparing two bytes.

To compare two bytes, the instruction CPI subtracts the contents of the memory location pointed to by the HL register from the accumulator. If the number in the accumulator is smaller than the number in the memory location, it sets the S (Sign) flag to indicate the negative result; otherwise, it resets the S flag to indicate the positive result. The contents of the accumulator or memory are not affected in this comparison. The jump instructions associated with the Sign flags are JP M (Jump On Minus, S = 1) and JP P (Jump On Positive, S = 0). In our illustration, if the number in the accumulator is larger than the number in the memory location, S = 0, and the instruction JP P, SKIP does not save the new byte from memory.

SUMMARY

In this chapter, instructions related to logic (AND, OR, XOR) operations, compare operations, bit rotation, and bit manipulation were introduced. The chapter is concluded with the illustrations of Z80 special instructions related to compare operations. General characteristics of these instructions are as follow:

1. Logic operations can be performed with the contents of the accumulator and the contents of a register, memory, or 8-bit data. The AND, OR, and XOR instructions reset the CY flag and modify other flags according to the result of an operation.
2. A byte can be compared with the contents of the accumulator; the byte can be direct 8-bit data or from a register or memory. The Compare instructions perform the comparison by subtracting the byte from the accumulator, and the comparison is indicated by setting appropriate flags without affecting the contents. When (A) < the byte, the CY flag is set; when (A) = the byte, the Z flag is set,

and when $(A) >$ the byte, the CY and Z flags are reset. All other flags are affected according to the result of the subtraction.

3. The rotate instructions can rotate bits in the accumulator, register, or memory either left or right by one position. Bit rotations can be performed either for eight bits or for nine bits including the CY flag. In either rotation, the status of the CY flag is determined by D_7 in the left rotation and by D_0 in the right rotation.
4. The shift instructions can shift each bit in the accumulator, register, or memory either left or right by one position. When bits are shifted to the left, bit D_7 is placed into the CY flag and 0 is inserted into bit D_0 . When bits are shifted to the right, bit D_0 is placed into the CY flag and 0 is inserted into bit D_7 .
5. Bit manipulation instructions can test, set, or reset any bit in a register or memory.

Two applications programs—searching for a maximum number in a data set and generating a square wave—were illustrated. The square wave program also illustrated how to design time delays. Errors that commonly occur in writing these programs were listed, and debugging techniques were suggested in the context of a counter program.

Finally, the Z80 special instructions related to block compare operations were illustrated. These Compare instructions can be used for searching a byte in memory.

ASSIGNMENTS

Section 9.1

1. Write instructions to load 80_{16} and $7F_{16}$ into registers B and C respectively. Logically AND the bytes and save the answer in memory INBUF. Specify the status of the S, Z, and CY flags after ANDing the bytes.
2. If the bytes in 1 are ORed instead of ANDed, specify the contents of the accumulator and the flag statuses (S, Z, CY).
3. Specify the contents of the accumulator and the flag statuses (S, Z, CY) after executing the instruction XOR A.
4. Write instructions to read PORT1. If the reading is 00_{16} , set the Z flag and jump back to read the port again. What is the reason to set the Z flag when the input reading is 00_{16} ? Specify a 1-byte logic instruction that can set the Z flag without affecting the input reading.
5. In 4, is AND A an appropriate instruction to set the Z flag? Explain your answer.
6. Write instructions to read the input port (INPUT) and mask bit D_7 .
7. Load bit pattern 97_{16} into register D and mask high-order bits D_7 – D_4 .
8. Write instructions to load BYTE1 and BYTE2 into registers D and E respectively. Check bit D_0 in both bytes, and if either one is at logic 1, turn on the indicator connected to bit D_0 at the output port OUT1.

9. Eight lights are connected to output port OUT1. These can be turned on from the corresponding switches from either of the input ports INPUT1 or INPUT2. Write instructions to read INPUT1 and INPUT2. If all the switches are off in both ports, continue to read the input ports. When a switch (or switches) is on in either port, turn on the corresponding light(s) at OUT1.
10. Write instructions to load 37_H into the accumulator and $6F_H$ into register B. Compare the two bytes and specify the statuses of the S, Z, and CY flags.
11. When BYTE2 in register B is compared with BYTE1 in the accumulator, the CY flag is reset. Explain the significance of the CY flag status.
12. The following instructions read the switches S_7-S_0 from port INPUT1 and $S_7'-S_0'$ from port INPUT2. When a switch is ON, it provides logic 1 to the corresponding data line (for example, S_7 to D_7). The readings are processed and used for decision making. Read the instructions, and answer the questions following.

```
START: LD HL, 2065H
        LD (HL), 80H
READ:  IN A, (INPUT1)
        LD B, A
        IN A, (INPUT2)
        AND B
        JP Z, READ
        LD B, A
        AND 80H
        CP (HL)
        JP Z, URGENT
        OUT (OUT1), A
        Continue---
```

- a. What is the output at OUT1 when switches S_0 , S_1 , S_3' , and S_7' are turned ON? Explain your answer.
- b. Does the program jump to location URGENT when switches S_7 , S_7' , and S_0 are ON, or does it go back to READ?
- c. Specify the output if switches S_7 , S_6 , S_5 , S_1 , S_0 from INPUT1 and S_5' , S_4' , S_1' , and S_0' from INPUT2 are ON.

Section 9.2

13. The accumulator contains the byte 77_H . What is the byte in the accumulator and the CY flag status after the execution of the instruction RRCA?
14. The accumulator contains the data byte $C1_H$, and the CY flag is 0. Specify the contents of the accumulator and the status of the CY flag if the instruction RLCA is executed twice.
15. In 14, specify the contents of the accumulator and the status of the CY flag if the instruction RLA is used instead of the instruction RLCA.
16. What is in the accumulator and the CY flag after the execution of the following instructions?

```

LD A, F3H
OR A
RLA
RRCA

```

17. Register B holds the byte $3F_H$, representing the values of two Hex keys, 3 and F. The accumulator holds 02_H , representing a new key. Specify the contents of register B after the execution of the following instructions and explain the function of the instruction OR A.

LD L, A	;Save new key in register L
LD C, 04H	;Set up C as a counter
LD A, B	;Get previous two keys
SHIFT: OR A	;The next four instructions shift low-order
RLA	;four bits of register B (now in A) into high-
DEC C	;order positions D_7-D_4 and clear D_3-D_0
JP NZ, SHIFT	
OR L	;Place bits of new key into D_3-D_0
LD B, A	;Save key bits in B

18. Write instructions using the masking technique and four RLCA instructions to perform the same shift function as in 17.
19. Write instructions to shift high-order bits D_7-D_4 of the byte in the accumulator into low-order position D_3-D_0 , and multiply the bits by eight.
20. In 19, mask the low-order bits D_3-D_0 and shift the remaining bits to the right by one position. Is the result the same as in 19?
21. Mask the high-order bits D_7-D_4 of the accumulator and add the remaining bits D_3-D_0 four times using the instruction ADD A, A. Explain the result.
22. Can you achieve the same result as in 21 by using the shift instruction?
23. Write instructions to reset the bit D_7 in the accumulator and check whether the number is odd. If it is odd, jump to the REJECT routine; otherwise continue.
24. Write instructions to check bits D_7 and D_0 in the accumulator, and if both bits are high, jump to the URGNCY routine.

Section 9.3

25. Rewrite Illustrative Program 1 (Section 9.3) to find the minimum number in a given data set.
Data (H) 32, F8, 6A, 47, 1F, AF, 97, 20, 2F, C2
26. A set of ten readings is stored in memory DATA. Write a program to check whether the byte 30_H exists in the set. If it does, stop checking, and display its memory location; otherwise output FFH.
Data (H) 48, 8F, C7, 68, 9F, 9C, 30, 33, B8, D9
27. A set of ten readings, representing the power consumption in watts of each

house in the area, is stored in memory INBUF. The limit on consumption per house is set at 200_{10} watts. Check each reading and count all the readings that exceed the limit and display the number.

Data (H) A9, B3, 98, C8, C7, F5, C8, 89, D2, E7

28. A set of eight current readings is stored in memory INBUF. The readings are expected to be positive ($<128_{10}$). Write a program to check each reading, reject the negative readings, and add the positive readings. Display the answer at the output port or store it in the output buffer memory OUTBUF.

Data (H) 74, 6F, A1, 7F, 76, 87, 5B, 8C

29. In 28, modify the program to add the positive readings until the sum exceeds FFH. If the addition generates a carry, stop the addition and display 01H at the output port; otherwise, display the sum.

Data (H) 27, A1, 2A, 1F, 38, 81, 19, 9A

Data (H) 87, 22, 5F, 3A, 47, 52, 35, 81

30. A data string is stored in memory INBUF, and the end of data string is indicated by the data byte 00H. Copy the data string into new memory OUTBUF.

Data (H) 67, 89, 7F, F5, C8, 9A, 4B, 00, F8, F8

Section 9.4

31. Calculate the period of the square wave in Illustrative Program 2 if the COUNT in the delay loop is changed to 44H.
32. Write a program to generate a square wave with the period of 750 μ s. Use bit D7 to output the square wave.
33. Write a program to generate a rectangular wave with a 300 μ s on-period and 500 μ s off-period.
34. In the following instructions, calculate the delay in LOOP2, LOOP1 (exclusive of LOOP2), and the total delay if the system clock is 4 MHz.

Instructions	T-states
START: LD B, 64H	7
LOOP1: LD C, FAH	7
LOOP2: NOP	4
NOP	4
DEC C	4
JP NZ, LOOP2	10
DEC B	4
JP NZ, LOOP1	10

35. The following instructions use two memory locations MEM1 and MEM1 + 1 as counters to set up delay loops. Calculate the delay in LOOP1, LOOP2 (exclusive of LOOP1), and the total delay (clock period = 0.5 μ s).

Instructions	T-states	Comments
START: LD HL, MEM1		;Set up HL as memory pointer
LD (HL), 32H		;Load MEM1 with count for LOOP1
LOOP1: INC HL	6	;Point to MEM1 + 1
LD (HL), F8H	10	;Load count for LOOP2
LOOP2: DEC (HL)	11	;Begin LOOP2
JP NZ, LOOP2	10	;Go back if MEM1 + 1 ≠ 0
DEC HL	6	;Point to MEM1
JP NZ, LOOP1	10	;Go back if MEM1 ≠ 0

Section 9.5

36. The following program checks eight numbers stored in memory INBUF, rejects the negative numbers, and adds the positive numbers. If the sum generates a carry, it displays 01_H for an overload condition; otherwise, it displays the sum. However, it appears that the program works only for certain data sets. Debug the program and execute it for the given three data sets. After debugging the program, when it works for Set 2, make sure that it also works for Set 3.

Set 1 (H) 77, 8F, 68, 32, 47, 92, 89, 6C Expected Output: 01_H
 Set 2 (H) 32, 10, 2A, 8A, A2, B5, 22, 15 Expected Output: A3_H
 Set 3 (H) 87, 2C, 19, 22, CF, F2, 41, D3 Expected Output: A8_H

PROGRAM

```

START: LD HL, INBUF      ;Set up HL as memory pointer
       LD C, 08H        ;Set up register C as a counter
       LD B, 00H        ;Clear B to save partial results
NEXT:  LD A, (HL)        ;Get the byte
       RLA             ;Place D7 in CY
       JP C, REJECT    ;If D7 = 1, reject number
       RRCA            ;If number is positive, restore it
       ADD A, B        ;Add the previous sum
       JP C, OVRLOD    ;If sum > FFH, it is overload
       LD B, A        ;Save the sum
REJECT: INC HL          ;Point to the next number
       DEC C           ;Update counter
       JP NZ, NEXT     ;If all numbers are not checked,
                         ;go back and get the next number
                         ;Display the sum
                         ;End
OVRLOD: LD A, 01H        ;Load overload indicator
        OUT (PORT1), A  ;Display overload signal
        HALT            ;End

```

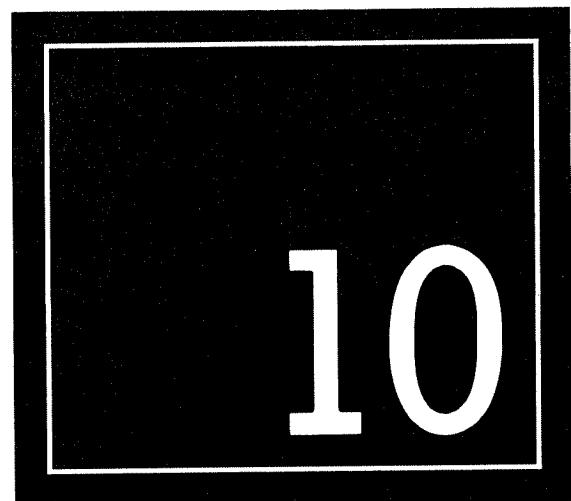
Sections 9.6 and 9.7

37. Write a program to transfer a block of data from memory INBUF to new memory locations OUTBUF. The end of the data string is indicated by 00_H . The suggested Z80 special instructions to be used are LDI (Load and Increment) and JR (Jump Relative).
38. Rewrite Illustrative Program 3 to find the minimum number in a given data set of ten readings.

Stacks and Subroutines

The **stack** is a group of memory locations in a system's R/W memory that is used to store register contents and memory addresses temporarily during the execution of a program. The starting location of the stack is defined by loading a 16-bit address into the stack pointer, and space is reserved, usually at the high end of the memory map. This method of information storage resembles the process of stacking books, one above another, so that information is always retrieved from the top of the stack; hence, the particular group of memory locations is called the stack. In this chapter, the processes of information storage into the stack and retrieval from the stack and associated instructions are introduced. An illustrative program demonstrates how to use these instructions to examine and manipulate the flags.

The latter part of the chapter deals with the subroutine technique. A **subroutine** is a group of instructions that performs a subtask (for example, time delay) that is required repeatedly in a program. The subroutine is written as a separate unit, apart from the main program, and can be called whenever it is necessary. When a main program calls a subroutine, the program execution is transferred to the subroutine, and after the completion of the subroutine, the program execution returns to the main program. The microprocessor uses the stack to store the return address.



The subroutine and the stack offer a great deal of flexibility in writing programs. The subroutine technique eliminates the need to repeat the instructions for a subtask; thus, memory is used efficiently and programs can be written concisely. The use of a stack can provide a practically unlimited number of microprocessor registers. When a subroutine is written, the contents of the registers being used by the calling program can be stored on the stack, and the registers can be reused in the subroutine to perform

the subtask. At the end of the routine, the register contents of the calling program can be retrieved. The illustrative program, Traffic Signal Controller, demonstrates the use of the subroutine technique.

In the industrial environment, a large software project is generally divided into subtasks called modules. These modules can be developed and

tested independently as subroutines by different programmers. This **modular approach to software design** provides flexibility and ease in writing programs. The modular approach is demonstrated by designing a BCD counter and its seven-segment display, and techniques are suggested for debugging modular programs.

OBJECTIVES

- Define the stack and initialize it at a given memory location using the stack pointer (register).
- 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 examined and how a given flag can be set or reset.
- Define the subroutine and explain its uses.
- Explain the sequence of 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 the CALL and RET (Return) instructions are executed.

- Write a subroutine for a given task.
- List and explain conditional Call and Return instructions.
- Explain multiple call, nested, and multiple ending subroutines.
- Explain the modular programming technique and demonstrate the technique by writing a program.

10.1 STACK

The stack is a group of memory locations in R/W memory, defined by loading a memory address into the *stack pointer (register)*.* The stack is used to store binary information temporarily during the execution of a program. Theoretically, the size of the stack is unlimited, restricted only by the available R/W memory in a microcomputer system.

In Z80 systems, the beginning of the stack is defined in the program by using the instruction LD SP, 16-bit (Load Stack Pointer), which loads the 16-bit address into the stack pointer (register). The contents of register pairs (BC, HL, for example, but not just a single register) can be stored in two consecutive stack memory locations by using the instruction PUSH and can be retrieved from the stack into register pairs by using the instruction POP. The microprocessor keeps track of the stack by incrementing or decrementing the address in the stack pointer (register). The address in the stack pointer (register) always points to the top of the stack and indicates that the next memory location (SP - 1) is available to store information.

Once the stack pointer is loaded with a 16-bit address—for example, 2099_H —the storing of information begins at the next location $SP - 1$ (2098_H) in the decreasing order. The contents of a register pair are stored at $SP - 1$ and $SP - 2$ (2098_H and 2097_H), and the

*Initially, we are using the term *stack pointer (register)* to emphasize the difference between the stack as memory and the stack pointer as a 16-bit register.

stack pointer is decremented by two from 2099_H to 2097_H . The storing of information can continue in the reversed numerical order (decreasing memory addresses). Therefore, as a general practice, the stack is initialized at the highest possible memory location to prevent the user program from being destroyed by the stack information. The process of information retrieval from the stack is opposite to the storing process; it begins at the location pointed to by the stack pointer whenever a POP instruction is executed, and the stack pointer is incremented twice. This process will be further clarified in Example 10.1.

The stack is shared by the programmer and the microprocessor to store information. The programmer can store and retrieve the contents of register pairs by using PUSH and POP instructions. Similarly, the microprocessor can automatically store and retrieve the contents of the program counter when a subroutine is called (discussed later in the chapter).

10.11 Stack Instructions

The instructions used to store and retrieve information to and from the stack are listed here.

Opcode	Operand	Description
LD	SP, 16-bit	Load 16-bit address into the stack pointer register. This is a load instruction, similar to other 16-bit load instructions discussed previously.
PUSH	rp	This is a 1- or 2-byte instruction and copies the contents of the specified register pair or index register onto the stack as described below. Instructions for four register pairs and index registers are listed here.
PUSH	rx	
PUSH	AF	The instruction first decrements the stack pointer (register) and copies the high-order byte of the register pair or the index register on the stack location $SP - 1$.
PUSH	BC	
PUSH	DE	
PUSH	HL	Then it again decrements the stack pointer and copies the low-order byte of the register pair or the index register onto the stack location $SP - 2$.
PUSH	IX	
PUSH	IY	
POP	rp	This is a 1- or 2-byte instruction and copies the contents of the top two locations of the stack into the specified register pair.
POP	rx	
POP	AF	First, the instruction copies the contents of the stack pointed to by SP into the low-order register (for example, register C of the BC pair) or as a low-order byte into the index register and then increments the stack pointer to $SP + 1$.
POP	BC	
POP	DE	
POP	HL	
POP	IX	
POP	IY	It copies the contents of the $SP + 1$ location into the high-order register (for example, register B of the BC pair) or as a high-order byte into the index register and increments the stack pointer to $SP + 2$.

The Z80 instruction set includes six PUSH and six POP instructions associated with six register pairs (AF, BC, DE, HL, IX, and IY), and these instructions belong to the data copy group; thus, the contents of the stack (source) are not modified, and no flags are affected, but the stack pointer is adjusted according to the instructions.

Example 10.1

The R/W Memory of the system ranges from 2000_H to $23FF_H$. A program is stored in memory locations from 2000_H to 2055_H , and the stack is initialized at the location 2400_H . Two segments of the program are shown below.

1. Explain why the stack is initialized at 2400_H when, in fact, the R/W memory extends up to $23FF_H$.
2. Explain the data transfer between the registers and the stack when PUSH and POP instructions are executed.
3. Modify the instructions to exchange the contents of BC and HL when the contents are retrieved from the stack.

PROGRAM

```
2000 LD SP, 2400H
2003 LD HL, 22A2H
2006 LD BC, 2110H
2009 LD A, (HL)
200A OR A
```



```
2010 PUSH BC
2011 PUSH HL
2012 PUSH AF
```



```
2035 POP AF
2036 POP HL
2037 POP BC
```



2055

- Because the stack pointer is initialized at 2400_H , the first available stack memory location for storage is $23FF_H$; the location 2400_H will never be used to store information with a PUSH instruction. This is an efficient way of using R/W memory.
- The first three instructions load the contents as shown in Figure 10.1. The instruction at location $2010H$ (PUSH BC) decrements the stack pointer to $SP-1$ and stores the contents of B (21_H) in location $23FFH$. It then decrements the stack pointer to $SP-2$ ($23FE_H$) and stores the contents of C (10_H) in the location $23FE_H$. Figure 10.2 shows

Solution

FIGURE 10.1
Register Contents

A		F
B	21_H	10_H
D		C
H	22_H	E
SP	2400	L

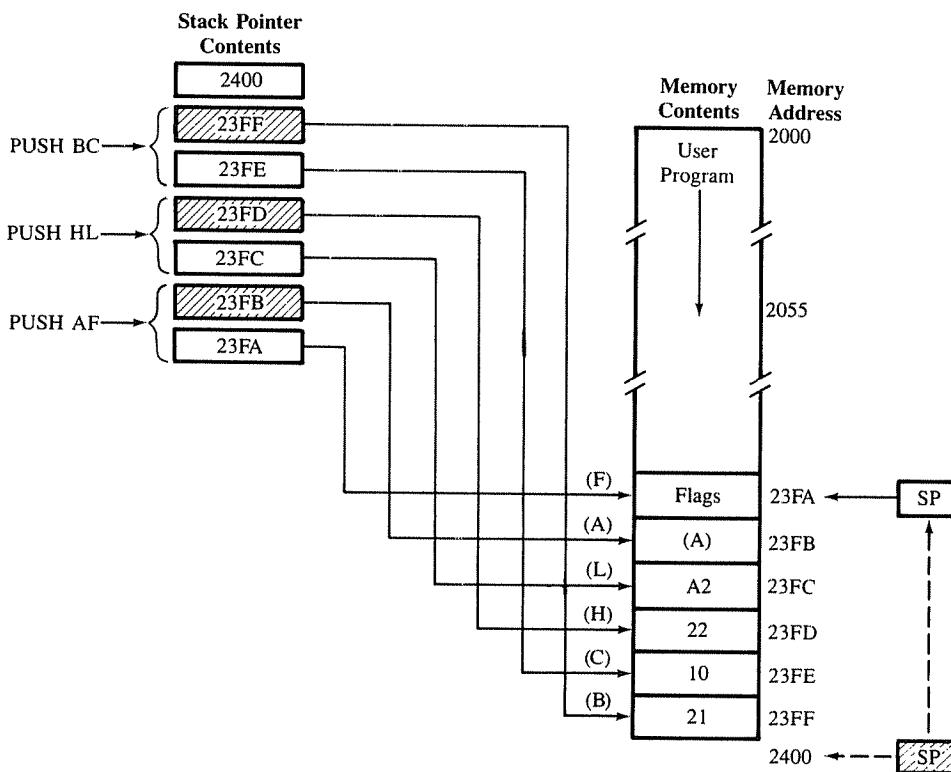


FIGURE 10.2
Contents of Stack Pointer and Stack After Execution of PUSH Instructions

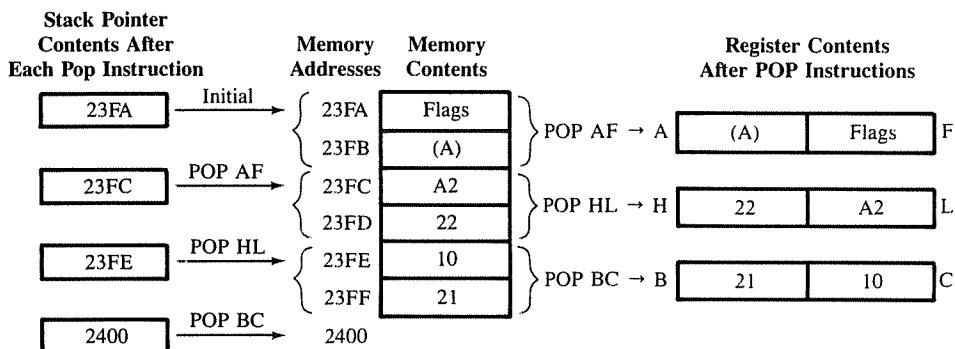


FIGURE 10.3

Register Contents After Execution of POP Instructions

the stack contents after the execution of three PUSH instructions; the stack pointer is at location $23FA_H$.

- Figure 10.3 shows how the contents are retrieved to their respective registers by POP instructions in the Last-In-First-Out (LIFO) sequence. By examining the sequence of PUSH and POP instructions, you may notice that the contents are placed on the stack in BC, HL, and AF sequence and retrieved in reversed order: AF, HL, and BC. This retrieval sequence is necessary to retrieve the original register contents.

The last two bytes placed on the stack by the instruction PUSH AF are on the top of the stack, and the stack pointer points to the top location ($23FA_H$). The instruction POP AF places the contents of the top location into the flag register, increments the stack pointer to $23FB_H$, places the contents of $23FB_H$ into the accumulator, and again increments the stack pointer to $23FC_H$. The process is repeated for the next two POP instructions, and the stack pointer returns to its original address, 2400_H .

- To exchange the contents of BC and HL registers during the retrieval of information from the stack is a simple task that can be accomplished by exchanging the positions of POP BC with the POP HL in the program. The critical aspect to remember about the POP instructions is that they copy the top of the stack into the specified register pair irrespective of how the bytes were stored on the stack.

10.12 Review of Important Concepts

The following points can be summarized from the above example:

- Memory locations in R/W Memory can be employed as temporary storage during the execution of a program by loading a 16-bit address (initializing) into the stack pointer (register). The contents of register pairs can be stored beginning from the next location ($SP - 1$).
- The stack space grows upward in the numerically decreasing order of memory addresses.
- 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.

4. The PUSH instructions are used to store contents of register pairs on the stack, and the POP instructions are used to retrieve the information from the stack. 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, respectively.
5. The storage and retrieval of data bytes on the stack should follow the LIFO (Last-In-First-Out) sequence.
6. Information in the stack locations is not destroyed until new information is stored in those locations.

10.13 Additional Instructions: Exchange

The Z80 includes an alternate set of registers, and these registers can be used to store the contents of general-purpose registers, the accumulator, and the flags. The contents can be saved by using Exchange instructions. The alternate registers serve a function similar to that of the stack. The following list also includes exchange instructions that exchange contents between an index register or the HL register and the stack.

Instruction	Description
EXX	Exchange the contents of general-purpose registers (BC, DE, HL) with the contents of their corresponding alternate registers.
EX AF, AF'	Exchange contents of the accumulator and the flag register with the contents of the corresponding alternate registers.
EX (SP), IX	Exchange the contents of an index register or HL register with the contents of the two top locations of the stack.
EX (SP), IY	
EX (SP), HL	
EX DE, HL	Exchange the contents of the DE register with the contents of HL register.

General Characteristics

1. These are 1-byte instructions (except instructions EX (SP), IX and EX (SP), IY) and are similar to copy instructions, except that the copying is performed both ways—from the source to destination and vice versa.
2. These instructions do not affect the flags.
3. The Exchange instructions related to the alternate registers can be used in place of PUSH and POP instructions for efficient execution.

ILLUSTRATIVE PROGRAM 1: EXAMINING AND MANIPULATING FLAGS

10.2

The following program demonstrates that the flags can be examined and manipulated if necessary. The program clears all the flags and shows that the increment instruction does not set the CY flag but does affect the other flags.

10.21 Problem Statement

Write a program to perform the following functions:

1. Clear the accumulator and all the flags.
 2. Load FF_H into the accumulator, increment the contents of the accumulator, and display how the S, Z, and CY flags are affected by the increment instruction.

10.22 Problem Analysis

The problem is concerned with clearing the flags and examining the flags after the increment instructions. There are no instructions in the set that can change the contents of the flag register directly. However, the flags can be examined and modified by storing the flags on the stack and retrieving them in any one of the general-purpose registers.

10.23 Program

START: LD SP, STACK	;Initialize the stack
LD DE, 0000H	;Load register DE with 00 to clear A and F registers
PUSH DE	;Place (DE) on stack
POP AF	;Clear accumulator and flags
LD A, FFH	;Load the given byte in A
INC A	;Increase (A) beyond FFH
PUSH AF	;Place flags on stack
POP DE	;Retrieve flags in register E
LD A, E	
AND 11000001B	;Mask all flags except S, Z, and CY
OUT (PORT1), A	;Display flags
HALT	;End of program

10.24 Program Description

After initializing the stack, register DE is cleared and the contents of DE placed on the stack. Assuming the stack is initialized at 2099_H , the instruction PUSH DE clears the locations 2098_H and 2097_H , and the stack pointer points to 2097_H . The next instruction POP AF copies the top of the stack into the flag register and clears the accumulator and the flags. (In this problem, we are not particularly interested in the contents of registers D and A.)

After loading the byte FF_H into the accumulator and incrementing it, the flags are placed on the stack and retrieved in register E. The masking instruction saves the status of S, Z, and CY flags and eliminates the others. The increment instruction increases the accumulator contents to 00; however, it does not affect the CY flag. Therefore, the output will be 40_H , indicating Z flag set and S and CY reset as shown:

Flags	S	Z	H	P/V	N	CY	
	0	1	0	0	1	0	
AND							
Masking Byte	1	1	0	0	0	1	(C1 _H)
	0	1	0	0	0	0	(40 ₁₆)

SUBROUTINE

10.3

A **subroutine** is a group of instructions written separately from the main program to perform a function that can be used repeatedly in the main program. For example, if a time delay is required between three successive events, a time delay subroutine can be written once, instead of three times. The subroutine is written separately from the main program, and is called by the main program when needed. The subroutine technique enables an efficient use of memory.

A subroutine is implemented with two associated instructions: Call (call a subroutine) and Return (return from the subroutine). The Call instruction is written in the main program (except in the nested subroutines) to call a subroutine, and the Return instruction is written in 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, are stored on the stack, and the program execution is transferred to the subroutine address. When the Return instruction is executed at the end of the subroutine, the memory address stored in the stack is retrieved and the sequence of execution is resumed in the main program. The procedure is demonstrated in Examples 10.2 and 10.3.

10.31 Subroutine Instructions

The Z80 microprocessor has two groups of instructions to implement the subroutine technique: unconditional and conditional.

Opcode	Operand	Description
CALL	16-bit	<p>Call subroutine unconditionally located at the memory address specified by 16-bit operand.</p> <p>This instruction places the address of the next instruction on the stack and transfers the program execution to the subroutine address.</p>
RET		<p>Return unconditionally from the subroutine.</p> <p>This instruction locates the return address on the top of the stack and transfers the program execution back to the calling program.</p>

Conditional Call and Return

CALL	Z, 16-bit	Call subroutine if Z flag is set (Z = 1)
CALL	NZ, 16-bit	Call subroutine if Z flag is reset (Z = 0)
CALL	C, 16-bit	Call subroutine if CY flag is set (C = 1)
CALL	NC, 16-bit	Call subroutine if CY flag is reset (C = 0)
CALL	M, 16-bit	Call (On Minus) if S flag is set (S = 1)
CALL	P, 16-bit	Call (On Plus) if S flag is reset (S = 0)
CALL	PE, 16-bit	Call (On Parity Even) if P/V flag is set (P/V=1)
CALL	PO, 16-bit	Call (On Parity Odd) if P/V flag is reset (P/V=0)

RET	Z	Return if Z flag is set (Z = 1)
RET	NZ	Return if Z flag is reset (Z = 0)
RET	C	Return if CY flag is set (C = 1)
RET	NC	Return if CY flag is reset (C = 0)
RET	M	Return (On Minus) if S flag is set (S = 1)
RET	P	Return (On Plus) if S flag is reset (S = 0)
RET	PE	Return (On Parity Even) if P/V flag is set (P/V = 1)
RET	PO	Return (On Parity Odd) if P/V flag is reset (P/V = 0)

General Characteristics

1. The Call instructions are 3-byte instructions; the second byte specifies the low-order byte and the third byte specifies the high-order byte of the subroutine address.
2. The Return instructions are 1-byte instructions.
3. A Call instruction must be used in conjunction with a Return instruction (conditional or unconditional) in the subroutine.

Example 10.2

The main program begins at 2000_H and has a Call instruction at location 2025_H . The subroutine is located at 2050_H ; it ends at location 2065_H with a Return instruction. Explain the flow of program execution.

Solution

The Call instruction is a 3-byte instruction and is stored in locations 2025_H , 2026_H , and 2027_H , and the subroutine is located from 2050_H to 2065_H (Figure 10.4).

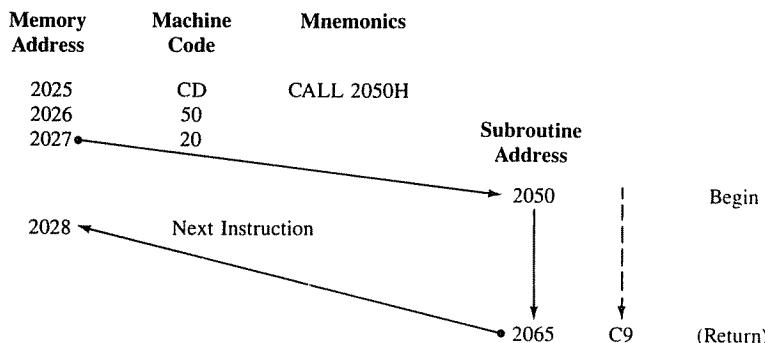


FIGURE 10.4
Program Flow in Call Execution

When the Z80 fetches the complete 3-byte instruction, the program counter holds 2028_{H} , always one address ahead of the execution. When the instruction is decoded, the Z80 recognizes that it is a Call instruction. It places the contents of the program counter (2028_{H}) onto the stack then transfers the program execution to location 2050_{H} as shown in Figure 10.4. At the end of the subroutine, when the Z80 decodes the Return instruction, it retrieves the address (2028_{H}) from the top of the stack and returns the execution back to the main program at 2028_{H} . The Return instruction can be functionally interpreted as Jump to a memory location, the address of which is stored on the top of the stack; however, the Jump instruction would not increment the address in the stack pointer.

Assuming the stack is initialized at 2099_{H} , explain the machine cycles showing the bus and register contents during the execution of the CALL instruction in Example 10.2.

Example
10.3

The CALL instruction, located at $2025-27_{H}$, has five machine cycles with 17 T-states. Because it is a 3-byte instruction, the first three machine cycles are concerned with fetching or reading the three machine codes of the Call instruction. Before the program is transferred to the subroutine location, the Z80 writes the contents of the program counter onto the stack; thus, the remaining two machine cycles deal with Memory Write operations. The sequence of events in each machine cycle is as follows:

Solution

1. M_1 —Opcode Fetch: In this machine cycle, the contents of the program counter (2025_{H}) are placed on the address bus; the program counter is incremented to 2026_{H} , and the instruction code CD is fetched using the data bus. After the instruction is decoded and executed, the stack pointer is decremented by one to 2098_{H} (see Figure 10.5).
2. M_2 and M_3 —Memory Read: These are two Memory Read operations during which the 16-bit address of the subroutine (2050_{H}) is fetched. The low-order address (50_{H}) is fetched during M_2 and placed into the internal register Z. The high-order address (20_{H}) is fetched during M_3 and placed into register W. The program counter is incremented once each machine cycle, to 2027_{H} and then to 2028_{H} , pointing to the next instruction (Figure 10.5).
3. M_4 and M_5 —Storing of Program Counter: Because this is a Call instruction, the contents of the stack pointer (2098_{H}) are placed onto the address bus in M_4 . The high-order byte of the program counter ($PCH = 20$) is placed onto the data bus and stored in the stack location 2098_{H} , and the stack pointer is decremented to 2097_{H} .

During machine cycle M_5 , the address (2097_{H}) from the stack pointer is placed onto the address bus, and the low-order byte (28_{H}) of the program counter is placed onto the data bus and stored on the stack at 2097_{H} . At the same time, the contents (2050_{H}) of the W and Z registers are placed onto the address bus, and the program execution is transferred to the subroutine. Figure 10.5 shows the bus and register contents during the execution of these five machine cycles.

Instruction: CALL 2025H

Memory Address	Code	Mnemonics
2025	CD	CALL 2050H
2026	50	
2027	20	

Machine Cycles	Stack Pointer 2099	Address Bus (AB)	Program counter (PCH) (PCL)	Data Bus (DB)	Internal Registers (W) (Z)
M ₁ Opcode Fetch	2098 (SP-1)	2025	2026	CD Opcode	—
M ₂ Memory Read		2026	2027	50 Operand	(50)
M ₃ Memory Read	2098	2027	2028	20 Operand	(20)
M ₄ Memory Write	2097 (SP-2)	2098		20 (PCH)	
M ₅ Memory Write		2097		28 (PCL)	(20) (50)
M ₁ Opcode Fetch of Next Instruction		2050 (W) (Z)	2051		2050

FIGURE 10.5
Register and Bus Contents During Execution of CALL Instruction

Example 10.4

Explain the machine cycles of the Return instruction located at 2065H in Example 10.2. Show the bus and register contents.

Solution

In Example 10.2, the RET instruction is stored at the end of the subroutine in memory 2065H. This is a 1-byte instruction and has three machine cycles. The contents of registers and buses during the execution of these machine cycles are shown in Figure 10.6.

M₁ is a normal Opcode Fetch cycle that fetches the code C9 and increments the program counter to 2066H. However, during M₂ the normal operation is suspended, and the address in the stack pointer (2097H) is placed onto the address bus in place of the

Instruction: RET

Memory Address	Code	Mnemonics	Stack Pointer	Contents of Stack Memory
2065	C9	RET	2097	2097 28 2098 20

Machine Cycles	Stack Pointer	Address Bus (AB)	Program Counter	Data Bus (DB)	Internal Registers (W) (Z)
M ₁ Opcode Fetch	2097	2065	2066	C9 Opcode	
M ₂ Memory Read	2098	2097		28 (Stack)	28
M ₃ Memory Read	2099	2098		20 (Stack-1)	20
M ₁ Opcode Fetch of Next Instruction		2028 (W) (Z)	2029		2028 (W) (Z)

FIGURE 10.6

Register and Bus Contents During Execution of RET Instruction

contents of the program counter. At the top of the stack, the return address (2028_H) is stored. During M₂ the low-order byte (28_H) is fetched and placed into the internal register Z, and during M₃ the high-order byte (20_H) is fetched and placed in register W. The stack pointer returns to its original address, 2099_H .

In the first cycle of the next instruction, the contents of the W and Z registers (2028_H) are placed onto the address bus, and the program sequence is transferred back to the main program at 2028_H .

ILLUSTRATIVE PROGRAM 2: TRAFFIC SIGNAL CONTROLLER

10.4

This program controls the traffic signal lights by turning them on or off at a specified interval. The subroutine technique is used to write the delay routine, and a register pair is used as a delay register. The significance of using the register pair is that the increment/decrement instructions do not affect any flags. Thus, a procedure needs to be built in to set the zero flag when the delay count in the register pair reaches zero.

10.41 Problem Statement

Write a program to provide the specified 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 and off by the data bits of an output port as shown.

Lights	Bit	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 only when the Green light is on.

10.42 Problem Analysis

The problem is primarily concerned with providing various time delays for a complete sequence of 40 seconds. The lights and signs can be turned on by providing logic 1s and turned off by providing logic 0s to appropriate data bits of the output port. 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
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	
0									
(15) ↓	0	1	0	0	0	0	0	1	= 41H
15									
(5) ↓	1	0	0	0	0	1	0	0	= 84H
20									
(20) ↓	1	0	0	1	0	0	0	0	= 90H
40									

The output needs three different codes at three different intervals as shown above. Three different delays (15, 5, and 20 seconds) can be conveniently obtained by writing a half-second delay subroutine and specifying the delay instructions as necessary. This is known as **parameter passing** (discussed in Section 10.5).

10.43 Program

```

LD SP, STACK      ;Initialize stack pointer
START: LD A, 01000001B ;Load bit pattern for Green light
                      ;and WALK sign
                      ;Turn on Green light and WALK sign
OUT (PORT1), A

```

LD B, 30	;Set up B to count 15 seconds
CALL DELAY	;Pass this information to DELAY routine
LD A, 10000100B	;Wait for 15 seconds
OUT (PORT1), A	;Load bit pattern for Yellow light and ;DON'T WALK sign
LD B, 10	;Turn on Yellow and DON'T WALK and turn ;off Green and WALK
CALL DELAY	;Set up B to count 5 seconds
LD A, 10010000B	;Wait for 5 seconds
OUT (PORT1), A	;Load bit pattern for Red and DON'T WALK
LD B, 40	;Turn on Red, keep DON'T WALK on, and turn ;off Yellow light
CALL DELAY	;Set up B to count 20 seconds
JP START	;Wait for 20 seconds
	;Go back to repeat the sequence
;This is 0.5 second delay routine and provides delay	
;according to the count in register B	
;Input: Appropriate delay count is specified in B; it is twice the number of	
desired seconds	
;Output: None	
;Registers Modified: B	
PUSH DE	;Save contents of DE and AF
PUSH AF	
LD DE, COUNT	;Load DE for 0.5 second delay
DEC DE	
LD A, D	;Place (D) in A for flag checking
OR E	;Set Z flag if (D) and (E) are both zero
JP NZ, LOOP	;Repeat if COUNT ≠ 0
DEC B	;End of 0.5 second delay, decrement B
JR NZ, WAIT	;Is it sufficient delay? If not, go back to WAIT
POP AF	;Retrieve contents of saved registers
POP DE	
RET	;Go back to main program

10.44 Program Description

Figure 10.7 shows the flowchart of this program; it includes the symbol of the predetermined process used for the delay subroutine. As shown in the flowchart (Figure 10.7), the main program turns on the specified lights and calls the delay routine. The program begins with loading the stack pointer; the initialization of the stack is essential to use Call instructions. The program loads the appropriate bit pattern into the accumulator, sends it to the output port, specifies the total delay in register B, and calls the delay routine. Register B is

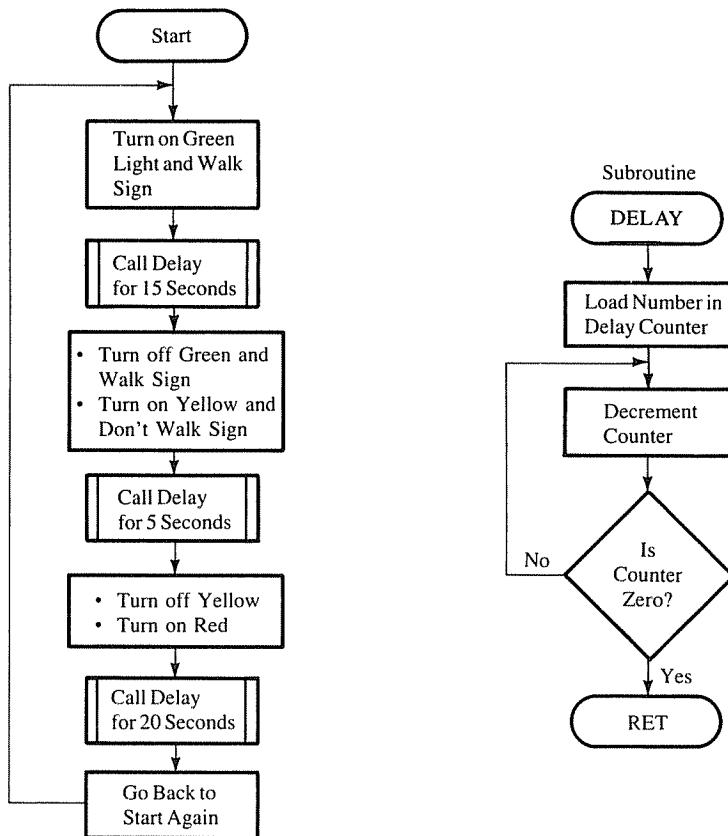


FIGURE 10.7
Flowchart for Traffic Signal Controller

loaded with a delay count in the main program, but the information in B is used in the subroutine.* This is called parameter passing.

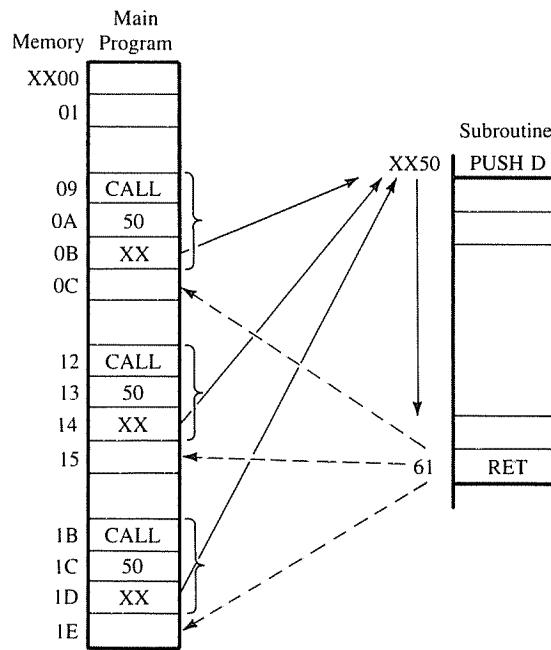
The `DELAY` subroutine is similar to delays discussed in the previous chapter. However, this routine has four additional features:

1. At the beginning of the subroutine, the registers being used (A, D, and E) for the delay loop are saved on the stack and retrieved before the end of the routine.
2. The subroutine ends with the `Return` instruction.
3. The delay loop uses register pair DE; however, the decrement instruction for register pairs does not affect any flags. When COUNT is decremented to 0, the Z flag needs to

*In this program, the delay counts are shown in decimal and the bit patterns are shown in binary; these numbers must be converted into Hex numbers for manual assembly. This is a standard industrial practice if you are using an assembler.

FIGURE 10.8

Multiple Call Subroutine



be set. To set the flag, the contents of register D are placed in A and logically ORed with the contents of E. The OR instruction sets the Z flag when both bytes, in D and E, are equal to 0; otherwise, the loop is repeated.

4. The WAIT loop includes a conditional relative Jump instruction, which requires 12 T-states if the specified condition is met; otherwise it takes seven T-states. In this example, when the Z flag is not set, the Jump instruction takes 12 T-states and the loop is repeated. When register B goes to zero, the Z flag is set and the Jump instruction takes seven T-states and the program proceeds to the next instruction. In general, the difference in execution time of these two conditions is insignificant.

The main program calls the subroutine three times as shown in Figure 10.8; this is known as a **multiple call subroutine**. In this subroutine, only two memory locations are needed for the stack. When a subroutine is called, the stack pointer is decremented by two, and when the program returns, the stack pointer is incremented by two; thus, the stack pointer returns to the initialization address.

SUBROUTINE DOCUMENTATION AND PARAMETER PASSING

10.5

In a large program, subroutines are scattered all over the memory map and are called from many locations. Information may be passed between a calling program and a subroutine, a

procedure called **parameter passing**. Typically, software design and implementation are performed by a team. Since subroutines and calls may be written by different team members, it is important to document a subroutine clearly so that the team members will know the consequences of using a subroutine. The documentation should include at least

1. Functions of a subroutine
2. Input/Output parameters
3. Registers modified—often called “destroyed”
4. List of other subroutines called by this subroutine.

The **DELAY** subroutine in Traffic Signal Controller shows one example of subroutine documentation.

FUNCTIONS OF THE SUBROUTINE

This is a brief summary of the purpose of the subroutine and what functions it performs. The user should be able to understand whether it is an appropriate subroutine without going through its instructions.

INPUT/OUTPUT PARAMETERS

The parameters (information) passed on to a subroutine are listed as inputs, and the parameters returned to the calling program are listed as outputs. For example, in the **DELAY** subroutine, how many times the loop is repeated is dependent on the count in register B. This parameter is supplied by the main program and used by the subroutine. Therefore, the count in register B becomes an input to the subroutine. However, this subroutine passes no information back to the main program, so there is no output.

When many parameters must be passed, R/W Memory locations are frequently used to store them, and memory pointers (HL or index registers) are used to point to those locations. The stack can also be used to store and pass parameters.

REGISTERS MODIFIED OR DESTROYED

Registers used in a subroutine may have been 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 to the calling program.

In the **DELAY** routine, the contents of DE and AF registers are saved on the stack at the beginning of the routine because these registers are used in the delay loop. Their contents are restored at the end of the routine using the LIFO method.

LIST OF SUBROUTINES CALLED

If a subroutine is calling other subroutines, the user should be provided with that information. This enables the user to check what parameters need to be passed to various subroutines and what registers are modified in the process.

ADVANCED SUBROUTINE CONCEPTS

10.6

Subroutines can be classified into various categories; commonly used categories include

1. Multiple Call
2. Nested
3. Multiple Ending

MULTIPLE CALL SUBROUTINE

This is a subroutine called from many locations in the main program. For example, the **DELAY** routine in Section 10.43 is a **multiple call subroutine**. These types of routines are easy to trace and need minimal stack space.

NESTED SUBROUTINE

A subroutine called by another subroutine is said to be **nested**. The extent of nesting is limited only by the number of available stack locations. When a subroutine calls another subroutine, all return addresses are stored on the stack.

Nested subroutines are shown in Figure 10.9. The main program calls Subroutine I from location 2050_{H} . The address of the next instruction, 2053_{H} , is placed on the stack, and the program is transferred to Subroutine I at 2090_{H} . Subroutine I calls Subroutine II from location $209A_{H}$. The address $209D_{H}$ is placed onto the stack, and the program is transferred to Subroutine II. Figure 10.9 shows how the sequence of execution returns to the main program.

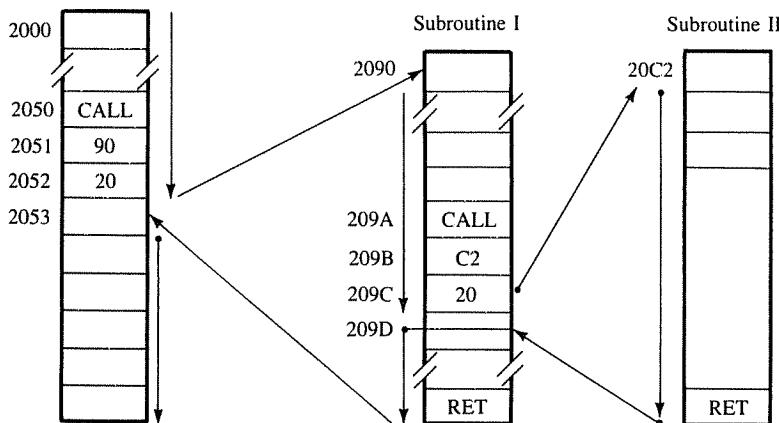
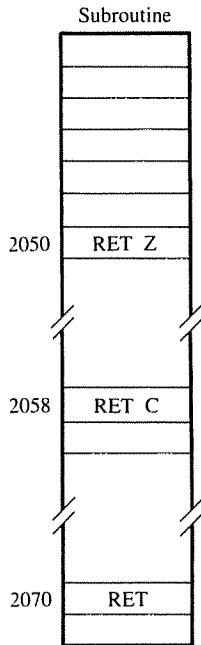


FIGURE 10.9
Nested Subroutines

FIGURE 10.10
Multiple Ending Subroutine



MULTIPLE ENDING SUBROUTINE

A subroutine that can be terminated at more than one place is called a **multiple ending subroutine** as shown in Figure 10.10. The subroutine has two conditional returns (RET Z—Return On Zero—and RET C—Return On Carry) and one unconditional return (RET). If the Z flag is set, the subroutine returns from location 2050_H, and if the CY flag is set, it returns from location 2058_H. If neither flag is set, the subroutine returns from location 2070_H.

10.7 ILLUSTRATIVE PROGRAM 3: BCD COUNTER AND ITS SEVEN-SEGMENT LED DISPLAY

This program is concerned with designing a BCD (Binary Coded Decimal) counter that counts in BCD and displays BCD digits at seven-segment LED output ports. This program demonstrates

1. How to adjust a binary number to a decimal number using the instruction DAA.
 2. Table look-up technique to get seven-segment codes.
 3. Nested and multiple ending subroutines.

10.71 Problem Statement

Design a BCD counter to count from 0 to 60_{BCD} and display each two-digit BCD count at two seven-segment LED ports (PORT0, PORT1); the time interval between each count

should be one second. The display should go up to 59, and when the count reaches 60, the counter should be reset to 00. The output ports have common cathode seven-segment LEDs, and the appropriate codes to display digits from 0 to 9 are stored in memory locations labelled as CODE.

10.72 Problem Analysis

This problem can be divided into three segments: (1) counting in BCD, (2) getting seven-segment LED code, and (3) creating a one-second delay. (See Section 11.5 for additional explanation on BCD numbers.)

COUNTING IN BCD

The designing of a binary (or Hex) counter using software instructions is a simple process; it involves clearing a register and incrementing the count at a given interval until the final count is reached. However, to count in BCD some adjustment is necessary; this is done by using the instruction DAA. The concepts underlying the DAA instruction are explained in the following paragraph.

The microprocessor is a binary machine; it does not recognize BCD numbers. In BCD, any number from A through F is invalid. When the count goes from digit 9 to A, it should be adjusted to 10 by adding 6 in binary as shown.

$$\begin{array}{r}
 0 \text{ A} = 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\
 + 0 \ 6 = 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \\
 \hline
 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 = 10
 \end{array}$$

The microprocessor interprets this number as equivalent to 10 in Hex; however, we view the low-order four bits as 0 and the high-order four bits as 1, stored side by side in an 8-bit register. This is called **packed BCD**. To display 10_{BCD} , the low-order and high-order bits need to be separated (called **unpacking**) and displayed by two seven-segment LEDs. Now let us see how we can adjust a larger number by using the same technique of adding 6 when the digit goes beyond 9. For example, when the count goes from 99 to 9A, in BCD, it should be 100.

$$\begin{array}{r}
 9 \text{ A} = 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\
 \text{The value of low-order} \\
 \text{four bits} > 9; \text{ add 6} \\
 \hline
 & + 0 \ 1 \ 1 \ 0 \\
 & \hline
 & 1 \ 0 \ 0 \ 0 \ 0
 \end{array}$$

Because of CY, the value of high-order bits > 9 ; add 6 and CY to high order bits.

$$\begin{array}{r}
 & + 0 \ 1 \ 1 \ 0 \\
 1 \ 0 \ 0 \ 0 \ 0 & 0 \ 0 \ 0 \ 0 \\
 \text{CY} & \text{CY}
 \end{array}
 = 1 \ 0 \ 0_{BCD}$$

As expected, this becomes a three-digit BCD number requiring three LEDs. The above steps are performed internally by the DAA instruction.

INSTRUCTION

DAA : Decimal Adjust Accumulator

1. This is a 1-byte instruction. After an arithmetic operation, this instruction adjusts an 8-bit number in the accumulator to form two packed BCD numbers by using the process described above.
2. It uses the H and C flags to perform the adjustment.
3. In arithmetic addition, if there is a carry from bit D₃ to bit D₄, the Half Carry flag (H) is set. Similarly, in subtraction, if there is a borrow from bit D₄, the H flag is set. The DAA instruction uses the H and CY flags internally to adjust the result to BCD digits. The CY flag is used in 16-bit addition; this is discussed in the next chapter.

It must be emphasized that the instruction DAA

- Adjusts the result of a BCD addition or subtraction.
- Does not convert a binary number into BCD numbers.

GETTING SEVEN-SEGMENT LED CODES

When a BCD number is to be displayed by a seven-segment LED, it should be converted into its seven-segment code. The code is determined by hardware considerations such as common cathode versus common anode LED. For example, to display 0 in the common anode LED, six elements are turned on by supplying logic 0 and the center element is turned off by supplying logic 1. (See Section 13.54 for hardware details of seven-segment LEDS and related codes.)

The code has no direct relationship to binary numbers. The codes of BCD digits (0–9) are stored sequentially in memory and obtained using the **table look-up technique**. To use the technique, the BCD number must be unpacked. For example, the BCD number 28 must be converted into binary equivalents of 02_{BCD} and 08_{BCD}. Then, the conversion program locates the code of a digit based on its magnitude and copies the code into the accumulator (or register) to send out to the seven-segment LED port.

ONE-SECOND DELAY

A one-second delay can be generated by using a register pair, similar to the DELAY subroutine in Traffic Signal Controller (Section 10.4). However, there is no need to pass a parameter to repeat the delay loop from the main program; it can be written into the routine itself.

10.73 Program

This program consists of a main program and five subroutines: UNPACK, DISPLAY, LOOKUP, DELAY, and UPDATE.

MAIN: LD SP, STACK	;Initialize the stack
LD B, 00H	;Load initial BCD count
NEXT: CALL UNPACK	;Unpack BCD number and store digits in output buffer
CALL DISPLAY	;Get codes and display at ports
CALL DELAY	;Wait for one second
CALL UPDATE	;Go to next count and adjust for BCD
JP NEXT	;Continue

UNPACK: ;This subroutine unpacks the BCD number from the accumulator
 ; and stores two unpacked BCD digits in memory output buffers
 ; BUFF1 and BUFF2.
 ;Input: Packed BCD number in register B
 ;Output: BCD₁ in BUFF1 and BCD₂ in BUFF2
 ;Registers modified: Accumulator and HL

```

LD  HL, BUFF1      ;Set up HL as memory pointer for BUFF1
LD  A, B           ;Get BCD number
AND 0FH            ;Keep low-order 4 bits (BCD1)
LD  (HL), A         ;Store BCD1 in BUFF1
INC  HL             ;Point to BUFF2
LD  A, B           ;Shift BCD2 to D3-D0 and insert 0 from left
SRL  A
SRL  A
SRL  A
SRL  A           ;Store BCD2 in BUFF2
LD  (HL), A
RET

```

DISPLAY: ;This subroutine gets BCD digits from the output buffer,
 ; calls the LOOKUP routine, and displays digits at the
 ; output ports with appropriate codes.

```

;Input: Memory pointer to BUFF2 in HL registers
;Output: None
;Registers modified: HL registers
;Calls LOOKUP subroutine
LD  A, (HL)        ;Get BCD2 from BUFF2
CALL LOOKUP        ;Get seven-segment code
OUT (PORT2), A     ;Display BCD2 at PORT2
DEC  HL             ;Move memory pointer to BUFF1
LD  A, (HL)        ;Get BCD1 from BUFF1
CALL LOOKUP        ;Get seven-segment code
OUT (PORT1), A     ;Display BCD1 at PORT1
RET

```

LOOKUP: ;This subroutine takes an unpacked BCD digit, updates the
 ; memory pointer, and gets the seven-segment code of the
 ; digit from memory.

```

;Input: Unpacked BCD digit in the accumulator
;Output: Seven-segment code in the accumulator
;Registers modified: Accumulator
PUSH HL            ;Save (HL) on the stack
LD  HL, CODE        ;Set up HL as memory pointer to LED code

```

ADD A, L	;Add memory pointer to digit to be displayed
LD L, A	;Update memory pointer to locate LED code
LD A, (HL)	;Get LED code
POP HL	;Retrieve from the stack
RET	;Return to the calling program
CODE: DB CODE0	;The following ten locations store
CODE1	; seven-segment LED codes from 0 to 9
CODE2	
CODE3	
CODE4	
CODE5	
CODE6	
CODE7	
CODE8	
CODE9	

UPDATE: ;This subroutine updates the BCD count and adjusts for BCD
 ;number. When the BCD count reaches 60, it resets the
 ;counter.

;Input: Count in register B	
;Output: Updated and BCD adjusted count in B	
;Registers modified: Accumulator and register B	
LD A, B	;Get the last count
ADD A, 01H	;Update the count
DAA	;Adjust for BCD
LD B, A	;Save the count
CP 60H	;Is count 60?
RET NZ	;If not, go back to main program
LD B, 00H	;Reset the counter
RET	

DELAY: ;This is a one-second delay routine

;Input/Output: None	
;Registers Modified: None	
PUSH DE	;Save contents of DE, AF, BC
PUSH AF	
PUSH BC	
LD B, 10	;Load B to repeat loop ten times
WAIT: LD DE, COUNT	;Load DE for 100 ms delay
LOOP: DEC DE	
LD A, D	;Place (D) in A for flag checking
OR E	;Set Z flag if (D) and (E) are both zero

JP NZ, LOOP	;Repeat if COUNT ≠ 0
DEC B	;End of 100 ms delay, decrement B
JP NZ, WAIT	;Is it one-second delay? If not, go back to WAIT
POP BC	;Retrieve contents of saved registers
POP AF	
POP DE	
RET	;Go back to the calling program

10.74 Program Description

In this program, the problem is divided into various small tasks, each of which is assigned to a subroutine. The main program initializes the stack and the counter and calls subroutines in a sequence.

The first subroutine unpacks the BCD number and stores the unpacked digits in the output buffer BUFF1 and BUFF2. The unpacking is performed by separating the two digits and shifting the high-order digit into low-order bit positions D₃–D₀. For example, if the BCD number is 27, the digit 07 will be stored in BUFF1 and 02 will be stored in BUFF2, both in binary format.

The next subroutine DSPLAY is an example of the nested routines. It gets the BCD digits from the output buffer and calls the LOOKUP subroutine. In the LOOKUP routine, the seven-segment codes are stored sequentially from 0 to 9, and the HL register points to the first code. The DSPLAY subroutine supplies a digit in the accumulator which is added to register L, and the sum is saved in L. In effect, the memory pointer is moved to the location where the code of the digit is stored. For example, if the memory pointer is initially at the starting code location 2050_H and the digit is 02, the pointer is shifted after addition to 2052_H, where the code for the digit 2 is stored. Then the code is moved to the accumulator and passed back to the subroutine DSPLAY, which displays the code at the output port PORT2. The same process is repeated for BCD₁. However, this technique will work properly only if the code is stored on the same memory page (see Assignment 13).

The UPDATE subroutine is an example of a multiple ending subroutine. This subroutine adds one to the previous count and adjusts for BCD using the instruction DAA. It compares the count with 60 and uses the conditional Return instruction to check for the Z flag. If Z is reset, the subroutine returns to the main program, and if it is set, it clears the register B to start the counter again.

MODULAR PROGRAMMING AND DEBUGGING

10.8

The program discussed in the last section is an example of industry-standard software even though it is a small program. The problem is divided into small tasks, and a subroutine is written for each task. The main program consists primarily of calling these subroutines. This is known as the **modular approach**.

The modular approach has several advantages. The approach provides flexibility in writing and modifying programs, especially in a team project. It is also easy to debug individual modules; each module can be tested separately for the expected outputs because each module has a specific task to perform. For example, the UNPACK subroutine is expected to accept a packed BCD number, unpack it, and store the digits in the output buffer. This can be written and tested separately independent of any other modules. Similar tests can be performed for the other modules. In addition, when all the modules are combined, the entire program can be debugged by setting breakpoints at the end of each module.

DEBUGGING MODULAR PROGRAMS

At the outset, it appears that if each module is tested individually, the whole program should work perfectly when they are combined, and the programmer can live happily ever after. But this almost never happens. As soon as the modules are combined, the interaction between the modules can cause quite a few trouble spots. Following is a list of common errors.

1. Mismatch between PUSH and POP instructions, the number of PUSH instructions being different from the number of POP instructions.
2. Improper sequence of PUSH and POP instructions (not following the LIFO sequence).
3. Destroying the contents in other modules.
4. Passing on wrong parameters or failure to pass parameters.
5. Failure to initialize the stack.
6. Failure to end a subroutine with a Return instruction.
7. Using a wrong flag for a conditional Call and Return instruction.

SUMMARY

- The stack is a group of memory locations in the system's R/W memory that is defined by loading an address into the stack pointer register.
- The programmer uses the stack to store and retrieve the contents of register pairs temporarily during the execution of a program, and the microprocessor uses the stack to store and retrieve the return address when a subroutine is called.
- The contents of register pairs are stored on the stack by using PUSH instructions, and retrieved by using POP instructions. The stack pointer tracks the storing and retrieving process by adjusting its address.
- When a PUSH instruction is executed, the stack pointer register is decremented once and the high-order byte stored; then the stack pointer is decremented again and low-order byte stored.
- When a POP instruction is executed, the byte at the top of the stack is retrieved in the low-order register and the stack pointer incremented; the next byte is retrieved in the high-order register and the stack pointer incremented again.

- The Exchange instructions (EXX and EX AF, AF') are used to exchange contents between general-purpose registers and alternate registers; these instructions can be used instead of PUSH and POP instructions for temporary storage.
- A subroutine is a group of instructions that performs a subtask of repeated occurrence; it is written separately from the main program.
- The program is transferred to a subroutine by using a Call instruction (conditional or unconditional) and returned to the calling program by using a Return (conditional or unconditional) instruction. A Call instruction should always be used in conjunction with a Return instruction.
- When a Call instruction is executed, the return address is stored on the stack before the program execution is transferred to the subroutine.
- When a Return instruction is executed, the program is transferred to the address stored at the top two locations of the stack.
- The DAA instruction adjusts a binary number in the accumulator to its BCD digits by using the Half-Carry (H) and the Carry (CY) flags. When a group of 4 bits (high or low) exceeds the magnitude 9_{10} , the instruction adds 6 and adjusts digits to their BCD values.
- In a seven-segment LED display, the codes required to display digits are unrelated to the binary values and dependent on hardware configuration. Therefore, the table look-up technique is used to display these digits.
- Modular programming is a software design process whereby the programming problem is divided into subtasks (or modules) with definite functions to perform, and each module is written and tested separately as a subroutine. This approach provides flexibility in writing and ease in debugging the program.

ASSIGNMENTS

1. Specify the contents of the BC registers and the stack pointer after the execution of the following instructions.

LD SP, 20F5H
LD HL, 2055H
PUSH HL
POP BC

2. Specify the address in the stack pointer and the contents of memory locations 2090_H – 2099_H after the execution of the following instructions.

LD SP, 219AH
XOR A
LD H, A
LD L, A
LD B, 05H

```

LOOP: PUSH HL
      DEC B
      JP NZ, LOOP

```

3. Read the following program and answer the questions.

No. Instructions

1. LD SP, 84F9H
2. LD HL, 8138H
3. LD BC, 0001H
4. LD DE, 235AH
5. LD A, D
6. OR A
7. PUSH HL
8. PUSH AF
9. PUSH BC



20. POP AF

- a. What is stored in the stack pointer after the execution of instruction 1?
- b. What are the contents of locations $84F8_H$ and $84F7_H$ after the execution of instruction 7 (PUSH HL)?
- c. Specify the bytes in the accumulator and the flag register after the execution of the instruction OR A.
- d. What is the address in the stack pointer when instruction 9 is executed and what are the contents of the two top locations of the stack?
- e. Specify the status of the S, Z, and C flags after the execution of instruction 20 (POP AF).
4. If the stack pointer is initialized at $20C8_H$ and the Call instruction located at 2052_H calls the subroutine at 2075_H , specify the contents of the stack locations $20C7_H$ and $20C6_H$ and the contents of the stack pointer after the execution of the Call instruction.
5. The following program has a delay subroutine located at 2060_H . Read the program and answer the questions following.

Memory Addresses	Instructions	Comments
2000	LD SP, 2100H	;Main Program
2003	LD HL, 2050H	
2006	LD BC, 000AH	
2009	CALL 2060H	
200C	OUT (PORT1), A	



```

2060  PUSH HL          ;Delay Subroutine
2061  PUSH BC
2062  LD HL, A2FFH
      |
      ↓
2072  POP BC
2073  POP HL
2074  RET

```

- When the Call instruction located at $2009H$ is executed, specify the contents of the stack location $20FEH$ and the address in the stack pointer.
 - List the stack locations and their contents after the execution of the instructions PUSH HL and PUSH BC in the subroutine.
 - List the address in the stack pointer and the contents of the BC registers after the execution of the instruction POP BC.
 - Where does the program return after the execution of the instruction RET at the end of the subroutine?
 - Specify the contents of the stack pointer after the execution of the RET instruction.
6. The following program is a continuous Hex counter with an appropriate delay between two counts. Read the program and answer the questions.

Memory Addresses	Instructions	Comments
2000	LD SP, 20FAH	;Main program
2003	LD HL, 2065H	
2006	LD BC, 0010H	
2009	LD A, 00H	
200B	OUT (PORT1), A	
200D	CALL 2065H	
2010	INC A	
2011	JP 200BH	
↓		
2065	PUSH HL	;Delay Subroutine
2066	PUSH BC	
2067	LD HL, 10FFH	
206A	DEC HL	
↓		
2070	POP BC	
2071	RET	

- a. List the stack locations and their contents after the instruction PUSH BC is executed.
 - b. Where does the program return after the execution of the RET instruction and what are the contents of the stack pointer?
 - c. What is being displayed at PORT1? Explain your answer.
7. Write a program to add the two Hex numbers 8A and 93. Store the sum at location OUTBUF and the flag status at location OUTBUF - 1. Initialize the stack at an appropriate memory address.
8. Write a program to initialize the stack pointer at memory location STACK, add two Hex numbers (97 and A1), and store the flag status in location STACK - 2 and the accumulator contents in location STACK - 1.
9. Write a program to meet the following specifications.
 - Initialize the stack pointer at XX99_H.
 - Clear the memory locations starting from XX90_H to XX9F_H.
 - Load register pairs BC, DE, and HL with data 024F_H, 4835_H, and 2050_H, respectively.
 - Store the contents of the registers BC, DE, and HL on the stack.
 - Execute the program and verify the memory locations from XX90_H to XX9F_H.

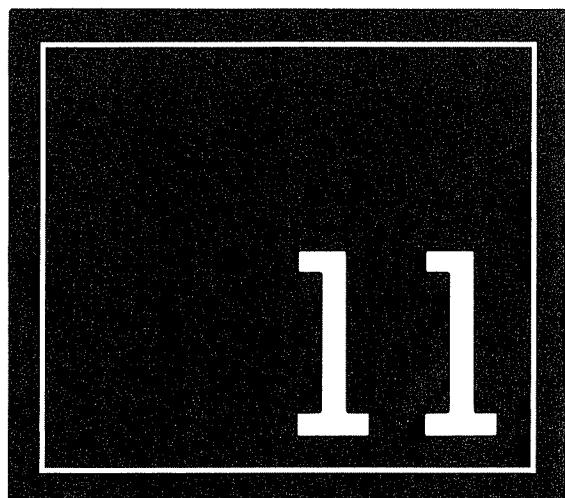
Note: For XX substitute high-order memory address (page number) of your system.
10. Write a program to clear the initial flags, load the data FFFF_H into register BC, and increment the register pair BC. Display the flag status at an output port or store it on the stack. Explain your results.
11. Write 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.
12. Write a program to simulate a flashing yellow light with a 750 ms period. Use bit D₇ to control the light. (Hint: To simulate flashing, the light must be turned on and off.)
13. Modify the subroutine LOOKUP to include the situation when the code is stored in a sequence but crosses the memory page boundary (for example, when the seven-segment code is stored from 20FA_H to 2103_H).

Application Programs and Software Design

In the last three chapters, most of the Z80 instructions were introduced and illustrated with examples and application programs, such as block transfer, time delays, bit manipulations, and some arithmetic operations. The subroutine technique, discussed in Chapter 10, provides us with the most powerful features of programming: modularity and flexibility.

In this chapter we first introduce the Z80 instructions dealing primarily with 16-bit operations (copy, exchange, and arithmetic) and illustrate them using the subroutine technique. The illustrations will include examples such as multiprecision addition, multiplication, division, and code conversions.

In computer applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) terminal is a standard Input/Output device in disk-based microcomputer systems. The code conversion subroutines demonstrated in this chapter are commonly used to exchange alphanumeric (letters and numbers) information between the microprocessor and peripherals such as terminals and printers.



OBJECTIVES

- Explain and illustrate the instructions used to copy or exchange 16-bit data between registers and memory locations.
- Explain and illustrate the instructions that perform addition or subtraction with carry.
- Write subroutines to perform arithmetic operations such as multiplying, dividing, and adding numbers larger than 8 bits.
- Write subroutines to perform code conversions for BCD, binary, and ASCII characters.
- Write a program to illustrate the modular approach in software design.

11.1 16-BIT OPERATIONS

Most of the instructions discussed in the last several chapters deal primarily with 8-bit data. However, there are instances when we need to manipulate data larger than eight bits, especially in arithmetic manipulations and stack operations. Even though the Z80 is an 8-bit microprocessor, its architecture allows specific combinations of 8-bit registers to form 16-bit registers. There are several instructions in the set to manipulate data larger than 8 bits. These instructions will be introduced in this section and illustrated with examples.

11.11 16-Bit Data Copy and Data Exchange Group

Opcode	Operand	Bytes	Description
LD	HL, (16-bit)	3	Load HL register from the contents of memory specified by 16-bit operand and the next memory location.
LD	(16-bit), HL	3	Load the contents of HL register into memory location specified by 16-bit operand and the next memory location.
LD	SP, HL	1	Load the contents of HL register into the stack pointer register.
JP	(HL)	1	Load the contents of HL register into the program counter. This instruction is equivalent to LD PC, HL.
EX	DE, HL	1	Exchange the contents of DE and HL.
EX	(SP), HL	1	Exchange the contents of HL register and the contents of the top two locations of the stack.

General Characteristics

1. The first four instructions copy 16-bit data between the HL register pair and memory, the stack pointer, and the program counter.
2. The Exchange instructions exchange (copy both ways) 16-bit data.
3. These instructions do not affect the flags.

The memory locations 2050_H and 2051_H contain $3F_H$ and 42_H respectively, and the register pair DE contains $856F_H$. Write instructions to exchange the contents of DE with the contents of the memory locations.

Example
11.1

Before Instructions:	D	85	6F	E	2050	3F	Memory Contents
					2051	42	

Solution

LD HL, (2050H) ;	H	42	3F	L	2050	3F	
					2051	42	
EX DE, HL ;	D	42	3F	E			
	H	85	6F	L			
LD (2050H), HL ;					2050	6F	
					2051	85	

11.12 Arithmetic Instructions: Addition With Carry

ADC A, r These instructions add the contents of the operand, the carry, and the accumulator, and the result is placed in the accumulator.
 ADC A, 8-bit Generally, these instructions are used to add numbers larger than 8 bits, as shown in Example 11.2.
 ADC A, (HL)

Registers BC contain 2793_H , and registers DE contain 3182_H . Add these two 16-bit numbers, and place the sum in memory locations 2050_H and 2051_H .

Example
11.2

Before Instructions:	B	27	93	C
	D	31	82	E

Solution

LD A, C	A	93	X	F
ADD A, E	A	15	C = 1	F
				$93_H \rightarrow (A)$
				$+ 82_H \rightarrow (E)$
				$\overline{1} \overline{15}_H$
LD L, A	H	15		L
				1 15 _H

LD A, D	A	31	C = 1	F
ADC A, B				$31_H \rightarrow (A)$
				$+ 27_H \rightarrow (B)$
				1 $\rightarrow (\text{Carry})$
LD H, A	H	59	15	L
LD (2050H), HL				$59_H \rightarrow A$

The 16-bit addition is performed in two stages: add low-order bytes ($93_H + 82_H$), and then add high-order bytes ($27_H + 31_H$). These two stages are necessary because the addition is performed with the contents of the accumulator, which can hold only eight bits. If the addition of the low-order eight bits generates a carry, it should be added to the ninth bit position of the 16-bit numbers.

In this example, the addition of 93_H and 82_H generates the sum 15_H and a carry. The sum (15_H) is stored in register L, and the carry is added as a ninth bit to the high-order bytes 27_H and 31_H by the instruction ADC A, B. The result 59_H is saved in register H. Finally, the sum of the two 16-bit numbers (contents of HL) is stored in memory locations 2050_H and 2051_H .

11.13 Arithmetic Instructions: Subtraction With Carry

SBC A, r These instructions subtract the contents of the operand and the borrow from the contents of the accumulator, and the result is placed in the accumulator.

SBC A, 8-bit

SBC A, (HL)

Example
11.3

Registers BC contain 8538_H and registers DE contain $62A5_H$. Subtract the contents of DE from the contents of BC, and place the result in BC.

Solution

LD A, C	85	38
SUB E		
LD C, A	— 62	— A5
LD A, B	— 1 ←	93
SBC D	—	—
LD B, A	22	93

This is a 16-bit subtraction performed in two stages similar to that of 16-bit addition in the previous example. The low-order byte $A5_H$ is first subtracted from 38_H . The result (93_H) is saved in register C, and the borrow generated by this operation is subtracted as a ninth bit

from the high-order byte 85_H ($85_H - 62_H - 1$ Borrow = 22_H). Finally, the result is saved in register B.

11.14 Arithmetic Instructions: 16-bit Addition

ADD HL, rp : Add register pair to register HL

ADD HL, BC : This instruction adds the contents of the operand (register

ADD HL, DE pair or stack pointer) to the contents of the HL register, and

ADD HL, HL the result is placed in the HL register.

ADD HL, SP The Carry flag is altered to reflect the result of the 16-bit addition. No other flags are affected.

These instructions use HL as a 16-bit accumulator.

The register HL contains a 16-bit number (4 Hex digits). Write instructions to shift all the digits by four positions to the left and clear bits D_3-D_0 . For example, if the register contains 1231_H , it should have 2310_H after the instructions are executed.

Example
11.4

The solution to this problem lies in the fact that a binary number added to itself shifts the number to the left by one position. To shift the number by four positions, we need to add the number four times. This is equivalent to shifting Hex digits to the left by one position. This is illustrated below in the comment section, beginning with the number 1231_H in HL register.

Solution

ADD HL, HL	;	$1231 + 1231 =$	2462_H	HL
ADD HL, HL	;	$2462 + 2462 =$	$48C4_H$	HL
ADD HL, HL	;	$48C4 + 48C4 =$	9188_H	HL
ADD HL, HL	;	$9188 + 9188 =$	2310_H	HL

A practical example of this problem is entering a 16-bit address in a single-board microcomputer using the Hex keyboard. When a new key is pressed, the display drops off the most significant Hex digit, shifts the three remaining digits to the left, and places the new key as the least significant digit.

11.15 Miscellaneous Instructions

These instructions are used in a bit manipulation, usually in conjunction with rotate instructions. (See Appendix A for illustrative examples.)

CCF: Complement the carry flag

If C = 1, the CY flag is reset, and
if C = 0, the CY flag is set.

SCF: Set the Carry Flag. C is set to logic 1.

11.2

ILLUSTRATIVE PROGRAM: MULTIPRECISION ADDITION

Even though the Z80 has an 8-bit accumulator, it can be used to add multibyte numbers. The instruction set includes several instructions specifying arithmetic operations with carry (for example, add with carry or subtract with carry). 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, when a carry is generated, it is added to bit D₀ of the accumulator in the next operation. Therefore, these instructions are used primarily in multibyte addition and subtraction as illustrated in this program, which adds two 32-bit numbers and stores the result in memory locations of the first number.

11.21 Problem Statement

Two 32-bit numbers, each occupying four memory locations starting with the low-order byte, are stored in locations BUF1 and BUF2. Write a subroutine to add the numbers and store the result in BUF1. Pass the following parameters from the main program to the subroutine: the addresses of BUF1 and BUF2 and the number of bytes in a number.

11.22 Problem Analysis

To add multibyte numbers, two issues need to be resolved: (1) how to add a carry generated by the addition of low-order bytes to the next order bytes, and (2) how to save multibyte results. The first issue can be resolved by using the instruction ADC (Add With Carry), and multibyte results can be stored in memory by assigning appropriate memory locations.

In this problem, we need to set up one counter to count the number of bytes to be added and two memory pointers to point to BUF1 and BUF2.

11.23 Program

```
MAIN: LD HL, BUF1      ;Initialize memory pointers
      LD DE, BUF2
      LD B, 04H      ;Set up register B as a counter to specify
                      ;the size of number to be added
      CALL ADBYTE   ;Call subroutine to add multibyte numbers
      HALT

ADBYTE: ;This subroutine adds any two multibyte numbers and saves the
        ;result in memory by replacing the first number.
        ;Input: Addresses of numbers to be added in HL and DE registers
        ;       : The size of the number in bytes in register B
        ;Output: None, the result is stored in memory
        ;Modifies registers: B, DE, and HL

START: XOR A          ;Clear carry
NEXT: LD A, (DE)      ;Get byte from BUF2
```

```

ADC A, (HL) ;Add byte from BUF1
LD (HL), A ;Save partial result
INC HL      ;Update memory pointers
INC DE
DEC B       ;Update counter for next addition
JR NZ, NEXT ;If counter ≠0, get next-order byte
RET

```

11.24 Program Description

The main program initializes the memory pointers DE and HL, sets up register B to count four bytes (32 bits), and calls the subroutine ADBYTE. The subroutine begins by clearing the CY flag; if a carry has been generated by the calling program in the previous operation, it must be cleared. The program adds the low-order bytes from BUF1 and BUF2 by using the memory pointers and the instruction ADC and saves the partial result in the first memory location of BUF1. In the following instructions, the pointers are upgraded to point to the next bytes, and the counter is decremented. The loop is repeated until all four bytes are added, and the complete result, starting from the low-order byte, is stored in four memory locations of BUF1.

This subroutine can be used to add numbers of any size specified by the calling program in register B; however, it does not account for a carry generated by the last addition. For example, in this program, if the sum is larger than 32 bits, the result will have an error. To subtract multibyte numbers, this subroutine can be used by replacing the instruction ADC with SBC.

BINARY MULTIPLICATION

11.3

Multiplication can be viewed as repeated addition. For example, the product of 20 and 5 (20×5) can be obtained by adding 20 five times. To write a program, we can set up a counter for five and add 20 until the counter is zero. It is, however, a rather inefficient technique for a large multiplier. A more efficient technique can be devised by following the model of manual multiplication of decimal numbers.

$$\begin{array}{r}
 125 \\
 \times 101 \\
 \hline
 125 \\
 + 000 \\
 + 125 \\
 \hline
 12625
 \end{array}$$

Step 1: $(125 \times 1) = 125$
Step 2: Shift left and add $(125 \times 0) + 000$
Step 3: Shift left and add $(125 \times 1) + 125$

In this example, the multiplier multiplies each digit of the multiplicand, starting from the right, and adds the product by shifting to the left. When the multiplier digit is 1, we add multiplicand (125), and when the multiplier digit is 0, we add 0 (or nothing). The

same process can be applied in binary multiplication. The binary multiplier has only two digits, 0 and 1. Therefore, we can set up the algorithm to check each digit of the multiplier; if the digit is 1, we will add the multiplicand, and if it is 0, we will skip the addition.

11.31 Problem Statement

A multiplicand is stored in memory location BUF1 and a multiplier is stored in location BUF1 + 1. Write a main program to

1. Transfer the two numbers from memory locations to the H and L registers.
2. Store the product in the output buffer OUTBUF.

Write a subroutine to

1. Multiply two unsigned numbers placed in registers H and L.
2. Return the result into the HL pair.

11.32 Problem Analysis

The problem is concerned with multiplying two unsigned 8-bit numbers. Since the result will be larger than 8 bits; it will require a 16-bit register for storage. To implement the algorithm discussed above, we need to set up a counter to check eight bits of the multiplier. When the multiplier bit is 1, the multiplicand can be added to itself by using the instruction (ADD HL, HL) for 16-bit addition.

11.33 Program

MAIN:	LD SP, STACK	;Initialize the stackpointer
	LD HL, (BUF1)	;Place multiplicand in L and multiplier in H
	EX DE, HL	;Place multiplicand in E and multiplier in D
	CALL MLTPLY	;Multiply two numbers
	LD (OUTBUF), HL	;Store the product in the output buffer
	HALT	
MLTPLY:	;This subroutine multiplies two 8-bit unsigned numbers.	
	;Input: Multiplicand in register E and multiplier in register D	
	;Output: Result in HL register	
	;Modifies registers: A, B, D, E, H, and L	
	LD A, D	;Transfer multiplier to accumulator
	LD D, 0	;Clear D to save partial result
	LD HL, 0	;Clear HL
	LD B, 08H	;Set up register B to count eight rotations
NXTBIT:	RRA	;Check if multiplier bit is 1
	JR NC, NOADD	;If not, skip adding multiplicand
	ADD HL, DE	;If multiplier is 1, add multiplicand to HL
		;and place partial result in HL

```

NOADD: EX DE, HL      ;Place multiplicand in HL
       ADD HL, HL    ;and shift left
       EX DE, HL      ;Retrieve shifted multiplicand
       DEC B          ;One operation is complete, decrement counter
       JR NZ, NXTBIT  ;Go back to next bit if not done
       RET

```

11.34 Program Description

1. The objective of the main program is to demonstrate uses of 16-bit data copy and exchange instructions. The main program transfers the two bytes—multiplier and multiplicand—from memory locations to the HL registers and places them in the DE registers by using the exchange instruction. It calls the MLTPLY routine and places the result in the output buffer.
2. The multiplier routine follows the format—add and shift to the left—illustrated earlier. The routine places the multiplier into the accumulator and rotates it eight times until the counter (B) becomes zero. Register D is cleared to use it for 16-bit addition (ADD HL, DE).
3. After each rotation, when a multiplier bit is 1, the instruction ADD HL, DE performs the addition, and ADD HL, HL performs the shifting of bits to the left. When a bit is 0, the subroutine skips the instruction ADD HL, DE and just performs the shifting.

BINARY DIVISION

11.4

The division of two numbers can be performed by subtracting the divisor repeatedly from the dividend (the number to be divided) until the remainder becomes smaller than the divisor; the number of times the subtraction is repeated then becomes the quotient. For example, to divide 19 by 5, we can subtract 5 three times from 19 until the remainder is 4, and the quotient is equal to 3. Conceptually, this procedure is simple, but it can be time-consuming. An efficient algorithm that imitates the manual division of two decimal numbers can be devised. For example, to divide 203 by 5, we perform the following steps.

Divisor	Dividend	Quotient
5	2 0 3	
5	2	0

R = 2

Step 1: Take the most significant digit (MSD) from the dividend and divide it by the divisor. If the divisor > MSD, place the quotient as 0; otherwise find the remainder.

Step 2: Take the next digit from the dividend and combine it with the remainder of Step 1 and divide the new number by the divisor. Find quotient and remainder.

$$\begin{array}{r}
 5 \quad \boxed{2 \ 0} \quad 0 \ 4 \\
 -2 \ 0 \\
 \hline
 R = 0 \ 0
 \end{array}$$

Step 3: Take the last digit from the dividend and combine it with the remainder of Step 2. Divide it by the divisor. If the divisor > partial dividend, the quotient is 0. Find the remainder.

$$\begin{array}{r}
 5 \quad \boxed{0 \ 0 \ 3} \quad 0 \ 4 \ 0 \\
 R = 3 \quad 4 \ 0
 \end{array}$$

In this example, the quotient is 40 and the remainder is 3. From this illustration, two critical points need to be emphasized: (1) when the divisor is larger than the partial dividend, the quotient is 0, and (2) the quotients of successive steps are not added but combined with the previous result in appropriate columns.

In Step 2, integer 20 is divided by 5 to obtain the quotient 4, and then the remainder is obtained by subtracting the product of 5 and 4 from 20. This algorithm appears to be complicated; however, in binary numbers, it can be performed simply by subtraction because the quotient can only be either 1 or 0. Similarly, the remainder is obtained by subtracting the divisor from the dividend; the product of the divisor and the quotient is the same as the divisor when the quotient is 1.

11.41 Problem Statement

Write necessary subroutines to divide two unsigned 8-bit numbers. The calling program places the dividend in register E and the divisor in register D. The subroutine should place the quotient in register L and the remainder in register C.

11.42 Problem Analysis

To implement the algorithm suggested in the above example, the following steps are necessary.

1. In Step 1, the MSD is separated from the dividend and divided by the divisor. In case of the binary number, the MSB can be isolated by rotating bit D_7 into the Carry flag and from the Carry flag into bit D_0 of the accumulator (or of any other register). Now, the MSB can be divided by subtracting the divisor; in binary numbers, the quotient can be either 0 or 1. If the Carry flag is set after the subtraction, the divisor is larger than the MSB; thus, the quotient is 0 and the MSB should be retained as a remainder. If the divisor can be subtracted from the MSB, the quotient is 1 and the remainder should be passed on to the next operation.
2. In the next operation, bit D_6 of the dividend is combined with the remainder by rotating the bits, and the above process is repeated until the dividend is rotated left eight times.

11.43 Program

DIVIDE: ;This subroutine divides two 8-bit integers.
;Input: Dividend in register D and divisor in register E
;Output: Quotient in register L and remainder in register C
;Registers modified: B, C, and L
;Calls two subroutines: DIV8 and RESULT

```

LD B, 08H      ;Set up register B to count eight rotations
LD L, 0        ;Clear L to save quotient
LD C, L        ;Clear C to save partial remainder
NXTBIT: CALL DIV8    ;Call routine to divide two numbers
CALL RESULT    ;Call routine to save results
DEC B          ;Decrement bit-count
JR NZ NXTBIT  ;If all bits are not checked, get next bit
RET
```

DIV8: ;This subroutine gets one bit at a time from the dividend to form a
;partial dividend and subtracts the divisor from it and passes the
;status of CY flag and the remainder to the subroutine RESULT
;Input : Dividend in register D and divisor in register E
; : Partial remainder in register C
;Output: Remainder in accumulator and CY flag status
;Registers modified: A, C,

```

LD A, D        ;Get dividend
RLCA          ;Shift bit into CY
LD D, A        ;Save remaining dividend
LD A, C        ;Place partial remainder in A
RLA           ;Combine CY bit with remainder to form partial
               ;dividend
CP E          ;Check if divisor > partial dividend
RET C          ;Return if CY = 1, divisor > partial dividend
SUB E          ;Subtract divisor from partial dividend
RET
```

RESULT: ;This subroutine saves the remainder and adjusts the quotient accord-
;ing to the result of the previous routine.
;Input : Remainder in the accumulator and CY status
;Output: Quotient in register L and remainder in register C

```

LD C, A        ;Save partial remainder
CCF           ;Set CY to 1 or 0 as a quotient
LD A, L        ;Get previous quotient
RLA           ;Add quotient from CY flag
LD L, A        ;Save partial quotient
RET
```

11.44 Program Description

The subroutine DIVIDE initializes register B to count 8-bit rotation; for a 16-bit division, the counter would be set up to count 16. Register L is cleared to save the quotient and register C to save the partial dividend. This routine calls the subroutines DIV8 and RESULT eight times until the counter = 0.

The primary function of the subroutine DIV8 is to take bit D₇ from the dividend (register D) and place the bit as D₀ into register C using the accumulator. This is similar to the procedure shown in the example—taking one digit from the dividend and combining it with the remainder from the previous step. The dividend from register D is copied into A and D₇ is shifted into CY flag. Then the accumulator contents are replaced by the partial remainder from register C, and the CY flag is shifted into the accumulator to form the partial dividend. The divisor (register E) is compared with the new partial dividend. If the divisor is larger than the dividend, the CY flag is set and the program returns; otherwise, the divisor is subtracted from the partial dividend before returning.

In the subroutine RESULT, the remainder is saved in register C, and the CY flag is complemented. When the divisor is larger than the partial dividend, the CY flag is set, but the quotient is zero. On the other hand, when the divisor can be subtracted from the partial dividend, the Carry is reset, but the quotient is 1. Thus the CY flag needs to be complemented before placing it as D₀ into register L.

11.5

ILLUSTRATIVE PROGRAM: BCD TO BINARY CONVERSION

In most microprocessor-based products, data are entered and recorded in decimal numbers. For example, in an instrumentation laboratory, readings such as voltage and current are maintained in decimal numbers, and data entry may be done 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 as a 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, the 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, to convert 72_{BCD} into its binary equivalent requires multiplying the second digit by 10, and adding the first digit.

Converting a two-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: BCD₁ and BCD₂.

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.

Convert 72_{BCD} into its binary equivalent.

Example
11.5

$$72_{10} = 0111\ 0010_{BCD}$$

Solution

Step 1: $0111\ 0010 \rightarrow 0000\ 0010$ Unpacked BCD₁
 $\rightarrow 0000\ 0111$ Unpacked BCD₂

Step 2: Multiply BCD₂ by 10 (7×10)

Step 3: Add BCD₁ to the answer in Step 2.

The multiplication of BCD₂ by 10 can be performed by various methods. One method is multiplication with repeated addition: add 7 ten times. This technique is illustrated in the next program.

11.51 Problem Statement

A two-digit BCD number between 0 and 99 is stored in an R/W memory location called the Input Buffer (INBUF). Write a program that utilizes an unpacking subroutine (UNPACK) 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).

11.52 Program

START:	LD SP, STACK	;Initialize stack pointer
	LD HL, INBUF	;Point HL index to the Input Buffer memory
	LD BC, OUTBUF	;Point BC index to the Output Buffer memory
	LD A, (HL)	;where binary number will be stored
	LD HL, BUFF1	;Get BCD number
	CALL UNPACK	;Set up HL as memory pointer for BUFF1
	CALL BCDBIN	;Call routine to unpack BCD number
	LD (BC), A	;Call BCD to binary conversion routine
	HALT	;Store binary number in the Output Buffer
		;End of program
UNPACK:	;This subroutine unpacks the BCD number from the accumulator	
	;and stores two unpacked BCD digits in output buffer	
	;memory BUFF1 and BUFF1 + 1	
	;Input: Packed BCD number in the accumulator, and storage address	

;for unpacked BCD in HL
 ;Output: None, but stores unpacked BCD in BUFF1 and BUFF1 + 1
 ;Registers modified: Accumulator and HL

```

PUSH BC
LD B, A           ;Save BCD number temporarily
AND 0FH           ;Keep low-order 4 bits BCD1
LD (HL), A        ;Store BCD1 in BUFF1
INC HL            ;Point to BUFF1 + 1
SRL B             ;Shift BCD2 to right to get it into bits D3-D0
SRL B
SRL B
SRL B
LD (HL), B        ;Store BCD2 in BUFF1 + 1
RET
POP BC

```

BCDBIN: ;This subroutine converts an unpacked BCD number into
 ;its binary equivalent.
 ;Input: Two unpacked BCD numbers in BUFF1 and BUFF1 + 1,
 ; low-order BCD1 in BUFF1 and BCD2 in BUFF1 + 1
 ;Output: A binary number in the accumulator.
 ;Registers modified: A, E, H, and L

```

LD HL, (BUFF1)    ;Get BCD1 in L and BCD2 in H registers
XOR A             ;Clear accumulator
LD E, 10          ;Set register E as multiplier of ten
SUM: ADD A, E      ;Add ten to accumulator and continue adding
               ;as many times as the value of BCD2
DEC H             ;Reduce BCD2 by one
JR NZ SUM         ;Is multiplication complete? If not repeat
ADD A, L          ;Add BCD1
RET

```

11.53 Program Description

1. In modular programming, the main program is concerned primarily with initializations and passing parameters on to subroutines. In this illustration, the main program initializes the stack pointer and two memory indexes. It brings the BCD number into the accumulator, initializes location BUFF1, and passes these parameters to the subroutine. It calls two subroutines: UNPACK and BCDBIN.
2. After the return from the subroutines, the main program stores the binary equivalent in the Output Buffer memory.
3. The conversion from BCD to binary involves multiplying BCD2 by its positional weighting factor (10) and adding BCD1 as explained in the example.

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 Assignment 19 at the end of this chapter. We could have used also the MLTPLY (Section 11.33) subroutine with a few modifications.

ILLUSTRATIVE PROGRAM: BINARY TO BCD CONVERSION

11.6

In most microprocessor-based products, numbers are displayed in decimal. However, since data processing inside the microprocessor is performed in binary, it is necessary to convert the binary results into their equivalent BCD numbers just before the display.

The conversion of Binary to BCD is performed by dividing the number by the powers of ten; the division can be performed either by the subtraction method or the algorithm shown in the subroutine DIVIDE (Section 11.43).

For example, assume the binary number is

$$1\ 1\ 1\ 1\ 1\ 1\ 1_2 (FF_{16}) = 255_{10}.$$

To represent this number in BCD requires 12 bits or three BCD digits, labelled here as BCD_3 (MSB), BCD_2 , and BCD_1 (LSB).

$$\begin{array}{cccccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ & BCD_3 & & & BCD_2 & & & BCD_1 & & \end{array}$$

The conversion can be performed as follows:

Step 1: If the number is less than 100, go to Step 2; otherwise, subtract 100 repeatedly until the remainder is less than 100. The quotient is the most significant BCD digit BCD_3 .

Example Quotient

$$\begin{array}{rcl} 255 & & \\ -100 = 155 & 1 & \\ -100 = 55 & 1 & \\ BCD_3 = 2 & & \end{array}$$

Step 2: If the number is less than 10, go to Step 3; otherwise subtract 10 repeatedly until the remainder is less than 10. The quotient is BCD_2 .

$$\begin{array}{rcl} 55 & & \\ -10 = 45 & 1 & \\ -10 = 35 & 1 & \\ -10 = 25 & 1 & \\ -10 = 15 & 1 & \\ -10 = 05 & 1 & \\ BCD_2 = 5 & & \end{array}$$

Step 3: The remainder from step 2 is BCD_1 .

$$BCD_1 = 5$$

These steps can be converted into a program as illustrated below.

11.61 Problem Statement

An 8-bit binary number is stored in the memory location BINBYT. Convert the number into BCD, and store each BCD as unpacked BCD digits in an output buffer.

11.62 Problem Analysis

The problem can be divided into three tasks.

1. The first task is to get the byte from memory. This can be part of the main program.
2. After getting the byte, it has to be divided by 100 and 10. These divisors need to be supplied.
3. Convert the binary number into BCD numbers and store them in the output buffer.

11.63 Program

This program converts an 8-bit binary number into a BCD number, so 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: LD SP, STACK      ; Initialize stack pointer
        LD HL, BINBYT    ; Point HL index to where binary number is stored.
        LD A, (HL)        ; Transfer byte
        CALL BINBCD      ; Call conversion subroutine
        HALT

BINBCD: ;This subroutine supplies the powers of ten in register B
        ;and calls the BCD conversion routine.
        ;Input: Binary number in the accumulator.
        ;Output: Powers of ten and stores BCD1 in the first output buffer.
        ;Calls BCD routine and modifies register B.
        LD HL, OUTBUF    ; Point HL index to output-buffer memory
        LD B, 100         ; Load 100 into register B (in manual
                           ; assembly 100 should be converted to 64H)
        CALL BCD         ; Call conversion
        LD B, 10          ; Load 10 into register B
        CALL BCD
        LD (HL), A        ; Store BCD1
        RET

BCD:  ;This subroutine converts a binary number into
      ;BCD and stores BCD2 and BCD3 in the output buffer.
```

```

;Input:Binary number in accumulator and powers of ten in B.
;Output: None, but stores BCD2 and BCD3 in output buffer.
;Modifies accumulator content.

LD (HL), 0FFH ;Load buffer with (zero minus one)

STORE: INC (HL)      ;Clear buffer and increment for each subtraction
       SUB B          ;Subtract power of ten from binary number
       JR NC, STORE   ;Is number larger than power of ten?
                      ;If yes, go back and add 1 to buffer
       ADD A, B        ; If no, add power of ten to get back remainder
       INC HL          ; Go to next buffer location
       RET

```

11.64 Program Description

This program illustrates the concepts of the nested subroutine and multiple call subroutine. The main program calls the subroutine BINBCD; in turn, the BINBCD calls the BCD subroutine twice.

1. The main program transfers the byte to be converted to the accumulator, and calls the BINBCD subroutine.
2. The subroutine BINBCD supplies the powers of ten by loading register B and the address of the first output buffer memory location, and calls the conversion routine BCD.
3. In the BCD conversion routine, the output buffer memory is used as a register whose contents are incremented in each subtraction loop. This step can also be achieved by using a register in the microprocessor. The BCD routine is called twice, once after loading register B with 100, and again after loading register B with 10.
4. During the first Call of BCD, the subroutine clears the output buffer, stores BCD₃, and points the HL registers to the next output buffer location. The instruction ADD A, B is necessary to restore the remainder because one extra subtraction is performed to check the borrow.
5. During the second Call of BCD, the subroutine again clears the output buffer, stores BCD₂, and points to the next buffer location. BCD₁ is already in the accumulator after the ADD instruction, which is stored in the third output buffer by the instruction LD (HL), A in the BINBCD subroutine.

ILLUSTRATIVE PROGRAM: ASCII TO BINARY CODE CONVERSION

11.7

A computer is a binary machine; it understands and communicates in binary language. However, human beings communicate using alphanumeric symbols (letters and numbers).

Therefore, we need to translate between alphanumeric symbols and the binary language; ASCII (The American Standard Code for Information Interchange) is a commonly used code for such translation. It is a seven-bit code with 128 (2^7) combinations. In the 8-bit word format, the ASCII code ranges from 00_H to $7F_H$, bit D₇ being 0. Each combination is assigned to a letter, a decimal number, or a machine command (see Appendix C). For example, hexadecimals 30_H to 39_H represent numerals 0 to 9, and 41_H to $5A_H$ represent capital letters from A to Z.

The ASCII keyboard is a standard input device for most microcomputers. 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 code $0\ 0\ 1\ 1\ 1\ 0\ 0\ 1$ (39_H), which must be converted to the binary equivalent of 09 (0 0 0 0 1 0 0 1) for arithmetic operations. Similarly, to display digit 9 at the video terminal, the microprocessor must send out the ASCII code 39_H . In some systems, bit D₇ of the ASCII code is used for the parity check. The parity check conveys the information whether the number of 1s in a transmitted ASCII byte is odd or even (see Chapter 15 for details). For example, in a system with the odd parity check, ASCII 9 will be transmitted with D₇ as 1 to keep the number of 1s odd in the byte; ASCII 9 will be $B9_H$. At this point, we do not need to know the details of parity check except that bit D₇ should be masked to translate from ASCII to binary code.

11.71 Problem Statement

Write a subroutine to convert an ASCII Hex digit (0 to F) into its binary equivalent. A calling program places the ASCII Hex digit including the parity bit into the accumulator.

11.72 Problem Analysis

The ASCII codes for digits 0 to 9 range from 30_H to 39_H , and for digits A to F, they range from 41_H to 46_H ; there is a break in the range. Therefore, to set up a conversion routine, we need to check two ranges. If the digit is between 0 and 9, it can be obtained by subtracting 30_H from the ASCII code. If it is between A to F, we need to subtract an additional 07_H from the remainder because there are seven ASCII codes between the code of 9 (39_H) and code of A (41_H).

11.73 Program

```
ASCBIN: ;This subroutine takes an ASCII Hex digit, strips the parity bit, and
        ; converts it into its binary equivalent. In the comment section, the routine
        ; is explained assuming ASCII F (46H) as an illustration.
        ;Input: ASCII Hex digit (with parity bit) in the accumulator
        ;Output: Binary equivalent in the accumulator
        ;Modifies the contents of the accumulator
```

		Example
;		A 0010 0110 46H
;		<u>0111 1111</u> 7FH
AND 7FH	;Mask parity bit	4 6 H
SUB 30H	;Subtract 0 bias from the digit	<u>-3 0 H</u>
CP 10	;Is the digit between 0 and 9?	1 6 H
RET C	;If yes, conversion done	
SUB 7	;If not, subtract 7 to find digit ; between A and F	1 6 H
RET		<u>0 7 H</u>
		0 F H

11.74 Program Description

This is an illustration of the multiple ending subroutine. If the digit is between 0 and 9, its comparison with 10 results in a return on the Carry flag. Otherwise, the subroutine returns after subtracting an additional 7. However, this routine does not check whether the ASCII character is beyond the range of Hex digits.

ILLUSTRATIVE PROGRAM: BINARY TO ASCII CODE CONVERSION

11.8

The binary to ASCII code conversion is necessary to display text or numbers at an ASCII terminal (or print at a printer). For example, to display digit 9 at the terminal, the micro-processor should send out 39_{16} . The following subroutine illustrates binary to ASCII conversion.

11.81 Problem Statement

Write a subroutine to convert a byte in the accumulator into two ASCII characters and store them in output buffer OUTBUF and OUTBUF + 1.

11.82 Problem Analysis

A byte in the accumulator is equivalent to 2 Hex digits. Therefore, the byte should be unpacked. The conversion process is opposite to that of the previous subroutine ASCBIN. If the digit is between 0 to 9, it is converted by adding 30_{16} , and if the digit is between A to F, an additional 07_{16} must be added to the digit.

11.83 Program

BINASC: ;This subroutine converts the byte in the accumulator into two ASCII
;characters and saves them in memory OUTBUF and OUTBUF + 1
;Input: Byte in the accumulator and the memory pointer for OUTBUF
;in BC

```

;Output: None; two ASCII characters are stored in OUTBUF
;Registers modified: A, B, C, H, and L
;Calls two subroutines: UNPACK and ASCII
CALL UNPACK      ;Unpack the byte from the accumulator and
                  ;place nibbles as low-order 4-bit in BUFF1 and
                  ;BUFF1 + 1
LD HL, (BUFF1)  ;Place unpacked nibbles into HL register
LD A, L          ;Place digit from BUFF1 into A for conversion
CALL ASCII       ;Convert into ASCII character
LD (BC), A        ;Store first ASCII in OUTBUF
INC BC           ;Memory pointer to OUTBUF + 1
LD A, H          ;Place digit from BUFF1 + 1 into A for conver-
                  ;sion
CALL ASCII       ;Convert second digit into ASCII character
LD (BC), A        ;Place second ASCII into OUTBUF + 1
HALT             ;End

UNPACK:          ;This subroutine is written in Section 11.53.

ASCII:           ;This subroutine converts low-order 4-bit into ASCII Hex code
;Input : Binary digit from 0 to F as low-order 4-bit in the accumulator
;Output: ASCII Hex code in the accumulator
;Modifies accumulator contents
CP 10            ;Is digit less than 10?
JR C, BASE        ;If yes, go to add ASCII base of 30H
ADD A, 07H         ;Add 7H to get code for digits between A and F
BASE:            ADD A, 30H        ;Add ASCII base 30H
                  RET

```

11.9 SOFTWARE DESIGN

In previous sections of this chapter, we illustrated various subroutines related to 16-bit arithmetic operations and code conversions. These subroutines are written as independent modules dealing with a simple task. We can now combine these independent modules to design a simple project. The following project can be viewed as part of a communication process between the microcomputer and its terminal. The project is concerned with how to process ASCII characters after they have been received, how to form binary numbers for arithmetic operations, and how to convert any arithmetic results into ASCII characters for display.

11.91 Project Statement

Four ASCII characters are stored sequentially in the input buffer INBUF. The first two characters represent an 8-bit multiplicand and the remaining two represent an 8-bit multiplier. Each number is stored low-order digit first, followed by the high-order digit.

Convert the ASCII characters into binary digits, multiply the numbers, and store the result in the output buffer OUTBUF as ASCII characters.

11.92 Project Analysis

This is a simple software design project, and it can be divided into various segments. To clarify the analysis, the steps are illustrated with the example of the following four ASCII characters: 32_H , 41_H , 46_H and 35_H (Figure 11.1).

1. First, we need to convert ASCII characters into their binary equivalents. Four ASCII characters will have four binary digits; thus, they can be placed as unpacked binary digits (02, 0A, 0F, 05) back into INBUF. (See Appendix C for ASCII table to obtain Hex equivalent for ASCII characters.)
2. The first two represent a multiplicand ($A2_H$) and the remaining two represent a multiplier ($5F_H$). These four digits need to be packed as two binary numbers (Figure 11.1(b)).
3. When these two 8-bit numbers are multiplied, the result can be as large as a 16-bit number (or four Hex digits). In our example, the result is $3C1E_H$.
4. To convert the result into ASCII characters, all four digits should be converted into unpacked digits as 0E, 01, 0C, and 03 (Figure 11.1(c)).
5. Now the unpacked digits can be converted into ASCII characters as 45_H , 31_H , 43_H , and 33_H and stored sequentially in the output buffer OUTBUF (Figure 11.1(d)).

11.93 Program

START:	LD SP, STACK	;Initialize the stack
	LD HL, INBUF	;Set up HL as memory pointer to ASCII characters
	LD B, 04H	;Set up register B to count ASCII characters
CHAR:	LD A, (HL)	;Get ASCII character
	CALL ASCBIN	;Convert into binary
	LD (HL), A	;Place unpacked binary digit into INBUF
	INC HL	;Next buffer memory location
	DEC B	;One conversion complete
	JR NZ, CHAR	;If all characters are not yet converted, get next one
	LD HL, INBUF	;Set up HL as memory pointer for unpacked digits
	CALL PACK	;Pack digits of the multiplicand
	LD E, A	;Place the multiplicand in register E
	CALL PACK	;Pack digits for the multiplier
	LD D, A	;Place the multiplier in register D
	CALL MLTPLY	;Multiply two numbers and place result in HL
	EX DE, HL	;Save the result in DE

	LD HL, OUTBUF	;Set up HL as memory pointer for OUTBUF
	LD A, E	;Get low-order byte of the result
	CALL UNPACK	;Unpack the low-order byte and store in ;OUTBUF
	LD HL, OUTBUF + 2	;Update memory pointer
	LD A, D	;Get high-order byte of the result
	CALL UNPACK	;Unpack high-order byte and store in ;OUTBUF
	LD HL, OUTBUF	;Set up HL as memory pointer for OUTBUF
	LD B, 04H	;Set up register B to count four digits
DIGIT:	LD A, (HL)	;Get binary digit
	CALL ASCII	;Convert binary digit into ASCII character
	LD (HL), A	;Save ASCII character in OUTBUF
	INC HL	;Next buffer memory location
	DEC B	;One conversion complete
	JR NZ, DIGIT	;If all digits are not yet converted, get next ;digit
	HALT	
PACK:	;This subroutine takes two unpacked digits from memory and packs ;them into an 8-bit number in the accumulator.	
	;Input: Memory address in HL	
	;Output: Packed number in the accumulator	
	;Modifies registers: A, B, H, and L	
	LD B, (HL)	;Get low-order unpacked digit
	INC HL	;Point to the next digit
	LD A, (HL)	;Get high-order unpacked digit
	SLA A	;Place digit into bit positions D ₇ –D ₄
	SLA A	;and clear D ₃ –D ₀ .
	SLA A	
	SLA A	
	OR B	;Pack both digits
	INC HL	;Point to the next memory location
	RET	

11.94 Program Description and Debugging

This program is made up of several subroutines written previously, and the comments explain the functions of each subroutine. This is a demonstration of how a problem can be divided into small modules and how these modules can be written as subroutines and combined into a program.

The program begins by initializing a memory pointer for ASCII characters stored in INBUF and the counter to get these characters into the microprocessor. The program segment starting with the label CHAR gets these characters and converts them into

unpacked digits using the subroutine ASCBIN. Figure 11.1(a) shows the process with the four specific ASCII characters as examples. The subroutine PACK converts them into binary numbers—a multiplicand and a multiplier—and stores them in registers E and D, respectively. The next subroutine MLTPLY multiplies these numbers and places the product $3C1EH$ into register HL (Figure 11.1(c)). The product is again unpacked and stored as four unpacked digits in memory OUTBUF. Finally, these digits are converted into ASCII characters by the subroutine ASCII (Figure 11.1(d)). We could have used the subroutine BINASC, with some modifications, instead of the last two subroutines.

The subroutines used in this project are taken from the previous programs, except PACK, which is included at the end of the program. In troubleshooting this program, you have to be careful in checking the parameters that are passed from one module to another. However, this type of program is easy to debug. The programmer can set up breakpoints at the end of a module and check register contents and parameters being passed. For example, a breakpoint can be set up just before the subroutine PACK is called, and the contents of memory INBUF can be verified. If unexpected results are found, you can examine the initialization instructions or the subroutine ASCBIN. On the other hand, if the expected results are found, you can proceed to check the output after the subroutine PACK. The keys to troubleshooting software are modularity and knowledge of the expected outputs at critical junctures.

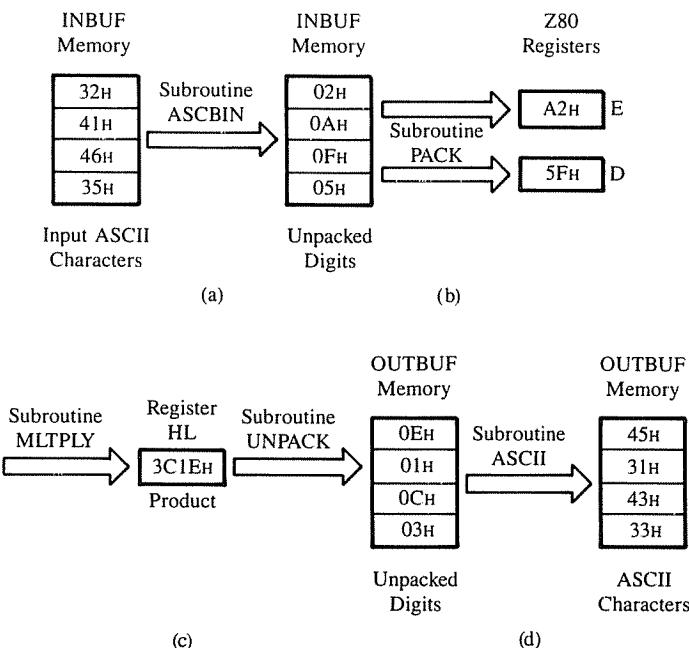


FIGURE 11.1
Memory and Register Contents After Execution of Subroutines

SUMMARY

This chapter illustrated subroutines dealing with arithmetic operations such as multiprecision addition, multiplication, and division, and code conversions for BCD and ASCII. The programs were written as independent modules, and the design of a simple software project was illustrated using these modules. The project demonstrated how to break down a given problem into small manageable modules, and how these modules can be written as subroutines and combined in a program to accomplish a given task.

However, single-board microcomputer systems are unsuitable for writing, coding, and debugging programs larger than fifty instructions. To write a large program, it is necessary to have access to an assembler and a disk-based system, as discussed in Chapter 7.

ASSIGNMENTS

Section 11.1

1. Register BC contains the 16-bit number $72F2_H$. Add $F5_H$ to the number and save the result in BC.
2. Two 16-bit numbers ($82F7_H$ and $24A2_H$) are stored in memory locations 2050_H to 2053_H , low-order byte first followed by the high-order byte. For example, the location 2050_H holds $F7_H$ and 2051_H holds 82_H . Add the numbers. If the sum is larger than 16-bit, call the ERROR routine; otherwise, store the sum in memory locations 2060_H and 2061_H .
3. Register BC contains $A7F2_H$ and register DE contains $5F18_H$. Add the numbers. If the sum is larger than 16-bit, call the OVRLOD routine; otherwise, save the result in register BC.
4. Register BC contains $87A9_H$. Subtract the byte $F8_H$ and save the result in register BC.
5. Register BC holds $F538_H$ and register DE holds $A279_H$. Subtract the contents of DE from the contents of BC and save the result in BC.
6. Register D holds the number $C4_H$. Shift the entire number to the left by four positions and clear bits D_3 – D_0 (the result should be 40_H).
7. Write instructions to get the address from the stack pointer and save it on the stack.
8. Assuming the HL register holds 2088_H , write a 1-byte instruction to transfer the program execution to 2088_H .

Section 11.2

9. Two 24-bit numbers, each occupying three memory locations, are stored in addresses starting with 2050_H and 2060_H . The numbers are stored with high-order

byte first; locations 2050_H and 2060_H hold the high-order byte of each number. Write a subroutine to add the numbers and store the result in memory locations starting with the low-order byte at 2070_H . The calling program should pass the memory addresses to the subroutine.

10. A string of 16-bit numbers is stored in memory locations starting at BUF1; the numbers are stored with low-order byte first. Write a subroutine to add the string and save the result in the output buffer OUTBUF; the result is limited to 24 bits. The calling program should supply the memory addresses and the length of the string.
11. Modify the program in 10 to increase the limit of the result to 32 bits.
12. In 10, save the contents of the stack pointer from the main program, point the stack pointer to the location BUF1, and transfer the readings to registers by using the POP instruction. Add the readings as in 10; however, the original contents of the stack pointer should be retrieved after the addition is completed.
13. 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 binary code of the key pressed into 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.
Hints: See Example 11.4 to shift the four low-order bits in a 16-bit register.

Sections 11.3 and 11.4

14. Rewrite the MLTPLY subroutine to multiply two 16-bit unsigned numbers; the multiplier is given in register DE and the multiplicand in register HL.
15. Rewrite the MLTPLY subroutine using the technique of successive addition for two 8-bit unsigned numbers.
16. Write a subroutine to divide two unsigned 8-bit numbers using the technique of successive subtraction. The calling program passes the dividend in register D and the divisor in register E.
17. Write a subroutine to divide two unsigned 16-bit numbers using the technique shown in Section 11.4. The calling program passes the dividend in register HL and the divisor in register DE.

Section 11.5

18. Modify the Illustrative Program: BCD to Binary Conversion as follows. The number of BCD digits to be converted is specified by the main program in register D and passed on as a parameter to the subroutine.
19. Rewrite the multiplication section of the BCDBIN routine using the RLCA (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).

20. An 8-bit packed BCD is in the accumulator. Save BCD_1 in one of the registers and delete BCD_1 from the accumulator, leaving BCD_2 in high-order positions D_7-D_4 . Clear the CY flag and shift BCD_2 to the right by one position. Explain that by rotating BCD_2 to the right once from the high-order position is equivalent to multiplying it by eight from the low-order position.

Section 11.6

21. Assume the STACK is defined as $20B8_H$ in the illustrative program. Specify the stack addresses and their (symbolic) contents when the BCD subroutine is called the second time.
22. 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 subroutine BCD to accommodate the changes in the main program, and eliminate the subroutine BINBCD.
23. Rewrite the program to convert a given number of binary data bytes into their BCD equivalent, 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 into those locations.
24. 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 FF_H .

Sections 11.7, 11.8 and 11.9

25. Rewrite the subroutine PACK using an appropriate Rotate instruction.
26. A set of ASCII Hex digits is stored in the input buffer. Write a program to convert these numbers into binary. Add these numbers in binary, and store the result in the output buffer.
27. Extend the program in 26 to convert the result from binary to ASCII Hex code.

CHAPTER 12:

Interrupts

CHAPTER 13:

Programmable Interface Devices

CHAPTER 14:

Programmable Timers and Counters

CHAPTER 15:

Serial I/O and Data Communication

CHAPTER 16:

Advanced Topics in Memory Design and DMA

Concepts

CHAPTER 17:

Designing Microprocessor-Based Products

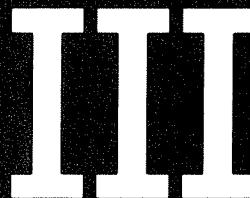
CHAPTER 18:

Trends in Microprocessor Technology

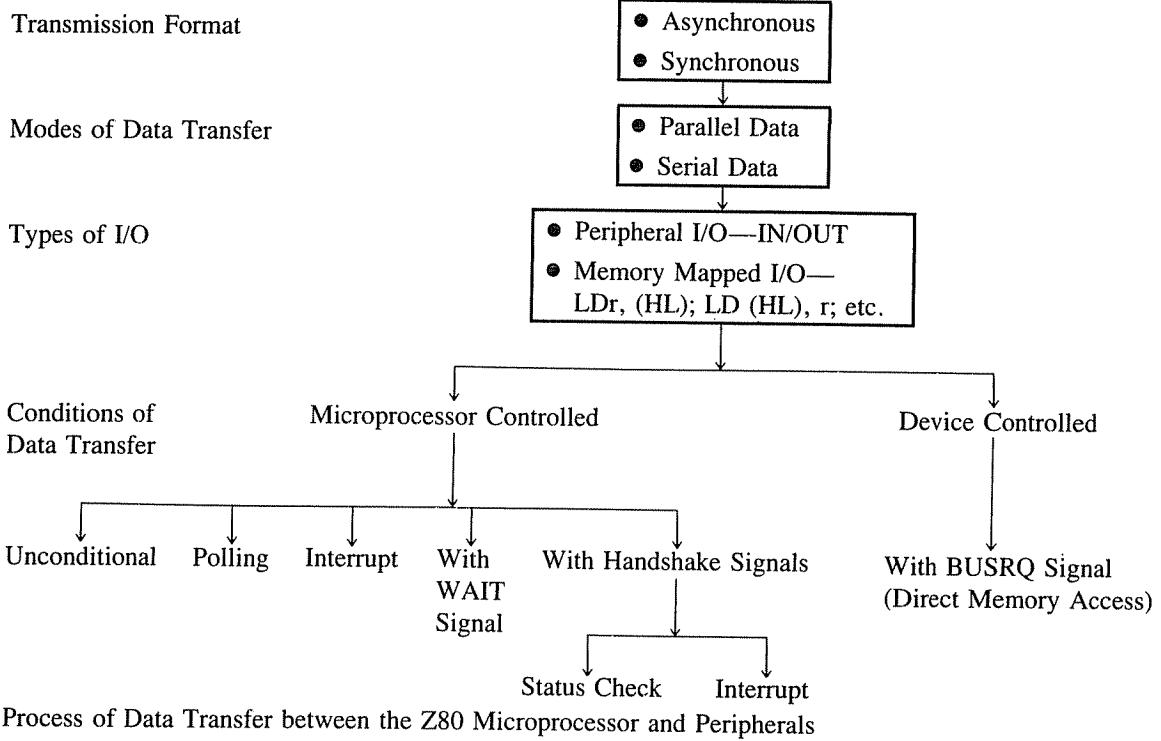
Part III of this book is concerned with the interfacing of peripherals using programmable I/O devices and design processes of microcomputer-based systems. The primary objectives of Part III are

1. To examine the concepts and processes of various data transfers between the microprocessor and peripherals.
2. To illustrate applications of programmable I/O interface devices.
3. To synthesize the concepts of microprocessor architecture, software, and interfacing by designing a simple microprocessor-based system.

The primary function of the microprocessor (MPU) is to accept data from such input devices as keyboards and A/D converters, read instructions from memory, process data according to the instructions, and send the results to such output devices as LEDs, printers, and video monitors. These input and output devices are called either **peripherals** or **I/Os**; memory can be viewed as a special type of I/O. The designing of logic circuits (hardware) and the writing of instructions (software) to enable the microprocessor to



Interfacing Peripherals, Programmable I/O Devices, Applications, and Design



communicate with these peripherals is called **interfacing**, and the logic circuits are called **I/O ports** or **interfacing devices**.

The microprocessor (MPU) communicates with the peripherals in either the **asynchronous** or **synchronous** format. Similarly, it can transfer data in one of two modes: **parallel I/O** or **serial I/O**. The Z80 identifies peripherals either as **memory-mapped I/O** or **peripheral I/O** based on their interfacing logic circuits (see Chapter 5). Data transfer between the microprocessor and peripherals can take place under various conditions, as shown in the chart. The modes, techniques, instructions, and conditions of data transfer are briefly described in the following paragraphs.

FORMATS OF DATA TRANSFER: SYNCHRONOUS AND ASYNCHRONOUS

Synchronous means at the same time; 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 between the microprocessor and 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 the same time. In the Z80, an 8-bit word is transferred simultaneously over the eight data lines as illustrated in Chapter 5. The peripherals 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 such peripherals as a teletype (TTY), CRT terminals, printers, and cassette tapes.

TYPES OF I/O: PERIPHERAL AND MEMORY-MAPPED

In Z80-based systems, I/O devices can be classified in two categories: peripheral I/Os or memory-mapped I/Os. In peripheral I/O, a peripheral is identified with an 8-bit address, and I/O related control signals are used for data transfer. 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 control signals.

CONDITIONS OF DATA TRANSFER

The process of data transfer between the microprocessor and peripherals is controlled either by the microprocessor or by the peripherals. 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 to 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. Micropro-

cessor-controlled data transfer can take place under five different conditions: (1) unconditional, (2) status check (also known as polling), (3) interrupt, (4) with WAIT signal, and (5) 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 Status Check (Polling) In this form of data transfer, the microprocessor is kept in a loop to check the status of a peripheral; this is also called polling. When the status condition is satisfied, data transfer is implemented. 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 form of data transfer, 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. The advantage of the interrupt technique is that the processor is free to perform other tasks rather than waiting in a status check or polling loop.

Data Transfer with WAIT Signal When peripheral response time is slower than the execution time of the microprocessor, the WAIT signal can be used to add T-states, thus extending the execution time. This technique provides sufficient time for the peripheral to complete the data transfer and is commonly used in a system with slow memory chips.

Data Transfer with Handshake Signals In this form of 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 these 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 microprocessor. 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 and is functionally similar to the polling method.

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 is peripheral-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 BUSRQ (Bus Request) signal to the microprocessor; the microprocessor acknowledges the request and releases its data bus, address bus, and control signals to the DMA controller; and data are transferred at high speed without the intervention of the microprocessor.

CHAPTER TOPICS

Chapter 12 is concerned with the Z80 interrupt I/O process, whereby an external peripheral can interrupt the processor and indicate its readiness for data

communication. The chapter discusses various modes of the Z80 interrupt and illustrates them with applications.

Chapter 13 deals with the programmable interface devices: the Z80 PIO and the Intel 8255A. These devices can be set up to perform various I/O tasks by instructions written in their control registers; they are thus called programmable devices. The chapter explains the basic concepts underlying the devices and illustrates various operational modes of these devices with examples.

Chapter 14 describes two programmable timers: the Z80 CTC and the Intel 8253. While time delays and counters were designed using software instructions in earlier chapters, this chapter illustrates the hardware approach.

Chapter 15 focuses on serial data communication, whereby data bits are transferred one bit at a time over one line. First, the chapter discusses the basic concepts in serial I/O and the software approach to serial data transfer. Then it illustrates how the concepts can be implemented using programmable serial interface devices such as the Z80 SIO and the Intel 8251A.

Chapter 16 is concerned with advanced topics in memory interfacing and concepts in the direct memory access (DMA). The topics include the need for Wait states, interfacing of dynamic memory, and data transfer using DMA techniques.

Chapter 17 discusses the design processes in a microprocessor-based product. The primary objective of this chapter is to synthesize the various concepts, both hardware and software, discussed in all the previous chapters; it includes the design of a single-board microcomputer.

Chapter 18 reviews various 8-bit, 16-bit, and 32-bit microprocessors and single-chip controllers and suggests trends in microprocessor technology.

PREREQUISITES

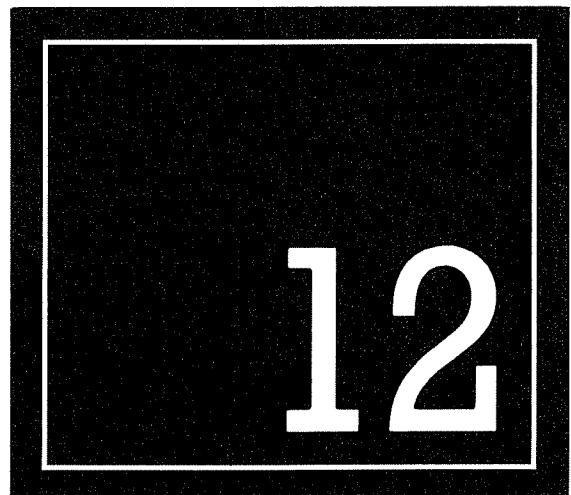
- Basic concepts of microprocessor architecture, memory, and I/Os (Part I).
- Familiarity with the Z80 instruction set and programming techniques (Part II).

Interrupts

In the introduction to Part III, we classified the processes of data transfer between the microprocessor and peripherals into five categories: unconditional, polling, interrupt, using Wait states and handshake signals. In this chapter, we focus on the interrupt process. 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 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**. A maskable interrupt request can be ignored or delayed by the microprocessor if it is performing some critical task; however, it has to respond to a nonmaskable request immediately. The maskable interrupt is somewhat like a telephone that can be kept off the hook if one is not interested in receiving any calls. The nonmaskable interrupt is like a smoke detector requiring immediate attention if set off.

The interrupt process allows the microprocessor to respond to these external requests for its attention or service on a demand basis and leaves the microprocessor free to perform other tasks. On the other hand, in the program controlled (or polled)



I/O, the microprocessor remains in a loop, continuously checking the I/O device and doing nothing else, until the device is ready for data transfer.

This chapter describes the basic concepts in the interrupt process and provides an overview of the Z80 nonmaskable interrupt and three modes of the maskable interrupt: Modes 0, 1, and 2. The interrupts are illustrated with examples and industrial applications, such as the interfacing of an A/D data converter. Finally, it includes discussion of how multiple interrupts are implemented with one interrupt line and how priorities are determined.

OBJECTIVES

- Explain an interrupt process and the difference between a nonmaskable and a maskable interrupt.
- List the modes of the maskable interrupt and differences among them.
- Explain the instructions EI, DI, and RST, and their functions in the Z80 interrupt process.
- List the eight steps to initiate and implement an interrupt in Z80.
- Design and implement an interrupt with a given RST instruction in Mode 0.
- Design and implement an interrupt in Mode 2 for a given memory address as a restart location.
- Interface an external device such as an A/D converter with the interrupt I/O.
- Explain how to connect multiple-interrupting peripherals with the INT interrupt line and how to determine their priorities using logic circuits.
- Explain how to use a RST instruction to implement a software breakpoint.

12.1 BASIC CONCEPTS IN INTERRUPT I/O

The interrupt I/O is a communication process through which the MPU can be interrupted by using one of the external request signals. In Chapter 3, we discussed briefly five such request signals (pins): Reset, Interrupt (INT), Nonmaskable Interrupt (NMI), Wait (WAIT), and Bus Request (BUSRQ) in the context of the Z80 architecture, and they are shown in Figure 12.1(a). In this chapter, we focus primarily on two external request signals, INT and NMI (Figure 12.1(b)), and their functions in the I/O communication process.

The interrupt signal can be compared with a telephone in a office; a person in the office can continue to work until interrupted by a phone ring. After answering the phone, the person can go back to work. In a microcomputer system, an external device can interrupt the microprocessor by using the interrupt signal. For example, let us assume the microcomputer is executing a program and occasionally needs to read data from a data converter whenever a new reading is available. The data converter can be interfaced with

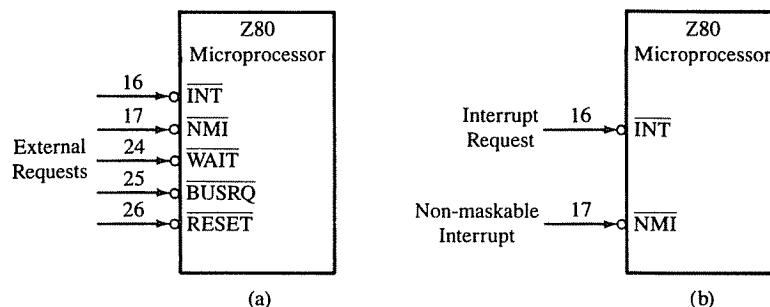


FIGURE 12.1

(a) Z80 External Request Signals (b) Z80 Interrupt Pins

the microprocessor using the interrupt line, so that it can interrupt the processor whenever a new data byte is available. The processor can then read the data byte and go back to executing the program. The interrupt I/O allows the processor to execute the program and also attend to various peripherals on a demand basis. Otherwise, in our example, the processor would be kept busy just reading data from the data converter input port.

What happens in the office of busy or high-powered executives when the phone rings? Generally, the secretary or the administrative assistant answers the phone, then transfers the call to the person you asked for; you may even have to go through one more secretary. When the Z80 microprocessor is interrupted, the program execution is transferred to specific locations in memory to get further instructions for what to do next; this group of instructions is called a service routine. In our example, the service routine may consist of reading a data byte. The Z80 microprocessor has various processes for transferring the program execution to these specific locations in memory similar to the various office protocols for answering telephones. These processes are called interrupt modes and are described in the next section.

12.11 Overview of Z80 Interrupts

The Z80 interrupts are divided into two groups: nonmaskable interrupt ($\overline{\text{NMI}}$, pin 17) and maskable interrupt (INT, pin 16). The maskable interrupt is the interrupt that can be masked, meaning it can be disabled or enabled. On the other hand, the nonmaskable interrupt cannot be disabled. The maskable interrupt has three different modes (Modes 0, 1, and 2), as shown in Figure 12.2, and they are explained in the following sections.

MASKABLE INTERRUPT

The Z80 maskable interrupt is controlled by the Interrupt Enable flip-flops (IFF1 and IFF2), which are internal to the processor. These flip-flops are set to logic 1 by using the

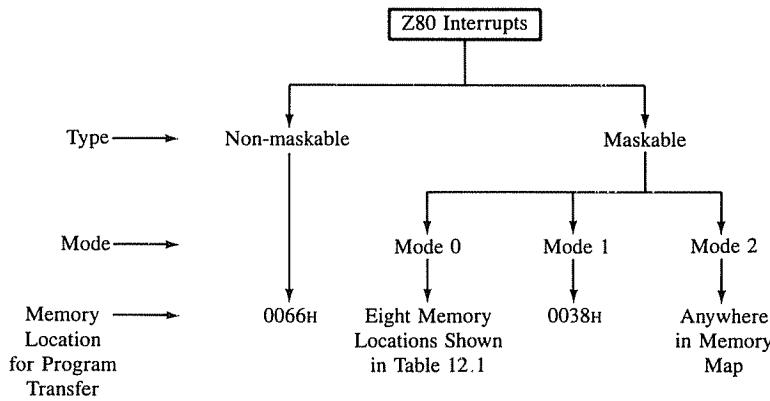


FIGURE 12.2
Z80 Interrupt Modes

software instruction EI to enable the interrupt process. The maskable interrupt can be disabled by using the instruction DI; this instruction resets the flip-flops IFF1 and IFF2.

Instruction: EI—Enable Interrupt

- This is a 1-byte instruction.
- The instruction sets the Interrupt Enable flip-flops IFF1 and IFF2 and enables the interrupt process.
- The interrupt process is disabled by the instruction DI, an interrupt acknowledgment by the Z80, or system Reset.

Instruction: DI—Disable Interrupt

- This is a 1-byte instruction.
- The instruction resets the Interrupt Enable flip-flops and disables the maskable interrupt.
- This instruction should be included in a program segment where an interrupt from an outside source cannot be tolerated.

The Z80 microprocessor can be interrupted from whatever it is doing if

- the flip-flops IFF1 and IFF2 are set (through software).
- the input to the interrupt signal $\overline{\text{INT}}$ (pin 16) is caused to go low by a signal from an external device or a peripheral until the microprocessor has time to sample the $\overline{\text{INT}}$. The $\overline{\text{INT}}$ signal is level sensitive, meaning it is not accepted (or stored) immediately on transition from high to low (see the details in Mode 0).

What happens after the Z80 is interrupted is dependent on the mode it has been programmed for by the programmer. The Z80 instruction set has three instructions to set the interrupt mode: IM 0 (Mode 0), IM 1 (Mode 1), and IM 2 (Mode 2).

Mode 0: The program execution can be transferred to one of the eight memory locations from 0000_H to 0038_H shown in Table 12.1 by using additional hardware (Refer to Section 12.2 for details).

Mode 1: The program execution is directly transferred to memory location 0038_H without any additional hardware.

Mode 2: The program execution can be transferred to any memory location by using external hardware and the address in the interrupt vector register I.

The next step is dependent on what is written at these memory locations; this is similar to what happens to your phone call when the secretary transfers it to the appropriate person. In the following sections, we discuss the hardware and software details of how to transfer the program execution to these memory locations and how to get back to the

TABLE 12.1
Restart Instructions

Mnemonics	Binary Code								Hex Code	Call Location (Hex)
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀		
RST 00H	1	1	0	0	0	1	1	1	C7	0000
RST 08H	1	1	0	0	1	1	1	1	CF	0008
RST 10H	1	1	0	1	0	1	1	1	D7	0010
RST 18H	1	1	0	1	1	1	1	1	DF	0018
RST 20H	1	1	1	0	0	1	1	1	E7	0020
RST 28H	1	1	1	0	1	1	1	1	EF	0028
RST 30H	1	1	1	1	0	1	1	1	F7	0030
RST 38H	1	1	1	1	1	1	1	1	FF	0038

program execution prior to the interrupt. Remember that the person you called has to get back to work after the phone call.

NONMASKABLE INTERRUPT

This interrupt is not controlled through the Interrupt Enable flip-flops. The instruction DI therefore has no effect on this interrupt, and the instruction EI is not necessary to enable it. This interrupt can be compared to the smoke detector in an office, rather than a telephone. When the smoke detector sets off the alarm, it has to be responded to immediately.

The NMI (pin 17) is an active low, edge-sensitive interrupt. When the NMI goes low, the Z80 completes the execution of the current instruction and transfers the execution to memory location 0066_H without any external hardware. The details of this interrupt process are discussed later in the chapter.

12.12 Restart (RST) Instructions and Mode 0

In the previous discussion, we mentioned that if the Z80 is programmed for Mode 0, the program execution can be transferred after an interrupt to one of the eight memory locations shown in Table 12.1. These memory locations are directly related to eight Restart (RST) instructions in the Z80 instruction set.

RESTART (RST) INSTRUCTIONS

These are 1-byte call instructions that transfer the program execution to a specific location on the page 00_H , as listed in Table 12.1. These instructions implicitly specify a 16-bit address in one opcode; this is called the Modified Page Zero Addressing mode. Let us recall, from Chapter 9, how the Call instructions are executed. When a Call instruction is executed, the Z80 first stores the address of the next instruction on the top of the stack, then transfers the program to the Call location. The RST instructions are executed similarly, as illustrated in Example 12.1.

**Example
12.1**

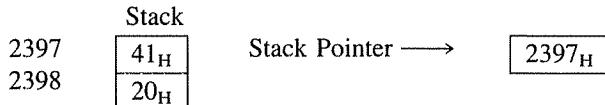
Specify the contents of the stack and the stack pointer after the execution of the instruction RST 30H. Where will the program execution be transferred?

Instructions

2000	LD	SP,	2399H
2003		↓	
		↓	
2040	RST	30H	
2041			

Solution

The stack pointer is initialized at 2399H. The RST 30H is a 1-byte call instruction to location 0030H. When the RST instruction is executed, the Z80 places the contents of the program counter 2041H (which is the address of the instruction following the RST instruction) on the stack, the stack pointer is decremented to 2397H, and the program execution is transferred to location 0030H. The contents of the stack and the stack pointer are as follows:



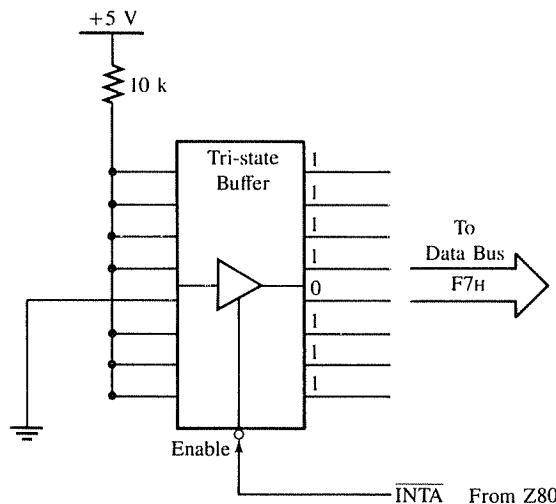
We can now use these RST instructions in conjunction with the interrupt Mode 0. We can insert one of the RST instructions into the microprocessor using external hardware. Figure 12.3 shows such a circuit. The input to the tri-state buffer is F7H, which is the code for the RST 30H instruction (see Table 12.1). When this buffer is enabled, the instruction RST 30H will be placed onto the data bus and brought into the microprocessor. When it is executed, the program will be transferred to the memory location 0030H. We now need to know how and when to enable the buffer. The buffer is enabled by the INTA signal, which is issued by the Z80 when it acknowledges an interrupt (this is explained in Section 12.14).

12.13 Interrupt Process in Mode 0

The interrupt in Mode 0 is compatible to the interrupt (INT) in the 8080 and (INTR) in the 8085 microprocessors. We selected Mode 0 to describe the interrupt process because it includes all the basic concepts in the interrupt I/O; other modes can be described as special cases of Mode 0.

One way to describe the Z80 interrupt process is with the analogy of the telephone in an office, this time with a blinking light instead of a ring. Assume that the office has one telephone serving eight engineers, and it is monitored by the secretary. The secretary is generally busy typing, and when the phone begins to blink, he or she stops typing to answer the phone. In order for the secretary to receive and respond to a telephone call, typically, the following activities take place:

FIGURE 12.3
A Circuit to Implement the
Instruction RST 30H



1. The telephone system is enabled, meaning that the receiver is on the hook.
2. The secretary glances at the light at certain intervals to check whether someone is calling.
3. When the light begins to blink, the secretary completes the sentence being typed, answers the phone, and waits for a response. Once the phone is picked up, the line is busy, and no more calls can be received on that line until the receiver is placed back on the hook.
4. The caller specifies the message; for our example, assume the caller is the manager of the group and wants to cancel the scheduled meeting with one of the engineers, Ms. Peterson. The secretary performs the following steps:
5. makes a pencil mark at the beginning of the next sentence on the typing draft as a reminder to start typing at that point later on.
6. places the receiver back on the hook.
7. informs Ms. Peterson about the cancellation of the meeting.
8. goes back to the pencil mark on the typing draft and starts typing again.

In some instances, steps 6 and 7 are interchanged, depending on the urgency of the request. If the request is critical and the secretary does not want to be interrupted again while attending to the request, step 7 will be performed first.

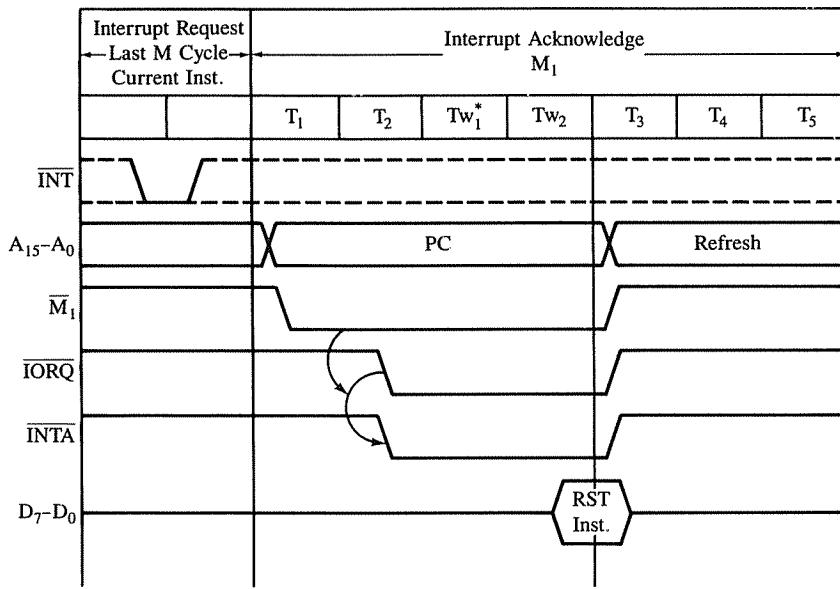
The Z80 interrupt process can be described in terms of these eight steps.

- Step 1:** The interrupt process should be enabled by writing the instruction EI, and the interrupt Mode 0 should be specified by the instruction IM 0 in the main program. This is similar to keeping the phone receiver on the hook.
- Step 2:** When the microprocessor is executing a program, it checks the INT line in the last T-state of each instruction.

- Step 3:** If the interrupt flip-flop is enabled and the INT signal goes active (low) and remains active until the end of the instruction being executed, the microprocessor samples the INT signal, completes the instruction, disables the interrupt flip-flop and sends a signal called INTA—Interrupt Acknowledge (active low). The processor cannot accept any further interrupt requests until the interrupt flip-flops are enabled again.
- Step 4:** The signal INTA is used to insert an instruction, preferably a restart (RST) instruction, through additional hardware, as shown in Figure 12.3.
- Step 5:** If the instruction is one of the RST instructions, the microprocessor saves the memory address of the next instruction on the stack and transfers the program to the memory location of the RST instruction. This is similar to the secretary's making a mark on the draft before walking to the engineer to relay the message.
- 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 mark made on the typing draft after the secretary was interrupted by the phone call and continuing to type.

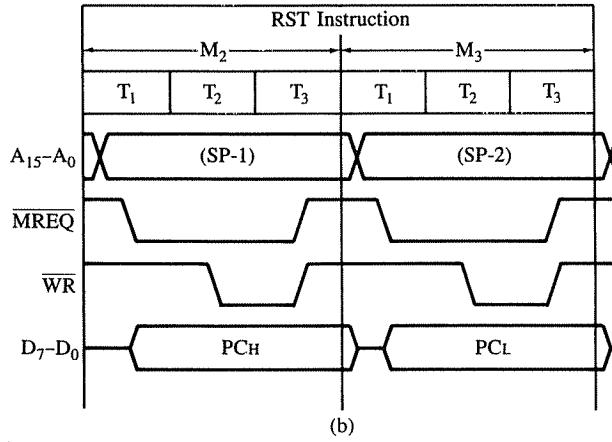
12.14 Interrupt Request/Acknowledge Machine Cycles

In Figure 12.3, the instruction RST 30H is built using resistors and a tri-state buffer. Figure 12.4(a) shows the timing of the Z80 Interrupt Request and Acknowledge. The interrupt signal (INT) is sampled by the Z80 with the rising edge of the last clock at the end of every instruction. If the Interrupt Enable flip-flop is already enabled by the EI instruction and the INT is low, the Z80 acknowledges the interrupt by generating the IORQ signal during the M₁ (Opcode Fetch) cycle. Normally, the MREQ control signal goes low during the M₁ cycle to read an opcode from memory. Thus, the interrupt is recognized when M₁ and IORQ are active (Figure 12.4(a)). By logically ANDing these two signals in a negative NAND gate (De Morgan's equivalent of an OR gate), we can generate the Interrupt Acknowledge (INTA) control signal. The INTA signal can be used to place an 8-bit instruction (such as RST) onto the data bus. Figure 12.4(a) shows that the Z80 adds two wait states during the interrupt M₁ cycle; these wait states allow sufficient time to determine priorities in multiple interrupts (this will be discussed later). Once the Z80 recognizes that the instruction received is a RST (Call) instruction, it issues two more machine cycles M₂ and M₃ to store the program counter on the stack. During M₁, 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 M₂ (Figure 12.4(b)), the address of the stack pointer minus one (SP – 1) location is placed onto the address bus, and the high-order address of the program counter is stored on the stack. During M₃, the



*T_{w1} and T_{w2} are Wait States, Automatically Inserted by the Z80

(a)



(b)

FIGURE 12.4

(a) Interrupt Request/Acknowledge and RST Fetch Machines Cycles (b) RST Instruction:
M₂, M₃ Machine Cycles

SOURCE: Courtesy of Zilog, Inc.

low-order address of the program counter is stored in the next location (SP-2) of the stack. Figure 12.4(b) shows that the machine cycles M_2 and M_3 are Memory Write cycles.

In the next instruction cycle, the program is transferred to location 0030_H , assuming we use the circuit shown in Figure 12.3 to insert the RST instruction. However, there is a space of eight memory locations between any two RST instructions; RST $38H$, the next restart instruction, begins at 0038_H . If the service routine requires more than eight locations, the routine is written somewhere in memory, and the jump instruction is written at 0030_H to specify the address of the service routine. All these steps are illustrated in the next section.

12.2

ILLUSTRATION: AN IMPLEMENTATION OF THE Z80 INTERRUPT IN MODE 0

The following example is concerned primarily with demonstrating the basic concepts in the interrupt I/O, rather than illustrating an industrial application. Hardware circuitry is kept to a minimum so that it can be easily built and tested in a laboratory. Similarly, programs are chosen for the ease of the implementation of the interrupt.

In this example, the Z80 MPU is kept busy counting in binary, and when it is interrupted, it flashes FF_H at one of the output ports. The program is illustrated with specific memory locations starting at 2000_H to explain the details of the interrupt process.

12.21 Problem Statement

The main program counts continuously in binary with a one-second delay between each count and displays the count at PORT0. A service routine is written at 2070_H to flash FF_H five times when the program is interrupted, with some appropriate delay between each flash.

1. Design a circuit to insert the instruction RST $30H$ when the MPU is interrupted by a push-button key.
2. Explain the interrupt process.

PROGRAM

Memory Address	Mnemonics	Comments
PORT0	EQU $07H$;Output port address
STACK	EQU $2099H$;Initial stack address
2000	LD SP, STACK	;Initialize stack pointer
2003	IM 0	;Set up interrupt in Mode 0
2004	EI	;Enable interrupt flip-flops
2005	LD A, $00H$;Start counter
2007	NXTCNT: OUT (PORT0), A	;Display count

2009	CALL DELAY	;Wait one second
200C	INC A	;Next count
200D	JP NXTCNT	;Continue

DELAY: ;Use delay subroutine illustrated in Chapter 10 (Section 10.7).

FLASH: ;Service routine to display FFH five times.

2070	PUSH BC	;Save register contents
2071	PUSH AF	
2072	LD B, 0AH	;Load register B for five flashes ;and five blanks
2074	LD C, 01H	;Parameter for one second delay
2076	LD A, 00H	;Load 00 to blank display
2078	DISPLAY: OUT (PORT0), A	;Output 00 and FFH alternately
207A	CALL DELAY	;Wait one second
207D	CPL	;Complement display byte
207E	DEC B	;Reduce count
207F	JP NZ, DISPLAY	
2082	POP AF	;Restore register contents
2083	POP BC	
2084	EI	;Enable interrupt process
2085	RET	;Service is complete, go back to main ;program

12.22 Circuit Design

The circuit is concerned with designing the instruction RST 30H. The machine code for the instruction is $F7_H$. We can design such an instruction by using the 74LS244, a tri-state buffer, as shown in Figure 12.5. All the input lines are tied high to represent logic 1 except line DI_3 , which is grounded to insert logic 0. The output lines of the buffer are connected to the data bus of the Z80 MPU. When the Enable signal of the buffer goes active (low), the input of the buffer, $1\ 1\ 1\ 1\ 0\ 1\ 1\ 1$ ($F7_H$), is placed onto the data bus.

The Interrupt Acknowledge (INTA) signal from the Z80 is generated by ANDing $\overline{M_1}$ and \overline{IORQ} in a negative NAND gate, and it is connected to the Enable line of the buffer. The INT line of the Z80 is pulled high through a 10 k resistor, and an interrupt is asserted by grounding the \overline{INT} through the push button key as shown in Figure 12.5.

12.23 Interrupt Operation

1. The main program initializes the stack pointer at 2099_H and enables the interrupts. The program will count continuously from 00_H to FF_H with a delay of one second between each count.
2. When the key is pushed to interrupt the processor, the \overline{INT} line goes low.
3. Assuming the key is pushed when the processor is executing the instruction OUT at memory location $2007H$, the following sequence of events occurs.

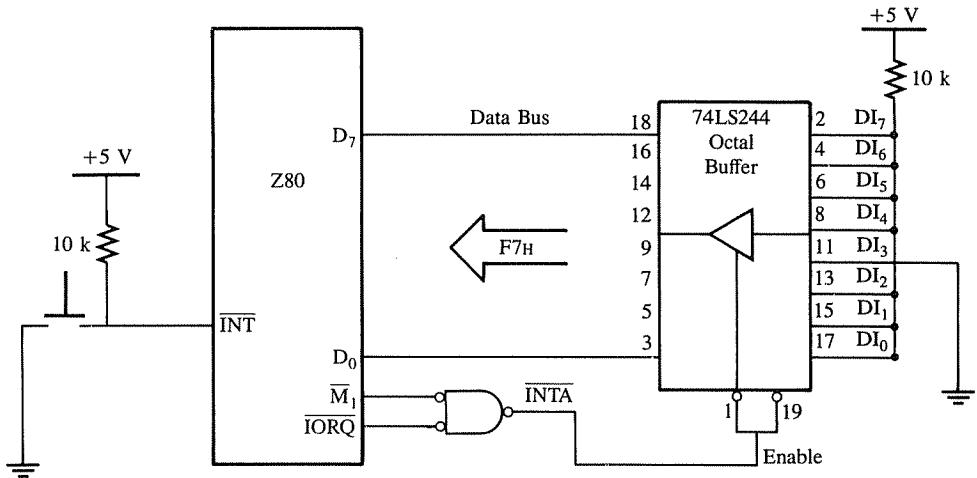


FIGURE 12.5
Schematic to Implement Mode 0 Interrupt

The Z80

- samples the INT line in the last T-state of the OUT instruction.
 - senses that the INT is low, and the interrupt is enabled.
 - completes the execution of the instruction OUT.
 - disables the interrupt, and sends out two signals: M₁ and IORQ.
4. The INTA (Interrupt Acknowledge) signal goes active at the output of the negative NAND gate.
 5. The INTA enables the buffer, and the code $F7_H$ is placed onto the data bus.
 6. The Z80 recognizes the instruction as one of the RST instructions. It saves the address 2009_H of the next instruction (CALL DELAY) on the stack at locations 2098_H and 2097_H .
 7. The program is transferred to memory location 0030_H . The locations 0030 , 0031 , and 0032_H should have the following Jump instruction to transfer the program to the service routine: JP $2070H$.
(Let us assume that this Jump instruction is already written at 0030_H by the system designer. Otherwise, you do not have access to write at 0030_H . Generally, in a system, ROM (or EPROM) is mapped into the initial memory locations. See the next section.)
 8. The program jumps to the service routine at 2070_H .
 9. The service routine saves the registers being used in the subroutine and loads the count ten into register B to output five flashes and also five blanks.
 10. The service routine enables the interrupt before returning to the main program.

11. When the service routine executes the RET instruction, the microprocessor retrieves the memory address 2009_H from the top of the stack and continues the binary counting in the main program.

12.24 Testing Interrupts on a Single-Board Microcomputer System

Step 7 in the preceding description assumes that you are designing the system and have access to locations in EPROM or ROM on page 00_H . 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. By writing one more Jump instruction at this location in R/W memory, we can transfer the program to location 2070_H .

12.25 Issues in Implementing Interrupts

In the above illustration, we deliberately avoided some of the complex issues in the interrupt I/O to maintain clarity in the discussion. These issues are discussed here.

1. Is there a minimum pulse width required for the \overline{INT} signal?

The interrupt in Mode 0 is level sensitive, meaning the pulse has to be active until the Z80 has time to sample it. The Z80 samples the \overline{INT} signal on the rising edge of the last clock cycle of every instruction, and the longest instruction in the Z80 set is 23 T-states. In the worst case, if the \overline{INT} goes active in the last cycle of an instruction, it may have to stay on for 23 clock periods.

2. How long can the \overline{INT} pulse stay low?

The \overline{INT} pulse can remain low until the interrupt flip-flops are set by the EI instruction in the service routine. If it remains low after the execution of the EI instruction, the processor will be interrupted again, as if it were a new interrupt.

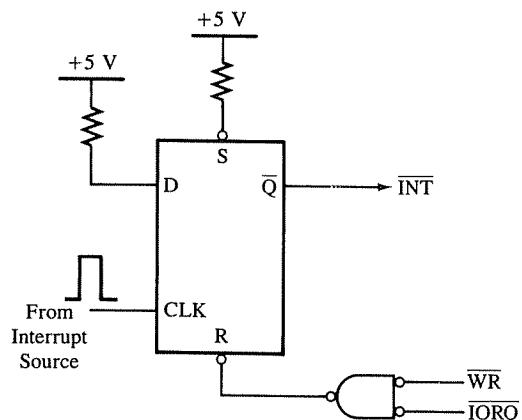
3. How can we keep the pulse long enough to interrupt the processor but not so long that it can be misinterpreted as a new interrupt?

One of the solutions to this dilemma is to latch the \overline{INT} pulse in a flip-flop and clear the flip-flop before enabling the interrupt again. This can be accomplished by interfacing the flip-flop as an output port and sending an OUT instruction to clear the flip-flop, as shown in Figure 12.6. In this case, the port address is irrelevant because any OUT instruction will reset the flip-flop.

4. 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 our illustration, the service routine enables the interrupt at the end of the service routine; in this case, the microprocessor

FIGURE 12.6
Latching and Clearing Interrupt Request



cannot be interrupted before the completion of this routine. If the instruction EI were written at the beginning of the routine, the microprocessor could be interrupted again during the service routine.

5. Is there any problem in connecting the key to interrupt the processor as shown in Figure 12.5?

Yes. When a mechanical push-button key is pressed or released, the metal contacts of the key momentarily bounce before giving a steady-state reading, as shown in Figure 12.7(a). The bounce can last for more than 20 ms, and if the interrupt service routine clears the flip-flop before the bounce is settled, the key bounce will be interpreted as a new interrupt.

The key bounce can be eliminated from the $\overline{\text{INT}}$ signal by connecting the key through a pair of NAND gates, as shown in Figure 12.7(b). Initially, the output of gate G_1 is logic 1 and that of gate G_2 is logic 0. When the key is pushed, and when it loses its contact with terminal A, the input A_1 to the gate G_1 goes high, but the input A_2 is still low. Thus, the output does not change. When the key makes the contact with terminal B, the input B_1 to gate G_2 goes low and the output of G_2 changes from logic 0 to logic 1. This changes the input A_2 from logic 0 to logic 1. Thus, the output of G_1 changes to logic 0.

The key bounce is eliminated because when the key bounces, meaning it bounces from no-contact to contact with the same terminal, the output will not change. In our illustration, the problems of the key bounce and the duration of the $\overline{\text{INT}}$ pulse are avoided by keeping the service routine unusually long.

6. What is the reason to have two flip-flops IFF1 and IFF2 to enable the interrupt?

In Mode 0, it does not make any difference. In the nonmaskable interrupt, the status of IFF1 is copied into IFF2 when the Z80 is interrupted, and the status is copied back into IFF1 at the end of the service routine (see Section 12.5).

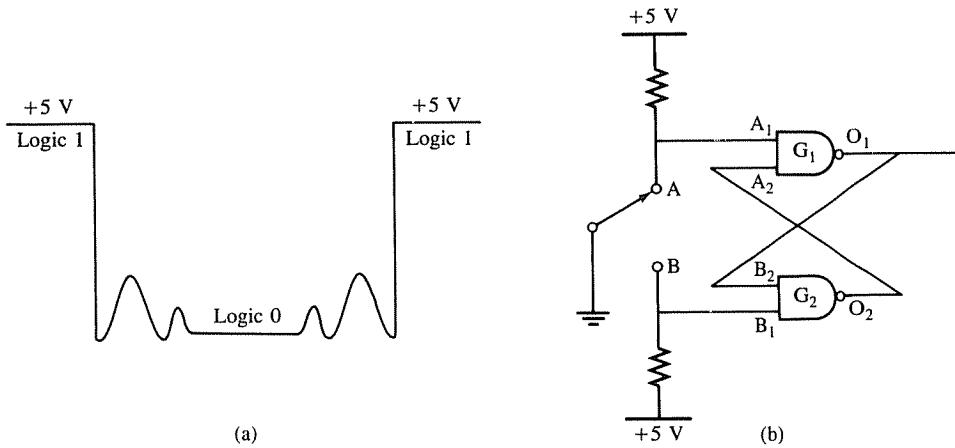


FIGURE 12.7
(a) Key Bounce (b) Key Debounce Using Two NAND Gates

7. *What is the difference between RET (Return) and RETI (Return from Interrupt) instructions?*

The RET instruction is used to return from a subroutine. The RETI is a specially designed instruction to reset the Z80 daisy-chain interrupt logic (see Section 13.35).

8. *When does EI in the service routine (memory location 2084_H) become effective?*

The EI instruction becomes effective immediately after the execution of the next instruction. In the service routine, the processor will not be prematurely interrupted until after the execution of the RET instruction.

9. *If the Interrupt Enable flip-flops are set and the \overline{INT} is low long enough, will an interrupt request always be acknowledged?*

The interrupt request will not be acknowledged if the \overline{BUSRQ} (Bus Request) is active or if the Z80 is servicing a higher priority request (see Section 12.6).

ILLUSTRATION: INTERFACING A/D CONVERTER IN INTERRUPT MODE 1

12.3

The interrupt Mode 1 is a special case of the interrupt Mode 0; all the basic concepts of the interrupt I/O discussed in the previous section are applicable in this mode. However, the implementation is simple because most of the external circuitry required for Mode 0 is already built into the Z80. After reviewing the interrupt I/O in Mode 1, we will discuss briefly the basic concepts in the A/D conversion and illustrate the interrupt Mode 1 by interfacing the ADC0801, an A/D converter on a chip manufactured by National Semi-

conductor. This A/D converter is specially designed to be compatible with the microprocessor control signals.

12.31 Interrupt Process in Mode 1

The initial requirements of Mode 1 to interrupt the Z80 processor are similar to those of Mode 0:

- The Interrupt Enable flip-flops should be set by the instruction EI.
- The mode should be specified.
- The INT signal should go low and stay low until the Z80 can sample it.

After the Z80 acknowledges the interrupt, it

- resets the Interrupt Enable flip-flops, thus disabling further interrupts.
- places the address of the next instruction (the contents of the program counter) on the stack.
- transfers the program execution to location 0038_H (without any external hardware).

The primary difference between Mode 0 and Mode 1 is that in Mode 1, the external hardware to insert the RST instruction is not necessary; it is already built into the processor, and it is activated by setting the Mode. The other difference is that the program can be transferred to only one location, unlike the eight RST instructions in Mode 0; Mode 1 uses the memory location of RST 38H.

12.32 Basic Concepts in A/D Conversion

The analog-to-digital (A/D) conversion is a process whereby an analog signal is represented by equivalent binary states. The relationship between the binary representation and the analog signal is dependent on the design of a converter. For example, we can determine that a 0–1 V signal can be represented by binary three digits; 1 V will be equivalent to all 1s and 0 will be represented by all three 0s. We can divide the entire range 0–1 V into eight different combinations of three bits and each combination can represent one level (1/8 V), called resolution, as shown in Figure 12.8. When the input analog signal is 0.5 V, the binary output will be 1 0 0, as shown in Figure 12.8(b).

One of the common design techniques used to convert an analog signal into its binary equivalent is called successive approximation.* To begin the process, a clock pulse, called the START pulse, is required. An A/D converter can take a few hundred microseconds to milliseconds to convert the analog signal into its binary equivalent; the time it takes to complete the conversion is called the conversion time. At the end of the conversion, the A/D converter generates a pulse called END OF CONVERSION or DATA READY.

To interface such a converter with the microprocessor, the following conditions need to be satisfied. The Z80 should

*You need not be familiar with data converter design technique to interface a data converter with the microprocessor.

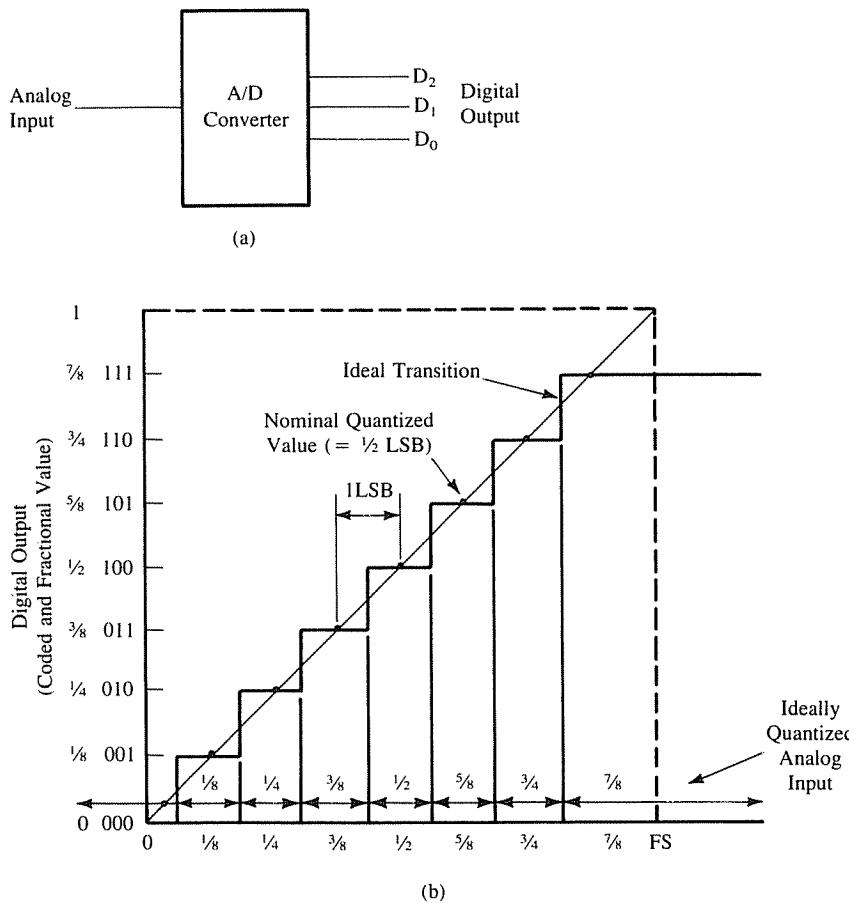


FIGURE 12.8
A 3-bit A/D Converter: (a) Block Diagram (b) Analog-to-Digital Conversion

- Provide a START pulse to initialize the conversion process by writing to the device as an output port.
- Wait until the end of the conversion.
- Read the binary equivalent when the DATA READY signal goes active.

We need to interpret these conditions in terms of circuitry and control signals. The microprocessor can communicate with any external device through a port address and its Read and Write control signals. To meet the above requirements we need to build

- one output port to send a START pulse,
- one input port with a latch so that the Z80 can read the binary data,
- a circuit to sense the end of the conversion.

Fortunately, manufacturers have begun to include latches, buffers, and control logic on the same chip with data converters so that data converters can be easily interfaced with the microprocessor. For our illustration, we selected the National Semiconductor ADC0801 because it has all the necessary interfacing circuitry built in.

THE ADC0801

This is an 8-bit A/D converter (Figure 12.9(a)) available as an integrated circuit on a chip. The analog signal is connected to $V_{in(+)}$, and the binary output is available on eight data lines DB_7-DB_0 . The maximum input signal can be +5 V or it can be connected as a differential input by using the pin $V_{in(-)}$. The signal V_{ref} is used to limit the maximum input voltage; if it is not connected externally, it is set for +5 V internally. The internal clock is determined by the RC network connected to pins 10 and 4.

To interface the A/D converter with the microprocessor requires three signals: \overline{CS} , \overline{WR} , and \overline{RD} . To start the conversion, the chip should be selected and \overline{WR} asserted low. When \overline{WR} goes low, the chip is reset, and when it goes high, the conversion begins. At the end of the conversion, it initiates the signal \overline{INTR} ; this can be used to interrupt the microprocessor. When the microprocessor reads the output, the \overline{INTR} is reset (see the timing waveforms in Figure 12.9(b)). The ADC0801 is ideally suited for interfacing as an interrupt I/O not only because it generates the \overline{INTR} pulse, but also because it is turned off after the data byte is read. This eliminates our concern about the \overline{INT} pulse width for the Z80 microprocessor.

12.33 Interfacing Circuit

To interface the ADC0801, three signals are necessary: \overline{WR} , \overline{RD} , and \overline{CS} . Figure 12.9(a) shows such a circuit. The output line O_0 of the 3-to-8 decoder is connected to the \overline{CS} signal of the converter. The converter is selected when the address on the address lines from A_7-A_0 is $F8_H$; thus, the converter is assigned the port address $F8_H$. To start the conversion, the Z80 should write to port $F8_H$; however, we are interested not in writing anything, but in asserting the \overline{WR} signal. At the end of the conversion, the converter asserts the \overline{INTR} , which is connected to the \overline{INT} signal of the Z80.

Assuming the Z80 interrupt is enabled and is set for Mode 1, the program will be transferred to memory location 0038_H . If the system is being designed, the service routine to read data can be written at 0038_H . Otherwise, a Jump instruction should have already been written here to give access to the service routine in system's R/W memory. When the service routine reads the data byte, the \overline{RD} signal will remove the interrupt.

12.34 Program

The following program is set up to collect a number of readings from the data converter, and the readings are stored in memory labelled as BUFFER. The number of readings to be recorded is defined by the term BYTE. To verify this illustration in a laboratory, the terms STACK, BUFFER, and BYTE need to be defined. The program has three segments: main program to initialize the parameters, the restart segment, and the service routine to record data.

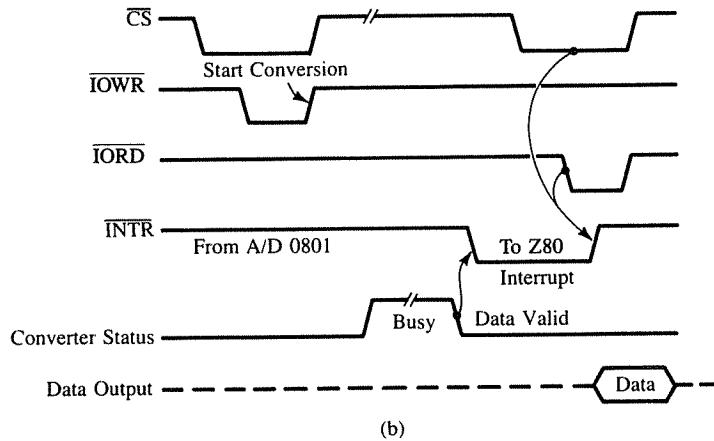
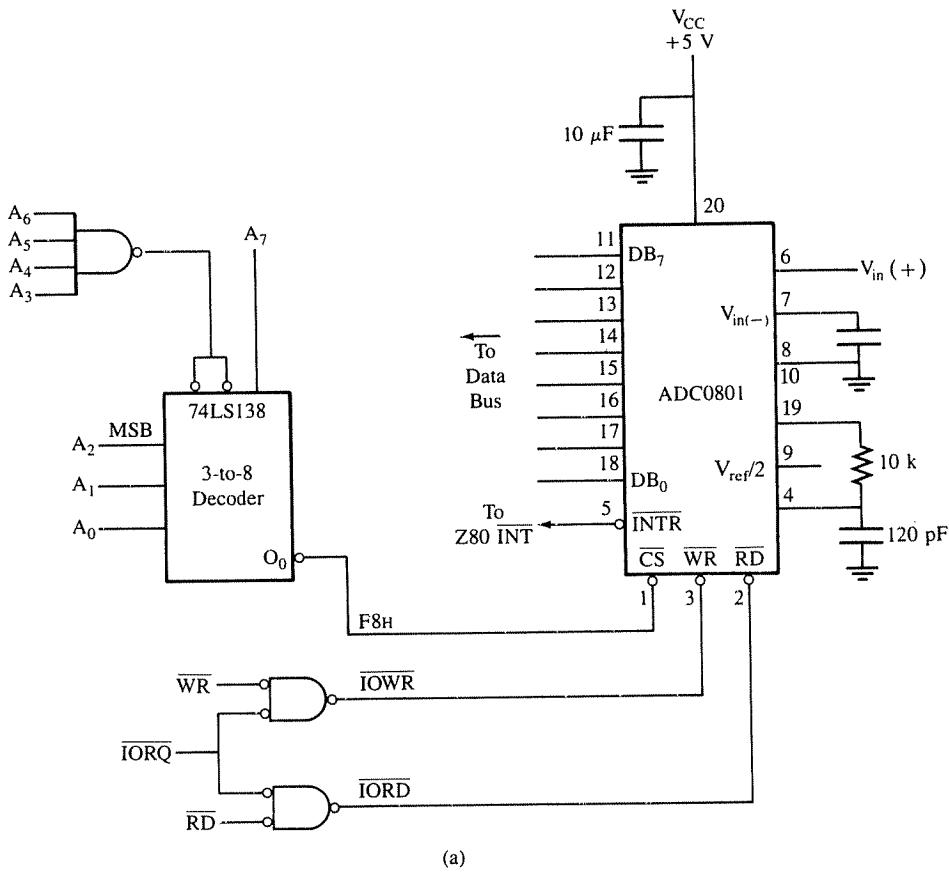


FIGURE 12.9

(a) Interfacing A/D Data Converter ADC0801 (b) Timing Diagram for Reading Data from A/D Converter

SOURCE: (b) Reprinted with permission of National Semiconductor Corporation.

MAIN PROGRAM

START:	LD SP, STACK	;Initialize stack pointer
	IM 1	;Set up interrupt Mode 1
	LD HL, BUFFER	;Set up HL as memory pointer
	LD B, BYTE	;Set up counter to count the number of readings
	EI	;Enable interrupt flip-flops
	OUT (F8H), A	;Start conversion
WAIT:	NOP	
	JP NZ, WAIT	;If all data readings are not yet recorded, wait
	HALT	
0038	JP ADC	;This is Mode 1 restart location; go to data ; converter service routine

SERVICE ROUTINE

ADC: ;This service routine reads data from the A/D converter, saves data in memory, and starts conversion for the next reading.
;Input: Address of memory pointer in HL and the number of readings to be recorded in register B.
;Modifies registers A, B, and HL

IN A, (F8H)	;Read data byte from the converter
EI	;Enable the interrupt
LD (HL), A	;Save data in memory
INC HL	;Next memory location
DEC B	;One reading is recorded, decrement counter
OUT (F8H), A	;Start conversion for the next reading
RET	

PROGRAM DESCRIPTION

The main program initializes the stack pointer, the memory pointer, and the counter. It enables the interrupt, starts the conversion by writing to port $F8_H$, and waits in the loop. In a real industrial application, the main program would have continued to monitor other activities.

When the conversion is complete, the data converter causes \overline{INTR} to go low, which interrupts the processor. Because the interrupt is set up in Mode 1, the Z80 disables the interrupt, stores the contents of the program counter on the stack, and transfers the program to 0038_H automatically. The Jump instruction at 0038_H transfers the program to the service routine.

The service routine first reads the output of the data converter, and the Read signal removes the interrupt from the INT pin. This logic is built inside the ADC0801; in the previous illustration, we needed to use a flip-flop to turn off the interrupt (see Figure 12.6). The routine enables the interrupt so that the subsequent interrupts can be accepted, then

upgrades the memory pointer, decrements the counter, and initiates the conversion for the next reading. The final instruction in the service routine (RET) is critical; when the RET instruction is executed, the Z80 gets the address from the top of the stack and returns to the main program. The main program has the instruction Jump On No Zero; the Zero flag is set in the service routine when the register B goes to zero.

INTERRUPT MODE 2

12.4

In Mode 0 and Mode 1, once the interrupt request is acknowledged, the program is transferred to specific locations on memory page 00. This is quite a limitation because these locations are, generally, reserved for ROM (or EPROM) and can be used for only eight interrupts. On the other hand, in Mode 2, the program control can be transferred to any memory location in the memory map. This is one of the powerful features of the Z80 microprocessor.

The process of interrupt request and acknowledge in Mode 2 is similar to that of Mode 0. However, the response to the interrupt request is quite different.

12.41 Interrupt Process in Mode 2

Assuming that the interrupt is enabled and set up to operate in Mode 2 by the instructions EI and IM 2, and that the INT signal goes low, the Z80 acknowledges the interrupt by asserting two signals M₁ and IORQ. The INTA signal, generated by ANDing M₁ with IORQ, is used to place a byte onto the data bus. The Z80 disables the interrupt and places the contents of the program counter onto the stack. This is similar to Mode 0.

However, in Mode 2, the Z80 interprets the eight bits from external hardware as the low-order byte of a 16-bit memory address rather than an instruction as in Mode 0. The Z80 takes the eight bits from the interrupt register (I) as the high-order byte and combines it with the external byte to form a 16-bit vector address or a pointer. The program is then transferred to the memory location pointed to by the 16-bit address. Let us assume it is $20F8_H$. Then the contents of the two memory locations $20F8_H$ and $20F9_H$ are interpreted as the 16-bit memory address of the service routine; the byte in $20F8_H$ is the low-order byte and the byte in $20F9_H$ the high-order byte. This process is demonstrated in the next example.

1. Write initialization instructions to set up the Z80 interrupt in Mode 2 and the interrupt vector with 20_H as the high-order byte.
2. Design a circuit to place the byte $F8_H$ onto the data bus using the INTA signal.
3. Specify the contents of the vectored memory locations if the service routine is located at $23A7_H$.

Example
12.2

Solution**1. Initialization Instructions**

MODE2: LD SP, STACK	;Initialize stack pointer
LD A, 20H	;Load high-order byte of interrupt vector
LD I, A	;Load register I with high-order byte
IM 2	;Set up Z80 interrupt in Mode 2
EI	;Enable interrupt flip-flops

2. Circuit for the Byte $F8_H$

Figure 12.10 shows the circuit to place the byte $F8_H$ onto the data bus. The input lines DI_0 – DI_2 of the tri-state buffer are grounded and the remaining lines are tied high. When the buffer is enabled by the interrupt acknowledge (\overline{INTA}) signal, the byte $F8_H$ is placed onto the data bus.

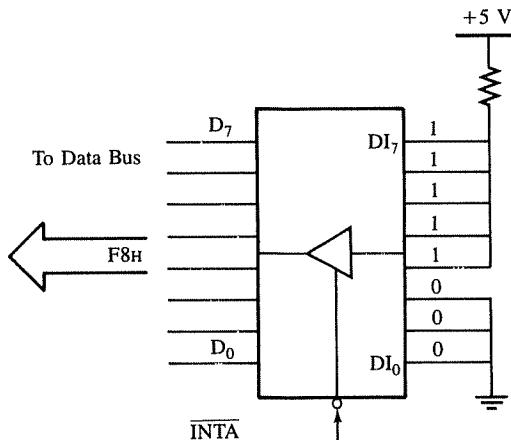
3. Memory Vector and Service Routine Addresses

When the Z80 acknowledges an interrupt request, it forms the memory vector by combining the contents of the interrupt register I (20_H) and the data byte ($F8_H$); thus, the address of the vector becomes $20F8_H$. To transfer the program control to the service routine located at $23A7_H$, the low-order byte $A7_H$ must be stored at location $20F8_H$ and the high-order byte 23_H at location $20F9_H$.

$$\begin{array}{rcl} 20F8 & \rightarrow & A7_H \\ 20F9 & \rightarrow & 23_H \end{array}$$

At this point, it is necessary to clarify the potential confusion in implementing Mode 2 interrupt. The 16-bit address formed by combining the byte in interrupt register I and the byte placed on the data bus is the vector address (or the memory pointer) and not the address of the service routine. The address of the service routine is located at memory

FIGURE 12.10
Schematic to Place the Byte $F8_H$
onto the Data Bus



locations pointed to by the vector. Thus, on one page of memory (256 bytes), a table of 128 vectors pointing to service routines can be stored. Another puzzling question is how to design multiple interrupts using only one $\overline{\text{INT}}$ signal; this question is discussed in Section 12.6.

NONMASKABLE INTERRUPT

12.5

The Z80 has a separate input (pin 17) for the **nonmaskable interrupt**. As mentioned before, this interrupt cannot be disabled and has the highest priority among the interrupts. When the Z80 acknowledges the $\overline{\text{NMI}}$, it transfers the program to memory location 0066_H . The $\overline{\text{NMI}}$ is used, generally, for emergency situations such as power failure or activities with high priority, such as a system clock.

12.5.1 Nonmaskable Interrupt Process

The nonmaskable interrupt differs from the maskable interrupts in the following ways.

- It cannot be disabled by the DI instruction and need not be enabled by the EI instruction; it is independent of the EI and DI instructions.
- It is edge sensitive, meaning it does not have to be active until the Z80 samples it.
- It has a higher priority than the maskable interrupts, meaning it will always be acknowledged at the end of the current instruction being executed if BUSRQ (Bus Request) is inactive.

The steps in the nonmaskable interrupt are as follows.

1. When the $\overline{\text{NMI}}$ is caused to go low by an external device, the interrupt request is latched internally on the falling edge of the $\overline{\text{NMI}}$.
2. When the Z80 samples the $\overline{\text{NMI}}$ (as well as $\overline{\text{INT}}$) in the last T-state of the instruction being executed, it accepts the $\overline{\text{NMI}}$ request after completing the current instruction if BUSRQ is inactive.
3. Once the $\overline{\text{NMI}}$ is accepted, the Z80 places the contents of the program counter on the stack.
4. The Z80 copies the status of the interrupt enable flip-flop IFF1, determined by the previously executed instructions EI (or DI), into IFF2 and resets IFF1 to prevent any interruptions from maskable interrupts.
5. The program is transferred to location 0066_H without any external hardware; this is similar to Mode 1 in the maskable interrupt.

If the service routine is terminated by the special instruction RETN (Return from Nonmaskable Interrupt), the Z80

6. copies IFF2 into IFF1 to restore the status of the maskable interrupt.
7. copies the contents of the top two locations of the stack into the program counter, and the program returns to the instruction where it was interrupted.

Example 12.3 Design a five-minute timer using a 60Hz power line as an interrupting source. The output ports should display minutes and seconds in BCD. At the end of the five-minute period, the display should be cleared, and the timer should continue into the next five-minute period.

Solution This is a five-minute timer designed with a 60Hz AC line. The circuit uses a step-down transformer, a wave-shaping network, and the nonmaskable interrupt (pin 17)—see Figure 12.11.

The AC line has a 60Hz frequency which interrupts the microprocessor every 1/60th of a second using the $\overline{\text{NMI}}$. After the interrupt, program control is transferred first to memory location 0066_{H} in the monitor program and then to the service routine TIMER. The $\overline{\text{NMI}}$ responds to a negative-edged trigger; thus, the interrupt need not be turned off even if the AC source has an eight millisecond pulse width. In this case, the service routine is completed in a few microseconds, but the next interrupt is recognized only when there is a pulse transition from positive to negative.

Monitor Program

0066	JP RWM	;This is $\overline{\text{NMI}}$ request, go to location in user ;memory
------	--------	-----------------------------------------------------------------------------

Main Program

LD SP, RAMTOP	;Initialize stack pointer
LD BC, 0000H	;Set up registers B for minutes and C for sec- ;onds
LD D, 60	;Set up register D to count 60 interrupts—in ;hand assembly the number 60 must be con- ;verted into a Hex number.

DSPLAY: LD A, B	;Display minutes at PORT 1
OUT (01H), A	
LD A, C	
OUT (02H), A	;Display seconds at PORT 2
JP DSPLAY	

RWM: JP TIMER	;This is $\overline{\text{NMI}}$, go to TIMER routine to upgrade ;the clock
---------------	---------------------------------------------------------------------------------

Interrupt Service Routine

TIMER: ; Section I	
DEC D	;One interrupt occurred, reduce count by one
JR NZ, END	;Is one second elapsed? If not, go back to main ;program

;Section II	
LD D, 60	;One second is complete, load register D again ;to count 60 interrupts

LD A, C	
ADD A, 01	;Increment “Second” register
DAA	;Decimal adjust “Seconds”
LD C, A	
CP 60H	;Save “Seconds”